

17 | 自动构建：如何使用 GitLab CI 构建镜像？

2023-01-16 王伟 来自北京



《云原生架构与GitOps实战》

[课程介绍 >](#)



讲述：王伟

时长 08:05 大小 7.38M



你好，我是王伟。


在上一节课，我们学习了如何使用 **GitHub Action** 自动构建镜像，我们通过为示例应用配置 **GitHub Action** 工作流，实现了自动构建，并将镜像推送到了 **Docker Hub** 镜像仓库。

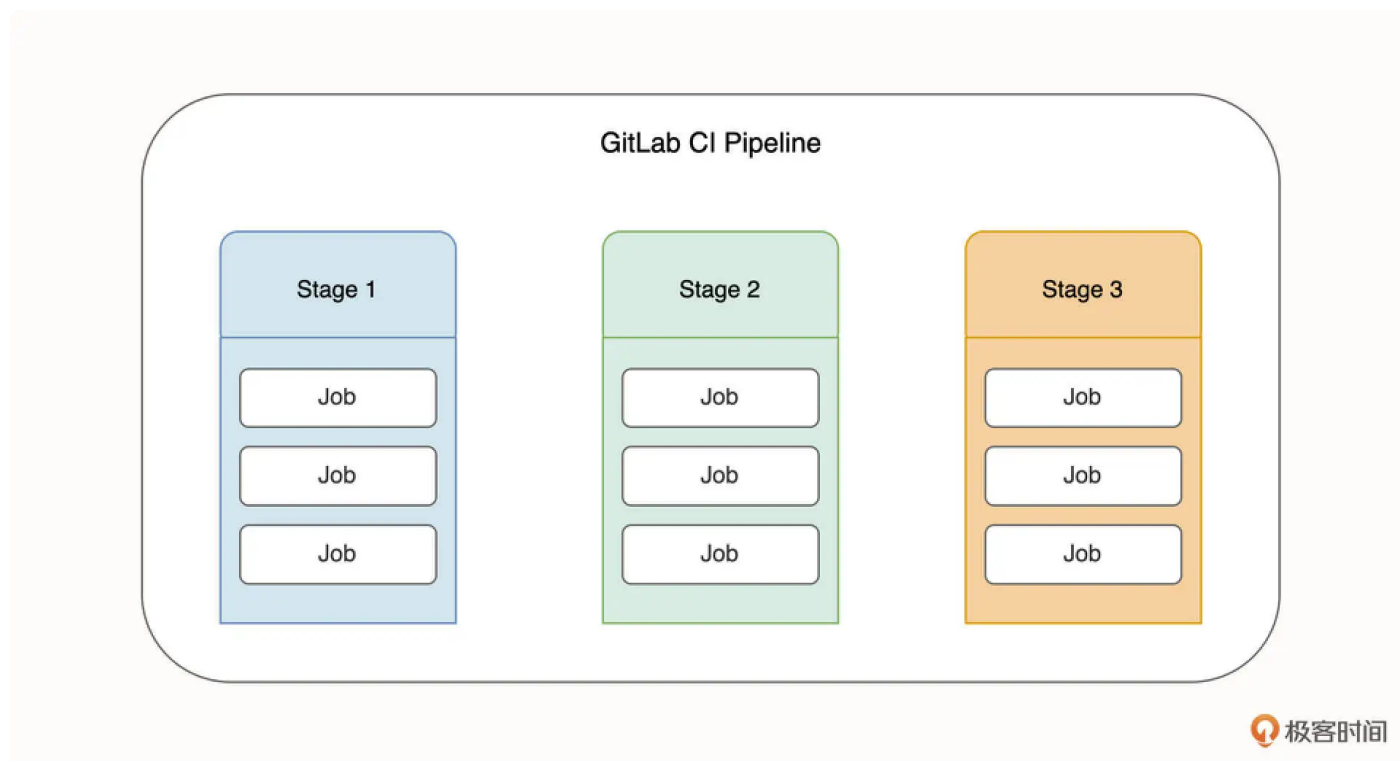
但是，要使用 **GitHub Action** 构建镜像，前提条件是你需要使用 **GitHub** 作为代码仓库，那么，如果我所在的团队使用的是 **GitLab** 要怎么做呢？

