Linux 内核源码剖析

——TCP/IP 实现

上册

樊东东　莫　澜　编著

机械工业出版社

本书详细论述了 Linux 内核 2.6.20 版本中 TCP/IP 的实现。书中给出了大量的源代码，通过对源代码的详细注释，帮助读者掌握 TCP/IP 的实现。本书根据协议栈层次，从驱动层逐步论述到传输层，包括驱动的实现、接口层的输入输出、IP 层的输入输出以及 IP 选项的处理、邻居子系统、路由、套接口及传输层等内容，全书基本涵盖了网络体系架构全部的知识点。特别是 TCP，包括 TCP 连接的建立和终止、输入与输出，以及拥塞控制的实现。

本书适用于熟悉 Linux 的基本使用方法，对 Linux 内核工作原理以及网络知识有一定的了解，而又极想更深入理解各个机制在 Linux 中的具体实现的用户，包括应用程序员和嵌入式程序员，以及网络管理员等。相关专业的科研人员在工作中遇到问题时，也可以查阅本书，理解相关内核部分的实现。此外，计算机相关专业的本科高年级学生和研究生，在学习相关课程（如操作系统、计算机网络等）时，可将本书作为辅助教程，与理论相结合以便更好地理解相应的知识点。

图书在版编目（CIP）数据

Linux 内核源码剖析：TCP/IP 实现/ 樊东东，莫澜编著．—北京：机械工业出版社，2010.12
ISBN 978-7-111-32373-0

Ⅰ．①L…　Ⅱ．①樊…②莫…　Ⅲ．①Linux 操作系统—机器代码程序—程序分析②计算机网络—通信协议　Ⅳ．①TP316.89②TN915.04

中国版本图书馆 CIP 数据核字（2010）第 212455 号

机械工业出版社（北京市百万庄大街 22 号　邮政编码 100037）
策划编辑：车　忱
责任编辑：车　忱
责任印制：乔　宇
三河市宏达印刷有限公司印刷
2011 年 1 月第 1 版・第 1 次印刷
184mm × 260mm・67.75 印张・1677 千字
0001 – 3000 册
标准书号：ISBN 978-7-111-32373-0
定价：142.00 元（上、下册）

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

电话服务
社服务中心：（010）88361066
销 售 一 部：（010）68326294
销 售 二 部：（010）88379649
读者服务部：（010）68993821

网络服务
门户网：http://www.cmpbook.com
教材网：http://www.cmpedu.com

封面无防伪标均为盗版

前　言

有人宣称 Linux 人才是未来 20 年 IT 职场中的王者。无论这种说法有多夸张，有一个事实是不可否认的，那就是，近年来 Linux 的市场份额不断增长，Linux 正在受到越来越多的关注乃至推崇。由于 Linux 可广泛地应用到各种系统，包括很多嵌入式系统上，以及其他的诸多优点，如开放、高效、丰富的网络功能等，这种趋势在可预计的未来还将持续。

目前，国内对 Linux 各方面的研究工作没有国外那样广泛和深入，相关的出版物水准参差不齐。特别是在网络的实现方面，有些著作针对性不强，有些则缺少了重要的传输层协议实现的论述，还有一些虽然有比较全面的介绍，却不够深入，选用的 Linux 版本也比较旧。

针对以上情况，本书选择了较新的 2.6.20 版本内核 TCP/IP 实现作详细的论述，在重要的细节处甚至逐行分析，并在此基础上，对代码背后的机制和原理作了深入的阐述，将各关键点连成一个整体，帮助读者理清整个 Linux 网络部分的脉络。作者本着严谨的态度，在写作过程中参阅了大量的中英文资料及相关的文档。相信本书可以成为那些希望深入了解 Linux 的 TCP/IP 协议栈、网络部分实现的人们的有力工具。

本书共有 33 章，通过自底向上的方法来论述 TCP/IP 的实现，从数据链路层开始，然后是网络层（IP、ICMP、IGMP、路由以及邻居子系统和 IP 组播），接下来是套接口，最后是传输层（TCP 和 UDP）。在学习的时候也可以采用自顶向下的方法（从传输层开始向下），或者结合以上两种方法。要理解一个系统的运行机制，对于一个专业人员来说，代码是最为直接也最为可靠的资源。Linux 普及面不断扩展，越来越多的人会想通过研读内核代码来了解 Linux 系统，一来是更好地解决具体工作中的相关问题，二来是从 Linux 这个高质量的操作系统中学习到更多的编程、架构等技术。本书的特点如下：

- 选择的内核版本新，内容不会在短期内过时。
- 对代码作了详细的论述，在此基础上进一步分析了代码背后的机制和原理，为读者理清了整个框架的脉络，帮助读者避免迷失在细节中。
- 写作过程中参阅了大量中英文资料和相关的文档，对内核代码更是作了长时间深入细致的研究和分析，在细节处反复推敲，以确保本书的质量。
- 书中有大量的图表，用来帮助读者直观地理解各个数据结构之间的联系及各个函数的调用关系等。

Linux 内核中的网络模块虽然是一个比较独立的模块，但仍会涉及内核中一些常用的技术和算法。因此读者在阅读本书之前需要掌握一些基础知识，对 TCP/IP 应有一定的了解。最好能够深入地阅读过 TCP/IP 相关的 RFC，了解 Linux 内核的运行机制，包括进程管理、内存管理、文件系统、系统调用、netlink、内核中多种锁（自旋锁、读写锁、RCU）等。当然在学习过程中，免不了要做些实验，因此也需要熟悉 Linux 提供的多种工具，包

括 ip、ifconfig 等命令。

本书使用的 Linux 内核版本为 2.6.20。也许读者想知道为什么选择这个版本。实际上在作者开始分析 TCP/IP 代码时，该版本是最新版本。只是因为 Linux 内核的更新很快，经过两年多的分析后，最新版本已经是 2.6.3x 了。即便如此，如果读者能够理解 2.6.20 版本的代码，则再读最新版本的代码时，应该不会太难。

由于本书内容繁多，漏洞和不足之处难免，请读者谅解并提出修改建议。作者的博客是 http://blog.chinaunix.net/u3/112243/。

在此，还要特别感谢网友衫秋南对本书提出宝贵的意见和建议。

上 册 目 录

第1章 预备知识

随着 Linux 的蓬勃发展和普及，想深入了解内核实现的技术人员也越来越多。而要真正深入了解内核，就需要阅读和分析内核源码。

Linux 内核源码可以从包括网络在内的很多途径得到，例如从 http://www.kernel.org 下载。当然，在很多发行版中，/usr/src/linux 目录也存放内核源码。目前最新的稳定版是 2.6.36.2。

很多技术人员畏惧阅读 Linux 内核源码，因为这样庞大而复杂的系统代码，阅读起来确实有很多困难。由于操作系统由多个模块组成，包括进程管理、内存管理、文件系统、驱动程序、网络等，而 Linux 内核也采用了面向对象技术进行设计和编码，因此实际上阅读源码也没有想象的那么高不可攀。只要有恒心，不必担心水平不够，困难都是可以克服的，很多知识可以边阅读源码边学习。

做任何事情都需要有方法和工具，同样，阅读 Linux 内核源码也是如此。由于内核源码非常庞大，因此不能全面铺开，而是要按照模块一个一个去攻克，本书的目的就是指导和帮助读者学习网络模块。要想比较顺利地阅读内核网络源码，事先最好对源码的知识背景有一定的了解。对于内核网络源码来讲，基本要求是：熟悉 C 语言，最好了解 GNU 对标准 C 的扩展；熟悉 GCC 编译器以及使用方法；熟悉操作系统的基本知识；熟悉 Linux 内核通用技术，包括内存管理、下半部、锁等；熟悉 TCP/IP 的原理。

本书讲述的代码来自 Linux-2.6.20，下载网址是 http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.20.tar.bz2。

俗话说：“工欲善其事，必先利其器”。像 Linux 内核源码这样的复杂程序，阅读时遇到的问题肯定会像一个越滚越大的雪球。例如一个函数经常进行多层次的调用，一个结构会有多个结构多层次的嵌套，而且会涉及多个其他的相关文件，对代码进行多层次跟踪后，说不定一下子还回不到原先阅读的函数或结构。因此需要一个好的阅读工具。

由于大部分技术人员和爱好者都比较熟悉 Windows 或 Linux 平台，所以在此介绍两个工具。

一个是 Windows 下的 Source Insight，它是一个共享软件，有 30 天免费期，可以从 http://www.sourcedyn.com 下载。安装和使用这个软件都非常简单，安装完成并运行后，先选择“Project”菜单下的“new”命令，新建一个工程，输入工程名，接着把欲读的源码加入（可以添加整个目录）后，软件就会分析所加的源码。分析完后，就可以进行阅读了。对于打开的阅读文件，如果想看某一变量的定义，可先把光标定位于该变量，然后点击工具条上的相应选项，该变量的定义就会显示出来，对于函数的定义与实现也可以同样操作。

另一个是 Windows 和 Linux 下的 Source-Navigator，可以从 http://sourceforge.net/projects/sourcenav/files/下载。它是 Red Hat 开发的一个 IDE，很适合用来阅读源代码——因为它能很好地解决文件定位和跳转问题，功能与 Source Insight 类似。运行 Source-Navigator 之后，该软件会建立源代码的 project 并扫描源代码文件，自动建立文件之间的索引，之后便可以进行阅读了。

读者也可在线阅读源代码，网址是 http://lxr.linux.no/#linux+v2.6.20/。

1.1 应用层配置诊断工具

在学习 Linux 网络模块实现的过程中，免不了要做些实验，来验证自己的想法或查看代码流程。Linux 提供了多种配置工具用于配置内核以及网络特性。

1.1.1 iputils

除了经常使用的 ping 外，iputils 包还包含 arping（用来生成 arp 请求），网络路由发现服务器 rdisc，以及其他一些程序（tracepath、clockdiff、tftpd 和 rarpd）。

1.1.2 net-tools

net-tools 网络工具包，最常用的就是 ifconfig，route，netstat 和 arp，还包括 hostname，nameif，plipconfig，rarp，slattach，ipmaddr，iptunnel 和 mii-tool 等工具。

1.1.3 iproute2

iproute2 是 Linux 特有的、新一代的网络配置工具包。其中 ip 是最常用的命令，它可以用来配置网络设备（ip link）、IP 地址（ip address）、路由（ip route）以及其他功能。tc 也是 IPROUTE2 工具包中常用的、用来配置流量控制的工具。除了以上两个工具，还包括 ss，nstat，ifstat，rtacct，arpd，lnstat，maketable，normal，pareto 和 paretonormal。

1.2 内核空间与用户空间的接口

内核提供接口给用户空间程序，以便用户进程进行信息的读取或配置。除了经典的系统调用接口，Linux 还可通过几个特殊的接口，包括虚拟文件系统和 netlink 套接口，获取相关信息。

procfs 和 sysctl 都可以导出内核内部信息，但是 procfs 主要用于导出只读数据，而 sysctl 导出的信息是可写的（只有超级用户可写）。如果导出的是只读数据，那么选择 procfs 还是 sysctl 就与导出数据的数量有关。如果导出的是简单变量，那么应该使用 sysctl。反之，如果导出大量复杂的数据，并且要求导出的数据有固定格式，就应该使用 procfs。

1.2.1 procfs

procfs 是一个虚拟文件系统，通常挂接在/proc 目录下，内核通过文件的形式将内部信息展现给用户空间程序。这些文件都不是磁盘文件，但是可以使用 cat 命令读取它们，或者通过重定向符号（>）写它们，甚至可以像普通文件一样设定它们的读写权限。

网络代码注册的文件一般位于/proc/net 目录下，在该目录下存在不少文件，有一种比较特殊的文件，比如 tcp、udp 等。这种文件的格式比较固定，Linux 称之为综合文件（synthetic files），它们最大的特点就是由一系列记录组成，类似于数据库表中的一条条记录。用这种格式来描述系统中的一些统计或状态是比较适合的，因此proc 文件系统特地增加了对这种文件类型的支持。因为它只存在于 proc 文件系统中，文件系统提供了对此文件框架的支持，而在使用此类型文件时，只需关心数据的处理即可。

大多数网络功能在初始化时，都会在/proc/net 中注册一个或多个这样的文件，无论初始化

动作发生在系统启动时还是模块加载时。当用户读取某个文件时，内核会调用一组内核函数来输出相应的信息。

创建和删除文件可以分别调用 proc_net_fops_create()和 proc_net_remove()，这两个函数分别包装了函数 create_proc_entry()和 remove_proc_enty()。需要注意的是 proc_net_fops_create()在创建文件的同时也初始化了这个文件的操作函数。来看下面的例子。

这是 IP 组播在/proc/net 目录下注册 ip_mr_vif 文件的例子：

```
1717 static struct file_operations ipmr_vif_fops = {
1718     .owner    = THIS_MODULE,
1719     .open     = ipmr_vif_open,
1720     .read     = seq_read,
1721     .llseek   = seq_lseek,
1722     .release = seq_release_private,
1723 };

1899 void __init ip_mr_init(void)
1900 {
....
1908 #ifdef CONFIG_PROC_FS
1909     proc_net_fops_create("ip_mr_vif", 0, &ipmr_vif_fops);
....
1911 #endif
1912 }
```

proc_net_fops_create 的三个参数所代表的含义是：文件名是 ip_mr_vif，文件属性是只读，文件的操作函数句柄是 ipmr_vif_fops。当用户读文件时，file_operations 结构里面的函数会以数据块的方式将数据返回给用户。这在读取大量相同类型数据的时候很有用。例如，可以一次读取组播虚拟接口。

ipmr_vif_open()会进行一系列初始化：它会注册一组函数指针，这些函数指针在 procfs 的例程中被调用，用于遍历返回给用户的数据：一个函数用于初始化打印动作，另一个用于向前移动一次指针，还有一个用于打印一个数据项。这些函数会在内部保存必要的上下文信息以便了解当前的打印项在哪里，以及可以从哪个地方重新开始打印等。

```
1686 static struct seq_operations ipmr_vif_seq_ops = {
1687     .start = ipmr_vif_seq_start,
1688     .next  = ipmr_vif_seq_next,
1689     .stop  = ipmr_vif_seq_stop,
1690     .show  = ipmr_vif_seq_show,
1691 };

1693 static int ipmr_vif_open(struct inode *inode, struct file *file)
1694 {
....
1702     rc = seq_open(file, &ipmr_vif_seq_ops);
1703     if (rc)
1704         goto out_kfree;
....
1715 }
```

1.2.2 sysctl(/proc/sys 目录)

sysctl 接口允许用户读取或者修改内核参数。在用户空间可以通过两种方法访问 sysctl 导出的变量。一种方法是通过 sysctl 系统调用，另一种是通过 procfs。如果内核支持 procfs，它会在/proc 目录下增加一个特殊的目录（/proc/sys），这个目录里包含了 sysctl 导出变量的列表。

procps 包里的 sysctl 命令可以用于配置 sysctl 接口导出的内核参数，实际上这个命令是通过/prco/sys 目录下的文件与内核通信的。大多数 Linux 发行版的内核都默认包含了 sysctl 的支持。

通过 sysctl 系统调用和命令来设置和获得运行时内核的配置参数是一种有效的方式，通过这种方式，用户可以在内核运行的任何时刻修改和获取内核的配置参数。例如，通过 cat /proc/sys/net/ipv4/ip_forward 来获取内核网络层是否允许转发 IP 数据报。通过 echo 1 > /proc/sys/net/ipv4/ip_forward 将内核网络层设置为允许转发 IP 数据报。通过 cat /proc/sys/kernel/ostype 可以获取操作系统的类型，其实也可以通过 sysctl 系统调用达到同样的效果。

```
#include <unistd.h>
#include <sys/syscall.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <linux/sysctl.h>

int _sysctl(struct __sysctl_args *args );

#define OSNAMESZ 100

int main(void)
{
    struct __sysctl_args args;
    char osname[OSNAMESZ];
    size_t osnamelth;
    int name[] = { CTL_KERN, KERN_OSTYPE };

    memset(&args, 0, sizeof(struct __sysctl_args));
    args.name = name;
    args.nlen = sizeof(name)/sizeof(name[0]);
    args.oldval = osname;
    args.oldlenp = &osnamelth;

    osnamelth = sizeof(osname);

    if (syscall(SYS__sysctl, &args) == -1) {
        perror("_sysctl");
        exit(EXIT_FAILURE);
    }
    printf("This machine is running %*s\n", osnamelth, osname);
    exit(EXIT_SUCCESS);
}
```

图 1-1 显示了以 root_table 为顶点形成的树状结构。

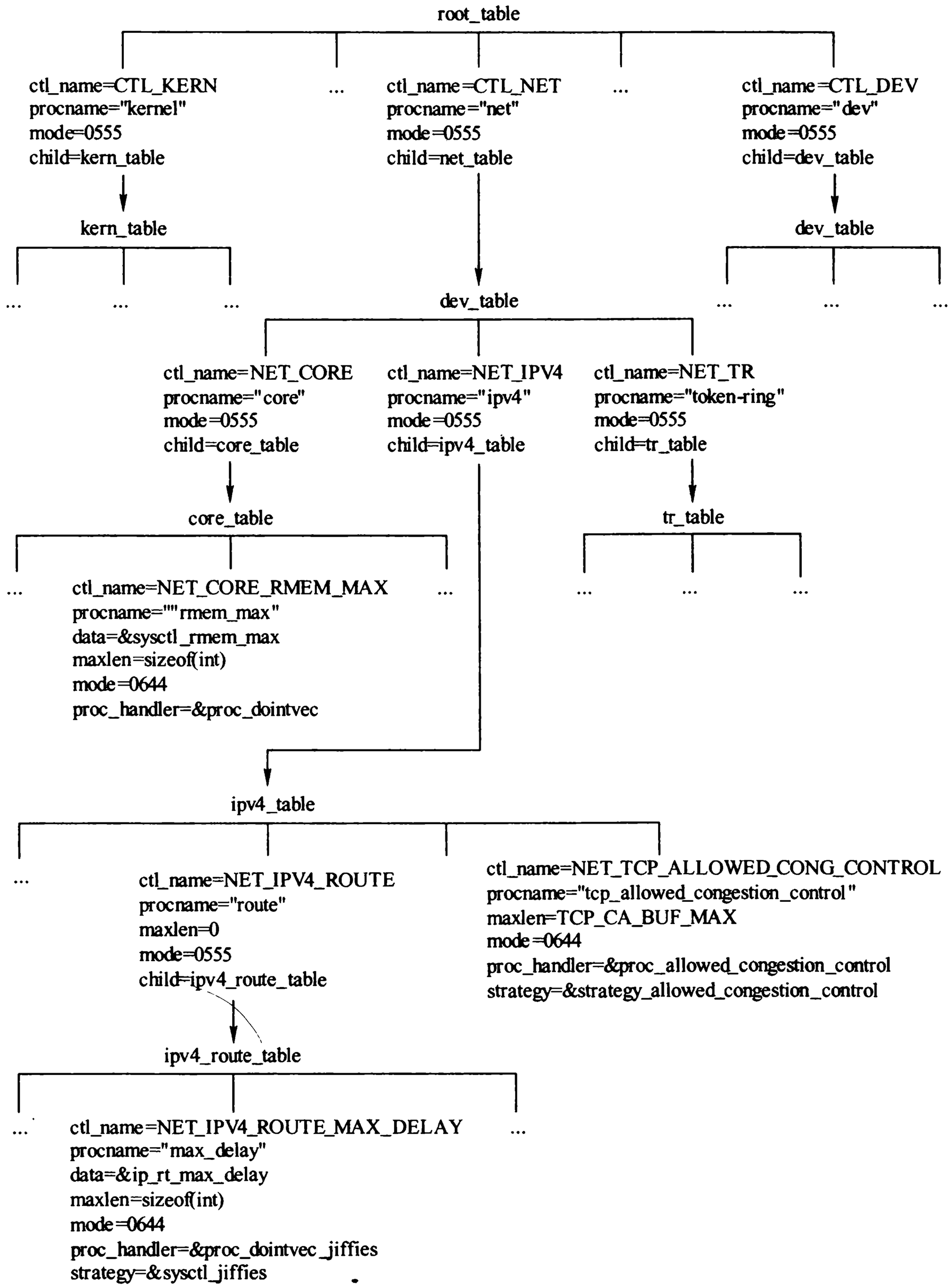

图 1-1 sysctl 的组织

1.2.3 sysfs(/sys 文件系统)

sysfs 是 Linux 2.6 提供的一种虚拟文件系统，这个文件系统不仅可以把设备和驱动程序的信息从内核空间导出到用户空间，也可以对设备和驱动进行配置。

sysfs 的目的是把一些原本在 procfs 中的设备独立出来，以设备树的形式呈现给用户。sysfs 最初名称为“driverfs”，当初只是为了要对新的驱动程序模型除错而开发出来的，然而随着代码的更新，新的“驱动程序模型”和“driverfs”证明了对内核中的其他子系统也有用处。Kobjects 开发出来之后，用于核心部件的管理机制，因此 driverfs 也被更名为 sysfs。

1.2.4 ioctl 系统调用

ioctl 系统调用可以操作一个文件，它通常用于实现特殊设备的操作，而这些操作在标准的文件系统调用中没有提供。ioctl 也可以操作套接口描述符，net-tools 工具包就是通过 ioctl 系统调用与内核交互的。

1.2.5 netlink 套接口

netlink 套接口是网络应用程序与内核通信最新、最常用的接口，IPROUTE2 包中的大多数命令都使用这个接口。netlink 套接口的描述在 RFC 3549 中。

netlink 套接口使用起来非常简单，通过套接口标准的 API 来打开、关闭，或者发送和接收信息。要在用户态创建一个 netlink 套接口，代码如下所示：

```
socket(PF_NETLINK, SOCK_DGRAM, NETLINK_ROUTE)
```

在内核中定义了多种协议，每一种都被协议栈中的一个或一组内核组件使用。例如上述代码中的 NETLINK_ROUTE 协议就会被多个模块使用，如路由和邻居协议，而 NETLINK_FIREWALL 用于防火墙模块。

1.3 网络 I/O 加速

尽管技术有了巨大的进步，但是 TCP/IP 协议栈的处理方式却几乎没有变化。也就是说，即便用户使用最先进的 CPU，依然要处理那些未经优化的 TCP/IP 协议，由此产生巨大的系统开销。例如，TCP/IP 的传输过程中需要封装、解包，这些动作对于处理器而言并不是一个复杂的过程，但是会占用处理器周期，而且网络带宽越高，这个问题越严重。系统开销的增大不仅仅表现在占用较多的处理器周期，还会导致处理网络相关数据时的内存访问效率降低。这又会进一步降低 CPU 效能和网络效率。

过去，网络流量较低，处理网络相关数据所产生的开销，远远低于用于执行正常任务的开销，所以并未引起重视。现在，随着网络流量大幅度提升，处理网络相关数据所产生的系统开销越来越不能忽视，甚至已经影响到了正常应用。现有几种解决方案：

（1）TSO（TCP Segmentation Offload）

通过网络设备上的专用处理器处理部分或者全部的封包，借此来降低对于系统处理器资源的占用，不过这种解决方案只对具有某些特征的数据包有效。

（2）RDMA（Remote Direct Memory Access，远程直接内存访问）

发送端系统直接将有效数据送至目的系统指定的内存中，无需移动数据包的时间消耗，因此大大提升了网络传输的效率。但是这种技术需要专用的网络设备，应用程序也需要进行修改，甚至还增加了一个 RDMA 层的封装过程，而且这种操作风险较高，因此目前看来还不是一个吸引人的解决方案。

（3）Onloading 技术

将系统处理器作为处理网络流量的第一引擎，尽可能地提升 CPU 处理网络数据包的效率，这种思想已经被英特尔借鉴。

目前，Linux 已经支持 TSO 和英特尔 I/O 加速技术，对于网络设备的 DMA 本书不作论述。要支持英特尔 I/O 加速选项 CONFIG_INTEL_IOATDMA，必须在编译前选择支持“Intel I/OAT

DMA support”，该选项的位置是“Device Drivers-> DMA Engine support-> Support for DMA engines-> Intel I/OAT DMA support”，见图 1-2。

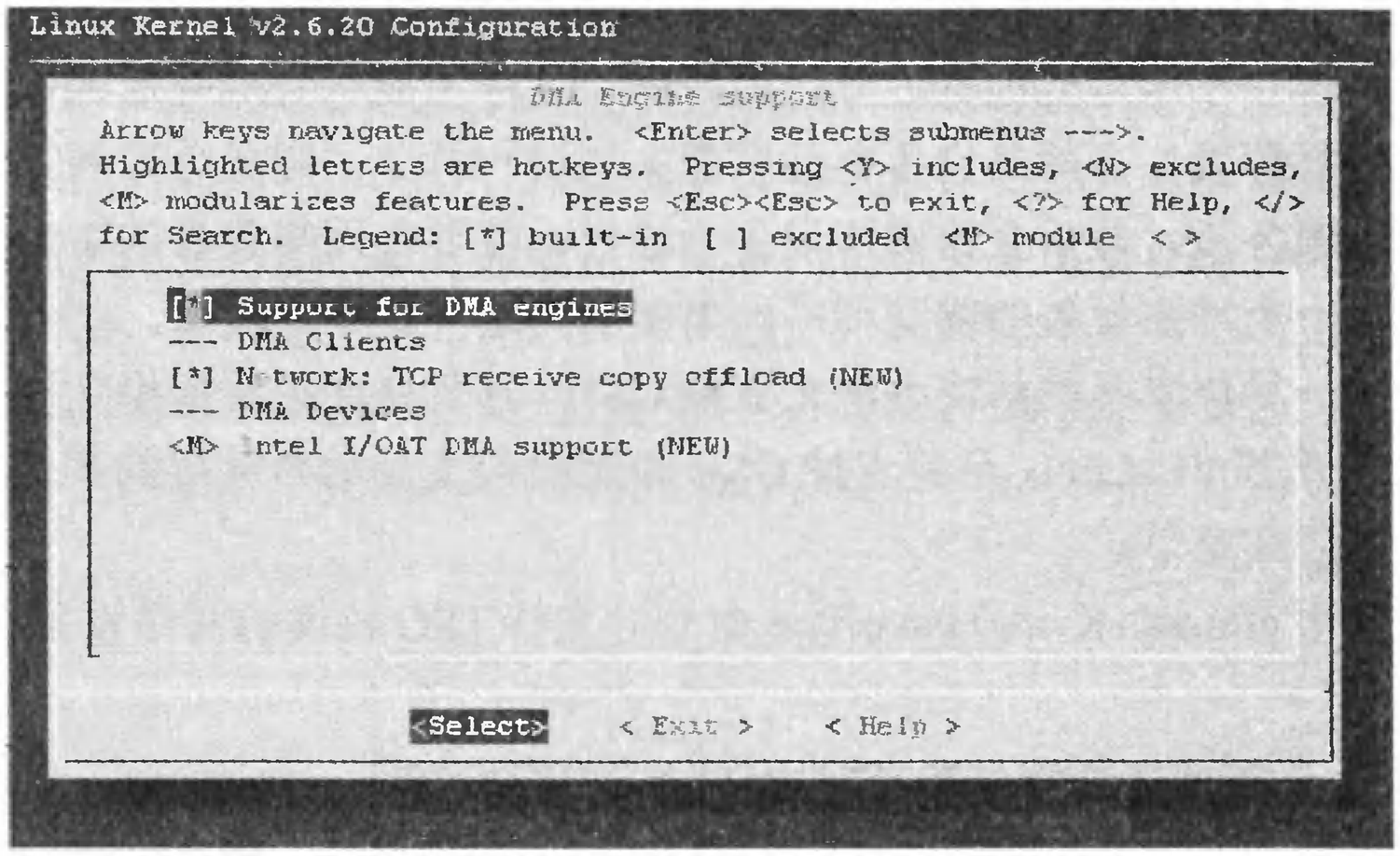

图 1-2 启用 CONFIG_INTEL_IOATDMA

而要启用 TCP receive copy offload 特性 CONFIG_NET_DMA，需要在编译前选择支持“Network: TCP receive copy offload”。该选项位于“Device Drivers-> DMA Engine support-> Support for DMA engines-> Network: TCP receive copy offload”。

1.3.1 TSO/GSO

TSO 是通过网络设备进行 TCP 段的分割，从而来提高网络性能的一种技术。较大的数据包（超过标准 1518B 的帧）可以使用该技术，使操作系统减少必须处理的数据数量以提高性能。通常，当请求大量数据时，TCP 发送方必须将数据拆分为 MSS 大小的数据块，然后进一步将其封装为数据包形式，以便最终可以在网络中进行传输。而当启用了 TSO 技术之后，TCP 发送方可以将数据拆分为 MSS 整数倍大小的数据块，然后将大块数据的分段直接交给网络设备处理，操作系统需要创建并传输的数据包数量更少，因此性能会有较大的提高。图 1-3 所示为标准帧和 TSO 技术特性比较。

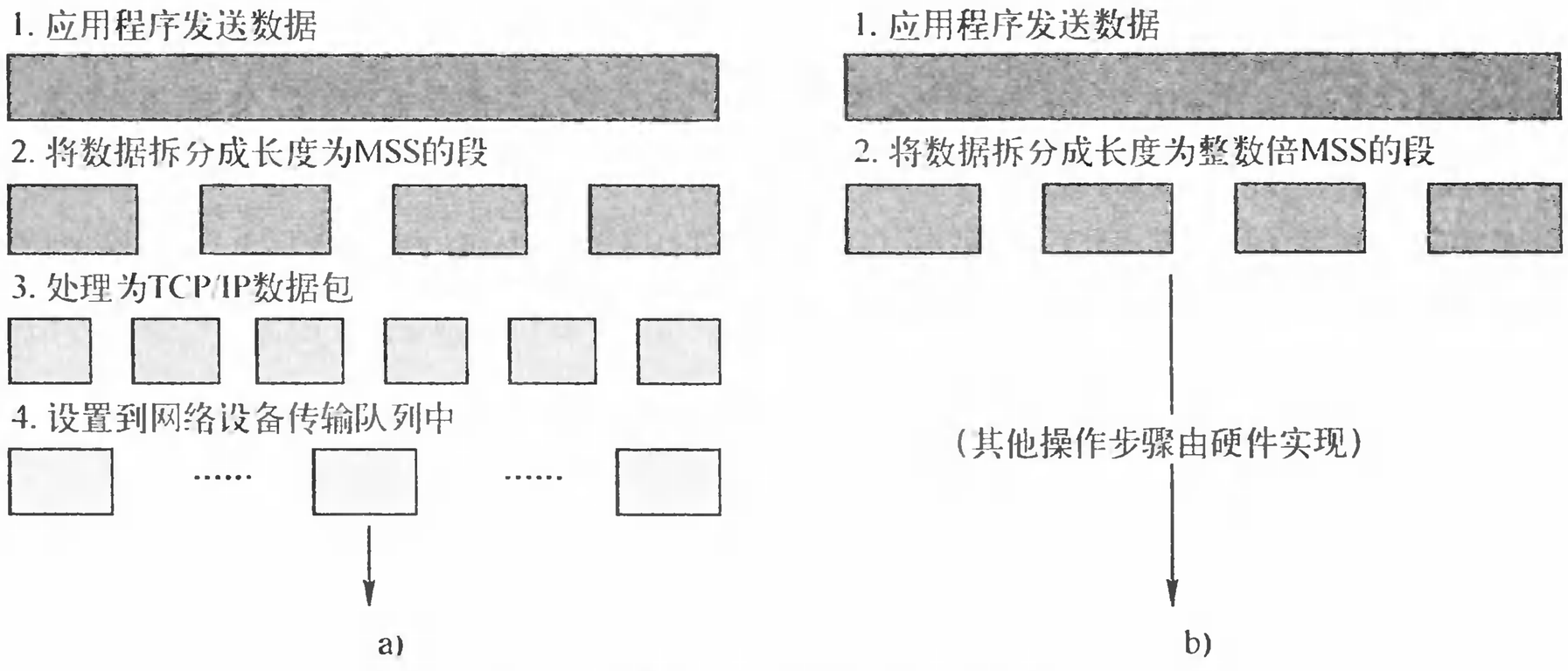

图 1-3 标准帧和 TSO 的处理过程

a) 不支持 TSO b) 启用 TSO 后

从前面有关 TSO 的论述可以看出，TSO 只是针对 TCP 协议的，使 TCP 协议在硬件上得到了有力的支持。事实上，这种概念也可以应用于其他的传输层协议，如 TCPv6，UDP，甚至 DCCP 等，这就是 GSO(Generic Segmentation Offload)。

性能提高的关键在于尽可能地推迟分段的时机，这样才能有效地降低成本。最理想的是在网络设备驱动里进行分段，在网络设备驱动里把大包进行拆分，组成分段列表，或在一块预先分配好的内存中重组各段，然后交给网络设备。这样，就要在网络设备的驱动里边来实现它，那么就需要修改每一个网络设备的驱动程序。事实上，这样做不大现实。

然而似乎有另一种更容易的解决办法来支持 GSO，那就是在把数据报文提交给网络设备驱动之前进行聚合/分散操作。Linux 目前支持 GSO 框架已经支持的传输层的其他协议。有关 GSO 方面的代码，参见后续章节。

应用层可以使用 ethtool -K eth0 tso off|on 命令对支持 TSO 特性的网络设备进行 TSO 功能的关闭和启用。

1.3.2 I/O AT

英特尔 I/O 加速技术（I/O AT）是一个整合于系统平台中的用于移动数据、访问数据和降低网络 I/O 过程中系统开销的 I/O 解决方案。I/O AT 可以帮助服务器应用程序更快、更高效地（占用更少的 CPU 周期）获取数据。升级到支持 I/O AT 的平台后，用户原来的应用程序可以立刻从中受益（比如可以同时访问更多的数据、处理更多用户的请求）而无需进行任何修改。

I/O AT 主要在三个方面解决网络 I/O 瓶颈：

- 降低系统开销。
- 实现流线型内存访问。
- 优化 TCP/IP 协议计算。

I/O AT 通过适度的中断、适度的内存访问、并行计算、数据移动和减少上下文切换等措施来降低系统开销。比如通过预取操作来提升内存访问和数据移动效率，直接访问子系统可用于卸载数据移动和异步拷贝，TSO 运算转移给网络设备或者板载 LAN 控制器(LOM)，并且在数据流和特定的处理器核心之间建立密切的联系。这些技术可以降低网络 I/O 过程中的 CPU 占用率，包括协议计算，并且明显地降低由于缓存未命中和 cache line bouncing 所引起的 CPU 停滞。

I/O AT 涉及服务器系统的以下几个方面：

（1）Network Flow Affinity：分割网络堆栈处理并且动态地分配到多个物理或者逻辑 CPU，这种方式可以使数据得到更快的处理。

（2）异步低成本拷贝：采用了增强型 DMA，可以用更少的 CPU 时钟从系统内存中的网络设备缓冲区复制有效数据到应用程序缓冲区，节约下来的 CPU 资源可以进一步提升应用程序的负载能力。

（3）通过优化 TCP/IP 栈改进 TCP/IP 协议：主要优化分离封包头部和有效数据的处理。结合与栈相关的改进可以降低处理协议的时钟数量。

1.4 其他

虽然网络模块比较独立，但是也使用了操作系统中一些通用的技术，比如内存管理和锁等。

下面简单地介绍 slab 分配器和 RCU，细节请参见相关资料。

1.4.1 slab 分配器

Linux 所使用的slab分配器的基础是Jeff Bonwick为SunOS操作系统首次引入的一种算法。与传统的内存管理模式相比，slab 缓存分配器有很多优点。首先，内核通常依赖于对小对象的分配，它们会在系统生命周期内进行无数次分配。slab 缓存分配器通过对类似大小的对象进行缓存而提供这种功能，从而避免了常见的碎片问题。slab 分配器还支持通用对象的初始化，从而避免了为同一目的而对一个对象重复进行初始化。最后，slab 分配器还可以支持硬件缓存对齐和着色，这允许不同缓存中的对象占用相同的缓存行，从而提高缓存的利用率并获得更好的性能。

1.4.2 RCU

RCU(Read-Copy Update)，顾名思义就是读-拷贝更新，它是基于其原理命名的。对于被RCU 保护的共享数据结构，“读者”不需要获得任何锁就可以访问它，但“写者”在访问它时将首先拷贝一个副本，然后对副本进行修改，最后使用一个回调机制在适当的时机把指向原来数据的指针重新指向新的被修改的数据。这个时机就是所有引用该数据的 CPU 都退出对共享数据的操作。

RCU 实际上是一种改进的 rwlock，“读者”不需要锁，不使用原子指令，几乎没有什么同步开销，而且在除 alpha 的所有架构上也不需要内存栅（Memory Barrier），因此不会导致锁竞争。而“写者”的同步开销比较大，它需要延迟数据结构的释放，复制被修改的数据结构。它还必须使用某种锁机制同步并行的其他“写者”的修改操作。“读者”必须提供一个信号给“写者”以便“写者”能够确定数据可以被安全地释放或修改的时机。有一个专门的垃圾收集器来探测“读者”的信号，一旦所有的“读者”都已经发送信号告知它们都未使用被 RCU 保护的数据结构，垃圾收集器就调用回调函数完成最后的数据释放或修改操作。RCU 与 rwlock 的不同之处是：它既允许多个“读者”同时访问被保护的数据，又允许多个“读者”和多个“写者”同时访问被保护的数据，“读者”没有任何同步开销，而“写者”的同步开销则取决于使用的“写者”间同步机制。

第 2 章　网络体系结构概述

2.1　引言

Linux 操作系统的特性之一就是它的网络协议栈。其最初实现源于 BSD 的网络协议栈，组织得非常好，在协议无关层（例如：通用套接口层、设备层）与各种具体网络协议层之间有着一整套统一而干净的接口。本书将从分层角度对 Linux 网络协议栈进行论述。

2.2　协议简介

虽然对网络的正式介绍一般都会参考 OSI（Open Systems Interconnection）模型，但由于 Internet 协议本身的模型只有四层，因此本书对 Linux 网络协议栈以如图 2-1 所示的四层进行介绍。

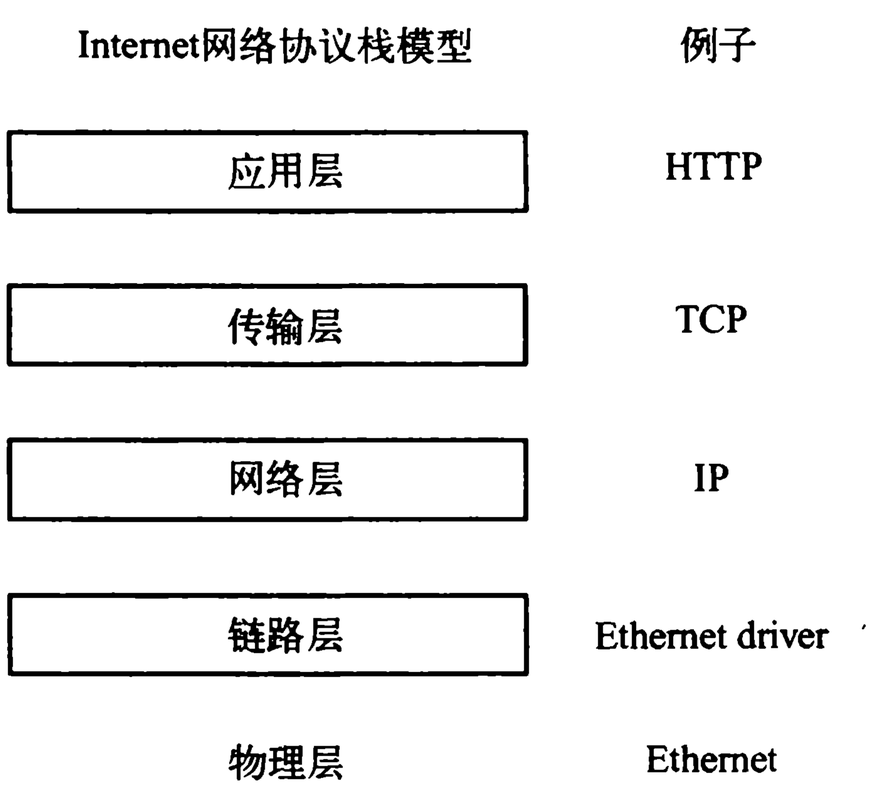

图 2-1　Internet 网络协议栈模型

协议栈的底部是链路层，链路层提供对物理层访问的设备驱动程序，物理层一般就是各种介质，例如串口链路或以太网设备。链路层上面是网络层，负责接收、发送或转发数据报。网络层的上一层为传输层，负责数据传输和数据控制，提供端到端数据交换机制。最上层是应用层，它通常是一个语义层，能够理解要传输的数据，例如，超文本传输协议（HTTP）就负责传输服务器和客户机之间对 Web 内容的请求与响应。

在实际应用中，链路层最常用的一种高速介质就是以太网了。当然链路层还包括一些串口协议，例如 SLIP（Serial Line Internet Protocol）、CSLIP（Compressed SLIP）和 PPP（Point-to-Point Protocol）。最常见的网络层协议是 IP（Internet Protocol），此外为了辅助网络层，还提供了一些满足其他需求的协议，比如 ICMP（Internet Control Message Protocol）和 ARP（Address Resolution Protocol）。传输层协议通常是 TCP（Transmission Control Protocol）和 UDP（User Datagram Protocol）。而应用层中则包含一些常见的协议，如标准的 Web 协议 HTTP，电子邮件协议 SMTP（Simple Mail Transfer Protocol）以及 FTP 等。

2.3　网络架构

图 2-2 显示了 Linux 网络协议栈的架构以及如何实现 Internet 模型。在最上面是用户空间中实现的应用层，而中间为内核空间中实现的网络子系统，底部为物理设备，提供了对网络的连接能力。其中中间部分正是本书的重点所在，在网络协议栈内部流动的是套接口缓冲区（SKB），用于在协议栈的底层、上层以及应用层之间传递报文数据。

网络协议栈顶部是系统调用接口，为用户空间中的应用程序提供一种访问内核网络子系统的接口。下面是一个协议无关层，它提供了一种通用方法来使用传输层协议。然后是传输层的具体协议，包括 TCP、UDP。在传输层下面是网络层。然后是邻居子系统，在邻居子系统存在的目标才是当前可以直接访问的。再下面是网络设备接口，提供了与各个设备驱动程序通信的通用接口。最底层是设备驱动程序。

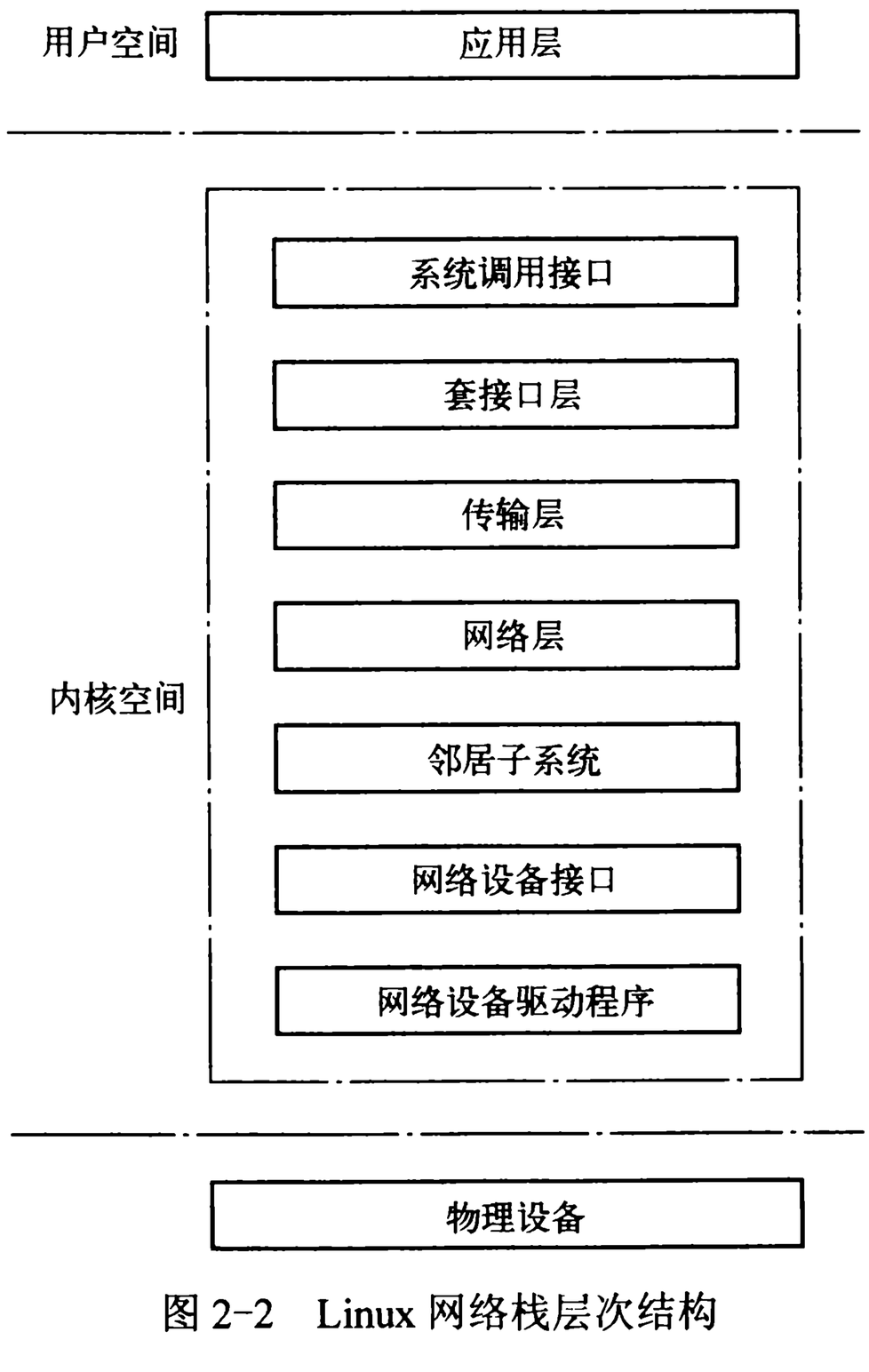

图 2-2　Linux 网络栈层次结构

2.4　系统调用接口

网络子系统提供了两种调用接口给用户进程。用户进程在进行网络调用时，通过系统特有的网络调用接口进入内核。在内核中，进一步调用 sys_socketcall()结束该过程，在 sys_socketcall()中会根据网络系统调用号调用具体的功能。

另一种系统调用接口是通过普通文件操作来访问网络子系统。虽然有很多操作是网络专用的，比如使用 socket 系统调用创建一个套接口，使用 connect 系统调用连接一个服务器等，但套接口的输入/输出操作可以被当成典型的文件读写操作来进行。

2.5 协议无关接口

通过网络协议栈通信都需要对套接口进行操作。套接口层是一个与协议无关的接口，它提供了一组接口来支持各种协议。套接口层不但可以支持典型的 TCP 和 UDP 协议，还可以支持 RAW 套接口、RAW 以太网和其他传输协议，如 SCTP（Stream Control Transmission Protocol）。

Linux 中用 socket 结构描述套接口，代表一条通信链路的一端，用来存储与该链路有关的所有信息。这些信息包括所使用的协议、协议的状态信息（包括源和目的地址）、到达的连接队列、数据缓存和可选标志等。其中最关键的成员是 sk 和 ops，前者指向与该套接口相关的传输控制块，后者指向特定传输协议的操作集，见图 2-3。

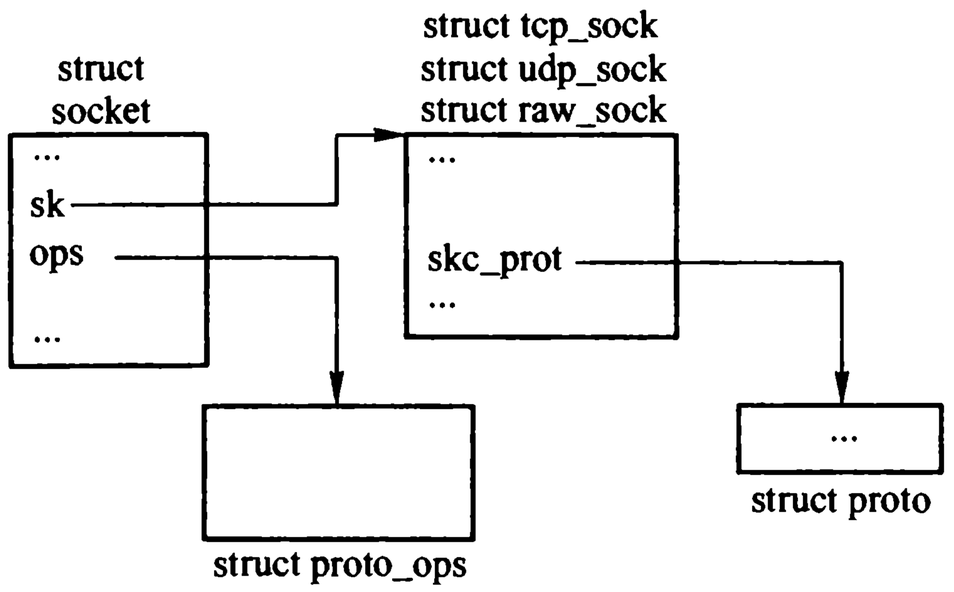

图 2-3 socket 结构与传输控制块的关系

2.6 传输层协议

如图 2-4 所示，套接口的 sk 字段指向与该套接口相关的传输控制块，传输层使用传输控制块存放套接口所需的信息。传输控制块按协议而异，TCP 传输控制块、UDP 传输控制块、RAW 传输控制块，分别对应 tcp_sock 结构（支持完整的 TCP 特性，包含了 TCP 为各连接维护的所有结点信息：两个方向的序号、窗口大小、重传次数等），udp_sock 结构（支持完整的 UDP 特性，包括两端 IP 地址、两端端口号、本端 IP 选项等）和 raw_sock 结构。

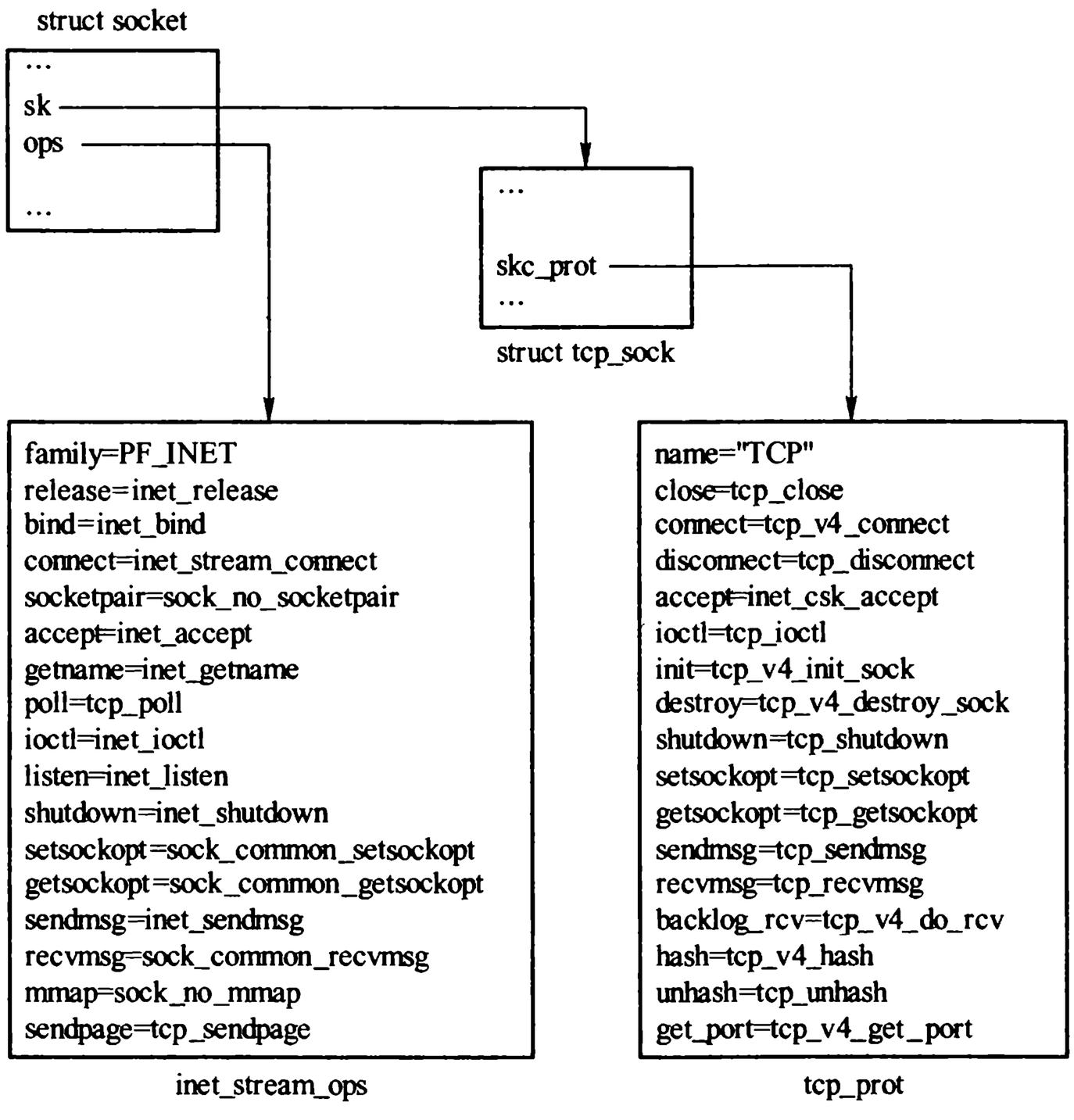

图 2-4 stream 类型套接口与传输控制块以及接口之间的关系

在 Linux 内核中定义传输控制块是非常巧妙的，它根据协议族和传输层协议的特点，分层次地定义了多个结构，以 TCP 为例，tcp_sock 结构是在 inet_sock 结构基础上构成的，而 inet_sock 结构则又是在 sock 结构基础上构成的。

套接口的 ops 字段则指向特定传输协议的操作集接口。proto_ops 结构中定义的接口函数是从套接口系统调用到传输层调用的入口，因此其成员与 socket 系统调用基本上是一一对应的。整个 proto_ops 结构就是一张套接口系统调用的跳转表，其中的某些操作会继续调用 proto 结构跳转表中的函数，从而进入具体的传输层或网络层的处理。TCP、UDP、RAW 套接口的传输层操作集分别是 inet_stream_ops、inet_dgram_ops、inet_sockraw_ops。

proto 结构中的操作实现从传输层到各具体传输层操作和网络层调用的跳转。proto 结构中的一部分成员与 proto_ops 结构中的成员对应，如 connect 等，在此暂且称之为传输层接口；而另外一部分对相关数据操作的函数调用，如 sendmsg，在通过此接口后进入网络层的处理，对 Internet 协议族而言即进入 IP 层。

2.7　套接口缓存

如图 2-5 所示，网络子系统中用来存储数据的缓冲区叫做套接口缓存，简称为 SKB。该缓存区能够处理可变长数据，即能够很容易地在数据区头尾部添加和移除数据，且尽量避免数据的复制。

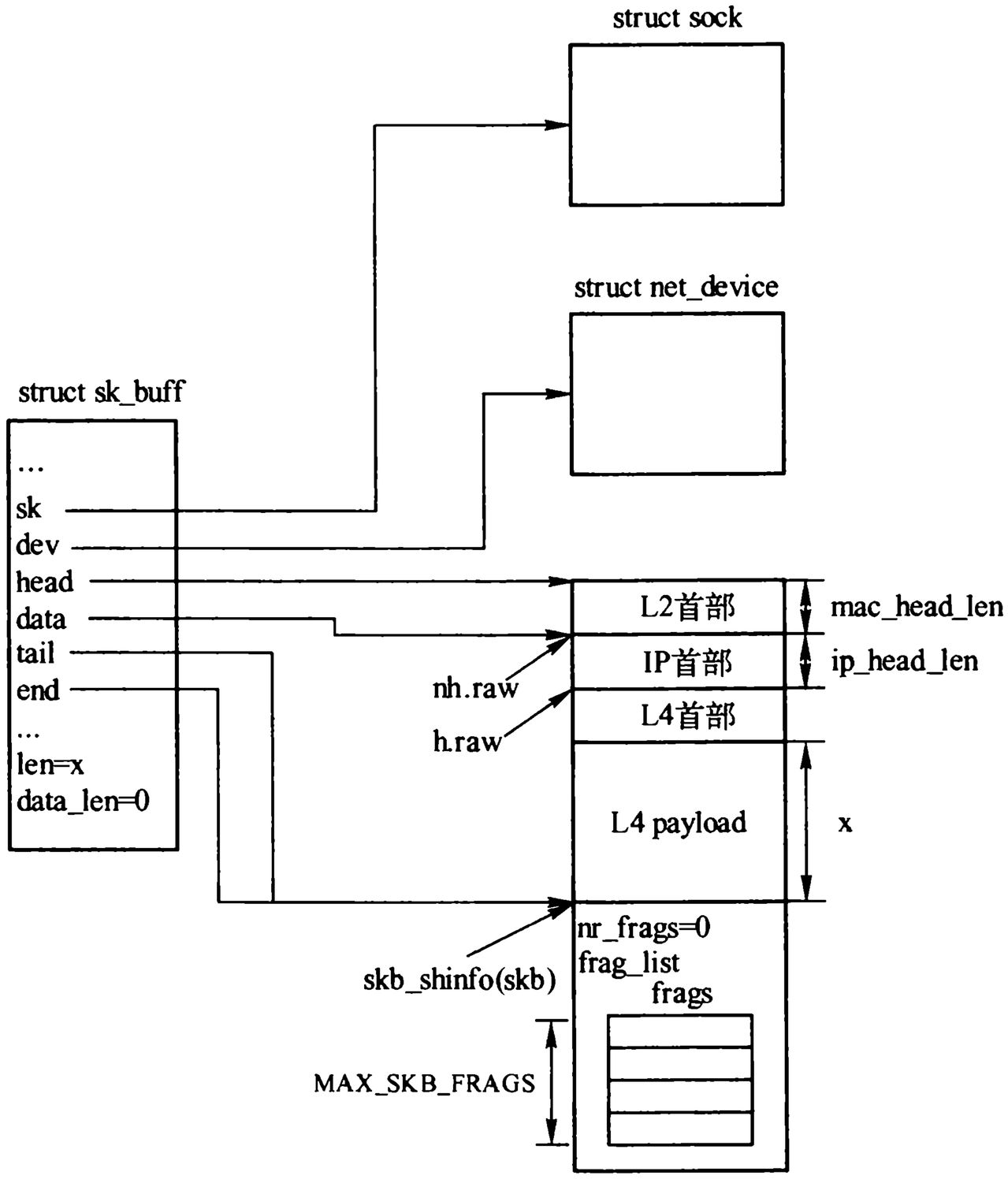

图 2-5　套接口缓冲区结构及其与其他结构的关系

每一个 SKB 都在设备结构中标识发送报文的目的地或接收报文的来源地。通常每个报文使

用一个 SKB 表示，各协议层报文头通过一组指针（th、iph 和 mac）进行定位。由于 SKB 是网络子系统中的数据管理核心，因此有很多管理函数是对它们进行操作的。

SKB 主要用于在网络驱动程序和应用程序之间传递、复制数据包。当应用程序要发送一个数据包时，数据通过系统调用提交到内核中，系统会分配一个 SKB 来存储数据，然后往下层传递，在传递给网络驱动后才将其释放。当网络设备接收到数据包后，同样要分配一个 SKB 来存储数据，然后往上传递，最终在数据复制到应用程序后释放。

2.8 设备无关接口

网络协议栈底部是一个与硬件无关的接口层，它将网络层的不同协议与各种网络设备连接在一起。设备无关接口提供了一组通用函数供底层网络设备驱动程序和上层协议栈调用，这样，当输出数据时协议栈不必关心底层的网络设备，而当输入数据时网络设备驱动同样也不必关心上层的协议栈。

协议栈向设备发送数据包时都需调用 dev_queue_xmit()。该函数对 SKB 进行排队，最终由底层设备驱动程序进行传输。

而接收报文通常是调用 netif_rx()实现的。当底层设备驱动程序接收一个报文时，就会通过调用 netif_rx()将报文的 SKB 上传至网络层。NAPI 是最近引入的一种接口，是一种中断机制与轮询机制的混合体，能有效地提高网络处理速度。通过 NAPI 技术，在网络重负荷时能显著地减少由于接收数据包而产生的硬中断的数量，对高频小数据包的处理非常有效。

2.9 设备驱动程序

网络设备由 net_device 结构来描述，每个网络设备都会有一个对应的实例，然后调用 register_netdevice()注册到系统中，注册过的网络设备可通过 unregister_netdevice()注销。

net_device 结构中包含了一个名为 hard_start_xmit 的接口，通常在初始化网络设备时设置该接口，实现向该网络设备输出数据包。当协议栈向设备中发送数据包时，最终会通过设备无关接口调用到此接口。在接收数据包时，支持 NAPI 驱动程序通常需实现 net_device 结构的 poll 接口。

2.10 网络模块源代码组织

图 2-6 显示了在网络体系架构中，IPv4 主要用到的一些目录，本书中的代码均来自这些目录。

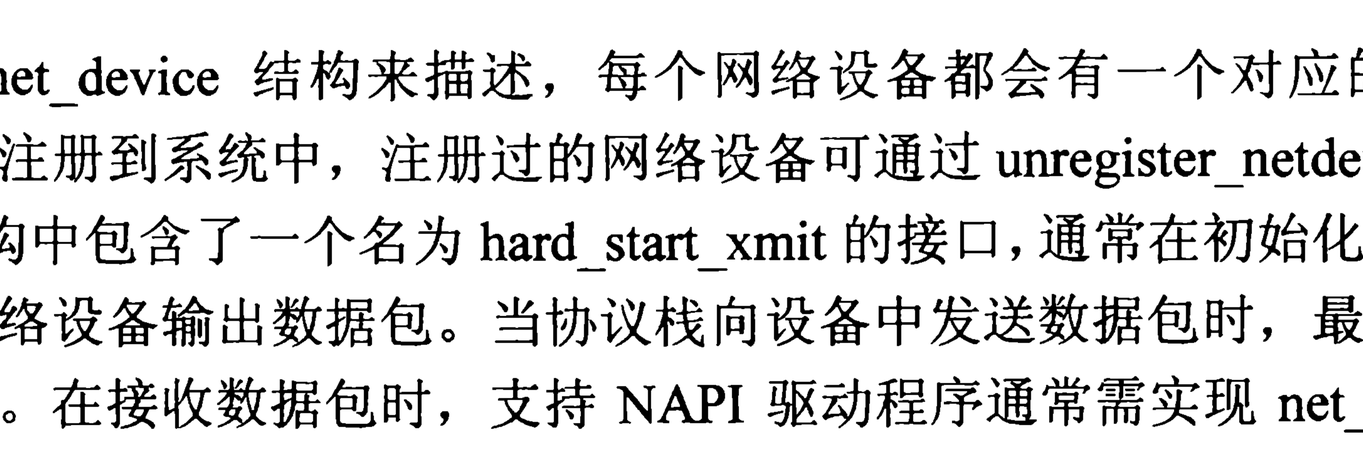

图 2-6 网络模块的目录树

第3章　套接口缓存

3.1　引言

网络协议中的操作对系统的存储及设计要求是非常高的，这些要求包括：

- 能够很方便地处理可变长缓存，因为接收和发送的数据报长度不是固定的。
- 能够很容易地实现在头尾部添加和移除数据，因为这些缓存区需要在不同网络层次间进行传递。
- 在添加和移除数据时能够尽量避免数据的复制。

以上这些操作将会直接影响网络处理的性能。在 Linux 网络子系统中称为套接口缓存（socket buffer），相应的数据结构为 sk_buffer，在以后的章节中将简称为“SKB”。本章中会对 SKB 以及内核中用于操作 SKB 的函数作出说明，而在本书以后的每一章几乎都会遇到 SKB，因此必须对其有深入了解。SKB 的主要用途是保存在进程和网络接口之间互相传递的用户数据，以及其他的一些信息（如套接口选项等）。

SKB 的操作函数及宏涉及以下文件：

- include/linux/skbuff.h，SKB 结构定义和 SKB 宏。
- net/core/skbuff.c，操作 SKB 的函数。

3.2　sk_buff 结构

sk_buffer 结构是 Linux 网络模块中最重要的数据结构之一，用以描述已接收或待发送的数据报文信息。其定义复杂，由很多成员组成，目的是为了便于在网络协议栈的各层间进行无缝传递，灵活处理。该结构是随着 Linux 内核的发展而发展的，到目前为止，结构已经很清晰，功能也比较丰富，其成员可大致分为以下几类：

- 与 SKB 组织相关的成员变量。
- 通用成员变量。
- 标志性变量。
- 与特性相关的成员变量。

SKB 在不同网络协议层之间传递，可被用于不同的网络协议，如二层的 MAC 或者是其他的链接层协议、三层的 IP 协议、四层的 TCP 或者 UDP 协议，其中某些成员变量会在该结构从一层向另一层传递时发生改变。四层向三层传递前会添加一个四层首部，同样，三层向二层传递前也会添加一个三层首部。添加首部比在不同层之间复制数据效率更高。由于在数据缓存区的头部添加数据意味着要修改指向数据缓存区的指针，这是个复杂的操作，所以内核提供了skb_reserve()来完成这个功能（参见 3.4.4 节）。协议栈中的每一层在往下一层传递 SKB 前，首先就是调用该函数在数据缓存区头部预留出的空间。后续章节中，会介绍 SKB 在不同协议层之间传递时，内核是如何在数据缓存区头部预留一定的空间以保证每一层都能把本层的协议首部

添加到数据缓存区的。

如果是向上层协议传递 SKB，则下层协议层的首部信息就没有用了。例如，二层首部只有在网络驱动处理二层协议时有用，三层是不会关心的。但是内核并没有把二层首部从数据缓存区中删除，而只是把有效载荷指针指向三层首部，这样做能在很大程度上提高效率。

3.2.1 网络参数和内核数据结构

在网络模块中同时也提供了很多有用的功能，虽然这些功能并不是必须的，但对现在的应用来讲是不可缺少的一部分，例如，防火墙、组播等。为了支持这些功能，一般都需要在内核数据结构 sk_buff 中添加相应的成员变量。因此，sk_buff 结构中包含很多像#ifdef 这样的预编译指令。例如，在 sk_buff 结构的最后可以看到如下代码：

```
230 struct sk_buff {
... ...
300 #ifdef CONFIG_NET_SCHED
301     __u16 tc_index;
302 #ifdef CONFIG_NET_CLS_ACT
303    __u16 tc_verd;
304 #endif
305 #endif
... ...
322 }
```

从以上的代码片断中可以看出，tc_index 成员变量只有在编译时定义了宏 CONFIG_NET_SCHED 时才有效。该符号可以在编译时通过选择特定的编译选项来定义。例如，“Networking->Networking options->QoS and/or fair queueing”，如图 3-1 所示。

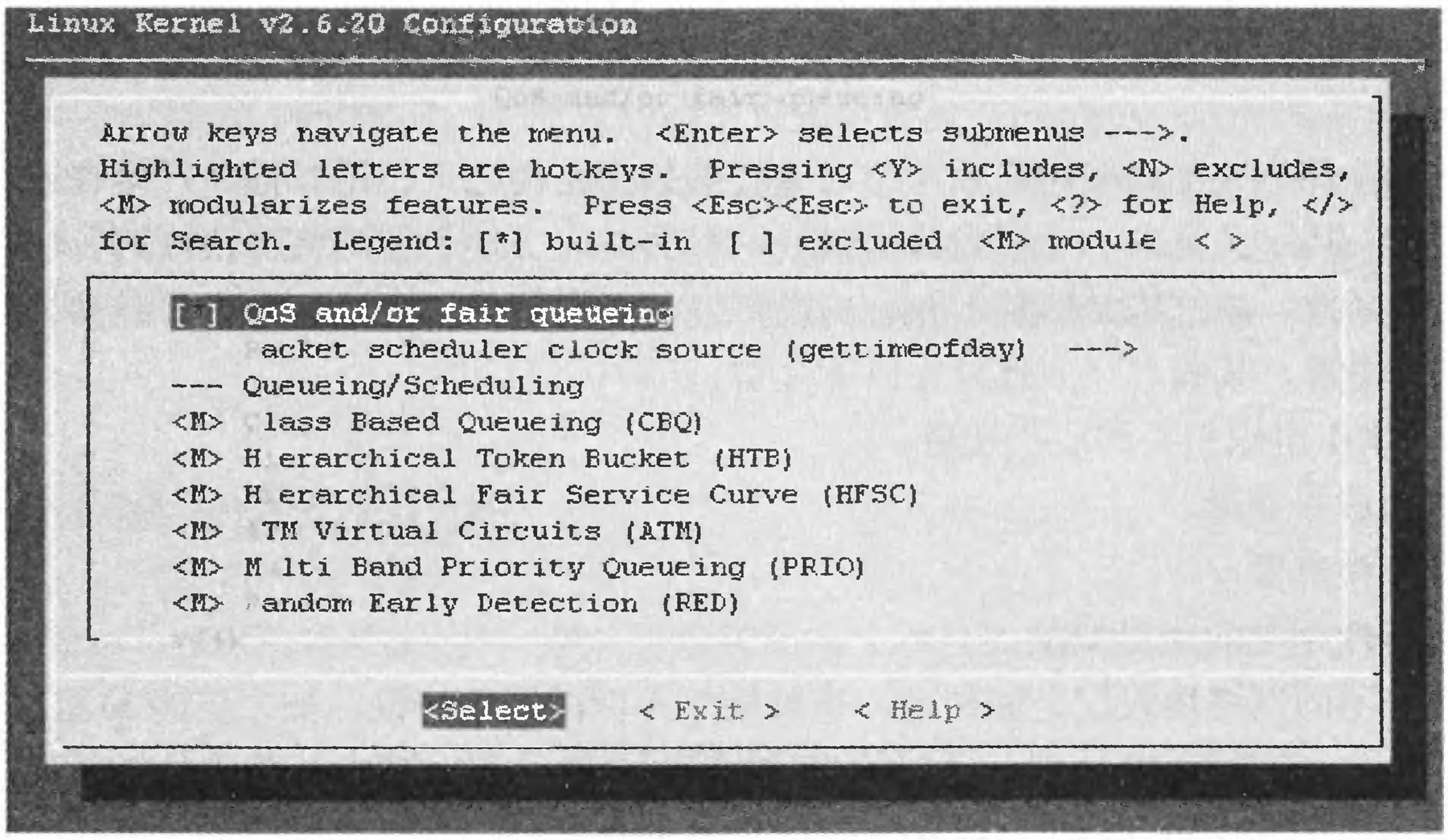

图 3-1 启用 CONFIG_NET_SCHED 宏

而嵌套的选项 CONFIG_NET_CLS_ACT(包分类器)只有在选择支持“QoS and/or fair queueing”时才能生效。例如，“Networking->Networking options->QoS and/or fair queueing->Actions”，如图 3-2 所示。

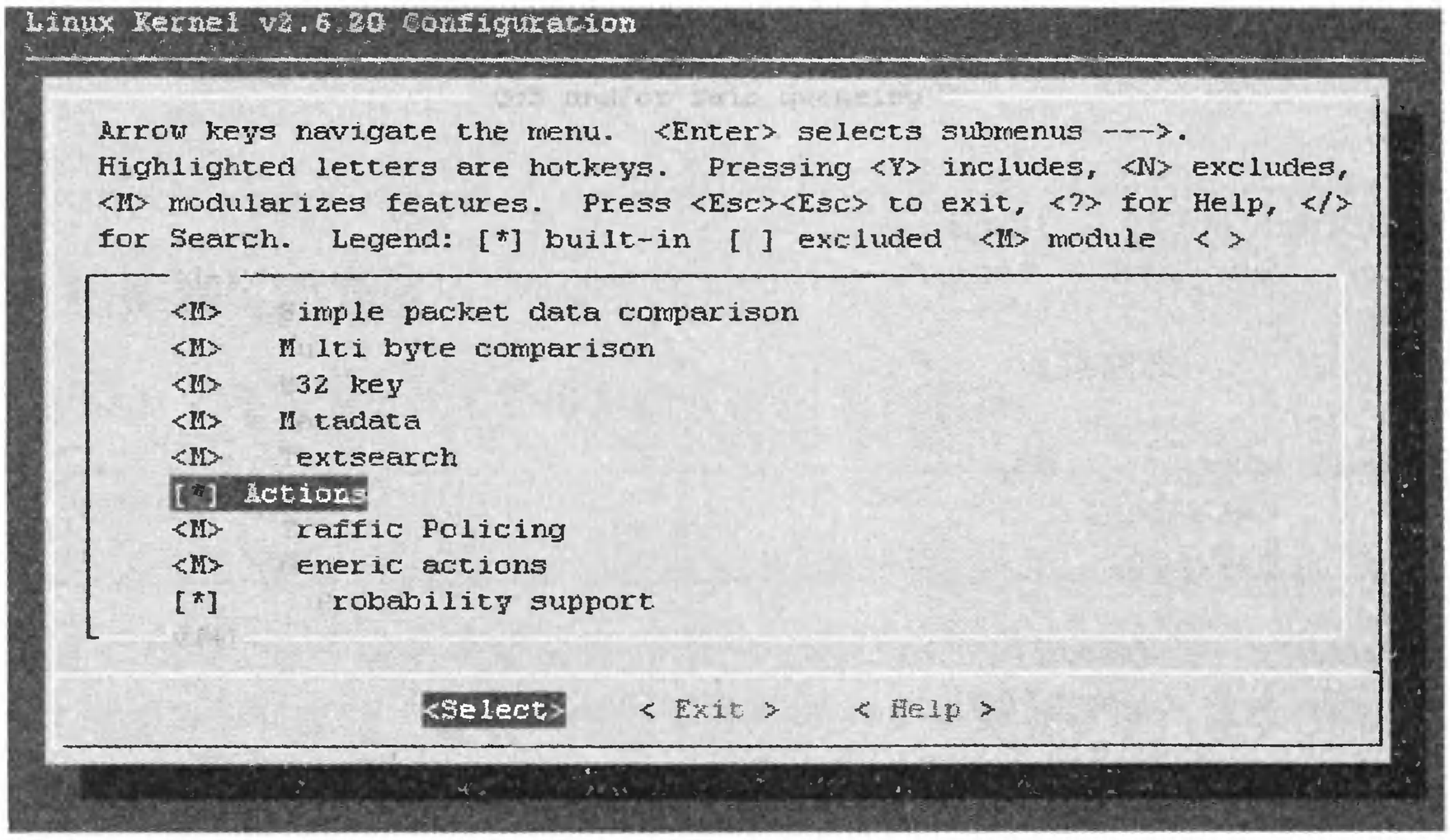

图 3-2　启用 CONFIG_NET_CLS_ACT 宏

需要指出的是，在上述例子中，QoS 是不能被编译成内核模块的。原因就是，内核编译之后，由某些选项所控制的数据结构是固定的而不是动态变化的。一般来说，如果某些选项修改了内核数据结构，则包含该选项的组件就不能被编译成内核模块。例如，在 sk_buff 中增加了一个 tc_index 字段。

SKB 有两个部分，一部分为 SKB 描述符（sk_buff 结构本身），另一部分为数据缓存区（参见 sk_buff 结构的 head 成员），以下是对 sk_buff 结构的详细描述，包括各个成员：

```
230 struct sk_buff {
231     /* These two members must be first. */
232     struct sk_buff        *next;
233     struct sk_buff        *prev;
234
235     struct sock       *sk;
236     struct skb_timeval    tstamp;
237     struct net_device     *dev;
238     struct net_device     *input_dev;
239
240     union {
241         struct tcphdr     *th;
242         struct udphdr     *uh;
243         struct icmphdr    *icmph;
244         struct igmphdr    *igmph;
245         struct iphdr      *ipiph;
246         struct ipv6hdr    *ipv6h;
247         unsigned char     *raw;
248     } h;
249
250     union {
251         struct iphdr      *iph;
252         struct ipv6hdr    *ipv6h;
253         struct arphdr     *arph;
254         unsigned char     *raw;
255     } nh;
256
```

```
257     union {
258         unsigned char    *raw;
259     } mac;
260
261     struct  dst_entry    *dst;
262     struct    sec_path    *sp;
263
270     char          cb[48];
271
272     unsigned int       len,
273               data_len,
274               mac_len;
275     union {
276         __wsum        csum;
277         __u32        csum_offset;
278     };
279     __u32           priority;
280     __u8           local_df:1,
281               cloned:1,
282               ip_summed:2,
283               nohdr:1,
284               nfctinfo:3;
285     __u8           pkt_type:3,
286               fclone:2,
287               ipvs_property:1;
288     __be16           protocol;
289
290     void           (*destructor)(struct sk_buff *skb);
291 #ifdef CONFIG_NETFILTER
292     struct nf_conntrack    *nfct;
293 #if defined(CONFIG_NF_CONNTRACK) || defined(CONFIG_NF_CONNTRACK_MODULE)
294     struct sk_buff       *nfct_reasm;
295 #endif
296 #ifdef CONFIG_BRIDGE_NETFILTER
297     struct nf_bridge_info    *nf_bridge;
298 #endif
299 #endif /* CONFIG_NETFILTER */
300 #ifdef CONFIG_NET_SCHED
301     __u16           tc_index;    /* traffic control index */
302 #ifdef CONFIG_NET_CLS_ACT
303     __u16           tc_verd;    /* traffic control verdict */
304 #endif
305 #endif
306 #ifdef CONFIG_NET_DMA
307     dma_cookie_t       dma_cookie;
308 #endif
309 #ifdef CONFIG_NETWORK_SECMARK
310     __u32           secmark;
311 #endif
312
313     __u32           mark;
314
315     /* These elements must be at the end, see alloc_skb() for details.  */
316     unsigned int       truesize;
317     atomic_t       users;
318     unsigned char       *head,
```

```
319                 *data,
320                 *tail,
321                 *end;
322 };
```

3.2.2　SKB 组织相关的变量

本节论述 SKB 的相关成员，主要用来构成 SKB 双向链表。

```
232 struct sk_buff *next
233 struct sk_buff *prev
```

在 sk_buff 结构中，有些成员变量只是为了便于组织该数据结构本身。例如，内核把 sk_buff 组织成一个双向链表，只是这个链表的组织比传统的双向链表要复杂，就像任何一个双向链表一样，sk_buff 结构中有两个指针 next 和 prev，next 指向下一个结点，prev 指向上一个结点。但这个链表还有个要求：每个 SKB 必须能被整个链表的头部快速找到。为了满足这个需求，在第一个 SKB 结点前面会插入另一个辅助的 sk_buff_head 结构的头结点，可以认为该 sk_buff_head 结构就是 SKB 链表的头结点。

```
108 struct sk_buff_head {
109     /* These two members must be first. */
110     struct sk_buff    *next;
111     struct sk_buff    *prev;
112
113     __u32        qlen;
114     spinlock_t    lock;
115 };

110 struct sk_buff *next
111 struct sk_buff *prev
```

通过这两个指针将 SKB 连接成一个双向链表。

113 __u32 qlen

SKB 链表中结点数，即队列长度。

114 spinlock_t lock

用来控制对 SKB 链表并发操作的自旋锁。

图 3-3 为由 sk_buff_head 和三个 SKB 组成的链表，相信会有助于理解 sk_buff_head 和 SKB 之间的关系。

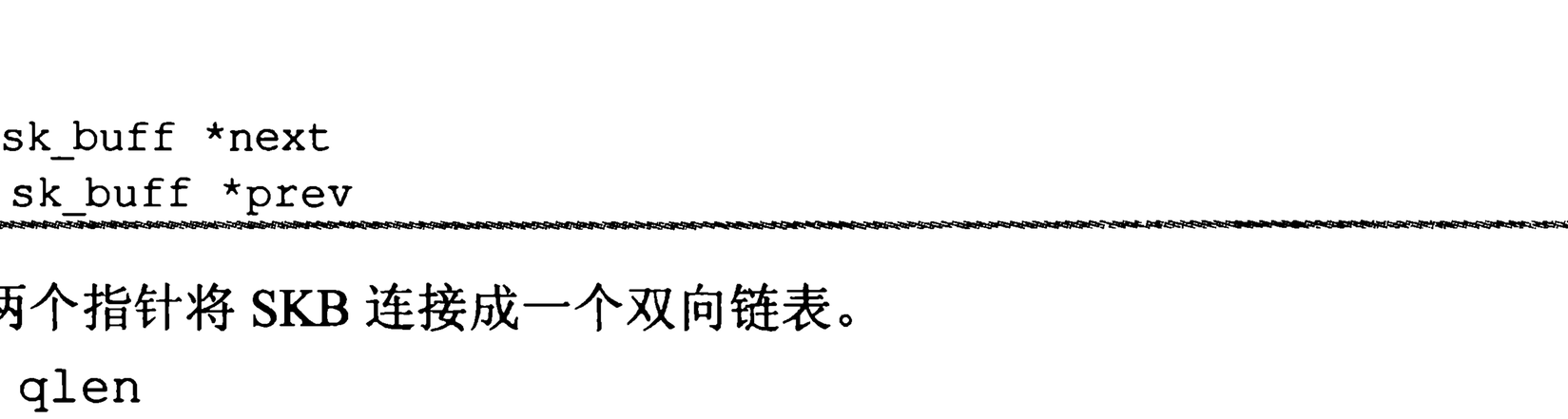

图 3-3　SKB 链表

3.2.3 数据存储相关的变量

本节论述 SKB 数据存储相关的成员，主要用来描述 SKB 的宿主、数据的长度等。

```
235 struct sock *sk
```

以上结构是 SKB 的宿主传输控制块。SKB 的宿主传输控制块在网络数据报文由本地发出或由本地接收时才有效，使传输控制块与套接口及用户应用程序相关。当一个 SKB 仅在二层或三层被转发时，即源 IP 地址和目的 IP 地址都不是本机地址时，指针值为 NULL。

```
272 unsigned int len
```

SKB 中数据部分长度，包括线性缓存区中数据长度（由 data 指向），SG 类型的聚合分散 I/O 的数据以及 FRAGLIST 类型的聚合分散 I/O 的数据长度（参见 3.3 节）。该字段值随着 SKB 从一个协议层向另一个协议层传递而改变，向上传递时，下层首部就不再需要了，而向下层传递时，需添加本层首部。因此，len 也包含了协议首部的长度，参见 3.4 节。

```
273 unsigned int data_len
```

SG 类型和 FRAGLIST 类型聚合分散 I/O 存储区中的数据长度。

```
274 unsigned int mac_len
```

二层首部长度。实际长度与网络的介质相关，在以太网中为以太网帧首部的长度。

```
317 atomic_t users
```

引用计数，用来标识有多少实体引用了该 SKB。其主要作用是确定释放所属 SKB 的时机，当计数器为零时，SKB 才能被释放。因此，每个引用该 SKB 的实体都必须在适当的时候递增和递减引用计数。注意，该计数器只保护 SKB 描述符，而 SKB 数据缓存区也有类似的计数器（skb_shared_info 结构的 dataref 成员），将在 3.3 节中介绍。在内核中，通常使用 skb_get()和 kfree_skb()操作 SKB 描述符引用计数。实际上 skb_get()在返回前先执行了 atomic_inc()操作，而 kfree_skb()则先调用了 atomic_dec_and_test()，当引用计数为 0 时，才会真正进行释放，否则只是简单地递减引用计数而已。

```
316 unsigned int truesize
```

整个数据缓存区的总长度，包括 SKB 描述符和数据缓存区部分（包括线性存储区和聚合分散 I/O 缓存区）。如果申请一个 len 字节的缓存区，alloc_skb()会将 truesize 初始化成 len+sizeof(sk_buff)，该字段随 len 而变化，参见 3.4.2 节。

```
318 unsigned char*head,*data,*tail,*end
```

head，end，data，tail 四个字段用来指向线性数据缓存区及数据部分的边界。head 和 end 分别指向缓存区的头与尾；而 data 和 tail 则分别指向数据的头与尾。在发送时，每一层协议会在 head 与 data 之间填充协议首部，还可能在 tail 和 end 之间添加数据，参见 3.4 节中相关函数。如图 3-4 所示，数据缓存区 end 处紧接着一个 skb_shared_info 结构。

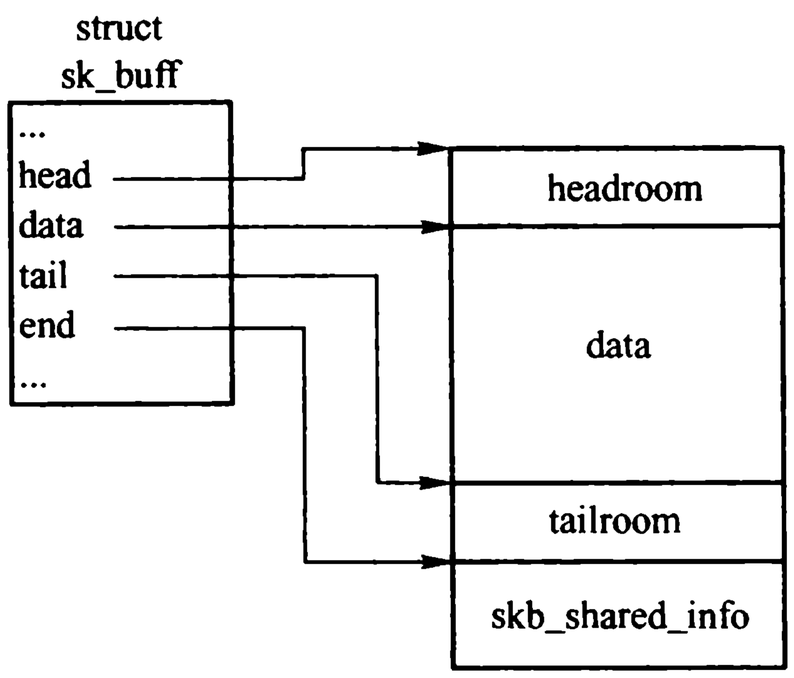

图 3-4 skb 的 head/end 指针和 rxsj data/tail 指针

```
290 void (*destructor)(struct sk_buff *skb)
```

SKB 析构函数指针，在释放 SKB 时被调用，完成某些必要的工作。如果 SKB 没有宿主传输控制块，则该函数指针通常为空，这种情况主要发生在被转发时；否则通常分别在 skb_set_owner_r()和 skb_set_owner_w()中被初始化成

sock_rfree()或 sock_wfree()。当释放 SKB 后，该 SKB 不再属于指定的传输控制块，因此要根据释放的 SKB 的 truesize 来调整传输控制块的接收和发送缓存区大小（sk_rmem_alloc 和 sk_wmem_alloc）。

3.2.4　通用的成员变量

本节论述 sk_buff 结构中与内核特定功能无关的成员，它们主要和网络协议、网络设备等有关。

```
236 struct skb_timeval tstamp
```

接收时间戳，或发送时间戳。通常在网络设备收到一个数据包之后，通过 netif_receive_skb()或 netif_rx()调用 net_timestamp()进行设置。

```
237 struct net_device *dev
```

网络设备指针。该字段的作用与该 SKB 是发送包还是接收包有关。在初始化网络设备驱动，分配接收缓存队列时，将该指针指向收到数据包的网络设备。例如，在 e100 系列的以太网驱动中，初始化时会建立 sk_buff 缓存队列，调用__netdev_alloc_skb()分配缓存，如果分配成功则设置 dev。

```
268 struct sk_buff *__netdev_alloc_skb(struct net_device *dev,
269       unsigned int length, gfp_t gfp_mask)
270 {
271    int node = dev->class_dev.dev ? dev_to_node(dev->class_dev.dev) : -1;
272    struct sk_buff *skb;
273
274     skb = __alloc_skb(length + NET_SKB_PAD, gfp_mask, 0, node);
275    if (likely(skb)) {
276       skb_reserve(skb, NET_SKB_PAD);
277       skb->dev = dev;
278    }
279    return skb;
280 }
```

发送数据包时，该指针指向输出数据包的网络设备，发送时设置该字段要比接收时的设置复杂得多，参见第 11 章。

Linux 支持多种形式的虚拟网络设备，并由一个虚拟网络设备驱动管理。当这个虚拟设备被使用时，dev 指针指向该虚拟设备的 net_device 结构。在输出时，虚拟设备驱动会在一组设备中选择其中的某个合适的设备，并将 dev 指针修改为指向这个设备的 net_device 结构。而在输入时，当原始网络设备接收到报文后，根据某种算法选择某个合适的虚拟网络设备，并将 dev 指针修改为指向这个虚拟设备的 net_device 结构。因此，在某些情况下，指向传输设备的指针会在包处理过程中改变。

```
238 struct net_device *input_dev
```

接收报文的原始网络设备。如果包是本地生成的，则该值为 NULL，主要用于流量控制。

```
240 union { } h
250 union { } nh
257 union { } mac
```

h，nh，mac 分别是指向各层协议首部的指针：h 指向四层协议首部，nh 指向三层协议首部，

mac 指向二层协议首部。每个指针的类型都是一个联合体，包含多个数据结构，每个数据结构都表示在这一层上可以解析的协议。例如，h 是一个包含内核所能解析的所有四层协议的数据结构联合体，如 tcphdr 结构、udphdr 结构等。每个联合体中都有一个 raw 变量，主要用于初始化，初始化完成后，后续的访问都是通过与协议相关的结构实例进行的。

当接收到一个包时，处理 n 层协议的函数从 n-1 层收到一个 SKB，其 skb->data 指向 n 层协议首部。处理 n 层协议的函数把本层首部指针（例如，三层对应的是 skb->nh 指针）初始化为 skb->data。在处理 n 层协议的函数结束时，在把包传递给 n+1 层处理函数前，会将 skb->data 指针指向 n 层协议首部的末尾，这正好是 n+1 层协议的首部，如图 3-5 所示。

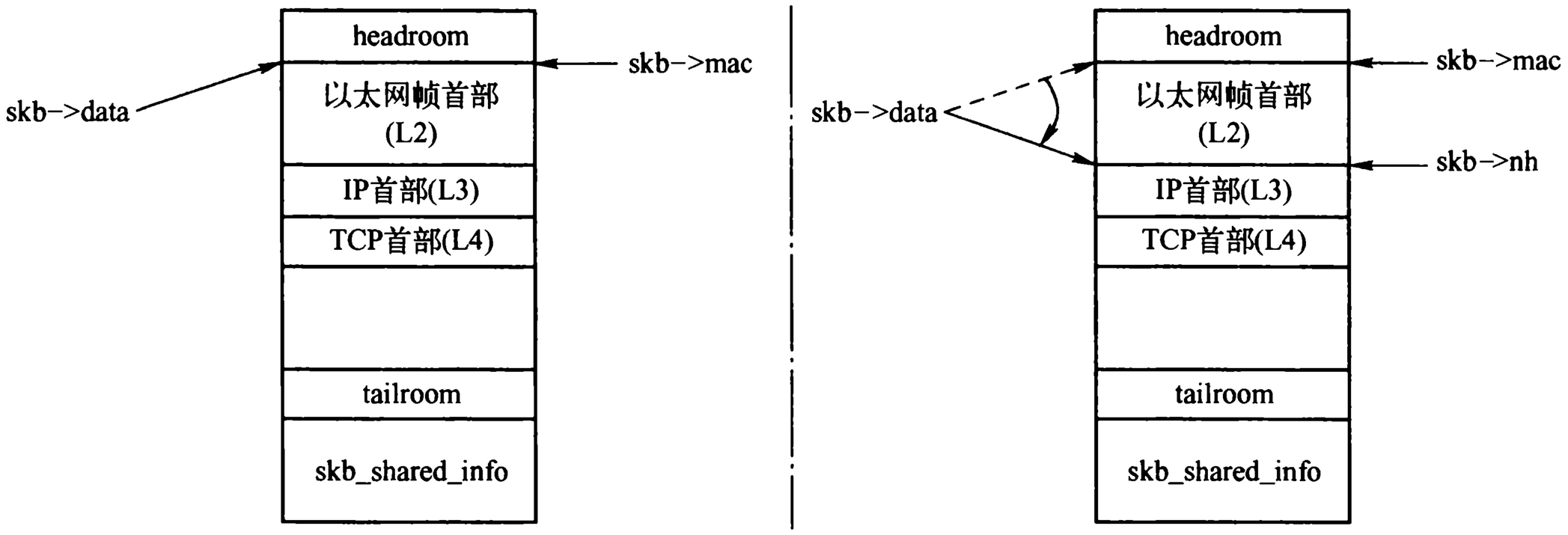

图 3-5　报文从二层向三层传递时 data 指针的变化

而发送包的过程刚好相反，由于每一层需为当前层添加新的协议首部，该过程要比接收包的过程复杂。

```
261 struct dst_entry *dst
```

目的路由缓存项。不管是输入的数据包还是输出的数据包，都需要经过路由子系统的查询得到目的路由缓存项之后，才能确定数据包的流向，否则查询不到路由的数据包最终只能丢弃。由于该结构比较复杂，且需了解其他子系统是如何工作的，此处不作论述，参见第 19～21 章。

```
270 char cb[48]
```

SKB 信息控制块，是每层协议的私有信息存储空间，由每一层协议自己维护并使用，并只在本层有效。在分配 SKB 时固定在 SKB 描述符中，其当前的大小是 48B，已经足够为每一层协议存储必要的私有信息了。在每个协议中，访问该字段的代码通常用宏实现以增强代码的可读性。例如，TCP 用这个成员存储数据，操作时由 tcp_skb_cb 结构来映射。

```
522 struct tcp_skb_cb {
...
529     __u32       seq;            /* Starting sequence number    */
530     __u32       end_seq;        /* SEQ + FIN + SYN + datalen   */
531     __u32       when;           /* used to compute rtt's       */
532     __u8        flags;          /* TCP header flags.           */
533
...
561 };
```

下面这个宏在 TCP 代码中用来访问 cb 字段。其实这个宏就是一个简单的类型转换：

```
#define TCP_SKB_CB(__skb)  ((struct tcp_skb_cb *)&((__skb)->cb[0]))
```

下面的例子是 TCP 模块在收到一个段时填充相关数据结构的代码。

```
1611 int tcp_v4_rcv(struct sk_buff *skb)
1612 {
...
1640
1641    th = skb->h.th;
1642    TCP_SKB_CB(skb)->seq = ntohl(th->seq);
1643    TCP_SKB_CB(skb)->end_seq = (TCP_SKB_CB(skb)->seq + th->syn + th->fin +
1644                skb->len - th->doff * 4);
1645    TCP_SKB_CB(skb)->ack_seq = ntohl(th->ack_seq);
1646    TCP_SKB_CB(skb)->when    = 0;
1647    TCP_SKB_CB(skb)->flags   = skb->nh.iph->tos;
1648    TCP_SKB_CB(skb)->sacked  = 0;
...
1658 }
```

想要进一步了解 cb 中的数据是如何被取出的，可参见 tcp_transmit_skb()，该函数被 TCP 用来向 IP 层发送一个分段。

```
275 union {
276    __wsum csum;
277     u32 csum offset;
278 }
282 __u8 ip_summed:2
```

csum 在校验状态为 CHECKSUM_NONE 时，用于存放所负载数据报的数据部分的校验和，为计算完成的传输层校验和做准备。csum_offset 在校验状态为 CHECKSUM_PARTIAL 时，记录传输层首部中的校验和字段的偏移。由于两种状态是互斥的，因此 csum 和 csum_offset 作为一个联合体来存储。

ip_summed 标记传输层校验和的状态，见表 3-1。

表 3-1　ip_summed 的取值

ip_summed	描述
CHECKSUM_NONE	表示硬件不支持，完全由软件来执行校验和
CHECKSUM_PARTIAL	表示由硬件来执行校验和
CHECKSUM_UNNECESSARY	表示没有必要执行校验和
CHECKSUM_COMPLETE	表示已经完成执行校验和

```
281 __u8 cloned
```

标记所属 SKB 是否已克隆。

```
285 __u pkt_type:
```

帧类型，分类是由二层目的地址来决定的，见表 3-2。对于以太网设备来说，该字段由 eth_type_trans()初始化。

表 3-2 pkt_type 的取值

pkt_type	描述
PACKET_HOST	数据包的 MAC 目的地址与收到它的网络设备的 MAC 地址相等，也就是说这个报文是发给接收该数据包的主机的
PACKET_BROADCAST	数据包的 MAC 目的地址是一个广播地址，而这个广播地址也是收到这个包的网络设备的广播地址
PACKET_MULTICAST	数据包的 MAC 目的地址是一个组播地址，而这个组播地址也是收到这个包的网络设备所注册的组播地址
PACKET_OTHERHOST	数据包的 MAC 目的地址与收到它的网络设备的地址完全不同（不管是单播、组播还是广播），因此，如果本机没有启用转发功能，则这个包会被丢弃
PACKET_OUTGOING	这个数据包将被发出。用到这个标记的功能包括 Decnet 协议，或者是为每个网络 tap 都复制份发出包的函数
PACKET_LOOPBACK	这个数据包发向 loopback 设备。由于有这个标记，在处理 loopback 设备时，内核可以跳过一些真实设备才需要的操作
PACKET_FASTROUTE	这个数据包由快速路由代码查找路由。快速路由功能在 2.6 内核中已经去掉

27 __u32 priority

发送或转发数据包 QoS 类别。如果包是本地生成的，套接口层会设置该字段；如果包是转发的，则 rt_tos2priority()会根据 IP 首部中 TOS 域来计算该字段值。

288 _be16 protocol

从二层设备角度看到的上层协议，即链路层承载的上层协议类型。典型的协议包括 IP、IPv6 和 ARP，见表 3-3。完整的列表在 include/linux/if_ether.h 中。由于每个协议都有各自处理接收数据包的函数，因此该域被设备驱动用来通知上层调用哪个协议处理函数。由于每个网络驱动都调用 netif_rx()来通知上层网络协议的处理函数，因此 protocol 必须在这些协议处理函数调用前初始化。

表 3-3 protocol 的取值

protocol	描述
ETH_P_IP	IPv4 报文
ETH_P_ARP	地址查询报文
ETH_P_8021Q	802.1Q VLAN 报文
ETH_P_PPP_DISC	PPPoE 发现消息报文
ETH_P_PPP_SES	PPPoE 会话消息报文

3.2.5 标志性变量

本节论述 SKB 标志相关的成员，主要用来标识 payload 是否被单独引用，是否允许分片，以及当前克隆的状态等。

283 _i8 nohdr

标识 payload 是否被单独引用，不存在协议首部。如果被引用，则决不能再修改协议首部，也不能通过 skb->data 来访问协议首部。

280 __u8 local_df

表示此 SKB 在本地允许分片。

286 __u8 fclone:2

当前克隆状态，见表 3-4。

表 3-4　fclone

fclone	描述
SKB_FCLONE_UNAVAILABLE	SKB 未被克隆
SKB_FCLONE_ORIG	在 skbuff_fclone_cache 分配的父 SKB，可以被克隆
SKB_FCLONE_CLONE	在 skbuff_fclone_cache 分配的子 SKB，从父 SKB 克隆得到的

3.2.6　特性相关的成员变量

Linux 内核是模块化的，在编译时可以选择包含或删除某些功能。因此，sk_buff 结构里面的一些成员变量只在内核选择支持某些功能时才有效，如防火墙（netfilter）或者 QoS 等。

```
284 __u8 nfctinfo
293 struct nf_conntrack *nfct
297 struct nf_bridge_info *nf_bridge
```

这些变量被 netfilter（防火墙）使用，本书不作论述。

```
301 __u16 tc_index
303 __u16 tc_verd
```

用于输入流量控制，本书不作论述。

```
262 struct sec_path *sp
```

IPSec 协议用来跟踪传输的信息，本书不作论述。

3.3　skb_shared_info 结构

如图 3-4 所示，在数据缓存区的末尾，即 end 指针所指向的地址起紧跟着有一个 skb_shared_info 结构，保存了数据块的附加信息。

```
133 struct skb_shared_info {
134     atomic_t    dataref;
135     unsigned short    nr_frags;
136     unsigned short    gso_size;
137     /* Warning: this field is not always filled in (UFO)! */
138     unsigned short    gso_segs;
139     unsigned short  gso_type;
140     __be32          ip6_frag_id;
141     struct sk_buff    *frag_list;
142     skb_frag_t    frags[MAX_SKB_FRAGS];
143 };
```

```
134 atomic_t dataref
```

引用计数器。当一个数据缓存区被多个 SKB 的描述符引用时，就会设置相应的计数。比如克隆一个 SKB。

```
140 __be32 ip6_frag_id
```

与 IPv6 相关，本书不作论述。

3.3.1　“零拷贝”技术

在通常的数据发送和接收操作中，一般都调用 read/write/send 系统调用。而实际上，这些

操作的效率是比较低下的，其原因在于，切换进程上下文时，需在内核空间与用户空间之间复制数据。如果有大量的数据要进行读写，则对性能会产生非常大的影响。为了解决该问题，提高读写性能，从 Linux 内核版本 2.1 起引入了 sendfile 系统调用，目的就是简化通过网络在两个本地文件之间进行数据传输的过程。sendfile 系统调用的引入，不仅减少了数据复制，还减少了上下文切换的次数。sendfile()的原型如下：

```
ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count)
```

图 3-6 和图 3-7 分别是 read+write 系统调用和 open+sendfile 系统调用过程中的数据复制，有助于读者更好地理解有关操作。

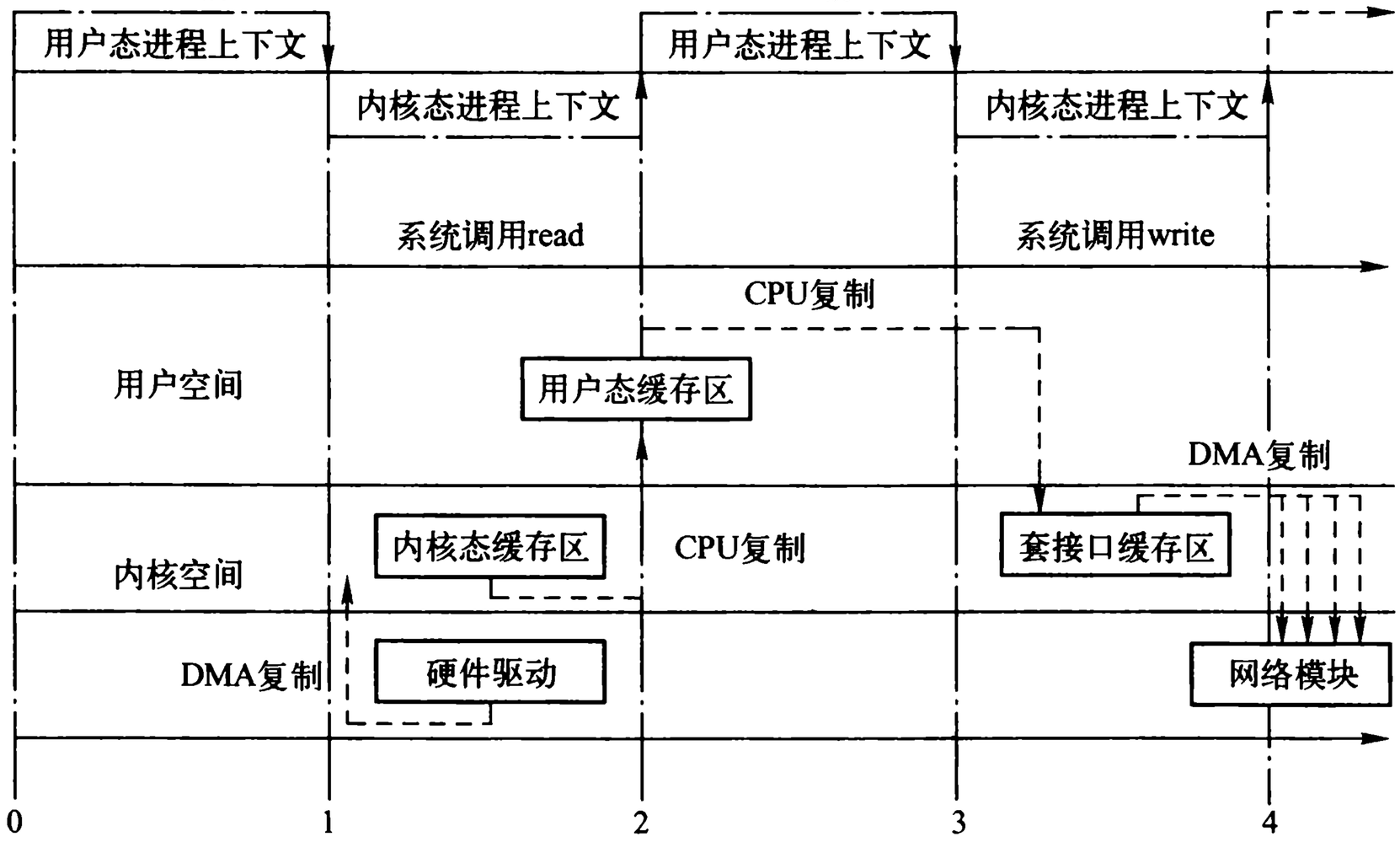

图 3-6　read+write 系统调用过程中的数据复制

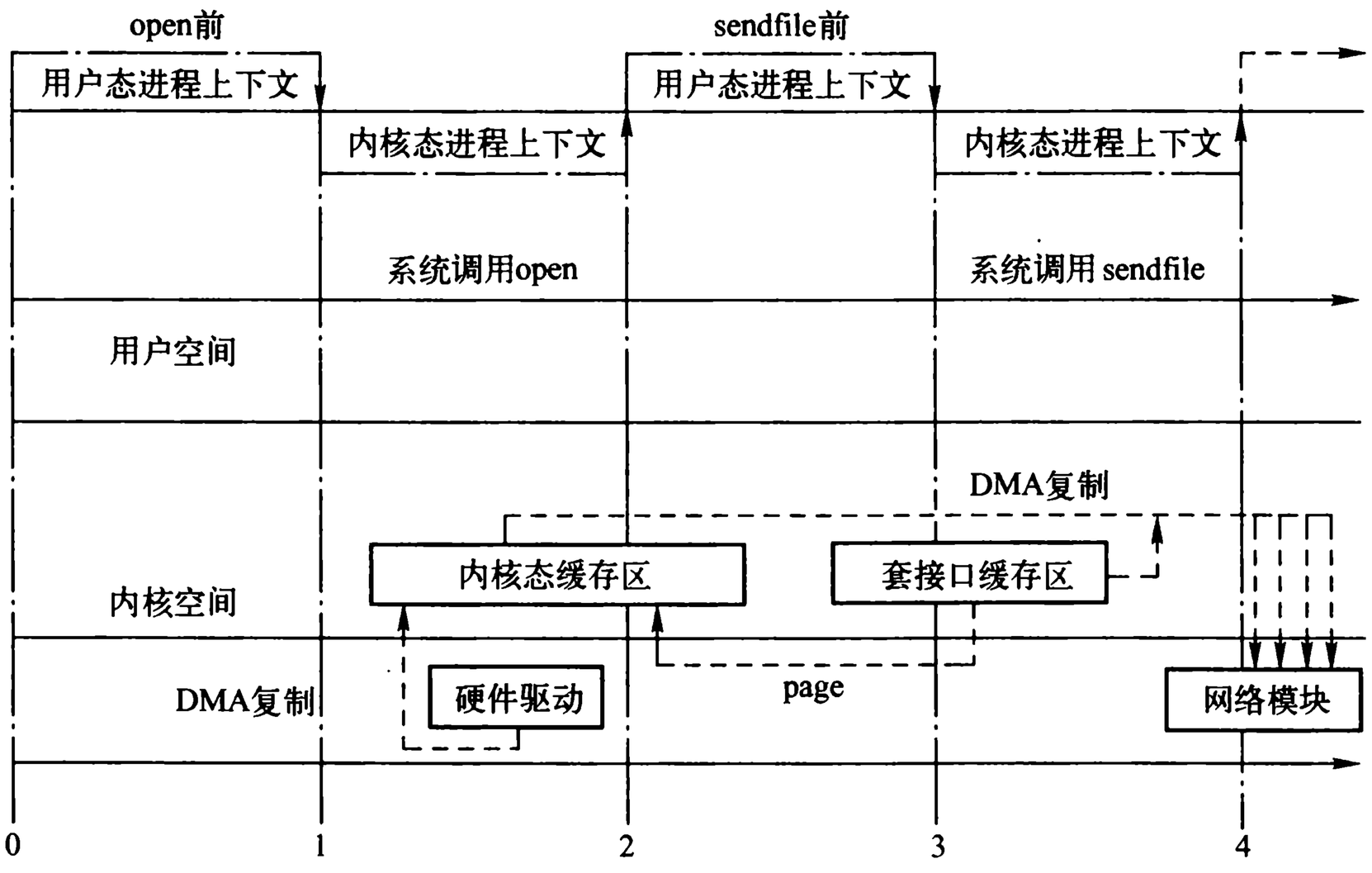

图 3-7　open+sendfile 系统调用过程中的数据复制

从图 3-6 中可以清楚地看到，在 read+write 处理过程中，有四次数据复制过程，即从硬件

的 DMA 缓存复制到内核缓存，从内核缓存复制到用户缓存，从用户缓存复制到内核缓存，以及从内核缓存复制到硬件 DMA 缓存。上面的过程中存在很多数据冗余，而某些冗余是完全可以消除的，这样可以减少开销、提高性能。

为了减少开销，可以从消除内核缓存区与用户缓存区之间的复制入手，而消除内核产生的数据冗余，需要网络适配器支持聚合操作特性。该特性意味着待发送的数据不要求存放在地址连续的内存空间中，而是可以分散在多个内存位置。在内核版本 2.4 中，SKB 描述符结构发生了改动，以适应聚合分散 I/O 操作的要求——这就是 Linux 中所谓的“零拷贝”。这种方式不仅减少了多个上下文切换，而且消除了数据冗余。从图 3-7 可以看到经过改进后的机制，明显减少了两次数据复制，整个过程中只有两次必要的复制：一是数据通过 DMA 模块被复制到内核缓存区中；二是由于数据并未被复制到套接口关联的缓存区内，而只是记录数据位置和长度的数据缓存区被加入到 SKB 中，因此 DMA 模块将数据直接从内核缓存区传递给协议模块即可。

3.3.2　对聚合分散 I/O 数据的支持

本节论述 SKB 的成员用来支持数据的聚合分散 I/O。

```
141 struct sk_buff *frag_list
135 unsigned short nr_frags
142 skb_frag_t frags[MAX_SKB_FRAGS]
```

nr_frags，frags 和 frag_list 与 IP 分片的存储有关。通常数据都存储在线性区域中，但当为了支持聚合分散 I/O，数据需要存储在支持聚合分散 I/O 的区域中。frags 和 frag_list 支持聚合分散 I/O 数据，而“零拷贝”技术也是需要聚合分散 I/O 支持的。

那么什么是聚合分散 I/O 呢？网络中创建一个发送报文的过程包括组合多个片。报文数据必须从用户空间复制到内核空间，同时加上网络协议栈各层的首部。这个组合可能要求相当数量的数据拷贝。但如果发送报文的网络接口能够支持聚合分散 I/O，报文就无需组装成一个单块，可避免大量拷贝。聚合分散 I/O 从用户空间启动“零拷贝”网络发送。内核传递发送报文给 hard_start_xmit()之前需要判断网络设置是否支持 NETIF_F_SG，不然只能对分散的报文进行线性化处理，再聚合成一个单独的报文。如果网络设备已经设置了该标志，接下来就要检查 nr_frags 的值，因为该字段确定了片段数，这些分散的片段以关联的方式存储在 frags 数组中。

接下来再来看一下 skb_frag_struct 结构。

```
120 #define MAX_SKB_FRAGS (65536/PAGE_SIZE + 2)
121
122 typedef struct skb_frag_struct skb_frag_t;
123
124 struct skb_frag_struct {
125     struct page *page;
126     __u16 page_offset;
127     __u16 size;
128 };
```

```
120 #define MAX_SKB_FRAGS (65536/PAGE_SIZE + 2)
```

MAX_SKB_FRAGS 为 frags 数组的大小，因此最多可支持 64K 个分片。

```
125 struct page *page
```

指向文件系统缓存页的指针。

126 __u16 page_offset

数据起始地址在文件系统缓存页中的偏移。

127 __u16 size

数据在文件系统缓存页面中使用的长度。

frag_list 有以下几种使用方法：

1）用于在接收分片组后链接多个分片，组成一个完整的 IP 数据报。

2）在 UDP 数据报的输出中，将待分片的 SKB 链接到第一个 SKB 中，然后在输出过程中能够快速地分片。

3）用于存放 FRAGLIST 类型的聚合分散 I/O 的数据包，如果输出网络设备支持 FRAGLIST 类型的聚合分散 I/O（目前只有回环设备支持），则可以直接输出。

frags 和 nr_frags 用于支持 SG 类型的聚合分散 I/O 分片。frags 是用于存放聚合分散 I/O 分片的数组。nr_frags 为当前使用聚合分散 I/O 分片的数量，即为 frags 数组中使用的数量，不超过 MAX_SKB_FRAGS。

skb_is_nonlinear()用来测试 SKB 是否存在聚合分散 I/O 缓存区，实际上就是判断 sk_buff 的 data_len 成员。而 skb_linearize()可以把含有聚合分散 I/O 缓存区中的数据线性化到线性缓存区中。线性过程中涉及数据复制，所以如果多次进行数据的复制，将严重影响系统性能，因此若无必要，不要进行数据的线性化。

图 3-8 所示为一个数据全部存储在线性存储区中的报文，从图中可以看到数据长度 len 为 x，即 data 到 tail 区域的长度，而聚合分散 I/O 存储区中的数据长度 data_len 为 0，说明这个报文没有报文聚合分散 I/O 数据，事实上在代码中也是根据该值作判断的。分片是由 skb_shared_info 结构中的 nr_frags 和 frag_list 控制的，在此 nr_frags 为 0，而 frag_list 指针为 NULL，这两个值再一次说明了这个报文没有分片。

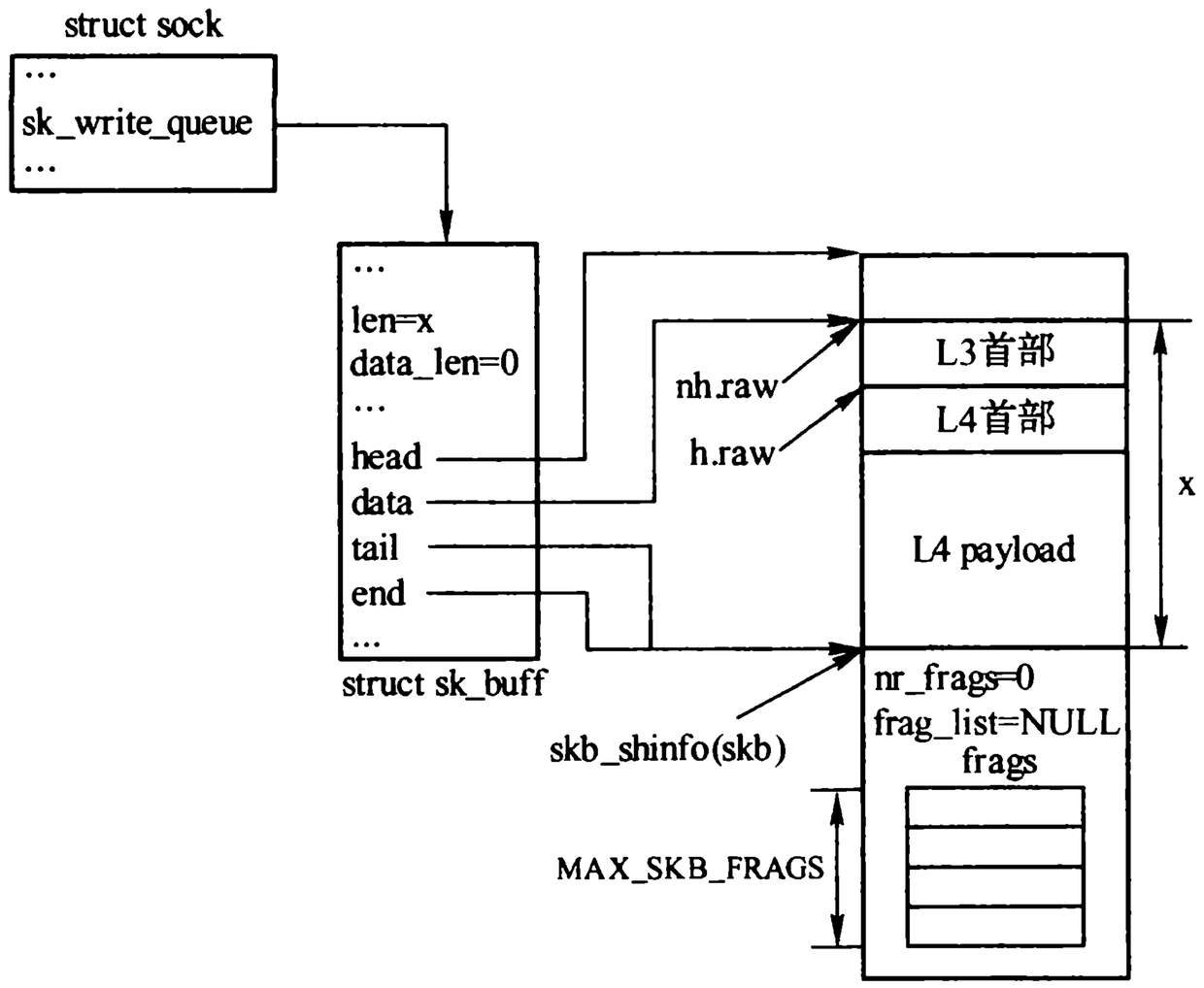

图 3-8　没有启用分片（包括聚合分散 I/O 的分片）的报文

图 3-9 所示为启用了聚合分散 I/O 的报文。从图中可以看到数据长度 len 为 x+S1+S2，其中 x 即 data 到 tail 区域的长度，而 S1 和 S2 分别为两个分片的长度；而聚合分散 I/O 数据长度 data_len 为 S1+S2，表示这个报文存在聚合分散 I/O 数据。skb_shared_info 结构中的 nr_frags 为

2，而 frag_list 为 NULL，这说明不是普通分片，而是聚合分散 I/O 分片，数量为 2。这两个分片指向同一个物理分页，各自在分页中的偏移和长度分别是 0/S1 和 S1/S2。

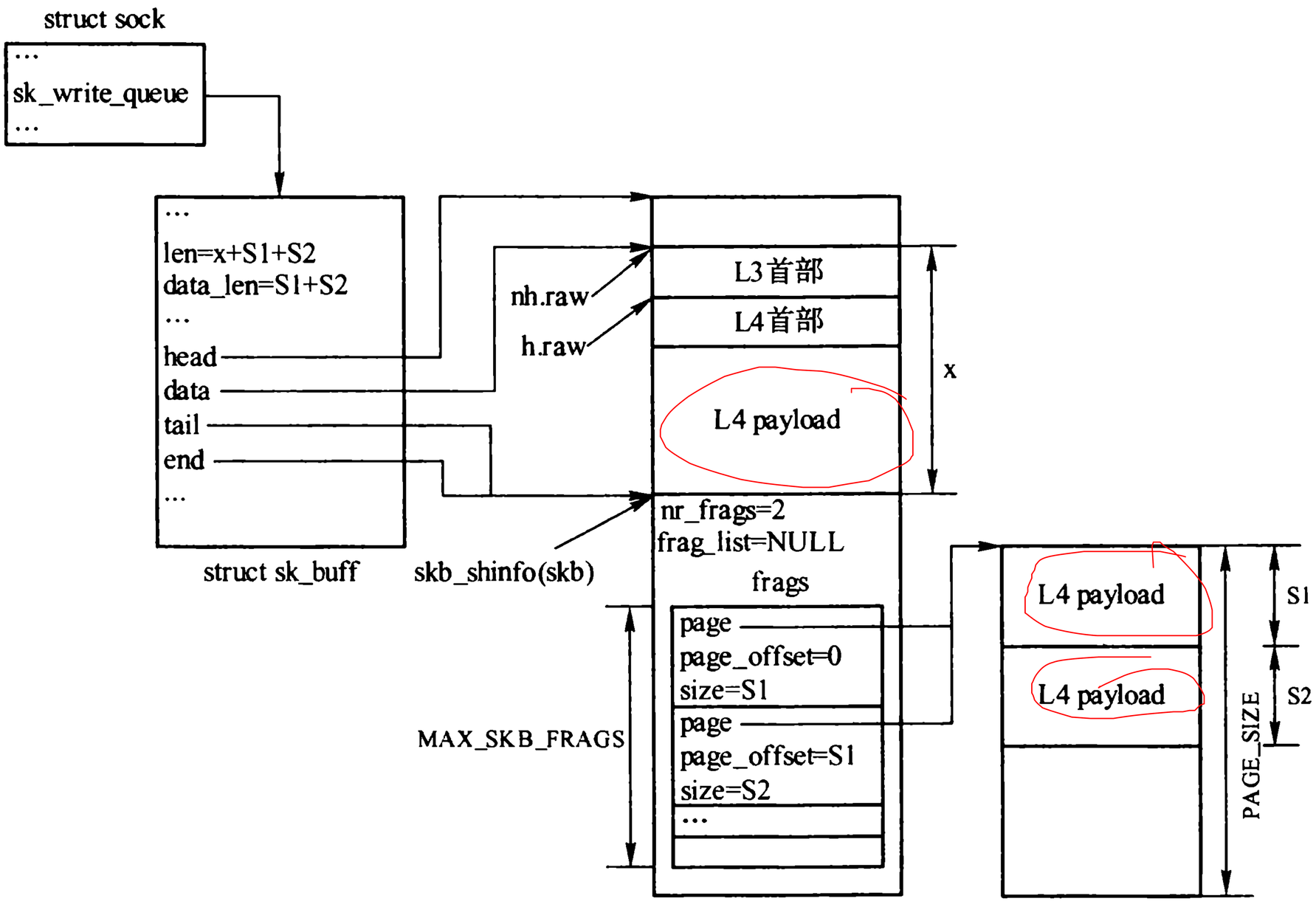

图 3-9　只启用聚合分散 I/O 分片的报文

图 3-10 所示为两个启用聚合分散 I/O 分片的报文共享内存。从图中可以看到报文 a 和 b 两个 SKB 数据长度 len 分别为 x+S1 和 y+S2，各自 skb_shared_info 结构中的 page 指向同一个页面，但在分页中的偏移和数据长度不同，偏移分别为 0 和 S1，长度分别为 S1 和 S2。

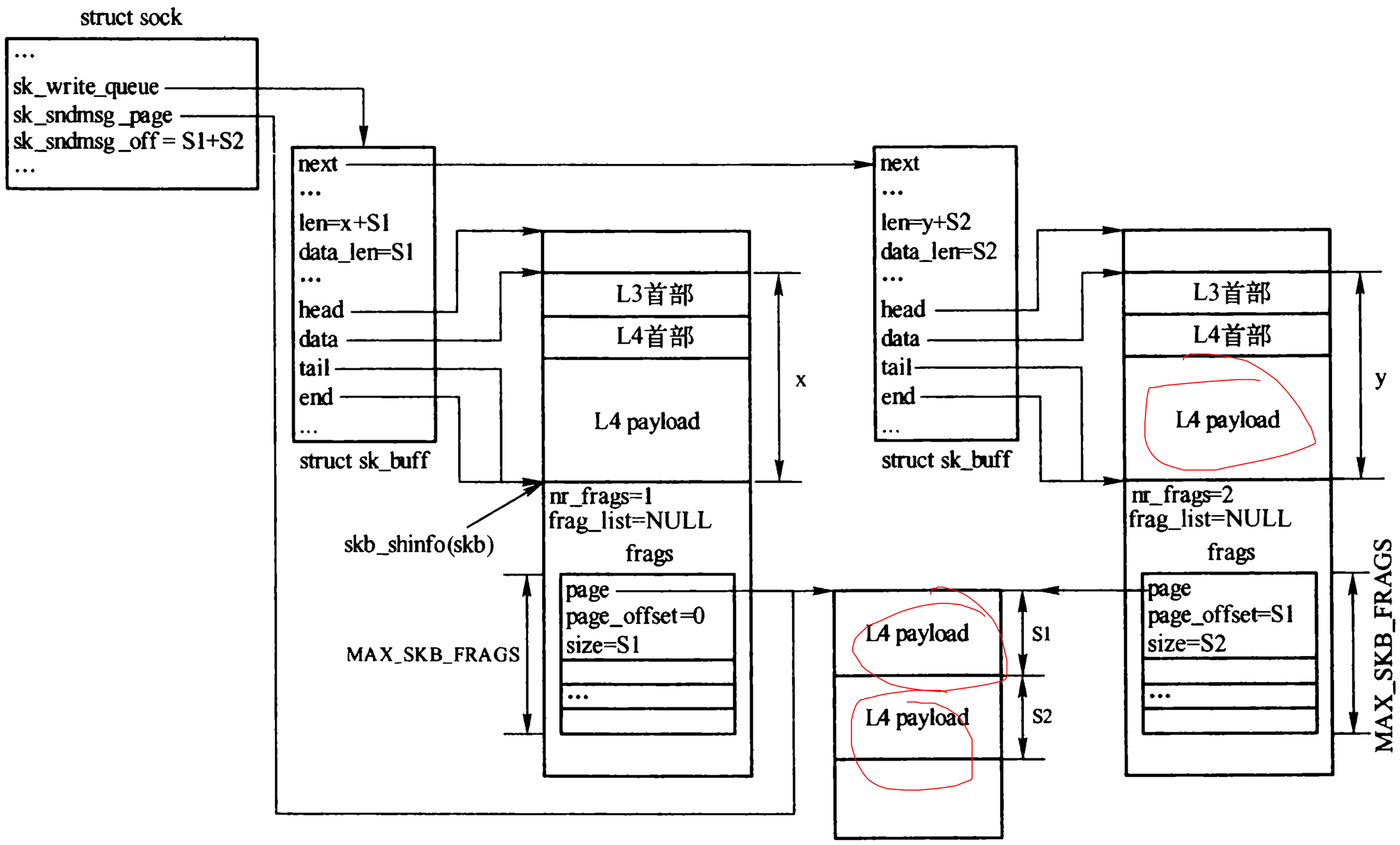

图 3-10　启用聚合分散 I/O 分片的两个报文共享内存

图 3-11 所示为 FRAGLIST 类型的分散聚合 I/O 数据的报文。从图中看到数据长度 len 为 x+S1，其中 x 为 data 到 tail 区域的长度，而 S1 为 FRAGLIST 类型的分散聚合 I/O 数据的报文的长度。data_len 为 S1，表示该报文存在分散聚合 I/O 数据。skb_shared_info 结构的 nr_frags 为 0，而 frag_list 不为 NULL，这说明该报文存在 FRAGLIST 类型的分散聚合 I/O 数据。

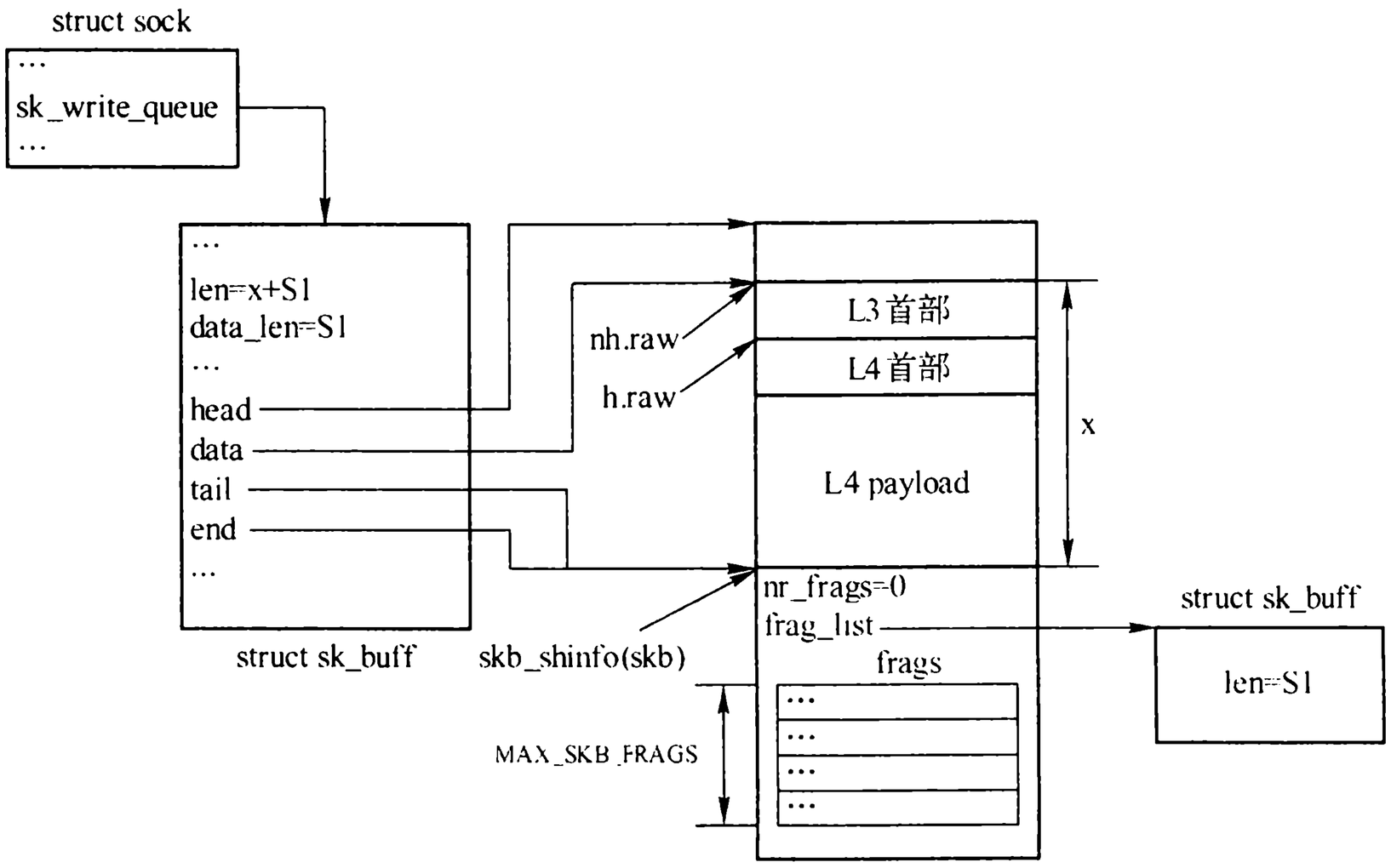

图 3-11 包含 FRAGLIST 类型的分散聚合 I/O 数据的报文

3.3.3 对 GSO 的支持

有些网络设备硬件可以完成一些传统上由 CPU 完成的任务，最常见的例子就是计算三层和四层校验和。有些网络设备甚至可以维护四层协议的状态机，由硬件完成分段或分片，因此传输层通过网络层提交给网络设备时可能是个 GSO 段，参见 1.3.1 节。本节论述 SKB 的成员都是用来支持 GSO 的。

```
136 unsigned short gso_size
```

生成 GSO 段时的 MSS，因为 GSO 段的长度是与发送该段的套接口中合适 MSS 的整数倍。

```
138 unsigned short gso_segs
```

GSO 段的长度是 gso_size 的倍数，即用 gso_size 来分割大段时产生的段数。

```
139 unsigned short gso_type
```

该 SKB 中的数据支持的 GSO 类型，见表 3-5。

表 3-5 gso_type 的取值

gso_type	描述
SKB_GSO_TCPV4	IPv4 的 TCP 段卸载
SKB_GSO_UDP	IPv4 的 UDP 分片卸载
SKB_GSO_DODGY	表明数据报是从一个不可信赖的来源发出的
SKB_GSO_TCP_ECN	IPv4 的 TCP 段卸载，当设置 TCP 首部的 CWR 时，使用此 gos_type。CWR 参见 29.4 节
SKB_GSO_TCPV6	IPv6 的 TCP 段卸载

3.3.4　访问 skb_shared_info 结构

需要注意的是，sk_buff 结构中并没有指向 skb_shared_info 结构的指针，可使用 skb_info 宏来访问 skb_shared_info 结构，skb_inf 宏简单地返回 sk_buff 结构的 end 指针。

```
#define skb_shinfo(SKB) ((struct skb_shared_info *)((SKB)->end))
```

下面的语句展示了如何使用 skb_info 宏来增加 skb_shared_info 结构中成员变量 dataref 的值。

```
skb_shinfo(skb)->dataref++;
```

3.4　管理函数

在网络模块中，有很多函数通常都比较短小而简单，内核用这些函数来操作 SKB 描述符中的变量或者 SKB 链表。在 skbuff.h 和 skbuff.c 中，很多函数都有两个版本，名字分别是 do_something()和__do_something()，通常前者是一个封装函数，会在后者的基础上加上合法性校验或者获取锁等操作。建议不要直接调用类似__do_something()的函数，除非能确认满足特定的调用条件，如加锁等。下面是 do_something()的通常形式：

```
static inline do_something(parameter_list)
{
    unsigned long flags;
    spin_lock_irqsave(...);
    __do_something(parameter_list)
    spin_unlock_irqrestore(...);
}
```

这些函数先获取锁，然后调用一个以两个下画线开头的同名函数，后者完成具体的操作且无需锁，最后释放锁。

3.4.1　SKB 的缓存池

网络模块中，有两个用来分配 SKB 描述符的高速缓存，在 SKB 模块初始函数 skb_init()中被创建。

```
2048 void __init skb_init(void)
2049 {
2050     skbuff_head_cache = kmem_cache_create("skbuff_head_cache",
2051                           sizeof(struct sk_buff),
2052                           0,
2053                           SLAB_HWCACHE_ALIGN|SLAB_PANIC,
2054                           NULL, NULL);
2055     skbuff_fclone_cache = kmem_cache_create("skbuff_fclone_cache",
2056                          (2*sizeof(struct sk_buff)) +
2057                          sizeof(atomic_t),
2058                          0,
2059                          SLAB_HWCACHE_ALIGN|SLAB_PANIC,
2060                          NULL, NULL);
2061 }
```

2050-2054 创建 skbuff_head_cache 高速缓存，一般情况下，SKB 都是从该高速缓存中分配的。

2055-2060 创建每次以两倍 SKB 描述符长度来分配空间的 skbuff_fclone_cache 高速缓存。如果在分配 SKB 时就知道可能被克隆，那么应该从这个高速缓存中分配空间，因为在这个高速缓存中分配 SKB 时，会同时分配一个后备的 SKB，以便将来用于克隆，这样在克隆时就不用再次分配 SKB 了，直接使用后备的 SKB 即可，这样做的目的主要是提高效率。

两个高速缓存的区别在于创建时指定的单位内存区域大小不同，skbuff_head_cache 的单位内存区域长度是 sizeof(struct sk_buff)，而 skbuff_fclone_cache 的单位内存区域长度是 2*sizeof(struct sk_buff)+sizeof(atomic_t)，即一对 SKB 和一个引用计数，可以说这一对 SKB 是“父子”关系，指向同一个数据缓存区，引用计数值为 0,1 或 2，用来表示这一对 SKB 中有几个已被使用，如图 3-12 所示。

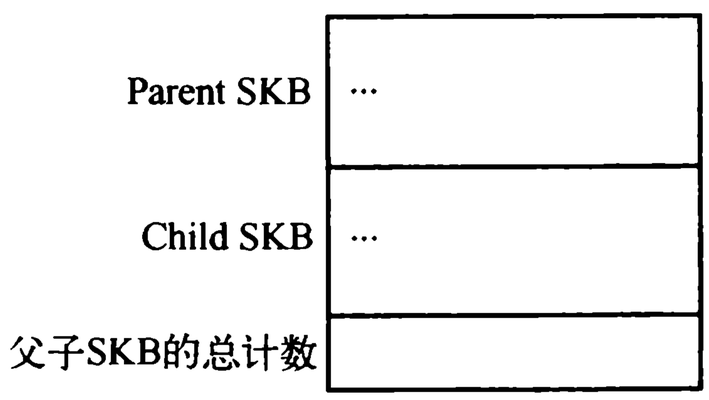

图 3-12　从 skbuff_fclone_cache 高速缓存分配到的 SKB

3.4.2　分配 SKB

1．alloc_skb()

alloc_skb()用来分配 SKB。数据缓存区和 SKB 描述符是两个不同的实体，这就意味着，在分配一个 SKB 时，需要分配两块内存，一块是数据缓存区，一块是 SKB 描述符。__alloc_skb()调用 kmem_cache_alloc_node()从高速缓存中获取一个 sk_buff 结构的空间，然后调用 kmalloc_node_track_caller()分配数据缓存区。参数说明如下：

- size，待分配 SKB 的线性存储区的长度。
- gfp_mask，分配内存的方式，见表 25-3。
- fclone，预测是否会克隆，用于确定从哪个高速缓存中分配。
- node，当支持 NUMA（非均匀质存储结构）时，用于确定何种区域中分配 SKB。NUMA 参见相关资料。

```
144 struct sk_buff *__alloc_skb(unsigned int size, gfp_t gfp_mask,
145                int fclone, int node)
146 {
147     struct kmem_cache *cache;
148     struct skb_shared_info *shinfo;
149     struct sk_buff *skb;
150     u8 *data;
151
152     cache = fclone ? skbuff_fclone_cache : skbuff_head_cache;
153
154     /* Get the HEAD */
155     skb = kmem_cache_alloc_node(cache, gfp_mask & ~__GFP_DMA, node);
156     if (!skb)
```

```
157         goto out;
158
159     /* Get the DATA. Size must match skb_add_mtu(). */
160     size = SKB_DATA_ALIGN(size);
161     data = kmalloc_node_track_caller(size + sizeof(struct skb_shared_info),
162             gfp_mask, node);
163     if (!data)
164         goto nodata;
165
166     memset(skb, 0, offsetof(struct sk_buff, truesize));
167     skb->truesize = size + sizeof(struct sk_buff);
168     atomic_set(&skb->users, 1);
169     skb->head = data;
170     skb->data = data;
171     skb->tail = data;
172     skb->end  = data + size;
173     /* make sure we initialize shinfo sequentially */
174     shinfo = skb_shinfo(skb);
175     atomic_set(&shinfo->dataref, 1);
176     shinfo->nr_frags  = 0;
177     shinfo->gso_size = 0;
178     shinfo->gso_segs = 0;
179     shinfo->gso_type = 0;
180     shinfo->ip6_frag_id = 0;
181     shinfo->frag_list = NULL;
182
183     if (fclone) {
184         struct sk_buff *child = skb + 1;
185         atomic_t *fclone_ref = (atomic_t *) (child + 1);
186
187         skb->fclone = SKB_FCLONE_ORIG;
188         atomic_set(fclone_ref, 1);
189
190         child->fclone = SKB_FCLONE_UNAVAILABLE;
191     }
192 out:
193     return skb;
194 nodata:
195     kmem_cache_free(cache, skb);
196     skb = NULL;
197     goto out;
198 }
```

152 根据参数 fclone 确定从哪个高速缓存中分配 SKB。

155 调用 kmem_cache_alloc_node()从选定的高速缓存中分配一个 SKB。在此从分配标志中去除 GFP_DMA，是为了不从 DMA 内存区域中分配 SKB 描述符，因为 DMA 内存区域比较小且有特定用途，没有必要用来分配 SKB 描述符。而后面分配数据缓存区时，就不会去掉 GFP_DMA 标志，因为很有可能数据缓存区就需要在 DMA 内存区域中分配，这样硬件可以直接进行 DMA 操作，参见 161~162 行。

160 在分配数据缓存区之前，强制对给定的数据缓存区大小 size 作对齐操作。

161-165 调用 kmalloc_node_track_caller()分配数据缓存区，其长度为 size 和 sizeof(struct skb_shared_info)之和，因为在缓存区尾部紧跟着一个 skb_shared_info 结构。

168-181 初始化新分配 SKB 描述符和 skb_shared_info 结构。

183-191 如果是 skbuff_fclone_cache 高速缓存中分配 SKB 描述符，则还需置父 SKB 描述符的 fclone 为 SKB_FCLONE_ORIG，表示可以被克隆；同时将子 SKB 描述符的 fclone 成员置为 SKB_FCLONE_UNAVAILABLE，表示该 SKB 还没有被创建出来；最后将引用计数置为 1。

最后 SKB 结构如图 3-13 所示，在图右边所示的内存块中部，可以看到对齐操作所带来的填充区域。需要说明的是，__alloc_skb()一般不被直接调用，而是被封装函数调用，如__netdev_alloc_skb()、alloc_skb()、alloc_skb_fclone()等函数。

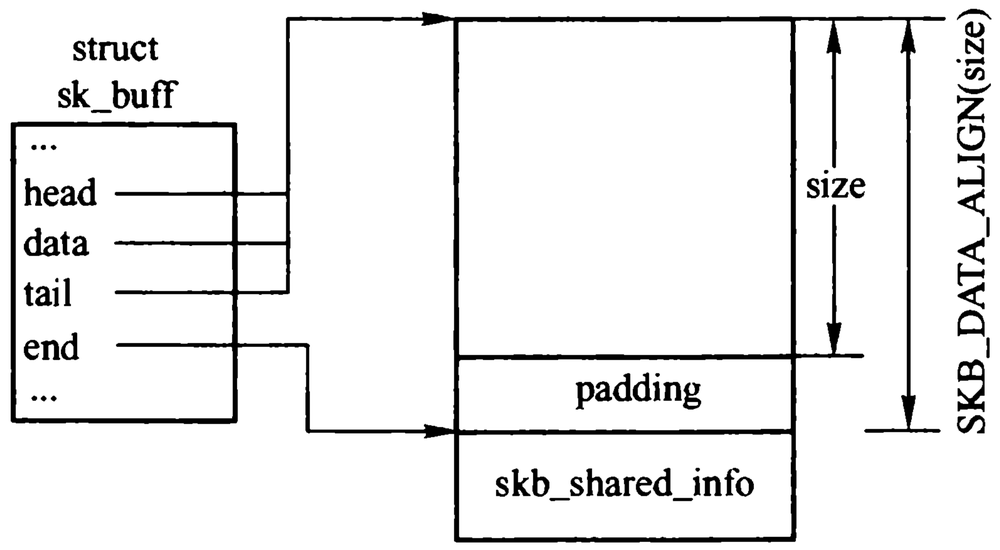

图 3-13　调用 alloc_skb()之后的套接口缓存结构

2. dev_alloc_skb()

dev_alloc_skb()也是一个缓存区分配函数，通常被设备驱动用在中断上下文中。这是一个 alloc_skb()的封装函数，因为是在中断处理函数中被调用的，因此要求原子操作（GFP_ATOMIC）。

```
1124 static inline struct sk_buff *dev_alloc_skb(unsigned int length)
1125 {
1126     return __dev_alloc_skb(length, GFP_ATOMIC);
1127 }
... ...
1103 static inline struct sk_buff *__dev_alloc_skb(unsigned int length,
1104                                               gfp_t gfp_mask)
1105 {
1106     struct sk_buff *skb = alloc_skb(length + NET_SKB_PAD, gfp_mask);
1107     if (likely(skb))
1108         skb_reserve(skb, NET_SKB_PAD);
1109     return skb;
1110 }
```

1108 调用 skb_reserve()在 skb->head 与 skb->data 之间预留 NET_SKB_PAD 个字节。NET_SKB_PAD 的定义在 skbuff.h 中，其值为 16。这部分空间将被填入硬件帧头，如 14B 的以太网帧头。

1126 以 GFP_ATOMIC 为内存分配优先级，表示分配过程为原子操作，不能被中断。

3.4.3　释放 SKB

dev_kfree_skb()和 kfree_skb()用来释放 SKB，把它返回给高速缓存。kfree_skb()可以直接调用，也可以通过封装函数 dev_kfree_skb()来调用。而 dev_kfree_skb()只是一个简单调用 kfree_skb()的宏，一般为设备驱动使用，与之功能相反的函数是 dev_alloc_skb()。这些函数只在 skb->users 为 1 的情况下才释放内存，否则只简单地递减 skb->users，因此假设 SKB 有三个引用者，那么只有第三次调用 dev_kfree_skb()或 kfree_skb()时才释放内存。kfree_skb()的流程如图 3-14 所示。

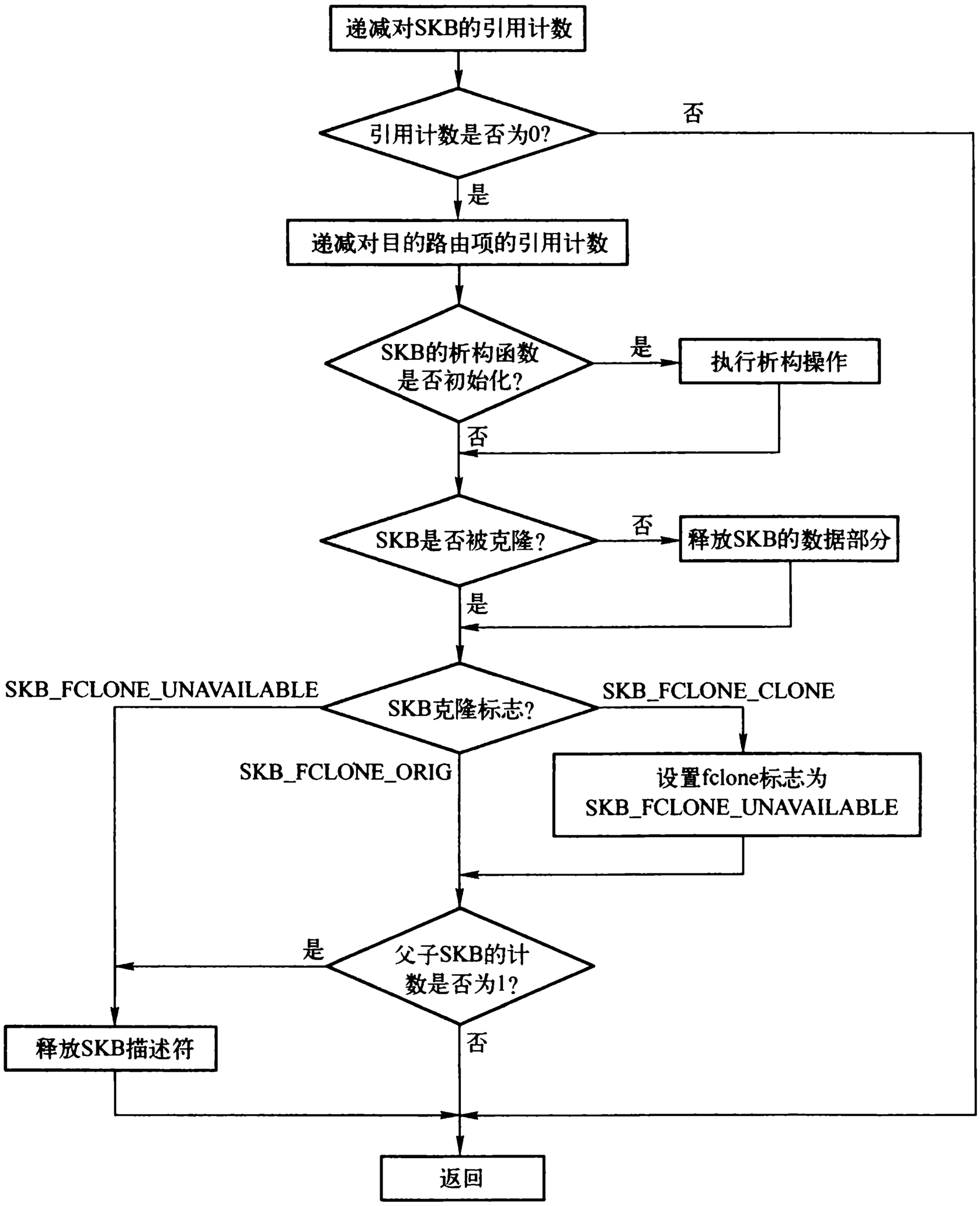

图 3-14　kfree_skb()流程

图 3-14 所示的流程显示了释放一个 SKB 的步骤：

1）kfree_skb()检测 sk_buff 结构的引用计数 users，如果不为 1，则说明此次释放后该 SKB 还将被用户占用，因此递减引用计数 users 后即返回；否则说明不再有其他用户占用该 sk_buff 结构，调用__kfree_skb()释放之。

2）SKB 描述符中包含一个 dst_entry 结构的引用，在释放 SKB 后，会调用 dst_release()来递减 dst_entry 结构的引用计数。

3）如果初始化了 SKB 的析构函数，则调用相应的函数。

4）一个 SKB 描述符是与一个存有真正数据的内存块，即数据区相关的。如果存在聚合分散 I/O 数据，该数据区底部的 skb_shared_info 结构还会包含指向聚合分散 I/O 数据的指针，同样需要释放这些分片所占用的内存。最后需把 SKB 描述符所占内存返回给 skbuff_head_cache 缓存。释放内存由 kfree_skbmem()处理，过程如下：

- 如果 SKB 没有被克隆，或者 payload 没有被单独引用，则释放 SKB 的数据缓存区，包括存储聚合分散 I/O 数据的缓存区和 SKB 描述符。
- 如果是释放从 skbuff_fclone_cache 中分配的父 SKB 描述符，且克隆计数为 1，则释放父

SKB 描述符。

- 如果是释放从 skbuff_fclone_cache 中分配的子 SKB 描述符，设置父 SKLB 的 fclone 字段为 SKB_FCLONE_UNAVAILABLE，在克隆计数为 1 的情况下，释放子 SKB 描述符。

3.4.4 数据预留和对齐

数据预留和对齐主要由 skb_reserve()、skb_put()、skb_push()以及 skb_pull()这几个函数来完成。

1. skb_reserve()

skb_reserve()在数据缓存区头部预留一定的空间，通常被用来在数据缓存区中插入协议首部或者在某个边界上对齐。它并没有把数据移出或移入数据缓存区，而只是简单地更新了数据缓存区的两个指针——分别指向负载起始和结尾的 data 和 tail 指针，图 3-15 展示了调用 skb_reserve()前后这两个指针的变化。

请注意：skb_reserve()只能用于空的 SKB，通常会在分配 SKB 之后就调用该函数，此时 data 和 tail 指针还一同指向数据区的起始位置，如图 3-15a 所示。例如，某个以太网设备驱动的接收函数，在分配 SKB 之后，向数据缓存区填充数据之前，会有这样的一条语句 skb_reserve(skb, 2)，这是因为以太网头长度为 14B，再加上 2B 就正好 16 字节边界对齐，所以大多数以太网设备都会在数据包之前保留 2B。

当 SKB 在协议栈中向下传递时，每一层协议都把 skb->data 指针向上移动，然后复制本层首部，同时更新 skb->len。这些操作都使用图 3-15 中所示的函数完成。

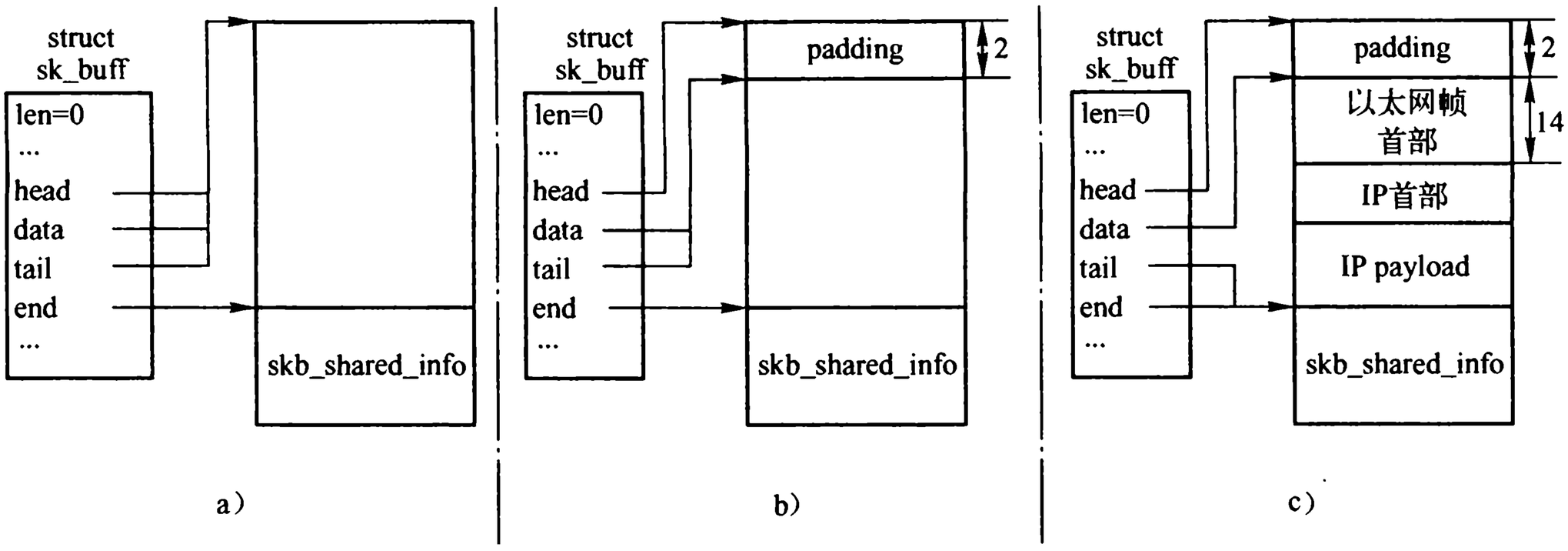

图 3-15 在接收过程中使用 skb_reserve()

a) 空的 SKB b) 头部预留 2 个字节 c) 复制以太网帧到 SKB

2. skb_push()

skb_push()在数据缓存区的前头加入一块数据，与 skb_reserve()类似，也并没有真正向数据缓存区中添加数据，而只是移动数据缓存区的头指针 data 和尾指针 tail。数据由其他函数复制到数据缓存区中。

函数执行步骤如下：

1）当 TCP 发送数据时，会根据一些条件，如 TCP 最大分段长度 MSS、是否支持聚合分散 I/O 等，分配一个 SKB。

2）TCP 需在数据缓存区的头部预留足够的空间，用来填充各层首部。MAX_TCP_HEADER 是各层首部长度的总和，它考虑了最坏的情况：由于 TCP 层不知道将要用哪个接口发送包，它为每一层预留了最大的首部长度，甚至还考虑了出现多个 IP 首部的可能性，因为在内核编译支持 IP over IP 的情况下，会遇到多个 IP 首部。

3）把 TCP 负载复制到数据缓存区。需要注意的是，图 3-16 只是一个例子，TCP 负载可能会被组织成其他形式，例如分片，在后续章节中将会看到一个分片的数据缓存区是什么样的。

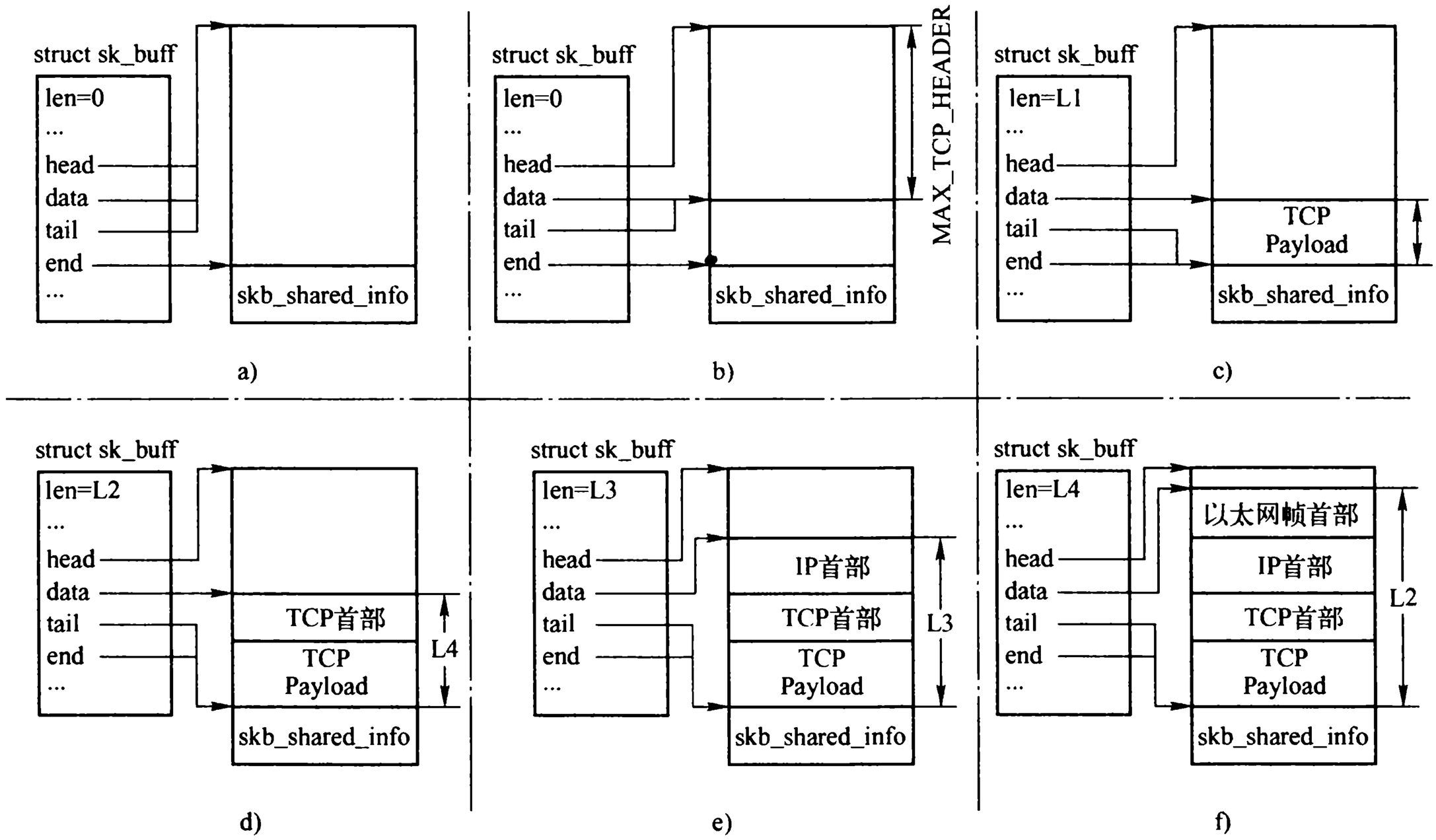

图 3-16　TCP 层向链路层传递时数据的填充过程

a) 空的 SKB　b) 在 SKB 的头部预留足够的空间　c) 复制 TCP 数据　d) 添加 TCP 首部　e) 添加 IP 首部　f) 添加以太网帧首部

4）TCP 层添加 TCP 首部。

5）SKB 传递到 IP 层，IP 层为数据包添加 IP 首部。

6）SKB 传递到链路层，链路层为数据包添加链路层首部。

3. skb_put()

skb_put()修改指向数据区末尾的指针 tail，使之往下移 len 字节，即使数据区向下扩大 len 字节，并更新数据区长度 len。调用 skb_put()前后，SKB 结构变化如图 3-17 所示。

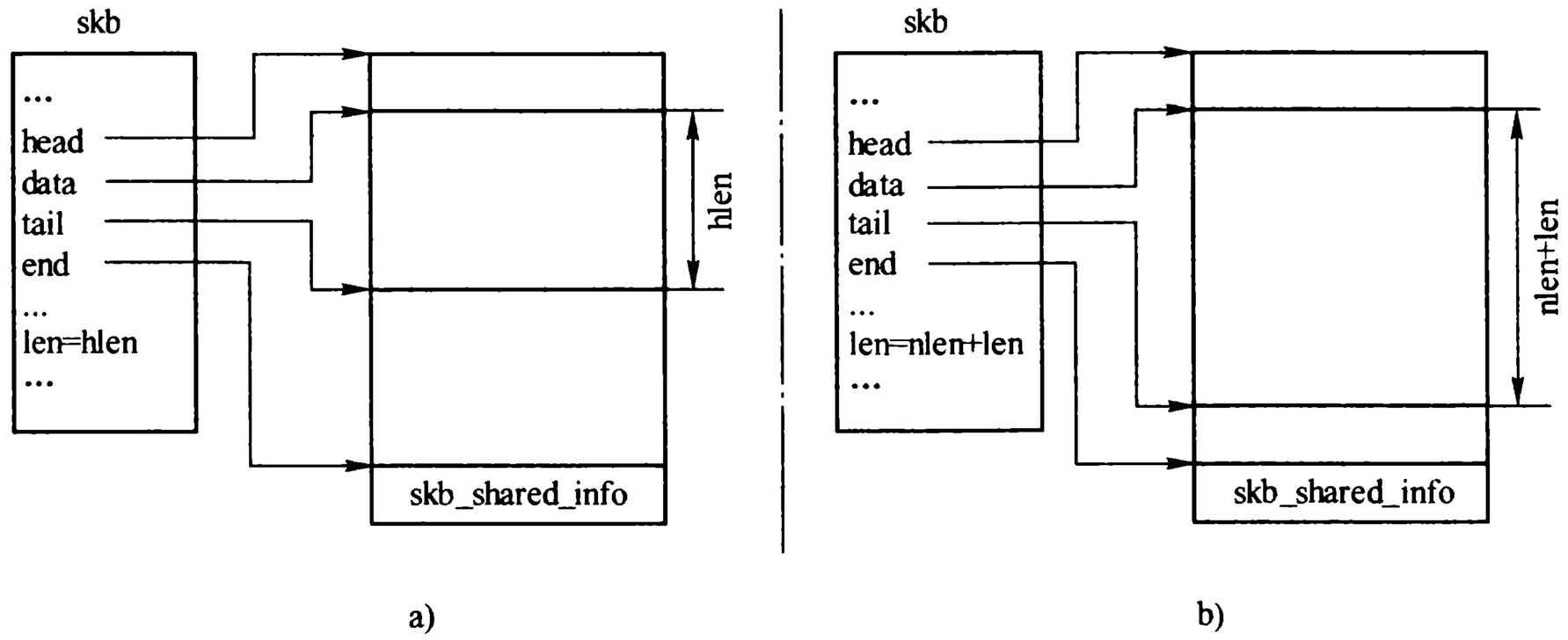

图 3-17　skb_put()示意

a) 调用前　b) 调用后

4. skb_pull()

skb_pull()通过将 data 指针往下移动，在数据区首部忽略 len 字节长度的数据，通常用于接收到数据包后在各层间由下往上传递时，上层忽略下层的首部。调用 skb_pull()前后，SKB 结

构变化如图 3-18 所示。

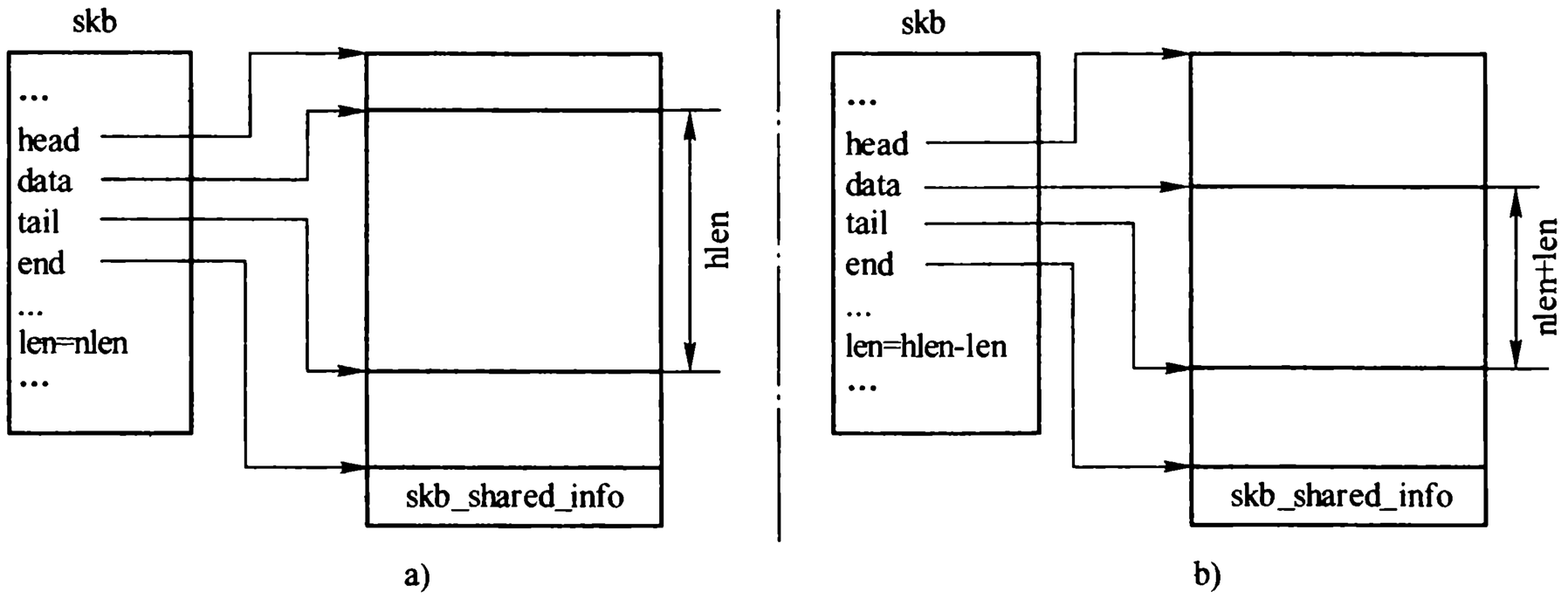

图 3-18　skb_pull()示意

a) 调用前　b) 调用后

3.4.5　克隆和复制 SKB

1．skb_clone()

