

[Tony Bai](#)

一个程序员的心路历程

- [关于我](#)
- [文章列表](#)

再谈Docker容器单机网络：利用iptables trace和ebtables log

- 十一月 6, 2017
- [0 条评论](#)

这大半年一直在搞Kubernetes。每次[搭建Kubernetes集群](#)，或多或少都会被Kubernetes的“[网络插件们](#)”折腾折腾。因此，要说目前Kubernetes中最难搞的是什么？个人觉得莫过于其Pod网络了，至少也是最难搞的之一。除此之外，以[Service](#)和Pod为中心的Kubernetes架构还大量利用[iptables规则](#)来实现Service的反向代理和负载均衡，这又与[Docker原生容器单机网络](#)实现所基于的[linux bridge](#)和[iptables规则](#)糅合在一起，让troubleshooting时的难度又增加了一些。

去年曾经花过一段研究[Docker网络](#)，但现在看来当时在某些关键环节的理解上还有些模糊，于是花了周末的闲暇时间对Docker容器单机网络做了一次再理解。这次重新认识利用上了iptables的Trace功能以及数据链路层的ebtables，让我可以更清晰地看到单机容器网络的网络数据流向。同时，有了容器网络理解这个基础，对后续解决K8s Pod网络问题也是大有裨益的。

本文从某个角度来说也可以理解为自我答疑，我不会从最最基础的[Docker](#)网络结构说起，对Docker容器单机网络结构不了解的童鞋，可以先看看我之前写的《[理解Docker单机容器网络](#)》和《[理解Docker容器网络之Linux Network Namespace](#)》两篇文章。

一、实验环境

1、主机环境和工具版本

Docker的默认单机容器网络从最初的版本开始就几乎没有变过，因此理论上下面的分析适用于Docker的大部分版本。我的实验环境如下：

```
Ubuntu 16.04.3 LTS (GNU/Linux 4.4.0-63-generic x86_64)
```

```
# docker version
Client:
 Version:      17.09.0-ce
 API version:  1.32
 Go version:   go1.8.3
 Git commit:   afd66d4
 Built:        Tue Sep 26 22:42:18 2017
 OS/Arch:      linux/amd64

Server:
 Version:      17.09.0-ce
 API version:  1.32 (minimum version 1.12)
 Go version:   go1.8.3
 Git commit:   afd66d4
 Built:        Tue Sep 26 22:40:56 2017
 OS/Arch:      linux/amd64
 Experimental: false

# iptables --version
iptables v1.6.0
# ebtables --version
ebtables v2.0.10-4 (December 2011)
```

2、容器网络及拓扑

我们需要制作一个用于实验的容器镜像。因为这里仅用ping包进行测试，这里我们仅基于ubuntu:14.04 base image制作一个简单的安装有必要网络工具的image：

```
//Dockerfile
```

```
From ubuntu:14.04
RUN apt-get update && apt-get install -y curl iptables
ENTRYPOINT ["tail", "-f", "/var/log/bootstrap.log"]
```

```
// 制作镜像:
```

```
# docker build -t foo:latest ./
```

启动两个容器:

```
# docker run --name c1 -d --cap-add=NET_ADMIN foo:latest
7a01a19d9328b39f094c9a9c76340d179baaf93afb52189816bcc79f8319cb64
# docker run --name c2 -d --cap-add=NET_ADMIN foo:latest
94a2f1841f6d95fd0682299b17c0aedb60c1047786c8e75b0f1ab7316a995409
```

容器启动后的网络信息汇总如下:

```
# ifconfig -a
docker0    Link encap:Ethernet  HWaddr 02:42:ff:27:17:4d
           inet addr:192.168.0.1  Bcast:0.0.0.0  Mask:255.255.240.0
           ... ...

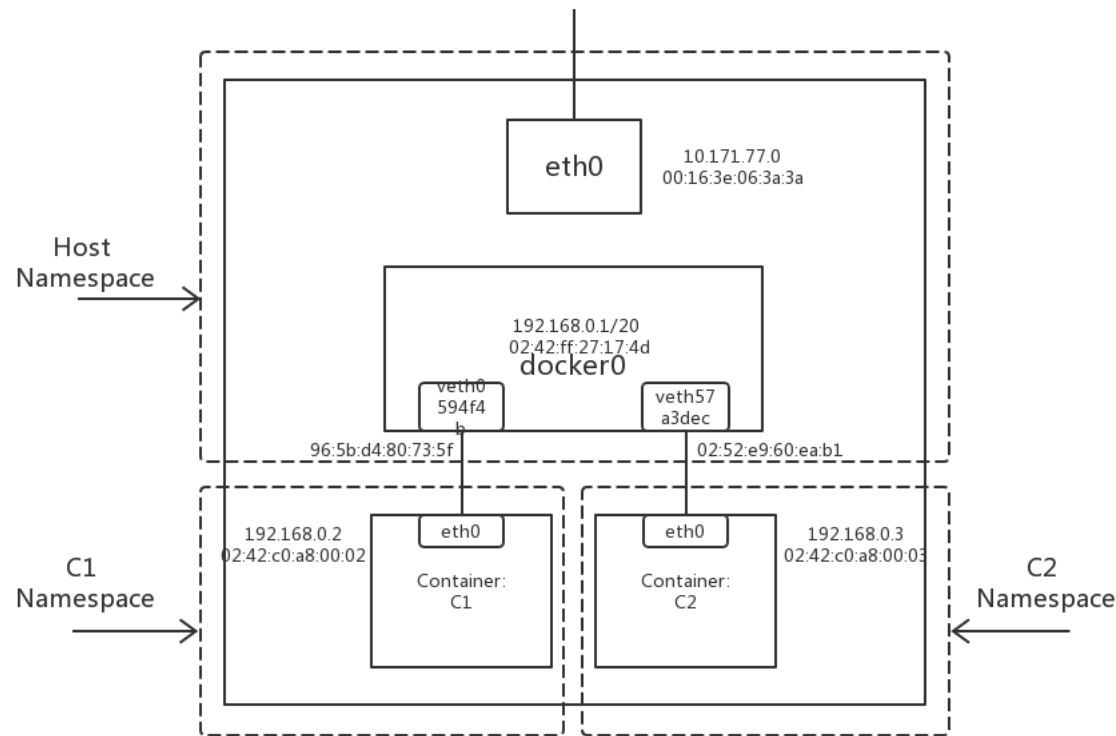
eth0       Link encap:Ethernet  HWaddr 00:16:3e:06:3a:3a
           inet addr:10.171.77.0  Bcast:10.171.79.255  Mask:255.255.248.0
           ... ...

lo         Link encap:Local Loopback
           inet addr:127.0.0.1  Mask:255.0.0.0
           ... ...

veth0594f4b Link encap:Ethernet  HWaddr 96:5b:d4:80:73:5f
           UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
           ... ...

veth57a3dec Link encap:Ethernet  HWaddr 02:52:e9:60:ea:b1
           UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
           ... ...
```

为了方便大家理解，这里附上一幅简易的容器网络拓扑:

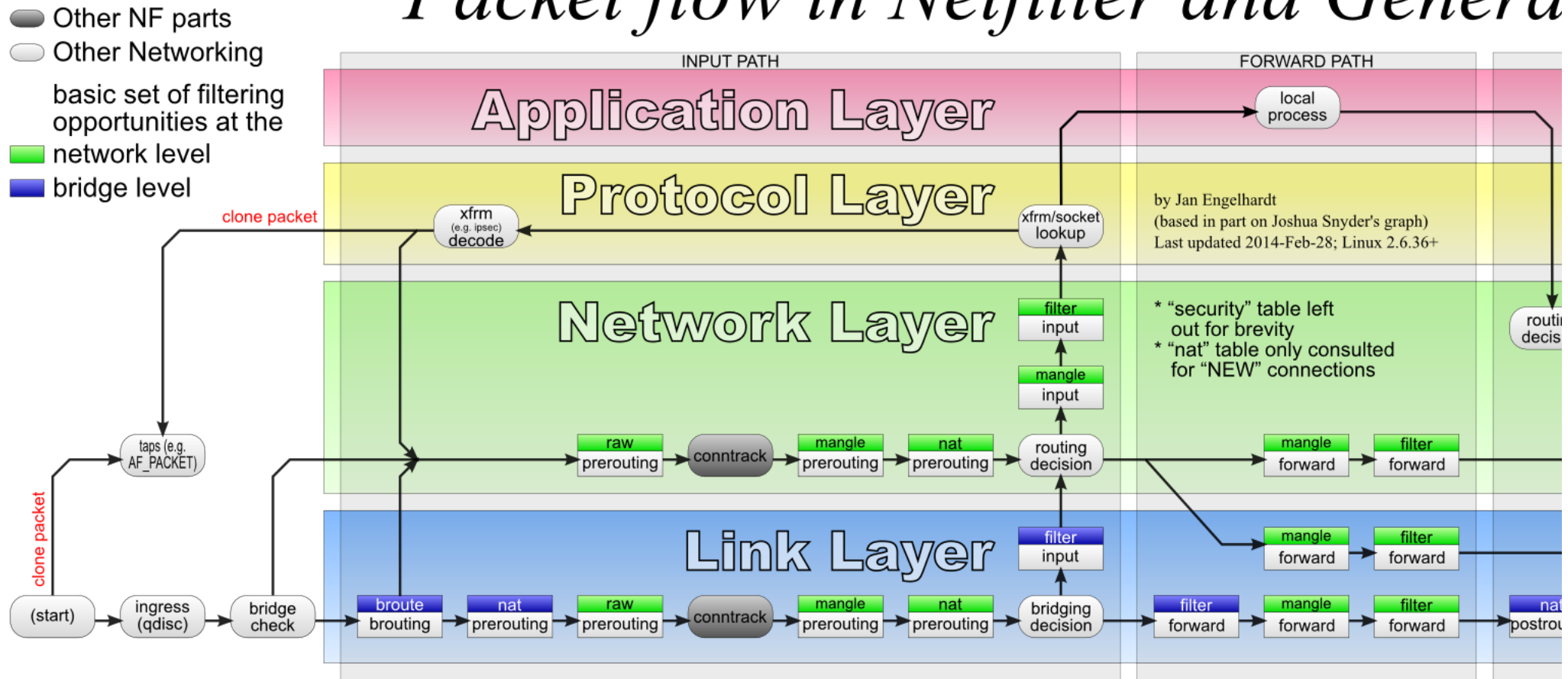


二、调试工具配置

Docker单机容器网络默认使用的是桥接网络，所有启动的容器均桥接在Docker引擎创建的docker0 linux bridge上，因此内核对Linux bridge的处理逻辑是理解Docker容器网络的关键。

