将 Docker 用于 ...

本节和下一节这两节内容是特别增加的 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。本节主要会为你介绍几种将 Docker 应用于 CI/CD pipeline 中的方式，下一节会介绍完整的实践。

通过前面内容的学习，我们已经知道了 Docker 是 C/S 架构，我们平时使用的 `docker` 命令是它的 CLI，通过 API 与 Docker Daemon 进行交互，最终由 Docker Daemon 完成实际的工作。

如果想要将 Docker 用于 CI/CD pipeline 中，正常情况下 `docker build`（构建镜像）和 `docker push`（推送镜像）都是必不可少的。构建和分发等相关的流程和原理，在前面内容中都已介绍过，这里就不再赘述，我们将重点放在“怎么用”上面。

CLI

CLI 是必不可少的，它是与 Docker Daemon 交互的途径，但除了官方的 Docker CLI 外，Docker 官方提供了 Go SDK 和 Python SDK，此外社区中也提供了其他语言的 SDK 可自行使用。

甚至如果你不怕麻烦的话，curl 也可以作为一个选项，比如：

复制

```
# 创建一个容器，并指定 name 为 hello-curl
(MoeLove) ➜  ~ curl --unix-socket /var/run/docker.sock -H "Content-Type: applicat
{"Id":"92a2199124805a5dcd44e58c76dbf818d3407865c7ec9e142188b4ef1f7102b7","Warnings

# 启动容器
(MoeLove) ➜  ~ curl --unix-socket /var/run/docker.sock -X POST http:/v1.40/contai

# 等待容器停止
(MoeLove) ➜  ~ curl --unix-socket /var/run/docker.sock -X POST http:/v1.40/contai
{"Error":null,"StatusCode":0}

# 查看日志
(MoeLove) ➜  ~ curl --output - --unix-socket /var/run/docker.sock -X GET http:/v1
hello curl CLI
```

当然，一般情况下我们都会选择官方的 CLI，因为它功能完整且用户体验优良，也更符合我们的操作习惯。

Docker Daemon

选完 CLI 后，我们来看看 Docker Daemon 方面的选择。整体而言，有两个方向：

1. 使用现有 Docker Daemon。
2. 为每次构建创建新的 Docker Daemon。我们来对比下这两个方向的选择。

使用现有 Docker Daemon

Docker Daemon 启动时，可以通过 `--host` 参数（简写为 `-H`）来指定 Daemon Socket 连接的位置，比如 `--host=unix:///var/run/docker.sock` 表示使用一个 Unix Domain Socket；而 `--host=tcp://0.0.0.0:2376` 表示 Docker Daemon 监听在 2376 端口上。

物理机环境

如果 CI/CD pipeline 是在物理机环境中运行，那么直接给 Docker CLI 传递 `-H` 参数，或者通过 `DOCKER_HOST` 环境变量指向 Docker Daemon 监听的位置即可。

这种情况比较普通，跟我们平时自己手动做测试或日常使用基本一致。需要注意的是以下几点：

- 如果使用 Unix Domain Socket（默认位置在 `/var/run/docker.sock`）默认需要 root 或者 `docker` 组的权限。
- 如果是在同一台机器，建议优先使用 Unix Domain Socket。
- 如果使用 TCP 端口的方式，在不做任何设置时，可以直接通过 **HTTP** 的方式进行访问；如果是在内网或者可信任网络中还好，如果是在不可信网络中，这种方式就不够安全了。你可以通过启动 Docker Daemon 时增加 `--tlsverify` 及其他 TLS 相关的参数来启用 TLS 支持，也需要注意现在只支持 TLS 1.0 及以上版本。SSLv3 及以下版本的协议由于安全问题已不再支持。

容器环境

我们的 CI/CD pipeline 也可以在 Docker 容器中执行，这样做的好处在于，环境可以很好的隔离开，以及可以享受更多更灵活的资源配置，包括使用类似 Kubernetes 之类的容器编排系统等。

在这种情况下，也有两种小的区别：

- 如果 Docker Daemon 是监听了 TCP Socket 的话，那么在容器中只要指定 DOCKER_HOST 环境变量，或是给 Docker CLI 传递 `-H` 选项，指定 Docker Daemon 的地址即可；
- 如果 Docker Daemon 使用了 Unix Domain Socket，则可以将 Socket 文件（默认是 /var/run/docker.sock）挂载进容器环境的 /var/run/docker.sock（或其他位置，通过给 Docker CLI 传递 `-H` 选项累配置具体地址），这样在容器环境中就可以使用主机上原本运行的 Docker Daemon 了。

