基金项目：否

作者简介：韩东（1991-），男，江苏省、盱眙人，硕士，江苏科技大学计算机科学与工程学院、研究方向为软件工程，容器虚拟化技术，DevOps。

**基于kubernetes容器集群环境的持续交付平台设计与实现**

韩东

（江苏科技大学 计算机科学与工程学院，江苏 镇江 212003）

**摘要：**传统软件交付模式偏向于手动，软件从源码到编译、打包、测试、交付生产经历不同的形态和系统环境。手动模式效率较低，且不同的系统环境存在差异容易引起意想不到的错误导致软件部署失败。Docker容器技术可以将应用及其依赖打包到一个可移植的镜像中，然后发布到任何运行容器技术的环境中，从而减少环境差异引起的软件交付失败。Kubernetes容器集群技术进一步提高容器技术的可用性，设计一套基于kubernetes的持续交付平台，能够实现软件交付的自动化图形化，提高软件交付效率。

**关键词：**容器集群；持续交付；Docker；Kubernetes；

1. **引言**

在云计算、大数据、人工智能飞速发展的背景下，出现了微服务、容器、DevOps等技术与概念。微服务是当今比较流行的软件架构，要求将一个整体的产品进行拆分，在功能上进一步解耦，从一个软件产品提供多个服务，拆分成各自提供不同服务的多个个体。以Docker为代表的容器技术从虚拟化技术发展而来，Docker将程序及其依赖打包成轻量级与可移植的容器中，便于部署与分发。相较于虚拟化技术，容器技术将操作系统划分不同的运行环境，占用资源更少。DevOps是一组过程、方法、系统的统称，用于促进开发，技术运营与质量保障部门之间的沟通、协作与整合。DevOps旨在让开发人员和运维人员更好的沟通合作，通过自动化的流程来使得整个软件过程更加便捷与可靠。

持续交付是DevOps中一个重要的方法，主要研究通过自动化的方法使得软件部署与交付变得更加便利。Docker镜像以其轻量级与可移植性更加有利于软件的部署与交付，将容器作为软件的运行环境更加有利于运维操作。Kubernetes是容器集群技术，用于自动部署，扩展和管理容器化的应用程序。面向Kubernetes环境的持续交付平台，需要深入分析Kubernetes系统结构，利用其提供的资源将组成应用程序的容器组合成逻辑单元。传统模式下，不同整个软件过程偏向于手动模式主要使用各自平台的命令行工具，交付过程低下，并且没有形成一定规范与约束的流程。建立一套统一的图形化的交付平台能够规范交付流程，减少交付失败的机率。

1. **持续交付平台设计**

持续交付平台旨在管理一个软件项目的完整的生命周期。从静态代码开始，到编译与打包，再到将程序包构建成docker镜像。Docker镜像如果只存在开发环境的机器中，则此镜像不便于共享和分发，还需要将镜像推送至镜像仓库中。最后将构建好的镜像按照软件架构的设计部署在容器集群中。整个过程中，软件项目从可编辑的源代码，到二进制程序包，再到容器镜像，整个过程中需要借助相关的平台和工具。例如，使用github作为代码版本公职工具，而docker镜像是要docker服务进行构建的。持续交付平台本身不是各项具体工作的实施者，而是交付流程的创建者，各平台的统一组织者，持续交平台通过是图形化界面的方式进行操作，以表单的方式配置各流程中的参数。

对于企业来说，首先要搭建一套基础设施服务。但受企业内部规定和限制，软件项目可能只在企业内部网络使用，并不想公开在互联网上，例如企业不希望自己的项目源代码托管在github上，不希望自己的镜像发布在Docker hub上，所以一些互联网的公共服务不能使用，需要企业自己在内部网络构建一套基础设施服务。