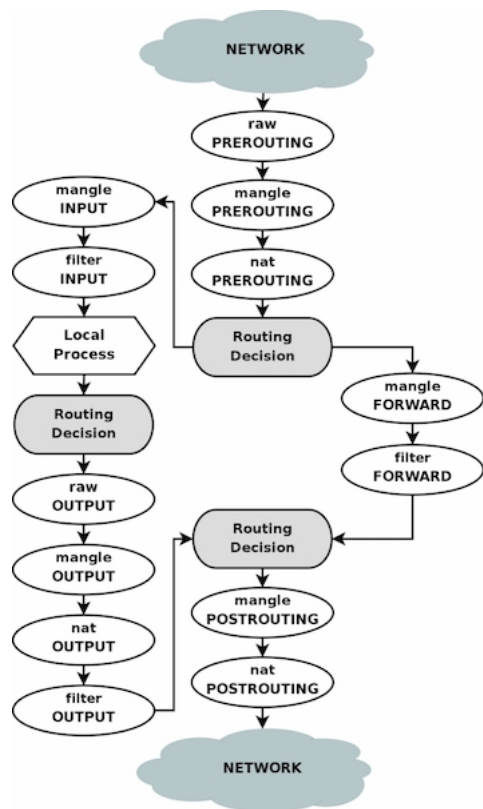
与硬件网桥/交换机不同的是，Linux Bridge还具备三层网络，即IP层的功能，也就是docker0既是一个网桥也是一个具备三层转发功能的网卡设备。传统意义上，按照iso网络七层规范，iptables工作在三层，而网桥是一个二层(数据链路层)设备，但Linux协议栈针对网桥设备的实现却在网络层的规则链(ebtables)中串接了iptables的规则链处理，即在二层也可以处理ip包，这是为了实现桥接透明防火墙的需要。但实现也会保证每个packet数据包仅会走一次iptables的某个chain，要么在linker layer走，要么在network layer走，不会出现在linker layer走一次，又在network layer重复走一次的情况。关于这种基于linux bridge的ebtables和iptables的交互规则，在netfilter官网的一篇名为《[ebtables/iptables interaction on a Linux-based bridge](#)》文档中有详细说明，这篇文章也是后续分析的一个重要参考。下面这幅图也是文章中提到的那幅netfilter数据流全图，后续在分析时会反复回到这幅图（后续简称为：全图）：

Packet flow in Netfilter and Genera



建议：右键在新标签中打开图片看大图

关于数据包在iptables的各条chain的流经图可以参见下面：



1、iptables TRACE target的设置

在本次实验中，我们主要需要查看数据包的流转路径，因此我们需要针对iptables的data flow进行跟踪。之前，我曾使用过iptables提供的LOG target或mark set&match方式来跟踪iptables中的数据流，但这两种方式都不理想，需要针对特定流程插入LOG target或match在入口包设定好的mark，对iptables规则的侵入较大，调试和观察也较为复杂；iptables自身提供了TRACE功能，一旦设定，当数据包匹配到任意chain上任意table的处理规则时，iptables会在系统日志(/var/log/syslog)中自动输出此时的数据包状态日志。

我们为iptables规则添加TRACE，TRACE target只能在iptables的raw表中添加，raw表中有两条iptables built-in chain: PREROUTING和OUTPUT，分别代表网卡数据入口和本地进程下推数据的出口。TRACE target就添加在这两条chain上，步骤如下：

```
# iptables -t raw -A OUTPUT -p icmp -j TRACE
# iptables -t raw -A PREROUTING -p icmp -j TRACE
```

注意：我们采用icmp协议(ping协议)进行测试，因此我们只TRACE icmp协议的请求和应答包。

2、ebtables的调试设置

我们的重点在iptables，为ebtables只是辅助，帮助我们看清数据包到底是在哪一层被hook进iptables的规则链中进行治疗的。因此我们在全图中的每个ebtables的built-in chain上都加上LOG（ebtables目前还不支持TRACE）：

```
# ebtables -t broute -A BROUTING -p ipv4 --ip-proto 1 --log-level 6 --log-ip --log-prefix "TRACE: eb:broute:BROUTING" -j ACCEPT
# ebtables -t nat -A OUTPUT -p ipv4 --ip-proto 1 --log-level 6 --log-ip --log-prefix "TRACE: eb:nat:OUTPUT" -j ACCEPT
# ebtables -t nat -A PREROUTING -p ipv4 --ip-proto 1 --log-level 6 --log-ip --log-prefix "TRACE: eb:nat:PREROUTING" -j ACCEPT
# ebtables -t filter -A INPUT -p ipv4 --ip-proto 1 --log-level 6 --log-ip --log-prefix "TRACE: eb:filter:INPUT" -j ACCEPT
# ebtables -t filter -A FORWARD -p ipv4 --ip-proto 1 --log-level 6 --log-ip --log-prefix "TRACE: eb:filter:FORWARD" -j ACCEPT
```

```
# ebtables -t filter -A OUTPUT -p ipv4 --ip-proto 1 --log-level 6 --log-ip --log-prefix "TRACE: eb:filter:OUTPUT" -j ACCEPT
# ebtables -t nat -A POSTROUTING -p ipv4 --ip-proto 1 --log-level 6 --log-ip --log-prefix "TRACE: eb:nat:POSTROUTING" -j ACCEPT
```

注意：这里--ip-proto 1 表示仅match icmp packet。

3、iptables和ebtables规则全文

启动两个容器并添加上述规则后，当前的iptables规则如下：（通过iptables-save输出的按table组织的rules）

```
# iptables-save
# Generated by iptables-save v1.6.0 on Sun Nov  5 14:50:46 2017
*raw

: PREROUTING ACCEPT [1564539:108837380]
: OUTPUT ACCEPT [1504962:130805835]
-A PREROUTING -p icmp -j TRACE
-A OUTPUT -p icmp -j TRACE
COMMIT
# Completed on Sun Nov  5 14:50:46 2017
# Generated by iptables-save v1.6.0 on Sun Nov  5 14:50:46 2017
*filter
: INPUT ACCEPT [1564535:108837044]
: FORWARD DROP [0:0]
: OUTPUT ACCEPT [1504968:130806627]

: DOCKER - [0:0]

: DOCKER-ISOLATION - [0:0]

: DOCKER-USER - [0:0]

-A FORWARD -j DOCKER-USER
-A FORWARD -j DOCKER-ISOLATION
-A FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -o docker0 -j DOCKER
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A FORWARD -i docker0 -o docker0 -j ACCEPT
-A DOCKER-ISOLATION -j RETURN
-A DOCKER-USER -j RETURN
COMMIT
# Completed on Sun Nov  5 14:50:46 2017
# Generated by iptables-save v1.6.0 on Sun Nov  5 14:50:46 2017
*nat

: PREROUTING ACCEPT [280:14819]
: INPUT ACCEPT [278:14651]
: OUTPUT ACCEPT [639340:38370263]

: POSTROUTING ACCEPT [639342:38370431]

: DOCKER - [0:0]

-A PREROUTING -m addrtype --dst-type LOCAL -j DOCKER
-A OUTPUT ! -d 127.0.0.0/8 -m addrtype --dst-type LOCAL -j DOCKER
-A POSTROUTING -s 192.168.0.0/20 ! -o docker0 -j MASQUERADE
-A DOCKER -i docker0 -j RETURN
COMMIT
# Completed on Sun Nov  5 14:50:46 2017
```

而ebtables的规则如下：

```
# ebtables-save
# Generated by ebtables-save v1.0 on Sun Nov  5 16:51:50 CST 2017
*nat
: PREROUTING ACCEPT
: OUTPUT ACCEPT
: POSTROUTING ACCEPT
-A PREROUTING -p IPv4 --ip-proto icmp --log-level info --log-prefix "TRACE: eb:nat:PREROUTING" --log-ip -j ACCEPT
-A OUTPUT -p IPv4 --ip-proto icmp --log-level info --log-prefix "TRACE: eb:nat:OUTPUT" --log-ip -j ACCEPT
```

```
-A POSTROUTING -p IPv4 --ip-proto icmp --log-level info --log-prefix "TRACE: eb:nat:POSTROUTING" --log-ip -j ACCEPT

*broute
:BROUTING ACCEPT
-A BROUTING -p IPv4 --ip-proto icmp --log-level info --log-prefix "TRACE: eb:broute:BROUTING" --log-ip -j ACCEPT

*filter
:INPUT ACCEPT
:FORWARD ACCEPT
:OUTPUT ACCEPT
-A INPUT -p IPv4 --ip-proto icmp --log-level info --log-prefix "TRACE: eb:filter:INPUT" --log-ip -j ACCEPT
-A FORWARD -p IPv4 --ip-proto icmp --log-level info --log-prefix "TRACE: eb:filter:FORWARD" --log-ip -j ACCEPT
-A OUTPUT -p IPv4 --ip-proto icmp --log-level info --log-prefix "TRACE: eb:filter:OUTPUT" --log-ip -j ACCEPT
```

