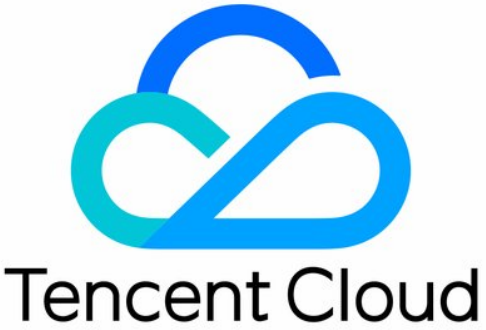
CODING DevOps

Continuous Integration



Contact Us

We are committed to providing personalized pre-sales consultation and technical after-sale support. Don't hesitate to contact us at 4009100100 or 95716 for any inquiries or concerns.

Contents

Continuous Integration
- Quick Start
- Compile Build Process
  - Text Editor
  - Process Configuration Details
  - Graphical editor
- Build plan configuration
  - Trigger rules
  - Environment variable
  - Build snapshot
  - Cache directory
- Build Node
  - Build Node Types
  - Default Build Environment
  - Custom Build Environment
  - From Definition Node
  - Worker Common Commands
  - Build Node Pool

Continuous Integration Quick Start

Last updated: 2024-09-05 16:30:12

The following text will demonstrate how to quickly deploy a project based on Node.js + Express + Docker using a CODING-CI template.
Watch video

Prerequisites

To use CODING-CI, you must activate the CODING DevOps service for your Tencent Cloud account. For more details, refer to Activate Service.

Open Project

1. Log in to CODING Console, click the team domain to enter the CODING page.
2. Click **Item** on the left side of the team homepage to enter the project list page and select the target project.
3. Select **Continuous Integration** from the menu on the left.

1. Create a build plan

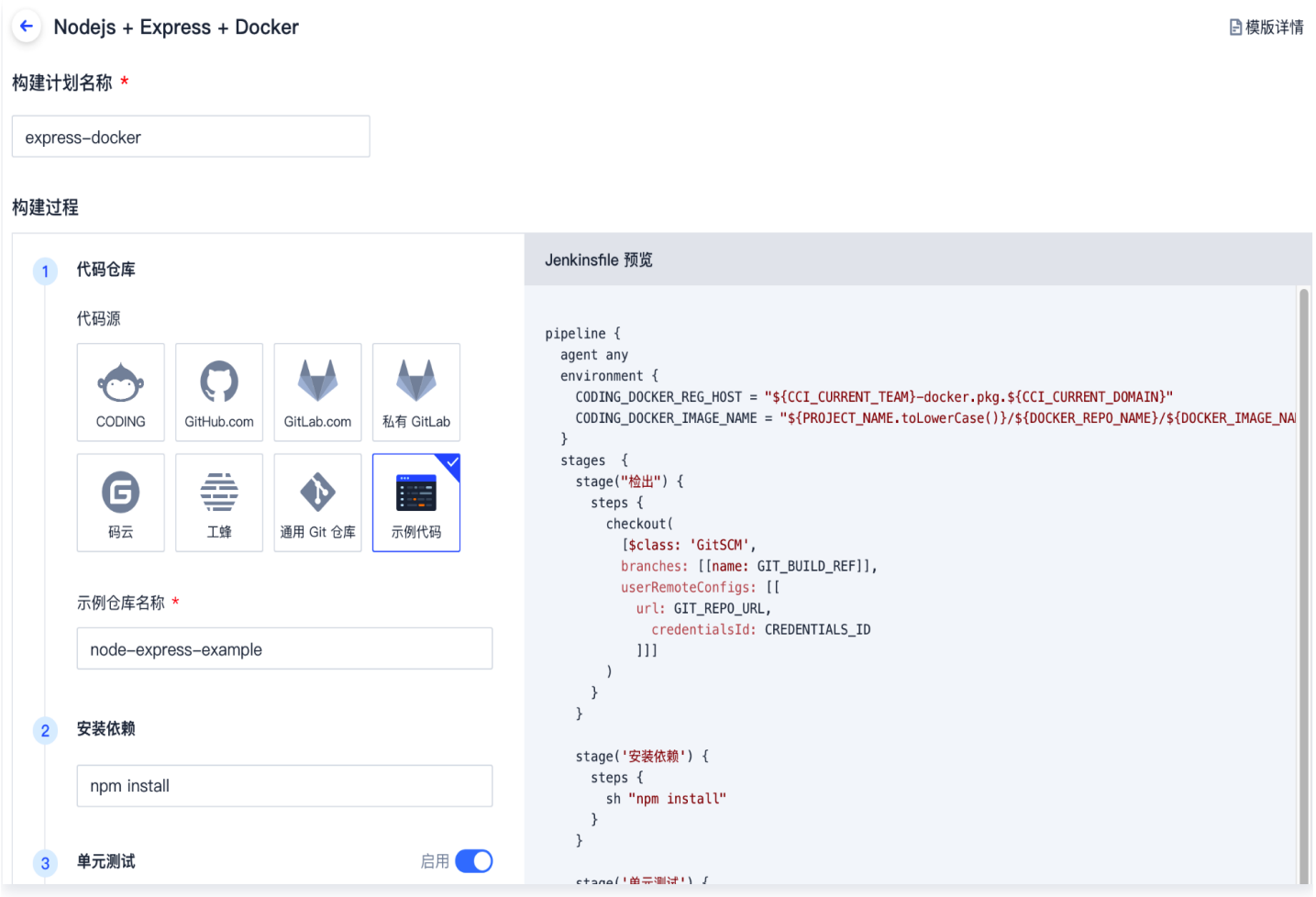
After entering the project, select **Continuous Integration > Build Plans** on the left, and click Create Build Plan. This plan will demonstrate how to fully automatically check out code > run unit tests > build a Docker image > push it to the Docker artifact repository > deploy it to a remote server (optional step) based on Node.js + Express.

2. Select a continuous integration (CI) template

Select a Node + Express + Docker continuous integration (CI) template.

3. Select code source

In the code repository field, select **Sample Code** as the code source. The system will automatically establish a sample code repository in your project.



4. Select an artifact repository

After the build plan is completed, the system generates a build result. Here, select the Docker artifact repository to push the results to. If there is no artifact repository in the project, you can quickly create one.

5. Enter remote server information (optional step)

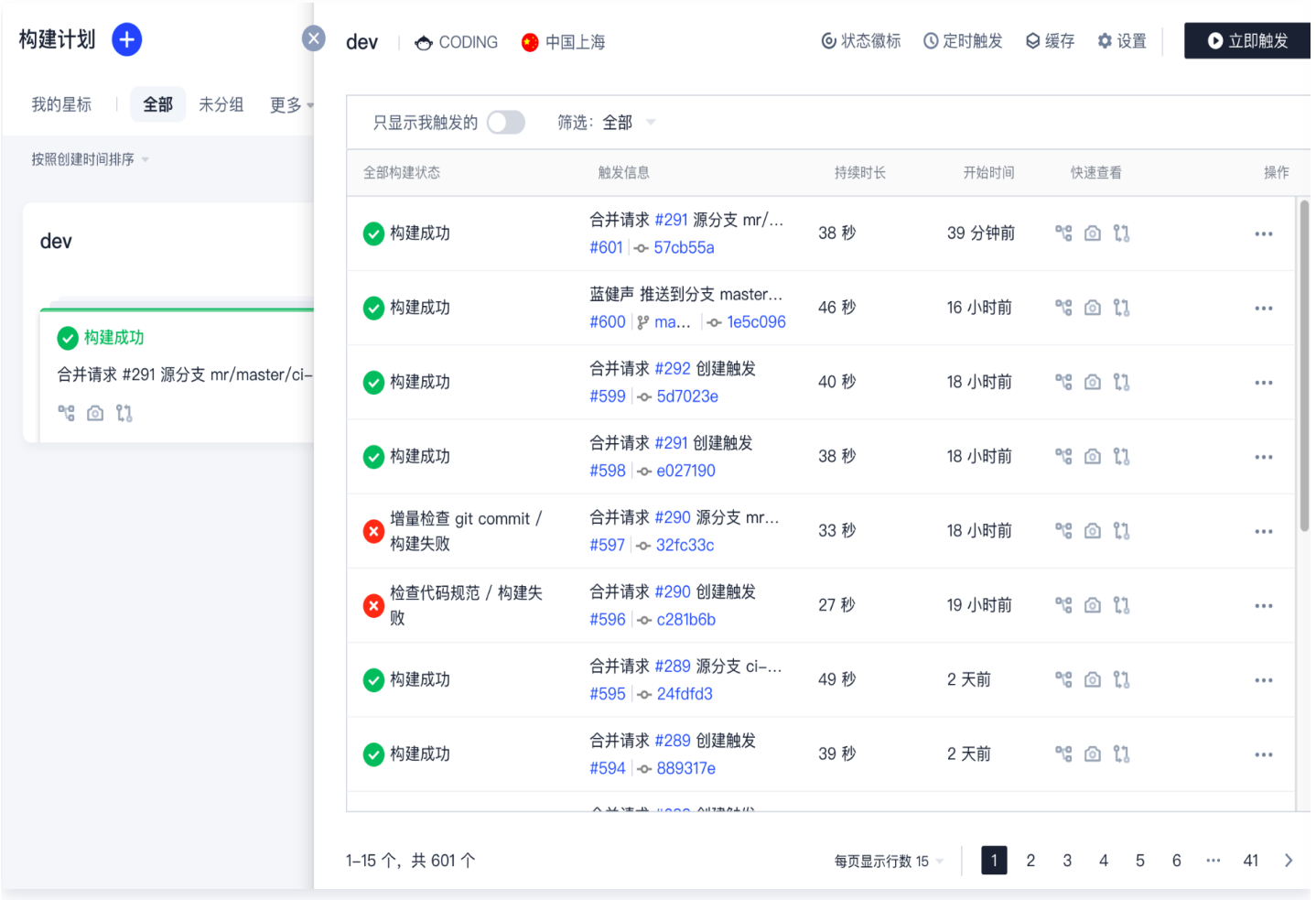
Enter the information of the remote server for deployment, including its IP address, port, and SSH login credentials. After ensuring the information is correct, wait for the build plan to complete. The artifacts will then be sent to the remote server, and you can preview the release result via a URL. If you do not need to deploy to a remote server temporarily, you can skip this step.

After clicking **Enter new credentials and authorize**, if you use an SSH private key to log in to the remote server, click **Manually enter existing SSH private keys** in the entry method. After entering the keys, view them in **Project Settings > Developer Options > Credential Management**.

If you do not know how to log in to a remote server using an SSH private key, click **Automatically create an SSH key pair** in the entry method, and manually set the public key in the target remote server's `~ssh/authorized_keys` folder.

6. Click create and view build result

Click **Confirm** to save the build plan. If **Trigger build after creation** is selected, the build plan will start immediately. During the execution of the build plan, you can view the build details on the build plan record list page.



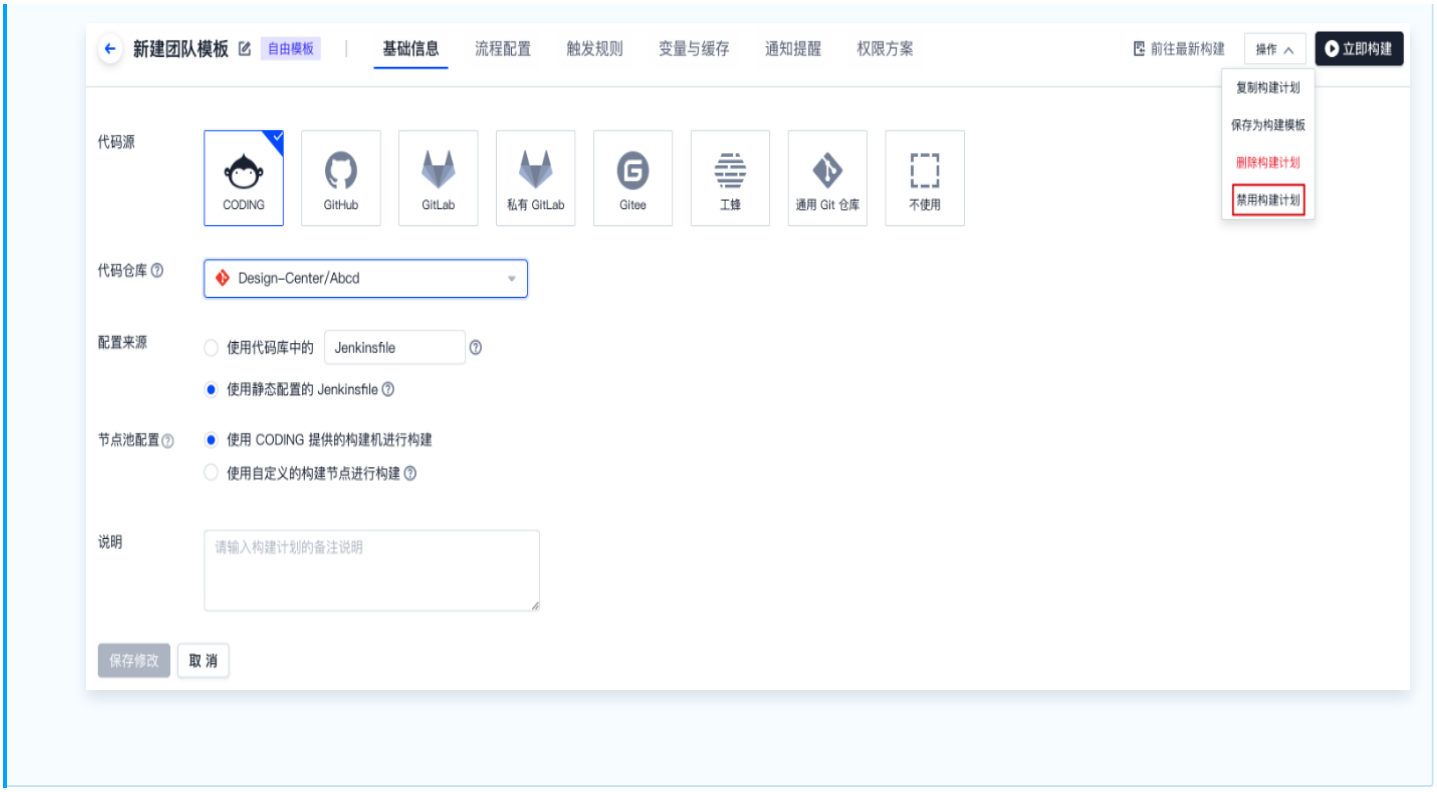
Click **Build Records** to check if each phase in the pipeline is successful, and also see the execution results and logs for each step command.

