一文详解 Kubernetes 的自动化部署实践

雨歌 DevOps时代 今天

本文从实践角度介绍如何结合我们常用的 Gitlab 与 Jenkins，通过 K8s 来实现项目的自动化部署，示例将包括基于 SpringBoot 的服务端项目与基于 Vue.js 的 Web 项目。

本文涉及到的工具与技术包括：

- Gitlab —— 常用的源代码管理系统
- Jenkins，Jenkins Pipeline —— 常用的自动化构建、部署工具，Pipeline 以流水线的方式将构建、部署的各个步骤组织起来
- Docker，Dockerfile —— 容器引擎，所有应用最终都要以 Docker 容器运行，Dockerfile 是 Docker 镜像定义文件
- Kubernetes —— Google 开源的容器编排管理系统
- Helm —— Kubernetes 的包管理工具，类似 Linux 的 yum，apt，或 Node 的 npm 等包管理工具，能将 Kubernetes 中的应用及相关依赖服务以包（Chart）的形式组织管理

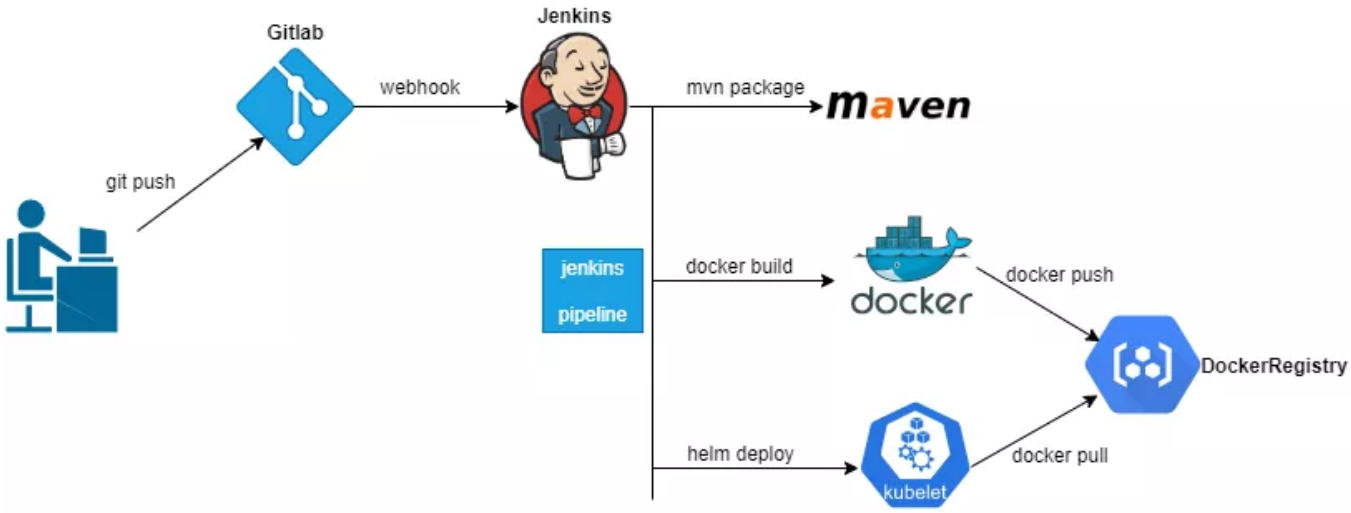
环境背景：

1. 已使用 Gitlab 做源码管理，源码按不同的环境建立了 develop（对应开发环境），pre-release（对应测试环境），master（对应生产环境）分支

2. 已搭建了 Jenkins 服务
3. 已有 Docker Registry 服务，用于 Docker 镜像存储（基于 Docker Registry 或Harbor 自建，或使用云服务，本文使用阿里云容器镜像服务）
4. 已搭建了 K8s 集群