对于iptables，我们还可以通过iptables命令输出另外一种组织形式的规则列表，我们这里列出filter和nat这两个重要的table的规则(输出规则number，便于后续match分析时查看)：

```
# iptables -nL --line-numbers -v -t filter
Chain INPUT (policy ACCEPT 2558K packets, 178M bytes)
num  pkts bytes target    prot opt in     out     source                 destination

Chain FORWARD (policy DROP 0 packets, 0 bytes)
num  pkts bytes target    prot opt in     out     source                 destination
1    10   840 DOCKER-USER  all  --  *      *        0.0.0.0/0             0.0.0.0/0
2    10   840 DOCKER-ISOLATION  all  --  *      *        0.0.0.0/0             0.0.0.0/0
3     7   588 ACCEPT     all  --  *      docker0  0.0.0.0/0             0.0.0.0/0          ctstate RELATED,ESTABLISHED
4     3   252 DOCKER     all  --  *      docker0  0.0.0.0/0             0.0.0.0/0
5     0     0 ACCEPT     all  --  docker0 !docker0  0.0.0.0/0             0.0.0.0/0
6     3   252 ACCEPT     all  --  docker0 docker0   0.0.0.0/0             0.0.0.0/0

Chain OUTPUT (policy ACCEPT 2460K packets, 214M bytes)
num  pkts bytes target    prot opt in     out     source                 destination

Chain DOCKER (1 references)
num  pkts bytes target    prot opt in     out     source                 destination

Chain DOCKER-ISOLATION (1 references)
num  pkts bytes target    prot opt in     out     source                 destination
1    10   840 RETURN    all  --  *      *        0.0.0.0/0             0.0.0.0/0

Chain DOCKER-USER (1 references)
num  pkts bytes target    prot opt in     out     source                 destination
1    10   840 RETURN    all  --  *      *        0.0.0.0/0             0.0.0.0/0

# iptables -nL --line-numbers -v -t nat
Chain PREROUTING (policy ACCEPT 884 packets, 46522 bytes)
num  pkts bytes target    prot opt in     out     source                 destination
1    881 46270 DOCKER    all  --  *      *        0.0.0.0/0             0.0.0.0/0          ADDRTYPE match dst-type LOCAL

Chain INPUT (policy ACCEPT 881 packets, 46270 bytes)
num  pkts bytes target    prot opt in     out     source                 destination

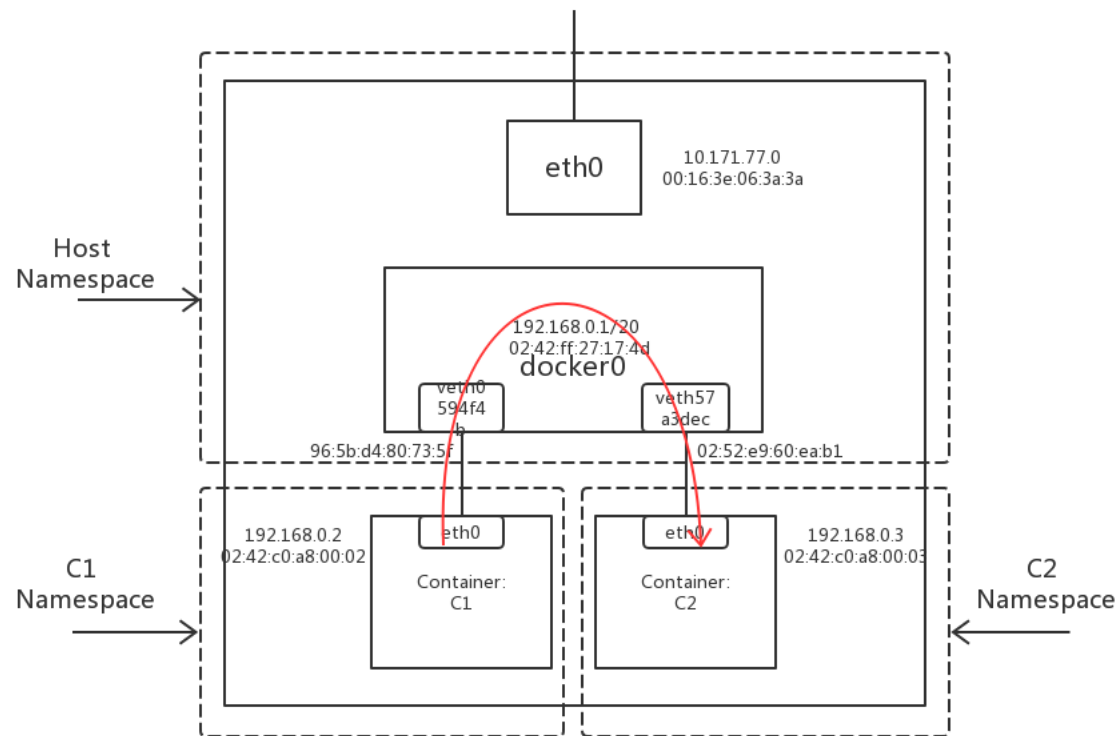
Chain OUTPUT (policy ACCEPT 1048K packets, 63M bytes)
num  pkts bytes target    prot opt in     out     source                 destination
1     0     0 DOCKER    all  --  *      *        0.0.0.0/0             !127.0.0.0/8        ADDRTYPE match dst-type LOCAL

Chain POSTROUTING (policy ACCEPT 1048K packets, 63M bytes)
num  pkts bytes target    prot opt in     out     source                 destination
1     0     0 MASQUERADE  all  --  *      !docker0  192.168.0.0/20         0.0.0.0/0

Chain DOCKER (2 references)
num  pkts bytes target    prot opt in     out     source                 destination
1     0     0 RETURN    all  --  docker0 *        0.0.0.0/0             0.0.0.0/0
```

三、Container to Container

下面，我们分三种情况来看容器网络的数据包是如何流动的，首先是Container to Container。



我们在容器C1中执行ping 3次 C2的命令：

```
# docker exec c1 ping -c 3 192.168.0.3
PING 192.168.0.3 (192.168.0.3) 56(84) bytes of data.
64 bytes from 192.168.0.3: icmp_seq=1 ttl=64 time=0.226 ms
64 bytes from 192.168.0.3: icmp_seq=2 ttl=64 time=0.159 ms
64 bytes from 192.168.0.3: icmp_seq=3 ttl=64 time=0.185 ms

--- 192.168.0.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1998ms
rtt min/avg/max/mdev = 0.159/0.190/0.226/0.027 ms
```

在容器c1(192.168.0.2)中，icmp request由ping程序(c1 namespace中的local process)发出。c1 network namespace中的路由表如下：

```
# docker exec c1 netstat -rn
Kernel IP routing table
Destination Gateway Genmask Flags MSS Window irtt Iface
0.0.0.0 192.168.0.1 0.0.0.0 UG 0 0 0 eth0
192.168.0.0 0.0.0.0 255.255.240.0 U 0 0 0 eth0
```

由于目标容器地址为192.168.0.3，在容器c1的直连网络上，走第二条直连路由（非默认路由），数据包通过eth0发出。

由于c1 namespace中的eth0通过veth机制连接在host namespace的docker0 bridge的一个Slave port上，因此上述数据包通过docker0 bridge的slave port: veth0594f4b流入docker0 bridge。

这里再强调一下linux bridge设备。Linux下的Bridge是一种虚拟设备，它依赖于一个或多个**从设备**。它不是内核虚拟出的和**从设备**同一层次的镜像设备，而是内核虚拟出的一个高一层次的设备，并把**从设备**虚拟化为端口port，同时处理各个**从设备**的数据收发及转发。bridge设备是建立在从设备之上的（这些从设备可以是实际设备，也可以是vlan设备等），并且我们可以为bridge准备一个IP（bridge设备的MAC地址是它所有从设备中最小的MAC地址），这样该主机就可以通过这个bridge设备与网络中的其它主机通信了。另外一旦某个网络设备被“插到”linux bridge上，这个网络设备将会变为bridge的**从设备**，被虚拟化为端口port，**从设备**的IP及MAC都不可用，好似被bridge剥夺了被内核网络栈处理的资格；它们被设置为接收任何包，对其流入的数据包的处理交由bridge完成，并最终由bridge设备来决定数据包的去向：接收到本机、转发或丢弃。

因此，位于host namespace的docker0 bridge从slave port: veth0594f4b收到icmp request后，我们不会看到veth0594f4b这一netdev被内核网络栈程序单独处理(比如：单独走一遍ebtables和iptables chains)，而是进入bridge处理逻辑（此时可以回顾一下上面的全图）。由于数据包已经进入到了**host namespace**，因此我们可以通过ebtables和iptables输出的Trace和log来跟踪数据包流转的路径了：

1、start -> bridgecheck -> linker layer

```
TRACE: eb:broute:BROUTING IN=veth0594f4b OUT= MAC source = 02:42:c0:a8:00:02 MAC dest = 02:42:c0:a8:00:03 proto = 0x0800 IP SRC=192.168.0.2 IP DST=192.168.0.3, IP tos=0x00, IP proto=1
TRACE: eb:nat:PREROUTING IN=veth0594f4b OUT= MAC source = 02:42:c0:a8:00:02 MAC dest = 02:42:c0:a8:00:03 proto = 0x0800 IP SRC=192.168.0.2 IP DST=192.168.0.3, IP tos=0x00, IP proto=1
```

从最初的trace log来看，在bridge check之后(发现it is a linux bridge)，数据包进入到linker layer中；并且在linker layer的BROUTING built-in chain之后，数据包没有被转移到上面的network layer，而是**继续linker layer的行程**：进入linker layer的nat:PREROUTING中。

2、call iptables chain rules in linker layer

结合全图中的图示和日志输出，在linker layer的nat:PREROUTING之后，linker layer调用了上层iptables的处理规则：raw:PREROUTING和nat:PREROUTING：

```
TRACE: raw:PREROUTING:policy:2 IN=docker0 OUT= PHYSIN=veth0594f4b MAC=02:42:c0:a8:00:03:02:42:c0:a8:00:02:08:00 SRC=192.168.0.2 DST=192.168.0.3 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=47066 DF PROTO=ICMP 1
TRACE: nat:PREROUTING:policy:2 IN=docker0 OUT= PHYSIN=veth0594f4b MAC=02:42:c0:a8:00:03:02:42:c0:a8:00:02:08:00 SRC=192.168.0.2 DST=192.168.0.3 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=47066 DF PROTO=ICMP 1
```

Trace target在数据包match table、chains的policy或rules时会输出日志，日志格式：“TRACE:tablename:chainname:type:rulenum”。当匹配到的是普通rules时，type=”rule”；当碰到一个user-defined chain的return target时，type=”return”；当匹配到built-in chain(比如：PREROUTING、INPUT、OUTPUT、FORWARD和POSTROUTING)的default policy时，type=”policy”。

从上面的日志输出来看，似乎PREROUTING chain的raw table中的Trace target不能被trace自身match，因此trace log输出的是匹配raw table built-in chain: PREROUTING的default policy: ACCEPT，num=2(policy和rules整体排序后的序号)；在PREROUTING chain的nat表中匹配时，Trace也仅匹配到了default policy，rule 1（target: Docker）没有匹配上；

