第 2 章 Chapter 3

Linux 虚拟网络基础

Neutron 在构建网络服务时，利用了很多 Linux 虚拟网络功能（Linux 内核中的虚拟网络设备以及其他网络功能）。为了对 Neutron 有一个全面的理解，掌握一些 Linux 虚拟网络知识是必要的。本章将从使用方法入手，对与 Neutron 密切相关的 Linux 虚拟网络功能进行简单介绍。

2.1 tap

Linux 在谈到 tap 时，经常会与 tun 并列谈论。两者都是操作系统内核中的虚拟网络设备。tap 位于二层，tun 位于三层。需要说明的是，这里所说的设备（Device）是 Linux 的概念，不是我们平时生活中所说的设备。比如，生活中，我们常常把一台物理路由器称为一台设备，如图 2-1 所示。

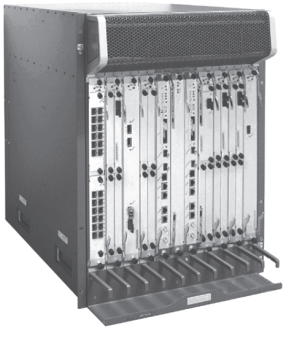


图 2-1 一台路由器设备

而 Linux 所说的设备，其背后指的是一个类似于数据结构、内核模块或设备驱动这样的含义。像 tun/tap 这样的设备，它的数据结构如下[㊀]：

```
struct tun_struct {
    char name[8];                       // 设备名
    unsigned long flags;                // 区分 tun 和 tap 设备
    struct fasync_struct *fasync;       // 文件异步通知结构
    wait_queue_head_t read_wait;        // 等待队列
```

㊀ 参考 http://blog.chinaunix.net/uid-7220314-id-208711.html。

```
    struct net_device dev;                  //Linux 抽象网络设备结构
    struct sk_buff_head txq;                //网络缓冲区队列
        struct net_device_stats stats;      //网卡状态信息结构
};
```

我们看到，甚至连数据结构，tap 与 tun 的定义都是同一个，两者仅仅是通过一个 Flag 来区分。不过从背后所承载的功能而言，两者还是有比较大的区别：tap 位于网络 OSI 模型的二层（数据链路层），tun 位于网络的三层。本节暂时只介绍 tap，tun 会在后面的章节专门介绍。

tap 从功能定位上来讲，位于数据链路层，数据链路层的主要协议有：

1）点对点协议（Point-to-Point Protocol）；

2）以太网（Ethernet）；

3）高级数据链路协议（High-Level Data Link Protocol）；

4）帧中继（Frame Relay）；

5）异步传输模式（Asynchronous Transfer Mode）。

但是 tap 只是与其中一种协议以太网（Ethernet）协议对应。所以，tap 有时也称为“虚拟以太设备”。

要想使用 Linux 命令行操作一个 tap，首先 Linux 得有 tun 模块（Linux 使用 tun 模块实现了 tun/tap），检查方法如下：

```
# 如果敲击 Linux 命令行 modinfo tun，有如下输出，则说明具有 tun 模块
modinfo tun
filename:       /lib/modules/4.5.5-300.fc24.x86_64/kernel/drivers/net/tun.ko.xz
alias:          devname:net/tun
alias:          char-major-10-200
......
```

当 Linux 版本具有 tun 模块时，还得看看其是否已经加载，检查方法如下：

```
lsmod | grep tun
tun    28672  2
```

如果已经加载，则会出现上述的“tun***”那一行。如果没有加载，则使用如下命令进行加载：

```
modprobe tun
```

当我们确认 Linux 加载了 tun 模块以后，我们还需要确认 Linux 是否有操作 tun/tap 的命令行工具 tunctl。在 Linux 中输入以下命令：

```
tunctl help
```

输入这个命令后，如果 Linux 有输出，则说明 OK，否则我们执行如下命令行以安装 tunctl：

```
yum install tunctl
```

具备了 tun 和 tunctl 以后，我们就可以创建一个 tap 设备了，命令行也很简单，如下：

```
tunctl -t tap_test
Set 'tap_test' persistent and owned by uid 0
```

我们可以通过如下命令来查看刚刚创建的 tap（名字是 tap_test）：

```
ip link list
...
13: tap_test: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT
    group default qlen 1000
link/ether 2e:72:30:19:4e:bb brd ff:ff:ff:ff:ff:ff
```

我们也可以通过如下命令来查看：

```
ifconfig -a
...
tap_test: flags=4098<BROADCAST,MULTICAST>  mtu 1500
ether 2e:72:30:19:4e:bb  txqueuelen 1000  (Ethernet)
...
```

通过上面的命令行输出，我们看到，这个 tap_test 还没有绑定 IP 地址。执行如下命令，给其绑定 IP 地址：

```
# 使用 ip addr 命令绑定 IP 地址命令
ip addr add local 192.168.100.1/24 dev tap_test
# 或者使用 ifconfig 命令绑定 IP 地址命令
ifconfig tap_test 192.168.100.1/24
```

使用 ifconfig -a 命令再查看一下：

```
ifconfig -a
......
tap_test: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 192.168.100.1  netmask 255.255.255.0  broadcast 0.0.0.0
        ......
```

配置完 IP 以后，一个 tap 设备就创建完毕了。我们会在后面的章节中通过测试用例，继续讲述 tap 的用法。

2.2　namespace

namespace 是 Linux 虚拟网络的一个重要概念。传统的 Linux 的许多资源是全局的，比如进程 ID 资源。而 namespace 的目的首先就是将这些资源做资源隔离。Linux 可以在一个 Host 内创建许多 namespace，于是那些原本是 Linux 全局的资源，就变成了 namespace 范围内的“全局”资源，而且不同 namespace 的资源互相不可见、彼此透明。

Linux 具体将哪些全局资源做了隔离呢？看 Linux 相应的代码最直接、最直观：

```
//nsproxy.h
struct nsproxy {
        atomic_t count;
        struct uts_namespace *uts_ns;
        struct ipc_namespace *ipc_ns;
        struct mnt_namespace *mnt_ns;
        struct pid_namespace *pid_ns;
        struct user_namespace *user_ns;
        struct net *net_ns;
};
```

以上 6 个资源，就是 Linux namespace 所隔离的资源，其基本含义如表 2-1 所示：

表 2-1 Linux namespace 隔离的资源

资源	含 义
uts_ns	UTS 为 Unix Timesharing System 的简称，包含内存名称、版本、底层体系结构等信息
ipc_ns	所有与进程间通信（IPC）有关的信息
mnt_ns	当前装载的文件系统
pid_ns	有关进程 ID 的信息
user_ns	资源配额的信息
net_ns	网络信息

从资源隔离的角度，Linux namespace 的示意图如图 2-2 所示：

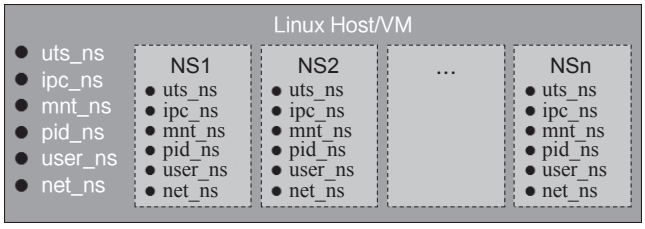


图 2-2 Linux namespace 示意图

图 2-2 表明，每个 namespace 里面将原本是全局资源的进行了隔离，彼此互相不可见。同时在 Linux 的 Host 或者 VM 中，当然也会有一套相关的资源。

单纯从网络的视角来看，一个 namespace 提供了一份独立的网络协议栈（网络设备接口、IPv4、IPv6、IP 路由、防火墙规则、sockets 等）。一个设备（Linux Device）只能位于一个 namespace 中，不同 namespace 中的设备可以利用 veth pair 进行桥接（veth pair 会在 2.3 节进行介绍）。

