# 持续集成

# 操作指南

# 产品文档

腾讯云

# 文档目录

# 操作指南
# 开通服务
# 主账号

最近更新时间：2023-01-05 18:13:25

## 操作场景

本文为您介绍腾讯云主账号如何开通并使用 CODING DevOps 服务，您可以按照以下几种情况有选择性操作。

## 主账号未开通服务

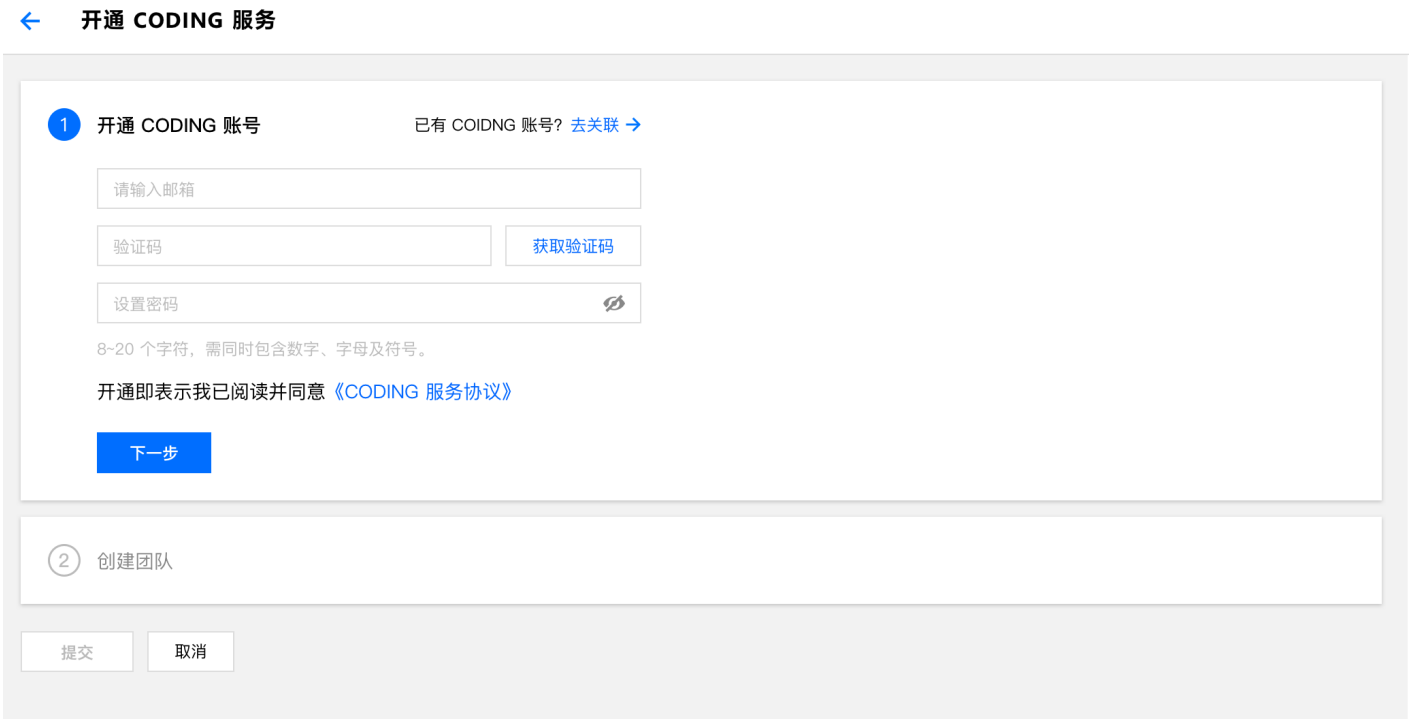
此场景适用于未曾在腾讯云中使用 CODING DevOps 服务的用户，请访问 CODING DevOps 控制台 并按照提示完成策略权限授权后，开通 CODING 服务。


> 说明：
> 腾讯云主账号服务开通后，您还需要 CODING 账号才能正常使用服务。

### 未注册过 CODING 账号

1. 若您未注册过 CODING 账号，请单击**没有 CODING 账号，去开通**。

2. 按照提示填写邮箱、验证码、密码进入下一步，输入团队名称和域名后完成团队创建。




### 关联已有 CODING 账号

若您有印象曾使用过 CODING 服务，但并非通过腾讯云控制台入口注册，那么可以选择将腾讯云账号与已有 CODING 账号相关联。请单击控制台中的**已有 CODING 账号，去关联**。

输入已注册的 CODING 账号完成关联操作，关联完成后将继承原有 CODING 团队数据。


### 主账号已开通服务

单击 控制台 中的**立即使用**或**团队域名**，跳转至 CODING 团队页后，开始使用 CODING 服务。

# 子用户

最近更新时间：2021-11-24 10:57:22

## 操作场景

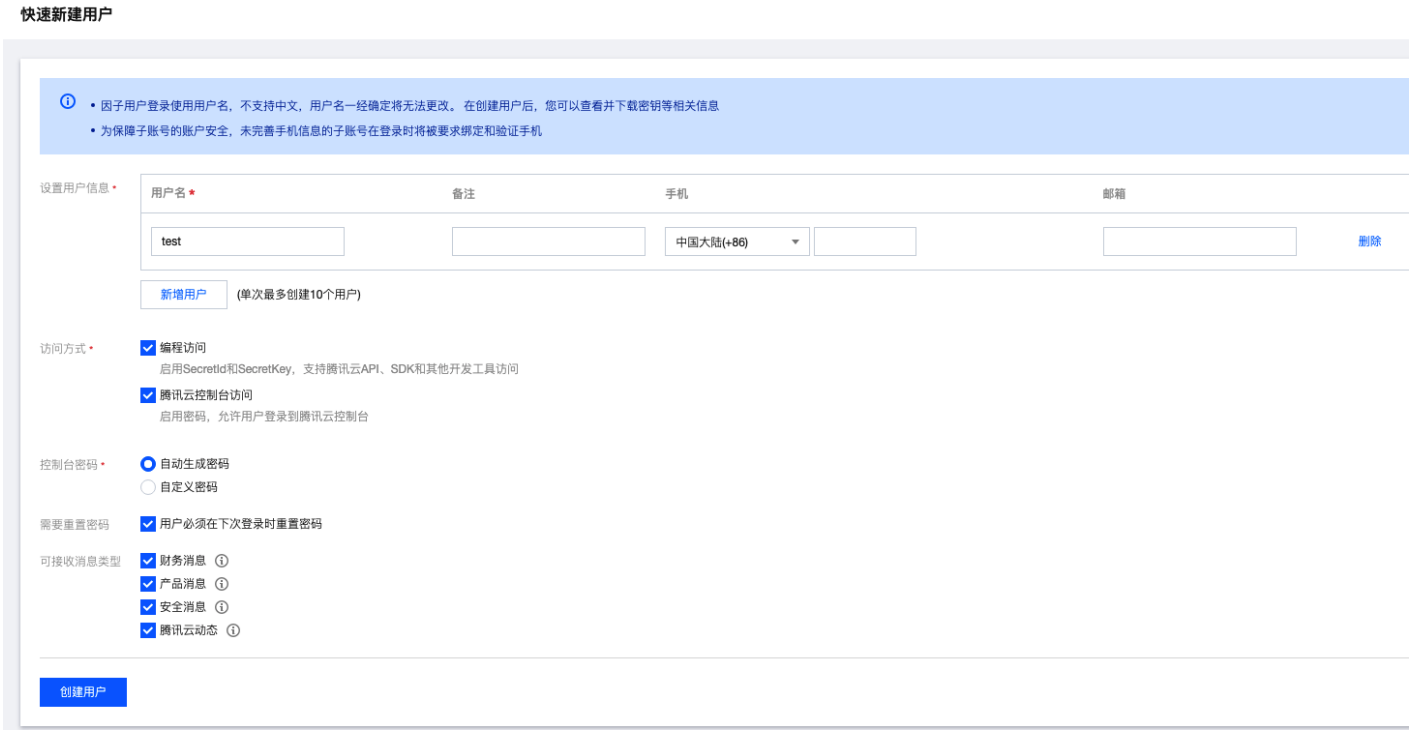
子用户是由主账号创建的实体，有确定的身份 ID 和身份凭证，能够登录并独立设置控制台，且具有 API 访问策略。若您希望通过腾讯云子用户使用 CODING DevOps 服务（以下简称服务），下文将按照多种场景进行指引。

1. 主账号已开通服务，需帮助子用户开通服务
2. 主账号未开通服务，需使用子用户协助主账号开通服务
3. 子用户已开通服务，但部分功能受限

## 主账号已开通服务

1. 主账号登录 CODING DevOps 控制台。
2. 单击**快速创建子用户**。

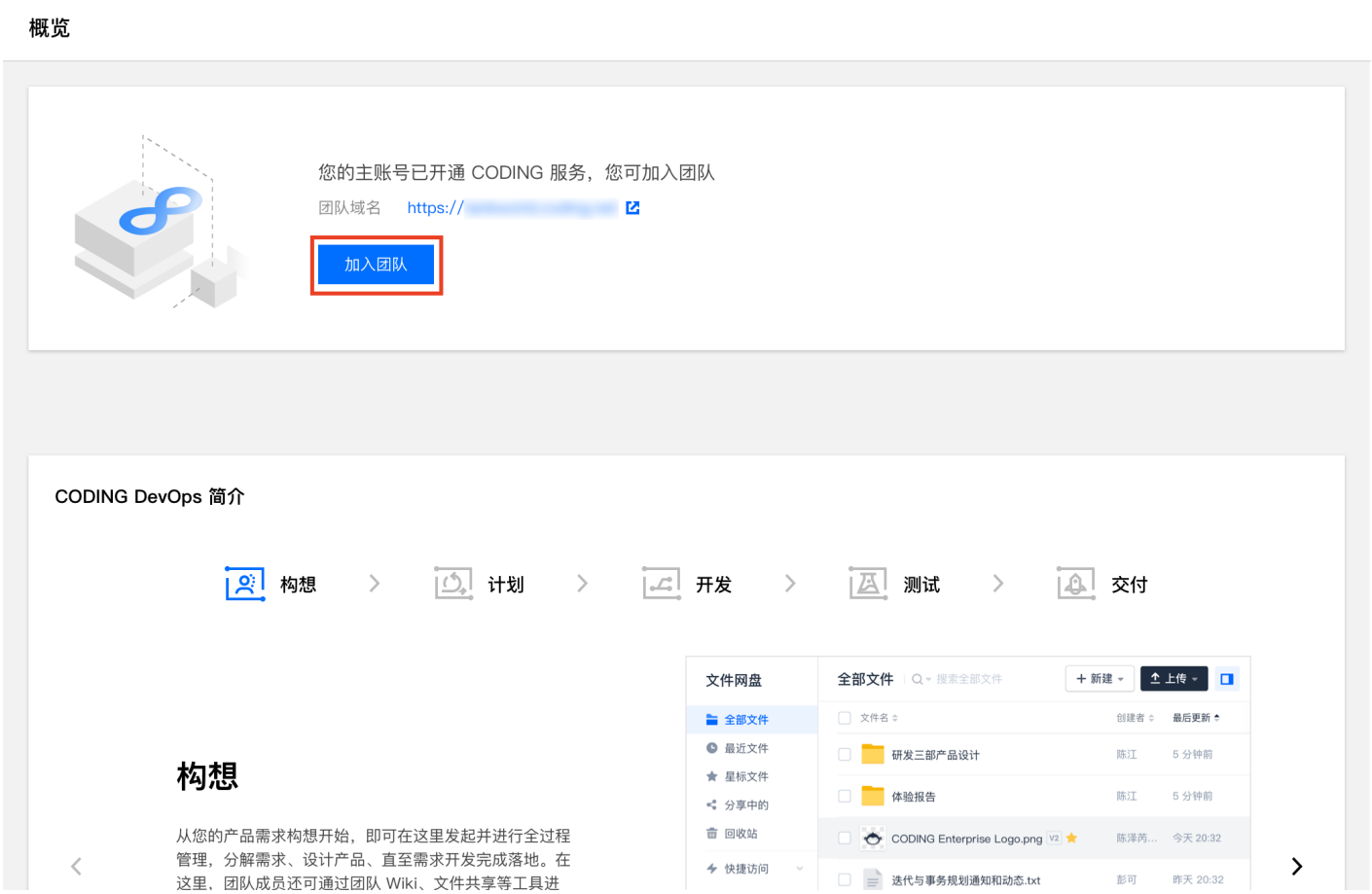

3. 在**快速新建用户**页面填写子用户相关信息，访问方式请同时勾选**编程访问**及**腾讯云控制台访问**，密码设置可选自动生成或自定义密码，单击**创建用户**后即可创建出拥有 CODING DevOps 策略的子用户。



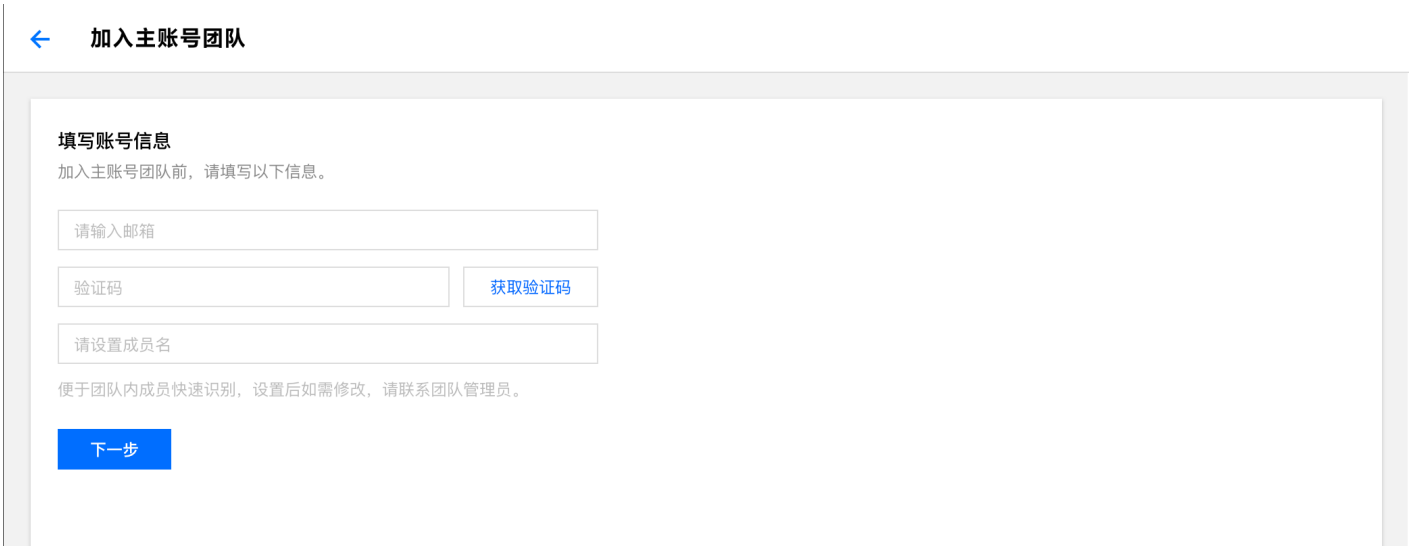

**注意：**

**新建子用户时，请务必勾选访问方式的编程访问和腾讯云控制台访问，避免子用户无法登录 CODING DevOps 控制台的情况。**

4. 创建完成后，使用子用户登录 控制台，单击**加入团队**。



按照提示填写账号信息后加入团队。




## 主账号未开通服务

### 使用主账号直接开通服务

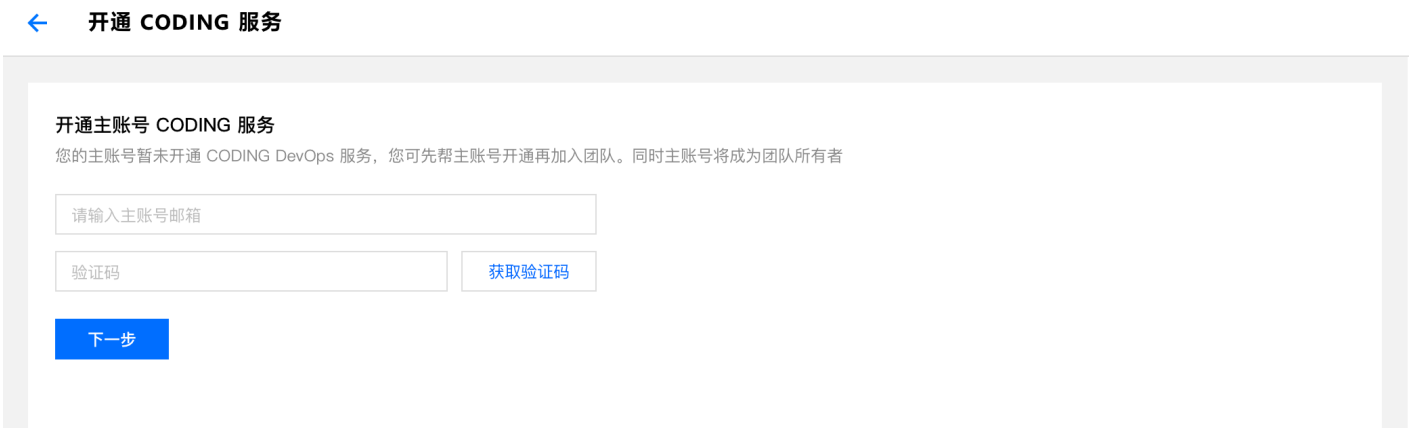
请联系主账号拥有者，参见 主账号，了解如何使用主账号直接开通服务。

### 使用子用户帮助主账号开通服务

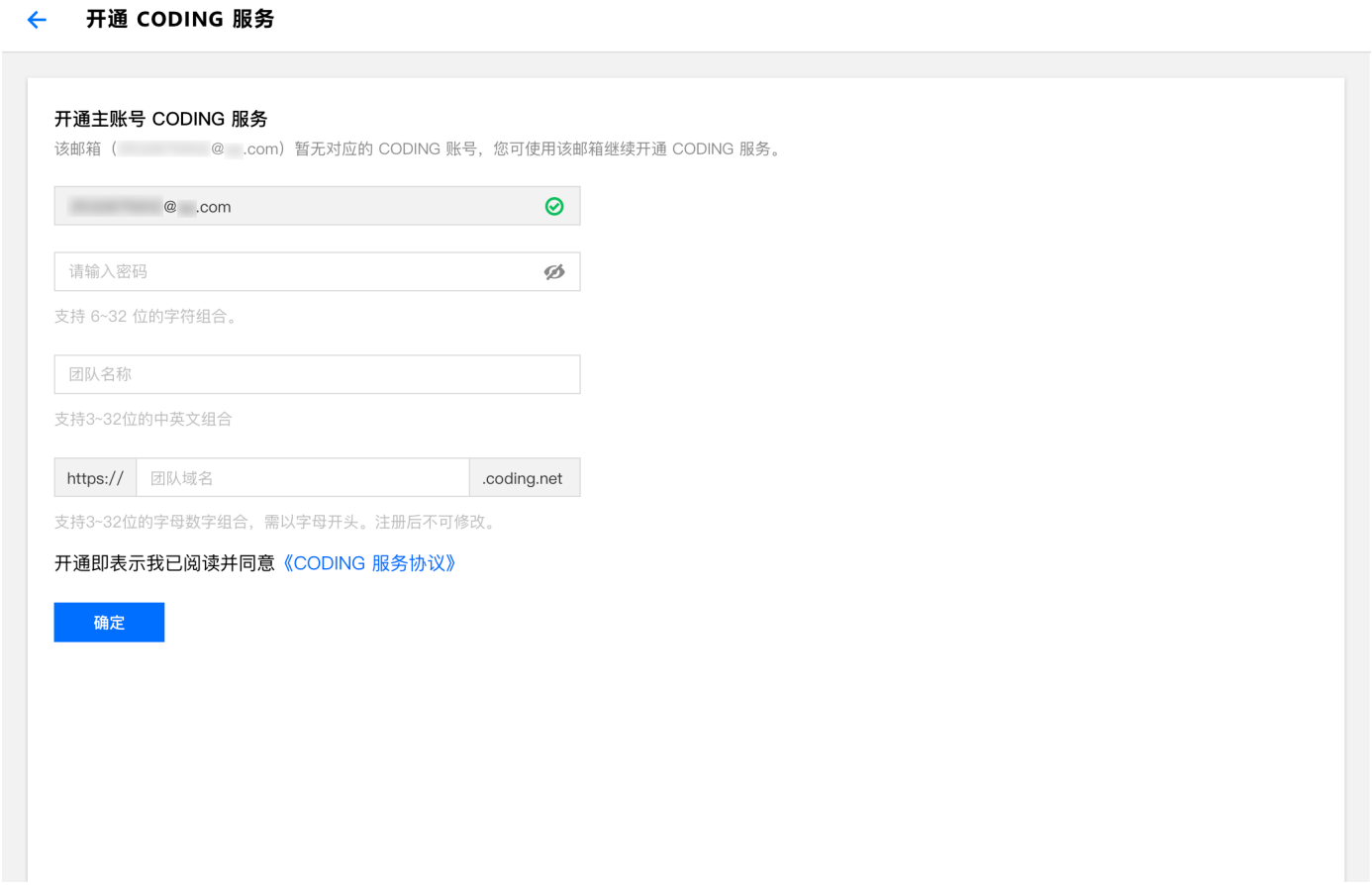
1. 子用户需前往 访问管理控制台，确认拥有 AdministratorAccess 权限策略，然后前往 CODING DevOps 控制台 > 概览页面。
2. 单击开通服务，进入主账号开通服务流程。

概览

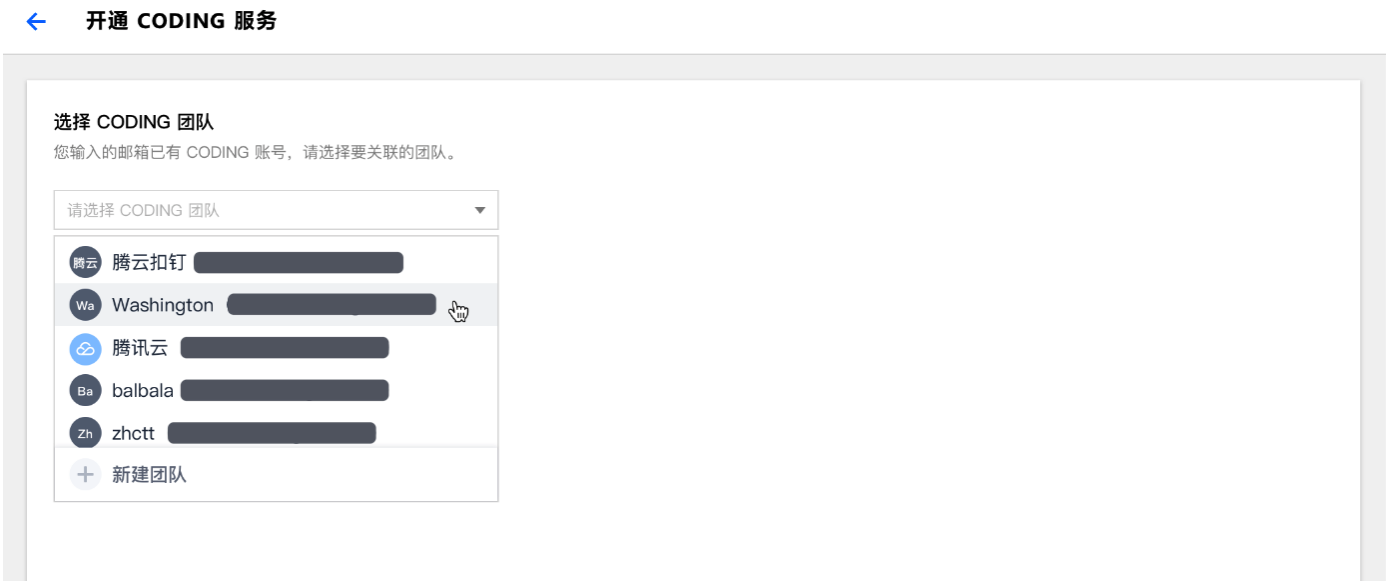
输入主账号邮箱与验证码完成开通。



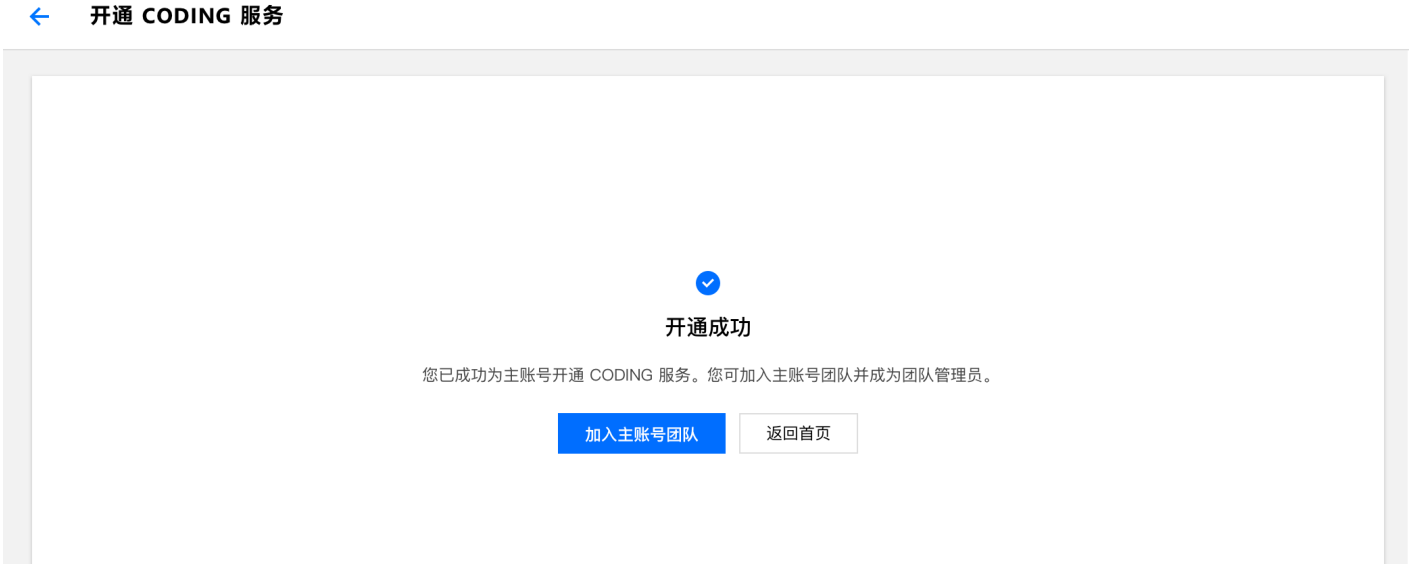

- 若主账号邮箱不存在，需进入主账号创建流程，输入密码与团队名称后开通账号。



- 若主账号邮箱已关联多个 CODING 团队，选择需要的团队即可。




3. 开通成功后，子用户即可加入主账号团队。




## 子用户已开通服务

子用户能否继续创建新的子用户，取决于 访问管理控制台 中所选择的策略。

当子用户具备 AdministratorAccess 策略时，控制台将出现**快速创建子用户**，意味着该子用户能够创建新的子用户。


若子用户仅具备 QcloudCODINGFULLAccess，控制台中将不再出现**快速创建子用户**，意味着该子用户仅具备服务的访问策略，无法继续新建子用户。

# 权限说明

最近更新时间：2021-08-25 15:28:21

在使用腾讯云 CODING Devops 产品的过程中，为了能够调取相关云资源，会遇到需要进行服务授权的场景。在使用该产品的过程中主要涉及 CODING_QCSRole 角色，本文档会展示此角色所包含的预设策略详情。

## CODING_QCSRole 角色

开通 CODING Devops 服务后，腾讯云会授予您的账户 CODING_QCSRole 角色的权限。该角色默认关联多个预设策略，对应策略会出现在该角色的已授权策略列表中。CODING_QCSRole 角色默认关联的预设策略包含如下：

- QcloudAccessForCODINGRole：CODING 服务对云资源的访问权限。
- QcloudAccessForCODINGRoleInAccessTKE：CODING 服务所需要的容器服务（TKE）权限。
- QcloudAccessForCODINGRoleInThroughTCR：CODING 服务所需要的容器镜像（TCR）权限。

### 预设策略 QcloudAccessForCODINGRole

#### 授权场景

当您已注册并登录腾讯云账号后，首次登录 CODING 控制台 时，需前往“访问管理”页面对当前账号赋予 CODING DevOps、访问管理（CAM）等权限。

#### 授权步骤

1. 首次登录 CODING 控制台 时会弹出**服务授权**窗口。
2. 单击**前往访问管理**，进入角色管理页面。
3. 单击**同意授权**，完成身份验证后即可成功授权。

#### 权限内容

- 访问管理相关

| 权限名称 | 权限说明 |
| --- | --- |
| cam:ListPolicies | 查询策略列表 |

- 容器镜像服务相关

| 权限名称 | 权限说明 |
| --- | --- |
| tcr:DescribeInstances | 查询实例信息 |
| tcr:CreateInstanceToken | 创建实例访问凭证 |
| tcr:DescribeNamespaces | 查询命名空间信息 |
| tcr:DescribeRepositories | 查询镜像仓库信息 |
| tcr:DescribeRepositoryOwnerPersonal | 查询个人版所有仓库 |

### 预设策略 QcloudAccessForCODINGRoleInAccessTKE

#### 授权场景

开通 CODING Devops 服务并完成 CODING_QCSRole 角色授权后，该策略将会与 CODING_QCSRole 角色相关联，完成操作后即可获得腾讯云容器服务（TKE）云资源的相关权限。

#### 授权步骤

该策略与预设策略 QcloudAccessForCODINGRole 同时授权，无需额外操作。

#### 权限内容

| 权限名称 | 权限说明 |
| --- | --- |
| ccs:DescribeCluster | 查询集群列表 |

| 权限名称 | 权限说明 |
| --- | --- |
| ccs:CreateClusterEndpoint | 创建集群访问端口 |
| ccs:CreateClusterEndpointVip | 创建托管集群外网访问端口 |
| ccs:ModifyClusterEndpointSP | 修改托管集群外网端口的安全策略 |
| ccs:DescribeClusterEndpointVipStatus | 查询托管集群开启外网端口流程状态 |

### 预设策略 QcloudAccessForCODINGRoleInThroughTCR

**授权场景**

开通 CODING Devops 服务并完成 CODING_QCSRole 角色授权后，该策略将会与 CODING_QCSRole 角色相关联，完成操作后即可获得腾讯云容器镜像服务（TCR）云资源的相关权限。

**授权步骤**

该策略与预设策略 QcloudAccessForCODINGRole 同时授权，无需额外操作。

**权限内容**

- 容器镜像服务相关

| 权限名称 | 权限说明 |
| --- | --- |
| tcr:CreateWebhookTrigger | 创建触发器 |
| tcr:ModifyWebhookTrigger | 更新触发器 |
| tcr:DeleteWebhookTrigger | 删除触发器 |
| tcr:DescribeWebhookTriggerLog | 获取触发器日志 |
| tcr:PushRepository | 推送镜像 |
| tcr:PushRepositoryPersonal | 个人版推送镜像 |

# 快速开始

最近更新时间：2023-08-09 10:16:57

下文将会演示如何利用 CODING 持续集成模板部署一个基于 Node.js + Express + Docker 的应用。

点击查看视频

## 前提条件

使用 CODING 持续集成的前提是，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

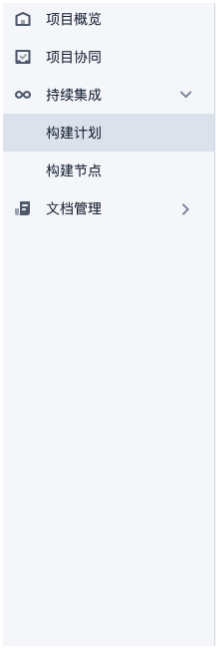
## 进入项目

1. 登录 CODING 控制台，单击团队域名进入 CODING 使用页面。
2. 单击团队首页左侧的**项目**，进入项目列表页，选择目标项目。
3. 进入左侧菜单中的**持续集成**。

## 快速开始

### 步骤1：创建构建计划

进入项目后，选择左侧的**持续集成** > **构建计划**选择新建构建计划。该计划将会演示如何基于 Nodejs + Express 实现全自动检出代码 > 单元测试 > 构建 Docker 镜像 > 推送到 Docker 制品库 > 部署到远端服务器（可选步骤）。





### 步骤2：选择持续集成模板

选择 Node + Express + Docker 持续集成模板。


**步骤3：选择代码源**

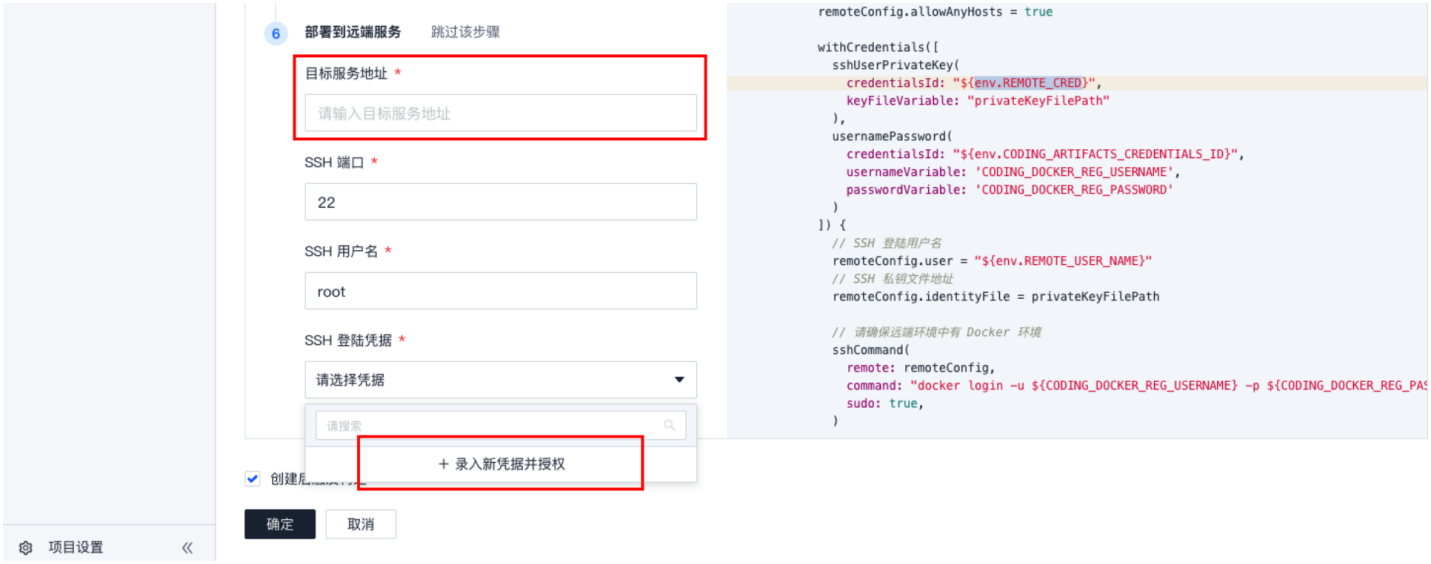
在示例项目中推荐在代码仓库栏选择**示例代码**作为代码源，这样系统会自动在您的项目中新建立一个示例代码仓库。您也可以在自定义构建计划中也可以选择已经创建好的代码仓库。


**步骤4：选择 CODING Docker 制品库**

构建计划结束后会生成一个构建结果，在这里选择拟推送到的 CODING Docker 制品库。若还没有制品库，可以利用快捷方式新建一个制品库。


### 步骤5：填写远端服务信息（可跳过）

填写拟部署的远端服务器信息，包含 IP 地址及端口等信息并录入服务器 SSH 登录凭据。信息填写正确无误后，待构建计划完成后会将制品发送至远端服务器中，通过一个网址便可预览发布后的效果。如果暂时不需要部署到远端服务器，可以选择跳过此步骤。

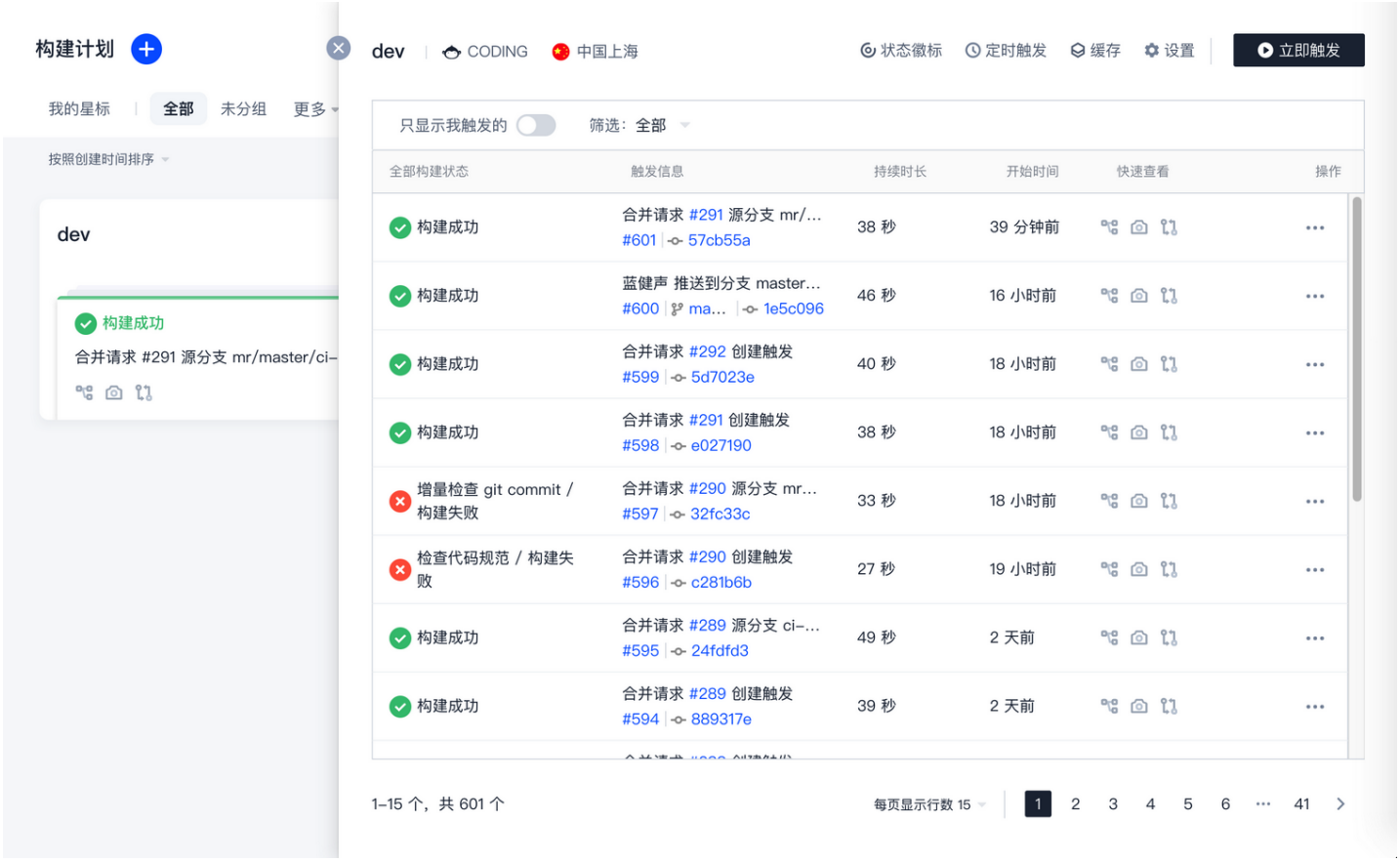



单击**录入新凭据并授权**后，若您本身是通过 SSH 私钥登录远程服务器的，在录入方式中则直接单击**手动录入已有 SSH 私钥**。录入完成后可以在**项目设置** > **开发者选项** > **凭据管理**中查看。