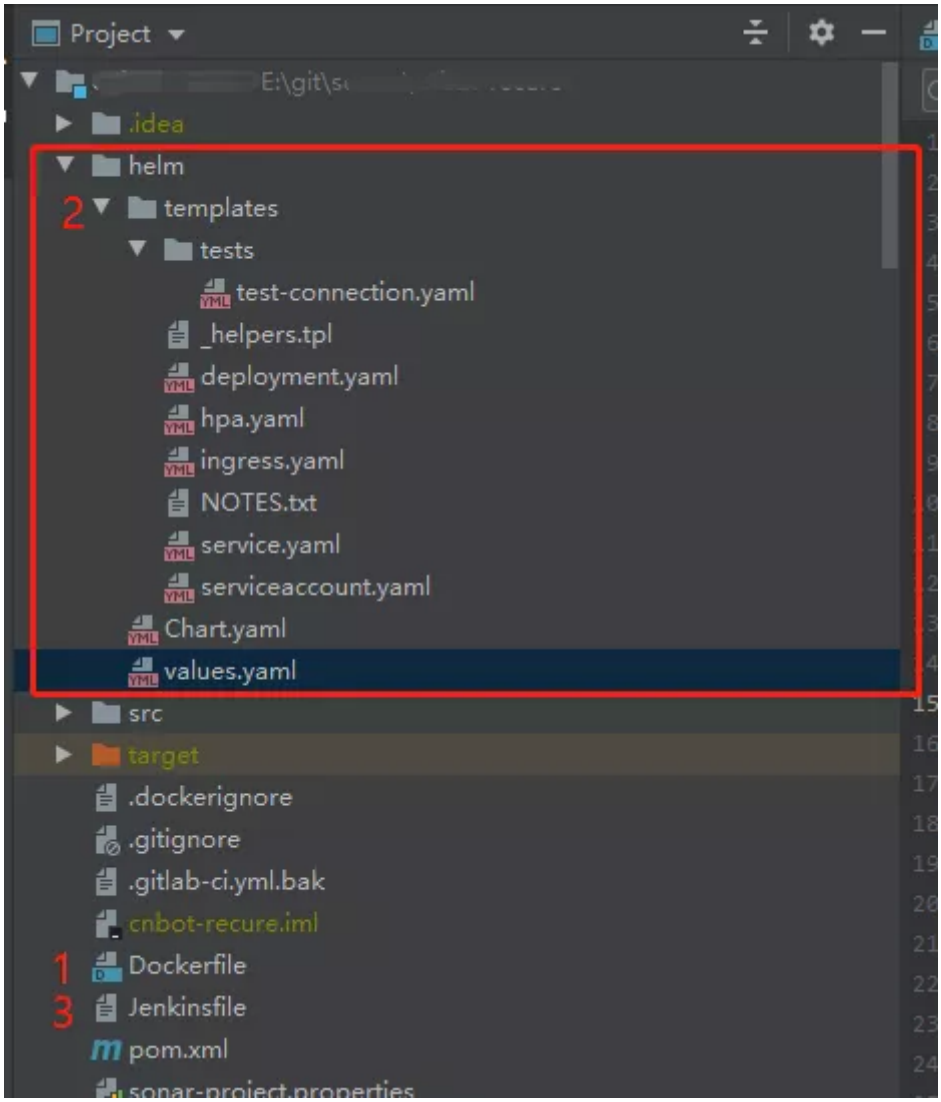
预期效果：

1. 分环境部署应用，开发环境、测试环境、生产环境分开来，部署在同一集群的不同 namespace，或不同集群中（比如开发测试部署在本地集群的不同 namespace中，生产环境部署在云端集群）
2. 配置尽可能通用化，只需要通过修改少量配置文件的少量配置属性，就能完成新项目的自动化部署配置
3. 开发测试环境在push代码时自动触发构建与部署，生产环境在 master 分支上添加版本 tag 并且 push tag 后触发自动部署
4. 整体交互流程如下图



项目配置文件

首先我们需要在项目的根路径中添加一些必要的配置文件，如下图所示



包括：

1. Dockerfile 文件，用于构建 Docker 镜像的文件（参考 Docker笔记（十一）：Dockerfile 详解与最佳实践）
2. Helm 相关配置文件，Helm 是 Kubernetes 的包管理工具，可以将应用部署相关的 Deployment，Service，Ingress 等打包进行发布与管理（Helm 的具体介绍我们后面再补充）
3. Jenkinsfile 文件，Jenkins 的 pipeline 定义文件，定义了各个阶段需执行的任务

Dockerfile

在项目根目录中添加一个 Dockerfile 文件（文件名就叫 Dockerfile），定义如何构建Docker 镜像，以 Spring Boot 项目为例，

```
FROM frolvlad/alpine-java:jdk8-slim
#在build镜像时可以通过 --build-args profile=xxx 进行修改
```

```
ARG profile
ENV SPRING_PROFILES_ACTIVE=${profile}
#项目的端口
EXPOSE 8000
WORKDIR /mnt

#修改时区
RUN sed -i 's/dl-cdn.alpinelinux.org/mirrors.ustc.edu.cn/g' /etc/apk/repositor
    && apk add --no-cache tzdata \
    && ln -sf /usr/share/zoneinfo/Asia/Shanghai /etc/localtime \
    && echo "Asia/Shanghai" > /etc/timezone \
    && apk del tzdata \
    && rm -rf /var/cache/apk/* /tmp/* /var/tmp/* $HOME/.cache

COPY ./target/your-project-name-1.0-SNAPSHOT.jar ./app.jar
ENTRYPOINT ["java", "-jar", "/mnt/app.jar"]
```

将 SPRING_PROFILES_ACTIVE 通过参数 profile 暴露出来，在构建的时候可以通过 —build-args profile=xxx 来进行动态设定，以满足不同环境的镜像构建要求。

> SPRING_PROFILES_ACTIVE 本可以在 Docker 容器启动时通过 docker run -e SPRING_PROFILES_ACTIVE=xxx 来设定，因这里使用 Helm 进行部署不直接通过docker run 运行，因此通过 ARG 在镜像构建时指定