Linux 操作 namespace 的命令是 ip netns。这个命令行的帮助如下：

```
ip netns help
Usage: ip netns list
```

```
ip netns add NAME
ip netns set NAME NETNSID
ip [-all] netns delete [NAME]
ip netns identify [PID]
ip netns pids NAME
ip [-all] netns exec [NAME] cmd ...
ip netns monitor
ip netns list-id
```

我们首先创建一个 namespace：

```
# 首先查看一下当前的 namespace 列表
ip netns list
# 因为当前没有 namespace，所以上面的命令行没有任何返回
# 创建一个 namespace，名字是 ns_test
ip netns add ns_test
# 再查看一下当前的 namespace 列表，发现有一个 namespace：ns_test
ip netns list
ns_test # 这个是 ip netns list 的返回值
```

当我们创建一个 namespace 以后，我们可以把原来创建的虚拟设备 tap_test 迁移到这个 namespace 里去，命令行如下：

```
ip link set tap_test netns ns_test
```

这个时候，我们在原来的 host/vm 里面再执行 ip link list 命令，就会发现这个设备 tap_test 消失了（因为搬迁到 namespace ns_test 里去了）。

那么，我们如何查看或者操作 namespace 里面的设备呢？其命令行格式为：

```
ip [-all] netns exec [NAME] cmd ...       // cmd 为想要操作的命令行
```

比如我们要管理 ns_test 里面的设备，执行命令如下：

（1）在 ns_test 里执行 ip link list

```
ip netns exec ns_test ip link list
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
3: tap_test: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group
    default qlen 1000
    link/ether aa:bb:84:9f:a5:0c brd ff:ff:ff:ff:ff:ff
```

（2）在 ns_test 里执行 ifconfig -a

```
ip netns exec ns_test ifconfig -a
lo: flags=8<LOOPBACK>  mtu 65536
        ......
tap_test: flags=4098<BROADCAST,MULTICAST>  mtu 1500
        ether aa:bb:84:9f:a5:0c  txqueuelen 1000  (Ethernet)
        ......
```

（3）绑定 IP 地址

```
ip netns exec ns_test ifconfig tap_test 192.168.50.1/24 up
```

（4）查看 IP 地址

```
ip netns exec ns_test ifconfig -a
lo: flags=8<LOOPBACK>  mtu 65536
        loop  txqueuelen 1  (Local Loopback)
        ......

tap_test: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 192.168.50.1  netmask 255.255.255.0  broadcast 192.168.50.255
        ether aa:bb:84:9f:a5:0c  txqueuelen 1000  (Ethernet)
        ......
```

namespace 先介绍到这里，在后面的相关测试用例中，我们还会继续介绍。

2.3 veth pair

veth pair 不是一个设备，而是一对设备，以连接两个虚拟以太端口。操作 veth pair，需要跟 namespace 一起配合，不然就没有意义。我们设计一个测试用例，如图 2-3 所示。

两个 namespace ns1/ns2 中各有一个 tap 组成 veth pair，两者的 IP 地址如图 2-4 所示，两个 IP 进行互 ping 测试。下面我们就一步一步实现这个用例。

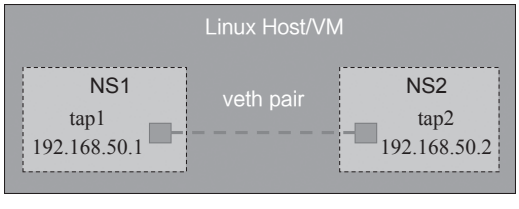


图 2-3 一个简单的 veth pair 测试用例

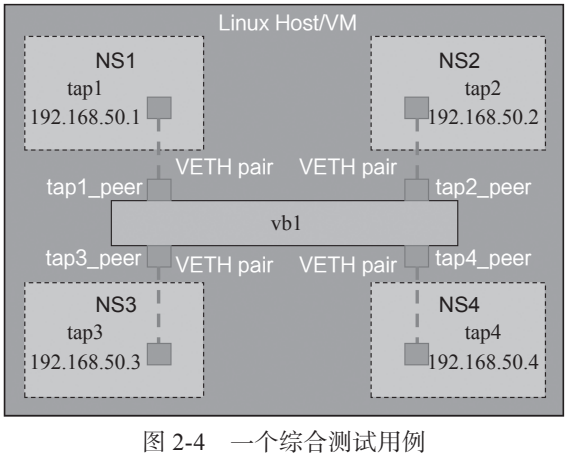


图 2-4 一个综合测试用例

```
# 创建 veth pair
ip link add tap1 type veth peer name tap2
```

```
# 创建 namespace: ns1、ns2
ip netns add ns1
ip netns add ns2
# 把两个 tap 分别迁移到对应的 namespace 中
ip link set tap1 netns ns1
ip link set tap2 netns ns2
# 分别给两个 tap 绑定 IP 地址
ip netns exec ns1 ip addr add local 192.168.50.1/24 dev tap1
ip netns exec ns2 ip addr add local 192.168.50.2/24 dev tap2
# 将两个 tap 设置为 up
ip netns exec ns1 ifconfig tap1 up
ip netns exec ns2 ifconfig tap2 up
# ping
ip netns exec ns2 ping 192.168.50.1
PING 192.168.50.1 (192.168.50.1) 56(84) bytes of data.
64 bytes from 192.168.50.1: icmp_seq=1 ttl=64 time=0.066 ms
......

ip netns exec ns1 ping 192.168.50.2
PING 192.168.50.2 (192.168.50.2) 56(84) bytes of data.
64 bytes from 192.168.50.2: icmp_seq=1 ttl=64 time=0.021 ms
......
```

通过上面的测试用例，我们了解了通过 veth pair 连接两个 namespace 的方法。但是，如果是 3 个 namespace 之间需要互通呢？或者多个 namespace 之间需要互通呢？ veth pair 只有一对 tap，无法胜任，怎么办？这就需要用到 Bridge/Switch。

2.4 Bridge

在 Linux 的语境里，Bridge（网桥）与 Switch（交换机）是一个概念，所以本文也不对两者进行区分。

Linux 实现 Bridge 功能的是 brctl 模块。在命令行里敲一下 brctl，如果能显示相关内容，则表示有此模块，否则还需要安装。安装命令是：

```
yum install bridge-utils
```

执行命令 brctl，显示的是 brctl 的帮助，如下：

```
brctl
Usage: brctl [commands]
commands:
    addbr         <bridge>              add bridge
    delbr         <bridge>              delete bridge
    addif         <bridge> <device>     add interface to bridge
    delif         <bridge> <device>     delete interface from bridge
    hairpin       <bridge> <port> {on|off}     turn hairpin on/off
```

```
setageing     <bridge> <time>              set ageing time
setbridgeprio <bridge> <prio>              set bridge priority
setfd         <bridge> <time>              set bridge forward delay
sethello      <bridge> <time>              set hello time
setmaxage     <bridge> <time>              set max message age
setpathcost   <bridge> <port> <cost>       set path cost
setportprio   <bridge> <port> <prio>       set port priority
show          [ <bridge> ]          show a list of bridges
showmacs      <bridge>              show a list of mac addrs
showstp       <bridge>              show bridge stp info
stp           <bridge> {on|off}     turn stp on/off
```

Bridge 本身的概念，本文不再啰唆。下面笔者通过一个综合测试用例来讲述 Bridge 的基本用法，同时也涵盖前面所述的几个概念：tap、namesapce、veth pair，如图 2-4 所示。

图 2-4 中，有 4 个 namespace，每个 namespace 都有一个 tap 与交换机上一个 tap 口组成 veth pair。这样 4 个 namespace 就通过 veth pair 及 Bridge 互联起来。