这节课，我会带你学习如何使用 **GitLab CI** 来自动构建镜像。我还是以示例应用为例，使用 **SaaS** 版的 **GitLab** 从零配置 **CI** 流水线。

需要注意的是，有些团队是以自托管的方式来使用 **GitLab** 的，也就是我们常说的私有部署的方式，它和 **SaaS** 版本的差异不大。如果你用的是私有化部署版本，同样可以按照这节课的流程来实践。

GitLab CI 简介

在正式使用 GitLab CI 之前，你需要先了解一些基本概念，你可以结合下面这张图来理解。 <https://shikey.com/>



这张图中出现了 Pipeline、Stage 和 Job 这几个概念，接下来我们分别了解一下。

Pipeline

Pipeline 指的是流水线，在 GitLab 中，当有新提交推送到仓库中时，会自动触发流水线。流水线包含一组 Stage 和 Job 的定义，它们负责执行具体的逻辑。

在 GitLab 中，Pipeline 是通过仓库根目录下的 `.gitlab-ci.yml` 文件来定义的。

此外，Pipeline 在全局也可以配置运行镜像、全局变量和额外服务镜像。

Stage

Stage 字面上的意思是“阶段”。在 GitLab CI 中，至少需要包含一个 Stage，上面这张图中有三个 Stage，分别是 Stage1、Stage2 和 Stage3，不同的 Stage 是按照定义的顺序依次执行的。如果其中一个 Stage 失败，则整个 Pipeline 都将失败，后续的 Stage 也都不会再继续执行。

Job

Job 字面上的意思是“任务”。实际上，Job 的作用是定义具体需要执行的 Shell 脚本，同时，Job 可以被关联到一个 Stage 上。当 Stage 执行时，它所关联的 Job 也会并行执行。



以自动构建镜像为例，我们可能需要在 1 个 Job 中定义 2 个 Shell 脚本步骤，它们分别是：

- 运行 `docker build` 构建镜像
- 运行 `docker push` 来推送镜像

费用

和 GitHub Action 一样，GitLab 也不能无限免费使用。对于 GitLab 免费账户，每个月有 400 分钟的 GitLab CI/CD 时长可供使用，超出时长则需要按量付费，你可以在 [🔗 这里](https://docs.gitlab.com/ee/ci/pricing/index.html) 查看详细的计费策略。

为示例应用创建 GitLab CI Pipeline

在简单学习了 GitLab CI 相关概念之后，**接下来我们进入到实战环节。**

我仍然以示例应用为例，介绍如何配置自动构建示例应用的前后端镜像流水线。在这个例子中，我们创建的流水线将实现以下这些步骤。

- 运行 `docker login` 登录到 Docker Hub。
- 运行 `docker build` 来构建前后端应用的镜像。
- 运行 `docker push` 推送镜像。

接下来，我们开始创建 GitLab CI Pipeline。

创建 `.gitlab-ci.yml` 文件

首先，将示例应用仓库克隆到本地。

```
1 $ git clone https://github.com/lyzhang1999/kubernetes-example.git
```

 复制代码

进入 `kubernetes-example` 目录。

```
1 $ cd kubernetes-example
```



然后，将下面的内容保存到 `.gitlab-ci.yml` 文件内。

复制代码

```
1 stages:
2   - build
3
4 image: docker:20.10.16
5
6 variables:
7   DOCKER_TLS_CERTDIR: "/certs"
8   DOCKERHUB_USERNAME: "lyzhang1999"
9
10 services:
11   - docker:20.10.16-dind
12
13 before_script:
14   - docker login -u $DOCKERHUB_USERNAME -p $DOCKERHUB_TOKEN
15
16 build_and_push:
17   stage: build
18   script:
19     - docker build -t $DOCKERHUB_USERNAME/frontend:$CI_COMMIT_SHORT_SHA ./front
20     - docker push $DOCKERHUB_USERNAME/frontend:$CI_COMMIT_SHORT_SHA
21     - docker build -t $DOCKERHUB_USERNAME/backend:$CI_COMMIT_SHORT_SHA ./backen
22     - docker push $DOCKERHUB_USERNAME/backend:$CI_COMMIT_SHORT_SHA
```

