CODING DevOps

实践教程

腾讯云

文档目录

实践教程
使用持续集成快速构建应用
自动构建微信小程序

最近更新时间：2023-09-11 16:05:29

概括

本文将借助于 CODING 的持续集成，手把手带您实现一个微信小程序的持续集成环境，从构建、发布、通知实现自动化，帮您告别繁琐重复性的劳动，解决黯然神伤的烦恼。
整个实现流程大致如下：

1. 创建 CODING DevOps 项目。
2. 创建构建计划，配置微信小程序代码上传白名单。
3. 配置微信小程序代码上传私钥到环境变量中。
4. 配置企业微信的 webhook 地址到环境变量中。
5. 配置构建计划，分为 4 个步骤（检出、编译、上传新版本、发送通知）。

下面我们来一步步实现它。

前置准备

- CODING DevOps 项目。
- 具有管理员权限的微信小程序账号。
- 企业微信机器人 WebHook 地址。
- 将 示例仓库 导入至 CODING 代码仓库中。

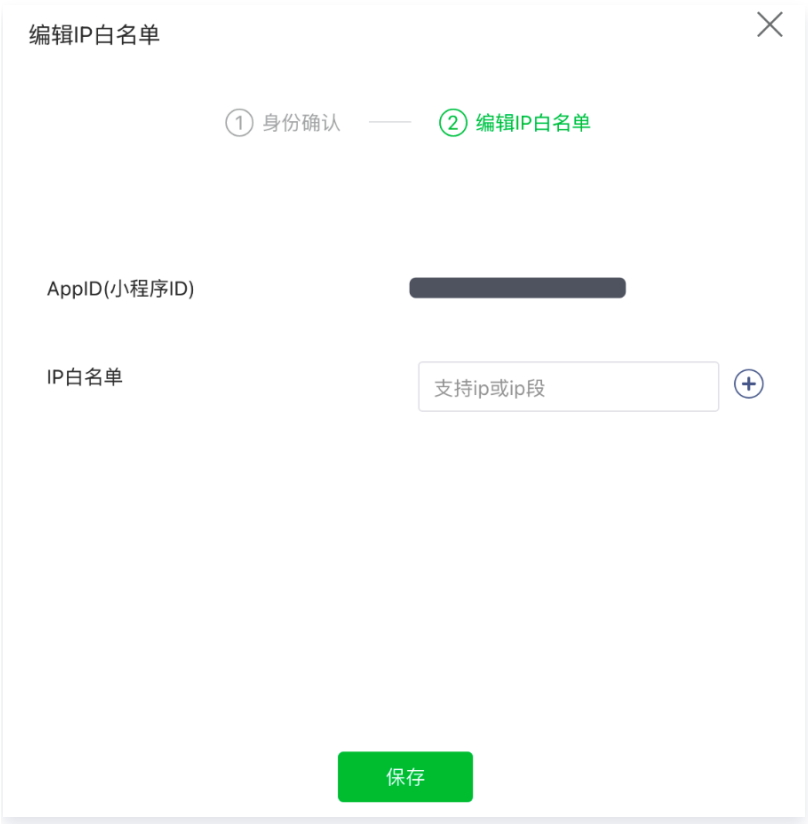
设置小程序白名单

此过程需要将构建任务的网络出口 IP 添加至小程序开发白名单中，您可以在构建计划的**基本信息**中获取出口 IP。


```
# 中国上海

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

# 中国香港

124.156.164.25/32
119.28.15.65/32

# 美国硅谷

170.106.136.17/32
```

