手动进行容器网...

本篇是第七部分“网络篇”的第四篇。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 与 iptables 之间的联系。本篇，我们来深入地了解下 Docker 到底是如何使用 iptables 为容器网络提供各类特性的。

在上篇，我们了解到 Docker 在启动容器时，会利用 iptables 创建一些规则，以达到对容器网络的控制。

本篇，我们通过介绍如何手动进行容器网络的管理，以便于对 Docker 容器网络管理有更深刻的认识。同样的为了避免环境的影响，本篇还是使用 Docker in Docker 的方式作为基础环境。

注意：本篇为了演示手动管理容器网络，所以单独传递了 `--iptables=false` 的参数。

复制

```
(MoeLove) ➜  ~ docker run --rm -d --privileged docker:19.03.7-dind dockerd --ipta
c1f4fae571bd6967e1ee7f7037c6fa33240ca9cf02b07b00eeae7de36294b072
(MoeLove) ➜  ~ docker exec -it $(docker ps -ql) sh
/ # iptables-save
# Generated by iptables-save v1.8.3 on Wed Mar 11 11:57:23 2020
*filter
:INPUT ACCEPT [0:0]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [1:40]
:DOCKER-USER - [0:0]
-A FORWARD -j DOCKER-USER
-A DOCKER-USER -j RETURN
COMMIT
# Completed on Wed Mar 11 11:57:23 2020
# Generated by iptables-save v1.8.3 on Wed Mar 11 11:57:23 2020
*nat
:PREROUTING ACCEPT [0:0]
:INPUT ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
:POSTROUTING ACCEPT [0:0]
COMMIT
# Completed on Wed Mar 11 11:57:23 2020
```

检查 bridge 网络的信息

在启动 Docker Daemon 之后，会自动创建 docker0 的 bridge，这里我们可以看到它的 IP 段是 172.18.0.1/16。

复制

```
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
2: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN
    link/ether 02:42:15:5e:2d:55 brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.1/16 brd 172.18.255.255 scope global docker0
       valid_lft forever preferred_lft forever
34: eth0@if35: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue sta
    link/ether 02:42:ac:11:00:04 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.4/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
```

启动测试容器

这里我专门准备了一个简单的 HTTP echo server 镜像，其中的程序代码是 https://github.com/watson/http-echo-server，当你请求它的时候，它可以返回你的请求信息。

使用这个镜像启动容器时，容器内程序默认监听 3005 端口，我们通过以下命令来启动该容器：

复制

```
/ # 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
Digest: sha256:bee64944d594162672f2650c48353de0e5733f7cb09e77cf6a308e8fe9870592
Status: Downloaded newer image for taobeier/echo:node
a50f73c72cbe8923dc84c354885947d32dba3e379bf95f6be8aede5da5689f60
```

然后验证该容器的状态：

复制

```
/ # docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
a50f73c72cbe        taobeier/echo:node   "docker-entrypoint.s…"   46 seconds ago
```

可以看到该容器已经正常运行，并且将容器内的 3005 端口映射到了本地的 3005 端口。我们来尝试进行访问下。

这里为了便于区分，我们修改下本地的命令行提示符，并且为了方便验证，我们安装一下 curl 工具。（当然，通过 nc 命令之类的进行测试也没问题的）

复制

```
# 修改命令行提示符
/ # export PS1="➜ "
# 安装 curl 工具
 ➜ apk add --no-cache -q curl
# 使用 curl 命令进行测试
 ➜ curl 127.0.0.1:3005
GET / HTTP/1.1
Host: 127.0.0.1:3005
User-Agent: curl/7.67.0
Accept: */*
```

可以看到通过映射的端口访问该容器提供的服务是正常的。

这里你也许会好奇，我们不是已经关闭 iptables 了吗，怎么还能正常访问？此处暂且不谈，我们下篇来聊这个问题。

验证容器内网络

我们进入该测试容器内，尝试访问容器内的服务：

复制

```
 ➜ docker exec -it $(docker ps -ql) sh
# 验证服务正在运行
/ # ps -ef
PID   USER     TIME  COMMAND
    1 root      0:00 node /usr/local/bin/http-echo-server 3005
   13 root      0:00 sh
   18 root      0:00 ps -ef
# 使用 curl 验证服务，发现 curl 尚未安装
/ # curl 127.0.0.1:3005
sh: curl: not found
```

curl 在容器内尚未安装，那我们来尝试安装 curl：

复制

```
/ # apk add --no-cache curl
fetch http://dl-cdn.alpinelinux.org/alpine/v3.9/main/x86_64/APKINDEX.tar.gz
WARNING: Ignoring http://dl-cdn.alpinelinux.org/alpine/v3.9/main/x86_64/APKINDEX.t
fetch http://dl-cdn.alpinelinux.org/alpine/v3.9/community/x86_64/APKINDEX.tar.gz
WARNING: Ignoring http://dl-cdn.alpinelinux.org/alpine/v3.9/community/x86_64/APKIN
ERROR: unsatisfiable constraints:
  curl (missing):
    required by: world[curl]
```