Helm 配置文件

Helm 是 Kubernetes 的包管理工具，将应用部署相关的 Deployment，Service，Ingress 等打包进行发布与管理（可以像 Docker 镜像一样存储于仓库中）。如上图中Helm 的配置文件包括：

```
helm                                - chart包的目录名
├── templates                       - k8s配置模版目录
│   ├── deployment.yaml             - Deployment配置模板，定义如何部署Pod
│   ├── _helpers.tpl                - 以下划线开头的文件，helm视为公共库定义文
│   ├── ingress.yaml                - Ingress配置模板，定义外部如何访问Pod提供
│   ├── NOTES.txt                   - chart包的帮助信息文件，执行helm instal
│   └── service.yaml                - Service配置模板，配置访问Pod的服务抽象，
|── values.yaml                     - chart包的参数配置文件，各模版文件可以引用
├── Chart.yaml                      - chart定义，可以定义chart的名字，版本号等
```

```
├── charts                                  - 依赖的子包目录，里面可以包含多个依赖的ch
```

我们可以在 Chart.yaml 中定义每个项目的 chart 名称（类似安装包名），如

```yaml
apiVersion: v2
name: your-chart-name
description: A Helm chart for Kubernetes

type: application
version: 1.0.0
appVersion: 1.16.0
```

在 values.yaml 中定义模板文件中需要用到的变量，如

```yaml
#部署Pod的副本数，即运行多少个容器
replicaCount: 1
#容器镜像配置
image:
  repository: registry.cn-hangzhou.aliyuncs.com/demo/demo
  pullPolicy: Always
  # Overrides the image tag whose default is the chart version.
  tag: "dev"
#镜像仓库访问凭证
imagePullSecrets:
  - name: aliyun-registry-secret
#覆盖启动容器名称
nameOverride: ""
fullnameOverride: ""
#容器的端口暴露及环境变量配置
container:
  port: 8000
  env: []
#ServiceAccount，默认不创建
serviceAccount:
  # Specifies whether a service account should be created
  create: false
  # Annotations to add to the service account
```

```
  annotations: {}
  name: ""

podAnnotations: {}

podSecurityContext: {}
  # fsGroup: 2000

securityContext: {}
  # capabilities:
  #   drop:
  #   - ALL
  # readOnlyRootFilesystem: true
  # runAsNonRoot: true
  # runAsUser: 1000
#使用NodePort的service，默认为ClusterIp
service:
  type: NodePort
  port: 8000
#外部访问Ingress配置，需要配置hosts部分
ingress:
  enabled: true
  annotations: {}
    # kubernetes.io/ingress.class: nginx
    # kubernetes.io/tls-acme: "true"
  hosts:
    - host: demo.com
      paths: ["/demo"]
  tls: []
  #  - secretName: chart-example-tls
  #    hosts:
  #      - chart-example.local

  #.... 省略了其它默认参数配置
```

这里在默认生成的基础上添加了 container 部分，可以在这里指定容器的端口号而不用去改模板文件（让模板文件在各个项目通用，通常不需要做更改），同时添加env的配置，可以在helm部署时往容器里传入环境变量。将Service type从默认的ClusterIp改为了NodePort。部署同类型的不同

项目时，只需要根据项目情况配置Chart.yaml与values.yaml两个文件的少量配置项，templates目录下的模板文件可直接复用。

部署时需要在K8s环境中从 Docker 镜像仓库拉取镜像，因此需要在K8s中创建镜像仓库访问凭证（imagePullSecrets）

```
# 登录Docker Registry生成/root/.docker/config.json文件
sudo docker login --username=your-username registry.cn-shenzhen.aliyuncs.com
# 创建 namespace develop（我这里是根据项目的环境分支名称建立namespace）
kubectl create namespace develop
# 在 namespace develop中创建一个secret
kubectl create secret generic aliyun-registry-secret
--from-file=.dockerconfigjson=/root/.docker/config.json
--type=kubernetes.io/dockerconfigjson --namespace=develop
```

