



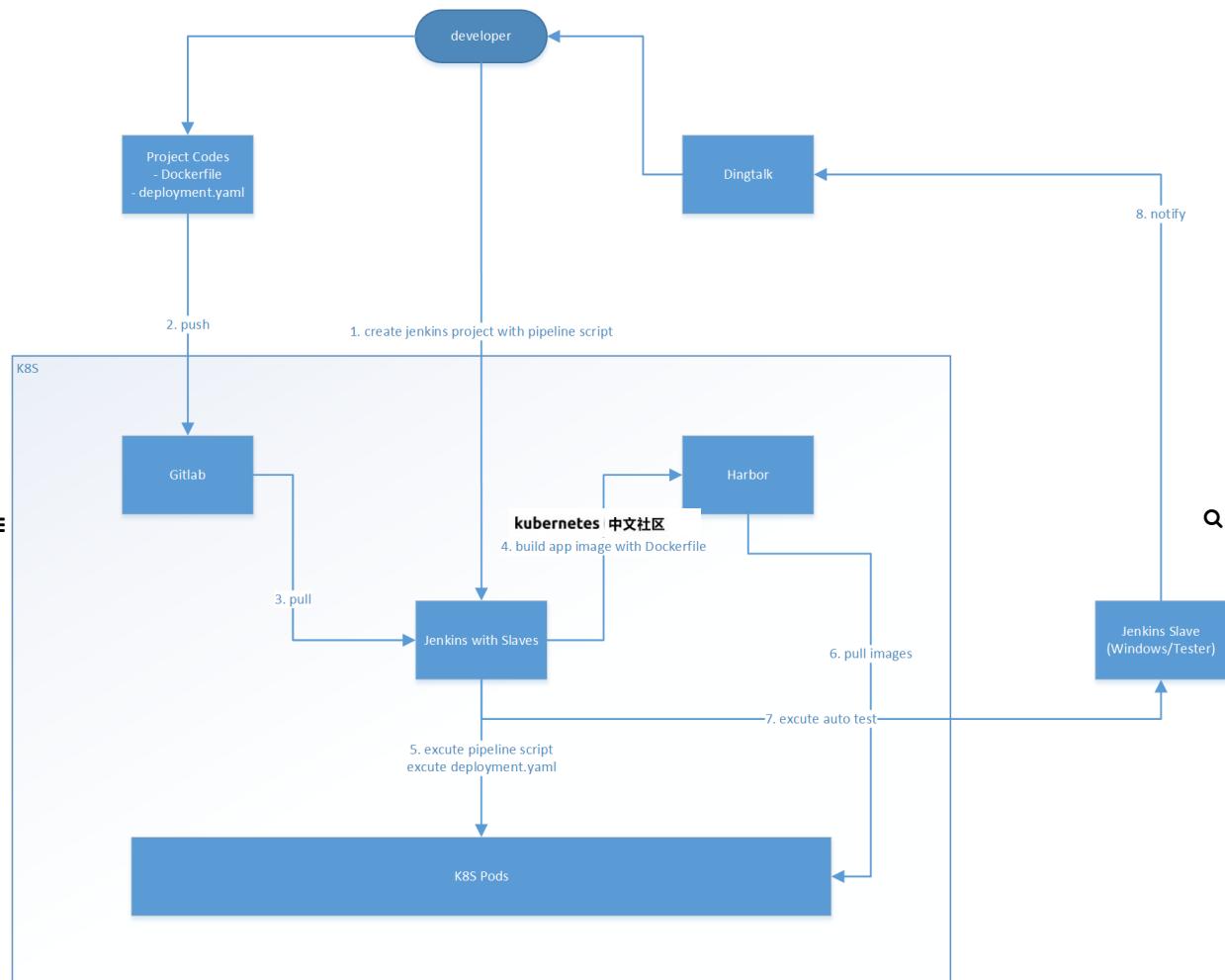
阿里云Kubernetes实战3-DevOps (<https://www.kubernetes.org.cn/4667.html>)

2018-10-11 16:27 WuRang (<https://www.kubernetes.org.cn/author/sonicrang>) 分类: Kubernetes实践分享/开发实战 (<https://www.kubernetes.org.cn/practice>) / Kubernetes教程/入门教程 (<https://www.kubernetes.org.cn/course>)
阅读(5225) 评论(1)

前言:

在上一篇文章中，我们已经在K8S集群部署了Jenkins、Harbor和EFK。作为本系列最后一篇文章，将通过实际案例串联所有的基础软件服务，基于K8S做DevOps。

整体的业务流程如下图所示：



一、一机多Jenkins Slave

由于业务需要，我们的自动化测试需要基于windows做web功能测试，每一个测试任务独占一个windows用户桌面，所以我们首先要给Jenkins配置几个Windows的Slave Node.在我之前的post《持续集成CI实施指南三-jenkins集成测试》(http://wurang.net/jenkins03_test/)中详细讲解了给Jenkins添加Node的方法步骤。本篇无需重复，但这里主要讲的是，如何在一台Windows服务器上搭建多个Jenkins Node，供多用户使用。

- 在目标机上建立多个用户，如下图所示：



- 用Administrator用户安装JDK
- 在Jenkins的节点管理建立三个Node，分别为WinTester01、WinTester02、WinTester03，配置如下

Node Configuration: WinTester01

Name: WinTester01
Description: (empty)
of executors: 1
Remote root directory: c:\jenkins\slave1
Labels: windows test
Usage: Only build jobs with label expressions matching this node
Launch method: Launch agent via Java Web Start
Availability: Keep this agent online as much as possible

Node Properties:

Environment variables

Name:	LANG
Value:	zh_CN.UTF-8

Tool Locations

Name:	(Git) Default	设置Git的环境变量
Home:	C:\Program Files\Git\bin\git.exe	

- 在目标机的Administrator，用IE打开Jenkins并进入节点管理，在WinTester01、WinTester02、WinTester03中分别点击“Launch”启动Slave

Connect agent to Jenkins one of these ways:

-  **Launch** Launch agent from browser
- Run from agent command line:

```
java -jar agent.jar -jnlpUrl https://ci.heygears.com/computer/WinTester01/
```

- 确认启动成功后，点击“File”下的“Install as service”

The screenshot shows two windows side-by-side. The left window is a Jenkins configuration screen with a red box highlighting the 'Install as a service' button under the 'File' menu. The right window is the Windows Services Manager showing a list of services. One service, 'Jenkins agent (jenkinsslave-c_jenkins_slave1)', is selected and highlighted with a blue selection bar. The list includes other services like Hyper-V PowerShell Direct Service, Hyper-V Time Synchronization Service, and various Microsoft system services.

- 三个Slave都启动后，可以在服务管理器看到

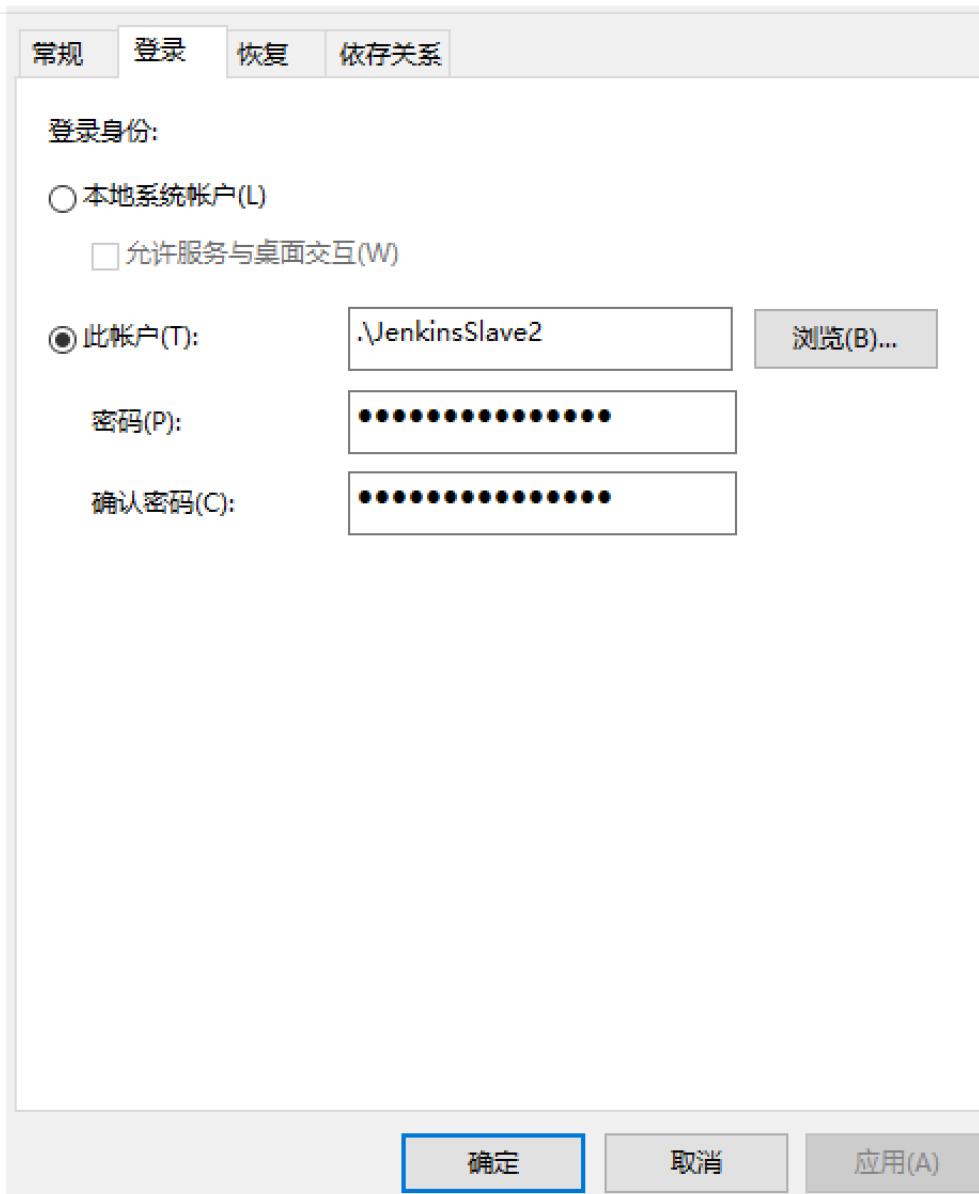
This screenshot shows the Windows Services Manager with three Jenkins agent services listed:

名称	描述
Jenkins agent (jenkinsslave-c_jenkins_slave1)	This service runs an agent for Jenkins automation server.
Jenkins agent (jenkinsslave-c_jenkins_slave2)	
Jenkins agent (jenkinsslave-c_jenkins_slave3)	

- 除了Jenkins Slave1无需配置，Slave2和Slave3都需要右键进入属性，修改登录用户分别为JenkinsSlave2和JenkinsSlave3

Jenkins agent (jenkinsslave-c_jenkins_slave2) 的属性(本地计算机)

X



通过上面的配置，可以在一台目标机部署三个用户对应三个Jenkins Slave以满足我们的业务需求。

二、二次开发Jenkins 钉钉通知插件

在整个DevOps的业务流程图上，我们想使用钉钉作为通知方式，相比邮件而言，实时性和扩展性都很高。在2018年4月，Jenkins的钉钉通知插件有两款，分别是Dingding JSON Pusher (<https://plugins.jenkins.io/dingding-json-pusher>)和Dingding notification plugin (<https://github.com/jenkinsci/dingding-notifications-plugin>)，前者长期未更新，已经不能使用，后者可以在非Pipeline模式下使用，对于Pipeline则有一些问题。虽然目前，Dingding notification plugin已经更新到1.9版本并支持了Pipeline，但在当时，我们不得不在1.4版本的基础上做二次开发。

整体开发经过参考《Jenkins项目实战之-钉钉提醒插件二次开发举例》 (<https://blog.csdn.net/u011541946/article/details/78634786>)，总体来说还是比较简单：