实现这个用例的命令行如下：

```
# 1）创建 veth pair
ip link add tap1 type veth peer name tap1_peer
ip link add tap2 type veth peer name tap2_peer
ip link add tap3 type veth peer name tap3_peer
ip link add tap4 type veth peer name tap4_peer
# 2）创建 namespace
ip netns add ns1
ip netns add ns2
ip netns add ns3
ip netns add ns4
# 3）把 tap 迁移到相应 namespace 中
ip link set tap1 netns ns1
ip link set tap2 netns ns2
ip link set tap3 netns ns3
ip link set tap4 netns ns4
# 4）创建 Bridge
brctl addbr br1
# 5）把相应 tap 添加到 Bridge 中
brctl addif br1 tap1_peer
brctl addif br1 tap2_peer
brctl addif br1 tap3_peer
brctl addif br1 tap4_peer
# 6）配置相应 tap 的 IP 地址
ip netns exec ns1 ip addr add local 192.168.50.1/24 dev tap1
ip netns exec ns2 ip addr add local 192.168.50.2/24 dev tap2
ip netns exec ns3 ip addr add local 192.168.50.3/24 dev tap3
ip netns exec ns4 ip addr add local 192.168.50.4/24 dev tap4
# 7）将 Bridge 及所有 tap 状态设置为 up
ip link set br1 up
ip link set tap1_peer up
ip link set tap2_peer up
```

```
ip link set tap3_peer up
ip link set tap4_peer up
ip netns exec ns1 ip link set tap1 up
ip netns exec ns2 ip link set tap2 up
ip netns exec ns3 ip link set tap3 up
ip netns exec ns4 ip link set tap4 up
# 8）现在就可以互相 ping 通了
ip netns exec ns1 ping 192.168.50.2
......
ip netns exec ns4 ping 192.168.50.1
```

2.5 Router

Linux 创建 Router 并没有像创建虚拟 Bridge 那样，有一个直接的命令 brctl，而且它间接的命令也没有，不能创建虚拟路由器……因为它就是路由器（Router）！

不过 Linux 默认没有打开路由转发功能。可以用这个命令验证一下：

```
less /proc/sys/net/ipv4/ip_forward
```

这个命令就是查看一下这个文件（/proc/sys/net/ipv4/ip_forward）的内容。该内容是一个数字。如果是“0”，则表示没有打开路由功能。把“0”修改为“1”，就是打开了 Linux 的路由转发功能：

```
echo "1" > /proc/sys/net/ipv4/ip_forward
```

这种打开方法，在机器重启以后就会失效了。一劳永逸的方法是修改配置文件“ /etc/sysctl.conf”，将 net.ipv4.ip_forward = 0 修改为 1，保存后退出即可。

下面我们仍然通过一个测试用例来直观感受一下 Router 的功能。测试用例组网图如图 2-5 所示：

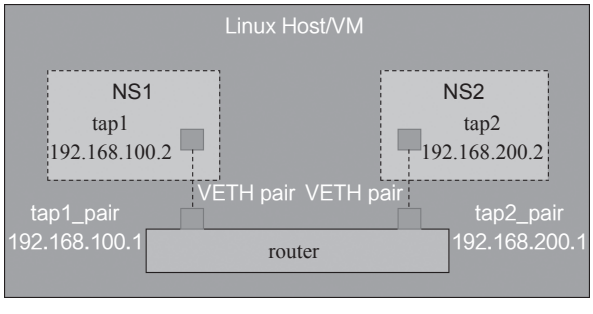
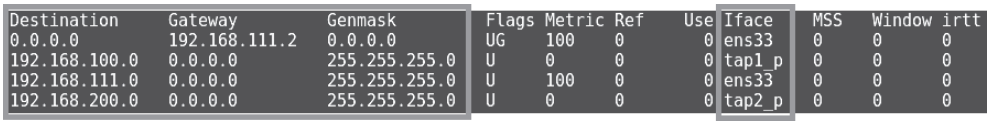


图 2-5 跨网段的 namespace/tap 互通测试组网图

在这个图 2-5 中，NS1/tap1 与 NS2/tap2 不在同一个网段中，中间需要经一个路由器进行转发才能互通。图中的 Router 是一个示意，其实就是 Linux 开通了路由转发功能。

当我们添加了 tap 并给其绑定 IP 地址时，Linux 会自动生成直连路由，如图 2-6 所示：



Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface	MSS	Window	irtt
0.0.0.0	192.168.111.2	0.0.0.0	UG	100	0	0	ens33	0	0	0
192.168.100.0	0.0.0.0	255.255.255.0	U	0	0	0	tap1_p	0	0	0
192.168.111.0	0.0.0.0	255.255.255.0	U	100	0	0	ens33	0	0	0
192.168.200.0	0.0.0.0	255.255.255.0	U	0	0	0	tap2_p	0	0	0

图 2-6　Linux 自动生成的路由表

下面我们根据测试用例组网图，创建设备。命令如下：

```
# 创建 veth pair
ip link add tap1 type veth peer name tap1_peer
ip link add tap2 type veth peer name tap2_peer
# 创建 namespace
ip netns add ns1
ip netns add ns2
# 将 tap 迁移到 namespace
ip link set tap1 netns ns1
ip link set tap2 netns ns2
# 配置 tap IP 地址
ip addr add local 192.168.100.1/24 dev tap1_peer
ip addr add local 192.168.200.1/24 dev tap2_peer
ip netns exec ns1 ip addr add local 192.168.100.2/24 dev tap1
ip netns exec ns2 ip addr add local 192.168.200.2/24 dev tap2
# 将 tap 设置为 up
ip link set tap1_peer up
ip link set tap2_peer up
ip netns exec ns1 ip link set tap1 up
ip netns exec ns2 ip link set tap2 up
```

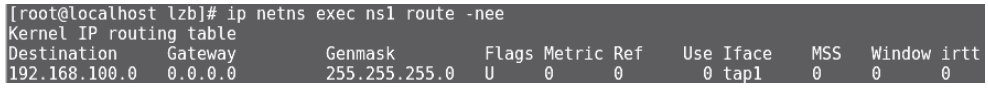
现在我们来做个测试，ping 一下：

```
ip netns exec ns1 ping 192.168.200.2
connect: Network is unreachable
```

ping 不通，网络不可达。我们查看一下 ns1 的路由表：

```
ip netns exec ns1 route -nee
```

图 2-7 是 ns1 的路由表截图，从图中可以看到，ns1 并没有到达 192.168.200.0/24 的路由表项，我们需要手工添加。命令行如下：



```
[root@localhost lzb]# ip netns exec ns1 route -nee
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface    MSS   Window irtt
192.168.100.0   0.0.0.0         255.255.255.0   U     0      0        0 tap1     0     0      0
```

图 2-7　ns1 的路由表

```
# ns1、ns2 都添加静态路由，分别到达对方的网段
ip netns exec ns1 route add -net 192.168.200.0 netmask 255.255.255.0 gw 192.
    168.100.1
ip netns exec ns2 route add -net 192.168.100.0 netmask 255.255.255.0 gw 192.
    168.200.1
```

这个时候，我们再来查看 ns1 的路由信息，ns1 已经具有到达 192.168.200.0/24 的路由表项，如图 2-8 所示：

```
ip netns exec ns1 route -nee
```

```
[root@localhost lzb]# ip netns exec ns1 route -nee
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface    MSS   Window irtt
192.168.100.0   0.0.0.0         255.255.255.0   U     0      0        0 tap1     0     0      0
192.168.111.0   192.168.100.1   255.255.255.0   UG    0      0        0 tap1     0     0      0
192.168.200.0   192.168.100.1   255.255.255.0   UG    0      0        0 tap1     0     0      0
```