Jenkinsfile

Jenkinsfile 是 Jenkins pipeline 配置文件，遵循 Groovy 语法，对于 Spring Boot 项目的构建部署， 编写 Jenkinsfile 脚本文件如下，

```
image_tag = "default"  //定一个全局变量，存储Docker镜像的tag（版本）
pipeline {
    agent any
    environment {
        GIT_REPO = "${env.gitlabSourceRepoName}"  //从Jenkins Gitlab插件中获取G
        GIT_BRANCH = "${env.gitlabTargetBranch}"  //项目的分支
        GIT_TAG = sh(returnStdout: true,script: 'git describe --tags --always'
        DOCKER_REGISTER_CREDS = credentials('aliyun-docker-repo-creds') //dock
        KUBE_CONFIG_LOCAL = credentials('local-k8s-kube-config')  //开发测试环境
        KUBE_CONFIG_PROD = "" //credentials('prod-k8s-kube-config') //生产环境的

        DOCKER_REGISTRY = "registry.cn-hangzhou.aliyuncs.com" //Docker仓库地址
        DOCKER_NAMESPACE = "your-namespace"  //命名空间
        DOCKER_IMAGE = "${DOCKER_REGISTRY}/${DOCKER_NAMESPACE}/${GIT_REPO}" //

        INGRESS_HOST_DEV = "dev.your-site.com"    //开发环境的域名
        INGRESS_HOST_TEST = "test.your-site.com"  //测试环境的域名
        INGRESS_HOST_PROD = "prod.your-site.com"  //生产环境的域名
```

```
    }
    parameters {
        string(name: 'ingress_path', defaultValue: '/your-path', description:
        string(name: 'replica_count', defaultValue: '1', description: '容器副本
    }

    stages {
        stage('Code Analyze') {
            agent any
            steps {
               echo "1. 代码静态检查"
            }
        }
        stage('Maven Build') {
            agent {
                docker {
                    image 'maven:3-jdk-8-alpine'
                    args '-v $HOME/.m2:/root/.m2'
                }
            }
            steps {
                echo "2. 代码编译打包"
                sh 'mvn clean package -Dfile.encoding=UTF-8 -DskipTests=true'
            }
        }
        stage('Docker Build') {
            agent any
            steps {
                echo "3. 构建Docker镜像"
                echo "镜像地址：${DOCKER_IMAGE}"
                //登录Docker仓库
                sh "sudo docker login -u ${DOCKER_REGISTER_CREDS_USR} -p ${DOC
                script {
                    def profile = "dev"
                    if (env.gitlabTargetBranch == "develop") {
                        image_tag = "dev." + env.GIT_TAG
                    } else if (env.gitlabTargetBranch == "pre-release") {
                        image_tag = "test." + env.GIT_TAG
                        profile = "test"
                    } else if (env.gitlabTargetBranch == "master"){
```

```
                        // master分支则直接使用Tag
                        image_tag = env.GIT_TAG
                        profile = "prod"
                    }
                    //通过--build-arg将profile进行设置，以区分不同环境进行镜像构建
                    sh "docker build  --build-arg profile=${profile} -t ${DOCK
                    sh "sudo docker push ${DOCKER_IMAGE}:${image_tag}"
                    sh "docker rmi ${DOCKER_IMAGE}:${image_tag}"
                }
            }
        }
        stage('Helm Deploy') {
            agent {
                docker {
                    image 'lwolf/helm-kubectl-docker'
                    args '-u root:root'
                }
            }
            steps {
                echo "4. 部署到K8s"
                sh "mkdir -p /root/.kube"
                script {
                    def kube_config = env.KUBE_CONFIG_LOCAL
                    def ingress_host = env.INGRESS_HOST_DEV
                    if (env.gitlabTargetBranch == "pre-release") {
                        ingress_host = env.INGRESS_HOST_TEST
                    } else if (env.gitlabTargetBranch == "master"){
                        ingress_host = env.INGRESS_HOST_PROD
                        kube_config = env.KUBE_CONFIG_PROD
                    }
                    sh "echo ${kube_config} | base64 -d > /root/.kube/config"
                    //根据不同环境将服务部署到不同的namespace下，这里使用分支名称
                    sh "helm upgrade -i --namespace=${env.gitlabTargetBranch}
                }
            }
        }
    }
}
```

