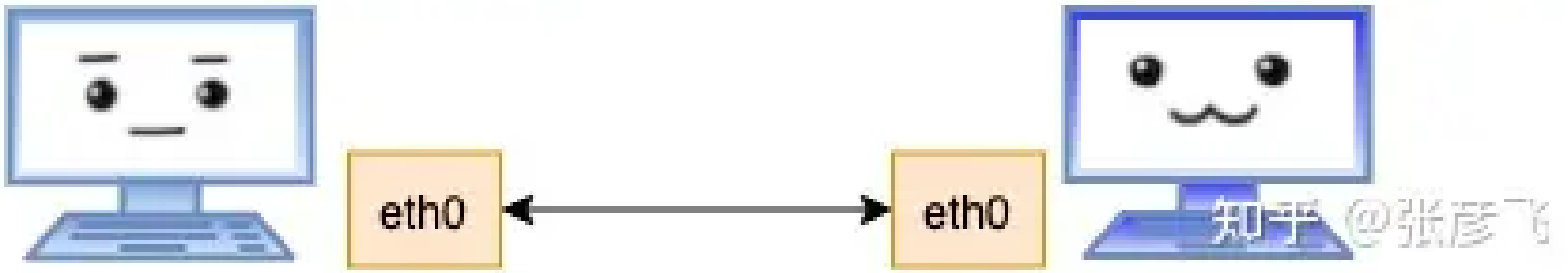
轻松理解 Docker 网络虚拟化基础之 veth 设备！

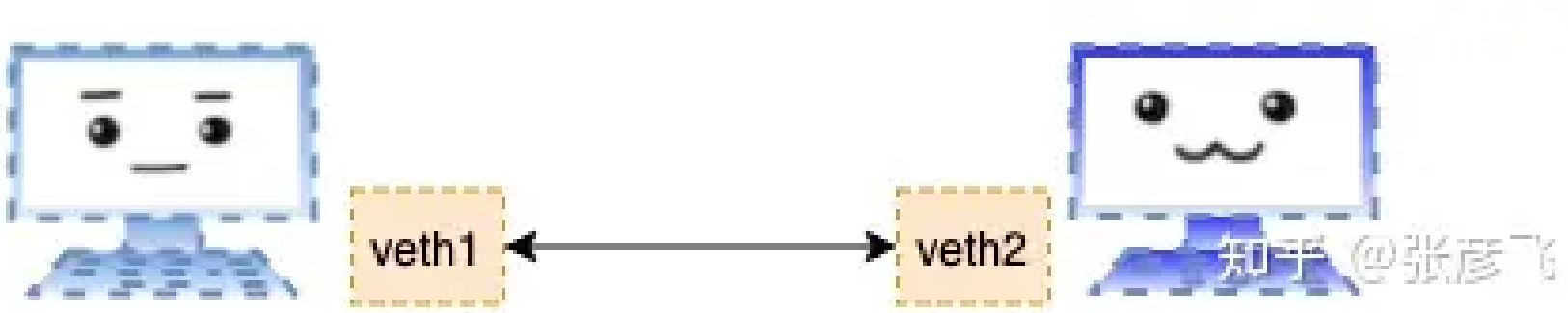
张彦飞

正如我在朋友圈里所说的，最近我又对网络虚拟化技术产生了浓厚的兴趣。迫切想搞明白在 Docker 等虚拟技术下，网络底层是如何运行的。今天我给大家带来的就是 Docker 网络虚拟化中的一个比较好理解的技术 - **veth**。

回想下在物理机组成的网络里，最基础，最简单的网络连接方式是什么？没错，那就是直接用一根交叉网线把两台电脑的网卡连起来。这样，一台机器发送数据，另外一台就能收到了。



那么，网络虚拟化实现的第一步，就是用软件来模拟这个简单的网络连接实现过程。实现的技术就是我们今天的主角 veth，它模拟了在物理世界里的两块网卡，以及一条网线。通过它可以将两个虚拟的设备连接起来，让他们之间相互通信。平时工作中在 Docker 镜像里我们看到的 eth0 设备，其实就是 veth。



事实上，这种软件模拟硬件方式我们一点儿也不陌生，我们本机网络 IO 里的 lo 回环设备也是这样一个用软件虚拟出来设备。 Veth 和 lo 的一点区别就是 veth 总是成双成对地出现。

我们今天就来深入地看看 veth 这个东东是咋工作的。

一、veth 如何使用

不像回环设备，绝大多数同学在日常工作中可能都没接触过 veth。所以本文咱们专门拉一小节出来介绍 veth 是如何使用的。

在 Linux 下，我们可以通过使用 ip 命令创建一对儿 veth。其中 link 表示 link layer的意思，即链路层。这个命令可以用于管理和查看网络接口，包括物理网络接口，也包括虚拟接口。

```
# ip link add veth0 type veth peer name veth1
```

使用 ip link show 来进行查看。

```
# ip link add veth0 type veth peer name veth1
# ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT
```

```
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT qlen 1000
    link/ether 6c:0b:84:d5:88:d1 brd ff:ff:ff:ff:ff:ff
3: eth1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT qlen 1000
    link/ether 6c:0b:84:d5:88:d2 brd ff:ff:ff:ff:ff:ff
4: veth1@veth0: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT qlen 1000
    link/ether 4e:ac:33:e5:eb:16 brd ff:ff:ff:ff:ff:ff
5: veth0@veth1: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT qlen 1000
    link/ether 2a:6d:65:74:30:fb brd ff:ff:ff:ff:ff:ff
```

和 eth0、lo 等网络设备一样，veth 也需要为其配置上 ip 后才能够正常工作。我们为这对儿 veth 分别来配置上 IP。

```
# ip addr add 192.168.1.1/24 dev veth0
# ip addr add 192.168.1.2/24 dev veth1
```

接下来，我们把这两个设备启动起来。

```
# ip link set veth0 up
# ip link set veth1 up
```

当设备启动起来以后，我们通过我们熟悉的 ifconfig 就可以查看到它们了。

```
# ifconfig
eth0: ......
lo: ......
veth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.1  netmask 255.255.255.0  broadcast 0.0.0.0
        ......
```

```
veth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.2  netmask 255.255.255.0  broadcast 0.0.0.0
        ......
```

现在，一对儿虚拟设备已经建立起来了。不过我们需要做一点准备工作，它们之间才可以进行互相通信。首先要关闭反向过滤 rp_filter，该模块会检查 IP 包是否符合要求，否则可能会过滤掉。然后再打开 accept_local，接收本机 IP 数据包。详细准备过程如下：

```
# echo 0 > /proc/sys/net/ipv4/conf/all/rp_filter
# echo 0 > /proc/sys/net/ipv4/conf/veth0/rp_filter
# echo 0 > /proc/sys/net/ipv4/conf/veth1/rp_filter
# echo 1 > /proc/sys/net/ipv4/conf/veth1/accept_local
# echo 1 > /proc/sys/net/ipv4/conf/veth0/accept_local
```

好了，我们在 veth0 上来 ping 一下 veth1。这两个 veth 之间可以通信了，欧耶！

```
# ping 192.168.1.2 -I veth0
PING 192.168.1.2 (192.168.1.2) from 192.168.1.1 veth0: 56(84) bytes of data.
64 bytes from 192.168.1.2: icmp_seq=1 ttl=64 time=0.019 ms
64 bytes from 192.168.1.2: icmp_seq=2 ttl=64 time=0.010 ms
64 bytes from 192.168.1.2: icmp_seq=3 ttl=64 time=0.010 ms
...
```

我在另外一个控制台上，还启动了 tcpdump 抓包，抓到的结果如下。

```
# tcpdump -i veth0
09:59:39.449247 ARP, Request who-has *** tell ***, length 28
09:59:39.449259 ARP, Reply *** is-at 4e:ac:33:e5:eb:16 (oui Unknown), length 28
```

```
09:59:39.449262 IP *** > ***: ICMP echo request, id 15841, seq 1, length 64
09:59:40.448689 IP *** > ***: ICMP echo request, id 15841, seq 2, length 64
09:59:41.448684 IP *** > ***: ICMP echo request, id 15841, seq 3, length 64
09:59:42.448687 IP *** > ***: ICMP echo request, id 15841, seq 4, length 64
09:59:43.448686 IP *** > ***: ICMP echo request, id 15841, seq 5, length 64
```

由于两个设备之间是首次通信的，所以 veth0 首先先发出了一个 arp request，veth1 收到后回复了一个 arp reply。然后接下来就是正常的 ping 命令下的 IP 包了。

二、veth 底层创建过程

在上一小节中，我们亲手创建了一对儿 veth 设备，并通过简单的配置就可以让他们之间互相进行通信了。那么在本小节中，我们看看在内核里，veth 到底是如何创建的。

Veth 相关源码位于 drivers/net/veth.c，其中初始化入口是 veth_init。

```
//file: drivers/net/veth.c
static __init int veth_init(void)
{
 return rtnl_link_register(&veth_link_ops);
}
```

在 veth_init 中注册了 veth_link_ops（veth 设备的操作方法），它包含了 veth 设备的创建、启动和删除等回调函数。

```
//file: drivers/net/veth.c
static struct rtnl_link_ops veth_link_ops = {
 .kind  = DRV_NAME,
```

```
 .priv_size = sizeof(struct veth_priv),
 .setup  = veth_setup,
 .validate = veth_validate,
 .newlink = veth_newlink,
 .dellink = veth_dellink,
 .policy  = veth_policy,
 .maxtype = VETH_INFO_MAX,
};
```