如果一个 SKB 会被不同的用户独立操作，而这些用户可能只是修改 SKB 描述符中的某些字段值，如 h、nh，则内核没有必要为每个用户复制一份完整的 SKB 描述及其相应的数据缓存区，而会为了提高性能，只作克隆操作。克隆过程只复制 SKB 描述符，同时增加数据缓存区的引用计数，以免共享数据被提前释放。完成这些功能的是 skb_clone()。一个使用包克隆的场景是，一个接收包程序要把该包传递给多个接收者，例如包处理函数或者一个或多个网络模块。原始的及克隆的 SKB 描述符的 cloned 值都会被设置为 1，克隆 SKB 描述符的 users 值置为 1，这样在第一次释放时就会释放掉。同时将数据缓存区引用计数 dataref 递增 1，因为又多了一个克隆 SKB 描述符指向它。图 3-19 演示的是已克隆的 SKB。

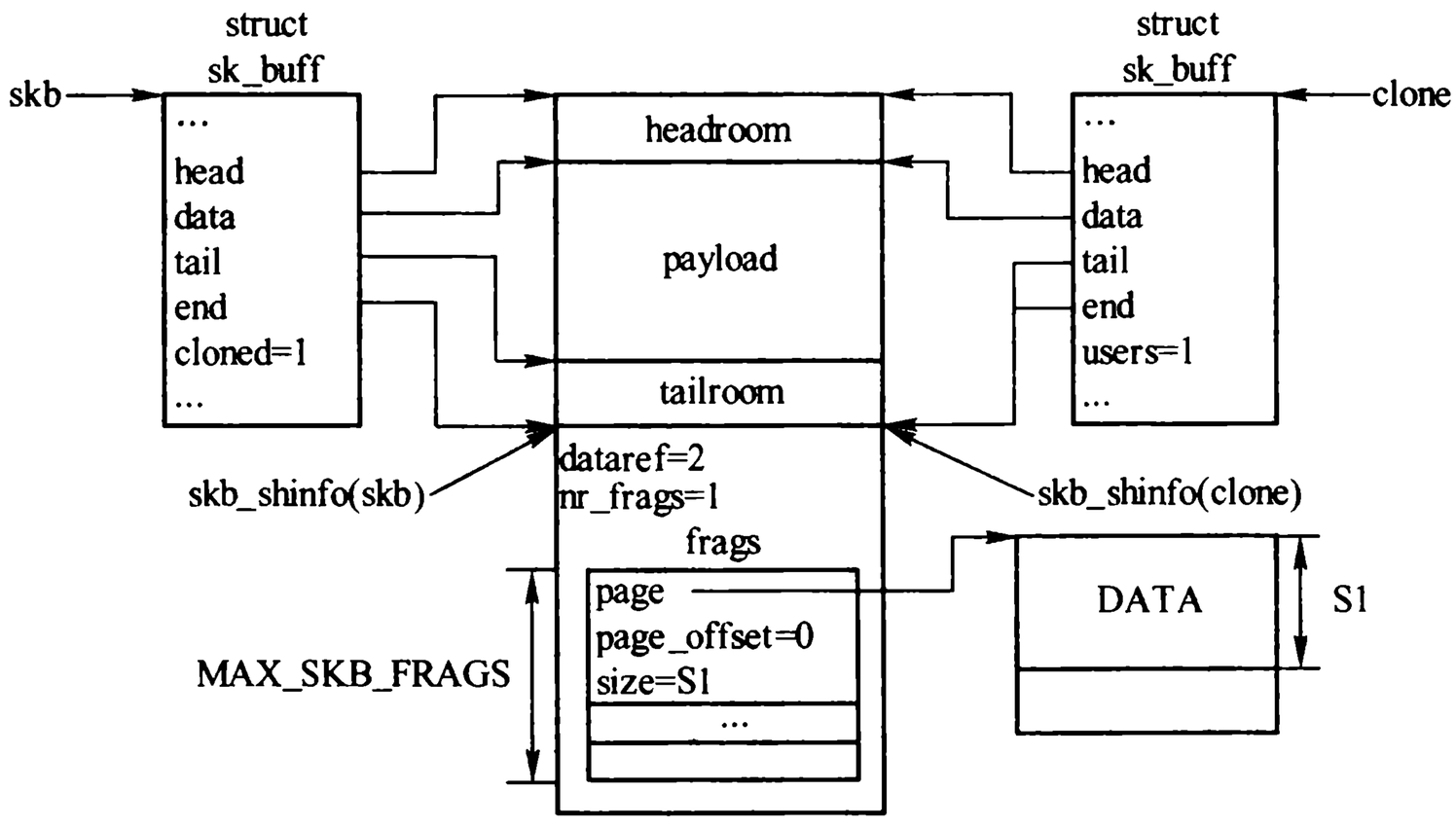

图 3-19　克隆后的 SKB

图 3-19 所示是一个存在聚合分散 I/O 缓存区的例子，这个数据缓存区的一些数据保存在分片结构数组 frags 中。skb_share_check()用来检查 SKB 引用计数 users，如果该字段表明 SKB 是被共享的，则克隆一个新的 SKB。一个 SKB 被克隆后，该 SKB 数据缓存区中的内容就不能再被修改，这也意味着访问数据的函数没有必要加锁。skb_cloned()可以用来测试 skb 的克隆状态。

```
432 struct sk_buff *skb_clone(struct sk_buff *skb, gfp_t gfp_mask)
```

```
433 {
434     struct sk_buff *n;
435
436     n = skb + 1;
437     if (skb->fclone == SKB_FCLONE_ORIG &&
438         n->fclone == SKB_FCLONE_UNAVAILABLE) {
439         atomic_t *fclone_ref = (atomic_t *) (n + 1);
440         n->fclone = SKB_FCLONE_CLONE;
441         atomic_inc(fclone_ref);
442     } else {
443         n = kmem_cache_alloc(skbuff_head_cache, gfp_mask);
444         if (!n)
445             return NULL;
446         n->fclone = SKB_FCLONE_UNAVAILABLE;
447     }
448
449 #define C(x) n->x = skb->x
450
451     n->next = n->prev = NULL;
452     n->sk = NULL;
453     C(tstamp);
454     C(dev);
455     C(h);
456     C(nh);
457     C(mac);
458     C(dst);
459     dst_clone(skb->dst);
460     C(sp);
461 #ifdef CONFIG_INET
462     secpath_get(skb->sp);
463 #endif
464     memcpy(n->cb, skb->cb, sizeof(skb->cb));
465     C(len);
466     C(data_len);
467     C(csum);
468     C(local_df);
469     n->cloned = 1;
470     n->nohdr = 0;
471     C(pkt_type);
472     C(ip_summed);
473     C(priority);
474 #if defined(CONFIG_IP_VS) || defined(CONFIG_IP_VS_MODULE)
475     C(ipvs_property);
476 #endif
477     C(protocol);
478     n->destructor = NULL;
479     C(mark);
480 #ifdef CONFIG_NETFILTER
481     C(nfct);
482     nf_conntrack_get(skb->nfct);
483     C(nfctinfo);
484 #if defined(CONFIG_NF_CONNTRACK) || defined(CONFIG_NF_CONNTRACK_MODULE)
485     C(nfct_reasm);
486     nf_conntrack_get_reasm(skb->nfct_reasm);
487 #endif
488 #ifdef CONFIG_BRIDGE_NETFILTER
489     C(nf_bridge);
490     nf_bridge_get(skb->nf_bridge);
491 #endif
492 #endif /*CONFIG_NETFILTER*/
```

```
493 #ifdef CONFIG_NET_SCHED
494     C(tc_index);
495 #ifdef CONFIG_NET_CLS_ACT
496     n->tc_verd = SET_TC_VERD(skb->tc_verd,0);
497     n->tc_verd = CLR_TC_OK2MUNGE(n->tc_verd);
498     n->tc_verd = CLR_TC_MUNGED(n->tc_verd);
499     C(input_dev);
500 #endif
501     skb_copy_secmark(n, skb);
502 #endif
503     C(truesize);
504     atomic_set(&n->users, 1);
505     C(head);
506     C(data);
507     C(tail);
508     C(end);
509
510     atomic_inc(&(skb_shinfo(skb)->dataref));
511     skb->cloned = 1;
512
513     return n;
514 }
```

436-438 由 fclone 标志来决定从哪个缓冲池中分配 SKB 描述符。如果紧邻的两个父子 SKB 描述符，前一个的 fclone 为 SKB_FCLONE_ORIG，后一个的 fclone 为 SKB_FCLONE_UNAVAILABLE，则说明这两个 SKB 描述符是从 skbuff_fclone_cache 缓冲池中分配的，且父 SKB 描述符还没有被克隆，即子 SKB 描述符还是空的。否则即从 skbuff_head_cache 缓冲池中分配一个新的 SKB 来用于克隆。

451-508 将父 SKB 描述符各字段值赋给子 SKB 描述符的对应字段。

504 设置子 SKB 描述符引用计数 users 为 1。

510 递增父 SKB 描述符中的数据区引用计数 skb_shared_info 结构的 dataref。

511 设置父 SKB 描述符的成员 cloned 为 1，表示该 SKB 已被克隆。

2. pskb_copy()

当一个函数不仅要修改 SKB 描述符，而且还要修改数据缓存区中的数据时，就需要同时复制数据缓存区。在这种情况下，程序员有两个选择。如果所修改的数据在 skb->head 和 skb->end 之间，可使用 pskb_copy()来复制这部分数据，如图 3-20 所示。

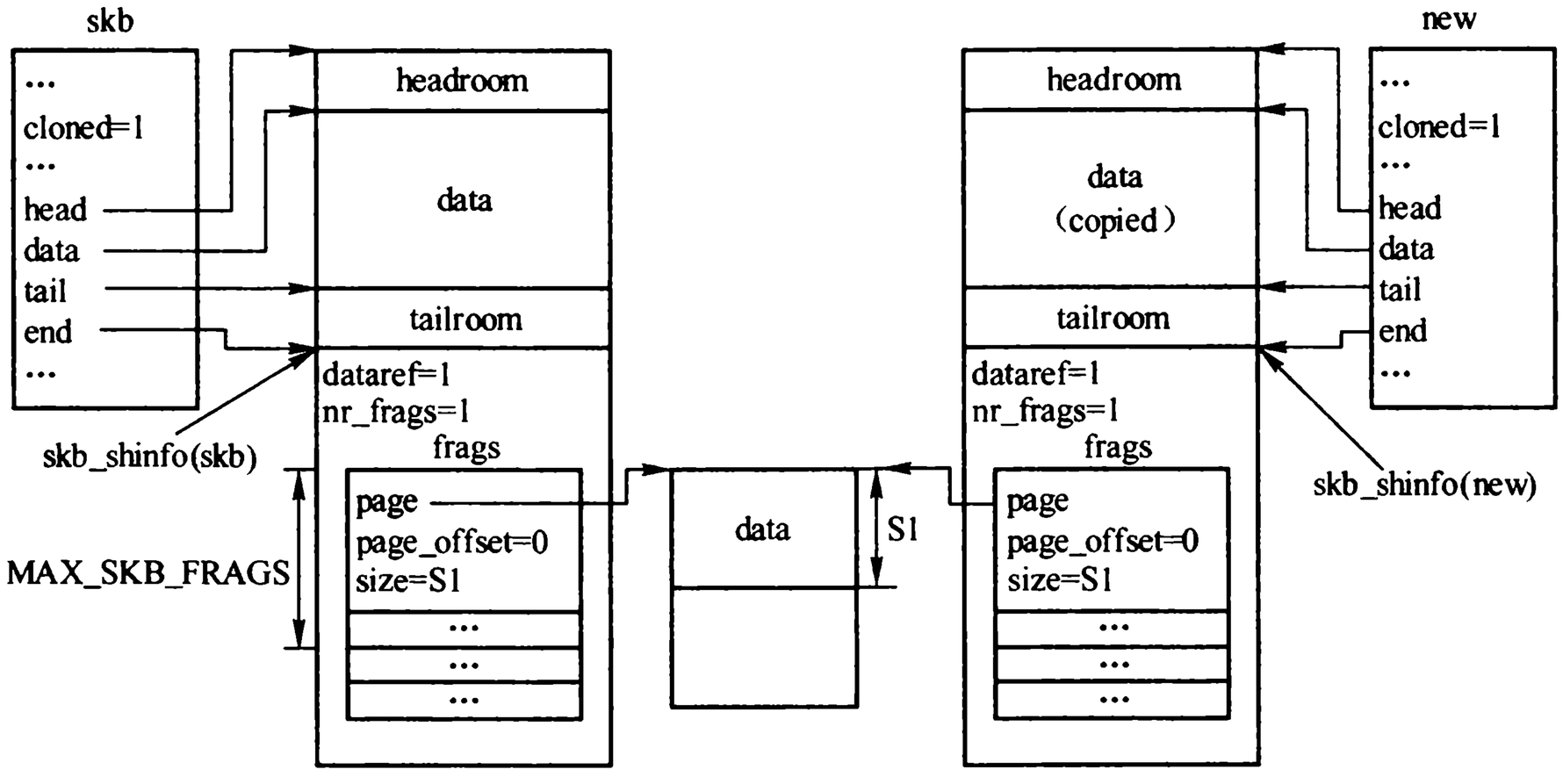

图 3-20 pskb_copy()示意

3. skb_copy()

如果同时需要修改聚合分散 I/O 存储区中的数据，就必须使用 skb_copy()，如图 3-21 所示。从前面的章节中看到，skb_shared_info 结构中也包含一个 SKB 链表 frag_list。该链表在 pskb_copy()和 skb_copy()中的处理方式与 frags 数组处理方式相同。

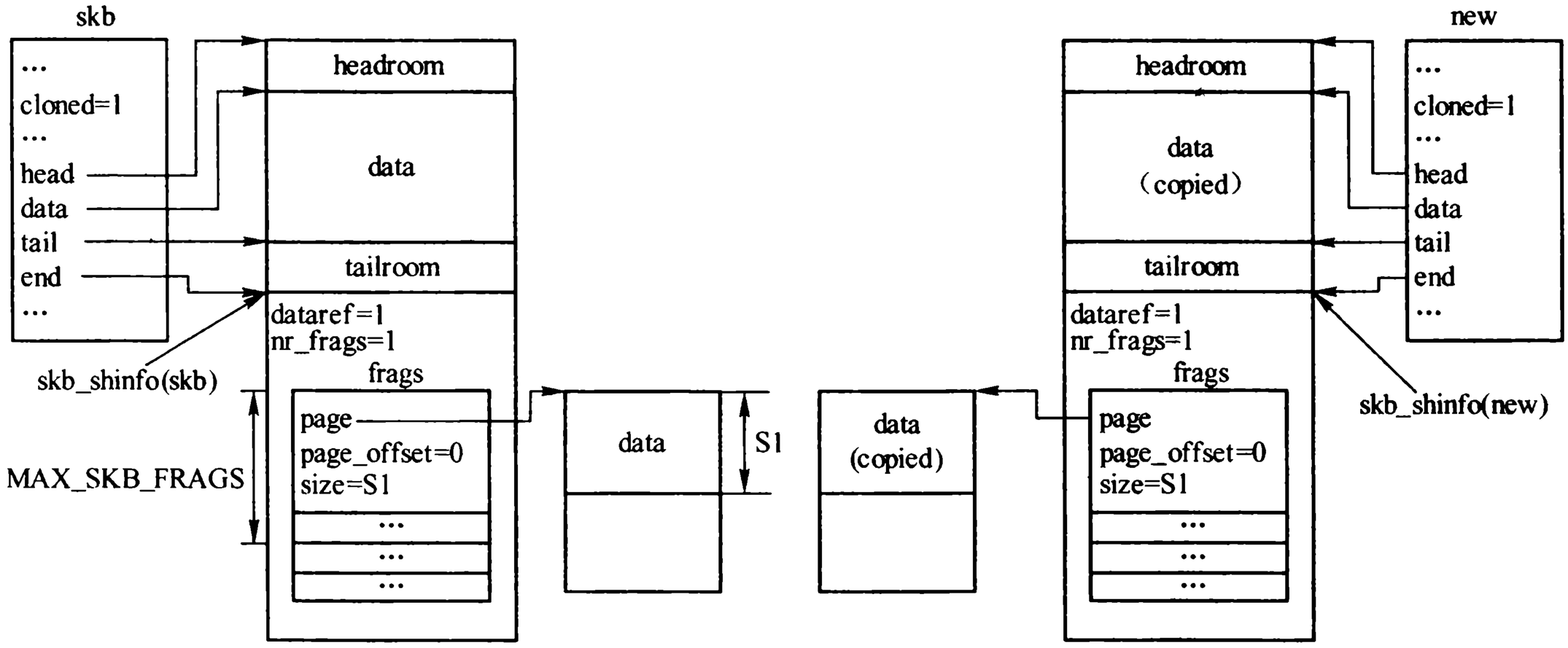

图 3-21　skb_copy()示意

```
587 struct sk_buff *skb_copy(const struct sk_buff *skb, gfp_t gfp_mask)
588 {
589     int headerlen = skb->data - skb->head;
590     /*
591      *    Allocate the copy buffer
592      */
593     struct sk_buff *n = alloc_skb(skb->end - skb->head + skb->data_len,
594                                   gfp_mask);
595     if (!n)
596         return NULL;
597
598     /* Set the data pointer */
599     skb_reserve(n, headerlen);
600     /* Set the tail pointer and length */
601     skb_put(n, skb->len);
602     n->csum        = skb->csum;
603     n->ip_summed = skb->ip_summed;
604
605     if (skb_copy_bits(skb, -headerlen, n->head, headerlen + skb->len))
606         BUG();
607
608     copy_skb_header(n, skb);
609     return n;
610 }
```

589-599 分配一个新的 SKB，即包括 SKB 描述符和数据缓存区，然后在指针 head 和 data 之间预留源数据缓存区 headroom 长度的空间。

601 将新 SKB 的 tail 指针和数据区长度 len 设置为与源 SKB 的一样。

605-608 复制数据。

在讨论本书中不同主题时，有时会强调某个特定函数需要克隆或者复制一个 SKB。在决定克隆或复制 SKB 时，各子系统程序员不能预测其他内核组件是否需要使用 SKB 中的原始数据。内核是模块化的，其状态变化是不可预测的，每个子系统都不知道其他子系统是如何操作数据缓存区的。因此，内核程序员需要记录各子系统对数据缓存区的修改，并且在修改数据缓存区前，复制一个新的数据缓存区，以免其他子系统需使用数据缓存区原始数据时出现错误。

3.4.6 链表管理函数

在对 SKB 链表的操作中，为了防止被其他异步操作打断，在操作前都必须先获取 SKB 头结点中（sk buff head 结构）的自旋锁，然后才能访问队列中的元素。以下是 SKB 链表操作函数。

（1）skb_queue_head_init()

初始化 SKB 链表的头结点，并创建一个空的 SKB 链表。

（2）skb_queue_head()和 skb_queue_tail()

将一个 SKB 加入到队列的头首部和尾部。__skb_queue_head()以 SKB 链表头结点 sk_buff_head 为 prev 参数调用__skb_queue_after()。

（3）skb_dequeue()和 skb_dequeue_tail()

从队列的首部和尾部取下一个 SKB。实际上，前一个函数的名字似乎更应该是 skb_dequeue_head，才保持和其他函数的名称风格一致。

（4）skb_queue_purge()

清空一个 SKB 链表。

（5）skb_queue_walk 宏

定义为一个 for 语句，用来按顺序遍历 SKB 链表中的每一个元素。

3.4.7 添加或删除尾部数据

1. skb_add_data()

skb_add_data()将指定用户空间的数据添加到 SKB 的数据缓存区的尾部，操作过程如图 3-22 所示。如果成功则返回 0，否则返回相应的错误码。参数 skb 为待添加数据的 SKB；from 为待添加的数据源，指向在用户空间的存储缓存区；copy 为待添加数据的长度。

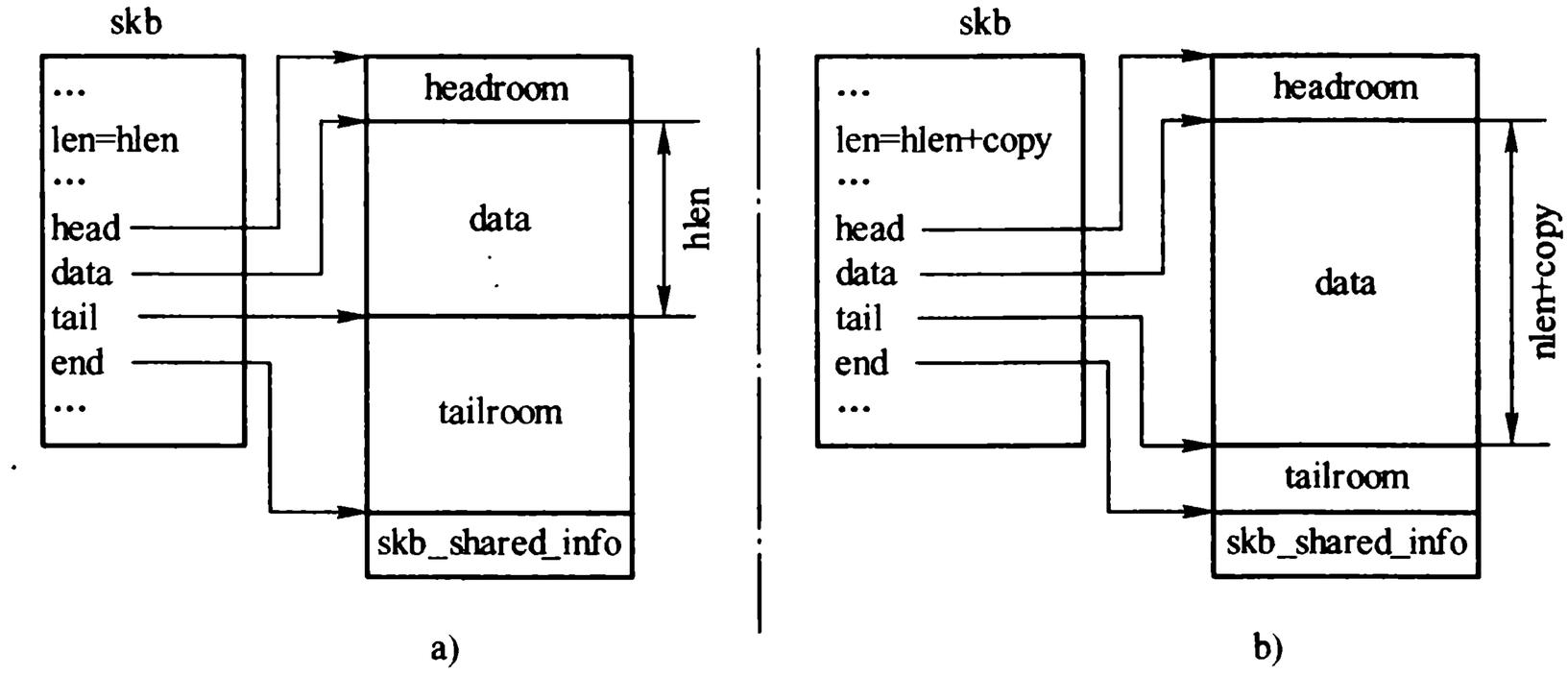

图 3-22 skb_add_data()示意

a) 操作前 b) 操作后

2. skb_trim()

skb_trim()根据指定长度删除 SKB 的数据缓存区尾部的数据，如果新长度大于当前长度，则不作处理，操作过程如图 3-23 所示。调用该函数的前提条件是，待操作的 SKB 的数据必须是线性存储的。参数 skb 为待操作的 SKB；len 为删除尾部数据后剩余的长度。

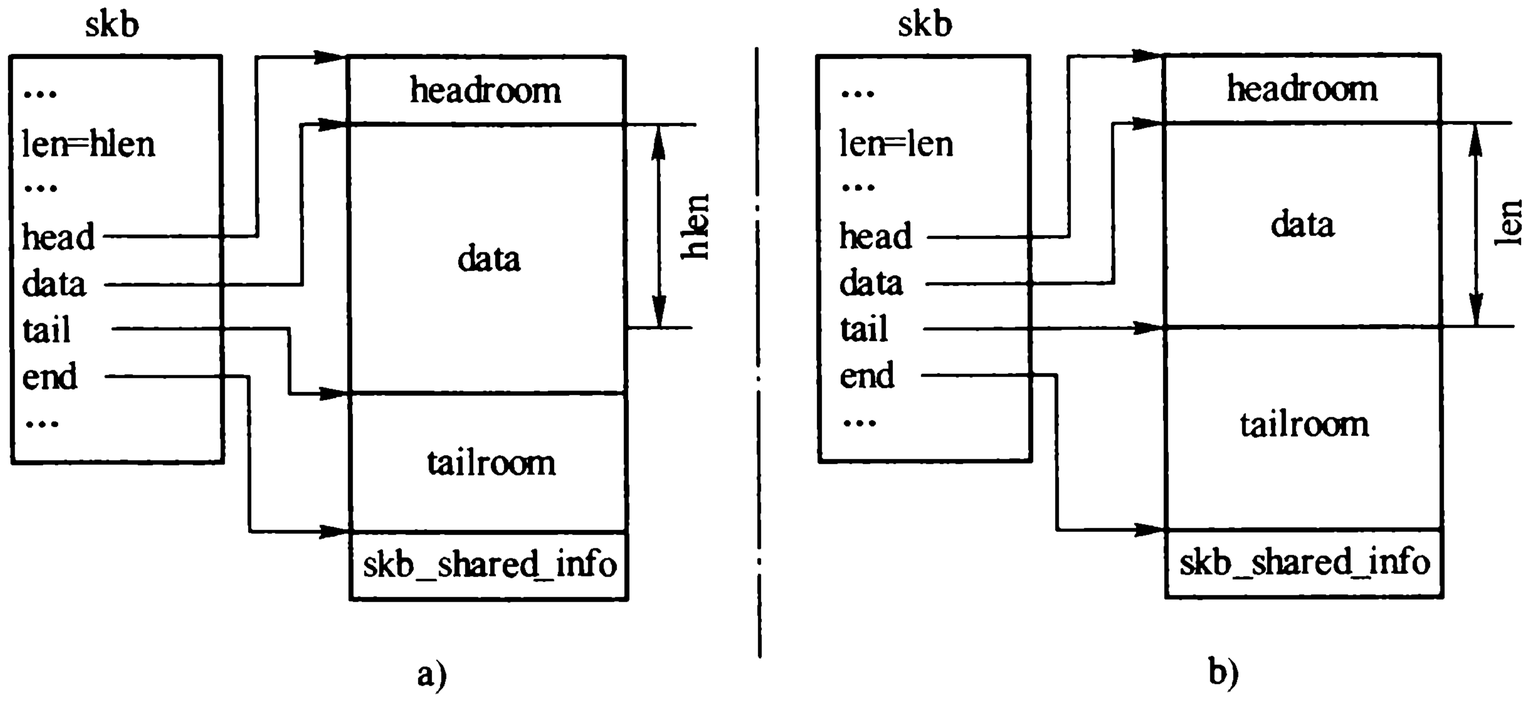

图 3-23　skb_trim()示意

a) 操作前　b) 操作后

3. pskb_trim()

pskb_trim()与 skb_trim()功能类似，也是根据指定长度删除 SKB 尾部的数据。不同的是，pskb_trim()是 skb_trim()的功能超集，不仅可以处理线性数据的 SKB，还可以处理非线性的 SKB。线性数据的处理过程与 skb_trim()相同，非线性数据操作过程如图 3-24 和图 3-25 所示。

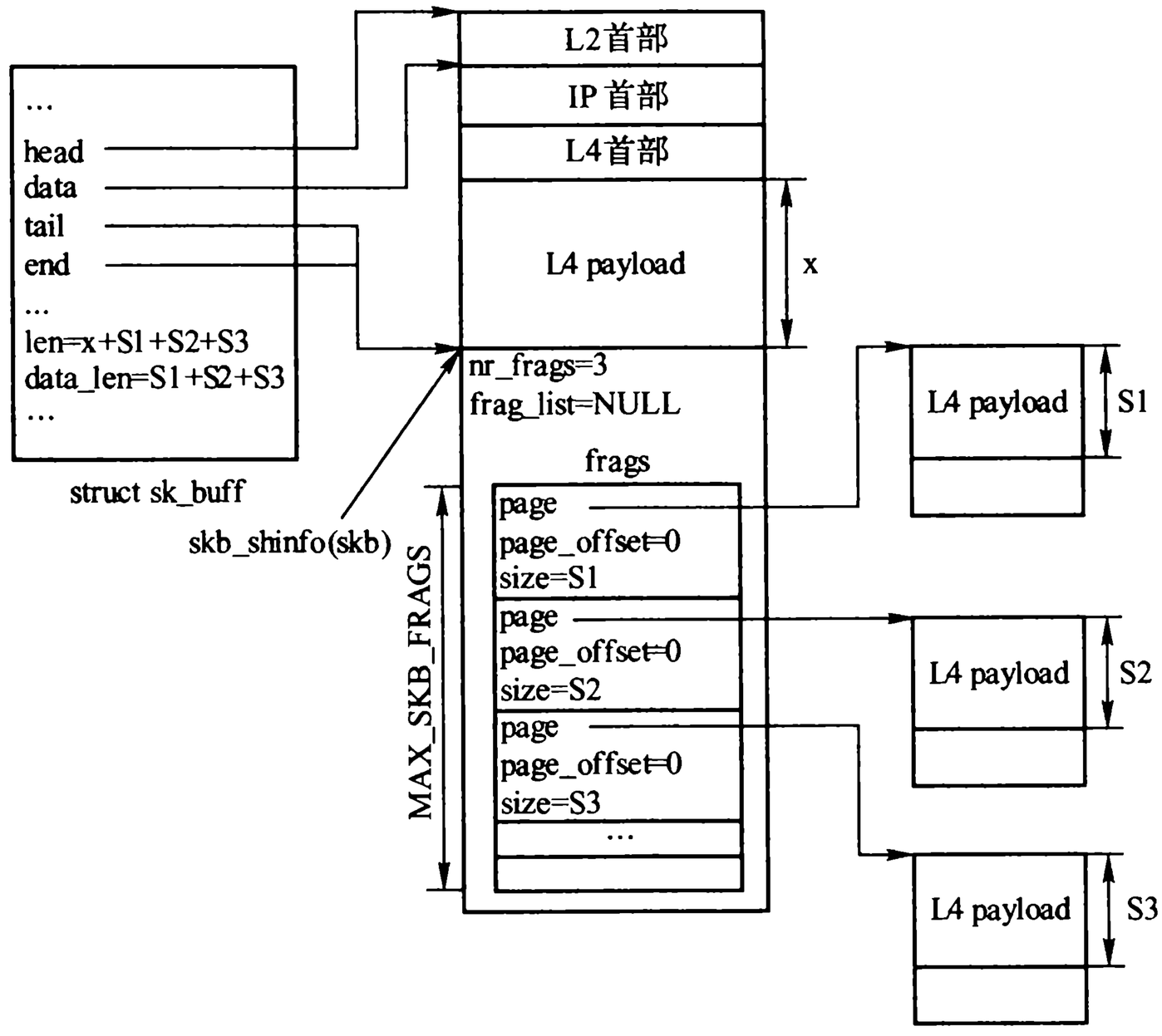

图 3-24　pskb_trim()示意——操作前

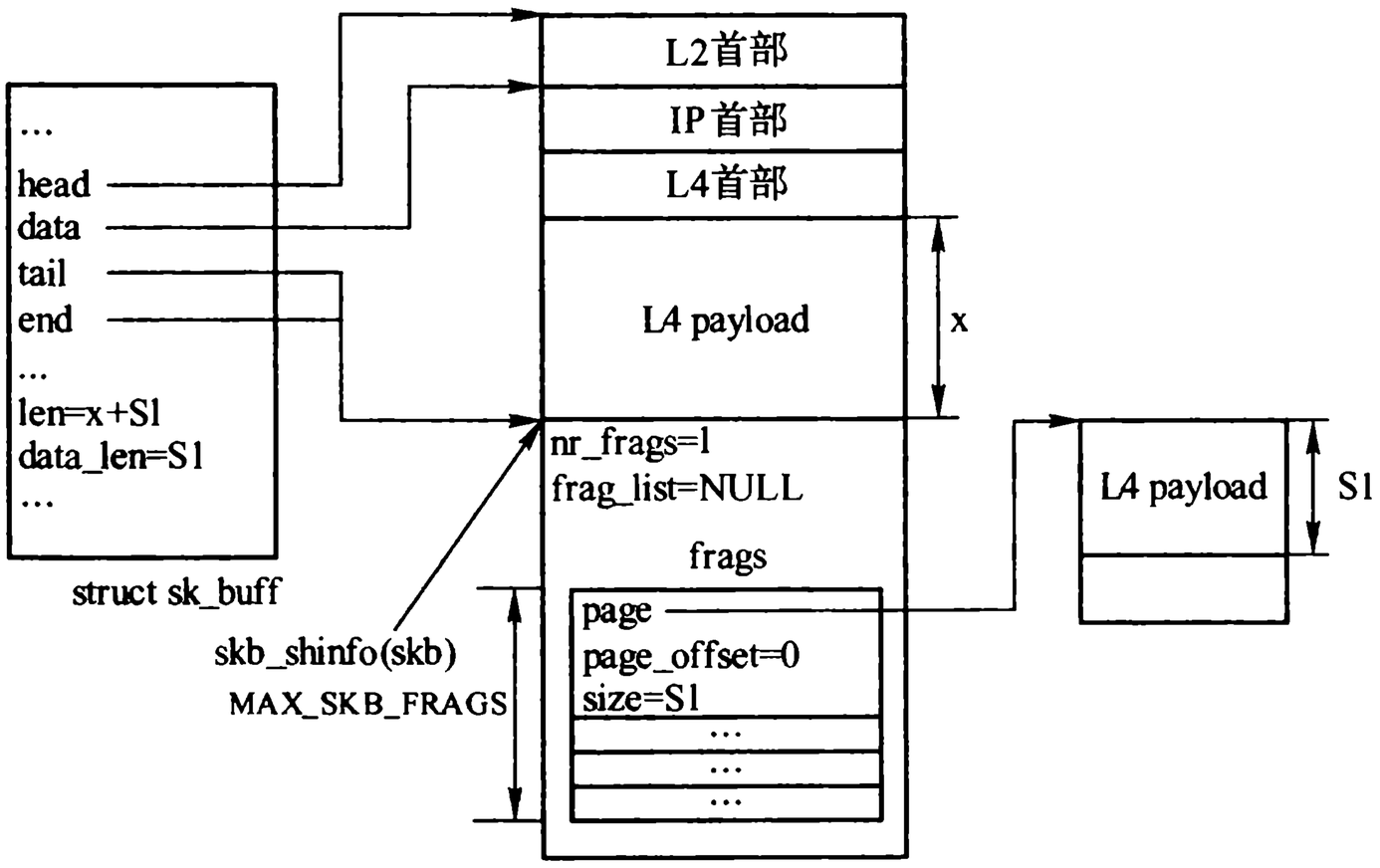

图 3-25　pskb_trim()示意——操作后

3.4.8　拆分数据：skb_split()

skb_split()可根据指定长度拆分 SKB，使得原 SKB 中的数据长度为指定的长度，而剩下的数据保存到拆分得到的 SKB 中。

由于 SKB 支持线性数据存储和聚合分散 I/O 存储，因此在拆分过程中需要考虑这些情况。当拆分数据的长度小于线性数据长度时比较容易处理，直接拆分线性数据区即可。如图 3-26 和图 3-27 所示，拆分长度 LEN 小于 hlen。

参数 skb 为待拆分的 SKB；skb1 为拆分得到的 SKB；len 为拆分后原 SKB 中数据的长度。

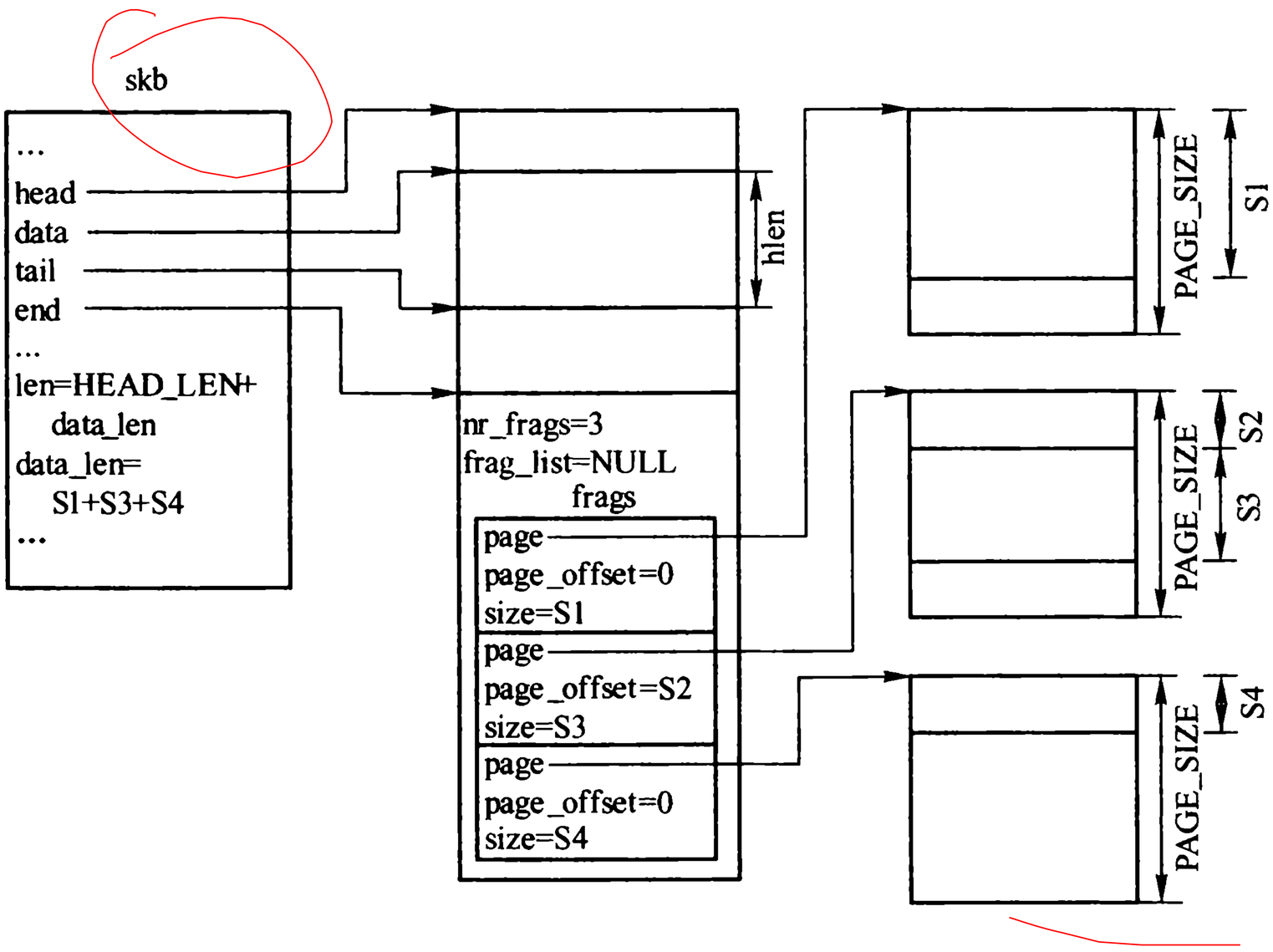

图 3-26　skb_split()示意——拆分前（拆分长度不大于线性数据长度）

若拆分数据的长度大于线性数据长度，则需要拆分非线性区域中的数据。如图 3-28 和图 3-29 所示，拆分长度 LEN 大于 hlen 并且 LEN 小于 hlen+S1。

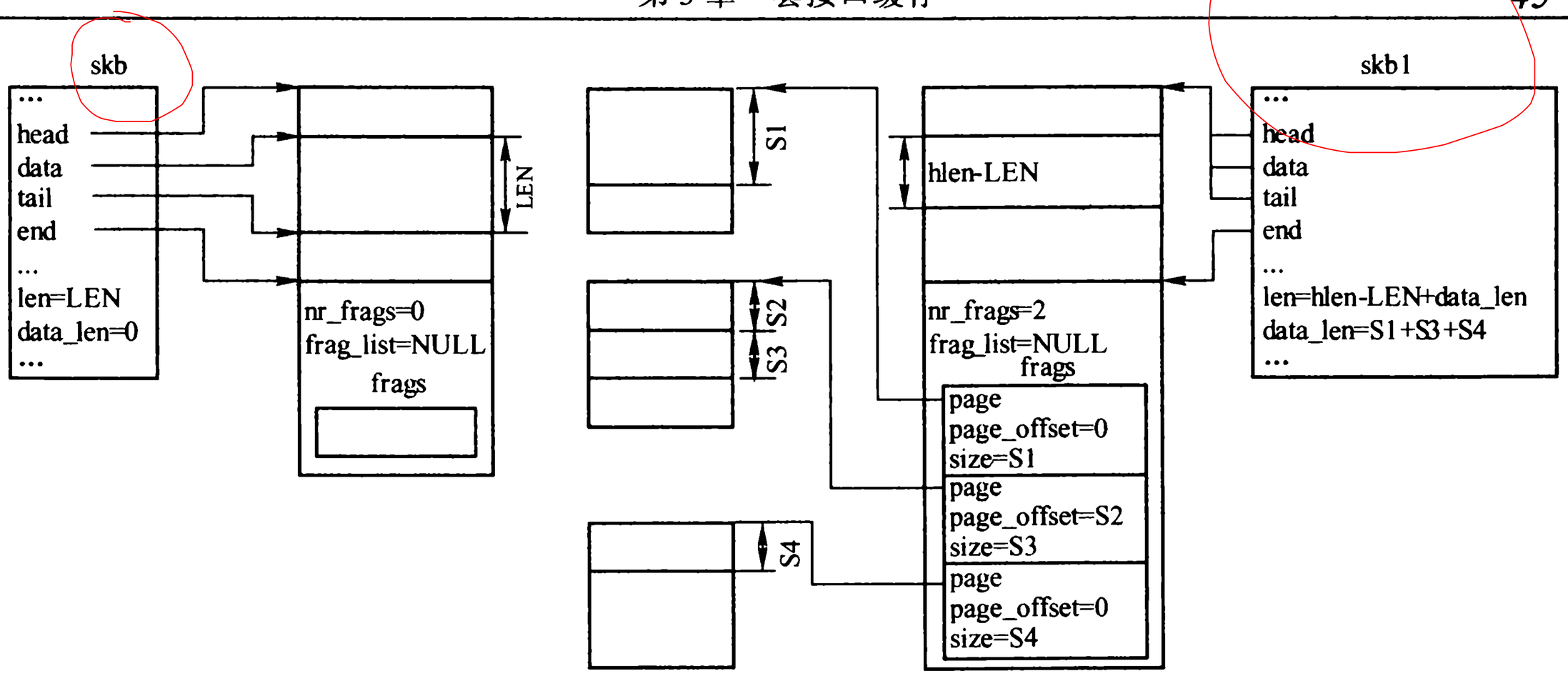

图 3-27　skb_split()示意——拆分后（拆分长度不大于线性数据长度）

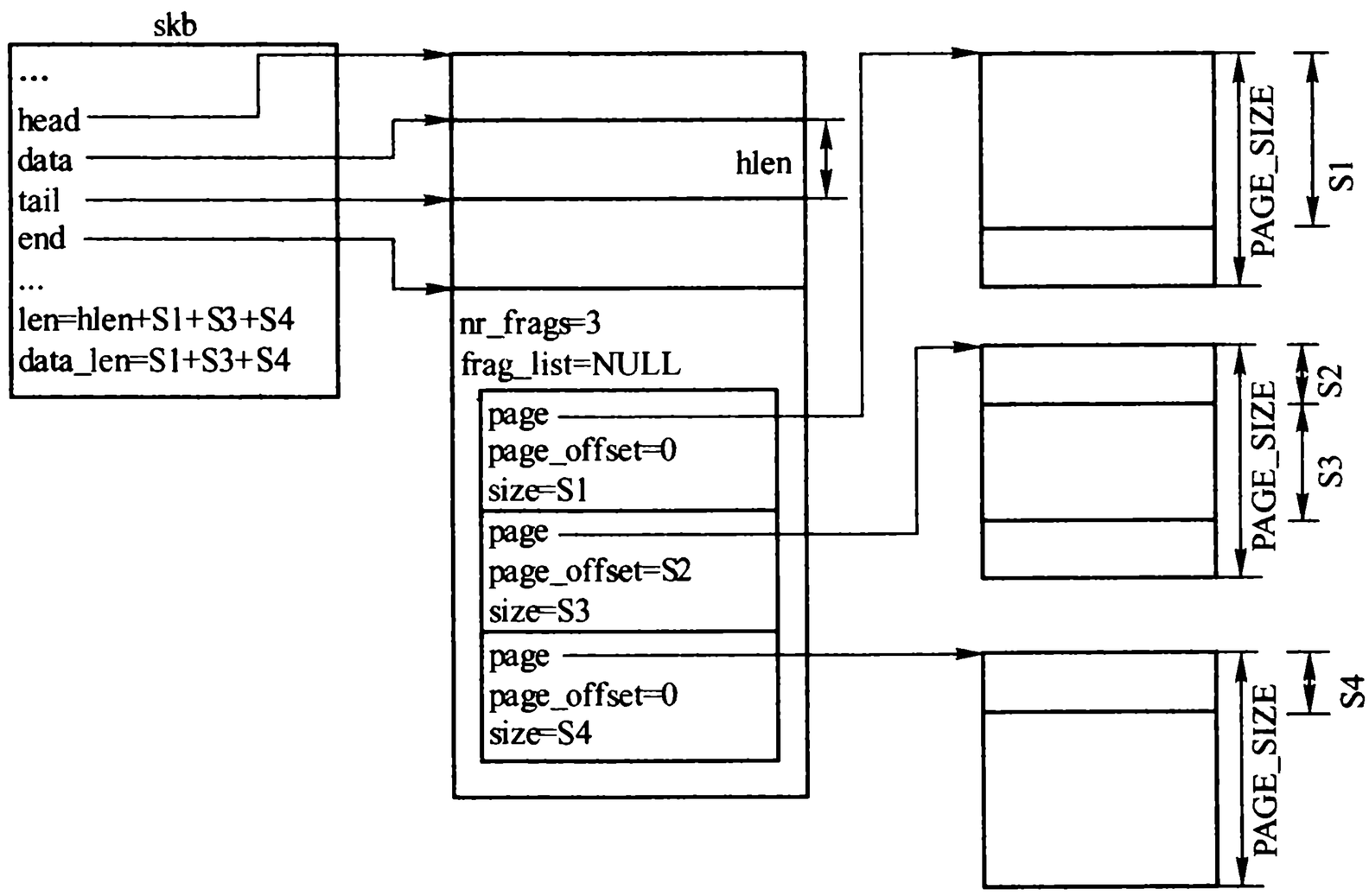

图 3-28　skb_split()示意——拆分前（拆分长度大于线性数据长度）

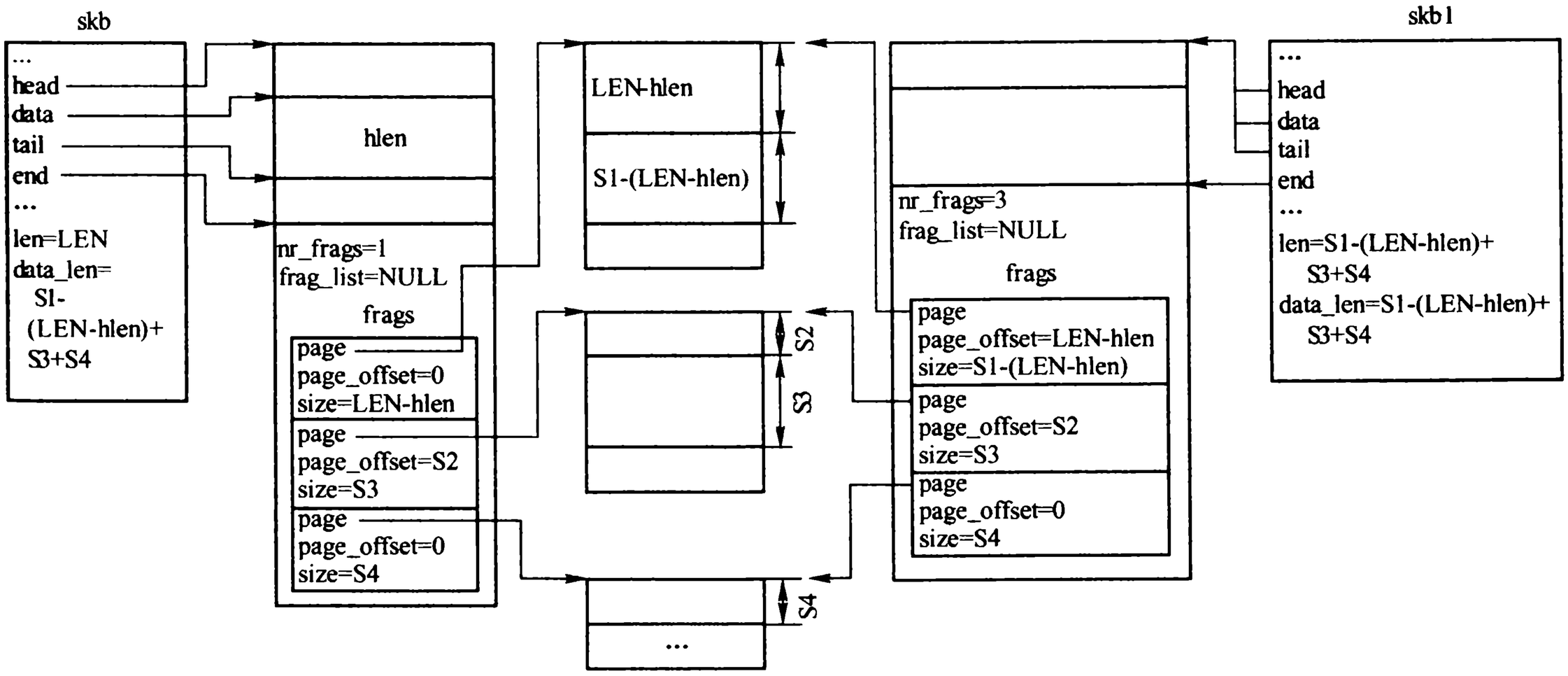

图 3-29　skb_split()示意——拆分后（拆分长度大于线性数据长度）

3.4.9 重新分配 SKB 的线性数据区：pskb_expand_head()

pskb_expand_head()根据指定长度重新扩展 headroom 和 tailroom 的空间，操作过程如图 3-30 所示。由于原先的线性空间已经固定，因此需要重新分配空间来扩展线性空间。该函数只是扩展数据区，不会修改 SKB 本身，且待扩展的 SKB 的引用计数 users 必须为 1。如果返回 0 表示扩展成功，否则原线性数据区不变。

参数 skb 为待操作的 SKB；nhead 为待扩展的 headroom 空间的长度；ntail 为待扩展的 tailroom 空间的长度；gfp_mask 为分配内存的方式及优先级。

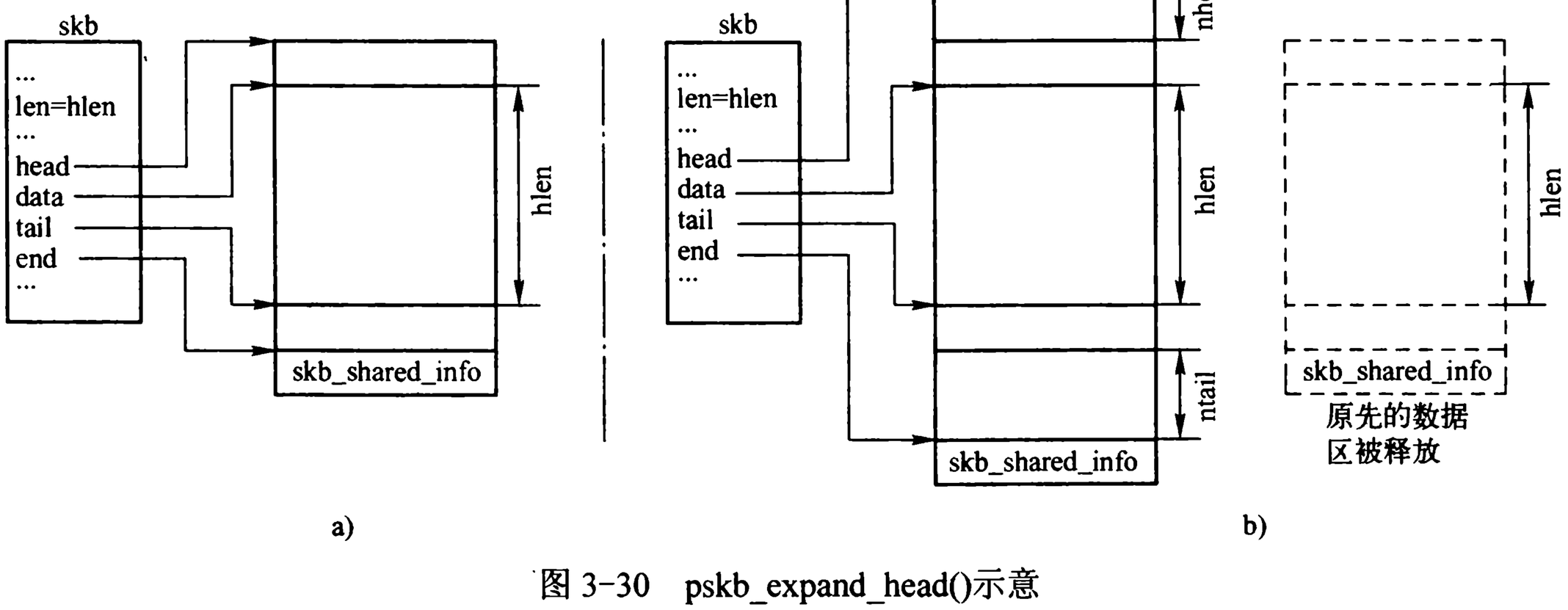

图 3-30 pskb_expand_head()示意

a) 操作前 b) 操作后

3.4.10 其他函数

（1）pskb_may_pull()

检测 SKB 中的数据是否有指定的长度。

（2）skb_queue_empty()

检测指定的 SKB 队列是否为空。

（3）skb_realloc_headroom()

根据一个指定的 SKB 得到一个新的 SKB，并确保新的 SKB 存在指定的 headroom 空间。

（4）skb_get()

引用并返回一个 SKB。

（5）skb_shared()

检测指定 SKB 是否被多次引用。

（6）skb_share_check()

检查并返回 SKB，当被检查的 SKB 被引用多次，则克隆此 SKB，并返回克隆得到的 SKB。

（7）skb_unshare()

检查并返回 SKB，当被检查的 SKB 被克隆时，则复制此 SKB，并返回复制得到的 SKB。

（8）skb_orphan()

使 SKB 不属于任何传输控制块。

（9）skb_cow()

确保 SKB 存在指定的 headroom 空间。如果不足，会重新分配。

（10）skb_pagelen()

获取 SKB 中线性数据区和 SG 类型的聚合分散 I/O 分片中数据的长度（不包括 frag_list 上数据的长度）。

第 4 章　网络模块初始化

4.1　引言

对于内核模块，不仅需要了解特定的模块实现了什么功能，还需要知道实现这些功能的代码何时被调用。内核中，各个子系统的初始化是一个必需而基本的过程，这些过程由内核根据自己的模式来处理。了解初始化体系结构的过程是非常重要的，有助于理解网络协议栈的核心模块。初始化过程包括网络设备驱动程序是如何初始化的以及各个协议是如何初始化的。

本章的目的在于展示内核是怎样处理初始化内核模块的，既包括静态连接到内核的模块，也包括动态加载到内核的模块，特别是网络设备驱动。本章涉及以下文件：

- include/linux/init.h，初始化相关的宏定义。
- include/asm-generic/vmlinux.lds.h，编译链接相关的宏定义。
- init/main.c，启动时的高级初始化。
- net/core/dev.c，网络设备注册、输入和输出等接口。
- drivers/net/e100.c，e100 驱动程序。

4.2　网络模块初始化顺序

Linux 内核的初始化过程之所以复杂，是因为它同时支持静态加载和动态加载内核模块。动态加载内核模块提高了系统的灵活性，但也因此需要考虑更多的方面。设备驱动程序可以静态地编译到内核中，也可以作为一个内核模块动态地装载和卸载。此外，由于支持热插拔设备，因此还需考虑在热插拔情况下的初始化工作。

系统启动初始化时，一旦进入 start_kernel()，则说明低级初始化已完成，接下来是对各种设备和子系统的初始化。图 4-1 所示为网络模块初始化流程中的函数调用关系。

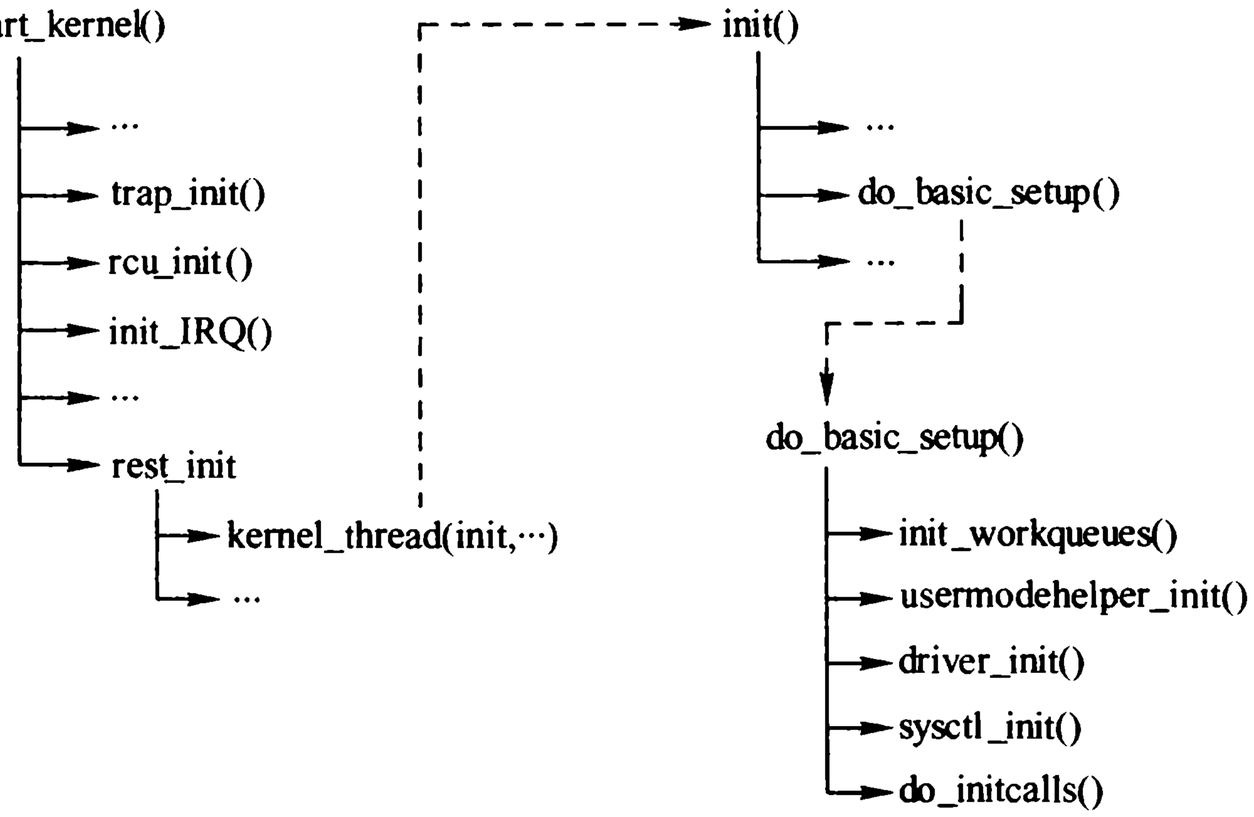

图 4-1　内核中网络模块初始化的函数调用关系

在内核的初始化过程中，初始化了很多模块，如图中的中断模块（init_IRQ()）等。最后会启动 init 内核线程继续进行初始化，完成初始化之后，init 内核线程将蜕变为用户进程。有关网络的初始化过程主要由 sysctl_init()和 do_initcalls()完成，前者主要完成对 sysctl 的初始化注册过程，而后者的功能相对复杂，不只是对网络模块进行初始化，只要运用了 initcall 技术的模块都会被初始化。

内核模块代码可以静态地连接到内核映像文件中，也可以动态地加载到内核空间中。通常，设备驱动程序和功能扩展模块都是在需要时动态加载的，有些协议族也可作为内核模块动态地加载，如 IPv6、UNIX 协议族等，而作为基本内核模块的 IPv4 则不能动态加载。

每个内核模块都必须提供两个函数：init_module()和 cleanup_module()，前者在内核加载模块时被调用来初始化模块，后者在内核卸载模块时被调用来释放该模块的资源。

在 2.4 版本以后的内核中提供了两个宏：module_init 和 module_exit，通过这两个宏允许程序员随意命名初始化函数和卸载函数。下面一个例子是 e100 网络设备驱动（drivers/net/e100.c）。

```
177  module_param(debug, int, 0);
178  module_param(eeprom_bad_csum_allow, int, 0);
179  MODULE_PARM_DESC(debug, "Debug level (0=none,...,16=all)");
180  MODULE_PARM_DESC(eeprom_bad_csum_allow, "Allow bad eeprom checksums");
...
2874 static int __init e100_init_module(void)
2875 {
.... ...
2881 }
2882
2883 static void __exit e100_cleanup_module(void)
2884 {
.... ...
2886 }
2887
2888 module_init(e100_init_module);
2889 module_exit(e100_cleanup_module);
```

在上述代码中还出现了两个宏：__init 和__exit。

对每个模块来说，这些宏允许内核在后台决定哪些代码需要包含在内核映像中，哪些代码不必包含在内核映像中，又有哪些代码仅仅在初始化时执行，等等。这样就省去了程序员在每个模块都要复制相同逻辑的麻烦。

很明显，正如上面例子所示，这些宏允许程序员替换条件编译指示符，它们必须提供如下两个服务：

- 定义加载该内核组件时所需执行的函数。
- 在各模块的初始化函数间定义某种顺序，以使内核组件之间相互依赖。

4.3　优化基于宏的标记

虽然模块的初始化比较复杂，但其实现的构思是非常巧妙的，提供给内核程序员的接口比较简单，细节部分都在编译期完成。对模块的初始化，一般通过 module_init 宏来登记初始化函

数。前面说过，设备驱动可以静态编译到内核中，也可以作为一个内核模块动态装载和卸载，这两种方法的初始化过程是不一样的，而 module_init 宏可以自动根据编译条件来选择不同的初始化方法。

对于静态编译到内核中的驱动程序模块，module_init 宏指示编译器把模块的入口函数放到一个特殊的段中，在内核初始化过程中，由 do_initcalls()调用这些函数从而达到初始化的效果。module_init 宏的定义相对复杂。静态编译条件下的 module_init 宏定义在 include/linux/init.h 中。

```
63  typedef int (*initcall_t)(void);
... ...
92  #define __define_initcall(level,fn,id) \
93      static initcall_t __initcall_##fn##id __attribute_used__ \
94      __attribute__((__section__(".initcall" level ".init"))) = fn
... ...
115 #define device_initcall(fn)    __define_initcall("6",fn,6)
... ...
120 #define __initcall(fn) device_initcall(fn)
... ...
179 #define module_init(x)    __initcall(x);
```

上述代码中__define_initcall 宏利用##定义一个新的已初始化的函数指针。例如：

module_init(e100_init_module);会在预处理时被替换为：

```
static  initcall_t  __initcall_e100_init_moduleid  __attribute_used__
__attribute__((__section__(".initcall6.init"))) = e100_init_module
```

__attribute__是 gcc 的关键字，用来指示编译器给符号设置指定的属性。module_init(x)宏的意义是在.initcall6.init 段中定义一个指向 x 的函数指针。文件 arch/i386/vmlinux.lds.S 中定义了执行文件的信息及各段内核代码的空间，如图 4-2 所示。

其中 INITCALLS 定义在 include/asm-generic/vmlinux.lds.h 中，代码如下所示：

```
213 #define INITCALLS                                       \
214     *(.initcall0.init)                                  \
215     *(.initcall0s.init)                                 \
... ...
229     *(.initcall7.init)                                  \
230     *(.initcall7s.init)
```

.initcall.init 定义在 arch/i386/vmlinux.lds.S 中，代码如下所示：

```
154   .initcall.init : AT(ADDR(.initcall.init) - LOAD_OFFSET) {
155       __initcall_start = .;
156     INITCALLS
157       __initcall_end = .;
158   }
```

也就是说，系统依靠全局变量__initcall_start 和__initcall_end 来记录.initcall.init 段的起始/结束地址。

从 do_initcalls()中可以看出，所有的初始化函数指针都存放在__initcall_start 与__initcall_ end

之间，初始化过程就是逐个调用这些函数。真正的初始化过程就变得比较简单，因为所有复杂工作都放在编译期，由编译器去处理了。从此侧面也可看出 gcc 功能的强大，它是随着 Linux 内核的发展而发展的，就像 C 语言随着 UNIX 发展一样，相辅相成，共同发展。

```
634 static void __init do_initcalls(void)
635 {
636     initcall_t *call;
637     int count = preempt_count();
638
639     for(call = __initcall_start; call < __initcall_end; call++){
... ...
651         result = (*call)();
... ...
671     }
672
673     /* Make sure there is no pending stuff from the initcall sequence */
674     flush_scheduled_work();
675 }
```

定义模块加载函数的宏 module_init 被定义在 include/linux/init.h 中。

```
63  typedef int (*initcall_t)(void);
... ...
213 #define module_init(initfn)                    \
214     static inline initcall_t __inittest(void)      \
215     { return initfn; }                  \
216     int init_module(void) __attribute__((alias(#initfn)));
```

在这种情况下，module_init(x)的实现在模块中定义一个别名为 init_module 的 x 函数。作为能动态加载的模块，如需初始化，Linux 规定初始化接口必须为 init_module，模块被加载时，init_module()系统调用会根据 x 得到初始化函数的地址，并调用之。通过__attribute__((alias(#initfn)))为函数定义一个别名。

Linux 内核使用各种各样的宏来标识函数和数据结构的特殊属性，例如标识初始化函数。绝大多数的宏都定义在 include/linux/init.h 文件中，这些宏很多是用来告诉链接器把这些具有特殊属性的代码或数据结构放到特殊的、专用的内存区(section)内。这样做，内核能以一种简单的方式很容易地访问一类具有特殊属性的代码或数据。

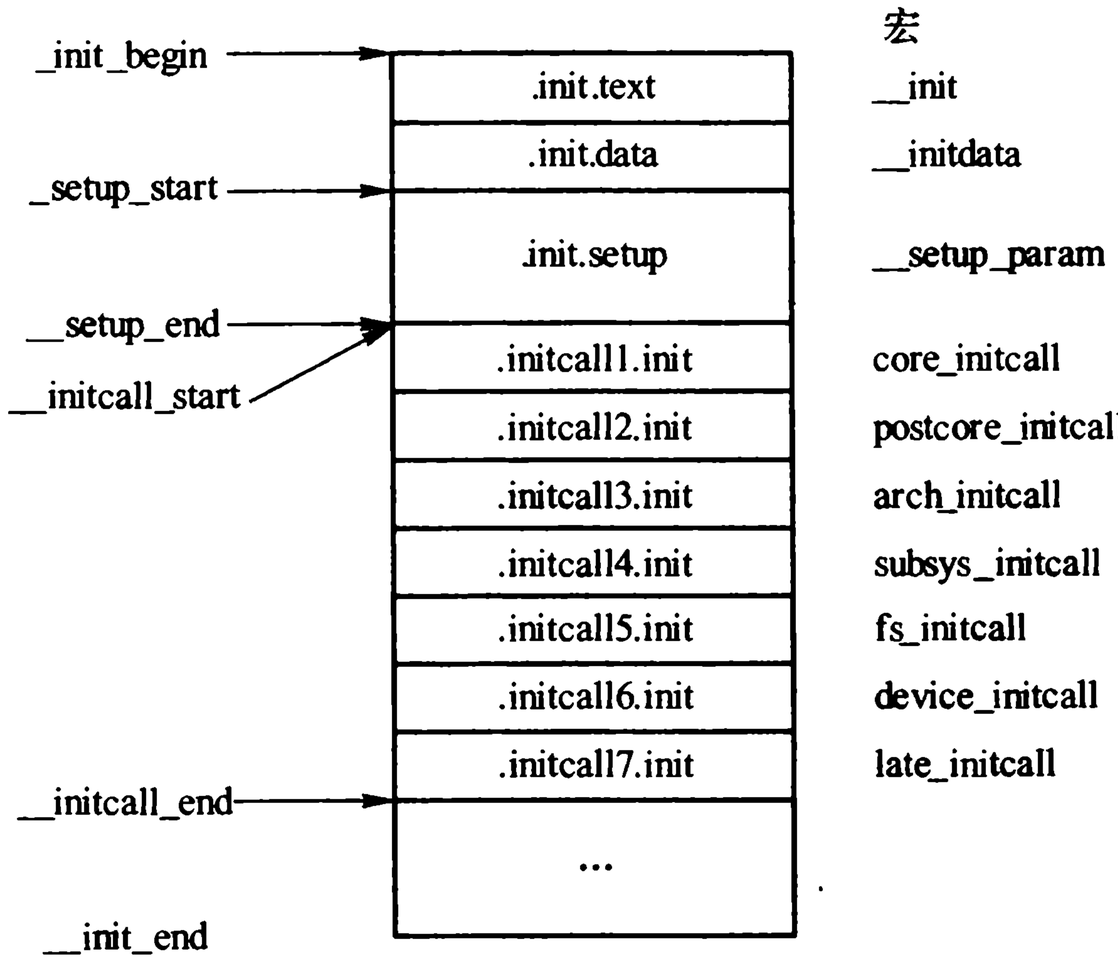

图 4-2　各段的空间分布以及对应的宏

图 4-2 为初始化代码所使用的部分内存区(section)示意图。左边是分隔每个区或节的开始与结束部分的指针名，右边是用来将数据和代码放到相应内存区的宏名。

表 4-1 和表 4-2 列出了部分用于标记程序或数据的宏并给出了简单的描述。

表 4-1 修饰函数的宏

宏	使用宏的函数说明
__init	启动时初始化函数，在启动阶段执行，通常只执行一次。后期不再需要，这种函数在初始化完成后被从内存中清除
__exit	和__init 匹配，相关内核组件卸载时调用，常用于 module_exit 所修饰的函数
core_initcall postcore_initcall arch_initcall subsys_initcall fs_initcall device_initcall late_initcall	用于标记启动时需要执行的初始化函数
__initcall	device_initcall 的别名
__exitcall	标识退出函数，相关内核组件卸载时调用。通常仅用于标记 module_exit 函数

表 4-2 初始化数据结构的宏

宏	使用宏的数据(结构)说明
__initdata	仅在启动时用于已初始化的数据结构
__exitdata	仅被由__exitcall 修饰的函数使用的数据结构，也有一层意思是：如果被__exitcall 修饰的函数即使不被使用，由__exitdata 修饰的数据也是正确的。因此，各种优化也可以被用于__exitdata 和__exitcall

以上两张表中一些宏都有以下几个特点：

- 绝大多数宏都是配对出现的，即一个修饰初始化加载，与其配对的一个修饰卸载过程，例如，__exit 和__init 匹配，__exitcalls 和__initcall 匹配。
- 需兼顾两方面，一方面是函数的执行时间，如__initcall、__exitcall，另一方面就是函数/数据放置的内存区。
- 同一个函数可以被多个宏标记，例如：e100_init_module 为宏__initcall 和__init 所标记，前者说明可以在启动时运行，后者说明一旦运行即可被释放。

4.4 网络设备处理层初始化

通过以上对初始化的分析，相信读者应该已经明白了其实现机制。现在来看一下网络模块中哪些地方进行了初始化，见表 4-3。

表 4-3 网络模块中初始化的函数（部分）

文件	初始化函数及宏	说明
net/socket.c	core_initcall(sock_init)	套接口层的初始化函数
net/core/sock.c	subsys_initcall(proto_init)	传输层的初始化函数
net/ipv4/af_inet.c	fs_initcall(inet_init)	Internet 协议族的初始化函数
net/core/dev.c	subsys_initcall(net_dev_init)	设备处理层的初始化函数
drivers/net/e100.c	module_init(e100_init_module)	e100 型号的网络设备驱动的初始化函数

其中 sock_init()、proto_init()、inet_init()初始化函数涉及网络的三层、四层协议，会在后续章节中介绍。本章感兴趣的是 net_dev_init()和 e100_init_module()这两个初始化函数。

```
3476 static int __init net_dev_init(void)
3477 {
3478     int i, rc = -ENOMEM;
```

```
3479
3480    BUG_ON(!dev_boot_phase);
3481
3482    if (dev_proc_init())
3483        goto out;
3484
3485    if (netdev_sysfs_init())
3486        goto out;
```

3482-3483 在 proc 文件系统中注册/proc/net/dev 和/proc/net/softnet_stat 文件，这两个文件是只读的，主要是对网络设备的状态和一些数据的统计。

3485-3486 在新型的 sysfs 设备文件系统的 class 中注册 net 结点。

```
3488    INIT_LIST_HEAD(&ptype_all);
3489    for (i = 0; i < 16; i++)
3490        INIT_LIST_HEAD(&ptype_base[i]);
3491
3492    for (i = 0; i < ARRAY_SIZE(dev_name_head); i++)
3493        INIT_HLIST_HEAD(&dev_name_head[i]);
3494
3495    for (i = 0; i < ARRAY_SIZE(dev_index_head); i++)
3496        INIT_HLIST_HEAD(&dev_index_head[i]);
```

3488-3490 初始化网络处理函数散列表 ptype_base。这些处理函数用来处理接收到的不同协议族报文。

3492-3496 初始化存放网络设备的散列表 dev name head 和 dev index head，前一个散列表关键字由设备名称计算获得，而后一个散列表关键字由设备接口索引计算获得。

```
3502    for_each_possible_cpu(i) {
3503        struct softnet_data *queue;
3504
3505        queue = &per_cpu(softnet_data, i);
3506        skb_queue_head_init(&queue->input_pkt_queue);
3507        queue->completion_queue = NULL;
3508        INIT_LIST_HEAD(&queue->poll_list);
3509        set_bit(__LINK_STATE_START, &queue->backlog_dev.state);
3510        queue->backlog_dev.weight = weight_p;
3511        queue->backlog_dev.poll = process_backlog;
3512        atomic_set(&queue->backlog_dev.refcnt, 1);
3513    }
```

3502-3513 初始化与 CPU 相关的接收队列。

```
3515    netdev_dma_register();
3516
3517    dev_boot_phase = 0;
3518
3519    open_softirq(NET_TX_SOFTIRQ, net_tx_action, NULL);
3520    open_softirq(NET_RX_SOFTIRQ, net_rx_action, NULL);
```

3515 与网络设备的 DMA 相关，本书不作论述。

3517 标识设备层初始化已完成。

3519-3520 在软中断系统中注册两个软中断 NET_TX_SOFTIRQ 和 NET_RX_SOFTIRQ，

用于网络数据的发送和接收。因为软中断的性能比较好，而网络数据的接收和发送对性能要求比较高，因此将软中断作为下半部来使用。

```
3522     hotcpu_notifier(dev_cpu_callback, 0);
3523     dst_init();
3524     dev_mcast_init();
3525     rc = 0;
3526 out:
3527     return rc;
3528 }
```

3522 在通知链表上注册一个回调函数，用来响应 CPU 热插拔事件。一旦接到通知，CPU 输入队列中的包逐一交由 netif_rx()处理。

3523 初始化目的路由缓存。

3524 初始化网络设备层的组播模块，并在 proc 文件系统中增加文件/proc/net/dev_mcast，用来存放内核中网络设备与 IP 组播相关的参数。

第5章 网络设备

5.1 PCI 设备

当前，PCI 接口设备已在 PC 上广泛使用，老式的 ISA 接口基本被淘汰。很多网络设备都是基于 PCI 接口的，因此尽管网络设备驱动比较特殊，但也要作为 PCI 驱动注册到内核中。

PCI 接口等定义、网络设备驱动相关定义涉及以下文件：

- include/linux/mod_devicetable.h，定义导出到用户空间的 PCI 设备信息。
- include/linux/pci.h，定义 PCI 接口驱动相关的结构、宏等。
- include/linux/netdevice.h，定义网络设备结构、宏等。
- include/linux/inetdevice.h，定义 IPv4 专用的网络设备相关的结构、宏等。
- net/core/dev.c，网络设备注册、输入和输出等接口。
- net/ethernet/eth.c，以太网网络设备驱动专用接口。
- net/core/link_watch.c，网络设备连接状态通知。
- drivers/net/e100.c，e100 驱动程序。

5.1.1 PCI 驱动程序相关结构

1. pci_device_id 结构

标准的 PCI 设备中都有一个配置寄存器，用来存放各种参数，其中的 vendorID（厂商 ID）、deviceID(设备 ID)、class(类代号)、subsystem vendorID（子系统厂商 ID）和 subsystem deviceID（子系统设备 ID）是我们所关注的。vendorID、deviceID 和 class 三个字段可以标识一个设备，而 subsystem vendorID 和 subsystem deviceID 则可以进一步区分相似的设备，但不是必须的。在系统中这些信息用 pci_device_id 结构来保存。

```
17 struct pci_device_id {
18     __u32 vendor, device;
19     __u32 subvendor, subdevice;
20     __u32 class, class_mask;
21     kernel_ulong_t driver_data;
22 };
```

18 __u32 vendor, device

vendor 为厂商 ID，用于标识硬件制造商。PCI Special Interest Group 维护一个全球厂商的编号注册表，制造商必须申请一个唯一编号并写入到设备的 vendorID 寄存器中。例如，每个 Intel 设备都会被标识为相同的厂商编号 0x8086。

device 为设备 ID，由制造商自己设置。

设备 ID 与厂商 ID 配对生成一个唯一的 32 位硬件设备标识符，驱动程序通常依靠此标识符来识别设备。

```
19 __u32 subvendor, subdevice
```

subvendor 和 subdevice 指定设备的 PCI 系统厂商和子系统设备 ID。如果驱动程序可以处理任何类型的子系统，则这两个字段应设置为 PCI_ANY_ID。

```
20 __u32 class, class_mask
```

class 和 class_mask 使驱动程序可以指定支持某一种 PCI 类设备。在 PCI 规范中描述了不同类的 PCI 设备（参见 PCI 规范文档）。如果驱动程序可以处理任何类型的子系统，则这两个字段也应设置为 PCI_ANY_ID。

```
21 kernel_ulong_t driver_data
```

保存与具体设备相关的私有信息。如 e100 网络设备驱动程序，同时支持 e100 多个不同设备 ID 的设备，可以说是一款通用的 e100 驱动程序。

```
186 #define INTEL_8255X_ETHERNET_DEVICE(device_id, ich) {\
187     PCI_VENDOR_ID_INTEL, device_id, PCI_ANY_ID, PCI_ANY_ID, \
188     PCI_CLASS_NETWORK_ETHERNET << 8, 0xFFFF00, ich }
189 static struct pci_device_id e100_id_table[] = {
190     INTEL_8255X_ETHERNET_DEVICE(0x1029, 0),
... ...
230     INTEL_8255X_ETHERNET_DEVICE(0x27DC, 7),
231     { 0, }
232 };
233 MODULE_DEVICE_TABLE(pci, e100_id_table);
```

上述代码中定义了一个名为 e100_id_table 的 pci_device_id 结构类型数组，可看出该驱动支持 40 多款 e100 型设备，只是功能有些差别，根据私有信息 driver_data 来辨别。

Linux 系统支持设备的热插拔功能，因此这些信息需要被导出到用户空间，这样当系统在热插拔时，内核热插拔模块会检测到相应的事件，然后通知用户程序 hotplug 根据厂商 ID 和设备 ID 等装载相应的驱动模块。导出的功能由 MODULE_DEVICE_TABLE 实现。

```
80  #ifdef MODULE
81  #define MODULE_GENERIC_TABLE(gtype,name)                \
82  extern const struct gtype##_id __mod_##gtype##_table    \
83    __attribute__ ((unused, alias(__stringify(name))))
... ...
87  #else  /* !MODULE */
... ...
90  #endif
... ...
139 #define MODULE_DEVICE_TABLE(type,name)          \
140   MODULE_GENERIC_TABLE(type##_device,name)
```

以上代码的233行，为e100_id_table创建了一个别名为__mod_pci_table的局部变量。depmod进程在所有模块中搜索符号__mod_pci_table，如果搜索命中，则将其从该模块中导出，添加到文件/lib/modules/{KERNEL_VERSION}/modules.pcimap 中。depmod 完成操作后，内核模块支持的所有 PCI 设备连同它们的模块名都在该文件中列出。当热插拔系统发现新的 PCI 设备时，使用 modules.pcimap 文件来寻找要装载的相应驱动程序。

2. pci_driver 结构

pci_driver 结构用来描述一个 PCI 设备，因此所有的 PCI 驱动都必须创建一个 pci_driver 结

构的实例，用来向 PCI 设备管理模块描述 PCI 驱动程序。

```
344 struct pci_driver {
345     struct list_head node;
346     char *name;
347     const struct pci_device_id *id_table;
348     int (*probe) (struct pci_dev *dev, const struct pci_device_id *id);
349     void (*remove) (struct pci_dev *dev);
350     int  (*suspend) (struct pci_dev *dev, pm_message_t state);
351     int (*suspend_late) (struct pci_dev *dev,pm_message_t state);
352     int  (*resume_early) (struct pci_dev *dev);
353     int  (*resume) (struct pci_dev *dev);    /* Device woken up */
354     int  (*enable_wake) (struct pci_dev *dev, pci_power_t state, int enable);  /*
    Enable wake event */
355     void (*shutdown) (struct pci_dev *dev);
356
357     struct pci_error_handlers *err_handler;
358     struct device_driver    driver;
359     struct pci_dynids dynids;
360
361     int multithread_probe;
362 };
```

346 char *name

驱动程序名，在内核中所有 PCI 驱动程序名都是唯一的，通常被设置为和驱动程序模块相同的名字。装载驱动程序后，系统会在/sys/bus/pci/drivers/下面建立相应的目录。

347 const struct pci_device_id *id_table

指向 pci_device_id 结构数组的指针。

348 int(*probe)(struct pci_dev *dev, const struct pci_device_id *id)

指向 PC 驱动中 probe 函数的指针。当有驱动被添加到内核时，会调用此接口进行设备的初始化。

349 void (*remove) (struct pci_dev *dev)

从系统中移除设备的接口，当设备从系统中被移除时被调用。

350 int (*suspend) (struct pci_dev *dev, pm_message_t state)

设备被挂起时被调用的接口，此接口可以不支持。

```
351 int (*suspend_late) (struct pci_dev *dev,pm_message_t state)
352 int (*resume_early) (struct pci_dev *dev)
353 int (*resume) (struct pci_dev *dev)
354 int (*enable_wake) (struct pci_dev *dev, pci_power_t state, int enable)
355 void (*shutdown) (struct pci_dev *dev)
357 struct pci_error_handlers *err_handler
358 struct device_driver driver
359 struct pci_dynids dynids
361 int multithread_probe
```

与 PCI 驱动程序相关，本书不作论述，可参见相关资料。

5.1.2 注册 PCI 驱动程序

这里以 e100 为例来说明驱动程序的注册过程。以下代码取自 e100 驱动程序，其中只留下

了与 PCI 驱动注册相关的函数框架，但并不妨碍对注册 PCI 驱动程序过程的理解。

```
2545 static int __devinit e100_probe(struct pci_dev *pdev,
2546    const struct pci_device_id *ent)
2547 {
.... ...
2697 }
2698
2699 static void __devexit e100_remove(struct pci_dev *pdev)
2700 {
.... ...
2713 }
2714
2715 #ifdef CONFIG_PM
2716 static int e100_suspend(struct pci_dev *pdev, pm_message_t state)
2717 {
.... ...
2742 }
2743
2744 static int e100_resume(struct pci_dev *pdev)
2745 {
.... ..
2759 }
2760 #endif /* CONFIG_PM */
2761
2762 static void e100_shutdown(struct pci_dev *pdev)
2763 {
.... ...
2782 }
.... ...
2854 static struct pci_error_handlers e100_err_handler = {
2855    .error_detected = e100_io_error_detected,
2856    .slot_reset = e100_io_slot_reset,
2857    .resume = e100_io_resume,
2858 };
2859
2860 static struct pci_driver e100_driver = {
2861    .name =        DRV_NAME,
2862    .id_table =    e100_id_table,
2863    .probe =       e100_probe,
2864    .remove =      __devexit_p(e100_remove),
2865 #ifdef CONFIG_PM
2866    /* Power Management hooks */
2867    .suspend =     e100_suspend,
2868    .resume =      e100_resume,
2869 #endif
2870    .shutdown =    e100_shutdown,
2871    .err_handler = &e100_err_handler,
2872 };
2873
2874 static int __init e100_init_module(void)
2875 {
.... ...
2880    return pci_register_driver(&e100_driver);
2881 }
2882
```

```
2883 static void __exit e100_cleanup_module(void)
2884 {
2885     pci_unregister_driver(&e100_driver);
2886 }
2887
2888 module_init(e100_init_module);
2889 module_exit(e100_cleanup_module);
```

e100_id_table 是 e100 驱动程序的 pci_device_id 结构类型列表。由 module_init 可知，e100_init_module()为 e100 驱动的初始化接口，在模块装载到内核中时被调用。函数中调用 pci_register_driver()为 e100 网络设备进行 PCI 驱动的注册。需要注意的是，只有当内核支持电源管理时才会把 e100_suspend()和 e100_resume()编译到模块中。

5.2 与网络设备有关的数据结构

5.2.1 net_device 结构

net_device 结构是网络驱动及接口层中最重要的结构，其中不但描述了接口方面的信息，还包括硬件信息，致使该结构很大很复杂。将接口和驱动完全整合在一起也许是设计上的一个失误，因此可以看到这样的一行注释“Actually, this whole structure is a big mistake”。

net_device 结构的成员大致可以分为以下几类：

- 硬件信息成员变量：与网络设备相关的底层硬件信息，如果是虚拟网络设备驱动，则这部分信息无效。
- 接口信息成员变量：本节介绍有关接口方面的信息，这些信息主要是为其他硬件类型的 setup()而设置的，对以太网来说是 ether_setup()，以太网络设备利用该函数设置大部分成员。
- 设备操作接口变量：设备的接口主要提供操作数据或控制设备的一些功能，如发送数据包的接口、激活和关闭设备的接口等。在这些接口中，有些是必须的，而有些是可选的，这与设备提供的特性有关。
- 辅助成员变量。

以下为 net_device 结构的定义，由于篇幅比较长，因此删去了一些注释。

```
274 struct net_device
275 {
... ...
282     char            name[IFNAMSIZ];
283     /* device name hash chain */
284     struct hlist_node   name_hlist;
285
... ...
290     unsigned long       mem_end;    /* shared mem end    */
291     unsigned long       mem_start;    /* shared mem start    */
292     unsigned long       base_addr;    /* device I/O address    */
293     unsigned int        irq;        /* device IRQ number    */
... ...
300     unsigned char       if_port;    /* Selectable AUI, TP,..*/
301     unsigned char       dma;        /* DMA channel        */
302
```

```
303     unsigned long       state;
304
305     struct net_device   *next;
306
307     /* The device initialization function. Called only once. */
308     int             (*init)(struct net_device *dev);
... ...
313     unsigned long       features;
... ...
342     struct net_device   *next_sched;
343
344     /* Interface index. Unique device identifier    */
345     int             ifindex;
346     int             iflink;
347
348
349     struct net_device_stats* (*get_stats)(struct net_device *dev);
... ...
353     const struct iw_handler_def *   wireless_handlers;
354     /*Instance data managed by the core of Wireless Extensions.*/
355     struct iw_public_data *    wireless_data;
356
357     const struct ethtool_ops *ethtool_ops;
... ...
366     unsigned int       flags;    /*interface flags (a la BSD)    */
367     unsigned short       gflags;
368        unsigned short         priv_flags; /* Like 'flags' but invisible to userspace.
*/
369     unsigned short       padded;/* How much padding added by alloc_netdev()*/
370
371     unsigned char       operstate; /* RFC2863 operstate */
372     unsigned char       link_mode; /*mapping policy to operstate*/
373
374     unsigned       mtu;    /* interface MTU value       */
375     unsigned short       type;    /* interface hardware type    */
376     unsigned short       hard_header_len;/*hardware hdr length */
377
378     struct net_device    *master; /* Pointer to master device of a group,
379                   * which this device is member of.
380                   */
381
382     /* Interface address info. */
383     unsigned char       perm_addr[MAX_ADDR_LEN]; /* permanent hw address */
384     unsigned char       addr_len;    /*hardware address length */
385     unsigned short         dev_id;    /* for shared network cards*/
386
387     struct dev_mc_list    *mc_list;   /*Multicast mac addresses*/
388     int             mc_count;    /* Number of installed mcasts    */
389     int             promiscuity;
390     int             allmulti;
391
392
393     /* Protocol specific pointers */
394
395     void             *atalk_ptr;    /* AppleTalk link     */
396     void            *ip_ptr;    /* IPv4 specific data    */
397     void                 *dn_ptr;       /*DECnet specific data */
398     void                 *ip6_ptr;      /* IPv6 specific data */
```

```
399     void            *ec_ptr;    /* Econet specific data    */
400     void            *ax25_ptr;    /* AX.25 specific data */
... ...
405     struct list_head    poll_list ___cacheline_aligned_in_smp;
406                     /* Link to poll list    */
407
408     int             (*poll) (struct net_device *dev, int *quota);
409     int             quota;
410     int             weight;
411     unsigned long       last_rx;    /* Time of last Rx    */
412     /* Interface address info used in eth_type_trans() */
413     unsigned char       dev_addr[MAX_ADDR_LEN];    /* hw address, (before bcast
414                         because most packets are unicast) */
415
416     unsigned char       broadcast[MAX_ADDR_LEN]; /*hw bcast add*/
... ...
422     spinlock_t      queue_lock ___cacheline_aligned_in_smp;
423     struct Qdisc        *qdisc;
424     struct Qdisc        *qdisc_sleeping;
425     struct list_head    qdisc_list;
426     unsigned long       tx_queue_len;    /* Max frames per queue allowed */
427
428     /* Partially transmitted GSO packet. */
429     struct sk_buff      *gso_skb;
430
431     /* ingress path synchronizer */
432     spinlock_t      ingress_lock;
433     struct Qdisc        *qdisc_ingress;
... ...
438     /* hard_start_xmit synchronizer */
439     spinlock_t      _xmit_lock ___cacheline_aligned_in_smp;
... ...
443     int             xmit_lock_owner;
444     void            *priv;    /* pointer to private data    */
445     int             (*hard_start_xmit) (struct sk_buff *skb,
446                                 struct net_device *dev);
447     /* These may be needed for future network-power-down code. */
448     unsigned long       trans_start;/*Time (in jiffies) of last Tx*/
449
450     int             watchdog_timeo; /* used by dev_watchdog() */
451     struct timer_list   watchdog_timer;
... ...
457     atomic_t        refcnt ___cacheline_aligned_in_smp;
458
459     /* delayed register/unregister */
460     struct list_head    todo_list;
461     /* device index hash chain */
462     struct hlist_node   index_hlist;
463
464     /* register/unregister state machine */
465     enum { NETREG_UNINITIALIZED=0,
466            NETREG_REGISTERED,   /* completed register_netdevice */
467            NETREG_UNREGISTERING, /* called unregister_netdevice */
468            NETREG_UNREGISTERED,    /* completed unregister todo */
469            NETREG_RELEASED,        /* called free_netdev */
470     } reg_state;
471
472     /* Called after device is detached from network. */
```

```
473     void            (*uninit)(struct net_device *dev);
474     /* Called after last user reference disappears. */
475     void            (*destructor)(struct net_device *dev);
476
477     /* Pointers to interface service routines.    */
478     int             (*open)(struct net_device *dev);
479     int             (*stop)(struct net_device *dev);
480 #define HAVE_NETDEV_POLL
481     int             (*hard_header) (struct sk_buff *skb,
482                         struct net_device *dev,
483                         unsigned short type,
484                         void *daddr,
485                         void *saddr,
486                         unsigned len);
487     int             (*rebuild_header)(struct sk_buff *skb);
488 #define HAVE_MULTICAST
489     void            (*set_multicast_list)(struct net_device *dev);
490 #define HAVE_SET_MAC_ADDR
491     int             (*set_mac_address)(struct net_device *dev,
492                         void *addr);
493 #define HAVE_PRIVATE_IOCTL
494     int             (*do_ioctl)(struct net_device *dev,
495                         struct ifreq *ifr, int cmd);
496 #define HAVE_SET_CONFIG
497     int             (*set_config)(struct net_device *dev,
498                         struct ifmap *map);
499 #define HAVE_HEADER_CACHE
500     int             (*hard_header_cache)(struct neighbour *neigh,
501                         struct hh_cache *hh);
502     void            (*header_cache_update)(struct hh_cache *hh,
503                         struct net_device *dev,
504                         unsigned char *  haddr);
505 #define HAVE_CHANGE_MTU
506     int         (*change_mtu)(struct net_device *dev, int new_mtu);
507
508 #define HAVE_TX_TIMEOUT
509     void            (*tx_timeout) (struct net_device *dev);
510
511     void            (*vlan_rx_register)(struct net_device *dev,
512                         struct vlan_group *grp);
513     void            (*vlan_rx_add_vid)(struct net_device *dev,
514                         unsigned short vid);
515     void            (*vlan_rx_kill_vid)(struct net_device *dev,
516                         unsigned short vid);
517
518     int             (*hard_header_parse)(struct sk_buff *skb,
519                         unsigned char *haddr);
520     int         (*neigh_setup)(struct net_device *dev, struct neigh_parms *);
521 #ifdef CONFIG_NETPOLL
522     struct netpoll_info   *npinfo;
523 #endif
524 #ifdef CONFIG_NET_POLL_CONTROLLER
525     void            (*poll_controller)(struct net_device *dev);
526 #endif
527
528     /* bridge stuff */
529     struct net_bridge_port   *br_port;
530
```

```
531     /* class/net/name entry */
532     struct class_device    class_dev;
533     /* space for optional statistics and wireless sysfs groups */
534     struct attribute_group  *sysfs_groups[3];
535 };
```

282 char name[IFNAMSIZ]

网络设备名。如果在注册设备时，驱动程序名中含有%d 格式化字符串，则使用一个从 0 开始分配的编号替换它，使之成为系统中唯一的名称。

284 struct hlist_node name_hlist

根据网络设备名以散列表的形式组织到 dev name head 散列表中，这样就可以通过网络设备名快速地定位到对应的网络设备。

290 unsigned long mem_end

291 unsigned long mem_start

mem_start 和 mem_end 是网络设备共享内存的起始和终止地址。根据约定，mem_end 的设置要确保(mem_end-mem_start)等于设备的可用内存容量。

292 unsigned long base_addr

网络接口 I/O 基地址，在探测设备时被初始化。ifconfig 命令可显示和修改当前值。

293 unsigned int irq

分配给设备的中断号，一般在初始化设备时被初始化。

300 unsigned char if_port

指定在多端口设备上使用哪个端口。例如，设备同时支持同轴电缆和双绞线以太网连接，即可使用该成员来指定使用的端口。但目前已很少使用该字段了。

301 unsigned char dma

为设备分配的 DMA 通道。该成员只对某些外设总线有用，如 ISA。

303 unsigned long state

设备状态，其中包含若干状态标志及 QoS 排队规则状态，见表 5-1。

表 5-1 state 的取值

state	描述
__LINK_STATE_XOFF	由于热插拔网络设备、缓存不够、网络设备硬件错误或者关闭或禁止硬件，从而关闭排队功能
__LINK_STATE_START	网络设备处于激活状态
__LINK_STATE_PRESENT	在电源管理中，当系统处于待机时，需要挂起各个设备，同时要记录各设备的待机前的状态，标识网络设备对系统是可用的。使用此标志用来记录待机前的设备状态，以使在系统恢复时来判断是否需要启动设备
__LINK_STATE_SCHED	标识网络驱动的数据发送是否在流量控制的调度中
__LINK_STATE_NOCARRIER	标识网络设备处理是否可传递状态，当网络设备不能传递数据时被设置。例如，网线是否被拔出等。此标志可由 netif_carrier_ok 检测，参见 5.8 节
__LINK_STATE_RX_SCHED	标识正在轮询接收数据包。对于通过中断结合轮询方式接收数据包网络设备，当数据包到达而触发中断时，在中断中便会设置该标志，表示进入接收数据包状态，然后激活接收数据包软中断，并结合轮询进行接收数据包，直至此次接收数据包完成。在此状态，即使有新的中断产生，也不会调度软中断
__LINK_STATE_LINKWATCH_PENDING	网络设备的连接状态发生改变，正在处理改变事件过程中，参见 5.8.1 节
__LINK_STATE_DORMANT	与 The Interfaces Group MIB（RFC 2863）中的 IfOperStatus 的 IF_OPER_DORMANT 相关，本书不作论述
__LINK_STATE_QDISC_RUNNING	进行流量控制，正在调度队列过程中

305 struct net_device *next

指向 net_device 结构链表下一个设备的指针，系统中所有网络设备是由它组织起来的。

308 int (*init)(struct net_device *dev)

驱动程序的初始化函数。如果该指针已设置，则 register_netdev()将调用该函数来完成对 net_device 结构的初始化。由于网络设备大多数是 PCI 接口，因此目前的做法大多是在注册 PCI 驱动后就进行相应的初始化工作，很少使用此接口来完成初始化工作了。

net_device 结构中的一些字段不能简单地初始化，因为可能对该设备类型无意义，不需要初始化这些函数指针，让其保持 NULL 值。为避免使用 NULL 指针，内核确保可选函数指针在调用前被初始化，例如 register_netdevice()中调用 init()。

```
2901     if (dev->init) {
2902        ret = dev->init(dev);
2903        if (ret) {
2904           if (ret > 0)
2905              ret = -EIO;
2906           goto out;
2907        }
2908     }
```

313 unsigned long features

接口支持的特性，见表 5-2。这些标志由内核管理，其中有些由接口在初始化期间设置，用来声明接口的各种能力及特性。

表 5-2 features 的取值

features	描述
NETIF_F_SG	SG 类型的聚合分散 I/O 标志。当一个数据包被分成了多个独立的内存段，并且接口又能传输这样的数据包，则需要设置 NETIF_F_SG
NETIF_F_FRAGLIST	FRAGLIST 类型的聚合分散 I/O 标志。表明接口能处理那些被分成块的数据包，在内核 2.6 中只有回环设备有此功能。通常通过 skb_shinfo(skb)->frag_list 链接起来的数据包
NETIF_F_IP_CSUM	接口仅能够校验 IP 数据报
NETIF_F_NO_CSUM	不需要执行校验和。如回环设备，由于数据包是通过系统内存传输，一般不会失败，也就不需要检查它们了
NETIF_F_HW_CSUM	由硬件进行校验
NETIF_F_HIGHDMA	如果设备可以在高端内存使用 DMA，设置该标志。如果不设置该标志，所有为驱动程序提供的数据包缓冲区将在低端内存中分配
NETIF_F_HW_VLAN_TX NETIF_F_HW_VLAN_RX NETIF_F_HW_VLAN_FILTER	支持收发 802.1q VLAN 数据包的硬件加速等处理
NETIF_F_VLAN_CHALLENGED	标识设备不支持硬件，支持 802.1q VLAN 数据包
NETIF_F_TSO	标识设备支持 TCP 段卸载
NETIF_F_UFO	标识设备支持 UDP 分片卸载
NETIF_F_GSO_ROBUST	标识设备支持对从一个不可信赖的来源发出的数据报进行段卸载
NETIF_F_TSO_ECN	IPv4 的 TCP 段卸载，当设置 TCP 首部的 CWR 时，使用此 gos_type。CWR 参见 29.4 节
NETIF_F_TSO6	标识设备支持 IPv6 的 TCP 段卸载
NETIF_F_GSO	标识设备支持某种 GSO。在输出数据报时会得到输出网络设备的特性，这样传输层可以根据输出网络设备的 GSO 特性处理输出的数据
NETIF_F_LLTX	标识通过网络设备输出数据包时是否需要上锁。设置时，输出数据包时不需要上锁

342 struct net_device *next_sched

用于链接那些已调度有数据包输出的网络设备的指针，参见8.2节。

```
345 int ifindex
```

网络设备的索引号。

```
346 int iflink
```

网络设备的唯一标识，主要用于虚拟隧道设备。

```
349 struct net_device_stats* (*get_stats)(struct net_device *dev)
```

提供给应用程序获得接口统计信息的接口。例如，在ifconfig命令获取网络设备统计信息时会调用此方法。

```
353 const struct iw_handler_def *wireless_handlers
355 struct iw_public_data *wireless_data
```

与无线通信相关的成员，本书不作论述。

```
357 const struct ethtool_ops *ethtool_ops
```

ethtool的操作接口。

```
366 unsigned int flags
```

标识接口特性，见表5-3。这些标志由内核管理，其中一些由接口在初始化期间设置，用来声明接口的各种能力及特性。

表5-3 flags的取值

flags	描述
IFF_UP	标志接口已激活并可以开始传输数据包
IFF_BROADCAST	标志该接口允许广播
IFF_DEBUG	标识调试模式。该标志可用来控制用于调试目的的大量printk调用。用户程序可通过ioctl设置或清除该标志
IFF_LOOPBACK	该标志只能对回环设备进行设置。内核检查IFF_LOOPBACK标志以判断接口是否为回环设备，而不是将lo作为特殊的接口名称进行判断
IFF_POINTOPOINT	该标志表明接口连接到点对点链路。这个标志由驱动程序设置，有时也由ifconfig设置。例如PPP驱动程序将设置该标志
IFF_NOARP	该标志标明接口不支持ARP。例如，点对点接口不需要运行ARP，因为如果运行了ARP，不但不能获得有用的信息，而且增加了网络传输量
IFF_PROMISC	设置该标志将激活混杂模式。默认情况下，以太网接口使用一个硬件过滤器来确保它只接收广播数据包，以及直接发送到接口硬件地址的数据包。像tcpdump这样的数据包侦听器会在接口上设置混杂模式，以使检索到通过传输介质的所有数据包
IFF_MULTICAST	该标志由驱动程序设置，标识该接口能够进行组播发送。ether_setup默认设置IFF_MULTICAST，因此如果驱动程序不支持组播，就必须在初始化时清除该标志
IFF_ALLMULTI	该标志告诉接口接收所有的组播数据包。仅仅在IFF_MULTICAST被设置的情况下，内核在主机执行组播路由时设置该标志。IFF_ALLMULTI对接口来讲是只读的
IFF_MASTER	主负载平衡群(bundle)
IFF_SLAVE	该标志由负载均衡代码使用。接口驱动程序无需了解该标志。从负载平衡群
IFF_PORTSEL	可以通过ifmap选择介质(media)类型
IFF_AUTOMEDIA	该标志表明设备能够在多种介质类型之间切换。例如，在非屏蔽双绞线和同轴电缆以太网之间。如果IFF_AUTOMEDIA被设置，设备会自动选择正确的介质类型。在实际情况中，该标志没有被使用
IFF_DYNAMIC	该标志由驱动程序设置，表示接口地址可改变。这标志没有被使用。接口关闭时丢弃地址
IFF_RUNNING	该标志表示接口已经启动并且正在运行，该标志主要用于兼容BSD，内核很少使用该标志
IFF_NOTRAILERS	Linux不使用该标志，只是为了与BSD兼容

```
367 unsigned short gflags
```

记录当前网络设备IFF_PROMISC和IFF_ALLMULTI的状态，用来配合flags的设置。

368 unsigned short priv_flags

与 flags 相似，见表 5-4。这些值对用户不可见，即不能由应用程序配置，其中标识了一些有关 bonding 驱动的内容，本书不作论述，感兴趣的读者可以参考相关资料。

表 5-4 priv_flags 的取值

priv_flags	描述
IFF_802_1Q_VLAN	标识是个 802.1Q VLAN 设备
IFF_EBRIDGE	标识是个以太网桥设备
IFF_SLAVE_INACTIVE	标识 bonding 的 slave 设备当前未激活
IFF_MASTER_8023AD	标识 bonding 为 802.3ad 模式
IFF_MASTER_ALB	标识 bonding 为 balance-alb 模式
IFF_BONDING	标识是 bonding 的 master 或 slave 设备
IFF_SLAVE_NEEDARP	标识 bonding 的 slave 设备支持 ARP

369 unsigned short padded

网络设备的 net_device 实例需 32 字节对齐，padded 为填充字节数。

371 unsigned char operstate

372 unsigned char link_mode

operstate 和 link_mode 与 The Interfaces Group MIB（RFC 2863）中的 IfOperStatus 相关，本书不作论述。

374 unsigned mtu

最大传输单元（MTU）。不同的网络设备对数据帧长度有不同的限制，以太网的 MTU 是 ETH_DATA_LEN，即 1500B。

375 unsigned short type

接口的硬件类型。ARP 模块处理中，用 type 来判断接口的硬件地址类型。对以太网接口该字段取值应为 ARPHRD_ETHER。

376 unsigned short hard_header_len

硬件首部的长度。以太网为 ETH_HLEN，即 14B。

378 struct net_device *master

在启用了 bonding 的网络负载均衡后，指向 bonding 的虚拟网络设备。

383 unsigned char perm_addr[MAX_ADDR_LEN]

硬件（MAC）地址，通常在初始化过程中从硬件中读取。

384 unsigned char addr_len

413 unsigned char dev_addr[MAX_ADDR_LEN]

416 unsigned char broadcast[MAX_ADDR_LEN]

硬件地址长度以及设备的硬件地址。以太网的地址长度是 6B，广播地址由 6 个 0xFF 字节组成。以太网驱动初始化时从设备中读取 MAC 地址后设置。在数据包交给驱动程序进行发送之前，要利用 MAC 地址生成正确的以太网数据首部。

385 unsigned short dev_id

未使用。

387 struct dev_mc_list *mc_list

388 int mc_count

这两个字段用来处理组播传输，mc_list 存储设置到该网络设备的组播硬件地址，而 mc_count 是 mc_list 所包含的项数。

```
183 struct dev_mc_list
184 {
185     struct dev_mc_list    *next;
186     __u8              dmi_addr[MAX_ADDR_LEN];
187     unsigned char         dmi_addrlen;
188     int               dmi_users;
189     int               dmi_gusers;
190 };
```

186 __u8 dmi_addr[MAX_ADDR_LEN]

用来存储组播硬件地址。

187 unsigned char dmi_addrlen

组播硬件地址的长度。

188 int dmi_users

不同的组播转换为组播硬件地址的数目。

190 int dmi_gusers

标识是不是通过 SIOCADDMULTI 选项添加的。

389 int promiscuity

设置网络设备混杂模式计数器。每一次设置或退出操作，该字段都会相应地加或减 1，只有当 promiscuity 为 0 时，网络设备才真正退出混杂模式。

390 int allmulti

设置网络设备接收所有组播包的计数器，每一次设置或退出操作，该字段都会相应地加或减 1，只有当 allmulti 为 0 时，网络设备才真正不再接收组播包。

395 void *atalk_ptr

396 void *ip_ptr

397 void *dn_ptr

398 void *ip6_ptr

399 void *ec_ptr

400 void *ax25_ptr

指向与特定协议族相关的配置块，如在 IPv4 中 ip_ptr 指向 in_device 结构的 IP 配置块。in_device 结构参见 6.1 节。

405 struct list_head poll_list ____cacheline_aligned_in_smp

net_device 结构实例通过该字段链接到 softnet_data 的 poll_list 成员上，参见 7.4 节的 softnet_data 结构。

408 int (*poll) (struct net_device *dev, int *quota)

NAPI 兼容驱动程序需提供该方法，用来以轮询模式操作接口，中断和轮询结合会显著提高性能。

409 int quota

读取数据包的配额，动态变化，由 netdev_budget 初始化，每次从网络设备中读取数据包后，

会从中减去本次读取的数据包数。当该配额小于或等于零时，结束当前轮询，等待下次轮询。这样即使某个网络设备有大量的数据包输入，也能保证其他网络设备能及时接收数据包。

在输入时，遍历网络设备轮询队列，从选定的网络设备中读取数据包，一旦已读取的数据包的数量超过配额，即停止本次读取，将该网络设备移至网络设备轮询队列的队尾，等待下次轮询。

```
410 int weight
```

数据包输入软中断中，单个网络设备读取数据包的配额。参见系统参数 dev_weight。

```
411 unsigned long last_rx
```

最近一次接收到数据包的时间，为接收时的 jiffies 值。

```
422 spinlock_t queue_lock ____cacheline_aligned_in_smp
```

进行队列操作自旋锁，防止多 CPU 并发操作。

```
423 struct Qdisc *qdisc
```

当前使用的根排队规则，配置的排队规则生效时由 qdisc_sleeping 设置。

```
424 struct Qdisc *qdisc_sleeping
```

当前配置的排队规则，生效时将被设置到 qdisc。

```
425 struct list_head qdisc_list
```

通过链表方式记录配置所在网络的所有排队规则。例如，使用了分类规则时，网络设备就会配置多个排队规则。

```
426 unsigned long tx_queue_len
```

可在设备发送队列中排队的最大数据包数。对以太网驱动默认值为 1000，也可以通过 ifconfig 修改，如 ifconfig eth0 txqueuelen 2000。

```
429 struct sk_buff *gso_skb
```

经软分割的 GSO 数据包，在输出流量控制过程的数据包输出软中断中，输出第一个数据包时，后续数据包暂时缓存到 gso_skb 上。在下次数据包输出软中断时，再从 gso_skb 中取出输出。

```
432 spinlock_t ingress_lock
```

防止多 CPU 并发排入输入排队规则的自旋锁。

```
433 struct Qdisc *qdisc_ingress
```

数据包输入的排队规则。

```
439 spinlock_t _xmit_lock ____cacheline_aligned_in_smp
```

发送数据包的自旋锁，以防止多 CPU 并发操作。

```
443 int xmit_lock_owner
```

正在通过该网络设备发送数据包的 CPU。当为-1 时，表示没有 CPU 通过当前网络设备发送数据包。

```
444 void *priv
```

私有数据，由 alloc_netdev()设置，最好通过 netdev_priv()进行访问。

```
445 int (*hard_start_xmit) (struct sk_buff *skb, struct net_device *dev)
```

驱动提供给上一层发送数据包的接口，在发送数据包时必定会调用该接口。

```
448 unsigned long trans_start
```

记录最近一次输出数据包的时间，为输出时的 jiffies 值。

450 int watchdog_timeo

网络层确定传输已超时，而调用驱动程序的 tx_timeout 接口的最短时间。

451 struct timer_list watchdog_timer

网络设备的软件狗，用于检测网络设备处于正常的工作状态时，是否存在由于关闭排队功能而导致发送超时的情况。一旦发生以上状况，就调用网络设备驱动的 tx_timeout 接口处理。

457 atomic_t refcnt ____cacheline_aligned_in_smp

网络设备的引用计数。

460 struct list_head todo_list

用来连接到 net todo list 链表上。

net_todo_list 链表包含已注销即将结束的网络设备。在调用 unregister_netdevice()注销设备后，会调用 net_set_todo()将注销的网络设备实例连接到 net_todo_list 上。然后 rtnl_unlock()会释放锁，并调用 netdev_run_todo()完成对网络设备的注销操作，参见 5.4.2 节。

462 struct hlist_node index_hlist

根据网络设备的索引，以散列表的形式组织到 dev_index_head 散列表中。这样就可以通过网络设备索引快速地定位到对应的网络设备。

465 enum { } reg_state

网络设备注册到链表中，用于标识网络设备的注册状态，见表 5-5。

表 5-5 reg_state 的取值

reg_state	描述
NETREG_UNINITIALIZED	网络设备实例，处于初始化未注册状态
NETREG_REGISTERED	完成网络设备的注册
NETREG_UNREGISTERING	正在注销网络设备，正从链表中移除
NETREG_UNREGISTERED	完成网络设备的注销（包括移除/sys 文件系统入口），已从链表中移除，但网络设备 net_device 结构实例还没有被释放
NETREG_RELEASED	即将释放网络设备 net_device 结构实例

473 void (*uninit)(struct net_device *dev)

驱动的销毁函数指针。如果设置了这个函数指针，则 unregister_netdevice()将调用该函数进行与 init 相反的操作。

475 void (*destructor)(struct net_device *dev)

destructor 一般不会被初始化，仅有少数虚拟设备使用它，通常被初始化为 free_netdev()或其封装函数，绝大多数设备驱动在 unregister_netdevice()后直接调用 free_netdev()。

478 int (*open)(struct net_device *dev)

启用设备函数指针，完成注册所需的系统资源，打开硬件及其所有设置。可用 ifconfig 命令启用网络设备。

479 int (*stop)(struct net_device *dev)

关闭设备函数指针，执行与启用时相反的操作。可用 ifconfig 工具关闭网络设备。

481 int (*hard_header) (struct sk_buff *skb, struct net_device *dev, unsigned short type, void *daddr, void *saddr, unsigned len);

根据先前检索到的源和目标硬件地址创建硬件首部。以太网网络设备对应的接口为

eth_header()。

```
487 int (*rebuild_header)(struct sk_buff *skb)
```

用来在传输包之前，完成 ARP 解析之后，重建硬件首部。

```
489 void (*set_multicast_list)(struct net_device *dev)
```

将组播地址列表更新到网络设备中。当设备的组播列表或设备标志发生变化时，调用此接口。

```
491 int (*set_mac_address)(struct net_device *dev, void *addr)
```

修改硬件地址接口，需网络设备支持该功能。

```
494 int (*do_ioctl)(struct net_device *dev, struct ifreq *ifr, int cmd)
```

ioctl 功能接口。如果设备不提供该功能，则将其设置为 NULL。

```
497 int (*set_config)(struct net_device *dev, struct ifmap *map)
```

修改设备接口配置接口，不过目前很多驱动程序通常提供其他接口。例如，ethtool 接口，参见 7.2 节。

```
500 int (*hard_header_cache)(struct neighbour *neigh, struct hh_cache *hh)
```

根据 ARP 查询的结果填充 hh_cache 结构。通常驱动程序都使用默认的 eth_header_cache()实现。

```
502 void (*header_cache_update)(struct hh_cache *hh, struct net_device
    *dev, unsigned char *haddr)
```

更新 hh_cache 结构中的目的地址。以太网设备使用 eth_header_cache_update()，参见 neigh_update_hhs()。

```
506 int (*change_mtu)(struct net_device *dev, int new_mtu)
```

如果驱动程序在修改 MTU 后需完成某些特定工作，则应该实现此函数，否则默认函数即可正确实现相关处理。

```
509 void (*tx_timeout) (struct net_device *dev)
```

如果数据包的传输在指定时间内失败（例如，数据包丢失或接口被上锁），这时会调用此接口，负责解决问题并重新开始数据包的传输。

```
511 void (*vlan_rx_register)(struct net_device *dev, struct vlan_group
    *grp)
513 void (*vlan_rx_add_vid)(struct net_device *dev, unsigned short vid)
515 void (*vlan_rx_kill_vid)(struct net_device *dev, unsigned short vid)
```

与 VLAN 相关的操作，本书不作论述。

```
518 int (*hard_header_parse)(struct sk_buff *skb, unsigned char *haddr)
```

从 skb 包含的数据包中获取 MAC 源地址，并将其复制到位于 haddr 的缓冲区，返回值为地址的长度。以太网设备使用 eth_header_parse()。

```
520 int (*neigh_setup)(struct net_device *dev, struct neigh_parms *)
```

用于设置与邻居子系统相关的参数，在创建邻居项时被回调，可以不实现。

```
522 struct netpoll_info *npinfo
```

网络设备的 netpoll 信息块，存储与 netpoll_info 相关的信息，由 netpoll_setup()设置。

```
525 void (*poll_controller)(struct net_device *dev)
```

该函数在禁止中断的情况下，要求驱动程序以轮询模式在接口上查询事件，通常用于特定

的内核网络任务中。例如，远程控制台和内核网络调试，模拟网络设备发生中断，从而进行中断处理，参见 7.9 节。

529 struct net_bridge_port *br_port

在创建一个桥设备时，指向 net_bridge_port 实例，本书不作论述。

532 struct class_device class_dev

534 struct attribute_group *sysfs_groups[3]

网络设备注册在/sys/class/net/中的实例。

5.2.2 网络设备有关结构的组织

net_device 结构中包含了驱动相关的所有信息，按信息的分类又把一些类型的信息组织到其他结构中，并嵌套在 net_device 结构中。例如与 IPv4 相关的配置存放在 in_device 结构中，它们之间的关系如图 5-1 所示。

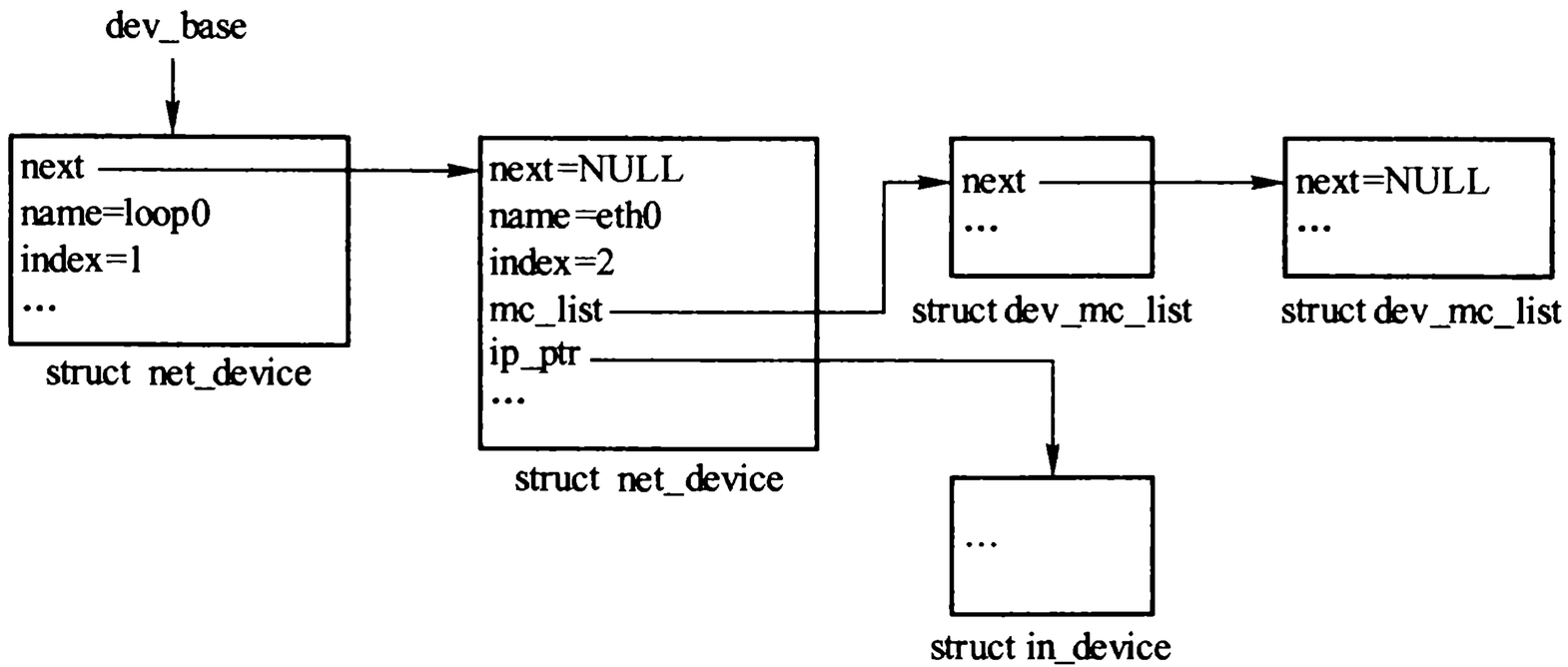

图 5-1 net_device 结构与 in_device 结构的关系

如图 5-2 所示，由于每个设备都有自定义的私有数据结构，net_device 结构全局链表可能连接不同长度的结点。分配说明如下：

- 图 5-2 展示了 net_device 内存分配效果。当调用 alloc_netdev()分配 net_device 结构时，与具体驱动程序有关的驱动程序私有数据块长度被传递给 alloc_netdev()，alloc_netdev()追加私有数据块到 net_device 结构实例的尾部。
- 图 5-2 也说明了 net_device 数据结构和可选的设备驱动私有数据结构之间的关系。通常，第二部分和第一部分一起分配，使用一个 kmalloc()就足够了，但也有一种情况，就是驱动程序更愿意自己分配其私有块。

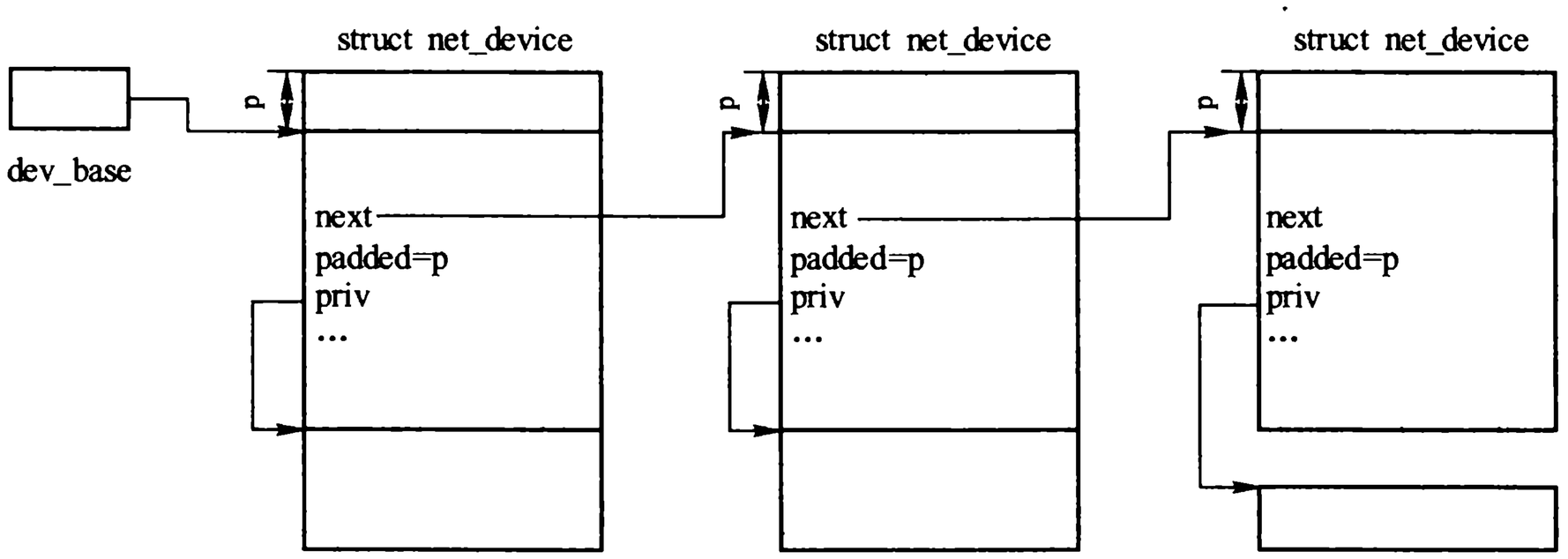

图 5-2 已注册设备全局链表

- dev_base 和 net_device 的 next 指针指向 net_device 结构的开始，而不是指向已分配块的开始。初始填充长度保存在 dev->padded 字段，该字段允许内核在适当的时候释放整个内存块。

如图 5-3 所示，注册后的 net_device 实例会被链入一个全局链表 dev_base 上，同时还加入到 dev_name_head 和 dev_index_head 这两个散列表，这些不同的链表使内核能够很容易查找需要的 net_device 实例。例如，获取一些统计信息，通过用户命令改变所有设备的配置，或者根据给定的标注查找匹配的设备。

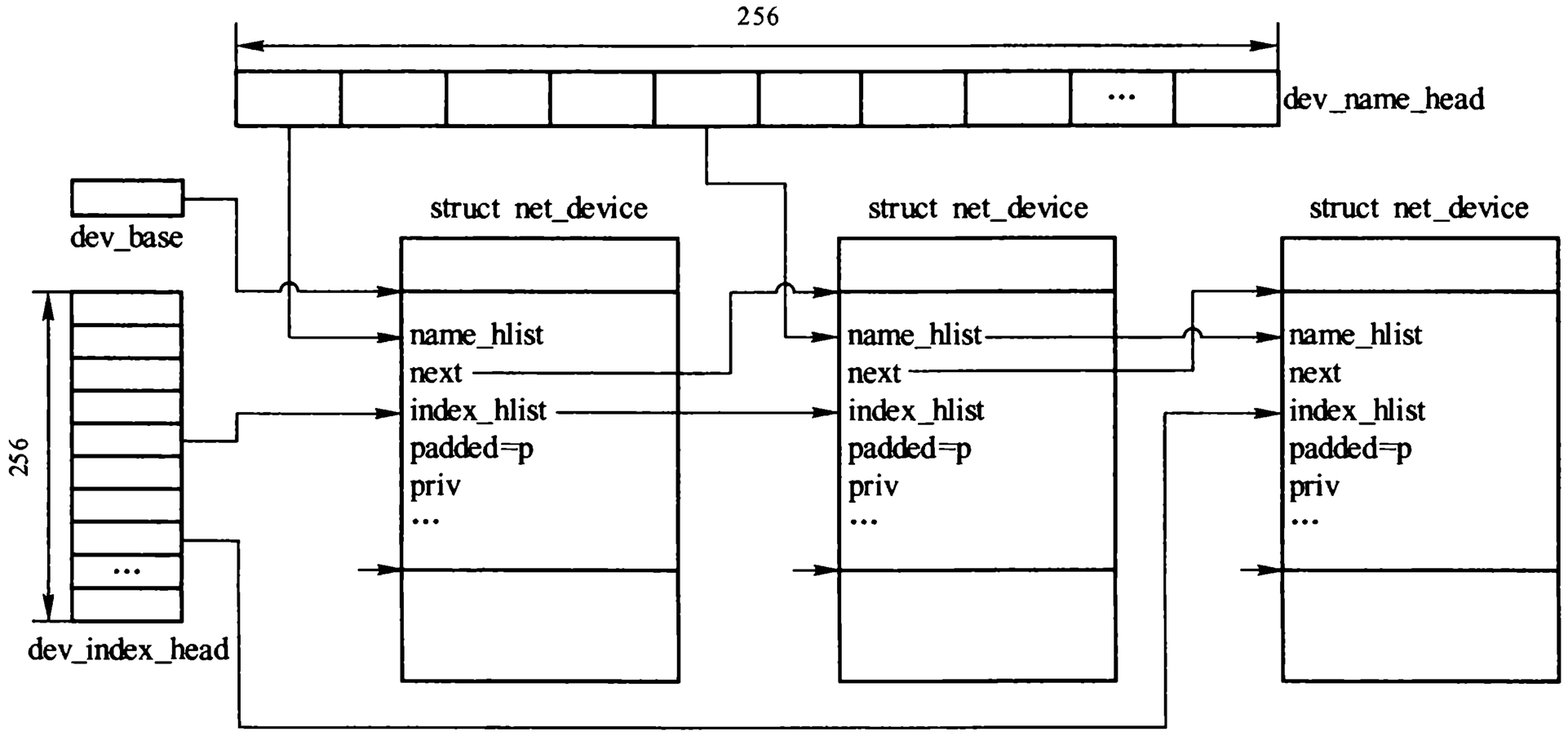

图 5-3 用于查找基于设备名和设备 ID 的设备实例的哈希表

dev_name_head 是以设备名为索引的散列表，非常有用。例如，当由 ioctl 接口改变一个配置时，早期的配置工具通过 ioctl 接口与内核交互，通常通过设备名来引用设备。

dev_index_head 是以设备 ID(dev->ifindex)为索引的散列表，交叉引用 net_device 结构通常要么存储设备 ID，要么存储指向 net_device 结构的指针。dev_index_head 对前一种情形很有用，新的 IP 配置工具通过 netlink 套接字和内核交互，通常通过设备 ID 引用设备。

绝大多数普通的查找都是基于设备名或设备 ID 的，这两种查找使用 dev_get_by_name()和 dev_get_by_index()实现的，在这两种查找中使用了以上两种散列表。而基于设备类型、MAC 地址等条件的搜索，则需通过 dev_base 链表来实现。在 dev_base 链表和两个散列表中的所有搜索，都由 dev_base_lock 锁保护。

5.2.3 相关函数

（1）struct net_device *dev_get_by_name(const char *name)

根据网络设备名获取网络设备。

（2）struct net_device *dev_get_by_index(int ifindex)

根据网络设备索引获取网络设备。

（3）struct net_device *dev_getbyhwaddr(unsigned short type, char *ha)

根据硬件地址获取网络设备。

（4）struct net_device *dev_getfirstbyhwtype(unsigned short type)

根据类型获取网络设备。

（5）struct net_device * dev_get_by_flags(unsigned short if_flags, unsigned short mask)
根据标志获取网络设备。

5.3 网络设备的注册

5.3.1 设备注册的时机

以下两种情形可导致触发注册网络设备：

（1）加载网络设备驱动程序

如果网络设备驱动程序被编译进内核，则将在启动时被初始化，在运行时被作为模块加载。无论初始化是否发生，由驱动程序控制的网络设备都会被注册。

这种情形适用于所有的总线类型，无论是总线体系结构还是模块初始化代码调用注册函数，结果都是一样的。例如：PCI 设备驱动程序如何加载以致执行 pci_driver->probe()，通常命名有几分像 *xxx*_probe 的函数，由驱动程序提供并兼顾设备注册。

（2）插入可热插拔网络设备

当用户插入一个热插拔网络设备时，内核通知其对应的驱动程序注册该网络设备。

5.3.2 分配 net_device 结构空间

1. alloc_netdev()

网络设备由 net device 结构定义，每个 net device 结构实例代表一个网络设备，该结构的实例由 alloc_netdev()分配空间。参数说明如下：

- sizeof_priv，指定用于存储驱动程序参数的私有数据块大小，参见 alloc_etherdrv()函数。
- name，设备名，通常是个前缀，相同前缀的设备会进行统一编号，以确保设备名唯一。
- setup，配置函数，用于初始化 net_device 结构实例的部分域，参见 ether_setup()函数。
- 返回值，如果不为 NULL，则为分配的 net_device 结构的指针，否则说明操作出错。

```
3172 struct net_device *alloc_netdev(int sizeof_priv, const char *name,
3173         void (*setup)(struct net_device *))
3174 {
3175     void *p;
3176     struct net_device *dev;
3177     int alloc_size;
3178
3179     BUG_ON(strlen(name) >= sizeof(dev->name));
3180
3181     /* ensure 32-byte alignment of both the device and private area */
3182     alloc_size = (sizeof(*dev) + NETDEV_ALIGN_CONST) & ~NETDEV_ALIGN_CONST;
3183     alloc_size += sizeof_priv + NETDEV_ALIGN_CONST;
3184
3185     p = kzalloc(alloc_size, GFP_KERNEL);
3186     if (!p) {
3187         printk(KERN_ERR "alloc_netdev: Unable to allocate device.\n");
3188         return NULL;
3189     }
3190
3191     dev = (struct net_device *)
```

```
3192        (((long)p + NETDEV_ALIGN_CONST) & ~NETDEV_ALIGN_CONST);
3193    dev->padded = (char *)dev - (char *)p;
3194
3195    if (sizeof_priv)
3196        dev->priv = netdev_priv(dev);
3197
3198    setup(dev);
3199    strcpy(dev->name, name);
3200    return dev;
3201 }
```

3182-3183 计算待分配 net_device 结构实例的长度，需要考虑 32 字节对齐的问题以及存储驱动程序参数的私有数据块大小。

3185-3189 分配 net_device 结构实例的内存。

3191-3193 将 net_device 结构实例的指针指向实例的内存开始地址 32 字节对齐处。

3195-3196 设置私有数据块的指针。

3198-3199 配置初始化 net_device 结构实例的部分成员。

3200 返回分配得到的 net_device 结构实例指针。

2. alloc_etherdev()

每个网络设备都会根据设备类型得到一个唯一名字，相同类型的设备，名称中的数字会按序递增。例如，多个太网设备的名称会是 eth0、eth1……。当设备名以 name%d（如 eth%d）的形式传递给 alloc_netdev()后，注册网络设备时会调用 dev_alloc_name()分配完整的名字，即根据设备类型将%d 改为第一个未分配的数字。

虽然 alloc_netdev()提供了分配 net_device 结构空间的功能，但是由于不同类型的设备有不同的名称及初始化函数，因此内核提供了一些封装 alloc_netdev()功能的函数。例如：alloc_etherdev()用于以太网设备，创建以字符串 eth 后跟唯一数字形式的设备名；而 ether_setup()是所有以太网网络设备的配置函数。通过封装 alloc_netdev()，alloc_etherdev()可设置以太网网络设备的私有数据块大小、设备名（"eth%d"）以及配置函数（ether_setup()）。

```
331 struct net_device *alloc_etherdev(int sizeof_priv)
332 {
333     return alloc_netdev(sizeof_priv, "eth%d", ether_setup);
334 }
```

3. ether_setup()

对于绝大多数普通的网络设备类型，都会用一个特定的 *xxx*_setup()初始化 net_device 实例的配置函数字段，这对所有同类型设备都是一样的。在 alloc_etherdev()中，将 ether _setup()作为第三个输入参数传递给 alloc_netdev()，ether_setup()就是以太网设备的 *xxx*_setup()。

```
296 void ether_setup(struct net_device *dev)
297 {
298     dev->change_mtu       = eth_change_mtu;
299     dev->hard_header    = eth_header;
300     dev->rebuild_header     = eth_rebuild_header;
301     dev->set_mac_address     = eth_mac_addr;
302     dev->hard_header_cache    = eth_header_cache;
303     dev->header_cache_update= eth_header_cache_update;
304     dev->hard_header_parse    = eth_header_parse;
```

```
305
306     dev->type        = ARPHRD_ETHER;
307     dev->hard_header_len     = ETH_HLEN;
308     dev->mtu         = ETH_DATA_LEN;
309     dev->addr_len        = ETH_ALEN;
310     dev->tx_queue_len    = 1000;    /* Ethernet wants good queues */
311     dev->flags       = IFF_BROADCAST|IFF_MULTICAST;
312
313     memset(dev->broadcast, 0xFF, ETH_ALEN);
314
315 }
```

5.3.3 网络设备注册过程

图 5-4 所示为在 e100 网络驱动程序的注册过程中，主要函数的调用过程。不同类型的网络设备，注册过程是相同的。

注册过程中，首先由 alloc_etherdev()为网络设备分配 net_device 结构，并初始化所有以太网设备通用的参数，然后初始化 net_device 结构其他部分，并调用 register_netdev()结束设备的注册过程。

注销过程比较简单，都会调用 unregister_netdevice()和 free_netdev()，其他操作与设备驱动程序特性相关，如释放设备使用的所有资源，即 IRQ、内存映像等。

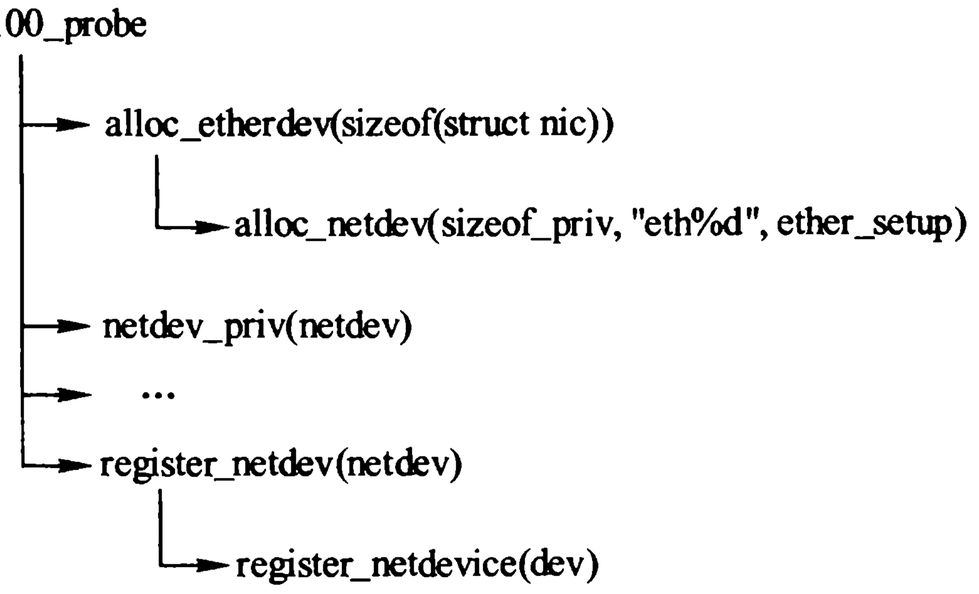

图 5-4 网络设备的注册和注销过程

注册过程并不是简单地把 net_device 结构实例插入到全局链表或相关的散列表中，还包括初始化 net_device 结构实例的部分成员，产生一个广播形式通知其他内核组件其已注册的消息，以及其他任务。网络设备由 register_netdev()注册，该函数只是简单地封装了 register_netdevice()，封装过程中为网络设备确定待注册的具体名称。

```
3012 int register_netdev(struct net_device *dev)
3013 {
3014     int err;
3015
3016     rtnl_lock();
3017
3018     /*
3019      * If the name is a format string the caller wants us to do a
3020      * name allocation.
3021      */
3022     if (strchr(dev->name, '%')) {
```

```
3023        err = dev_alloc_name(dev, dev->name);
3024        if (err < 0)
3025            goto out;
3026    }
3027
3028    err = register_netdevice(dev);
3029 out:
3030    rtnl_unlock();
3031    return err;
3032 }
```

当待注册的网络设备名确定之后，便调用 register_netdevice()注册网络设备，并将网络设备描述符注册到系统中。完成注册后，会发送 NETDEV_REGISTER 消息到 netdev_chain 通知链中，使得所有对设备注册感兴趣的模块都能接收消息。

```
2877 int register_netdevice(struct net_device *dev)
2878 {
2879    struct hlist_head *head;
2880    struct hlist_node *p;
2881    int ret;
2882
2883    BUG_ON(dev_boot_phase);
2884    ASSERT_RTNL();
2885
2886    might_sleep();
2887
2888    /* When net_device's are persistent, this will be fatal. */
2889    BUG_ON(dev->reg_state != NETREG_UNINITIALIZED);
2890
2891    spin_lock_init(&dev->queue_lock);
2892    spin_lock_init(&dev->_xmit_lock);
2893    dev->xmit_lock_owner = -1;
2894 #ifdef CONFIG_NET_CLS_ACT
2895    spin_lock_init(&dev->ingress_lock);
2896 #endif
2897
2898    dev->iflink = -1;
2899
2900    /* Init, if this function is available */
2901    if (dev->init) {
2902        ret = dev->init(dev);
2903        if (ret) {
2904            if (ret > 0)
2905                ret = -EIO;
2906            goto out;
2907        }
2908    }
2909
2910    if (!dev_valid_name(dev->name)) {
2911        ret = -EINVAL;
2912        goto out;
2913    }
2914
2915    dev->ifindex = dev_new_index();
2916    if (dev->iflink == -1)
```

```
2917        dev->iflink = dev->ifindex;
2918
2919    /* Check for existence of name */
2920    head = dev_name_hash(dev->name);
2921    hlist_for_each(p, head) {
2922        struct net_device *d
2923            = hlist_entry(p, struct net_device, name_hlist);
2924        if (!strncmp(d->name, dev->name, IFNAMSIZ)) {
2925            ret = -EEXIST;
2926             goto out;
2927        }
2928     }
2929
2930    /* Fix illegal SG+CSUM combinations. */
2931    if ((dev->features & NETIF_F_SG) &&
2932        !(dev->features & NETIF_F_ALL_CSUM)) {
2933        printk(KERN_NOTICE "%s: Dropping NETIF_F_SG since no checksum feature.\n",
2934               dev->name);
2935        dev->features &= ~NETIF_F_SG;
2936    }
2937
2938    /* TSO requires that SG is present as well. */
2939    if ((dev->features & NETIF_F_TSO) &&
2940        !(dev->features & NETIF_F_SG)) {
2941        printk(KERN_NOTICE "%s: Dropping NETIF_F_TSO since no SG feature.\n",
2942               dev->name);
2943        dev->features &= ~NETIF_F_TSO;
2944    }
2945    if (dev->features & NETIF_F_UFO) {
2946        if (!(dev->features & NETIF_F_HW_CSUM)) {
2947            printk(KERN_ERR "%s: Dropping NETIF_F_UFO since no "
2948                   "NETIF_F_HW_CSUM feature.\n",
2949                         dev->name);
2950            dev->features &= ~NETIF_F_UFO;
2951        }
2952        if (!(dev->features & NETIF_F_SG)) {
2953            printk(KERN_ERR "%s: Dropping NETIF_F_UFO since no "
2954                   "NETIF_F_SG feature.\n",
2955                   dev->name);
2956            dev->features &= ~NETIF_F_UFO;
2957        }
2958    }
2959
2960    /*
2961     *    nil rebuild_header routine,
2962     *    that should be never called and used as just bug trap.
2963     */
2964
2965    if (!dev->rebuild_header)
2966        dev->rebuild_header = default_rebuild_header;
2967
2968    ret = netdev_register_sysfs(dev);
2969    if (ret)
2970        goto out;
2971    dev->reg_state = NETREG_REGISTERED;
2972
```

```
2973    /*
2974     *    Default initial state at registry is that the
2975     *    device is present.
2976     */
2977
2978    set_bit(__LINK_STATE_PRESENT, &dev->state);
2979
2980    dev->next = NULL;
2981    dev_init_scheduler(dev);
2982    write_lock_bh(&dev_base_lock);
2983    *dev_tail = dev;
2984    dev_tail = &dev->next;
2985    hlist_add_head(&dev->name_hlist, head);
2986    hlist_add_head(&dev->index_hlist, dev_index_hash(dev->ifindex));
2987    dev_hold(dev);
2988    write_unlock_bh(&dev_base_lock);
2989
2990    /* Notify protocols, that a new device appeared. */
2991    raw_notifier_call_chain(&netdev_chain, NETDEV_REGISTER, dev);
2992
2993    ret = 0;
2994
2995 out:
2996    return ret;
2997 }
```

2886 2.6 版本的内核支持内核抢占，might_sleep 宏检查是否需要重新调度，如果是，则进行调度，无论此时进程执行在内核空间还是用户空间。

2891-2898 初始化网络设备描述符中的部分成员，包括用于锁操作的成员等。

2901-2908 如果设备驱动程序提供了初始化函数，则进行相关初始化。

2910-2913 通过 dev_valid_name()检测待注册的网络设备名是否有效。

2915-2917 调用 dev_new_index()为设备分配一个唯一索引号和一个用于虚拟隧道设备的唯一标识。索引号由一个 32 位计数器产生，每当一个新设备加到系统中计数器就会递增。

2920-2928 将网络设备添加到 dev_name_head 散列表中，并检测是否存在同名的网络设备。

2931-2936 检测网络设备 NETIF_F_SG 和 NETIF_F_SG 特性的组合，只有在网络设备支持校验和计算的情况下，网络设备才能支持 SG 类型的聚合分散 I/O，因为 SG 类型的聚合分散 I/O 特性没有传输层硬件校验和支持是无用的。

2939-2944 检测网络设备 NETIF_F_TSO 和 NETIF_F_SG 特性的组合。TSO 需要 SG 类型的聚合分散 I/O 的支持，因此在后者不被支持时也将被禁用。

2945-2958 检测网络设备的 NETIF_F_UFO 和 NETIF_F_HW_CSUM，以及 NETIF_F_SG 特性的组合。UFO 需要 NETIF_F_HW_CSUM 和 SG 类型的聚合分散 I/O 的支持，因此在后两者不被支持时也将被禁用。

2965-2966 初始化网络设备用于重建硬件首部的 rebuild_header 接口。

2968-2970 将网络设备信息注册到 sysfs 文件系统中。

2971 将网络设备的注册状态设置到 NETREG_REGISTERED，表示注册已完成。

2978 设置网络设备的__LINK_STATE_PRESENT 标志，标识设备对系统是可用的。

2980-2988 初始化网络设备排队规则，并注册到网络设备的链表和相关散列表上。

2991 最后通过 netdev_chain 通知链通知所有对设备注册感兴趣的其他内核模块。

5.3.4 注册设备的状态迁移

网络设备通过 register_netdev()和 unregister_netdev()注册与注销。在注册、注销以及释放过程中，伴随着网络设备注册状态的迁移。

图 5-5 所示为网络设备各种注册状态的迁移过程，以及其他关键函数调用的地方。

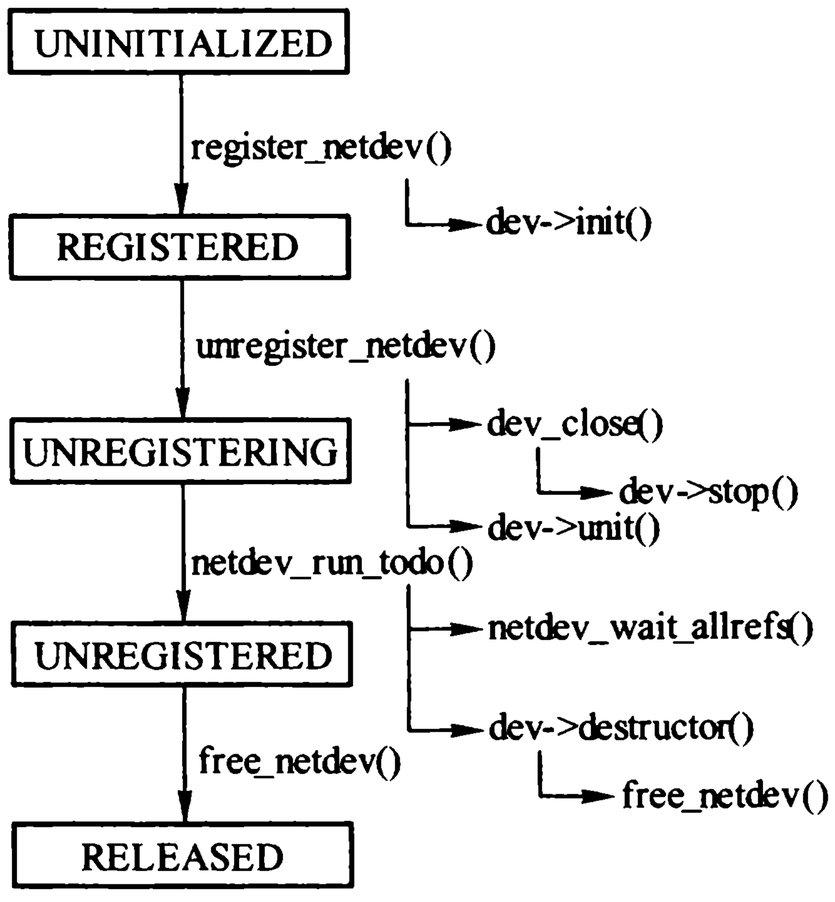

图 5-5 网络设备注册状态

- 网络设备的初始化状态为 UNINITIALIZED，在调用 register_netdev()完成注册后，状态便迁移到 REGISTERED。
- 处于 REGISTERED 状态的网络设备，通过 unregister_netdev()注销后，迁移到 UNREGISTERING 状态。同时 net_device 的两个虚拟函数 init()和 uninit()在注册和注销设备时分别初始化和清除私有数据。
- 在衔接操作中，设备真正被注销要到所有对该实例的相关引用都释放时才执行，netdev_wait_allrefs()直到条件满足才会返回，此时状态迁移到 UNINITIALIZED。
- 注销后的网络设备在调用 free_netdev()后，释放之前，状态迁移到 RELEASED。

5.3.5 设备注册状态通知

内核其他模块和用户空间应用程序可能都想知道网络设备注册、注销、打开、关闭的时间，因此提供了两个产生事件通知的途径，即 netdev_chain 通知链和 netlink 的 RTMGRP_LINK 组播组。内核模块只要注册到 netdev_chain 通知链上，网络设备的相关事件都会通知该模块，而用户空间应用程序只要注册到 netlink 的 RTMGRP_LINK 组播组，网络设备的相关事件也会通知到该应用程序。

1. netdev_chain 通知链

内核模块可以通过 register_netdevice_notifier()将处理网络设备事件的函数注册到 netdev_chain 通知链中，之后可以通过 unregister_netdevice_notifier()注销。并且可以对一个或多个事件感兴趣。系统中有多个内核模块注册到 netdev_chain 通知链，如路由（路由子系统通过通知链增删设备相关的路由入口）、防火墙（防火墙缓存了当前已禁止的设备的数据包，必须丢

弃这些包或者根据自己的策略作其他处理)、协议处理（当改变本地设备的 MAC 地址时，ARP 表必须作相应的更新)。

需要注意的是，注册到通知链时，register_netdevice_notifier()会将以前的 NETDEV_REGISTER 和 NETDEV_UP 通知重发给系统中当前注册的模块。

目前提供了以下几个事件，见表 5-6。

表 5-6 网络设备的事件

网络设备的事件	描述
NETDEV_UP	网络设备使能时触发该事件，由 dev_open 产生
NETDEV_GOING_DOWN	网络设备将要被禁止时触发该事件，由 dev_close 产生
NETDEV_DOWN	网络设备已经被禁止时触发该事件，由 dev_close 产生
NETDEV_REGISTER	网络设备已经完成注册时触发该事件，由 register_netdevice 产生
NETREG_UNREGISTERING	网络设备正在进行注销时触发该事件，由 unregister_netdevice 产生
NETDEV_UNREGISTER	网络设备已经完成注销时触发该事件，由 unregister_netdevice 产生
NETDEV_REBOOT	网络设备由于硬件错误而重启，目前不使用，保留
NETDEV_CHANGEADDR	网络设备硬件地址(或相关联的广播地址)已经改变
NETDEV_CHANGENAME	网络设备的名称发生改变
NETDEV_CHANGE	网络设备状态或配置发生改变时触发该事件，通常网络设备的 flags 标志改变时产生

2. netlink 链接通知

当设备状态或配置改变时，通知被发送到连接组播组 RTMGRP_LINK(带有 rtmsg_ifinfo 接口信息)。事实上，那些发送到连接组播组 RTMGRP_LINK 的通知也是由 netdev_chain 通知链驱动的，为能及时通知 netlink 链，在 netdev_chain 通知链也注册了一个实例，通过该实例发送通知到 netlink 链上。

5.3.6 引用计数

net_device 结构实例直到对其的所有引用都释放后才会被释放，net_device 结构的 refcnt 字段用来保存引用计数，在 dev_hold()和 dev_put()会对该字段分别递增和递减。

当 register_netdevice()注册设备时，引用计数 refcnt 被初始化为 1，但网络设备不像其他内核对象在引用计数减为 0 时由 *xxx*_put()释放，而是直到从内核注销时，即调用 unregister_netdevice()时，引用计数才减为 0，然后释放设备。

在这之前调用 dev_put()并不真正释放并删除 net_device 实例，而只是释放一次引用。在注销后设备就不再可用，并且会发送 NETDEV_UNREGISTER 通知给 netdev_chain 通知链，以便所有的引用者在接收到通知后释放它们对设备的引用。

5.4 网络设备的注销

5.4.1 设备注销的时机

以下两种情形会导致设备注销：

- 卸载网络设备驱动程序。这只适用于驱动程序作为模块加载的情形，而不适于编译进内

核的情形。当管理员卸载网络设备驱动程序时，所有相关网络设备的驱动程序都被注销，例如实际网络设备被卸载后，所有相关的虚拟网络设备都将被注销。

- 移除热拔插网络设备。当用户移除内核支持热拔插功能的系统中正在运行的热拔插网络设备时，网络设备驱动程序被注销。

5.4.2 网络设备注销过程

图 5-6 所示为在 e100 网络驱动程序的注销过程中，主要函数的调用过程。对其他类型的网络设备，注销过程是相同的。

1. unregister_netdevice()

通常通过调用 unregister_netdev()来完成网络设备的注销，实际上，该函数只是调用了 unregister_netdevice()，在调用前后作了同步控制。当解锁后会激活衔接操作，如图 5-7 所示。

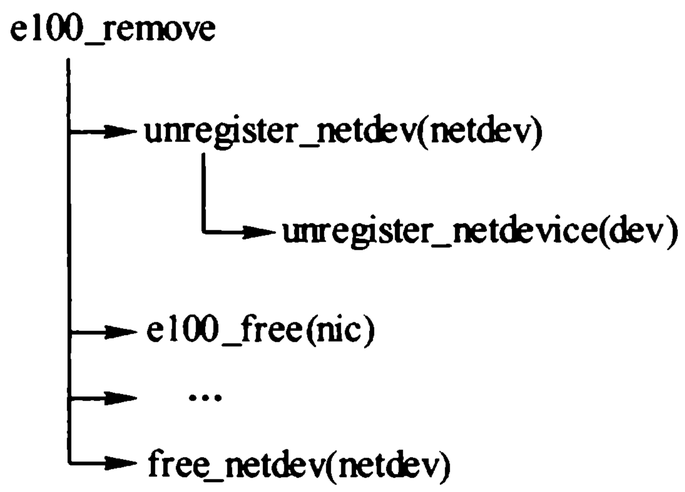

图 5-6 e100 网络设备的注销过程

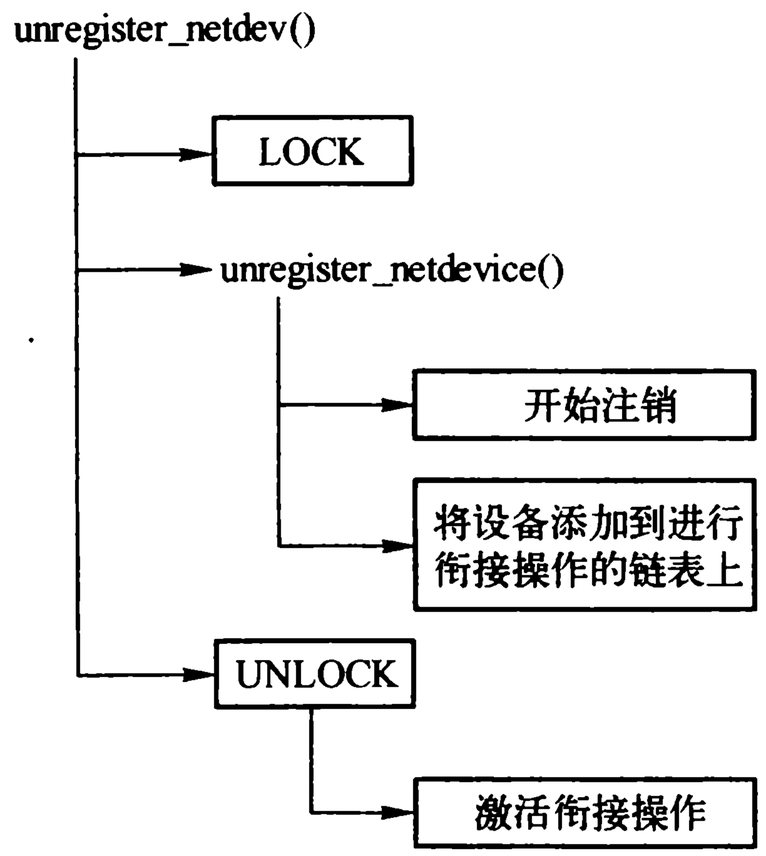

图 5-7 unregister_netdev()与衔接操作

为了注销网络设备，内核和相关的网络设备驱动程序需要撤销所有注册时执行的操作，以及以下操作：

1）通过 dev_close()禁止网络设备。

2）释放所有分配的资源，如 IRQ、I/O 内存、I/O 端口等。

3）从全局队列 dev_base、dev_name_head 和 dev_index_head 散列表中移除 net_device 实例。

4）一旦实例的引用为 0，就释放 net_device 实例、驱动程序私有数据结构及其他连接到它的内存块。net_device 实例由 free_netdev()释放，如果内核编译时支持 sysfs，free_netdev()会让 sysfs 来负责释放。

5）移除添加到 proc 和 sys 文件系统的任何文件。

```
3250 int unregister_netdevice(struct net_device *dev)
3251 {
3252     struct net_device *d, **dp;
3253
3254     BUG_ON(dev_boot_phase);
3255     ASSERT_RTNL();
3256
3257     /* Some devices call without registering for initialization unwind. */
3258     if (dev->reg_state == NETREG_UNINITIALIZED) {
```

```
3259        printk(KERN_DEBUG "unregister_netdevice: device %s/%p never "
3260                "was registered\n", dev->name, dev);
3261        return -ENODEV;
3262    }
3263
3264    BUG_ON(dev->reg_state != NETREG_REGISTERED);
3265
3266    /* If device is running, close it first. */
3267    if (dev->flags & IFF_UP)
3268        dev_close(dev);
```

3258-3262 如果设备处于 NETREG_UNINITIALIZED 状态，即未初始化状态，则输出信息后返回。

3267-3268 如果设备还没有被关闭，则调用 dev_close()进行关闭。

```
3270    /* And unlink it from device chain. */
3271    for (dp = &dev_base; (d = *dp) != NULL; dp = &d->next) {
3272        if (d == dev) {
3273            write_lock_bh(&dev_base_lock);
3274            hlist_del(&dev->name_hlist);
3275            hlist_del(&dev->index_hlist);
3276            if (dev_tail == &dev->next)
3277                dev_tail = dp;
3278            *dp = d->next;
3279            write_unlock_bh(&dev_base_lock);
3280            break;
3281        }
3282    }
3283    if (!d) {
3284        printk(KERN_ERR "unregister net_device: '%s' not found\n",
3285                dev->name);
3286        return -ENODEV;
3287    }
3288
3289    dev->reg_state = NETREG_UNREGISTERING;
3290
3291    synchronize_net();
```

3270-3287 将待注销的网络设备实例从全局链表 dev_base 及 dev_name_head、dev_index_head 散列表中移除。移除后不能阻止内核子系统使用该设备，它们仍然拥有指向该 net device 结构实例的指针，只有当引用计数为 0 时才会真正释放实例。

3289 将网络设备实例状态设置为 NETREG_UNREGISTERING，即未注册状态。

3291 同步数据包的接收处理。

```
3293    /* Shutdown queueing discipline. */
3294    dev_shutdown(dev);
3295
3296
3297    /* Notify protocols, that we are about to destroy
3298       this device. They should clean all the things.
3299    */
3300    raw_notifier_call_chain(&netdev_chain, NETDEV_UNREGISTER, dev);
3301
```

```
3302    /*
3303     *    Flush the multicast chain
3304     */
3305    dev_mc_discard(dev);
3306
3307    if (dev->uninit)
3308        dev->uninit(dev);
3309
3310    /* Notifier chain MUST detach us from master device. */
3311    BUG_TRAP(!dev->master);
3312
3313    /* Finish processing unregister after unlock */
3314    net_set_todo(dev);
3315
3316    synchronize_net();
3317
3318    dev_put(dev);
3319    return 0;
3320 }
```

3294 释放所有与设备相关的队列规则实例。

3300 发送 NETDEV_UNREGISTER 消息到 netdev_chain 通知链上，以便通知对设备状态改变有兴趣的其他内核组件。

3305 释放设置到网络设备上的组播 MAC 地址等信息。

3307-3308 进行驱动程序相关的销毁操作，通常是销毁那些在 init 中初始化的数据。

3314-3319 通过 net_set_todo()调用 net_run_todo()结束注销过程，调用 dev_put()递减引用计数。net_run_todo()从 sysfs 中注销设备，并使网络设备进入完成注销状态，等到所有的引用都释放后，调用 dev->destructor()结束注销过程。

2．衔接操作：netdev_run_todo()

net_device 结构实例的改变受 rtnl_mutex 原子量的保护，在修改 net_device 实例前后需要调用 rtnl_lock()和 rtnl_unlock()。

因此 unregister_netdev()在开始时获得锁，在返回前释放锁。一旦 unregister_netdevice()完成了它的工作，会通过 net_set_todo()将完成注销的 net_device 结构实例加入到 net_todo_list 中，这个链表包含了注销已结束的设备。由于互斥变量 net_todo_run_mutex 控制了其串行化，因此在同一时刻仅能有一个 CPU 运行 net_run_todo()。

netdev_run_todo()函数用来处理队列 net_todo_list 上的网络设备，继续处理相关的注销事务。主要是注销 sysfs 中该设备的结点。注销时，等待设备的引用计数为 0，再调用设备自身的 destructor()，完成注销过程。

```
3108 static DEFINE_MUTEX(net_todo_run_mutex);
3109 void netdev_run_todo(void)
3110 {
3111     struct list_head list;
3112
3113     /* Need to guard against multiple cpu's getting out of order. */
3114     mutex_lock(&net_todo_run_mutex);
3115
3116     /* Not safe to do outside the semaphore.  We must not return
3117      * until all unregister events invoked by the local processor
```

```
3118     * have been completed (either by this todo run, or one on
3119     * another cpu).
3120     */
3121    if (list_empty(&net_todo_list))
3122       goto out;
3123
3124    /* Snapshot list, allow later requests */
3125    spin_lock(&net_todo_list_lock);
3126    list_replace_init(&net_todo_list, &list);
3127    spin_unlock(&net_todo_list_lock);
3128
3129    while (!list_empty(&list)) {
3130       struct net_device *dev
3131          = list_entry(list.next, struct net_device, todo_list);
3132       list_del(&dev->todo_list);
3133
3134       if (unlikely(dev->reg_state != NETREG_UNREGISTERING)) {
3135          printk(KERN_ERR "network todo '%s' but state %d\n",
3136                 dev->name, dev->reg_state);
3137          dump_stack();
3138          continue;
3139       }
3140
3141       netdev_unregister_sysfs(dev);
3142       dev->reg_state = NETREG_UNREGISTERED;
3143
3144       netdev_wait_allrefs(dev);
3145
3146       /* paranoia */
3147       BUG_ON(atomic_read(&dev->refcnt));
3148       BUG_TRAP(!dev->ip_ptr);
3149       BUG_TRAP(!dev->ip6_ptr);
3150       BUG_TRAP(!dev->dn_ptr);
3151
3152       /* It must be the very last action,
3153        * after this 'dev' may point to freed up memory.
3154        */
3155       if (dev->destructor)
3156          dev->destructor(dev);
3157    }
3158
3159 out:
3160    mutex_unlock(&net_todo_run_mutex);
3161 }
```