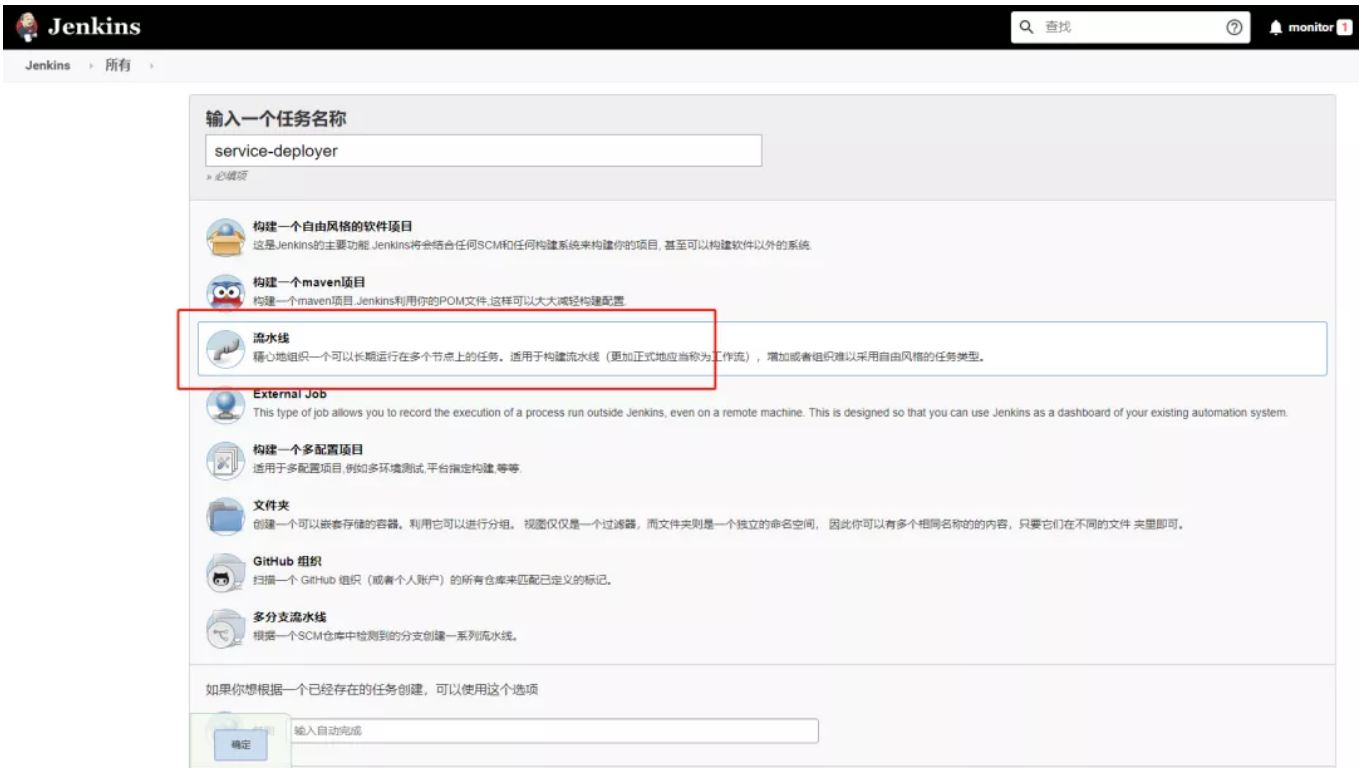
Jenkinsfile定义了整个自动化构建部署的流程：

1. Code Analyze，可以使用 SonarQube 之类的静态代码分析工具完成代码检查，这里先忽略
2. Maven Build，启动一个 Maven 的 Docker 容器来完成项目的 maven 构建打包，挂载 maven 本地仓库目录到宿主机，避免每次都需要重新下载依赖包
3. Docker Build，构建 Docker 镜像，并推送到镜像仓库，不同环境的镜像通过tag区分，开发环境使用 dev.commitId 的形式，如 dev.88f5822，测试环境使用 test.commitId，生产环境可以将 webhook 事件设置为 tag push event，直接使用 tag名称
4. Helm Deploy，使用helm完成新项目的部署，或已有项目的升级，不同环境使用不同的参数配置，如访问域名，K8s 集群的访问凭证kube_config等