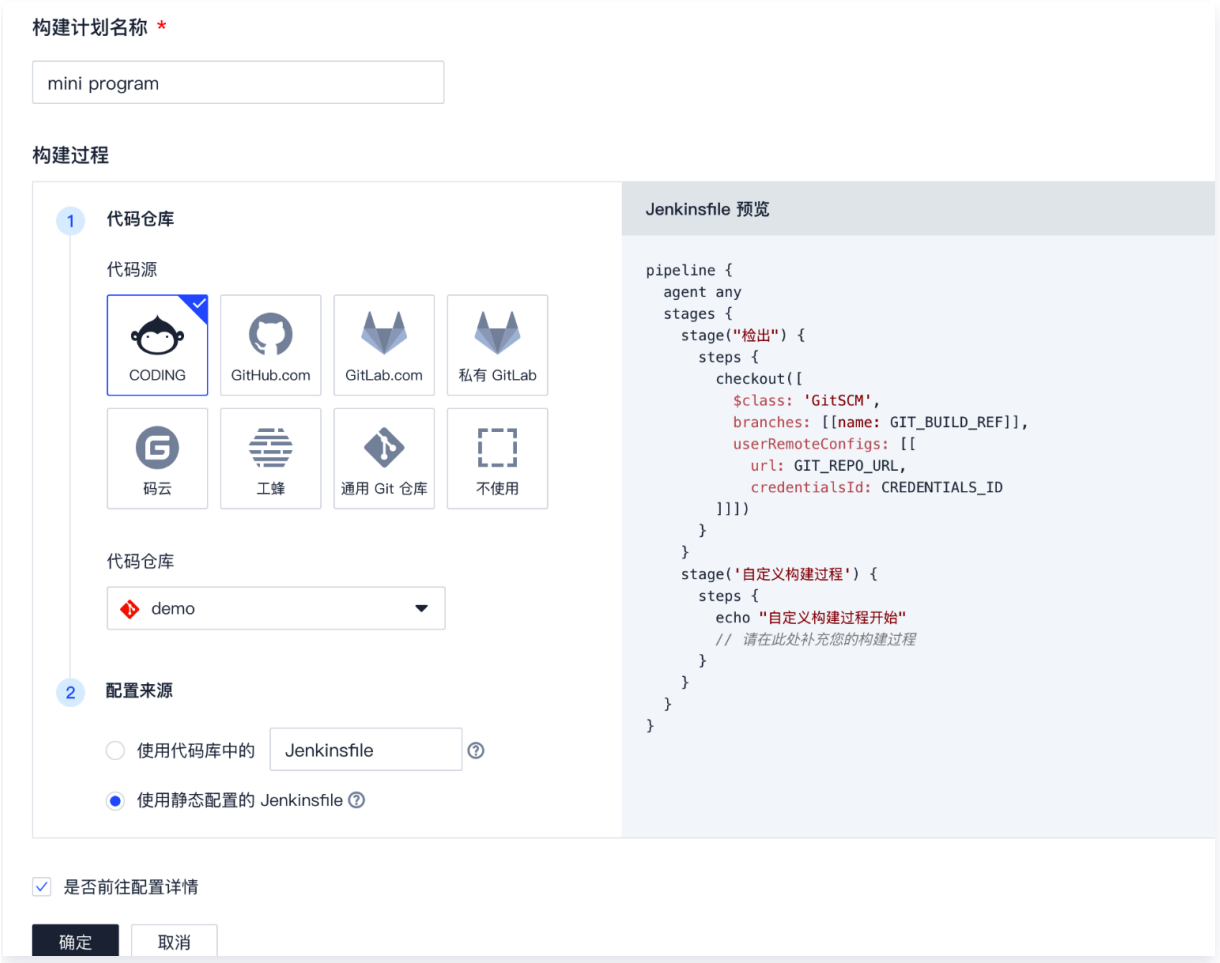
```
170.106.83.77/32
```

前往微信小程序的管理后台，单击左侧菜单栏中的**开发 > 开发者设置 > 小程序代码上传 > 编辑 IP 白名单**，添加需要的出口地址。



创建构建计划

在**持续集成**中新建构建计划，选择**自定义构建过程**模板。



在配置详情中参考 Jenkinsfile 编写构建过程。

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 '开始安装依赖'
```

```
            sh 'npm install'
            echo '开始构建...'
            sh 'npm run build'
            echo '构建完成'
        }
    }
    stage('上传新版本') {
        steps {
          withCredentials([sshUserPrivateKey(credentialsId:
"${env.privatekey}",keyFileVariable: 'identity')]) {
            sh 'node upload.js -p ${identity}'
          }
        }
    }
    stage('发送新版本通知') {
        steps {
            sh 'node notification.js -u ${WECHAT_WEBHOOK}'
        }
    }
  }
}
```

添加环境变量

持续集成过程中，我们总会将一些配置（例如：账号密码/版本号等）信息以环境变量的形式注入到构建过程中。在本实践中需要将以下两个凭据以环境变量的形式添加至构建计划中。

- 微信小程序代码上传私钥。
- 企业微信机器人 webhook 地址。

微信小程序代码上传私钥

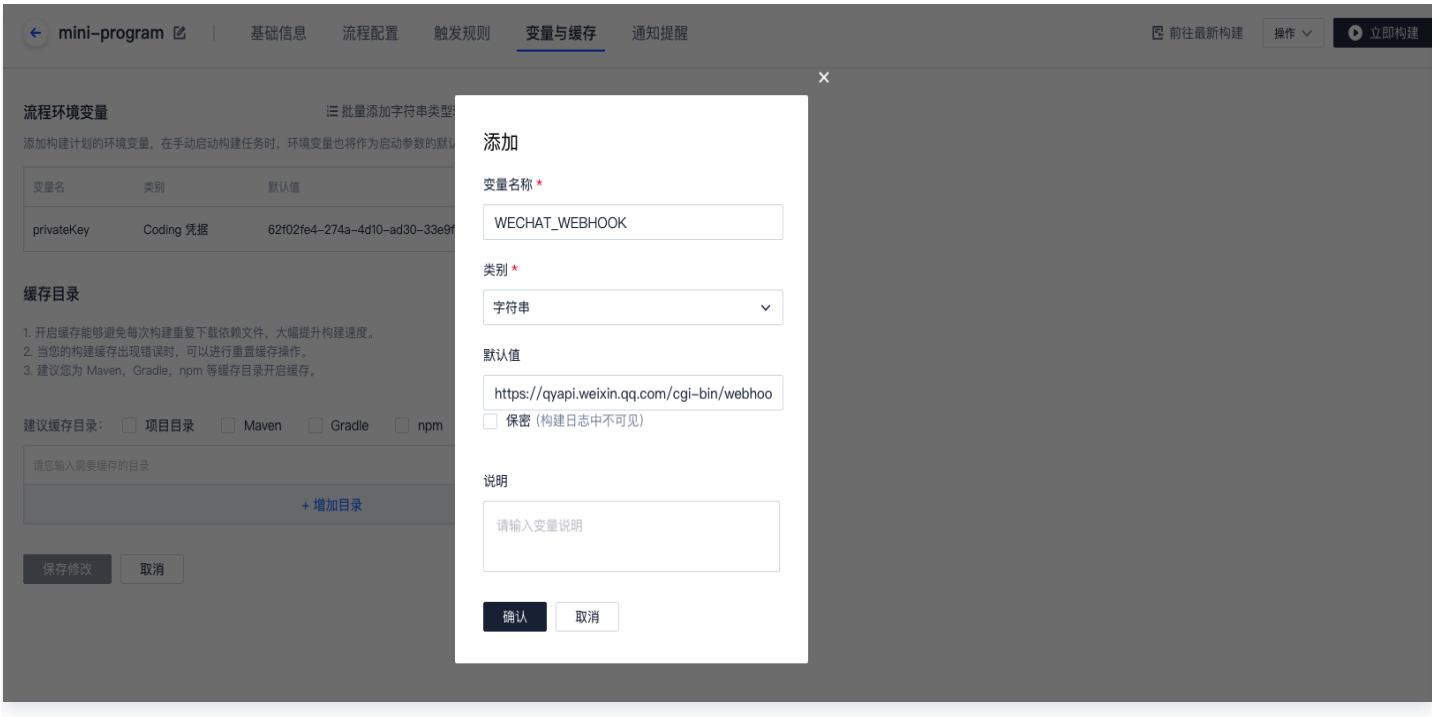
前往微信管理后台：**开发 > 开发设置 > 小程序代码上传**获取上传私钥与 AppID。

将信息导入至 CODING 项目中的**项目设置 > 开发者选项 > 凭据管理 > 录入凭据 > 选择 SSH 私钥凭据类型**，复制私钥内容粘贴至凭据中。CODING 会对您的私钥进行加密保存，杜绝明文暴露在工程文件中。同时还需要勾选**授权所有持续集成构建计划**。

创建完成后将生成一串凭据 ID，将其导入至变量与缓存中。

上传机器人 webhook

新建 群聊机器人 后，复制机器人的 webhook 地址后，以字符串的形式粘贴至**变量与缓存**中。



构建阶段细节

示例项目的代码是从微信开发者工具中抽离的关于小程序/小游戏项目代码的编译模块。开发者可不打开小程序开发者工具，独立使用已导入的示例仓库进行小程序代码的上传、预览等操作。
在上文中，我们将小程序上传代码的凭证加到环境变量，通过在 Jenkinsfile 定义 `withCredentials` 参数即可快速提取凭证。
提取到凭证后，调用了一个 upload.js 脚本。此部分代码涉及到了代码的上传和预览二维码的生成。

```
const ci = require('miniprogram-ci')
const path = require('path');
const fs = require("fs");
const argv = require('minimist')(process.argv.slice(2));
const package = require('./package.json')
const appDirectory = fs.realpathSync(process.cwd());
const ProjectConfig = require('./dist/project.config.json');
const previewPath = path.resolve(appDirectory, './preview.jpg');


(async () => {
    try {
        const project = new ci.Project({
            appid: ProjectConfig.appid,
            type: "miniProgram",
            projectPath: path.resolve(appDirectory, './dist'),
            privateKeyPath: argv.p,
            ignores: ["node_modules/**/*"],
        })
        await ci.upload({
            project,
            version: package.version,
            desc: package.versionDesc,
            setting: {
                ...ProjectConfig.setting
            },
            onProgressUpdate: console.log,
        })
        await ci.preview({
            project,
            version: package.version,
            desc: package.versionDesc,
            qrcodeFormat: "image",
            qrcodeOutputDest: previewPath,
            setting: {
```

```
                ...ProjectConfig.setting
            },
            onProgressUpdate: console.log,
        })
    } catch (e) {
        console.error(e);
        process.exit(1);
    }

})()
```

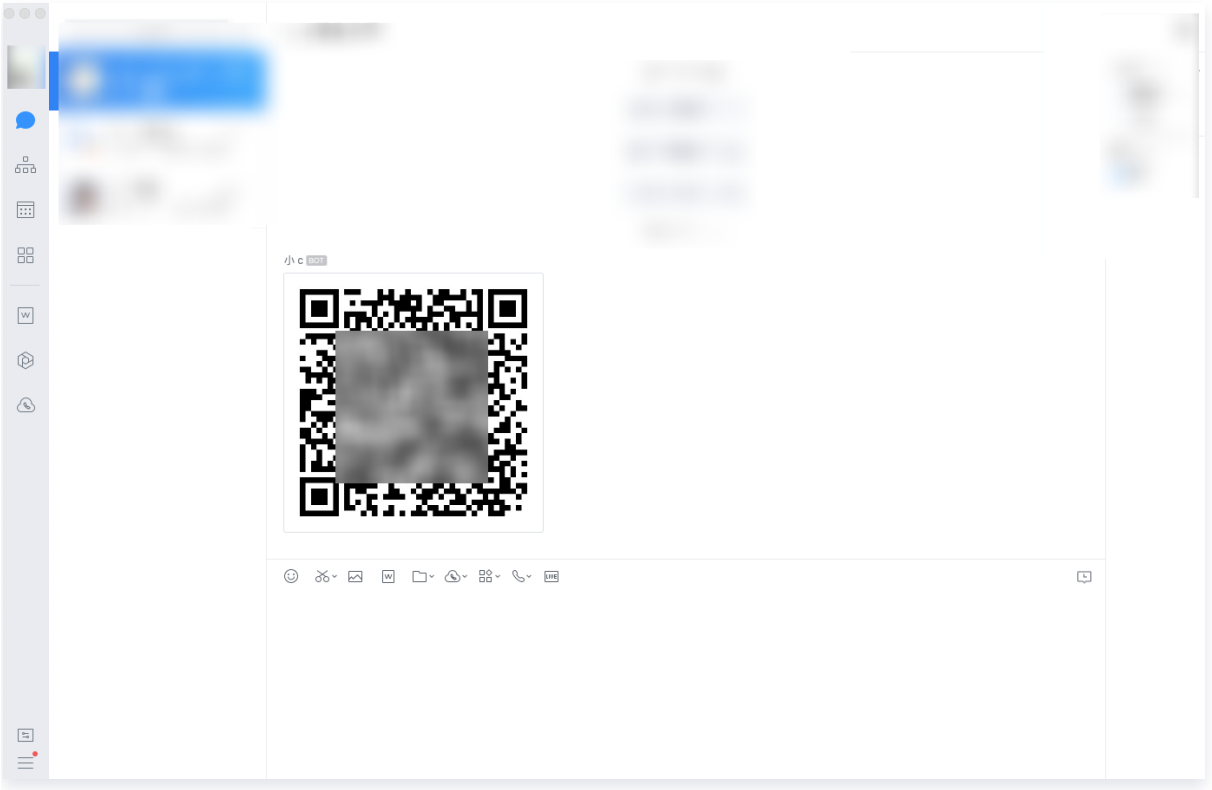
通知阶段

原理为直接发送请求，触发 webhook 后将发送预览二维码。关于企业微信 API 请参见 群机器人配置说明 。

```
const md5File = require('md5-file')
const axios = require('axios');
const path = require('path');
const argv = require('minimist')(process.argv.slice(2));
const fs = require("fs");
const appDirectory = fs.realpathSync(process.cwd());

const previewPath = path.resolve(appDirectory, './preview.jpg');


function sendQrCode (imageBase64, hash) {
    return axios({
        headers: { "Content-Type": 'application/json' },
        method: 'post',
        url: argv.u,
        data: {
            "msgtype": "image",
            "image": {
                "base64": imageBase64,
                "md5": hash
            }
        }
    });
}

(async () => {
    try {
        const imageData = fs.readFileSync(previewPath);
```

```
        const hash = md5File.sync(previewPath)
        const imageBase64 = imageData.toString("base64");
        await sendQrCode(imageBase64, hash);