我们先来看下 veth 设备的创建函数 veth_newlink，**这是理解 veth 的关键之处。**

```
//file: drivers/net/veth.c
static int veth_newlink(struct net *src_net, struct net_device *dev,
    struct nlattr *tb[], struct nlattr *data[])
{
 ...
 //创建
 peer = rtnl_create_link(net, ifname, &veth_link_ops, tbp);

 //注册
 err = register_netdevice(peer);
 err = register_netdevice(dev);
 ...

 //把两个设备关联到一起
 priv = netdev_priv(dev);
 rcu_assign_pointer(priv->peer, peer);

 priv = netdev_priv(peer);
```

```
 rcu_assign_pointer(priv->peer, dev);
}
```

在 veth_newlink 中，我们看到它通过 register_netdevice 创建了 peer 和 dev 两个网络虚拟设备。接下来的 netdev_priv 函数返回的是网络设备的 private 数据，priv->peer 就是一个指针而已。

```
//file: drivers/net/veth.c
struct veth_priv {
 struct net_device __rcu *peer;
 atomic64_t  dropped;
};
```

两个新创建出来的设备 dev 和 peer 通过 priv->peer 指针来完成结对。其中 dev 设备里的 priv->peer 指针指向 peer 设备， peer 设备里的 priv->peer 指向 dev。

接着我们再看下 veth 设备的启动过程。

```
//file: drivers/net/veth.c
static void veth_setup(struct net_device *dev)
{
 //veth的操作列表，其中包括veth的发送函数veth_xmit
 dev->netdev_ops = &veth_netdev_ops;
 dev->ethtool_ops = &veth_ethtool_ops;
 ......
}
```

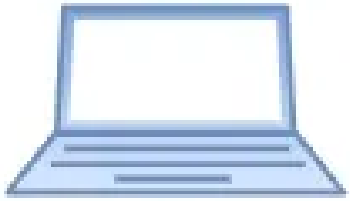
其中 `dev->netdev_ops = &veth_netdev_ops` 这行也比较关键。veth_netdev_ops 是 veth 设备的操作函数。例如发送过程中调用的函数指针 ndo_start_xmit，对于 veth 设备来说就会调用到 veth_xmit。这个在下一个小节里我们会用到。

```
//file: drivers/net/veth.c
static const struct net_device_ops veth_netdev_ops = {
 .ndo_init            = veth_dev_init,
 .ndo_open            = veth_open,
 .ndo_stop            = veth_close,
 .ndo_start_xmit      = veth_xmit,
 .ndo_change_mtu      = veth_change_mtu,
 .ndo_get_stats64     = veth_get_stats64,
 .ndo_set_mac_address = eth_mac_addr,
};
```

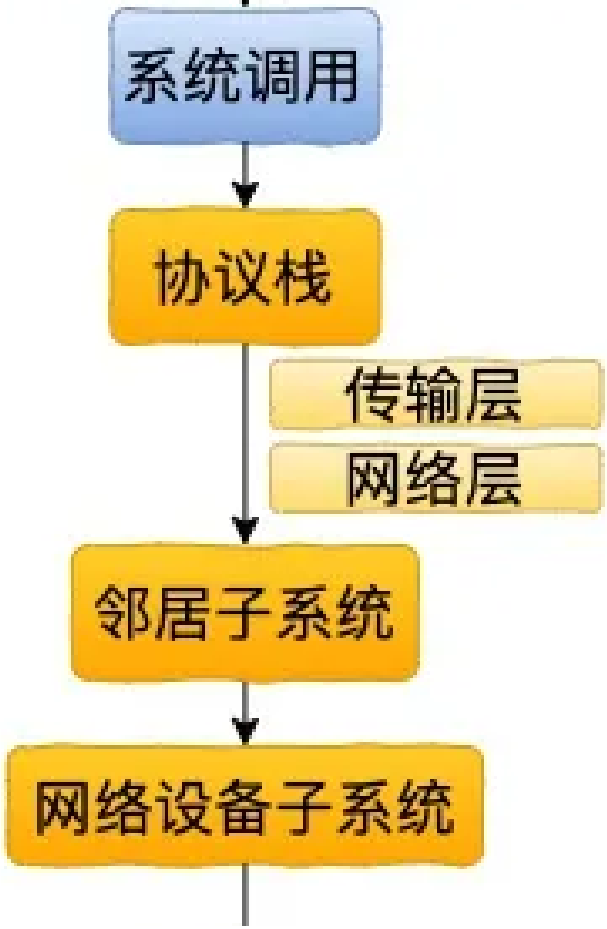
三、veth 网络通信过程

在**《25 张图，一万字，拆解 Linux 网络包发送过程》**和**《图解Linux网络包接收过程》**两篇文章中，我们系统介绍了 Linux 网络包的收发过程。 在**《127.0.0.1 之本机网络通信过程知多少 ?》**中我们又详细讨论了基于回环设备 lo 的本机网络 IO 过程。

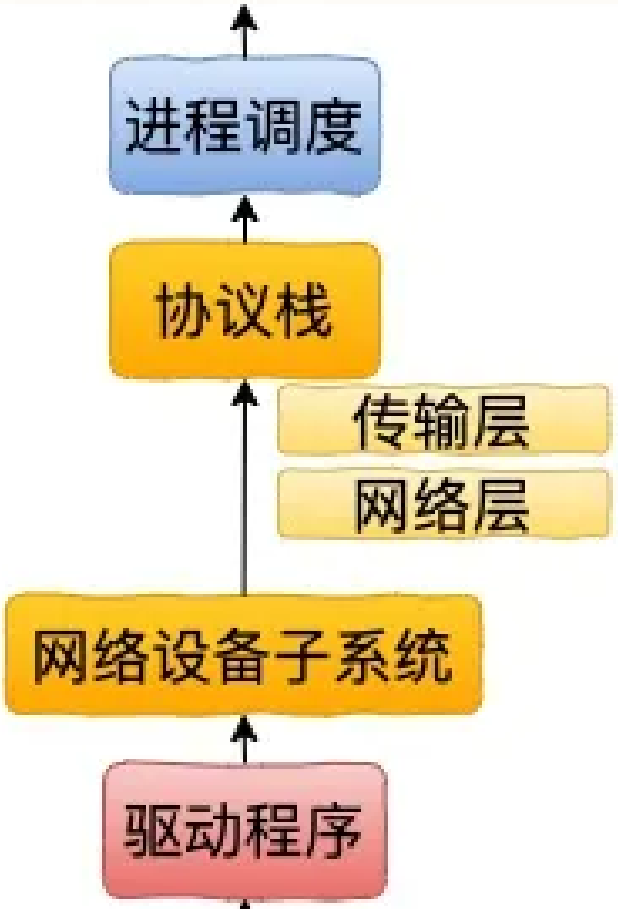
我们回顾一下基于回环设备 lo 的本机网络过程。在发送阶段里，流程分别是 send 系统调用 => 协议栈 => 邻居子系统 => 网络设备层 => 驱动。在接收阶段里，流程分别是软中断 => 驱动 => 网络设备层 => 协议栈 => 系统调用返回。过程图示如下：

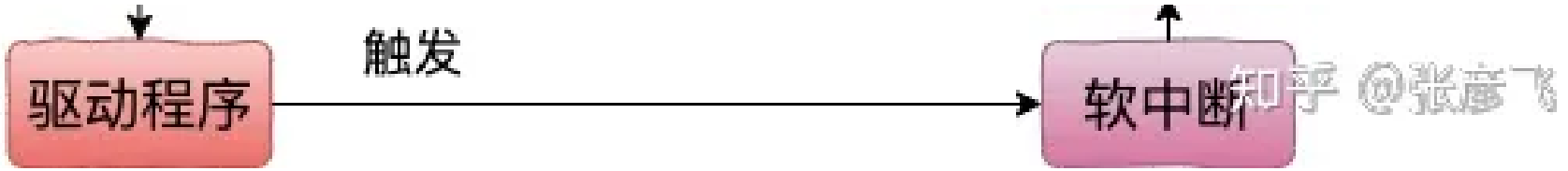


```c
int main(){
    send(fd, buf, ...);
}
```



```c
int main(){
    recvfrom(fd, buff, ...);
    printf("Received %s\n", buff);
}
```





基于 veth 的网络 IO 过程和上面这个过程图几乎完全一样。和 lo 设备所不同的就是使用的驱动程序不一样，马上我们就能看到。

网络设备层最后会通过 ops->ndo_start_xmit 来调用驱动进行真正的发送。

```
//file: net/core/dev.c
int dev_hard_start_xmit(struct sk_buff *skb, struct net_device *dev,
   struct netdev_queue *txq)
{
 //获取设备驱动的回调函数集合 ops
 const struct net_device_ops *ops = dev->netdev_ops;

 //调用驱动的 ndo_start_xmit 来进行发送
 rc = ops->ndo_start_xmit(skb, dev);
 ...
}
```