- 修改“src/main/java/com/ztbsuper/dingtalk/DingTalkNotifier.java”，钉钉的消息API类型有文本、link、markdown、card等，我们这里把通知接口改成文本类型

```

public class DingTalkNotifier extends Notifier implements SimpleBuildStep {

    private String accessToken;
    private String message;
    private String imageUrl;
    private String messageUrl;

    @DataBoundConstructor
    public DingTalkNotifier(String accessToken, String message, String imageUrl, String messageUrl) {
        this.accessToken = accessToken; //钉钉的accesstoken
        this.message = message; //消息主体
        this.imageUrl = imageUrl; //缩略图
        this.messageUrl = messageUrl; //消息的链接来源,一般是jenkins的build url
    }

    public String getAccessToken() {
        return accessToken;
    }
    public String getMessage() {
        return message;
    }
    public String getImageUrl() {
        return imageUrl;
    }
    public String getMessageUrl() {
        return messageUrl;
    }

    @Override
    public void perform(@Nonnull Run<?, ?> run, @Nonnull FilePath filePath, @Nonnull Launcher launcher, @Nonnull TaskListener taskListener) throws Inter
        String buildInfo = run.getFullName();
        if (!StringUtil.isBlank(message)) {
            sendMessage(LinkMessage.builder()
                .title(buildInfo)
                .picUrl(imageUrl)
                .text(message)
                .messageUrl(messageUrl)
                .build());
        }
    }

    private void sendMessage(DingMessage message) {
        DingTalkClient dingTalkClient = DingTalkClient.getInstance();
        try {
            dingTalkClient.sendMessage(accessToken, message);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Override
    public BuildStepMonitor getRequiredMonitorService() {
        return BuildStepMonitor.NONE;
    }

    @Symbol("dingTalk")
    @Extension
    public static final class DescriptorImpl extends BuildStepDescriptor<Publisher> {

        @Override
        public boolean isApplicable(Class<? extends AbstractProject> aClass) {
            return true;
        }

        @Nonnull
        @Override
        public String getDisplayName() {
            return Messages.DingTalkNotifier_DescriptorImpl_DisplayName();
        }
    }
}

```

- 用maven打包

maven需要安装java环境,为了方便,我直接run一个maven的docker image,编译完成后把hpi文件send出来

- 在jenkins的插件管理页面上传hpi文件

Updates Available Installed Advanced

HTTP Proxy Configuration

Server:

Port:

User name: admin

Password:

No Proxy Host:

Upload Plugin

You can upload a .hpi file to install a plugin from outside the central plugin repository.

File: 未选择任何文件

- 在钉钉群中开启自定义机器人

群设置

X

[查看更多](#)