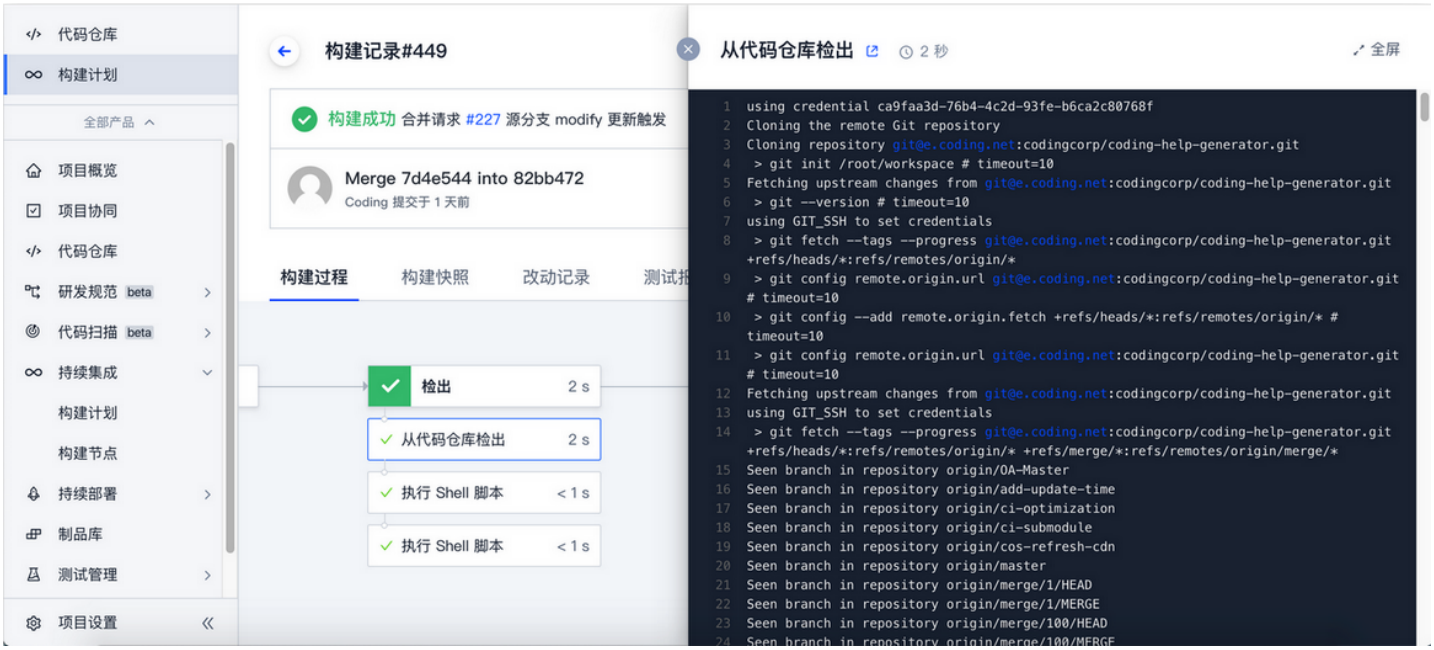

**若不知道如何使用 SSH 私钥登录远程服务器，请在录入方式中单击自动创建 SSH 密钥对，并需要手动将公钥配置到目标远端服务的 ~ssh/authorized_ keys 文件夹中。**

### 步骤6：单击创建并查看构建结果

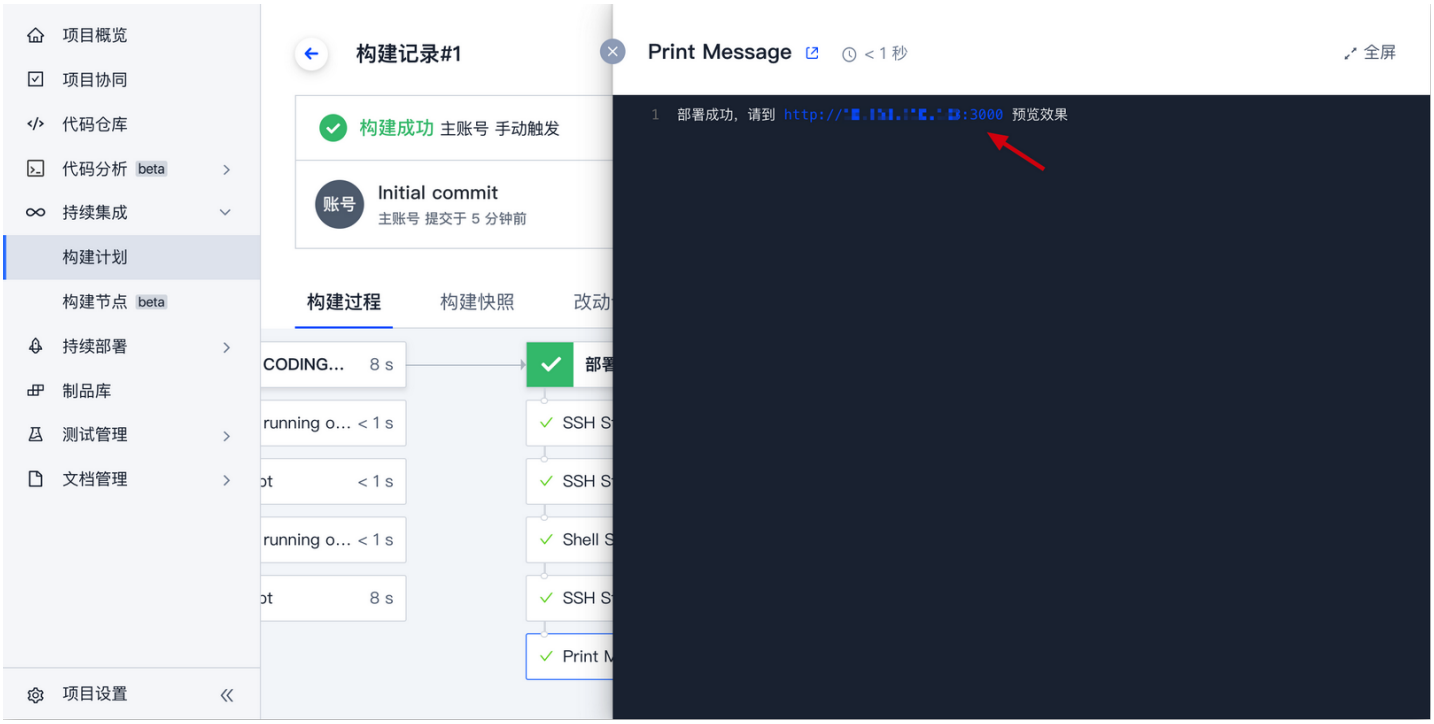
**单击确定保存构建计划。如果勾选了创建后触发构建，构建计划会立即开始执行。在构建计划执行的过程中，可以在构建计划记录列表页查看构建详情。**




**单击构建记录可以查看流水线上每一个阶段是否运行成功，还可以看到每一个步骤命令的具体的执行效果和日志。**




若步骤五运行正常，则可以在构建计划中看到制品输出的网址信息。




**步骤7：修改远端服务器信息**

在步骤五中您已经配置了远端服务器的地址和 SSH 密钥。如果需要更改，可以到**持续集成计划> 设置 > 变量与缓存**中更改。


**说明：**

**构建计划创建成功后，当不需要构建该计划时支持禁用构建计划。已禁用的构建计划不会被触发，启用后可正常运行。**


## 更多内容

您可以通过持续集成计划设置自定义持续集成构建环节。

### 基础信息

- 云服务器与自定义构建节点

### 配置流程

- 使用图形化界面配置构建流程
- 构建不同类型的制品并交付到制品库

**触发规则**

- 配置持续集成的触发方式

**变量与缓存**

- 调取安全凭据并配置到环境变量
- 配置项目缓存

# 编写构建流程

# 文本编辑器

最近更新时间：2021-12-31 14:14:33

本文为您介绍如何使用持续集成中的文本编辑器。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
   构建任务的流程本质上是在遵循配置文件中所定义的流程与步骤。CODING 持续集成全面兼容 Jenkinsfile，在文本编辑器中编写的配置文件所需遵循的语法规范与 Jenkinsfile 保持一致即可成功运行。
3. 进入项目后，单击左侧的**持续集成**，单击**构建计划的设置**。


4. 单击**流程配置**中的**文本编辑器**。

构建过程语法遵循 Jenkinsfile 语法，详情请参考下方文档：

- Jenkinsfile 官方文档
- 配置详情

# 流程配置详情

最近更新时间：2022-01-04 09:36:04

本文主要用于辅助编写构建过程，以及说明各项步骤的参数详情。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能，单击**构建计划 > 文本编辑器**并在其中进行编排。

## 代码仓库

### Git

用以检出当前项目的 Git 仓库源代码。本指令是 checkouk 指令的一个简易写法。
参数列表：

| 参数 | 类型 |
| --- | --- |
| Git 地址 url | string |
| 分支 branch | string |
| 变更日志 changelog | string |
| 身份认证 ID credentialsId | string |
| Poll poll | boolean |

#### 从版本控制检出

通用的检出 SCM（Git，SVN）代码。
这个步骤返回一个 Map 格式的内容。例如您用 Git，您可以这样：

```
def scmVars = checkout scm
def commitHash = scmVars.GIT_COMMIT
// or
def commitHash = checkout(scm).GIT_COMMIT
```

参数 scm 是一个可配置 SCM 类型的对象，目前支持的有如下：

- GitSCM 示例写法：

```
checkout([$class: 'GitSCM', branches: [[name: env.GIT_BUILD_REF]],
userRemoteConfigs: [[url: env.GIT_REPO_URL]]])
```

userRemoteConfigs 参数列表：

| 参数 | 类型 |
| --- | --- |
| locations | 对象数组。 |
| remote | string。 |
| credentialsId | string。 |
| local | string，指明一个本地目录（相对于 workspace）作为检出代码的位置。 |
| depthOption | string，对应 --depth 的内容。默认值是无穷大，详情请参见 Subversion。 |
| ignoreExternalsOption | boolean。 |

- SubversionSCM 从 SVN 服务器检出代码。示例写法：

```
checkout([$class: 'SubversionSCM', remote: 'http://sv-server/repository/trunk'])
```

参数列表：

| 参数 | 类型 |
| --- | --- |
| locations | 对象数组。 |
| remote | string。 |
| credentialsId | string。 |
| local | string，指明一个本地目录（相对于 workspace）作为检出代码的位置。 |
| depthOption | string，对应 --depth 的内容。默认值是无穷大，详情请参见 Subversion。 |
| ignoreExternalsOption | boolean。 |

## 构建过程

### 子节点

参数列表：

label：类型 string 环境标签名称，例如 java-8。

### 收集构建物

把构建结果（例如 jar，war，apk 等）收集起来。请注意，这里收集的构建产物会跟随这个构建历史一起保存和删除，这里只是一个临时的保存空间，更建议用户使用”构建物管理“来进行构建结果的版本化管理。

参数列表：

| 参数 | 类型 |
| --- | --- |
| artifacts | string，可以使用通配符 * 来指定要收集的文件的路径模式，符合 Apache Ant 的路径规则，只允许指定工作空间内的文件。 |
| allowEmptyArchive | boolean，可选；通常情况下，本指令在找不到符合收集模式的文件时会导致构建失败。这个选项如果设置为 true，那么当没有构建产物的时候，构建过程只会发出一个警告，不会导致失败。 |
| caseSensitive | boolean，可选；默认对文件路径规则的匹配是大小写敏感的，如果设置为 false，则不区分大小写。 |
| defaultExcludes | boolean，可选。 |
| excludes | boolean，可选；在设定的路径模式内可以排除一部分文件，同样支持 Apache Ant 的路径规则。 |
| fingerprint | boolean，可选；收集的时候同时计算文件的 hash 信息。 |
| onlyIfSuccessful | boolean，可选；只有当构建成功的时候才收集。 |

### 执行 Shell 脚本

执行一段 Shell 脚本，示例：

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

### 收集 JUnit 测试报告

收集 JUnit 和 TestNG 的测试报告（xml 格式），您可以指定收集哪些 xml 文件，例如 **/build/test-reports/*.xml 需要注意的是不要把那些不是报告文件的 xml 包含在内了，您可以用逗号隔开写多个规则。
参数列表：

| 参数 | 类型 |
| --- | --- |
| testResults | string。 |
| allowEmptyResults | boolean，可选，允许测试报告文件不存在或为空。 |
| keepLongStdio | boolean，可选，所有测试日志都会保留，即便是那些通过的测试用例。 |

## 其他

### 变更目录子步骤

变更目录子步骤。可以在 dir 块内填充若干子步骤，这些子步骤将会在指定的路径目录内执行。

参数列表：
path：类型 string。

### 睡眠

即暂停一段时间，直至所设定的截止时间。与 Unix 的 sleep xxx 类似。

参数列表：

- time：类型 int。
- unit：只能在 NANOSECONDS，MICROSECONDS，MILLISECONDS，SECONDS，MINUTES，HOURS，DAYS 中选择一个。

### 错误信号

发送一个错误信号，往往用在需要根据条件终止部分执行过程的时候。您也可以使用 throw new Excepiton（），但使用 error 步骤可以避免打印过长的异常栈。
参数列表：
message：类型 string。

### 当前目录

把当前目录路径以字符串类型作为返回结果。
参数列表：
tmp：类型 boolean 可选参数，如果选择了，则返回一个跟工作空间关联的临时目录。通常用于在不想污染工作空间目录，又想存放一些临时文件之类的场景。

### 写文件

把指定内容写入文件。
参数列表：

- file：类型 string。
- text：类型 string。
- encoding：类型 string 文件的编码，如果留空将会根据当前运行环境平台默认编码处理，如果遇到 Binary 文件，则会自动被以 Base64 编码后的结果返回。

### 读文件

从相对路径读取文件，并把文件内容作为字符串返回。
参数列表：

- file：类型 string 相对路径地址（相对于工作空间目录）。
- encoding：类型 string 文件的编码，如果留空将会根据当前运行环境平台默认编码处理，如果遇到 Binary 文件，则会自动被以 Base64 编码后的结果返回。

### 重试子步骤

重试一个指定的块直至达到设定的最多重试次数。如果在执行过程中正常结束，则不再重试，如果执行过程中出现异常，则会不断重试直至达到指定的最大重试次数，如果最后一次尝试出现异常，则构建过程会被终止。
参数列表：
count：类型 int。

### 限时子步骤

限定时间执行块内的过程。如果时间到了，将会抛出一个异常 org.jenkinsci.plugins.workflow.steps.FlowInterruptedException 单位参数是可选的，默认是分钟。
参数列表：

- time：类型 int。
- activity：类型 boolean，以日志没有新的内容来计时，而不是以绝对执行时间来计时。
- unit：只能在 NANOSECONDS，MICROSECONDS，MILLISECONDS，SECONDS，MINUTES，HOURS，DAYS 中选择一个。

### 捕获错误子步骤

这里指定的子步骤中的错误将会被捕获。

### 计时子步骤

这里指定的子步骤的执行时间将会以 Unix 时间戳的形式被记录。

### 循环子步骤

这里指定的子步骤将会被循环执行指定次数。

### 条件循环子步骤

这里指定的子步骤将会被循环执行，直到子步骤的结果返回 true。

### 打印消息

在日志中打印消息。
参数列表：
message：类型 string。

### 执行任意的 Pipeline 脚本

添加此步骤可以执行任意的 Pipeline 脚本。

### 执行 Groovy 源文件

构建过程将在此位置执行 Groovy 源文件，示例：

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

### 执行 Yarn 审计

此步骤在指定的目录里执行 yarn audit，并且可以在持续集成结果页面看到对 yarn 依赖审查的漏洞情况。

参数列表：

- directory：类型 string，可选。填写 yarn.lock 所在目录，默认在项目根目录执行。
- collectResult：类型 boolean，可选。收集 Yarn Audit 报告。

### 执行 Npm 审计

此步骤在指定的目录里执行 npm audit，并且可以在持续集成结果页面看到对 npm 依赖审查的漏洞情况。
参数列表：

- directory：类型 string，可选。填写 package.json 所在目录，默认在项目根目录执行。
- collectResult：类型 boolean，可选。收集 Npm Audit 报告。

### 合并合并请求

合并代码。您可以在此步骤合并一个指定的合并请求。
参数列表：

| 参数 | 类型 |
| --- | --- |
| token | string，项目令牌。 |
| depot | string，仓库名称。 |
| mrResourceId | string，指定资源 ID。 |
| commitMessage | string，合并提交信息模板。 |
| deleteSourceBranch | boolean，可选，删除源分支。 |
| fastForward | boolean，可选，尝试 Fast-Forward 模式合并。 |

### 评论合并请求

评论合并请求。您可以在此步骤评论一个指定的合并请求。
参数列表：

| 参数 | 类型 |
| --- | --- |
| token | string，项目令牌。 |
| depot | string，仓库名称。 |
| mrResourceId | string，指定资源 ID。 |
| commentContent | string，评论内容模板。 |

# 图形化编辑器

最近更新时间：2021-12-31 14:14:58

本文为您介绍如何在构建计划设置中使用图形化编辑器。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

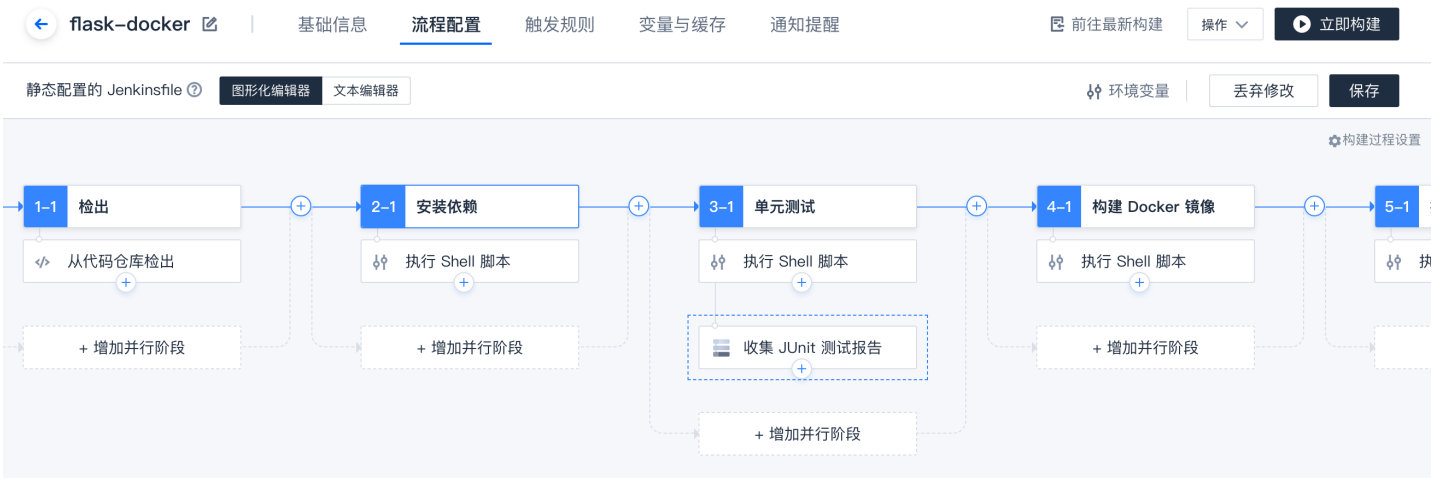
## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能，单击**构建计划 > 文本编辑器**并在其中进行编排。

## 功能介绍

通过命令行文编编辑 Jenkinfile（构建过程描述文件）是最基础人机交互模式，CODING 在基于文本编辑的核心功能上设计了图形化编辑视图，能够兼容命令行编辑中的大部分自定义操作。创新性实现了构建过程边写边看、能够为用户带来所见即所得的直观构建过程编辑体验。

在**构建计划设置 > 流程配置**中选择**图形化编辑器**进行使用。




## 构建流程中的概念

不论图形化或文本类型，编辑器本质上都是用来方便用户查看与编辑构建流程的核心—— Jenkinsfile（过程描述文件），因此在讨论编辑器之前需理解“**过程描述文件**”的几个重要概念。

本篇文档主要聚焦于声明式文件的语法规则。

### 流水线

**流水线**是可自定义的工作模型，它定义了交付软件的完整流程，一般包含构建、测试和部署等阶段。

### 执行环境

执行环境描述了整个**流水线**执行过程或者某个**阶段**的执行环境，必须出现在**描述文件**顶格或者每一个**阶段**里。

| 是否必须 | 是 |
| --- | --- |
| 参数列表 | 见下文 |
| 允许的位置 | 必须出现在**描述文件**顶格或者每一个**阶段**里 |

### 阶段

一个**阶段**定义了一系列紧密相关的**步骤**。每个**阶段**在整条流水线中各自承担了独立、明确的责任。例如“构建阶段”、“测试阶段”或“部署阶段”。通常来讲，所有的实际构建过程都放置在**阶段**里面。

| 是否必须 | 至少一个 |
| --- | --- |
| 参数列表 | 一个强制的字符串类型参数，用以指明阶段名称 |
| 允许的位置 | 在**阶段（stage）**区块内部 |

### 阶段列表

**阶段列表**包含了一系列的**阶段**，一个**阶段列表**最少包含一个**阶段**。**流水线**里必须要有且仅有一个**阶段列表**。

| 是否必须 | 是 |
| --- | --- |
| 参数列表 | 无 |
| 允许的位置 | 在**流水线（pipeline）**内只能出现一次 |

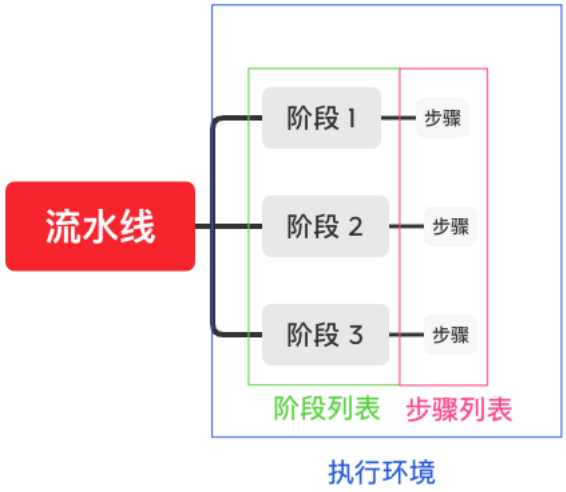
### 步骤列表

**步骤列表**描述了一个**阶段**内具体要做什么事，具体要执行什么命令。例如有一个**步骤（step）**需要系统打印一条“构建中...”的消息，即执行命令 **echo '构建中...'**。

| 是否必须 | 是 |
| --- | --- |
| 参数列表 | 无 |
| 允许的位置 | 在每一个**阶段（stage）**块内 |

### 并行

**并行**用来声明一些并行执行的**阶段**，通常适用于**阶段**与**阶段**之间不存在依赖关系的情况下，用来加快执行速度。注意任何含**并行**区块下的**阶段**不能再设置**执行环境**。



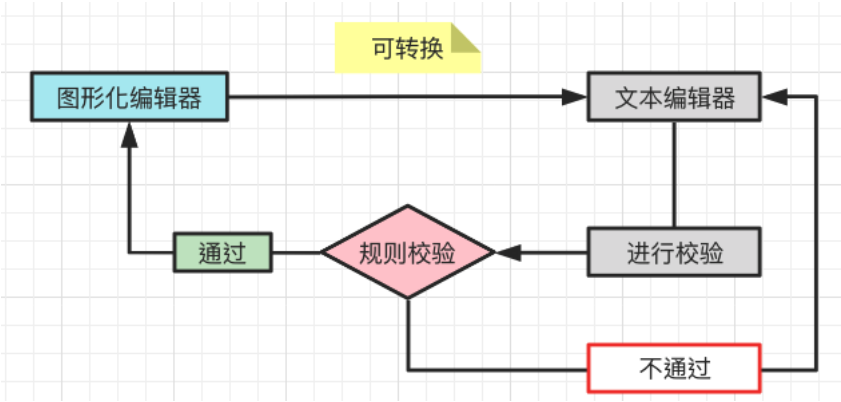

### 示例文件

```
pipeline {
agent any
stages {
stage('检出') {
steps {
sh 'ci-init'
checkout([$class: 'GitSCM', branches: [[name: env.GIT_BUILD_REF]],
userRemoteConfigs: [[url: env.GIT_REPO_URL]]])
}
}
stage('构建') {
steps {
echo '构建中...'
sh 'make'
```

```
echo '构建完成.'
}
}
stage('测试') {
steps {
echo '单元测试中...'
sh 'make check'
junit 'reports/**/*.xml'
echo '单元测试完成.'
}
}
stage('部署') {
steps {
echo '部署中...'
sh 'make publish'
echo '部署完成'
}
}
}
}
```

### 编辑器间转换

图形化编辑器本质上是预设好的代码文本，所以能够无缝转变为文本编辑器。反之则不行，因为文本编辑器上增删的代码文本必须经过“规则校验”的判定程序，通过后才能转换为可编辑视图。




就自定义操作范围而言，文本编辑器所支持的范围比图形化编辑器更大。因图形化编辑器预设了大量常用的步骤，因此适用模式、标准化的工作；而文本编辑器没有限制，仅需满足 Jenkins 语法要求即可，适用于执行具体而特定的任务。

# 配置构建计划
# 触发规则

最近更新时间：2023-05-31 16:35:14

本文为您介绍如何在构建计划设置中设置触发规则。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

## 功能介绍

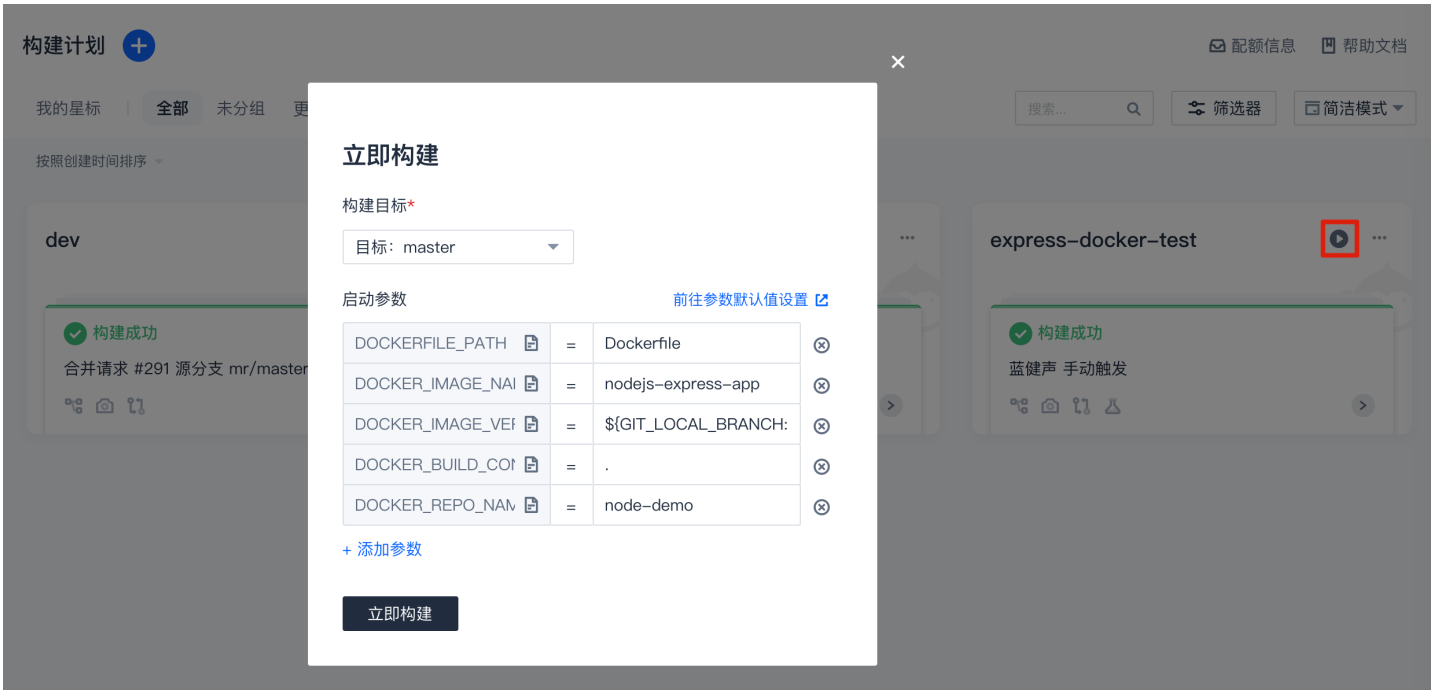
在持续集成计划的配置过程中，您可以按需设置构建计划运行的触发规则，触发规则中包含了构建计划运行的频率与触发的条件。每一个持续集成的构建计划，都会支持以下几种触发方式：

- 手动触发
- 代码变更触发
- 定时触发
- API 触发

上述的多种触发方式可以同时使用。

## 手动触发

您可以主动触发一个构建计划，手动触发时，可输入对应的构建参数，构建参数将以环境变量的形式加入到构建环境中。
在构建计划页面，单击**立即构建**，在弹框中按需选择构建目标（标签、分支、修订版本），输入需要的构建参数完成触发构建。




## 代码变更触发

配置了代码变更触发的构建计划，会自动监听本构建计划中所选择的代码仓库，根据其变化来自动触发构建计划。



**代码更新**

- 推送到 <分支> 时触发构建。
  只有指定的分支更新代码时，才触发构建。
- 推送新标签时触发构建。
  只有创建了新的 git tag，才会触发构建。
- 推送到分支时触发构建。
  任意分支更新都会触发构建。
- 符合分支或标签规则时构建。
  支持正则匹配 ref 全称或者简称：
  i. 如 refs/heads/master 和 master 都能匹配 master 分支触发；
  ii. 如希望 master 和 dev 更新时才触发构建可使用：^refs/heads/(master|dev)。

**合并请求**

目前合并请求会在以下几种情况下执行构建：

- 发起合并请求时触发

- 合并请求的源分支发生变更
- 合并请求的目标分支发生变更
- 归并合并请求时触发

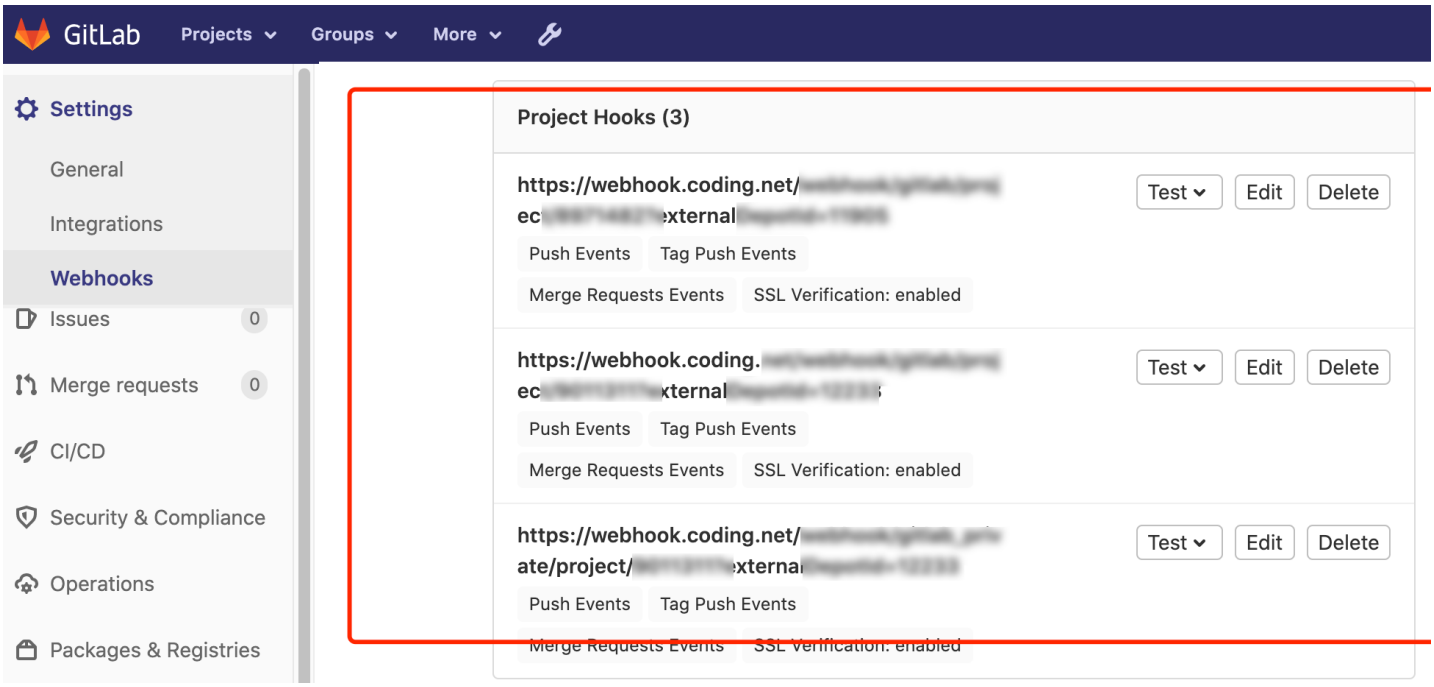
> 说明：
> **合并请求执行与代码更新时执行的不同之处在于，合并请求构建会构建源分支与目标分支合并后的结果，可以尽早发现集成中的错误。**


**自动取消相同构建**

您可以在设置中勾选是否取消**自动取消相同版本号**以及**自动取消相同合并请求**所触发的构建（仅保留最新一个）。


**GitLab 私有云**

绑定 GitLab 私有云 时，会自动创建 GitLab Webhook，后续事件自动通知 CODING，然后匹配上述触发规则设置。



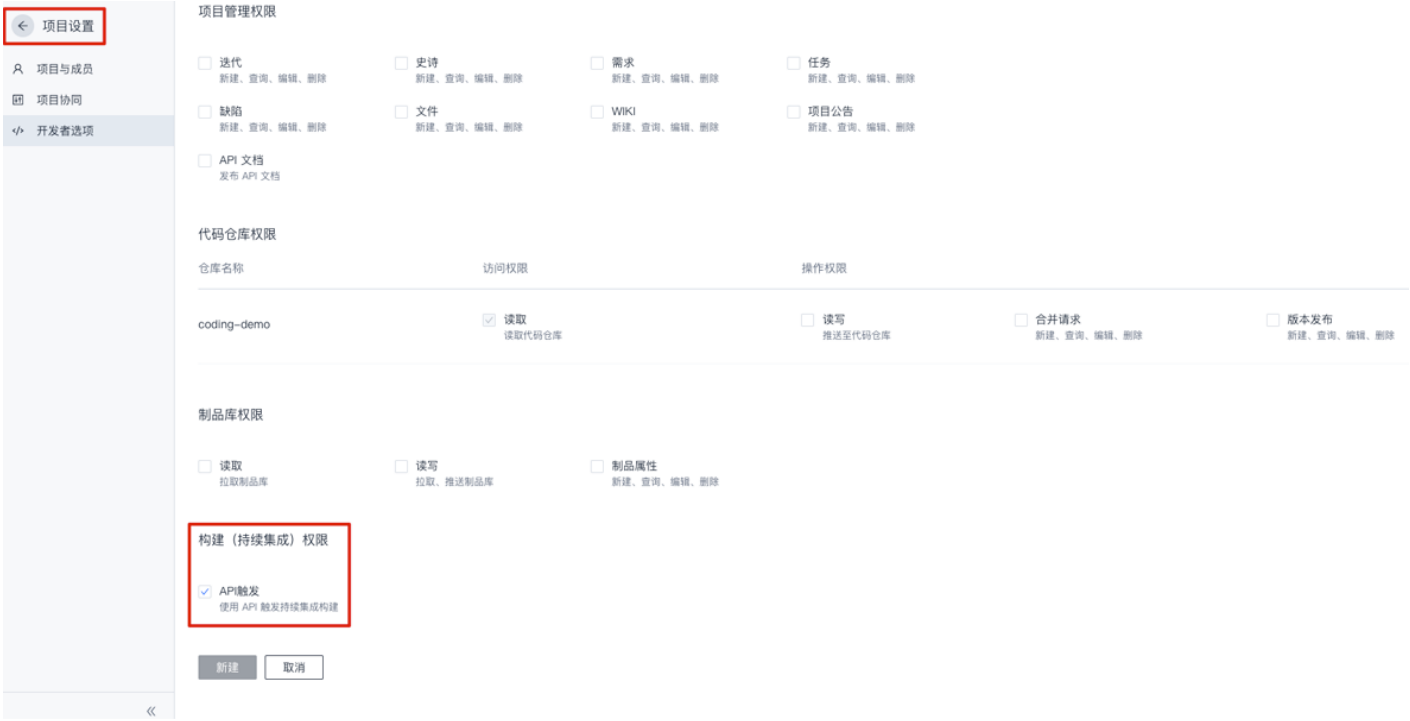

## 定时触发

通过给构建计划添加定时触发配置，您可以周期性或在某个具体的时间点，自动触发一个构建计划，产生一个具体的构建任务。
您可以为一个构建计划添加多个定时触发，没有前后优先级之分，多个定时触发有时间重合的，依然会触发多次构建。


| 类型 | 说明 |
| --- | --- |
| 触发条件（代码无变化时不重复触发定时任务） | 若选中分支代码与上次触发对比无变化，即使到达触发时间，也不会触发构建。 |
| 日期选择 | 您可以选择一周内的多个日期。 |
| 周期触发 | 您可以选定00:00 – 24:00之间的任意时间为周期（精确到小时），按照选中的间隔触发任务。 |
| 单次触发 | 您可以在00:00 – 24:00之间选择任意时间为触发时间点（精确到分钟）。 |

## API 触发

1. 在使用此项功能之前，请确保您已经在**项目设置 > 开发者选项 > 项目令牌 > 新建令牌**中生成了具备持续集成 API 触发权限的令牌。




2. 生成具备相应权限的令牌后便能够调用构建计划中的 API 触发接口。单击生成 curl 命令触发示例后即可生成相应的调用命令。


### API 说明

项目令牌调用 CODING 持续集成 API 时所使用的认证方式为 Basic Auth，下面是关于 API 接口的详细信息和相关参数。

**触发构建任务**

```
POST https://< TEAM_GK >.coding.net/api/cci/job/< JOB_ID >/trigger
```

**请求 body**

```
{
"ref": "master",
"envs": [
{
"name": "my-params-1",
"value": "hello",
"sensitive": 1
},
{
"name": "my-params-2",
"value": "world",
"sensitive": 0
}
]
}
```

**返回 body**

```
{
"code": 0
}
```

**参数说明**

| 参数名字 | 参数位置 | 是否必填 | 类型 | 默认值 | 说明 |
| --- | --- | --- | --- | --- | --- |
| ref | body | 否 | string | master | 构建目标的 ref（ commit sha / tag / branch ），若构建计划不使用代码仓库可以忽略 |
| envs | body | 否 | string | env[] | 构建计划的启动参数 |

**envItem**

| 参数名字 | 参数位置 | 是否必填 | 类型 | 默认值 | 说明 |
| --- | --- | --- | --- | --- | --- |
| name | envItem.name | 是 | string | master | 构建计划的启动参数的名称 |
| value | envItem.value | 否 | string | 无 | 构建计划的启动值 |
| sensitive | envItem.sensitive | 否 | number | 0 | 是否将启动参数设置为保密，设置保密后日志中不可见。1 为保密，0 为明文执行具体而特定的任务 |

# 环境变量

最近更新时间：2021-12-31 14:15:15

本文为您介绍如何在构建计划设置中设置环境变量。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

## 功能介绍

持续集成过程中，我们总会将一些配置（如：账号密码或版本号等）信息以环境变量的形式注入到构建过程中。CODING 持续集成支持多种环境变量使用形式，您可以同时使用以下几种方式来为构建过程注入环境变量，其优先级为从上到下（排在前面的配置优先级最高）：

- Jenkinsfile 中的 withEnv
- Jenkinsfile 中的 environment
- 构建计划（Job）中的启动参数
- 构建计划（Job）设置中的环境变量
- 构建过程中系统内置的环境变量

下文将详细介绍这几种方式的详细说明。

## Jenkinsfile 中的 withEnv 和 environment

1. 您可以在 Jenkinsfile 中使用 environment 来定义环境变量（如下所示）：

```
pipeline {
agent any
environment {
MY_PROJECT = 'project-1'
MY_TEAM = 'team-1'
}
stages {
stage('Build') {
steps {

echo "MY_PROJECT is ${MY_PROJECT}"
echo "MY_TEAM is ${MY_TEAM}"
// 输出内容如下所示：
// MY_PROJECT is project-1
// MY_TEAM is team-1
}
}
}
}
```

2. 在构建过程中，可能需要在不同的阶段使用同名的环境变量。可以使用 withEnv 来针对部分操作设置环境变量，避免全局的环境变量污染。 withEnv 中所执行的 step，都将优先使用 withEnv 设置的环境变量。具体效果可以参考以下例子：

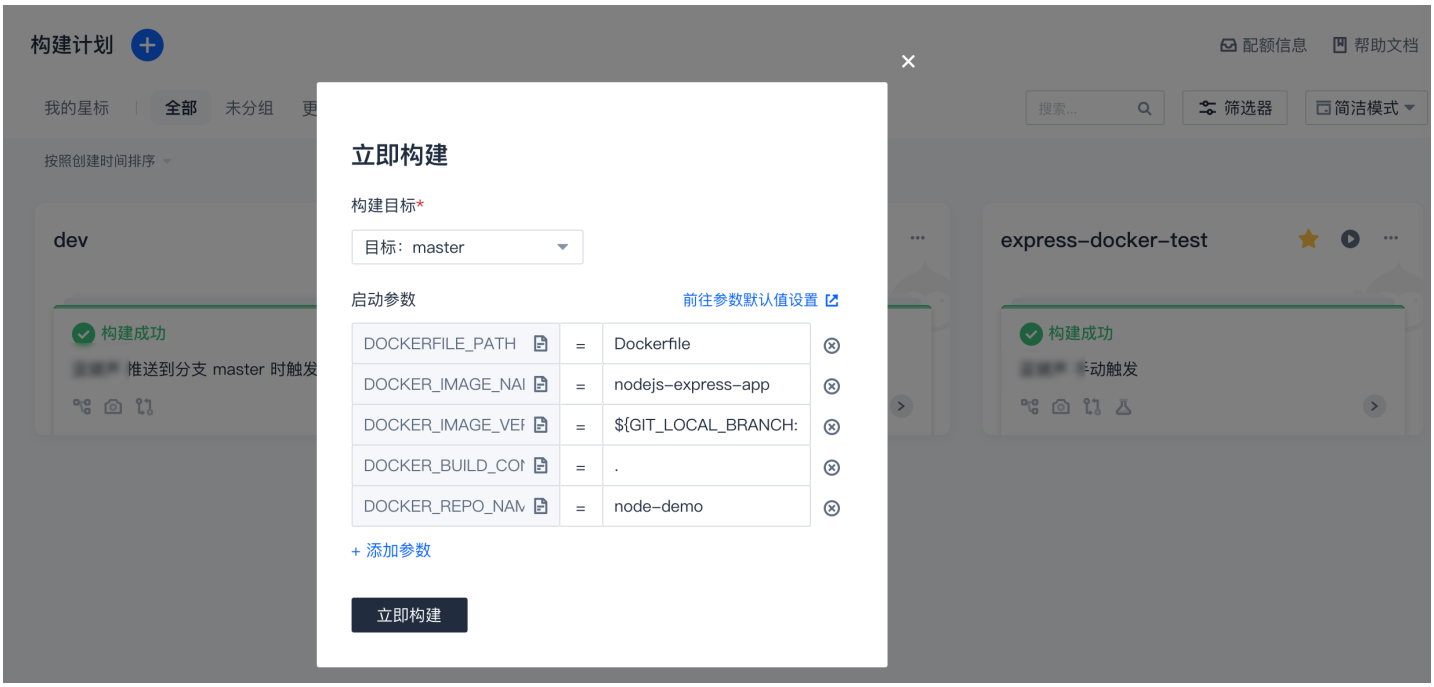
```
pipeline {
agent any
```

```
environment {
MY_PROJECT = 'project-1'
MY_TEAM = 'team-1'
}
stages {
stage('Build') {
steps {

echo "MY_PROJECT is ${MY_PROJECT}"
echo "MY_TEAM is ${MY_TEAM}"
// 输出内容如下所示：
// MY_PROJECT is project-1
// MY_TEAM is team-1

// withEnv 中设置的环境变量只对作用域下的 step 有效，优先级高于 environment
withEnv(['MY_PROJECT=project-2']) {

echo "MY_PROJECT is ${MY_PROJECT}"
echo "MY_TEAM is ${MY_TEAM}"
// 输出内容如下所示：
// MY_PROJECT is project-2
// MY_TEAM is team-1

}
}
}
}
}
```

如果您想了解更多 Jenkinsfile 环境变量相关内容，请参见 使用环境变量。

## 构建计划中的启动参数

优先级仅次于 Jenkinsfile 中配置的环境变量，您可以在启动构建计划时，选择或填写对应的环境变量值。




## 构建计划设置中的环境变量

除了在 Jenkinsfile 中硬编码环境变量，还可以在构建计划的配置中进行设置。目前 CODING 支持添加以下四种类别的环境变量：字符串、单选、多选、Coding 凭据。您还可以将构建计划中的环境变量配置视为启动参数的默认值。


**构建过程中系统内置的环境变量**

CODING 持续集成在构建过程中，会为每一个构建任务注入对应的环境变量，默认注入的环境变量列表您可以在构建快照中查看：


| 序号 | 变量名 | 变量值 |
| --- | --- | --- |
| 1 | DOCKER_REPO_NAME | node-demo |
| 2 | DOCKER_BUILD_CONTEXT | . |
| 3 | DOCKER_IMAGE_VERSION | ${GIT_LOCAL_BRANCH:-branch}-${GIT_COMMIT} |
| 4 | DOCKER_IMAGE_NAME | nodejs-express-app |
| 5 | DOCKERFILE_PATH | Dockerfile |
| 6 | WORKSPACE 系统 | /root/workspace |
| 7 | ANDROID_SDK_ROOT 系统 | /root/programs/android-sdk |
| 8 | CI 系统 | true |
| 9 | JOB_ID 系统 | [redacted] |
| 10 | CCI_JOB_NAME 系统 | express-docker-test |