图 2-8　增加静态路由后的 ns1 的路由表

再重新 ping 一下，通了：

```
ip netns exec ns1 ping 192.168.200.2
PING 192.168.200.2 (192.168.200.2) 56(84) bytes of data.
64 bytes from 192.168.200.2: icmp_seq=1 ttl=63 time=0.040 ms
......
ip netns exec ns2 ping 192.168.100.2
PING 192.168.100.2 (192.168.100.2) 56(84) bytes of data.
64 bytes from 192.168.100.2: icmp_seq=1 ttl=63 time=0.030 ms
......
```

2.6　tun

tun 是一个网络层（IP）的点对点设备，它启用了 IP 层隧道功能。Linux 原生支持的三层隧道，可以通过命令行 ip tunnel help 查看：

```
ip tunnel help
Usage: ip tunnel { add | change | del | show | prl | 6rd } [ NAME ]
       [ mode { ipip | gre | sit | isatap | vti } ] [ remote ADDR ] [ local ADDR ]
       [ [i|o]seq ] [ [i|o]key KEY ] [ [i|o]csum ]
       [ prl-default ADDR ] [ prl-nodefault ADDR ] [ prl-delete ADDR ]
       [ 6rd-prefix ADDR ] [ 6rd-relay_prefix ADDR ] [ 6rd-reset ]
       [ ttl TTL ] [ tos TOS ] [ [no]pmtudisc ] [ dev PHYS_DEV ]

Where: NAME := STRING
       ADDR := { IP_ADDRESS | any }
       TOS  := { STRING | 00..ff | inherit | inherit/STRING | inherit/00..ff }
       TTL  := { 1..255 | inherit }
       KEY  := { DOTTED_QUAD | NUMBER }
```

可以看到，Linux 一共原生支持 5 种三层隧道（tunnel），如表 2-2 所示：

表 2-2　Linux 原生支持的三层隧道

隧道	简　述
ipip	IP in IP，在 IPv4 报文的基础上再封装一个 IPv4 报文头，属于 IPv4 in IPv4

（续）

隧道	简　述
gre	通用路由封装（Generic Routing Encapsulation），定义了在任意一种网络层协议上封装任意一个其他网络层协议的协议，属于 IPv4/IPv6 over IPv4
sit	这个跟 ipip 类似，只不过是用一个 IPv4 的报文头封装 IPv6 的报文，属于 IPv6 over IPv4
isatap	站内自动隧道寻址协议，一般用于 IPv4 网络中的 IPv6/IPv4 节点间的通信
vti	全称是 Virtual Tunnel Interface，为 IPsec 隧道提供了一个可路由的接口类型

下面我们就用一个具体的测试用例来讲述 tun。测试用例组网如图 2-9 所示：

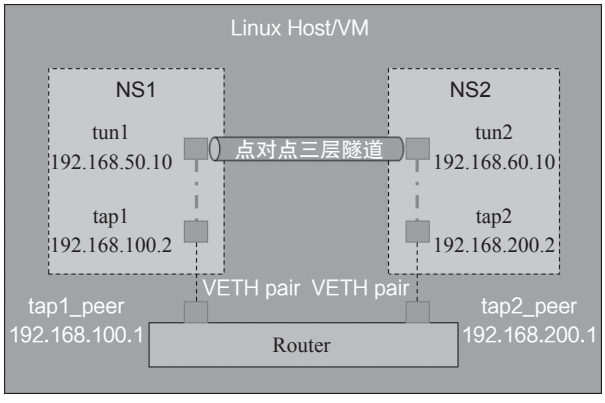


图 2-9　tun 测试组网图

图 2-9 中的 tun1、tun2，如果我们先忽略的话，剩下的就是我们在 2.5 节中讲述过的内容。

测试用例的第一步，就是使图中的 tap1 与 tap2 配置能通，这里我们不再重复。

当 tap1 和 tap2 配通以后，如果我们不把图 2-10 中的 tun1 和 tun2 暂时当做 tun 设备，而是当做两个“死”设备（比如当做是两个不做任何配置的网卡），那么这个时候 tun1 和 tun2 就像两个孤岛，不仅互相不通，而且跟 tap1、tap2 也没有关系。我们可以用一个更形象的图来示意，如图 2-10 所示：

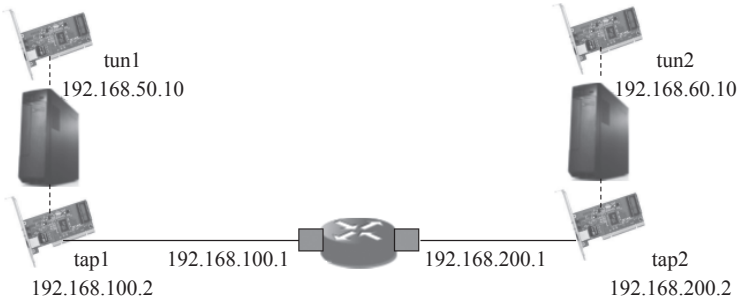


图 2-10　tun1、tun2 像两个孤岛

这个时候，我们就需要对 tun1、tun2 做相关配置，以使这两个两个孤岛能够互相通信。我们以 ipip tunnel 为例进行配置。

首先我们要加载 ipip 模块，Linux 默认是没有加载这个模块的。通过命令行 lsmod | grep ip 进行查看，查看结果如图 2-11 所示。

```
[root@localhost lzb]# lsmod | grep ip
ip6table_filter        16384  0
ip6_tables             28672  1 ip6table_filter
```

图 2-11　Linux 默认没有加载 ipip 模块

我们可以通过命令 modprobe ipip 来加载 ipip 模块。执行完此命令以后再查看，就能看到 ipip 模块被加载进入 Linux，如图 2-12 所示。

```
[root@localhost lzb]# modprobe ipip
[root@localhost lzb]# lsmod | grep ip
ipip                   16384  0
tunnel4                16384  1 ipip
ip_tunnel              24576  1 ipip
ip6table_filter        16384  0
ip6_tables             28672  1 ip6table_filter
```

图 2-12　Linux 加载 ipip 模块

加载了 ipip 模块以后，我们就可以创建 tun，并且给 tun 绑定一个 ipip 隧道（tunnel），命令行如下：

```
# 1）在 ns1 上创建 tun1 和 ipip tunnel
ip netns exec ns1 ip tunnel add tun1 mode ipip remote 192.168.200.2 local 192.
    168.100.2 ttl 255
ip netns exec ns1 ip link set tun1 up
ip netns exec ns1 ip addr add 192.168.50.10 peer 192.168.60.10 dev tun1
# 2）在 ns2 上创建 tun2 和 ipip tunnel
ip netns exec ns2 ip tunnel add tun2 mode ipip remote 192.168.100.2 local 192.
    168.200.2 ttl 255
ip netns exec ns2 ip link set tun2 up
ip netns exec ns2 ip addr add 192.168.60.10 peer 192.168.50.10 dev tun2
```

这个命令行需要做一个解释，如表 2-3 所示：

表 2-3　命令行 ip tunnel add 的解释

命令行	解　　释
ip netns exec ns1 ip tunnel add tun1 mode ipip remote 192.168.200.2 local 192.168.100.2 ttl 255	① ip netns exec ns1：在 ns1 上操作 ② ip tunnel add tun1 mode ipip：创建一个 tun 类型的设备 tun1，并且隧道模式是 ipip ③ remote 192.168.200.2 local 192.168.100.2：这个隧道的外层 IP 地址是：远端 192.168.200.2，近端（本地）192.168.100.2，就是两个 namespace 中的两个对应 tap ④ ttl 255：就是 ttl = 255
ip netns exec ns1 ip link set tun1 up	启动 tun1
ip netns exec ns1 ip addr add 192.168.50.10 peer 192.168.60.10 dev tun1	ip addr add 192.168.50.10 peer 192.168.60.10 dev tun1：设备 tun1 是一个点对点的设备，它自己的 IP 是 192.168.50.10，它的对端 IP 是 192.168.60.10。这两个 IP 地址就是 ipip 隧道的内层 IP