持续交付流程是整个交付平台的核心环节，基于各基础平台的持续交付平台的交付流程如图1所示：

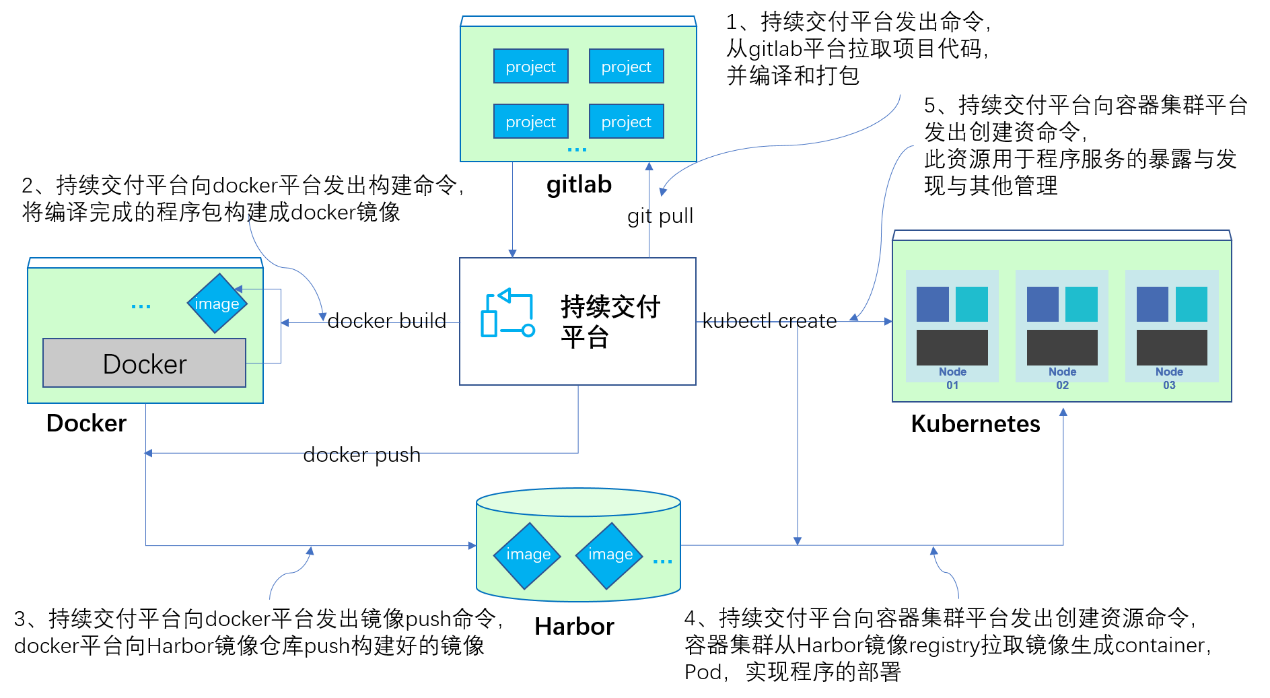


图1

1. 持续交付平台从代码托管平台gitlab上拉取指定项目的源码，并且编译和打包。
2. 持续交付平台根据项目的Dockerfile文件，向Docker平台发出构建镜像命令，docker平台会根据项目的Dockerfile进行镜像的构建。所以每个项目必须配置Dockerfile文件。
3. 镜像构建完成后，此镜像仅存在于Docker平台中，持续交付平台还需要驱动Docker平台将镜像推送至指定的镜像仓库中，等待部署与交付生产环境。
4. 持续交付平台根据项目的架构设计，创建kubernetes Deployment资源，指定相关的镜像和副本数，交付至容器集群中。
5. 如果没有其他操作，此时正在运行的软件服务只能在容器集群内部访问，持续交付平台还需要命令容器集群平台创建kubernetes Service资源，用于服务的发现。此时整个软件项目完成交付，并提供访问。
6. **持续交付平台实现**

实现持续交付平台的重点内容在如何在平台上操作各个基础服务平台完成构建打包、镜像构建与推送、容器集群中的各个资源的创建和管理。幸运的是，各个平台都提供了不同语言版本的客户端，并且开源。持续交付平台需要集成这些客户端，使用这些客户端提供的API实现平台化的操作。如何使用这些API实现编码是关键问题。

2.1、客户端选择

持续交付平台使用Java语言实现使用Maven进行项目构建，因此需要选择各个平台的Java版本的客户端。各平台的客户端Maven依赖详情如下1所示：

|  |  |  |  |
| --- | --- | --- | --- |
| 平台 | git | docker | kubernetes |
| 客户端依赖 | groupId:org.eclipse.jgit  artifactId:org.eclipse.jgit  version:5.5.0.201909110433-r | groupId:com.spotify artifactId:docker-client version:8.16.0 | groupId:io.fabric8 artifactId:kubernetes-api  version:3.0.12 |