If Step 5 runs normally, you can see the artifact output URL in the build plan.

7. Modify remote server information

In Step 5, you have already configured the remote server address and SSH key. To change this, go to **Continuous Integration Plan > Settings > Variables and caching**.

Note:
After the build plan is successfully created, you can disable the build plan if it is no longer needed. Disabled build plans will not be triggered, and they can be re-enabled for normal operation.



Compile Build Process Text Editor

Last updated: 2024-09-05 16:30:28

This article introduces how to use the text editor in Continuous Integration.

Prerequisites

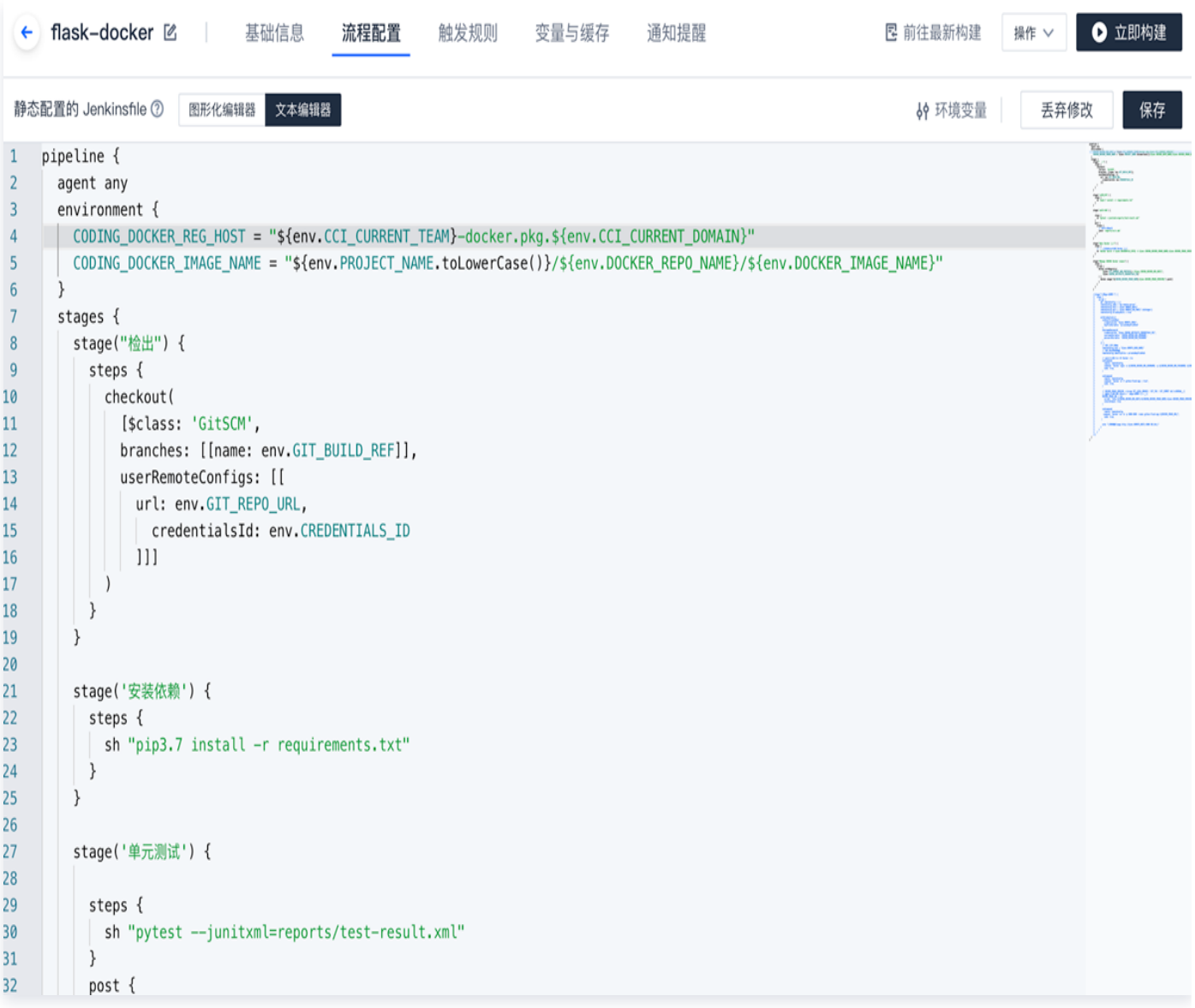
Before configuring the CODING-CI build environment, you must activate the CODING DevOps service for your Tencent Cloud account. For details, please refer to Activate Services.

Open Project

1. log in to CODING console, click **Team Domain** to enter the CODING page.
2. After entering the target project, click **Continuous Integration** on the left, and click **Settings** of the build plan.

   In essence, build tasks follow the processes and steps defined in configuration files. CODING Continuous Integration (CODING-CI) is fully compatible with Jenkinsfiles. Configuration files composed in the text editor can be run as long as they follow the syntax specifications of Jenkinsfiles.

3. Click the **Process configuration** and then select **Text Editor**.



Follow the syntax for Jenkinsfile in the build process. For more information, refer to the following documentation:

- Official Jenkinsfile Documentation
- Configuration Details

Process Configuration Details

Last updated: 2024-09-05 16:30:42

This article is primarily used to assist in the writing of the build process and explain the parameter details of each step.

Prerequisites

Before configuring the CODING-CI build environment, you must activate the CODING DevOps service for your Tencent Cloud account. For details, please refer to Activate Services.

Open Project

1. log in to CODING console, click **Team Domain** to enter the CODING page.
2. After entering the target project, click on the left side **Continuous Integration**, click **Build Plan > Text Editor** and arrange within it.

Code Repository

Git

Used to check out the Git repository source code of the current project. This command is a simplified version of the checkout command.
Parameter List:

Parameter	Type
Git Address `url`	string
Branch `branch`	string
Change Log `changelog`	string
Identity Authentication ID `credentialsId`	string
Poll `poll`	boolean

Check out from Version Control

General checkout SCM (Git, SVN) code.
This step returns content in Map Format. For example, if you use Git, you can do this:

```
def scmVars = checkout scm
```

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

The parameter `scm` is a configurable SCM type object. The currently supported ones are as follows:

- GitSCM Example Usage:

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

userRemoteConfigs Parameter List:

Parameter	Type
locations	Object array.
remote	string.
credentialsId	string.
local	string, specify a local directory (relative to workspace) as the location for checking out code.
depthOption	string, corresponds to the content of --depth. The default value is infinity. For details, refer to Subversion.
ignoreExternalsOption	boolean.

- SubversionSCM checks out code from the SVN server. Example syntax:

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

Parameter List:

Parameter	Type
locations	Object array.
remote	string.
credentialsId	string.

local	string, specify a local directory (relative to workspace) as the location for checking out code.
depthOption	string, corresponds to the content of --depth. The default value is infinity. For details, refer to Subversion.
ignoreExternalsOption	boolean.

Build Process

Subnode

Parameter List:
`label` : type string Environment Tag name, for example, java-8.

Collect Build Artifacts

Collect build results (e.g., jar, war, apk, etc.). Note that the collected build artifacts will be saved and deleted along with this build history. This is only a temporary storage space. It is recommended to use "Build Management" for versioning build results.
Parameter List:

Parameter	Type
artifacts	string, wildcards * can be used to specify the path pattern of files to be collected, conforming to the Apache Ant path rule. Only files within the workspace can be specified.
allowEmptyArchive	boolean, optional; under normal circumstances, this directive will cause the build to fail if no files matching the collection pattern are found. If this option is set to true, the build process will issue a warning instead of failing when no build artifacts are found.
caseSensitive	boolean, optional; by default, the matching of file path rules is case sensitive. If set to false, case sensitivity will be ignored.
defaultExcludes	boolean, Optional.
excludes	boolean, Optional; You can exclude some files within the specified path mode, also supports Apache Ant path rules
fingerprint	boolean, Optional; Calculate the file's hash information at the time of collection.

onlyIfSuccessful	boolean, Optional; Collect only if the build is successful.

Execute Shell Script

Execute a Shell script, example:

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

Collect JUnit Test Report

Collect JUnit and TestNG test reports (XML Format). You can specify which XML files to collect, for example `**/build/test-reports/*.xml` . Note that XML files that are not report files should not be included. You can use commas to separate multiple rules.
Parameter List:

Parameter	Type
testResults	string.
allowEmptyResults	boolean, Optional, allows the test report file to be missing or empty.
keepLongStdio	boolean, Optional, all test logs will be retained, even those of passed test cases.

Other

Change Directory Sub-steps

Change directory sub-steps. You can fill in several sub-steps within the DIR block, and these sub-steps will be executed within the specified path directory.
Parameter List:

`path` : type string.

Sleep

Pause for a period of time until the specified deadline. Similar to Unix's `sleep xxx` .
Parameter List:

- `time` : Type: int.
- `unit` : Can only select one from `NANOSECONDS` , `MICROSECONDS` , `MILLISECONDS` , `SECONDS` , `MINUTES` , `HOURS` , `DAYS` .

Error

Send an error signal, often used when it is necessary to terminate part of the execution process based on conditions. You can also use throw new Exception(), but using the error step can avoid printing an excessively long exception stack.
Parameter List:
`message` : Type: string.

Current Directory

Return the current directory path as a string.
Parameter List:
`tmp` : Type: boolean, optional. If selected, it returns a temporary directory associated with the workspace. Typically used when you want to store some temporary files without contaminating the workspace directory.

Write File

Write the specified content to a file. Parameter List:

- `file` : Type: string.
- `text` : Type: string.
- `encoding` : Type: string. Specifies the file encoding. If left blank, the default encoding of the current runtime platform will be used. For binary files, it will automatically be returned as a Base64-encoded result.

Read File

Read a file from a relative path and return its content as a string. Parameter List:

- `file` : Type: string. Relative path address (relative to the workspace directory).
- `encoding` : Type: string. Specifies the file encoding. If left blank, the default encoding of the current runtime platform will be used. For binary files, it will automatically be returned as a Base64-encoded result.

Retry substeps

Retry a specified block until the maximum retry limit is reached. If it ends normally during execution, it will not retry. If an abnormal termination occurs, it will keep retrying until it reaches the maximum retry limit. If the last attempt fails, the build process will be terminated.
Parameter List:
`count` : Type: int.

Time limit substeps

Execute the process within the block with a time limit. If time runs out, an `org.jenkinsci.plugins.workflow.steps.FlowInterruptedException` will be thrown. The unit parameter is optional and defaults to minutes. Parameter List:

- `time` : Type: int.
- `activity` : Type: boolean. Time is measured based on the absence of new log content instead of absolute execution time.
- `unit` : Can only select one from `NANOSECONDS` , `MICROSECONDS` , `MILLISECONDS` , `SECONDS` , `MINUTES` , `HOURS` , `DAYS` .

Catch error substeps

Errors in the specified substeps will be caught.

Timer substeps

Execution time of the specified substeps will be recorded as Unix timestamps.

Loop substeps

The specified substeps will be executed repeatedly for a defined number of times.

Conditional loop substeps

The specified substeps will be executed repeatedly until the result of the substeps returns `true` .

Print message

Print a message in the log.
Parameter List:
`message` : Type: `string`

Execute arbitrary Pipeline script

Adding this step allows you to execute any Pipeline script.

Execute Groovy source file

The build process will execute the Groovy source file at this location.
Example:

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

Execute Yarn Audit

This step performs yarn audit in the specified directory, and the vulnerabilities of yarn dependencies can be seen on the Continuous Integration results page.
Parameter List:

- `directory` : Type: string, optional. Fill in the directory where yarn.lock is located. By default, it executes at the project root directory.
- `collectResult` : Type: boolean, optional. Collect the Yarn Audit report.

Execute Npm Audit

This step performs npm audit in the specified directory, and the vulnerabilities of npm dependencies can be seen on the Continuous Integration results page.
Parameter List:

- `directory` : Type: string, optional. Fill in the directory where package.json is located. By default, it executes at the project root directory.
- `collectResult` : Type: boolean, optional. Collect the Npm Audit report.

Merge Merge Request

Merge Code. You can merge a specified merge request in this step.
Parameter List:

- `token` : Type: string. Project token.
- `depot` : Type: string. Repository name.
- `mrResourceId` : Type: string. Specify Resource ID.
- `commitMessage` : Type: string. Merge commit message template.

- `deleteSourceBranch` : Type: boolean, optional. Delete source branch.
- `fastForward` : Type: boolean, optional. Attempt fast-forward merge.

Review Merge Request

Review Merge Request. You can comment on a specified merge request in this step.
Parameter List:

- `token` : Type: string. Project token.
- `depot` : Type: string. Repository name.
- `mrResourceId` : Type: string. Specify Resource ID.
- `commentContent` : Type: string. Comment content template.

Graphical editor

Last updated: 2024-09-05 16:30:55

This article introduces how to use the Graphical Editor in the Build Plan settings.

Prerequisites

Before configuring the CODING-CI build environment, you must activate the CODING DevOps service for your Tencent Cloud account. For details, please refer to Activate Services.

Open Project

1. log in to CODING console, click **Team Domain** to enter the CODING page.
2. After entering the target project, click on the left side **Continuous Integration**, click **Build Plan > Text Editor** and arrange within it.

Description of the Feature

Editing a Jenkinsfile (a file that describes a build process) using a command-line editor is the most basic mode of human-computer interaction. Based on its core text editing function, CODING has been designed with an innovative graphical editor that is compatible with most custom command-line operations. Enjoy an intuitive WYSIWYG editing experience as you can view while building.
In **Build Plan Settings > Process Configuration**, select **Graphical Editor** to use.

Build Process Concepts

Regardless of being graphical or textual, editors are fundamentally designed to facilitate users in viewing and editing the core of the build process— Jenkinsfile (Process Description File). Therefore, understanding the important concepts of the "Process Description File" is necessary before discussing editors.
This document focuses on the syntax rules for declarative files.

Pipeline

The pipeline is a self-definition work model. It defines the complete process of delivering software, generally including phases such as build, test, and deployment.

Execution Environment

The execution environment describes the execution environment of the entire process or a certain stage of executing a pipeline. It must appear in the top grid of a descriptive file or at every stage.

Is it necessary	Yes
Parameter list	See below
Permitted location	Must appear at the top of the description file or in each phase

Stage

A stage defines a series of closely related steps. Each stage takes on an independent and clear responsibility within the entire pipeline. For example, "Build stage", "Test stage", or "Deployment stage". Generally speaking, all the actual build processes are placed within these stages.