说明　把上面的命令行脚本中的 ipip 换成 gre，其余不变，就创建了一个 gre 隧道的 tun 设备对。

当做完上述配置，两个 tun 就可以互通了，如图 2-13 所示：

```
[root@localhost lzb]# ip netns exec ns1 ping 192.168.60.10
PING 192.168.60.10 (192.168.60.10) 56(84) bytes of data.
64 bytes from 192.168.60.10: icmp_seq=1 ttl=64 time=0.086 ms
64 bytes from 192.168.60.10: icmp_seq=2 ttl=64 time=0.054 ms
64 bytes from 192.168.60.10: icmp_seq=3 ttl=64 time=0.071 ms
^C
--- 192.168.60.10 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2000ms
rtt min/avg/max/mdev = 0.054/0.070/0.086/0.014 ms
[root@localhost lzb]#
[root@localhost lzb]# ip netns exec ns2 ping 192.168.50.10
PING 192.168.50.10 (192.168.50.10) 56(84) bytes of data.
64 bytes from 192.168.50.10: icmp_seq=1 ttl=64 time=0.039 ms
64 bytes from 192.168.50.10: icmp_seq=2 ttl=64 time=0.052 ms
^C
--- 192.168.50.10 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1000ms
```

图 2-13　两个 tun 互相 ping

因为我们说 tun 是一个设备，那么我们可以通过 ifconfig 这个命令，来看看这个设备的信息：

```
ip netns exec ns1 ifconfig -a
......
tun1: flags=209<UP,POINTOPOINT,RUNNING,NOARP>  mtu 1480
        inet 192.168.50.10  netmask 255.255.255.255  destination 192.168.60.10
        tunnel   txqueuelen 1  (IPIP Tunnel)
        ......
```

可以看到，tun1 是一个 ipip tunnel 的一个端点，IP 是 192.168.50.10，其对端 IP 是 192.168.60.10。

我们再看看路由表，如图 2-14 所示：

```
[root@localhost lzb]# ip netns exec ns1 route -nee
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface    MSS   Window irtt
192.168.60.10   0.0.0.0         255.255.255.255 UH    0      0        0 tun1     0     0      0
192.168.100.0   0.0.0.0         255.255.255.0   U     0      0        0 tap1     0     0      0
192.168.111.0   192.168.100.1   255.255.255.0   UG    0      0        0 tap1     0     0      0
192.168.200.0   192.168.100.1   255.255.255.0   UG    0      0        0 tap1     0     0      0
```

图 2-14　增加了 tun 设备的 ns1 的路由表

框中的内容告诉我们，到达目的地 192.168.60.10 的路由的一个直连路由直接从 tun1 出去即可。

2.7　iptables

iptables 与前文介绍的 tap/tun 等不同，它并不是一个网络设备。不过它们又有相同点：都是 Linux 的软件。通过 iptables 可以实现防火墙、NAT 等功能，不过这句话也对，也不对。

说它对，我们确实是通过 iptables 相关的命令行，实现了防火墙、NAT 的功能；说它不对，是因为 iptables 其实只是一个运行在用户空间的命令行工具，真正实现这些功能的是运行在内核空间的 netfilter 模块。它们之间的关系如图 2-15 所示[⊖]。

我们不必太在意这个图是什么意思，那样有点偏离主题，只需有个直观的感觉即可。本节所要描述的内容位于图中“iptables 命令”方框。

iptables 内置了三张表：filter、nat 和 mangle。filter 和 nat 顾名思义，是为了实现防火墙和 NAT 功能而服务的。mangle，翻译成汉语是“乱砍；损坏；用轧布机研光”等意思，它在这里指的是“主要应用在修改数据包内容上，用来做流量整形”。

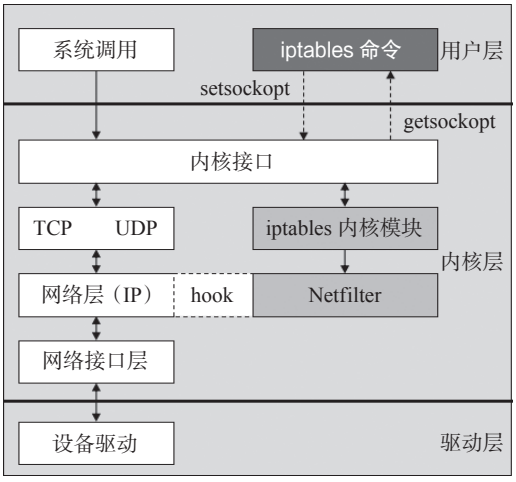


图 2-15　iptables 与 netfilter 的关系图

说明　iptables 还内置了另外 2 张表 raw 和 security，这里不详细介绍了。

iptables 内置的既是三张表，也是三条链（chain），或者换个角度说，iptables 内置的是三种策略（policy），而这些策略，是由不同规则（rule）串接而成。什么叫规则呢？我们以防火墙为例，讲述一条规则：

```
iptables -A INPUT -i eth0 -p icmp -j ACCEPT
```

这条规则表达的意思是：**允许所有从 eth0 端口进入且协议是 ICMP 的报文可以接受（可以进入下一个流程）的**。

这就是一条规则，至于 iptables 的命令行格式（语法）只是一个表象，它的本质是对进入的 IP 报文进行说明，如：符合什么样的条件（比如本条命令的条件是“允许所有从 eth0 端口进入且协议是 ICPM 的报文”）、做什么样的处理（比如本条命令的处理是“接受”，可以进入下一个流程）。

iptables 可以定义很多策略 / 规则，从图 2-16 中我们知道，这些规则最终会传递到内核 netfilter 模块，netfilter 模块会根据这些规则做相应的处理。netfilter 的处理方式是：从报文进入本机（linux host 或 vm）的那一刻起，到报文离开本机的那一刻止，中间这段时间（或者是发自本机的报文，从报文准备发送的那一刻，到报文离开本机的那一刻止，中间这段时间），netfilter 会在某些时刻点插入处理模块，这些处理模块根据相应的策略 / 规则对报文进行处理。

至于 nat、filter、mangle 三张表也可以这么理解：仅仅是为了达到不同的目的（功能）而实现的三个模块而已。

⊖　参考 http://blog.chinaunix.net/uid-23069658-id-3160506.html。

netfilter 插入的这些时刻点如图 2-16 所示：

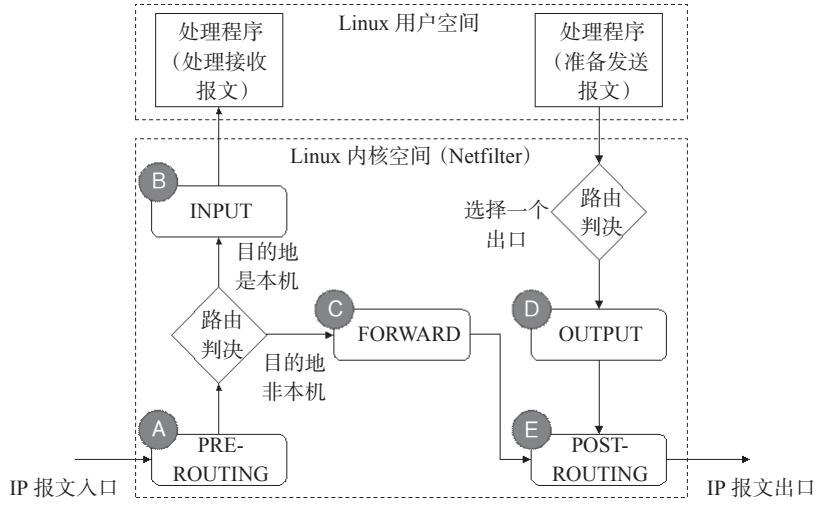


图 2-16　Netfilter 处理报文的时刻点

在这些时刻点中，上文提到三张表（模块）并不是所有的时刻都可以处理。在同一个时刻点，也可以有多个模块进行处理，那么这些模块就有一个处理顺序，谁先处理，谁后处理。这么说有点绕，具体请参见图 2-17：