请注意，你需要将上面的 `variables.DOCKERHUB_USERNAME` 环境变量替换为你的 **Docker Hub 用户名**。

接下来，结合上面提到的概念，我简单介绍一下这个 Pipeline。

`stages` 字段定义了阶段，在这个 Pipeline 中，我们定义了一个 `build` 阶段。

`image` 字段定义了运行镜像，也就是说，GitLab CI 将会使用 `docker:20.10.16` 镜像来启动一个容器，并在容器内运行 Pipeline。

`variables` 字段定义了全局变量，其中，`DOCKER_TLS_CERTDIR` 变量是用来共享 Docker 证书的。`DOCKERHUB_USERNAME` 变量是 Docker Hub 的用户名。

services 字段定义了一个额外的镜像，你可以把它理解成一个额外的容器，它将和 image 字段定义的容器相互协作，这两个容器可以相互访问。



before_script 定义了 Pipeline 最开始的 Shell 脚本，它将会在 Job 运行之前执行。在这里，我们运行了 docker login 命令来登录到 Docker Hub，以便获得推送镜像的权限。请注意，\$DOCKERHUB_USERNAME 变量的值来源于我们在 variables 定义的值，\$DOCKERHUB_TOKEN 是一个在 GitLab UI 界面定义的变量，我们稍后会在 GitLab 平台添加。

build_and_push 字段定义了一个 Job，“build_and_push”实际上是 Job 的名称，你也可以更改这个名称。build_and_push.stage 字段定义了 Job 所属的 Stage，也就是 build 阶段。build_and_push.script 字段定义了执行的具体的 Shell 脚本，它们是按顺序执行的。在这里，我们分别构建了 frontend 和 backend 的镜像，并将它们推送到 Docker Hub 仓库。其中，\$CI_COMMIT_SHORT_SHA 是一个内置变量，它可以获取到当前的 short commit id。

创建 GitLab 仓库并推送

创建完 .gitlab-ci.yml 文件后，接下来我们将示例应用推送到 GitLab 上。首先，你需要通过 [这个页面](#) 来为你自己创建新的代码仓库，仓库名设置为 kubernetes-example。

创建完成后，将刚才克隆的 kubernetes-example 仓库的 remote url 配置为你刚才创建仓库的 Git 地址。

复制代码

```
1 $ git remote set-url origin YOUR_GITLAB_REPO_URL
```

然后，将 `kubernetes-example` 推送到你的 `GitLab` 仓库中。在这之前，你可能需要配置 `SSH Key`，你可以参考 [这个链接](https://shikey.com/) 来配置，这里就不再赘述了。