    } catch(e) {
        console.error(e);
        process.exit(1);
    }
})()
```

当我们把代码上传，发布新版本之后，就会往企业微信群上发送一个预览二维码，通知群上的同事进行预览体验



更多扩展

版本号和版本说明没有集中管理，目前是读取 package.json 文件里的 version 和 versionDesc 参数。若需要进行版本控制，可以尝试通过 CODING 代码仓库的 tag 来管理版本，同时配置通过 tag 来触发构建。

自动化发布 AI 应用

最近更新时间：2023-09-11 16:05:29

本文将介绍如何基于 Tensorflow.js 开发一款 logo 识别应用，并且在此过程中演示如何通过 CODING DevOps 实现自动化构建。

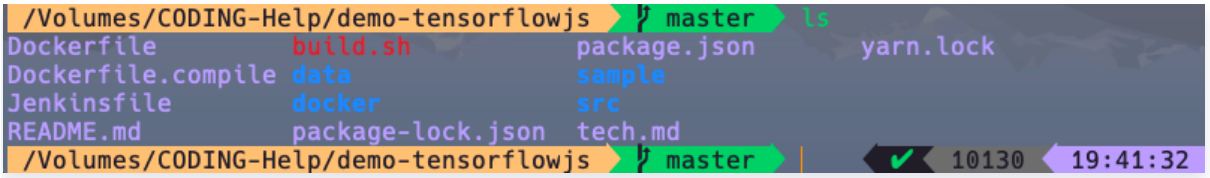
前置准备

- git
- nodejs
- yarn
- docker（仅本地构建与运行此应用需要）
- CODING 项目
- Docker 制品仓库，权限需设置为公开，制品仓库命名为 `build`

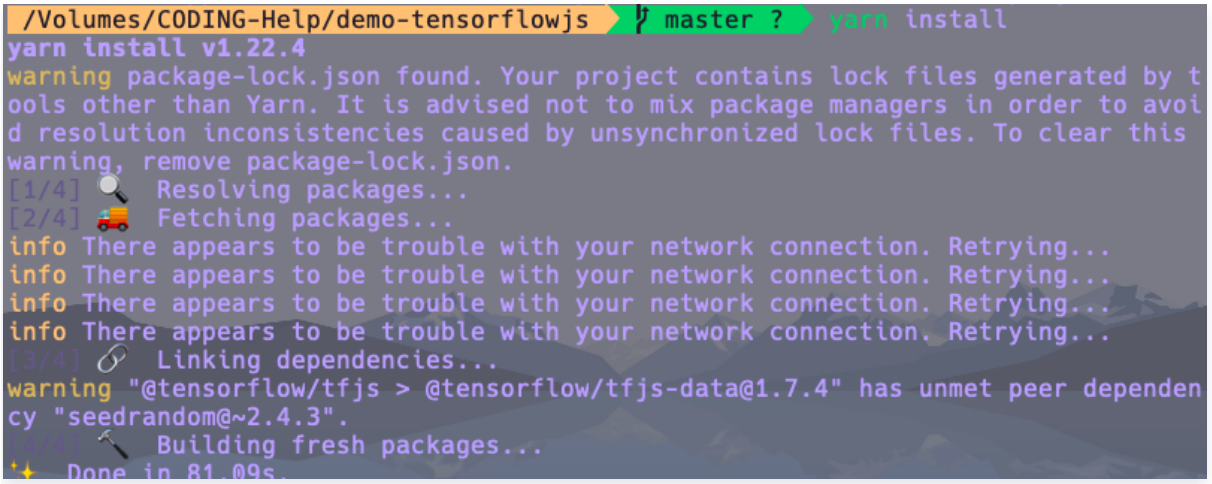
初始化

在创建代码仓库时选择**导入外部仓库**，粘贴示例仓库地址。

导入成功后，将代码拉取至本地中。



接下来可以运行 `yarn install` 命令并在本地安装依赖。



安装完成后运行 `./build.sh --local` 命令进行本地构建。应用开发完成后还需要进行容器化，以方便应用传播与测试。每次开发后都会生成一个新的制品，若要手动重复打包再上传至制品仓库，此过程未免过于繁琐。借助持续集成工具，能够在每次开发后自动触发构建并上传至制品仓库，解放生产力。并且在构建的过程中还能够配置通知机制，及时获得构建反馈。

创建构建计划

进入任一项目后，单击左侧菜单栏的**持续集成**，新建构建计划时选择**自定义过程**模板。

跳转至配置详情后参考以下 Jenkinsfile 修改构建过程。

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()
          }
        }

      }
    }
  }
  environment {
    CODING_DOCKER_REG_HOST = "${env.CCI_CURRENT_TEAM}-
docker.pkg.${env.CCI_CURRENT_DOMAIN}"
    CODING_DOCKER_IMAGE_NAME =
"${env.PROJECT_NAME.toLowerCase()}/${env.DOCKER_REPO_NAME}/${env.DOCKER_
IMAGE_NAME}"
  }
}
```

此流水线脚本大致分为三个阶段，检出阶段为拉取代码，构建阶段为运行构建脚本，推送阶段为把 docker 制品推送到制品库。构建阶段主要运行脚本文件 `build.sh` 主要内如下：

```
#!/bin/bash

docker build -t compiler -f Dockerfile.compile .

if [ "$1" = "--local" ]
then
    docker build -t logo-reg .
else
    docker build -f $DOCKERFILE_PATH -t
$CODING_DOCKER_IMAGE_NAME:$GIT_COMMIT $DOCKER_BUILD_CONTEXT
fi
```

此脚本的设计思路为分阶段构建方案。以 `Dockerfile.compile` 作为构建基础镜像的构建文件、`Dockerfile` 作为实际运行镜像的构建文件。使用这种方法可以使得实际运行的镜像不包含构建应用所需要的环境，大大减少镜像体积，构建后的镜像仅为 149 Mb。此外，此脚本还可以通过 `--push` 参数来灵活的区分云上构建环境与本地构建环境。

配置触发规则

持续集成支持多种触发方式，例如代码源触发、定时触发、API 触发及手动触发。其中代码源触发又可配置为推送到指定分支或标签触发，触发方式多样，可满足绝大部分场景需要。
如前言中所说，我们希望把更多的精力放在代码开发上，尽量减少构建所带来的干扰。因此可以设置触发规则，例如通过配置如下正则表达式，当分支名满足规则后即可自动触发构建。

```
^refs/(heads/(release|release-.*|build-.*|feat-.*|fix-.*|test-.*|mr/.*))
```

设置变量与缓存

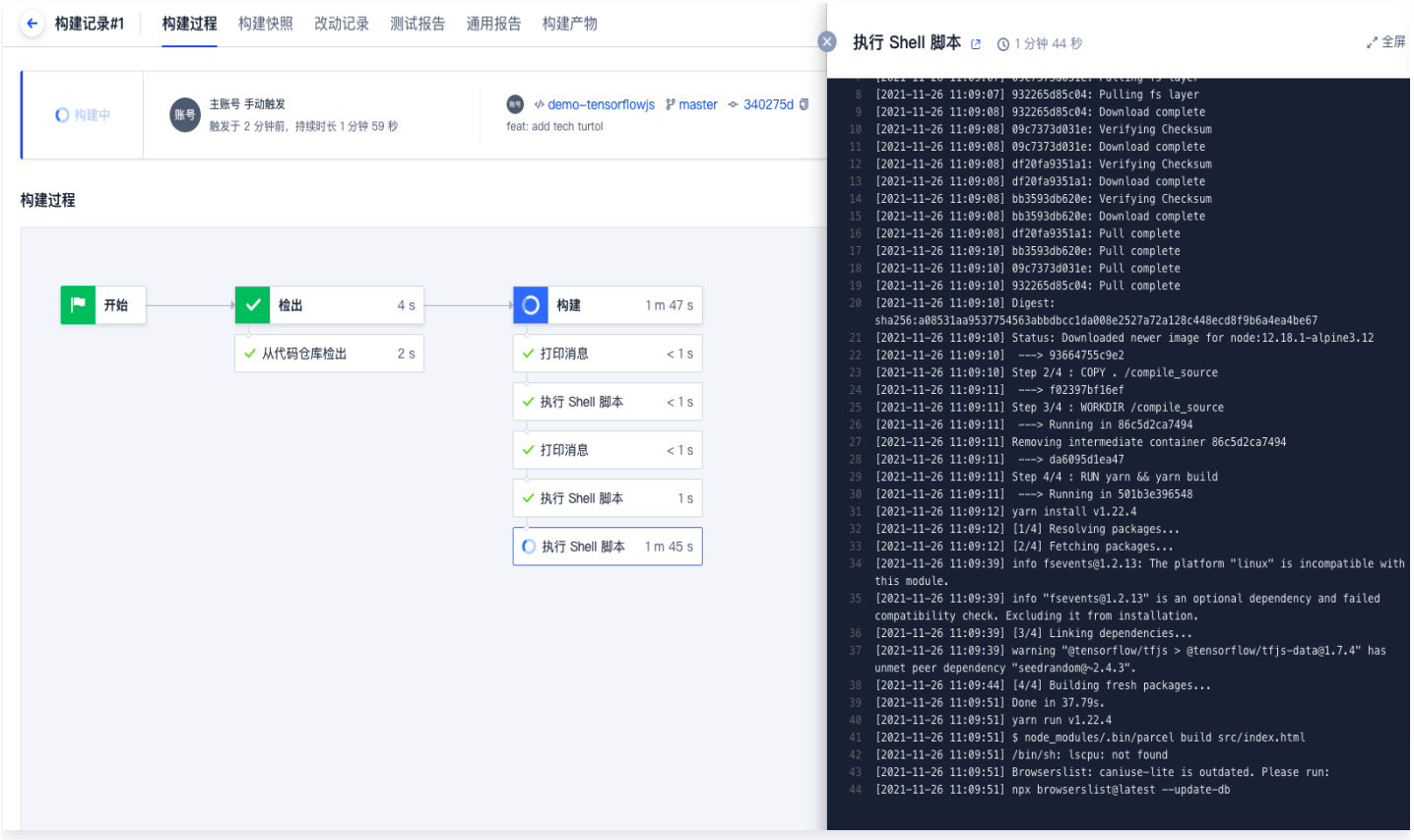
持续集成过程中，我们总会将一些配置（例如：账号密码/版本号等）信息以环境变量的形式注入到构建过程中。

所涉及的环境变量如下：

变量名	默认值
DOCKER_IMAGE_NAME	logo-reg
DOCKER_BUILD_CONTEXT	.
DOCKERFILE_PATH	Dockerfile
DOCKER_IMAGE_VERSION	${GIT_LOCAL_BRANCH:-branch}-${GIT_COMMIT}
DOCKER_REPO_NAME	build