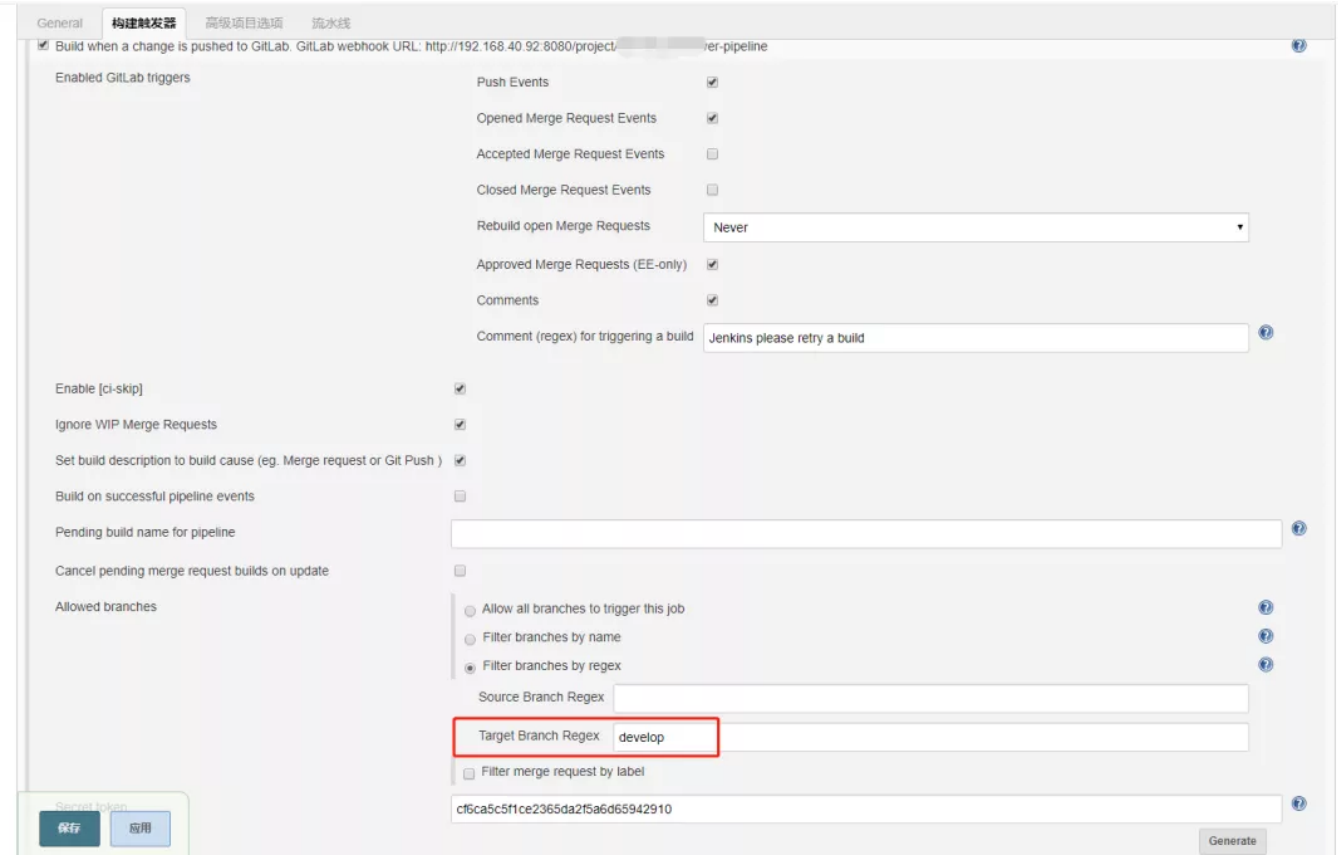
Jenkins 配置

Jenkins 任务配置

在 Jenkins 中创建一个 pipeline 的任务，如图

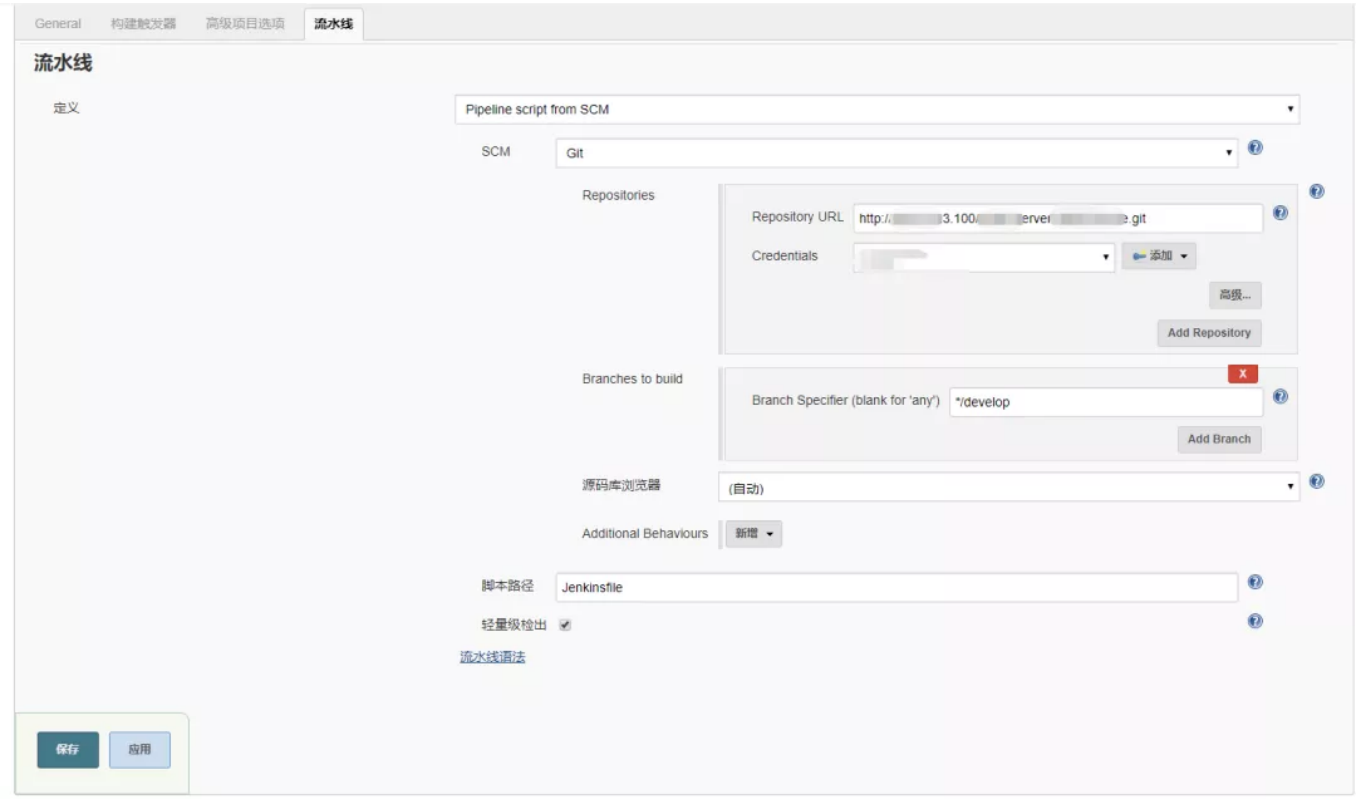


配置构建触发器，将目标分支设置为 develop 分支，生成一个 token，如图



记下这里的“GitLab webhook URL”及token值，在Gitlab配置中使用。

配置流水线，选择“Pipeline script from SCM”从项目源码中获取pipeline脚本文件，配置项目Git地址，拉取源码凭证等，如图