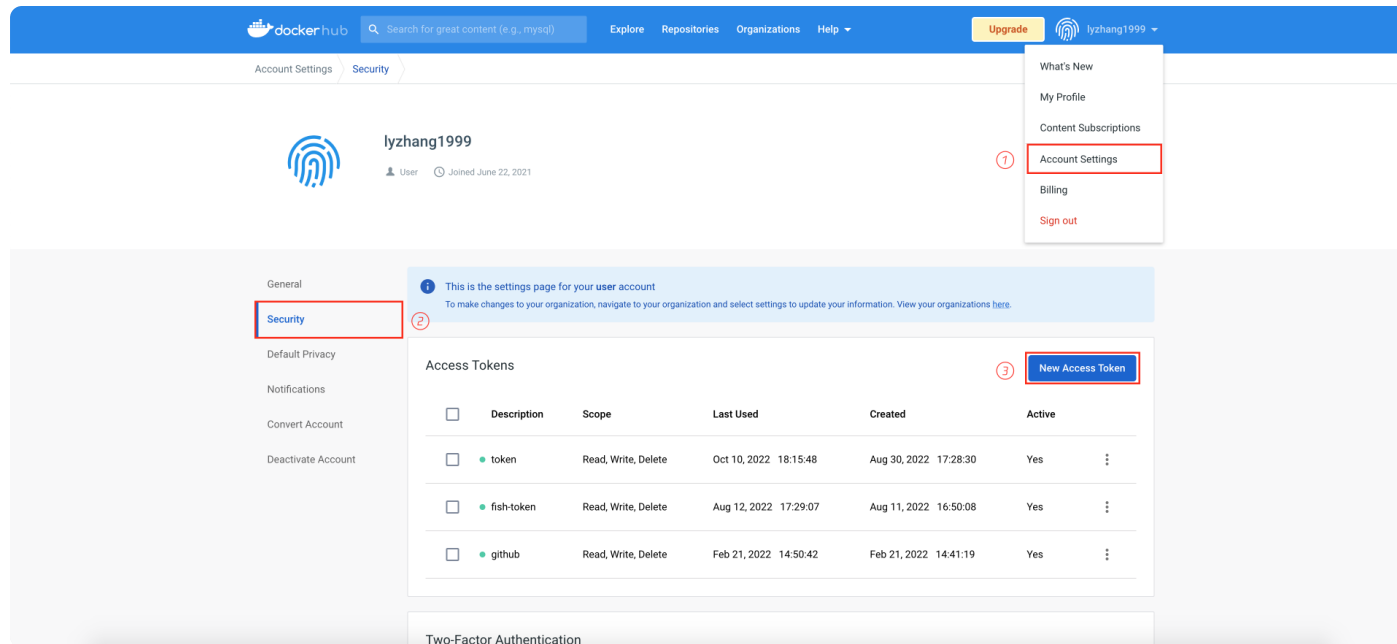
复制代码

```
1 $ git add .
2 $ git commit -a -m 'first commit'
3 $ git branch -M main
4 $ git push -u origin main
```

创建 Docker Hub Secret

创建完 `.gitlab-ci.yml` 文件后，接下来我们需要创建 `Docker Hub Secret`，它将会为工作流提供推送镜像的权限。

首先，使用你注册的账号密码登录 <https://hub.docker.com/>。然后，点击右上角的“用户名”，选择“Account Settings”，并进入左侧的“Security”菜单。



然后，点击右侧的“New Access Token”按钮创建一个新的 Token。

New Access Token

A personal access token is similar to a password except you can have many tokens and revoke access to each one at any time. [Learn more](#)



Access Token Description *

github action

Access permissions

Read, Write, Delete



Read, Write, Delete tokens allow you to manage your repositories.

Cancel

Generate

输入描述，然后点击“Genarate”按钮生成 Token。

Copy Access Token

When logging in from your Docker CLI client, use this token as a password. [Learn more](#)



ACCESS TOKEN DESCRIPTION

github action

ACCESS PERMISSIONS

Read, Write, Delete

To use the access token from your Docker CLI client:

1. Run `docker login -u lyzhang1999`
2. At the password prompt, enter the personal access token.

dckr_pat_ARKScInQx73ek



WARNING: This access token will only be displayed once. It will not be stored and cannot be retrieved. Please be sure to save it now.