在**《127.0.0.1 之本机网络通信过程知多少 ?》**一文中，我们提到过对于回环设备 lo 来说 netdev_ops 是 loopback_ops。那么上面发送过程中调用的 ops->ndo_start_xmit 对应的就是 loopback_xmit。

```
//file:drivers/net/loopback.c
static const struct net_device_ops loopback_ops = {
 .ndo_init      = loopback_dev_init,
 .ndo_start_xmit= loopback_xmit,
```

```
 .ndo_get_stats64 = loopback_get_stats64,
};
```

回顾本文上一小节中，对于 veth 设备来说，它在启动的时候将 netdev_ops 设置成了 veth_netdev_ops。那 ops->ndo_start_xmit 对应的具体发送函数就是 veth_xmit。这就是在整个发送的过程中，唯一和 lo 设备不同的地方所在。我们来简单看一下这个发送函数的代码。

```
//file: drivers/net/veth.c
static netdev_tx_t veth_xmit(struct sk_buff *skb, struct net_device *dev)
{
 struct veth_priv *priv = netdev_priv(dev);
 struct net_device *rcv;

 //获取 veth 设备的对端
 rcv = rcu_dereference(priv->peer);

 //调用 dev_forward_skb 向对端发包
 if (likely(dev_forward_skb(rcv, skb) == NET_RX_SUCCESS)) {
 }
```

在 veth_xmit 中主要就是获取一下当前 veth 设备，然后向对端把数据发送过去就行了。发送到对端设备的工作是由 dev_forward_skb 函数来处理的。

```
//file: net/core/dev.c
int dev_forward_skb(struct net_device *dev, struct sk_buff *skb)
{
 skb->protocol = eth_type_trans(skb, dev);
 ...
```

```
 return netif_rx(skb);
}
```

先调用了 eth_type_trans 将 skb 的所属设备改为了刚刚取到的 veth 的对端设备 rcv。

```
//file: net/ethernet/eth.c
__be16 eth_type_trans(struct sk_buff *skb, struct net_device *dev)
{
 skb->dev = dev;
 ...
}
```

接着调用 netif_rx，这块又和 lo 设备的操作一样了。在该方法中最终会执行到 enqueue_to_backlog 中（netif_rx -> netif_rx_internal -> enqueue_to_backlog）。在这里将要发送的 skb 插入 softnet_data->input_pkt_queue 队列中并调用 ____napi_schedule 来触发软中断，见下面的代码。

```
//file: net/core/dev.c
static int enqueue_to_backlog(struct sk_buff *skb, int cpu,
          unsigned int *qtail)
{
 sd = &per_cpu(softnet_data, cpu);
 __skb_queue_tail(&sd->input_pkt_queue, skb);

 ...
 ____napi_schedule(sd, &sd->backlog);
}

//file:net/core/dev.c
```

```
static inline void ____napi_schedule(struct softnet_data *sd,
         struct napi_struct *napi)
{
 list_add_tail(&napi->poll_list, &sd->poll_list);
 __raise_softirq_irqoff(NET_RX_SOFTIRQ);
}
```

当数据发送完唤起软中断后，veth 对端的设备开始接收。 和发送过程不同的是，所有的虚拟设备的收包 poll 函数都是一样的，都是在设备层被初始化成了 process_backlog。

```
//file:net/core/dev.c
static int __init net_dev_init(void)
{
 for_each_possible_cpu(i) {
  sd->backlog.poll = process_backlog;
 }
}
```

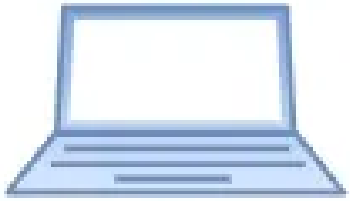
所以 veth 设备的接收过程和 lo 设备完全一样。想再看看这块过程的同学就请参考**《127.0.0.1 之本机网络通信过程知多少 ?》**一文中的第三节吧。大致流程是 net_rx_action 执行到 deliver_skb，然后送到协议栈中。

```
|--->net_rx_action()
    |--->process_backlog()
        |--->__netif_receive_skb()
            |--->__netif_receive_skb_core()
             |---> deliver_skb
```

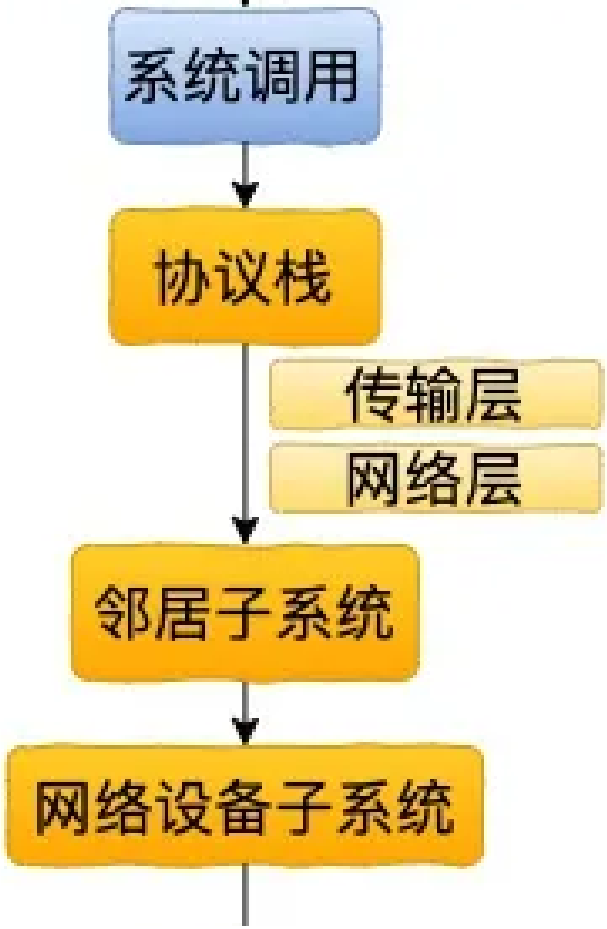
总结

由于大部分的同学在日常工作中一般不会接触到 veth，所以在看到 Docker 相关的技术文中提到这个技术时总会以为它是多么的高深。

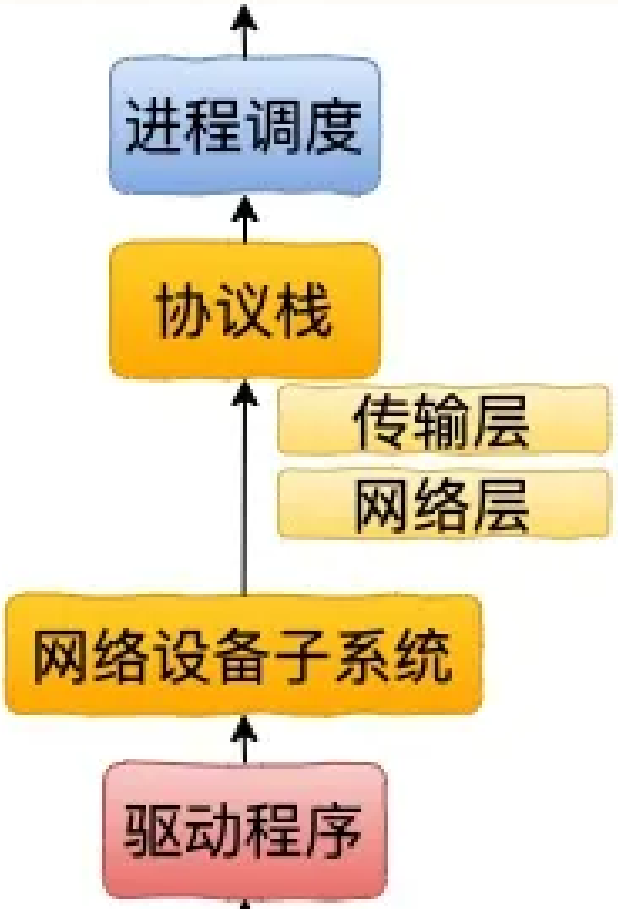
其实从实现上来看，虚拟设备 veth 和我们日常接触的 lo 设备非常非常的像。连基于 veth 的本机网络 IO 通信图其实都是我直接从 127.0.0.1 的那篇文章里复制过来的。只要你看过飞哥的**127.0.0.1 之本机网络通信过程知多少 ?!** 这篇，理解起来 veth 简直不要太容易。

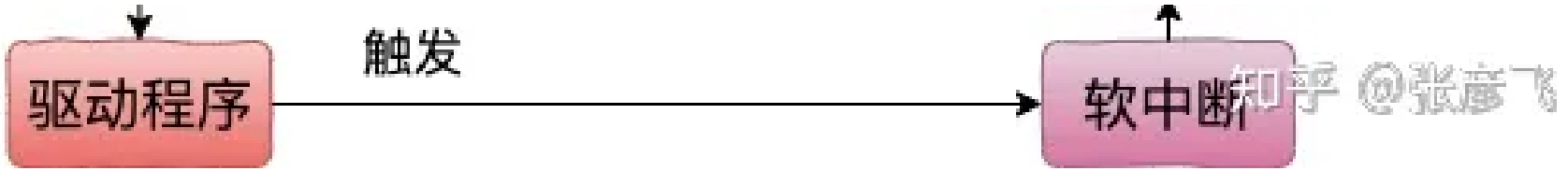


```c
int main(){
    send(fd, buf, ...);
}
```



```c
int main(){
    recvfrom(fd, buff, ...);
    printf("Received %s\n", buff);
}
```