Is it necessary	At least one
Parameter list	A required string parameter that specifies the name of a stage
Permitted location	Inside the stage block

Stage List

The stage list includes a series of phases, with at least one phase in a stage list. There should be one and only one stage list in the pipeline.

Is it necessary	Yes
Parameter list	No

Permitted location	Can only appear once in the pipeline

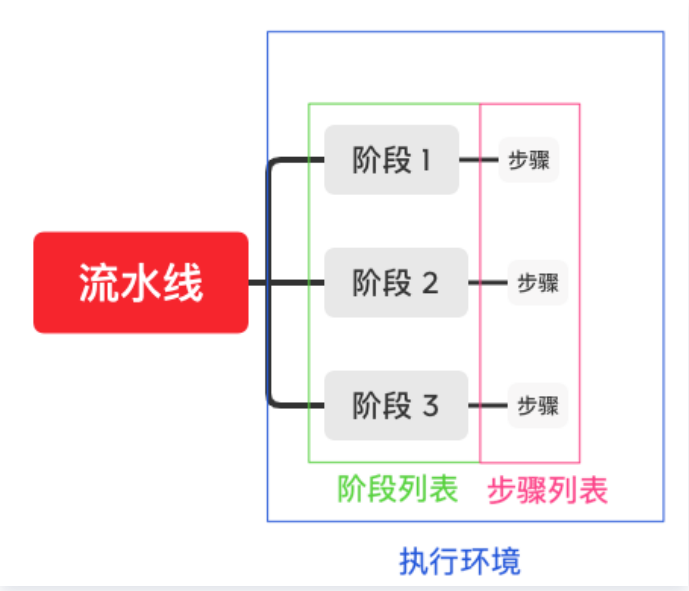
Step List

The step list describes what exactly needs to be done within a stage, and which commands need to be executed. For example, there is a step that requires the system to print a "Building..." message, which is executed with the command echo ' Building...'.

Is it necessary	Yes
Parameter list	No
Permitted location	In every stage block

Parallel

Parallel is used to declare several stages that will execute concurrently, typically suited when there are no dependencies between these stages to speed up execution. Note that stages within any parallel block cannot set an execution environment.



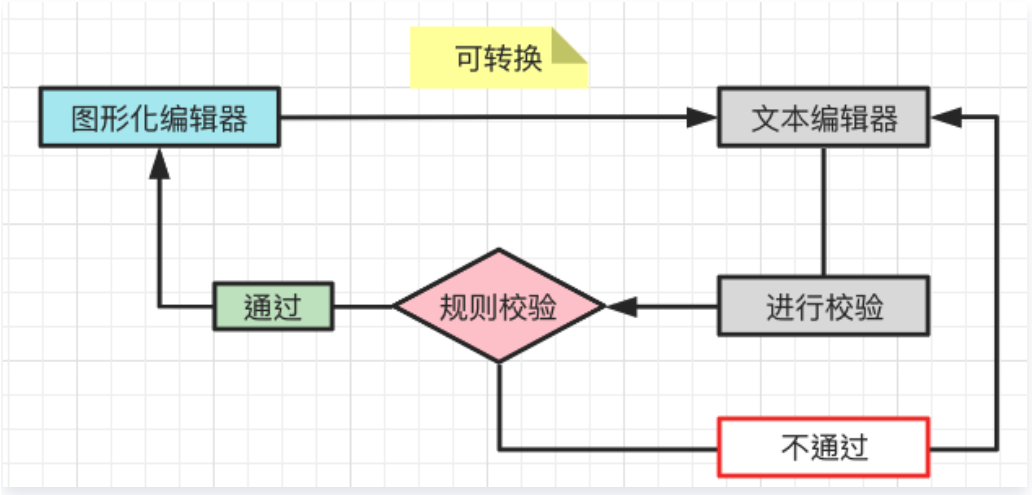
Sample File

```
pipeline {
  agent any
  stages {
    stage('Detection') {
      steps {
        sh 'ci-init'
        checkout([$class: 'GitSCM', branches: [[name:
env.GIT_BUILD_REF]],
```

```
                                    userRemoteConfigs: [[url:
env.GIT_REPO_URL]]])
      }
    }
    stage('Build') {
      steps {
        echo 'Building...'
        sh 'make'
        echo 'Build complete.'
      }
    }
    stage('Testing') {
      steps {
        echo 'Running unit tests...'
        sh 'make check'
        junit 'reports/**/*.xml'
        echo 'Unit tests complete.'
      }
    }
    stage('Deployment') {
      steps {
        echo 'Deploying...'
        sh 'make publish'
        echo 'Deployment complete.'
      }
    }
  }
}
```

Switching Between Editors

In essence, the graphical editor is preset code, allowing you to switch seamlessly to the text editor. However, you cannot switch from the text editor to the graphical editor. Code added or deleted in the text editor must pass a "rule check" before it can be converted into an editable view.



The text editor supports a wider range of custom operations than the graphical editor. As the graphical editor is preset with numerous commonly used steps, you can use it for pattern-based and standardized work. The text editor has no limitations and only requires conformance to Jenkins syntax, lending itself to specific and special tasks.

Build plan configuration
Trigger rules

Last updated: 2024-09-05 16:31:12

This article introduces how to set trigger rules in the build plan settings.

Prerequisites

Before configuring the CODING-CI build environment, you must activate the CODING DevOps service for your Tencent Cloud account. For details, please refer to Activate Services.

Open Project

1. log in to CODING console, click **Team Domain** to enter the CODING page.
2. After entering the target project, select **Continuous Integration** from the left navigation bar

Description of the Feature

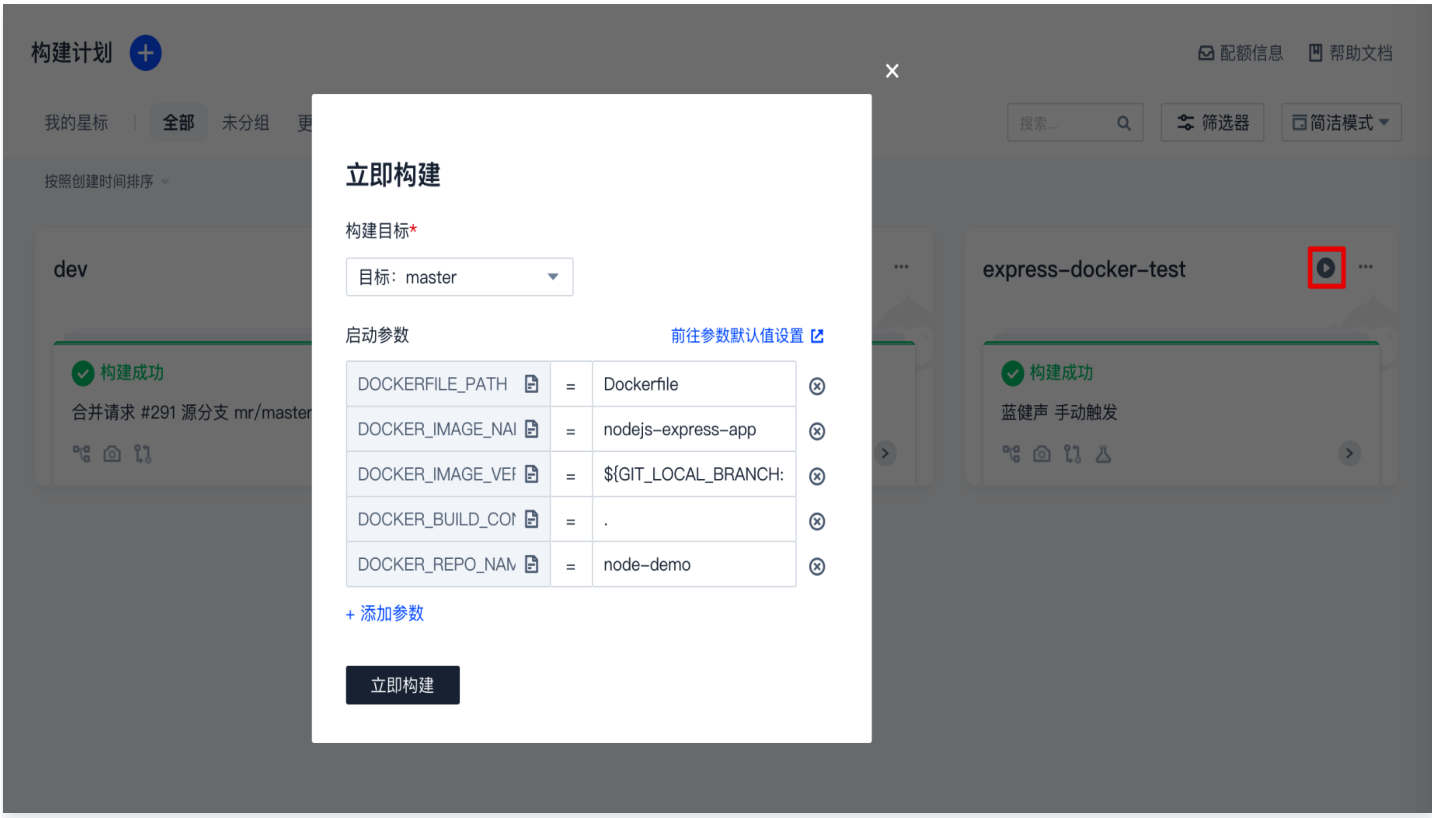
During the configuration process of a continuous integration plan, you can set the trigger rules for the build plan to run as needed. The trigger rules include the frequency and conditions for running the build plan. Each continuous integration build plan supports the following trigger methods:

- Manual trigger.
- Code source trigger.
- Scheduled trigger.
- API trigger.

Multiple triggering methods mentioned above can be used simultaneously.

Manual trigger

You can manually trigger a build plan by entering the build parameters, which will be added to the build environment in the form of environment variables.
On the build plan page, click **Build Now**, select the build target (Tag, Branch, Revision Version) as needed in the pop-up, and enter the required build parameters to trigger the build.



Code source trigger

A build plan configured with code source trigger will automatically monitor the code repository selected in this build plan and automatically trigger the build plan according to its changes.

Code updates

If **Automatically execute on code update** is selected without specifying any code source rules, any code submission will trigger the assembly line.

To set detailed build trigger conditions, you can enforce them through a combination of rules. There's an 'and' relationship between multiple rules; a build is triggered only when all conditions are met. Currently, continuous integration supports the following types of code commit trigger rules:

- **Include code branch or Tag**: The build will be triggered only when the specified code branch or Tag updates the code.
- **Exclude code branch or Tag**: The build will be triggered when the code is updated in cases other than the specified code branch or Tag.
- **Include file path**: The build will be triggered only when the specified file path updates the code.
- **Exclude file path**: The build will be triggered when the code is updated in paths other than the specified file path.
- **Include creator**: The build will be triggered only when the specified member updates the code.
- **Exclude creator**: The build will be triggered when the code is updated by members other than the specified member.

The specified code branch or Tag supports regular expression match for the full or short ref name:

1. For example, both refs/heads/master and master can match the master branch trigger.
2. If you want to trigger a build only when the master or dev branch is updated, use: `^refs/heads/(master|dev)` .

Merge requests

Triggering continuous integration through merge requests can simulate scenarios that might occur after the source branch is merged into the target branch, exposing issues early. You can choose under what conditions the merge request will execute a build:

- Triggered when a merge request is initiated.
- Triggered when a merge request is merged.

- Triggered when the source branch of a merge request changes.

  Indicates that the continuous integration tasks will listen for changes in the source branch. For example, if the source branch of a merge request is `issue/new`, the build plan will listen to the `issue/new` branch, and any new code submissions will automatically trigger the continuous integration tasks.
- Automatically cancel identical merge requests.
- Triggered when the target branch of a merge request changes.

  Indicates that the continuous integration tasks will listen for changes in the target branch. For example, if the target branch of a merge request is the `master` branch, the build plan will listen to the `master` branch, and any code changes will automatically trigger the continuous integration tasks.

If the above configurations are selected without specifying any code source rules, any merge requests will trigger the assembly line.

To set detailed build trigger conditions, you can enforce them through a combination of rules. There's an 'and' relationship between multiple rules; a build is triggered only when all conditions are met. Currently, continuous integration provides the following types of merge request trigger rules:

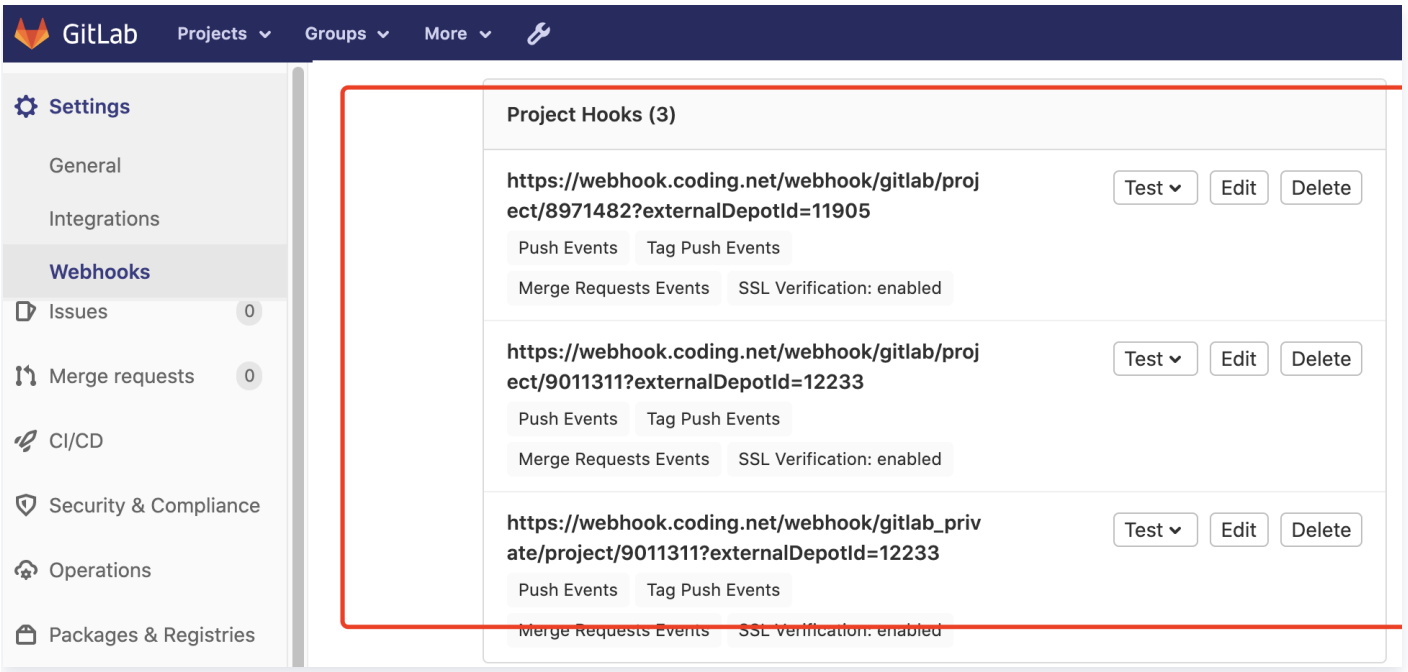
- **Include code branch**: A merge request that includes the specified code branch (supports regular expression matching) will trigger the build.
- **Exclude code branch**: A merge request excluding the specified code branch will trigger the build.
- **Include file path**: A merge request that includes the specified file path will trigger the build.
- **Exclude file path**: A merge request excluding the specified file path will trigger the build.
- **Include creator**: A merge request from the specified member will trigger the build.
- **Exclude creator**: A merge request from members other than the specified member will trigger the build.