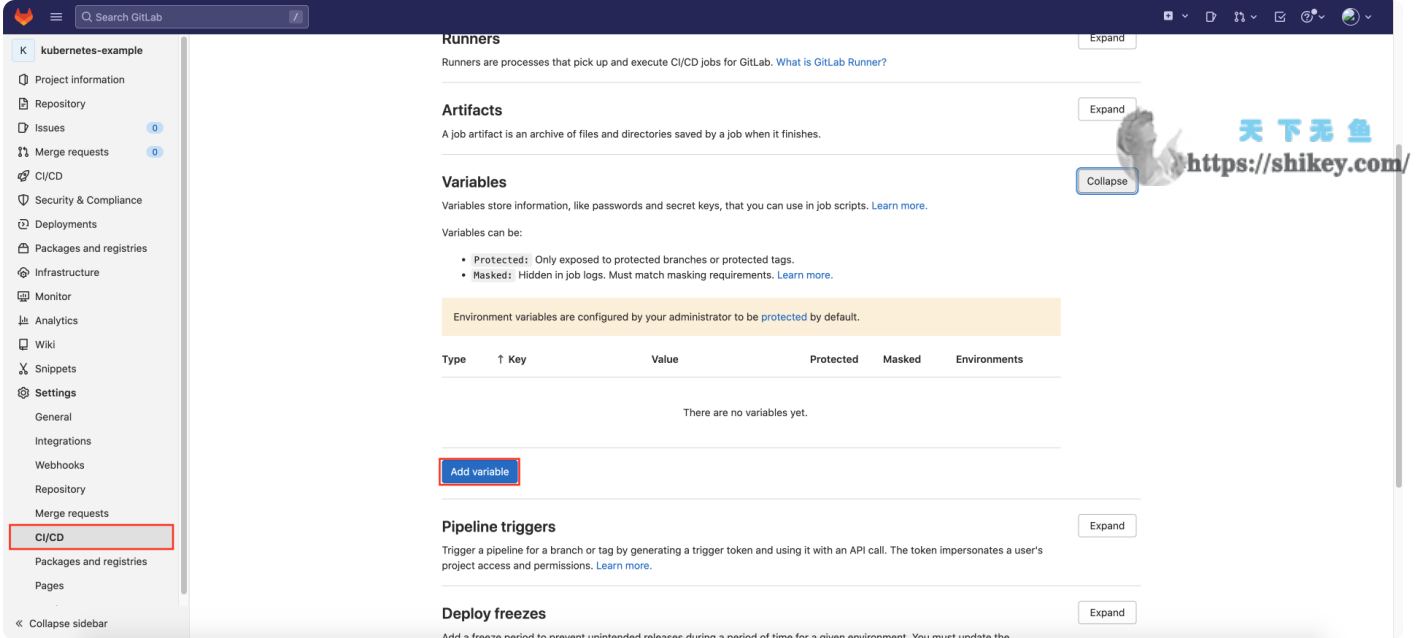
Copy and Close

点击“Copy and Close”将 Token 复制到剪贴板。请注意，一旦窗口关闭，我们就无法再次查看这个 Token 了，所以请务必复制并在其他地方保存下来。

创建 GitLab CI Variables

创建完 Docker Hub Token 之后，我们就可以创建 GitLab CI Variables 了，也就是要为 Pipeline 提供 DOCKERHUB_TOKEN 变量值。

进入 kubernetes-example 仓库的 Settings 页面，点击左侧的“CI/CD”，然后点击右侧的“Variables”展开菜单，接着点击“Add variable”来创建新的 Variables。如下图所示。



在弹出的输入框中，将 Key 填写为 DOCKERHUB_TOKEN。

将 Value 填写为刚才我们复制的 Docker Hub Token，其他选项保持默认，点击“Add variable”创建变量值，如下图所示。

Add variable

Key

DOCKERHUB_TOKEN

Value

dckr_pat_DcarpvfYSNY58pzh

Type

Variable

Environment scope ?

All (default)

Flags

☒ Protect variable ?

Export variable to pipelines running on protected branches and tags only.

☐ Mask variable ?

Variable will be masked in job logs. Requires values to meet regular expression requirements. [More information](#)