这里有一点奇怪的是mangle table没有任何输出，即便是default policy的也没有，原因暂不明。

3、bridge decision

根据全图和后续的日志，我们得到了**bridge decision**的结果：继续在linker layer上处理数据包，一路向右。不过在处理的路径上依旧调用了iptables的规则：

```
TRACE: eb:filter:FORWARD IN=veth0594f4b OUT=veth57a3dec MAC source = 02:42:c0:a8:00:02 MAC dest = 02:42:c0:a8:00:03 proto = 0x0800 IP SRC=192.168.0.2 IP DST=192.168.0.3, IP tos=0x00, IP proto=1
TRACE: filter:FORWARD:rule:1 IN=docker0 OUT=docker0 PHYSIN=veth0594f4b PHYSOUT=veth57a3dec MAC=02:42:c0:a8:00:03:02:42:c0:a8:00:02:08:00 SRC=192.168.0.2 DST=192.168.0.3 LEN=84 TOS=0x00 PREC=0x00 TTL=64
TRACE: filter:DOCKER-USER:return:1 IN=docker0 OUT=docker0 PHYSIN=veth0594f4b PHYSOUT=veth57a3dec MAC=02:42:c0:a8:00:03:02:42:c0:a8:00:02:08:00 SRC=192.168.0.2 DST=192.168.0.3 LEN=84 TOS=0x00 PREC=0x00
TRACE: filter:FORWARD:rule:2 IN=docker0 OUT=docker0 PHYSIN=veth0594f4b PHYSOUT=veth57a3dec MAC=02:42:c0:a8:00:03:02:42:c0:a8:00:02:08:00 SRC=192.168.0.2 DST=192.168.0.3 LEN=84 TOS=0x00 PREC=0x00 TTL=64
TRACE: filter:DOCKER-ISOLATION:return:1 IN=docker0 OUT=docker0 PHYSIN=veth0594f4b PHYSOUT=veth57a3dec MAC=02:42:c0:a8:00:03:02:42:c0:a8:00:02:08:00 SRC=192.168.0.2 DST=192.168.0.3 LEN=84 TOS=0x00 PREC=
TRACE: filter:FORWARD:rule:4 IN=docker0 OUT=docker0 PHYSIN=veth0594f4b PHYSOUT=veth57a3dec MAC=02:42:c0:a8:00:03:02:42:c0:a8:00:02:08:00 SRC=192.168.0.2 DST=192.168.0.3 LEN=84 TOS=0x00 PREC=0x00 TTL=64
TRACE: filter:DOCKER:return:1 IN=docker0 OUT=docker0 PHYSIN=veth0594f4b PHYSOUT=veth57a3dec MAC=02:42:c0:a8:00:03:02:42:c0:a8:00:02:08:00 SRC=192.168.0.2 DST=192.168.0.3 LEN=84 TOS=0x00 PREC=0x00 TTL=6
TRACE: filter:FORWARD:rule:6 IN=docker0 OUT=docker0 PHYSIN=veth0594f4b PHYSOUT=veth57a3dec MAC=02:42:c0:a8:00:03:02:42:c0:a8:00:02:08:00 SRC=192.168.0.2 DST=192.168.0.3 LEN=84 TOS=0x00 PREC=0x00 TTL=64
```

bridge decision决定的依据或则规则是什么呢？《[ebtables/iptables interaction on a Linux-based bridge](#)》一文给了我们一些答案：

The bridge's decision for a frame can be one of these:

- * bridge it, if the destination MAC address is on another side of the bridge;
- * flood it over all the forwarding bridge ports, if the position of the box with the destination MAC is unknown to the bridge;
- * pass it to the higher protocol code (the IP code), if the destination MAC address is that of the bridge or of one of its ports;
- * ignore it, if the destination MAC address is located on the same side of the bridge.

不过即便按照这几条规则，我依然有一定困惑，那就是真实的处理是：依旧在linker layer，但掺杂了上层网络层的处理规则。

另外，你可能会发现iptables log里MAC值的格式很怪异(比如：**MAC=02:42:c0:a8:00:03:02:42:c0:a8:00:02:08:00**)，非常long。其实这个MAC值是一个组合：Souce MAC, Destination MAC和 frame type的组合。

```
02:42:c0:a8:00:03: Destination MAC=00:60:dd:45:67:ea
02:42:c0:a8:00:02: Source MAC=00:60:dd:45:4c:92
08:00 : Type=08:00 (ethernet frame carried an IPv4 datagram)
```

4、eb:nat:POSTROUTING -> nat:POSTROUTING -> egress(qdisc)

最后packet进入linker layer的POSTROUTING built-in chain:

```
TRACE: eb:nat:POSTROUTING IN= OUT=veth57a3dec MAC source = 02:42:c0:a8:00:02 MAC dest = 02:42:c0:a8:00:03 proto = 0x0800 IP SRC=192.168.0.2 IP DST=192.168.0.3, IP tos=0x00, IP proto=1
TRACE: nat:POSTROUTING:policy:2 IN= OUT=docker0 PHYSIN=veth0594f4b PHYSOUT=veth57a3dec SRC=192.168.0.2 DST=192.168.0.3 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=47066 DF PROTO=ICMP TYPE=8 CODE=0 ID=90 SEQ=1
```

iptables nat:POSTROUTING没有匹配上docker引擎增加的那条target为DOCKER的rule，于是输出了default policy的日志。

进入到egress(qdisc)后，相当于数据包到了bridge上的另一个slave port(veth57a3dec)上，此时数据包必须被送回网络上，于是进入到容器C2的eth0中。离开了host namespace，我们的日志便追踪不到了。

容器c2因为所在的network namespace是独立于host namespace的，因此有自己的iptables规则（如果未设置，均为默认accept），不受host namespace中的iptables的影响。

5、“消失”的iptables的nat:PREROUTING和nat:POSTROUTING

C2容器回复ping response的路径与request甚为相似，这里一次性将全部日志列出：

```
TRACE: eb:broute:BROUTING IN=veth57a3dec OUT= MAC source = 02:42:c0:a8:00:03 MAC dest = 02:42:c0:a8:00:02 proto = 0x0800 IP SRC=192.168.0.3 IP DST=192.168.0.2, IP tos=0x00, IP proto=1
TRACE: eb:nat:PREROUTING IN=veth57a3dec OUT= MAC source = 02:42:c0:a8:00:03 MAC dest = 02:42:c0:a8:00:02 proto = 0x0800 IP SRC=192.168.0.3 IP DST=192.168.0.2, IP tos=0x00, IP proto=1
TRACE: raw:PREROUTING:policy:2 IN=docker0 OUT= PHYSIN=veth57a3dec MAC=02:42:c0:a8:00:02:02:42:c0:a8:00:03:08:00 SRC=192.168.0.3 DST=192.168.0.2 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=5962 PROTO=ICMP TYPE=8
TRACE: eb:filter:FORWARD IN=veth57a3dec OUT=veth0594f4b MAC source = 02:42:c0:a8:00:03 MAC dest = 02:42:c0:a8:00:02 proto = 0x0800 IP SRC=192.168.0.3 IP DST=192.168.0.2, IP tos=0x00, IP proto=1
TRACE: filter:FORWARD:rule:1 IN=docker0 OUT=docker0 PHYSIN=veth57a3dec PHYSOUT=veth0594f4b MAC=02:42:c0:a8:00:02:02:42:c0:a8:00:03:08:00 SRC=192.168.0.3 DST=192.168.0.2 LEN=84 TOS=0x00 PREC=0x00 TTL=64
TRACE: filter:DOCKER-USER:return:1 IN=docker0 OUT=docker0 PHYSIN=veth57a3dec PHYSOUT=veth0594f4b MAC=02:42:c0:a8:00:02:02:42:c0:a8:00:03:08:00 SRC=192.168.0.3 DST=192.168.0.2 LEN=84 TOS=0x00 PREC=0x00
TRACE: filter:FORWARD:rule:2 IN=docker0 OUT=docker0 PHYSIN=veth57a3dec PHYSOUT=veth0594f4b MAC=02:42:c0:a8:00:02:02:42:c0:a8:00:03:08:00 SRC=192.168.0.3 DST=192.168.0.2 LEN=84 TOS=0x00 PREC=0x00 TTL=64
TRACE: filter:DOCKER-ISOLATION:return:1 IN=docker0 OUT=docker0 PHYSIN=veth57a3dec PHYSOUT=veth0594f4b MAC=02:42:c0:a8:00:02:02:42:c0:a8:00:03:08:00 SRC=192.168.0.3 DST=192.168.0.2 LEN=84 TOS=0x00 PREC=0x00
TRACE: filter:FORWARD:rule:3 IN=docker0 OUT=docker0 PHYSIN=veth57a3dec PHYSOUT=veth0594f4b MAC=02:42:c0:a8:00:02:02:42:c0:a8:00:03:08:00 SRC=192.168.0.3 DST=192.168.0.2 LEN=84 TOS=0x00 PREC=0x00 TTL=64
TRACE: eb:nat:POSTROUTING IN= OUT=veth0594f4b MAC source = 02:42:c0:a8:00:03 MAC dest = 02:42:c0:a8:00:02 proto = 0x0800 IP SRC=192.168.0.3 IP DST=192.168.0.2, IP tos=0x00, IP proto=1
```

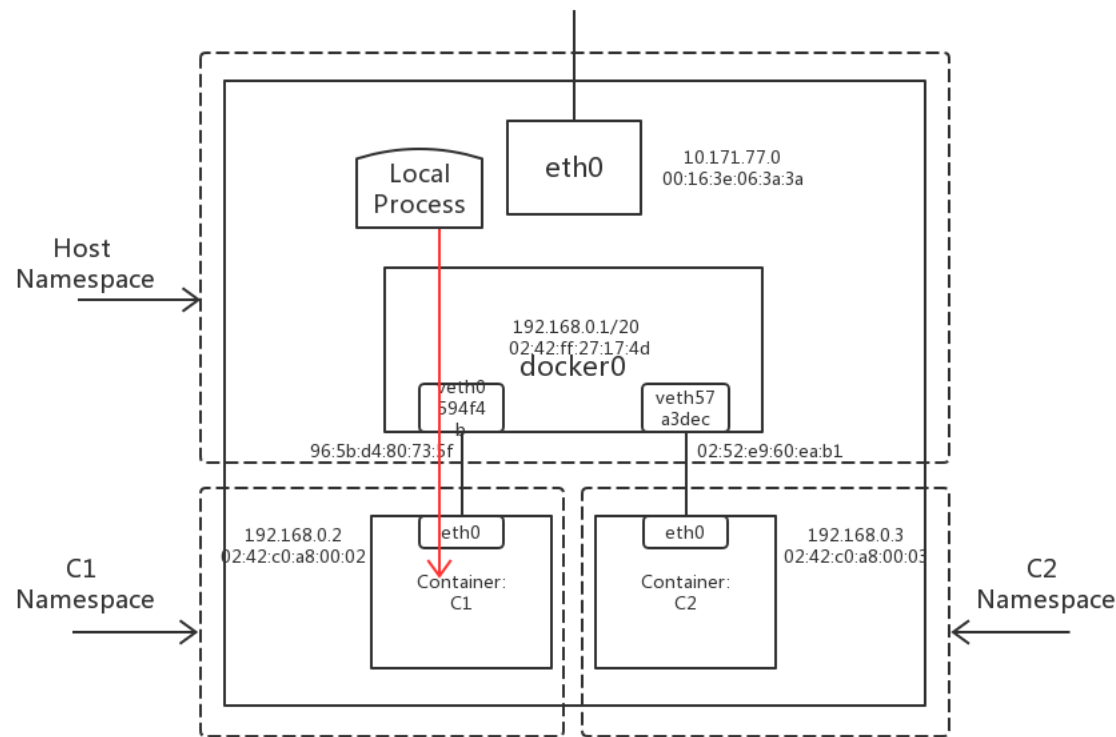
仔细观察，我们发现虽然与request的路径类似，但依旧有不同：**iptables的nat:PREROUTING和nat:POSTROUTING消失了**。Why? iptables就是这么设计的。iptables会跟踪connection的状态，当一个connection的首个包经过一次后，connection的状态由NEW变成了ESTABLISHED；对于ESTABLISHED的connection的后续packets，内核会自动按照该connection的首个包在nat:PREROUTING和nat:POSTROUTING环节的处理方式进行处理，而不再流经这两个链中的nat表逻辑。而ebtables中似乎没有这个逻辑。

