揭秘 Docker 网络：手动实现 Docker 桥接网络

探索云原生

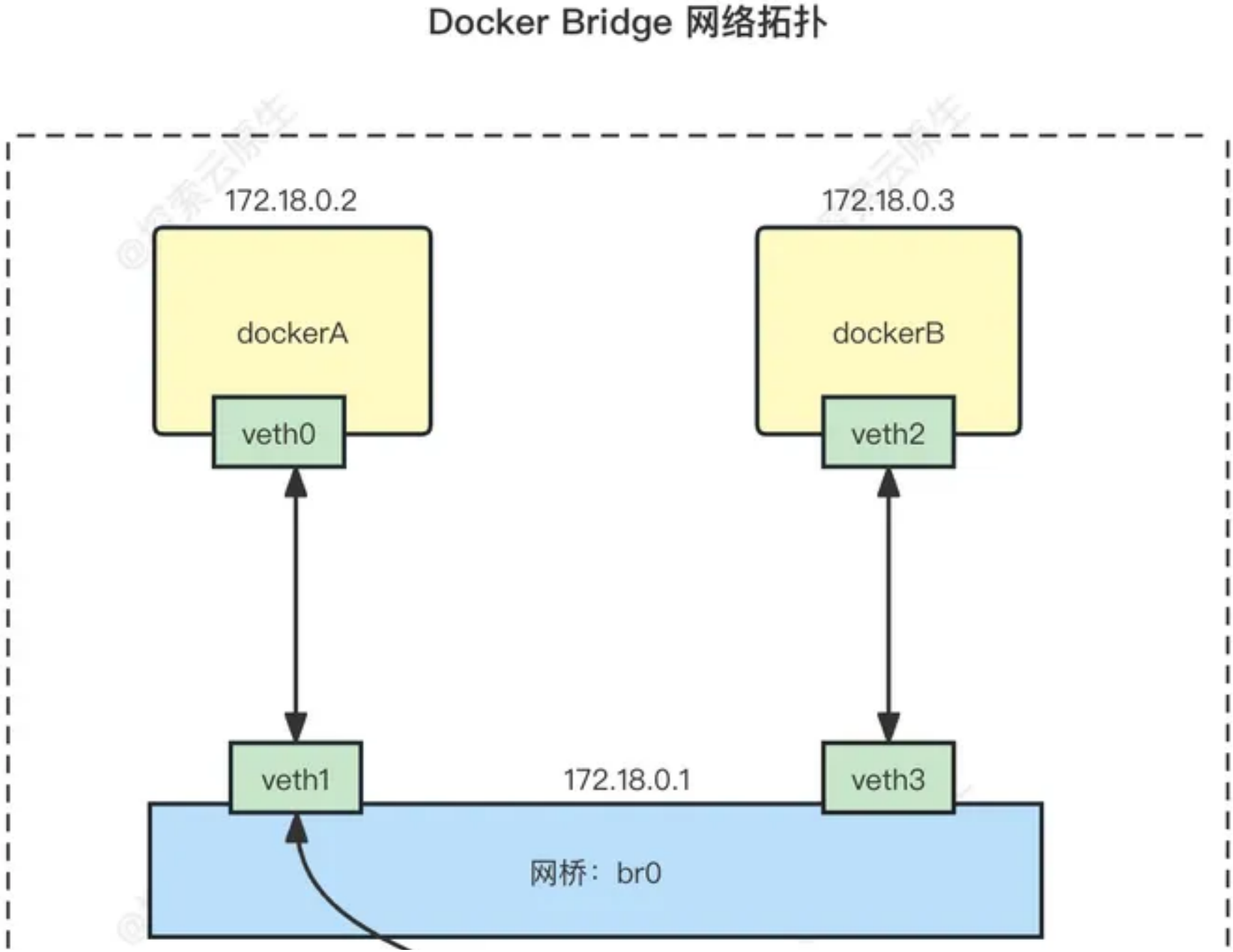
本文将带领读者探索 Docker 桥接网络模型的内部机制，通过 veth pair、bridge、iptables 等关键技术手动实现 Docker 桥接网络模型，揭示网络背后的运作原理。