3114 通过互斥变量 net_todo_run_mutex 控制其串行化，因此在同一时刻仅能有一个 CPU 运行 net_run_todo()。

3121-3122 net_todo_list 队列不为空时才能继续处理。

3125-3137 从 net_todo_list 队列中取出待处理的网络设备描述符到临时队列中，并清除 net_todo_list 队列，使得系统在衔接操作时，其他 CPU 能继续操作 net_todo_list 队列。

3129 循环处理链表上的网络设备描述符，直至完成为止。

3130-3132 取出一个待处理的网络设备描述符，并将其从临时链表中移除。

3134-3139 由于当前网络设备正处于注销过程中，因此再次检测该网络设备的状态。

3141-3142 将该网络设备从 sysfs 文件系统中移除，并将状态设置为 NETREG_UNREGISTERED，即未注册状态。

3144 等待直到待注销的网络设备没有引用为止。

3155-3156 调用网络设备自身的 destructor()，完成注销过程。

3159-3160 返回前解锁 net_todo_run_mutex 互斥变量。

netdev_wait_allrefs()由一个循环组成，直到网络设备的引用计数值减到 0 结束。等待过程中每秒发送一个 NETDEV_UNREGISTER 通知，每 10s 发送一个警告到控制台。在发送通知时，如果发生连接状态改变事件，则一定要处理，流程如图 5-8 所示。

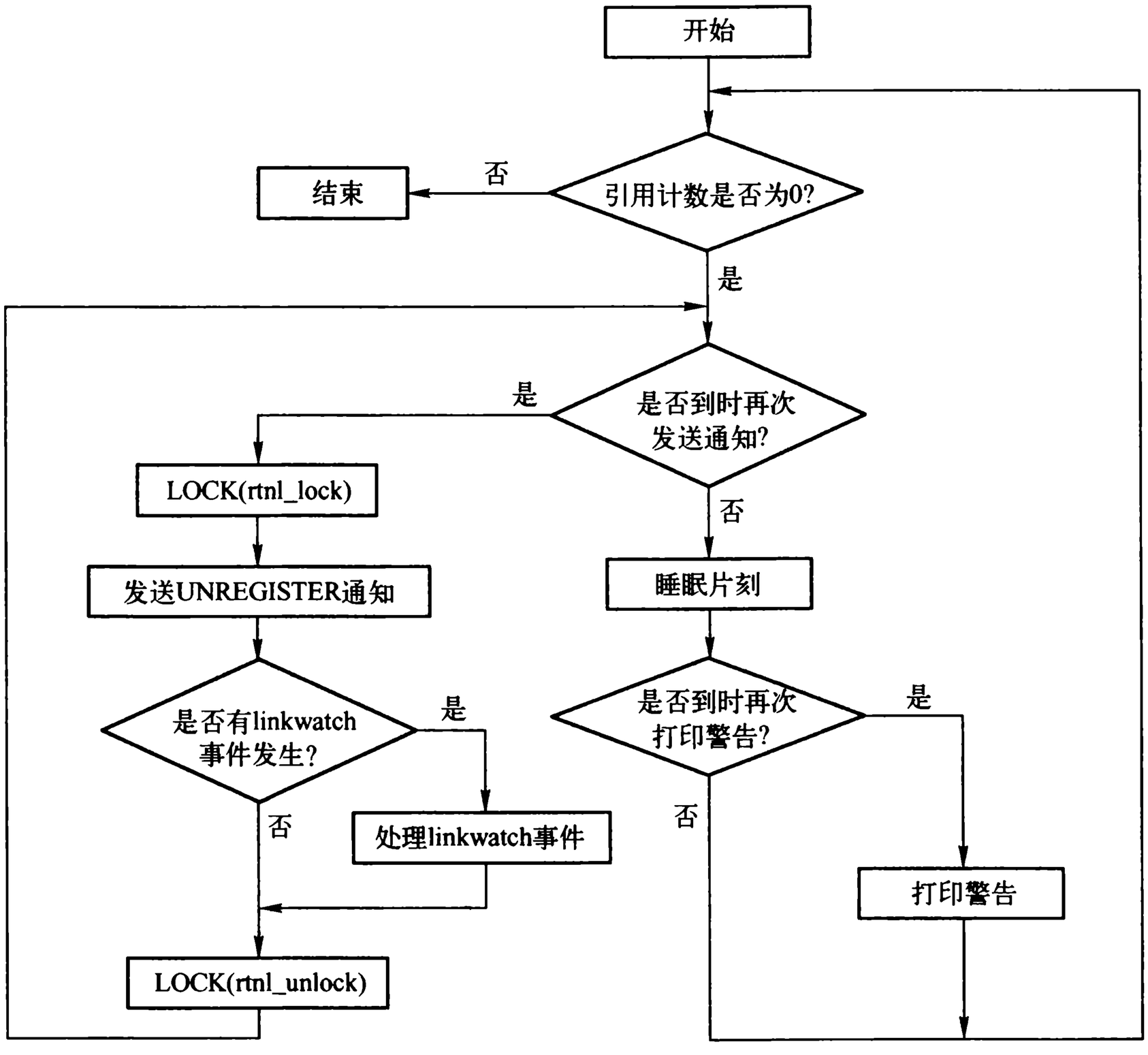

图 5-8 netdev_wait_allrefs()流程

```
3046 static void netdev_wait_allrefs(struct net_device *dev)
3047 {
3048    unsigned long rebroadcast_time, warning_time;
3049
3050    rebroadcast_time = warning_time = jiffies;
3051    while (atomic_read(&dev->refcnt) != 0) {
3052       if (time_after(jiffies, rebroadcast_time + 1 * HZ)) {
3053          rtnl_lock();
3054
3055          /* Rebroadcast unregister notification */
3056          raw_notifier_call_chain(&netdev_chain,
3057                   NETDEV_UNREGISTER, dev);
3058
3059          if (test_bit(__LINK_STATE_LINKWATCH_PENDING,
```

```
3060                &dev->state)) {
3061            /* We must not have linkwatch events
3062             * pending on unregister. If this
3063             * happens, we simply run the queue
3064             * unscheduled, resulting in a noop
3065             * for this device.
3066             */
3067            linkwatch_run_queue();
3068        }
3069
3070        __rtnl_unlock();
3071
3072            rebroadcast_time = jiffies;
3073        }
3074
3075        msleep(250);
3076
3077        if (time_after(jiffies, warning_time + 10 * HZ)) {
3078            printk(KERN_EMERG "unregister_netdevice: "
3079                  "waiting for %s to become free. Usage "
3080                  "count = %d\n",
3081                  dev->name, atomic_read(&dev->refcnt));
3082            warning_time = jiffies;
3083        }
3084    }
3085 }
```

3051 循环等待，直至待处理的网络设备引用值为 0。

3052-3073 在等待中，每一秒广播一次 NETDEV_UNREGISTER 消息。网络设备在注销期间，如果发生连接状态改变事件，则一定要处理，参见 5.8 节。

3075 进行短暂的睡眠。

3077-3082 在等待过程中，当等待时间超过 10s，则会每 10s 打印一次告警信息。

5.5 网络设备的启用

设备一旦注册后即可使用，但必须在用户或用户空间应用程序使能后才可以收发数据。因为注册到系统中的网络设备，其初始状态是关闭的，此时是不能传输数据的，必须激活后，网络设备才能进行数据的传输。在应用层，可以通过 ifconfig up 命令（最终是通过 ioctl 的 SIOCSIFFLAGS）来激活网络设备。而 SIOCSIFFLAGS 命令是通过 dev_change_flags()调用 dev_open()来激活网络设备。

dev_open()将网络设备从关闭状态转到激活状态，并发送一个 NETDEV_UP 消息到网络设备状态改变通知链上。

```
836 int dev_open(struct net_device *dev)
837 {
838     int ret = 0;
839
840     /*
841      *    Is it already up?
842      */
```

```
843
844     if (dev->flags & IFF_UP)
845         return 0;
846
847     /*
848      *    Is it even present?
849      */
850     if (!netif_device_present(dev))
851         return -ENODEV;
852
853     /*
854      *    Call device private open method
855      */
856     set_bit(__LINK_STATE_START, &dev->state);
857     if (dev->open) {
858         ret = dev->open(dev);
859         if (ret)
860             clear_bit(__LINK_STATE_START, &dev->state);
861     }
862
863      /*
864      *    If it went open OK then:
865      */
866
867     if (!ret) {
868         /*
869          *    Set the flags.
870          */
871         dev->flags |= IFF_UP;
872
873         /*
874          *    Initialize multicasting status
875          */
876         dev_mc_upload(dev);
877
878         /*
879          *    Wakeup transmit queue engine
880          */
881         dev_activate(dev);
882
883         /*
884          *    ... and announce new interface.
885          */
886         raw_notifier_call_chain(&netdev_chain, NETDEV_UP, dev);
887     }
888     return ret;
889 }
```

844-845 若网络设备已经启用，则无需再进行操作。

850-851 如果网络设备已经被挂起，则不能被激活。

856-861 设置网络设备的启用状态标志。如果实现 open 函数，则根据具体硬件注册系统资源，使能硬件，并对设备作其他的一些设置。

867-888 如果启用网络设备成功，则设置网络设备的已启用标志，并更新组播地址列表到网络设备中，网络设备设置为传递状态。调用 dev_activate()初始化用于流量控制的排队规

则，并启动监视定时器。如果用户没有配置流量控制，则指定为默认的先进先出(FIFO)队列。最后，发送 NETDEV_UP 消息到网络设备状态改变通知链上，以通知对设备使能有兴趣的其他内核组件。

5.6 网络设备的禁用

既然可以启用网络设备，当然也可以禁止网络设备，网络设备一旦关闭后就不能传输数据了。网络设备能被用户命令明确地或被其他事件隐含地禁止。在应用层，可以通过 ifconfig down 命令（最终是通过 ioctl 的 SIOCSIFFLAGS）来关闭网络设备，或者在网络设备注销时被禁止。SIOCSIFFLAGS 命令通过 dev_change_flags()，根据网络设备当前的状态来确定调用 dev_close()关闭网络设备。

dev_close()将网络设备从激活状态转换到关闭状态，并发送 NETDEV_GOING_DOWN 和 NETDEV_DOWN 消息到网络设备状态改变通知链上。

```
900 int dev_close(struct net_device *dev)
901 {
902     if (!(dev->flags & IFF_UP))
903         return 0;
904
905     /*
906      *    Tell people we are going down, so that they can
907      *    prepare to death, when device is still operating.
908      */
909     raw_notifier_call_chain(&netdev_chain, NETDEV_GOING_DOWN, dev);
910
911     dev_deactivate(dev);
912
913     clear_bit(__LINK_STATE_START, &dev->state);
914
915     /* Synchronize to scheduled poll. We cannot touch poll list,
916      * it can be even on different cpu. So just clear netif_running(),
917      * and wait when poll really will happen. Actually, the best place
918      * for this is inside dev->stop() after device stopped its irq
919      * engine, but this requires more changes in devices. */
920
921     smp_mb__after_clear_bit(); /* Commit netif_running(). */
922     while (test_bit(__LINK_STATE_RX_SCHED, &dev->state)) {
923         /* No hurry. */
924         msleep(1);
925     }
926
927     /*
928      *    Call the device specific close. This cannot fail.
929      *    Only if device is UP
930      *
931      *    We allow it to be called even after a DETACH hot-plug
932      *    event.
933      */
934     if (dev->stop)
935         dev->stop(dev);
```

```
936
937     /*
938      *    Device is now down.
939      */
940
941     dev->flags &= ~IFF_UP;
942
943     /*
944      * Tell people we are down
945      */
946     raw_notifier_call_chain(&netdev_chain, NETDEV_DOWN, dev);
947
948     return 0;
949 }
```

902-903 若网络设备未启用，则无需再进行操作。

909 在关闭网络设备之前，发送 NETDEV_GOING_DOWN 消息到网络设备状态改变通知链，以便通知对设备禁止有兴趣的内核组件。

911-913 调用 dev_deactivate()禁止出口队列规则，确保该设备不再用于传输，并停止不再需要的监控定时器，将网络设备设置为禁止传递数据包状态，设置对应标志。

921-925 在关闭时，如果网络设备正在轮询接收数据包，则需等待，直至此次接收数据包完成才能继续后续的关闭操作。

934-935 如果实现了 stop 函数，则根据具体硬件执行与 open 相反的操作。

941 成功关闭网络设备后去掉其已启用标志。

946完成关闭设备后，发送 NETDEV_DOWN 消息到网络设备状态改变通知链上，通知对设备禁止感兴趣的内核组件。

5.7 与电源管理交互

若内核支持电源管理，则在系统进入挂起模式或被唤醒时，网络设备驱动程序会获得相应的通知。pci_driver 结构的 suspend()和 resume()指针根据内核是否支持电源管理来初始化。下面显示了 e100 网络设备驱动如何初始化 pci_driver 实例。

```
2860 static struct pci_driver e100_driver = {
2861     .name =         DRV_NAME,
2862     .id_table =     e100_id_table,
2863     .probe =        e100_probe,
2864     .remove =       __devexit_p(e100_remove),
2865 #ifdef CONFIG_PM
2866     /* Power Management hooks */
2867     .suspend =      e100_suspend,
2868     .resume =       e100_resume,
2869 #endif
2870     .shutdown =     e100_shutdown,
2871     .err_handler = &e100_err_handler,
2872 };
```

当系统进入挂起模式时，调用设备驱动提供的 suspend()，以使设备驱动执行挂起操作。电源管理状态改变并不影响设备的注册状态 dev->reg_state，但设备状态 dev->state 会改变。

5.7.1 挂起设备

当设备挂起时，设备驱动会响应并处理该事件。例如，基于 PCI 的网络设备，会调用 PCI 设备 pci_driver 的 suspend()。e100 网络设备驱动的 suspend 函数为 e100_suspend()，除了提供驱动程序规范规定的功能外，还需完成其他一些额外的功能。

```
2716 static int e100_suspend(struct pci_dev *pdev, pm_message_t state)
2717 {
2718     struct net_device *netdev = pci_get_drvdata(pdev);
2719     struct nic *nic = netdev_priv(netdev);
2720
2721     if (netif_running(netdev))
2722         netif_poll_disable(nic->netdev);
2723     del_timer_sync(&nic->watchdog);
2724     netif_carrier_off(nic->netdev);
2725     netif_device_detach(netdev);
2726
2727     pci_save_state(pdev);
2728
2729     if ((nic->flags & wol_magic) | e100_asf(nic)) {
2730         pci_enable_wake(pdev, PCI_D3hot, 1);
2731         pci_enable_wake(pdev, PCI_D3cold, 1);
2732     } else {
2733         pci_enable_wake(pdev, PCI_D3hot, 0);
2734         pci_enable_wake(pdev, PCI_D3cold, 0);
2735     }
2736
2737     pci_disable_device(pdev);
2738     free_irq(pdev->irq, netdev);
2739     pci_set_power_state(pdev, PCI_D3hot);
2740
2741     return 0;
2742 }
```

2721-2722 如果网络设备处于激活状态，则等待网络设备完成轮询接收数据包。

2723 删除监视网络设备工作状态的定时器。

2724-2725 显式调用 netif_carrier_off()，使设备驱动处于不可传递数据状态，并关闭网络设备的排队功能。

2727-2739 特定网络设备驱动的操作。

5.7.2 唤醒设备

当设备被唤醒时，其设备驱动会响应并处理该事件。例如，调用 PCI 设备 pci_driver 的 resume()。设置 dev->state 的__LINK_STATE_PRESENT 标志位，表示设备现在可用了。如果设备在挂起前被使能，则用 netif_wake_queue()重新使能其入口队列，并由流量控制重新开启其监视定时器。以 e100 网络设备驱动的 resume 函数 e100_resume()为例。

```
2744 static int e100_resume(struct pci_dev *pdev)
2745 {
2746     struct net_device *netdev = pci_get_drvdata(pdev);
```

```
2747    struct nic *nic = netdev_priv(netdev);
2748
2749    pci_set_power_state(pdev, PCI_D0);
2750    pci_restore_state(pdev);
2751    /* ack any pending wake events, disable PME */
2752    pci_enable_wake(pdev, 0, 0);
2753
2754    netif_device_attach(netdev);
2755    if (netif_running(netdev))
2756       e100_up(nic);
2757
2758    return 0;
2759 }
```

5.8 侦测连接状态改变

通常，网络设备会定时地侦测网络设备是否处于可传递状态。当状态发生变化时，会调用netif_carrier_on()或 netif_carrier_off()来通知内核。从网络设备插拔网线或网线另一端的设备（hub、网桥、路由器）关闭或禁止，都会导致连接状态改变。

当设备驱动侦测到设备传递信号时，会调用 netif_carrier_on()。

```
251 void netif_carrier_on(struct net_device *dev)
252 {
253    if (test_and_clear_bit(__LINK_STATE_NOCARRIER, &dev->state))
254       linkwatch_fire_event(dev);
255    if (netif_running(dev))
256       __netdev_watchdog_up(dev);
257 }
```

253-254 清除标识设备状态的__LINK_STATE_NOCARRIER 标志位。如果此前已处于不可传递数据状态，则生成一个连接状态改变事件并提交给 linkwatch_fire_event()处理。

255-256 如果设备被使能，则启动一个监视定时器，该定时器由流量控制使用，以侦测是否发送失败或获得一个时间戳（这种情况定时器超时）。

当设备驱动侦测到设备丢失信号时，会调用 netif_carrier_off()。

```
259 void netif_carrier_off(struct net_device *dev)
260 {
261    if (!test_and_set_bit(__LINK_STATE_NOCARRIER, &dev->state))
262       linkwatch_fire_event(dev);
263 }
```

设置标识设备状态的__LINK_STATE_NOCARRIER 标志位。如果状态发生了变化，则生成一个连接状态改变事件并提交给 linkwatch_fire_event()处理。

5.8.1 调度处理连接状态改变事件

连接状态改变事件由 lw_event 结构来定义，该结构相当简单：仅包含一个关联到 net_device 结构的指针，此外，一个未处理事件以链表方式连接到 lweventlist，等待 linkwatch_event 工作

队列对其进行处理。

```
43 struct lw_event {
44     struct list_head list;
45     struct net_device *dev;
46 };
```

lw_event 结构并不包含任何区分信号传递的检测与丢失的参数。这是因为没有区别的必要，内核只需知道连接状态发生了变化，有对设备的引用就足够了。对任何设备，在 lweventlist 链表中，绝对不存在多个 lw_event 实例，因为没有必要去记录变化的历史轨迹：要么连接可操作，要么不可以，两次状态改变等于没有变化，三次变化等价于一次改变。因此，当设备已有正在处理的连接状态改变事件时，新事件不进入队列。

相关函数如下：

（1）linkwatch_fire_event()

linkwatch_fire_event()用于将新事件加入队列，并调度 linkwatch_work 工作队列进行处理，流程图如图 5-9 所示。

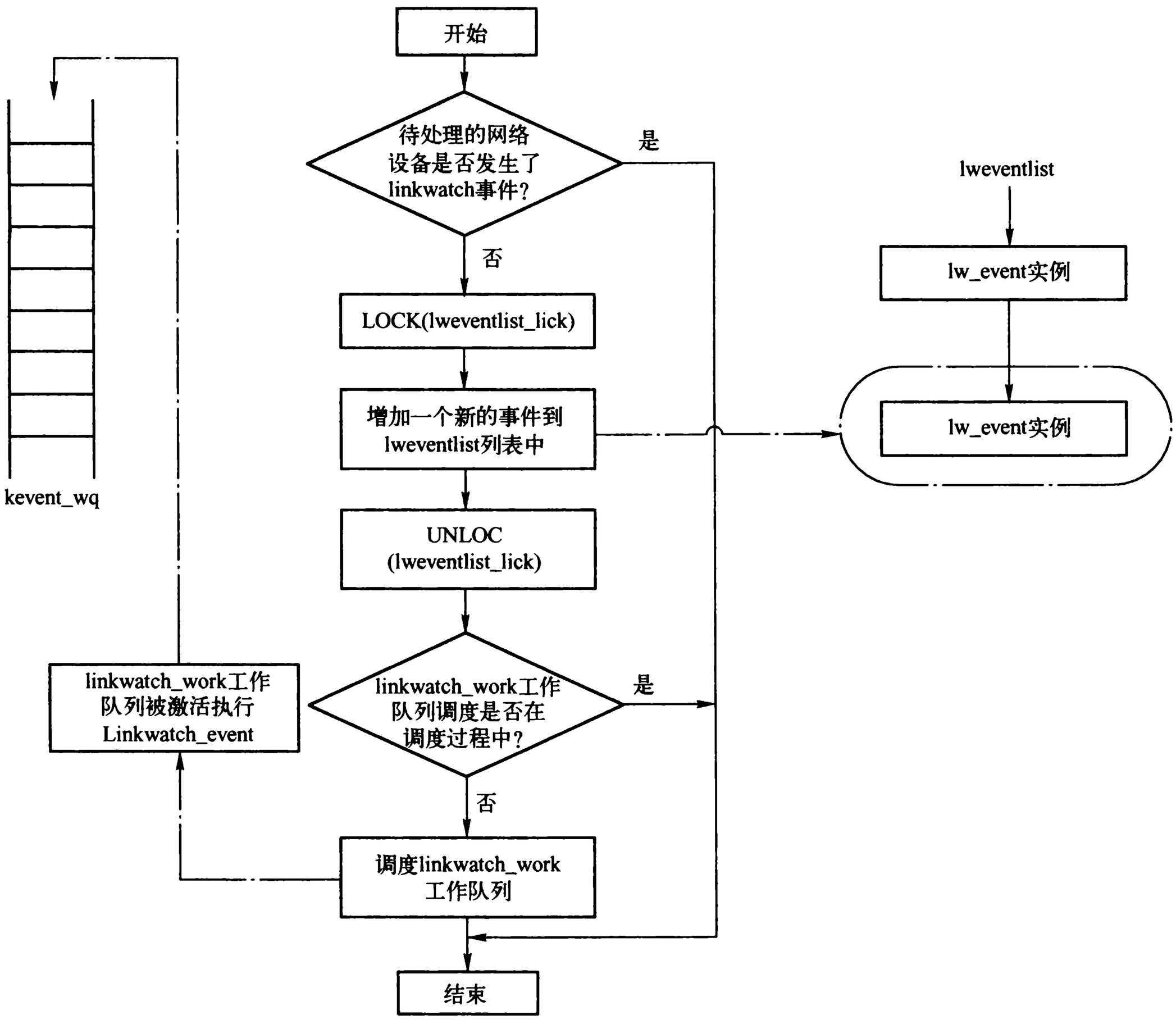

图 5-9　linkwatch_fire_event()流程

一旦确认网络设备的连接状态发生了变化，便会设置 dev->state 的__LINK_STATE_LINKWATCH_PENDING 标志位，然后将连接状态发生变化的 lw_event 实例添加到 lweventlist

链表上，调度 linkwatch_work 工作队列处理未处理的事件，为了避免 linkwatch_work 工作队列处理太频繁，限制最多每秒一次。

```
146 void linkwatch_fire_event(struct net_device *dev)
147 {
148     if (!test_and_set_bit(__LINK_STATE_LINKWATCH_PENDING, &dev->state)) {
149         unsigned long flags;
150         struct lw_event *event;
151
152         if (test_and_set_bit(LW_SE_USED, &linkwatch_flags)) {
153             event = kmalloc(sizeof(struct lw_event), GFP_ATOMIC);
154
155             if (unlikely(event == NULL)) {
156                 clear_bit(__LINK_STATE_LINKWATCH_PENDING, &dev->state);
157                 return;
158             }
159         } else {
160             event = &singleevent;
161         }
162
163         dev_hold(dev);
164         event->dev = dev;
165
166         spin_lock_irqsave(&lweventlist_lock, flags);
167         list_add_tail(&event->list, &lweventlist);
168         spin_unlock_irqrestore(&lweventlist_lock, flags);
169
170         if (!test_and_set_bit(LW_RUNNING, &linkwatch_flags)) {
171             unsigned long delay = linkwatch_nextevent - jiffies;
172
173             /* If we wrap around we'll delay it by at most HZ. */
174             if (delay > HZ)
175                 delay = 0;
176             schedule_delayed_work(&linkwatch_work, delay);
177         }
178     }
179 }
```

148 网络设备的连接状态发生改变，便会设置 dev->state 的__LINK_STATE_LINKWATCH_PENDING 标志位，表示正在处理连接状态改变事件。

152-161 如果静态分配的 lw_event 实例 singleevent 没被使用，则优先使用之（singleevent 被使用后，会设置 linkwatch_flags 的 LW_SE_USED 标志），当有多个其他事件时才分配额外的 lw_event 实例。

163-168 在设置与事件相关的网络设备后，将其添加到 lweventlist 队列上。由于该网络设备有连接状态改变事件等待处理，因此需增加对该网络设备的引用，以免被提前释放。

170-177 添加到队列后，如果此时 linkwatch_work 工作队列没有被调度，则调度它。为避免该工作队列处理太频繁，限制最多每秒执行一次。

（2）linkwatch_event()

linkwatch_event()为 linkwatch_work 工作队列的例程，一旦 linkwatch_work 工作队列被激活，就会调用该函数。首先清除 linkwatch_flags 的 LW_RUNNING 标志位，然后由 linkwatch_run_queue()处理网络设备连接状态改变事件。

```
130 static void linkwatch_event(struct work_struct *dummy)
131 {
132     /* Limit the number of linkwatch events to one
133      * per second so that a runaway driver does not
134      * cause a storm of messages on the netlink
135      * socket
136      */
137     linkwatch_nextevent = jiffies + HZ;
138     clear_bit(LW_RUNNING, &linkwatch_flags);
139
140     rtnl_lock();
141     linkwatch_run_queue();
142     rtnl_unlock();
143 }
```

（3）linkwatch_run_queue()

linkwatch_run_queue()用于取出并遍历所有的连接状态改变事件，在网络设备启用状态下，根据当前的网络连接状态打开或关闭网络设备的排队功能，并且发送相应的通知消息。

```
91  void linkwatch_run_queue(void)
92  {
93      struct list_head head, *n, *next;
94
95      spin_lock_irq(&lweventlist_lock);
96      list_replace_init(&lweventlist, &head);
97      spin_unlock_irq(&lweventlist_lock);
98
99      list_for_each_safe(n, next, &head) {
100         struct lw_event *event = list_entry(n, struct lw_event, list);
101         struct net_device *dev = event->dev;
102
103         if (event == &singleevent) {
104             clear_bit(LW_SE_USED, &linkwatch_flags);
105         } else {
106             kfree(event);
107         }
108
109         /* We are about to handle this device,
110          * so new events can be accepted
111          */
112         clear_bit(__LINK_STATE_LINKWATCH_PENDING, &dev->state);
113
114         rfc2863_policy(dev);
115         if (dev->flags & IFF_UP) {
116             if (netif_carrier_ok(dev)) {
117                 WARN_ON(dev->qdisc_sleeping == &noop_qdisc);
118                 dev_activate(dev);
119             } else
120                 dev_deactivate(dev);
121
122             netdev_state_change(dev);
123         }
124
125         dev_put(dev);
```

```
126     }
127 }
```

95-97 取出 lweventlist 链表上的未处理事件，取出后 lweventlist 链表为空，此时又可再添加事件。

99 遍历处理链表上所有的连接状态改变事件。

100-107 获取连接状态改变事件的网络设备后，释放存储事件的 lw_event 实例。如果是 singleevent，则清除 linkwatch_flags 的 LW_SE_USED 标志位。

112 清除 dev->state 的__LINK_STATE_LINKWATCH_PENDING 标志位，表示网络设备又可以接受连接状态改变事件了。

115-123 网络设备在启用状态下，根据当前的网络连接状态打开/关闭网络设备的排队功能，并发送 NETDEV_CHANGE 通知给 netdev_chain 通知链，发送 RTM_NEWLINK 通知给 RTMGRP_LINK RTnetlink 组。

125 完成对连接状态改变事件的处理后，释放对该网络设备的引用。

5.8.2 linkwatch 标志

net/core/linkwatch.c 中定义了两个用于设置全局变量 linkwatch_flags 的标志，见表 5-7。

表 5-7 linkwatch 的取值

linkwatch_flags	描述
LW_RUNNING	当 linkwatch_work 工作队列调度时，该标志位被设置。当 linkwatch_work 工作队列激活时，该标志位被清除
LW_SE_USED	由于存储事件的 lweventlist 链表上通常不止有一个元素，为了提高性能，静态分配的 lw_event 数据结构总是作为第一个元素，当有多个其他的事件时才分配额外的 lw_event 实例

5.9 从用户空间配置设备相关信息

系统提供了多种工具来配置和获取网络设备的媒介状态和硬件参数。这些工具包括 ifconfig，mii-tool，ethtool 和 ip。通过 man 手册可获取这些命令的语法类型细节。

5.9.1 ethtool

图 5-10 显示了通过 ethtool 和 mii-tool 用户工具配置网络设备，而引发各主要函数间的调用关系，最终调用 dev->ethtool_ops 或 dev->ioctl 来实现。dev->ethtool_ops 是提供给 ethtool 的操作接口，并非所有的设备驱动程序都支持这个特性，或者即使支持也并不支持所有的函数，该接口在驱动初始化过程中设置。

目前网络设备的配置只支持 ioctl 系统调用接口，因此用户空间与内核函数的接口就是 ioctl 系统调用。图 5-10 展示了用户空间命令 ethtool 如何结束调用内核空间中的 dev_ethtool()，以及该函数的流程，函数接口如何和媒体独立接口（Media Independent Interface，MII）内核交互。

dev_ethtool()首先进行有效性检查，然后基于 ifreq 数据结构提供的来自用户空间的命令类型，调用相应的处理函数 ethtool_*xxx*()，而这些函数封装了 dev->ethtool_ops->*xxx* 函数指针。因为即便是支持 ethtool 的驱动程序也无需支持所有的 ethtool_ops 函数，因此如果是不被支持的函数就返回 EOPNOTSUPP，操作不支持错误。

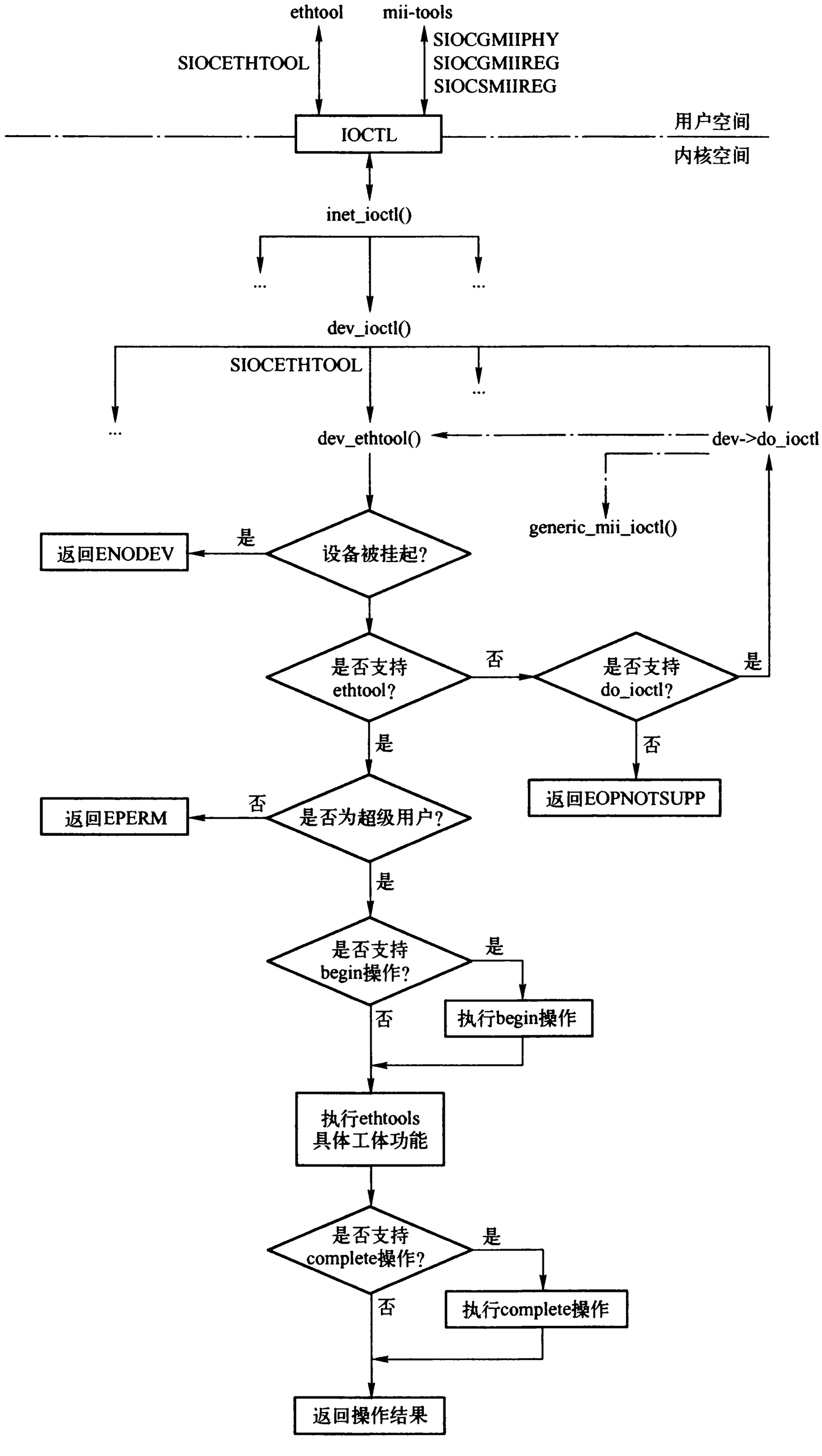

图 5-10　设备配置 ioctl 接口

对不支持 ethtool 的驱动程序，当 dev_ethtool()被调用去处理一个命令时，会试着调用 dev->do_ioctl()来处理该命令。有可能驱动程序也不支持 dev->do_ioctl，在这种情况下，返回 -EOPNOTSUPP。

do_ioctl()也有可能回调 dev_ethtool()。例如虚拟设备，会简单地让与实际设备关联的设备驱动来处理这些命令(参见 net/8021q/vlan_dev.c 中的 vlan_dev_ioctl())。

5.9.2 媒体独立接口

MII（Media Independent Interface）是一个 IEEE 标准规范，它描述了网络控制芯片和物理介质芯片之间的接口。用户可以使用这些接口来使能、禁止、协商网络设备。并非所有的网络设备都提供该接口。

Linux 下用于和 MII 交互的最常见的工具是 mii-tools。如图 5-10 所示，ethtool 和内核交互也是通过 ioctl，内核提供了一组 ioctl 命令来处理 MII，这些命令主要由基于特定网络设备注册器的读写函数组成。

ioctl 命令被设备驱动传递到 dev->do_ioctl()，该函数能够以以下两种方式来处理命令：

- 仅识别是三个 MII ioctl 命令中的哪一个，并调用设备驱动代码进行处理。这是最常见的情况。
- 依赖内核 MII 库驱动程序(drivers/net/mii.c)，调用 generic_mii_ioctl()处理输入命令，尤其是虚拟设备，有可能会用 dev->do_ioctl()识别和处理非 MII 命令。

5.10 虚拟网络设备

虚拟网络设备是建立在一个或多个真实设备之上的抽象设备。虚拟设备与真实设备的对应关系可以是一对一，也可以是多对一或者一对多，如图 5-11 所示。并且在虚拟设备之上还可以创建新的虚拟设备。

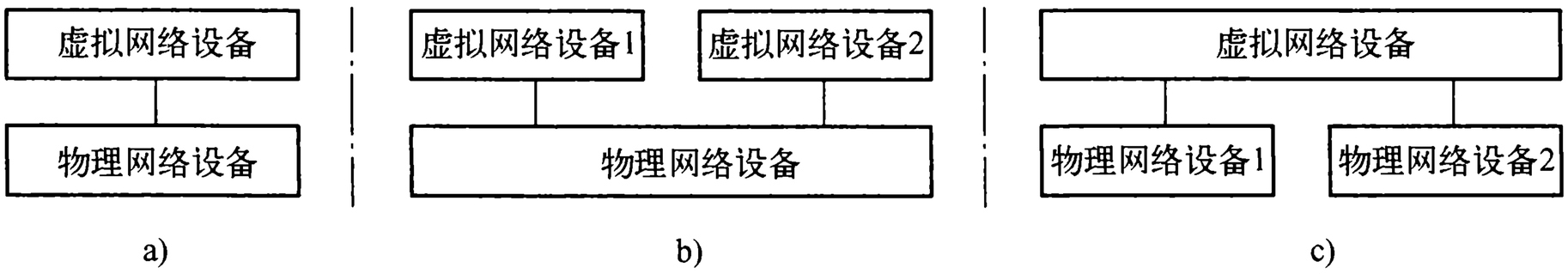

图 5-11 虚拟设备与物理设备的关系

a) 一对 b) 多对 c) 一对多

Linux 允许创建各种类型的虚拟设备，表 5-8 中列出了几种虚拟网络设备类型，但并不完整。

表 5-8 虚拟网络设备类型

虚拟网络设备类型	描述
Bonding	这种类型的虚拟设备可以捆绑多个物理设备并把它们当作一个设备使用，通过不同的算法将流量分散到这些物理设备。通常，Bonding 接口会监视它所绑定的物理设备，关闭某个物理设备后 Bonding 接口能继续处理数据，当绑定的所有的网络设备关闭后，Bonding 接口才会不得不被禁止，因为此时没有任何工作着的物理设备。虚拟设备与网络设备的对应关系如图 5-11c 所示
802.1Q	这种类型是 IEEE 对 802.3 协议首部的扩展标准，支持对 VLAN 首部进行处理。通过 802.1Q 协议，可以定义虚拟局域网。多个虚拟 VLAN 网络设备可以绑定同一个物理设备。因为它们依赖同一个物理设备，因此当被绑定的设备关闭后，所有与之绑定的 VLAN 设备都会被关闭。虚拟设备与网络设备的对应关系如图 5-11b 所示
Bridging	桥虚拟设备。虚拟设备与网络设备的对应关系如图 5-11c 所示
Aliasing interfaces	最初，这个功能的目的是允许在一个物理网络设备上创建多个虚拟设备（比如 eth0:0，eth0:1 等），每个虚拟设备都可以配置自己的 IP 地址等属性。目前，由于网络代码的改进，已经不需要创建多个虚拟设备就可以在一个设备上配置多个 IP 地址了。然而在某些情况下（特别是配置路由时），为一个接口创建多个虚拟接口可以使配置更简单。别名虚拟设备与网络设备的对应关系如图 5-11b 所示
Tunnel interfaces	实现 IP-over-IP 隧道（IPIP）和通用路由封装(GRE)协议时需要创建一种虚拟设备。虚拟设备与网络设备的对应关系如图 5-11a 所示

虚拟设备必定与物理设备或其他虚拟设备绑定，因此它与内核交互的方式与物理网络设备必定存在一些不同与关联，见表 5-9。

表 5-9 虚拟设备与物理设备在与内核交互的方式上的区别

比较处	描述
初始化	大多数虚拟设备与物理设备类似，都有一个 net_device 结构的实例。通常，大多数虚拟设备的函数指针都被初始化为一个封装函数，这些封装函数最终通常都会调用真实设备中的相关函数
配置	对于系统提供设备，通常都会提供与之相应、特定的配置工具来配置虚拟设备，因为虚拟设备的某些参数是无法通过 ifconfig 之类的工具进行配置的
外部接口	通常，每个虚拟设备都会在/proc 文件系统中导出一个文件或目录。导出信息的复杂度和详细程度与设备相关。该文件或目录是附加的，不会替换绑定物理设备的文件
发送	如果虚拟设备和物理设备的对应关系不是一对一，虚拟设备的发送过程就需要包含选择哪个真实设备发送包的算法。由于 QoS 是配置在单个设备上的，因此，如果虚拟设备和真实设备的对应关系是多对多，则会对流量控制功能产生影响
接收	由于虚拟设备不需要与系统真实资源的交互（例如，注册 IRQ 处理程序或者分配 I/O 端口和 I/O 内存），因此它们接收的包来自物理设备，而由物理设备完成与系统资源的交互 不同类型的虚拟设备，它们接收包的过程也不尽相同。例如，802.1Q 设备需要先注册一个以太网包类型 ID，只有真实设备上收到的与此 ID 相同的包才传递给虚拟设备。与之相反，桥设备可以接收相关设备上收到的所有包
外部通知	内核中其他模块的特定的事件对待虚拟设备的处理是和物理设备一样的，是直接发送给虚拟设备的。比如 Bonding 设备，如果与之绑定的多个物理设备中的一个设备失效，在设备组中分配流量的算法就必须知道这种情况，从而避免把包发送给已经失效的物理设备 而对于硬件触发的通知是不能到达虚拟设备的，因为虚拟设备和硬件之间没有直接的联系（例如：PCI 电源管理）
注册	虚拟设备有时会调用 register_netdevice()和 unregister_netdevice()而不是它们的封装函数来进行注册或注销，并且由自己处理锁问题。它们处理锁问题时需要保持锁的时间比物理设备长一些。这种情况下，锁可能被误用且需要持有更长的时间，以保护可以用其他方式保护的额外的代码段
注销	物理设备不能被用户命令注销或销毁，它们仅能被禁止，并只在设备驱动程序卸载时被注销。与此相对，虚拟设备能够被用户命令创建或注销

第6章　IP编址

尽管IP是属于主机的，但在IP模型中，IP地址是设置到主机的网络接口而不是系统本身。因特网上的系统通常可划分为两类：主机和路由器。主机通常只有一个网络接口，是IP数据报的发送方或接收方。而路由器通常有多个网络接口，因而会有多个IP地址。当IP数据报向其目标方移动时，路由器将IP数据报从一个网络转发到下一个网络。为执行这个功能，路由器用各种专用路由协议来交换关于网络拓扑的信息。

本章讨论这些实用的设置、维护IP地址的函数，这些函数涉及以下文件：

- include/linux/inetdevice.h，定义IPv4专用的网络设备相关的结构、宏等。
- net/ipv4/devinet.c，支持IPv4特性的设备操作接口。

6.1　接口和IP地址

6.1.1　主IP地址、从属IP地址和IP别名

主IP地址是同一寻址范围中第一个配置到网络设备上的地址，而配置多个IP地址的网络设备可能有多种寻址范围，因此一个网络设备可以配置多个不同寻址范围的主IP地址。

而通过从属IP地址和IP别名，可以为一个网络设备添加多个同一寻址范围的IP地址。

6.1.2　IP地址的组织

net_device结构中包含了驱动相关的所有信息，先按分类把同一类型的信息组织到其他结构中，然后嵌套到net_device结构中。例如，与IPv4相关的配置存放在in_device结构中，IP地址、子网掩码、广播地址等信息存放到in_ifaddr结构中等，这些结构之间的关系如图6-1所示。

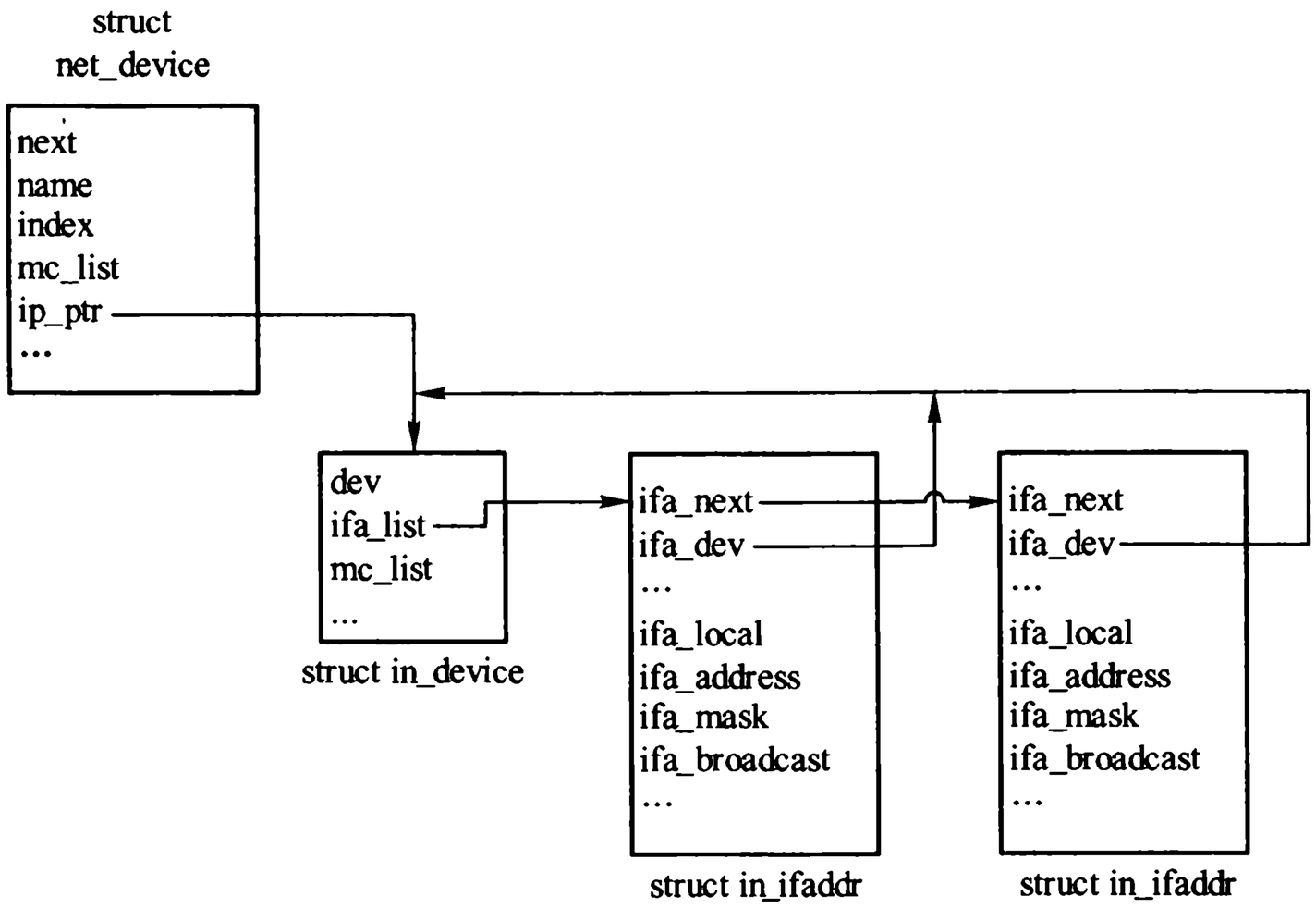

图6-1　net_device结构及地址信息

6.1.3 in_device 结构

IP 配置块，网络设备层与 IPv4 相关的配置都存放在 in_device 结构中，应用层可以通过 ip 或 ifconfig 工具来修改这些配置。该结构实例的地址保存在 net_device 结构成员 ip_ptr 中，可通过 in_dev_get()访问它。访问结束后，必须调用 in_dev_put()，因为 in_dev_get()成功获取 in_device 实例后会递增其引用计数，以防止在访问期间实例被释放，而 in_dev_put()则会递减实例的引用计数，当引用计数为 0 时，才真正释放实例。为设备设置第一个 IP 地址时，在 inetdev_init()中分配并初始化 in_device 结构实例。

```
39 struct in_device
40 {
41     struct net_device   *dev;
42     atomic_t        refcnt;
43     int             dead;
44     struct in_ifaddr    *ifa_list;    /* IP ifaddr chain          */
45     rwlock_t        mc_list_lock;
46     struct ip_mc_list   *mc_list;    /*IP multicast filter chain*/
47     spinlock_t      mc_tomb_lock;
48     struct ip_mc_list   *mc_tomb;
49     unsigned long       mr_v1_seen;
50     unsigned long       mr_v2_seen;
51     unsigned long       mr_maxdelay;
52     unsigned char       mr_qrv;
53     unsigned char       mr_gq_running;
54     unsigned char       mr_ifc_count;
55     struct timer_list    mr_gq_timer;    /* general query timer */
56     struct timer_list    mr_ifc_timer;  /*interface change timer*/
57
58     struct neigh_parms    *arp_parms;
59     struct ipv4_devconf    cnf;
60     struct rcu_head       rcu_head;
61 };
```

41 struct net_device *dev

指向所属的网络设备。

42 atomic_t refcnt

引用计数器。

43 int dead

为 1 时标识所在的 IP 配置块将要被释放，不再允许访问其成员。

44 struct in_ifaddr *ifa_list

指向 in_ifaddr 结构链表。in_ifaddr 结构中存储了网络设备的 IP 地址，因为一个网络设备可配置多个 IP 地址，因此需要用链表来存储，in_ifaddr 结构参见 6.1.4 节。

45 rwlock_t mc_list_lock

46 struct ip_mc_list *mc_list

47 spinlock_t mc_tomb_lock

48 struct ip_mc_list *mc_tomb

49 unsigned long mr_v1_seen

50 unsigned long mr_v2_seen
51 unsigned long mr_maxdelay
52 unsigned char mr_qrv
53 unsigned char mr_gq_running
54 unsigned char mr_ifc_count
55 struct timer_list mr_gq_timer
56 struct timer_list mr_ifc_timer

以上多个成员与组播的设置有关，本章不作论述，参见第 15 章和第 16 章。

58 struct neigh_parms *arp_parms

指向 neigh_parms 结构实例，存储一些与 ARP 有关的参数，参见第 17 章和第 18 章。

59 struct ipv4_devconf cnf

一些针对网络设备接口的 IPv4 设置，参见 11.6 节。

60 struct rcu_head rcu_head

运用 RCU 机制释放所在的 IP 配置块。

6.1.4　in_ifaddr 结构

IP 地址块，用于存储主机的 IP 地址、子网掩码、广播地址，这些配置属于主机，但又是配置到网络设备上的，因此与网络设备也相关。一个网络设备配置多少个 IP 地址，就有多少个 IP 地址块。

```
88  struct in_ifaddr
89  {
90      struct in_ifaddr    *ifa_next;
91      struct in_device    *ifa_dev;
92      struct rcu_head       rcu_head;
93      __be32          ifa_local;
94      __be32          ifa_address;
95      __be32          ifa_mask;
96      __be32          ifa_broadcast;
97      __be32          ifa_anycast;
98      unsigned char       ifa_scope;
99      unsigned char       ifa_flags;
100     unsigned char       ifa_prefixlen;
101     char          ifa_label[IFNAMSIZ];
102 };
```

90 struct in_ifaddr *ifa_next

一个网络设备的多个 IP 地址块用 ifa_next 构成链表。

91 struct in_device *ifa_dev

指向所属的 IP 配置块。

92 struct rcu_head rcu_head

主要运用 RCU 机制来释放对应的 IP 地址块。

93 __be32 ifa_local
94 __be32 ifa_address

这两个成员都是用来存储 IP 地址的，但在不同设备接口特性下意义不同：

- 在配置了支持广播的设备上，ifa_local 和 ifa_address 是一样的，都是本地的 IP 地址。
- 如果是点对点，则 ifa_address 存储点对点对端的 IP 地址，而 ifa_local 存储本地的 IP 地址。

95 __be32 ifa_mask

IP 地址的子网掩码。

96 __be32 ifa_broadcast

网络设备的广播地址。

97 __be32 ifa_anycast

目前未使用。

98 unsigned char ifa_scope

寻址范围，见表 6-1。

表 6-1 ifa_scope 的取值

ifa_scope	描述
RT_SCOPE_UNIVERSE	此地址可以在任何地方使用，这是大多数的地址默认值
RT_SCOPE_SITE	此地址在一个本地封闭系统中的内部路由
RT_SCOPE_LINK	此地址只在一个局域网内有意义，比如广播地址
RT_SCOPE_HOST	此地址只用于主机内部通信，比如回环地址
RT_SCOPE_NOWHERE	此地址的目的地址不存在

99 unsigned char ifa_flags

IP 地址属性，见表 6-2。

表 6-2 ifa_flags 的取值

ifa_flags	描述
IFA_F_SECONDARY	从属 IP 地址
IFA_F_NODAD	禁止重复地址检测，目前仅在 IPv6 中使用
IFA_F_HOMEADDRESS	移动 IPv6 的家乡地址，目前仅在 IPv6 中使用
IFA_F_DEPRECATED	此 IP 地址不鼓励使用，但也不禁止
IFA_F_TENTATIVE	尝试阶段的 IP 地址，仍在进行地址检测工作
IFA_F_PERMANENT	用户的永久性地址。如果没有此标识，则会动态地配置 IP 地址

100 unsigned char ifa_prefixlen

子网掩码长度。

101 char ifa_label[IFNAMSIZ]

地址标签，通常是网络设备名或网络设备别名。

6.2 函数

6.2.1 inetdev_init()

inetdev_init()为通过参数指定的网络设备分配并绑定 IP 配置块。

```
146 struct in_device *inetdev_init(struct net_device *dev)
147 {
148     struct in_device *in_dev;
149
150     ASSERT_RTNL();
151
152     in_dev = kzalloc(sizeof(*in_dev), GFP_KERNEL);
153     if (!in_dev)
154         goto out;
155     INIT_RCU_HEAD(&in_dev->rcu_head);
156     memcpy(&in_dev->cnf, &ipv4_devconf_dflt, sizeof(in_dev->cnf));
157     in_dev->cnf.sysctl = NULL;
158     in_dev->dev = dev;
159     if ((in_dev->arp_parms = neigh_parms_alloc(dev, &arp_tbl)) == NULL)
160         goto out_kfree;
161     /* Reference in_dev->dev */
162     dev_hold(dev);
163 #ifdef CONFIG_SYSCTL
164     neigh_sysctl_register(dev, in_dev->arp_parms, NET_IPV4,
165                 NET_IPV4_NEIGH, "ipv4", NULL, NULL);
166 #endif
167
168     /* Account for reference dev->ip_ptr (below) */
169     in_dev_hold(in_dev);
170
171 #ifdef CONFIG_SYSCTL
172     devinet_sysctl_register(in_dev, &in_dev->cnf);
173 #endif
174     ip_mc_init_dev(in_dev);
175     if (dev->flags & IFF_UP)
176         ip_mc_up(in_dev);
177
178     /* we can receive as soon as ip_ptr is set -- do this last */
179     rcu_assign_pointer(dev->ip_ptr, in_dev);
180 out:
181     return in_dev;
182 out_kfree:
183     kfree(in_dev);
184     in_dev = NULL;
185     goto out;
186 }
```

152-154 分配一个 IP 配置块。

156-158 初始化 IP 配置块中的一些成员，包括 rcu 链表、IPv4 配置的默认值，以及所属网络设备。

159-160 为 IP 配置块分配邻居协议参数配置块，并根据 ARP 表初始化。

163-166 为邻居子系统注册相关的系统参数。

171-173 为 IP 注册相关的系统参数。

174 初始化 IGMP 模块。

175-176 如果网络设备已启用，则初始化该网络设备上的组播信息，例如，将该网络设备加入到 224.0.0.1 组播组等操作。

180-182 操作成功，返回分配并绑定成功的 IP 配置块，否则返回 NULL。

6.2.2 inetdev_destroy()

inetdev_destroy()通常在设备注销时被调用，释放指定的 IP 配置块。

```
194 static void inetdev_destroy(struct in_device *in_dev)
195 {
196     struct in_ifaddr *ifa;
197     struct net_device *dev;
198
199     ASSERT_RTNL();
200
201     dev = in_dev->dev;
202     if (dev == &loopback_dev)
203         return;
204
205     in_dev->dead = 1;
206
207     ip_mc_destroy_dev(in_dev);
208
209     while ((ifa = in_dev->ifa_list) != NULL) {
210         inet_del_ifa(in_dev, &in_dev->ifa_list, 0);
211         inet_free_ifa(ifa);
212     }
213
214 #ifdef CONFIG_SYSCTL
215     devinet_sysctl_unregister(&in_dev->cnf);
216 #endif
217
218     dev->ip_ptr = NULL;
219
220 #ifdef CONFIG_SYSCTL
221     neigh_sysctl_unregister(in_dev->arp_parms);
222 #endif
223     neigh_parms_release(&arp_tbl, in_dev->arp_parms);
224     arp_ifdown(dev);
225
226     call_rcu(&in_dev->rcu_head, in_dev_rcu_put);
227 }
```

201-203 不释放回环设备的 IP 配置块。

205 标识待释放的 IP 配置块正处在释放过程中。

207 销毁组播相关的配置，如停止相关定时器等。

209-212 删除并释放所有的 IP 地址块。

215 注销 IP 相关的系统参数。

218 将网络设备指向 IP 配置块的指针设置为 NULL。

221 注销邻居子系统相关的系统参数。

223 释放 IP 配置块中的邻居协议参数配置块。

224 通过 RCU 机制释放 IP 配置块。

6.2.3 inet_select_addr()

在通过输出网络设备向目的地址发送报文时，如果没有指定源地址，会调用 inet_select_addr()

来根据给定设备、目的地址和作用范围，获取给定作用范围内的主 IP 地址作为源地址。参数说明如下：

- dev，获取源地址的网络设备。
- dst，发送报文的目的地址，见表 6-3。

表 6-3　dst 的取值

dst	描述
不为 0	返回与目的地址在同一子网的 IP 地址（输出网络设备上配置的不同地址属于不同子网）
等于 0	返回本地地址

- scope，地址作用的范围，见表 6-4。

表 6-4　scope 的取值

scope	描述
RT_SCOPE_HOST	当报文被送往本地
RT_SCOPE_LINK	当报文被送给只在本地链路上有意义的地址，诸如广播、受限广播和本地组播
RT_SCOPE_UNIVERSE	当报文发送到通往远程非直连目的地

```
381 __be32 inet_select_addr(const struct net_device *dev, __be32 dst, int scope)
382 {
383     __be32 addr = 0;
384     struct in_device *in_dev;
385
386     rcu_read_lock();
387     in_dev = __in_dev_get_rcu(dev);
388     if (!in_dev)
389         goto no_in_dev;
390
391     for_primary_ifa(in_dev) {
392         if (ifa->ifa_scope > scope)
393             continue;
394         if (!dst || inet_ifa_match(dst, ifa)) {
395             addr = ifa->ifa_local;
396             break;
397         }
398         if (!addr)
399             addr = ifa->ifa_local;
400     } endfor_ifa(in_dev);
```

先检测该网络设备上 IPv4 配置块是否有效，通过检测后遍历 IPv4 配置块的本地 IP 地址列表，获取第一个满足条件（如 scope 和 dst）的本地地址。

```
401 no_in_dev:
402     rcu_read_unlock();
403
404     if (addr)
405         goto out;
406
407     /* Not loopback addresses on loopback should be preferred
408        in this case. It is importnat that lo is the first interface
```

```
909       in dev_base list.
910     */
911    read_lock(&dev_base_lock);
912    rcu_read_lock();
913    for (dev = dev_base; dev; dev = dev->next) {
914       if ((in_dev = __in_dev_get_rcu(dev)) == NULL)
915          continue;
916
917       for_primary_ifa(in_dev) {
918          if (ifa->ifa_scope != RT_SCOPE_LINK &&
919             ifa->ifa_scope <= scope) {
920             addr = ifa->ifa_local;
921             goto out_unlock_both;
922          }
923       } endfor_ifa(in_dev);
924    }
925 out_unlock_both:
926    read_unlock(&dev_base_lock);
927    rcu_read_unlock();
928 out:
929    return addr;
930 }
```

904-905 如果获得满足条件的地址，则将其返回。

913-924 如果给定配置的地址都不满足由 scope 和 dst 限定的条件，则尝试其他设备是否存在满足所要求的 scope 的一个 IP 地址。

6.2.4 inet_confirm_addr()

用来确认参数中指定的本地地址是否存在。参数说明如下：

- dev，用来确定是否存在指定本地地址的网络设备，如果为 NULL，则表示在所有的网络设备上确认本地地址。
- dst，目的 IP 地址，当其不为 0 时，则待确定的本地地址必须与该地址在同一子网内。
- local，待确认的本地地址，当其为 0 时，则自动选择一个本地地址。
- scope，确认本地地址时允许的最大范围。

```
976 __be32 inet_confirm_addr(const struct net_device *dev,__be32 dst,__be32 local,int scope)
977 {
978     __be32 addr = 0;
979     struct in_device *in_dev;
980
981     if (dev) {
982        rcu_read_lock();
983        if ((in_dev = __in_dev_get_rcu(dev)))
984           addr = confirm_addr_indev(in_dev, dst, local, scope);
985        rcu_read_unlock();
986
987        return addr;
988     }
989
990     read_lock(&dev_base_lock);
991     rcu_read_lock();
992     for (dev = dev_base; dev; dev = dev->next) {
```

```
993         if ((in_dev = __in_dev_get_rcu(dev))) {
994             addr = confirm_addr_indev(in_dev, dst, local, scope);
995             if (addr)
996                 break;
997         }
998     }
999     rcu_read_unlock();
1000    read_unlock(&dev_base_lock);
1001
1002    return addr;
1003 }
```

981-987 如果指定网络设备，则在该网络设备上确认本地 IP 地址。确认过程如下：获取网络设备的 IP 配置块，调用 confirm_addr_indev()在指定的 IP 配置块中查找与参数 local 给出的 IP 地址相同，与参数 dst 给出的 IP 地址在相同子网内，且范围小于 scope 的本地地址。

990-1002 当未指定网络设备时，则在所有的网络设备上确认本地 IP 地址。

6.2.5　inet_addr_onlink()

根据指定网络设备的 IP 配置块，检查两个给定的 IP 地址是否同属于一个子网。

```
229 int inet_addr_onlink(struct in_device *in_dev, __be32 a, __be32 b)
230 {
231     rcu_read_lock();
232     for_primary_ifa(in_dev) {
233         if (inet_ifa_match(a, ifa)) {
234             if (!b || inet_ifa_match(b, ifa)) {
235                 rcu_read_unlock();
236                 return 1;
237             }
238         }
239     } endfor_ifa(in_dev);
240     rcu_read_unlock();
241     return 0;
242 }
```

6.2.6　inetdev_by_index()

inetdev_by_index()根据网络设备索引号获取对应网络设备的 IP 配置块。

```
420 struct in_device *inetdev_by_index(int ifindex)
421 {
422     struct net_device *dev;
423     struct in_device *in_dev = NULL;
424     read_lock(&dev_base_lock);
425     dev = __dev_get_by_index(ifindex);
426     if (dev)
427         in_dev = in_dev_get(dev);
428     read_unlock(&dev_base_lock);
429     return in_dev;
430 }
```

425 根据索引获取对应的网络设备。

426-429 如果获得的网络设备有效，则返回其 IP 配置块，否则返回 NULL。

6.2.7 inet_ifa_byprefix()

inet_ifa_byprefix()在正在配置的输入设备的主 IP 地址中查找与前缀和掩码匹配的 IP 地址，如成功则返回匹配的地址。

```
434 struct in_ifaddr *inet_ifa_byprefix(struct in_device *in_dev, __be32 prefix,
435                                    __be32 mask)
436 {
437     ASSERT_RTNL();
438
439     for_primary_ifa(in_dev) {
440         if (ifa->ifa_mask == mask && inet_ifa_match(prefix, ifa))
441             return ifa;
442     } endfor_ifa(in_dev);
443     return NULL;
444 }
```

6.2.8 inet_abc_len()

inet_abc_len()根据指定的 IP 地址获取默认掩码长度。默认掩码长度见表 6-5。

表 6-5 默认掩码长度

地址	默认掩码长度
0 地址	0
A 类地址	8
B 类地址	16
C 类地址	24

```
582 static __inline__ int inet_abc_len(__be32 addr)
583 {
584     int rc = -1;    /* Something else, probably a multicast. */
585
586       if (ZERONET(addr))
587           rc = 0;
588     else {
589         __u32 haddr = ntohl(addr);
590
591         if (IN_CLASSA(haddr))
592             rc = 8;
593         else if (IN_CLASSB(haddr))
594             rc = 16;
595         else if (IN_CLASSC(haddr))
596             rc = 24;
597     }
598
599       return rc;
600 }
```

6.3　IP 地址的设置

6.3.1　netlink 接口

net-tools 包中的命令 ifconfig 是通过 ioctl 接口对网络设备进行相应的操作和配置的。而 Linux 提供的功能更强大的配置工具 IPROUTE2 包，则是通过 Linux 特有的 netlink 接口对 IP 地址进行操作的。

图 6-2 所示为通过 netlink 接口操作 IP 地址的主要函数。

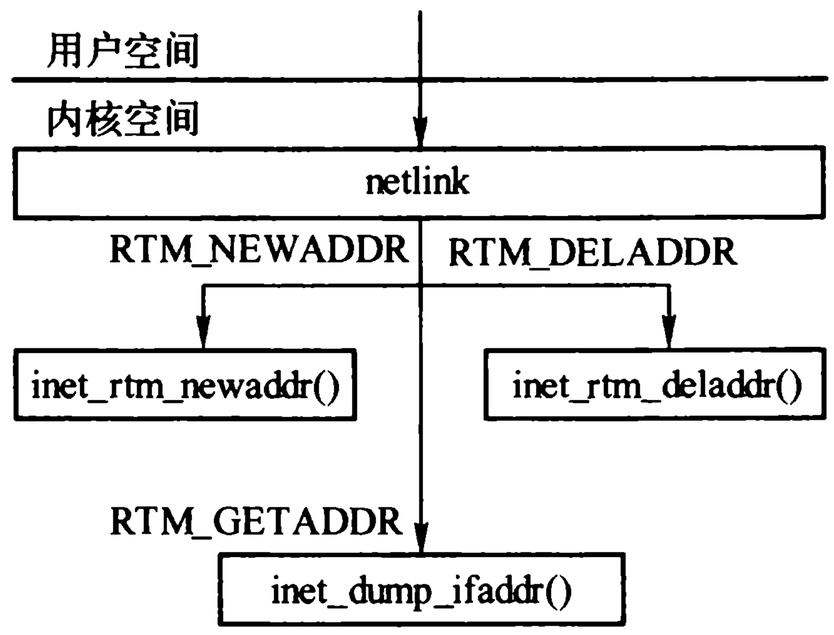

图 6-2　通过 netlink 接口操作 IP 地址的主要函数

1．netlink 消息结构

添加、删除和获取 IP 地址的 netlink 消息格式如图 6-3 所示。

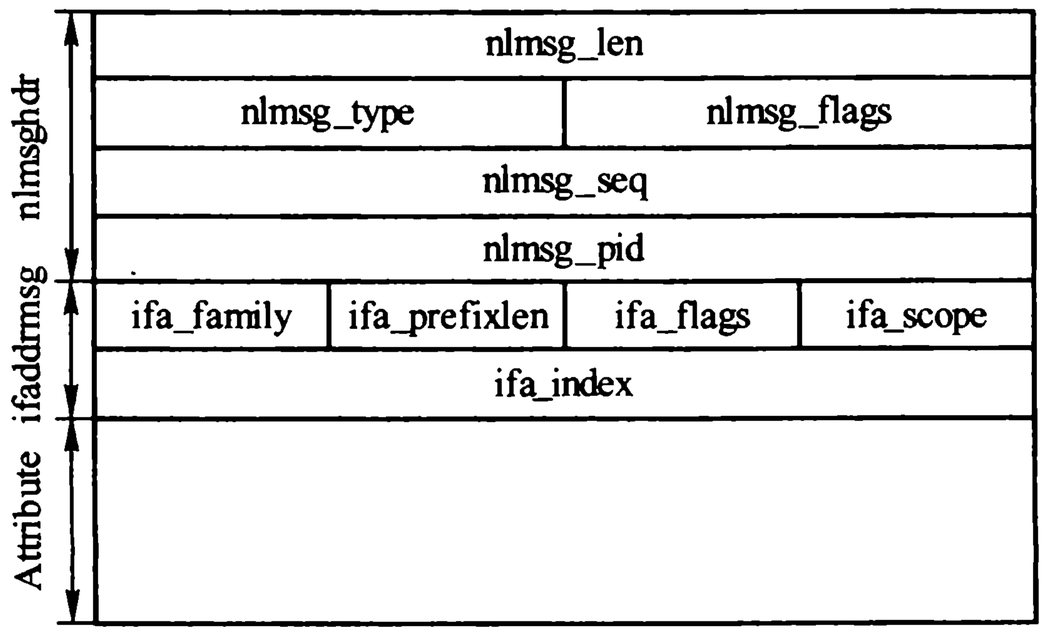

图 6-3　操作 IP 地址的 netlink 消息格式

nlmsfhdr 结构是 netlink 消息的首部，ifaddrmsg 结构是配置地址消息。这两个结构紧接在一起，当然 nlmsghdr 结构必须按 4 字节对齐，这在 nlmsg_data()中可以很清楚地看到。该函数的参数是一个 nlmsghdr 结构指针，返回的 void 指针实际上就是 ifaddrmsg 结构指针，函数中只是简单地将 nlmsghdr 结构地址加上其对齐后的长度 NLMSG_HDRLEN，由此通过 nlmsghdr 结构获得 ifaddrmsg 结构。ifaddrmsg 结构各字段的意义如下：

- ifa_family，8 位，标识操作的地址所属的地址族，如 AF_INET 和 AF_INET6 等。
- ifa_prefixlen，8 位，标识地址掩码长度。
- ifa_flags，8 位，地址标识，见表 6-2。

- ifa_scope，8 位，标识寻址范围，见表 6-1。
- ifa_index，32 位，标识操作的网络设备索引。
- Attribute，属性，标识配置的各种值，见表 6-6。

表 6-6 Attribute 的取值

属性	描述
IFA_ADDRESS	本地的 IP 地址或点对点对端的 IP 地址
IFA_LOCAL	网络设备的本地 IP 地址
IFA_LABEL	当设置 IP 时用的网络设备名称或别名标签
IFA_BROADCAST	网络设备的广播地址
IFA_ANYCAST	未使用
IFA_CACHEINFO	缓存地址信息，IPv4 中未使用

2. inet_rtm_newaddr()

当通过 netlink，操作类型为 RTM_NEWADDR 添加 IP 地址时，会调用此函数。

```
565 static int inet_rtm_newaddr(struct sk_buff *skb, struct nlmsghdr *nlh, void *arg)
566 {
567     struct in_ifaddr *ifa;
568
569     ASSERT_RTNL();
570
571     ifa = rtm_to_ifaddr(nlh);
572     if (IS_ERR(ifa))
573         return PTR_ERR(ifa);
574
575     return __inet_insert_ifa(ifa, nlh, NETLINK_CB(skb).pid);
576 }
```

571-573 从配置 IP 地址的消息中获取地址信息。

575 将 IP 地址配置到指定的网络设备上。

3. inet_rtm_deladdr()

当通过 netlink，操作类型为 RTM_DELADDR 删除 IP 地址时，会调用此函数。

```
446 static int inet_rtm_deladdr(struct sk_buff *skb, struct nlmsghdr *nlh, void *arg)
447 {
448     struct nlattr *tb[IFA_MAX+1];
449     struct in_device *in_dev;
450     struct ifaddrmsg *ifm;
451     struct in_ifaddr *ifa, **ifap;
452     int err = -EINVAL;
453
454     ASSERT_RTNL();
455
456     err = nlmsg_parse(nlh, sizeof(*ifm), tb, IFA_MAX, ifa_ipv4_policy);
457     if (err < 0)
458         goto errout;
459
460     ifm = nlmsg_data(nlh);
461     in_dev = inetdev_by_index(ifm->ifa_index);
462     if (in_dev == NULL) {
```

```
463         err = -ENODEV;
464         goto errout;
465     }
466
467     __in_dev_put(in_dev);
468
469     for (ifap = &in_dev->ifa_list; (ifa = *ifap) != NULL;
470          ifap = &ifa->ifa_next) {
471         if (tb[IFA_LOCAL] &&
472             ifa->ifa_local != nla_get_be32(tb[IFA_LOCAL]))
473             continue;
474
475         if (tb[IFA_LABEL] && nla_strcmp(tb[IFA_LABEL], ifa->ifa_label))
476             continue;
477
478         if (tb[IFA_ADDRESS] &&
479             (ifm->ifa_prefixlen != ifa->ifa_prefixlen ||
480             !inet_ifa_match(nla_get_be32(tb[IFA_ADDRESS]), ifa)))
481             continue;
482
483         __inet_del_ifa(in_dev, ifap, 1, nlh, NETLINK_CB(skb).pid);
484         return 0;
485     }
486
487     err = -EADDRNOTAVAIL;
488 errout:
489     return err;
490 }
```

456-465 解析 netlink 报文，获取配置参数。

469-485 根据本地地址、标签以及掩码查找待删除的 IP 地址块，如果查找命中，则将其删除并释放。

6.3.2　inet_insert_ifa()

inet_insert_ifa()用来添加一个 IP 地址。通常在设置广播地址、点对点对端地址和地址掩码时，先调用 inet_del_ifa()清除原有的信息，然后再调用 inet_insert_ifa()进行设置。

```
392 static int inet_insert_ifa(struct in_ifaddr *ifa)
393 {
394     return __inet_insert_ifa(ifa, NULL, 0);
395 }

340 static int __inet_insert_ifa(struct in_ifaddr *ifa, struct nlmsghdr *nlh,
341                               u32 pid)
342 {
343     struct in_device *in_dev = ifa->ifa_dev;
344     struct in_ifaddr *ifa1, **ifap, **last_primary;
345
346     ASSERT_RTNL();
347
348     if (!ifa->ifa_local) {
349         inet_free_ifa(ifa);
350         return 0;
351     }
```

```
352
353     ifa->ifa_flags &= ~IFA_F_SECONDARY;
354     last_primary = &in_dev->ifa_list;
355
356     for (ifap = &in_dev->ifa_list; (ifa1 = *ifap) != NULL;
357          ifap = &ifa1->ifa_next) {
358         if (!(ifa1->ifa_flags & IFA_F_SECONDARY) &&
359             ifa->ifa_scope <= ifa1->ifa_scope)
360             last_primary = &ifa1->ifa_next;
361         if (ifa1->ifa_mask == ifa->ifa_mask &&
362             inet_ifa_match(ifa1->ifa_address, ifa)) {
363             if (ifa1->ifa_local == ifa->ifa_local) {
364                 inet_free_ifa(ifa);
365                 return -EEXIST;
366             }
367             if (ifa1->ifa_scope != ifa->ifa_scope) {
368                 inet_free_ifa(ifa);
369                 return -EINVAL;
370             }
371             ifa->ifa_flags |= IFA_F_SECONDARY;
372         }
373     }
374
375     if (!(ifa->ifa_flags & IFA_F_SECONDARY)) {
376         net_srandom(ifa->ifa_local);
377         ifap = last_primary;
378     }
379
380     ifa->ifa_next = *ifap;
381     *ifap = ifa;
382
383     /* Send message first, then call notifier.
384        Notifier will trigger FIB update, so that
385        listeners of netlink will know about new ifaddr */
386     rtmsg_ifa(RTM_NEWADDR, ifa, nlh, pid);
387     blocking_notifier_call_chain(&inetaddr_chain, NETDEV_UP, ifa);
388
389     return 0;
390 }
```

353 先清除地址的从属标志，因为配置的地址是主 IP 地址还是从属 IP 地址，并非根据标志而是根据当前已配置的 IP 地址。

356-373 在所有主 IP 地址中查找，如果存在相同寻址范围的地址，则本次添加的 IP 地址为从属 IP 地址。而如果已配置了相同的地址，则返回错误码-EEXIST。

375-381 如果配置的是第一个地址，则先添加熵到伪随机数引擎中，然后将地址添加到 IP 配置块中。

386 通过 netlink 发送 RTM_NEWADDR 消息给感兴趣的用户进程。

387 通过 inetaddr_chain 通知链发送添加 IP 地址事件和 IP 地址信息给感兴趣的其他内核模块。

6.3.3 inet_del_ifa()

inet_del_ifa()用来删除一个 IP 地址。通常在设置广播地址、点对点对端地址和地址掩码时，先调用 inet_del_ifa()清除原有的信息，然后再调用 inet_insert_ifa()进行设置。而通过 netlink 删

除地址时，则直接调用__inet_del_ifa()。参数说明如下：

- in_dev,待删除 IP 地址所属的 IP 配置块。
- ifap，待删除 IP 地址块指针的地址。
- destroy，标识删除后是否释放 IP 地址块。

```
334 static void inet_del_ifa(struct in_device *in_dev, struct in_ifaddr **ifap,
335             int destroy)
336 {
337     __inet_del_ifa(in_dev, ifap, destroy, NULL, 0);
338 }
```

```
244 static void __inet_del_ifa(struct in_device *in_dev, struct in_ifaddr **ifap,
245             int destroy, struct nlmsghdr *nlh, u32 pid)
246 {
247     struct in_ifaddr *promote = NULL;
248     struct in_ifaddr *ifa, *ifa1 = *ifap;
249     struct in_ifaddr *last_prim = in_dev->ifa_list;
250     struct in_ifaddr *prev_prom = NULL;
251     int do_promote = IN_DEV_PROMOTE_SECONDARIES(in_dev);
252
253     ASSERT_RTNL();
254
255     /* 1. Deleting primary ifaddr forces deletion all secondaries
256      * unless alias promotion is set
257      **/
258
259     if (!(ifa1->ifa_flags & IFA_F_SECONDARY)) {
260         struct in_ifaddr **ifap1 = &ifa1->ifa_next;
261
262         while ((ifa = *ifap1) != NULL) {
263             if (!(ifa->ifa_flags & IFA_F_SECONDARY) &&
264                 ifa1->ifa_scope <= ifa->ifa_scope)
265                 last_prim = ifa;
266
267             if (!(ifa->ifa_flags & IFA_F_SECONDARY) ||
268                 ifa1->ifa_mask != ifa->ifa_mask ||
269                 !inet_ifa_match(ifa1->ifa_address, ifa)) {
270                 ifap1 = &ifa->ifa_next;
271                 prev_prom = ifa;
272                 continue;
273             }
274
275             if (!do_promote) {
276                 *ifap1 = ifa->ifa_next;
277
278                 rtmsg_ifa(RTM_DELADDR, ifa, nlh, pid);
279                 blocking_notifier_call_chain(&inetaddr_chain,
280                         NETDEV_DOWN, ifa);
281                 inet_free_ifa(ifa);
282             } else {
283                 promote = ifa;
284                 break;
285             }
286         }
287     }
```

```
288
289     /* 2. Unlink it */
290
291     *ifap = ifa1->ifa_next;
292
293     /* 3. Announce address deletion */
294
295     /* Send message first, then call notifier.
296        At first sight, FIB update triggered by notifier
297        will refer to already deleted ifaddr, that could confuse
298        netlink listeners. It is not true: look, gated sees
299        that route deleted and if it still thinks that ifaddr
300        is valid, it will try to restore deleted routes... Grr.
301        So that, this order is correct.
302      */
303     rtmsg_ifa(RTM_DELADDR, ifa1, nlh, pid);
304     blocking_notifier_call_chain(&inetaddr_chain, NETDEV_DOWN, ifa1);
305
306     if (promote) {
307
308        if (prev_prom) {
309           prev_prom->ifa_next = promote->ifa_next;
310           promote->ifa_next = last_prim->ifa_next;
311           last_prim->ifa_next = promote;
312        }
313
314        promote->ifa_flags &= ~IFA_F_SECONDARY;
315        rtmsg_ifa(RTM_NEWADDR, promote, nlh, pid);
316        blocking_notifier_call_chain(&inetaddr_chain,
317              NETDEV_UP, promote);
318        for (ifa = promote->ifa_next; ifa; ifa = ifa->ifa_next) {
319           if (ifa1->ifa_mask != ifa->ifa_mask ||
320              !inet_ifa_match(ifa1->ifa_address, ifa))
321                 continue;
322           fib_add_ifaddr(ifa);
323        }
324
325     }
326     if (destroy) {
327        inet_free_ifa(ifa1);
328
329        if (!in_dev->ifa_list)
330           inetdev_destroy(in_dev);
331     }
332  }
```

259-287 如果删除的是主 IP 地址，则需对从属 IP 地址作相应的处理。如果没有启用 promote_secondaries，则删除所有该主 IP 地址的从属 IP 地址，否则选择一个从属 IP 地址，升级为主 IP 地址。

291 先将待删除的 IP 地址块从链表中摘除，后续操作中再根据 destroy 作处理。

303 通过 netlink 发送 RTM_DELADDR 消息给感兴趣的用户进程。

304 通过 inetaddr_chain 通知链发送删除 IP 地址事件和 IP 地址信息给感兴趣的其他内核模块。

306-325 如果启用了 promote_secondaries，将选择到的从属 IP 地址升级为主 IP 地址，发

送从属 IP 地址升级为主 IP 地址消息。并通过 fib_add_ifaddr()将从属 IP 地址相关的路由表项添加到 ip_fib_local_table 路由表中。

326-331 如果根据 destroy 需释放，则通过 RCU 机制释放 IP 地址块。在删除掉最后一个地址后，释放所在的 IP 配置块。

6.4　ioctl

应用程序对套接口有关接口层地址的 ioctl 操作，最终由 devinet_ioctl()来处理。各函数之间的调用关系及与各命令的关系如图 6-4 所示。

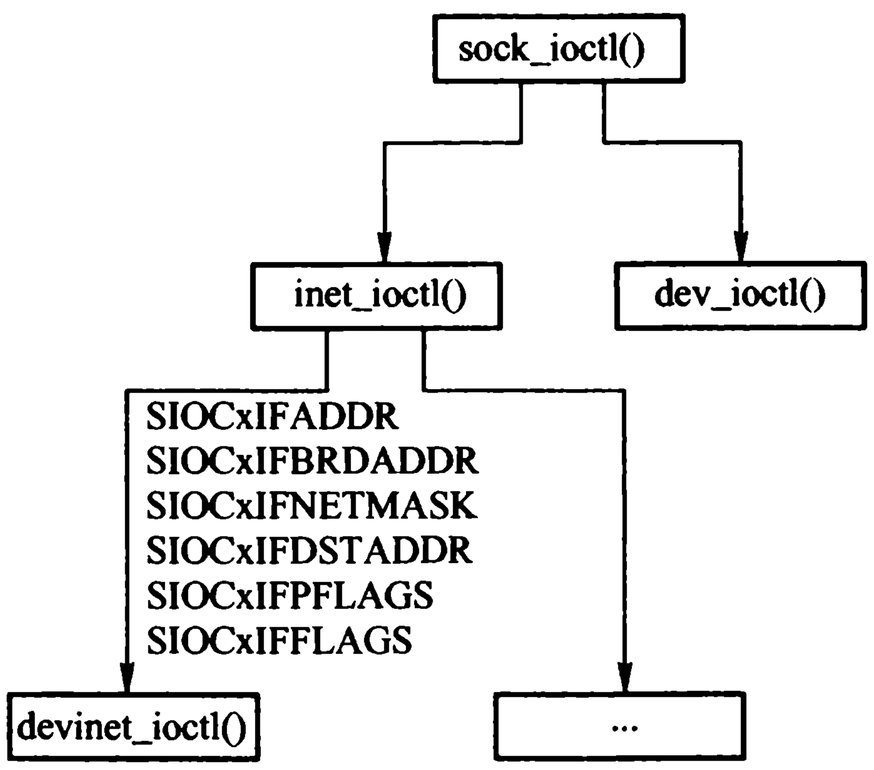

图 6-4　接口层的 ioctl 调用过程

```
603 int devinet_ioctl(unsigned int cmd, void __user *arg)
604 {
605     struct ifreq ifr;
606     struct sockaddr_in sin_orig;
607     struct sockaddr_in *sin = (struct sockaddr_in *)&ifr.ifr_addr;
608     struct in_device *in_dev;
609     struct in_ifaddr **ifap = NULL;
610     struct in_ifaddr *ifa = NULL;
611     struct net_device *dev;
612     char *colon;
613     int ret = -EFAULT;
614     int tryaddrmatch = 0;
615
616     /*
617      *    Fetch the caller's info block into kernel space
618      */
619
620     if (copy_from_user(&ifr, arg, sizeof(struct ifreq)))
621         goto out;
622     ifr.ifr_name[IFNAMSIZ - 1] = 0;
623
624     /* save original address for comparison */
625     memcpy(&sin_orig, sin, sizeof(*sin));
626
627     colon = strchr(ifr.ifr_name, ':');
628     if (colon)
```

```
629         *colon = 0;
630
631 #ifdef CONFIG_KMOD
632     dev_load(ifr.ifr_name);
633 #endif
634
635     switch(cmd) {
636     case SIOCGIFADDR:    /* Get interface address */
637     case SIOCGIFBRDADDR:    /* Get the broadcast address */
638     case SIOCGIFDSTADDR:    /* Get the destination address */
639     case SIOCGIFNETMASK:    /* Get the netmask for the interface */
640         /* Note that these ioctls will not sleep,
641            so that we do not impose a lock.
642            One day we will be forced to put shlock here (I mean SMP)
643          */
644         tryaddrmatch = (sin_orig.sin_family == AF_INET);
645         memset(sin, 0, sizeof(*sin));
646         sin->sin_family = AF_INET;
647         break;
648
649     case SIOCSIFFLAGS:
650         ret = -EACCES;
651         if (!capable(CAP_NET_ADMIN))
652             goto out;
653         break;
654     case SIOCSIFADDR:    /* Set interface address (and family) */
655     case SIOCSIFBRDADDR:    /* Set the broadcast address */
656     case SIOCSIFDSTADDR:    /* Set the destination address */
657     case SIOCSIFNETMASK:    /* Set the netmask for the interface */
658         ret = -EACCES;
659         if (!capable(CAP_NET_ADMIN))
660             goto out;
661         ret = -EINVAL;
662         if (sin->sin_family != AF_INET)
663             goto out;
664         break;
665     default:
666         ret = -EINVAL;
667         goto out;
668     }
669
670     rtnl_lock();
671
672     ret = -ENODEV;
673     if ((dev = __dev_get_by_name(ifr.ifr_name)) == NULL)
674         goto done;
675
676     if (colon)
677         *colon = ':';
678
679     if ((in_dev = __in_dev_get_rtnl(dev)) != NULL) {
680         if (tryaddrmatch) {
681             /* Matthias Andree */
682             /* compare label and address (4.4BSD style) */
683             /* note: we only do this for a limited set of ioctls
684                and only if the original address family was AF_INET.
```

```
685             This is checked above. */
686          for (ifap = &in_dev->ifa_list; (ifa = *ifap) != NULL;
687               ifap = &ifa->ifa_next) {
688             if (!strcmp(ifr.ifr_name, ifa->ifa_label) &&
689                sin_orig.sin_addr.s_addr ==
690                      ifa->ifa_address) {
691                break; /* found */
692             }
693          }
694       }
695       /* we didn't get a match, maybe the application is
696          4.3BSD-style and passed in junk so we fall back to
697             comparing just the label */
698          if (!ifa) {
699             for (ifap = &in_dev->ifa_list; (ifa = *ifap) != NULL;
700                  ifap = &ifa->ifa_next)
701                if (!strcmp(ifr.ifr_name, ifa->ifa_label))
702                   break;
703          }
704       }
705
706       ret = -EADDRNOTAVAIL;
707       if (!ifa && cmd != SIOCSIFADDR && cmd != SIOCSIFFLAGS)
708          goto done;
709
710       switch(cmd) {
:
831       }
832    done:
833       rtnl_unlock();
834    out:
835       return ret;
836    rarok:
837       rtnl_unlock();
838       ret = copy_to_user(arg, &ifr, sizeof(struct ifreq)) ? -EFAULT : 0;
839       goto out;
840    }
```

620-622 从用户空间复制配置参数。

625 将原始的配置参数保存起来用于后续的比较操作。

627-629 配置的设备名中如果存在“:”，则表示配置了别名。由于需要根据名称作操作，因此先将该设备名截断，后续再恢复。

632 根据网络设备名，加载相应的设备驱动模块。

635-668进行相关校验。对于获取操作，则检测地址族是否为AF_INET；对于设置操作，则必须要有相应的特权；而对于SIOCSIFADDR、SIOCSIFBRDADDR、SIOCSIFDSTADDR和SIOCSIFNETMASK操作，地址族也必须是AF_INET。

673-674 根据网络设备名获取网络设备。

676-677 恢复配置参数中的标签别名。

679-704 取IP配置块，及用户地址标签对应的设备地址结构。

679-708 设置地址和标志。SIOCSIFFLAGS是设置网络设备的标志，SIOCSIFADDR是添加IP地址，这两个操作不针对现有的IP地址块。而其他操作，如SIOCGIFBRDADDR，都

是针对现有的 IP 地址块，如果不存在与配置参数中的标签或地址匹配的 IP 地址块，则不能继续操作。

710-831 针对具体的命令进行操作。

632-839 完成操作，返回相应的操作结果。

IP 地址操作相关命令如下：

（1）SIOCGIFADDR

获取指定网络设备的本地 IP 地址。

```
711     case SIOCGIFADDR:    /* Get interface address */
712         sin->sin_addr.s_addr = ifa->ifa_local;
713         goto rarok;
```

（2）SIOCGIFBRDADDR

获取指定网络设备的组播地址。

```
715     case SIOCGIFBRDADDR:    /* Get the broadcast address */
716         sin->sin_addr.s_addr = ifa->ifa_broadcast;
717         goto rarok;
```

（3）SIOCGIFDSTADDR

在点对点连接的情况下，获取指定网络设备点对点对端的 IP 地址。

```
719     case SIOCGIFDSTADDR:    /* Get the destination address */
720         sin->sin_addr.s_addr = ifa->ifa_address;
721         goto rarok;
```

（4）SIOCGIFNETMASK

获取指定网络设备的地址掩码。

```
723     case SIOCGIFNETMASK:    /* Get the netmask for the interface */
724         sin->sin_addr.s_addr = ifa->ifa_mask;
725         goto rarok;
```

（5）SIOCSIFFLAGS

设置网络设备的标志。

```
727     case SIOCSIFFLAGS:
728         if (colon) {
729             ret = -EADDRNOTAVAIL;
730             if (!ifa)
731                 break;
732             ret = 0;
733             if (!(ifr.ifr_flags & IFF_UP))
734                 inet_del_ifa(in_dev, ifap, 1);
735             break;
736         }
737         ret = dev_change_flags(dev, ifr.ifr_flags);
738         break;
```

728-736对于关闭网络设备，如果指定了网络设备别名，并且存在与之对应的 IP 地址块，

则需要删除释放该IP地址块。

737 将标志设置到网络设备中。

（6）SIOCSIFADDR

设置指定网络设备的本地地址。

```
740     case SIOCSIFADDR:    /* Set interface address (and family) */
741        ret = -EINVAL;
742        if (inet_abc_len(sin->sin_addr.s_addr) < 0)
743           break;
744
745        if (!ifa) {
746           ret = -ENOBUFS;
747           if ((ifa = inet_alloc_ifa()) == NULL)
748              break;
749           if (colon)
750              memcpy(ifa->ifa_label, ifr.ifr_name, IFNAMSIZ);
751           else
752              memcpy(ifa->ifa_label, dev->name, IFNAMSIZ);
753        } else {
754           ret = 0;
755           if (ifa->ifa_local == sin->sin_addr.s_addr)
756              break;
757           inet_del_ifa(in_dev, ifap, 0);
758           ifa->ifa_broadcast = 0;
759           ifa->ifa_anycast = 0;
760        }
761
762        ifa->ifa_address = ifa->ifa_local = sin->sin_addr.s_addr;
763
764        if (!(dev->flags & IFF_POINTOPOINT)) {
765           ifa->ifa_prefixlen = inet_abc_len(ifa->ifa_address);
766           ifa->ifa_mask = inet_make_mask(ifa->ifa_prefixlen);
767           if ((dev->flags & IFF_BROADCAST) &&
768              ifa->ifa_prefixlen < 31)
769              ifa->ifa_broadcast = ifa->ifa_address |
770                        ~ifa->ifa_mask;
771        } else {
772           ifa->ifa_prefixlen = 32;
773           ifa->ifa_mask = inet_make_mask(32);
774        }
775        ret = inet_set_ifa(dev, ifa);
776        break;
```

742-743 根据本地地址默认的掩码长度，校验本地地址的有效性。

745-752 如果尚未分配IP地址块，则进行分配，并将网络设备别名或网络设备名设置到地址标签中。

757 首先将对应IP地址块从地址列表中删除。

762 然后设置本地地址。

764-774 接着根据接口是否为点对点设备，来设置子网掩码长度和子网掩码。如果是非点对点设备，则根据地址的掩码长度和网络掩码设置标准广播地址；否则网络掩码长度为32。

775 最后将配置信息再添加到IP地址块列表中。

（7）SIOCSIFBRDADDR

设置指定网络设备的组播地址。

```
778     case SIOCSIFBRDADDR:    /* Set the broadcast address */
779         ret = 0;
780         if (ifa->ifa_broadcast != sin->sin_addr.s_addr) {
781             inet_del_ifa(in_dev, ifap, 0);
782             ifa->ifa_broadcast = sin->sin_addr.s_addr;
783             inet_insert_ifa(ifa);
784         }
785         break;
```

如果原有的组播地址与待设置的组播地址不等，则先将对应 IP 地址块从地址列表中删除，然后再将配置信息添加到 IP 地址块列表中。

（8）SIOCSIFDSTADDR

在点对点连接的情况下，设置指定网络设备点对点对端的 IP 地址。

```
787     case SIOCSIFDSTADDR:    /* Set the destination address */
788         ret = 0;
789         if (ifa->ifa_address == sin->sin_addr.s_addr)
790             break;
791         ret = -EINVAL;
792         if (inet_abc_len(sin->sin_addr.s_addr) < 0)
793             break;
794         ret = 0;
795         inet_del_ifa(in_dev, ifap, 0);
796         ifa->ifa_address = sin->sin_addr.s_addr;
797         inet_insert_ifa(ifa);
798         break;
```

789-790 只有当原有的网络设备点对点对端 IP 地址与待设置的地址不等时，才有必要进行设置。

792-793 校验待设置的 IP 地址是否有效。

795-797 先将对应 IP 地址块从地址列表中删除，然后再将待设置的 IP 地址设置到 IP 地址块中并添加到 IP 地址块列表。

（9）SIOCSIFNETMASK

设置指定网络设备的地址掩码。

```
800     case SIOCSIFNETMASK:    /* Set the netmask for the interface */
801
802         /*
803          *    The mask we set must be legal.
804          */
805         ret = -EINVAL;
806         if (bad_mask(sin->sin_addr.s_addr, 0))
807             break;
808         ret = 0;
809         if (ifa->ifa_mask != sin->sin_addr.s_addr) {
810             __be32 old_mask = ifa->ifa_mask;
811             inet_del_ifa(in_dev, ifap, 0);
812             ifa->ifa_mask = sin->sin_addr.s_addr;
```

```
813             ifa->ifa_prefixlen = inet_mask_len(ifa->ifa_mask);
814
815             /* See if current broadcast address matches
816              * with current netmask, then recalculate
817              * the broadcast address. Otherwise it's a
818              * funny address, so don't touch it since
819              * the user seems to know what (s)he's doing...
820              */
821             if ((dev->flags & IFF_BROADCAST) &&
822                 (ifa->ifa_prefixlen < 31) &&
823                 (ifa->ifa_broadcast ==
824                  (ifa->ifa_local|~old_mask))) {
825                     ifa->ifa_broadcast = (ifa->ifa_local |
826                                           ~sin->sin_addr.s_addr);
827                 }
828                 inet_insert_ifa(ifa);
829             }
830             break;
```

806-807 检测待设置的掩码是否有效。

809 原有的掩码与待设置的掩码不等时，才有必要进行设置。

811-813 先将对应 IP 地址块从地址列表中删除，接着如果目前的广播地址与当前的网络掩码匹配，则重新计算广播地址，最后将其设置到 IP 地址块中，并添加到 IP 地址块列表中。

6.5　inetaddr_chain 通知链

内核模块可以通过 register_inetaddr_notifier()将处理 IP 地址事件的函数注册到 inetaddr_chain 通知链中，之后可以通过 unregister_inetaddr_notifier()注销，处理代码中可以对一个或多个事件感兴趣。Linux 系统中有多个内核模块注册到 inetaddr_chain 通知链，如路由、SCTP、ATM。目前提供的 IP 地址事件见表 6-7。

表 6-7　IP 地址事件

IP 地址事件	描述
NETDEV_UP	添加了 IP 地址
NETDEV_DOWN	删除了 IP 地址

第7章　接口层的输入

当数据包到达网络设备时，通常会触发硬件中断。系统在不支持软中断时，数据包的输入过程只能完全在硬件中断中处理。在这种情况下，虽然可以完成数据包的输入，但硬件中断处理所占用的 CPU 资源过多，导致系统对其他硬件响应不够及时。

在有些情况（某些嵌入式设备）下，数据包到达网络设备时并不会触发硬件中断，在这种情况下，只能通过定时器轮询网络设备的状态，当发现有数据包到达时，才从网络设备中读取数据包并输入到协议栈。这种情况下，数据包的输入及时完全需要依靠定时器触发的频率，如果频率过高，可能会过多地消耗 CPU 资源，而频率过低时，数据包的吞吐量便过分低下。

还有一种方法能极大地提高网络处理速度，条件是需要支持硬件中断和软中断。当数据包到达网络设备时，会触发硬中断，将设备添加到轮询队列中，然后关闭硬中断并激活软中断。在软中断中，读取轮询队列内网络设备中的数据包。

本章将论述网络设备通用接口、netpoll 接口及 netconsole 驱动等，涉及以下文件：

- include/linux/if.h，定义 IPv4 专用的接口层使用的结构、宏等。
- include/linux/netdevice.h，定义网络设备结构、宏等。
- include/linux/netpoll.h，定义 netpoll 的接口、宏等。
- net/core/dev.c，网络设备注册、输入和输出等接口。
- drivers/net/e100.c，e100 驱动程序。
- net/ipv4/af_inet.c，IPv4 专用的网络层、传输层接口。
- net/core/netpoll.c，支持 netpoll 的接口。
- drivers/net/netconsole.c，netconsole 驱动程序。

7.1　系统参数

系统参数如下所示：

- dev_weight，在数据包输入软中断中，单个网络设备可读取的报文配额。在达到该配额后，就不再继续从对应的网络设备读取报文，而是将其在网络设备轮询队列内从队首移动至队尾，直到下次轮询调整配额后，可再次读取报文。这样做可以保证每个网络设备都有相同的机会来读取报文。默认值为 64。
- netdev_budget，在数据包输入软中断中后，所有网络设备可从网络设备轮询队列中读取的报文总配额。当从网络设备读取的总报文数达到该值时，会结束本次读取，并产生一个新的数据包输入软中断。默认值为 300。
- netdev_max_backlog，非 NAPI 链路层的缓存队列长度上限。默认值为 1000。
- message_burst 与 message_cost，用来限制那些来自网络模块的警告消息记录到内核日志的频率，以防止拒绝服务攻击。其中 message_cost 为记录的时间间隔，单位为秒，默认值为 5；而 message_burst 是最多能连续记录警告消息的次数，默认值为 10。

7.2　接口层的 ioctl

应用程序对套接口 ioctl 操作，如果是有关接口层的，则由 dev_ioctl()和 inet_ioctl()进行处理。各函数之间的调用关系及其与各命令的关系如图 7-1 所示。

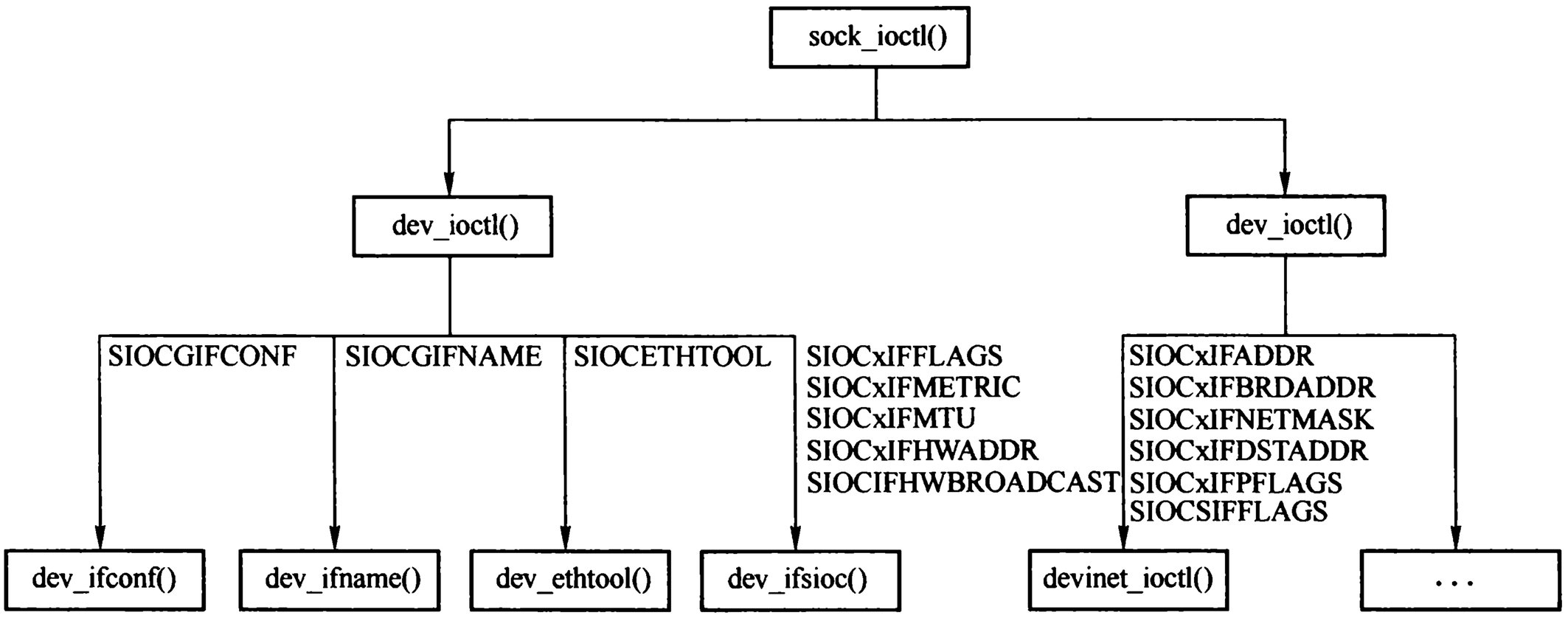

图 7-1　接口层 ioctl 的函数调用关系

7.2.1　SIOCxIFxxx 类命令

在对 SIOCxIFxxx 类命令进行获取或设置操作时，都是通过 ifreq 结构来传递相应的值，或者将 ifreq 结构作为传递接口的一个部分。

```
155 struct ifreq
156 {
157 #define IFHWADDRLEN    6
158     union
159     {
160         char    ifrn_name[IFNAMSIZ];        /* if name, e.g. "en0" */
161     } ifr_ifrn;
162
163     union {
164         struct    sockaddr ifru_addr;
165         struct    sockaddr ifru_dstaddr;
166         struct    sockaddr ifru_broadaddr;
167         struct    sockaddr ifru_netmask;
168         struct  sockaddr ifru_hwaddr;
169         short    ifru_flags;
170         int    ifru_ivalue;
171         int    ifru_mtu;
172         struct  ifmap ifru_map;
173         char    ifru_slave[IFNAMSIZ];    /* Just fits the size */
174         char    ifru_newname[IFNAMSIZ];
175         void __user *    ifru_data;
176         struct    if_settings ifru_settings;
177     } ifr_ifru;
178 };
```

```
158 union { } ifr_ifrn
160 char ifrn_name[IFNAMSIZ]
```

网络设备名。在执行 SIOCGIFNAME 命令时，存储获得的网络设备名；而在执行其他命令时，存储指定的网络设备名，由该网络设备名得到对应的网络设备。

```
union { } ifr_ifru
164 struct sockaddr ifru_addr
```

用来获取和设置指定网络设备的广播地址、点对点对端地址、本地地址以及网络掩码。

```
165 struct sockaddr ifru_dstaddr
```

原本用来获取和设置点对点对端地址的，目前尚未使用。

```
166 struct sockaddr ifru_broadaddr
```

原本用来获取和设置网络设备广播地址的，目前尚未使用。

```
167 struct sockaddr ifru_netmask
```

原本用来获取和设置网络设备掩码的，目前尚未使用。

```
168 struct sockaddr ifru_hwaddr
```

用来获取和设置网络设备的硬件地址。

```
169 short ifru_flags
```

用来获取和设置网络设备的标志。

```
170 int ifru_ivalue
```

在通过 ioctl()获取/设置网络设备的属性等操作时，标识网络设备的索引号、网络设备输出队列的长度。

```
171 int ifru_mtu
```

用来获取和设置网络设备的 MTU。

```
172 struct ifmap ifru_map
```

用来获取和设置网络设备的硬件参数，与 net_device 结构的 mem_start、mem_end、base_addr、irq、dma 以及 if_port 成员一一对应。

```
173 char ifru_slave[IFNAMSIZ]
```

目前未使用。

```
174 char ifru_newname[IFNAMSIZ]
```

用于设置网络设备新的名称。

```
175 void __user *ifru_data
```

在执行 SIOCETHTOOL 命令时，根据不同的子命令对应不同的结构，参见 dev_ethtool()。

```
176 struct if_settings ifru_settings
```

用来设置相关设备及协议，如高级数据链路控制（HDLC）等。

1. SIOCGIFCONF

SIOCGIFCONF 返回系统中所有接口的配置信息。目前该命令只在 AF_INET 地址族中有效。

```
205 struct ifconf
206 {
207     int    ifc_len;            /* size of buffer    */
208     union
209     {
210         char __user *ifcu_buf;
```

```
211         struct ifreq __user *ifcu_req;
212     } ifc_ifcu;
213 };
```

207 int ifc_len

操作前，标识 ifc_ifcu 所指向缓存区的长度；操作完成后，标识缓存区中实际有效数据的长度。

208 union { } ifc_ifcu

指向用户空间中的接口配置信息缓存区。

例如，获取系统中配置的所有接口的配置信息的代码如下所示：

```
struct ifreq buf[MAXINTERFACES];
struct ifconf ifc;

ifc.ifc_len = sizeof(buf);
ifc.ifc_buf = (caddr_t)buf;
ioctl(fd, SIOCGIFCONF, (char *)&ifc);
```

2. SIOCGIFNAME 和 SIOCSIFNAME

SIOCGIFNAME 和 SIOCSIFNAME 获取和设置指定网络设备的接口名。另参见 ifreq 结构的 ifrn_name 和 ifru_newname 成员。

3. SIOCGIFFLAGS 和 SIOCSIFFLAGS

SIOCGIFFLAGS 和 SIOCSIFFLAGS 获取和设置指定网络设备的标志。另参见 ifreq 结构的 ifru_flags 成员和 net_device 结构的 flags 成员。

4. SIOCGIFMETRIC 和 SIOCSIFMETRIC

SIOCGIFMETRIC 和 SIOCSIFMETRIC 获取和设置指定网络设备的 metric，目前尚未实现。

5. SIOCGIFMTU 和 SIOCSIFMTU

SIOCGIFMTU 和 SIOCSIFMTU 获取和设置指定网络设备的 MTU。参见 ifreq 结构的 ifru_mtu 成员和 net_device 结构的 mtu 成员。

6. SIOCGIFHWADDR 和 SIOCSIFHWADDR

SIOCGIFHWADDR 和 SIOCSIFHWADDR 获取和设置指定网络设备的硬件地址。参见 ifreq 结构的 ifru_hwaddr 成员和 net_device 结构的 dev_addr 成员。

7. SIOCGIFMAP 和 SIOCSIFMAP

SIOCGIFMAP 和 SIOCSIFMAP 读取和设置接口的硬件参数，包括网络设备所使用的共享内存的起始/终止地址、网络接口的 I/O 基地址、分配给设备的中断号、分配给设备的 DMA 通道以及在多端口设备上指定使用的端口，这些参数并不是对每一个网络设备都是有效的，而是取决于具体的硬件设备和驱动程序。参见 ifmap 结构和 net_device 结构的 mem_start、mem_end、base_addr、irq、dma 和 if_port 成员。

8. SIOCGIFINDEX

SIOCGIFINDEX 获取指定网络设备的索引号。另参见 ifreq 结构的 ifru_ivalue 成员和 net_device 结构的 ifindex 成员。

9. SIOCGIFTXQLEN 和 SIOCSIFTXQLEN

SIOCGIFTXQLEN 和 SIOCSIFTXQLEN 获取和设置指定网络设备的传输队列长度。另参见 ifreq 结构的 ifru_ivalue 成员和 net_device 结构的 tx_queue_len 成员。

10. SIOCGIFADDR 和 SIOCSIFADDR

SIOCGIFADDR 和 SIOCSIFADDR 获取和设置指定网络设备的本地地址。另参见 ifreq 结构的 ifru_addr 成员和 in_ifaddr 结构的 ifa_local 成员。

11. SIOCSIFPFLAGS 和 SIOCGIFPFLAGS

SIOCSIFPFLAGS 和 SIOCGIFPFLAGS 获取和设置指定网络设备扩展的标志，目前未实现。

12. SIOCGIFNETMASK 和 SIOCSIFNETMASK

SIOCGIFNETMASK 和 SIOCSIFNETMASK 获取和设置指定网络设备的地址掩码。参见 ifreq 结构的 ifru_addr 成员和 in_ifaddr 结构的 ifa_mask 成员。

13. SIOCADDMULTI 和 SIOCDELMULTI

SIOCADDMULTI 和 SIOCDELMULTI 添加和删除指定网络设备的组播过滤器，只有组播地址在这些指定值中的组播报文才能接收。参见 ifreq 结构的 ifru_hwaddr 成员和 in_ifaddr 结构的 ifa_broadcast 成员。

14. SIOCSIFHWBROADCAST

SIOCSIFHWBROADCAST 获取、添加或删除指定网络设备的硬件广播地址。参见 ifreq 结构的 ifru_hwaddr 成员和 net_device 结构的 broadcast 成员。

15. SIOCGIFDSTADDR 和 SIOCSIFDSTADDR

SIOCGIFDSTADDR 和 SIOCSIFDSTADDR 在点对点连接的情况下，获取和设置指定网络设备点对点对端的 IP 地址。参见 ifreq 结构的 ifru_addr 成员和 in_ifaddr 结构的 ifa_address 成员。

16. SIOCGIFBRDADDR 和 SIOCSIFBRDADDR

SIOCGIFBRDADDR 和 SIOCSIFBRDADDR 获取和设置指定网络设备的组播地址。参见 ifreq 结构的 ifru_addr 成员和 net_device 结构的 mc_list 成员。

7.2.2 SIOCETHTOOL

SIOCETHTOOL 是以太网设备的专用操作命令，支持多个子命令，不同的子命令有不同的命令结构，其中有些操作是成对的。如 ETHTOOL_GSET 和 ETHTOOL_SSET 是获取和设置相应的设置，命令结构为 ethtool_cmd 结构；而 ETHTOOL_GLINK 则用来获取与网线的连接状态，命令结构为 ethtool_value 结构。应用程序通过 ETHTOOL_GLINK 获取与网线的连接状态的代码如下所示。

```
int get_nic_link_status(const char *if_name)
{
    int fd;
    struct ifreq ifr;
    struct ethtool_value edata;

    edata.cmd = ETHTOOL_GLINK;
    edata.data = 0;

    memset(&ifr, 0, sizeof(ifr));
    strncpy(ifr.ifr_name, if_name, sizeof(ifr.ifr_name) - 1);
    ifr.ifr_data = (char *)&edata;

    if ((fd = socket(AF_INET, SOCK_DGRAM, 0)) == 0) {
        return -1;
    }
```

```
    if(ioctl(fd, SIOCETHTOOL, &ifr) == -1) {
        close(fd);
        return -1;
    }

    close(skfd);
    return edata.data;
}
```

net_device 结构中的 ethtool_ops 字段为一组函数指针集合，通过它来提供 Ethtool 接口。如果设备驱动程序不支持这个特性，则会试着调用 dev->do_ioctl()处理此命令。

7.2.3　私有命令

1．SIOCGMIIPHY

SIOCGMIIPHY 获取 MII 的 PHY 地址。

2．SIOCGMIIREG 和 SIOCSMIIREG

SIOCGMIIREG 和 SIOCSMIIREG 读取和设置 MII 的 PHY 寄存器。

7.3　初始化

在系统启动时，net_dev_init()的初始化优先级是 subsys_initcall，用来初始化相关接口层，如，注册记录相关统计信息的 proc 文件，初始化每个 CPU 的 softnet_data，注册网络报文输入/输出软中断以及处理例程，注册响应 CPU 状态变化的回调函数等。

```
3476 static int __init net_dev_init(void)
3477 {
3478     int i, rc = -ENOMEM;
3479
3480     BUG_ON(!dev_boot_phase);
3481
3482     if (dev_proc_init())
3483         goto out;
3484
3485     if (netdev_sysfs_init())
3486         goto out;
3487
3488     INIT_LIST_HEAD(&ptype_all);
3489     for (i = 0; i < 16; i++)
3490         INIT_LIST_HEAD(&ptype_base[i]);
3491
3492     for (i = 0; i < ARRAY_SIZE(dev_name_head); i++)
3493         INIT_HLIST_HEAD(&dev_name_head[i]);
3494
3495     for (i = 0; i < ARRAY_SIZE(dev_index_head); i++)
3496         INIT_HLIST_HEAD(&dev_index_head[i]);
3497
3498     /*
3499      *    Initialise the packet receive queues.
3500      */
3501
3502     for_each_possible_cpu(i) {
```

```
3503        struct softnet_data *queue;
3504
3505        queue = &per_cpu(softnet_data, i);
3506        skb_queue_head_init(&queue->input_pkt_queue);
3507        queue->completion_queue = NULL;
3508        INIT_LIST_HEAD(&queue->poll_list);
3509        set_bit(__LINK_STATE_START, &queue->backlog_dev.state);
3510        queue->backlog_dev.weight = weight_p;
3511        queue->backlog_dev.poll = process_backlog;
3512        atomic_set(&queue->backlog_dev.refcnt, 1);
3513    }
3514
3515    netdev_dma_register();
3516
3517    dev_boot_phase = 0;
3518
3519    open_softirq(NET_TX_SOFTIRQ, net_tx_action, NULL);
3520    open_softirq(NET_RX_SOFTIRQ, net_rx_action, NULL);
3521
3522    hotcpu_notifier(dev_cpu_callback, 0);
3523    dst_init();
3524    dev_mcast_init();
3525    rc = 0;
3526 out:
3527    return rc;
3528 }
```

3482-3483 注册接口层用于显示网络设备收发数据包统计信息的“/proc/net/dev”文件，以及用于显示每个 CPU 的 softnet_stat 统计信息的“/proc/net/softnet_stat”文件。

3485-3486 为网络设备创建 sys 文件系统。

3488-3490 初始化 ptype_all 链表和 ptype_base 散列表。

3492-3493 初始化 dev_name_head 散列表，该散列表用来根据网络设备名获取网络设备。

3495-3496 初始化 dev_index_head 散列表，该散列表用来根据网络设备索引号获取网络设备。

3502-3513 初始化每个 CPU 的 softnet_data，包括完成发送数据包的等待释放队列，以及非 NAPI 驱动的输入队列，轮询函数。

3515 为网络子系统注册一个 DMA 客户端。

3519-3520 注册网络报文输入/输出软中断及其处理例程。

3522 注册响应 CPU 状态变化的回调函数。当 CPU 状态发生变化时，会调用 dev_cpu_callback()，来处理状态发生变化 CPU 的 softnet_data 中相关队列。

3523 注册响应网络状态变化的回调函数。当网络状态发生变化时，会调用 dev_cpu_callback()，处理各 CPU 的 softnet_data 中相关队列。

3524 注册接口层用来显示相关网络设备组播硬件地址的“/proc/net/dev_mcast”文件。

7.4 softnet_data 结构

softnet_data 结构描述了与网络软中断处理相关的报文输入和输出队列，每个 CPU 有一个单独的 softnet_data 实例，因此在操作该结构中的成员时不必加锁。该结构在接口层与网络层之间起着承上启下的作用，如图 7-2 所示。

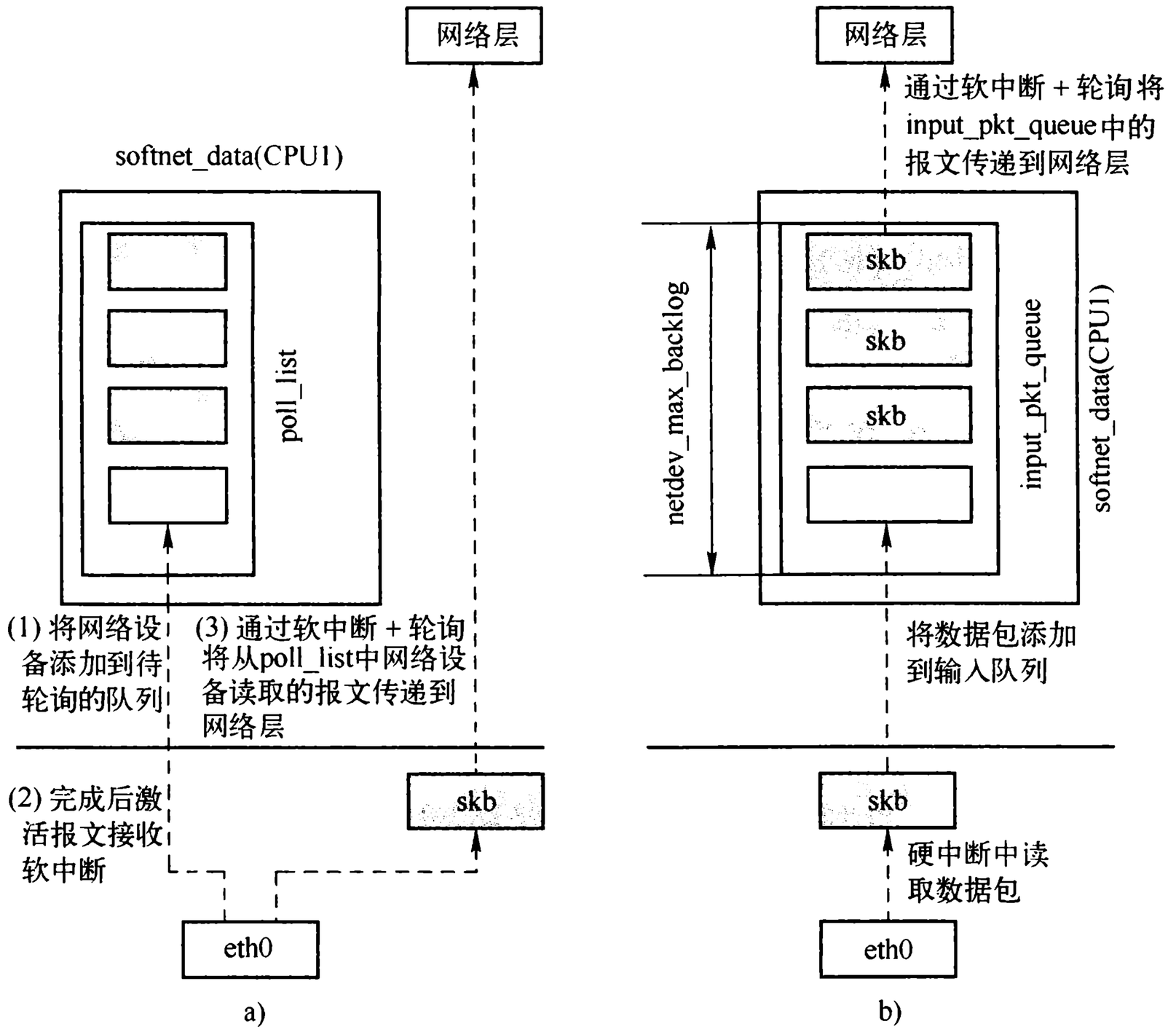

图 7-2　softnet_data 与接口层和网络层之间的关系

a) NP NAPI 方式　b) 非 NAPI 方式

```
616 struct softnet_data
617 {
618     struct net_device   *output_queue;
619     struct sk_buff_head   input_pkt_queue;
620     struct list_head   poll_list;
621     struct sk_buff      *completion_queue;
622
623     struct net_device   backlog_dev;    /* Sorry. 8) */
624 #ifdef CONFIG_NET_DMA
625     struct dma_chan      *net_dma;
626 #endif
627 };
```

```
618 struct net_device *output_queue
```

数据包输出软中断中输出数据包的网络设备队列。处于报文输出状态的网络设备添加到该队列上，在数据包输出软中断中，会遍历该队列，从网络设备的排队规则中获取数据包并输出。

```
61 struct sk_buff_head input_pkt_queue
```

非 NAPI 的接口层缓存队列。对于非 NAPI 的驱动，通常在硬中断中或通过轮询读取报文后，调用 netif_rx()将接收到的报文传递到上层，即先将报文缓存到 input_pkt_queue 队列中，然后产生一个数据包输入软中断，由软中断例程将报文传递到上层。这在接口接收数据包的速率比协议栈和应用层快时非常有用。队列长度上限参见系统参数 netdev_max_backlog。

```
620 struct list_head poll_list
```

网络设备轮询队列。处于报文接收状态的网络设备链接到该队列上，在数据包输入软中断中，会遍历该队列，通过轮询方式接收报文。

621 struct sk_buff *completion_queue

完成发送数据包的等待释放队列。需在适当的时机释放已完成发送的数据包，在发送报文软中断中会检测该队列。是否将完成发送的数据包添加到该队列，与具体的执行环境相关，发送完成之后调用 dev_kfree_skb_any()，如果正处于中断处理过程中或中断被禁止，则会将待释放的 SKB 添加到 completion_queue 队列，否则直接将其释放。

623 struct net_device backlog_dev

用于非 NAPI 驱动的虚拟网络设备，不代表具体的网络设备，用来兼容非 NAPI 的驱动。通过该虚拟网络设备的 poll 回调函数在接收报文软中断中，从非 NAPI 的接口层缓存队列 input_pkt_queue 中获取报文向上层传递。

625 struct dma_chan *net_dma

网络设备的 DMA 相关，本书不作论述。

7.5 NAPI 方式

NAPI 是中断机制与轮询机制的混合体，能有效地提高网络处理速度。在网络负荷较重时，NAPI 技术能显著减少由于接收到数据包而产生的硬中断数量，对高速率、短长度的数据包的处理非常有效。

NAPI 的实现方法：当一批数据包中的第一个数据包到达网络设备时，会以硬中断的方式通知系统；在硬中断例程中，系统将该设备添加到 CPU 的设备轮询队列中，并关闭中断，同时激活数据包输入软中断；由软中断例程遍历轮询队列中的网络设备，从中读取数据包。这样，在内核从网络设备中接收报文的过程中，若有新的报文到来，NAPI 也无需再执行中断例程，而只需维护网络设备轮询队列，就能读取到新的报文。当全部设备读取完成后，再打开中断。图 7-3 为 NAPI 的实现流程图。

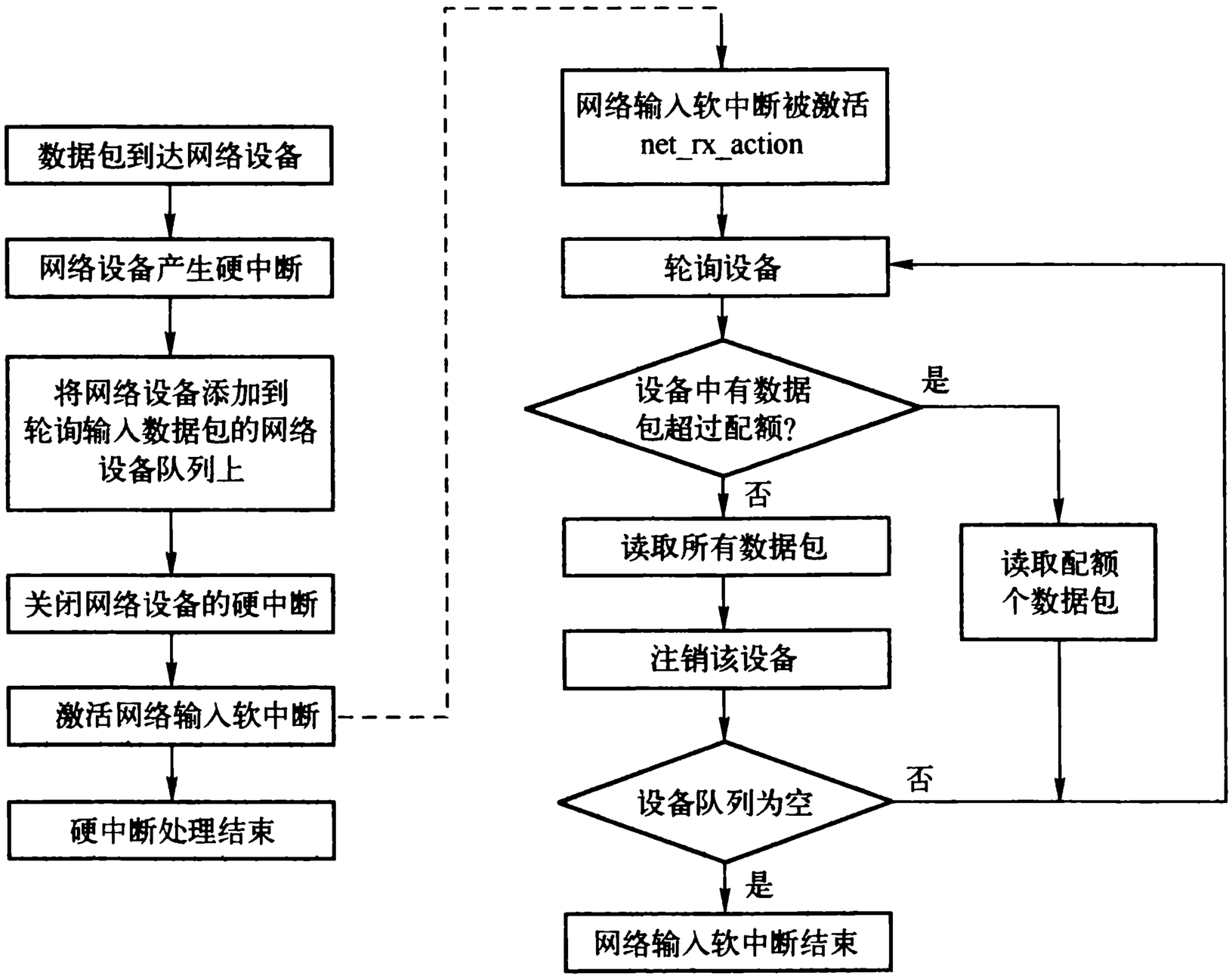

图 7-3 NAPI 方式输入报文流程

7.5.1　网络设备中断例程

e100_intr()为 e100 网络设备驱动程序的中断处理例程。当有网络数据包到达网络设备时，网络设备会触发中断，然后由 e100_intr()进行处理。

```
1951 static irqreturn_t e100_intr(int irq, void *dev_id)
1952 {
1953     struct net_device *netdev = dev_id;
1954     struct nic *nic = netdev_priv(netdev);
1955     u8 stat_ack = readb(&nic->csr->scb.stat_ack);
1956
1957     DPRINTK(INTR, DEBUG, "stat_ack = 0x%02X\n", stat_ack);
1958
1959     if(stat_ack == stat_ack_not_ours ||    /* Not our interrupt */
1960        stat_ack == stat_ack_not_present)    /* Hardware is ejected */
1961         return IRQ_NONE;
1962
1963     /* Ack interrupt(s) */
1964     writeb(stat_ack, &nic->csr->scb.stat_ack);
1965
1966     /* We hit Receive No Resource (RNR); restart RU after cleaning */
1967     if(stat_ack & stat_ack_rnr)
1968         nic->ru_running = RU_SUSPENDED;
1969
1970     if(likely(netif_rx_schedule_prep(netdev))) {
1971         e100_disable_irq(nic);
1972         __netif_rx_schedule(netdev);
1973     }
1974
1975     return IRQ_HANDLED;
1976 }
```

1959-1968 检测该中断是不是由网络设备激活，或者检测网络设备是否有效，以及响应硬件中断等操作。这些不是本书的关注点，可以跳过。

1970-1973 如果网络设备工作正常且系统尚未轮询接收报文，则禁止该网络设备的中断，并将该网络设备添加到网络设备轮询队列中，然后更新该网络设备读取报文数量的配额，最后激活接收报文软中断。否则，说明系统正在轮询接收报文，无需再激活接收报文软中断。

7.5.2　网络输入软中断

net_rx_action()为网络输入软中断的处理例程，当网络设备有数据包输入时，非 NAPI 和 NAPI 的网络设备驱动程序，一般都会激活网络输入软中断进行处理，以提高系统的性能。

```
1904 static void net_rx_action(struct softirq_action *h)
1905 {
1906     struct softnet_data *queue = &__get_cpu_var(softnet_data);
1907     unsigned long start_time = jiffies;
1908     int budget = netdev_budget;
1909     void *have;
```

```
1910
1911    local_irq_disable();
1912
1913    while (!list_empty(&queue->poll_list)) {
1914        struct net_device *dev;
1915
1916        if (budget <= 0 || jiffies - start_time > 1)
1917            goto softnet_break;
1918
1919        local_irq_enable();
1920
1921        dev = list_entry(queue->poll_list.next,
1922                 struct net_device, poll_list);
1923        have = netpoll_poll_lock(dev);
1924
1925        if (dev->quota <= 0 || dev->poll(dev, &budget)) {
1926            netpoll_poll_unlock(have);
1927            local_irq_disable();
1928            list_move_tail(&dev->poll_list, &queue->poll_list);
1929            if (dev->quota < 0)
1930                dev->quota += dev->weight;
1931            else
1932                dev->quota = dev->weight;
1933        } else {
1934            netpoll_poll_unlock(have);
1935            dev_put(dev);
1936            local_irq_disable();
1937        }
1938    }
1939 out:
1940 #ifdef CONFIG_NET_DMA
1941    /*
1942     * There may not be any more sk_buffs coming right now, so push
1943     * any pending DMA copies to hardware
1944     */
1945    if (net_dma_client) {
1946        struct dma_chan *chan;
1947        rcu_read_lock();
1948        list_for_each_entry_rcu(chan, &net_dma_client->channels, client_node)
1949            dma_async_memcpy_issue_pending(chan);
1950        rcu_read_unlock();
1951    }
1952 #endif
1953    local_irq_enable();
1954    return;
1955
1956 softnet_break:
1957    __get_cpu_var(netdev_rx_stat).time_squeeze++;
1958    __raise_softirq_irqoff(NET_RX_SOFTIRQ);
1959    goto out;
1960 }
```

1906 获取所在 CPU 的 softnet_data。

1908 获取本次软中断的接收报文配额。

1913 遍历网络设备轮询队列上的网络设备，轮询接收这些网络设备上的报文。

1916-1917 本次读取报文数量的总配额已用完，或者软中断处理过程超时大于 1ms，则跳转到标签 softnet_break 处作处理。

1921-1937 从网络设备轮询队列上取出队首的网络设备，如果该网络设备本次读取报文的配额已用完，或者经过一次轮询读取报文后还有报文未读取，则将该网络设备移动到网络设备轮询队列的队尾，等待下次轮询。重新设置该网络设备的读取报文配额，由此不难看出，如果需读取的报文数超过配额，则可能会经过多次轮询才能再次读取到数据。

1945-1951 与网络设备的 DMA 相关，本书不作论述。

1956-1959 对本次读取报文数量的总配额已用完，或者软中断处理过程超时大于 1ms 情况的处理，则结束本次软中断，统计所在 CPU 软中断处理次数，最后开中断。

7.5.3　轮询处理

e100_poll()为 e100 网络设备驱动程序的轮询处理函数，在网络输入软中断处理中，通过函数指针调用该函数。参数说明如下：

- netdev，当前轮询的网络设备。
- budget，该网络设备在本次数据包输入软中断中读取报文配额。

```
1978 static int e100_poll(struct net_device *netdev, int *budget)
1979 {
1980     struct nic *nic = netdev_priv(netdev);
1981     unsigned int work_to_do = min(netdev->quota, *budget);
1982     unsigned int work_done = 0;
1983     int tx_cleaned;
1984
1985     e100_rx_clean(nic, &work_done, work_to_do);
1986     tx_cleaned = e100_tx_clean(nic);
1987
1988     /* If no Rx and Tx cleanup work was done, exit polling mode. */
1989     if((!tx_cleaned && (work_done == 0)) || !netif_running(netdev)) {
1990         netif_rx_complete(netdev);
1991         e100_enable_irq(nic);
1992         return 0;
1993     }
1994
1995     *budget -= work_done;
1996     netdev->quota -= work_done;
1997
1998     return 1;
1999 }
```

1985 调用 e100_rx_clean()从网络设备中读取接收到的报文，并由 netif_receive_skb()输入到上层协议中。work_done 为已读取的报文数。

1986 释放已发送出去的报文。

1989-1993 如果待输出和输入的报文都已处理完成，则退出轮询模式，并从网络设备轮询队列中删除该网络设备，最后开中断。

1995-1996 更新数据包输入软中断中读取报文总配额和当前网络设备的读取报文配额。

7.6 非 NAPI 方式

1. netif_rx()

netif_rx()将从网络设备中接收的报文加入到接口层缓存队列中，以便让上层协议来处理。一般情况下，插入队列的过程都是会成功的，通过队列可以有效地防止由于上层接收拥塞而导致丢弃已接收报文的现象。通常用 NAPI 方式实现的网络设备驱动，不会调用该接口来接收报文。参数 skb 为从网络设备中接收的报文，返回值见表 7-1。

表 7-1 netif_rx()的返回值

返回值	描述
NET_RX_SUCCESS	上层没有发生拥塞
NET_RX_CN_LOW	上层发生低度拥塞
NET_RX_CN_MOD	上层发生中度拥塞
NET_RX_CN_HIGH	上层发生高度拥塞
NET_RX_DROP	报文被丢弃

```
1574 int netif_rx(struct sk_buff *skb)
1575 {
1576     struct softnet_data *queue;
1577     unsigned long flags;
1578
1579     /* if netpoll wants it, pretend we never saw it */
1580     if (netpoll_rx(skb))
1581         return NET_RX_DROP;
1582
1583     if (!skb->tstamp.off_sec)
1584         net_timestamp(skb);
1585
1586     /*
1587      * The code is rearranged so that the path is the most
1588      * short when CPU is congested, but is still operating.
1589      */
1590     local_irq_save(flags);
1591     queue = &__get_cpu_var(softnet_data);
1592
1593     __get_cpu_var(netdev_rx_stat).total++;
1594     if (queue->input_pkt_queue.qlen <= netdev_max_backlog) {
1595         if (queue->input_pkt_queue.qlen) {
1596 enqueue:
1597             dev_hold(skb->dev);
1598             __skb_queue_tail(&queue->input_pkt_queue, skb);
1599             local_irq_restore(flags);
1600             return NET_RX_SUCCESS;
1601         }
1602
```

```
1603        netif_rx_schedule(&queue->backlog_dev);
1604        goto enqueue;
1605    }
1606
1607    __get_cpu_var(netdev_rx_stat).dropped++;
1608    local_irq_restore(flags);
1609
1610    kfree_skb(skb);
1611    return NET_RX_DROP;
1612 }
```

1580-1581 调用 netpoll_rx()将数据包传递给 netpoll 模块，如果有 netpoll 实例接收了，则不再传递到协议栈处理。netpoll 参见 7.9 节。

1583-1584 记录接收报文的时间戳。只有在使能 SO_TIMESTAMP 套接口选项时才记录时间戳。

1591 获取 CPU 的接口层缓存队列。

1593 更新当前 CPU 接口层接收报文数。

1594 如果链路层缓存队列未满，则将该报文加入队列，否则说明上层处理严重阻塞，丢弃该报文。

1595-1601 输入队列不为空，说明该队列由于数据包较多需等待下次软中断处理，因此只需将该数据包加入输入队列即可。

1603-1604 输入队列为空，说明该队列没有被软中断处理过，因此将数据包添加到输入队列，并将虚拟网络设备 backlog_dev 添加到网络设备轮询队列，最后激活数据包输入软中断。在数据包输入软中断处理例程中，会调用 backlog_dev 的轮询函数 process_backlog()，最终将报文传递到上层协议。

1607-1611 如果接口层缓存队列已满，则丢弃接收到的报文，同时更新当前 CPU 接口层丢弃报文数。

2．process_backlog()

process_backlog()为非 NAPI 方式下，虚拟网络设备的轮询函数。当虚拟网络设备 backlog_dev 添加到网络设备轮询队列后，在数据包输入软中断中会调用 process_backlog()进行数据包的输入。参数说明如下：

- backlog_dev，进行轮询的虚拟的网络设备。
- budget，在数据包输入软中断中，网络设备读取报文的配额。

```
1857 static int process_backlog(struct net_device *backlog_dev, int *budget)
1858 {
1859     int work = 0;
1860     int quota = min(backlog_dev->quota, *budget);
1861     struct softnet_data *queue = &__get_cpu_var(softnet_data);
1862     unsigned long start_time = jiffies;
1863
1864     backlog_dev->weight = weight_p;
1865     for (;;) {
1866         struct sk_buff *skb;
1867         struct net_device *dev;
1868
```

```
1869        local_irq_disable();
1870        skb = __skb_dequeue(&queue->input_pkt_queue);
1871        if (!skb)
1872            goto job_done;
1873        local_irq_enable();
1874
1875        dev = skb->dev;
1876
1877        netif_receive_skb(skb);
1878
1879        dev_put(dev);
1880
1881        work++;
1882
1883        if (work >= quota || jiffies - start_time > 1)
1884            break;
1885
1886    }
1887
1888    backlog_dev->quota -= work;
1889    *budget -= work;
1890    return -1;
1891
1892 job_done:
1893    backlog_dev->quota -= work;
1894    *budget -= work;
1895
1896    list_del(&backlog_dev->poll_list);
1897    smp_mb__before_clear_bit();
1898    netif_poll_enable(backlog_dev);
1899
1900    local_irq_enable();
1901    return 0;
1902 }
```

1865 循环处理输入队列 input_pkt_queue 中的报文，直至队列中的所有报文都处理完，或是达到处理报文的配额，或是处理超时。

1870-1872 获取当前待处理报文，获取失败则说明队列为空，即所有报文都已经处理完了，跳转到 job_done 处进行处理。

1877 将当前报文传递到上层协议或转发。

1881 统计本次读取的报文数，用于之后更新网络设备的读取报文配额。

1883-1884 如果处理报文数已达到配额，或处理超时，则结束本次报文输入。

1888 更新所在网络设备的读取报文配额。

1889 更新网络设备轮询队列上所有网络设备的读取报文总配额。

1890 返回非零，表示还有数据包需要输入。

1892-1901 所有数据包都处理完成后，首先更新所在网络设备的读取报文配额和网络设备轮询队列上所有网络设备的读取报文总配额，然后将当前网络设备从网络设备轮询队列中删除并退出轮询状态，最后返回零，表示所有数据包处理完成。

7.7　接口层输入报文的处理

7.7.1　报文接收例程

packet_type 结构为网络层输入接口，系统支持多种协议族，因此每个协议族都会实现一个报文例程。此结构的功能是在链路层和网络层之间起到了桥梁的作用。在以太网上，当以太网帧到达主机后，会根据协议族的报文类型调用相应的网络层接收处理函数。

```
553 struct packet_type {
554     be16            type;  /* This is really htons(ether_type).*/
555     struct net_device   *dev;    /* NULL is wildcarded here     */
556     int             (*func) (struct sk_buff *,
557                              struct net_device *,
558                              struct packet_type *,
559                              struct net_device *);
560     struct sk_buff      *(*gso_segment)(struct sk_buff *skb,
561                                 int features);
562     int             (*gso_send_check)(struct sk_buff *skb);
563     void            *af_packet_priv;
564     struct list_head    list;
565 };
```

```
554 __be16 type
```

标识以太网帧或其他链路层报文承载网络层报文的协议号。

```
555 struct net_device *dev
```

接收从指定网络设备输入的数据包。如果为 NULL，则表示接收来自全部网络设备的数据包。

```
556 int (*func) (struct sk_buff *, struct net_device *, struct packet_type
*, struct net_device *)
```

协议入口接收处理函数。第一个参数为待输入的报文，第二个参数为当前处理该报文的网络设备，第三个参数为报文类型，第四个参数为报文的原始输入网络设备。

通常情况下当前处理该报文的网络设备与报文的原始输入网络设备是同一个网络设备，但在某些情况会是不同的网络设备（例如，启用了 bonding 来实现负载均衡和失效保护），此时，原始的输入设备为物理网络设备，而当前处理该报文的网络设备是虚拟网络设备。

```
560 struct sk_buff *(*gso_segment)(struct sk_buff *skb, int features)
```

```
562 int (*gso_send_check)(struct sk_buff *skb)
```

GSO 是网络设备支持传输层的一个功能。

当 GSO 数据报输出时到达网络设备，如果网络设备不支持 GSO 的情况，则需要传输层对输出的数据报重新进行 GSO 分段和校验和的计算。因此需要网络层提供接口给设备层，能够访问到传输层的 GSO 分段和校验和的计算功能，对输出的数据报进行分段和执行校验和。

gso_segment 接口的作用是回调传输层 GSO 分段方法对大段进行分段。IPv4 中实现的函数为 inet_gso_segment()。

gso_send_check 接口的作用是回调传输层在分段之前对伪首部进行校验和的计算。IPv4 中

实现的函数为 inet_gso_send_check()。

563 void *af_packet_priv

用来存储各协议族的私有数据，在原始套接口中，用于标识是原始套接口的传输控制块。

564 struct list_head list

连接不同协议族报文接收例程的链表。

IPv4 的 packet_type 结构实例 ip_packet_type 定义如下，ip_rcv()是 IP 数据报的接收处理函数。

```
1247 static struct packet_type ip_packet_type = {
1248     .type = __constant_htons(ETH_P_IP),
1249     .func = ip_rcv,
1250     .gso_send_check = inet_gso_send_check,
1251     .gso_segment = inet_gso_segment,
1252 };
```

很多 packet_type 实例以散列表的形式存储在 ptype_base 中。通过 dev_add_pack()将 packet_type 实例添加到 ptype_base 中，通过 dev_remove_pack()将指定的 packet_type 实例从 ptype_base 中删除。然而对于 PF_PACKET 协议族，type 为 ETH_P_ALL 的 packet_type 实例是不注册到 ptype_base 散列表中的，而是注册在 ptype_all 链表中。

在系统初始化时，各个协议族的初始化函数会调用 dev_add_pack()将各自的 packet_type 实例注册到 ptype_base 中。ptype_base 散列表如图 7-4 所示。

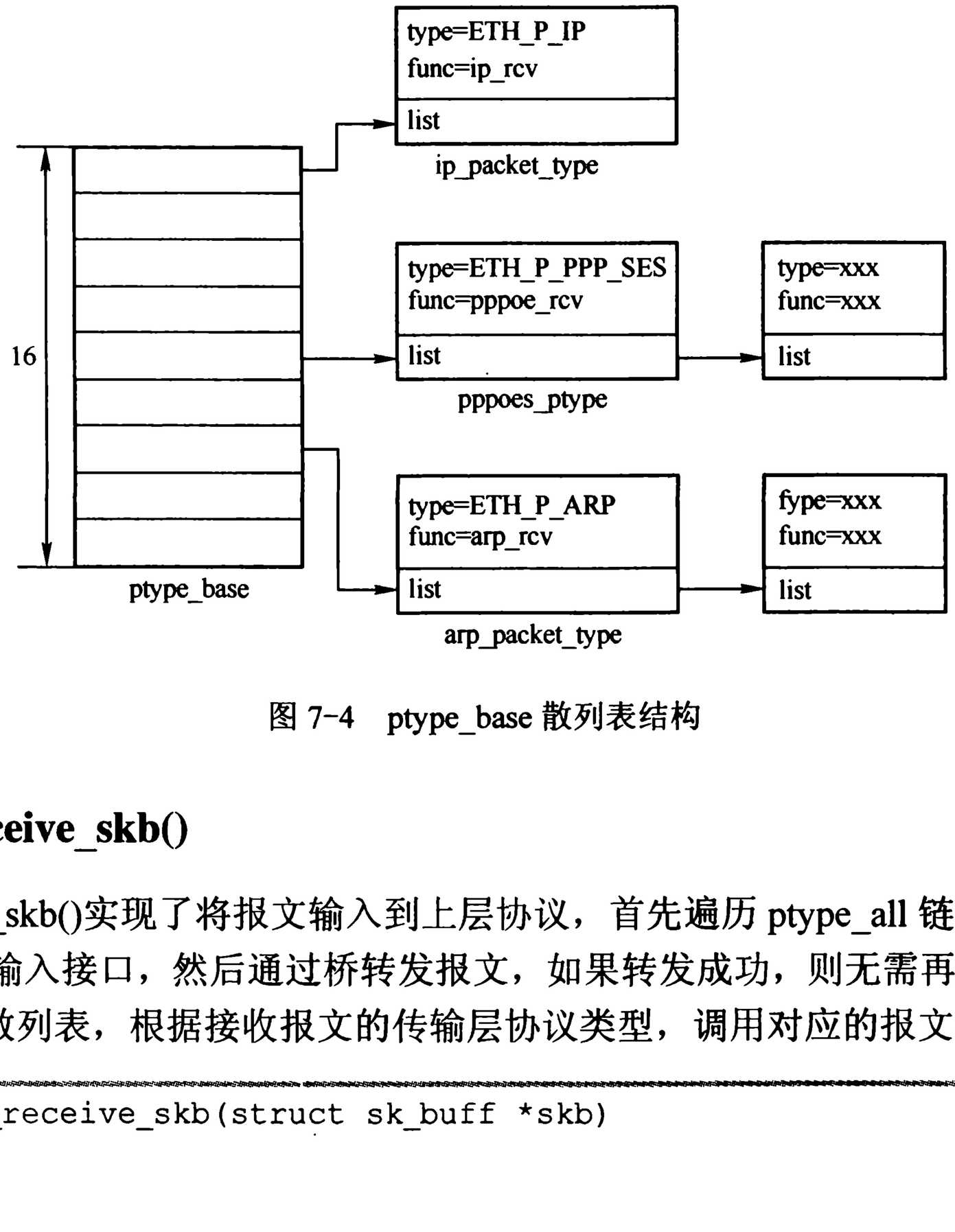

图 7-4　ptype_base 散列表结构

7.7.2　netif_receive_skb()

netif_receive_skb()实现了将报文输入到上层协议，首先遍历 ptype_all 链表，输入一份报文到 ptype_all 链表输入接口，然后通过桥转发报文，如果转发成功，则无需再输入到本地，否则遍历 ptype_base 散列表，根据接收报文的传输层协议类型，调用对应的报文接收例程。

```
1764 int netif_receive_skb(struct sk_buff *skb)
1765 {
```

```
1766    struct packet_type *ptype, *pt_prev;
1767    struct net_device *orig_dev;
1768    int ret = NET_RX_DROP;
1769    __be16 type;
1770
1771    /* if we've gotten here through NAPI, check netpoll */
1772    if (skb->dev->poll && netpoll_rx(skb))
1773        return NET_RX_DROP;
1774
1775    if (!skb->tstamp.off_sec)
1776        net_timestamp(skb);
1777
1778    if (!skb->input_dev)
1779        skb->input_dev = skb->dev;
1780
1781    orig_dev = skb_bond(skb);
1782
1783    if (!orig_dev)
1784        return NET_RX_DROP;
1785
1786    __get_cpu_var(netdev_rx_stat).total++;
1787
1788    skb->h.raw = skb->nh.raw = skb->data;
1789    skb->mac_len = skb->nh.raw - skb->mac.raw;
1790
1791    pt_prev = NULL;
1792
1793    rcu_read_lock();
1794
1795 #ifdef CONFIG_NET_CLS_ACT
1796    if (skb->tc_verd & TC_NCLS) {
1797        skb->tc_verd = CLR_TC_NCLS(skb->tc_verd);
1798        goto ncls;
1799    }
1800 #endif
1801
1802    list_for_each_entry_rcu(ptype, &ptype_all, list) {
1803        if (!ptype->dev || ptype->dev == skb->dev) {
1804            if (pt_prev)
1805                ret = deliver_skb(skb, pt_prev, orig_dev);
1806            pt_prev = ptype;
1807        }
1808    }
1809
1810 #ifdef CONFIG_NET_CLS_ACT
1811    if (pt_prev) {
1812        ret = deliver_skb(skb, pt_prev, orig_dev);
1813        pt_prev = NULL; /* noone else should process this after*/
1814    } else {
1815        skb->tc_verd = SET_TC_OK2MUNGE(skb->tc_verd);
1816    }
1817
1818    ret = ing_filter(skb);
1819
```

```
1820    if (ret == TC_ACT_SHOT || (ret == TC_ACT_STOLEN)) {
1821        kfree_skb(skb);
1822        goto out;
1823    }
1824
1825    skb->tc_verd = 0;
1826 ncls:
1827 #endif
1828
1829    if (handle_bridge(&skb, &pt_prev, &ret, orig_dev))
1830        goto out;
1831
1832    type = skb->protocol;
1833    list_for_each_entry_rcu(ptype, &ptype_base[ntohs(type)&15], list) {
1834        if (ptype->type == type &&
1835            (!ptype->dev || ptype->dev == skb->dev)) {
1836            if (pt_prev)
1837                ret = deliver_skb(skb, pt_prev, orig_dev);
1838            pt_prev = ptype;
1839        }
1840    }
1841
1842    if (pt_prev) {
1843        ret = pt_prev->func(skb, skb->dev, pt_prev, orig_dev);
1844    } else {
1845        kfree_skb(skb);
1846        /* Jamal, now you will not able to escape explaining
1847         * me how you were going to use this. :-)
1848         */
1849        ret = NET_RX_DROP;
1850    }
1851
1852 out:
1853    rcu_read_unlock();
1854    return ret;
1855 }
```

1771-1772 如果是通过 NAPI 方式输入报文的，则在此需将数据包传递给 netpoll 模块，如果有 netpoll 实例接收了，就不再传递到协议栈处理。

1775-1776 记录接收报文时间戳。另参见 SO_TIMESTAMP 套接口选项。

1778-1779 设置原始接收到报文的网络设备。

1781-1784 获取该报文的输入网络设备。有关 bonding 本书不作论述。

1786 统计该 CPU 接收到的数据包数。

1789 获取以太网帧首部的长度。

1796-1799，1810-1827 与包分类器相关，本书不作论述。

1802-1808 遍历 ptype_all 链表，输入一份报文到 ptype_all 链表中协议族。

1829-1830 通过桥转发报文。如果成功，则不需要再输入到本地了，否则还需要将报文输入到本地。与桥有关的资料，读者可自行参阅，本书不作论述。

1832-1850 遍历 ptype_base 散列表，通过接收到报文的传输层协议类型得到与之对应的报文接收例程，然后调用该例程接收报文。

7.7.3　dev_queue_xmit_nit()

对于通过 socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL))创建的原始套接口，不但可以接收从外部输入的数据包，而且对于由本地输出的数据包，如果满足条件，也同样能接收。

dev_queue_xmit_nit()就是用来接收由本地输出的数据包，在链路层的输出过程中，会调用此函数，将满足条件的数据包输入到 RAW 套接口，参见 8.1.2 节。参数说明如下：

- skb，待输出的数据包，如果满足条件，则输入到原始套接口。
- dev，输出数据包的网络设备，如果满足条件，则从该网络设备输入到原始套接口。

```
1061 static void dev_queue_xmit_nit(struct sk_buff *skb, struct net_device *dev)
1062 {
1063     struct packet_type *ptype;
1064
1065     net_timestamp(skb);
1066
1067     rcu_read_lock();
1068     list_for_each_entry_rcu(ptype, &ptype_all, list) {
1069         /* Never send packets back to the socket
1070          * they originated from - MvS (miquels@drinkel.ow.org)
1071          */
1072         if ((ptype->dev == dev || !ptype->dev) &&
1073             (ptype->af_packet_priv == NULL ||
1074              (struct sock *)ptype->af_packet_priv != skb->sk)) {
1075             struct sk_buff *skb2= skb_clone(skb, GFP_ATOMIC);
1076             if (!skb2)
1077                 break;
1078
1079             /* skb->nh should be correctly
1080                set by sender, so that the second statement is
1081                just protection against buggy protocols.
1082              */
1083             skb2->mac.raw = skb2->data;
1084
1085             if (skb2->nh.raw < skb2->data ||
1086                 skb2->nh.raw > skb2->tail) {
1087                 if (net_ratelimit())
1088                     printk(KERN_CRIT "protocol %04x is "
1089                            "buggy, dev %s\n",
1090                            skb2->protocol, dev->name);
1091                 skb2->nh.raw = skb2->data;
1092             }
1093
1094             skb2->h.raw = skb2->nh.raw;
1095             skb2->pkt_type = PACKET_OUTGOING;
1096             ptype->func(skb2, skb->dev, ptype, skb->dev);
1097         }
1098     }
1099     rcu_read_unlock();
1100 }
```

1065 记录该数据包输入的时间戳。

1068 遍历 ptype_all 链表，查找所有符合输入条件的原始套接口，并循环将数据包输入到满足条件的套接口。

1072-1074 数据包的输出设备与套接口的输入设备相符或者套接口不指定输入设备，并且该数据包不是由当前用于比较的套接口输出的（由原始套接口输出的数据包不会再次输入给自己），此时该原始套接口满足条件，数据包可以输入。

1075-1077 由于该数据包是额外输入到这个原始套接口的，因此需要克隆一个数据包。

1083-1092 校验数据包是否有效。

1094-1096 将数据包输入到原始套接口。

7.8 响应 CPU 状态的变化

每个 CPU 都有各自的 softnet_data，通常情况下 CPU 都能处理 softnet_data 中的输出队列和输入队列等。当 CPU 状态变化时，有一个状态需要特殊处理，那就是 CPU_DEAD，此时 CPU 已无法工作，因此需将该 CPU 的 softnet_data 输入输出队列中的报文转交给其他 CPU 处理。为了能响应 CPU 状态的变化，在接口层初始化函数中通过 hotcpu_notifier()注册了响应 CPU 状态变化的回调函数 dev_cpu_callback()。参数说明如下：

- nfb，包括用来响应 CPU 状态变化回调函数的信息块。
- action，状态发生变化的 CPU 的当前状态。
- ocpu，状态发生变化的 CPU。

```
3343 static int dev_cpu_callback(struct notifier_block *nfb,
3344                 unsigned long action,
3345                 void *ocpu)
3346 {
3347     struct sk_buff **list_skb;
3348     struct net_device **list_net;
3349     struct sk_buff *skb;
3350     unsigned int cpu, oldcpu = (unsigned long)ocpu;
3351     struct softnet_data *sd, *oldsd;
3352
3353     if (action != CPU_DEAD)
3354         return NOTIFY_OK;
3355
3356     local_irq_disable();
3357     cpu = smp_processor_id();
3358     sd = &per_cpu(softnet_data, cpu);
3359     oldsd = &per_cpu(softnet_data, oldcpu);
3360
3361     /* Find end of our completion_queue. */
3362     list_skb = &sd->completion_queue;
3363     while (*list_skb)
3364         list_skb = &(*list_skb)->next;
3365     /* Append completion queue from offline CPU. */
3366     *list_skb = oldsd->completion_queue;
3367     oldsd->completion_queue = NULL;
3368
```

```
3369    /* Find end of our output_queue. */
3370    list_net = &sd->output_queue;
3371    while (*list_net)
3372       list_net = &(*list_net)->next_sched;
3373    /* Append output queue from offline CPU. */
3374    *list_net = oldsd->output_queue;
3375    oldsd->output_queue = NULL;
3376
3377    raise_softirq_irqoff(NET_TX_SOFTIRQ);
3378    local_irq_enable();
3379
3380    /* Process offline CPU's input_pkt_queue */
3381    while ((skb = __skb_dequeue(&oldsd->input_pkt_queue)))
3382       netif_rx(skb);
3383
3384    return NOTIFY_OK;
3385 }
```

3353-3354 只处理 CPU_DEAD 状态，处于该状态的 CPU 已不能再处理其 softnet_data 上的相关队列了，因此需作相应的处理。

3357-3359 获取状态发生变化 CPU 的 softnet_data 以及当前 CPU 的 softnet_data。

3362-3367 将状态发生变化 CPU 的 completion_queue 队列中的报文转移到当前 CPU 的 completion_queue 队列。

3370-3375 将状态发生变化 CPU 的 output_queue 队列中的报文转移到当前 CPU 的 output_queue 队列。

3377 经过以上操作，当前 CPU 的 softnet_data 中可能存在完成输出和等待输出的报文，因此再次激活数据包输出软中断，以便释放完成输出的报文，输出等待发送的报文。

3381-3382 最后处理状态发生变化 CPU 的 input_pkt_queue 队列，将队列上的报文输入到上层协议。

7.9　netpoll

netpoll 实际上只是一种框架和一些接口，只有依赖这个框架和接口实现的 netpoll 实例，netpoll 才能发挥它的功能。例如 netconsole，可以将输出到 console 的信息通过 UDP 发送到监控主机。

7.9.1　netpoll 相关结构

netpoll 实例有两种类型：能接收数据包和不能接收数据包。但无论能否接收数据包，netpoll 实例必须实现 netpoll 并注册，唯一区别是，不接收数据包的 netpoll 实例可以不用实现 rx_hook 接口。netpoll 结构用来描述接收和发送的必要信息，可以称之为 netpoll 实例配置块。

```
15 struct netpoll {
16    struct net_device *dev;
17    char dev_name[IFNAMSIZ];
18    const char *name;
19    void (*rx_hook)(struct netpoll *, int, char *, int);
```

```
20
21     u32 local_ip, remote_ip;
22     u16 local_port, remote_port;
23     u8 local_mac[ETH_ALEN], remote_mac[ETH_ALEN];
24 };
```

16 struct net_device *dev

netpoll 实例所绑定的网络设备，通过该网络设备接收和发送报文。

17 char dev_name[IFNAMSIZ]

netpoll 实例所绑定的网络设备名，如“eth0”。

18 const char *name

netpoll 实例名，比如“netconsole”。

19 void (*rx_hook)(struct netpoll *, int, char *, int)

netpoll 实例报文接收处理例程。如果只希望通过 netpoll 接收报文，则需实现该函数，而如果只用于输出，则无需实现。netpoll 实例接收处理的报文，不会再向上传递到协议栈。

21 u32 local_ip, remote_ip

本机及远端 IP 地址。

22 u16 local_port, remote_port

本机及远端 UDP 端口号。

23 u8 local_mac[ETH_ALEN], remote_mac[ETH_ALEN]

本机及远端 MAC 地址。

当支持 netpoll 时，描述网络设备的 net_device 结构，必须实现 npinfo 成员，即网络设备的 netpoll 信息块，包括输入标志、报文接收例程、ARP 输入队列等。

```
26 struct netpoll_info {
27     atomic_t refcnt;
28     spinlock_t poll_lock;
29     int poll_owner;
30     int rx_flags;
31     spinlock_t rx_lock;
32     struct netpoll *rx_np; /* netpoll that registered an rx_hook */
33     struct sk_buff_head arp_tx; /* list of arp requests to reply to */
34     struct sk_buff_head txq;
35     struct delayed_work tx_work;
36 };
```

27 atomic_t refcnt

引用计数。

28 spinlock_t poll_lock

并发访问锁，以确保同一时刻只有一个 CPU 调用网络设备的 poll 接口进行轮询操作。

29 int poll_owner

标识正在从网络设备中读取数据包的 CPU。为-1 时，表示没有 CPU 在从网络设备中读取数据包。通常情况是上一次数据包输入软中断没能处理完的达到网络设备的报文，等待下一次数据包输入软中断处理。在两次数据包输入软中断处理间隙，只能由在其他 CPU 执行的 netpoll 实例在可能的情况下顺便读取报文。

30 int rx_flags

标识接收特性，见表 7-2。

表 7-2　rx_flags 的取值

rx_flags	描述
NETPOLL_RX_ENABLED	所在 netpoll 实例允许输入的报文
NETPOLL_RX_DROP	尚未使用

31 spinlock_t rx_lock

并发访问锁，以确保同一时刻只有一个 CPU 在进行相关 netpoll 的输入操作。

32 struct netpoll *rx_np

为 netpoll 接收而注册的相关信息。虽然该字段指向的 netpoll 实例是用来描述 netpoll 接收的相关信息的，但并不意味着它只描述接收的相关信息，实际上还用于描述发送的相关信息，只是不必设置某些成员，如 rx_hook。

33 struct sk_buff_head arp_tx

存储接收到的 ARP 报文。

34 struct sk_buff_head txq

35 struct delayed_work tx_work

如果 netpoll 输出调用 netpoll_send_skb()没有能成功输出数据包或者设备繁忙，则将待输出报文缓存到 txq 队列中，待 tx_work 工作队列再尝试将它们输出。

为了能实现 netpoll 接收报文功能，还需要实现网络设备的 poll_controller 函数，该函数原型为：void (*poll_controller)(struct net_device *dev)。该函数用来模拟网络设备发生中断，进行中断处理。以 e100 型号网络设备驱动程序为例，poll_controller 函数实现为：

```
2002 static void e100_netpoll(struct net_device *netdev)
2003 {
2004     struct nic *nic = netdev_priv(netdev);
2005
2006     e100_disable_irq(nic);
2007     e100_intr(nic->pdev->irq, netdev);
2008     e100_tx_clean(nic);
2009     e100_enable_irq(nic);
2010 }
```

7.9.2　注册 netpoll 实例

定义了 netpoll 实例配置块之后，必须先注册，之后才能正常地进行输入或输出。netpoll 实例的注册函数为 netpoll_setup()。参数 np 是待注册的 netpoll 实例配置块。

```
615 int netpoll_setup(struct netpoll *np)
616 {
617     struct net_device *ndev = NULL;
618     struct in_device *in_dev;
619     struct netpoll_info *npinfo;
620     unsigned long flags;
621     int err;
```

```
622
623     if (np->dev_name)
624         ndev = dev_get_by_name(np->dev_name);
625     if (!ndev) {
626         printk(KERN_ERR "%s: %s doesn't exist, aborting.\n",
627                 np->name, np->dev_name);
628         return -ENODEV;
629     }
630
631     np->dev = ndev;
```

根据指定的网络名获取对应的网络设备。如果找到指定的网络设备，则将其与 netpoll 实例相关联，否则无法注册 netpoll 实例，打印错误信息后返回错误码。

```
632     if (!ndev->npinfo) {
633         npinfo = kmalloc(sizeof(*npinfo), GFP_KERNEL);
634         if (!npinfo) {
635             err = -ENOMEM;
636             goto release;
637         }
638
639         npinfo->rx_flags = 0;
640         npinfo->rx_np = NULL;
641         spin_lock_init(&npinfo->poll_lock);
642         npinfo->poll_owner = -1;
643
644         spin_lock_init(&npinfo->rx_lock);
645         skb_queue_head_init(&npinfo->arp_tx);
646         skb_queue_head_init(&npinfo->txq);
647         INIT_DELAYED_WORK(&npinfo->tx_work, queue_process);
648
649         atomic_set(&npinfo->refcnt, 1);
650     } else {
651         npinfo = ndev->npinfo;
652         atomic_inc(&npinfo->refcnt);
653     }
```

获取网络设备的 netpoll 信息块。如果该网络设备没有 netpoll 信息块，则需要为其创建，并进行相关的初始化，如接收标志、tx_work 工作队列等。

```
655     if (!ndev->poll_controller) {
656         printk(KERN_ERR "%s: %s doesn't support polling, aborting.\n",
657                 np->name, np->dev_name);
658         err = -ENOTSUPP;
659         goto release;
660     }
```

如果要使用 netpoll，则必须实现网络设备的 poll_controller 接口，否则注册失败。

```
662     if (!netif_running(ndev)) {
663         unsigned long atmost, atleast;
664
665         printk(KERN_INFO "%s: device %s not up yet, forcing it\n",
666                 np->name, np->dev_name);
667
```

```
668         rtnl_lock();
669         err = dev_open(ndev);
670         rtnl_unlock();
671
672         if (err) {
673            printk(KERN_ERR "%s: failed to open %s\n",
674                  np->name, ndev->name);
675            goto release;
676         }
677
678         atleast = jiffies + HZ/10;
679         atmost = jiffies + 4*HZ;
680         while (!netif_carrier_ok(ndev)) {
681            if (time_after(jiffies, atmost)) {
682               printk(KERN_NOTICE
683                     "%s: timeout waiting for carrier\n",
684                     np->name);
685               break;
686            }
687            cond_resched();
688         }
689
690         /* If carrier appears to come up instantly, we don't
691          * trust it and pause so that we don't pump all our
692          * queued console messages into the bitbucket.
693          */
694
695         if (time_before(jiffies, atleast)) {
696            printk(KERN_NOTICE "%s: carrier detect appears"
697                  " untrustworthy, waiting 4 seconds\n",
698                  np->name);
699            msleep(4000);
700         }
701      }
```