后续的ping的第二个、第三个流程也印证了上述设计，这里仅列出ping request packet 2:

```
TRACE: eb:broute:BROUTING IN=veth0594f4b OUT= MAC source = 02:42:c0:a8:00:02 MAC dest = 02:42:c0:a8:00:03 proto = 0x0800 IP SRC=192.168.0.2 IP DST=192.168.0.3, IP tos=0x00, IP proto=1
TRACE: eb:nat:PREROUTING IN=veth0594f4b OUT= MAC source = 02:42:c0:a8:00:02 MAC dest = 02:42:c0:a8:00:03 proto = 0x0800 IP SRC=192.168.0.2 IP DST=192.168.0.3, IP tos=0x00, IP proto=1
TRACE: raw:PREROUTING:policy:2 IN=docker0 OUT= PHYSIN=veth0594f4b MAC=02:42:c0:a8:00:03:02:42:c0:a8:00:02:08:00 SRC=192.168.0.2 DST=192.168.0.3 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=47310 DF PROTO=ICMP TYPE=8
TRACE: eb:filter:FORWARD IN=veth0594f4b OUT=veth57a3dec MAC source = 02:42:c0:a8:00:02 MAC dest = 02:42:c0:a8:00:03 proto = 0x0800 IP SRC=192.168.0.2 IP DST=192.168.0.3, IP tos=0x00, IP proto=1
TRACE: filter:FORWARD:rule:1 IN=docker0 OUT=docker0 PHYSIN=veth0594f4b PHYSOUT=veth57a3dec MAC=02:42:c0:a8:00:03:02:42:c0:a8:00:02:08:00 SRC=192.168.0.2 DST=192.168.0.3 LEN=84 TOS=0x00 PREC=0x00 TTL=64
TRACE: filter:DOCKER-USER:return:1 IN=docker0 OUT=docker0 PHYSIN=veth0594f4b PHYSOUT=veth57a3dec MAC=02:42:c0:a8:00:03:02:42:c0:a8:00:02:08:00 SRC=192.168.0.2 DST=192.168.0.3 LEN=84 TOS=0x00 PREC=0x00
TRACE: filter:FORWARD:rule:2 IN=docker0 OUT=docker0 PHYSIN=veth0594f4b PHYSOUT=veth57a3dec MAC=02:42:c0:a8:00:03:02:42:c0:a8:00:02:08:00 SRC=192.168.0.2 DST=192.168.0.3 LEN=84 TOS=0x00 PREC=0x00 TTL=64
TRACE: filter:DOCKER-ISOLATION:return:1 IN=docker0 OUT=docker0 PHYSIN=veth0594f4b PHYSOUT=veth57a3dec MAC=02:42:c0:a8:00:03:02:42:c0:a8:00:02:08:00 SRC=192.168.0.2 DST=192.168.0.3 LEN=84 TOS=0x00 PREC=0x00
TRACE: filter:FORWARD:rule:3 IN=docker0 OUT=docker0 PHYSIN=veth0594f4b PHYSOUT=veth57a3dec MAC=02:42:c0:a8:00:03:02:42:c0:a8:00:02:08:00 SRC=192.168.0.2 DST=192.168.0.3 LEN=84 TOS=0x00 PREC=0x00 TTL=64
TRACE: eb:nat:POSTROUTING IN= OUT=veth57a3dec MAC source = 02:42:c0:a8:00:02 MAC dest = 02:42:c0:a8:00:03 proto = 0x0800 IP SRC=192.168.0.2 IP DST=192.168.0.3, IP tos=0x00, IP proto=1
```

全部日志内容请参见：[docker-bridge-network-demo-iptables-trace-log.txt文件](#)，这里不赘述。

四、Local Process to Container



很多”疑难”环节在上面的container to container数据流分析时已经做了解惑，因此后续local process to container和container to external流程将不会再细致描述，说明会略微泛泛一些，不那么细致。

我们在host上执行ping C1三次：

```
# ping -c 3 192.168.0.2
PING 192.168.0.2 (192.168.0.2) 56(84) bytes of data.
64 bytes from 192.168.0.2: icmp_seq=1 ttl=64 time=0.160 ms
64 bytes from 192.168.0.2: icmp_seq=2 ttl=64 time=0.105 ms
64 bytes from 192.168.0.2: icmp_seq=3 ttl=64 time=0.131 ms

--- 192.168.0.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2000ms
rtt min/avg/max/mdev = 0.105/0.132/0.160/0.022 ms
```

1、local process -> routing decision -> iptables OUTPUT chain

ping request数据包从本地的ping process发出，根据目的地址路由后，选择docker0作为OUT设备：

```
TRACE: raw:OUTPUT:policy:2 IN= OUT=docker0 SRC=192.168.0.1 DST=192.168.0.2 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=18692 DF PROTO=ICMP TYPE=8 CODE=0 ID=30245 SEQ=1 UID=0 GID=0
TRACE: mangle:OUTPUT:policy:1 IN= OUT=docker0 SRC=192.168.0.1 DST=192.168.0.2 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=18692 DF PROTO=ICMP TYPE=8 CODE=0 ID=30245 SEQ=1 UID=0 GID=0
TRACE: nat:OUTPUT:policy:2 IN= OUT=docker0 SRC=192.168.0.1 DST=192.168.0.2 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=18692 DF PROTO=ICMP TYPE=8 CODE=0 ID=30245 SEQ=1 UID=0 GID=0
TRACE: filter:OUTPUT:policy:1 IN= OUT=docker0 SRC=192.168.0.1 DST=192.168.0.2 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=18692 DF PROTO=ICMP TYPE=8 CODE=0 ID=30245 SEQ=1 UID=0 GID=0
```

奇怪的是这次mangle chain居然有trace log输出:(。

2、进入linker layer：iptables POSTROUTING -> ebtables OUTPUT -> ebtables POSTROUTING

由于是OUT是bridge设备，因此要进入到ebtable中走一遭：

```
TRACE: mangle:POSTROUTING:policy:1 IN= OUT=docker0 SRC=192.168.0.1 DST=192.168.0.2 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=18692 DF PROTO=ICMP TYPE=8 CODE=0 ID=30245 SEQ=1 UID=0 GID=0
TRACE: nat:POSTROUTING:policy:2 IN= OUT=docker0 SRC=192.168.0.1 DST=192.168.0.2 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=18692 DF PROTO=ICMP TYPE=8 CODE=0 ID=30245 SEQ=1 UID=0 GID=0
TRACE: eb:nat:OUTPUT IN= OUT=veth57a3dec MAC source = 02:42:ff:27:17:4d MAC dest = 02:42:c0:a8:00:02 proto = 0x0800 IP SRC=192.168.0.1 IP DST=192.168.0.2, IP tos=0x00, IP proto=1
TRACE: eb:filter:OUTPUT IN= OUT=veth57a3dec MAC source = 02:42:ff:27:17:4d MAC dest = 02:42:c0:a8:00:02 proto = 0x0800 IP SRC=192.168.0.1 IP DST=192.168.0.2, IP tos=0x00, IP proto=1
TRACE: eb:nat:POSTROUTING IN= OUT=veth57a3dec MAC source = 02:42:ff:27:17:4d MAC dest = 02:42:c0:a8:00:02 proto = 0x0800 IP SRC=192.168.0.1 IP DST=192.168.0.2, IP tos=0x00, IP proto=1
TRACE: eb:nat:OUTPUT IN= OUT=veth0594f4b MAC source = 02:42:ff:27:17:4d MAC dest = 02:42:c0:a8:00:02 proto = 0x0800 IP SRC=192.168.0.1 IP DST=192.168.0.2, IP tos=0x00, IP proto=1
TRACE: eb:filter:OUTPUT IN= OUT=veth0594f4b MAC source = 02:42:ff:27:17:4d MAC dest = 02:42:c0:a8:00:02 proto = 0x0800 IP SRC=192.168.0.1 IP DST=192.168.0.2, IP tos=0x00, IP proto=1
TRACE: eb:nat:POSTROUTING IN= OUT=veth0594f4b MAC source = 02:42:ff:27:17:4d MAC dest = 02:42:c0:a8:00:02 proto = 0x0800 IP SRC=192.168.0.1 IP DST=192.168.0.2, IP tos=0x00, IP proto=1
```

icmp的response和container to container类似，入口走的是linker layer(由于是桥设备)，在bridge decision后，走到INPUT chain：

```
TRACE: eb:broute:BRROUTING IN=veth0594f4b OUT= MAC source = 02:42:c0:a8:00:02 MAC dest = 02:42:ff:27:17:4d proto = 0x0800 IP SRC=192.168.0.2 IP DST=192.168.0.1, IP tos=0x00, IP proto=1
TRACE: eb:nat:PREROUTING IN=veth0594f4b OUT= MAC source = 02:42:c0:a8:00:02 MAC dest = 02:42:ff:27:17:4d proto = 0x0800 IP SRC=192.168.0.2 IP DST=192.168.0.1, IP tos=0x00, IP proto=1
TRACE: raw:PREROUTING:policy:2 IN=docker0 OUT= PHYSIN=veth0594f4b MAC=02:42:ff:27:17:4d:02:42:c0:a8:00:02:08:00 SRC=192.168.0.2 DST=192.168.0.1 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=56535 PROTO=ICMP TYPE=8
TRACE: mangle:PREROUTING:policy:1 IN=docker0 OUT= PHYSIN=veth0594f4b MAC=02:42:ff:27:17:4d:02:42:c0:a8:00:02:08:00 SRC=192.168.0.2 DST=192.168.0.1 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=56535 PROTO=ICMP TYPE=8
TRACE: eb:filter:INPUT IN=veth0594f4b OUT= MAC source = 02:42:c0:a8:00:02 MAC dest = 02:42:ff:27:17:4d proto = 0x0800 IP SRC=192.168.0.2 IP DST=192.168.0.1, IP tos=0x00, IP proto=1
TRACE: mangle:INPUT:policy:1 IN=docker0 OUT= PHYSIN=veth0594f4b MAC=02:42:ff:27:17:4d:02:42:c0:a8:00:02:08:00 SRC=192.168.0.2 DST=192.168.0.1 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=56535 PROTO=ICMP TYPE=8
TRACE: filter:INPUT:policy:1 IN=docker0 OUT= PHYSIN=veth0594f4b MAC=02:42:ff:27:17:4d:02:42:c0:a8:00:02:08:00 SRC=192.168.0.2 DST=192.168.0.1 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=56535 PROTO=ICMP TYPE=8
```

