

2015

TCP/IP 网络协议栈

基于 linux 3.10

分为上篇和下篇，上篇共九章，侧重于 TCP/IP 数据收发流程，即 OSI 模型的 IP 和 TCP 层，下篇也是九章，并不属于 TCP/IP 本身，但是多少和网络有关且常用到，比如 LC-trie 路由、netfilter 包过滤防火、还有一些网络相关的命令行工具等，文末给出 IPV6 的协议栈模型图，此外还给出了测试源码。

葛世超
定价 0 元
2015/4/22



第零章 内核网络相关配置选项	5
0.1 Kconfig 选项	5
0.2 ip-sysctl 意义	8
第一章 网络子系统初始化	9
1.1 网络初始化函数调用顺序	9
1.2 调用函数浅析	12
1.3 inet_init	16
1.4 总结	21
第二章 主机到网络层（网卡）	22
2.1 TCP/IP 协议栈模型	22
2.2 网卡数据结构	22
2.3 网卡注册流程	26
第三章 套接字相关数据结构	31
3.1 socket 对应的内核结构体	31
3.2 struct proto_ops	33
3.3 struct proto	33
3.4 sk_buff(SKB)	34
3.5 softnet_data	35
3.6 struct packet_type	36
3.7 一些名词简称	36
第四章 网络层接收数据包流程	37
4.1 主机到网络层的过渡	37
4.2 进入网络层	40
第五章 传输层（tcp）到网络层（ip）	44
第六章 应用层	49
第七章 tcp 发送（传输层）	55
7.2 MSS	69
第八章 tcp 接（传输层）	70
第九章 tcp 拥塞控制	79
9.1 CUBIC 拥塞控制	79
cubic 使用的算法	79

cubic 慢启动门限阈值.....	79
9.2 cubic 拥塞代码实现.....	82
慢启动 slow start	82
拥塞避免 congestion avoid	83
快速重传和快速恢复.....	85
下篇 杂项汇总	86
第十章 网络工具	87
10.1 ss.....	87
10.2 netstat	88
列标题	88
常用选项	89
10.3 netstress	89
10.4 netperf 参考	90
10.5 iperf	90
10.6 iptraf	90
10.7 TcpDump	91
10.7.1 数据过滤	91
10.7.2 输入输出	92
10.8 nicstat	92
10.8.1 nicstat 的安装:	92
10.8.2 nicstat 使用	93
10.9 ethtool 工具:	95
第十一章 Linux 包过滤防火墙-netfilter iptables	96
11.1 netfilter 框架	96
11.2 防火墙规则表	104
11.2.1 xt_init 初始化防火墙表	106
11.2.2 规则的组成.....	108
11.3 防火墙规则遍历	110
11.3.2 Hook 函数	117
11.4 iptables	123
第十二章 路由	124
12.1 路由核心数据结构	124

12.2 LC-trie (字典树、单词查找树)	127
12.3 ifconfig	132
12.3.1 /proc/net/路由下路由信息.....	132
12.3.2 路由通知链函数的注册	133
12.3.3 ifconfig 调用流程	134
12.3.4 put_child.....	153
12.4 route 添加路由项	154
12.5 路由缓存	161
12.5.1 路由缓存的查找	161
12.5.2 路由缓存的创建	162
12.5.3 路由缓存的内存管理	166
12.6 路由查找	166
12.6.1 相关数据结构	166
12.6.2 接收包路由项查找	170
第十三章 网络命名空间	177
13.1 命名空间创建	177
13.2 网络命名空间管理	178
第十四章 netlink 机制	181
14.1 netlink 支持的通信	181
14.2 netlink 用户空间 API	184
14.3 netlink 内核空间 API	184
第十五章 提升网络性能技术	188
15.1 TSO/GSO	189
15.2 LRO/GRO.....	191
15.3 RSS (Receive Side Scaling) 队列:	196
15.4 RPS (Receive Packet Steering) 队列:	196
15.5 RFS(Receive Flow Steering), Accelerated Receive Flow Steering.....	199
15.6 XPS (Transmit Packet Steering)	200
第十六章 PHY	203
16.1 PHY	203
16.2 MAC 驱动	205
16.3 PHY 驱动	209

16.3.1 PHY 初始化.....	209
16.3.2 PHY 驱动实例.....	211
16.3.3 PHY 状态机.....	213
第十七章 ping-icmp.....	218
第十八章 ipv6 简介	223
附录 tcp 测试程序.....	224
TCP/IP 状态图	227
参考文献.....	228

第零章 内核网络相关配置选项

0.1 Kconfig 选项

基于嵌入式，可能略有不同

packet protocol 被直接和网络设备通信的应用程序使用，其没有使用内核的其它协议，像 tcpdump 支持需要使能该选项，af_packet。

`<*> Packet socket`

支持 PF_PACKET 套接字，ss 之类工具监控接口（eth0...）会使用这类套接字

`<> Packet: sockets monitoring interface`

//UNIX 域套接字，即使没有联网 Xwindow 和 syslog 也会使用 UNIX 域套接字。强烈建议该选项为 Y

`<*> Unix domain sockets`

支持 ss 工具使用的 Unix 域套接字来监控 interface

`<*> UNIX: socket monitoring interface`

支持 XFRM (Transformation)，对接收到的数据包经过路由时会被修改；

`<> Transformation user configuration interface`

`[] Transformation sub policy support`

`[] Transformation migrate database`

`[] Transformation statistics`

PF_KEYv2 套接字协议族，如果使用移植于 KAME 的 IPsec 工具，该选项需要。

`<> PF_KEY sockets`

会使内核增加 400KB

`[*] TCP/IP networking`

多播，内核增加 2KB，对于 MBONE (Multicast backbone)，一个应用场景是影音节目的全球广播。

`[*] IP: multicasting`

这个选项用于支持网络数据包的 forward 和 redistribute，并不包括路由的基本配置。

`[*] IP: advanced router`

//路由的 TRIE 表统计，测试 TRIE 算法的性能

`[] FIB TRIE statistics`

通常路由根据接收到的数据包最终目的地址决策数据包的命运，如果使用策略路由，那么源地址、TOS 也会被考虑进去。

`[] IP: policy routing`

通常，对一个数据包路由表会明确给出一个路径；如果配置该选项，对一个给定的数据包将可能存在多种路径，路由会将这些路径当成开销是一样的，对路径的选择将是不确定的。

`[] IP: equal cost multipath`

klogd 将导出路由信息。

[] IP: verbose route monitoring

内核启动时将允许设备的 IP 地址和路由表的自动配置。配置的依据是内核命令行或者 BOOTP、RARP 协议。无盘系统启动需要配置此选项。

[] IP: kernel level autoconfiguration

隧道，将一个协议的数据封装在另一个协议中，通过一个支持封装协议的通道发送。这里是 IP 封装 IP 的隧道支持，可用于支持主机伪装和移动 IP

< > IP: tunneling

解 GRE (Generic Routing Encapsulation) 包，使用 ip_gre 和 pptp (point to point Tunning Protocol) 点对点隧道协议，则需要配置该选项。

< > IP: GRE demultiplexer

多目的地址路由支持。MBNOE

[] IP: multicast routing

内核维持一个 IP 映射到 MAC 的 cache，ARP 协议负责该映射，如果想支持用户空间 daemon 完成地址解析，这里配置上就行

[] IP: ARP daemon support

TCP/IP 网络易受 SYN 攻击，DOS 攻击阻止了合法用户建立连接；SYN cookie 方法使用加密的方法能够在主机收到攻击时仍然可以通信。

[] IP: TCP syncookie support

支持 IPsec AH (Authentication Header)，见 <http://en.wikipedia.org/wiki/IPsec>

< > IP: AH transformation

支持 IPsec ESP (Encapsulating Security Payload)

< > IP: ESP transformation

IP Payload Compression Protocol (IPComp) (RFC3173), IPsec 需要

< > IP: IPComp transformation

Support for IPsec transport mode

< > IP: IPsec transport mode

Support for IPsec tunnel mode

< > IP: IPsec tunnel mode

Support for IPsec BEET mode

< > IP: IPsec BEET mode

Support for Large Receive Offload (ipv4/tcp)

<*> Large Receive Offload (ipv4/tcp)

Support for INET (TCP, DCCP, etc) socket monitoring interface used by native Linux tools such as ss. ss is included in iproute2

< > INET: socket monitoring interface

various TCP congestion control CUBIC TCP、H-TCP、TCP Westwood+、Binary Increase Congestion (BIC) control，默认使用 cubic 算法

[] TCP: advanced congestion control --->

RFC2385 specifies a method of giving MD5 protection to TCP sessions.

[] *TCP: MD5 Signature Option support (RFC2385)*

<*> *The IPv6 protocol* --->

网络数据包 security marking

[] *Security Marking*

PHY 设备对数据包进行时间戳标记

[] *Timestamping in PHY devices*

netfilter, 1、透明代理 2、包过滤防火墙。

[*] *Network packet filtering framework (Netfilter)* --->

Datagram Congestion Control Protocol

< > *The DCCP Protocol* --->

Stream Control Transmission Protocol

< > *The SCTP Protocol* --->

RDS (Reliable Datagram Sockets) protocol, provides reliable, sequenced delivery of datagrams over Infiniband, iWARP, or TCP.

< > *The RDS Protocol*

Transparent Inter Process Communication (TIPC) protocol,

< > *The TIPC Protocol* --->

ATM is a high-speed networking technology for Local Area Networks and Wide Area Networks.

< > *Asynchronous Transfer Mode (ATM)*

对于 PVC (permanent virtual circuit) 和 SVC (switched virtual circuits) 下的基于 ATM (Asynchronous Transfer Mode) 的经典 IP 支持

<M> *Classical IP over ATM*

如果邻居没有发现时，则不发送“ICMP host unreachable”消息

[] *Do NOT send ICMP if no neighbor*

模拟 LAN

<M> *LAN Emulation (LANE) support*

ATM 之上的 Multi-Protocol 使得 ATM 边缘设备（边缘设备是指提供服务入口点的设备，如路由器等）和 ATM 主机在子网边界建立直接的 ATM 虚拟电路。

<M> *Multi-Protocol Over ATM (MPOA) support*

<M> *RFC1483/2684 Bridged protocols*

[] *Per-VC IP filter kludge*

< > *Layer Two Tunneling Protocol (L2TP)* --->

以太网桥支持。

< > *802.1d Ethernet Bridging*

[*] *IGMP/MLD snooping*

[] *VLAN filtering (NEW)*

802.1Q VLAN 支持

802.1Q/802.1ad VLAN Support

GVRP (GARP VLAN Registration Protocol) support

MVRP (Multiple VLAN Registration Protocol) support (NEW)

DECnet Support

ANSI/IEEE 802.2 LLC type 2 Support

The IPX protocol

Appletalk protocol support

CCITT X.25 Packet Layer

LAPB Data Link Driver

Phonet protocols family

IEEE Std 802.15.4 Low-Rate Wireless Personal Area Networks support

QoS and/or fair queueing --->

Data Center Bridging support

B.A.T.M.A.N. Advanced Meshing Protocol

BLA (Bridge Loop Avoidance)

Bridge Loop Avoidance

Open vSwitch < > Virtual Socket protocol

NETLINK: mmapped IO

NETLINK: socket monitoring interface

Network priority cgroup

Berkeley Packet Filter filtering, /proc/sys/net/core/bpf_jit_enable

enable BPF Just In Time compiler

0.2 ip-sysctl 意义

见 Documentation/networking/ip-sysctl.txt

第一章 网络子系统初始化

1.1 网络初始化函数调用顺序

《Linux 系统启动那些事—基于 Linux 3.10 内核》提到系统启动时会调用一系列的初始化函数，初始化函数使用 `include/init.h` 中的宏定义，这些宏的顺序显示了初始化函数调用的顺序。即由 `pure_initcall` 函数定义的函数先于 `core_initcall` 定义的函数，依此类推。

```
#define pure_initcall(fn) __define_initcall(fn, 0)
#define core_initcall(fn) __define_initcall(fn, 1)
#define core_initcall_sync(fn) __define_initcall(fn, 1s)
#define postcore_initcall(fn) __define_initcall(fn, 2)
#define postcore_initcall_sync(fn) __define_initcall(fn, 2s)
#define arch_initcall(fn) __define_initcall(fn, 3)
#define arch_initcall_sync(fn) __define_initcall(fn, 3s)
#define subsys_initcall(fn) __define_initcall(fn, 4)
#define subsys_initcall_sync(fn) __define_initcall(fn, 4s)
#define fs_initcall(fn) __define_initcall(fn, 5)
#define fs_initcall_sync(fn) __define_initcall(fn, 5s)
#define rootfs_initcall(fn) __define_initcall(fn, rootfs)
#define device_initcall(fn) __define_initcall(fn, 6)
#define device_initcall_sync(fn) __define_initcall(fn, 6s)
#define late_initcall(fn) __define_initcall(fn, 7)
#define late_initcall_sync(fn) __define_initcall(fn, 7s)
```

网络子系统代码定义于 `net` 目录和 `drivers/net` 目录下，前一个 `net`/ 目录包含偏协议栈上层的代码，而 `drivers/net` 目录下的代码则更多的偏于协议栈下层，即数据链路层和物理层（主要 MAC 和 PHY 驱动，链路层协议的其它部分由硬件实现）。千兆网卡的数据链路层和物理层包含的内容如下图所示。

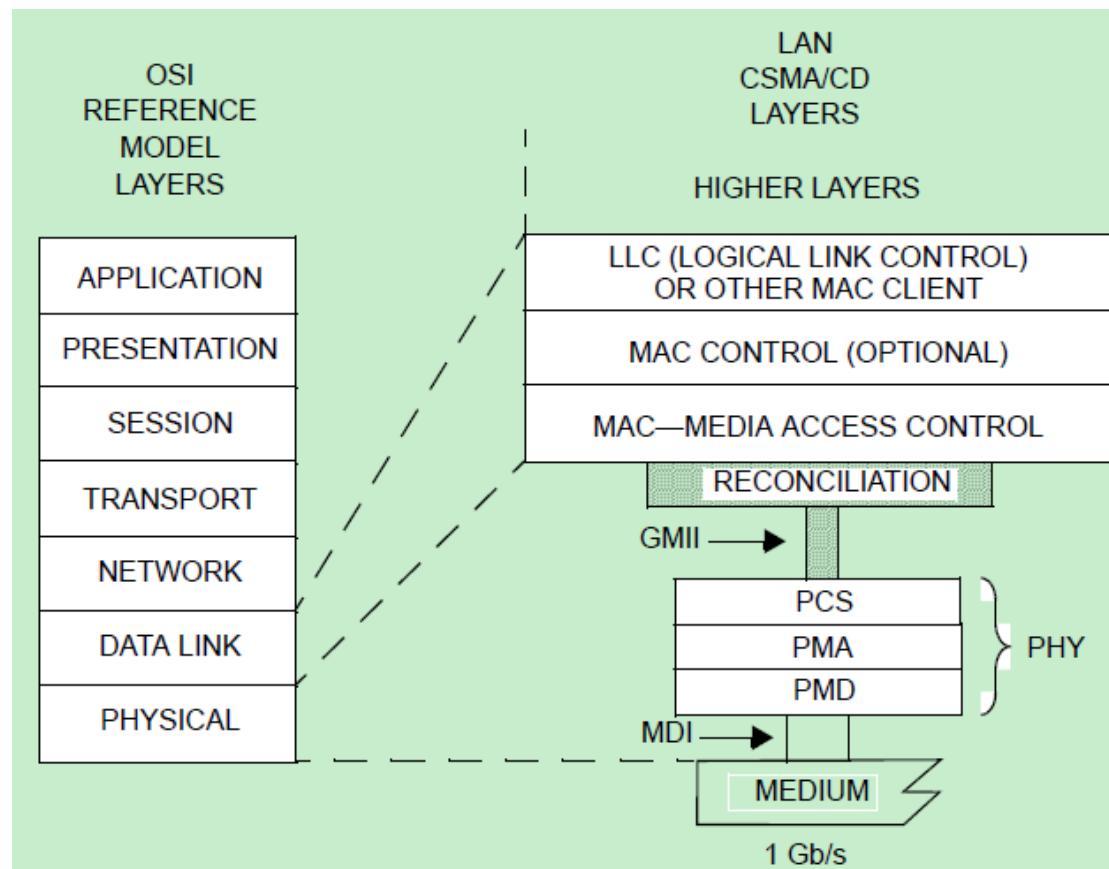


图 1.1 GMII OSI 参考模型，摘自 IEEE802.3_2012 section3

在上述的两个目录下找到如下和网络初始化相关的函数：

net/ 目录下：

```
./core/net_namespace.c:436:pure_initcall(net_ns_init);
./core/sock.c:2566:core_initcall(net_inuse_init);
./core/netpoll.c:1218:core_initcall(netpoll_init);
./socket.c:2654:core_initcall(sock_init);           /* early initcall */
./netlink/af_netlink.c:2940:core_initcall(netlink_proto_init);
./core/fib_rules.c:772:subsys_initcall(fib_rules_init);
./core/sock.c:2844:subsys_initcall(proto_init);
./core/neighbour.c:3036:subsys_initcall(neigh_init);
./core/dev.c:6336:subsys_initcall(net_dev_init);
./nfc/core.c:973:subsys_initcall(nfc_init);
./nfc/hci/core.c:950:subsys_initcall(nfc_hci_init);
./mac80211/main.c:1157:subsys_initcall(ieee80211_init);
./irda/irmod.c:205:subsys_initcall(irda_init);
./atm/common.c:906:subsys_initcall(atm_init);
./ieee802154/wpan-class.c:213:subsys_initcall(wpan_phy_class_init);
./wireless/core.c:1152:subsys_initcall(cfg80211_init);
./wireless/wext-core.c:366:subsys_initcall(wireless_nlevent_init);
./rfkill/core.c:1292:subsys_initcall(rfkill_init);
./sched/act_api.c:1138:subsys_initcall(tc_action_init);
./sched/sch_api.c:1831:subsys_initcall(pktsched_init);
./sched/cls_api.c:627:subsys_initcall(tc_filter_init);
./iucv/iucv.c:2122:subsys_initcall(iucv_init);
./netlink/genetlink.c:1012:subsys_initcall(genl_init);
./bluetooth/af_bluetooth.c:723:subsys_initcall(bt_init);
./ipv4/cipso_ipv4.c:2365:subsys_initcall(cipso_v4_init);
./netlabel/netlabel_kapi.c:1100:subsys_initcall(netlbl_init);
./core/sysctl_net_core.c:266:fs_initcall(sysctl_core_init);
./ipv6/ip6_offload.c:281:fs_initcall(ipv6_offload_init);
./sunrpc/sunrpc_syms.c:126:fs_initcall(init_sunrpc);
./ipv4/af_inet.c:1691:fs_initcall(ipv4_offload_init);
./ipv4/af_inet.c:1826:fs_initcall/inet_init);
./unix/af_unix.c:2454:fs_initcall(af_unix_init);
./ipv4/tcp_fastopen.c:92:late_initcall(tcp_fastopen_init);
./ipv4/tcp_cong.c:145:late_initcall(tcp_congestion_default);
./ipv4/ipconfig.c:1543:late_initcall(ip_auto_config);
```

drivers/net 目录下：

```
./phy/phy_device.c:1146:subsys_initcall(phy_init);
```

phy 的初始化见第十一章。

还有一类函数也会在系统初始化时被调用，这些函数使用 module_init 宏进行定义了，
基于 tcp/ip V4 协议的 module_init 宏在 /net/ipv4/ 目录下，module_init 定义的相关函数如下。

```
//cubic 算法是 Linux 现在采用的拥塞控制算法。
./tcp_cubic.c:488:module_init(cubictcp_register);
//TCPW 是专门针对无线网环境提出的协议，针对 ACK 估算流量。
./tcp_westwood.c:299:module_init(tcp_westwood_register);
./xfrm4_mode_transport.c:77:module_init(xfrm4_transport_init);
// tunnel 即隧道，被用于在公网上传输私网数据，也就是 VPN。实现类似于数据结构中的栈，把数据报文封装在新的报文中，通过第三方协议(比如 IP 协议)传输到对端，对端进行解封，重新路由。
./tunnel4.c:190:module_init(tunnel4_init);
./esp4.c:739:module_init(esp4_init);
./ipcomp.c:192:module_init(ipcomp4_init);
./tcp_veno.c:229:module_init(tcp_veno_register);
./tcp_bic.c:237:module_init(bictcp_register);
```

```

./xfrm4_tunnel.c:114:module_init(ipip_init);
./gre.c:247:module_init(gre_init);
./tcp_vegas.c:334:module_init(tcp_vegas_register);
./xfrm4_mode_beet.c:153:module_init(xfrm4_beet_init);
./ah4.c:553:module_init(ah4_init);
./tcp_yeah.c:255:module_init(tcp_yeah_register);
./tcp_illinois.c:352:module_init(tcp_illinois_register);
./tcp_hybla.c:187:module_init(hybla_register);
./netfilter.c:206:module_init(ipv4_netfilter_init);
./inet_diag.c:1199:module_init(inet_diag_init);
./tcp_scalable.c:57:module_init(tcp_scalable_register);
./ip_vti.c:899:module_init(vti_init);
./tcp_diag.c:66:module_init(tcp_diag_init);
./tcp_lp.c:339:module_init(tcp_lp_register);
./tcp_htcp.c:310:module_init(htcp_register);
./udp_diag.c:212:module_init(udp_diag_init);
./ip_gre.c:1018:module_init(ipgre_init);
./xfrm4_mode_tunnel.c:190:module_init(xfrm4_mode_tunnel_init);
./tcp_probe.c:252:module_init(tcpprobe_init);
./tcp_highspeed.c:182:module_init(hstcp_register);
netfilter 是基于包过滤防火墙相关内容。其内容见第十章。
./netfilter/nf_nat_pptp.c:310:module_init(nf_nat_helper_pptp_init);
./netfilter/ipt_ah.c:90:module_init(ah_mt_init);
./netfilter/iptable_raw.c:88:module_init(iptable_raw_init);
./netfilter/nf_conntrack_l3proto_ipv4.c:552:module_init(nf_conntrack_l3proto_ipv4_init);
./netfilter/ipt_CLUSTERIP.c:748:module_init(clusterip_tg_init);
./netfilter/iptable_security.c:109:module_init(iptable_security_init);
./netfilter/arp_tables.c:1913:module_init(arp_tables_init);
./netfilter/ipt_ULOG.c:496:module_init(ulog_tg_init);
./netfilter/ipt_MASQUERADE.c:177:module_init(masquerade_tg_init);
./netfilter/iptable_mangle.c:147:module_init(iptable_mangle_init);
./netfilter/ipt_ECN.c:137:module_init(ecn_tg_init);
./netfilter/ip_tables.c:2269:module_init(ip_tables_init);
./netfilter/iptable_nat.c:333:module_init(iptable_nat_init);
./netfilter/nf_nat_proto_gre.c:142:module_init(nf_nat_proto_gre_init);
./netfilter/ipt_REJECT.c:212:module_init(reject_tg_init);
./netfilter/nf_nat_snmp_basic.c:1311:module_init(nf_nat_snmp_basic_init);
./netfilter/arpt_mangle.c:90:module_init(arpt_mangle_init);
./netfilter/iptable_filter.c:109:module_init(iptable_filter_init);
./netfilter/nf_defrag_ipv4.c:125:module_init(nf_defrag_init);
./netfilter/nf_nat_l3proto_ipv4.c:280:module_init(nf_nat_l3proto_ipv4_init);
./netfilter/ipt_rpfilter.c:146:module_init(rpfilter_mt_init);
./netfilter/nf_nat_h323.c:622:module_init(init);
./netfilter/arptable_filter.c:90:module_init(arptable_filter_init);
./ipip.c:481:module_init(ipip_init);

```

在 net/xfrm 目录下的各文件大致功能如下，该目录内核主要实现 IPsec 协议，和 TCP、IP 协议位置是等同的一个协议。

xfrm_state.c: xfrm 状态管理
 xfrm_policy.c: xfrm 策略管理
 xfrm_algo.c: 算法管理
 xfrm_hash.c: HASH 计算函数
 xfrm_input.c: 安全路径(sec_path)处理,用于进入的 ipsec 包
 xfrm_user.c: netlink 接口的 SA 和 SP 管理
 在 net/ipv4 目录下的和 ipsec 相关各文件大致功能说明如下:
 ah4.c: IPV4 的 AH 协议处理
 esp4.c: IPV4 的 ESP 协议处理
 ipcomp.c: IP 压缩协议处理

xfrm4_input: 接收的 IPV4 的 IPSEC 包处理
xfrm4_output: 发出的 IPV4 的 IPSEC 包处理
xfrm4_state: IPV4 的 SA 处理
xfrm4_policy: IPV4 的策略处理
xfrm4_tunnel: IPV4 的通道处理
xfrm4_mode_transport: 传输模式
xfrm4_mode_tunnel: 通道模式
xfrm4_mode_beet: BEET 模式

1.2 调用函数浅析

1.2.1 Core 目录下

`net_ns_init` 是网络命名空间初始化函数, Linux 目前实现了六种命名空间, 分别是 `mount` 命名空间, `UTS` 命名空间, `IPC` 命名空间, `PID` 命名空间, 网络命名空间和 `user` 命名空间, 这些命名空间目前最主要的应用是在虚拟化技术上, 关于命名空间入门文章, 可以参看《linux namespace-之使用》一文, 这里初始化只完成了 `root` 用户命名空间的初始化。进程的

`struct nsproxy *nsproxy;` 字段指向命名空间。

`net_inuse_init` 注册一个网络命名空间子系统, 将该子系统挂载在 `first_device` 表示的链表上, 以后在创建新的命名空间时也会挂接到该链表上, 并将该网络命名空间子系统和 `net_inuse_ops` 包含的 `init` 和 `exit` 函数进行绑定, 后面再创建或者销毁网络命名空间是相应的调用这里的 `init` 和 `exit` 函数。

`netpoll_init` 初始化内核下的 `sk_buff(skb_pool)`, `netpoll` 是一个框架和一些接口, 其和网络的关系非常类似于 `VFS` 和文件系统的关系, 主要用于网络控制台 `net console` 和内核远程调试 `KGDBoE`。

`sock_init` 完成

- 1、网络 `sysctl` 接口注册, `sysctl` 是将内核参数导出到 `proc` 目录下, 并且数可以在用户空间修改, Linux 服务器性能调优时对该 `sysctl` 参数设置的比较多。
- 2、该函数还会初始化两个 `slab` 缓存, 它们是 `skbuff_head_cache` 和 `skbuff_fclone_cache`, 使用 `slab` 缓存而不是 `kmalloc` 动态申请内存的原因是速度, Linux 网络通信使用的是 `socket` 编程, 用户空间的数据以及网络协议栈的各种头信息在数据在协议栈中传递时使用上述两种 `cache`, 这两种 `cache` 的区别体现在 `sk_buff` 内的数据是否需要改变, 如果需要改变, 那么其它使用该 `sk_buff` 的进程将可能看到不一致的 `sk_buff`, `skbuff_fclone_cache` 就是针对这种场景而生的。
- 3、套接字文件系统初始化, 对于网络数据和其它类型的数据一样都是通过文件系统方式存取的, 将 `sock_fs_type` 结构体表示的类型文件系统注册到 `file_systems` 文件系统全局链表上。注册成功后会挂载该文件系统。
- 4、注册 `netfilter` 的 `hook` 函数, `netfilter` 和 `iptables` 被称为 Linux 防火墙, `iptables` 是用户空间配置网络防火墙规则的接口, 这些用户空间的规则会转换为内核空间 `netfilter` 的规则表中的规则, `netfilter` 基于包过滤原则, 涉及规则、表以及 `hook` 检测点三个部分。

`netlink_proto_init`: 初始化 `netlink` 机制, 该机制用于应用程序和内核通信之用。该函数将 `netlink` 协议初始化到网络协议链表中, 然后为 `netlink` 机制分配 32 个 `netlink_table` 表结构, 并且初始化该结构的 `hash` 表头。然后调用 `netlink_add_usersock_entry` 将上面 32 个 `netlink_table` 表中的一个初始化为 `NETLINK_USERSOCK` 类型的表, 该 `netlink` 表用户和用户空间通信, 此外还会调用 `sock_register` 注册 `netlink` 协议族操作集, 该操作集中的 `netlink_create`

用于初始化一个 sock，最后改函数会在 proc 文件系统下注册 netlink 文件。netlink 的用户空间编程可以参考 generic_netlink_howto。

`fib_rules_init` 路由初始化，路由在内核中称为 FIB (forward information base)，该函数除了对每一个网络命名空间初始化 `rules_ops` 和 `rules_mod_lock` 字段，该函数首先利用上面的 netlink 机制注册三个用户空间和内核路由子系统通信的辅助函数，它们是新路由规则的添加，路由规则的删除和路由规则导出。然后使用通知链机制 (notification chain) 注册一个有关路由的回调函数，当网络设备注册和注销时，该网络设备对应的路由项相应的会需要相应的添加和删除操作，这个功能就是这里注册的回调函数完成的。`proto_init`，在 proc 目录下为每一个网络命名空间创建 `protocols` 目录。

`neigh_init`: 邻居协议初始化。

`net_dev_init` 函数初始化 DEV 模块，在系统启动阶段调用该函数遍历设备列表并且过滤掉初始化失败的设备。

在 /proc/net 目录下创建 `ptype`, `softnet_stat`, `dev`; 这三个文件是全局性的，反映了系统网络情况，`dev` 是针对设备的统计，有几张网络对应几个 eth 端口，外加 lo (回环)，其反映的是系统接收收数据包的情况，包括，接收到的总字节数、包数量、错误数等；`ptype` 是按照包的类型统计的，`ipv4`、`arp`、`ipv6` 等。内核打印 `ptype` 信息的函数是：

```
static int ptype_seq_show(struct seq_file *seq, void *v)
{
    struct packet_type *pt = v;

    if (v == SEQ_START_TOKEN)
        seq_puts(seq, "Type Device      Function\n");
    else if (pt->dev == NULL || dev_net(pt->dev) == seq_file_net(seq)) {
        if (pt->type == htons(ETH_P_ALL))
            seq_puts(seq, "ALL ");
        else
            seq_printf(seq, "%04x", ntohs(pt->type));

        seq_printf(seq, " %-8s %pf\n",
                  pt->dev ? pt->dev->name : "", pt->func);
    }
    return 0;
}
```

`softnet_stat` 则是区分 CPU 个数的，几个就对应几行，每一行是一颗 CPU 的统计数据。其打印这些信息的函数是：

```
static int softnet_seq_show(struct seq_file *seq, void *v)
{
    struct softnet_data *sd = v;

    seq_printf(seq, "%08x %08x %08x %08x %08x %08x %08x %08x %08x\n",
               sd->processed, sd->dropped, sd->time_squeeze, 0,
               0, 0, 0, 0, /* was fastroute */
               sd->cpu_collision, sd->received_rps);

    return 0;
}
```

该函数接下来是 `ptype_base` 和 `ptype_all` 链表的初始化，这些链表用于组织协议处理函数，见图 1.2，`dev_add_pack` 函数会将协议添加到该链表上，哈希的方法是去主机序的低四个比特作为 hash 的键值，这些协议类型如下：

* 0800 IP
* 8100 802.1Q VLAN
* 0001 802.3

```
* 0002 AX.25
* 0004 802.2
* 8035 RARP
* 0005 SNAP
* 0805 X.25
* 0806 ARP
* 8137 IPX
* 0009 Localtalk
* 86DD IPv6
```

可以看到，冲突的只有 RARP/RARP/X.25，查找冲突时会遍历该链表。

`offload_base` 是和 `gso/gro` 有关的队列，将分段操作推迟到网卡硬件完成。见十五章。

调用 `netdev_init` 函数对网络命名空间中的每一个网络 `struct net` 中的 `dev_name_head` 和 `dev_index_head` 链表初始化。一个网络子命名空间可能使用多个网卡，这里的 `name` 和 `index` 就是用来跟踪这些网卡信息的，回环也被作为一个网卡来看待。

接下来会完成接收队列的初始化，这个初始化是针对 per-CPU 类型的变量 `softnet_data`, `softnet_stat` 的统计信息源于此。

`dev_boot_phase` 赋值成零，这个变量存在没有什么意义了，初始化完成就将其设置成 0，如果下一次再次进入 `net_dev_init()` 函数，可以断定出错了，这也是该函数开始处 `BUG_ON(!dev_boot_phase)` 的意义所在。

注册回环设备，回环设备在调试网络协议栈的正确性还是挺有帮助的，另外还注册了一个网络设备被 `remove` 了的操作函数集。

然后，注册了两个软中断服务函数，它们分别是数据包发送和数据包接收服务函数：

```
open_softirq(NET_TX_SOFTIRQ, net_tx_action);
open_softirq(NET_RX_SOFTIRQ, net_rx_action);
```

最后注册了两个通知链，一个用于 CPU 的热插拔事件，一个用于网卡注销或者 `down` 的事件处理，主要工作就是将该网卡对应的路由项禁掉：

```
hotcpu_notifier(dev_cpu_callback, 0); //注册 CPU 状态变化时的回调函数。
dst_init();注册一个通知函数，当网络设备或者端口状态变化时会回调 dst_dev_event 函数，该函数根据端口可用与否，对缓存的路由项进行管理。
```

`tc_action_init()` 注册三个 `netlink` 函数，

`tc_filter_init`

用于流量控制，

`cipso_v4_init`

`netlbl_init`: 早期 Linux 关注于本地数据安全性，并不太关注网络上数据通信的安全。

`netlabel` 增加了内核对数据包打标签的功能。其采用 CIPSO (Common IP Security Option) 标签方法。CIPSO 是系统间协议，包括一系列描述发送数据包进程的安全等级或者内容。CIPSO 用户定义一个 DOI (domain of interpretation)，DOI 其解释这些标签的意义，这样就可以让通信的两端确定对方的进程是否有权限进行通信。DOI 和标签被放在每个 IP 包的可选字段。`Netlabel` 的作用是将 CIPSO 的信息放在发送出去的数据包中，并且检查收到的数据包的标间。其使用 Linux Security Module (LSM) 钩子函数实现加标签和标签检查。`Netlabel` 的管理通过 `netlink` 套接字，也有一些用户空间的配置工具 <http://netlabel.sourceforge.net/>，`netlbl_init` 两个重要的工作一个是使用 `netlbl_domhsh_init` 初始化 DOI，一个是 `netlbl_netlink_init` 创建 `netlink` 套接字初始化。

`sysctl_core_init`: 调用 `_register_sysctl_table` 在 `/proc/sys` 目录下注册 `control table` 的叶子，即 `/proc/sys/net/core` 目录，目录的内容是 `net_core_table` 决定的，而第一个参数 `init_net` 决定了所注册的网络空间是初始网络空间。`core` 目录下的内容是一些有关系统性能的参数，如 `wmem_max`、`rmem_max` 等，根据使用场景和服务器硬件资源的不同，最优参数也不一样，

这些参数可用来系统性能调优，而 `sysctl` 提供了一个方便的方法，使应用程序空间用户能够方便的修改内核一些参数。

```
static __init int sysctl_core_init(void)
{
    register_net_sysctl(&init_net, "net/core", net_core_table);
    return register_pernet_subsys(&sysctl_core_ops);
}
```

`ipv6_offload_init/ipv4_offload_init` 用来注册 UDP 和 TCP 协议下的分段操作，用于将分段操作提交给网卡完成，分段操作推迟执行能够减少网络协议栈上的开销。`TSO` 是针对 `tcp` 协议的，即使没有网卡的支持，推迟分片操作总是能减小系统开销。`scatter/gatherIO`。`GSO` 是 `generic segment offload` 简写，其在 `MTU` (`maximum Transmit unit`) 远远小于 `64K` 时才更加有效。

```
static int __init ipv4_offload_init(void)
{
    if (inet_add_offload(&udp_offload, IPPROTO_UDP) < 0)
        pr_crit("%s: Cannot add UDP protocol offload\n", __func__);
    if (inet_add_offload(&tcp_offload, IPPROTO_TCP) < 0)
        pr_crit("%s: Cannot add TCP protocol offload\n", __func__);

    dev_add_offload(&ip_packet_offload);
    return 0;
}
```

`inet_add_offload` 用于将协议对应的回调函数添加到 `inet_offloads` 数组中，`cmpxchg` 的意义是将第一个和第二个参数比较，如果相等就把第三个参数赋给第一个，如果不相等，返回第一个参数的内容。在更新 `nexthop` 路由缓存项时也会调用 `cmpxchg` 以确定是否需要更新缓存项，这个函数这里的意义就是根据协议号，将回调函数添加到 `net_offloads` 上，如果该协议号上已经有回调函数则什么也不做。

```
int inet_add_offload(const struct net_offload *prot, unsigned char protocol)
{
    return !cmpxchg((const struct net_offload **)&inet_offloads[protocol],
                    NULL, prot) ? 0 : -1;
}
```

`ip_packet_offload` 是 `gso` 方式下的回调函数集，`dev_add_offload` 用于注册 `offload` 处理函数，其实就是将回调函数集添加到内核链表 `offload_base` 上。

```
static struct packet_offload ip_packet_offload __read_mostly = {
    .type = cpu_to_be16(ETH_P_IP),
    .callbacks = {
        .gso_send_check = inet_gso_send_check,
        .gso_segment = inet_gso_segment,
        .gro_receive = inet_gro_receive,
        .gro_complete = inet_gro_complete,
    },
};
```

`tcp_fastopen_init`:这里的 `fast open` 是针对 `TCP` 建立连接的三次握手而言的，其主要特性是在利用握手是的 `SYN` 报文来传输应用数据，这样客户端和服务器之间通信时可以减少一次往返时间。由此可见这是一种非标准的方式，但是其确确实实提高了用户体验，正逐渐流行起来。其详细内容可以参看 `rfc7413` 标准。

`tcp_congestion_default`: 设置默认的 `tcp` 拥塞控制算法，目前 `Linux` 使用的是一个称之为三次方的 `cubic` 算法。

```
static int __init tcp_congestion_default(void)
{
    return tcp_set_default_congestion_control(CONFIG_DEFAULT_TCP_CONG);
```

```
}
```

1.2.2 drivers/net 目录下:

`phy_init` 是初始化链路层芯片用的，`PHY` 就是 `physical` 的简称，其作用有两个，一个是 `mdio` 总线的初始化，`mdio` 总线是 802.3 协议规定的总线，所有 `PHY` 设备必须提供该接口，该接口用于 `PHY` 工作状态的设置，比如速率、双工、`link` 以及自协商等。另外一个就是注册 `PHY` 设备的驱动，由于 `PHY` 在 802.3 中的规定很详细，其最长用的前十五个寄存器 `PHY` 规范已经定好用途了，所以这里注册了一个通用的 `PHY` 驱动，称为 `genphy_driver`，其包括对 `PHY` 设置，状态获取等操作，操作的最终落实是通过 `mdio` 总线完成的。关于 `PHY` 专门有一章，之所以专门设置一章是因为，在 Linux 网络驱动这块，分为两大类一类是协议栈，这类通常关注 OSI 七层模型中的 IP 层级以上，另一类是链路层及以下，就是 `MAC` 层和 `PHY` 层，嵌入式底层驱动中又常常和 `PHY` 或者 `switch` 打交道，所以专门对 `PHY` 的方方面面给出了一章的篇幅。

`inet_init`:这个函数的工作是非常重要的，后面几章关于 TCP/IP 收发的章节内容就依赖于这里 `inet_init` 构建的协议栈基本框架。所以个部分单独作为一个小节来讨论了。

1.3 inet_init

`inet_init` 函数初始化 internet 协议族的协议栈。

net/ipv4/af_inet.c

```
1696 static int __init inet_init(void)
1697 {
1698     struct inet_protosw *q; //ip 协议注册套接字接口之用
1699     struct list_head *r;
1700     int rc = -EINVAL;
1701
//判断 sk_buff 的 cb 成员是否小于 inet_skb_parm 的 size, 如果是则 BUG, cb 的定义是 charcb[48] __aligned(8);该字段被称为控制块，协议栈的每一层都可以使用该字段,在 IP 分片时就会使用该字段。inet_skb_parm 是定义于 include/net/ip.h 文件
//的结构体，该结构体 flags 成员用于标记 packet 的状态，frag_max_size 记录的是数据包的不分片的最大 size，如果该值大
//于 MTU, maximum Transmit Unit, 那么对于发送出去的数据包是会进行分片操作的。
1702     BUILD_BUG_ON(sizeof(struct inet_skb_parm) > FIELD_SIZEOF(struct sk_buff, cb));
1703
//sysctl_local_reserved_ports 是 ip_local_reserved_ports, ip_local_reserved_ports 是在
///proc/sys/net/ipv4/ip_local_reserved_ports, 它是控制表的名字，其对应的内容是 sysctl_local_reserved_ports 的内容，该文件
//作用是用来预留网络端口，就是为使用固定端口的第三方应用程序预留。ip_local_reserved_ports 是针对系统管理员（用户空间
//间），而 sysctl_local_reserved_ports 是针对内核开发人员定义的变量。它们本质上值的是同一个意思，sysctl 的很多接口，它
//们用户空间和内核空间的命名方法和这里的很相似。
1704     sysctl_local_reserved_ports = kzalloc(65536 / 8, GFP_KERNEL);
1705     if (!sysctl_local_reserved_ports)
1706         goto out;
1707
//下面是协议的注册，tcp、udp、raw 和 ping。它们会组成一张和其它部分有着千丝万缕的大大的表。
1708     rc = proto_register(&tcp_prot, 1);
1709     if (rc)
1710         goto out_free_reserved_ports;
1711
1712     rc = proto_register(&udp_prot, 1);
1713     if (rc)
1714         goto out_unregister_tcp_proto;
1715
1716     rc = proto_register(&raw_prot, 1);
```

```

1717     if (rc)
1718         goto out_unregister_udp_proto;
1719
1720     rc = proto_register(&ping_prot, 1);
1721     if (rc)
1722         goto out_unregister_raw_proto;
1723
1724     /*
1725      * Tell SOCKET that we are alive...
1726      */
1727
1728     (void)sock_register(&inet_family_ops);
1729
1730 #ifdef CONFIG_SYSCTL
1731     ip_static_sysctl_init();
1732 #endif
1733
1734     tcp_prot.sysctl_mem = init_net.ipv4.sysctl_tcp_mem;
1735
1736     /*
1737      * Add all the base protocols.
1738      */
1739
1740     if (inet_add_protocol(&icmp_protocol, IPPROTO_ICMP) < 0)
1741         pr_crit("%s: Cannot add ICMP protocol\n", __func__);
1742     if (inet_add_protocol(&udp_protocol, IPPROTO_UDP) < 0)
1743         pr_crit("%s: Cannot add UDP protocol\n", __func__);
1744     if (inet_add_protocol(&tcp_protocol, IPPROTO_TCP) < 0)
1745         pr_crit("%s: Cannot add TCP protocol\n", __func__);
1746 #ifdef CONFIG_IP_MULTICAST
1747     if (inet_add_protocol(&igmp_protocol, IPPROTO_IGMP) < 0)
1748         pr_crit("%s: Cannot add IGMP protocol\n", __func__);
1749 #endif
1750
1751     /* Register the socket-side information for inet_create. */
1752     for (r = &inetsw[0]; r < &inetsw[SOCK_MAX]; ++r)
1753         INIT_LIST_HEAD(r);
1754
1755     for (q = inetsw_array; q < &inetsw_array[INETSW_ARRAY_LEN]; ++q)
1756         inet_register_protosw(q);
1757
1758     /*
1759      * Set the ARP module up
1760      */
1761
1762     arp_init();
1763
1764     /*
1765      * Set the IP module up
1766      */
1767
1768     ip_init();
1769
1770     tcp_v4_init();
1771
1772     /* Setup TCP slab cache for open requests. */
1773     tcp_init();
1774

```

```

1775 /* Setup UDP memory threshold */
1776 udp_init();
1777
1778 /* Add UDP-Lite (RFC 3828) */
1779 udpLite4_register();
1780
1781 ping_init();
1782
1783 /*
1784 * Set the ICMP layer up
1785 */
1786
1787 if (icmp_init() < 0)
1788     panic("Failed to create the ICMP control socket.\n");
1789
1790 /*
1791 * Initialise the multicast router
1792 */
1793 #if defined(CONFIG_IP_MROUTE)
1794 if (ip_mr_init())
1795     pr_crit("%s: Cannot init ipv4 mroute\n", __func__);
1796 #endif
1797 /*
1798 * Initialise per-cpu ipv4 mibs
1799 */
1800
1801 if (init_ipv4_mibs())
1802     pr_crit("%s: Cannot init ipv4 mibs\n", __func__);
1803
1804 ipv4_proc_init();
1805
1806 ipfrag_init();
1807
1808 dev_add_pack(&ip_packet_type);
1809
1810 rc = 0;
1811 out:
1812     return rc;
1813 out_unregister_raw_proto:
1814     proto_unregister(&raw_prot);
1815 out_unregister_udp_proto:
1816     proto_unregister(&udp_prot);
1817 out_unregister_tcp_proto:
1818     proto_unregister(&tcp_prot);
1819 out_free_reserved_ports:
1820     kfree(sysctl_local_reserved_ports);
1821     goto out;
1822 }

```

1728 行用于，注册 inet 协议族。

```

static const struct net_proto_family inet_family_ops = {
    .family = PF_INET,
    .create = inet_create,
    .owner = THIS_MODULE,
};

```

sock_register 获得 net_family_lock 自旋锁在下面局部全局数组 net_families 中添加 PF_INET 对应的成员。

```
static const struct net_proto_family __rcu *net_families[NPROTO] __read_mostly;
```

1731 ip_static_sysctl_init(), 其作用是创建/proc/sys/net/ipv4/route 文件, 从该文件的名称可以知道是和路由相关的, 该文件包含路由参数包括垃圾回收的时间间隔等, 系统管理员可以通过这里对系统性能进行调优。inet_init 函数的初衷是初始化 inet 协议, 但把路由 sysctl 参数接口初始化放在这里显得有点不伦不类, 至少其它的/proc/sys/net/ipv4/下的 sysctl 接口的初始化方法和这里的不一样。

1734 行内核空间称为 sysctl_mem , 而用户空间的对应的是 tcp_mem , 这个变量是个有三个元素的数组, 其三个值是 tcp 占用内存和系统压力关系的描述, 对于 server 而言, 该值相关是需要关注的, mysql 有一个参数就是用于限定 MySQL 连接数的, 其意义就是在限制内存上。

1740-1749 行是基础协议的初始化, 它们都是调用 inet_add_protocol 函数将对应的协议添加到 inet_protos 的 hash 表上。tcp_protocol 结构体, 这里先留个印象, 后面还会在见到的。

```
static const struct net_protocol tcp_protocol = {
    .early_demux = tcp_v4_early_demux,
    .handler = tcp_v4_rcv,
    .err_handler = tcp_v4_err,
    .no_policy = 1,
    .netns_ok = 1,
};
```

1752 行, 为 inet_create 注册套接字侧信息, 在用户空间编程时创建 inet 协议族套接字到内核里就会调用该函数完成实际的创建工作。不同的协议族套接字创建函数不一样。1756 行的函数就会使用到 inetsw 这个链表。可以先看一下图 1.2, 建立一个它们之间组织架构的关系印象。

1762 行, arp 协议初始化。根据 IP 地址获取物理地址的协议, 在使用 ping 命令也许你会注意到一个现象, 有时使用 ping 命令时, ping 的第一次输出延迟是后续延迟的几十倍, 后续再 ping, 第一次延迟和后续的延迟相差无几, 第一次延迟较大是知道 IP 但是不知道 MAC 地址原因, 所以先发送了一个地址解析请求, 获得目标 ip 对应的物理地址之后, 设备才是真正发送 ICMP ping 包。

1768 ip 路由和 peer 子系统初始化, 因为 ip 是无状态连接, 内核为了提升性能, 路由子系统会使用 peer 保存一些信息, peer 子系统使用 avi 树保存这些信息。

1770 tcp_v4_init 初始化 tcp_hashinfo 结构体, 该结构用于根据套接字状态, 其有三个 hash 链表用于该套接字状态管理:

ehash 记录已经建立连接的套接字,
bhash 用于记录正在绑定的套接字,
listening_hash 记录正在侦听的套接字;

它的一个使用实例是 tcp 接收函数 tcp_v4_rcv 会依据 tcp_hashinfo 查找对应的套接字, 查找的那行代码如下:

```
sk = __inet_lookup_skb(&tcp_hashinfo, skb, th->source, th->dest);
```

此外, 还会注册 tcp 下的 skb 的管理结构体 tcp_sk_ops, 主要设置 ipv4.sysctl_tcp_ecn 字段, ecn 是 (explicit congestion notification) 用于拥塞功能, 使用到 IP 首部的 89bit 的 TOS 字段。

1773 行 tcp_init, 为打开的 tcp 请求建立 slab 缓存。这里缓存是上面 tcp_hashinfo 的三个成员 ehash、bhash、listening_hash 建立缓存, 套接字的操作比较频繁, 使用 cache 能够提高效率。初始化 tcp 的 sysctl 一些参数值, 另外使用如下函数:

```
tcp_metrics_init();
tcp_register_congestion_control();
tcp_metrics_init(); 初始化 tcp metrics 接口以及 netlink 接口, tcp metrics 的目标是 tcp 的吞吐量*。tcp metrics 有三个基本的规则[1]:
```

传输时间比: 实际传输时间/理想传输时间, 实际传输时间通过传递数据包的时间戳获得, 理想传输时间由最大 tcp 吞吐量获得。

$$\text{tcp 效率: 效率} = \frac{\text{传输字节数} - \text{重传字节数}}{\text{传输字节数}} * 100$$

缓存延迟:

$$\text{平均 RTT(round-trip time): 平均 RTT} = \frac{\text{传输中的总 RTTs}}{\text{测试的总时间 (秒)}}$$

Baseline RTT: 在网络没有发生拥塞时的固有延迟。

$$\text{缓存延迟} = \frac{\text{平均 RTT} - \text{Baseline RTT}}{\text{Baseline RTT}} * 100$$

tcp_register_congestion_control: 根据函数命名知道这是在注册 tcp 拥塞控制接口, 前面提到过内核现在使用的 tcp 拥塞控制算法“cubic”, 这里注册的使拥塞控制处理函数, 包括:

- `tcp_reno_ssthresh`, 慢启动函数, 设置慢启动阈值, 初始值为拥塞窗口的一半, 但最小值是 2.
- `tcp_reno_cong_avoid` 拥塞控制函数, 包括慢启动和拥塞避免两个部分。
- `tcp_reno_min_cwnd` 拥塞窗长最小化, 实际上是就是对当前套接字对应的慢启动阈值减半。

tcp_tasklet_init: 初始化一个 tasklet, 内核把这个 tasklet 称之为 TSQ(TCP SMALL QUEUES), 其作用是保持每个 cpu 的 tcp 发送队列里的 skb 尽量的少, 以减少 RTT 和 bufferbloat。

udp_init: udp 协议的初始化, 初始化 udp 的控制表的 hash 成员, 其有两个 hash 表, 第一个表用于本地端口套接字的 hash, 第二个表用于本地端口、本地地址套接字的 hash; 此外, 和 tcp 一样初始化了若干 sysctl 控制接口。

udplite4_register: 轻量级用户数据包协议, rfc3828 协议规范。该功能从 2.6.20 开始支持。其将对数据包的校验以及校验推迟到用户决定, 其对于网络不是很好的视频监控应用场景轻量级 udp 比对载荷进行完成校验的 UDP 协议具有一定优势。

1781 行 `ping_init`: 初始化 ping 哈希表。

1787 行 `icmp_init`, 初始化 icmp 协议, 即初始化 struct net 关于 icmp 协议的相关成员, 这些成员包括:

- icmp 套接字成员 `icmp_sk`。
- 发送缓存 `sk_sndbuf` 为 $2 * 64K$, `sk_buff` 和 `skb_shared_info` 大小均为 $64K$ 。
- 标记 `sk_flags` 设置成 `SOCK_USE_WRITE_QUEUE`, 这就意味着 `sock_wfree` 函数会调用 `sk->sk_write_space` 方法来完成。
- 将 `pmtudisc` (路径最大传输单元) 规则设置成不分片。
- 若干 sysctl 接口。

1794 `ip_mr_init` 多播路由初始化。

1801 `init_ipv4_mibs`, 该函数调用参数 `ipv4_mib_ops` 的 init 函数 `ipv4_mib_init_netipv4` 进行 MIB (Management Information Bases) 初始化, 该函数调用的大多数函数都有一个 snmp 前缀, snmp (Simple Network Management Protocol) 源于 IETI (Internet Engineering Task Force) 规范, SNMP 主要用于管理和监控网络设备的状态, 通过这些状态可以知道这些网络设备的状态如 (接口状态、IP 地址、流量等), 双十一时网络流量的冲击导致网卡爆掉的可能性会比平时高些, 就算不是双十一, 网卡也是可能变得有问题的, 如果不监控网络设备的状态, 很可能导致库存中心和各地缓存不一致性, 导致秒杀超卖等。当然电商公司有专门的图像监控管理工具。这些网卡等的信息都存储在 MIB 里, 获得每台服务器的 MIB 信息 (SNMP 协议)。

1804 `ipv4_proc_init` 在 /proc/net 目录下创建若干文件。raw, tcp, udp, icmp, sockstat, netstat 和 snmp 文件, 这些文件统计了网络上的信息, netstat 命令输出的信息通过解析 /proc/net/netstat 文件获得。

1806 `ipfrag_init`, ip 分片操作初始化, 在以太网上 1500 是定义的最大数据包长, 通常不会产生分片, 产生分片多半源于 PMTU (路径最大 MTU), 不能满足该值时, 可能会产生分片操作, 新的技术会不在 ip 层分片, 而将分片推迟到网卡才实际进行, 见十五章。

`ip4 frags_ctl_register` 在 /proc/sys/net/ipv4 目录下创建两个 ip 分片参数, `ip4 frags_ctl_register` 和 `ipfrag_max_dist`; 分片操作的初始化如下:

```

void __init ipfrag_init(void)
{
    ip4 frags_ctl_register();
    register_pernet_subsys(&ip4 frags_ops);
    ip4 frags.hashfn = ip4 hashfn;
    ip4 frags.constructor = ip4 frag_init;
    ip4 frags.destructor = ip4 frag_free;
    ip4 frags(skb_free = NULL;
    ip4 frags.qsize = sizeof(struct ipq);
    ip4 frags.match = ip4 frag_match;
    ip4 frags.frag_expire = ip_expire;
    ip4 frags.secret_interval = 10 * 60 * HZ;
    inet frags_init(&ip4 frags);
}

```

1808: dev_add_pack 注册 Ip 包处理函数到网络协议栈，其参数会被链接到内核列表上。

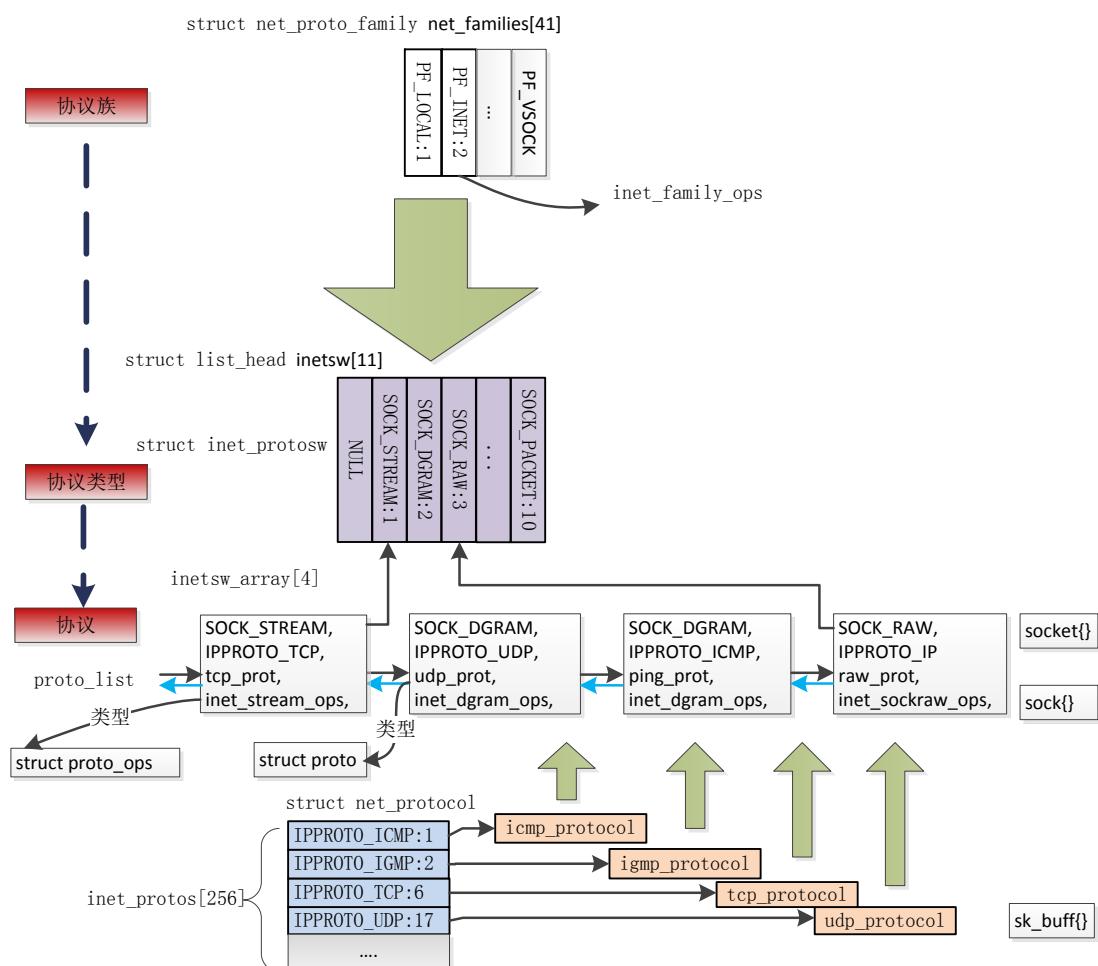


图 1.2 INET 协议栈初始化。

1.4 总结

本章根据 linux 内核下网络相关代码的初始化顺序浏览了各初始化函数的作用和意义，最后将 `inet_init` 函数完成 `inet` 协议族的注册过程进行了细致的分析，并以一张图给出了协议族、协议类型和协议之间的初始化以及它们和套接字的关系，但是这里并没有深入协议的各个字段去看各个字段的作用和意义。这些字段的作用和意义在后面的章节会看到。

第二章 主机到网络层（网卡）

2.1 TCP/IP 协议栈模型

网络协议栈常用 OSI 七层模型，实际上 Linux 网络协议栈使用的却是四层模型，图 2.1 展示了 OSI 七层和四层模型它们之间各层对应关系。图 2.1 的最左侧一列是数据在协议栈上各层的称谓。frame 位于主机到网络层 (Layer1)，packet 位于 Layer2，segment 位于 Layer3，data 或 message 位于 Layer4；当然对于 OSI 七层模型而言，由下至上依次为 Layer1~Layer7，即物理层为 Layer1。

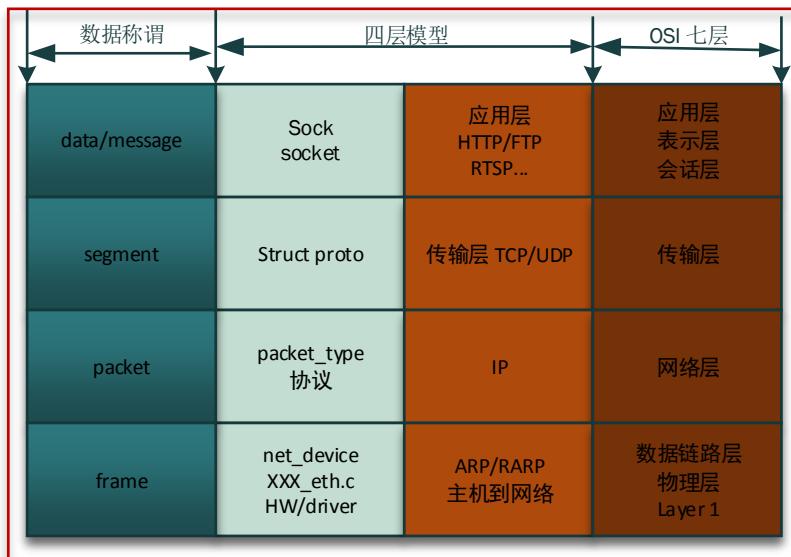


图 2.1 TCP/IP 协议栈的四层和七层模型

Layer1 层对应到 OSI 七层模型是物理层，在四层模型中，这一层其实没有与之对应的层，这里偏重传输层和网络层，即 tcp/ip 协议栈。

2.2 网卡数据结构

和 tcp/ip 协议栈相比主机到网络层和硬件芯片打交道更多，因为这层的主要工作是转交给一个称为网卡(NIC)和PHY的硬件设备完成的，不同网卡性能还有区别，比如速率、offload、校验和、DMA、RSS、SG(scatter gather)I/O 等。内核将网卡应该具有的公共特性抽象成了一个称为 net_device 的结构体，因为各 NIC (单网卡亦或集成与 SOC 的网卡) 它们或多或少有些不同，现实应用中会将 net_device 实例、NIC 特有特性、NIC 特有功能封装到一个结构体内使用，使用 register_netdev 将 net_device 实例注册到内核。该结构体的定义如下：

```
1040 struct net_device{  
1041     char         name[IFNAMSIZ]; //这个是网卡所呈现的接口的名称，如 eth0、eth1 等  
1048     /* device name hash chain, please keep it close to name[] */  
1049     struct hlist_node  name_hlist;  
1051 }
```

```

/*ifconfig 命令配置别名时使用，在单网卡多接口用以区分网段时会使用到，在IPROUTE 工具中，提供了新方法。*/
1053     char          *ifalias;
/*网卡使用的内存、IO 和中断信息*/
1059     unsigned long   mem_end; /* shared mem end */
1060     unsigned long   mem_start; /* shared mem start */
1061     unsigned long   base_addr; /* device I/O address */
1062     unsigned int    irq; /* device IRQ number */

1071     struct list_head dev_list; //设备链表
1072     struct list_head napi_list; //napi 链表
1073     struct list_head unreg_list;
1074     struct list_head upper_dev_list; /* List of upper devices */

/*这些特性指，如分散聚集 IO, UFO, 校验和等等，见 include/linux/netdev_features.h 的 17~71 行。*/
1078     netdev_features_t features;
1079     /* user-changeable features */
1080     netdev_features_t hw_features;
1081     /* user-requested features */
1082     netdev_features_t wanted_features;

1092     /* Interface index. Unique device identifier */
1093     int      ifindex; //接口索引，每个设备对应一个唯一的索引值
1094     int      iflink;
1095

/*该网卡相关的统计信息，如接收、发送的数据包总数相关信息等。netstat -i 或者 ifconfig 看到的关于数据包统计信息源于此处。
这些命令行根据在下篇有提及*/
1096     struct net_device_stats stats;
1097     atomic_long_t    rx_dropped; /* dropped packets by core network
1098                      * Do not use this in drivers.
1099                      */
/*这里的 ops 非常重要，网卡发送接收数据的函数就在这里，不同的设备不同，这也是网络驱动编写者需要完成的工作。*/
1109     const struct net_device_ops *netdev_ops;
1110     const struct ethtool_ops *ethtool_ops; //ethtool 工具的操作方法的集合。
1112     /* Hardware header description */
1113     const struct header_ops *header_ops; //

1114
1115     unsigned int    flags; /* interface flags (a la BSD) */
1116     unsigned int    priv_flags; /* Like 'flags' but invisible to userspace.
1117                      * See if.h for definitions. */
1118     unsigned short  gflags;
1119     unsigned short  padded; /* How much padding added by alloc_netdev() */

1120
1121     unsigned char   operstate; /* RFC2863 operstate */
1122     unsigned char   link_mode; /* mapping policy to operstate */

1123
1124     unsigned char   if_port; /* Selectable AUI, TP,... */
1125     unsigned char   dma; /* DMA channel */

1126
1127     unsigned int    mtu; /* interface MTU value 其会影响到分片操作 */
1128     unsigned short  type; /* interface hardware type 见 if_arp.h 文件*/
1129     unsigned short  hard_header_len; /* hardware hdr length */

1138     /* Interface address info.*/
1139     unsigned char   perm_addr[MAX_ADDR_LEN]; /* permanent hw address */
1140     unsigned char   addr_assign_type; /* hw address assignment type */
1141     unsigned char   addr_len; /* hardware address length */
1142     unsigned char   neigh_priv_len;

```

```

1143     unsigned short      dev_id;      /* for shared network cards */

//混杂模式标志，即数据包目的地址非本机也会被收集到，wireshark、tcpdump 会将网卡置于该工作模式
1156     unsigned int      promiscuity;
1157     unsigned int      allmulti;
1169     struct in_device __rcu *ip_ptr; /* IPv4 specific data */

1176 /*
1177  * Cache lines mostly used on receive path (including eth_type_trans())
1178 */
1179     unsigned long      last_rx;    /* Time of last Rx
1180                         * This should not be set in
1181                         * drivers, unless really needed,
1182                         * because network stack (bonding)
1183                         * use it if/when necessary, to
1184                         * avoid dirtying this cache line.
1185
1186
1187 /* Interface address info used in eth_type_trans() */
1188     unsigned char      *dev_addr;   /* hw address, (before bcast    MAC 地址
1189                               because most packets are
1190                               unicast) */

/*发送队列的记录信息*/
1214     struct netdev_queue *_tx ____cacheline_aligned_in_smp;
1215
1216 /* Number of TX queues allocated at alloc_netdev_mq() time */
1217     unsigned int      num_tx_queues;
1218
1219 /* Number of TX queues currently active in device */
1220     unsigned int      real_num_tx_queues;
1221
1222 /* root qdisc from userspace point of view */
1223     struct Qdisc      *qdisc; //queue discipline, 会影响数据包的收发。
1224
1225     unsigned long      tx_queue_len; /* Max frames per queue allowed */
1226     spinlock_t        tx_global_lock;

/*register_netdevice 注册该网卡，ifconfig up/down 等，会修改网卡的设备的状态，所有状态均在此。*/
1259 /* register/unregister state machine */
1260     enum { NETREG_UNINITIALIZED,
1261             NETREG_REGISTERED, /* completed register_netdevice */
1262             NETREG_UNREGISTERING, /* called unregister_netdevice */
1263             NETREG_UNREGISTERED, /* completed unregister todo */
1264             NETREG_RELEASED, /* called free_netdev */
1265             NETREG_DUMMY, /* dummy device for NAPI poll */
1266 } reg_state:8; 1267
1268     bool dismantle; /* device is going do be freed */
1269
1270     enum {
1271         RTNL_LINK_INITIALIZED,
1272         RTNL_LINK_INITIALIZING,
1273 } rtnl_link_state:16; //rtnl, 是 route Netlink 的缩写，描述其状态。
1274
1282 #ifdef CONFIG_NET_NS
1283 /* Network namespace this network device is inside */
1284     struct net      *nd_net; //网络命名空间,container 机制会用到
1285 #endif
1286

```

```

1287 /* mid-layer private */
1288 union {
1289     void             *ml_priv;
1300
/*注意这里的__percpu，这就意味着这些成员再 SMP 情况下，每个 CPU 核都有一个副本以提高效率*/
1290     struct pcpu_lstats __percpu *lstats; /* loopback stats */
1291     struct pcpu_tstats __percpu *tstats; /* tunnel stats */
1292     struct pcpu_dstats __percpu *dstats; /* dummy stats */
1293     struct pcpu_vstats __percpu *vstats; /* veth stats */
1294 };
1300
1301 /* class/net/name entry */
1302 struct device      dev; // 通用的设备模型结构体，该结构体描述的是设备都需要的成员
1303
1304 /* rtnealink link ops */
1305 const struct rtnl_link_ops *rtnl_link_ops; //ip 和 tc 工具的内核支持机制 Netlink 的操作集
1306
1307
1308 /* for setting kernel sock attribute on TCP connection setup */
1309
/*GSO 是 generic segment offload，是 TSO (TCP segment offload) 的升级，segment 数据原本在 TCP 层，但是为了效率，现在网卡自己支持分片操作，所以有些情况下会将分片操作推迟到网卡去完成。这些网络多队列等新特性在下部中会提及*/
1310 #define GSO_MAX_SIZE      65536
1311 unsigned int        gso_max_size;
1312 #define GSO_MAX_SEGS      65535
1313 u16                 gso_max_segs;
1314
/*拥塞控制相关*/
1315 u8 num_tc;
1316 struct netdev_tc_txq tc_to_txq[TC_MAX_QUEUE];
1317 u8 prio_tc_map[TC_BITMASK + 1];
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329 /* phy device may attach itself for hardware timestamping */
//PHY 内容，参看十六章。
1330 struct phy_device *phydev;
1331
1332 };

```

该数据结构占 298 行，是 Linux 内核中比较大的数据结构了，该结构主要就是描述网卡的状态、能力、操作集、用户空间的接口支持等。网卡的函数操作集如下：

```

906 struct net_device_ops {
//当网卡注册时仅会调用一次，该函数用于进一步的完成设备特定的初始化工作。
907     int      (*ndo_init)(struct net_device *dev);
908     void      (*ndo_uninit)(struct net_device *dev);           //网卡注销或者注册失败时调用。
//打开网卡时会调用，ifconfig, ip 等命令会触发，当将网卡转到 up 时调用。
909     int      (*ndo_open)(struct net_device *dev);
910     int      (*ndo_stop)(struct net_device *dev);            //当将网卡转到 down 时调用。
911     netdev_tx_t  (*ndo_start_xmit)(struct sk_buff *skb,
912                                     struct net_device *dev);          //图 2.1 中的 frame 就是通过该函数发送到 RJ45 线上（实际上是配置网卡内部发送寄存器，启动硬件发送）
913     int      (*ndo_set_mac_address)(struct net_device *dev,
914                                     void *addr);
915     int      (*ndo_validate_addr)(struct net_device *dev);
916     int      (*ndo_do_ioctl)(struct net_device *dev,
917                             struct ifreq *ifr, int cmd);       //ifconfig 使用的 ioctl 方法，会调用该接口将命令发送到网
卡，但是 ip 和 tc 工具使用 Netlink 方法和此不同。
918     int      (*ndo_set_config)(struct net_device *dev,
919                             struct ifmap *map);

```

```
...
}

header_ops 用于协议的头部信息处理，处理方式如下：
```

```
265 struct header_ops {
266     int (*create)(struct sk_buff *skb, struct net_device *dev,
267                   unsigned short type, const void *daddr,
268                   const void *saddr, unsigned int len);           //创建一个协议头
269     int (*parse)(const struct sk_buff *skb, unsigned char *haddr); //获得 packet 包对应的硬件地址，集 MAC 地址。
270     int (*rebuild)(struct sk_buff *skb);
271     int (*cache)(const struct neighbour *neigh, struct hh_cache *hh, __be16 type);
272     void (*cache_update)(struct hh_cache *hh,
273                          const struct net_device *dev,
274                          const unsigned char *haddr);
275};
```

2.3 网卡注册流程

上面的网卡数据结构在网卡驱动架构中是处于核心的地位，在注册网卡时就使用到了，TCP/IP 网络协议栈的 Layer1 还是比较重要的，毕竟所有的数据均通过这里，下面是网卡的注册流程，绝大多数情况下网卡的注册实际上就是 MAC 控制器的注册，通常会外接 PHY 芯片，补充一下这里注册实例：