所有环境变量汇总如下，按照不同的触发规则（代码更新时触发、定时触发、合并请求时触发）进行分类介绍：

| 序号 | 变量名 | 变量含义 | 代码更新时触发 | 定时触发 |
| --- | --- | --- | --- | --- |
| 1 | CREDENTIALS_ID | 部署私钥凭据 CredentialsId 用于拉取仓库 | ✓ | ✓ |
| 2 | DOCKER_REGISTRY_CREDENTIALS_ID | docker 私钥凭据 CredentialsId（等同于 CODING_ARTIFACTS_CREDENTIALS_ID） | ✓ | ✓ |
| 3 | CODING_ARTIFACTS_CREDENTIALS_ID | 制品库私钥凭据 CredentialsId 用于拉取项目内的制品库 | ✓ | ✓ |
| 4 | GIT_HTTP_URL | HTTPS 协议代码仓库地址 | ✓ | ✓ |
| 5 | GIT_BUILD_REF | 构建对应的 Git 修订版本号 | ✓ | ✓ |
| 6 | GIT_DEPLOY_KEY | 代码仓库的部署公钥 | ✓ | ✓ |
| 7 | GIT_COMMIT | 当前版本的修订版本号 | ✓ | ✓ |
| 7 | GIT_COMMIT_SHORT | 修订版本号的前 7 位 | ✓ | ✓ |
| 8 | GIT_PREVIOUS_COMMIT | 前一个构建运行编号的修订版本号 | ✓ | ✓ |
| 9 | GIT_AUTHOR_EMAIL | 本版本最新提交作者邮箱 | ✓ | ✓ |
| 10 | GIT_SSH_URL | 协议代码仓库地址 | ✓ | ✓ |
| 11 | GIT_COMMITTER_NAME | 本版本最新提交者名称 | ✓ | ✓ |
| 12 | GIT_AUTHOR_NAME | 本版本最新提交作者名称 | ✓ | ✓ |
| 13 | REF | 要构建的版本 | ✓ | ✓ |
| 14 | GIT_PREVIOUS_SUCCESSFUL_COMMIT | 前一个构建运行成功的修订版本号 | ✓ | ✓ |

| 序号 | 变量名 | 变量含义 | 代码更新时触发 | 定时触发 |
| --- | --- | --- | --- | --- |
| 15 | GIT_COMMITTER_EMAIL | 本版本最新提交者名称 | ✓ | ✓ |
| 16 | GIT_BRANCH | 触发构建的分支 | ✓ | ✓ |
| 17 | GIT_URL | 仓库 SSH 协议地址 | ✓ | ✓ |
| 18 | GIT_LOCAL_BRANCH/BRANCH_NAME | 本地分支名称 | ✓ | ✓ |
| 19 | FETCH_REF_SPECS | git 要检出的 refs | ✓ | ✓ |
| 20 | GIT_REPO_URL | 仓库 SSH 地址 | ✓ | ✓ |
| 21 | JOB_ID | 构建计划 ID | ✓ | ✓ |
| 22 | JOB_NAME | 构建计划名称 | ✓ | ✓ |
| 23 | CI_BUILD_NUMBER | 构建编号 | ✓ | ✓ |
| 24 | PROJECT_ID | 项目 ID | ✓ | ✓ |
| 25 | PROJECT_NAME | 项目名称 | ✓ | ✓ |
| 26 | PROJECT_WEB_URL | 项目网页地址 | ✓ | ✓ |
| 27 | PROJECT_API_URL | 项目后端 API 地址 | ✓ | ✓ |
| 28 | PROJECT_TOKEN | 项目令牌密码用于读取项目 | ✓ | ✓ |
| 29 | PROJECT_TOKEN_GK | 项目令牌用户名 | ✓ | ✓ |
| 30 | GIT_TAG | 触发构建的 Git 标签 (仅在使用标签构建的时候才会有) | ✓ | – |
| 31 | DEPOT_NAME | 当前使用的代码仓库名称 | ✓ | – |
| 32 | CCI_CURRENT_PROJECT_COMMON_CREDENTIALS_ID (即将上线) | 内置项目令牌的 CredentialsId | ✓ | – |
| 33 | CCI_CURRENT_TEAM（即将上线） | 当前构建环境的企业名，如：myteam.coding.net 中的 myteam | ✓ | – |
| 34 | CCI_CURRENT_DOMAIN（即将上线） | 当前构建环境的域名，如： myteam.coding.net 中的 coding.net | ✓ | – |
| 35 | MR_RESOURCE_ID | 合并请求 ID | – | – |
| 36 | MR_TARGET_BRANCH | 合并请求目标分支名 | – | – |
| 37 | MR_TARGET_SHA | 合并请求目标分支版本号 | – | – |
| 38 | MR_MERGED_SHA | 模拟合并完的版本号 | – | – |
| 39 | MR_SOURCE_BRANCH | 合并请求源分支名 | – | – |
| 40 | MR_STATUS | 合并请求状态 | – | – |
| 41 | MR_SOURCE_SHA | 合并请求源分支版本号 | – | – |

# 构建快照

最近更新时间：2021-12-31 14:15:20

本文为您介绍如何使用构建快照功能。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

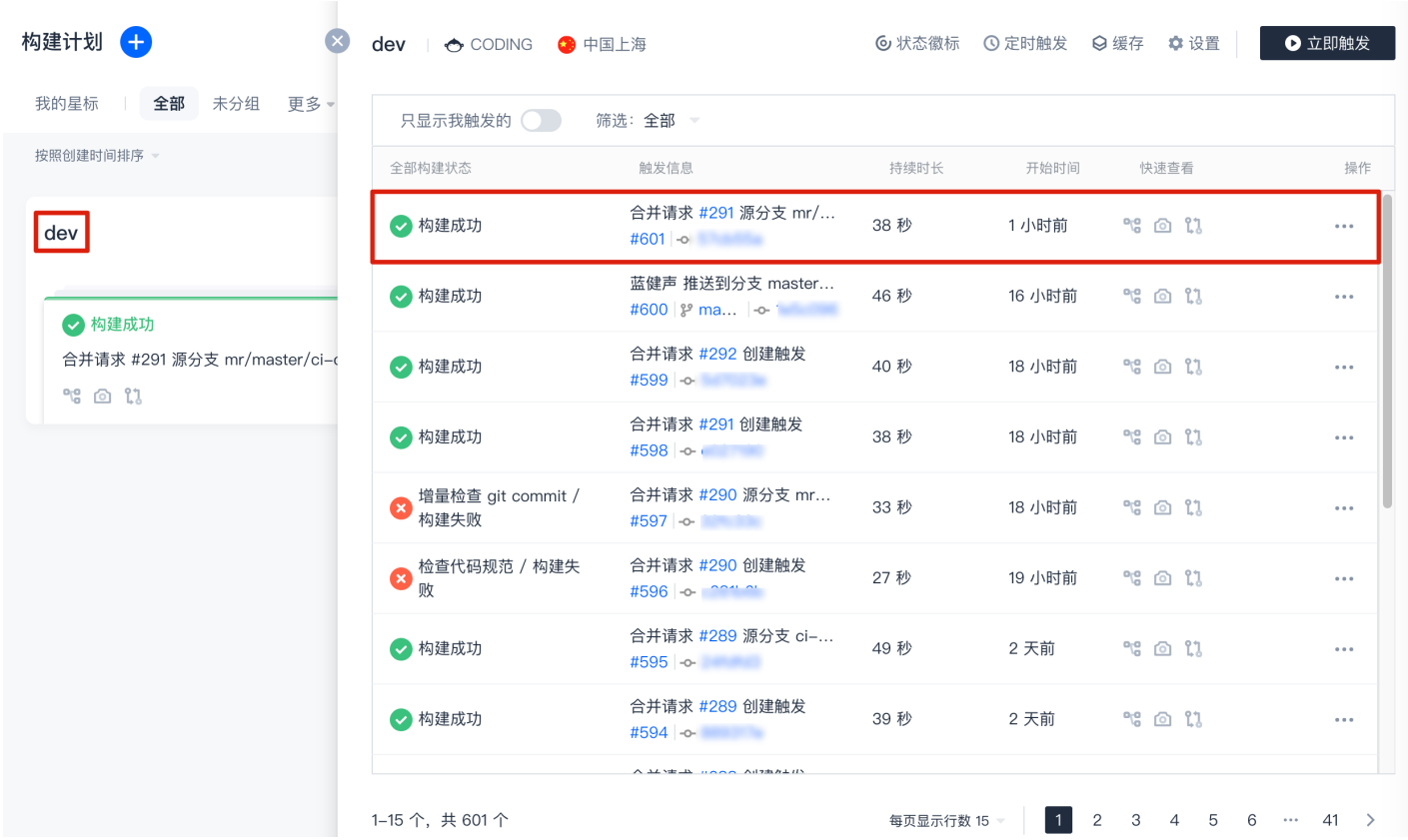
1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

## 功能介绍

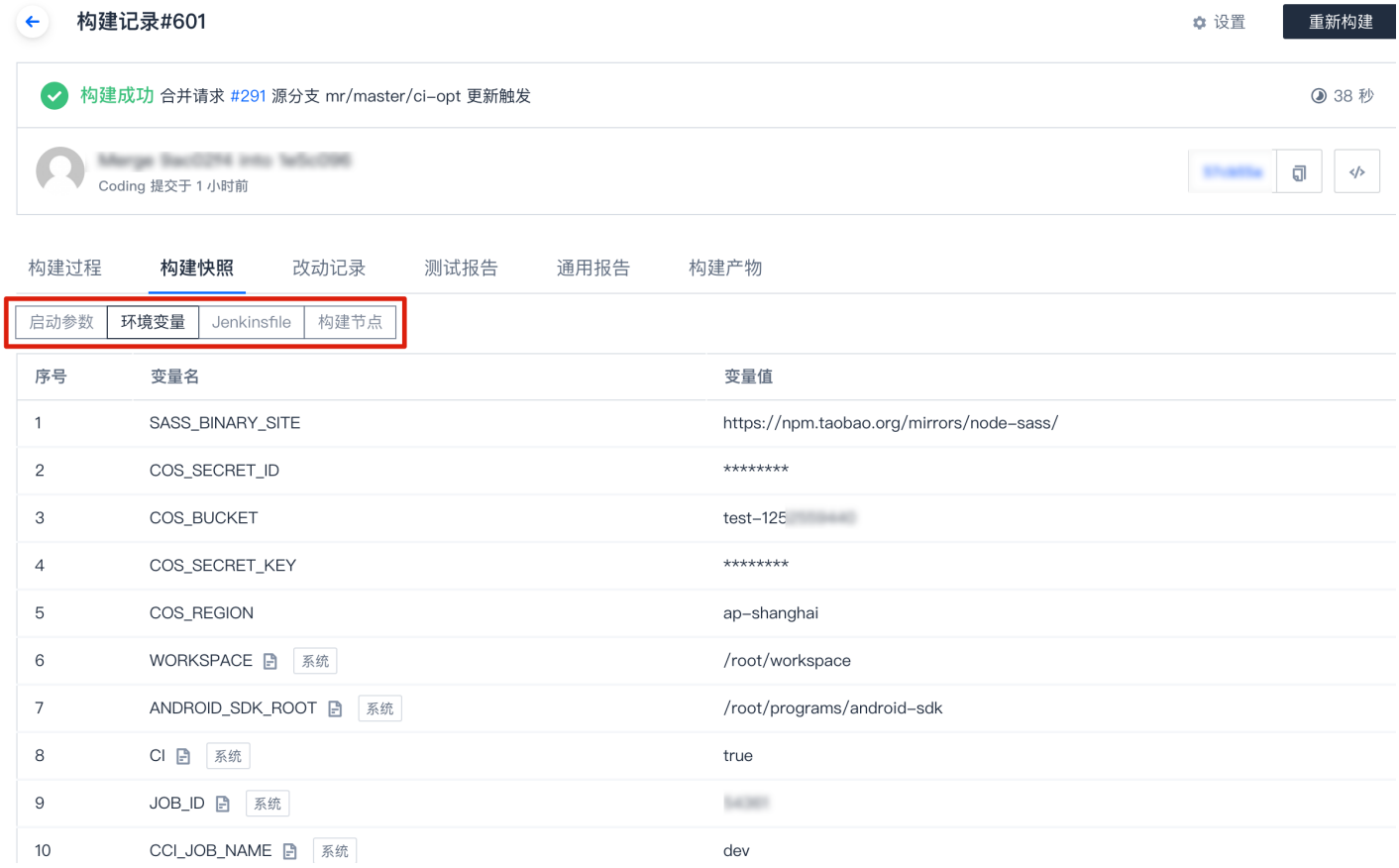
您在持续集成的每一个构建任务，都有可能使用到不同的配置文件或构建参数，为了方便您回顾构建任务的执行过程，CODING 持续集成提供了构建任务的构建快照查看功能。构建快照功能让您能清晰地了解到每次构建记录的配置参数。

## 查看构建配置

1. 在项目中单击**持续集成** > **构建计划**，单击单次构建计划的标题即可查看该计划的所有构建记录，单击进入任一一条构建记录：



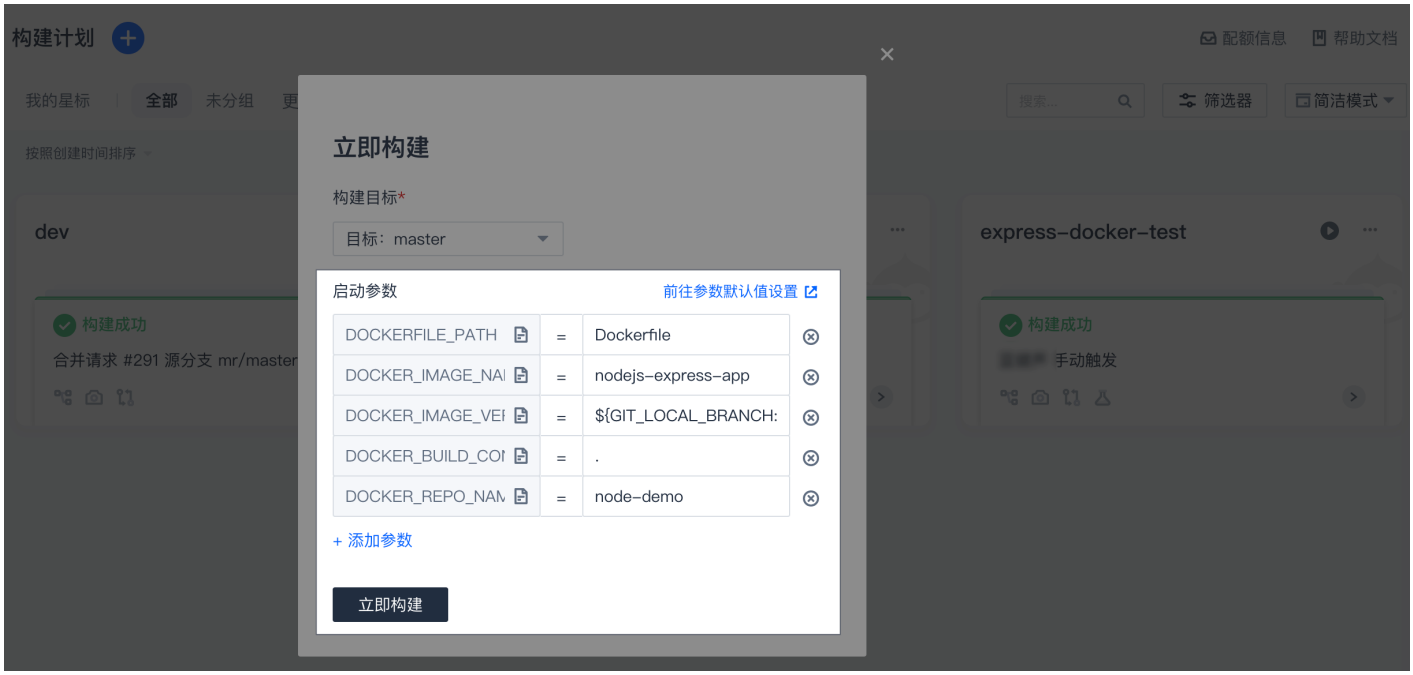
2. 在构建记录中，单击**构建快照**就可以看到每次构建记录的配置快照：启动参数、环境变量、流程配置文件。




| 序号 | 变量名 | 变量值 |
| --- | --- | --- |
| 1 | SASS_BINARY_SITE | https://npm.taobao.org/mirrors/node-sass/ |
| 2 | COS_SECRET_ID | ******** |
| 3 | COS_BUCKET | test-125 |
| 4 | COS_SECRET_KEY | ******** |
| 5 | COS_REGION | ap-shanghai |
| 6 | WORKSPACE 系统 | /root/workspace |
| 7 | ANDROID_SDK_ROOT 系统 | /root/programs/android-sdk |
| 8 | CI 系统 | true |
| 9 | JOB_ID 系统 | |
| 10 | CCI_JOB_NAME 系统 | dev |

### 启动参数

1. 启动参数是您在启动构建任务时，输入的参数内容，会以环境变量的形式注入到构建任务的运行环境中。



2. 在构建完成后，您可以在构建快照中看到配置的启动参数。


| 序号 | 变量名 | 变量值 |
| --- | --- | --- |
| 1 | DOCKER_IMAGE_VERSION | ${GIT_LOCAL_BRANCH:-branch}-${GIT_COMMIT} |
| 2 | DOCKER_IMAGE_NAME | logo-reg |
| 3 | DOCKERFILE_PATH | Dockerfile |
| 4 | DOCKER_REPO_NAME | build |
| 5 | DOCKER_BUILD_CONTEXT | . |

**环境变量**

1. 这里仅包含启动任务时配置的环境变量，不包含所有在运行过程中产生或者动态设置的环境变量。


2. 在环境变量选项卡内，用户可以参看到任务启动时，系统和用户为构建任务设置的环境变量。


| 序号 | 变量名 | 变量值 |
| --- | --- | --- |
| 1 | DOCKER_REPO_NAME | build |
| 2 | DOCKER_IMAGE_VERSION | ${GIT_LOCAL_BRANCH:-bran |
| 3 | DOCKERFILE_PATH | Dockerfile |
| 4 | DOCKER_BUILD_CONTEXT | . |
| 5 | DOCKER_IMAGE_NAME | logo-reg |

### 流程配置文件

通过流程配置选项卡，您可以看到此次构建记录使用的配置文件（Jenkinsfile）。


```
pipeline {
  agent any
  stages {
    stage('检出') {
      steps {
        checkout([$class: 'GitSCM',
        branches: [[name: env.GIT_BUILD_REF]],
        userRemoteConfigs: [[
          url: env.GIT_REPO_URL,
          credentialsId: env.CREDENTIALS_ID
        ]]])
      }
    }
    stage('构建') {
      steps {
        echo '显示环境变量'
        sh 'printenv'
        echo '构建中...'
        sh 'docker version'
        sh './build.sh'
        echo '构建完成.'
      }
    }
    stage('推送到 CODING Docker 制品库') {
      steps {
        script {
          docker.withRegistry(
            "${CCI_CURRENT_WEB_PROTOCOL}://${env.CODING_DOCKER_REG_HOST}",
            "${env.CODING_ARTIFACTS_CREDENTIALS_ID}"
          ) {
            docker.image("${env.CODING_DOCKER_IMAGE_NAME}:${env.GIT_COMMIT}").push()
          }
        }
```

# 缓存目录

最近更新时间：2023-05-25 10:06:07

本文为您介绍如何使用缓存目录功能。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

## 功能介绍

本地项目在按安装依赖包时会把下载的文件缓存起来，以供下次安装使用。例如使用 npm install 命令后会在项目中生成 ./node_modules，缓存储存在 ~/.npm 目录，后者体积更小，更通用。

- 默认构建节点
  CODING 会为每个构建计划自动分配计算资源，构建完毕即销毁，每次构建都会自动重新分配一台构建节点，因此需要指定**缓存目录**用于加速下次构建。
- 自定义构建节点
  若选择自行接入计算资源，并在构建计划中选择通过自定义构建节点执行任务，那么构建完毕不会销毁服务器，故无需指定**缓存目录**。

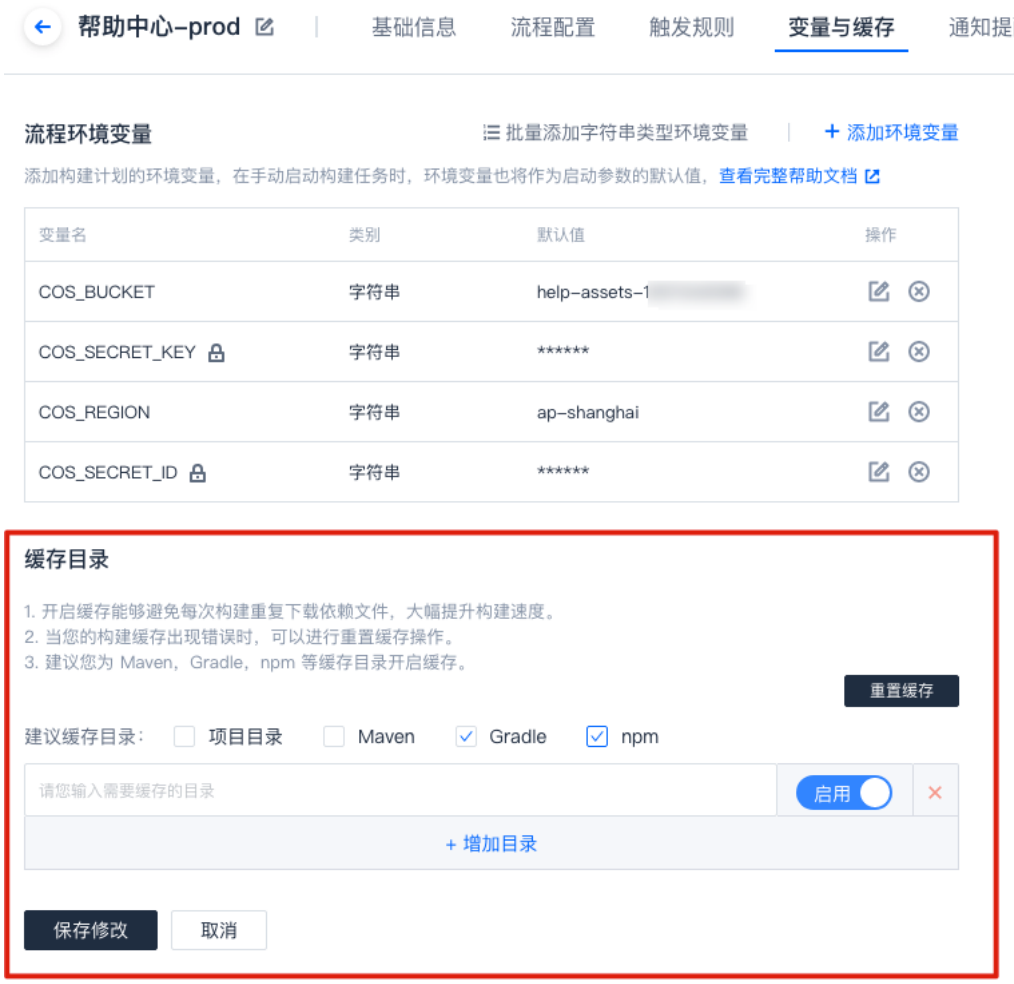
如果在持续集成中使用 Docker 时需要把**缓存目录**挂载至 Docker 中。

## 默认构建节点

1. CODING 为构建计划提供基础的任务计算资源，每次任务都会分配一台云服务器，构建环境为 Linux 系统、分配 root 用户权限，缓存目录如下：

| 包管理工具 | 缓存目录 |
| --- | --- |
| Maven | /root/.m2/ |
| Gradle | /root/.gradle/ |
| npm | /root/.npm/ |
| composer | /root/.cache/composer/ |
| pip3 | /root/.cache/pip/ |
| yarn | /usr/local/share/.cache/yarn/ |

2. 您可以在构建计划设置中的**变量与缓存**中勾选缓存目录，如果未找到目标目录还支持自行录入。




### Docker 构建环境

**若在构建计划中使用 Docker 环境，那么需先行前往变量与缓存中勾选缓存目录，再挂载至 Docker 中。**

**Jenkinsfile**

```
pipeline {
agent any
stages {
stage('检出') {
steps {
checkout([
$class: 'GitSCM',
branches: [[name: env.GIT_BUILD_REF]],
userRemoteConfigs: [[url: env.GIT_REPO_URL, credentialsId: env.CREDENTIALS_ID]]
])
}
}
stage('Java 缓存') {
agent {
docker {
image 'adoptopenjdk:11-jdk-hotspot'
args '-v /root/.gradle/:/root/.gradle/ -v /root/.m2/:/root/.m2/'
reuseNode true
}
}
```

```
steps {
sh './gradlew test'
}
}
stage('npm 缓存') {
steps {
script {
docker.image('node:14').inside('-v /root/.npm/:/root/.npm/') {
sh 'npm install'
}
}
}
}
}
}
```

## 自定义构建节点

在自定义构建节点中使用 Docker 环境时需按照服务器用户名找到对应的缓存目录，例如当 Ubuntu 服务器的默认用户名为 ubuntu 时，缓存目录为 /home/ubuntu/.npm/ ，那么对应的代码为：

```
docker.image('node:14').inside('-v /home/ubuntu/.npm/:/root/.npm/') {
sh 'npm install'
}
```

### 缓存 Docker 基础镜像

1. 如果每次构建都需要拉取 Docker 基础镜像，例如 Dockerfile 基础镜像、CI agent 镜像，那么通常会耗费大量时间，此时就可以通过缓存进行加速。参考以下 Jenkinsfile ，修改镜像名称即可复用：

```
pipeline {
agent any
environment{
DOCKER_CACHE_EXISTS = fileExists '/root/.cache/docker/php-8.0-cli.tar'
}
stages {
stage('加载缓存') {
when { expression { DOCKER_CACHE_EXISTS == 'true' } }
steps {
sh 'docker load -i /root/.cache/docker/php-8.0-cli.tar'
}
}
stage('使用镜像（请修改此段）') {
agent {
docker {
image 'php:8.0-cli'
args '-v /root/.cache/:/root/.cache/'
reuseNode 'true'
}
}
steps {
sh "php -v"
}
}
stage('生成缓存（仅运行一次）') {
when { expression { DOCKER_CACHE_EXISTS == 'false' } }
```

```
steps {
sh 'mkdir -p /root/.cache/docker/'
sh 'docker save -o /root/.cache/docker/php-8.0-cli.tar php:8.0-cli'
}
}
}
}
```

2. 在缓存目录处增加 /root/.cache/ 路径，此时第二次的构建耗时明显更短，说明缓存已生效：


> ⚠ 注意：
> 缓存镜像会逐渐过时，建议定时清除，与官方更新保持一致。

### 保存 Dockerfile

在持续集成中使用 Dockerfile 作为构建环境，需要运行 docker build 命令用以初始化，较为不便。可以将已构建的 Docker 镜像保存到仓库，方便二次拉取复用。

#### Jenkinsfile

```
// 创建 CODING Docker 制品库，获取用户名、密码和仓库地址
sh "docker login -u $DOCKER_USER -p $DOCKER_PASSWORD my-team-docker.pkg.coding.net"

// 使用 Dockerfile 的 md5 做 tag
md5 = sh(script: "md5sum Dockerfile | awk '{print \$1}'", returnStdout: true).trim()
imageFullName = "my-team-docker.pkg.coding.net/my-project/my-repo/my-app:dev-${md5}"

// 检查镜像是否已存在远端仓库
dockerNotExists = sh(script: "docker manifest inspect $imageFullName > /dev/null", returnStatus: true)
def testImage = null
if (dockerNotExists) {
testImage = docker.build("$imageFullName", "--build-arg APP_ENV=testing ./")
sh "docker push $imageFullName"
} else {
testImage = docker.image(imageFullName)
}

// 使用镜像进行自动化测试
testImage.inside("-e 'APP_ENV=testing'") {
stage('test') {
echo 'testing...'
sh 'ls'
echo 'test done.'
```

```
}
}
```

**代码解释：在 shell 中执行下列命令，通过返回值可以判断镜像是否已存在。**

```
$ docker manifest inspect ecoding/foo:bar
no such manifest
$ echo $?
1
```

# 构建环境依赖包

# Go

最近更新时间：2022-01-04 09:41:43

本文为您介绍如何在构建环境中添加 Go 依赖包。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

Go 命令直接使用**代码仓库**存放依赖包，执行 go get 命令时会调用 git clone 命令拉取代码。如果使用私有包，则需配置私有代码库的用户名和密码。

## 导入代码仓库密码

在**项目设置** > **开发者选项**中导入用户名和密码，通过环境变量的方式调取能够防止敏感信息暴露在构建计划中。


## 使用环境变量

1. 把用户名和密码填入持续集成的环境变量：


2. 在文本编辑器中配置 git url，将环境变量填入其中：

```
pipeline {
agent any
stages {
stage('检出') {
steps {
checkout([
$class: 'GitSCM',
```

```
branches: [[name: GIT_BUILD_REF]],
userRemoteConfigs: [[
url: GIT_REPO_URL,
credentialsId: CREDENTIALS_ID
]]])
}
}
stage('安装依赖') {
steps {
sh 'git config --global url."https://${GO_GET_USER}:${GO_GET_PASSWORD}@e.coding.net/codes-farm/go-demo/".insteadOf "https://e.coding.net/codes-farm/go-demo/"'
sh 'go get e.coding.net/codes-farm/go-demo/labstack-echo'
}
}
}
}
```

3. 在起始阶段配置依赖安装环节后，运行持续集成将自动进行代码拉取。

# Maven

最近更新时间：2021-12-31 14:16:03

本文为您介绍如何在构建环境中添加 Maven 依赖包。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

Java 使用 **Maven 制品库**存放依赖包，使用 Maven 或 Gradle 等构建工具进行依赖管理和安装。本文以 Gradle 为例，而 Maven 老项目可使用命令一键升级到 Gradle：

```
gradle init --type pom
```

## Gradle bin

Gradle 构建时会先下载 bin，官方下载链接在境外，内地用户访问可能很慢，建议切换为腾讯云镜像，即修改项目中的 gradle/wrapper/gradle-wrapper.properties：

```
# 腾讯云镜像
distributionUrl=https\://mirrors.cloud.tencent.com/gradle/gradle-6.8.1-bin.zip

# 默认境外
# distributionUrl=https\://services.gradle.org/distributions/gradle-6.8.1-bin.zip
```

## 公共制品库

Maven 公共制品库在境外，CODING 持续集成已内置国内镜像加速，无需配置。如本地开发需要加速，按照以下代码修改 ~/.gradle/init.gradle：

```
def repoConfig = {
all { ArtifactRepository repo ->
if (repo instanceof MavenArtifactRepository) {
def url = repo.url.toString()
if (url.contains('repo1.maven.org/maven2')
|| url.contains('jcenter.bintray.com')
|| url.contains('maven.google.com')
|| url.contains('plugins.gradle.org/m2')
|| url.contains('repo.spring.io/libs-milestone')
|| url.contains('repo.spring.io/plugins-release')
|| url.contains('repo.grails.org/grails/core')
|| url.contains('repository.apache.org/snapshots')
) {
println "gradle init: [buildscript.repositories] (${repo.name}: ${repo.url}) removed"
remove repo
}
}
}

// 腾讯云 maven 镜像聚合了：central、jcenter、google、gradle-plugin
maven { url 'https://mirrors.cloud.tencent.com/nexus/repository/maven-public/' }
```

```
maven { url 'https://maven.aliyun.com/repository/central' }
maven { url 'https://maven.aliyun.com/repository/jcenter' }
maven { url 'https://maven.aliyun.com/repository/google' }
maven { url 'https://maven.aliyun.com/repository/gradle-plugin' }
maven { url 'https://maven.aliyun.com/repository/spring' }
maven { url 'https://maven.aliyun.com/repository/spring-plugin' }
maven { url 'https://maven.aliyun.com/repository/grails-core' }
maven { url 'https://maven.aliyun.com/repository/apache-snapshots' }
}

allprojects {
buildscript {
repositories repoConfig
}

repositories repoConfig
}
```

## 私有制品库

使用 Maven 私有制品库需先获得用户名和密码，请参见 搭建团队级制品库。

### gradle.properties

将制品库地址和用户名和密码添加到项目的 build.gradle 同一级目录下的 gradle.properties 文件中：

```
codingArtifactsMavenUrl=https://codes-farm-maven.pkg.coding.net/repository/share/build/
codingArtifactsUsername=无需填写
codingArtifactsPassword=无需填写
```

### build.gradle

```
repositories {
maven {
url codingArtifactsMavenUrl
credentials {
username = codingArtifactsUsername
password = codingArtifactsPassword
}
}
}

dependencies {
implementation 'com.tencent:cloudpay:1.6'
implementation '[GROUP_ID]:[ARTIFACT_ID]:[VERSION]'
}
```

#### 本地构建

把用户名和密码作为参数传入构建命令：

```
./gradlew build -Dorg.gradle.project.codingArtifactsUsername=foo -Dorg.gradle.project.codingArtifactsPassword=bar
```

#### 持续集成构建

**把用户名和密码填入环境变量：**


```
pipeline {
agent any
stages {
stage('检出') {
steps {
checkout([$class: 'GitSCM', branches: [[name: env.GIT_BUILD_REF]],
userRemoteConfigs: [[url: env.GIT_REPO_URL, credentialsId: env.CREDENTIALS_ID]]])
}
}
stage('编译') {
steps {
sh "./gradlew build -Dorg.gradle.project.codingArtifactsUsername=$CODING_ARTIFACTS_USERNAME -Dorg.gradle.project.codingArtifactsPassword
=$CODING_ARTIFACTS_PASSWORD"
}
}
}
}
```

# PHP

最近更新时间：2021-12-31 14:16:06

本文为您介绍如何在构建环境中添加 PHP 依赖包。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

> 说明：
> PHP 有两种常用扩展依赖包：
> - C 扩展：使用 pecl 安装。
> - PHP 扩展：使用 composer 安装。

## pecl

PHP 可使用 docker-php-ext-install 或 pecl 命令安装扩展：

```
pipeline {
agent {
docker {
reuseNode 'true'
registryUrl 'https://coding-public-docker.pkg.coding.net'
image 'public/docker/php:8.0'
// image 'public/docker/php:7.4' 以及 7.3、7.2、7.1、5.6
args '-v /var/run/docker.sock:/var/run/docker.sock -v /usr/bin/docker:/usr/bin/docker'
}
}
stages {
stage('安装依赖') {
steps {
// 第 1 种方式：PHP 内置扩展
// Possible values for ext-name:
// bcmath bz2 calendar ctype curl dba dom enchant exif fileinfo filter ftp gd gettext gmp
// hash iconv imap interbase intl json ldap mbstring mysqli oci8 odbc opcache pcntl pdo
// pdo_dblib pdo_firebird pdo_mysql pdo_oci pdo_odbc pdo_pgsql pdo_sqlite pgsql phar posix pspell
// readline recode reflection session shmop simplexml snmp soap sockets sodium spl standard
// sysvmsg sysvsem sysvshm tidy tokenizer wddx xml xmlreader xmlrpc xmlwriter xsl zend_test zip
sh 'apt-get update && apt-get install -y libbz2-dev'
sh 'docker-php-ext-install bz2'
sh 'php -i | grep bz2'
// 第 2 种方式：第三方 pecl 扩展
sh "pecl install imagick"
sh 'docker-php-ext-enable imagick'
sh 'php -i | grep imagick'
}
}
}
}
```

## 公共 composer 制品库

composer 公共制品库 Packagist 在境外，内地用户访问可能很慢，建议切换为腾讯云镜像：

```
composer config -g repos.packagist composer https://mirrors.cloud.tencent.com/composer/

composer config -g repo.packagist composer https://mirrors.aliyun.com/composer/

# 恢复默认官方源（境外）
# composer config -g --unset repos.packagist
```

## 私有 composer 制品库

使用私有制品库需先获得用户名和密码，请参见 搭建团队级制品库。

### composer.json

1. 进入 PHP 项目目录，设置制品库地址：

```
composer config repos.private-composer composer https://codes-farm-composer.pkg.coding.net/composer-demo/private-composer
```

2. 可以看到 composer.json 发生了变化，将它提交到代码库。

```
api-php$ git diff composer.json
diff --git a/composer.json b/composer.json
index [redacted]
--- a/composer.json
+++ b/composer.json
@@ -65,5 +65,11 @@
         "post-create-project-cmd": [
             "@php artisan key:generate --ansi"
         ]
+    },
+    "repositories": {
+        "private-composer": {
+            "type": "composer",
+            "url": "https://codes-farm-composer.pkg.coding.net/
composer-demo/private-composer"
+        }
     }
 }
```

### auth.json

3. 进入 PHP 项目目录，设置制品库用户名/密码：

```
composer config http-basic.codes-farm-composer.pkg.coding.net pt03xe33nvww 0ad2d123456
```

4. 可以看到生成了 auth.json，将它忽略掉，不要提交到代码库。

```
api-php$ composer config http-basic.codes-farm-composer.pkg.coding.
net p
api-php$ cat auth.json
{
    "http-basic": {
        "codes-farm-composer.pkg.coding.net": {
            "username": "
            "password": "
        }
    }
}
api-php$ tail -n 2 .gitignore
yarn-error.log
auth.json
```

### 本地安装

1. 本地安装私有包：

```
composer require codes-farm/socialite-providers:0.3.0
```

2. 然后将 composer.lock 提交到代码库。

### 持续集成构建

把用户名和密码填入环境变量：


```
pipeline {
agent {
docker {
reuseNode 'true'
registryUrl 'https://coding-public-docker.pkg.coding.net'
image 'public/docker/php:8.0'
args '-v /var/run/docker.sock:/var/run/docker.sock -v /usr/bin/docker:/usr/bin/docker'
}
}
stages {
stage('检出') {
steps {
checkout([$class: 'GitSCM', branches: [[name: env.GIT_BUILD_REF]],
userRemoteConfigs: [[url: env.GIT_REPO_URL, credentialsId: env.CREDENTIALS_ID]]])
}
}
stage('安装依赖') {
steps {
sh "composer config http-basic.codes-farm-composer.pkg.coding.net ${CODING_ARTIFACTS_USERNAME} ${CODING_ARTIFACTS_PASSWORD}"
```

```
sh "composer install"
}
}
}
}
```

## 构建制品

# Composer

最近更新时间：2023-06-15 11:36:40

本文为您介绍如何在使用持续集成快速构建 Docker 类型制品。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

## 功能介绍

本文将给出如何使用持续集成任务构建 Composer 制品的示例 Jenkinsfile。构建完成后可以使用预置插件便捷的上传至 CODING 制品仓库中。在使用该功能之前，请确保您对 Maven 类型制品库有初步了解，详情请参见 Composer。

## Jenkinsfile

```
pipeline {
agent {
docker {
reuseNode 'true'
registryUrl 'https://coding-public-docker.pkg.coding.net'
image 'public/docker/php:8.0'
// image 'public/docker/php:7.4' 以及 7.3、7.2、7.1、5.6
args '-v /var/run/docker.sock:/var/run/docker.sock -v /usr/bin/docker:/usr/bin/docker -v /root/.cache:/root/.cache'
}
}
stages {
stage('检出') {
steps {
checkout([
$class: 'GitSCM',
branches: [[name: GIT_BUILD_REF]],
userRemoteConfigs: [[
url: GIT_REPO_URL,
credentialsId: CREDENTIALS_ID
]]])
}
}
stage('安装依赖') {
steps {
sh 'curl https://mirrors.cloud.tencent.com/composer/composer.phar -o /usr/local/bin/composer && chmod +x /usr/local/bin/composer'
sh 'composer config -g repos.packagist composer https://mirrors.cloud.tencent.com/composer/'
sh 'find /etc/apt/ -name "*.list" -print0 | xargs -0 sed -i -E "s/[a-z]+.debian.org/mirrors.cloud.tencent.com/g"'
sh 'apt-get update && apt-get install -y libzip-dev unzip zip zlib1g-dev'
sh 'docker-php-ext-install zip'
sh 'composer install --no-dev'
sh 'composer install'
}
}
stage('检查代码规范') {
```

```
steps {
sh './vendor/bin/phpcs --extensions=php --standard=PSR12 src/ tests/'
sh './vendor/bin/phpmd . text phpmd.xml --exclude vendor'
}
}
stage('单元测试') {
steps {
sh 'XDEBUG_MODE=coverage ./vendor/bin/phpunit --coverage-clover coverage.xml --coverage-filter src/ tests/'
}
}
stage('发布私有包') {
steps {
script {
if (env.TAG_NAME =~ /.*/ ) {
ARTIFACT_VERSION = "${env.TAG_NAME}"
} else {
ARTIFACT_VERSION = "dev-${env.BRANCH_NAME.replace('/', '-')}"
}
}
sh 'rm composer.lock'
sh 'zip -r composer-package.zip . -x "./vendor/*"'
withCredentials([
usernamePassword(
// CODING 持续集成的环境变量中内置了一个用于上传到当前项目制品库的凭证
credentialsId: env.CODING_ARTIFACTS_CREDENTIALS_ID,
usernameVariable: 'CODING_ARTIFACTS_USERNAME',
passwordVariable: 'CODING_ARTIFACTS_PASSWORD'
)]) {
script {
sh "curl -f -T composer-package.zip -u ${CODING_ARTIFACTS_USERNAME}:${CODING_ARTIFACTS_PASSWORD} https://${env.CCI_CURRENT_TEAM}-composer.pkg.coding.net/westore/composer?version=${ARTIFACT_VERSION}"
}
}
}
}
}
}
```

## 环境变量

**如果发布到本项目的制品库，无需设置环境变量。如果发布到其他项目，请参见 外部制品库认证。**
**团队成员在进行本地开发时，可以使用自己的账号和密码，在 Linux/macOS 中修改** ~/.composer/auth.json：

```
{
"http-basic": {
"your-team-composer.pkg.coding.net": {
"username": "",
"password": ""
}
}
}
```

# Docker

最近更新时间：2021-12-31 14:16:16

本文为您介绍如何在使用持续集成快速构建 Docker 类型制品。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的  ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

## 功能介绍

本文将给出如何使用持续集成任务构建 Docker 镜像的示例 Jenkinsfile。构建完成后可以使用预置插件便捷的上传至 CODING 制品仓库中。在使用该功能之前，请确保您对 Docker 类型制品库有初步了解，详情请参见 Docker。

## Jenkinsfile

```
pipeline {
agent any
stages {
stage('检出') {
steps {
checkout([
$class: 'GitSCM',
branches: [[name: env.GIT_BUILD_REF]],
userRemoteConfigs: [[url: env.GIT_REPO_URL, credentialsId: env.CREDENTIALS_ID]]
])
}
}
stage('构建 Docker 镜像') {
steps {
script {
ARTIFACT_VERSION = "1.2.0"
// 注意：创建项目时链接标识不要使用下划线，而是连字符，例如 My Project 的标识应为 my-project
// 请修改 build/my-api 为你的制品库名称和镜像名称
CODING_DOCKER_IMAGE_NAME = "${env.PROJECT_NAME.toLowerCase()}/build/my-api"
// 本项目内的制品库已内置环境变量 CODING_ARTIFACTS_CREDENTIALS_ID，无需手动设置
docker.withRegistry("https://${env.CCI_CURRENT_TEAM}-docker.pkg.coding.net", "${env.CODING_ARTIFACTS_CREDENTIALS_ID}") {
docker.build("${CODING_DOCKER_IMAGE_NAME}:${ARTIFACT_VERSION}").push()
}
}
}
}
}
}
```

