**加我微信可进群学习交流：**

**微信号：**

luckylucky421302

也可通过扫描下面二维码添加



**课程更新的知识点会通过微信公众号免费分享给大家，可以关注我的公众号**

****

**原文文档如下：**

https://mp.weixin.qq.com/s?\_\_biz=MzU0NjEwMTg4Mg==&mid=2247485894&idx=1&sn=ef0dba192c1dfca4bd1521b4f038a4ef&chksm=fb6386d3cc140fc52b263b55dc793c9873cc249fd793c40da4a8061eab1d3d183f07ac80a240&token=1133627899&lang=zh\_CN#rd

**一、Tekton是什么？**

Tekton是一个谷歌开源的kubernetes原生CI/CD系统，功能强大且灵活，开源社区也正在快速的迭代和发展壮大。 简单的理解，就是all in kubernetes，所以用容器化的方式构建容器镜像是必然，另外，基于kubernetes CRD定义的pipeline流水线也是Tekton最重要的特征。

**二、为什么要用k8s原生的CI/CD工具Tekton？**

持续集成是云原生应用的支柱技术之一，因此在交付基于云原生的一些支撑产品的时候，CICD是一个无法拒绝的需求。为了满足这种需要，自然而然会想到Jenkins(X)或者 Gitlab 进行集成，然而这两个东西虽说功能强大，却也不是为了做螺丝钉而设计的，其中包含了大量的周边功能，并非我们产品的需要，并且其接口和 Pipeline 设计也不太容易复用和提供给用户进行定制，而 Tekton 这个东西可以实现很多功能：

1.Kubernetes原生Tekton的所有配置都是使用CRD方式进行编写存储的，非常易于检索和使用。

2.配置和流程分离 Tekton 的 Pipeline 和配置可以分开编写，使用名称进行引用。

3.轻量级核心的Pipeline非常轻便，适合作为组件进行集成，另外也有周边的 Dashboard、Trigger、CLI 等工具，能够进一步挖掘其潜力。

4.可复用、组合的Pipeline构建方式非常适合在集成过程中对Pipeline进行定制。

**三、安装**

**1.构建tekton需要的镜像如下：**

xianchao/tekton-controller:v0.12.0

xianchao/tekton-webhook:v0.12.0

镜像的压缩包所在的百度网盘地址如下：

链接：https://pan.baidu.com/s/1MvoPKe\_8KKhNntXZJE87ew

提取码：yvcb

如果机器不能上网，可以把镜像上传到k8s的各个node节点，手动解压：

docker load -i tekton-controller.tar.gz

docker load -i tekton-webhook.tar.gz

**2.安装Tekton，需要有一个k8s集群，我用的是k8s1.18.2，k8s1.16+以上版本都可以**

把release.yaml上传到k8s的master节点，通过kubectl apply更新yaml文件：

kubectl apply -f release.yaml

构建tekton需要的release.yaml文件所在百度网盘链接如下：

链接：https://pan.baidu.com/s/1MvoPKe\_8KKhNntXZJE87ew

提取码：yvcb

**3.查看pod是否创建成功：**

kubectl get pods -n tekton-pipelines

显示如下，说明创建成功：

NAME READY STATUS RESTARTS AGE

tekton-pipelines-controller-c557867f6-sfnnt 1/1 Running 0 2m50s

tekton-pipelines-webhook-c9677d7b5-4l82k 1/1 Running 0 2m50s

kubectl api-versions

可看到新出现如下api：

tekton.dev/v1alpha1

tekton.dev/v1beta1

**四、Tekton概念**

Tekton 为 Kubernetes 提供了多种 CRD 资源对象，可用于定义我们的流水线，主要有以下几个资源对象：

1）Task：表示执行命令的一系列步骤，task 里可以定义一系列的 steps，例如编译代码、构建镜像、推送镜像等，每个 step 实际由一个 Pod 执行。

2）TaskRun：task 只是定义了一个模版，taskRun 才真正代表了一次实际的运行，当然你也可以自己手动创建一个taskRun，taskRun创建出来之后，就会自动触发task描述的构建任务。

3）Pipeline：一组任务，表示一个或多个task、PipelineResource 以及各种定义参数的集合。

4）PipelineRun：类似task和taskRun的关系，pipelineRun也表示某一次实际运行的 pipeline，下发一个 pipelineRun CRD 实例到 Kubernetes后，同样也会触发一次 pipeline 的构建。

5）PipelineResource：表示 pipeline 输入资源，比如 github 上的源码，或者 pipeline 输出资源，例如一个容器镜像或者构建生成的 jar 包等。

**五、案例演示**

**注意：**下面实验需要的yaml文件所在百度网盘链接如下：

链接：https://pan.baidu.com/s/1MvoPKe\_8KKhNntXZJE87ew

提取码：yvcb

接下来我们演示使用一个简单的golang程序，应用程序代码，测试及dockerfile文件可在如下地址获取：

<https://github.com/luckylucky421/tekton-demo>

1.clone应用程序代码进行测试，创建一个task任务，对应的task-test.yaml文件内容如下：

apiVersion: tekton.dev/v1beta1

kind: Task

metadata:

name: test

spec:

resources:

inputs:

- name: repo

type: git

steps:

- name: run-test

image: golang:1.14-alpine

workingDir: /workspace/repo

command: ["go"]

args: ["test"]

resources 定义了我们的任务中定义的步骤所需的输入内容，这里我们的步骤需要 Clone 一个 Git 仓库作为 go test 命令的输入。Tekton 内置了一种 git 资源类型，它会自动将代码仓库 Clone 到 /workspace/$input\_name 目录中，由于我们这里输入被命名成 repo，所以代码会被 Clone 到 /workspace/repo 目录下面。然后下面的 steps 就是来定义执行运行测试命令的步骤，这里我们直接在代码的根目录中运行 go test 命令即可，需要注意的是命令和参数需要分别定义。

通过kubectl apply更新yaml文件：

kubectl apply -f task-test.yaml

2.创建pipelineresource资源对象

现在我们定义完成了一个 Task 任务，但是该任务并不会立即执行，我们必须创建一个 TaskRun 引用它并提供所有必需输入的数据才行。这里我们就需要将 git 代码库作为输入，我们必须先创建一个 PipelineResource 对象来定义输入信息，创建一个名为 pipelineresource.yaml 的资源清单文件，内容如下所示：

apiVersion: tekton.dev/v1alpha1

kind: PipelineResource

metadata:

name: xianchao-tekton-example

spec:

type: git

params:

- name: url

value: https://github.com/luckylucky421/tekton-demo

- name: revision

value: master

通过kubectl apply 更新yaml文件：

kubectl apply -f pipelineresource.yaml

3.创建一个TaskRun任务，taskrun.yaml文件如下：

apiVersion: tekton.dev/v1beta1

kind: TaskRun

metadata:

name: testrun

spec:

taskRef:

name: test

resources:

inputs:

- name: repo

resourceRef:

name: xianchao-tekton-example

这里通过taskRef引用上面定义的Task和git仓库作为输入，resourceRef也是引用上面定义的PipelineResource资源对象。现在我们创建这个资源对象过后，就会开始运行了：

通过kubectl apply更新yaml文件：

kubectl apply -f taskrun.yaml

创建后，我们可以通过查看 TaskRun 资源对象的状态来查看构建状态

**kubectl get taskrun**

显示如下：

NAME SUCCEEDED REASON STARTTIME COMPLETIONTIME

testrun Unknown Pending 29s

**kubectl get pods**

显示如下：

NAME READY STATUS RESTARTS AGE

testrun-pod-kxjdp 0/2 Init:ErrImagePull 0 66s

**kubectl describe pods testrun-pod-kxjdp**

显示如下：

Normal BackOff 6s (x4 over 115s) kubelet, node1 Back-off pulling image "busybox@sha256:a2490cec4484ee6c1068ba3a05f89934010c85242f736280b35343483b2264b6"

Warning Failed 6s (x4 over 115s) kubelet, node1 Error: ImagePullBackOff

说明镜像拉取失败了，可以把busybox这个镜像压缩包传到k8s的node节点，手动解压即可：

docker load -i busybox.tar.gz

docker load -i tekton-entrypoint.tar.gz

docker load -i tekton-kubeconfigwriter.tar.gz

docker load -i tekton-creds-init.tar.gz

docker load -i tekton-git-init.tar.gz

docker load -i tekton-imagedigestexporter.tar.gz

docker load -i tekton-pullrequest-init.tar.gz

docker load -i tekton-gcs-fetcher.tar.gz

docker load -i tianon.tar.gz

镜像的压缩包在百度网盘，地址如下：

链接：https://pan.baidu.com/s/1MvoPKe\_8KKhNntXZJE87ew

提取码：yvcb

kubectl get pods

显示如下：

NAME READY STATUS RESTARTS AGE

testrun-pod-l9bbh 2/2 Running 0 16s

kubectl get taskrun

显示如下：

NAME SUCCEEDED REASON STARTTIME COMPLETIONTIME

testrun Unknown Running 12s

当任务执行完成后， Pod 就会变成 Completed 状态了：

kubectl get pods

显示如下：

NAME READY STATUS RESTARTS AGE

testrun-pod-l9bbh 0/2 Completed 0 85s

我们可以通过kubectl describe命令来查看任务运行的过程，首先就是通过 initContainer 中的一个 busybox 镜像将代码 Clone 下来，然后使用任务中定义的镜像来执行命令。当任务执行完成后， Pod 就会变成 Completed 状态了：

我们可以查看容器的日志信息来了解任务的执行结果信息

kubectl logs testrun-pod-l9bbh --all-containers

显示如下：

{"level":"info","ts":1594307950.1498194,"caller":"git/git.go:136","msg":"Successfully cloned https://github.com/luckylucky421/tekton-demo @ c6c2a85091d538a13c44f85bcee9e861c362b0d3 (grafted, HEAD, origin/master) in path /workspace/repo"}

{"level":"info","ts":1594307950.2021587,"caller":"git/git.go:177","msg":"Successfully initialized and updated submodules in path /workspace/repo"}

PASS

ok \_/workspace/repo 0.017s

我们可以看到我们的测试已经通过了。

六、总结：

我们已经在 Kubernetes 集群上成功安装了 Tekton，定义了一个 Task，并通过 YAML 清单和创建 TaskRun 对其进行了测试。