662-676 如果网络设备还未启动，则启动之。

678-688 如果网络设备当前不能传输数据包，则需等待，直到网络设备可以传输数据包或者等待时间达到4s为止。

695-700在打开网络设备后的很短时间内（0.1s）就发现网络设备可传输数据包，这是不可信的，需等待4s，以确保网络设备真正可传输数据包。

```
703      if (is_zero_ether_addr(np->local_mac) && ndev->dev_addr)
704         memcpy(np->local_mac, ndev->dev_addr, 6);
705
706      if (!np->local_ip) {
707         rcu_read_lock();
708         in_dev = __in_dev_get_rcu(ndev);
709
710         if (!in_dev || !in_dev->ifa_list) {
711            rcu_read_unlock();
712            printk(KERN_ERR "%s: no IP address for %s, aborting\n",
713                  np->name, np->dev_name);
714            err = -EDESTADDRREQ;
715            goto release;
```

```
716         }
717
718         np->local_ip = ntohl(in_dev->ifa_list->ifa_local);
719         rcu_read_unlock();
720         printk(KERN_INFO "%s: local IP %d.%d.%d.%d\n",
721                np->name, HIPQUAD(np->local_ip));
722     }
723
724     if (np->rx_hook) {
725         spin_lock_irqsave(&npinfo->rx_lock, flags);
726         npinfo->rx_flags |= NETPOLL_RX_ENABLED;
727         npinfo->rx_np = np;
728         spin_unlock_irqrestore(&npinfo->rx_lock, flags);
729     }
```

703-704 如果待注册 netpoll 实例未设置本地网络设备的 MAC 地址，则从网络设备中获取。

706-722 如果待注册 netpoll 实例未设置本地 IP 地址，则从网络设备 IP 配置块中获取。

724-729 如果待注册 netpoll 实例实现了 rx_hook 接口，则将绑定网络设备的 netpoll 信息块标记为 NETPOLL_RX_ENABLED，表示注册的 netpoll 实例可以接收数据包，并设置用于 netpoll 接收的相关信息。

```
731     /* fill up the skb queue */
732     refill_skbs();
733
734     /* last thing to do is link it to the net device structure */
735     ndev->npinfo = npinfo;
736
737     /* avoid racing with NAPI reading npinfo */
738     synchronize_rcu();
739
740     return 0;
```

732 为发送 UDP 和 ARP 应答报文，创建 SKB 的缓存区。发送报文时，如果分配 SKB 失败，则从 SKB 的缓存区中直接获取。

735 将 netpoll 信息块绑定到网络设备上。

738 阻塞 RCU 的写操作，直到 NAPI 中所有读执行单元完成对临界区的访问。

```
742 release:
743     if (!ndev->npinfo)
744         kfree(npinfo);
745     np->dev = NULL;
746     dev_put(ndev);
747     return err;
748 }
```

处理在注册过程中出现的异常，如需释放相关资源等。

7.9.3 netpoll 的输入

netpoll_rx()为 netpoll 输入的入口函数，在 netif_rx()和 netif_receive_skb()中被调用。netpoll 的输入只处理两种数据报，如果是 ARP 报文，则将其添加到 ARP 报文队列 arp_tx 中，等待模拟中断处理。如果是 IP 数据报，则将其输入到 netpoll 实例中，流程图如图 7-5 所示。

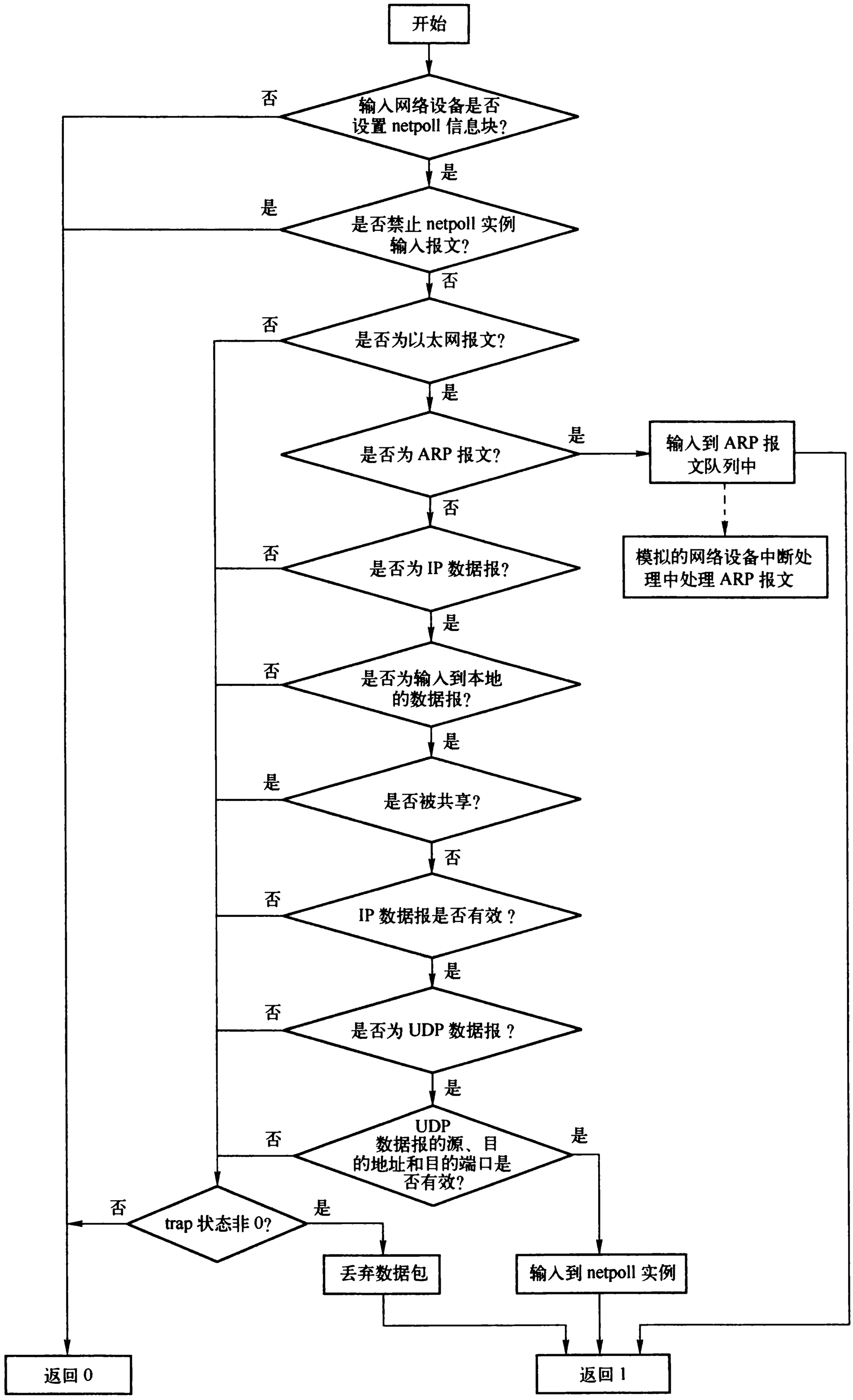

图 7-5　netpoll 输入流程

参数 skb 为输入数据包；返回值见表 7-3。

表 7-3 netpoll_rx()的返回值

返回值	描述
0	netpoll 实例没有接收，需要送到协议栈处理
非 0	netpoll 实例已经处理，无需再送到协议栈

```
49 static inline int netpoll_rx(struct sk_buff *skb)
50 {
51     struct netpoll_info *npinfo = skb->dev->npinfo;
52     unsigned long flags;
53     int ret = 0;
54
55     if (!npinfo || (!npinfo->rx_np && !npinfo->rx_flags))
56         return 0;
57
58     spin_lock_irqsave(&npinfo->rx_lock, flags);
59     /* check rx_flags again with the lock held */
60     if (npinfo->rx_flags && __netpoll_rx(skb))
61         ret = 1;
62     spin_unlock_irqrestore(&npinfo->rx_lock, flags);
63
64     return ret;
65 }
```

55-56 如果输入的网络设备没有设置 netpoll 信息块，或者网络设备的 netpoll 信息块标志为禁止 netpoll 实例输入的报文，则直接返回。

58-64 调用__netpoll_rx()接收数据包，并返回相应的接收状态。

1. trap 状态

在 netpoll 模块中有一个 trapped 全局变量，用来标识 trap 状态，见表 7-4。可以调用 netpoll_set_trap()来设置当前的 trap 状态。

表 7-4 trap 的取值

trap 状态	对应处理
非 0	处理所有的数据包，不满足条件的数据包会丢弃，也不会传递到协议栈
0	只处理满足条件的数据包，不满足条件的数据包仍会传递到协议栈进行处理

2. netpoll_poll()

通过模拟方式触发网络设备中断，然后利用其他 CPU 协助进行数据包轮询输入处理，以及处理接收到的 ARP 报文。

```
153 void netpoll_poll(struct netpoll *np)
154 {
155     if (!np->dev || !netif_running(np->dev) || !np->dev->poll_controller)
156         return;
157
158     /* Process pending work on NIC */
159     np->dev->poll_controller(np->dev);
160     if (np->dev->poll)
161         poll_napi(np);
162
```

```
163     service_arp_queue(np->dev->npinfo);
164
165     zap_completion_queue();
166 }
```

159 调用网络设备的 poll_controller 接口，模拟网络设备因接收到数据包而产生中断。

160-161 如果网络设备实现了 poll 接口，则调用 poll_napi()进行轮询读取报文。

163 处理 arp_tx 队列中的 ARP 报文。

165 释放完成发送的数据包。

```
119 static void poll_napi(struct netpoll *np)
120 {
121     struct netpoll_info *npinfo = np->dev->npinfo;
122     int budget = 16;
123
124     if (test_bit(__LINK_STATE_RX_SCHED, &np->dev->state) &&
125         npinfo->poll_owner != smp_processor_id() &&
126         spin_trylock(&npinfo->poll_lock)) {
127         npinfo->rx_flags |= NETPOLL_RX_DROP;
128         atomic_inc(&trapped);
129
130         np->dev->poll(np->dev, &budget);
131
132         atomic_dec(&trapped);
133         npinfo->rx_flags &= ~NETPOLL_RX_DROP;
134         spin_unlock(&npinfo->poll_lock);
135     }
136 }
```

如果正处于两次数据包输入软中断处理间隙，且当前 CPU 并不是上一次处理数据包输入软中断的 CPU，则此时可调用网络设备 poll 接口进行轮询接收报文。为了确保只在一个 CPU 上处理，需在处理前上锁。

3. ARP 报文处理

如果输入__netpoll_rx()的是 ARP 报文，且 trap 所有的数据包，则将 ARP 报文添加到 arp_tx 队列，等待处理。

```
432 int __netpoll_rx(struct sk_buff *skb)
433 {
...
445     /* check if netpoll clients need ARP */
446     if (skb->protocol == __constant_htons(ETH_P_ARP) &&
447         atomic_read(&trapped)) {
448         skb_queue_tail(&npi->arp_tx, skb);
449         return 1;
450     }
...
506 }
```

在发送过程中会通过模拟方式触发网络设备中断，然后使用其他 CPU 来进行数据包轮询输入处理，最后处理接收到的 ARP 报文。

```
153 void netpoll_poll(struct netpoll *np)
```

```
154 {
...
163     service_arp_queue(np->dev->npinfo);
...
166 }
```

service_arp_queue()会轮询获取并处理指定 netpoll 信息块的 arp_tx 队列中 ARP 报文，直至处理完为止。

```
138 static void service_arp_queue(struct netpoll_info *npi)
139 {
140     struct sk_buff *skb;
141
142     if (unlikely(!npi))
143         return;
144
145     skb = skb_dequeue(&npi->arp_tx);
146
147     while (skb != NULL) {
148         arp_reply(skb);
149         skb = skb_dequeue(&npi->arp_tx);
150     }
151 }
```

arp_reply()才是真正处理 ARP 报文的函数。目前只处理硬件地址类型为以太网，协议类型为 Internet 协议，ARP 操作码为 ARP 请求的 ARP 报文。

```
337 static void arp_reply(struct sk_buff *skb)
338 {
339     struct netpoll_info *npinfo = skb->dev->npinfo;
340     struct arphdr *arp;
341     unsigned char *arp_ptr;
342     int size, type = ARPOP_REPLY, ptype = ETH_P_ARP;
343     __be32 sip, tip;
344     unsigned char *sha;
345     struct sk_buff *send_skb;
346     struct netpoll *np = NULL;
347
348     if (npinfo->rx_np && npinfo->rx_np->dev == skb->dev)
349         np = npinfo->rx_np;
350     if (!np)
351         return;
```

从报文的输入网络设备中获取 netpoll 接收的相关信息。若 netpoll 接收的相关信息无效，则不能应答 ARP 报文。

```
353     /* No arp on this interface */
354     if (skb->dev->flags & IFF_NOARP)
355         return;
356
357     if (!pskb_may_pull(skb, (sizeof(struct arphdr) +
358                 (2 * skb->dev->addr_len) +
359                 (2 * sizeof(u32)))))
360         return;
```

354-355 如果输入 ARP 报文的网络设备不支持 ARP，则直接返回。

357-360 通过检测接收到的 ARP 报文的长度，来确定 ARP 报文是否有效，若无效，则直接返回，不做处理。

```
362     skb->h.raw = skb->nh.raw = skb->data;
363     arp = skb->nh.arph;
364
365     if ((arp->ar_hrd != htons(ARPHRD_ETHER) &&
366          arp->ar_hrd != htons(ARPHRD_IEEE802)) ||
367         arp->ar_pro != htons(ETH_P_IP) ||
368         arp->ar_op != htons(ARPOP_REQUEST))
369         return;
```

365-369 待处理 ARP 报文的硬件地址类型必须为以太网，协议类型必须为 IPv4 协议，ARP 操作码为 ARP 请求。

```
371     arp_ptr = (unsigned char *)(arp+1);
372     /* save the location of the src hw addr */
373     sha = arp_ptr;
374     arp_ptr += skb->dev->addr_len;
375     memcpy(&sip, arp_ptr, 4);
376     arp_ptr += 4;
377     /* if we actually cared about dst hw addr, it would get copied here */
378     arp_ptr += skb->dev->addr_len;
379     memcpy(&tip, arp_ptr, 4);
380
381     /* Should we ignore arp? */
382     if (tip != htonl(np->local_ip) || LOOPBACK(tip) || MULTICAST(tip))
383         return;
```

371-375 获取 ARP 发送方的硬件地址以及 IP 地址等，用来构造 ARP 应答报文。

376-383 检测 ARP 报文的目的地址，以确保必须是 netpoll 实例配置的本地地址、回环地址或组播地址，对其他地址的 ARP 报文不作处理。

```
385     size = sizeof(struct arphdr) + 2 * (skb->dev->addr_len + 4);
386     send_skb = find_skb(np, size + LL_RESERVED_SPACE(np->dev),
387                 LL_RESERVED_SPACE(np->dev));
388
389     if (!send_skb)
390         return;
```

385-390 根据 ARP 请求报文长度分配 ARP 应答报文的 SKB。

```
392     send_skb->nh.raw = send_skb->data;
393     arp = (struct arphdr *) skb_put(send_skb, size);
394     send_skb->dev = skb->dev;
395     send_skb->protocol = htons(ETH_P_ARP);
396
397     /* Fill the device header for the ARP frame */
398
399     if (np->dev->hard_header &&
400         np->dev->hard_header(send_skb, skb->dev, ptype,
401                 sha, np->local_mac,
```

```
402                 send_skb->len) < 0) {
403         kfree_skb(send_skb);
404         return;
405     }
```

设置输出 ARP 报文的网络设备等，然后填充以太网帧首部。

```
407     /*
408      * Fill out the arp protocol part.
409      *
410      * we only support ethernet device type,
411      * which (according to RFC 1390) should always equal 1 (Ethernet).
412      */
413
414     arp->ar_hrd = htons(np->dev->type);
415     arp->ar_pro = htons(ETH_P_IP);
416     arp->ar_hln = np->dev->addr_len;
417     arp->ar_pln = 4;
418     arp->ar_op = htons(type);
419
420     arp_ptr=(unsigned char *)(arp + 1);
421     memcpy(arp_ptr, np->dev->dev_addr, np->dev->addr_len);
422     arp_ptr += np->dev->addr_len;
423     memcpy(arp_ptr, &tip, 4);
424     arp_ptr += 4;
425     memcpy(arp_ptr, sha, np->dev->addr_len);
426     arp_ptr += np->dev->addr_len;
427     memcpy(arp_ptr, &sip, 4);
428
429     netpoll_send_skb(np, send_skb);
430 }
```

最后填充 ARP 报文的各部分内容后调用 netpoll_send_skb()将其输出。

4. IP 数据报的处理

netpoll 输入的入口函数为 netpoll_rx()，在此函数中，经过必要的校验后调用__netpoll_rx()进行输入处理。如果是 ARP 报文，且 trap 所有的数据包，则会将其加到 arp_tx 队列中，等待其他时机处理。对于 IP 数据报，则会检测是不是 netpoll 实例接收的数据包。如果是，才输入到 netpoll 实例中，否则进入协议栈进行处理。

```
432 int __netpoll_rx(struct sk_buff *skb)
433 {
434     int proto, len, ulen;
435     struct iphdr *iph;
436     struct udphdr *uh;
437     struct netpoll_info *npi = skb->dev->npinfo;
438     struct netpoll *np = npi->rx_np;
439
440     if (!np)
441         goto out;
442     if (skb->dev->type != ARPHRD_ETHER)
443         goto out;
```

440-441 如果 netpoll 信息块中不存在 netpoll 接收的相关信息，则不能接收处理报文，根

据情况丢弃或传递到协议栈。

442-443 **netpoll** 只处理以太网帧，如果接收到的不是以太网帧，则同样不作处理，根据情况丢弃或传递到协议栈。

```
445     /* check if netpoll clients need ARP */
446     if (skb->protocol == __constant_htons(ETH_P_ARP) &&
447         atomic_read(&trapped)) {
448         skb_queue_tail(&npi->arp_tx, skb);
449         return 1;
450     }
```

如果接收到的是 ARP 报文，且允许接收 ARP 报文，则将 ARP 报文添加到 arp_tx 队列，等待处理，最后返回 1。

```
452     proto = ntohs(eth_hdr(skb)->h_proto);
453     if (proto != ETH_P_IP)
454         goto out;
455     if (skb->pkt_type == PACKET_OTHERHOST)
456         goto out;
457     if (skb_shared(skb))
458         goto out;
```

接收到的报文有以下三种情况的都不作处理，根据情况丢弃或传递到协议栈。一是非 IP 数据报，二是数据包 MAC 目的地址与报文输入网络设备的 MAC 地址不同，三是数据包被共享。

```
460     iph = (struct iphdr *)skb->data;
461     if (!pskb_may_pull(skb, sizeof(struct iphdr)))
462         goto out;
463     if (iph->ihl < 5 || iph->version != 4)
464         goto out;
465     if (!pskb_may_pull(skb, iph->ihl*4))
466         goto out;
467     if (ip_fast_csum((u8 *)iph, iph->ihl) != 0)
468         goto out;
469
470     len = ntohs(iph->tot_len);
471     if (skb->len < len || len < iph->ihl*4)
472         goto out;
473
474     if (iph->protocol != IPPROTO_UDP)
475         goto out;
476
477     len -= iph->ihl*4;
478     uh = (struct udphdr *)(((char *)iph) + iph->ihl*4);
479     ulen = ntohs(uh->len);
480
481     if (ulen != len)
482         goto out;
483     if (checksum_udp(skb, uh, ulen, iph->saddr, iph->daddr))
484         goto out;
485     if (np->local_ip && np->local_ip != ntohl(iph->daddr))
486         goto out;
487     if (np->remote_ip && np->remote_ip != ntohl(iph->saddr))
488         goto out;
```

```
489     if (np->local_port && np->local_port != ntohs(uh->dest))
490         goto out;
```

460-472 校验 IP 数据报的有效性，凡有以下情况的都不作处理，根据情况丢弃或传递到协议栈。

- 报文长度小于标准 IP 首部长度（不带 IP 选项）。
- IP 首部长度小于 20B，或不是 IP4 版。
- 报文长度小于 IP 首部中标识的首部长度。
- IP 首部中的校验和无效。
- 报文长度小于 IP 首部中标识的报文长度，或 IP 首部中表示的报文长度小于 IP 首部中表示的首部长度。

474-475 netpoll 实例只接收 UDP 数据报，其他报文都不作处理，根据情况丢弃或传递到协议栈。

477-484 如果 UDP 数据报数据长度异常或 UDP 首部中校验和异常，则不作处理，根据情况丢弃或传递到协议栈。

485-490 检测是否为 netpoll 实例待接收的报文。以下情况下不作处理，根据情况丢弃或传递到协议栈。

- 接收报文的目的地址与 netpoll 实例配置的本地地址不符。
- 接收报文的源地址与 netpoll 实例配置的对端地址不符。
- 接收报文的目的端口与 netpoll 实例配置的本地接收端口不符。

```
492     np->rx_hook(np, ntohs(uh->source),
493             (char *)(uh+1),
494             ulen - sizeof(struct udphdr));
495
496     kfree_skb(skb);
497     return 1;
```

通过检测后，最后调用 netpoll 实例的报文接收处理例程进行接收。

```
499 out:
500     if (atomic_read(&trapped)) {
501         kfree_skb(skb);
502         return 1;
503     }
504
505     return 0;
506 }
```

未通过检测的报文，根据 trap 的状态作处理：如果设置了 trap，则将报文丢弃，返回 1；否则返回 0，表示要传递给协议栈处理。

7.9.4 netpoll 的输出

1. netpoll_send_udp()

netpoll_send_udp()是 netpoll 提供的输出接口，在 netpoll 实例中如果需要输出数据，调用该函数即可，前提是需要注册 netpoll 实例控制块。netpoll 输出的函数调用关系如图 7-6 所示。

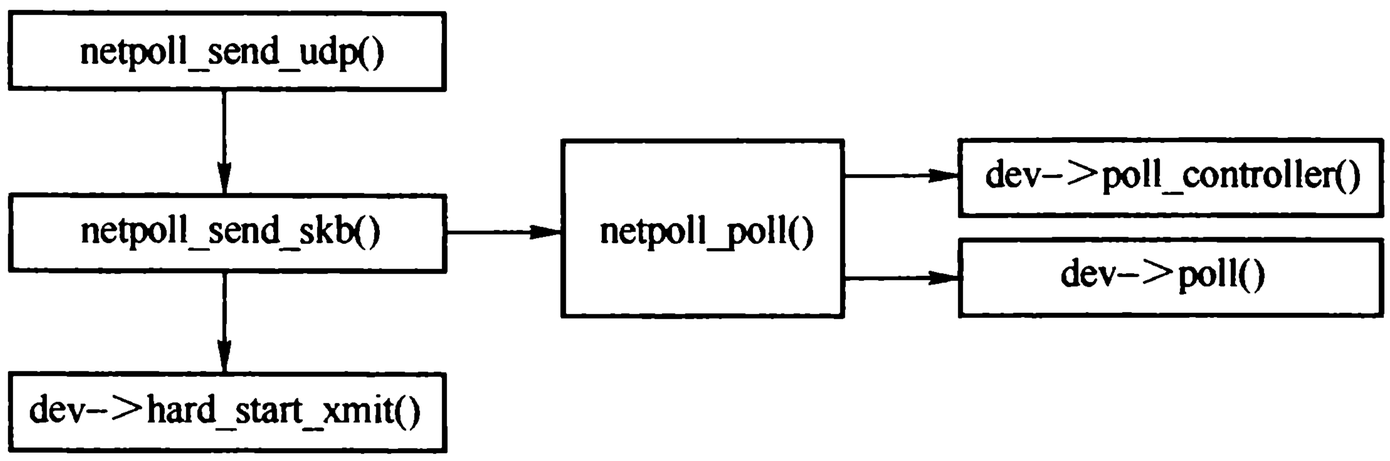

图 7-6　netpoll 输出的函数调用关系

参数说明如下：

- np，已注册的 netpoll 实例控制块。
- msg，待发送的数据。
- len，待发送的数据的长度。

```
280 void netpoll_send_udp(struct netpoll *np, const char *msg, int len)
281 {
282     int total_len, eth_len, ip_len, udp_len;
283     struct sk_buff *skb;
284     struct udphdr *udph;
285     struct iphdr *iph;
286     struct ethhdr *eth;
287
288     udp_len = len + sizeof(*udph);
289     ip_len = eth_len = udp_len + sizeof(*iph);
290     total_len = eth_len + ETH_HLEN + NET_IP_ALIGN;
291
292     skb = find_skb(np, total_len, total_len - len);
293     if (!skb)
294         return;
295
296     memcpy(skb->data, msg, len);
297     skb->len += len;
298
299     skb->h.uh = udph = (struct udphdr *) skb_push(skb, sizeof(*udph));
300     udph->source = htons(np->local_port);
301     udph->dest = htons(np->remote_port);
302     udph->len = htons(udp_len);
303     udph->check = 0;
304     udph->check = csum_tcpudp_magic(htonl(np->local_ip),
305                                     htonl(np->remote_ip),
306                                     udp_len, IPPROTO_UDP,
307                                     csum_partial((unsigned char *)udph, udp_len, 0));
308     if (udph->check == 0)
309         udph->check = CSUM_MANGLED_0;
310
311     skb->nh.iph = iph = (struct iphdr *)skb_push(skb, sizeof(*iph));
312
313     /* iph->version = 4; iph->ihl = 5; */
314     put_unaligned(0x45, (unsigned char *)iph);
315     iph->tos      = 0;
316     put_unaligned(htons(ip_len), &(iph->tot_len));
317     iph->id       = 0;
318     iph->frag_off = 0;
```

```
319     iph->ttl     = 64;
320     iph->protocol = IPPROTO_UDP;
321     iph->check    = 0;
322     put_unaligned(htonl(np->local_ip), &(iph->saddr));
323     put_unaligned(htonl(np->remote_ip), &(iph->daddr));
324     iph->check    = ip_fast_csum((unsigned char *)iph, iph->ihl);
325
326     eth = (struct ethhdr *) skb_push(skb, ETH_HLEN);
327     skb->mac.raw = skb->data;
328     skb->protocol = eth->h_proto = htons(ETH_P_IP);
329     memcpy(eth->h_source, np->local_mac, 6);
330     memcpy(eth->h_dest, np->remote_mac, 6);
331
332     skb->dev = np->dev;
333
334     netpoll_send_skb(np, skb);
335 }
```

288-294 根据待发送数据的长度分配 SKB。

296-297 将待发送的数据复制到 UDP 数据报中。

299-309 设置 UDP 数据报首部中的各个域，包括源、目的端口、长度等。

311-324 设置 IP 数据报首部中的各个域，包括 tos、id 等。

326-330 设置以太网帧首部中的各个域，包括源、目的硬件地址等。

333 设置输出 UDP 数据报的网络设备。

334 完成构建 UDP 数据报后调用 netpoll_send_skb()输出 UDP 数据报。

2. netpoll_send_skb()

netpoll_send_skb()用来发送一个数据包，可以是 UDP 数据报，也可以是 ARP 数据包。

```
236 static void netpoll_send_skb(struct netpoll *np, struct sk_buff *skb)
237 {
238     int status = NETDEV_TX_BUSY;
239     unsigned long tries;
240      struct net_device *dev = np->dev;
241      struct netpoll_info *npinfo = np->dev->npinfo;
242
243      if (!npinfo || !netif_running(dev) || !netif_device_present(dev)) {
244          __kfree_skb(skb);
245          return;
246      }
247
248     /* don't get messages out of order, and no recursion */
249     if (skb_queue_len(&npinfo->txq) == 0 &&
250             npinfo->poll_owner != smp_processor_id()) {
251         unsigned long flags;
252
253         local_irq_save(flags);
254         if (netif_tx_trylock(dev)) {
255             /* try until next clock tick */
256             for (tries = jiffies_to_usecs(1)/USEC_PER_POLL;
257                     tries > 0; --tries) {
258                 if (!netif_queue_stopped(dev))
259                     status = dev->hard_start_xmit(skb, dev);
260
```

```
261                if (status == NETDEV_TX_OK)
262                   break;
263
264                /* tickle device maybe there is some cleanup */
265                netpoll_poll(np);
266
267                udelay(USEC_PER_POLL);
268             }
269             netif_tx_unlock(dev);
270          }
271          local_irq_restore(flags);
272       }
273
274       if (status != NETDEV_TX_OK) {
275          skb_queue_tail(&npinfo->txq, skb);
276          schedule_delayed_work(&npinfo->tx_work,0);
277       }
278    }
```

243-246 如果网络设备尚未创建 netpoll 信息块，或是网络设备尚未打开，或是网络设备被挂起，此时不能输出而只能丢弃待输出数据包。

249-272 如果 txq 队列为空，且当前 CPU 不是信息块中指定的 CPU，说明正处于两次数据包输入软中断间隙，且当前 CPU 并不是上一次处理数据包输入软中断的 CPU，此时可利用当前 CPU 输出数据包或者模拟网络设备中断输入数据包。

调用网络设备的 hard_start_xmit 接口输出数据包时，网络设备必须处于开启状态。如果成功，则完成本次输出，否则会尝试多次，以尽量将数据包成功输出。

274-277 如果没有成功输出数据包或者设备繁忙，则将数据包添加到 txq 队列中，然后重新调度 tx_work 工作队列后再次尝试输出。

7.9.5　tx_work 工作队列

调用 netpoll_send_skb()，在多数情况下能将数据包发送出去，但有时也会由于意外情况或设备繁忙导致没能成功输出数据包，此时将待输出的报文缓存到 tx_work 工作队列中再次尝试输出。tx_work 工作队列在 netpoll_setup()中初始化：

```
INIT_DELAYED_WORK(&npinfo->tx_work, queue_process);
```

queue_process()为 tx_work 工作队列的例程，当 tx_work 工作队列被激活时，即调用该函数。

```
53 static void queue_process(struct work_struct *work)
54 {
55    struct netpoll_info *npinfo =
56       container_of(work, struct netpoll_info, tx_work.work);
57    struct sk_buff *skb;
58    unsigned long flags;
59
60    while ((skb = skb_dequeue(&npinfo->txq))) {
61       struct net_device *dev = skb->dev;
62
63       if (!netif_device_present(dev) || !netif_running(dev)) {
64          __kfree_skb(skb);
65          continue;
66       }
```

```
67
68          local_irq_save(flags);
69          netif_tx_lock(dev);
70          if (netif_queue_stopped(dev) ||
71              dev->hard_start_xmit(skb, dev) != NETDEV_TX_OK) {
72              skb_queue_head(&npinfo->txq, skb);
73              netif_tx_unlock(dev);
74              local_irq_restore(flags);
75
76              schedule_delayed_work(&npinfo->tx_work, HZ/10);
77              return;
78          }
79          netif_tx_unlock(dev);
80          local_irq_restore(flags);
81      }
82  }
```

60 遍历并处理 txq 队列中的报文，直到全部报文已被输出，或网络设备关闭，或输出失败。

63-66 如果输出网络设备正被挂起或未启用，则丢弃输出包。

68-80 输出过程中，如果网络设备关闭或输出失败，则重新调度 tx_work 工作队列，等待下次再尝试输出。

7.9.6 netpoll 实例：netconsole

netconsole 可以将本机的 dmesg 系统信息，通过网络的方式输出到另一台主机上。这样就可以实现远程监控某些主机 dmesg 信息，给开发人员调试内核提供了非常便捷的途径。

netconsole 依赖 netpoll 实现，最终是通过 netpoll 的 netpoll_send_udp()将信息输出到监控主机的。netpoll 并不依赖网络协议栈，因此内核在网络以及 I/O 子系统尚未可用时，依然能发送或接收数据包，只要实现了网络设备驱动程序并满足某些特性即可。

netconsole 提供了一种通过网络监控调试信息的便捷方法，配置也十分简单。使用时确保以模块或静态形式编译进了内核。netconsole 的使用仍有一些限制，目前只能支持 IPv4 协议族的 UDP 协议及以太网络设备。

使用时，监控主机可以使用 syslogd 或 netcat 工具。syslogd 只能使用特定的 UDP 端口 514，而 netcat 工具可以指定任意未被使用的 UDP 端口进行监控。例如，使用 netcat 通过 4098 端口进行接收的 netcat 命令为：

```
# netcat -l -p 4098 -u
```

需在被监控主机上加载运行 netconsole 模块，加载时必须配置相关的端口、IP 地址等信息，命令如下：

```
#modprobe netconsole netconsole=6600@10.58.1.2/eth0,4098@10.58.1.1/
00:1F:29:EA:66:68
```

其中，6600 是本地端口；10.58.1.2/eth0 为本机 IP 地址和网络设备名；4098@10.58.1.1/00:1F:29:EA:66:68 是监控主机端口、IP 地址以及硬件地址。

1. netconsole 的参数

netconsole 参数是在加载内核模块时设置的，因此需要通过 module_param_string 宏获取参数。

```
53 static char config[256];
54 module_param_string(netconsole, config, 256, 0);
```

```
55 MODULE_PARM_DESC(netconsole, " netconsole=[src-port]@[src-ip]/[dev], [tgt-port]
@<tgt-ip>/[tgt-macaddr]\n");
```

53 config 是以字符串形式保存的加载 netconsole 模块时的外部参数。

54 通过 module_param_string 宏，将 netconsole 模块的 netconsole 参数值复制到 config 中。

55 加载模块的参数格式。

```
57  static struct netpoll np = {
58      .name = "netconsole",
59      .dev_name = "eth0",
60      .local_port = 6665,
61      .remote_port = 6666,
62      .remote_mac = {0xff, 0xff, 0xff, 0xff, 0xff, 0xff},
63  };
64  static int configured = 0;
```

定义 netconsole 的默认配置块，包括 netpoll 实例名、绑定的输出网络设备、本地端口以及监控主机接收端口和硬件地址。

```
94  static int option_setup(char *opt)
95  {
96      configured = !netpoll_parse_options(&np, opt);
97      return 1;
98  }
```

调用 netpoll_parse_options()解析模块参数，把得到的参数值设置到 netconsole 的 netpoll 信息块中。

2. console

由于要对 console 的信息进行捕获，因此必须实现一个 console 结构实例并将其注册到系统中。

```
88  static struct console netconsole = {
89      .name = "netcon",
90      .flags = CON_ENABLED | CON_PRINTBUFFER,
91      .write = write_msg
92  };
```

定义了 netconsole 模块的 console 实例，设置了相应的名称以及标志，最重要的是实现了 write 接口。

当信息输出到 console 时，会调用 write 接口 write_msg()。而在 write_msg()中，将待输出的信息，通过 netpoll_send_udp 接口输出到监控主机。

```
68  static void write_msg(struct console *con, const char *msg, unsigned int len)
69  {
70      int frag, left;
71      unsigned long flags;
72
73      if (!np.dev)
74          return;
75
76      local_irq_save(flags);
```

```
77
78      for(left = len; left; ) {
79          frag = min(left, MAX_PRINT_CHUNK);
80          netpoll_send_udp(&np, msg, frag);
81          msg += frag;
82          left -= frag;
83      }
84
85      local_irq_restore(flags);
86  }
```

73-74 如果 netconsole 没有绑定输出网络设备，则不能输出信息。

78-83 如果输出的信息长度大于 MAX_PRINT_CHUNK，则需要分成多个 UDP 数据报进行输出。

3. 初始化

init_netconsole()为 netconsole 模块的初始化函数，加载 netconsole 模块时调用。主要完成获取配置 netpoll 实例以及 console 的注册。

```
102 static int init_netconsole(void)
103 {
104     int err;
105
106     if(strlen(config))
107         option_setup(config);
108
109     if(!configured) {
110         printk("netconsole: not configured, aborting\n");
111         return 0;
112     }
113
114     err = netpoll_setup(&np);
115     if (err)
116         return err;
117
118     register_console(&netconsole);
119     printk(KERN_INFO "netconsole: network logging started\n");
120     return 0;
121 }
```

106-107 如果加载 netconsole 时设置了参数，则解析参数，并将解析得到的值设置到 netpoll 控制块中。

109-112 如果未设置参数或解析参数出错，则因得不到必要的参数而加载失败。

114-116 注册 netpoll 实例。

118-120 注册 netconsole 并返回。

```
123 static void cleanup_netconsole(void)
124 {
125     unregister_console(&netconsole);
126     netpoll_cleanup(&np);
127 }
```

卸载 netconsole 时，需注销 netconsole 和 netpoll 实例。

第 8 章　接口层的输出

第 7 章介绍了每个 CPU 有一个单独的 softnet_data 实例，用来存储与网络软中断处理相关的报文输出和输出队列。在输出过程中会用到 softnet_data 中的 output_queue 和 completion_queue 队列。softnet_data 与接口层和网络层的关系见图 8-1。

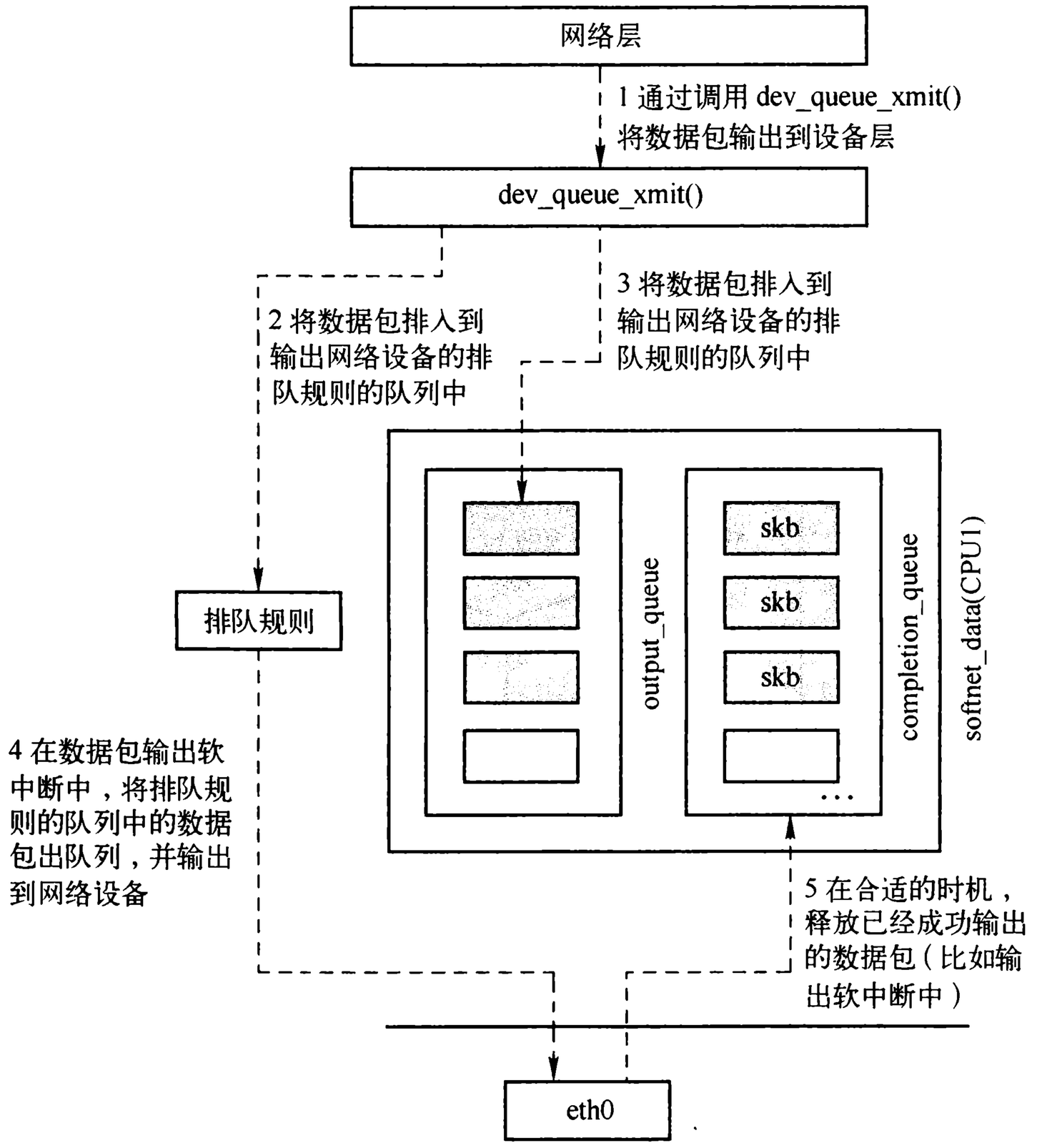

图 8-1　softnet_data 与接口层和网络层的关系

本章论述的网络设备通用接口和 e100 驱动程序涉及以下文件：

- include/linux/netdevice.h，定义网络设备结构、宏等。
- net/core/dev.c，网络设备注册、输入和输出等接口。
- drivers/net/e100.c，e100 驱动程序。

8.1　输出接口

8.1.1　dev_queue_xmit()

dev_queue_xmit()的流程如图 8-2 所示。若支持流量控制，则将待输出的数据包根据规则加入到输出网络设备队列中排队，并在合适的时机激活网络输出软中断，依次将报文从队列中取

出通过网络设备输出。若不支持量控制，则直接将数据包从网络设备输出。

如果提交失败，则会返回相应的错误码，然而返回成功也并不能确保数据包被成功发送，因为有可能由于拥塞而导致流量控制机制将数据包丢弃。

调用 dev_queue_xmit()函数输出数据包，前提是必须启用中断，只有启用中断之后才能激活下半部。

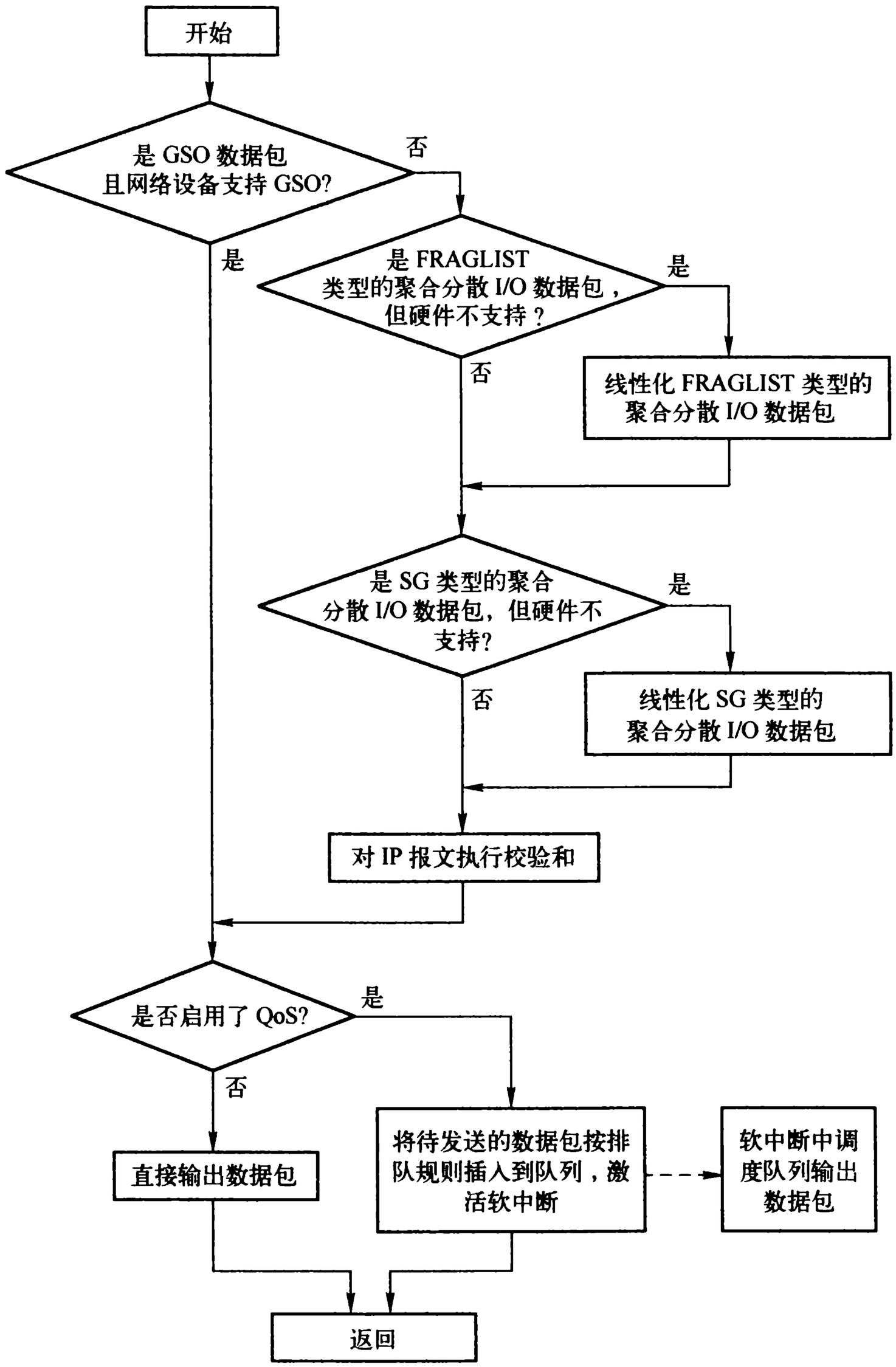

图 8-2 dev_queue_xmit()流程

```
1421 int dev_queue_xmit(struct sk_buff *skb)
1422 {
1423     struct net_device *dev = skb->dev;
1424     struct Qdisc *q;
1425     int rc = -ENOMEM;
1426
1427     /* GSO will handle the following emulations directly. */
```

```
1428    if (netif_needs_gso(dev, skb))
1429        goto gso;
1430
1431    if (skb_shinfo(skb)->frag_list &&
1432        !(dev->features & NETIF_F_FRAGLIST) &&
1433        __skb_linearize(skb))
1434        goto out_kfree_skb;
1435
1436    /* Fragmented skb is linearized if device does not support SG,
1437     * or if at least one of fragments is in highmem and device
1438     * does not support DMA from it.
1439     */
1440    if (skb_shinfo(skb)->nr_frags &&
1441        (!(dev->features & NETIF_F_SG) || illegal_highdma(dev, skb)) &&
1442        __skb_linearize(skb))
1443        goto out_kfree_skb;
1444
1445    /* If packet is not checksummed and device does not support
1446     * checksumming for this protocol, complete checksumming here.
1447     */
1448    if (skb->ip_summed == CHECKSUM_PARTIAL &&
1449        (!(dev->features & NETIF_F_GEN_CSUM) &&
1450         (!(dev->features & NETIF_F_IP_CSUM) ||
1451          skb->protocol != htons(ETH_P_IP))))
1452            if (skb_checksum_help(skb))
1453                goto out_kfree_skb;
```

1428-1429 如果是 GSO 数据包，且网络设备支持 GSO 数据包的处理，则跳转到 gso 标签处对 GSO 数据包直接处理。

1431-1434 对于 FRAGLIST 类型的聚合分散 I/O 数据包，如果输出网络设备不支持 FRAGLIST 类型的聚合分散 I/O（目前只有回环设备支持），则需将其线性化。若线性化失败，则丢弃该数据包，发送失败。

1440-1443 对于 SG 类型的聚合分散 I/O 数据包，如果输出网络设备不支持 SG 类型的聚合分散 I/O，则需将其线性化。如果网络设备不支持在高端内存使用 DMA 但高端内存中有分片，此时也需要将数据包线性化。若线性化失败，则丢弃该数据包，发送失败。

1448-1453 如果待输出的数据包由硬件来执行校验和（尚未执行校验和），但网络设备不支持硬件执行校验和，不支持对 IP 报文执行校验和，则在此处计算校验和。若校验和失败，则丢弃该数据包，发送失败。

```
1455 gso:
1456     spin_lock_prefetch(&dev->queue_lock);
1457
1458     /* Disable soft irqs for various locks below. Also
1459      * stops preemption for RCU.
1460      */
1461     rcu_read_lock_bh();
1462
1463     /* Updates of qdisc are serialized by queue_lock.
1464      * The struct Qdisc which is pointed to by qdisc is now a
1465      * rcu structure - it may be accessed without acquiring
1466      * a lock (but the structure may be stale.) The freeing of the
1467      * qdisc will be deferred until it's known that there are no
```

```
1468     * more references to it.
1469     *
1470     * If the qdisc has an enqueue function, we still need to
1471     * hold the queue_lock before calling it, since queue_lock
1472     * also serializes access to the device queue.
1473     */
1474
1475     q = rcu_dereference(dev->qdisc);
1476 #ifdef CONFIG_NET_CLS_ACT
1477     skb->tc_verd = SET_TC_AT(skb->tc_verd,AT_EGRESS);
1478 #endif
1479     if (q->enqueue) {
1480         /* Grab device queue */
1481         spin_lock(&dev->queue_lock);
1482         q = dev->qdisc;
1483         if (q->enqueue) {
1484             rc = q->enqueue(skb, q);
1485             qdisc_run(dev);
1486             spin_unlock(&dev->queue_lock);
1487
1488             rc = rc == NET_XMIT_BYPASS ? NET_XMIT_SUCCESS : rc;
1489             goto out;
1490         }
1491         spin_unlock(&dev->queue_lock);
1492     }
```

1475 获取输出网络设备的排队规程。rcu_dereference()在 RCU 读临界部分中取出一个 RCU 保护的指针。在需要内存屏障的体系中进行内存屏障，目前只有 Alpha 体系需要。

1476-1478 与包分类器相关，本书不作论述。

1479 如果获取的排队规程定义了“入队”操作，说明启用了 QoS。

1481-1491 将待发送的数据包按排队规则插入到队列，然后进行流量控制，调度队列输出数据包，完成后返回。

```
1506     if (dev->flags & IFF_UP) {
1507         int cpu = smp_processor_id(); /* ok because BHs are off */
1508
1509         if (dev->xmit_lock_owner != cpu) {
1510
1511             HARD_TX_LOCK(dev, cpu);
1512
1513             if (!netif_queue_stopped(dev)) {
1514                 rc = 0;
1515                 if (!dev_hard_start_xmit(skb, dev)) {
1516                     HARD_TX_UNLOCK(dev);
1517                     goto out;
1518                 }
1519             }
1520             HARD_TX_UNLOCK(dev);
1521             if (net_ratelimit())
1522                 printk(KERN_CRIT "Virtual device %s asks to "
1523                        "queue packet!\n", dev->name);
1524         } else {
1525             /* Recursion is detected! It is possible,
1526              * unfortunately */
```

```
1527            if (net_ratelimit())
1528               printk(KERN_CRIT "Dead loop on virtual device "
1529                     "%s, fix it urgently!\n", dev->name);
1530         }
1531      }
1532
1533      rc = -ENETDOWN;
1534      rcu_read_unlock_bh();
1535
1536 out_kfree_skb:
1537      kfree_skb(skb);
1538      return rc;
1539 out:
1540      rcu_read_unlock_bh();
1541      return rc;
1542 }
```

1506 如果设备已打开但未启用 QoS，则直接输出数据包。

1509-1530 HARD_TX_LOCK/HARD_TX_UNLOCK 是一对操作，在这两个操作之间不能再次调用 dev_queue_xmit 接口。因此如果正在用该网络设备发送数据包的 CPU 又调用 dev_queue_xmit()输出数据包，则说明代码有 bug，需输出告警信息。

否则，首先需加锁，以防止其他 CPU 的并发操作，然后在网络设备处于开启状态时，调用 dev_hard_start_xmit()输出数据包到网络设备。

1533-1534 如果网络设备处于关闭状态，则返回相应的错误码。

1536-1538 凡跳转到此处的都是输出数据包时出现错误的，如聚合分散 I/O 数据包线性化失败等，丢弃数据包。

1539-1541 完成数据包输出后，返回相应结果。

8.1.2 dev_hard_start_xmit()

dev_hard_start_xmit()将待输出的数据包提交给网络设备的输出接口，完成数据包的输出。

```
1343 int dev_hard_start_xmit(struct sk_buff *skb, struct net_device *dev)
1344 {
1345      if (likely(!skb->next)) {
1346           if (netdev_nit)
1347                dev_queue_xmit_nit(skb, dev);
1348
1349           if (netif_needs_gso(dev, skb)) {
1350                if (unlikely(dev_gso_segment(skb)))
1351                     goto out_kfree_skb;
1352                if (skb->next)
1353                     goto gso;
1354           }
1355
1356           return dev->hard_start_xmit(skb, dev);
1357      }
```

1345 如果输出是单个数据包(通常情况下都是输出单独数据包)。

1346-1347 如果应用层通过 socket(AF_PACKET，SOCK_RAW, htons(ETH_P_ALL))创建的原始套接口，则需发送一份数据包给这样的套接口。

1349-1356 如果待输出数据包是 GSO 数据包，但网络设备不支持相应的特性，则调用 dev_gso_segment()对 GSO 数据包进行软分割。如果经分割后仍是一个数据包，则直接调用网络设备的 hard_start_xmit 接口输出数据包。然而，通常一个 GSO 数据包经软分割，会生成多个链接起来的数据包，如果是这样的话就需跳转到 gso 标签处，逐个处理数据包。

```
1359 gso:
1360     do {
1361         struct sk_buff *nskb = skb->next;
1362         int rc;
1363
1364         skb->next = nskb->next;
1365         nskb->next = NULL;
1366         rc = dev->hard_start_xmit(nskb, dev);
1367         if (unlikely(rc)) {
1368             nskb->next = skb->next;
1369             skb->next = nskb;
1370             return rc;
1371         }
1372         if (unlikely(netif_queue_stopped(dev) && skb->next))
1373             return NETDEV_TX_BUSY;
1374     } while (skb->next);
1375
1376     skb->destructor = DEV_GSO_CB(skb)->destructor;
1377
1378 out_kfree_skb:
1379     kfree_skb(skb);
1380     return 0;
1381 }
```

1360-1374 当一个 GSO 数据包经过软分割，生成多个链接起来的数据包后，需逐个处理数据包。通过调用网络设备的 hard_start_xmit 接口（e100 网络设备驱动中为 e100_xmit_frame()）输出数据包，如果发生错误，则返回相应错误码。

1376 成功发送了所有的数据包后，需恢复 SKB 原先的析构函数。

1378-1380 如果调用 dev_gso_segment()对 GSO 数据包进行软分割失败，会跳转到此丢弃数据包。

8.1.3 e100 的输出接口：e100_xmit_frame()

e100_xmit_frame()为 e100 网络设备的 hard_start_xmit 接口的实现，最终将数据包输出到硬件。

8.2 网络输出软中断

8.2.1 netif_schedule()

激活数据包输出软中断有多个接口，而 netif_schedule()是最常用的。

```
635 static inline void netif_schedule(struct net_device *dev)
636 {
637     if (!test_bit(__LINK_STATE_XOFF, &dev->state))
638         __netif_schedule(dev);
639 }
```

如果网络设备没有关闭排队功能，则激活网络输出软中断。

```
1103 void __netif_schedule(struct net_device *dev)
1104 {
1105     if (!test_and_set_bit(__LINK_STATE_SCHED, &dev->state)) {
1106         unsigned long flags;
1107         struct softnet_data *sd;
1108
1109         local_irq_save(flags);
1110         sd = &__get_cpu_var(softnet_data);
1111         dev->next_sched = sd->output_queue;
1112         sd->output_queue = dev;
1113         raise_softirq_irqoff(NET_TX_SOFTIRQ);
1114         local_irq_restore(flags);
1115     }
1116 }
```

如果输出网络设备没有处于流量控制的调度中，则将该网络设备链接到 softnet_data 中的 output_queue 队列上，然后激活网络输出软中断对该队列进行处理。

8.2.2　net_tx_action()

net_tx_action()是数据包输出软中断的例程，一旦被激活便会遍历 output_queue 队列中待处理的输出网络设备，然后调用 qdisc_run()在合适的时机发送数据包。数据包输出软中断通常由 netif_schedule()激活。

```
1644 static void net_tx_action(struct softirq_action *h)
1645 {
1646     struct softnet_data *sd = &__get_cpu_var(softnet_data);
1647
1648     if (sd->completion_queue) {
1649         struct sk_buff *clist;
1650
1651         local_irq_disable();
1652         clist = sd->completion_queue;
1653         sd->completion_queue = NULL;
1654         local_irq_enable();
1655
1656         while (clist) {
1657             struct sk_buff *skb = clist;
1658             clist = clist->next;
1659
1660             BUG_TRAP(!atomic_read(&skb->users));
1661             __kfree_skb(skb);
1662         }
1663     }
1664
1665     if (sd->output_queue) {
1666         struct net_device *head;
1667
1668         local_irq_disable();
1669         head = sd->output_queue;
1670         sd->output_queue = NULL;
1671         local_irq_enable();
```

```
1672
1673        while (head) {
1674            struct net_device *dev = head;
1675            head = head->next_sched;
1676
1677            smp_mb__before_clear_bit();
1678            clear_bit(__LINK_STATE_SCHED, &dev->state);
1679
1680            if (spin_trylock(&dev->queue_lock)) {
1681                qdisc_run(dev);
1682                spin_unlock(&dev->queue_lock);
1683            } else {
1684                netif_schedule(dev);
1685            }
1686        }
1687    }
1688 }
```

1648-1663 如果当前 CPU 的 softnet_data 中存在已完成输出待释放的数据包，则遍历 completion_queue 队列，释放该队列中所有数据包。

1665-1687 如果当前 CPU 的 softnet_data 中存在待处理的输出网络设备，则遍历 output_queue 队列，调用 qdisc_run()来发送数据包或者再次调度数据包输出软中断，在合适的时机发送数据包。

8.3 网络设备不支持 GSO 时的处理

如果网络设备不支持 GSO 特性，则在接口层输出 GSO 报文时需由传输层进行软分段，这不是一个简单的过程。首先要得到该报文所属的协议族，根据协议族得到并调用对应的接口，然后再根据传输层协议得到并调用对应的接口，整个过程如图 8-3 所示。

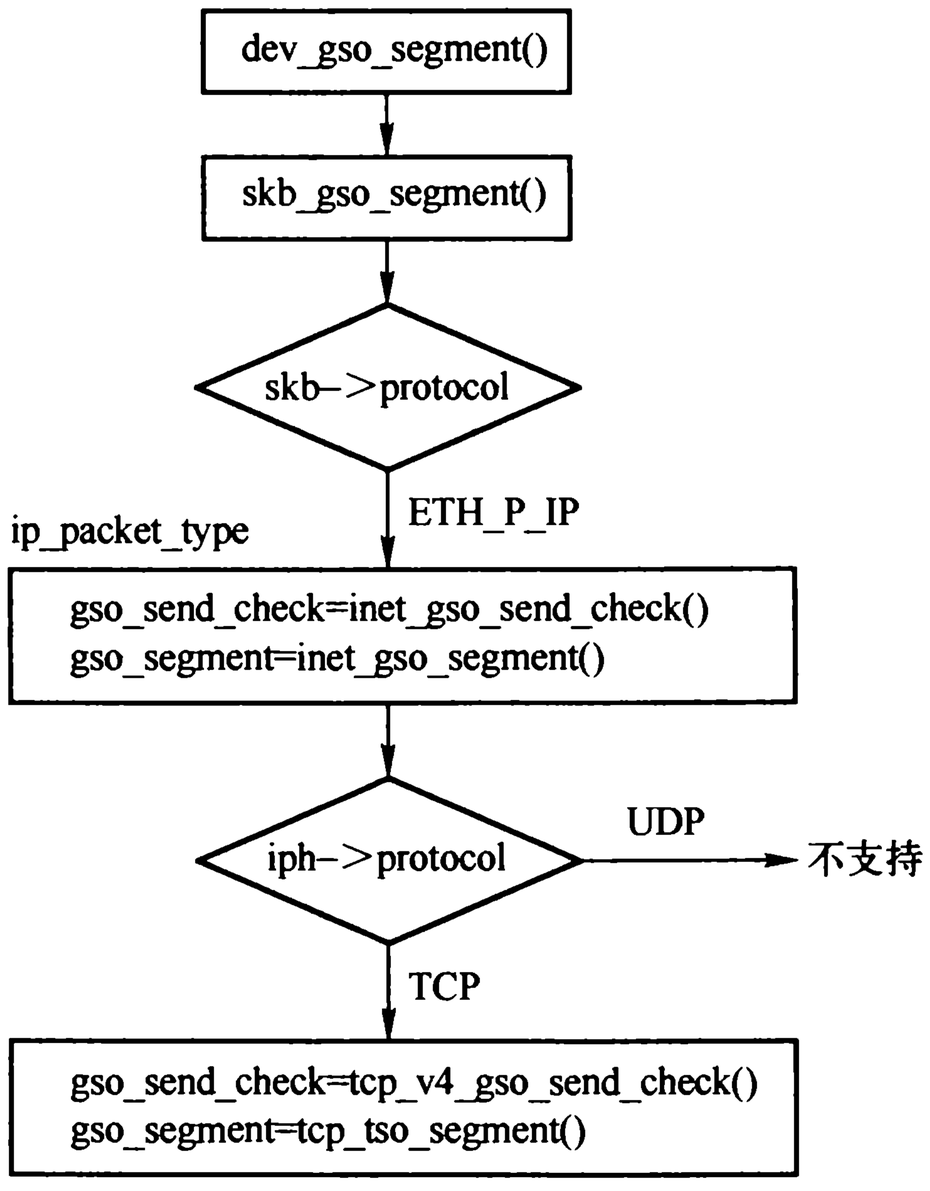

图 8-3 输出时网络设备不支持 GSO 时进行 GSO 分段的调用流程

8.3.1　dev_gso_cb 私有控制块

GSO 段经分段后所得到的段通过 skb->next 链接在一起。当释放 GSO 段的时候，需要将这些链接在一起的段同时释放，为此需要一个特定的分段 GSO 段析构函数—— dev_gso_skb_destructor()。而原先的析构函数需将其保存在 SKB 中作为 GSO 控制块的 dev_gso_cb 结构中。

```
1290 struct dev_gso_cb {
1291     void (*destructor)(struct sk_buff *skb);
1292 };

1294 #define DEV_GSO_CB(skb) ((struct dev_gso_cb *)(skb)->cb)

1296 static void dev_gso_skb_destructor(struct sk_buff *skb)
1297 {
1298     struct dev_gso_cb *cb;
1299
1300     do {
1301         struct sk_buff *nskb = skb->next;
1302
1303         skb->next = nskb->next;
1304         nskb->next = NULL;
1305         kfree_skb(nskb);
1306     } while (skb->next);
1307
1308     cb = DEV_GSO_CB(skb);
1309     if (cb->destructor)
1310         cb->destructor(skb);
1311 }
```

1300-1306 删除并释放除第一个之外的 SKB。

1308-1310 最后调用原先的析构函数释放第一个 SKB。

8.3.2　dev_gso_segment()

dev_gso_segment()通过调用 skb_gso_segment()来分割 GSO 段。

```
1320 static int dev_gso_segment(struct sk_buff *skb)
1321 {
1322     struct net_device *dev = skb->dev;
1323     struct sk_buff *segs;
1324     int features = dev->features & ~(illegal_highdma(dev, skb) ?
1325                      NETIF_F_SG : 0);
1326
1327     segs = skb_gso_segment(skb, features);
1328
1329     /* Verifying header integrity only. */
1330     if (!segs)
1331         return 0;
1332
1333     if (unlikely(IS_ERR(segs)))
1334         return PTR_ERR(segs);
1335
1336     skb->next = segs;
```

```
1337     DEV_GSO_CB(skb)->destructor = skb->destructor;
1338     skb->destructor = dev_gso_skb_destructor;
1339
1340     return 0;
1341 }
```

1324 获取输出网络设备的聚合分散 I/O 特性。

1327-1334 根据输出网络设备的聚合分散 I/O 特性，对段进行软 GSO 分段，分割后得到的段通过 skb->next 链接在一起。

1336-1340 分段成功后，需保存 SKB 原来的析构函数，然后重新设置为特定的分段 GSO 段析构函数 dev_gso_skb_destructor()。

8.3.3 skb_gso_segment()

skb_gso_segment()的作用是分段 GSO 段，返回通过 skb->next 链接在一起的段，如果返回 NULL，则表示 GSO 段没有进行分段。参数说明如下：

- skb，待分割的 GSO 数据包。
- features，输出网络设备支持的 GSO 特性。

```
1214 struct sk_buff *skb_gso_segment(struct sk_buff *skb, int features)
1215 {
1216     struct sk_buff *segs = ERR_PTR(-EPROTONOSUPPORT);
1217     struct packet_type *ptype;
1218     __be16 type = skb->protocol;
1219     int err;
1220
1221     BUG_ON(skb_shinfo(skb)->frag_list);
1222
1223     skb->mac.raw = skb->data;
1224     skb->mac_len = skb->nh.raw - skb->data;
1225     __skb_pull(skb, skb->mac_len);
1226
1227     if (unlikely(skb->ip_summed != CHECKSUM_PARTIAL)) {
1228         if (skb_header_cloned(skb) &&
1229             (err = pskb_expand_head(skb, 0, 0, GFP_ATOMIC)))
1230             return ERR_PTR(err);
1231     }
1232
1233     rcu_read_lock();
1234     list_for_each_entry_rcu(ptype, &ptype_base[ntohs(type) & 15], list) {
1235         if (ptype->type == type && !ptype->dev && ptype->gso_segment) {
1236             if (unlikely(skb->ip_summed != CHECKSUM_PARTIAL)) {
1237                 err = ptype->gso_send_check(skb);
1238                 segs = ERR_PTR(err);
1239                 if (err || skb_gso_ok(skb, features))
1240                     break;
1241                 __skb_push(skb, skb->data - skb->nh.raw);
1242             }
1243             segs = ptype->gso_segment(skb, features);
1244             break;
1245         }
1246     }
1247     rcu_read_unlock();
```

```
1248
1249     __skb_push(skb, skb->data - skb->mac.raw);
1250
1251     return segs;
1252 }
```

1221 在 GSO 软分段之前，先检测一下 frag_list 链表，此时该链表应为空。

1223-1225 在 GSO 软分段之前，先去掉以太网帧首部。

1227-1231 如果待分割的 SKB 是克隆的，则需重新分配 SKB 的线性数据区。

1233-1247 根据输出报文的协议类型查找与之对应的 GSO 接口。如果支持 GSO 接口，则去掉 IP 首部，然后再调用 gso_segment 接口对大段进行分割，返回相应错误码。

1249 无论是否完成 GSO 分段，最终都需重新添加以太网帧首部。

1251 返回相应错误码。

第9章　流 量 控 制

9.1　通过流量控制后输出

Linux 内核网络协议栈在 2.2 版本以前，并未实现对服务质量的支持。所有 IP 数据报的传输都是采用 FIFO（先进先出），尽最大努力传输的处理机制。在早期网络数据量不多的情况下，一旦发生拥塞，路由器就直接丢弃数据包，也并未出现很大的问题。但是随着计算机网络的发展和网络数据量的急剧增长，以及多媒体和 VoIP 应用对延时有着较高的要求，路由器简单丢弃数据包的处理方法已经不再适合这些应用和当前的网络状况了。因此在新的版本中增加了 QoS 功能，实现了服务质量，即针对不同的需求，提供不同服务质量的网络服务功能。虽然 Linux 也支持输入方向上的流量控制，但使用最频繁的却是输出方向上的流量控制，即本章所讲述的内容。

在链路层，每个数据包通过邻居子系统之后，都由 dev_queue_xmit()来进行输出，然后根据输出网络设备的排队规程来确定是否通过 QoS 之后发送，还是直接调用网卡驱动注册的发送函数把数据包发送出去。在启用了 QoS 的情况下，首先将待发送的数据包排入到 QoS 队列中，然后激活数据包输出软中断，最后在数据包输出软中断例程中，从 QoS 队列中获取优先级最高的数据包，调用 dev_hard_start_xmit()将数据包输出到网络设备。流量控制在接口层输出的位置如图 9-1 所示。

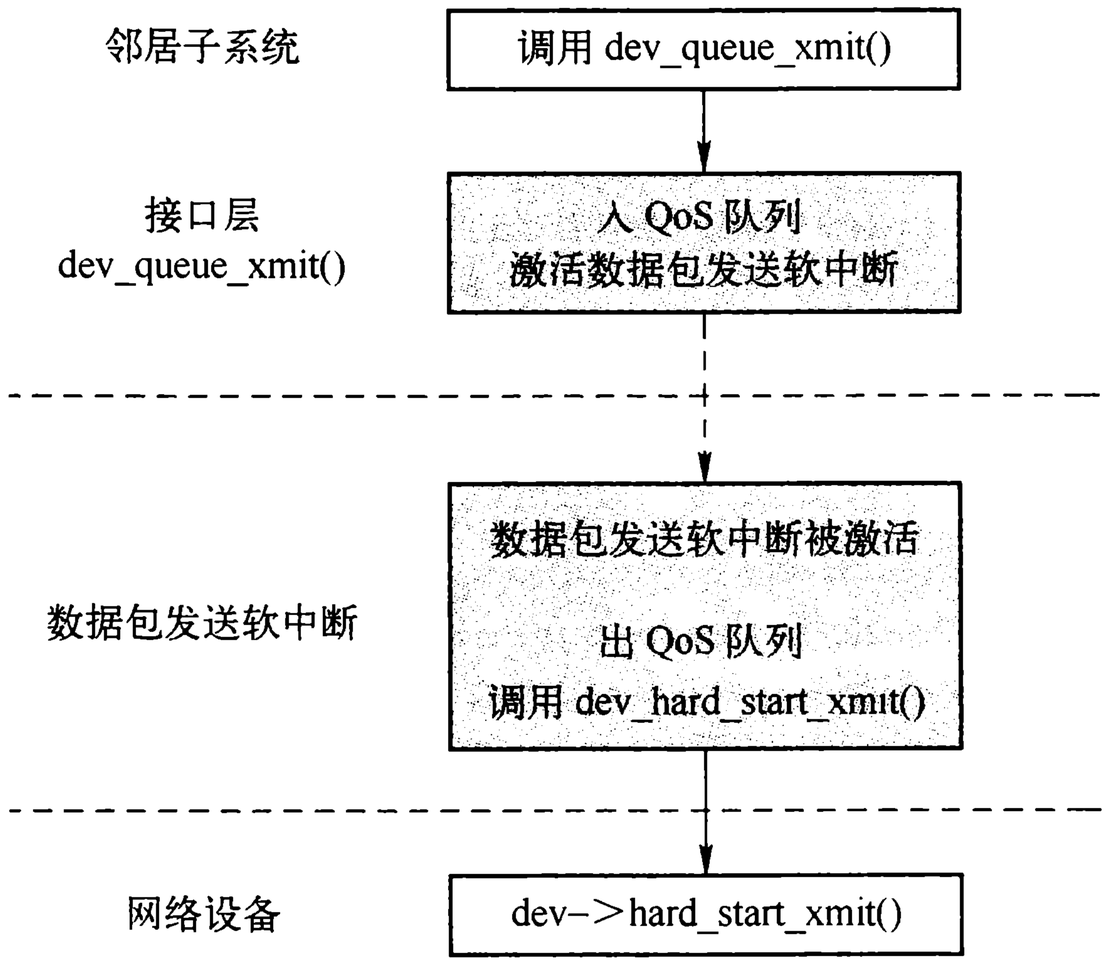

图 9-1　流量控制在接口层输出的位置

本章论述的排队规则、类和过滤器的实现涉及以下文件：

- include/net/pkt_sched.h，定义了操作排队规则的宏等。
- include/net/sch_generic.h，定义排队规则、类和过滤器的结构、宏等。

- net/sched/sch_api.c，操作排队规则的接口函数。
- net/sched/sch_generic.c，基本的数据包调度函数。
- net/core/dev.c，网络设备注册、输入和输出等接口。

9.1.1 dev_queue_xmit()

dev_queue_xmit()首先获取输出网络设备的排队规程，如果获取的排队规程定义了“入队”操作，就说明已启用了 QoS，因此将待发送的数据包按排队规则加入到队列中，然后进行流量控制，调度队列输出数据包。

```
1421 int dev_queue_xmit(struct sk_buff *skb)
1422 {
.... ...
1475    q = rcu_dereference(dev->qdisc);
1476 #ifdef CONFIG_NET_CLS_ACT
1477    skb->tc_verd = SET_TC_AT(skb->tc_verd,AT_EGRESS);
1478 #endif
1479    if (q->enqueue) {
1480       /* Grab device queue */
1481       spin_lock(&dev->queue_lock);
1482       q = dev->qdisc;
1483       if (q->enqueue) {
1484          rc = q->enqueue(skb, q);
1485          qdisc_run(dev);
1486          spin_unlock(&dev->queue_lock);
1487
1488          rc = rc == NET_XMIT_BYPASS ? NET_XMIT_SUCCESS : rc;
1489          goto out;
1490       }
1491       spin_unlock(&dev->queue_lock);
1492    }
.... ...
1542 }
```

在数据包加入队列之后，调用 qdisc_run()来调度数据包输出软中断，在合适的时机发送数据包。真正调度之前，需检测网络设备是否处于启用状态，并且是否处于流量控制调度队列过程中。

```
223 static inline void qdisc_run(struct net_device *dev)
224 {
225    if (!netif_queue_stopped(dev) &&
226       !test_and_set_bit(__LINK_STATE_QDISC_RUNNING, &dev->state))
227       __qdisc_run(dev);
228 }
```

真正实现调度数据包输出软中断的是__qdisc_run()。

```
184 void __qdisc_run(struct net_device *dev)
185 {
186    if (unlikely(dev->qdisc == &noop_qdisc))
187       goto out;
188
189    while (qdisc_restart(dev) < 0 && !netif_queue_stopped(dev))
```

```
190         /* NOTHING */;
191
192 out:
193     clear_bit(__LINK_STATE_QDISC_RUNNING, &dev->state);
194 }
```

186-187 如果输出网络设备的排队规则还处于初始状态，则没必要调度数据包输出软中断，退出流量控制调度队列过程状态。

189-190 调用 qdisc_restart()进行发送，直到发送完毕或网络设备为停止状态为止。

192-193 如果输出网络设备没有设置排队规则，则退出流量控制调度队列过程状态。

9.1.2 qdisc_restart()

qdisc_restart()的主要作用是从排队规则中获取一个可以输出的报文，然后将其输出到网络设备。参数如下所示：

- dev，输出数据包的网络设备。
- 返回值，见表 9-1。

表 9-1 qdisc_restart()的返回值

返回值	描　述
0	表示排队规则的队列已空
大于 0	表示队列中还有数据包，但由于网络设备拥塞导致发送失败而需等待下次发送
小于 0	表示队列中还有数据包，但由于上锁失败或发送成功而需等待下次发送

```
91  static inline int qdisc_restart(struct net_device *dev)
92  {
93      struct Qdisc *q = dev->qdisc;
94      struct sk_buff *skb;
95
96      /* Dequeue packet */
97      if (((skb = dev->gso_skb)) || ((skb = q->dequeue(q)))) {
98          unsigned nolock = (dev->features & NETIF_F_LLTX);
99
100          dev->gso_skb = NULL;
```

97-98 如果还有上次没发送完的 GSO 数据包，则将其取出，否则从排队规则的队列中取出待输出的数据包。一旦取到待输出的数据包，则开始进行输出，同时获取网络设备输出数据包时是否需要上锁的标志。

100 无论输出的是不是缓存的 GSO 数据包，都要将输出设备上用于缓存 GSO 数据包的指针清零。如果本次输出后，还有未发送完的 GSO 数据包，则重新设置该指针。

```
102         /*
103          * When the driver has LLTX set it does its own locking
104          * in start_xmit. No need to add additional overhead by
105          * locking again. These checks are worth it because
106          * even uncongested locks can be quite expensive.
107          * The driver can do trylock like here too, in case
108          * of lock congestion it should return -1 and the packet
109          * will be requeued.
```

```
110         */
111        if (!nolock) {
112           if (!netif_tx_trylock(dev)) {
113           collision:
114              /* So, someone grabbed the driver. */
115
116              /* It may be transient configuration error,
117                 when hard_start_xmit() recurses. We detect
118                 it by checking xmit owner and drop the
119                 packet when deadloop is detected.
120              */
121              if (dev->xmit_lock_owner == smp_processor_id()) {
122                 kfree_skb(skb);
123                 if (net_ratelimit())
124                    printk(KERN_DEBUG"Deadloop on netdevice %s, fix it urgently!\n",
   dev->name);
125                 return -1;
126              }
127              __get_cpu_var(netdev_rx_stat).cpu_collision++;
128              goto requeue;
129           }
130        }
```

111-130 如果输出网络设备设置了输出数据包时需要上锁的标志，则调用 netif_tx_trylock() 上锁。若上锁失败，则需根据原因作相应的处理：如果正在通过该网络设备发送数据包的 CPU 是当前 CPU，则说明代码有 bug，需要提示并释放待输出的数据包；其他情况（如由于其他 CPU 的锁定等），则统计相应数据后，将报文缓存起来或重新排入队列。

```
132        {
133           /* And release queue */
134           spin_unlock(&dev->queue_lock);
135
136           if (!netif_queue_stopped(dev)) {
137              int ret;
138
139              ret = dev_hard_start_xmit(skb, dev);
140              if (ret == NETDEV_TX_OK) {
141                 if (!nolock) {
142                    netif_tx_unlock(dev);
143                 }
144                 spin_lock(&dev->queue_lock);
145                 return -1;
146              }
147              if (ret == NETDEV_TX_LOCKED && nolock) {
148                 spin_lock(&dev->queue_lock);
149                 goto collision;
150              }
151           }
152
153           /* NETDEV_TX_BUSY - we need to requeue */
154           /* Release the driver */
155           if (!nolock) {
156              netif_tx_unlock(dev);
157           }
158           spin_lock(&dev->queue_lock);
```

```
159             q = dev->qdisc;
160         }
161
162         /* Device kicked us out :(
163            This is possible in three cases:
164
165            0. driver is locked
166            1. fastroute is enabled
167            2. device cannot determine busy state
168               before start of transmission (f.e. dialout)
169            3. device is buggy (ppp)
170          */
```

136-151 再次检测网络设备是否关闭了排队功能，如果处于启用状态，则调用dev_hard_start_xmit()将数据包从输出网络设备中输出。输出成功，则返回-1，表示队列中还有数据包等待发送。如果由于输出锁已上锁而发送失败，且输出数据包时无需上锁，则跳转到collision 标签处处理，或者是代码有 bug，或者是由于其他 CPU 锁定而需等待。

155-159 如果由于缓存不够，网络设备硬件错误等原因而关闭排队功能，则获取网络设备当前使用的排队规则。

```
172 requeue:
173         if (skb->next)
174             dev->gso_skb = skb;
175         else
176             q->ops->requeue(skb, q);
177             netif_schedule(dev);
178             return 1;
179     }
180     BUG_ON((int) q->q.qlen < 0);
181     return q->q.qlen;
182 }
```

172-178 将由于其他 CPU 锁定而不能输出的 GSO 数据包缓存到输出网络设备上；而其他数据包则重新排入队列。然后再次调度数据包输出软中断，进行下一次数据包的输出。最后返回 1，表示队列中还有数据包，但由于上锁失败而等待发送。

181 如果出队列失败，说明队列中已没有数据包或者本次没有数据包需要输出，因此直接跳转到此处作处理，返回队列中的数据包数。

9.2 构成流量控制的三种元素

构成流量控制的基本元素有三种，分别是排队规则、类和过滤器。

在启用了流量控制的情况下，每个网络设备至少会配置一个排队规则。待发送报文输出到链路层时，被按规则排入到该排队规则的队列中。发送时，取符合条件的等待发送报文发送，其余的报文在传输之前不得被移除或释放。目前已实现了多种排队规则，包括简单的 FIFO 缓冲和令牌桶等，而精确的排队规则中往往需要管理多个队列。

类定义在排队规则中，与排队规则紧密关联，可通过排队规则的类来区分报文。在排队规则中，可以把报文分配到不同的类中，例如，根据报文的优先级来分类。排队规则可以没有类，也可以有多个类。一个分类的排队规则可以拥有多个类，类内再包含排队规则。

过滤器通常用来将输出报文分配到排队规则的分类中。一个过滤器包含若干的匹配条件，如果符合匹配条件，就按此过滤器进行分类。

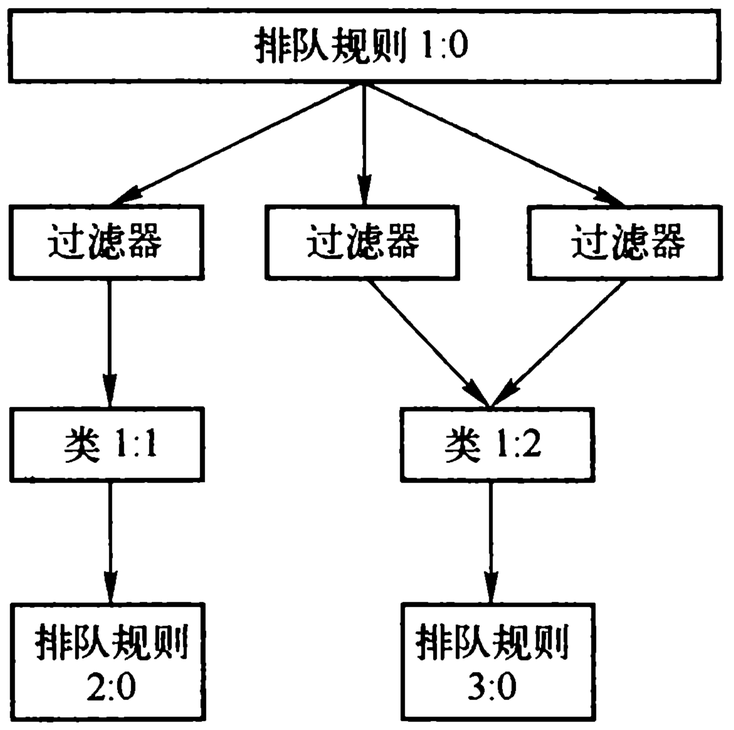

图 9-2 流量控制树

图 9-2 所示为流量控制树的结构。根排队规则对外只提供两个接口，即入队和出队操作。enqueue 接口根据过滤规则检查待输出报文，并由过滤器将报文分配到匹配的类中。如果没有相匹配的过滤规则，则可通过默认过滤器来分配。分类后面还存在其他的排队规则，整个结构就像一棵树，与外界的接口是根排队规则。图中，排队规则 1:0 就是提供给外部接口的根排队规则；根排队规则中存在两个类，类 1:1 和 2:2；排队规则 2:0 和 3:0 则为内部的排队规则，用户可以配置，但不能与之直接操作。通过这种层次结构，使得在流量控制树中可实现多种调度算法，配置相当灵活。

图 9-3 所示为流量控制与接口层输出之间的关联。在调用 dev_queue_xmit()进行输出时，如果支持流量控制，则会调用根排队规则的 enqueue 接口进行入队，而在 qdisc_restart()中出队时，则调用根排队规则的 dequeue 接口。至于入队之后分配到具体的哪个类，进入到哪个子排队规则，则完全由内部实现。

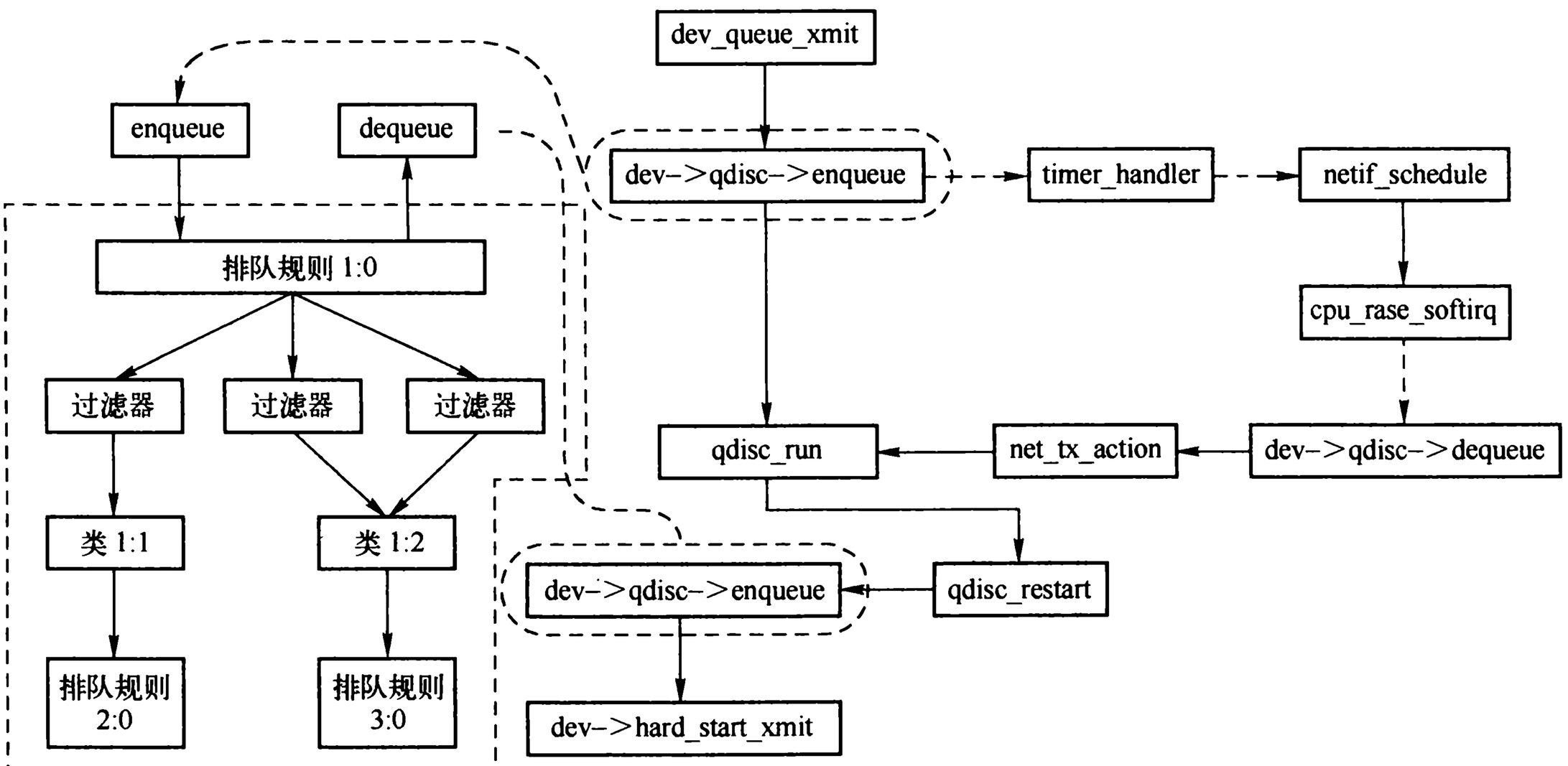

图 9-3 流量控制与接口层输出之间的关联

9.2.1 排队规则

排队规则至少有一个队列，可能比较简单，如 FIFO 排队规则，但也有更复杂的，如令牌桶等。通常，排队规则包括无类和有类两种。无类的排队规则比较简单，内部不包含可配置子类的队列规定；而有类的排队规则，可包含更多的类，其中每个类又可包含一个排队规则，而这个排队规则也可以是分类的或是无类的。因此，对于 FIFO 排队规则，虽然从外部来说它是无类的，但实际上它也算是分类的，只是从用户的角度看，无法使用 tc 工具进行配置。

1. Qdisc 结构

描述排队规则的结构是 Qdisc，如下所示。

```
26 struct Qdisc
27 {
28     int             (*enqueue)(struct sk_buff *skb, struct Qdisc *dev);
29     struct sk_buff *    (*dequeue)(struct Qdisc *dev);
30     unsigned        flags;
31 #define TCQ_F_BUILTIN   1
32 #define TCQ_F_THROTTLED   2
33 #define TCQ_F_INGRESS   4
34     int             padded;
35     struct Qdisc_ops    *ops;
36     u32             handle;
37     u32             parent;
38     atomic_t        refcnt;
39     struct sk_buff_head   q;
40     struct net_device   *dev;
41     struct list_head   list;
42
43     struct gnet_stats_basic   bstats;
44     struct gnet_stats_queue   qstats;
45     struct gnet_stats_rate_est    rate_est;
46     spinlock_t      *stats_lock;
47     struct rcu_head    q_rcu;
48     int             (*reshape_fail)(struct sk_buff *skb,
49                 struct Qdisc *q);
50
51     /* This field is deprecated, but it is still used by CBQ
52      * and it will live until better solution will be invented.
53      */
54     struct Qdisc        *__parent;
55 };
```

```
28 int (*enqueue)(struct sk_buff *skb, struct Qdisc *dev)
```

指向排队规则提供的 enqueue 操作接口，参见 Qdisc_ops 结构的 enqueue。

```
29 struct sk_buff * (*dequeue)(struct Qdisc *dev)
```

指向排队规则提供的 dequeue 操作接口，参见 Qdisc_ops 结构的 dequeue。

```
30 unsigned flags
```

排队规则的标志，见表 9-2。

表 9-2　flags 的取值

flags	描述
TCQ_F_BUILTIN	标识排队规则是空的排队规则，在删除释放时不需要做过多的资源释放
TCQ_F_THROTTLED	标识排队规则正处于由于限制而延时出队的状态中
TCQ_F_INGRESS	标识排队规则是输入排队规则

```
34 int padded
```

由于存储排队规则的内存需 32 字节对齐，而通过动态分配得到的内存起始地址不一定 32 字节对齐，因此需要通过填充将排队规则对齐到 32 字节处。

35 struct Qdisc_ops *ops

排队规则提供的操作接口，参见9.2.1.2节。

36 u32 handle

排队规则、类和过滤器都有一个32位的标识，称为句柄。

排队规则实例的标识分为主编号部分和副编号部分。其中主编号由用户分配，范围从0x0001到0x7FFFF，如果用户指定主编号为0，那么内核将在0x8000和0xFFFF之间分配一个主编号。而除了输入根排队规则编号为FFFF:FFF1和输出根排队规则编号为FFFF:FFFF外，其他排队规则的副编号总为空。需要注意的是，这些编号和设备文件的主编号及副编号没有任何关系。

标识在单个网络设备是唯一的，但在多个网络设备之间可以有重复。

37 u32 parent

父结点的句柄。

38 atomic_t refcnt

引用计数。

39 struct sk_buff_head q

队列中当前数据包数。

40 struct net_device *dev

所属网络设备。

41 struct list_head list

通过链表方式链接到所配置的网络设备上。

43 struct gnet_stats_basic bstats

记录入队报文总字节数和入队报文总数。

44 struct gnet_stats_queue qstats

记录队列相关的统计数据，包括当前队列中报文数、报文总长度，丢弃报文数、重新入队报文数以及由于限制而延时出队次数。

45 struct gnet_stats_rate_est rate_est

队列当前的速率，包括以字节和报文数为单位两种。

46 spinlock_t *stats_lock

信息统计操作自旋锁，防止多CPU并发，同net_device结构中的queue_lock。

47 struct rcu_head q_rcu

通过本字段在没有对象再使用该排队规则时释放该排队规则。

48 int (*reshape_fail)(struct sk_buff *skb, struct Qdisc *q)

用于实现更复杂的流量控制机制，很少排队规则会实现此接口。当一个外部队列向内部队列传递报文时，可能出现报文必须被丢弃的情况，如当没有可用缓冲区时。如果外部排队规则实现了该回调函数，那么这时就可以被内部排队规则调用。

54 struct Qdisc *__parent

本字段已过时，但目前仍在CBQ排队规则中被使用，在有更好的办法解决之前，仍将存在。

2. Qdisc_ops结构

Qdisc_ops结构用来描述队列操作的接口，每个排队规则都必须要实现该接口。

```
86  struct Qdisc_ops
87  {
88      struct Qdisc_ops    *next;
89      struct Qdisc_class_ops    *cl_ops;
90      char            id[IFNAMSIZ];
91      int             priv_size;
92
93      int             (*enqueue)(struct sk_buff *, struct Qdisc *);
94      struct sk_buff *    (*dequeue)(struct Qdisc *);
95      int             (*requeue)(struct sk_buff *, struct Qdisc *);
96      unsigned int        (*drop)(struct Qdisc *);
97
98      int             (*init)(struct Qdisc *, struct rtattr *arg);
99      void            (*reset)(struct Qdisc *);
100     void            (*destroy)(struct Qdisc *);
101     int             (*change)(struct Qdisc *, struct rtattr *arg);
102
103     int             (*dump)(struct Qdisc *, struct sk_buff *);
104     int             (*dump_stats)(struct Qdisc *, struct gnet_dump *);
105
106     struct module       *owner;
107 };
```

88 struct Qdisc_ops *next

用于链接已注册的各种排队规则的操作接口。

89 struct Qdisc_class_ops *cl_ops

所在规则提供的类操作接口。

90 char id[IFNAMSIZ]

内部使用的标识符，通常是排队规则名。

91 int priv_size

附属在排队规则上的私有信息块大小。该信息块通常与排队规则一起分配内存，紧跟在排队规则之后，可由 qdisc_priv()获取。

93 int (*enqueue)(struct sk_buff *, struct Qdisc *)

将待输出数据包加入到排队规则中的函数，参数 skb 为待入队数据包，dev 为排队规则。返回值见表 9-3。

表 9-3 enqueue()的返回值

返回值	描述
NET_XMIT_SUCCESS	报文被排队规则接受，成功入队
NET_XMIT_DROP	报文入队失败，被丢弃
NET_XMIT_CN	报文入队失败，由于拥塞而被丢弃，比如由于缓冲区溢出
NET_XMIT_POLICED	报文入队失败，由于限制机制检测到违背了某条规则而被丢弃，比如超出了允许的速率
NET_XMIT_BYPASS	报文被排队规则接受，成功入队。但是它将通不过正常的 dequeue()离开排队规则

94 struct sk_buff *(*dequeue)(struct Qdisc *)

数据包从指定的排队规则队列中出队函数，返回值指向下一个可能被发送的报文。当返回值为 NULL 时，有可能排队规则队列中已不再有等待的报文，或者是没有预备好发送的数据包。

当一个排队规则存在多个队列时，等待报文总数 Qdisc->q.qlen 必须是有效的。

```
95 int (*requeue)(struct sk_buff *, struct Qdisc *)
```

将先前出队的报文重新排入到队列中的函数。不同于 enqueue()的是，重新入队的报文需被放置到它出队前在排队规则队列中所处的位置上。该接口通常用于报文要发送而由 dequeue()出队列后，因某个不可预见的原因最终又未能发送的情况。

```
96 unsigned int (*drop)(struct Qdisc *)
```

从队列中移除并丢弃一个报文的函数。

```
98 int (*init)(struct Qdisc *, struct rtattr *arg)
```

初始化新实例化的排队规则的函数。

```
99 void (*reset)(struct Qdisc *)
```

初始化排队规则函数，需完成清空队列，重置计数器，删除定时器等。如果所属排队规则内部还有其他的排队规则，那么它们的 reset()也会被递归调用。

```
100 void (*destroy)(struct Qdisc *)
```

用于释放排队规则在初始化和运行时申请资源的函数。

```
101 int (*change)(struct Qdisc *, struct rtattr *arg)
```

用来改变排队规则参数的函数。

```
103 int (*dump)(struct Qdisc *, struct sk_buff *)
104 int (*dump_stats)(struct Qdisc *, struct gnet_dump *)
```

用于输出排队规则的配置参数和统计数据的函数。

3. 排队规则相关函数

（1）dev_init_scheduler()

初始化排队规则的相关数据，在注册网络设备的 register_netdevice()中被调用。

```
582 void dev_init_scheduler(struct net_device *dev)
583 {
584     qdisc_lock_tree(dev);
585     dev->qdisc = &noop_qdisc;
586     dev->qdisc_sleeping = &noop_qdisc;
587     INIT_LIST_HEAD(&dev->qdisc_list);
588     qdisc_unlock_tree(dev);
589
590     dev_watchdog_init(dev);
591 }
```

584-588 初始化排队规则的相关数据。排队规则为 noop_qdisc，对应的 noop_enqueue()，noop_dequeue()，noop_requeue()中，并没有对数据包进行任何的分类或者排队，而是直接丢弃任何待发送的报文。所以此时网络设备还不能发送任何数据包，必须等网络设备启用之后才能发送数据包。

590 初始化网络设备的软件狗。

（2）pktsched_init()

pktsched_init()由宏 subsys_initcall 添加到初始化列表中，在系统初始化时调用。主要用于初始化 RT_NETLINK 接口，并且调用 register_qdisc()注册 bfifo 和 pfifo 排队规则。

（3）register_qdisc()

将新的排队规则注册到系统中，即链接到 qdisc_base 链表上。同一种排队规则只能注册一次，因此在链接到 qdisc_base 链表之前，必须先检查是否已经存在相同标识符的排队规则。通常在排队规则的模块初始化时被调用，而默认的排队规则则在 pktsched_init()中注册。

（4）unregister_qdisc()

将指定的排队规则从系统中注销，即从 qdisc_base 链表上删除。如果待注销的排队规则尚未注册，则不作任何操作，返回相应的错误码。

（5）qdisc_lookup_ops()

根据排队规则名得到在系统已注册的排队规则。主要在创建排队规则的 qdisc_create()中被调用。

（6）qdisc_alloc()

分配内存建立排队规则，并初始化排队规则的相关成员，然后将排队规则与排队规则的操作接口绑定起来，设置排队规则的 enqueue 和 dequeue 接口以及排队规则安装的网络设备，最后返回已建立的排队规则。该函数在 qdisc_create()和 qdisc_create_dflt()中被调用。

（7）qdisc_create_dflt()

用于创建默认的排队规则。首先通过 qdisc_alloc()建立起排队规则并与操作接口绑定，然后设置新建排队规则的父排队规则，最后调用 init 接口初始化。

（8）qdisc_create()

用于创建指定的排队规则。首先根据指定的排队规则标识符在已注册的排队规则中查找是否已存在对应的排队规则，如果不存在，则根据标识符动态加载对应排队规则的模块，完成后会再次尝试查找，如果再次失败，则说明创建排队规则失败。然后调用 qdisc_alloc()新建排队规则并与操作接口绑定。接着设置排队规则的句柄，如果用户提供的句柄为空，则调用 qdisc_alloc_handle()分配一个可用的句柄。最后调用 init 接口初始化，并将其记录到绑定网络设备的 qdisc_list 链表上。

（9）qdisc_lookup()

根据排队规则句柄在配置在网络设备上的排队规则中查找对应的排队规则。

（10）dev_activate()

初始化用于流量控制的排队规则，并启动监视定时器。

```
514 void dev_activate(struct net_device *dev)
515 {
516     /* No queueing discipline is attached to device;
517      create default one i.e. pfifo_fast for devices,
518      which need queueing and noqueue_qdisc for
519      virtual interfaces
520  */
521
522     if (dev->qdisc_sleeping == &noop_qdisc) {
523         struct Qdisc *qdisc;
524         if (dev->tx_queue_len) {
525             qdisc = qdisc_create_dflt(dev, &pfifo_fast_ops,
526                       TC_H_ROOT);
527             if (qdisc == NULL) {
528                 printk(KERN_INFO "%s: activation failed\n", dev->name);
529                 return;
530             }
```

```
531             write_lock(&qdisc_tree_lock);
532             list_add_tail(&qdisc->list, &dev->qdisc_list);
533             write_unlock(&qdisc_tree_lock);
534         } else {
535             qdisc =  &noqueue_qdisc;
536         }
537         write_lock(&qdisc_tree_lock);
538         dev->qdisc_sleeping = qdisc;
539         write_unlock(&qdisc_tree_lock);
540     }
541
542     if (!netif_carrier_ok(dev))
543         /* Delay activation until next carrier-on event */
544         return;
545
546     spin_lock_bh(&dev->queue_lock);
547     rcu_assign_pointer(dev->qdisc, dev->qdisc_sleeping);
548     if (dev->qdisc != &noqueue_qdisc) {
549         dev->trans_start = jiffies;
550         dev_watchdog_up(dev);
551     }
552     spin_unlock_bh(&dev->queue_lock);
553 }
```

522-540 如果网络设备当前尚未安装有效的排队规则，则需安装默认的排队规则。默认的排队规则有两种，根据网络设备支持的发送队列最大帧数来设置。网络设备支持排队时，则安装默认排队规则 pfifo；不支持排队时，则安装默认排队规则 noqueue。noqueue 是不支持队列的排队规则，因为它不支持 enqueue 接口。当通过 dev_queue_xmit()输出报文时，发现不支持 enqueue 接口，就直接发送报文，而不再进行排队操作。

542-544 检测网络设备是否处于可传递数据包状态。如果不是，则延迟应用已安装的排队规则，直到转变为可传递数据包状态。

546-554 如果处于可传递数据包状态，则应用已安装的排队规则，并打开网络设备的软件狗，监视报文发送是否超时。

（11）dev_deactivate()

将网络设备的排队规则设置为空规则，停止发送报文及监视报文发送是否超时。

```
555 void dev_deactivate(struct net_device *dev)
556 {
557     struct Qdisc *qdisc;
558
559     spin_lock_bh(&dev->queue_lock);
560     qdisc = dev->qdisc;
561     dev->qdisc = &noop_qdisc;
562
563     qdisc_reset(qdisc);
564
565     spin_unlock_bh(&dev->queue_lock);
566
567     dev_watchdog_down(dev);
568
```

```
569     /* Wait for outstanding dev_queue_xmit calls. */
570     synchronize_rcu();
571
572     /* Wait for outstanding qdisc_run calls. */
573     while (test_bit(__LINK_STATE_QDISC_RUNNING, &dev->state))
574         yield();
575
576     if (dev->gso_skb) {
577         kfree_skb(dev->gso_skb);
578         dev->gso_skb = NULL;
579     }
580 }
```

559-565 将网络设备的排队规则设置为空规则，并重新初始化排队规则。

567 关闭网络设备的软件狗，停止监视报文发送是否超时。

570 暂时让出处理器而阻塞，唤醒 dev_queue_xmit()的执行进程，完成 dev_queue_xmit()的调用。

573-574 如果网络设备还在进行流量控制的调度队列过程中，尽量让出处理器，尽可能快地完成 qdisc_run()的调用。

576-579 最后，如果网络设备上还有缓存 GSO 报文，则将其释放。

9.2.2 类

1. xxx_class 结构

类可以看作逻辑上独立的元素，与排队规则的关系是很紧密的，并且每个类至少被绑定到一个过滤器。由于其实现的方式，不可能采用通用的结构来描述类。通过过滤器可以将报文分配到排队规则的不同类中。通常类结构定义如以下代码所示。实际上各种类的实现存在着巨大的差别，但每个类都有一个与排队规则相似的唯一类标识符 classid，用于标识一种类。类的主编号对应于关联的排队规则，而辅编号指定该排队规则的类，辅编号的范围从 0x0 到 0xFFFF，并只在一个排队规则的所有类中是唯一的。

```
struct xxx_class
{
    u32             classid;
    ......
    struct tcf_proto    *filter_list;
    ......
};
```

2. Qdisc_class_ops 结构

Qdisc_class_ops 结构是用于类操作的接口，排队规则实现了分类就必须实现该接口。

```
57 struct Qdisc_class_ops
58 {
59     /* Child qdisc manipulation */
60     int             (*graft)(struct Qdisc *, unsigned long cl,
61                     struct Qdisc *, struct Qdisc **);
62     struct Qdisc *          (*leaf)(struct Qdisc *, unsigned long cl);
63     void             (*qlen_notify)(struct Qdisc *, unsigned long);
```

```
64
65    /* Class manipulation routines */
66    unsigned long       (*get)(struct Qdisc *, u32 classid);
67    void                (*put)(struct Qdisc *, unsigned long);
68    int                 (*change)(struct Qdisc *, u32, u32,
69                        struct rtattr **, unsigned long *);
70    int                 (*delete)(struct Qdisc *, unsigned long);
71    void                (*walk)(struct Qdisc *, struct qdisc_walker * arg);
72
73    /* Filter manipulation */
74    struct tcf_proto **   (*tcf_chain)(struct Qdisc *, unsigned long);
75    unsigned long       (*bind_tcf)(struct Qdisc *, unsigned long,
76                        u32 classid);
77    void                (*unbind_tcf)(struct Qdisc *, unsigned long);
78
79    /* rtnetlink specific */
80    int                 (*dump)(struct Qdisc *, unsigned long,
81                        struct sk_buff *skb, struct tcmsg*);
82    int                 (*dump_stats)(struct Qdisc *, unsigned long,
83                        struct gnet_dump *);
84 };
```

60 int (*graft)(struct Qdisc *,unsigned long cl,struct Qdisc *, struct Qdisc **)

用于将一个排队规则绑定到一个类，并返回先前绑定到这个类的排队规则。

62 struct Qdisc * (*leaf)(struct Qdisc *, unsigned long cl)

获取当前绑定到所在类的排队规则。

63 void (*qlen_notify)(struct Qdisc *, unsigned long)

用于响应队列长度变化。

66 unsigned long (*get)(struct Qdisc *, u32 classid)

根据给定的类标识符从排队规则中查找对应的类，并引用该类，该类的引用计数递增1。

67 void (*put)(struct Qdisc *, unsigned long)

递减指定类的引用计数。如果引用计数为0，则删除释放此类。

68 int(*change)(struct Qdisc *,u32, u32,struct rtattr**,unsigned long *)

用于变更指定类的参数，如果该类不存在则新建之。

70 int (*delete)(struct Qdisc *, unsigned long)

用于删除并释放指定的类。首先会递减该类的引用计数，然后如果引用计数递减后为0，删除释放之。

71 void (*walk)(struct Qdisc *, struct qdisc_walker * arg)

遍历一个排队规则的所有类，取回实现了回调函数类的配置数据及统计信息。

74 struct tcf_proto ** (*tcf_chain)(struct Qdisc *, unsigned long)

获取绑定到该类的过滤器所在链表的首结点。

75 unsigned long (*bind_tcf)(struct Qdisc *, unsigned long, u32 classid)

在一个过滤器正准备绑定到指定的类之前被调用，通过类标识获取类，首先递增类引用计数，然后是一些其他的检查。

77 void (*unbind_tcf)(struct Qdisc *, unsigned long)

在过滤器完成绑定到指定的类后被调用，递减类引用计数。

```
80 int (*dump)(struct Qdisc *, unsigned long, struct sk_buff *skb, struct tcmsg*)
82 int (*dump_stats)(struct Qdisc *, unsigned long, struct gnet_dump *)
```

用于输出类的配置参数和统计数据。

3. 类相关函数

支持类的排队规则提供了很多函数，包括将队列绑定到类的函数和改变或输出类信息的函数。

（1）qdisc_graft()

用来将一个排队规则绑定到一个类，并返回先前绑定到该类的排队规则。调用该函数时，通常需要绑定的排队规则支持分类，否则调用会失败。

先检查是否存在一个父结点或者该排队规则本身是不是流量控制树的根。如果存在父结点，那么先将类标识符映射到内部标识符，接着调用 graft()将排队规则绑定到类中。如果是流量控制树的根，调用 dev_graft_qdisc()绑定根排队规则和类。

（2）dev_graft_qdisc()

为网络设备安装指定的根排队规则。

```
314 static struct Qdisc *
315 dev_graft_qdisc(struct net_device *dev, struct Qdisc *qdisc)
316 {
317     struct Qdisc *oqdisc;
318
319     if (dev->flags & IFF_UP)
320         dev_deactivate(dev);
321
322     qdisc_lock_tree(dev);
323     if (qdisc && qdisc->flags&TCQ_F_INGRESS) {
324         oqdisc = dev->qdisc_ingress;
325         /* Prune old scheduler */
326         if (oqdisc && atomic_read(&oqdisc->refcnt) <= 1) {
327             /* delete */
328             qdisc_reset(oqdisc);
329             dev->qdisc_ingress = NULL;
330         } else {  /* new */
331             dev->qdisc_ingress = qdisc;
332         }
333
334     } else {
335
336         oqdisc = dev->qdisc_sleeping;
337
338         /* Prune old scheduler */
339         if (oqdisc && atomic_read(&oqdisc->refcnt) <= 1)
340             qdisc_reset(oqdisc);
341
342         /* ... and graft new one */
343         if (qdisc == NULL)
344             qdisc = &noop_qdisc;
345         dev->qdisc_sleeping = qdisc;
346         dev->qdisc = &noop_qdisc;
347     }
```

```
348
349     qdisc_unlock_tree(dev);
350
351     if (dev->flags & IFF_UP)
352         dev_activate(dev);
353
354     return oqdisc;
355 }
```

319-320 如果网络设备处于启用状态，则将其排队规则设置为空规则，停止发送报文。

323-334 如果安装的是输入排队规则，则初始化当前的输入排队规则，然后安装新的输入排队规则。

336-347 如果安装的是输出排队规则，则初始化当前的输出排队规则，然后安装新的输出排队规则，并将当前应用的排队规则设置为空规则。

351-352 在网络设备启用状态下，应用刚安装的排队规则。

354 返回原先的排队规则。

（3）qdisc_leaf()

根据父结点和类标识符，获取当前绑定到所在类的排队规则。

9.2.3 过滤器

当报文被送到一个具有多个类的排队规则中时，排队规则将调用 tc_classify()。该函数检查过滤器是否接受 skb->protcol 所指定的协议，然后调用过滤器的 classify()做出决定是否接受或分到哪个类中。

1. tcf_proto 结构

过滤器在逻辑上是独立于类的，在排队规则中入队的报文其所属于的类是由过滤器决定的。通过过滤器，将入队的报文根据条件分配到符合条件的类中。描述流量控制过滤器的结构是 tcf_proto。

```
141 struct tcf_proto
142 {
143     /* Fast access part */
144     struct tcf_proto    *next;
145     void            *root;
146     int             (*classify)(struct sk_buff*, struct tcf_proto*,
147                     struct tcf_result *);
148     __be16          protocol;
149
150     /* All the rest */
151     u32             prio;
152     u32             classid;
153     struct Qdisc        *q;
154     void            *data;
155     struct tcf_proto_ops    *ops;
156 };
```

144 struct tcf_proto *next

用来将多个过滤器链接在一起形成一个链表。

```
145 void *root
```

```
154 void *data
```

存储与特定过滤器特性相关的数据。

```
146 int(*classify)(struct sk_buff*,struct tcf_proto*,struct tcf_result *)
```

报文分类函数，参见 tcf_proto_ops 的 classify。

```
148 __be16 protocol
```

进行报文过滤的网络层协议号，用得最多的是 IP 协议，即 ETH_P_IP。

```
151 u32 prio
```

优先级，用来对应用于同一协议的过滤器进行排序。按 prio 的从小到大次序遍历过滤器，报文将被分配到第一个具有匹配规则的过滤器中。

```
152 u32 classid
```

父排队规则的类标识符。

```
153 struct Qdisc *q
```

父排队规则。

```
155 struct tcf_proto_ops *ops
```

过滤器对应的操作接口。

2. tcf_proto_ops 结构

tcf_proto_ops 是用来描述过滤器的结构，如果排队规则实现了分类，则必须实现过滤器用来分类。

```
116 struct tcf_proto_ops
117 {
118     struct tcf_proto_ops    *next;
119     char                    kind[IFNAMSIZ];
120
121     int                     (*classify)(struct sk_buff*, struct tcf_proto*,
122                             struct tcf_result *);
123     int                     (*init)(struct tcf_proto*);
124     void                    (*destroy)(struct tcf_proto*);
125
126     unsigned long           (*get)(struct tcf_proto*, u32 handle);
127     void                    (*put)(struct tcf_proto*, unsigned long);
128     int                     (*change)(struct tcf_proto*, unsigned long,
129                             u32 handle, struct rtattr **,
130                             unsigned long *);
131     int                     (*delete)(struct tcf_proto*, unsigned long);
132     void                    (*walk)(struct tcf_proto*, struct tcf_walker *arg);
133
134     /* rtnetlink specific */
135     int                     (*dump)(struct tcf_proto*, unsigned long,
136                             struct sk_buff *skb, struct tcmsg*);
137
138     struct module           *owner;
139 };
```

```
118 struct tcf_proto_ops *next
```

用来将已注册过滤器链接到 tcf_proto_base 链表上的指针。

```
119 char kind[IFNAMSIZ]
```

过滤器标识符，通常是过滤器名。

121 int (*classify)(struct sk_buff*, struct tcf_proto*, struct tcf_result *)

报文分类函数，返回值见表9-4。

表9-4 classify()的返回值

返回值	描述
TC_POLICE_OK	报文被过滤器接受
TC_POLICE_RECLASSIFY	报文违背了合法参数（例如最大速率），应该被分配到一个不同的类。不过，该报文仍然没有被抛弃，排队规则可以通过不同的类传输此报文
TC_POLICE_SHOT	报文被过滤器接受，但是过滤器又将其丢弃了，因为它违背了合法参数
TC_POLICE_UNSPEC	过滤器应用的规则与报文不匹配，它应该被送到下一个过滤器或过滤元素

输出参数为tcf_result结构，用来保存报文所属类，包括类标识符和内部标识符。

```
struct tcf_result
{
    unsigned long    class;
    u32        classid;
};
```

如果每个类都存在一个独立的过滤器实例，内部标识符就可以简单地被置为可用。如果内部标识符没有被写到结果结构中，那么classid就必须被映射到排队规则的内部标识符上，通常通过线性搜索来实现。

123 int (*init)(struct tcf_proto*)

过滤器初始化函数，通常在创建过滤器之后被调用。

124 void (*destroy)(struct tcf_proto*)

删除并释放过滤器函数。

126 unsigned long (*get)(struct tcf_proto*, u32 handle)

将一个过滤器元素的句柄映射到一个内部过滤器标识符，实际上是过滤器实例指针，并将其返回。

127 void (*put)(struct tcf_proto*, unsigned long)

解除对由get()得到的过滤器的引用。

128int (*change)(struct tcf_proto*, unsigned long, u32 handle, struct rtattr **, unsigned long *)

用于配置一个新过滤器或是变更一个已存在的过滤器的配置。

131 int (*delete)(struct tcf_proto*, unsigned long)

用来删除一个过滤器的某个元素。过滤器可以在内部分离为过滤器元素，u32类型的句柄被分配给这些内部元素。过滤器如何分离和管理，即是用线性列表还是更有效的数据结构，如散列表，依赖于具体的实现。

132 void (*walk)(struct tcf_proto*, struct tcf_walker *arg)

遍历所有的元素并且调用回调函数取得配置数据和统计数据。

135 int (*dump)(struct tcf_proto*, unsigned long, struct sk_buff *skb, struct tcmsg*)

用于输出过滤器或过滤器元素的配置参数统计数据。

3. 过滤器相关函数

（1）tc_filter_init()

初始化过滤器创建、删除等操作的 netlink 接口。

（2）register_tcf_proto_ops()

将过滤器注册到系统过滤器列表 tcf_proto_base 链表上，只有注册过的过滤器才能被使用。首先查看是否已经存在相同类型的过滤器，如果不存在，那么该过滤器将被追加到过滤器列表的末尾。

（3）unregister_tcf_proto_ops()

注销已注册到系统过滤器列表上的过滤器。如果待注销的过滤器尚未注册，则不做任何操作，返回相应的错误码。

（4）tcf_proto_lookup_ops()

根据过滤器名获得已注册到系统的过滤器。

9.3 默认的 FIFO 排队规则

Linux 默认的排队规则是先进先出（FIFO）排队规则。顾名思义，先进先出排队规则对于任何数据包都不会特殊对待。但事实上，该排队规则内部有三个队列，其中 0 队列优先级最高，1 队列次之，2 队列最低。也就是说当 0 队列还有数据包等待发送时，1 队列数据包就不会被处理，1 队列和 2 队列之间的关系也是如此。

在进入 FIFO 排队规则时，会根据数据包的 TOS 标记，把带有“最小延迟”标记的数据包排入 0 队列。需要注意的是，不能使用 tc 命令向 FIFO 排队规则队列中添加其他的排队规则的数据包，虽然排队规则之间可能有些类似的行为。

根据数据包的优先级，即 IP 首部中的 TOS 字段，排入到相应的队列。TOS 结构如图 9-4 所示，取值说明如表 9-5 所示。

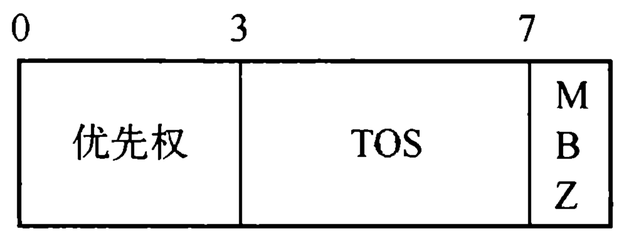

图 9-4 TOS 结构

表 9-5 TOS 字段的取值

TOS	描述
1000	最小延迟(md)
0100	最大吞吐量(mt)
0010	最大可靠性(mr)
0001	最小成本(mmc)
0000	正常服务

从表 9-5 可以看出，事实上 4 位的 TOS 每一个位代表着一种选择，这些选择是可以组合的，此外由于 TOS 后面还有一位，取值总是为 0，因此 TOS 字段的实际值是上述值的 2 倍。实际取值及其意义见表 9-6。

这些 TOS 值通常由应用程序来设置它们，比如 TELNET 的 TOS 为二进制的 1000。可以参考 RFC 1349 得到更多的信息。

表 9-6 TOS 字段与队列

TOS	Bits	意义	Linux	优先级	队列
0x0	0	正常服务	0	最好效果	1
0x2	1	mmc	1	填充	2
0x4	2	mr	0	最好效果	1
0x6	3	mmc+mr	0	最好效果	1
0x8	4	mt	2	大量传输	2
0xa	5	mmc+mt	2	大量传输	2
0xc	6	mr+mt	2	大量传输	2
0xe	7	mmc+mr+mt	2	大量传输	2
0x10	8	md	6	交互	0
0x12	9	mmc+md	6	交互	0
0x14	10	mr+md	6	交互	0
0x16	11	mmc+mr+md	6	交互	0
0x18	12	mt+md	4	交互+大量传输	1
0x1a	13	mmc+mt+md	4	交互+大量传输	1
0x1c	14	mr+mt+md	4	交互+大量传输	1
0x1e	15	mmc+mr+mt+md	4	交互+大量传输	1

FIFO 排队规则的操作接口为 pfifo_fast_ops，如下所示。

```
409 static struct Qdisc_ops pfifo_fast_ops = {
410     .id         =   "pfifo_fast",
411     .priv_size    =    PFIFO_FAST_BANDS * sizeof(struct sk_buff_head),
412     .enqueue    =    pfifo_fast_enqueue,
413     .dequeue    =    pfifo_fast_dequeue,
414     .requeue    =    pfifo_fast_requeue,
415     .init       =    pfifo_fast_init,
416     .reset       =    pfifo_fast_reset,
417     .dump       =    pfifo_fast_dump,
418     .owner       =    THIS_MODULE,
419 };
```

FIFO 排队规则比较简单，没有分类和过滤器，因此数据结构之间的关系也相对简单，Qdisc 实例的操作接口指向 pfifo_fast_ops，尾部的私有信息中也只是 3 个队列，如图 9-5 所示。

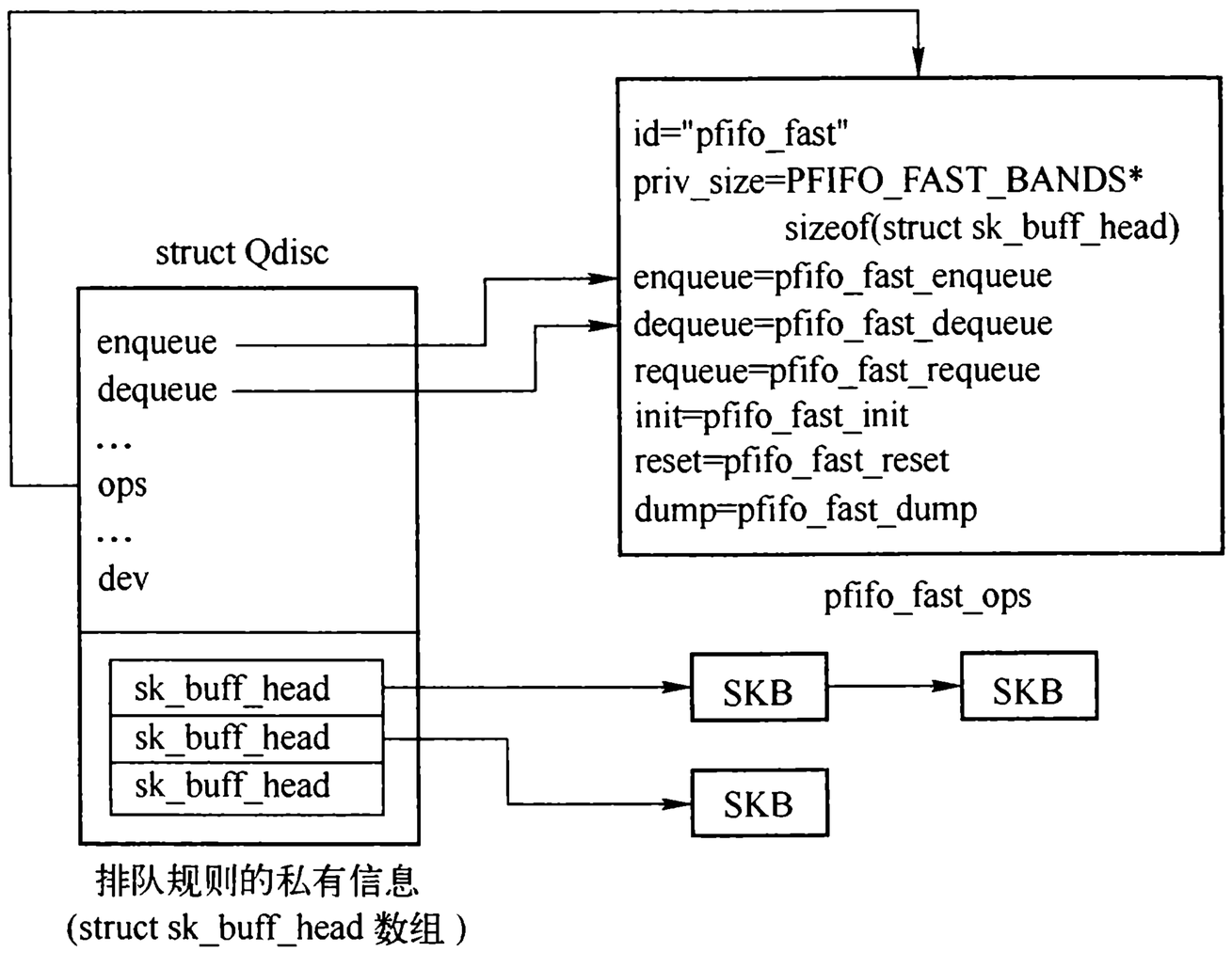

图 9-5 FIFO 排队规则的数据组织结构

9.3.1 pfifo_fast_init()

pfifo_fast_init()是 FIFO 排队规则的初始化函数，用来初始化三个优先级队列。

```
398 static int pfifo_fast_init(struct Qdisc *qdisc, struct rtattr *opt)
399 {
400     int prio;
401     struct sk_buff_head *list = qdisc_priv(qdisc);
402
403     for (prio = 0; prio < PFIFO_FAST_BANDS; prio++)
404         skb_queue_head_init(list + prio);
405
406     return 0;
407 }
```

pfifo 排队规则对外部来说是单个队列，因为队列不可配置，实际在内部存在三个队列。在初始化接口中，初始化三个优先级队列。

9.3.2 pfifo_fast_reset()

通过 reset()调用之后，排队规则恢复到初始化状态。

```
374 static void pfifo_fast_reset(struct Qdisc* qdisc)
375 {
376     int prio;
377     struct sk_buff_head *list = qdisc_priv(qdisc);
378
379     for (prio = 0; prio < PFIFO_FAST_BANDS; prio++)
380         __qdisc_reset_queue(qdisc, list + prio);
381
382     qdisc->qstats.backlog = 0;
383     qdisc->q.qlen = 0;
384 }
```

释放三个优先级队列中的所有数据包，并清零相关数据。

9.3.3 pfifo_fast_enqueue()

pfifo_fast_enqueue()是 FIFO 排队规则入队函数的实现，根据数据包的优先级将其加到对应的队列中。

```
341 static int pfifo_fast_enqueue(struct sk_buff *skb, struct Qdisc* qdisc)
342 {
343     struct sk_buff_head *list = prio2list(skb, qdisc);
344
345     if (skb_queue_len(list) < qdisc->dev->tx_queue_len) {
346         qdisc->q.qlen++;
347         return __qdisc_enqueue_tail(skb, qdisc, list);
348     }
349
350     return qdisc_drop(skb, qdisc);
351 }
```

343 prio2list()将数据包优先级转换为数组下标，得到队列索引数组 prio2band 中应插入的

队列。参见 prio2list()和 prio2band 数组。

345-348 如果队列中的数据包数还未达到上限，则将数据包排到对应优先级队列的队尾。

350 如果队列中的数据包数已达到上限，则丢弃待入队的数据包。

9.3.4 pfifo_fast_dequeue()

pfifo_fast_dequeue()是 FIFO 排队规则出队函数的实现，从最优先的队列中得到数据包，将其出队。

```
353 static struct sk_buff *pfifo_fast_dequeue(struct Qdisc* qdisc)
354 {
355     int prio;
356     struct sk_buff_head *list = qdisc_priv(qdisc);
357
358     for (prio = 0; prio < PFIFO_FAST_BANDS; prio++) {
359         if (!skb_queue_empty(list + prio)) {
360             qdisc->q.qlen--;
361             return __qdisc_dequeue_head(qdisc, list + prio);
362         }
363     }
364
365     return NULL;
366 }
```

按优先级高到低的顺序遍历所有队列，从队列首部获取待输出的数据包，一旦取到即将其返回。因此，只有当高优先级的队列为空，才会从下一优先级的队列中获取数据包。

9.3.5 pfifo_fast_requeue()

当报文由于发送而出队，但又因为某个不可预见的原因最终未能被发送时，便需要将这个数据包重新加入队列，等待下次发送。requeue()用来重新将数据包加入到相应优先级队列的队首。

```
368 static int pfifo_fast_requeue(struct sk_buff *skb, struct Qdisc* qdisc)
369 {
370     qdisc->q.qlen++;
371     return __qdisc_requeue(skb, qdisc, prio2list(skb, qdisc));
372 }
```

370 重新累计排队规则中数据包数。

371 调用__qdisc_requeue()将数据包加入到对应优先级队列的队首。

9.4 netlink 的 tc 接口

Linux 的流量控制是 Linux 特有的，因此它并不需要与 UNIX 兼容。而 netlink 也是 Linux 用于在内核空间与用户空间之间通信的特有方式，目的也是用来配置内核参数，因此流量控制的配置方式顺理成章地选择了 netlink。tc 就是通过 netlink 对流量控制进行配置的命令，由于流量控制本身的复杂性，从而导致了 tc 命令也相对较为复杂。

图 9-6 所示为通过该 netlink 接口来配置流量控制的主要函数。

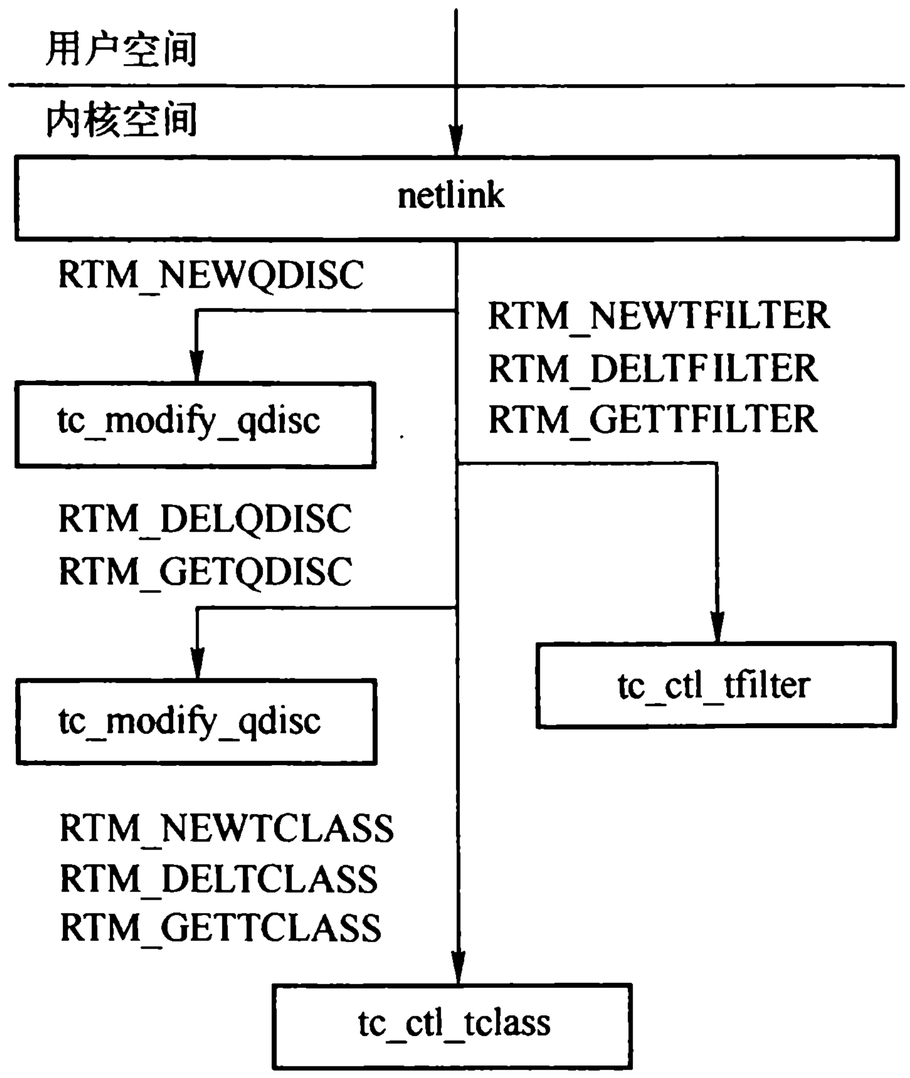

图 9-6　netlink 接口

图 9-7 所示为配置流量控制的 netlink 消息结构，包括创建/修改/删除排队规则、类以及过滤器，该接口通用于输入和输出的排队规则。

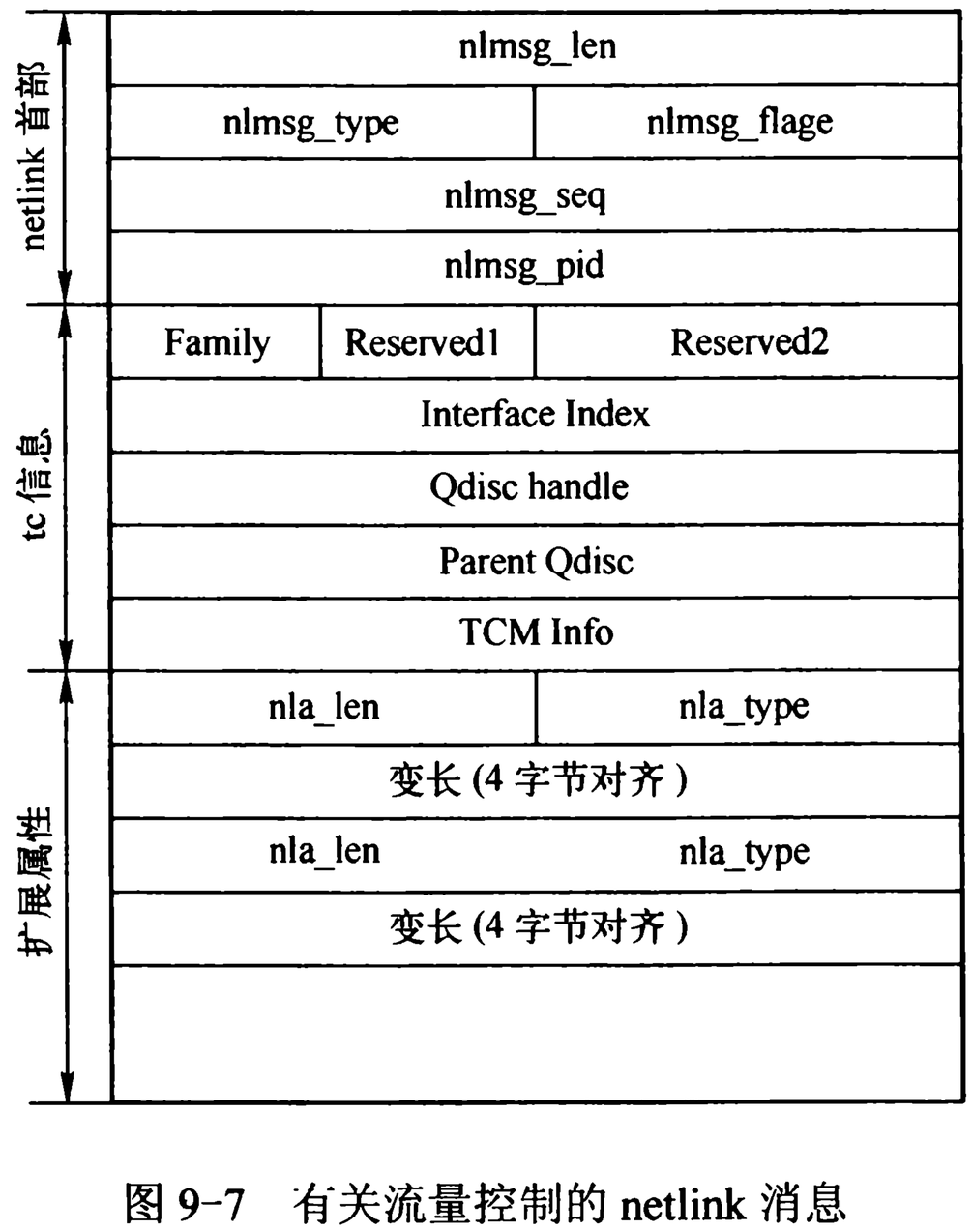

图 9-7　有关流量控制的 netlink 消息

其中各字段的意义如下：

- Family，8 位，标识排队规则所绑定的地址族，如 AF_INET、AF_INET6 等。
- Interface Index，32 位，待配置排队规则的网络设备索引。
- Qdisc handle，32 位，待创建或修改排队规则的句柄。
- Parent Qdisc，32 位，创建或修改排队规则的父排队规则，由此可形成分层的流量控制。如果该字段值为 TC_H_ROOT，则说明是根排队规则。

- TCM Info，32 位，对于排队规则，通常取值为 1，如果排队规则正在被使用，则为排队规则的引用计数；对于类，是与类绑定的排队规则句柄；对于过滤器，则高 16 位为优先级，低 16 位为报文的网络层协议号。
- 扩展属性，配置流量控制用到的一些属性，见表 9-7。

表 9-7 扩展属性

属性	描述
TCA_KIND	流量控制中组件的名称
TCA_STATS	流量控制中组件的一般统计信息
TCA_RATE	用来计算速率的统计信息快照
TCA_XSTATS	流量控制中组件的特定统计信息
TCA_OPTIONS	流量控制中组件的特定属性

图 9-8 所示为一个简单的用于配置 FIFO 队列流量控制的 netlink 消息。该队列的算法是当队列中缓存的报文数超过 100 时，丢弃后来的报文。配置在网络设备 0 上，父排队规则为 100:0，排队规则句柄为 100:1。

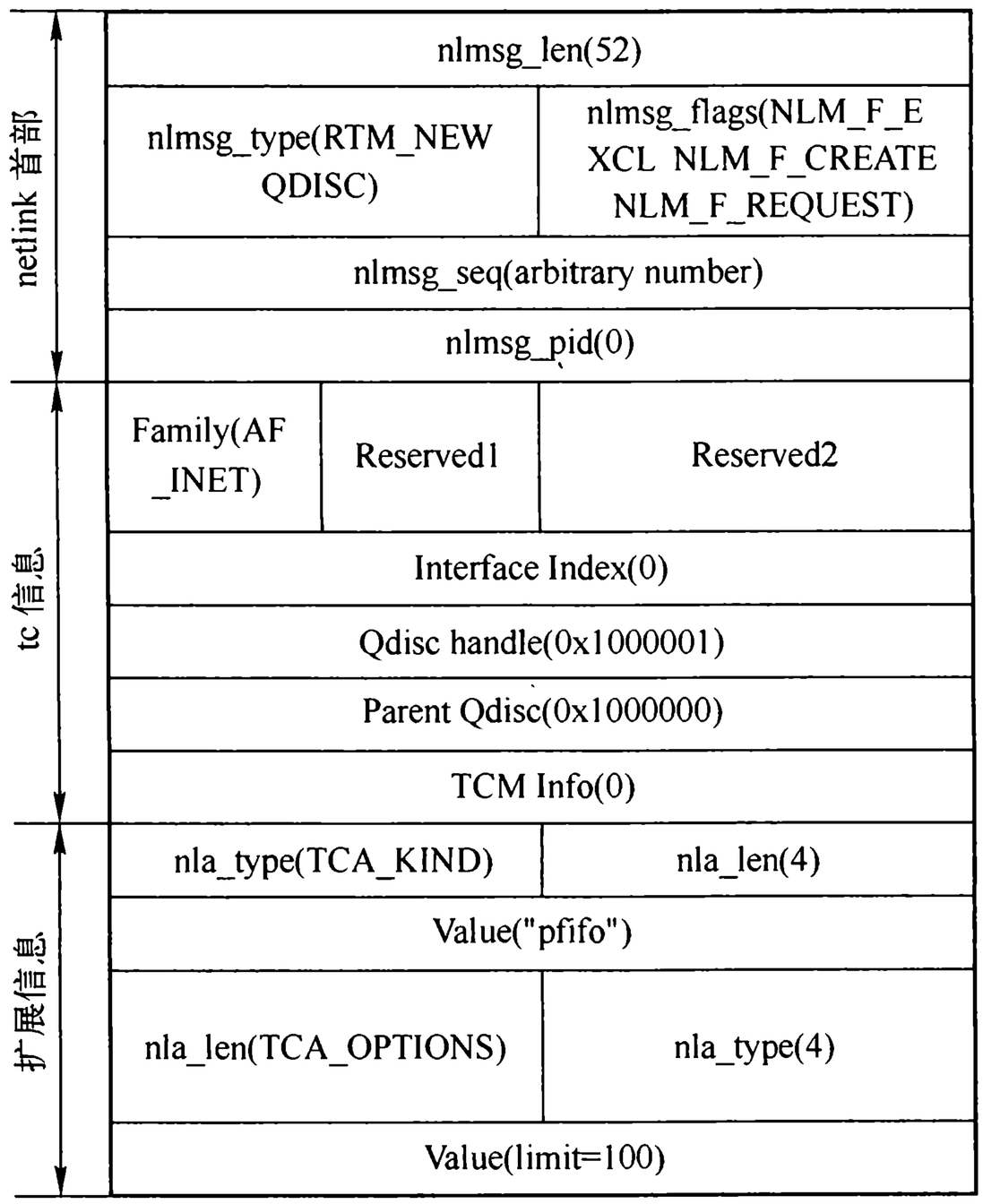

图 9-8 配置流量控制的例子

9.5 排队规则的创建接口

当用 tc 工具创建排队规则时，通过 netlink 后最终由 tc_modify_qdisc()来处理。参数如下所示：

- skb，创建排队规则所用的 netlink 报文。
- n，netlink 报文的首部。

- arg，配置排队规则消息的扩展属性。

```
639 static int tc_modify_qdisc(struct sk_buff *skb, struct nlmsghdr *n, void *arg)
640 {
641     struct tcmsg *tcm;
642     struct rtattr **tca;
643     struct net_device *dev;
644     u32 clid;
645     struct Qdisc *q, *p;
646     int err;
647
648 replay:
649     /* Reinit, just in case something touches this. */
650     tcm = NLMSG_DATA(n);
651     tca = arg;
652     clid = tcm->tcm_parent;
653     q = p = NULL;
654
655     if ((dev = __dev_get_by_index(tcm->tcm_ifindex)) == NULL)
656         return -ENODEV;
```

650 将配置排队规则的 netlink 消息映射成流量控制消息。

655-656 获取配置队列的网络设备，该网络设备必须存在，否则配置失败。

```
658     if (clid) {
659         if (clid != TC_H_ROOT) {
660             if (clid != TC_H_INGRESS) {
661                 if ((p = qdisc_lookup(dev, TC_H_MAJ(clid))) == NULL)
662                     return -ENOENT;
663                 q = qdisc_leaf(p, clid);
664             } else { /*ingress */
665                 q = dev->qdisc_ingress;
666             }
667         } else {
668             q = dev->qdisc_sleeping;
669         }
670
671         /* It may be default qdisc, ignore it */
672         if (q && q->handle == 0)
673             q = NULL;
674
675         if (!q || !tcm->tcm_handle || q->handle != tcm->tcm_handle) {
676             if (tcm->tcm_handle) {
677                 if (q && !(n->nlmsg_flags&NLM_F_REPLACE))
678                     return -EEXIST;
679                 if (TC_H_MIN(tcm->tcm_handle))
680                     return -EINVAL;
681                 if ((q = qdisc_lookup(dev, tcm->tcm_handle)) == NULL)
682                     goto create_n_graft;
683                 if (n->nlmsg_flags&NLM_F_EXCL)
684                     return -EEXIST;
685                 if (tca[TCA_KIND-1] && rtattr_strcmp(tca[TCA_KIND-1], q->ops->id))
686                     return -EINVAL;
687                 if (q == p ||
688                     (p && check_loop(q, p, 0)))
```

```
689                 return -ELOOP;
690             atomic_inc(&q->refcnt);
691             goto graft;
692         } else {
693             if (q == NULL)
694                 goto create_n_graft;
695
696             /* This magic test requires explanation.
697              *
698              *   We know, that some child q is already
699              *   attached to this parent and have choice:
700              *   either to change it or to create/graft new one.
701              *
702              *   1. We are allowed to create/graft only
703              *   if CREATE and REPLACE flags are set.
704              *
705              *   2. If EXCL is set, requestor wanted to say,
706              *   that qdisc tcm_handle is not expected
707              *   to exist, so that we choose create/graft too.
708              *
709              *   3. The last case is when no flags are set.
710              *   Alas, it is sort of hole in API, we
711              *   cannot decide what to do unambiguously.
712              *   For now we select create/graft, if
713              *   user gave KIND, which does not match existing.
714              */
715             if ((n->nlmsg_flags&NLM_F_CREATE) &&
716                 (n->nlmsg_flags&NLM_F_REPLACE) &&
717                 ((n->nlmsg_flags&NLM_F_EXCL) ||
718                  (tca[TCA_KIND-1] &&
719                   rtattr_strcmp(tca[TCA_KIND-1], q->ops->id))))
720                 goto create_n_graft;
721         }
722     }
```

658-722 处理待配置排队规则存在父排队规则的情况。

659-669 获取排队规则。如果父排队规则是输出/输入根排队规则，则直接从网络设备上获取相应的排队规则，前者为 qdisc_sleeping，后者为 qdisc_ingress；否则根据父排队规则句柄的高 16 位，在网络设备上查找对应的排队规则，然后取其叶子结点。

672-673 如果排队规则是默认排队规则，则认为当前没有配置排队规则。

675-722 如果当前没有配置排队规则，或者未指定待创建排队规则的句柄，又或者指定的排队规则句柄与当前配置排队规则的不一致，则需要做更多的处理。

在指定配置排队规则句柄的情况下：

- 如果当前已经配置了排队规则，并且没有设置可替换标志，则返回已存在错误码。
- 如果指定了排队规则的句柄副编号，则返回参数无效错误码。
- 通过以上校验后，又如果在指定网络设备的排队规则中未找到相同句柄的排队规则，则创跳转到 create_n_graft 建排队规则。
- 如果指定网络设备的排队规则中已存在相同句柄的排队规则，并且设置了 NLM_F_EXCL 标志，则返回已存在错误码。
- tc 命令中存在算法名且与排队规则中的算法名不同，则返回参数无效错误码。

- 算法出现回环情况，则返回相应的错误码。
- 以上情况都未出现，则在递增排队规则引用计数后跳转到 graft 处处理。

在未指定配置排队规则句柄的情况下，如果当前没有配置排队规则，则创建排队规则。如果当前配置了排队规则且可以覆盖，则修改排队规则的属性。

```
723     } else {
724         if (!tcm->tcm_handle)
725             return -EINVAL;
726         q = qdisc_lookup(dev, tcm->tcm_handle);
727     }
```

723-727 在未指定父结点句柄的情况下，如果也没有指定排队规则句柄，则返回参数无效错误码；否则根据指定的排队规则句柄，从网络设备上获取排队规则。

```
729     /* Change qdisc parameters */
730     if (q == NULL)
731         return -ENOENT;
732     if (n->nlmsg_flags&NLM_F_EXCL)
733         return -EEXIST;
734     if (tca[TCA_KIND-1] && rtattr_strcmp(tca[TCA_KIND-1], q->ops->id))
735         return -EINVAL;
736     err = qdisc_change(q, tca);
737     if (err == 0)
738         qdisc_notify(skb, n, clid, NULL, q);
739     return err;
```

730-731 如果未得到需修改参数的排队规则，则返回错误码。

732-733 待修改参数的排队规则，不能设置有 NLM_F_EXCL 标志。

734-735 如果 tc 命令中存在算法名且与排队规则中的算法名不同，则返回参数无效错误码。

736-739 如果修改排队规则参数成功，则发送确认消息给修改配置的进程；否则返回相应的错误码。

```
741 create_n_graft:
742     if (!(n->nlmsg_flags&NLM_F_CREATE))
743         return -ENOENT;
744     if (clid == TC_H_INGRESS)
745         q = qdisc_create(dev, tcm->tcm_parent, tca, &err);
746         else
747         q = qdisc_create(dev, tcm->tcm_handle, tca, &err);
748     if (q == NULL) {
749         if (err == -EAGAIN)
750             goto replay;
751         return err;
752     }
```

741-752 创建排队规则。

742-743 待创建排队规则必须有 NLM_F_CREATE 标志。

744-747 在创建排队规则时，输入排队规则和输出排队规则适用的句柄是不同的。

748-751 创建失败，如果错误码为 EAGAIN，则跳转到 replay 处重新尝试创建排队规则；否则返回相应的错误码。

```
754 graft:
755     if (1) {
756         struct Qdisc *old_q = NULL;
757         err = qdisc_graft(dev, p, clid, q, &old_q);
758         if (err) {
759             if (q) {
760                 spin_lock_bh(&dev->queue_lock);
761                 qdisc_destroy(q);
762                 spin_unlock_bh(&dev->queue_lock);
763             }
764             return err;
765         }
766         qdisc_notify(skb, n, clid, old_q, q);
767         if (old_q) {
768             spin_lock_bh(&dev->queue_lock);
769             qdisc_destroy(old_q);
770             spin_unlock_bh(&dev->queue_lock);
771         }
772     }
773     return 0;
774 }
```

如果成功创建排队规则或者修改排队规则参数，则需重新绑定类。如果绑定失败，则释放新创建的排队规则，并返回相应的错误码；如果成功，则发送确认消息给创建或修改配置的进程，并释放原先绑定的排队规则。

9.5.1 类的创建接口

使用 tc 工具配置排队规则类，通过 netlink 后最终由 tc_ctl_tclass()来处理。

```
900 static int tc_ctl_tclass(struct sk_buff *skb, struct nlmsghdr *n, void *arg)
901 {
902     struct tcmsg *tcm = NLMSG_DATA(n);
903     struct rtattr **tca = arg;
904     struct net_device *dev;
905     struct Qdisc *q = NULL;
906     struct Qdisc_class_ops *cops;
907     unsigned long cl = 0;
908     unsigned long new_cl;
909     u32 pid = tcm->tcm_parent;
910     u32 clid = tcm->tcm_handle;
911     u32 qid = TC_H_MAJ(clid);
912     int err;
913
914     if ((dev = __dev_get_by_index(tcm->tcm_ifindex)) == NULL)
915         return -ENODEV;
```

914-915 获取创建类所在的网络设备，该网络设备必须存在，否则配置失败。

```
917     /*
918       parent == TC_H_UNSPEC - unspecified parent.
919       parent == TC_H_ROOT   - class is root, which has no parent.
920       parent == X:0     - parent is root class.
921       parent == X:Y     - parent is a node in hierarchy.
922       parent == 0:Y     - parent is X:Y, where X:0 is qdisc.
```

```
923
924         handle == 0:0    - generate handle from kernel pool.
925         handle == 0:Y    - class is X:Y, where X:0 is qdisc.
926         handle == X:Y    - clear.
927         handle == X:0    - root class.
928      */
929
930     /* Step 1. Determine qdisc handle X:0 */
931
932     if (pid != TC_H_ROOT) {
933        u32 qid1 = TC_H_MAJ(pid);
934
935        if (qid && qid1) {
936           /* If both majors are known, they must be identical. */
937           if (qid != qid1)
938              return -EINVAL;
939        } else if (qid1) {
940           qid = qid1;
941        } else if (qid == 0)
942           qid = dev->qdisc_sleeping->handle;
943
944        /* Now qid is genuine qdisc handle consistent
945          both with parent and child.
946
947          TC_H_MAJ(pid) still may be unspecified, complete it now.
948        */
949        if (pid)
950           pid = TC_H_MAKE(qid, pid);
951     } else {
952        if (qid == 0)
953           qid = dev->qdisc_sleeping->handle;
954     }
```

932-954 确定与该类绑定的排队规则及其父排队规则的句柄，确定的规则如表 9-8 和表 9-9 所示。

表 9-8　确定父排队规则句柄的规则

Parent	描述
TC_H_UNSPEC	无效父排队规则的句柄
TC_H_ROOT	根排队规则的类　没有父排队规则
X:0	父排队规则是根排队规则
X:Y	父排队规则是一个普通的结点
0:Y	如果父排队规则为 X.Y　则排队规则为 X:0

表 9-9　确定排队规则句柄的规则

handle	描述
0:0	系统自动进行分配
0:Y	当类的句柄为 X.Y　则与之绑定的排队规则的句柄为 X:0
X.Y	清除
X:0	根类

932-938 如果父排队规则不是根排队规则，且排队规则与父排队规则句柄的高 16 位，即

主编号不同，不是真正的父子关系，则返回参数无效错误码。

939-940 如果没有指定排队规则的主编号，则从父排队规则中获取。

941-942 如果排队规则和父排队规则的主编号都没有指定，则从网络设备当前配置的排队规则 qdisc_sleeping 中获取。

949-950 确定了父排队规则的主编号后，重新计算父排队规则的句柄。

951-954 如果父排队规则为根排队规则，且未指定排队规则的主编号，则从网络设备当前配置的排队规则 qdisc_sleeping 中获取。

```
956     /* OK. Locate qdisc */
957     if ((q = qdisc_lookup(dev, qid)) == NULL)
958        return -ENOENT;
959
960     /* An check that it supports classes */
961     cops = q->ops->cl_ops;
962     if (cops == NULL)
963        return -EINVAL;
964
965     /* Now try to get class */
966     if (clid == 0) {
967        if (pid == TC_H_ROOT)
968           clid = qid;
969     } else
970        clid = TC_H_MAKE(qid, clid);
971
972     if (clid)
973        cl = cops->get(q, clid);
```

957-958 根据句柄查找创建或修改类绑定的排队规则。如果不存在，则返回相应的错误码。

961-963 获取所在排队规则的类操作接口。如果不存在，则返回参数无效错误码。

966-973 尝试获取排队规则的类。

```
975     if (cl == 0) {
976        err = -ENOENT;
977        if (n->nlmsg_type != RTM_NEWTCLASS || !(n->nlmsg_flags&NLM_F_CREATE))
978           goto out;
979     } else {
980        switch (n->nlmsg_type) {
981        case RTM_NEWTCLASS:
982           err = -EEXIST;
983           if (n->nlmsg_flags&NLM_F_EXCL)
984              goto out;
985           break;
986        case RTM_DELTCLASS:
987           err = cops->delete(q, cl);
988           if (err == 0)
989              tclass_notify(skb, n, q, cl, RTM_DELTCLASS);
990           goto out;
991        case RTM_GETTCLASS:
992           err = tclass_notify(skb, n, q, cl, RTM_NEWTCLASS);
993           goto out;
994        default:
995           err = -EINVAL;
```

```
996             goto out;
997         }
998     }
```

975-978 如果没有得到排队规则的类，且不是新建类命令，则跳转到 out 处做处理。

980 如果得到了排队规则的类，则根据操作类型作相应的处理。

981-985 创建类操作，且有 NLM_F_EXCL 标志（表示如果存在则不创建），则跳转到 out 处做处理；否则进行修改配置操作。

986-990 删除操作，调用 delete 接口进行。如果删除成功，则发送相应消息到配置应用程序。

991-993 获取操作，因为已获取，因此直接发相应消息到配置应用程序，然后跳转到 out 处做处理。

994-996 其他都是无效操作。

```
1000     new_cl = cl;
1001     err = cops->change(q, clid, pid, tca, &new_cl);
1002     if (err == 0)
1003         tclass_notify(skb, n, q, new_cl, RTM_NEWTCLASS);
1004
1005 out:
1006     if (cl)
1007         cops->put(q, cl);
1008
1009     return err;
1010 }
```

1000-1003 无论是新建类还是修改类参数都通过 change 接口进行，并在成功后发送相应消息到配置应用程序。

1005-1009 最后递减对该类的引用计数，返回相应的错误码。

9.5.2 过滤器的创建接口

使用 tc 工具配置类的过滤器，通过 netlink 后最终由 tc_ctl_tfilter()来处理。

```
130 static int tc_ctl_tfilter(struct sk_buff *skb, struct nlmsghdr *n, void *arg)
131 {
132     struct rtattr **tca;
133     struct tcmsg *t;
134     u32 protocol;
135     u32 prio;
136     u32 nprio;
137     u32 parent;
138     struct net_device *dev;
139     struct Qdisc  *q;
140     struct tcf_proto **back, **chain;
141     struct tcf_proto *tp;
142     struct tcf_proto_ops *tp_ops;
143     struct Qdisc_class_ops *cops;
144     unsigned long cl;
145     unsigned long fh;
146     int err;
147
148 replay:
```

```
149     tca = arg;
150     t = NLMSG_DATA(n);
151     protocol = TC_H_MIN(t->tcm_info);
152     prio = TC_H_MAJ(t->tcm_info);
153     nprio = prio;
154     parent = t->tcm_parent;
155      cl = 0;
```

150-153 获取优先级以及用于分类的网络层协议号。

154 获取配置过滤器的父排队规则。

```
157     if (prio == 0) {
158         /* If no priority is given, user wants we allocated it. */
159         if (n->nlmsg_type != RTM_NEWTFILTER || !(n->nlmsg_flags&NLM_F_CREATE))
160             return -ENOENT;
161         prio = TC_H_MAKE(0x80000000U,0U);
162     }
```

157-162 如果没有指定优先级，则自动为其分配一个优先级。

```
164     /* Find head of filter chain. */
165
166     /* Find link */
167     if ((dev = __dev_get_by_index(t->tcm_ifindex)) == NULL)
168         return -ENODEV;
```

167-168 获取创建过滤器所在的网络设备，该网络设备必须存在，否则配置失败。

```
170     /* Find qdisc */
171     if (!parent) {
172         q = dev->qdisc_sleeping;
173         parent = q->handle;
174     } else if ((q = qdisc_lookup(dev, TC_H_MAJ(t->tcm_parent))) == NULL)
175         return -EINVAL;
```

171-175 如果没有指定父排队规则句柄，则使用网络设备上配置的排队规则qdisc_sleeping；否则根据指定的父排队规则句柄在网络设备上查找。

```
177     /* Is it classful? */
178     if ((cops = q->ops->cl_ops) == NULL)
179         return -EINVAL;
180                 •
181     /* Do we search for filter, attached to class? */
182     if (TC_H_MIN(parent)) {
183         cl = cops->get(q, parent);
184         if (cl == 0)
185             return -ENOENT;
186     }
187
188     /* And the last stroke */
189     chain = cops->tcf_chain(q, cl);
190     err = -EINVAL;
191     if (chain == NULL)
192         goto errout;
193
```

```
194     /* Check the chain for existence of proto-tcf with this priority */
195     for (back = chain; (tp=*back) != NULL; back = &tp->next) {
196        if (tp->prio >= prio) {
197           if (tp->prio == prio) {
198              if (!nprio || (tp->protocol != protocol && protocol))
199                 goto errout;
200           } else
201              tp = NULL;
202           break;
203        }
204     }
```

178-179 排队规则必须支持类操作接口，才可以创建过滤器。

182-186 如果获取绑定在父排队规则的类失败，则返回相应的错误码。

189-192 获取绑定到类的过滤器所在链表的首结点。如果首结点有效，则遍历该链表，根据协议号和优先级查找对应的过滤器。

```
206     if (tp == NULL) {
207        /* Proto-tcf does not exist, create new one */
208
209        if (tca[TCA_KIND-1] == NULL || !protocol)
210           goto errout;
211
212        err = -ENOENT;
213        if (n->nlmsg_type != RTM_NEWTFILTER || !(n->nlmsg_flags&NLM_F_CREATE))
214           goto errout;
215
216
217        /* Create new proto tcf */
218
219        err = -ENOBUFS;
220        if ((tp = kzalloc(sizeof(*tp), GFP_KERNEL)) == NULL)
221           goto errout;
222        err = -EINVAL;
223        tp_ops = tcf_proto_lookup_ops(tca[TCA_KIND-1]);
224        if (tp_ops == NULL) {
225 #ifdef CONFIG_KMOD
226           struct rtattr *kind = tca[TCA_KIND-1];
227           char name[IFNAMSIZ];
228
229           if (kind != NULL &&
230              rtattr_strlcpy(name, kind, IFNAMSIZ) < IFNAMSIZ) {
231              rtnl_unlock();
232              request_module("cls_%s", name);
233              rtnl_lock();
234              tp_ops = tcf_proto_lookup_ops(kind);
235              /* We dropped the RTNL semaphore in order to
236               * perform the module load.  So, even if we
237               * succeeded in loading the module we have to
238               * replay the request.  We indicate this using
239               * -EAGAIN.
240               */
241              if (tp_ops != NULL) {
242                 module_put(tp_ops->owner);
243                 err = -EAGAIN;
```

```
244                 }
245             }
246 #endif
247             kfree(tp);
248             goto errout;
249         }
250         tp->ops = tp_ops;
251         tp->protocol = protocol;
252         tp->prio = nprio ? : tcf_auto_prio(*back);
253         tp->q = q;
254         tp->classify = tp_ops->classify;
255         tp->classid = parent;
256         if ((err = tp_ops->init(tp)) != 0) {
257             module_put(tp_ops->owner);
258             kfree(tp);
259             goto errout;
260         }
261
262         qdisc_lock_tree(dev);
263         tp->next = *back;
264         *back = tp;
265          qdisc_unlock_tree(dev);
```

206 查找指定的过滤器失败，需新建过滤器。

209-210 检测待创建过滤器名和协议是否有效。

212-214 必须是新建过滤器命令，才能创建过滤器。

219-221 为创建过滤器分配内存。

222-223 检测过滤器操作结构。

224-249 在过滤器操作结构不存在的情况下，如果允许加载模块，则加载对应的模块；否则释放已分配内存跳转到 errout 处做处理。

250-260 为新创建过滤器结构设置各字段值，并初始化。

262-265 将新建过滤器添加到过滤器列表中。

```
267     } else if (tca[TCA_KIND-1] && rtattr_strcmp(tca[TCA_KIND-1], tp->ops->kind))
268         goto errout;
```

267-268 如果查找到指定的过滤器，则还需校验过滤器名。

```
270     fh = tp->ops->get(tp, t->tcm_handle);
271
272     if (fh == 0) {
273         if (n->nlmsg_type == RTM_DELTFILTER && t->tcm_handle == 0) {
274             qdisc_lock_tree(dev);
275             *back = tp->next;
276             qdisc_unlock_tree(dev);
277
278             tfilter_notify(skb, n, tp, fh, RTM_DELTFILTER);
279             tcf_destroy(tp);
280             err = 0;
281             goto errout;
282         }
283
284         err = -ENOENT;
```

```
285         if (n->nlmsg_type != RTM_NEWTFILTER || !(n->nlmsg_flags&NLM_F_CREATE))
286             goto errout;
287     } else {
288         switch (n->nlmsg_type) {
289         case RTM_NEWTFILTER:
290             err = -EEXIST;
291             if (n->nlmsg_flags&NLM_F_EXCL)
292                 goto errout;
293             break;
294         case RTM_DELTFILTER:
295             err = tp->ops->delete(tp, fh);
296             if (err == 0)
297                 tfilter_notify(skb, n, tp, fh, RTM_DELTFILTER);
298             goto errout;
299         case RTM_GETTFILTER:
300             err = tfilter_notify(skb, n, tp, fh, RTM_NEWTFILTER);
301             goto errout;
302         default:
303             err = -EINVAL;
304             goto errout;
305         }
306     }
```

270 获取过滤器句柄映射到一个内部过滤器标识符。

272-286 在没有得到内部过滤器标识符的情况下，如果是删除过滤器操作，且没有指定待删除过滤器句柄，则执行删除操作；否则除创建过滤器外，其他操作都是无效的。

288 在获取到内部过滤器标识符的情况下，根据操作类型作相应的处理。

289-293 新建操作，如果有 NLM_F_EXCL 标志，即表示存在则不创建，则返回相应的错误码；否则进行创建过滤器操作。

294-298 删除操作，调用 delete 接口进行，如果成功则发送相应消息到配置应用程序。

299-301 获取操作，因为已经获取，直接发相应消息到配置应用程序。

302-304 其他都为无效操作。

```
308     err = tp->ops->change(tp, cl, t->tcm_handle, tca, &fh);
309     if (err == 0)
310         tfilter_notify(skb, n, tp, fh, RTM_NEWTFILTER);
```

308-310 无论是新建过滤器还是修改过滤器参数都通过 change 接口进行，成功后发送相应消息到配置应用程序。

```
312 errout:
313     if (cl)
314         cops->put(q, cl);
315     if (err == -EAGAIN)
316         /* Replay the request. */
317         goto replay;
318     return err;
319 }
```

313-314 返回前递减对该过滤器的引用计数。

315-317 如果错误码为 EAGAIN，则跳转到 replay 处再次进行尝试处理。

318 最后返回相应的错误码。

第 10 章　Internet 协议族

本章将描述几个同时支持多种网络协议的数据结构，并以 Internet 协议为例，说明在系统初始化时对这些数据结构的构造与初始化。本章的内容为以后对 IP 协议处理层及套接口处理层的讨论提供了必要的背景知识，而这两部分的内容将分别在第 11、22 章中论述。

Linux 目前最多可支持 32 种协议族，每个协议族用一个 net_proto_family 结构实例来表示，在系统初始化时，以各协议族对应的协议族常量为下标，调用 sock_register()将结构注册到全局数组 net_families[NPROTO]中（NPROTO 为 32）。此外还有一个地址族的概念，地址族用地址族常量来标识，到目前为止，协议族常量和地址族常量是一一对应的，且值相同，表 10-1 列出了一些常用协议族和地址族常量。

表 10-1　公共的协议和地址族常量

协议族	地址族	协议
PF_INET	AF_INET	Internet
PF_OSI, PF_ISO	AF_OSI, AF_ISO	OSI
PF_LOCAL, PF_UNIX	AF_LOCAL, AF_UNIX	本地 IPC(UNIX)
PF_ROUTE	AF_ROUTE	路由表
PF_LINK	AF_LINK	链路层（例如以太网）

本章论述的套接口层结构的定义、用于传输层协议的结构等定义以及 IPv4 协议族的初始化涉及以下文件：

- include/linux/net.h，定义套接口层的结构、宏和函数原型。
- include/linux/protocol.h，定义注册传输层协议的结构、宏和函数原型。
- net/ipv4/af_inet.c，网络层和传输层接口。
- net/ipv4/ip_input.c，IP 数据报的输入。