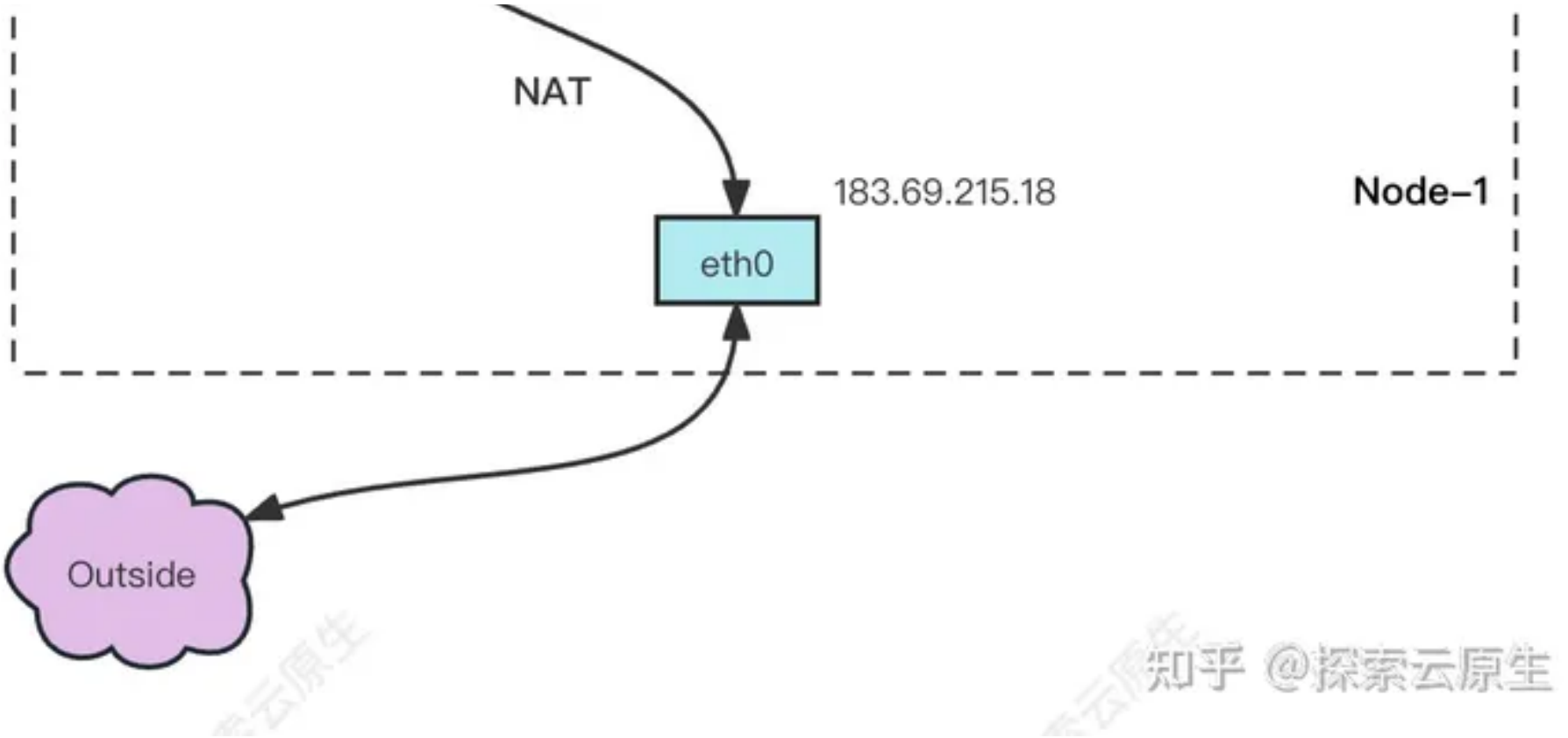
1. 概述

Docker 有多种网络模型，对于单机上运行的多个容器，可以使用缺省的 bridge 网络驱动。

我们按照下图创建网络拓扑，让容器之间网络互通，从容器内部可以访问外部资源，同时，容器内可以暴露服务让外部访问。

桥接网络的一个拓扑结构如下：


Docker Bridge 网络拓扑
172.18.0.2
172.18.0.3
dockerA
dockerB
veth0
veth2
veth1
172.18.0.1
veth3
网桥：br0