## 常见问题

### 如何自动生成版本号?

请参见 自动生成版本号。

# Generic

最近更新时间：2021-12-31 14:16:21

本文为您介绍如何在使用持续集成快速构建 Generic 类型制品。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

持续集成生成的文件可保存到文件制品库，适合 React/VUE 等前端项目、Android/iOS/C++ 等客户端项目。

## Jenkinsfile

```
pipeline {
agent any
stages {
stage('检出') {
steps {
checkout([
$class: 'GitSCM',
branches: [[name: GIT_BUILD_REF]],
userRemoteConfigs: [[
url: GIT_REPO_URL,
credentialsId: CREDENTIALS_ID
]]])
}
}
stage('构建') {
steps {
sh 'npm install'
sh 'npm run build'
sh 'tar -zcvf shop-web.tar.gz dist'
}
}
stage('保存到制品库') {
steps {
codingArtifactsGeneric(files: 'shop-web.tar.gz', repoName: 'my-generic', version: '1.0.0')
}
}
}
}
```

## 示例


## 常见问题

**如何自动生成版本号?**

请参见 自动生成版本号。

# Maven

最近更新时间：2021-12-31 14:16:26

本文为您介绍如何在使用持续集成快速构建 Maven 类型制品。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

## 功能介绍

本文将给出如何使用持续集成任务构建 Maven 制品的示例 Jenkinsfile。构建完成后可以使用预置插件便捷的上传至 CODING 制品仓库中。在使用该功能之前，请确保您对 Maven 类型制品库有初步了解，详情请参见 Maven。

## Jenkinsfile 配置

```
pipeline {
agent any
stages {
stage('检出') {
steps {
checkout([$class: 'GitSCM', branches: [[name: env.GIT_BUILD_REF]],
userRemoteConfigs: [[url: env.GIT_REPO_URL, credentialsId: env.CREDENTIALS_ID]]])
}
}
stage('发布到 maven 制品库') {
steps {
withCredentials([
usernamePassword(
// CODING 持续集成的环境变量中内置了一个用于上传到当前项目制品库的凭证
credentialsId: env.CODING_ARTIFACTS_CREDENTIALS_ID,
usernameVariable: 'CODING_ARTIFACTS_USERNAME',
passwordVariable: 'CODING_ARTIFACTS_PASSWORD'
)]) {
withEnv([
"CODING_ARTIFACTS_USERNAME=${CODING_ARTIFACTS_USERNAME}",
"CODING_ARTIFACTS_PASSWORD=${CODING_ARTIFACTS_PASSWORD}"
]) {
sh 'mvn clean install'
sh 'mvn deploy -s ./settings.xml'
}
}
}
}
}
}
```

## settings.xml 配置

在代码库里创建 settings.xml，按照制品库指引页给出的代码，修改下面的 id：

```
BRACKET-FILTER
<?xml version="1.0" encoding="UTF-8"?>
<settings>
<servers>
<server>
<id>my-team-maven-demo-maven</id>
<username>${env.CODING_ARTIFACTS_USERNAME}</username>
<password>${env.CODING_ARTIFACTS_PASSWORD}</password>
</server>
</servers>
</settings>
```

## pom.xml 配置

修改代码库里的 pom.xml，按照制品库指引页给出的代码，修改下面的 id、name 和 url：

```
BRACKET-FILTER
<project>
<!-- 必要属性 -->
<groupId>[GROUP_ID]</groupId>
<artifactId>[ARTIFACT_ID]</artifactId>
<version>[VERSION]</version>

<!-- 自定义仓库 -->
<distributionManagement>
<repository>
<!--必须与 settings.xml 的 id 一致-->
<id>my-team-maven-demo-maven</id>
<name>maven</name>
<url>https://my-team-maven.pkg.coding.net/repository/maven-demo/maven/</url>
</repository>
</distributionManagement>
</project>
```

## 环境变量配置

1. 如果发布到**本项目**的制品库，无需设置环境变量。如果发布到**其他项目**，请参见 外部制品库认证。
2. 团队成员在进行本地开发时，可以使用自己的账号和密码，在 Linux/macOS 中这样设置：

```
export CODING_ARTIFACTS_USERNAME=lilei@example.com
export CODING_ARTIFACTS_PASSWORD=123456
```

## 示例

构建记录#25

构建成功 手动触发

构建过程 构建快照 改动记录 测试报告 通用报告 构建产物

开始
检出 < 1 s
从代码仓库检出 < 1 s
构建 16 s
执行 Shell 脚本 < 1 s
Checks if running o... < 1 s
执行 Shell 脚本 < 1 s
Checks if running o... < 1 s
执行 Shell 脚本 7 s
打印消息 < 1 s
执行 Shell 脚本 3 s
打印消息 < 1 s

制品库

my-maven
Maven 仓库 | 团队内

my-maven

类型 Maven | 权限 团队内

指引 包列表

搜索...

| 包名 | 最新推送版本 |
| --- | --- |
| farm.codes:bar -- | 1.0-SNAPSHOT |

# Npm

最近更新时间：2021-12-31 14:16:31

本文为您介绍如何在使用持续集成快速构建 Npm 类型制品。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

## 功能介绍

本文将给出如何使用持续集成任务构建 Npm 制品的示例 Jenkinsfile。构建完成后可以使用预置插件便捷的上传至 CODING 制品仓库中。在使用该功能之前，请确保您对 Npm 类型制品库有初步了解，详情请参见 Npm。

## npmrc

创建 .npmrc 文件，提交到代码库，参考代码：

```
registry=https://my-team-npm.pkg.coding.net/my-project/my-npm-repo/
always-auth=true
//my-team-npm.pkg.coding.net/my-project/my-npm-repo/:username=${CODING_ARTIFACTS_USERNAME}
//my-team-npm.pkg.coding.net/my-project/my-npm-repo/:_password=${CODING_ARTIFACTS_PASSWORD_ENCODED}
//my-team-npm.pkg.coding.net/my-project/my-npm-repo/:email=zhang@example.com
```

## Jenkinsfile

```
pipeline {
agent any
stages {
stage('检出') {
steps {
checkout([
$class: 'GitSCM',
branches: [[name: GIT_BUILD_REF]],
userRemoteConfigs: [[
url: GIT_REPO_URL,
credentialsId: CREDENTIALS_ID
]]])
}
}
stage('安装依赖') {
steps {
sh 'rm -rf /usr/lib/node_modules/npm/'
dir ('/root/.cache/downloads') {
sh 'wget -nc "https://coding-public-generic.pkg.coding.net/public/downloads/node-linux-x64.tar.xz?version=v16.13.0" -O node-v16.13.0-linux-x64.tar.xz | true'
sh 'tar -xf node-v16.13.0-linux-x64.tar.xz -C /usr --strip-components 1'
}
withCredentials([
```

```
usernamePassword(
// CODING 持续集成的环境变量中内置了一个用于上传到当前项目制品库的凭证
credentialsId: env.CODING_ARTIFACTS_CREDENTIALS_ID,
usernameVariable: 'CODING_ARTIFACTS_USERNAME',
passwordVariable: 'CODING_ARTIFACTS_PASSWORD'
)]) {
script {
sh '''
CODING_ARTIFACTS_PASSWORD_ENCODED=$(echo -n "${CODING_ARTIFACTS_PASSWORD}" | base64)
echo "CODING_ARTIFACTS_USERNAME=${CODING_ARTIFACTS_USERNAME}" >> $CI_ENV_FILE
echo "CODING_ARTIFACTS_PASSWORD_ENCODED=${CODING_ARTIFACTS_PASSWORD_ENCODED}" >> $CI_ENV_FILE
'''
readProperties(file: env.CI_ENV_FILE).each {key, value -> env[key] = value }
}
}
sh 'npm install'
}
}
stage('检查代码规范') {
steps {
sh 'npm run lint'
}
}
stage('单元测试') {
steps {
sh 'npm run test'
}
}
stage('发布私有包 npm') {
when {
anyOf {
tag '*'
}
}
steps {
sh 'npm publish'
}
}
}
}
```

## 环境变量配置

1. 如果发布到**本项目**的制品库，无需设置环境变量。如果发布到**其他项目**，请参见 外部制品库认证。
2. 团队成员在进行本地开发时，可以使用自己的账号和密码，在 Linux/macOS 中这样设置：

```
export CODING_ARTIFACTS_USERNAME=lilei@example.com
export CODING_ARTIFACTS_PASSWORD=123456
```

## 常见问题

### 如何找到 .npmrc 文件?

```
npm config list
```

**如何创建凭据?**

请参见 使用凭据进行认证。

# 推送至外部制品库

最近更新时间：2021-12-31 14:16:35

本文为您介绍如何在使用持续集成快速构建制品后推送至外部制品库。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

## 功能介绍

例如，您希望使用 A 项目的持续集成任务将构建后的制品推送至 B 项目或其他第三方制品仓库中，将分为以下两步：

1. 在 B 项目中新建项目令牌并获取第三方制品仓库的用户名与密码。
2. 在 A 项目中的持续集成任务中添加环境变量。

### 创建项目令牌

1. 在 B 项目的**项目设置** > **开发者选项**中创建**项目令牌**，填写名称后勾选**制品仓库**的**读写**权限。


2. 创建后会获取令牌的用户名与密码。


### 使用令牌

在 A 项目的持续集成任务中，录入环境变量，填写在 B 项目中已创建的令牌用户名与密码或第三方制品仓库的用户名与密码，建议使用两项变量名：

```
CODING_ARTIFACTS_USERNAME
CODING_ARTIFACTS_PASSWORD
```


**Jenkinsfile 参考**

```
stage('发布到 maven 制品库') {
steps {
echo "${env.CODING_ARTIFACTS_USERNAME}"
sh 'mvn clean install'
sh 'mvn deploy -s ./settings.xml'
}
}
```

# 自动生成版本号

最近更新时间：2021-12-31 14:16:40

本文为您介绍如何在持续集成中自动生成版本号。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

在持续集成中打包时，需要根据场景生成不同的版本号：

| 场景 | 版本号规则 | 版本号示例 | 常用环境 |
| --- | --- | --- | --- |
| 合并请求 | mr-{合并请求 ID}-{hash} | mr-123-3a11e12 | 开发或测试 |
| 合并后（或推送分支） | {分支名}-{hash} | main-3a11e12 | 测试 |
| 推送 tag | {tag} | 1.2.0 | 预发布或生产 |

有两种方式可以实现。

## 插件

使用**魔法版本号**插件：

```
stage('打包') {
steps {
useCustomStepPlugin(key: 'coding-public:magic-version', version: 'latest')
script {
readProperties(file: env.CI_ENV_FILE).each {key, value -> env[key] = value }
}
echo "${env.MAGIC_VERSION}"
// codingArtifactsGeneric(files: 'web.tar.gz', repoName: 'my-generic', version: env.MAGIC_VERSION)
}
}
```

## if 判断

写代码进行判断：

```
stage('打包') {
steps {
script {
if (env.TAG_NAME =~ /.*/ ) {
ARTIFACT_VERSION = "${env.TAG_NAME}"
} else if (env.MR_SOURCE_BRANCH =~ /.*/ ) {
ARTIFACT_VERSION = "mr-${env.MR_RESOURCE_ID}-${env.GIT_COMMIT_SHORT}"
} else {
ARTIFACT_VERSION = "${env.BRANCH_NAME.replace('/', '-')}-${env.GIT_COMMIT_SHORT}"
}
}
```

```
echo "${ARTIFACT_VERSION}"
// codingArtifactsGeneric(files: 'web.tar.gz', repoName: 'my-generic', version: ARTIFACT_VERSION)
}
}
```

# 推送至 TCR 镜像仓库

最近更新时间：2021-12-31 14:16:46

本文为您介绍如何在使用持续集成快速构建制品后推送至 TCR 镜像仓库。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

本文将通过示例项目介绍如何使用持续集成构建制品并交付至腾讯云 TCR 镜像仓库。

## 前置准备

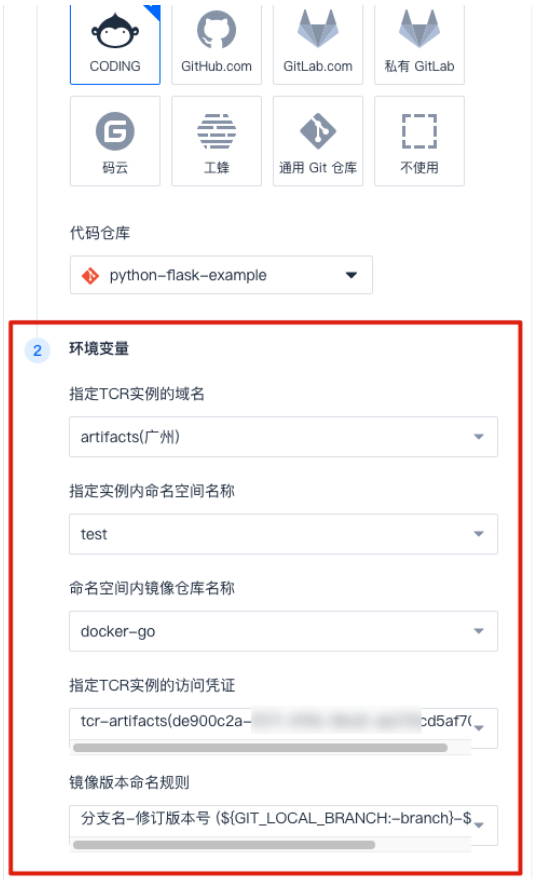
- 导入 示例仓库。
- 开通 TCR 服务 并创建镜像仓库。
- 绑定 腾讯云账号。

## 创建持续集成任务

进入任一项目，单击进入左侧菜单栏的**持续集成**功能，创建构建计划后选择**构建镜像并推送至 TCR 企业版**模板。


## 配置持续集成任务

1. 在构建过程中需选择**示例仓库**，在环境变量中选择已在腾讯云中创建的 TCR 镜像仓库服务。

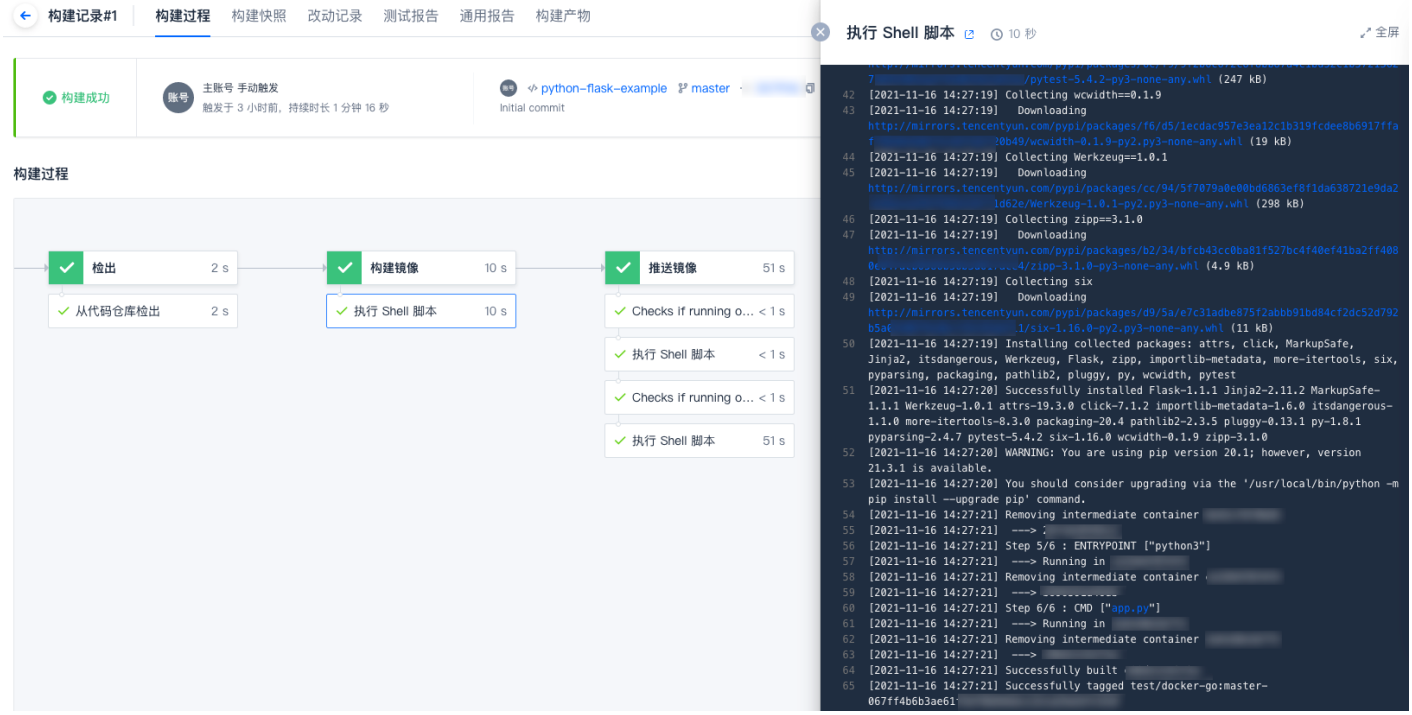



```
environment {
  DOCKER_REGISTRY_HOSTNAME = "${TCR_REGISTRY_HOSTNAME}"
  DOCKER_REGISTRY_CREDENTIAL = "${TCR_REGISTRY_CREDENTIAL}"
  DOCKER_REPOSITORY_NAME = "${TCR_NAMESPACE_NAME}/${TCR_REPOSITORY_NAME}"
  DOCKER_IMAGE_NAME = "${TCR_IMAGE_NAME}"
}
stages {
  stage("检出") {
    steps {
      checkout([
        $class: 'GitSCM',
        branches: [[name: GIT_BUILD_REF]],
        userRemoteConfigs: [[
          url: GIT_REPO_URL,
          credentialsId: CREDENTIALS_ID
      ]]])
    }
  }
  stage('构建镜像') {
    steps {
      // 确保仓库中有可用的 Dockerfile
      sh "docker build -t ${DOCKER_REPOSITORY_NAME}:${DOCKER_IMAGE_NAME} ."
    }
  }
  stage('推送镜像') {
    steps {
      script {
        docker.withRegistry("https://${DOCKER_REGISTRY_HOSTNAME}", "${DOCKER_REGISTRY_CREDENTIAL}") {
          docker.image("${DOCKER_REPOSITORY_NAME}:${DOCKER_IMAGE_NAME}").push()
        }
      }
    }
  }
}
}
```

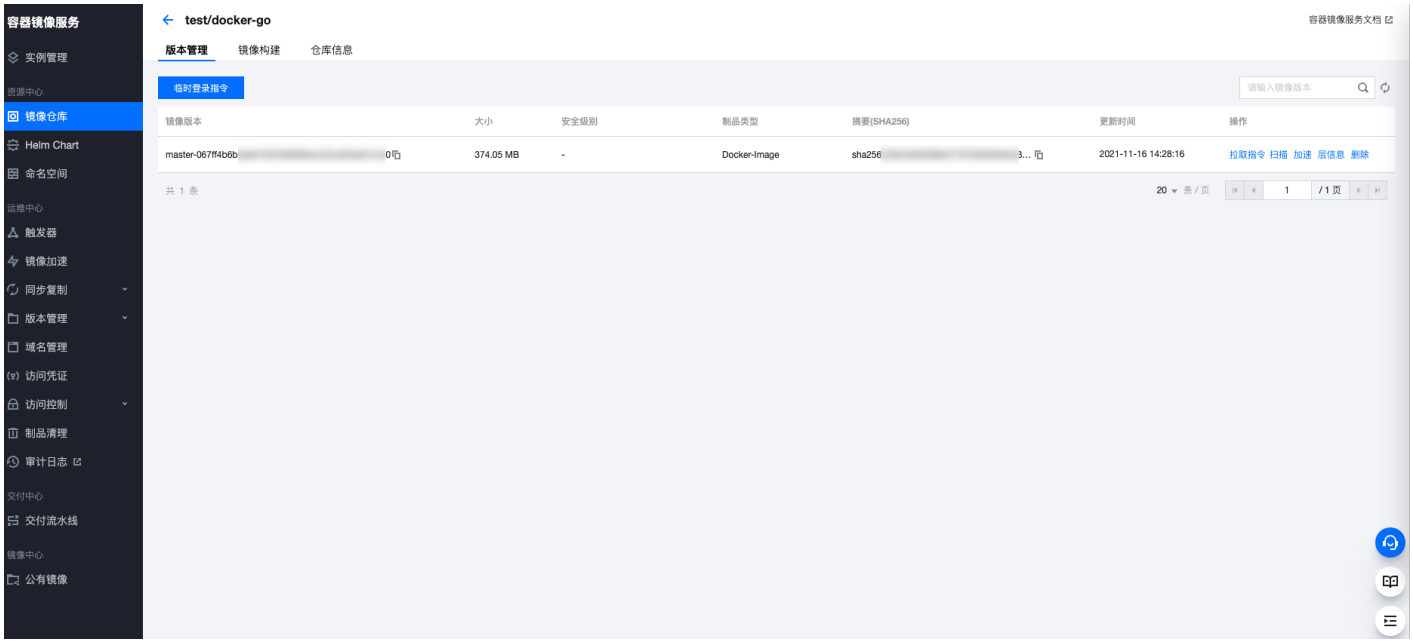
2. 其中，访问凭据需点选一键录入 TCR 凭据并使用。


## 触发任务

1. 配置完成后，触发持续集成任务即可自动构建制品并完成发布。您可以在构建过程中查询各项步骤的运行详情与完成情况。




2. 持续集成任务完成后，制品将自动交付至 TCR 服务中的镜像仓库。




3. 如果您希望通过手动方式直接推送制品至 TCR 镜像仓库，请参见 企业版快速入门。

# 构建节点
# 构建节点类型

最近更新时间：2021-12-31 14:16:52

本文为您介绍构建节点的类型与差别。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

## 功能介绍

当您在使用 CODING 持续集成进行构建时，本质上是调用计算资源作为**构建节点**完成构建任务。您可以选择使用官方默认提供的云计算资源或自行接入自定义构建节点两种方式运行构建任务。

## 默认构建节点

CODING 官方提供中国上海、中国香港、美国硅谷三地的计算资源用于执行构建任务，计算资源的限额策略为：


基础包和标准版团队为免费，请参见 CODING。
默认节点内置了构建环境，其中预装了开发语言 SDK、命令行工具等服务，请参见 默认节点环境。

## 自定义构建节点

在实际的开发项目中，所涉及的开发环境可能是多种多样的，当官方节点的构建环境无法承载项目的持续集成要求时，例如需要使用 macOS Xcode 构建 iOS 应用时，就可以通过接入自定义类型节点（物理机、云服务器和容器等）运行特定任务。
更多关于自定义构建节点的内容请参见 自定义节点。

# 默认节点环境

最近更新时间：2021-12-31 14:16:56

本文为您介绍默认节点的构建环境。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

## 功能介绍

构建任务由构建节点执行，而构建环境指的便是构建节点中内置的系统底层环境，预装了开发语言 SDK、命令行工具等服务。
构建环境有以下三类：

- 默认环境
- 自定义版本
- Docker 环境

若开发项目对运行环境有特定要求，如 swift 项目需要在 macOS 环境下运行，可以参考 自定义构建节点自行接入构建节点。

## 默认环境

在构建计划的开始环节中选择构建环境。


对应的 Jenkinsfile 为 agent any：

```
pipeline {
agent any
stages {
stage("检出") {...}
stage("检查代码规范") {...}
}
}
```

CODING 云服务器为 Ubuntu 系统，预装了以下 SDK 和命令行工具：

| SDK | 命令行工具 |
| --- | --- |
| • android-sdk: 26.1.1<br>• build-essential<br>• dotnet-core: 2.2<br>• elixir: 1.8.1<br>• erlang: Erlang/OTP 21<br>• go: 1.14.4<br>• java: 1.8.0_191<br>• nodejs: 10<br>• php: 8.0、7.4、7.3<br>• python3/pip3: 3.9、3.8、3.7<br>• python: 2.7.12<br>• ruby: 2.6.0 | • bundler: 1.17.2<br>• cmake: 3.5.1<br>• composer：1.10.8<br>• coscmd：1.8.5.36<br>• docker-compose: 1.26.0<br>• docker：20.10.6<br>• git-lfs: 2.7.2<br>• git：2.28.0<br>• gradle: 4.10.2<br>• helm: 2.13.1<br>• jq: 1.5-1-a5b5cbe<br>• kubectl: 1.18.4<br>• maven: 3.6.3<br>• mercurial: 3.7.3<br>• pigz: 2.3.1<br>• rancher: 2.2.0<br>• rvm: 1.29.7<br>• sshpass: 1.05<br>• svn: 1.9.3<br>• tccli: 3.0.67.1<br>• vsftpd: 3.0.3<br>• yarn: 1.15.2<br>• axcl: 2.5 |

> ⚠ 注意：
> 预装的软件版本有限且定期升级，而各个项目所需要的版本可能不同。

## 自定义版本

在持续集成中可自行下载安装软件的各种版本，如果官网下载较慢，我们亦提供了 镜像服务 以供下载。如需增加制品，欢迎提交至 开源项目。

### Go

```
stage('Go') {
steps {
// 建议设置「缓存目录」 /root/.cache/downloads
sh 'rm -rf /root/programs/go'
dir ('/root/.cache/downloads') {
sh 'wget -nc "https://coding-public-generic.pkg.coding.net/public/downloads/go-linux-amd64.tar.gz?version=1.17.3" -O go-linux-amd64-1.17.3.tar.gz | true'
sh 'tar -zxvf go-linux-amd64-1.17.3.tar.gz -C /root/programs'
}
sh 'go version'
}
}
```

### Helm

```
stage('Helm') {
steps {
dir ('/root/.cache/downloads') {
sh 'wget -nc "https://coding-public-generic.pkg.coding.net/public/downloads/helm-linux-amd64.tar.gz?version=v3.7.1" -O helm-linux-amd64-v3.7.1.tar.gz | true'
sh "tar -zxvf helm-linux-amd64-v3.7.1.tar.gz -C \$HELM_BIN linux-amd64/helm --strip-components 1"
}
```

```
sh 'helm version'
}
}
```

### kubectl

```
stage('kubectl') {
steps {
dir ('/root/.cache/downloads') {
sh 'wget -nc "https://coding-public-generic.pkg.coding.net/public/downloads/kubectl-linux-amd64?version=v1.22.4" -O kubectl-linux-amd64-v1.22.4 | true'
sh 'cp kubectl-linux-amd64-v1.22.4 /usr/local/bin/kubectl'
}
sh 'chmod +x /usr/local/bin/kubectl'
sh 'kubectl version --client'
}
}
```

### Node.js

```
stage('Node.js') {
steps {
sh 'rm -rf /usr/lib/node_modules/npm/'
dir ('/root/.cache/downloads') {
sh 'wget -nc "https://coding-public-generic.pkg.coding.net/public/downloads/node-linux-x64.tar.xz?version=v16.13.0" -O node-v16.13.0-linux-x64.tar.xz | true'
sh 'tar -xf node-v16.13.0-linux-x64.tar.xz -C /usr --strip-components 1'
// sh 'wget -nc "https://coding-public-generic.pkg.coding.net/public/downloads/node-linux-x64.tar.xz?version=v14.18.2" -O node-v14.18.2-linux-x64.tar.xz | true'
// sh 'tar -xf node-v14.18.2-linux-x64.tar.xz -C /usr --strip-components 1'
// 更多版本：v12.22.7、v17.2.0
}
sh 'node -v'
}
}
```

### PHP

```
pipeline {
agent {
docker {
reuseNode 'true'
registryUrl 'https://coding-public-docker.pkg.coding.net'
image 'public/docker/php:8.0'
// image 'public/docker/php:7.4' 以及 7.3、7.2、7.1、5.6
args '-v /var/run/docker.sock:/var/run/docker.sock -v /usr/bin/docker:/usr/bin/docker'
}
}
stages {
stage('安装依赖') {
steps {
// Possible values for ext-name:
// bcmath bz2 calendar ctype curl dba dom enchant exif fileinfo filter ftp gd gettext gmp
// hash iconv imap interbase intl json ldap mbstring mysqli oci8 odbc opcache pcntl pdo
```

```
// pdo_dblib pdo_firebird pdo_mysql pdo_oci pdo_odbc pdo_pgsql pdo_sqlite pgsql phar posix pspell
// readline recode reflection session shmop simplexml snmp soap sockets sodium spl standard
// sysvmsg sysvsem sysvshm tidy tokenizer wddx xml xmlreader xmlrpc xmlwriter xsl zend_test zip
sh 'apt-get update && apt-get install -y libbz2-dev'
sh 'docker-php-ext-install bz2'
sh 'php -i | grep bz2'
}
}
}
}
```

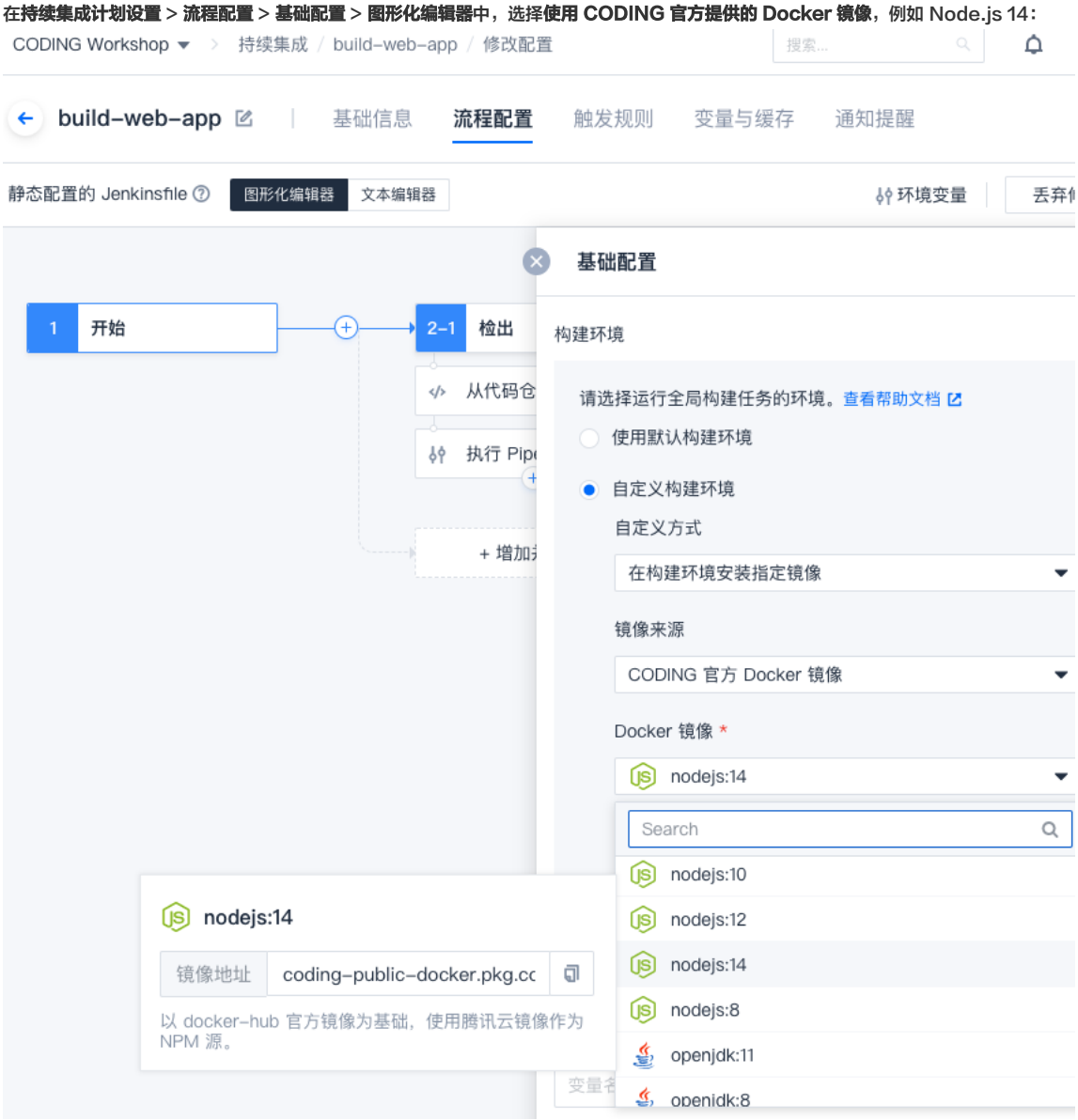
## Docker 环境

CODING 持续集成为您提供了默认构建环境，若默认环境中预装的 SDK 版本和命令行工具无法满足您的要求，还可以通过在持续集成中使用 Docker 构建环境来解决。您可以通过以下方式使用 Docker 构建环境：

- CODING 官方提供的镜像
- 使用已托管至项目级制品库的 Docker 镜像
  适用于项目层级的标准构建环境，保障项目内镜像安全，方便管理，通过项目令牌您也可以拉取其他项目的镜像。
- 使用指定 Registry 地址（默认为 Docker Hub）的 Docker 镜像
- 使用 Dockerfile 脚本构建环境

### CODING Docker 镜像

在**持续集成计划设置 > 流程配置 > 基础配置 > 图形化编辑器中，选择使用 CODING 官方提供的 Docker 镜像**，例如 Node.js 14：




**对应的 Jenkinsfile 参考：**

```
pipeline {
agent {
docker {
reuseNode 'true'
registryUrl 'https://coding-public-docker.pkg.coding.net'
image 'public/docker/nodejs:14'
}
}
stages {
stage('Test') {
steps {
sh 'node --version'
}
}
}
}
```

**项目制品库 Docker 镜像**

本章节以装有进程管理工具 pm2 的 node.js 12 环境为例，分步骤演示如何将**自建镜像推送至制品仓库** > **使用自建镜像**作为构建环境。

**步骤一：构建 Docker 镜像**

1. 新建目录，创建 Dockerfile 如下：

```
# 指定 node.js 版本为 node 12，默认从 Docker Hub 上拉取
FROM node:12

# 安装 pm2
RUN npm install pm2 -g
COPY . .

# 设置容器启动时的命令
CMD [ "pm2-runtime", "start" ]
```

2. 运行指令 docker build -t pm2-test .；-t 指定镜像名称；

```
Step 1/4 : FROM node:12
...
Step 2/4 : RUN npm install pm2 -g
...
Step 3/4 : COPY . .
...
Step 4/4 : CMD [ "pm2-runtime", "start" ]
---> Running in 46cc5081cb4f
Removing intermediate container 46cc5081cb4f
---> 5f8335fa91d4
Successfully built 5f8335fa91d4
Successfully tagged pm2-test:latest
```

**步骤二：推送镜像到 CODING 制品库**

1. 进入 CODING 制品库，选择已有制品库或新建制品库，输入密码后点击**生成个人令牌访问令牌生成配置**。


2. 复制后在终端输入命令进行登录。
3. 按照操作指引提示，为本地镜像打标签。

```
docker tag pm2-test ******/test-dd/test/pm2-test
```

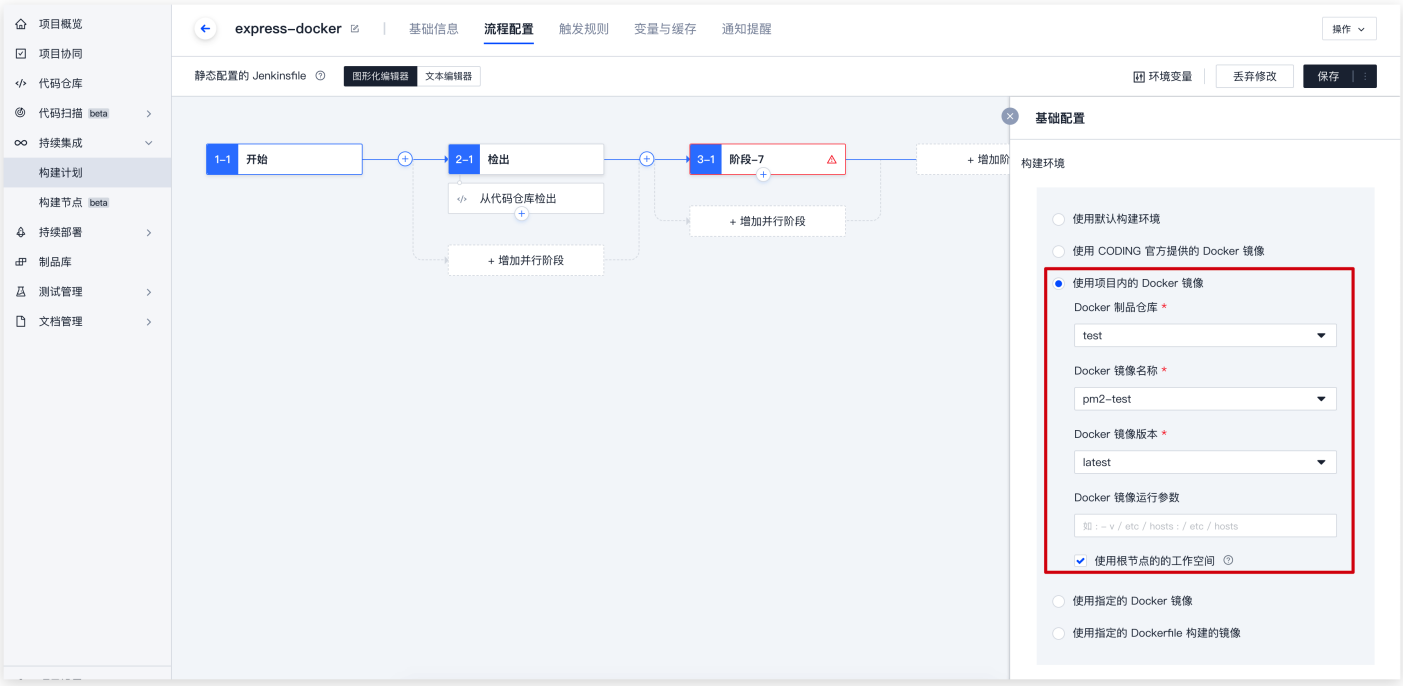
4. 推送您的 docker 镜像到 CODING 制品库

```
docker push ******/test-dd/test/pm2-test
The push refers to repository [*****************/test-dd/test/pm2-test]
809e73e276b8: Pushed
9159d4abedcd: Pushed
...
latest: digest: sha256:ccecda5071e60593d1be44ea27d4ec5b35f6a5f6872fb9 size: 2634
```

5. 推送成功后可以在镜像列表找到您的镜像。

**步骤三：在持续集成中使用镜像作为构建环境**

进入**持续集成设置 > 流程配置**，选择**使用项目内的 Docker 镜像**，选择对应制品库和镜像。




**指定地址的 Docker 镜像**

**Docker 镜像**为必填项，需要填入您的镜像名称。**Registry 地址**需填写的格式为不带路径的 URL 地址，例如：

正确：`https://codes-farm-docker.pkg.coding.net`

错误：`https://codes-farm-docker.pkg.coding.net/laravel-demo/laravel-docker/`


**若拉取私有镜像，需 录入凭据 后填写 Registry 认证凭据 ID。**

**对应的 Jenkinsfile：**

```
pipeline {
agent {
docker {
image 'node:14-alpine'
reuseNode 'true'
}
}
stages {
stage('Test') {
steps {
sh 'node --version'
}
}
}
}
```

**Dockerfile 构建环境**

若项目已经使用 Docker，建议将 Dockerfile 提交到代码库，用它作为持续集成构建环境。Dockerfile 示例代码：

```
FROM php:8.0-apache
RUN apt-get update \
&& apt-get install -y unzip
```

Jenkinsfile：

```
pipeline {
agent any
stages {
stage('Checkout') {
steps {
checkout([
$class: 'GitSCM',
branches: [[name: env.GIT_BUILD_REF]],
userRemoteConfigs: [[url: env.GIT_REPO_URL, credentialsId: env.CREDENTIALS_ID]]
])
}
}
stage('Use Docker') {
agent {
dockerfile {
filename 'Dockerfile' // 可选，自定义 Dockerfile 文件名
dir 'build' // 可选，Dockerfile 所在目录
additionalBuildArgs '--build-arg version=1.0.2' // 可选，docker build 自定义参数
}
}
stages {
stage('Test') {
steps {
sh 'php -v'
sh 'unzip -v'
}
}
}
}
}
}
```

若构建次数频繁，而不想将时间浪费在 docker build 过程上，那么可以通过使用 Jenkins Dockerfile 保存镜像用于下次构建，从而节省大量时间，请参见 保存 Dockerfile 镜像。

### 在阶段中使用 Docker

Jenkinsfile 参考：

```
pipeline {
agent none
stages {
stage('Back-end') {
agent {
docker {
image 'maven:3-alpine'
reuseNode 'true'
}
}
```

```
steps {
sh 'mvn --version'
}
}
stage('Front-end') {
agent {
docker {
image 'node:14-alpine'
reuseNode 'true'
}
}
steps {
sh 'node --version'
}
}
}
}
```