以上我们可以与到非桥设备的ping做比对，我们在host上ping 另外一个LAN中的host：

```
# ping -c 1 10.28.61.30
PING 10.28.61.30 (10.28.61.30) 56(84) bytes of data.
64 bytes from 10.28.61.30: icmp_seq=1 ttl=57 time=1.09 ms

--- 10.28.61.30 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.093/1.093/1.093/0.000 ms
```

得到的trace log如下：

```
icmp request:

TRACE: raw:OUTPUT:policy:2 IN= OUT=eth0 SRC=10.171.77.0 DST=10.28.61.30 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=4494 DF PROTO=ICMP TYPE=8 CODE=0 ID=30426 SEQ=1 UID=0 GID=0
TRACE: mangle:OUTPUT:policy:1 IN= OUT=eth0 SRC=10.171.77.0 DST=10.28.61.30 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=4494 DF PROTO=ICMP TYPE=8 CODE=0 ID=30426 SEQ=1 UID=0 GID=0
TRACE: nat:OUTPUT:policy:2 IN= OUT=eth0 SRC=10.171.77.0 DST=10.28.61.30 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=4494 DF PROTO=ICMP TYPE=8 CODE=0 ID=30426 SEQ=1 UID=0 GID=0
TRACE: filter:OUTPUT:policy:1 IN= OUT=eth0 SRC=10.171.77.0 DST=10.28.61.30 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=4494 DF PROTO=ICMP TYPE=8 CODE=0 ID=30426 SEQ=1 UID=0 GID=0
TRACE: mangle:POSTROUTING:policy:1 IN= OUT=eth0 SRC=10.171.77.0 DST=10.28.61.30 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=4494 DF PROTO=ICMP TYPE=8 CODE=0 ID=30426 SEQ=1 UID=0 GID=0
TRACE: nat:POSTROUTING:policy:2 IN= OUT=eth0 SRC=10.171.77.0 DST=10.28.61.30 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=4494 DF PROTO=ICMP TYPE=8 CODE=0 ID=30426 SEQ=1 UID=0 GID=0

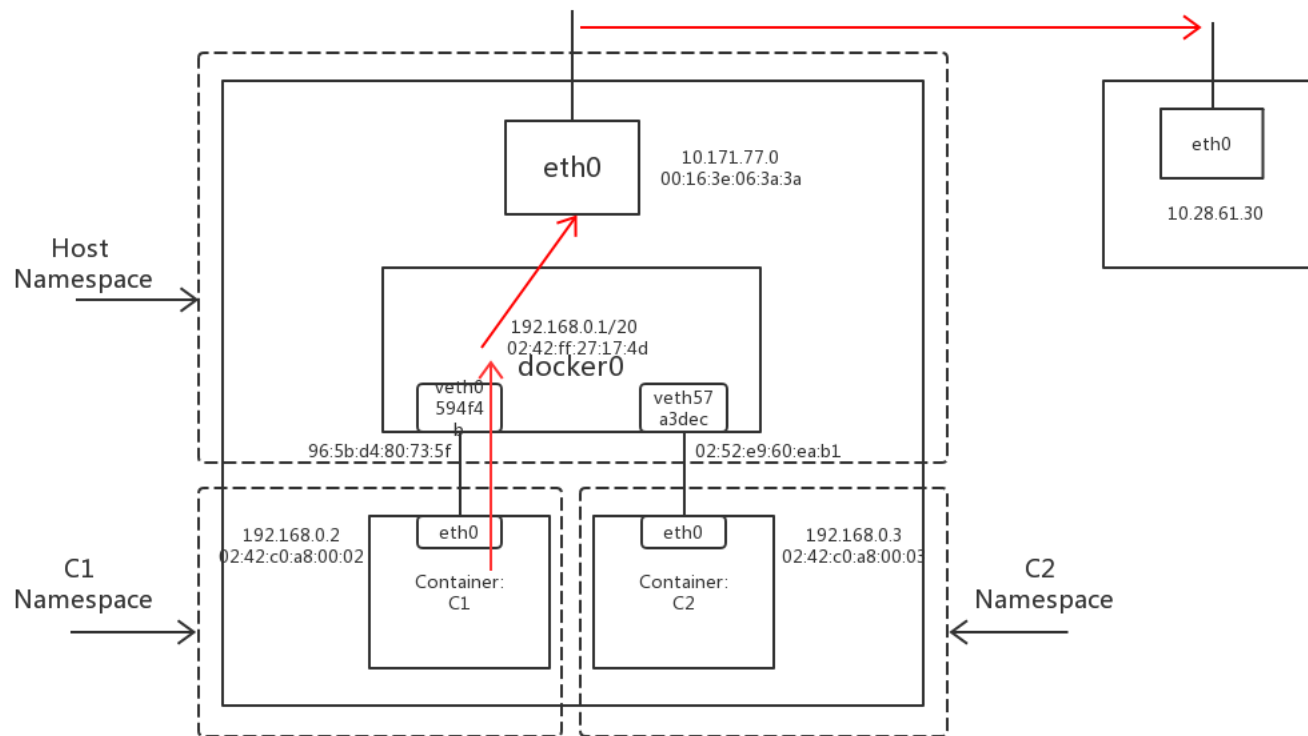
icmp response:

TRACE: raw:PREROUTING:policy:2 IN=eth0 OUT= MAC=00:16:3e:06:3a:3a:00:2a:6a:aa:12:7c:08:00 SRC=10.28.61.30 DST=10.171.77.0 LEN=84 TOS=0x00 PREC=0x00 TTL=57 ID=61118 PROTO=ICMP TYPE=0 CODE=0 ID=30426 SEQ=1
TRACE: mangle:PREROUTING:policy:1 IN=eth0 OUT= MAC=00:16:3e:06:3a:3a:00:2a:6a:aa:12:7c:08:00 SRC=10.28.61.30 DST=10.171.77.0 LEN=84 TOS=0x00 PREC=0x00 TTL=57 ID=61118 PROTO=ICMP TYPE=0 CODE=0 ID=30426 SEQ=1
TRACE: mangle:INPUT:policy:1 IN=eth0 OUT= MAC=00:16:3e:06:3a:3a:00:2a:6a:aa:12:7c:08:00 SRC=10.28.61.30 DST=10.171.77.0 LEN=84 TOS=0x00 PREC=0x00 TTL=57 ID=61118 PROTO=ICMP TYPE=0 CODE=0 ID=30426 SEQ=1
TRACE: filter:INPUT:policy:1 IN=eth0 OUT= MAC=00:16:3e:06:3a:3a:00:2a:6a:aa:12:7c:08:00 SRC=10.28.61.30 DST=10.171.77.0 LEN=84 TOS=0x00 PREC=0x00 TTL=57 ID=61118 PROTO=ICMP TYPE=0 CODE=0 ID=30426 SEQ=1
```

可以对照着全图看出在request出去时，发现OUT设备不是bridge，直接走network layer的iptables rules，并从xfrm lookup出去，走到egress(qdisc); response回来时，进行bridge check后，发现IN设备eth0不是bridge，因此直接上到network layer，走iptables chain rules到local process。ebtables的log一行也没有输出。

后续的两个icmp request&response大致相同，并且依旧不走nat PREROUTING和nat POSTROUTING，因为不再是NEW connection。

五、Container to External



我们在c1 容器中ping 外部的一个节点三次:

```
# docker exec c1 ping -c 3 10.28.61.30
PING 10.28.61.30 (10.28.61.30) 56(84) bytes of data.
64 bytes from 10.28.61.30: icmp_seq=1 ttl=56 time=1.32 ms
64 bytes from 10.28.61.30: icmp_seq=2 ttl=56 time=1.30 ms
64 bytes from 10.28.61.30: icmp_seq=3 ttl=56 time=1.21 ms

--- 10.28.61.30 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 1.219/1.280/1.323/0.060 ms
```

1、start -> bridgecheck -> linker layer

和Container to Container的开端很类似，在bridge check后，数据流进入linker layer(docker0 is a bridge)，并在该层进行iptables PREROUTING rules的处理，直到bridge decision之前：

```
TRACE: eb:broute:BROUTING IN=veth0594f4b OUT= MAC source = 02:42:c0:a8:00:02 MAC dest = 02:42:ff:27:17:4d proto = 0x0800 IP SRC=192.168.0.2 IP DST=10.28.61.30, IP tos=0x00, IP proto=1
TRACE: eb:nat:PREROUTING IN=veth0594f4b OUT= MAC source = 02:42:c0:a8:00:02 MAC dest = 02:42:ff:27:17:4d proto = 0x0800 IP SRC=192.168.0.2 IP DST=10.28.61.30, IP tos=0x00, IP proto=1
TRACE: raw:PREROUTING:policy:2 IN=docker0 OUT= PHYSIN=veth0594f4b MAC=02:42:ff:27:17:4d:02:42:c0:a8:00:02:08:00 SRC=192.168.0.2 DST=10.28.61.30 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=57351 DF PROTO=ICMP 1
TRACE: mangle:PREROUTING:policy:1 IN=docker0 OUT= PHYSIN=veth0594f4b MAC=02:42:ff:27:17:4d:02:42:c0:a8:00:02:08:00 SRC=192.168.0.2 DST=10.28.61.30 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=57351 DF PROTO=ICMP 1
TRACE: nat:PREROUTING:policy:2 IN=docker0 OUT= PHYSIN=veth0594f4b MAC=02:42:ff:27:17:4d:02:42:c0:a8:00:02:08:00 SRC=192.168.0.2 DST=10.28.61.30 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=57351 DF PROTO=ICMP 1
```