表1

2.2、编程实现

在编码实现环节，省略了一般性通用性功能的编码实现如平台的注册/登录，而重点聚焦持续交付流程中的核心功能的编码实现。持续交付平台需要和各个基础平台进行交互，编码的重点就在于如何利用集成的各平台客户端提供的API进行交互的实现。

1. 将项目源码克隆至本地。持续交付平台与gitlab代码托管平台进行交互，需要传递代码仓库的url和仓库的用户名，密码以及保存至本地的目录。具体编码实现如下：

public static void cloneGitRepository(GitRepository gitRepository, String localPath) throws Exception {

//设置代码仓库的url, GitRepository封装了gitlab账户的url，用户名，密码

CloneCommand cc = Git.cloneRepository().setURI(gitRepository.getUrl());

//传入仓库认证：用户名，密码

cc.setCredentialsProvider(

new UsernamePasswordCredentialsProvider(gitRepository.getUsername(),

gitRepository.getPassword()));

//设置保存至本地的目录

cc.setDirectory(new File(localPath)).call();

}

1. 创建Docker镜像。将编译完成的程序包按照Dockerfile文件构建Docker镜像。需要在操作系统的环境变量中配置docker服务所在的地址，如：DOCKER\_HOST=tcp://192.168.18.150:2375。Docker客户端读取DOCKER\_HOST的值与Docker服务建立连接。具体编码实现如下：

public String buildImage(String dockerFileDir, String tag) throws Exception {  
 //读取DOCKER\_HOST的环境变量，创建docker客户端，并建立连接  
 final DockerClient dockerClient = DefaultDockerClient.fromEnv().build();  
 //读取Dockerfile所在的目录，创建Dockerfile对象  
 File dockerFile = new File(dockerFileDir);  
 Path dockerFilePath = dockerFile.toPath();  
 //docker客户端发起build命令创建docker镜像，并给镜像打上tag  
 DockerClient.BuildParam buildParam = new DockerClient.BuildParam("t", tag);  
 //客户端向docker服务平台发出创建镜像的请求，返回ImageId  
 return dockerClient.build(dockerFilePath, buildParam);  
}

1. 推送镜像至Harbor镜像仓库。持续交付平台与Docker平台交互，将docker镜像推送至harbor镜像仓库中。一般来说在企业内部使用的是私有仓库，需要提供仓库的用户名和密码。编码实现如下：

public void pushImage(String username, String pw, String image) throws Exception {  
 final DockerClient dockerClient = DefaultDockerClient.fromEnv().build();  
 //配置Harbor系统账户  
 final RegistryAuth registryAuth = RegistryAuth.create(username, pw, null,

HarborServerAddress, null, null);

//客户端将指定的镜像推送至镜像仓库中  
 dockerClient.push(image, registryAuth);  
}

1. 创建Deployment资源实现应用程序的部署。持续交付平台与kubernetes进行交互，创建Deployment资源实例，由Deployment指挥 Kubernetes 如何创建和更新应用程序的实例。创建 Deployment 后，Kubernetes master 将应用程序实例调度到集群中的各个节点上。首先需要创建kubernetes客户端，默认读取当前系统用户文件夹下的.kube/config文件，该文件是创建kubernetes集群时生成的配置文件，config文件中描述的kubernetes master节点的地址，以及其他认证信息，然后与kubernetes master建立连接。需要指定应用程序的容器镜像以及要运行的副本数。对于私有仓库需要指定仓库的用户名，密码。具体编码实现如下：

public List<LocalObjectReference> newSecrets(DockerRepository auth ) {

//DockerRepository类封装了Harbor账户所需的url,用户名密码等信息  
 List<LocalObjectReference> secrets = new ArrayList<>();  
 LocalObjectReference secret = new LocalObjectReference();

secret.setAdditionalProperty("docker-registry", auth.getRegistry());

//以下省略docker-server, docker-username, docker-password等属性设置的编码实现

**…**  
 secret.setName(auth.getRepositoryName());  
 secrets.add(secret);  
 return secrets;  
}

public PodTemplateSpec newPodTemplate(Map<String, String> podLabels, String image, String containerName, DockerRepository auth) {