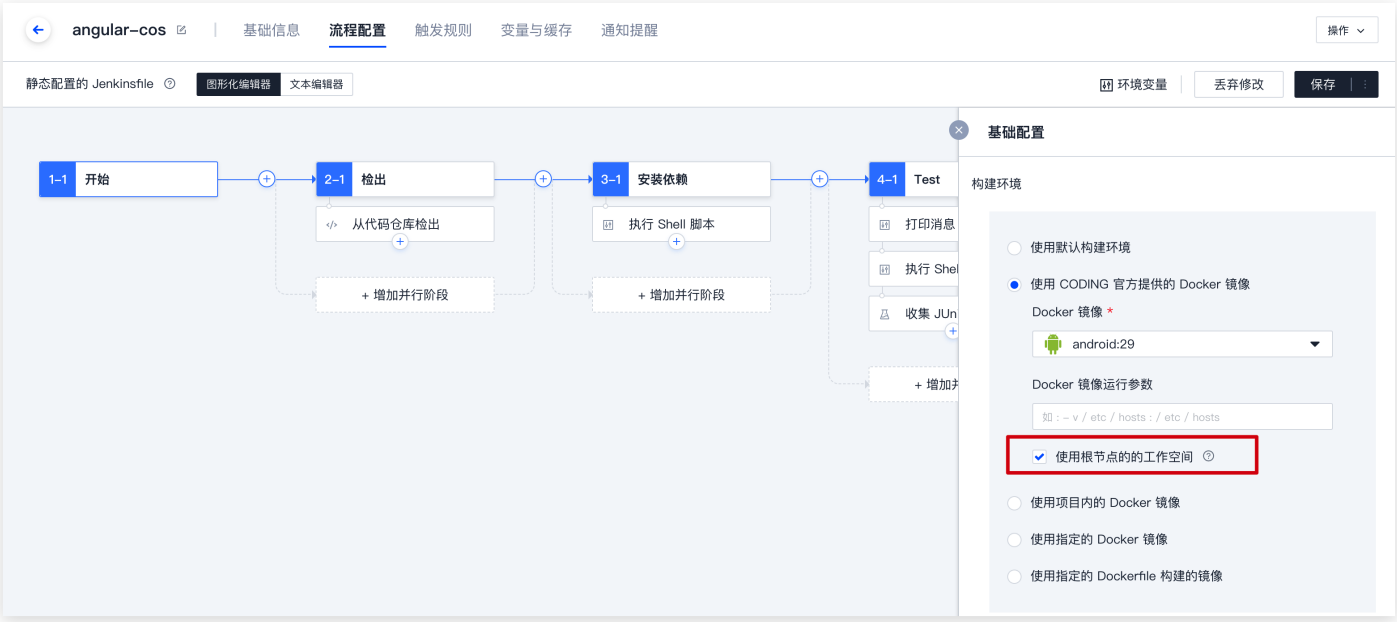
### 多个 Docker 后台

自动化测试往往需要临时的基础设施（例如 MySQL、Redis、Elasticsearch）。那么创建一个桥接网络，在其中启动多个 Docker 后台，测试完毕自动删除。

```
node {
stage("检出") {
checkout([
$class: 'GitSCM',
branches: [[name: GIT_BUILD_REF]],
userRemoteConfigs: [[
url: GIT_REPO_URL,
credentialsId: CREDENTIALS_ID
]]])
}
stage('准备数据库') {
sh 'docker network create bridge1'
sh(script:'docker run --net bridge1 --name mysql -d -e "MYSQL_ROOT_PASSWORD=my-secret-pw" -e "MYSQL_DATABASE=test_db" mysql:5.7', retu
rnStdout: true)
sh(script:'docker run --net bridge1 --name redis -d redis:5', returnStdout: true)
}
docker.image('ecoding/php:8.0').inside("--net bridge1 -v \"${env.WORKSPACE}:/root/code\" -e 'APP_ENV=testing' -e 'DB_DATABASE=test_db'" +
" -e 'DB_USERNAME=root' -e 'DB_PASSWORD=my-secret-pw' -e 'DB_HOST=mysql' -e 'REDIS_HOST=redis'" +
" -e 'APP_KEY=base64:tbgOBtYci7i7cdx5RiFE3KZzUkRtJfbU3Ibj5uPdL8U='") {
sh 'composer install'

stage('单元测试') {
sh 'XDEBUG_MODE=coverage ./vendor/bin/phpunit --coverage-html storage/reports/tests/ --log-junit storage/test-results/junit.xml --coverage-text
tests/'
junit 'storage/test-results/junit.xml'
codingHtmlReport(name: '测试覆盖率报告', path: 'storage/reports/tests/')
}
}
}
```

### 根节点工作空间

将自定义 Docker 用作构建环境时，可以选择是否使用根节点的工作空间。勾选该选项后，当前阶段的 Docker 容器会和流水线在同一台构建节点中运行，可以获取流水线工作空间下根目录保存的所有文件。




**对应的 Jenkinsfile 参数为 reuseNode ，类型：Boolean，默认为 false：**

```
pipeline {
agent {
docker {
registryUrl 'https://coding-public-docker.pkg.coding.net'
image 'public/docker/android:29'
}
}
stages {
// 代码被检出到 pipeline agent 的工作空间根目录下
stage('检出代码') {
steps {
checkout([
$class: 'GitSCM',
branches: [[name: env.GIT_BUILD_REF]],
userRemoteConfigs: [[url: env.GIT_REPO_URL, credentialsId: env.CREDENTIALS_ID]]
])
}
}
stage('单元测试') {
agent {
dockerfile {
// 默认在当前节点工作空间根目录下找名为 「Dockerfile」的文件构建环境
filename 'Dockerfile'
// 如果 reuseNode 为 false，则无法找到之前检出到 pipeline agent 的工作空间根目录下的 Dockerfile
reuseNode true
}
}
steps {
sh 'npm run test:ci'
junit '*.xml'

}
}
}
}
```

### 执行 Docker 命令

在 Jenkins Docker 环境中执行 docker 命令时，需要挂载外部云服务器的 docker socket，否则会报错： docker: command not found

```
pipeline {
agent {
docker {
image 'ecoding/php:8.0'
reuseNode 'true'
// 挂载外部云服务器的 docker socket
args '-v /var/run/docker.sock:/var/run/docker.sock -v /usr/bin/docker:/usr/bin/docker'
}
}
stages {
stage('自定义阶段') {
steps {
sh 'php -v'
}
}
stage('构建 Docker 镜像') {
steps {
sh 'docker -v'
script {
docker.withRegistry("https://${env.CCI_CURRENT_TEAM}-docker.pkg.coding.net", "${env.CODING_ARTIFACTS_CREDENTIALS_ID}") {
//docker.build("foo:bar").push()
}
}
}
}
}
}
```

若希望在自定义构建节点中运行 Docker 命令，在节点中先行安装 Docker 服务即可开始使用。

## 参考资料

在流水线中使用 Docker 的 agent 语法说明

# 自定义节点

最近更新时间：2023-08-01 16:34:23

本文为您介绍如何使用自定义节点。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

## 功能介绍

在实际的开发项目中，所涉及的开发环境可能是多种多样的。当默认节点的构建环境无法承载项目的运行要求时，例如需使用 macOS Xcode 构建 iOS 应用，那么就可以通过接入自定义类型节点（物理机/云服务器/容器等）运行特定任务。
接入构建节点时需指定接入的构建节点池，构建节点不能游离节点池而单独存在。构建节点池 分为团队与项目两种类型。

> 注意：
> 使用自定义节点相当于将您的构建机纳入 CODING 的管控范围。为了保障您的机器和内网的安全，请务必妥善保管您的用户名和密码，并建议开启登录二次认证。在必要时，您还需要制定构建机内网的安全策略，以确保您的机器和内网不受到未经授权的访问和攻击。

## 接入节点

接入自定义节点的过程本质上是在节点中运行 Worker 服务，单击了解 Worker 常用命令。
目前支持 macOS、Windows、Linux 环境接入至构建计划节点池。

### 推荐配置

- CPU 8 核或以上
- 内存 16 GB 或以上

### 环境依赖

- **Python 3.6, Python 3.7, Python 3.8, Python 3.9**
  单击访问 项目的地址 进行下载。
- **Git ⩾ 2.8**
  单击访问 项目的地址 进行下载。
- **Java 8, Java 11**
  单击访问 项目的地址 进行下载。
- **Jenkins**

jenkins.war 文件下载命令：

```
curl -fL "https://coding-public-generic.pkg.coding.net/cci/release/cci-agent/jenkins.war?version=2.293-cci" -o jenkins.war
```

jenkins_home.zip 文件下载命令：

```
curl -fL "https://coding-public-generic.pkg.coding.net/cci/release/jenkinsHome.zip?version=latest" -o jenkins_home.zip
```

**Windows:** 进入 C:/ 目录，创建 codingci/tools 目录。在其中下载 jenkins.war 、jenkins_home.zip 文件，并在 tools/ 目录解压 jenkins_home.zip 文件。
**Linux:** 进入 /root/ 目录，创建 codingci/tools 目录。在其中下载 jenkins.war 、jenkins_home.zip 文件，并在 tools/ 目录解压 jenkins_home.zip 文件。
**macOS:** 进入 ~/ 目录，创建 codingci/tools 目录。在其中下载 jenkins.war 、jenkins_home.zip 文件，并在 tools/ 目录解压 jenkins_home.zip 文件。

# macOS

## 命令接入

1. 进入构建节点，选择**接入新节点 > macOS**，接入方式选择 Bash，在接入配置中选择对应节点池，单击**生成接入配置并复制**。


2. 在终端中输入命令后，等待服务下载完成。安装完成后可以使用以下命令进行验证：

```
qci_worker version
```

命令接入的默认安装目录为 /root/codingci，若希望指定安装目录，请参见 指定安装目录。

## 手动接入

使用手动接入方式前请确保节点 已满足上文中的**环境依赖**要求。

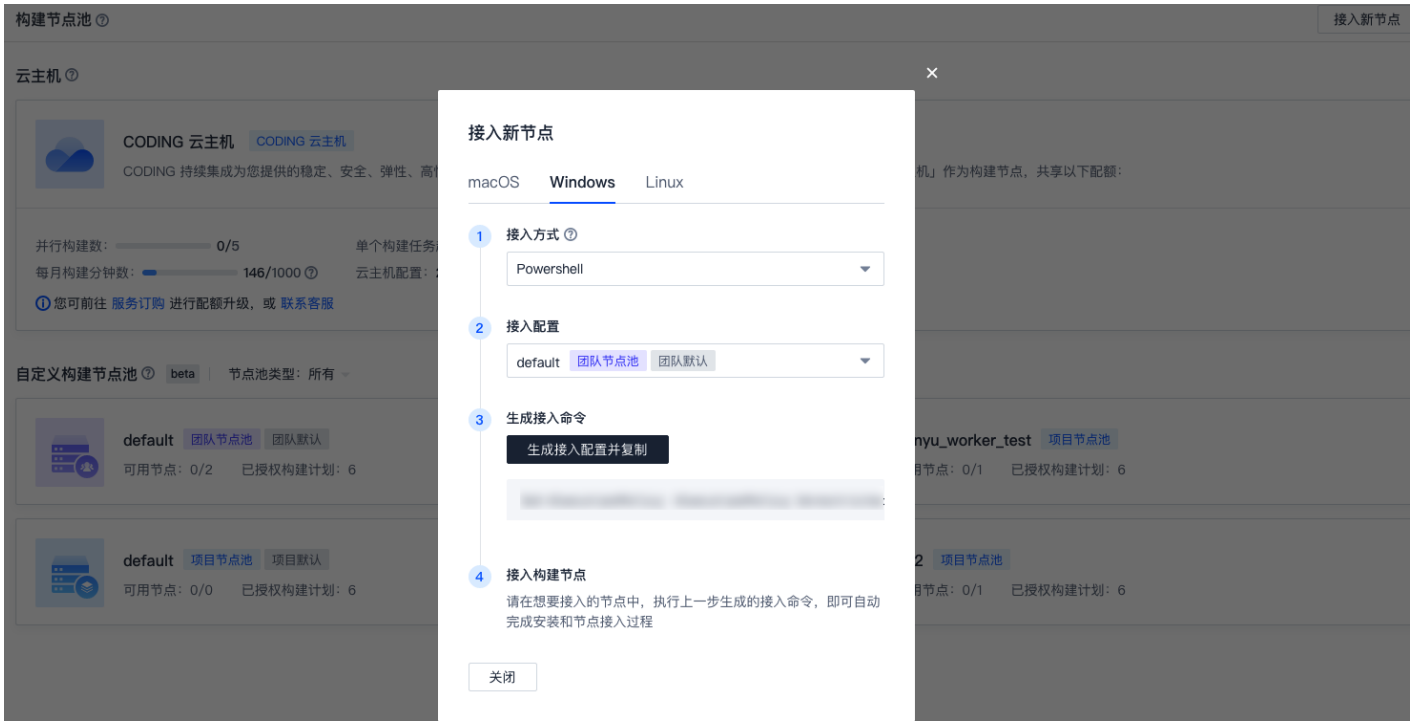
1. 接入方式选择手动接入，按照提示在终端中输入命令安装客户端，即安装 Worker 服务。


2. 选择拟接入的节点池，单击**一键生成并复制**，生成初始化命令。
3. 在终端中执行已自动生成的客户端启动命令，让构建节点保持在线状态。

## Windows

### 命令接入

1. 选择**持续集成** > **构建节点**中，单击右上角的**接入新节点**，选中 Windows，选择 Powershell 接入方式。




2. 安装完成后可以使用以下命令进行验证：

```
qci_worker version
```

**命令接入的默认安装目录为 /root/codingci，若希望指定安装目录，请参见 指定安装目录。**

### 手动接入

**使用手动接入方式前请确保节点 已满足上文中的环境依赖要求。**

**1. 接入方式选择手动接入，按照提示在终端中输入命令安装客户端，即安装 Worker 服务。**


**2. 选择拟接入的节点池，单击一键生成并复制，生成初始化命令。**

**3. 在终端中执行已自动生成的客户端启动命令，让构建节点保持在线状态。**

## Linux

### 命令接入

1. 选择**持续集成 > 构建节点**中，单击右上角的**接入新节点**，选中 Linux，选择 Bash 接入方式。设置完成后在 Linux 环境中运行已生成的接入配置命令。


2. 安装完成后可以使用以下命令进行验证：

```
qci_worker version
```

命令接入的默认安装目录为 /root/codingci，若希望指定安装目录，请参见 指定安装目录。

### 手动接入

使用手动接入方式前请确保节点 已满足上文中的环境依赖要求。

1. 接入方式选择手动接入，按照提示在终端中输入命令安装客户端，即安装 Worker 服务。


2. 选择拟接入的节点池，单击**一键生成并复制**，生成初始化命令。
3. 在终端中执行已自动生成的客户端启动命令，让构建节点保持在线状态。

## 启动守护进程

安装完成后，在构建节点上需要运行守护进程，用以监听并获取由 CODING 后台下发的 CI 任务。以下为运行 和 删除命令行：

```
# 在后台运行
qci_worker up -d

# 在前台运行
qci_worker up

# 暂时停止运行
qci_worker stop
```

## 指定安装目录

若您需要将 worker 服务安装至指定目录中，需使用手动接入方式。例如安装至 /opt/qci-workspace 目录，您需要进行以下操作：

### 步骤1. 下载 Jenkins 服务

Linux: 创建 /opt/qci-workspace/tools 目录。在其中下载 jenkins.war、jenkins_home.zip 文件，并在 tools/ 目录解压 jenkins_home.zip 文件。

- jenkins.war 文件下载命令

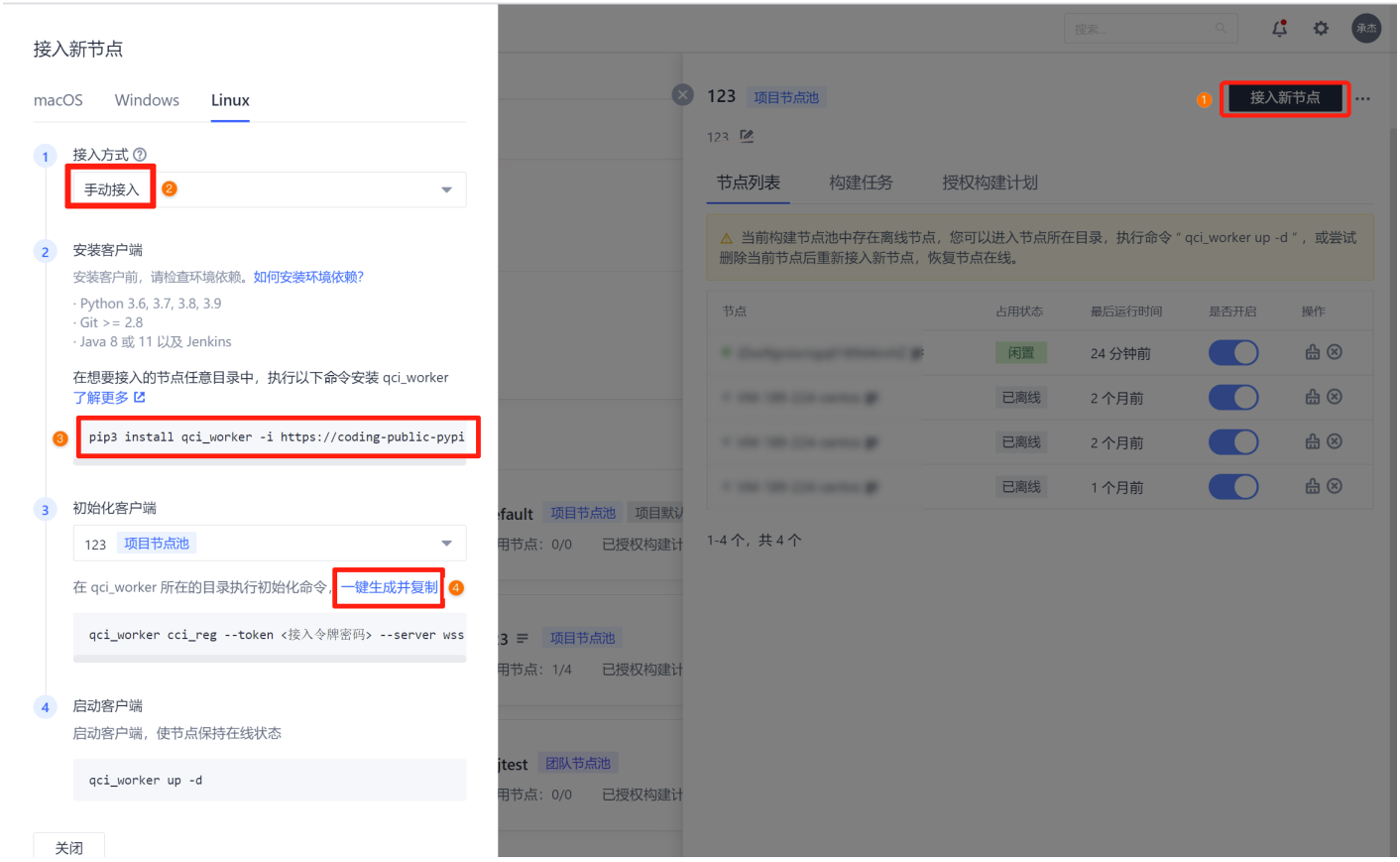
```
curl -fL "https://coding-public-generic.pkg.coding.net/cci/release/cci-agent/jenkins.war?version=2.293-cci" -o jenkins.war
```

- jenkins_home.zip 文件下载命令

```
curl -fL "https://coding-public-generic.pkg.coding.net/cci/release/jenkinsHome.zip?version=latest" -o jenkins_home.zip
```

### 步骤2. 手动接入并注册

1. 按照下图路径获取安装命令，在终端中运行命令，安装 worker 客户端。




2. 复制上图第三步的**初始化客户端**命令后，在末尾添加 --home /opt/qci-workspace 参数用以指定安装目录。

```
root@i  cci_reg --token  --server wss://jiyunkeji.coding.net --home /opt/qci-workspace
token:  a440
host input: jiyunkeji.coding.net
host: jiyunkeji.coding.net
home: /opt/qci-workspace
in docker: False
auto: False
https://jiyunkeji.coding.net/api/ci/machine/register
{"Fingerprint": "220330  30163e14cd6f", "ClientId": "", "Token": "86cee3983  58ea440", "AgentName": "iZwz9gv  vhZ", "AgentPoolName": "defau
, "AgentVersion": "1.  ", "AgentVersionDetail": "1.  ", "JenkinsHomeVersion": "v15", "JenkinsVersion": "2.  ", "JenkinsWorkspace": "/opt/qci-workspace/tools/jenkins_home/workspace",
sAutoRegister": false, "NodeStat": {"Home": "/opt/qci-workspace", "Ipv4": ["172.  ", "17  "], "Os": {"Hostname": "iZwz9g  ", "Os": "Linux", "BitSize": 64, "Dis
bution": "", "DistributionVersion": "", "Arch": "x86_64", "KernelVersion": "Linux-4.15.0-147-generic-x86_64-with-Ubuntu-18.04-bionic"}, "Cpu": {"PhysicalCpuCore": 1, "LogicalCpuCore": 1, "
elName": ""}, "Memory": {"Total": 2  , "RamTotal": 2  , "RamUsed": 17  , "RamFree": 6  , "RamAvailable": 1  4, "SwapTotal": 0, "SwapFree": 0}, "Disk": {"Tota
 21  , "Free": 15  , "Used": 18  }}}
200
```

### 步骤3. 启动服务

**运行命令：**

```
qci_worker up -d
```

# Worker 常用命令

最近更新时间：2021-12-31 14:17:05

本文为您介绍 qci-worker 服务的常用命令。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

## 功能介绍

接入自定义构建节点时将在环境中安装 Worker 服务，并由此服务调度 CI 构建任务的下发与计算资源的分配。因此熟悉 Worker 服务的常用命令能够更好地配合 CI 构建任务，不同操作系统下的安装方法请参见 自定义节点。

## 常用配置项命令

### 注册

```
qci_worker reg_cci --token token --server server --home home
--token 项目令牌，必填
--server 指定的接入服务，非必填
--home 指定工作目录
```

示例：

```
qci_worker cci_reg --token db6fd4d6a2****************85d55c44a2262f3e543f --server ws://codingcorp.nh113vufq.dev.coding.io --home ~/.codin
gqci
```

### 启动服务

```
qci_worker up -d
```

### 重启服务

```
qci_worker stop
qci_worker up -d
```

### 手动删除节点

```
qci_worker stop #停止 qci_worker
qci_worker remove # 后台删除节点
```

## 修改配置

若需要让指定的 Jenkins 配置项生效，需要先停止 Jenkins 服务进程，然后重启 qci_worker 服务。

```
qci_worker config JENKINS_HOST=127.0.0.1 # 指定 Jenkins 启动 host
qci_worker config JENKINS_PORT=15740 # 指定 Jenkins 启动 port
```

# 构建节点池

最近更新时间：2021-12-31 14:17:10

本文为您介绍构建节点池。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

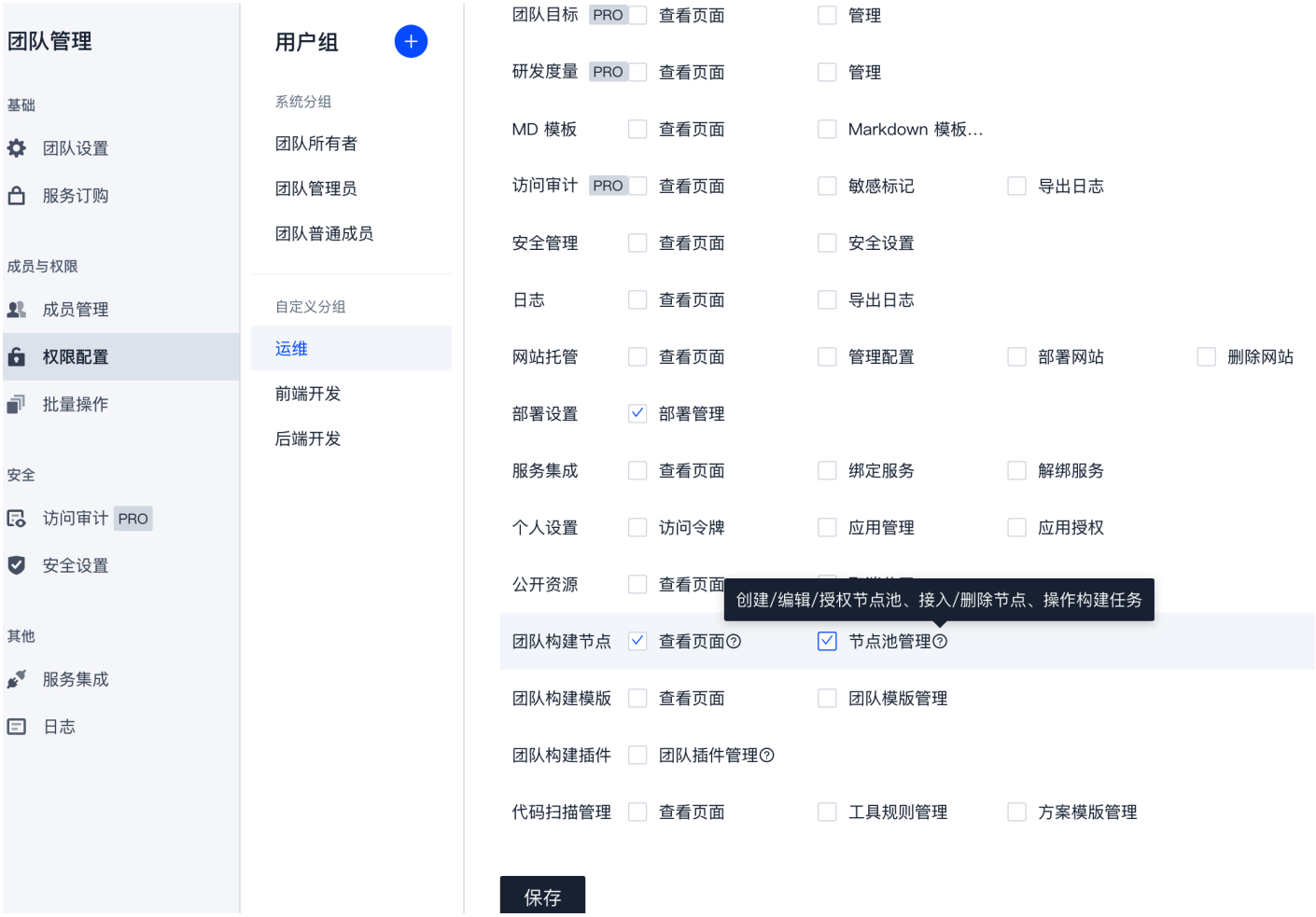
1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

## 功能介绍

构建节点池是构建节点的集合，在使用自定义的构建节点时，需要将构建节点接入构建节点池，并通过将构建计划节点池配置来指定构建节点池。


## 权限控制

1. 用户组需具备**团队构建节点**权限才能进行创建和删除构建节点池等操作。单击右上角头像下拉处的**团队管理** > **权限配置**为用户组勾选开启相应的权限。




2. 单个项目内支持设置多个构建节点池，每个构建节点池支持接入多个构建节点。在构建节点池详情中的节点列表可以查看节点状态并对其进行管理。

## 节点状态

- 闲置：构建节点此时空闲。
- 占用：构建节点已被分配到构建任务中使用。
- 准备中：构建节点正在准备构建环境。
- 开启：只有处于开启状态的节点才能被分配使用，如果关闭节点不会影响正在运行的构建任务。
- 删除：节点将会脱离 CODING 持续集成服务，但只会删除工作空间和相关的配置信息，之前产生的全局缓存文件仍会保留。

- 构建节点池详情内可以查看节点的构建记录。


1. 构建节点池默认授权给所有构建计划，您也可以选择只授权给指定的构建计划（支持多选）。

2. 在构建计划的基础信息设置中可以修改相应的节点池配置。构建计划默认使用 CODING 提供的云服务器，您也可以选择其它项目内配置的节点池进行构建。

# 管理构建计划
# 分组管理

最近更新时间：2022-01-04 09:44:13

本文为您介绍如何分组管理构建计划。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

## 星标与分组

构建计划还支持星标与分组，可以帮助快速定位到自己关注的构建计划。

**星标功能**

个人选项，设置后仅对个人生效。单击单个构建计划区域中的星标按钮后，可以在**我的星标** tab 栏中仅查看星标构建计划。


**分组功能**

全局选项，仅开放给**持续集成管理**权限的用户。设置后的构建计划分组归类对项目内成员可见，方便项目内构建计划的整理。

1. 单击**更多** > **创建分组**可以创建分组，单击后输入分组名即可创建。


还可以修改分组名称、排序，也可以创建和删除分组。

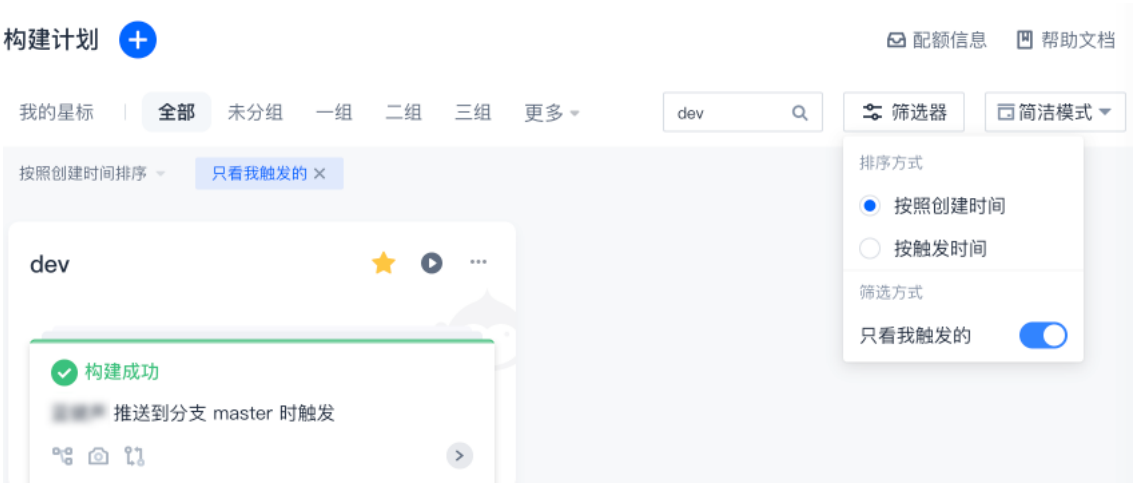
> 注意：

删除分组不会删除分组中的构建计划，分组删除后，分组内的构建计划将会被归类到**未分组**类别。


2. 单击**批量整理构建计划**可以进入构建计划整理页面，可以一次性选择多个构建计划至同一个分组当中。添加完成后就可以在单独的分组 tab 页中看到勾选的构建计划。


## 筛选与排序

在构建计划页面右侧的搜索栏中可以根据构建计划名称进行筛选。选择**筛选器 > 只看我触发的**，构建计划中将会显示由用户本人最新一次触发的构建记录，并且在构建记录展示页，也可以开启**筛选器**。




除此之外，还可以按照构建计划最新构建记录的触发时间排序。

# 构建计划模板

最近更新时间：2022-01-04 10:25:56

本文为您介绍如何设置统一的团队构建计划模板。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

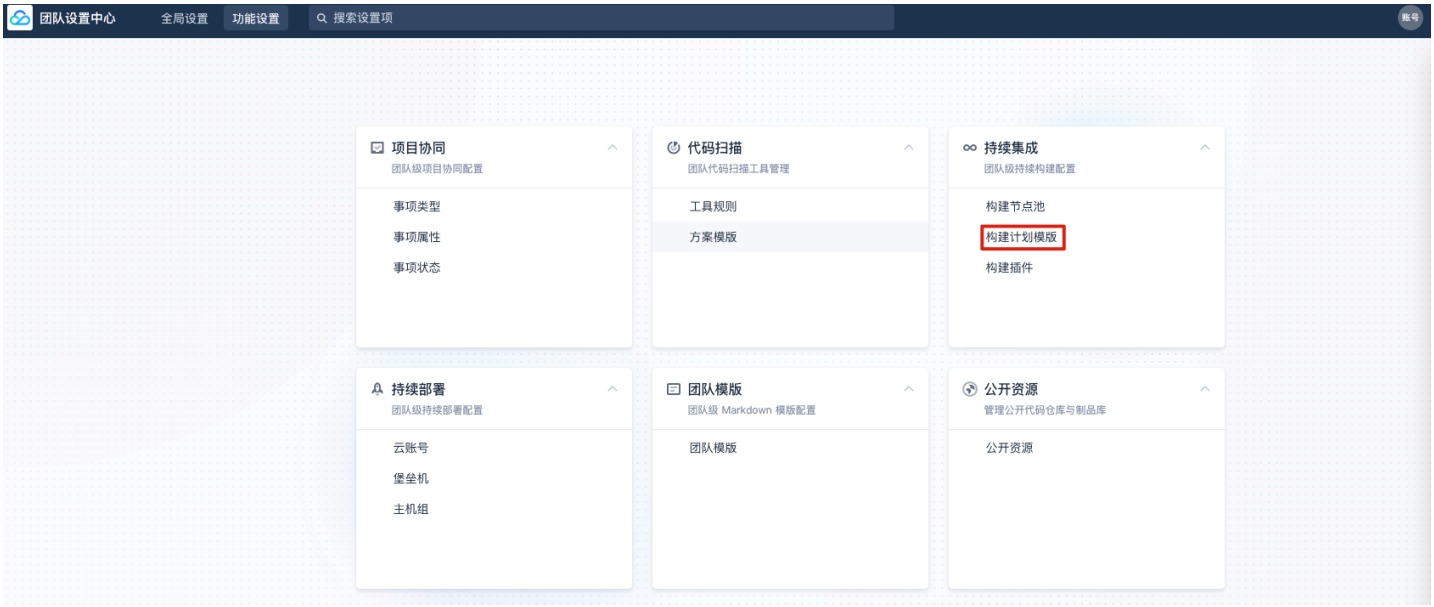
1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

## 功能介绍

CODING 持续集成支持编制统一的**构建计划模板**。一次配置，就能在团队内的跨项目协作中复用统一而规范的构建计划模板，优化团队成员配置构建流程的效率，集中管理团队中通用的构建计划。

## 新建构建计划模板

单击团队首页右上角的齿轮图标 进入团队设置中心，您可以在**功能设置** > **构建计划模板**中新建团队构建模板。




## 编辑构建计划模板

在模板内能够编辑流程配置、基础配置、触发规则以及变量与缓存。

### 流程配置

您可以使用**图形化编辑器**或**文本编辑器**编写该构建模板的执行流程。图形化编辑器的优势在于能够获得边看边写的可视化体验，在图形化编辑器中增删的所有步骤都可以转换成文本，但反之则不一定能够兼容。


**基础配置**

在基础配置页能够更改模板名称、图标。单击右侧的操作下拉菜单能够执行删除模板或 同步模板。


当模板有更新时，模板作者可以通过 同步模板 操作将更新同步至所有使用该模板创建的构建计划。该操作将覆盖对应构建计划的配置，单击查看 场景举例。

**触发规则**

支持代码源触发、定时触发及手动触发。与普通构建计划下的触发规则设置方式一致，具体说明请参见 触发规则。

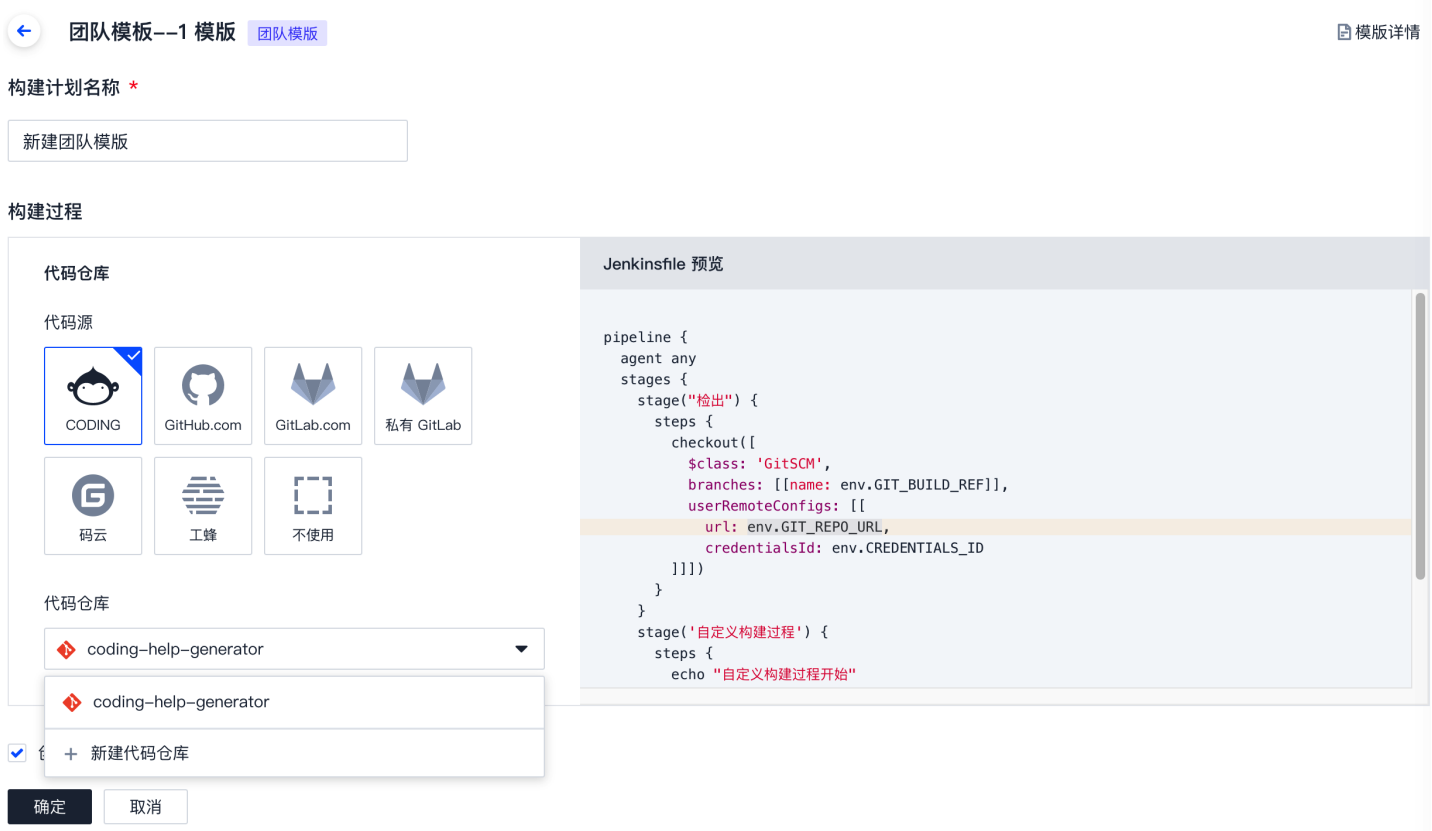
**变量与缓存**

您可以在此处添加构建计划的环境变量。在手动启动构建任务时，环境变量也将作为启动参数的默认值，具体配置说明请参见 环境变量。

## 使用构建计划模板

模板作者完成团队构建模板的编写后，其他团队成员就可以在任一项目中使用该构建计划模板。


构建计划左上角处会标注为构建计划模板，使用者可以按照项目需求选择不同的代码源。




创建成功后，构建计划流程、触发配置、环境变量及默认值与模板保持一致，使用者可以按照项目的实际情况进行修改。基于模板做出的不会影响模板内容，如需修改模板请前往设置中心的**功能设置 > 构建计划模板**处进行更新。

## 同步构建计划模板

当团队内部使用了某项构建计划模板作为其他构建计划的基石时，修改构建计划模板后使用同步功能，就能够让其他构建计划向构建计划模板对齐。

**使用场景举例**：A 团队已在内部全面推行持续集成构建规范，大部分构建计划皆是基于某一规范性构建计划模板进行编写。随着项目推进，旧有规范亟待更新，此时模板作者

仅需完成构建计划模板的更新并使用同步功能，就能让其他构建计划与模板进行对齐。


同步功能并不会覆盖其他构建计划中的所有内容，下图为同步所导致的**变更**效果图：

| 团队构建计划模版 T1 | | |
|---|---|---|
| 变量名 | 变量值 | 备注 |
| ENV_1 | Hello | |
| ENV_2 | World | |

修改模版

| 团队构建计划模版 T1 | | |
|---|---|---|
| 变量名 | 变量值 | 备注 |
| ENV_1 | Hello | |
| ENV_2 | World | |
| ENV_4 | My | 模版新增环境变量 |

使用模版创建

| 构建计划-A (使用团队模版 T1 创建) | | |
|---|---|---|
| 变量名 | 变量值 | 说明 |
| ENV_1 | Hello | |
| ENV_2 | World | |
| ENV_3 | MyCustom | 比原有模版多一个值 |

通过模版同步

| 构建计划-A (使用团队模版 T1 创建) | | |
|---|---|---|
| 变量名 | 变量值 | 说明 |
| ENV_1 | Hello | |
| ENV_2 | World | |
| ENV_3 | MyCustom | 同步后保留 |
| ENV_4 | My | 通过模版同步新增的环境变量 |

| 构建计划-B (使用团队模版 T1 创建) | | |
|---|---|---|
| 变量名 | 变量值 | 说明 |
| ENV_1 | Hello | |
| ENV_2 | World2 | 与模版的默认值不同 |

| 构建计划-B (使用团队模版 T1 创建) | | |
|---|---|---|
| 变量名 | 变量值 | 说明 |
| ENV_1 | Hello | |
| ENV_2 | World | 同步后值将被模版的覆盖 |
| ENV_4 | My | 通过模版同步新增的环境变量 |

> ⚠ 注意：
> 使用同步功能前请确保已知晓该操作可能会造成的影响。

# 系统插件
# 上传 Generic 类型制品

最近更新时间：2022-03-25 15:13:53

本文为您介绍如何使用持续集成插件上传 Generic 类型制品。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

## 功能介绍

在实际的生产环境中，有着许多重复性较强的工作。CODING 持续集成具备插件功能，它能够帮助您高效处理繁琐重复的工作，并且还可以通过自定义参数来解决个性化需求。CODING 更多的内置插件将在后续版本中支持。目前 CODING 的持续集成已支持以下便捷的插件：

- 上传 Generic 类型制品
- 调取已上传的凭据
- 在合并请求中自动添加评审者
- 人工确认

## 使用插件上传 Generic 类型制品

在 CODING 持续集成任务构建过程当中，您可以选择将构建物上传至 CODING 制品库。Generic 制品上传插件能够让您更方便地在持续集成当中上传 Generic 类型构建物，目前单文件大小最高支持5GB。在使用该功能之前，请确保您对 Generic 类型制品库有初步了解，详情请参见 使用 Generic 制品库。