Docker Bridge 网络拓扑

上述网络拓扑实现了：

- 1）让容器之间网络互通
- 2）从容器内部可以访问外部资源
- 3）容器内可以暴露服务让外部访问。

根据网络拓扑图可以看到，容器内的数据通过 veth pair 设备传递到宿主机上的网桥上，最终通过宿主机的 eth0 网卡发送出去（或者再通过 veth pair 进入到另外的容器），而接收数据的流程则恰好相反。

2. 预备知识

这里对本文会用到的相关网络知识做一个简单介绍。

veth pair

> 相关笔记： veth-pair

`Veth` 是成对出现的两张虚拟网卡，从一端发送的数据包，总会在另一端接收到。

利用 `Veth` 的特性，我们可以将一端的虚拟网卡"放入"容器内，另一端接入虚拟交换机。这样，接入同一个虚拟交换机的容器之间就实现了网络互通。

> 即：通过 veth 来突破 network namespace 的封锁

bridge

> 相关笔记：Linux bridge

我们可以认为 `Linux bridge` 就是**虚拟交换机**，连接在同一个 `bridge` 上的容器组成局域网，不同的 `bridge` 之间网络是隔离的。

> `docker network create [NETWORK NAME]` 实际上就是创建出虚拟交换机。

交换机是工作在数据链路层的网络设备，它转发的是二层网络包。最简单的转发策略是将到达交换机输入端口的报文，广播到所有的输出端口。当然更好的策略是在转发过程中进行学习，记录交换机端口和MAC地址的映射关系，这样在下次转发时就能够根据报文中的MAC地址，发送到对应的输出端口。

NAT

过程中需要使用 NAT 技术，修改源地址或者目的地址，一般使用 iptables 来实现。

> 相关笔记：iptables

NAT（Network Address Translation），是指网络地址转换。

因为容器中的IP和宿主机的IP是不一样的，为了保证发出去的数据包能正常回来，需要对IP层的源IP/目的IP进行转换。

- SNAT：源地址转换
- DNAT：目的地址转换

SNAT

Source Network Address Translation，源地址转换，用于修改数据包中的源地址。

比如上图中的 eth0 ip 是 `183.69.215.18`，而容器 dockerA 的 IP 却是 `172.187.0.2`。

因此容器中发出来的数据包，`源IP` 肯定是 `172.187.0.2`，如果就这样不处理直接发出去，那么接收方处理后发回来的响应数据包的 `目的IP` 自然就会填成 `172.187.0.2`，那么我们肯定接收不到这个响应了。

因此在将容器中的数据包通过 eth0 网卡发送出去之前，需要进行 SNAT 把源 ip 改为 eth0 的 ip，也就是 `183.69.215.18`。

这样接收方响应时将源 IP `183.69.215.18` 作为目的 IP，这样我们才能收到返回的数据。

DNAT

Destination Network Address Translation：目的地址转换，用于修改数据包中的目的地址。

如果发出去做了 SNAT，源IP改成了宿主机的 `183.69.215.18`，那么回来的响应数据包目的IP自然就是 `183.69.215.18`，我们(宿主机)可以成功收到这个响应。

但是如果直接把源IP是 `183.69.215.18` 的数据包发到容器里面去，由于容器IP是 `172.187.0.2` ,那肯定不会处理这个包，所以宿主机收到响应包需要进行 DNAT，将目的 IP 地址从 `183.69.215.18` 改成容器中的 `172.187.0.2` 。