PodTemplateSpec podTemplateSpec = new PodTemplateSpec();

ObjectMeta podMeta = new ObjectMeta();

podMeta.setLabels(podLabels);

PodSpec podSpec = new PodSpec();

List<LocalObjectReference> secrets = newSecrets(auth);

podSpec.setImagePullSecrets(secrets);

Container container = new Container();

container.setImage(image);

container.setName(containerName);

//设置镜像拉取策略，Always表示总是拉取新的镜像

container.setImagePullPolicy("Always");

List<Container> containers = new ArrayList<>();

containers.add(container);

podSpec.setContainers(containers);

podTemplateSpec.setMetadata(podMeta);

podTemplateSpec.setSpec(podSpec);

return podTemplateSpec;

}

public void newDeployment(String deploymentName, String ns, int rs, Map<String, String> labels, Map<String, String> podSelector) {

//创建客户端

Config config = new ConfigBuilder().build();

KubernetesClient client = new DefaultKubernetesClient(config);

//创建Deployment资源控制器对象

Deployment deployment = new DeploymentBuilder().build();

deployment.setKind(KubernetesConstraint.KUBERNETES\_DEPLOYMENT);

//创建Deployment Metadata对象并设置属性

ObjectMeta deploymentMeta = new ObjectMeta();

//设置该Deployment所在的命名空间

deploymentMeta.setNamespace(ns);

//设置该Deployment的名称

deploymentMeta.setName(deploymentName);

//设置该Deployment的标签

deploymentMeta.setLabels(labels);

deployment.setMetadata(deploymentMeta);

//创建Deployment的Spec属性对象

DeploymentSpec spec = new DeploymentSpec();

//设置副本数

spec.setReplicas(rs);

LabelSelector labelSelector = new LabelSelector();

//目标pod的标签选择器

labelSelector.setMatchLabels(podSelector);

spec.setSelector(labelSelector);

//通过Pod模板创建Pod

PodTemplateSpec podTemplateSpec = newPodTemplate();

spec.setTemplate(podTemplateSpec);

deployment.setSpec(spec);

client.apps().deployments().create(deployment);

}

1. 创建Service资源实现服务的暴露与发现。此时，kubernetes为Deployment管理的Pod只创建了集群内部的虚拟IP，只有集群内部才能访问。Service资源可以将程序运行的服务暴露出去，并且具有负载均衡的功能。创建Service资源需要指定Service的name属性，且需要指定需要作用于Pod的标签。具体编码实现如下：

public void newService(String serviceName, String ns, Map<String, String> labels, Map<String, String> selector) {  
 Config config = new ConfigBuilder().build();  
 KubernetesClient client = new DefaultKubernetesClient(config);  
 Service service = new Service();  
 //设置service资源类型名  
 service.setKind(KubernetesConstraint.KUBERNETES\_SERVICE);  
 //新建并设置Service meta对象  
 ObjectMeta serviceMeta = new ObjectMeta();  
 serviceMeta.setName(serviceName);  
 serviceMeta.setNamespace(ns);  
 serviceMeta.setLabels(labels);  
 //新建并设置Service spec对象  
 ServiceSpec serviceSpec = new ServiceSpec();

//设置service资源类型  
 serviceSpec.setType("NodePort");  
 serviceSpec.setSelector(selector);  
 ServicePort port = new ServicePort("http", port, targetPort, KubernetesConstraint.PROTOCOL\_TCP,

new IntOrString(80));  
 List<ServicePort> list = new ArrayList<>();  
 list.add(port);  
 serviceSpec.setPorts(list);  
 service.setSpec(serviceSpec);  
 service.setMetadata(serviceMeta);  
 client.services().create(service);  
}

1. **案例测试**

通过创建一个springboot微服务程序来验证持续交付平台的功能是否达到预期。测试程序能够读取主机名，来验证负载均衡功能是否实现。主要测试能否正确构建Docker镜像并推送至Harbor镜像仓库中；能否通过持续交付平台在kubernetes中部署该程序，并创建副本；能否通过出持续交付平台正确创建服务，实现从集群外部网络进行访问。

（1）、镜像构建与推送测试。构建docker镜像主要使用Dockerfile文件，Dockerfile文件中执行构建的命令，测试程序的Dockerfile如下所示：