Automatically cancel identical builds

You can check the settings to **Automatically cancel identical version numbers** and **Automatically cancel identical merge requests** triggered builds (only the latest one will be preserved).

Private GitLab

Bind Private GitLab : When binding a private GitLab, a GitLab Webhook will be automatically created. Subsequent events automatically notify CODING, matching the above-set trigger rules.



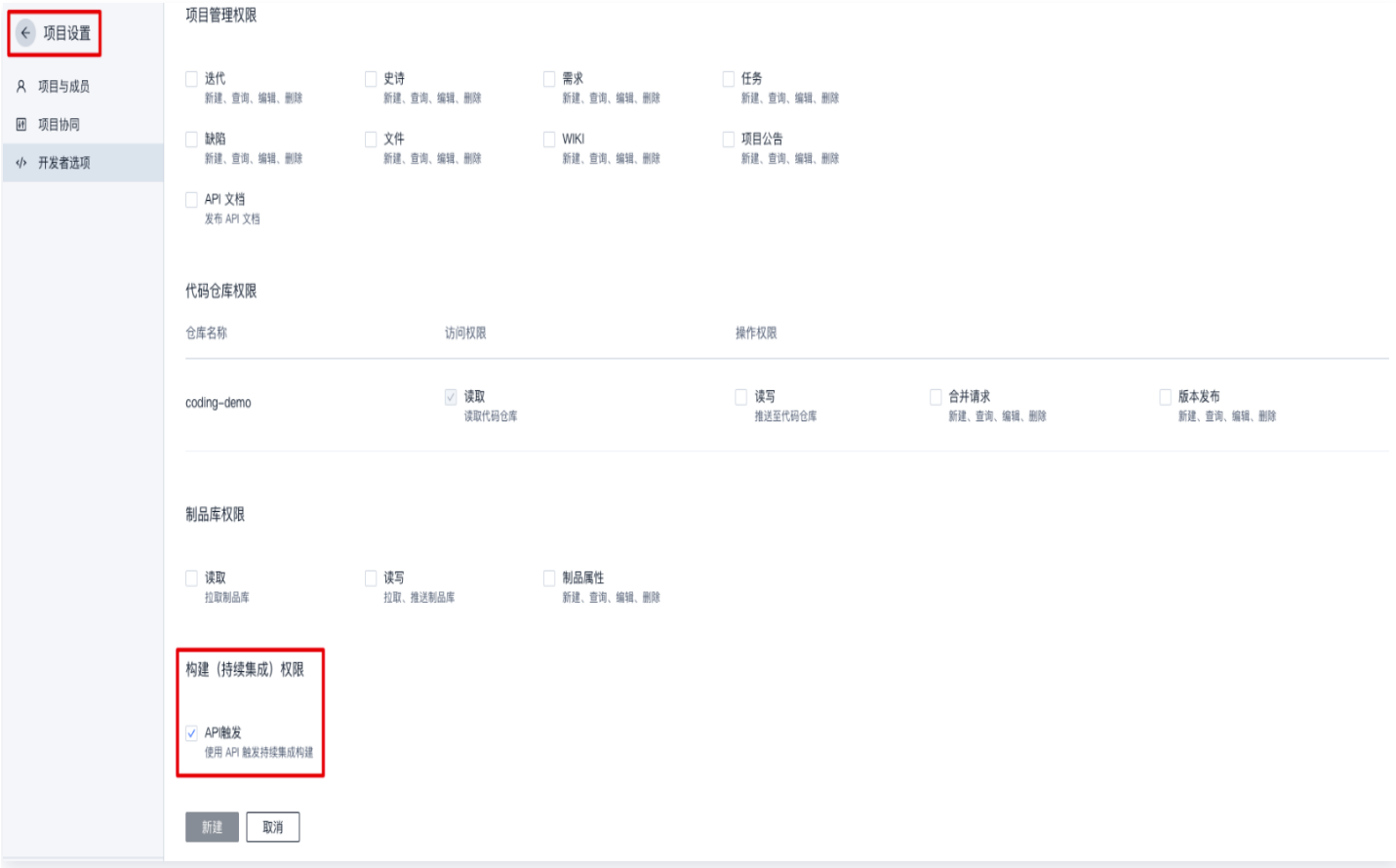
Scheduled trigger

By configuring a scheduled trigger for a build plan, you can set the build plan to be triggered periodically or at specific times to generate build tasks.
You can add multiple scheduled triggers to a build plan with no order of priority. If scheduled triggers overlap, multiple builds are triggered.

Configuration Item	Description
Select branch	If the selected branch code has not changed compared to the last trigger, even if the trigger time is reached, it will not trigger the scheduled task for the build.
Select Date	You can select multiple dates within a week.
Trigger method	• Periodic trigger: You can select any time between 00:00 – 24:00 as the period (accurate to the hour) to trigger tasks at the selected interval. • Single trigger: You can select any time between 00:00 and 24:00 as the trigger time point (accurate to the minute).

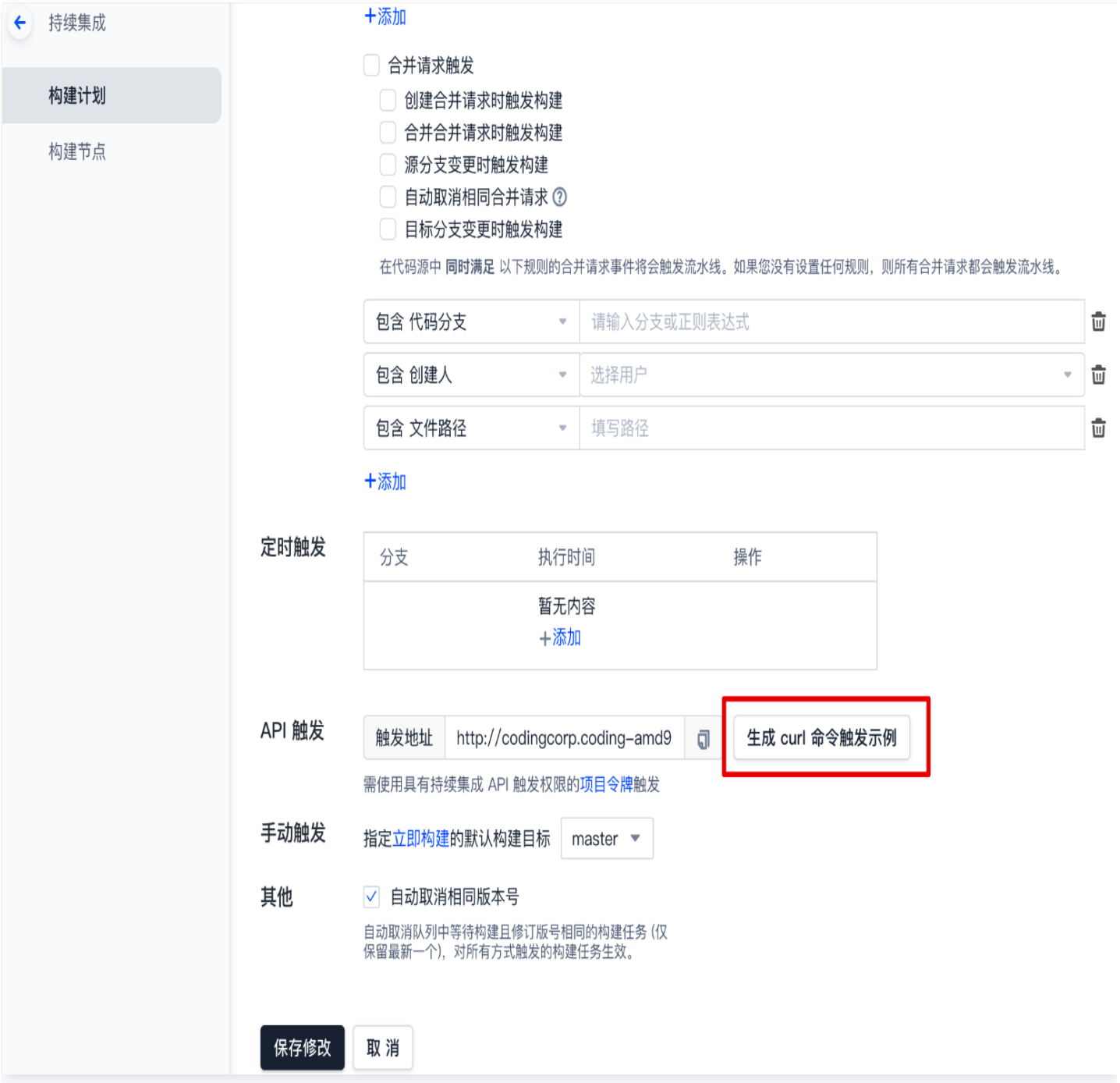
API trigger

Before using this feature, please ensure you have generated a token with Continuous Integration API trigger permissions in **Project Settings > Developer Options > Project Token > Create New Token.**



After generating the token with the corresponding permissions, you can call the API trigger interface in the build plan. Click to generate the CURL command trigger example to generate

the corresponding call command.



API

When the project token calls the CODING-CI API, the authentication method is `Basic Auth`. Below are the API details and relevant parameters.

Trigger build task

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

Request body

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

Return body

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

Parameter description

Parameter name	Parameter location	Required	Type	Default Value	Description
ref	body	No	string	master	Built target ref (`commit sha / tag / branch`), ignore if the code repository isn't used in the build plan
envs	body	No	env[]	-	Startup parameter of build plan

envItem

Parameter name	Parameter location	Required	Type	Default Value	Description
name	envItem.n	Yes	string	mast	Name of startup parameter of

	ame			er	build plan
value	envItem.value	No	string	–	Startup value of build plan
sensitive	envItem.sensitive	No	number	0	Whether to keep startup parameters confidential and do not show startup parameters in log. 1: confidential, 0: plaintext

Environment variable

Last updated: 2024-09-05 16:32:05

This article introduces how to set environment variables in the Build Plan settings.

Prerequisites

Before configuring the CODING-CI build environment, you must activate the CODING DevOps service for your Tencent Cloud account. For details, please refer to Activate Services.

Open Project

1. log in to CODING console, click **Team Domain** to enter the CODING page.
2. After entering the target project, select **Continuous Integration** from the left navigation bar

Description of the Feature

During continuous integration, we often inject some configuration information (such as account password / version number, etc.) into the build process as environment variables. CODING-CI supports multiple forms of environment variables, and you can inject environment variables into the build process using the following methods in order of priority: (the first has the highest priority):

- withEnv in Jenkinsfile.
- environment in Jenkinsfile.
- Startup parameters in the Build Plan (Job).
- Environment variables in the Build Plan (Job) settings.
- System built-in environment variables during the build process.

The following sections will provide detailed explanations of these methods.

withEnv and environment

1. You can use environment in Jenkinsfile to define environment variables (as shown below):

```
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}"
                // The output is as follows:
                // MY_PROJECT is project-1
                // MY_TEAM is team-1
            }
        }
    }
}
```

2. During the build process, you may need to use the same environment variables at different stages. You can use withEnv to set environment variables for specific operations to avoid global environment variable pollution. All steps executed within withEnv will prioritize using the environment variables set by withEnv. You can refer to the following example for specific effects:

```
  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}"
                // The output is as follows:
                // MY_PROJECT is project-1
                // MY_TEAM is team-1

                // The environment variables set in withEnv are only
effective for steps within the scope and have a higher priority than
environment
                withEnv(['MY_PROJECT=project-2']) {

                    echo "MY_PROJECT is ${MY_PROJECT}"
                    echo "MY_TEAM is ${MY_TEAM}"
```

```
                    // The output is as follows:
                    // MY_PROJECT is project-2
                    // MY_TEAM is team-1

                }
            }
        }
    }
}
```

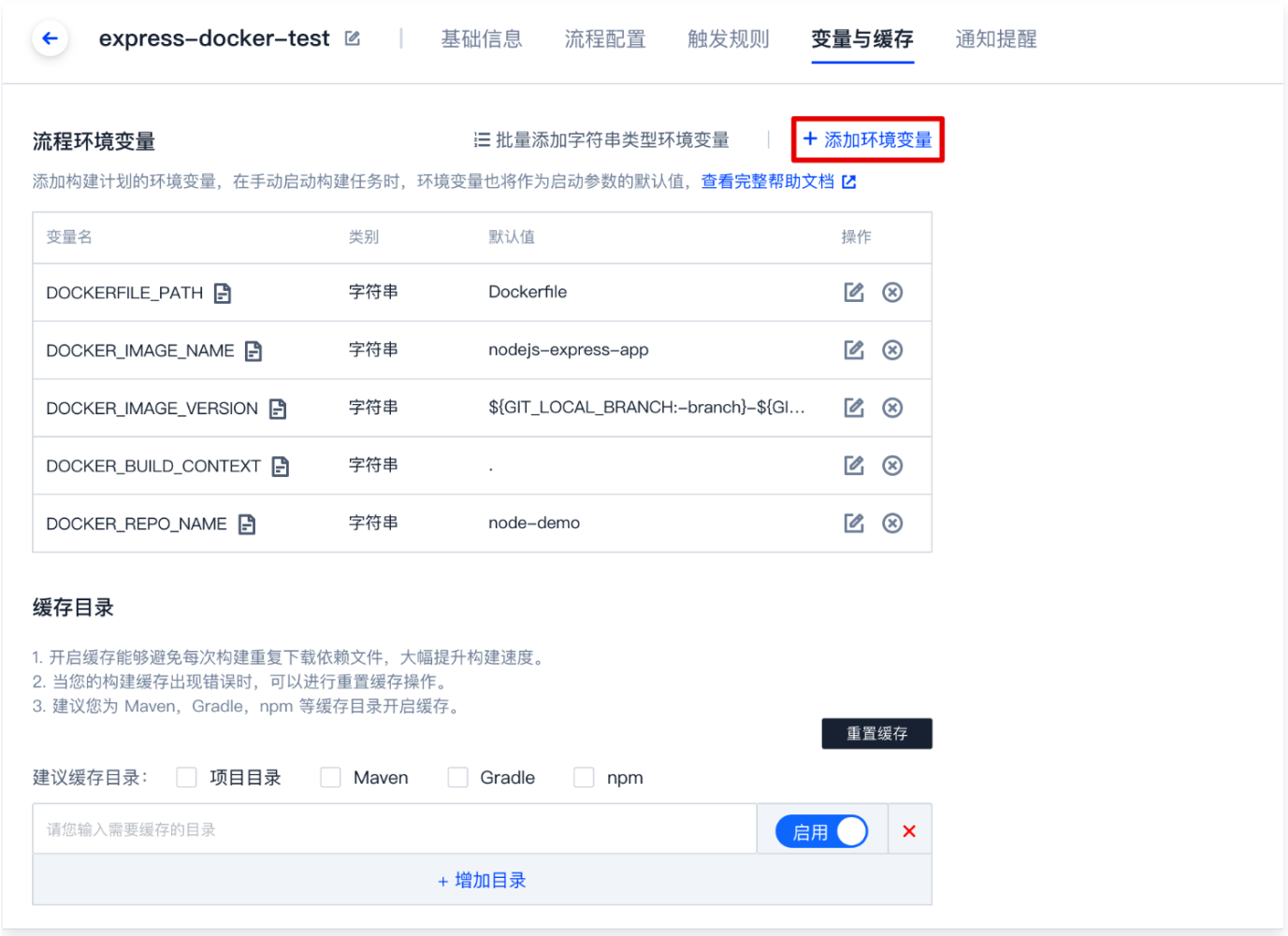
If you want to learn more about Jenkinsfile environment variables, please refer to Using an Environment Variable.