Cancel

Add variable

触发 GitLab CI Pipeline

到这里，准备工作已经全部完成了。请注意，如果你使用的是 GitLab SaaS 版，**那么你需要先绑定信用卡才能使用 CI/CD 的免费额度。**



接下来我们尝试触发 GitLab CI Pipeline。

首先，向仓库提交一个空 commit。

复制代码

```
1 $ git commit --allow-empty -m "Trigger Build"
```

然后，使用 git push 来推送到仓库，**这将触发 Pipeline。**

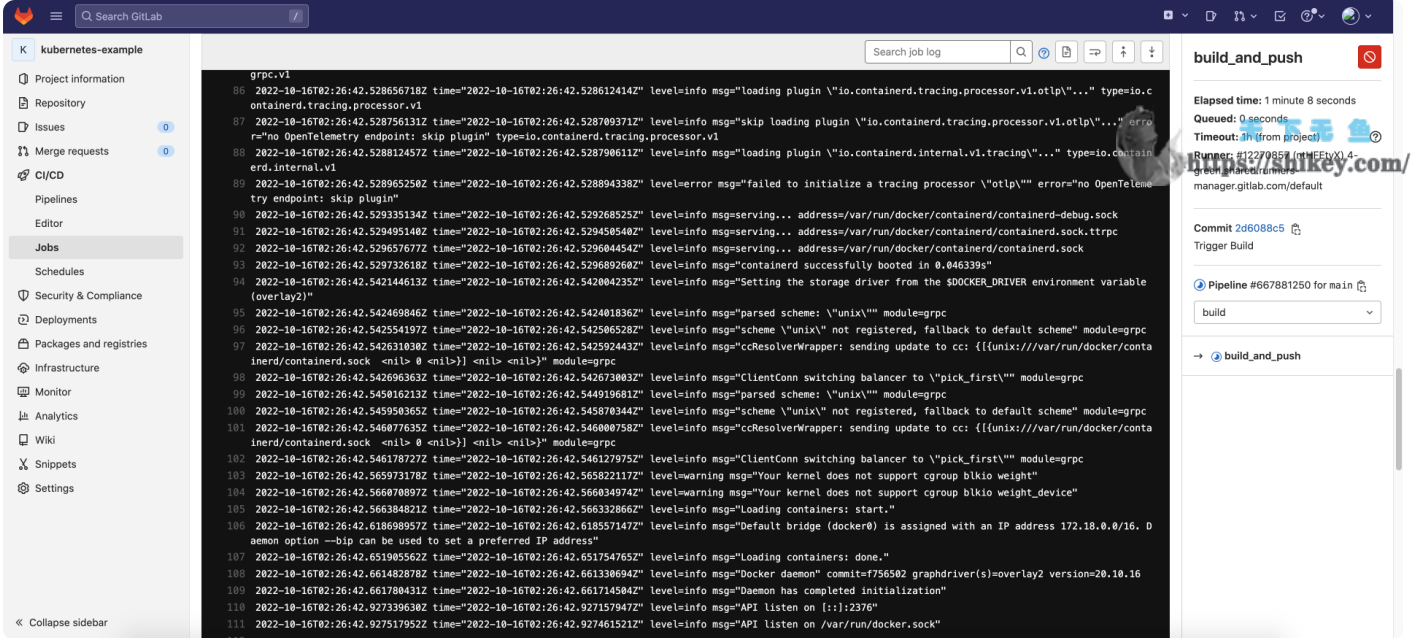
复制代码

```
1 $ git push origin main
```

接下来，进入 kubernetes-example 仓库的“CI/CD”页面，你会看到我们刚才触发的流水线。

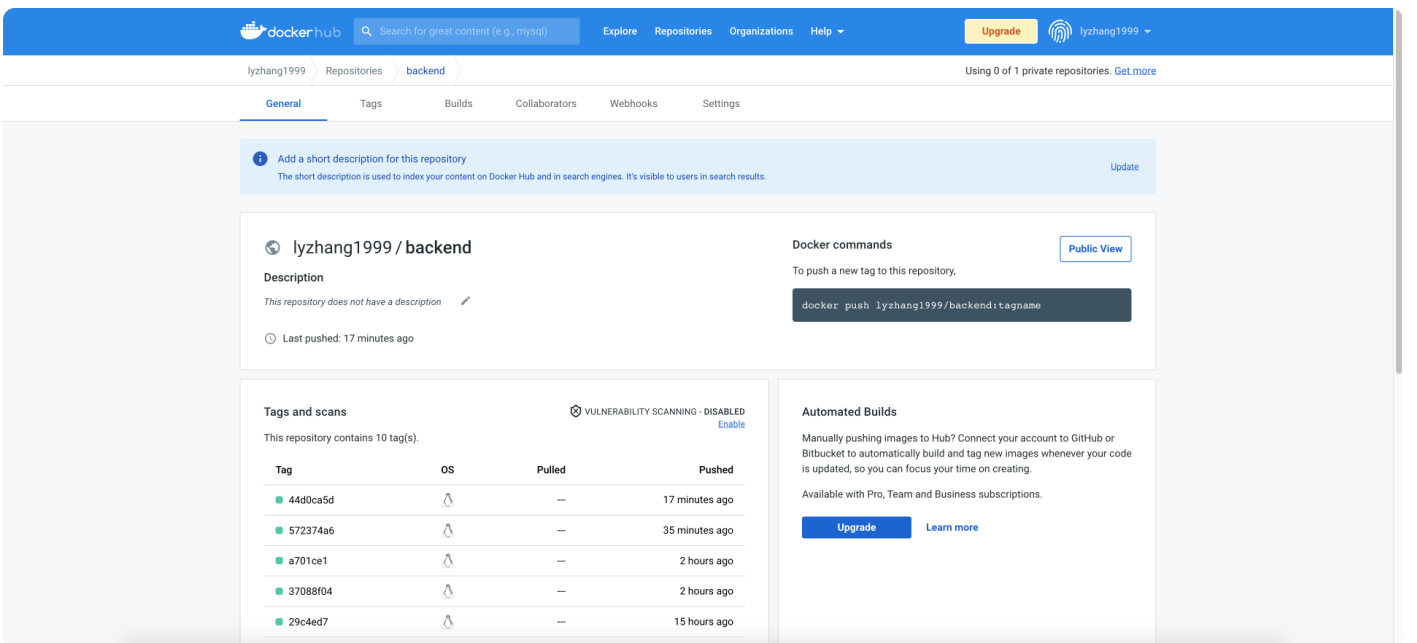
Status	Pipeline	Triggerer	Stages
running in progress	Trigger Build #667881250 main -> 2d6088c5 wangwei	vyat	
passed 00:04:30 13 minutes ago	add gitlab ci yml file #667878484 main -> 44d0ca5d wangwei	vyat	
passed 00:04:31 32 minutes ago	add gitlab ci yml file #667871864 main -> 572374a6 wangwei	vyat	
canceled 00:01:47 1 hour ago	add gitlab ci yml file #667859769 main -> a701ce1f wangwei	vyat	
passed 00:04:35 1 hour ago	add gitlab ci yml file #667855749 main -> 37088f04 wangwei	vyat	
failed 10 hours ago	add gitlab ci yml file #667739601 main -> 37088f04 wangwei	vyat	