- 1、这里给的例子基于 SOC 芯片上的 MAC 控制器注册。其不挂载在 PCI 总线上。
- 2、通常意义上常说的网络驱动程序编写也指的是下面所写的内容，由于是设备驱动代码了，所以代码不再 net 目录下，而在 drivers 目录下了。

drivers/net/ethernet/目录下的代码，该目录下每个厂商会对应一个子目录。

网卡注册流程：

```
static struct platform_driver XXYY_driver = {
    .probe  = XXYY_drv_probe,
    .remove = XXYY_drv_remove,
    .driver = {
        .name = "XXYY-eth",
        .owner = THIS_MODULE,
        .of_match_table= XXYY_eth_dt_ids,
    },
};

module_platform_driver(XXYY_driver);
```

module_platform_driver 在注册设备驱动时，会调用回调函数 probe 完成特定硬件设备需要的一些工作，代码框架如下：

```
static int XXYY_drv_probe(struct platform_device *pdev)
{
/* device_node 设备树相关*/
    struct device_node *np = pdev->dev.of_node;
    struct net_device *ndev;
    struct XXYY_info *ip;
    struct resource *res;
    const char *macaddr;
    int ret_val = 0;
/*XXYY_info 是对 net_device 结构体的封装*/
    ndev = alloc_etherdev(sizeof(struct XXYY_info)); //
```

```

lp = netdev_priv(ndev);
res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
lp->regbase = devm_ioremap(&pdev->dev, res->start, resource_size(res));
ndev->irq = platform_get_irq(pdev, 0); //如果中断使能，那么在接收和发送数据包时会用到该中断号。
lp->ndev = ndev;
lp->mii_bus.read = &XXYY_mdio_read, //PHY 设备读写，参考《PHY Linux 驱动》
lp->mii_bus.write,
lp->mii_bus.reset = &XXYY_mdio_reset,
ret_val = of_mdiobus_register(&lp->new_bus, pdev->dev.of_node);
lp->phydev = phy_find_first(&lp->new_bus);
ether_setup(ndev); //该函数用于初始化以太网设备通用的字段，见后面讲述。
//这里再一次看见了 struct net_device_ops 结构体，这是需要驱动程序编写者根据芯片手册完成的。
ndev->netdev_ops = &ambeth_netdev_ops;
netif_napi_add(ndev, &lp->napi, XXYY_napi, XXYY_NAPI_WEIGHT); //NAPI 主机到网络层再来看这里的含义。
ret_val = register_netdev(ndev); //前文所述的将网络设备注册到 Linux 核心的函数。
platform_set_drvdata(pdev, ndev); //platform 总线上的信息记录
return 0;
}

```

ether_setup(ndev)，该函数用于初始化以太网设备通用的字段，函数的如下：

```

void ether_setup(struct net_device *dev)
{
    dev->header_ops= &eth_header_ops;
    dev->type  = ARPHRD_ETHER;
    dev->hard_header_len = ETH_HLEN;
    dev->mtu  = ETH_DATA_LEN;
    dev->addr_len  = ETH_ALEN;
    dev->tx_queue_len= 1000;
    /* Ethernet wants good queues */
    dev->flags  = IFF_BROADCAST|IFF_MULTICAST;
    dev->priv_flags|= IFF_TX_SKB_SHARING;
    memset(dev->broadcast, 0xFF, ETH_ALEN);
}

```

函数的操作集如下：

```

static const struct net_device_ops XXYY _netdev_ops = {
    .ndo_open  = XXYY_open,
    .ndo_stop  = XXYY_stop,
    .ndo_start_xmit= XXYY_hard_start_xmit,
    .ndo_set_rx_mode= XXYY_set_multicast_list,
    .ndo_set_mac_address  = XXYY_set_mac_address,
    .ndo_validate_addr= eth_validate_addr,
    .ndo_do_ioctl  = XXYY_ioctl,
    .ndo_change_mtu= eth_change_mtu,
    .ndo_tx_timeout= XXYY_timeout,
    .ndo_get_stats= XXYY_get_stats,
};

```

netif_napi_add 添加一个 napi 服务函数。

```

void netif_napi_add(struct net_device *dev, struct napi_struct *napi,
    int (*poll)(struct napi_struct *, int), int weight)
{
    INIT_LIST_HEAD(&napi->poll_list);
    napi->gro_count = 0;
    napi->gro_list = NULL;
    napi->skb = NULL;
    napi->poll = poll;
    napi->weight = weight;
    list_add(&napi->dev_list, &dev->napi_list);
}

```

```

napi->dev = dev;
set_bit(NAPI_STATE_SCHED, &napi->state);
}

```

注册完毕后，PHY 设备通过自协商选择合适的网络速率，MAC 层这时也注册到 Linux 核心了，网卡和数据以及 PHY 的关联，见图 1.2。

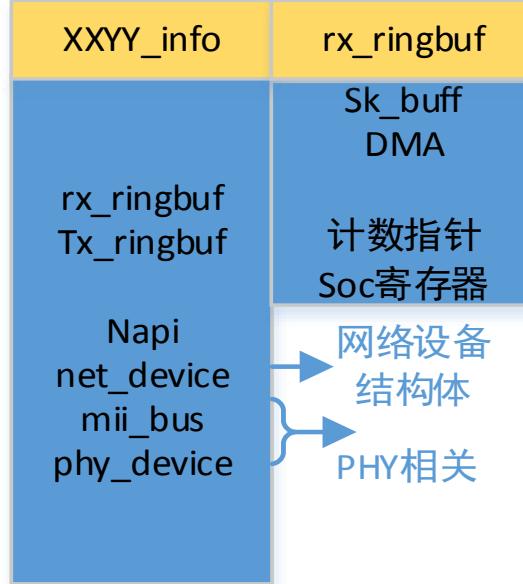


图 1.2 网卡结构体关键字段组织

该图中的 sk_buff，在 Linux 内核代码中通常将 sk_buff 注释成 SKB，后面沿用此注释法。

SKB 存放数据，发送和接收维护各自的队列，虽然通常一个网卡的接收和发送使用同一套 DAM 控制器，但是其接收和发送数据包的缓存是区分开的。XXYY_netdev_ops 结构体中并没有接收数据包的方法，其实由于数据包到达时刻的不确定性而采用了中断的方式接收数据，为了节约中断资源，在 XXYY_open 中才注册中断服务函数，通常其它类型的设备驱动程序都是这么做的，在 close 设备时，将中断号释放掉，以便其它设备可以使用该中断号。open 函数的原型如下：

```

static int XXYY_open(struct net_device *ndev)
{
    int ret_val = 0;
    struct XXYY_info *lp;

    lp = (struct XXYY_info *)netdev_priv(ndev);
    ret_val = XXYY_start_hw(ndev); //特定于 SOC 的函数，配置相关寄存器使能该网卡设备
    ret_val = request_irq(ndev->irq, XXYY_interrupt,
    //注册中断服务函数，发送和接收共享该中断，服务函数中判断是接收还是发送中断。
    IRQF_SHARED | IRQF_TRIGGER_HIGH, ndev->name, ndev);
    napi_enable(&lp->napi); //NAPI 接口
    netif_start_queue(ndev); //发送使能
    netif_carrier_off(ndev);
    ret_val = XXYY_phy_start(lp); //复位一下 PHY 设备
    return ret_val;
}

```

接收中断调度 NAPI，发送中断则执行发送端代码，napi 主体思想如下：

- 常规流程在接收数据包时，一个数据包到达网卡时，网卡产生中断，通知 CPU 数

据到来，CPU 响应该次接收，当下一个数据包到来时，再次重复上述过程，这个过程的特点是一个中断对应一个数据包，为了省去中断带来的资源开销，能不能一个中断多收几个数据包呢？轮询就可以实现一个中断接收多个数据包，流程变为当数据包到来产生中断时，CPU 关中断，不停的轮询网卡是否有新数据到来，接收数据包个数的上限值（和网卡接收缓存大小相关，常取 buffer 的 2/3~1/3 之间）或者超时可以作为退出条件。

- 发送也是这么概念，一次发送缓冲区大小的 1/3~2/3，然后再使能发送中断，网卡发送完数据后会产生发送中断。

```
static inline void XXYY_interrupt_rx(struct XXYY_info *lp, u32 irq_status)
{
    napi_schedule(&lp->napi);
}
```

napi 所完成的工作就是实质将数据向上层发送

```
static inline void XXYY_napi_rx(struct ambeth_info *lp, u32 status, u32 entry)
{
    struct sk_buff *skb;
    skb = lp->rx.rng_rx[entry].skb;
    netif_receive_skb(skb); //该函数将数据从主机到网络发送到网络层
}
```

netif_receive_skb(skb); 该函数将数据从主机到网络层发送到网络层，即 IP 层；其调用 __netif_receive_skb 将 NAPI (napi_struct) 方法添加到轮询表，并设置软中断 NET_RX_SOFTIRQ。软中断服务函数 net_rx_action 将会执行 poll 函数，关闭中断并轮询网卡，超时或者接收数据包数量超限时会退出。

IP 层发送数据到主机到网络层使用的接口函数是：

int dev_queue_xmit(struct sk_buff *skb)，还会经过流控环节，路由也会咨询一些流控信息。这个接口是 IP 和主机到网络层的接口，后面还会遇到。和 IP 层连通的函数是 ip_recv() 和 ip_out()，下面以一张图来结束本章的内容。

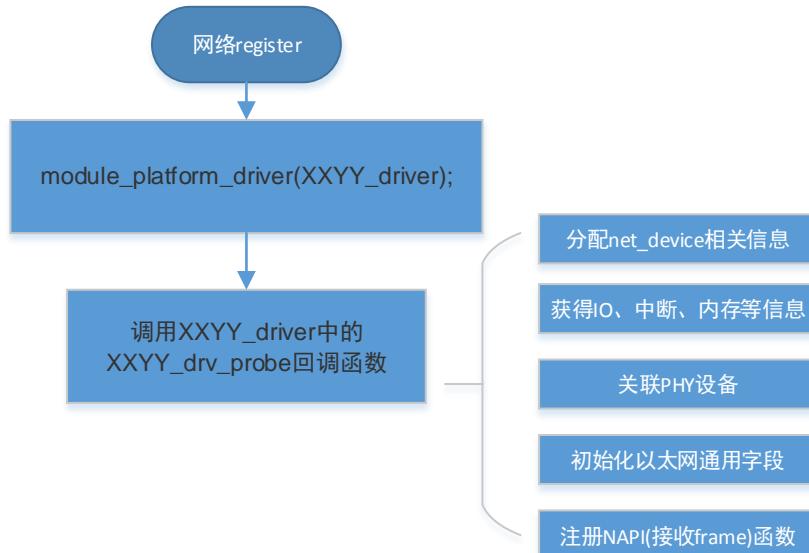


图 1.3 网卡注册流程

第三章 套接字相关数据结构

本章是对 socket 通信过程中使用到的比较重要的数据结构罗列和意义的阐述，在阅读其它层的代码前，先来看几个重要的数据结构，这几个数据结构贯穿四层模型。

3.1 socket 对应的内核结构体

在用户空间使用 socket() 函数创建一个套接字。对应的系统调用就是：

asmlinkage long sys_socketcall(int call, unsigned long __user *args); 该系统调用的定义在 net/socket.c 文件的 2436 行，调用流程中比较重要的一个函数是：

```
int __sock_create(struct net *net, int family, int type, int protocol,
struct socket **res, int kern)
{
    struct socket *sock;
    const struct net_proto_family *pf;

    sock = sock_alloc(); //内存空间分配
    //根据对应的协议族 (protocol family) 创建对应的 sock。
    err = pf->create(net, sock, protocol, kern);
    if (err < 0)
        goto out_module_put;
}
```

该函数首先创建一个 struct socket 的类型结构体，该结构体对应于用户空间的 socket，socket 的参数之一是协议族，对于 Internet 协议，create 的函数原型是 inet_create，internet 对应协议族在内核中的表示如下：

```
static const struct net_proto_family inet_family_ops = {
    .family = PF_INET,
    .create = inet_create,
    .owner = THIS_MODULE,
};
```

这里的 inet_create 函数作用是创建一个 inet 协议族下的套接字，并且初始化其中的一些成员。

该套接字传递到内核后，内核会创建 struct socket 存储来该数据结构：

```
struct socket {
    socket_state state;          //标记 sock 状态，如 SS_CONNECTED、SS_CONNECTING 等
    short type;                  //socket 类型 SOCK_STREAM、SOCK_DGRAM、SOCK_RAW 等。
    unsigned long flags;          //socket flag 如 SOCK_ASYNC_NOSPACE
    struct socket_wq __rcu *wq;
    struct file *file;            //垃圾回收的文件指针
    struct sock *sk;              //因特网内部协议的 socket 表示，对于 PF_INET 协议，inet_create 会创建该成员的各个字段。
    const struct proto_ops *ops;   //协议族相关的操作函数集
};
```

在应用层 socket 表示套接字，在网络层（IP 层）struct sock 对应应用层中的套接字。

```
<include/net/sock.h>
struct sock {
    socket_lock_t sk_lock; //该 sock 的访问锁
    struct sk_buff_head *sk_receive_queue; //接收到的数据包都放在这个 sk_buff_head 所指向的队列的头上。
```

```

struct {
    atomic_t rmem_alloc;
    int len;
    struct sk_buff *head;
    struct sk_buff *tail;
} sk_backlog; //对于接收的 frame，其由 IP 层存放在 backlog 上，后通过 tcp 的函数进行接收。
#define sk_rmem_alloc sk_backlog.rmem_alloc

int sk_forward_alloc; //对于到达的 frame 非本机，允许 forward 将会被发送出去
#ifndef CONFIG_RPS //网卡新特性，下篇涉及
__u32 sk_rxhash;
#endif

atomic_t sk_drops; //丢弃的 sock 计数器
int sk_rcvbuf;

#ifndef CONFIG_XFRM
struct xfrm_policy *sk_policy[2]; //流控策略，属于安全机制
#endif

unsigned long
sk_flags;
struct dst_entry *sk_rx_dst; //接收流向的
struct dst_entry __rcu *sk_dst_cache; //路由项的 cache
spinlock_t sk_dst_lock; //路由锁
int sk_sndbuf;
struct sk_buff_head sk_write_queue; //发送队列

/*sock 信息、状态的一些标志*/
unsigned int sk_shutdown : 2,
sk_no_check : 2,
sk_userlocks : 4,
sk_protocol : 8,
sk_type : 16;
gfp_t sk_allocation; //sock 动态获申请内存的 Flag 标志。
/*网卡的一些信息也记录到这里了*/
netdev_features_tsk_route_caps;
netdev_features_tsk_route_nocaps;
int sk_gso_type;
unsigned int sk_gso_max_size;
u16 sk_gso_max_segs;

/*sock 的一些错误统计信息在此处*/
struct sk_buff_head sk_error_queue;
struct proto *sk_prot_creator;
rwlock_t sk_callback_lock;
int sk_err,
sk_err_soft;
unsigned short sk_ack_backlog;
unsigned short sk_max_ack_backlog;
__u32 sk_priority;

/*接收和发送的时间戳*/
long sk_rcvtimeo;
long sk_sndtimeo;
void *sk_protinfo;
struct timer_listsk_timer;
ktime_t sk_stamp;
struct socket *sk_socket;

/*分片信息*/
struct page fragsk_frag;
struct sk_buff *sk_send_head; /*分片头信息*/

/*sock 自带的一些函数指针集*/
void (*sk_state_change)(struct sock *sk);

```

```

void (*sk_data_ready)(struct sock *sk, int bytes);
void (*sk_write_space)(struct sock *sk);
void (*sk_error_report)(struct sock *sk);
int (*sk_backlog_rcv)(struct sock *sk, struct sk_buff *skb);
void (*sk_destruct)(struct sock *sk);
};


```

3.2 struct proto_ops

```

<net/ipv4/af_inet.c>
const struct proto_ops inet_stream_ops = {
    .family     = PF_INET,//协议族
    .owner      = THIS_MODULE,
    .release    = inet_release,
    .bind       = inet_bind,
    .connect    = inet_stream_connect,
    .socketpair = sock_no_socketpair,
    .accept     = inet_accept,
    .getname    = inet_getname,
    .poll       = tcp_poll,
    .ioctl      = inet_ioctl,
    .listen     = inet_listen,
    .shutdown   = inet_shutdown,
    .setsockopt = sock_common_setsockopt,
    .getsockopt = sock_common_getsockopt,
    .sendmsg    = inet_sendmsg,
    .recvmsg    = inet_recvmsg,
};


```

上面函数的指针用户空间时常会用到，`sendmsg` 和 `recvmsg` 是介于应用层和传输层之间的收发函数。

3.3 struct proto

```

<net/ipv4/tcp_ipv4.c>
struct proto tcp_prot = {
    .name = "TCP",
    .owner = THIS_MODULE,
    .close = tcp_close,
    .connect = tcp_v4_connect, //建立连接使用到的函数，对应于用户空间的 connect 函数
    .disconnect = tcp_disconnect,
    .accept = inet_csk_accept,
    .ioctl = tcp_ioctl,
    .init = tcp_v4_init_sock,
    .destroy = tcp_v4_destroy_sock,
    .shutdown = tcp_shutdown,
    .setsockopt = tcp_setsockopt,
};


```

该结构体描述的是 `tcp` 处理各种任务的若干函数，这些任务包括 `tcp` 链接的建立、控制等，这些数据都是由 `sk_buff_head`（用于描述套接字缓存区头）结构体管理，数据本身会存

在 2.4 节描述的 sk_buff 里。这个 buffer 使用通过 proc 的 slabinfo 可以看到，曾在一个嵌入式视频监控设备上就遇到过由于 WiFi 导致的 sk_buff_head 和 sk_buff 不定时异常增大的情况。

```
struct sk_buff_head {  
    /* These two members must be first. */  
    struct sk_buff *next; //下一个数据存放指针  
    struct sk_buff *prev; //前一个数据存放指针, next 和 prev 会串接成一个双链表, qlen 用于标记双链表的长度  
    __u32 qlen; //标记  
    spinlock_t lock; //保护该结构体的锁  
};
```

3. 4 sk_buff (SKB)

SKB 存储了用户要求传递的数据，这些数据可能源于视频、图像、文本等，应用层传递到 TCP/IP 协议栈的数据会保存在 sk_buff，不论是 http 还是 rtsp，数据会一直存在 sk_buff 的结构成员中直到从网卡发送出去，接收也是类似的。网络数据包收发如此频繁，可以想象该结构体必然针对协议实现特点、处理流程以及内存等方面做了一些优化。

```
<include/linux/skbuff.h>  
struct sk_buff {  
    /* These two members must be first. */  
    struct sk_buff *next; //指向该 SKB 的后一个 SKB, 其头就是上面 sk_buff_head 指定的成员。  
    struct sk_buff *prev; //指向该 SKB 的前一个 SKB  
    ktime_t tstamp; //数据包到达的时间戳  
    struct sock *sk; //对应的 sock 成员，即应用程序的 socket 在内核的代表，  
    struct net_device *dev; //网络设备，数据到达的网络设备或者数据离开的网络设备  
    //control buffer, 协议栈很多地方都使用到了这个字段来存储一些会使用到的信息。  
    char cb[48] __aligned(8);  
    unsigned long _skb_refdst; //目的入口项  
#ifdef CONFIG_XFRM  
    struct sec_path *sp; //xfrm 安全机制使用, Security path.  
#endif  
    unsigned int len, //数据实际长度值  
    data_len; //数据的长度，和真实长度的区别在于可能有 padding  
    __u16 mac_len, //MAC 的长度  
    hdr_len; //拷贝 skb 时，可更改的头长度  
    union {  
        __wsum csum; //校验和  
        struct {  
            __u16 csum_start; //校验和计算起始地址  
            __u16 csum_offset; //从 csum_start 开始的校验，这部分校验和会被存储。  
        };  
    };  
    __u32 priority; //packet 排队的优先级  
    kmemcheck_bitfield_begin(flags1);  
    __u8 local_df:1; //允许本地分片的标志  
    cloned:1; //标记头是否可能拷贝，如果不对数据执行更改操作，则只会拷贝头。  
    ip_summed:2; //驱动程序填写的 IP 层校验和标志。  
    nohdr:1; //负载使用  
    nfctinfo:3; //SKB 和 tcp 连接的关系  
    __u8 pkt_type:3; //packet 所属的类  
    fclone:2; //复制状态标志，标识该 SKB 是复制的。  
    ipvs_property:1; //该 SKB 为 ipvs 所有。IP virtual Server, 负载均衡, netfilter 框架调用  
    peeked:1; //标志标识统计信息是否还要更新
```

```

nf_trace:1;//netfilter 包跟踪标志
kmemcheck_bitfield_end(flags1);
__be16 protocol; //packet 所属的协议
void (*destructor)(struct sk_buff *skb); //解析函数
int skb_if; //该 packet 所在设备的接口索引
__u32 rxhash; //接收数据包的哈希标志
__u16 queue_mapping; //支持多队列网卡设备的队列映射
kmemcheck_bitfield_begin(flags2);
__u8 pfmemalloc:1;
__u8 ooo_okay:1;
__u8 l4_rxhash:1;
__u8 wifi_acked_valid:1;
__u8 wifi_acked:1;
__u8 no_fcs:1;
__u8 head_frag:1;
sk_buff_data_tinner_transport_header; //MAC 头、IP 头、tcp 头。前三个是指封装过的。
sk_buff_data_tinner_network_header;
sk_buff_data_tinner_mac_header;
sk_buff_data_transport_header;
sk_buff_data_tnetwork_header;
sk_buff_data_tmac_header;
/* These elements must be at the end, see alloc_skb() for details. */
sk_buff_data_ttail;
sk_buff_data_tend; //数据的相关指针
unsigned char *head, *data;
unsigned int truesize;
atomic_t users;
};


```

3.5 softnet_data

softnet_data 是一个 per-CPU 变量，即每个 CPU 都有一个自己的 softnet_data 结构体，相比只有一个该结构体由多个 CPU 共享的变量，每个 CPU 都有一个队列可以减少锁操作。该结构体管理接收和发送的数据。定义于 include/linux/netdevice.h 文件。

```

struct softnet_data {
    struct Qdisc *output_queue; //有数据包要发送的设备。
    struct Qdisc **output_queue_tailp; //上述结构体的待处理的最后一个元素的指针。
    struct list_head poll_list;
    struct sk_buff *completion_queue; //已经成功发送，占用的空间可以释放了。
    struct sk_buff_head process_queue;
/* stats */
    unsigned int processed; //每一个处理该数据包的进程会将这里的计数器加 1，以标记有多少个进程在其 sk_buff。
    unsigned int time_squeeze;
    unsigned int cpu_collision;
    unsigned int received_rps;
#ifdef CONFIG_RPS //网卡多队列，Receive Packet Steering，网卡的硬件特性。
    struct softnet_data*rps_ipi_list;
/* Elements below can be accessed between CPUs for RPS */
    struct call_single_data* __cacheline_aligned_in_smp;
    struct softnet_data*rps_ipi_next;
    unsigned int cpu;
    unsigned int input_queue_head;
    unsigned int input_queue_tail;
#endif
    unsigned int dropped;
};


```

```
struct sk_buff_headininput_pkt_queue; //在 net_dev_initz 中初始化，在网卡驱动程序处理以前，sk_buff 链接到该链表上。
struct napi_structbacklog; //NAPI 处理最开始的那两个元素。
};
```

3.6 struct packet_type

```
struct packet_type {
    __be16 type; /* This is really htons(ether_type). 标记类型，对于 IP 而言是 cpu_to_be16(ETH_P_IP),*/
    struct net_device*dev; /* NULL is wildcarded here */
//该函数是四层网络模型中的网络层的函数，对于 ipv4 是 ip_rcv，这是在之三文章中 tcp/ip 协议栈的网络层从网络到主机层接收数据包的函数。
    int (*func)(struct sk_buff *, struct net_device *, struct packet_type *, struct net_device *);
    bool (*id_match)(struct packet_type *ptype, struct sock *sk);
    void *af_packet_priv;
    struct list_headlist;
};
```

netif_receive_skb 函数会从 ptype_base 协议链表上查找和数据包的 type 对应的 func 处理程序，对于 IP 数据包，其类型是 ETH_P_IP，服务函数是 ip_rcv。此外还有和 TCP/IP 相关的一些重要数据结构。

3.7 一些名词简称

```
csk ---connection sock
icsk--- inet connection sock
ca --congestion avoid
cwr congestion window reduction (cwnd reduction)
ECN: Explicit Congestion Notification
SACK:selective ACK
PSH (1 bit) – Push function. Asks to push the buffered data to the receiving application
TIME-WAIT :
(either server or client) represents waiting for enough time to pass to be sure the remote TCP received the acknowledgment of its connection termination request.
```

第四章 网络层接收数据包流程

4.1 主机到网络层的过渡

从 `netif_receive_skb(struct sk_buff *skb)` 函数开始，网卡收到数据包后产生中断通知 CPU 有数据到达，在中断服务函数中触发接收软中断，等待内核在适当的时间调度 NAPI 方式的接收函数完成数据的接收，并非所有网卡或者 MAC 控制器都是支持 NAPI 方法（需要硬件能支持）的，NAPI 服务函数最重要的工作就是调用 `netif_receive_skb` 将数据从主机到网络层送到网络层，其函数参数在第一章叙述过，收到的数据最终能不能送到应用层，是和拥塞控制、路由、协议层如何处理该数据包相关的，由于该函数是在软中断中调用的，所以该函数执行时硬件中断是开启的，这就意味着可能前一次 MAC 接收到的数据还没有传递到网络层时，MAC 又接收到数据又产生新中断，而新的数据需要存放在一个通常被称为 DMA 缓冲区的地方，这也意味着要支持 NAPI 方式就需要多个缓冲区，这也是硬件为支持 NAPI 方式必须支持的一个特性。

```
int netif_receive_skb(struct sk_buff *skb)
{
//接收到的数据包的时间戳，netdev_tstamp_pqueue 在 dev.c 文件里被初始化成了 1，作为在 SKB 加入队列前是否打上时间
//戳的标志
    net_timestamp_check(netdev_tstamp_pqueue, skb);
    if (skb_defer_rx_timestamp(skb))
        return NET_RX_SUCCESS;
//RPS 是 Receive Packet Steering 的缩写，也许你还看到过 GSO, TSO, RFS 等等，这些特性后面专门有个讲，这里就略过了。
#define CONFIG_RPS
    if (static_key_false(&rps_needed)) {
        struct rps_dev_flow voidflow, *rflow = &voidflow;
        int cpu, ret;
        rcu_read_lock();
        cpu = get_rps_cpu(skb->dev, skb, &rflow); //RPS 见十五章
        if (cpu >= 0) {
            ret = enqueue_to_backlog(skb, cpu, &rflow->last_qtail);
            rcu_read_unlock();
            return ret;
        }
        rcu_read_unlock();
    }
#endif
    return __netif_receive_skb(skb);
}
```

经过几层调用后 `__netif_receive_skb` 会调用 `__netif_receive_skb_core` 函数。

```
3419 static int __netif_receive_skb_core(struct sk_buff *skb, bool pfmemalloc)
3420 {
3421     struct packet_type *ptype, *pt_prev;// packet_type 在第一章就见过了
3422     rx_handler_func_t *rx_handler;
3423     struct net_device *orig_dev;
3424     struct net_device *null_or_dev;
3425     bool deliver_exact = false;
3426     int ret = NET_RX_DROP;
3448 another_round:
3449     skb->skb_ifindex = skb->dev->ifindex; //这是在 sk_buff 中提到的接口索引
3450
3451     __this_cpu_inc(softnet_data.processed); //这也是第二章中提到的 per-CPU 变量，增加正在处理标志计数器。
```

```
//这里是sniffer嗅探器相关的代码，ptype_all包括了所以协议类型，通过wireshark或者tcpdump工具将网卡至于混杂模式下
//抓取数据，会执行这段代码
3470     list_for_each_entry_rcu(ptype, &ptype_all, list) {
3471         if (!ptype->dev || ptype->dev == skb->dev) {
3472             if (pt_prev) //分片相关
3473                 ret = deliver_skb(skb, pt_prev, orig_dev);
3474             pt_prev = ptype;
3475         }
3476     }
//解引用设备接收处理函数，如何存在会向上层发送接收到数据包
3500     rx_handler = rcu_dereference(skb->dev->rx_handler);
3501     if (rx_handler) {
3502         if (pt_prev) {
3503             ret = deliver_skb(skb, pt_prev, orig_dev);
3504             pt_prev = NULL;
3505         }
//根据处理结果，判断接下来对数据包如何进一步处理。
3506     switch (rx_handler(&skb)) {
//数据包已成功接收，不需要再处理
3507         case RX_HANDLER_CONSUMED:
3508             ret = NET_RX_SUCCESS;
3509             goto unlock;
//当rx_handler改变过skb->dev时，在接收回路中再一次处理。
3510         case RX_HANDLER_ANOTHER:
3511             goto another_round;
//不使用匹配的方式，精确传递。
3512         case RX_HANDLER_EXACT:
3513             deliver_exact = true;
//忽略rx_handler的影响。
3514         case RX_HANDLER_PASS:
3515             break;
3516         default:
3517             BUG();
3518     }
3519 }
//如果前面判段是精确发送方式，那么把null_or_dev设置成精确传送的设备。
3532     null_or_dev = deliver_exact ? skb->dev : NULL;
3533
3534     type = skb->protocol;
//根据套接字类型处理，802.3的type值是0001。
3535     list_for_each_entry_rcu(ptype,
3536         &ptype_base[nhtos(type) & PTYPE_HASH_MASK], list) {
3537         if (ptype->type == type &&
3538             (ptype->dev == null_or_dev || ptype->dev == skb->dev ||
3539             ptype->dev == orig_dev)) {
3540             if (pt_prev)
3541                 ret = deliver_skb(skb, pt_prev, orig_dev);
3542             pt_prev = ptype;
3543         }
3544     }
3565 }
```

由上面可以看出，各种接收数据包情况下最后的函数均指向了 `deliver_skb` 函数，到这里，代码不再在 `/net/core/dev.c` 中了，转到了 `net/ipv4/ip_input.c` 了，这里真真切切的将代码转到了 IP（网络）层了。

net/core/dev.c

```
1679 static inline int deliver_skb(struct sk_buff *skb,  
1680 struct packet_type *pt_prev,
```

```

1681             struct net_device *orig_dev)
1682 {
//在接收端要将分片的数据包重组，这里判断是否因缺少分片包而导致一个 skb 成为一个孤儿 skb。显然是不太可能是孤儿进
//程的,所以这里使用了 unlikely 知道 gcc 编译器优化程序，以减少指令流水被打断的概率。
1683     if (unlikely(skb_orphan_frags(skb, GFP_ATOMIC)))
1684         return -ENOMEM;
1685     atomic_inc(&skb->users); //增加 skb 的使用者计数器
1686     return pt_prev->func(skb, skb->dev, pt_prev, orig_dev);
1687 }

```

1686 行，对于 ip 数据包，这里的 func 是 ip_packet_type 结构体中的指针。为了弄清楚这个函数到底是怎么指向 ip_rcv 的需要向下接着看。以太网初始化函数 inet_init 调用了

```

dev_add_pack(&ip_packet_type);
static int __init inet_init(void)
{
    /*注册了四个种协议类型*/
    rc = proto_register(&tcp_prot, 1);
    rc = proto_register(&udp_prot, 1);
    rc = proto_register(&raw_prot, 1);
    rc = proto_register(&ping_prot, 1);
//对应协议初始化，ping 隶属 icmp 协议，在以前版本中没有独立出来，不过由于其重要性，所以默认各个系统都支持了。
    arp_init();
    ip_init();
    tcp_v4_init();
    tcp_init();
    udp_init();
    ping_init();
    ipv4_proc_init();

    dev_add_pack(&ip_packet_type);
}

```

dev_add_pack 函数原型和参数如下：

```

void dev_add_pack(struct packet_type *pt)
{
    struct list_head *head = ptype_head(pt);
    spin_lock(&ptype_lock);
    list_add_rcu(&pt->list, head);
    spin_unlock(&ptype_lock);
}
/参数/ af_inet.c
static struct packet_type ip_packet_type __read_mostly = {
    .type = cpu_to_be16(ETH_P_IP),
    .func = ip_rcv,
};

```

ptype_head 函数定义如下：

```

net/core/dev.c
struct list_head ptype_base[PTYPE_HASH_SIZE] __read_mostly;
static inline struct list_head *ptype_head(const struct packet_type *pt)
{
    if (pt->type == htons(ETH_P_ALL))
        return &ptype_all;
    else
        return &ptype_base[nthos(pt->type) & PTYPE_HASH_MASK];
}

```

4.2 进入网络层

到这里已经获得了 ip_rcv 的来历，并且知道 ip_rcv 已经被调用来处理接收到的 frame 了。

```
net/ipv4/ip_input.c
375 int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt, struct net_device *orig_dev)
376 {
    /* 如果 packet 的目的地址是其它主机，简单丢弃该数据包而不做处理 */
    383     if (skb->pkt_type == PACKET_OTHERHOST)
    384         goto drop;
    /*该宏跟新包相关统计信息，如丢弃、接收、过载等 */
    387     IP_UPD_PO_STATS_BH(dev_net(dev), IPSTATS_MIB_IN, skb->len);
    //检查该 skb 是否共享的，如果共享则会复制该数据包，在遇到内存申请失败时，会更新 discard 计数器。
    389     if ((skb = skb_share_check(skb, GFP_ATOMIC)) == NULL) {
    390         IP_INC_STATS_BH(dev_net(dev), IPSTATS_MIB_INDISCARDS);
    391         goto out;
    392     }
    393     /* 对接收的数据包 IP 头进行判断，如果 IP 头小于最短长度，即 IP 头有错，那么执行 395 行。
    394     if (!pskb_may_pull(skb, sizeof(struct iphdr)))
    395         goto inhdr_error;
    397     iph = ip_hdr(skb);
    409     /*头长和版本号判断。*/
    410     if (iph->ihl < 5 || iph->version != 4)
    411         goto inhdr_error;
    412     /*处理头可选项字段*/
    413     if (!pskb_may_pull(skb, iph->ihl*4))
    414         goto inhdr_error;
    416     iph = ip_hdr(skb);
    417 /*ip 校验*/
    418     if (unlikely(ip_fast_csum((u8 *)iph, iph->ihl)))
    419         goto csum_error;
    /*防火墙钩子函数，防火墙规则验证通过后对调用最后一个参数对应的函数，即 ip_rcv_finish */
    445     return NF_HOOK(NFPROTO_IPV4, NF_INET_PRE_ROUTING, skb, dev, NULL,
    446                     ip_rcv_finish);
    456 }
```

这个函数的处理逻辑思路非常清楚，首先是对 IP 数据包的正确性进行相关的验证，包括协议头长度、协议的版本号、IP 校验和等。然后调用传送数据的核心函数进行相应的函数（防火墙钩子函数）进行数据处理。

445 行，NF_HOOK 不是宏，而是一个函数，通常称为钩子函数，NF 是 netfilter 的缩写，netfilter 在 Linux 内核中被称为包过滤防火墙，netfilter 的内容见十一章。

在编译内核时没有配置 netfilter 时，NF_HOOK 最后一个参数被调用，此例中即执行 ip_forward_finish 函数；否则进入 HOOK 点，执行通过 nf_register_hook() 登记的功能（这句话表达意义的可能比较含糊，实际是进入 nf_hook_slow() 函数，再由它执行登记的函数）。

钩子函数不是重点，重点是 ip_rcv_finish 函数，该函数完成两个功能，其一是根据数据包目的地址和路由配置将数据包送给本地(local)主机或者发送出去(forward)，处理 IP 头字段中的可选字段。这里涉及到一个路由的概念，路由就是为 packet 找到合适的归宿，这个归宿可能是数据包被传送到本机协议栈上层，也可能通过网卡发送出去，关于 Linux 内核的 trie

路由内容见十二章。

```
311 static int ip_rcv_finish(struct sk_buff *skb)
312 {
313     const struct iphdr *iph = ip_hdr(skb); //获得 ip 头, 20 字节内容
314     struct rtable *rt; //rtable 是路由缓存
315     /*判断 skb 的路由缓存是否已有路由项，对于发往本机的数据（回环）该套接字的会包含路由项，如果没有路由项会调用
316      ip_route_input_noref 为 skb 选择一个合适的路由项，并将该信息存储在 skb 指向的路由选项指针*/
317     if (!skb_dst(skb)) {
318         int err = ip_route_input_noref(skb, iph->daddr, iph->saddr,
319                                       iph->tos, skb->dev);
320         if (unlikely(err)) {//找不到路由项，则进行出错处理
321             if (err == -EXDEV)
322                 NET_INC_STATS_BH(dev_net(skb->dev),
323                                   LINUX_MIB_IPRPFILTER);
324             goto drop;
325         }
326     }
327     //这里获得上面 skb 指向的路由项入口，_skb_refdst 是套接字的一个成员，如果 refcount 没有使用，则最低一个 bit 是 1，其它
328     //比特用于指示路由项的入口地址，根据 skb 类型，对多播和广播的统计信息如下，
329     rt = skb_rtable(skb);
330     if (rt->rt_type == RTN_MULTICAST) {
331         IP_UPD_PO_STATS_BH(dev_net(rt->dst.dev), IPSTATS_MIB_INMCAST,
332                             skb->len);
333     } else if (rt->rt_type == RTN_BROADCAST)
334         IP_UPD_PO_STATS_BH(dev_net(rt->dst.dev), IPSTATS_MIB_INBCAST,
335                             skb->len);
336     return dst_input(skb);// 对数据包的实际处理函数。
337 drop:
338     kfree_skb(skb);
339     return NET_RX_DROP; //如果 packet 被丢弃了，没有进行 forward 或者 local deliver 操作则返回 NET_RX_DROP。
340 }
```

到这里，从 ip_rcv 和 ip_rcv_finish 函数的注释中可以看出，ip 层首先检查 IP 头的正确性，然后根据 skb 找到对应的路由信息，找到路由信息后就可以调用 365 行的函数 dst_input 完成实际的接收工作了。关于网络层的路由查找见第十二章，365 行函数的原型如下：

```
static inline int dst_input(struct sk_buff *skb)
{
    return skb_dst(skb)->input(skb);
}
```

input 指针已经在 333 行建立路由项时进行赋值了，对于 local deliver，函数就是 ip_local_deliver，forward 就是 ip_forward。

```
244 int ip_local_deliver(struct sk_buff *skb)
245 {
246     //对于分片的 packet 就逆向合并成一个 packet，有些网卡硬件有 offload 特性，这个特性可以将分片和解分的过程在网卡处实现
247     //而不需要在 ip 层实现。此外，现在一般网络上的路由器都能支持 1500 的以太网数据包，有些还支持巨形包，分片的情况现在
248     //已经很少了。
249     if (ip_is_fragment(ip_hdr(skb))) {
250         if (ip_defrag(skb, IP_DEFRAG_LOCAL_DELIVER))
251             return 0;
252     }
253     //再一次看到 NF_HOOK 函数，这是钩子函数起作用的第二个地方，ip_local_deliver_finish 函数，完成实际的处理工作。
254     return NF_HOOK(NFPROTO_IPV4, NF_INET_LOCAL_IN, skb, skb->dev, NULL,
255                   ip_local_deliver_finish);
256 }
```

ip_local_deliver_finish()和 ip_rcv()在同一个文件里。

```
189 static int ip_local_deliver_finish(struct sk_buff *skb)
190 {
191     struct net *net = dev_net(skb->dev);
//移动 skb->data 指针跳过 ip 头，获得 tcp 层起始地址。
193     __skb_pull(skb, skb_network_header_len(skb));
195     rcu_read_lock();
196     {
197         int protocol = ip_hdr(skb)->protocol; //获得 ip 层的协议，V4 协议，传输层。
198         const struct net_protocol *ipprot; //对于 v4 版本该结构体的初始定义如下。
199     ****
200     这篇文章是按照 TCP-IP 来跟踪网络协议栈的，tcp 对应的 net_protocol 对应结构体
201     static const struct net_protocol tcp_protocol = {
202         .early_demux = tcp_v4_early_demux,
203         .handler = tcp_v4_rcv, //对应接收函数的指针。
204         .err_handler = tcp_v4_err,
205         .no_policy = 1,
206         .netns_ok = 1,
207     };
208     ****
209     resubmit:
210         raw = raw_local_deliver(skb, protocol); // raw socket 的 deliver 的方式。
211         ipprot = rcu_dereference(inet_protos[protocol]); //根据 protocol 获得 net_protocol 对应函数的指针。
212         if (ipprot != NULL) { //如果找到，对于 tcp 协议其将是 tcp_protocol，执行的是这段代码
213             int ret;
214             if (!ipprot->no_policy) {//使用策略
215                 if (!xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb)) {
216                     kfree_skb(skb);
217                     goto out;
218                 }
219             }
220             ret = ipprot->handler(skb);
221             if (ret < 0) {
222                 protocol = -ret;
223                 goto resubmit;
224             }
225             IP_INC_STATS_BH(net, IPSTATS_MIB_INDELIVERS);
226         } else { //对于 raw socket 的操作。
227             if (!raw) {
228                 if (xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb)) {
229                     IP_INC_STATS_BH(net, IPSTATS_MIB_INUNKNOWNPROTOS);
230                 }
231                 //icmp 协议的 packet 的 deliver 方式，该协议还是挺重要的，IP 获取，路由表信息的建立等会用到该协议 skb，这就验证了前面
232                 //说的，这个 sk_buff 贯串整个协议栈，只有在必要时才会赋值该 sk_buff 的一个副本。至此 ip 到 tcp 的接收流程已经梳理完毕。
233             }
234         }
235     out:
236         rcu_read_unlock(); //rcu 锁就是读写锁，支持并发读，而不支持并发写，为了防止写的 starvation，有写请求时会阻塞
237         读，可见写的优先级高于读，但是写不会抢占读。
```

239 }

215 行，结合 198 行上面的 .handler= tcp_v4_rcv, 可知，调用了 tcp_v4_rcv 其参数是需要接收的 skb 套接字。

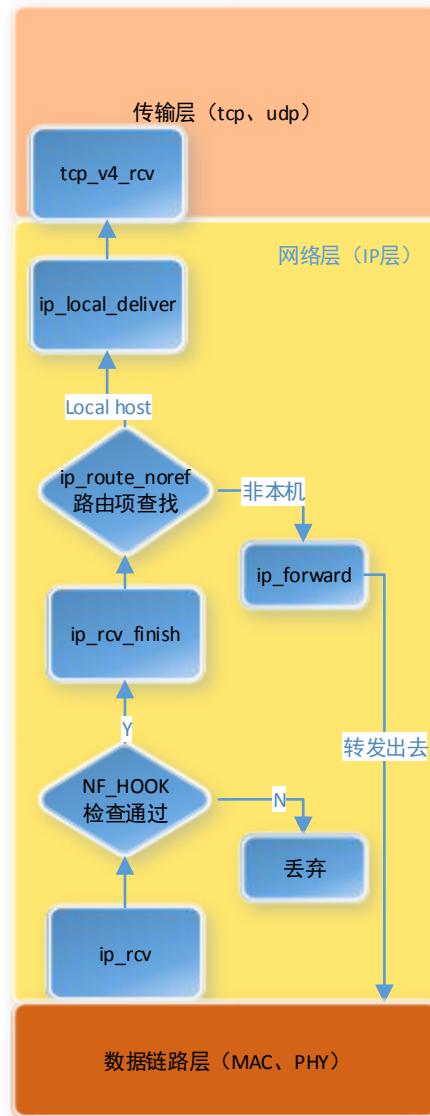


图 4.2 网络层接收函数调用流程

第五章 传输层（tcp）到网络层（ip）

根据数据的流向跟踪代码，由于数据发送是从 tcp 层到网络层再到网络到主机层，所以先来看 tcp 层向 ip 层发送数据的函数。

tcp 的发送函数和接收函数一样位于 net/ipv4/文件夹，文件名是 tcp_output.c 文件，传输层和网络层联系的函数是 tcp_transmit_skb(...):

在进入该函数时，先看该函数用到的一个重要的数据结构，其定义于 net/dccp/ipv4.c 文件，919 行是发送时用到的函数。

```
918 static const struct inet_connection_sock_af_ops dccp_ipv4_af_ops = {  
919     .queue_xmit    = ip_queue_xmit,  
920     .send_check    = dccp_v4_send_check,  
921     .rebuild_header = inet_sk_rebuild_header,  
922     .conn_request   = dccp_v4_conn_request,  
923     .syn_recv_sock  = dccp_v4_request_recv_sock,  
924     .net_header_len = sizeof(struct iphdr),  
925     .setsockopt     = ip_setsockopt,  
926     .getsockopt     = ip_getsockopt,  
927     .addr2sockaddr = inet_csk_addr2sockaddr,  
928     .sockaddr_len   = sizeof(struct sockaddr_in),  
929     .bind_conflict   = inet_csk_bind_conflict,  
934 };
```

919 行的函数服务于 ip 层，也就是四层模型中的网络层，我们先看传输层的函数 tcp_transmit_skb，该函数会建立 IP 层的头，并将该头传递给网络层。

```
net/ipv4/tcp_output.c  
840 static int tcp_transmit_skb(struct sock *sk, struct sk_buff *skb, int clone_it,  
841             gfp_t gfp_mask)  
842 {  
843     const struct inet_connection_sock *icsk = inet_csk(sk);  
844     struct inet_sock *inet;  
845     struct tcp_sock *tp;  
961     err = icsk->icsk_af_ops->queue_xmit(skb, &inet->cork.fl);  
}
```

icsk_af_ops->queue_xmit 的两个参数，struct sk_buff *skb 已经在第二章叙述过了，inet 的原型是 inet_sock 是 inet 的套接字的代表。

```
struct inet_sock {  
    /* sk and pinet6 has to be the first two members of inet_sock */  
    struct sock sk; //祖先的类  
    /* Socket demultiplex comparisons on incoming packets. */  
#define inet_daddr sk.__sk_common.skc_daddr //外部 ipv4 地址  
#define inet_rcv_saddr sk.__sk_common.skc_rcv_saddr //绑定的本地 ipv4 地址  
#define inet_addrpair sk.__sk_common.skc_addrpair  
#define inet_dport sk.__sk_common.skc_dport //目的端口号  
#define inet_num sk.__sk_common.skc_num //本地端口号  
#define inet_portpair sk.__sk_common.skc_portpair  
    __be32 inet_saddr;  
    __s16 uc_ttl; //单播的 TTL, time to live  
    __u16 cmsg_flags;  
    __be16 inet_sport;  
    __u16 inet_id;  
    struct ip_options_rcu __rcu
```

```

*inet_opt;
int rx_dst_ifindex;
__u8 tos;
__u8 min_ttl;
__u8 mc_ttl;
__u8 pmtudisc;
__u8 recverr:1,
is_icsk:1,
freebind:1,
hdrincl:1,
mc_loop:1,
transparent:1,
mc_all:1,
nodefrag:1;
__u8 rcv_tos;
int uc_index;
int mc_index;//多播设备索引
__be32 mc_addr;
struct ip_mc_socklist __rcu
*mc_list;
struct inet_cork_full
cork; //每个分片的 ip packet 构建 ip 头时用到的数据结构
};

```

ip_queue_xmit 这个函数就是网络层的传输函数了；

```

326 int ip_queue_xmit(struct sk_buff *skb, struct flowi *fl)
327 {
328     struct sock *sk = skb->sk;
329     struct inet_sock *inet = inet_sk(sk);
330     rt = ip_route_output_ports(sock_net(sk), fl4, sk, //获取输出信息的路由表，路由信息会填充到 skb 的 dst 字段去。
331                             daddr, inet->inet_saddr,
332                             inet->inet_dport,
333                             inet->inet_sport,
334                             sk->sk_protocol,
335                             RT_CONN_FLAGS(sk),
336                             sk->sk_bound_dev_if);
340     res = ip_local_out(skb); //发送出去
412 }

```

第 403 行是发送 packet 的函数， res = ip_local_out(skb); //发送出去

```

static inline int dst_output(struct sk_buff *skb)
{
    return skb_dst(skb)->output(skb);
}
int ip_local_out(struct sk_buff *skb)
{
    err = __ip_local_out(skb);
    if (likely(err == 1))
        err = dst_output(skb);
}

```

由上面两个函数，真正调用的其实是 rt->dst.output = ip_output 指向的函数，该函数将主要工作放到 ip_finish_output 去完成，这和接收时的方法很类似，并且这里也有一个钩子函数。

```

298 int ip_output(struct sk_buff *skb)
299 {
300     struct net_device *dev = skb_dst(skb)->dev;

```

```

301
302     IP_UPD_PO_STATS(dev_net(dev), IPSTATS_MIB_OUT, skb->len);
303
304     skb->dev = dev;
305     skb->protocol = htons(ETH_P_IP);
306
307     return NF_HOOK_COND(NFPROTO_IPV4, NF_INET_POST_ROUTING, skb, NULL, dev,
308                         ip_finish_output,
309                         !(IPCB(skb)->flags & IPSKB_REROUTED));
310 }

```

ip_finish_output 函数的定义如下，CONFIG_NETFILTER 和 CONFIG_XFRM 都是和安全相关的机制。231 行判断 ip 是否被分片了，通常以太网上的数据没有经过分片操作，如果分片了使用 ip_fragment 来处理，处理完了调用 ip_finish_output2 完成实质的发送工作。

```

222 static int ip_finish_output(struct sk_buff *skb)
223 {
224 #if defined(CONFIG_NETFILTER) && defined(CONFIG_XFRM)
225     /* Policy lookup after SNAT yielded a new policy */
226     if (skb_dst(skb)->xfrm != NULL) {
227         IPCB(skb)->flags |= IPSKB_REROUTED;
228         return dst_output(skb);
229     }
230 #endif
231     if (skb->len > ip_skb_dst_mtu(skb) && !skb_is_gso(skb))
232         return ip_fragment(skb, ip_finish_output2);
233     else
234         return ip_finish_output2(skb);
235 }

```

ip_finish_output2 对 packet 的头进行处理，然后根据路由表寻找下一跳地址，这里还涉及一层邻居协议，这个没有路由那么庞大。

```

166 static inline int ip_finish_output2(struct sk_buff *skb)
167 {
196     nexthop = __force u32 rt_nexthop(rt, ip_hdr(skb)->daddr);
197     neigh = __ipv4_neigh_lookup_noref(dev, nexthop);
198     if (unlikely(!neigh))
199         neigh = __neigh_create(&arp_tbl, &nexthop, dev, false);
200     if (!IS_ERR(neigh)) {
201         int res = dst_neigh_output(dst, neigh, skb);
202     }

```

neigh 是 struct neighbour 类型的结构体，这个数据邻居协议的范畴。197 是邻居协议查找合适的网卡设备，198 行如果找不到则执行 199 行的函数创建一个，在获得邻居项之后执行 201 行的函数。该函数位于 include/net/dst.h，该函数未定义于 ipv4 的目录下，这也意味着数据将传递到主机到网络层了。

```

393 static inline int dst_neigh_output(struct dst_entry *dst, struct neighbour *n,
394                                     struct sk_buff *skb)
395 {
396     const struct hh_cache *hh;
397
398     if (dst->pending_confirm) {
399         unsigned long now = jiffies;
400
401         dst->pending_confirm = 0;
402         /* avoid dirtying neighbour */

```

```

403     if (n->confirmed != now)
404         n->confirmed = now;
405     }
406
407     hh = &n->hh;
408     if ((n->nud_state & NUD_CONNECTED) && hh->hh_len)
409         return neigh_hh_output(hh, skb); //支持硬件缓存头方法的发送
410     else
411         return n->output(n, skb); //不支持硬件缓存头的方法的发送。
412 }

```

第 411 行调用的是 `neigh_resolve_output`, 该函数是通过路由表查到的, 不论硬件支不支持硬件头缓存, `neigh_resolve_output` 都会被调用到, 它来源如下:

```

static const struct neigh_ops arp_generic_ops = {
    .family = AF_INET,
    .solicit = arp_solicit,
    .error_report = arp_error_report,
    .output = neigh_resolve_output,
    .connected_output =
    neigh_connected_output,
};

//有硬件头缓存的函数操作集
static const struct neigh_ops arp_hh_ops = {
    .family = AF_INET,
    .solicit = arp_solicit,
    .error_report = arp_error_report,
    .output = neigh_resolve_output,
    .connected_output =neigh_resolve_output,
};

```

获得上述 `output` 函数的过程就是路由的过程, 还是来看看 `neigh_resolve_output`, 它定义于 `net/core/neighbour.c` 文件。这时真正的离开了 IP 层, 该函数的 1310 会调用之一文章讲述的函数将数据实际的发送出去。

```

1286 int neigh_resolve_output(struct neighbour *neigh, struct sk_buff *skb)
1287 {
1288     struct dst_entry *dst = skb_dst(skb);
1289     int rc = 0;
1290
1291     if (!dst)
1292         goto discard;
1293
1294     if (!neigh_event_send(neigh, skb)) {
1295         int err;
1296         struct net_device *dev = neigh->dev;
1297         unsigned int seq;
1298
1299         if (dev->header_ops->cache && !neigh->hh.hh_len)
1300             neigh_hh_init(neigh, dst);
1301
1302         do {
1303             __skb_pull(skb, skb_network_offset(skb));
1304             seq = read_seqbegin(&neigh->ha_lock);
1305             err = dev_hard_header(skb, dev, ntohs(skb->protocol),
1306                                   neigh->ha, NULL, skb->len);
1307         } while (read_seqretry(&neigh->ha_lock, seq));
1308

```

```

1309     if (err >= 0)
1310         rc = dev_queue_xmit(skb);

1313     }
}

```

最后还是看一张图来回顾 tcp/ip 发送数据的流程。

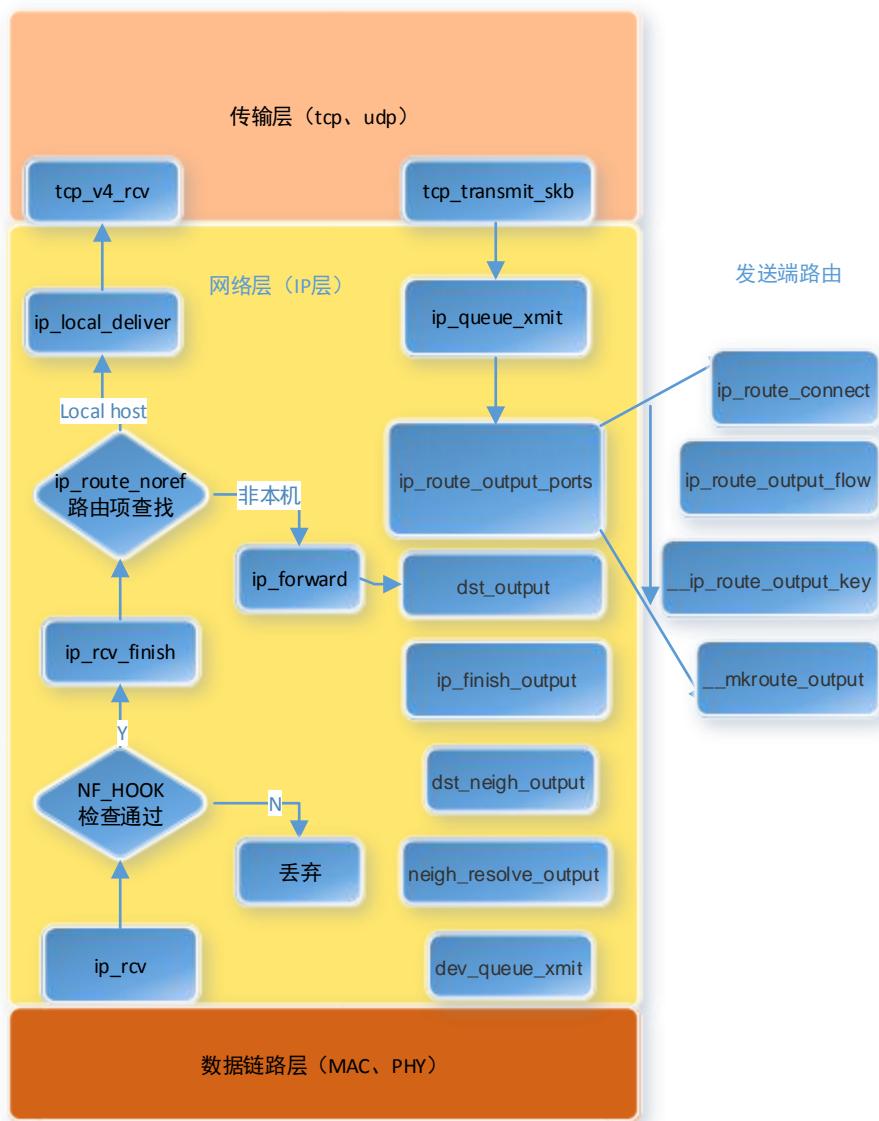


图 5.1 网络层函数调用流程

第六章 应用层

前述文章代码由下至上分析，这次我们来个从上至下的分析，从应用层开始，看看应用层（用户空间）是如何将数据传递给内核的。在应用程序编程时，常常可以看见如下接口：

```
sockfd = socket(AF_INET, SOCK_STREAM, 0)
connect(sockfd, (struct sockaddr *)&server_addr, sizeof(struct sockaddr))
recv(sockfd, recv_buffer, BUFFER_SIZE, 0)
bind(sockfd, (struct sockaddr *)&server_addr, sizeof(struct sockaddr))
listen(sockfd, 5) == -1
accept(sockfd, (struct sockaddr *)&client_addr, (socklen_t *)&sin_size)
send(sock_fd, snd_buff, BUFFER_SIZE, 0)
```

这些同接口通常用于客户端和服务器端的程序中，这是应用程序调用的接口经由 C 库会调用内核里的实现函数，这里的调用通常称为系统调用。

```
//include/linux/syscall.h
asm linkage long sys_socketcall(int call, unsigned long __user *args);
```

该函数的定义由下面的 2460 行的 SYSCALL_DEFINE2 定义，上面的 recv 等和 sys_socketcall 函数定义于同一文件中，这里就只列出 sys_socketcall 函数了。

```
//net/socket.c
2460 SYSCALL_DEFINE2(socketcall, int, call, unsigned long __user *, args)
2461 {
2462     unsigned long a[AUDITSC_ARGS];
2463     unsigned long a0, a1;

2474     /* copy_from_user should be SMP safe. */
2475     if (copy_from_user(a, args, len))
...
2482     a0 = a[0];
2483     a1 = a[1];
2484
2485     switch (call) {
2486         case SYS_SOCKET:
2487             err = sys_socket(a0, a1, a[2]);
2488             break;
2489         case SYS_BIND:
2490             err = sys_bind(a0, (struct sockaddr __user *)a1, a[2]);
2491             break;
2492         case SYS_CONNECT:
2493             err = sys_connect(a0, (struct sockaddr __user *)a1, a[2]);
2494             break;
2495         case SYS_LISTEN:
2496             err = sys_listen(a0, a1);
...
2502         case SYS_GETSOCKNAME:
2503             err =
2504                 sys_getsockname(a0, (struct sockaddr __user *)a1,
2505                               (int __user *)a[2]);
2506             break;
2507         case SYS_GETPEERNAME:
2508             err =
2509                 sys_getpeername(a0, (struct sockaddr __user *)a1,
2510                               (int __user *)a[2]);
2511             break;
2512     case SYS_SOCKETPAIR:
```

```

2513     err = sys_socketpair(a0, a1, a[2], (int __user *)a[3]);
2514     break;
2515 case SYS_SEND:
2516     err = sys_send(a0, (void __user *)a1, a[2], a[3]);
2517     break;
2518 case SYS_SENDTO:
2519     err = sys_sendto(a0, (void __user *)a1, a[2], a[3],
2520                      (struct sockaddr __user *)a[4], a[5]);
2521     break;
2522 case SYS_RECV:
2523     err = sys_recv(a0, (void __user *)a1, a[2], a[3]);
2524     break;
2525 case SYS_RECVFROM:
2526     err = sys_recvfrom(a0, (void __user *)a1, a[2], a[3],
2527                        (struct sockaddr __user *)a[4],
2528                        (int __user *)a[5]);
...
2563 }

```

C 库已经对网络方面的套接字函数均调用的是 `sys_socketcall` 函数。该函数的若干 case 语句表明其只充当了分配器的角色，该函数会调用对应的系统调用，由于它们的套路都一样，这里只看 2487 行的 socket 的创建函数 `err = sys_socket(a0, a1, a[2])`，

```

//net/socket.c
1361 SYSCALL_DEFINE3(socket, int, family, int, type, int, protocol)
1362 {

1381     retval = sock_create(family, type, protocol, &sock);
1382     if (retval < 0)
1383         goto out;

1396 }

```

1381 行各参数的意义根据创建的套接字类型不同而有所差异，在最开始给出的创建套接字的原型如下：

```
sockfd = socket(AF_INET, SOCK_STREAM, 0)
```

这里 `family` 对应第一参数，即协议族，这关系到后面数据所经过的协议栈，第二个参数是套接字类型，`SOCK_STREAM` 是面向连接的流式套接字，常用于 TCP 服务，最后的协议参数是协议，为 0 表示由程序选择合适的协议类型。`sock_create` 函数的定义如下：

```

1349 int sock_create(int family, int type, int protocol, struct socket **res)
1350 {
1351     return __sock_create(current->nsproxy->net_ns, family, type, protocol, res, 0);
1352 }

```

该函数调用 `__sock_create` 完成实际的创建工作，它的两个下划线表示它是内核中比较重要的函数，对其修改需要慎重。第一个参数 `current->nsproxy->net_ns` 是一个命名空间相关的内容，命名空间和 container 实现一起用于实现轻量级虚拟化技术 LXC。`current` 是描述当前进程的结构体，其类型是 `task_struct`，该结构体是内核最庞大的结构体之一，相关的字段涉及内核的其它子系统，`__sock_create` 就是在当前进程的网络命名空间中创建一个应用程序指定的套接字，不同命名空间中看到的网络拓扑可以是不一样的，这样可以带来隔离的好处。这里来看 `__sock_create`，命名空间的内容就不在这里涉及了。

```
1236 int __sock_create(struct net *net, int family, int type, int protocol,
```

```

1237     struct socket **res, int kern)
1238 {
1275     sock = sock_alloc();
1296     pf = rcu_dereference(net_families[family]);
1311     err = pf->create(net, sock, protocol, kern);
1346 }

```

1275 行是创建一个 inode 和 socket 对象，inode 是文件系统的索引节点，其和超级块一起用于管理文件系统，VFS 层会使用这些信息以实现不同的文件系统使用相对统一的接口管理，1296 行是根据相应的协议族找到对应的创建函数。tcp/ip 协议族注册，在《网络子系统初始化过程》文末略有提及，这里使用的 socket 创建函数是 net/ipv4/af_inet.c 函数。这个 socket 创建由具体的协议族完成。

```

static const struct net_proto_family inet_family_ops = {
    .family = PF_INET,
    .create = inet_create,
    .owner   = THIS_MODULE,
};

```

到这里已经能够理清 socket 的创建过程是如何从应用程序到内核的网络协议栈的，这时我们可以看 socket 接收和发送数据的接口了，这里给出它们的流程图，具体代码参看内核的实现。

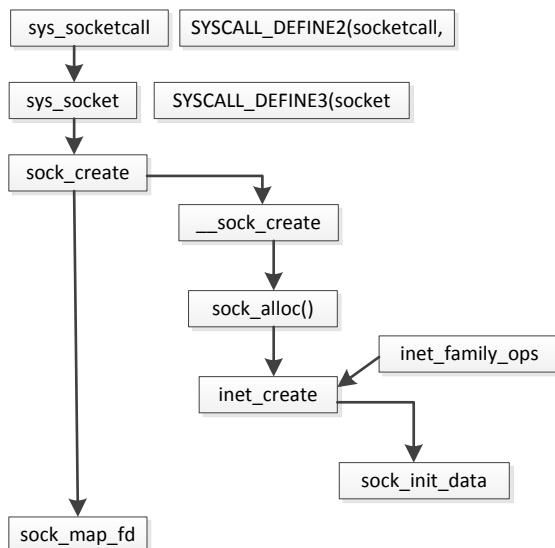


图 6.1 流式套接字创建流程

接下来看看应用层的发送和接收是如何同传输层进行通信的，由上面的发送和接收图，可以看出，对于发送则 sock_sendmsg 和 inet_sendmsg 之间的调用，对于接收是 sock_recvmsg 和 inet_recvmsg 之间的交互。

对于发送，在 sock_sendmsg 和 inet_sendmsg 之间 __sock_sendmsg_nosec 会被调用，该函数的 626 行的根据对应的协议栈的发送函数进行发送数据。

```

616 static inline int __sock_sendmsg_nosec(struct kiocb *iocb, struct socket *sock,
617                                         struct msghdr *msg, size_t size)
618 {
619     struct sock_iocb *si = iocb_to_siocb(iocb);

```

```

620
621     si->sock = sock;
622     si->scm = NULL;
623     si->msg = msg;
624     si->size = size;
625
626     return sock->ops->sendmsg(iocb, sock, msg, size);
627 }

```

在流式套接字中的 sendmsg 和 recvmsg 这两个函数会被调用到。

```

const struct proto_ops inet_stream_ops = {
    .family      = PF_INET,
    .owner       = THIS_MODULE,
    .release     = inet_release,
    .bind        = inet_bind,
    .connect     = inet_stream_connect,
    .socketpair   = sock_no_socketpair,
    .accept      = inet_accept,
    .getname     = inet_getname,
    .poll        = tcp_poll,
    .ioctl       = inet_ioctl,
    .listen      = inet_listen,
    .shutdown    = inet_shutdown,
    .setsockopt  = sock_common_setsockopt,
    .getsockopt  = sock_common_getsockopt,
    .sendmsg     = inet_sendmsg,
    .recvmsg     = inet_recvmsg,
};

```

inet_recvmsg 和 inet_sendmsg 于 inet_stream_ops 都定义于 net/ipv4/af_inet.c 函数，其又调用具体的协议进行处理，这里是 tcp_prot 协议中的 tcp_recvmsg 和 tcp_sendmsg 函数。

```

int inet_recvmsg(struct kiocb *iocb, struct socket *sock, struct msghdr *msg,
                  size_t size, int flags)
{
    err = sk->sk_prot->recvmsg(iocb, sk, msg, size, flags & MSG_DONTWAIT,
                                 flags & ~MSG_DONTWAIT, &addr_len);
}
int inet_sendmsg(struct kiocb *iocb, struct socket *sock, struct msghdr *msg,
                  size_t size)
{
    return sk->sk_prot->sendmsg(iocb, sk, msg, size);
}

```

tcp_prot 协议的成员如下，这些成员的初始化是在网络子系统初始化时完成的，在《网络子系统初始化过程》。

```

struct proto tcp_prot = {
    .name = "TCP",
    .close = tcp_close,
    .connect = tcp_v4_connect,
    .disconnect = tcp_disconnect,
    .accept = inet_csk_accept,
    .ioctl = tcp_ioctl,
    .init = tcp_v4_init_sock,
    .destroy = tcp_v4_destroy_sock,
    .shutdown = tcp_shutdown,
    .setsockopt = tcp_setsockopt,
    .getsockopt = tcp_getsockopt,

```

```
.recvmsg = tcp_recvmsg,  
.sendmsg = tcp_sendmsg,
```

```
};
```

到这里可以有个困惑，那就是为什么 `sock->ops->sendmsg(iocb, sock, msg, size)` 会调用 `inet_sendmsg`，又为什么 `sk->sk_prot->sendmsg(iocb, sk, msg, size)` 对应 `tcp_sendmsg`；这就又要回到上面的套接字创建函数 `inet_create` 了。

```
275 static int inet_create(struct net *net, struct socket *sock, int protocol,  
276                         int kern)  
277 {  
278     struct sock *sk;  
279     struct inet_protosw *answer;  
280     struct inet_sock *inet;  
281     struct proto *answer_prot;  
282     list_for_each_entry_rcu(answer, &inetsw[sock->type], list) {  
283  
284         /* Check the non-wild match. */  
285         if (protocol == answer->protocol) {  
286             if (protocol != IPPROTO_IP)  
287                 break;  
288         } else {  
289             /* Check for the two wild cases. */  
290             if (IPPROTO_IP == protocol) {  
291                 protocol = answer->protocol;  
292                 break;  
293             }  
294             if (IPPROTO_IP == answer->protocol)  
295                 break;  
296         }  
297         err = -EPROTONOSUPPORT;  
298     }  
299     sock->ops = answer->ops;  
300     answer_prot = answer->prot;  
301     answer_no_check = answer->no_check;  
302     answer_flags = answer->flags;  
303     sk = sk_alloc(net, PF_INET, GFP_KERNEL, answer_prot);  
304  
305     ...  
306 }
```

第 297 行的 `inetsw` 是一个双链表数组，这些数组的下标对应于用户空间的参数，如 `STREAM`, `DGRAM`, `RAW` 等，根据这个链表可以找到 `struct inet_protosw` 类型的 `answer` 成员，这个链表在 `tcp/ip` 协议栈初始化时就完成了，该链表的形式如下，流式套接字对应于第一个成员。第 343 的 `ops` 对于与 `inetsw_array[]` 的第一成员的 `inet_stream_ops`，至此，找到了 `inet_sendmsg` 的来历了。

```
static struct inet_protosw inetsw_array[] =  
{  
{  
.type =      SOCK_STREAM,  
.protocol =   IPPROTO_TCP,  
.prot =       &tcp_prot,  
.ops =        &inet_stream_ops,  
.no_check =   0,
```

```

.flags =     INET_PROTOSW_PERMANENT |
             INET_PROTOSW_ICSK,
},
{
.type =     SOCK_DGRAM,
.protocol = IPPROTO_UDP,
.prot =     &udp_prot,
.ops =     &inet_dgram_ops,
.no_check = UDP_CSUM_DEFAULT,
.flags =     INET_PROTOSW_PERMANENT,
},
...
};

```

inet_create 函数的 352 sk = sk_alloc(net, PF_INET, GFP_KERNEL, answer_prot), 其中 answer_prot 在 344 行被赋值成了 &tcp_prot 成员。

```

//net/core/sock.c
1363 struct sock *sk_alloc(struct net *net, int family, gfp_t priority,
1364                         struct proto *prot)
1365 {
1366     struct sock *sk;
1367
1368     sk = sk_prot_alloc(prot, priority | __GFP_ZERO, family);
1370     sk->sk_family = family; //协议族信息保留
1375     sk->sk_prot = sk->sk_prot_creator = prot;
1383
1384     return sk;
1385 }

```

该函数首先调用 sk_prot_alloc 创建一个 sock 对象，然后在 1375 行将该对象的 sk_prot 成员赋值成 prot，也就是 tcp_prot，该函数返回 sock 类型的指针以做进一步的处理工作，比如调用 tcp_prot 的初始化函数 tcp_v4_init_sock。该结构体的指针即是传输层的，这样就切换到 tcp 协议处理了。

接收的过程一样，函数 tcp 层函数的指针同发送指针一样位于 tcp_prot 结构体中。

第七章 tcp 发送（传输层）

由第五章可知，`sock_recvmsg` 和 `tcp_sendmsg` 用于 `tcp` 层和应用层的接口，由第四章可知，`tcp_v4_rcv` 和 `tcp_transmit_skb` 是传输层和网络层之间的接口，现在来看看 `tcp_sendmsg` 是如何到 `tcp_transmit_skb`，`tcp_v4_rcv` 又是如何到 `sock_recvmsg` 的。

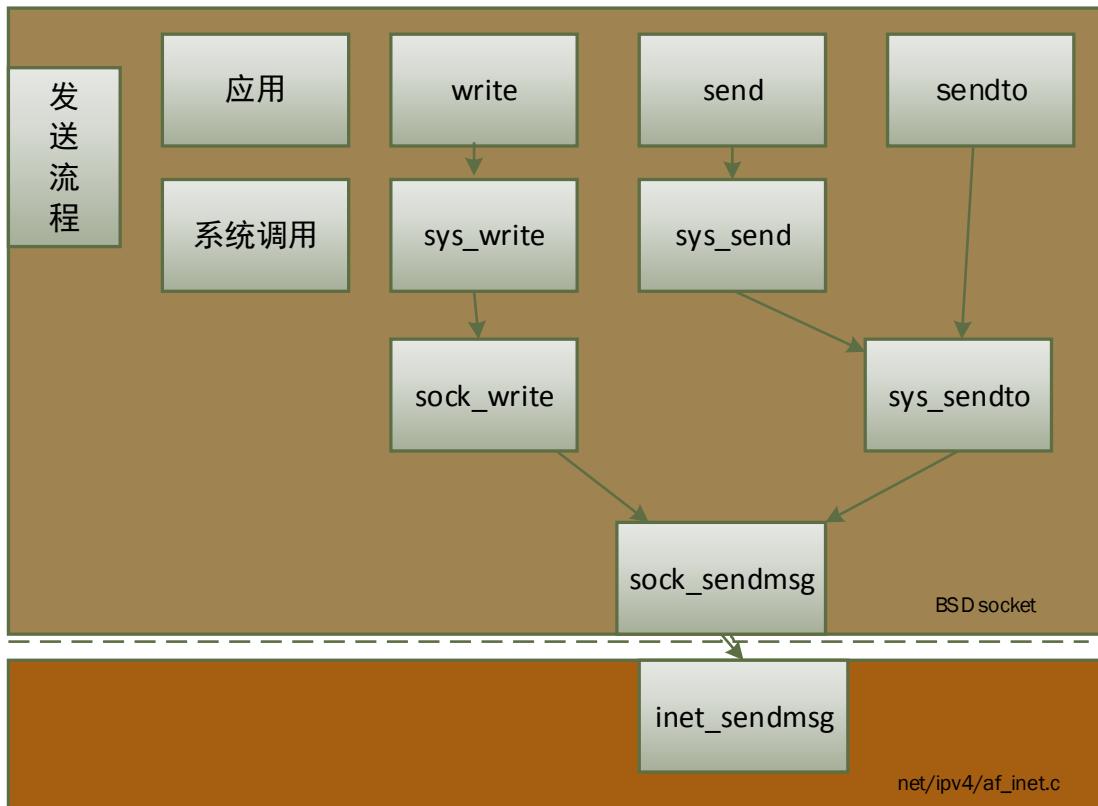


图 7.1 套接字发送

`sys_send` 的参数意义如下，`fd` 是套接字 ID，`buff` 是要发送的内容，`len` 是要发送的长度。

net/socket.c

```
SYSCALL_DEFINE4(send, int, fd, void __user *, buff, size_t, len, unsigned int, flags)
{
    return sys_sendto(fd, buff, len, flags, NULL, 0);
}
```

