适用于生产环境...

> 上一篇和本篇这两篇内容是特别增加的 CI/CD 相关的内容，主要目标是将前面所学内容应用于实践中。上篇主要会为你介绍几种将 Docker 应用于 CI/CD pipeline 中的方式，本篇会介绍完整的实践。

上一篇，我们的重点是“如何用 Docker”，分别介绍了 CI/CD pipeline 如何在物理机环境及容器环境使用 Docker。你可以对比实际的需求或是权衡利弊来选择以下方案中的一种或者多种：

- 指定 DOCKER_HOST 的环境变量，通过 HTTP/HTTPS 的方式访问 Docker Daemon 无论它是在本机或者是启动在远端服务器上；
- 挂载 Docker Daemon 的 Unix Domain Socket（默认是 /var/run/docker.sock）通常用于在容器环境内构建使用；
- 使用 Docker in Docker 的方式，将 Docker Daemon 启动在容器内，可以做到比较好的环境隔离，用完后可以删掉。

虽然以上方案各有利弊，但在 CI/CD pipeline 的环境下，我个人比较推荐使用 Docker in Docker（DIND）的这种方式，因为 CI/CD pipeline 每天可能会触发很多次，那么**隔离性**和**资源清理和回收**就是我考虑比较多的方面了。

在每次触发 pipeline 时，使用容器启动一个 Docker Daemon，所有 pipeline 彼此独立，用完后即将其销毁，释放资源。

CI/CD pipeline 的组成

有了上述内容的铺垫，我们来具体到本篇的主题“CI/CD pipeline”中。这里我所指的“CI/CD”包括持续集成，持续交付和持续部署三个方面。

就实际使用而言，CI 的部分多数情况是在做打包、构建、测试。当然这里有个值得思考的问题：

> 在 CI 环节中，你会使用源代码进行测试还是测试构建好的镜像？

这里我给出的建议是：**如果是单元测试，或是语法 lint 之类的，那你可以选择直接源码跑测试。**因为这种需求用源码跑测试最快也最方便；**如果是接口测试或者集成测试，那自然不必多说，选择构建好的镜像跑测试**，因为接口测试或集成测试是运行时测试，使用构建好的镜像测试，才能保障最终交付的产物是真正经过测试的。

至于持续交付，只要将在 CI 环节中经过测试的 Docker image 推送到 Docker registry 即可。这部分中涉及到的 Docker 相关的知识在前面内容中都已经介绍了。

最后就是持续部署了，持续部署这个环节，如果说简单一点即：**将经过充分测试的 Docker image 部署在生产环境，并启动即可。**但在实际的环节中，部署过程取决于你真实的环境。比如说如果你使用了 Kubernetes 作为容器的编排工具，那在部署的时候，你完全可以直接更改

镜像；或者你也可以使用其他的工具完成部署操作。

以 GitLab CI 为例

上一篇的最后，我写下了下面这段话：

> 在做 CI/CD pipeline 时，我们可以自己实现一个 CI 平台，也可以利用很多现成的工具。在 CNCF 的全景图中可以看到很多，我个人比较推荐 GitLab CI，除了因为它比较灵活外，它的功能也比较完善，我在使用 GitLab CI 时，用的比较多的方式就是 DIND。

本篇我们也以 GitLab CI 为例。示例项目的完整代码可以从 GitLab 仓库中获取。选择以 GitLab CI 为例除了前面提到的原因外，另一个重要的原因在于：当我们谈论 CI/CD 时，其基础仍然是代码，所以代码仓库是根本。GitLab CI 是 GitLab 原生提供的功能，与代码仓库的的联动非常自然也无需其他特殊配置。

这里先来介绍下示例项目：hello_world.py 便是我们实际的应用程序。

复制

```
#!/usr/bin/env python
# coding=utf-8
import tornado.ioloop
import tornado.web


class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.write("Hello, world")


def make_app():
    return tornado.web.Application([
        (r"/", MainHandler),
    ])


if __name__ == "__main__":
    app = make_app()
    app.listen(8888)
    tornado.ioloop.IOLoop.current().start()
```

这个程序创建了一个简单的 server 监听 8888 端口，访问 `/` 将直接返回 `Hello, world`。接下来看看对应的测试逻辑，在 test*hello*world.py 文件中：