你可以点击流水线的状态（running）进入流水线详情页面。



在流水线的详情页面，我们能看见流水线的每一个 Job 的状态还有运行时输出的日志。

当工作流运行完成后，进入到 Docker Hub frontend 或者 backend 镜像的详情页，你将看到刚才 GitLab CI 自动构建并推送的新版本镜像。



到这里，我们便完成了使用 GitLab CI 自动构建镜像。最终实现的效果是，当我们向仓库推送新的提交时，GitLab 流水线将自动构建 frontend 和 backend 镜像，并且每一个 commit id 都会对应一个镜像版本。

总结

在这节课，我为你介绍了如何使用 GitLab CI 来自动构建镜像，并讲解了 Pipeline、Stage 和 Job 几个重要概念。



GitLab CI 是通过在仓库根目录创建 `.gitlab-ci.yml` 文件来定义流水线的，这和 GitHub 有明显的差异。在这节课的例子中，`.gitlab-ci.yml` 文件定义的内容也相对简单，它基本上和我们在本地构建镜像所运行的命令以及顺序是一致的。

此外，相比较 GitHub Action Workflow，GitLab CI 省略了触发器和检出代码的配置步骤，并且，在 GitLab CI 中我们是通过 DiND 的方式来运行流水线的，也就是在容器的运行环境下启动另一个容器来运行流水线，而 GitHub Action 则是通过虚拟机的方式来运行流水线。

和 GitHub Action 相比较，它们除了流水线文件内容不一样以外，其他的操作例如创建 GitLab 仓库、创建 Docker Hub Secret 以及创建 GitLab CI Variables 等步骤都是差不多的。

最终，当我们有新的推送到仓库时，GitLab CI 将运行自动构建镜像的流水线，并且每次提交的 commit id 都会对应一个镜像版本，和 GitHub Action Workflow 一样，**也实现了代码版本和制品版本的对应关系。**

思考题

最后，给你留一道简单的思考题吧。

请你尝试改造 `.gitlab-ci.yml`，使其同时支持构建 linux/amd64 和 linux/arm64 两个平台的镜像，并和我分享改动之后的 YAML。我们下节课见。

分享给需要的人，Ta购买本课程，你将得 18 元

 生成海报并分享

 赞 0  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。



精选留言 (6)



Waylon

2023-01-16 来自上海

stages:

- build

image: docker:20.10.16

variables:

DOCKER_TLS_CERTDIR: "/certs"

DOCKERHUB_USERNAME: "lyzhang1999"

PLATFORM: "linux/amd64,linux/arm64"

services:

- docker:20.10.16-dind

before_script:

- docker buildx create --name builder

- docker buildx use builder

- docker buildx inspect --bootstrap

- docker login -u \$DOCKERHUB_USERNAME -p \$DOCKERHUB_TOKEN

build_and_push:

stage: build

script:

-

- docker buildx build --platform \$PLATFORM -t \$DOCKERHUB_USERNAME/frontend:\$CI_COMMIT_SHORT_SHA ./frontend

- docker push \$DOCKERHUB_USERNAME/frontend:\$CI_COMMIT_SHORT_SHA

- docker buildx build --platform \$PLATFORM -t \$DOCKERHUB_USERNAME/backend:\$CI_COMMIT_SHORT_SHA ./backend

- docker push \$DOCKERHUB_USERNAME/backend:\$CI_COMMIT_SHORT_SHA

作者回复: 正确，使用了变量和 before_script，非常棒的例子！



ghostwritten

2023-01-29 来自美国

遇到两个问题不过解决了：

```
$ docker buildx create --name builder
```

```
error: could not create a builder instance with TLS data loaded from environment. Please use `docker context create <context-name>` to create a context for current environment and then create a builder instance with `docker buildx create <context-name>`
```

```
Cleaning up project directory and file based variables
```

```
00:01
```

```
ERROR: Job failed: exit code 1
```

```
$ docker push $DOCKERHUB_USERNAME/frontend:$CI_COMMIT_SHORT_SHA
```

```
The push refers to repository [docker.io/ghostwritten/frontend]
```

```
An image does not exist locally with the tag: ghostwritten/frontend
```

```
Cleaning up project directory and file based variables
```

```
00:01
```

```
ERROR: Job failed: exit code 1
```

这是我的.gitlab-ci.yaml:

```
stages:
```

```
- build
```

```
image: docker:20.10.16
```

```
variables:
```

```
  DOCKER_TLS_CERTDIR: "/certs"
```

```
  DOCKERHUB_USERNAME: "ghostwritten"
```

```
  PLATFORM: "linux/amd64,linux/arm64"
```

```
services:
```

```
- docker:20.10.16-dind
```

```
before_script:
```

```
- docker context create builder
```

```
- docker buildx create --name builder --use builder
```

```
- docker buildx use builder
```

```
- docker buildx inspect --bootstrap
```

```
- docker login -u $DOCKERHUB_USERNAME -p $DOCKERHUB_TOKEN
```

```
build_and_push:
```

```
  stage: build
```



天下无鱼

<https://shikey.com/>

script:

```
- docker buildx build --platform $PLATFORM -t $DOCKERHUB_USERNAME/frontend:$CI_COMMIT_SHORT_SHA ./frontend --push
- docker buildx build --platform $PLATFORM -t $DOCKERHUB_USERNAME/backend:$CI_COMMIT_SHORT_SHA ./backend --push
```



Jich

2023-01-16 来自上海

私有化部署的gitlab执行CI需要先安装个runner的吧

作者回复: 是的, 私有化部署的 GitLab 需要先配置好 runner。



Promise

2023-01-16 来自浙江

老师怎么使用DinD的方式来进行CI呀。能不能有一篇文章或者教程专门讲一下使用DinD来构建镜像

作者回复: 下一节课马上会讲到如何用 Tekton 构建镜像, 也就是直接在 K8s 集群中通过 DinD 的方式来构建镜像。



那风在极客

2023-01-16 来自广东

DIND 不是被弃用了吗?

作者回复: 安全扫描弃用了 DinD 的方式, CI/CD 使用 DinD 的方式运行仍然是官方推荐的方式之一。



无名无姓

2023-01-16 来自北京

.gitlab-ci.yml : 这个文件是自动生成的么

作者回复: 需要自己创建, 为了方便大家参考, 示例应用里已经包含这个文件了, 你可以先删除掉, 然后再进行实践。



天下无鱼

<https://shikey.com/>