`flags` 的参数意义如表 7-1：

flags	说明	recv	send
MSG_DONTROUTE	绕过路由表查找		•
MSG_DONTWAIT	仅本操作非阻塞	•	•
MSG_OOB	发送或接收带外数据	•	•
MSG_PEEK	窥看外来消息	•	
MSG_WAITALL	等待所有数据	•	

```
char snd_buff[BUFFER_SIZE] = "tcp/ip protocol stack";
send(sock_fd,snd_buff,BUFFER_SIZE,0);
```

sys_sendto 的参数 fd 是 sock_fd, 之前的应用程序中创建的, buff 对应的是 snd_buff, len 对应的是 BUFFER_SIZE, flags 对应就是 0, addr 是 NULL, addr_len 是 0。

这里的 buff 是用户空间的地址, 内核空间和用户空间的交互需要使用 copy_from_user 和 copy_to_user 接口。

net/socket.c

```
1754 SYSCALL_DEFINE6(sendto, int, fd, void __user *, buff, size_t, len,
1755         unsigned int, flags, struct sockaddr __user *, addr,
1756         int, addr_len)
1757 {
1758     struct socket *sock;
...
1761     struct msghdr msg;
1762     struct iovec iov;
//根据套接字 ID, 查找应用程序 socket () 调用第 6 章的 sys_sock () 创建的套接字。
1767     sock = sockfd_lookup_light(fd, &err, &fput_needed);...
1771     iov.iov_base = buff;//用户空间的地址, 被记录于这个字段了。
1772     iov.iov_len = len; //此用于记录用户空间需要传递数据的字节数, 对应这里就是 BUFFER_SIZE。
1773     msg.msg_name = NULL; //如果查看之一文章, message 在四层模型中位于应用层和传输层之间。后面的发送将使用
msg 结构体。
1774     msg.msg_iov = &iov; //将地址信息存入 msg 结构体中。
1775     msg.msg_iovlen = 1;
1776     msg.msg_control = NULL;
1777     msg.msg_controllen = 0;
1778     msg.msg_namelen = 0;
...
1788     msg.msg_flags = flags;
1789     err = sock_sendmsg(sock, &msg, len);
...
1794     return err;
1795 }
```

sys_sendto 将有价值的信息都保存了 msg 中, 接着调用了 sock_sendmsg 函数, 该函数的第一个参数是创建的套接字, 第二个参数包含了要发送数据的一些信息, 最后一个参数是待发送数据的长度, 对于 32 位机型, 该长度总是小于等于 32 位无符号数所能表示的最大长度。msg 的各字段:

```
struct msghdr {
void *msg_name; /* Socket name, 上例中其赋值为了 NULL*/
int msg_namelen; /* Length of name ,长度被赋值为 0 了*/
struct iovec *msg_iov; /* Data blocks, 这里存放了用户空间待发送数据的首地址和发送数据的字节数*/
__kernel_size_t msg_iovlen; /* Number of blocks, 发送的 block 数, 只有一个, 这里赋值等于 1*/
void *msg_control; /* Per protocol magic (eg BSD file descriptor passing), 协议相关控制信息, NULL */
__kernel_size_t msg_controllen; /* Length of cmsg list , 长度同样为 NULL*/
unsigned int msg_flags; /*这个 flag 使用的是 socket 传递进来的参数 0*/
};
```

sock_sendmsg()是对 __sock_sendmsg_nosec()函数的封装,

```
//net/socket.c
616 static inline int __sock_sendmsg_nosec(struct kiocb *iocb, struct socket *sock,
617             struct msghdr *msg, size_t size)
618 {
619     struct sock_iocb *si = kiocb_to_siocb(iocb);
620
621     si->sock = sock;
```

```

622     si->scm = NULL;
623     si->msg = msg;
624     si->size = size;
625
626     return sock->ops->sendmsg(iocb, sock, msg, size);
627 }
628
629 static inline int __sock_sendmsg(struct kiocb *iocb, struct socket *sock,
630                                 struct msghdr *msg, size_t size)
631 {
632     int err = security_socket_sendmsg(sock, msg, size);
633
634     return err ?: __sock_sendmsg_nosec(iocb, sock, msg, size);
635 }
636
637 int sock_sendmsg(struct socket *sock, struct msghdr *msg, size_t size)
638 {
639     struct kiocb iocb; //io 控制块，每个 IO 请求都会对应一个该结构体。
640     struct sock_iocb siocb; //套接字的 io 控制块，每个套接字 IO 请求会对应一个该结构体。
641     int ret;
642
643     init_sync_kiocb(&iocb, NULL); //该函数的工作见下文
644     iocb.private = &siocb; //将套接字 IO 控制块，存放在私有字段。
//调用上面 629 行的函数，632 行安全检查，目前只是个框架，无实质内容，检查符合安全后调用 634 行的__sock_sendmsg_nosec
进行发送。
645     ret = __sock_sendmsg(&iocb, sock, msg, size);
646     if (-EIOCBQUEUED == ret)
647         ret = wait_on_sync_kiocb(&iocb);
648     return ret;
649 }

```

__sock_sendmsg_nosec 函数参数如下：

iocb: io 控制块，在 `sock_sendmsg()` 中定义；
sock: 对应根源是应用程序创建的套接字；
msg: 存放的是用户空间待发送的数据地址和以字节计数的长度。
size: 待发送数据的字节数。

626 的函数具体设置在第五章中提过了，是 `inet_sendmsg()` 函数。

```

//include/linux/aio.h
74 static inline void init_sync_kiocb(struct kiocb *kiocb, struct file *filp)
75 {
76     *kiocb = (struct kiocb) {
77         .ki_users = ATOMIC_INIT(1), //引用计数设置成一。
78         .ki_ctx = NULL, //设置成 0，表示是同步 IO 操作。
79         .ki_filp = filp, //filp 传递的参数是 NULL。
80         .ki_obj.tsk = current, //描述当前进程的结构体
81     };
82 }

//net/ipv4/af_inet.c，该函数的参数意义参考上面__sock_sendmsg_nosec()。
758 int inet_sendmsg(struct kiocb *iocb, struct socket *sock, struct msghdr *msg,
759                     size_t size)
760 {
763     sock_rps_record_flow(sk); //软中断均衡（多核绑定）
770     return sk->sk_prot->sendmsg(iocb, sk, msg, size);
771 }

```

770 行调用的函数在第五章提到过该函数，对于 tcp 协议调用的是 `tcp_sendmsg()` 函数。

net/ipv4/tcp.c

```
1016 int tcp_sendmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
1017         size_t size)
1018 {
1019     struct iovec *iov;
1020     struct tcp_sock *tp = tcp_sk(sk);
1021     struct sk_buff *skb;
1022     int iovlen, flags, err, copied = 0;
1023     int mss_now = 0, size_goal, copied_syn = 0, offset = 0;
1024     bool sg;
1025     long timeo;
1026
1027     lock_sock(sk);
1028
1029     flags = msg->msg_flags;
//如果设置了 fastopen 标志，则在发送 SYN 同步包时，也会发送一部分数据。
1030     if (flags & MSG_FASTOPEN) {
1031         err = tcp_sendmsg_fastopen(sk, msg, &copied_syn);
1032         if (err == -EINPROGRESS && copied_syn > 0)
1033             goto out;
1034         else if (err)
1035             goto out_err;
1036         offset = copied_syn;
1037     }
//该值是发送等待超时值，如果设置了 MSG_DONTWAIT 标志，则是非阻塞 IO，发送完立即返回，否则则会等待一个超时值。
1038     timeo = sock_sndtimeo(sk, flags & MSG_DONTWAIT);
1039
1040 /* 等待 tcp 建立连接，TCPF_ESTABLISHED 和 TCPF_CLOSE_WAIT 均位于 tcp 已建立连接（connect 函数会完成
 * 三次握手）状态，对于 TCP Fast Open 模式，可以在连接完全建立前发送数据包
 * 等待的时长是 1039 行获得的参数。 */
1041     if (((1 << sk->sk_state) & ~(TCPF_ESTABLISHED | TCPF_CLOSE_WAIT)) &&
1042         !tcp_passive_fastopen(sk)) {
1043         //不处在可发送情况下，则需要等待 timeo 时间，以便建立连接
1044         if ((err = sk_stream_wait_connect(sk, &timeo)) != 0)
1045             goto do_error;
1046     }
//tp->repair 是指 tcp 连接收到了破坏，需要进行修复，这种情况在虚拟机或者 LXC 被切到新的物理设备上，可能使用的 NICs
//改变，这就需要对 tcp 的收发队列进行处理，以使 tcp 连接不会因为切换而断开。
1047     if (unlikely(tp->repair)) {
1048         if (tp->repair_queue == TCP_RECV_QUEUE) {
1049             copied = tcp_send_rcvq(sk, msg, size);
1050             goto out;
1051         }
1052         err = -EINVAL;
1053         if (tp->repair_queue == TCP_NO_QUEUE) //如果设置了需要 repair，但是队列又什么也没有，则出错
1054             goto out_err;
1055     }
1056
1057     /* This should be in poll */
1058     clear_bit(SOCK_ASYNC_NOSPACE, &sk->sk_socket->flags);
1059
1060 /**
1061 ** 获取 mss 值，根据 pmtu 和接收方通告窗口大小设置 mss 值，另外还要考虑自身 NICs 是否支持 GSO，如果支持，则 tcp
** 向 IP 发送的数据包文大小将是 MSS 的整数倍，最大 64K。
1062 */
1063
1064     mss_now = tcp_send_mss(sk, &size_goal, flags);
```

```

1068
1069 /* Ok commence sending.*/
1070 iovlen = msg->msg iovlen; //用户空间传输数据的字节长度
1071 iov = msg->msg iov; //用户空间待传递数据的用户空间地址。
1072 copied = 0; //已经拷贝的数据字节长度
1073
1074 err = -EPIPE;
1075 if (sk->sk_err || (sk->sk_shutdown & SEND_SHUTDOWN))
1076     goto out_err;
//向量 IO( scatter/gather ) , 可以从一个 stream 向多个 buffer 写，或者从多个 bufer 向一个 stream 中写。
1078 sg = !(sk->sk_route_caps & NETIF_F_SG);
//判断应用程序要发送的数据是否全部发送了，如果没有则执行 while 循环体内的代码。
1080 while (--iovlen >= 0) {
1081     size_t seglen = iov->iov_len;
1082     unsigned char __user *from = iov->iov_base;
1083
1084     iov++;
1085     if (unlikely(offset > 0)) { /* Skip bytes copied in SYN */
1086         if (offset >= seglen) {
1087             offset -= seglen;
1088             continue;
1089         }
1090         seglen -= offset;
1091         from += offset;
1092         offset = 0;
1093     }
1094
1095     while (seglen > 0) {
/*max=size_goal=tp->xmit_size_goal,表示发送数据报到达网络设备时数据段的最大长度，该长度用来分割数据，TCP 发送报文时，每个 SKB 的大小不能超过该值。在不支持 GSO 或 TSO 情况下，xmit_size_goal 就等于 MSS；而如果支持 GSO，则 xmit_size_goal 会是 MSS 的整数倍。数据报发送到网络设备后再由 NICs 根据 MSS 进行分割。*/
1096         int copy = 0;
1097         int max = size_goal;
//获取传输控制块发送队列的尾部的那个 SKB，因为只有队尾的那个 SKB 才有可能存在剩余空间的
1098         skb = tcp_write_queue_tail(sk);
//判断 sk_send_head 节点上待发送的 sk_buff 是否为空，sk_buff 在第二章中有过叙述。
1099         if (tcp_send_head(sk)) {
//CHECKSUM_NONE 表示 csum 值无意义，通常需要 tcp 层自己校验，但多数网卡有硬件校验功能。
1100             if (skb->ip_summed == CHECKSUM_NONE)
1101                 max = mss_now;
1102             copy = max - skb->len; //获得有效载荷（payload）
1103         }
1104     }
//如果 copy 小于零说明，传输的数据长度超出了 tcp 允许的长度，这时需要进行分片操作。
1105     if (copy <= 0) {
1106 new_segment:
1107         /* Allocate new segment. If the interface is SG,
1108             * allocate skb fitting to single page.
1109             */
1110         if (!sk_stream_memory_free(sk)) //判断发送缓冲区剩余空闲有没有
1111             goto wait_for_sndbuf;
1112
1113         skb = sk_stream_alloc_skb(sk,
1114                                 select_size(sk, sg),
1115                                 sk->sk_allocation);
1116         if (!skb)
1117             goto wait_for_memory;
1118
1119

```

```

1120      /*
1121       * Check whether we can use HW checksum.
1122       */
1123     if (sk->sk_route_caps & NETIF_F_ALL_CSUM)
1124         skb->ip_summed = CHECKSUM_PARTIAL;
1125
1126     skb_entail(sk, skb);
1127     copy = size_goal;
1128     max = size_goal;
1129 }
1130
1131     /* Try to append data to the end of skb, 预留填充以符合包长度规定 */
1132     if (copy > seglen)
1133         copy = seglen;
1134
1135     /* Where to copy to? */
//返回由 sk_stream_alloc()在 sk_buff 末尾分配的空间。即 sk_buff 的线性存储区底部是否还有空间
1136     if (skb_availroom(skb) > 0) {
1137         /* We have some space in skb head. Superb! */
1138         copy = min_t(int, copy, skb_availroom(skb));
1139         err = skb_add_data_nocache(sk, skb, from, copy); //如果还有，则将数据由用户空间进行复制到 skb 中。
1140         if (err)
1141             goto do_fault;
1142     } else { //如果没有空间，则将数据复制到 scatter/gather IO 类型的页中。
1143         bool merge = true; //标识最后一个页中是否有数据。
1144         int i = skb_shinfo(skb)->nfrags; //获取分片的数量
1145         struct page_frag *pfrag = sk_page_frag(sk); //获得 cache 中的分片页
/*在分片列表(frags)中使用原有分片(返回相应分片的指针)或分配新页来存放数据，如果不成功则等待存储空间*/
1147         if (!sk_page_frag_refill(sk, pfrag))
1148             goto wait_for_memory;
1149
1150     * 如果传输控制块(sock)中的缓存页 pfrag，不是当前 skb->shared_info 中的最后一个分片(分散聚集 IO 页面)所在的页面，则
直接使用该页面，
* 将其添加 到分片列表(分散聚集 IO 页面数组)中，否则说明传输控制块(sock)中的缓存页 pfrag 就是分散聚集 IO 页面的最后
一个页面，则直接向其中拷贝数据即可。
1151     if (!skb_can_coalesce(skb, i, pfrag->page,
1152                           pfrag->offset)) {
1153         if (i == MAX_SKB_FRAGS || !sg) {
1154             tcp_mark_push(tp, skb);
1155             goto new_segment;
1156         }
1157         merge = false;
1158     }
1159
1160     copy = min_t(int, copy, pfrag->size - pfrag->offset);
/*拷贝数据至 skb 中非线性区分片(分散聚集 IO 页面)中*/
1161     if (!sk_wmem_schedule(sk, copy))
1162         goto wait_for_memory;
1163
1164     err = skb_copy_to_page_nocache(sk, from, skb,
1165                                     pfrag->page,
1166                                     pfrag->offset,
1167                                     copy);
1168     if (err)
1169         goto do_error;
1170
1171     /* Update the skb. */
1172     if (merge) {

```

```

    /*增加分片大小*/
1173         skb_frag_size_add(&skb_shinfo(skb)->frags[i - 1], copy);
1174     } else {
/*如果是复制到一个全新的页面分段中，则需要更新的有关分段信息就会多一些，如分段数据的长度、页内偏移、分段数量等。调用 skb_fill_page_desc()来完成。如果标识最近一次分配页面的 sk_sndmsg_page 不为空，则增加对该页面的引用。否则说明复制了数据的页面是新分配的，且没有使用完，在增加对该页面的引用的同时，还需更新 sk_sndmsg_page 的值。如果新分配的页面已使用完，就无须更新 sk_sndmsg_page 的值了，因为如果 SKB 未超过段上限，那么下次必定还会分配新的页面，因此在此处就省去了对 off+copy=PAGE_SIZE 这条分支的处理。*/
1175         skb_fill_page_desc(skb, i, pfrag->page,
1176                           pfrag->offset, copy);
1177         get_page(pfrag->page);
1178     }
1179     pfrag->offset += copy;
1180 }
/*如果复制的数据长度为零，则取消 TCPHDR_PSH 标志，将意味着数据不会被发送*/
1182     if (!copied)
1183         TCP_SKB_CB(skb)->tcp_flags &= ~TCPHDR_PSH;
/*更新发送队列中的最后一个序号 write_seq，以及数据包的最后一个序列 end_seq，初始化 gso 分段数 gso_segs。*/
1185     tp->write_seq += copy;
1186     TCP_SKB_CB(skb)->end_seq += copy;
1187     skb_shinfo(skb)->gso_segs = 0;
/*更新指向数据源的指针和已复制字节数。*/
1189     from += copy;
1190     copied += copy;
/*如果所有数据已全部复制到 SKB 中，则跳转到 out 处理。*/
1191     if ((seglen == copy) == 0 && iovlen == 0)
1192         goto out;
/*如果当前 SKB 中的数据小于 max，说明还可以往里填充数据，或者发送的是带外数据(MSG_OOB，紧急)，则跳过以下发送过程，继续复制数据到 SKB*/
1194     if (skb->len < max || (flags & MSG_OOB) || unlikely(tp->repair))
1195         continue;

/*检查是否必须立即发送，即检查自上次发送后产生的数据是否已超过对方曾经通告过的最大窗口值的一半。如果必须立即发送，则设置 TCPHDR_PSH 标志后调用 __tcp_push_pending_frames()，在发送队列上从 sk_send_head 开始把 SKB 发送出去。*/
1197     if (forced_push(tp)) {
1198         tcp_mark_push(tp, skb);
1199         __tcp_push_pending_frames(sk, mss_now, TCP_NAGLE_PUSH);
1200     } else if (skb == tcp_send_head(sk))
/*如果没有必要立即发送，且发送队列上只存在这个段，则调用 tcp_push_one()只发送当前段。*/
1201         tcp_push_one(sk, mss_now);
1202     continue;
/*套接口的发送缓存是有大小限制的，当发送队列中的数据段总长度超过发送缓冲区的长度上限时，就不能再分配 SKB 了，只能等待。设置 SOCK_NOSPACE 标志，表示套接口发送缓冲区已满。*/
1204 wait_for_sndbuf:
1205     set_bit(SOCK_NOSPACE, &sk->sk_socket->flags);
/*跳到这里，意味着内存分配失败*/
1206 wait_for_memory:
/*虽然分配 SKB 失败，但是如果之前有数据从用户空间复制过来，则调用 tcp_push()将其发送出去。
其中第三个参数中去掉 MSG_MORE 标志，表示本次发送没有更多的数据了。
因为分配 SKB 失败，因此可以加上 TCPHDR_PSH 标志，第五个参数使用 nagle 算法，可能会推迟发送。*/
1207     if (copied)
1208         tcp_push(sk, flags & ~MSG_MORE, mss_now, TCP_NAGLE_PUSH);
/*调用 sk_stream_wait_memory()进入睡眠，等待内存空闲的信号，如果在超时时间内没有得到该信号，则跳转到 do_error 处执行。*/
1210     if ((err = sk_stream_wait_memory(sk, &timeo)) != 0)
1211         goto do_error;

```

```

/*等待内存未超时，有空闲内存可用。睡眠后，MSS 有可能发生了变化，所以重新获取当前的 MSS 和 TSO 分段段长，然后继续循环复制数据。*/
1213         mss_now = tcp_send_mss(sk, &size_goal, flags);
1214     }
1215 }
/*发送过程中正常的退出。*/
1217 out:
1218     if (copied)
/*如果已有复制的数据，则调用 tcp_push()将其发送出去，是否立即发送取决于 nagle 算法。*/
1219         tcp_push(sk, flags, mss_now, tp->nonagle);
1220     release_sock(sk);
1221     return copied + copied_syn;
/*在复制数据异常时进入到这里。*/
1223 do_fault:
1224     if (!skb->len) {
1225         tcp_unlink_write_queue(skb, sk);
1226         /* It is the one place in all of TCP, except connection
1227          * reset, where we can be unlinking the send_head.
1228          */
1229         tcp_check_send_head(sk, skb);
1230         sk_wmem_free_skb(sk, skb);
1231     }
1232 /*如果已复制了部分数据，那么即使发生了错误，也可以发送数据包，因此跳转到 out 处*/
1233 do_error:
1234     if (copied + copied_syn)
1235         goto out;
/*如果没有复制数据，则调用 sk_stream_error()来获取错误码。然后对传输层控制块解锁后返回错误码。*/
1236 out_err:
1237     err = sk_stream_error(sk, flags, err);
1238     release_sock(sk);
1239     return err;
1240 }

```

第 1220 行和 1222 行都会调用 `tcp_write_xmit()` 发送数据。这里走 1222 行的路线

```

void tcp_push_one(struct sock *sk, unsigned int mss_now)
{
    struct sk_buff *skb = tcp_send_head(sk);

    tcp_write_xmit(sk, mss_now, TCP_NAGLE_PUSH, 1, sk->sk_allocation);
}

```

`tcp_write_xmit` 的参数意义如下：

`sk`: 源于应用程序，其使用 `AF_INET` 创建的 `socket` 套接字

`mss_now`: maximum segment size，参考 `tso` 和 `gso` 后给出的值。

`TCP_NAGLE_PUSH`: `nagle` 算法标志，起初用于解决拥塞；该标志表示当一个数据段不足 `MSS` 时，其会推迟这个数据段的发送，直到数据填充完一个 `MSS`。

`push_one`: 标志要发送的 `packet` 数，当其>0 时，会确保至少发送一个 `packet`。

`gfp`: 分配内存时的标志

```

1811 static bool tcp_write_xmit(struct sock *sk, unsigned int mss_now, int nonagle,
1812                             int push_one, gfp_t gfp)
1813 {
1814     struct tcp_sock *tp = tcp_sk(sk);
1815     struct sk_buff *skb;
1816     unsigned int tso_segs, sent_pkts;

```

```

1817     int cwnd_quota;
1818     int result;
1819
1820     sent_pkts = 0;      //记录发送的数据包的个数，初始值为 0
1821     //push_one 用于记录要发送报文的个数，对于只发送一个报文则跳过 if 语句。
1822     if (!push_one) {
1823         /* Do MTU probing. */
1824         result = tcp_mtu_probe(sk); //要发送多个的话，会检查 MTU 值，这个 MTU 值会影响分片操作。
1825         if (!result) {
1826             return false;
1827         } else if (result > 0) {
1828             sent_pkts = 1;
1829         }
1830     }
1831
1832     while ((skb = tcp_send_head(sk))) { //while 循环用于检测发送队列头部是否有 sk_buffer 需要发送。
1833         unsigned int limit;
1834
1835         //tso/gso 字段用于在数据传递到 IP 层之前进行分片。
1836         tso_segs = tcp_init_tso_segs(sk, skb, mss_now);
1837         BUG_ON(!tso_segs);
1838
1839         if (unlikely(tp->repair) && tp->repair_queue == TCP_SEND_QUEUE)
1840             goto repair; /* Skip network transmission */
1841
1842         cwnd_quota = tcp_cwnd_test(tp, skb);
1843         if (!cwnd_quota) { //如果不能在发送了
1844             if (push_one == 2)
1845                 /* Force out a loss probe pkt. */
1846                 cwnd_quota = 1; //如果设置 push_one == 2, 那么强制 cwnd_quota =1, 后面会发送至少一个数据包。
1847             else
1848                 break; //跳出循环，不发送数据了。
1849         }
1850
1851         if (unlikely(!tcp_snd_wnd_test(tp, skb, mss_now))) //检测是否 SKB 至少第一个 segment 位于发送窗口中，否，则跳出 while 循环。
1852             break;
1853
1854         if (tso_segs == 1) { //对于无 tso/gso 情况，即未分片。
1855             if (unlikely(!tcp_nagle_test(tp, skb, mss_now, //unlikely 预示着 tcp_nagle_test 返回值是 true, true 的
1856                                         (tcp_skb_is_last(sk, skb) ?
1857                                         nonagle : TCP_NAGLE_PUSH)))) //返回值意味着 Nagle 拥塞控制算法允许发送该数据。
1858                 break;
1859         } else {
1860             if (!push_one && tcp_tso_should_defer(sk, skb)) //多个 skb 时，需要计算是否延迟发送
1861                 break;
1862         }
1863
1864         /* TSQ : sk_wmem_alloc accounts skb truesize,
1865          * including skb overhead. But thats OK.
1866          */
1867         if (atomic_read(&sk->sk_wmem_alloc) >= sysctl_tcp_limit_output_bytes) { //发送数据是否大于用户使用 sysctl
1868             //设置/proc 输出允许的最大字节数？
1869             set_bit(TSQ_THROTTLED, &tp->tsq_flags);
1870             break;
1871     }

```

```

1871     limit = mss_now;
1872     if (tso_segs > 1 && !tcp_urg_mode(tp)) //存在分片，且非 urgent 模式，使用 MSS 计算发送数据的限制。
1873         limit = tcp_mss_split_point(sk, skb, mss_now,
1874             min_t(unsigned int,
1875                 cwnd_quota,
1876                 sk->sk_gso_max_segs));
1877
1878     if (skb->len > limit &&
1879         unlikely(tso_fragment(sk, skb, limit, mss_now, gfp))) //如果发送数据大于上面计算的 limit 值，其启用
tso_fragment()进行分片操作。
1880     break;
/* 以上 6 行：根据条件，可能需要对 SKB 中的报文进行分段处理，分段的报文包括两种：
 * 一种是普通的用 MSS 分段的报文，另一种则是 TSO 分段的报文。
能否发送报文主要取决于两个条件：一是报文需完全在发送窗口中，而是拥塞窗口未满。
第一种报文，应该不会再分段了，因为在 tcp_sendmsg()中创建报文的 SKB 时已经根据 MSS 处理了，而第二种报文，则
一般情况下都会大于 MSS，因为通过 TSO 分段的段有可能大于拥塞窗口的剩余空间，如果是这样，就需要以发送窗口和拥塞
窗口的最小值作为段长对报文再次分段。*/
1881
1882     TCP_SKB_CB(skb)->when = tcp_time_stamp; //时间戳更新。
1883
1884     if (unlikely(tcp_transmit_skb(sk, skb, 1, gfp))) //发送数据
1885     break;
1886
1887 repair:
1888     /* Advance the send_head. This one is sent out.
1889      * This call will increment packets_out.
1890      */
1891     tcp_event_new_data_sent(sk, skb); //更新统计，启动重传定时器。
1892
1893     tcp_minshall_update(tp, mss_now, skb); //更新 struct tcp_sock 中的 snd_sml 字段，用于表示是否启用 Nagle 算法。
1894     sent_pkts += tcp_skb_pcount(skb);
...
1911 }

```

第 1884 行注释的比较简单，该函数将构建数据的 tcp 头，并将其由 tcp 传递到 Ip 层，
函数定义同样位于 `tcp_output.c` 文件中。该函数参数的意义；

`sk`：关联到应用程序创建的套接字。

`sk_buff`：存放的是发送数据信息

`clone_it`：是否 `clone` 传递进来的数据报。上面设置了 1，为传递。

`gfp_mask`：申请内存的标志

```

828 static int tcp_transmit_skb(struct sock *sk, struct sk_buff *skb, int clone_it,
829                             gfp_t gfp_mask)
830 {
831     const struct inet_connection_sock *icsk = inet_csk(sk);
832     struct inet_sock *inet;
833     struct tcp_sock *tp;
834     struct tcp_skb_cb *tcb;
835     struct tcp_out_options opts;
836     unsigned int tcp_options_size, tcp_header_size;
837     struct tcp_md5sig_key *md5;
838     struct tcphdr *th;
...
843     /* 如果拥塞控制需要时间戳，则必须在复制前获得时间戳；
844      * 并不是所有拥塞算法都会用到时间戳，TCP_CONG_RTT_STAMP，高精度 RTT，Round-Trip Time，传输延迟

```

```

845     */
846     if (icsk->icsk_ca_ops->flags & TCP_CONG_RTT_STAMP)
847         __net_timestamp(skb);
848
849     if (likely(clone_it)) { //传递进来的参数指定是否需要复制报文
850         const struct sk_buff *fclone = skb + 1;
851
852         if (unlikely(skb->fclone == SKB_FCLONE_ORIG &&
853                     fclone->fclone == SKB_FCLONE_CLONE))
854             NET_INC_STATS_BH(sock_net(sk),
855                             LINUX_MIB_TCPSPURIOUS_RTX_HOSTQUEUES);
856
857         if (unlikely(skb_cloned(skb)))
858             skb = pskb_copy(skb, gfp_mask); //执行 copy 操作
859         else
860             skb = skb_clone(skb, gfp_mask); //执行 clone 操作
861         if (unlikely(!skb))
862             return -ENOBUFS;
863     }
864     /*获取 INET 层和 TCP 层的传输控制块、skb 中的 TCP 私有数据块。*/
865     inet = inet_sk(skb);
866     tp = tcp_sk(skb);
867     tcb = TCP_SKB_CB(skb);
868     memset(&opts, 0, sizeof(opts));
869
/*根据 TCP 选项重新调整 TCP 首部的长度。*/
/*判断当前 TCP 报文是否是 SYN 段，因为有些选项只能出现在 SYN 报文中，需做特别处理。*/
870     if (unlikely(tcb->tcp_flags & TCPHDR_SYN))
871         tcp_options_size = tcp_syn_options(sk, skb, &opts, &md5);
872     else
873         tcp_options_size = tcp_established_options(sk, skb, &opts,
874                                         &md5);
/*tcp 首部的总长度等于可选长度加上 struct tcphdr。*/
875     tcp_header_size = tcp_options_size + sizeof(struct tcphdr);
876
/*如果已发出但未确认的数据包数目为零，则只初始化拥塞控制，并开始跟踪该连接的 RTT。*/
877     if (tcp_packets_in_flight(tp) == 0)
878         tcp_ca_event(sk, CA_EVENT_TX_START);
879
880     /* if no packet is in qdisc/device queue, then allow XPS to select
881      * another queue.
882     */
883     skb->ooo_okay = sk_wmem_alloc_get(sk) == 0;
884 /*调用 skb_push()在数据部分的头部添加 TCP 首部，长度即为之前计算得到的那个 tcp_header_size，实际上是把 data 指针往上移。*/
885     skb_push(skb, tcp_header_size);
886     skb_reset_transport_header(skb);
887
888     skb_orphan(skb); //孤儿一个 Buffer，如果 skb 有解析函数，则会调用它自带的解析函数，孤儿并不意味着 buffer 不存在，意味着不再为先前所有者所有，意味着可以修改一些字段//
//更新 sk_buff 类型的 skb 的相关字段
889     skb->sk = sk;
890     skb->destructor = (sysctl_tcp_limit_output_bytes > 0) ?
891         tcp_wfree : sock_wfree;
892     atomic_add(skb->truesize, &sk->sk_wmem_alloc);
893
894     /* Build TCP header and checksum it.构建 tcp 头并做校验 */

```

```

895     th = tcp_hdr(skb);
896     th->source      = inet->inet_sport;
897     th->dest        = inet->inet_dport;
898     th->seq         = htonl(tcb->seq);
899     th->ack_seq     = htonl(tp->rcv_nxt);
900     *((__be16 *)th) + 6) = htons(((tcp_header_size >> 2) << 12) |
901                               tcb->tcp_flags);
902 /*分两种情况设置 TCP 首部的接收窗口的大小*/
903     if (unlikely(tcb->tcp_flags & TCPHDR_SYN)) {
904         /* RFC1323: The window in SYN & SYN/ACK segments
905          * is never scaled.
906         */
907         th->window = htons(min(tp->rcv_wnd, 65535U));
908     } else {
909         th->window = htons(tcp_select_window(sk));
910     }
911 /*初始化 TCP 首部的校验码和紧急指针，可选字段，具体请参考 TCP 协议中的首部定义。*/
912     th->check      = 0;
913     th->urg_ptr    = 0;
914     /* The urg_mode check is necessary during a below snd_una win probe, 紧急指针 */
915     if (unlikely(tcp_urg_mode(tp) && before(tcb->seq, tp->snd_up))) {
916         if (before(tp->snd_up, tcb->seq + 0x10000)) {
917             th->urg_ptr = htons(tp->snd_up - tcb->seq);
918             th->urg = 1;
919         } else if (after(tcb->seq + 0xFFFF, tp->snd_nxt)) {
920             th->urg_ptr = htons(0xFFFF);
921             th->urg = 1;
922         }
923     }
924     tcp_options_write((__be32 *)th + 1, tp, &opts); //可选字段
925     if (likely((tcb->tcp_flags & TCPHDR_SYN) == 0))
926 /*ENC (explicit congestion notification), 基于显示反馈的拥塞控制协议，此外还有 DCA(delay-based congestion avoidance),
927 基于路径延迟，LCA(loss-based congestion avoidance), 基于丢包反馈。
TCP_ECN_send: 为已建立连接的套接字的即将发送的 packet 设置该 ECN 标志
*/
928     TCP_ECN_send(sk, skb, tcp_header_size);
929 /*MD5 (Message Digest Algorithm) 算法，计算机安全领域广泛使用的一种散列函数，用以提供消息的完整性保护。
*/
930 #ifdef CONFIG_TCP_MD5SIG
931     /* Calculate the MD5 hash, as we have all we need now */
932     if (md5) {
933         sk_nocaps_add(sk, NETIF_F_GSO_MASK);
934         tp->af_specific->calc_md5_hash(opts.hash_location,
935                                         md5, sk, NULL, skb);
936     }
937     icsk->icsk_af_ops->send_check(sk, skb);
938     if (likely(tcb->tcp_flags & TCPHDR_ACK))
939         tcp_event_ack_sent(sk, tcp_skb_pcount(skb));
940     if (skb->len != tcp_header_size)

```

```
944     tcp_event_data_sent(tp, sk);
945
946     if (after(tcb->end_seq, tp->snd_nxt) || tcb->seq == tcb->end_seq)
947         TCP_ADD_STATS(sock_net(sk), TCP_MIB_OUTSEGS,
948                         tcp_skb_pcount(skb));
949
/*调用发送接口 queue_xmit 发送报文，进入到 ip 层，如果失败返回错误码。在 TCP 中该接口实现函数为 ip_queue_xmit()*/
950     err = icsk->icsk_af_ops->queue_xmit(skb, &inet->cork.fl);
951     if (likely(err <= 0))
952         return err;
953
954     top_enter_cwr(sk, 1);
955
956     return net_xmit_eval(err);
957 }
```

ip_queue_xmit 参考网络层发送侧程序。最后将程序的流梳理一遍~！

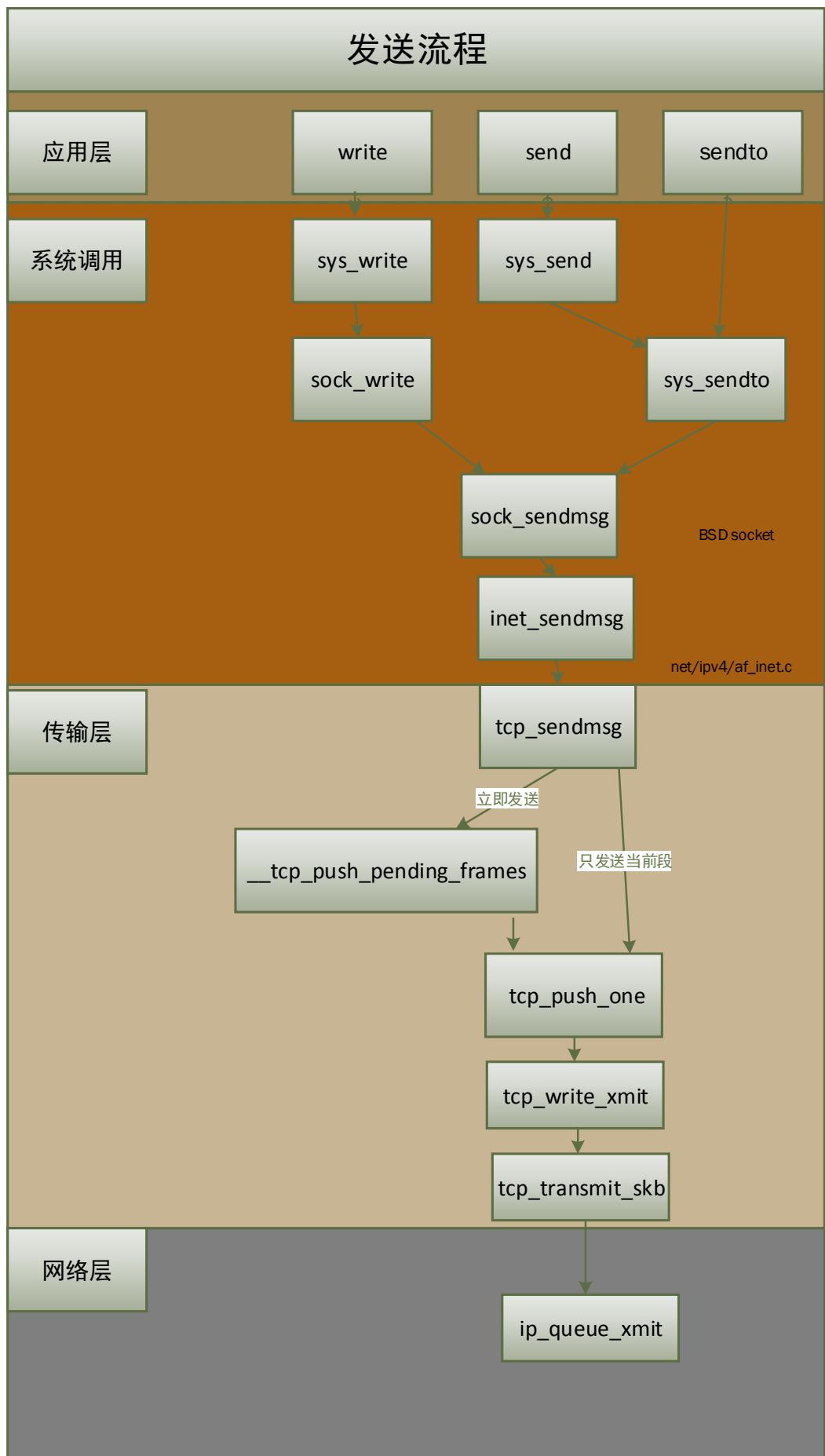


图 7.2 tcp 发送函数调用流程

7.2 MSS

Tcp_output.c

tp->rx_opt.user_mss 是用户使用 TCP_MAXSEG 选项调用 setsockopt 函数设置;
tp->rx_opt.mss_clamp 是 tcp 连接建立时协商的 MSS, 其实 user_mss 和 SYN 包收到的 mss 中较小的那个值。

inet_csk(sk)->icsk_pmtu_cookie 是近的 PMTU

tp->mss_cache 当前有效的发送 MSS, 包括出 SACKs 外的所有选项, 会参考当前的 pmtu, 但是不会超过 tp->rx_opt.mss_clamp。

inet_csk(sk)->icsk_pmtu_cookie and tp->mss_cache 只在 tcp_sync_mss 可写, 其它函数只读。

MSS 最大值的计算是: PMTU-IP 头-TCP 头-IP 选项-TCP 选项。

Net/ipv4/tcp_output.c

```
1296 unsigned int tcp_sync_mss(struct sock *sk, u32 pmtu)
1297 {
1298     struct tcp_sock *tp = tcp_sk(sk);
1299     struct inet_connection_sock *icsk = inet_csk(sk);
1300     int mss_now;
1301
1302     if (icsk->icsk_mtup.search_high > pmtu)
1303         icsk->icsk_mtup.search_high = pmtu;
1304
1305     mss_now = tcp_mtu_to_mss(sk, pmtu); //根据 MTU (路径最大传输单元) 获得当前最大的 MSS。
//根据对端设备的最大窗口将发送设备的窗口设置成一半大小的, 窗口的最小值是 68U-TCP 头
1306     mss_now = tcp_bound_to_half_wnd(tp, mss_now);
1307
1308     /* And store cached results */
1309     icsk->icsk_pmtu_cookie = pmtu; //保存最近一次 PMTU (path maximum transmit unit)
//是否使能了最大传输单元控制? 如果是, 则选取设置值和当前得到的最大传输单元中较小的值作为 mtu
1310     if (icsk->icsk_mtup.enabled)
1311         mss_now = min(mss_now, tcp_mtu_to_mss(sk, icsk->icsk_mtup.search_low));
//缓存 mss
1312     tp->mss_cache = mss_now;
1313
1314     return mss_now;
1315 }
```

第八章 tcp 接（传输层）

在 IP 层的接收，提到其调用的函数是 int `tcp_v4_rcv(struct sk_buff *skb)`，其参数是包含数据信息的 `sk_buff`。这个函数定义于：

```
net/ipv4/tcp_ipv4.c
1961 int tcp_v4_rcv(struct sk_buff *skb)
1962 {
1963     const struct iphdr *iph;           //ip 层头标识字段
1964     const struct tcphdr *th;          //tcp 层头标识字段
1965     struct sock *sk;
1966     int ret;
1967     struct net *net = dev_net(skb->dev); //命名空间会使用到该结构体
//PACKET_HOST 指示该数据包目的是本机，类似的还有 PACKET_BROADCAST 等
1969     if (skb->pkt_type != PACKET_HOST)
1970         goto discard_it;
1971
1972     /* Count it even if it's bad */
1973     TCP_INC_STATS(net, TCP_MIB_INSEGS); // 统计信息更新
1974
/*判断 segment 的头长度是否合法，有三种情况：
1、sizeof(struct tcphdr) <= skb 的 tcp 头长度；则满足 segment 的 tcp 头长度要求。通常都会满足该要求。
2、sizeof(struct tcphdr) > skb 头+数据长度之和；出错丢弃该包。
3、介于 1、2 两种情况之间的长度，这种情况在数据包分片时会遇到。
*/
1975     if (!pskb_may_pull(skb, sizeof(struct tcphdr)))
1976         goto discard_it;
1977
/*获得传输层头*/
1978     th = tcp_hdr(skb);
1979
/*再一次判断数据包的合法性，doff 是 tcp 包中记录的该 packet 的首部长度，*/
1980     if (th->doff < sizeof(struct tcphdr) / 4)
1981         goto bad_packet;
1982     if (!pskb_may_pull(skb, th->doff * 4))
1983         goto discard_it;
1984
1985     /* An explanation is required here, I think.
1986      * Packet length and doff are validated by header prediction,
1987      * provided case of th->doff==0 is eliminated.
1988      * So, we defer the checks. */
1989
/*skb->csum:存放硬件或者软件计算的 payload 的 checksum 不包括伪头，但是是否有意义由 skb->ip_summed 的值决定。
CHECKSUM_UNNECESSARY 表示网卡或者协议栈已经计算和验证了 L4 层的头和校验值。也就是计算了 tcp udp 的伪头。
CHECKSUM_COMPLETE 表示硬件进行了计算，计算结果存储在 skb->csum 中。
如果硬件完成了 tcp 的头校验，那么 skb->ip_summed & CHECKSUM_UNNECESSARY，的值等于 1，这就是
skb_csum_unnecessary 的返回值，否则调用 tcp_v4_checksum_init 判断硬件是否有该特性，如果有则使用硬件计算，如果没有只能多消耗点 cpu 完成计算了。
*/
1990     if (!skb_csum_unnecessary(skb) && tcp_v4_checksum_init(skb))
1991         goto csum_error;
1992
/*这里再一次获得 tcp 的头，由于 tcp_hdr 返回的是指针，并且在计算校验和时可能更改过地址，所以这里再一次赋值*/
1993     th = tcp_hdr(skb);
1994     iph = ip_hdr(skb);
/*TCP 控制块初始化，这里初始化使用的都是 tcp 头中字段信息*/
1995     TCP_SKB_CB(skb)->seq = ntohl(th->seq);
```

```

1995     TCP_SKB_CB(skb)->end_seq = (TCP_SKB_CB(skb)->seq + th->syn + th->fin +
1996             skb->len - th->doff * 4);
1997     TCP_SKB_CB(skb)->ack_seq = ntohs(th->ack_seq);
1998     TCP_SKB_CB(skb)->when    = 0;
1999     TCP_SKB_CB(skb)->ip_dsfield = ipv4_get_dsfield(iph);
2000     TCP_SKB_CB(skb)->sacked  = 0;
2001
2002     /*查找对应的 skb, 也就是用户空间使用 socket()创建的套接字在内核中的对应的 sock*/
2003     sk = __inet_lookup_skb(&tcp_hashinfo, skb, th->source, th->dest);
2004     if (!sk)
2005         goto no_tcp_socket;
2006
2007     /*如果这是主动关闭了套接字, 则套接字的状态就是 TCP_TIME_WAIT, 被动的一端状态是 TCP_CLOSE*/
2008     if (sk->sk_state == TCP_TIME_WAIT)
2009         goto do_time_wait;
2010
2011     /*查看其生存时间 TTL(time to live)是否超时, 如果超时统计一下然后丢弃*/
2012     if (unlikely(iph->ttl < inet_sk(sk)->min_ttl)) {
2013         NET_INC_STATS_BH(net, LINUX_MIB_TCPMINTTLDROP);
2014         goto discard_and_reuse;
2015     }
2016
2017     /*当内核定义了 CONFIG_XFRM 变量时, 会启用 xfrm 安全检查, 对于不满足安检的丢弃.*/
2018     if (!xfrm4_policy_check(sk, XFRM_POLICY_IN, skb))
2019         goto discard_and_reuse;
2020
2021     /*和桥接防火墙相关的, 如果配置实用桥接防火墙, 则 skb 中的 nfct、nfct_reasm、nf_bridge 字段会被使用到*/
2022     nf_reset(skb);
2023
2024     /*套接字过滤器, 如果 pkt_len 是 0, 则要丢弃该包, 如果 skb->len 小于 pkt_len 则要接收该数据包, 继续往下走到 2022 行.*/
2025     if (sk_filter(sk, skb))
2026         goto discard_and_reuse;
2027
2028     /*dev 是 net_device 结构体, 在一文中对该结构体已经详细描述过了*/
2029     skb->dev = NULL;
2030
2031     /*获得套接字的嵌套自旋锁*/
2032     bh_lock_sock_nested(sk);
2033     ret = 0;
2034
2035     /*sock_owned_by_user 会判断(sk)->sk_lock.owned, 该字段用于指示 sock 目前是否处于用户空间, 在 BH 或者中断情况下是没有被这个函数(进程)own 的, 之一的文章中的 NAPI 就会启用 BH, BH 中的数据是没有被用户对应的进程拥有*/
2036     if (!sock_owned_by_user(sk)) {
2037
2038         /*没有被 own 的情况处理, 如果是支持 DMA 操作的, 则 net_dma_find_channel 获得 DMA 通道号, 然后调用 tcp_v4_do_rcv 进行实际的数据接收*/
2039 #ifdef CONFIG_NET_DMA
2040         struct tcp_sock *tp = tcp_sk(sk);
2041         if (!tp->ucopy.dma_chan && tp->ucopy.pinned_list)
2042             tp->ucopy.dma_chan = net_dma_find_channel();
2043         if (tp->ucopy.dma_chan)
2044             ret = tcp_v4_do_rcv(sk, skb);
2045         else
2046             ret = -EOPNOTSUPP;
2047     #endif
2048
2049         if (!tcp_pqueue(sk, skb))
2050             ret = tcp_v4_do_rcv(sk, skb);
2051     }
2052
2053     /*已经被 own 的情况处理, 需要持有 per-socket 锁, sk_add_backlog 注释见后面*/
2054     } else if (unlikely(sk_add_backlog(sk, skb,

```

```

2040             sk->sk_rcvbuf + sk->sk_sndbuf))) {
2041         bh_unlock_sock(sk);
2042         NET_INC_STATS_BH(net, LINUX_MIB_TCPBACKLOGDROP);
2043         goto discard_and_reuse;
2044     }
2045     /*释放前面获得的锁*/
2046     bh_unlock_sock(sk);
2047     sock_put(sk);
2048
2049     return ret;
2110 }

```

该函数调用 `sk_add_backlog()` 接收数据包，其位于 `include/linux/sock.h`；

`backlog` 队列比较特殊，需要持有锁才能操作该队列，为了降低延迟，使用了一个技巧 `rmem_alloc` 用于 64 为体系结构的一个填充，和 `backlog` 本身没有关系。

```

750 /* OOB backlog add */
751 static inline void __sk_add_backlog(struct sock *sk, struct sk_buff *skb)
752 {
753     /* dont let skb dst not refcounted, we are going to leave rcu lock */
754     skb_dst_force(skb);
755
756     /*将 skb 添加到 backlog 上，原链表为空添加方法*/
757     if (!sk->sk_backlog.tail)
758         sk->sk_backlog.head = skb;
759     else
760         /*原链表非空的添加方法*/
761         sk->sk_backlog.tail->next = skb;
762     sk->sk_backlog.tail = skb;
763     skb->next = NULL;
764 }

770 static inline bool sk_rcvqueues_full(const struct sock *sk, const struct sk_buff *skb,
771                                     unsigned int limit)
772 {
773     /* sk->sk_backlog.len 为 backlog 队列长度，sk_rmem_alloc 是上面提到的填充字段，qsize 是发送队列和接收队列的长度之和 */
774     unsigned int qsize = sk->sk_backlog.len + atomic_read(&sk->sk_rmem_alloc);
775
776     return qsize > limit;
777 }
778 /* The per-socket spinlock must be held here. */
779 static inline __must_check int sk_add_backlog(struct sock *sk, struct sk_buff *skb,
780                                               unsigned int limit)
781 {
782     /*队列长度不合理，则执行 783 行*/
783     if (sk_rcvqueues_full(sk, skb, limit))
784         return -ENOBUFS;
785     __sk_add_backlog(sk, skb);
786     sk->sk_backlog.len += skb->truesize;
787     return 0;
788 }

```

到这里没有继续调用函数了，只看到数据放在了 backlog 上就没有继续显示调用其它函数进行进一步的接收工作。

接下来从上至下看，在第五章中提到 tcp_recvmsg 完成将数据由应用程序真正送入传输层的工作。

```
1560 int tcp_recvmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
1561         size_t len, int nonblock, int flags, int *addr_len)
1562 {
/*判断 sock 的状态，TCP_LISTEN 用于被动接收*/
1579     if (sk->sk_state == TCP_LISTEN)
1580         goto out;
1581
/*应用设置的超时值在此设置，若未设置，则超时功能不启用*/
1582     timeo = sock_rcvtimeo(sk, nonblock);
1583
1584     /* 紧急数据处理，OOB 是 out of band，带外信号，规范中使用带外传输紧急和重要的数据*/
1585     if (flags & MSG_OOB)
1586         goto recv_urg;
/*//待拷贝的下一个序列号*/
1603     seq = &tp->copied_seq;
/*MSG_PEEK 预读标志，MSG_PEEK 标志会将套接字接收队列中的可读的数据拷贝到缓冲区，但不会使套接字接收队列中的
数据减少，*/
下一次调用 recv 函数
//仍然能够读到相同数据
*/
1604     if (flags & MSG_PEEK) {
1605         peek_seq = tp->copied_seq;
1606         seq = &peek_seq;
1607     }
/*应用程序设置 MSG_WAITALL 时，target 等于 len 用户空间要接收的数据长度，否则置为 1*/
1609     target = sock_rcvlowat(sk, flags & MSG_WAITALL, len);
1633     do {
1634         u32 offset;
1648         skb_queue_walk(&sk->sk_receive_queue, skb) {
/*已经接收到的 segment 和当前接收到的 segment 的序列号的差值*/
//如果用户的缓冲区(即用户 malloc 的 buf)长度够大，offset 一般是 0。
//即“下次准备拷贝数据的序列号”==此时获取报文的起始序列号
//什么情况下 offset >0 呢？很简答，如果用户缓冲区 12 字节，而这个 skb 有 120 字节
//那么一次 recv 系统调用，只能获取 skb 中的前 12 个字节，下一次执行 recv 系统调用
//offset 就是 12 了，表示从第 12 个字节开始读取数据，前 12 个字节已经读取了。
//那这个“已经读取 12 字节”这个消息，存在哪呢？
//在*seq = &tp->copied_seq;中
1658         offset = *seq - TCP_SKB_CB(skb)->seq;
1659         if (tcp_hdr(skb)->syn)
1660             offset--;
1661         if (offset < skb->len)
1662             goto found_ok_skb;
1663         if (tcp_hdr(skb)->fin)
1664             goto found_fin_ok;
1668     }
//执行到了这里，表明小循环中 break 了，既然 break 了，说明 sk_receive_queue 中
//已经没有 skb 可以读取了
//如果没有执行到这里说明前面的小循环中执行了 goto，读到有用的 skb，或者读到 fin 都会 goto。
//没有 skb 可以读取，说明什么？
//可能性 1：当用户第一次调用 recv 时，压根没有数据到来
//可能性 2：skb->len 一共 20 字节，假设用户调用一次 recv，读取 12 字节，再调用 recv，
```

```

//读取 12 字节，此时 skb 由于上次已经被读取了 12 字节，只剩下 8 字节。
//于是代码的逻辑上，再会要求获取 skb，来读取剩下的 8 字节。

//可能性 1 的情况下，copied == 0，肯定不会进这个 if。后续将执行休眠
//可能性 2 的情况下，情况比较复杂。可能性 2 表明数据没有读够用户想要的 len 长度
//虽然进程上下文中，没有读够数据，但是可能我们在读数据的时候
//软中断把数据放到 backlog 队列中了，而 backlog 对队列中的数据或许恰好让我们读够数
//据。

//copied 读了数据的，copied 肯定>=1，而 target 是 1 或者 len
//copied 只能取 0(可能性 1)，或者 0~len(可能性 2)
//copied >= target 表示我们取得我们想要的数据了，何必进行休眠，直接 return
//如果 copied 没有达到我们想要的数据，则看看 sk_backlog 是否为空
//空的话，尽力了，只能尝试休眠
//非空的话，还有一线希望，我们去 sk_backlog 找找数据，看看是否能够达到我们想要的
//数据大小

//我觉得 copied == target 是会出现的，但是出现的话，也不会进现在这个流程
//说明情况下 copied == target
1670     /* Well, if we have backlog, try to process it now yet. */
1671
1672     if (copied >= target && !sk->sk_backlog.tail)
1673         break;
1674
1675     if (copied) {
//可能性 2，拷贝了数据，但是没有拷贝到指定大小
1676         if (sk->sk_err ||
1677             sk->sk_state == TCP_CLOSE ||
1678             (sk->sk_shutdown & RCV_SHUTDOWN) ||
1679             !timeo ||
1680             signal_pending(current))
1681         break;
1682     } else {
//可能性 1
1683         if (sock_flag(sk, SOCK_DONE))
1684             break;
1685
1686         if (sk->sk_err) {
1687             copied = sock_error(sk);
1688             break;
1689         }
1690
1691         if (sk->sk_shutdown & RCV_SHUTDOWN)
1692             break;
1693
1694         if (sk->sk_state == TCP_CLOSE) {
1695             if (!sock_flag(sk, SOCK_DONE)) {
1696                 /* This occurs when user tries to read
1697                  * from never connected socket.
1698                 */
1699                 copied = -ENOTCONN;
1700                 break;
1701             }
1702             break;
1703         }
1704 //是否是阻塞的，不是，就 return 了。
1705         if (!timeo) {
1706             copied = -EAGAIN;

```

```

1707         break;
1708     }
1709
1710     if (signal_pending(current)) {
1711         copied = sock_intr_errno(timeo);
1712         break;
1713     }
1714 }
1716 tcp_cleanup_rbuf(sk, copied);
1717
1718 if (!sysctl_tcp_low_latency && tp->ucopy.task == user_recv) {
1719     /* Install new reader */
1720     if (!user_recv && !(flags & (MSG_TRUNC | MSG_PEEK))) {
1721         user_recv = current;
1722         tp->ucopy.task = user_recv;
1723         tp->ucopy.iov = msg->msg iov;
1724     }
1725
1726     tp->ucopy.len = len;
1727     if (!skb_queue_empty(&tp->ucopy.prequeue))
1728         goto do_prequeue;
1729
1730     /* __ Set realtime policy in scheduler __ */
1731 }
1732 if (copied >= target) {
1733     /* Do not sleep, just process backlog. */
1734     release_sock(sk);
1735     lock_sock(sk);
1736 } else
//在此处睡眠了，将在tcp_prequeue函数中调用wake_up_interruptible_poll唤醒
1737     sk_wait_data(sk, &timeo);
//软中断会判断用户是正在读取检查并且睡眠了，如果是的话，就直接把数据拷贝
//到prequeue队列，然后唤醒睡眠的进程。因为进程睡眠，表示没有读到想要的字节数
//此时，软中断有数据到来，直接给进程，这样进程就能以最快的速度被唤醒。
1738
1785     if (user_recv) {
1786         int chunk;
1787
1788         /* __ Restore normal policy in scheduler __ */
1789
1790         if ((chunk = len - tp->ucopy.len) != 0) {
1791             NET_ADD_STATS_USER(sock_net(sk), LINUX_MIB_TCPDIRECTCOPYFROMBACKLOG, chunk);
1792             len -= chunk;
1793             copied += chunk;
1794         }
1795
1796         if (tp->rcv_nxt == tp->copied_seq &&
1797             !skb_queue_empty(&tp->ucopy.prequeue)) {
1798             do_prequeue:
1799                 tcp_prequeue_process(sk);
1800
1801                 if ((chunk = len - tp->ucopy.len) != 0) {
1802                     NET_ADD_STATS_USER(sock_net(sk), LINUX_MIB_TCPDIRECTCOPYFROMPREQUEUE,
1803                     chunk);
1803                     len -= chunk;
1804                     copied += chunk;
1805                 }
1806             }

```

```

1807      }
1808      if ((flags & MSG_PEEK) &&
1809          (peek_seq - copied - urg_hole != tp->copied_seq)) {
1810          net_dbg_ratelimited("TCP(%s:%d): Application bug, race in MSG_PEEK\n",
1811                               current->comm,
1812                               task_pid_nr(current));
1813          peek_seq = tp->copied_seq;
1814      }
1815      continue;
1816
//skb 中还有多少聚聚没有拷贝。

```

//正如前面所说的，`offset` 是上次已经拷贝了的，这次从 `offset` 开始接下去拷贝

```

1817  found_ok_skb:
1818      /* Ok so how much can we use? */
1819      used = skb->len - offset;
//很有可能 used 的大小，即 skb 剩余长度，依然大于用户的缓冲区大小(len)。所以依然
//只能拷贝 len 长度。一般来说，用户还得执行一次 recv 系统调用。直到 skb 中的数据读完

```

```

1820      if (len < used)
1821          used = len;
1822
1823      /* Do we have urgent data here? */
1824      if (tp->urg_data) {
1825          u32 urg_offset = tp->urg_seq - *seq;
1826          if (urg_offset < used) {
1827              if (!urg_offset) {
1828                  if (!sock_flag(sk, SOCK_URGINLINE)) {
1829                      ++*seq;
1830                      urg_hole++;
1831                      offset++;
1832                      used--;
1833                      if (!used)
1834                          goto skip_copy;
1835                  }
1836              } else
1837                  used = urg_offset;
1838          }
1839      }
1840
1841      if (!(flags & MSG_TRUNC)) {
1842          {
//一般都会进这个 if，进行数据的拷贝，把能够读到的数据，放到用户的缓冲区
1843              err = skb_copy_datagram_iovec(skb, offset,
1844                                           msg->msg iov, used);
1845              if (err) {
1846                  /* Exception. Bailout! */
1847                  if (!copied)
1848                      copied = -EFAULT;
1849                  break;
1850              }
1851          }
1852      }
1853
//更新标志位，seq 是指针，指向了 tp->copied_seq

```

//`used` 是我们有能力拷贝的数据大小，即已经拷贝到用户缓冲区的大小
//正如前面所说，如果用户的缓冲区很小，一次 `recv` 拷贝不玩 `skb` 中的数据，
//我们需要保存已经拷贝了的大小，下次 `recv` 时，从这个大小处继续拷贝。
//所以需要更新 `copied_seq`。

```

1882     *seq += used;
1883     copied += used;
1884     len -= used;
1885
1886     tcp_rcv_space_adjust(sk);
1887
1888 skip_copy:
1889     if (tp->urg_data && after(tp->copied_seq, tp->urg_seq)) {
1890         tp->urg_data = 0;
1891         tcp_fast_path_check(sk);
1892     }
//这个就是判断我们是否拷贝完了 skb 中的数据，如果没有 continue
//这种情况下，len 经过 len -= used;，已经变成 0，所以 continue 的效果相当于
//退出了这个大循环。可以理解，你只能拷贝 len 长度，拷贝完之后，那就 return 了。

//还有一种情况 used + offset == skb->len，表示 skb 拷贝完了。这时我们只需要释放 skb
//下面会讲到
1893     if (used + offset < skb->len)
1894         continue;
//看看这个数据报文是否含有 fin，含有 fin，则 goto 到 found_fin_ok
1896     if (tcp_hdr(skb)->fin)
1897         goto found_fin_ok;
//执行到这里，标明 used + offset == skb->len，报文也拷贝完了，那就把 skb 摘链释放
1898     if (!(flags & MSG_PEEK)) {
1899         sk_eat_skb(sk, skb, copied_early);
1900         copied_early = false;
1901     }
//这个 continue 不一定是退出大循环，可能还会执行循环。
//假设用户设置缓冲区 12 字节，你 skb->len 长度 20 字节。
//第一次 recv 读取了 12 字节，skb 剩下 8，下一次调用 recv 再想读取 12，
//但是只能读取到这 8 字节了。
//此时 len 变量长度为 4，那么这个 continue 依旧在这个循环中，
//函数还是再次从 do 开始，使用 skb_queue_walk，找 skb
//如果 sk_receive_queue 中 skb 仍旧有，那么继续读，直到 len == 0
//如果没有 skb 了，我们怎么办？我们的 len 还有 4 字节怎么办？
//这得看用户设置的 recv 函数阻塞与否，即和 timeo 变量相关了。
1902     continue;
1903
1904 found_fin_ok:
1905     /* Process the FIN. */
1906     ++*seq;
1907     if (!(flags & MSG_PEEK)) {
//把 skb 从 sk_receive_queue 中摘链
1908         sk_eat_skb(sk, skb, copied_early);
1909         copied_early = false;
1910     }
1911     break;
1912 } while (len > 0);
1913
//到这里是大循环退出
//休眠过的进程，然后退出大循环，才满足 if (user_recv) 条件
1914 if (user_recv) {
1915     if (!skb_queue_empty(&tp->ucopy.prequeue)) {
1916         int chunk;
1917
1918         tp->ucopy.len = copied > 0 ? len : 0;
1919
1920         tcp_prequeue_process(sk);

```

```
1921
1922     if (copied > 0 && (chunk = len - tp->ucopy.len) != 0) {
1923         NET_ADD_STATS_USER(sock_net(sk), LINUX_MIB_TCPCRECTCOPYFROMPREQUEUE, chunk);
1924         len -= chunk;
1925         copied += chunk;
1926     }
1927 }
1928 //数据读取完毕, 清零
1929     tp->ucopy.task = NULL;
1930     tp->ucopy.len = 0;
1931 }
1932 /* According to UNIX98, msg_name/msg_namelen are ignored
1933    * on connected socket. I was just happy when found this 8) --ANK
1934    */
1935
1936
1937 /* Clean up data we have read: This will do ACK frames. /
1938 //很重要, 将更新缓存, 并且适当的时候发送 ack
1939     tcp_cleanup_rbuf(sk, copied);
1940
1941     release_sock(sk);
1942     return copied;
1943
1944
1945 out:
1946     release_sock(sk);
1947     return err;
1948
1949
1950 recv_urg:
1951     err = tcp_recv_urg(sk, msg, len, flags);
1952     goto out;
1953
1954 recv_sndq:
1955     err = tcp_peek_sndq(sk, msg, len);
1956     goto out;
1957 }
```

第九章 tcp 拥塞控制

Linux 提供丰富的拥塞控制算法，这些算法包括 Vegas、Reno、HSCTP (High Speed TCP)、Westwood、BIC-TCP、CUBIC、STCP (Scalable TCP)、Hybla 以及 Veno 等，对于 Linux3.10 而言，这些算法在添加到内核时会被注册到同一个链表。

9.1 CUBIC 拥塞控制

`tcp_sock` 函数使用到的控制拥塞变量如下：

`snd_cwnd`: 拥塞控制窗口的大小

`snd_ssthresh`: 慢启动门限，如果 `snd_cwnd` 值小于此值这处于慢启动阶段。

`snd_cwnd_cnt`: 当超过慢启动门限时，该值用于降低窗口增加的速率。

`snd_cwnd_clamp`: `snd_cwnd` 能够增加到的最大尺寸。

`snd_cwnd_stamp`: 拥塞控制窗口有效的最后一次时间戳

`snd_cwnd_used`: 用于标记在使用的拥塞窗口的高水位值，当 `tcp` 连接的数量被应用程序限制而不是被网络所限制时，该变量用于下调 `snd_cwnd` 值。

Linux 也支持用户空间动态插入拥塞控制算法，通过 `tcp_cong.c` 注册，拥塞控制使用的函数通过向 `tcp_register_congestion_control` 传递 `tcp_congestion_ops` 实现，用户插入的拥塞控制算法需要支持 `ssthresh` 和 `con_avoid`。

`tp->ca_priv` 用于存放拥塞控制私有数据。`tcp_ca(tp)` 返回值是指向该地址空间的。

当前有三种拥塞控制算法：

最简单的源于 TCP reno (高速、高伸缩性)。

其次是更复杂的 BIC 算法、Vegas 和 Westwood+ 算法，这类算法对拥塞的控制会依赖于其它事件。

优秀的 TCP 拥塞控制算法是复杂的，这需要在公平和性能之间权衡。

当前 Linux 系统使用的拥塞控制算法取决于 `sysctl` 接口的 `net.ipv4.tcp_congestion_control`。缺省的拥塞控制算法是最后注册的算法 (LIFO)，如果全部编译成模块，则将使用 reno 算法，如果使用缺省的 Kconfig 配置，CUBIC 算法将会编译进内核 (不是编译成 module)，并且内核将使用 CUBIC 算法作为默认的拥塞控制算法。

cubic 使用的算法

窗长增长函数：

$$W(t) = C(t - K)^3 + W_{max} \quad \dots \dots \dots (1)$$

C 是 cubic 参数，t 是自上一次窗口减少的时间，K 是上述函数在没有丢包时从 W 增加到 W_{max} 所花费的时间周期。其计算公式是

$$K = \sqrt[3]{\frac{W_{max}\beta}{C}} \quad \dots \dots \dots (2)$$

在拥塞避免阶段收到 ACK 时。CUBIC 在下一个 RTT 使用公式 1 计算窗口增长率。其将 $W(t + RTT)$ 设置成拥塞窗口大小。

根据当前拥塞窗口大小，CUBIC 有三种状态，TCP 状态 (t 时刻窗长小于标准 TCP 窗长)、凹区域(拥塞窗口小于 W_{max})、凸区域(拥塞窗口大于 W_{max})。

cubic 慢启动门限阈值

该方法在快速和长距离网络上使用立方函数修改拥塞线性窗口。该方法使窗口的增加独立于 RTT (round trip times)，这使得具有不同 RTT 的流具有相对均等的网络带宽。到达稳定

阶段，CUBIC 在稳定阶段将急速向饱和点增加，但是快到饱和点时增加速度会变慢。该特性使得 CUBIC 在带宽延迟积（BDP bandwidth and delay product）较大时具有很好的可扩展性、稳定性和公平性。立方根计算方法 Newton-Raphson，误差约为 0.195%。

首先找慢启动门限初始值 `snd_ssthresh`，在 TCP 套接字初始化时 `tcp_prot` 的 `init` 成员会被调用，该函数直接指向 `tcp_v4_init_sock()`。下列代码片段的 2163 行对套接字进行初始化。

```
net/ipv4/tcp_ipv4.c
2159 static int tcp_v4_init_sock(struct sock *sk)
2160 {
//icsk—意为 inet connection sock
2161     struct inet_connection_sock *icsk = inet_csk(sk);
//套接字初始化
2163     tcp_init_sock(sk);
//ipv4 连接的套接字操作函数集
2165     icsk->icsk_af_ops = &ipv4_specific;
2166
2171     return 0;
2172 }
2852 struct proto tcp_prot = {
2853     .name          = "TCP",
2854     .owner         = THIS_MODULE,
2855     .close         = tcp_close,
2856     .connect       = tcp_v4_connect,
2857     .disconnect    = tcp_disconnect,
2858     .accept        = inet_csk_accept,
2859     .ioctl         = tcp_ioctl,
2860     .init          = tcp_v4_init_sock,
2897 }
```

`tcp_init_sock()` 用于初始化套接字，由于 `sk_alloc()` 函数在为套接字分配内存时，已经将一些变量的初始值设置为了 0，所以 `tcp_init_sock()` 并没有初始化所有变量。

<net/ipv4/tcp.c>

```
372 void tcp_init_sock(struct sock *sk)
373 {
374     struct inet_connection_sock *icsk = inet_csk(sk);
//如 9.1 节所述，tcp_sock 的结构体中包含了拥塞控制所需的各种变量
375     struct tcp_sock *tp = tcp_sk(sk);
//存放乱序 tcp 包的套接字链表初始化
377     skb_queue_head_init(&tp->out_of_order_queue);
//重传、延迟 ack 以及探测定时器初始化。
378     tcp_init_xmit_timers(sk);
//记录套接字直接拷贝到用户空间信息的结构体 ucopy 初始化
379     tcp_pqueue_init(tp);
380     INIT_LIST_HEAD(&tp->tsq_node);
//重传超时值，起始值设置为 1s。
382     icsk->icsk_rto = TCP_TIMEOUT_INIT;
//mdev -- medium deviation, 用于 RTT 测量的均方差
383     tp->mdev = TCP_TIMEOUT_INIT;
//初始拥塞窗口大小，初始值 10，这就意味着窗长在大于 10 时才会进入拥塞算法，而一开始进入的是慢启动阶段。
390     tp->snd_cwnd = TCP_INIT_CWND;
//慢启动门限 0x7FFFFFFF
395     tp->snd_ssthresh = TCP_INFINITE_SSTHRESH;
//拥塞窗口最大窗长
396     tp->snd_cwnd_clamp = ~0;
//mss maximum segment size，初始值设置为 536，不包括 SACKS (selective ACK)
397     tp->mss_cache = TCP_MSS_DEFAULT;
398
399     tp->reordering = sysctl_tcp_reordering;
```

```

400     tcp_enable_early_retrans(tp);
401     icsk->icsk_ca_ops = &tcp_init_congestion_ops;
//时间戳偏移
403     tp->tsoffset = 0;
//套接字当前状态 sysctl_tcp_rmem[1]对应的是 default, [0]是 min, [2]最大值
405     sk->sk_state = TCP_CLOSE;
//发送和接收 buffer,
416     sk->sk_sndbuf = sysctl_tcp_wmem[1];
417     sk->sk_rcvbuf = sysctl_tcp_rmem[1];
423 }