2、ebtable filter:INPUT -> routing decision -> iptables FORWARD

目的地址为外部host ip，需要三层介入转发，于是数据包经由eb:filter:INPUT向上走到达network layer的routing decision，根据路由表，将包转发到eth0：

```
TRACE: mangle:FORWARD:policy:1 IN=docker0 OUT=eth0 PHYSIN=veth0594f4b MAC=02:42:ff:27:17:4d:02:42:c0:a8:00:02:08:00 SRC=192.168.0.2 DST=10.28.61.30 LEN=84 TOS=0x00 PREC=0x00 TTL=63 ID=57351 DF PROTO=ICMP
TRACE: filter:FORWARD:rule:1 IN=docker0 OUT=eth0 PHYSIN=veth0594f4b MAC=02:42:ff:27:17:4d:02:42:c0:a8:00:02:08:00 SRC=192.168.0.2 DST=10.28.61.30 LEN=84 TOS=0x00 PREC=0x00 TTL=63 ID=57351 DF PROTO=ICMP
TRACE: filter:DOCKER-USER:return:1 IN=docker0 OUT=eth0 PHYSIN=veth0594f4b MAC=02:42:ff:27:17:4d:02:42:c0:a8:00:02:08:00 SRC=192.168.0.2 DST=10.28.61.30 LEN=84 TOS=0x00 PREC=0x00 TTL=63 ID=57351 DF PROTO=ICMP
TRACE: filter:FORWARD:rule:2 IN=docker0 OUT=eth0 PHYSIN=veth0594f4b MAC=02:42:ff:27:17:4d:02:42:c0:a8:00:02:08:00 SRC=192.168.0.2 DST=10.28.61.30 LEN=84 TOS=0x00 PREC=0x00 TTL=63 ID=57351 DF PROTO=ICMP
TRACE: filter:DOCKER-ISOLATION:return:1 IN=docker0 OUT=eth0 PHYSIN=veth0594f4b MAC=02:42:ff:27:17:4d:02:42:c0:a8:00:02:08:00 SRC=192.168.0.2 DST=10.28.61.30 LEN=84 TOS=0x00 PREC=0x00 TTL=63 ID=57351 DF PROTO=ICMP
TRACE: filter:FORWARD:rule:5 IN=docker0 OUT=eth0 PHYSIN=veth0594f4b MAC=02:42:ff:27:17:4d:02:42:c0:a8:00:02:08:00 SRC=192.168.0.2 DST=10.28.61.30 LEN=84 TOS=0x00 PREC=0x00 TTL=63 ID=57351 DF PROTO=ICMP
```

3、iptables nat:POSTROUTING match rule 1

由于要流出到主机外，因此在最后iptables nat:POSTROUTING中，数据包匹配到rule 1，即做MASQUERADE，将数据包源地址更换为host ip: 10.171.77.0。

```
TRACE: mangle:POSTROUTING:policy:1 IN= OUT=eth0 PHYSIN=veth0594f4b SRC=192.168.0.2 DST=10.28.61.30 LEN=84 TOS=0x00 PREC=0x00 TTL=63 ID=57351 DF PROTO=ICMP TYPE=8 CODE=0 ID=94 SEQ=1
TRACE: nat:POSTROUTING:rule:1 IN= OUT=eth0 PHYSIN=veth0594f4b SRC=192.168.0.2 DST=10.28.61.30 LEN=84 TOS=0x00 PREC=0x00 TTL=63 ID=57351 DF PROTO=ICMP TYPE=8 CODE=0 ID=94 SEQ=1
```

4、iptables prerouting、forward、postrouting -> ebtables output、postrouting

返回的应答由于IN设备为eth0，因此直接上到network layer进行iptables chain的处理。在路由后，OUT设备为docker0(bridge设备)，因此在最后的环节需要下降到linker layer做output和postrouting处理：

```
TRACE: raw:PREROUTING:policy:2 IN=eth0 OUT= MAC=00:16:3e:06:3a:3a:00:2a:6a:aa:12:7c:08:00 SRC=10.28.61.30 DST=10.171.77.0 LEN=84 TOS=0x00 PREC=0x00 TTL=57 ID=58706 PROTO=ICMP TYPE=0 CODE=0 ID=94 SEQ=1
TRACE: mangle:PREROUTING:policy:1 IN=eth0 OUT= MAC=00:16:3e:06:3a:3a:00:2a:6a:aa:12:7c:08:00 SRC=10.28.61.30 DST=10.171.77.0 LEN=84 TOS=0x00 PREC=0x00 TTL=57 ID=58706 PROTO=ICMP TYPE=0 CODE=0 ID=94 SEQ=1
TRACE: mangle:FORWARD:policy:1 IN=eth0 OUT=docker0 MAC=00:16:3e:06:3a:3a:00:2a:6a:aa:12:7c:08:00 SRC=10.28.61.30 DST=192.168.0.2 LEN=84 TOS=0x00 PREC=0x00 TTL=56 ID=58706 PROTO=ICMP TYPE=0 CODE=0 ID=94 SEQ=1
TRACE: filter:FORWARD:rule:1 IN=eth0 OUT=docker0 MAC=00:16:3e:06:3a:3a:00:2a:6a:aa:12:7c:08:00 SRC=10.28.61.30 DST=192.168.0.2 LEN=84 TOS=0x00 PREC=0x00 TTL=56 ID=58706 PROTO=ICMP TYPE=0 CODE=0 ID=94 SEQ=1
TRACE: filter:DOCKER-USER:return:1 IN=eth0 OUT=docker0 MAC=00:16:3e:06:3a:3a:00:2a:6a:aa:12:7c:08:00 SRC=10.28.61.30 DST=192.168.0.2 LEN=84 TOS=0x00 PREC=0x00 TTL=56 ID=58706 PROTO=ICMP TYPE=0 CODE=0 ID=94 SEQ=1
TRACE: filter:FORWARD:rule:2 IN=eth0 OUT=docker0 MAC=00:16:3e:06:3a:3a:00:2a:6a:aa:12:7c:08:00 SRC=10.28.61.30 DST=192.168.0.2 LEN=84 TOS=0x00 PREC=0x00 TTL=56 ID=58706 PROTO=ICMP TYPE=0 CODE=0 ID=94 SEQ=1
TRACE: filter:DOCKER-ISOLATION:return:1 IN=eth0 OUT=docker0 MAC=00:16:3e:06:3a:3a:00:2a:6a:aa:12:7c:08:00 SRC=10.28.61.30 DST=192.168.0.2 LEN=84 TOS=0x00 PREC=0x00 TTL=56 ID=58706 PROTO=ICMP TYPE=0 CODE=0 ID=94 SEQ=1
TRACE: filter:FORWARD:rule:3 IN=eth0 OUT=docker0 MAC=00:16:3e:06:3a:3a:00:2a:6a:aa:12:7c:08:00 SRC=10.28.61.30 DST=192.168.0.2 LEN=84 TOS=0x00 PREC=0x00 TTL=56 ID=58706 PROTO=ICMP TYPE=0 CODE=0 ID=94 SEQ=1
TRACE: mangle:POSTROUTING:policy:1 IN= OUT=docker0 SRC=10.28.61.30 DST=192.168.0.2 LEN=84 TOS=0x00 PREC=0x00 TTL=56 ID=58706 PROTO=ICMP TYPE=0 CODE=0 ID=94 SEQ=1
TRACE: eb:nat:OUTPUT IN= OUT=veth0594f4b MAC source = 02:42:ff:27:17:4d MAC dest = 02:42:c0:a8:00:02 proto = 0x0800 IP SRC=10.28.61.30 IP DST=192.168.0.2, IP tos=0x00, IP proto=1
TRACE: eb:filter:OUTPUT IN= OUT=veth0594f4b MAC source = 02:42:ff:27:17:4d MAC dest = 02:42:c0:a8:00:02 proto = 0x0800 IP SRC=10.28.61.30 IP DST=192.168.0.2, IP tos=0x00, IP proto=1
TRACE: eb:nat:POSTROUTING IN= OUT=veth0594f4b MAC source = 02:42:ff:27:17:4d MAC dest = 02:42:c0:a8:00:02 proto = 0x0800 IP SRC=10.28.61.30 IP DST=192.168.0.2, IP tos=0x00, IP proto=1
```

后续的请求和应答基本类似，少的还是nat PREROUTING和nat POSTROUTING，因为不再是NEW connection。

六、小结

个人赶脚：iptables的规则还是太复杂了，再加上bridge的ebtable规则，让人有些眼花缭乱。尤其是kube-proxy的规则又与docker的规则糅合在一起，iptables的rules列表就显得更为冗长和复杂了。但目前kube-proxy稳定版依然以iptables为主要实现机制，不过kube-proxy对ipvs的支持也已经在路上了(kubernetes 1.8中ipvs处于alpha阶段)，希望后续我们能有更多的选择。

此次实验全部日志内容参见：[docker-bridge-network-demo-iptables-trace-log.txt文件](#)。

七、参考资料

- 《[iptables debugging](#)》
- 《[ebtables/iptables interaction on a Linux-based bridge](#)》
- 《[Traversing of tables and chains](#)》
- 《[Linux Bridge – how it works](#)》
- “[docker-explain network](#)”
- 《[Linux下的虚拟Bridge实现](#)》

微博：[@tonybai](#) [cn](#)

微信公众号：iamtonybai

github.com: <https://github.com/bigwhite>

© 2017, [bigwhite](#). 版权所有.

Related posts:

- 1. [理解Docker容器端口映射](#)
- 2. [理解Docker单机容器网络](#)
- 3. [理解Kubernetes网络之Flannel网络](#)
- 4. [理解Docker跨多主机容器网络](#)
- 5. [理解Docker容器网络之Linux Network Namespace](#)

添加新评论

称呼

邮箱

网站

提交评论

Capatcha

如发现本站页面被黑，比如：挂载广告、挖矿等恶意代码，请朋友们及时[联系我](#)。十分感谢！

赞助商广告位1

图片广告+链接跳转

广告首页展示，欢迎合作

商务合作请联系bigwhite.cn AT aliyun.com

欢迎使用邮件订阅我的博客

输入邮箱订阅本站，只要有新文章发布，就会第一时间发送邮件通知你哦！

名字:

邮箱:

马上订阅



这里是 [Tony Bai](#)的个人Blog，欢迎访问、订阅和留言！ 订阅Feed请点击上面图片。

如果您觉得这里的文章对您有帮助，请扫描上方二维码进行捐赠，加油后的Tony Bai将会为您呈现更多精彩的文章，谢谢！

如果您希望通过微信捐赠，请用微信客户端扫描下方二维码：



如果您希望通过比特币或以太币捐赠，可以扫描下方二维码：

比特币：



以太坊：



如果您喜欢通过微信浏览本站内容，可以扫描下方二维码，订阅本站官方微信订阅号“iamtonybai”；点击二维码，可直达本人官方微博主页^_^：



本站Powered by Digital Ocean VPS。

[选择Digital Ocean VPS主机，即可获得10美元现金充值，可免费使用两个月哟！](#) 著名主机提供商Linode 10\$优惠码：linode10，在 [这里注册](#)即可免费获得。阿里云推荐码：**1WFZ0V**，**立享9折！**






我的业余项目

- [smspush短信发送平台](#)