只不过和 lo 设备相比，veth 是为了虚拟化技术而生的，所以它多了个结对的概念。在创建函数 veth_newlink 中，一次性就创建了两个网络设备出来，并把对方分别设置成了各自的 peer。在发送数据的过程中，找到发送设备的 peer，然后发起软中断让对方收取就算完事了。

怎么样，是不是 So easy！

欢迎前来光顾飞哥的技术圈子！

公众号：**开发内功修炼**

GitHub - yanfeizhang/coder-kung-fu: 开发内功修炼

发布于 2021-09-17 07:04

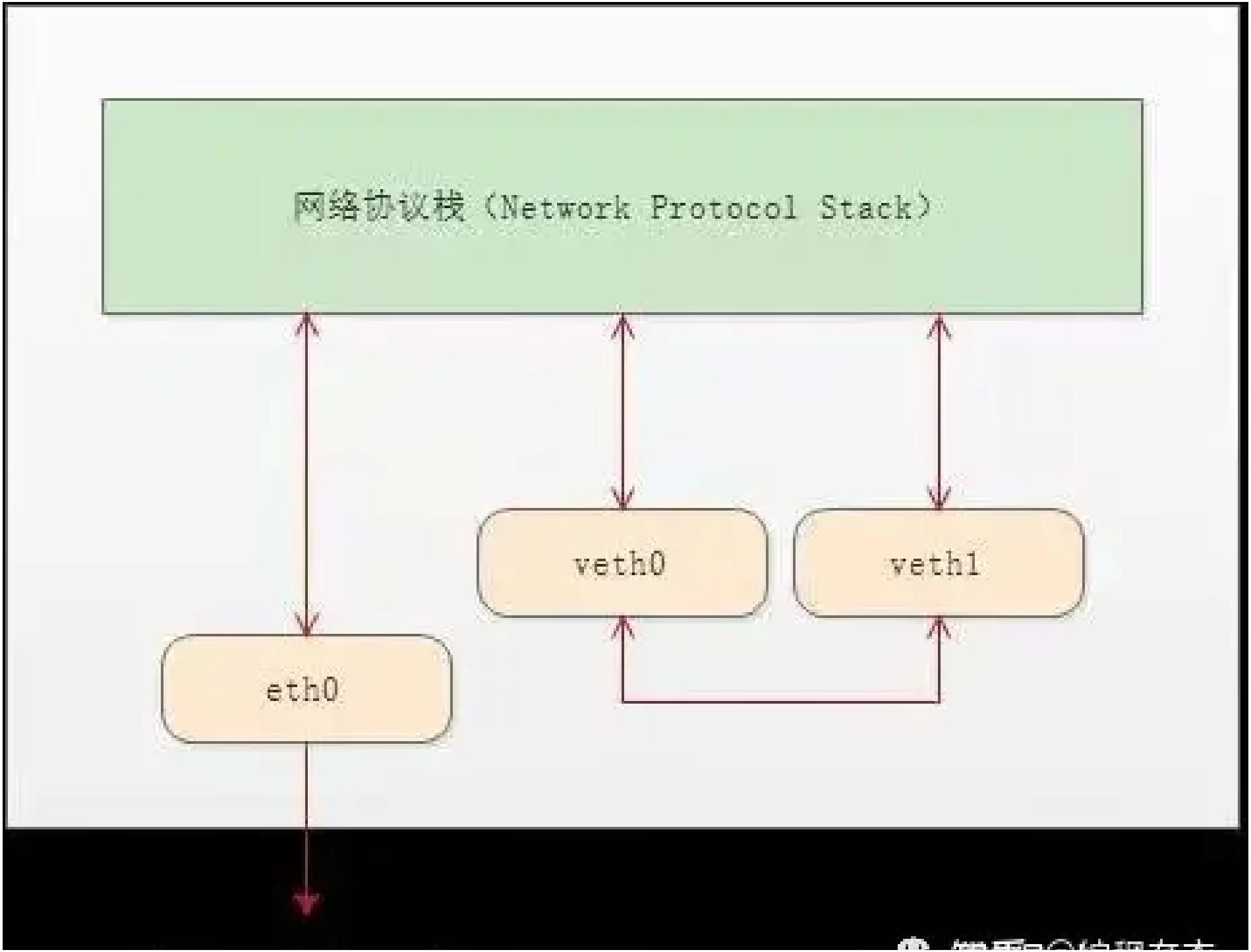
网络虚拟化　Docker　容器虚拟化

详解 - Linux 虚拟网络设备 veth-pair

其所以然
不时幻想的程序员

01 veth-pair 是什么#

顾名思义，veth-pair 就是一对的虚拟设备接口，它都是成对出现的。一端连着协议栈，一端彼此相连着。如下图所示：


网络协议栈（Network Protocol Stack）
veth0
veth1
eth0

正因为有这个特性，它常常充当着一个桥梁，连接着各种虚拟网络设备，典型的例子像“两个 namespace 之间的连接”，“Bridge、OVS 之间的连接”，“Docker 容器之间的连接” 等等，以此构建出非常复杂的虚拟网络结构，比如 OpenStack Neutron。

02 veth-pair 的连通性#

我们给上图中的 veth0 和 veth1 分别配上 IP：10.1.1.2 和 10.1.1.3，然后从 veth0 ping 一下 veth1。创建veth pair，设置其ip并将其开启

```
sudo ip link add veth0 type veth peer name veth1
sudo ip addr add 10.1.1.2/24 dev veth0
sudo ip addr add 10.1.1.3/24 dev veth1
sudo ip link set veth0 up
sudo ip link set veth1 up

ping -I veth0 10.1.1.3 -c 2
```

理论上它们处于同网段，是能 ping 通的，但结果却是 ping 不通。

抓个包看看，`tcpdump -nnt -i veth0`

```
root@ubuntu:~# tcpdump -nnt -i veth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
```

```
listening on veth0, link-type EN10MB (Ethernet), capture size 262144 bytes
ARP, Request who-has 10.1.1.3 tell 10.1.1.2, length 28
ARP, Request who-has 10.1.1.3 tell 10.1.1.2, length 28
```

可以看到，由于 veth0 和 veth1 处于同一个网段，且是第一次连接，所以会事先发 ARP 包，但 veth1 并没有响应 ARP 包。

经查阅，这是由于我使用的 Ubuntu 系统内核中一些 ARP 相关的默认配置限制所导致的，需要修改一下配置项：

```
echo 1 > /proc/sys/net/ipv4/conf/veth1/accept_local
echo 1 > /proc/sys/net/ipv4/conf/veth0/accept_local
echo 0 > /proc/sys/net/ipv4/conf/all/rp_filter
echo 0 > /proc/sys/net/ipv4/conf/veth0/rp_filter
echo 0 > /proc/sys/net/ipv4/conf/veth1/rp_filter
```

完了再 ping 就行了。

```
root@ubuntu:~# ping -I veth0 10.1.1.3 -c 2
PING 10.1.1.3 (10.1.1.3) from 10.1.1.2 veth0: 56(84) bytes of data.
64 bytes from 10.1.1.3: icmp_seq=1 ttl=64 time=0.047 ms
64 bytes from 10.1.1.3: icmp_seq=2 ttl=64 time=0.064 ms

--- 10.1.1.3 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 3008ms
rtt min/avg/max/mdev = 0.047/0.072/0.113/0.025 ms
```

我们对这个通信过程比较感兴趣，可以抓包看看。

对于 veth0 口：

```
root@ubuntu:~# tcpdump -nnt -i veth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on veth0, link-type EN10MB (Ethernet), capture size 262144 bytes
ARP, Request who-has 10.1.1.3 tell 10.1.1.2, length 28
ARP, Reply 10.1.1.3 is-at 5a:07:76:8e:fb:cd, length 28
IP 10.1.1.2 > 10.1.1.3: ICMP echo request, id 2189, seq 1, length 64
IP 10.1.1.2 > 10.1.1.3: ICMP echo request, id 2189, seq 2, length 64
IP 10.1.1.2 > 10.1.1.3: ICMP echo request, id 2189, seq 3, length 64
IP 10.1.1.2 > 10.1.1.3: ICMP echo request, id 2244, seq 1, length 64
```

对于 veth1 口：

```
root@ubuntu:~# tcpdump -nnt -i veth1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on veth1, link-type EN10MB (Ethernet), capture size 262144 bytes
ARP, Request who-has 10.1.1.3 tell 10.1.1.2, length 28
ARP, Reply 10.1.1.3 is-at 5a:07:76:8e:fb:cd, length 28
IP 10.1.1.2 > 10.1.1.3: ICMP echo request, id 2189, seq 1, length 64
IP 10.1.1.2 > 10.1.1.3: ICMP echo request, id 2189, seq 2, length 64
IP 10.1.1.2 > 10.1.1.3: ICMP echo request, id 2189, seq 3, length 64
IP 10.1.1.2 > 10.1.1.3: ICMP echo request, id 2244, seq 1, length 64
```

奇怪，我们并没有看到 ICMP 的 `echo reply` 包，那它是怎么 ping 通的?