Startup parameters in the build plan

Second only to the environment variables configured in the Jenkinsfile, you can choose or fill in the corresponding environment variable value when starting the build plan.

Adding an environment variable

In addition to hard-coding environment variables in the Jenkinsfile, you can also set them in the build plan configuration. Currently, CODING supports adding the following four categories of environment variables: String, Single Selection, Multiple Selection, and CODING Credentials. You can also consider the environment variable configuration in the build plan as the default value for startup parameters.



System Built-in environment variables

During the build process, CODING-CI will inject corresponding environment variables for each construct task. You can view the list of default injected environment variables in the build snapshot:

All environment variables are summarized as follows and categorized by different trigger rules (Triggered by code changes, Scheduled trigger, When a merge request is initiated):

Serial number	Variable name	Variable meanings	Triggered by code changes	Scheduled trigger	When a merge request is initiated
1	CREDENTIALS_ID	Deployment Private Key CredentialsId for pulling the repository	✓	✓	✓
2	DOCKER_REGISTRY_CREDENTIALS_ID	Docker Private Key CredentialsId (equivalent to CODING_ARTIFACTS_CREDENTIALS_ID)	✓	✓	✓

3	CODING_ARTIFACTS_CREDENTIALS_ID	Artifact Repository Private Key CredentialsId for pulling artifacts within the project	✓	✓	✓
4	GIT_HTTP_URL	HTTPS Protocol Code Repository Address	✓	✓	✓
5	GIT_BUILD_REF	Build Corresponding Git Revision Version Number	✓	✓	✓
6	GIT_DEPLOY_KEY	Deployment Public Key of the Code Repository	✓	✓	✓
7	GIT_COMMIT	Current Version's Revision Version Number	✓	✓	✓
7	GIT_COMMIT_SHORT	First 7 Characters of the Revision Version Number	✓	✓	✓
8	GIT_PREVIOUS_COMMIT	Previous Build's Revision Version Number	✓	✓	✓
9	GIT_AUTHOR_EMAIL	Latest Committer Email of this Version	✓	✓	✓
10	GIT_SSH_URL	Protocol Code Repository Address	✓	✓	✓
11	GIT_COMMITTER_NAME	Latest Committer Name of this Version	✓	✓	✓
12	GIT_AUTHOR_NAME	Latest Author Name of this Version	✓	✓	✓
13	REF	Version to Build	✓	✓	✓
14	GIT_PREVIOUS_SUCCESSFUL_COMMIT	Previous Build's Successful Revision Version Number	✓	✓	✓
15	GIT_COMMITTER_EMAIL	Latest Committer Name of this Version	✓	✓	✓
16	GIT_BRANCH	Branch Triggering the Build	✓	✓	✓
17	GIT_URL	Repository SSH Protocol Address	✓	✓	✓

18	GIT_LOCAL_BRANCH/BRANCH_NAME	Local Branch Name	✓	✓	✓
19	FETCH_REF_SPECS	Git Refs to check out	✓	✓	✓
20	GIT_REPO_URL	Repository SSH Address	✓	✓	✓
21	JOB_ID	Build Plan ID	✓	✓	✓
22	JOB_NAME	Build Plan Name	✓	✓	✓
23	CI_BUILD_NUMBER	Build Number	✓	✓	✓
24	PROJECT_ID	Project ID	✓	✓	✓
25	PROJECT_NAME	Project name	✓	✓	✓
26	PROJECT_WEB_URL	Project Web Address	✓	✓	✓
27	PROJECT_API_URL	Project Backend API Address	✓	✓	✓
28	PROJECT_TOKEN	Project Token Password for reading the project	✓	✓	✓
29	PROJECT_TOKEN_GK	Project Token Username	✓	✓	✓
30	GIT_TAG	Triggered Build Git Tag (only available when using Tag for the build)	✓	–	–
31	DEPOT_NAME	Current Code Repository Name	✓	–	–
32	CCI_CURRENT_PROJECT_COMMON_CREDENTIALS_ID (coming soon)	Built-in Project Token CredentialsId	✓	–	–
33	CCI_CURRENT_TEAM (coming soon)	Enterprise Name of the current build environment, e.g., myteam in myteam.coding.net	✓	–	–

34	CCI_CURRENT_DOMAIN (coming soon)	Domain name of the current build environment, e.g., coding.net in myteam.coding.net	✓	–	–
35	MR_RESOURCE_ID	Merge Request ID	–	–	✓
36	MR_TARGET_BRANCH	Merge Request Target Branch Name	–	–	✓
37	MR_TARGET_SHA	Merge Request Target Branch Version Number	–	–	✓
38	MR_MERGED_SHA	Simulated Merged Version Number	–	–	✓
39	MR_SOURCE_BRANCH	Merge Request Source Branch Name	–	–	✓
40	MR_STATUS	Merge Request Status	–	–	✓
41	MR_SOURCE_SHA	Merge Request Source Branch Version Number	–	–	✓

Build snapshot

Last updated: 2024-09-05 16:32:20

This article introduces how to use the Build Snapshot feature.

Prerequisites

Before configuring the CODING-CI build environment, you must activate the CODING DevOps service for your Tencent Cloud account. For details, please refer to Activate Services.

Open Project

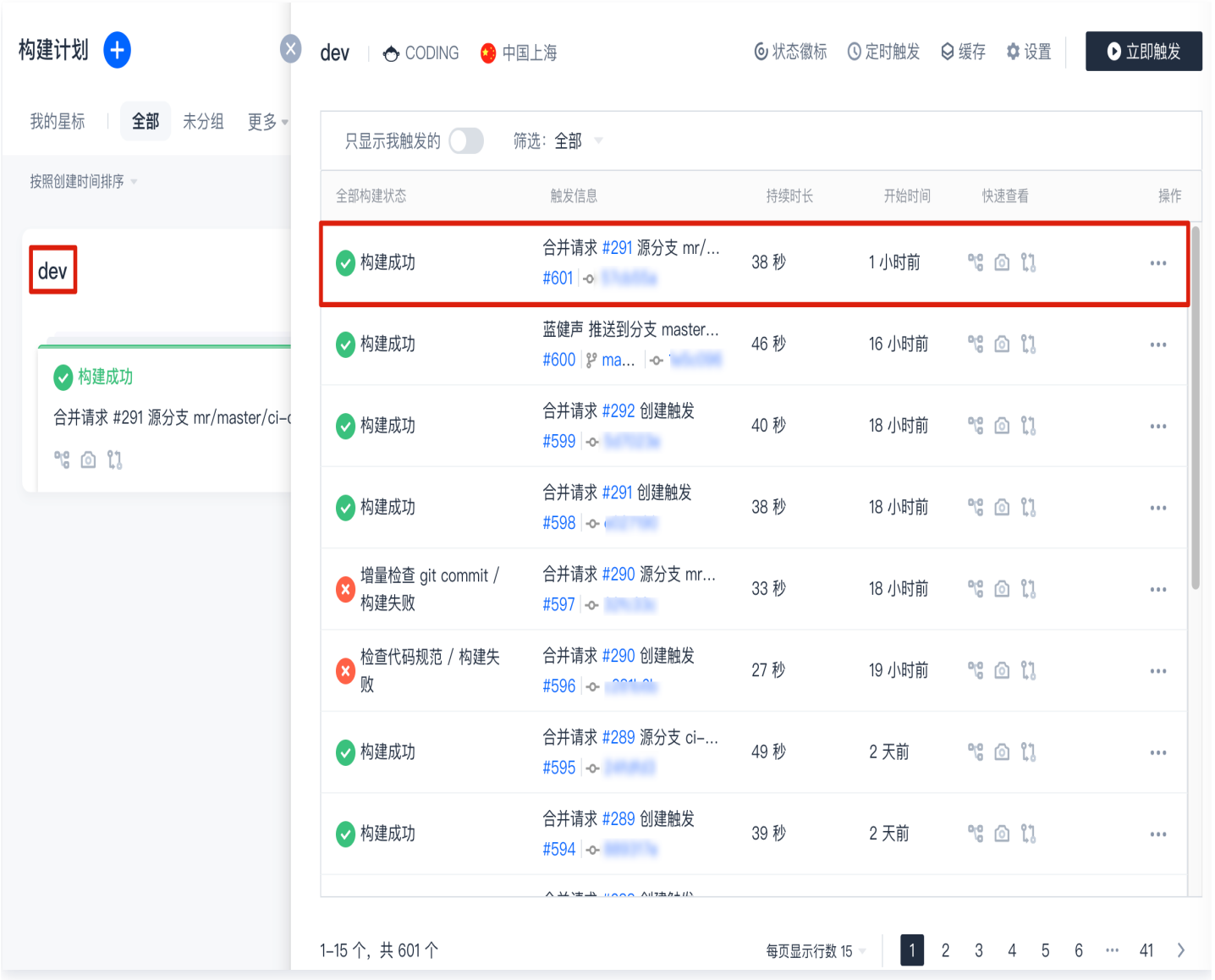
1. log in to CODING console, click **Team Domain** to enter the CODING page.
2. After entering the target project, select **Continuous Integration** from the left navigation bar

Description of the Feature

You may use different configuration files or build parameters for each build task in Continuous Integration. CODING Continuous Integration (CODING-CI) features build snapshots to allow you to review the execution process of a build task. Build snapshots clearly show the configuration parameters of every build record.

View build configuration

1. In the project, click **Continuous Integration > Build Plan**. Click the title of a single build plan to view all build records. Click to enter any build record:



2. In the build record, click **Build Snapshot** to see the configuration snapshot of each build record: Startup Parameters, Environment Variables, Process Configuration File.

Startup parameter

1. The startup parameters are the parameters that you entered when starting a build task. They are incorporated into the run environment of the build task as environment variables.

2. After the build is complete, you can view the configured Startup Parameters in the Build Snapshot.

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	.

Environment variable

1. The environment variables only include those you configured when starting the task, and exclude all environment variables generated or dynamically set in the run process.

2. In the Environment Variables tab, users can see the environment variables set by the system and users when the task is started.

Process configuration file

Select the tab for the process configuration to view the configuration file (Jenkinsfile) used for the build record.


```
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()
          }
        }
```

Cache directory

Last updated: 2024-09-05 16:32:35

This article introduces how to use the cache directory feature.

Prerequisites

Before configuring the CODING-CI build environment, you must activate the CODING DevOps service for your Tencent Cloud account. For details, please refer to Activate Services.

Open Project

1. log in to CODING console, click **Team Domain** to enter the CODING page.
2. After entering the target project, select **Continuous Integration** from the left navigation bar

Description of the Feature

When installing dependencies for a local project, the downloaded files are cached for the next installation. For example, after running the `npm install` command, ./`node_modules` is generated in the project and cached in the `~/.npm` directory, which is more compact and universal.

- Default build nodes

  CODING will automatically allocate computing resources for each build plan and destroy them upon completion. Each build will automatically reassign a build node, so a cache directory needs to be specified to speed up the next build.
- Custom build nodes

  If you choose to use your own computing resources and select a custom build node to execute tasks in the build plan, the server will not be destroyed upon completion, so specifying a cache directory is unnecessary.

When using Docker in continuous integration, mount the cache directory to the Docker container.

Default build nodes

1. CODING provides the basic computing resources for build plans. A CVM is assigned for each task in a Linux build environment with root user permissions, and the cache directory is as follows:

Package management tool	Cache directory
Maven	/root/.m2/

Gradle	/root/.gradle/
npm	/root/.npm/
composer	/root/.cache/composer/
yarn	/usr/local/share/.cache/yarn/

2. You can select the cache directory in the **Variables and Caches** section of the build plan settings. If the target directory is not found, you can manually enter it.

Docker build environment

If you are using a Docker environment in the build plan, first select the cache directory in the **Variables and Caches** section, then mount it to Docker.

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('Java cache') {
      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 cache') {
      steps {
        script {
          docker.image('node:14').inside('-v /root/.npm/:/root/.npm/') {
            sh 'npm install'
          }
        }
      }
    }
  }
}
```

Custom build nodes

When using a Docker environment in a custom build node, find the cache directory corresponding to the server username. For example, if the default username for Ubuntu servers is ubuntu, the cache directory is `/home/ubuntu/.npm/`. The corresponding code would be:

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

Cache basic docker images

1. If the basic Docker images, such as the `Dockerfile` base image and CI agent image, need to be pulled for every build, it can take a lot of time, so caching can speed up the process. Refer to the following `Jenkinsfile`. Modify the image name to reuse it:

```
pipeline {
  agent any
  environment{
    DOCKER_CACHE_EXISTS = fileExists '/root/.cache/docker/php-8.0-
cli.tar'
  }
  stages {
    stage('Load cache') {
      when { expression { fileExists(DOCKER_CACHE_PATH).equals(true) }
}
      steps {
        sh 'docker load -i /root/.cache/docker/php-8.0-cli.tar ||
true'
      }
    }
    stage('Use Image (Please modify this section)') {
      agent {
        docker {
          image 'php:8.0-cli'
          args '-v /root/.cache/:/root/.cache/'
          reuseNode 'true'
        }
      }
      steps {
        sh "php -v"
      }
    }
    stage('Generate cache (run once only)') {
      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. Add the `/root/.cache/` path to the cache directory. The build time is significantly reduced on the second build, indicating the cache is working:

> **Note:**
> Cache images can become outdated over time, so it's recommended to clean them regularly to stay in sync with official updates.