```

CUBIC 算法慢启动门限 ssthresh 在两种情况下会得到更新,一种是在接收到 ack 应答包,另一种是在发生拥塞时, 慢启动门限回退。对应使用到的处理函数分别是 bictcp_acked()和 bictcp_recalc_ssthresh()。

```

434 static struct tcp_congestion_ops cubictcp __read_mostly = {
//CUBIC 算法变量初始化, 在 tcp 三次连接时, 回调用其初始化套接字的拥塞控制变量。
435     .init      = bictcp_init,
//拥塞时慢启动门限回退计算
436     .ssthresh  = bictcp_recalc_ssthresh,
437     .cong_avoid = bictcp_cong_avoid, //拥塞控制
438     .set_state  = bictcp_state, //如果拥塞状态是 TCP_CA_Loss, Reset 拥塞算法 CUBIC 的各种变量
//拥塞窗口回退。
439     .undo_cwnd = bictcp_undo_cwnd,
//当 tcp_ack 调用 tcp_clean_rtx_queue 将收到应答的数据包从重传队列删除时, 会调用 bictcp_acked 更新慢启动阈值
440     .pkts_acked = bictcp_acked,
441     .owner     = THIS_MODULE,
442     .name      = "cubic",
443 };

```

在 tcp_ack()函数收到 ack 包时, 会调用 tcp_clean_rtx_queue()将已经收到应答包的帧从重传队列删除, 在这个函数的末尾会调用 bictcp_acked()更新慢启动门限值。

```

396 static void bictcp_acked(struct sock *sk, u32 cnt, s32 rtt_us)
397 {
398     const struct inet_connection_sock *icsk = inet_csk(sk);
399     const struct tcp_sock *tp = tcp_sk(sk);
400     struct bictcp *ca = inet_csk_ca(sk);
401     u32 delay;
//1) 混合慢启动 (train 和 delaed) 标志 hystart 默认是开启的, 2) 当前窗长 snd_cwnd 应该满足小于 tcp_init_sock()函数设置
//值, 3) hystart_low_window 是内核设置的最小拥塞窗口值 16。
429     if (hystart && tp->snd_cwnd <= tp->snd_ssthresh &&
430         tp->snd_cwnd >= hystart_low_window)
431         hystart_update(sk, delay);
432 }

```

起始时慢启动门限设置成了很大的值 0x7FFFFFFF, 由 429 和 431 可知, snd_cwnd 会一直增加知道该值大于等于 hystart_low_window (16) 时, 将调用 hystart_update 更新慢启动门限值。

```

358 static void hystart_update(struct sock *sk, u32 delay)
359 {
360     struct tcp_sock *tp = tcp_sk(sk);
361     struct bictcp *ca = inet_csk_ca(sk);
362
363     if (!(ca->found & hystart_detect)) {
364         u32 now = bictcp_clock();
//不论是 train 方法还是 delayed 方法满足离开慢启动条件, 这里将当前的 snd_cwnd 设置成新的慢启动门限, 即由 0x7FFFFFFF
//设置成 16。
388     if (ca->found & hystart_detect)
389         tp->snd_ssthresh = tp->snd_cwnd;

```

```
390     }
391 }
```

9.2 cubic 拥塞代码实现

慢启动 slow start

`tcp_ack()`在正确接收到应答包后，有如下代码：

```
icsk->icsk_ca_ops->cong_avoid(sk, ack, in_flight);
该代码调用 tcp_cubic.c 文件的 437 行函数。
net/ipv4/tcp_cubic.c
305 static void bictcp_cong_avoid(struct sock *sk, u32 ack, u32 in_flight)
306 {
307     struct tcp_sock *tp = tcp_sk(sk);
308     struct bictcp *ca = inet_csk_ca(sk);
//检查发送出去还没收到 ACK 包的数量是否已达到拥塞控制窗口上限，达到则返回。
310     if (!tcp_is_cwnd_limited(sk, in_flight))
311         return;
//当前窗长小于慢启动门限，则进入慢启动控制，否则进入拥塞避免
313     if (tp->snd_cwnd <= tp->snd_ssthresh) {
//判断是否需要重置 sk 的 CUBIC 算法使用到的变量。
314         if (hystart && after(ack, ca->end_seq))
315             bictcp_hystart_reset(sk);
//慢启动处理函数
316         tcp_slow_start(tp);
317     } else {
//更新 ca (congestion avoid) 的 cnt 成员，拥塞避免函数会使用该成员
318         bictcp_update(ca, tp->snd_cwnd);
//拥塞避免处理算法
319         tcp_cong_avoid_ai(tp, ca->cnt);
320     }
321
322 }
```

RFC2861，检查是否被应用程序或者拥塞窗口限制，其参数 `in_flight` 是已经发送但是还没有经过确认的数据包，如果被限制则返回 1，说明需要进行拥塞控制，否则不需要拥塞控制。

```
net/ipv4/tcp_cong.c
284 bool tcp_is_cwnd_limited(const struct sock *sk, u32 in_flight)
285 {
286     const struct tcp_sock *tp = tcp_sk(sk);
287     u32 left;
//未确认包数量大于等于当前的窗长，返回 true
289     if (in_flight >= tp->snd_cwnd)
290         return true;
//还可以发送的窗口剩余量
292     left = tp->snd_cwnd - in_flight;
//判断 SK 的 sk_route_caps 成员是否支持 gso，这是软件实现的功能。
293     if (sk_can_gso(sk) &&
// tcp_tso_win_divisor 是 sysctl 接口，即一个 TSO 帧可以占用拥塞窗口长度的百分比。
294         left * sysctl_tcp_tso_win_divisor < tp->snd_cwnd &&
//最大 GSO 段的大小
295         left * tp->mss_cache < sk->sk_gso_max_size &&
//最多 GSO 段的个数
296         left < sk->sk_gso_max_segs)
```

```

297         return true;
//没有使用tcp_tso_win_divisor时，最多TSO可以延迟发送的MSS的最多个数
298     return left <= tcp_max_tso_deferred_mss(tp);
299 }

```

不论是reno还是cubic等拥塞控制算法，它们使用慢启动处理函数是一样的。当前3.10版本的内核支持RFC2581基本规范。

```

net/ipv4/tcp_cong.c
309 void tcp_slow_start(struct tcp_sock *tp)
310 {
311     int cnt; /* increase in packets */
312     unsigned int delta = 0;
313     u32 snd_cwnd = tp->snd_cwnd;
//如果管理员使用sysctl接口，配置了慢启动增加值，就按照管理员的设置来，否则会以指数方式增加窗长
320     if (sysctl_tcp_max_ssthresh > 0 && tp->snd_cwnd > sysctl_tcp_max_ssthresh)
321         cnt = sysctl_tcp_max_ssthresh >> 1; /* limited slow start */将慢启动门限除以二。
322     else
323         cnt = snd_cwnd; /* exponential increase */
//snd_cwnd_cnt在拥塞发生时会被重置0，否则其值是一直增长的，如果起始snd_cwnd等于10
325     tp->snd_cwnd_cnt += cnt;
326     while (tp->snd_cwnd_cnt >= snd_cwnd) { //窗长+1
327         tp->snd_cwnd_cnt -= snd_cwnd;
328         delta++;
329     }
330     tp->snd_cwnd = min(snd_cwnd + delta, tp->snd_cwnd_clamp); //发送窗长不能超限
331 }

```

拥塞避免 congestion avoid

拥塞避免：从慢启动可以看到，`cwnd`可以很快的增长上来，从而最大程度利用网络带宽资源，但是`cwnd`不能一直这样无限增长下去，一定需要某个限制。TCP使用了一个叫慢启动门限(`ssthresh`)的变量，当`cwnd`超过该值后，慢启动过程结束，进入拥塞避免阶段。对于大多数TCP实现来说，`ssthresh`的值是65536(同样以字节计算)。拥塞避免的主要思想是加法增大，也就是`cwnd`的值不再指数级往上升，开始加法增加。此时当窗口中所有的报文段都被确认时，`cwnd`的大小加1，`cwnd`的值就随着RTT开始线性增加，这样就可以避免增长过快导致网络拥塞，慢慢的增加调整到网络的最佳值。

Cubic窗长更新函数如下，更新的公式参考公式1、2：

```

207 static inline void bictcp_update(struct bictcp *ca, u32 cwnd)
208 {
209     u64 offs;
210     u32 delta, t, bic_target, max_cnt;
211
212     ca->ack_cnt++; /* count the number of ACKs */
213
214     if (ca->last_cwnd == cwnd &&
215         (s32)(tcp_time_stamp - ca->last_time) <= HZ / 32)
216         return;
217
218     ca->last_cwnd = cwnd;
219     ca->last_time = tcp_time_stamp;
220
221     if (ca->epoch_start == 0) {
222         ca->epoch_start = tcp_time_stamp; /* record the beginning of an epoch */
223         ca->ack_cnt = 1; /* start counting */
224         ca->tcp_cwnd = cwnd; /* syn with cubic */

```

```

225     if (ca->last_max_cwnd <= cwnd) {
226         ca->bic_K = 0;
227         ca->bic_origin_point = cwnd;
228     } else {
229         /* Compute new K based on
230          * (wmax-cwnd) * (srtt>>3 / HZ) / c * 2^(3*bictcp_HZ)
231          */
232         ca->bic_K = cubic_root(cube_factor
233             * (ca->last_max_cwnd - cwnd));
234         ca->bic_origin_point = ca->last_max_cwnd;
235     }
236 }
237 }

//254-303 参考公式 1 和公式 2.
254     t = ((tcp_time_stamp + msecs_to_jiffies(ca->delay_min>>3)
255           - ca->epoch_start) << BICTCP_HZ) / HZ;
256
257     if (t < ca->bic_K)      /* t - K */
258         offs = ca->bic_K - t;
259     else
260         offs = t - ca->bic_K;
261
262     /* c/rtt * (t-K)^3 */
263     delta = (cube_rtt_scale * offs * offs * offs) >> (10+3*BICTCP_HZ);
264     if (t < ca->bic_K)                      /* below origin*/
265         bic_target = ca->bic_origin_point - delta;
266     else                                     /* above origin*/
267         bic_target = ca->bic_origin_point + delta;
268
269     /* cubic function - calc bictcp_cnt*/
270     if (bic_target > cwnd) {
271         ca->cnt = cwnd / (bic_target - cwnd);
272     } else {
273         ca->cnt = 100 * cwnd;                /* very small increment*/
274     }
275
276     /*
277      * The initial growth of cubic function may be too conservative
278      * when the available bandwidth is still unknown.
279      */
280     if (ca->last_max_cwnd == 0 && ca->cnt > 20)
281         ca->cnt = 20;    /* increase cwnd 5% per RTT */
282
283     /* TCP Friendly */
284     if (tcp_friendliness) {
285         u32 scale = beta_scale;
286         delta = (cwnd * scale) >> 3;
287         while (ca->ack_cnt > delta) {        /* update tcp cwnd */
288             ca->ack_cnt -= delta;
289             ca->tcp_cwnd++;
290         }
291
292         if (ca->tcp_cwnd > cwnd){ /* if bic is slower than tcp */
293             delta = ca->tcp_cwnd - cwnd;
294             max_cnt = cwnd / delta;
295             if (ca->cnt > max_cnt)
296                 ca->cnt = max_cnt;
297         }

```


下篇 杂项汇总

第十章 网络工具

在测试 io 设备时，常常会用到 iostat、iotop 工具，在查看内存时常常用到 vmstat、free、slabtop 工具，在查看调度器时，常常使用 mpstat、top 以及 ps 工具。这里来说说网络相关的工具，有性能分析、网络管理、状态查看类工具。下面的一张图显示了网络相关工具和其作用的层次关系。

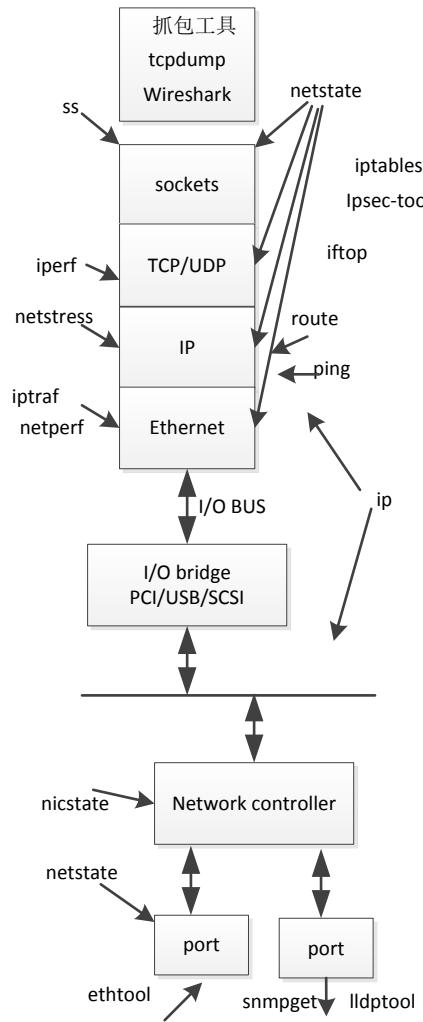


图 10.1 网络工具

由上至下的来看该这些工具。

10.1 ss

ss 是 iproute2 包提供的工具，该包此外还支持非常重要的 ip 命令，ip 命令设计的初衷是替代 ifconfig(net-tools 工具包，net-tools 支持网络接口配置，路由表以及 ARP 表的管理)，但是目前 pc 和服务器上通常对两种工具都支持，但是嵌入式环境对 ip 的支持需要开发者完成移植工作。ss 工具由于导出套接字的统计信息，和图中的 netstat 作用类似，但是可以显

示更过的 TCP 和网络状态信息。ss -h 可以查看帮助信息。

1、查看所有打开的 tcp 端口使用如下命令：

```
ss -tnap
```

2、可以使用-4 或者-6 参数指定 tcp 端口是 IPv4 还是 IPv6：

```
ss -tnap4
```

3、查看 udp 使用如下命令：

```
ss -unap
```

4、统计信息使用如下命令：

```
ss -s
```

例如如下是该命令的输出：

```
Total: 684 (kernel 0)
```

```
TCP: 111 (estab 75, closed 0, orphaned 0, synrecv 0, timewait 0/0), ports 0
```

	Transport Total	IP	IPv6
*	0	-	-
RAW	0	0	0
UDP	43	29	14
TCP	111	96	15
INET	154	125	29
Frag	0	0	0

5、还可以使用连接状态过滤选项如下：

```
ss -o state established '( dport =:ssh or sport =:ssh )'
```

10.2 netstat

netstat 工具

```
netstat [-vWnNcaeol] [<Socket> ...]
```

```
netstat { [-vWeenNac] -i | [-cWnNe] -M | -s }
```

路由表、实际的网络连接以及每一个网络接口设备的状态信息。Netstat 用于显示与 IP、TCP、UDP 和 ICMP 协议相关的统计数据，一般用于检验本机各端口的网络连接情况。

列标题

Name [接口](#)的名字

Mtu 接口的最大传输单位

Net/Dest 接口所在的[网络](#)

Address 接口的 IP 地址

Ipkts 接收到的数据包数目

errs 接收到时已损坏的数据包数目

Opkts 发送的数据包数目

Oerrs 发送时已损坏的数据包数目

Collisions 由这个接口所记录的网络冲突数目

常用选项

netstat -s

——本选项能够按照各个协议分别显示其统计数据。如果你的应用程序（如 Web 浏览器）运行速度比较慢，或者不能显示 Web 页之类的数据，那么你就可以用本选项来查看一下所显示的信息。你需要仔细查看统计数据的各行，找到出错的[关键字](#)，进而确定问题所在。

netstat -e

——本选项用于显示关于以太网的统计数据，它列出的项目包括传送数据报的总字节数、错误数、删除数，包括发送和接收量（如发送和接收的字节数、数据包数[1|1])，或有广播的数量。可以用来统计一些基本的网络流量。

netstat -r

——本选项可以显示关于路由表的信息，类似于后面所讲使用 `routeprint` 命令时看到的信息。除了显示有效路由外，还显示当前有效的连接。

netstat -a

——本选项显示一个所有的有效连接信息列表,包括已建立的连接(ESTABLISHED),也包括[监听](#)连接请求(LISTENING)的那些连接。

netstat -n

——显示所有已建立的有效连接。

netstat -p

——显示协议名查看某协议使用情况

10.3 netstress

netstress 是一个 DDoS 和网络压测工具，DDos(分布式拒绝服务)，对电子商务网站而言，如果这种攻击在双十一对其攻击，目前其没有很好的方法处理。测试的攻击类型包括 SYN , ACK , FIN , UDP , ICMP , HTTP , Mixed , DNS 。

```
$ sudo ./netstress_fullrandom
--saddr,      -s:  source address
--sport,      -p:  source port
--daddr,      -d:  destination address
--dport,      -P:  destination port
--file,       -f:  the full path for the file of dns server list for
ampdns flood
--attack,     -a:  type of attack (ack, syn, fin, udp, dns, ampdns,
igmp, winbomb, win98,
              get, post, syncook, isssyn)
```

```
--process, -n: number of processes
--buffer, -b: size of UDP packet
--dnsqname, -N: hostname which will be queried
--dnsqtype, -t: type of dns query (a, ns, cname, soa, wks, ptr,
hinfo, minfo, mx, txt)
--useragent, -u: user agent parameter for http get flood
--help, -h: shows this message
```

这里有一个源 IP 是 1.1.1.1、端口号是 8888 到目的 IP 2.2.2.2、端口号 9999 的 SYN 同步包的攻击，只使用一个线程进行攻击。

```
$ sudo ./netstress_fullrandom -s 1.1.1.1 -p 8888 -d 2.2.2.2 -P 9999 -a
syn -n 1
```

10.4 netperf 参考

《netperf 网络性能检测工具-嵌入式》

10.5 iperf

iperf 是一个网络性能测试工具。Iperf 可以测试最大 TCP 和 UDP 带宽性能。Iperf 具有多种参数和 UDP 特性，可以根据需要调整。Iperf 可以报告带宽，延迟抖动和数据包丢失。

1) TCP 测试

服务器执行: ./iperf -s -i 1 -w 1M 这里是指定 Linux，如果是 iperf -s 则是 windows 平台上命令。默认包大小为 8kbyte/s

客户端执行: ./iperf -c host -i 1 -w 1M

其中-w 表示 TCP window size，host 需替换成服务器地址。

2) UDP 测试

服务器执行: ./iperf -u -s

客户端执行: ./iperf -u -c 10.255.255.251 -b 900M -i 1 -w 1M -t 60

其中-b 表示使用多少带宽，1G 的线路你可以使用 900M 进行测试。

10.6 iptraf

网络流量实时监控工具，功能比 nload 更强大，可以监控所有的流量，IP 流量，按协议分的流量，还可以设置过滤器等

源码安装

```
wget ftp://iptraf.seul.org/pub/iptraf/iptraf-3.0.0.tar.gz
```

```
tar zxvf iptraf-3.0.0.tar.gz
```

```
cd iptraf-3.0.0
```

`./Setup`

直接运行 iptraf，后有一个如下的菜单提示，然后进入相关的选项查看

IP 流量监视(IP traffic monitor)

网络接口的一般信息统计(General Interface Statistics)

网络接口的细节信息统计(Detailed Interface Statistics)

统计分析(Statistical Breakdowns)

局域网工作站统计(LAN Station Statistics)

过滤器(Filters...)

配置(Configure...)

退出(Exit)

10.7 TcpDump

可以将网络中传送的数据包的“头”完全截获下来提供分析。它支持针对网络层、协议、主机、网络或端口的过滤，并提供 and、or、not 等逻辑语句来帮助你去掉无用的信息。

10.7.1 数据过滤

不带任何参数的 TcpDump 将搜索系统中第一个网络接口，并显示它截获的所有数据，这些数据对我们不一定全都需要，而且数据太多不利于分析。所以，我们应当先想好需要哪些数据，TcpDump 提供以下参数供我们选择数据：

`-b` 在网络层上选择协议，包括 ip、arp、rarp、ipx 都是这一层的。

例如：`tcpdump -b arp` 将只显示网络中的 arp 即地址转换协议信息。

`-i` 选择过滤的网络接口，如果是作为路由器至少有两个网络接口，通过这个选项，就可以只过滤指定的接口上通过的数据。例如：

`tcpdump -i eth0` 只显示通过 eth0 接口上的所有报头。

`src`、`dst`、`port`、`host`、`net`、`ether`、`gateway` 这几个选项又分别包含 `src`、`dst`、`port`、`host`、`net`、`ehost` 等附加选项。他们用来分辨数据包的来源和去向，`src host 192.168.0.1` 指定源主机 IP 地址是 192.168.0.1，`dst net 192.168.0.0/24` 指定目标是网络 192.168.0.0。以此类推，`host` 是与其指定主机相关无论它是源还是目的，`net` 是与其指定网络相关的，`ether` 后面跟的不是 IP 地址而是物理地址，而 `gateway` 则用于网关主机。可能有点复杂，看下面例子就知道了：

`tcpdump src host 192.168.0.1 and dst net 192.168.0.0/24`

过滤的是源主机为 192.168.0.1 与目的网络为 192.168.0.0 的报头。

`tcpdump ether src 00:50:04:BA:9B and dst.....`

过滤源主机物理地址为 XXX 的报头（为什么 `ether src` 后面没有 `host` 或者 `net`？物理地址当然不可能有网络喽）。

`Tcpdump src host 192.168.0.1 and dst port not telnet`

过滤源主机 192.168.0.1 和目的端口不是 telnet 的报头。

ip icmp arp rarp 和 tcp、 udp、 icmp 这些选项等都要放到第一个参数的位置，用来过滤数据报的类型。

例如：

```
tcpdump ip src.....
```

只过滤数据-链路层上的 IP 报头。

```
tcpdump udp and src host 192.168.0.1
```

只过滤源主机 192.168.0.1 的所有 udp 报头。

10.7.2 输入输出

TcpDump 提供了足够的参数来让我们选择如何处理得到的数据，如下所示：

-l 可以将数据重定向。

如 `tcpdump -l >tcpcap.txt` 将得到的数据存入 `tcpcap.txt` 文件中。

-n 不进行 IP 地址到主机名的转换。

如果不使用这一项，当系统中存在某一主机的主机名时，TcpDump 会把 IP 地址转换为主机名显示，就像这样：`eth0 < ntc9.1165> router.telnet`，使用 -n 后变成了：`eth0 < 192.168.0.9.1165 > 192.168.0.1.telnet`。

-nn 不进行端口名称的转换。

上面这条信息使用 -nn 后就变成了：`eth0 < ntc9.1165 > router..23`。

-N 不打印出默认的域名。

还是这条信息 -N 后就是：`eth0 < ntc9.1165 > router.telnet`。

-O 不进行匹配代码的优化。

-t 不打印 UNIX 时间戳，也就是不显示时间。

-tt 打印原始的、未格式化过的时间。

-v 详细的输出，也就比普通的多了个 TTL 和服务类型

该工具可以结合 wireshark 使用，结果更直观，更容易看，在编写网络应用程序时，在 windows 上安装 wireshark，对通信的数据包进行分析，这种方法常常能获得非常有价值的信息。

10.8 nicstat

网络流量统计实用工具

10.8.1 nicstat 的安装：

- `wget http://nchc.dl.sourceforge.net/project/nicstat/nicstat-1.92.tar.gz`
- `# tar zxvf nicstat-1.92.tar.gz`

```

• # cd nicstat-1.92
• # cp Makefile.Linux Makefile
• # uname -m
• x86_64
• # diff Makefile Makefile.Linux ## 如果不是 64 位机器则不需要修改
  Makefile
• 17c17
• < CFLAGS =      $(COPT)
• ---
• > CFLAGS =      $(COPT) -m32
• 25c25
• < CPUTYPE = x86_64
• ---
• > CPUTYPE =      i386
• # make && make install

```

10.8.2 nicstat 使用

在解压包目录 nicstat-1.92 内,有个 nicstat.sh 脚本.

查看网卡速度(-l):

```
[root@CentOS192 nicstat-1.92]# ./nicstat.sh -l
Int      Loopback    Mbit/s Duplex State
lo        Yes          -   unkn     up
eth0      No           1000    full     up
```

间隔 3 秒,查看 2 次结果(留意%Util 和 Sat):

```
[root@centos192 nicstat-1.92]# ./nicstat 3 2
Time      Int    rKB/s    wKB/s    rPk/s    wPk/s    rAvs    wAvs %Util    Sat
06:19:46    lo     0.72     0.72     2.15     2.15   341.2   341.2  0.00    0.00
06:19:46    eth0    0.89     0.15     1.37     0.92   660.4   163.6  0.00    0.00
Time      Int    rKB/s    wKB/s    rPk/s    wPk/s    rAvs    wAvs %Util    Sat
06:19:49    lo     0.00     0.00     0.00     0.00     0.00     0.00  0.00    0.00
06:19:49    eth0    0.02     0.12     0.33     0.33   66.00   354.0  0.00    0.00
```

Time 列:表示当前采样的响应时间.

lo and eth0 : 网卡名称.

rKB/s : 每秒接收到千字节数.

wKB/s : 每秒写的千字节数.

rPk/s : 每秒接收到的数据包数目.

wPk/s : 每秒写的数据包数目.

rAvs : 接收到的数据包平均大小.

wAvs : 传输的数据包平均大小.

%Util : 网卡利用率(百分比).

Sat : 网卡每秒的错误数.网卡是否接近饱满的一个指标.尝试去诊断网络问题的时候,推荐使用-x 选项去查看更多的统计信息.

查看扩展信息(-x 和 -s):

```
[root@centos192 nicstat-1.92]# ./nicstat 3 2 -x
```

Time	RdKB	WrKB	RdPkt	WrPkt	IErr	OErr	Coll	NoCP	Defer	%Util
06:33:57										
lo	0.69	0.69	2.08	2.08	0.00	0.00	0.00	0.00	0.00	0.00
eth0	0.86	0.14	1.33	0.89	0.00	0.00	0.00	0.00	0.00	0.00
06:34:00										
lo	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
eth0	0.02	0.12	0.33	0.33	0.00	0.00	0.00	0.00	0.00	0.00

```
[root@centos192 nicstat-1.92]# ./nicstat.sh -s
```

Time	Int	rKB/s	wKB/s
06:37:48	lo	0.685	0.685
06:37:48	eth0	0.848	0.140

查看 tcp 相关信息(-t):[root@centos192 nicstat-1.92]# ./nicstat.sh -t

Time	InKB	OutKB	InSeg	OutSeg	Reset	AttF	%ReTX	InConn	OutCon	
Drops										
TCP	0.00	0.00	4.01	3.50	0.00	0.01	0.000	0.05	0.09	0.0

InKB : 表示每秒接收到的千字节.

OutKB : 表示每秒传输的千字节.

InSeg : 表示每秒接收到的 TCP 数据段(TCP Segments).

OutSeg : 表示每秒传输的 TCP 数据段(TCP Segments).

Reset : 表示 TCP 连接从 ESTABLISHED 或 CLOSE-WAIT 状态直接转变为 CLOSED 状态的次数.

AttF : 表示 TCP 连接从 SYN-SENT 或 SYN-RCVD 状态直接转变为 CLOSED 状态的次数,再加上 TCP 连接从 SYN-RCVD 状态直接转变为 LISTEN 状态的次数

%ReTX : 表示 TCP 数据段(TCP Segments)重传的百分比.即传输的 TCP 数据段包含有一个或多个之前传输的八位字节.

InConn : 表示 TCP 连接从 LISTEN 状态直接转变为 SYN-RCVD 状态的次数.

OutCon : 表示 TCP 连接从 CLOSED 状态直接转变为 SYN-SENT 状态的次数.

Drops : 表示从完成连接(completed connection)的队列和未完成连接(incomplete connection)的队列中丢弃的连接次数.

查看 udp 相关信息(-u):

```
[root@centos192 nicstat-1.92]# ./nicstat.sh -u
06:39:42           InDG   OutDG   InErr   OutErr
UDP            0.01    0.01    0.00    0.00
```

InDG : 每秒接收到的 UDP 数据报(UDP Datagrams)

OutDG : 每秒传输的 UDP 数据报(UDP Datagrams)

InErr : 接收到的因包含错误而不能被处理的数据包

OutErr : 因错误而不能成功传输的数据包.

10.9 ethtool 工具:

```
ethtool ethX /
ethtool -h //显示 ethtool 的命令帮助(help)
ethtool -i ethX //查询 ethX 网口的相关信息
ethtool -d ethX //查询 ethX 网口注册性信息
ethtool -r ethX //重置 ethX 网口到自适应模式
ethtool -S ethX //查询 ethX 网口收发包统计
ethtool -s ethX [speed 10|100|1000]\ //设置网口速率 10/100/1000M
[duplex half|full]\ //设置网口半/全双工
[autoneg on|off]\ //设置网口是否自协商
[port tp|auj|bnc|mii]\ //设置网口类型
ip 命令, 对 OBJECT 中的对象进行配置, 命令功能很强大。源于 iproute2 工具, 默认已安装, 使用内核的 Netlink 机制和内核交互。
Usage: ip [ OPTIONS ] OBJECT { COMMAND | help }
      ip [ -force ] -batch filename
where  OBJECT := { link | addr | addrlabel | route | rule | neigh | ntable |
                  tunnel | tuntap | maddr | mroute | mrule | monitor | xfrm |
                  netns }

OPTIONS := { -V[ersion] | -s[tatistics] | -d[etails] | -r[esolve] |
             -f[amily] { inet | inet6 | ipx | dnet | link } |
             -l[oops] { maximum-addr-flush-attempts } |
             -o[neline] | -t[imestamp] | -b[atch] [filename] |
             -rc[vbuf] [size]}
```

第十一章 Linux 包过滤防火墙-netfilter iptables

11.1 netfilter 框架

netfilter 从 Linux2.4 引入 linux 内核，是现在 3.10 版本的防火墙框架，该框架可实现数据包过滤、数据包处理、地址伪装、透明代理、动态网络地址转换（NetworkAddress Translation, NAT），以及基于用户及媒体访问控制（MediaAccess Control, MAC）地址的过滤和基于状态的过滤、包速率限制等。

netfilter 为每种[网络协议](#)（IPv4、IPv6 等）定义一套 hook 函数。Netfilter 在 Linux 中的框架如图 11.1.1 所示，后文是基于源码对这张拓扑的分析，但是 ipv6 的内容就没有涉及了，主要还是以 ipv4/tcp 协议为主线的。

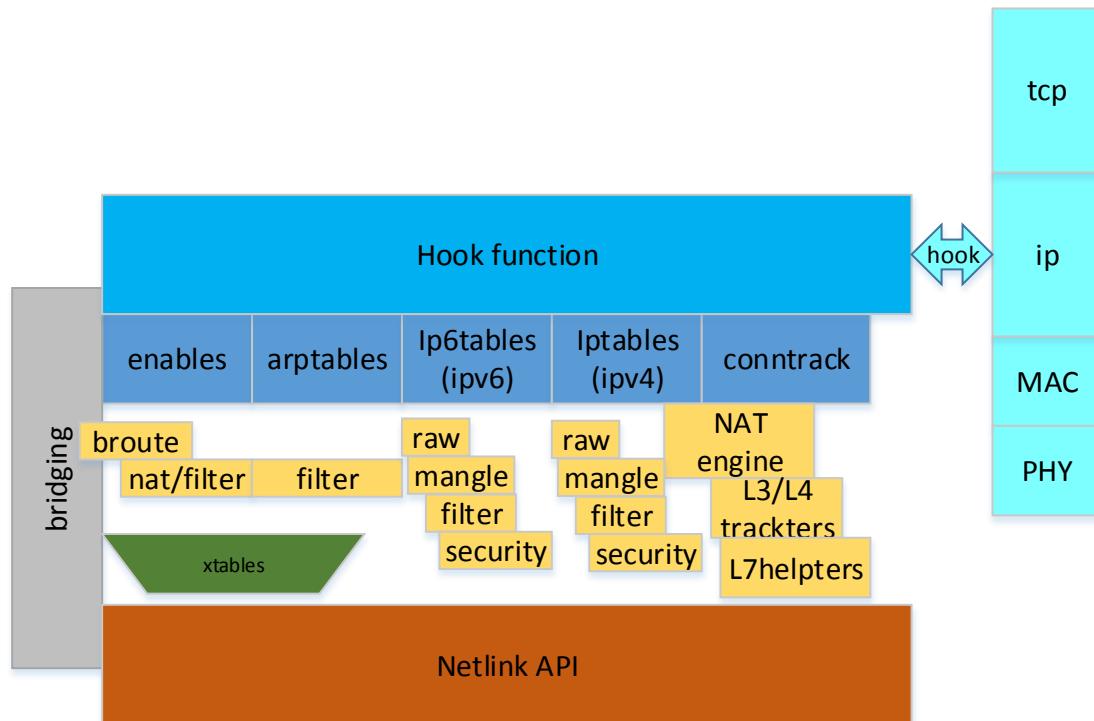


图 11.1.1 netfilter 框架

ipv4 有 5 个[hook 函数](#)，这些 hook 函数在数据报流过协议栈的 5 个关键点被调用，与相应的规则链进行比较，规则链存放在表中，这些表包括 nat、mangle、raw、filter、security 等，根据检查结果，将决定数据包的命运：

- 原样放回 IPv4 协议栈，继续向上层递交；
- 经过修改，再放回网络协议栈；
- 丢弃。

数据在 netfilter 框架中的流通如图 11.1.2 所示。

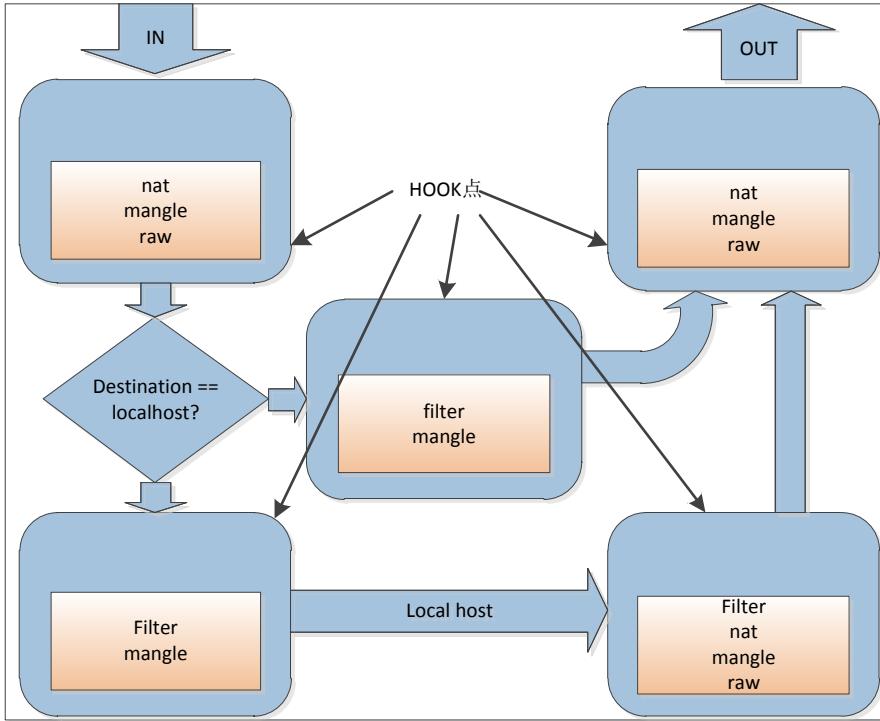


图 11.1.2 五个 hook 点及各点的表

在第一章的时候，对 `module_init` 初始化的函数被跳过了，这里先看看和防火墙的初始化工作吧。以下文件目录前缀均是/`/net/ipv4`。

```

./netfilter.c:206:module_init(ipv4_nfnetfilter_init);
./netfilter/ipt_ah.c:90:module_init(ah_mt_init);
./netfilter/iptable_raw.c:88:module_init(iptable_raw_init);
./netfilter/iptable_security.c:109:module_init(iptable_security_init);
./netfilter/arp_tables.c:1913:module_init(arp_tables_init);
./netfilter/iptable_mangle.c:147:module_init(iptable_mangle_init);
./netfilter/ip_tables.c:2269:module_init(ip_tables_init);
./netfilter/iptable_nat.c:333:module_init(iptable_nat_init);
./netfilter/iptable_filter.c:109:module_init(iptable_filter_init);
./netfilter/nf_defrag_ipv4.c:125:module_init(nf_defrag_init);
./netfilter/ipt_rpfilter.c:146:module_init(rpfilter_mt_init);
./netfilter/arptable_filter.c:90:module_init(arptable_filter_init);

```

`ipv4_nfnetfilter_init` 用于为 INET 协议族中的协议初始化 netfilter，netfilter 支持 internet 的协议类型有以下几种：

```

NFPROTO_UNSPEC = 0,
NFPROTO_IPV4   = 2,
NFPROTO_ARP    = 3,
NFPROTO_BRIDGE = 7,
NFPROTO_IPV6   = 10,
NFPROTO_DECNET = 12,

```

不同的协议类型的 netfilter 的具体细节并不一样，这里仅以 ipv4 协议对 netfilter 实现进

行追踪，所以本章接下来的内容不加说明则其属于 ipv4 的范畴。

`ipv4_nfnetfilter_init`: 该函数注册 internet 协议族的 nfnetfilter，其参数 `nf_ip_afinfo` 的 afinfo 就是 address family information 缩写，该函数就是将 `nf_ip_afinfo` 结构体挂接到 `nf_afinfo` 的数组上去，`nf_afinfo` 数组定义于同名文件的开始处。

```
const struct nf_afinfo __rcu *nf_afinfo[NFPROTO_NUMPROTO] __read_mostly;
```

`nf_ip_afinfo` 结构体定义如下：

```
static const struct nf_afinfo nf_ip_afinfo = {  
    .family = AF_INET,  
    .checksum = nf_ip_checksum,  
    .checksum_partial = nf_ip_checksum_partial,  
    .route = nf_ip_route,  
    .saveroute = nf_ip_saveroute,  
    .reroute = nf_ip_reroute,  
    .route_key_size = sizeof(struct ip_rt_info),  
};
```

上述注册的函数是 nfnetfilter 对 ip 层数据包的处理函数，`checksum` 是 cpu 计算 ip 头校验和验证，`checksum_partial` 是因为现在有些网卡自带校验和计算，它们可以解决 cpu 的资源，后面的三个都是用来路由的，后面遇到再看。

`ah_mt_init`: 注册防火墙表中规则项的 match 函数，该函数用于头信息匹配判断。

```
iptable_security_init;  
iptable_raw_init;  
iptable_mangle_init;  
iptable_nat_init;  
iptable_filter_init;
```

这五张是内核防火墙使用的表，这里注册了这几张表，这几张表将构成一个链式结，表和链的拓扑结构可以参看 11.2.1。由于这几张表的初始化过程差别不大，并且 filter 常用，所以这里就只看 filter 表的初始化了。

```
static int __init iptable_filter_init(void)  
{  
    ret = register_pernet_subsys(&iptable_filter_net_ops); /* Register hooks */  
    filter_ops = xt_hook_link(&packet_filter, iptable_filter_hook);  
}
```

又见 `register_pernet_subsys`，`iptable_filter_net_ops` 会被添加 `first_device_ops` 链表上，该函数注册一个网络命名空间子系统。

```
int register_pernet_subsys(struct pernet_operations *ops)  
{  
    int error;  
  
    mutex_lock(&net_mutex);  
  
    error = register_pernet_operations(first_device, ops);  
  
    mutex_unlock(&net_mutex);
```

```

    return error;
}

}

```

如果 iptable_filter_net_ops 有 init 成员，则 init 成员会被调用。

```

static struct pernet_operations iptable_filter_net_ops = {

    .init = iptable_filter_net_init,
    .exit = iptable_filter_net_exit,
};

}

```

回调函数 iptable_filter_net_init 函数如下

```

58 static int __net_init iptable_filter_net_init(struct net *net)

59 {
60     struct ipt_replace *repl;
61
62     repl = ipt_alloc_initial_table(&packet_filter);
63     if (repl == NULL)
64         return -ENOMEM;
65     /* Entry 1 is the FORWARD hook */
66     ((struct ipt_standard *)repl->entries)[1].target.verdict =
67         forward ? -NF_ACCEPT - 1 : -NF_DROP - 1;
68
69     net->ipv4.iptable_filter =
70         ipt_register_table(net, &packet_filter, repl);
71     kfree(repl);
72     return PTR_RET(net->ipv4.iptable_filter);
73 }

```

62 行 packet_filter 定义如下，调用 xt_alloc_initial_table 宏进行初始化。

```

#define FILTER_VALID_HOOKS ((1 << NF_INET_LOCAL_IN) | \
                        (1 << NF_INET_FORWARD) | \
                        (1 << NF_INET_LOCAL_OUT))

static const struct xt_table packet_filter = {

    .name      = "filter",
    .valid_hooks= FILTER_VALID_HOOKS,
    .me        = THIS_MODULE,
    .af        = NFPROTO_IPV4,
    .priority   = NF_IP_PRI_FILTER,
};

}

```

62 的函数 xt_alloc_initial_table 用于创建表的规则链，这个函数的具体实现比较复杂，先看完整个函数的流程再回过头来细细分析该函数的实现细节。

66 行将 FORWARD 的 hook 项设置为接受，forward 默认值是 true，当然也可以在加载

模块时动态改变该选择为 flase，这样就不支持非本机数据包的转发功能了。

69 向内核注册 filter 表。

再回到 62 行，这里该函数内的宏经过展开处理了，以函数的形式展现在这里了，该“函数”的参数是上面的 packe_filer，即 filter 表结构。

```
void *ipt_alloc_initial_table(const struct xt_table *info)
{
    unsigned int hook_mask = info->valid_hooks; //LOCAL_IN、FORWARD、LOCAL_OUT
    unsigned int nhooks = hweight32(hook_mask); //这里得到 3，上面 hookmask 对应三个 hook 点。
    unsigned int bytes = 0, hooknum = 0, i = 0;
```

看到函数的最后，知道返回值是 tbl，而这里的结构体内嵌的三个结构体是 tbl 的组成，三个结构体的数据结构拓扑图如图 11.1.3。

```
struct {
    struct ipt_replace repl;
    struct ipt_standard entries[nhooks];
    struct ipt_error term;
} *tbl = kzalloc(sizeof(*tbl), GFP_KERNEL);
if (tbl == NULL)
    return NULL;
strncpy(tbl->repl.name, info->name, sizeof(tbl->repl.name));
tbl->term = (struct ipt_error)IPT_ERROR_INIT;
tbl->repl.valid_hooks = hook_mask;
tbl->repl.num_entries = nhooks + 1;
tbl->repl.size = nhooks * sizeof(struct ipt_standard) + sizeof(struct ipt_error);
for (; hook_mask != 0; hook_mask >= 1, ++hooknum) {
    if (!(hook_mask & 1))
        continue;
    tbl->repl.hook_entry[hooknum] = bytes;
    tbl->repl.underflow[hooknum] = bytes;
    tbl->entries[i++] = (struct ipt_standard) IPT_STANDARD_INIT(NF_ACCEPT);
    bytes += sizeof(struct ipt_standard);
}
return tbl;
}
```

看到函数的最后，知道返回值是 tbl，而这里

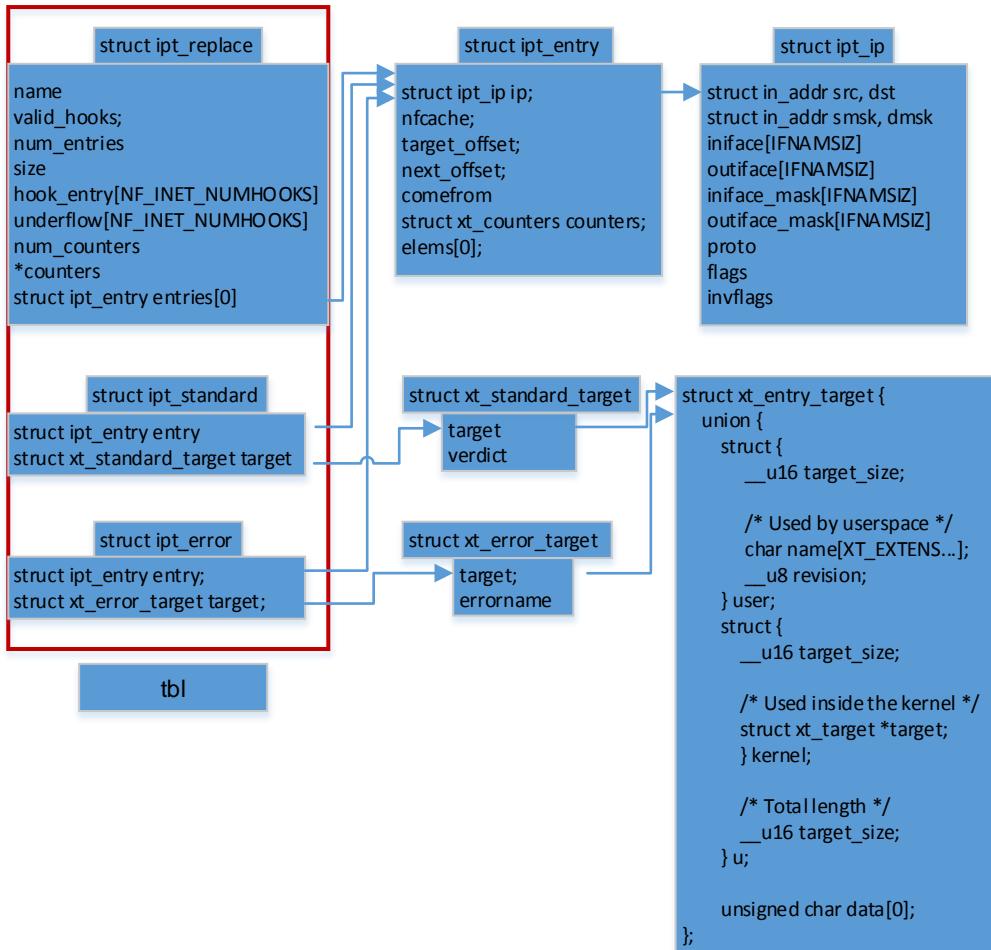


图 11.1.3 `tbl` 数据结构拓扑图

`ipt_table` 各字段的意义如下：

`name`: 其所属的表，对于 `filter` 表这里会赋值为 `filter`；

`valid_hooks`: 该表能够作用的 hook 点，对于 `filter` 表，有 `IN`、`FORWARD` 和 `OUT` 三个 hook 点。

`num_entries`: `entry` 的入口点数目，为 hook 数目加一。

`Size`: 所有 `entry` 项的 `size` 之和。

`hook_entry`: hook 点的入口项。

`Underflow`: `underflow` 入口点。

`num_counters`、`counters`: 兼容旧的 netfilter 所用。

`entries`: hook 入口项。

`tbl` 的初始化的结果如图 11.1.4 所示，图中的 `LEN` 是为缩小图像大小而自己标记的一个值，对应的三个 hook 点和它的 hook 入口项，图中棕色字段均为初始化过程中设置的值。

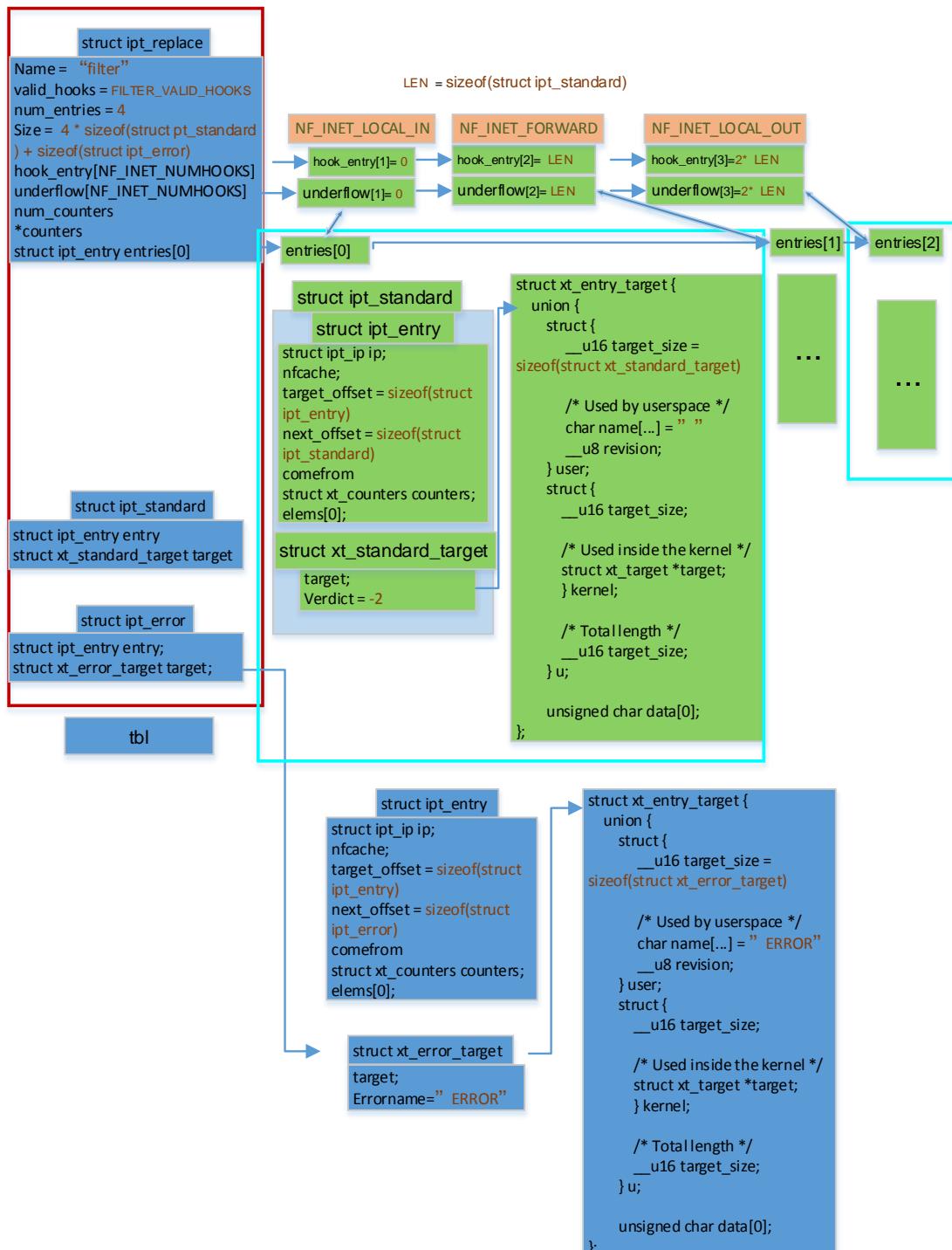


图 11.1.4 tbl 初始化后各字段初始值

ipt_register_table 用于向内核注册 filter 表，

```

2058 struct xt_table *ipt_register_table(struct net *net,
2059                                     const struct xt_table *table,
2060                                     const struct ipt_replace *repl)
2061 {

```

```

2062     int ret;
2063     struct xt_table_info *newinfo;
2064     struct xt_table_info bootstrap = {0};
2065     void *loc_cpu_entry;
2066     struct xt_table *new_table;
2067
2068     newinfo = xt_alloc_table_info(repl->size);
2069     if (!newinfo) {
2070         ret = -ENOMEM;
2071         goto out;
2072     }
2073
2074     /* choose the copy on our node/cpu, but dont care about preemption */
2075     loc_cpu_entry = newinfo->entries[raw_smp_processor_id()];
2076     memcpy(loc_cpu_entry, repl->entries, repl->size);
2077
2078     ret = translate_table(net, newinfo, loc_cpu_entry, repl);
2079     if (ret != 0)
2080         goto out_free;
2081
2082     new_table = xt_register_table(net, table, &bootstrap, newinfo);
2083     if (IS_ERR(new_table)) {
2084         ret = PTR_ERR(new_table);
2085         goto out_free;
2086     }
2087
2088     return new_table;
2089 }

```

2068 行, 根据前面 repl 的 size 大小为每一个核申请内存, 如果 repl 的 size 大于一个页, 内核会使用 vmalloc_node 申请, 否则使用 kmalloc_node 申请。

2075 行, 获得 SMP 情况下, 本地 cpu 的入口地址;

2076 行, 将图 11.1.4 中的三个 entries 拷贝到本地 cpu 的表中。

2078 行, 将 repl 的相关信息复制到 newinfo 中, 因为内核中防火墙的表是由 xt_table 表示的而 xt_table_info 描述防火墙表自身信息的。

2082 行, xt_register_table 用于将表注册到 struct net 的 struct netns_xt xt; 字段的链表上去, 并将 xt_table 的 prive 字段设置成描述表信息的 xt_table_info 类型的成员。最后返回生成的表。该返回的表由于是 ipv4 协议的, 所以在 struct net 表示的网络中将其成员 struct netns_ipv4 ipv4 的 iptable_filter 成员赋值成生成的表。

xt_hook_link 创建和注册了 filter 表的 hook 函数, 这里的 hook 函数是 iptable_filter_hook。

```

struct nf_hook_ops *xt_hook_link(const struct xt_table *table, nf_hookfn *fn)
{
    unsigned int hook_mask = table->valid_hooks;
    uint8_t i, num_hooks = hweight32(hook_mask);
    uint8_t hooknum;
    struct nf_hook_ops *ops;

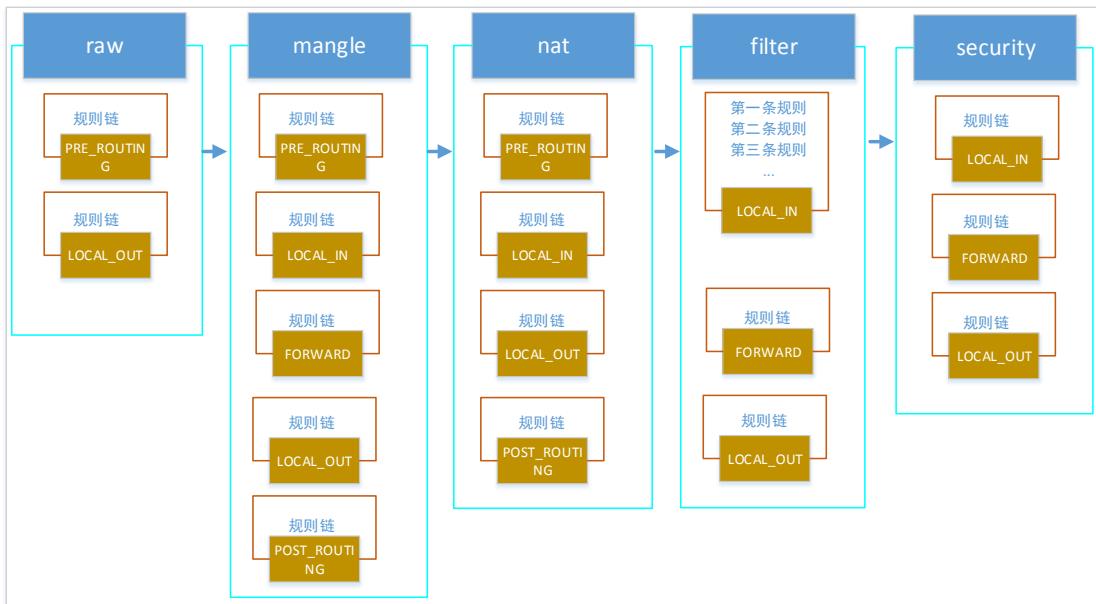
    for (i = 0, hooknum = 0; i < num_hooks && hook_mask != 0;
         hook_mask >>= 1, ++hooknum) {
        if (!(hook_mask & 1))
            continue;
        ops[i].hook      = fn;
        ops[i].owner     = table->me;
        ops[i].pf        = table->af;
        ops[i].hooknum   = hooknum;
        ops[i].priority = table->priority;
        ++i;
    }
    ret = nf_register_hooks(ops, num_hooks);
}

```

这里注册的 `iptable_filter_hook` 钩子函数不放在这个小节，其内容会放到规则表的遍历一节。

11.2 防火墙规则表

现行 Linux 下主要有 `raw`、`mangle`、`nat`、`filter` 和 `security` 这五张表，每张表各自又包含 `PRE_ROUTING`、`LOCAL_IN`、`FPRWARD`、`LOCAL_OUT` 和 `POST_ROUTING` 这五条链的一个组合，每一条链又会包含若干的规则项，源地址、目的地址、传输协议（TCP/UDP/ICMP）以及服务类型（HTTP、SNMP 等）就包含在这些规则项中，`raw` 用于数据跟踪处理、`mangle` 表用于数据包的重构、`nat` 表用于网络地址转换、`filter` 表用于包过滤、`security` 表是新引入的用于安全处理；目前这些表规则的用户空间的配置工具是 `iptables`，`iptables` 用于修改这些规则项，这些表的优先级按照图中从左至右的顺序递减。本小节主要分析下面这张表中的规则项。



11.2.1 表、链、规则关系

防火墙的每条规则由以下三个部分组成。

- `ipt_entry`
- `ipt_entry_match/xt_entry_match`
- `ipt_entry_target/xt_entry_targett`

防火墙的每一条规则由 `ipt_entry` 定义，每一条规则包括三个部分，1) IP 头 2) 和 match 相关的 3) 如果 match 和规则匹配则会被执行的 target。在 11.1 节我们已经见过 `ipt_entry` 了，并且在第一节中，有一个 `module_init` 宏定义的函数 `ip_tables_init`，其用于注册 netfilter 的 target 和 match 项。

```

2208 static int __net_init ip_tables_net_init(struct net *net)
2209 {
2210     return xt_proto_init(net, NFPROTO_IPV4);
2211 }
2218 static struct pernet_operations ip_tables_net_ops = {
2219     .init = ip_tables_net_init,
2220     .exit = ip_tables_net_exit,
2221 };
2223 static int __init ip_tables_init(void)
2224 {
2225     int ret;
2227     ret = register_pernet_subsys(&ip_tables_net_ops);
2231     /* No one else will be downing sem now, so we won't sleep */
2232     ret = xt_register_targets(ip_builtin_tg, ARRAY_SIZE(ip_builtin_tg));
2235     ret = xt_register_matches(ip_builtin_mt, ARRAY_SIZE(ip_builtin_mt));
2239     /* Register setsockopt */

```

```

2240     ret = nf_register_sockopt(&ipt_sockopts);
...
2255 }

```

2227 行会调用 ip_tables_net_init，该函数在 proc/net/ 目录下创建：“ip_tables_names”、“ip_tables_matches”、“ip_tables_targets”这三个文件。

11.2.1 xt_init 初始化防火墙表

防火墙的规则由 struct xt_af 类型的 xt 统一来管理，在系统初始化时会初始化该表。

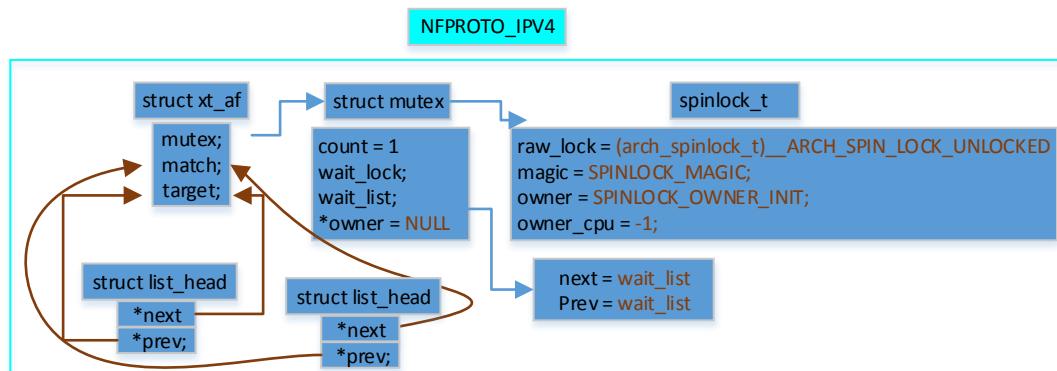
net/filter/x_tables.c

```

1372 static int __init xt_init(void)
1373 {
1381     xt = kmalloc(sizeof(struct xt_af) * NFPROTO_NUMPROTO, GFP_KERNEL);
1385     for (i = 0; i < NFPROTO_NUMPROTO; i++) {
1386         mutex_init(&xt[i].mutex);
1391         INIT_LIST_HEAD(&xt[i].target);
1392         INIT_LIST_HEAD(&xt[i].match);
1393     }
1394     rv = register_pernet_subsys(&xt_net_ops);
1398 }
1406 module_init(xt_init);

```

1381 行，根据支持的协议类型申请内存，这些协议包括 ipv4、arp、bridge、ipv6、DECNET 等。xt 的 ipv4 项初始化后数据结构拓扑如图 11.2.2，由于和其它协议使用同一个循环体完成的，所以其它协议的 xt 初始化和这里的类似。



11.2.2 ipv4 xt 初始化

1394 行是初始化网络空间中的防火墙表项。

```

static int __net_init xt_net_init(struct net *net)
{
    for (i = 0; i < NFPROTO_NUMPROTO; i++)
        INIT_LIST_HEAD(&net->xt.tables[i]);
}

```

这里的 xt 是 struct netns_xt 类型的，这里将该网络命名空间中的所有协议的表初始化一下。

回到 ip_tables_init 函数的 2232、2235 行接着看，其主要将下述定义的 ipt_builtin_tg 和 ipt_builtin_mt 添加到 xt 链表上去，这是 xt 表的默认项。链表的结构图见图 11.2.3。

```
2161 static struct xt_target ipt_builtin_tg[] __read_mostly = {  
2162     {  
2163         .name          = XT_STANDARD_TARGET,  
2164         .targetsize    = sizeof(int),  
2165         .family        = NFPROTO_IPV4,  
2166     },  
2167     {  
2168         .name          = XT_ERROR_TARGET,  
2169         .target        = ipt_error,  
2170         .targetsize    = XT_FUNCTION_MAXNAMELEN,  
2171         .family        = NFPROTO_IPV4,  
2172     },  
2173 };  
2174  
2175 static struct xt_match ipt_builtin_mt[] __read_mostly = {  
2176     {  
2177         .name          = "icmp",  
2178         .match         = icmp_match,  
2179         .matchsize     = sizeof(struct ipt_icmp),  
2180         .checkentry    = icmp_checkentry,  
2181         .proto         = IPPROTO_ICMP,  
2182         .family        = NFPROTO_IPV4,  
2183     },  
2184 };
```

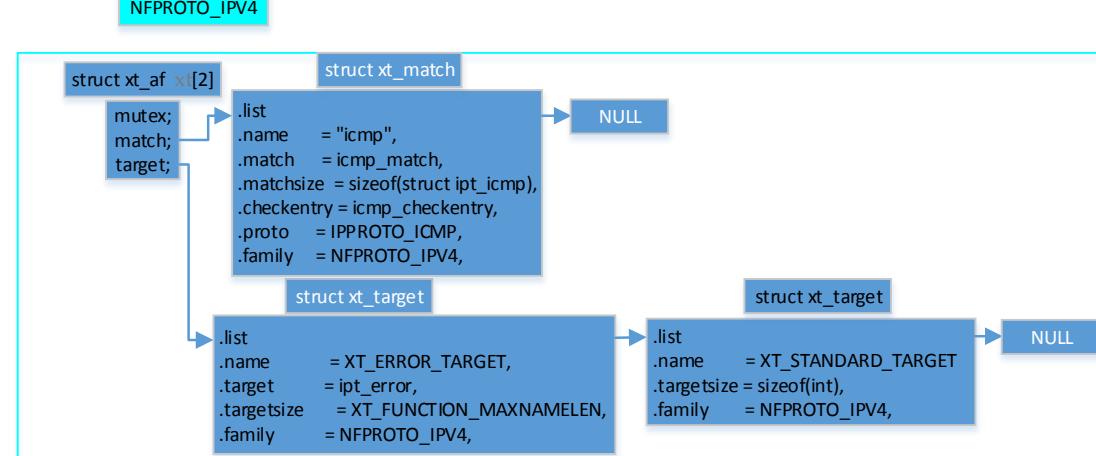


图 11.2.4 链表结构拓扑图

11.2.2 规则的组成

规则依次存放，在 `ipt_entry` 中，其 `target_offset` 和 `next_offset` 字段分别标记了 target 偏移和下一个规则的偏移，在初始化处可以看到它们实现的细节。

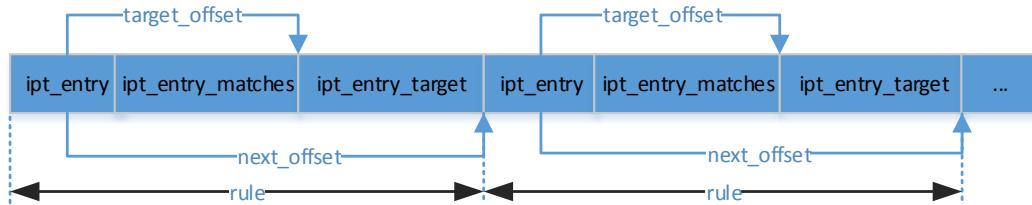


图 11.2.5 规则拓扑

//该结构体包含源地址、目的地址、接口、协议等相关信息，这也是一条规则需要的信息。

```
69 struct ipt_ip {  
70     /* Source and destination IP addr */  
71     struct in_addr src, dst;      //它实际上就是大端格式的 32bit 无符号整数。  
72     /* Mask for src and dest IP addr */  
73     struct in_addr smsk, dmsk;  
74     char iniface[IFNAMSIZ], outiface[IFNAMSIZ];  
75     unsigned char iniface_mask[IFNAMSIZ], outiface_mask[IFNAMSIZ];  
76  
77     /* Protocol, 0 = ANY */  
78     __u16 proto;  
79  
80     /* Flags word */  
81     __u8 flags;  
82     /* Inverse flags */  
83     __u8 invflags;  
84 };  
  
include/uapi/linux/netfilter/x_tables.h  
  
10 struct xt_entry_match {  
11     union {  
12         struct {  
13             __u16 match_size;  
14  
15             /* Used by userspace */  
16             char name[XT_EXTENSION_MAXNAMELEN];  
17             __u8 revision;  
18         } user;
```

```

19     struct {
20         __u16 match_size;
21
22         /* Used inside the kernel */
23         struct xt_match *match;
24     } kernel;
25
26         /* Total length */
27         __u16 match_size;
28     } u;
29
30     unsigned char data[0];
31 };

```

include/linux/netfilter/x_tables.h

```

105 struct xt_match {
106     struct list_head list;
107
108     const char name[XT_EXTENSION_MAXNAMELEN];
109     u_int8_t revision;
110
111     /* Return true or false: return FALSE and set *hotdrop = 1 to
112         force immediate packet drop. */
113     /* Arguments changed since 2.6.9, as this must now handle
114         non-linear skb, using skb_header_pointer and
115         skb_ip_make_writable. */
116     bool (*match)(const struct sk_buff *skb,
117                   struct xt_action_param *);
118
119     /* Called when user tries to insert an entry of this type. */
120     int (*checkentry)(const struct xt_mtchk_param *);
121
122     /* Called when entry of this type deleted. */
123     void (*destroy)(const struct xt_mtdtor_param *);
124 #ifdef CONFIG_COMPAT
125     /* Called when userspace align differs from kernel space one */
126     void (*compat_from_user)(void *dst, const void *src);
127     int (*compat_to_user)(void __user *dst, const void *src);

```

```

128 #endif

129     /* Set this to THIS_MODULE if you are a module, otherwise NULL */

130     struct module *me;

131

132     const char *table;

133     unsigned int matchsize;

137     unsigned int hooks;

138     unsigned short proto;

139

140     unsigned short family;

141 };

```

11.3 防火墙规则遍历

防火墙遍历规则链中的具体规则的入口函数是 `ipt_do_table`, 内核使用图 11.3.1 所示的 6 个函数调用该入口函数。对于 ipv4 而言, `iptable_filter_hook` 是其入口的直接函数。

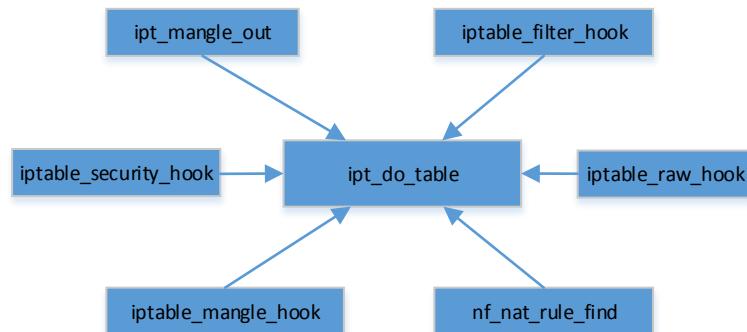


图 11.3.1 防火墙规则遍历点

```

35 static unsigned int
36 iptable_filter_hook(unsigned int hook, struct sk_buff *skb,
37     const struct net_device *in, const struct net_device *out,
38     int (*okfn)(struct sk_buff *))
39 {
42     if (hook == NF_INET_LOCAL_OUT &&
43         (skb->len < sizeof(struct iphdr) ||
44          ip_hdrlen(skb) < sizeof(struct iphdr)))
45     /* root is playing with raw sockets. */
46     return NF_ACCEPT;
48     net = dev_net((in != NULL) ? in : out);
49     return ipt_do_table(skb, hook, in, out, net->ipv4.iptable_filter);
}

```

42~46 对于 raw 类型的数据包, 直接放行;

48 行，如果 in 非空，则说明是入数据包，否则是出数据包。

49 行，调用 `ipt_do_table` 查找规则链。

`ipt_do_table` 参数的意义如下：

- `skb`, 对应的 socket buffer
- `in` 到达数据包的设备，如果是发送操作，则这里是 `NULL`
- `out` 发送数据包的设备，如果是接收操作，这里将是 `NULL`
- `table`, 防火墙的表，`ipv4` 则是初始化时的 `net->ipv4.iptable_filter` 表

```
288 unsigned int
289 ipt_do_table(struct sk_buff *skb,
290         unsigned int hook,
291         const struct net_device *in,
292         const struct net_device *out,
293         struct xt_table *table)
294 {
295     static const char nulldevname[IFNAMSIZ] __attribute__((aligned(sizeof(long))));
296     const struct iphdr *ip;
297     /* Initializing verdict to NF_DROP keeps gcc happy. */
298
/* verdict 是遍历规则以后返回的值，可能的值如下。
#define NF_DROP 0
#define NF_ACCEPT 1
#define NF_STOLEN 2
#define NF_QUEUE 3
#define NF_REPEAT 4
#define NF_STOP 5
*/
298     unsigned int verdict = NF_DROP;
299     const char *indev, *outdev;
300     const void *table_base;
301     struct ipt_entry *e, **jumpstack;
302     unsigned int *stackptr, origptr, cpu;
303     const struct xt_table_info *private;
304     struct xt_action_param acpar;
305     unsigned int addend;
306
307     /* Initialization */
308     ip = ip_hdr(skb);
309     indev = in ? in->name : nulldevname;
310     outdev = out ? out->name : nulldevname;
```



```

348 no_match:
349         e = ipt_next_entry(e);
350         continue;
351     }
352
353     xt_ematch_foreach(ematch, e) {
354         acpar.match      = ematch->u.kernel.match;
355         acpar.matchinfo = ematch->data;
356         if (!acpar.match->match(skb, &acpar))
357             goto no_match;
358     }
359
360     ADD_COUNTER(e->counters, skb->len, 1);
361
362     t = ipt_get_target(e);
363     IP_NF_ASSERT(t->u.kernel.target);
364     /* Standard target? */
365     if (!t->u.kernel.target->target) {
366         int v;
367
368         v = ((struct xt_standard_target *)t)->verdict;
369         if (v < 0) {
370             /* Pop from stack? */
371             if (v != XT_RETURN) {
372                 verdict = (unsigned int)(-v) - 1;
373                 break;
374             }
375             if (*stackptr <= origptr) {
376                 e = get_entry(table_base,
377                               private->underflow[hook]);
378                 pr_debug("Underflow (this is normal) "
379                         "to %p\n", e);
380             } else {
381                 e = jumpstack[--*stackptr];
382                 pr_debug("Pulled %p out from pos %u\n",
383                         e, *stackptr);
384             }
385             e = ipt_next_entry(e);