复制

```
#!/usr/bin/env python
# coding=utf-8
from tornado.testing import AsyncHTTPTestCase, unittest

import hello_world


class TestHelloApp(AsyncHTTPTestCase):
    def get_app(self):
        return hello_world.make_app()

    def test_homepage(self):
        response = self.fetch('/')
        self.assertEqual(response.code, 200)
        self.assertEqual(response.body, b'Hello, world')
```

检查返回状态码和返回的内容是否符合预期。

接下来看看 Dockerfile 的内容：

复制

```
FROM python:3.8-slim

WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt

EXPOSE 8888

COPY hello_world.py .
CMD [ "python", "hello_world.py" ]
```

在之前内容中我们介绍过，为了更好地利用 Docker 的构建缓存，优先将不常变更的内容写在前面。

接下来进入正题，先看看下整体的 GitLab CI 的配置文件：

复制

```
image: docker

services:
  - docker:dind

variables:
  DOCKER_DRIVER: overlay2
  DOCKER_TLS_CERTDIR: ''

stages:
  - test
  - build
  - deploy

test:
  stage: test
  image: python:3.8-slim
  script:
    - pip install -r requirements.txt
    - python -m tornado.testing test_hello_world

build:
  stage: build
  script:
    - docker login -u gitlab-ci-token -p $CI_JOB_TOKEN $CI_REGISTRY
    - docker build -t $CI_REGISTRY/$CI_PROJECT_PATH:$CI_COMMIT_SHORT_SHA .
    - docker push $CI_REGISTRY/$CI_PROJECT_PATH:$CI_COMMIT_SHORT_SHA
    - docker logout $CI_REGISTRY

deploy:
  stage: deploy
  script:
    - echo 'Just for deploy.'
```

整个 pipeline 分为 test、build 和 deploy 三个 stage，分别对应测试、构建、部署环节。

对于测试环节，很常规地进行安装依赖，以及运行测试。当测试执行成功后，便开始构建镜像以及将镜像推送到镜像仓库。最终是部署环节，我这里没有写具体实现，正如我前面写的，这依赖于你具体使用的部署工具或者平台。

这里，我推荐一个很有用的工具 Flux，是一个用于 Kubernetes 上的 GitOps 工具，如果你在使用 Kubernetes，我建议你可以了解下这个工具。

常见问题解析

DIND 如何禁用 TLS

接下来，我对构建环节做下介绍，这也是很多人在使用 GitLab CI 或者使用 DIND 时，可能遇到的问题。

```
services:
  - docker:dind

variables:
  DOCKER_DRIVER: overlay2
  DOCKER_TLS_CERTDIR: ''
```

复制

这里使用了一个 `docker:dind` 的 service，并且由于 Docker 在新版本中可以启用 TLS 的支持，现在默认的 `docker:dind` 会自动生成证书，并对连接启用 TLS。所以在 variables 中配置了 `DOCKER_TLS_CERTDIR: ''` 以禁用 `docker:dind` 自动生成证书。

DIND 如何利用构建缓存

我们已经介绍过，使用 DIND 的话，在每次执行 pipeline 时，便会新创建一个容器，运行 Docker Daemon 在构建完成后便销毁它。 所以这样并不能很好地利用构建缓存。

所以我们可以将构建环节修改如下：

```
build:
  stage: build
  script:
    - docker login -u gitlab-ci-token -p $CI_JOB_TOKEN $CI_REGISTRY
    - docker pull $CI_REGISTRY/$CI_PROJECT_PATH:latest
    - docker build --cache-from $CI_REGISTRY/$CI_PROJECT_PATH:latest -t $CI_REGISTRY/$

    - docker push $CI_REGISTRY/$CI_PROJECT_PATH:$CI_COMMIT_SHORT_SHA
    - docker push $CI_REGISTRY/$CI_PROJECT_PATH:latest
    - docker logout $CI_REGISTRY
```

复制

每次构建一个 latest 的镜像 tag，并且在构建镜像的时候通过 `--cache-from` 使用它作为缓存进行镜像的构建。

总结

本篇，我为你介绍了通用型的 CI/CD pipeline，并对其中可能遇到的问题进行了介绍。并且通过一个实际的项目示例为你介绍了如何使用 GitLab CI pipeline。

当你在实际使用 CI/CD pipeline 的时候，除去以上介绍的内容外，还需要考虑更多的业务需求；或者是考虑自己实际的情况，比如部署平台或工具、如何平滑升级等。

希望本篇的内容能作为使用 GitLab CI 做 CI/CD pipeline 一个参考，下篇我们正式进入“架构篇”的学习。

下一章