Save Dockerfiles

If you are using a Dockerfile as the build environment in Continuous Integration, instead of running the `docker build` command at initialization, save the built Docker images to a repository to pull and reuse them again.

Jenkinsfile

```
// Creates a CODING Docker repository and obtains the username,
password, and repository URL
sh "docker login -u $DOCKER_USER -p $DOCKER_PASSWORD my-team-
docker.pkg.coding.net"

// Use MD5 of Dockerfile as 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}"

// Check if images exist in remote repository
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)
}

// Use images for automated testing
testImage.inside("-e 'APP_ENV=testing'") {
    stage('test') {
        echo 'testing...'
        sh 'ls'
        echo 'test done.'
    }
}
```

Code explanation: Execute the following commands in the shell to determine if the image already exists based on the return value.

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

Cache is also one of the best ways to speed up the build process, but different tools and languages have different solutions. The following will enumerate several common scenarios and acceleration schemes.

Default node using cache

For the default node in Continuous Integration, if you build directly on the host machine, Maven and Gradle will download dependencies to /root/.m2 and /root/.gradle respectively, similarly, Npm will download dependencies to /root/.npm.

Docker Self-Defined build environment using cache

You can refer to Dockerfile Best Practices. This document is an official Docker documentation that proposes some measures to help us write excellent Dockerfiles.

Abstracting the FROM image

If you have multiple services and each Dockerfile needs to install some tools via apt-get, you should abstract your FROM image by installing the necessary tools and use it as the FROM image in the Dockerfile. For the team, managing the base image is very necessary. It not only reduces build time but also uniformly fixes security vulnerabilities, increases built-in tools, and greatly reduces maintenance costs for developers and operations staff.

Self-Defined build environment using cache

Continuous Integration can use not only the official build environment but also Docker self-defined build environments. A self-defined build environment is essentially running a docker container with build instructions executed within this environment.
Example: If you need to use Java18 and Maven to build a program:

```
pipeline {
  agent {
    docker {
      reuseNode 'true'
      registryUrl 'https://coding-public-docker.pkg.coding.net'
      image 'public/docker/openjdk:18-2022'
      args '-v /root/.m2:/root/.m2 -v /usr/bin/docker:/usr/bin/docker -v
/var/run/docker.sock:/var/run/docker.sock'
    }
  }

  stages {
    stage('Checkout') {
      steps {
        checkout([
          $class: 'GitSCM',
          branches: [[name: env.GIT_BUILD_REF]],
          extensions: [[$class: 'CloneOption', depth: 1, noTags: false,
shallow: true]],
          userRemoteConfigs: [[url: env.GIT_REPO_URL, credentialsId:
env.CREDENTIALS_ID]]
        ])
      }
    }
```

```
    stage('Compile') {
      steps {

        sh "mvn clean install"
      }
    }

    stage('push ') {
      steps {
        sh "echo password | docker login xxx-docker.pkg.coding.net -u
xxx  --password-stdin  "
        sh "docker build -t myjava:latest ."
        sh "docker push myjava:latest "
      }
    }
  }
}
```

Please note

```
args '-v /usr/bin/docker:/usr/bin/docker -v
/var/run/docker.sock:/var/run/docker.sock -v /root/.m2:/root/.m2 '
```

- `-v /usr/bin/docker:/usr/bin/docker -v /var/run/docker.sock:/var/run/docker.sock` mounts docker and docker.sock to the self-defined environment container, enabling the use of docker in subsequent stages and steps.
- `-v /root/.m2:/root/.m2` maps the host machine's /root/.m2 to the container's /root/.m2 in the self-defined build environment, ensuring that the dependencies downloaded by mvn clean install in the container will be stored in the container's /root/.m2.
- Check the variable and cache Maven, so the next build will utilize this cache.

Build Node
Build Node Types

Last updated: 2024-09-05 16:32:51

This article introduces you to the types and differences of build nodes.

Prerequisites

Before configuring the CODING-CI build environment, you must activate the CODING DevOps service for your Tencent Cloud account. For details, please refer to Activate Services.

Open Project

1. log in to CODING console, click **Team Domain** to enter the CODING page.
2. After entering the target project, select **Continuous Integration** from the left navigation bar

Description of the Feature

When you use CODING-CI for builds, you are essentially calling computing resources as **Build Nodes** to complete build tasks. You can choose to use the cloud computing resources officially provided by default or connect your own Custom Build Nodes to run build tasks.

Default build nodes

The default nodes have a built-in build environment, pre-installed with development language SDKs, TCCLI, and other services. See Default Node Environment.

Egress IP Address:

Egress IP addresses used by default build nodes:

```
# Shanghai, China

43.143.12.0/24
43.142.234.0/24
43.142.23.0/24
124.220.180.0/24
81.68.101.0/24
111.231.92.0/24
101.33.207.0/24
43.136.72.0/24
```

```
# Hong Kong (China)

124.156.164.25/32
119.28.15.65/32

# Silicon Valley, USA

170.106.136.17/32
170.106.83.77/32
```

Custom build nodes

In actual development projects, the development environments involved may vary. When the build environment of official nodes cannot meet the continuous integration requirements of a project, such as needing macOS Xcode to build an iOS app, you can run specific tasks by connecting to Custom Build Nodes (physical machines/virtual machines/containers, etc.). For more information on Custom Build Nodes, see Custom Nodes.

Default Build Environment

Last updated: 2024-09-05 16:33:04

This article introduces the build environment for default nodes.

Prerequisites

Before configuring the CODING-CI build environment, you must activate the CODING DevOps service for your Tencent Cloud account. For details, please refer to Activate Services.

Open Project

1. log in to CODING console, click **Team Domain** to enter the CODING page.
2. After entering the target project, select **Continuous Integration** from the left navigation bar

Description of the Feature

Build tasks are executed by build nodes, and the build environment refers to the system underlying environment built into the build nodes, pre-installed with development language SDKs, TCCLI, and other services.
The build environment includes the following categories:

- Default Environment
- Custom Build Environment

If the development project has specific requirements for the runtime environment, such as a swift project needing to run in a macOS environment, you can refer to Custom Build Nodes to manually integrate the build nodes.

Default Environment

Select the build environment in the initial steps of the build plan.

The corresponding `Jenkinsfile` is `agent any`:

```
pipeline {
  agent any
  stages  {
    stage("checkout") {...}
    stage("code inspection") {...}
  }
}
```

CODING Cloud Hosting is an Ubuntu system, pre-installed with the following SDKs and TCCLI:

SDK	Command Line Interface
• android-sdk: 26.1.1 • build-essential • dotnet-core: 2.2 • elixir: 1.8.1 • erlang: Erlang/OTP 21 • go: 1.14.4 • java: 1.8.0_191 • nodejs: 10 • php: 8.0,7.4,7.3 • python3/pip3: 3.9,3.8,3.7 • python: 2.7.12	• bundler: 1.17.2 • cmake: 3.5.1 • composer:1.10.8 • coscmd:1.8.5.36 • docker-compose: 1.26.0 • docker:20.10.6 • git-lfs: 2.7.2 • git:2.28.0 • gradle: 7.0.2 • helm: 2.13.1 • jq: 1.5-1-a5b5cbe

- ruby: 2.6.0

- kubectl: 1.18.4
- maven: 3.6.3
- mercurial: 3.7.3
- pigz: 2.3.1
- rancher: 2.2.0
- rvm: 1.29.7
- sshpass: 1.05
- svn: 1.9.3
- tccli: 3.0.67.1
- vsftpd: 3.0.3
- yarn: 1.15.2
- axel: 2.5

Note:
The pre-installed software versions are limited and periodically updated, and the versions needed for various projects may differ.

Custom Build Environment

Last updated: 2024-09-05 16:33:21

This article introduces how to set up a custom build environment.

Prerequisites

Before configuring the CODING-CI build environment, you must activate the CODING DevOps service for your Tencent Cloud account. For details, please refer to Activate Services.

Open Project

1. log in to CODING console, click **Team Domain** to enter the CODING page.
2. After entering the target project, select **Continuous Integration** from the left navigation bar

Public Node Custom Version

In continuous integration, you can download and install various versions of software. If the download from the official website is slow, we offer an image service for download. To add artifacts, please submit them to the open source project.

Go

```
stage('Go') {
  steps {
    // It's recommended to set the "cache directory" to
/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'
```

```
      // More versions: 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' and 7.3, 7.2, 7.1, and 5.6
      args '-v /var/run/docker.sock:/var/run/docker.sock -v
/usr/bin/docker:/usr/bin/docker'
    }
  }
  stages {
    stage('Install dependency') {
      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 Environment

CODING-CI provides a default build environment. If the pre-installed SDK version and TCCLI in the default environment do not meet your requirements, you can also use the Docker Build Environment within continuous integration. The following methods can be used to utilize the Docker Build Environment:

- Images provided by CODING officially.
- Use Docker images already hosted in the project-level artifact repository.
- Standard build environment suitable for the project hierarchy to ensure the security of project images. You can also pull images from other projects using the project token.
- Use Docker images from a specified registry address (default is Docker Hub).
- Use Dockerfile scripts to build the environment.

CODING Docker Image

In **Continuous Integration Plan Settings > Process Configuration > Basic Configuration > Graphical Editor**, select **use** the **CODING official Docker image**, for example, Node.js 14:

Refer to the `Jenkinsfile`:

```
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'
      }
    }
  }
}
```

Project Artifact Repository Docker Image

The following content uses a Node.js 12 environment configured with the process management tool pm2 as an example to demonstrate step-by-step how to **push a custom image to the artifact repository** and **use the custom image** as a build environment.

Step 1: Build the Docker Image

1. Create a new directory and create the Dockerfile as follows:

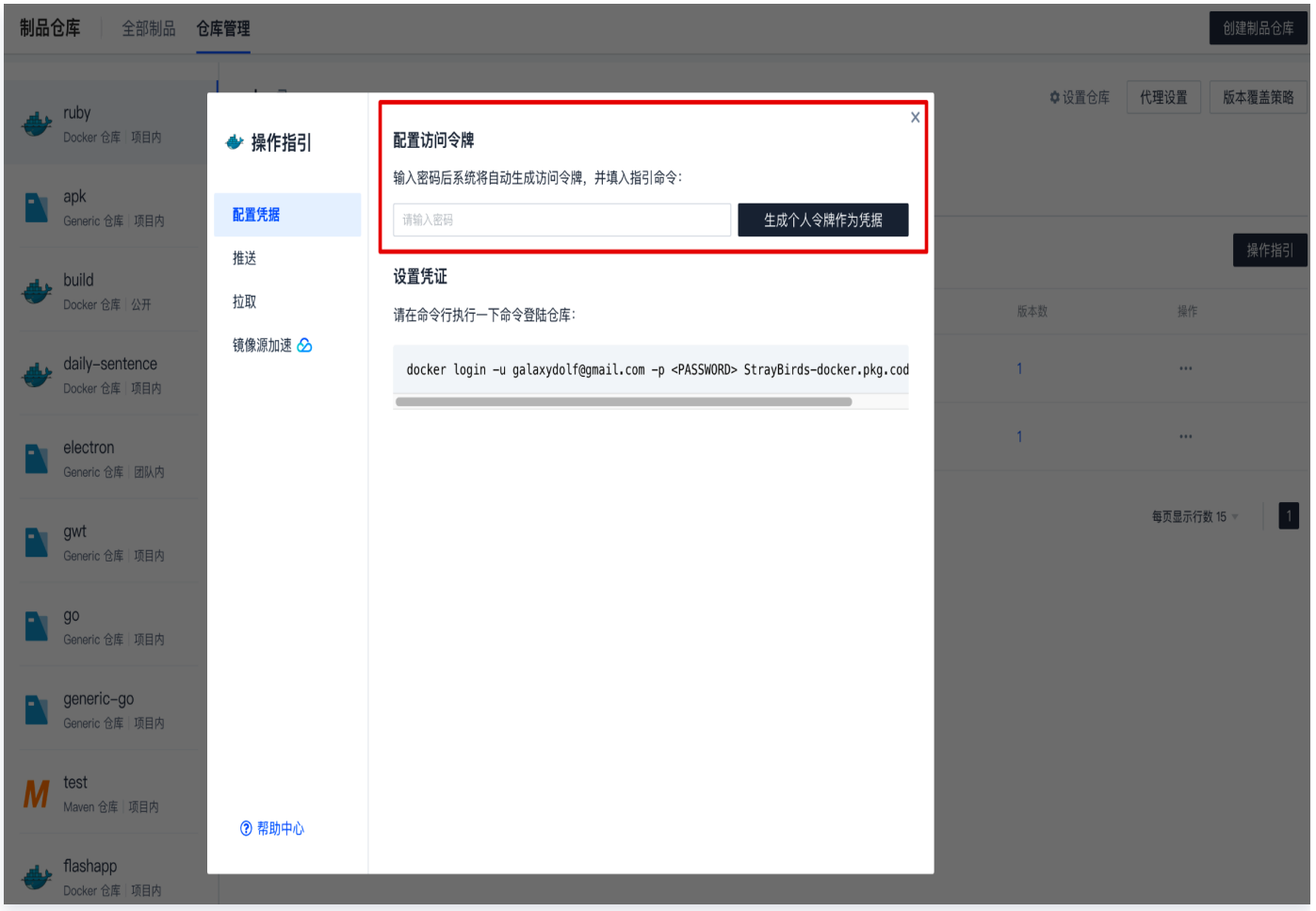
```
# Specify node.js version as node 12, by default pulled from Docker
Hub
FROM node:12
# Install pm2
RUN npm install pm2 -g

COPY . .
# Set the command when the container starts
CMD [ "pm2-runtime", "start" ]
```

2. Run the command `docker build -t pm2-test .`; `-t` specifies the image name.