群机器人

>

所在分组

群聊 >

第三方加密

未开通 >

我在本群的昵称

未设置 ⚡

置顶聊天

消息免打扰

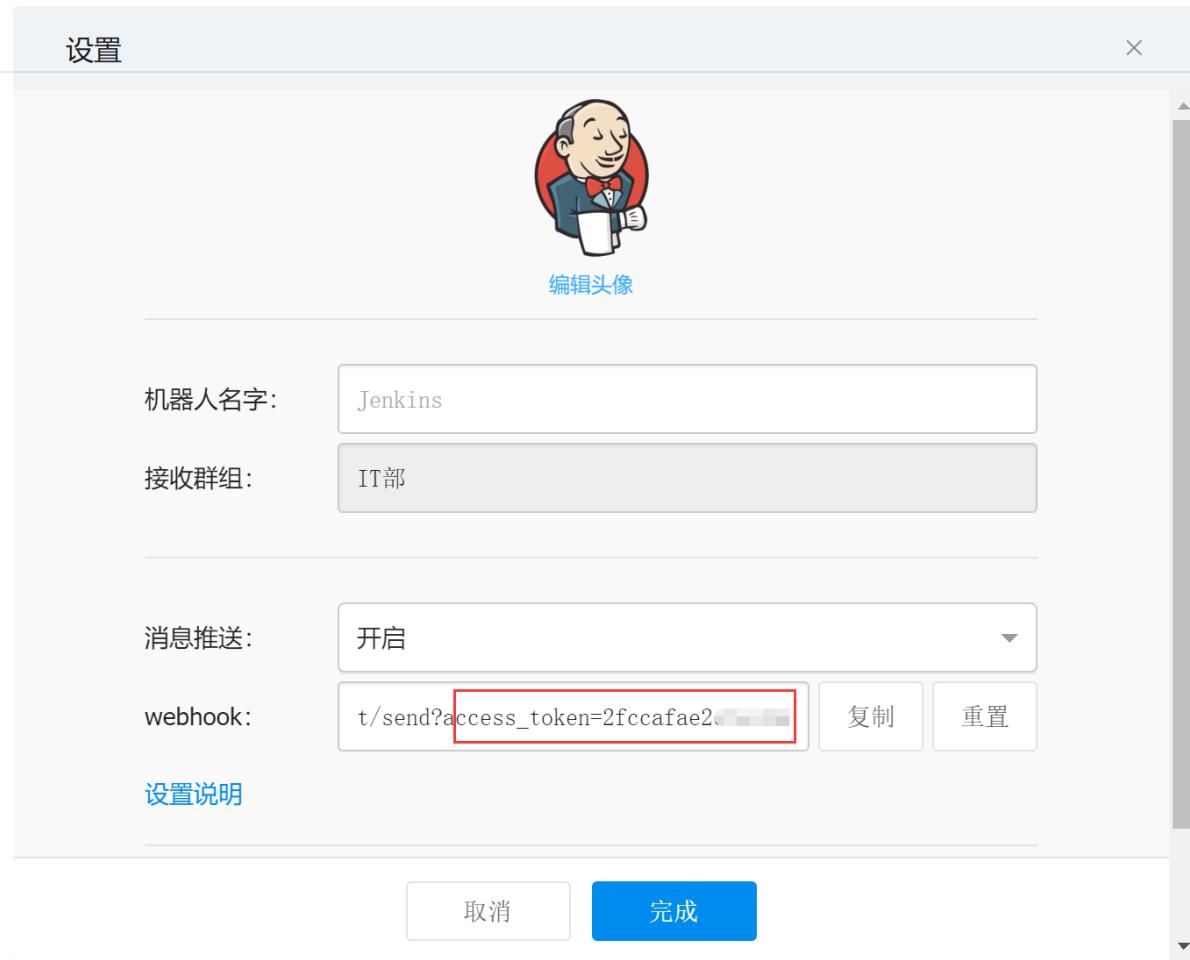
群管理

>

清空聊天记录

退出群聊

- 找到access token



- 在jenkins pipeline中可以使用以下命令发送信息到钉钉群

```
dingTalk accessToken:"2fccafaexxxx",message:"信息",imageUrl:"图片地址",messageUrl:"消息链接"
```

三、 DevOps解决方案

针对每一个软件项目增加部署目录，目录结构如下：

- _deploy
 - master
 - deployment.yaml
 - Dockerfile
 - other files
 - test
 - deployment.yaml
 - Dockerfile
 - other files

master和test文件夹用于区分测试环境与生产环境的部署配置

Dockerfile和其他 files 用于生成应用或服务的镜像

如前端vue和nodejs项目的Dockerfile：

```
# 前端项目运行环境的Image，从Harbor获取
FROM xxx/xxx/frontend:1.0.0
RUN mkdir -p /workspace/build && mkdir -p /workspace/run
COPY . /workspace/build
# 编译，生成执行文件，并删除源文件
RUN cd /workspace/build/frontend && \
  cnpm install && \
  npm run test && \
  cp -r /workspace/build/app/* /workspace/run && \
  rm -rf /workspace/build && \
  cd /workspace/run && \
  cnpm install
# 运行项目，用npm run test或run prod区分测试和生产环境
CMD cd /workspace/run && npm run test
```

又如dotnet core项目的Dockerfile：

```
# dotnet项目编译环境的Image，从Harbor获取
FROM xxx/xxx/aspnetcore-build:2 AS builder
WORKDIR /app
COPY . .
# 编译
RUN cd /app/xxx
RUN pwd && ls -al && dotnet restore
RUN dotnet publish -c Release -o publish

# dotnet项目运行环境的Image，从Harbor获取
FROM xxx/xxx/aspnetcore:2
WORKDIR /publish
COPY --from=builder /app/xxx/publish .
# 重命名配置文件，中缀test、prod用于区分测试环境和生产环境
RUN mv appsettings.test.json appsettings.json
# 运行
ENTRYPOINT ["dotnet", "xxx.dll"]
```

deployment.yaml用于执行应用或服务在k8s上的部署

由于deployment有很多配置项可以抽离成公共配置，所以deployment的配置有很多占位变量，占位变量用两个#中间加变量名表示，如下所示：

```

apiVersion: v1
kind: Namespace
metadata:
  name: #namespace#
  labels:
    name: #namespace#
---
apiVersion: v1
data:
  .dockerconfigjson: xxxxxxxxxxxxxxxxxxxxxxxx
kind: Secret
metadata:
  name: regcred
  namespace: #namespace#
type: kubernetes.io/dockerconfigjson
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: #app#-deploy
  namespace: #namespace#
  labels:
    app: #app#-deploy
spec:
  replicas: #replicas#
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: #app#
    spec:
      containers:
        - image: #image#
          name: #app#
          ports:
            - containerPort: #port#
              name: #app#
          securityContext:
            privileged: #privileged#
          volumeMounts:
            - name: log-volume
              mountPath: #log#
        - image: #filebeatImage#
          name: filebeat
          args:
            - "-c", "/etc/filebeat.yml"
          securityContext:
            runAsUser: 0
          volumeMounts:
            - name: config
              mountPath: /etc/filebeat.yml
              readOnly: true
              subPath: filebeat.yml
            - name: log-volume
              mountPath: /var/log/container/
      volumes:
        - name: config
          configMap:
            defaultMode: 0600
            name: filebeat-config
        - name: log-volume
          emptyDir: {}
      imagePullSecrets:
        - name: regcred
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: filebeat-config
  namespace: #namespace#
  labels:
    app: filebeat
data:
  filebeat.yml: |- 
    filebeat.inputs:
      - type: log
        enabled: true
        paths:
          - /var/log/container/*.log
    output.elasticsearch:
      hosts: ["#es#"]
      tags: ["#namespace#-#app#"]
---
apiVersion: v1
kind: Service
metadata:
  name: #app#-service
  namespace: #namespace#
  labels:
    app: #app#-service
spec:
  ports:
    - port: 80
      targetPort: #port#
  selector:

```

```

app: #app#
---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: #app#-ingress
  namespace: #namespace#
  annotations:
    nginx.ingress.kubernetes.io/proxy-body-size: "0"
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: #host#
    http:
      paths:
      - path: #urlPath#
        backend:
          serviceName: #app#-service
          servicePort: 80

```

其中几个关键变量的解释如下：

- dockerconfigjson：因为所有的镜像需要从Harbor获取，而Harbor的镜像如果设置为私有权限，就需要提供身份验证，这里的dockerconfigjson就是Harbor的身份信息。生成dockerconfigjson的方法如下：
 - 进入K8S任何一个节点，删除“~/.docke/config.json”文件
 - 使用命令“`docker login harbor地址`”登录harbor
 - 通过命令“`cat ~/.docke/config.json`”可以看到harbor的身份验证信息
 - 使用命令“`cat /root/.docke/config.json | base64 -w 0`”对信息编码，将生成后的编码填写到deployment.yaml的dockerconfigjson节点即可
- namespace：同一个项目的不同k8s组件应置于同一个namespace，所以namespace可统一配置，在我们的项目实践中，生产环境的namespace为“项目名”，测试环境的namespace为“项目名-test”
- app：应用或服务名称
- image：应用或服务的镜像地址
- replicas：副本数量
- port：应用或服务的Pod开放端口
- log：应用或服务的日志路径，在本系列的第二篇文章中，提到我们的日志方案是给每个应用或服务配一个filebeat，放在同一Pod中，这里只需告知应用或服务的日志的绝对路径，filebeat就能将日志传递到ES中，日志的tag命名方式为“namespace-app”
- host：在本系列的第一篇文章中，讲了使用nginx ingress做服务暴露与负载。这里的host就是给nginx ingress设置的域名，端口默认都是80，如果需要https，则在外层使用阿里云SLB转发
- urlPath：很多情况下，如微服务，需要通过相同的域名，不同的一级目录将请求分发到不同的后台，在nginx中，就是location的配置与反向代理，比如host的配置是确定了域名aaa.bbb.com，而urlPath的配置是确定aaa.bbb.com/user/getuser将会被转发到用户服务podIP:podPort/getuser中

以上所有的占位变量都是在Pipeline Script中赋值，关于Jenkins Pipeline的相关内容介绍这里不再多讲，还是去看官方文档 (<https://jenkins.io/doc/book/pipeline/>)靠谱。我们这里将k8s的部署文件deployment.yaml与Jenkinsfile结合，即可做到一个deployment.yaml能适配所有项目，一个Pipeline Script模板能适配所有项目，针对不同的项目，只需在Pipeline Script中给占位变量赋值，大大降低了配置复杂度。下面是一个项目的Jenkins配置示例：

General

Description:

[Plain text] [Preview]

Discard old builds

Do not allow concurrent builds

Do not allow the pipeline to resume if the master restarts

GitLab Connection: [dropdown]

Pipeline speed/durability override

Preserve stashes from completed builds

This project is parameterized

Throttle builds

Build Triggers

Build after other projects are built

Build periodically

Build when a change is pushed to GitLab. GitLab webhook URL: https://gitlab.com/api/v4/projects/1234567/hooks/1234567

Enabled GitLab triggers: Push Events

Opened Merge Request Events

Accepted Merge Request Events

Closed Merge Request Events

Rebuild open Merge Requests: Never

Approved Merge Requests (EE-only)

Comments

Comment (regex) for triggering a build: Jenkins please retry a build

Poll SCM

Disable this project

Quiet period

Trigger builds remotely (e.g., from scripts)

Advanced Project Options

Advanced...