执行构建

触发持续集成后，您可以在构建过程中看到各步骤的运行情况。



下载制品

构建完成后，可以看到在 build 制品仓库中已有新的制品，可以根据操作指引拉取至本地中。

运行应用

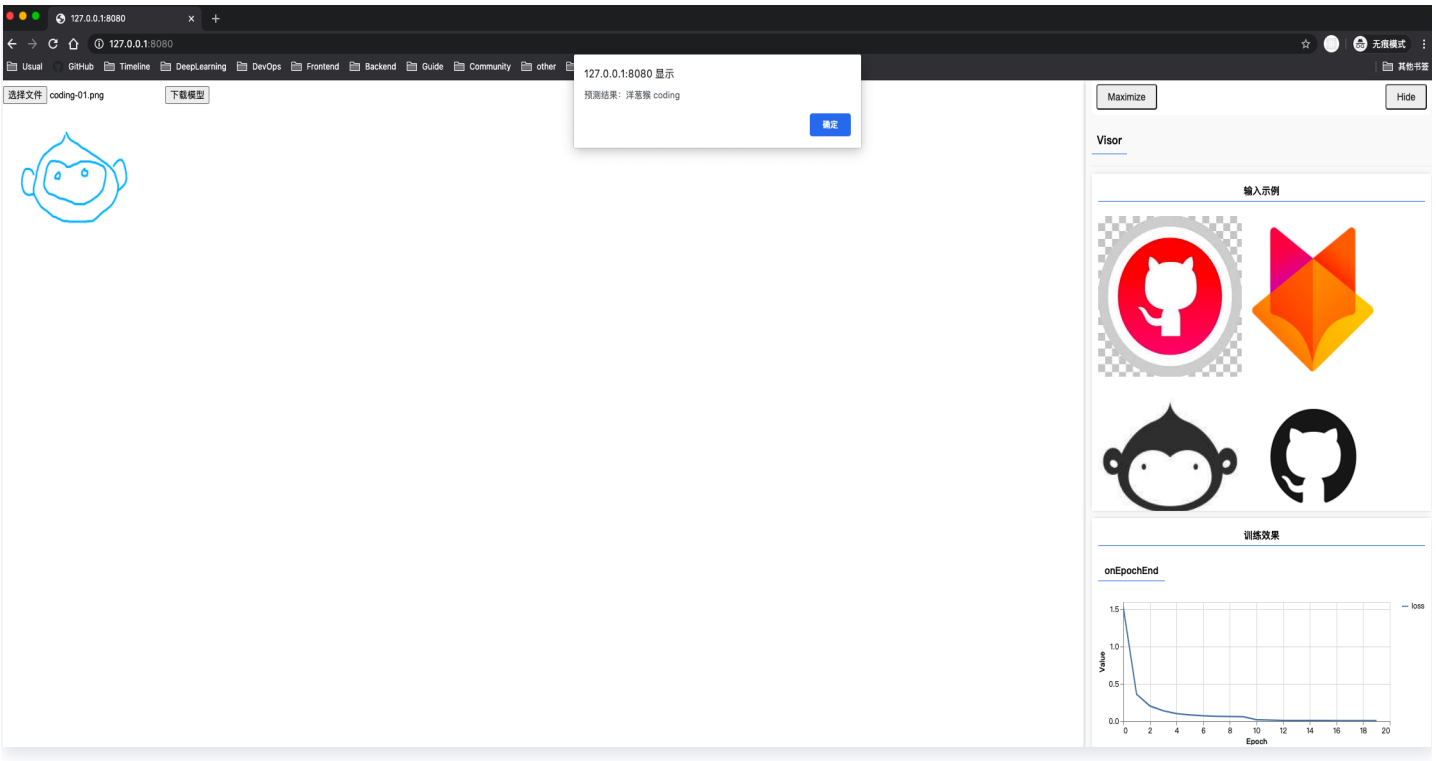
使用以下命令，运行已拉取的制品，即可开始通过机器学习以辨别 CODING、GitHub、GitLab Logo。

```
# 将命令中的仓库地址替换为自己制品仓库的地址。

docker run -p 8080:80 StrayBirds-docker.pkg.coding.net/demo/build/logo-reg:340275df1ecf1b2e7800a237ebceb10ceee7161c
```

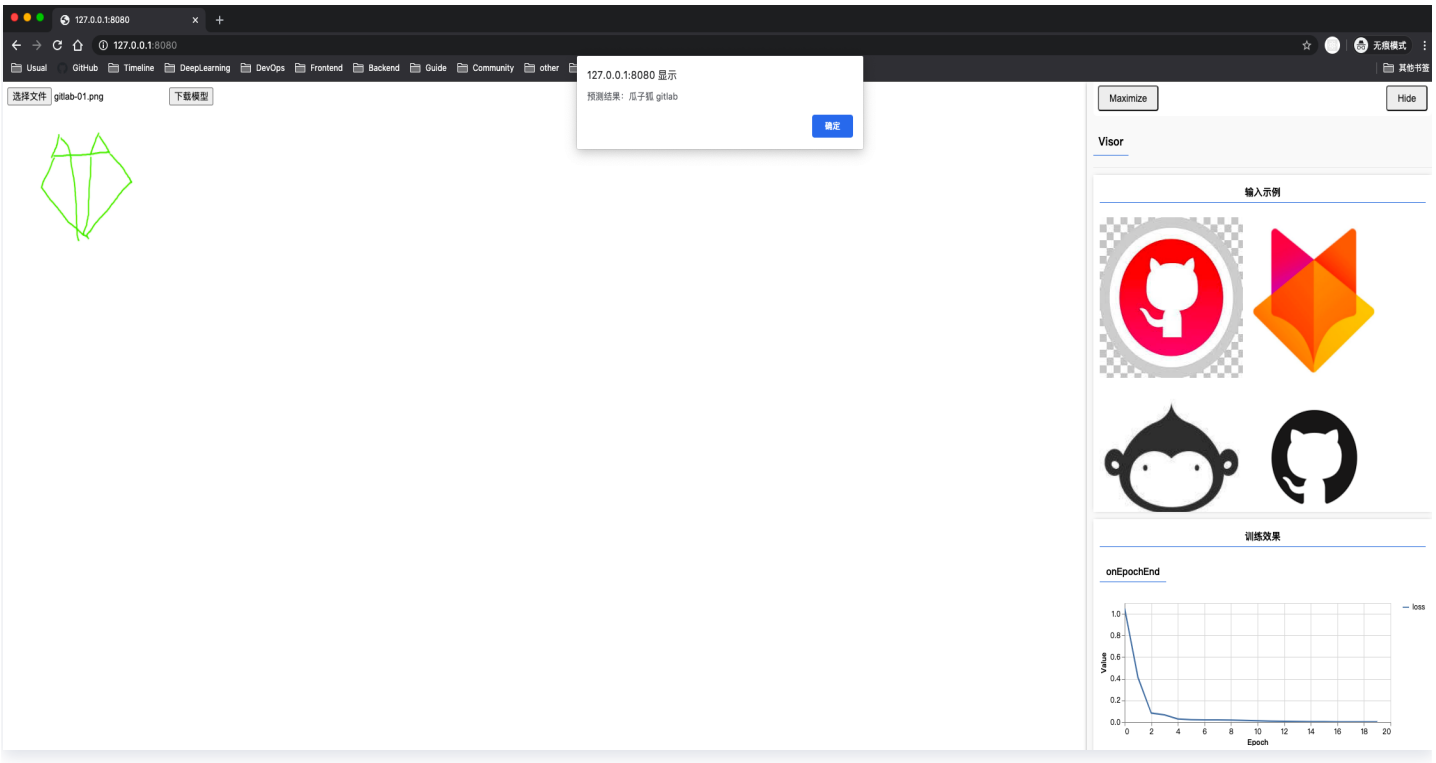
```
/Volumes/CODING-Help/demo-tensorflowjs  master  docker run -p 8080:80 Str
ayBirds-docker.pkg.coding.net/demo/build/logo-reg:340275df1ecf1b2e7800a237ebce
b10ceee7161c
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to per
form configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-def
ault.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf
.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/co
nf.d/default.conf
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates
.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes
.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2021/11/25 09:43:41 [notice] 1#1: using the "epoll" event method
2021/11/25 09:43:41 [notice] 1#1: nginx/1.21.4
2021/11/25 09:43:41 [notice] 1#1: built by gcc 10.2.1 20210110 (Debian 10.2.1-
6)
2021/11/25 09:43:41 [notice] 1#1: OS: Linux 5.10.25-linuxkit
2021/11/25 09:43:41 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1048576:1048576
2021/11/25 09:43:41 [notice] 1#1: start worker processes
2021/11/25 09:43:41 [notice] 1#1: start worker process 31
```

浏览器打开 `http://127.0.0.1:8080`，等待数分钟后，右下角训练损失几乎将为 0 即为训练完毕（若不为 0，说明训练过程收到不可逆干扰，请刷新页面即可重新训练）。上传任一 CODING、GitHub、GibLab 的图标文件，此应用可准确的预测出图片属于哪个 logo。

预测 CODING logo：



预测 GitLab logo：



可以看出，此应用具有相当高的准确率！

总结

本文通过一个基于 Tensorflow.js 开发的 AI 应用项目讲解了如何使用持续集成与制品仓库。借用 CODING DevOps 平台的这些功能， 我们解放本地算力，省去了人为的不必要劳动， 提高了生产力。
除此之外，持续集成可以构建任何应用（无论是终端、后端、甚至机器学习应用）。部署与构建不再是编程中的烦恼，专注于代码，专注于业务，繁琐之事皆可放行交由平台。

将项目发布至集群

最近更新时间：2023-09-11 16:05:29

本文将通过示例项目，演示如何从代码仓库开始，最终将项目自动化发布至集群中。

操作步骤

步骤1：导入示例代码仓库

先在团队创建全功能 DevOps 项目。

在代码仓库中导入开源 示例代码仓库 。

步骤2：创建持续集成任务

1. 选择项目内左侧产品栏**持续集成**，单击右上角**创建构建计划**，选择**部署**分类下的“推送到 Kubernetes”模板。

2. 示例仓库中已含有 `Dockerfile` 文件，因此仅需自定义镜像名称。若制品仓库中没有 Docker 类型仓库，可以在第三步单击页面上的**创建新的制品库**。