```
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
```

Step 2: Push the Image to CODING-AR

1. Enter CODING-AR, select an existing artifact repository or create a new artifact repository, and input the password and then click **generate a personal token as credentials.**



2. Copy it and then input the command in the terminal to log in to .
3. Follow the operational guide to tag the local image.

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

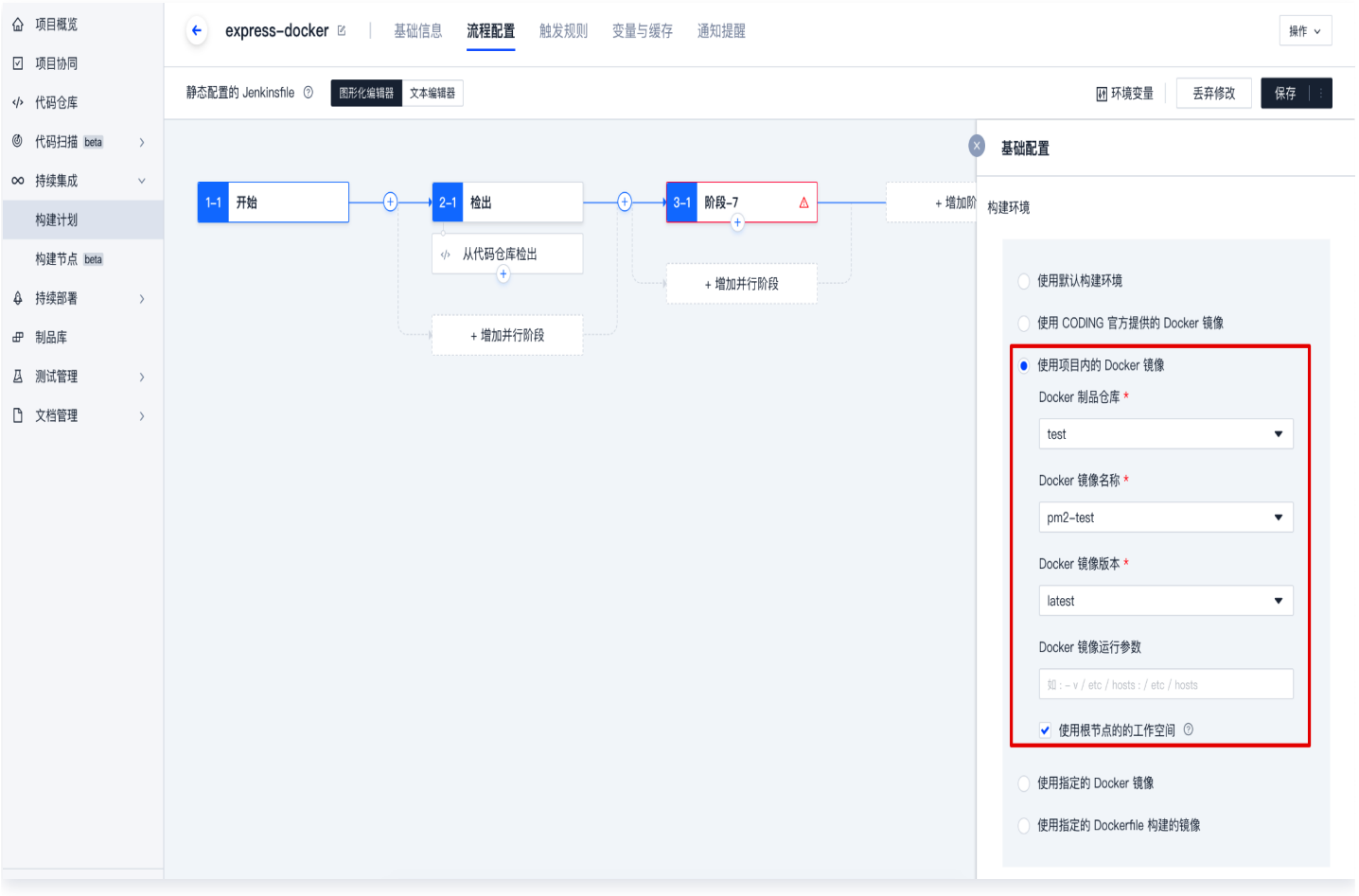
4. Push your docker image to CODING-AR.

```
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. Once the push is successful, you can find your image in the image list.

Step 3: Use the image as the build environment in Continuous Integration

Go to **Continuous Integration Settings > Pipeline Configuration**, select the project's Docker image, and choose the corresponding artifact repository and image.



Specify the address of the Docker image

The Docker image is a required field and must contain your image name. The **Registry Address** should be a URL without a path, such as:
Correct: https://codes-farm-docker.pkg.coding.net
Incorrect: https://codes-farm-docker.pkg.coding.net/laravel-demo/laravel-docker/

If pulling a private image, enter the credentials first and then fill the **Registry Authentication Credentials ID**.

Corresponding Jenkinsfile:

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

```
        sh 'node --version'
      }
    }
  }
}
```

Dockerfile Build Environment

If the project already uses Docker, it is recommended to submit the `Dockerfile` to the code repository and use it as the build environment for Continuous Integration. `Dockerfile` example code:

```
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' // Optional, the filename of the
definition Dockerfile
          dir 'build' // Optional, directory where the Dockerfile is
located
          additionalBuildArgs '--build-arg version=1.0.2' // Optional,
additional docker build arguments
        }
```

```
        }
        stages {
          stage('Test') {
            steps {
              sh 'php -v'
              sh 'unzip -v'
            }
          }
        }
      }
    }
```

If the builds are frequent and you do not want to waste time in the `docker build` process, you can save the image using Jenkins Dockerfile for the next build, thereby saving a lot of time. Please refer to Save Dockerfile Image.

Use Docker in stages

Refer to the Jenkinsfile:

```
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'
      }
    }
  }
}
```

Multiple Docker Daemons

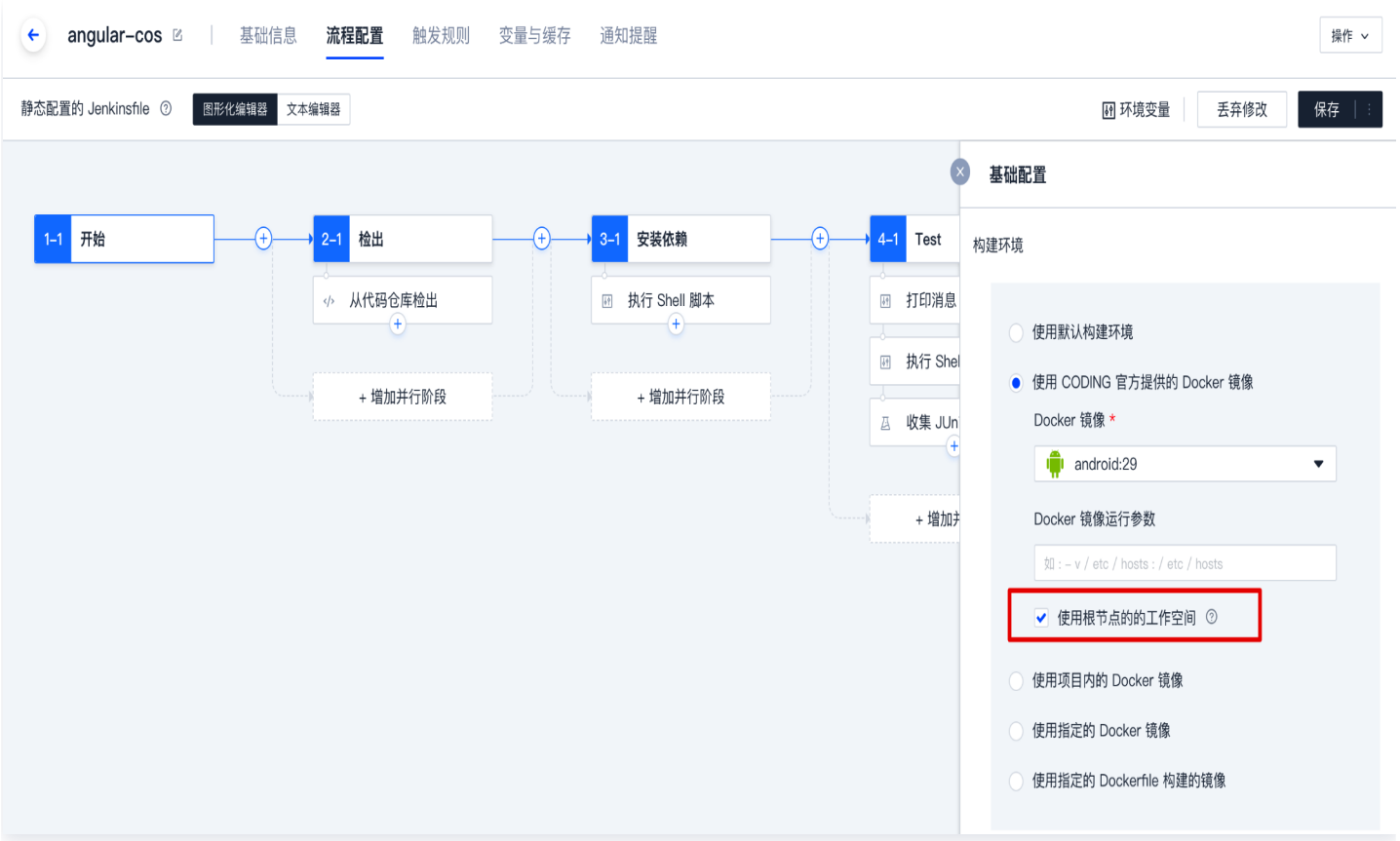
Automated testing often requires temporary infrastructure (e.g., MySQL, Redis, Elasticsearch). Therefore, create a bridged network, start multiple Docker daemons within it, and delete them automatically after testing is complete.

```
node {
  stage("checkout") {
    checkout([
      $class: 'GitSCM',
      branches: [[name: GIT_BUILD_REF]],
      userRemoteConfigs: [[
        url: GIT_REPO_URL,
        credentialsId: CREDENTIALS_ID
    ]]])
  }
  stage('prepare database') {
    sh 'docker network create bridge1'
    sh(script:'docker run --net bridge1 --name mysql -d -e
"MYSQL_ROOT_PASSWORD=my-secret-pw" -e "MYSQL_DATABASE=test_db"
mysql:5.7', returnStdout: true)
    sh(script:'docker run --net bridge1 --name redis -d redis:5',
returnStdout: true)
  }
  docker.image('ecoding/php:8.0').inside("--net bridge1 -v
\"${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:tbgOBtYci7i7cdx5RiFE3KZzUkRtJfbU3lbj5uPdL8U='") {
    sh 'composer install'

    stage('unit test') {
      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: 'Test Coverage Report', path:
'storage/reports/tests/')
      }
    }
}
```

Root Node Workspace

When using a custom Docker as a build environment, you can choose whether to use the root node workspace. If this option is selected, the Docker container for the current stage will run on the same build node as the pipeline, allowing access to all files saved in the root directory under the pipeline workspace.



The corresponding `Jenkinsfile` parameter is `reuseNode`, type: Boolean, default is false:

```
pipeline {
  agent {
    docker {
      registryUrl 'https://coding-public-docker.pkg.coding.net'
      image 'public/docker/android:29'
    }
  }
```

```
  stages {
    // Code is checked out to the root directory of the pipeline agent's
workspace
    stage('checkout code') {
      steps {
        checkout([
          $class: 'GitSCM',
          branches: [[name: env.GIT_BUILD_REF]],
          userRemoteConfigs: [[url: env.GIT_REPO_URL, credentialsId:
env.CREDENTIALS_ID]]
        ])
      }
    }
    stage('unit test') {
      agent {
        dockerfile {
          // By default, searches for a file named "Dockerfile" in the
root directory of the current node's workspace to build the environment
          filename 'Dockerfile'
          // If reuseNode is false, the previously checked out
Dockerfile in the root directory of the pipeline agent's workspace
cannot be found
          reuseNode true
        }
      }
      steps {
        sh 'npm run test:ci'
        junit '*.xml'

      }
    }
  }
}
```

Execute Docker commands

When running `docker` commands in a Jenkins Docker environment, you need to mount the external virtual machine's Docker socket, otherwise the error will occur: `docker: command not found`.

```
pipeline {
  agent {
    docker {
```

```
      image 'ecoding/php:8.0'
      reuseNode 'true'
      // Mount the external virtual machine's Docker socket
      args '-v /var/run/docker.sock:/var/run/docker.sock -v
/usr/bin/docker:/usr/bin/docker'
    }
  }
  stages {
    stage('Definition Phase') {
      steps {
        sh 'php -v'
      }
    }
    stage('Build Docker image') {
      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()
          }
        }
      }
    }
  }
}
```

If you wish to run Docker commands in the Definition build node, simply install the Docker service on the node to get started.

From Definition Node

Last updated: 2024-09-05 16:33:40

This article introduces how to use the Definition Node.

Prerequisites

Before configuring the CODING-CI build environment, you must activate the CODING DevOps service for your Tencent Cloud account. For details, please refer to Activate Services.

Open Project

1. log in to CODING console, click **Team Domain** to enter the CODING page.
2. After entering the target project, select **Continuous Integration** from the left navigation bar

Description of the Feature

In actual development projects, the involved development environments can be diverse. When the build environment of the default node cannot meet the project's runtime requirements, such as needing to use macOS Xcode to build an iOS application, you can use the Definition Node (Physical machine/Virtual machine/Container, etc.) to run specific tasks.
When accessing the build node, you need to specify the build node pool to be accessed. Build nodes cannot exist independently without being part of a node pool. Build Node Pool can be of two types: team and project.

> ⚠ **Note:**
> Using the Definition Node is equivalent to bringing your build machine under the control of CODING. To ensure the security of your machine and intranet, please keep your username and password safe, and it is recommended to enable secondary authentication for log in. If necessary, you also need to formulate security policies for the intranet of the build machine to ensure that your machine and intranet are not subject to unauthorized access and attacks.

Access Node

The process of accessing the Definition Node essentially involves running the `Worker` service in the node, click to learn Worker Common Commands.
Currently, macOS, Windows, and Linux environments are supported for access to the build plan node pool.

Recommended configuration

- CPU: 8 cores or above.
- Memory: 16 GB or above.

Environmental Dependencies

- **Python 3.6, Python 3.7, Python 3.8, Python 3.9**

  Click to visit Project Address for downloading.
- **Git ⩾ 2.8**

  Click to visit Project Address for downloading.
- **Java 8, Java 11**

  Click to visit Project Address for downloading.
- **Jenkins**

Windows: Go to the `C:/` directory, create the `codingci/tools` directory. Download the jenkins.war and jenkins_home.zip files there, and unzip the jenkins_home.zip file in the `tools/` directory.
Linux: Go to the `/root/` directory, create the `codingci/tools` directory. Download the jenkins.war and jenkins_home.zip files there, and unzip the jenkins_home.zip file in the `tools/` directory.

macOS: Go to the `~/` directory, create the `codingci/tools` directory. Download the jenkins.war and jenkins_home.zip files there, and unzip the jenkins_home.zip file in the `tools/` directory.