```

```

392         }
393         continue;
394     }
395     if (table_base + v != ipt_next_entry(e) &&
396         !(e->ip.flags & IPT_F_GOTO)) {
397         if (*stackptr >= private->stacksize) {
398             verdict = NF_DROP;
399             break;
400         }
401         jumpstack[(*stackptr)++] = e;
402         pr_debug("Pushed %p into pos %u\n",
403                 e, *stackptr - 1);
404     }
405
406     e = get_entry(table_base, v);
407     continue;
408 }
409
410 acpar.target = t->u.kernel.target;
411 acpar.targinfo = t->data;
412
413 verdict = t->u.kernel.target->target(skb, &acpar);
414 /* Target might have changed stuff. */
415 ip = ip_hdr(skb);
416 if (verdict == XT_CONTINUE)
417     e = ipt_next_entry(e);
418 else
419     /* Verdict */
420     break;
421 } while (!acpar.hotdrop);
422 pr_debug("Exiting %s; resetting sp from %u to %u\n",
423         __func__, *stackptr, origptr);
424 *stackptr = origptr;
425 xt_write_recseq_end(addend);
426 local_bh_enable();
435 }

```

该函数的返回值是防火墙对套接字 buffer 中数据包匹配规则后的结果，有以下几种可能：

```

#define NF_DROP 0
#define NF_ACCEPT 1
#define NF_STOLEN 2
#define NF_QUEUE 3
#define NF_REPEAT 4
#define NF_STOP 5
#define NF_MAX_VERDICT NF_STOP

```

317~323 根据 得到的数据包头 得到的数据包头 得到的数据包头 得到的数据包头 , 将分段 、 头长 等字段 等字段 保存在 保存在 `acpar acpar acpar` 变量 中。

325 验证 `hook hookhook` 点是否合法。 点是否合法。 点是否合法。 点是否合法。

328`private` 是 `struct xt_table_info` 类型的变量, `ipv4` 使用 `translate_table` 函数获得表信息的, 这里将表信息存放在 `private` 字段。

329 获得本地 CPU 号。

330 获得本地 CPU 的防火墙规则表的入口点地址。

335 行, 根据得到的规则点地址和 `hook` 入口点, 获得入口项, 它们的关系可以见图 11.1.4。

```

static inline struct ipt_entry *
get_entry(const void *base, unsigned int offset)
{
    return (struct ipt_entry *)(base + offset);
}

```

这里 `base` 就是基地址, `offset` 值对应于图 11.1.4 中的 0、LEN、2*LEN; 其中 LEN 等于 `sizeof (ipt_standard)`;

341~421 遍历规则链中的规则表。

342~343`target` 和 `match` 的意义参看图 11.2.5 就能明白, `ematch` 用于规则匹配和匹配上后执行的 `target` 动作。

346~347 判断规则的匹配性了, 其第一个参数是根据 `skb buffer` 获得的 `ip` 头, 第四个参数是一个规则指向的 `ip` 向, 第五个参数是分片标志。

```

74 static inline bool
75 ip_packet_match(const struct iphdr *ip,
76         const char *indev,
77         const char *outdev,
78         const struct ipt_ip *ipinfo,
79         int isfrag)
80 {
81     unsigned long ret;
82
83 #define FWINV(bool, invflg) ((bool) ^ !!(ipinfo->invflags & (invflg)))
84
85     if (FWINV((ip->saddr&ipinfo->smsk.s_addr) != ipinfo->src.s_addr,

```

```

86     IPT_INV_SRCIP) ||
87     FWINV((ip->daddr&ipinfo->dmsk.s_addr) != ipinfo->dst.s_addr,
88     IPT_INV_DSTIP)) {
89     dprintf("Source or dest mismatch.\n");
90
91     dprintf("SRC: %pI4. Mask: %pI4. Target: %pI4.%s\n",
92             &ip->saddr, &ipinfo->smsk.s_addr, &ipinfo->src.s_addr,
93             ipinfo->invflags & IPT_INV_SRCIP ? "(INV)" : "");
94     dprintf("DST: %pI4 Mask: %pI4 Target: %pI4.%s\n",
95             &ip->daddr, &ipinfo->dmsk.s_addr, &ipinfo->dst.s_addr,
96             ipinfo->invflags & IPT_INV_DSTIP ? "(INV)" : "");
97     return false;
98 }
99
100    ret = ifname_compare_aligned(indev, ipinfo->iniface, ipinfo->iniface_mask);
101
102    if (FWINV(ret != 0, IPT_INV_VIA_IN)) {
103        dprintf("VIA in mismatch (%s vs %s).%s\n",
104                indev, ipinfo->iniface,
105                ipinfo->invflags&IPT_INV_VIA_IN ? "(INV)" : "");
106        return false;
107    }
108
109    ret = ifname_compare_aligned(outdev, ipinfo->outiface, ipinfo->outiface_mask);
110
111    if (FWINV(ret != 0, IPT_INV_VIA_OUT)) {
112        dprintf("VIA out mismatch (%s vs %s).%s\n",
113                outdev, ipinfo->outiface,
114                ipinfo->invflags&IPT_INV_VIA_OUT ? "(INV)" : "");
115        return false;
116    }
117
118    /* Check specific protocol */
119    if (ipinfo->proto &&
120        FWINV(ip->protocol != ipinfo->proto, IPT_INV_PROTO)) {
121        dprintf("Packet protocol %hi does not match %hi.%s\n",
122                ip->protocol, ipinfo->proto,

```

```

123         ipinfo->invflags&IPT_INV_PROTO ? "(INV)":"");
124     return false;
125 }
126
127 /* If we have a fragment rule but the packet is not a fragment
128 * then we return zero */
129 if (FWINV((ipinfo->flags&IPT_F_FRAG) && !isfrag, IPT_INV_FRAG)) {
130     dprintf("Fragment rule but not fragment.%s\n",
131             ipinfo->invflags & IPT_INV_FRAG ? "(INV)" : "");
132     return false;
133 }
134
135     return true;
136 }
```

85~98 匹配源地址和目的地址

100~116 匹配接收和发送接口，如 eth0、eth1 等

119~125 匹配协议

129~133 匹配分片

上述有任一项违反规则项，则说明规则并不匹配，则 349 行获得下一个规则，重复进行上述检查，直至获得一个匹配的规则。

353~358,根据规则项，遍历该规则项对应的 match 函数，使用 match 对 skb 进行检查，如果 match 了，则向下进行，否则继续遍历规则链。

360 统计验证过的数据字节数和 packet 数，字节数即 len，packet 就是简单加一。

362 根据 match 的项，获得对应的 target，将调用 target 对 skb 处理以决定其最后的命运。

363~408 标准类型的 target 执行流程。

413 用户定义类型的 target 执行，iptables 配置？

11.3.2 Hook 函数

结构体

```

static struct nf_hook_ops ipv4_defrag_ops[] = {
{
    .hook = ipv4_conntrack_defrag,
    .owner = THIS_MODULE,
    .pf = NFPROTO_IPV4,
    .hooknum = NF_INET_PRE_ROUTING,
    .priority = NF_IP_PRI_CONNTRACK_DEFRAG,
},
{
    .hook      = ipv4_conntrack_defrag,
    .owner      = THIS_MODULE,
```

```

(pf      = NFPROTO_IPV4,
.hooknum = NF_INET_LOCAL_OUT,
.priority = NF_IP_PRI_CONNTRACK_DEFRAG,
),
);

```

11.3.2.2 钩子函数注册

```

net/ipv4/netfilter/nf_defrag_ipv4.c

static int __init nf_defrag_init(void)
{
    return nf_register_hooks(ipv4_defrag_ops, ARRAY_SIZE(ipv4_defrag_ops));
}

int nf_register_hooks(struct nf_hook_ops *reg, unsigned int n)
{
    unsigned int i;
    int err = 0;

    for (i = 0; i < n; i++) {
        err = nf_register_hook(&reg[i]);
    }
}

int nf_register_hook(struct nf_hook_ops *reg)
{
    struct nf_hook_ops *elem;
    int err;

    err = mutex_lock_interruptible(&nf_hook_mutex);
    if (err < 0)
        return err;

    list_for_each_entry(elem, &nf_hooks[reg->pf][reg->hooknum], list) {
        if (reg->priority < elem->priority)
            break;
    }

    list_add_rcu(&reg->list, elem->list.prev);

    mutex_unlock(&nf_hook_mutex);
    return 0;
}

```

11.3.2.3 Hook 函数的调用实例

```
int ip_output(struct sk_buff*skb)
{
    struct net_device *dev= skb_dst(skb)->dev;

    IP_UPD_PO_STATS(dev_net(dev),IPSTATS_MIB_OUT, skb->len);

    skb->dev = dev;
    skb->protocol = htons(ETH_P_IP);

    return NF_HOOK_COND(NFPROTO_IPV4,NF_INET_POST_ROUTING, skb,NULL, dev,
                        ip_finish_output,
                        !(IPCB(skb)->flags& IPSKB_REROUTED));
}

static inline int
NF_HOOK_COND(uint8_t pf, unsigned int hook, struct sk_buff*skb,
            struct net_device*in, struct net_device*out,
            int(*okfn)(struct sk_buff*), bool cond)
{
    int ret;

    if (!cond ||
        ((ret = nf_hook_thresh(pf, hook, skb, in, out, okfn, INT_MIN)) == 1))
        ret = okfn(skb);

    return ret;
}
```

11.3.2.4 nf_hook_thresh

```
/**
 * nf_hook_thresh- call a netfilterhook
 *
 * Returns 1 if the hook has allowed the packet to pass. The function
 * okfn must be invoked by the caller in this case. Any other return
 * value indicates the packet has been consumed by the hook.
 */
static inline int nf_hook_thresh(u_int8_t pf, unsigned int hook,
                                struct sk_buff *skb,
```

```

• struct net_device *indev,
• struct net_device *outdev,
• int (*okfn)(struct sk_buff *), intthresh)

{

• if (nf_hooks_active(pf,hook))

• return nf_hook_slow(pf,hook,skb,indev,outdev,okfn,thresh);

• return 1;

}

/* Returns 1 if okfn() needs to be executed by the caller,
 * -EPERM for NF_DROP, 0 otherwise. */

int nf_hook_slow(u_int8_t pf, unsigned int hook, struct sk_buff *skb,
struct net_device *indev,
struct net_device *outdev,
int (*okfn)(struct sk_buff *),
int hook_thresh)

{
    struct nf_hook_ops *elem;
    unsigned int verdict;
    int ret = 0;

    /* We may already have this, but read-locks nest anyway */
    rcu_read_lock();
}

```

```

elem = list_entry_rcu(&nf_hooks[pf][hook], struct nf_hook_ops, list);

next_hook:

verdict = nf_iterate(&nf_hooks[pf][hook], skb, hook, indev,
                    outdev, &elem, okfn, hook_thresh);

if (verdict == NF_ACCEPT || verdict == NF_STOP) {

    ret = 1;

} else if ((verdict & NF_VERDICT_MASK) == NF_DROP) {

    kfree_skb(skb);

    ret = NF_DROP_GETERR(verdict);

    if (ret == 0)

        ret = -EPERM;

} else if ((verdict & NF_VERDICT_MASK) == NF_QUEUE) {

```

```

int err = nf_queue(skb, elem, pf, hook, indev, outdev, okfn,
verdict >> NF_VERDICT_QBITS);

if (err < 0) {
    if (err == -ECANCELED)
        goto next_hook;
    if (err == -ESRCH &&
        (verdict & NF_VERDICT_FLAG_QUEUE_BYPASS))
        goto next_hook;
    kfree_skb(skb);
}
}

rcu_read_unlock();

return ret;
}

```

11.3.2.4 ipv4 hook 函数调用点

- ./ipv4/ip_output.c:273: NF_HOOK(NFPROTO_IPV4, NF_INET_POST_ROUTING,
- ./ipv4/ip_output.c:289: NF_HOOK(NFPROTO_IPV4, NF_INET_POST_ROUTING, newskb,
- ./ipv4/ip_output.c:100: return nf_hook(NFPROTO_IPV4, NF_INET_LOCAL_OUT, skb, NULL,
- ./ipv4/ipmr.c:1780: NF_HOOK(NFPROTO_IPV4, NF_INET_FORWARD, skb, skb->dev, dev,
- ./ipv4/ip_forward.c:183: return NF_HOOK(NFPROTO_IPV4, NF_INET_FORWARD, skb, skb->dev, dev,
- ./ipv4/xfrm4_input.c:64: NF_HOOK(NFPROTO_IPV4, NF_INET_PRE_ROUTING, skb, skb->dev, NULL,
- ./ipv4/ip_input.c:255: return NF_HOOK(NFPROTO_IPV4, NF_INET_LOCAL_IN, skb, skb->dev, NULL,
- ./ipv4/ip_input.c:445: return NF_HOOK(NFPROTO_IPV4, NF_INET_PRE_ROUTING, skb, dev, NULL,
- ./ipv4/arp.c:686: NF_HOOK(NFPROTO_ARP, NF_ARP_OUT, skb, NULL, skb->dev, dev_queue_xmit);
- ./ipv4/arp.c:967: return NF_HOOK(NFPROTO_ARP, NF_ARP_IN, skb, dev, NULL, arp_process);
- ./ipv4/raw.c:398: err = NF_HOOK(NFPROTO_IPV4, NF_INET_LOCAL_OUT, skb, NULL,

```

include/linux/netfilter.h

extern struct list_head nf_hooks[NFPROTO_NUMPROTO][NF_MAX_HOOKS];

include/uapi/linux/netfilter.h

enum nf_inet_hooks {
    NF_INET_PRE_ROUTING,
    NF_INET_LOCAL_IN,
    NF_INET_FORWARD,
    NF_INET_LOCAL_OUT,
    NF_INET_POST_ROUTING,

```

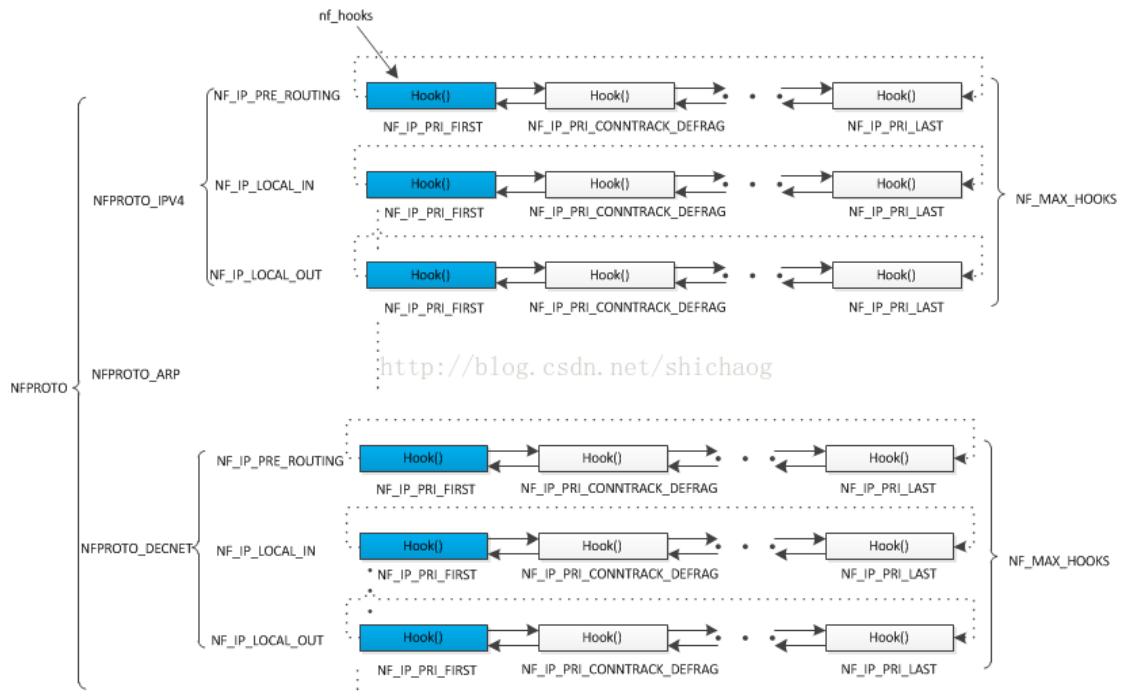
```

    NF_INET_NUMHOOKS
};

enum {
    NPROTO_UNSPEC = 0,
    NPROTO_IPV4 = 2,
    NPROTO_ARP = 3,
    NPROTO_BRIDGE = 7,
    NPROTO_IPV6 = 10,
    NPROTO_DECNET = 12,
    NPROTO_NUMPROTO,
};

}

```



NF_HOOK 宏的参数分别为：

[1]:NF_IP_PRE_ROUTING: 刚刚进入网络层的数据包通过此点（刚刚进行完版本号，校验和等检测），目的地址转换在此点进行；

[2]:NF_IP_LOCAL_IN: 经路由查找后，送往本机的通过此检查点，INPUT 包过滤在此点进行；

[3]:NF_IP_FORWARD: 要转发的包通过此检测点，FORWARD 包过滤在此点进行；

[4]:NF_IP_POST_ROUTING: 所有即将通过网络设备出去的包通过此检测点，内置的源地址转换功能（包括地址伪装）在此点进行；

[5]:NF_IP_LOCAL_OUT: 本机进程发出的包通过此检测点，OUTPUT 包过滤在此点进行。

NF_HOOK 宏的参数分别为：

1. pf: 协议族名，netfilter 架构同样可以用于 IP 层之外，因此这个变量还可以有诸如 PF_INET6, PF_DECnet 等名字。

2.hook: HOOK 点的名字，对于 IP 层，就是取上面的五个值；
3(skb: 不用多解释了吧；
4.indev: 进来的设备，以 struct net_device 结构表示；
5.outdev: 出去的设备，以 struct net_device 结构表示；
(后面可以看到，以上五个参数将传到用 nf_register_hook 登记的处理函数中。)
6.okfn: 是个函数指针，当所有的该 HOOK 点的所有登记函数调用完后，转而走此流程。
这些点是已经在内核中定义好的，除非你是这部分内核代码的维护者，否则无权增加或修改，而在此检测点进行的处理，则可由用户指定。像 packet filter,NAT,connection track 这些功能，也是以这种方式提供的。正如 netfilter 的当初的设计目标——提供一个完善灵活的框架，为扩展功能提供方便。
如果我们想加入自己的代码，便要用 nf_register_hook 函数，其函数原型为：
`int nf_register_hook(struct nf_hook_ops *reg)`

11.4 iptables

`iptables` 组件是一种工具，也称为用户空间（userspace），它使插入、修改和除去信息包过滤表中的规则变得容易。

从 3.13 版本开始，内核引入 `nftables`，该工具的作用和 `iptables` 类似，该工具设计的目的是替代 `iptables`，正处于开发循环周期中，可以参考 howto 指导说明。网址：
<https://home.regit.org/netfilter-en/nftables-quick-howto/>

邻居协议

rfc1812，在 ipv4 中使用 arp 协议，ipv6 使用 ndp，邻居发现协议；邻居协议作用于 L3 到 L2 层，用于将 IP 地址转换为 MAC 地址，这过程涉及到 icmp 协议包的收发。邻居协议的通用框架是 neighbor subsystem，ARP 和 NDP 协议在其上面。

第十二章 路由

12.1 路由核心数据结构

路由分为策略路由和多路路由，策略路由会参考用户设置的一些路由策略。路由可以使用tcp/ip分析的下篇系列文章之一中的route和ip route工具设置。路由分为路由部分包括三大块，路由缓存、路由表、路由信息查找。策略路由常用于安全和统计(经济)方面。多路路由允许对于一个给定的目的地址分配多个下一跳入口。这常被用于主备(可靠性、鲁棒性)路由。

路由表的构建途径：

通过用户命令[route(ioctl)、ip route(netlink)]静态配置

通过路由协议动态配置，这些协议是BGP（Border Gateway Protocol）、EGP（Exterior Gateway Protocol）以及OSPF（Open Shortest Path First）

这一章的内容基于route方法，其它的配置路由的方法不在这一章中，但是上面的方法区别在于配置方法，而对应调用的路由核心函数以及操作的核心路由数据结构是一样的，这一章的主要内容就是关于这些核心函数和核心数据结构的。

路由相关数据结构在include/net/route.h

```
struct ip_rt_acct {  
    __u32 o_bytes; //发送数据的字节数  
    __u32 o_packets;  
    __u32 i_bytes;  
    __u32 i_packets;  
};
```

这个结构体在ip_rcv_finish中被使用到，由于网络数据包的统计，分别按照byte和packet两种方法计数，ip_rcv_finish在网络层接收中分析过，这里会再一次看到在网络层被跳过的关于路由相关的代码，下面的代码片段就是上面统计信息被赋值的一个地方：

```
static int ip_rcv_finish(struct sk_buff *skb)  
{  
    #ifdef CONFIG_IP_ROUTE_CLASSID  
    if (unlikely(skb_dst(skb)->tclassid)) {  
        struct ip_rt_acct *st = this_cpu_ptr(ip_rt_acct);  
        u32 idx = skb_dst(skb)->tclassid;  
        st[idx&0xFF].o_packets++;  
        st[(idx>>16)&0xFF].o_bytes += skb->len;  
        st[(idx>>16)&0xFF].i_packets++;  
        st[(idx>>16)&0xFF].i_bytes += skb->len;  
    }  
    #endif  
}
```

由上面的使用可以知道，定义了基于路由的分类器就会使用该字段。该字段根据idx索引可构成具有256个成员的数组。其初始化在ip_rt_init中完成。

rt_cache_stat

路由表缓存的统计信息，除了输入输出路由信息统计，还有垃圾回收信息。

fib_result

查找路由表会得到此结构。

```
struct fib_result {  
    unsigned char prefixlen;  
    unsigned char nh_sel;  
    unsigned char type;
```

```
unsigned char scope;
u32 tclassid;
struct fib_info *fi;
struct fib_table *table;
struct list_head *fa_head;
};
```

struct fib_rule

策略路由使用的结构。

```
struct fib_rule {
struct list_headlist;
atomic_t refcnt;
int iifindex;
int oifindex;
u32 mark;
u32 mark_mask;
u32 pref;
u32 flags;
u32 table;
u8 action;
u32 target;
struct fib_rule __rcu*ctarget;
char iifname[IFNAMSIZ];
char oifname[IFNAMSIZ];
struct rcu_headrcu;
struct net * fr_net;
};
```

struct flowi

流量控制，作为路由表查找的键值。

```
struct flowi {
union {
struct flowi_common__fl_common;
struct flowi4 ip4;
struct flowi6 ip6;
struct flowidndn;
} u;
#define flowi_oif u.__fl_common.flowic_oif
#define flowi_iif u.__fl_common.flowic_iif
#define flowi_mark u.__fl_common.flowic_mark
#define flowi_tos u.__fl_common.flowic_tos
#define flowi_scope u.__fl_common.flowic_scope
#define flowi_proto u.__fl_common.flowic_proto
#define flowi_flags u.__fl_common.flowic_flags
#define flowi_secid u.__fl_common.flowic_secid
} __attribute__((__aligned__(BITS_PER_LONG/8)));

```

fib_table

路由表在内核中的表示为 fib_table 的一个结构体，其定义位于 include/net/ip_fib.h 文件。

```
struct fib_table {
struct hlist_nodetb_hlist;
u32 tb_id;
int tb_default;
int tb_num_default;
unsigned long tb_data[0];
};
```

tb_id 用于标识路由表所属，其可选字段在 include/uapi/linux/rtnetlink.h 文件中；

```
enum rt_class_t {
RT_TABLE_UNSPEC=0,
```

```

/* User defined values */
    RT_TABLE_COMPAT=252,
    RT_TABLE_DEFAULT=253,
    RT_TABLE_MAIN=254,
    RT_TABLE_LOCAL=255,
    RT_TABLE_MAX=0xFFFFFFFF
};


```

从枚举类型名称,如果没有使用策略路由,那么只有 `RT_TABLE_MAIN` 和 `RT_TABLE_LOCAL` 两种类型的路由表存在。

```
struct fib_info
```

多个路由项共享该一些字段:

```

struct fib_info {
    struct hlist_nodefib_hash;
    struct hlist_nodefib_lhash;
    struct net *fib_net;
    int fib_treeref;
    atomic_t fib_clntref;
    unsigned int fib_flags;
    unsigned char fib_dead;
    unsigned char fib_protocol;
    unsigned char fib_scope;
    unsigned char fib_type;
    __be32 fib_prefsrc;
    u32 fib_priority;
    u32 *fib_metrics;
#define fib_mtu fib_metrics[RTAX_MTU-1]
#define fib_window fib_metrics[RTAX_WINDOW-1]
#define fib_rtt fib_metrics[RTAX_RTT-1]
#define fib_advmss fib_metrics[RTAX_ADVMS-1]
    int fib_nhs;
#ifdef CONFIG_IP_ROUTE_MULTIPATH
    int fib_power;
#endif
    struct rcu_headrcu;
    struct fib_nh fib_nh[0];
#define fib_dev fib_nh[0].nh_dev
};


```

路由项别名:

```

struct fib_alias {
    struct list_headfa_list;
    struct fib_info*fa_info;
    u8 fa_tos;
    u8 fa_type;
    u8 fa_state;
    struct rcu_headrcu;
};


```

下一跳, 使用 `route` 或者 `ip route` 可以添加。

```

struct fib_nh {
    struct net_device*nh_dev;
    struct hlist_nodenh_hash;
    struct fib_info*nh_parent;
    unsigned int nh_flags;
    unsigned char nh_scope;
#ifdef CONFIG_IP_ROUTE_MULTIPATH
    int nh_weight;
    int nh_power;

```

```

#endif
#ifndef CONFIG_IP_ROUTE_CLASSID
__u32 nh_tclassid;
#endif
int nh_oif;
__be32 nh_gw;
__be32 nh_saddr;
int nh_saddr_genid;
struct rtable __rcu * __percpu *nh_percpu_rth_output;
struct rtable __rcu *nh_rth_input;
struct fnhe_hash_bucket *nh_exceptions;
};

struct dst_entry 路由表入口项
struct dst_ops 路由入口项的操作函数集，如垃圾回收函数就在这里。
struct rtable 路由表
路由三大块：路由缓存、路由表、路由信息查找，这三大块依赖的重要数据结构在
路由子系统初始化时完成。
路由子系统初始化：
2655 int __init ip_rt_init(void)
2656 {
/*ip_rt_acct 字段的意义在前面*/
2659 #ifndef CONFIG_IP_ROUTE_CLASSID
2660     ip_rt_acct = __alloc_percpu(256 * sizeof(struct ip_rt_acct), __alignof__(struct ip_rt_acct));
2661     if (!ip_rt_acct)
2662         panic("IP: failed to allocate ip_rt_acct\n");
2663 #endif
//创建 rtable 大小的路由表缓存，这是上述路由三大块中的路由缓存使用的，没有使用 malloc 的原因是加速网络数据包的传递
2665     ipv4_dst_ops.kmem_cachep =
2666         kmem_cache_create("ip_dst_cache", sizeof(struct rtable), 0,
2667                         SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL);
2668
2669     ipv4_dst_blackhole_ops.kmem_cachep = ipv4_dst_ops.kmem_cachep;
/*路由项垃圾回收门限，对无效的路由项回收其占用内存的门限，初始设置门限为最大值*/
2677     ipv4_dst_ops.gc_thresh = ~0;
/*路由表膨胀的最大尺寸，#define INT_MAX ((int)(~0U>>1))*/
2678     ip_rt_max_size = INT_MAX;
/*****************/
注册两种类型的内核通知链，netdev_chain 和 inetaddr_chain，这两种通知链对应的回调函数是 inetdev_event 和
fib_inetaddr_event，设备的状态的改变（up、down、register 和 unregistere）会调用回调函数：
devinet_init---向--netdev_chain---添加--inetdev_event 回调函数； 回调函数完成设备的初始化、删除等操作
ip_fib_init---向--netdev_chain---添加--fib_netdev_event 回调函数； 回调函数完成该设备相关路由表的使能、禁止、刷新等操作。
    -向--inetaddr_chain---添加--fib_inetaddr_event 回调函数； 回调函数完成该设备相关路由表路由项的添加和删除
操作。
********************/
2680     devinet_init();
2681     ip_fib_init();
2696     return rc;
2697 }

```

12.2 LC-trie（字典树、单词查找树）

现在内核采用的是 **trie** 算法组织路由表，这里的 **trie** 其实就是 **tree** 的意思。本章会以 **ifconfig** 和 **route** 两个命令作为引子，以这两个例子详细看一下 **trie** 路由算法在 Linux 下的实现，本节是现在内核默认 **trie** 路由算法的一个简单的算法简介，并将 **trie** 路由算法的节点、

叶子和 Linux 内核下具体的数据结构对应起来。至于早期的哈希算法，这里就丝毫没有涉及了。

先从一个引子说起，如果让你使用百度词典查找 **apple** 这个单词，会发现在你输入一个字母后，其下拉栏会显示若干的备选单词。如图 12.2.1 显示的，会有 **appliance**、**apple** 等提示单词，问题来了，百度是按照什么规则给出的提示的呢？

- 1、首先这些单词必须遵循字母顺序，不能用户输入 **a**，下来栏中出现个 **b** 打头的单词。
- 2、首先这些词在字典里必须是存在的，或者是一些组织机构的，总而言之就是这个词目前是存在的。
- 3、这些词并没有按照 26 个字母表的顺序给出，图中很明显，**ace** 比 **and** 要排在前面，这里加入了词频（概率）权重因子。

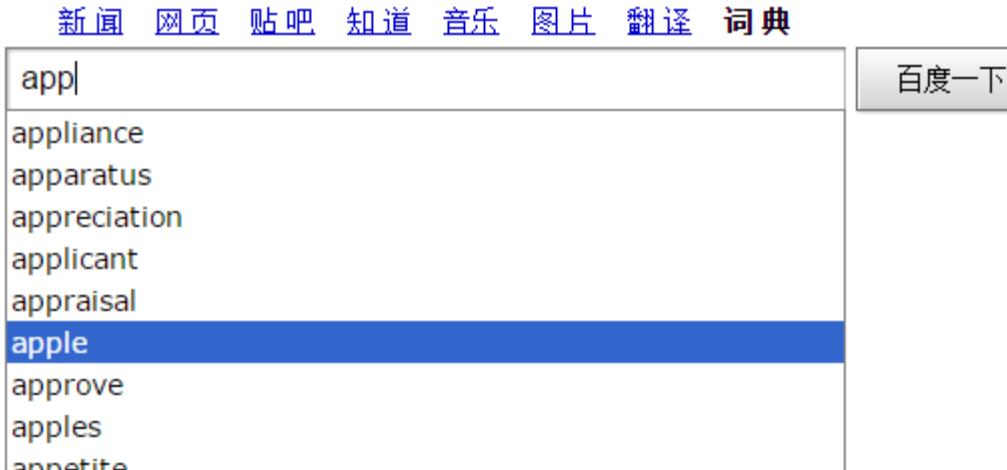


图 12.2.1 **apple** 字典提示

路由的算法就有点类似上面下拉栏的实现算法。但是上述频率的概念没有在路由算法本身体现，所以这里也就略过。下面还是以单词为例来看看 LC-trie 是如何组织的。图 12.2.1 中蓝色一栏就是我们要找的单词。

图 12.1.2 是对要查找的单词构建的一颗字典树，虚线左右两边都对应这个树，它们的不同在于深度。我们以左边的示例来说明字典树是如何组织的，对于 **apple** 这个单词，

- 1、树的根为空，NULL
- 2、最后一个字符是 **e**，**e** 被称为叶子
- 3、中间的字符，如 **a**、**p**、**l** 等，被称为节点。
- 4、尽量利用前缀节点，比如 **approve** 和 **apple** 有相同的前缀 **app**，黄色那个支路就是用来表示 **approve** 的。

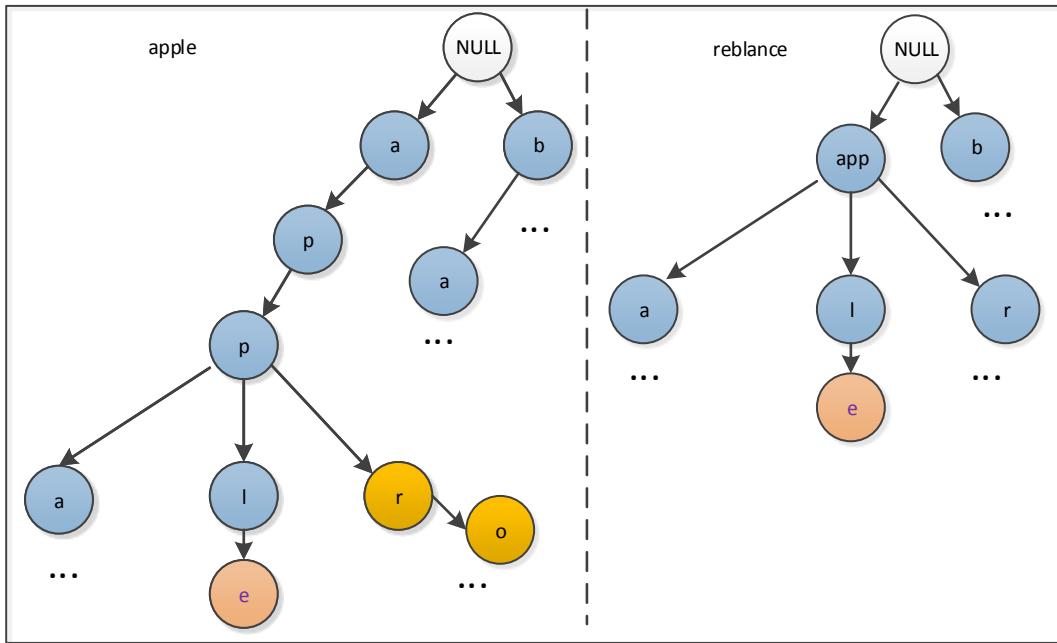


图 12.2.2 apple 字典树拓扑

图 12.2.2 中左右两幅图是用来说说明 rebalance 这个概念的，这棵树不能太瘦也不能太胖，也即其广度和深度要在一个合理的比率上。所以当我们觉得这棵树太瘦时，可以进行压缩，比如将 app 压缩成一个节点，这就意味着先前复用 a、ap 的节点会被创建，因为这时没有 a、ap 节点了。如果太胖，那就拉长，就是上述过程的逆过程。

上述的过程虽然和内核管理路由表的方法有点差别，但思想是一样的，图 12.1.3 是具有 10.12.39.0 和 192.168.0.10 两项的路由表。这个 10.12.39.0 并不是使用 route add 192.168.0.10 eth0 命令分配的，而是使用 ifconfig eth0 10.12.39.221 netmask 255.255.255.0 配置本机 IP 地址时设置的。至于为什么配置本机 IP 地址的 ifconfig 会设置这么一个路由项，主要原因是规范中要求：主机号全零的用于标识一个网段，这个地址会被用作路由项，不能被分配给主机使用，Linux 默认在设置主机 IP 时，都会配置一个主机号全零的 IP 作为路由项。这个内容在 12.2 节中会看到代码的实现。

图 12.2.4 是在图 12.2.3 的基础之上使用 route 命令添加一项组成的路由表拓扑图，需要说明的是这两幅图是路由项组织的核心结构，这里略去了其和路由缓存、arp 缓存、网络命名空间等之间的交互。图中黄色部分是数据结构，淡蓝色部分是对应数据结构的成员，成员的等号右边是其具有的值。

Struct fib_table: 代表一个路由表，

tb_hlist, 对于 ipv4 是`&net->ipv4.fib_table_hash[TABLE_MAIN_INDEX]`指向的哈希表，用于索引该路由项，对于本机 IP 地址由`&net->ipv4.fib_table_hash[TABLE_LOCAL_INDEX]`哈希表来指向。

tb_id 用于标识表的 ID 号，对于路由出去的表 id 是 254，本地 IP 表则 id 是 255。

tb_data[0] 零长数组，这个数组用于指向 struct trie 结构体，这个结构体就是对

struct rt_trie_node 结构体的封装，该结构体只有 **parent** 和 **key** 两个成员，红色框已经表明它们的所属关系。

struct tnode 结构体的前两项也是 **parent** 和 **key**，这就意味着可以使用强制类型转换方法和 **struct rt_trie_node** 互换使用。内核经常使用这个技巧。**tnode** 自身是 **trie node** 的意思，就是字典树的节点的意义，对应图 12.1.1 的蓝色节点，**tnode** 的最后一个成员是 ***child[0]**，这是又是一个零长数组，图中画出了 0 和 1 两个成员，实际上可能还有 2...，**full_children** 与

`empty_children` 之和等于零长数组成员的个数。

`Pos`: 比较起始位置

`Bits`: 比较的位数。

这两成员算是 `trie` 路由算法的核心成员, `pos` 指定了一个 32 位 ipv4 地址比较的起始位置, `bits` 指定了比较的比特数, 因为 IPV4 的地址是 32 位的, `pos` 和 `bits` 是 0~31 之间的数值。

由于 10.12.39.0 路由项先于 192.168.0.10 路由项存在这张路由表中, 当插入 192.168.0.10 时, 它会和根节点开始查找, 192 和 10 的二进制比较结果就是, 它们的第一个 bit 不同, 192 的第一个 bit 是 1, 而 10 的四位二进制的第一个 bit 是 0, 23bit 的 IPv4 地址使用 0~31 标记, 这就是 `tnode` 中 `pos` 等于 0, `bits` 等于 1 的由来。

`leaf` 对应图 12.2.3 中的黄色节点, 用于标记叶子节点。

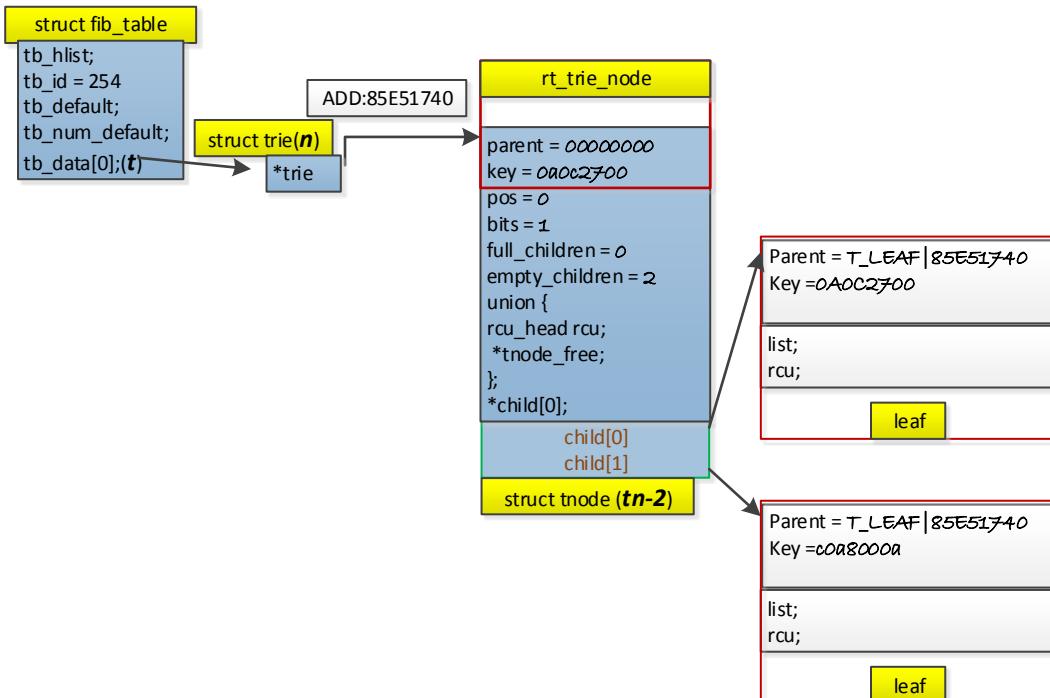


图 12.2.3 具有 10.12.39.0 和 192.168.0.10 两项的路由表

在插入 192.168.0.100 时, 很明显, 首先遍历 `tnode`, 找到 `pos` 是 0, `bits` 是 1 的一个节点 (实际上只有这么一个节点), `tnode` 的 `children`, 发现有相同的前缀 192.168.0 项, 即 192.168.0.100 和 192.168.0.10 从第 25 个 bit 不同, 并且不同的那个 bit 的值是 1, 这里会创建一个 `tnode`, 并将 `children` 赋值成适当的叶子。这一过程参看图 12.1.4, 和图 12.1.3 对比可以使这一过程更加明晰。当这棵树添加完成了以后, 最后会去判断这棵树是否需要 `rebalance`。

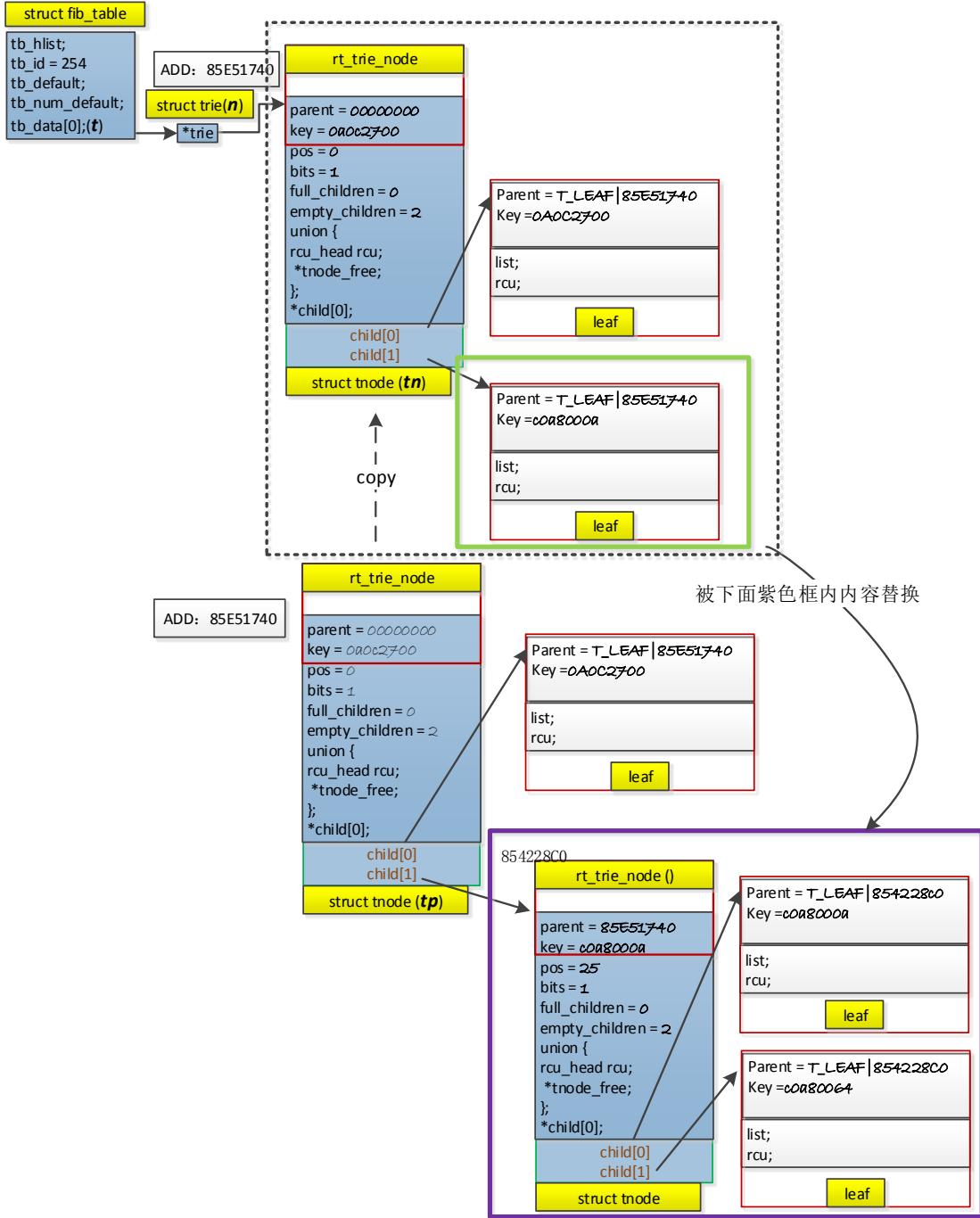


图 12.2.4 10.12.39.0/192.168.0.10/192.168.0.100 三项路由表项

对路由项的核心算法以及路由表项的组织有了一个了解以后，下面就正式进入源码及的剖析，由于代码的分支情况及其繁多，所以会以一个主线分析代码，即使主线不包括的代码也会注释它们的功能。

最后为了加深对 pos 和 bits 的奥妙了理解，这里给出 3 个 IP 地址分别是 10.12.39.0, 192.168.0.10 和 192.168.0.100 这三项，这三项和图 12.1.4 是对应的。插入的顺序是 10.12.39.0、192.168.0.10 和 192.168.0.100，插入 192.168.0.10 时，其发现和 10.12.39.0 的第 0 个比特不同，所以这时会将先前的 10.12.39.0 作为一个 tnode，tnode 的 pos 设置为 0，bits 设置成 1，黄色的那里只有一列，在插入 192.168.0.100 时，其和 192.168.0.10 有相同的前缀，所以 192.168.0.10 赋值到一个 tnode，其 pos 设置为 25，bits 设置为 1。查找时先遍

历 tnode，在遍历 leaf，这个过程会在后面结合具体代码来看。图中虚线右边是 rebalance 的一个示例。

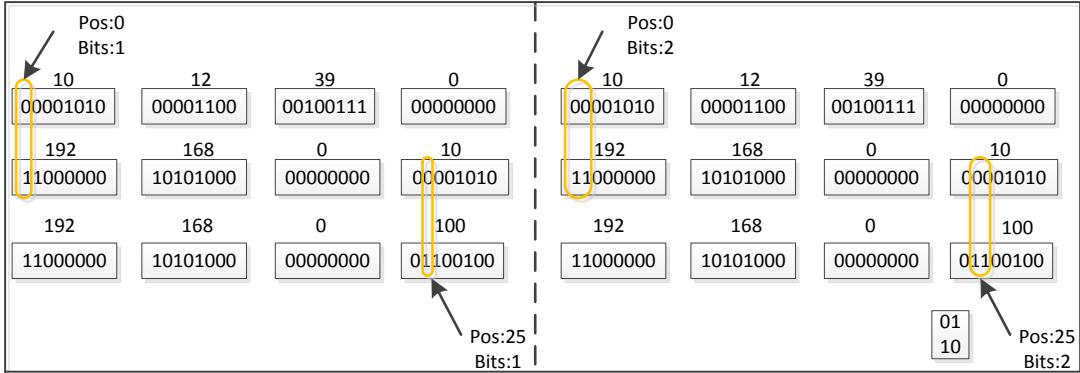


图 12.2.5 10.12.39.0/192.168.0.10/192.168.0.100 三项路由表的 pos 和 bits

12.3 ifconfig

12.3.1 /proc/net/路由下路由信息

在/proc/net/目录下 fib_trie 和 fib_triestat 这两个文件，这两文件包含了一些 trie 路由的信息。fib_trie 用于显示路由表的树状图，fib_triestat 是 trie 树的一些统计信息。

没有分配 IP 地址时的路由信息：

```
# cat /proc/net/fib_trie
# cat /proc/net/fib_triestat
Basic info: size of leaf: 20 bytes, size of tnode: 28 bytes.
Local:
    Aver depth:      0.00
    Max depth:       0
    Leaves:          0
    Prefixes:         0
    Internal nodes:  0

    Pointers: 0
    Null ptrs: 0
    Total size: 0 kB
Main:
    Aver depth:      0.00
    Max depth:       0
    Leaves:          0
    Prefixes:         0
    Internal nodes:  0

    Pointers: 0
    Null ptrs: 0
    Total size: 0 kB
```

没有分配 IP 地址时的路由信息都是空的，这很合逻辑，在使用 ifconfig eth0 10.12.39.221 netmask 255.255.255.0 配置本机 IP 时，路由信息的内容非常的多。在 fib_trie 中可以看到有 Local 和 Main 这两个字段，这两个字段分别对应于两张表，Local 用于表示自己的 IP 地址，而 Main 用于路由出去的 IP 地址。

在配置 10.12.39.221 时，LOCAL 表中包含了主机号全零和主机号全一的 IP 地址，这都是协议上规定的特殊地址。

第 3 行 10.12.39.0/24，这里的 24 对应于 12.2 节中 trie 字段的 pos，这里 10.12.39.0 和 10.12.192 的第一个不同的比特就是第 24 个（从 0 位置计）比特。同样第 6 行中的 26 也是根据 10.12.39.221 和 10.12.39.255 之间的差别。有“+”的行，表明其是一个 tnode（节点），“|”则表明了其是一个 leaf（叶子）。每个叶子下面“/”的标记了该叶子的属性，host、LOCAL 以及 BROADCAST 是表的类型，每对应这么一项，意味着插入了一次，比如这里的叶子 10.12.39.0 既属于 link 类型又属于 BROADCAST 类型，这两种类型的表项会对应于两次插入路由表的动作。

`fib_triestat` 统计了树的一些统计信息，在 12.2 节中所述的 rebalance 操作就是依赖这些统计信息对树进行均衡的。

```
1# cat /proc/net/fib_trie
2Local:
3  +- 10.12.39.0/24 1 0 0
4    |-- 10.12.39.0
5      /32 link BROADCAST
6      +- 10.12.39.192/26 1 0 0
7        |-- 10.12.39.221
8          /32 host LOCAL
9          |-- 10.12.39.255
10         /32 link BROADCAST
11Main:
12   |-- 10.12.39.0
13     /24 link UNICAST

15# cat /proc/net/fib_triestat
16Basic info: size of leaf: 20 bytes, size of tnode: 28 bytes.
17Local:
18  Aver depth:    1.66
19  Max depth:    2
20  Leaves:       3
21  Prefixes:     3
22  Internal nodes: 2
23    1: 2
24  Pointers:    4
25Null ptrs: 0
26Total size: 1 kB
27Main:
28  Aver depth:    0.00
29  Max depth:    0
30  Leaves:       1
31  Prefixes:     1
32  Internal nodes: 0
33
34  Pointers:    0
35Null ptrs: 0
36Total size: 1 kB
```

12.3.2 路由通知链函数的注册

前面已经见到过 `ip_fib_init` 函数，这个函数在 `net/ipv4/fib_frontend.c` 文件中，并且在初始化时会被调用。

```
1075 static struct notifier_block fib_inetaddr_notifier = {
1076   .notifier_call = fib_inetaddr_event,
1077 };
1168 void __init ip_fib_init(void)
1169 {
```

```

1170     rtnl_register(PF_INET, RTM_NEWRROUTE, inet_rtm_newroute, NULL, NULL);
1171     rtnl_register(PF_INET, RTM_DELROUTE, inet_rtm_delroute, NULL, NULL);
1172     rtnl_register(PF_INET, RTM_GETROUTE, NULL, inet_dump_fib, NULL);
1173
1174     register_pernet_subsys(&fib_net_ops);
1175     register_netdevice_notifier(&fib_netdev_notifier);
1176     register_inetaddr_notifier(&fib_inetaddr_notifier);
1177
1178     fib_trie_init();
1179 }

```

其 1176 行注册的结构体在 1075 行，这节就以注册的 `fib_inetaddr_event` 函数开始，当使用 `ifconfig eth0 10.12.39.221 netmask 255.255.255.0` 设置设备 IP 地址时，这个函数就会被触发调用。1175 行的 `fib_netdev_notifier` 函数也会在该设备获得地址的时候出发调用，进而会配置 12.3.1 节中那些未指定 IP 地址。

12.3.3 ifconfig 调用流程

`ifconfig` 配置 IP 地址的函数调用，第一个内核里被调用到的函数是 `inet_ioctl()`，经过 `devinet_ioctl` 函数会调用 `fib_inetaddr_event()`:

```

net/ipv4/af_inet.c inet_ioctl()
net/ipv4/devinet.c devinet_ioctl()

```

`fib_inetaddr_event()` 这个函数是在上一节注册的通知链函数，从这个函数的命名（`fib forward information base`）就可以看出其和路由是有直接关系关系的，这节就从这个函数开始。

好了从现在开始就要正式接触 Linux 内核网络路由代码细节了。由 12.3.1 节可知，一个 `ifconfig` 命令会在 `Local` 表中配置三项路由项，在 `Main` 表中配置一项路由项，本节所述内容是 `ifconfig` 在 `Local` 表创建路由项所经历的代码片段，当然在操作 `Main` 表时用的都是同一套代码。

图 12.3.1 给出的是的 `ifconfig` 在创建一个本地 IPv4 地址时所调用的函数，在后面继续创建路由项时同样调用这些函数，只是执行的逻辑分支会不一样，为了让脉络更加清晰，图 12.3.1 是添加 10.12.39.221 这一项路由项的函数调用流程。

首先在 `fib_inetaddr_event` 检测导师是 `NETDEV_UP` 事件发生，其会调用 `fib_add_ifaddr` 函数处理设备 UP 事件，`fib_add_ifaddr` 的源码如下：

```

net/ipv4/fib_frontend.c

734 void fib_add_ifaddr(struct in_ifaddr *ifa)
735 {
736     struct in_device *in_dev = ifa->ifa_dev;
737     struct net_device *dev = in_dev->dev;
738     struct in_ifaddr *prim = ifa;
739     __be32 mask = ifa->ifa_mask;
740     __be32 addr = ifa->ifa_local;
741     __be32 prefix = ifa->ifa_address & mask;
742
743     if (ifa->ifa_flags & IFA_F_SECONDARY) {
744         prim = inet_ifa_byprefix(in_dev, prefix, mask);
745         if (prim == NULL) {
746             pr_warn("%s: bug: prim == NULL\n", __func__);
747             return;
748         }
749     }
750
751     fib_magic(RTM_NEWRROUTE, RTN_LOCAL, addr, 32, prim);

```

```

752
753     if (!(dev->flags & IFF_UP))
754         return;
755
756     /* Add broadcast address, if it is explicitly assigned. */
757     if (ifa->ifa_broadcast && ifa->ifa_broadcast != htonl(0xFFFFFFFF))
758         fib_magic(RTM_NEWRROUTE, RTN_BROADCAST, ifa->ifa_broadcast, 32, prim);
759
760     if (!ipv4_is_zeronet(prefix) && !(ifa->ifa_flags & IFA_F_SECONDARY) &&
761         (prefix != addr || ifa->ifa_prefixlen < 32)) {
762         fib_magic(RTM_NEWRROUTE,
763             dev->flags & IFF_LOOPBACK ? RTN_LOCAL : RTN_UNICAST,
764             prefix, ifa->ifa_prefixlen, prim);
765
766         /* Add network specific broadcasts, when it takes a sense */
767         if (ifa->ifa_prefixlen < 31) {
768             fib_magic(RTM_NEWRROUTE, RTN_BROADCAST, prefix, 32, prim);
769             fib_magic(RTM_NEWRROUTE, RTN_BROADCAST, prefix | ~mask,
770                     32, prim);
771         }
772     }
773 }

```

743~749 如果 flag 参数指定了配置 IP 对象为从属设备或者临时设备，但是根据索引找不到该设备，返回错误。

751 行，在 ID 等于 255 的表中，插入 IP 地址类型为 2 的 IP 地址 dd270c0a。ID 等于 255 的表示是 LOCAL 表，即该表中的所有 IP 项的目的地址均是本机。类型见 IP 地址类型。

760~770 处理广播包的路由项。主机号全为 1 和主机号全为 0，实际的过程比较复杂，这里就不展开了，这里以 751 行添加的过程来说明路由表项是如何添加的。

IP 地址的类型如下

include/uapi/linux/rtnetlink.h

```

191 enum {
192     RTN_UNSPEC, //未定义
193     RTN_UNICAST, //单播，网关或者直接路由
194     RTN_LOCAL, //host 地址，目的地址是本机
195     RTN_BROADCAST, //广播地址，广播收广播发
197     RTN_ANYCAST, //广播接收数据，单播发送数据，IPV6 协议规定的地址类型
199     RTN_MULTICAST, //多播
200     RTN_BLACKHOLE, //丢弃
201     RTN_UNREACHABLE, //目的不可达
202     RTN_PROHIBIT, //管理员禁止 IP 地址
203     RTN_THROW, /* Not in this table */
204     RTN_NAT, //需要进行网络地址转换
205     RTN_XRESOLVE, //外部解析
206     __RTN_MAX
207 };

```

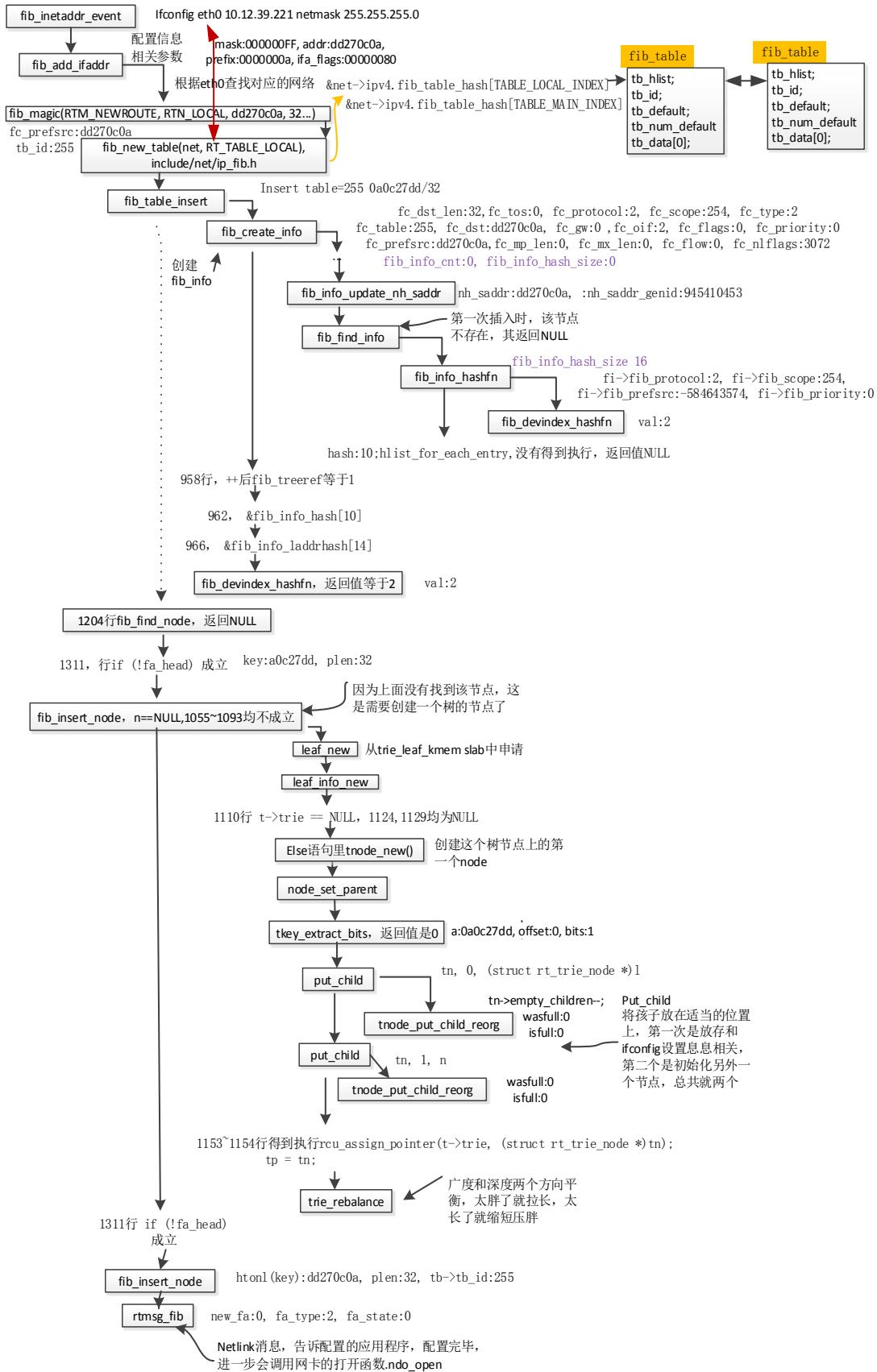


图 12.3.1 ifconfig 配置 IP 地址调用流程

路由项的管理集中在 `fib_frontend.c`、`fib_semantics.c` 和 `fib_trie.c` 这三个文件，后面的代

码也集中在这三个文件中。接着 `fib_magic` 函数的调用。这个函数的三个参数：

Cmd: ioctl 传递的命令参数，这里是 `RTM_NEWRROUTE`，对应添加路由项，此外前面提到的 netfilter 也是在这个 `magic` 函数里处理的。

Type: 这个参数是路由项的类型，前面已经介绍过了，这里传递的参数是 `RTN_LOCAL`，配置的 IP 地址的目的端就是本机。

Dst: 是一个 32bit 的 IP 地址，是要添加到路由表项，`ifa` 是一些附属配置信息。

net/ipv4/fib_frontend.c

```
696 static void fib_magic(int cmd, int type, __be32 dst, int dst_len, struct in_ifaddr *ifa)
697 {
698     struct net *net = dev_net(ifa->ifa_dev->dev);
699     struct fib_table *tb;
700     struct fib_config cfg = {
701         .fc_protocol = RTPROT_KERNEL,
702         .fc_type = type,
703         .fc_dst = dst,
704         .fc_dst_len = dst_len,
705         .fc_prefsrc = ifa->ifa_local,
706         .fc_oif = ifa->ifa_dev->dev->ifindex,
707         .fc_nlflags = NLM_F_CREATE | NLM_F_APPEND,
708         .fc_nlinfo = {
709             .nl_net = net,
710         },
711     };
712
713     if (type == RTN_UNICAST)
714         tb = fib_new_table(net, RT_TABLE_MAIN);
715     else
716         tb = fib_new_table(net, RT_TABLE_LOCAL);
717
718     if (tb == NULL)
719         return;
720
721     cfg.fc_table = tb->tb_id;
722
723     if (type != RTN_LOCAL)
724         cfg.fc_scope = RT_SCOPE_LINK;
725     else
726         cfg.fc_scope = RT_SCOPE_HOST;
727
728     if (cmd == RTM_NEWRROUTE)
729         fib_table_insert(tb, &cfg);
730     else
731         fib_table_delete(tb, &cfg);
732 }
```

700~710 行，将用户传递的配置参数使用内核下的数据结构保护起来。

713~716 通过类型，索引要添加到的表。

include/net/ip_fib.h

```
204 static inline struct fib_table *fib_get_table(struct net *net, u32 id)
205 {
206     struct hlist_head *ptr;
207
208     ptr = id == RT_TABLE_LOCAL ?
209         &net->ipv4.fib_table_hash[TABLE_LOCAL_INDEX] :
210         &net->ipv4.fib_table_hash[TABLE_MAIN_INDEX];
211     return hlist_entry(ptr->first, struct fib_table, tb_hlist);
```

```

212 }
213
214 static inline struct fib_table *fib_new_table(struct net *net, u32 id)
215 {
216     return fib_get_table(net, id);
217 }

```

这个函数的第一个参数是一个 `net`, 代表了一个网络, 其附属于一个网络命名空间, 第二个参数是 716 行的 `RT_TABLE_LOCAL`。索引返回的类型是 `fib_table`, 在路由表核心数据结构时, 介绍过 `fib_table` 代表的是路由表的大类, 这些路由表可以多打 256 张, 这么多张的表使用哈希法索引。

`fib_get_table` 的 209 行就是返回 `ipv4` 的路由项, `&net->ipv4.fib_table_hash[TABLE_LOCAL_INDEX]`。

718~719 判断是否得到了该表项, 依据就是这个表的地址非 `NULL`, 实际上在初始化时这个表已经得到了存储表项的空间。

721~726 将参数保存在 `cfg` 结构体内。

728~729 将 `cfg` 中保存的配置路由项的信息, 插入到由 `fib_get_table` 获得的表中。

路由项的插入工作就是在 `fib_table_insert` 这个函数中完成的, 这个函数大体分为四个部分:

1、路由信息获取, 每一个路由项都会对应一个路由信息, 并且一个路由信息可以对应多个路由项。获取含义是先查找, 如果找不到路由信息则根据传递的参数创建路由信息。

2、`Trie` 树节点 `tnode`。存在多个路由项, 并且他们有共同的前缀时, 共同的前缀会被设定为 `tnode`。

3、`Trie` 树叶子 `leaf`, 如果一个路由项不在路由表中, 但是其和其它已经在路由表中的项具有相同的前缀, 则会创建对应的叶子, 在 `rebalance` 操作时, `tnode` 和 `leaf` 因深度和广度原因都可能被调整。

4、插入操作, 就是将新创建的叶子插入到 `trie` 树上去, 也就对应创建了一个路由项。

`net/ipv4/fib_frontend.c`

```

1172 int fib_table_insert(struct fib_table *tb, struct fib_config *cfg)
1173 {
1174     struct trie *t = (struct trie *) tb->tb_data;
1175     struct fib_alias *fa, *new_fa;
1176     struct list_head *fa_head = NULL;
1177     struct fib_info *fi;
1178     int plen = cfg->fc_dst_len;
1179     u8 tos = cfg->fc_tos;
1180     u32 key, mask;
1181     int err;
1182     struct leaf *l;
//图 12.1.2 中 plen 值等于 32。下面的判断条件满足。
1184     if (plen > 32)
1185         return -EINVAL;
//图 12.3.1 中, fc_dst 是 0a0c27dd, 即 10.12.39.221, 用户空间配置的路由地址
1187     key = ntohl(cfg->fc_dst);
//mask 等于~0
1191     mask = htonl(inet_make_mask(plen));
//验证 key (目的地址) 和目的地址掩码的正确性。
1193     if (key & ~mask)
1194         return -EINVAL;
//经过掩码匹配, 得到真正的键值 key 等于 10.12.39.221
1196     key = key & mask;
//对应于四个部分中的第一个部分, 路由信息获取, 见后文

```

```

1198 fi = fib_create_info(cfg);
1199 if (IS_ERR(fi)) {
1200     err = PTR_ERR(fi);
1201     goto err;
1202 }
//对应于四个部分的第二、三两个部分，遍历 tnode 获取，获取叶子 leaf，见后文。
1204 l = fib_find_node(t, key);
1205 fa = NULL;
//在使用 ifconfig 向 LOCAL 表插入 10.12.39.221 前，路由表时空的，所以这里不可能找到对应的 fib_table 和 leaf 信息。所以
//这里 fi 返回值是新创建的路由信息记录项，l==NULL，fa==NULL，直接跳至 1293 行，对于如果有相同前缀的路由 IP 项，
//fib_find_node 返回的是一个 tnode。
1207 if (l) {
//get_fa_head 根据 leaf 信息，获得 fib_alias 链表的头指针。
1208     fa_head = get_fa_head(l, plen);
//遍历上述链表，找到合适的服务类型和优先级，合适的意义是优先级要高于这里传递的参数
1209     fa = fib_find_alias(fa_head, tos, fi->fib_priority);
1210 }
//fa 在首次配置 IP 地址时为空，第一遍先跳过下面对这个 if 语句的注释吧，第二遍再来看。
1223 if (fa && fa->fa_tos == tos &&
1224     fa->fa_info->fib_priority == fi->fib_priority) {
1225     struct fib_alias *fa_first, *fa_match;
1226
1227     err = -EEXIST;
1228     if (cfg->fc_nliflags & NLM_F_EXCL)
1229         goto out;
1230
1231 /* We have 2 goals:
1232     * 1. Find exact match for type, scope, fib_info to avoid
1233     * duplicate routes
1234     * 2. Find next 'fa' (or head), NLM_F_APPEND inserts before it
1235 */
1236     fa_match = NULL;
1237     fa_first = fa;
1238     fa = list_entry(fa->fa_list.prev, struct fib_alias, fa_list);
1239     list_for_each_entry_continue(fa, fa_head, fa_list) {
1240         if (fa->fa_tos != tos)
1241             break;
1242         if(fa->fa_info->fib_priority != fi->fib_priority)
1243             break;
1244         if (fa->fa_type == cfg->fc_type &&
1245             fa->fa_info == fi) {
1246             fa_match = fa;
1247             break;
1248         }
1249     }
1250
1251     if (cfg->fc_nliflags & NLM_F_REPLACE) {
1252         struct fib_info *fi_drop;
1253         u8 state;
1254
1255         fa = fa_first;
1256         if (fa_match) {
1257             if (fa == fa_match)
1258                 err = 0;
1259             goto out;
1260         }
1261         err = -ENOBUFS;
1262     }
//上面的代码已然没有找到 fib alias，所以这里创建新的 fib alias。并在 1266~1273 行对其成员进行初始化。

```

```

1262     new_fa =kmem_cache_alloc(fn_alias_kmem, GFP_KERNEL);
1263     if (new_fa == NULL)
1264         goto out;
1265
1266     fi_drop = fa->fa_info;
1267     new_fa->fa_tos = fa->fa_tos;
1268     new_fa->fa_info = fi;
1269     new_fa->fa_type =cfg->fc_type;
1270     state = fa->fa_state;
1271     new_fa->fa_state = state &~FA_S_ACSESSED;
1272
1273     list_replace_rcu(&fa->fa_list,&new_fa->fa_list);
1274     alias_free_mem_rcu(fa);
1275
1276     fib_release_info(fi_drop);
1277     if (state & FA_S_ACSESSED)
1278         rt_cache_flush(cfg->fc_nlinfo.nl_net);
1279     rtmsg_fib(RTM_NEWRROUTE,htonl(key), new_fa, plen,
1280             tb->tb_id,&cfg->fc_nlinfo, NLM_F_REPLACE);
1281
1282     goto succeeded;
1283 }
1284 /* Error if we find a perfect match which
1285 * uses the same scope, type, and nexthop
1286 * information.
1287 */
1288 if (fa_match)
1289     goto out;
1290
1291 if (!(cfg->fc_nlflags & NLM_F_APPEND))
1292     fa = fa_first;
1293 }
1294 err = -ENOENT;
//由图 12.3.1 可以知道 fc_nlflags 的值等于 1024，就是 0x400，正好等于这里的（NLM_F_CREATE），所以还要接着 1297 行看。
1295 if (!(cfg->fc_nlflags & NLM_F_CREATE))
1296     goto out;
1297
1298 err = -ENOBUFS;
//从 fn_alias_kmem 上分配一块 cache，这个 cache 用于 flash 别名系统
1299 new_fa = kmem_cache_alloc(fn_alias_kmem, GFP_KERNEL);
1300 if (new_fa == NULL)
1301     goto out;
1302
//向 fib_alias 中插入相关的路由信息，并将其成员 fa_info 指向前面的 fib_info 结构体。
1303 new_fa->fa_info = fi;
1304 new_fa->fa_tos = tos;
1305 new_fa->fa_type = cfg->fc_type;
1306 new_fa->fa_state = 0;
1307 /*
1308 * Insert new entry to the list.
1309 */
//fa_head 在 1208 行没有得到复制，这里 fa_head == NULL，fib_insert_node 对应于第四个部分，插入操作，见后文
1310 if (!fa_head) {
1311     fa_head = fib_insert_node(t, key, plen);
1312     if (unlikely(!fa_head)) {
1313         err = -ENOMEM;
1314         goto out_free_new_fa;
1315     }
1316 }
```

```

1317 }
1318 //跟新 tb_num_default 字段
1319 if (!plen)
1320     tb->tb_num_default++;
//处理 fa_list 链表
1322 list_add_tail_rcu(&new_fa->fa_list,
1323     (fa ? &fa->fa_list :fa_head));
1324
1325 rt_cache_flush(cfg->fc_nlinfo.nl_net);
发送 netlink 消息
1326 rtmmsg_fib(RTM_NEWRROUTE, htonl(key), new_fa, plen, tb->tb_id,
1327     &cfg->fc_nlinfo, 0);
1328 succeeded:
1329 return 0;
//差错处理
1331 out_free_new_fa:
1332 kmem_cache_free(fn_alias_kmem, new_fa);
1333 out:
1334 fib_release_info(fi);
1335 err:
1336 return err;
1337 }