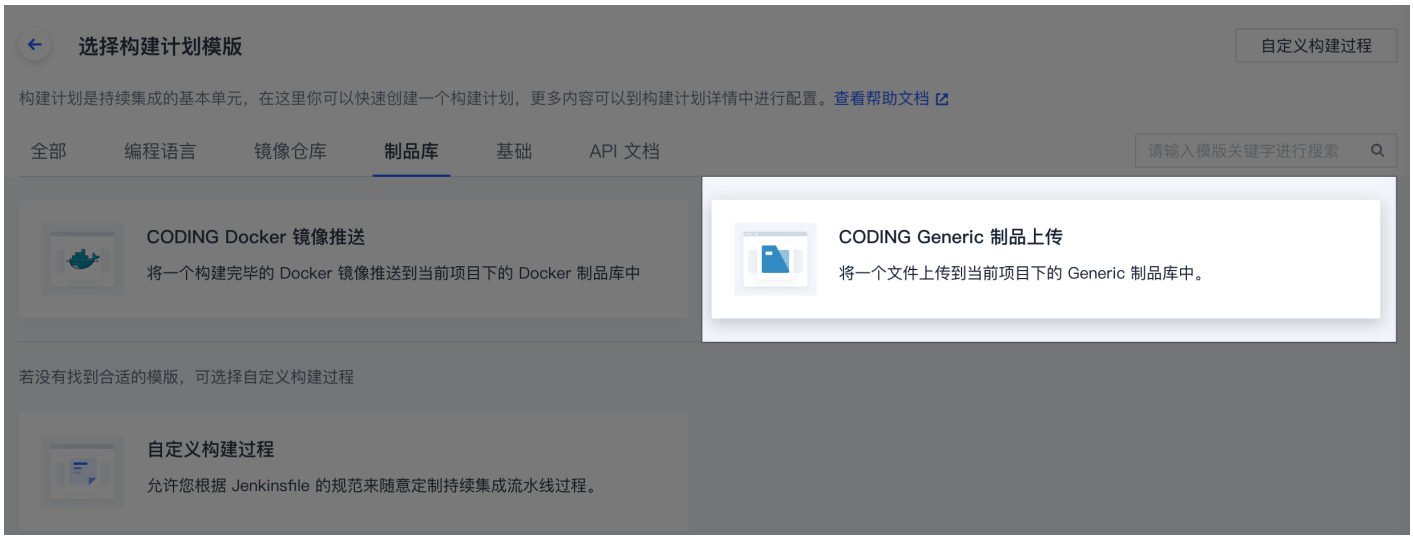
## 快速开始

您可以选择使用固定模板或 Jenkinsfile 配置两种方式来上传 Generic 类型制品。

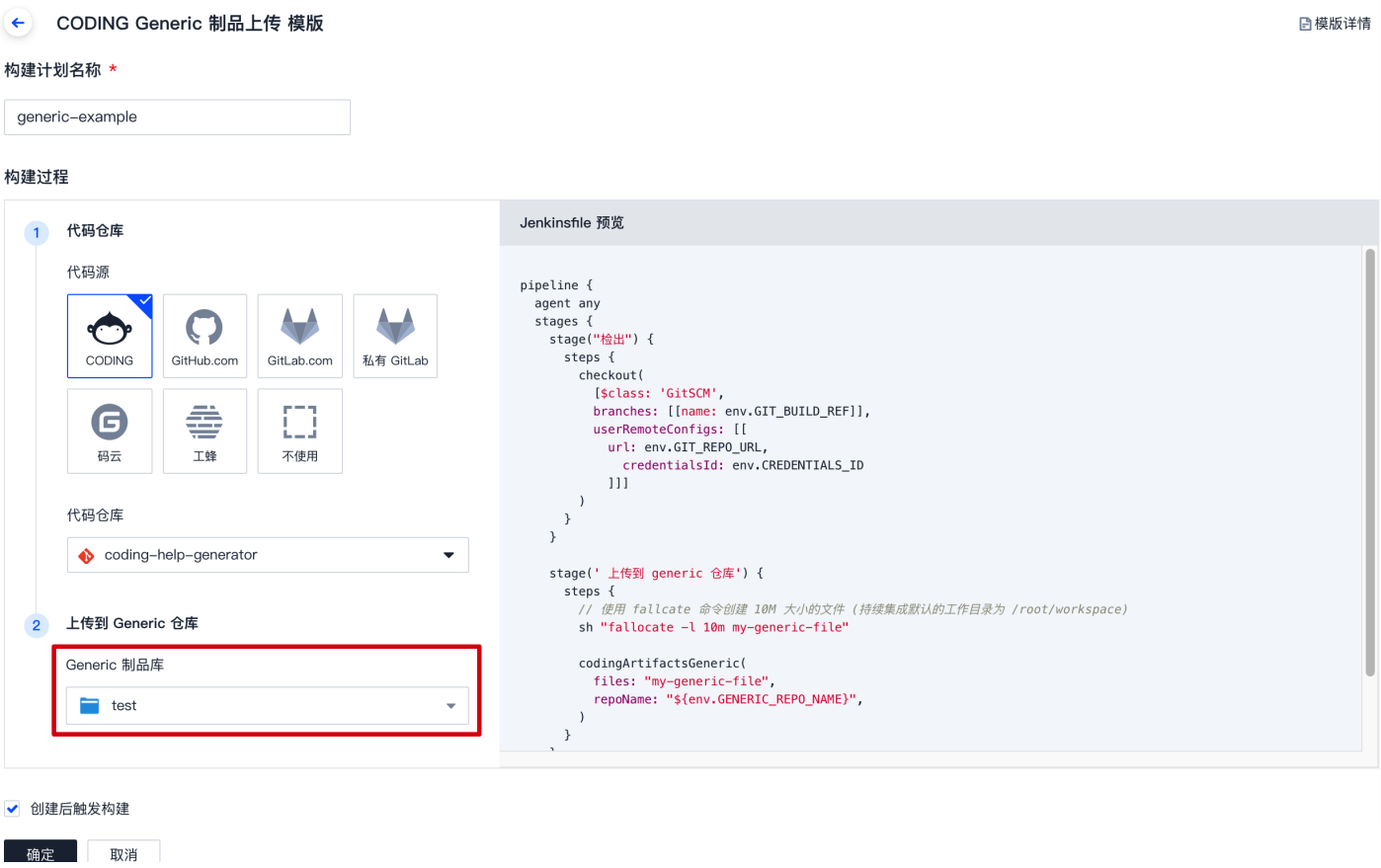
### 固定模板

1. 在使用插件上传前，请确保您已经创建了 Generic 类型制品库，下方的演示将会以 test 制品库为例。


2. 单击新建构建计划后，选择制品库中的 CODING Generic 制品上传。



3. 在默认值选中刚刚创立的 test 制品库，您也可以在代码中填写您的制品库地址。




4. 勾选创建后触发构建，单击确定后完成构建。
5. 构建完成后在制品库中会出现刚刚上传的文件。


## Jenkinsfile 配置

1. 在选择代码仓库的时候，请确保代码仓库中已含有如下配置的 Jenkinsfile 文件。

```
pipeline {
agent any
stages {
stage('上传到 generic 仓库') {
steps {
// 使用 fallcate 命令创建 10M 大小的文件 (持续集成默认的工作目录为 /root/workspace)
sh 'fallocate -l 10m my-generic-file'

codingArtifactsGeneric(
files: 'my-generic-file',
```

```
repoName: 'myrepo', //此处填写您的仓库参数，例如${env.GENERIC_REPO_NAME}
)
}
}
}
}
```

2. 在流程配置中也可以对 Jenkinsfile 配置进行修改。


## 更多参数演示

```
pipeline {
agent any
stages {
stage('上传到 generic 仓库') {
steps {
// 使用 fallcate 命令创建 10M 大小的文件（持续集成默认的工作目录为 /root/workspace)
sh 'fallocate -l 10m my-generic-file'

codingArtifactsGeneric(
files: 'my-generic-file',
repoName: 'myrepo',
)
}
}
}
}
```

## 参数说明

| 参数名称 | 必填 | 参数类型 | 图形化参数类型 | 默认值 |
| --- | --- | --- | --- | --- |
| files | 是 | - | 需要上传的文件列表，支持通配符 build/libs/**/xx | - |
| repoName | 是（若单独设置了 repoURL，则可以不填） | string | 制品库名称 | - |
| version | 否 | string | string | latest |
| zip | 否 | - | string | string |

| 参数名称 | 必填 | 参数类型 | 图形化参数类型 | 默认值 |
| --- | --- | --- | --- | --- |
| credentialsId | 否 | string | 凭据（用户名+密码） | env.CODING_ARTIFACTS_CREDENTIALS_ID |
| repoURL | 否 | string | string | https://<env.CCI_CURRENT_TEAM>-generic.<env.CCI_CURRENT_DOMAIN>/<env.PROJECT_NAME>/<params.repoName |
| withBuildProps | 否 | boolean | boolean | true |
| workspace | 否 | string | 否 | - |

# 调取已录入的凭据

最近更新时间：2022-03-25 15:13:58

本文为您介绍如何使用持续集成插件调取已录入的凭据。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

## 功能介绍

在持续部署中，有时候会需要用到第三方供应商所提供的服务，这时候就需要调用相关的凭据来获取权限。目前持续部署功能已集成内置插件，支持快速调取相关凭据。

## 使用插件快速调取已录入的凭据

在 CODING 持续集成任务构建过程当中，如果将 Github 的账号密码等重要信息硬编码在配置文件内，将会有极大的安全隐患。通过 CODING 的 凭据管理 功能，将凭据 ID 写入配置文件中进行服务调用。在接下来的插件功能使用中，请确保您已将凭据托管至 CODING 中。

### 快速开始

下面以调取凭据管理中的云 API 密钥为例，演示如何使用 Jenkinsfile 配置快速调取已录入的凭据。

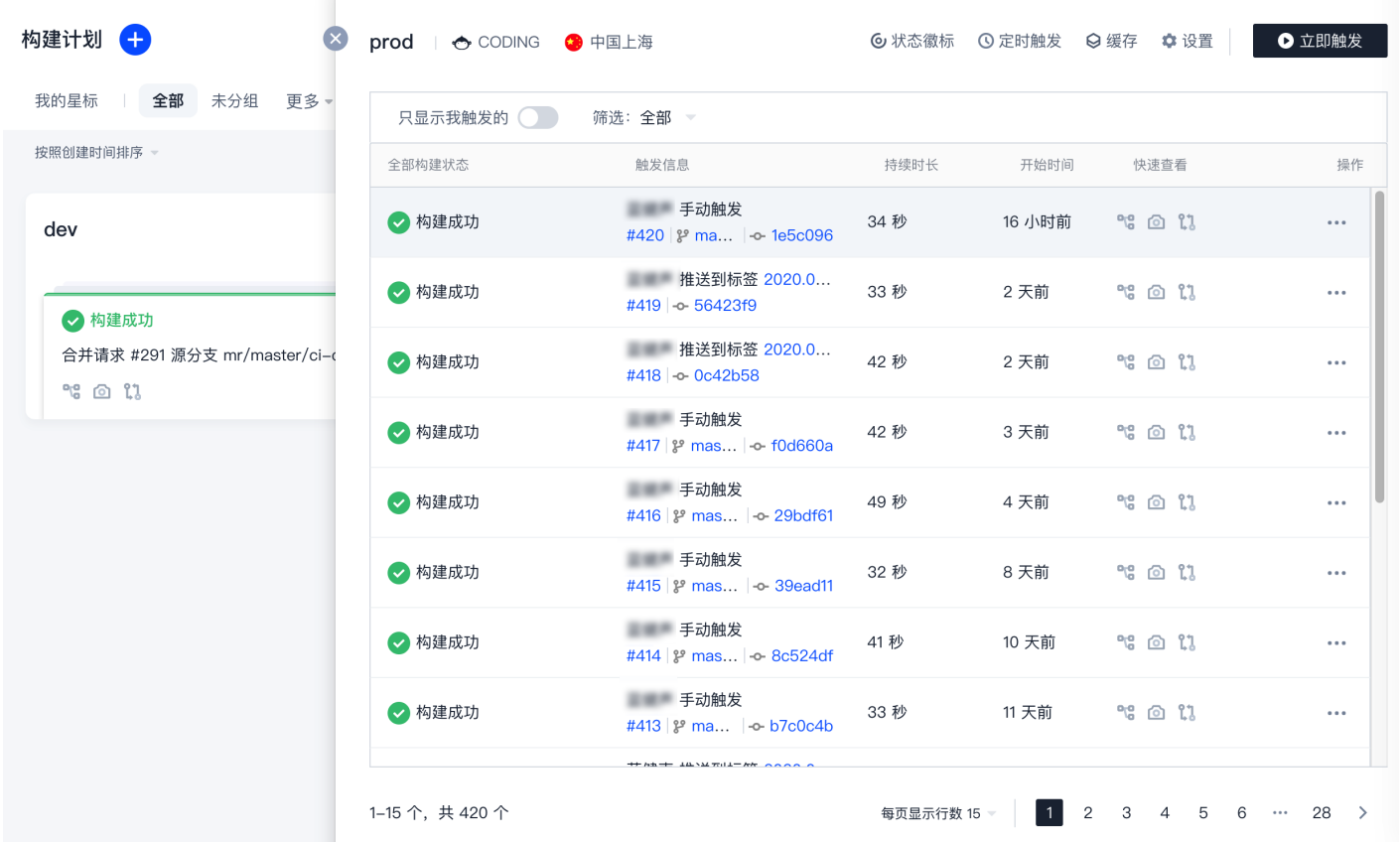
1. 将获取到的 API 密钥上传至 CODING 进行托管以获得凭据 ID。


2. 在变量与缓存中单击选择**增加环境变量** > **类别选择 CODING 凭据** > **选择您需调取的凭据**。


3. 在构建于部署中新建计划列表，并填写相应的 Jenkinsfile 配置。
   Jenkinsfile 配置：

```
pipeline {
agent any
stages {
stage('获取云 API 密钥') {
steps {

withCredentials([cloudApi(credentialsId: '此处填写您上传凭据后所生成的凭据 ID', secretIdVariable: 'CLOUD_API_SECRET_ID', secretKeyVariable: 'CLOUD_API_SECRET_KEY')]) {
sh 'CLOUD_API_SECRET_ID=${CLOUD_API_SECRET_ID}'
sh 'CLOUD_API_SECRET_KEY=${CLOUD_API_SECRET_KEY}'
}
withCredentials([[$class: 'CloudApiCredentialsBinding', credentialsId: '此处填写您上传凭据后所生成的凭据 ID', secretIdVariable: 'CLOUD_API_SECRET_ID', secretKeyVariable: 'CLOUD_API_SECRET_KEY']]) {
sh 'CLOUD_API_SECRET_ID=${CLOUD_API_SECRET_ID}'
sh 'CLOUD_API_SECRET_KEY=${CLOUD_API_SECRET_KEY}'
}
}
}
}
}
```

**4. 构建完成**




**参数说明**

| 参数名称 | 是否必填 | 说明 |
| --- | --- | --- |
| credentialsId | 是 | 需要获取的凭据 ID，仅支持云 API 类型的凭据。 |
| secretIdVariable | 是 | secretId 环境变量的名称，会用配置名称注入环境变量。 |
| secretKeyVariable | 是 | secretKey 环境变量的名称，会用配置名称注入环境变量。 |

# 自动添加合并请求评审者

最近更新时间：2022-03-28 10:43:36

本文为您介绍如何使用持续集成插件自动添加合并请求评审者。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

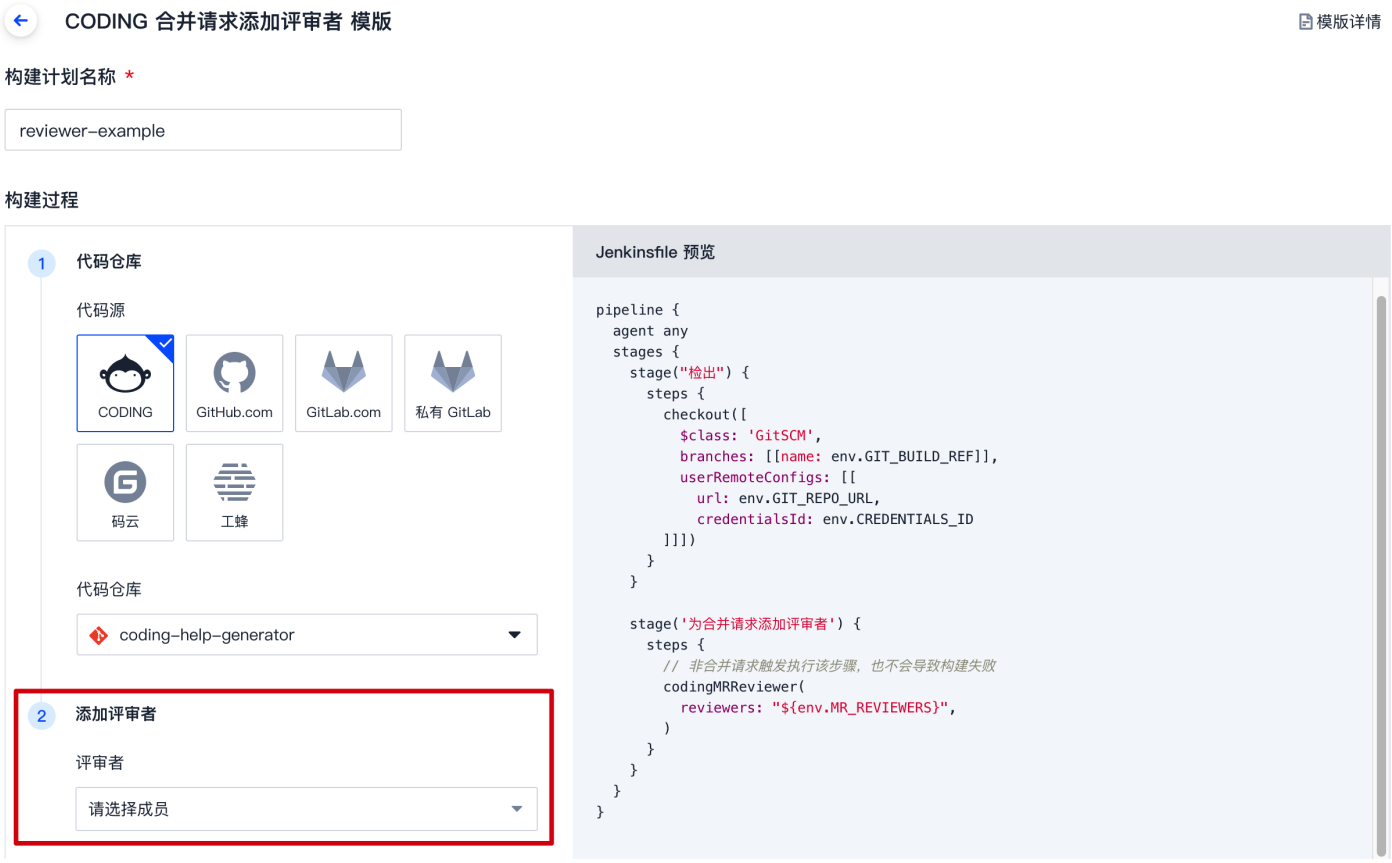
## 功能介绍

CODING 的内置插件支持用户在持续集成中触发合并请求功能时，系统自动为合并请求添加评审者。您可以使用内置的模板或使用 Jenkinsfile 配置来实现此功能。

## 内置模板

1. 新建构建计划后，选择 **CODING 合并请求添加评审者**模板。

2. 选择指定的评审者。




3. 在触发合并请求时，系统会自动添加评审者。


## Jenkinsfile 配置

```
pipeline {
agent any
stages {
stage('为合并请求添加评审者') {
steps {
codingMRReviewer(
reviewers: '此处填写评审者邮箱',
```

```
)
}
}
}
}
```

将上述配置写入代码仓库的 Jenkinsfile 配置文件后，在本地使用 git push origin localbranch:mr/targetbranch/localbranch 命令就可以实现一下效果：
创建**合并请求 > 触发构建 > 自动化测试和构建 > 自动添加合并请求评审者**。

## 更多参数演示

```
pipeline {
agent any
stages {
stage('为合并请求添加评审者') {
steps {
codingMRReviewer(
reviewers: 'coding@coding.com,test2,coding',
mrResourceId: '${env.MR_RESOURCE_ID}',
credentialsId: '${env.CODING_ARTIFACTS_CREDENTIALS_ID}',
withBranchAdmin: false,
)
}
}
}
}
```

## 参数说明

| 参数名称 | 是否必填 | 文本参数类型 | 图形化参数类型 | 默认值 | 说明 |
| --- | --- | --- | --- | --- | --- |
| reviewers | 是 | string | 项目成员（多选） | - | 需要添加 coding （项目外 |
| mrResourceId | 否 | string | string | ${env.MR_RESOURCE_ID} | 需要添加 系统环境 MR 时之 评审者） |
| credentialsId | 否 | string | 凭据（用户名+密码） | ${env.CCI_CURRENT_PROJECT_COMMON_CREDENTIALS_ID} | 用于发起 须为项目 CCI_C |
| (尚未支持) withBranchAdmin | 否 | boolean | boolean | false | 自动邀请 配置。 |

## 常见问题

**如果配置的 reviewer 不存在会导致构建失败吗?**
不会，但是如果 reviewer 没有配置，则会构建失败。若无法在当前的团队内找到所有对应的 reviewers， 则会跳过**添加评审者**的操作，并将当前 stage 的构建状态标记为 UNSTABLE（不稳定的），并且不会导致构建的失败。
您可以在日志中看到对应的警告信息，会提示无法在当前项目 my-project 中找到评审者 test2、test3。

若只有部分 reviewer 未找到，则只能添加已经存在的 reviewer，同理也将标记为 UNSTABLE（不稳定的），并在日志中输出对应的警告信息。


**如果 mrResourceId 对应的合并请求已经合并或者没有配置 mrResourceID 会导致构建失败吗？**

不会，若 mrResourceId 没有配置且 ${env.MR_RESOURCE_ID} 不存在，也不会导致构建失败，但同样会输出相关的警告日志。

至于不存在或已经合并的情况，会将对应的 stage 标记为 UNSTABLE (不稳定的)， 然后输出对应的构建日志。

**多次添加重复的 reviewer 会导致构建失败吗?**

不会，内置插件会自动跳过已经添加的 reviewer。

# 人工确认

最近更新时间：2022-03-25 15:14:13

本文为您介绍如何使用持续集成插件为构建过程添加人工确认机制。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

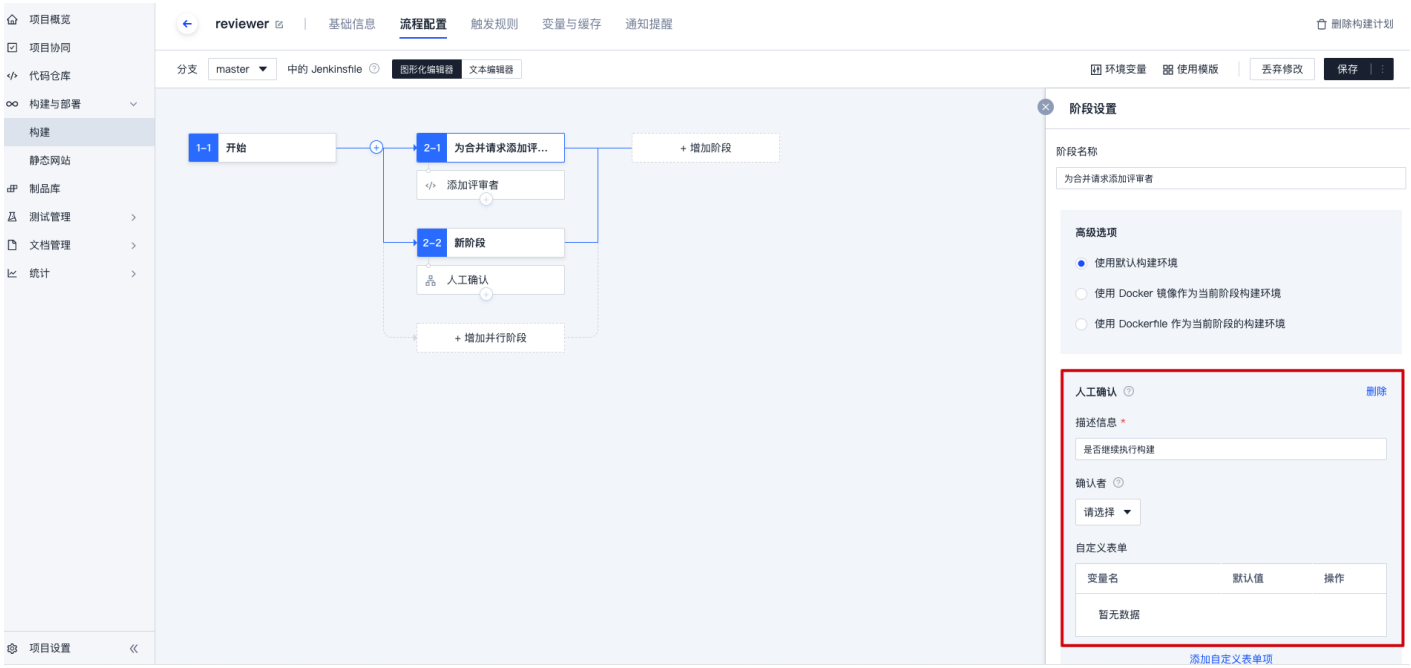
1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

## 功能介绍

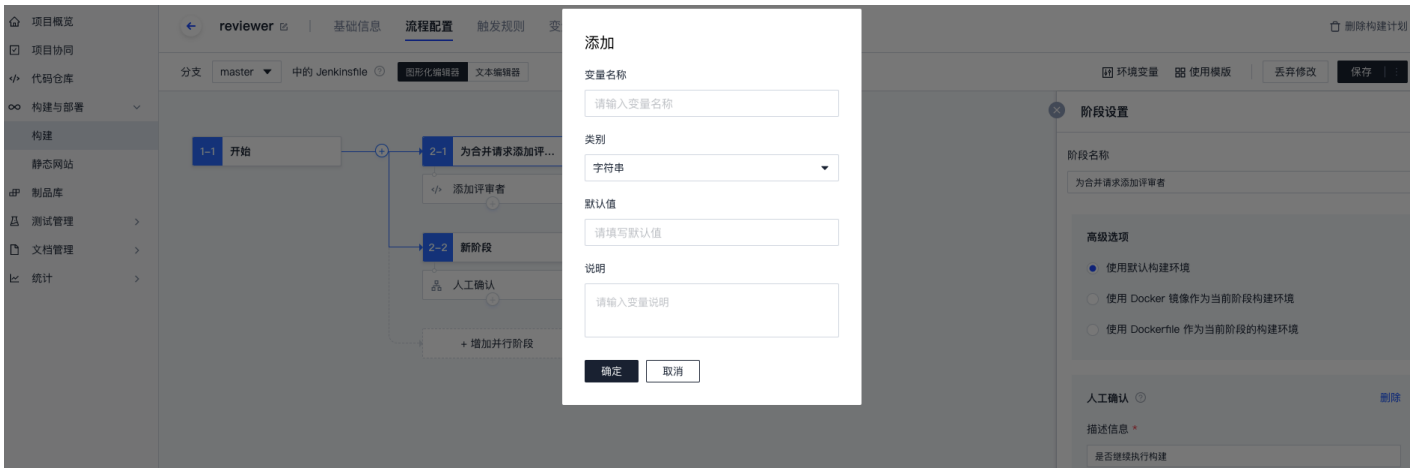
在构建任务里的流程配置中加入人工确认步骤，便可以指定确认者和编排自定义表单。设置完成后，当持续集成运行至相应的阶段后将会暂停构建，而后通知确认者，待完成确认后方才进行下一步构建过程。目前支持可视化编排和文本编辑配置两种方式开启人工确认。

## 可视化编排

1. 在**持续构建 > 流程配置 > 阶段设置**中点击开启人工确认并完成相应的功能设置。



2. 填写自定义功能表单，具体参数说明请参见 基础参数。




3. 在并行阶段中也支持添加人工确认。


## 文本编辑配置

在**持续集成 > 流程配置 > 文本编辑器**中填写相关配置。

### 快速开始

```
pipeline {
agent any
stages {
stage('人工确认演示') {
input {
message "是否继续执行构建"
}
steps {
echo "Hello World"
}
}
}
}
```

### 阶段中的语法演示

```
pipeline {
agent any
```

```
stages {
stage('人工确认演示') {
input {
message '是否继续执行构建'
submitter 'coding@coding.com,tester@coding.com'
parameters {
string(name: 'tag', defaultValue: '20191111.1', description: '请输入希望发布的版本号')
choice(name: 'testing_env', choices:['testing-1','testing-2','testing-3'], description: '请选择要部署测试环境')
booleanParam(name: 'qa', defaultValue: false, description: '测试是否通过')
}
}
steps {
echo "您希望发布的版本号为 ${tag}"
echo "您希望部署的测试环境为 ${testing_env}"
echo "本地测试是否通过: ${qa}"
}
}
}
}
```

**步骤中的语法演示**

```
pipeline {
agent any
stages {
stage("人工确认演示") {
steps {
input (
message: '是否继续执行构建',
submitter: 'coding@coding.com,tester@coding.com',
parameters:[
string(name: 'tag', defaultValue: '20191111.1', description: '请输入希望发布的版本号'),
choice(name: 'testing_env', choices:['testing-1','testing-2','testing-3'], description: '请选择要部署测试环境'),
booleanParam(name: 'qa', defaultValue: false, description: '测试是否通过'),
]
)
echo "您希望发布的版本号为 ${tag}"
echo "您希望部署的测试环境为 ${testing_env}"
echo "本地测试是否通过: ${qa}"
}
}
}
}
```

## 参数说明

**基础参数**

| 参数名称 | 必填 | 文本参数类型 | 图形化参数类型 | 说明 |
| --- | --- | --- | --- | --- |
| message | 是 | string | string | 人工确认的描述信息，将会显示在人工确认的界面上。 |
| submitter | 否 | string | 项目成员（多选） | 人工确认的确认者，只有指定的确认者或项目管理员才能操作，多个确认者以逗号分隔，如：ZhangSan、LiSi。支持填写项目内用户的邮箱或 GK（项目外或者不存在成员，会添加无效）。 |

| 参数名称 | 必填 | 文本参数类型 | 图形化参数类型 | 说明 |
| --- | --- | --- | --- | --- |
| parameters | 否 | 自定义人工确认表单，目前支持**字符串、单选和布尔值**三种类型，表单的值在确认后将会以 Jenkinsfile 变量的形式注入到环境中。 | - | |

**确认者的具体规则说明**

- 非必填项。
- 会通知确认者前往对应的构建任务页面进行操作。
- 只有选定的确认者和项目管理员才能对人工确认步骤进行确认操作。
- 若没有配置确认者，则会自动向触发者发送通知提醒。
- 若没有配置确认者，则项目内所有成员均可以进行操作。

### parameters - string 字符串类型

| 参数名称 | 是否必填 | 说明 |
| --- | --- | --- |
| name | 是 | 参数名称，表单的值在确认后将会以 Jenkinsfile 变量的形式注入到环境中。 |
| defaultValue | 否 | 单表的默认值。 |
| description | 否 | 参数说明，用于显示在人工确认的界面上辅助说明。 |

### parameters - choice 单选类型

| 参数名称 | 是否必填 | 说明 |
| --- | --- | --- |
| name | 是 | 参数名称，表单的值在确认后将会以 Jenkinsfile 变量的形式注入到环境中。 |
| defaultValue | 否 | 单表的默认值。 |
| description | 否 | 参数说明，用于显示在人工确认的界面上辅助说明。 |

### parameters - booleanParam 布尔值类型

| 参数名称 | 是否必填 | 说明 |
| --- | --- | --- |
| name | 是 | 参数名称，表单的值在确认后将会以 Jenkinsfile 变量的形式注入到环境中。 |
| defaultValue | 否 | 单表的默认值。 |
| description | 否 | 参数说明，用于显示在人工确认的界面上辅助说明。 |

## 设置超时时间

使用**限时子步骤**插件，在子步骤中添加**人工确认**，即可自定义超时时间。


```
stage('先审后发') {
steps {
timeout(time: 5, activity: false, unit: 'MINUTES') {
input(message: '是否继续执行构建', submitter: 'support@coding.net')
}
//sh 'coscmd upload -r ./dist /'
}
}
```

## 常见问题

**人工确认会有超时的限制么？**

有超时的限制。使用 CODING 提供的云服务器所执行的持续集成任务都会受到团队配额的超时时间限制。如果执行期间超过配额的时间，将自动停止掉构建任务。若使用了自定义构建节点，则不受团队配额的时间限制。

**如何使用人工确认的方式来进行 Debug？**

可以使用如下配置命令进行调试。

```
pipeline {
agent any
stages {
stage("debug") {
```

```
steps {
script {
while (true) {
def cmd = input (
message: '请输入想要执行的命令',
parameters: [
string(defaultValue: '', description: '', name: 'cmd')
])
try {
// 若输入为 exit， 则退出 Debug
if (cmd == "exit") {
break
}
sh "${cmd}"
} catch (e) {
print e
}
}
echo 'hello world'
}

}
}
}
}
```

# 收集通用报告

最近更新时间：2023-06-06 16:34:05

本文为您介绍如何使用持续集成插件收集通用报告。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

## 功能介绍

在您使用持续集成进行构建时，可能会使用不同工具或插件生成 HTML 格式的报告，例如单元测试报告、代码覆盖率报告，以及 API 文档等。在添加 CODING 通用报告收集插件并进行简单配置后，即可在**构建记录** > **通用报告**中直接浏览或快速下载收集到的各类 HTML 格式报告。

### 快速开始

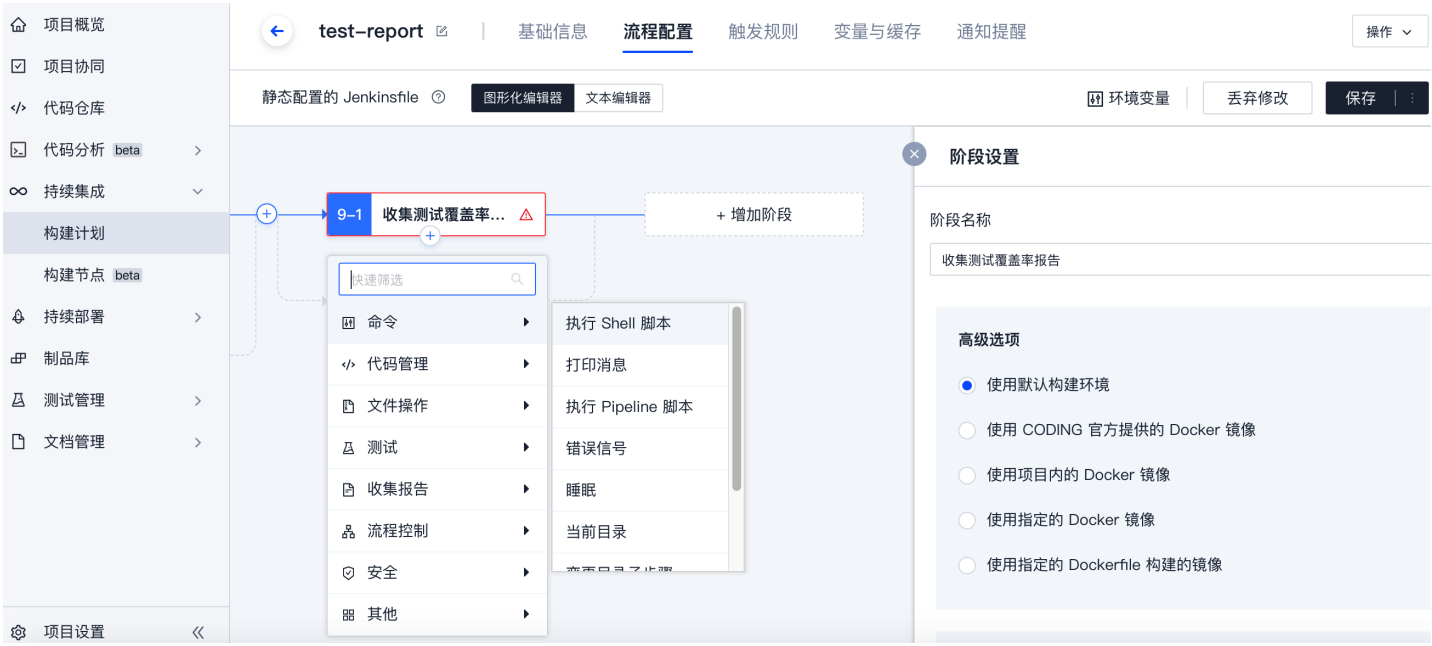
进入构建计划的设置页面，您可以通过图形化界面或文本编辑器两种方式添加步骤，收集在构建计划过程所产生的各类 HTML 格式报告。下面将会以搜集测试覆盖率 HTML 报告为例，逐个说明相应的配置方法，演示如何使用 CODING 通用报告插件收集用户自定义报告。

## 使用图形化界面收集通用报告

CODING 持续集成的**图形化编辑器**在基于编辑 Jenkinsfile（过程描述文件）的核心功能之上，设计了可视化视图，同时也兼容绝大部分自定义操作，实现了边写边看、所见即所得的直观编辑体验。


**步骤1：添加步骤**

新建或选择构建计划，进入流程配置界面。在**单元测试**步骤后添加**收集测试覆盖率报告**阶段，并增加执行 shell 脚本命令步骤。




### 步骤2：生成测试覆盖率报告

调用项目中配置的脚本生成测试覆盖率报告。示例中的测试覆盖率报告脚本默认会在项目目录下创建名为 coverage 的文件夹保存测试报告文件，入口文件为 index.html；您也可以根据自身使用的测试覆盖率报告生成工具进行命令调整。


### 步骤3：添加步骤

在生成覆盖率报告脚本的下一步添加**收集通用报告**步骤。


## 步骤4：填写配置

填写收集通用报告配置，详情请参见 参数说明 部分。


## 步骤5：构建完成后查看报告

**单击保存，完成插件配置。构建计划运行完成后前往构建记录 > 通用报告页面查看并下载测试覆盖率报告。**


## 使用文本编辑器收集通用报告

**文本编辑器本质上是在编写 Jenkinsfile，它定义了持续集成中的工作流水线（pipeline），其不仅实现了对步骤的流式化封装和管理，也是持续集成中的基本功能单位。流水线可以顺序执行，也可以并行执行。**

### 步骤1：进入文本编辑器

**新建或选择构建计划，进入设置界面。单击流程配置界面，选择文本编辑器。**


### 步骤2：增加步骤

**在单元测试步骤后增加收集代码测试覆盖率报告步骤。**

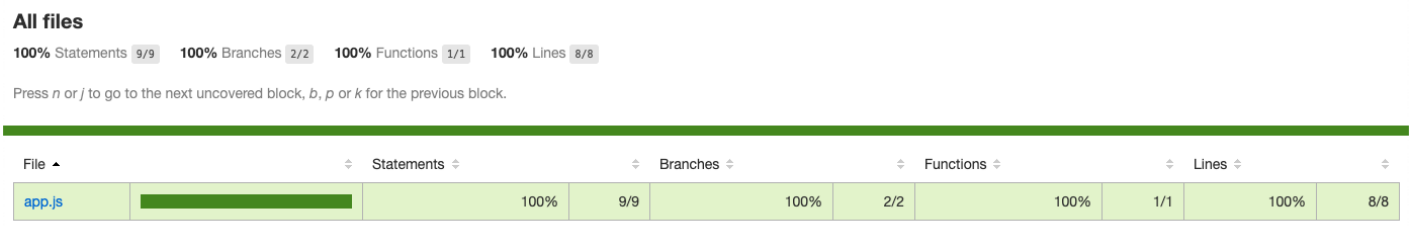
```
pipeline {
agent any

// ...

stage('收集代码测试覆盖率报告') {
steps {
// 生成覆盖率报告，默认保存在项目目录下 coverage 文件夹下。您也可以根据自身使用的测试覆盖率报告生成工具进行命令调整
sh 'npm run coverage'
// 使用 CODING 通用报告收集插件收集测试覆盖率报告，配置详情请见“参数说明”部分
codingHtmlReport(
name: 'my-report',
tag: 'test',
path: 'coverage',
des: 'This is the coverage report',
entryFile: 'index.html'
```

```
)
}
}

// ...
}
```

### 步骤3：构建完成后查看报告

1. 单击**保存**，完成插件配置。构建计划运行完成后前往**构建记录** > **通用报告**页面查看并下载测试覆盖率报告。


2. 单击查看就可以直接在新的 Tab 看到测试覆盖率报告。




### 更多参数演示

```
pipeline {
agent any
stages {
stage("html-report") {
steps {
// 在 my-dir 文件夹中生成一个 index.html 文件
sh 'mkdir my-dir'
sh 'echo hello coding > my-dir/index.html'

// 上传到通用报告中
codingHtmlReport(
name: 'my-test-report',
path: 'my-dir',
entryFile: 'index.html',
tag: 'test',
des: 'something'
)
}
}
}
}
```

**参数说明**

| 参数名称 | 图形化界面对应选项名 | 必填 | 文本参数类型 | 图形化参数类型 | 默认值 | 说明 |
| --- | --- | --- | --- | --- | --- | --- |
| name | 报告名称 | 是 | string | string | - | 通用报告的名称，应在构建记录下为唯一值，不得超过50个字符。 |
| path | 路径 | 是 | string | string | - | 通用报告所在的路径，可以为文件夹或者文件。 |
| entryFile | 入口文件 | 否 | string | string | index.html | 若 path 为文件夹时，可以为 path 设置一个入口文件的位置，默认为 index.html，entryFile 配置的路径将相对于 path。若 path 为文件时，此参数无效。 |
| tag | 标签 | 否 | string | string | - | 通用报告的标签，支持通过自定义标签来给报告分类，不得超过30个字符。 |
| des | 描述 | 否 | string | string(textarea) | - | 通用报告描述，不得超过300个字符。 |

# 上传 API 文档

最近更新时间：2022-03-25 15:14:23

本文为您介绍如何使用持续集成插件上传 API 文档。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

## 功能介绍

CODING 持续集成插件**读取代码生成 API 文档**可以提取代码中的注释，生成 API 文档并发布。

## 在代码中编写注释

在代码中编写注释，请参见 OpenAPI/Swagger 编写与导入指南。
并且在本地调试通过，例如 PHP Laravel Swagger 生成文档的命令是：

```
php artisan l5-swagger:generate
ls storage/api-docs/api-docs.json
```

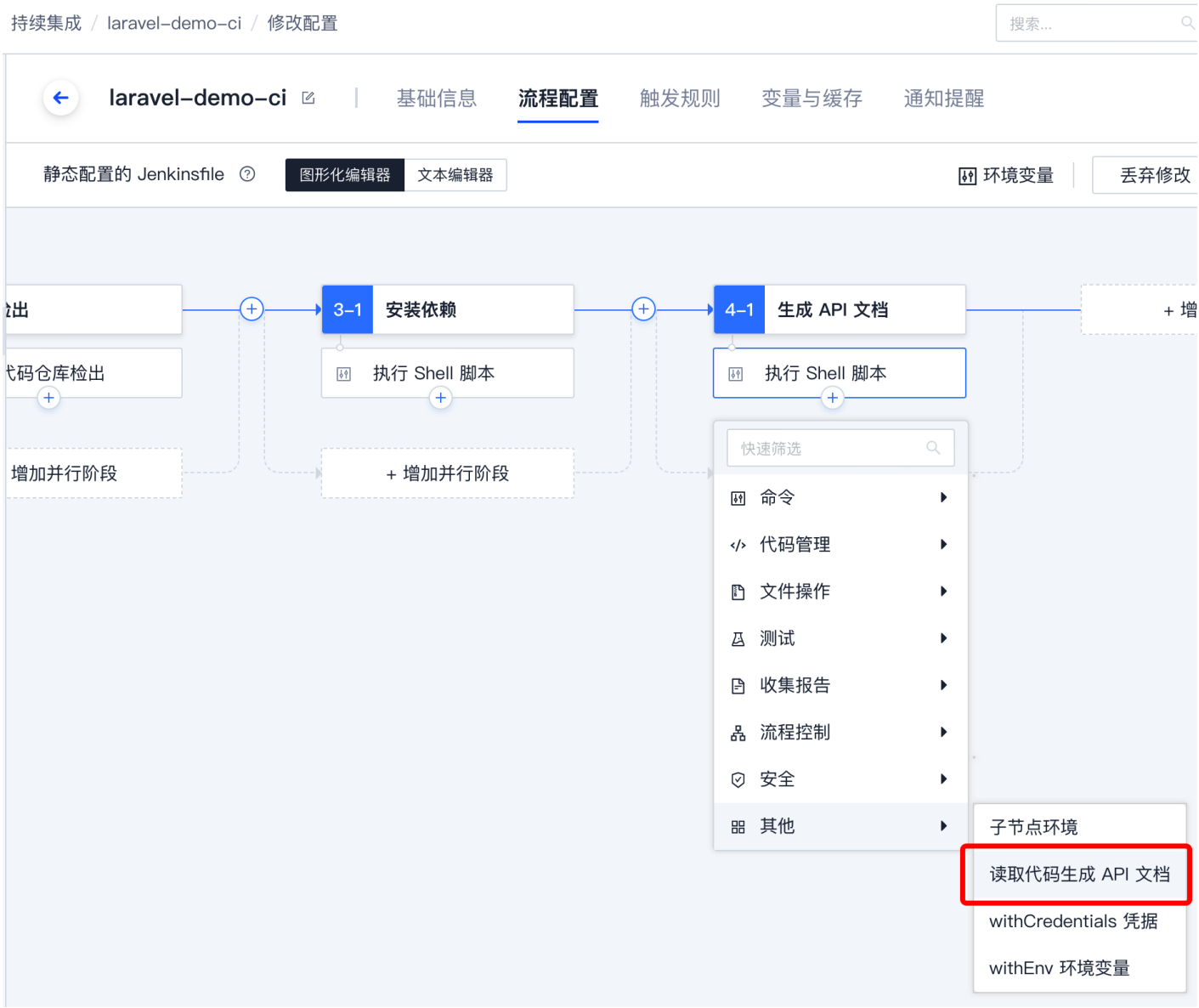
## 创建空的 API 文档

在**文档管理** > **API 文档**中创建一个空的 API 文档。


## 使用图形化界面生成并上传 API 文档

1. 使用**持续集成**的图形化编辑器，添加一个步骤**执行 Shell 脚本**，填入生成 API 文档的命令。


2. 再添加一个步骤**读取代码生成 API 文档**，**语言和注释库**选择**其他**，填写之前生成的 json 文件路径，并且选择之前创建的 **API 文档**。