Pipeline

Definition: Pipeline script

Script:

```

10 // --- MOUNTS ---
11 // nameSpace: dentlab-test_dentlab
12 namespace = "dentlab-test"
13 // hostname
14 host = "test.api.dent-lab.com"
15 // appname
16 app = "api"
17 // port

```

对于一个项目，我们只需配置Trigger和Pipeline，上图“Do not allow concurrent builds”也是通过Pipeline的配置生成的。Pipeline Script示例如下：

```

pipeline {
    // 指定项目在label为jnlp-agent的节点上构建，也就是Jenkins Slave in Pod
    agent { label 'jnlp-agent' }
    // 对应Do not allow concurrent builds
    options {
        disableConcurrentBuilds()
    }
    environment {
        // ----- 以下内容，每个项目可能均有不同，按需修改 -----
        //author: 用于钉钉通知
        author="张三"
        // branch: 分支，一般是test、master，对应git从哪个分支拉取代码，也对应究竟执行_deploy文件夹下的test配置还是master配置
        branch = "test"
        // namespace: myproject-test, myproject, 命名空间一般是项目名称，测试环境加test
        namespace = "myproject-test"
        // hostname: 对应deployment中的host
        host = "test.aaa.bbb.com"
        // appname: 对应deployment中的app
        app = "myserver"
        // port: 对应deployment中的port
        port= "80"
        // replicas: 对应deployment中的replicas
        replicas = 2
        //git repo path: git的地址
        git="git@git.aaa.bbb.com/xxx.git"
        //log: 对应deployment中的log
        log="publish/logs"
        // ----- 以下内容，一般所有的项目都一样，不经常修改 -----
        // harbor inner address
        repoHost = "192.168.0.1:23280"
        // harbor的账号密码信息，在jenkins中配置用户名/密码形式的认证信息，命名成harbor即可
        harborCreds = credentials('harbor')
        // filebeat的镜像地址
        filebeatImage="${repoHost}/common/filebeat:6.3.1"
        // es的内网访问地址
        es="elasticsearch-logging.kube-system:9200"
    }
    // ----- 以下内容无需修改 -----
    stages {
        // 开始构建前清空工作目录
        stage ("CleanWS"){
            steps {
                script {
                    try{
                        deleteDir()
                    }catch(err){
                        echo "${err}"
                        sh 'exit 1'
                    }
                }
            }
        }
        // 拉取
        stage ("CheckOut"){
            steps {
                script {
                    try{
                        checkout([$class: 'GitSCM', branches: [[name: "${branch}"]], doGenerateSubmoduleConfigurations: false, extensions: [], submoduleCfg: []])
                    }catch(err){
                        echo "${err}"
                        sh 'exit 1'
                    }
                }
            }
        }
        // 构建
        stage ("Build"){
            steps {
                script {
                    try{
                        // 登录 harbor
                        sh "docker login -u ${harborCreds_USR} -p ${harborCreds_PSW} ${repoHost}"
                        sh "date +%Y%m%d%H%M%S > timestamp"
                        // 镜像tag用时间戳代表
                        tag = readFile('timestamp').replace("\n", "").replace("\r", "")
                        repoPath = "${repoHost}/${namespace}/${app}:${tag}"
                        // 根据分支，进入_deploy下对应的不同文件夹，通过dockerfile打包镜像
                        sh "cp _deploy/${branch}/* ./"
                        sh "docker login -u ${harborCreds_USR} -p ${harborCreds_PSW} ${repoHost}"
                        sh "docker build -t ${repoPath} ."
                    }catch(err){
                        echo "${err}"
                        sh 'exit 1'
                    }
                }
            }
        }
        // 镜像推送到harbor
        stage ("Push"){
            steps {
                script {
                    try{
                        sh "docker push ${repoPath}"
                    }catch(err){
                        echo "${err}"
                        sh 'exit 1'
                    }
                }
            }
        }
    }
}