10.1　net_proto_family 结构

对于不同的协议族，其传输层的结构和实现有着巨大的差异，因此其各自的套接口创建函数也会有很大区别，而 net_proto_family 结构屏蔽了这些区别，使得各协议族在初始化时，可以统一用 sock_register()注册到 net_families 数组中。因此实际上 net_proto_family 结构提供了一个协议族到套接口创建之间的接口。

```
170 struct net_proto_family {
171     int         family;
172     int         (*create)(struct socket *sock, int protocol);
173     struct module    *owner;
174 };
```

```
171 int family
```

是协议族对应的协议族常量，Internet 协议族是 PF_INET。

```
172 int (*create)(struct socket *sock, int protocol)
```

协议族的套接口创建函数指针，每个协议族都有不同的实现方式，如图 10-1 所示。

Internet 协议族的 net_proto_family 结构实例为 inet_family_ops，套接口创建函数为 inet_create()，参见 22.7.2 节。

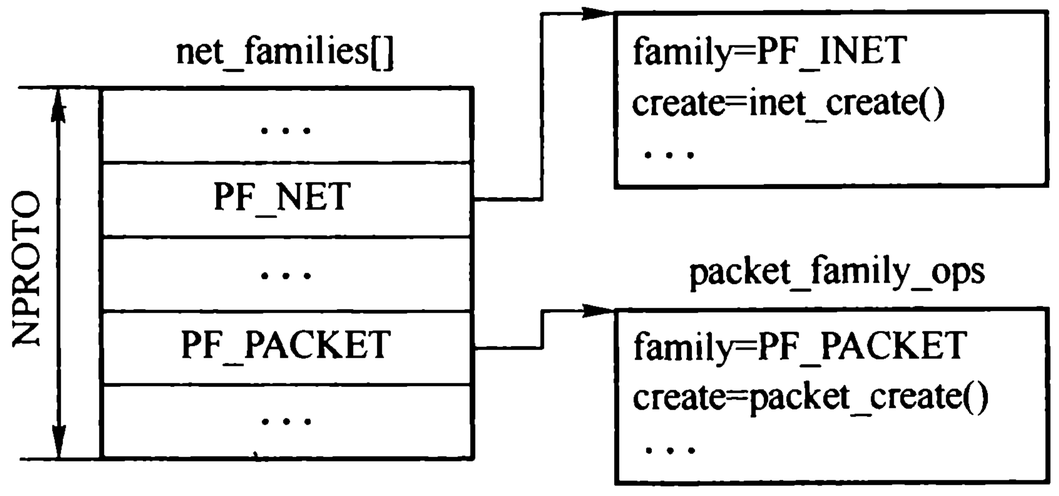

图 10-1　net_families 结构的定义

10.2　inet_protosw 结构

inet_protosw 只是一个比较重要的结构，每次创建套接口时会用到，此结构也只在套接口层起作用。

```
69 struct inet_protosw {
70     struct list_head list;
73     unsigned short  type;/* This is the 2nd argument to socket(2).*/
74     unsigned short  protocol; /* This is the L4 protocol number. */
76     struct proto    *prot;
77     const struct proto_ops *ops;
79     int             capability;
83     char            no_check;   /* checksum on rcv/xmit/none? */
84     unsigned char    flags;     /* See INET_PROTOSW_* below.  */
85 };
```

```
70 struct list_head list
```

用于初始化时在散列表中将 type 值相同的 inet_protosw 结构实例连接成链表。

```
73 unsigned short type
```

标识套接口的类型，对于 Internet 协议族共有三种类型 SOCK_STREAM、SOCK_DGRAM 和 SOCK_RAW，与应用程序层创建套接口函数 socket()的第二个参数 type 取值恰好对应。

```
74 unsigned short protocol
```

标识协议族中四层协议号，Internet 协议族中的值包括 IPPROTO_TCP、IPPROTO_UDP 等。

76 struct proto *prot

套接口网络层接口。TCP 为 tcp_prot；UDP 为 udp_prot；原始套接口则是 raw_prot。

77 const struct proto_ops *ops

套接口传输层接口。与网络接口层之间的关系参见第 25 章。TCP 为 inet_stream_ops；UDP 为 inet_dgram_ops；原始套接口则是 inet_sockraw_ops。

79 int capability

当大于零时，需要检验当前创建套接口的进程是否有这种能力。TCP 和 UDP 均为-1，表示无需进行能力的检验，只有原始套接口为 CAP_NET_RAW。

83 char no check

标识是否需要执行校验和。对于 TCP 来说，进行校验和是必须的，因此值是唯一的，为 0。注意此处 0 为要校验。而 RAW 和 UDP 的 no_check 值见表 10-2。

表 10-2　no_check 取值

no_check	描述
UDP_CSUM_NOXMIT	表示在发送时不做校验和
UDP_CSUM_NORCV	表示在接收时不做校验和
UDP_CSUM_DEFAULT	示进行正常的校验和操作

84 unsigned char flags

辅助标志，用于初始化传输控制块的 is_icsk 成员，见表 10-3。

表 10-3　flags 取值

flags	描述
INET_PROTOSW_REUSE	标识端口是否能被重用
INET_PROTOSW_PERMANENT	标识此协议不能被替换或卸载
INET_PROTOSW_ICSK	标识是不是连接类型的套接口

由此可以知道 TCP 协议是不能被替换和卸载的，且 TCP 套接口是连接型的套接口；UDP 协议也是不能被替换和卸载的；而原始套接口端口可重用。

inetsw_array 数组包含了三个 inet_protosw 结构的实例，分别对应 TCP、UDP 和原始套接口。在 Internet 协议族初始化函数 inet_init()中，调用 inet_register_protosw()将 inetsw_array 数组中的 inet_protosw 结构实例，以其 type 值为 key 组织到散列表 inetsw 中，也就是说各协议族中 type 值相同而 protocol 值不同的 inet_protosw 结构实例，在 inetsw 散列表中以 type 为关键字连接成链表，通过 inetsw 散列表可以找到所有协议族的 inet_protosw 结构实例，如图 10-2 所示。

inet_register_protosw()将 inet_protosw 实例注册到 inetsw 散列表中，而 inet_unregister_protosw()可将指定的 inet_protosw 实例从 inetsw 散列表中注销，但是注意：被标识为 INET_PROTOSW_PERMANENT 的 inet_protosw 实例不能重载或注销。把 inet_protosw 实例注册到 inet_protosw 散列表，或是从该散列表中注销，都是一个动态的过程，不但能在系统初始化时进行，在系统运行时也可以进行。

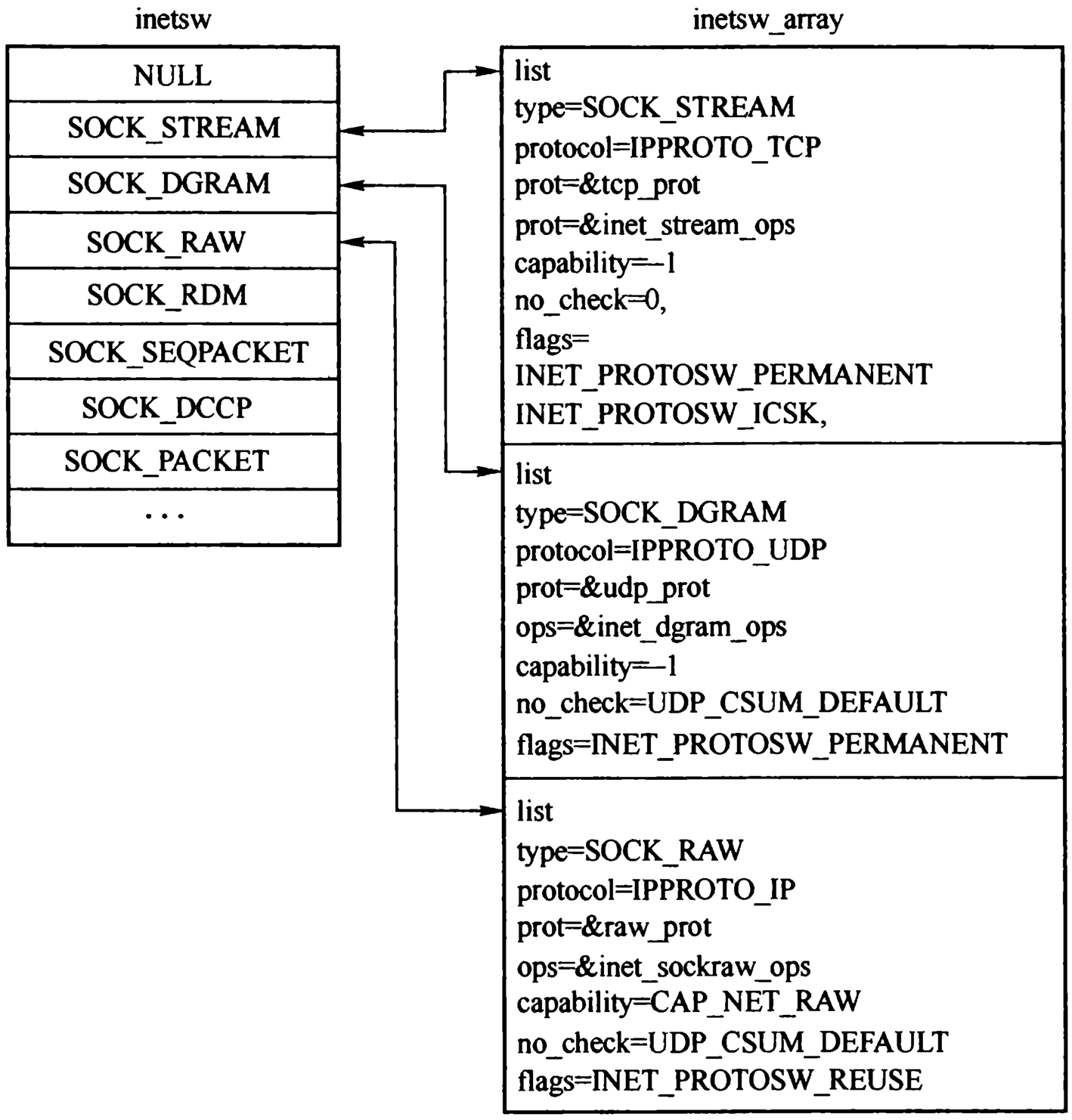

图 10-2　inetsw 散列表与 inet_protosw 数组

10.3　net_protocol 结构

net_protocol 是一个非常重要的结构，定义了协议族中支持的传输层协议以及传输层的报文接收例程。此结构是网络层和传输层之间的桥梁，当网络数据报从网络层流向传输层时，会调用此结构中的传输层协议数据报接收处理函数。注意：此处说“传输层”并不准确，事实上包括了 ICMP 和 IGMP 协议。

```
36 struct net_protocol {
37     int           (*handler)(struct sk_buff *skb);
38     void           (*err_handler)(struct sk_buff *skb, u32 info);
39     int           (*gso_send_check)(struct sk_buff *skb);
40     struct sk_buff          *(*gso_segment)(struct sk_buff *skb,
41                             int features);
42     int           no_policy;
43 };
```

```
37 int (*handler)(struct sk_buff *skb)
```

传输层协议数据报接收处理函数指针，当网络层接收 IP 数据报之后，根据 IP 数据报所指示传输层协议，调用对应传输层 net_protocol 结构的该例程接收报文。各传输层接收函数见表 10-4。

表 10-4　传输层的接收函数

传输层协议	接收函数
TCP	tcp_v4_rcv()
UDP	udp_rcv()
IGMP	igmp_rcv()
ICMP	icmp_rcv()

```
38 void (*err_handler)(struct sk_buff *skb, u32 info)
```

在 ICMP 模块中接收到差错报文后，会解析差错报文，并根据差错报文中原始的 IP 首部，调用对应传输层的异常处理函数 err_handler()。各传输层异常处理函数见表 10-5。

表 10-5　传输层异常处理函数

传输层协议	异常处理函数
TCP	tcp_v4_err()
UDP	udp_err()
IGMP	无

```
39 int (*gso_send_check)(struct sk_buff *skb)
40 struct sk_buff *(*gso_segment)(struct sk_buff *skb, int features)
```

GSO 是网络设备支持传输层的一个功能。

当 GSO 数据报输出时到达网络设备，如果网络设备不支持 GSO 的情况，则需要传输层对输出的数据报重新进行 GSO 分段和校验和的计算。因此需要网络层提供接口给设备层，能够访问到传输层的 GSO 分段和校验和的计算功能，对输出的数据报进行分段和执行校验和。

gso_send_check 接口就是回调传输层在分段之前对伪首部进行校验和的计算。

gso_segment 接口就是回调传输层 GSO 分段方法对大段进行分段。

TCP 中实现的函数为 tcp_v4_gso_send_check()和 tcp_tso_segment()，UDP 不支持 GSO。

```
42 int no_policy
```

标识在路由时是否进行策略路由。TCP 和 UDP 默认不进行策略路由。

内核中为 Internet 协议族定义了 4 个 net_protocol 结构实例——icmp_protocol，udp_protocol，tcp_protocol 和 igmp_protocol，分别与 ICMP、UDP、TCP、IGMP 协议一一对应。在 Internet 协议族初始化时，调用 inet_add_protocol()将它们注册到 net_protocol 结构指针数组 inet_protos[MAX_INET_PROTOS]中。在系统运行过程中，随时可以用内核模块加载/卸载方式，调用函数 inet_add_protocol()/inet_del_protocol()将 net_protocol 结构实例注册到 inet_protos[]数组中，或从中删除。

图 10-3 所示为 Internet 协议族中的初始化后的 inet_protos 列表。

完成对 inet_protos[]数组的初始化之后，系统运行时，当有数据报文从网络层流向上层时，就可以通过其上层协议号，在 inet_protos[]数组中找到对应的 net_protocol 结构实例，然后调用该结构中的传输层协议数据报接收处理函数，参见 11.10.1 节。

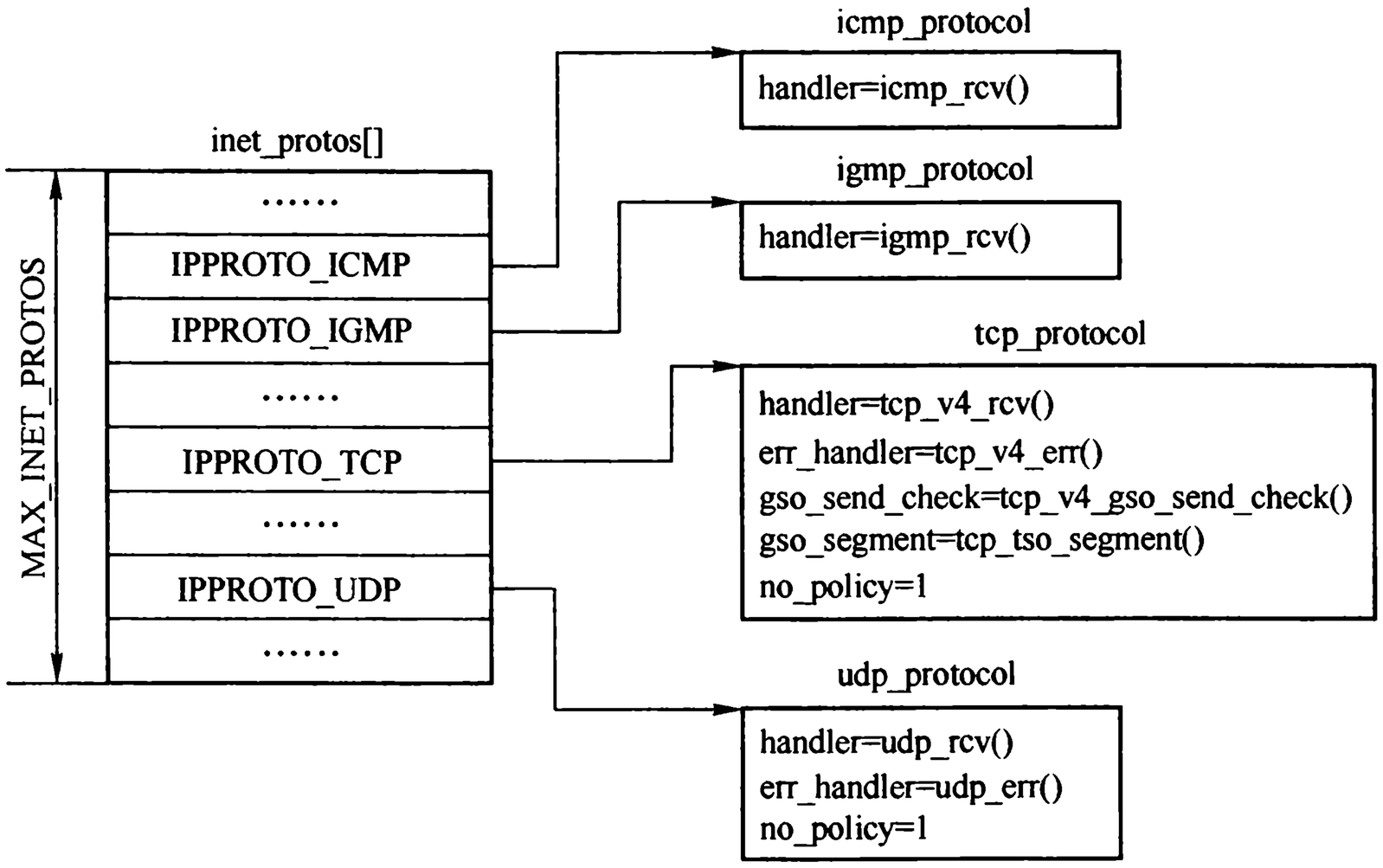

图 10-3　初始化后的 inet_protos 列表

```
199 static inline int ip_local_deliver_finish(struct sk_buff *skb)
200 {
    ... ...
217         hash = protocol & (MAX_INET_PROTOS - 1);
226        if ((ipprot=rcu_dereference(inet_protos[hash]))!= NULL) {
227            int ret;
236            ret = ipprot->handler(skb);
242        } else {
243            if (!raw_sk) {
251            kfree_skb(skb);
252        }
    ... ...
258 }
```

10.4　Internet 协议族的初始化

Internet 协议族的初始化函数为 inet_init()，通过 fs_initcall(inet_init)，将 inet_init 加到内核的初始化列表中，保证了此函数会在系统启动时被调用。

```
1254 static int __init inet_init(void)
1255 {
1256    struct sk_buff *dummy_skb;
1257    struct inet_protosw *q;
1258    struct list_head *r;
1259    int rc = -EINVAL;
1260
1261    BUILD_BUG_ON(sizeof(struct inet_skb_parm) > sizeof(dummy_skb->cb));
1262
1263    rc = proto_register(&tcp_prot, 1);
1264    if (rc)
1265       goto out;
1266
1267    rc = proto_register(&udp_prot, 1);
1268    if (rc)
```

```
1269        goto out_unregister_tcp_proto;
1270
1271    rc = proto_register(&raw_prot, 1);
1272    if (rc)
1273        goto out_unregister_udp_proto;
1274
1275    /*
1276     *    Tell SOCKET that we are alive...
1277     */
1278
1279      (void)sock_register(&inet_family_ops);
1280
1281    /*
1282     *    Add all the base protocols.
1283     */
1284
1285    if (inet_add_protocol(&icmp_protocol, IPPROTO_ICMP) < 0)
1286        printk(KERN_CRIT "inet_init: Cannot add ICMP protocol\n");
1287    if (inet_add_protocol(&udp_protocol, IPPROTO_UDP) < 0)
1288        printk(KERN_CRIT "inet_init: Cannot add UDP protocol\n");
1289    if (inet_add_protocol(&tcp_protocol, IPPROTO_TCP) < 0)
1290        printk(KERN_CRIT "inet_init: Cannot add TCP protocol\n");
1291 #ifdef CONFIG_IP_MULTICAST
1292    if (inet_add_protocol(&igmp_protocol, IPPROTO_IGMP) < 0)
1293        printk(KERN_CRIT "inet_init: Cannot add IGMP protocol\n");
1294 #endif
1295
1296    /* Register the socket-side information for inet_create. */
1297    for (r = &inetsw[0]; r < &inetsw[SOCK_MAX]; ++r)
1298        INIT_LIST_HEAD(r);
1299
1300    for (q = inetsw_array; q < &inetsw_array[INETSW_ARRAY_LEN]; ++q)
1301        inet_register_protosw(q);
1302
1303    /*
1304     *    Set the ARP module up
1305     */
1306
1307    arp_init();
1308
1309      /*
1310       *    Set the IP module up
1311       */
1312
1313    ip_init();
1314
1315    tcp_v4_init(&inet_family_ops);
1316
1317    /* Setup TCP slab cache for open requests. */
1318    tcp_init();
1319
1320    /* Add UDP-Lite (RFC 3828) */
1321    udplite4_register();
1322
1323    /*
1324     *    Set the ICMP layer up
1325     */
```

```
1326
1327    icmp_init(&inet_family_ops);
1328
1329    /*
1330     *    Initialise the multicast router
1331     */
1332 #if defined(CONFIG_IP_MROUTE)
1333    ip_mr_init();
1334 #endif
1335    /*
1336     *    Initialise per-cpu ipv4 mibs
1337     */
1338
1339    if(init_ipv4_mibs())
1340       printk(KERN_CRIT "inet_init: Cannot init ipv4 mibs\n"); ;
1341
1342    ipv4_proc_init();
1343
1344    ipfrag_init();
1345
1346    dev_add_pack(&ip_packet_type);
1347
1348    rc = 0;
1349 out:
1350    return rc;
1351 out_unregister_udp_proto:
1352    proto_unregister(&udp_prot);
1353 out_unregister_tcp_proto:
1354    proto_unregister(&tcp_prot);
1355    goto out;
1356 }
```

1263-1273 初始化 tcp_prot、udp_prot 和 raw_prot 的 slab，并把它们加到 proto_list 链表中，以便支持/proc/net/文件系统。

1279 在套接口层注册 Internet 协议族，即让套接口层支持 Internet 协议族。

1285-1294 将系统中的常用传输层协议及传输层的报文接收例程注册到 inet_protos[]数组中。

1285-1294 Internet 协议族中的所有 net_protocol 结构实例注册 inet_protos[]数组中，参见 10.3 节。

1296-1301 初始化 inetsw 散列表，并调用 net_register_protosw()将数组 inetsw_array[]中 Internet 协议族所有 inet_protosw 实例注册到 inetsw 散列表中，参见 10.2 节。

1307 初始化 ARP 模块，参见第 18 章。

1313 初始化 IP 模块，参见第 11 章。

1315 创建一个内部的 TCP 套接口，主要用来发送 RST 段和 ACK 段。

1318 初始化 TCP 模块，参见第 26 章。

1321 增加对 UDP-Lite 协议的支持，参见 33 章。

1327 初始化 ICMP 模块，参见第 14 章。

1333 初始化组播路由模块，参见第 15 章。

1339 初始化 IPv4 的 MIB，本书不作论述。

1342 初始化/proc/net 文件系统。

1344 初始化 IP 分片模块，参见第 13 章。

1346 注册 Internet 协议族报文类型及报文接收处理函数。

第 11 章　IP：网际协议

11.1　引言

IP 是 TCP/IP 协议族中最为核心的协议，所有的 TCP、UDP、ICMP 及 IGMP 数据都是以 IP 数据报的形式传输的，见图 11-1。

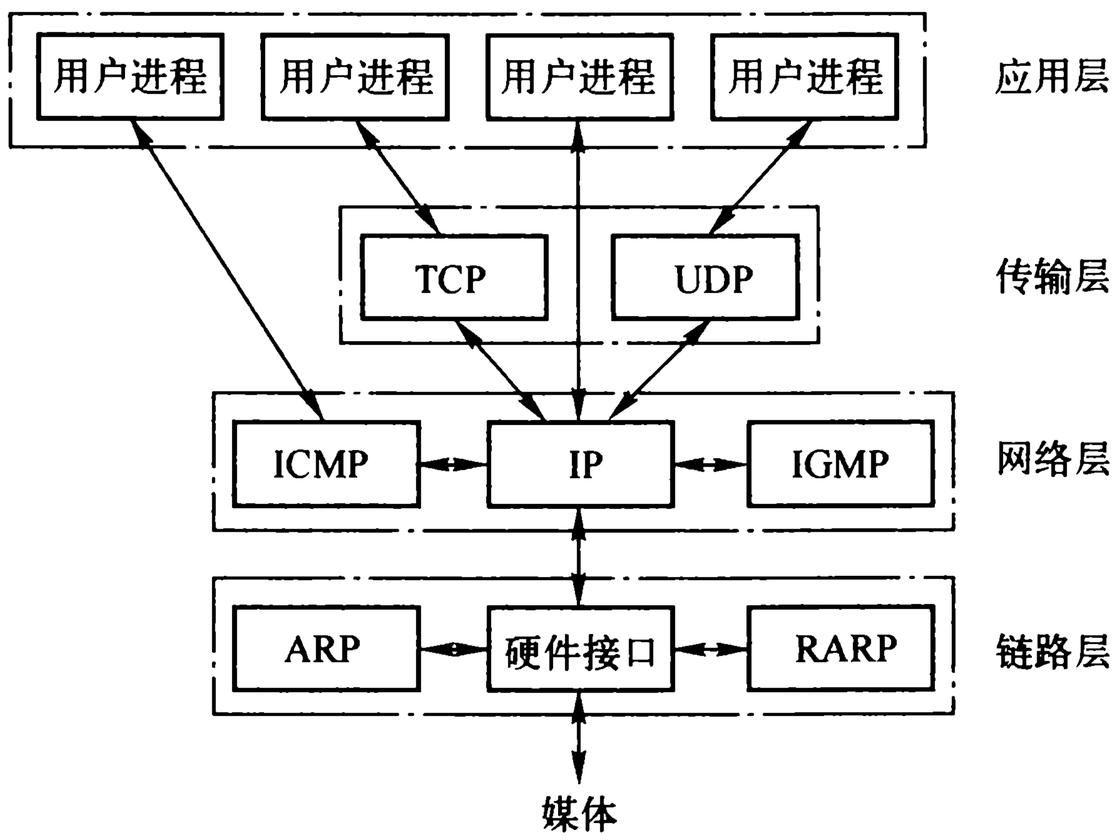

图 11-1　IPv4 协议族中不同层次的协议

IP 是一种不可靠的协议，也就是说，它并不能保证每个 IP 数据报都能成功地到达目的地，而只是提供最好的传输服务。如果发生某种错误（例如：某个路由器暂时用完了缓冲区），IP 有一个简单的错误处理算法，即丢弃该数据报，然后发送 ICMP 消息报给发送方。每个数据报的处理是相互独立的，因此 IP 数据报可以不按发送顺序接收。任何可靠性都必须由上层协议来提供，如 TCP。

IP 数据报的输入、输出和转发，数据报控制信息处理以及对端信息块的管理涉及以下文件：

- include/net/ip.h，定义 IP 层相关的结构、宏和函数原型。
- include/linux/inetdevice.h，定义 IPv4 专用的网络设备相关的结构、宏等。
- include/linux/errqueue.h，定义差错处理相关的结构。
- include/net/inetpeer.h，定义对端信息块的结构、宏和函数原型。
- include/net/dst.h，定义目的路由缓存相关结构、宏和函数原型。
- net/ipv4/ip_output.c，IP 数据报的输出。
- net/ipv4/ip_sockglue.c，IP 层套接口选项。
- net/ipv4/ip_input.c，IP 数据报的输入。
- net/ipv4/ip_forward.c，IP 数据报的转发。
- net/ipv4/inetpeer.c，对端信息块的管理。
- net/ipv4/af_inet.c，网络层和传输层接口。

11.1.1 IP 首部

图 11-2 为 IP 数据报的格式，不包含选项字段的 IP 首部长为 20B。

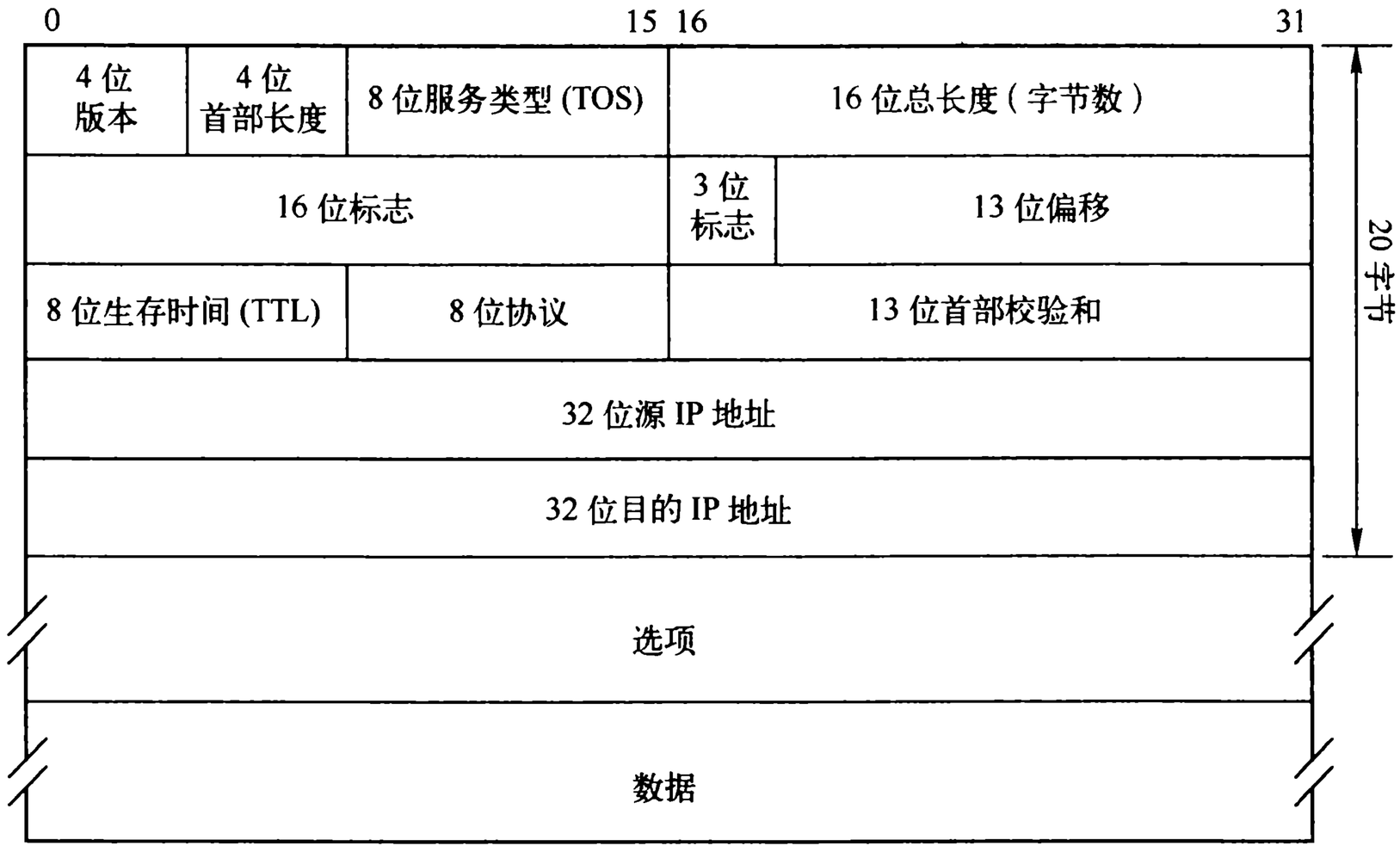

图 11-2 IP 数据报格式

由于 TCP/IP 首部中所有的二进制数据在网络中都是以网络字节序传输的，因此以非网络字节序存储二进制数据的机器，必须在传输数据之前把首部转换成网络字节序。

（1）协议版本号，目前最常用的 IP 版本号还是 4，称为 IPv4。

（2）首部长度，首部长度指的是首部（包括所有选项）占有的 32 位字数。该段占 4 位，因此首部最长可达 60B。对没有任何选项的 IP 数据报，该字段值是 5。

（3）服务类型，服务类型（TOS）字段包括一个现已被忽略的 3 位的优先权子字段，4 位的 TOS 子字段和一个未用但必须置为 0 的位。TOS 的 4 位中的每一位分别代表最小时延、最大吞吐量、最高可靠性和最小费用，如果 4 位均为 0，那么就意味着是一般服务。

（4）总长度，总长度字段是指整个 IP 数据报的长度，以字节为单位。利用首部长度字段和总长度字段，就可以知道 IP 数据报中数据内容的起始位置和长度。总长度字段是必要的，因为一些数据链路（如以太网），需填充一些数据以达到最小长度。尽管以太网的最小帧长为 46B，但 IP 数据有可能会更短。如果没有总长度字段，那么 IP 层就不知道 46B 中有多少是 IP 数据报的数据。由于该字段长为 16 位，因此 IP 数据报最长可达 65535B。尽管可以传送一个 65535B 长的 IP 数据报，但是大多数的链路层都会对它进行分片，而且主机也要求不能接收超过 576B 的数据报。由于 TCP 会把用户数据分成若干片，因此一般来说这个限制不会影响 TCP。但对 UDP 的应用，如 RIP，TFTP，BOOTP，DNS，以及 SNMP，它们通常都限制用户数据报长度为 512B，小于 576B。当数据报被分片时，该字段值会随之变化。

（5）标识，标识字段唯一标识主机发送的每一份数据报。通常每发送一份数据报其值就会递增 1。

（6）TTL 生存时间，TTL（time-to-live）生存时间字段设置了数据报可以经过的最多路由器数，指定了数据报的生存时间。TTL 的初始值由源主机设置，通常为 32 或 64，数据报每经过一个路由器，其值就减去 1，直至 0 时，丢弃数据报，并发送 ICMP 报文通知源主机。

（7）协议，协议字段可以识别是哪个传输层协议通过 IP 传送数据。

（8）首部检验和，首部检验和字段是根据 IP 首部计算的检验和，而不对首部后面的数据进

行计算。传输层协议 ICMP、IGMP、UDP 和 TCP 在各自的首部中均含有同时覆盖首部和数据的检验和。

计算一份数据报 IP 检验和的方法如下：首先把检验和字段置为 0；然后将整个首部看成由一串 16 位字组成，对其中的每个 16 位进行二进制反码求和，结果存在检验和字段中。当收到一份 IP 数据报后，同样对首部中每个 16 位字进行二进制反码的求和。由于接收方在计算过程中包含了发送方存在首部中的检验和，因此，如果首部在传输过程中没有发生任何差错，那么接收方计算的结果应该为全 1。如果结果不是全 1，即是检验和错误，IP 丢弃收到的数据报，但是不生成差错报文，而由上层去发现丢失的数据报确定是否要进行重传。ICMP、IGMP、UDP 和 TCP 都采用相同的检验和算法。

（9）选项，是数据报中的一个可变长的可选信息，参见第 12 章。

11.1.2　IP 数据报的输入与输出

网络层处于传输层和链路层之间，同时还需要与路由表以及邻居子系统打交道。在输入数据时，提供输入接口给链路层调用，并调用传输层的输入接口将数据传递到传输层。在输出数据时，提供输出接口给传输层调用，并调用链路层的输出接口将数据输出到链路层。输入和输出过程中，都需要查找路由，通过 netfilter 处理等操作。图 11-3 显示了 IP 层主要函数之间的调用关系。

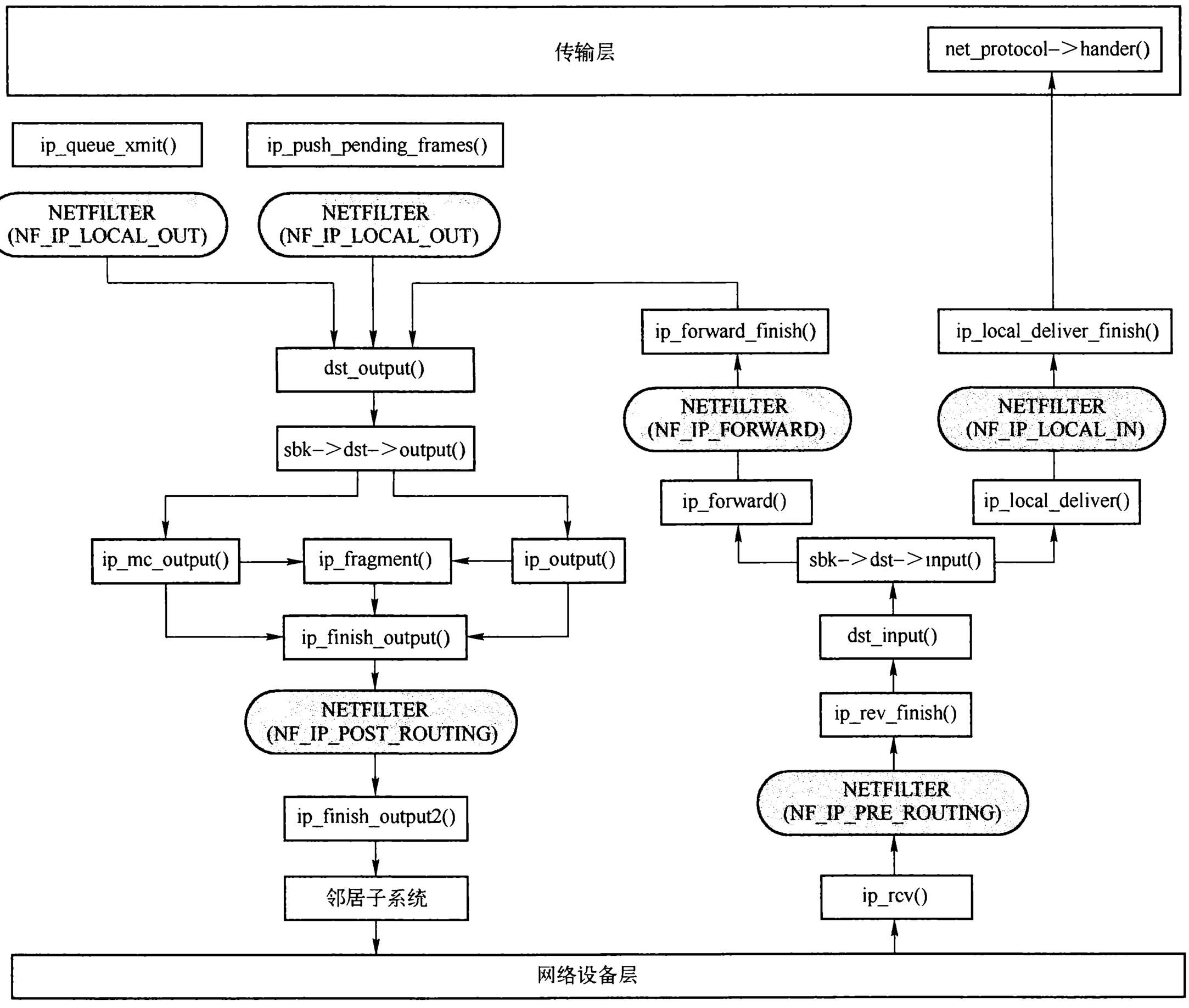

图 11-3　IP 层主要函数调用关系

11.2 IP 的私有信息控制块

IP 层在 SKB 中有个信息控制块 inet_skb_parm 结构，存储在 skb_buff 结构的 cb 成员中。IP 层用宏 IPCB 访问该结构以增强代码的可读性。这个私有的信息控制块主要存储 IP 选项，及在 IP 处理中需要设置的标志。在 IP 层，无论是输入还是输出都需要处理 IP 选项。例如，在输入时，ip_rcv_options()会解析 IP 首部中的选项保存到 inet_skb_parm 结构的 opt 成员中；在输出时，ip_options_build()会根据 inet_skb_parm 结构的 opt 将其组织后在 IP 首部中生成选项，而在转发时，ip_forward_options()会根据选项作适当的处理。相关标志会在处理过程中根据处理的状况进行设置，例如 IPSKB_FORWARDED 标志会在组播数据报转发后设置，而 IPSKB_FRAG_COMPLETE 标志则会在 IP 分片完成后对每个分片进行设置。

```
34 struct inet_skb_parm
35 {
36     struct ip_options    opt;        /* Compiled IP options        */
37     unsigned char        flags;
38
39 #define IPSKB_FORWARDED       1
40 #define IPSKB_XFRM_TUNNEL_SIZE    2
41 #define IPSKB_XFRM_TRANSFORMED    4
42 #define IPSKB_FRAG_COMPLETE    8
43 #define IPSKB_REROUTED       16
44 };
    ... ...
53 #define IPCB(skb) ((struct inet_skb_parm*)((skb)->cb))
```

36 struct ip_options opt

IP 选项，接收 IP 数据报时解析 IP 首部中的选项到该字段，而发送 IP 数据报时根据该字段在 IP 首部中生成选项，参见 12.2 节。

37 unsigned char flags

处理 IP 数据报时的一些标志，见表 11-1。

表 11-1 flags 的取值

flags	描述
IPSKB_FORWARDED	组播包已经转发过
IPSKB_XFRM_TUNNEL_SIZE	IPSEC 中，按安全路由链表的安全路由处理数据时会检测 IP 数据报的尺寸，并设置 IPSKB_XFRM_TUNNEL_SIZE 标志，以后不会再对其进行检测
IPSKB_XFRM_TRANSFORMED	IPSEC 中，按安全路由链表的安全路由处理数据后设置 IPSKB_XFRM_TRANSFORMED 标志，之后有该标志的数据包，在 NAT 操作后将不会对存在该标志的数据包再进行特殊检查
IPSKB_FRAG_COMPLETE	完成分片
IPSKB_REROUTED	IPSEC 中，每执行一次 IPSEC 封装处理过程，封装结束后的数据包会设置 IPSKB_REROUTED 标志。标识 IPSKB_REROUTED 的数据报不能再进行转发

11.3 系统参数

系统参数如下：

- ip_default_ttl，设置 IP 数据报的默认生存时间值。对于某个套接口发出的数据报可以通

过 IP_TTL 选项来设置。默认值为 64，一般来说，这是比较合适的，但如果网络规模较大，则需增大该值。

- ip_dynaddr，用于在拨号连接情况下，使系统可将 IP 数据报的源地址修改为一个新的 IP 地址，同时中断原有 TCP 会话，用新地址重新发送 SYN 请求包，开始一个新的 TCP 会话。当发生 IP 欺骗时，该参数可使系统立即改变伪装地址，见表 11-2。

表 11-2　ip_dynaddr 的取值

ip_dynaddr	描述
0（默认）	禁止动态地址功能
1	启用动态地址功能
2	使用冗余模式启用动态地址功能

- ip_forward，标识内核是否转发 IP 数据报，见表 11-3。

表 11-3　ip_forward 的取值

ip_forward	描述
0（默认）	禁止转发 IP 数据报
1	允许转发 IP 数据报

- ip_local_port_range，在自动选择绑定端口时，TCP 和 UDP 使用本地端口的范围，TCP 和 UDP 自动绑定端口号在它们之间（包括两端）。根据系统中的内存容量，配置建议值见表 11-4。

表 11-4　ip_local_port_range 的配置值

主机内存容量	配置值
内存容量>128MB	32768～61000
内存容量<128MB	1024～4999 或者更少

用于向外连接的端口范围默认情况下其实很小，为 1024~4999。如果在启用 tcp_tw_recycle 时，在每秒建立 2000 个连接时，这个范围是不够的。

- ip_no_pmtu_disc，在传输路径上检测出的最大可能的 MTU，即路径 MTU，见表 11-5。在默认情况下不对 TCP 套接口执行路径 MTU 发现。如果在路径上配置了防火墙（例如，用来丢弃所有 ICMP 报文，或者误配置了接口，如设置了一个两端 MTU 不同的端对端连接），路径 MTU 发现有可能失败。

表 11-5　ip_no_pmtu_disc 的取值

ip_no_pmtu_disc	描述
0	启用路径 MTU 发现功能
1	禁止启用路径 MTU 发现功能

- ip_nonlocal_bind，标识是否允许进程绑定到非本地地址。默认值为 0，表示禁止进程绑定到非本地地址。IP_FREEBIND 套接口选项也有相同的功能，只是前者控制整个

系统而后者只控制当前该套接口。

- forwarding，标识具体网络设备是否启用了 IP 数据报转发功能。另请参见 devinet_sysctl_forward()。
- mc_forwarding，标识是否启用组播路由功能。若要启用该功能，则必须在编译内核时启用 CONFIG_IP_MROUTE 选项，并且启动了组播路由服务进程。可配置到整个系统或具体网络设备。
- accept_redirects，标识是否接受 ICMP 重定向消息。可配置到整个系统或具体网络设备。
- secure_redirects，标识是否接收只来自网关的 ICMP 重定向消息。可配置到整个系统或具体网络设备。默认值为 1，表示接收。此参数与 shared_media 之间的关系见表 11-6。
- shared_media，标识对路由器是否发送，对主机是否接收 RFC1620 共享媒体重定向消息。可配置到整个系统或具体网络设备。默认值为 1，表示启用。此参数优先于 secure_redirects，与 secure_redirects 之间的关系见表 11-6。

表 11-6 shared_media 与 secure_redirects 的关系

shared_media	secure_redirects
启用	secure_redirects 失效，在接收到 ICMP 重定向消息后不必再检测 secure_redirects
禁用	secure_redirects 正常启用，在接收到 ICMP 重定向消息后检测 secure_redirects

- rp_filter，标识是否对数据报的源地址进行检查，即反向路径回溯源地址验证，以防止外部攻击者使用 IP 欺骗进入内部网络。如果在允许 IP 数据报转发的情况下没有打开该设置，将易导致受到来自外部的 IP 欺骗。默认值为 0，不启用该功能，但有些程序可能会在启动时自动将其打开。可配置到整个系统或具体网络设备。
- send_redirects，标识路由器是否可以发送重定向消息。默认值为 1，表示启用。可配置到整个系统或具体网络设备。
- accept_source_route，标识是否接收存在源路由选项的 IP 数据报。默认值通常根据不同的角色使用不同的值，路由器为 1，主机为 0。可配置到整个系统或具体网络设备。
- medium_id，用来区分不同的媒介，见表 11-7。不同的网络设备可以使用不同的值，使其中只有一个能接收到广播包。可配置到整个系统或具体网络设备。

proxy_arp 的特性是允许 ARP 报文在两个不同的网络介质中转发。通常，该参数被用来配合 proxy_arp 实现。

表 11-7 medium_id

medium_id	描述
0（默认）	表示网络介质接受其上的媒介
-1	表示未知媒介

- bootp_relay，标识是否接收源地址为 0.b.c.d，且目的地址不是本地的数据报。如果启用，BOOTP 中继服务进程会转发这样的数据报。默认值为 0，表示禁止此功能。可配置到整个系统或具体网络设备。
- log_martians，标识当遇到错误的 IP 地址时，记录到内核的日志中。需在路由模块中打开 CONFIG_IP_ROUTE_VERBOSE 编译选项才能使用该参数。可配置到整个系统或具

体网络设备。

- tag，可以根据实际情况设置需要的值，没有特定用途，默认值为 0。
- disable_xfrm，标识是否禁止 IPSEC 加密，本书不作论述。可配置到整个系统或具体网络设备。
- disable_policy，标识是否禁止 IPSEC 策略，本书不作论述。可配置到整个系统或具体网络设备。
- promote_secondaries，标识在删除主地址时，第二 IP 地址是否能升级为主 IP 地址，见表 11-8。

表 11-8　promote_secondaries

promote_secondaries	描述
0	当主 IP 地址被删除时，第二 IP 地址也被删除
1	当主 IP 地址被删除时，第二 IP 地址成为新的主 IP 地址

11.4　初始化

ip_init()用来初始化 IP 层，包括初始化路由模块，对端信息管理模块和组播 proc 文件的注册。系统启动时，由初始化网络子系统的 inet_init()调用该函数。

11.5　IP 层套接口选项

在第 24 章会说明 IP 层套接口选项入口为 ip_setsockopt()。该函数会首先判断选项的级别，如果不是 SOL_IP 级别，则返回无效协议，否则调用 do_ip_setsockopt()进行具体的选项处理。而在 do_ip_setsockopt()中，组播路由相关的选项由 ip_mroute_setsockopt()来处理，其他选项则由该函数自己来处理。

IP 层套接口选项如下：

（1）IP_OPTIONS。设置或获取将由套接口发送的每个数据报 IP 首部中的 IP 选项，最长可达 40B。该参数是一个指向数据报含选项和选项长度的存储缓冲区的指针。可能出现的选项包括：

1）安全和控制限制：RFC 1108。

2）记录路由：每个路由器都会将自己的 IP 地址添加到转发数据报的首部中。

3）时间标志（时间戳）：每个路由器都会增添各自的时间戳。

4）松散源路由选择：访问路径中需包含选项内列出的 IP 地址。

5）严格源路由选择：只能访问选项内列出的 IP 地址。

需注意的是，并不是所有的主机和路由器都支持所有这些选项。

设置 IP 选项的代码如下，会用到 IP 选项信息块，其类型为 ip_options 结构，ip_options 结构参见 12.2 节。

```
444         case IP_OPTIONS:
445         {
446             struct ip_options * opt = NULL;
```

```
447             if (optlen > 40 || optlen < 0)
448                goto e_inval;
449             err = ip_options_get_from_user(&opt, optval, optlen);
450             if (err)
451                break;
452             if (inet->is_icsk) {
453                struct inet_connection_sock *icsk = inet_csk(sk);
454 #if defined(CONFIG_IPV6) || defined(CONFIG_IPV6_MODULE)
455                if (sk->sk_family == PF_INET ||
456                    (!((1 << sk->sk_state) &
457                       (TCPF_LISTEN | TCPF_CLOSE)) &&
458                     inet->daddr != LOOPBACK4_IPV6)) {
459 #endif
460                   if (inet->opt)
461                      icsk->icsk_ext_hdr_len -= inet->opt->optlen;
462                   if (opt)
463                      icsk->icsk_ext_hdr_len += opt->optlen;
464                   icsk->icsk_sync_mss(sk, icsk->icsk_pmtu_cookie);
465 #if defined(CONFIG_IPV6) || defined(CONFIG_IPV6_MODULE)
466                }
467 #endif
468             }
469             opt = xchg(&inet->opt, opt);
470             kfree(opt);
471             break;
472          }
```

447-448 检测设置值的长度，IP OPTIONS 选项要求长度不能超过 40B。

449-451 调用 ip_options_get_from_user()分配存放 IP_OPTIONS 的空间，并将选项数据从用户空间复制到此缓存。

452-458 如果是基于连接的传输控制块，则需根据原 IP 选项和待设置 IP 选项的长度调整传输控制块中标识 IP 首部选项长度的 icsk_ext_hdr_len。

469-471 最后将待设置的 IP 选项信息块设置到传输控制块中，并释放原先的 IP 选项信息块。

（2）IP_PKTINFO。控制是否允许通过 IP_PKTOPTIONS 选项或 recvmsg 系统调用来获取与本端地址等信息相关的 IP_PKTOPTIONS 选项。

（3）IP_RECVTTL。控制是否允许通过 IP_PKTOPTIONS 选项或 recvmsg 系统调用来获取发送组播数据报的 TTL。

（4）IP_RECVTOS。控制是否允许通过 recvmsg 系统调用来获取数据报的 TOS，目前不支持 SOCK_STREAM 套接口。

（5）IP_RECVOPTS。控制是否允许通过 recvmsg 系统调用来获取当前 IP 数据报首部中的 IP 选项，目前不支持 SOCK_STREAM 套接口。

（6）IP_RETOPTS。类似于 IP_RECVOPTS 选项，不同的是获取未处理时间戳选项和记录路由选项的 IP 选项。

（7）IP_PASSSEC。控制是否允许通过 recvmsg 系统调用获取有关安全的信息，Linux 2.6.20 不支持 SOCK_STREAM 套接口。

（8）IP_TOS。设置或获取从该套接口输出 IP 数据报的服务类型（TOS）字段。用来在网络上区分包的优先级。TOS 是个单字节的字段，表 11-9 为已定义的标准，最多只能声明这些 TOS 值中的一个，其他的都无效，应清除。

表 11-9　TOS

TOS	描述
IPTOS_LOWDELAY	用于交互式通信，尽量减少时间延迟
IPTOS_THROUGHPUT	用来优化吞吐量
IPTOS_RELIABILITY	用来作可靠性优化
IPTOS_MINCOST	被用作“填充数据”

默认情况下，数据报的 TOS 字段用 IPTOS_LOWDELAY 来填充。应用程序设置这个选项时，需要有 CAP_NET_ADMIN 能力。优先级也能以与协议无关的方式通过套接口层的 SO_PRIORITY 选项来设置。

（9）IP_TTL。设置输出 IP 数据报的生存时间，有效值为 1 至 255。IP 数据报每经过一个路由器，路由器都会判断 TTL 值，并将该值递减 1，一旦发现其为 0，即丢弃数据报，以免由于路由循环，造成数据报无休止地在网络中“游荡”。

（10）IP_HDRINCL。若将 IP_HDRINCL 选项设置为 1，则发送函数会将 IP 首部作为数据的一部分包含在发送数据的前部，并促使接收函数也将 IP 首部作为数据的一部分。因此，在发送的时候，必须在数据前部包括完整的 IP 首部，并正确填写其中的每个字段。此选项只对 RAW 套接口有效。

（11）IP_MTU_DISCOVER。标识某个套接口是否启用路径 MTU 发现功能。

系统级别的默认值是这样的：STREAM 套接口由系统 ip_no_pmtu_disc 控制，而其他套接口都被都屏蔽了。对于非 STREAM 套接口而言，用户应该按照 MTU 的大小对数据分块并在必要的情况下进行重传。如果设置了该标识，内核会拒绝比已知路径 MTU 大的数据报。

当允许路径 MTU 发现时，内核会自动记录每个目的主机的路径 MTU。为了获得连接的路径 MTU 初始估计值，可用connect()将一个数据报套接口连接到目的地址，并调用带 IP_MTU 选项的getsockopt()获取该值。参见 25.2.3 节的传输控制块的 pmtudisc 成员。

（12）IP_RECVERR。标识是否允许接收扩展的可靠错误信息。如果在数据报套接口或 RAW 套接口上设置了该选项，则所有产生的错误会加入到套接口的错误队列中。用户可以通过调用设置了 MSG_ERRQUEUE 标识的recvmsg()来接收套接口操作中接收到的错误，包括 SOL_IP 级别 IP_RECVERR 类型错误，以及用 sock_extended_err 结构描述的辅助错误信息。该选项对非连接的套接口处理错误非常有用。

在 STREAM 套接口上，IP_RECVERR 会有细微的不同。它并不保存下次超时的错误，而是立即传递所有进来的错误给用户。这对 TCP 连接时间很短的情况很有用，因为它要求快速的错误处理。使用该选项要小心，因为不允许从路由转移和其他正常条件下正确地进行恢复，它使得 TCP 变得不可靠，并且破坏了协议的规范。注意 TCP 没有错误队列；MSG_ERRQUEUE 对于 STREAM 套接口是非法的，因此所有错误都会由套接口函数返回，或者只返回 SO_ERROR。

对于 RAW 套接口而言，IP_RECVERR 允许传递所有接收到的 ICMP 错误给应用程序。

（13）IP_MTU。获取当前套接口的当前已知路径 MTU，只在套接口连接后才有效。

```
1031         case IP_MTU:
1032         {
1033             struct dst_entry *dst;
1034             val = 0;
1035             dst = sk_dst_get(sk);
```

```
1036            if (dst) {
1037                val = dst_mtu(dst);
1038                dst_release(dst);
1039            }
1040            if (!val) {
1041                release_sock(sk);
1042                return -ENOTCONN;
1043            }
1044            break;
1045        }
```

（14）IP_ROUTER_ALERT。IP 路由器警告选项，只对 RAW 套接口有效。对用户空间中的诸如 RSVP 后台守护进程之类的程序很有用。带有 IP 路由器警告选项的 IP 数据报不会在内核中被转发，而是由应用程序去转发它们，参见 12.11 节。

（15）IP_PKTOPTIONS。设置了 IP_PKTINFO 或 IP_RECVTTL 选项后，可通过此选项获取相关的信息。该选项只对 STREAM 类型的套接口有效。

设置了 IP_PKTINFO 选项，可获取以 pktinfo 结构存储的辅助信息。

```
struct in_pktinfo
{
    int         ipi_ifindex;
    struct in_addr    ipi_spec_dst;
    struct in_addr    ipi_addr;
};
```

ipi_ifindex 为发送组播数据报的网络设备的索引号。

ipi_spec_dst 和 ipi_addr 都是本地的源地址。

对于只能通过 recvmsg 系统调用来获取相应信息的 DGRAM 和 RAW 类型的套接口，in_pktinfo 结构中这两个字段的意义会有所不同。

设置 IP_RECVTTL 选项，可以获取发送辅助信息组播数据报的 TTL。而对于只能通过 recvmsg 获取相应信息的 DGRAM 和 RAW 类型套接口，获取的不是组播数据报的 TTL，而是普通 IP 数据报的 TTL。

```
1104        case IP_PKTOPTIONS:
1105        {
1106            struct msghdr msg;
1107
1108            release_sock(sk);
1109
1110            if (sk->sk_type != SOCK_STREAM)
1111                return -ENOPROTOOPT;
1112
1113            msg.msg_control = optval;
1114            msg.msg_controllen = len;
1115            msg.msg_flags = 0;
1116
1117            if (inet->cmsg_flags & IP_CMSG_PKTINFO) {
1118                struct in_pktinfo info;
1119
1120                info.ipi_addr.s_addr = inet->rcv_saddr;
1121                info.ipi_spec_dst.s_addr = inet->rcv_saddr;
```

```
1122                info.ipi_ifindex = inet->mc_index;
1123                put_cmsg(&msg, SOL_IP, IP_PKTINFO, sizeof(info), &info);
1124            }
1125            if (inet->cmsg_flags & IP_CMSG_TTL) {
1126                int hlim = inet->mc_ttl;
1127                put_cmsg(&msg, SOL_IP, IP_TTL, sizeof(hlim), &hlim);
1128            }
1129            len -= msg.msg_controllen;
1130            return put_user(len, optlen);
1131        }
```

（16）IP_FREEBIND。用于标识是否允许绑定非本地主机地址。

（17）IP_IPSEC_POLICY 和 IP_XFRM_POLICY。用来设置 IPSEC 相关策略，设置进程需有 CAP_NET_ADMIN 能力，本书不作论述。

11.6　ipv4_devconf 结构

ipv4_devconf 结构是网络设备接口的 IPv4 系统配置。在内核中有一个名为 ipv4_devconf 的系统全局变量，该配置对所有接口有效。另外，每个网络设备的 IP 控制块中也都存在一份配置，但该配置只对所在网络设备有效。

```
11 struct ipv4_devconf
12 {
13     int   accept_redirects;
14     int   send_redirects;
15     int   secure_redirects;
16     int   shared_media;
17     int   accept_source_route;
18     int   rp_filter;
19     int   proxy_arp;
20     int   bootp_relay;
21     int   log_martians;
22     int   forwarding;
23     int   mc_forwarding;
24     int   tag;
25     int   arp_filter;
26     int   arp_announce;
27     int   arp_ignore;
28     int   arp_accept;
29     int   medium_id;
30     int   no_xfrm;
31     int   no_policy;
32     int   force_igmp_version;
33     int   promote_secondaries;
34     void  *sysctl;
35 };
```

13 int accept_redirects

标识是否接收 ICMP 重定向报文。

14 int send_redirects

标识是否启用 ICMP 重定向报文输出。

15 int secure_redirects

标识是否接收 ICMP 重定向报文，但只针对具有路由功能的网关。

16 int shared_media

标识是否启用发送(路由器)或接收(主机)RFC1620 共享媒体重定向。

17 int accept_source_route

标识是否接收带有 SRR 选项的数据报。

18 int rp_filter

标识是否启用通过反向路径回溯进行源地址验证(在 RFC1812 中定义)。

19 int proxy_arp

标识是否启用 arp 代理功能。

20 int bootp_relay

标识是否接收源地址为 0.b.c.d，目的地址不是本机的数据报。用来支持 BOOTP 转发服务进程，该进程将捕获并转发该包。目前尚未实现。

21 int log_martians

标识是否记录那些非法地址的数据报到内核日志中。

22 int forwarding

在 ipv4_devconf 中，标识是否启用 IP 数据报转发功能；而在 IP 配置块中，标识是否启用所在网络设备的 IP 数据报转发功能。

23 int mc_forwarding

在 ipv4_devconf 中，标识是否进行组播路由；而在 IP 配置块中，表示当前虚拟接口数。

24 int tag

留给用户根据需要设置。

25 int arp_filter

允许从其他网络设备输出 ARP 应答，参见 18.2 节和 18.11 节。

26 int arp_announce

输出 ARP 请求时，由 IP 数据报确定源 IP 地址的规则，参见 18.2 节和 18.7 节。

27 int arp_ignore

接收 ARP 请求报文的过滤规则，参见 18.2 节和 18.7 节。

28 int arp_accept

处理非 ARP 请求而接收到的 ARP 应答，参见 18.2 节和 18.11 节。

29 int medium_id

用来区分不同的媒介，参见 11.3 节中的 medium_id 系统参数。

30 int no_xfrm

标识是否启用 XFRM，只用于 IPsec，本书不作论述。

31 int no_policy

标识是否启用策略路由。

32 int force_igmp_version

强制当前启用的 IGMP 版本。

33 int promote_secondaries

标识在删除主地址时，第二 IP 地址是否能升级为主 IP 地址，见 11.3 节中的 promote_secondaries

系统参数。

11.7　套接口的错误队列

在传输控制块中有一个用于保存错误信息的队列 sk_error_queue，当 ICMP 接收到差错信息或者 UDP 套接口和 RAW 套接口输出报文出错时，会产生描述错误信息的 SKB 添加到该队列上。应用程序为能通过系统调用获取详细的错误信息，需要设置 IP_RECVERR 套接口选项，之后可通过参数 flags 为 MSG_ERRQUEUE 的 recvmsg 系统调用来获取详细的出错信息。

UDP 套接口和 RAW 套接口在调用 recvmsg 接收数据时，可以设置 MSG_ERRQUEUE 标志，只从套接口的错误队列上接收错误而不接收其他数据，参见 33.16 节。而实现这个功能是通过调用 ip_recv_error()来完成的。

有关错误信息的数据流及函数调用关系见图 11-4。

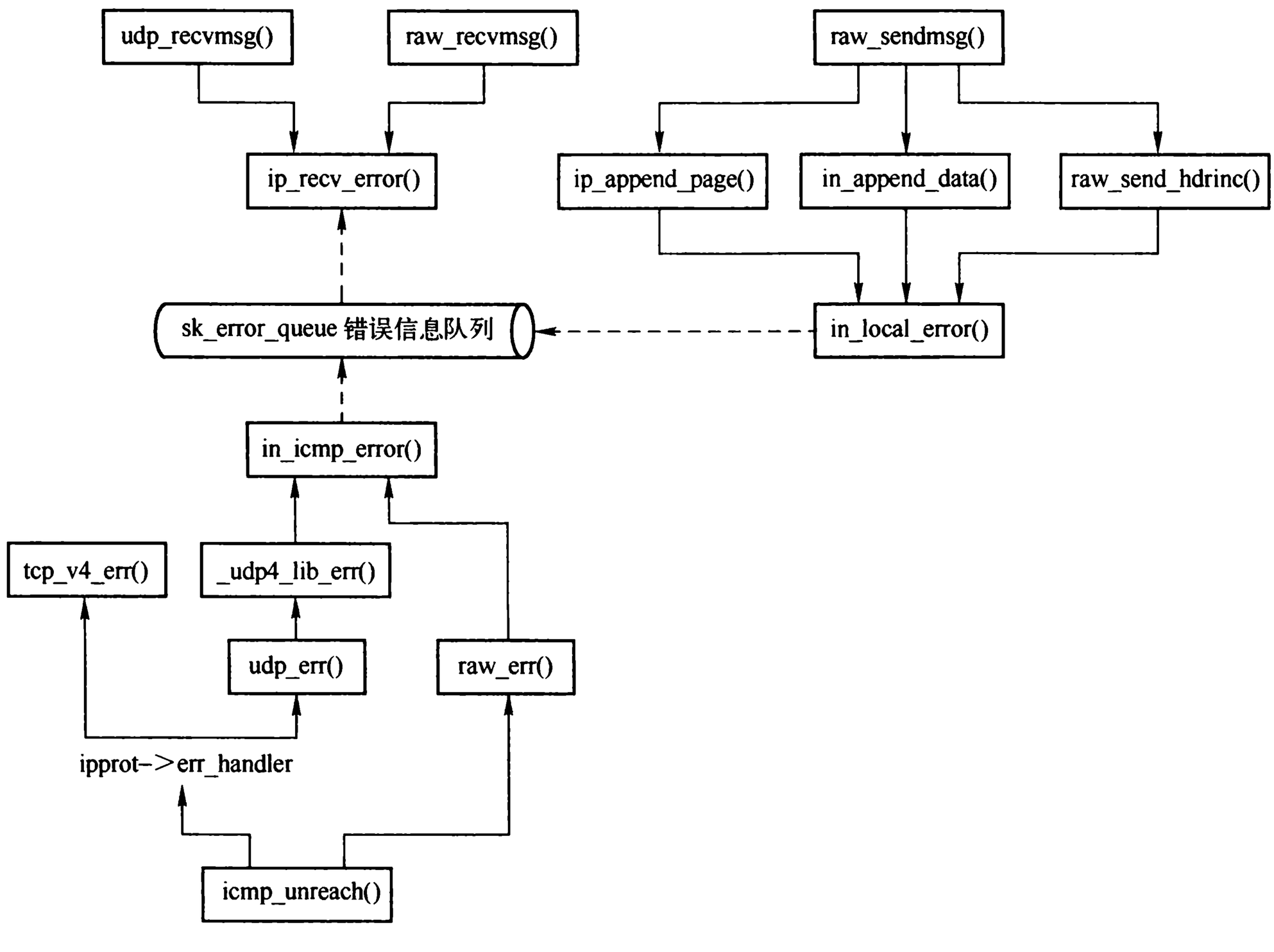

图 11-4　错误信息的数据流及函数调用关系

在基于连接的套接口上，IP_RECVERR 意义则会有所不同。并不保存错误信息到错误队列中，而是立即传递所有接收到的错误信息给用户进程。这对基于短连接的 TCP 应用是很有用的，因为 TCP 要求快速的错误处理。需要注意的是，TCP 没有错误队列，MSG_ERRQUEUE 对于基于连接的套接口是无效的。

错误信息传递给用户进程时，并不将错误信息作为报文的内容传递给用户进程，而是以错误信息块的形式保存在 SKB 控制块中，通常通过 SKB_EXT_ERR 来访问 SKB 控制块中的错误

信息块。

```
#define SKB_EXT_ERR(skb) ((struct sock_exterr_skb *) ((skb)->cb))
```

错误信息块定义为 sock_exterr_skb 结构，由于该错误也在 IP 层中处理，为了与 IP 控制块兼容，错误信息块的前部由 IP 控制块组成，之后才是其出错信息。

```
31 struct sock_exterr_skb
32 {
33     union {
34         struct inet_skb_parm    h4;
35 #if defined(CONFIG_IPV6) || defined (CONFIG_IPV6_MODULE)
36         struct inet6_skb_parm    h6;
37 #endif
38     } header;
39     struct sock_extended_err    ee;
40     u16                addr_offset;
41     __be16                port;
42 };
```

33 union {} header

与 IP 控制块兼容，可以存储 IP 选项信息。

39 struct sock_extended_err ee

记录出错信息，参见以下 sock_extended_err 结构的描述。

```
04 struct sock_extended_err
05 {
06     __u32    ee_errno;
07     __u8    ee_origin;
08     __u8    ee_type;
09     __u8    ee_code;
10     __u8    ee_pad;
11     __u32    ee_info;
12     __u32    ee_data;
13 };
```

06 __u32 ee_errno

出错信息的错误码。

07 __u8 ee_origin

标识出错信息的来源，见表 11-10。

表 11-10　ee_origin 的取值

ee_origin	描述
SO_EE_ORIGIN_LOCAL	出错信息来自本地
SO_EE_ORIGIN_ICMP	出错信息来自 ICMP 消息

08 __u8 ee_type

在出错信息来自 ICMP 消息的情况下，标识 ICMP 差错消息的类型；其他来源均为 0。

09 __u8 ee_code

在出错信息来自 ICMP 消息的情况下，标识 ICMP 差错消息的编码；其他来源均为 0。

```
10 __u8 ee_pad
```

目前未使用，填充为 0。

```
11 __u32 ee_info
```

出错信息的扩展信息，其意义随出错信息的错误码具体而定。例如，当接收到目的不可达需要分片差错信息时，为下一跳的 MTU。

```
12 __u32 ee_data
```

目前未使用，填充为 0。

```
40 u16 addr offset
```

导致出错的原始数据报的目的地址在负载 ICMP 报文的 IP 数据报中的偏移量。

```
41 __be16 port
```

对于 UDP 是出错报文的目的端口，而其他情况都为 0。

11.7.1　添加 ICMP 差错信息

当接收到 ICMP 差错信息时，ICMP 模块会根据出错的原始数据报的传输层协议，调用传输层的差错处理例程，而传输层的差错处理例程会进而调用 ip_icmp_error()将出错信息添加到输出该出错数据报的传输控制块错误队列上。参数说明如下：

- sk，输出错误数据报的传输控制块。
- skb，从 ICMP 模块传递到传输层的 ICMP 差错报文。
- err，发生错误的错误码。
- port，对于 UDP 是出错报文的目的端口，其他情况下都为 0。
- info，出错信息的扩展信息。
- payload，指向产生 ICMP 差错的原始数据报中应用层的内容。

```
256 void ip_icmp_error(struct sock *sk, struct sk_buff *skb, int err,
257             __be16 port, u32 info, u8 *payload)
258 {
259     struct inet_sock *inet = inet_sk(sk);
260     struct sock_exterr_skb *serr;
261
262     if (!inet->recverr)
263         return;
264
265     skb = skb_clone(skb, GFP_ATOMIC);
266     if (!skb)
267         return;
268
269     serr = SKB_EXT_ERR(skb);
270     serr->ee.ee_errno = err;
271     serr->ee.ee_origin = SO_EE_ORIGIN_ICMP;
272     serr->ee.ee_type = skb->h.icmph->type;
273     serr->ee.ee_code = skb->h.icmph->code;
274     serr->ee.ee_pad = 0;
275     serr->ee.ee_info = info;
276     serr->ee.ee_data = 0;
277         serr->addr_offset = (u8*)&(((struct iphdr*)(skb->h.icmph+1))->daddr) -
    skb->nh.raw;
```

```
278     serr->port = port;
279
280     skb->h.raw = payload;
281     if (!skb_pull(skb, payload - skb->data) ||
282         sock_queue_err_skb(sk, skb))
283         kfree_skb(skb);
284 }
```

262-263 如果不允许接收扩展的错误信息，则返回。

265-280 由 ICMP 差错报文克隆出用于报告出错信息的报文，并填充各种错误信息。

281-283 将报告出错信息的数据报添加到传输控制块的错误队列，并唤醒等待该传输控制块的进程。

11.7.2 添加由本地产生的差错信息

当 UDP 套接口或 RAW 套接口发送数据时，如果待发送数据的长度超过 IP 数据报能负载的长度，会调用 ip_local_error()将数据报数据超长的出错信息添加到输出该出错报文的传输控制块错误队列上。ip_local_error()的实现和 ip_icmp_error()的实现类似，区别在于出错信息来源为本地。

```
286 void ip_local_error(struct sock *sk, int err, __be32 daddr, __be16 port, u32 info)
287 {
288     struct inet_sock *inet = inet_sk(sk);
289     struct sock_exterr_skb *serr;
290     struct iphdr *iph;
291     struct sk_buff *skb;
292
293     if (!inet->recverr)
294         return;
295
296     skb = alloc_skb(sizeof(struct iphdr), GFP_ATOMIC);
297     if (!skb)
298         return;
299
300     iph = (struct iphdr*)skb_put(skb, sizeof(struct iphdr));
301     skb->nh.iph = iph;
302     iph->daddr = daddr;
303
304     serr = SKB_EXT_ERR(skb);
305     serr->ee.ee_errno = err;
306     serr->ee.ee_origin = SO_EE_ORIGIN_LOCAL;
307     serr->ee.ee_type = 0;
308     serr->ee.ee_code = 0;
309     serr->ee.ee_pad = 0;
310     serr->ee.ee_info = info;
311     serr->ee.ee_data = 0;
312     serr->addr_offset = (u8*)&iph->daddr - skb->nh.raw;
313     serr->port = port;
314
315     skb->h.raw = skb->tail;
316     __skb_pull(skb, skb->tail - skb->data);
317
318     if (sock_queue_err_skb(sk, skb))
```

```
319         kfree_skb(skb);
320 }
```

11.7.3　读取错误信息

通常，recvmsg()是用来接收远端发送到所在套接口的数据的，但也可通过设置 flags 为 MSG_ERRQUEUE 来读取传输控制块错误队列上的错误信息。在 UDP 套接口和 RAW 套接口的 recvmsg()的实现中，先检测是否存在 MSG_ERRQUEUE 标识，如果有，则直接调用 ip_recv_error() 从传输控制块的错误队列中读取错误信息后返回，参见 33.16 节。参数说明如下：

- sk，从套接口错误队列上读取错误信息的传输控制块。
- msg，用于读取数据的消息头中有关对数据块的描述。
- len，用户空间提供缓存区的长度。

```
325 int ip_recv_error(struct sock *sk, struct msghdr *msg, int len)
326 {
327     struct sock_exterr_skb *serr;
328     struct sk_buff *skb, *skb2;
329     struct sockaddr_in *sin;
330     struct {
331         struct sock_extended_err ee;
332         struct sockaddr_in    offender;
333     } errhdr;
334     int err;
335     int copied;
336
337     err = -EAGAIN;
338     skb = skb_dequeue(&sk->sk_error_queue);
339     if (skb == NULL)
340         goto out;
```

从传输控制块的错误队列上获取队首的那个错误信息数据报。如果没有错误信息数据报，则返回错误码 EAGAIN，表示没有错误信息，可重试来获取错误信息。

```
342     copied = skb->len;
343     if (copied > len) {
344         msg->msg_flags |= MSG_TRUNC;
345         copied = len;
346     }
347     err = skb_copy_datagram_iovec(skb, 0, msg->msg_iov, copied);
348     if (err)
349         goto out_free_skb;
350
351     sock_recv_timestamp(msg, sk, skb);
```

设置每次获取错误数据报的长度。如果错误数据报的长度超过缓存区长度，则复制数据的长度上限为缓存区长度，丢弃未能复制的数据，在返回的数据中添加 MSG_TRUNC 标志，表示有数据未能完全读取。

通过 skb_copy_datagram_iovec()将数据报中的数据复制到缓存中，并记录时间戳。

```
353     serr = SKB_EXT_ERR(skb);
354
```

```
355     sin = (struct sockaddr_in *)msg->msg_name;
356     if (sin) {
357        sin->sin_family = AF_INET;
358        sin->sin_addr.s_addr = *(__be32*)(skb->nh.raw + serr->addr_offset);
359        sin->sin_port = serr->port;
360        memset(&sin->sin_zero, 0, sizeof(sin->sin_zero));
361     }
```

将产生差错报文的原始数据报的目的地址和目的端口复制到消息头中。

```
363     memcpy(&errhdr.ee, &serr->ee, sizeof(struct sock_extended_err));
364     sin = &errhdr.offender;
365     sin->sin_family = AF_UNSPEC;
366     if (serr->ee.ee_origin == SO_EE_ORIGIN_ICMP) {
367        struct inet_sock *inet = inet_sk(sk);
368
369        sin->sin_family = AF_INET;
370        sin->sin_addr.s_addr = skb->nh.iph->saddr;
371        sin->sin_port = 0;
372        memset(&sin->sin_zero, 0, sizeof(sin->sin_zero));
373        if (inet->cmsg_flags)
374           ip_cmsg_recv(msg, skb);
375     }
376
377     put_cmsg(msg, SOL_IP, IP_RECVERR, sizeof(errhdr), &errhdr);
```

获取差错报文错误信息块中的出错信息。

如果出错信息是来自 ICMP 的差错消息，则还需获取相关的源地址等信息。并且如果需要获取报文控制信息，则调用 ip_cmsg_recv()将报文控制信息复制到消息头中。

将出错扩展信息复制到消息头控制消息区域中。

```
379     /* Now we could try to dump offended packet options */
380
381     msg->msg_flags |= MSG_ERRQUEUE;
382     err = copied;
383
384     /* Reset and regenerate socket error */
385     spin_lock_bh(&sk->sk_error_queue.lock);
386     sk->sk_err = 0;
387     if ((skb2 = skb_peek(&sk->sk_error_queue)) != NULL) {
388        sk->sk_err = SKB_EXT_ERR(skb2)->ee.ee_errno;
389        spin_unlock_bh(&sk->sk_error_queue.lock);
390        sk->sk_error_report(sk);
391     } else
392        spin_unlock_bh(&sk->sk_error_queue.lock);
393
394 out_free_skb:
395     kfree_skb(skb);
396 out:
397     return err;
398 }
```

获取出错信息后，设置 MSG_ERRQUEUE 标志，表示接收到的是出错信息。然后检测

错误队列是否为空，如果不为空，则唤醒等待和异步等待该传输控制块的进程。最后返回复制的字节数。

11.8　报文控制信息

设置了 IP_PKTINFO 等报文控制信息相关的套接口选项之后，UDP 套接口或 RAW 套接口在通过 recvmsg 系统调用接收数据的同时，可获得该报文的报文控制信息。

UDP 套接口或 RAW 套接口在通过 sendmsg 系统调用输出数据时，可以将报文控制信息复制到输出数据的消息头中。这样，输出数据时会将消息头中报文控制信息读取到 IP 选项信息块中，构建 IP 数据报时会根据 IP 选项信息块构造 IP 首部中的选项。

11.8.1　IP 控制信息块

IP 控制信息块由 ipcm_cookie 结构描述，存储有关输出的控制信息，包括发送 UDP 数据报的源地址、输出网络设备接口索引以及 IP 首部中的选项，在整个输出过程中起传递信息的作用。

```
46 struct ipcm_cookie
47 {
48     __be32          addr;
49     int             oif;
50     struct ip_options    *opt;
51 };
```

```
48 __be32 addr
```

UDP 数据报或 RAW 数据报的目的地址。只有当存在 IP 选项时才设置，用作源路由选项的最后一跳地址。

```
49 int oif
```

UDP 数据报或 RAW 数据报的输出网络设备。

```
50 struct ip_options *opt
```

如果不为 NULL，则指向发送数据报的 IP 选项信息。

11.8.2　报文控制信息的输出

UDP 套接口和 RAW 套接口在通过 sendmsg 系统调用输出数据时，会检测消息头中是否存在控制信息，如果存在，则调用 ip_cmsg_send()将控制信息获取到一个 IP 控制信息块中。参数说明如下：

- msg，输出数据的消息头，包括控制信息等。
- ipc，用来保存控制信息的临时信息块。

```
166 int ip_cmsg_send(struct msghdr *msg, struct ipcm_cookie *ipc)
167 {
168     int err;
169     struct cmsghdr *cmsg;
170
171     for (cmsg = CMSG_FIRSTHDR(msg); cmsg; cmsg = CMSG_NXTHDR(msg, cmsg)) {
172         if (!CMSG_OK(msg, cmsg))
173             return -EINVAL;
```

```
174         if (cmsg->cmsg_level != SOL_IP)
175             continue;
176         switch (cmsg->cmsg_type) {
177         case IP_RETOPTS:
178             err = cmsg->cmsg_len - CMSG_ALIGN(sizeof(struct cmsghdr));
179             err = ip_options_get(&ipc->opt, CMSG_DATA(cmsg), err < 40 ? err : 40)
180             if (err)
181                 return err;
182             break;
183         case IP_PKTINFO:
184         {
185             struct in_pktinfo *info;
186             if (cmsg->cmsg_len != CMSG_LEN(sizeof(struct in_pktinfo)))
187                 return -EINVAL;
188             info = (struct in_pktinfo *)CMSG_DATA(cmsg);
189             ipc->oif = info->ipi_ifindex;
190             ipc->addr = info->ipi_spec_dst.s_addr;
191             break;
192         }
193         default:
194             return -EINVAL;
195         }
196     }
197     return 0;
198 }
```

171 遍历消息头中各种类型的控制信息。

172-173 检测消息头中标识控制信息长度，即检验消息头中标识控制信息的组成是否正确。

174-175 输出控制信息的级别必须为 SOL_IP。

177-182 处理 IP_RETOPTS 类型的控制信息，解析并生成 IP 选项信息。

183-191 处理 IP_PKTINFO 类型的控制信息，获取报文指定的输出网络设备和目的地址。

193-195 除 IP_RETOPTS 及 IP_PKTINFO 之外的控制信息类型直接返回错误码 EINVAL。

11.8.3 报文控制信息的输入

ip_cmsg_recv()用于获取报文控制信息，在 UDP 套接口和 RAW 套接口的 recvmsg 例程中，当设置了 IP_PKTINFO 等报文控制信息相关的套接口选项后，无论是接收错误信息还是正常的数据，都会被调用。在获取每一种报文控制信息前，会将报文控制信息标志向右移位，然后根据标志检测是否需要获取对应的报文控制信息。参数说明如下：

- msg，用来读取数据的消息头。
- skb，用来接收错误信息或正常数据的报文。

```
131 void ip_cmsg_recv(struct msghdr *msg, struct sk_buff *skb)
132 {
133     struct inet_sock *inet = inet_sk(skb->sk);
134     unsigned flags = inet->cmsg_flags;
135
136     /* Ordered by supposed usage frequency */
137     if (flags & 1)
138         ip_cmsg_recv_pktinfo(msg, skb);
139     if ((flags>>=1) == 0)
```

```
140         return;
141
142     if (flags & 1)
143         ip_cmsg_recv_ttl(msg, skb);
144     if ((flags>>=1) == 0)
145         return;
146
147     if (flags & 1)
148         ip_cmsg_recv_tos(msg, skb);
149     if ((flags>>=1) == 0)
150         return;
151
152     if (flags & 1)
153         ip_cmsg_recv_opts(msg, skb);
154     if ((flags>>=1) == 0)
155         return;
156
157     if (flags & 1)
158         ip_cmsg_recv_retopts(msg, skb);
159     if ((flags>>=1) == 0)
160         return;
161
162     if (flags & 1)
163         ip_cmsg_recv_security(msg, skb);
164 }
```

137-140 若设置了 IP PKTINFO 套接口选项，则获取 IP PKTINFO 控制信息。IP_PKTINFO 控制信息由 in_pktinfo 结构描述，内容包括数据报的目的地址、输入网络设备以及源地址。

142-145 若设置了 IP_TTL 套接口选项，则获取 IP_TTL 控制信息，内容为输入 IP 数据报首部中的 TTL。

147-150 若设置了 IP_TTL 套接口选项，则获取 IP_TOS 控制信息，内容为输入 IP 数据报首部中的 TOS。

152-155 若设置了 IP_RECVOPTS 套接口选项，则获取 IP_RECVOPTS 控制信息，内容为输入 IP 数据报首部中的 IP 选项。

157-160 若设置了 IP_RETOPTS 套接口选项，则获取 IP_RETOPTS 控制信息，内容为输入 IP 数据报首部中未处理过的 IP 选项。

162-163 若设置了 IP_PASSSEC 套接口选项，则获取 SCM_SECURITY 类型的控制信息，内容为有关安全的信息。

11.9　对端信息块

对端信息块由 inet_peer 结构描述，用来保存对端的一些信息，包括对端的地址以及传输层使用的时间戳等。主要用于在组装 IP 数据报时防止分片攻击（参见 13.3 节），在建立 TCP 连接时检测连接请求段是否有效以及其序号是否回绕（参见第 31 章）。

对端信息块以 v4addr 为关键字，peer_root 为根，组织成 AVL 树，见图 11-5。

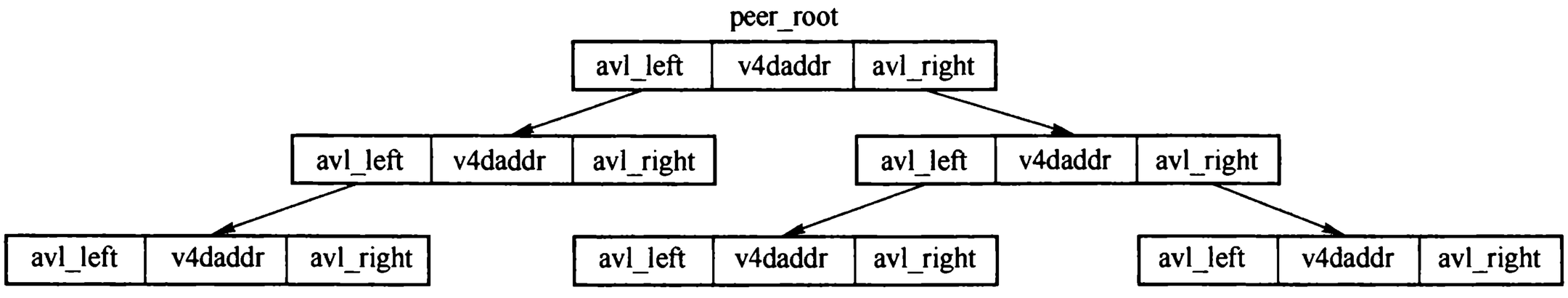

图 11-5　对端信息块组织方式

```
18 struct inet_peer
19 {
20     /* group together avl_left,avl_right,v4daddr to speedup lookups */
21     struct inet_peer    *avl_left, *avl_right;
22     __be32              v4daddr;    /* peer's address */
23     __u16               avl_height;
24     __u16               ip_id_count;    /* IP ID for the next packet */
25     struct inet_peer    *unused_next, **unused_prevp;
26     __u32               dtime;      /* the time of last use of not
27                           * referenced entries */
28     atomic_t            refcnt;
29     atomic_t            rid;        /* Frag reception counter */
30     __u32               tcp_ts;
31     unsigned long       tcp_ts_stamp;
32 };
```

21 struct inet_peer *avl_left, *avl_right

23 __u16 avl_height

用来将对端信息块组成 AVL 树。

25 struct inet_peer *unused_next, **unused_prevp

用来链接到 inet_peer_unused_head 链表上。该链表上的对端信息块都是当前闲置的，可回收的。

22 __be32 v4daddr

对端的 IP 地址。

24 __u16 ip_id_count

一个单调递增值，用来设置 IP 分片首部中的 id 域。根据对端地址初始化为随机值。

26 __u32 dtime

记录该对端信息块引用计数为 0 的时间。当闲置的时间超出指定的时间时，就会被回收。

28 atomic_t refcnt

引用计数器，标识当前被使用的次数。当引用计数为 0，表示该对端信息块没有被使用。

29 atomic_t rid

递增 ID，对端发送分片的计数器。参见 ipq 结构中的 rid 成员。

30 __u32 tcp_ts

TCP 中，记录最后一个 ACK 段到达的时间。参见 26.4.4 节的 tcp_options_received 结构中的 ts_recent 成员。

31 unsigned long tcp_ts_stamp

TCP 中，记录接收到的段中的时间戳，设置 ts_recent 的时间。参见 26.4.4 节的

tcp_options_received 结构中的 ts_recent_stamp 成员。

11.9.1　系统参数

系统参数如下：

（1）inet_peer_gc_maxtime

进行垃圾回收的最大时间间隔。在没有内存压力的情况下（如对端信息块的数量很少的时候），会使用较长的时间作为此次进行垃圾回收的时间间隔，但不会大于 inet_peer_gc_maxtime。默认值为 120。

（2）inet_peer_gc_mintime

进行垃圾回收的最小时间间隔。在有较大内存压力的情况下（如对端信息块的数量超过 inet_peer_threshold 时），则使用 inet_peer_minttl 作为此次进行垃圾回收的时间间隔。默认值为 10。

（3）inet_peer_maxttl

对端信息块的最长生存期。在没有内存压力的情况下（如对端信息块的数量很少时），会使用较长的时间作为此次进行垃圾回收的对端信息块生存期的阈值，但不会大于 inet_peer_maxttl。默认值为 600。

（4）inet_peer_minttl

对端信息块的最短生存期。在有较大内存压力的情况下（如对端信息块的数量超过 inet_peer_threshold 的时候），则使用 inet_peer_minttl 作为此次进行垃圾回收的对端信息块生存期的阈值。默认值为 120。

（5）inet_peer_threshold

用于计算垃圾回收最大时间间隔及对端信息块生存期阈值。

当前对端信息块数大于此阈值时，使用 inet_peer_gc_mintime 作为本次垃圾回收的时间间隔，否则根据 inet_peer_gc_maxtime 计算本次垃圾回收的时间间隔。

当前对端信息块数大于此阈值时，使用 inet_peer_minttl 作为本次垃圾回收的对端信息块生存期的阈值，否则根据 inet_peer_maxttl 计算本次垃圾回收对端信息块生存期的阈值。该参数默认值会根据系统内存大小动态调整，见表 11-11。算法见 inet_initpeers()。

表 11-11　inet_peer_threshold 的取值

内存容量	inet_peer_threshold 取值
大于 32MB	65664
16～32MB（包括 32MB）	32832
8～16MB（包括 16MB）	16416
小于或等于 8MB	8208

11.9.2　对端信息块的创建和查找

对端信息块的创建和查找都是通过 inet_getpeer()来实现的，由参数 create 来区分是创建还是查找。首先查找指定地址的对端信息块，如果查找命中，则返回查找的结果，否则，当 create 为 0 时返回 NULL，非 0 时创建新的对端信息块并添加到 AVL 树中，并返回该新创建的对端信息块。参数说明如下：

- daddr，待创建或查找对端信息块的 IP 地址。
- create，标识创建还是查找。

```
370 struct inet_peer *inet_getpeer(__be32 daddr, int create)
371 {
372     struct inet_peer *p, *n;
373     struct inet_peer **stack[PEER_MAXDEPTH], ***stackptr;
374
375     /* Look up for the address quickly. */
376     read_lock_bh(&peer_pool_lock);
377     p = lookup(daddr);
378     if (p != peer_avl_empty)
379         atomic_inc(&p->refcnt);
380     read_unlock_bh(&peer_pool_lock);
381
382     if (p != peer_avl_empty) {
383         /* The existing node has been found. */
384         /* Remove the entry from unused list if it was there. */
385         unlink_from_unused(p);
386         return p;
387     }
388
389     if (!create)
390         return NULL;
```

376-380 根据地址在 AVL 树中查找对应的对端信息块，如果查找命中，则增加对该对端信息块的引用计数。

382-387 如果查找到的对端信息块已经添加到了 inet_peer_unused_head 队列上，则先将其删除，以免被当作垃圾回收，然后返回查找到的对端信息块。

389-390 如果根据地址没有找到相应的对端信息块，且是查找操作，则返回 NULL。

```
392     /* Allocate the space outside the locked region. */
393     n = kmem_cache_alloc(peer_cachep, GFP_ATOMIC);
394     if (n == NULL)
395         return NULL;
396     n->v4daddr = daddr;
397     atomic_set(&n->refcnt, 1);
398     atomic_set(&n->rid, 0);
399     n->ip_id_count = secure_ip_id(daddr);
400     n->tcp_ts_stamp = 0;
401
402     write_lock_bh(&peer_pool_lock);
403     /* Check if an entry has suddenly appeared. */
404     p = lookup(daddr);
405     if (p != peer_avl_empty)
406         goto out_free;
407
408     /* Link the node. */
409     link_to_pool(n);
410     n->unused_prevp = NULL; /* not on the list */
411     peer_total++;
412     write_unlock_bh(&peer_pool_lock);
413
414     if (peer_total >= inet_peer_threshold)
```

```
415         /* Remove one less-recently-used entry. */
416         cleanup_once(0);
417
418     return n;
```

392-400 如果根据地址没找到相应的对端信息块，且允许创建，则从高速缓存中分配对端信息块，并设置相应的值。

402-412 先检查是否有同样地址的对端信息块已添加到 AVL 树，因为在分配对端信息块时，其他模块有可能已创建了相同地址的对端信息块。如果有，则不使用刚创建的对端信息块并将其释放。如果没有，则将新创建的对端信息块添加到 AVL 树并更新当前的对端信息块数。

414-416 如果当前对端信息块数量超过了 inet_peer_threshold，则调用 cleanup_once()释放 inet_peer_unused_head 队首的对端信息块。

418 返回创建的对端信息块。

```
420 out_free:
421     /* The appropriate node is already in the pool. */
422     atomic_inc(&p->refcnt);
423     write_unlock_bh(&peer_pool_lock);
424     /* Remove the entry from unused list if it was there. */
425     unlink_from_unused(p);
426     /* Free preallocated the preallocated node. */
427     kmem_cache_free(peer_cachep, n);
428     return p;
429 }
```

在创建了对端信息块之后添加到 AVL 树时，有其他模块创建了相同地址的对端信息块，此时需释放刚分配的对端信息块，而使用其他模块已创建的对端信息块。

11.9.3　对端信息块的删除

当使用完对端信息块之后，需要将其删除并释放。实际上，inet_putpeer()只是将该对端信息块添加到 inet_peer_unused_head 队列上，表示该对端信息块当前没有被使用。而真正的删除和释放，由垃圾回收机制来处理。

```
461 void inet_putpeer(struct inet_peer *p)
462 {
463     spin_lock_bh(&inet_peer_unused_lock);
464     if (atomic_dec_and_test(&p->refcnt)) {
465         p->unused_prevp = inet_peer_unused_tailp;
466         p->unused_next = NULL;
467         *inet_peer_unused_tailp = p;
468         inet_peer_unused_tailp = &p->unused_next;
469         p->dtime = (__u32)jiffies;
470     }
471     spin_unlock_bh(&inet_peer_unused_lock);
472 }
```

当待删除的对端信息块的引用计数为 0 时，表示没有被使用，此时将它添加到 inet_peer_unused_head 队列上，等待垃圾回收或再次被使用。

11.9.4 垃圾回收

对端信息块一旦未被使用，即进入闲置状态。处于闲置状态的对端信息块消耗同样的内存，因此当闲置状态的对端信息块过期后，需将其回收。

处理垃圾回收的方式有两种——同步和异步。同步方式通常是在创建对端信息块时发现当前对端信息块数达到 net_peer_threshold 时触发，而异步方式则是在定时器时间到时触发。

1．对端信息块的释放

正在使用的对端信息块是不会过期的，只有闲置的对端信息块才有可能过期，因为对端信息块一旦闲置，就会被添加到 inet_peer_unused_head 队列中，并记录其闲置的时间。在同步或异步清理时，一旦发现闲置时间达到阈值，对端信息块就会过期。对端信息块的过期与 inet_peer_maxttl、inet_peer_minttl 和 inet_peer_threshold 有关。

cleanup_once()用来检测 inet_peer_unused_head 队列中第一个闲置的对端信息块，一旦检测到该对端信息块闲置时间达到阈值，即将其释放。

参数 ttl 是用来检测对端信息块闲置时间的阈值。

```
333 static int cleanup_once(unsigned long ttl)
334 {
335     struct inet_peer *p;
336
337     /* Remove the first entry from the list of unused nodes. */
338     spin_lock_bh(&inet_peer_unused_lock);
339     p = inet_peer_unused_head;
340     if (p != NULL) {
341         __u32 delta = (__u32)jiffies - p->dtime;
342         if (delta < ttl) {
343             /* Do not prune fresh entries. */
344             spin_unlock_bh(&inet_peer_unused_lock);
345             return -1;
346         }
347         inet_peer_unused_head = p->unused_next;
348         if (p->unused_next != NULL)
349             p->unused_next->unused_prevp = p->unused_prevp;
350         else
351             inet_peer_unused_tailp = p->unused_prevp;
352         p->unused_prevp = NULL; /* mark as not on the list */
353         /* Grab an extra reference to prevent node disappearing
354          * before unlink_from_pool() call. */
355         atomic_inc(&p->refcnt);
356     }
357     spin_unlock_bh(&inet_peer_unused_lock);
358
359     if (p == NULL)
360         /* It means that the total number of USED entries has
361          * grown over inet_peer_threshold.  It shouldn't really
362          * happen because of entry limits in route cache. */
363         return -1;
364
365     unlink_from_pool(p);
366     return 0;
367 }
```

339-357 如果 inet_peer_unused_head 队列不为空，则获取其上第一个闲置的对端信息块。

检测该对端信息块的闲置时间是否达到阈值。如果达到，则将其从队列中摘除，否则返回非 0，表示没有对端信息块可释放。

359-363 如果 inet_peer_unused_head 队列为空，则返回非 0，表示没有端信息块可释放。

365 调用 unlink_from_pool()将对端信息块从 AVL 树中删除并释放。

366 返回 0，表示 inet_peer_unused_head 队列中可能还有对端信息块可释放。

2. 同步清理

在创建对端信息块时，如果检测到当前对端信息块数达到阈值，就会调用 cleanup_once()释放 inet_peer_unused_head 队首的对端信息块。peer_total 为当前对端信息块数。由于内存压力较大，因此此处调用 cleanup_once()参数 ttl 值为 0，即必须释放一个闲置的对端信息块。

```
370 struct inet_peer *inet_getpeer(__be32 daddr, int create)
371 {
...
414     if (peer_total >= inet_peer_threshold)
415         /* Remove one less-recently-used entry. */
416         cleanup_once(0);
...
429 }
```

3. 异步清理

同步垃圾回收用于处理内存压力较大的特殊情况，事实上这种情况比较少见，同时也是非常影响性能的。因此为了避免这种特殊情况的出现或降低其出现的概率，使用 peer_periodic_timer 定时器来进行周期性地垃圾回收，该定时器的处理例程为 peer_check_expire()。

peer_check_expire 定时器的初始间隔时间在 inet_initpeers()中设置，而在运行中，则会根据当前对端信息块的数量是否达到 inet_peer_threshold，进行动态计算。因此，间隔时间与 inet_peer_maxttl、inet_peer_minttl 和 inet_peer_threshold 有着密切的关系。

```
432 static void peer_check_expire(unsigned long dummy)
433 {
434     unsigned long now = jiffies;
435     int ttl;
436
437     if (peer_total >= inet_peer_threshold)
438         ttl = inet_peer_minttl;
439     else
440         ttl = inet_peer_maxttl
441                 - (inet_peer_maxttl - inet_peer_minttl) / HZ *
442                     peer_total / inet_peer_threshold * HZ;
443     while (!cleanup_once(ttl)) {
444         if (jiffies != now)
445             break;
446     }
447
448     /* Trigger the timer after inet_peer_gc_mintime .. inet_peer_gc_maxtime
```

```
449     * interval depending on the total number of entries (more entries,
450     * less interval). */
451    if (peer_total >= inet_peer_threshold)
452       peer_periodic_timer.expires = jiffies + inet_peer_gc_mintime;
453    else
454       peer_periodic_timer.expires = jiffies
455          + inet_peer_gc_maxtime
456          - (inet_peer_gc_maxtime - inet_peer_gc_mintime) / HZ *
457             peer_total / inet_peer_threshold * HZ;
458    add_timer(&peer_periodic_timer);
459 }
```

437-440 根据当前对端信息块数计算本次垃圾回收的对端信息块生存期阈值。当前对端信息块数大于 inet_peer_threshold 时，使用 inet_peer_minttl 作为本次垃圾回收的对端信息块生存期阈值，否则根据 inet_peer_maxttl 来计算本次垃圾回收的对端信息块生存期阈值。

443-446 循环检测并删除闲置时间达到阈值的对端信息块。

451-458 根据当前对端信息块数计算 peer_check_expire 定时器下次激活时间，并重新设置该定时器。当前对端信息块数大于 inet_peer_threshold 时，使用 inet_peer_gc_mintime 作为本次垃圾回收的时间间隔，否则根据 inet_peer_gc_maxtime 来计算本次垃圾回收的时间间隔。

11.10 IP 数据报的输入处理

IPv4 的 IP 数据报接收例程定义如下所示：

```
1247 static struct packet_type ip_packet_type = {
1248    .type = __constant_htons(ETH_P_IP),
1249    .func = ip_rcv,
1250    .gso_send_check = inet_gso_send_check,
1251    .gso_segment = inet_gso_segment,
1252 };
```

1248 定义 IPv4 的 IP 数据报协议号为 ETH_P_IP。

1249 IPv4 的 IP 数据报接收处理函数为 ip_rcv()。

1250-1251 IP 层支持 GSO 分段及校验和的计算的回调函数分别为 inet_gso_segment()和 inet_gso_send_check()。

（1）ip_rcv()

当网络设备接收到报文时，会根据网络层的协议号从 ptype_base 散列表中找到对应的接收函数，由该函数来处理接收。

IPv4 的数据报类型 ip_packet_type 是在网络初始化时通过 dev_add_pack()注册到系统的 ptype_base 散列表中的，对应的接收函数为 ip_rcv()。ip_rcv()处理完成并经 PRE-ROUTING 点 netfilter 处理后，再由 ip_rcv_finish()处理。在 ip_rcv_finish()中，根据数据报的路由信息，决定这个数据报是转发还是输入到本机。由此产生两条路径，输入到本机由 ip_local_deliver()处理，而转发由 ip_forward()处理。参数说明如下：

- skb，接收到的 IP 数据报。

- dev，接收到的 IP 数据报当前的输入网络设备。
- pt，输入此数据报的网络层输入接口，未使用。
- orig_dev，接收到的 IP 数据报原始的输入网络设备。

```
373 int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt,
    struct net_device *orig_dev)
374 {
375     struct iphdr *iph;
376     u32 len;
377
378 /* When the interface is in promisc. mode, drop all the crap
379  * that it receives, do not try to analyse it.
380  */
381     if (skb->pkt_type == PACKET_OTHERHOST)
382         goto drop;
383
384     IP_INC_STATS_BH(IPSTATS_MIB_INRECEIVES);
385
386     if ((skb = skb_share_check(skb, GFP_ATOMIC)) == NULL) {
387         IP_INC_STATS_BH(IPSTATS_MIB_INDISCARDS);
388         goto out;
389     }
390
391     if (!pskb_may_pull(skb, sizeof(struct iphdr)))
392         goto inhdr_error;
```

381-382 丢弃不去往本地的数据报，此处只接收发往本机的数据报。

386-389 检测接收到的数据报是否为一个共享数据包。如果是，则必须复制一个副本，再作进一步处理，因为在处理过程中可能会修改数据报中的信息。

391-392 通过判断数据报长度来检测数据报是否有效，不能小于 IP 首部长度。

```
394     iph = skb->nh.iph;
395
396 /*
397  *    RFC1122: 3.1.2.2 MUST silently discard any IP frame that fails the checksum.
398  *
399  *    Is the datagram acceptable?
400  *
401  *    1.    Length at least the size of an ip header
402  *    2.    Version of 4
403  *    3.    Checksums correctly. [Speed optimisation for later, skip loopback
    checksums]
404  *    4.    Doesn't have a bogus length
405  */
406
407     if (iph->ihl < 5 || iph->version != 4)
408         goto inhdr_error;
409
410     if (!pskb_may_pull(skb, iph->ihl*4))
411         goto inhdr_error;
412
413     iph = skb->nh.iph;
414
415     if (unlikely(ip_fast_csum((u8 *)iph, iph->ihl)))
```

```
416         goto inhdr_error;
417
418     len = ntohs(iph->tot_len);
419     if (skb->len < len || len < (iph->ihl*4))
420         goto inhdr_error;
411
422 /* Our transport medium may have padded the buffer out. Now we know it
423  * is IP we can trim to the true length of the frame.
424  * Note this now means skb->len holds ntohs(iph->tot_len).
425  */
426     if (pskb_trim_rcsum(skb, len)) {
427         IP_INC_STATS_BH(IPSTATS_MIB_INDISCARDS);
428         goto drop;
429     }
```

407-408 由于 IP 数据报首部长度至少为 20B（即 5 个 32 位），因此 IP 数据报首部中首部长度域若小于 5，则说明长度异常。或者 IP 版本域不为 4，则版本异常。遇到以上两种情况，作相应的异常处理。

410-411 根据 IP 数据报首部中的首部长度值重新检测 IP 数据报是否有效。

413-415 根据 IP 数据报首部中的校验和检测 IP 首部是否有效。

418-420 根据 IP 数据报首部中的数据包总长度值检测数据报是否有效。

426-429 根据 IP 数据报首部中的数据包总长度值重新设置 skb 的长度。

```
431     /* Remove any debris in the socket control block */
432     memset(IPCB(skb), 0, sizeof(struct inet_skb_parm));
433
434     return NF_HOOK(PF_INET, NF_IP_PRE_ROUTING, skb, dev, NULL,
435                    ip_rcv_finish);
```

432 将 skb 中的 IP 控制块清零，以便后续对 IP 选项的处理。

434 最后通过 netfilter 模块处理后，调用 ip_rcv_finish()完成 IP 数据报的输入。

```
437 inhdr_error:
438     IP_INC_STATS_BH(IPSTATS_MIB_INHDRERRORS);
439 drop:
440         kfree_skb(skb);
441 out:
442         return NET_RX_DROP;
443 }
```

在此处理无效数据报。

（2）ip_rcv_finish()

ip_rcv_finish()在 ip_rcv()中当 IP 数据报经过 netfilter 模块处理后被调用。完成的主要功能是，如果还没有为该数据报查找输入路由缓存，则调用 ip_route_input()为其查找输入路由缓存。接着处理 IP 数据报首部中的选项，最后根据输入路由缓存输入到本地或转发。

```
331 static inline int ip_rcv_finish(struct sk_buff *skb)
332 {
333     struct iphdr *iph = skb->nh.iph;
334
335 /*
```

```
336 *    Initialise the virtual path cache for the packet. It describes
337 *    how the packet travels inside Linux networking.
338 */
339     if (skb->dst == NULL) {
340         int err = ip_route_input(skb, iph->daddr, iph->saddr, iph->tos,
341                                  skb->dev);
342         if (unlikely(err)) {
343             if (err == -EHOSTUNREACH)
344                 IP_INC_STATS_BH(IPSTATS_MIB_INADDRERRORS);
345             goto drop;
346         }
347     }
```

如果还没有为该数据报查找输入路由缓存，则调用 ip_route_input()为其查找输入路由缓存。若查找失败，则将该数据报丢弃。

```
349 #ifdef CONFIG_NET_CLS_ROUTE
350     if (unlikely(skb->dst->tclassid)) {
351         struct ip_rt_acct *st = ip_rt_acct + 256*smp_processor_id();
352         u32 idx = skb->dst->tclassid;
353         st[idx&0xFF].o_packets++;
354         st[idx&0xFF].o_bytes+=skb->len;
355         st[(idx>>16)&0xFF].i_packets++;
356         st[(idx>>16)&0xFF].i_bytes+=skb->len;
357     }
358 #endif
```

与路由表的 classifier 标签相关，本书不作论述。

```
360     if (iph->ihl > 5 && ip_rcv_options(skb))
361         goto drop;
362
363     return dst_input(skb);
364
365 drop:
366         kfree_skb(skb);
367         return NET_RX_DROP;
368 }
```

360-361 根据长度判断 IP 首部中是否存在选项，如果有，则调用 ip_rcv_options()处理 IP 选项。

363 最后根据输入路由缓存决定输入到本地或转发，最终前者调用 ip_local_deliver()，后者调用 ip_forward()。

11.10.1　IP 数据报输入到本地

（1）ip_local_deliver()

输入到本地的 IP 数据报由 ip_local_deliver()处理，该函数由 ip_rcv_finish()根据输入路由缓存调用。先判断接收到的数据报是不是分片，若是分片，则需将分片重组。如果不是分片或分片重组得到完整 IP 数据报，则本地输入点通过 netfilter 处理之后调用 ip_local_deliver_finish()完成数据报的本地输入。

```
263 int ip_local_deliver(struct sk_buff *skb)
264 {
265 /*
266  *   Reassemble IP fragments.
267  */
268
269     if (skb->nh.iph->frag_off & htons(IP_MF|IP_OFFSET)) {
270         skb = ip_defrag(skb, IP_DEFRAG_LOCAL_DELIVER);
271         if (!skb)
272             return 0;
273     }
274
275     return NF_HOOK(PF_INET, NF_IP_LOCAL_IN, skb, skb->dev, NULL,
276                    ip_local_deliver_finish);
277 }
```

269-270 如果接收到的 IP 数据报是分片，则调用 ip_defrag()进行重组，其标志为 IP_DEFRAG_LOCAL_DELIVER。

271-273 如果 ip_defrag()返回为 0，则表示 IP 数据报分片尚未到齐，重组没有完成，直接返回。非 0，则返回值是已完成重组的 IP 数据报 skb 缓存的指针，需传送到传输层进行处理。

275-276 经过 netfilter 处理后，调用 ip_local_deliver_finish()，将组装完成的 IP 数据报传送到传输层作处理。

（2）ip_local_deliver_finish()

此函数将输入数据报从网络层传递到传输层。过程如下：

1）首先，在数据报传递给传输层之前，去掉 IP 首部。

2）接着，如果是 RAW 套接口接收数据报，则需复制一份副本，输入到接收该数据报的套接口。

3）最后，通过传输层的接收例程，将数据报传递到传输层，由传输层进行处理。

```
199 static inline int ip_local_deliver_finish(struct sk_buff *skb)
200 {
201     int ihl = skb->nh.iph->ihl*4;
202
203     __skb_pull(skb, ihl);
204
205         /* Point into the IP datagram, just past the header. */
206         skb->h.raw = skb->data;
207
208     rcu_read_lock();
209     {
210         /* Note: See raw.c and net/raw.h, RAWV4_HTABLE_SIZE==MAX_INET_PROTOS */
211         int protocol = skb->nh.iph->protocol;
212         int hash;
213         struct sock *raw_sk;
214         struct net_protocol *ipprot;
215
216     resubmit:
217         hash = protocol & (MAX_INET_PROTOS - 1);
218         raw_sk = sk_head(&raw_v4_htable[hash]);
219             .
```

```
220         /* If there maybe a raw socket we must check - if not we
221          * don't care less
222          */
223         if (raw_sk && !raw_v4_input(skb, skb->nh.iph, hash))
224             raw_sk = NULL;
225
226         if ((ipprot = rcu_dereference(inet_protos[hash])) != NULL) {
227             int ret;
228
229             if (!ipprot->no_policy) {
230                 if (!xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb)) {
231                     kfree_skb(skb);
232                     goto out;
233                 }
234                 nf_reset(skb);
235             }
236             ret = ipprot->handler(skb);
237             if (ret < 0) {
238                 protocol = -ret;
239                 goto resubmit;
240             }
241             IP_INC_STATS_BH(IPSTATS_MIB_INDELIVERS);
242         } else {
243             if (!raw_sk) {
244                 if (xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb)) {
245                     IP_INC_STATS_BH(IPSTATS_MIB_INUNKNOWNPROTOS);
246                     icmp_send(skb, ICMP_DEST_UNREACH,
247                               ICMP_PROT_UNREACH, 0);
248                 }
249             } else
250                 IP_INC_STATS_BH(IPSTATS_MIB_INDELIVERS);
251             kfree_skb(skb);
252         }
253     }
254 out:
255     rcu_read_unlock();
256
257     return 0;
258 }
```

203 在数据报传递给传输层之前，先去掉 IP 首部。

211 从 IP 首部中获取传输层协议号，用于计算哈希值。

217-224 处理 RAW 套接口，先根据传输层协议号得到哈希值，然后查看 raw_v4_htable 散列表中以该值为关键字的哈希桶是否为空。如果不为空，则说明创建了 RAW 套接口，复制该数据报的副本输入到注册到该桶中的所有套接口。

226-241 通过查找 inet_protos 数组，确定是否注册了与 IP 首部中传输层协议号一致的传输层协议。若查找命中，则执行对应的传输层接收例程。

242-252 如果没有相应的协议传输层接收该数据报，则释放该数据报。在释放前，如果是 RAW 套接口没有接收或接收异常，则还需产生一个目的不可达 ICMP 报文给发送方。

11.10.2　IP 数据报的转发

IP 数据报的转发由 ip_forward()处理，该函数在 ip_rcv_finish()中通过输入路由缓存被调用，

流程图见图 11-6。

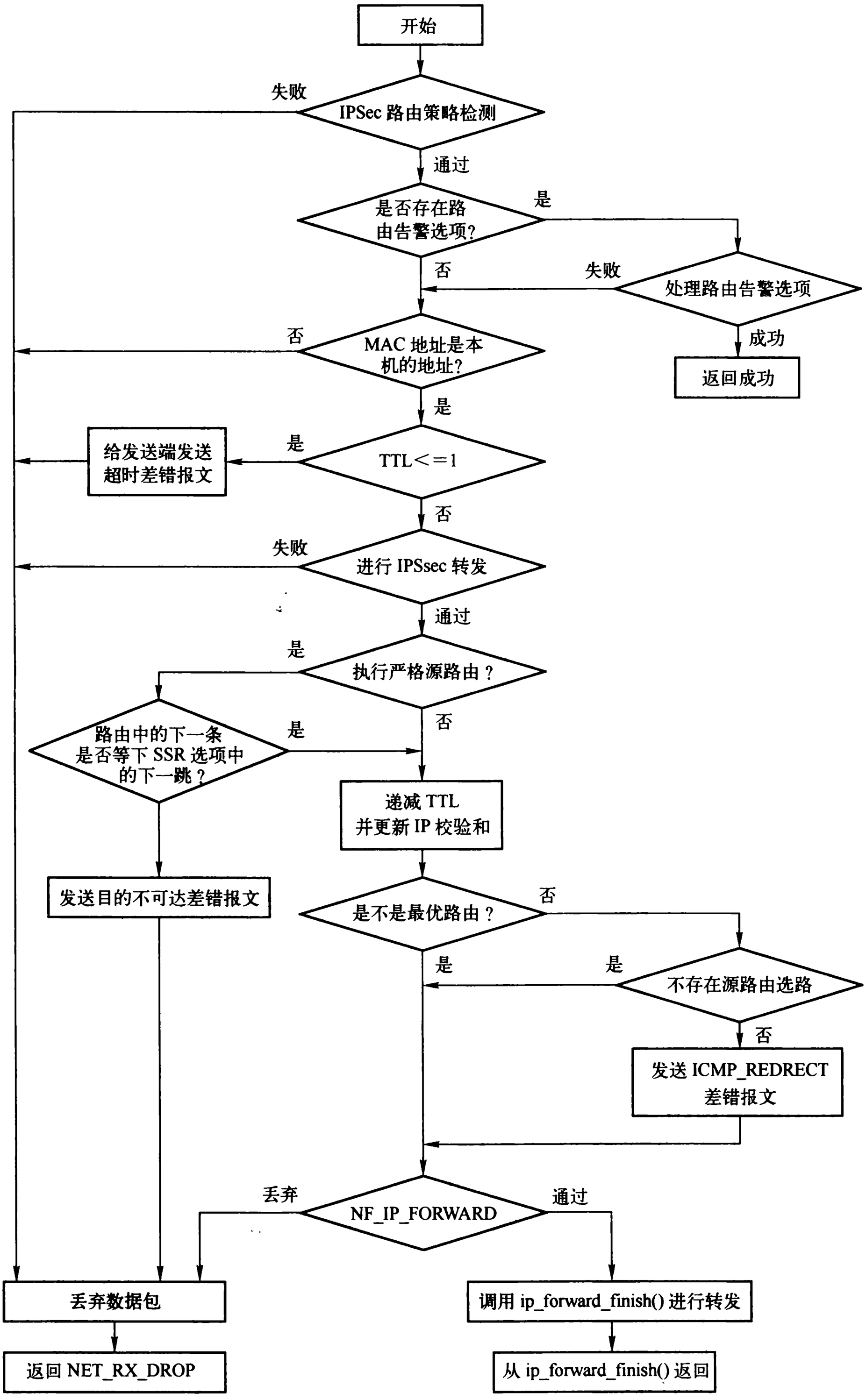

图 11-6　ip_forward()流程图

```
56  int ip_forward(struct sk_buff *skb)
57  {
58      struct iphdr *iph;    /* Our header */
59      struct rtable *rt;    /* Route we use */
60      struct ip_options * opt    = &(IPCB(skb)->opt);
61
62      if (!xfrm4_policy_check(NULL, XFRM_POLICY_FWD, skb))
63          goto drop;
64
65      if (IPCB(skb)->opt.router_alert && ip_call_ra_chain(skb))
66          return NET_RX_SUCCESS;
67
68      if (skb->pkt_type != PACKET_HOST)
69          goto drop;
70
71      skb->ip_summed = CHECKSUM_NONE;
72
73      /*
74       *    According to the RFC, we must first decrease the TTL field. If
75       *    that reaches zero, we must reply an ICMP control message telling
76       *    that the packet's lifetime expired.
77       */
78      if (skb->nh.iph->ttl <= 1)
79              goto too_many_hops;
80
81      if (!xfrm4_route_forward(skb))
82          goto drop;
83
84      rt = (struct rtable*)skb->dst;
85
86      if (opt->is_strictroute && rt->rt_dst != rt->rt_gateway)
87          goto sr_failed;
88
89      /* We are about to mangle packet. Copy it! */
90      if (skb_cow(skb, LL_RESERVED_SPACE(rt->u.dst.dev)+rt->u.dst.header_len))
91          goto drop;
```

62-63 调用 xfrm4_policy_check()查找 IPsec 策略数据库。如果查找失败，则丢弃该数据报。

65-66 如果数据报中存在路由警告选项，则调用 ip_call_ra_chain()将数据报输入给对路由警告选项感兴趣的用户进程。如果成功，则不再转发数据报。

68-69 承载该 IP 数据报的以太网帧目的地址与收到它的网络设备的 MAC 地址相等才能转发，也就是说发给接收该数据包的主机的数据包才接收。

71 由于在转发过程中可能会修改 IP 首部，因此将 ip_summed 设置为 CHECKSUM_NONE，在后续的输出时还得由软件来执行校验和。

78-79 待转发数据的报生存时间为 0 时，丢弃该数据报，并发送超时 ICMP 报文到发送方。

81-82 进行 IPsec 路由选路和转发处理，如果失败，则丢弃该数据报。

84-87 如果数据报启用严格源路由选项，且数据报的下一跳不是网关，则发送超时 ICMP 报文到发送方，并丢弃该数据报。

90-91 确保 SKB 有指定长度的 headroom 空间。当 SKB 的 headroom 空间小于指定长度或者克隆 SKB 时，会新建 SKB 缓冲并释放对原包的引用。

```
92      iph = skb->nh.iph;
```

```
93
94      /* Decrease ttl after skb cow done */
95      ip_decrease_ttl(iph);
96
97      /*
98       *    We now generate an ICMP HOST REDIRECT giving the route
99       *    we calculated.
100      */
101     if (rt->rt_flags&RTCF_DOREDIRECT && !opt->srr)
102        ip_rt_send_redirect(skb);
103
104     skb->priority = rt_tos2priority(iph->tos);
105
106        return NF_HOOK(PF_INET, NF_IP_FORWARD, skb, skb->dev, rt->u.dst.dev,
107                       ip_forward_finish);
```

92-95 经过路由器，IP 数据报生存时间递减 1。

101-102 如果该数据报的输出路由存在重定向标志，且该数据报中不存在源路由选项，则向发送方发送重定向 ICMP 报文。

106-107 经 netfilter 处理后，调用 ip_forward_finish()，完成 IP 层的转发操作。

```
109 sr_failed:
115
116 too_many_hops:
117         /* Tell the sender its packet died... */
118         IP_INC_STATS_BH(IPSTATS_MIB_INHDRERRORS);
119         icmp_send(skb, ICMP_TIME_EXCEEDED, ICMP_EXC_TTL, 0);
120 drop:
121     kfree_skb(skb);
122     return NET_RX_DROP;
123 }
```

109-123 当处理过程中发生异常时跳转到此处进行相关的异常处理。

116-119 发送超时 ICMP 报文给发送方。

120-122 丢弃数据报。

ip_forward_finish()完成输入 IP 数据报的转发。

```
44  static inline int ip_forward_finish(struct sk_buff *skb)
45  {
46      struct ip_options * opt    = &(IPCB(skb)->opt);
47
48      IP_INC_STATS_BH(IPSTATS_MIB_OUTFORWDATAGRAMS);
49
50      if (unlikely(opt->optlen))
51          ip_forward_options(skb);
52
53      return dst_output(skb);
54  }
```

50-51 处理转发 IP 数据报中的 IP 选项，包括记录路由选项和时间戳选项。

53 通过路由缓存将数据报输出，最终会调用单播的输出函数 ip_output()或组播的输出函数 ip_mc_output()。

11.11　IP 数据报的输出处理

11.11.1　IP 数据报输出到设备

无论是转发的数据报还是从本地输出的数据报，经过路由之后都要输出到网络设备，而输出到网络设备的接口就是 ip_output()。但该函数并不直接被调用，而是通过输出数据报目的路由缓存项中的输出接口调用。

从本地输出的数据报查找到目的路由缓存项，并通过 NF_IP_LOCAL_OUT 点的 netfilter 处理之后，便会调用 dst_output()输出数据报到网络设备，当然输出到网络设备并不是直接的，而是通过邻居子系统进行的。转发的数据报与从本地输出的数据报类似，通过 NF_IP_FORWARD 点的 netfilter 处理之后，调用 dst_output()输出数据报到网络设备。

相关函数如下：

（1）dst_output0()

封装了输出数据报目的路由缓存项中的输出接口。

```
226 static inline int dst_output(struct sk_buff *skb)
227 {
228     return skb->dst->output(skb);
229 }
```

（2）ip_output()

对于单播数据报，目的路由缓存项中的输出接口 output 是 ip_output()。

```
270  int ip_output(struct sk_buff *skb)
271  {
272      struct net_device *dev = skb->dst->dev;
273
274      IP_INC_STATS(IPSTATS_MIB_OUTREQUESTS);
275
276      skb->dev = dev;
277      skb->protocol = htons(ETH_P_IP);
278
279      return NF_HOOK_COND(PF_INET, NF_IP_POST_ROUTING, skb, NULL, dev,
280                          ip_finish_output,
281                          !(IPCB(skb)->flags & IPSKB_REROUTED));
282  }
```

276-277 设置数据报的输出网络设备和数据报网络层协议类型。

279-281 经 netfilter 处理后，调用 ip_finish_output()继续 IP 数据报的输出。

（3）ip_finish_output()

此函数的主要功能是：如果数据报大于 MTU，则调用 ip_fragment()分片，否则调用 ip_finish_output2()输出。

```
196  static inline int ip_finish_output(struct sk_buff *skb)
197  {
198  #if defined(CONFIG_NETFILTER) && defined(CONFIG_XFRM)
199      /* Policy lookup after SNAT yielded a new policy */
200      if (skb->dst->xfrm != NULL) {
```

```
201         IPCB(skb)->flags |= IPSKB_REROUTED;
202         return dst_output(skb);
203     }
204 #endif
205     if (skb->len > dst_mtu(skb->dst) && !skb_is_gso(skb))
206         return ip_fragment(skb, ip_finish_output2);
207     else
208         return ip_finish_output2(skb);
209 }
```

198-204 netfilter 和 IPSec 相关处理，本书不作论述。

205-206 如果数据包长度大于 MTU，则调用 ip_fragment()对 IP 数据报进行分片。

207-208 否则，调用 ip_finish_output2()作进一步处理。

（4）ip_finish_output2()

此函数通过邻居子系统将数据报输出到网络设备。

```
164 static inline int ip_finish_output2(struct sk_buff *skb)
165 {
166     struct dst_entry *dst = skb->dst;
167     struct net_device *dev = dst->dev;
168     int hh_len = LL_RESERVED_SPACE(dev);
169
170     /* Be paranoid, rather than too clever. */
171     if (unlikely(skb_headroom(skb) < hh_len && dev->hard_header)) {
172         struct sk_buff *skb2;
173
174         skb2 = skb_realloc_headroom(skb, LL_RESERVED_SPACE(dev));
175         if (skb2 == NULL) {
176             kfree_skb(skb);
177             return -ENOMEM;
178         }
179         if (skb->sk)
180             skb_set_owner_w(skb2, skb->sk);
181         kfree_skb(skb);
182         skb = skb2;
183     }
184
185     if (dst->hh)
186         return neigh_hh_output(dst->hh, skb);
187     else if (dst->neighbour)
188         return dst->neighbour->output(skb);
189
190     if (net_ratelimit())
191         printk(KERN_DEBUG "ip_finish_output2: No header cache and no neighbour!\n");
192     kfree_skb(skb);
193     return -EINVAL;
194 }
```

168-183 检测 skb 的前部空间是否还能存储链路层首部。如果不够，则重新分配更大存储区的 SKB，并释放原 skb。

185-188 如果缓存了链路层的首部，则调用 neigh_hh_output()输出数据报。否则，若存在对应的邻居项，则通过邻居项的输出方法输出数据报。

190-193 如果既没有缓存链路层的首部，又不存在对应的邻居项，在这种情况下，不能输出，释放该数据报。

11.11.2　TCP 输出的接口

在 TCP 中，将 TCP 段打包成 IP 数据报的方法根据 TCP 段类型的不同而有多种接口。其中最常用的就是 ip_queue_xmit()，而 ip_build_and_send_pkt()和 ip_send_reply()只有在发送特定段时才会被调用。

1．ip_queue_xmit()

ip_queue_xmit()是 TCP 传输中被调用得最多的函数，普通的数据输出最后都是由它进行打包处理的，流程图见 11-7。

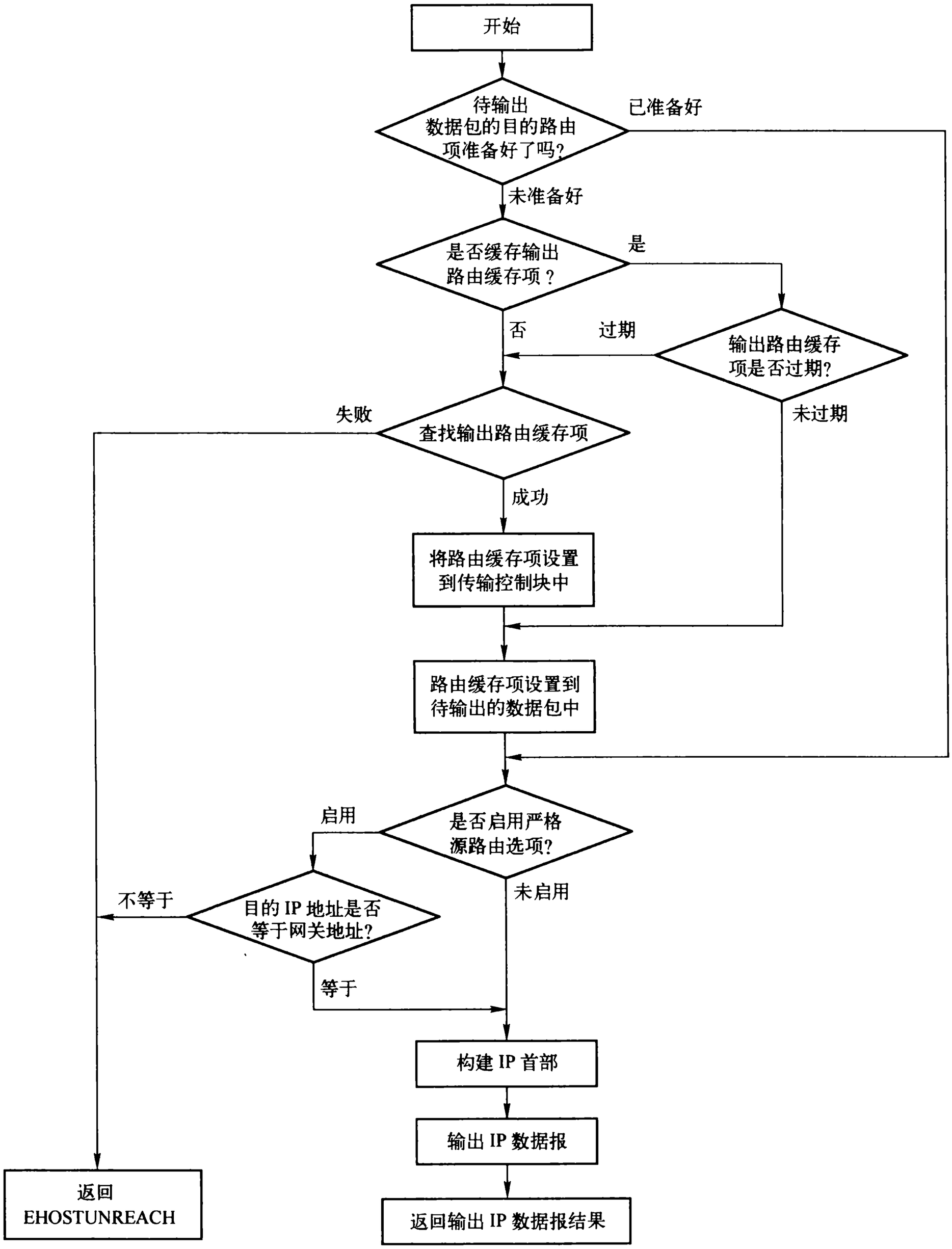

图 11-7　ip_queue_xmit()流程图

参数说明如下：

- skb，待封装成 IP 数据报的 TCP 段。
- ipfragok，标识待输出的数据是否已经完成分片。由于在调用该函数时 ipfragok 参数总为 0，因此输出的 IP 数据报是否已分片取决于是否启用 PMTU 发现。

```
284  int ip_queue_xmit(struct sk_buff *skb, int ipfragok)
285  {
286      struct sock *sk = skb->sk;
287      struct inet_sock *inet = inet_sk(sk);
288      struct ip_options *opt = inet->opt;
289      struct rtable *rt;
290      struct iphdr *iph;
291
292      /* Skip all of this if the packet is already routed,
293       * f.e. by something like SCTP.
294       */
295      rt = (struct rtable *) skb->dst;
296      if (rt != NULL)
297          goto packet_routed;
```

295-297 如果待输出的数据报已准备好路由缓存，则无需再查找路由，直接跳转到 packet_routed 处作处理。

```
299      /* Make sure we can route this packet. */
300      rt = (struct rtable *)__sk_dst_check(sk, 0);
301      if (rt == NULL) {
302          __be32 daddr;
303
304          /* Use correct destination address if we have options. */
305          daddr = inet->daddr;
306          if(opt && opt->srr)
307              daddr = opt->faddr;
308
309          {
310              struct flowi fl = { .oif = sk->sk_bound_dev_if,
311                                  .nl_u = { .ip4_u =
312                                            { .daddr = daddr,
313                                              .saddr = inet->saddr,
314                                              .tos = RT_CONN_FLAGS(sk) } },
315                                  .proto = sk->sk_protocol,
316                                  .uli_u = { .ports =
317                                             { .sport = inet->sport,
318                                               .dport = inet->dport } } };
319
320              /* If this fails, retransmit mechanism of transport layer will
321               * keep trying until route appears or the connection times
322               * itself out.
323               */
324              security_sk_classify_flow(sk, &fl);
325              if (ip_route_output_flow(&rt, &fl, sk, 0))
326                  goto no_route;
327          }
328          sk_setup_caps(sk, &rt->u.dst);
329      }
```

```
330      skb->dst = dst_clone(&rt->u.dst);
```

如果输出该数据报的传输控制块中缓存了输出路由缓存项，则需检测该路由缓存项是否过期。

如果过期，重新通过输出网络设备、目的地址、源地址等信息查找输出路由缓存项。如果查找到对应的路由缓存项，则将其缓存到传输控制块中，否则丢弃该数据包。

如果未过期，则直接使用缓存在传输控制块中的路由缓存项。

```
332  packet_routed:
333      if (opt && opt->is_strictroute && rt->rt_dst != rt->rt_gateway)
334          goto no_route;
335
336      /* OK, we know where to send it, allocate and build IP header. */
337      iph =(struct iphdr *)skb_push(skb, sizeof(struct iphdr) + (opt ? opt->optlen :
0));
338      *((__be16 *)iph) = htons((4 << 12) | (5 << 8) | (inet->tos & 0xff));
339      iph->tot_len = htons(skb->len);
340      if (ip_dont_fragment(sk, &rt->u.dst) && !ipfragok)
341          iph->frag_off = htons(IP_DF);
342      else
343          iph->frag_off = 0;
344      iph->ttl      = ip_select_ttl(inet, &rt->u.dst);
345      iph->protocol = sk->sk_protocol;
346      iph->saddr    = rt->rt_src;
347      iph->daddr    = rt->rt_dst;
348      skb->nh.iph   = iph;
349      /* Transport layer set skb->h.foo itself. */
350
351      if (opt && opt->optlen) {
352          iph->ihl += opt->optlen >> 2;
353          ip_options_build(skb, opt, inet->daddr, rt, 0);
354      }
355
356      ip_select_ident_more(iph, &rt->u.dst, sk,
357                (skb_shinfo(skb)->gso_segs ?: 1) - 1);
358
359      /* Add an IP checksum. */
360      ip_send_check(iph);
361
362      skb->priority = sk->sk_priority;
363
364      return NF_HOOK(PF_INET, NF_IP_LOCAL_OUT, skb, NULL, rt->u.dst.dev,
365                 dst_output);
364
367  no_route:
368      IP_INC_STATS(IPSTATS_MIB_OUTNOROUTES);
369      kfree_skb(skb);
370      return -EHOSTUNREACH;
371  }
```

333 查找到输出路由后，先进行严格源路由选项的处理。如果存在严格源路由选项，并且数据报的下一跳地址和网关地址不一致，则丢弃该数据报。

337-360 设置 IP 首部中各字段的值。如果存在 IP 选项，则在 IP 数据报首部中构建 IP 选项。

362 设置输出数据报的 QoS 类别。

364-365 最后通过 netfilter 处理后，由 dst_output()处理数据报的输出。

367-370 如果查找不到对应的路由缓存项，在此处理，将该数据报丢弃。

2. ip_build_and_send_pkt()

此函数用于在 TCP 建立连接过程中，打包输出 SYN+ACK 类型的 TCP 段。参数说明如下：

- skb，待封装成 IP 数据报的 TCP 段。
- sk，输出该 TCP 段的传输控制块。
- saddr，输出该 TCP 段的源 IP 地址。
- daddr，输出该 TCP 段的目的 IP 地址。
- opt，IP 选项信息。

```
121  int ip_build_and_send_pkt(struct sk_buff *skb, struct sock *sk,
122                __be32 saddr, __be32 daddr, struct ip_options *opt)
123  {
124      struct inet_sock *inet = inet_sk(sk);
125      struct rtable *rt = (struct rtable *)skb->dst;
126      struct iphdr *iph;
127
128      /* Build the IP header. */
129      if (opt)
130          iph=(struct iphdr *)skb_push(skb,sizeof(struct iphdr) + opt->optlen);
131      else
132          iph=(struct iphdr *)skb_push(skb,sizeof(struct iphdr));
133
134      iph->version  = 4;
135      iph->ihl      = 5;
136      iph->tos      = inet->tos;
137      if (ip_dont_fragment(sk, &rt->u.dst))
138          iph->frag_off = htons(IP_DF);
139      else
140          iph->frag_off = 0;
141      iph->ttl      = ip_select_ttl(inet, &rt->u.dst);
142      iph->daddr    = rt->rt_dst;
143      iph->saddr    = rt->rt_src;
144      iph->protocol = sk->sk_protocol;
145      iph->tot_len  = htons(skb->len);
146      ip_select_ident(iph, &rt->u.dst, sk);
147      skb->nh.iph   = iph;
148
149      if (opt && opt->optlen) {
150          iph->ihl += opt->optlen>>2;
151          ip_options_build(skb, opt, daddr, rt, 0);
152      }
153      ip_send_check(iph);
154
155      skb->priority = sk->sk_priority;
156
157      /* Send it out. */
158      return NF_HOOK(PF_INET, NF_IP_LOCAL_OUT, skb, NULL, rt->u.dst.dev,
159                     dst_output);
160  }
```

129-132 为数据报建立 IP 首部。如果存在 IP 选项，则 IP 首部长度需适当地增加。

134-147 设置 IP 首部中的各域。其中 frag_off 根据输出套接口的 pmtudisc 值来设置，id 取值根据 IP 数据报是否分片而不同，不分片的 IP 数据报的 id 取自套接口中的 id 成员，而对于 IP 分片，则从对端信息块的 ip_id_count 中获取。

149-152 如果存在 IP 选项，则在 IP 数据报首部中构建 IP 选项。

153 为 IP 数据报计算校验和。

155 设置输出数据报的 QoS 类别。

158-159 最后通过 netfilter 处理后，调用 dst_output()处理数据报的输出。

3. ip_send_reply()

主要用于构成并输出 RST 和 ACK 段，在 tcp_v4_send_reset()和 tcp_v4_send_ack()中被调用。参数说明如下：

- sk，输出 TCP 段的传输控制块。
- skb，对方发送过来的 TCP 段。
- arg，传递给 ip_send_reply()的一些参数集合，包括待输出的数据、TCP 伪首部校验和以及 TCP 首部中校验和字段在首部中的偏移。

```
136 struct ip_reply_arg {
137     struct kvec iov[1];
138     __wsum       csum;
139     int       csumoffset; /* u16 offset of csum in iov[0].iov_base */
140               /* -1 if not needed */
141 };
```

137struct kvec iov[1]

标识待输出数据存储的位置及长度。类型如下所示：

```
28 struct kvec {
29     void *iov_base;/*and that should *never* hold a userland pointer*/
30     size_t iov_len;
31 };
```

iov_base 标识数据存储的位置，iov_len 则是数据的长度。

138 __wsum csum

TCP 伪首部的校验和。

139 int csumoffset

TCP 首部中校验和字段在首部中的偏移。通过偏移，在 IP 层就可以将计算得到的 TCP 校验和直接设置到 TCP 首部中。

- len，输出数据的长度。

```
1327 void ip_send_reply(struct sock *sk, struct sk_buff *skb, struct ip_reply_arg
     *arg,
1328           unsigned int len)
1329 {
1330     struct inet_sock *inet = inet_sk(sk);
1331     struct {
1332         struct ip_options   opt;
1333         char           data[40];
```

```
1334    } replyopts;
1335    struct ipcm_cookie ipc;
1336    __be32 daddr;
1337    struct rtable *rt = (struct rtable*)skb->dst;
1338
1339    if (ip_options_echo(&replyopts.opt, skb))
1340       return;
1341
1342    daddr = ipc.addr = rt->rt_src;
1343    ipc.opt = NULL;
1344
1345    if (replyopts.opt.optlen) {
1346       ipc.opt = &replyopts.opt;
1347
1348       if (ipc.opt->srr)
1349          daddr = replyopts.opt.faddr;
1350    }
1351
1352    {
1353       struct flowi fl = { .nl_u = { .ip4_u =
1354                     { .daddr = daddr,
1355                    .saddr = rt->rt_spec_dst,
1356                    .tos = RT_TOS(skb->nh.iph->tos) } },
1357                  /* Not quite clean, but right. */
1358                  .uli_u = { .ports =
1359                       { .sport = skb->h.th->dest,
1360                         .dport = skb->h.th->source } },
1361                  .proto = sk->sk_protocol };
1362       security_skb_classify_flow(skb, &fl);
1363       if (ip_route_output_key(&rt, &fl))
1364          return;
1365    }
1366
1367    /* And let IP do all the hard work.
1368
1369       This chunk is not reenterable, hence spinlock.
1370       Note that it uses the fact, that this function is called
1371       with locally disabled BH and that sk cannot be already spinlocked.
1372     */
1373    bh_lock_sock(sk);
1374    inet->tos = skb->nh.iph->tos;
1375    sk->sk_priority = skb->priority;
1376    sk->sk_protocol = skb->nh.iph->protocol;
1377    ip_append_data(sk, ip_reply_glue_bits, arg->iov->iov_base, len, 0,
1378              &ipc, rt, MSG_DONTWAIT);
1379    if ((skb = skb_peek(&sk->sk_write_queue)) != NULL) {
1380       if (arg->csumoffset >= 0)
1381          *((__sum16*)skb->h.raw+arg->csumoffset)=csum_fold(csum_add(skb->csum,
    arg->csum));
1382       skb->ip_summed = CHECKSUM_NONE;
1383       ip_push_pending_frames(sk);
1384    }
```

```
1385
1386     bh_unlock_sock(sk);
1387
1388     ip_rt_put(rt);
1389 }
```

1339-1340 从待输出的 IP 数据报中得到选项，用于处理源路由选项。

1342-1343 根据对方发送过来的数据报的输入路由，获取对方的 IP 地址。

1345-1350 如果输入的 IP 数据报启用了源路由选项，则将得到下一跳的 IP 地址作为目的地址。

1352-1365 根据目的地址、源地址等查找输出到对方的路由。如果查找命中，则可以输出数据报，否则中止输出。

1374-1376 根据输入的 SKB 更新一些属性到传输控制块中，如 TOS、优先级等。

1377-1384 先将数据添加到输出队列末尾的 SKB 中，或将数据复制到新生成的 SKB 中并添加到输出队列中。然后，如果输出队列不为空，则计算第一个 SKB 的传输层校验和，并将其发送出去。

11.11.3　UDP 输出的接口

1. ip_append_data()

如果说 ip_append_data()只是 UDP 套接口和 RAW 套接口的输出接口，也不完全正确，因为在 TCP 中用于发送 ACK 和 RST 段的函数 ip_send_reply()最终也调用了该函数。

ip_append_data()是一个比较复杂的函数，主要是将接收到的大数据包分成多个小于或等于 MTU 的 SKB，为网络层要实现的 IP 分片作准备。例如，假设待发送的数据包大小为 4000B，先前输出队列非空，且最后一个 SKB 还没填满，剩余 500B。这时传输层调用 ip_append_data()，则首先会将有剩余空间的 SKB 填满。当网络设备支持聚合分散 I/O 时，便会将数据写到 frags 指向的页面中，如果相关的页面已经填满，则会再分配一个新的页面。接着，进入下次循环，每次循环都分配一个 SKB，通过 getfrag 将数据从传输层复制数据，并将其添加到输出队列的末尾，直至复制完所有待输出的数据，流程图见 11-8。

ip_append_data()在多处被调用，包括 UDP、TCP、RAW 套接口以及 ICMP。因此在复制数据时，有时只复制传输层负载部分，传输层首部会后续添加（UDP），有时则需复制包括传输层首部的全部数据（ICMP）。参数说明如下：

（1）sk，输出数据的传输控制块。该传输控制块还提供其他一些信息，如 IP 选项等。

（2）getfrag，用于复制数据到 SKB 中。不同的传输层，由于特性不同，因此对应复制的方法也不一样，见表 11-12。函数原型如下所示：

```
int getfrag(void *from, char *to, int offset, int len, int odd, struct sk_buff *skb)
```

其中参数说明如下：

- from，标识待复制数据存储的位置。
- to，标识数据待复制到的目的地。
- offset，待复制数据在数据存储位置的偏移，数据从此位置开始复制。
- len，待复制数据的长度。

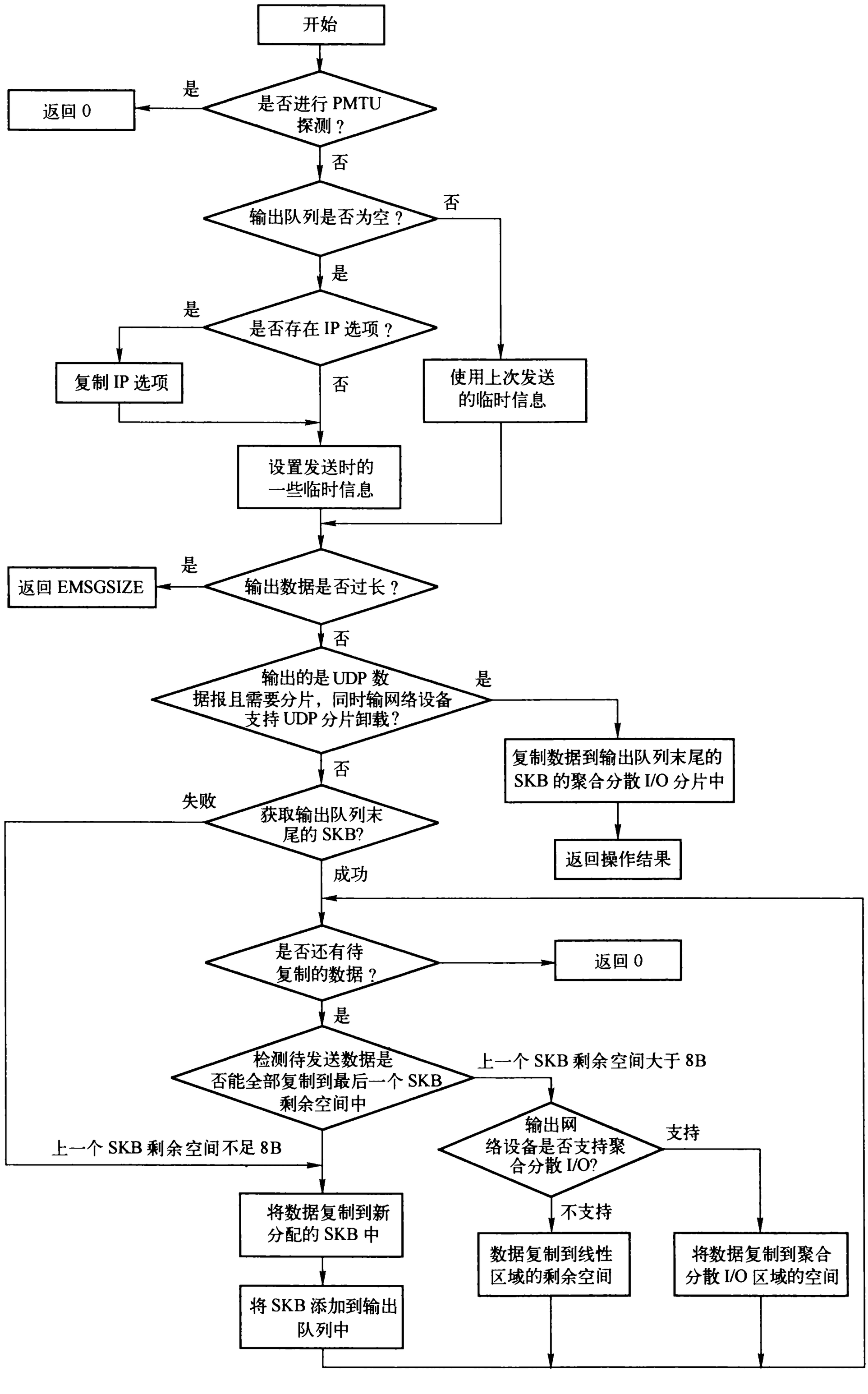

图 11-8　ip_append_data()流程图

- odd，从上一个 SKB 中剩余下来并复制到此 SKB 中的数据长度。如果为奇数，则后续数据的校验和计算时的 16 位数据的高 8 位和低 8 位的值是颠倒的，因此需要将后续数据的校验和高低 8 位对调。
- skb，复制数据的 SKB，计算得到的数据部分的校验和暂存到 SKB 中，为计算完成的传输层校验和做准备。

表 11-12　多个 getfrag()

不同 getfrag()	描述

ip_generic_getfrag	用于复制 UDP 套接口和 RAW 套接口的数据。参数 from 指向的地址为 iovec 结构数组，每个 IO 向量中的数据存储的位置由 iov_base 标识，长度由 iov_len 标识
	struct iovec
	{
	void __user *iov_base;　/* BSD uses caddr_t (1003.1g requires void *) */
	__kernel_size_t iov_len; /* Must be size_t (1003.1g) */
	};

udplite_getfrag	用于复制轻量级 UDP 的数据。参数 from 指向的地址为 iovec 结构，待复制数据存储的位置和长度同样由 iovec 结构标识
ip_reply_glue_bits	用于在 TCP 中复制 RST 和 ACK 段的数据。参数 from 指向的地址直接为数据待存储的位置
icmp_glue_bits	用于复制 ICMP 报文。参数 from 指向的地址为 icmp_bxm 结构，待复制数据存储的位置由 icmp_bxm 结构的 skb 和 offset 标识

（3）from，输出数据所在的数据块地址，它指向用户空间或内核空间，该参数会传递给 getfrag()。

（4）length，输出数据的长度。

（5）transhdrlen，传输层首部长度。

（6）ipc，传递到 IP 层的临时信息块。

（7）rt，输出该数据的路由缓存项，在调用此函数之前传输控制块已经缓存路由缓存项或者已经通过 ip_route_output_flow()查找到了输出数据的路由缓存项。

（8）flags，输出数据的一些标志（例如 MSG_MORE 等），见表 23-1。

```
764  int ip_append_data(struct sock *sk,
765             int getfrag(void *from, char *to, int offset, int len,
766                    int odd, struct sk_buff *skb),
767             void *from, int length, int transhdrlen,
768             struct ipcm_cookie *ipc, struct rtable *rt,
769             unsigned int flags)
770  {
771      struct inet_sock *inet = inet_sk(sk);
772      struct sk_buff *skb;
773
774      struct ip_options *opt = NULL;
775      int hh_len;
776      int exthdrlen;
777      int mtu;
778      int copy;
779      int err;
780      int offset = 0;
781      unsigned int maxfraglen, fragheaderlen;
782      int csummode = CHECKSUM_NONE;
783
784      if (flags&MSG_PROBE)
```

```
785         return 0;
```

776 exthdrlen 用于记录 IPsec 中扩展首部的长度，未启用 IPsec 时为 0。

784-785 如果使用 MSG_PROBE 标识，实际上并不会进行真正的数据传递，而是进行路径 MTU 的探测。

```
787     if (skb_queue_empty(&sk->sk_write_queue)) {
788        /*
789         * setup for corking.
790         */
791        opt = ipc->opt;
792        if (opt) {
793           if (inet->cork.opt == NULL) {
794                    inet->cork.opt = kmalloc(sizeof(struct ip_options) + 40,
    sk->sk_allocation);
795              if (unlikely(inet->cork.opt == NULL))
796                 return -ENOBUFS;
797           }
798           memcpy(inet->cork.opt, opt, sizeof(struct ip_options)+opt->optlen);
799           inet->cork.flags |= IPCORK_OPT;
800           inet->cork.addr = ipc->addr;
801        }
802        dst_hold(&rt->u.dst);
803        inet->cork.fragsize = mtu = dst_mtu(rt->u.dst.path);
804        inet->cork.rt = rt;
805        inet->cork.length = 0;
806        sk->sk_sndmsg_page = NULL;
807        sk->sk_sndmsg_off = 0;
808        if ((exthdrlen = rt->u.dst.header_len) != 0) {
809           length += exthdrlen;
810           transhdrlen += exthdrlen;
811        }
812     } else {
813        rt = inet->cork.rt;
814        if (inet->cork.flags & IPCORK_OPT)
815           opt = inet->cork.opt;
816
817        transhdrlen = 0;
818        exthdrlen = 0;
819        mtu = inet->cork.fragsize;
820     }
```

787-811 如果传输控制块的输出队列为空，则需要为传输控制块设置一些临时信息。

如果输出数据报中存在 IP 选项，则将 IP 选项信息复制到临时信息块中，并设置 IPCORK_OPT，表示临时信息块中存在 IP 选项。由于存在 IP 选项，因此需设置临时信息块中的目的地址，因为在 IP 选项中存在源路由选项。

同时还设置 IP 数据报分片大小，输出路由缓存、初始化当前发送数据报中数据的长度（如果启用了 IPSec，则还要加上 IPSec 首部的长度）等。

812-820 如果传输控制块的输出队列不为空，则使用上次的输出路由、IP 选项以及分片长度。

```
821     hh_len = LL_RESERVED_SPACE(rt->u.dst.dev);
822
823     fragheaderlen = sizeof(struct iphdr) + (opt ? opt->optlen : 0);
```

```
824     maxfraglen = ((mtu - fragheaderlen) & ~7) + fragheaderlen;
825
826     if (inet->cork.length + length > 0xFFFF - fragheaderlen) {
827        ip_local_error(sk, EMSGSIZE, rt->rt_dst, inet->dport, mtu-exthdrlen);
828        return -EMSGSIZE;
829     }
830
831     /*
832      * transhdrlen > 0 means that this is the first fragment and we wish
833      * it won't be fragmented in the future.
834      */
835     if (transhdrlen &&
836        length + fragheaderlen <= mtu &&
837        rt->u.dst.dev->features & NETIF_F_ALL_CSUM &&
838        !exthdrlen)
839        csummode = CHECKSUM_PARTIAL;
840
841     inet->cork.length += length;
842     if (((length > mtu) && (sk->sk_protocol == IPPROTO_UDP)) &&
843           (rt->u.dst.dev->features & NETIF_F_UFO)) {
844
845        err = ip_ufo_append_data(sk, getfrag, from, length, hh_len,
846                  fragheaderlen, transhdrlen, mtu,
847                  flags);
848        if (err)
849           goto error;
850        return 0;
851     }
```

821-823 获取链路层首部及 IP 首部（包括选项）的长度。

824 IP 数据报的数据需 4 字节对齐，为加速计算直接将 IP 数据报的数据根据当前 MTU 8 字节对齐，然后重新得到用于分片的长度。

826-829 如果输出的数据长度超出一个 IP 数据报能容纳的长度，则向输出该数据报的套接口发送 EMSGSIZE 出错信息。

835-839 如果 IP 数据报没有分片，且输出网络设备支持硬件执行校验和，则设置 CHECKSUM_PARTIAL，表示由硬件来执行校验和。

841-851 如果输出的是 UDP 数据报且需要分片，同时输出网络设备支持 UDP 分片卸载（UDP fragmentation offload），则由 ip_ufo_append_data()进行分片输出处理。

```
853     /* So, what's going on in the loop below?
854      *
855      * We use calculated fragment length to generate chained skb,
856      * each of segments is IP fragment ready for sending to network after
857      * adding appropriate IP header.
858      */
859
860     if ((skb = skb_peek_tail(&sk->sk_write_queue)) == NULL)
861        goto alloc_new_skb;
862
863     while (length > 0) {
864        /* Check if the remaining data fits into current packet. */
865        copy = mtu - skb->len;
866        if (copy < length)
```

```
867             copy = maxfraglen - skb->len;
868         if (copy <= 0) {
869             char *data;
870             unsigned int datalen;
871             unsigned int fraglen;
872             unsigned int fraggap;
873             unsigned int alloclen;
874             struct sk_buff *skb_prev;
875 alloc_new_skb:
876             skb_prev = skb;
877             if (skb_prev)
878                 fraggap = skb_prev->len - maxfraglen;
879             else
880                 fraggap = 0;
881
882             /*
883              * If remaining data exceeds the mtu,
884              * we know we need more fragment(s).
885              */
886             datalen = length + fraggap;
887             if (datalen > mtu - fragheaderlen)
888                 datalen = maxfraglen - fragheaderlen;
889             fraglen = datalen + fragheaderlen;
890
891             if ((flags & MSG_MORE) &&
892                 !(rt->u.dst.dev->features&NETIF_F_SG))
893                 alloclen = mtu;
894             else
895                 alloclen = datalen + fragheaderlen;
896
897             /* The last fragment gets additional space at tail.
898              * Note, with MSG_MORE we overallocate on fragments,
899              * because we have no idea what fragment will be
900              * the last.
901              */
902             if (datalen == length + fraggap)
903                 alloclen += rt->u.dst.trailer_len;
904
905             if (transhdrlen) {
906                 skb = sock_alloc_send_skb(sk,
907                         alloclen + hh_len + 15,
908                         (flags & MSG_DONTWAIT), &err);
909             } else {
910                 skb = NULL;
911                 if (atomic_read(&sk->sk_wmem_alloc) <=
912                     2 * sk->sk_sndbuf)
913                     skb = sock_wmalloc(sk,
914                                alloclen + hh_len + 15, 1,
915                                sk->sk_allocation);
916                 if (unlikely(skb == NULL))
917                     err = -ENOBUFS;
918             }
919             if (skb == NULL)
920                 goto error;
921
922             /*
```

```
923             *    Fill in the control structures
924             */
925            skb->ip_summed = csummode;
926            skb->csum = 0;
927            skb_reserve(skb, hh_len);
928
929            /*
930             *    Find where to start putting bytes.
931             */
932            data = skb_put(skb, fraglen);
933            skb->nh.raw = data + exthdrlen;
934            data += fragheaderlen;
935            skb->h.raw = data + exthdrlen;
936
937            if (fraggap) {
938               skb->csum = skb_copy_and_csum_bits(
939                  skb_prev, maxfraglen,
940                  data + transhdrlen, fraggap, 0);
941               skb_prev->csum = csum_sub(skb_prev->csum,
942                           skb->csum);
943               data += fraggap;
944               pskb_trim_unique(skb_prev, maxfraglen);
945            }
946
947            copy = datalen - transhdrlen - fraggap;
948            if (copy > 0 && getfrag(from, data + transhdrlen, offset, copy, fraggap,
    skb) < 0) {
949               err = -EFAULT;
950               kfree_skb(skb);
951               goto error;
952            }
953
954            offset += copy;
955            length -= datalen - fraggap;
956            transhdrlen = 0;
957            exthdrlen = 0;
958            csummode = CHECKSUM_NONE;
959
960            /*
961             * Put the packet on the pending queue.
962             */
963            __skb_queue_tail(&sk->sk_write_queue, skb);
964            continue;
965         }
966
967         if (copy > length)
968            copy = length;
969
970         if (!(rt->u.dst.dev->features&NETIF_F_SG)) {
971            unsigned int off;
972
973            off = skb->len;
974            if (getfrag(from, skb_put(skb, copy),
975                     offset, copy, off, skb) < 0) {
976               __skb_trim(skb, off);
977               err = -EFAULT;
```

```
978                goto error;
979             }
980          } else {
981             int i = skb_shinfo(skb)->nr_frags;
982             skb_frag_t *frag = &skb_shinfo(skb)->frags[i-1];
983             struct page *page = sk->sk_sndmsg_page;
984             int off = sk->sk_sndmsg_off;
985             unsigned int left;
986
987             if (page && (left = PAGE_SIZE - off) > 0) {
988                if (copy >= left)
989                   copy = left;
990                if (page != frag->page) {
991                   if (i == MAX_SKB_FRAGS) {
992                      err = -EMSGSIZE;
993                      goto error;
994                   }
995                   get_page(page);
996                    skb_fill_page_desc(skb, i, page, sk->sk_sndmsg_off, 0);
997                   frag = &skb_shinfo(skb)->frags[i];
998                }
999             } else if (i < MAX_SKB_FRAGS) {
1000               if (copy > PAGE_SIZE)
1001                  copy = PAGE_SIZE;
1002               page = alloc_pages(sk->sk_allocation, 0);
1003               if (page == NULL)  {
1004                  err = -ENOMEM;
1005                  goto error;
1006               }
1007               sk->sk_sndmsg_page = page;
1008               sk->sk_sndmsg_off = 0;
1009
1010               skb_fill_page_desc(skb, i, page, 0, 0);
1011               frag = &skb_shinfo(skb)->frags[i];
1012               skb->truesize += PAGE_SIZE;
1013               atomic_add(PAGE_SIZE, &sk->sk_wmem_alloc);
1014            } else {
1015               err = -EMSGSIZE;
1016               goto error;
1017            }
1018            if(getfrag(from,page_address(frag->page)+frag->page_offset+frag->size,
    offset, copy, skb->len, skb) < 0) {
1019               err = -EFAULT;
1020               goto error;
1021            }
1022            sk->sk_sndmsg_off += copy;
1023            frag->size += copy;
1024            skb->len += copy;
1025            skb->data_len += copy;
1026         }
1027         offset += copy;
1028         length -= copy;
1029      }
1030
1031      return 0;
```

860-861 获取输出队列末尾的 SKB，如果获取不到，说明输出队列为空，则需分配一个新的 SKB 用于复制数据。

863 循环处理待输出数据，直至所有的数据都处理完成。

865-867 检测待发送数据是否能全部复制到最后一个 SKB 的剩余空间中。如果可以，则说明是 IP 分片中的上一个分片，可以不用 4 字节对齐，否则需要 4 字节对齐，因此用 8 字节对齐后的 MTU 减去上一个 SKB 的数据长度，得到上一个 SKB 的剩余空间大小，也就是本次复制数据的长度。

当本次复制数据的长度 copy 小于或等于 0 时，说明上一个 SKB 已经填满或空间不足 8B，需要分配新的 SKB。

当 copy 大于 0 时，说明上一个 SKB 有剩余空间，数据可以复制到该 SKB 中去。

868-895 如果上一个 SKB 已经填满或空间不足 8B，或者不存在上一个 SKB，则将数据复制到新分配的 SKB 中去。

876-880 如果上一个 SKB（通常是在调用 ip_append_data()时，输出队列中的最后一个 SKB）中存在多于 8 字节对齐的 MTU 的数据，则这些数据需移动到当前 SKB 中，确保最后一个 IP 分片之外的数据能够 4 字节对齐，因此要计算移动到当前 SKB 的数据长度。

886-895 如果剩余数据的长度超过 MTU，则需要更多的分片。

886 计算需复制到新 SKB 中的数据长度。因为如果前一个 SKB 还能容纳数据，则有一部分数据会复制到前一个 SKB 中。

887-888 如果剩余的数据一个分片不够容纳，则根据 MTU 重新计算本次可发送的数据长度。

889 根据本次复制的数据长度以及 IP 首部长度，计算三层首部及其数据的总长度。

891-895 如果后续还有数据要输出且网络设备不支持聚合分散 I/O，则将 MTU 作为分配 SKB 的长度，使分片达到最长，为后续的数据预备空间。否则按数据的长度（包括 IP 首部）分配 SKB 的空间即可。

902-903 如果是最后一个分片，且是根据目的路由启用 IPsec 的情况，则可能需要多分配一些空间来支持 IPsec。

905-920 根据是否存在传输层首部，确定用何种方法分配 SKB。

如果存在传输层首部，则可以确定该分片为分片组中的第一个分片，因此在分配 SKB 时需要考虑更多的情况，如输出操作是否超时，传输层是否发生未处理的致命错误，发送通道是否已关闭等。当分片不是第一个分片时，则无需考虑以上情况。

925-926 填充用于校验的控制信息。

927-935 为数据报预留用于存放二层首部、三层首部和数据的空间，并设置 SKB 中指向三层和四层的指针。

937-945 如果上一个 SKB 的数据超过 8 字节对齐 MTU，则将超出数据和传输层首部复制到当前 SKB，重新计算校验和，并以 8 字节对齐 MTU 为长度截取上一个 SKB 的数据。

947-953 传输层首部和上个 SKB 多出的数据已复制，接着复制剩下的数据。

954-958 完成本次复制数据，计算下次需复制数据的地址及剩余数据的长度。传输层首部已经复制，因此需要将传输层首部的 transhdrlen 置为 0，同时 IPsec 首部长度 exthdrlen 也置为 0。

963-964 将复制完数据的 SKB 添加到输出队列的尾部，接着复制剩下的数据。

967-968 如果上个 SKB 剩余的空间大于剩余待发送的数据长度，则剩下的数据可以一次完成。

970-979 如果输出网络设备不支持聚合分散 I/O，则将数据复制到线性区域的剩余空间。

980-1026 如果输出网络设备支持聚合分散 I/O，则将数据复制到聚合分散 I/O 区域的空间。

981-985 获取分片数组中最后一个分片指向的页面，以及已使用数据在分片的页内偏移。

987-998 如果缓存页面存在且尚有剩余空间，则先使用缓存页面，将数据复制到缓存页面。

988-989 如果待复制数据大于页面中剩余的空间，则根据页面剩余空间重新设定本次复制数据的长度，确保待复制数据的长度不超过页面中剩余空间的长度。

990-998 如果传输控制块的缓存页不是最后一个聚合分散 I/O 页面，则直接使用该页面，将其添加到聚合分散 I/O 页面数组中，并确定将数据复制到的页面。

999-1013 如果缓存页面不存在或页面没有剩余空间，且传输控制块的聚合分散 I/O 页面数量未达到上限，则添加新的分片，并分配新的页面。

1000-1001 如果待复制数据大于页面的长度，则根据页面长度重新设定本次复制数据的长度，确保待复制的数据长度不超过页面的长度。

1002-1013 分配页面并安装到 SKB 的聚合分散 I/O 页面数组中，确定将数据复制到该页面。同时刷新传输控制块中指向的缓存页面，更新传输控制块发送使用的缓存大小。