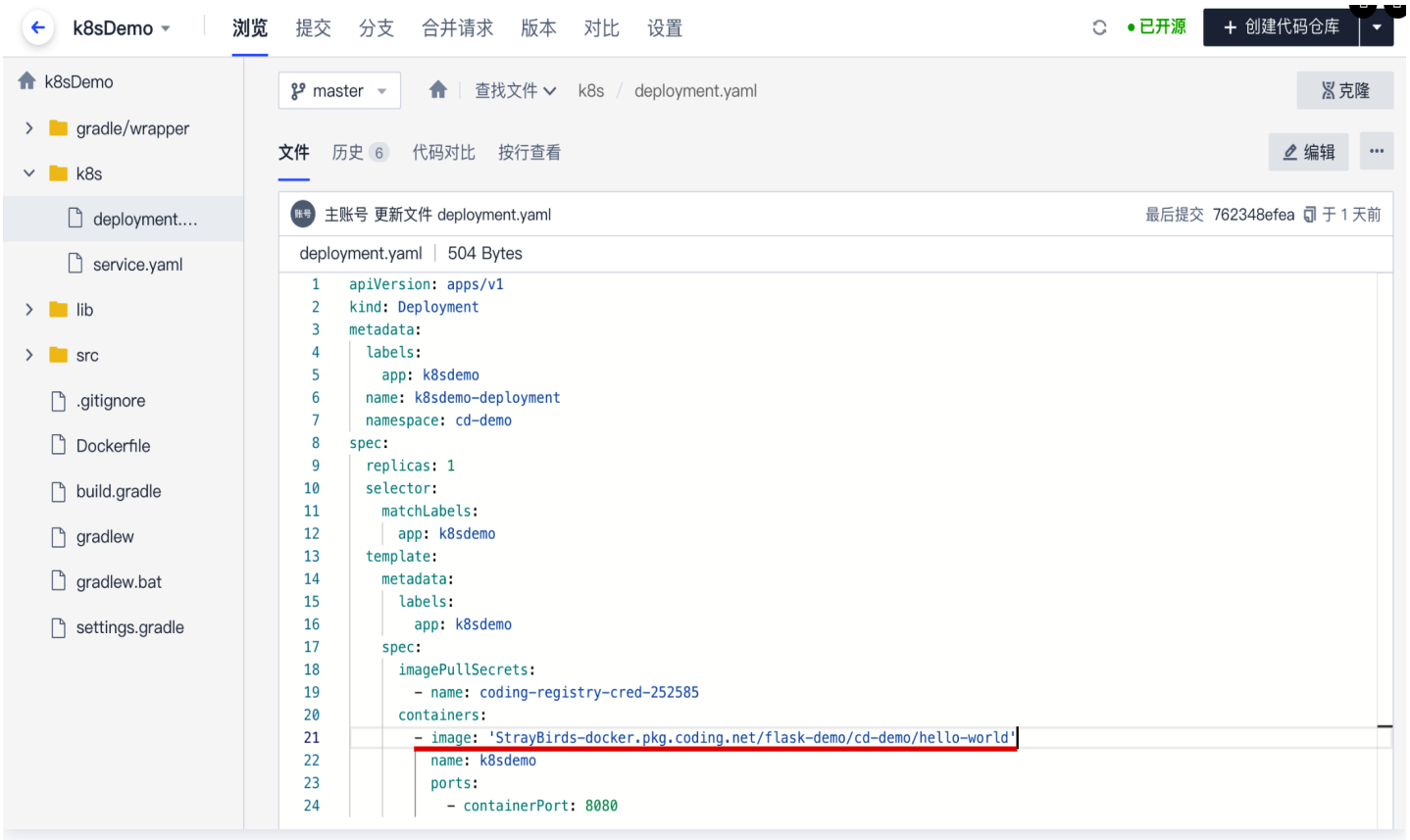
3. 创建完成后，前往持续集成设置的触发规则，勾选代码源触发。当 master 分支有代码变更时将自动触发持续集成任务，你也可以配合团队工作流，自定义多种自动触发方式。

步骤3：配置目标制品仓库

软件制品是指由源码编译打包生成的二进制文件，不同的开发语言对应着不同格式的二进制文件，这些文件通常可以直接运行在服务器上，用以支撑应用运行。持续集成任务运行成功后，前往制品仓库中就可以查看已推送的制品，您可以参见 Docker 在本地进行镜像推拉。

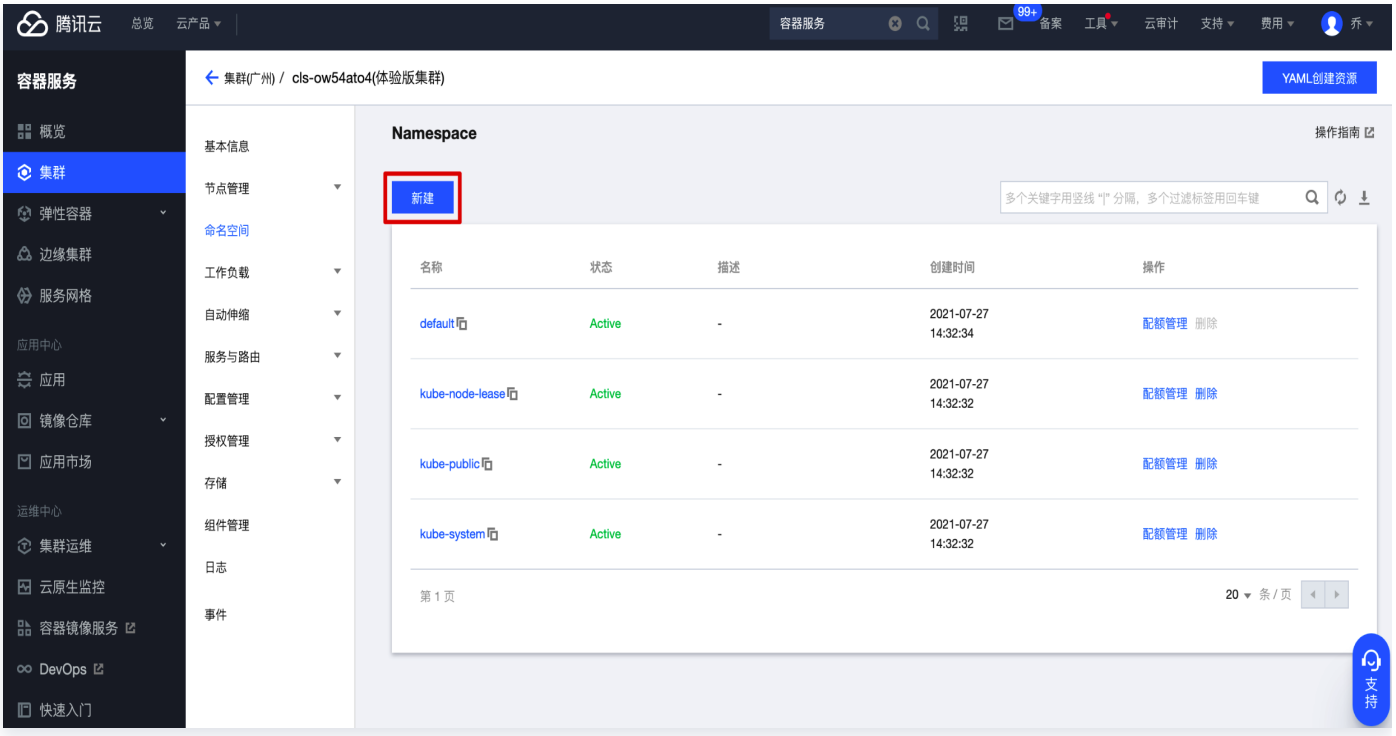
获取制品仓库的拉取链接。

将制品的拉取地址填写至代码仓库中 `/k8s/deployment.yaml` 中的 `image` 参数中。



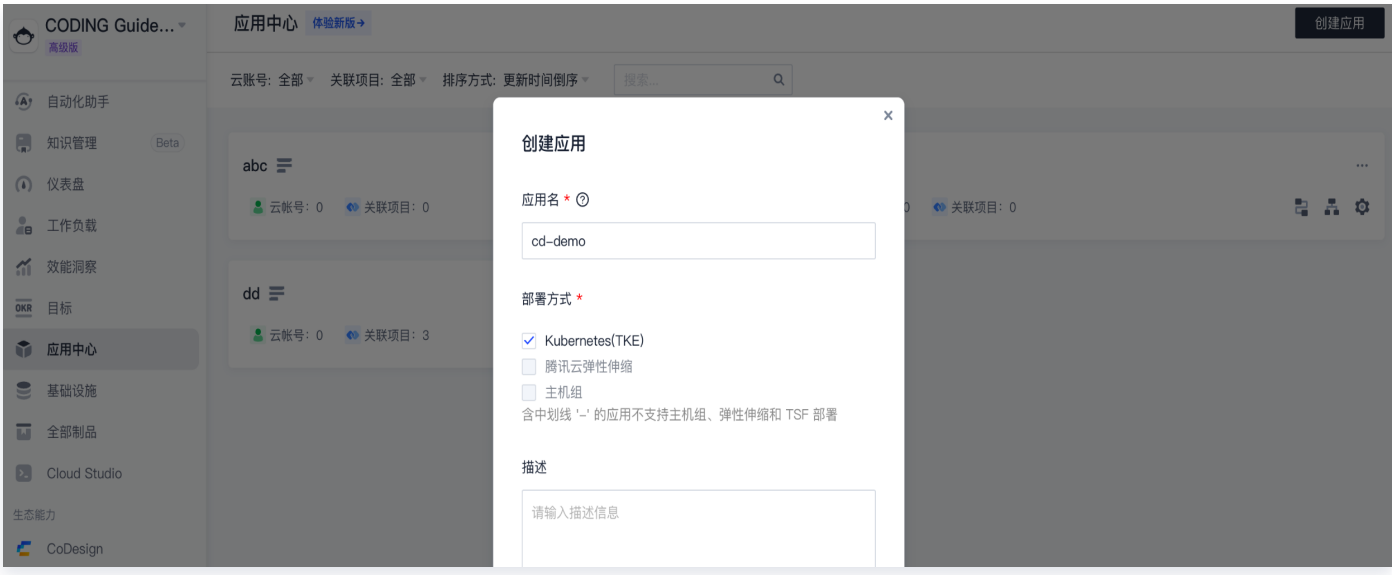
步骤4：创建云端容器服务

1. 前往腾讯云 控制台，单击**开通容器服务**。在集群中新建命名空间（Namespace）用于存储自动生成的制品仓库访问凭证，本文中所使用的集群命名为： cd-demo。