但是发现安装失败，看报错信息应该是网络问题。我们暂时先不管它，先使用 nc 命令来验证下我们的服务：

复制

```
/ # nc 127.0.0.1 3005
Hello
HTTP/1.1 200 OK
Date: Wed Mar 11 2020 17:19:55 GMT+0000 (Coordinated Universal Time)
Connection: close
Content-Type: text/plain
Access-Control-Allow-Origin: *

Hello
^Cpunt!
```

发现使用 nc 验证服务，服务状态都正常。

我们回过头来看看，为何通过 apk 命令安装不了 curl。我们首先想到这可能是个网络问题。

调试容器内网络

测试是否能访问公网

最直接的那就先用 ping 命令试试看吧。发现域名无法解析，换成 ping 公网 IP 也没有响应。

复制

```
/ # ping -c 1 moelove.info
ping: bad address 'moelove.info'
/ # ping -c 1 1.1.1.1
PING 1.1.1.1 (1.1.1.1): 56 data bytes

--- 1.1.1.1 ping statistics ---
1 packets transmitted, 0 packets received, 100% packet loss
```

抓包定位问题

在容器内使用 tcpdump 进行抓包，排查当前的网络问题：

```
/ # tcpdump  -i eth0  -nn
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
18:18:17.092560 IP 172.18.0.2 > 1.1.1.1: ICMP echo request, id 13824, seq 0, leng
18:18:22.447064 ARP, Request who-has 172.18.0.1 tell 172.18.0.2, length 28
18:18:22.447198 ARP, Reply 172.18.0.1 is-at 02:42:15:5e:2d:55, length 28
```

上面是我在容器内执行 `ping -c 1 1.1.1.1` 时的抓包结果，可以看到这里只有发出的包（request），但是没有回包（reply）。

我们思考下为何会有这样的情况出现呢?

容器内数据包通过 eth0 出去，会到达网关 docker0，之后到达其真正的外网网卡访问到外部。接收到响应后，同样是需要原路返回的。

这里就有个问题了，容器内的 IP 地址对于外部来说其实是不可见的。访问外网的话，也是也先通过 docker0 再通过 eth0 网卡才能出去。

上一篇，我们介绍过 Docker 会使用 iptables 的 NAT，那我们此处不妨也进行尝试。

尝试修正网络问题

由于只涉及的是地址转换，那肯定选择 nat 表进行操作了，其中可用的链有很多，但通常类似这样的需求，我们选择 POSTROUTING 来操作，因为 POSTROUTING 是流量离开之前的最后一个修改信息的机会了。

再来就是我们需要指定下包的来源，现在容器的网段是 172.18.0.0/16，并且刚才也说过了，容器中包要访问到外网，那必须不是从 docker0 流出的，我们来为它做 SNAT 的配置。

翻译过来的 iptables 规则也就是：

```
➜ iptables -t nat -A POSTROUTING -s 172.18.0.0/16 ! -o docker0 -j SNAT --to-sourc
```

现在我们将这条规则应用到主机上，并在容器内再次尝试 ping 一下外网地址。

复制

```
/ # ping -c 1 1.1.1.1
PING 1.1.1.1 (1.1.1.1): 56 data bytes
64 bytes from 1.1.1.1: seq=0 ttl=50 time=188.451 ms

--- 1.1.1.1 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 188.451/188.451/188.451 ms
```

至此，我们就手动用 iptables 完成了一个容器访问外部网络的需求。

总结

本篇，我为你介绍了如何使用 iptables 完成容器网络的控制。这里只是介绍了一种场景，而 Docker 为我们自动生成的 iptables 规则其实涵盖了很多其他的方面，并且它也通过创建自定义链的方式，以避免污染其他规则。

关于 Docker 和 iptables 可挖掘的内容还有很多，如果对网络感兴趣的读者，建议继续深入研究。

同时，在本文中我留下了一个问题：“当我们关闭 iptables 时，为何容器映射在主机的端口还可以正常对外提供服务？”下一篇，我们将来回答这个问题，并更加深入的理解另一个 Docker 重要的组件。