1014-1017 如果缓存页面不存在或页面没有剩余空间，且传输控制块的聚合分散 I/O 缓存块数达到上限，则说明待发送的数据太长，已经超过 IP 数据报的最大长度。

1018-1021 将待发送的数据复制到指定的页面上，并根据校验和标志确定执行校验和的方式。

1022-1025 复制完数据后，更新数据尾端在最后一个分片页内的偏移及分片页面中已使用的长度、SKB 的数据部分长度、分片中数据的长度。

1027-1028 完成本次复制数据，计算下次需要复制数据的地址及剩余数据的长度。

```
1033 error:
1034     inet->cork.length -= length;
1035     IP_INC_STATS(IPSTATS_MIB_OUTDISCARDS);
1036     return err;
1037 }
```

处理复制过程中的错误。

2. ip_ufo_append_data()

ip_ufo_append_data()实现复制数据到输出队列末尾那个 SKB 的聚合分散 I/O 页面中，参数参见 ip_append_data()。

```
698  static inline int ip_ufo_append_data(struct sock *sk,
699             int getfrag(void *from, char *to, int offset, int len,
700                    int odd, struct sk_buff *skb),
701            void *from, int length, int hh_len, int fragheaderlen,
702            int transhdrlen, int mtu,unsigned int flags)
703  {
704     struct sk_buff *skb;
705     int err;
706
707     /* There is support for UDP fragmentation offload by network
708      * device, so create one single skb packet containing complete
709      * udp datagram
710      */
711     if ((skb = skb_peek_tail(&sk->sk_write_queue)) == NULL) {
712        skb = sock_alloc_send_skb(sk,
713           hh_len + fragheaderlen + transhdrlen + 20,
```

```
714             (flags & MSG_DONTWAIT), &err);
715
716         if (skb == NULL)
717             return err;
718
719         /* reserve space for Hardware header */
720         skb_reserve(skb, hh_len);
721
722         /* create space for UDP/IP header */
723         skb_put(skb,fragheaderlen + transhdrlen);
724
725         /* initialize network header pointer */
726             skb->nh.raw = skb->data;
727
728             /* initialize protocol header pointer */
729             skb->h.raw = skb->data + fragheaderlen;
730
731             skb->ip_summed = CHECKSUM_PARTIAL;
732             skb->csum = 0;
733             sk->sk_sndmsg_off = 0;
734         }
735
736         err = skb_append_datato_frags(sk,skb, getfrag, from,
737                       (length - transhdrlen));
738         if (!err) {
739             /* specify the length of each IP datagram fragment*/
740             skb_shinfo(skb)->gso_size = mtu - fragheaderlen;
741             skb_shinfo(skb)->gso_type = SKB_GSO_UDP;
742             __skb_queue_tail(&sk->sk_write_queue, skb);
743
744             return 0;
745         }
746         /* There is not enough support do UFO ,
747          * so follow normal path
748          */
749         kfree_skb(skb);
750         return err;
751     }
```

711-734 如果输出队列不为空，则获取输出队列末尾的那个 SKB，否则分配新的 SKB，并为新分配的 SKB 在线性空间中保留出链路层、网络层、传输层首部空间，然后初始化 SKB 的相关校验信息，初始化传输控制块的 sk_sndmsg_off，表示分片内没有数据。

736-750 将数据复制到聚合分散 I/O 页面中，若复制成功，则设置 SKB 的 gso_size 和 gso_type，然后将 SKB 添加到输出队列末尾；否则释放该 SKB。

（注：笔者在阅读这段代码时，发现一个 bug，在输出队列未空的情况下，获得队列末尾的 SKB，但并未把它取下，因此该 SKB 还处于输出队列末尾，如果将数据成功复制到聚合分散 I/O 页面，则会将 SKB 再一次添加到队列中，这样虽然不会出错，但是多余。而如果数据复制失败，则最后会释放该 SKB，但因为并没有从输出队列取下该 SKB 而直接释放了，会导致输出队列异常。）

skb_append_datato_frags()将数据复制到 SKB 的聚合分散 I/O 的页面中，直到数据复制完成或 SKB 空间已满，参数参见 ip_append_data()。

```
1838 int skb_append_datato_frags(struct sock *sk, struct sk_buff *skb,
1839          int (*getfrag)(void *from, char *to, int offset,
1840                int len, int odd, struct sk_buff *skb),
1841          void *from, int length)
1842 {
1843    int frg_cnt = 0;
1844    skb_frag_t *frag = NULL;
1845    struct page *page = NULL;
1846    int copy, left;
1847    int offset = 0;
1848    int ret;
1849
1850    do {
1851       /* Return error if we don't have space for new frag */
1852       frg_cnt = skb_shinfo(skb)->nr_frags;
1853       if (frg_cnt >= MAX_SKB_FRAGS)
1854          return -EFAULT;
1855
1856       /* allocate a new page for next frag */
1857       page = alloc_pages(sk->sk_allocation, 0);
1858
1859       /* If alloc_page fails just return failure and caller will
1860        * free previous allocated pages by doing kfree_skb()
1861        */
1862       if (page == NULL)
1863          return -ENOMEM;
1864
1865       /* initialize the next frag */
1866       sk->sk_sndmsg_page = page;
1867       sk->sk_sndmsg_off = 0;
1868       skb_fill_page_desc(skb, frg_cnt, page, 0, 0);
1869       skb->truesize += PAGE_SIZE;
1870       atomic_add(PAGE_SIZE, &sk->sk_wmem_alloc);
1871
1872       /* get the new initialized frag */
1873       frg_cnt = skb_shinfo(skb)->nr_frags;
1874       frag = &skb_shinfo(skb)->frags[frg_cnt - 1];
1875
1876       /* copy the user data to page */
1877       left = PAGE_SIZE - frag->page_offset;
1878       copy = (length > left)? left : length;
1879
1880       ret = getfrag(from, (page_address(frag->page) +
1881             frag->page_offset + frag->size),
1882             offset, copy, 0, skb);
1883       if (ret < 0)
1884          return -EFAULT;
1885
1886       /* copy was successful so update the size parameters */
1887       sk->sk_sndmsg_off += copy;
1888       frag->size += copy;
1889       skb->len += copy;
1890       skb->data_len += copy;
1891       offset += copy;
1892       length -= copy;
1893
```

```
1894     } while (length > 0);
1895
1896     return 0;
1897 }
```

1852-1854 获取 SKB 当前聚合分散 I/O 页面数，用于检测 frags 数组是否已用完。

1857-1870 如果 frags 数组还可以存放页面，则分配一个空闲页面并安装到 frags 数组中。然后更新传输控制块缓存最近一次分配的页面、空闲区域在页面内的偏移，以及 SKB 的大小和为发送而分配的所有 SKB 数据区的总大小。

1873-1884 将数据复制到新分配的分页中。

1887-1890 更新空闲区域在页面内的偏移、数据在分片的文件系统缓存页面中使用的长度及 SKB 数据长度。

1891-1892 完成本次复制数据，计算下次需复制数据的地址以及剩余数据长度。

3. ip_push_pending_frames()

将输出队列上的多个分片合成一个完整的 IP 数据报，并通过 ip_output()输出。

```
1185 int ip_push_pending_frames(struct sock *sk)
1186 {
1187     struct sk_buff *skb, *tmp_skb;
1188     struct sk_buff **tail_skb;
1189     struct inet_sock *inet = inet_sk(sk);
1190     struct ip_options *opt = NULL;
1191     struct rtable *rt = inet->cork.rt;
1192     struct iphdr *iph;
1193     __be16 df = 0;
1194     __u8 ttl;
1195     int err = 0;
1196
1197     if ((skb = __skb_dequeue(&sk->sk_write_queue)) == NULL)
1198         goto out;
1199     tail_skb = &(skb_shinfo(skb)->frag_list);
1200
1201     /* move skb->data to ip header from ext header */
1202     if (skb->data < skb->nh.raw)
1203         __skb_pull(skb, skb->nh.raw - skb->data);
1204     while ((tmp_skb = __skb_dequeue(&sk->sk_write_queue)) != NULL) {
1205         __skb_pull(tmp_skb, skb->h.raw - skb->nh.raw);
1206         *tail_skb = tmp_skb;
1207         tail_skb = &(tmp_skb->next);
1208         skb->len += tmp_skb->len;
1209         skb->data_len += tmp_skb->len;
1210         skb->truesize += tmp_skb->truesize;
1211         __sock_put(tmp_skb->sk);
1212         tmp_skb->destructor = NULL;
1213         tmp_skb->sk = NULL;
1214     }
1215
1216     /* Unless user demanded real pmtu discovery (IP_PMTUDISC_DO), we allow
1217      * to fragment the frame generated here. No matter, what transforms
1218      * how transforms change size of the packet, it will come out.
1219      */
1220     if (inet->pmtudisc != IP_PMTUDISC_DO)
1221         skb->local_df = 1;
```

```
1222
1223    /* DF bit is set when we want to see DF on outgoing frames.
1224     * If local_df is set too, we still allow to fragment this frame
1225     * locally. */
1226    if (inet->pmtudisc == IP_PMTUDISC_DO ||
1227        (skb->len <= dst_mtu(&rt->u.dst) &&
1228         ip_dont_fragment(sk, &rt->u.dst)))
1229        df = htons(IP_DF);
1230
1231    if (inet->cork.flags & IPCORK_OPT)
1232        opt = inet->cork.opt;
1233
1234    if (rt->rt_type == RTN_MULTICAST)
1235        ttl = inet->mc_ttl;
1236    else
1237        ttl = ip_select_ttl(inet, &rt->u.dst);
1238
1239    iph = (struct iphdr *)skb->data;
1240    iph->version = 4;
1241    iph->ihl = 5;
1242    if (opt) {
1243        iph->ihl += opt->optlen>>2;
1244        ip_options_build(skb, opt, inet->cork.addr, rt, 0);
1245    }
1246    iph->tos = inet->tos;
1247    iph->tot_len = htons(skb->len);
1248    iph->frag_off = df;
1249    ip_select_ident(iph, &rt->u.dst, sk);
1250    iph->ttl = ttl;
1251    iph->protocol = sk->sk_protocol;
1252    iph->saddr = rt->rt_src;
1253    iph->daddr = rt->rt_dst;
1254    ip_send_check(iph);
1255
1256    skb->priority = sk->sk_priority;
1257    skb->dst = dst_clone(&rt->u.dst);
1258
1259    /* Netfilter gets whole the not fragmented skb. */
1260    err = NF_HOOK(PF_INET, NF_IP_LOCAL_OUT, skb, NULL,
1261                  skb->dst->dev, dst_output);
1262    if (err) {
1263        if (err > 0)
1264            err = inet->recverr ? net_xmit_errno(err) : 0;
1265        if (err)
1266            goto error;
1267    }
1268
1269 out:
1270    inet->cork.flags &= ~IPCORK_OPT;
1271    kfree(inet->cork.opt);
1272    inet->cork.opt = NULL;
1273    if (inet->cork.rt) {
1274        ip_rt_put(inet->cork.rt);
1275        inet->cork.rt = NULL;
1276    }
1277    return err;
1278
1279 error:
1280    IP_INC_STATS(IPSTATS_MIB_OUTDISCARDS);
```

```
1281     goto out;
1282 }
```

1197-1198 如果输出队列为空，则无需再作输出处理了。

1199 获取输出队列中 fraglist 链表，用于存放处理后的分片。

1203 如果 SKB 的 data 指针位置不准，调整到 IP 首部处，因为接着处理的是 IP 数据报，需要填充 IP 首部。

1204-1214 去除后续 SKB 中的 IP 首部后，链接到第一个 SKB 的 fraglist 上，组成一个分片，为后续的分片作准备。

1220-1221 在不启用路径 MTU 发现时，允许对输出数据报进行分片。

1226-1229 如果启用了路径 MTU 发现功能，或者输出数据报的长度小于 MTU 且本传输控制块输出的 IP 数据报不能分片，则给 IP 首部添加禁止分片标志。

1231-1232 如果 IP 选项信息已保存到传输控制块中，则获取 IP 选项信息指针，准备用于构建 IP 首部中的选项。

1234-1237 获取生存时间，用来设置到待输出的 IP 首部中。

1239-1254 构建 IP 首部，设置 IP 首部中各字段，包括 IP 选项等。

1256-1257 设置输出数据报的优先级及目的路由。

1260-1267 通过 NF_IP_LOCAL_OUT 的 netfilter 处理后，由 dst_output()输出数据报。

1269-1277 无论是否成功输出 IP 数据报，完成后都要删除保存到传输控制块中的 IP 选项信息。

1279-1281 处理输出数据报发生错误的情况。

11.12　IP 层对 GSO 的支持

IP 层 GSO 操作接口只是提供接口给链路层来访问传输层，因此 IP 层实现的 GSO 接口并没有过多的功能，而更多地在于通过传输层接口调用传输层提供的 GSO 操作接口，从而实现从链路层访问传输层提供的 GSO 分段功能。

11.12.1　inet_gso_segment()

inet_gso_segment()是 IP 层 gso_segment 接口的实现函数，根据分段数据报获取对应的传输层接口，并完成 GSO 分段后，对分段后的 IP 数据报重新计算校验和。

```
1135 static struct sk_buff *inet_gso_segment(struct sk_buff *skb, int features)
1136 {
1137     struct sk_buff *segs = ERR_PTR(-EINVAL);
1138     struct iphdr *iph;
1139     struct net_protocol *ops;
1140     int proto;
1141     int ihl;
1142     int id;
1143
1144     if (unlikely(skb_shinfo(skb)->gso_type &
1145             ~(SKB_GSO_TCPV4 |
1146               SKB_GSO_UDP |
1147               SKB_GSO_DODGY |
```

```
1148                SKB_GSO_TCP_ECN |
1149                0)))
1150        goto out;
1151
1152    if (unlikely(!pskb_may_pull(skb, sizeof(*iph))))
1153        goto out;
1154
1155    iph = skb->nh.iph;
1156    ihl = iph->ihl * 4;
1157    if (ihl < sizeof(*iph))
1158        goto out;
1159
1160    if (unlikely(!pskb_may_pull(skb, ihl)))
1161        goto out;
1162
1163    skb->h.raw = __skb_pull(skb, ihl);
1164    iph = skb->nh.iph;
1165    id = ntohs(iph->id);
1166    proto = iph->protocol & (MAX_INET_PROTOS - 1);
1167    segs = ERR_PTR(-EPROTONOSUPPORT);
1168
1169    rcu_read_lock();
1170    ops = rcu_dereference(inet_protos[proto]);
1171    if (likely(ops && ops->gso_segment))
1172        segs = ops->gso_segment(skb, features);
1173    rcu_read_unlock();
1174
1175    if (!segs || unlikely(IS_ERR(segs)))
1176        goto out;
1177
1178    skb = segs;
1179    do {
1180        iph = skb->nh.iph;
1181        iph->id = htons(id++);
1182        iph->tot_len = htons(skb->len - skb->mac_len);
1183        iph->check = 0;
1184        iph->check = ip_fast_csum(skb->nh.raw, iph->ihl);
1185    } while ((skb = skb->next));
1186
1187 out:
1188    return segs;
1189 }
```

1144-1150 校验待软 GSO 分段的 SKB，其 gsotype 是否存在其他非法的值。

1151-1152 待分段数据报长度至少大于 IP 首部长度，否则必定是无效 IP 数据报。

1155-1158 检测 IP 首部中长度字段是否有效。

1160-1161 再次通过 IP 首部中长度字段检测 IP 数据报长度是否有效。

1163-1167 取出 IP 首部中 ID 字段值，用于分片数据报的 IP 首部中 ID 的基础值。取出 IP 首部中的协议字段值，用于定位与之对应的传输层协议接口。

1169-1173 根据 IP 首部中协议字段值，获取对应的传输层协议接口，然后通过 gso_segment 接口（参见 tcp_tso_segment()），对 TCP 段进行软 GSO 分段，分段得到的新段通过 next 链接在原先的段之后。

1178-1185 循环对分段得到所有的段进行 IP 校验和计算。

1187-1188 最后，如果分段成功，则返回分段后 SKB 链表中的第一个 SKB，否则返回相应的错误码。

11.12.2　inet_gso_send_check()

inet_gso_send_check()是 IP 层 gso_send_check 接口的实现函数，在分段之前对伪首部进行校验和计算。

```
1101 static int inet_gso_send_check(struct sk_buff *skb)
1102 {
1103     struct iphdr *iph;
1104     struct net_protocol *ops;
1105     int proto;
1106     int ihl;
1107     int err = -EINVAL;
1108
1109     if (unlikely(!pskb_may_pull(skb, sizeof(*iph))))
1110         goto out;
1111
1112     iph = skb->nh.iph;
1113     ihl = iph->ihl * 4;
1114     if (ihl < sizeof(*iph))
1115         goto out;
1116
1117     if (unlikely(!pskb_may_pull(skb, ihl)))
1118         goto out;
1119
1120     skb->h.raw = __skb_pull(skb, ihl);
1121     iph = skb->nh.iph;
1122     proto = iph->protocol & (MAX_INET_PROTOS - 1);
1123     err = -EPROTONOSUPPORT;
1124
1125     rcu_read_lock();
1126     ops = rcu_dereference(inet_protos[proto]);
1127     if (likely(ops && ops->gso_send_check))
1128         err = ops->gso_send_check(skb);
1129     rcu_read_unlock();
1130
1131 out:
1132     return err;
1133 }
```

1109-1110 待分段数据报长度至少大于 IP 首部长度，否则必定是无效 IP 数据报。

1112-1115 检测 IP 首部中长度字段是否有效。

1117-1118 再次通过 IP 首部中长度字段来检测 IP 数据报长度是否有效。

1120-1123 取出 IP 首部中协议字段值，用于定位与之对应的传输层协议接口。

1125-1129 根据 IP 首部中协议字段值，获取对应的传输层协议接口，然后通过 gso_send_check 接口（参见 tcp_v4_gso_send_check()），重新对 TCP 段伪首部进行校验和计算。

1131-1132 返回操作结果。

第 12 章　IP 选项处理

RFC 791 和 RFC 1122 描述了 IP 选项和处理规则。本章将介绍 IP 选项的格式和处理，传输层如何指定 IP 数据报内的 IP 选项，以及网络层在转发 IP 数据报时对 IP 选项的处理。IP 选项的处理涉及以下文件：

- include/net/inet_sock.h，定义 IP 选项，TCP 连接请求块等结构。
- net/ipv4/ip_options.c，IP 选项的处理。
- net/ipv4/ip_input.c，IP 数据报的输入。

12.1　IP 选项

由于 IP 首部中可以存在选项，且可以同时存在多个选项，因此 IP 首部的长度是可变的，IPv4 允许选项最长可达 40 字节。选项的格式有单字节和多字节两种，单字节的即只包括一个字节的选项类型，而多字节的则除一个字节的类型之外，还包括选项长度以及选项数据等。

IP 选项类型的 8 位由 3 部分组成，见图 12-1。3 部分的含义见表 12-1。

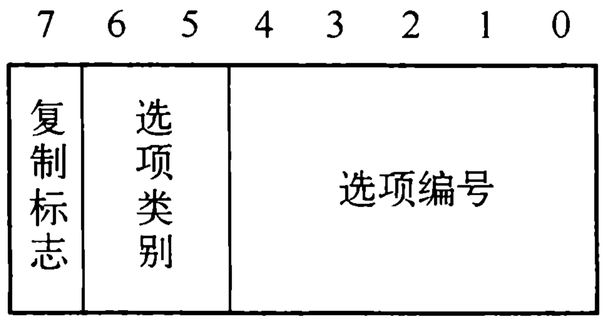

图 12-1　选项类型组成

表 12-1　选项类型

选项类型	描述
复制标志（copied）	标识该选项是否需要被复制到所有分片中。0 为只需复制到第一个分片中；1 则需复制到所有分片中
选项类别（class）	0 为控制；1 为保留；2 为调试和测量；3 为保留
选项编号（number）	是类中的再细分

常用的 IP 选项如表 12-2 所示。

表 12-2　常用的 IP 选项

复制标志	选项类别	选项编号	选项长度	描述
0	0	0	–	选项列表的结束。该选项仅占用 1 个字节　没有长度字节
0	0	1	–	没有操作。该选项仅占用 1 个字节　没有长度字节，用于对齐填充
1	0	2	变长	安全。用于携带安全、分隔、用户组和同 DOD 要求兼容的处理限制码
1	0	3	变长	宽松源路由。用来基于源提供的信息为 IP 数据报选路
1	0	9	变长	严格源路由。用来基于源提供的信息为 IP 数据报选路
0	0	7	变长	记录路径　用来跟踪一个 IP 数据报的路径
1	0	8	4	流 ID，用来携带流标识
0	2	4	变长	记录时间戳

12.1.1　选项列表的结束符

此选项指示了选项列表的结束，即是所有选项的结束，而不是某一个选项的结束，这一点需要注意，见图 12-2。

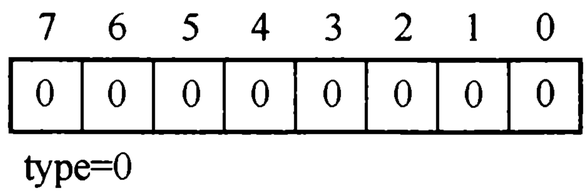

图 12-2　选项列表结束

12.1.2　空操作

此选项一般在选项间或选项尾使用，用于 32 位边界对齐操作的填充，见图 12-3。

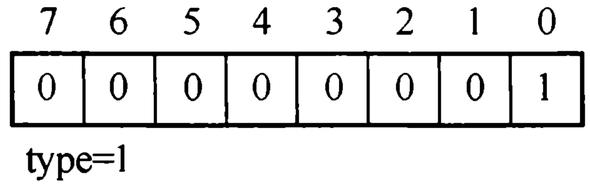

图 12-3　空操作

12.1.3　安全选项

此选项为主机提供了发送安全级别、Compartments、处理限制及传输控制码（TCC），见图 12-4。

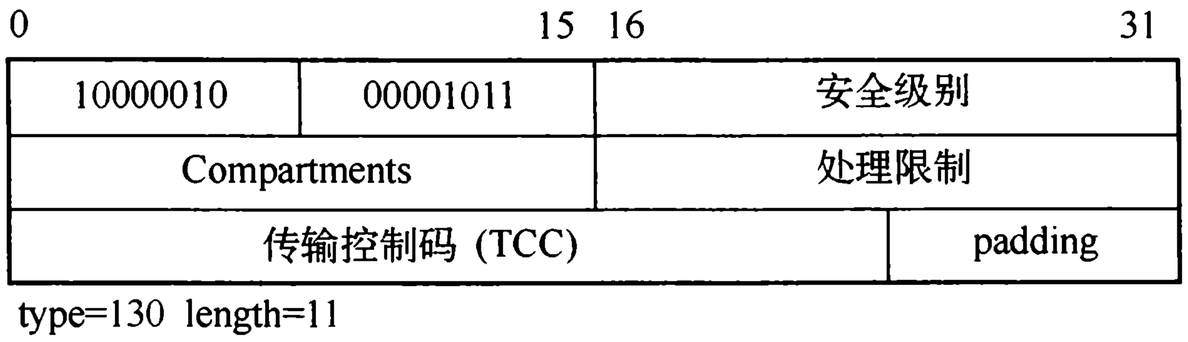

图 12-4　安全选项格式

此选项在数据报分片时必须被复制。需说明的是 padding 部分，如果该选项之后还有选项，则为无操作，否则为选项结束符。

（1）安全级别

安全级别占 16 位，安全共分 16 级，其中 8 级保留，见表 12-3。

表 12-3　安全级别

安全级别	描述
00000000 00000000	Unclassified
11110001 00110101	Confidential
01111000 10011010	EFTO
10111100 01001101	MMMM
01011110 00100110	PROG

（续）

安全级别	描述
10101111 00010011	Restricted
11010111 10001000	Secret
01101011 11000101	Top Secret
00110101 11100010	保留
10011010 11110001	保留
01001101 01111000	保留
00100100 10111101	保留
00010011 01011110	保留
10001001 10101111	保留
11000100 11010110	保留
11100010 01101011	保留

（2）Compartments

Compartments 占 16 位，全为 0 代表传送的信息是非 compartment 的。

（3）处理限制

处理限制占 16 位，控制值和版本标记是由字母和数字组成的，具体说明请参阅 Defense Intelligence Agency 网站。

（4）传输控制码

传输控制码占 24 位，它提供控制流量的方法。TCC 的值为三字母词，在 HQ DCA Code 530 中有说明。

12.1.4 严格源路由选项

严格源路由（SSRR）选项要求数据报必须严格按照发送方规定的路径经过每一个路由器。这些路由器应该是一一相连的，每两个指定的路由器之间不能有其他未指定的路由器，且路由器的顺序是不能改变的。如果数据报在传输时无法直接到达下一跳指定的路由器，路由器就会丢弃该数据报，然后产生一个源路由失败的目的不可达的 ICMP 差错报文报告给发送方，见图 12-5。

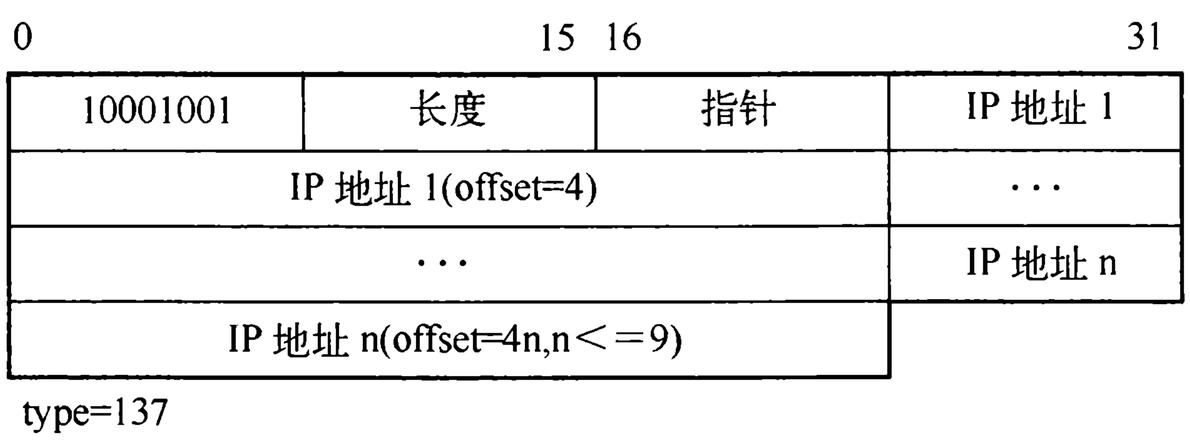

图 12-5 严格源站路由选项格式

此选项在数据报分片的时候也必须被复制（copied=1）。

- 长度，长度标识选项的长度。
- 指针，指向该设备的下下一跳 IP 地址，最小的合法值为 4。当一个路由器接收到数据报时，指针指向的是该路由器的下一跳 IP 地址，路由器转发数据报前要将指针值加 4，这

样发出去的数据报的指针又指向其下下一跳 IP 地址了。

- 路径信息，存储数据报应该经过的路由器地址。由于数据报首部长度的限制，源路由 IP 地址表中最多只能有 9 个 IP 地址项。

从图 12-6 至图 12-9 看出，数据报的传输过程中，其 IP 首部中的目标地址、选项中的指针和路径信息都是不断变化的。一开始，发送方 IP 层从上层得到的路径信息是从第一个要到达的路由器开始到最终要到达的接收方，IP 层在组织 IP 首部时，会将第一个要到达的路由器地址作为 IP 首部中的目标地址，将剩下的地址放入严格源路由选项的路径信息中，并将选项指针指向路径信息中的第一个地址。此后目标地址始终指向下一跳的地址，而选项中指针指向的始终是下下一跳地址。因此每经过一个路由器，目标地址和指针指向的地址值互换，然后指针值加 4。

可以这样理解：指针指向的那个地址之前是已经经过的路由器地址，目标地址以及指针指向的那个地址及之后的地址是从下一跳到接收方的地址，到达接收方后路径信息中就是从发送方到接收方之间的路由器地址。

发送方 IP 地址：192.168.110.11
路径信息

1492.168.110.1	192.168.120.1	192.168.130.1	192.168.140.1

图 12-6　发送方 IP 地址及发送方 IP 层从上层接收到的路径信息

目的地址：192.168.110.11
严格源路由选项

10001001	15	4	192.168.120.1	192.168.130.1	192.168.140.1

图 12-7　发送方生成的 IP 首部

目的地址：192.168.120.1
严格源路由选项

10001001	15	8	192.168.110.1	192.168.130.1	192.168.140.1

图 12-8　到达 192.168.120.1 时的 IP 首部

目的地址：192.168.130.1
严格源路由选项

10001001	15	12	192.168.110.1	192.168.120.1	192.168.140.1

图 12-9　离开 192.168.120.1 时的 IP 首部

12.1.5　宽松源路由选项

宽松源路由（LSRR）选项与严格源路由选项类似，见图 12-10。不同的是，宽松源路由在选项的 IP 地址表中并不列出一条完备而严格的路径，而是只给出路径中的某些关键点。在关键点之间可以通过路由器的自动路由选择功能进行路由。此选项在数据报分片的时候也必须被复制。

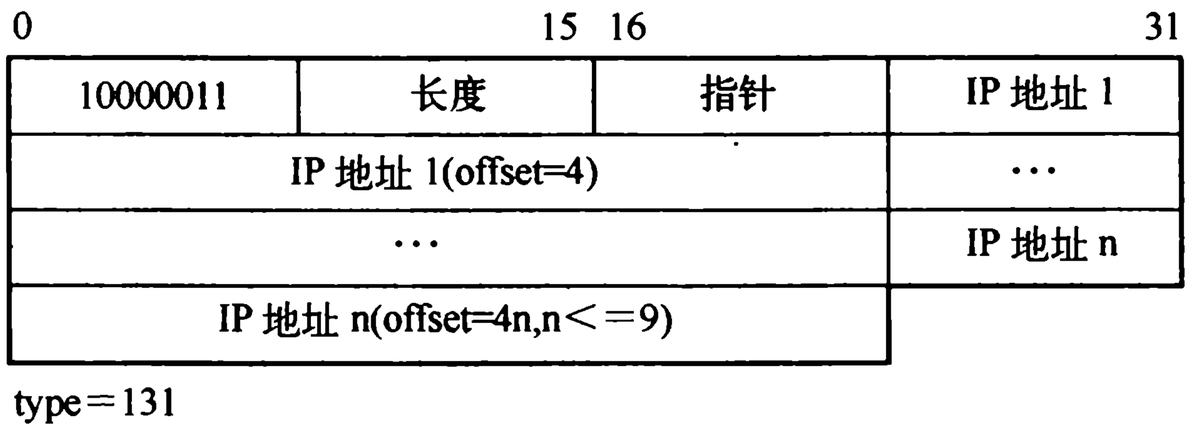

图 12-10　宽松源路由选项格式

12.1.6　记录路由选项

记录路由(RR)选项用于记录 IP 数据报从发送方到接收方所经过的路径上各个路由器的 IP 地址，见图 12-11。

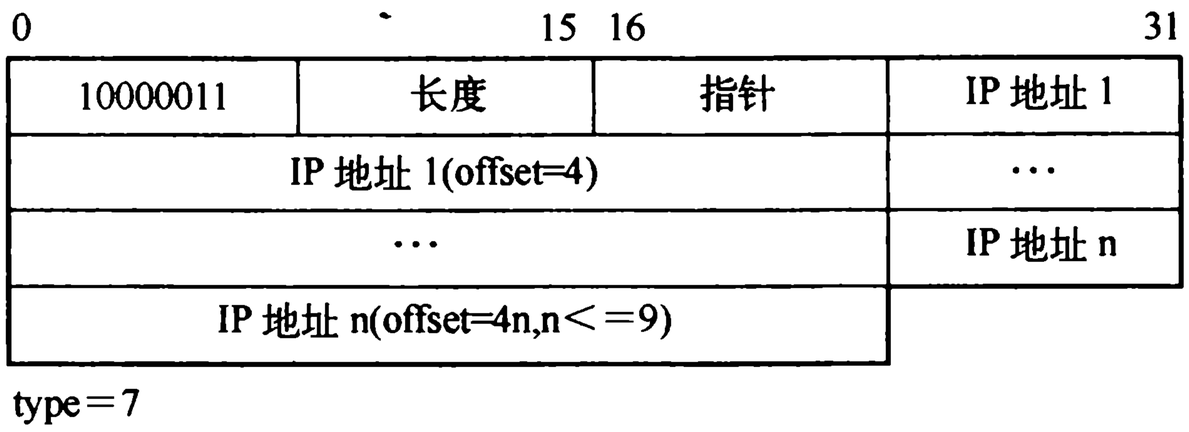

图 12-11　记录路径选项格式

此选项在数据报分片时不需要复制，仅保留在第一个分片中。

- 长度，标识选项的长度，由发送方主机对地址数作预先估计。
- 指针，指向地址区域中下一个可存放地址的位置，最小合法值是 4。在记录路径时，将指针与选项长度进行比较，若小于选项长度，就将该路由器的地址填入到指针所指的地址区域，然后将指针往后移动 4B，指向下一个可用的地址块。若预计的地址表空间不足以记录全部路径，则不再记录余下的 IP 地址。
- 路径区域，用于记录传输过程中经过的路由器的 IP 地址，由发送方主机对地址数的估计预先分配，其长度不会因为添加地址值而发生改变，该区域的初始值必须为 0。一旦路径区域中路径数据已满（即指针超过了长度），数据报还会继续被转发，但不再记录新的地址值。而如果还有空间，却不足以容纳要插入的整个地址，则该数据报被认为是错误的，将被丢弃。在上述两种情况下，路由器都会发送一个参数问题 ICMP 报文到发送方。

12.1.7　流标识选项

流标识(SID)选项提供了一种在不支持流概念的网络上携带 16 位 SATNET 流标识的方式，见图 12-12。此选项必须在数据报分片时被复制。

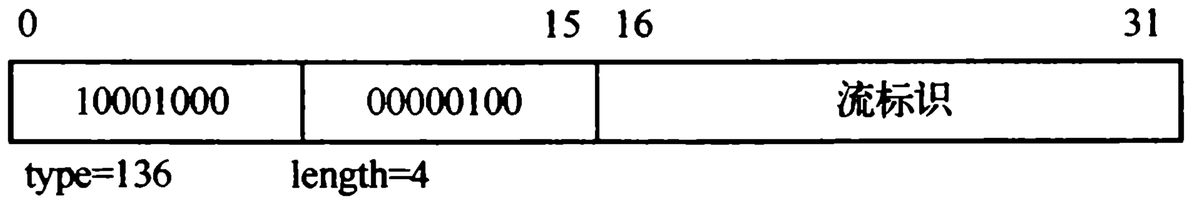

图 12-12　流标识选项格式

12.1.8　时间戳选项

时间戳选项用于记录 IP 数据报经过路由器时的当地时间，根据时间戳可以估算 IP 数据报从一个路由器到另一个路由器所花费的时间，从而帮助分析网络的吞吐率和负载情况，见图 12-13。

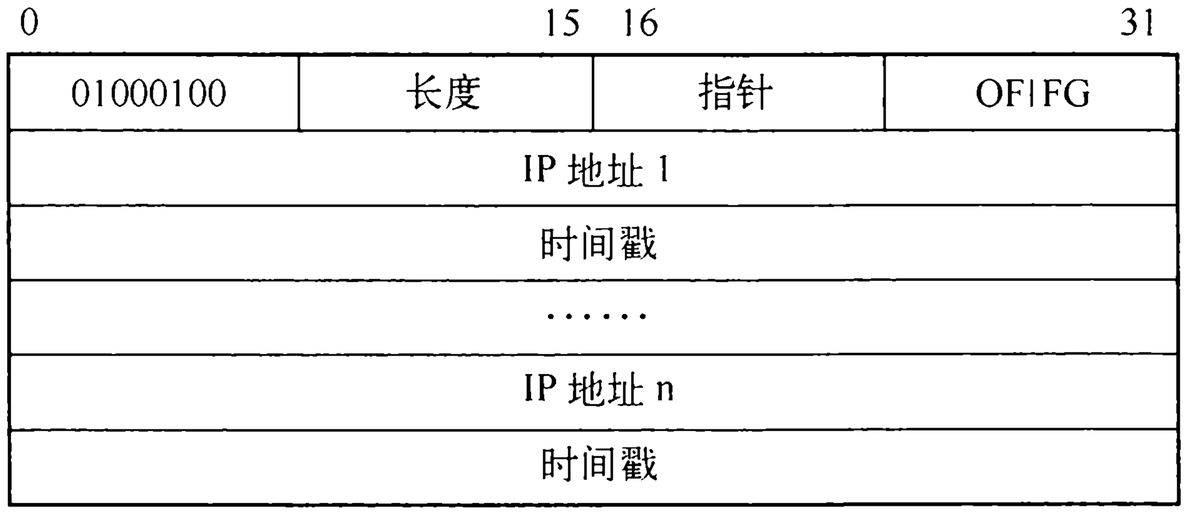

图 12-13　时间戳选项格式

此选项在数据报分片时不需要复制，仅保留在第一个分片中。

（1）指针，指向地址区域中下一个可存放时间戳的位置，最小合法值是 5。当指针大于长度的时候，表示时间戳区域已满。

（2）OF，溢出字段，用于记录因预留空间不够而未能记录下来的时间戳的个数。若溢出计算本身溢出，也就是未能记录下的时间戳个数超过 15 个，路由器将丢弃该数据报，并产生一个参数问题 ICMP 报文到发送方。

（3）FG，标志字段用于定时时间戳选项的格式，见表 12-4。

表 12-4　FG 取值

FG	描述
0	只记录时间戳，以连续的 32 位字存储
1	记录 IP 地址和时间戳
3	发送方对选项列表进行初始化，存放了 4 个 IP 地址和 4 个取值为 0 的时间戳值。只有当列表中的下一个 IP 地址与当前路由器地址匹配时，才记录它的时间戳

（4）时间戳区域，发送方应为选项分配一个足够大的时间戳数据区域，以容纳预期的所有时间戳。时间戳数据区域的初始内容必须设置为 0。如果还有空间却不足以插入一个完整的时间戳，数据报会被丢弃并返回一个参数问题 ICMP 报文到发送方。

选项中的每个时间戳均为 32 位，标准时间戳采用世界时间表示，从午夜起计时，单位为毫秒。凡不符合此标准的都必须将时间戳的最高位置设为 1，表示非标准值。

12.1.9　路由器警告选项

路由器警告选项是提醒路由器需要更仔细地检查这个包，对内容需要做特殊处理，见图 12-14。

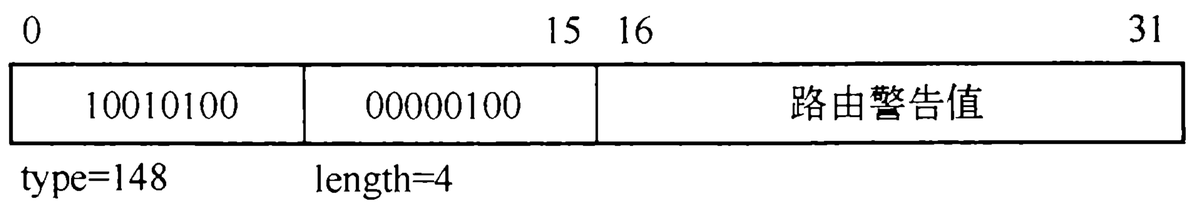

图 12-14　路由器警告选项格式

路由器警告选项主要用于 RSVP（资源预定）协议，参见 RFC 2113。路由警告选项的值意义见表 12-5。

表 12-5 路由警告选项的值

路由警告选项的值	描述
0	路由器将检查报文
1—65535	保留

12.2 ip_options 结构

IP 选项信息块，在 IP 选项处理中，IP 选项信息块 ip_options 是最常用的结构，用来描述相关的 IP 选项。该结构可以用在 SKB 中，描述所在 SKB 的数据报中存在的选项，参见 IP 层中信息控制块 inet_skb_parm 结构；也可以单独使用，如通过 IP_OPTIONS 选项设置和获取所在套接口发送数据报 IP 首部中的 IP 选项。

```
38 struct ip_options {
39     __be32       faddr;
40     unsigned char    optlen;
41     unsigned char    srr;
42     unsigned char    rr;
43     unsigned char    ts;
44     unsigned char    is_data:1,
45              is_strictroute:1,
46              srr_is_hit:1,
47              is_changed:1,
48              rr_needaddr:1,
49              ts_needtime:1,
50              ts_needaddr:1;
51     unsigned char    router_alert;
52     unsigned char    cipso;
53     unsigned char    __pad2;
54     unsigned char    __data[0];
55 };
```

39 __be32 faddr

存在宽松源路由或严格源路由选项时，用来记录下一跳的 IP 地址。

40 unsigned char optlen

标识 IP 首部中选项所占的字节数。

41 unsigned char srr

记录宽松源路由或严格源路由选项在 IP 首部中的偏移量，即选项的第一个字节的地址减去 IP 首部的第一个字节的地址。

42 unsigned char rr

用于记录记录路径选项在 IP 首部中的偏移量。

43 unsigned char ts

用于记录时间戳选项在 IP 首部中的偏移量。

44 unsigned char is_data:1

标识该IP选项是否有数据，若有则存放在__data字段起始的存储空间内，即紧跟在ip_option结构后面。这里的数据不只是选项数据，而是整个选项内容。

45 is_strictroute:1

标识该选项是 IPOPT_SSRR，而不是 IPOPT_LSRR。

46 srr_is_hit:1

表示目的地址是从源路由选项选出的。

47 is_changed:1

标识是否修改过 IP 首部，如果是则需要重新计算 IP 首部校验和。

48 rr_needaddr:1

标识有 IPOPT_RR 选项，需要记录 IP 地址。

49 ts_needtime:1

50 ts_needaddr:1

ts_needtime 标识有 IPOPT_TIMESTAMP 选项，需要记录时间戳。

ts_needaddr 标识有 IPOPT_TIMESTAMP 选项，需要记录 IP 地址。

51 unsigned char router_alert

标识 IPOPT_RA 选项。路由器警告选项，表示路由器应该更仔细地检查这个数据报，参见 RFC 2113。

52 unsigned char cipso

用于记录商业 IP 安全选项在 IP 首部中的偏移量。

53 unsigned char __pad2

未使用。

54 unsigned char __data[0]

若选项有数据则从该字段开始，使之紧跟在 ip_option 结构后面，最多不能超过 40B。

12.3　在 IP 数据报中构建 IP 选项

ip_options_build()用来根据 IP 选项信息块构建数据报中 IP 首部的 IP 选项。如果有严格源路由选项，则将目标地址写入该选项的末尾；对于记录路由和时间戳这两个选项，如果不是分片，则在选项中填写相应的值；如果是分片则将它们替换为空操作。参数说明如下：

- skb，待构建的 IP 选项的 IP 数据报。
- opt，用来在 IP 数据报中构建 IP 选项的 IP 选项信息块结构。
- daddr，待构建 IP 选项 IP 数据报的目的地址，此目的地址是真正的目的地址，即接收方，而不是 IP 首部中作为目的地址的下一跳地址。
- rt，待构建 IP 选项 IP 数据报的输出路由缓存。
- is_frag，标识待构建 IP 选项的 IP 数据报是不是分片。

```
40  void ip_options_build(struct sk_buff * skb, struct ip_options * opt,
41                __be32 daddr, struct rtable *rt, int is_frag)
42  {
43     unsigned char * iph = skb->nh.raw;
44
45     memcpy(&(IPCB(skb)->opt), opt, sizeof(struct ip_options));
```

```
46      memcpy(iph+sizeof(struct iphdr), opt->__data, opt->optlen);
47      opt = &(IPCB(skb)->opt);
48      opt->is_data = 0;
49
50      if (opt->srr)
51         memcpy(iph+opt->srr+iph[opt->srr+1]-4, &daddr, 4);
52
53      if (!is_frag) {
54         if (opt->rr_needaddr)
55            ip_rt_get_source(iph+opt->rr+iph[opt->rr+2]-5, rt);
56         if (opt->ts_needaddr)
57            ip_rt_get_source(iph+opt->ts+iph[opt->ts+2]-9, rt);
58         if (opt->ts_needtime) {
59            struct timeval tv;
60            __be32 midtime;
61            do_gettimeofday(&tv);
62            midtime = htonl((tv.tv_sec % 86400) * 1000 + tv.tv_usec / 1000);
63            memcpy(iph+opt->ts+iph[opt->ts+2]-5, &midtime, 4);
64         }
65         return;
66      }
67      if (opt->rr) {
68         memset(iph+opt->rr, IPOPT_NOP, iph[opt->rr+1]);
69         opt->rr = 0;
70         opt->rr_needaddr = 0;
71      }
72      if (opt->ts) {
73         memset(iph+opt->ts, IPOPT_NOP, iph[opt->ts+1]);
74         opt->ts = 0;
75         opt->ts_needaddr = opt->ts_needtime = 0;
76      }
77   }
```

43 取得在 SKB 中 IP 首部存储的地址。

45-48 将源 IP 选项信息块及其后面紧跟的选项数据复制到 SKB 对应的存储区域中，并将 opt 指向 SKB 中的 ip_options 结构。注意：这里该结构的 is_data 字段是设置为 0 的，也就是说此结构后不跟选项内容，SKB 中选项信息和选项内容是分别存放的。

50-51 如果有严格源路由选项，则将目的地址复制到源路由选项地址列表的末尾，iph[opt->srr+1]是取得该选项的长度。

53-66 如果该数据报不是 IP 分片，且存在记录路由/时间戳选项，则通过输出路由缓存获取源地址填写到记录路由选项/时间戳选项的地址部分中，获取当前的时间填写到时间戳选项中。

67-76 如果该数据报不是 IP 分片，但存在记录路由/时间戳选项，则将这些选项替换成无操作。

12.4 复制 IP 数据报中选项到指定的 ip_options 结构

ip_options_echo()从接收到的 IP 数据报中复制选项内容及其信息到指定的 ip_options 结构中。通常是为了利用这些选项创建一个回应报文，或是在用户层获取接收到的 IP 数据报中的 IP 选项，例如回应一个 ICMP 报文的请求；或者接收到错误报文后给发送方发送 ICMP 差错报文；或者在 TCP 中给发送方发送 ACK 段等。参数说明如下：

- dopt，复制选项的目标选项信息块结构。
- skb，复制选项的源 IP 数据报。

```
88  int ip_options_echo(struct ip_options * dopt, struct sk_buff * skb)
89  {
90      struct ip_options *sopt;
91      unsigned char *sptr, *dptr;
92      int soffset, doffset;
93      int   optlen;
94      __be32   daddr;
95
96      memset(dopt, 0, sizeof(struct ip_options));
97
98      dopt->is_data = 1;
99
100     sopt = &(IPCB(skb)->opt);
101
102     if (sopt->optlen == 0) {
103        dopt->optlen = 0;
104        return 0;
105     }
106
107     sptr = skb->nh.raw;
108     dptr = dopt->__data;
109
110     if (skb->dst)
111        daddr = ((struct rtable*)skb->dst)->rt_spec_dst;
112     else
113        daddr = skb->nh.iph->daddr;
```

96-98 初始化目标选项信息块 dopt，各字段都清零，又由于需将选项复制到 dopt 之后，设置其 is_data 标志为 1。

100 获得数据报中选项信息块的地址。

102-105 如果数据报中没有 IP 选项，则将目标选项信息块的选项长度设置为 0 后直接返回。

107-108 获取数据报中 IP 首部的起始地址，以及复制选项数据的目标起始地址。

110-113 获取该数据报的目的地址，用于记录路由选项。当存在路由项时，为路由项的 rt_spec_dst 字段（该字段称为首选源地址），否则是数据报 IP 首部中的目的地址。此处的所谓目标地址其实就是当前设备的本地地址。

选项如下：

（1）复制记录路由选项

```
115     if (sopt->rr) {
116        optlen  = sptr[sopt->rr+1];
117        soffset = sptr[sopt->rr+2];
118        dopt->rr = dopt->optlen + sizeof(struct iphdr);
119        memcpy(dptr, sptr+sopt->rr, optlen);
120        if (sopt->rr_needaddr && soffset <= optlen) {
121           if (soffset + 3 > optlen)
122              return -EINVAL;
123           dptr[2] = soffset + 4;
124           dopt->rr_needaddr = 1;
125        }
```

```
126         dptr += optlen;
127         dopt->optlen += optlen;
128     }
```

116-119 将数据报中的记录路径选项内容复制到目标选项信息块 dopt 的__data 字段开始区域中。

120-122 如果原数据报中需要记录路由，而剩余空间却不足以放下一个地址值，则出错返回。

123 更新选项中的指针值。

124 更新标志，标识有 IPOPT_RR 选项，需要记录 IP 地址。

126 将指针移动到下一个选项处。

127 更新选项长度值。

（2）复制时间戳选项

```
129     if (sopt->ts) {
130         optlen = sptr[sopt->ts+1];
131         soffset = sptr[sopt->ts+2];
132         dopt->ts = dopt->optlen + sizeof(struct iphdr);
133         memcpy(dptr, sptr+sopt->ts, optlen);
134         if (soffset <= optlen) {
135             if (sopt->ts_needaddr) {
136                 if (soffset + 3 > optlen)
137                     return -EINVAL;
138                 dopt->ts_needaddr = 1;
139                 soffset += 4
140             }
141             if (sopt->ts_needtime) {
142                 if (soffset + 3 > optlen)
143                     return -EINVAL;
144                 if ((dptr[3]&0xF) != IPOPT_TS_PRESPEC) {
145                     dopt->ts_needtime = 1;
146                     soffset += 4;
147                 } else {
148                     dopt->ts_needtime = 0;
149
150                     if (soffset + 8 <= optlen) {
151                         __be32 addr;
152
153                         memcpy(&addr, sptr+soffset-1, 4);
154                         if (inet_addr_type(addr) != RTN_LOCAL) {
155                             dopt->ts_needtime = 1;
156                             soffset += 8;
157                         }
158                     }
159                 }
160             }
161             dptr[2] = soffset;
162         }
163         dptr += optlen;
164         dopt->optlen += optlen;
165     }
```

130-133 将数据报 IP 首部中的时间戳选项内容复制到目标选项信息块 dopt 后数据区域中。

134-140 检测原数据报在需要记录地址的情况下，剩余空间是否够容纳一个 IP 地址值，

如果够则设置 dopt 的 ts_needaddr 标志。

141-160 在需要记录时间戳情况下，当选项标志为 0 或 1 时，或当选项标志为 3 而指针指向的地址类型不是 RTN_LOCAL 时，即本地不是数据报的接收方，才设置 ts_needtime 为 1。

161 更新时间戳选项的指针值。

163-164 最后，将指针移动到下一个选项处并更新选项长度。

163-164 将指针移动到下一个选项处并更新选项长度值。

（3）复制源路由选项

```
166     if (sopt->srr) {
167        unsigned char * start = sptr+sopt->srr;
168        __be32 faddr;
169
170        optlen  = start[1];
171        soffset = start[2];
172        doffset = 0;
173        if (soffset > optlen)
174           soffset = optlen + 1;
175        soffset -= 4;
176        if (soffset > 3) {
177           memcpy(&faddr, &start[soffset-1], 4);
178           for (soffset-=4, doffset=4; soffset > 3; soffset-=4, doffset+=4)
179              memcpy(&dptr[doffset-1], &start[soffset-1], 4);
180           /*
181            * RFC1812 requires to fix illegal source routes.
182            */
183           if (memcmp(&skb->nh.iph->saddr, &start[soffset+3], 4) == 0)
184              doffset -= 4;
185        }
186        if (doffset > 3) {
187           memcpy(&start[doffset-1], &daddr, 4);
188           dopt->faddr = faddr;
189           dptr[0] = start[0];
190           dptr[1] = doffset+3;
191           dptr[2] = 4;
192           dptr += doffset+3;
193           dopt->srr = dopt->optlen + sizeof(struct iphdr);
194           dopt->optlen += doffset+3;
195           dopt->is_strictroute = sopt->is_strictroute;
196        }
197     }
```

167-185 从 IP 首部中复制源路由选项到 IP 选项信息块的存储区域中。

186-196 如果成功复制源路由选项，则更新目标 ip_options 结构和目标选项区域中的一些字段值。

（4）复制商业 IP 安全选项

```
198     if (sopt->cipso) {
199        optlen  = sptr[sopt->cipso+1];
200        dopt->cipso = dopt->optlen+sizeof(struct iphdr);
201        memcpy(dptr, sptr+sopt->cipso, optlen);
202        dptr += optlen;
203        dopt->optlen += optlen;
```

```
204     }
```

198-204 从 IP 首部中复制商业 IP 安全选项到 IP 选项信息块中。

```
205     while (dopt->optlen & 3) {
206         *dptr++ = IPOPT_END;
207         dopt->optlen++;
208     }
209     return 0;
210 }
```

205-209 最后，如果选项长度没有对齐，则填充结束符号将其对齐。

12.5 处理待发送 IP 分片中的选项

ip_options_fragment()用来清理掉复制标志为 0 的选项，将它们填充为无操作，因为这些选项对一个完整的 IP 数据报只需处理一次，而无需对每个分片都处理。这样的选项其实也就包括了时间戳选项和记录路由选项，通过这种方法使协议首部的长度保持不变，只需重新计算校验和。

```
218 void ip_options_fragment(struct sk_buff * skb)
219 {
220     unsigned char * optptr = skb->nh.raw + sizeof(struct iphdr);
221     struct ip_options * opt = &(IPCB(skb)->opt);
222     int  l = opt->optlen;
223     int  optlen;
224
225     while (l > 0) {
226         switch (*optptr) {
227         case IPOPT_END:
228             return;
229         case IPOPT_NOOP:
230             l--;
231             optptr++;
232             continue;
233         }
234         optlen = optptr[1];
235         if (optlen<2 || optlen>l)
236           return;
237         if (!IPOPT_COPIED(*optptr))
238             memset(optptr, IPOPT_NOOP, optlen);
239         l -= optlen;
240         optptr += optlen;
241     }
242     opt->ts = 0;
243     opt->rr = 0;
244     opt->rr_needaddr = 0;
245     opt->ts_needaddr = 0;
246     opt->ts_needtime = 0;
247     return;
248 }
```

225-241 遍历所有的选项，直到遇到选项列表结束符返回。忽略无操作，对于那些无需复制到分片中的选项，则将其值修改为无操作。

242-246 修改 SKB 中对应的选项信息标志。

12.6　解析 IP 选项

ip_options_compile()会被两个函数调用：ip_options_get_finish()和 ip_rcv_options()，分别对应了发送和接收两个方向。注意：在发送时的调用方式是 ip_option_compile(opt, NULL)，而在接收时其调用语句是 ip_option_compile(NULL, skb)，这是因为发送和接收时，待解析的 IP 选项以及解析后的 IP 选项信息块所存储的位置是不同的——发送时 IP 选项存储在参数 opt 的__data 字段起始的区域中，解析得到的信息会保存在 opt 中；接收时 IP 选项存储在参数 skb 的 nh.raw 指向的 IP 首部中，解析得到的信息则保存在 SKB 的 cb 中。因此，opt 和 skb 两者不能同时为 NULL，且当 opt 的 is_data 为 0 时，skb 也不能为 NULL。

```
256 int ip_options_compile(struct ip_options * opt, struct sk_buff * skb)
257 {
258     int l;
259     unsigned char * iph;
260     unsigned char * optptr;
261     int optlen;
262     unsigned char * pp_ptr = NULL;
263     struct rtable *rt = skb ? (struct rtable*)skb->dst : NULL;
264
265     if (!opt) {
266         opt = &(IPCB(skb)->opt);
267         iph = skb->nh.raw;
268         opt->optlen = ((struct iphdr *)iph)->ihl*4 - sizeof(struct iphdr);
269         optptr = iph + sizeof(struct iphdr);
270         opt->is_data = 0;
271     } else {
272         optptr = opt->is_data ? opt->__data : (unsigned char*)&(skb->nh.iph[1]);
273         iph = optptr - sizeof(struct iphdr);
274     }
```

获得待解析的 IP 选项地址及解析后 ip_options 结构的地址。当 opt 为 NULL 时，解析 skb->nh.raw 指向的 IP 首部中的 IP 选项到 skb 的信息控制块；而当 opt 不为 NULL 且 opt->is_data 不为 0 时，则解析紧跟在 opt 之后的 IP 选项到 opt；opt 和 opt->is_data 同时为 0，则解析存储于 skb->nh.iph 内的 IP 首部中的 IP 选项到 opt 中。

opt 是最终要放入选项信息的 ip_options 结构。

iph 指向选项所在的 IP 首部。

optptr 指向 IP 首部中的选项。

```
276     for (l = opt->optlen; l > 0; ) {
277         switch (*optptr) {
278             case IPOPT_END:
279             for (optptr++, l--; l>0; optptr++, l--) {
280                 if (*optptr != IPOPT_END) {
281                     *optptr = IPOPT_END;
282                     opt->is_changed = 1;
283                 }
284             }
```

```
285             goto eol;
286               case IPOPT_NOOP:
287             l--;
288             optptr++;
289             continue;
290         }
291         optlen = optptr[1];
292         if (optlen<2 || optlen>l) {
293             pp_ptr = optptr;
294             goto error;
295          }
```

276 循环解析各个选项内容。

278-285 首先如果检测到选项列表结束符，则将后面所剩余的全部空间都设置为结束符，因为修改了选项内容，从而需重新计算校验和，因此设置 opt->is_changed 为 1，然后结束解析返回。

286-289 如果检测到空操作符，则修改循环变量 l，并将指针移动到下一个选项处后，直接跳入下一次循环。

291-295 校验当前待处理选项的长度值是否有效，若无效，则结束解析，转至出错处理。在此之所以判断 optlen<2，是因为除了上面已经处理过的选项列表结束符和无操作符外，其他选项长度都大于或等于 2，即所谓“多字节选项”。

处理源路由选项如下：

```
296         switch (*optptr) {
297             case IPOPT_SSRR:
298               case IPOPT_LSRR:
299             if (optlen < 3) {
300                 pp_ptr = optptr + 1;
301                 goto error;
302             }
303             if (optptr[2] < 4) {
304                 pp_ptr = optptr + 2;
305                 goto error;
306             }
307             /* NB: cf RFC-1812 5.2.4.1 */
308             if (opt->srr) {
309                 pp_ptr = optptr;
310                 goto error;
311             }
312             if (!skb) {
313                 if (optptr[2] != 4 || optlen < 7 || ((optlen-3) & 3)) {
314                     pp_ptr = optptr + 1;
315                     goto error;
316                 }
317                 memcpy(&opt->faddr, &optptr[3], 4);
318                 if (optlen > 7)
319                     memmove(&optptr[3], &optptr[7], optlen-7);
320             }
321             opt->is_strictroute = (optptr[0] == IPOPT_SSRR);
322             opt->srr = optptr - iph;
323              break;
```

297-302 校验待处理源路由选项的长度值是否有效。

303-306 校验待处理源路由选项的指针值是否有效。

308-311 IP 选项信息块 opt 中源路由选项若已处理过，则无需再处理。

312-320 显然这是针对发送的，先再次校验选项中的指针及长度的有效性：对于选项指针值其最小值为 4，对于选项长度值，除了选项类型、选项长度以及选项指针的三字节外，至少应该可以容纳一个 IP 地址，且扣除了前面的三字节应 4 字节对齐。作为发送方，应取出第一个地址作为下一跳地址，并在路径列表中多于一个地址时，将剩余的所有地址往前移动一个位置。

321-322 根据选项类型标识是不是严格源路由选项，并记录源路由选项在 IP 首部中的偏移量。

处理记录路由选项如下：

```
324             case IPOPT_RR:
325           if (opt->rr) {
326              pp_ptr = optptr;
327              goto error;
328           }
329           if (optlen < 3) {
330              pp_ptr = optptr + 1;
331              goto error;
332           }
333           if (optptr[2] < 4) {
334              pp_ptr = optptr + 2;
335              goto error;
336           }
337           if (optptr[2] <= optlen) {
338              if (optptr[2]+3 > optlen) {
339                 pp_ptr = optptr + 2;
340                 goto error;
341              }
342              if (skb) {
343                 memcpy(&optptr[optptr[2]-1], &rt->rt_spec_dst, 4);
344                 opt->is_changed = 1;
345              }
346              optptr[2] += 4;
347              opt->rr_needaddr = 1;
348           }
349           opt->rr = optptr - iph;
350            break;
```

325-328 若 IP 选项信息块 opt 中记录路由选项已处理过，则无需再次处理。

329-332 校验待处理记录路由选项的长度值是否有效。

333-336 校验待处理记录路由选项的指针值是否有效。

337-348 存储 IP 地址的数据区有效的情况下，如果是接收，则将本地源地址填入到记录路由选项中，并设置需重新计算校验和标识，最后调整选项指针。

349 标记记录路由选项在 IP 首部中的偏移量。

处理时间戳选项如下：

```
351             case IPOPT_TIMESTAMP:
352           if (opt->ts) {
353              pp_ptr = optptr;
```

```
354                goto error;
355             }
356             if (optlen < 4) {
357                pp_ptr = optptr + 1;
358                goto error;
359             }
360             if (optptr[2] < 5) {
361                pp_ptr = optptr + 2;
362                goto error;
363              }
```

352-355 若 IP 选项信息块 opt 中时间戳选项已处理过，则无需再次处理。

356-359 校验待处理时间戳选项的长度值是否有效。

360-363 校验待处理时间戳选项的指针值是否有效。

```
364             if (optptr[2] <= optlen) {
365                __be32 *timeptr = NULL;
366                if (optptr[2]+3 > optptr[1]) {
367                   pp_ptr = optptr + 2;
368                   goto error;
369                }
370                switch (optptr[3]&0xF) {
371                     case IPOPT_TS_TSONLY:
372                   opt->ts = optptr - iph;
373                   if (skb)
374                      timeptr = (__be32*)&optptr[optptr[2]-1];
375                   opt->ts_needtime = 1;
376                   optptr[2] += 4;
377                   break;
378                    case IPOPT_TS_TSANDADDR:
379                   if (optptr[2]+7 > optptr[1]) {
380                      pp_ptr = optptr + 2;
381                      goto error;
382                   }
383                   opt->ts = optptr - iph;
384                   if (skb) {
385                      memcpy(&optptr[optptr[2]-1], &rt->rt_spec_dst, 4);
386                      timeptr = (__be32*)&optptr[optptr[2]+3];
387                   }
388                   opt->ts_needaddr = 1;
389                   opt->ts_needtime = 1;
390                   optptr[2] += 8;
391                   break;
392                    case IPOPT_TS_PRESPEC:
393                   if (optptr[2]+7 > optptr[1]) {
394                      pp_ptr = optptr + 2;
395                      goto error;
396                   }
397                   opt->ts = optptr - iph;
398                   {
399                      __be32 addr;
400                      memcpy(&addr, &optptr[optptr[2]-1], 4);
401                      if (inet_addr_type(addr) == RTN_UNICAST)
402                         break;
403                      if (skb)
```

```
404                     timeptr = (__be32*)&optptr[optptr[2]+3];
405                 }
406                 opt->ts_needtime = 1;
407                 optptr[2] += 8;
408                 break;
409                  default:
410                 if (!skb && !capable(CAP_NET_RAW)) {
411                    pp_ptr = optptr + 3;
412                    goto error;
413                 }
414                 break;
415             }
416             if (timeptr) {
417                struct timeval tv;
418                __be32  midtime;
419                do_gettimeofday(&tv);
420                midtime = htonl((tv.tv_sec % 86400) * 1000 + tv.tv_usec / 1000);
421                memcpy(timeptr, &midtime, sizeof(__be32));
422                opt->is_changed = 1;
423             }
424          } else {
425             unsigned overflow = optptr[3]>>4;
426             if (overflow == 15) {
427                pp_ptr = optptr + 3;
428                goto error;
429             }
430             opt->ts = optptr - iph;
431             if (skb) {
432                optptr[3] = (optptr[3]&0xF)|((overflow+1)<<4);
433                opt->is_changed = 1;
434             }
435          }
436           break;
```

364-369 在时间戳区域还未被填满时，首先检测时间戳选项指针是否有效。

370 针对时间戳选项中标志 FG 的不同取值分别作处理。

371-377 如果只记录时间戳，则标记时间戳选项在 IP 首部中的偏移量，同时若是转发数据报，则取得时间戳的记录位置，后续将会把时间戳复制到该位置。

378-391 如果需同时记录 IP 地址和时间戳，则首先检测剩余空间是否够记录一个 IP 地址和一个时间戳，如果够则将本机 IP 地址复制到选项中，同时取得时间戳的记录位置，后续会将时间戳复制到该位置。

392-408 根据发送方在列表中指定的 IP 地址记录时间戳，首先检测剩余空间是否够记录一个 IP 地址和一个时间戳，如果够则还需确保选项中指定的地址不是广播地址，然后取得时间戳的记录位置。

416-423 如果之前取得了时间戳的记录位置，则取得时间值并复制该值的记录位置，记得设置选项信息块 opt 的 is_changed 字段。

425-434 若时间戳区域已满，此时如果 OF 标志溢出，则跳转到出错处理，否则若是接收，则重新计算 OF 标志。

处理路由器警告选项如下：

```
437                case IPOPT_RA:
438              if (optlen < 4) {
439                 pp_ptr = optptr + 1;
440                 goto error;
441              }
442              if (optptr[2] == 0 && optptr[3] == 0)
443                 opt->router_alert = optptr - iph;
444               break;
```

438-441 校验待处理的路由器警告选项其长度值是否有效。

442-443 如果路由器警告选项值有效，则标记路由器警告选项在 IP 首部中的偏移量。

处理商业 IP 安全选项如下：

```
445                case IPOPT_CIPSO:
446              if ((!skb && !capable(CAP_NET_RAW)) || opt->cipso) {
447                 pp_ptr = optptr;
448                 goto error;
449              }
450              opt->cipso = optptr - iph;
451                 if (cipso_v4_validate(&optptr)) {
452                 pp_ptr = optptr;
453                 goto error;
454              }
455               break;
```

446-449 如果是接收数据报，则操作该选项的进程必须具有操作 RAW 套接口和 PACKET 套接口的能力。

450 标记商业 IP 安全选项在 IP 首部中的偏移量。

451-454 检测商业 IP 安全选项是否有效。

其他选项如下：

```
456                case IPOPT_SEC:
457               case IPOPT_SID:
458               default:
459              if (!skb && !capable(CAP_NET_RAW)) {
460                 pp_ptr = optptr;
461                 goto error;
462              }
463              break;
464        }
465        l -= optlen;
466        optptr += optlen;
467     }
```

456-464 对于流标识选项与安全选项，无需作特别的处理，但如果是接收数据报，则操作选项的进程必须具有操作 RAW 套接口和 PACKET 套接口的能力。

465-466 每处理完一个选项后，将指针往后移动到下一个选项，以便继续处理后面的选项。

```
469 eol:
470     if (!pp_ptr)
471        return 0;
```

```
472
473 error:
474     if (skb) {
475         icmp_send(skb, ICMP_PARAMETERPROB, 0, htonl((pp_ptr-iph)<<24));
476     }
477     return -EINVAL;
478 }
```

469-471 在处理选项过程中，若遇到选项列表结束符，则跳转到这里处理。

473-477 在处理选项过程中，无论何时遇到错误都会跳转到此处处理，即如果是接收数据报，则需给该 IP 数据报的发送方发送一个参数问题 ICMP 差错报文。

12.7　还原在校验 IP 选项时修改的 IP 选项

ip_options_undo()用来还原通过 ip_options_compile()处理过的 IP 选项信息块。如果数据报中存在源路由选项、记录路由选项或时间戳选项，将它们恢复到 ip_options_compile()处理之前。此操作通常是在应用层为获取 IP 选项而复制 IP 选项到用户程序之前被调用。

```
485 void ip_options_undo(struct ip_options * opt)
486 {
487     if (opt->srr) {
488         unsigned  char * optptr = opt->__data+opt->srr-sizeof(struct  iphdr);
489         memmove(optptr+7, optptr+3, optptr[1]-7);
490         memcpy(optptr+3, &opt->faddr, 4);
491     }
492     if (opt->rr_needaddr) {
493         unsigned  char * optptr = opt->__data+opt->rr-sizeof(struct  iphdr);
494         optptr[2] -= 4;
495         memset(&optptr[optptr[2]-1], 0, 4);
496     }
497     if (opt->ts) {
498         unsigned  char * optptr = opt->__data+opt->ts-sizeof(struct  iphdr);
499         if (opt->ts_needtime) {
500             optptr[2] -= 4;
501             memset(&optptr[optptr[2]-1], 0, 4);
502             if ((optptr[3]&0xF) == IPOPT_TS_PRESPEC)
503                 optptr[2] -= 4;
504         }
505         if (opt->ts_needaddr) {
506             optptr[2] -= 4;
507             memset(&optptr[optptr[2]-1], 0, 4);
508         }
509     }
510 }
```

487-491 如果存在源路由选项，则路径列表中的所有地址往后移动一个位置，然后将目的地址，即下一跳地址，重新复制到路径列表的第一个地址处。

492-496 如果存在记录路由选项，则将保存到路径列表中的本地地址删除。

497-509 如果存在时间戳选项，则根据记录时间戳和记录地址标志，将保存的时间戳或本地地址删除。

12.8 处理转发 IP 数据报中的 IP 选项

ip_forward_options()会向将转发的数据报添加所需的关于本地 IP 的信息，包括记录路由选项和时间戳选项。

```
561 void ip_forward_options(struct sk_buff *skb)
562 {
563     struct   ip_options * opt    = &(IPCB(skb)->opt);
564     unsigned char * optptr;
565     struct rtable *rt = (struct rtable*)skb->dst;
566     unsigned char *raw = skb->nh.raw;
567
568     if (opt->rr_needaddr) {
569         optptr = (unsigned char *)raw + opt->rr;
570         ip_rt_get_source(&optptr[optptr[2]-5], rt);
571         opt->is_changed = 1;
572     }
573     if (opt->srr_is_hit) {
574         int srrptr, srrspace;
575
576         optptr = raw + opt->srr;
577
578         for ( srrptr=optptr[2], srrspace = optptr[1];
579               srrptr <= srrspace;
580               srrptr += 4
581             ) {
582             if (srrptr + 3 > srrspace)
583                 break;
584             if (memcmp(&rt->rt_dst, &optptr[srrptr-1], 4) == 0)
585                 break;
586         }
587         if (srrptr + 3 <= srrspace) {
588             opt->is_changed = 1;
589             ip_rt_get_source(&optptr[srrptr-1], rt);
590             skb->nh.iph->daddr = rt->rt_dst;
591             optptr[2] = srrptr+4;
592         } else if (net_ratelimit())
593             printk(KERN_CRIT "ip_forward(): Argh! Destination lost!\n");
594         if (opt->ts_needaddr) {
595             optptr = raw + opt->ts;
596             ip_rt_get_source(&optptr[optptr[2]-9], rt);
597             opt->is_changed = 1;
598         }
599     }
600     if (opt->is_changed) {
601         opt->is_changed = 0;
602         ip_send_check(skb->nh.iph);
603     }
604 }
```

568-572 如果需要记录 IP 地址，则获取本地地址并设置到 IP 记录路由选项中。

573-599 如果目的地址是从源路由选项指定的，则还需要判断输出路由缓存的目的地址是否存在于源路由选项中。如果存在，则根据输出路由缓存的目的地址重新设置 IP 首部中的目的地址。

600-603 一旦 IP 首部作了修改，就需要重新计算 IP 数据报首部的校验和。

12.9　处理 IP 数据报的源路由选项

ip_options_rcv_srr()检查输入数据报中的宽松源路由及严格源路由选项，并根据源路由选项更新 IP 数据报的下一跳地址。

```
606 int ip_options_rcv_srr(struct sk_buff *skb)
607 {
608     struct ip_options *opt = &(IPCB(skb)->opt);
609     int srrspace, srrptr;
610     __be32 nexthop;
611     struct iphdr *iph = skb->nh.iph;
612     unsigned char * optptr = skb->nh.raw + opt->srr;
613     struct rtable *rt = (struct rtable*)skb->dst;
614     struct rtable *rt2;
615     int err;
616
617     if (!opt->srr)
618         return 0;
```

617-618 如果待处理的数据报中没有宽松或严格源路由选项，则不处理直接返回。

```
620     if (skb->pkt_type != PACKET_HOST)
621         return -EINVAL;
622     if (rt->rt_type == RTN_UNICAST) {
623         if (!opt->is_strictroute)
624             return 0;
625         icmp_send(skb, ICMP_PARAMETERPROB, 0, htonl(16<<24));
626         return -EINVAL;
627     }
628     if (rt->rt_type != RTN_LOCAL)
629         return -EINVAL;
```

620-621 待处理 IP 数据报其接收方必须是本地主机，否则返回参数无效错误。

622-627 在路由类型为 RTN_UNICAST，即网关或直接连接的路由情况下执行严格源路由是会有问题的。此时会发送一个参数错误 ICMP 差错报文给发送方，并返回参数无效错误。

628-629 待处理 IP 数据报的路由目的地址必须为本地主机，否则返回参数无效错误。

```
631     for (srrptr=optptr[2], srrspace = optptr[1]; srrptr <= srrspace; srrptr +=
4) {
632         if (srrptr + 3 > srrspace) {
633             icmp_send(skb, ICMP_PARAMETERPROB, 0, htonl((opt->srr+2)<<24));
634             return -EINVAL;
635         }
636         memcpy(&nexthop, &optptr[srrptr-1], 4);
637
638         rt = (struct rtable*)skb->dst;
639         skb->dst = NULL;
640         err = ip_route_input(skb, nexthop, iph->saddr, iph->tos, skb->dev);
641         rt2 = (struct rtable*)skb->dst;
642         if (err || (rt2->rt_type != RTN_UNICAST && rt2->rt_type != RTN_LOCAL)) {
```

```
643             ip_rt_put(rt2);
644             skb->dst = &rt->u.dst;
645             return -EINVAL;
646         }
647         ip_rt_put(rt);
648         if (rt2->rt_type != RTN_LOCAL)
649             break;
650         /* Superfast 8) loopback forward */
651         memcpy(&iph->daddr, &optptr[srrptr-1], 4);
652         opt->is_changed = 1;
653     }
654     if (srrptr <= srrspace) {
655         opt->srr_is_hit = 1;
656         opt->is_changed = 1;
657     }
658     return 0;
659 }
```

631-659 根据源路由选项更新 IP 数据报的下一跳地址。

631 遍历源路由选项中的地址。

632-635 校验源路由选项的路径列表是否还能至少容纳下一个 IP 地址值，如果不能，则给发送方发送一个参数问题 ICMP 差错报文。

636-652 通过输入路由方式来判断是否抵达源路由选项中的某一站，一旦确定本地为源路由选项中的某一站，则获取下一跳的 IP 地址作为该数据报的目的地址，并设置 is_changed，表示该 IP 数据报作了修改。

651-652 一旦本地为源路由选项中的某一站，则获取下一跳的 IP 地址作为该数据报的目的地址，并设置 is_changed，表示该 IP 数据报作了修改。

654-657 如果源路由选项的路径列表没有遍历完，则说明该 IP 数据报的目的地址是从源路由选项选出的，因此需设置 srr_is_hit 标志，待转发时需要进一步处理。同时还需要设置 is_changed 标志，标识需重新计算 IP 数据报的首部校验和。

12.10 解析并处理 IP 首部中的 IP 选项

ip_rcv_options()用于解析并处理接收到的 IP 首部中的 IP 选项。当网络层接收到 IP 数据报时，先校验该 IP 数据报有效性，然后在转发或上传到本地的上层协议之前，需要解析该数据报中的 IP 选项并处理源路由选项。

```
279 static inline int ip_rcv_options(struct sk_buff *skb)
280 {
281     struct ip_options *opt;
282     struct iphdr *iph;
283     struct net_device *dev = skb->dev;
284
285     /* It looks as overkill, because not all
286        IP options require packet mangling.
287        But it is the easiest for now, especially taking
288        into account that combination of IP options
289        and running sniffer is extremely rare condition.
290                          --ANK (980813)
```

```
291     */
292     if (skb_cow(skb, skb_headroom(skb))) {
293         IP_INC_STATS_BH(IPSTATS_MIB_INDISCARDS);
294         goto drop;
295     }
296
297     iph = skb->nh.iph;
298
299     if (ip_options_compile(NULL, skb)) {
300         IP_INC_STATS_BH(IPSTATS_MIB_INHDRERRORS);
301         goto drop;
302     }
303
304     opt = &(IPCB(skb)->opt);
305     if (unlikely(opt->srr)) {
306         struct in_device *in_dev = in_dev_get(dev);
307         if (in_dev) {
308             if (!IN_DEV_SOURCE_ROUTE(in_dev)) {
309                 if (IN_DEV_LOG_MARTIANS(in_dev) &&
310                     net_ratelimit())
311                     printk(KERN_INFO "source route option "
312                            "%u.%u.%u.%u -> %u.%u.%u.%u\n",
313                            NIPQUAD(iph->saddr),
314                            NIPQUAD(iph->daddr));
315                 in_dev_put(in_dev);
316                 goto drop;
317             }
318
319             in_dev_put(in_dev);
320         }
321
322         if (ip_options_rcv_srr(skb))
323             goto drop;
324     }
325
326     return 0;
327 drop:
328     return -1;
329 }
```

292-295 确保数据报足够的头部空间。

299-302 调用 ip_options_compile()解析 skb 的 nh.raw 指向的 IP 首部中 IP 选项到 skb 的 cb 中，IP 层的私有数据 cb 为一个 IP 选项信息块。

304-324 如果存在源路由选项，并且系统允许接收带源路由选项的 IP 数据报，则接收并处理 IP 数据报的源路由选项，否则丢弃该数据报。

12.11　路由警告选项的处理

内核通常不会直接处理路由警告选项，而只是将包含路由警告选项的数据报向上传递给应用服务程序，由需要该选项的进程来处理。处理带路由警告选项数据报的套接口时必须已使能了 IP_ROUTER_ALERT 套接口选项的原始套接口。

通过使能 IP_ROUTER_ALERT 套接口选项，使需要处理带路由警告选项数据报的套接口

加到 ip_ra_chain 链表中（ip_ra_control()参见 15.7.1 节）。当接收到带路由警告选项数据报时，IP 及 IP 组播模块都会调用 ip_call_ra_chain()，将数据报传递给 ip_ra_chain 链表上符合条件的套接口。如果套接口绑定了某个输出网络设备，那就只能接收从该网络设备输入的数据报。

12.12 由控制信息生成 IP 选项信息块

UDP 套接口和 RAW 套接口的输出数据中如果带有控制信息，则需根据携带的控制信息生成相应的 IP 选项信息块，用于生成待输出 IP 数据报的 IP 选项。ip_options_get()就是用来完成这个功能的，参数说明如下：

- optp，一个 IP 选项信息块指针的指针，用来返回生成的 IP 选项信息块。
- data，欲输出控制信息的指针。
- optlen，欲输出控制信息的长度。

```
550 int ip_options_get(struct ip_options **optp, unsigned char *data, int optlen)
551 {
552     struct ip_options *opt = ip_options_get_alloc(optlen);
553
554     if (!opt)
555         return -ENOMEM;
556     if (optlen)
557         memcpy(opt->__data, data, optlen);
558     return ip_options_get_finish(optp, opt, optlen);
559 }
```

552-555 为控制信息创建一个新的 IP 选项信息块。

556-557 将控制信息复制到以 IP 选项信息块__data 字段开始的区域中。

558 调用 ip_options_get_finish()完成对控制信息中 IP 选项的解析。

```
521 static int ip_options_get_finish(struct ip_options **optp,
522                   struct ip_options *opt, int optlen)
523 {
524     while (optlen & 3)
525         opt->__data[optlen++] = IPOPT_END;
526     opt->optlen = optlen;
527     opt->is_data = 1;
528     if (optlen && ip_options_compile(opt, NULL)) {
529         kfree(opt);
530         return -EINVAL;
531     }
532     kfree(*optp);
533     *optp = opt;
534     return 0;
535 }
```

524-525 如果 IP 选项内容不是以 4 字节对齐的，则将未对齐部分用选项列表结束符填充，使之对齐。

526-527 设置 IP 选项信息块中选项长度以及有数据标志。

528-531 调用 ip_options_compile()解析 IP 选项信息块各字段值。

532-534 返回新创建并已填充解析后信息的 IP 选项信息块。

第 13 章　IP 的分片与组装

当要发送的 IP 数据报的长度超出了最大传输单位 MTU，且允许分片时，就会进行 IP 分片。IP 分片通常发生在网络环境中，例如在以太网环境中 MTU 为 1500B，而如果传输的数据报大于 1500B，就需要用到分片技术，经分片后才能传输此数据报。通常，使用 UDP 协议发送的数据报很容易导致 IP 分片，而 TCP 协议是基于流的传输，通常不会产生分片。

IP 数据报被分片后，各分片(fragment)分别组成一个具有 IP 首部的分组，并各自独立地选择路由，在其分别抵达目的主机后，目的主机的 IP 层会在传送给传输层之前将接收到的所有分片重装成一个完整的 IP 数据报。可以这么理解，IP 数据报是通信双方 IP 层之间的传输单元，分片是 IP 层和二层之间的传输单元，而分片对于传输层是透明的。

尽管分片对于传输层是透明的，但在快速分片时，传输层似乎能够感知到分片的存在，而提前将要传递给 IP 层的数据按 MTU 分割完成。但在以后的论述中读者会看到，无论如何传输层是无法替代 IP 层完成全部分片工作的，例如 IP 首部的设置。

IP 数据报输出时的分片以及输入 IP 数据报分片的组装涉及以下文件：

- net/ipv4/ip_output.c，IP 数据报的输出。
- net/ipv4/ip_fragment.c，IP 数据报分片的组装。

13.1　系统参数

系统参数如下：

- ipfrag_high_thresh，可用于组装 IP 数据报的内存上限值，在进行组装 IP 分片时，如果占用的缓存超过了此参数，会调用 ip_evictor()进行垃圾收集。
- ipfrag_low_thresh，可用于组装 IP 数据报的内存下限值，在调用 ip_evictor()进行垃圾收集时，一旦组装 IP 分片所占用的缓存下降到此参数，就会中止垃圾收集。
- ipfrag_max_dist，允许接收来自同一个源 IP 地址的 IP 分片数量的上限值，通常用于防御 DoS 攻击，默认值为 64，当 ipfrag_max_dist 为 0 时，表示没有任何限制。
- ipfrag_secret_interval，定时重组 ipq 散列表的时间间隔，默认值为 600s。
- ipfrag_time，待组装 IP 分片在内存中允许保留的时间，默认值为 30s。

13.2　分片

在 ip_finish_output()将数据报发送出去之前，该数据报包括本地发送的数据报也包括转发的数据报，会对 skb 缓存区的数据长度进行检测，如果超出了输出设备的 MTU 值，且又符合其他一些条件，则调用 ip_fragment()对数据报进行分片；否则直接调用 ip_finish_output2()输出

到数据链路层。

目前有两种分片的处理方式：快速分片和慢速分片。如图 13-1 所示，整个分片过程需要完成的工作，除了将三层的有效负载根据 MTU 分成一个个片段之外，还需要为每个分片设置 IP 首部，更新 IP 校验和等。在快速分片中，将数据分割成片段已经由传输层完成，三层只需将这些片段组成 IP 分片；而慢速分片则需完成全部的工作，即对一个完整的 IP 数据报根据 MTU 值循环进行分片，直至完成。正因为快速分片需要传输层的协助，因此只对本地发送的数据报才有可能，快速分片前 skb 缓存区组成如图 13-2 所示。

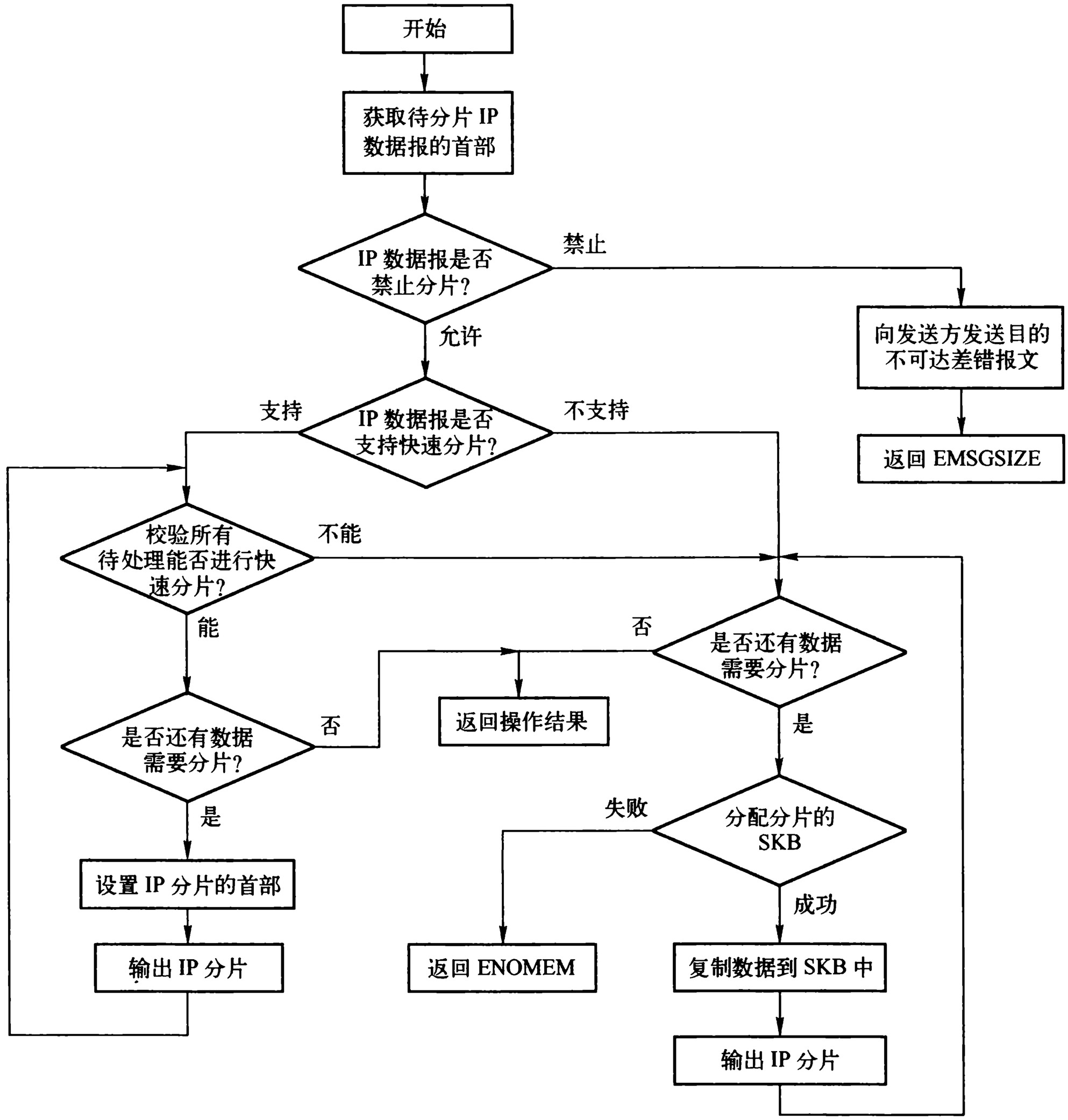

图 13-1　ip_fragment()流程

当要将一个 IP 数据报从本地发送或转发出去时，如果发现该 IP 数据报大于当前的 MTU 或路径 MTU，则调用 ip_fragment()将数据报分片后再发送出去。参数说明如下：

- skb，待分片后发送或转发的 IP 数据报，即原始数据报，该数据报应该包含已初始化的 IP 首部。

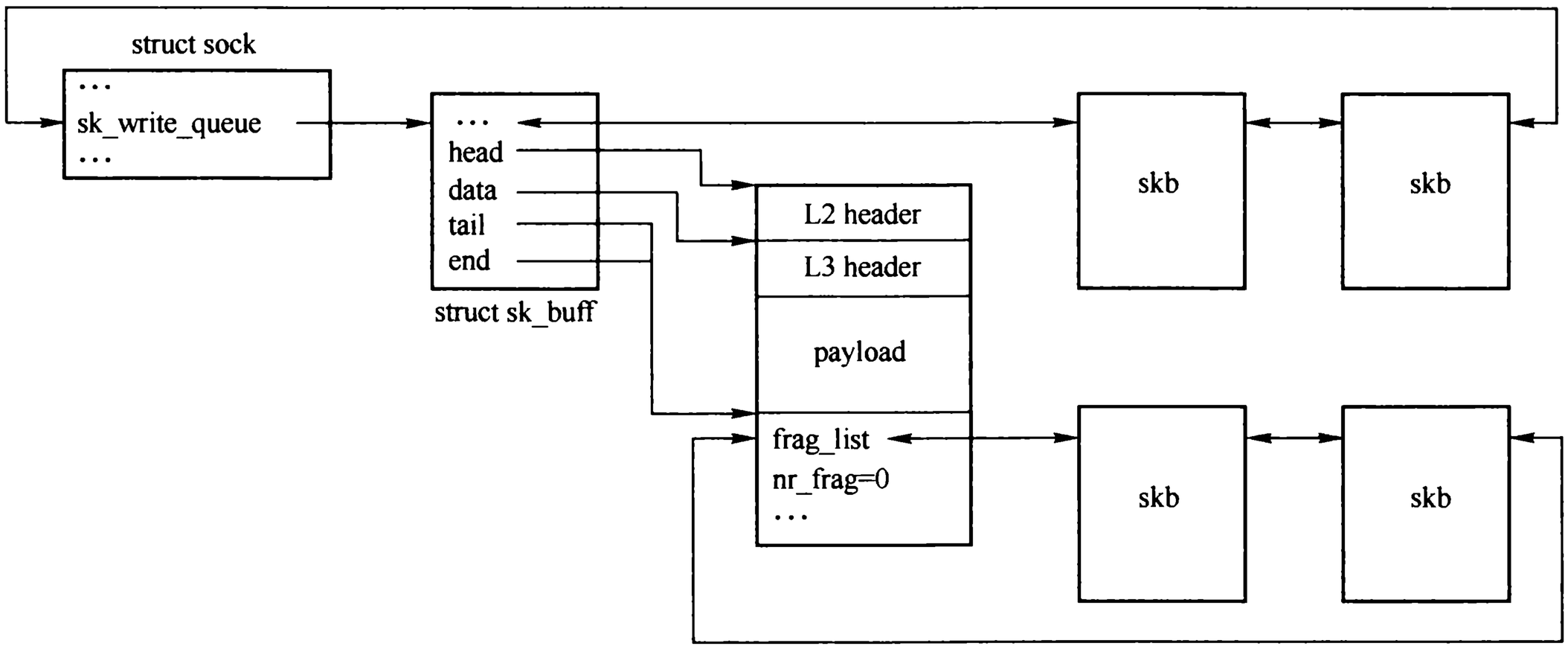

图 13-2　快速分片前的 skb 缓存区

- output，将完成分片输出的回调函数，IPv4 中为 ip_finish_output2()。

```
415 int ip_fragment(struct sk_buff *skb, int (*output)(struct sk_buff*))
416 {
417     struct iphdr *iph;
418     int raw = 0;
419     int ptr;
420     struct net_device *dev;
421     struct sk_buff *skb2;
422     unsigned int mtu, hlen, left, len, ll_rs, pad;
423     int offset;
424     __be16 not_last_frag;
425     struct rtable *rt = (struct rtable*)skb->dst;
426     int err = 0;
427
428     dev = rt->u.dst.dev;
429
430     /*
431      *    Point into the IP datagram header.
432      */
433
434     iph = skb->nh.iph;
435
436     if (unlikely((iph->frag_off & htons(IP_DF)) && !skb->local_df)) {
437         IP_INC_STATS(IPSTATS_MIB_FRAGFAILS);
438         icmp_send(skb, ICMP_DEST_UNREACH, ICMP_FRAG_NEEDED,
439               htonl(dst_mtu(&rt->u.dst)));
440         kfree_skb(skb);
441         return -EMSGSIZE;
442     }
```

434 获取待分片 IP 数据报的 IP 首部。

436-442 如果待分片 IP 数据报禁止分片，则调用 icmp_send()向发送方发送一个原因为需要分片而设置了不分片标志的目的不可达 ICMP 报文，并丢弃报文，即设置 IP 状态为分片失败，

释放 skb，返回消息过长错误码。

```
444     /*
445      *    Setup starting values.
446      */
447
448     hlen = iph->ihl * 4;
449     mtu = dst_mtu(&rt->u.dst) - hlen;    /* Size of data space */
450      IPCB(skb)->flags |= IPSKB_FRAG_COMPLETE;
```

448-449 获取待分片 IP 数据报的 IP 首部长度，以及扣除了 IP 首部长度的 MTU 值。

450 在分片之前先给 IP 数据报的控制块设置 IPSKB_FRAG_COMPLETE 标志，标识完成分片。

13.2.1 快速分片

当传输层已将数据分块，并将这些块链接在 skb_shinfo(skb)->frag_list，此时则可以通过快速分片进行处理。

```
452     /* When frag_list is given, use it. First, check its validity:
453      * some transformers could create wrong frag_list or break existing
454      * one, it is not prohibited. In this case fall back to copying.
455      *
456      * LATER: this step can be merged to real generation of fragments,
457      * we can switch to copy when see the first bad fragment.
458      */
459     if (skb_shinfo(skb)->frag_list) {
460         struct sk_buff *frag;
461         int first_len = skb_pagelen(skb);
462
463         if (first_len - hlen > mtu ||
464             ((first_len - hlen) & 7) ||
465             (iph->frag_off & htons(IP_MF|IP_OFFSET)) ||
466             skb_cloned(skb))
467              goto slow_path;
```

459 判断此 IP 数据报是否支持快速分片，即如果该数据报第一个 SKB 的 skb_shared_info 结构中的 frag_list 上链接有分片的 SKB，则说明传输层已经为快速分片做好了准备，参见 33.14 节和 11.11.3 节。

461 获得此 IP 数据报第一个分片数据长度，包括 SG 类型聚合分散 I/O 数据区中的数据，参见 3.4.10 节。

463-467 对第一个分片作检测。要进行快速分片，还需要对传输层传递的所有 SKB 做一些判断，在以下四种情况下是不能进行快速分片的：

- 有分片长度大于 MTU。
- 除最后一个分片外还有分片长度未 8 字节对齐。
- IP 首部中的 MF 或片偏移不为 0，说明 SKB 不是一个完整的 IP 数据报。
- 此 SKB 被克隆。

```
469         for (frag = skb_shinfo(skb)->frag_list; frag; frag = frag->next) {
470             /* Correct geometry. */
```

```
471             if (frag->len > mtu ||
472                 ((frag->len & 7) && frag->next) ||
473                 skb_headroom(frag) < hlen)
474                 goto slow_path;
475
476             /* Partially cloned skb? */
477             if (skb_shared(frag))
478                 goto slow_path;
479
480             BUG_ON(frag->sk);
481             if (skb->sk) {
482                 sock_hold(skb->sk);
483                 frag->sk = skb->sk;
484                 frag->destructor = sock_wfree;
485                 skb->truesize -= frag->truesize;
486             }
487         }
```

469 遍历后继所有分片，作相关设置。

471-474 在遍历分片中，要继续对分片做校验。包括分片的长度是否超过 MTU，不是最后一个分片的长度有没有 8 字节对齐了，缓存区头部是否有足够空间来存放 IP 首部，否则不适合进行快速分片。

471-478 先还是对分片的校验，在以下三种情况下是不能进行快速分片的：

- 有分片长度大于 MTU。
- 除最后一个分片外还有分片长度未 8 字节对齐。
- 有分片没有为二层首部留出足够的空间。

477-478 判断每个分片是否被克隆，如果被克隆，则也不适合进行快速分片。

481-486 递增数据报所属传输控制块的引用计数，当前分片的传输控制块字段指向该传输控制块。然后设置 skb 释放回调函数。最后修改第一个分片的缓存区总长度，减去当前分片的长度。

```
489          /* Everything is OK. Generate! */
490
491         err = 0;
492         offset = 0;
493         frag = skb_shinfo(skb)->frag_list;
494         skb_shinfo(skb)->frag_list = NULL;
495         skb->data_len = first_len - skb_headlen(skb);
496         skb->len = first_len;
497         iph->tot_len = htons(first_len);
498         iph->frag_off = htons(IP_MF);
499         ip_send_check(iph);
```

491-492 初始化错误码及分片偏移。

493-494 保存 frag_list 指针，并将该指针置为 NULL。

分片以后，每一个分片都将是一个独立的 IP 数据报，因此在此将原来链接在第一个分片 frag_list 上的所有后继分片 skb 取下，同时为了进一步对这些后继分片 skb 进行处理，需保存该指针。

495-496 重新设置第一个分片的数据长度和缓存区长度。

497-498 设置第一个分片 IP 首部中的总长度字段和 MF 标志位。

499 重新计算第一个分片 IP 首部校验和。

```
501         for (;;) {
502             /* Prepare header of the next frame,
503              * before previous one went down. */
504             if (frag) {
505                 frag->ip_summed = CHECKSUM_NONE;
506                 frag->h.raw = frag->data;
507                 frag->nh.raw = __skb_push(frag, hlen);
508                 memcpy(frag->nh.raw, iph, hlen);
509                 iph = frag->nh.iph;
510                 iph->tot_len = htons(frag->len);
511                 ip_copy_metadata(frag, skb);
512                 if (offset == 0)
513                     ip_options_fragment(frag);
514                 offset += skb->len - hlen;
515                 iph->frag_off = htons(offset>>3);
516                 if (frag->next != NULL)
517                     iph->frag_off |= htons(IP_MF);
518                 /* Ready, complete checksum */
519                 ip_send_check(iph);
520             }
521
522             err = output(skb);
523
524             if (!err)
525                 IP_INC_STATS(IPSTATS_MIB_FRAGCREATES);
526             if (err || !frag)
527                 break;
528
529             skb = frag;
530             frag = skb->next;
531             skb->next = NULL;
532         }
```

501-532 从第二个分片开始循环设置每个分片的 skb 及 IP 首部（此时第一个分片已设置完成），然后将所有的分片（包括第一个分片）发送出去。

504 在发送当前分片之前，需先完成对后一个分片的相应设置，其中一些是根据当前分片来设置的。

505 设置后一个分片的校验和完全由软件处理。

506-507 设置后一个分片 skb 中指向三层和四层首部的指针。

508-510 将当前分片的 IP 首部复制给后一个分片，并修改后一个分片 IP 首部的总长度字段。

511 根据当前分片的 skb 填充后一个分片 skb 中的参数。

512-513 如果是在处理第一个分片，则调用 ip_options_fragment()将第二个分片 skb 中无需复制到每个分片的 IP 选项都填充为 IPOPT_NOOP，此后所有的分片选项部分都简单地复制上一个的即可。ip_options_fragment()参见 12.5 节。

514-519 设置后一个分片 IP 首部的片偏移值、MF 标志以及校验和。

522 调用参数中给出的输出回调函数 output()，将当前分片发送出去。

524-525 对 MIB 的 IPSTATS_MIB_FRAGCREATES 数据进行统计。

526-527 如果发送当前分片失败，或已无后续分片，则结束分片和发送。也就是说一旦有一个分片发送失败，则剩余分片都将不再发送。

529-531 从链表中取下下一个待处理分片。

```
534         if (err == 0) {
535             IP_INC_STATS(IPSTATS_MIB_FRAGOKS);
536             return 0;
537         }
538
539         while (frag) {
540             skb = frag->next;
541             kfree_skb(frag);
542             frag = skb;
543         }
544         IP_INC_STATS(IPSTATS_MIB_FRAGFAILS);
545         return err;
546     }
```

534-537 如果发送成功所有分片，则对 MIB 的 IPSTATS_MIB_FRAGOKS 数据进行统计后返回。

539-544 如果有分片发送失败，则释放所有未发送 IP 分片，对 MIB 的 IPSTATS_MIB_FRAGFAILS 数据进行统计后返回。

13.2.2 慢速分片

当待输出的 IP 数据报不支持快速分片时，则只能通过慢速分片处理，尽管性能会低一些。

```
548 slow_path:
549     left = skb->len - hlen;         /* Space per frame */
550     ptr = raw + hlen;           /* Where to start from */
551
552     /* for bridged IP traffic encapsulated inside f.e. a vlan header,
553      * we need to make room for the encapsulating header
554      */
555     pad = nf_bridge_pad(skb);
556     ll_rs = LL_RESERVED_SPACE_EXTRA(rt->u.dst.dev, pad);
557     mtu -= pad;
558
559     /*
560      *    Fragment the datagram.
561      */
562
563     offset = (ntohs(iph->frag_off) & IP_OFFSET) << 3;
564     not_last_frag = iph->frag_off & htons(IP_MF);
```

548 进行慢速分片处理，在慢速分片处理中，涉及缓存区复制。

549 获取待分片的 IP 数据报的数据长度，此处减去 hlen 是为二层首部留出空间。

550 获取 IP 数据报中数据区指针。

555，557 如果是桥转发基于 VLAN 的 IP 数据报，则需获得 VLAN 首部长度，在后面分配 skb 缓存区时留下相应的空间，同时还需修改 MTU 值。

556 获得二层首部长度。

563 获得 IP 首部中的片偏移值，即每个分片起始处在原始数据报中位置，该值是 13 位的，因此要乘 8。

564 取的 MF 位值，MF 值除最后一个分片外都应该置为 1，表示该分片之后还有分片。

```
566     /*
567      *    Keep copying data until we run out.
568      */
569
570     while(left > 0)    {
571        len = left;
572        /* IF: it doesn't fit, use 'mtu' - the data space left */
573        if (len > mtu)
574           len = mtu;
575        /* IF: we are not sending upto and including the packet end
576           then align the next start on an eight byte boundary */
577        if (len < left)    {
578           len &= ~7;
579         }
```

570 循环对 left 长度的数据进行分片，为每一个分片创建一个新的 SKB。

573-574 如果剩余数据的长度大于 MTU，则以 MTU 为分片长度进行分片；否则就以剩余数据的长度作为分片长度，显然后一种情况只会出现在最后一个分片。

577-579 除非是最后一个分片，否则分片不包括 IP 首部的数据部分，需 8 字节对齐。

```
580         /*
581          *    Allocate buffer.
582          */
583
584        if ((skb2 = alloc_skb(len+hlen+ll_rs, GFP_ATOMIC)) == NULL) {
585           NETDEBUG(KERN_INFO "IP: frag: no memory for new fragment!\n");
586           err = -ENOMEM;
587           goto fail;
588         }
```

584-588 为分片分配一个 SKB，其长度为分片长、IP 首部长，以及二层首部长之和。

```
590         /*
591          *    Set up data on packet
592          */
593
594         ip_copy_metadata(skb2, skb);
595         skb_reserve(skb2, ll_rs);
596         skb_put(skb2, len + hlen);
597         skb2->nh.raw = skb2->data;
598         skb2->h.raw = skb2->data + hlen;
```

594 根据原始数据报 skb 填充分片新分配的 SKB，包括类型、优先级、协议等等。

595-596 分别为二层首部、分片加 IP 首部留出相应的空间和数据空间，以备后面填充数据。

597-598 设置新 SKB 中三层首部指针和四层首部指针。

```
600         /*
601          *    Charge the memory for the fragment to any owner
```

```
602         *    it might possess
603         */
604
605        if (skb->sk)
606           skb_set_owner_w(skb2, skb->sk);
607
608        /*
609         *    Copy the packet header into the new buffer.
610         */
611
612        memcpy(skb2->nh.raw, skb->data, hlen);
613
614        /*
615         *    Copy a block of the IP datagram.
616         */
617        if (skb_copy_bits(skb, ptr, skb2->h.raw, len))
618           BUG();
619        left -= len;
620
621        /*
622         *    Fill in the new header fields.
623         */
624        iph = skb2->nh.iph;
625        iph->frag_off = htons((offset >> 3));
```

605-606 设置新 SKB 的宿主，包括递增数据报传输控制块的引用数，设置 SKB 的释放回调函数等。

612 复制 IP 首部到新 SKB 中。

617-619 复制分片数据，并更新原始数据报剩余未分片数据量。此处不同于上面复制 IP 首部时直接调用 memcpy()，而是调用了 skb_copy_bits()，是因为 skb 中的数据存储有多种可能性，而 skb_copy_bits()可以处理这些细节。

624-625 设置分片的片偏移字段，对于第一个分片，该值即原始 IP 数据报的片偏移字段值。

```
627         /* ANK: dirty, but effective trick. Upgrade options only if
628         * the segment to be fragmented was THE FIRST (otherwise,
629         * options are already fixed) and make it ONCE
630         * on the initial skb, so that all the following fragments
631         * will inherit fixed options.
632         */
633        if (offset == 0)
634           ip_options_fragment(skb);
635
636        /*
637         *    Added AC : If we are fragmenting a fragment that's not the
638         *          last fragment then keep MF on each bit
639         */
640        if (left > 0 || not_last_frag)
641           iph->frag_off |= htons(IP_MF);
642        ptr += len;
643        offset += len;
644
645        /*
646         *    Put this fragment into the sending queue.
647         */
```

```
648         iph->tot_len = htons(len + hlen);
```

633-634 根据条件清理掉 IP 选项中一些选项，设置为 IPOPT_NOOP。因为这些选项只要针对一个 IP 数据报处理一次即可，没有必要每个分片都处理。

633-634 第一个分片的 IP 选项是完整的，而其余分片中只存在某些选项，因此其余分片中无需复制到分片的 IP 选项都需用 IPOPT_NOOP 填充。因为分片中的选项从原始 IP 数据报中复制过来，而在此作一次这样的处理，可以保证之后分片 IP 选项的正确，参见 12.5 节。

640-641 如果不是最后一个分片，则设置 IP 首部中标识字段的 MF 位。

642-644 更新后一个分片在整个原始数据报中的偏移量，以及后一个分片在当前被分片数据报中的偏移量。这两个偏移量是有区别的，因为一个数据报在传输过程中可能被多次分片，因此当前被分片的数据报也有可能是另外一个数据报的分片。

648 设置分片 IP 首部中总长度字段。

```
650         ip_send_check(iph);
651
652         err = output(skb2);
653         if (err)
654             goto fail;
655
656         IP_INC_STATS(IPSTATS_MIB_FRAGCREATES);
657     }
658     kfree_skb(skb);
659     IP_INC_STATS(IPSTATS_MIB_FRAGOKS);
660     return err;
661
662 fail:
663     kfree_skb(skb);
664     IP_INC_STATS(IPSTATS_MIB_FRAGFAILS);
665     return err;
666 }
```

650 重新计算该分片的 IP 首部校验和。

652-654 调用参数中给出的输出回调函数 output()，将当前分片发送出去。

656 对 MIB 的 IPSTATS_MIB_FRAGCREATES 数据进行统计。

658-660 完成所有分片及发送之后，释放被分片的 IP 数据报。同时对 MIB 的 IPSTATS_MIB_FRAGOKS 数据进行统计。

13.3 组装

在接收方，一个由发送方发出的原始 IP 数据报，其所有分片将被重新组合，然后才能提交到上层协议。每一个将被重新组合的 IP 数据报都用一个 ipq 结构实例来表示，因此先来看 ipq 这个非常重要的结构。

13.3.1 ipq 结构

为了能高效地组装分片，用于保存分片的数据结构必须能够做到以下几点：

1）快速定位属于某一数据报的一组分片。

2）在属于某一数据报的一组分片中快速插入新的分片。

3）有效地判断一个数据报的所有分片是否已经全部接收。

4）具有组装超时机制，如果在重组完成之前定时器溢出，则删除该数据报的所有内容。

内核中使用 ipq 结构及 ipq 散列表来达到以上特点。其中 ipq 结构用来存储一个完整 IP 数据报的全部分片，该结构体保存有足够的信息，在所有分片到达后，将它们还原成原始数据报。

```
76  struct ipq {
77      struct hlist_node list;
78      struct list_head lru_list;    /* lru list member          */
79      u32       user;
80      __be32        saddr;
81      __be32        daddr;
82      __be16        id;
83      u8        protocol;
84      u8        last_in;
85  #define COMPLETE        4
86  #define FIRST_IN        2
87  #define LAST_IN           1
88
89      struct sk_buff    *fragments;    /* linked list of received fragments    */
90      int       len;        /* total length of original datagram    */
91      int       meat;
92      spinlock_t    lock;
93      atomic_t    refcnt;
94      struct timer_list timer;    /* when will this queue expire?         */
95      struct timeval    stamp;
96      int           iif;
97      unsigned int    rid;
98      struct inet_peer *peer;
99  };
```

77 `struct hlist_node list`

用来在 ipq_hash 散列表链接成双向链表。

78 `struct list_head lru_list`

用来将 ipq 连接到全局链表 ipq_lru_list 链表中。ipq_lru_list 链表用于垃圾收集，当 IP 组装模块消耗的内存大于规定的上限时，会遍历该链表清除符合条件的分片。

79 `u32 user`

标识分片来源，部分值见表 13-1。

表 13-1　user 的取值（部分）

user	描述
IP_DEFRAG_LOCAL_DELIVER	来自网络其他主机或是本地环回接口的分片
IP_DEFRAG_CALL_RA_CHAIN	含有路由警告选项的 IP 分片

80 `__be32 saddr`

81 __be32 daddr

82 __be16 id

83 u8 protocol

以上四个字段的值都来源于 IP 首部，用来唯一确定分片来自哪个 IP 数据报。

84 u8 last_in

标志，见表 13-2。

表 13-2 last_in 的取值

last_in	描述
COMPLETE	所有分片都已到达，可进行组装
FIRST_IN	第一个分片已到达，其特殊之处在于只有第一个分片包含了所有的 IP 选项
LAST_IN	最后一个分片已到达，最后一个分片带有原始数据报的长度信息

89 struct sk_buff *fragments

用来链接已经接收到的分片。

90 int len

当前已收到分片中 offset 最大的那个分片的 offset 值加上其长度值，即分片末尾处在整个原始数据报中的位置，因此当收到最后一个分片后该字段值将更新为原始数据报的长度。

91 int meat

已接收到的所有分片总长度，因此可以用 len 和 meat 来判断一个 IP 数据报的所有分片是否已到齐。

92 spinlock_t lock

自旋锁，在 SMP 环境下，处理 ipq 及分片链时需上锁。

93 atomic_t refcnt

引用计数。

94 struct timer_list timer

组装超时定时器，组装分片非常耗资源，因此不可能无休止地等待分片的到达。

95 struct timeval stamp

记录最后一个分片到达的时间，在组装数据报时用该值作为时间戳。

96 int iif

接收最后一个分片的网络设备索引号。当分片组装失败时，用该设备发送分片组装超时 ICMP 出错报文，即类型为 ICMP_TIME_EXCEEDED，代码为 ICMP_EXC_FRAGTIME。ip_expire()参见 13.3.4 节。

97 unsigned int rid

已接收到分片的计数器。可通过对端信息块 peer 中的分片计数器和该分片计数器来防止 DoS 攻击。

98 struct inet_peer *peer

记录发送方的一些信息，参见 11.9 节。

图 13-3 显示了 ipq 散列表、ipq 分片头和分片之间的关系。

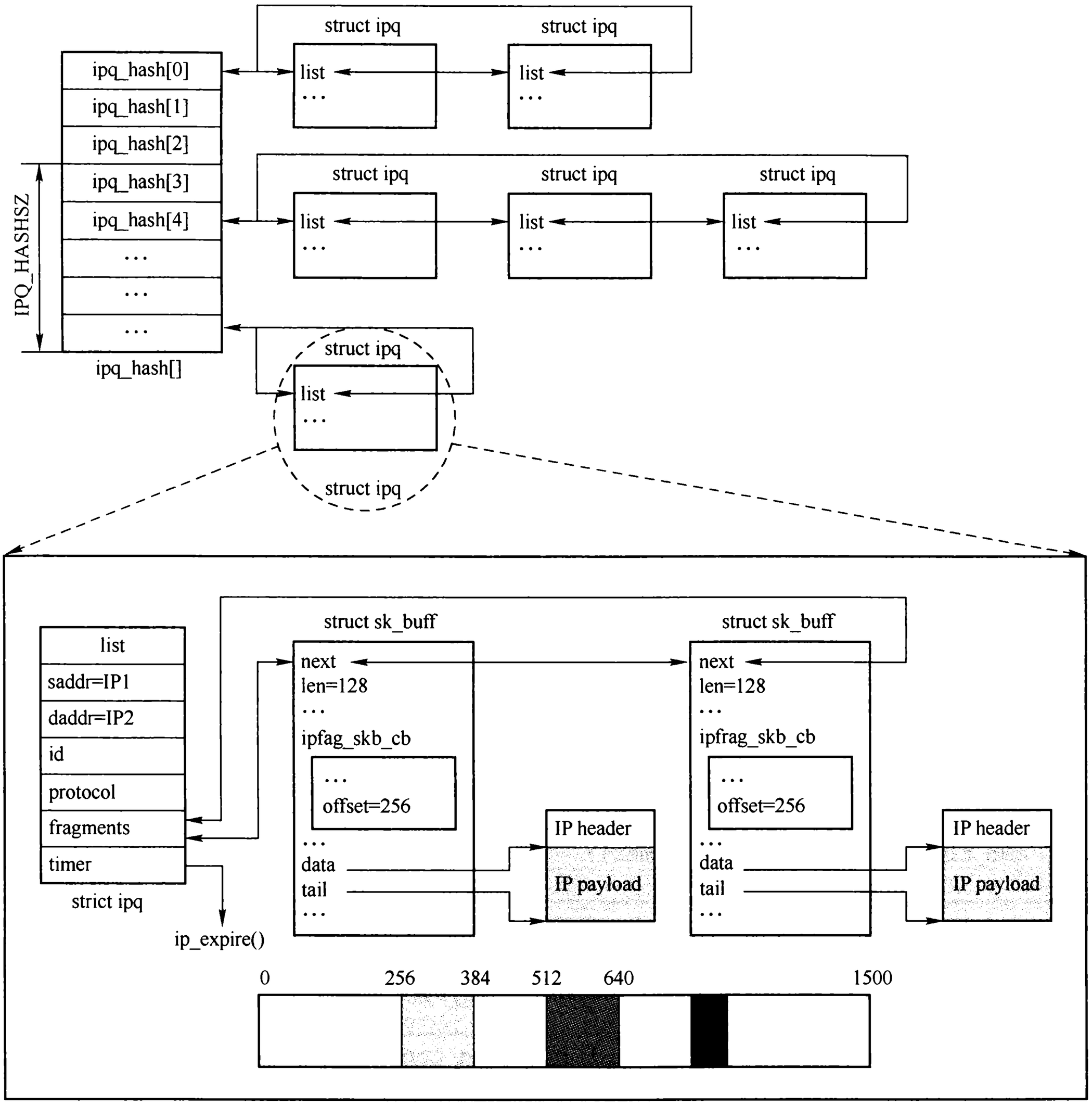

图 13-3　ipq 散列表

13.3.2　ipq 散列表和链表的维护

（1）ipq_kill()

将组装定时器超时的 ipq 上从 ipq 散列表及 ipq_lru_list 链表中删除，但并不释放。首先停止该 ipq 上的组装定时器，并递减 ipq 的引用计数。然后调用 ipq_unlink()，将 ipq 从散列表和链表中删除，引用计数再减一。最后设置 ipq 的 last_in 为 COMPLETE 状态，之所以这样设置，是因为只有 COMPLETE 状态的 ipq 才能被释放。

（2）ipq_put()

用来释放 ipq 及分片。实际上，ipq_put()只是封装了 ip_frag_destroy()，而后者真正实现了对 ipq 和分片的释放。在调用 ip_frag_destroy()之前，会先用 atomic_dec_and_test()对引用计数判断。

13.3.3 ipq 散列表的重组

所有的分片重组都是通过 ipq 散列表进行的。开始时 ipq 在散列表中的分布会比较均匀，此时的处理性能是最高的，但经过一段时间，不断有 ipq 连接到散列表或从散列表中删除，散列表中 ipq 的分布变得不再均匀，处理性能会大大降低，因此需要定时对散列表进行重组。这样做同时也是为了能够抵御 DoS 攻击。

1. ipfrag_init()

ipq 散列表的定时重组是通过 ipfrag_secret_timer 定时器实现的，在 ipfrag_init()中对 ipfrag_secret_timer 定时器初始化。此外，函数还初始化了 ipfrag_hash_rnd 变量，该变量主要用来与 IP 首部中的源地址、目的地址等构成 ipq 散列表的关键字。在每次 ipq 散列表进行重组时，都会将 ipfrag_hash_rnd 更新为一个新的随机值，并重新设置 ipfrag_secret_timer 定时器，时间跨度为 10min。

```
134 int sysctl_ipfrag_secret_interval __read_mostly = 10 * 60 * HZ;
    ...
737 void ipfrag_init(void)
738 {
739     ipfrag_hash_rnd = (u32) ((num_physpages ^ (num_physpages>>7)) ^
740                (jiffies ^ (jiffies >> 6)));
741
742     init_timer(&ipfrag_secret_timer);
743     ipfrag_secret_timer.function = ipfrag_secret_rebuild;
744     ipfrag_secret_timer.expires = jiffies + sysctl_ipfrag_secret_interval;
745     add_timer(&ipfrag_secret_timer);
746 }
```

2. ipfrag_secret_rebuild()

ipfrag_secret_rebuild()是 ipfrag_secret_timer 定时器的例程，用来对全局的 ipq 散列表进行重组。

```
136 static void ipfrag_secret_rebuild(unsigned long dummy)
137 {
138     unsigned long now = jiffies;
139     int i;
140
141     write_lock(&ipfrag_lock);
142     get_random_bytes(&ipfrag_hash_rnd, sizeof(u32));
143     for (i = 0; i < IPQ_HASHSZ; i++) {
144         struct ipq *q;
145         struct hlist_node *p, *n;
146
147         hlist_for_each_entry_safe(q, p, n, &ipq_hash[i], list) {
148             unsigned int hval = ipqhashfn(q->id, q->saddr,
149                           q->daddr, q->protocol);
150
151             if (hval != i) {
152                 hlist_del(&q->list);
153
154                 /* Relink to new hash chain. */
155                 hlist_add_head(&q->list, &ipq_hash[hval]);
156             }
157         }
```

```
158     }
159     write_unlock(&ipfrag_lock);
160
161     mod_timer(&ipfrag_secret_timer, now + sysctl_ipfrag_secret_interval);
162 }
```

141 ipfrag_lock 是 ipq 散列表的读写锁，在对散列表重构前需对此锁进行写上锁。

142 调用 get_random_bytes()，重新获取 ipfrag_hash_rnd 的随机值。

143-156 遍历 ipq 散列表中所有的 ipq，根据新的 ipfrag_hash_rnd 值把这些 ipq 重新连接到散列表相应桶中。

159 写解锁 ipfrag_lock。

161 重新设置重构定时器的下次到期时间。

13.3.4　超时 IP 分片的清除

由于网络中各种情况复杂，一个 IP 数据报的分片有可能不能全部抵达目的地址，而该数据报已到达的分片会占用大量资源，此外也为了抵御 DoS 攻击，因此需要设置一个时钟，一旦超时，数据报的分片还未全部抵达，则将其已到达的所有分片都清除。

1．ip_frag_create()

每当接收到一个属于新的 IP 数据报的分片时，会为其创建对应的 iqp，并初始化其组装超时定时器。参数 user 参见表 13-1。

```
352 static struct ipq *ip_frag_create(struct iphdr *iph, u32 user)
353 {
354     struct ipq *qp;
355
356     if ((qp = frag_alloc_queue()) == NULL)
357         goto out_nomem;
358
359     qp->protocol = iph->protocol;
360     qp->last_in = 0;
361     qp->id = iph->id;
362     qp->saddr = iph->saddr;
363     qp->daddr = iph->daddr;
364     qp->user = user;
365     qp->len = 0;
366     qp->meat = 0;
367     qp->fragments = NULL;
368     qp->iif = 0;
369     qp->peer = sysctl_ipfrag_max_dist ? inet_getpeer(iph->saddr, 1) : NULL;
370
371     /* Initialize a timer for this entry. */
372     init_timer(&qp->timer);
373     qp->timer.data = (unsigned long) qp;    /* pointer to queue    */
374     qp->timer.function = ip_expire;         /* expire function     */
375     spin_lock_init(&qp->lock);
376     atomic_set(&qp->refcnt, 1);
377
378     return ip_frag_intern(qp);
379
380 out_nomem:
    LIMIT_NETDEBUG(KERN_ERR "ip_frag_create: no memory left !\n");
```

```
382     return NULL;
383 }
```

2. ip_expire()

组装超时定时器例程为 ip_expire()，当定时器被激活时，清除在规定时间内没有完成组装的 ipq 及其所有分片。

```
278 static void ip_expire(unsigned long arg)
279 {
280     struct ipq *qp = (struct ipq *) arg;
281
282     spin_lock(&qp->lock);
283
284     if (qp->last_in & COMPLETE)
285         goto out;
286
287     ipq_kill(qp);
288
289     IP_INC_STATS_BH(IPSTATS_MIB_REASMTIMEOUT);
290     IP_INC_STATS_BH(IPSTATS_MIB_REASMFAILS);
291
292     if ((qp->last_in&FIRST_IN) && qp->fragments != NULL) {
293         struct sk_buff *head = qp->fragments;
294         /* Send an ICMP "Fragment Reassembly Timeout" message. */
295         if ((head->dev = dev_get_by_index(qp->iif)) != NULL) {
296             icmp_send(head, ICMP_TIME_EXCEEDED, ICMP_EXC_FRAGTIME, 0);
297             dev_put(head->dev);
298         }
299     }
300 out:
301     spin_unlock(&qp->lock);
302     ipq_put(qp, NULL);
303 }
```

282 在操作分片链表之前先上锁 ipq 自旋锁。

284-285 若 ipq 当前已是 COMPLETE 的状态，则不作处理，直接跳转到释放 ipq 及其所有的分片处。

287 将 ipq 从 ipq 散列表和 ipq_lru_list 链表中删除。

289-290 对 MIB 的 ReasmTimeout 和 ReasmFails 数据进行统计。

292-299 如果第一个分片已经到达，则发送分片组装超时 ICMP 出错报文。

301 在完成 ipq 分片链表的操作之后解自旋锁。

302 最后释放 ipq 及其所有 IP 分片。

13.3.5 垃圾收集

为了控制 IP 组装所占用的内存，设置了两个阈值 ipfrag_high_thresh 和 ipfrag_low_thresh。当前 ipq 散列表占用的内存量存储在全局变量 ip_frag_mem 中，当 ip_frag_mem 大于 ipfrag_high_thresh 值时，需调用 ip_evictor()对散列表进行清理，直到 ip_frag_mem 降低到 ipfrag_low_thresh。这两个阈值可以在系统运行时通过 proc 文件系统修改。

ip_evictor()主要对 ipq 中的分片进行条件性的清理。在所有的 ipq 中，如果分片没有到齐，

则被删除。

```
244 static void ip_evictor(void)
245 {
246     struct ipq *qp;
247     struct list_head *tmp;
248     int work;
249
250     work = atomic_read(&ip_frag_mem) - sysctl_ipfrag_low_thresh;
251     if (work <= 0)
252         return;
```

在清理之前再次对当前消耗的内存量作测试，如果少于 sysctl_ipfrag_low_thresh，则不进行清理。

```
254     while (work > 0) {
255         read_lock(&ipfrag_lock);
256         if (list_empty(&ipq_lru_list)) {
257             read_unlock(&ipfrag_lock);
258             return;
259         }
260         tmp = ipq_lru_list.next;
261         qp = list_entry(tmp, struct ipq, lru_list);
262         atomic_inc(&qp->refcnt);
263          read_unlock(&ipfrag_lock);
```

255、257、263 操作 ipq 前后需要进行加锁和解锁。

256-259 如果 ipq_lru_list 链表为空，则对 ipfrag_lock 解读锁后返回。

260-263 否则递增 ipq 的引用计数。

```
265         spin_lock(&qp->lock);
266         if (!(qp->last_in&COMPLETE))
267             ipq_kill(qp);
268         spin_unlock(&qp->lock);
269
270         ipq_put(qp, &work);
271         IP_INC_STATS_BH(IPSTATS_MIB_REASMFAILS);
272     }
273 }
```

265、268 在删除分片前后需要做同步保护。

266-267 如果分片还没到齐，则 ipq 上从 ipq 散列表及 ipq_lru_list 链表中删除。注意 ipq_kill() 只删除不释放，参见 13.3.2 节。

270 调用 ipq_put()，真正删除 ipq 及其所有分片。

271 统计 MIB 的 IPSTATS_MIB_REASMFAILS 数据。

13.3.6　相关分片组装函数

1. ip_find()

ip_find()根据分片的 IP 首部以及 user 标志在 ipq 散列表中寻找相对应的 ipq，如果没有找到，则将为其创建新的 ipq。参数 user 参见表 13-1。

```
388 static inline struct ipq *ip_find(struct iphdr *iph, u32 user)
389 {
390     __be16 id = iph->id;
391     __be32 saddr = iph->saddr;
392     __be32 daddr = iph->daddr;
393     __u8 protocol = iph->protocol;
394     unsigned int hash;
395     struct ipq *qp;
396     struct hlist_node *n;
397
398     read_lock(&ipfrag_lock);
399     hash = ipqhashfn(id, saddr, daddr, protocol);
400     hlist_for_each_entry(qp, n, &ipq_hash[hash], list) {
401         if(qp->id == id         &&
402            qp->saddr == saddr   &&
403            qp->daddr == daddr   &&
404            qp->protocol == protocol &&
405            qp->user == user) {
406             atomic_inc(&qp->refcnt);
407             read_unlock(&ipfrag_lock);
408             return qp;
409         }
410     }
411     read_unlock(&ipfrag_lock);
412
413     return ip_frag_create(iph, user);
414 }
```

398、411 遍历 ipq 散列表时进行同步保护。

399 调用 ipqhashfn()计算 hash 值。

400-410 遍历该桶的 ipq 链表查找相应的 ipq，如果找到则返回该 ipq，否则说明这是一个属于新 IP 数据报的分片，需要创建一个新的 ipq。

413 调用 ip_frag_create()，创建新的 ipq。

2. ip_frag_create()

ip_frag_create()创建一个新的 ipq 结构并将其插入到 ipq 散列表中。其参数与 ip_find()的参数相同。

3. ip_frag_intern()

ip_frag_intern()将新创建的 ipq 插入到 ipq 散列表和 ipq_lru_list 链表中。

```
307 static struct ipq *ip_frag_intern(struct ipq *qp_in)
308 {
309     struct ipq *qp;
310 #ifdef CONFIG_SMP
311     struct hlist_node *n;
312 #endif
313     unsigned int hash;
314
315     write_lock(&ipfrag_lock);
316     hash = ipqhashfn(qp_in->id, qp_in->saddr, qp_in->daddr,
317             qp_in->protocol);
318 #ifdef CONFIG_SMP
319     /* With SMP race we have to recheck hash table, because
```

```
320     * such entry could be created on other cpu, while we
321     * promoted read lock to write lock.
322     */
323    hlist_for_each_entry(qp, n, &ipq_hash[hash], list) {
324       if(qp->id == qp_in->id       &&
325          qp->saddr == qp_in->saddr    &&
326          qp->daddr == qp_in->daddr    &&
327          qp->protocol == qp_in->protocol &&
328          qp->user == qp_in->user) {
329           atomic_inc(&qp->refcnt);
330           write_unlock(&ipfrag_lock);
331           qp_in->last_in |= COMPLETE;
332           ipq_put(qp_in, NULL);
333           return qp;
334        }
335    }
336 #endif
```

315、347 遍历 ipq 散列表时进行同步保护。

316 调用 ipqhashfn()计算 hash 值。

318-336 如果支持 SMP，则在把 ipq 加入到 ipq 散列表之前，需检测当前 ipq 是否已存在于 ipq 散列表中了，因为有可能别的处理器已把该 ipq 插入到 ipq 散列表中了。

```
337     qp = qp_in;
338
339    if (!mod_timer(&qp->timer, jiffies + sysctl_ipfrag_time))
340        atomic_inc(&qp->refcnt);
```

339-340 安装 ipq 的组装超时定时器，定时为 sysctl_ipfrag_time。

```
342     atomic_inc(&qp->refcnt);
343    hlist_add_head(&qp->list, &ipq_hash[hash]);
344    INIT_LIST_HEAD(&qp->lru_list);
345    list_add_tail(&qp->lru_list, &ipq_lru_list);
346    ip_frag_nqueues++;
347    write_unlock(&ipfrag_lock);
348    return qp;
349 }
```

342 递增 ipq 的引用计数。

343-345 将 ipq 插入到 ipq 散列表和 ipq_lru_list 链表中。

346 对 ipq 的数量进行计数。

4. ip_frag_queue()

ip_frag_queue()将分片 SKB 添加到由 qp 指定的 ipq 的分片链表中。参数说明如下：

- qp，将添加 IP 分片的 ipq。
- skb，接收到待添加到 ipq 中的 IP 分片。

```
467 static void ip_frag_queue(struct ipq *qp, struct sk_buff *skb)
468 {
469     struct sk_buff *prev, *next;
470     int flags, offset;
471     int ihl, end;
```

```
472
473     if (qp->last_in & COMPLETE)
474         goto err;
475
476     if (!(IPCB(skb)->flags & IPSKB_FRAG_COMPLETE) &&
477         unlikely(ip_frag_too_far(qp)) && unlikely(ip_frag_reinit(qp))) {
478         ipq_kill(qp);
479         goto err;
480      }
```

473-474 对 last_in 标志是 COMPLETE 的 ipq，即分片已全部接收到的 ipq，则释放该分片后返回。

476-480 若不是有本地生成的分片，则调用 ip_frag_too_far()检测该分片是否存在 DoS 攻击嫌疑。如果受到 DoS 攻击，则调用 ip_frag_reinit()释放 ipq 所有分片。

```
482      offset = ntohs(skb->nh.iph->frag_off);
483     flags = offset & ~IP_OFFSET;
484     offset &= IP_OFFSET;
485     offset <<= 3;       /* offset is in 8-byte chunks */
486     ihl = skb->nh.iph->ihl * 4;
489      end = offset + skb->len - ihl;
```

482-486 分别取出 IP 首部中的标志位、片偏移以及首部长度字段，并计算片偏移值和首部长度值。IP 首部中的片偏移字段为 13 位，表示的是 8 字节的倍数，而首部长度字段是首部占 32 位字的数目。

489 计算分片末尾处在原始数据包中的位置。图 13-4 是 ihl、offset、len 和 end 的示意图。

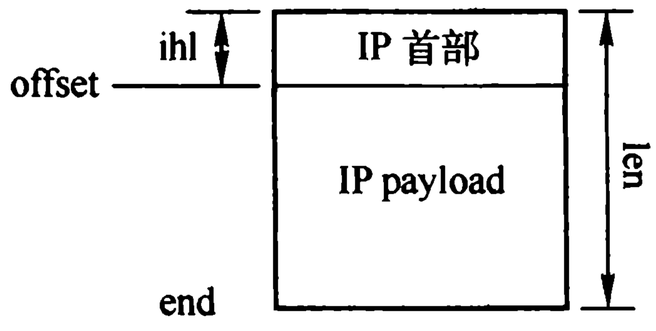

图 13-4　IP 分片示意图

```
491     /* Is this the final fragment? */
492     if ((flags & IP_MF) == 0) {
496         if (end < qp->len ||
497             ((qp->last_in & LAST_IN) && end != qp->len))
498             goto err;
499         qp->last_in |= LAST_IN;
500         qp->len = end;
501     } else {
502         if (end&7) {
503             end &= ~7;
504             if (skb->ip_summed != CHECKSUM_UNNECESSARY)
505                 skb->ip_summed = CHECKSUM_NONE;
506         }
507         if (end > qp->len) {
508             /* Some bits beyond end -> corruption. */
509             if (qp->last_in & LAST_IN)
510                 goto err;
```

```
511             qp->len = end;
512         }
513     }
514     if (end == offset)
515         goto err;
```

492-500 如果是最后一个分片，则先对该分片进行检验：如果其末尾小于原始数据包长度，或者 ipq 已有 LAST_IN 标志并且分片末尾不等于原始数据包长度，则表示出错。通过检验后，对 ipq 设置 LAST_IN 标志，并将完整数据报长度存储在 ipq 的 len 字段中。

502-506 如果不是最后一个分片，其数据长度又不 8 字节对齐，则将其截为 8 字节对齐。如果需要计算校验和，则强制设置由软件来计算校验和。这是因为截断了 IP 有效负载，改变了长度，需重新计算校验和。

507-513 在最后一个分片没有到达的情况下，如果当前分片的末尾在整个数据报中的位置大于 ipq 中 len 字段的值，则更新 len 字段；若此数据报有异常，则直接丢弃。因为 ipq 结构的 len 字段始终保持所有已接收到的分片中分片末尾在数据报中位置的最大值，而只有在收到最后一个分片后，len 值才是整个数据报的长度。

514-515 如果分片的数据区长度为零，则该分片有异常，直接丢弃。

```
517     if (pskb_pull(skb, ihl) == NULL)
518         goto err;
519     if (pskb_trim_rcsum(skb, end-offset))
520         goto err;
```

517-518 调用 pskb_pull()去掉 IP 首部，只保留数据部分。

519-520 调用 pskb_trim_rcsum()将 skb 数据区长度调整为(end-offset)，IP 有效负载长度。

```
522     /* Find out which fragments are in front and at the back of us
523      * in the chain of fragments so far.  We must know where to put
524      * this fragment, right?
525      */
526     prev = NULL;
527     for(next = qp->fragments; next != NULL; next = next->next) {
528         if (FRAG_CB(next)->offset >= offset)
529             break;    /* bingo! */
530         prev = next;
531     }
```

确定分片在分片链表中的位置。因为各分片很可能不按顺序到达目的端，而 ipq 分片链表上的分片是按分片偏移值从小到大的顺序链接在一起的。

```
533     /* We found where to put this one.  Check for overlap with
534      * preceding fragment, and, if needed, align things so that
535      * any overlaps are eliminated.
536      */
537     if (prev) {
538         int i = (FRAG_CB(prev)->offset + prev->len) - offset;
539
540         if (i > 0) {
541             offset += i;
542             if (end <= offset)
543                 goto err;
```

```
544             if (!pskb_pull(skb, i))
545                 goto err;
546             if (skb->ip_summed != CHECKSUM_UNNECESSARY)
547                 skb->ip_summed = CHECKSUM_NONE;
548         }
549     }
```

检测和上一个分片的数据是否有重叠，i 是重叠部分数据长度，如果有重叠则调用 pskb_pull() 去掉重叠部分。

```
551     while (next && FRAG_CB(next)->offset < end) {
552         int i = end - FRAG_CB(next)->offset; /* overlap is 'i' bytes */
553
554         if (i < next->len) {
555             /* Eat head of the next overlapped fragment
556              * and leave the loop. The next ones cannot overlap.
557              */
558             if (!pskb_pull(next, i))
559                 goto err;
560             FRAG_CB(next)->offset += i;
561             qp->meat -= i;
562             if (next->ip_summed != CHECKSUM_UNNECESSARY)
563                 next->ip_summed = CHECKSUM_NONE;
564             break;
565         } else {
566             struct sk_buff *free_it = next;
567
568             /* Old fragment is completely overridden with
569              * new one drop it.
570              */
571             next = next->next;
572
573             if (prev)
574                 prev->next = next;
575             else
576                 qp->fragments = next;
577
578             qp->meat -= free_it->len;
579             frag_kfree_skb(free_it, NULL);
580         }
581     }
```

如果和后一个分片的数据有重叠，则还需要判断重叠部分的数据长度是否超过下一个分片的数据长度，没有超过则调整下一个分片，超过则需要释放下一个分片后再检查与后面第二个分片的数据是否有重叠，如此反复，直到完成后面对所有分片的检测。调整分片的片偏移值、已接收分片总长度等。

```
583     FRAG_CB(skb)->offset = offset;
584
585     /* Insert this fragment in the chain of fragments. */
586     skb->next = next;
587     if (prev)
588         prev->next = skb;
589     else
```

```
590         qp->fragments = skb;
591
592      if (skb->dev)
593          qp->iif = skb->dev->ifindex;
594     skb->dev = NULL;
595     skb_get_timestamp(skb, &qp->stamp);
596     qp->meat += skb->len;
597     atomic_add(skb->truesize, &ip_frag_mem);
598     if (offset == 0)
599         qp->last_in |= FIRST_IN;
600
601     write_lock(&ipfrag_lock);
602     list_move_tail(&qp->lru_list, &ipq_lru_list);
603     write_unlock(&ipfrag_lock);
604
605     return;
606
607 err:
608     kfree_skb(skb);
609 }
```

583 记录当前分片的偏移值。

586-590' 将当前的分片插入到 ipq 分片队列中的相应位置。

595 更新 ipq 的时间戳。

596 累计该 ipq 已收到分片的总长度。

597 累计分片组装模块所占的内存。

598-599 如果片偏移值为 0，则说明当前分片是第一个分片，设置 FIRST_IN 标志。

601-603 调整所属 ipq 在 ipq_lru_list 链表中的位置，这是为了在占用内存超过阈值时可先释放最久未用的那些分片。

5. ip_frag_reasm()

此函数用于组装已到齐的所有分片。当原始数据报的所有分片都已到齐时，会调用此函数组装分片。参数说明如下：

- qp，是存储待组装分片队列的 ipq。
- dev，输入分片的网络设备。

```
614 static struct sk_buff *ip_frag_reasm(struct ipq *qp, struct net_device *dev)
615 {
616     struct iphdr *iph;
617     struct sk_buff *fp, *head = qp->fragments;
618     int len;
619     int ihlen;
620
621     ipq_kill(qp);
626     /* Allocate a new buffer for the datagram. */
627     ihlen = head->nh.iph->ihl*4;
628     len = ihlen + qp->len;
629
630     if(len > 65535)
631         goto out_oversize;
632
633     /* Head of list must not be cloned. */
```

```
634     if (skb_cloned(head) && pskb_expand_head(head, 0, 0, GFP_ATOMIC))
635         goto out_nomem;
```

621 要开始组装了，因此调用 ipq_kill()将此 ipq 结点从 ipq 散列表和 ipq_lru_list 链表中断开，并删除定时器。

627-631 计算原始数据报包括 IP 首部的总长度，如果该长度值超过 64KB 则丢弃。

630-631 IP 数据报总长过了限值则丢弃。

634-635 在组装分片时，所有的分片都会组装到第一个分片上，因此第一个分片不能是克隆的，如果是克隆的，则需为分片组装重新分配一个 SKB。

```
637     /* If the first fragment is fragmented itself, we split
638      * it to two chunks: the first with data and paged part
639      * and the second, holding only fragments. */
640     if (skb_shinfo(head)->frag_list) {
641         struct sk_buff *clone;
642         int i, plen = 0;
643
644         if ((clone = alloc_skb(0, GFP_ATOMIC)) == NULL)
645             goto out_nomem;
646         clone->next = head->next;
647         head->next = clone;
648         skb_shinfo(clone)->frag_list = skb_shinfo(head)->frag_list;
649         skb_shinfo(head)->frag_list = NULL;
650         for (i=0; i<skb_shinfo(head)->nr_frags; i++)
651             plen += skb_shinfo(head)->frags[i].size;
652         clone->len = clone->data_len = head->data_len - plen;
653         head->data_len -= clone->len;
654         head->len -= clone->len;
655         clone->csum = 0;
656         clone->ip_summed = head->ip_summed;
657         atomic_add(clone->truesize, &ip_frag_mem);
658     }
```

分片队列的第一个 SKB 不能既带有数据，又带有分片，即其 frag_list 上不能有分片 skb，如果有则重新分配一个 SKB。最终的效果是，head 自身不包括数据，其 frag_list 上链接着所有分片的 SKB。这也是 SKB 的一种表现形式，不一定是一个连续的数据块，但最终会调用 skb_linearize()将这些数据都复制到一个连续数据块中。

```
660     skb_shinfo(head)->frag_list = head->next;
661     skb_push(head, head->data - head->nh.raw);
662     atomic_sub(head->truesize, &ip_frag_mem);
663
664     for (fp=head->next; fp; fp = fp->next) {
665         head->data_len += fp->len;
666         head->len += fp->len;
667         if (head->ip_summed != fp->ip_summed)
668             head->ip_summed = CHECKSUM_NONE;
669         else if (head->ip_summed == CHECKSUM_COMPLETE)
670             head->csum = csum_add(head->csum, fp->csum);
671         head->truesize += fp->truesize;
672         atomic_sub(fp->truesize, &ip_frag_mem);
673     }
```

把所有分片组装起来即将分片链接到第一个 SKB 的 frag_list 上，同时还需要遍历所有分片，重新计算 IP 数据报长度以及校验和等。

```
675     head->next = NULL;
676     head->dev = dev;
677     skb_set_timestamp(head, &qp->stamp);
678
679     iph = head->nh.iph;
680     iph->frag_off = 0;
681     iph->tot_len = htons(len);
682     IP_INC_STATS_BH(IPSTATS_MIB_REASMOKS);
683     qp->fragments = NULL;
684     return head;
685
686 out_nomem:
687      LIMIT_NETDEBUG(KERN_ERR "IP: queue_glue: no memory for gluing "
688                     "queue %p\n", qp);
689     goto out_fail;
690 out_oversize:
691     if (net_ratelimit())
692        printk(KERN_INFO
693           "Oversized IP packet from %d.%d.%d.%d.\n",
694           NIPQUAD(qp->saddr));
695 out_fail:
696     IP_INC_STATS_BH(IPSTATS_MIB_REASMFAILS);
697     return NULL;
698 }
```

679-681 重置首部长度、片偏移、标志位和总长度。

683 既然各分片都已处理完，释放 ipq 的分片队列。

13.3.7　分片组装

如图 13-5 所示，IP 分片组装的过程分为三步：

1）首先判断接收到的 IP 数据报是否是一个分片，如果是则在传递给上层协议之前，需先进行分片组装。

2）以(daddr，saddr，protocol，id，ipfrag_hash_rnd)计算键值，在 ipq 散列表中找到分片所属的 ipq，并按序插入到该 ipq 的分片链表中。

3）如果此时该 ipq 的所有分片已经全部到达，则将这些分片组装成一个完整的 IP 数据报。

ip_defrag()用来对分片进行组装，返回值不为 0 则表示成功组装一个完整 IP 数据报，否则组装失败。参数说明如下：

- skb，新接收到的 IP 数据报。
- user，分片来源，见表 13-1。

```
701 struct sk_buff *ip_defrag(struct sk_buff *skb, u32 user)
702 {
703     struct iphdr *iph = skb->nh.iph;
704     struct ipq *qp;
705     struct net_device *dev;
706
707     IP_INC_STATS_BH(IPSTATS_MIB_REASMREQDS);
```

```
708
709     /* Start by cleaning up the memory. */
710     if (atomic_read(&ip_frag_mem) > sysctl_ipfrag_high_thresh)
711         ip_evictor();
```

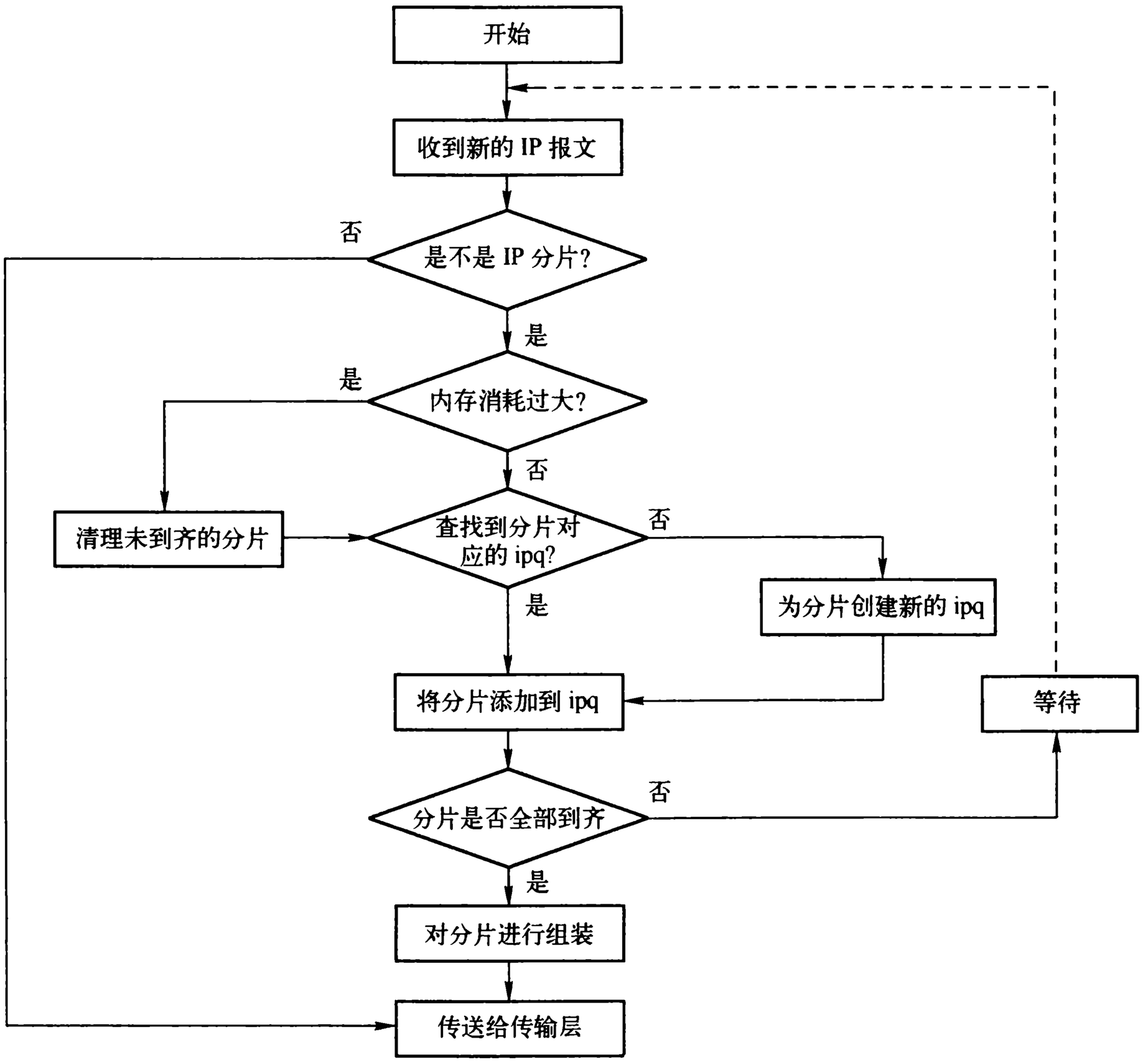

图 13-5　分片组装流程

707 统计 MIB 的 IPSTATS_MIB_REASMREQDS 数据。

710-711 如果 ipq 散列表消耗的内存大于指定值时，需调用 ip_evictor()清理分片。

```
713     dev = skb->dev;
714
715     /* Lookup (or create) queue header */
716     if ((qp = ip_find(iph, user)) != NULL) {
717         struct sk_buff *ret = NULL;
718
719         spin_lock(&qp->lock);
720
721         ip_frag_queue(qp, skb);
722
723         if (qp->last_in == (FIRST_IN|LAST_IN) &&
724             qp->meat == qp->len)
725             ret = ip_frag_reasm(qp, dev);
726
727         spin_unlock(&qp->lock);
```

```
728         ipq_put(qp, NULL);
729         return ret;
730     }
```

713 获取接收数据报的网络设备指针。

716 调用 ip_find()在 ipq 散列表中查找分片所属的 ipq，如果找不到则创建一个新的 ipq，如果返回值为 NULL 则说明查找及创建均告失败。ip_find()参见 13.3.6 节。

719 在操作 ipq 分片链表之前进行上锁。

719、727 操作 ipq 分片链表时要进行同步保护。

721 将分片插入到 ipq 分片链表的适当位置。

723-725 当该 ipq 的第一个分片和最后一个分片都已收到，且如果已接收数据报的总长度与原始数据报的长度相等，则说明该原始数据报的所有分片都已到齐，因此调用 ip_frag_reasm()组装分片。

729 删除 iqp 及其所有分片。

```
732     IP_INC_STATS_BH(IPSTATS_MIB_REASMFAILS);
733     kfree_skb(skb);
734     return NULL;
735 }
```

732 统计 MIB 的 IPSTATS_MIB_REASMFAILS 数据。

733 释放分片。

第 14 章　ICMP：Internet 控制报文协议

ICMP 是网络层的一个协议，可以看作 IP 协议的附属协议，因为它主要被 IP 用来与其他主机或路由器交换错误报文及其他需注意的信息。当然，更高层协议（TCP/UDP）甚至有些用户进程也可能用到 ICMP 报文。

注册 ICMP 协议和 ICMP 协议的处理涉及以下文件：

- net/ipv4/icmp.c，ICMP 协议的处理。
- net/ipv4/af_inet.c，网络层和传输层接口。

14.1　ICMP 报文结构

图 14-1 是 ICMP 报文的整体结构，从图中可以看出 ICMP 报文是被封装在 IP 数据报中的。在分析代码时，会具体分析所遇到报文的特定结构。

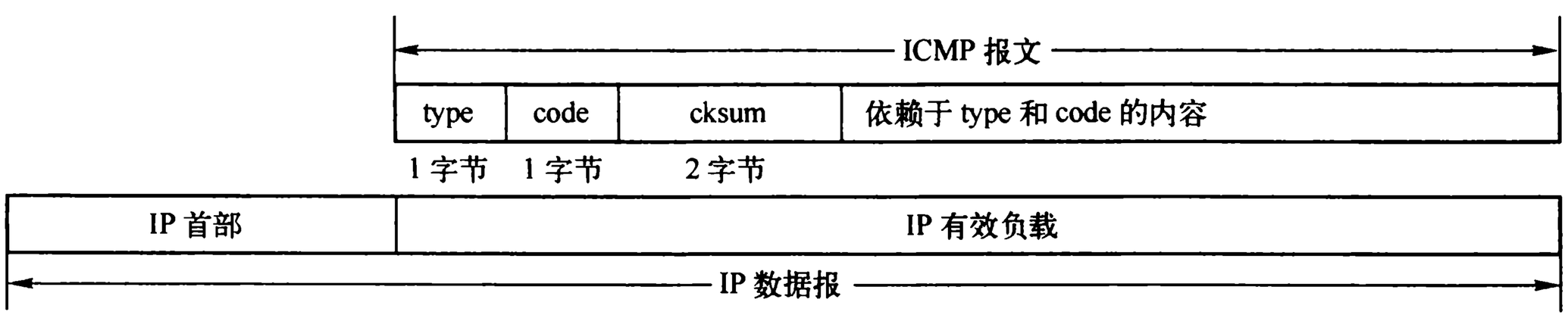

图 14-1　ICMP 报文格式

其中，type 标识 ICMP 报文的类型；code 标识 ICMP 报文的编码；cksum 是 ICMP 报文的校验和。

14.2　注册 ICMP 报文类型

ICMP 的 net_protocol 结构为 icmp_protocol，定义了接收 ICMP 报文例程为 icmp_rcv()。

```
1211 static struct net_protocol icmp_protocol = {
1212     .handler =    icmp_rcv,
1213 };
```

14.3　系统参数

ICMP 系统参数如下：

（1）icmp_echo_ignore_all

用于确定接收或忽略请求回显报文，见表 14-1。

表 14-1　icmp_echo_ignore_all 取值

icmp_echo_ignore_all	描述
0（默认值）	接收所有的请求回显报文
1	忽略所有的请求回显报文

（2）icmp_echo_ignore_broadcasts

用于确定接收或忽略回显的目的地址为子网地址的报文，见表 14-2。

表 14-2　icmp_echo_ignore_broadcasts 取值

icmp_echo_ignore_broadcasts	描述
0（默认值）	接收请求回显的目的地址为子网地址的报文
1	忽略请求回显的目的地址为子网地址的报文

（3）icmp_ignore_bogus_error_responses

目的不可达并且目的 IP 地址为广播地址的 ICMP 报文是无效的，但有些路由器可能会发送这些错误的报文。因此利用这个开关来确定是否忽略这样的无效的 ICMP 报文，见表 14-3。

表 14-3　icmp_ignore_bogus_error_responses 取值

icmp_ignore_bogus_error_responses	描述
0	忽略“目的不可达并且目的 IP 地址为广播地址”这样无效的 ICMP 报文
1（默认值）	接收“目的不可达并且目的 IP 地址为广播地址”这样无效的 ICMP 报文

（4）icmp_errors_use_inbound_ifaddr

选择输出 ICMP 报文的源 IP 地址和设备的方法，见表 14-4。

表 14-4　icmp_errors_use_inbound_ifaddr 取值

icmp_errors_use_inbound_ifaddr	描述
0（默认值）	在主机所有的网络设备上选择首选地址作为源地址来发送 ICMP 信息
1	导致输出这个 ICMP 报文的输入接口上的首选地址作为源地址来发送 ICMP 消息

（5）icmp_ratemask 与 icmp_ratelimit

icmp_ratelimit 限制 ICMP 报文与 icmp_ratemask 位图类型相匹配的输出速率，当 icmp_ratelimit 为 0 表示禁止限速。

icmp_ratemask 为限制 ICMP 报文的类型，默认值为 0000001100000011000 (6168)，详细意义见表 14-5。

表 14-5　icmp_ratemask 取值

icmp_ratemask 位值	描述
0	回显应答
3	目的不可达
4	源端被关闭

（续）

icmp_ratemask 位值	描述
5	重定向
8	回显请求
11	超时
12	参数问题
13	时间戳请求
14	时间戳应答
15	信息请求
16	信息应答
17	地址掩码请求
18	地址掩码应答

14.4 ICMP 的初始化

ICMP 初始化函数是 icmp_init()，在 inet_init()中被调用，其主要功能是为每个 CPU 创建一个基于原始流、IPPROTO_ICMP 协议类型的套接口供内核使用。

```
1105 void __init icmp_init(struct net_proto_family *ops)
1106 {
1107     struct inet_sock *inet;
1108     int i;
1109
1110     for_each_possible_cpu(i) {
1111         int err;
1112
1113         err = sock_create_kern(PF_INET, SOCK_RAW, IPPROTO_ICMP,
1114                                &per_cpu(__icmp_socket, i));
1115
1116         if (err < 0)
1117             panic("Failed to create the ICMP control socket.\n");
1118
1119         per_cpu(__icmp_socket, i)->sk->sk_allocation = GFP_ATOMIC;
1120
1121         /* Enough space for 2 64K ICMP packets, including
1122          * sk_buff struct overhead.
1123          */
1124         per_cpu(__icmp_socket, i)->sk->sk_sndbuf =
1125             (2 * ((64 * 1024) + sizeof(struct sk_buff)));
1126
1127         inet = inet_sk(per_cpu(__icmp_socket, i)->sk);
1128         inet->uc_ttl = -1;
1129         inet->pmtudisc = IP_PMTUDISC_DONT;
1130
1131         /* Unhash it so that IP input processing does not even
1132          * see it, we do not wish this socket to see incoming
1133          * packets.
1134          */
1135             per_cpu(__icmp_socket, i)->sk->sk_prot->unhash(per_cpu(__icmp_socket,
     i)->sk);
```

```
1136     }
1137 }
```

1110 针对每个 CPU 作以下操作。

1113-1117 为每个 CPU 创建一个基于原始流、IPPROTO_ICMP 协议类型的套接口。

1119 设置基于该传输控制块的内存分配方式为 GFP_ATOMIC。

1124-1125 设置传输控制块发送缓冲区的大小。

1127-1129 初始化单播报文的 TTL 值为-1，表示从目的路由缓存项的度量值中获取，默认值为 64。初始化 pmtudisc 为 IP_PMTUDISC_DONT，即不执行路径 MTU 发现，有关路径 MTU 参见 30.8 节。

1135 为了避免 ICMP 套接口接收到报文，因此调用传输层接口上的 unhash()（参见 raw_v4_unhash()），将套接口从原始流套接口散列表 raw_v4_htable 中取下。

14.5　输入处理

ICMP 的输入处理函数是 icmp_rcv()。ICMP 报文到达时，IP 层通过 inet_protos[IPPROTO_ICMP] 找到该函数进行输入处理。进入 icmp_rcv()后，首先对 ICMP 报文中的参数作适当的校验，然后就会根据 ICMP 报文的类型进行不同的处理。

一种类型的 ICMP 报文对应一个 icmp_control 结构，在内核中定义了一个该结构类型的数组 icmp_pointers[NR_ICMP_TYPES+1]用来管理 ICMP 报文，icmp_control 结构如下所示。

```
216 struct icmp_control {
217     int output_entry;    /* Field for increment on output */
218     int input_entry;    /* Field for increment on input */
219     void (*handler)(struct sk_buff *skb);
220     short   error;        /* This ICMP is classed as an error message */
221 };
```

217-218 ICMP 报文的统计值，每输出/输入一个相应类型的 ICMP 报文，output_entry 和 input_entry 递增 1。

219 对输入该类型 ICMP 报文的处理函数。

220 error 值为 1 表示是一个差错 ICMP 报文；为 0 则是一个查询 ICMP 报文。

以下是 icmp_pointers 数组的定义，由于各类型 ICMP 报文的 icmp_control 结构实例定义都大同小异，因此省略一些类型的定义，这不会影响到阅读。其中类型为 ICMP_ECHOREPLY、ICMP_TIMESTAMPREPLY、ICMP_INFO_REQUEST、ICMP_INFO_REPLY 的 ICMP 报文不需要响应；未定义类型为 1、2、6、7 的 ICMP 报文；而类型为 9、10 的 ICMP 报文是新增加的，并不为所有的系统支持，目前只有 Solaris 2.x 支持这两种类型的 ICMP 报文。

```
996  static const struct icmp_control icmp_pointers[NR_ICMP_TYPES + 1] = {
     .... ...
1014     [ICMP_DEST_UNREACH] = {
1015         .output_entry = ICMP_MIB_OUTDESTUNREACHS,
1016         .input_entry = ICMP_MIB_INDESTUNREACHS,
1017         .handler = icmp_unreach,
1018         .error = 1,
1019     },
```

```
1020    [ICMP_SOURCE_QUENCH] = {
1021        .output_entry = ICMP_MIB_OUTSRCQUENCHS,
1022        .input_entry = ICMP_MIB_INSRCQUENCHS,
1023        .handler = icmp_unreach,
1024        .error = 1,
1025    },
1026    [ICMP_REDIRECT] = {
1027        .output_entry = ICMP_MIB_OUTREDIRECTS,
1028        .input_entry = ICMP_MIB_INREDIRECTS,
1029        .handler = icmp_redirect,
1030        .error = 1,
1031    },
        .... ...
1044    [ICMP_ECHO] = {
1045        .output_entry = ICMP_MIB_OUTECHOS,
1046        .input_entry = ICMP_MIB_INECHOS,
1047        .handler = icmp_echo,
1048    },
        .... ...
1061    [ICMP_TIME_EXCEEDED] = {
1062        .output_entry = ICMP_MIB_OUTTIMEEXCDS,
1063        .input_entry = ICMP_MIB_INTIMEEXCDS,
1064        .handler = icmp_unreach,
1065        .error = 1,
1066    },
1067    [ICMP_PARAMETERPROB] = {
1068        .output_entry = ICMP_MIB_OUTPARMPROBS,
1069        .input_entry = ICMP_MIB_INPARMPROBS,
1070        .handler = icmp_unreach,
1071        .error = 1,
1072    },
1073    [ICMP_TIMESTAMP] = {
1074        .output_entry = ICMP_MIB_OUTTIMESTAMPS,
1075        .input_entry = ICMP_MIB_INTIMESTAMPS,
1076        .handler = icmp_timestamp,
1077    },
        .... ...
1093    [ICMP_ADDRESS] = {
1094        .output_entry = ICMP_MIB_OUTADDRMASKS,
1095        .input_entry = ICMP_MIB_INADDRMASKS,
1096        .handler = icmp_address,
1097    },
1098    [ICMP_ADDRESSREPLY] = {
1099        .output_entry = ICMP_MIB_OUTADDRMASKREPS,
1100        .input_entry = ICMP_MIB_INADDRMASKREPS,
1101        .handler = icmp_address_reply,
1102    },
1103 };
```

1014-1019 定义了类型为 ICMP_DEST_UNREACH，目的不可达，ICMP 报文的 icmp_control 结构实例。

1020-1025 定义了类型为 ICMP_SOURCE_QUENCH，源端被关闭，ICMP 报文的 icmp_control 结构实例。

1061-1066 定义了类型为 ICMP_TIME_EXCEEDED，超时，ICMP 报文的 icmp_control

结构实例。

1067-1072 定义了类型为 ICMP_PARAMETERPROB，参数问题，ICMP 报文的 icmp_control 结构实例。

以上类型的处理函数都为 icmp_unreach()。

1026-1031 定了类型为 ICMP_REDIRECT，重定向，ICMP 报文的 icmp_control 结构实例，处理函数为 icmp_redirect()。

1044-1048 定义了类型为 ICMP_ECHO，请求回显，ICMP 报文的 icmp_control 结构实例，处理函数为 icmp_echo()。

1073-1077 定义了类型为 ICMP_TIMESTAMP，时间戳请求，ICMP 报文的 icmp_control 结构实例，处理函数为 icmp_timestamp()。

1093-1097 定义了类型为 ICMP_ADDRESS，地址掩码请求，ICMP 报文的 icmp_control 结构实例，处理函数为 icmp_address()。

1098-1102 定义了类型为 ICMP_ADDRESSREPLY，地址掩码应答，ICMP 报文的 icmp_control 结构实例，处理函数为 icmp_address_reply()。

图 14-2 描述了 ICMP 报文的接收处理的函数调用过程。

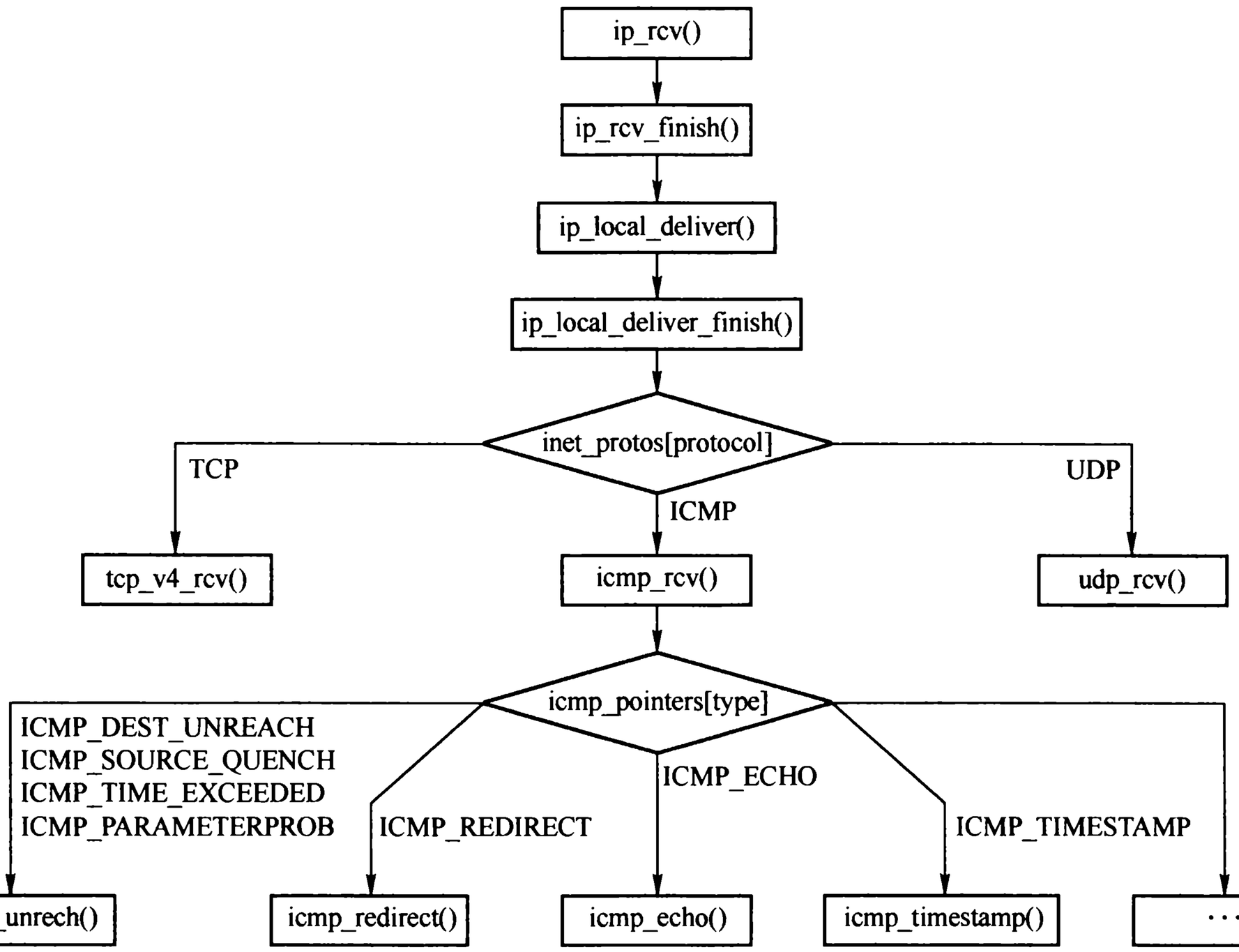

图 14-2　ICMP 报文接收处理

icmp_rcv()的实现如以下代码所示：

```
925 int icmp_rcv(struct sk_buff *skb)
926 {
927     struct icmphdr *icmph;
928     struct rtable *rt = (struct rtable *)skb->dst;
929
```

```
930     ICMP_INC_STATS_BH(ICMP_MIB_INMSGS);
931
932     switch (skb->ip_summed) {
933     case CHECKSUM_COMPLETE:
934        if (!csum_fold(skb->csum))
935           break;
936        /* fall through */
937     case CHECKSUM_NONE:
938        skb->csum = 0;
939        if (__skb_checksum_complete(skb))
940           goto error;
941     }
942
943     if (!pskb_pull(skb, sizeof(struct icmphdr)))
944        goto error;
945
946     icmph = skb->h.icmph;
947
948     /*
949      *    18 is the highest 'known' ICMP type. Anything else is a mystery
950      *
951      *    RFC 1122: 3.2.2  Unknown ICMP messages types MUST be silently
952      *         discarded.
953      */
954     if (icmph->type > NR_ICMP_TYPES)
955         goto error;
```