其实这里 `echo reply` 走的是 localback 口，不信抓个包看看：

```
root@ubuntu:~# tcpdump -nnt -i lo
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
```

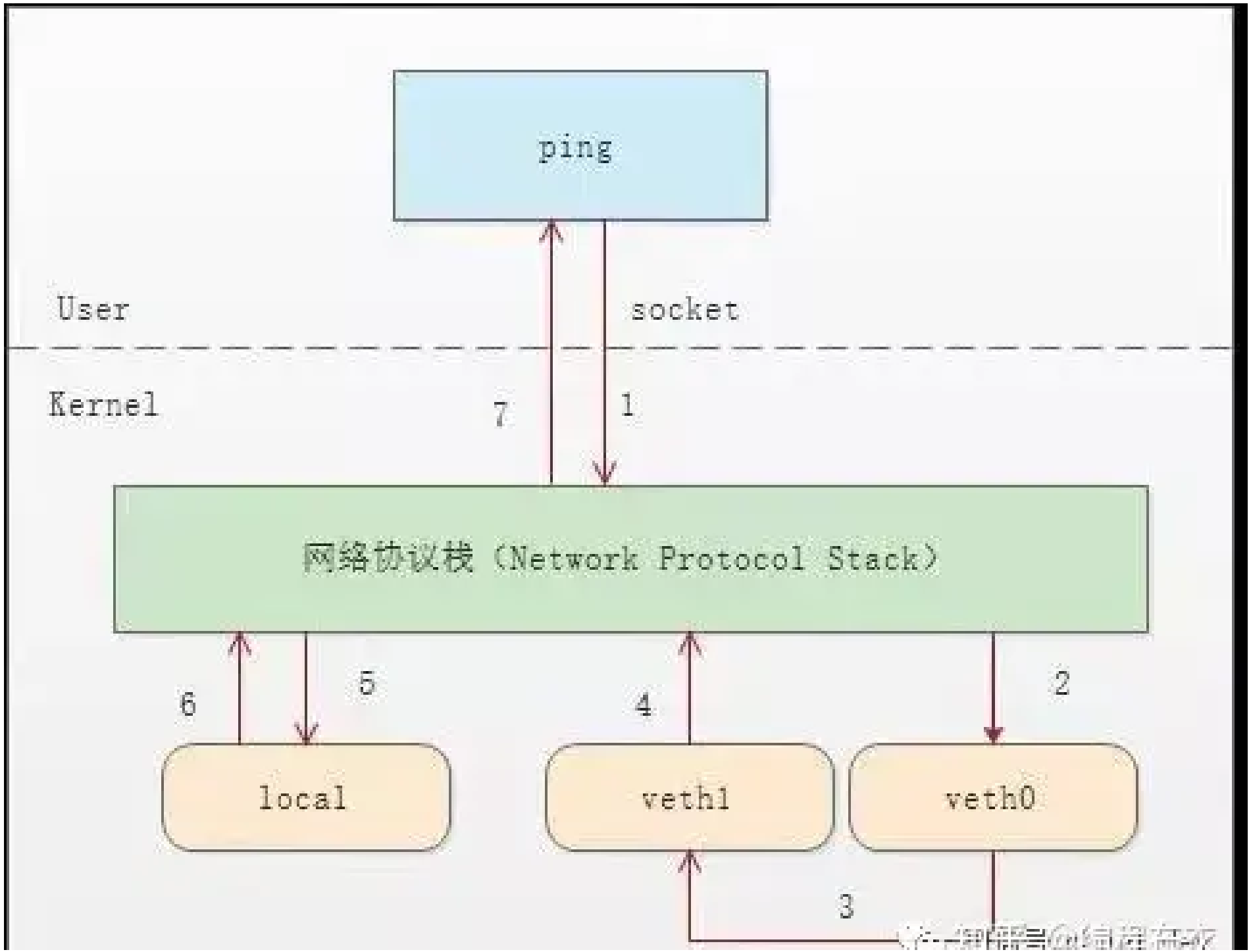
```
listening on lo, link-type EN10MB (Ethernet), capture size 262144 bytes
IP 10.1.1.3 > 10.1.1.2: ICMP echo reply, id 2244, seq 1, length 64
IP 10.1.1.3 > 10.1.1.2: ICMP echo reply, id 2244, seq 2, length 64
IP 10.1.1.3 > 10.1.1.2: ICMP echo reply, id 2244, seq 3, length 64
IP 10.1.1.3 > 10.1.1.2: ICMP echo reply, id 2244, seq 4, length 64
```

为什么?

我们看下整个通信流程就明白了。

1. 首先 ping 程序构造 ICMP `echo request`，通过 socket 发给协议栈。
2. 由于 ping 指定了走 veth0 口，如果是第一次，则需要发 ARP 请求，否则协议栈直接将数据包交给 veth0。
3. 由于 veth0 连着 veth1，所以 ICMP request 直接发给 veth1。
4. veth1 收到请求后，交给另一端的协议栈。
5. 协议栈看本地有 10.1.1.3 这个 IP，于是构造 ICMP reply 包，查看路由表，发现回给 10.1.1.0 网段的数据包应该走 localback 口，于是将 reply 包交给 lo 口（会优先查看路由表的 0 号表，`ip route show table 0` 查看）。
6. lo 收到协议栈的 reply 包后，啥都没干，转手又回给协议栈。
7. 协议栈收到 reply 包之后，发现有 socket 在等待包，于是将包给 socket。
8. 等待在用户态的 ping 程序发现 socket 返回，于是就收到 ICMP 的 reply 包。

整个过程如下图所示:


ping
User
socket
Kernel
7
1
网络协议栈（Network Protocol Stack）
6
5
4
2
local
veth1
veth0
3

03 两个 namespace 之间的连通性#

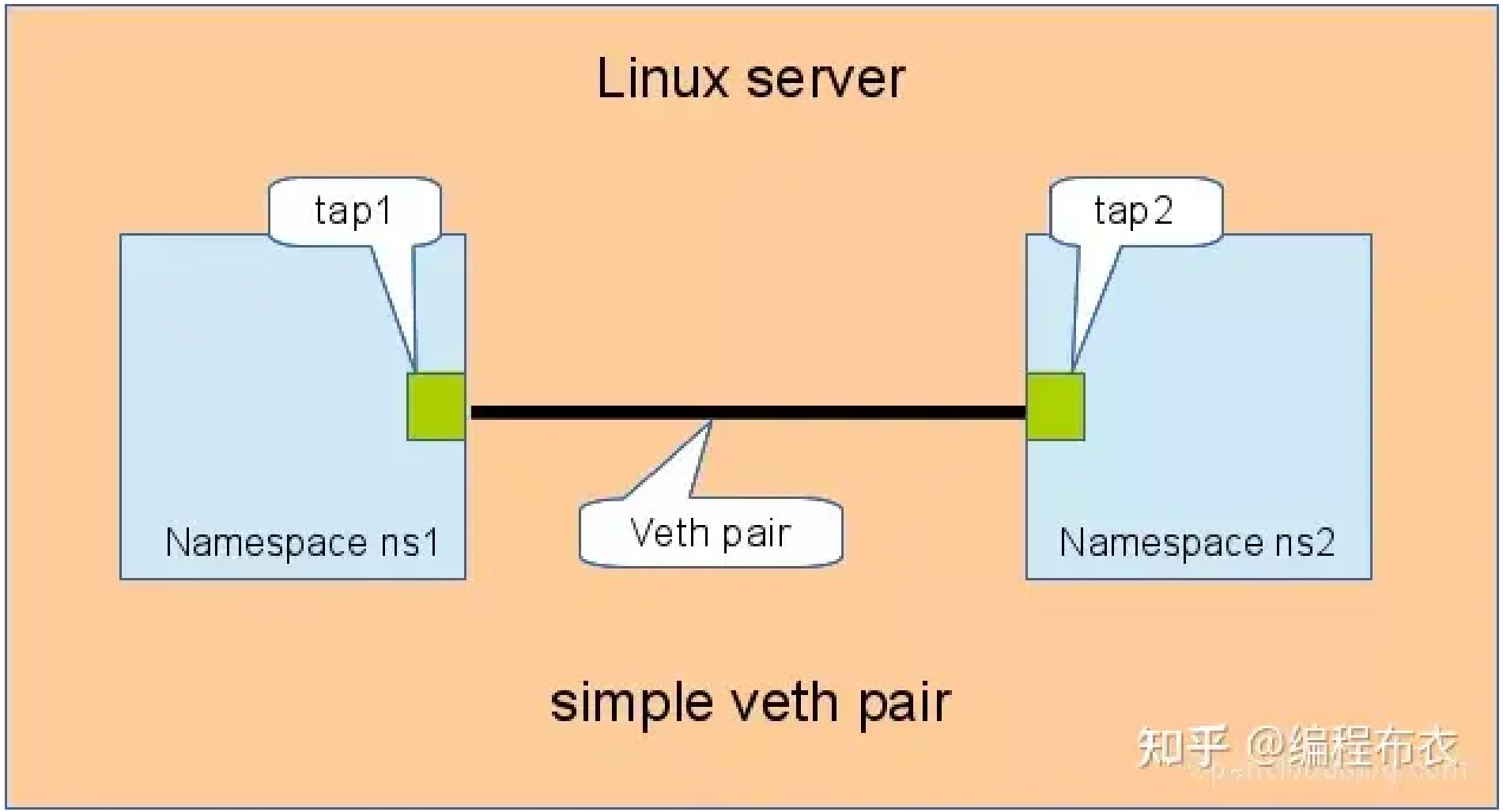
namespace 是 Linux 2.6.x 内核版本之后支持的特性，主要用于资源的隔离。有了 namespace，一个 Linux 系统就可以抽象出多个网络子系统，各子系统间都有自己的网络设备，协议栈等，彼此之间互不影响。