## Jenkinsfile

**也可以使用持续集成的文本编辑器，填入以下代码：**

```
pipeline {
agent {
docker {
image 'sinkcup/laravel-demo:6-dev'
args '-v /root/.composer:/root/.composer'
reuseNode true
}
}
stages {
stage('检出') {
steps {
checkout([$class: 'GitSCM', branches: [[name: env.GIT_BUILD_REF]],
userRemoteConfigs: [[url: env.GIT_REPO_URL, credentialsId: env.CREDENTIALS_ID]]])
}
}
stage('安装依赖') {
steps {
sh 'composer install'
}
}
stage('生成 API 文档') {
steps {
sh 'php artisan l5-swagger:generate'
codingReleaseApiDoc(apiDocId: '1', apiDocType: 'specificFile', resultFile: 'storage/api-docs/api-docs.json')
}
}
}
}
```

**手动或自动执行构建计划，成功后，即可通过文档管理 > API 文档中的链接进行访问。**

# 在阶段末尾执行插件

最近更新时间：2022-03-28 10:45:41

本文为您介绍如何在构建阶段末尾执行插件。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

## 功能介绍

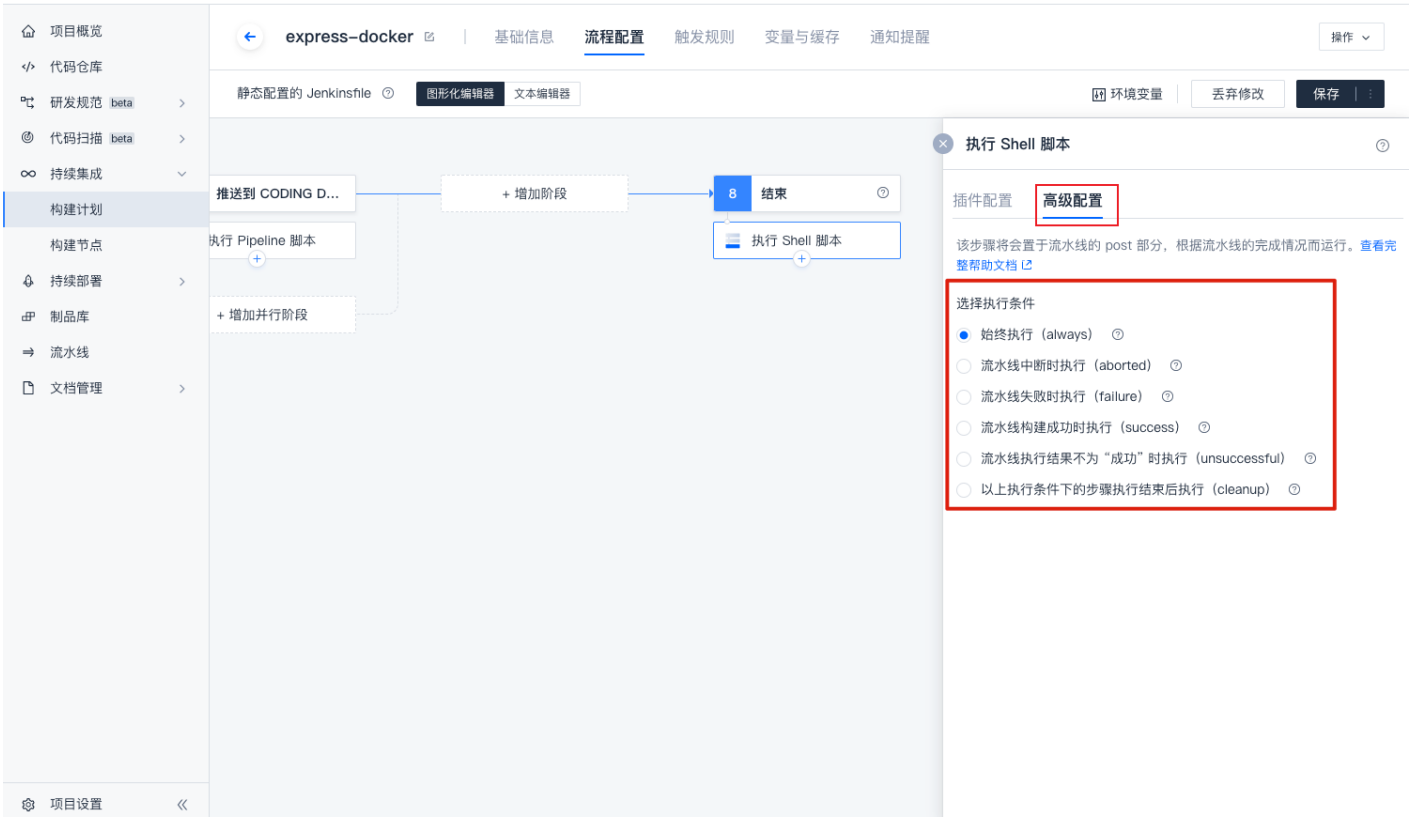
在配置持续集成的过程中，有一些步骤需要根据流水线与阶段的执行情况决定是否执行。
例如：当**单元测试**步骤执行失败时，**收集测试报告**步骤就没有必要执行了。因此我们可以为**收集测试报告**步骤添加判断条件，让它在之前的**单元测试**阶段执行成功后方才执行。

您可以通过持续集成步骤里的**置底执行**开关控制实现上述效果。

## 使用图形化编辑器配置流水线与阶段末尾执行

### 流水线末尾执行

1. 在流水线末尾的**结束阶段**单击+添加步骤，进入**高级配置**，选择执行条件。默认为始终执行，单击 执行条件详情 查看详情。



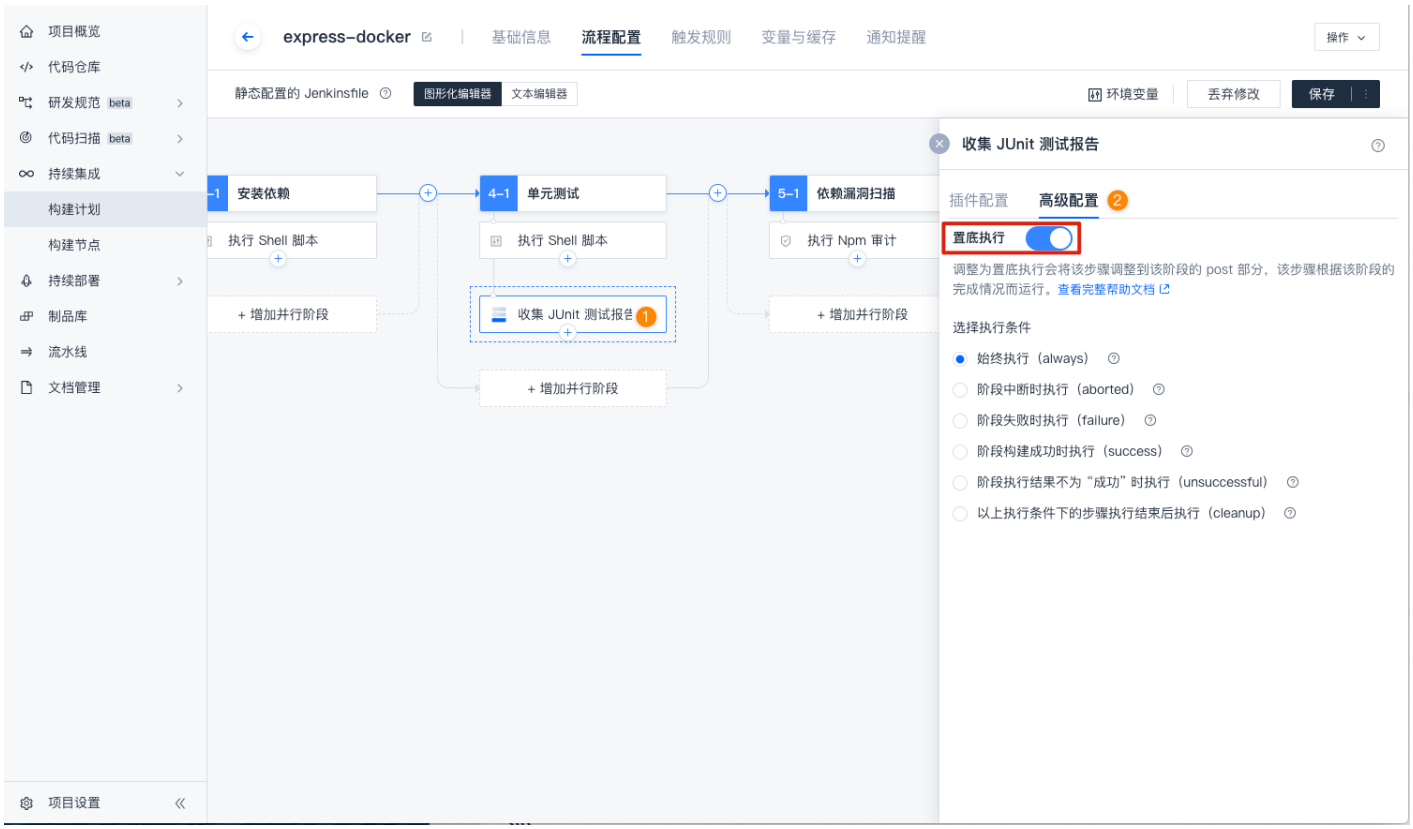

2. 配置成功后，在流水线的最后阶段，构建任务会根据执行条件执行步骤。相同执行条件的步骤将会按照结束阶段里的设置顺序执行，不同执行条件下的步骤执行顺序互不影响。

### 阶段末尾执行

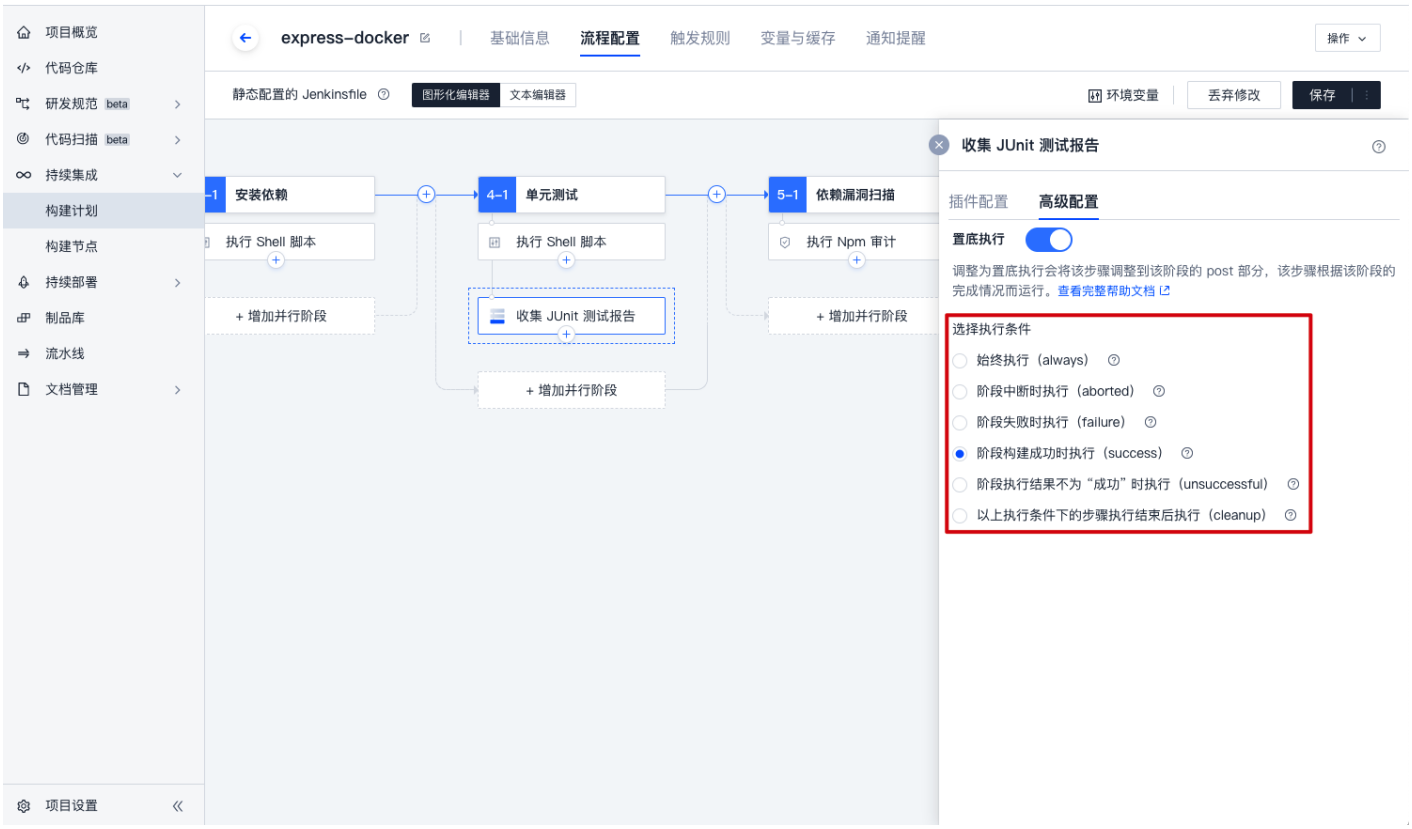
如果您根据阶段的执行情况设置特定步骤，可以将阶段内的步骤设置为**置底执行**。

1. 添加一个普通步骤。


2. 进入高级配置，打开**置底执行**开关，该步骤会自动跳到**置底执行**区域。



3. 选择执行条件，默认为始终执行。查看执行条件详情请到 执行条件详情 。




## Jenkinsfile

```
pipeline {
agent any
stages {
stage('检出') {
steps {
checkout([$class: 'GitSCM',
branches: [[name: env.GIT_BUILD_REF]],
userRemoteConfigs: [[
url: env.GIT_REPO_URL,
credentialsId: env.CREDENTIALS_ID
]]])
}
}
stage('安装依赖') {
steps {
sh 'npm install'
}
}
stage('单元测试') {
// 该区域的步骤将会在单元测试阶段末尾执行
post {
// 当单元测试阶段执行成功，才会执行 success 区域中的测试报告收集步骤
success {
junit 'target/surefire-reports/*.xml'
}
}
steps {
sh 'npm test'
```

```
}
}
}
// 该区域的步骤将会在流水线末尾执行
post {
// always 区域步骤会始终执行
always {
sh 'echo hello CODING'
}
}
}
```

### 执行条件详情

- always：无论当前阶段的完成状态如何，都允许运行该步骤。
- aborted：当前阶段被中断时，允许运行该步骤（例如手动终止）。
- failure：若当前阶段失败时，允许运行该步骤。
- success：当前阶段运行成功后，允许运行该步骤。
- unsuccessful：当前阶段的构建结果产出失败后，允许运行该步骤。
- cleanup：待其他条件执行完毕后再执行该步骤。

# 代码扫描

最近更新时间：2022-04-01 15:07:06

本文为您介绍如何使用持续集成中的代码扫描插件。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

## 功能介绍

持续集成支持执行内置的**代码扫描**插件，通过便捷的设置就能在持续集成任务中触发代码扫描功能。辅以不同的 触发规则 还能够自定义触发代码扫描的条件，这样便能够及时发现潜藏的代码缺陷、安全漏洞以及不规范代码，把控构建产物质量。

> 说明
> 目前代码扫描插件仅支持 CODING 代码仓库。

## 快速开始

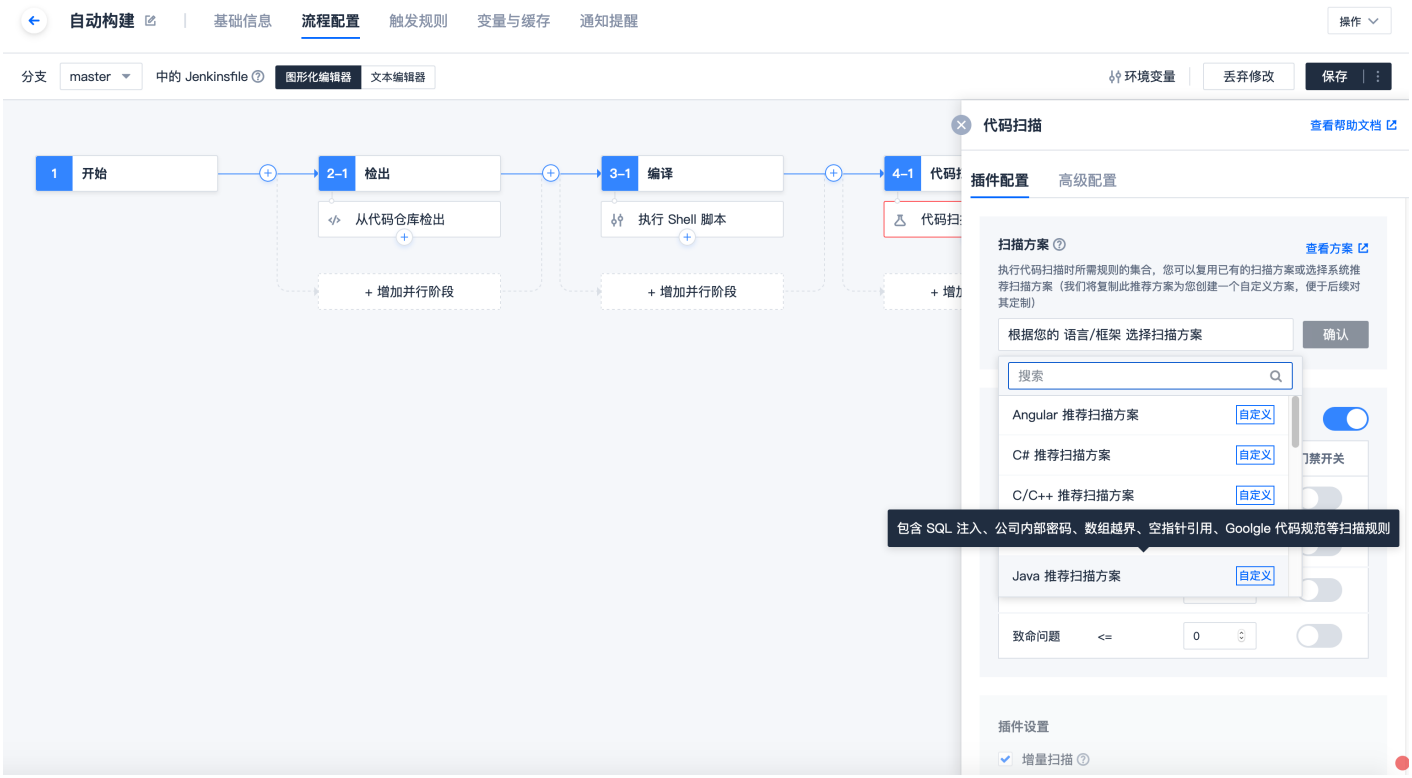
本文档通过可视化界面配置代码扫描插件。进入任一项目后，依次单击并进入**持续集成** > **构建计划** > **设置** > **流程配置**。接下来将以扫描 Java 项目为例，说明如何配置并使用代码扫描插件。

### 添加扫描插件

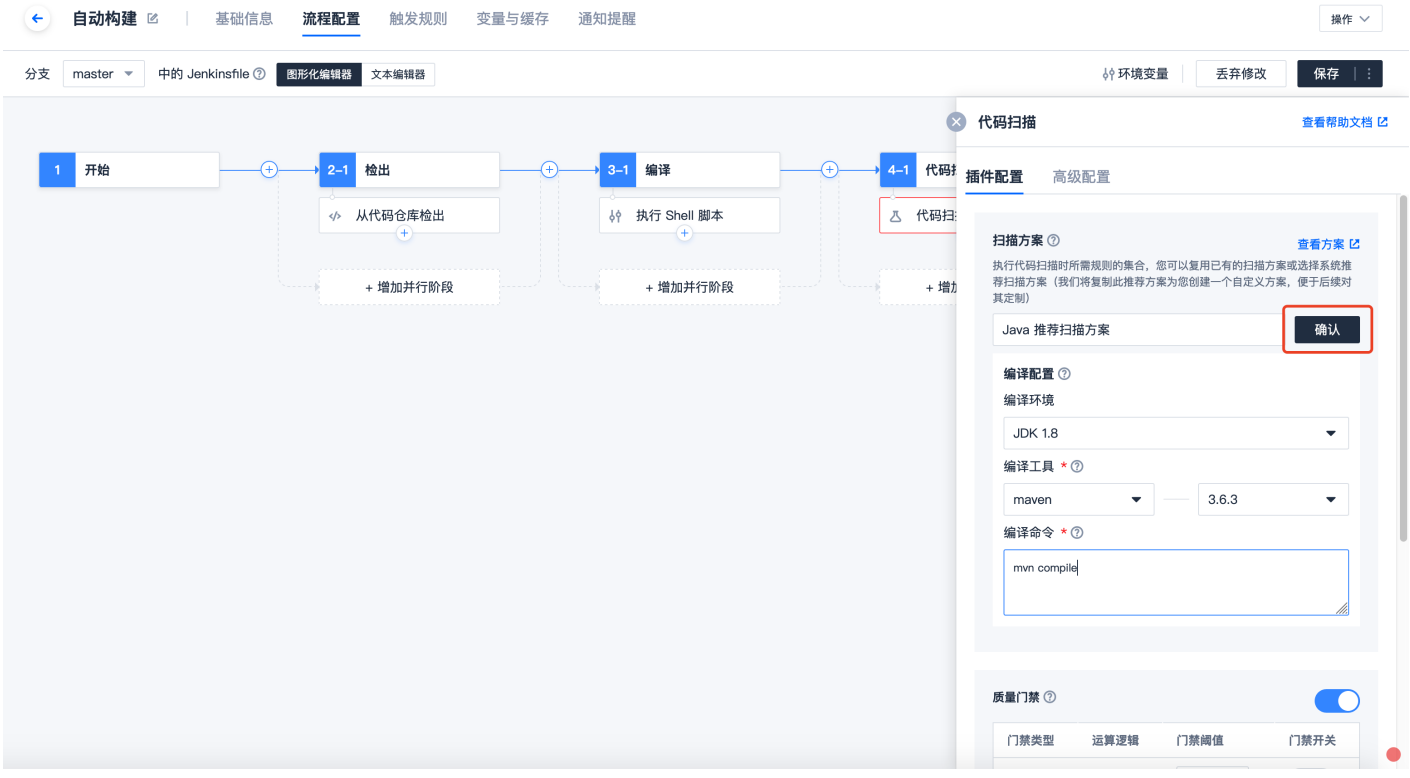
新建或选择构建计划，进入流程配置界面。建议在较早阶段如**单元测试**环节前就新增代码扫描阶段，然后在**质量管理**环节中添加代码扫描插件。


### 选择扫描方案

1. 根据项目本身选择开发语言，在此例子中我们选择 **Java 推荐扫描方案**。

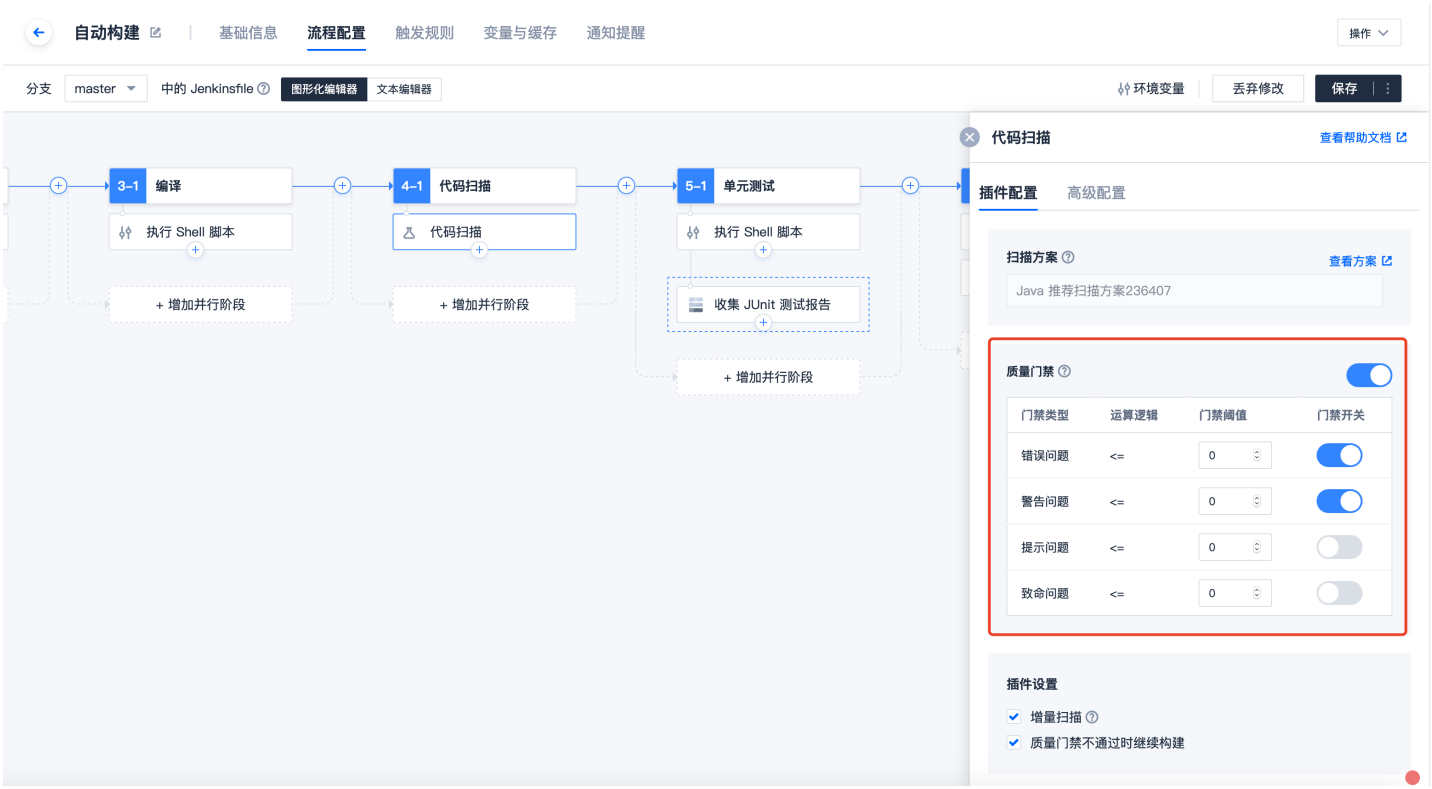



2. 因该扫描方案中包含编译型规则，所以需要进行配置编译以追踪更深层次的缺陷，添加扫描方案后单击确认。



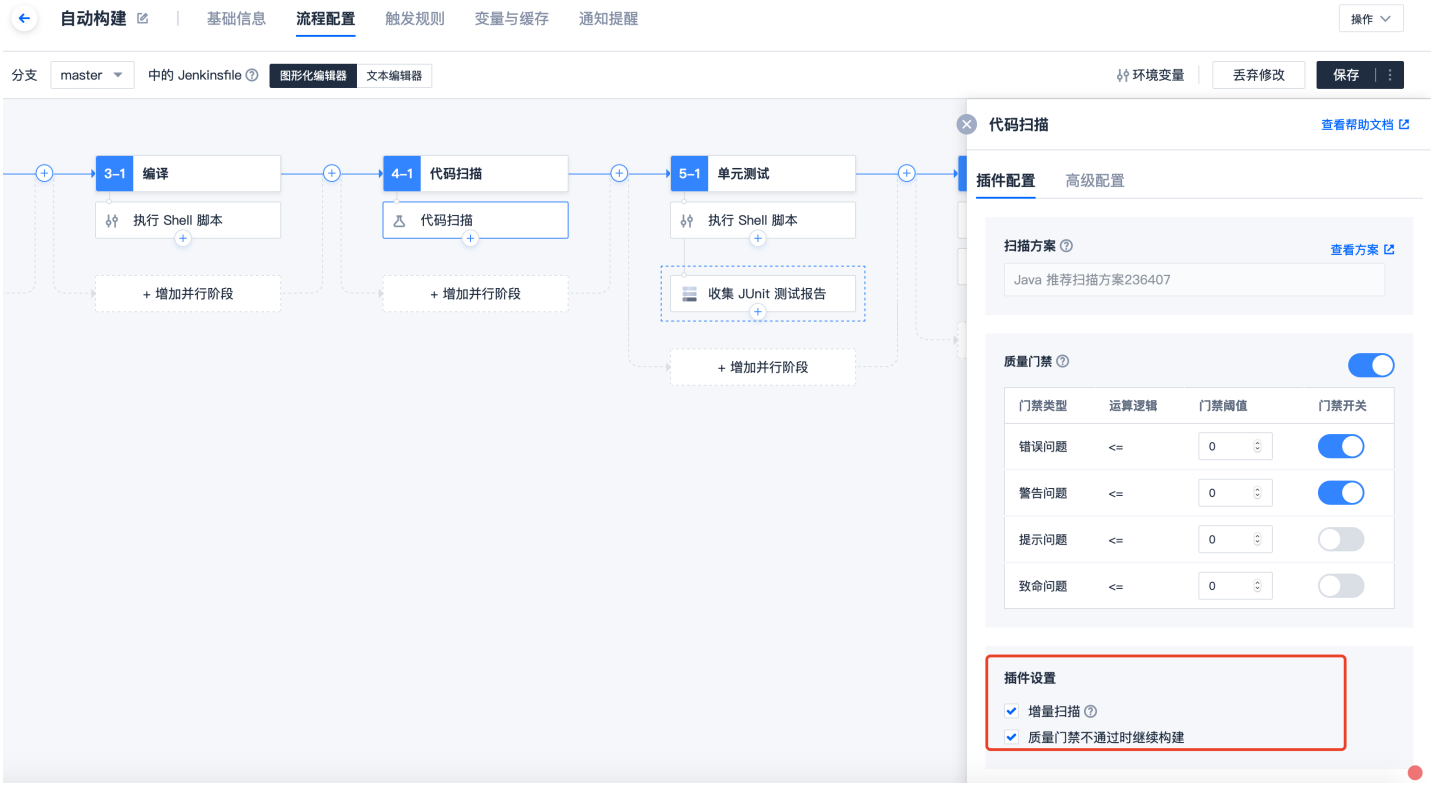

**配置质量门禁**

依据团队及项目的实际情况，为团队内不同权限的成员设置合适的质量门禁。在持续集成阶段就实现对错误问题与致命问题的有效管控。




## 配置插件设置

在插件设置中您还可以决定是否勾选在**质量门禁不通过时继续构建**选项。这取决于该构建计划的业务含义，建议在发布至正式环境的构建计划配置中取消勾选，即若门禁不通过则终止流水线，规避可能存在的风险。




## 查看扫描结果

扫描执行完成后，可以在插件的日志中了解质量门禁情况。单击日志中的链接，可以跳转至问题列表中进行查看。

# 镜像更新至 K8s 集群

最近更新时间：2022-03-25 15:14:38

本文为您介绍如何使用插件自动将镜像更新至 K8s 集群。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

## 功能介绍

当您在持续集成中将镜像构建完毕并推送至制品库后，此插件支持进一步将镜像推送至 Kubernetes 集群。您可以通过 持续集成模板 或在 流程配置 使用此插件。

## 模板创建

1. 进入持续集成功能页后，点击右上角的**创建构建计划**按钮，在**部署**中选择模板。

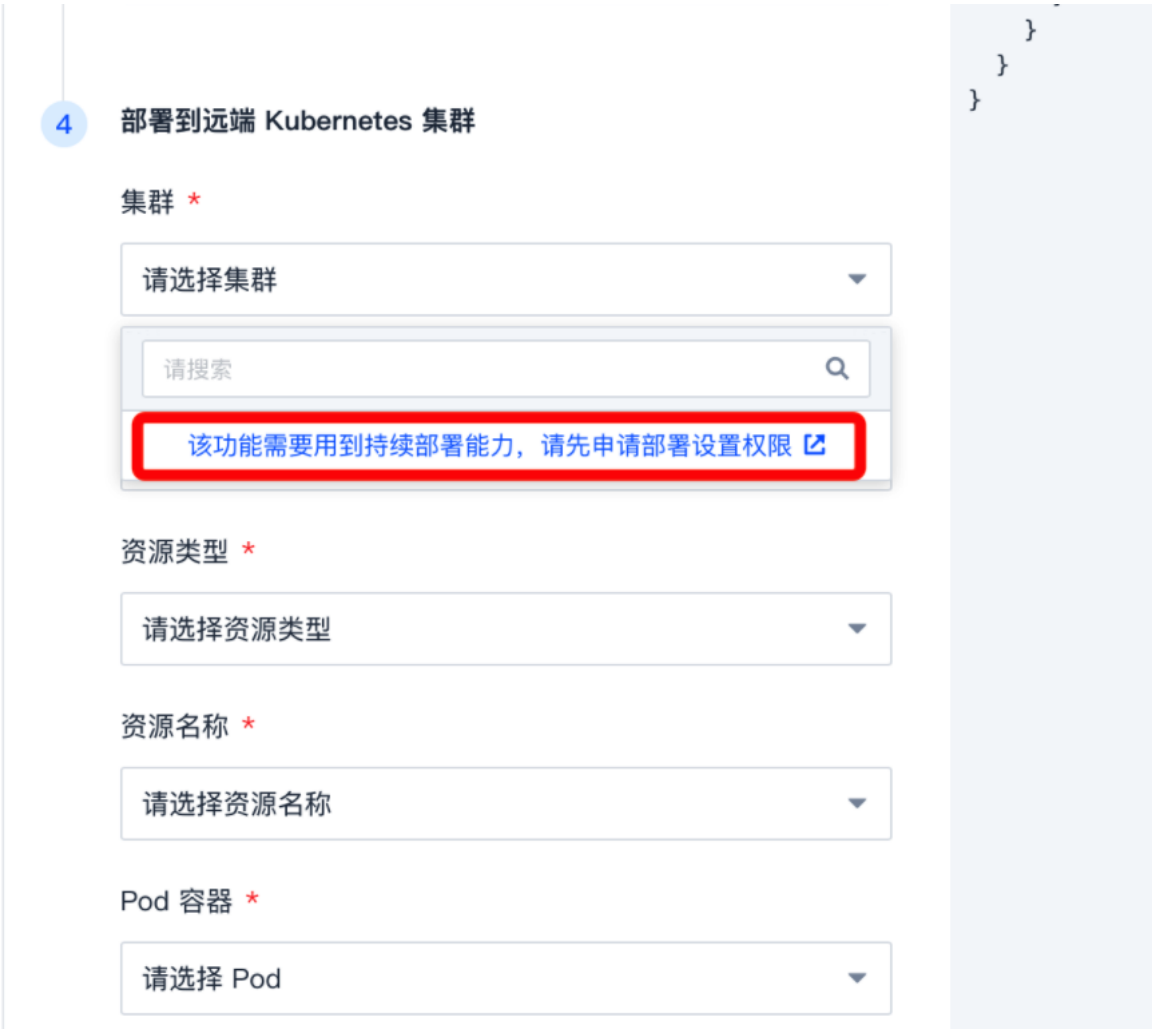
2. 在第三步中选择镜像上传的目标制品库，并在第四步中填写 Kubernetes 相关配置。


```
            namespace: "${CD_NAMESPACE_NAME}",
            manifestType: "${CD_MANIFEST_TYPE}",
            manifestName: "${CD_MANIFEST_NAME}",
            containerName: "${CD_CONTAINER_NAME}",
            credentialId: "${CD_CREDENTIAL_ID}",
            personalAccessToken: "${CD_PERSONAL_ACCESS_TOKEN}",
          ])
        }
      }
    }
  }
}
```

填写说明：

| 表单项 | 是否必填 | 说明 |
| --- | --- | --- |
| 集群 | 是 | 镜像更新的目标集群。 |
| 命名空间 | 是 | 已部署工作负载所在的命名空间。 |
| 资源类型 | 是 | 需要更新的工作负载类型。 |
| 资源名称 | 是 | 需要更新的工作负载名称。 |
| Pod 容器 | 是 | 一个 Pod 可能包含多个容器，此处指定需要升级的容器名称。 |

3. 因部署功能可以直接管控集群的资源，属于敏感权限。若没有部署设置权限，需向管理员申请。




4. 填写完成后勾选**创建后触发构建**，在构建过程中可以查看镜像更新的详细信息。


5. 跳转至持续部署页面后，查看发布单详情和容器运行详情。

## 流程配置创建

除了通过模板的方式创建，还可以在持续集成中的流程配置中添加插件。


## Jenkinsfile

```
pipeline {
agent any
stages {
stage('部署到远端 Kubernetes 集群') {
steps {
cdDeploy(deployType: 'PATCH_IMAGE', application: '${CCI_CURRENT_TEAM}', pipelineName: '${PROJECT_NAME}-${CCI_JOB_NAME}-${CD_CREDENTIAL_INDEX}', image: '"${CODING_DOCKER_IMAGE_NAME}:${DOCKER_IMAGE_VERSION}"', cloudAccountName: 'k8s', namespace: 'liaohongkun-1', manifestType: 'Deployment', manifestName: 'nginx-deployment-1', containerName: 'nginx', credentialId: '2f74c95a644xxxxxxxxxbbc461a049b')
}
}
}
}
```

## 参数说明

| 参数名称 | 图形化界面对应选项名 | 必填 | 文本参数类型 | 图形化参数类型 | 默认值 |
| --- | --- | --- | --- | --- | --- |
| deployType | - | 是 | string | string | PATCH_IMAGE |
| application | - | 是 | string | string | ${CCI_CURRENT_TEAM} |
| pipelineName | - | 是 | string | string | ${PROJECT_NAME}-${CCI_JOB_NAME}-${CD_CREDENTIAL_I |

| 参数名称 | 图形化界面对应选项名 | 必填 | 文本参数类型 | 图形化参数类型 | 默认值 |
| --- | --- | --- | --- | --- | --- |
| credentialId | - | 是 | string | string | - |
| image | 镜像 | 是 | string | string | - |
| cloudAccountName | 集群 | 是 | string | string | - |
| namespace | 命名空间 | 是 | string | string | - |
| manifestType | 资源类型 | 是 | string | string | - |
| manifestName | 资源名称 | 是 | string | string | - |
| containerName | Pod 容器名称 | 是 | string | string | - |

### 环境变量

| 变量名称 | 必填 | 参数类型 | 保密 | 说明 |
| --- | --- | --- | --- | --- |
| CD_PERSONAL_ACCESS_TOKEN | 是 | string | 是 | 权限为 project:deployment 的个人访问令牌 |

### 注意事项

部署属于敏感操作。由于部署设置权限限制的原因，需要生成 CD 发布凭据（与项目和构建计划绑定）进行授权对应的构建计划，因此不支持复制构建计划。

# 错误信号

最近更新时间：2022-03-25 15:14:43

本文为您介绍如何使用错误信号插件中断构建过程。

## 前提条件

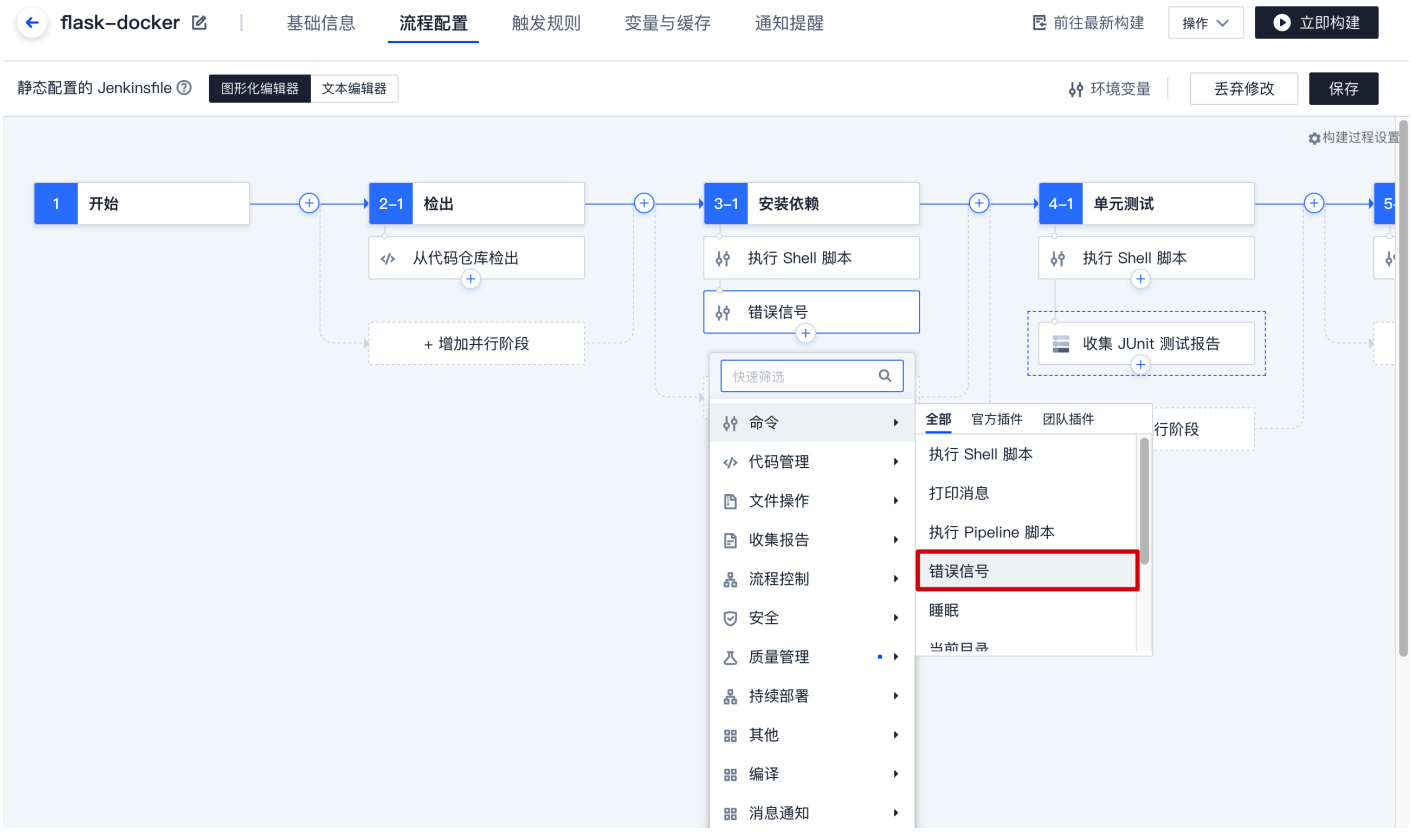
设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