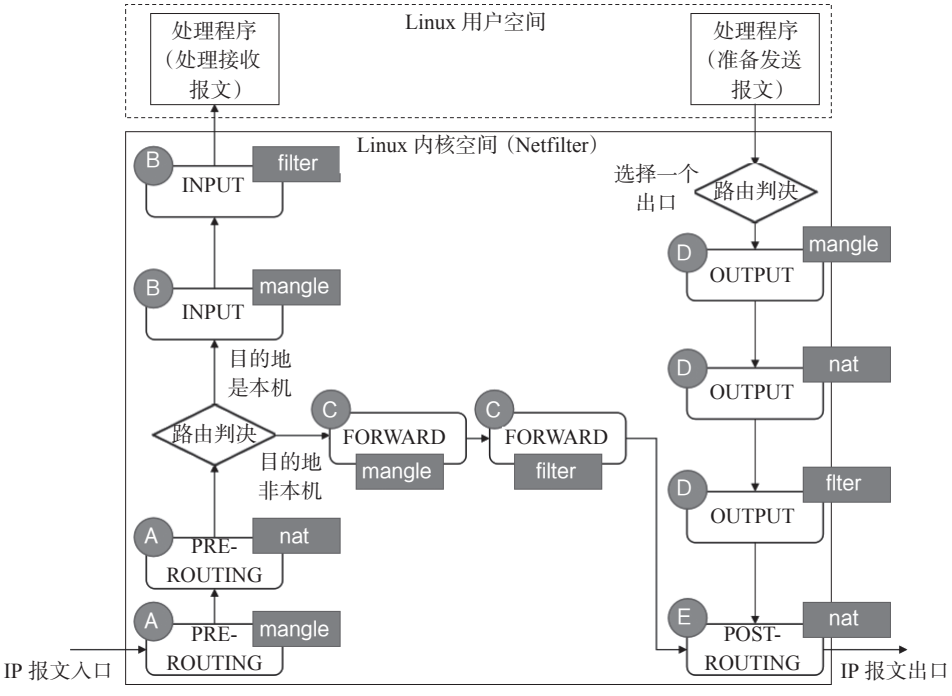


图 2-17　Netfilter 处理报文的详细时刻点

图中的几个关键时刻点，含义如下：

1）PREROUTING：报文进入网络接口尚未进入路由之前的时刻；

2）INPUT：路由判断是本机接收的报文，准备从内核空间进入到用户空间的时刻；

3）FORWARD：路由判断不是本机接收的报文，需要路由转发，路由转发的那个时刻；

4）OUTPUT：本机报文需要发出去，经过路由判断选择好端口以后，准备发送的那一刻；

5）POSTROUTING：FORWARD/OUTPUT 已经完成，报文即将出网络接口的那一刻。

三张表，所能对应的时刻点，如表 2-4 所示：

表 2-4　三张表所能处理的时刻点

表名	时刻点
mangle	PREROUTING, INPUT, FORWARD, OUTPUT
nat	PREROUTING, OUTPUT, POSTROUTING
filter	INPUT, FORWARD, OUTPUT

这三张表（三个模块）在这些时刻点，到底是做什么处理呢？下面我们逐个讲述。

2.7.1　NAT

1. NAT 的基本概念

在讲述 nat 这张表做何处理之前，我们首先介绍一下 NAT 的基本概念。

NAT（Network Address Translation，网络地址转换），顾名思义，就是从一个 IP 地址转换为另一个 IP 地址。当然，这里面的根本原因还是 IP 地址不够用的问题（解决 IP 地址枯竭的方法一个是 IPv6，另一个就是 NAT）。所以，NAT，大家基本做的还是公网地址与私网地址的互相转换。如果一定要在公网地址之间互相转换，或者私网地址之间互相转换，技术上是支持的，只是这样的场景非常非常少。

NAT，从实现技术角度来说，分为：静态 NAT、动态 NAT 和端口多路复用三种方案。

（1）静态 NAT（Static NAT）

静态 NAT（Static NAT），有两个特征（如图 2-18 所示）。

①私网 IP 地址与公网 IP 地址的转换规则是静态指定的，比如 10.10.10.1 与 50.0.0.1 互相转换，这个是静态指定好的。

②私网 IP 地址与公网 IP 地址是 1∶1，即一个私网 IP 地址对应 1 个公网 IP 地址。

图 2-18　静态 NAT

（2）动态 NAT

一般情况是公网 IP 比私网 IP 地址少的时候，用到动态 NAT 方案。如果公网 IP 地址比私网 IP 地址还多（或者相等），则用静态 NAT 就可以了，没必要这么麻烦。

动态 NAT，就是一批私网 IP 与公网 IP 地址之间不是固定的转换关系，而是在 IP 报文处理过程中由 NAT 模块进行动态匹配。虽然，公网 IP 比私网 IP 地址少，但是，同时在线的私网

IP 需求小于等于公网 IP 数量，不然某些私网 IP 将得不到正确的转换，从而导致网络通信失败。

动态 NAT，有三个特征（如图 2-19 所示）：

①私网与公网 IP 地址之间不是固定匹配转换的，而是变化的；

②两者之间的转换规则不是静态指定的，而是动态匹配的；

图 2-19 动态 NAT

③私网 IP 地址与公网 IP 地址之间是 m：n，一般 m < n

（3）端口多路复用 /PAT

如果私网 IP 地址有多个，而公网 IP 地址只有一个，那么，静态 NAT 显然是不行了，动态 NAT 也基本不行（只有一个公网 IP，不够用）。此时，就需要用到端口多路复用。多个私网 IP 映射到同一个公网 IP，不同的私网 IP 利用端口号进行区分，这里的端口号指的是 TCP/UDP 端口号。所以端口复用又叫 PAT（Port Address Translation）。

端口多路复用（PAT）的特征是（如图 2-20 所示）：

①私网 IP：公网 IP = m：1；

②以公网 IP + 端口号来区分私网 IP。

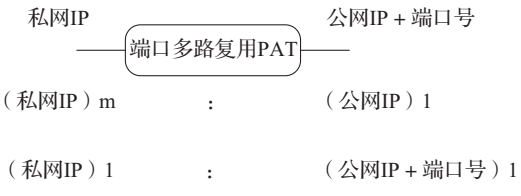


图 2-20 端口多路复用

（4）SNAT/DNAT

前面说的是静态 NAT（Static NAT）、动态 NAT。很遗憾，不能简称 SNAT、DNAT，因为 SNAT/DNAT 有另外的含义，是另外的缩写。要区分 SNAT（Source Network Address Translation，源地址转换）与 DNAT（Destination Network Address Translation，目的地址转换）这两个功能可以简单地由连接发起者是谁来区分。

①内部地址要访问公网上的服务时（如 Web 访问），内部地址会主动发起连接，由路由器或者防火墙上的网关对内部地址做个地址转换，将内部地址的私有 IP 转换为公网的公有 IP，网关的这个地址转换称为 SNAT，主要用于内部共享 IP 访问外部。

②当内部需要提供对外服务时（如对外发布 Web 网站），外部地址发起主动连接，由路由器或者防火墙上的网关接收这个连接，然后将连接转换到内部，此过程是由带有公网 IP 的网关替代内部服务来接收外部的连接，然后在内部做地址转换，此转换称为 DNAT，主要用于内部服务对外发布。