如果各个 namespace 之间需要通信，怎么办呢，答案就是用 veth-pair 来做桥梁。

根据连接的方式和规模，可以分为“直接相连”，“通过 Bridge 相连” 和 “通过 OVS 相连”。

3.1 直接相连#

直接相连是最简单的方式，如下图，一对 veth-pair 直接将两个 namespace 连接在一起。



给 veth-pair 配置 IP，测试连通性：

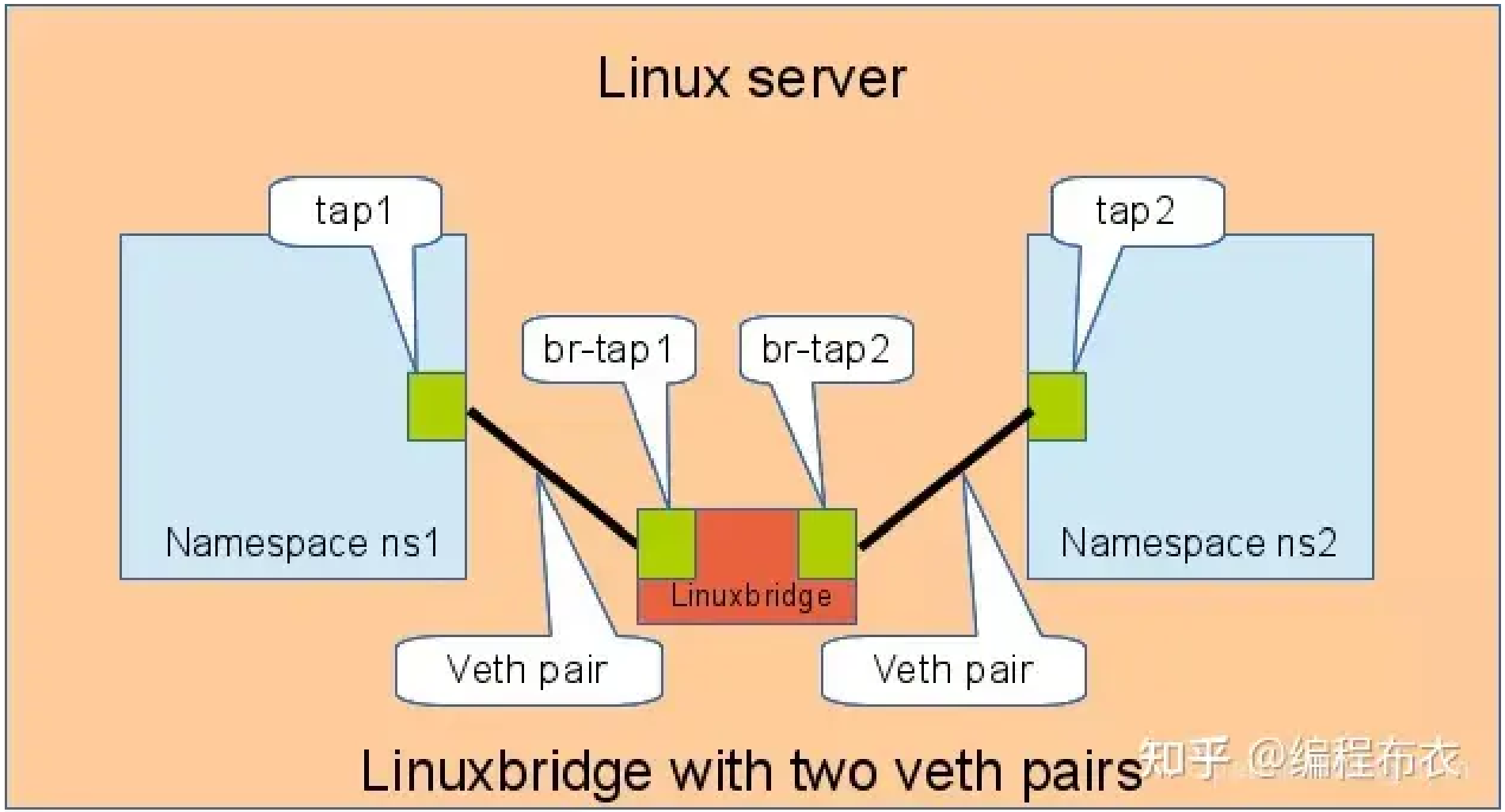
```
# 创建 namespace
ip netns a ns1
ip netns a ns2
```

```
# 创建一对 veth-pair veth0 veth1
ip l a veth0 type veth peer name veth1

# 将 veth0 veth1 分别加入两个 ns
ip l s veth0 netns ns1
ip l s veth1 netns ns2

# 给两个 veth0 veth1 配上 IP 并启用
ip netns exec ns1 ip a a 10.1.1.2/24 dev veth0
ip netns exec ns1 ip l s veth0 up
ip netns exec ns2 ip a a 10.1.1.3/24 dev veth1
ip netns exec ns2 ip l s veth1 up

# 从 veth0 ping veth1
[root@localhost ~]# ip netns exec ns1 ping 10.1.1.3
PING 10.1.1.3 (10.1.1.3) 56(84) bytes of data.
64 bytes from 10.1.1.3: icmp_seq=1 ttl=64 time=0.073 ms
64 bytes from 10.1.1.3: icmp_seq=2 ttl=64 time=0.068 ms

--- 10.1.1.3 ping statistics ---
15 packets transmitted, 15 received, 0% packet loss, time 14000ms
rtt min/avg/max/mdev = 0.068/0.084/0.201/0.032 ms
```

3.2 通过 Bridge 相连#

Linux Bridge 相当于一台交换机，可以中转两个 namespace 的流量，我们看看 veth-pair 在其中扮演什么角色。

如下图，两对 veth-pair 分别将两个 namespace 连到 Bridge 上。



同样给 veth-pair 配置 IP，测试其连通性：

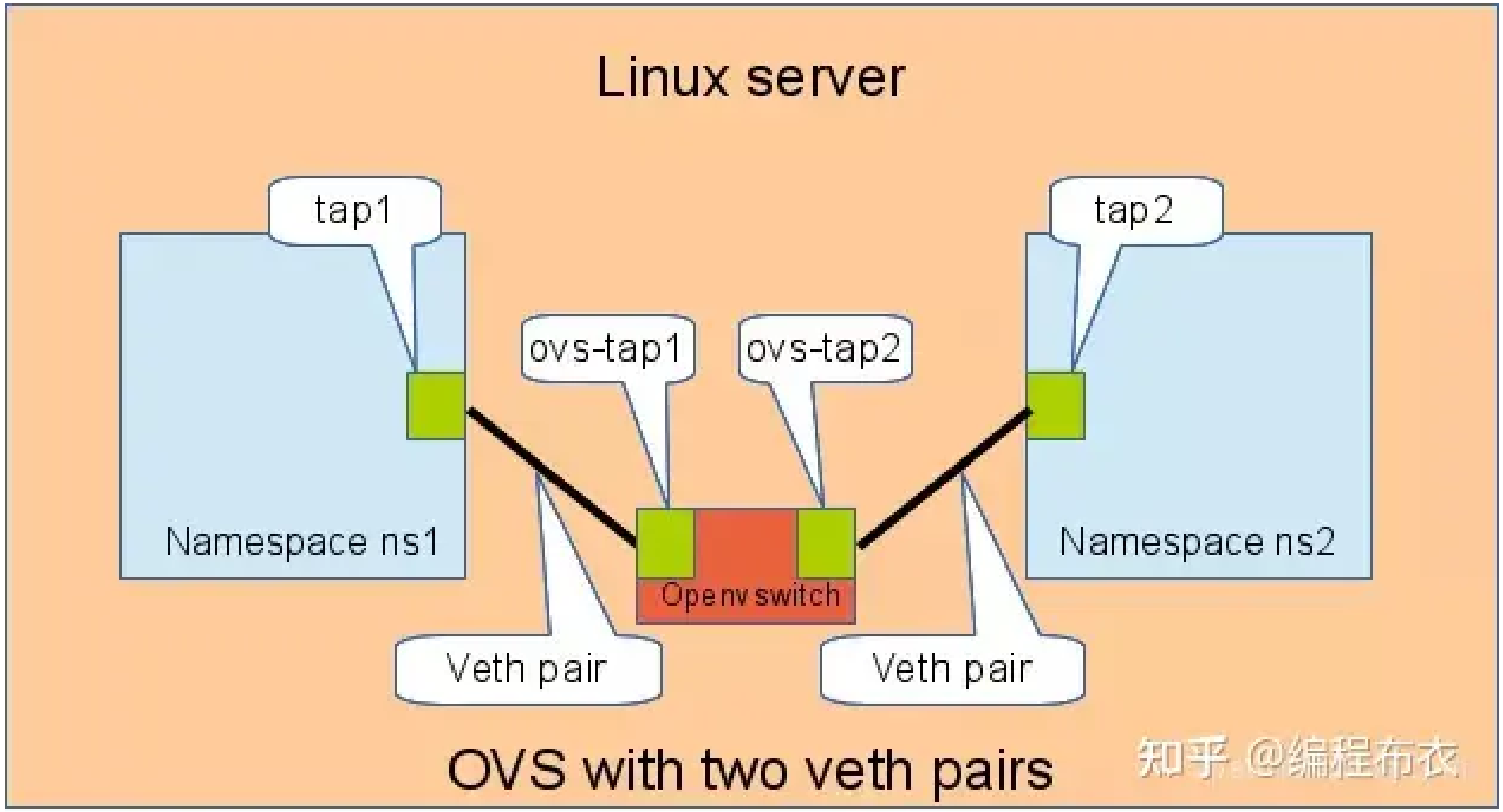
```
# 首先创建 bridge br0
ip l a br0 type bridge
ip l s br0 up
```

```
# 然后创建两对 veth-pair
ip l a veth0 type veth peer name br-veth0
ip l a veth1 type veth peer name br-veth1

# 分别将两对 veth-pair 加入两个 ns 和 br0
ip l s veth0 netns ns1
ip l s br-veth0 master br0
ip l s br-veth0 up

ip l s veth1 netns ns2
ip l s br-veth1 master br0
ip l s br-veth1 up

# 给两个 ns 中的 veth 配置 IP 并启用
ip netns exec ns1 ip a a 10.1.1.2/24 dev veth0
ip netns exec ns1 ip l s veth0 up

ip netns exec ns2 ip a a 10.1.1.3/24 dev veth1
ip netns exec ns2 ip l s veth1 up

# veth0 ping veth1
[root@localhost ~]# ip netns exec ns1 ping 10.1.1.3
PING 10.1.1.3 (10.1.1.3) 56(84) bytes of data.
64 bytes from 10.1.1.3: icmp_seq=1 ttl=64 time=0.060 ms
64 bytes from 10.1.1.3: icmp_seq=2 ttl=64 time=0.105 ms

--- 10.1.1.3 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.060/0.082/0.105/0.024 ms
```