保存即完成了项目开发环境的Jenkins配置。测试环境只需将对应的分支修改为pre-release 即可

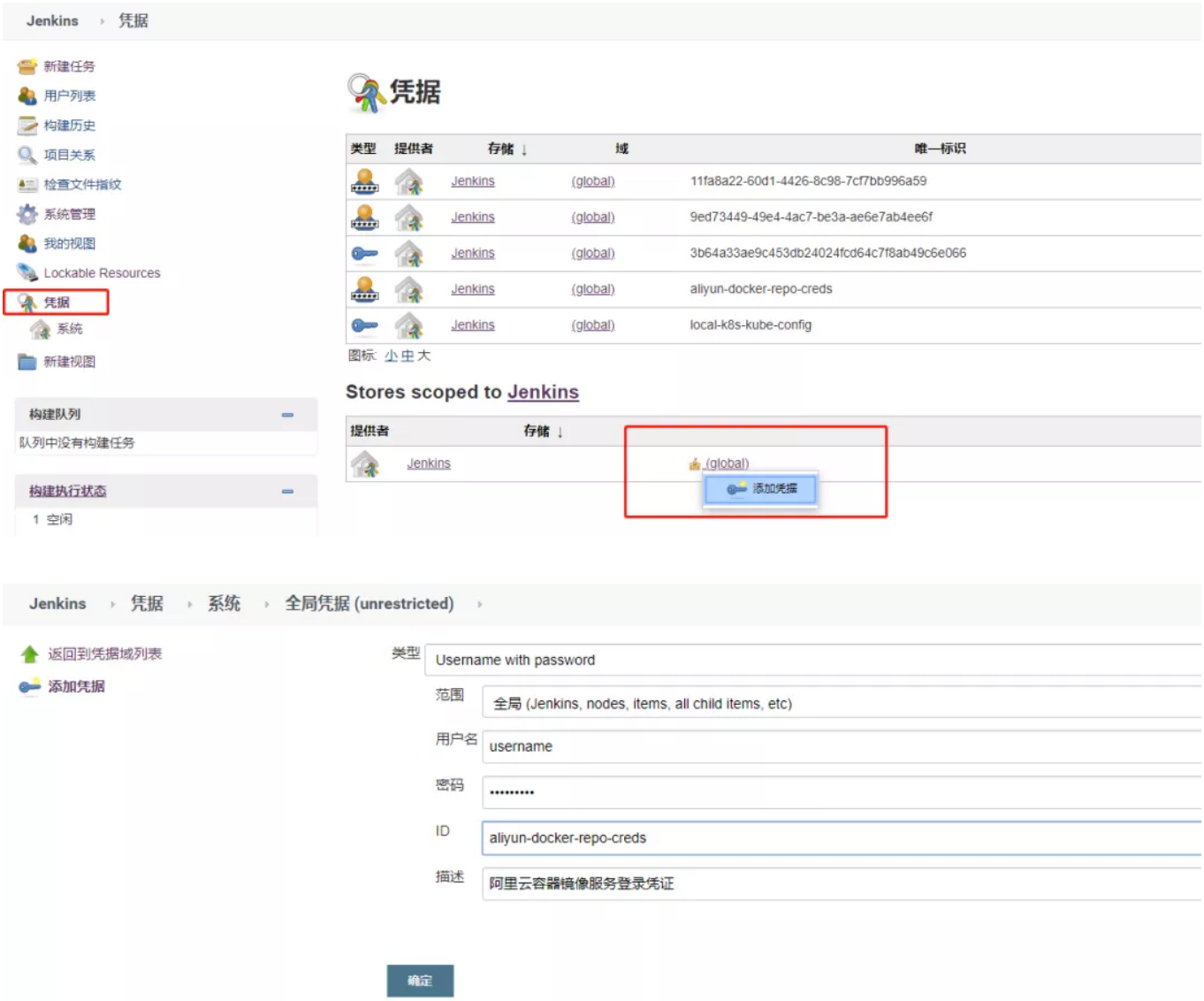
Jenkins 凭据配置

在 Jenkinsfile 文件中，我们使用到了两个访问凭证——Docker Registry凭证与本地K8s的kube凭证，

```
DOCKER_REGISTER_CREDS = credentials('aliyun-docker-repo-creds') //docker regist
KUBE_CONFIG_LOCAL = credentials('local-k8s-kube-config')  //开发测试环境的kube凭
```