> **Note:**
> Pre-installing the above environments is a prerequisite for integrating a self-definition node. To ensure the build plan runs smoothly after integration, different environments may still need to be pre-installed depending on the task or plugin requirements. You can refer to Default Node Environment to install common SDKs and TCCLI.

macOS

Command Integration

1. Go to the build node, select **Integrate New Node > macOS**, choose Bash as the integration method, select the corresponding node pool in the integration configuration, click **Generate Access Configuration and Copy**.

2. After entering the command in the terminal, wait for the service to be downloaded. After the installation is complete, you can use the following command for verification:

```
qci_worker version
```

The default installation directory for command integration is `/root/codingci`. If you want to specify the installation directory, please refer to Specify Installation Directory.

Manual Integration

Before using the manual integration method, please ensure that the node meets the Environment Dependencies requirements mentioned above.

1. Select Manual Integration as the integration method, follow the prompts to enter the command in the terminal to install the client, that is, to install the `Worker` service.

2. Select the node pool you want to integrate, click **One-Click Generation and Copy** to generate the initialization command.
3. Execute the automatically generated client startup command in the terminal to keep the build node online.

Windows

Command Integration

1. Select **Continuous Integration > Build Nodes**, click the **Integrate New Node** button in the upper right corner, choose Windows, and select the Powershell integration method.

2. After installation, you can use the following command for verification:

```
qci_worker version
```

The default installation directory for command integration is `/root/codingci`. If you want to specify the installation directory, please refer to Specify Installation Directory.

Manual Integration

Before using the manual integration method, please ensure that the node meets the Environment Dependencies requirements mentioned above.

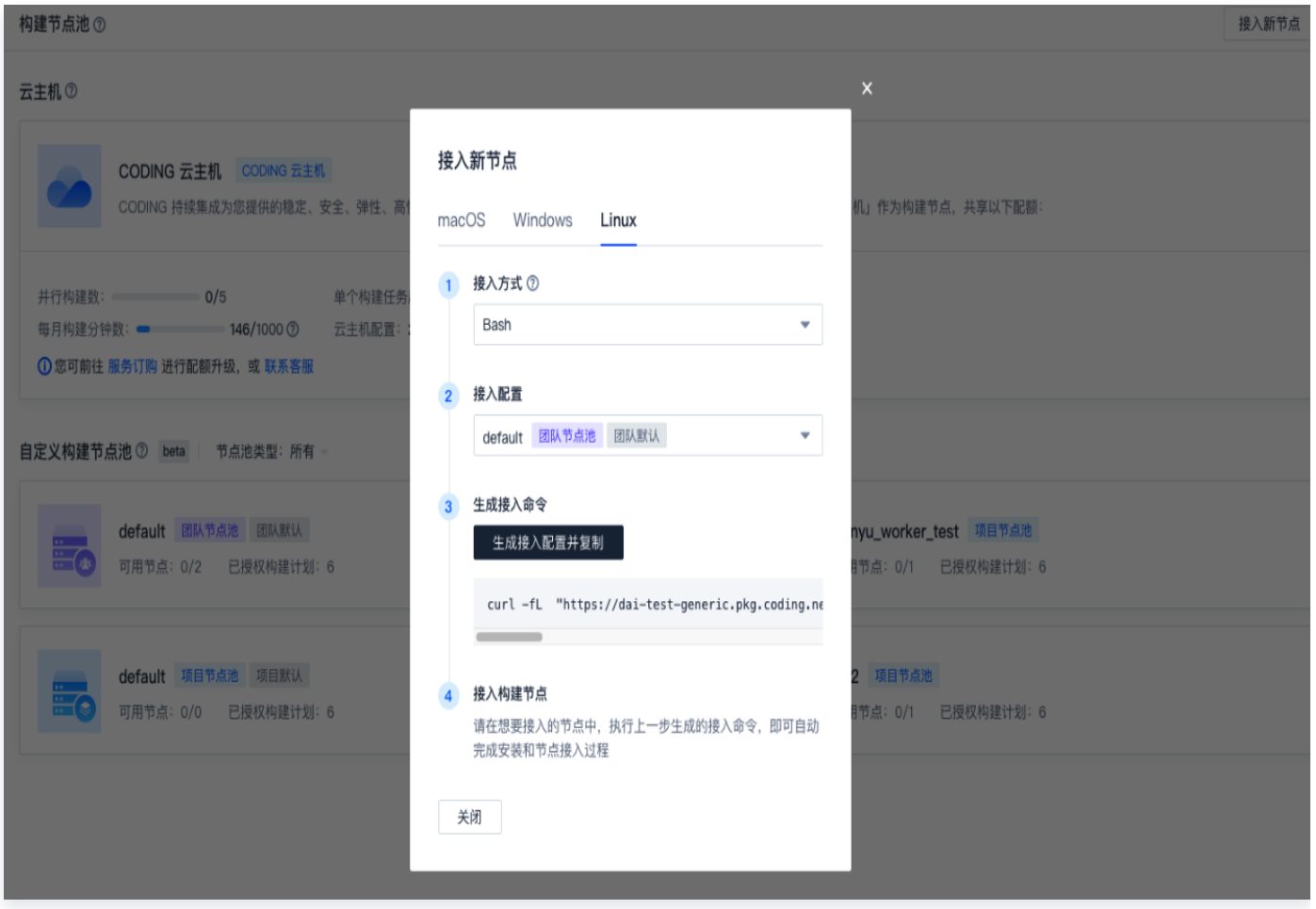
1. Select Manual Integration as the integration method, follow the prompts to enter the command in the terminal to install the client, that is, to install the `Worker` service.

2. Select the node pool you want to integrate, click **One-Click Generation and Copy** to generate the initialization command.
3. Execute the automatically generated client startup command in the terminal to keep the build node online.

Linux

Command Integration

1. Select **Continuous Integration > Build Nodes**, click the **Integrate New Node** button in the upper right corner, choose Linux, and select the Bash integration method. After configuration, run the generated integration command in the Linux environment.



2. After installation, you can use the following command for verification:

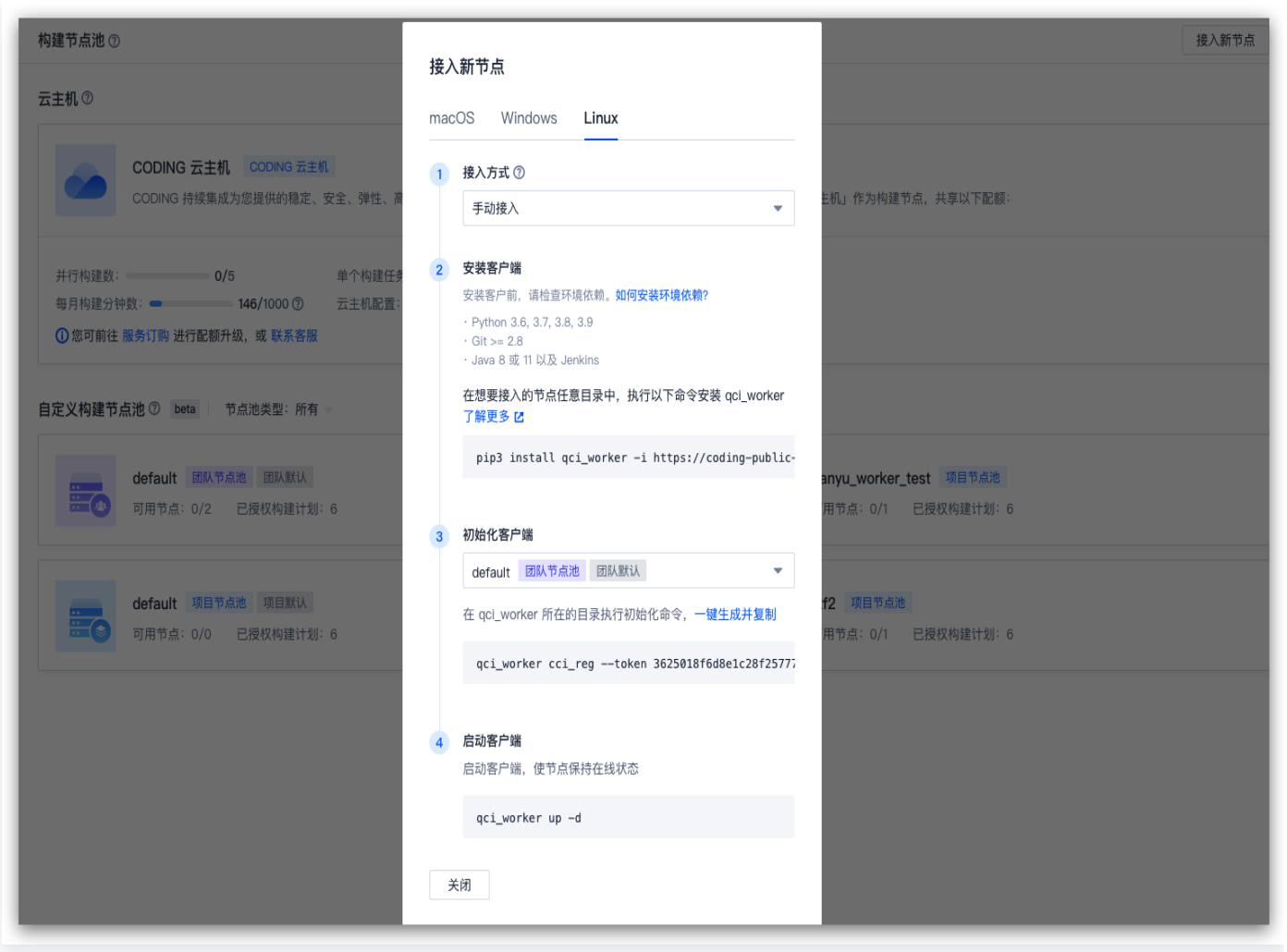
```
qci_worker version
```

The default installation directory for command integration is `/root/codingci`. If you want to specify the installation directory, please refer to Specify Installation Directory.

Manual Integration

Before using the manual integration method, please ensure that the node meets the Environment Dependencies requirements mentioned above.

1. Select Manual Integration as the integration method, follow the prompts to enter the command in the terminal to install the client, that is, to install the `Worker` service.



2. Select the node pool you want to integrate, click **One-Click Generation and Copy** to generate the initialization command.
3. Execute the automatically generated client startup command in the terminal to keep the build node online.

Start the Daemon Process

After installation, you need to run the daemon process on the build node to listen for and retrieve CI tasks issued by the CODING backend. Below are the run/delete command lines:

```
# Run in the background
qci_worker up -d

# Run in the foreground
qci_worker up

# Temporarily stop running
```

```
qci_worker stop
```

Worker Common Commands

Last updated: 2024-09-05 16:33:54

This article introduces common commands for the qci-worker service.

Prerequisites

Before configuring the CODING-CI build environment, you must activate the CODING DevOps service for your Tencent Cloud account. For details, please refer to Activate Services.

Open Project

1. log in to CODING console, click **Team Domain** to enter the CODING page.
2. After entering the target project, select **Continuous Integration** from the left navigation bar

Description of the Feature

When accessing from a Definition build node, the `Worker` service will be installed in the environment. This service schedules CI build tasks and allocates computing resources. Therefore, being familiar with common commands of the `Worker` service will help better coordinate CI build tasks. For installation methods on different operating systems, please refer to Definition Nodes.

Common Configuration Item Commands

Signup

```
qci_worker reg_cci  --token  token   --server server  --home home
--token Project Token, required
--server  Specified Access Service, not required
--home    Specify Working Directory
```

Example:

```
qci_worker cci_reg  --token db6fd4d6a2fc7d753a2985d55c44a2262f3e543f  --
server ws://codingcorp.nh113vufq.dev.coding.io --home ~/.codingqci
```

Startup Services

```
qci_worker up -d
```

Restarting Service

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

Manually Delete Node

```
qci_worker stop   #Stop qci_worker
qci_worker remove  #Remove Node in Background
```

Modifying Configuration

To apply the specified Jenkins configuration, you need to stop the Jenkins service process first, then restart the `qci_worker` service.

```
qci_worker config  JENKINS_HOST=127.0.0.1   #Specify Jenkins startup
host
qci_worker config  JENKINS_PORT=15740       #Specify Jenkins startup
port

qci_worker config JENKINS_ENCODING='UTF-8'
qci_worker config JENKINS_OPT="-Xms4096m -Xmx4096m" #Specify the JVM
memory allocation for Jenkins

qci_worker config PACKAGE_AUTO_UPDATE=0 #Turn off auto-updates
```

Build Node Pool

Last updated: 2024-09-05 16:34:08

This article introduces the build node pool.

Prerequisites

Before configuring the CODING-CI build environment, you must activate the CODING DevOps service for your Tencent Cloud account. For details, please refer to Activate Services.

Open Project

1. log in to CODING console, click **Team Domain** to enter the CODING page.
2. After entering the target project, select **Continuous Integration** from the left navigation bar

Description of the Feature

A build node pool is a collection of build nodes. When using self-defined build nodes, you need to integrate them into the build node pool and specify the build node pool by configuring the build plan node pool.

Permission Control

User groups must have **Team Build Node** permissions to create/delete build node pools and perform related operations. Navigate to the left sidebar **Team Settings Center > Global Settings > Team Permission Scheme** and check the appropriate permissions for the user group.

Multiple build node pools can be set within a single project. Each node pool can integrate multiple build nodes. You can view the node status and manage them in the node list under the build node pool details.

Node Status

- **Idle**: The build node is currently idle.
- **Occupancy**: The build node has been allocated for a build task.
- **Preparing**: The build node is preparing the build environment.
- **Enabled**: Only nodes in an enabled state can be allocated for use. Disabling a node will not affect the ongoing build tasks.
- **Delete**: The node will be disconnected from the CODING-CI service, but only the workspace and related configuration information will be deleted; previously generated global cache files will remain.

You can view the build records of nodes in the build node pool details.

1. The build node pool is by default authorized for all build plans. You can also choose to authorize it for specific build plans (multi-selection supported).

2. You can modify the corresponding node pool configuration in the basic information settings of the build plan. By default, the build plan uses the Cloud Host provided by CODING. Other node pools configured within the project can also be selected for build.