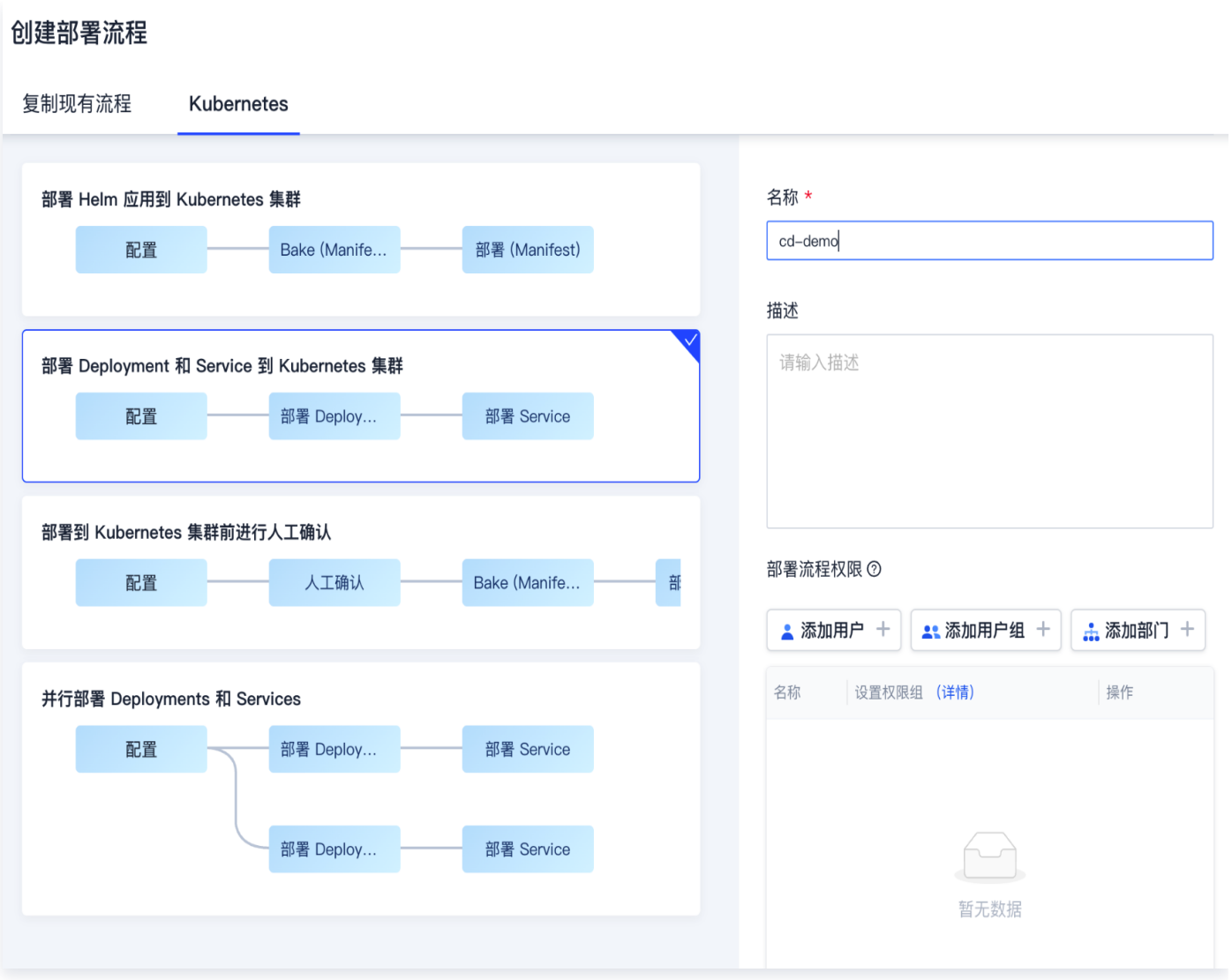


2. 新建命名空间后，返回 CODING 站点，单击首页左侧的**基础设施**，在**云账号**中绑定腾讯云账号。

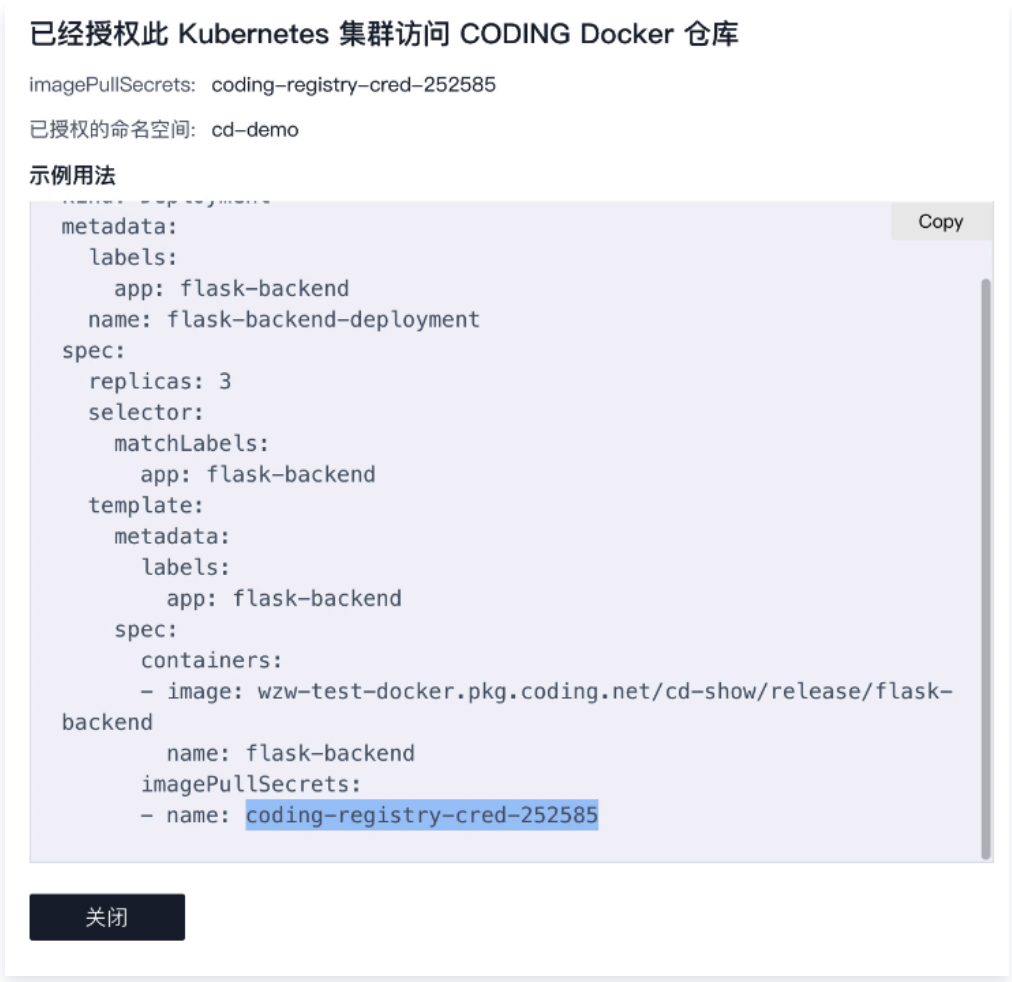
3. 成功添加云账号后，在**应用中心**（旧版）中单击**创建应用**，填写应用名与选择部署方式。



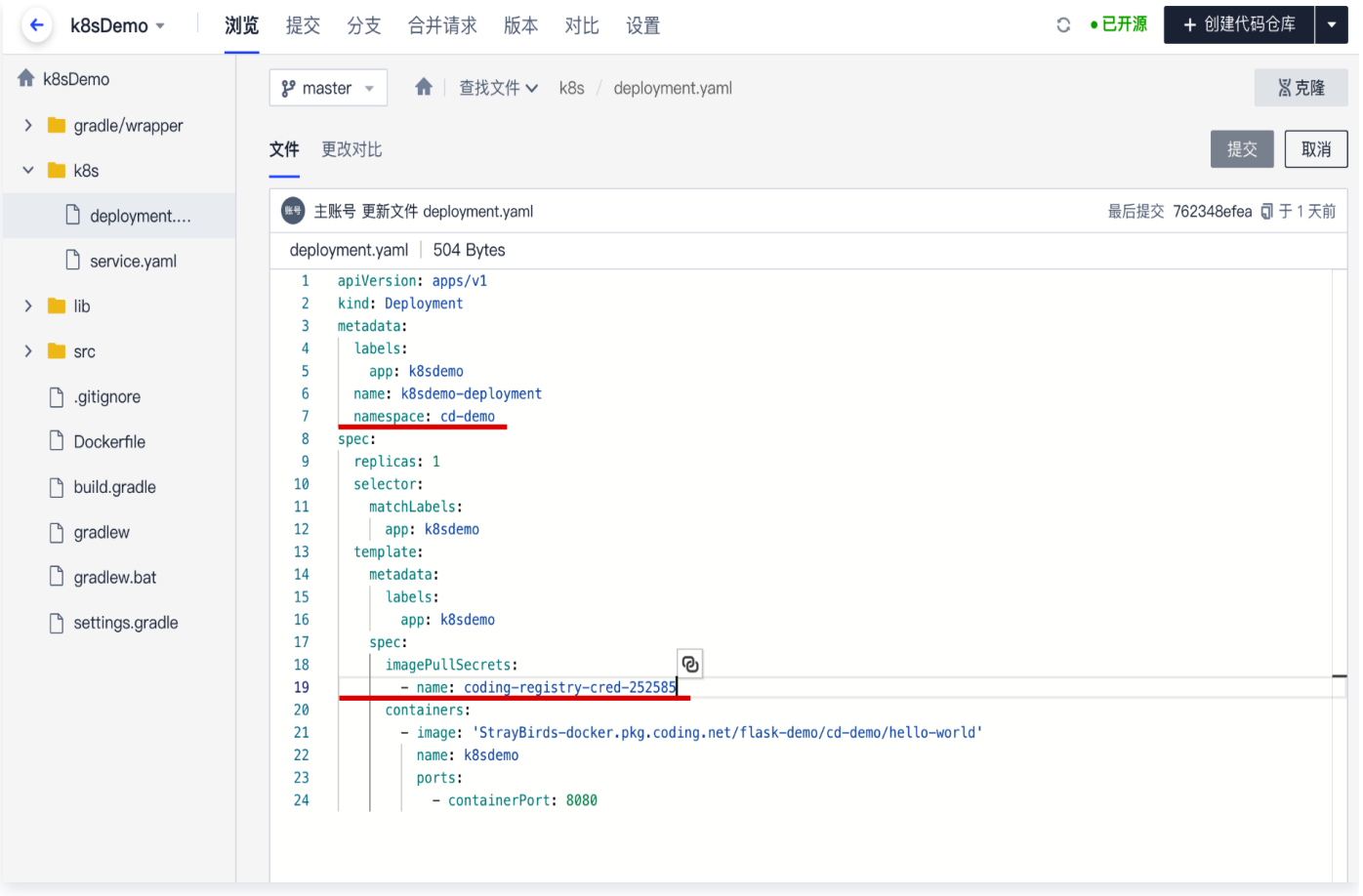
4. 选择**部署到 Kubernetes 集群**模板，填写名称与描述后完成创建。