这样容器才能正常处理该数据。

3. 演示

实验环境 Ubuntu 20.04

环境准备

首先需要创建对应的容器，veth pair 设备以及 bridge 设备 并分配对应 IP。

创建“容器”

从前面的背景知识（深入理解 Docker 核心原理：Namespace、Cgroups 和 Rootfs）了解到，容器的本质是 `Namespace + Cgroups + rootfs` 。因此本实验我们可以仅仅创建出 `Namespace` 网络隔离环境来模拟容器行为：

```
$ sudo ip netns add ns1
$ sudo ip netns add ns2
$ sudo ip netns show
ns2
ns1
```

创建 Veth pairs

```
sudo ip link add veth0 type veth peer name veth1
sudo ip link add veth2 type veth peer name veth3
```

查看一下：

```
$ ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: ens3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT group default qlen 1000
    link/ether fa:16:3e:9b:9b:33 brd ff:ff:ff:ff:ff:ff
3: veth1@veth0: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 9a:45:4c:f9:77:eb brd ff:ff:ff:ff:ff:ff
4: veth0@veth1: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether fe:5a:a1:3b:94:9b brd ff:ff:ff:ff:ff:ff
5: veth3@veth2: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 96:d2:e4:ea:9a:1d brd ff:ff:ff:ff:ff:ff
6: veth2@veth3: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
```

将 Veth 的一端放入“容器”

将 veth 的一端移动到对应的 `Namespace` 就相当于把这张网卡加入到’容器‘里了。

```
sudo ip link set veth0 netns ns1
sudo ip link set veth2 netns ns2
```

查看宿主机上的网卡

```
$ ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: ens3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT group default qlen 1000
    link/ether fa:16:3e:9b:9b:33 brd ff:ff:ff:ff:ff:ff
3: veth1@if4: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 9a:45:4c:f9:77:eb brd ff:ff:ff:ff:ff:ff link-netns ns1
5: veth3@if6: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 96:d2:e4:ea:9a:1d brd ff:ff:ff:ff:ff:ff link-netns ns2
```

发现少了两个，然后进入容器对应 `Namespace` 查看一下容器中的网卡：

```
$ sudo ip netns exec ns1 ip link show
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
4: veth0@if3: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether fe:5a:a1:3b:94:9b brd ff:ff:ff:ff:ff:ff link-netnsid 0
$ sudo ip netns exec ns2 ip link show
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
6: veth2@if5: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 8e:6a:e4:a0:50:ce brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

可以看到， `veth0` 和 `veth2` 确实已经放到“容器”里去了。

创建bridge

一般使用 `brctl` 进行管理，不是自带的工具，需要先安装一下：

```
sudo apt-get install bridge-utils
```

创建bridge `br0`：

```
sudo brctl addbr br0
```

- **将Veth的另一端接入bridge**

```
sudo brctl addif br0 veth1
sudo brctl addif br0 veth3
```

查看接入效果：

```
$ sudo brctl show
bridge name     bridge id               STP enabled     interfaces
br0             8000.361580fa3c8b       no              veth1
                                                        veth3
```

可以看到，两个网卡 `veth1` 和 `veth3` 已经“插”在 `bridge` 上。

至此，veth pair 已经一端在容器里，一端在宿主机网桥上了，大致拓扑结构完成。

分配IP并启动

- **为bridge分配IP地址，激活上线**

```
sudo ip addr add 172.18.0.1/24 dev br0
sudo ip link set br0 up
```

- **为"容器“内的网卡分配IP地址，并激活上线**

docker0容器：

```
sudo ip netns exec ns1 ip addr add 172.18.0.2/24 dev veth0
sudo ip netns exec ns1 ip link set veth0 up
```

docker1容器：

```
sudo ip netns exec ns2 ip addr add 172.18.0.3/24 dev veth2
sudo ip netns exec ns2 ip link set veth2 up
```

- **Veth另一端的网卡激活上线**

