docker-proxy 的...

本篇是第七部分"网络篇"的第五篇。在这个部分，我会为你由浅入深的介绍 Docker 网络相关的内容。包括 Docker 网络基础及其实现和内部原理等。上篇，我为你介绍了 Docker 如何利用 iptables 为容器提供网络。本篇，我们深入了解下之前提到的 Docker 的一个组件 docker-proxy 的工作原理。

在之前的《Docker 核心架构及拆解（中）》我已经为你介绍过 docker-proxy 的基本应用。

这里我们稍作简单的复习：dockerd 在启动时提供了一个 `--userland-proxy` 的参数，用于控制是否启用 userland proxy 来处理回环（loopback）流量，并且可以通过 `--userland-proxy-path` 参数自定义 userland proxy 二进制文件的路径。

复制

```
(MoeLove) ➜  ~ dockerd --help |grep userland-proxy
      --userland-proxy                         Use userland proxy for loopback tr
      --userland-proxy-path string              Path to the userland proxy binary
```

默认使用的是 docker-proxy，它支持的参数如下：

复制

```
(MoeLove) ➜  ~ docker-proxy -h
Usage of docker-proxy:
  -container-ip string
        container ip
  -container-port int
        container port (default -1)
  -host-ip string
        host ip
  -host-port int
        host port (default -1)
  -proto string
        proxy protocol (default "tcp")
```

可以看到，它支持的参数代表着容器和主机的两端，以及所用的协议。

我们来深入 docker-proxy 的源码来看看它具体做了什么事情。

源码解析

docker-proxy 的源代码在 https://github.com/moby/libnetwork/tree/master/cmd/proxy目录中，Docker 在构建发布的时候，会一起构建并分发 docker-proxy 的二进制文件。

先来看看代码的目录结构吧：

```
(MoeLove) ➜  proxy git:(master) tree
.
├── main.go
├── network_proxy_test.go
├── proxy.go
├── sctp_proxy.go
├── stub_proxy.go
├── tcp_proxy.go
└── udp_proxy.go

0 directories, 7 files
```

main.go 是 docker-proxy 的入口文件，主要用来接收参数和接收中止信号的。

proxy.go 定义了一个 Proxy 的接口，其余的 *_proxy.go 文件中就是在实现该接口。以 tcp_proxy.go 为例，它的实现逻辑其实很简单：

```
    var wg sync.WaitGroup
    var broker = func(to, from *net.TCPConn) {
        io.Copy(to, from)
        from.CloseRead()
        to.CloseWrite()
        wg.Done()
    }

    wg.Add(2)
    go broker(client, backend)
    go broker(backend, client)
```

简单地将一端的流量直接拷贝到另一端，以实现其功能。接下来我们看看它的具体应用。

应用

还记得上一篇 《手动进行容器网络的管理》我们遗留的问题吗？

> 为什么当我们关闭 iptables 时，容器映射在主机的端口还可以正常对外提供服务？

我们先来恢复上一篇中的环境，使用 Docker In Docker 的方式，启动一个全新的环境，并传递 `--iptables=false` 参数。

```
(MoeLove) ➜  ~ docker run --rm -d --privileged docker:19.03.7-dind dockerd --ipta
fcaeb843fdf6964cfc2fe2834863ce0eee2ecb6b6355d3c3f32c59ec2d11f43d
```

进入该环境，启动一个容器并进行端口映射。

```
(MoeLove) ➜  ~ docker exec -it $(docker ps -ql) sh
/ # docker run --rm -d -p 3005:3005 taobeier/echo:node
Unable to find image 'taobeier/echo:node' locally
node: Pulling from taobeier/echo
e7c96db7181b: Pull complete
7b373bfb6ac5: Pull complete
fd38342e0337: Pull complete
5269cc77d334: Pull complete
526867dc7fdc: Pull complete
7d9b4277b71c: Pull complete
Digest: sha256:2c09919f86bf4f257da2454fa946d25c6ee173e4e37c0206885db6a12ca9e404
Status: Downloaded newer image for taobeier/echo:node
6d3af95fbc89823e80fa2d7a174aaea406183ab2fd1f4837dfded766fd3f0e19
/ # docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
6d3af95fbc89        taobeier/echo:node   "docker-entrypoint.s…"   19 seconds ago
```

请求 3005 端口，可以看到容器是可以正常提供服务的。

```
/ # wget -q -O - 127.0.0.1:3005
GET / HTTP/1.1
Host: 127.0.0.1:3005
User-Agent: Wget
Connection: close
```

iptables 中没有任何端口映射相关的规则。我们来查看现在端口监听的情况：

```
/ # netstat  -ntlp
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PI
tcp        0      0 :::3005                 :::*                    LISTEN      37
```

可以看到 3005 端口是由 docker-proxy 进程监听的。所以这也就回答了我们在上篇中的问题，**本地端口映射是可以直接通过 docker-proxy 完成的**，查看进程的相关信息，可以看到 docker-proxy 传递的参数的确是本地和容器的两端。通过这种方式来完成了本地端口映射的功

能。

```
/ # ps -ef |grep docker-proxy
  370 root      0:00 /usr/local/bin/docker-proxy -proto tcp -host-ip 0.0.0.0 -hos
  458 root      0:00 grep docker-proxy
```

总结

本篇，我为你介绍了 docker-proxy 的工作原理及其应用。可以看到，它本身逻辑并不复杂，但却为我们提供了非常重要的功能。这也符合了 KISS（Keep It Simple，Stupid）的设计理念。

下篇，我将为你介绍 Docker 内部 DNS 的原理，以此来了解 Docker 在容器互联中为我们提供的便利。