927-955 处理前先对 ICMP 报文作必要的检测。

932-941 检测 ICMP 报文的校验和。如果此报文已经经过了校验操作，则需要对得到的校验和进行验证。如果由软件来执行校验和，则调用__skb_checksum_complete()进行校验和检测。

943-946 丢弃 ICMP 首部，获得 ICMP 报文内容。如果报文有异常，则跳转到 error 出错处理。

954-955 检测 ICMP 报文的类型，如果其值超过了现有类型的最大值，此 ICMP 报文类型无效，跳转到 error 出错处理。

```
958     /*
959     *    Parse the ICMP message
960     */
961
962     if (rt->rt_flags & (RTCF_BROADCAST | RTCF_MULTICAST)) {
963        /*
964         *    RFC 1122: 3.2.2.6 An ICMP_ECHO to broadcast MAY be
965         *      silently ignored (we let user decide with a sysctl).
966         *    RFC 1122: 3.2.2.8 An ICMP_TIMESTAMP MAY be silently
967         *      discarded if to broadcast/multicast.
968         */
969        if ((icmph->type == ICMP_ECHO ||
970             icmph->type == ICMP_TIMESTAMP) &&
971           sysctl_icmp_echo_ignore_broadcasts) {
972           goto error;
973        }
974        if (icmph->type != ICMP_ECHO &&
975           icmph->type != ICMP_TIMESTAMP &&
976           icmph->type != ICMP_ADDRESS &&
```

```
977             icmph->type != ICMP_ADDRESSREPLY) {
978             goto error;
979         }
980     }
```

958-980 处理通过组播或广播方式发送的 ICMP 报文。

969-973 在开启了 icmp_echo_ignore_broadcasts 的情况下，如果组播或广播报文是请求回显或时间戳请求类型的 ICMP 报文应该被忽略，因此跳转到 error 出错处理。

974-978 除了请求回显、时间戳请求、地址掩码请求和地址掩码应答类型的 ICMP 报文可以接收之外，其他类型的 ICMP 报文都不支持组播或广播。当然前两种类型要在不开启 icmp_echo_ignore_broadcasts 的情况下才能被接收。

```
982     ICMP_INC_STATS_BH(icmp_pointers[icmph->type].input_entry);
983     icmp_pointers[icmph->type].handler(skb);
```

982-983 由 ICMP 报文的类型，在 icmp_pointers 数组中找到该类型的 icmp_control 结构实例，先更新统计值 input_entry，然后调用处理函数处理接收到的 ICMP 报文。

```
985 drop:
986     kfree_skb(skb);
987     return 0;
988 error:
989     ICMP_INC_STATS_BH(ICMP_MIB_INERRORS);
990     goto drop;
991 }
```

985-991 出错处理代码，在处理过程中发现有异常的报文，都会跳转到这里，更新统计值 ICMP_MIB_INERRORS 后丢弃报文。

14.5.1　差错处理

目的不可达、源端被关闭、超时、参数错误这四种类型的差错 ICMP 报文，都是由同一个函数 icmp_unreach()来处理的，对其中目的不可达、源端被关闭着两种类型 ICMP 报文因要提取某些信息而需作一些特殊的处理，而另外一些则不需要，根据差错报文中的信息直接调用传输层的错误处理例程。

```
598 static void icmp_unreach(struct sk_buff *skb)
599 {
600     struct iphdr *iph;
601     struct icmphdr *icmph;
602     int hash, protocol;
603     struct net_protocol *ipprot;
604     struct sock *raw_sk;
605     u32 info = 0;
606
607     /*
608      *    Incomplete header ?
609      *     Only checks for the IP header, there should be an
610      *    additional check for longer headers in upper levels.
611      */
612
```

```
613     if (!pskb_may_pull(skb, sizeof(struct iphdr)))
614         goto out_err;
615
616     icmph = skb->h.icmph;
617     iph   = (struct iphdr *)skb->data;
618
619     if (iph->ihl < 5) /* Mangled header, drop. */
620         goto out_err;
```

613-614 检测 ICMP 报文的 IP 首部是否正常。

616-617 获取 ICMP 报文的 ICMP 首部指针以及导致差错的数据报中的 IP 首部指针，并通过该 IP 首部的首部长度字段来校验该 IP 首部是否正常，ICMP 报文中导致差错的数据报中的 IP 首部应该不少于 20B。

（1）目的不可达处理

图 14-3 是 ICMP 不可达报文的一般格式，其第二个 32 位字必须为 0，但是当代码为 4 时即需要分片但设置了不分片标志，路径 MTU 发现机制借用了其中的低 16 位，允许路由器把出接口的 MTU 填写在此，见图 14-4。

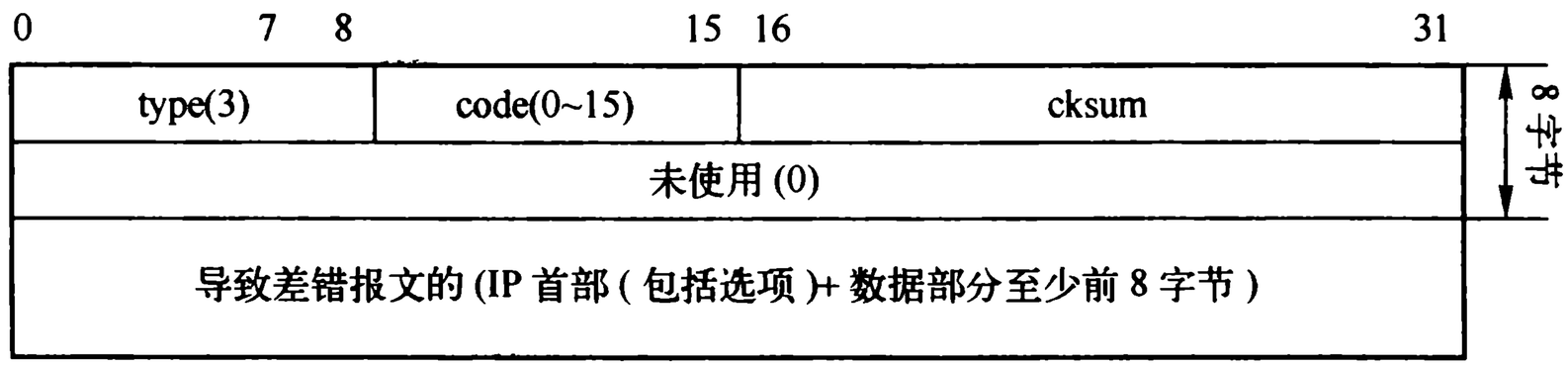

图 14-3 目的不可达差错报文

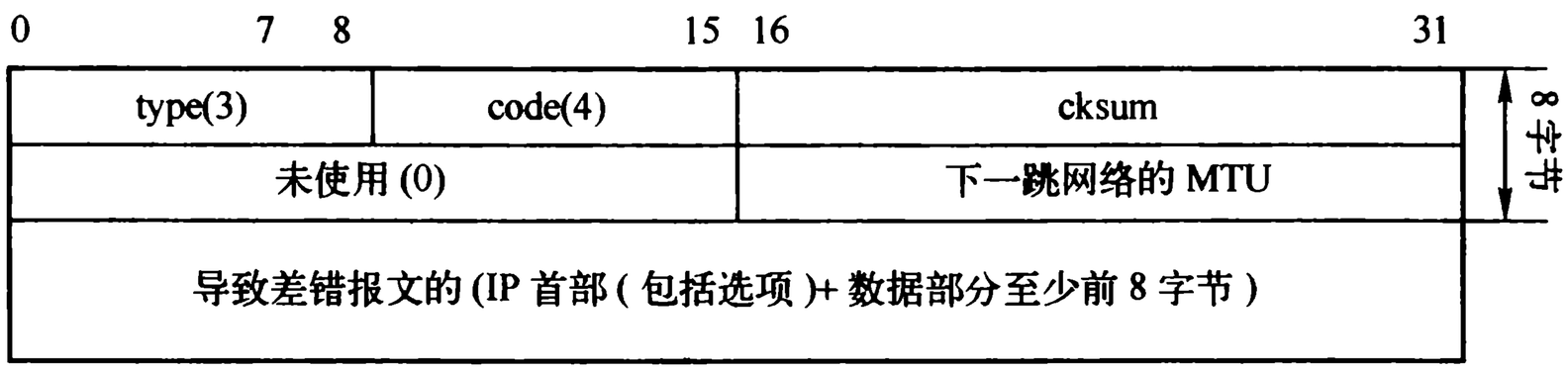

图 14-4 目的不可达需要分片差错报文

```
622     if (icmph->type == ICMP_DEST_UNREACH) {
623         switch (icmph->code & 15) {
624         case ICMP_NET_UNREACH:
625         case ICMP_HOST_UNREACH:
626         case ICMP_PROT_UNREACH:
627         case ICMP_PORT_UNREACH:
628             break;
629         case ICMP_FRAG_NEEDED:
630             if (ipv4_config.no_pmtu_disc) {
631                 LIMIT_NETDEBUG(KERN_INFO "ICMP: %u.%u.%u.%u: "
632                                "fragmentation needed "
633                                "and DF set.\n",
634                                NIPQUAD(iph->daddr));
```

```
635             } else {
636                 info = ip_rt_frag_needed(iph,
637                             ntohs(icmph->un.frag.mtu));
638                 if (!info)
639                     goto out;
640             }
641             break;
642         case ICMP_SR_FAILED:
643             LIMIT_NETDEBUG(KERN_INFO "ICMP: %u.%u.%u.%u: Source "
644                         "Route Failed.\n",
645                        NIPQUAD(iph->daddr));
646             break;
647         default:
648             break;
649         }
650         if (icmph->code > NR_ICMP_UNREACH)
651             goto out;
```

622-623 按其 code 不同分别处理各种目的不可达 ICMP 报文。

624-628 其中网络不可达、主机不可达、协议不可达、端口不可达四种目的不可达 ICMP 报文无需特殊处理。

629-641 处理目的不可达需要分片的差错报文。如果系统禁止使用路径 MTU 发现，则只是打印些信息。否则，则调用 ip_rt_frag_needed()更新路由缓存项并获取有效的 PMTU，参见 20.11 节。

642-646 处理源站选路失败报文，打印相关信息。

650-651 如果目的不可达报文的代码超过最大值，则该报文无效，直接返回。

（2）参数问题处理

图 14-5 是 ICMP 参数问题差错报文格式。

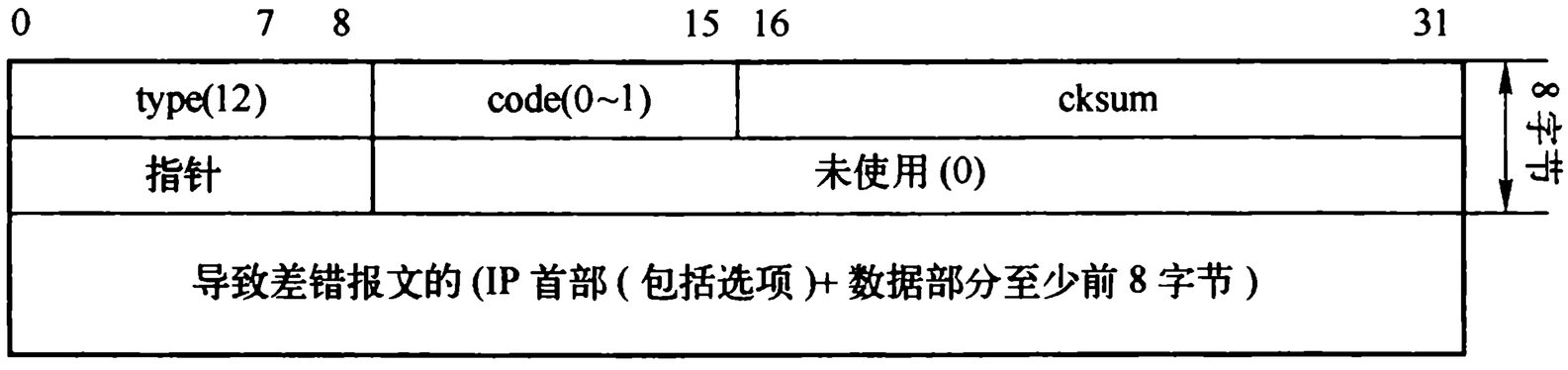

图 14-5　参数问题差错报文

```
652     } else if (icmph->type == ICMP_PARAMETERPROB)
653         info = ntohl(icmph->un.gateway) >> 24;
```

652-653 处理参数问题的差错报文，获取 ICMP 首部中的指针值。从图 14-5 可以看到，指针值存储在 ICMP 报文第 2 个 32 位字的高 8 位，因此获得该值后需右移 24 位。

（3）调用对应传输层错误处理例程

目的不可达需要分片类型和参数问题类型的 ICMP 报文，已经获得了所需的相关错误信息，而其他类型的 ICMP 报文不需要获取错误信息，包括超时和源端被关闭差错报文，见图 14-6 和图 14-7。

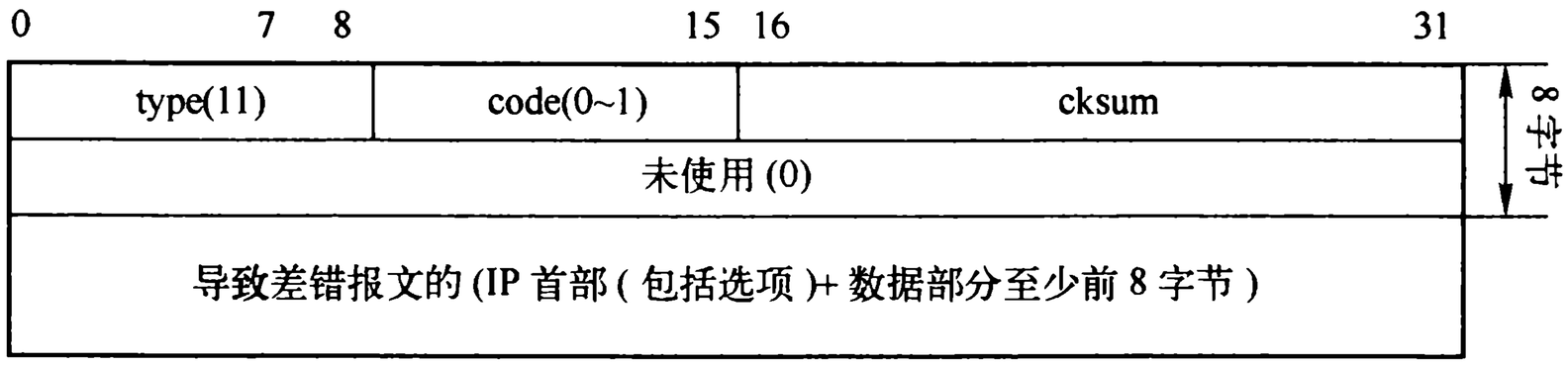

图 14-6　超时差错报文

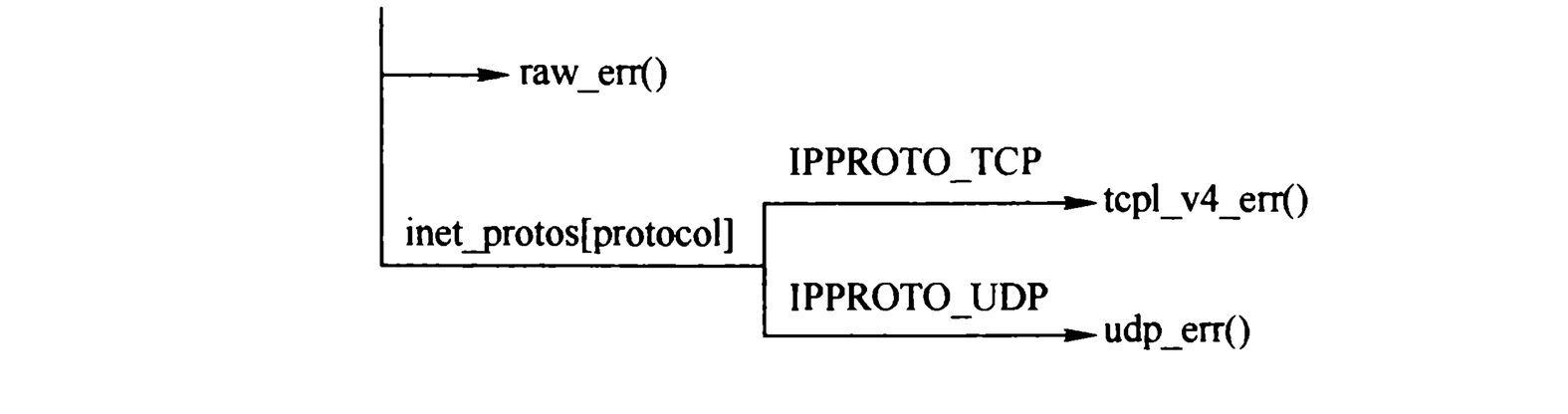

图 14-7　源抑制差错报文

接下来就需要这些错误信息传递给对应的传输层，由传输层作处理，或把这些错误信息进一步传递给应用层。不同的传输层的差错处理也不同，见图 14-8。

图 14-8　调用对应传输层错误处理例程

```
655     /*
656      *    Throw it at our lower layers
657      *
658      *    RFC 1122: 3.2.2 MUST extract the protocol ID from the passed
659      *         header.
660      *    RFC 1122: 3.2.2.1 MUST pass ICMP unreach messages to the
661      *         transport layer.
662      *    RFC 1122: 3.2.2.2 MUST pass ICMP time expired messages to
663      *         transport layer.
664      */
665
666     /*
667      *    Check the other end isnt violating RFC 1122. Some routers send
668      *    bogus responses to broadcast frames. If you see this message
669      *    first check your netmask matches at both ends, if it does then
670      *    get the other vendor to fix their kit.
671      */
672
673     if (!sysctl_icmp_ignore_bogus_error_responses &&
674         inet_addr_type(iph->daddr) == RTN_BROADCAST) {
675         if (net_ratelimit())
676             printk(KERN_WARNING "%u.%u.%u.%u sent an invalid ICMP "
```

```
677                     "type %u, code %u "
678                     "error to a broadcast: %u.%u.%u.%u on %s\n",
679                 NIPQUAD(skb->nh.iph->saddr),
680                 icmph->type, icmph->code,
681                 NIPQUAD(iph->daddr),
682                 skb->dev->name);
683         goto out;
684     }
```

673-684 根据系统参数 icmp_ignore_bogus_error_responses 来确定接收或忽略“目的不可达并且目的 IP 地址为广播地址”这样无效的 ICMP 报文。如果忽略，则在接收到这样的 ICMP 报文后，会记录相应的告警信息。net_ratelimit()是内核打印限速函数，返回 TRUE 时可打印调试信息。

```
686     /* Checkin full IP header plus 8 bytes of protocol to
687      * avoid additional coding at protocol handlers.
688      */
689     if (!pskb_may_pull(skb, iph->ihl * 4 + 8))
690         goto out;
691
692     iph = (struct iphdr *)skb->data;
693     protocol = iph->protocol;
694
695     /*
696      *    Deliver ICMP message to raw sockets. Pretty useless feature?
697      */
698
699     /* Note: See raw.c and net/raw.h, RAWV4_HTABLE_SIZE==MAX_INET_PROTOS */
700     hash = protocol & (MAX_INET_PROTOS - 1);
701     read_lock(&raw_v4_lock);
702     if ((raw_sk = sk_head(&raw_v4_htable[hash])) != NULL) {
703         while ((raw_sk = __raw_v4_lookup(raw_sk, protocol, iph->daddr,
704                                          iph->saddr,
705                                          skb->dev->ifindex)) != NULL) {
706             raw_err(raw_sk, skb, info);
707             raw_sk = sk_next(raw_sk);
708             iph = (struct iphdr *)skb->data;
709         }
710     }
711     read_unlock(&raw_v4_lock);
712
713     rcu_read_lock();
714     ipprot = rcu_dereference(inet_protos[hash]);
715     if (ipprot && ipprot->err_handler)
716         ipprot->err_handler(skb, info);
717     rcu_read_unlock();
718
719 out:
720     return;
721 out_err:
722     ICMP_INC_STATS_BH(ICMP_MIB_INERRORS);
723     goto out;
724 }
```

689-717 调用对应传输层错误处理例程。

689-693 检测 ICMP 报文中导致差错报文的（IP 首部（包括选项）+原始 IP 数据报中数据的前 8 字节）内容长度是否正常。接着获取 ICMP 报文中导致差错报文的 IP 首部和上层协议号。

700-711 通过传输层协议号，首先在 raw_v4_htable 散列表中查找是否有对应的原始套接口的传输控制块，如果有，则调用 raw_err()将差错报文传递上去。由于原始套接口传输控制块的存在，应用程序可以组装各种协议的数据报，如 TCP 段、UDP 数据报等。这里判断不了到底是原始套接口发送的报文导致的差错，还是由 TCP 或 UDP 套接口发送报文导致的差错。因此只能先把差错发给原始套接口（如果存在），然后再发给 TCP 或 UDP 套接口。

713-717 通过传输层协议号，在 inet_protos 数组中找到相应传输层协议的 net_protocol 结构实例，然后调用该实例中定义的传输层差错处理例程，见图 14-8。

14.5.2 重定向处理

当一个路由器发现一个主机使用的是非优化路由时，会发送一个 ICMP 报文给该主机，请求其改变路由，这样的一种 ICMP 报文就是重定向报文。因此这种类型的 ICMP 报文只能由路由器生成，为主机使用。重定向差错报文格式见图 14-9。

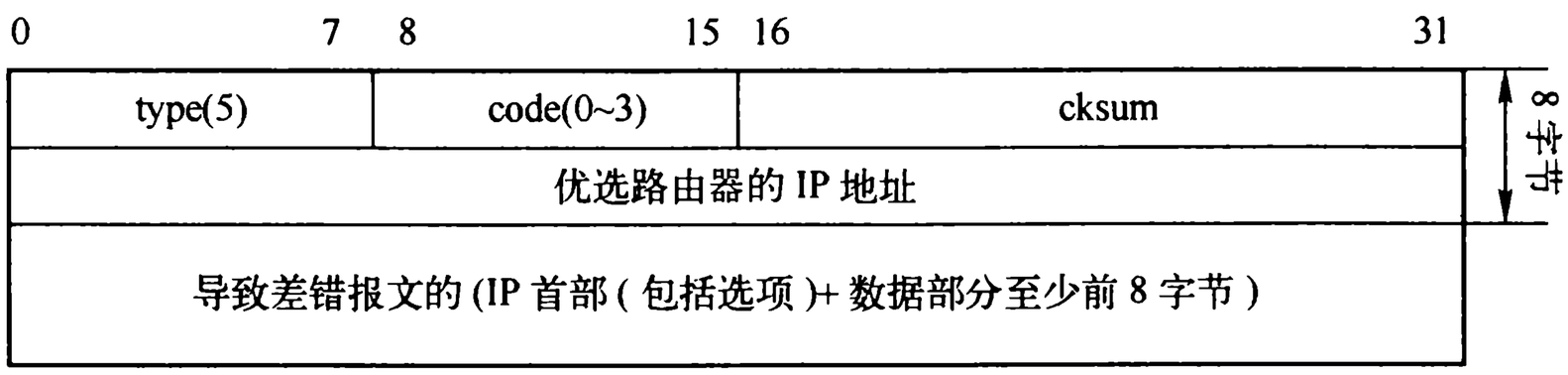

图 14-9 重定向差错报文

在主机处理 ICMP 重定向报文时，涉及三个 IP 地址，非常容易混淆：一是导致重定向的 IP 地址；二是发送重定向报文的路由器 IP 地址；三是优选路由器 IP 地址。这三个 IP 地址在 ICMP 重定向报文的不同部分，对应图 14-10 中的①②③。

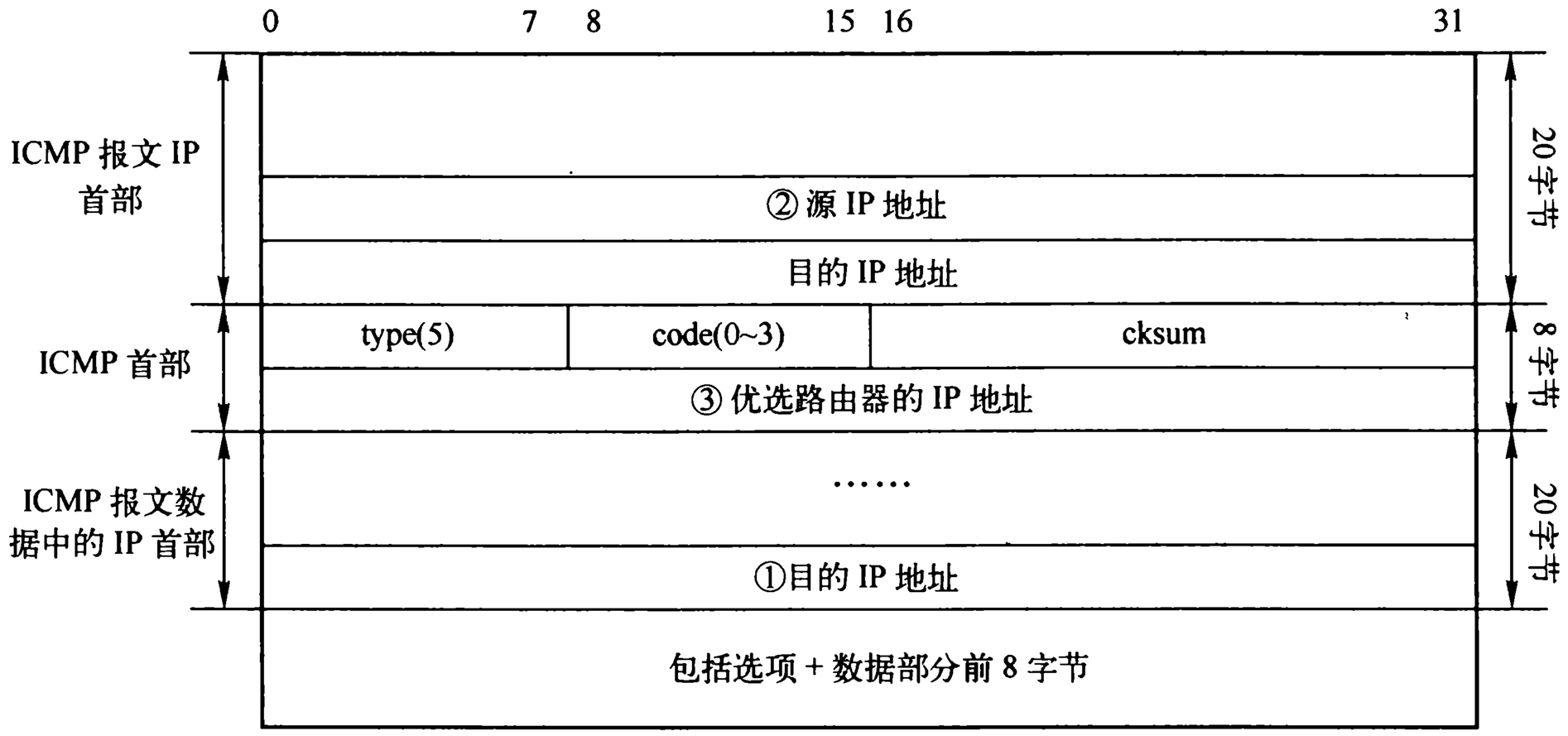

图 14-10 ICMP 重定向报文中涉及的几个 IP 地址

icmp_redirect()用来处理重定向报文。处理过程如下：首先通过报文的长度来检测重定向 ICMP 报文是否正常；然后通过对报文代码的过滤后，使用 ip_rt_redirect()处理正常的重定向

ICMP 报文。

```
731 static void icmp_redirect(struct sk_buff *skb)
732 {
733     struct iphdr *iph;
734
735     if (skb->len < sizeof(struct iphdr))
736         goto out_err;
737
738     /*
739      *    Get the copied header of the packet that caused the redirect
740      */
741     if (!pskb_may_pull(skb, sizeof(struct iphdr)))
742         goto out;
743
744     iph = (struct iphdr *)skb->data;
745
746     switch (skb->h.icmph->code & 7) {
747     case ICMP_REDIR_NET:
748     case ICMP_REDIR_NETTOS:
749         /*
750          * As per RFC recommendations now handle it as a host redirect.
751          */
752     case ICMP_REDIR_HOST:
753     case ICMP_REDIR_HOSTTOS:
754         ip_rt_redirect(skb->nh.iph->saddr, iph->daddr,
755                        skb->h.icmph->un.gateway,
756                        iph->saddr, skb->dev);
757         break;
758       }
759 out:
760     return;
761 out_err:
762     ICMP_INC_STATS_BH(ICMP_MIB_INERRORS);
763     goto out;
764 }
```

14.5.3　请求回显

请求回显和回显应答 ICMP 报文属于查询报文，主要用于测试网络中另一台主机是否可达。向欲测试主机发送一份 ICMP 回显请求，等待其返回 ICMP 回显应答，如果能收到，则表明该主机是可达的，这也是网络工具 ping 程序的实现原理。图 14-11 为请求回显和回显应答报文的格式。

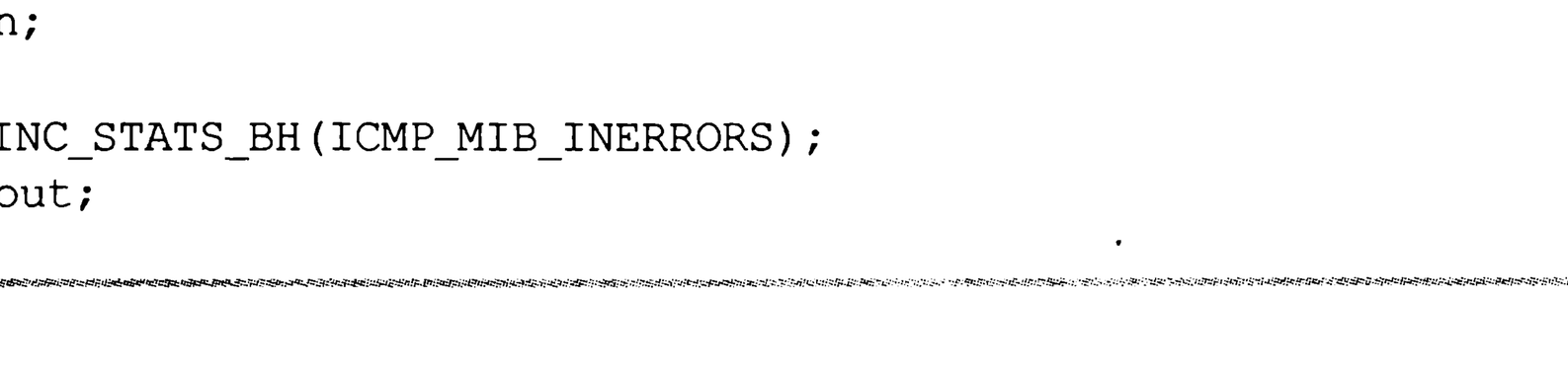

图 14-11　请求回显和回显应答报文

在讨论请求回显报文的处理之前，先说明一个结构体。icmp_bxm 结构是 ICMP 报文的一组信息，在处理 ICMP 报文时经常用此结构作为参数在函数间传递，其定义如下所示。

```
100 struct icmp_bxm {
101     struct sk_buff *skb;
102     int offset;
103     int data_len;
104
105     struct {
106         struct icmphdr icmph;
107         __be32         times[3];
108     } data;
109     int head_len;
110     struct ip_options replyopts;
111     unsigned char  optbuf[40];
112 };
```

101 struct sk_buff *skb

指向 ICMP 报文的 SKB。

102 int offset

当输出 ICMP 差错报文时，offset 为 IP 首部在导致差错报文中的偏移量。

当输出回显应答和时间戳应答报文时，offset 为选项数据在请求回显的 ICMP 报文中的偏移量。

103 int data_len

需要复制到输出 ICMP 报文的数据长度。

105 struct { } data

icmph 为 ICMP 报文的 ICMP 首部；当输出时间戳应答时，times 用来存储相应的时间戳：times[0]从时间戳请求报文中获得，times[1]和 times[2]取当前时间为接收请求和发送应答时间。

109 int head_len

ICMP 首部的长度。

110 struct ip_options replyopts

临时存储引发 ICMP 报文的输入报文的 IP 选项，用于构成待输出 ICMP 应答报文的 IP 选项等操作。

111 unsigned char optbuf[40]

未使用。

icmp_echo()用于处理回显请求报文，输出回显应答报文，如果设置了忽略回显请求报文的系统参数 sysctl_icmp_echo_ignore_all，则直接返回，否则根据请求 ICMP 报文设置 icmp_bxm 结构，然后将该结构传递给 icmp_reply()创建并发送回显应答 ICMP 报文。

```
778 static void icmp_echo(struct sk_buff *skb)
779 {
780     if (!sysctl_icmp_echo_ignore_all) {
781         struct icmp_bxm icmp_param;
782
783         icmp_param.data.icmph      = *skb->h.icmph;
784         icmp_param.data.icmph.type = ICMP_ECHOREPLY;
785         icmp_param.skb             = skb;
```

```
786         icmp_param.offset       = 0;
787         icmp_param.data_len       = skb->len;
788         icmp_param.head_len       = sizeof(struct icmphdr);
789         icmp_reply(&icmp_param, skb);
790     }
791 }
```

14.5.4　时间戳请求

一个主机可以向另一个主机发送一个 ICMP 时间戳请求报文以查询当前的时间，返回的应答值是从午夜开始计算的毫秒数，因此时间的精确度可达到毫秒级，但无法获知当时的日期。时间戳请求和应答报文的格式见图 14-12。

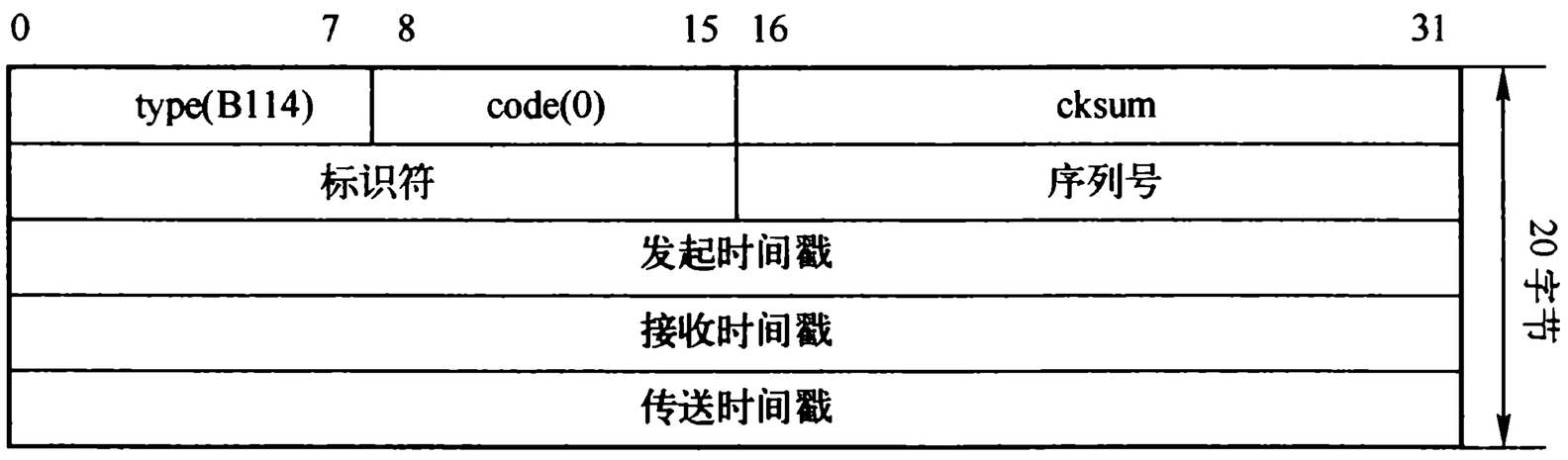

图 14-12　时间戳请求和应答报文

icmp_timestamp()处理时间戳请求报文。调用 do_gettimeofday()获取系统当前的时间作为接收请求和发送应答时间设置到 icmp_bxm 结构的时间戳数组中，然后根据请求报文进一步设置该结构，最后调用 icmp_reply()创建并发送时间戳应答 ICMP 报文。

```
800 static void icmp_timestamp(struct sk_buff *skb)
801 {
802     struct timeval tv;
803     struct icmp_bxm icmp_param;
804     /*
805      *    Too short.
806      */
807     if (skb->len < 4)
808         goto out_err;
809
810     /*
811      *    Fill in the current time as ms since midnight UT:
812      */
813     do_gettimeofday(&tv);
814     icmp_param.data.times[1] = htonl((tv.tv_sec % 86400) * 1000 +
815                 tv.tv_usec / 1000);
816     icmp_param.data.times[2] = icmp_param.data.times[1];
817     if (skb_copy_bits(skb, 0, &icmp_param.data.times[0], 4))
818         BUG();
819     icmp_param.data.icmph      = *skb->h.icmph;
820     icmp_param.data.icmph.type = ICMP_TIMESTAMPREPLY;
821     icmp_param.data.icmph.code = 0;
822     icmp_param.skb         = skb;
823     icmp_param.offset      = 0;
824     icmp_param.data_len      = 0;
```

```
825     icmp_param.head_len       = sizeof(struct icmphdr) + 12;
826     icmp_reply(&icmp_param, skb);
827 out:
828     return;
829 out_err:
830     ICMP_INC_STATS_BH(ICMP_MIB_INERRORS);
831     goto out;
832 }
```

14.5.5 地址掩码请求和应答

ICMP 地址掩码请求用于无盘系统在引导过程中获取自己的子网掩码。地址掩码请求和应答 ICMP 报文的格式见图 14-13。

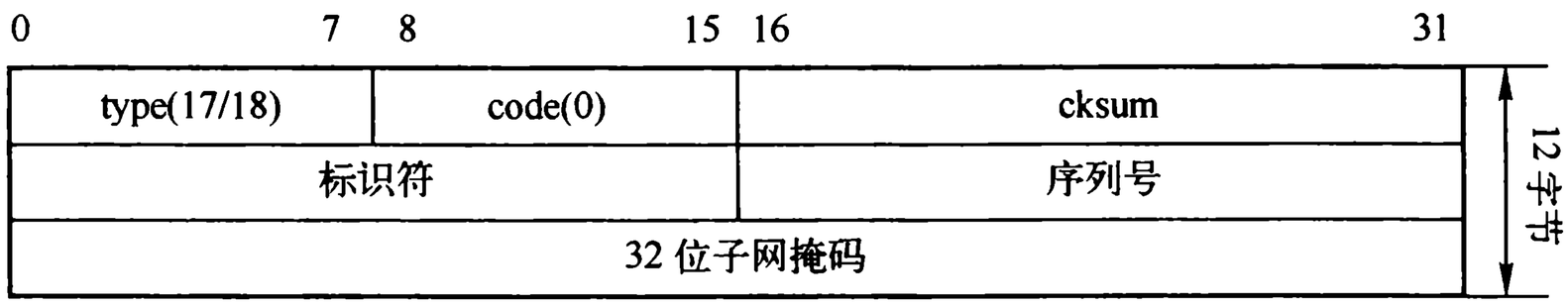

图 14-13 地址掩码请求和应答报文

RFC 规定，除非系统是地址掩码的授权代理，否则必须关闭该功能，不能发送地址掩码应答 ICMP 报文。

14.6 输出处理

14.6.1 发送 ICMP 报文

icmp_send()用于输出各种指定类型和编码的 ICMP 报文，但用该函数不能应答目的地址为组播或广播类型的硬件地址或 IP 地址的报文。参数说明如下：

- skb_in，引发差错的报文，由该报文可获得输入路由缓存，以及作为输出 ICMP 报文数据的原始 IP 首部。
- type，待输出 ICMP 报文的类型。
- code，待输出 ICMP 报文的编码。
- info，ICMP 报文的具体信息，因类型、编码而异，如：对于目的不可达差错报文为下一跳的 MTU，对于重定向差错报文为优选路由器 IP 地址等。

```
433 void icmp_send(struct sk_buff *skb_in, int type, int code, __be32 info)
434 {
435     struct iphdr *iph;
436     int room;
437     struct icmp_bxm icmp_param;
438     struct rtable *rt = (struct rtable *)skb_in->dst;
439     struct ipcm_cookie ipc;
440     __be32 saddr;
441     u8  tos;
442
443     if (!rt)
```

```
444         goto out;
445
446     /*
447      *    Find the original header. It is expected to be valid, of course.
448      *    Check this, icmp_send is called from the most obscure devices
449      *    sometimes.
450      */
451     iph = skb_in->nh.iph;
452
453     if ((u8 *)iph < skb_in->head || (u8 *)(iph + 1) > skb_in->tail)
454         goto out;
455
456     /*
457      *    No replies to physical multicast/broadcast
458      */
459     if (skb_in->pkt_type != PACKET_HOST)
460         goto out;
461
462     /*
463      *    Now check at the protocol level
464      */
465     if (rt->rt_flags & (RTCF_BROADCAST | RTCF_MULTICAST))
466         goto out;
467
468     /*
469      *    Only reply to fragment 0. We byte re-order the constant
470      *    mask for efficiency.
471      */
472     if (iph->frag_off & htons(IP_OFFSET))
473         goto out;
```

438、443 校验待输出 ICMP 报文的输出路由缓存是否有效。

451-454 获得引发差错报文的 IP 首部，并通过检测其长度来判断是否有效。

459-466 函数只能用单播方式来发送 ICMP 报文。

472-473 输出的 ICMP 报文不支持分片，一旦 IP 首部中分片偏移量不为零，即禁止输出。

```
475     /*
476      *    If we send an ICMP error to an ICMP error a mess would result..
477      */
478     if (icmp_pointers[type].error) {
479         /*
480          *    We are an error, check if we are replying to an
481          *    ICMP error
482          */
483         if (iph->protocol == IPPROTO_ICMP) {
484             u8 _inner_type, *itp;
485
486             itp = skb_header_pointer(skb_in,
487                         skb_in->nh.raw +
488                         (iph->ihl << 2) +
489                         offsetof(struct icmphdr,
490                             type) -
491                         skb_in->data,
492                         sizeof(_inner_type),
493                         &_inner_type);
```

```
494             if (itp == NULL)
495                goto out;
496
497             /*
498              *   Assume any unknown ICMP type is an error. This
499              *   isn't specified by the RFC, but think about it..
500              */
501             if (*itp > NR_ICMP_TYPES ||
502                icmp_pointers[*itp].error)
503                goto out;
504         }
505     }
```

483-505 如果输入报文是 ICMP 差错报文，并由此引发输出 ICMP 报文，则需要检测该输入 ICMP 差错报文的类型。

```
507     if (icmp_xmit_lock())
508         return;
```

507-508 icmp_reply()也会发送 ICMP 报文，在此需确保同时只发送一个 ICMP 报文。

```
514     saddr = iph->daddr;
515     if (!(rt->rt_flags & RTCF_LOCAL)) {
516         if (sysctl_icmp_errors_use_inbound_ifaddr)
517             saddr = inet_select_addr(skb_in->dev, 0, RT_SCOPE_LINK);
518         else
519             saddr = 0;
520     }
521
522     tos = icmp_pointers[type].error ? ((iph->tos & IPTOS_TOS_MASK) |
523                     IPTOS_PREC_INTERNETCONTROL) :
524                    iph->tos;
525
526     if (ip_options_echo(&icmp_param.replyopts, skb_in))
527         goto out_unlock;
```

514-520 如果输入报文的目的地址为本机，则将该地址作为输出 ICMP 报文的源地址；否则根据 icmp_errors_use_inbound_ifaddr，获取现有网络设备上首选地址或获取根据引发输出这个 ICMP 错误报文的输入接口的首选地址作为源地址。

522-524 根据输入报文 IP 首部 tos 字段获得输出 ICMP 报文 IP 首部 tos 字段值。

526-527 解析并存储输入报文中的 IP 选项。

```
534     icmp_param.data.icmph.type     = type;
535     icmp_param.data.icmph.code     = code;
536     icmp_param.data.icmph.un.gateway = info;
537     icmp_param.data.icmph.checksum     = 0;
538     icmp_param.skb      = skb_in;
539     icmp_param.offset = skb_in->nh.raw - skb_in->data;
540     icmp_out_count(icmp_param.data.icmph.type);
541     inet_sk(icmp_socket->sk)->tos = tos;
542     ipc.addr = iph->saddr;
543     ipc.opt = &icmp_param.replyopts;
```

534-543 设置待输出 ICMP 报文的类型、编码、附加信息、源地址、IP 选项等。

```
545     {
546         struct flowi fl = {
547             .nl_u = {
548                 .ip4_u = {
549                     .daddr = icmp_param.replyopts.srr ?
550                         icmp_param.replyopts.faddr :
551                         iph->saddr,
552                     .saddr = saddr,
553                     .tos = RT_TOS(tos)
554                 }
555             },
556             .proto = IPPROTO_ICMP,
557             .uli_u = {
558                 .icmpt = {
559                     .type = type,
560                     .code = code
561                 }
562             }
563         };
564         security_skb_classify_flow(skb_in, &fl);
565         if (ip_route_output_key(&rt, &fl))
566             goto out_unlock;
567     }
568
569     if (!icmpv4_xrlim_allow(rt, type, code))
570         goto ende;
```

545-567 根据源地址、TOS、协议以及 ICMP 报文的类型和编码，获取输出路由缓存。

569-570 检测输出 ICMP 报文的类型和编码检，对自定义类型报文、目的不可达需要分片差错报文以及从回环设备上输出的报文始终允许输出，而其他情况，则需通过 icmp_ratelimit() 和 icmp_ratemask()来判断当前能否输出。

```
572     /* RFC says return as much as we can without exceeding 576 bytes. */
573
574     room = dst_mtu(&rt->u.dst);
575     if (room > 576)
576         room = 576;
577     room -= sizeof(struct iphdr) + icmp_param.replyopts.optlen;
578     room -= sizeof(struct icmphdr);
579
580     icmp_param.data_len = skb_in->len - icmp_param.offset;
581     if (icmp_param.data_len > room)
582         icmp_param.data_len = room;
583     icmp_param.head_len = sizeof(struct icmphdr);
584
585     icmp_push_reply(&icmp_param, &ipc, rt);
586 ende:
587     ip_rt_put(rt);
588 out_unlock:
589     icmp_xmit_unlock();
590 out:;
591 }
```

574-590 完成计算待输出 ICMP 报文 ICMP 首部长度，以及 ICMP 报文中原始 IP 报文的长度后，输出该 ICMP 报文。

14.6.2 发送回显应答和时间戳应答报文

（1）icmp_reply()

一般的差错和请求 ICMP 报文是通过 icmp_send()来发送的，而回显应答和时间戳应答报文则是通过 icmp_reply()来输出的。参数说明如下：

- icmp_param，待输出回显应答/时间戳应答 ICMP 报文的 icmp_bxm 结构。
- skb，输入回显请求或时间戳请求报文的 SKB。

```
378 static void icmp_reply(struct icmp_bxm *icmp_param, struct sk_buff *skb)
379 {
380     struct sock *sk = icmp_socket->sk;
381     struct inet_sock *inet = inet_sk(sk);
382     struct ipcm_cookie ipc;
383     struct rtable *rt = (struct rtable *)skb->dst;
384     __be32 daddr;
385
386     if (ip_options_echo(&icmp_param->replyopts, skb))
387         return;
388
389     if (icmp_xmit_lock())
390         return;
391
392     icmp_param->data.icmph.checksum = 0;
393     icmp_out_count(icmp_param->data.icmph.type);
```

383 从输入报文中获得输入路由，用于获取该报文的发送方等信息。

386-387 解析并获取输入报文的 IP 选项到 icmp_param 中。

389-390 需确保同时只发送一个 ICMP 报文，因为 icmp_send()也能发送 ICMP 报文。

392-393 初始化输出 ICMP 报文的校验值为 0，然后更新该类型 ICMP 报文统计计数。

```
395     inet->tos = skb->nh.iph->tos;
396     daddr = ipc.addr = rt->rt_src;
397     ipc.opt = NULL;
398     if (icmp_param->replyopts.optlen) {
399         ipc.opt = &icmp_param->replyopts;
400         if (ipc.opt->srr)
401             daddr = icmp_param->replyopts.faddr;
402     }
403     {
404         struct flowi fl = { .nl_u = { .ip4_u =
405                       { .daddr = daddr,
406                     .saddr = rt->rt_spec_dst,
407                     .tos = RT_TOS(skb->nh.iph->tos) } },
408                 .proto = IPPROTO_ICMP };
409         security_skb_classify_flow(skb, &fl);
410         if (ip_route_output_key(&rt, &fl))
411             goto out_unlock;
412     }
```

395-402 从输入路由中获得发送方地址，作为应答报文的目的地址；如果输入的请求报文中存在 IP 选项，并启用了源站选路，则将源站选路的下一站的 IP 地址作为目的地址。

403-412 根据目的地址、源地址等信息得到待输出 ICMP 报文的路由项。

```
413     if (icmpv4_xrlim_allow(rt, icmp_param->data.icmph.type,
414                            icmp_param->data.icmph.code))
415         icmp_push_reply(icmp_param, &ipc, rt);
416     ip_rt_put(rt);
417 out_unlock:
418     icmp_xmit_unlock();
419 }
```

413-418 检测输出 ICMP 报文的类型和编码检，对自定义类型报文、目的不可达需要分片差错报文以及从回环设备上输出的报文始终允许输出，而其他情况，则需通过 icmp_ratelimit()和 icmp_ratemask()来判断当前能否输出。如果允许输出，则调用 icmp_push_reply()输出该 ICMP 报文。

（2）icmp_push_reply()

此函数用来创建待发送 ICMP 报文，然后将其添加到传输控制块的发送缓冲队列中，完成后，如果套接口的输出队列上还有未输出的报文，则计算 ICMP 报文的校验和并将其输出。参数说明如下：

- icmp_param，待输出 ICMP 报文的 icmp_bxm 结构。
- ipc，待输出 ICMP 报文的 ipcm_cookie 结构，包括目的地址、输出网络设备号以及 IP 选项；
- rt，待输出 ICMP 报文的路由缓存项。

```
347 static void icmp_push_reply(struct icmp_bxm *icmp_param,
348                              struct ipcm_cookie *ipc, struct rtable *rt)
349 {
350     struct sk_buff *skb;
351
352     if (ip_append_data(icmp_socket->sk, icmp_glue_bits, icmp_param,
353                        icmp_param->data_len+icmp_param->head_len,
354                        icmp_param->head_len,
355                        ipc, rt, MSG_DONTWAIT) < 0)
356         ip_flush_pending_frames(icmp_socket->sk);
357     else if ((skb = skb_peek(&icmp_socket->sk->sk_write_queue)) != NULL) {
358         struct icmphdr *icmph = skb->h.icmph;
359         __wsum csum = 0;
360         struct sk_buff *skb1;
361
362         skb_queue_walk(&icmp_socket->sk->sk_write_queue, skb1) {
363             csum = csum_add(csum, skb1->csum);
364         }
365         csum = csum_partial_copy_nocheck((void *)&icmp_param->data,
366                                          (char *)icmph,
367                                          icmp_param->head_len, csum);
368         icmph->checksum = csum_fold(csum);
369         skb->ip_summed = CHECKSUM_NONE;
370         ip_push_pending_frames(icmp_socket->sk);
371     }
```

```
372 }
```

352-356 IP 层接口函数 ip_append_data()，创建待发送 ICMP 报文的 SKB，并将其添加到传输控制块的发送缓冲队列中，参见 11.11.3 节。

如果函数 ip_append_data()处理失败，则调用 ip_flush_pending_frames()释放一些资源，如未输出的 ICMP 报文、临时 IP 选项、路由等。

357-371 如果 ip_append_data()处理成功，并且套接口的输出队列上还有未输出的报文，则计算 ICMP 报文的校验和并将其输出。

第 15 章　IP 组播

IP 组播模块提供了创建和删除转发缓存的基本功能，但只有通过组播路由协议守护进程（通常是 mrouted），依靠路由协议（如静态路由、ospf、rip、bgp、eigrp）来生成转发缓存，才能真正实现组播功能。

因此，IP 组播模块提供用于创建和删除转发缓存和虚拟接口套接口选项，供组播路由协议守护进程来操作。创建转发缓存过程通常如下：当接收到组播报文后，便根据组播报文的源和目的地址查找组播转发缓存，对于第一次的组播报文，通常找不到对应的组播转发缓存，因此为其创建了一个临时的转发缓存，然后给组播路由协议守护进程发送 IGMPMSG_NOCACHE 报告。当组播路由协议守护进程接收到 IGMPMSG_NOCACHE 报告后，便在协议维护的组播路由表里选路，然后通过套接口选项创建新的转发缓存，完成后组播报文便可以转发了。

IP 组播的输入、输出和转发涉及以下文件：

- include/linux/mroute.h，定义 IP 组播相关的虚拟接口结构、组播转发缓存结构等。
- net/ipv4/ipmr.c，IP 组播的输入和转发。
- net/ipv4/ip_output.c，IP 数据报的输出，包括 IP 组播的输出。

15.1　初始化

初始化 IP 组播，主要为 IP 组播建立环境，包括创建组播转发缓存池，为临时路由转发缓存建立定时器等。ip_mr_init()在 inet_init()中被调用来初始化 IP 组播。

```
1899 void __init ip_mr_init(void)
1900 {
1901     mrt_cachep = kmem_cache_create("ip_mrt_cache",
1902                     sizeof(struct mfc_cache),
1903                     0, SLAB_HWCACHE_ALIGN|SLAB_PANIC,
1904                     NULL, NULL);
1905     init_timer(&ipmr_expire_timer);
1906     ipmr_expire_timer.function=ipmr_expire_process;
1907     register_netdevice_notifier(&ip_mr_notifier);
1908 #ifdef CONFIG_PROC_FS
1909     proc_net_fops_create("ip_mr_vif", 0, &ipmr_vif_fops);
1910     proc_net_fops_create("ip_mr_cache", 0, &ipmr_mfc_fops);
1911 #endif
1912 }
```

1901-1904 为组播转发缓存池 mrt_cachep 分配空间，通过 slab 缓存池来分配固定长度的空间可以极大地提高性能并消除碎片。

1905-1906 初始化临时路由转发缓存的定时器。通过定时器，可以删除在规定时间内未添加组播转发缓存对应的临时路由转发缓存，参见 15.4 节。

1907 将 ip_mr_notifier 注册到网络设备通知链中，这样当网络设备状态等发生变化时可以及时地进行响应。

1908-1911 如果编译时设置了支持 proc 文件系统，则在 proc 文件系统中创建 ip_mr_vif 和 ip_mr_cache 结点，之后可以通过这两个文件来查看虚拟接口和组播转发缓存的相关信息。

15.2 虚拟接口

组播报文可以通过两条途径来收发：一是直接通过 LAN 网络设备收发；二是打包成二级单播 IP 数据报，然后通过隧道传输。为避免在整个 IP 组播的实现中区分这两种情况，引入了一个抽象的概念，即虚拟接口（VIF）。

内核中，虚拟接口由 vif_device 结构来描述，由标志 flags 来区分虚拟接口当前描述的是物理网络设备还是 IP-IP 隧道。

```
137 struct vif_device
138 {
139     struct net_device    *dev;          /* Device we are using */
140     unsigned long    bytes_in,bytes_out;
141     unsigned long    pkt_in,pkt_out;       /* Statistics          */
142     unsigned long    rate_limit;       /* Traffic shaping (NI)    */
143     unsigned char    threshold;       /* TTL threshold        */
144     unsigned short    flags;          /* Control flags        */
145     __be32       local,remote;       /* Addresses(remote for tunnels)*/
146     int       link;          /* Physical interface index    */
147 };
```

139 struct net_device *dev

该虚拟接口对应的物理网络设备。

140 unsigned long bytes_in,bytes_out

bytes_in 和 bytes_out 分别是通过该虚拟接口输入和输出的组播报文总字节数。

141 unsigned long pkt_in,pkt_out

pkt_in 和 pkt_out 分别是通过此虚拟接口输入和输出的总组播报文数。

142 unsigned long rate_limit

用于流线限制，目前暂未使用。

143 unsigned char threshold

组播报文 TTL 的阈值，目前暂未使用。

144 unsigned short flags

标识虚拟接口的类型，见表 15-1。

表 15-1 flags 的取值

flags	描述
0	当前采用物理网络设备方式
VIFF_TUNNEL	当前采用 IP-IP 隧道方式
VIFF_SRCRT	未使用
VIFF_REGISTER	在支持 PIM-SM 协议时，创建用于接收 PIM 协议报文的网络设备

145 __be32 local,remote

当采用物理网络设备方式时，local 为网络设备的 IP 地址，remote 则无效。

当采用 IP-IP 隧道方式时，local 为隧道起点地址，remote 为隧道终点地址。

```
146 int link
```

对应物理网络设备的索引。

系统中所有的虚拟网络设备都存储在静态的 vif_table[MAXVIFS]数组中，MAXVIFS 为 32，因此目前最多能描述 32 个虚拟接口，这也就是说与一台组播路由器直连的组播路由器最多为 32 台。

```
84 static struct vif_device vif_table[MAXVIFS]
85 static int maxvif
```

maxvif 是当前所有正在被使用的虚拟接口数组元素下标最大值+1，用于检测虚拟接口的访问有效性。图 15-1 显示了 vif_table 数组和 vif_device 结构的组织方式。

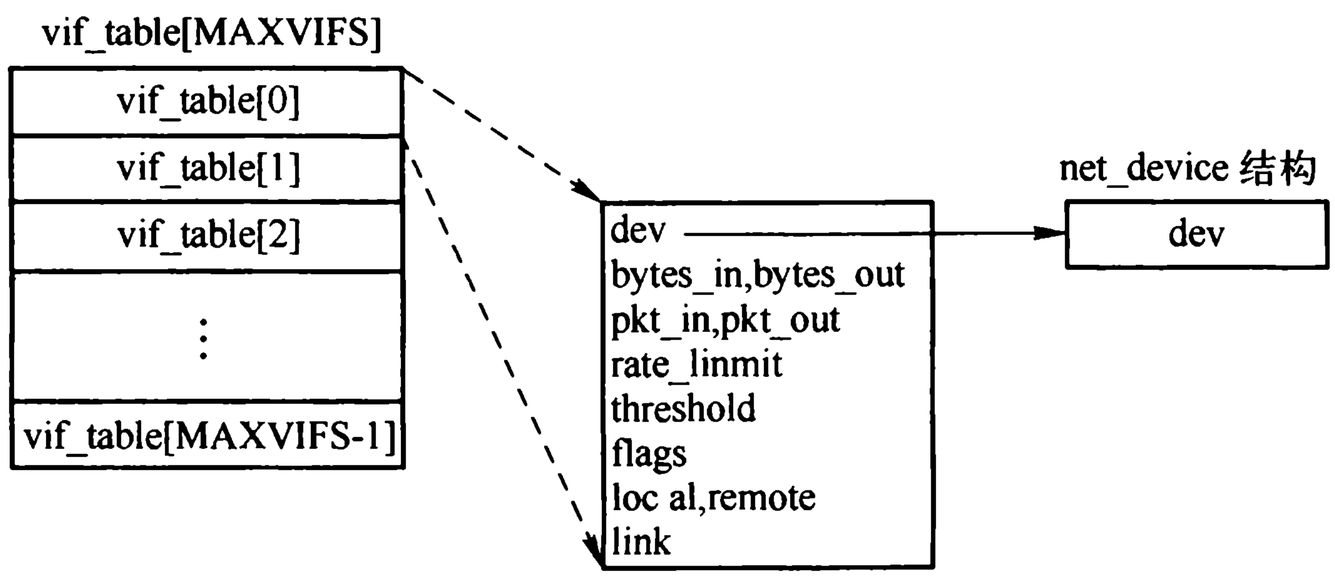

图 15-1　虚拟接口 vif_table 数组

vif_table 数组中的每个虚拟接口都代表着一个物理网络设备或是一条隧道。组播路由守护进程（mrouted）可使用套接口选项 MRT_ADD_VIF 和 MRT_DEL_VIF 来添加和删除数组中的虚拟接口。

15.2.1　虚拟接口的添加

在说明虚拟接口添加和删除之前，先来讨论一个在添加和删除虚拟接口中都会用到的数据结构 vifctl，该结构用来描述虚拟接口的信息。

```
54 struct vifctl {
55     vifi_t    vifc_vifi;       /* Index of VIF */
56     unsigned char vifc_flags;    /* VIFF_ flags */
57     unsigned char vifc_threshold;    /* ttl limit */
58     unsigned int vifc_rate_limit;    /* Rate limiter values (NI) */
59     struct in_addr vifc_lcl_addr;    /* Our address */
60     struct in_addr vifc_rmt_addr;    /* IPIP tunnel addr */
61 };
```

```
55 vifi_t vifc_vifi
```

虚拟接口的索引号，也就是该虚拟接口在 vif_table 数组中的下标。

```
56 unsigned char vifc_flags
```

标识虚拟接口的类型，见表 15-1。

```
57 unsigned char vifc_threshold
```

组播报文 TTL 的阈值，目前未使用。

```
58 unsigned int vifc_rate_limit
```

用于流线限制，目前未使用。

```
59 struct in_addr vifc_lcl_addr
60 struct in_addr vifc_rmt_addr
```

当采用物理网络设备方式时，vifc_lcl_addr 为网络设备的 IP 地址，vifc_rmt_addr 无效。

当采用 IP-IP 隧道方式时，vifc_lcl_addr 为隧道起点地址，vifc_rmt_addr 为隧道终点地址。

添加虚拟接口是由 vif_add()实现的，该函数在以 MRT_ADD_VIF 为 optname 参数值调用 ip_mroute_setsockopt()时被激活。参数说明如下：

- vifc，待添加虚拟接口的信息结构体。
- mrtsock，标识当前配置虚拟接口的是否是组播路由协议守护进程。组播路由协议守护进程不能操作并非由它自己创建的虚拟接口。

```
387 static int vif_add(struct vifctl *vifc, int mrtsock)
388 {
389     int vifi = vifc->vifc_vifi;
390     struct vif_device *v = &vif_table[vifi];
391     struct net_device *dev;
392     struct in_device *in_dev;
393
394     /* Is vif busy ? */
395     if (VIF_EXISTS(vifi))
396         return -EADDRINUSE;
```

获取待设置的虚拟接口，并检查该虚拟接口的网络设备是否可用。

```
398     switch (vifc->vifc_flags) {
399 #ifdef CONFIG_IP_PIMSM
400     case VIFF_REGISTER:
401         /*
402          * Special Purpose VIF in PIM
403          * All the packets will be sent to the daemon
404          */
405         if (reg_vif_num >= 0)
406             return -EADDRINUSE;
407         dev = ipmr_reg_vif();
408         if (!dev)
409             return -ENOBUFS;
410         break;
411 #endif
412     case VIFF_TUNNEL:
413         dev = ipmr_new_tunnel(vifc);
414         if (!dev)
415             return -ENOBUFS;
416         break;
```

```
417     case 0:
418         dev = ip_dev_find(vifc->vifc_lcl_addr.s_addr);
419         if (!dev)
420             return -EADDRNOTAVAIL;
421         dev_put(dev);
422         break;
423     default:
424         return -EINVAL;
425      }
```

399-411 如果编译支持 IP_PIMSM 选项，且添加虚拟接口标志为 VIFF_REGISTER，则调用 ipmr_reg_vif()创建用于接收 PIM 协议报文的网络设备。

412-416 当添加虚拟接口标志为 VIFF_TUNNEL 时，调用 ipmr_new_tunnel()根据配置信息来配置并获取隧道网络设备。

417-422 当添加虚拟接口标志为 0 时，调用 ip_dev_find()根据本地地址获取对应的网络设备，并递增该设备的引用计数。

```
427     if ((in_dev = __in_dev_get_rtnl(dev)) == NULL)
428         return -EADDRNOTAVAIL;
429     in_dev->cnf.mc_forwarding++;
430     dev_set_allmulti(dev, +1);
431     ip_rt_multicast_event(in_dev);
```

427-428 若获取网络设备的 IP 配置块失败，则返回相应的错误码。

429 更新该网络设备上组播转发标志。

430-431 更新网络设备 allmulti 标志，当其值大于 0 时，在该设备上启用接收组播报文功能。接着刷新路由。

```
436     v->rate_limit=vifc->vifc_rate_limit;
437     v->local=vifc->vifc_lcl_addr.s_addr;
438     v->remote=vifc->vifc_rmt_addr.s_addr;
439     v->flags=vifc->vifc_flags;
440     if (!mrtsock)
441         v->flags |= VIFF_STATIC;
442     v->threshold=vifc->vifc_threshold;
443     v->bytes_in = 0;
444     v->bytes_out = 0;
445     v->pkt_in = 0;
446     v->pkt_out = 0;
447     v->link = dev->ifindex;
448     if (v->flags&(VIFF_TUNNEL|VIFF_REGISTER))
449         v->link = dev->iflink;
450
451     /* And finish update writing critical data */
452     write_lock_bh(&mrt_lock);
453     dev_hold(dev);
454     v->dev=dev;
455 #ifdef CONFIG_IP_PIMSM
456     if (v->flags&VIFF_REGISTER)
457         reg_vif_num = vifi;
458 #endif
```

```
459     if (vifi+1 > maxvif)
460         maxvif = vifi+1;
461     write_unlock_bh(&mrt_lock);
462     return 0;
463 }
```

最后将配置信息填充到虚拟接口数据结构中，并更新目前虚拟接口索引的最大值maxvif。

15.2.2 虚拟接口的删除：vif_delete()

vif_delete()实现虚拟接口的删除，此函数在以 MRT_DEL_VIF 为 optname 参数值调用ip_mroute_setsockopt()时被激活。参数 vifi 是待删除虚拟接口的索引。

实现比较简单：首先根据参数给出的索引得到虚拟接口；然后暂存该虚拟接口对应物理网络设备后，将该字段置为 NULL；接着更新 maxvif；更新网络设备的 allmulti 标志，当该值为 0 时，取消该网络设备上的接收组播报文功能；同时更新网络设备的组播转发标志；刷新路由；最后，如果是 VIFF_TUNNEL 或 VIFF_REGISTER 类型的虚拟接口，则还要注销先前创建的网络设备。

15.2.3 查找虚拟接口：ipmr_find_vif()

ipmr_find_vif()根据参数给出的网络设备遍历 vif_table 数组查找对应的虚拟接口，该函数通常在输入或转发组播报文时被调用。

15.3 组播转发缓存

组播转发缓存（Multicast Forwarding Cache，MFC）用于存储组播转发所需的全部信息，例如：组播地址、组播报文发送方地址等。MFC 是实现组播转发的核心结构，实际上就是组播转发路由。

在 Linux 中组播转发缓存由 mfc_cache 结构来描述。

```
151 struct mfc_cache
152 {
153     struct mfc_cache *next;          /* Next entry on cache line    */
154     __be32 mfc_mcastgrp;            /* Group the entry belongs to    */
155     __be32 mfc_origin;              /* Source of packet             */
156     vifi_t mfc_parent;              /* Source interface             */
157     int mfc_flags;                  /* Flags on line          */
158
159     union {
160         struct {
161             unsigned long expires;
162             struct sk_buff_head unresolved;    /* Unresolved buffers       */
163         } unres;
164         struct {
165             unsigned long last_assert;
166             int minvif;
167             int maxvif;
```

```
168             unsigned long bytes;
169             unsigned long pkt;
170             unsigned long wrong_if;
171             unsigned char ttls[MAXVIFS];    /* TTL thresholds       */
172         } res;
173     } mfc_un;
174 };
```

153 struct mfc_cache *next

指向下一个组播转发缓存，用于构成组播转发缓存散列表。

154 __be32 mfc_mcastgrp

155 __be32 mfc_origin

mfc_mcastgrp 是组播报文的组播地址，mfc_origin 是组播报文发送方的 IP 地址，两者结合构成组播转发缓存散列表的键值。

156 vifi_t mfc_parent

虚拟接口在 vif_table 数组中的索引，正是该虚拟接口接收了存储在本组播转发缓存的报文。

157 int mfc_flags

组播转发缓存标志，见表 15-2。

表 15-2　mfc_flags 取值

mfc_flags 取值	描述
MFC_STATIC	静态创建组播转发缓存，不是由 mrouted 进程创建的，因此 mrouted 进程不能操作它
MFC_NOTIFY	通知用户进程的路由缓存，路由已经发生了修改

159 union { } mfc_un

mfc_un 是由 unres 和 res 组成的一个联合。

```
160 struct {
161     unsigned long expires;
162     struct sk_buff_head unresolved;
163 } unres
```

组播路由守护进程 mrouted 仍然没有结束路由选择时，会用 unres 代表组播转发缓存中的缓存项。组播报文一旦到达某个虚拟接口，在组播转发缓存中会为此创建一个 mfc_cache 结构的缓存项。

161-162 路由守护进程选出并设置组播转发缓存的超时时间。

在组播转发过程中，如果找不到该组播报文的转发路由，但能找到对应的输入虚拟接口，这时并不会丢弃报文，而是把报文缓存到 unresolved 队列中，并创建一条临时组播转发缓存，设置超时时间 expires（未起作用），然后再通知守护进程需要创建一条这样的组播转发缓存。如果守护进程未在指定的时间内创建该报文的路由，则会删除这条临时路由以及缓存的报文，参见 15.4 节。

```
164 struct {
165     unsigned long last_assert;
166     int minvif;
```

```
167     int maxvif;
168     unsigned long bytes;
169     unsigned long pkt;
170     unsigned long wrong_if;
171     unsigned char ttls[MAXVIFS];
172 } res
165 unsigned long last_assert
```

记录最近一次发送警告消息的时间，用来控制发送警告消息的频率。在设置了 MRT ASSERT 的情况下，如果发现设置了 MRT PIM 或是对应的转发组播报文的 TTL 阈值小于 255，且离上次发送警告消息的时间已超过阈值，则调用 ipmr_cache_report()给应用程序发送 IGMPMSG_NOCACHE 消息。

```
166 int minvif
167 int maxvif
```

用来限定目前可使用虚拟接口的范围，在此范围内的虚拟接口可以用来发送本组播转发缓存的中的报文。通过 minvif 和 maxvif 指明了索引范围，这样可以节省复制组播报文时的计算时间。当然，maxvif 不能超过 MAXVIFS。

```
168 unsigned long bytes
```

满足此组播转发缓存转发的组播报文的字节数总和，包括在找不到对应输入的虚拟接口的情况下不会被转发的数量。

```
169 unsigned long pkt
```

满足此组播转发缓存转发的组播报文的个数总和，包括在找不到对应输入的虚拟接口的情况下不会被转发的数量。

```
170 unsigned long wrong_if
```

在组播转发过程中，出现能找到组播报文转发路由，但找不到其对应输入虚拟接口的次数。

```
171 unsigned char ttls[MAXVIFS]
```

用来确定是否用 vif_table 数组中下标相同的那个虚拟接口来转发报文，只有当组播报文的 TTL 值大于等于 ttls 数组元素值时，才能由对应的虚拟接口转发报文。通过这种方法，构成了转发组播报文的 TTL 阈值。

所有的组播转发缓存 mfc_cache 结构实例都链接在静态的散列表 mfc_cache_array 中：

```
#define MFC_LINES       64
static struct mfc_cache *mfc_cache_array[MFC_LINES];
```

从 mfc_cache_array 数组和 mfc_cache 结构可以知道，MFC 由散列表构成，所有组播转发缓存形成大小为 MFC_LINES(64)的 MFC 散列表。键值由组播地址和组播报文发送方地址计算得到，算法如下（其中 a 为组播地址，b 为组播报文发送方地址）：

```
#define MFC_HASH(a,b)    ((((__force u32)(__be32)a)^(((__force
u32)(__be32)b)>>2))&(MFC_LINES-1))
```

组播转发缓存的结构如图 15-2 所示。

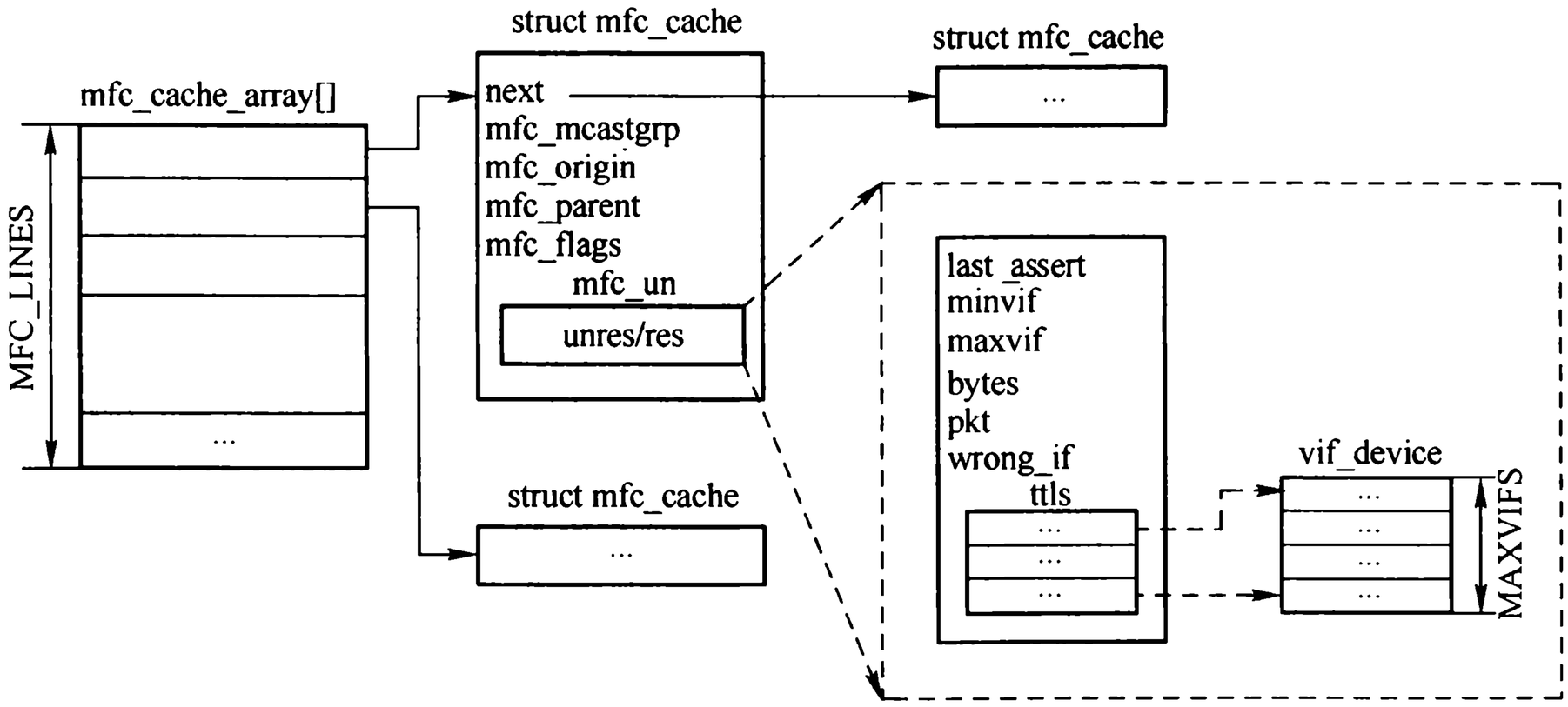

图 15-2　组播转发缓存的结构

15.3.1　组播转发缓存的创建

添加虚拟接口由 ipmr_mfc_add()实现，该函数在以 MRT_ADD_MFC 为 optname 参数调用 ip_mroute_setsockopt()时被激活。函数原型为

static int ipmr_mfc_add(struct mfcctl *mfc, int mrtsock)，

参数说明如下：

- mfc，待创建组播转发缓存的信息，由 mfcctl 结构描述。该结构的字段与 mfc_cache 结构的字段是对应的，在创建组播转发缓存时，将根据该结构来设置 mfc_cache，在此不再对该结构展开。mfc_cache 结构参见 15.3 节。
- mrtsock，表示进行配置的是不是组播路由协议守护进程。不是由组播路由协议守护进程进程创建的，组播路由协议守护进程不能对它进行操作。

创建组播转发缓存的逻辑很简单，首先根据组播源和组播地址在 mfc_cache_array 散列表中查找，如果存在有对应的组播转发缓存项，便更新该缓存项的 TTL 阈值，否则创建新的缓存项并添加到 mfc_cache_array 散列表中；然后在临时组播转发缓存队列 mfc_unres_queue 中查找是否存在相同的缓存项，如果存在，则转发缓存在临时组播转发缓存中的组播报文，释放该临时组播转发缓存。

15.3.2　组播转发缓存的删除

ipmr_mfc_delete()实现虚拟接口添加，该函数在以 MRT_DEL_MFC 为 optname 参数调用 ip_mroute_setsockopt()时被激活，其原型为：

static int ipmr_mfc_delete(struct mfcctl *mfc)

15.3.3　组播转发缓存的查找

在转发组播报文时，会根据组播报文的源地址和组播地址在组播转发缓存散列表中查找对应的组播转发缓存项，只有能找到才可以进入组播转发的下一个环节，否则至少暂时不能转发。

ipmr_cache_find()用来在查找转发组播报文的组播转发缓存项，原理比较简单，通过源地址和组播地址得到散列表的键值，然后再在该键值对应的链表上根据组播源和组播地址进行查找。

15.3.4 向组播路由守护进程发送报告

报告的结构如图 15-3 所示，从图中可以看出，IGMPMSG_NOCACHE 和 IGMPMSG_WRONGVIF 类型报告的首部为 IGMP 协议，负载为 igmpmsg 结构；而 IGMPMSG_WHOLEPKT 类型报告的首部为 igmpmsg 结构，负载部分为完整的 IP 数据报文，这可能是 PIM 报文也可能是组播报文。

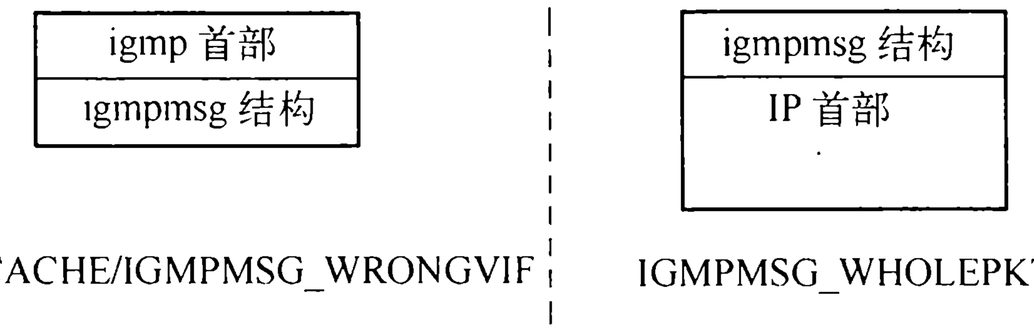

图 15-3 报告格式

igmpmsg 结构定义如下，为了能与 IP 首部对应起来，其长度也是 20B，但有些成员暂时还未使用。

```
114 struct igmpmsg
115 {
116     __u32 unused1,unused2;
117     unsigned char im_msgtype;       /* What is this */
118     unsigned char im_mbz;           /* Must be zero */
119     unsigned char im_vif;           /* Interface (this ought to be a vifi_t!)*/
120     unsigned char unused3;
121     struct in_addr im_src,im_dst;
122 };
```

117 unsigned char im_msgtype

标识报告的类型，见表 15-3。

表 15-3 im_msgtype

im_msgtype	描述
IGMPMSG_NOCACHE	在转发组播路由时，找不到对应的组播转发缓存项，但又存在对应输入的虚拟接口，此时会发送 IGMPMSG_NOCACHE 类型的报告给路由守护进程，参见 15.3 节
IGMPMSG_WRONGVIF	如果收到的组播报文的输入设备不在虚拟接口列表中，通常情况下，IGMPMSG_WRONGVIF 类型的报告用于在 PIM-SM 组播路由器之间选择最短路径或者触发一个 PIM 声明（assert）消息，参见 15.3 节
IGMPMSG_WHOLEPKT	如果在支持 PIM V2 时，路由守护进程添加了 VIFF_REGISTER 标志的虚拟接口，则所有 PIM V2 报文和组播报文都会以 IGMPMSG_WHOLEPKT 类型的报告发送给路由守护进程，由路由守护进程处理，参见 15.3 节

119 unsigned char im_vif

触发报告的 IP 数据报的输入虚拟接口索引号。

121 struct in_addr im_src,im_dst

触发报告的 IP 数据报首部中的源地址和目的地址。

向组播路由守护进程发出有关组播转发路由缓存的报告由 ipmr_cache_report()完成，组播路由守护进程接收到这样的报告之后，可以根据标志进行相应的处理。该函数的参数说明如下：

- pkt，当前接收到的组播报文。

- vifi，组播报文接收的虚拟接口。
- assert，向路由守护进程报告的类型，见表 15-3。

```
542 static int ipmr_cache_report(struct sk_buff *pkt, vifi_t vifi, int assert)
543 {
544     struct sk_buff *skb;
545     int ihl = pkt->nh.iph->ihl<<2;
546     struct igmphdr *igmp;
547     struct igmpmsg *msg;
548     int ret;
549
550 #ifdef CONFIG_IP_PIMSM
551     if (assert == IGMPMSG_WHOLEPKT)
552         skb = skb_realloc_headroom(pkt, sizeof(struct iphdr));
553     else
554 #endif
555         skb = alloc_skb(128, GFP_ATOMIC);
556
557     if(!skb)
558         return -ENOBUFS;
```

550-558 为生成报告分配 SKB。

550-554 如果是 IGMPMSG_WHOLEPKT 的报告，因为需要将接收到的报文数据打包后发给路由守护进程，因此调用 skb_realloc_headroom()复制接收到的原报文，然后在新报文的 SKB 的 headroom 部分留出一个 IP 首部长的空间。

555 对于 IGMPMSG_NOCACHE 和 IGMPMSG_WRONGVIF 类型的报告，则直接分配 128B 的 SKB 就足够了。

557-558 如果分配失败，则返回 ENOBUFS 错误码。

```
560 #ifdef CONFIG_IP_PIMSM
561     if (assert == IGMPMSG_WHOLEPKT) {
562         /* Ugly, but we have no choice with this interface.
563           Duplicate old header, fix ihl, length etc.
564           And all this only to mangle msg->im_msgtype and
565           to set msg->im_mbz to "mbz" :-)
566          */
567         msg = (struct igmpmsg*)skb_push(skb, sizeof(struct iphdr));
568         skb->nh.raw = skb->h.raw = (u8*)msg;
569         memcpy(msg, pkt->nh.raw, sizeof(struct iphdr));
570         msg->im_msgtype = IGMPMSG_WHOLEPKT;
571         msg->im_mbz = 0;
572          msg->im_vif = reg_vif_num;
573         skb->nh.iph->ihl = sizeof(struct iphdr) >> 2;
574         skb->nh.iph->tot_len = htons(ntohs(pkt->nh.iph->tot_len) + sizeof(struct
    iphdr));
575     } else
576 #endif
```

561-574 如果是 IGMPMSG_WHOLEPKT 类型报告，则在 SKB 中将数据部分往头部推一个 IP 首部长度，即将原来属于头部的一个 IP 首部长度的空间计入数据部分。并设置报告首部中 igmpmsg 结构的各成员，包括报告类型、输入虚拟接口等。

```
577     {
583     skb->nh.iph = (struct iphdr *)skb_put(skb, ihl);
584     memcpy(skb->data,pkt->data,ihl);
585     skb->nh.iph->protocol = 0;   /*Flag to the kernel this is a route add */
586     msg = (struct igmpmsg*)skb->nh.iph;
587     msg->im_vif = vifi;
588     skb->dst = dst_clone(pkt->dst);
589
590     /*
591      *    Add our header
592      */
593
594     igmp=(struct igmphdr *)skb_put(skb,sizeof(struct igmphdr));
595     igmp->type    =
596     msg->im_msgtype = assert;
597     igmp->code    =    0;
598     skb->nh.iph->tot_len=htons(skb->len);          /* Fix the length */
599     skb->h.raw = skb->nh.raw;
600         }
```

583-588 如果是 IGMPMSG_NOCACHE/IGMPMSG_WRONGVIF 类型报告，则将触发报告报文的 IP 首部复制到 SKB 中。

594-599 添加报告的 IGMP 结构类型的首部，并设置首部中的值。

```
602     if (mroute_socket == NULL) {
603        kfree_skb(skb);
604        return -EINVAL;
605     }
606
607     /*
608      *    Deliver to mrouted
609      */
610     if ((ret=sock_queue_rcv_skb(mroute_socket,skb))<0) {
611        if (net_ratelimit())
612           printk(KERN_WARNING "mroute: pending queue full, dropping entries.\n");
613        kfree_skb(skb);
614     }
615
616     return ret;
617 }
```

602-605 如果没有设置组播路由套接口，则释放刚才创建的报告。

610-616 否则，则调用 sock_queue_rcv_skb()将报告发送给路由守护进程。

15.4 临时组播转发缓存

目前内核已完全支持 IGMPv3 协议，而 IGMPv3 协议最重要的特征是增加了对组播源的过滤。因此在添加组播转发路由项时需要包括源地址和组播地址，而在转发组播报文时查找对应的组播转发路由缓存项也是通过源地址和组播地址得到的。

尽管路由守护进程会添加一些特定的组播路由缓存项，但有些缓存项是在转发过程中得到的。如果在转发组播报文过程中没有找到对应的缓存项，却又存在该组播报文的虚拟接口，这

时，便会调用 ipmr_cache_unresolved()在 mfc_unres_queue 队列中创建一条临时的组播转发路由缓存项，然后再调用 ipmr_cache_report()向组播路由守护进程发送报告，让路由守护进程添加相应的组播路由，最后激活定时器。当定时器超时，如果组播路由守护进程还未能判定并添加路由，则定时器处理函数会将该临时组播转发路由缓存项删除。

15.4.1　临时组播转发缓存队列

临时组播转发缓存项都存储在单向链表 mfc_unres_queue 中，而 cache_resolve_queue_len 则是该链表的当前长度，用于控制“删除临时组播转发缓存的定时器”的操作，同时也用来限制临时组播转发缓存队列的长度，目前其上限值为 10，当长度达到该上限时，即使还有未确定的组播转发缓存项也不会再创建临时组播转发缓存项了。

```
94 static struct mfc_cache *mfc_unres_queue; /*Queue of unresolved entries*/
95 static atomic_t cache_resolve_queue_len;        /* Size of unresolved    */
```

15.4.2　创建临时组播转发缓存

ipmr_cache_unresolved()完成四件事：一是创建临时组播转发缓存项；二是向组播路由守护进程发送 IGMPMSG_NOCACHE 类型报告；三是启动定时器；四是缓存该组播报文到对应的临时组播转发缓存项中，但一个临时组播转发缓存项最多存 3 个报文。函数参数说明如下：

- vifi，输入该转发组播报文的网络设备索引号。
- skb，查找不到组播转发缓存的组播报文。

```
623 static int
624 ipmr_cache_unresolved(vifi_t vifi, struct sk_buff *skb)
625 {
626     int err;
627     struct mfc_cache *c;
628
629     spin_lock_bh(&mfc_unres_lock);
630     for (c=mfc_unres_queue; c; c=c->next) {
631         if (c->mfc_mcastgrp == skb->nh.iph->daddr &&
632             c->mfc_origin == skb->nh.iph->saddr)
633             break;
634     }
```

630-634 根据组播报文的源地址和组播地址，在临时组播转发缓存队列 mfc_unres_queue 中查找是否已存在对应的临时组播转发缓存项，如果存在，则只需将该组播报文缓存到找到的临时组播转发缓存项中。

```
636     if (c == NULL) {
637         /*
638          *    Create a new entry if allowable
639          */
640
641         if (atomic_read(&cache_resolve_queue_len)>=10 ||
642             (c=ipmr_cache_alloc_unres())==NULL) {
643             spin_unlock_bh(&mfc_unres_lock);
644
```

```
645             kfree_skb(skb);
646             return -ENOBUFS;
647         }
648
649         /*
650          *    Fill in the new cache entry
651          */
652         c->mfc_parent=-1;
653         c->mfc_origin=skb->nh.iph->saddr;
654         c->mfc_mcastgrp=skb->nh.iph->daddr;
655
656         /*
657          *    Reflect first query at mrouted.
658          */
659         if ((err = ipmr_cache_report(skb, vifi, IGMPMSG_NOCACHE))<0) {
660             /* If the report failed throw the cache entry
661                out - Brad Parker
662              */
663             spin_unlock_bh(&mfc_unres_lock);
664
665             kmem_cache_free(mrt_cachep, c);
666             kfree_skb(skb);
667             return err;
668         }
669
670         atomic_inc(&cache_resolve_queue_len);
671         c->next = mfc_unres_queue;
672         mfc_unres_queue = c;
673
674         mod_timer(&ipmr_expire_timer, c->mfc_un.unres.expires);
675     }
```

636-675 如果找不到对应的临时组播转发缓存，则进行创建，并复位定时器。

641-647 如果临时组播转发缓存队列长度已经达到上限值 10，则不能再创建新的临时组播转发缓存，只能丢弃该组播报文。否则调用 ipmr_cache_alloc_unres()分配一条新的临时组播转发缓存。

652-654 初始化新分配的临时组播转发缓存，包括源地址、组播地址等。

659-668 向组播路由守护进程发送一个 IGMPMSG_NOCACHE 类型的报告，通知组播路由守护进程来处理该未确定组播路由缓存。如果报告发送失败，则释放报告及组播报文。

670-672 将新创建的临时组播转发缓存项添加到 mfc_unres_queue 队列中。

674 重新复位定时器。

```
677     /*
678      *    See if we can append the packet
679      */
680     if (c->mfc_un.unres.unresolved.qlen>3) {
681         kfree_skb(skb);
682         err = -ENOBUFS;
683     } else {
684         skb_queue_tail(&c->mfc_un.unres.unresolved,skb);
685         err = 0;
686     }
687
```

```
688     spin_unlock_bh(&mfc_unres_lock);
689     return err;
690 }
```

680-682 如果组播报文对应的临时组播转发缓存中，缓存组播报文数目已达到上限值 3，则只能丢弃该组播报文。

683-686 否则将该组播报文缓存到临时组播转发缓存项中。

ipmr_cache_alloc_unres()用于在 mrt_cachep 缓存池中分配一个临时组播转发缓存项，并初始化其定时器。

```
490 static struct mfc_cache *ipmr_cache_alloc_unres(void)
491 {
492     struct mfc_cache *c=kmem_cache_alloc(mrt_cachep, GFP_ATOMIC);
493     if(c==NULL)
494         return NULL;
495     memset(c, 0, sizeof(*c));
496     skb_queue_head_init(&c->mfc_un.unres.unresolved);
497     c->mfc_un.unres.expires = jiffies + 10*HZ;
498     return c;
499 }
```

15.4.3　用于超时而删除临时组播转发缓存的定时器

ipmr_cache_unresolved()创建临时组播转发缓存项后，发送 IGMPMSG_NOCACHE 报告给组播路由守护进程，接着启动定时器，如果在该定时器到期时，组播路由守护进程还未处理该临时组播转发缓存项，则定时器处理函数 ipmr_expire_process()会将其从临时组播转发缓存队列中删除。

```
327 static void ipmr_expire_process(unsigned long dummy)
328 {
329     unsigned long now;
330     unsigned long expires;
331     struct mfc_cache *c, **cp;
332
333     if (!spin_trylock(&mfc_unres_lock)) {
334         mod_timer(&ipmr_expire_timer, jiffies+HZ/10);
335         return;
336     }
337
338     if (atomic_read(&cache_resolve_queue_len) == 0)
339         goto out;
```

333-336 如果当前临时组播转发缓存队列正在被访问，则只能重新设置定时器后返回，稍后再处理。

338-339 临时组播转发缓存队列长度为 0，则无需处理。

```
341     now = jiffies;
342     expires = 10*HZ;
343     cp = &mfc_unres_queue;
344
345     while ((c=*cp) != NULL) {
```

```
346         if (time_after(c->mfc_un.unres.expires, now)) {
347             unsigned long interval = c->mfc_un.unres.expires - now;
348             if (interval < expires)
349                 expires = interval;
350             cp = &c->next;
351             continue;
352         }
353
354         *cp = c->next;
355
356         ipmr_destroy_unres(c);
357     }
```

341-357 清除已经超时的临时组播转发缓存项。

343、345 遍历临时组播转发缓存队列。

346-352 获取临时组播转发缓存队列中所有未超时缓存项所剩超时时间的最小值，用于之后设置 ipmr_expire_timer 值。

354-356 如果发现有超时的缓存项，则调用 ipmr_destroy_unres()将其删除并释放。

```
359     if (atomic_read(&cache_resolve_queue_len))
360         mod_timer(&ipmr_expire_timer, jiffies + expires);
361
362 out:
363     spin_unlock(&mfc_unres_lock);
364 }
```

如果对临时组播转发缓存队列处理之后，还存在临时组播转发缓存项，则需重新设置定时器超时时间。

15.4.4 释放临时组播缓存项中保存的临时组播报文

释放一个临时组播路由缓存项时，需先释放该缓存项中保存的待转发组播报文，如果报文是用户进程通过 netlink 方式获取组播路由信息的报文，则还需向用户报告 NLMSG_ERROR 错误。

当用户进程通过 netlink 方式获取组播路由信息时，会根据源地址和组播地址来查询组播路由缓存，如果查找失败也会创建临时组播转发路由缓存，并缓存报文，参见 15.3 节。

```
299 static void ipmr_destroy_unres(struct mfc_cache *c)
300 {
301     struct sk_buff *skb;
302     struct nlmsgerr *e;
303
304     atomic_dec(&cache_resolve_queue_len);
305
306     while((skb=skb_dequeue(&c->mfc_un.unres.unresolved))) {
307         if (skb->nh.iph->version == 0) {
308             struct nlmsghdr *nlh = (struct nlmsghdr *)skb_pull(skb, sizeof(struct
    iphdr));
309             nlh->nlmsg_type = NLMSG_ERROR;
310             nlh->nlmsg_len = NLMSG_LENGTH(sizeof(struct nlmsgerr));
311             skb_trim(skb, nlh->nlmsg_len);
312             e = NLMSG_DATA(nlh);
```

```
313             e->error = -ETIMEDOUT;
314             memset(&e->msg, 0, sizeof(e->msg));
315
316             rtnl_unicast(skb, NETLINK_CB(skb).pid);
317         } else
318             kfree_skb(skb);
319     }
320
321     kmem_cache_free(mrt_cachep, c);
322 }
```

304 递减临时组播转发缓存队列的长度。

306-319 遍历待释放临时组播转发缓存项中保存的组播报文，如果该报文是用户进程通过 netlink 方式获取组播路由信息的报文，则报告 NLMSG_ERROR 错误；否则释放该组播报文。

322 最后释放该临时组播转发缓存项。

15.5　外部事件

当一个网络设备的状态发生变化时，IP 组播模块通过注册到通知链中的 ip_mr_notifier 收到通知，然后调用 ipmr_device_event()来处理该事件。ipmr_device_event()只关心 NETDEV_UNREGISTER 事件，遍历 vif_table 数组，删除与该网络设备相关的虚拟接口。

```
1075 static int ipmr_device_event(struct notifier_block *this, unsigned long event,
     void *ptr)
1076 {
1077     struct vif_device *v;
1078     int ct;
1079     if (event != NETDEV_UNREGISTER)
1080         return NOTIFY_DONE;
1081     v=&vif_table[0];
1082     for(ct=0;ct<maxvif;ct++,v++) {
1083         if (v->dev==ptr)
1084             vif_delete(ct);
1085     }
1086     return NOTIFY_DONE;
1087 }
```

15.6　组播套接口选项

15.6.1　IP_MULTICAST_TTL

IP_MULTICAST_TTL 选项用于获取和设置输出组播报文的 TTL。组播报文 TTL 可设定成从 0 到 255 之间的任何值，默认值为 1。设置代码如下所示：

```
unsigned char ttl;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL, &ttl, sizeof(ttl));
```

15.6.2 IP_MULTICAST_LOOP

IP_MULTICAST_LOOP 选项用于开启和禁用组播回环：0 为禁止；1 为允许。

```
unsigned char loop;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_LOOP, &loop, sizeof(loop));
```

15.6.3 IP_MULTICAST_IF

IP_MULTICAST_IF 选项可为发送组播设置默认的网络设备接口。在具有多个网络设备接口的主机上，应用程序可能希望使用系统默认值之外的接口。

```
struct in_addr addr;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_IF, &addr, sizeof(addr));
```

addr 是希望使用的输出接口本地 IP 地址，通过该地址可以找到对应的网络设备。当地址为 INADDR_ANY 时，表示根据目的地址和端口在路由表中查找合适的输出网络设备。

此外还可以通过 ip_mreqn 结构来进行设置。

```
struct ip_mreqn mreqn;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_IF, &mreqn, sizeof(mreqn));
```

相比通过 in_addr 结构来设置 IP_MULTICAST_IF 更为灵活的一点是，通过 imr_ifindex 成员在应用层可以指定组播报文输出的网络设备接口，当然也可以不设。结构中的 imr_address 成员为希望输出接口的本地 IP 地址，成员 imr_multiaddr 在设置过程中忽略。ip_mreqn 结构如下所示：

```
120 struct ip_mreqn
121 {
122     struct in_addr    imr_multiaddr;        /*IP multicast address of group */
123     struct in_addr    imr_address;        /* local IP address of interface */
124     int       imr_ifindex;        /* Interface index */
125 };
```

实现 IP_MULTICAST_IF 选项的代码如下。

```
565         case IP_MULTICAST_IF:
566         {
567             struct ip_mreqn mreq;
568             struct net_device *dev = NULL;
569
570             if (sk->sk_type == SOCK_STREAM)
571                 goto e_inval;
572             /*
573              *    Check the arguments are allowable
574              */
575
576             err = -EFAULT;
577             if (optlen >= sizeof(struct ip_mreqn)) {
578                 if (copy_from_user(&mreq,optval,sizeof(mreq)))
579                     break;
```

```
580             } else {
581                memset(&mreq, 0, sizeof(mreq));
582                if (optlen >= sizeof(struct in_addr) &&
583                   copy_from_user(&mreq.imr_address,optval,sizeof(struct in_addr)))
584                   break;
585             }
586
587             if (!mreq.imr_ifindex) {
588                if (mreq.imr_address.s_addr == INADDR_ANY) {
589                   inet->mc_index = 0;
590                   inet->mc_addr  = 0;
591                   err = 0;
592                   break;
593                }
594                dev = ip_dev_find(mreq.imr_address.s_addr);
595                if (dev) {
596                   mreq.imr_ifindex = dev->ifindex;
597                   dev_put(dev);
598                }
599             } else
600                dev = __dev_get_by_index(mreq.imr_ifindex);
601
602
603             err = -EADDRNOTAVAIL;
604             if (!dev)
605                break;
606
607             err = -EINVAL;
608             if (sk->sk_bound_dev_if &&
609                mreq.imr_ifindex != sk->sk_bound_dev_if)
610                break;
611
612             inet->mc_index = mreq.imr_ifindex;
613             inet->mc_addr  = mreq.imr_address.s_addr;
614             err = 0;
615             break;
616          }
```

570-571 校验设置 IP_MULTICAST_IF 套接口的类型。组播不支持流类型套接口，如果设置的是流类型套接口，则返回 EINVAL 错误码。

576-585 从用户空间复制设置的值。

576 先设置错误码值为 EFAULT，然后进行选项值的复制，一旦复制失败，直接返回即可。

577-579 如果 optlen 参数给出的 IP_MULTICAST_IF 选项值长度达到 ip_mreqn 结构的长度，则按 ip_mreqn 结构从用户空间复制数据到内核空间。

580-585 否则只复制希望的输出接口本地 IP 地址。

587-600 获取输出的网络设备。

587 如果应用程序没有指定输出设备，则需根据设置的 IP 地址获得输出网络设备。

588-593 如果希望的输出接口 IP 地址为 INADDR_ANY，则在输出组播报文时，根据目的地址和端口在路由表中查找合适的输出网络设备。因此，把传输控制块中的组播输出设备和组播源地址都设置为 0 即可。

594-598 如果设置了的希望输出接口地址，则调用 ip_dev_find()根据地址查找对应的输出

设备，并获取索引号。

599-600 如果应用程序指定了输出网络设备索引号，则根据索引号获取对应的网络设备。

603-605 如果获取网络设备失败，则返回 EADDRNOTAVAIL 错误。

607-610 如果通过选项 SO_BINDTODEVICE 绑定的输出网络设备与此次设置的组播输出设备不同，则 IP_MULTICAST_IF 设置无效。

612-615 最后设置传输控制块中的组播输出设备和组播源地址后成功返回。

15.6.4 IP_ADD_MEMBERSHIP

通过使用 IP ADD MEMBERSHIP 选项将网络接口和套接口加入到指定的组播组，就可以在指定网络接口上接收属于指定组播组的 IP 组播数据了。

```
struct ip_mreq mreq;
setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq, sizeof(mreq));
```

ip_mreq 为组播组有关信息的结构体，定义如下：

```
114 struct ip_mreq
115 {
116     struct in_addr imr_multiaddr;    /* IP multicast address of group */
117     struct in_addr imr_interface;    /* local IP address of interface */
118 };
```

imn_multiaddr 为待加入组播组的组地址；而 imr_interface 为待加入组播的网络接口地址。

也可以通过 ip_mreqn 结构来进行设置：

```
struct ip_mreqn mreqn;
setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreqn, sizeof(mreqn));
```

与 ip_mreq 结构相比，ip_mreqn 结构多了一个成员 imr_ifindex，应用程序可以通过该字段指定加入组播组的网络设备。

15.6.5 IP_DROP_MEMBERSHIP

IP_DROP_MEMBERSHIP 选项用于在一个指定的本地网络接口上离开一个组播组，设置时除选项值外，其余参数和加入一个组播组的相同。如果本地接口没有指定（即其值为 INADDR_ANY），那么第一个匹配的组播组的组成员关系将被去掉。

```
struct ip_mreq mreq;
setsockopt(sock, IPPROTP_IP, IP_DROP_MEMBERSHIP, &mreq, sizeof(sreq));
```

同样也可以通过 ip_mreqn 结构来设置：

```
struct ip_mreqn mreqn;
setsockopt(sock, IPPROTO_IP, IP_DROP_MEMBERSHIP, &mreqn, sizeof(mreqn));
```

如果一个进程加入一个组后，就算是从未明确地离开那个组，但当套接口关闭时，成员关系也会被自动去掉。同一主机上的多个进程都加入同一个组也是可以的，在这种情况下，该主机一直是那个组的成员，直到最后一个进程离开那个组。

实现 IP_ADD_MEMBERSHIP 和 IP_DROP_MEMBERSHIP 选项的代码如下：

```
618         case IP_ADD_MEMBERSHIP:
619         case IP_DROP_MEMBERSHIP:
620         {
621             struct ip_mreqn mreq;
622
623             if (optlen < sizeof(struct ip_mreq))
624                 goto e_inval;
625             err = -EFAULT;
626             if (optlen >= sizeof(struct ip_mreqn)) {
627                 if(copy_from_user(&mreq,optval,sizeof(mreq)))
628                     break;
629             } else {
630                 memset(&mreq, 0, sizeof(mreq));
631                 if (copy_from_user(&mreq,optval,sizeof(struct ip_mreq)))
632                     break;
633             }
634
635             if (optname == IP_ADD_MEMBERSHIP)
636                 err = ip_mc_join_group(sk, &mreq);
637             else
638                 err = ip_mc_leave_group(sk, &mreq);
639             break;
640         }
```

623-624 校验参数 optlen 所给出的 IP_ADD_MEMBERSHIP 和 IP_DROP_MEMBERSHIP 选项值长度，不能小于 ip_mreq 结构的长度。

625-633 如果长度值大于等于 ip_mreqn 结构的长度，则按 ip_mreqn 结构复制到内核中。否则按 ip_mreq 结构复制。

637-638 加入组调用 ip_mc_join_group()进行处理，离开组则调用 ip_mc_leave_group()进行处理，参见 16.9 节。

15.6.6　IP_MSFILTER

IGMPv3 增加了“源过滤”特性，IP_MSFILTER 选项即用来配置组播源过滤，在指定的网络接口上根据配置的组播源对特定的组播报文进行 EXCLUDE 或 INCLUDE 模式的过滤。

```
struct ip_msfilter msfilter;
setsockopt(sock, IPPROTO_IP, IP_MSFILTER, &msfilter, sizeof(msfilter));
```

ip_msfilter 为配置组播源过滤的相关信息结构，定义如下：

```
133 struct ip_msfilter {
134     __be32          imsf_multiaddr;
135     __be32          imsf_interface;
136     __u32           imsf_fmode;
137     __u32           imsf_numsrc;
138     __be32          imsf_slist[1];
139 };
```

imsf_multiaddr 为源过滤组播地址；

imsf_interface 为源过滤网络设备接口；

imsf_fmode 为过滤模式，MCAST_EXCLUDE/MCAST_INCLUDE；

imsf_numsrc 为过滤源地址的数量；

imsf_slist 为过滤源地址列表。

实现 IP_MSFILTER 选项的代码如下：

```
641         case IP_MSFILTER:
642         {
643             extern int sysctl_igmp_max_msf;
644             struct ip_msfilter *msf;
645
646             if (optlen < IP_MSFILTER_SIZE(0))
647                 goto e_inval;
648             if (optlen > sysctl_optmem_max) {
649                 err = -ENOBUFS;
650                 break;
651             }
652             msf = kmalloc(optlen, GFP_KERNEL);
653             if (msf == 0) {
654                 err = -ENOBUFS;
655                 break;
656             }
657             err = -EFAULT;
658             if (copy_from_user(msf, optval, optlen)) {
659                 kfree(msf);
660                 break;
661             }
662             /* numsrc >= (1G-4) overflow in 32 bits */
663             if (msf->imsf_numsrc >= 0x3ffffffcU ||
664                 msf->imsf_numsrc > sysctl_igmp_max_msf) {
665                 kfree(msf);
666                 err = -ENOBUFS;
667                 break;
668             }
669             if (IP_MSFILTER_SIZE(msf->imsf_numsrc) > optlen) {
670                 kfree(msf);
671                 err = -EINVAL;
672                 break;
673             }
674             err = ip_mc_msfilter(sk, msf, 0);
675             kfree(msf);
676             break;
677         }
```

646-651 校验 IP_MSFILTER 选项值的长度：可以没有过滤的组播源地址，但不能超过辅助缓冲区的上限。

652-661 分配空间并把选项值从用户空间复制到内核空间中。

663-673 校验过滤源地址的数量，既不得超过硬性上限值 0x3ffffffcU，也不得超过配置上限值 igmp_max_msf，同时还不能超过过滤源地址列表中的实际数量。

674-675 最后调用 ip_mc_msfilter()进行配置，完成后释放在内核中分配的配置信息结构。

15.6.7　IP_BLOCK_SOURCE 和 IP_UNBLOCK_SOURCE

IP_BLOCK_SOURCE 选项用于阻塞组中指定的组播源，而 IP_UNBLOCK_SOURCE 选项则开通先前被阻塞的组播源。

```
struct ip_mreq_source mreqs;
setsockopt(sock, IPPROTO_IP, IP_BLOCK_SOURCE, &mreqs, sizeof(mreqs));
setsockopt(sock, IPPROTO_IP, IP_UNBLOCK_SOURCE, &mreqs, sizeof(mreqs));
```

ip_mreq_source 为阻塞/开通组播源的相关信息结构，定义如下：

```
127 struct ip_mreq_source {
128     __be32          imr_multiaddr;
129     __be32          imr_interface;
130     __be32          imr_sourceaddr;
131 };
```

对 IP_BLOCK_SOURCE/IP_UNBLOCK_SOURCE 选项，该结构体的各字段意义如下：

imr_multiaddr 是阻塞或开通组播源的组播组地址；

imr_interface 是阻塞或开通组播源的网络接口地址；

imr_sourceaddr 是阻塞或开通的组播源。

15.6.8　IP_ADD_SOURCE_MEMBERSHIP 和 IP_DROP_SOURCE_MEMBERSHIP

IP_ADD_SOURCE_MEMBERSHIP 选项使网络接口和套接口加入指定的组播组，并且以 INCLUDE 模式配置组播源过滤，而 IP_DROP_SOURCE_MEMBERSHIP 选项用于离开一个组播组。

```
struct ip_mreq_source mreqs;
setsockopt(sock, IPPROTO_IP, IP_ADD_SOURCE_MEMBERSHIP, &mreqs, sizeof(mreqs));
setsockopt(sock, IPPROTO_IP, IP_DROP_SOURCE_MEMBERSHIP, &mreqs, sizeof(mreqs));
```

虽然 IP_ADD_SOURCE_MEMBERSHIP/IP_DROP_SOURCE_MEMBERSHIP 和 IP_BLOCK_SOURCE/IP_UNBLOCK_SOURCE 一样都使用了 ip_mreq_source 结构，但该结构各字段具体的意义是不同的：

对于 IP_ADD_SOURCE_MEMBERSHIP 和 IP_DROP_SOURCE_MEMBERSHIP，意义如下：

imr_multiaddr 为待加入或离开的组播组地址；

imr_interface 为待加入或离开组播组的网络接口地址；

imr_sourceaddr 在 IP_ADD_SOURCE_MEMBERSHIP 选项时为 INCLUDE，而在 IP_DROP_SOURCE_MEMBERSHIP 选项时无效。

实现此四个选项的代码如下：

```
678         case IP_BLOCK_SOURCE:
679         case IP_UNBLOCK_SOURCE:
680         case IP_ADD_SOURCE_MEMBERSHIP:
681         case IP_DROP_SOURCE_MEMBERSHIP:
682         {
683             struct ip_mreq_source mreqs;
```

```
684             int omode, add;
685
686             if (optlen != sizeof(struct ip_mreq_source))
687                goto e_inval;
688             if (copy_from_user(&mreqs, optval, sizeof(mreqs))) {
689                err = -EFAULT;
690                break;
691             }
692             if (optname == IP_BLOCK_SOURCE) {
693                omode = MCAST_EXCLUDE;
694                add = 1;
695             } else if (optname == IP_UNBLOCK_SOURCE) {
696                omode = MCAST_EXCLUDE;
697                add = 0;
698             } else if (optname == IP_ADD_SOURCE_MEMBERSHIP) {
699                struct ip_mreqn mreq;
700
701                mreq.imr_multiaddr.s_addr = mreqs.imr_multiaddr;
702                mreq.imr_address.s_addr = mreqs.imr_interface;
703                mreq.imr_ifindex = 0;
704                err = ip_mc_join_group(sk, &mreq);
705                if (err && err != -EADDRINUSE)
706                   break;
707                omode = MCAST_INCLUDE;
708                add = 1;
709             } else /* IP_DROP_SOURCE_MEMBERSHIP */ {
710                omode = MCAST_INCLUDE;
711                add = 0;
712             }
713             err = ip_mc_source(add, omode, sk, &mreqs, 0);
714             break;
715          }
```

686-691 经过选项值长度校验后，将选项值从用户空间复制到内核空间中。

692-697 当选项为 IP_BLOCK_SOURCE/IP_UNBLOCK_SOURCE 时，把将作为 ip_mc_source()参数的 mode 设置为 MCAST_EXCLUDE 模式，而 add 指示阻塞/开通。

698-708 当选项为 IP_ADD_SOURCE_MEMBERSHIP 时，将套接口和网络设备加入到指定的组，完成后并设置 mode 为 MCAST_INCLUDE 模式，add 为添加。

709-711 当选项为 IP_DROP_SOURCE_MEMBERSHIP 时，将 mode 设置为 MCAST_INCLUDE 模式，add 为离开。

713 最后调用 ip_mc_source()阻塞或开通组播源，参见 16.11 节。

15.6.9 MCAST_JOIN_GROUP

MCAST_JOIN_GROUP 选项与 IP_ADD_MEMBERSHIP 选项相类似，使网络接口和套接口加入指定的组播组，然后就可以在指定网络接口上接收属于特定组播组的 IP 组播数据了。

```
struct group_req groupr;
setsockopt(sock, IPPROTO_IP, MCAST_JOIN_GROUP, &groupr, sizeof(groupr));
```

group_req 为待加入组播组的相关信息结构，定义如下：

```
145 struct group_req
146 {
147     __u32                gr_interface;   /* interface index */
148     struct __kernel_sockaddr_storage gr_group;    /* group address */
149 };
```

gr_interface 为待加入组播的网络接口索引号，gr_group 为待加入组播组的组地址。

15.6.10　MCAST_LEAVE_GROUP

MCAST_LEAVE_GROUP 选项与 IP_DROP_MEMBERSHIP 选项类似，用于在一个指定的本地网络接口上离开一个组播组，设置时和 MCAST_JOIN_GROUP 选项一样用 group_req 结构。

```
struct ip_mreq mreq;
setsockopt(sock, IPPROTP_IP, IP_DROP_MEMBERSHIP, &mreq, sizeof(sreq));
```

MCAST_JOIN_GROUP 和 MCAST_LEAVE_GROUP 选项的实现与 IP_ADD_MEMBERSHIP 和 IP_DROP_MEMBERSHIP 选项类似，最后都是调用 ip_mc_join_group()和 ip_mc_leave_group()实现加入和离开组的功能。

15.6.11　MCAST_BLOCK_SOURCE 和 MCAST_UNBLOCK_SOURCE

MCAST_BLOCK_SOURCE 选项用于阻塞组中指定的组播源，而 MCAST_UNBLOCK_SOURCE 选项则开通先前被阻塞的组播源。

```
struct group_source_req gsourcer;
setsockopt(sock, IPPROTO_IP, MCAST_BLOCK_SOURCE, &gsourcer, sizeof(gsourcer));
setsockopt(sock, IPPROTO_IP, MCAST_UNBLOCK_SOURCE, &gsourcer, sizeof(gsourcer));
```

group_source_req 为待阻塞或开通组播源的相关信息结构：

```
151 struct group_source_req
152 {
153     __u32                gsr_interface;   /* interface index */
154     struct __kernel_sockaddr_storage gsr_group;    /* group address */
155     struct __kernel_sockaddr_storage gsr_source;    /* source address */
156 };
```

对 MCAST_BLOCK_SOURCE 和 MCAST_UNBLOCK_SOURCE，该结构各字段的意义如下：

gsr_interface 为阻塞或开通组播源的网络接口。

gsr_group 为阻塞或开通组播源待加入组播组的组地址。

gsr_source 为阻塞或开通的组播源。

15.6.12　MCAST_JOIN_SOURCE_GROUP 和 MCAST_LEAVE_SOURCE_GROUP

MCAST_JOIN_SOURCE_GROUP 选项使网络接口和套接口加入指定的组播组，并且以 INCLUDE 模式配置组播源过滤，而 MCAST_LEAVE_SOURCE_GROUP 选项用于离开一个组播组。

```
struct group_source_req gsourcer;
setsockopt(sock, IPPROTO_IP, MCAST_JOIN_SOURCE_GROUP, &gsourcer, sizeof(gsourcer));
```

```
setsockopt(sock, IPPROTO_IP, MCAST_LEAVE_SOURCE_GROUP, &gsourcer, sizeof(gsourcer));
```

虽然 MCAST_JOIN_SOURCE_GROUP 和 MCAST_LEAVE_SOURCE_GROUP 选项和 MCAST_BLOCK_SOURCE 和 MCAST_UNBLOCK_SOURCE 选项一样都使用了 group_source_req 结构，但结构字段的具体意义是不同的：

对于 MCAST_JOIN_SOURCE_GROUP 和 MCAST_LEAVE_SOURCE_GROUP，意义如下：

gsr_interface 为待加入或离开组播组的网络接口。

gsr_group 为待加入或离开组播组的组地址。

gsr_source 当 MCAST_JOIN_SOURCE_GROUP 选项时为 INCLUDE，而当 MCAST_LEAVE_SOURCE_GROUP 选项时无效。

MCAST_BLOCK_SOURCE 和 MCAST_UNBLOCK_SOURCE 选项的实现与 IP_BLOCK_SOURCE 和 IP_UNBLOCK_SOURCE 选项的实现，MCAST_JOIN_SOURCE_GROUP 和 MCAST_LEAVE_SOURCE_GROUP 选项与 IP_ADD_SOURCE_MEMBERSHIP 和 IP_DROP_SOURCE_MEMBERSHIP 选项的实现类似，参见之前的说明。

15.6.13 MCAST_MSFILTER

MCAST_MSFILTER 与 IP_MSFILTER 选项类似，也是用于配置组播源过滤的，在指定的网络接口上根据配置的组播源对特定的组播报文进行 EXCLUDE 或 INCLUDE 模式的过滤。

```
struct group_filter gfilter;
setsockopt(sock, IPPROTO_IP, IP_MSFILTER, &gfilter, sizeof(gfilter));
```

group_filter 为配置组播源过滤的相关信息结构：

```
158 struct group_filter
159 {
160     __u32                gf_interface;   /* interface index */
161     struct __kernel_sockaddr_storage gf_group;   /* multicast address */
162     __u32                gf_fmode;   /* filter mode */
163     __u32                gf_numsrc;   /* number of sources */
164     struct __kernel_sockaddr_storage gf_slist[1];  /* interface index */
165 };
```

gf_interface 为源过滤的网络设备接口。

gf_group 为源过滤的组播地址。

gf_fmode 为过滤模式，MCAST_EXCLUDE/MCAST_INCLUDE。

gf_numsrc 为过滤源地址的数量。

gf_slist 为过滤源地址列表。

MCAST_MSFILTER 选项与 IP_MSFILTER 选项的实现相类似，最后都是调用 ip_mc_msfilter()来完成具体的过滤，参见 15.6.6 节。

15.7 组播选路套接口选项

组播选路套接口选项用于启用和停止组播选路，添加和删除虚拟网络设备以及添加和删除组

播转发缓存。通常这些选项都是由组播路由协议守护进程调用 ip_mroute_setsockopt()来实现的。

15.7.1　MRT_INIT

MRT_INIT 用于启用组播选路，通常是在组播路由协议守护进程初始化时被调用，在启用组播选路之后、停止组播选路之前，再调用该选项是不会起作用的。

```
872         case MRT_INIT:
873            if (sk->sk_type != SOCK_RAW ||
874                inet_sk(sk)->num != IPPROTO_IGMP)
875                return -EOPNOTSUPP;
876            if(optlen!=sizeof(int))
877                return -ENOPROTOOPT;
878
879            rtnl_lock();
880            if (mroute_socket) {
881                rtnl_unlock();
882                return -EADDRINUSE;
883            }
884
885            ret = ip_ra_control(sk, 1, mrtsock_destruct);
886            if (ret == 0) {
887                write_lock_bh(&mrt_lock);
888                mroute_socket=sk;
889                write_unlock_bh(&mrt_lock);
890
891                ipv4_devconf.mc_forwarding++;
892            }
893            rtnl_unlock();
894            return ret;
```

873-875 必须是类型为 RAW，且本地端口为 IPPROTO_IGMP 的套接口才能进行 MRT_INIT 选项设置。

876-877 校验选项值长度。

880-883 已经启用了组播选路，则不能再次进行 MRT_INIT 选项设置，直接返回 EADDRINUSE 错误码。

885-892 将进行 MRT_INIT 选项调用的套接口添加到 ip_ra_chain 散列表中，以便使其有机会处理路由告警选项。然后设置 mroute_socket，并启用组播路由。

ip_ra_control()的第三个参数是将套接口从 ip_ra_chain 散列表上删除的析构函数，参见 12.11 节。这里的 mrtsock_destruct()是将组播路由协议守护进程的套接口从 ip_ra_chain 散列表中删除的回调函数，激活时停止组播路由，并将 mroute_socket 设置为 NULL，然后删除并释放所有的虚拟接口、组播转发缓存和临时组播转发缓存。

15.7.2　MRT_DONE

MRT_DONE 用来停止组播选路，通常在组播路由协议守护进程退出时被调用。被调用时，将组播路由协议守护进程的套接口从 ip_ra_chain 散列表中删除。

```
895         case MRT_DONE:
896            if (sk!=mroute_socket)
```

```
897                 return -EACCES;
898             return ip_ra_control(sk, 0, NULL);
```

15.7.3 MRT_ADD_VIF 和 MRT_DEL_VIF

MRT_ADD_VIF 和 MRT_DEL_VIF 选项用于添加和删除虚拟网络设备，通常是由组播路由协议守护进程来操作的。

15.7.4 MRT_ADD_MFC 和 MRT_DEL_MFC

MRT_ADD_MFC 和 MRT_DEL_MFC 选项用于增加和删除组播转发缓存，通常也是由组播路由协议守护进程来操作的。

15.7.5 MRT_ASSERT

MRT_ASSERT 选项用于启用或禁止 PIM 警告。

15.8 组播的 ioctl

ipmr_ioctl()是支持 IP 组播 ioctl 的例程，由 raw_ioctl()调用，而 raw_ioctl()进而被 inet_ioctl()调用，包括 SIOCGETVIFCNT 和 SIOCGETSGCNT 两个命令。

15.8.1 SIOCGETVIFCNT

SIOCGETVIFCNT 命令用来获取通过虚拟接口组播包的统计信息，获得的信息通过 sioc_vif_req 结构返回。sioc_vif_req 结构如下：

```
100 struct sioc_vif_req
101 {
102     vifi_t   vifi;        /* Which iface */
103     unsigned long icount;    /* In packets */
104     unsigned long ocount;    /* Out packets */
105     unsigned long ibytes;    /* In bytes */
106     unsigned long obytes;    /* Out bytes */
107 };
```

vifi 是要获取统计信息的虚拟接口在 vif_table 数组中的索引。

icount 和 ocount 是通过该虚拟接口的输入和输出总组播包数。

ibytes 和 obytes 是通过该虚拟接口的输入和输出组播包的总字节数。

15.8.2 SIOCGETSGCNT

SIOCGETSGCNT 命令用来获取指定组播源及目的地址的组播包的统计信息，获得的信息通过 sioc_sg_req 结构返回。sioc_sg_req 结构如下：

```
87 struct sioc_sg_req
88 {
89     struct in_addr src;
90     struct in_addr grp;
91     unsigned long pktcnt;
```

```
92     unsigned long bytecnt;
93     unsigned long wrong_if;
94 };
```

src 和 grp 分别是指定的组播源及组播地址。

pktcnt 和 bytecnt 是总组播包数和组播包的总字节数。

wrong_if 为在组播转发过程中发生失败的次数。

15.9　组播报文的输入

对于输入到本地或转发的组播报文，在经过 netfilter 处理之后就会调用 ip_rcv_finish()正式进入输入的处理。先调用 ip_route_input()进行输入路由的查询，如果发现目的地址为组播地址，就会按照组播地址的规则查找路由，查找到组播的输入路由后，组播报文接收处理函数为 ip_mr_input()。参见 19.10 节的 ip_route_input_mc()。

```
1338 int ip_mr_input(struct sk_buff *skb)
1339 {
1340     struct mfc_cache *cache;
1341     int local = ((struct rtable*)skb->dst)->rt_flags&RTCF_LOCAL;
1342
1343     /* Packet is looped back after forward, it should not be
1344        forwarded second time, but still can be delivered locally.
1345      */
1346     if (IPCB(skb)->flags&IPSKB_FORWARDED)
1347         goto dont_forward;
1348
1349     if (!local) {
1350             if (IPCB(skb)->opt.router_alert) {
1351                 if (ip_call_ra_chain(skb))
1352                     return 0;
1353             } else if (skb->nh.iph->protocol == IPPROTO_IGMP){
1354                 /* IGMPv1 (and broken IGMPv2 implementations sort of
1355                    Cisco IOS <= 11.2(8)) do not put router alert
1356                    option to IGMP packets destined to routable
1357                    groups. It is very bad, because it means
1358                    that we can forward NO IGMP messages.
1359                  */
1360                 read_lock(&mrt_lock);
1361                 if (mroute_socket) {
1362                     nf_reset(skb);
1363                     raw_rcv(mroute_socket, skb);
1364                     read_unlock(&mrt_lock);
1365                     return 0;
1366                 }
1367                 read_unlock(&mrt_lock);
1368             }
1369     }
```

1341-1369 检测处理不发往本地的特殊报文。

1341 通过报文目的路由中 rt_flags 标志来获取该组播包能否是发往本地的标志。

1346-1347 如果该组播包已经转发过，则跳过转发部分代码。由于组播回环在组播转发

之后进行，因此组播转发之后需要设置标志，以免在进行回环时再次进行转发。

1350-1352 如果报文不是发送往本地的，则检测 IP 的路由警告选项，如果存在该选项，则说明不能转发这个报文，应该传递给那些设置了 IP_ROUTER_ALERT 套接口选项的用户进程进行处理，参见 12.11 节。

1353-1368 IGMP 协议报文不能放置路由警告选项，也不能由路由器转发，因此需要由 mroute_socket（mrouted 进程）套接口接收。mroute_socket 参见 15.7.1 节和 15.7.2 节。

```
1371     read_lock(&mrt_lock);
1372     cache = ipmr_cache_find(skb->nh.iph->saddr, skb->nh.iph->daddr);
1373
1374     /*
1375      *    No usable cache entry
1376      */
1377     if (cache==NULL) {
1378        int vif;
1379
1380        if (local) {
1381           struct sk_buff *skb2 = skb_clone(skb, GFP_ATOMIC);
1382           ip_local_deliver(skb);
1383           if (skb2 == NULL) {
1384              read_unlock(&mrt_lock);
1385              return -ENOBUFS;
1386           }
1387           skb = skb2;
1388        }
1389
1390        vif = ipmr_find_vif(skb->dev);
1391        if (vif >= 0) {
1392           int err = ipmr_cache_unresolved(vif, skb);
1393           read_unlock(&mrt_lock);
1394
1395           return err;
1396        }
1397        read_unlock(&mrt_lock);
1398        kfree_skb(skb);
1399        return -ENODEV;
1400     }
1401
1402     ip_mr_forward(skb, cache, local);
1403
1404     read_unlock(&mrt_lock);
1405
1406     if (local)
1407        return ip_local_deliver(skb);
1408
1409     return 0;
```

1372 根据组播报文的源地址和目的地址查找组播转发缓存。

1377-1399 处理找不到组播转发缓存的情况。

1377-1388 在找不到组播转发缓存的情况下处理发往本地的报文，需克隆一个报文。

1390-1396 虽然找不到组播转发缓存，但输入网络设备创建了虚拟网络设备，则先添加一个临时组播转发缓存，并把需要转发的报文缓存下来，同时通知 mrouted 进程需要添加这样

的一条组播转发缓存。

1397-1399 如果既找不到组播转发缓存，输入网络设备也没有创建虚拟网络设备，则说明该组播包并没有加入某个组；或者该组播包无效，因此释放之。

1402-1407 组播包的转发。

1402 找到了该组播包的组播转发缓存后，即可调用 ip_mr_forward()转发该组播包。

1406-1407 如果该组播包是发送到本地的，则还需调用 ip_local_deliver()处理。

```
1411 dont_forward:
1412     if (local)
1413         return ip_local_deliver(skb);
1414     kfree_skb(skb);
1415     return 0;
1416 }
```

1412-1413 如果报文是发给本地的，则调用 ip_local_deliver()将报文上传。

1414-1415 如果报文既不转发又不发送到本地，则不做处理直接释放。

15.10　组播报文的转发

15.10.1　ip_mr_forward()

在处理输入的组播报文时，如果查找到组播转发缓存，则说明该组播报文需要转发。组播报文的转发由 ip_mr_forward()完成，参数说明如下：

- skb，待转发的组播报文。
- cache，允许该组播包进行转发的转发缓存。
- local，当 local 为非 0 时，表示该组播包转发后是否还需要发送到本地，因此需要克隆一份组播报文进行转发。

```
1252 static int ip_mr_forward(struct sk_buff *skb, struct mfc_cache *cache, int local)
1253 {
1254     int psend = -1;
1255     int vif, ct;
1256
1257     vif = cache->mfc_parent;
1258     cache->mfc_un.res.pkt++;
1259     cache->mfc_un.res.bytes += skb->len;
```

更新该组播转发缓存转发的总字节数和总报文数，包括在找不到对应输入设备虚拟接口的情况下不会被转发的报文。

```
1261     /*
1262      * Wrong interface: drop packet and (maybe) send PIM assert.
1263      */
1264     if (vif_table[vif].dev != skb->dev) {
1265         int true_vifi;
1266
1267         if (((struct rtable*)skb->dst)->fl.iif == 0) {
1268             /* It is our own packet, looped back.
1269                Very complicated situation...
```

```
1270
1271            The best workaround until routing daemons will be
1272            fixed is not to redistribute packet, if it was
1273            send through wrong interface. It means, that
1274            multicast applications WILL NOT work for
1275            (S,G), which have default multicast route pointing
1276            to wrong oif. In any case, it is not a good
1277            idea to use multicasting applications on router.
1278          */
1279         goto dont_forward;
1280       }
1281
1282       cache->mfc_un.res.wrong_if++;
1283       true_vifi = ipmr_find_vif(skb->dev);
1284
1285       if (true_vifi >= 0 && mroute_do_assert &&
1286          /* pimsm uses asserts, when switching from RPT to SPT,
1287             so that we cannot check that packet arrived on an oif.
1288             It is bad, but otherwise we would need to move pretty
1289             large chunk of pimd to kernel. Ough... --ANK
1290           */
1291          (mroute_do_pim || cache->mfc_un.res.ttls[true_vifi] < 255) &&
1292          time_after(jiffies,
1293                cache->mfc_un.res.last_assert + MFC_ASSERT_THRESH)) {
1294          cache->mfc_un.res.last_assert = jiffies;
1295          ipmr_cache_report(skb, true_vifi, IGMPMSG_WRONGVIF);
1296       }
1297       goto dont_forward;
1298    }
```

1264-1298 处理找不到对应输入的虚拟接口的情况。

1267-1280 在找不到该组播报文输入设备虚拟接口，且输入网络设备有异常的情况下，不能转发报文。

1282 更新转发过程失败计数。

1283-1297 根据组播报文的输入网络设备重新查找对应的虚拟接口，一旦查找成功，并且设置了需要发送路由警告，同时接收 pim 协议或转发缓存中对应网络设备的 TTL 有效时，发送 IGMPMSG_WRONGVIF 类型的报告，用于在 PIM-SM 组播路由器之间选择最短路径或者触发一个 PIM 声明（assert）消息。

1297 找不到转发缓存，不能转发组播报文。

```
1300    vif_table[vif].pkt_in++;
1301    vif_table[vif].bytes_in+=skb->len;
```

更新对应虚拟接口的输入组播报文总字节数和输入组播报文总数。

```
1303    /*
1304     *    Forward the frame
1305     */
1306    for (ct = cache->mfc_un.res.maxvif-1; ct >= cache->mfc_un.res.minvif; ct--) {
1307       if (skb->nh.iph->ttl > cache->mfc_un.res.ttls[ct]) {
1308          if (psend != -1) {
1309             struct sk_buff *skb2 = skb_clone(skb, GFP_ATOMIC);
1310             if (skb2)
```

```
1311                    ipmr_queue_xmit(skb2, cache, psend);
1312            }
1313            psend=ct;
1314        }
1315    }
1316    if (psend != -1) {
1317        if (local) {
1318            struct sk_buff *skb2 = skb_clone(skb, GFP_ATOMIC);
1319            if (skb2)
1320                ipmr_queue_xmit(skb2, cache, psend);
1321        } else {
1322            ipmr_queue_xmit(skb, cache, psend);
1323            return 0;
1324        }
1325    }
```

1306-1325 向各个虚拟接口转发组播报文。

1306-1315 遍历组播转发缓存的 ttls 数组，比较组播报文的 TTL 与虚拟设备的 TTL 阈值，报文只能从阈值小于报文 TTL 的虚拟接口中输出。

1316-1325 从最后一个符合条件的虚拟接口转发报文，如果报文需要发送到本地，则克隆一个报文发送，否则直接发送。

```
1327 dont_forward:
1328    if (!local)
1329        kfree_skb(skb);
1330    return 0;
1331 }
```

在不能转发的情况下，如果也无需发送到本地，则释放该报文。

15.10.2　ipmr_queue_xmit()

ipmr_queue_xmit()用来转发组播报文，在 ip_mr_forward()中被调用。参数说明如下：

- skb，待转发的组播报文。
- cache，允许该组播包进行转发的转发缓存。
- vifi，组播报文输出设备对应的虚拟接口。

```
1138 static void ipmr_queue_xmit(struct sk_buff *skb, struct mfc_cache *c, int vifi)
1139 {
1140    struct iphdr *iph = skb->nh.iph;
1141    struct vif_device *vif = &vif_table[vifi];
1142    struct net_device *dev;
1143    struct rtable *rt;
1144    int   encap = 0;
1145
1146    if (vif->dev == NULL)
1147        goto out_free;
```

检测输出虚拟接口对应的物理网络设备是否有效。

```
1149 #ifdef CONFIG_IP_PIMSM
1150    if (vif->flags & VIFF_REGISTER) {
```

```
1151            vif->pkt_out++;
1152            vif->bytes_out+=skb->len;
1153                 ((struct net_device_stats*)netdev_priv(vif->dev))->tx_bytes += skb->len;
1154            ((struct net_device_stats*)netdev_priv(vif->dev))->tx_packets++;
1155            ipmr_cache_report(skb, vifi, IGMPMSG_WHOLEPKT);
1156            kfree_skb(skb);
1157            return;
1158        }
1159 #endif
```

将 IGMPMSG_WHOLEPKT 类型的报告发送给路由守护进程。

```
1161    if (vif->flags&VIFF_TUNNEL) {
1162        struct flowi fl = { .oif = vif->link,
1163                    .nl_u = { .ip4_u =
1164                        { .daddr = vif->remote,
1165                          .saddr = vif->local,
1166                          .tos = RT_TOS(iph->tos) } },
1167                    .proto = IPPROTO_IPIP };
1168        if (ip_route_output_key(&rt, &fl))
1169            goto out_free;
1170        encap = sizeof(struct iphdr);
1171    } else {
1172        struct flowi fl = { .oif = vif->link,
1173                    .nl_u = { .ip4_u =
1174                        { .daddr = iph->daddr,
1175                          .tos = RT_TOS(iph->tos) } },
1176                    .proto = IPPROTO_IPIP };
1177        if (ip_route_output_key(&rt, &fl))
1178            goto out_free;
1179    }
1180
1181    dev = rt->u.dst.dev;
```

1161-1179 如果输出设备对应的虚拟接口是隧道类型，则根据隧道终点 IP 地址查找路由。否则，根据组播报文的目的地址查找路由，如果查找不到路由，则丢弃该组播报文。

1181 获取组播报文输出的物理网络设备。

```
1183    if (skb->len+encap > dst_mtu(&rt->u.dst) && (ntohs(iph->frag_off) & IP_DF))
{
1184        /* Do not fragment multicasts. Alas, IPv4 does not
1185          allow to send ICMP, so that packets will disappear
1186          to blackhole.
1187         */
1188
1189        IP_INC_STATS_BH(IPSTATS_MIB_FRAGFAILS);
1190        ip_rt_put(rt);
1191        goto out_free;
1192    }
1193
1194    encap += LL_RESERVED_SPACE(dev) + rt->u.dst.header_len;
1195
1196    if (skb_cow(skb, encap)) {
1197        ip_rt_put(rt);
1198        goto out_free;
```

```
1199     }
```

1183-1192 如果报文长度大于路径 MTU，且报文设置不允许分片位，则不能转发该组播报文，递减对已找到路由的引用，释放该报文。

1194-1199 检测转发的组播报文其 skb 的 headroom 部分是否有足够的空间。

```
1201     vif->pkt_out++;
1202     vif->bytes_out+=skb->len;
1203
1204     dst_release(skb->dst);
1205     skb->dst = &rt->u.dst;
1206     iph = skb->nh.iph;
1207     ip_decrease_ttl(iph);
1208
1209     /* FIXME: forward and output firewalls used to be called here.
1210      * What do we do with netfilter? -- RR */
1211     if (vif->flags & VIFF_TUNNEL) {
1212        ip_encap(skb, vif->local, vif->remote);
1213        /* FIXME: extra output firewall step used to be here. --RR */
1214        ((struct ip_tunnel *)netdev_priv(vif->dev))->stat.tx_packets++;
1215        ((struct ip_tunnel *)netdev_priv(vif->dev))->stat.tx_bytes+=skb->len;
1216     }
1217
1218     IPCB(skb)->flags |= IPSKB_FORWARDED;
1219
1220     /*
1221      * RFC1584 teaches, that DVMRP/PIM router must deliver packets locally
1222      * not only before forwarding, but after forwarding on all output
1223      * interfaces. It is clear, if mrouter runs a multicasting
1224      * program, it should receive packets not depending to what interface
1225      * program is joined.
1226      * If we will not make it, the program will have to join on all
1227      * interfaces. On the other hand, multihoming host (or router, but
1228      * not mrouter) cannot join to more than one interface - it will
1229      * result in receiving multiple packets.
1230      */
1231     NF_HOOK(PF_INET, NF_IP_FORWARD, skb, skb->dev, dev,
1232        ipmr_forward_finish);
1233     return;
1235 out_free:
1236     kfree_skb(skb);
1237     return;
1238 }
```

1201-1233 对组播报文进行转发。

1201-1202 更新从该虚拟接口输出的报文总字节数和报文总数。

1204-1205 用查询得到的输出路由缓存更新 skb 的目的路由缓存。

1206-1207 递减转发组播报文的 TTL。

1211-1216 处理 IP-IP 隧道类型虚拟接口，这不属于本书范围的内容，不作展开。

1218 设置 IP 控制块的 IPSKB_FORWARDED 标志，标识该报文已转发。

1231-1232 通过 netfilter 的 FORWARD 点对报文进行过滤等规则的检查之后，调用 ipmr_forward_finish()进行组播报文的输出处理。

1235 如果不能转发组播报文，跳转到这里释放报文后返回。

15.11 组播报文的输出

对于从本地输出或是需进行转发的组播报文，如果输出路由查找成功，便可以输出，输出处理函数为 ip_mc_output()，参见 20.4.2 节。

```
211 int ip_mc_output(struct sk_buff *skb)
212 {
213     struct sock *sk = skb->sk;
214     struct rtable *rt = (struct rtable*)skb->dst;
215     struct net_device *dev = rt->u.dst.dev;
216
217     /*
218      *    If the indicated interface is up and running, send the packet.
219      */
220     IP_INC_STATS(IPSTATS_MIB_OUTREQUESTS);
221
222     skb->dev = dev;
223     skb->protocol = htons(ETH_P_IP);
```

设置输出报文的输出网络设备及协议。

```
225     /*
226      *    Multicasts are looped back for other local users
227      */
228
229     if (rt->rt_flags&RTCF_MULTICAST) {
230         if ((!sk || inet_sk(sk)->mc_loop)
231 #ifdef CONFIG_IP_MROUTE
240             && ((rt->rt_flags&RTCF_LOCAL) || !(IPCB(skb)->flags&IPSKB_FORWARDED))
241 #endif
242         ) {
243             struct sk_buff *newskb = skb_clone(skb, GFP_ATOMIC);
244             if (newskb)
245                 NF_HOOK(PF_INET, NF_IP_POST_ROUTING, newskb, NULL,
246                     newskb->dev,
247                     ip_dev_loopback_xmit);
248         }
249
250         /* Multicasts with ttl 0 must not go beyond the host */
251
252         if (skb->nh.iph->ttl == 0) {
253             kfree_skb(skb);
254             return 0;
255         }
256     }
```

229-248 在以下条件下，则需要克隆一份报文输出到回环接口。

- 组播报文没有宿主，即不是从本地输出的组播报文。
- 组播报文需发送回路。
- 在编译支持 IP 组播的情况下，同时该组播报文是输入到本地的。

- 组播报文还未转发。

252-255 如果待输出组播报文的 TTL 为 0，则将其释放。

```
258     if (rt->rt_flags&RTCF_BROADCAST) {
259        struct sk_buff *newskb = skb_clone(skb, GFP_ATOMIC);
260        if (newskb)
261           NF_HOOK(PF_INET, NF_IP_POST_ROUTING, newskb, NULL,
262              newskb->dev, ip_dev_loopback_xmit);
263     }
```

如果输出路由类型为广播类型，则需克隆一份发送给回环设备。

```
265     return NF_HOOK_COND(PF_INET, NF_IP_POST_ROUTING, skb, NULL, skb->dev,
266                ip_finish_output,
267                !(IPCB(skb)->flags & IPSKB_REROUTED));
268 }
```

最后，通过 netfilter 模块处理后，调用 ip_finish_output()将该组播报文输出。

第 16 章　IGMP：Internet 组管理协议

IGMP 协议被 IPv4 系统（主机或路由器）用来向邻接的组播路由器报告自身的组成员关系。需要注意的是，IP 组播路由器本身也可能是一个或多个组播组的成员。在这种情况下，它既会执行协议的“组播路由器部分”，为它的组播路由协议收集成员信息，也会执行协议的“组成员部分”，把自己的成员关系通知自己、其他主机以及邻接的组播路由器。

IGMP 协议还用于其他的 IP 组播管理功能，这是通过使用组成员报告之外的其他消息类型来实现的。

RFC1112 中描述的第一版是第一个获得广泛使用的版本，也是第一个成为因特网标准的版本。而 RFC2236 中说明的第二版增加了对“快速离开”的支持，大大减少了组播路由器获知相连网络的某一个组中已经没有组成员所花费的时间。RFC3376 中描述的第三版增加了对“源过滤”的支持，即系统有能力在发往某个特定组播地址的数据报中通告，只希望接收某些特定源的报文以支持特定源组播，或者只希望接收除了某些特定源的报文。

IGMP 协议的处理涉及以下文件：

- include/linux/inetdevice.h，定义 IPv4 专用的网络设备相关的结构、宏等。
- include/linux/igmp.h，定义 IGMP 报文等结构、宏和函数原型。
- net/ipv4/igmp.c，IGMP 协议的实现。

16.1　in_device 结构中的组播参数

网络设备层与 IPv4 相关的配置都存放在 in_device 结构中，包括有关组播的配置信息。

```
39 struct in_device
40 {
   .. ...
45    rwlock_t        mc_list_lock;
46    struct ip_mc_list    *mc_list;    /*IP multicast filter chain*/
47    spinlock_t        mc_tomb_lock;
48    struct ip_mc_list    *mc_tomb;
49    unsigned long        mr_v1_seen;
50    unsigned long        mr_v2_seen;
51    unsigned long        mr_maxdelay;
52    unsigned char        mr_qrv;
53    unsigned char        mr_gq_running;
54    unsigned char        mr_ifc_count;
55    struct timer_list     mr_gq_timer;    /* general query timer */
56    struct timer_list     mr_ifc_timer;  /*interface change timer*/
   .. ...
61 };
```

45 rwlock_t mc_list_lock

用于控制访问 mc_list 的读写锁。

46 struct ip_mc_list *mc_list

所属设备加入的组播地址列表。

47 spinlock_t mc_tomb_lock

用于控制访问 mc_tomb 的自旋锁。

48 struct ip_mc_list *mc_tomb

记录所属接口最近离开组播组的信息，用于报告“组播源列表改变到不希望接收记录”类型的组报告。当所属接口加入到 mc_tomb 上的组播组时，该组播组会从这里删除。

49 unsigned long mr_v1_seen

50 unsigned long mr_v2_seen

用于判断 IGMP 报文为第一版还是第二版的超时时间。在接收到第一版或第二版的 IGMP 报文之后，会设置该值，通常超时时间为 400s。

51 unsigned long mr_maxdelay

当主机收到一个普通查询后发送报告的延迟时间上限。

52 unsigned char mr_qrv

健壮变量。如果预计网络中会丢失报文，则可以增加健壮变量。通常情况下应该大于或等于 2，默认值为 2。

53 unsigned char mr_gq_running

用于标识 mr_gq_timer 定时器是否处于运行中。

54 unsigned char mr_ifc_count

当输出 IGMPv3 报告报文超时时，重传 IGMPv3 报告报文次数的最大值。与健壮变量相关。

55 struct timer_list mr_gq_timer

延迟发送回复普通查询报告的定时器。

56 struct timer_list mr_ifc_timer

重传 IGMPv3 报告报文定时器。

16.2　ip_mc_list 结构

ip_mc_list 结构描述网络设备加入组播组的配置块，用来维护网络设备组播的状态，包括组播地址、过滤模式、源地址等。

```
170 struct ip_mc_list
171 {
172     struct in_device    *interface;
173     __be32              multiaddr;
174     struct ip_sf_list   *sources;
175     struct ip_sf_list   *tomb;
176     unsigned int        sfmode;
177     unsigned long       sfcount[2];
178     struct ip_mc_list   *next;
179     struct timer_list   timer;
180     int                 users;
181     atomic_t            refcnt;
182     spinlock_t          lock;
183     char                tm_running;
```

```
184     char            reporter;
185     char            unsolicit_count;
186     char            loaded;
187     unsigned char       gsquery;    /* check source marks? */
188     unsigned char       crcount;
189 };
```

172 struct in_device *interface

指向所属网络设备的 IP 配置块。

173 __be32 multiaddr

网络设备加入的组播地址。

174 struct ip_sf_list *sources

源地址过滤列表，ip_sf_list 结构说明如下。

```
160 struct ip_sf_list
161 {
162     struct ip_sf_list   *sf_next;
163     __be32          sf_inaddr;
164     unsigned long       sf_count[2];    /* include/exclude counts */
165     unsigned char       sf_gsresp;    /* include in g & s response? */
166     unsigned char       sf_oldin;    /* change state */
167     unsigned char       sf_crcount;    /* retrans. left to send */
168 };
```

162 struct ip_sf_list *sf_next

用于将多个组播源地址信息块链接起来。

163 __be32 sf_inaddr

需过滤的组播源。

164 unsigned long sf_count[2]

不同过滤模式（EXCLUDE 和 INCLUDE）且指定组播源过滤地址的套接口在该接口加入组播组的数量。

165 unsigned char sf_gsresp

标识当前该组播源地址是否应答指定组或源的查询。

166 unsigned char sf_oldin

组播源地址状态标志。在设置组播源过滤列表前，标识对应组播源地址的过滤是否处于同一种过滤模式，参见 16.10.3 节。

167 unsigned char sf_crcount

当前还可重传“源列表改变记录”报告的次数。

175 struct ip_sf_list *tomb

记录所有最近被阻止的组播源。

176 unsigned int sfmode

所属接口的组播源过滤模式，EXCLUDE 或 INCLUDE。

177 unsigned long sfcount[2]

EXCLUDE 或 INCLUDE 过滤模式的套接口在本接口加入组的播组数。

注意：不要与 ip_sf_list 结构的 sf_count 混淆，sfcount 统计单位是组播组，而 sf_count 统计

单位是组播源。

178 struct ip_mc_list *next

将多个组播组信息块用链表的形式连接起来。

179 struct timer_list timer

接口定时器，当一个网络设备接口新加入到一个组播组，需要向组播路由器发送一个报告，通知该组播路由器需向本地网络转发该组的数据报。

180 int users

引用计数，记录当前在本接口上加入了该组播组的套接口数。

181 atomic_t refcnt

接口定时器 timer 的引用计数，只有当接口定时器停止时，所在的组播配置块才能释放。

182 spinlock_t lock

用来控制访问接口定时器 timer 的自旋锁。

183 char tm_running

标志 timer 当前是否正在运行。

184 char reporter

标志当前是否正要开始发送 IGMP 报告，1 为是，0 为否。

185 char unsolicit_count

在一个接口新加入到一个组播组时发送主动报告的次数，默认值为 IGMP_Unsolicited_Report_Count，即 2。

186 char loaded

标识在本接口加入的组播组，其硬件地址是否已经加载到网络设备上。

187 unsigned char gsquery

标识当前是否进行指定组和源的查询。

188 unsigned char crcount

用来控制发送 V3 的过滤模式改变记录类型报告的次数，与健壮变量相关。每发送一次，crcount 会递减 1，直至为 0。

16.3　系统参数

系统参数如下：

- igmp_max_memberships，一个套接口允许加入的组播组数上限，默认值为 20 个。
- igmp_max_msf，一组播地址的源过滤列表数上限，默认值为 10 个。
- force_igmp_version，强制主机或网络接口支持的 IGMP 版本。有效的值为 1 或 2，其他值则表示根据接收到 IGMP 报文的实际版本和时间来判断为第一版还是第二版。

16.4　IGMP 的版本与协议结构

16.4.1　IGMP 的版本

在网络中，很可能存在支持 IGMPv3 的路由器，而还有主机不支持 IGMPv3 协议。因此，

为了能与旧版本 IGMP 的主机兼容，IGMPv3 路由器必须运行在 IGMPv1 或 IGMPv2 兼容模式。IGMPv3 为每一个组记录维护一个兼容模式，组的兼容模式由组的兼容模式变量来决定，该变量可能是下列各值中的一个：IGMPv1，IGMPv2 和 IGMPv3。每条组记录都有这么一个变量，其值取决于该组所接收到的成员报告的版本，以及该组的旧版本主机存在定时器。

为了更好地在不同版本的 IGMP 间进行切换，路由器为每一个组维护了一个 IGMPv1 主机存在定时器和一个 IGMPv2 主机存在定时器。当收到一个 IGMPv1 成员关系报告时，IGMPv1 主机存在定时器会被设置为旧版本主机存在超时时间，而对 IGMPv2 成员关系报告同样也会设置相应的定时器。

当收到一个更旧版本的报告（即比当前兼容模式的版本更旧），或者特定的定时器事件发生，组记录的组兼容模式就会发生改变。例如：

- 当 IGMPv1 主机存在定时器超时，如果当前正在运行 IGMPv2 主机存在定时器，则切换到 IGMPv2 的兼容模式；否则直接切换到 IGMPv3 兼容模式。
- 当一个 IGMPv2 主机存在定时器超时并且没有 IGMPv1 主机存在定时器在运行，就切换到 IGMPv3 兼容模式。

需要注意的是，当一个组切换回 IGMPv3 模式时，需要一些时间去重新获取指定源的状态信息。指定源的信息通过下一个普通查询获取，但是应当被阻止的源直到下一个组成员关系查询后才能被阻止。

组兼容模式变量的值取决于在上一个旧版本主机存在超时时间内，是否收到一个更旧版本的报告，见表 16-1。

表 16-1 组兼容模式的设置规则

组兼容模式	定时器状态
IGMPv3(默认)	IGMPv2 主机存在没有运行 IGMPv1 主机存在也没有运行
IGMPv2	IGMPv2 主机存在正运行 IGMPv1 主机存在没有运行
IGMPv1	IGMPv1 主机存在正运行

如果一台路由器收到一个报告，该报告造成该路由器的旧版本主机存在定时器被更新并且其兼容模式也需发生相应的变化，则它应当立即切换其兼容模式。

同样为了能与旧版本 IGMP 的路由器兼容，网络上的 IGMPv3 主机也必须运行在 IGMPv1 或 IGMPv2 兼容模式。IGMPv3 使用了两个变量来记录接收到 V1 和 V2 查询报文的时间，用来判断 IGMP 报文为 V1 还是 V2 的超时时间。在接收到 V1 或 V2 的 IGMP 报文之后，会设置该值，通常超时时间为 400s。然后，在处理过程中通过宏 IGMP_V1_SEEN 和 IGMP_V2_SEEN 来检测当前 IGMP 协议的版本。

```
131  #define IGMP_V1_SEEN(in_dev) (ipv4_devconf.force_igmp_version == 1 || \
132          (in_dev)->cnf.force_igmp_version == 1 || \
133          ((in_dev)->mr_v1_seen && \
134          time_before(jiffies, (in_dev)->mr_v1_seen)))
135  #define IGMP_V2_SEEN(in_dev) (ipv4_devconf.force_igmp_version == 2 || \
136          (in_dev)->cnf.force_igmp_version == 2 || \
137          ((in_dev)->mr_v2_seen && \
138           time_before(jiffies, (in_dev)->mr_v2_seen)))
```

16.4.2　第一版和第二版的 IGMP 报文结构

IGMP 报文通过 IPv4 数据报进行封装，IP 协议号是 2。每一个 IGMP 消息的 IP 首部的生存时间都是 1，这意味 IGMP 只能管理子网内部的组成员关系。因特网控制的 IP 优先级和 IP 路由器警告选项负载在它的 IP 首部中。图 16-1 和图 16-2 分别给出了第一版和第二版的 IGMP 报文格式，内核中使用 igmphdr 结构来描述。

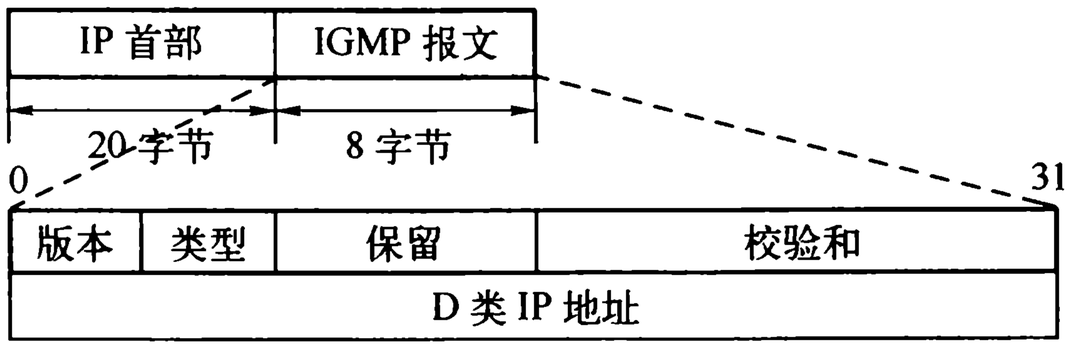

图 16-1　第一版的 IGMP 报文格式

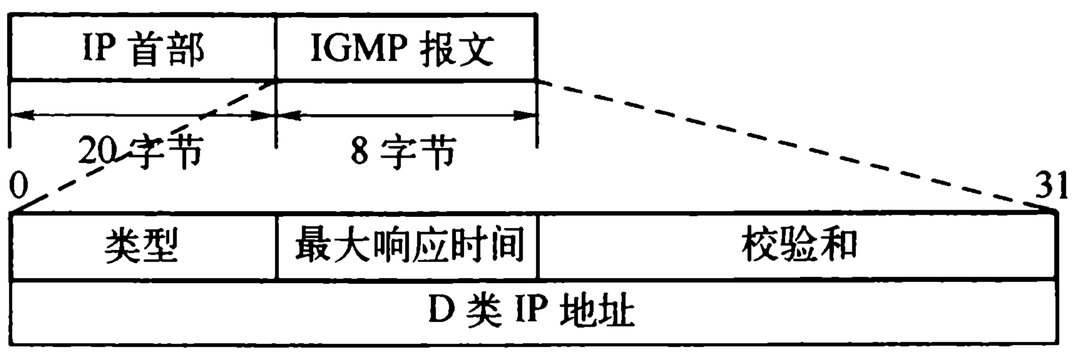

图 16-2　第二版的 IGMP 报文格式

报文组成如下所示：

- 版本，所使用的 IGMP 版本号。
- 类型，IGMP 消息的类型。
- 最大响应时间，在发送一个响应报告之前所允许的最大时间。
- 校验和，IGMP 消息的校验和。

16.4.3　第三版的 IGMP 查询报文结构

第三版的 IGMP 增加了对“源过滤”的支持，即系统能够在其发往某一特定组播组的查询中报告它只对某些来自特定源地址的数据或者除了某些特定源地址之外的数据感兴趣。这可被组播路由协议利用来避免把某些来自特定源地址的组播数据报发往对它不感兴趣的网络。第三版的 IGMP 查询报文格式见图 16-3。

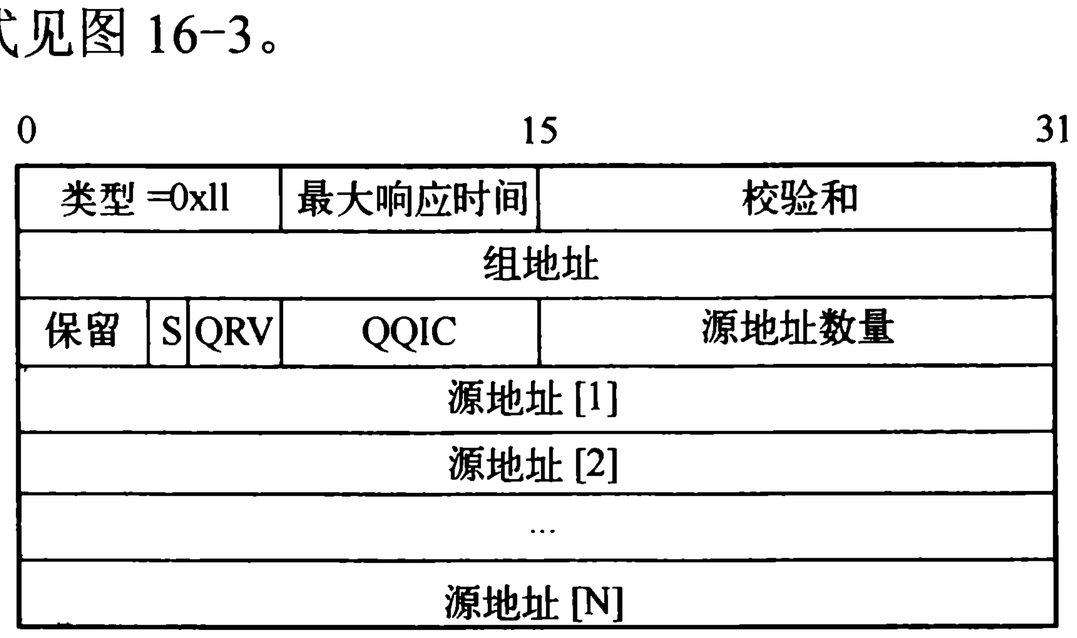

图 16-3　第三版的 IGMP 查询报文格式

报文组成如下所示：

- 最大响应代码，指定在发送一个响应报告之前所允许的最大时间。实际允许的时间，被称为最大响应时间，其单位为 0.1s。
- 校验和，对整个 IGMP 数据报以 16bit 为一段取反后求和得到校验和。为了计算校验和，校验和字段开始时必须被设置成 0。而在收到一个数据后，处理它之前必须先对其校验和进行验证。
- 组地址，在发送一个普通查询的时候，组地址字段必须被置 0。而当发送一个指定组查询或指定组和源查询时，必须被设置成要查询对象的 IP 组地址。
- 保留，在传输时必须置 0，在接收时必须忽略。
- S 标志(禁止路由器处理)，当该标志设置成 1 的时候，表示任何接收路由器禁止更新它们在收到查询时需更新的那些定时器，但不禁止查询者选举或在路由器上执行其作为一个组成员的主机端查询处理。
- QRV(查询者的健壮变量)，如果不为 0，则其中包含一个被查询者使用的健壮变量值，而如果该值超过 QRV 字段最大值 7，则 QRV 将被设成 0。路由器取最近收到的查询中的 QRV 值作为其自身的健壮变量值，除非最近收到的 QRV 为 0，在这种情况下，接收者使用默认的健壮变量值，或者是一个静态配置的值。
- QQIC(查询者的查询间隔代码)，指定查询者使用的查询间隔。实际的间隔，称为查询者的查询间隔(QQI)，以秒为单位。
- 源地址数量，表明该查询中存在多少个源地址。在普通查询或指定组查询中该字段值为 0，在指定组和源的查询中则为非 0。源地址数量受到查询所经过网络 MTU 的限制，例如，在 MTU 为 1500B 的以太网上，含有路由器警告选项的 IP 首部占去 24B，除源地址数量之外的 IGMP 字段占去 12B，还有 1464B 用于源地址，这就限制了源地址的数量最多只能有 366(1464/4)。
- 源地址[N]，n 个 IP 单播地址数组，n 是源地址数量字段的值。

16.4.4 第三版的 IGMP 报告结构

第三版成员关系报告由 IP 子系统发出，用来向邻接路由器报告当前的组播接收状态，或者修改它们的接口的组播接收状态。内核中使用 igmpv3_report 结构来描述第三版的 IGMP 报告结构，见图 16-4。

0	15	31
类型 =0x22	保留	校验和
保留		组记录数量
组记录 [1]		
组记录 [2]		
…		
组记录 [M]		

图 16-4 第三版的 IGMP 报告结构

（1）组记录数量

标明报告存在多少个组记录。

（2）组记录[i]

每一个组记录字段是一整块数据，包含关于发送者在报告发送接口上某一个组播组成员关系的信息。内核中用 igmpv3_grec 结构来描述组记录结构，见图 16-5。

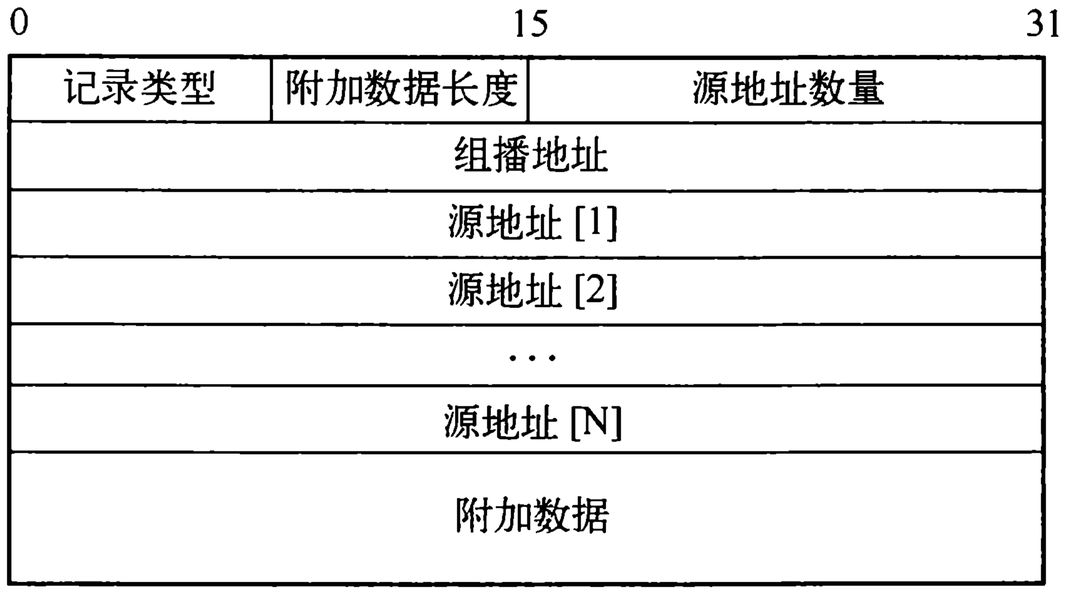

图 16-5　第三版的 IGMP 报告中组记录结构

（3）记录类型

在一个报告消息中，有一定数量的不同类型的组记录，各种记录类型如下所示。

“当前状态记录”用于响应在一个接口上收到的查询。它报告了接口跟某一个组播 IP 地址相关的当前的接收状态。当前状态记录的记录类型见表 16-2。

表 16-2　当前状态记录

名称	描述
IGMPV3_MODE_IS_INCLUDE	标明接口关于某一指定组播地址的过滤模式为 INCLUDE。该组记录中的源地址[i]字段含有该接口的关于该组播地址的源列表(如果非空)
IGMPV3_MODE_IS_EXCLUDE	标明接口关于某一指定组播地址的过滤模式为 EXCLUDE。该组记录中的源地址[i]字段含有该接口的关于该组播地址的源列表(如果非空)