2. Netfilter 中的 NAT Chain

说 chain 有点太学究，大白话就是时刻点。通过前文介绍我们知道，NAT 一共在三个时刻点对 IP 报文做了处理。下面我们一个一个描述。

（1）NAT-PREROUTING（DNAT）

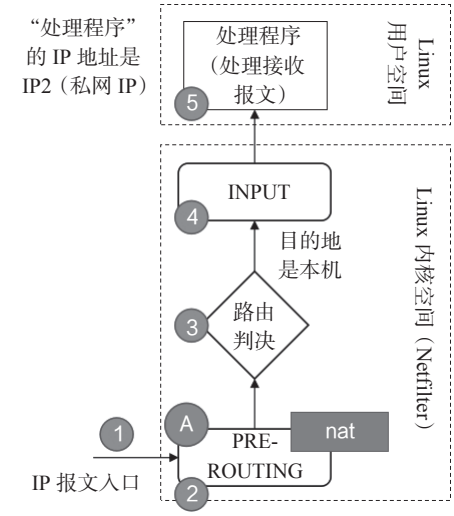
NAT-PREROUTING（DNAT）的处理时刻点，如图 2-21 所示。

IP 报文流的顺序是图中的“1-2-3-4-5”，在图 2-21 中 A 处，即 PREROUTING 时刻点进行 NAT 处理。IP 报文的目的地址是 IP1(公网 IP)，这个 IP1 就是 Linux 内核空间对外（公网）

呈现的 IP 地址（说明这样的 IP 地址可以有多个）。当报文到达 PREROUTING 这个时刻点时，NAT 模块会做处理。如果需要（即提前做了相关 NAT 配置），NAT 模块会将目的 IP 从 IP1 转换成 IP2（这个是提前配置好的），这也就是所谓的 DNAT。

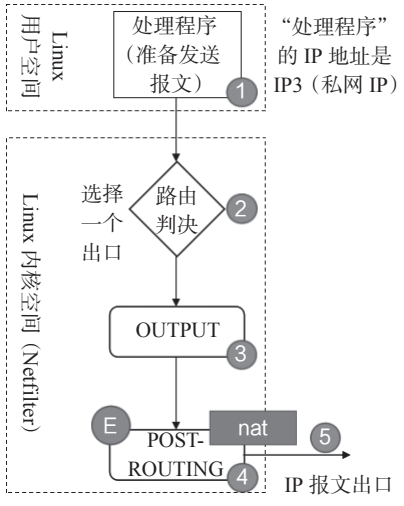
（2）NAT-POSTROUTING（SNAT）

NAT-POSTROUTING（SNAT）的处理时刻点，如图 2-22 所示：



IP 报文的目的地址是 IP1（公网 IP）也是 Linux 内核空间对外呈现的 IP

图 2-21　NAT-PREROUTING（DNAT）



NAT 模块会将 IP 报文的源地址修改 IP1（公网 IP）也是 Linux 内核空间对外呈现的 IP

图 2-22　NAT-POSTROUTING（SNAT）

IP 报文流的顺序是图中的“1-2-3-4-5”，在图中“E”处，即 POSTROUTING 时刻点进行 NAT 处理。IP 报文的源地址是 IP3（私网 IP），这个报文最后经过 POSTROUTING 这个时刻点时，如果需要（即提前做了相关 NAT 配置），NAT 模块会做处理。NAT 模块会将源 IP 从 IP3 转换成 IP1（这个是提前配置好的），这也就是所谓的 SNAT。这个 IP1 就是 Linux 内核空间对外（公网）呈现的 IP 地址（说明，这样的 IP 地址可以有多个）。

（3）NAT-OUTPUT（DNAT）

NAT-OUTPUT（DNAT）的处理时刻点，如图 2-23 所示：

图 2-23 给人一种“迷惑 / 诡异”的感觉，这个 IP 报文是谁发出来的？如果我们把 Linux 内核空间（Netfilter）往“外”设想一下，把它想象成一个网元，比如防火墙（防火墙里可以有 NAT 功能），这个防火墙自己对外发送一个报文。这个报文在 D 处，即 OUTPUT 时刻点，会

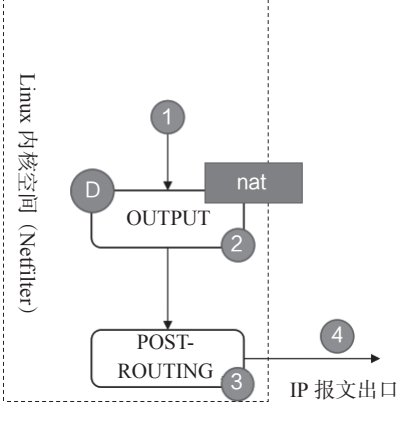


图 2-23　NAT-OUTPUT（DNAT）

做一个 DNAT。这样这个报文不需要在“3”处，即 POSTROUTING 时刻点再做 NAT，因为内核空间的 IP 源地址已经是公网 IP，而目的地址已经在“2”处，即 D 处 /OUTPUT 时刻点已经做了 DNAT。

（4）小结

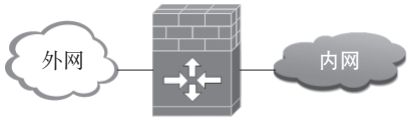
Linux 内核空间 Netfilter 模块的 NAT 处理，一共有三个 Chain(处理时刻点)，如表 2-5 所示：

表 2-5 Linux 内核空间 Netfilter 模块的 NAT 处理

流	流描述	Chain	NAT 类型	NAT 说明
流 1	流从外部到达 Linux 用户空间（私网 IP）	PREROUTING	DNAT	将目的 IP 从公网 IP（Linux 内核空间对应的 IP）转换到私网 IP（Linux 用户空间对应的 IP）
流 2	流从 Linux 用户空间（私网 IP）到达外部	POSTROUTING	SNAT	将源 IP 从私网 IP（Linux 用户空间对应的 IP）转换到公网 IP（Linux 内核空间对应的 IP）
流 3	流从 Linux 内核空间（公网 IP）到达外部	OUTPUT	DNAT	

2.7.2 Firewall

iptables 中的 Firewall（防火墙）概念，属于网络防火墙的概念，如图 2-24 所示：



【方案一】
if 符合某些规则，通过
else 不能通过
【方案二】
if 符合某些规则，不能通过
else 通过

图 2-24 防火墙基本概念

图中的“某些”规则决定了其是否属于“网络”防火墙。iptables 中的防火墙的这些规则就是基于 TCP/IP 协议栈的规则，所以我们称之为网络防火墙。这些规则有：

1）in-interface（入网络接口名），数据包从哪个网络接口进入；

2）out-interface（出网络接口名），数据包从哪个网络接口输出；

3）protocol（协议类型），数据包的协议，如 TCP、UDP 和 ICMP 等；

4）source（源地址（或子网）），数据包的源 IP 地址（或子网）；

5）destination（目标地址（或子网）），数据包的目标 IP 地址（或子网）；

6）sport（源端口号），数据包的源端口号；

7）dport（目的端口号），数据包的目的端口号。

符合这些规则的，可以设置为通过（ACCEPT），反之，则不通过（DROP）。或者，符合这些规则的，设置为不通过（DROP）；反之，则通过（ACCEPT）。

比如，我们在前面介绍的一个例子：

```
iptables -A INPUT -i eth0 -p icmp -j ACCEPT
```

这个就表示：所有从 eth0 端口进入且协议是 ICMP 的报文可以通过（ACCEPT）。如果仅有此一条规则，那么潜台词就是：其余的报文，都不允许通过（DROP）。