小结

无论 CI/CD pipeline 是以物理机方式或是以容器的方式执行，如果是要使用现有的 Docker Daemon 的话，最终无非就是寻找一种可以在 CI/CD pipeline 中访问 Docker Daemon 的方式。

通过 TCP 访问的方式，优点在于 Docker Daemon 可以与 CI/CD pipeline 不在同一个机器上，（可以做集群）分摊一些负载。或是将 Docker Daemon 部署在性能和网络都更好的机器上，专门用来完成构建和分发镜像等消耗资源的操作。

如果采用挂载 /var/run/docker.sock 的方式，优点在于简单，速度快。当然缺点也比较明显，就是负载都在同一台机器上，压力会比较大。同时，因为所有的 pipeline 都操作同一个 Docker Daemon 当然也有可能会有冲突的情况存在。

归根结底，使用现有 Docker Daemon 最大的缺陷就在于无法真正完成隔离，所以也就有了下面这种为每个 CI/CD pipeline 创建一个 Docker Daemon 的方案。

创建新的 Docker Daemon

我们可以使用 Docker In Docker（DIND）的方式来为每个 CI/CD pipeline 启动一个（组）Docker Daemon 的容器，这样每个 pipeline 使用的都将是一个完全隔离的 Docker Daemon。

复制

```
(MoeLove) ➜  ~ docker run --rm -d --privileged docker:dind
9765a2eacf953155ff2a2ebbf37f1c5978955277be15316ab2364e1ce6269b18
(MoeLove) ➜  ~ docker exec $(docker ps -ql) docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             ST
(MoeLove) ➜  ~ docker exec $(docker ps -ql) docker run alpine echo 'hello dind'
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
89d9c30c1d48: Pulling fs layer
89d9c30c1d48: Verifying Checksum
89d9c30c1d48: Download complete
89d9c30c1d48: Pull complete
Digest: sha256:c19173c5ada610a5989151111163d28a67368362762534d8a8121ce95cf2bd5a
Status: Downloaded newer image for alpine:latest
hello dind
```

需要注意的是，当前 DIND 模式需要一些特殊权限，所以我们在启动的时候需要为它传递一个 `--privileged` 的参数。

而 CI/CD pipeline 有以下几种办法可以使用它。

直接使用 TCP Socket

就像前面介绍的一样，我们先获取该 DIND 容器的 IP，然后给 Docker CLI 指定 `-H` 选项让它连接 DIND 中的 Docker Daemon。

复制

```
(MoeLove) ➜  ~ docker exec $(docker ps -ql) hostname -i
172.17.0.5
(MoeLove) ➜  ~ docker -H tcp://172.17.0.5:2376 ps
Error response from daemon: Client sent an HTTP request to an HTTPS server.
```

很明显上面的命令报错了，从报错信息也可以看出来这是说 Client 给 HTTPS 服务发送了 HTTP 请求。这也是我上面为你介绍的当 Docker Daemon 开启远程 TCP 连接时，建议开启 TLS 的支持。执行以下命令，将相关认证信息拷贝至本地：

复制

```
(MoeLove) ➜  ~ mkdir dind
(MoeLove) ➜  ~ docker cp $(docker ps -ql):/certs/client dind
(MoeLove) ➜  ~ docker -H tcp://172.17.0.5:2376 --tlsverify --tlscacert dind/c
CONTAINER ID        IMAGE               COMMAND                  CREATED
0b2bb7cae547        alpine              "echo 'hello dind'"      14 minutes ago
```

可以看到传递了相关的证书信息就可以正常使用了。

当然，如果在启动 DIND 容器的时候，你传递 `-e DOCKER_TLS_CERTDIR=''` 参数的话，就会禁用到 TLS 相关的功能，那你直接连接容器 IP:2375 端口即可访问。

总结

本篇，我为你介绍了将 Docker 用于 CI/CD pipeline 的几种方式，包括直接使用物理机上 Docker Daemon 的 TCP socket 连接，挂载 Docker Daemon 的 Unix Domain Socket 或者是创建 Docker In Docker 的容器等。

在实际的应用中，应当考虑自己的实际情况，以及在各种方案中进行权衡。虽然 Docker In Docker 可以为我们创建一个很好的隔离环境，但它生命周期一般都很短暂（一般与 pipeline 保持一致），在使用时由于每次都创建新的，它有时候不方便我们利用缓存之类的。

当然，你可能会好奇本节说将 Docker 用于 CI/CD pipeline 中，为何到现在也尚未看到具体的 pipeline 配置或者形式呢？我将这些内容放到了下一篇，本节我们就集中在“用 Docker”的部分。

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，用得比较多的方式就是 DIND。

下一节，我来为你介绍比较通用的生产环境在用的 CI/CD pipeline。