这两个凭证需要在 Jenkins 中创建。

添加 Docker Registry 登录凭证，在 Jenkins 凭据页面，添加一个用户名密码类型的凭据，如图



添加 K8s 集群的访问凭证，在 master 节点上将 /root/.kube/config 文件内容进行 base64 编码，

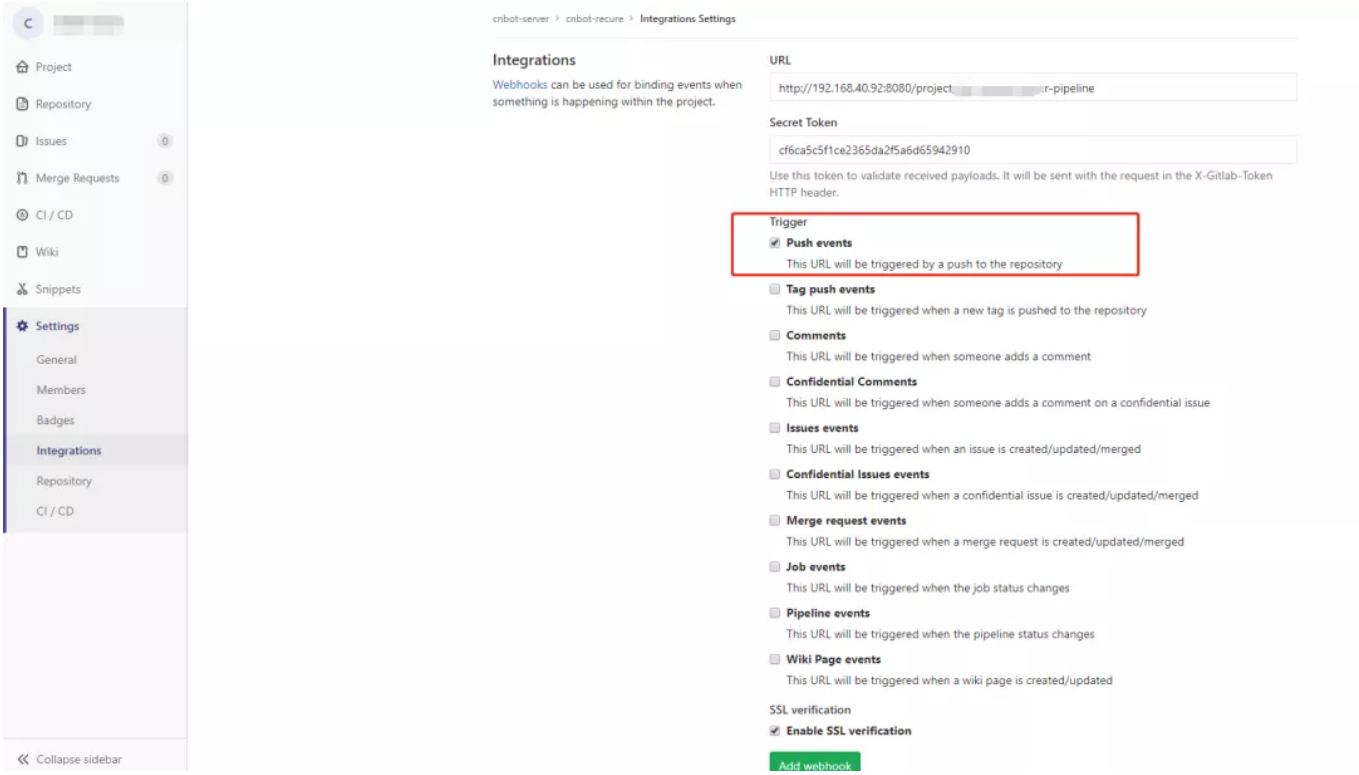
```
base64 /root/.kube/config > kube-config-base64.txt
cat kube-config-base64.txt
```

使用编码后的内容在 Jenkins 中创建一个 Secret text 类型的凭据，如图

在 Secret 文本框中输入 base64 编码后的内容。

Gitlab 配置

在 Gitlab 项目的 Settings - Integrations 页面配置一个 webhook，在 URL 与 Secret Token 中填入前面 Jenkins 触发器部分的“GitLab webhook URL”及token值，选中“Push events”作为触发事件，如图



开发、测试环境选择“Push events”则在开发人员push代码，或merge代码到develop，pre-release分支时，就会触发开发或测试环境的Jenkins pipeline任务完成自动化构建；生产环境选择“Tag push events”，在往master分支push tag时触发自动化构建。如图为pipeline构建视图

总结

本文介绍使用 Gitlab+Jenkins Pipeline+Docker+Kubernetes+Helm 来实现 Spring Boot项目的自动化部署，只要稍加修改即可应用于其它基于Spring Boot的项目（具体修改的地方在源码的Readme 文件中说明）。

来源：https://segmentfault.com/a/1190000022637144