5. 接下来需导入云账号的 `imagePullSecrets` 至代码仓库中。在**基础设施 > 云账号**中单击查看详情后，复制名称。



6. 粘贴至代码仓库中的 `deployment.yaml` 文件中，同时在 `namespace` 参数一栏中填写在上文中所创建的命名空间 `cd-demo` 。



同一层级的 `service.yaml` 文件中的 `namespace` 内容也需保持一致。

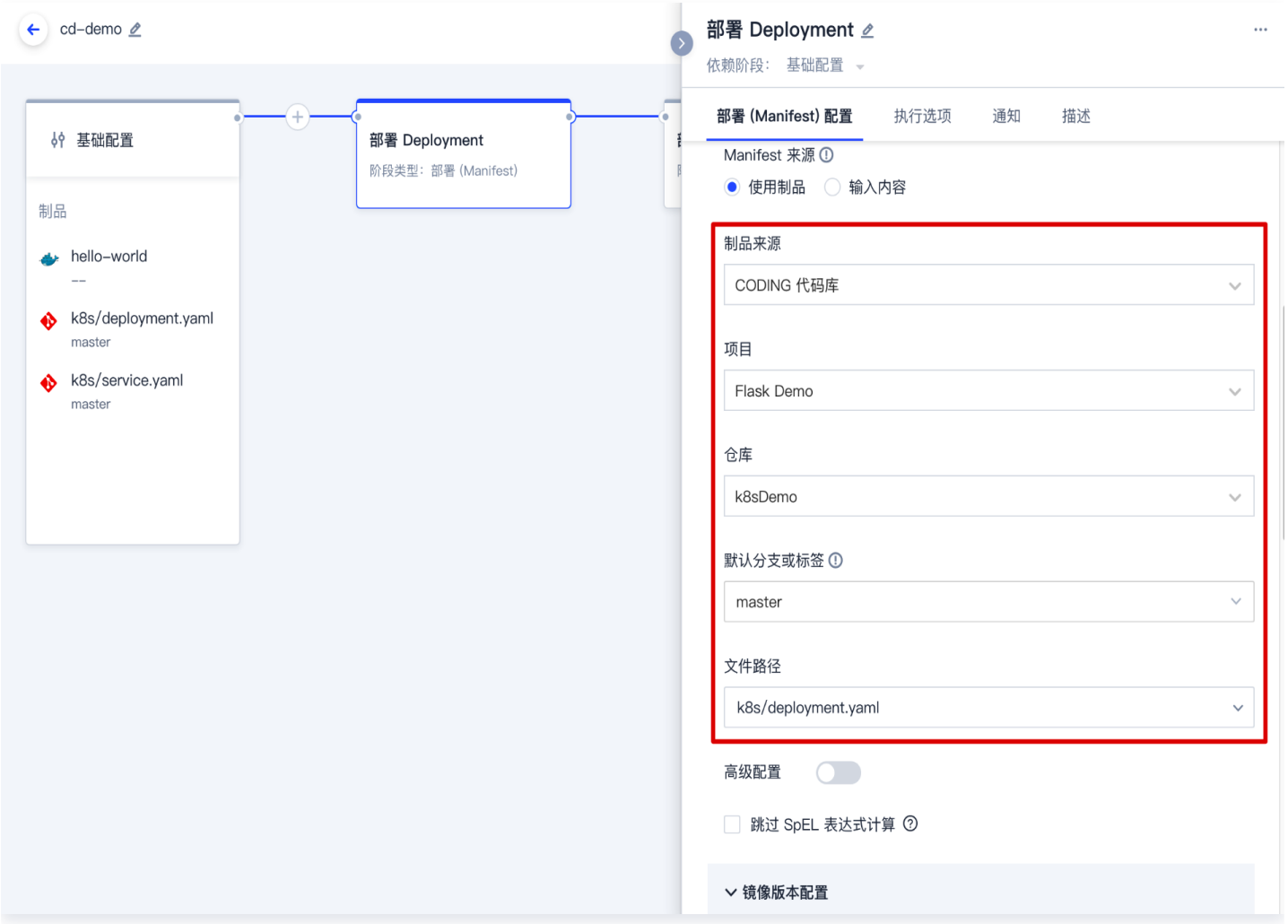
步骤5：自动化发布至集群

进入部署流程配置页面，可以为此流程设定：

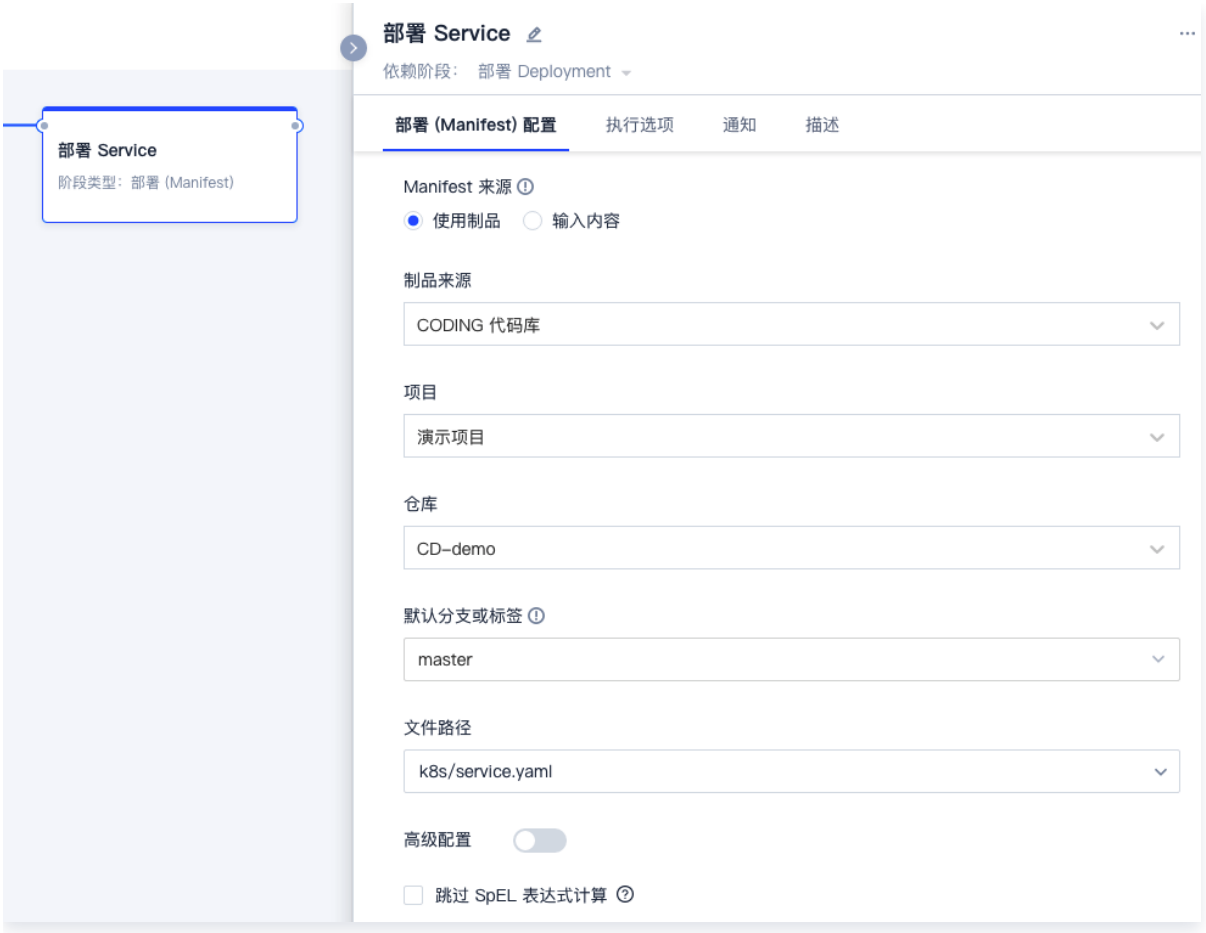
- 流程的执行选项（在此示例中保持默认即可）

- 部署 Deployment 阶段以及部署 Service 阶段所需制品
- 手动或自动触发

首先配置部署（Manifest）阶段。基础设置选择已绑定的云账号，在 Manifest 来源选择：**CODING 代码库**，填写相应的路径。

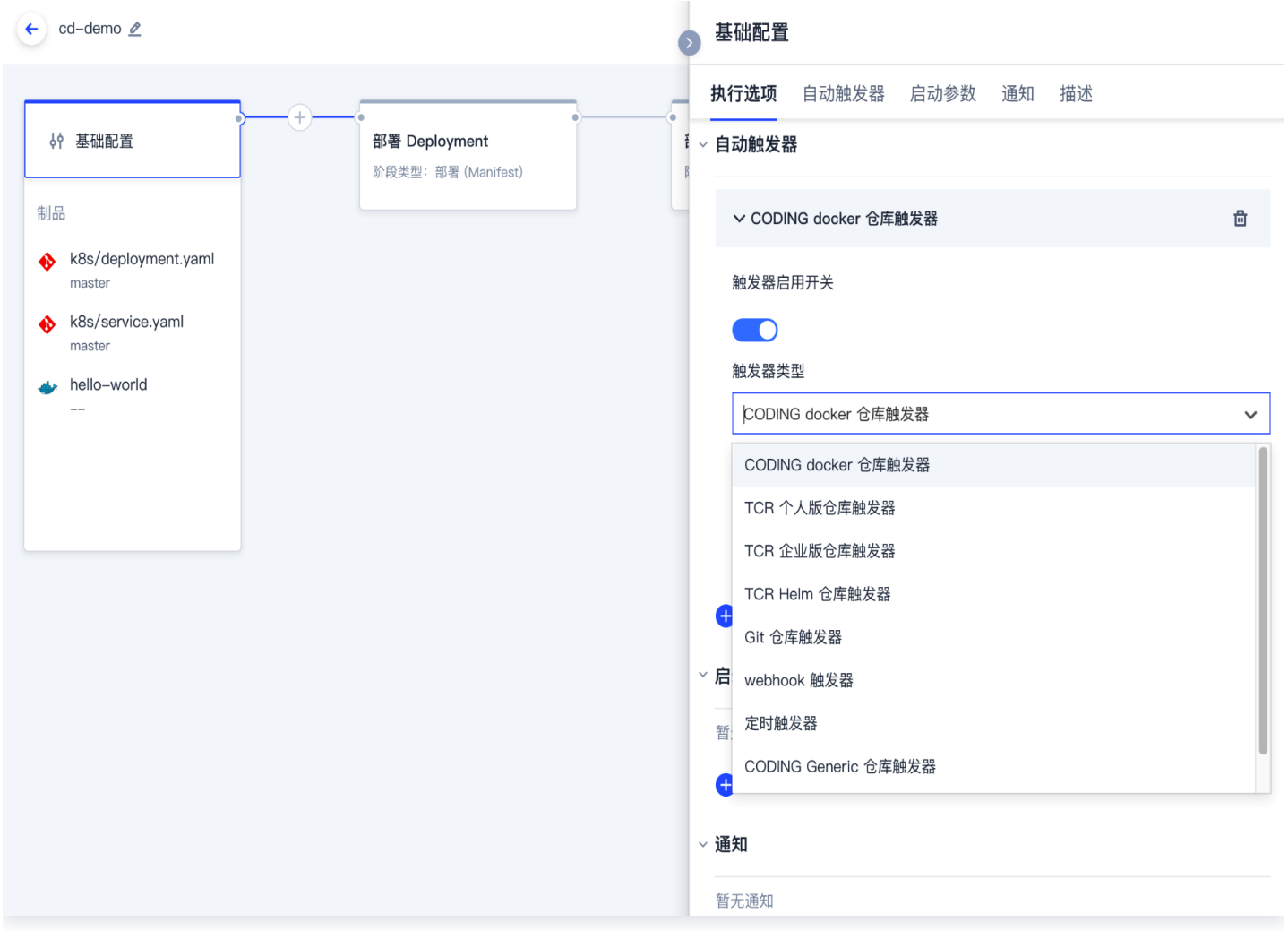