```

1198 行 `fib_create_info` 获得一个路由信息记录结构体，当要获得的对象不存在时，会创建一个新的路由信息结构体 `fib_info`。

```

net/ipv4/fib_semantics.c
773 struct fib_info *fib_create_info(struct fib_config *cfg)
774 {
775     int err;
776     struct fib_info *fi = NULL;
777     struct fib_info *ofi;
778     int nhs = 1;
779     struct net *net = cfg->fc_nlinfo.nl_net;
//用户空间传递进来的 tc_type 值是 2，所以这里的检查类型的有效性通过。
781     if (cfg->fc_type > RTN_MAX)
782         goto err_inval;
//本地地址 LOCAL 的值是 RT_SCOPE_HOST, 254，用户传递进来的 fc_scope 等于 254（图 13.3.1），这里检查也通过。
784     /* Fast check to catch the most weird cases */
785     if (fib_props[cfg->fc_type].scope > cfg->fc_scope)
786         goto err_inval;
796     err = -ENOBUFS;
//对于由于前表项为空，所以统计路由信息技术变量 fib_info_cnt 的值等于 0，索引路由信息的哈希数组大小变量
//fib_info_hash_size 值等于 0。
797     if (fib_info_cnt >= fib_info_hash_size) {
798         unsigned int new_size = fib_info_hash_size << 1;
799         struct hlist_head *new_info_hash;
800         struct hlist_head *new_laddrhash;
801         unsigned int bytes;
//由于 fib_info_hash_size 确实为 0，所以这里的 new_size 将被赋值成 16。
803         if (!new_size)
804             new_size = 16;
805         bytes = new_size * sizeof(struct hlist_head *);
//为哈希信息和哈希地址存储分配内存，如果小于一个页使用 kzalloc，大于一个页使用 __get_free_pages。
806         new_info_hash = fib_info_hash_alloc(bytes);
807         new_laddrhash = fib_info_hash_alloc(bytes);
//分配失败的处理，如果没有失败，调用 fib_info_hash_move 进行处理。
808         if (!new_info_hash || !new_laddrhash) {

```

```

809     fib_info_hash_free(new_info_hash, bytes);
810     fib_info_hash_free(new_laddrhash, bytes);
811 } else
//根据先前申请到的内存，设置 fib_info_hash 和 fib_info_laddrhash 这两个 fib_semantics.c 文件内的全局变量。这两变量用于
链接所有的 fib_info 结构体。
812     fib_info_hash_move(new_info_hash, new_laddrhash, new_size);
813
814     if (!fib_info_hash_size)
815         goto failure;
816 }
//申请路由信息空间，注意零长数组指向了下一跳 fib_nh(nh—next hop)。
818 fi = kzalloc(sizeof(*fi)+nhs*sizeof(struct fib_nh), GFP_KERNEL);
819 if (fi == NULL)
820     goto failure;
/*用户空间没有配置 metric， metirc 用于配置下一跳的个数，不指定的情况下赋值如下。
*const u32 dst_default_metrics[RTAX_MAX + 1] = {
*    [RTAX_MAX] = 0xdeadbeef,
*};
*/
821 if (cfg->fc_mx) {
822     fi->fib_metrics = kzalloc(sizeof(u32) * RTAX_MAX, GFP_KERNEL);
823     if (!fi->fib_metrics)
824         goto failure;
825 } else
826     fi->fib_metrics = (u32 *) dst_default_metrics;
//fib_info_cnt 用于计数有意义的 fib_info 个数。
827 fib_info_cnt++;
//下面的赋值，见图 12.3.2，fib_info 结构体赋值，值来源于用户配置。
829 fi->fib_net = hold_net(net);
830 fi->fib_protocol = cfg->fc_protocol;
831 fi->fib_scope = cfg->fc_scope;
832 fi->fib_flags = cfg->fc_flags;
833 fi->fib_priority = cfg->fc_priority;
834 fi->fib_prefsrc = cfg->fc_prefsrc;
835 fi->fib_type = cfg->fc_type;
//初始化下一跳的成员，对于没有配置等价路由的，这个循环只会执行一次，对于等价路由，有几个等价的路由项就会执行几
//次。将下一跳的 nh_parent 指向 fib_info 字段。分配一个 percpu 变量，这会为每个核创建一个对应的 rtable，rtable 是路由缓
//存部分的，其将加速路由项的查找，路由缓存参考 11.5 节。
837 fi->fib_nhs = nhs;
838 change_nexthops(fi) {
839     nexthop_nh->nh_parent = fi;
840     nexthop_nh->nh_pcpu_rth_output = alloc_percpu(struct rtable __rcu *);
841     if (!nexthop_nh->nh_pcpu_rth_output)
842         goto failure;
843 } endfor_nexthops(fi)
//用户空间没有传递 metric，跳过。
845 if (cfg->fc_mx) {
846     struct nlattr *nla;
847     int remaining;
848
849     nla_for_each_attr(nla, cfg->fc_mx, cfg->fc_mx_len, remaining) {
850         int type = nla_type(nla);
851
852         if (type) {
853             u32 val;
854
855             if (type > RTAX_MAX)
856                 goto err_inval;

```

```

857         val = nla_get_u32(nla);
858         if (type == RTAX_AdVMSS&& val > 65535 - 40)
859             val = 65535 - 40;
860         if (type == RTAX_MTU&& val > 65535 - 15)
861             val = 65535 - 15;
862         fi->fib_metrics[type - 1] =val;
863     }
864 }
865 }

//用户空间也没有配置等价路由，执行 else 语句。
867 if (cfg->fc_mp) {
868 #ifdef CONFIG_IP_ROUTE_MULTIPATH
869     err = fib_get_nhs(fi, cfg->fc_mp, cfg->fc_mp_len, cfg);
870     if (err != 0)
871         goto failure;
872     if (cfg->fc_oif && fi->fib_nh->nh_oif !=cfg->fc_oif)
873         goto err_inval;
874     if (cfg->fc_gw && fi->fib_nh->nh_gw != cfg->fc_gw)
875         goto err_inval;
876 #endif CONFIG_IP_ROUTE_CLASSID
877     if (cfg->fc_flow && fi->fib_nh->nh_tclassid !=cfg->fc_flow)
878         goto err_inval;
879#endif
880#else
881     goto err_inval;
882#endif
883 } else {
//对下一跳的赋值，见图 12.3.2。赋值的来源同样是用户空间。
884     struct fib_nh *nh = fi->fib_nh;
885
886     nh->nh_oif = cfg->fc_oif;
887     nh->nh_gw = cfg->fc_gw;
888     nh->nh_flags = cfg->fc_flags;
889 #ifdef CONFIG_IP_ROUTE_CLASSID
890     nh->nh_tclassid = cfg->fc_flow;
891     if (nh->nh_tclassid)
892         fi->fib_net->ipv4.fib_num_tclassid_users++;
893#endif
894 #ifdef CONFIG_IP_ROUTE_MULTIPATH
895     nh->nh_weight = 1;
896#endif
897 }

//fc_type 类型是 2，即单播，该类型的 error 成员值是 0，执行 else 语句。else 语句判断 type 字段是否合法。
898 if (fib_props[cfg->fc_type].error) {
899     if (cfg->fc_gw || cfg->fc_oif || cfg->fc_mp)
900         goto err_inval;
901     goto link_it;
902 }
903 } else {
904     switch (cfg->fc_type) {
905     case RTN_UNICAST:
906     case RTN_LOCAL:
907     case RTN_BROADCAST:
908     case RTN_ANYCAST:
909     case RTN_MULTICAST:
910         break;
911     default:
912         goto err_inval;
913 }

```

```

914 }
//范围大于 RT_SCOPE_HOST (253) 就出错了。用于限定路由的范围。
916 if (cfg->fc_scope > RT_SCOPE_HOST)
917     goto err_inval;
//fc_scope 的值等于 254，执行 if 语句里的内容，当使用 route 命令配置一个 (RT_SCOPE_LINK) 范围地址，执行 else 语句
919 if (cfg->fc_scope == RT_SCOPE_HOST) {
//fi 是在 818 行创建的指向 fib_info 的结构体，该结构体的部分程序员在 821~843 被赋值，这里是处理其 nh (next hop) 字段。
920     struct fib_nh *nh = fi->fib_nh;
921
922     /* Local address is added. */
923     if (nhs != 1 || nh->nh_gw)
924         goto err_inval;
//目的地址是本机，所以 scope 赋值成 RT_SCOPE_NOWHERE,该 scope 范围不会向外发送数据包。
925     nh->nh_scope = RT_SCOPE_NOWHERE;
//根据索引号获得 net_device 结构体，net_device 结构体用于标记网卡，其实就是“eth0”结构体，该网卡用来发送数据。
926     nh->nh_dev = dev_get_by_index(net, fi->fib_nh->nh_oif);
927     err = -ENODEV;
928     if (nh->nh_dev == NULL)
929         goto failure;
930 } else {
931     change_nexthops(fi);
//fib_check_nh 检查下一跳语义的正确性,
932     err = fib_check_nh(cfg, fi, nexthop_nh);
933     if (err != 0)
934         goto failure;
935 } endfor_nexthops(fi)
936 }

// prefsrc 是 dd270c0a，并且检查也通过。
938 if (fi->fib_prefsrc) {
939     if (cfg->fc_type != RTN_LOCAL || !cfg->fc_dst ||
940         fi->fib_prefsrc != cfg->fc_dst)
941         if (inet_addr_type(net, fi->fib_prefsrc) != RTN_LOCAL)
942             goto err_inval;
943 }

//获得对应接口的源地址信息，将这些信息存放在下一跳 nexthop_nh 中。
945 change_nexthops(fi) {
946     fib_info_update_nh_saddr(net, nexthop_nh);
947 } endfor_nexthops(fi)

//在 fib_info_hash 所标示的哈希表中，见图 12.3.2 左上角，紫色部分；看看要插入的路由项是否已经存在了，如果存在，则//955
行直接返回，由于用户空间的配置路由项原本内核路由表中没有，所以接着 958 行继续。
949 link_it:
950     ofi = fib_find_info(fi);
951     if (ofi) {
952         fi->fib_dead = 1;
953         free_fib_info(fi);
954         ofi->fib_treeref++;
955         return ofi;
956     }

//路由树引用计数++
958     fi->fib_treeref++;
959     atomic_inc(&fi->fib_clntref);

//获得保护该路由信息记录项的锁。这个 fib_info_hash 表是局部全局的。以安全将上面创建的 fi 添加到 fib_info_hash 链表上。
960     spin_lock_bh(&fib_info_lock);

//将这个新的 fib_info，添加到管理这个结构的全局哈希表上。
961     hlist_add_head(&fi->fib_hash,
962                     &fib_info_hash[fib_info_hashfn(fi)]);
969     change_nexthops(fi) {
970         struct hlist_head *head;

```

```
971     unsigned int hash;
972
973     if (!nexthop_nh->nh_dev)
974         continue;
//ifindex 的值等于 2, 对应于 eth0。
975     hash = fib_devindex_hashfn(nexthop_nh->nh_dev->ifindex);
976     head = &fib_info_devhash[hash];
// fib_info_devhash 是存储设备的哈希表, 将 nexthop 指向设备的哈希元素添加到 fib_info_devhash 链表上。
977     hlist_add_head(&nexthop_nh->nh_hash, head);
978 } endfor_nexthops(fi)
979 spin_unlock_bh(&fib_info_lock);
//最后这里返回已经添加到相应管理链表上的 fib_info 的结构体信息 fi。
980 return fi;
992 }
```

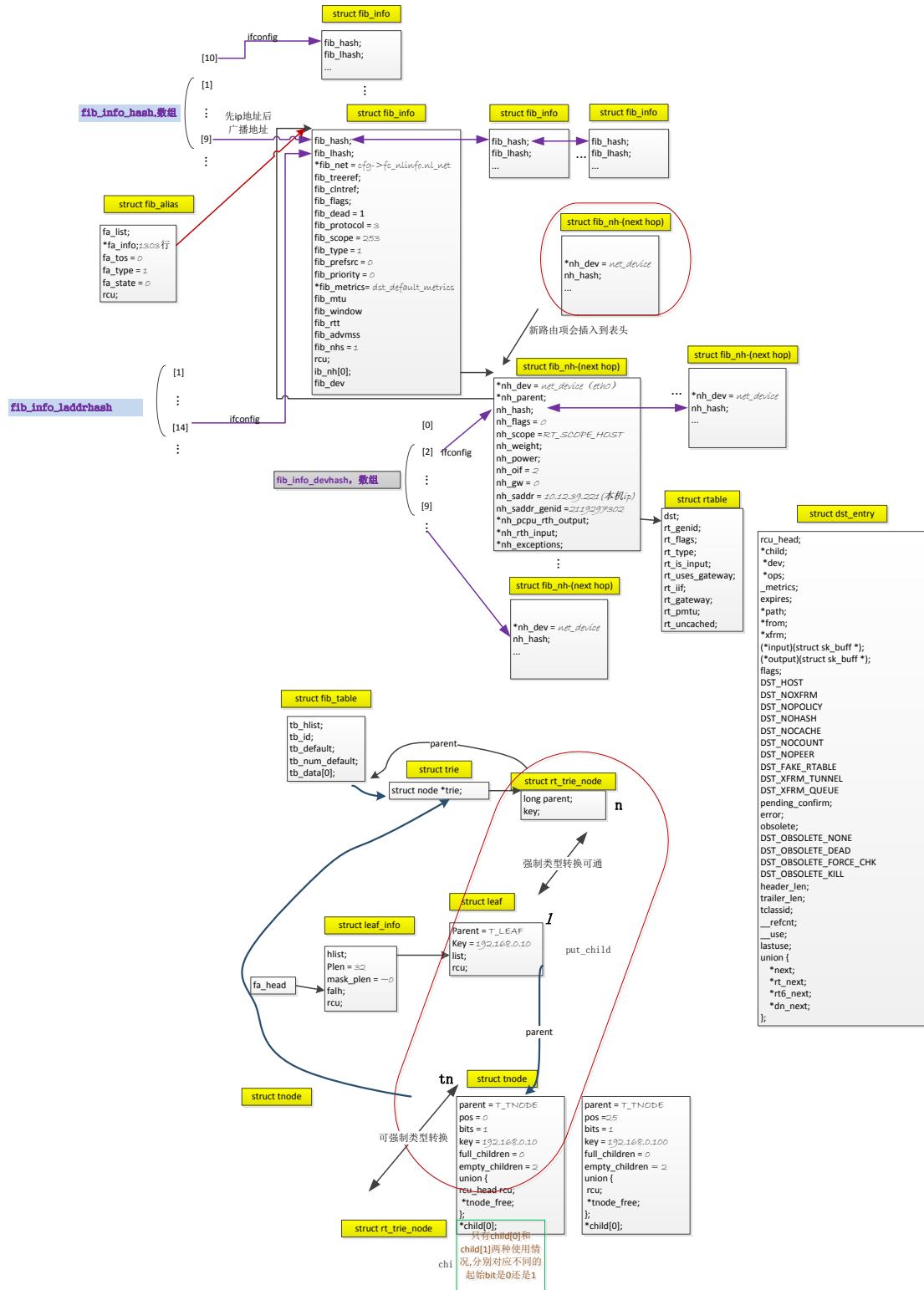


图 12.3.2 ifconfig 使用到的数据结构

`fib_find_info` 参数值是前面创建的 `fib_info` 结构体，这个函数就是在管理 `fib_info` 的哈希链表 `fib_info_hash` 查找先前创建的 `fib_info` 信息是否已经存在这张链表了，如果存在，那么说明 `fib_info` 没有必要在创建了，直接重复利用就好了。

```
298 static struct fib_info *fib_find_info(const struct fib_info *nfi)
```

299 {

```
300     struct hlist_head *head;
```

```

301     struct fib_info *fi;
302     unsigned int hash;
//计算 fib_info 的哈希索引值，该哈希值依赖于 fib_protocol、fib_scope、fib_prefsrc、fib_priority 以及设备 index。
304     hash = fib_info_hashfn(nfi);
//直接得到传递进来了 fib_info 所对应的哈希表头，如果在这个表中能够找到就直接返回，如果找不到返回 NULL，返回去以后
//的函数查看返回值，是 NULL 的话，说明这项并不存在，就会将这个 fib_info 添加到 fib_info_hash 链表上。
305     head = &fib_info_hash[hash];
//遍历哈希表，依次对比哈希表的每一个成员的下列字段和传递进来要找的 fib_info 是否相同。
307     hlist_for_each_entry(fi, head, fib_hash) {
308         if (!net_eq(fi->fib_net, nfi->fib_net))
309             continue;
310         if (fi->fib_nhs != nfi->fib_nhs)
311             continue;
312         if (nfi->fib_protocol == fi->fib_protocol &&
313             nfi->fib_scope == fi->fib_scope &&
314             nfi->fib_prefsrc == fi->fib_prefsrc &&
315             nfi->fib_priority == fi->fib_priority &&
316             nfi->fib_type == fi->fib_type &&
317             memcmp(nfi->fib_metrics, fi->fib_metrics,
318                     sizeof(u32) * RTAX_MAX) == 0 &&
319             ((nfi->fib_flags ^ fi->fib_flags) & ~RTNH_F_DEAD) == 0 &&
320             (nfi->fib_nhs == 0 || nh_comp(fi, nfi) == 0))
321             return fi;
322     }
323
324     return NULL;
325 }

```

继续回到 fib_table_insert 的 1204 行 fib_find_node，该函数用于查找叶子节点。页叶子节点和 tnode 都是用来表示字典树节点，它们处在这棵树的位置不一样，两者并不完全相同，但是前两个成员是一样的，两者之间的互相转换的技巧前面已经遇到过了。这个函数的参数，key 就是要添加的 192.168.0.10 IP 地址，第一个参数见图 12.3.2 路由表拓扑的底部。这个函数查找的过程是这样的：首先遍历 tnode，然后是遍历 leaf，这个过程还是很容易理解的。

```

955 static struct leaf *
956 fib_find_node(struct trie *t, u32 key)
957 {
958     int pos;
959     struct tnode *tn;
960     struct rt_trie_node *n;
961
962     pos = 0;
//将传递进来的 trie 节点转换成 tnode 类型。
963     n = rcu_dereference_rtnl(t->trie);
//************************************************************
//在插入唯一一项 10.12.39.0 时，这一项就是一个 leaf。
//在添加 192.168.0.10 时，由于 n 指向 10.12.39.0 的 rt_trie_node 类结构体，并不是 NULL。
// n->parent:00000001, n->key:0a0c2700。
//而在添加 192.168.0.10 时，
// n->parent:00000000, n->key:0a0c2700
n->parent 用于存放其父节点的地址，这个父节点是 tnode，如果不存在则是 NULL (0000000*)。该成员的最低 bit 用于标识
这个节点是 tnode 还是 leaf。如果是 tnode 则最低一个 bit 是 0，如果是 leaf 则最低一个 bit 是 1。
#define T_TNODE 0
#define T_LEAF 1
在插入 192.168.0.10 时，由于 10.12.39.0 是个 leaf，所以这个循环被跳过了。而插入 192.168.0.100 时，由于有 tnode 存在，
所以这里 while 循环就不会被跳过了，这个 tnode 是图 12.3.2 的上面两行产生的，至于 pos 为 25 的则是 192.168.0.100 插入之
后才会存在的 tnode。
*****

```

```

965     while (n != NULL && NODE_TYPE(n) == T_TNODE) {
966         tn = (struct tnode *) n;
967
968         check_tnode(tn);
//970~977 非常经典的几行代码，就这么短短的几行代码能够处理更正情况下与要查找的 key 最接近的 tnode。
970         if (tkey_sub_equals(tn->key, pos, tn->pos-pos, key)) {
971             pos = tn->pos + tn->bits;
972             n = tnode_get_child_rcu(tn,
973                 tkey_extract_bits(key,
974                     tn->pos,
975                     tn->bits));
976         } else
977             break;
978     }

//如果这个 if 语句满足，说明我们要插入的项已经存在这个路由表了，这是一次重复插入操作。
981     if (n != NULL && IS_LEAF(n) && tkey_equals(key, n->key))
982         return (struct leaf *)n;
983
984     return NULL;
985 }

```

为了说明遍历 tnode 的过程，看图 12.3.2，如果根据该图张图不能想象出 trie 的拓扑结构，参考图 12.2.4，它们是一样的拓扑结构。在这个基础上，这里在次插入 192.168.0.11 这个项。

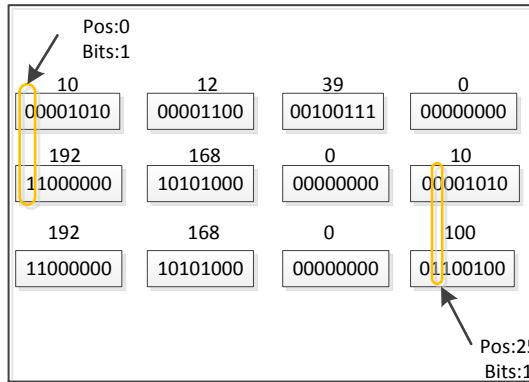


图 12.3.2 路由项查找实例

Key 值就是 192.168.0.11，t 参数指向 10.12.39.0 这个 tnode。

遍历过程如下：

1、遍历第一个 tnode，if 语句因 tn->pos-pos 的等于零成立。遍历 trie 树第一个 tnode。

tkey_extract_bits 提取 key (192.168.0.11) 的 pos 位置开始的 bits 位的值，这里就是 1。tnode_get_child_rcu 转变成 tnode_get_child_rcu(tn,1)，这个函数获得 tnode 的 child 成员，这里就是 child[1]，

2、再一次遍历 tnode 的第二个节点。这个 tn->pos 是 25，tn->bits 是 1。971 行的 pos 将变成 26，这就意味着从第 26 个 bit 开始比较。然后比较的 bits 数是 1。这时得到是 child[0]，参考图 12.2.4，该节点挂载的是 192.168.0.10。

3、该函数返回以后会，外部函数会将这个 child 指向的 leaf 节点，变成指向一个 tnode 节点，tnode 的孩子节点分别指向 192.168.0.10 和 192.168.0.11。

1312 行是核心的插入操作，为了使这一过程更具代表性，这里结合图 12.2.4 以插入 192.168.0.100 为例进行源码分析，在插入 192.168.0.100 之前的状态见图 12.2.3，只存在 key 等于 10.12.39.0 的 tnode，和 child[0] 指向 key 等于 10.12.39.0 的 leaf 和 child[1] 指向 key 值等

于 192.168.0.10 的 leaf 节点。在插入 192.168.0.100 时，fib_insert_node 的参数意义如下：

- 1、t 指向 key 等于 10.12.39.0 的 tnode；
- 2、key 是需要插入的目的 IP 192.168.0.100
- 3、plen 是前缀长度，这里是 32。

```
1023 static struct list_head *fib_insert_node(struct trie *t, u32 key, int plen)
1024 {
1025     int pos, newpos;
1026     struct tnode *tp = NULL, *tn = NULL;
1027     struct rt_trie_node *n;
1028     struct leaf *l;
1029     int missbit;
1030     struct list_head *fa_head = NULL;
1031     struct leaf_info *li;
1032     t_key cindex;
1033
1034     pos = 0;
//rcu 类型变量引用。
1035     n = rtnl_dereference(t->trie);
//由于 t->trie 是 key 等于 10.12.39.0 的 tnode，所以这个 while 的语句判断第一次是满足的。
1055     while (n != NULL && NODE_TYPE(n) == T_TNODE) {
//从类型 rt_trie_node 到 tnode 强制类型转换，内核常用技巧
1056         tn = (struct tnode *) n;
//tnode 验证，tnode != NULL，并且该 tnode 的 pos 和 bits 成员之和需要小于 32。
1058         check_tnode(tn);
//下面这段代码和 fib_find_node 一样。
1060         if (tkey_sub_equals(tn->key, pos, tn->pos-pos, key)) {
1061             tp = tn; //图 12.2.4 中的 copy 来自这行代码。
1062             pos = tn->pos + tn->bits; //获得比较的起始位置
//返回 tnode 下合适的孩子，这个孩子可能是 tnode 也可能是 leaf，对于这里情况返回 192.168.0.10 leaf。
1063             n = tnode_get_child(tn,
1064                     tkey_extract_bits(key, //根据传递进来的 key，查 tnode 处不同的 bits。
1065                         tn->pos,
1066                         tn->bits));
1067
1068             BUG_ON(n && node_parent(n) != tn);
1069         } else
1070             break;
1071     }
1072
//这里两个条件满足，但是第三个条件不满足，实际上如果第三个条件满足，这就说明该路由项已经在该路由表中了，没有必要再次添加该路由项。
1083     if (n != NULL && IS_LEAF(n) && tkey_equals(key, n->key)) {
1084         l = (struct leaf *) n;
1085         li = leaf_info_new(plen);
1086
1087         if (!li)
1088             return NULL;
//将 leaf 串接成一个链表，leaf_info 结构体的 falh 成员完成这一工作。
1090         fa_head = &li->falh;
1091         insert_leaf_info(&l->list, li);
1092         goto done;
1093     }
//如果 1083 行 if 语句不满足，说明路由表中不存在这么一项，创建一个新的 leaf，该 leaf 用于插入项的索引。
1094     l = leaf_new();
//申请内存失败处理
1096     if (!l)
```

```

1097         return NULL;
//这里将 192.168.0.100 作为 key 值传递给 leaf，并根据前缀长度创建一个记录 leaf 信息的 li (leaf_info 结构体)
1099     l->key = key;
1100     li = leaf_info_new(plen);
//记录 leaf 信息的结构体内存申请失败的处理。
1102     if (!li) {
1103         free_leaf(l);
1104         return NULL;
1105     }
//将 leaf_info 和 leaf 建立关系，即将 leaf_info 的 falh 成员指向 leaf 的 list。
1107     fa_head = &li->falh;
1108     insert_leaf_info(&l->list, li);
//上面已经看到 n 显然不等于 NULL。
1110     if (t->trie && n == NULL) {
1111         /* Case 2: n is NULL, and will just insert a new leaf */
1112
1113         node_set_parent((struct rt_trie_node *)l, tp);
1114
1115         cindex = tkey_extract_bits(key, tp->pos, tp->bits);
1116         put_child(tp, cindex, (struct rt_trie_node *)l);
1117     } else {
//tp 在前面得到赋值(copy)，tp->pos+tp->bits 将等于 1，这就意味着 key 等于 192.168.0.100 的插入项从第一个比特开始比较。
1124         if (tp)
1125             pos = tp->pos+tp->bits;
1126         else
1127             pos = 0;
1128
1129         if (n) {
/*****
265 static inline int tkey_mismatch(t_key a, int offset, t_key b)
266 {
267     t_key diff = a ^ b;
268     int i = offset;
269
270     if (!diff)
271         return 0;
272     while ((diff << i) >> (KEYLENGTH-1) == 0)
273         i++;
274     return i;
275 }
//key:c0a80064,pos:1,n->key:0a0c2700
将这三个参数带入上面的函数，KEYLENGTH 值是 32，可得 25，源于 192.168.0.100 和 192.168.0.10 二进制不同的起始比特
*****/
1130         newpos = tkey_mismatch(key, pos, n->key);
//根据 n->key 和 newpos 创建一个 tnode，这对应于图 12.2.4 中的 key 等于 192.168.0.10 的 tnode。
1131         tn = tnode_new(n->key, newpos, 1);
1132     } else {
//首次需要创建 tnode 的情况，这是一个特殊情况，一个路由表，只会调用一次这里的函数。
1133         newpos = 0;
1134         tn = tnode_new(key, newpos, 1); /* First tnode */
1135     }
//tn 创建失败的处理
1137     if (!tn) {
1138         free_leaf_info(li);
1139         free_leaf(l);
1140         return NULL;
1141     }
//这个函数就是设置这里创建的 tn (tree node) 的父节点。注意看图 12.2.4 中的 ADD 开始字符串，这里的赋值使用就是 ADD

```

```

//冒号后字符串,实际上是地址, 注意图中它们的赋值关系。
1143     node_set_parent((struct rt_trie_node *)tn, tp);
//从第 25 个比特比起, 它们不同的第一个比特是 1,
1145     missbit = tkey_extract_bits(key, newpos, 1);
//在 tnode 的 child[1], 插入 key 值等于 192.168.0.100 的 leaf。见 12.3.4 小节。
//在 tnode 的 child[0], 插入 key 值等于 192.168.0.10 的 leaf.
1146     put_child(tn, missbit, (struct rt_trie_node *)l);
1147     put_child(tn, 1-missbit, n);
//1061 行, 如果有 copy, 那么需要跟新原来 tnode 的 child, 其 child[1]变成了一个具有两个 leaf 的 tnode。
1149     if (tp) {
//这里找到 tnode 插入的起始位置。
1150         cindex = tkey_extract_bits(key, tp->pos, tp->bits);
//put_child 为插入操作。
1151         put_child(tp, cindex, (struct rt_trie_node *)tn);
1152     } else {
1153         rcu_assign_pointer(t->trie, (struct rt_trie_node *)tn);
1154         tp = tn;
1155     }
1156 }
//设置后的参数合理性检查, 温和的使用了 warn。
1158     if (tp && tp->pos + tp->bits > 32)
1159         pr_warn("fib_trie tp=%p pos=%d, bits=%d, key=%0x plen=%d\n",
1160                 tp, tp->pos, tp->bits, key, plen);
1161
1162     /* Rebalance the trie */
//提到过好多次的 rebalance 操作。图 12.3.2 中黄色的列, bits 值在前面的操作都是一个比特, 在 rebalance 中可能会变成多个
//比特。
1164     trie_rebalance(t, tp);
1165 done:
1166     return fa_head;
1167 }

```

后面是发送一个消息, 网卡接收到消息后, 执行函数 `fib_netdev_event()` 函数, 这个函数会再次调用 `fib_inetaddr_event`, 这个函数这次会按顺序添加子网号全 1、全 0 以及主机号为全 1 (10.12.39.255 广播地址, 10.12.39.221 也是要接收发往这个地址的数据的)。其过程和上面的类似, 插入的节点 hash 值也是相同的 (图中表 `ifconfig` 的数组索引)。

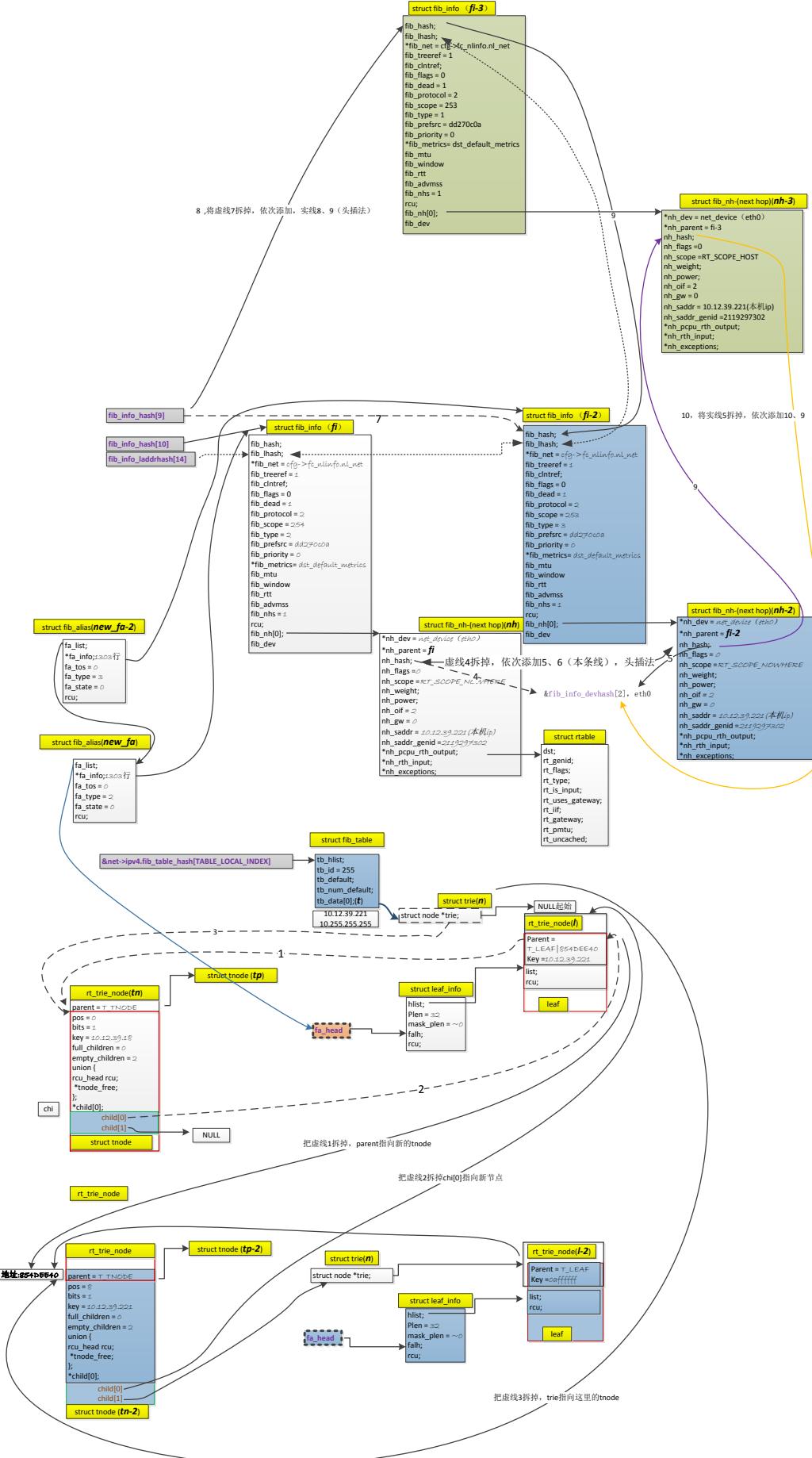


图 12.3.3 由 ifconfig 引发的前三次路由表更新过程

12.3.4 put_child

12.3.3 小节可以知道核心的插入函数是 `put_child`, 所以这里单独一个小节来看一下插入的过程是如何完成的。这节的情况还是继续 12.3.3 小节。但是只打算分析 1164 行, 其它的读者自行完成。

```
475 static inline int tnode_full(const struct tnode *tn, const struct rt_trie_node *n)
476 {
//对于 NULL 和 leaf 情况, 返回值一定是 0, 用于更新 full_children 计数器。
477     if (n == NULL || IS_LEAF(n))
478         return 0;
//对于 tnode 情况的判断方法。
480     return ((struct tnode *) n)->pos == tn->pos + tn->bits;
481 }
483 static inline void put_child(struct tnode *tn, int i,
484                             struct rt_trie_node *n)
485 {
486     tnode_put_child_reorg(tn, i, n, -1);
487 }
488
489 /*
490  * Add a child at position i overwriting the old value.
491  * Update the value of full_children and empty_children.
492 */
493
494 static void tnode_put_child_reorg(struct tnode *tn, int i, struct rt_trie_node *n,
495                                   int wasfull)
496 {
497     struct rt_trie_node *chi = rtnl_dereference(tn->child[i]);
498     int isfull;
499
//传递进来的 n 非空, chi 这是还没有获得有效值, 所以会执行 505 行的 else 语句。对于插入操作, 基本都是对
//empty_children 减一操作
502     /* update emptyChildren */
503     if (n == NULL && chi != NULL)
504         tn->empty_children++;
505     else if (n != NULL && chi == NULL)
506         tn->empty_children--;
507
//对于添加路由项, 传递进来的值都是-1
509     if (wasfull == -1)
510         wasfull = tnode_full(tn, chi);
511
512     isfull = tnode_full(tn, n);
513     if (wasfull && !isfull)
514         tn->full_children--;
515     else if (!wasfull && isfull)
516         tn->full_children++;
517
*****
最低一个 bit 设置 node 的类型, 要么 tnode, 要么 leaf, 高位存放地址 (ADD)
208 static inline void node_set_parent(struct rt_trie_node *node, struct tnode *ptr)
209 {
210     smp_wmb();
211     node->parent = (unsigned long)ptr | NODE_TYPE(node);
212 }
```

```
*****
518     if (n)
519         node_set_parent(n, tn);
520
//添加的操作实际上非常的简单，就是一个 rcu 变量的赋值，将 tnode 的 child 成员赋值成适当的值就好了。
521     rcu_assign_pointer(tn->child[i], n);
522 }
```

483 参数：

tn 是 key 值为 0a0c2700 的 tnode，

i 值是 1

i 是 key 等于 192.168.0.100 的 leaf，(先前已经创建并初始化好了)。

该函数调用了，486 tnode_put_child_reorg，完成添加工作。

479 行原来 tnode 的 child[1]是指向 key 值等于 192.168.0.10 的 leaf。

502~506 跟新孩子计数器。

509 行，判断 chi 是否“满”，由于这里的 chi 是键值为 192.168.0.10 的 leaf，所以其返回值是 0，

512 行，判断插入 leaf，192.168.0.100 是否满，显然返回值是 0。

12.4 route 添加路由项

路由这章的介绍和前面的章节分开，从工具说起，在 Linux 下经常使用 route 工具配置路由项和网关。下面的这段代码来自配置网关的用户空间程序的代码片段，从这个函数可以看出，创建一个套接字，并调用 ioctl 方法完成网关的设置。

```
memcpy(ifr.ifr_name, interface_name, sizeof(ifr.ifr_name));
(void)ioctl(sockfd, SIOGIFINDEX, &ifr);
v4_rt.rtmsg_ifindex = ifr.ifr_ifindex;

memcpy(&v4_rt.rtmsg_gateway, gateway, sizeof(struct in_addr));
/*添加路由*/
if (ioctl(sockfd, SIOCADDRT, &v4_rt) < 0)
{
    SAFE_CLOSE(sockfd);
    BASEFUN_DBG(RT_ERROR, "add route ioctl error and errno=%d\n", errno);
    return -1;
}
SAFE_CLOSE(sockfd);
```

这个 ioctl 函数对应到内核下最终的路由配置函数是 ip_rt_ioctl，该函数位于 net/ipv4/fib_front.c 文件。Linux 源码中路由的表示结构体是 fib*，FIB (forward information base)，所以后面会出现好多以 fib 开始的程序片段。

```
480 int ip_rt_ioctl(struct net *net, unsigned int cmd, void __user *arg)
481 {
482     struct fib_config cfg;
483     struct rtentry rt;
484     int err;
485
486     switch (cmd) {
487         case SIOCADDRT: /* Add a route */
488         case SIOCDELRT: /* Delete a route */
489             if (!ns_capable(net->user_ns, CAP_NET_ADMIN))
490                 return -EPERM;
491
492             if (copy_from_user(&rt, arg, sizeof(rt)))
```

```

493         return -EFAULT;
494
495         rtnl_lock();
496         err = rtentry_to_fib_config(net, cmd, &rt, &cfg);
497         if (err == 0) {
498             struct fib_table *tb;
499
500             if (cmd == SIOCDELRT) {
501                 tb = fib_get_table(net, cfg.fc_table);
502                 if (tb)
503                     err = fib_table_delete(tb, &cfg);
504                 else
505                     err = -ESRCH;
506             } else {
507                 tb = fib_new_table(net, cfg.fc_table);
508                 if (tb)
509                     err = fib_table_insert(tb, &cfg);
510                 else
511                     err = -ENOBUFS;
512             }
513
514             /* allocated by rtentry_to_fib_config() */
515             kfree(cfg.fc_mx);
516         }
517         rtnl_unlock();
518         return err;
519     }
520     return -EINVAL;
521 }

```

根据 cmd 参数 SIOCADDRT 可以跟踪到 switch 内执行的程序代码段，496 行 rtentry_to_fib_config 用于将用户空间的路由配置信息转变到内核记录配置信息的结构体 cfg 中，在图 12.4.2 中等号右边的信息就是来自用户空间的配置信息。

图 12.4.1 是使用 route 命令添加了一个路由项、一个网关和导出路由项，内核下保存配置信息的成员 cfg 是 struct fib_config 类型的一个结构体，该结构体的成员值在图 12.4.1 中列了出来。

```

include/net/ip_fib.h
26 struct fib_config {
27     u8          fc_dst_len;
28     u8          fc_tos;
29     u8          fc_protocol;
30     u8          fc_scope;
31     u8          fc_type;
32     /* 3 bytes unused */
33     u32         fc_table;
34     __be32      fc_dst;
35     __be32      fc_gw;
36     int         fc_oif;
37     u32         fc_flags;
38     u32         fc_priority;
39     __be32      fc_prefsrc;
40     struct nlattr *fc_mx;
41     struct rtnexthop *fc_mp;
42     int         fc_mx_len;
43     int         fc_mp_len;
44     u32         fc_flow;
45     u32         fc_nlflags;

```

```

46     struct nl_info      fc_nlinfo;
47 };

```

添加一个路由项

```

# route add 192.168.0.10 dev eth0
[ 513.232587] fc_dst_len:32, fc_tos:0, fc_protocol:3, fc_scope:253,
[ 513.232607] fc_type:1, fc_table:0, fc_dst:A00A8C0, fc_gw:0,
[ 513.232620] fc_oif:2, fc_flags:0, fc_priority:0, fc_prefsrc:0,
[ 513.232634] fc_mx_len:0, fc_mp_len:0, fc_flow:0, fc_nlflags:1024
#

```

添加一个网关

```

# route add default gw 10.12.39.100 eth0
[ 97.492551] fc_dst_len:0, fc_tos:0, fc_protocol:3, fc_scope:0,
[ 97.492572] fc_type:1, fc_table:0, fc_dst:0, fc_gw:64270C0A,
[ 97.492585] fc_oif:2, fc_flags:0, fc_priority:0, fc_prefsrc:0,
[ 97.492599] fc_mx_len:0, fc_mp_len:0, fc_flow:0, fc_nlflags:1024
#

```

路由项输出信息

```

# route
Kernel IP routing table
Destination     Gateway            Genmask        Flags Metric Ref    Use Iface
192.168.0.10   *                 255.255.255.255 UH    0      0        0 eth0
10.12.39.0     *                 255.255.255.0   U      0      0        0 eth0
default         10.12.39.100     0.0.0.0        UG    0      0        0 eth0
#

```

图 12.4.1 route 命令配置路由信息

根据上面信息，将各项内容表述在如下图形中：



图 12.4.2 cfg 信息

由于应用空间传递的命令是 SIOCADDRT (代码片段一)，所以执行 507~511 行的代码。
fib_new_table 用于获得一个和用户匹配类型的路由表，如果在现有的路由表中没有要找的类型，则会创建这个类型的路由表。

```

75 struct fib_table *fib_new_table(struct net *net, u32 id)
76 {
77     struct fib_table *tb;
78     unsigned int h;
79

```

```

80     if (id == 0)
81         id = RT_TABLE_MAIN;
82     tb = fib_get_table(net, id);
83     if (tb)
84         return tb;
85
86     tb = fib_trie_table(id);
87     if (!tb)
88         return NULL;
89
90     switch (id) {
91     case RT_TABLE_LOCAL:
92         net->ipv4.fib_local = tb;
93         break;
94
95     case RT_TABLE_MAIN:
96         net->ipv4.fib_main = tb;
97         break;
98
99     case RT_TABLE_DEFAULT:
100        net->ipv4.fib_default = tb;
101        break;
102
103    default:
104        break;
105    }
106
107    h = id & (FIB_TABLE_HASHSZ - 1);
108    hlist_add_head_rcu(&tb->tb_hlist, &net->ipv4.fib_table_hash[h]);
109    return tb;
110 }

111 struct fib_table *fib_get_table(struct net *net, u32 id)
112 {
113     struct fib_table *tb;
114     struct hlist_head *head;
115     unsigned int h;
116
117     if (id == 0)
118         id = RT_TABLE_MAIN;
119     h = id & (FIB_TABLE_HASHSZ - 1);
120
121     rCU_read_lock();
122     head = &net->ipv4.fib_table_hash[h];
123     hlist_for_each_entry_rcu(tb, head, tb_hlist) {
124         if (tb->tb_id == id) {
125             rCU_read_unlock();
126             return tb;
127         }
128     }
129 }
130 rCU_read_unlock();
131 return NULL;
132 }
```

82 行 `fib_get_table` 用户查找现有的路由表，路由表的类型定义在 `include/uapi/linux/rtnetlink.h` 文件中：

```

enum rt_class_t {
    RT_TABLE_UNSPEC=0,
    /* User defined values */
```

```

RT_TABLE_COMPAT=252,
RT_TABLE_DEFAULT=253,
RT_TABLE_MAIN=254,
RT_TABLE_LOCAL=255,
RT_TABLE_MAX=0xFFFFFFFF
};

```

虽然这里最大路由项的定义值是 RT_TABLE_MAX，但是其实在不启动等价路由【Equal-cost multi-path routing(ECMP) (CONFIG_IP_ROUTE_MULTIPATH)】时，配置 CONFIG_IP_MULTIPLE_TABLES 时有 256 张路由表。启用等价路由时只会有两张路由表，在存在多条网络路径的网络节点会配置使能。

图 12.4.3 的路由表拓扑结构中，net 代表了一个网络命名空间，其指针成员 ipv4 指向 inet 协议字段，ipv4 的 fib_table_hash 是一个数组，其指向是哈希链表，该表表映射的就是具体路由表的类型，路由表的类型从 0 开始一直到 255，图中第一行的 RT_TABLE_DEFAULT 就等于 253 依次类推，所以在查找路由表时，只需要记得你找的是 MAIN 表还是 LOCAL，而不需要关心是 254 还是 255 之类的数值了，这很类似于域名概念。每一个路由表由 struct fib_table 类型表示。命名空间中的哈希字段指向具体路由表的 tb_list 字段，并且处在同一个类型的路由表之间也是通过这个字段联系在一起的。

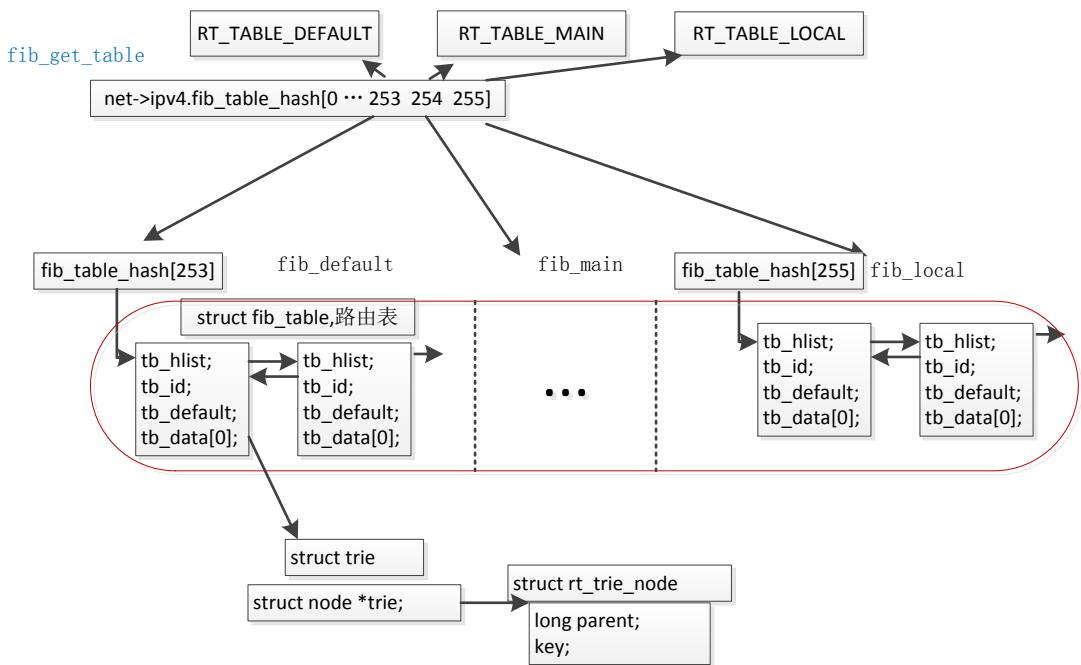


图 12.4.3 路由表拓扑

fc_table 项指示的选择哪一张表插入用户配置的路由项，fib_new_table 的 82 行 fib_get_table 根据路由表的 id(根据图 12.1.2 其传递进来的值等于 0) 字段查询指定类型的路由表。81 行将 id 赋值为 RT_TABLE_MAIN 。

120 行得到 h 等于 RT_TABLE_MAIN 。

将 123 行结合图 12.1.3 就可以看出其找到 main 表。

124~128 根据路由表的 tb_list 字段遍历路由表，寻找路由表的 tb_id 等于 RT_TABLE_MAIN 的表。找到就返回该表，没找到就返回 NULL，让其创建该表项。

83~84，如果 fib_get_table 找到了指定的表，则就是向这个表插入一个路由项，如果没有找到，说明需要创建一个这个表，接着向下 86 行即用于创建一个表。创建路由表的函数位于 fib_trie.c 文件，在 2.6.38 及以前 Linux 默认使用的是 hash 方法管理路由表和路由项，到 Linux3.10，内核默认使用 LC-trie 方法，中文里常把这个算法称为字典或者单词查找树，

路由使用的关于单词查找树的代码都在 `fib_trie.c`。这个算法在分析代码时遇到了再说。

```
net/ipv4/fib_trie.c
1970 struct fib_table *fib_trie_table(u32 id)
1971 {
1972     struct fib_table *tb;
1973     struct trie *t;
1974
1975     tb = kmalloc(sizeof(struct fib_table) + sizeof(struct trie),
1976                  GFP_KERNEL);
1977     if (tb == NULL)
1978         return NULL;
1979
1980     tb->tb_id = id;
1981     tb->tb_default = -1;
1982     tb->tb_num_default = 0;
1983
1984     t = (struct trie *) tb->tb_data;
1985     memset(t, 0, sizeof(*t));
1986
1987     return tb;
1988 }
```

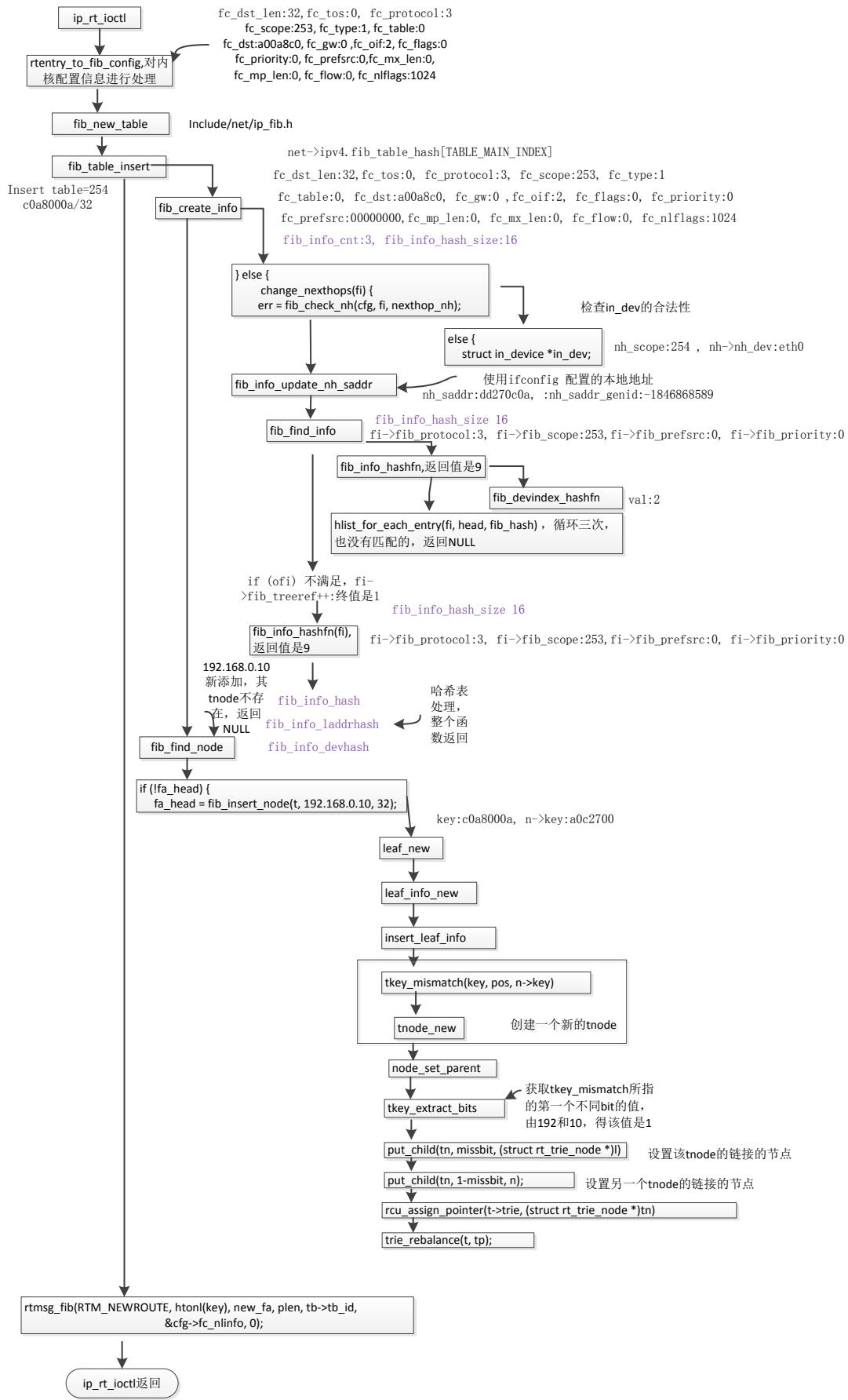
1975 行创建一个路由表，路由表的定义如下，这里的内存申请将零长数组 `tb_data` 初始化为 `struct trie` 成员。**1980~1985** 初始化该路由表的各个字段。

```
struct fib_table {
    struct hlist_node  tb_hlist;
    u32                tb_id;
    int                tb_default;
    int                tb_num_default;
    unsigned long      tb_data[0];
};
```

`fib_new_table` 的 90~105 行将网络命名空间的 `fib_main` 字段指向这里创建的 `MAIN` 表。
108 行再将 `MAIN` 表放到数组 `fib_table_hash` 里，以便后续的索引。

再回退到 `ip_rt_ioctl` 的 509 行，`fib_table_insert` 用于向路由表中插入一个路由项了。可以知道插入操作是基于 `trie` 方法的。其插入文件位于 `net/ipv4/fib_trie.c` 文件。这个函数有点长，但是还是要一行一行的过。

其参数是上面找到的或者创建的路由表，配置项就是前面用户空间配置的信息，该信息见图 12.4.2。



12.5 路由缓存

路由缓存是为了加速路由项的查找而是用的缓存技术，路由缓存涉及三个部分，输入输出路由缓存的查找、输入输出路由缓存的创建以及输入输出路由缓存的内存管理（垃圾回收等）。

12.5.1 路由缓存的查找

对于输入数据包，其路由缓存的查找函数是 ip_route_input()，参数意义：

Skb: 输入数据报

Dst、src: 用于查找的源和目的 ipv4 地址。

Tos: ip 头选项

Devin: 数据报输入设备

Include/net/route.h

```
168 static inline int ip_route_input(struct sk_buff *skb, __be32 dst, __be32 src,
169             u8 tos, struct net_device *devin)
170 {
171     int err;
172
173     rCU_read_lock();
174     err = ip_route_input_noref(skb, dst, src, tos, devin);
175     if (!err)
176         skb_dst_force(skb);
177     rCU_read_unlock();
178
179     return err;
180 }
```

由上述函数可见，其是对 ip_route_input_noref() 函数的封装。

Net/ipv4/route.c

```
1762 int ip_route_input_noref(struct sk_buff *skb, __be32 daddr, __be32 saddr,
1763             u8 tos, struct net_device *dev)
1764 {
1765     int res;
1766
1767     rCU_read_lock();
1768
1769     /* Multicast recognition logic is moved from route cache to here.
1770      The problem was that too many Ethernet cards have broken/missing
1771      hardware multicast filters :-( As result the host on multicasting
1772      network acquires a lot of useless route cache entries, sort of
1773      SDR messages from all the world. Now we try to get rid of them.
1774      Really, provided software IP multicast filter is organized
1775      reasonably (at least, hashed), it does not result in a slowdown
1776      comparing with route cache reject entries.
1777      Note, that multicast routers are not affected, because
1778      route cache entry is created eventually.
1779 */
1780     if (ipv4_is_multicast(daddr)) // 判断目的地址是否是多播地址，即是否在 D 类地址范围，224 开始。
1781         struct in_device *in_dev = __in_dev_get_rcu(dev);
1782
1783     if (in_dev) {
1784         int our = ip_check_mc_rcu(in_dev, daddr, saddr,
1785             ip_hdr(skb)->protocol);
1786         if (our
```

```

1787 #ifdef CONFIG_IP_MROUTE
1788         ||
1789         (!ipv4_is_local_multicast(daddr) &&
1790          IN_DEV_MFORWARD(in_dev))
1791 #endif
1792         ) {
1793         int res = ip_route_input_mc(skb, daddr, saddr,
1794                                     tos, dev, our);
1795         rCU_read_unlock();
1796         return res;
1797     }
1798 }
1799 rCU_read_unlock();
1800 return -EINVAL;
1801 }
1802 res = ip_route_input_slow(skb, daddr, saddr, tos, dev);
1803 rCU_read_unlock();
1804 return res;
1805 }

```

12.5.2 路由缓存的创建

在接收数据报时，网络协议栈 `ip_route_input_noref()` 被调用来查找路由项，该函数在处理完数据报目的地址是多播情况后，调用 `ip_route_input_slow()` 函数为其它类型目的地址的数据报查找路由项，`fib_lookup` 会被调用在 LOCAL 和 MAIN 表中查找路由项，基于 `trie` 树的 LOCAL 和 MAIN 只存放了 ip 地址，`trie` 树本身并没有存放流控一些信息，而流控信息又会影响路由查找的最终结果。

`ip_mkroute_input()` 是接收数据报情况的路由项创建函数。

```

1559 static int ip_mkroute_input(struct sk_buff *skb,
1560                               struct fib_result *res,
1561                               const struct flowi4 *fl4,
1562                               struct in_device *in_dev,
1563                               __be32 daddr, __be32 saddr, u32 tos)
1564 {
//多路径路由，可用于聚合操作，Linux 除了支持网络的多路径，其它资源也可以使用聚合操作，如磁盘等，这在服务器上常用。
//聚合之类的代名词有，port trunking, link bundling, bonding, teaming.
1565 #ifdef CONFIG_IP_ROUTE_MULTIPATH
1566     if (res->fi && res->fi->fib_nhs > 1)
1567         fib_select_multipath(res);
1568 #endif
1569
1570     /* create a routing cache entry */
1571     return __mkroute_input(skb, res, in_dev, daddr, saddr, tos);//创建一个路由缓存项
1572 }

```

`__mkroute_input()` 函数如下：

其 `res` 参数是 `fib_lookup()` 函数查找到的路由项结果，根据这一结果创建缓存路由项。

```

1471 static int __mkroute_input(struct sk_buff *skb,
1472                             const struct fib_result *res,
1473                             struct in_device *in_dev,
1474                             __be32 daddr, __be32 saddr, u32 tos)
1475 {
1476     struct rtable *rth;
1477     int err;
1478     struct in_device *out_dev;
1479     unsigned int flags = 0;

```

```

1480     bool do_cache;
1481     u32 itag;
1482
1483     /* get a working reference to the output device */
1484     out_dev = __in_dev_get_rcu(FIB_RES_DEV(*res)); //根据 fib_lookup()结果得到输出设备。
1485     if (out_dev == NULL) {
1486         net_crit_ratelimited("Bug in ip_route_input_slow(). Please report.\n");
1487         return -EINVAL;
1488     }
1489 //验证源地址的合法性，对于由 IPsec 保护的数据包，反向过滤技术 rp_filter (reverse-pathfiltering) 会被忽略
1490     err = fib_validate_source(skb, saddr, daddr, tos, FIB_RES_OIF(*res),
1491                             in_dev->dev, in_dev, &itag);
1492     if (err < 0) {
1493         ip_handle_martian_source(in_dev->dev, in_dev, skb, daddr,
1494                                   saddr);
1495
1496         goto cleanup;
1497     }
1498 //do_cache 变量用于标识是否创建路由缓存。
1499     do_cache = res->fi && !itag;
1500 //路由缓存项最优性检查，非最优则添加 RTCF_DOREDIRECT 标志。
1501     if (out_dev == in_dev && err && IN_DEV_TX_REDIRECTS(out_dev) &&
1502         (IN_DEV_SHARED_MEDIA(out_dev) ||
1503          inet_addr_onlink(out_dev, saddr, FIB_RES_GW(*res)))) {
1504         flags |= RTCF_DOREDIRECT;
1505         do_cache = false;
1506     }
1507 //非 IP 报文情况，不需要创建路由缓存项
1508     if (skb->protocol != htons(ETH_P_IP)) {
1509         /* Not IP (i.e. ARP). Do not create route, if it is
1510            * invalid for proxy arp. DNAT routes are always valid.
1511            *
1512            * Proxy arp feature have been extended to allow, ARP
1513            * replies back to the same interface, to support
1514            * Private VLAN switch technologies. See arp.c.
1515            */
1516         if (out_dev == in_dev &&
1517             IN_DEV_PROXY_ARP_PVLAN(in_dev) == 0) {
1518             err = -EINVAL;
1519             goto cleanup;
1520         }
1521 //验证当前路由缓存有效性，如果有效则不需要新创建一个路由缓存项。
1522     if (do_cache) {
1523         rth = rcu_dereference(FIB_RES_NH(*res).nh_rth_input);
1524         if (rt_cache_valid(rth)) {
1525             skb_dst_set_noref(skb, &rth->dst);
1526             goto out;
1527         }
1528     }
1529 //为路由缓存项分配存储空间
1530     rth = rt_dst_alloc(out_dev->dev,
1531                       IN_DEV_CONF_GET(in_dev, NOPOLICY),
1532                       IN_DEV_CONF_GET(out_dev, NOXFRM), do_cache);
1533     if (!rth) {
1534         err = -ENOBUFS;
1535         goto cleanup;
1536     }

```

```

//初始化路由缓存项
1538     rth->rt_genid = rt_genid(dev_net(rth->dst.dev));
1539     rth->rt_flags = flags;
1540     rth->rt_type = res->type;
1541     rth->rt_is_input = 1;
1542     rth->rt_iif      = 0;
1543     rth->rt_pmtu     = 0;
1544     rth->rt_gateway  = 0;
1545     rth->rt_uses_gateway = 0;
1546     INIT_LIST_HEAD(&rth->rt_uncached);
1547
1548     rth->dst.input = ip_forward;
1549     rth->dst.output = ip_output;
//根据路由缓存项 rth, 设置 next hop 成员。
1551     rt_set_nexthop(rth, daddr, res, NULL, res->fi, res->type, itag); //创建路由缓存项。
1552     skb_dst_set(skb, &rth->dst); //为收到的数据报设置路由缓存项
1553 out:
1554     err = 0;
1555 cleanup:
1556     return err;
1557 }

```

输出路由项创建函数是__mkroute_output()。

```

1809 static struct rtable * __mkroute_output(const struct fib_result *res,
1810                                         const struct flowi4 *fl4, int orig_oif,
1811                                         struct net_device *dev_out,
1812                                         unsigned int flags)
1813 {
1814     struct fib_info *fi = res->fi;
1815     struct fib_nh_exception *fnhe;
1816     struct in_device *in_dev;
1817     u16 type = res->type;
1818     struct rtable *rth;
1819     bool do_cache;
1820
1821     in_dev = __in_dev_get_rcu(dev_out);
1822     if (!in_dev)
1823         return ERR_PTR(-EINVAL);
//回环设备路由项
1825     if (unlikely(!IN_DEV_ROUTE_LOCALNET(in_dev)))
1826         if (ipv4_is_loopback(fl4->saddr) && !(dev_out->flags & IFF_LOOPBACK))
1827             return ERR_PTR(-EINVAL);
//ip 地址全为 1。
1829     if (ipv4_is_lbcast(fl4->daddr))
1830         type = RTN_BROADCAST;
//多播时
1831     else if (ipv4_is_multicast(fl4->daddr))
1832         type = RTN_MULTICAST;
//网络号全为 0, 则出错
1833     else if (ipv4_is_zeronet(fl4->daddr))
1834         return ERR_PTR(-EINVAL);
1835
1836     if (dev_out->flags & IFF_LOOPBACK)
1837         flags |= RTCF_LOCAL;
1838
1839     do_cache = true;
1840     if (type == RTN_BROADCAST) {
1841         flags |= RTCF_BROADCAST | RTCF_LOCAL;

```

```

1842         fi = NULL;
1843     } else if (type == RTN_MULTICAST) {
1844         flags |= RTCF_MULTICAST | RTCF_LOCAL;
1845         if (!ip_check_mc_rcu(in_dev, fl4->daddr, fl4->saddr,
1846                               fl4->flowi4_proto))
1847             flags &= ~RTCF_LOCAL;
1848         else
1849             do_cache = false;
1850         /* If multicast route do not exist use
1851          * default one, but do not gateway in this case.
1852          * Yes, it is hack.
1853          */
1854         if (fi && res->prefixlen < 4)
1855             fi = NULL;
1856     }
1857     fnhe = NULL;
1858     do_cache &= fi != NULL;
1859     if (do_cache) {
1860         struct rtable __rcu **prth;
1861         struct fib_nh *nh = &FIB_RES_NH(*res);
1862
1863         fnhe = find_exception(nh, fl4->daddr);
1864         if (fnhe)
1865             prth = &fnhe->fnhe_rth;
1866         else {
1867             if (unlikely(fl4->flowi4_flags &
1868                         FLOWI_FLAG_KNOWN_NH &&
1869                         !(nh->nh_gw &&
1870                           nh->nh_scope == RT_SCOPE_LINK))) {
1871                 do_cache = false;
1872                 goto add;
1873             }
1874         }
1875         prth = __this_cpu_ptr(nh->nh_pcpu_rth_output);
1876     }
1877     rth = rcu_dereference(*prth);
1878     if (rt_cache_valid(rth)) {
1879         dst_hold(&rth->dst);
1880         return rth;
1881     }
1882 }
1883
1884 add:
1885     rth = rt_dst_alloc(dev_out,
1886                         IN_DEV_CONF_GET(in_dev, NOPOLICY),
1887                         IN_DEV_CONF_GET(in_dev, NOXFRM),
1888                         do_cache);
1889     if (!rth)
1890         return ERR_PTR(-ENOBUFS);
1891
1892     rth->dst.output = ip_output;
1893
1894     rth->rt_genid = rt_genid(dev_net(dev_out));
1895     rth->rt_flags = flags;
1896     rth->rt_type = type;
1897     rth->rt_is_input = 0;
1898     rth->rt_iif = orig_oif ?: 0;
1899     rth->rt_pmtu = 0;
1900     rth->rt_gateway = 0;

```

```

1901     rth->rt_uses_gateway = 0;
1902     INIT_LIST_HEAD(&rth->rt_uncached);
1903
1904     RT_CACHE_STAT_INC(out_slow_tot);
1905
1906     if (flags & RTCF_LOCAL)
1907         rth->dst.input = ip_local_deliver;
1908     if (flags & (RTCF_BROADCAST | RTCF_MULTICAST)) {
1909         if (flags & RTCF_LOCAL &&
1910             !(dev_out->flags & IFF_LOOPBACK)) {
1911             rth->dst.output = ip_mc_output;
1912             RT_CACHE_STAT_INC(out_slow_mc);
1913         }
1914 #ifdef CONFIG_IP_MROUTE
1915         if (type == RTN_MULTICAST) {
1916             if (IN_DEV_MFORWARD(in_dev) &&
1917                 !ipv4_is_local_multicast(f14->daddr)) {
1918                 rth->dst.input = ip_mr_input;
1919                 rth->dst.output = ip_mc_output;
1920             }
1921         }
1922 #endif
1923     }
1924
1925     rt_set_nexthop(rth, f14->daddr, res, fnhe, fi, type, 0);
1926
1927     return rth;
1928 }
```

12.5.3 路由缓存的内存管理

12.6 路由查找

在 12.2 和 12.3 节，主要关注的是 trie 树的构建和管理，并不涉及路由查找的内容，由于管理和维护一个 trie 树本身是一件复杂的事，所以内核没有将查找路由表需要的信息也放在 trie 树中存储，而是使用了其它的一些数据结构来辅助查找过程，这节先引入这些数据结构，然后剖析查找过程，查找的总体思想是首先检查参数的合法性，查找 trie (tnode , leaf) 树，trie 树本身存储的是 key 值，没有其它 过多的信息，这些信息包括 tos，也即流控，所以需要查找一些辅助数据结构，查找到了以后，如果对应的路由项没有缓存，会创建一个路由缓存。

12.6.1 相关数据结构

本节所述的数据结构指的是辅助数据结构，并没有将使用到的 trie 树相关数据结构罗列出来。关于这几个数据结构间的关系见图 12.3.3。

fib_result

fib_result 用于存放路由查找的结果
 include/net/ip_fib.h
 struct fib_result {
 unsigned char prefixlen; //前缀长度
 unsigned char nh_sel; //nexthop 索引
 unsigned char type; //type 和 scope 是类型和范围

```

    unsigned char      scope;
    u32            tclassid;
    struct fib_info *fi; //路由项对应的信息
    struct fib_table *table; //该结果源于的表项
    struct list_head *fa_head; //fib alias 链表指针。
};


```

fib_info

```

struct fib_info {
    struct hlist_node   fib_hash; //fib_info 信息索引，局部全局数组 fib_info_hash 使用
    struct hlist_node   fib_lhash; //同样是局部全局数组 fib_info_laddrhash 索引
    struct net          *fib_net; //对应设备所在网络命名空间中的一些信息。
    int                fib_treeref; //trie 树引用计数
    atomic_t           fib_clnref;
    unsigned int        fib_flags;

//标记该 fib_info 是否可用，当值是 1 时，标志该路由项失效，查找时将不会参考该值。
    unsigned char        fib_dead;
    unsigned char        fib_protocol; //支持的协议
    unsigned char        fib_scope; //范围
    unsigned char        fib_type; //类型
    __be32              fib_prefsrc; //前缀
    u32                 fib_priority; //优先级
    u32                 *fib_metrics; //metrics 相关内容

#define fib_mtu fib_metrics[RTAX_MTU-1]
#define fib_window fib_metrics[RTAX_WINDOW-1]
#define fib_rtt fib_metrics[RTAX_RTT-1]
#define fib_advmss fib_metrics[RTAX_ADMSS-1]
    int                 fib_nhs; //记录 fib_nh[0] 零长数组具有的下一跳的个数。

#ifndef CONFIG_IP_ROUTE_MULTIPATH
    int                 fib_power;
#endif
    struct rcu_head      rru;
    struct fib_nh       fib_nh[0];
#define fib_dev          fib_nh[0].nh_dev
};


```

该数据结构元素的初始化实例可以参考图 12.3.3，图中的 fi、fi-2、fi-3 分别对应于三种路由项情况。

fib_alias

虽然套接字数据包路由项 **key** 可能不同，例如路由到 192.168.0.10 和路由到 192.168.0.100 的两项，但是不同套接字数据包的 **type** 或者 **tos** 是相同的，对于一个大规模的路由而言，的类型不同套接字数据包的 **type** 和 **tos** 相同的概率还是比较大的，为了节约空间和时间，就将这些字段抽象出来了，**tos** 和 **type** 组合的类型有限，这样可以使这个有限的组合应对无限多路由项。**fa_list** 用于串接所有的 **fib_alias** 类型的结构体，以便于管理 **fa_alias** 结构体。

```

struct fib_alias {
    struct list_head   fa_list;
    struct fib_info     *fa_info; //见上
    u8                  fa_tos;
    u8                  fa_type; // UNICAST、LOCAL、BROADCAST
    u8                  fa_state;
    struct rcu_head     rru;
};