# 基础镜像

FROM openjdk:8-jdk-alpine

# 对应pom.xml文件中的dockerfile-maven-plugin插件buildArgs配置项JAR\_FILE的值

ARG JAR\_FILE=target/\*.jar

# 复制打包完成后的jar文件到/opt目录下

COPY ${JAR\_FILE} /opt/app.jar

# 启动容器时执行

ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/opt/app.jar"]

# 使用端口80

EXPOSE 80

使用持续交付平台将镜像打上标签：hub.devops.com.helloworld/hello-app:v1，并推送至Harbor镜像仓库中。推送成功后，Habror平台显示如图2：



图2

（2）、部署测试。通过持续交付平台创建Deployment资源，资源名称为：springboot-app-depolyment；镜像为：hub.devops.com/helloworld/hello-app:v1；标签为：app=helloworld；副本数为4。资源创建成功后，通过命令查看kubernetes集群中Deployment资源的状态如图3所示：



图3

图中显示成功创建4个副本，并且成功运行，查看Pod的状态如图4所示：

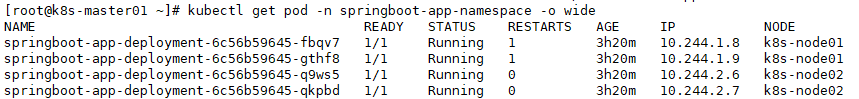


图4

图中显示4个Pod正在运行。

（3）、测试service资源能否将服务暴露出去。持续交付平台创建service资源，类型为NodePort；节点端口为：30715；目标Pod内部端口为：80；需要作用于Pod的标签为：app=helloworld。使用命令查看创建结果如图5所示：



图5

使用Linux curl命令进行服务的访问，curl每次只访问192.168.18.22：30715，结果显示每次访问的返回的主机名每4次重复，说明service资源正常工作，每次访问服务，service使用轮询算法进行负载均衡，如图6所示：

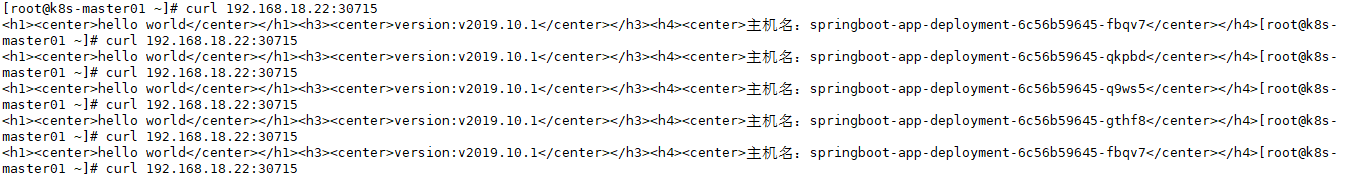


图6

测试表明，持续交付平台能够正常的完成交付流程，有效的提高交付效率。

**4、结语**

整个持续交付平台更像是各种基础服务平台的client端的集成者和各基础服务平台的工作命令的发出者,是面向实际工作平台的统一管理者。本文从软件工程理论角度分析了持续交付平台出现的背景，将微服务、容器、DevOps统一起来。分析生产实践中持续交付的流程，给出持续交付平台需要运行的基础服务环境。在实现部分，给出了如何利用各平台的开源客户端进行项目源码的拉取、编译、构建镜像、创建kubernetes Deployment资源进行应用程序的部署、创建kubernetes Service资源进行服务的暴露与发现等实际的编码实现。在案例测试中，通过创建springboot微服务程序来验证平台的可用性，测试结果证明软件交付效率大大提升。整个持续交付平台满足生产级别的功能需求，具有较大的使用价值。

**参考文献**

[1]徐正，杨鹤标.基于Kubernetes调度器的服务质量[J].软件导刊,2018,17(11):73-76.

[2]杨保华，戴王剑，曹亚仑.Docker技术入门与实践[M].北京:机械工业出版社,2018:3-9.

[3]Marko Luksa.Kubernetes in Action中文版[M].七牛容器云团队[译].北京:电子工业出版社.2019:1-20.

[4]小枣菌.DevOps到底是什么意思[EB/OL].<https://zhuanlan.zhihu.com/p/91371659>.