3.3 通过 OVS 相连#

OVS 是第三方开源的 Bridge，功能比 Linux Bridge 要更强大，对于同样的实验，我们用 OVS 来看看是什么效果。

如下图所示：



同样测试两个 namespace 之间的连通性：

```
# 用 ovs 提供的命令创建一个 ovs bridge
ovs-vsctl add-br ovs-br

# 创建两对 veth-pair
ip l a veth0 type veth peer name ovs-veth0
ip l a veth1 type veth peer name ovs-veth1

# 将 veth-pair 两端分别加入到 ns 和 ovs bridge 中
ip l s veth0 netns ns1
ovs-vsctl add-port ovs-br ovs-veth0
ip l s ovs-veth0 up

ip l s veth1 netns ns2
ovs-vsctl add-port ovs-br ovs-veth1
ip l s ovs-veth1 up

# 给 ns 中的 veth 配置 IP 并启用
ip netns exec ns1 ip a a 10.1.1.2/24 dev veth0
ip netns exec ns1 ip l s veth0 up

ip netns exec ns2 ip a a 10.1.1.3/24 dev veth1
ip netns exec ns2 ip l s veth1 up

# veth0 ping veth1
[root@localhost ~]# ip netns exec ns1 ping 10.1.1.3
PING 10.1.1.3 (10.1.1.3) 56(84) bytes of data.
64 bytes from 10.1.1.3: icmp_seq=1 ttl=64 time=0.311 ms
64 bytes from 10.1.1.3: icmp_seq=2 ttl=64 time=0.087 ms
```

```
^C
--- 10.1.1.3 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.087/0.199/0.311/0.112 ms
```

总结#

veth-pair 在虚拟网络中充当着桥梁的角色，连接多种网络设备构成复杂的网络。

veth-pair 的三个经典实验，直接相连、通过 Bridge 相连和通过 OVS 相连。

参考#

opencloudblog.com/?...

segmentfault.com/a/1190...

发布于 2023-12-06 04:47 · IP 属地美国

veth peer　venn net

Linux虚拟网络设备之veth

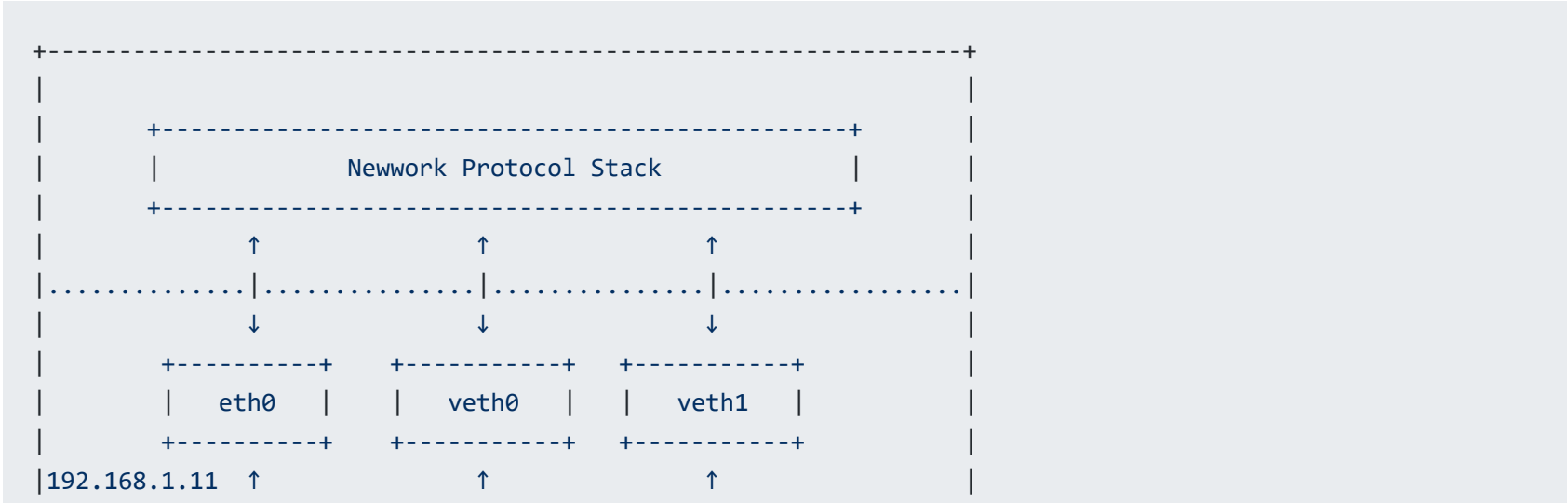
public0821 2017-05-01 阅读 8 分钟

有了上一篇关于tun/tap的介绍之后，大家应该对虚拟网络设备有了一定的了解，本篇将接着介绍另一种虚拟网络设备veth。

veth设备的特点

- veth和其它的网络设备都一样，一端连接的是内核协议栈。
- veth设备是成对出现的，另一端两个设备彼此相连
- 一个设备收到协议栈的数据发送请求后，会将数据发送到另一个设备上去。

下面这张关系图很清楚的说明了veth设备的特点：



```
|              |               +---------------+                |
|              |         192.168.2.11     192.168.2.1           |
+--------------|------------------------------------------------+
               ↓
          Physical Network
```

上图中，我们给物理网卡eth0配置的IP为192.168.1.11， 而veth0和veth1的IP分别是192.168.2.11和192.168.2.1。

示例

我们通过示例的方式来一步一步的看看veth设备的特点。

只给一个veth设备配置IP

先通过ip link命令添加veth0和veth1，然后配置veth0的IP，并将两个设备都启动起来

```
dev@debian:~$ sudo ip link add veth0 type veth peer name veth1
dev@debian:~$ sudo ip addr add 192.168.2.11/24 dev veth0
dev@debian:~$ sudo ip link set veth0 up
dev@debian:~$ sudo ip link set veth1 up
```

这里不给veth1设备配置IP的原因就是想看看在veth1没有IP的情况下，veth0收到协议栈的数据后会不会转发给veth1。

ping一下192.168.2.1，由于veth1还没配置IP，所以肯定不通

```
dev@debian:~$ ping -c 4 192.168.2.1
PING 192.168.2.1 (192.168.2.1) 56(84) bytes of data.
```

```
From 192.168.2.11 icmp_seq=1 Destination Host Unreachable
From 192.168.2.11 icmp_seq=2 Destination Host Unreachable
From 192.168.2.11 icmp_seq=3 Destination Host Unreachable
From 192.168.2.11 icmp_seq=4 Destination Host Unreachable

--- 192.168.2.1 ping statistics ---
4 packets transmitted, 0 received, +4 errors, 100% packet loss, time 3015ms
pipe 3
```