文章

- [Go modules：最小版本选择](#)
- [Kubernetes Deployment故障排除图解指南](#)
- [计算重现性：一些挑战](#)
- [Go官方发布的go.dev给gopher们带来了什么](#)
- [Go语言开源十周年](#)
- [Go语言项目的安全评估技术](#)
- [图解中文字符编码-Go语言例解](#)
- [Go语言的遗产](#)
- [Go 1.13中值得关注的几个变化](#)
- [如何在Ubuntu 18.04 Server上部署Kubernetes集群](#)

评论

-  bigwhite 在 [Go语言TCP Socket编程](#)
这个就是Wordpress自带的功能啊。
-  PierreVon 在 [Go语言TCP Socket编程](#)
你这个评论回复功能做的好厉害，请问是用的sql还是nosql存储的评论，前端是怎么渲染的呀？
-  爱玩编程的大二学弟 在 [也谈Go的可移植性](#)
受益匪浅

-  [Chris](#) 在 [关于我](#)
你好，请问是否能交换博客链接？ <https://jingine.com>
-  [yyli](#) 在 [理解Golang包导入](#)
謝謝樓主
-  [BillZong](#) 在 [Go modules：最小版本选择](#)
翻译很准确到位，谢谢又Get到不少知识点
-  [bigwhite](#) 在 [如何在Go语言中使用Websockets：最佳工具与行动指南](#)
文中有标注：这是一篇译文。原文在<https://yalantis.com/blog/how-to-b...>
-  [wei](#) 在 [如何在Go语言中使用Websockets：最佳工具与行动指南](#)
本文的源码能否公开一下？有些片段文中没有展示.....
-  [lostpg](#) 在 [也谈goroutine调度器](#)
请问 G 的抢占调度那部分，「可以看出，如果一个 G 任务运行 10ms，sysmon 就会认为其运...
-  [bigwhite](#) 在 [HTTPS服务的Kubernetes ingress配置实践](#)
<https://github.com/bigwhite/experiments>下面的ingress-...
- [下一页 »](#)

分类

- [光影汇](#) (7)
- [影音坊](#) (36)
- [思考控](#) (66)
- [技术志](#) (601)
- [教育记](#) (1)
- [杂货铺](#) (75)
- [生活簿](#) (154)
- [职场录](#) (14)
- [读书吧](#) (14)
- [运动迷](#) (107)
- [驴友秀](#) (40)

标签

[Blog](#) [Blogger](#) [C](#) [container](#) [C++](#) [docker](#) [GCC](#) [github](#) [GNU](#) [Go](#) [Golang](#) [Google](#) [Java](#) [k8s](#) [Kernel](#) [Kubernetes](#) [Linux](#) [M10](#) [Opensource](#) [Programmer](#) [Python](#) [Solaris](#) [Subversion](#) [Ubuntu](#) [Unix](#) [Windows](#) [世界杯](#) [博](#)
[客](#) [学习](#) [容器](#) [工作](#) [巴萨](#) [开源](#) [思考](#) [感悟](#) [摄影](#) [旅游](#) [梅西](#) [球王](#) [生活](#) [程序员](#) [编译器](#) [翻译](#) [西甲](#) [足球](#)

归档

- [2019 年十二月](#) (2)
- [2019 年十一月](#) (6)
- [2019 年十月](#) (5)
- [2019 年九月](#) (4)
- [2019 年八月](#) (5)
- [2019 年七月](#) (1)
- [2019 年六月](#) (2)
- [2019 年五月](#) (1)
- [2019 年四月](#) (4)
- [2019 年三月](#) (2)
- [2019 年二月](#) (1)
- [2019 年一月](#) (2)
- [2018 年十一月](#) (3)
- [2018 年十月](#) (1)
- [2018 年九月](#) (1)
- [2018 年七月](#) (1)
- [2018 年六月](#) (4)
- [2018 年五月](#) (2)
- [2018 年四月](#) (1)
- [2018 年三月](#) (3)
- [2018 年二月](#) (3)
- [2018 年一月](#) (7)
- [2017 年十二月](#) (5)
- [2017 年十一月](#) (4)
- [2017 年十月](#) (3)
- [2017 年九月](#) (2)
- [2017 年八月](#) (3)
- [2017 年七月](#) (4)
- [2017 年六月](#) (8)
- [2017 年五月](#) (5)
- [2017 年四月](#) (3)
- [2017 年三月](#) (2)
- [2017 年二月](#) (5)
- [2017 年一月](#) (7)
- [2016 年十二月](#) (7)
- [2016 年十一月](#) (7)
- [2016 年十月](#) (3)
- [2016 年九月](#) (2)
- [2016 年八月](#) (1)
- [2016 年六月](#) (2)
- [2016 年五月](#) (2)
- [2016 年四月](#) (2)
- [2016 年三月](#) (2)
- [2016 年二月](#) (3)
- [2016 年一月](#) (2)
- [2015 年十二月](#) (1)
- [2015 年十一月](#) (1)

- [2015 年十月](#) (1)
- [2015 年九月](#) (3)
- [2015 年八月](#) (5)
- [2015 年七月](#) (6)
- [2015 年六月](#) (4)
- [2015 年五月](#) (1)
- [2015 年四月](#) (2)
- [2015 年三月](#) (2)
- [2015 年一月](#) (2)
- [2014 年十二月](#) (5)
- [2014 年十一月](#) (8)
- [2014 年十月](#) (9)
- [2014 年九月](#) (2)
- [2014 年八月](#) (1)
- [2014 年七月](#) (1)
- [2014 年五月](#) (2)
- [2014 年四月](#) (5)
- [2014 年三月](#) (4)
- [2014 年二月](#) (1)
- [2014 年一月](#) (1)
- [2013 年十二月](#) (3)
- [2013 年十一月](#) (5)
- [2013 年十月](#) (6)
- [2013 年九月](#) (4)
- [2013 年八月](#) (5)
- [2013 年七月](#) (6)
- [2013 年六月](#) (2)
- [2013 年五月](#) (6)
- [2013 年四月](#) (3)
- [2013 年三月](#) (7)
- [2013 年二月](#) (4)
- [2013 年一月](#) (6)
- [2012 年十二月](#) (8)
- [2012 年十一月](#) (10)
- [2012 年十月](#) (5)
- [2012 年九月](#) (3)
- [2012 年八月](#) (10)
- [2012 年七月](#) (4)
- [2012 年六月](#) (2)
- [2012 年五月](#) (4)
- [2012 年四月](#) (10)
- [2012 年三月](#) (8)
- [2012 年二月](#) (6)
- [2012 年一月](#) (6)
- [2011 年十二月](#) (4)
- [2011 年十一月](#) (4)
- [2011 年十月](#) (5)
- [2011 年九月](#) (8)
- [2011 年八月](#) (7)

- [2011 年七月](#) (6)
- [2011 年六月](#) (7)
- [2011 年五月](#) (8)
- [2011 年四月](#) (6)
- [2011 年三月](#) (10)
- [2011 年二月](#) (7)
- [2011 年一月](#) (10)
- [2010 年十二月](#) (7)
- [2010 年十一月](#) (6)
- [2010 年十月](#) (7)
- [2010 年九月](#) (12)
- [2010 年八月](#) (8)
- [2010 年七月](#) (3)
- [2010 年六月](#) (5)
- [2010 年五月](#) (4)
- [2010 年四月](#) (2)
- [2010 年三月](#) (6)
- [2010 年二月](#) (4)
- [2010 年一月](#) (6)
- [2009 年十二月](#) (6)
- [2009 年十一月](#) (6)
- [2009 年十月](#) (5)
- [2009 年九月](#) (8)
- [2009 年八月](#) (8)
- [2009 年七月](#) (8)
- [2009 年六月](#) (2)
- [2009 年五月](#) (5)
- [2009 年四月](#) (7)
- [2009 年三月](#) (12)
- [2009 年二月](#) (9)
- [2009 年一月](#) (15)
- [2008 年十二月](#) (9)
- [2008 年十一月](#) (5)
- [2008 年十月](#) (10)
- [2008 年九月](#) (13)
- [2008 年八月](#) (13)
- [2008 年七月](#) (3)
- [2008 年六月](#) (1)
- [2008 年五月](#) (7)
- [2008 年四月](#) (4)
- [2008 年三月](#) (9)
- [2008 年二月](#) (11)
- [2008 年一月](#) (15)
- [2007 年十二月](#) (11)
- [2007 年十一月](#) (14)
- [2007 年十月](#) (4)
- [2007 年九月](#) (5)
- [2007 年八月](#) (1)
- [2007 年七月](#) (10)

- [2007 年六月](#) (10)
- [2007 年五月](#) (10)
- [2007 年四月](#) (8)
- [2007 年三月](#) (15)
- [2007 年二月](#) (4)
- [2007 年一月](#) (17)
- [2006 年十二月](#) (18)
- [2006 年十一月](#) (9)
- [2006 年十月](#) (11)
- [2006 年九月](#) (6)
- [2006 年八月](#) (5)
- [2006 年七月](#) (22)
- [2006 年六月](#) (35)
- [2006 年五月](#) (24)
- [2006 年四月](#) (26)
- [2006 年三月](#) (25)
- [2006 年二月](#) (18)
- [2006 年一月](#) (15)
- [2005 年十二月](#) (10)
- [2005 年十一月](#) (10)
- [2005 年九月](#) (13)
- [2005 年八月](#) (11)
- [2005 年七月](#) (6)
- [2005 年六月](#) (2)
- [2005 年五月](#) (3)
- [2005 年四月](#) (6)
- [2005 年三月](#) (1)
- [2005 年一月](#) (15)
- [2004 年十二月](#) (9)
- [2004 年十一月](#) (14)
- [2004 年十月](#) (2)
- [2004 年九月](#) (2)

私人

- [我的女儿](#)

链接

- [@douban](#)
- [@flickr](#)
- [@github](#)
- [@googlecode](#)
- [@picasa](#)
- [@slideshare](#)
- [@twitter](#)
- [@weibo](#)
- [Hoterran](#)
- [Lionel Messi](#)

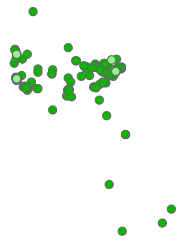
- [Puras He](#)
- [梦想风暴](#)
- [磊磊落落的博客](#)
- [过眼云烟](#)

开源项目

- [buildc](#)
- [cbehave](#)
- [lcut](#)

翻译项目

- [C语言编码风格和标准](#)
- [《Programming in Haskell》中文翻译项目](#)



02261572 [View My Stats](#)

© 2020 [Tony Bai](#). 由 [Wordpress](#) 强力驱动. 模板由[cho](#)制作.