```

fib_flowi4

该数据结构在图 12.3.3 中并未显示出来，路由查找过程中会使用到该数据结构

结构根据输入输出网络设备、ip 层和 tcp 层一些字段对流量进行分类。flowi4 (flow_inet_4) 是 Internet 4 协议的流控之意。对应的还有 flowi6 的流控，其实该结构体就是对 flowi_common 的封装，这么做好处是不言而喻的，提高代码复用率的同时增加了代码维护的灵活性。

```
struct flowi4 {
    struct flowi_common    __fl_common;
#define flowi4_oif      __fl_common.flowic_oif //output Interface
#define flowi4_iif      __fl_common.flowic_iif //input interface
#define flowi4_mark     __fl_common.flowic_mark
#define flowi4_tos      __fl_common.flowic_tos
#define flowi4_scope    __fl_common.flowic_scope
#define flowi4_proto    __fl_common.flowic_proto
#define flowi4_flags    __fl_common.flowic_flags
#define flowi4_secid   __fl_common.flowic_secid

    /* (saddr,daddr) must be grouped, same order as in IP header */
    __be32            saddr; //源地址
    __be32            daddr;//目的地址

    union flowi_uli    uli;
#define fl4_sport       uli.ports.sport //tcp 层源端口号
#define fl4_dport       uli.ports.dport//tcp 层目的端口号
#define fl4_icmp_type  uli.icmpt.type
#define fl4_icmp_code  uli.icmpt.code
#define fl4_ipsec_spi  uli.spi
#define fl4_mh_type    uli.mht.type
#define fl4_gre_key    uli.gre_key
} __attribute__((__aligned__(BITS_PER_LONG/8)));
```

fib_common

```
struct flowi_common {
    int    flowic_oif;
    int    flowic_iif;
    __u32 flowic_mark;
    __u8   flowic_tos;
    __u8   flowic_scope;
    __u8   flowic_proto;
    __u8   flowic_flags;
#define FLOWI_FLAG_ANYSRC      0x01
#define FLOWI_FLAG_CAN_SLEEP   0x02
#define FLOWI_FLAG_KNOWN_NH    0x04
    __u32 flowic_secid;
};
```

rtable

套接字将会绑定该结构体。

```
struct rtable {
    struct dst_entry   dst;

    int        rt_genid;
    unsigned int rt_flags;
    __u16      rt_type;
    __u8       rt_is_input;
    __u8       rt_uses_gateway;

    int        rt_iif;
```

```

/* Info on neighbour */
__be32 rt_gateway;

/* Miscellaneous cached information */
u32 rt_pmtu;

struct list_head rt_uncached;
};


```

dst_entry

```

struct dst_entry {
    struct rcu_head          rCU_head;
    struct dst_entry *child;
    struct net_device *dev;
    struct dst_ops *ops;
    unsigned long _metrics;
    unsigned long expires;
    struct dst_entry *path;
    struct dst_entry *from;
#define CONFIG_XFRM
    struct xfrm_state *xfrm;
#else
    void __pad1;
#endif
    int (*input)(struct sk_buff *);
    int (*output)(struct sk_buff *);

    unsigned short flags;
#define DST_HOST 0x0001
#define DST_NOXFRM 0x0002
#define DST_NOPOLICY 0x0004
#define DST_NOHASH 0x0008
#define DST_NOCACHE 0x0010
#define DST_NOCOUNT 0x0020
#define DST_NOPEER 0x0040
#define DST_FAKE_RTABLE 0x0080
#define DST_XFRM_TUNNEL 0x0100
#define DST_XFRM_QUEUE 0x0200

    unsigned short pending_confirm;

    short error;

    /* A non-zero value of dst->obsolete forces by-hand validation
     * of the route entry. Positive values are set by the generic
     * dst layer to indicate that the entry has been forcefully
     * destroyed.
     *
     * Negative values are used by the implementation layer code to
     * force invocation of the dst_ops->check() method.
     */
    short obsolete;
#define DST_OBSOLETE_NONE 0
#define DST_OBSOLETE_DEAD 2
#define DST_OBSOLETE_FORCE_CHK -1
#define DST_OBSOLETE_KILL -2
    unsigned short header_len; /* more space at head required */
};


```

```

        unsigned short           trailer_len; /* space to reserve at tail */

#endif CONFIG_IP_ROUTE_CLASSID
        __u32                  tclassid;
#else
        __u32                  __pad2;
#endif

/*
 * Align __refcnt to a 64 bytes alignment
 * (L1_CACHE_SIZE would be too much)
 */
#endif CONFIG_64BIT
        long                  __pad_to_align_refcnt[2];
#endif

/*
 * __refcnt wants to be on a different cache line from
 * input/output/ops or performance tanks badly
 */
atomic_t      __refcnt; /* client references*/
int          __use;
unsigned long    lastuse;
union {
        struct dst_entry   *next;
        struct rtable __rcu *rt_next;
        struct rt6_info      *rt6_next;
        struct dn_route __rcu  *dn_next;
};

};

};
```

12.6.2 接收包路由项查找

在 `ip_input.c` 文件中，`ip_rcv_finish()` 函数用于处理接收到的数据包，这里将重心倾向于路由这块。`skb` 中没有找到路由项，即缓存中寻找路由项失败，需要调用 `ip_route_input_noref` 到路由表中查找，如果是回环包，则 `skb` 的路由缓存有路由项，即路由 `cache` 命中。

*多播地址寻找路由项函数: ip_route_input_mc

*单播地址寻找路由项函数: ip_route_input_slow

```
314 static int ip_rcv_finish(struct sk_buff *skb)
315 {
//根据套接字获得 ip 头
316     const struct iphdr *iph = ip_hdr(skb);
317     struct rtable *rt;
/ sysctl_ip_early_demux 是二进制值，该值用于对发往本地数据包的优化。当前仅对建立连接的套接字起作用。
319     if (sysctl_ip_early_demux && !skb_dst(skb)) {
320         const struct net_protocol *ipprot;
321         int protocol = iph->protocol;
322
323         ipprot = rcu_dereference(inet_protos[protocol]);
324         if (ipprot && ipprot->early_demux) {
325             ipprot->early_demux(skb);
326             /* must reload iph, skb->head might have changed */
327             iph = ip_hdr(skb);
328         }
329     }
330
//如果套接字的 dst 字段没有指向一个路由项，如果没有则调用 ip_route_input_noref 进行查找。
```

```

335         if (!skb_dst(skb)) {
336             int err = ip_route_input_noref(skb, iph->daddr, iph->saddr,
337                                         iph->tos, skb->dev);
338             if (unlikely(err)) {
339                 if (err == -EXDEV)
//更新基于 tcp/ip 因特网的 MIB (management information base) 信息, RFC1213
340                     NET_INC_STATS_BH(dev_net(skb->dev),
341                                     LINUX_MIB_IPRPFILTER);
342                 goto drop;
343             }
344         }
345
//对套接字可选字段的处理。ip_rcv_options(skb)会调用 ip_options_rcv_srr(skb)
357         if (iph->ihl > 5 && ip_rcv_options(skb))
358             goto drop;
//获得路由表
360         rt = skb_rtable(skb);
//多播和广播时的信息传递。
361         if (rt->rt_type == RTN_MULTICAST) {
362             IP_UPD_PO_STATS_BH(dev_net(rt->dst.dev), IPSTATS_MIB_INMCAST,
363                                 skb->len);
364         } else if (rt->rt_type == RTN_BROADCAST)
365             IP_UPD_PO_STATS_BH(dev_net(rt->dst.dev), IPSTATS_MIB_INBCAST,
366                                 skb->len);
/*向 tcp 层传递 packet*/
368         return dst_input(skb);
373 }

```

336 行 ip_route_input_noref 对于 tcp/ip 接收数据包进行路由寻址。

```

net/ipv4/route.c
/*参数的意义
skb: 传递进来的 skb_buff,
dst: 目的地地址
src: 源地址
tos: type of service, ip 头中的服务类型
devin: 网卡设备
*/
int ip_route_input_noref(struct sk_buff *skb, __be32 daddr, __be32 saddr,
1763                         u8 tos, struct net_device *dev)
1764 {
1765     int res;
1766
1767     rcu_read_lock();
1768
//多播的处理
1780     if (ipv4_is_multicast(daddr)) {
1781         struct in_device *in_dev = __in_dev_get_rcu(dev);
1782
1783         if (in_dev) {
1784             int our = ip_check_mc_rcu(in_dev, daddr, saddr,
1785                                       ip_hdr(skb)->protocol);
1786             if (our
1787                 ) {
1788                 int res = ip_route_input_mc(skb, daddr, saddr,
1789                                         tos, dev, our);
1790                 rcu_read_unlock();
1791                 return res;
1792             }
1793         }
1794     }

```

```

1798         }
1799         rCU_read_unlock();
1800         return -EINVAL;
1801     }
//除多播以外情况的处理。
1802     res = ip_route_input_slow(skb, daddr, saddr, tos, dev);
1803     rCU_read_unlock();
1804     return res;
1805 }

```

1793 行是多播地址寻找路由项函数 ip_route_input_mc。

```

1373 static int ip_route_input_mc(struct sk_buff *skb, __be32 daddr, __be32 saddr,
1374                               u8 tos, struct net_device *dev, int our)
1375 {
//申请并初始化 dst_entry，由于 rtable 的第一个成员就是 dst_entry，多以这里直接进行赋值，没有使用策略路由
1403     rth = rt_dst_alloc(dev_net(dev)->loopback_dev,
1404                         IN_DEV_CONF_GET(in_dev, NOPOLICY), false, false);
1405     if (!rth)
1406         goto e_nobufs;
1407
//初始化 rtable 字段的其它项。
1411     rth->dst.output = ip_rt_bug;
1412
1413     rth->rt_genid = rt_genid(dev_net(dev));
1414     rth->rt_flags = RTCF_MULTICAST;
1415     rth->rt_type = RTN_MULTICAST;
1416     rth->rt_is_input= 1;
1417     rth->rt_iif = 0;
1418     rth->rt_pmtu = 0;
1419     rth->rt_gateway = 0;
1420     rth->rt_uses_gateway = 0;
1421     INIT_LIST_HEAD(&rth->rt_uncached);
1422     if (our) {
1423         rth->dst.input= ip_local_deliver;
1424         rth->rt_flags |= RTCF_LOCAL;
1425     }
//将 rtable 的 dst 成员地址赋值给 skb。
1433     skb_dst_set(skb, &rth->dst);
1434     return 0;
1442 }

```

ip_route_input_noref 根据传递进来的目的地址判断是多播还是单播，多播使用 ip_route_input_mc(skb, daddr, saddr, tos, dev, our) 处理，单播使用 ip_route_input_slow(skb, daddr, saddr, tos, dev)。为了使函数的脉络看起来更为清晰，这里省去函数变量的定义、路由地址的合法性检查以及一些错误处理代码，只保留了正常情况下路由处理相关代码。

```

1585 static int ip_route_input_slow(struct sk_buff *skb, __be32 daddr, __be32 saddr,
1586                               u8 tos, struct net_device *dev)
1587 {
//上面之所以要检查源和目的地址，是因为路由会使用该信息。
//流量分类信息初始化，也是流控
1637     fl4.flowi4_oif = 0;
1638     fl4.flowi4_iif = dev->ifindex;
1639     fl4.flowi4_mark = skb->mark;
1640     fl4.flowi4_tos = tos;
1641     fl4.flowi4_scope = RT_SCOPE_UNIVERSE;
1642     fl4.daddr = daddr;

```

```

1643     fl4.saddr = saddr;
//路由查找路由查找的结果存放在 res (results) 里。
1644     err = fib_lookup(net, &fl4, &res);
1645     if (err != 0)
1646         goto no_route;
//根据路由查找结果，创建一个路由缓存项。
1667     err = ip_mkroute_input(skb, &res, &fl4, in_dev, daddr, saddr, tos);
1668 out:    return err;
1669
1760 }

```

1644 行路由查找函数

```

include/net/ip_fib.h
219 static inline int fib_lookup(struct net *net, const struct flowi4 *fip,
220                               struct fib_result *res)
221 {
222     struct fib_table *table;
//获得 id 等于 LOCAL 的路由，见 12.3 节。并查找其中的路由项， 查找过程见后面。
224     table = fib_get_table(net, RT_TABLE_LOCAL);
225     if (!fib_table_lookup(table, fip, res, FIB_LOOKUP_NOREF))
226         return 0;
//获得 id 等于 MAIN 的路由，见 12.3 节。并查找其中的路由项， 查找过程见后面。
228     table = fib_get_table(net, RT_TABLE_MAIN);
229     if (!fib_table_lookup(table, fip, res, FIB_LOOKUP_NOREF))
230         return 0;
231     return -ENETUNREACH;
232 }

```

225 行和 229 行的函数和 12.3 节提到的大多数函数一样，也位于 `fib_trie.c` 函数里。它是路由查找的核心函数，由于这个函数有些部分直接或者间接在 12.3 节有过叙述，该函数看起来也稍微容易些。这个函数先查找 `tnode` 然后查找 `leaf`，查询的结果

```

1405 int fib_table_lookup(struct fib_table *tb, const struct flowi4 *fip,
1406                         struct fib_result *res, int fib_flags)
1407 {
1408     struct trie *t = (struct trie *) tb->tb_data;
1409     int ret;
1410     struct rt_trie_node *n;
1411     struct tnode *pn;
1412     unsigned int pos, bits;
1413     t_key key = ntohl(fip->daddr);
1414     unsigned int chopped_off;
1415     t_key cindex = 0;
1416     unsigned int current_prefix_length = KEYLENGTH;
1417     struct tnode *cn;
1418     t_key pref_mismatch;
1419
1420     rCU_read_lock();
1421
1422     n = rcu_dereference(t->trie);
1423     if (!n)
1424         goto failed;
//首先查看 fib_table 指向的是否仅仅是 leaf，而没有 tnode，对于 fib_table 只有一个 leaf 的情况下，直接调用 check_leaf 进行
//验证
1430     /* Just a leaf? */
1431     if (IS_LEAF(n)) {
1432         ret = check_leaf(tb, t, (struct leaf *)n, key, fip, res, fib_flags);

```

```

1433         goto found;
1434     }
//对于是 tnode 的情况的处理
1436     pn = (struct tnode *) n;
//该变量用于记录已经匹配到的比特数。
1437     chopped_off = 0;
1438
1439     while (pn) {
1440         pos = pn->pos;
1441         bits = pn->bits;
1443         if (!chopped_off)
//找到不同的 bit, 这是为了获得孩子节点。
1444             cindex = tkey_extract_bits(mask_pfx(key, current_prefix_length),
1445                                         pos, bits);
//获得孩子节点, 这个孩子节点可能是 tnode 也可能是 leaf
1447     n = tnode_get_child_rcu(pn, cindex);
//
1449     if (n == NULL) {
1453         goto backtrace;
1454     }
//如果孩子节点是一个 leaf 节点, 则调用 check_leaf 检查是否是需要的路由项, 并将结果存放在 res 中。
1456     if (IS_LEAF(n)) {
1457         ret = check_leaf(tb, t, (struct leaf *)n, key, flp, res, fib_flags);
1458         if (ret > 0)
1459             goto backtrace;
1460         goto found;
1461     }
//如果孩子节点是一个 tnode, 则需要进行迭代到其孩子进行上述查找过程。
1463     cn = (struct tnode *)n;
//第一次进入该函数这里的 current_prefix_length 的长度是不会小于 pos+bits 的。
1494     if (current_prefix_length < pos+bits) {
1495         if (tkey_extract_bits(cn->key, current_prefix_length,
1496                               cn->pos - current_prefix_length)
1497                         || !(cn->child[0]))
1498             goto backtrace;
1499     }
1532
1533     pref_mismatch = mask_pfx(cn->key ^ key, cn->pos);
//当根据 pos 和 bits 值没有找到搜索的 key 的话, 进入前缀匹配模式。
/*************************************************************/
***该模式存在意义如下:
***对于 ipv4 路由表有如下两项:
***192.168.20.16/28
***192.168.0.0/16
***如果需要查找 192.168.20.19 则上面两个都是匹配的, 但是取哪个好呢? 内核使用最常匹配原则, 即认为 192.168.20.16
***子网掩码长度是 28)这一项是匹配的, 通常 default 项的前缀是最短的, 其作用是在其他路由项均不能匹配时会使用***default
*项 [摘自维基百科, longest prefix match], 这个函数的 chopped_off 就是忽略 prefix 的长度, 这样匹配成功的概率会***变大。
对于图 12.2.4 的情况的 trie 树, 查找 192.168.0.100 路由项的情况是不会进入 backtrace 标号开始的语句的。
************************************************************/
1540     if (pref_mismatch) {
1541         /* fls(x) = __fls(x) + 1 */
1542         int mp = KEYLENGTH - __fls(pref_mismatch) - 1;
1543
1544         if (tkey_extract_bits(cn->key, mp, cn->pos - mp) != 0)
1545             goto backtrace;
1546
1547         if (current_prefix_length >= cn->pos)
1548             current_prefix_length = mp;

```

```

1549         }
1550
1551         pn = (struct tnode *)n; /* Descend */
1552         chopped_off = 0;
1553         continue;
1554
1555 backtrace:
1556         chopped_off++;
1557
1558         /* As zero don't change the child key (cindex) */
1559         while ((chopped_off <= pn->bits)
1560                 && !(cindex & (1<<(chopped_off-1))))
1561                 chopped_off++;
1562
1563         /* Decrease current_... with bits chopped off */
1564         if (current_prefix_length > pn->pos + pn->bits - chopped_off)
1565             current_prefix_length = pn->pos + pn->bits
1566             - chopped_off;
1567
1568         /*
1569          * Either we do the actual chop off according or if we have
1570          * chopped off all bits in this tnode walk up to our parent.
1571          */
1572
1573         if (chopped_off <= pn->bits) {
1574             cindex &= ~(1 << (chopped_off-1));
1575         } else {
1576             struct tnode *parent = node_parent_rcu((struct rt_trie_node *) pn);
1577             if (!parent)
1578                 goto failed;
1579
1580             /* Get Child's index */
1581             cindex = tkey_extract_bits(pn->key, parent->pos, parent->bits);
1582             pn = parent;
1583             chopped_off = 0;
1584
1585 found:
1586     rCU_read_unlock();
1587     return ret;
1588 }
```

回到 ip_route_input_slow 的 1667 行，在没有使用多路路由技术的情况下，只是对 __mkroute_input() 函数的封装。该函数用于为接收到的套接字数据创建路由项缓存。该函数的第二个参数 res 是前面查找的结果。

```

net/ipv4/route.c
1471 static int __mkroute_input(struct sk_buff *skb,
1472                             const struct fib_result *res,
1473                             struct in_device *in_dev,
1474                             __be32 daddr, __be32 saddr, u32 tos)
1475 {
1476     struct rtable *rth;
1477     int err;
1478     struct in_device *out_dev;
1479     unsigned int flags = 0;
1480     bool do_cache;
1481     u32 itag;
1482 }
```

```

//该函数给了数据包的源地址、输入 Interface 以及目的地址、oif、tos; 检查源地址的正确性，例如不能是广播地址和 local 地
//址,
1490     err = fib_validate_source(skb, saddr, daddr, tos, FIB_RES_OIF(*res),
1491                               in_dev->dev, in_dev, &itag);
1492     if (err < 0) {
1493         ip_handle_martian_source(in_dev->dev, in_dev, skb, daddr,
1494                                   saddr);
1495
1496         goto cleanup;
1497     }
1498
//创建一个 dst_entry 入口项，并将其赋值给 rth, rtable 的第一个字段就是指向 dst_entry。该函数还对 dst_entry 进行了初始化。
1530     rth = rt_dst_alloc(out_dev->dev,
1531                       IN_DEV_CONF_GET(in_dev, NOPOLICY),
1532                       IN_DEV_CONF_GET(out_dev, NOXFRM), do_cache);
1533     if (!rth) {
1534         err = -ENOBUFS;
1535         goto cleanup;
1536     }
//rtable 相关字段初始化
1538     rth->rt_genid = rt_genid(dev_net(rth->dst.dev));
1539     rth->rt_flags = flags;
1540     rth->rt_type = res->type;
1541     rth->rt_is_input = 1;
1542     rth->rt_iif = 0;
1543     rth->rt_pmtu = 0;
1544     rth->rt_gateway = 0;
1545     rth->rt_uses_gateway = 0;
1546     INIT_LIST_HEAD(&rth->rt_uncached);
1547
1548     rth->dst.input = ip_forward;
1549     rth->dst.output = ip_output;
//rtable 的最后一项是 nexthop, 这里设置 rth 的 nexthop 项, 并设置路由缓存。
1551     rt_set_nexthop(rth, daddr, res, NULL, res->fi, res->type, itag);
//将 rtable 的 dst_entry 入口项设置成 skb 的路由项。4.2 节的路由内容至此结束。
1552     skb_dst_set(skb, &rth->dst);
1553 out:
1554     err = 0;
1555 cleanup:
1556     return err;
1557 }
```

第十三章 网络命名空间

有两篇翻译博文《[Lxc 之二—网络设置](#)》和《[linux namespace-之使用](#)》；LXC 文章中关于网络的设置是从用户空间配置的，从该文章可以知道网络命名空间的一些基本概念和其提供的功能。而《[linux namespace-之使用](#)》包括了网络命名空间管理、配置以及使用，这比 LXC 译文更接近网络命名空间的实现，但是都是基于用户空间的，这章是关于 Linux 网络命名空间内核源码。在 Linux 中，每一个网络空间都使用 struct net 表示。

13.1 命名空间创建

在当前 Linux 下，对进程而言一个命名空间包括五个具体的命名空间，mnt、uts、ipc、pid 以及 net，在 os 启动时，其会初始化一个称为 init 的初始命名空间，并将该命名空间给创建的进程，在后续创建进程时，将根据创建的 flag 确定是否创建新的命名空间。在图 13.1.1 中，进程 1、2 都指向了 init 命名空间，在创建进程 3 时，明确指定了复制网络命名空间，其它的命名空间依然会继承（copy）。

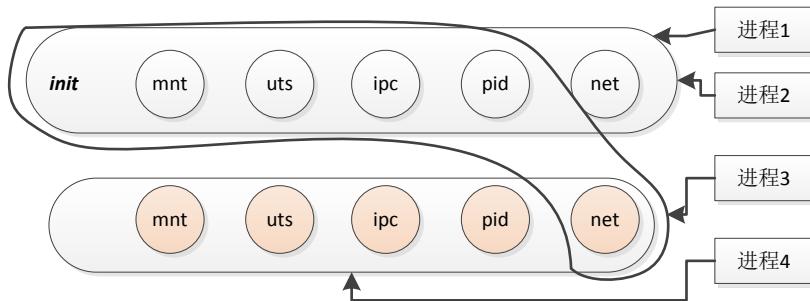


图 13.1.1 命名空间和进程的组合

创建命名空间的系统调用如下，nstype 是创建进程时指定的创建命名空间的标志。

`kernel/nsproxy.c`

```
239 SYSCALL_DEFINE2(setns, int, fd, int, nstype)
240 {
241     const struct proc_ns_operations *ops;
242     struct task_struct *tsk = current; 获得当前进程描述结构体
243     struct nsproxy *new_nsproxy;
// 创建一个新的命名空间的调用
258     new_nsproxy = create_new_namespaces(0, tsk, current_user_ns(), tsk->fs);
//proc 目录下信息注册
264     err = ops->install(new_nsproxy, ei->ns);
//将新创建的命名空间 new_nsproxy 赋值给当前进程，即替换掉以前的命名空间。
269     switch_task_namespaces(tsk, new_nsproxy);
273 }
```

258 调用的函数依然在 `nsproxy.c` 文件，该函数定义见 59 行。

```
59 static struct nsproxy *create_new_namespaces(unsigned long flags,
60     struct task_struct *tsk, struct user_namespace *user_ns,
61     struct fs_struct *new_fs)
62 {
63     struct nsproxy *new_ns;
64     int err;
//从 nsproxy_cachep 中申请一个 nsproxy 缓存，并将引用计数置 1,
66     new_ns = create_nsproxy();
/mnt 命名空间
```

```

70     new_nsp->mnt_ns = copy_mnt_ns(flags, tsk->nsproxy->mnt_ns, user_ns, new_fs);
//uts 命名空间
76     new_nsp->uts_ns = copy_utsname(flags, user_ns, tsk->nsproxy->uts_ns);
//ipc 命名空间
82     new_nsp->ipc_ns = copy_ipcs(flags, user_ns, tsk->nsproxy->ipc_ns);
//ns 命名空间
88     new_nsp->pid_ns = copy_pid_ns(flags, user_ns, tsk->nsproxy->pid_ns);
//网络命名空间
94     new_nsp->net_ns = copy_net_ns(flags, user_ns, tsk->nsproxy->net_ns);
117 }

```

类似图 13.1.1 中进程 3 那样需要部分复制命名空间时（clone 系统调用），
copy_namespaces() 将被调用。

```

123 int copy_namespaces(unsigned long flags, struct task_struct *tsk)
124 {
125     struct nsproxy *old_ns = tsk->nsproxy; //当前进程命名空间
126     struct user_namespace *user_ns = task_cred_xxx(tsk, user_ns);
127     struct nsproxy *new_ns;
128     int err = 0;
129
130     if (!old_ns)
131         return 0;
//引用计数原子加 1
133     get_nsproxy(old_ns);
//复制标志
135     if (!(flags & (CLONE_NEWNS | CLONE_NEWUTS | CLONE_NEWIPC |
136                     CLONE_NEWPID | CLONE_NEWWNET)))
137         return 0;
//权限
139     if (!ns_capable(user_ns, CAP_SYS_ADMIN)) {
140         err = -EPERM;
141         goto out;
142     }
143
//在复制 ipc 空间时，存在不同命名空间信号量的问题，切换到新的命名空间意味着原来的信号量将不可达，undolist（未处理
//信号的链表必须和新的 ipc 空间分离），如果标志设置有 CLONE_SYSVSEM，意味着要求信号量共享，那就意味着不能创
//建新的 ipc 空间了。
151     if ((flags & CLONE_NEWIPC) && (flags & CLONE_SYSVSEM)) {
152         err = -EINVAL;
153         goto out;
154     }
//创建命名空间
156     new_ns = create_new_namespaces(flags, tsk, user_ns, tsk->fs);
157     if (IS_ERR(new_ns)) {
158         err = PTR_ERR(new_ns);
159         goto out;
160     }
//将进程的命名空间空间指针指向新的命名空间
162     tsk->nsproxy = new_ns;
163
164 out:
165     put_nsproxy(old_ns);
166     return err;
167 }

```

13.2 网络命名空间管理

网络命名空间操作主要在 `net/core/net_namespace.c`, 接 13.1 节的 `copy_net_ns()` 函数。该函数返回值是一个网络空间。创建网络命名空间时以创建网络命名空间进程的 `nsproxy->net_ns` 为网络命名空间的原型, 这是因为不同的网络命名空间很多的基础设施是一样的, 比如对 `loopback` 回环接口的支持、`TCP` 的支持、`UDP` 支持、`IP` 支持等等。

```
238 struct net *copy_net_ns(unsigned long flags,
239                         struct user_namespace *user_ns, struct net *old_net)
240 {
241     struct net *net;
242     int rv;
//如果 flag 中 CLONE_NEWNET 没有设置, 说明不需要创建新的网络命名空间, 直接返回原来的命名空间。否则创建新的网
//络命名空间
244     if (!(flags & CLONE_NEWNET))
245         return get_net(old_net);
//从 net_cachep 为新的网络命名空间申请内存。
247     net = net_alloc();
//增加 user 命名空间的引用计数
251     get_user_ns(user_ns);
252
253     mutex_lock(&net_mutex);
//调用 pernet_list 上注册的服务函数, 对这个新的网络命名空间执行 init。
254     rv = setup_net(net, user_ns);
255     if (rv == 0) {
256         rtnl_lock();
257         list_add_tail_rcu(&net->list, &net_namespace_list); // net_namespace_list 为所有命名空间串接的链表。
258         rtnl_unlock();
259     }
260     mutex_unlock(&net_mutex);
261     if (rv < 0) { //出错处理
//user 命名空间计数值减一
262         put_user_ns(user_ns);
//释放 net 命名空间。
263         net_drop_ns(net);
264         return ERR_PTR(rv);
265     }
266     return net;
267 }
```

254 行的 `setup_net()` 如下, 其主要工作就是遍历 `pernet_list` 获得对应的 `struct pernet_operations` 结构体, 然后将 `struct pernet_operations` 结构对应的实例的 `init` 成员函数作用于新创建的命名空间。

```
150 static __net_init int setup_net(struct net *net, struct user_namespace *user_ns)
151 {
166     list_for_each_entry(ops, &pernet_list, list) {
167         error = ops_init(ops, net);
168         if (error < 0)
169             goto out_undo;
170     }
189 }
```

`struct pernet_operations` 的实例还是很多的, 以下截图展示了其中的一部分, 对于 `arp.c` 文件, 其 `arp_ops` 的 `arp_net_init()` 实例会在 167 行被调用, 以初始化新网络命名空间 `arp` 协议在 `proc` 目录的接口。

```
④ Af_inet.c (net\ipv4 - 副 本 ):1649
    static __net_initdata struct pernet_operations ipv4_mib_ops = {
        .init = ipv4_mib_init_net,
        .exit = ipv4_mib_exit_net,
④ Af_inet.c (net\ipv4):1648
    static __net_initdata struct pernet_operations ipv4_mib_ops = {
        .init = ipv4_mib_init_net,
        .exit = ipv4_mib_exit_net,
④ Arp.c (net\ipv4):1433
    static struct pernet_operations arp_net_ops = {
        .init = arp_net_init,
        .exit = arp_net_exit,
④ Arptable_filter.c (net\ipv4\netfilter):59
    static struct pernet_operations arptable_filter_net_ops = {
        .init = arptable_filter_net_init,
        .exit = arptable_filter_net_exit,
④ Arp_tables.c (net\ipv4\netfilter):1870
    static struct pernet_operations arp_tables_net_ops = {
        .init = arp_tables_net_init,
        .exit = arp_tables_net_exit,
④ Blocklayout.c (fs\nfs\blocklayout):1412
    static struct pernet_operations nfs4blocklayout_net_ops = {
        .init = nfs4blocklayout_net_init,
        .exit = nfs4blocklayout_net_exit,
④ Bond_main.c (drivers\net\bonding):4960
    static struct pernet_operations bond_net_ops = {
        .init = bond_net_init,
        .exit = bond_net_exit,
④ Br.c (net\bridge):29
    static struct pernet_operations br_net_ops = {
        .exit = br_net_exit,
    };
};
```

图 13.2.1 struct pernet_operations 实例

其它网络命名空间的函数如下：

get_net_ns_by_fd()根据文件描述获得网络命名空间

get_net_ns_by_pid()根据进程 ID 获得网络命名空间。

net_ns_init()网络命名空间初始化，pure_initcall 声明，在系统启动时会被调用初始化网络命名空间。

register_pernet_subsys()注册图 13.2.1 中的各种操作集，调用 register_pernet_operations()，完成，实质工作在__register_pernet_operations()函数中完成真正的注册。

unregister_pernet_subsys 注销

第十四章 netlink 机制

Netlink 基于网络的消息机制，能够让用户和内核空间进行通信，12.3 节提到的 ifconfig 是使用 ioctl 方法和内核通信的，而 ip 命令则是使用 netlink 和内核通信的。该机制初衷是为网络服务的，但是现在起应用范围已经大大扩展。

14.1 netlink 支持的通信

用户空代码使用实例，发送消息时内核使用同一套代码，也就是说调用这套消息机制代码除了可以发送 netlink 消息还可以发送其它消息，但是这些消息又各有不同，并且 netlink 本身也分为好多种，内核在处理这些不同时，使用了两个结构体解决这个问题。

```
struct msghdr {
    void * msg_name; /* Socket name */
    int msg_namelen; /* Length of name */
    struct iovec * msg iov; /* Data blocks */
    __kernel_size_t msg_ivolen; /* Number of blocks */
    void * msg_control; /* Per protocol magic (eg BSD file descriptor passing) */
    __kernel_size_t msg_controllen; /* Length of cmsg list */
    unsigned int msg_flags;
};
```

该结构体用于描述不同的消息，msg iov 存放的是消息内容，针对 netlink 消息有 nlmsghdr 头来描述。

```
struct nlmsghdr {
    __u32 nlmsg_len; /* Length of message including header */
    __u16 nlmsg_type; /* Message content */
    __u16 nlmsg_flags; /* Additional flags */
    __u32 nlmsg_seq; /* Sequence number */
    __u32 nlmsg_pid; /* Sending process port ID */
};
```

下面的代码片段展示了 netlink 的基本用法。

```
12 #define SEND_TEST_DATA "Hello Word"
13
14 struct event_msg{
15     unsigned int event;
16     unsigned int sub_event;
17     unsigned int len;
18     unsigned char data[0];
19 };
20 /* DEMO SUB EVENT */
21 #define LOOP_UNICAST 1
22 #define LOOP_BROADCAST 2
23
24 #define PRIVATE_EVENT_GROUP 2
25 #define NETLINK_TEST 17
26 #define MAX_PAYLOAD 512 /* maximum payload size*/
27 #define TEST_CNT 100000
28
29 struct hello_info {
30     unsigned int idx; //idx
31     unsigned int irq_type;
32     unsigned long timestamp; //jiffies
33 };
34
```

```

35
36
37 int main(int argc, char* argv[])
38 {
39     int i;
40     struct sockaddr_nl src_addr, dest_addr;
41     struct nlmsghdr *nlh = NULL;
42     struct iovec iov;
43     int sock_fd;
44     struct msghdr message,recv_msg;
45     struct event_msg *msg;
46     struct alarm_info *alarm_info;

//创建 netlink 套接字，第三个参数是 netlink 协议类型,所有的类型见下文。
48     sock_fd = socket(PF_NETLINK, SOCK_RAW,NETLINK_TEST);
49     memset(&message, 0, sizeof(message));
50     memset(&src_addr, 0, sizeof(src_addr));
51     src_addr.nl_family = AF_NETLINK;
52     src_addr.nl_pid = getpid();
53     src_addr.nl_groups = PRIVATE_EVENT_GROUP ;
54
/*****
*****struct sockaddr_nl {
*****    __kernel_sa_family_t    nl_family; /* AF_NETLINK */
*****    unsigned short    nl_pad;        /* zero */
*****    __u32        nl_pid;        /* port ID */
*****    __u32        nl_groups; /* multicast groups mask */
*****};
*****/



54     bind(sock_fd, (struct sockaddr*)&src_addr, sizeof(src_addr)); //将 netlink 套接字和 netlink 地址绑定。
55     memset(&dest_addr, 0, sizeof(dest_addr));
56
57     dest_addr.nl_family = AF_NETLINK;
58     dest_addr.nl_pid = 0; /* For Linux Kernel */
59     dest_addr.nl_groups = PRIVATE_EVENT_GROUP;
60
61     nlh=(struct nlmsghdr *)malloc(NLMSG_SPACE(MAX_PAYLOAD));
62     /* 参看图 14.1: */
63     nlh->nlmsg_len = NLMSG_SPACE(MAX_PAYLOAD);
64     nlh->nlmsg_pid = getpid(); /* self pid */
65     nlh->nlmsg_flags = 0;
66
67     for (i = 0;i < TEST_CNT;i++){
68         /* Fill in the netlink message payload */

//将 netlink 信息和 msg 关联起来。
70     msg = NLMSG_DATA(nlh); //消息头的首部存放 netlink 的头，见图 14.1。
71     msg->event = 0;
72     msg->sub_event = (i%2) + 1;
73     msg->len = sizeof(SEND_TEST_DATA);//Hello Word 字符串在 payload 里了，见图 14.1。
74     strcpy(msg->data, SEND_TEST_DATA);
75
76     //printf("test %d time: %s\n",i,(msg->sub_event == LOOP_UNICAST) ? "UNICAST" : "BROADCAST");
77     iov.iov_base = (void *)nlh; //这边管理 netlink 头，也是关联 msg，在 70 行，netlink 和 msg 的关系就确定了。
78     iov.iov_len = nlh->nlmsg_len;
79     message.msg_name = (void *)&dest_addr;
80     message.msg_namelen = sizeof(dest_addr);
81     message.msg iov = &iov;

```

```

82     message.msg iovlen = 1;
83
84     sendmsg(sock_fd, &message, 0); //发送消息给内核
85     /* Read message from kernel */
86
87     memset(nlh, 0, NLMSG_SPACE(MAX_PAYLOAD));
88     recvmsg(sock_fd, &message, 0); //从内核接收消息
89     msg = NLMSG_DATA(nlh);
90     hello_info = (struct hello_info *)&msg->data;
91     if (msg->event == 3)
92     {
93         printf("recv event %d sub_event %d alarm_info.\n", msg->event, msg->sub_event);
94     }
95     close(sock_fd);
96     return 0;
97 }
98
99
100 }
```

Netlink 消息类型

Include/uapi/linux/netlink.h

```

8 #define NETLINK_ROUTE      0 /* Routing/device hook          */
9 #define NETLINK_UNUSED      1 /* Unused number                */
10 #define NETLINK_USERSOCK    2 /* Reserved for user mode socket protocols */
11 #define NETLINK_FIREWALL    3 /* Unused number, formerly ip_queue */
12 #define NETLINK_SOCK_DIAG   4 /* socket monitoring           */
13 #define NETLINK_NFLOG        5 /* netfilter/iptables ULOG */
14 #define NETLINK_XFRM        6 /* ip security */
15 #define NETLINK_SELINUX     7 /* SELinux event notifications */
16 #define NETLINK_ISCSI       8 /* Open-iSCSI */
17 #define NETLINK_AUDIT      9 /* auditing */
18 #define NETLINK_FIB_LOOKUP  10
19 #define NETLINK_CONNECTOR   11
20 #define NETLINK_NETFILTER   12 /* netfilter subsystem */
21 #define NETLINK_IP6_FW      13
22 #define NETLINK_DNRTMSG     14 /* DECnet routing messages */
23 #define NETLINK_KOBJECT_UEVENT 15 /* Kernel messages to userspace */
24 #define NETLINK_GENERIC     16
25 /* leave room for NETLINK_DM (DM Events) */
26 #define NETLINK_SCSITRANSPORT 18 /* SCSI Transports */
27 #define NETLINK_ECRYPTFS    19
28 #define NETLINK_RDMA        20
29 #define NETLINK_CRYPTO      21 /* Crypto layer */
30
31 #define NETLINK_INET_DIAG   NETLINK_SOCK_DIAG
32
33 #define MAX_LINKS 32
```

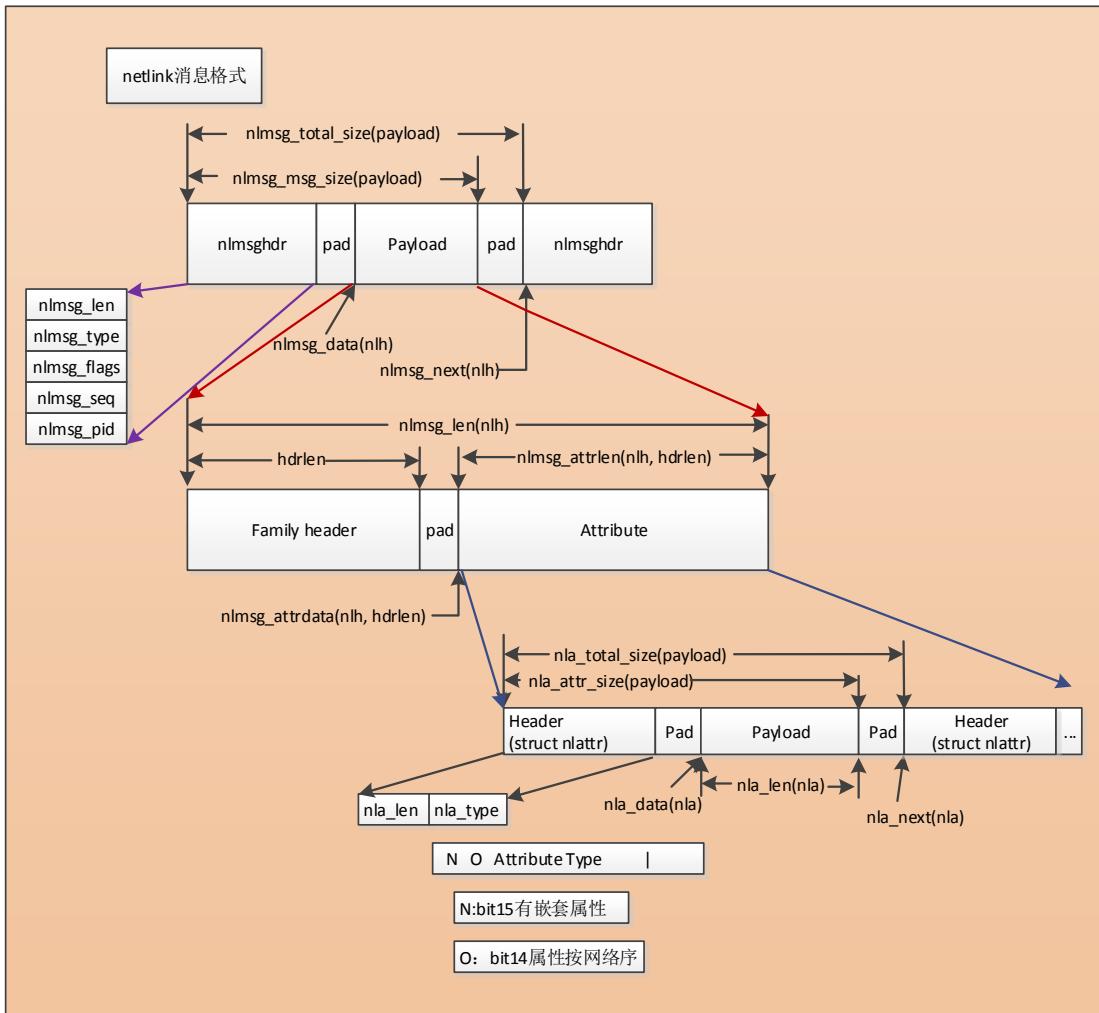


图 14.1 netlink 消息格式

14.2 netlink 用户空间 API

在第六章提到套接字创建的系统调用时，提到实际的套接字创建是由具体的协议族的套接字创建函数来完成的，其调用形如 `err = pf->create(net, sock, protocol, kern);`；在 `af_netlink.c` 文件中 `netlink` 协议族的创建 `netlink` 套接字的注册的结构体如下。

```
static const struct net_proto_family netlink_family_ops = {
    .family = PF_NETLINK,
    .create = netlink_create,
    .owner   = THIS_MODULE, /* for consistency 8 */
};
```

其创建 `netlink` 套接字的过程和 `inet` 套接字是一样的。和 `inet` 协议很相似，`netlink` 实现相关的主要代码在 `af_netlink.c` (`inet` 也有一个 `af_inet.c`) 文件。

14.3 netlink 内核空间 API

14.2 节的内容是针对用户空间的，这本小节则是针对内核而言的。Netlink 内核创建 API 位于 `include/linux/netlink.h`。在 14.1 节的 `netlink` 应用程序调用 14.2 节的 `netlink` 套接字创建 API 创建 `netlink` 套接字，并发送了一个 `netlink` 消息，在内核侧有对应的 `netlink` 套接字接收

应用程序发送的消息。内核侧创建 netlink 消息方法和应用程序调用的接口并不一样。接收应用程序发送的消息的内核侧驱动程序 netlink 创建的 netlink 代码片段可以看出。312~319 行可以看到 netlink 套接字创建接口的参数随着内核版本的升级发生了一些变化。本文基于 3.10 内核，所以创建的 API 是 318 行代码中的 netlink_kernel_create。

```
304 static int event_notify_init(void)
305 {
306     struct sock *nlsock = NULL;
307     int ret;
308     struct netlink_kernel_cfg cfg = {
309         .input = event_notify_receive_skb,
310     };
311
312 #if (LINUX_VERSION_CODE < KERNEL_VERSION(2, 6, 24))
313     nlsock = netlink_kernel_create(EVENT_NOTIFY, 0, event_notify_receive_skb, THIS_MODULE);
314 #elif (LINUX_VERSION_CODE < KERNEL_VERSION(3, 6, 0))
315     nlsock = netlink_kernel_create(&init_net, EVENT_NOTIFY, 0,
316                                     event_notify_receive_skb, NULL, THIS_MODULE);
317 #else
318     nlsock = netlink_kernel_create(&init_net, EVENT_NOTIFY, &cfg);
319 #endif
320
321     if (nlsock) {
322         en_nlsock = nlsock;
323         ret = init_events(events_group);
324         if (ret) {
325             printk(KERN_ERR "some events init fail\n");
326         }
327         return 0;
328     }else{
329         printk(KERN_ERR "create netlink %d error\n",EVENT_NOTIFY);
330         return -1;
331     }
332 }
```

Netlink 内核侧的创建函数实际是对 __netlink_kernel_create 的封装。

```
af_netlink.c
54 static inline struct sock *
55 netlink_kernel_create(struct net *net, int unit, struct netlink_kernel_cfg *cfg)
56 {
57     return __netlink_kernel_create(net, unit, THIS_MODULE, cfg);
58 }
```

其调用的函数位于 af_netlink.c，*net 指向所在的网络命名空间，unit 是 netlink 协议类型，module 是模块所有者信息，cfg 存放的是 netlink 内核配置参数，配置参数中的 input 成员用于处理接收到的消息，对于上面的驱动程序只初始化了 cfg 的 input 函数指针。该回调函数在应用程序调用 sendmsg() 发送消息时被调用。

```
2229 struct sock *
2230 __netlink_kernel_create(struct net *net, int unit, struct module *module,
2231                         struct netlink_kernel_cfg *cfg)
2232 {
2233     struct socket *sock;
2234     struct sock *sk;
2235     struct netlink_sock *nlk;
2236     struct listeners *listeners = NULL;
2237     struct mutex *cb_mutex = cfg ? cfg->cb_mutex : NULL;
2238     unsigned int groups;
2239 }
```

```

2240     BUG_ON(!nl_table);
2241
2242     if (unit < 0 || unit >= MAX_LINKS)
2243         return NULL;
//为 sock 申请套接字存储空间，并将套接字类型设置为 SOCK_DGRAM。
2244     if (sock_create_lite(PF_NETLINK, SOCK_DGRAM, unit, &sock))
2245         return NULL;
//按 netlink 机制需要初始化套接字相应的成员。注意是在初始网络命名空间中完成的。
2246     if (__netlink_create(&init_net, sock, cb_mutex, unit) < 0)
2247         goto out_sock_release_nosk;
//更新套接字命名空间
2248     sk = sock->sk;
2249     sk_change_net(sk, net);
2250
2251     if (!cfg || cfg->groups < 32)
2252         groups = 32;
2253     else
2254         groups = cfg->groups;
2255
2256     listeners = kzalloc(sizeof(*listeners) + NLGRPSZ(groups), GFP_KERNEL);
2257     if (!listeners)
2258         goto out_sock_release;
2259
2260     sk->sk_data_ready = netlink_data_ready;
2261     if (cfg && cfg->input)
2262         nlk_sk(sk)->netlink_rcv = cfg->input; //设置 netlink 消息接收处理函数。
2263
2264     if (netlink_insert(sk, net, 0))
2265         goto out_sock_release;
2266
2267     nlk = nlk_sk(sk);
2268     nlk->flags |= NETLINK_KERNEL_SOCKET;
2269
2270     netlink_table_grab(); //将进程放到 nl_table_wait 等待链表上，并调度其它进程。
2271     if (!nl_table[unit].registered) {
2272         nl_table[unit].groups = groups;
2273         rcu_assign_pointer(nl_table[unit].listeners, listeners);
2274         nl_table[unit].cb_mutex = cb_mutex;
2275         nl_table[unit].module = module;
2276         if (cfg) {
2277             nl_table[unit].bind = cfg->bind;
2278             nl_table[unit].flags = cfg->flags;
2279         }
2280         nl_table[unit].registered = 1;
2281     } else {
2282         kfree(listeners);
2283         nl_table[unit].registered++;
2284     }
2285     netlink_table_ungrab();
2286     return sk;
2287
2288
2289 out_sock_release:
2290     kfree(listeners);
2291     netlink_kernel_release(sk);
2292     return NULL;
2293
2294
2295 out_sock_release_nosk:
2296     sock_release(sock); //内核关闭 netlink 套接字 API。

```

```
2304     return NULL;  
2305 }
```

Netlink 内核发送消息的内核空间 API 是：

```
netlink_unicast  
netlink_broadcast
```

发送消息的代码片段如下：

```
if (pid) {  
    /* unicast */  
    NETLINK_CB(skb).portid = pid;  
    ret = netlink_unicast(nlsock, skb, pid, MSG_DONTWAIT); //单播发送法  
} else {  
    /* broadcast */  
    NETLINK_CB(skb).dst_group = group;  
    ret = netlink_broadcast(nlsock, skb, 0, group, 0); //广播发送法
```

第十五章 提升网络性能技术

本章的这些特性在于提高网络性能，这些技术中的一些需要 NICs 的支持，一些特性是软件实现的。这些特性在服务器上使用较为流行，但是对于嵌入式领域，尤其是视频等网络等带宽需求较大的应用场景，这些特性也可能被使用。

以太网数据包长是 1500 个字节，如果要传输大量的数据，TCP 层就需要将这些数据分成若干小片 MSS (Maximum Segment Size)，然后再将这些小片发送出去。

如果将一个大数据包（通常是 MSS 整数倍）送给网卡，那么网卡就需要完成分片工作，将数据分片成 1460 字节+40 字节的包头信息，传送给 PHY 芯片。这就意味着网卡需要支持这类的特性才能这么使用。网卡特性的跟踪相对而言比较简单，在网卡注册的时候，就将网卡支持的一些特性放在了一个 flag 中了。

这些特性被称为 offload 特性，其寓意就是将 CPU 从繁重的工作中 offload 了，就意味着 CPU 可以有更多的时间让给操作系统其它功能部分了。

rx-checksumming TCP 接收的校验和由网卡验证。

tx-checksumming: 网卡完成发送 TCP 头校验和的计算

scatter-gather: 分散/聚集 IO，网卡支持该特性，可以减少内存的拷贝。

(TSO)tcp-segmentation-offload: TCP、DCCP 数据包分片(依赖 tx-checksumming 和 scatter-gather)

(UFO)udp-fragmentation-offload: 支持 UDP 分片，ip 层不需要对最大 64K 的数据进行分片。

(GSO)generic-segmentation-offload: 支持 tcp、UDP 和 ipv6 分片，由软件模拟来完成，于硬件无关。

(GRO)generic-receive-offload: 支持 tcp、UDP 和 ipv6 分片，由软件模拟来完成，于硬件无关。

(LRO)large-receive-offload: 小包聚合成大包传递给协议栈，减小上层开销。

```
root@u:/home/ge/ethtool-2.6.36# ./ethtool -k eth0
Offload parameters for eth0:
rx-checksumming: off
tx-checksumming: off
scatter-gather: off
tcp-segmentation-offload: off
udp-fragmentation-offload: off
generic-segmentation-offload: off
generic-receive-offload: on
large-receive-offload: off
rx-vlan-offload: off
tx-vlan-offload: off
ntuple-filters: off
receive-hashing: off
root@u:/home/ge/ethtool-2.6.36#
```

```
Offload parameters for eth0:
rx-checksumming: off
tx-checksumming: on
scatter-gather: on
tcp-segmentation-offload: on
udp-fragmentation-offload: off
generic-segmentation-offload: on
generic-receive-offload: on
large-receive-offload: off
rx-vlan-offload: on
tx-vlan-offload: on
ntuple-filters: off
receive-hashing: off
```

15.1 TSO/GSO

在发送数据包时，提到 `tcp_write_xmit` 函数，该函数会调用 `tcp_init_tso_segs`，这个函数初始化一个 `skb` 的 `tso` 状态。

```
1429 static int tcp_init_tso_segs(const struct sock *sk, struct sk_buff *skb,
1430                                unsigned int mss_now)
1431 {
1432     int tso_segs = tcp_skb_pcount(skb); // 获得 skb 被分成了多少个 tso 段，准确来说是 tso 特性支持了多个段。
// 当 1、当前分段成员值为 0
// 2、skb 的 mss 和当前的 mss 值不相等，则需要初始化 skb 的 TSO 相关成员
1434     if (!tso_segs || (tso_segs > 1 && tcp_skb_mss(skb) != mss_now)) {
1435         tcp_set_skb_tso_segs(sk, skb, mss_now); // 设置 skb 的 TSO 成员。见下
1436         tso_segs = tcp_skb_pcount(skb); // 获得准确的 tso 段个数。
1437     }
1438     return tso_segs;
1439 }
//
977 static void tcp_set_skb_tso_segs(const struct sock *sk, struct sk_buff *skb,
978                                   unsigned int mss_now)
979 {
// 1、数据长度小于当前 mss; 2、网卡不支持 gso; 3、ip_summed 原本存放的是校验和，但如果等于 CHECKSUM_NONE，// 则表示网卡没能完成校验和的计算，网卡不支持硬件计算校验和就会这样。以上三种情况没有必要分片。
980     if (skb->len <= mss_now || !sk_can_gso(sk) ||
981         skb->ip_summed == CHECKSUM_NONE) // CHECKSUM_NONE,--device failed to checksum this packet.
982         /* Avoid the costly divide in the normal
983          * non-TSO case.
984          */
985         skb_shinfo(skb)->gso_segs = 1;
986         skb_shinfo(skb)->gso_size = 0;
987         skb_shinfo(skb)->gso_type = 0;
988     } else {
989         skb_shinfo(skb)->gso_segs = DIV_ROUND_UP(skb->len, mss_now); // 倍数关系
990         skb_shinfo(skb)->gso_size = mss_now;
// gso 类型，比如 SKB_GSO_TCPV4, SKB_GSO_UDP...
991         skb_shinfo(skb)->gso_type = sk->sk_gso_type;
992     }
993 }
```

拥塞窗口

```
// 返回 true，则需要增加拥塞窗口，否则不需要增加拥塞窗口。
284 bool tcp_is_cwnd_limited(const struct sock *sk, u32 in_flight)
285 {
286     const struct tcp_sock *tp = tcp_sk(sk);
287     u32 left;
// 已发送还未 ack 数据包等于发送拥塞窗口长度，
288     if (in_flight >= tp->snd_cwnd)
289         return true;
// 还可以发送的数据量。
290     left = tp->snd_cwnd - in_flight;
//
291     if (sk_can_gso(sk) &&
292         left * sysctl_tcp_tso_win_divisor < tp->snd_cwnd &&
293         left * tp->mss_cache < sk->sk_gso_max_size &&
294         left < sk->sk_gso_max_segs)
295         return true;
```

```

298     return left <= tcp_max_tso_deferred_mss(tp);
299 }

```

为了最小化 TSO 分片，延迟发送数据包。TSO Nagle 算法。返回值是 1，表示可以延迟发送，返回 0，则立即发送。

```

1596 static bool tcp_tso_should_defer(struct sock *sk, struct sk_buff *skb)
1597 {
1598     struct tcp_sock *tp = tcp_sk(sk);
1599     const struct inet_connection_sock *icsk = inet_csk(sk);
1600     u32 send_win, cong_win, limit, in_flight;
1601     int win_divisor;
// TCPHDR_FIN 意味着是 close() 调用触发的关闭套接字连接，则立即发送。
1603     if (TCP_SKB_CB(skb)->tcp_flags & TCPHDR_FIN)
1604         goto send_now;
// 正常状态，则立即发送。
1606     if (icsk->icsk_ca_state != TCP_CA_Open)
1607         goto send_now;
1608
// 上一个 skb 被延迟了，且超过一个时钟，则立即发送。
1610     if (tp->tso_deferred &&
1611         (((u32)jiffies << 1) >> 1) - (tp->tso_deferred >> 1) > 1)
1612         goto send_now;
// 发送但没收到 ack 包，包括重传的数据包。
1614     in_flight = tcp_packets_in_flight(tp);
// 通告窗口剩余量
1618     send_win = tcp_wnd_end(tp) - TCP_SKB_CB(skb)->seq;
// 拥塞窗口剩余量。
1621     cong_win = (tp->snd_cwnd - in_flight) * tp->mss_cache;
// 而这较小值，作为发送门限。
1623     limit = min(send_win, cong_win);
1624
1625     /* If a full-sized TSO skb can be sent, do it. */
1626     if (limit >= min_t(unsigned int, sk->sk_gso_max_size,
1627                         sk->sk_gso_max_segs * tp->mss_cache))
1628         goto send_now;
1629
1630     /* Middle in queue won't get any more data, full sendable already? */
1631     if ((skb != tcp_write_queue_tail(sk)) && (limit >= skb->len))
1632         goto send_now;
1633
1634     win_divisor = ACCESS_ONCE(sysctl_tcp_tso_win_divisor);
1635     if (win_divisor) {
// 一个 RTT 内能够发送的最大字节数。
1636         u32 chunk = min(tp->snd_wnd, tp->snd_cwnd * tp->mss_cache);
1637
1638         /* If at least some fraction of a window is available,
1639          * just use it.
1640          */
1641         chunk /= win_divisor; // 单个 TSO 占用的发送量。
1642         if (limit >= chunk)
1643             goto send_now;
1644     } else {
1645         /* Different approach, try not to defer past a single
1646          * ACK. Receiver should ACK every other full sized
1647          * frame, so if we have space for more than 3 frames
1648          * then send now.

```

```

1649         */
1650     if (limit > tcp_max_tso_deferred_mss(tp) * tp->mss_cache)
1651         goto send_now;
1652     }
1653
1654     /* Ok, it looks like it is advisable to defer.
1655      * Do not rearm the timer if already set to not break TCP ACK clocking.
1656      */
1657     if (!tp->tso_deferred)
1658         tp->tso_deferred = 1 | (jiffies << 1); //记录延迟时间戳
1659
1660     return true;
1661
1662 send_now:
1663     tp->tso_deferred = 0;
1664     return false;
1665 }

```

15.2 LRO/GRO

15.1 节所述特性用在 skb 的发送流程上, 而这一节的特性用在了 skb 的接收流程上, LRO (Large Receive Offload) 针对 TCP 数据包计数, 将多个数据包聚合到一个大 skb 发送给上层协议栈, 这样提高网络效率, GRO 是更通用的版本。

LRO 管理结构体 `Include/linux/inet_lro.h`

```

74 struct net_lro_mgr {
75     struct net_device *dev; //关联 LRO 功能和网络设备
76     struct net_lro_stats stats; //LRO 统计信息
77
78     /* LRO features */
79     unsigned long features; //如下#define 定义的特性
80 #define LRO_F_NAPI           //通过 NAPI 方式传递 Packet
81 #define LRO_F_EXTRACT_VLAN_ID 2 // VLAN 和 8021Q
82     u32 ip_summed; //在启用分片模式时当生成的 SKB 没有被添加到分片管理链表 ip_summed 会被置位
83     u32 ip_summed_aggr; /* 聚合成大的 SKBs 校验 CHECKSUM_UNNECESSARY 或者 CHECKSUM_NONE */
84
85     int max_desc; //LRO 描述符最多个数, 每一个 LRO 描述符对应一路 TCP 流。
86     int max_aggr; //聚合成一个 SKB 的最多包数目
87
88     int frag_align_pad; /* 当使用分片时为 L3 在生成的 SKB 对其头而预留的填充
89
90     struct net_lro_desc *lro_arr; /* Array of LRO descriptors */
// get_skb_header: returns tcp and ip header for packet in SKB
91     int (*get_skb_header)(struct sk_buff *skb, void **ip_hdr,
92                         void **tcpudp_hdr, u64 *hdr_flags, void *priv);
93
94     /* hdr_flags: */
95 #define LRO_IPV4 1 /* ip_hdr is IPv4 header */
96 #define LRO_TCP 2 /* tcpudp_hdr is TCP header */
97
98     /*
99      * get_frag_header: returns mac, tcp and ip header for packet in SKB
100     *
101     * @hdr_flags: Indicate what kind of LRO has to be done
102     *             (IPv4/IPv6/TCP/UDP)
103     */
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118

```

```

119     int (*get_frag_header)(struct skb_frag_struct *frag, void **mac_hdr,
120                           void **ip_hdr, void **tcpudp_hdr, u64 *hdr_flags,
121                           void *priv);
122 };

```

LRO 属于 NICs 特性，在具体 NICs 驱动程序接收数据包时会调用 `lro_receive_skb` 函数以 `lro` 方式接收数据包。

```

496 void lro_receive_skb(struct net_lro_mgr *lro_mngr,
497                       struct sk_buff *skb,
498                       void *priv)
499 {
500     if (__lro_proc_skb(lro_mngr, skb, priv)) {
501         if (lro_mngr->features & LRO_F_NAPI)
502             netif_receive_skb(skb);
503         else
504             netif_rx(skb);
505     }
506 }

```

通常数据包会聚合到最大值，但是对延迟要求苛刻可能立即发送，这会调用 `lro_flush`

GRO 是 LRO 的加强版，其使用范围比 LRO 要广。然后 `_napi_gro_receive` 会调用 `dev_gro_receive`，`dev_gro_receive` 会先调用 `ptype->gso_receive`，一般而言就是 ip 协议对应的 `inet_gso_receive`

`inet_gro_receive` 主要做如下事情：

首先拿到 ip 包头，然后对包头做 check，如果 check passed 则开始遍历 `napi_struct->gro_list`，根据 `ip saddr, daddr, tos, protocol` 等来和那些之前在二层有可能是同一 flow 的 `skb` 进行判断，如果不一致就把 `same_flow` 置 0，当然光是 `slow_flow` 并不能就此开始 merge，还要进行 flush 的判断，任何 flush 判断不过都会放弃 merge 而调用直接调用 `skb_gro_flush` 函数交到协议栈上层去处理

ip 层结束之后，会继续 tcp 层的 `gro_receive`，调用 `tcp_gro_receive`，其核心也是遍历 `napi_struct->gro_list`，基于 source addr 判断是否是 `same_flow`，对是否需要 flush 做计算，这里提一下关于 ack 一致的要求，ack 一致说明是同一个 tcp payload 被 tso/gso 分段之后的结果，所以是必需条件

如果 tcp 也认为不需要 flush，那么会进到 `skb_gro_receive` 中，这个函数就是用来合并的，第一个参数是 `gro_list` 里的那个 `skb`，第二个是新来的 `skb`，这里不多说了，我推荐的博客文章里讲的很清楚了。其实就分两种情况，如果是 scatter-gather 的 `skb` 包，则把新 `skb` 里的 `frags` 的数据放到 `gro_list` 的 `skb` 对应的 `frags` 数据后面；否则 `skb` 数据都在 `skb` 的线性地址中，这样直接 alloc 一个新的 `skb`，把新 `skb` 挂到 `frag_list` 里面，最后放到原来 `gro_list` 的位置上；如果 `gro_list` 的 `skb` 已经有了 `frag_list`，那么就直接挂进去好了

现在返回到 `dev_gro_receive` 中了，这时如果需要 flush 或者 `same_flow` 为 0，说明需要传给上层协议栈了，此时调用 `napi_gro_complete`

走到最后一种情况即这个 `skb` 是个新的 flow，那么就加到 `gro_list` 的链表中

最后提下，所谓 flush 是指把现有 `gro_list` 中的 `skb` flush 到上层协议，千万别搞反了

```

1539 struct packet_type {
1540     __be16          type; /* This is really htons(ether_type). */
1541     struct net_device *dev; /* NULL is wildcarded here */
1542     int           (*func) (struct sk_buff *,

```

```

1543             struct net_device *,
1544             struct packet_type *,
1545             struct net_device *);
1546     bool (*id_match)(struct packet_type *ptype,
1547                      struct sock *sk);
1548     void *af_packet_priv;
1549     struct list_head list;
1550 };
1551
1552 struct offload_callbacks {
1553     struct sk_buff *(*gso_segment)(struct sk_buff *skb,
1554                                    netdev_features_t features);
1555     int (*gso_send_check)(struct sk_buff *skb);
1556     struct sk_buff **(*gro_receive)(struct sk_buff **head,
1557                                    struct sk_buff *skb);
1558     int (*gro_complete)(struct sk_buff *skb);
1559 };
1560
1561 struct packet_offload {
1562     __be16 type; /* This is really htons(ether_type). */
1563     struct offload_callbacks callbacks;
1564     struct list_head list;
1565 };
Include/linux/netdevice.h
306 struct napi_struct {
307     /* The poll_list must only be managed by the entity which
308      * changes the state of the NAPI_STATE_SCHED bit. This means
309      * whoever atomically sets that bit can add this napi_struct
310      * to the per-cpu poll_list, and whoever clears that bit
311      * can remove from the list right before clearing the bit.
312      */
313     struct list_head poll_list;
314
315     unsigned long state;
316     int weight;
317     unsigned int gro_count;
318     int (*poll)(struct napi_struct *, int);
319 #ifdef CONFIG_NETPOLL
320     spinlock_t poll_lock;
321     int poll_owner;
322 #endif
323     struct net_device *dev;
324     struct sk_buff *gro_list;
325     struct sk_buff *skb;
326     struct list_head dev_list;
327 };

```

Net/core/dev.c

```

3829 static gro_result_t napi_skb_finish(gro_result_t ret, struct sk_buff *skb)
3830 {
3831     switch (ret) {
3832         case GRO_NORMAL:
3833             if (netif_receive_skb(skb))
3834                 ret = GRO_DROP;
3835             break;
3836
3837         case GRO_DROP:
3838             kfree_skb(skb);

```

```

3839         break;
3840
3841     case GRO_MERGED_FREE:
3842         if (NAPI_GRO_CB(skb)->free == NAPI_GRO_FREE_STOLEN_HEAD)
3843             kmem_cache_free(skbuff_head_cache, skb);
3844         else
3845             __kfree_skb(skb);
3846         break;
3847
3848     case GRO_HELD:
3849     case GRO_MERGED:
3850         break;
3851     }
3852
3853     return ret;
3854 }
3855
3856 static void skb_gro_reset_offset(struct sk_buff *skb)
3857 {
3858     const struct skb_shared_info *pinfo = skb_shinfo(skb);
3859     const skb_frag_t *frag0 = &pinfo-> frags[0];
3860
3861     NAPI_GRO_CB(skb)->data_offset = 0;
3862     NAPI_GRO_CB(skb)->frag0 = NULL;
3863     NAPI_GRO_CB(skb)->frag0_len = 0;
3864
3865     if (skb->mac_header == skb->tail &&
3866         pinfo->nr_frags &&
3867         !PageHighMem(skb_frag_page(frag0))) {
3868         NAPI_GRO_CB(skb)->frag0 = skb_frag_address(frag0);
3869         NAPI_GRO_CB(skb)->frag0_len = skb_frag_size(frag0);
3870     }
3871 }
3872
3873 gro_result_t napi_gro_receive(struct napi_struct *napi, struct sk_buff *skb)
3874 {
3875     skb_gro_reset_offset(skb);
3876
3877     return napi_skb_finish(dev_gro_receive(napi, skb), skb);
3878 }

3736 static enum gro_result dev_gro_receive(struct napi_struct *napi, struct sk_buff *skb)
3737 {
3738     struct sk_buff **pp = NULL;
3739     struct packet_offload *ptype;
3740     __be16 type = skb->protocol;
3741     struct list_head *head = &offload_base;
3742     int same_flow;
3743     enum gro_result ret;
3744
3745     if (!(skb->dev->features & NETIF_F_GRO) || netpoll_rx_on(skb))
3746         goto normal;
3747
3748     if (skb_is_gso(skb) || skb_has_frag_list(skb))
3749         goto normal;
3750
3751     gro_list_prepare(napi, skb);
3752

```

```

3753     rcu_read_lock();
3754     list_for_each_entry_rcu(ptype, head, list) {
3755         if (ptype->type != type || !ptype->callbacks.gro_receive)
3756             continue;
3757
3758         skb_set_network_header(skb, skb_gro_offset(skb));
3759         skb_reset_mac_len(skb);
3760         NAPI_GRO_CB(skb)->same_flow = 0;
3761         NAPI_GRO_CB(skb)->flush = 0;
3762         NAPI_GRO_CB(skb)->free = 0;
3763
3764         pp = ptype->callbacks.gro_receive(&napi->gro_list, skb);
3765         break;
3766     }
3767     rcu_read_unlock();
3768
3769     if (&ptype->list == head)
3770         goto normal;
3771
3772     same_flow = NAPI_GRO_CB(skb)->same_flow;
3773     ret = NAPI_GRO_CB(skb)->free ? GRO_MERGED_FREE : GRO_MERGED;
3774
3775     if (pp) {
3776         struct sk_buff *nskb = *pp;
3777
3778         *pp = nskb->next;
3779         nskb->next = NULL;
3780         napi_gro_complete(nskb);
3781         napi->gro_count--;
3782     }
3783
3784     if (same_flow)
3785         goto ok;
3786
3787     if (NAPI_GRO_CB(skb)->flush || napi->gro_count >= MAX_GRO_SKBS)
3788         goto normal;
3789
3790     napi->gro_count++;
3791     NAPI_GRO_CB(skb)->count = 1;
3792     NAPI_GRO_CB(skb)->age = jiffies;
3793     skb_shinfo(skb)->gso_size = skb_gro_len(skb);
3794     skb->next = napi->gro_list;
3795     napi->gro_list = skb;
3796     ret = GRO_HELD;
3797
3798 pull:
3799     if (skb_headlen(skb) < skb_gro_offset(skb)) {
3800         int grow = skb_gro_offset(skb) - skb_headlen(skb);
3801
3802         BUG_ON(skb->end - skb->tail < grow);
3803
3804         memcpy(skb_tail_pointer(skb), NAPI_GRO_CB(skb)->frag0, grow);
3805
3806         skb->tail += grow;
3807         skb->data_len -= grow;
3808
3809         skb_shinfo(skb)->frags[0].page_offset += grow;
3810         skb_frag_size_sub(&skb_shinfo(skb)->frags[0], grow);

```

```

3811
3812     if (unlikely(!skb_frag_size(&skb_shinfo(skb)->frags[0]))) {
3813         skb_frag_unref(skb, 0);
3814         memmove(skb_shinfo(skb)->frags,
3815                 skb_shinfo(skb)->frags + 1,
3816                 --skb_shinfo(skb)->nr_frags * sizeof(skb_frag_t));
3817     }
3818 }
3819
3820 ok:
3821     return ret;
3822
3823 normal:
3824     ret = GRO_NORMAL;
3825     goto pull;
3826 }

```

15.3 RSS (Receive Side Scaling) 队列:

网卡多队列技术，硬件支持这一特性，将受到的数据包发往不同的队列，已在不同的CPU上处理。RSS流程如图15.2，NICs在接收到串行数据包后，根据数据包的L3层和L4层头信息，硬件计算哈希值，不同的哈希值对应于不同的队列，即将数据包按哈希值分配到不同的队列上，不同的队列有不同的CPU处理，这样就实现了负载均衡。这一特性在drivers/net/ethernet/XXX等各种网卡驱动中设置网卡RSS控制寄存器来实现。目前内核支持的队列个数取8和CPU个数中较小的那个值。每个队列都有一个IRQ号，对应中断的服务CPU号（一个或多个）使用affinity设置，见Documentation/IRQ-affinity.txt。

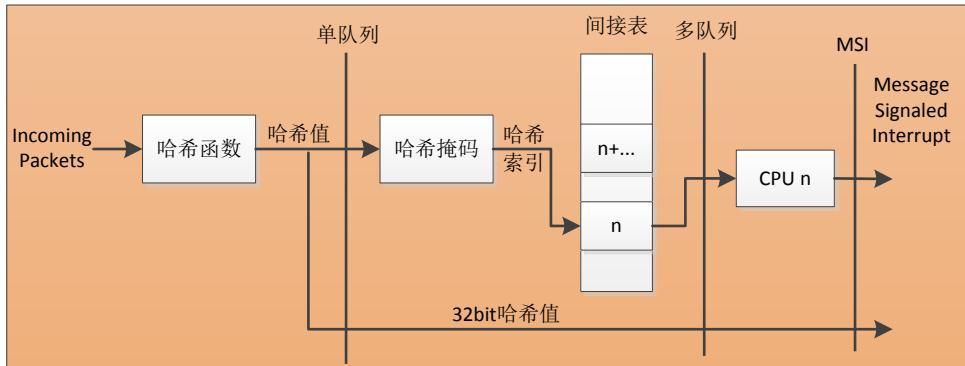


图 15.2 RSS 队列处理流程

15.4 RPS (Receive Packet Steering) 队列:

RPS (Receive Packet Steering) 源于谷歌 patch，软件方法实现 RSS。将数据包的源IP和目的IP、源端口和目的端口计算得到一个哈希值，根据该哈希值决定了软中断执行的核号，这种方法来均衡软中断在各个核的分布，以其达到负载均衡的目的。

```

sysctl -w net.core.rps_sock_flow_entries=
/sys/class/net/eth*/queues/rx-*/*rps_flow_cnt
/sys/class/net/eth*/queues/rx-*/*rps_cpus