配置部署 Service 阶段时步骤同上，但在文件路径处需选择 `k8s/service.yaml` 文件。



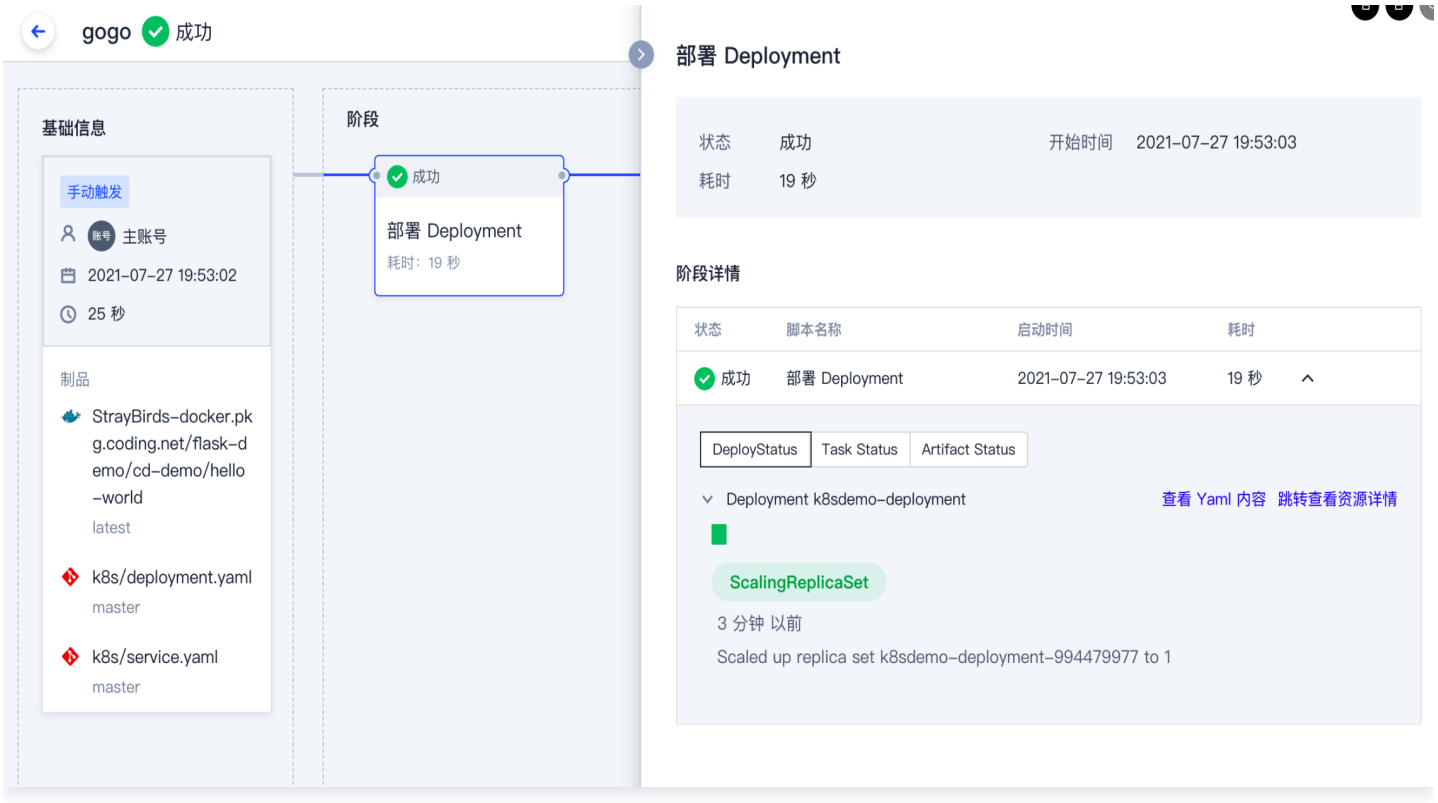
镜像版本配置默认选择自动获取镜像来源。若设置自定义版本规则，将仅传送特定的 image 版本信息号至集群中。

完成部署阶段配置后，您可以在**基础配置**中选择触发器类型，选择 Docker 仓库触发器。当开发人员更新代码仓库并使用 CI 将镜像打包推送至制品库后，Docker 镜像的更新将自动触发部署流程，并将应用发布至 Kubernetes（TKE）集群，完成后可以在基础设施页面查看并确认应用是否发布成功。

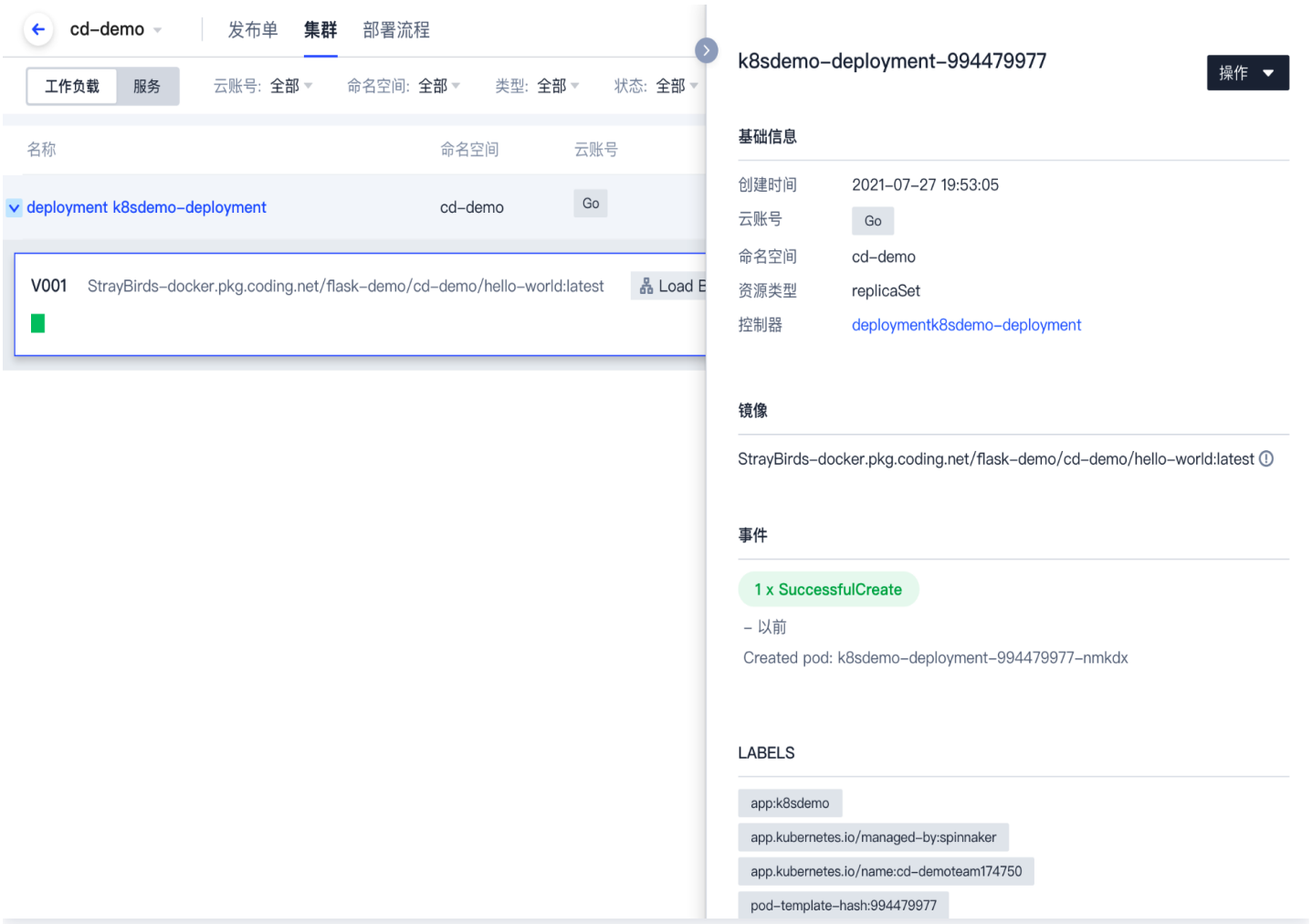


步骤6：发布成功

发布成功后，可以查看发布的制品及启动参数及阶段执行详情等信息。



当需要查看某个资源在集群中的运行状态时，单击集群下的工作负载即可查看详情（例如工作负载的 Pod 实例，日志等信息）。



在腾讯云的容器服务中查看工作负载。