```

```

        }
    }
}

// 使用pipeline script中复制的变量替换deployment.yaml中的占位变量，执行deployment.yaml进行部署
stage ("Deploy"){
    steps {
        script {
            try{
                sh "sed -i 's|#namespace|${namespace}|g' deployment.yaml"
                sh "sed -i 's|#app|${app}|g' deployment.yaml"
                sh "sed -i 's|#image|${repoPath}|g' deployment.yaml"
                sh "sed -i 's|#port|${port}|g' deployment.yaml"
                sh "sed -i 's|#host|${host}|g' deployment.yaml"
                sh "sed -i 's|#replicas|${replicas}|g' deployment.yaml"
                sh "sed -i 's|#log|${log}|g' deployment.yaml"
                sh "sed -i 's|#filebeatImage|${filebeatImage}|g' deployment.yaml"
                sh "sed -i 's|#es|${es}|g' deployment.yaml"
                sh "sed -i 's|#redisImage|${redisImage}|g' deployment.yaml"
                sh "cat deployment.yaml"
                sh "kubectl apply -f deployment.yaml"
            }catch(err){
                echo "${err}"
                sh 'exit 1'
            }
        }
    }
}

post {
    // 使用钉钉插件进行通知
    always {
        script {
            def msg = "[${author}] 你把服务器搞挂了，老詹喊你回家改BUG！"
            def imageUrl = "https://www.iconsdn.com/icons/preview/red/x-mark-3-xxl-2.png"
            if (currentBuild.result=="SUCCESS"){
                imageUrl= "http://icons.iconarchive.com/icons/paomedia/small-n-flat/1024/sign-check-icon-2.png"
                msg = "[${author}] 发布成功，干得不错！"
            }
            dingTalk accessToken:"xxxx",message:"${msg}",imageUrl:"${imageUrl}",messageUrl:"${BUILD_URL}"
        }
    }
}
}
}

```

发布完成后，可以参考《持续集成CI实施指南三-jenkins集成测试》 (http://wurang.net/jenkins03_test/)，做持续测试，测试结果也可通过钉钉通知。最后我们利用自建的运维平台，监控阿里云ECS状态、K8S各组件状态、监控ES中的日志并做异常抓取和报警。形成一整套DevOps模式。

综上，对于每个项目，我们只需维护Dockerfile，并在Jenkins创建持续集成项目时，填写项目所需的参数变量。进阶情况下，也可定制性的修改deployment文件与pipeline script，满足不同的业务需要。至此，完结，撒花！

来源：http://wurang.net/alicloud_kubernetes_03/



关注微信公众号，加入社区



http://www.csdn.net/u/14122114/article/details/52447535?utm_source=share.php?

上一篇：阿里云Kubernetes实战2-搭建基础服务 (<https://www.kubernetes.org.cn/4666.html>) | 下一篇：在Kubernetes上搭建RabbitMQ Cluster (<https://www.kubernetes.org.cn/4679.html>)

[url=https%3A%2F%2Fwww.kubernetes.org.cn%2F4667.html&title=%E9%98%BF%E9%87%8C%e4%ba%91](https://www.kubernetes.org.cn/2F4667.html&title=%E9%98%BF%E9%87%8C%e4%ba%91)

标签： CI/CD (<https://www.kubernetes.org.cn/tags/cicd>) DevOps (<https://www.kubernetes.org.cn/tags/devops>) 阿里云 (<https://www.kubernetes.org.cn/tags/%e9%99%bf%e9%87%8c%e4%ba%91>)

相关推荐

- 石油巨头如何与Kubernetes, DevOps共舞？ (<https://www.kubernetes.org.cn/4906.html>)
- 阿里云Kubernetes实战2-搭建基础服务 (<https://www.kubernetes.org.cn/4666.html>)
- 阿里云Kubernetes实战1-集群搭建与服务暴露 (<https://www.kubernetes.org.cn/4630.html>)
- Google推出Kubernetes二进制授权，强制镜像经完整CI/CD才能部署 (<https://www.kubernetes.org.cn/4484.html>)
- 运行于kubernetes的php应用的devops方案 (<https://www.kubernetes.org.cn/4430.html>)
- 灵雀云DevOps平台C位出道背后的“神秘力量” (<https://www.kubernetes.org.cn/4328.html>)
- 容器厂商灵雀云，共同助力国家DevOps标准体系建设 (<https://www.kubernetes.org.cn/4217.html>)
- 客户案例 | 灵雀云容器PaaS平台助力知名股份制银行金融科技革新 (<https://www.kubernetes.org.cn/4160.html>)

评论 1



社区交流

8 1 5 7 2 .

提交评论

昵称	昵称 (必填)
邮箱	邮箱 (必填)
网址	网址



asdfasdf

f (<http://asdf>) 2个月前 (10-24)

#1