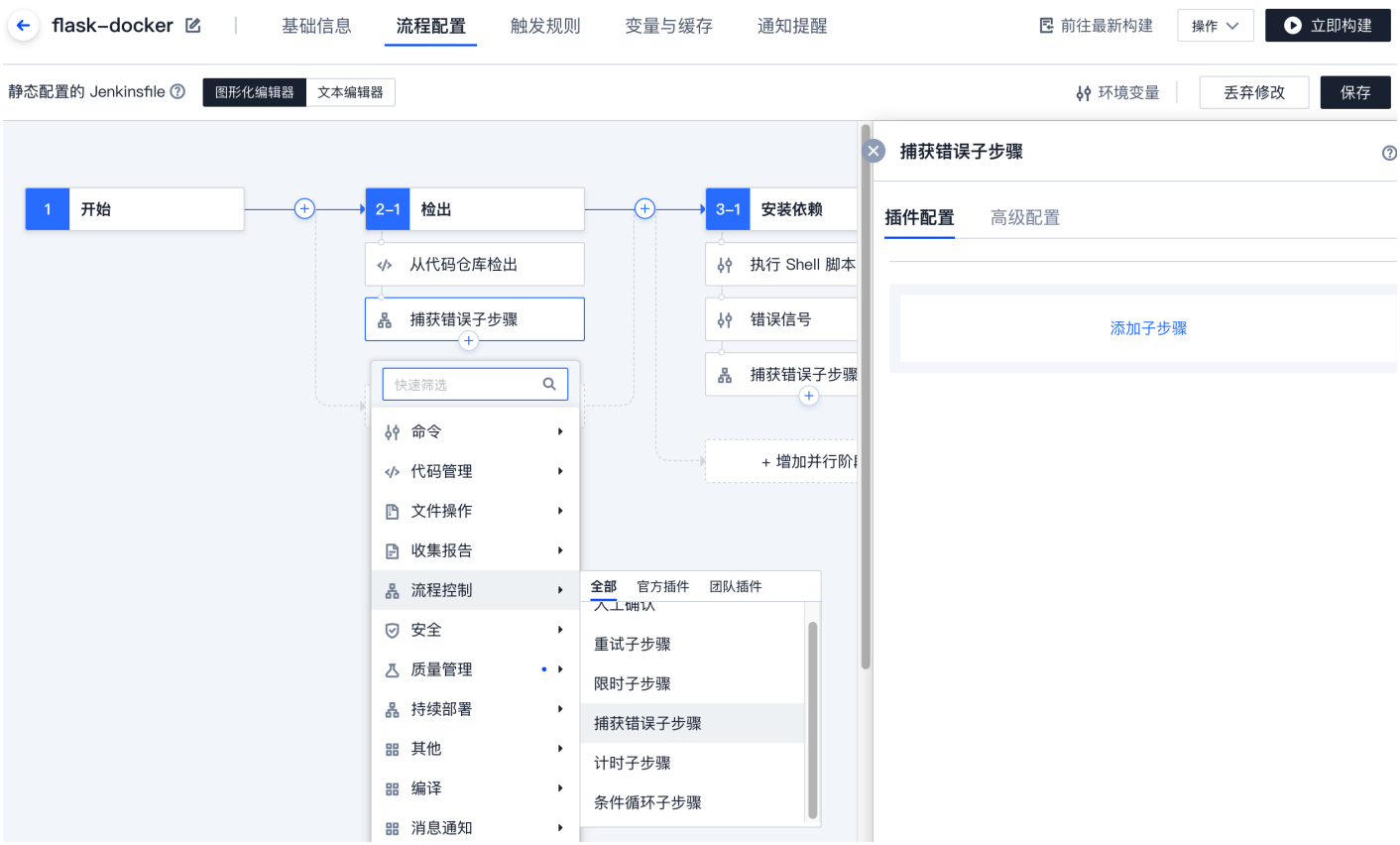
1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

## 功能介绍

- 持续集成的**错误信号**步骤可以理解为构建的终止符，运行至此步骤后将停止余下步骤，直接中断构建过程。




- 在持续集成中添加**捕获错误子步骤**，能够将运行结果作为是否中断持续集成任务的信号。若成功运行，则继续执行余下步骤，即使失败也将执行余下步骤，但构建任务会被判断为失败。


flask-docker
基础信息
流程配置
触发规则
变量与缓存
通知提醒
前往最新构建
操作
立即构建
静态配置的 Jenkinsfile
图形化编辑器
文本编辑器
环境变量
丢弃修改
保存
1 开始
2-1 检出
从代码仓库检出
捕获错误子步骤
3-1 安装依赖
执行 Shell 脚本
错误信号
捕获错误子步骤
+ 增加并行阶
快速筛选
命令
代码管理
文件操作
收集报告
流程控制
安全
质量管理
持续部署
其他
编译
消息通知
全部
官方插件
团队插件
重试子步骤
限时子步骤
捕获错误子步骤
计时子步骤
条件循环子步骤
捕获错误子步骤
插件配置
高级配置
添加子步骤

# 邮件通知

最近更新时间：2022-03-25 15:14:48

本文为您介绍如何使用邮件通知插件及时获取构建状态。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

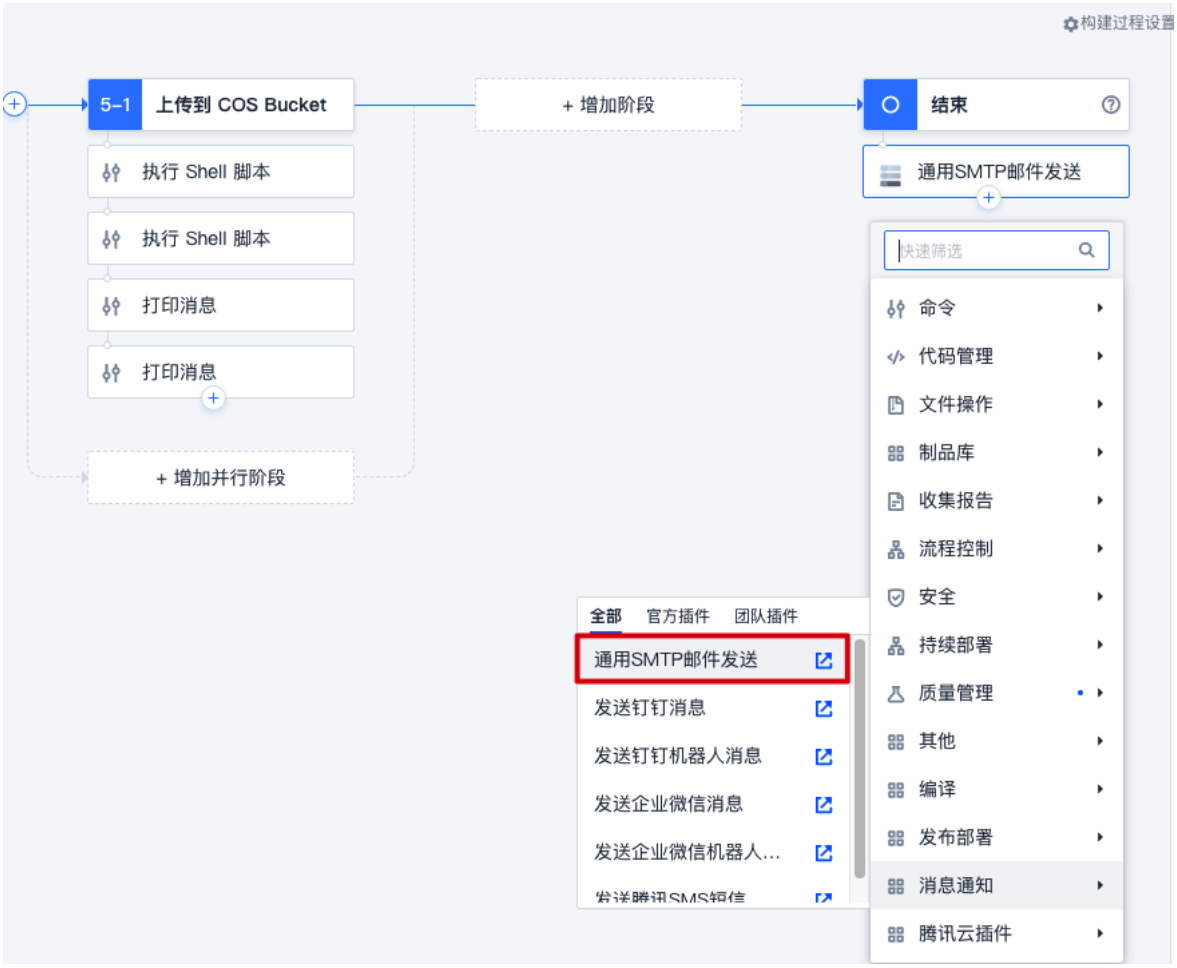
## 功能介绍

若想要知悉重要的持续集成任务是否成功运行，可以在构建流程中添加 SMTP 邮件通知插件。任务结束后发送邮件至指定人员以及时查看本次构建所涉及的详细参数。本文将演示如何在构建失败时自动发送邮件通知。

您还可以阅读 第三方服务通知 了解如何使用微信、企业微信、企业微信群机器人、钉钉机器人接受通知信息。

## 添加插件

1. 单击持续集成设置，在流程配置中使用图形化编辑器在结束阶段中添加 SMTP 邮件发送插件。




2. 填写邮件标题，在正文中可以添加环境变量以说明此次构建中所生成的关键信息。例如在本文中使用了 ${GIT_REPO_URL} 与 ${GIT_BUILD_REF} 环境变量分别说明代码仓库地址与 Git 修订版本号。


> **说明**
> **发件人邮箱授权码与服务地址请咨询邮件服务提供商。**

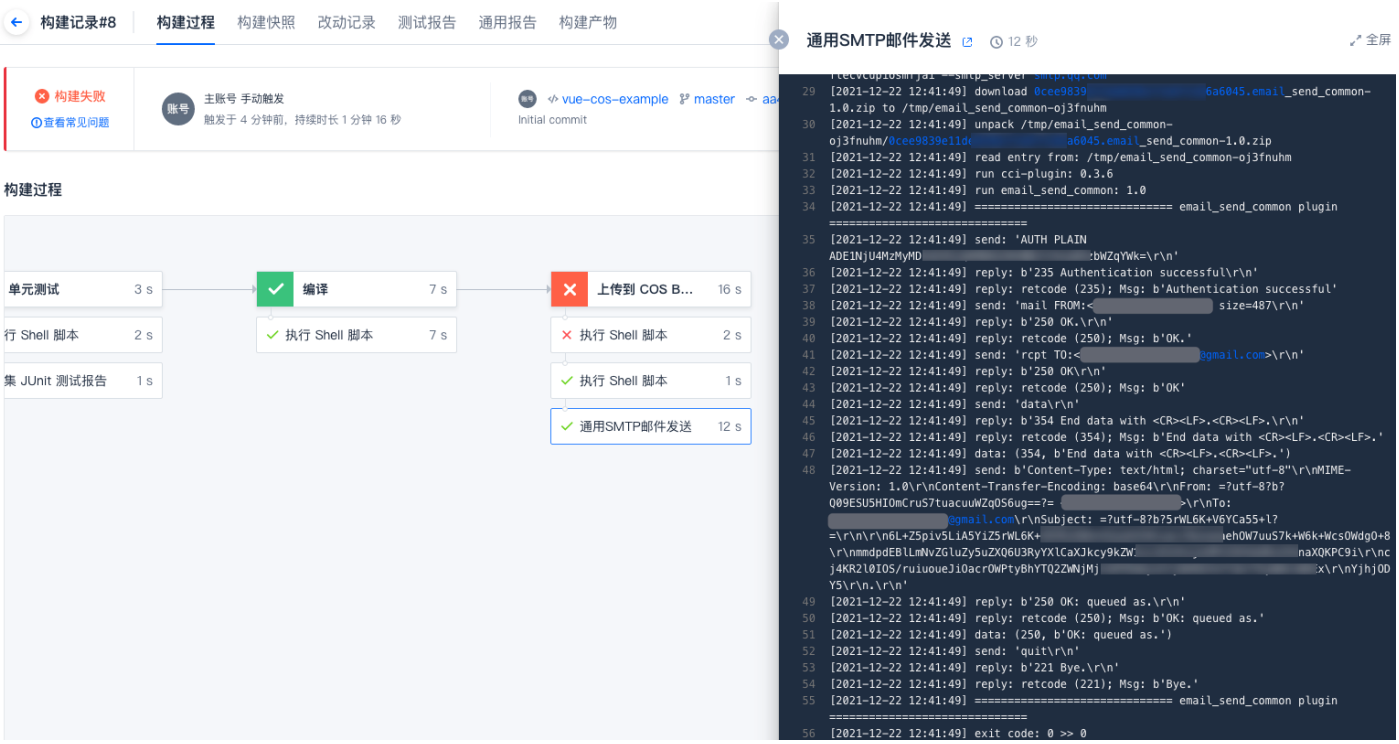
**单击阅读 环境变量 获取更多系统内置环境变量以增加邮件中需要通知的信息，如提交作者邮箱、提交者名称等信息。**

## 条件执行

通过在结束阶段中的高级配置启用 置底执行，您可以为邮件通知的启用触发条件，例如在本文中我们设置当本次构建任务运行失败时再进行邮件通知。


## 通知结果

1. 触发持续集成后，当任务运行失败后将启用邮件发送步骤进行提醒。



2. 邮件样式如下，您也可以通过 html 语言自定义邮件样式。

CODING 邮件机器人
发送至 我

15:13 (15分钟前)

# 自定义团队插件
# 产品简介

最近更新时间：2023-06-06 16:35:08

本文为您介绍自定义团队插件功能。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

## 功能介绍

实际的研发过程中，各团队所需要的插件能力各异，自定义团队插件功能能够让成员编写并发布插件。将内部得心应手的工具或命令封装成**自定义插件**，方便团队内其他项目与成员快速复用。

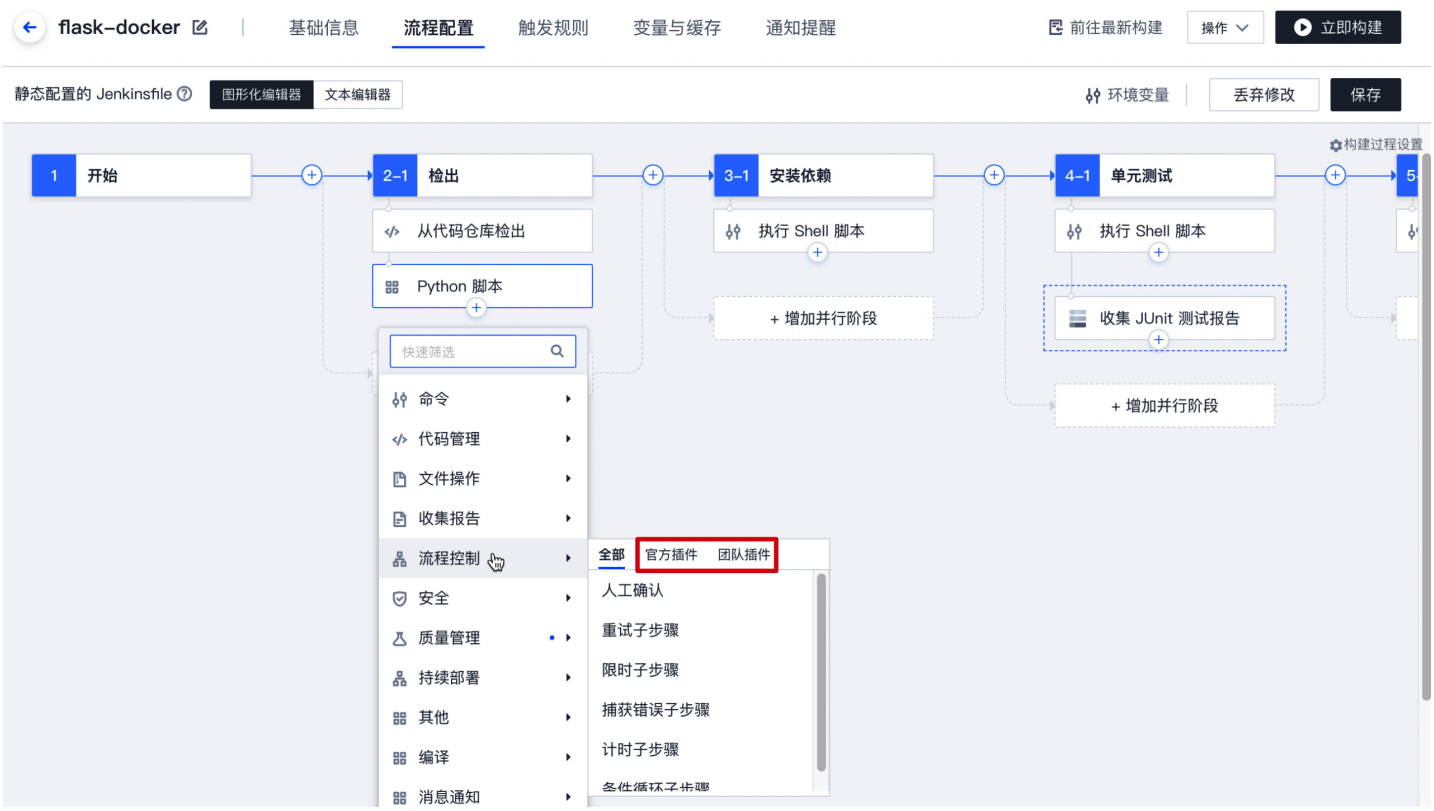

## 基础使用

团队首页右上角的齿轮图标 进入团队设置中心，单击**功能设置** > **构建插件**，您可以在此处看到官方插件、团队插件与个人提交但未公开的插件。在插件详情中查看名称、描述与版本号。


**您可以通过流水线编排界面或编译命令行两种方式使用插件功能。**

**图形化编辑器**

**在阶段内添加步骤时选择是否启用官方插件或团队自定义插件。**




**文本编辑器**

**支持使用命令行的方式直接编辑 Jenkinsfile，参考语句示例如下：**

```
useCustomStepPlugin(key: 'exec_py_script', version: '1.0', params: [site_packages:'false',requirements:'false'])
// key 为 插件的 ID，version 为版本号（默认使用最新版本，随插件升级而自动升级），params 为当前插件所需要填写的参数。
```

## 插件开发toolkit

插件开发并不限制语言与环境，仅需满足声明文件的规范要求即可上传至 CODING，通过校验后就能在持续集成任务中使用，单击了解 开发指引。
若您希望更高效的进行插件开发，系统亦提供了一套便利的开发工具 qci-plugin，详情请参见 qci-plugin。

## 插件上传upload

1. 插件开发不限制语言，但需保证插件逻辑能够以命令行的方式运行，并且 CI 任务的构建环境需具备插件的运行环境，单击了解如何配置 构建环境。
2. 开发完成后，遵循声明格式编写插件的描述文件，打包插件代码后以 ZIP 包形式上传至 CODING。当声明文件通过检查并进行二次确认发布后，当前插件将以私有插件的状态进入可选列表，在流水线编排时仅作者才能添加与使用。


3. 在实际的构建计划中完成调试后，作者可以将此插件标记成公开，此时团队内其他成员也将可以看到并使用此插件。

# 开发指引

最近更新时间：2022-05-31 17:10:00

本文为您介绍如何开发自定义团队插件。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

## 目录结构

插件结构范例：

```
- my-plugin-project // 您的项目目录
- my-script.xx // 构建插件执行脚本或入口文件，支持任意语言组织（需执行环境具备，如需特殊环境可使用容器）
- qciplugin.yml // 构建插件声明文件，定义您的构建插件名称、版本、参数等信息
```

以上两个文件为必填项。qciplugin.yml 为 声明文件，用于声明插件启动时执行该脚本，插件执行器会在运行时读取该文件，确定插件的参数要求及执行方式。插件文件不限制开发环境与语言，目录下的其它文件和子目录结构无其他要求。

> 注意：
> 插件命名请避免使用下划线（_）与点号（.）开头。

## 机制说明

插件文件需读取 声明文件 qciplugin.yml 的内容并使用两部分配置：

- variables：用于确定从当前插件需要使用者传入哪些参数，只有声明在 variables 里的参数才会在流水线编排时提供输入表单。
- entry：用于确定插件要怎样执行脚本命令，并且会将 variables 里收集到的参数，根据 entry 里声明的占位符进行填充。

假设开发者制作了一个命令行脚本，并接收 -input 参数，执行完成后输出 status.json 文件：

```
python ./run.py --input "hello world" --status ./status.json
```

那么 qciplugin.yml 文件的配置声明示例如下：

```
id: myplugin

variables:
- name: input
type: text
label: 输入内容
required: true

entry:
#------------------- 程序入口 -----------------# #- 参数占位符 -# #---------------- 固定参数 ---------------#
start: $QCI_PLUGIN_EXECUTABLE $QCI_PLUGIN_RUNTIME/run.py $input --status $QCI_PLUGIN_RUNTIME/status.json
```

| 参数 | 介绍 |
| --- | --- |

| | |
|---|---|
| variabels | 参数声明启动插件时，需要一个 input 参数，运行时需要将 input 参数值替换 start 节点中的 $input 位置，实际上是替换 -input “value” 并执行 start 命令。 |
| -status | 是一个固定参数(不需要用户传入)，允许直接填写。 |
| $QCI_PLUGIN_EXECUTABLE | 是一个指向在 qci-plugin 中运行 Python 命令的特殊值，用来避免环境中可能出现不同的 Python 版本所可能导致的不统一。 |
| $QCI_PLUGIN_RUNTIME | 是一个特殊的值，用以指向插件目录。因为 qci-plugin 在执行机中，pwd 指向的是集成目录而非插件安装目录，故使用 “./“ 这类相对目录不能等同于插件所在目录，需要使用 $QCI_PLUGIN_RUNTIME 替代插件目录从而找到正确的命令行脚本。 |

当运行以下命令时：

```
qci-plugin myplugin --input "hello world"
```

实际执行的命令如下（以执行环境为准）：

```
/usr/bin/python3 /data/__qci/__qci_tool_path/mysql-sjbh2b3/run.py --input "hello world" --status /data/__qci/__qci_tool_path/mysql-sjbh2b3/status.json
```

## 轮询结果

部分插件调用涉及第三方服务，需要长时间轮询结果。CI 任务运行时有超时限制，如果需要长时间轮询请参考下文对插件进行改造。

### 向 server 提交取消超时请求

您可以在 qciplugin.yml 上声明 timeout: false：

```
# 插件版本, 用于定义插件包的版本, 做版本管理使用
version: '1.0'

# 插件ID, 必填
id: plugin_demo

# 插件中文名称
name: 插件 demo

# 添加此配置, 取消 server 超时检查
timeout: false
```

若希望细粒度的控制，可以在轮询声明中使用 qciplugin 的 SDK：

```
from qciplugin import run_forever_context

# 使用 with 语法, sdk 会帮助您处理所有事情
with run_forever_context():

# 轮询
while True:
do_something()
```

### 保证轮询过程中有内容输出

CI 任务在运行时会检查一段时间内脚本是否有输出，无输出则中断执行，请确保在轮询或等待过程中一直有内容输出：

```
import time
from qciplugin import run_forever_context
```

```
# 使用 with 语法, sdk 会帮助您处理所有事情
with run_forever_context():

# 轮询
while True:
print('.') # 打印一些内容
do_something()
time.sleep(30) # 等待 30 秒再次查询结果
```

## 状态上报

插件支持设置单个 status 节点指向 status.json 文件。插件执行完毕后，执行器会读取里面的内容。status.json 可以包含几个部分：

| 参数 | 是否必选 | 说明 |
| --- | --- | --- |
| status | 可选 | 可以是”success”，“failure”，“error”。<br>• success：对应 CODING CI 状态为 “成功”。<br>• failure：对应 CODING CI 状态为 “失败”，例如检查出错。<br>• error：对应 CODING CI 状态为 “命令异常”，例如工具本身的错误。 |
| status_code | – | 失败的状态码，可以由开发者自己定义。 |
| description | 可选 | status 描述，支持 markdown 格式。 |
| url | – | 指向一个外部的 url 链接。 |
| title | 可选 | 请尽量简明。 |
| report_di | 可选 | 指向要上传到文件服务器的路径。 |
| report_html | 可选 | 指定 report_dir 时, 链接的首页文件。 |
| metrics | – | 元数据指标，如果有报元数据指标，需要包含这个字段。 |

示例：

- 单纯上报结果

```
{
"status": "success",
"title": "单击查看报告链接",
"url": "http://your.xxx.com"
}
```

- 失败时上报状态码

```
{
"status": "failure",
"status_code": 1001, // 自定义状态码, 便于统计
"title": "单击查看报告链接",
"url": "http://your.xxx.com"
}
```

- 上报本地文件

```
{
"status": "success",
"title": "单击查看报告链接",
"report_html": "./report/index.html", // report_html 是访问链接
"report_dir": "./report" // report_dir 上传整个目录, 需要跟 report_html 搭配使用
}
```

- 上报指标

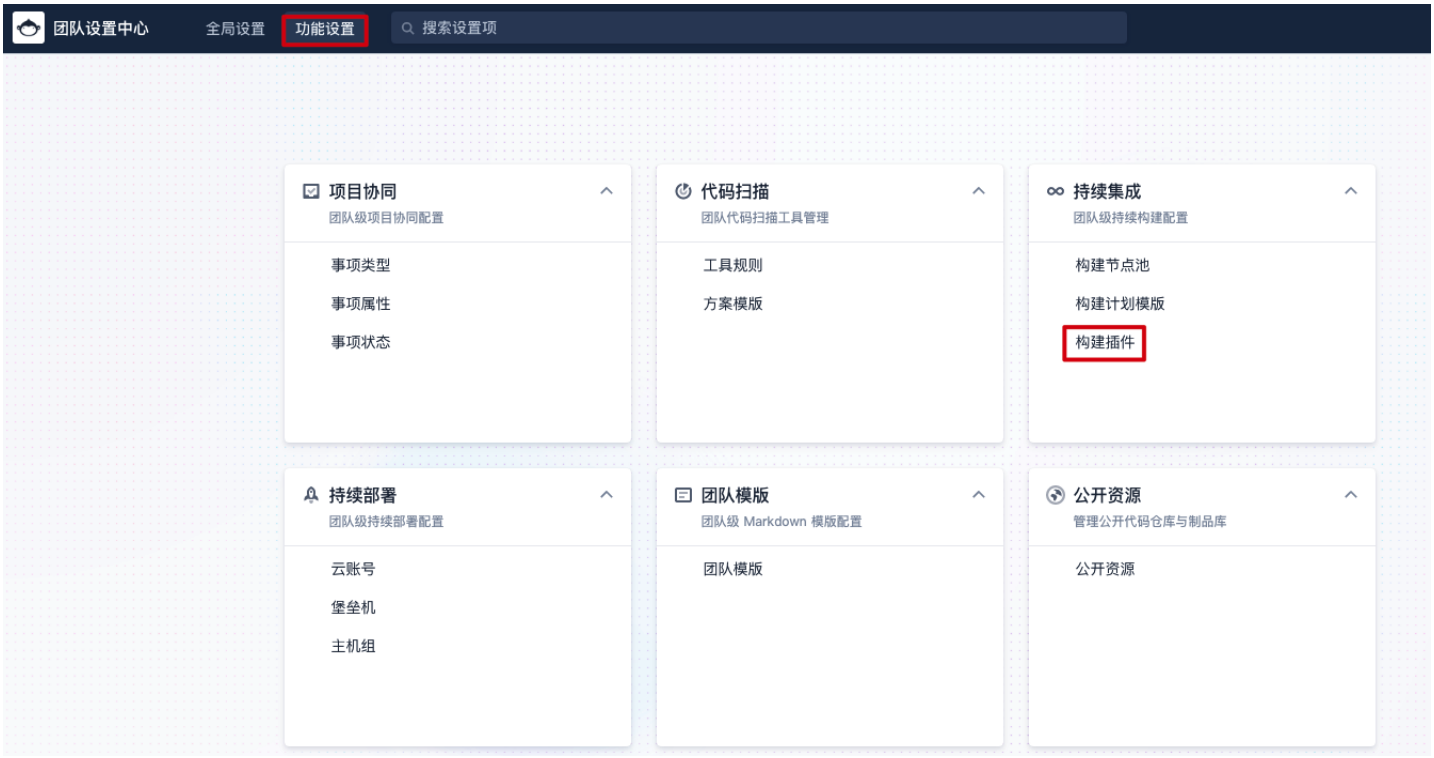
```
{
"status": "failure",
"status_code": 1001,
"metrics": {
"files": 100,
"errors": 8,
"warning": 10,
...
}
}
```

## 示例插件

您可以参考此公开 代码仓库 查看示例插件。

## 插件上传

单击团队首页右上角的齿轮图标 进入团队设置中心，在**功能设置** > **构建插件**中上传插件文件。



# 声明文件

最近更新时间：2022-03-25 15:15:04

本文为您介绍自定义团队插件中的声明文件。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

声明文件需命名为 qciplugin.yml，文件内容需包含插件的 ID，名称，参数，执行入口等配置信息。插件执行器将读取信息并上传至 CODING 后自动生成插件运行所需的 UI 表单，在运行时也需要读取信息并作为执行参考。

## 示例

```
# 插件版本, 用于定义插件包的版本, 做版本管理使用
version: '1.0'

# 插件ID
id: plugin_demo

# 插件中文名称
name: 插件 demo

# 插件描述
description: 插件 demo 描述

# 插件分类
category: build

# 声明插件使用的参数
variables:
- name: arg1
type: text
label: 参数1
help: this argument help description。
required: true
- name: arg2
...
- name: env1
...

# 执行入口配置, 声明如何运行插件脚本
entry:
# 插件执行依赖
install:
- $QCI_PLUGIN_EXECUTABLE -m pip install -r $QCI_PLUGIN_RUNTIME/requirements.txt --user

# 插件环境变量
env:
ENV: $env1
```

```
# 插件启动入口
start: $QCI_PLUGIN_EXECUTABLE $QCI_PLUGIN_RUNTIME/run.py $arg1 $arg2 --status $QCI_PLUGIN_RUNTIME/status.json

# 插件状态文件
status: $QCI_PLUGIN_RUNTIME/status.json
```

### 基本信息

插件的基本信息包括：

| 参数 | 说明 |
| --- | --- |
| version | 插件支持版本管理，版本号由开发者自行维护，必须是字符串。 |
| ID | 插件唯一标识，英文+下划线。 |
| name | 插件名称，请使用中文名称，会作为流水线 UI 配置的插件名。 |
| description | 插件简介，非必填，请简短介绍插件的用途，会作为流水线 UI 配置的插件 Tips。如需要展示更多插件相关的文档，请参考：插件文档一节。 |
| category | 插件分类，会在流水线 UI 配置中，进行分组，如果填写不在以下分类的值，可能会导致 UI 配置找不到插件。<br>• build：编译<br>• test：测试<br>• security：安全<br>• release：发布部署<br>• message：消息通知<br>• scm：代码管理<br>• other：其他 |
| author | 插件作者，非必填。 |

## variables

插件参数 variables 选项是描述流水线的 UI 配置时表单的渲染参考，插件执行器在启动时会根据选项处理参数。

> ⚠ 注意：
> 插件参数不等同于开发者自身脚本参数，虽然大部分情况下插件参数与执行参数相同。具体差别请参见 插件运行机制。

variables 选项是一个数组，使用 yaml 格式描述为：

```
variables:
# 请注意 "-" 以及缩进
- name: arg1
type: text
label: 参数1
help: this argument help description。
required: true
- name: arg2
...
- name: env1
...
```

variables 每个元素可以有如下属性：

- **name**（string）：参数名，必须是英文名+数字+下划线，作为 UI 配置以及执行器使用的参数名，在没有配置 resolve 参数时也默认为传入给插件脚本的参数名。
- **label**（string）：参数中文名，用于 UI 配置时展示。
- **type**（string）：参数类型，用于 UI 配置时渲染不同的 UI，同时也根据此类型，决定执行器传入插件脚本的行为：
  - text：文本类型，UI 默认使用单行文本框。
  - choice：选项类型，使用 choice 时，需跟 options 参数一并使用，UI 默认使用下拉框。

  - bool：布尔类型，UI 使用复选框。
- **widget**（string）：UI 组件类型，除了 type 的默认 UI 类型外，还可以根据 widget 的不同，设定不同的组件。
  - type == text，可以支持的 widget 参数：
    - input（默认）：单行文本框。
    - textarea：多行文本框。
    - multiline-textarea：（特殊）多行文本框，文本框每一行是一个参数，即参数用换行符分隔。
    - userchooser：用户选择器，可使用 options 设置选项。
  - type == choice，可以支持的 widget 参数：
    - select(默认)：下拉框。
    - checkbox：复选框。
    - radio：单选框。
    - remote-select：远程下拉框。
  - type == bool，不支持 widget 参数，UI 组件会生成一个复选框。
- **required**（可选，boolean）：是否必填。
- **default**（可选，string | boolean）：默认值。

> ⚠ 注意：
> 注意：只能使用字符串（type == text|choice）和布尔（type == bool）类型。

- **placeholder**（可选，string）：UI 组件的 placeholder，仅在 UI 为文本框时生效。
- **help**（可选，string）：帮助信息。
- **options**（可选，object | array）：UI 组件选项，根据 type 与 widget 的不同，可以有如下的配置：
  - type == choice 需要配置每个选项的 label 和 value，示例：

```
options:
- label: label1
value: value1
- label: label2
value: value2
```

  - type == text && widget == userchooser 可选配置，示例：

```
options:
singleton: true / false (是否单选/复选)
useEnv: true / false (是否可以输入环境变量)
```

- **resolve**（可选，string | null）：调整执行器传入给脚本参数名，默认是长参数，即：-{name}，可由开发者自行调整（由于可以随意定制字符串，请注意命令行参数规范）：
  - 长参数：-foo，这是默认设置。
  - 短参数：-f。
  - 无参数：~（注意是一个波浪号），传入参数会变化，具体参考 type 与传入参数值的关系中 bool 类型一节。
- **advanced**（可选，boolean）：是否是高级选项，advanced 设置成 true，前端会将此字段放到高级选项栏里，只有用户展开高级选项才能看到，适合插件参数太多，部分参数可以选填的情况。

## entry

插件执行入口 entry 选项是用于描述插件如何执行插件脚本的配置。
entry 选项的数据结构为**字典**，使用 yaml 格式描述为：

```
# 执行入口配置, 声明如何运行插件脚本
entry:
# 插件执行依赖
install:
- $QCI_PLUGIN_EXECUTABLE -m pip install -r $QCI_PLUGIN_RUNTIME/requirements.txt --user

# 插件环境变量
```

```
env:
ENV: $env1

# 插件启动入口
start: $QCI_PLUGIN_EXECUTABLE $QCI_PLUGIN_RUNTIME/run.py $arg1 $arg2 --status $QCI_PLUGIN_RUNTIME/status.json

# 插件状态文件
status: $QCI_PLUGIN_RUNTIME/status.json
```

- install（array）：一个数组，声明插件脚本运行前需要执行的命令，可以使用 variables 定义的参数占位符 $var。
- start（string）：声明插件脚本运行命令行，可以使用 variables 定义的参数占位符 $var，如果有固定参数可以一并写入。
- env（可选，object）：插件环境变量，可以使用 variables 定义的占位符 $var，可以让 variables 里定义的参数通过环境变量提供给插件脚本，而不是参数。
- status（可选，string）：插件脚本可以写入 status.json 决定程序是否执行成功，如没有此节点，会根据程序的退出码判定。

关于插件如何运行请参见 开发指引。

## 如何使用占位符

qciplugin.yml 里 install，start 节点均可以使用占位符做参数传入。占位符由 variables 定义的参数决定，没有声明 variabels 则占位符无效。占位符的行为会与 variables 中 type 和 widget 有直接关系。

- type == text 时。

```
# 定义
variables:
- name: arg1
type: text

entry:
start: python run.py $arg1

---

# 启动命令
qci-plugin myplugin --arg1 "hello world"
# 实际命令
python run.py --arg1 "hello world"

---

# 不传 arg1 时, 不会给脚本传入 --arg1
# 启动命令, 不传 arg1
qci-plugin myplugin
# 实际命令, 不传 arg1
python run.py
```

- type == text && widget == multiline-textarea 时，参数为多项参数。使用 Python 开发时可以参考文档。

```
# 定义
variables:
- name: arg1
type: text
widget: multiline-textarea # 注意这里

entry:
start: python run.py $arg1
```

```
---

# 启动命令
qci-plugin myplugin --arg1 "foo" "bar" "baz"
# 实际命令
python run.py --arg1 "foo" "bar" "baz"
```

- type == choice && widget == select|radio|remote-select 时。

```
# 定义
variables:
- name: arg1
type: choice
options:
- label: label1
value: value1
- label: label2
value: value2


entry:
start: python run.py $arg1

---

# 启动命令
qci-plugin myplugin --arg1 "hello world"
# 实际命令
python run.py --arg1 "hello world"
```

- type == choice && widget == checkbox(复选) 时，参数是多项参数，Python 开发时，参考使用 nargs=(+|*)。

```
# 定义
variables:
- name: arg1
type: choice
options:
- label: label1
value: value1
- label: label2
value: value2


entry:
start: python run.py $arg1

---

# 启动命令
qci-plugin myplugin --arg1 "value1" "value2"
# 实际命令
python run.py --arg1 "value1" "value2"
```

- type == bool 时，根据参数，直接标记，Python 开发时：参考中将 action="store_true"，即可获取到正确的参数值。

```
# 定义
variables:
- name: arg1
type: bool

entry:
start: python3 run.py $arg1

---

# 启动命令, 注意不带参数值
qci-plugin myplugin --arg1
# 实际命令, 注意不带参数值
python run.py --arg1

---

# 启动命令, 参数为 false, 不传 arg1
qci-plugin myplugin
# 实际命令, 参数为 false, 不传 arg1
python run.py
```

**如果 resolve 设置为~, bool 传入值有如下变化：**

```
# 定义
variables:
- name: arg1
type: bool
resolve: ~

entry:
start: python3 run.py $arg1

---
# arg1 = true 命令行传入 1
python run.py 1
# arg1 = false 命令行传入 0
python run.py 0
```

## 如何使用 env 环境变量

**entry 中允许开发者定义用户输入的值通过环境变量传入，便于开发者隐藏不希望通过命令行参数传入的值，同样可以使用 `$var` 占位符：**

```
variables:
- name: arg1
...
- name: arg2
...

entry:
env:
MY_ENV: $arg2
start: python run.py $arg1

---
```

```
# 插件启动命令为:
qci-plugin myplugin --arg1 value1 --arg2 value2

# 执行的命令行为:
python run.py --arg1 value1
```

而 arg2 的值，通过环境变量 MY_ENV 获得。

## 关于 status

status.json 文件允许开发者自行写入状态来覆写程序自身的退出码：

- 没有 status，脚本是否正常执行，是由程序的退出码控制：
  - 0：正常
  - 非 0：错误
- 有 status 则以 status 为准：

```
id: myplugin

variables:
- name: input
type: text
label: 输入内容
required: true

entry:
start: $QCI_PLUGIN_EXECUTABLE $QCI_PLUGIN_RUNTIME/run.py $input --status $QCI_PLUGIN_RUNTIME/status.json

status: $QCI_PLUGIN_RUNTIME/status.json
```

在 start 节点中, 通过参数传入 status.json ，脚本读取参数来指定写入路径并在 status 节点中指定该路径。插件安装目录不等同于集成目录，若隐式写 status.json，虽然插件也可以读取，但集成时容易出现并发冲突和重名问题。

若想了解更多 status 信息, 请参见 开发指引。

# qci-plugin

最近更新时间：2023-06-15 11:36:12

本文为您介绍 qci-plugin 工具详情。

## 前提条件

设置 CODING 持续集成中构建环境前，您的腾讯云账号需要开通 CODING DevOps 服务，详情请参见 开通服务。

## 进入项目

1. 登录 CODING 控制台，单击**团队域名**进入 CODING 使用页面。
2. 单击页面右上角的 ，进入项目列表页面，单击**项目图标**进入目标项目。
3. 进入左侧菜单栏的**持续集成**功能。

为提升插件开发的便利性，插件系统提供一套基于 Python 环境的开发工具 qci-plugin；它的设计理念是以一种简单的方式将构建代码与构建脚本分离，让构建脚本（编译、检查、测试、发布）得到尽可能的复用，具备以下特点：

- 基于 Python 3.x 环境，开发简单。
- 使用命令行方式执行，与开发普通的命令行（CMD）脚本基本无异。
- 可以通过简单的声明式实现在 CODING CI 渲染配置组件，便于流水线配置和编辑，同时也保留直接执行的能力。
- 支持结果上报。
- 支持个性化的结果展示。

借助此工具能够快速创建插件结构、运行和测试插件逻辑、提交插件，具体请参见 开发指引。

### 环境安装

- **依赖安装**：Python 3.6.8及以上，请参见 下载地址。
- **安装执行器**

  执行器的包已上传至 公开制品仓库，您可以通过命令行快速安装。

```
pip install qciplugin -i https://coding-public-pypi.pkg.coding.net/cci/qci/simple/ --trusted-host coding-public-pypi.pkg.coding.net --extra-index https://mirrors.tencent.com/pypi/simple/
```

  验证是否安装成功：

```
qci-plugin -h
```

### 运行机制

插件通过 qci-plugin 命令启动：

```
# 运行本地插件, 第一个参数为 qciplugin.yml 所在目录的路径
qci-plugin ./ --arg1 "hello world"

# 使用最新的公开版本运行, 第一个参数为 "插件 ID"
qci-plugin my_plugin_id --arg1 "hello world"

# 使用指定的版本运行
qci-plugin my_plugin_id@1.0 --arg1 "hello world"
```

### 本地调试

插件支持在本地直接运行调试。添加 QCI_PLUGIN_DEBUG=1 环境变量可以查看更多调试信息，运行中的警告提醒不会影响实际执行。

```
qci-plugin [local path] [arguments...]
```

```
// local path: 插件本地路径(指 qciplugin.yml 所在目录)
// arguments: 插件参数
```