Netfilter 中的 Firewall，会在三个时刻点，进行处理，如图 2-25 所示：

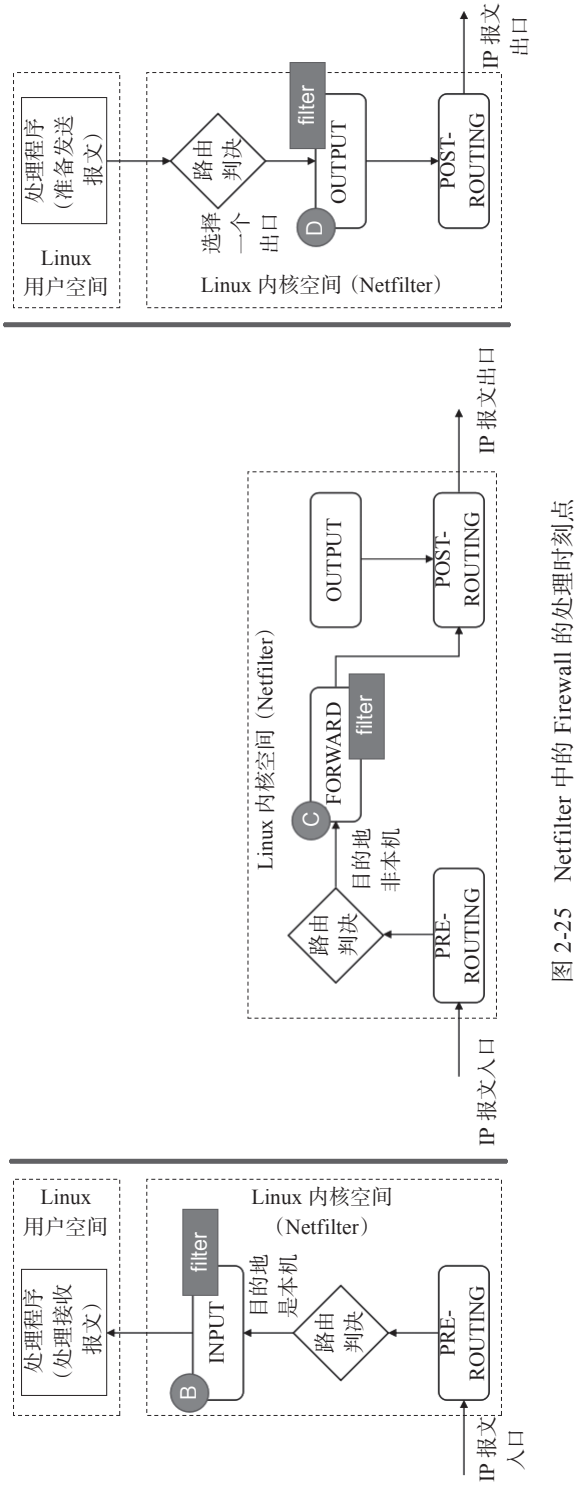


图 2-25　Netfilter 中的 Firewall 的处理时刻点

图 2-25 中三部分代表三种流，每种流的处理时刻点图中已经标明，这里不再详叙。

2.7.3 mangle

mangle 表主要用于修改数据包的 ToS（Type of Service，服务类型）、TTL（Time to Live，生存周期）以及为数据包设置 Mark 标记，以实现 QoS(Quality of Service，服务质量）调整以及策略路由等应用。

netfilter 模块中的 mangle 处理的时刻点如图 2-26 所示：

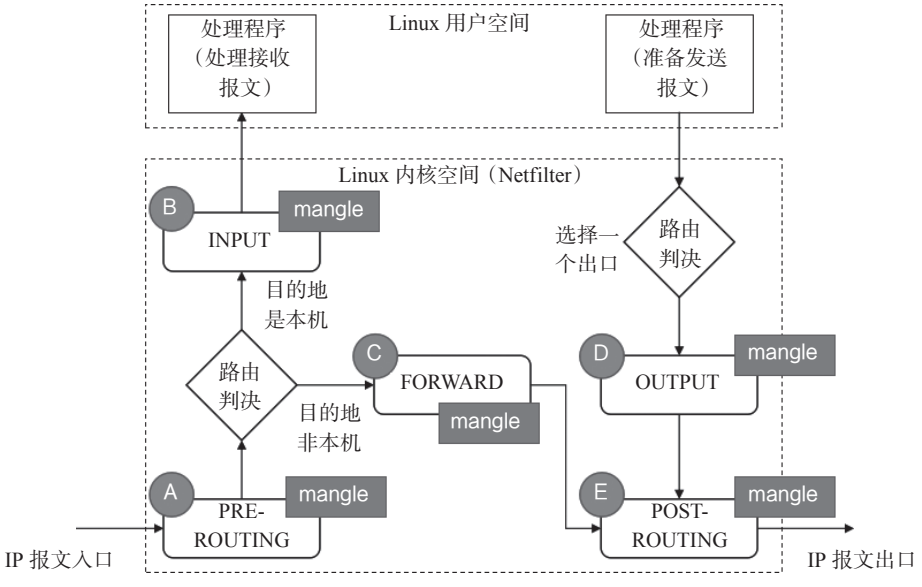


图 2-26 Netfilter 中的 Mangle 的处理时刻点

2.8 本章小结

tap、tun、veth pair 在 Linux 中都被称为设备，但是在与日常概念的类比中，常常被称作接口。Neutron 利用这些“接口”进行 Bridge 之间的连接、Bridge 与 VM(虚拟机）的连接、Bridge 与 Router 之间的连接。三者与物理网卡之间的对比关系，如图 2-27 所示。

反而是 Router、Bridge 这些在 Linux 中没有被称为设备的网络功能，反而在日常概念中常常被称为设备。Bridge 提供二层转发功能，Router 提供三层转发功能。Router 还常常借助 iptable 提供 SNAT/DNAT 功能。Bridge 也常常借助 iptable 提供 Firewall 功能。

在 Neutron 中，隔离是一个非常重要的特性，利用 namespace 做隔离也是 Neutron 的一个非常重要的手段。

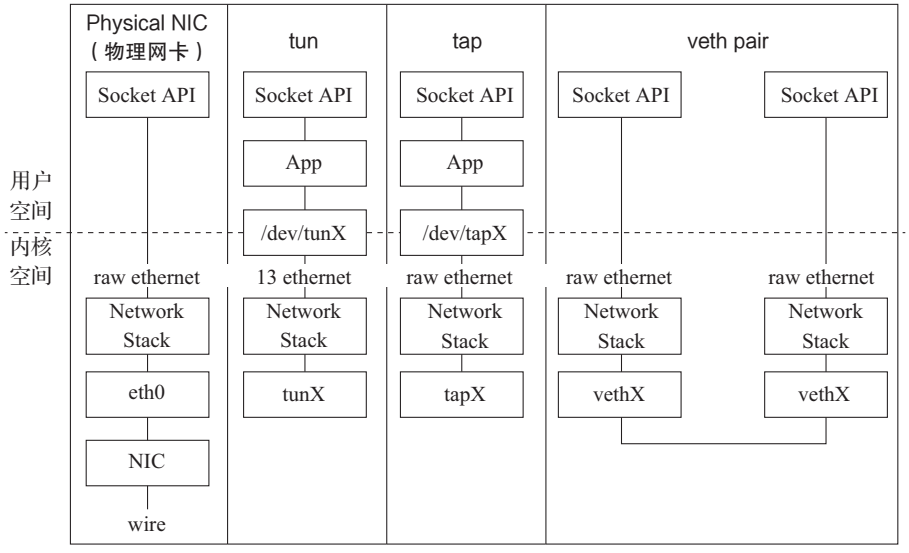


图 2-27　物理与虚拟设备对比关系