```
sudo ip link set veth1 up
sudo ip link set veth3 up
```

至此，整个拓扑结构搭建完成，且所有设备都分配好 ip 并上线。

测试

容器互通

测试从容器 `docker0` ping 容器 `docker1`，测试之前先用 tcpdump 抓包，等会好分析：

```
sudo tcpdump -i br0 -n
```

在新窗口执行 ping 命令：

```
sudo ip netns exec ns1 ping -c 3 172.18.0.3
```

`br0` 上的抓包数据如下：

```
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on br0, link-type EN10MB (Ethernet), capture size 262144 bytes
12:35:18.285705 ARP, Request who-has 172.18.0.3 tell 172.18.0.2, length 28
12:35:18.285903 ARP, Reply 172.18.0.3 is-at e2:31:15:64:bd:39, length 28
12:35:18.285908 IP 172.18.0.2 > 172.18.0.3: ICMP echo request, id 13829, seq 1, length 64
12:35:18.286034 IP 172.18.0.3 > 172.18.0.2: ICMP echo reply, id 13829, seq 1, length 64
12:35:19.309392 IP 172.18.0.2 > 172.18.0.3: ICMP echo request, id 13829, seq 2, length 64
12:35:19.309589 IP 172.18.0.3 > 172.18.0.2: ICMP echo reply, id 13829, seq 2, length 64
12:35:20.349350 IP 172.18.0.2 > 172.18.0.3: ICMP echo request, id 13829, seq 3, length 64
12:35:20.349393 IP 172.18.0.3 > 172.18.0.2: ICMP echo reply, id 13829, seq 3, length 64
12:35:23.309404 ARP, Request who-has 172.18.0.2 tell 172.18.0.3, length 28
12:35:23.309517 ARP, Reply 172.18.0.2 is-at 2e:93:7e:33:b0:ed, length 28
```

可以看到，先是 `172.18.0.2` 发起的 `ARP` 请求，询问 `172.18.0.3` 的 `MAC` 地址，然后是 `ICMP` 的请求和响应，最后是 `172.18.0.3` 的ARP请求。

因为接在同一个bridge `br0` 上，所以是二层互通的局域网。

同样，从容器 `docker1` `ping` 容器 `docker0` 也是通的：

```
sudo ip netns exec ns2 ping -c 3 172.18.0.2
```

宿主机访问容器

在“容器” `docker0` 内启动服务，监听80端口：

```
sudo ip netns exec ns1 nc -lp 80
```

在宿主机上执行telnet，可以连接到 `docker0` 的80端口：

```
$ telnet 172.18.0.2 80
Trying 172.18.0.2...
Connected to 172.18.0.2.
Escape character is '^]'.
```

可以联通。

容器访问外网

这部分稍微复杂一些，需要配置 NAT 规则。

1）配置容器内路由

需要配置容器内的路由，这样才能把网络包从容器内转发出来。

具体就是：**将bridge设置为“容器”的缺省网关。**让非 `172.18.0.0/24` 网段的数据包都路由给 `bridge`，这样数据就从“容器”跑到宿主机上来了。

```
sudo ip netns exec ns1 ip route add default via 172.18.0.1 dev veth0
sudo ip netns exec ns2 ip route add default via 172.18.0.1 dev veth2
```

查看“容器”中的路由规则

```
$ sudo ip netns exec ns1 ip route
default via 172.18.0.1 dev veth0
172.18.0.0/24 dev veth0 proto kernel scope link src 172.18.0.2
```

可以看到，非 172.18.0.0 网段的数据都会走默认规则，也就是发送给网关 172.18.0.1。

2）宿主机开启转发功能并配置转发规则

在宿主机上配置内核参数，允许IP forwarding，这样才能把网络包转发出去。

```
sudo sysctl net.ipv4.conf.all.forwarding=1
```

还有就是要配置 iptables FORWARD 规则

首先确认 `iptables FORWARD` 的缺省策略：