但为什么ping不通呢？是到哪一步失败的呢？

先看看抓包的情况，从下面的输出可以看出，veth0和veth1收到了同样的ARP请求包，但没有看到ARP应答包：

```
dev@debian:~$ sudo tcpdump -n -i veth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on veth0, link-type EN10MB (Ethernet), capture size 262144 bytes
20:20:18.285230 ARP, Request who-has 192.168.2.1 tell 192.168.2.11, length 28
20:20:19.282018 ARP, Request who-has 192.168.2.1 tell 192.168.2.11, length 28
20:20:20.282038 ARP, Request who-has 192.168.2.1 tell 192.168.2.11, length 28
20:20:21.300320 ARP, Request who-has 192.168.2.1 tell 192.168.2.11, length 28
20:20:22.298783 ARP, Request who-has 192.168.2.1 tell 192.168.2.11, length 28
20:20:23.298923 ARP, Request who-has 192.168.2.1 tell 192.168.2.11, length 28

dev@debian:~$ sudo tcpdump -n -i veth1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on veth1, link-type EN10MB (Ethernet), capture size 262144 bytes
20:20:48.570459 ARP, Request who-has 192.168.2.1 tell 192.168.2.11, length 28
20:20:49.570012 ARP, Request who-has 192.168.2.1 tell 192.168.2.11, length 28
20:20:50.570023 ARP, Request who-has 192.168.2.1 tell 192.168.2.11, length 28
20:20:51.570023 ARP, Request who-has 192.168.2.1 tell 192.168.2.11, length 28
```

```
20:20:52.569988 ARP, Request who-has 192.168.2.1 tell 192.168.2.11, length 28
20:20:53.570833 ARP, Request who-has 192.168.2.1 tell 192.168.2.11, length 28
```

为什么会这样呢？了解ping背后发生的事情后就明白了：

1. ping进程构造ICMP echo请求包，并通过socket发给协议栈，
2. 协议栈根据目的IP地址和系统路由表，知道去192.168.2.1的数据包应该要由192.168.2.11口出去
3. 由于是第一次访问192.168.2.1，且目的IP和本地IP在同一个网段，所以协议栈会先发送ARP出去，询问192.168.2.1的mac地址
4. 协议栈将ARP包交给veth0，让它发出去
5. 由于veth0的另一端连的是veth1，所以ARP请求包就转发给了veth1
6. veth1收到ARP包后，转交给另一端的协议栈
7. 协议栈一看自己的设备列表，发现本地没有192.168.2.1这个IP，于是就丢弃了该ARP请求包，这就是为什么只能看到ARP请求包，看不到应答包的原因

给两个veth设备都配置IP

给veth1也配置上IP

```
dev@debian:~$ sudo ip addr add 192.168.2.1/24 dev veth1
```

再ping 192.168.2.1成功（由于192.168.2.1是本地IP，所以默认会走lo设备，为了避免这种情况，这里使用ping命令带上了-I参数，指定数据包走指定设备）

```
dev@debian:~$ ping -c 4 192.168.2.1 -I veth0
PING 192.168.2.1 (192.168.2.1) from 192.168.2.11 veth0: 56(84) bytes of data.
64 bytes from 192.168.2.1: icmp_seq=1 ttl=64 time=0.032 ms
```

```
64 bytes from 192.168.2.1: icmp_seq=2 ttl=64 time=0.048 ms
64 bytes from 192.168.2.1: icmp_seq=3 ttl=64 time=0.055 ms
64 bytes from 192.168.2.1: icmp_seq=4 ttl=64 time=0.050 ms

--- 192.168.2.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3002ms
rtt min/avg/max/mdev = 0.032/0.046/0.055/0.009 ms
```

> 注意：对于非debian系统，这里有可能ping不通，主要是因为内核中的一些ARP相关配置导致veth1不返回ARP应答包，如ubuntu上就会出现这种情况，解决办法如下：
> root@ubuntu:~# echo 1 > /proc/sys/net/ipv4/conf/veth1/accept_local
> root@ubuntu:~# echo 1 > /proc/sys/net/ipv4/conf/veth0/accept_local
> root@ubuntu:~# echo 0 > /proc/sys/net/ipv4/conf/all/rp_filter
> root@ubuntu:~# echo 0 > /proc/sys/net/ipv4/conf/veth0/rp_filter
> root@ubuntu:~# echo 0 > /proc/sys/net/ipv4/conf/veth1/rp_filter

再来看看抓包情况，我们在veth0和veth1上都看到了ICMP echo的请求包，但为什么没有应答包呢？上面不是显示ping进程已经成功收到了应答包吗？

```
dev@debian:~$ sudo tcpdump -n -i veth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on veth0, link-type EN10MB (Ethernet), capture size 262144 bytes
20:23:43.113062 IP 192.168.2.11 > 192.168.2.1: ICMP echo request, id 24169, seq 1, length 64
20:23:44.112078 IP 192.168.2.11
> 192.168.2.1: ICMP echo request, id 24169, seq 2, length 64
20:23:45.111091 IP 192.168.2.11 > 192.168.2.1: ICMP echo request, id 24169, seq 3, length 64
20:23:46.110082 IP 192.168.2.11 > 192.168.2.1: ICMP echo request, id 24169, seq 4, length 64


dev@debian:~$ sudo tcpdump -n -i veth1
```

```
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on veth1, link-type EN10MB (Ethernet), capture size 262144 bytes
20:24:12.221372 IP 192.168.2.11 > 192.168.2.1: ICMP echo request, id 24174, seq 1, length 64
20:24:13.222089 IP 192.168.2.11 > 192.168.2.1: ICMP echo request, id 24174, seq 2, length 64
20:24:14.224836 IP 192.168.2.11 > 192.168.2.1: ICMP echo request, id 24174, seq 3, length 64
20:24:15.223826 IP 192.168.2.11 > 192.168.2.1: ICMP echo request, id 24174, seq 4, length 64
```

看看数据包的流程就明白了：

1. ping进程构造ICMP echo请求包，并通过socket发给协议栈，
2. 由于ping程序指定了走veth0，并且本地ARP缓存里面已经有了相关记录，所以不用再发送ARP出去，协议栈就直接将该数据包交给了veth0
3. 由于veth0的另一端连的是veth1，所以ICMP echo请求包就转发给了veth1
4. veth1收到ICMP echo请求包后，转交给另一端的协议栈
5. 协议栈一看自己的设备列表，发现本地有192.168.2.1这个IP，于是构造ICMP echo应答包，准备返回
6. 协议栈查看自己的路由表，发现回给192.168.2.11的数据包应该走lo口，于是将应答包交给lo设备
7. lo接到协议栈的应答包后，啥都没干，转手又把数据包还给了协议栈（相当于协议栈通过发送流程把数据包给lo，然后lo再将数据包交给协议栈的接收流程）
8. 协议栈收到应答包后，发现有socket需要该包，于是交给了相应的socket
9. 这个socket正好是ping进程创建的socket，于是ping进程收到了应答包

抓一下lo设备上的数据，发现应答包确实是从lo口回来的：

```
dev@debian:~$ sudo tcpdump -n -i lo
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on lo, link-type EN10MB (Ethernet), capture size 262144 bytes
20:25:49.590273 IP 192.168.2.1 > 192.168.2.11: ICMP echo reply, id 24177, seq 1, length 64
20:25:50.590018 IP 192.168.2.1 > 192.168.2.11: ICMP echo reply, id 24177, seq 2, length 64
```

```
20:25:51.590027 IP 192.168.2.1 > 192.168.2.11: ICMP echo reply, id 24177, seq 3, length 64
20:25:52.590030 IP 192.168.2.1 > 192.168.2.11: ICMP echo reply, id 24177, seq 4, length 64
```

试着ping下其它的IP

ping 192.168.2.0/24网段的其它IP失败，ping一个公网的IP也失败：

```
dev@debian:~$ ping -c 1 -I veth0 192.168.2.2
PING 192.168.2.2 (192.168.2.2) from 192.168.2.11 veth0: 56(84) bytes of data.
From 192.168.2.11 icmp_seq=1 Destination Host Unreachable

--- 192.168.2.2 ping statistics ---
1 packets transmitted, 0 received, +1 errors, 100% packet loss, time 0ms

dev@debian:~$ ping -c 1 -I veth0 baidu.com
PING baidu.com (111.13.101.208) from 192.168.2.11 veth0: 56(84) bytes of data.
From 192.168.2.11 icmp_seq=1 Destination Host Unreachable

--- baidu.com ping statistics ---
1 packets transmitted, 0 received, +1 errors, 100% packet loss, time 0ms
```

从抓包来看，和上面第一种veth1没有配置IP的情况是一样的，ARP请求没人处理

```
dev@debian:~$ sudo tcpdump -i veth1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on veth1, link-type EN10MB (Ethernet), capture size 262144 bytes
02:25:23.223947 ARP, Request who-has 192.168.2.2 tell 192.168.2.11, length 28
02:25:24.224352 ARP, Request who-has 192.168.2.2 tell 192.168.2.11, length 28
02:25:25.223471 ARP, Request who-has 192.168.2.2 tell 192.168.2.11, length 28
02:25:27.946539 ARP, Request who-has 123.125.114.144 tell 192.168.2.11, length 28
```

```
02:25:28.946633 ARP, Request who-has 123.125.114.144 tell 192.168.2.11, length 28
02:25:29.948055 ARP, Request who-has 123.125.114.144 tell 192.168.2.11, length 28
```

结束语

从上面的介绍中可以看出，从veth0设备出去的数据包，会转发到veth1上，如果目的地址是veth1的IP的话，就能被协议栈处理，否则连ARP那关都过不了，IP forward啥的都用不上，所以不借助其它虚拟设备的话，这样的数据包只能在本地协议栈里面打转转，没法走到eth0上去，即没法发送到外面的网络中去。

下一篇将介绍Linux下的网桥，到时候veth设备就有用武之地了。

参考

- Linux Switching – Interconnecting Namespaces

网络 linux