Docker 网络原理

本篇是第七部分“网络篇”的第八篇。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 内部 DNS 的原理，本篇，我们来总结下 Docker 网络原理。

Docker 网络相关的原理及实践，在前面我已经基本都为大家介绍到了。本篇，对这些内容做下总结，以及扩展。

我们知道 Docker 支持多种网络模式，通过 `docker info` 命令可进行查询：

复制

```
(MoeLove) ➜  ~ docker info --format "{{ .Plugins.Network }}"
[bridge host ipvlan macvlan null overlay]
```

它默认会创建三种单机版的网络：

复制

```
(MoeLove) ➜  ~ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
f88eb99ba426        bridge              bridge              local
e07891351b5e        host                host                local
46716de300ca        none                null                local
```

null 就不再多说了就是不提供对外网络，host 网络是与主机共享网络堆栈，bridge 之前内容中也详细介绍过了，默认是走 docker0 作为 bridge，如果是使用 `docker network create` 新建 bridge，则可以使用 Docker 提供的内部 DNS 等相关特性。

除了这三种外，单机模式下，还支持 container 容器网络，即将新创建的容器加入到已运行的容器所在的 Network Namespace，共享网络堆栈。例如：

复制

```
# 启动一个 Redis 容器
(MoeLove) ➜  ~ docker run --rm -d redis:alpine
583ed56f82806e30a78dbc7b08fe1b9cca096bb5ad72a7808a1b9f5897aaa14e

# 启动另一个容器，并加入刚才创建的容器的网络
(MoeLove) ➜  ~ docker run -it --rm --network container:$(docker ps -ql) redis:alp
# 查看当前运行中的进程，发现并不存在 redis-server
/data # ps -ef
PID   USER     TIME  COMMAND
    1 root      0:00 sh
    8 root      0:00 ps -ef


# 执行 redis-cli ping 可以得到正常的响应
/data # redis-cli ping
PONG

### 查询当前在监听的端口，可以看到 6379 端口正在被监听
/data # netstat -tpl
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PI
tcp        0      0 0.0.0.0:6379            0.0.0.0:*               LISTEN      -
tcp        0      0 :::6379                 :::*                    LISTEN      -
```

端口映射

当我们在使用创建或启动容器时，使用了 `-p` 或者 `-P` 参数，那么 Docker 会使用 iptables 和 docker-proxy 完成端口映射，并让我们的数据包可以正确的传输。

在前面内容中，我也已经介绍过我们可以通过自己手动设置 iptables 规则来管理容器的网络。但，如非必要，我还是建议你使用 Docker 来管理容器的 iptables 相关规则。

这样在你遇到问题的时，可以先免除你自己设置 iptables 所造成的影响。当然，如果你所在的环境对网络要求比较严格，那自己去管理 iptables 规则也可以的。需要注意的是，设置 iptables 规则请采用精简模式，即： **非必要的不配置，没搞清楚的不增加。**

此外，对于 Docker 自带的 docker-proxy，如果你想要调试，或者排查容器网络中 4 层或者 7 层问题的时，你可以使用 socat 进行辅助。

socat 可以完成类似 docker-proxy 的功能，连接两端。但同时它可以将其中传输的数据包输出，便于排查问题。

内部 DNS

Docker 提供了自定义网络，可用于单机容器方便的组网，以及通过容器名称或者使用 `--network-alias` 设置别名进行名称解析。

此外，Docker 也有一个已经过期的用于容器网络连接的参数，即 `--link`。但此特性已经被置为废弃，请尽可能不要使用或依赖此功能。

如果在用到 Docker 内置 DNS 的时候，在 Docker v19.03.8 及之前版本都有可能会因 DNS server 解析时偶发的 Bug，导致 docker daemon 异常退出。详细情况可查看：

> Docker 仓库中对应的 issue

我将此问题的修复已合并至 Docker 的下个大版本（v20.03），目前也正在移植至 v19.03+ 版本中。

veth-pair

你是否还记得在《自己动手写容器（下）》那一篇中，我曾为你介绍过 veth-pair?

事实上当你创建了新的 network，并使用此 network 启动新的容器时，Docker 会在同时创建一个 veth-pair，通过 `ip a` 查看其结果类似下面这样：

```
复制
3: br-c5faf46172f7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
    link/ether 02:42:a8:17:83:b8 brd ff:ff:ff:ff:ff:ff
    inet 172.19.0.1/16 brd 172.19.255.255 scope global br-c5faf46172f7
       valid_lft forever preferred_lft forever
5: veth10bbd40@if4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue maste
    link/ether 86:9c:e1:5c:9f:78 brd ff:ff:ff:ff:ff:ff link-netnsid 1
```

在容器内也可以看到对应的网卡信息（注意它们的标号）：

```
复制
# ip link show eth0
4: eth0@if5: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state
    link/ether 02:42:ac:13:00:02 brd ff:ff:ff:ff:ff:ff
```

通过这种方式以实现网络互通。关于 veth-pair 的应用其实很广泛，包括 Kubernetes 的网络等也都有使用到。

总结

本篇，我为你总结了“网络篇”容器网络原理相关的内容，并重新联系到了之前介绍过的 `veth-pair` 相关的内容。

Docker 网络涉及的内容很多，尤其是会涉及到很多关于 iptables 相关的内容。

Docker 为我们提供了很多网络方面的特性，在使用的时候，可根据需求灵活使用，选择最适合自己的方式。

下一篇，我们将进入本专栏的最后一个部分“生态篇”，带你了解 Docker 容器生态相关的其他扩展技术，包括 containerd、runc、Kubernetes 等，以及会介绍如何参与到 Docker 上游项目中，最后会分享 Docker 未来的走向。