```
$ sudo iptables -t filter -L FORWARD
Chain FORWARD (policy ACCEPT)
```

```
target     prot opt source               destination
```

一般都是 ACCEPT，如果如果缺省策略是 `DROP`，需要设置为 `ACCEPT`：

```
sudo iptables -t filter -P FORWARD ACCEPT
```

3）宿主机配置 SNAT 规则

```
sudo iptables -t nat -A POSTROUTING -s 172.18.0.0/24 ! -o br0 -j MASQUERADE
```

上面的命令的含义是：在 `nat` 表的 `POSTROUTING` 链增加规则，当数据包的源地址为 `172.18.0.0/24` 网段，出口设备不是 `br0` 时，就执行 `MASQUERADE` 动作。

`MASQUERADE` 也是一种源地址转换动作，它会动态选择宿主机的一个IP做源地址转换，而 `SNAT` 动作必须在命令中指定固定的IP地址。

测试能否访问外网：

```
$ sudo ip netns exec ns1 ping -c 3 114.114.114.114
PING 114.114.114.114 (114.114.114.114) 56(84) bytes of data.
64 bytes from 114.114.114.114: icmp_seq=1 ttl=80 time=21.1 ms
64 bytes from 114.114.114.114: icmp_seq=2 ttl=89 time=19.5 ms
64 bytes from 114.114.114.114: icmp_seq=3 ttl=86 time=19.2 ms
```

外部访问容器

外部访问容器需要进行 DNAT，把目的IP地址从宿主机地址转换成容器地址。

```
sudo iptables -t nat -A PREROUTING  ! -i br0 -p tcp -m tcp --dport 80 -j DNAT --to-destination 172.18.0.2:80
```

上面命令的含义是：在 `nat` 表的 `PREROUTING` 链增加规则，当输入设备不是 `br0` ，目的端口为80 时，做目的地址转换，将宿主机IP替换为容器IP。

测试一下

在“容器”docker0内启动服务：

```
sudo ip netns exec ns1 nc -lp 80
```

在**和宿主机同一个局域网的远程主机**访问宿主机 IP:80

```
telnet 192.168.2.110 80
```

确认可以访问到容器内启动的服务。

不过由于只在 `PREROUTING` 链上做了 DNAT，因此直接在宿主机上访问是不行，需要本机访问的话可以添加下面这个 iptables 规则，直接在 OUTPUT 链上增加 DNAT 规则：

> 这样其他节点来的流量和本机直接访问流量都可以正常 DNAT 了。

```
sudo iptables -t nat -A OUTPUT -p tcp -m tcp --dport 80 -j DNAT --to-destination 172.18.0.2:80
```

添加后再本机直接测试：

```
telnet 192.168.2.110 80
```

这下可以成功连上了。

环境恢复

删除虚拟网络设备

```
sudo ip link set br0 down
sudo brctl delbr br0
sudo ip link  del veth1
sudo ip link  del veth3
```

`iptablers` 和 `Namesapce` 的配置在机器重启后被清除。

4. 小结

本文主要通过 Linux 提供的各种虚拟设备以及 iptables 模拟出了 Docker bridge 网络模型，并测试了几种场景的网络互通。

> 实际上 `docker network` 就是使用了 `veth` 、 `Linux bridge` 、 `iptables` 等技术，帮我们创建和维护网络。

具体分析一下：

- 首先 docker 就是一个进程，主要利用 Linux Namespace 进行隔离。
- 为了跨 Namespace 通信，就用到了 Veth pair。

- 然后多个容器都使用 Veth pair 互相连通的话，不好管理，所以加入了 Linux Bridge，所有 veth 一端在容器中，一端直接和 bridge 连接，这样就好管理多了。
- 然后容器和外部网络要进行通信，于是又要用到 iptables 的 NAT 规则进行地址转换。

5. 参考

iptables 笔记

veth-pair 笔记

Docker bridge networks

Docker单机网络模型动手实验

发布于 2024-01-30 12:28 · IP 属地重庆

Docker Linux