```

两个重要的函数

该结构体包含一个RX队列。Include/linux/netdevice.h

```

struct rps_map {
    //零长数组的长度
    unsigned int len;
    struct rcu_head rcu;
    //能够处理 RPS 数据包的 cpu ID 数组。
    u16 cpus[0];
};

struct netdev_rx_queue {
    //保存一个 RPS 映射，映射的对象是能够处理这个队列的 CPU 核数组。
    struct rps_map __rcu *rps_map;
    //流控映射表
    struct rps_dev_flow_table __rcu *rps_flow_table;
    struct kobject kobj;
    struct net_device *dev;
} __cacheline_aligned_in_smp;

get_rps_cpu 由 netif_receive_skb (NAPI) 和 netif_rx 函数调用，为一个给定的 skb，从
RPS 接收队列映射关系中返回处理该数据包的 CPU ID。哈希值计算可以网卡计算。

Net/core/dev.c

2939 static int get_rps_cpu(struct net_device *dev, struct sk_buff *skb,
2940                         struct rps_dev_flow **rflowp)
2941 {
2942     struct netdev_rx_queue *rxqueue;
2943     struct rps_map *map;
2944     struct rps_dev_flow_table *flow_table;
2945     struct rps_sock_flow_table *sock_flow_table; //包含 skb 期望在那个核被处理信息。
2946     int cpu = -1;
2947     u16 tcpu;
    //获得网卡对应的接收队列
2949     if (skb_rx_queue_recorded(skb)) { //判断是否支持多队列，一个队列对应于一个 netdev_rx_queue
2950         u16 index = skb_get_rx_queue(skb); //获得 skb 所在队列的索引
2951         if (unlikely(index >= dev->real_num_rx_queues)) { //索引范围合法性检查
2952             WARN_ONCE(dev->real_num_rx_queues > 1,
2953                     "%s received packet on queue %u, but number "
2954                     "of RX queues is %u\n",
2955                     dev->name, index, dev->real_num_rx_queues);
2956             goto done;
2957         }
2958         rxqueue = dev->_rx + index; //队列首地址+队列索引，获得 skb 所在的队列。
2959     } else
2960         rxqueue = dev->_rx; //不支持多队列，直接返回即可。
    //rcu 变量引用方法，获得 rps_map 员。2962 行判断该成员是否为 NULL，如果不为空，那么可能存在能够处理该队列的 CPU
//ID 了
2962     map = rcu_dereference(rxqueue->rps_map);
2963     if (map) {
2964         if (map->len == 1 &&
2965             !rcu_access_pointer(rxqueue->rps_flow_table)) {//
2966             tcpu = map->cpus[0]; //获得 RPS 对应的 CPU ID。
2967             if (cpu_online(tcpu)) //判断该 CPU 是否被启动（多个 CPU 情况，不一定每个 CPU 核都被启用）
2968                 cpu = tcpu;
2969             goto done;
2970         }
2971     } else if (!rcu_access_pointer(rxqueue->rps_flow_table)) {
2972         goto done;
2973     }
    //获得网络头
2975     skb_reset_network_header(skb);
2976     if (!skb_get_rxhash(skb)) //获得计算得到的哈希值。

```

```

2977         goto done;
//2979~3016 流控处理, 见 RFS 一节
2979     flow_table = rCU_dereference(rxqueue->rps_flow_table);
2980     sock_flow_table = rCU_dereference(rps_sock_flow_table); //包含映射了应用程序处理的流控信息。
2981     if (flow_table && sock_flow_table) {
2982         u16 next_cpu;
2983         struct rps_dev_flow *rflow;
2984
2985         rflow = &flow_table->flows[skb->rxhash & flow_table->mask];
2986         tcpu = rflow->cpu; //流控处理期望所在的 CPU
2987
2988         next_cpu = sock_flow_table->ents[skb->rxhash &
2989             sock_flow_table->mask]; //流控处理期望在该 CPU 上被处理
2990
3002         if (unlikely(tcpu != next_cpu) && //两者不一致, 需要进一步判断是否需要进行切换 CPU。
3003             (tcpo == RPS_NO_CPU || !cpu_online(tcpo) ||
3004             ((int)(per_cpu(softnet_data, tcpo).input_queue_head -
3005                 rflow->last_qtail)) >= 0)) {
3006             tcpo = next_cpu;
3007             rflow = set_rps_cpu(dev, skb, rflow, next_cpu); //见 RFS 队列。
3008         }
3009
3010         if (tcpo != RPS_NO_CPU && cpu_online(tcpo)) { //RPS 设置的 CPU 启用, 并且这里流控处理核也是该 CPU
3011             *rflowp = rflow;
3012             cpu = tcpo;
3013             goto done;
3014         }
3015     }
3016
//不做流控处理会到达这里, 根据计算得到的哈希值, 获取对应的 CPU 核号。
3017     if (map) {
3018         tcpo = map->cpus[((u64) skb->rxhash * map->len) >> 32];
3019
3020         if (cpu_online(tcpo)) { //CPU 处于开启状态。
3021             cpu = tcpo;
3022             goto done;
3023         }
3024     }
3025
3026 done:
3027     return cpu;
3028 }

```

enqueue_to_backlog 用于将受到的 skb 放到对应 CPU 的 RPS 队列上。

```

net/core/dev.c
3106 static int enqueue_to_backlog(struct sk_buff *skb, int cpu,
3107                                 unsigned int *qtail)
3108 {
3109     struct softnet_data *sd;
3110     unsigned long flags;
3111
3112     sd = &per_cpu(softnet_data, cpu); //获得第二个参数 cpu 所指 ID 的 cpu 的 softnet_data 成员。
3113
3114     local_irq_save(flags);
3115
3116     rps_lock(sd);
//如果存储于接收队列 (input_pkt_queue) 的 skb 个数大于设备设定的最大值, 则到 3137 行, 更新丢弃计数并丢弃该数据包
3117     if (skb_queue_len(&sd->input_pkt_queue) <= netdev_max_backlog) {
3118         if (skb_queue_len(&sd->input_pkt_queue)) {

```

```

3119 enqueue:
3120         __skb_queue_tail(&sd->input_pkt_queue, skb); //将数据包放到队列尾部
3121         input_queue_tail_incr_save(sd, qtail); //更新计数
3122         rps_unlock(sd);
3123         local_irq_restore(flags);
3124         return NET_RX_SUCCESS;
3125     }
3126
//NAPI 方式，一个硬件中断会接收多个数据包。
3130     if (!__test_and_set_bit(NAPI_STATE_SCHED, &sd->backlog.state)) {
3131         if (!rps_ipi_queued(sd)) //检查该 sd 是否是其它 CPU 的，如果是其它 CPU，则____napi_schedule 被调用。
3132             ____napi_schedule(sd, &sd->backlog);
3133     }
3134     goto enqueue;
3135 }
3136
3137     sd->dropped++; //丢弃计数更新
3138     rps_unlock(sd);
3139
3140     local_irq_restore(flags);
3141
3142     atomic_long_inc(&skb->dev->rx_dropped);
3143     kfree_skb(skb); //丢弃该 skb
3144     return NET_RX_DROP;
3145 }

```

15.5 RFS(Receive Flow Steering), Accelerated Receive Flow Steering

不论是 RSS 还是 RPS 考虑都是将收到的数据包进行哈希散列，这没有考虑到应用程序所在的 CPU 号，加入一个等待数据包的网络进程正在 5 号 CPU 上等待数据包，而 RPS 计算得到的哈希值是 1，这就意味着数据包将由 1 号 CPU 接收。这会导致一个问题，那就是缓存命中失败。RFS 利用 RPS 机制将数据包插入指定 CPU 的 backlog 队列，并唤醒那个 CPU 执行。该函数在 RPS 一节调用到。全局数据流表(rps_sock_flow_table)的总数可以通过下面的参数来设置：

```

/proc/sys/net/core/rps_sock_flow_entries
每个队列的数据流表总数可以通过下面的参数来设置:
/sys/class/net/<dev>/queues/rx-<n>/rps_flow_cnt

```

```

2889 static struct rps_dev_flow *
2890 set_rps_cpu(struct net_device *dev, struct sk_buff *skb,
2891             struct rps_dev_flow *rflow, u16 next_cpu)
2892 {
2893     if (next_cpu != RPS_NO_CPU) {
2894 #ifdef CONFIG_RFS_ACCEL
2895         struct netdev_rx_queue *rxqueue;
2896         struct rps_dev_flow_table *flow_table;
2897         struct rps_dev_flow *old_rflow;
2898         u32 flow_id;
2899         u16 rxq_index;
2900         int rc;
2901
2902         /* Should we steer this flow to a different hardware queue? */

```

```

//skb_rx_queue_recorded 判断 skb 多队列映射支持是否置位, 没置位直接 goto out
// rx_cpu_rmap 设备是否支持 CPU 亲和度
//dev->features & NETIF_F_NTUPLE 判断 NIC 是否支持 n 元匹配过滤
//上述任何一个回答是否, 将执行 goto out;
2903     if (!skb_rx_queue_recorded(skb) || !dev->rx_cpu_rmap ||
2904         !(dev->features & NETIF_F_NTUPLE))
2905         goto out;
2906     rxq_index = cpu_rmap_lookup_index(dev->rx_cpu_rmap, next_cpu);
2907     if (rxq_index == skb_get_rx_queue(skb)) //接收队列在期望的 CPU 上, 直接返回就行了?
2908         goto out;
2909
2910     rxqueue = dev->rx + rxq_index; //获得期望 CPU 的接收队列索引
2911     flow_table = rcu_dereference(rxqueue->rps_flow_table);
2912     if (!flow_table)
2913         goto out;
2914     flow_id = skb->rxhash & flow_table->mask;
//ndo_rx_flow_steer 是 NIC 硬件支持 RFS 配置接口, CONFIG_RFS_ACCEL 配置选项用于表示是否硬件支持
2915     rc = dev->netdev_ops->ndo_rx_flow_steer(dev, skb,
2916                                              rxq_index, flow_id);
2917     if (rc < 0)
2918         goto out;
2919     old_rflow = rflow;
2920     rflow = &flow_table->flows[flow_id]; //将流放在期望 CPU 的流控结构里
2921     rflow->filter = rc; //接收过滤器标志
2922     if (old_rflow->filter == rflow->filter)
2923         old_rflow->filter = RPS_NO_FILTER;
2924     out:
2925 #endif
2926     rflow->last_qtail =
2927         per_cpu(softnet_data, next_cpu).input_queue_head;
2928 }
2929
2930     rflow->cpu = next_cpu;
2931     return rflow;
2932 }
2933

```

15.6 XPS (Transmit Packet Steering)

上面都是对接收数据包的处理, **XPS** 是对发送数据包的多队列处理。XPS 通过将发送数据包的 CPU 映射到一个队列的方法来实现。这类似于 RPS 不过是在发送端的“RPS”, RPS 基于数据包队列选择处理的 CPU, 而 XPS 则是基于发送数据包的 CPU 选择发送队列。

一个发送队列可以由一个或多个 CPU 支持, 配置参数位于:

`/sys/class/net/eth<n>/queues/tx-<n>/xps_cpus`

XPS 支持

```

struct xps_map {
    unsigned int len; //记录 queues[0]成员长度
    unsigned int alloc_len;
    struct rcu_head rcu;
    u16 queues[0]; //队列数组
};

#define XPS_MAP_SIZE(_num) (sizeof(struct xps_map) + (_num) * sizeof(u16)) //映射 num 个队列时, xps_map 的 size
#define XPS_MIN_MAP_ALLOC ((L1_CACHE_BYTES - sizeof(struct xps_map)) \
    / sizeof(u16))
//NICs 的 XPS 映射, 映射由 CPU 索引

```

```

struct xps_dev_maps {
    struct rcu_head rru;
    struct xps_map __rcu *cpu_map[0];
};

#define XPS_DEV_MAPS_SIZE (sizeof(struct xps_dev_maps) +
(nr_cpu_ids * sizeof(struct xps_map *)))

```

在 `dev_queue_xmit` 函数发送数据包时，会调用 `netdev_pick_tx` 选定发送队列。由 `skb` 和 `dev` 信息获得对应的发送队列索引函数位于 `net/core/flow_dissector.c` 文件。

```

364 struct netdev_queue *netdev_pick_tx(struct net_device *dev,
365             struct sk_buff *skb)
366 {
367     int queue_index = 0;
// real_num_tx_queues 是 NICs 当前发送数据包队列活跃的队列数。如果不是 1，则表示启用了多队列技术。
369     if (dev->real_num_tx_queues != 1) {
370         const struct net_device_ops *ops = dev->netdev_ops;//获得 NICs 操作方法
371         if (ops->ndo_select_queue)//NICs 支持队列选择。
372             queue_index = ops->ndo_select_queue(dev, skb);
373         else
374             queue_index = __netdev_pick_tx(dev, skb);//见下面。
375         queue_index = dev_cap_txqueue(dev, queue_index);
376     }
377
378     skb_set_queue_mapping(skb, queue_index);//存储设备多队列映射索引到 skb 中。
379     return netdev_get_tx_queue(dev, queue_index);
380 }

```

`__netdev_pick_tx` 用于获取发送队列映射索引，如果映射索引变化，那么路由项相应的也要跟着变化。

```

337 u16 __netdev_pick_tx(struct net_device *dev, struct sk_buff *skb)
338 {
339     struct sock *sk = skb->sk;
340     int queue_index = sk_tx_queue_get(sk); //获得 skb 的发送队列映射的索引值。
// 如果 queue_index 非法，表明发送队列索引值需要重建，或者 ooo_okay 允许映射到发送队列的 skb 被改变标志置位，则重新//映射
342     if (queue_index < 0 || skb->ooo_okay ||
343         queue_index >= dev->real_num_tx_queues) {
344         int new_index = get_xps_queue(dev, skb); //计算新的映射索引值，见下文。
345         if (new_index < 0) //返回值小于 0，说明不支持 XPS。
346             new_index = skb_tx_hash(dev, skb); //根据设备是否支持发送队列映射方法获取发送队列索引
347
//新的索引值和原理的索引值，不一致，则需要更换 skb 的路由项，以实现流控
348     if (queue_index != new_index && sk) {
349         struct dst_entry *dst =
350             rcu_dereference_check(sk->sk_dst_cache, 1); //获得路由项
351
352         if (dst && skb_dst(skb) == dst)
353             sk_tx_queue_set(sk, queue_index);
354
355     }
356
357     queue_index = new_index;
358 }
359
360     return queue_index;
361 }

```

```
299 static inline int get_xps_queue(struct net_device *dev, struct sk_buff *skb)
300 {
301 #ifdef CONFIG_XPS
302     struct xps_dev_maps *dev_maps;
303     struct xps_map *map;
304     int queue_index = -1;
305
306     rcu_read_lock();
307     dev_maps = rcu_dereference(dev->xps_maps);
308     if (dev_maps) {
309         map = rcu_dereference(
310             dev_maps->cpu_map[raw_smp_processor_id()]); //获取当前 CPU 的 XPS 流控映射表
311         if (map) {
312             if (map->len == 1)//XPS 流控映射表只有一项，则直接返回该项。
313                 queue_index = map->queues[0];
314             else {
315                 u32 hash;
316                 if (skb->sk && skb->sk->sk_hash)// sk_hash 是协议查找表的哈希值。
317                     hash = skb->sk->sk_hash;
318                 else//否则，需要根据协议计算这个哈希值。
319                     hash = (__force u16) skb->protocol ^
320                         skb->rxhash;//接收队列的哈希值。
321                 hash = jhash_1word(hash, hashrnd); //上述哈希值再次哈希
322                 queue_index = map->queues[
323                     ((u64)hash * map->len) >> 32]; //新的发送队列索引值
324             }
325             if (unlikely(queue_index >= dev->real_num_tx_queues))
326                 queue_index = -1;
327         }
328     }
329     rcu_read_unlock();
330
331     return queue_index;
332 #else
333     return -1;
334 #endif
335 }
```

第十六章 PHY

16.1 PHY

本章和OSI模型中的物理层和数据链路层关系密切。在嵌入式SOC上，通常集成有ARM核和MAC控制器，以及增加数据传输带宽的MAC专用DMA，对这种形式的SOC通常使用外接物理PHY设备的方法，外接的PHY芯片如RTL8201F、88E1111、88E6096等，集成型以太网控制器集成了MAC和PHY，如DM9000、RTL8139CP等，它们常用于没有MAC控制器的SOC上，如S3c2440。MAC控制器将收到的数据通过MII、SMII、GMII、RGMI等接口将数据传递给CPU。向下PHY将MAC将数据转换成模拟信号通过RJ45向外传输、或者通光模块将PHY的模拟信号转换成光信息传输。

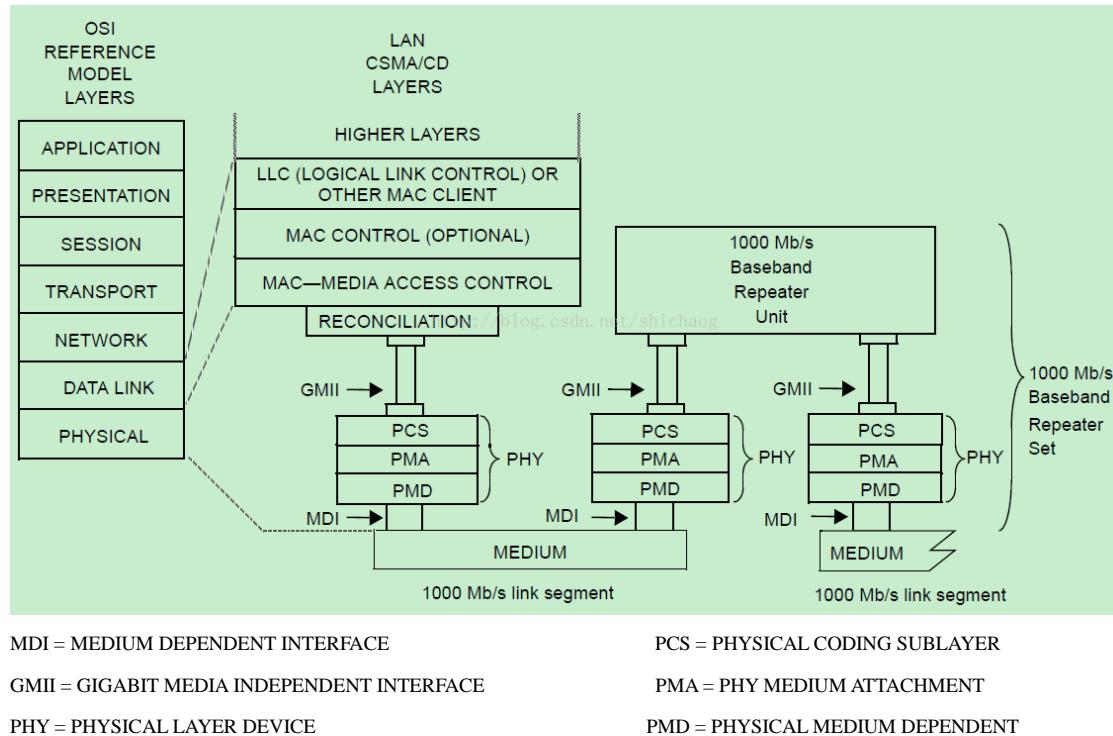


图 16.1.1 千兆以太网架构 ieee802.3 clause34

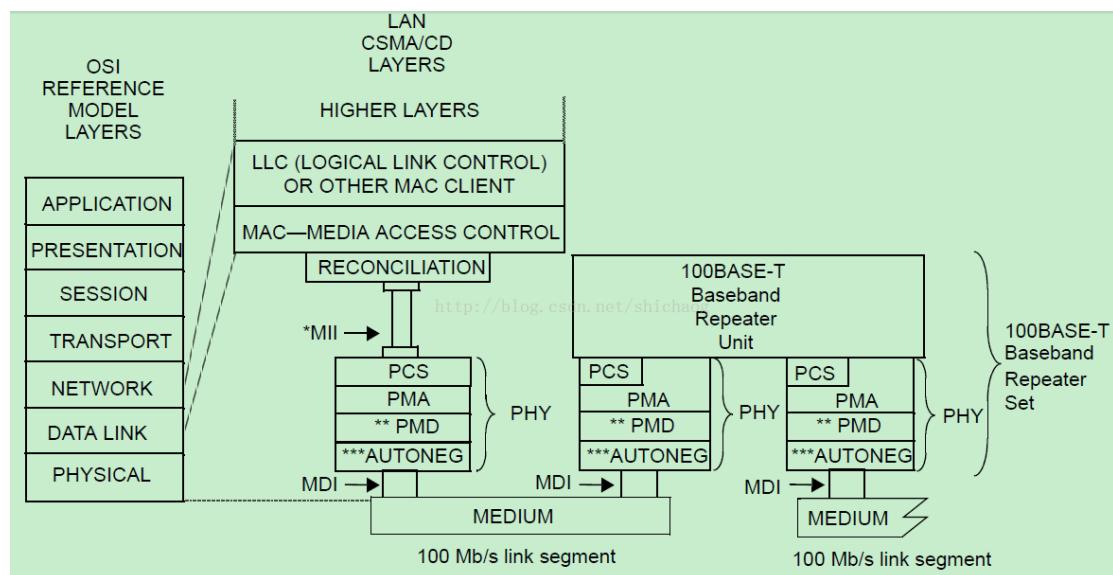


图 16.1.2 百兆以太网 clause21

100M/1000M 常用的编码格式分别是 4B/5B 和 8B/10B；100M-TX 具有自适应功能，在两个网卡连接后，各自会发送 Fast Link Pulse 脉冲，通过该脉冲检测出双方通信速率和各自通信模式，并根据该模式自动选择最优的工作模式。这种自动模式选择由 PHY 芯片实现，通常 PHY 芯片将其称之为自协商（Auto Negotiation）这自适应功能在万兆模式就不再支持了。

PHY 将信号按如下的格式进行传输：

前导符+开始位+目的 MAC 地址+源 MAC 地址+类型长度+数据+padding (optional)
+32bitCRC

对于百兆，

前导符是：

10101010 10101010 10101010 10101010 10101010 10101010 10101010 10101010

开始位是：

10101011

PHY 和 MAC 连接的方式主要有三种：

- 集成型，距离最短
- 同一块 PCB 上，通过铜制导线相连
- 通过非屏蔽双绞线、屏蔽双绞线、光纤相连

RECONCILIATION SUBLAYER 就是针对第三中情况而设计，其使 MAC 层可以用一个方法向 PHY 发送和接收数据而不必关心和 PHY 的具体连接方式。

MII 和 GMII 实际上是承载信号传输的总线定义，它们的时钟线和数据信号形式略有差别。控制 PHY 工作的总线是 mdio。

clause22.2.4 参看 PHY 寄存器及其每个 bit 的定义。所有 PHY 芯片遵循该手册的定义，常用到的是前五个寄存器。

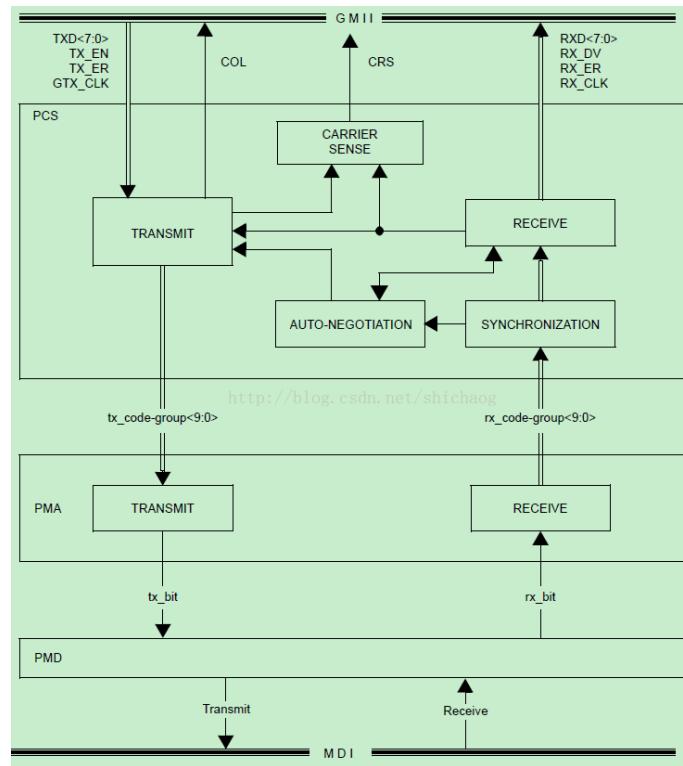


图 16.2.1 1000BASE-X PCS 和 PMA 的职责细分

从上图可以看到 PCS 和 MAC 之间的传输都是 8bit, 而 PCS 和 PMA 之间的传输数据位宽变成了 10bit。

PCS 层主要完成 PCS 传输、载波侦听、同步、PCS 接收和自协商。

PMA 实现 8B 到 10B 的映射功能。

PMD 实现一个串行器和解串器的功能, 即 PMA 传递过来的是并行 10bit 分成 10 次, 每次一个 bit 传输到 PMD 层。最终在网线或者光纤上传递是一个比特一个比特串行传输的。这部分通常包括: 混合信号处理技术减少近端反射、自适应均衡、基线漂移校正 BLW、串扰消除、回波消除、时钟回复、错误校正。

LLC 识别网络协议, 对其进行封装。

16.2 MAC 驱动

单独一个 PHY 是没法进行数据传输的, 还有 MAC 控制器也是需要初始化的。由于不同厂商的 MAC 控制器细节不同, 所以这里并不详细。

```
24 static int XXX_drv_probe(struct platform_device *pdev)
25 {
26     struct device_node *np = pdev->dev.of_node;
27     struct net_device *ndev;
28     struct XXX_info *lp;
29     struct resource *res;
30     const char *macaddr;
31     int ret_val = 0;
32
33 //alloc_etherdev 为以太网设备申请空间
34     ndev = alloc_etherdev(sizeof(struct XXX_info));
35     if (ndev == NULL) {
36         dev_err(&pdev->dev, "alloc_etherdev fail.\n");
37         return -ENOMEM;
38     }
39 //lp 的指针用于存储 34 行申请的以太网设备都有字段+一些满足 MAC 厂商特有功能的字段
40     lp = netdev_priv(ndev);
41 //设备树获得 MAC 控制器基地址并映射该地址
42     res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
43     lp->regbase = devm_ioremap(&pdev->dev, res->start, resource_size(res));
44 //MAC 中断号
45     ndev->irq = platform_get_irq(pdev, 0);
46 //设置 private 字段
47     SET_NETDEV_DEV(ndev, &pdev->dev);
48     ndev->devdma_mask = pdev->dev.dma_mask;
49     ndev->dev.coherent_dma_mask = pdev->dev.coherent_dma_mask;
50
51     spin_lock_init(&lp->lock);
52     lp->ndev = ndev;
53     lp->msg_enable = netif_msg_init(msg_level, NETIF_MSG_DRV);
54 //设备树解析
55     XXX_of_parse(np, lp);
56
57     if (lp->ipc_tx)
58         ndev->features |= NETIF_F_HW_CSUM;
59     /* request gpio for PHY reset control */
60     if (gpio_is_valid(lp->rst_gpio)) {
61         ret_val = devm_gpio_request(&pdev->dev, lp->rst_gpio, "phy reset");
62         gpio_direction_output(lp->rst_gpio, !lp->rst_gpio_active);
```

```

97      }
//mdio 读写方法。
100     lp->new_bus.name = "XXX MII Bus",
101     lp->new_bus.read = &XXX_mdio_read,
102     lp->new_bus.write = &XXX_mdio_write,
103     lp->new_bus.reset = &XXX_mdio_reset,
104     snprintf(lp->new_bus.id, MII_BUS_ID_SIZE, "%s", pdev->name);
105     lp->new_bus.priv = lp;
106     lp->new_bus.irq = kmalloc(sizeof(int)*PHY_MAX_ADDR, GFP_KERNEL);

114     lp->new_bus.parent = &pdev->dev;
115     lp->new_bus.state = MDIOBUS_ALLOCATED;
116
//这行会注册该设备，mdiobus_register 会被调用，完成 mdio 驱动注册。
118     ret_val = of_mdiobus_register(&lp->new_bus, pdev->dev.of_node);
//找到第一个 PHY 设备，该 MAC 将使用这个 PHY 设备进行通信
125     lp->phydev = phy_find_first(&lp->new_bus);
/* 初始化该以太网设备
ndev->header_ops= &eth_header_ops;
ndev->type = ARPHRD_ETHER;
ndev->hard_header_len = ETH_HLEN;
ndev->mtu = ETH_DATA_LEN;
ndev->addr_len= ETH_ALEN;
ndev->tx_queue_len= 1000; /* Ethernet wants good queues */
ndev->flags = IFF_BROADCAST|IFF_MULTICAST;
ndev->priv_flags|= IFF_TX_SKB_SHARING;

memset(dev->broadcast, 0xFF, ETH_ALEN);
*/
137     ether_setup(ndev);
//net device operations 初始化，这是一个非常重要的函数操作集，这是各厂商针对各自 MAC 控制器而写，初始化、打开、发送、多播、超时、邻居表初始化等
138     ndev->netdev_ops = &XXX_netdev_ops;
//NAPI 注册，该功能使得一个接收或者发送中断可以发送多个数据包，这样提高数据包的收发效率。
139     ndev->watchdog_timeo = XXX_TX_WATCHDOG;
140     netif_napi_add(ndev, &lp->napi, XXX_napi, XXX_NAPI_WEIGHT);
//解析设备树，获得 MAC 地址
142     macaddr = of_get_mac_address(pdev->dev.of_node);
143     if (macaddr)
144         memcpy(ndev->dev_addr, macaddr, ETH_ALEN);
145
146     if (!is_valid_ether_addr(ndev->dev_addr))
147         eth_hw_addr_random(ndev);
//设置设备不可用状态
149     XXX_disable(lp);
153     if (gpio_is_valid(lp->rst_gpio))
154         gpio_set_value_cansleep(lp->rst_gpio, lp->rst_gpio_active);
//ethtool 工具支持
156     SET_ETHTOOL_OPS(ndev, &XXX_ethtool_ops);
//注册网卡设备，138 行提及的初始化成员 ndo_init 会被调用完成设备的初始化。
157     ret_val = register_netdev(ndev);
163     platform_set_drvdata(pdev, ndev);
164     dev_notice(&pdev->dev, "MAC Address[%pM].\n", ndev->dev_addr);
165
166     return 0;
178 }

179 static const struct of_device_id XXX_eth_dt_ids[] = {
180     { .compatible = "XXX,eth" },

```

```

181     /* sentinel */
182 };
183 static struct platform_driver XXX_driver = {
184     .probe      = XXX_drv_probe,
185     .remove     = XXX_drv_remove,
186     .driver = {
187         .name   = "XXX-eth",
188         .owner  = THIS_MODULE,
189         .of_match_table = XXX_eth_dt_ids,
190     },
191 };
192
193 module_platform_driver(XXX_driver);

```

193module_platform_driver 会创建 module_init(XXX_driver)和 module_exit(XXX_driver)两个宏，XXX_driver 的 probe 函数会被调用。

24~178 都是 XXX_drv_probe 的内容。

这里的 MAC 控制器使用 device-tree 解析设备，这部分内容在《Linux 系统启动那些事——基于 Linux 3.10 内核》有涉及，这里略过若干行。

```

struct net_device_ops {
//网络设备注册时会被调用
int (*ndo_init)(struct net_device *dev);
//打开一个网卡，配置硬件使能，注册中断服务函数，使能 NAPI，使能数据发送队列，初始化 DMA，检测 link 状态，//调用 linkwatch_fire_event 通知内核其它部分该事件，以让其它部分进行相应的处理，如路由表项的更新。启动 PHY。
int (*ndo_open)(struct net_device *dev);
//禁止发送、禁止 NAPI，释放中断号，停止 PHY、设备无载波调用 linkwatch_fire_event 通知内核其它部分该事件。
int (*ndo_stop)(struct net_device *dev);
//网络设备发送数据包函数，这部分是 SOC 息息相关的 DMA 和相关环形缓冲区的管理。
netdev_tx_t (*ndo_start_xmit) (struct sk_buff *skb,
                                struct net_device *dev);
//MAC 地址重置
int (*ndo_set_mac_address)(struct net_device *dev,
                           void *addr);
//设备 MAC 地址合法性验证
int (*ndo_validate_addr)(struct net_device *dev)
//用户空间 ioctl 的内核支持。
int (*ndo_do_ioctl)(struct net_device *dev,
                     struct ifreq *ifr, int cmd);
//网卡设备的管理
int (*ndo_set_config)(struct net_device *dev,
                      struct ifmap *map);
//MTU 变更
int (*ndo_change_mtu)(struct net_device *dev,
                      int new_mtu);
//邻居设置
int (*ndo_neigh_setup)(struct net_device *dev,
                       struct neigh_parms *);
//发送超时
void (*ndo_tx_timeout) (struct net_device *dev);
};

16.3 mdio 总线初始化
driver/net/phy/mdio_bus.c
91 static struct class mdio_bus_class = {
92     .name      = "mdio_bus",
93     .dev_release = mdiobus_release,
94 };
447 struct bus_type mdio_bus_type = {

```

```

448     .name      = "mdio_bus",
449     .match     = mdio_bus_match,
450     .pm        = MDIO_BUS_PM_OPS,
451     .dev_attrs = mdio_dev_attrs,
452 };
453 EXPORT_SYMBOL(mdio_bus_type);
454
455 int __init mdio_bus_init(void)
456 {
457     int ret;
458
459     ret = class_register(&mdio_bus_class);
460     if (!ret) {
461         ret = bus_register(&mdio_bus_type);
462         if (ret)
463             class_unregister(&mdio_bus_class);
464     }
465
466     return ret;
467 }

```

459 在/sys/class/目录下注册一个 mdio_bus 类， mdiobus_release 是删除这个类的方法。
用户空间可以通过这个类得到 mdio 总线的信息。

461 注册一个驱动核心层， bus_register 注册 mdio 总线，后面 PHY 设备和 PHY 设备驱动会被挂载在这个总线上， I2C 等总线也是调用这个接口进行注册的。

还记得上一节的 XXX_drv_probe 函数的第 118 行的 of_mdiobus_register 函数不？这个函数注册 mii 总线并且根据设备树创建 PHY 设备。

```

33 int of_mdiobus_register(struct mii_bus *mdio, struct device_node *np)
34 {
35     struct phy_device *phy;
36     struct device_node *child;
37     const __be32 *paddr;
38     u32 addr;
39     bool is_c45, scanphys = false;
40     int rc, i, len;
41
//禁止自举探测，因为在设备树中已经有 PHY 设备的信息了。
44     mdio->phy_mask = ~0;
45
//清除 PHY 设备的中断。
47     if (mdio->irq)
48         for (i=0; i<PHY_MAX_ADDR; i++)
49             mdio->irq[i] = PHY_POLL;
//获得 PHY 在设备树中的节点
51     mdio->dev.of_node = np;
//这里的 mdiobus_register 并不是总线的注册，而是调用 mdiobus_scan 扫描 PHY 设备，但是对于设备树方法会跳过
//mdiobus_scan 方法，因为 PHY 设备的相关信息已经存放在设备树的节点里了。得到 PHY 设备后将其挂接到 mii 总线上。
54     rc = mdiobus_register(mdio);
55     if (rc)
56         return rc;
57
58     /*59~108 循环子节点，为存在的每一个 PHY 创建一个 phy_device 表示结构体 */
59     for_each_available_child_of_node(np, child) {
//PHY 地址，一个 mii 总线最多支持 32 个 PHY 设备，所以这里的值为 0~31。
60         paddr = of_get_property(child, "reg", &len);
61         addr = be32_to_cpup(paddr);

```

```

//802.3-c45 和 802.3-c22 两种标准的 PHY ID 读取略有区别。
82     is_c45 = of_device_is_compatible(child,
83             "ethernet-phy-ieee802.3-c45");
//验证 PHY ID 信息正确性。正确的话会调用 phy_device_create 创建 phy_device 结构体来表示 PHY 设备。这个结构体设置的信息主要有：
//速率、双工、link、以及将 PHY 状态设置为 PHY_DOWN 还创建一个内核守护进程，用于维护 PHY 状态变化。
84     phy = get_phy_device(mdio, addr, is_c45);
//增加该节点的引用计数。
95     of_node_get(child);
96     phy->dev.of_node = child;
//得到了 PHY 的所有信息，这里注册 PHY 设备。主要是将 mii 总线上存放 PHY 设备的 phy_map 数组中，数组的索引是 PHY 的物理地址，最大是 31。
99     rc = phy_device_register(phy);
108 }
//114~161 处理在 PHY 设备树中没有被赋予地址的 PHY 设备的初始化。过程同上。
114     for_each_available_child_of_node(np, child) {
115         /* Skip PHYs with reg property set */
116         paddr = of_get_property(child, "reg", &len);
117         if (paddr)
118             continue;
119         is_c45 = of_device_is_compatible(child,
120                 "ethernet-phy-ieee802.3-c45");
121
122         for (addr = 0; addr < PHY_MAX_ADDR; addr++) {
123             /* skip already registered PHYs */
124             if (mdio->phy_map[addr])
125                 continue;
126             phy = get_phy_device(mdio, addr, is_c45);
127             if (!phy || IS_ERR(phy))
128                 continue;
129             /* Associate the OF node with the device structure so it
130              * can be looked up later */
131             of_node_get(child);
132             phy->dev.of_node = child;
133
134             /* All data is now stored in the phy struct;
135              * register it */
136             rc = phy_device_register(phy);
137             dev_info(&mdio->dev, "registered phy %s at address %i\n",
138                     child->name, addr);
139             break;
140         }
141     }
142
143     return 0;
144 }

```

16.3 PHY 驱动

16.3.1 PHY 初始化

PHY 层的初始化，PHY 驱动通常可以使用缺省内核 PHY 驱动，该驱动是按照 ieee802.3 协议规定的标准来设计的。

PHY 层从 driver/net/phy/phy_device.c 文件开始。

```

1112 static struct phy_driver genphy_driver = {
1113     .phy_id      = 0xffffffff,

```

```

1114     .phy_id_mask    = 0xffffffff,
1115     .name        = "Generic PHY",
1116     .config_init   = genphy_config_init,
1117     .features      = 0,
1118     .config_aneg   = genphy_config_aneg,
1119     .read_status    = genphy_read_status,
1120     .suspend       = genphy_suspend,
1121     .resume        = genphy_resume,
1122     .driver        = {.owner= THIS_MODULE, },
1123 };
1124
1125 static int __init phy_init(void)
1126 {
1127     int rc;
//mdio 总线初始化
1128     rc = mdio_bus_init();
1129     if (rc)
1130         return rc;
//PHY 设备注册。
1131     rc = phy_driver_register(&genphy_driver);
1132     if (rc)
1133         mdio_bus_exit();
1134
1135     return rc;
1136
1137 }
1138
1139
1140 subsys_initcall(phy_init);
1141
1142 对 driver_register 的封装，该函数用于向 mdio 总线上注册 genphy_driver，并扫描 mdio 总线，如果发现有没有驱动的设备，会使用这个驱动进行绑定，一个驱动可以对应多个设备。从其名称可以知道，这个驱动具有通用性。
1143
1144 include/uapi/linux/mii.h
1145
#define MII_BMCR 0x00/* Basic mode control register */
#define MII_BMSR 0x01/* Basic mode status register */
#define BMCR_ISOLATE 0x0400/* Isolate data paths from MII */
#define BMCR_PDOWN 0x0800/* Enable low power state */
#define BMCR_ANENABLE 0x1000/* Enable auto negotiation */
1146
drivers/net/phy/phy_device.c
1147 int genphy_resume(struct phy_device *phydev)
1148 {
1149     int value;
1150
1151     mutex_lock(&phydev->lock);
1152
1153     /*
1154     *static inline int phy_read(struct mii_phy* phy, int reg)
1155     *{
1156     * return phy->mdio_read(phy->dev, phy->mii_id, reg);
1157     *}
1158     */
1159
1160     value = phy_read(phydev, MII_BMCR);
1161     phy_write(phydev, MII_BMCR, (value & ~BMCR_PDOWN));
1162
1163     mutex_unlock(&phydev->lock);
1164
1165     return 0;
1166
1167 }

```

MII_BMCR 寄存器的各位定义如下，其第十一个 bit 置位，以上电，唤醒 PHY 设备，其它的函数操作和这里的类似，参考手册就能看懂。就不再赘述了。

Bit(s)	Name	Description	R/W ^a
0.15	Reset	1 = PHY reset 0 = normal operation	R/W SC
0.14	Loopback	1 = enable loopback mode 0 = disable loopback mode	R/W
0.13	Speed Selection (LSB)	0.6 0.13 1 1 = Reserved 1 0 = 1000 Mb/s 0 1 = 100 Mb/s 0 0 = 10 Mb/s	R/W
0.12	Auto-Negotiation Enable	1 = enable Auto-Negotiation process 0 = disable Auto-Negotiation process	R/W
0.11	Power Down	1 = power down 0 = normal operation ^b	R/W
0.10	Isolate	1 = electrically Isolate PHY from MII or GMII 0 = normal operation ^b	R/W
0.9	Restart Auto-Negotiation	1 = restart Auto-Negotiation process 0 = normal operation ^b	R/W SC
0.8	Duplex Mode	1 = full duplex 0 = half duplex	R/W
0.7	Collision Test	1 = enable COL signal test 0 = disable COL signal test	R/W
0.6	Speed Selection (MSB)	0.6 0.13 1 1 = Reserved 1 0 = 1000 Mb/s 0 1 = 100 Mb/s 0 0 = 10 Mb/s	R/W
0.5	Unidirectional enable	When bit 0.12 is one or bit 0.8 is zero, this bit is ignored. When bit 0.12 is zero and bit 0.8 is one: 1 = Enable transmit from media independent interface regardless of whether the PHY has determined that a valid link has been established 0 = Enable transmit from media independent interface only when the PHY has determined that a valid link has been established	R/W
0.4:0	Reserved	Write as 0, ignore on read	R/W

图：控制寄存器各 bit 定义,摘自 ieee802.3 clause 22.2.4.1

16.3.2 PHY 驱动实例

剩下一个问题就是驱动的编写了。

```
<phy.h>
#define PHY_BASIC_FEATURES (SUPPORTED_10baseT_Half | \
SUPPORTED_10baseT_Full | \
SUPPORTED_100baseT_Half | \
SUPPORTED_100baseT_Full | \
SUPPORTED_Autoneg | \
SUPPORTED_TP | \
SUPPORTED_MII)
static int 8201f_config_init(struct phy_device *phydev)
{
    int val;
    u32 features;
    /* For now, I'll claim that the generic driver supports
     * all possible port types */
    features =PHY_BASIC_FEATURES;
```

```

/* Do we support autonegotiation? */
val = phy_read(phydev, MII_BMSR);
if (val < 0)
    return val;
if (val & BMSR_ANEGCAPABLE)
    features |= SUPPORTED_Autoneg;
if (val & BMSR_100FULL)
    features |= SUPPORTED_100baseT_Full;
if (val & BMSR_100HALF)
    features |= SUPPORTED_100baseT_Half;
if (val & BMSR_10FULL)
    features |= SUPPORTED_10baseT_Full;
if (val & BMSR_10HALF)
    features |= SUPPORTED_10baseT_Half;
if (val & BMSR_ESTATEN) {
    val = phy_read(phydev, MII_ESTATUS);
}
if (val < 0)
    return val;
if (val & ESTATUS_1000_TFULL)
    features |= SUPPORTED_1000baseT_Full;
if (val & ESTATUS_1000_THALF)
    features |= SUPPORTED_1000baseT_Half;
}
phydev->supported = features;
phydev->advertising = features;
return 0;
}

int 8201f_config_aneg(struct phy_device *phydev)
{
    int result;
    if (AUTONEG_ENABLE != phydev->autoneg)
        return genphy_setup_forced(phydev);
    result = genphy_config_advert(phydev);
    if (result < 0) /* error */
        return result;
    if (result == 0) {
        /* Advertisement hasn't changed, but maybe aneg was never on to
         * begin with? Or maybe phy was isolated? */
        int ctl = phy_read(phydev, MII_BMCR);
        if (ctl < 0)
            return ctl;
        if (!(ctl & BMCR_ANENABLE) || (ctl & BMCR_ISOLATE))
            result = 1; /* do restart aneg */
    }
    /* Only restart aneg if we are advertising something different
     * than we were before.*/
    if (result > 0)
        result = genphy_restart_aneg(phydev);
    return result;
}

static struct phy_driver 8201f_driver = {
    .phy_id      = 0x001cc810,
    .name        = "Realtek 8201f",
    .phy_id_mask = 0xffffffff,
    .features    = PHY_BASIC_FEATURES,
    .config_init = 8201f_config_init,
    .config_aneg = 8201f_config_aneg,
    .read_status = genphy_read_status,
}

```

```

.driver      = { .owner = THIS_MODULE,},
};

static int __init 8201f_init(void)
{
    int ret;
    ret = phy_driver_register(&8201f_driver);
    if (ret)
        phy_driver_unregister(&8201f_driver);
    return ret;
}
static void __exit 8201f_exit(void)
{
    phy_driver_unregister(&8201f_driver);
}

module_init(8201f_init);
module_exit(8201f_exit);

```

16.3.3 PHY 状态机

Windows 桌面的右下角有个表示网络状态的图标，如果将无线网以及有线网从 RJ45 拔出，会发现显示网络图标的图标变成一个黄色三角形，三角形里面还有一个感叹号，如果将鼠标放上去，显示无网络访问，插上网线或者打开无线网，右下角的图标会显示连上网络信号的强度。这个过程就涉及到 PHY 的状态是如何的，并且还需要某种机制通知网络协议栈。这节就是 Linux 下 PHY 状态的管理，PHY 状态的要比上连上和断开这两种状态多很多。

一个 PC 可以有多张网卡，每张网卡会对应一个 PHY 物理芯片，每一个 PHY 芯片状态会由一个 PHY 状态机管理，在注册 PHY 设备时这个状态机会被创建。其创建流程见图 16.3.1。其起点 of_mdioibus_register 是在 16.2 节的 XXX_drv_probe 函数调用的。

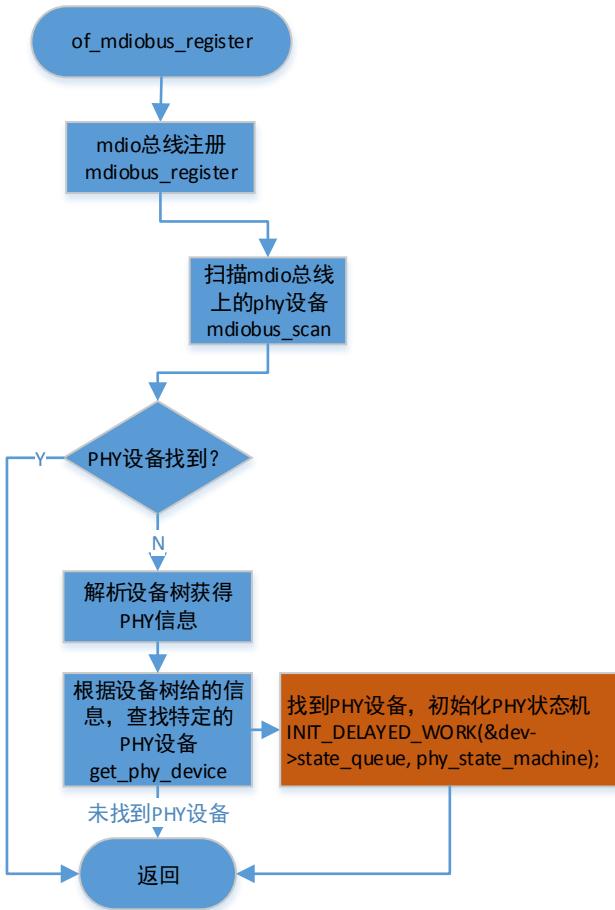


图 16.3.3 PHY 状态机创建

`INIT_DELAYED_WORK(&dev->state_queue, phy_state_machine);`
其第一个参数传递一个 `delayed_work` 类型的结构体。

```

struct delayed_work {
    struct work_struct work;
    struct timer_list timer;
};

#define INIT_DELAYED_WORK(_work, _func) \
do { \
    INIT_WORK(&(_work)->work, (_func)); \
    init_timer(&(_work)->timer); \
} while (0)

```

`phy_state_machine()` 这个函数由上面注册的 `timer` 间隔一秒钟调用一次。

PHY 各状态如下：

DOWN: PHY 设备和 PHY 驱动还未准备好，处于这个状态的 PHY，`probe` 方法将被调用，`probe` 函数将 PHY 设置为 STARTING 或者 READY 状态。

STARTING: PHY 设备正在启动，以太网驱动还未准备好。

READY: PHY 已经初始化完毕，可以接收和发送数据包，但是控制器还没有。`phy_probe()` 会设置这个状态。

PENDING: PHY 设备正在启动中，以太网驱动已经准备好。`phy_start()` 会设置这个状态。

UP: PHY 和与之连接的 MAC 控制器已经可以工作，中断在这个状态下将是能。设置自协商 AN 的定时器。

AN: 正在自协商 link 状态，当前 link 状态是 down，`phy_timer()` 在检测到 PHY 处于 UP 状态时会设置，在 PHY 使能了自协商时 `config_aneg()` 会设置这个状态。

自协商结果：

没有 link，状态设置成 NOLINK，

有 link，状态设置成 RUNNING，调用 adjust_link

超时，重试自协商

自协商未完成，且不支持 magic_aneg，状态被设置成 FORCING

NOLINK：PHY 已经初始化完毕，但是物理链路并不存在（网线、光纤拔出）

timer 注意到 link 通了，状态切到 RUNNING

config_aneg 切到 AN

phy_stop 切到 HALTED 状态

FORCING：PHY 被强制设定

link 已经可以工作，状态设置到 RUNNING

link down，重试 FORCING

phy_stop 切到 HALTED 状态

RUNNING：PHY 在接收、发送数据

当轮询 PHY 状态时 timer 会设置 CHANGELINK 标志，

irq 会设置 CHANGELINK

config_aneg 切到 AN

phy_stop 切到 HALTED 状态

CHANGELINK：link 状态改变

link 连通，timer 将状态设置为 RUNNING

link 不通，timer 将状态设置为 NOLINK

phy_stop 切到 HALTED 状态

HALTED：PHY 已经准备好，但是轮询和中断还未完成；或者 PHY 处于错误状态。

phy_start 将状态设置为 RESUMING

RESUMING：PHY 处于 HALTED 状态，这里设置，让其再次运行。

FORCING 或者 AN 完成时，状态将被设置为 RUNNING

AN 未完成，状态将被设置成 AN

phy_stop 将状态设置成 HALTED

PHY 设备状态以及各状态的设置函数如图 16.3.2：

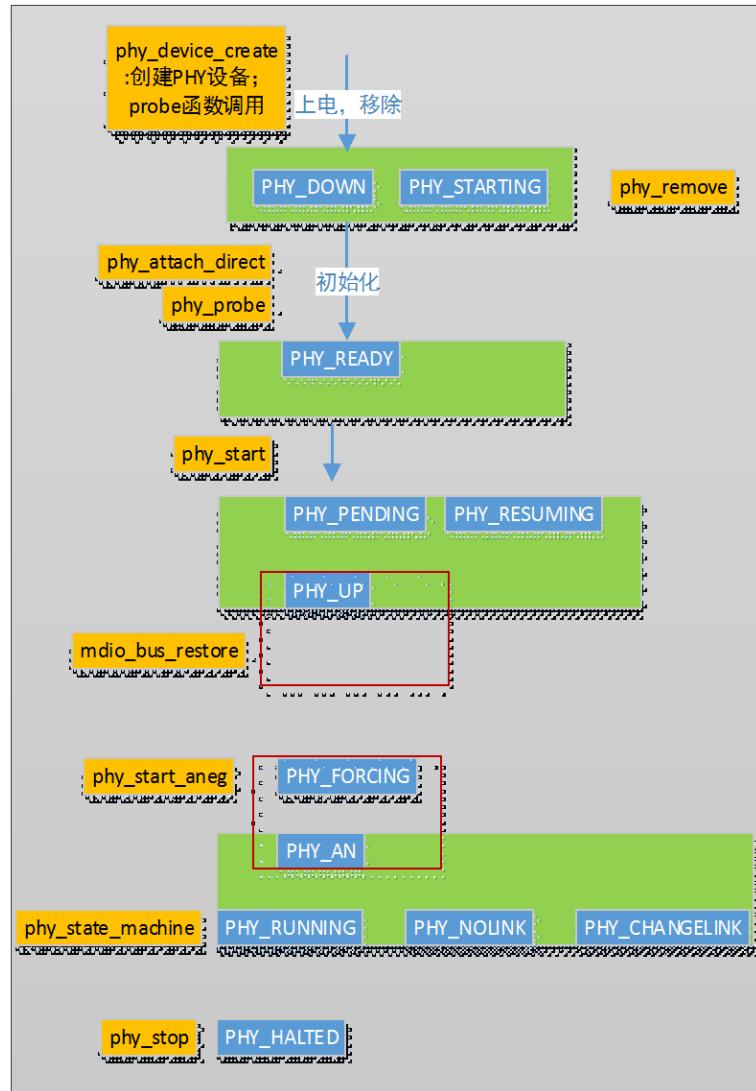


图 16.3.2 PHY 设备及状态管理函数

`phy_state_machine` 会调用下面三个函数，其中 `netif_carrier_off`、`netif_carrier_on` 用于向内核通知链路层载波信号存不存在，至于检测工作，是由 PHY 芯片完成，PHY 芯片的一个特性是自协商，也有一种叫 switch 的芯片也支持自协商，但是 switch 包含了多个 PHY，PHY 完成检测物理线路上是否有载波，PHY 的检测是由硬件逻辑电路的状态机完成的。

```

netif_carrier_off
phy_aneg_done
netif_carrier_on

```

这里来看看一个物理芯片 PHY 状态发生改变是如何通知给 CPU 的。`netif_carrier_on` 和 `netif_carrier_off` 都会判断 PHY 设备的载波状态，对于 on，要设置载波状态，off 则清除载波状态，然后调用 `linkwatch_fire_event` 发送事件给内核。`linkwatch_fire_event` 的定义如下：

```

void linkwatch_fire_event(struct net_device *dev)
{
    if (!test_and_set_bit(__LINK_STATE_LINKWATCH_PENDING, &dev->state)) {
        linkwatch_add_event(dev);
    } else if (!urgent)
        return;

    linkwatch_schedule_work(urgent);
}

```

在网络设备一节中提到完了设备的表示结构体是 struct net_device,，其有一个字段是 link_watch_list，该链表是用来存放 link 发生改变时的事件，所以 __linkwatch_run_queue 工做就是将发生 link 状态改变的事件的网络设备 net_device 的 link_watch_list 链表挂到 lweventlist。lweventlist 在 net/core/link_watch.c 一开始就声明的链表。

```
static LIST_HEAD(lweventlist);
static void linkwatch_add_event(struct net_device *dev)
{
    if (list_empty(&dev->link_watch_list)) {
        list_add_tail(&dev->link_watch_list, &lweventlist);
    }
}
```

对 linkwatch_event 的调用被间接调用了，linkwatch_schedule_work 调用 schedule_delayed_work(&linkwatch_work, 0); 向系统全工作队列 system_wq 添加一项，就是下面 DECLARE_DELAYED_WORK 声明的 delayed 工作。

```
static DECLARE_DELAYED_WORK(linkwatch_work, linkwatch_event);
```

linkwatch_event 将被内核调度运行，函数如下：

```
static void linkwatch_event(struct work_struct *dummy)
{
    rtnl_lock();
    __linkwatch_run_queue(time_after(linkwatch_nextevent, jiffies));
    rtnl_unlock();
}
```

__linkwatch_run_queue 遍历 lweventlist 链表上产生 link 状态改变的设备，并将其从链表上移除，并清除 net_device 的 __LINK_STATE_LINKWATCH_PENDING 标志。然后发送 NETDEV_CHANGE 通知前面注册的 netdev_chain 通知链。

第十七章 ping-icmp

Ping 命令经常被使用到， ping 命令使用的是 icmp 协议，
头文件：

```
#include <sys/socket.h>
#include <sys/time.h>
#include <sys/types.h>

#include <netdb.h>

#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <netinet/in.h>

#include <unistd.h>

#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#include <ctype.h>

#include <errno.h>

#define IP_HEAD_LEN 20
#define ICMP_LEN 8
#define BUFFER_SIZE 50 * 1024

int ip_fd,p_id;

extern int packet_len;
extern int sequence;

struct sockaddr_in send_addr,from_addr;

char send_buf[1024];

struct hostent *host;
struct timeval tv;

int flag;
```

校验和计算

```
unsigned short ip_checksum(unsigned short *pcheck, int check_len)
{
    int nleft = check_len;

    int sum = 0;

    unsigned short *p=pcheck;

    unsigned short result = 0;

    while(nleft >1)
    {
        sum += *p++;
    }
```

```

        nleft-=sizeof(unsigned short);
    }
    if(nleft == 1)
    {
        *(unsigned char *)(&result)=*(unsigned char *)p;
        sum += result;
    }

    sum=(sum>>16)+(sum &0xffff);

    sum+=(sum>>16);

    result = ~sum;

    return result;
}

```

发送

```

int sequence = 0;
int packet_len = IP_HEAD_LEN+ICMP_LEN;

void send_ip()
{
    if(sequence != 0 && !flag)
        printf("Destination Host Unreachable\n");

    int len;
    struct icmphdr *icmp_p;

    icmp_p =(struct icmphdr *)send_buf;

    icmp_p->type = ICMP_ECHO;

    icmp_p->code = 0;

    (icmp_p->un).echo.id = p_id;

    (icmp_p->un).echo.sequence = sequence++;

    gettimeofday((struct timeval *) (icmp_p +1),NULL);

    len = sizeof(struct timeval)+packet_len;

    icmp_p->checksum = 0;

    icmp_p->checksum = ip_checksum((unsigned short *)icmp_p,len);

    if(sendto(ip_fd,send_buf,len,0,(struct sockaddr *)&send_addr,sizeof(send_addr))<0)
    {
        printf("send to error\n");
        exit(1);
    }
}

```

接收

```
void recv_ip()
```

```

{
    char recv_buf[1024];

    int len,n;

    struct ip *ip_p;

    struct timeval *timeval_now,*time_send;

    struct timeval now;

    struct icmphdr *icmp_p;

    float delay;

    int iphead_len,icmp_len;

    for(;;)
    {
        n = recvfrom(ip_fd,recv_buf,sizeof(recv_buf),0,NULL,NULL);
        if(n<0)
        {
            if(errno == EINTR)
            {
                continue;
            }
            else
            {
                printf("recvfrom error\n");
                continue;
            }
        }
        ip_p=(struct ip*)recv_buf;

        iphead_len = ip_p->ip_hl<<2;

        icmp_p = (struct icmphdr *)(recv_buf +iphead_len);
        icmp_len = n-iphead_len;

        if(icmp_len < 8)
            printf("error icmp len=%d\n",icmp_len);

        if(icmp_p->type == ICMP_ECHOREPLY)
        {
            if((icmp_p->un).echo.id != p_id)
                return;
            if(icmp_len <16)
                printf("icmplen = %d\n",icmp_len);
            flag = 1;
        }

        gettimeofday(&now,NULL);

        time_now =&now;

        time_send =(struct timeval *) (icmp_p +1);

        if((time_now->tv_usec-=time_send->tv_usec)<0)

```

```

    {
        time_now->tv_sec--;
        time_now->tv_usec +=1000000;
    }

    time_now->tv_sec -=time_send->tv_sec;

    delay=time_now->tv_sec*1000.0+time_now->tv_usec/1000.0;

    printf("%d(%d) bytes from %s: icmp_seq=%d ttl=%d time=%.3fms\n",icmp_len, n,
inet_ntoa(send_addr.sin_addr), (icmp_p->un).echo.sequence,ip_p->ip_ttl, delay);
}

}

```

Ping 包回显

```

void handler_alarm(int signo)
{
    send_ip();
    alarm(1);
}

void ping(char *argv)
{
    struct sigaction act;

    act.sa_handler = handler_alarm;
    act.sa_flags = 0;
    sigemptyset(&act.sa_mask);
    sigaction(SIGALRM,&act,NULL);

    p_id= getpid();

//    setsockopt(ip_fd,SOL_SOCKET,SO_RCVTIMEO, &tv,sizeof(struct timeval));
//    setsockopt(ip_fd,SOL_SOCKET,SO_RCVBUF,&BUFFER_SIZE,sizeof(BUFFER_SIZE));

    send_addr.sin_family = AF_INET;
    if( (host= gethostbyname(argv))== NULL)
    {
        printf("get host by name error\n");
        exit(0);
    }
    memcpy(&send_addr.sin_addr,(struct in_addr *)host->h_addr,host->h_length);

    printf("ping %s(%s) %d(%d) bytes of data\n",argv,inet_ntoa(send_addr.sin_addr),sizeof(struct timeval),sizeof(struct
timeval)+IP_HEAD_LEN+ICMP_LEN);

    flag = 0;

    raise(SIGALRM);

    recv_ip();
}

```

套接字申请

```
int main(int argc,char **argv)
{
    if(argc !=2 )
    {
        printf("usage:ping <hostip_address>\n");
        exit(1);
    }

    ip_fd=socket(AF_INET,SOCK_RAW,IPPROTO_ICMP);

    if(ip_fd <0 )
    {
        printf("raw socket error\n");
        exit(1);
    }

    setuid( getpid() );

    ping(argv[1]);
}
```

第十八章 ipv6 简介

根据思科 Visual Networking Index (VNI) 全球移动数据流量 2014-2019 预测白皮书。到 2019 年，54% (62 亿) 的全球移动设备可以通过 ipv6 连接移动网，而这一数字在 2014 年为 27% (20 亿)；此外在 2014 年移动网络平均速率是 1683kbps，而到 2019 年这一速率将达到近 4Mbps，基于 3G 后者更高的通信标准的速率将是平均速率的 3 倍，智能手机的速率将达到 10Mbps。

以笔者所在的视频监控行业的码流来看，现在 720p 的低清晰度视频的码流是 2Mbps，而高清的 1080p 视频码流达到 2Mbps。这就意味着到 2019 年，实时视频通信将成为现实。现在基于手机等移动工具的通信是发文字短信和语音短信。到 2019 年将是视频短信，此外实时视频通信将会普及，加上现在炒的如火如荼的物联网、智能硬件等，联网能够在节约硬件成本的同时极大的提高设备的智能性。IPv6 技术以其固有的优势，将会逐渐的侵蚀 IPv4 的市场份额。

本章没有像 IPv4 那部分那样详细的介绍 IPv6 协议。这部分内容有些是基于笔者在视频监控领域工作中遇到的，还有些是基于一些文档、白皮书和部分源码完成的。

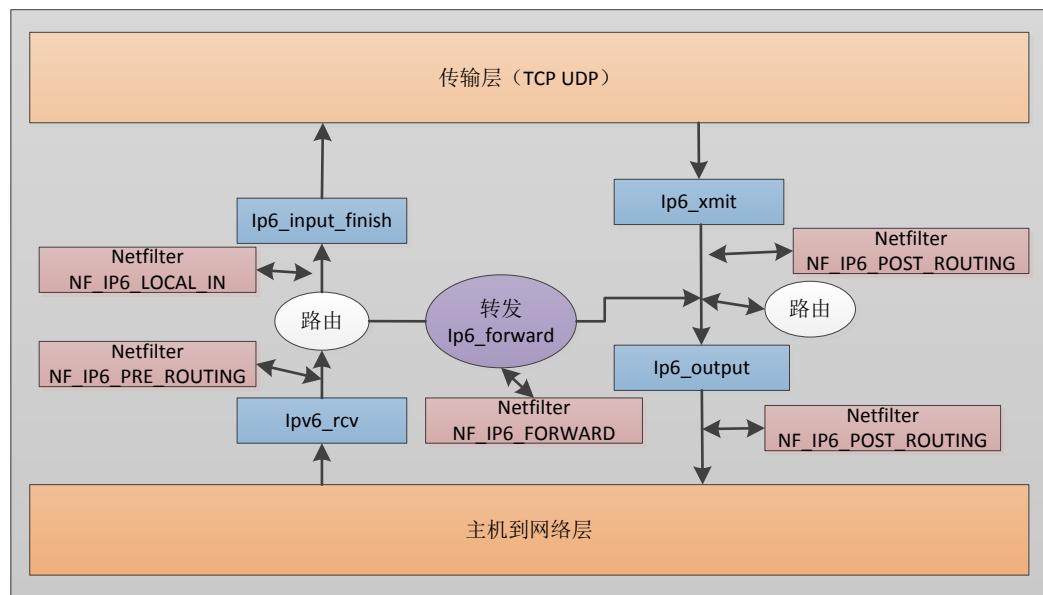


图 18.1 IPv6 实现框架

附录 tcp 测试程序

```
1 /*client.c*/
2
3 #include <sys/types.h>
4 #include <sys/socket.h>
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <string.h>
8 #include <sys/ioctl.h>
9 #include <unistd.h>
10 #include <netdb.h>
11 #include <netinet/in.h>
12
13 #define PORT      11910
14 #define BUFFER_SIZE 1024
15
16 int main(int argc, char *argv[])
17 {
18     int sockfd, sendbytes;
19     char buf[BUFFER_SIZE];
20     struct hostent *host;
21     struct sockaddr_in serv_addr;
22
23     if(argc < 3)
24     {
25         fprintf(stderr,"USAGE: ./client Hostname(or ip address) Text\n");
26         exit(1);
27     }
28
29     /*地址解析函数*/
30     if ((host = gethostbyname(argv[1])) == NULL)
31     {
32         perror("gethostbyname");
33         exit(1);
34     }
35
36     memset(buf, 0, sizeof(buf));
37     sprintf(buf, "%s", argv[2]);
38
39     /*创建 socket*/
40     if ((sockfd = socket(AF_INET,SOCK_STREAM,0)) == -1)
41     {
42         perror("socket");
43         exit(1);
44     }
45
46     /*设置 sockaddr_in 结构体中相关参数*/
47     serv_addr.sin_family = AF_INET;
48     serv_addr.sin_port = htons(PORT);
49     serv_addr.sin_addr = *((struct in_addr *)host->h_addr);
50     bzero(&(serv_addr.sin_zero), 8);
51
52     /*调用 connect 函数主动发起对服务器端的连接*/
53     if(connect(sockfd,(struct sockaddr *)&serv_addr, sizeof(struct sockaddr))== -1)
54     {
55         perror("connect");
```

```
56         exit(1);
57     }
58
59     /*发送消息给服务器端*/
60     if ((sendbytes = send(sockfd, buf, strlen(buf), 0)) == -1)
61     {
62         perror("send");
63         exit(1);
64     }
65     close(sockfd);
66     exit(0);
67 }
```

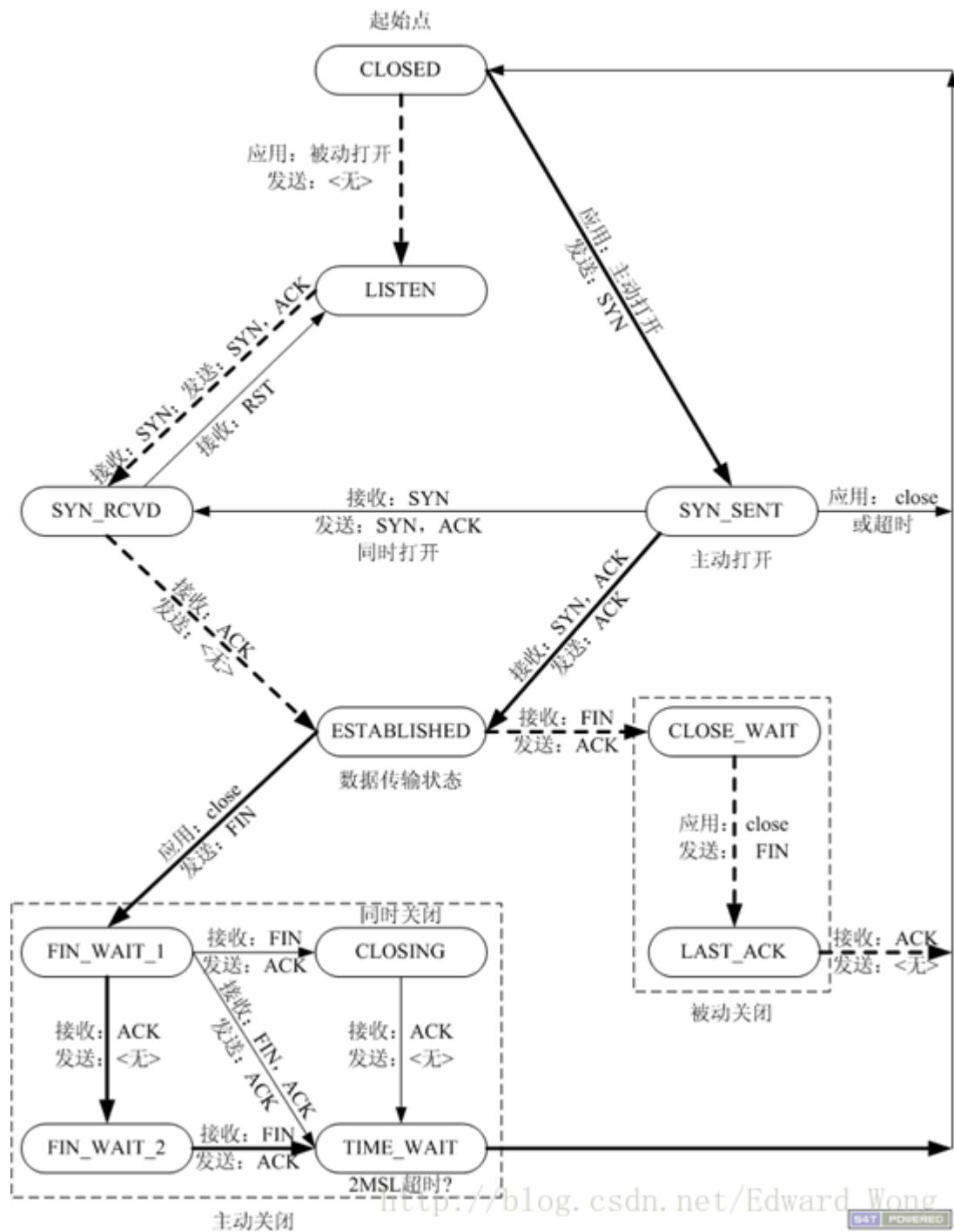
Server.c

```
1 #include <sys/types.h>
2 #include <sys/socket.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <sys/ioctl.h>
7 #include <unistd.h>
8 #include <netinet/in.h>
9
10#define PORT          11910
11#define BUFFER_SIZE    1024
12#define MAX_QUE_CONN_NM 5
13
14int main()
15{
16    struct sockaddr_in server_sockaddr, client_sockaddr;
17    int sin_size, recvbytes;
18    int sockfd, client_fd;
19    char buf[BUFFER_SIZE];
20
21    /*建立 socket 连接*/
22    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
23    {
24        perror("socket");
25        exit(1);
26    }
27    printf("Socket id = %d\n", sockfd);
28
29    /*设置 sockaddr_in 结构体中相关参数*/
30    server_sockaddr.sin_family = AF_INET;
31    server_sockaddr.sin_port = htons(PORT);
32    server_sockaddr.sin_addr.s_addr = INADDR_ANY;
33    bzero(&(server_sockaddr.sin_zero), 8);
34
35    int i = 1; /*使得重复使用本地地址与套接字进行绑定 */
36    setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &i, sizeof(i));
37
38    /*绑定函数 bind*/
39    if (bind(sockfd, (struct sockaddr *)&server_sockaddr, sizeof(struct sockaddr)) == -1)
40    {
41        perror("bind");
42        exit(1);
43    }
44    printf("Bind success!\n");
```

```
45
46     /*调用 listen 函数*/
47     if (listen(sockfd, MAX_QUE_CONN_NM) == -1)
48     {
49         perror("listen");
50         exit(1);
51     }
52     printf("Listening....\n");
53
54     /*调用 accept 函数， 等待客户端的连接*/
55     if ((client_fd = accept(sockfd, (struct sockaddr *)&client_sockaddr, &sin_size)) == -1)
56     {
57         perror("accept");
58         exit(1);
59     }
60
61     /*调用 recv 函数接收客户端的请求*/
62     memset(buf , 0, sizeof(buf));
63     if ((recvbytes = recv(client_fd, buf, BUFFER_SIZE, 0)) == -1)
64     {
65         perror("recv");
66         exit(1);
67     }
68     printf("Received a message: %s\n", buf);
69     close(sockfd);
70     exit(0);
71 }
```

TCP/IP 状态图

TCP 状态转换图



参考文献

- [1] RFC 6349 - Framework for TCP Throughput Testing
- [2] CUBIC: A New TCPFriendly HighSpeed TCP Variant