“过滤模式改变记录”是当本地的接口层相关于某一特定组播 IP 地址的过滤模式的改变的时候(即从 INCLUDE 变到 EXCLUDE，或者从 EXCLUDE 变到 INCLUDE)，包含该记录报告是从发生改变的那个接口上发出来的。过滤模式改变记录的记录类型见表 16-3。

表 16-3　过滤模式改变记录

名称	描述
IGMPV3_CHANGE_TO_INCLUDE	标明接口关于某一指定的组播地址的过滤模式改变到 INCLUDE。该组记录中的源地址[i]字段含有该指定组播地址相关的新的源列表(如果非空)
IGMPV3_CHANGE_TO_EXCLUDE	标明接口关于某一指定的组播地址的过滤模式改变到 EXCLUDE。该组记录中的源地址[i]字段含有该指定组播地址相关的新的源列表(如果非空)

“源列表改变记录”是当本地的接口层相关于某一特定组播 IP 地址的源列表发生改变，并且该改变不跟过滤模式的改变产生冲突时发出。包含该记录的该报告是从发生改变的那个接口上发出来的。源列表改变记录的记录类型见表 16-4。

表 16-4　源列表改变记录

名称	描述
IGMPV3_ALLOW_NEW_SOURCES	标明组记录中的源地址[i]字段含有系统希望接收的发往某一组播地址的新的源的列表。如果这是对一个 INCLUDE 列表的改变，那么这些地址会被添加到列表中，如果这是对一个 EXCLUDE 列表的改变，那么这些地址会被从列表中删除
IGMPV3_BLOCK_OLD_SOURCES	标明组记录中的源地址[i]字段含有系统不希望再接收的发往某一组播地址的源的列表。如果这是对一个 INCLUDE 列表的改变，那么这些地址会被从列表中删除，如果这是对一个 EXCLUDE 列表的改变，那么这些地址会被添加到列表中

（4）附加数据长度

组记录中辅助数据的实际长度，单位是 32bit 字。如果为 0，就表示辅助数据不存在。

（5）源地址数量

标明在组记录中存在多少个源地址。

（6）组播地址

标明该组记录从属的组播 IP 地址。

（7）源地址[i]

一个含有 n 个单播地址的数组，n 就是源数量字段值。

（8）附加数据

如果存在辅助数据字段，则含有关于该组记录的一些附加信息。IGMPv3 没有定义任何辅助数据，因此，IGMPv3 的实现在任何传输的组记录中都不应该含有任何辅助数据，即必须把 Aux Data Len 字段置 0，并且必须在收到的所有组记录中忽略辅助数据的存在。

16.5 IGMP 报文的输入

对于非路由器，IGMP 报文输入由 igmp_rcv()处理。这是由在 inet_init()中注册的 igmp_protocol 结构定义的。

```
918 int igmp_rcv(struct sk_buff *skb)
919 {
920     /* This basically follows the spec line by line -- see RFC1112 */
921     struct igmphdr *ih;
922     struct in_device *in_dev = in_dev_get(skb->dev);
923     int len = skb->len;
924
925     if (in_dev==NULL) {
926         kfree_skb(skb);
927         return 0;
928     }
929
930     if (!pskb_may_pull(skb, sizeof(struct igmphdr)))
931         goto drop;
932
933     switch (skb->ip_summed) {
934     case CHECKSUM_COMPLETE:
935         if (!csum_fold(skb->csum))
936             break;
937         /* fall through */
938     case CHECKSUM_NONE:
939         skb->csum = 0;
940         if (__skb_checksum_complete(skb))
941             goto drop;
942     }
943
944     ih = skb->h.igmph;
945     switch (ih->type) {
946     case IGMP_HOST_MEMBERSHIP_QUERY:
947         igmp_heard_query(in_dev, skb, len);
948         break;
```

```
949     case IGMP_HOST_MEMBERSHIP_REPORT:
950     case IGMPV2_HOST_MEMBERSHIP_REPORT:
951     case IGMPV3_HOST_MEMBERSHIP_REPORT:
952        /* Is it our report looped back? */
953        if (((struct rtable*)skb->dst)->fl.iif == 0)
954           break;
955        /* don't rely on MC router hearing unicast reports */
956        if (skb->pkt_type == PACKET_MULTICAST ||
957           skb->pkt_type == PACKET_BROADCAST)
958           igmp_heard_report(in_dev, ih->group);
959        break;
960     case IGMP_PIM:
961 #ifdef CONFIG_IP_PIMSM_V1
962           in_dev_put(in_dev);
963           return pim_rcv_v1(skb);
964 #endif
965        case IGMP_DVMRP:
966        case IGMP_TRACE:
967        case IGMP_HOST_LEAVE_MESSAGE:
968        case IGMP_MTRACE:
969        case IGMP_MTRACE_RESP:
970           break;
971        default:
972           break;
973        }
974
975 drop:
976        in_dev_put(in_dev);
977        kfree_skb(skb);
978        return 0;
979 }
```

922-928 检测输入报文的网络设备的 IP 配置块是否有效。

930-931 通过判断报文长度的方法来简单地检测报文的有效性。

933-942 进行校验和处理。

946-948 由 igmp_heard_query()处理 IGMP 查询报文。

949-959 处理 IGMP 报告报文。

960-964 如果在编译时支持了 PIM 报文，则调用 pim_rcv_v1()处理之。

965-970 其他类型的 IGMP 报文全都忽略。

16.6　函数

16.6.1　ip_mc_find_dev()

ip_mreqn 结构包含组播地址、本地地址和网络设备索引，ip_mc_find_dev()根据这三个条件可获取网络设备的 IP 配置块，过程如下：

- 如果有指定网络设备索引，则根据该索引获取对应网络设备的 IP 配置块。
- 如果没有指定网络设备索引或没有获取到指定网络设备的 IP 配置块，则根据本地地址获取对应网络设备。

- 如果还是没有获取到网络设备，则将组播地址作为目的地址通过查找路由方式获取输出网络设备以及 IP 配置块。

16.6.2 ip_check_mc()

在路由子系统中，ip_check_mc()在查询路由时被调用，根据目的组播地址、源地址和三层协议类型来确定是否允许继续处理，检测过程如下：

（1）检测指定的网络设备是否已加入待检测的组播组。

（2）在满足第一个条件的情况下，如果待检测的是 IGMP 协议，则允许处理。

（3）如果待检测的不是 IGMP 协议，则需要检测源地址。对于没有指定源地址的，暂时允许处理；若指定了源地址，则需要通过源地址列表中的过滤。

16.7 成员关系查询

IGMP 从第一版发展到第三版，目前查询消息有三种类型的变体，见表 16-5。

表 16-5 查询消息类型

类型	描述
普通查询	由组播路由器发出，用于获知邻接接口（即查询所传输的网络中相连的接口）的完整的组播接收状态。在一个普通查询中，组地址字段和源数量(N)字段都为 0
指定组查询	由组播路由器发出，用于获知邻接接口中跟某一个 IP 地址相关的组播接收状态。在指定组查询中，“组地址”字段含有需要查询的那个组地址，源数量(N)字段为 0
指定组和源查询	由组播路由器发出，用于获知邻接接口是否需要接收来自指定的这些源的、发往指定组的组播数据报。在一个指定组和源的查询中，组地址字段含有要查询的组播地址，源地址[i]字段含有相关的源地址

系统接收到一个查询后不会立即响应，而是会把这个响应延迟一个随机时间，该时间在由接收到的查询消息中最大响应代码计算得到的最大响应时间的范围内。一个系统可能在不同的接口上接收到多个不同类型的查询（如普通查询，指定组查询，指定组和源的查询），每一个查询都需要相应的延迟响应。

在调度一个查询的响应之前，系统必须考虑到前一次调度还未完成的响应，大多数情况下，会重新调度一个组合的响应。因此，系统必须能够维护下列状态：

- 每一个接口上都有一个定时器，用来调度普通查询的响应。
- 每一个接口有与特定组相关的定时器，用来调度对指定组查询和指定组和源的查询的响应。
- 每一个接口有与特定组相关的源列表，被放置在指定组和源查询的响应中。

当一个带有路由器警告选项的查询到达接口时，那么系统的状态就是报告，报告的延迟时间从范围（0，最大响应时间）中获取。这里，最大响应时间是从接收到的查询中最大响应代码计算得到的。下面的规则用于确定一个报告是否需要被调度以及被调度报告的类型，这些规则依次被考虑，当发现第一个匹配规则时，即被应用：

- 如果还存在对前一个普通查询的未完成响应，它在被选择的延迟之前被调度，就不需要再调度额外的响应。
- 如果收到的查询是一个普通查询，接口的定时器在选定的延迟之后，必须调度一个普通查询的响应。
- 如果收到的查询是一个指定组查询，或者是一个指定组和源的查询，并且不存在该组的

前一个查询的未完成的响应，那么组定时器就用于调度一个报告。如果收到的查询是一个指定组和源的查询，那么被查询的源列表必须被记录下来，用于生成响应。

- 如果对该组的前一个查询已经存在未完成的响应，并且新的查询是一个指定组的查询，或者已经被记录下来的该组的源列表是空的，那么这个组的源列表被清空，并使用组定时器调度单个的响应。新的响应在未完成报告的剩余时间和选定的延迟两个时间中的较早那个时间被发送出去。
- 如果收到的查询是一个指定组和源的查询，并且该组存在一个拥有非空源列表的未完成响应。那么组的源列表被增长以含有新的查询的源列表，并且使用组的定时器调度一个单个的响应。新的响应在未完成报告的剩余时间和选定的延迟两个时间中的较早的那个时间被发送出去。

当未完成报告相关的定时器超时，系统在相关的接口上传送一个或多个报告消息，消息中含有一个或多个当前状态记录，如下：

- 如果超时的定时器是接口定时器（即它是一个响应普通查询的报告），那么为指定的拥有接收状态的接口上的每一个组播地址发送一个当前状态记录。当前状态记录含有该组播地址和相关的过滤模式（MODE_IS_INCLUDE 或者 MODE_IS_EXCLUDE），以及源列表。多个当前状态记录被尽可能地打包到一个报告消息中。
- 如果超时的定时器是组定时器，并且该组的记录下来的源列表是空的（即它是一个未完成的指定组的查询的响应），那么，当且仅当接口对该组地址有一个接收状态时，为该地址发送一个当前状态记录。当前状态记录含有该组播地址，它相关的过滤模式（MODE_IS_INCLUDE 或者 MODE_IS_EXCLUDE）以及源列表。
- 如果超时的定时器是组定时器，并且该组的记录下来的源列表是非空的（即它是一个未完成的指定组和源的查询的响应），那么，当且仅当接口对该组地址有一个接收状态时，用于响应的当前状态记录的内容由接口的状态和未完成响应记录来决定，见表 16-6。

表 16-6　接口状态与未完成响应记录

接口状态	未完成响应记录中的源集	当前状态记录
INCLUDE(A)	B	IS_IN(A*B)
EXCLUDE(A)	B	IS_IN(B-A)

如果结果的当前状态记录有一个空的源地址集，就不会发送响应。

最后，在需要的报告消息产生之后，清除所有与被报告组相关的源列表。

```
825 static void igmp_heard_query(struct in_device *in_dev, struct sk_buff *skb,
826     int len)
827 {
828     struct igmphdr         *ih = skb->h.igmph;
829     struct igmpv3_query *ih3 = (struct igmpv3_query *)ih;
830     struct ip_mc_list    *im;
831     __be32               group = ih->group;
832     int                max_delay;
833     int                mark = 0;
834
835
836     if (len == 8) {
837         if (ih->code == 0) {
```

```
838             /* Alas, old v1 router presents here. */
839
840             max_delay = IGMP_Query_Response_Interval;
841             in_dev->mr_v1_seen = jiffies +
842                IGMP_V1_Router_Present_Timeout;
843             group = 0;
844          } else {
845             /* v2 router present */
846             max_delay = ih->code*(HZ/IGMP_TIMER_SCALE);
847             in_dev->mr_v2_seen = jiffies +
848                IGMP_V2_Router_Present_Timeout;
849          }
850          /* cancel the interface change timer */
851          in_dev->mr_ifc_count = 0;
852          if (del_timer(&in_dev->mr_ifc_timer))
853             __in_dev_put(in_dev);
854          /* clear deleted report items */
855           igmpv3_clear_delrec(in_dev);
```

836 长度为 8 的 IGMP 报文是 V1 或 V2。

837-849 当主机收到一个查询后，不会立即发送一个报告，而是要延迟一段时间。V1 延迟时间为固定的 10s，而 V2 的延迟时间从查询报文中获取。然后，设置用于判断 IGMP 报文为 V1 或 V2 的超时时间。

851-855 由于接收到的是 IGMPv1 查询报文，因此需要清除有关 V3 的信息，重传 IGMPv3 报告报文的最大次数值 mr_ifc_count 清零，停止重传 IGMPv3 报告报文定时器 mr_ifc_timer，清除最近被阻止的组播源。

```
856       } else if (len < 12) {
857             return;    /* ignore bogus packet; freed by caller */
```

丢弃长度大于 8 小于 12（字节）的无效 IGMP 报文。

```
858       } else { /* v3 */
859          if (!pskb_may_pull(skb, sizeof(struct igmpv3_query)))
860             return;
861
862          ih3 = (struct igmpv3_query *) skb->h.raw;
863          if (ih3->nsrcs) {
864             if (!pskb_may_pull(skb, sizeof(struct igmpv3_query)
865                      + ntohs(ih3->nsrcs)*sizeof(__be32)))
866                return;
867             ih3 = (struct igmpv3_query *) skb->h.raw;
868          }
869
870          max_delay = IGMPV3_MRC(ih3->code)*(HZ/IGMP_TIMER_SCALE);
871          if (!max_delay)
872             max_delay = 1;    /* can't mod w/ 0 */
873          in_dev->mr_maxdelay = max_delay;
874          if (ih3->qrv)
875             in_dev->mr_qrv = ih3->qrv;
876          if (!group) { /* general query */
877             if (ih3->nsrcs)
878                return;    /* no sources allowed */
```

```
879             igmp_gq_start_timer(in_dev);
880             return;
881         }
882         /* mark sources to include, if group & source-specific */
883         mark = ih3->nsrcs != 0;
884     }
```

859-868 其他情况则可能都是 IGMPv3 报文，先检测报文长度是否有效。

870-873 从查询报文中获取发送普通查询报告的最大延迟时间。

874-875 从查询报文中获取健壮变量。

876-881 在 V3 的查询报文中没有指定组，则说明是普通查询，因此其源地址数量应该为 0，否则报文无效。返回前，启动普通查询定时器，一旦普通查询定时器激活即发送当前状态记录报告，参见 16.8.4 节。

883 标记查询报文中是否存在源地址。

```
886     /*
887      * - Start the timers in all of our membership records
888      *   that the query applies to for the interface on
889      *   which the query arrived excl. those that belong
890      *   to a "local" group (224.0.0.X)
891      * - For timers already running check if they need to
892      *   be reset.
893      * - Use the igmp->igmp_code field as the maximum
894      *   delay possible
895      */
896     read_lock(&in_dev->mc_list_lock);
897     for (im=in_dev->mc_list; im!=NULL; im=im->next) {
898         int changed;
899
900         if (group && group != im->multiaddr)
901             continue;
902         if (im->multiaddr == IGMP_ALL_HOSTS)
903             continue;
904         spin_lock_bh(&im->lock);
905         if (im->tm_running)
906             im->gsquery = im->gsquery && mark;
907         else
908             im->gsquery = mark;
909         changed = !im->gsquery ||
910                 igmp_marksources(im, ntohs(ih3->nsrcs), ih3->srcs);
911         spin_unlock_bh(&im->lock);
912         if (changed)
913             igmp_mod_timer(im, max_delay);
914     }
915     read_unlock(&in_dev->mc_list_lock);
916 }
```

897 最后处理 IGMPv3 指定组和源的查询。

897-903 遍历 IP 配置块中网络设备接口加入的组播组，查找除 224.0.0.1 之外的指定组。

905-910 将是否进行指定组和源查询的标志设置到 gsquery，然后根据查询报文中的源地址信息，标记指定组播组的源地址过滤列表中各源地址是否应答指定组和源的查询。

912-913 如果有需要发送指定组和源的报告，则启动接口定时器。

16.8 成员关系报告

16.8.1 最近离开组播组列表的维护

IGMPv3 的成员关系报告中增加了组记录类型的报告，当某个套接口离开一个组播组时，该组播组中的组播源地址通常为不希望接收组播的源地址，因此对于“组播源列表改变到不希望接收记录”类型的组报告，也需要在网络设备接口上维护那些刚离开的组播组。这些刚离开的组播组配置块以链表的方式保存到当前网络设备配置块中的 mc_tomb 成员上。

在套接口完成离开组播组时，会调用 igmpv3_add_delrec()，根据离开组播组配置信息创建与之一致的信息块添加到 mc_tomb 链表上。

在套接口完成加入组播组时，会调用 igmpv3_del_delrec()，在 mc_tomb 上查找对应的组播组配置块，如果存在，则删除组播地址的被阻止源和组播组配置块。

16.8.2 is_in()

is_in()用来判断指定组播组的特定组播源是否属于特定的组记录类型，参数说明如下：

- pmc，指定的组播组描述块。
- psf，指定组播组的源地址描述块。
- type，组记录类型，用于判断是否属于此组记录类型。
- gdeleted，用于判断是不是刚离开的组播组的标识。
- sdeleted，用于判断是不是被阻止的组播源的标识。

```
227 static int is_in(struct ip_mc_list *pmc, struct ip_sf_list *psf, int type,
228    int gdeleted, int sdeleted)
229 {
230    switch (type) {
231    case IGMPV3_MODE_IS_INCLUDE:
232    case IGMPV3_MODE_IS_EXCLUDE:
233       if (gdeleted || sdeleted)
234          return 0;
235       if (!(pmc->gsquery && !psf->sf_gsresp)) {
236          if (pmc->sfmode == MCAST_INCLUDE)
237             return 1;
238          /* don't include if this source is excluded
239           * in all filters
240           */
241          if (psf->sf_count[MCAST_INCLUDE])
242             return type == IGMPV3_MODE_IS_INCLUDE;
243          return pmc->sfcount[MCAST_EXCLUDE] ==
244             psf->sf_count[MCAST_EXCLUDE];
245       }
246        return 0;
```

满足“当前状态记录”的判断条件如下：

- 用于判断的标志不是离开的组播组，并且也不是被阻塞的组播源。
- 当前的报告不是针对指定组和源的查询，并且源地址没有应答指定组和源的查询。

- 若同时满足以上条件，且满足以下任何一个条件，此时组播组的组播源确定属于指定的组记录类型。
- 组播组的源过滤模式为 INCLUDE。
- INCLUDE 过滤模式的组播源过滤地址的套接口在该接口加入组播组的数量不为 0 且用于判断的类型为 IGMPV3_MODE_IS_INCLUDE。
- EXCLUDE 过滤模式的组播源过滤地址的套接口在该接口加入组播组的数量与 EXCLUDE 过滤模式的组播源过滤地址的套接口在该接口加入组播组的数量相等。

```
247      case IGMPV3_CHANGE_TO_INCLUDE:
248          if (gdeleted || sdeleted)
249              return 0;
250          return psf->sf_count[MCAST_INCLUDE] != 0;
251      case IGMPV3_CHANGE_TO_EXCLUDE:
252          if (gdeleted || sdeleted)
253              return 0;
254          if (pmc->sfcount[MCAST_EXCLUDE] == 0 ||
255              psf->sf_count[MCAST_INCLUDE])
256              return 0;
257          return pmc->sfcount[MCAST_EXCLUDE] ==
258              psf->sf_count[MCAST_EXCLUDE];
```

247-250 满足“过滤模式改变到 INCLUDE 记录”的条件如下：

- 用于判断的标志不是离开的组播组，并且也不是被阻塞的组播源。
- INCLUDE 过滤模式的组播源过滤地址的套接口在该接口加入组播组的数量不为 0。

251-258 满足“过滤模式改变到 EXCLUDE 记录”的条件如下：

- 用于判断的标志不是离开的组播组，并且也不是被阻塞的组播源。
- EXCLUDE 过滤模式的组播源过滤地址的套接口在该接口加入组播组的数量不为 0，并且INCLUDE过滤模式的组播源过滤地址的套接口在该接口加入组播组的数量必须为0。
- EXCLUDE 过滤模式的组播源过滤地址的套接口在该接口加入组播组的数量与 EXCLUDE 过滤模式的组播源过滤地址的套接口在该接口加入组播组的数量相等。

```
259      case IGMPV3_ALLOW_NEW_SOURCES:
260          if (gdeleted || !psf->sf_crcount)
261              return 0;
262          return (pmc->sfmode == MCAST_INCLUDE) ^ sdeleted;
263      case IGMPV3_BLOCK_OLD_SOURCES:
264          if (pmc->sfmode == MCAST_INCLUDE)
265              return gdeleted || (psf->sf_crcount && sdeleted);
266          return psf->sf_crcount && !gdeleted && !sdeleted;
267      }
268      return 0;
269  }
```

259-262 满足“希望接收的发往某一组播地址的组记录”的条件为：

- 用于判断的不是离开的组播组，并且系统当前还可重传“源列表改变记录”报告。
- 如果这是对一个 INCLUDE 列表的改变，那么这些地址会被添加到列表中，如果是对一个 EXCLUDE 列表的改变，那么这些地址会被从列表中删除。

263-266 满足“不希望接收的发往某一组播地址的组记录”的条件为：

- 系统还可重传“源列表改变记录”报告。
- 如果是对一个 INCLUDE 列表的改变，用于判断的不是离开的组播组和阻塞的组播源。
- 如果是对一个 EXCLUDE 列表的改变，用于判断的必须不是离开的组播组和被阻塞的组播源。

16.8.3 add_grec()

add_grec()用来组成或发送各种类型的 V3 报告报文。参数说明如下：

- skb，存储 V3 报告报文的 SKB。当为 NULL 时，则在必要时会自动分配。
- pmc，组播组信息。
- type，V3 报告中组记录的类型，参见 16.4.4 节。
- gdeleted，是否有套接口离开的组播组的标志。
- sdeleted，是否有源地址被阻止的标志。

```
391 static struct sk_buff *add_grec(struct sk_buff *skb, struct ip_mc_list *pmc,
392     int type, int gdeleted, int sdeleted)
393 {
394     struct net_device *dev = pmc->interface->dev;
395     struct igmpv3_report *pih;
396     struct igmpv3_grec *pgr = NULL;
397     struct ip_sf_list *psf, *psf_next, *psf_prev, **psf_list;
398     int scount, stotal, first, isquery, truncate;
399
400     if (pmc->multiaddr == IGMP_ALL_HOSTS)
401         return skb;
```

393 根据组播组获取对应网络设备上的 IP 配置块。

398 scount 用于标记每个记录的源地址数；stotal 用于标记所有记录中总源地址数。

400-401 对于特殊的 224.0.0.1 组播组无需作处理。

```
403     isquery = type == IGMPV3_MODE_IS_INCLUDE ||
404         type == IGMPV3_MODE_IS_EXCLUDE;
405     truncate = type == IGMPV3_MODE_IS_EXCLUDE ||
406         type == IGMPV3_CHANGE_TO_EXCLUDE;
407
408     stotal = scount = 0;
```

403-404 检测发送报告是否是对普通查询的报告。

405-406 检测当前组成报告的组记录类型是否是 IGMPV3_MODE_IS_EXCLUDE 或 IGMPV3_CHANGE_TO_EXCLUDE，并设置标志。

408 清零 stotal 和 scount。

```
410     psf_list = sdeleted ? &pmc->tomb : &pmc->sources;
411
412     if (!*psf_list)
413         goto empty_source;
414
415     pih = skb ? (struct igmpv3_report *)skb->h.igmph : NULL;
```

410-412 根据源是否被阻塞标识，确定是从 tomb 还是从 sources 获取源地址。如果获取

失败，则不处理源地址。

415 如果当前 SKB 有效，则获取 IGMPv3 报文的首部。

```
418     if (truncate) {
419        if (pih && pih->ngrec &&
420           AVAILABLE(skb) < grec_size(pmc, type, gdeleted, sdeleted)) {
421           if (skb)
422              igmpv3_sendpack(skb);
423           skb = igmpv3_newpack(dev, dev->mtu);
424        }
425     }
```

对于 IGMPV3_MODE_IS_EXCLUDE 或 IGMPV3_CHANGE_TO_EXCLUDE 类型的组记录，如果当前报告报文有效，同时当前报文已容纳不下指定组播组、指定记录类型的组记录时，则将当前报文输出，然后分配一个新的报文。

grec_size()用来计算指定组播组、指定记录类型的组记录长度。

```
426     first = 1;
427     psf_prev = NULL;
428     for (psf=*psf_list; psf; psf=psf_next) {
429        __be32 *psrc;
430
431        psf_next = psf->sf_next;
432
433        if (!is_in(pmc, psf, type, gdeleted, sdeleted)) {
434           psf_prev = psf;
435           continue;
436        }
437
438        /* clear marks on query responses */
439        if (isquery)
440           psf->sf_gsresp = 0;
441
442        if (AVAILABLE(skb) < sizeof(__be32) +
443           first*sizeof(struct igmpv3_grec)) {
444           if (truncate && !first)
445              break;    /* truncate these */
446           if (pgr)
447              pgr->grec_nsrcs = htons(scount);
448           if (skb)
449              igmpv3_sendpack(skb);
450           skb = igmpv3_newpack(dev, dev->mtu);
451           first = 1;
452           scount = 0;
453        }
454        if (first) {
455           skb = add_grhead(skb, pmc, type, &pgr);
456           first = 0;
457        }
458        psrc = (__be32 *)skb_put(skb, sizeof(__be32));
459        *psrc = psf->sf_inaddr;
460        scount++; stotal++;
461        if ((type == IGMPV3_ALLOW_NEW_SOURCES ||
462             type == IGMPV3_BLOCK_OLD_SOURCES) && psf->sf_crcount) {
```

```
463             psf->sf_crcount--;
464             if ((sdeleted || gdeleted) && psf->sf_crcount == 0) {
465                 if (psf_prev)
466                     psf_prev->sf_next = psf->sf_next;
467                 else
468                     *psf_list = psf->sf_next;
469                 kfree(psf);
470                 continue;
471             }
472         }
473         psf_prev = psf;
474     }
```

426 表示需要添加一条新的组记录。

428-431 遍历该组播信息块中的源列表，添加组记录。

433-436 判断指定组播组的特定组播源是否是特定的组记录类型。

439-440 如果是普通查询，则清除源地址是否应答指定组和源查询的标志 sf_gsresp。

442-453 如果当前发送 IGMP 报文的 SKB 已不够容纳一条记录，则将原先的报文输出，然后分配一个新的 SKB。或者，如果没有 SKB，则直接分配新的 SKB。

454-457 如果新分配的 SKB 并且尚未添加记录首部，则填充组记录首部。

458-460 添加源地址到记录中，并计数。

461-472 对于“源列表改变记录”类型的记录，递减还可重传该类型的报告的次数。如果递减后次数为 0，说明不能再重传了，删除该记录。

```
476 empty_source:
477     if (!stotal) {
478         if (type == IGMPV3_ALLOW_NEW_SOURCES ||
479             type == IGMPV3_BLOCK_OLD_SOURCES)
480             return skb;
481         if (pmc->crcount || isquery) {
482             /* make sure we have room for group header */
483             if (skb && AVAILABLE(skb)<sizeof(struct igmpv3_grec)) {
484                 igmpv3_sendpack(skb);
485                 skb = NULL; /* add_grhead will get a new one */
486             }
487             skb = add_grhead(skb, pmc, type, &pgr);
488         }
489     }
490     if (pgr)
491         pgr->grec_nsrcs = htons(scount);
492
493     if (isquery)
494         pmc->gsquery = 0;    /* clear query state on report */
495     return skb;
496 }
```

477-489 如果没有源地址，并且是“源列表改变记录”类型记录，则直接返回该 SKB。而对于普通查询或还可发送 V3 过滤模式改变记录类型报告，如果当前用于发送 IGMP 报文的 SKB 已不够容纳一条记录，则先将该报文输出，然后分配一个新的 SKB，否则直接分配新的 SKB 并填充新记录首部。

490-491 设置报文中组记录数。

493-494 最后在返回 SKB 前，如果是普通查询，则需清除指定组或源查询标志。

16.8.4　普通查询的报告

1. igmp_gq_start_timer()

IGMPv3 的普通查询报告是由普通查询定时器激活的，在接收到 IGMPv3 普通查询消息之后会调用 igmp_gq_start_timer()。

此函数用来启动普通查询定时器，首先获取下次激活普通查询定时器的时间，然后标识普通查询定时器已启动，最后重新设置普通查询定时器的激活时间。

```
185 static void igmp_gq_start_timer(struct in_device *in_dev)
186 {
187     int tv = net_random() % in_dev->mr_maxdelay;
188
189     in_dev->mr_gq_running = 1;
190     if (!mod_timer(&in_dev->mr_gq_timer, jiffies+tv+2))
191         in_dev_hold(in_dev);
192 }
```

2. igmp_gq_timer_expire()

igmp_gq_timer_expire()是普通查询定时器 mr_gq_timer 的例程，在 IGMP 模块初始化函数 ip_mc_init_dev()中设置，当普通查询定时器被激活时，就会调用该函数。先取消普通查询定时器已启动标识，然后调用 igmpv3_send_report()发送 V3 当前状态记录类型报告。

```
694 static void igmp_gq_timer_expire(unsigned long data)
695 {
696     struct in_device *in_dev = (struct in_device *)data;
697
698     in_dev->mr_gq_running = 0;
699     igmpv3_send_report(in_dev, NULL);
700     __in_dev_put(in_dev);
701 }
```

3. igmpv3_send_report()

igmpv3_send_report()用来发送 V3 当前状态记录类型报告。如果没有指定组播组，则需报告网络接口加入的除 224.0.0.1 外的所有组播组；否则只需报告指定组播组。组装 V3 报告由 add_grec()完成，然后调用 igmpv3_sendpack()将组装好的报文输出，在组装过程中，如果发现报文已经完整，也会将其输出。参数说明如下：

- in_dev，加入组播组网络接口的 IP 配置块。
- pmc，要报告的指定组播组，如果为 NULL 则报告网络接口加入的所有组播组。

```
498 static int igmpv3_send_report(struct in_device *in_dev, struct ip_mc_list *pmc)
499 {
500     struct sk_buff *skb = NULL;
501     int type;
502
503     if (!pmc) {
504         read_lock(&in_dev->mc_list_lock);
```

```
505         for (pmc=in_dev->mc_list; pmc; pmc=pmc->next) {
506             if (pmc->multiaddr == IGMP_ALL_HOSTS)
507                 continue;
508             spin_lock_bh(&pmc->lock);
509             if (pmc->sfcount[MCAST_EXCLUDE])
510                 type = IGMPV3_MODE_IS_EXCLUDE;
511             else
512                 type = IGMPV3_MODE_IS_INCLUDE;
513             skb = add_grec(skb, pmc, type, 0, 0);
514             spin_unlock_bh(&pmc->lock);
515         }
516         read_unlock(&in_dev->mc_list_lock);
517     } else {
518         spin_lock_bh(&pmc->lock);
519         if (pmc->sfcount[MCAST_EXCLUDE])
520             type = IGMPV3_MODE_IS_EXCLUDE;
521         else
522             type = IGMPV3_MODE_IS_INCLUDE;
523         skb = add_grec(skb, pmc, type, 0, 0);
524         spin_unlock_bh(&pmc->lock);
525     }
526     if (!skb)
527         return 0;
528     return igmpv3_sendpack(skb);
529 }
```

503-516 如果没有指定组播组，则说明需报告网络接口加入的所有组播组。因此，遍历该网络接口加入的除 224.0.0.1 外所有的组播组，根据不同过滤模式（EXCLUDE 或 INCLUDE）的套接口在该接口加入组播组的数量来确定记录类型，然后调用 add_grec()组成或发送各种类型的 V3 报告报文。

517-525 如果有指定组播组，则根据不同过滤模式（EXCLUDE 或 INCLUDE）的套接口在该接口加入指定组播组的数量确定记录类型，然后调用 add_grec()组成或发送各种类型的 V3 报告报文。

526-528 如果 add_grec()返回的 SKB 有效，则将其输出。

16.8.5 V1 和 V2 的报告以及 V3 的当前状态记录报告

1. igmp_start_timer()

igmp_start_timer()用来启动组查询定时器，首先获取下次激活组查询定时器的时间，然后标识组查询定时器已启动，最后重新设置组查询定时器的激活时间。

```
176 static void igmp_start_timer(struct ip_mc_list *im, int max_delay)
177 {
178     int tv=net_random() % max_delay;
179
180     im->tm_running=1;
181     if (!mod_timer(&im->timer, jiffies+tv+2))
182         atomic_inc(&im->refcnt);
183 }
```

2. igmp_mod_timer()

igmp_mod_timer()用于启动或重新设定组查询定时器。首先将发送主动报告的次数 unsolicit_count

清零。然后删除组查询定时器，如果删除成功，这说明之前已启动了组查询定时器，且组查询定时器原激活时间早于此次修改的上限，则仍然使用原激活时间，否则重新设定激活时间。

```
202 static void igmp_mod_timer(struct ip_mc_list *im, int max_delay)
203 {
204     spin_lock_bh(&im->lock);
205     im->unsolicit_count = 0;
206     if (del_timer(&im->timer)) {
207         if ((long)(im->timer.expires-jiffies) < max_delay) {
208             add_timer(&im->timer);
209             im->tm_running=1;
210             spin_unlock_bh(&im->lock);
211             return;
212         }
213         atomic_dec(&im->refcnt);
214     }
215     igmp_start_timer(im, max_delay);
216     spin_unlock_bh(&im->lock);
217 }
```

3. igmp_stop_timer()

igmp_stop_timer()用来停止已启动的组查询定时器。

```
164 static __inline__ void igmp_stop_timer(struct ip_mc_list *im)
165 {
166     spin_lock_bh(&im->lock);
167     if (del_timer(&im->timer))
168         atomic_dec(&im->refcnt);
169     im->tm_running=0;
170     im->reporter = 0;
171     im->unsolicit_count = 0;
172     spin_unlock_bh(&im->lock);
173 }
```

4. igmp_timer_expire()

igmp_timer_expire()是普通查询定时器的例程，当普通查询定时器被激活时，会调用该函数。首先取消普通查询定时器已启动标识，然后调用 igmpv3_send_report()发送 V3 当前状态记录类型报告。

```
725 static void igmp_timer_expire(unsigned long data)
726 {
727     struct ip_mc_list *im=(struct ip_mc_list *)data;
728     struct in_device *in_dev = im->interface;
729
730     spin_lock(&im->lock);
731     im->tm_running=0;
732
733     if (im->unsolicit_count) {
734         im->unsolicit_count--;
735         igmp_start_timer(im, IGMP_Unsolicited_Report_Interval);
736     }
737     im->reporter = 1;
738     spin_unlock(&im->lock);
```

```
739
740     if (IGMP_V1_SEEN(in_dev))
741         igmp_send_report(in_dev, im, IGMP_HOST_MEMBERSHIP_REPORT);
742     else if (IGMP_V2_SEEN(in_dev))
743         igmp_send_report(in_dev, im, IGMPV2_HOST_MEMBERSHIP_REPORT);
744     else
745         igmp_send_report(in_dev, im, IGMPV3_HOST_MEMBERSHIP_REPORT);
746
747     ip_ma_put(im);
748 }
```

5. igmp_send_report()

igmp_send_report()用来发送各个版本的报告报文和离开报文。参数说明如下：

- in_dev，网络接口的 IP 配置块。
- pmc，加入或离开组的信息块。
- type，IGMP 报文的类型，包括各个版本的报告报文和离开报文。

```
625 static int igmp_send_report(struct in_device *in_dev, struct ip_mc_list *pmc,
626     int type)
627 {
628     struct sk_buff *skb;
629     struct iphdr *iph;
630     struct igmphdr *ih;
631     struct rtable *rt;
632     struct net_device *dev = in_dev->dev;
633     __be32    group = pmc ? pmc->multiaddr : 0;
634     __be32    dst;
635
636     if (type == IGMPV3_HOST_MEMBERSHIP_REPORT)
637         return igmpv3_send_report(in_dev, pmc);
638     else if (type == IGMP_HOST_LEAVE_MESSAGE)
639         dst = IGMP_ALL_ROUTER;
640     else
641         dst = group;
```

如果是 V3 报告，则调用 igmpv3_send_report()来处理，其他三种类型的报文则继续由后面的代码处理。如果是离开组的报文，则目的组播组地址应为 224.0.0.2，告知所在子网内所有的路由器；其他情况的报文则目的地址为所加入的组播组地址。

```
643     {
644         struct flowi fl = { .oif = dev->ifindex,
645                     .nl_u = { .ip4_u = { .daddr = dst } },
646                     .proto = IPPROTO_IGMP };
647         if (ip_route_output_key(&rt, &fl))
648             return -1;
649     }
650     if (rt->rt_src == 0) {
651         ip_rt_put(rt);
652         return -1;
653     }
```

由输出设备和目的地址在路由子系统中查找路由缓存项，只有查找命中才能输出指定的 IGMP 报文。

```
655     skb=alloc_skb(IGMP_SIZE+LL_RESERVED_SPACE(dev), GFP_ATOMIC);
656     if (skb == NULL) {
657         ip_rt_put(rt);
658         return -1;
659     }
660
661     skb->dst = &rt->u.dst;
662
663     skb_reserve(skb, LL_RESERVED_SPACE(dev));
664
665     skb->nh.iph = iph = (struct iphdr *)skb_put(skb, sizeof(struct iphdr)+4);
666
667     iph->version  = 4;
668     iph->ihl      = (sizeof(struct iphdr)+4)>>2;
669     iph->tos      = 0xc0;
670     iph->frag_off = htons(IP_DF);
671     iph->ttl      = 1;
672     iph->daddr    = dst;
673     iph->saddr    = rt->rt_src;
674     iph->protocol = IPPROTO_IGMP;
675     iph->tot_len  = htons(IGMP_SIZE);
676     ip_select_ident(iph, &rt->u.dst, NULL);
677     ((u8*)&iph[1])[0] = IPOPT_RA;
678     ((u8*)&iph[1])[1] = 4;
679     ((u8*)&iph[1])[2] = 0;
680     ((u8*)&iph[1])[3] = 0;
681     ip_send_check(iph);
682
683     ih = (struct igmphdr *)skb_put(skb, sizeof(struct igmphdr));
684     ih->type=type;
685     ih->code=0;
686     ih->csum=0;
687     ih->group=group;
688     ih->csum=ip_compute_csum((void *)ih, sizeof(struct igmphdr));
689
690     return NF_HOOK(PF_INET, NF_IP_LOCAL_OUT, skb, NULL, rt->u.dst.dev,
691                dst_output);
692 }
```

创建 IGMP 报文，设置报文目的地址、IGMP 报文所在 IP 报文首部各个域以及 IGMP 报文各个域，最后将其输出。

16.8.6　主动发送组关系报告

1. igmp_ifc_event()

当一个网络设备通过 IGMPv3 协议加入或离开某个组播组，或是对指定组播组开通或阻塞某个组播源时，会通过 igmp_ifc_event()来启动 mr_ifc_timer 定时器，主动报告相关信息。

首先判断当前使用的 IGMP 协议版本，如果不是 V3，则不作任何处理；然后从健壮变量中得到最多可重传 IGMPv3 报告报文的次数，最后启动 mr_ifc_timer 定时器。

```
715 static void igmp_ifc_event(struct in_device *in_dev)
716 {
717     if (IGMP_V1_SEEN(in_dev) || IGMP_V2_SEEN(in_dev))
718         return;
```

```
719     in_dev->mr_ifc_count = in_dev->mr_qrv ? in_dev->mr_qrv :
720        IGMP_Unsolicited_Report_Count;
721     igmp_ifc_start_timer(in_dev, 1);
722 }
```

2. igmp_ifc_timer_expire()

igmp_ifc_timer_expire()是 mr_ifc_timer 定时器的例程，在 IGMP 模块初始化函数 ip_mc_init_dev()中设置，当定时器触发时会调用该函数来报告已离开的组以及被阻止的源，如果还可重传 IGMPv3 报告报文，则再次启动 mr_ifc_timer 定时器。

```
703 static void igmp_ifc_timer_expire(unsigned long data)
704 {
705     struct in_device *in_dev = (struct in_device *)data;
706
707     igmpv3_send_cr(in_dev);
708     if (in_dev->mr_ifc_count) {
709        in_dev->mr_ifc_count--;
710        igmp_ifc_start_timer(in_dev, IGMP_Unsolicited_Report_Interval);
711     }
712     __in_dev_put(in_dev);
713 }
```

3. igmpv3_send_cr()

igmpv3_send_cr()用于报告已离开的组以及被阻止的源，在 mr_ifc_timer 定时器激活时由 igmp_ifc_timer_expire()调用。

```
552 static void igmpv3_send_cr(struct in_device *in_dev)
553 {
554     struct ip_mc_list *pmc, *pmc_prev, *pmc_next;
555     struct sk_buff *skb = NULL;
556     int type, dtype;
557
558     read_lock(&in_dev->mc_list_lock);
559     spin_lock_bh(&in_dev->mc_tomb_lock);
560
561     /* deleted MCA's */
562     pmc_prev = NULL;
563     for (pmc=in_dev->mc_tomb; pmc; pmc=pmc_next) {
564        pmc_next = pmc->next;
565        if (pmc->sfmode == MCAST_INCLUDE) {
566           type = IGMPV3_BLOCK_OLD_SOURCES;
567           dtype = IGMPV3_BLOCK_OLD_SOURCES;
568           skb = add_grec(skb, pmc, type, 1, 0);
569           skb = add_grec(skb, pmc, dtype, 1, 1);
570        }
571        if (pmc->crcount) {
572           if (pmc->sfmode == MCAST_EXCLUDE) {
573              type = IGMPV3_CHANGE_TO_INCLUDE;
574              skb = add_grec(skb, pmc, type, 1, 0);
575           }
576           pmc->crcount--;
577           if (pmc->crcount == 0) {
578              igmpv3_clear_zeros(&pmc->tomb);
579              igmpv3_clear_zeros(&pmc->sources);
```

```
580             }
581         }
582         if (pmc->crcount == 0 && !pmc->tomb && !pmc->sources) {
583             if (pmc_prev)
584                 pmc_prev->next = pmc_next;
585             else
586                 in_dev->mc_tomb = pmc_next;
587             in_dev_put(pmc->interface);
588             kfree(pmc);
589         } else
590             pmc_prev = pmc;
591     }
592     spin_unlock_bh(&in_dev->mc_tomb_lock);
```

563-564 遍历已离开的组，获取组信息块，处理其中的组播源。

565-570 添加或发送被阻止源记录报告。

571-581 如果当前还可以发送 V3 过滤模式改变记录类型报告，则添加或发送过滤模式改变到 INCLUDE 记录报告。一旦发送次数使用完，即将相关源信息删除。

582-590 如果目前已不能发送 V3 过滤模式改变记录类型报告，且不存在相关源信息，则将该组播组信息块删除并释放。

```
594     /* change recs */
595     for (pmc=in_dev->mc_list; pmc; pmc=pmc->next) {
596         spin_lock_bh(&pmc->lock);
597         if (pmc->sfcount[MCAST_EXCLUDE]) {
598             type = IGMPV3_BLOCK_OLD_SOURCES;
599             dtype = IGMPV3_ALLOW_NEW_SOURCES;
600         } else {
601             type = IGMPV3_ALLOW_NEW_SOURCES;
602             dtype = IGMPV3_BLOCK_OLD_SOURCES;
603         }
604         skb = add_grec(skb, pmc, type, 0, 0);
605         skb = add_grec(skb, pmc, dtype, 0, 1);    /* deleted sources */
606
607         /* filter mode changes */
608         if (pmc->crcount) {
609             if (pmc->sfmode == MCAST_EXCLUDE)
610                 type = IGMPV3_CHANGE_TO_EXCLUDE;
611             else
612                 type = IGMPV3_CHANGE_TO_INCLUDE;
613             skb = add_grec(skb, pmc, type, 0, 0);
614             pmc->crcount--;
615         }
616         spin_unlock_bh(&pmc->lock);
617     }
618     read_unlock(&in_dev->mc_list_lock);
```

595 遍历已加入的组，获取组信息块，处理其中的组播源。

597-605 添加或发送被阻止源和新开通源记录报告。

608-615 如果目前还可发送 V3 过滤模式改变记录类型报告，则添加或发送过滤模式改变记录报告。

```
620     if (!skb)
```

```
621         return;
622     (void) igmpv3_sendpack(skb);
623 }
```

如果 SKB 有效，则调用 igmpv3_sendpack()将其输出。

16.9 维护套接口组播状态

为了能实现对 IGMPv3 中增加的“源过滤”的支持，需维护两个组播接收状态：一是套接口组播状态，二是网络设备组播状态（参见 16.10 节）。

当套接口新加入一个组播组时，系统为这个套接口记录期望的组播接收状态，该状态概念上由网络接口，组播地址，过滤模式和一组源列表记录组成。套接口的状态会变化，见图 16-6。

如果所请求的过滤模式是 INCLUDE，且所请求的源列表为空，那么如果存在与所请求的接口和组播地址相关的入口，则将之删除；否则忽略请求。

如果所请求的过滤模式是 EXCLUDE，且所请求的源列表非空，那么如果存在与请求的接口和组播地址相关的入口，即修改成含有所请求的过滤模式和源列表；否则就根据参数指定的请求创建一个新的入口。

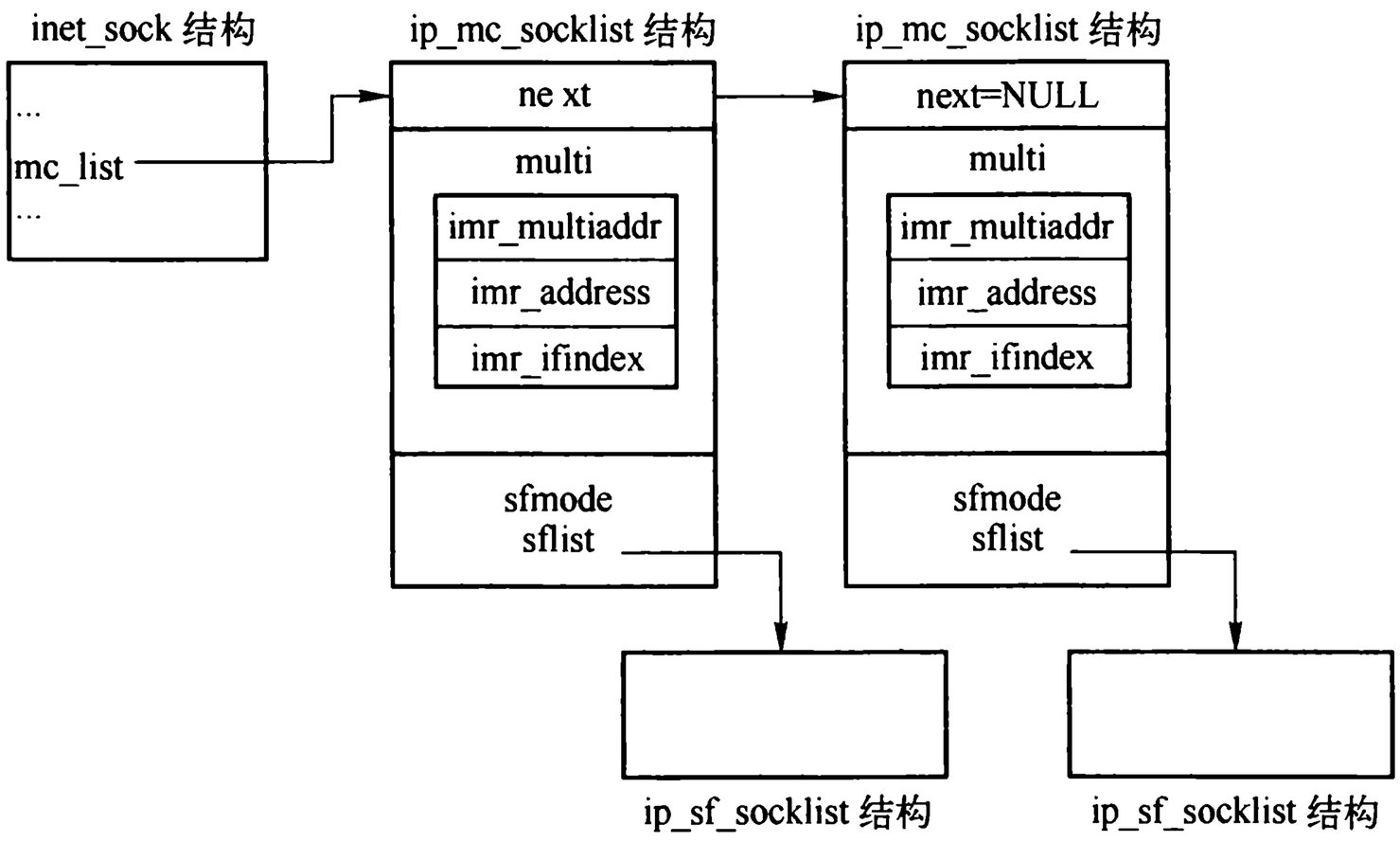

图 16-6 套接口组播状态维护结构

ip_mc_socklist 结构用来描述所属套接口已加入的组播组，以链表的形式链接在传输控制块的 mc_list 上。

```
152 struct ip_mc_socklist
153 {
154     struct ip_mc_socklist   *next;
155     struct ip_mreqn         multi;
156     unsigned int            sfmode;         /* MCAST_{INCLUDE,EXCLUDE} */
157     struct ip_sf_socklist   *sflist;
158 };
```

154 struct ip_mc_socklist *next

用来将多个 ip_mc_socklist 实例链接起来形成链表。

155 struct ip_mreqn multi

对应组播组信息，包括组播地址，网络接口及其本地地址。

156 unsigned int sfmode

组播源过滤模式，见表 16-7。

表 16-7　sfmode

sfmode	描述
MCAST_INCLUDE	只能接收组播源为源列表中的组播报文
MCAST_EXCLUDE	只能接收组播源为源列表中除外的组播报文

157 struct ip_sf_socklist *sflist

源列表，由 ip_sf_socklist 结构描述。源列表的过滤模式见表 16-7。

```
136 struct ip_sf_socklist
137 {
138     unsigned int        sl_max;
139     unsigned int        sl_count;
140     __be32              sl_addr[0];
141 };
```

138 unsigned sl_max

当前分配的空间能存储源地址数上限。

139 unsigned int sl_count

当前存储源地址数。

140 __be32 sl_addr[0]

通过动态分配，用于存储源地址的空间。

16.9.1　套接口加入组播组

ip_mc_join_group()将套接口加入组播组，用于维护套接口加入组播组的状态。完成后最后调用 ip_mc_inc_group()，将指定网络设备加入到组播组中。

```
1717 int ip_mc_join_group(struct sock *sk , struct ip_mreqn *imr)
1718 {
1719     int err;
1720     __be32 addr = imr->imr_multiaddr.s_addr;
1721     struct ip_mc_socklist *iml=NULL, *i;
1722     struct in_device *in_dev;
1723     struct inet_sock *inet = inet_sk(sk);
1724     int ifindex;
1725     int count = 0;
1726
1727     if (!MULTICAST(addr))
1728         return -EINVAL;
```

1727-1728 如果参数给定的要加入的组播组地址不是组地址，则返回 EINVAL 错误。

```
1730     rtnl_lock();
1731
1732     in_dev = ip_mc_find_dev(imr);
```

```
1733
1734    if (!in_dev) {
1735        iml = NULL;
1736        err = -ENODEV;
1737        goto done;
1738    }
1739
1740    err = -EADDRINUSE;
1741    ifindex = imr->imr_ifindex;
1742    for (i = inet->mc_list; i; i = i->next) {
1743        if (i->multi.imr_multiaddr.s_addr == addr &&
1744            i->multi.imr_ifindex == ifindex)
1745            goto done;
1746        count++;
1747    }
```

1732-1738 如果获取组播组网络设备接口的 IPv4 in_device 结构失败，则可能找不对对应的网络设备，返回 ENODEV 错误。

1740-1747 在对应的传输控制块中查找该套接口是否已加入该组，如果是则返回 EADDRINUSE 错误。在查找的同时，还会统计该套接口加入组播组数。

```
1748    err = -ENOBUFS;
1749    if (count >= sysctl_igmp_max_memberships)
1750        goto done;
```

该套接口加入组播组数达到了系统设置的上限，则不能再加入某个组了，返回 ENOBUFS。

```
1751    iml = sock_kmalloc(sk,sizeof(*iml),GFP_KERNEL);
1752    if (iml == NULL)
1753        goto done;
1754
1755    memcpy(&iml->multi, imr, sizeof(*imr));
1756    iml->next = inet->mc_list;
1757    iml->sflist = NULL;
1758    iml->sfmode = MCAST_EXCLUDE;
1759    inet->mc_list = iml;
```

1751-1759 分配 ip_mc_socklist 结构将其加入到传输控制块的 mc_list 链表中。设置过滤模式为 MCAST_EXCLUDE 并且源地址为空，表示接收所有源地址发送指定组播报文。

```
1760    ip_mc_inc_group(in_dev, addr);
1761    err = 0;
1762 done:
1763    rtnl_unlock();
1764    return err;
1765 }
```

1760 调用 ip_mc_inc_group()更新网络设备接口的组播状态。

16.9.2 套接口离开组播组

ip_mc_leave_group()用来使指定的套接口离开指定的组播组。

```
1789 int ip_mc_leave_group(struct sock *sk, struct ip_mreqn *imr)
1790 {
1791     struct inet_sock *inet = inet_sk(sk);
1792     struct ip_mc_socklist *iml, **imlp;
1793     struct in_device *in_dev;
1794     __be32 group = imr->imr_multiaddr.s_addr;
1795     u32 ifindex;
1796     int ret = -EADDRNOTAVAIL;
1797
1798     rtnl_lock();
1799     in_dev = ip_mc_find_dev(imr);
1800     ifindex = imr->imr_ifindex;
1801     for (imlp = &inet->mc_list; (iml = *imlp) != NULL; imlp = &iml->next){
1802         if (iml->multi.imr_multiaddr.s_addr != group)
1803             continue;
1804         if (ifindex) {
1805             if (iml->multi.imr_ifindex != ifindex)
1806                 continue;
1807         } else if (imr->imr_address.s_addr .&& imr->imr_address.s_addr !=
1808                 iml->multi.imr_address.s_addr)
1809             continue;
1810
1811         (void) ip_mc_leave_src(sk, iml, in_dev);
1812
1813         *imlp = iml->next;
1814
1815         if (in_dev)
1816             ip_mc_dec_group(in_dev, group);
1817         rtnl_unlock();
1818         sock_kfree_s(sk, iml, sizeof(*iml));
1819         return 0;
1820     }
1821     if (!in_dev)
1822         ret = -ENODEV;
1823     rtnl_unlock();
1824     return ret;
1825 }
```

1794 由参数给出的组播组信息结构中获取待离开的组播组地址。

1799-1800 获取组播组的 in_device 结构和网络设备索引备。

1801-1820 遍历套接口所加入的组播组列表，查找待离开组播组的信息，找到后，将其从套接口中删除，说明套接口离开了该组播组。然后接口离开该组播组，删除设备 IP 配置块中相关的组播信息，以及网络设备中该组播组对应的硬件地址。

16.10　维护网络设备组播状态

除了每一个套接口的组播接收状态，系统同时也要为每一个网络设备接口维护一个组播接收状态。这个状态概念上由组播地址、过滤模式和一组源列表记录组成。对于一个给定的网络设备接口，每个组播地址最多存在一条记录。网络设备接口状态是从套接口状态继承过来的，当不同的套接口在同一个组播地址和接口上具有不同的过滤模式和源列表时，两者可能就不同了。网络接口组播状态维护结构见图 16-7。

例如，A 和 B 两个套接口通过同一个接口加入了相同的组播组，并且源过滤模式也相同，只是各自的源列表不同。这个接口组播源列表为 A、B 两个套接口组播源列表的并集。此时，当 IP 层收到一个来自接口的 IP 组播数据报后，接下来是该发往 A 还是 B 套接口上侦听的进程，取决于套接口的组播接收状态。

从套接口状态继承接口状态的基本规则是：对每一个出现在套接口状态中不同的网络设备接口和组播地址组合，为该网络设备接口上的该组播地址创建一个网络设备接口记录，就像所有的套接口的记录中都含有网络接口和组播地址组合。

如果一个套接口记录含有 EXCLUDE 模式，那么网络设备记录的过滤模式就是 EXCLUDE，并且网络设备记录的源列表是所有 EXCLUDE 模式套接口记录源列表的交集减去所有 INCLUDE 模式套接口源列表的交集。例如，在接口 i 上，组播地址 m 的套接口记录见表 16-8。

表 16-8　接口上的组播地址 m 的套接口记录

套接口	网络设备	过滤模式	组播地址
s1	i	EXCLUDE	a,b,c,d
s2	i	EXCLUDE	b,c,d,e
s3	i	INCLUDE	d,e,f

因此相应的，接口 i 上的网络设备记录就是 EXCLUDE{a,b,c}。

如果所有的记录的过滤模式都为 INCLUDE，那么接口记录的过滤模式就是 INCLUDE，接口记录的源列表就是所有套接口记录源列表的合集。

如果一个组的所有套接口都是 EXCLUDE 状态，那么具体的实现中，就不能把表示这个组的接口记录表示为 EXCLUDE 模式。如果在计算一个接口的接口记录时受到系统资源的限制，那么应立即向请求该操作的应用程序返回一个错误。

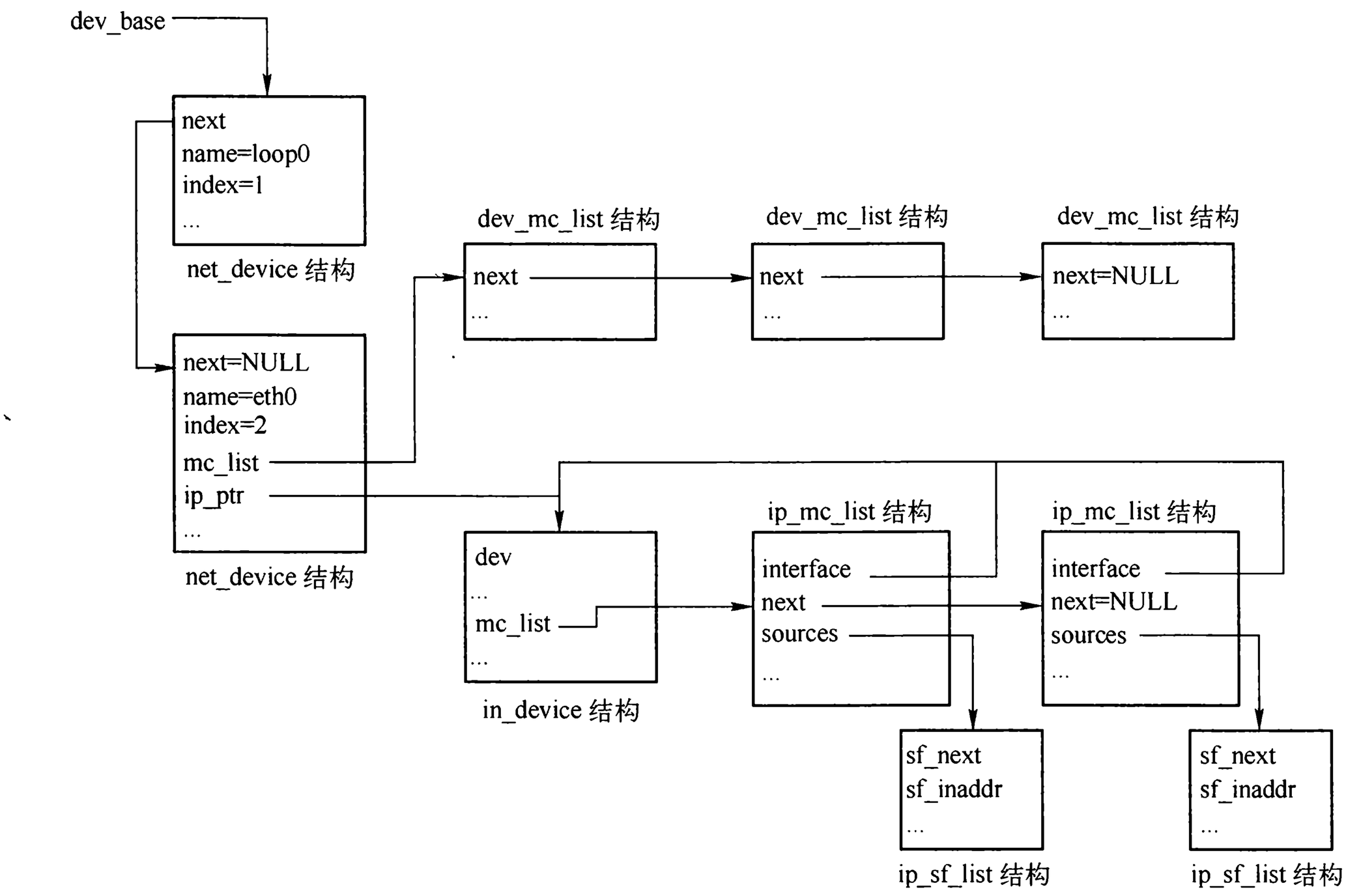

图 16-7　网络接口组播状态维护结构

16.10.1　被阻止的组播源列表的维护

IGMPv3 的成员关系报告中增加了组记录类型的报告，对于“组播源列表改变到不希望接收记录”类型的组报告，需要在网络设备接口上维护那些被阻止的组播源地址。这些被阻止的组播源地址以链表的形式保存到每个组播组配置块的 tomb 成员上。

在通过 IP_MSFILTER 等选项设置过滤组播源地址，通过 IP_BLOCK_SOURCE 等选项阻塞组播源时，以及通过 IP_DROP_MEMBERSHIP 等选项离开某个组播组时，如果没有套接口在使用原先设置的过滤组播源地址，则会将其移动到 tomb，该过程由 ip_mc_del_src()实现。

在给指定的组播组添加或删除组播源过滤源地址之前，会调用 sf_markstate()来记录该组播组上各个过滤源地址的组播源地址状态，标记对应组播源地址的过滤是否完全处于同一种过滤模式。

在完成添加或删除组播源过滤源地址之后，如果该组播组的源过滤模式未发生变化，则会获取当前组播组上各个过滤源地址的组播源地址状态，并根据与操作之前的状态相比较的结果更新 tomb。组播源地址从不完全处于同一种过滤模式到完全处于同一种过滤模式，会从 tomb 中删除一项，反之从完全处于同一种过滤模式到不完全处于同一种过滤模式，则会在 tomb 中添加一项。

16.10.2　网络设备加入组播组

为了能实现 IGMPv3 中增加的对“源过滤”的支持，需要维护两个组播接收状态：一是套接口组播接收状态，二是网络设备接口组播接收状态。而后者概念上由如下形式的一组记录组成：（组播地址、过滤模式、源地址）。

如图 16-8 所示，在网络设备章节中了解到，描述网络设备的 net_device 结构，其 ip_ptr 成员是一个指向 IPv4 协议族数据块 in_device 结构的指针，而 in_device 结构的 mc_list 成员就是用于维护网络设备接口组播接收状态的，它指向一个 ip_mc_list 结构，从而构成由 ip_mc_list 结构组成的链表。

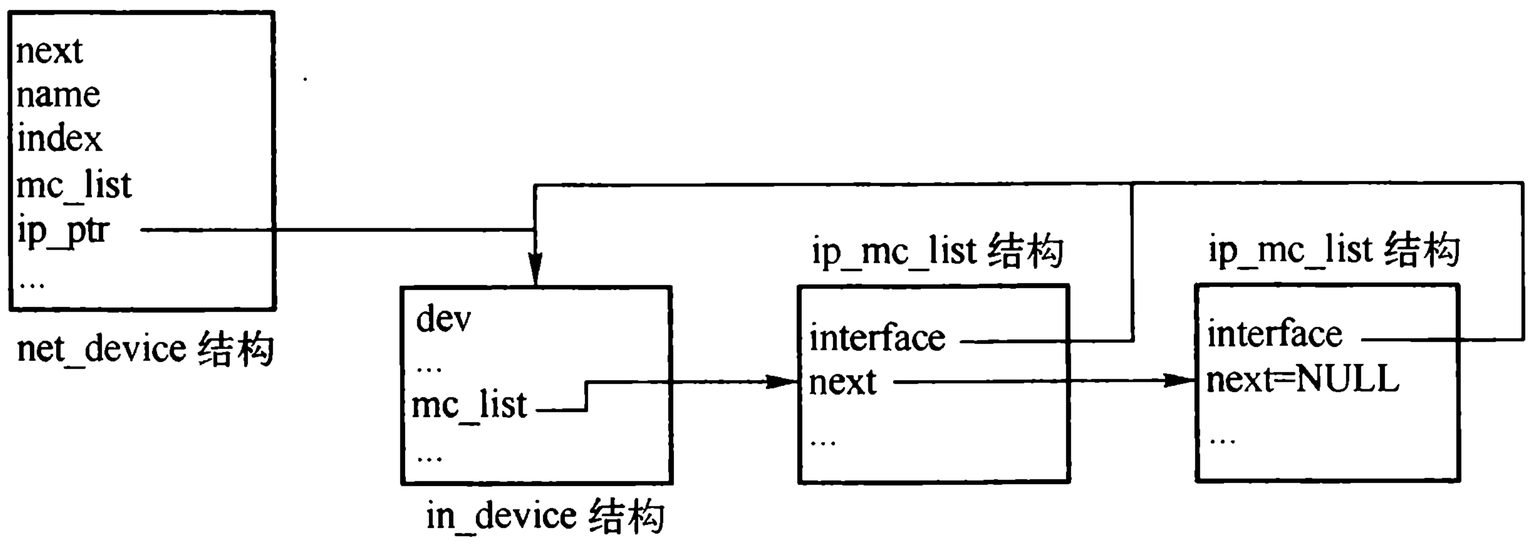

图 16-8　in_device 结构中的 mc_list

1. ip_mc_inc_group()

ip_mc_inc_group()实现将指定的网络设备加入到指定的组播组中。当一个套接口完成加入到一个组播组后，便会调用该函数来将对应的网络接口加入到组播组中。

```
1196 void ip_mc_inc_group(struct in_device *in_dev, __be32 addr)
1197 {
1198     struct ip_mc_list *im;
```

```
1199
1200    ASSERT_RTNL();
1201
1202    for (im=in_dev->mc_list; im; im=im->next) {
1203        if (im->multiaddr == addr) {
1204            im->users++;
1205            ip_mc_add_src(in_dev, &addr, MCAST_EXCLUDE, 0, NULL, 0);
1206            goto out;
1207        }
1208    }
```

遍历网络设备接口已加入的组播组列表，检测该网络设备接口是否已经加入了该组播组。如果是，则更新该组播组的引用计数以及源过滤模式和源列表。

```
1210    im = kmalloc(sizeof(*im), GFP_KERNEL);
1211    if (!im)
1212        goto out;
1213
1214    im->users=1;
1215    im->interface=in_dev;
1216    in_dev_hold(in_dev);
1217    im->multiaddr=addr;
1218    /* initial mode is (EX, empty) */
1219    im->sfmode = MCAST_EXCLUDE;
1220    im->sfcount[MCAST_INCLUDE] = 0;
1221    im->sfcount[MCAST_EXCLUDE] = 1;
1222    im->sources = NULL;
1223    im->tomb = NULL;
1224    im->crcount = 0;
1225    atomic_set(&im->refcnt, 1);
1226    spin_lock_init(&im->lock);
1227 #ifdef CONFIG_IP_MULTICAST
1228    im->tm_running=0;
1229    init_timer(&im->timer);
1230    im->timer.data=(unsigned long)im;
1231    im->timer.function=&igmp_timer_expire;
1232    im->unsolicit_count = IGMP_Unsolicited_Report_Count;
1233    im->reporter = 0;
1234    im->gsquery = 0;
1235 #endif
1236    im->loaded = 0;
1237    write_lock_bh(&in_dev->mc_list_lock);
1238    im->next=in_dev->mc_list;
1239    in_dev->mc_list=im;
1240    write_unlock_bh(&in_dev->mc_list_lock);
```

创建并设置组播组配置块后，将其添加到网络接口的 IP 配置块中。

```
1241 #ifdef CONFIG_IP_MULTICAST
1242    igmpv3_del_delrec(in_dev, im->multiaddr);
1243 #endif
1244    igmp_group_added(im);
1245    if (!in_dev->dead)
1246        ip_rt_multicast_event(in_dev);
1247 out:
```

```
1248     return;
1249 }
```

1242 由于网络设备接口加入了一个组，因此需要在 mc_tomb 上查找并删除组播地址的被阻止的源和组播组配置块。

1244 完成加入组播组，将组播组的硬件地址等信息添加到网络设备上并发送报告。

1245-1246 最后刷新路由缓存。

2. ip_mc_add_src()

ip_mc_add_src()用来更新组播源过滤列表和源过滤模式到指定网络设备接口和组播组的组记录中。在网络设备接口加入一个组播组时，或者通过套接口选项设置组播源过滤列表时被调用。参数说明如下：

- in_dev，网络设备 IP 配置块。
- pmca，网络设备接口加入的组播地址。
- sfmode，组播源过滤模式，有效值为 EXCLUDE 和 INCLUDE。
- sfcount，组播源列表中的源地址数。
- psfsrc，用于更新的组播源列表。
- delta，是否更新 ip_mc_list 结构中的 sfcount 的标志，为 0 时表示更新。

```
1626 static int ip_mc_add_src(struct in_device *in_dev, __be32 *pmca, int sfmode,
1627           int sfcount, __be32 *psfsrc, int delta)
1628 {
1629     struct ip_mc_list *pmc;
1630     int   isexclude;
1631     int   i, err;
1632
1633     if (!in_dev)
1634         return -ENODEV;
1635     read_lock(&in_dev->mc_list_lock);
1636     for (pmc=in_dev->mc_list; pmc; pmc=pmc->next) {
1637         if (*pmca == pmc->multiaddr)
1638             break;
1639     }
1640     if (!pmc) {
1641         /* MCA not found?? bug */
1642         read_unlock(&in_dev->mc_list_lock);
1643         return -ESRCH;
1644     }
1645     spin_lock_bh(&pmc->lock);
1646     read_unlock(&in_dev->mc_list_lock);
1647
1648 #ifdef CONFIG_IP_MULTICAST
1649     sf_markstate(pmc);
1650 #endif
1651     isexclude = pmc->sfmode == MCAST_EXCLUDE;
1652     if (!delta)
1653         pmc->sfcount[sfmode]++;
1654     err = 0;
1655     for (i=0; i<sfcount; i++) {
1656         err = ip_mc_add1_src(pmc, sfmode, &psfsrc[i], delta);
1657         if (err)
1658             break;
```

```
1659    }
1660    if (err) {
1661        int j;
1662
1663        pmc->sfcount[sfmode]--;
1664        for (j=0; j<i; j++)
1665            (void) ip_mc_del1_src(pmc, sfmode, &psfsrc[i]);
1666    } else if (isexclude != (pmc->sfcount[MCAST_EXCLUDE] != 0)) {
1667 #ifdef CONFIG_IP_MULTICAST
1668        struct in_device *in_dev = pmc->interface;
1669        struct ip_sf_list *psf;
1670 #endif
1671
1672        /* filter mode change */
1673        if (pmc->sfcount[MCAST_EXCLUDE])
1674            pmc->sfmode = MCAST_EXCLUDE;
1675        else if (pmc->sfcount[MCAST_INCLUDE])
1676            pmc->sfmode = MCAST_INCLUDE;
1677 #ifdef CONFIG_IP_MULTICAST
1678        /* else no filters; keep old mode for reports */
1679
1680        pmc->crcount = in_dev->mr_qrv ? in_dev->mr_qrv :
1681            IGMP_Unsolicited_Report_Count;
1682        in_dev->mr_ifc_count = pmc->crcount;
1683        for (psf=pmc->sources; psf; psf = psf->sf_next)
1684            psf->sf_crcount = 0;
1685        igmp_ifc_event(in_dev);
1686    } else if (sf_setstate(pmc)) {
1687        igmp_ifc_event(in_dev);
1688 #endif
1689    }
1690    spin_unlock_bh(&pmc->lock);
1691    return err;
1692 }
```

1633-1644 检测加入组播组的网络设备接口其 IP 配置块是否有效。并查找该设备已加入的组播组，通常都能查找命中，这里只是再次进行校验。

1649 标记对应组播源地址的过滤是否完全处于同一种过滤模式。

1651-1659 更新在该接口加入组播组的过滤模式为 EXCLUDE 和 INCLUDE 的套接口数，然后逐个将源地址更新到该组播组的源过滤列表中。

1660-1665 在逐个将源地址更新到该组播组源过滤列表的过程中，如果发生错误，则已完成的更新需回滚。

1666-1685 更新成功后，如果更新前后对应过滤模式的加入组播组套接口数不同，则需相应地更新该组播的源过滤模式；然后设置用于控制发送报告次数的 mr_ifc_count 和 crcount，同时重新设置每个源地址的 sf_crcount；最后，如果当前使用的是 IGMPv3 协议，则启动 mr_ifc_timer 组定时器。

1686-1689 更新成功，且更新前后的源过滤模式相同，则调用 sf_setstate()根据状态更新 tomb 链表，并启动 mr_ifc_timer 组定时器。

3. igmp_group_added()

当完成加入组播组时，调用 igmp_group_added()将组播组的硬件地址等信息添加到网络设

备上并发送报告。

```
1157 static void igmp_group_added(struct ip_mc_list *im)
1158 {
1159     struct in_device *in_dev = im->interface;
1160
1161     if (im->loaded == 0) {
1162         im->loaded = 1;
1163         ip_mc_filter_add(in_dev, im->multiaddr);
1164     }
1165
1166 #ifdef CONFIG_IP_MULTICAST
1167     if (im->multiaddr == IGMP_ALL_HOSTS)
1168         return;
1169
1170     if (in_dev->dead)
1171         return;
1172     if (IGMP_V1_SEEN(in_dev) || IGMP_V2_SEEN(in_dev)) {
1173         spin_lock_bh(&im->lock);
1174         igmp_start_timer(im, IGMP_Initial_Report_Delay);
1175         spin_unlock_bh(&im->lock);
1176         return;
1177     }
1178     /* else, v3 */
1179
1180     im->crcount = in_dev->mr_qrv ? in_dev->mr_qrv :
1181         IGMP_Unsolicited_Report_Count;
1182     igmp_ifc_event(in_dev);
1183 #endif
1184 }
```

1161-1164 将该组播组对应的硬件地址加载到网络设备上，并设置相应的标志，参见 16.13 节。

1167-1168 “224.0.0.1”组播地址是作为特殊情况处理的，从不需向该地址发送报告。

1170-1171 校验当前设备 IP 配置块是否有效。

1172-1182 新加入一个组，需主动发送报告。对于 V1 和 V2 的 IGMP 协议，启动接口定时器，而对于 V3 则启动组定时器。

16.10.3　网络设备离开组播组

1. ip_mc_leave_src()

ip_mc_leave_src()实现删除该设备 IP 配置块中相关的组播信息。

```
1767 static int ip_mc_leave_src(struct sock *sk, struct ip_mc_socklist *iml,
1768                 struct in_device *in_dev)
1769 {
1770     int err;
1771
1772     if (iml->sflist == 0) {
1773         /* any-source empty exclude case */
1774         return ip_mc_del_src(in_dev, &iml->multi.imr_multiaddr.s_addr,
1775             iml->sfmode, 0, NULL, 0);
1776     }
1777     err = ip_mc_del_src(in_dev, &iml->multi.imr_multiaddr.s_addr,
```

```
1778            iml->sfmode, iml->sflist->sl_count,
1779            iml->sflist->sl_addr, 0);
1780    sock_kfree_s(sk, iml->sflist, IP_SFLSIZE(iml->sflist->sl_max));
1781    iml->sflist = NULL;
1782    return err;
1783 }
```

如果套接口离开的组播组，其源地址过滤列表为空，则直接调用 ip_mc_del_src()删除 IP 配置块中相关的组播组信息，否则还需要释放源地址过滤列表。

2. ip_mc_del_src()

用于删除配置在网络设备上指定组播组的指定组播源过滤列表。参数说明如下：

- in_dev，所在网络设备的 IP 配置块。
- pmca，待删除组播源所在的组播组。
- sfmode，该组播组当前源过滤的模式。
- sfcount，删除组播源地址数。
- psfsrc，待删除的组播源地址。
- delta，标识是否更新 ip_mc_list 结构中的 sfcount，0 为更新。

```
1453 static int ip_mc_del_src(struct in_device *in_dev, __be32 *pmca, int sfmode,
1454            int sfcount, __be32 *psfsrc, int delta)
1455 {
1456    struct ip_mc_list *pmc;
1457    int    changerec = 0;
1458    int    i, err;
1459
1460    if (!in_dev)
1461       return -ENODEV;
1462    read_lock(&in_dev->mc_list_lock);
1463    for (pmc=in_dev->mc_list; pmc; pmc=pmc->next) {
1464       if (*pmca == pmc->multiaddr)
1465          break;
1466    }
1467    if (!pmc) {
1468       /* MCA not found?? bug */
1469       read_unlock(&in_dev->mc_list_lock);
1470       return -ESRCH;
1471    }
```

检测 IP 配置块是否有效，然后在该接口加入的组播中查找待离开的组播组。

```
1472    spin_lock_bh(&pmc->lock);
1473    read_unlock(&in_dev->mc_list_lock);
1474 #ifdef CONFIG_IP_MULTICAST
1475    sf_markstate(pmc);
1476 #endif
1477    if (!delta) {
1478       err = -EINVAL;
1479       if (!pmc->sfcount[sfmode])
1480          goto out_unlock;
1481       pmc->sfcount[sfmode]--;
1482    }
1483    err = 0;
```

```
1484    for (i=0; i<sfcount; i++) {
1485        int rv = ip_mc_del1_src(pmc, sfmode, &psfsrc[i]);
1486
1487        changerec |= rv > 0;
1488        if (!err && rv < 0)
1489            err = rv;
1490    }
1491    if (pmc->sfmode == MCAST_EXCLUDE &&
1492        pmc->sfcount[MCAST_EXCLUDE] == 0 &&
1493        pmc->sfcount[MCAST_INCLUDE]) {
1494 #ifdef CONFIG_IP_MULTICAST
1495        struct ip_sf_list *psf;
1496 #endif
1497
1498        /* filter mode change */
1499        pmc->sfmode = MCAST_INCLUDE;
1500 #ifdef CONFIG_IP_MULTICAST
1501        pmc->crcount = in_dev->mr_qrv ? in_dev->mr_qrv :
1502            IGMP_Unsolicited_Report_Count;
1503        in_dev->mr_ifc_count = pmc->crcount;
1504        for (psf=pmc->sources; psf; psf = psf->sf_next)
1505            psf->sf_crcount = 0;
1506        igmp_ifc_event(pmc->interface);
1507    } else if (sf_setstate(pmc) || changerec) {
1508        igmp_ifc_event(pmc->interface);
1509 #endif
1510    }
1511 out unlock:
1512    spin_unlock_bh(&pmc->lock);
1513    return err;
1514 }
```

1475 标识对应组播源地址的过滤是否完全处于同一种过滤模式。

1477-1482 更新该接口离开组播组后过滤模式为 EXCLUDE 和 INCLUDE 的套接口数。

1484-1490 根据源地址逐个更新该组播组的源过滤列表。

1491-1506 更新完成后，如果原先组播源过滤模式为 MCAST_EXCLUDE，且加入组播组 MCAST_EXCLUDE 过滤模式的套接口数为 0，则更新过滤模式为 MCAST_INCLUDE。过滤模式发生变化，需重新设定 crcount 和 mr_ifc_count，然后更新 sf_crcount，最后启动 mr_ifc_timer 组定时器。

1507-1510 如果更新前后的源过滤模式相同，则调用 sf_setstate()根据状态更新 tomb 链表，一旦更新了 tomb 链表或组播源列表发生了变化，即需启动 mr_ifc_timer 组定时器。

3. ip_mc_del1_src()

ip_mc_del1_src()用来从网络设备删除单个用于过滤的组播源地址。

```
1404 static int ip_mc_del1_src(struct ip_mc_list *pmc, int sfmode,
1405     __be32 *psfsrc)
1406 {
1407     struct ip_sf_list *psf, *psf_prev;
1408     int rv = 0;
1409
1410     psf_prev = NULL;
1411     for (psf=pmc->sources; psf; psf=psf->sf_next) {
```

```
1412        if (psf->sf_inaddr == *psfsrc)
1413            break;
1414        psf_prev = psf;
1415    }
1416    if (!psf || psf->sf_count[sfmode] == 0) {
1417        /* source filter not found, or count wrong =>  bug */
1418        return -ESRCH;
1419    }
```

在指定的组播组中查找指定的用于过滤的组播源。如果查找失败或查找命中的源地址不存在对应的模式，则返回错误码 ESRCH。

```
1420    psf->sf_count[sfmode]--;
1421    if (psf->sf_count[sfmode] == 0) {
1422        ip_rt_multicast_event(pmc->interface);
1423    }
```

更新指定过滤模式组播源地址的套接口在该接口加入的组播组数。如果数量为 0，则需刷新路由缓存。

```
1424    if (!psf->sf_count[MCAST_INCLUDE] && !psf->sf_count[MCAST_EXCLUDE]) {
1425 #ifdef CONFIG_IP_MULTICAST
1426        struct in_device *in_dev = pmc->interface;
1427 #endif
1428
1429        /* no more filters for this source */
1430        if (psf_prev)
1431            psf_prev->sf_next = psf->sf_next;
1432        else
1433            pmc->sources = psf->sf_next;
1434 #ifdef CONFIG_IP_MULTICAST
1435        if (psf->sf_oldin &&
1436            !IGMP_V1_SEEN(in_dev) && !IGMP_V2_SEEN(in_dev)) {
1437            psf->sf_crcount = in_dev->mr_qrv ? in_dev->mr_qrv :
1438                IGMP_Unsolicited_Report_Count;
1439            psf->sf_next = pmc->tomb;
1440            pmc->tomb = psf;
1441            rv = 1;
1442        } else
1443 #endif
1444            kfree(psf);
1445    }
1446    return rv;
1447 }
```

如果不同过滤模式的套接口在该接口加入组播组的数量都为 0，则表示已经没有套接口使用该源地址作为组播源，因此需删除该组播源的描述块。如果当前使用的是 IGMPv3 协议，则将该组播源描述块移动到该组播组描述块的 tomb 中，否则直接将其释放。

4. ip_mc_dec_group()

当主机所有的套接口都离开指定组播组之后，需将组播组的硬件地址等信息从网络设备上删除并发送离开组消息，这个功能由 ip_mc_dec_group()实现。

遍历 IP 配置块中的组播组，查找离开的组播组，如果离开的组播组已经没有套接口，则将

组播组的硬件地址等信息从网络设备上删除，并将该组播组信息块从 IP 配置块中删除。参数说明如下：

- in_dev，网络设备接口的 IP 控制块。
- addr，待离开的组播组。

```
1255 void ip_mc_dec_group(struct in_device *in_dev, __be32 addr)
1256 {
1257     struct ip_mc_list *i, **ip;
1258
1259     ASSERT_RTNL();
1260
1261     for (ip=&in_dev->mc_list; (i=*ip)!=NULL; ip=&i->next) {
1262         if (i->multiaddr==addr) {
1263             if (--i->users == 0) {
1264                 write_lock_bh(&in_dev->mc_list_lock);
1265                 *ip = i->next;
1266                 write_unlock_bh(&in_dev->mc_list_lock);
1267                 igmp_group_dropped(i);
1268
1269                 if (!in_dev->dead)
1270                     ip_rt_multicast_event(in_dev);
1271
1272                 ip_ma_put(i);
1273                 return;
1274             }
1275             break;
1276         }
1277     }
1278 }
```

5. igmp_group_dropped()

igmp_group_dropped()在组播组的硬件地址等信息已经配置到网络设备上的情况下，会将其删除。对于特殊的组播组 IGMP_ALL_HOSTS，无需发送报告。

```
1120 static void igmp_group_dropped(struct ip_mc_list *im)
1121 {
1122     struct in_device *in_dev = im->interface;
1123 #ifdef CONFIG_IP_MULTICAST
1124     int reporter;
1125 #endif
1126
1127     if (im->loaded) {
1128         im->loaded = 0;
1129         ip_mc_filter_del(in_dev, im->multiaddr);
1130     }
1131
1132 #ifdef CONFIG_IP_MULTICAST
1133     if (im->multiaddr == IGMP_ALL_HOSTS)
1134         return;
1135
1136     reporter = im->reporter;
1137     igmp_stop_timer(im);
1138
1139     if (!in_dev->dead) {
```

```
1140        if (IGMP_V1_SEEN(in_dev))
1141            goto done;
1142        if (IGMP_V2_SEEN(in_dev)) {
1143            if (reporter)
1144                igmp_send_report(in_dev, im, IGMP_HOST_LEAVE_MESSAGE);
1145            goto done;
1146        }
1147        /* IGMPv3 */
1148        igmpv3_add_delrec(in_dev, im);
1149
1150        igmp_ifc_event(in_dev);
1151    }
1152 done:
1153 #endif
1154     ip_mc_clear_src(im);
1155 }
```

1127-1130 如果所在接口加入的组播组的硬件地址已经加载到网络设备上，则在网络设备上删除组播组对应的组播硬件地址。

1133-1134 对特殊组播组 IGMP_ALL_HOSTS，无需发送报告。

1136-1137 停止已启动的组查询定时器。

1139-1151 如果当前的 IP 配置块有效，则在支持 IGMPv2 时发送 IGMP_HOST_LEAVE_MESSAGE 报告，在支持 IGMPv3 时根据离开组播组配置信息创建与之一致的信息块添加到 mc_tomb 链表上。当通过 IGMPv3 协议离开组播组后，则通过 igmp_ifc_event()来启动 mr_ifc_timer 定时器，主动报告相关信息。

1154 接口已经离开该组播组，因此需删除并释放该组播组所有最近被阻止的组播源以及源地址过滤列表。

16.11 ip_mc_source()

ip_mc_source()用于阻塞或开通组播源。IP_BLOCK_SOURCE 和 IP_UNBLOCK_SOURCE 选项，以及组播源加入或离开组播组的 IP_ADD_SOURCE_MEMBERSHIP 和 IP_DROP_SOURCE_MEMBERSHIP 选项，在加入或离开了指定的组播组后，还会调用该函数阻塞或开通组播源。

- add，标识进行何种选项操作，IP_BLOCK_SOURCE 和 IP_ADD_SOURCE_MEMBERSHIP 选项则为 1，IP_UNBLOCK_SOURCE 和 IP_DROP_SOURCE_MEMBERSHIP 选项则为 0。
- omode，进行组播源过滤的模式，为 MCAST_EXCLUDE 或 MCAST_INCLUDE。
- sk，操作组播组的套接口。
- mreqs，指向如下结构：

```
struct ip_mreq_source {
    __be32        imr_multiaddr;    /*阻塞和开通组播源的组播组，或者加入和离开的组播组*/
    __be32        imr_interface;    /*本机源地址*/
    __be32        imr_sourceaddr;   /*组播源过滤地址*/
};
```

- ifindex，与该组播组关联的网络设备索引。当 ifindex 为 0 时，则根据本地地址获取网络设备。

```
1827 int ip_mc_source(int add, int omode, struct sock *sk, struct
1828     ip_mreq_source *mreqs, int ifindex)
1829 {
1830     int err;
1831     struct ip_mreqn imr;
1832     __be32 addr = mreqs->imr_multiaddr;
1833     struct ip_mc_socklist *pmc;
1834     struct in_device *in_dev = NULL;
1835     struct inet_sock *inet = inet_sk(sk);
1836     struct ip_sf_socklist *psl;
1837     int leavegroup = 0;
1838     int i, j, rv;
1839
1840     if (!MULTICAST(addr))
1841         return -EINVAL;
1842
1843     rtnl_lock();
1844
1845     imr.imr_multiaddr.s_addr = mreqs->imr_multiaddr;
1846     imr.imr_address.s_addr = mreqs->imr_interface;
1847     imr.imr_ifindex = ifindex;
1848     in_dev = ip_mc_find_dev(&imr);
1849
1850     if (!in_dev) {
1851         err = -ENODEV;
1852         goto done;
1853     }
1854     err = -EADDRNOTAVAIL;
```

1840-1841 检测操作的组播地址是否有效。

1845-1853 获取操作组播组的网络设备的 IP 控制块，并检测其有效性。

```
1856     for (pmc=inet->mc_list; pmc; pmc=pmc->next) {
1857         if (pmc->multi.imr_multiaddr.s_addr == imr.imr_multiaddr.s_addr
1858             && pmc->multi.imr_ifindex == imr.imr_ifindex)
1859             break;
1860     }
1861     if (!pmc) {             /* must have a prior join */
1862         err = -EINVAL;
1863         goto done;
1864     }
1865     /* if a source filter was set, must be the same mode as before */
1866     if (pmc->sflist) {
1867         if (pmc->sfmode != omode) {
1868             err = -EINVAL;
1869             goto done;
1870         }
1871     } else if (pmc->sfmode != omode) {
1872         /* allow mode switches for empty-set filters */
1873         ip_mc_add_src(in_dev, &mreqs->imr_multiaddr, omode, 0, NULL, 0);
1874         ip_mc_del_src(in_dev, &mreqs->imr_multiaddr, pmc->sfmode, 0,
1875             NULL, 0);
1876         pmc->sfmode = omode;
1877     }
```

1856-1864 获取套接口控制块中对应的组播信息块，确保套接口已加入待离开的组播组。

1866-1871 如果之前已设置了源过滤，则必须与原先的过滤模式相同，否则无效。

1871-1877 反之，如果之前没有设置源过滤，且过滤模式与现有的不同，则重新设置过滤模式。

```
1879     psl = pmc->sflist;
1880     if (!add) {
1881        if (!psl)
1882           goto done;    /* err = -EADDRNOTAVAIL */
1883        rv = !0;
1884        for (i=0; i<psl->sl_count; i++) {
1885           rv = memcmp(&psl->sl_addr[i], &mreqs->imr_sourceaddr,
1886              sizeof(__be32));
1887           if (rv == 0)
1888              break;
1889        }
1890        if (rv)       /* source not found */
1891           goto done;    /* err = -EADDRNOTAVAIL */
1892
1893        /* special case - (INCLUDE, empty) == LEAVE_GROUP */
1894        if (psl->sl_count == 1 && omode == MCAST_INCLUDE) {
1895           leavegroup = 1;
1896           goto done;
1897        }
1898
1899        /* update the interface filter */
1900        ip_mc_del_src(in_dev, &mreqs->imr_multiaddr, omode, 1,
1901           &mreqs->imr_sourceaddr, 1);
1902
1903        for (j=i+1; j<psl->sl_count; j++)
1904           psl->sl_addr[j-1] = psl->sl_addr[j];
1905        psl->sl_count--;
1906        err = 0;
1907        goto done;
1908     }
```

1880 add 为 0，表示需阻塞某个源。

1881-1882 如果该组播组中没有源过滤列表，则无需作处理。

1883-1891 在组播组的源过滤列表中，查找需阻塞的源地址，如果找不到指定的源地址，则同样无需作处理。

1894-1896 如果当前源过滤列表中只有一个源地址，且当前过滤模式为 INCLUDE，此时不接收任何组播地址，因此作为离开组处理。

1900-1907 根据网络设备和套接口上的源地址过滤列表。

```
1911     if (psl && psl->sl_count >= sysctl_igmp_max_msf) {
1912        err = -ENOBUFS;
1913        goto done;
1914     }
1915     if (!psl || psl->sl_count == psl->sl_max) {
1916        struct ip_sf_socklist *newpsl;
1917        int count = IP_SFBLOCK;
```

```
1918
1919        if (psl)
1920            count += psl->sl_max;
1921        newpsl = sock_kmalloc(sk, IP_SFLSIZE(count), GFP_KERNEL);
1922        if (!newpsl) {
1923            err = -ENOBUFS;
1924            goto done;
1925        }
1926        newpsl->sl_max = count;
1927        newpsl->sl_count = count - IP_SFBLOCK;
1928        if (psl) {
1929            for (i=0; i<psl->sl_count; i++)
1930                newpsl->sl_addr[i] = psl->sl_addr[i];
1931            sock_kfree_s(sk, psl, IP_SFLSIZE(psl->sl_max));
1932        }
1933        pmc->sflist = psl = newpsl;
1934    }
1935    rv = 1;    /* > 0 for insert logic below if sl_count is 0 */
1936    for (i=0; i<psl->sl_count; i++) {
1937        rv = memcmp(&psl->sl_addr[i], &mreqs->imr_sourceaddr,
1938            sizeof(__be32));
1939        if (rv == 0)
1940            break;
1941    }
1942    if (rv == 0)        /* address already there is an error */
1943        goto done;
1944    for (j=psl->sl_count-1; j>=i; j--)
1945        psl->sl_addr[j+1] = psl->sl_addr[j];
1946    psl->sl_addr[i] = mreqs->imr_sourceaddr;
1947    psl->sl_count++;
1948    err = 0;
1949    /* update the interface list */
1950    ip_mc_add_src(in_dev, &mreqs->imr_multiaddr, omode, 1,
1951            &mreqs->imr_sourceaddr, 1);
```

1911-1914 以下处理的是开通某个组播源的情况，首选校验组播源过滤列表是否达到上限。

1915-1934 如果该组播组中没有源过滤列表，则需分配空间来存储组播源地址。否则又如果用于存储组播源地址的空间已用完，则需重新分配更大空间来存储组播源地址，在将原有的组播源地址复制到新空间后，还需释放原有的空间。最后将新分配的组播源地址空间设置到该组播组控制块上。

1935-1951 在组播源地址列表中查找是否已经存在相同的源地址，只有在没有的情况下才将该组播源地址添加到源过滤列表中，同时将源地址更新到网络设备的组播配置中。

```
1952 done:
1953    rtnl_unlock();
1954    if (leavegroup)
1955        return ip_mc_leave_group(sk, &imr);
1956    return err;
1957 }
```

如果作为离开组播组处理，则调用 ip_mc_leave_group()进行相应的操作。

16.12 ip_mc_msfilter()

ip_mc_msfilter()用来设置组播源过滤列表，在通过套接口选项 IP_MSFILTER 和 MCAST_MSFILTER 进行设置时，读取设置参数并经过必要的校验后就会调用该函数进行组播源过滤列表的设置。

- sk，设置组播源过滤列表的套接口。
- msf，指向如下结构：

```
struct ip_msfilter {
    __be32        imsf_multiaddr;    /*设置组播源过滤的组播地址。*/
    __be32        imsf_interface;    /*设置组播源过滤的源地址。*/
    __u32         imsf_fmode;        /*设置组播源过滤的模式。*/
    __u32         imsf_numsrc;       /*设置组播源过滤的源地址数。*/
    __be32        imsf_slist[1];     /*设置组播源过滤的源地址列表。*/
};
```

- ifindex，设置组播源过滤的网络设备索引。

```
1959 int ip_mc_msfilter(struct sock *sk, struct ip_msfilter *msf, int ifindex)
1960 {
1961     int err = 0;
1962     struct ip_mreqn    imr;
1963     __be32 addr = msf->imsf_multiaddr;
1964     struct ip_mc_socklist *pmc;
1965     struct in_device *in_dev;
1966     struct inet_sock *inet = inet_sk(sk);
1967     struct ip_sf_socklist *newpsl, *psl;
1968     int leavegroup = 0;
1969
1970     if (!MULTICAST(addr))
1971         return -EINVAL;
1972     if (msf->imsf_fmode != MCAST_INCLUDE &&
1973         msf->imsf_fmode != MCAST_EXCLUDE)
1974         return -EINVAL;
1975
1976     rtnl_lock();
1977
1978     imr.imr_multiaddr.s_addr = msf->imsf_multiaddr;
1979     imr.imr_address.s_addr = msf->imsf_interface;
1980     imr.imr_ifindex = ifindex;
1981     in_dev = ip_mc_find_dev(&imr);
1982
1983     if (!in_dev) {
1984         err = -ENODEV;
1985         goto done;
1986     }
```

1970-1974 检测进行设置组播源过滤列表的组播地址和过滤模式。

1978-1986 获取进行设置的网络设备的 IP 控制块，并检测该 IP 控制块的有效性。

```
1988     /* special case - (INCLUDE, empty) == LEAVE_GROUP */
```

```
1989     if (msf->imsf_fmode == MCAST_INCLUDE && msf->imsf_numsrc == 0) {
1990        leavegroup = 1;
1991        goto done;
1992     }
```

如果源地址过滤模式为 INCLUDE 并且源地址列表为空，则表示离开一个组，因此作为离开组处理。

```
1994     for (pmc=inet->mc_list; pmc; pmc=pmc->next) {
1995        if (pmc->multi.imr_multiaddr.s_addr == msf->imsf_multiaddr &&
1996           pmc->multi.imr_ifindex == imr.imr_ifindex)
1997           break;
1998     }
1999     if (!pmc) {          /* must have a prior join */
2000        err = -EINVAL;
2001        goto done;
2002     }
```

在套接口控制块中查找该套接口加入的组播组中是否存在该组播组。套接口必须已加入该组播组，否则操作无效。

```
2003     if (msf->imsf_numsrc) {
2004        newpsl = sock_kmalloc(sk, IP_SFLSIZE(msf->imsf_numsrc),
2005                              GFP_KERNEL);
2006        if (!newpsl) {
2007           err = -ENOBUFS;
2008           goto done;
2009        }
2010        newpsl->sl_max = newpsl->sl_count = msf->imsf_numsrc;
2011        memcpy(newpsl->sl_addr, msf->imsf_slist,
2012           msf->imsf_numsrc * sizeof(msf->imsf_slist[0]));
2013        err = ip_mc_add_src(in_dev, &msf->imsf_multiaddr,
2014           msf->imsf_fmode, newpsl->sl_count, newpsl->sl_addr, 0);
2015        if (err) {
2016           sock_kfree_s(sk, newpsl, IP_SFLSIZE(newpsl->sl_max));
2017           goto done;
2018        }
2019     } else {
2020        newpsl = NULL;
2021        (void) ip_mc_add_src(in_dev, &msf->imsf_multiaddr,
2022                  msf->imsf_fmode, 0, NULL, 0);
2023     }
```

如果存在一个或多个组播源过滤的源地址，则分配空间将这些组播源过滤的源地址设置到网络设备的 IP 配置块中；否则根据过滤模式调整源过滤设置。

```
2024     psl = pmc->sflist;
2025     if (psl) {
2026        (void) ip_mc_del_src(in_dev, &msf->imsf_multiaddr, pmc->sfmode,
2027           psl->sl_count, psl->sl_addr, 0);
2028        sock_kfree_s(sk, psl, IP_SFLSIZE(psl->sl_max));
2029     } else
2030        (void) ip_mc_del_src(in_dev, &msf->imsf_multiaddr, pmc->sfmode,
2031           0, NULL, 0);
```

```
2032     pmc->sflist = newpsl;
2033     pmc->sfmode = msf->imsf_fmode;
2034     err = 0;
```

根据套接口中原先的源地址过滤列表，在网络设备的 IP 配置块删除相应的源过滤列表。完成后，将新的源地址过滤列表和过滤模式设置到套接口中。

```
2035 done:
2036     rtnl_unlock();
2037     if (leavegroup)
2038         err = ip_mc_leave_group(sk, &imr);
2039     return err;
2040 }
```

如果作为离开组播组处理，则调用 ip_mc_leave_group()进行相应的操作。

16.13 网络设备组播硬件地址的管理

一个套接口加入到一个组播组后，不仅仅套接口和网络设备加入了组播组，还需要在对应网络设备上添加组播组对应的组播硬件地址，这个过程由 ip_mc_filter_add()实现。该函数通常在完成套接口和网络设备配置之后被调用。其实现过程是，首先通过 arp_mc_map()根据组播地址得到对应的组播硬件地址，然后通过 dev_mc_add()将组播硬件地址设置到网络设备 net_device 结构的 mc_list 成员中。

同样，一个套接口离开到一个组播组后，需要做与加入组播组相反的操作，在网络设备上删除组播组对应的组播硬件地址，这由 ip_mc_filter_del()实现，实现过程与 ip_mc_filter_add()正相反。该函数也通常在完成套接口和网络设备的配置之后被调用。

第 17 章　邻居子系统

17.1　什么是邻居子系统

所谓邻居，是指在同一个 IP 局域网内的主机，或者邻居之间在三层上仅相隔一跳的距离。而邻居子系统，则提供了三层协议地址与二层协议地址之间的映射关系，此外还提供二层首部缓存，以加速发送数据包。

在发送数据的时候，先进行路由查找，如果找到到目的地地址的路径，再查看邻居表中是否存在相应的映射关系，如果没有则新建邻居项；然后判断邻居项是否为可用状态，如不可用则把数据报存至发送缓存队列后发送请求；在接收到请求应答后，将对应邻居项置为可用，并将其缓存队列中的数据包发送出去；如果在指定时间内未收到响应包，则将对应邻居项置为无效状态。

邻居子系统的实现涉及以下文件：

- include/linux/inetdevice.h，定义 IPv4 专用的网络设备相关的结构、宏等。
- include/net/neighbour.h，定义邻居项等结构、宏和函数原型。
- net/core/neighbour.c，邻居子系统的实现。

17.2　系统参数

邻居子系统的参数有：

- mcast_solicit，多播或广播在标识一个邻居项不可达之前最多尝试解析的次数。默认值为 3 次。
- ucast_solicit，在请求 ARP 守护进程前尝试发送单播探测的次数。默认值为 3 次。
- app_solicit，在退回到多播探测之前，通过 netlink 向用户空间的 ARP 守护进程最多可发送探测的次数。默认值为 0 次。
- retrans_time，重传一个请求前延迟的 jiffies 值。默认值为 1s。
- base_reachable_time，邻居项有效期初始值。一旦发现了一个邻居项，则该邻居项至少在一段时间内有效，时间长度为在 base_reachable_time/2 和 3*base_reachable_time/2 之间的一个随机值。一个邻居项如果能接收到来自更高层的协议的正反馈，则其有效性会被延长。默认值为 30s。
- delay_first_probe_time，在一个邻居项过期后发送第一个探测之前的延迟时间，默认值为 5s。
- gc_stale_time，确定多久检测一次邻居项过期。如果一个邻居项被确定为过期，则在向它发送数据之前会再次确认。默认值为 60s。
- unres_qlen，最多允许其他网络层在每个未解析地址上排队的数据报数。默认值为 3。
- proxy_qlen，允许在 proxy-ARP 地址上排队的数据报数。默认值为 64。

- anycast_delay，在响应一个 IPv6 邻居请求消息之前最多延迟的 jiffies 值，对 anycast 的支持还没实现。默认值为 1s。
- proxy_delay，当接收到一个对未知代理 ARP 地址的 ARP 请求时，将延迟 proxy_delay jiffies 响应。这用来在一些情况下防止 ARP 报文的洪泛。默认值为 0.8s。
- locktime，一个 ARP 项至少在缓存中保存的 jiffies 值。在存在多于一个的可能映射的情况下，这通常是因为网络配置错误，用来防止 ARP 缓存抖动。默认值为 1s。
- gc_interval，垃圾回收处理邻居项的时间间隔，默认值为 30s。
- gc_thresh1，缓存中最少要保持多少条邻居项，如果缓存中的邻居项数少于该值，则不会执行垃圾回收。默认值为 128。
- gc_thresh2，缓存中能保持邻居项数目的软上限（垃圾回收机制允许在回收前保持缓存中的邻居项数超过该值达 5s），默认值为 512。
- gc_thresh3，缓存中最多能保持多少条邻居项的硬上限，一旦缓存中实际的邻居项数超过该值即执行垃圾回收。默认值为 1024。
- retrans_time_ms，同 retrans_time，但单位为毫秒。
- base_reachable_time_ms，同 base_reachable_time，但单位为毫秒。

17.3 邻居子系统的结构

邻居子系统的结构由多个数据结构构成，图 17-1 为邻居子系统的结构图。

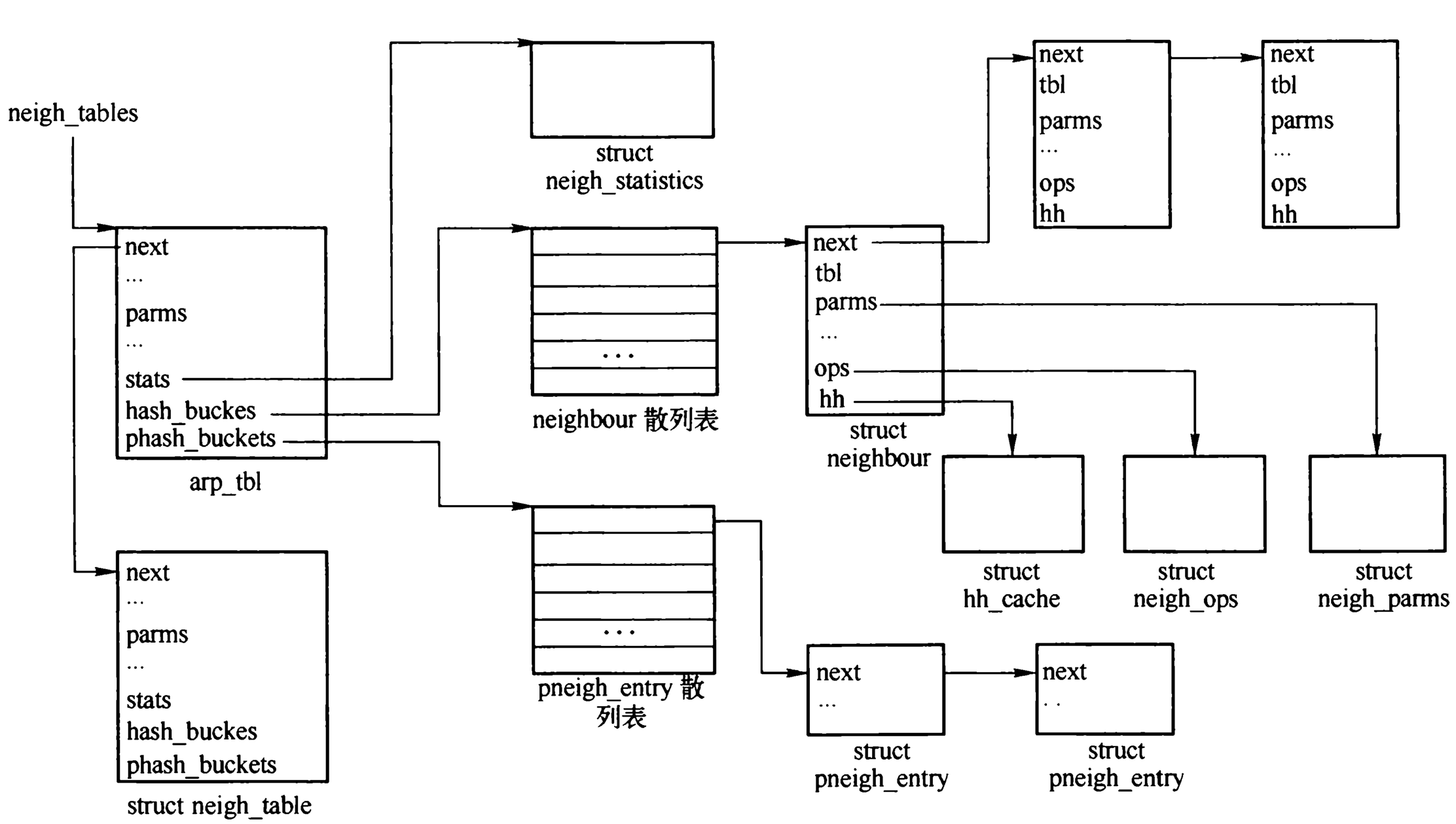

图 17-1 邻居子系统结构

17.3.1 neigh_table 结构

neigh_table 结构用来存储与邻居协议相关的参数、功能函数，以及邻居项散列表，一个

neigh_table 结构实例对应一个邻居协议，所有的实例都链接在全局链表 neigh_tables 中。对于 ARP 协议，其 neigh_table 结构实例是 arp_tbl。

```
138 struct neigh_table
139 {
140     struct neigh_table   *next;
141     int          family;
142     int          entry_size;
143     int          key_len;
144    _u32          (*hash)(const void *pkey, const struct net_device *);
145     int          (*constructor)(struct neighbour *);
146     int          (*pconstructor)(struct pneigh_entry *);
147     void         (*pdestructor)(struct pneigh_entry *);
148     void         (*proxy_redo)(struct sk_buff *skb);
149     char         *id;
150     struct neigh_parms   parms;
151     /* HACK. gc_* shoul follow parms without a gap! */
152     int          gc_interval;
153     int          gc_thresh1;
154     int          gc_thresh2;
155     int          gc_thresh3;
156     unsigned long       last_flush;
157     struct timer_list    gc_timer;
158     struct timer_list    proxy_timer;
159     struct sk_buff_head   proxy_queue;
160     atomic_t       entries;
161     rwlock_t       lock;
162     unsigned long       last_rand;
163     struct kmem_cache       *kmem_cachep;
164     struct neigh_statistics   *stats;
165     struct neighbour   **hash_buckets;
166     unsigned int       hash_mask;
167    __u32          hash_rnd;
168     unsigned int       hash_chain_gc;
169     struct pneigh_entry   **phash_buckets;
170 #ifdef CONFIG_PROC_FS
171     struct proc_dir_entry   *pde;
172 #endif
173 };
```

140 struct neigh_table *next

用来连接到 neigh_tables 中，该链表中除了 ARP 的 arp_tbl，还有 IPV6T 和 DECNE 所使用邻居协议的邻居表实例 nd_tbl 和 dn_neigh_table 等。

141 int family

邻居协议所属的地址族，ARP 为 AF_INET。

142 int entry_size

邻居项结构的大小。对 arp_tbl 来说，初始化为 sizeof(neighbour)+4，这是因为在 ARP 中 neighbour 结构的最后一个成员零长数组 primary_key，实际指向一个 IPv4 地址，因此其中的 4 是一个 IPv4 地址的长度。

143 int key_len

哈希函数所使用的 key 的长度。实际上哈希函数使用的 key 是三层协议地址，因此在

IPv4 中的 key 就是 IP 地址，因此这个值就是 4。

```
144 __u32 (*hash)(const void *pkey, const struct net_device *)
```

哈希函数，用来计算哈希值，ARP 中为 arp_hash()。

```
145 int (*constructor)(struct neighbour *)
```

邻居表项初始化函数，用于初始化一个新的 neighbour 结构实例中与协议相关的字段。在 ARP 中，该函数为 arp_constructor()，由邻居表项创建函数 neigh_create()中调用。

```
146 int (*pconstructor)(struct pneigh_entry *)
147 void (*pdestructor)(struct pneigh_entry *)
```

这两个函数分别在创建和释放一个代理项时被调用。IPv4 并没有使用，只在 IPv6 中被使用。

```
148 void (*proxy_redo)(struct sk_buff *skb)
```

用来处理在 neigh_table->proxy_queue 缓存队列中的代理 ARP 报文，参见 18.12.3 节。

```
149 char *id;
```

用来分配 neighbour 结构实例的缓冲池名字符串，arp_tlb 的该字段为“arp_cache”。

```
150 struct neigh_parms parms
```

存储一些与协议相关的可调节参数。如重传超时时间、proxy_queue 队列长度等，参见 17.3.4 节。

```
152 int gc_interval
```

垃圾回收时钟 gc_timer 的到期间隔时间，每当该时钟到期即触发一次垃圾回收，该字段初始值为 30s，参见 17.11 节。

```
153 int gc_thresh1
154 int gc_thresh2
155 int gc_thresh3
```

此三个阈值对应于内存对邻居项作垃圾回收处理的不同级别：

- 如果缓存中的邻居项数少于 gc_thresh1，则不会执行垃圾回收。
- 如果邻居项数目超过 gc_thresh2，则在新建邻居项时若超过五秒未刷新，必须立即刷新并强制垃圾回收。
- 如果邻居项数目超过 gc_thresh3，则在新建邻居项时，必须立即刷新并强制垃圾回收。

```
156 unsigned long last_flush
```

记录最近一次调用 neigh_forced_gc()强制刷新邻居表的时间，用来作为是否进行垃圾回收的判断条件。

```
157 struct timer_list gc_timer
```

垃圾回收定时器，参见 17.11 节。

```
158 struct timer_list proxy_timer
```

处理 proxy_queue 队列的定时器，当 proxy_queue 队列为空时，第一个 ARP 报文加入到队列就会启动该定时器。该定时器在 neigh_table_init()中初始化，处理例程为 neigh_proxy_process()，参见 17.14 节。

```
159 struct sk_buff_head proxy_queue
```

对于接收到的需要进行代理的 ARP 报文，会先将其缓存到 proxy_queue 队列中，在定时器处理函数中再对其进行处理。

160 atomic_t entries

整个表中邻居项的数目，在用 neigh_alloc()创建和用 neigh_destroy()释放邻居项时计数，参见 17.11 节。

161 rwlock_t lock

用于控制邻居表的读写锁。例如 neigh_lookup()只需要读邻居表，而 neigh_periodic_timer()则需要读写邻居表。

162 unsigned long last_rand

用于记录 neigh_parms 结构中 reachable_time 成员最近一次被更新的时间。

163 struct kmem_cache *kmem_cachep

用来分配 neighbour 结构实例的 slab 缓存，在 neigh_table_init()中初始化。

164 struct neigh_statistics *stats

有关邻居表中邻居项的各类统计数据，参见 17.3.6 节。

165 struct neighbour **hash_buckets

用于存储邻居项的散列表，该散列表在分配邻居项时，如果邻居项数超出散列表容量，可动态扩容，参见 17.8 节。

166 unsigned int hash_mask

邻居项散列表桶数减 1，以方便用来计算关键字。

167 __u32 hash_rnd

随机数，用来在 hash_buckets 散列表扩容时计算关键字，以免受到 ARP 攻击，参见 17.8 节。

168 unsigned int hash_chain_gc

保存下一次将进行垃圾回收处理的桶序号，如果超过最大值 hash_mash 则从散列表的第一桶开始，参见 17.11 节。

169 struct pneigh_entry **phash_buckets

存储 ARP 代理三层协议地址的散列表，在 neigh_table_init_no_netlink()中完成初始化。

171 struct proc_dir_entry *pde

如果支持 proc 文件系统，则用来在/proc/net/stat/下注册 arp_cache 文件，在 neigh_table_init_no_netlink()中完成注册。

17.3.2　neighbour 结构

邻居项使用 neighbour 结构来描述，该结构存储了邻居的相关信息，包括状态、二层和三层协议地址、提供给三层协议的函数指针，还有定时器和缓存的二层首部等。需要注意的是，一个邻居并不代表一个主机，而是一个三层协议地址，对于配置了多接口的主机，一个主机将对应多个三层协议地址。

```
89  struct neighbour
90  {
91      struct neighbour    *next;
92      struct neigh_table  *tbl;
93      struct neigh_parms  *parms;
94      struct net_device       *dev;
95      unsigned long       used;
96      unsigned long       confirmed;
97      unsigned long       updated;
```

```
98    _u8           flags;
99    _u8           nud_state;
100   _u8           type;
101   _u8           dead;
102    atomic_t       probes;
103    rwlock_t       lock;
104    unsigned char       ha[ALIGN(MAX_ADDR_LEN, sizeof(unsigned long))];
105    struct hh_cache       *hh;
106    atomic_t       refcnt;
107    int          (*output)(struct sk_buff *skb);
108    struct sk_buff_head   arp_queue;
109    struct timer_list   timer;
110    struct neigh_ops    *ops;
111    u8           primary_key[0];
112 };
```

91 struct neighbour *next

通过 next 把邻居项插入到散列表桶链表上，总在桶的前部插入新的邻居项。

92 struct neigh_table *tbl

指向相关协议的 neigh_table 结构实例，即该邻居项所在的邻居表。如果该邻居项对应的是一个 IPv4 地址，则该字段指向 arp_tbl。

93 struct neigh_parms *parms

用于调节邻居协议的参数。在创建邻居项函数 neigh_create()中，首先调用 neigh_alloc()分配一个邻居项，在该函数中使用邻居表的 parms 对邻居项的该字段进行初始化，接着 neigh_create()调用邻居表的 constructor()，对于 arp_tbl 是 arp_constructor()，对邻居项作特定的设置时将该字段修改为协议相关设备的参数。

94 struct net_device *dev

通过此网络设备可访问到该邻居。对每个邻居来说，只能有一个可用来访问该邻居的网络设备。

95 unsigned long used

最近一次被使用时间。该字段值并不总是与数据传输同步更新，当邻居不处于 NUD_CONNECTED 状态时，该值在 neigh_event_send()更新中；而当邻居处于 NUD_CONNECTED 状态时，该值有时会通过 gc_timer 定时器处理函数更新。

96 unsigned long confirmed

记录最近一次确认该邻居可达性的时间，用于描述邻居的可达性，通常是接收到来自该邻居的报文后更新。传输层通过 neigh_confirm()来更新，邻居子系统则通过 neigh_update()更新。

97 unsigned long updated

记录最近一次被 neigh_update()更新的时间。不要将 updated 与 confirmed 混淆，它们各自针对不同的特性。该字段值在邻居状态发生变化时更新。

98 __u8 flags

记录邻居项的一些标志与特性，见表 17-1。

表 17-1 flags 取值

flags	描述
NTF_ROUTER	此标志只使用于 IPv6，标识该邻居为一个路由器

99 __u8 nud_state

标识邻居项的状态，参见 17.5 节。

100 __u8 type

邻居项地址的类型。对于 ARP，是在创建邻居项时 arp_constructor()中设置的。该类型与路由表项类型意义相同，最经常使用的类型如 RTN_UNICAST，RTN_LOCAL，RTN_BROADCAST，RTN_ANYCAST 和 RTN_MULTICAST，见表 19-3。

101 __u8 dead

生存标志。如果设置为 1，则意味着该邻居项正在被删除，最终通过垃圾回收将其删除。

102 atomic_t probes

尝试发送请求报文而未能得到应答的次数，该值在定时器处理函数中被检测，当该值达到指定的上限时，该邻居项便进入 NUD_FAILED 状态。

103 rwlock_t lock

用来控制访问邻居项的读写锁。

104 unsigned char ha[ALIGN(MAX_ADDR_LEN, sizeof(unsigned long))]

与存储在 primary_key 中的三层协议地址相对应的二进制二层硬件地址。以太网地址长度为 6B，而其他链路层协议地址也许会更长，但通常不会超过 32B，因此该数组长度设定为 MAX_ADDR_LEN，即 32。

105 struct hh_cache *hh

指向缓存的二层协议首部 hh_cache 结构实例链表，参见 17.3.7 节。

106 atomic_t refcnt

引用计数。

107 int (*output)(struct sk_buff *skb)

输出函数，用来将报文输出到该邻居。在邻居项的整个生命周期中，由于其状态是不断变化的，从而导致该函数指针会指向不同的输出函数。例如，当该邻居可达时会调用 neigh_connect()将 output 设置为 neigh_ops->connected_output，参见 17.15 节。

108 struct sk_buff_head arp_queue

当邻居项状态处于无效时，用来缓存要发送的报文。如当邻居项处于 NUD_INCOMPLETE 状态时，发送第一个报文需要新的邻居项，调用 neigh_resolve_output()，要发送的报文被缓存到 arp_queue 队列中，在该邻居可达后，再从 arp_queue 队列中取出报文输出到该邻居。

109 struct timer_list timer

用来管理多种超时情况的定时器，参见 17.13 节。

110 struct neigh_ops *ops

指向邻居项函数指针表实例。每一种邻居协议都提供 3 到 4 种不同的邻居项函数指针表，实际用哪一种还需要根据三层协议地址的类型、网络设备的类型等，参见 17.3.3 节。

111 u8 primary_key[0]

存储哈希函数使用的三层协议地址。该实际使用空间是根据三层协议地址长度动态分配的，例如，IPv4 为 32 位目标 IP 地址。

17.3.3 neigh_ops 结构

邻居项函数指针表由在邻居的生存周期中不同时期被调用的多个函数指针组成。其中有多个函数指针是实现三层（IPv4 中的 IP 层）与 dev_queue_xmit()之间的调用桥梁，适用于不同的状态，另参见 18.5 节。

neigh_ops 结构实际上是一个函数指针表，包含了一组函数指针，这些函数在一个 neighbour 实例的整个生命周期内会被使用到，由此实现了三层和二层的 dev_queue_xmit()之间的转接。

```
114 struct neigh_ops
115 {
116     int             family;
117     void            (*solicit)(struct neighbour *, struct sk_buff*);
118     void            (*error_report)(struct neighbour *, struct sk_buff*);
119     int             (*output)(struct sk_buff*);
120     int             (*connected_output)(struct sk_buff*);
121     int             (*hh_output)(struct sk_buff*);
122     int             (*queue_xmit)(struct sk_buff*);
123 };
```

116 int family

标识所属的地址族，比如 ARP 为 AF_INET 等。

117 void (*solicit)(struct neighbour *, struct sk_buff*)

发送请求报文函数。在发送第一个报文时，需要新的邻居项，发送报文被缓存到 arp_queue 队列中，然后会调用 solicit()发送请求报文，参见 17.13 节。

118 void (*error_report)(struct neighbour *, struct sk_buff*)

当邻居项缓存着未发送的报文，而该邻居项又不可达时，被调用来向三层报告错误的函数，参见 17.13 节。ARP 中为 arp_error_report()，最终会给报文发送方发送一个主机不可达的 ICMP 差错报文。

119 int (*output)(struct sk_buff*)

最通用的输出函数，可用于所有情况。此输出函数实现了完整的输出过程，因此存在较多的校验与操作，以确保报文的输出，因此该函数相对较消耗资源。此外，不要将 neigh_ops->output()与 neighbour->output()混淆。

120 int (*connected_output)(struct sk_buff*)

在确定邻居可达时，即状态为 NUD_CONNECTED 时使用的输出函数。由于所有输出所需要的信息都已具备，因此该函数只是简单地添加二层首部，也因此比 output()快得多。

121 int (*hh_output)(struct sk_buff*)

在已缓存了二层首部的情况下使用的输出函数，参见 17.15.3 节。

122 int (*queue_xmit)(struct sk_buff*)

实际上，以上几个输出接口，除了 hh_output 外，并不真正传输数据包，只是在准备好二层首部之后，调用 queue_xmit 接口。

17.3.4　neigh_parms 结构

邻居协议参数配置块，用于存储可调节的邻居协议参数，如重传超时时间、proxy_queue 队列长度等。一个邻居协议对应一个参数配置块，而每一个网络设备的 IPv4 的配置块中也存在一个存放默认值的邻居配置块。

```
34  struct neigh_parms
35  {
36      struct net_device *dev;
37      struct neigh_parms *next;
38      int    (*neigh_setup)(struct neighbour *);
39      void    (*neigh_destructor)(struct neighbour *);
40      struct neigh_table *tbl;
41
42      void    *sysctl_table;
43
44      int dead;
45      atomic_t refcnt;
46      struct rcu_head rcu_head;
47
48      int    base_reachable_time;
49      int    retrans_time;
50      int    gc_staletime;
51      int    reachable_time;
52      int    delay_probe_time;
53
54      int    queue_len;
55      int    ucast_probes;
56      int    app_probes;
57      int    mcast_probes;
58      int    anycast_delay;
59      int    proxy_delay;
60      int    proxy_qlen;
61      int    locktime;
62  };
```

36 struct net_device *dev

指向该 neigh_parms 实例所对应的网络设备，在通过 neigh_parms_alloc()创建 neigh_parms 实例时设置。

37 struct neigh_parms *next

通过 next 将属于同一个协议族的所有 neigh_parms 实例链接在一起，每个 neigh_table 实例都有各自的 neigh_parms 队列。

38 int (*neigh_setup)(struct neighbour *)

39 void (*neigh_destructor)(struct neighbour *)

提供给那些仍在使用老式接口设备的初始化和销毁接口。net_device 结构中也有一个 neigh_setup 成员函数指针，不要与之混淆。

40 struct neigh_table *tbl

指向该 neigh_parms 实例所属的邻居表。

42 void *sysctl_table

邻居表的 sysctl 表，对 ARP 是在 ARP 模块初始化函数 arp_init()中对其初始化的，这样用户可以通过 proc 文件系统来读写邻居表的参数。

```
44 int dead
```

该字段值如果为 1，则该邻居参数实例正在被删除，不能再使用，也不能再创建对应网络设备的邻居项。例如，在网络设备禁用时调用 neigh_parms_release()设置。

```
45 atomic_t refcnt
```

引用计数。

```
46 struct rcu_head rcu_head
```

为控制同步访问而设置的参数。

```
48 int base_reachable_time
```

```
51 int reachable_time
```

base_reachable_time 为计算 reachable_time 的基准值；而 reachable_time 为 NUD_REACHABLE 状态超时时间，该值为随机值，介于 base_reachable_time 和 1.5 倍的 base_reachable_time 之间，通常每 300s 在 neigh_periodic_timer()中更新一次。

```
49 int retrans_time
```

用于重传 ARP 请求报文的超时时间。主机在输出一个 ARP 请求报文之后的 retrans_time 个 jiffies 内，如果没有接收到应答报文，则会重新输出一个新的 ARP 请求报文。

```
50 int gc_staletime
```

一个邻居项如果持续闲置（没有被使用）时间达到 gc_staletime，且没有被引用，则会将被删除。

```
52 int delay_probe_time
```

邻居项维持在 NUD_DELAY 状态 delay_probe_time 之后进入 NUD_PROBE 状态；或者，处于 NUD_REACHABLE 状态的邻居项闲置时间超过 delay_probe_time 后，直接进入 NUD_DELAY 状态，参见 17.13 节。

```
54 int queue_len
```

proxy_queue 队列长度上限。

```
55 int ucast_probes
```

发送并确认可达的单播 ARP 请求报文数目。

```
56 int app_probes
```

地址解析时，应用程序（通常是 arpd）可发送 ARP 请求报文的数目，参见 18.7.2 节、17.13 节和 17.15.2 节。

```
57 int mcast_probes
```

为了解析一个邻居地址，可发送的广播 ARP 请求报文数目。需要注意的是 app_probes 和 mcast_probes 之间是互斥的，ARP 发送的是多播报文，而非广播报文。

```
58 int anycast_delay
```

未使用。

```
59 int proxy_delay
```

处理代理请求报文可延时的时间，参见 17.14.2 节。

```
60 int proxy_qlen
```

proxy_queue 队列的长度的上限。

```
61 int locktime
```

当邻居项最近两次更新的时间间隔小于该值时，用覆盖的方式来更新邻居项。例如，当有多个在同一网段的代理 ARP 服务器答复对相同地址的查询，参见 18.11.2 节。

17.3.5　pneigh_entry 结构

pneigh_entry 结构实例用来保存允许代理的条件，只有和结构中的接收设备以及目标地址相匹配才能代理。所有 pneigh_entry 实例都存储在邻居表的 phash_buckets 散列表中，称之为代理项。可通过 ip neigh add proxy 命令添加。

```
125 struct pneigh_entry
126 {
127     struct pneigh_entry    *next;
128     struct net_device       *dev;
129     u8          flags;
130     u8          key[0];
131 };
```

```
127 struct pneigh_entry *next
```

将 pneigh_entry 结构实例链接到 phash_buckets 散列表的一个桶内。

```
128 struct net_device *dev
```

通过该网络设备接收到的 ARP 请求报文才能代理。

```
129 u8 flags
```

标志，见表 17-2。

表 17-2　flags 的取值

Flags	描述
NTF_PROXY	代理表项标志。用 ip 命令在代理的邻居时会添加此标志，比如 ip neigh add proxy 10.0.0.4 dev eth0

```
130 u8 key[0]
```

存储三层协议地址，存储空间根据 neigh_table 结构的 key_len 字段分配。只有目标地址和该三层协议地址匹配的 ARP 请求报文才能代理。

17.3.6　neigh_statistics 结构

neigh_statistics 结构用来存储统计信息，一个该结构实例对应一个网络设备上的一种邻居协议。

```
64  struct neigh_statistics
65  {
66      unsigned long allocs;        /* number of allocated neighs */
67      unsigned long destroys;       /* number of destroyed neighs */
68      unsigned long hash_grows;    /* number of hash resizes */
69
70      unsigned long res_failed;    /* nomber of failed resolutions */
71
72      unsigned long lookups;        /* number of lookups */
73      unsigned long hits;       /* number of hits (among lookups) */
74
```

```
75      unsigned long rcv_probes_mcast;    /* number of received mcast ipv6 */
76      unsigned long rcv_probes_ucast; /* number of received ucast ipv6 */
77
78      unsigned long periodic_gc_runs;    /* number of periodic GC runs */
79      unsigned long forced_gc_runs;    /* number of forced GC runs */
80  };
```

66 unsigned long allocs

记录已分配的 neighbour 结构实例总数，包括已释放的实例。

67 unsigned long destroys

在 neigh_destroy()中删除的邻居项总数。

68 unsigned long hash_grows

扩容 hash_buckets 散列表的次数，参见 17.8 节。

70 unsigned long res_failed

尝试解析一个邻居地址的失败次数。这并不是发送 ARP 请求报文的次数，而是对于一个邻居来说，在 neigh_timer_handler()中所有尝试都失败之后才进行计数。

72 unsigned long lookups

调用 neigh_lookup()的总次数。

73 unsigned long hits

调用 neigh_lookup()成功返回总次数。

75 unsigned long rcv_probes_mcast

76 unsigned long rcv_probes_ucast

IPv6 分别用来标识接收到发往组播或单播地址的 ARP 请求报文总数。

78 unsigned long periodic_gc_runs

79 unsigned long forced_gc_runs

分别记录调用 neigh_periodic_timer()或 neigh_forced_gc()的次数。

17.3.7 hh_cache 结构

hh_cache（hardware header）结构用来缓存二层首部，这样就可以复制而不是逐个域地设置二层首部，从而加速报文的输出，但也并非所有的设备驱动程序都需要缓存二层首部。

```
192 struct hh_cache
193 {
194     struct hh_cache *hh_next;    /* Next entry                    */
195     atomic_t    hh_refcnt;    /* number of users                    */
196 /*
197  * We want hh_output, hh_len, hh_lock and hh_data be a in a separate
198  * cache line on SMP.
199  * They are mostly read, but hh_refcnt may be changed quite frequently,
200  * incurring cache line ping pongs.
201  */
202     be16        hh_type ____cacheline_aligned_in_smp;
203                 /* protocol identifier, f.e ETH_P_IP
204                                  *  NOTE:  For VLANs, this will be the
205                                  *  encapuslated type. --BLG
206                                  */
207     u16         hh_len;         /* length of header */
```

```
208     int       (*hh_output)(struct sk_buff *skb);
209     seqlock_t   hh_lock;
210
211     /* cached hardware header; allow for machine alignment needs.        */
212 #define HH_DATA_MOD   16
213 #define HH_DATA_OFF(__len) \
214     (HH_DATA_MOD - (((__len - 1) & (HH_DATA_MOD - 1)) + 1))
215 #define HH_DATA_ALIGN(__len) \
216     (((__len)+(HH_DATA_MOD-1))&~(HH_DATA_MOD - 1))
217     unsigned long   hh_data[HH_DATA_ALIGN(LL_MAX_HEADER) / sizeof(long)];
218 };
```

194 struct hh_cache *hh_next

通过 hh_next 将同属于一个邻居项的多个 hh_cache 实例链接起来，参见 17.15.2 节的 neigh_hh_init()。

195 atomic_t hh_refcnt

引用计数。

202 __be16 hh_type

缓存的硬件首部中指明的三层协议类型。

207 u16 hh_len

缓存的二层首部长度。

208 int (*hh_output)(struct sk_buff *skb)

报文输出函数，如同 neigh 结构的 output 一样，由 neigh->ops 中的某个输出接口初始化。

209 seqlock_t hh_lock

用于保护 hh_cache 的自旋锁。

217 unsigned long hh_data[HH_DATA_ALIGN(LL_MAX_HEADER)/sizeof(long)]

用来存放二层首部，对于以太网则是以太网帧的首部。

17.4　邻居表的初始化

邻居表由 neigh_table_init()初始化。对 arp_tbl 的初始化，在 ARP 模块初始化时由 arp_init()调用。实际上，邻居表的初始化工作大部分是由 neigh_table_init_no_netlink()完成的，neigh_table_init()只是调用该函数，然后将初始化后的邻居表链接到 neigh_tables 链表上。

邻居表的初始化实质上是初始化 neigh_table 结构中的成员。初始化过程如下所示：

- 以 base_reachable_time 为基准值调用随机函数来初始化 reachable_time。
- 如果还未分配该邻居表的邻居项缓存池，则分配之。
- 分配邻居表的 neigh_statistics 实例。
- 如果编译时启用了 proc 文件系统，则在该文件系统中创建相应的文件，/proc/net/stat/arp_cache 文件。
- 创建存储邻居项的 hash_buckets 散列表。
- 创建存储代理项的 phash_buckets 散列表。
- 得到用来计算 hash_buckets 散列表关键字的随机值。
- 初始化垃圾回收定时器 gc_timer。

- 初始化处理代理 ARP 报文定时器 proxy_timer。

17.5 邻居项的状态机

邻居项有一个对于管理和维护邻居表来说非常重要的成员，nud_state，用来表示该邻居项当前所处的状态。邻居项主要在七种重要状态之间进行转换，以下一一细述。

邻居表通过垃圾回收定时器定期扫描所有邻居表项，一方面从邻居表中清除状态为 NUD_FAILED，以及长时间未被使用的邻居表项，即上次使用时间距当前时间已超过垃圾收集过期时间的表项；另一方面识别那些确认时间已超时的邻居表项，将它们从连接状态（NUD_CONNECTED）改为过期状态 NUD_STALE，状态迁移图如图 17-2 所示。

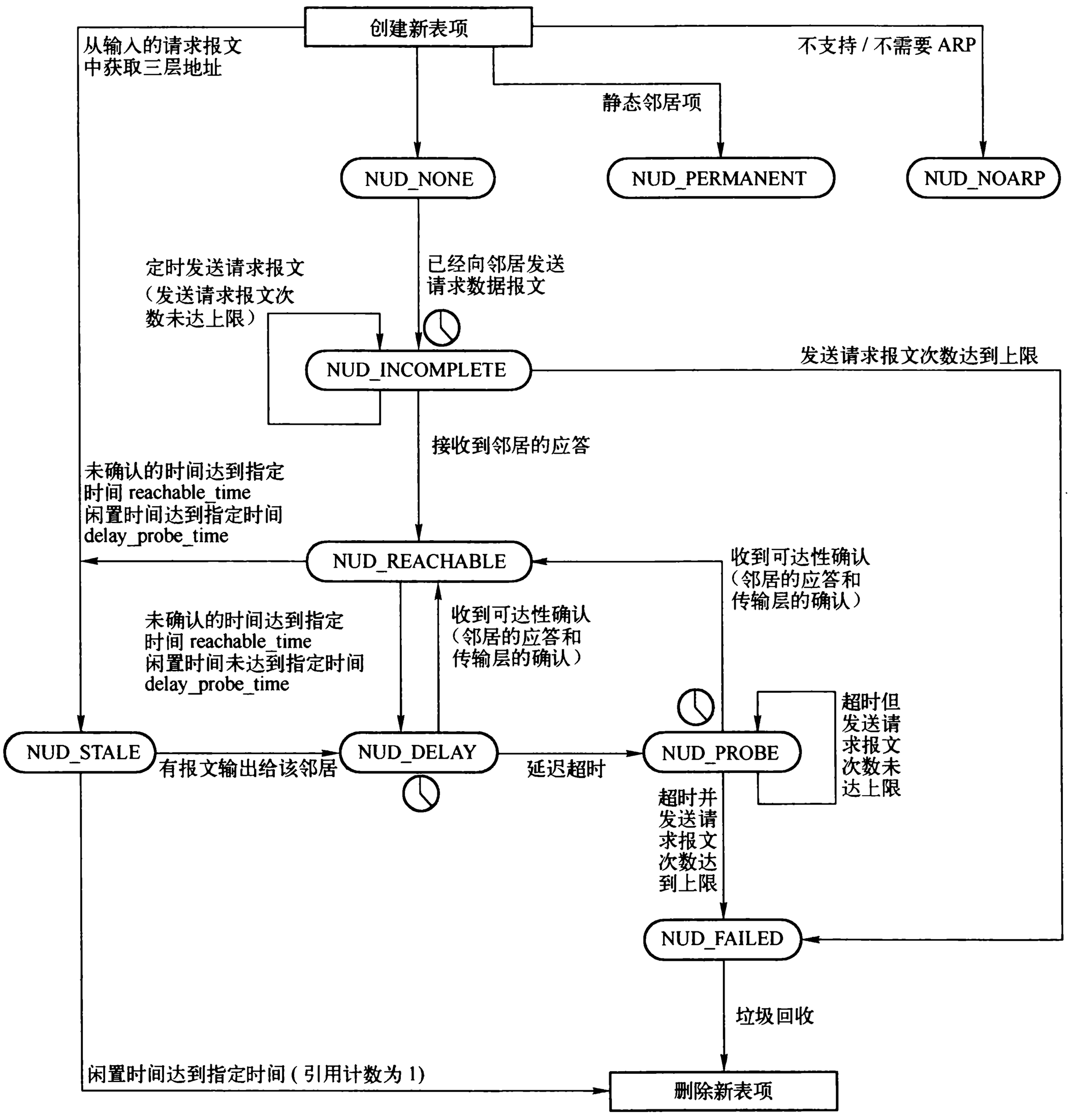

图 17-2 邻居项状态迁移

各种状态如下所示：

（1）NUD_NONE 状态

邻居项刚建立时处于的初始状态。

在此状态下，还没有硬件地址可使用，因此还不能发送请求报文。此时，一旦有报文要输出到该邻居，便会触发对该邻居硬件地址的请求，进入 NUD_INCOMPLETE 状态，并缓存发送的报文。参见 17.7.1 节的 neigh_alloc()和 17.15.2 节的 neigh_resolve_output()。

（2）NUD_INCOMPLETE 状态

请求报文已发送，但尚未收到应答的状态。

在此状态下，还没有解析到硬件地址，因此尚无可用的硬件地址，如果有报文要输出到此邻居，会将其先缓存起来。当进入此状态时，会启动一个定时器，如果在定时器到期时还未接收到邻居的回应，则会重复发送请求报文，直至解析成功或者尝试发送请求报文的次数达到上限，解析成功进入 NUD_REACHABLE 状态，否则如果尝试发送请求报文的次数达到上限，便进入 NUD_FAILED 状态。

（3）NUD_REACHABLE 状态

可达状态，已经得到并缓存了邻居的硬件地址。

进入该状态时，首先设置邻居项相关的 output 函数指针（该状态下使用 neigh_ops 结构的 connected_output），然后查看是否存在要发送给该邻居的报文，如果有，则将其发送出去。如果在该状态下闲置时间达到指定上限时，便会进入 NUD_STALE 状态，参见 17.13 节。

（4）NUD_STALE 状态

过期状态。

在该状态一旦有报文要输出到该邻居，则会进入 NUD_DELAY 状态并将该报文输出；如果在该状态闲置时间达到指定上限，且此时的引用计数为 1，则通过垃圾回收机制将其删除，参见 17.15.2 节。

（5）NUD_DELAY 状态

报文已发出，需得到邻居的可达性确认的状态。

在该状态在延迟的指定时间内未收到确认，便会进入 NUD_PROBE 状态，否则进入 NUD_REACHABLE 状态。在该状态下，报文的输出不受限制，使用慢速发送过程。

（6）NUD_PROBE 状态

过渡状态，类似 NUD_INCOMPLETE 状态。

在未接收到邻居的应答或确认时也会定时地重发请求，直至收到邻居的应答、确认或尝试发送请求报文的次数达到上限，如果收到邻居的应答或确认，则进入 NUD_REACHABLE 状态；如果尝试发送请求报文的次数达到上限，则进入 NUD_FAILED 状态。在该状态下，报文的输出也不受限制，使用慢速发送过程。

（7）NUD_FAILED 状态

由于没有接收到应答而无法访问状态。

从状态变迁图中可以看到，有两种情况下邻居项会进入 NUD_FAILED 状态，一是在刚创建时有报文要发送，但解析地址不成功，二是邻居项处于 NUD_PROBE 状态是有报文要发送，却没有收到应答或确认。

（8）NUD_NOARP 状态

标识邻居无需将三层协议地址映射到二层地址协议的支持。

（9）NUD_PERMANENT 状态

该状态一般通过应用层命令设置，邻居项的硬件地址已静态配置，也无需将三层协议地址映射到二层地址协议的支持，也不会被垃圾回收，参见 17.11 节。

以上为邻居项的各个状态，但在代码中根据各个状态的特征将其分成了三大类状态，每类状态都是多个状态的组合。

```
#define NUD_IN_TIMER    (NUD_INCOMPLETE|NUD_REACHABLE|NUD_DELAY|NUD_PROBE)
#define   NUD_VALID   (NUD_PERMANENT|NUD_NOARP|NUD_REACHABLE|NUD_PROBE|NUD_STALE|
NUD_DELAY)
#define NUD_CONNECTED    (NUD_PERMANENT|NUD_NOARP|NUD_REACHABLE)
```

（1）NUD_IN_TIMER

定时器状态，表示邻居项在此类状态下设置一个定时器，其实这在状态变迁图上已经能很明白地看到那些状态是设置了定时器的。

（2）NUD_VALID

有效状态，表示在这些状态下该邻居项是有效的。

（3）NUD_CONNECTED

连接状态，在这些状态下可以直接发送数据包给该邻居。

17.6 邻居项的添加与删除

在多种情况下会引发邻居项的添加，例如，在应用层通过 ip 和 arp 命令可以添加邻居项；在添加的路由项与路径项绑定时也会触发创建邻居项；当接收到并非请求的应答同样可能会触发创建邻居项。

邻居项的删除，接口比较统一，除了通过 ip 和 arp 命令可以实现外，其他都是在垃圾回收中删除，参见 17.11 节。

17.6.1 netlink 接口

Linux 为了兼容 UNIX，也提供 net-tools 包中的有关命令，如 arp，它们都是通过 ioctl 接口对路由进行相应的操作和配置的。此外 linux 还提供了功能更为强大的配置工具——IPROUTE2 包，它则是通过 linux 特有的 netlink 接口对路由进行相应的操作和配置的。

图 17-3 展示了通过这 netlink 接口来操作路由表的主要函数。

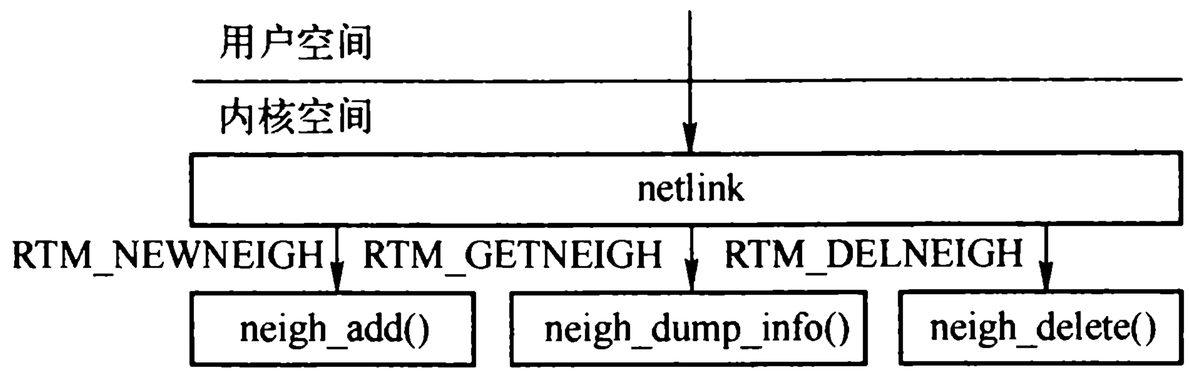

图 17-3 rtnetlink 接口

图 17-4 为添加、删除和获取邻居表项的 netlink 消息结构。

其中各字段的意义如下：

（1）ndm_family

8 位，标识操作的邻居项所属的地址族，如 AF_INET、AF_INET6 等。

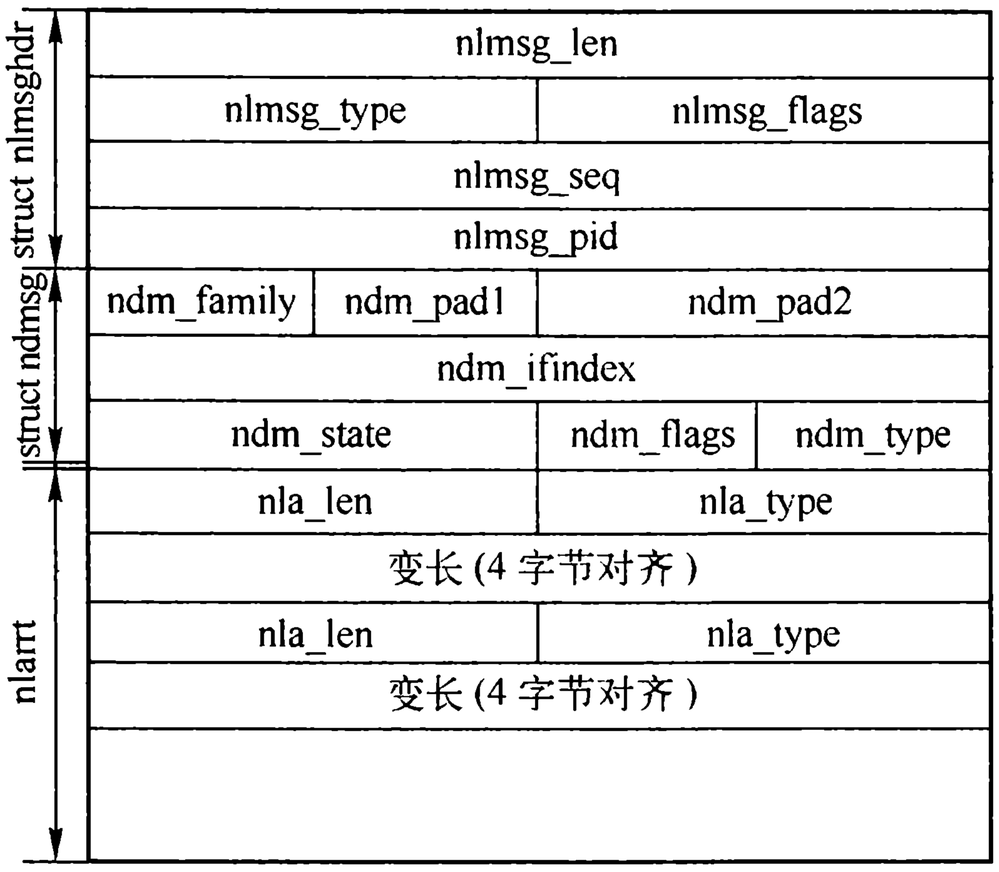

图 17-4　有关操作邻居项的 netlink 消息

（2）ndm_ifindex

32 位，邻居项的网络设备索引。

（3）ndm_state

16 位，邻居项的状态，即在 17.5 节中描述的那些状态。

（4）ndm_flags

8 位，邻居项标志，见表 17-3。

表 17-3　ndm_flags 的取值

Flags	描述
NTF_PROXY	操作的是一个代理项，参见 17.14 节
NTF_ROUTER	邻居是个 IPv6 路由器

（5）ndm_type

邻居项地址的类型。参见 neighbour 结构的 type 成员。

（6）扩展属性

用来描述邻居项的一些扩展属性，这些扩展属性值与协议相关，有些是变长的，如邻居三层协议地址长度在 IPv4 中为 4B，而在 IPv6 中为 16B；有些是可选的，因此扩展属性的数目也是可变的。扩展属性的类型如表 17-4 所示。

表 17-4　邻居项的扩展属性

属性	描述
NDA_UNSPEC	无效的类型
NDA_DST	邻居的三层协议地址，IPv4 中为 IP 地址
NDA_LLADDR	邻居的二层地址，以太网中为以太网地址
NDA_CACHEINFO	获取邻居项的统计信息，包括该邻居接收到报文最新的时间，最后使用时间，最后一次被更新的时间和当前引用计数。只适用于 RTM_GETNEIGH 类型的操作，参见 neigh_fill_info() 和 nda_cacheinfo 结构
NDA_PROBES	获取邻居项重试发送 ARP 报文请求的次数。只适用于 RTM_GETNEIGH 类型的操作，参见 neigh_fill_info()

通过 netlink，当操作类型为 RTM_NEWNEIGH 对邻居进行配置时，neigh_add()就被激活。通常，使用的命令为 ip neighbour add/change/replace。

当操作类型为 RTM_DELNEIGH 时，执行命令 ip neighbour del，neigh_delete()被激活。neigh_delete()与 neigh_add()类似，首先解析配置邻居的消息，然后查找对应的邻居项并删除。

```
1503 int neigh_add(struct sk_buff *skb, struct nlmsghdr *nlh, void *arg)
1504 {
1505     struct ndmsg *ndm;
1506     struct nlattr *tb[NDA_MAX+1];
1507     struct neigh_table *tbl;
1508     struct net_device *dev = NULL;
1509     int err;
1510
1511     err = nlmsg_parse(nlh, sizeof(*ndm), tb, NDA_MAX, NULL);
1512     if (err < 0)
1513         goto out;
1514
1515     err = -EINVAL;
1516     if (tb[NDA_DST] == NULL)
1517         goto out;
```

从消息的后部，即邻居信息之后，获取变长的扩展属性，并校验确保存在 NDA_DST 类型的扩展属性。

```
1519     ndm = nlmsg_data(nlh);
1520     if (ndm->ndm_ifindex) {
1521         dev = dev_get_by_index(ndm->ndm_ifindex);
1522         if (dev == NULL) {
1523             err = -ENODEV;
1524             goto out;
1525         }
1526
1527         if (tb[NDA_LLADDR] && nla_len(tb[NDA_LLADDR]) < dev->addr_len)
1528             goto out_dev_put;
1529     }
```

由邻居项的网络设备索引获取对应的网络设备。如果存在二层地址扩展属性，则需校验。

```
1531     read_lock(&neigh_tbl_lock);
1532     for (tbl = neigh_tables; tbl; tbl = tbl->next) {
1533         int flags = NEIGH_UPDATE_F_ADMIN | NEIGH_UPDATE_F_OVERRIDE;
1534         struct neighbour *neigh;
1535         void *dst, *lladdr;
1536
1537         if (tbl->family != ndm->ndm_family)
1538             continue;
1539         read_unlock(&neigh_tbl_lock);
1540
1541         if (nla_len(tb[NDA_DST]) < tbl->key_len)
1542             goto out_dev_put;
1543         dst = nla_data(tb[NDA_DST]);
1544         lladdr = tb[NDA_LLADDR] ? nla_data(tb[NDA_LLADDR]) : NULL;
1545
1546         if (ndm->ndm_flags & NTF_PROXY) {
```

```
1547            struct pneigh_entry *pn;
1548
1549            err = -ENOBUFS;
1550            pn = pneigh_lookup(tbl, dst, dev, 1);
1551            if (pn) {
1552                pn->flags = ndm->ndm_flags;
1553                err = 0;
1554            }
1555            goto out_dev_put;
1556        }
1557
1558        if (dev == NULL)
1559            goto out_dev_put;
1560
1561        neigh = neigh_lookup(tbl, dst, dev);
1562        if (neigh == NULL) {
1563            if (!(nlh->nlmsg_flags & NLM_F_CREATE)) {
1564                err = -ENOENT;
1565                goto out_dev_put;
1566            }
1567
1568            neigh = __neigh_lookup_errno(tbl, dst, dev);
1569            if (IS_ERR(neigh)) {
1570                err = PTR_ERR(neigh);
1571                goto out_dev_put;
1572            }
1573        } else {
1574            if (nlh->nlmsg_flags & NLM_F_EXCL) {
1575                err = -EEXIST;
1576                neigh_release(neigh);
1577                goto out_dev_put;
1578            }
1579
1580            if (!(nlh->nlmsg_flags & NLM_F_REPLACE))
1581                flags &= ~NEIGH_UPDATE_F_OVERRIDE;
1582        }
1583
1584        err = neigh_update(neigh, lladdr, ndm->ndm_state, flags);
1585        neigh_release(neigh);
1586        goto out_dev_put;
1587    }
1588
1589    read_unlock(&neigh_tbl_lock);
1590     err = -EAFNOSUPPORT;
....
1597 }
```

1532-1538 遍历 neigh_tables 链表中所有的邻居表，获取与消息中给出的地址族相一致的邻居表。

1541-1544 从扩展属性值中获取相关的信息等待处理。

1546-1556 调用 pneigh_lookup()添加一个代理项，参见 17.14.1 节。

1558-1559 添加邻居项前，确保该邻居的输出网络设备不能为空。

1561 调用 neigh_lookup()根据邻居项的地址以及输出网络设备，在邻居表的邻居散列表中查找对应的邻居项。

1562-1572 如果没有在邻居表中找到对应的邻居项，且 netlink 添加邻居项消息首部的 nlmsg_flags 域中存在 NLM_F_CREATE 标志，该标志表示不存在即创建之，则调用 neigh_lookup_errno()创建并添加相应的邻居项到散列表中。

1573-1582 在邻居表已存在对应的邻居项的情况下，如果 netlink 添加邻居项消息首部的 nlmsg_flags 域中有 NLM_F_EXCL 标志，该标志表示如已存在则不作替换，则返回已存在错误码 EEXIST；如果 nlmsg_flags 域不存在 NLM_F_REPLACE 标志，即替换现有的邻居项，则在将调用的 neigh_update()参数 flags 中去掉 NEIGH_UPDATE_F_OVERRIDE 标志，flags 标志见表 17-5。

1584-1586 最后调用 neigh_update()更新指定的邻居项，参见 17.10 节。

17.6.2 ioctl

为了兼容 UNIX 命令，Linux 除了提供功能强大的 netlink 之外，还提供了 ioctl 接口，因此传统的 UNIX 命令可以通过 ioctl 对邻居表项进行添加和删除，参见 18.13 节。

17.6.3 路由表项与邻居项的绑定

当调用 arp_bind_neighbour()将路由缓存项与邻居绑定时，如果路由项不存在对应的邻居项，则会创建一个邻居项，然后与之绑定，参见 18.15 节。

17.6.4 接收到的并非请求的应答

对于那些并非由请求接收到的应答，由于此时邻居项并不存在，如果系统允许接收，需在邻居表中为该应答创建邻居项，参见 18.11.2 节。

17.7 邻居项的创建与初始化

17.7.1 neigh_alloc()

在创建邻居项时，需先分配一邻居项实例，分配成功后才能继续后续的初始化。分配邻居项实例的工作由 neigh_alloc()实现，其唯一的参数 tbl 为待分配邻居项所在的邻居表。

```
239 static struct neighbour *neigh_alloc(struct neigh_table *tbl)
240 {
241     struct neighbour *n = NULL;
242     unsigned long now = jiffies;
243     int entries;
244
245     entries = atomic_inc_return(&tbl->entries) - 1;
246     if (entries >= tbl->gc_thresh3 ||
247         (entries >= tbl->gc_thresh2 &&
248          time_after(now, tbl->last_flush + 5 * HZ))) {
249         if (!neigh_forced_gc(tbl) &&
250             entries >= tbl->gc_thresh3)
251             goto out_entries;
252     }
```

首先递增待分配邻居项所属邻居表中邻居项的数目，如果该数目超过 gc_thresh3，或者超

过 gc_thresh2 且已超过五秒未刷新，则必须立即刷新并强制垃圾回收。如果垃圾回收失败且邻居项数目大于 gc_thresh3，则不能分配新的邻居项实例。垃圾回收参见 17.11 节。

```
254     n = kmem_cache_alloc(tbl->kmem_cachep, GFP_ATOMIC);
255     if (!n)
256         goto out_entries;
257
258     memset(n, 0, tbl->entry_size);
```

在缓冲池中分配邻居项实例并清零。

```
260     skb_queue_head_init(&n->arp_queue);
261     rwlock_init(&n->lock);
262     n->updated      = n->used = now;
263     n->nud_state      = NUD_NONE;
264     n->output      = neigh_blackhole;
265     n->parms      = neigh_parms_clone(&tbl->parms);
266     init_timer(&n->timer);
267     n->timer.function = neigh_timer_handler;
268     n->timer.data      = (unsigned long)n;
269
270     NEIGH_CACHE_STAT_INC(tbl, allocs);
271     n->tbl          = tbl;
272     atomic_set(&n->refcnt, 1);
273     n->dead          = 1;
274 out:
275     return n;
276
277 out_entries:
278     atomic_dec(&tbl->entries);
279     goto out;
280 }
```

邻居项实例分配后，需作一些与协议无关的初始化，如更新使用时间；当前的状态设置为 NUD_NONE；从所在邻居表的配置参数克隆配置参数；初始化状态处理定时器等。

17.7.2　neigh_create()

neigh_create()用来完整地创建一个邻居项，并将其添加到散列表上，最后返回指向该邻居项的指针。参数说明如下：

- tbl，待创建邻居项所属的邻居表，在 ARP 中为 arp_tbl。
- pkey，三层协议地址，作为散列表的关键字。
- dev，该邻居项的输出设备。

```
383 struct neighbour *neigh_create(struct neigh_table *tbl, const void *pkey,
384                   struct net_device *dev)
385 {
386     u32 hash_val;
387     int key_len = tbl->key_len;
388     int error;
389     struct neighbour *n1, *rc, *n = neigh_alloc(tbl);
390
391     if (!n) {
```

```
392         rc = ERR_PTR(-ENOBUFS);
393         goto out;
394     }
```

调用 neigh_alloc()分配一个邻居项实例，只有分配成功后才能进行后续的与协议相关的初始化。

```
396     memcpy(n->primary_key, pkey, key_len);
397     n->dev = dev;
398     dev_hold(dev);
399
400     /* Protocol specific setup. */
401     if (tbl->constructor &&    (error = tbl->constructor(n)) < 0) {
402         rc = ERR_PTR(error);
403         goto out_neigh_release;
404     }
405
406     /* Device specific setup. */
407     if (n->parms->neigh_setup &&
408         (error = n->parms->neigh_setup(n)) < 0) {
409         rc = ERR_PTR(error);
410         goto out_neigh_release;
411     }
```

396-397 将三层协议地址和输出设备设置到邻居项中。需要注意的是，由于邻居子系统支持多种邻居协议，因此在设置三层协议地址时要根据其长度 key_len 来复制，参见 17.3.1 节。

401-404 执行与协议相关的初始化函数，ARP 中为 arp_constructor()，参见 18.8 节。

407-411 执行设备启动函数，IPv4 中无需该函数的支持。

```
413     n->confirmed = jiffies - (n->parms->base_reachable_time << 1);
414
415     write_lock_bh(&tbl->lock);
416
417     if (atomic_read(&tbl->entries) > (tbl->hash_mask + 1))
418         neigh_hash_grow(tbl, (tbl->hash_mask + 1) << 1);
419
420     hash_val = tbl->hash(pkey, dev) & tbl->hash_mask;
421
422     if (n->parms->dead) {
423         rc = ERR_PTR(-EINVAL);
424         goto out_tbl_unlock;
425     }
426
427     for (n1 = tbl->hash_buckets[hash_val]; n1; n1 = n1->next) {
428         if (dev == n1->dev && !memcmp(n1->primary_key, pkey, key_len)) {
429             neigh_hold(n1);
430             rc = n1;
431             goto out_tbl_unlock;
432         }
433     }
434
435     n->next = tbl->hash_buckets[hash_val];
436     tbl->hash_buckets[hash_val] = n;
437     n->dead = 0;
```

```
438     neigh_hold(n);
439     write_unlock_bh(&tbl->lock);
440     NEIGH_PRINTK2("neigh %p is created.\n", n);
441     rc = n;
442 out:
443     return rc;
444 out_tbl_unlock:
445     write_unlock_bh(&tbl->lock);
446 out_neigh_release:
447     neigh_release(n);
448     goto out;
449 }
```

413 初始化邻居项的确认时间。

417-418 当邻居项所属的邻居表，其邻居项数目当计入当前正在创建的邻居项后，超过了 hash_buckets 散列表的容量时，就需将散列表扩容一倍，参见 17.8 节。

422-425 邻居项配置参数正在被删除，不能再使用，因此也就不能再继续创建邻居项了。

427-434 遍历邻居表中对应的散列表桶，如果相应的邻居项已存在，则使用已有的邻居项，递增其引用计数，释放新创建的邻居项。

435-441 如果不存在，则把邻居项添加到散列表中，并更新 dead 标志，递增引用计数。

17.8　邻居项散列表的扩容

在创建邻居项时，如果在计入要创建的邻居项后，邻居表邻居项的数目超过了邻居散列表的容量，就会调用 neigh_hash_grow()扩容邻居散列表，参数说明如下：

- tbl，待扩容邻居项散列表所属的邻居表，ARP 中为 arp_tbl。
- new_entries，扩容后邻居项散列表的容量。

```
306 static void neigh_hash_grow(struct neigh_table *tbl, unsigned long new_
    entries)
307 {
308     struct neighbour **new_hash, **old_hash;
309     unsigned int i, new_hash_mask, old_entries;
310
311     NEIGH_CACHE_STAT_INC(tbl, hash_grows);
312
313     BUG_ON(new_entries & (new_entries - 1));
314     new_hash = neigh_hash_alloc(new_entries);
315     if (!new_hash)
316         return;
317
318     old_entries = tbl->hash_mask + 1;
319     new_hash_mask = new_entries - 1;
320     old_hash = tbl->hash_buckets;
321
322     get_random_bytes(&tbl->hash_rnd, sizeof(tbl->hash_rnd));
323     for (i = 0; i < old_entries; i++) {
324         struct neighbour *n, *next;
325
```

```
326         for (n = old_hash[i]; n; n = next) {
327             unsigned int hash_val = tbl->hash(n->primary_key, n->dev);
328
329             hash_val &= new_hash_mask;
330             next = n->next;
331
332             n->next = new_hash[hash_val];
333             new_hash[hash_val] = n;
334         }
335     }
336     tbl->hash_buckets = new_hash;
337     tbl->hash_mask = new_hash_mask;
338
339     neigh_hash_free(old_hash, old_entries);
340 }
```

314-316 调用 neigh_hash_alloc()为邻居项散列表重新分配内存，在该函数中，根据待分配的内存量大于 PAGE_SIZE 与否来确定使用 kzalloc()或是 get_free_pages()分配内存。

322 重新计算随机值 hash_rnd。

323-337 先将原先邻居项散列表中的邻居项移动到扩容后的邻居项散列表中，然后将新邻居项散列表及其 hash_mask 保存到邻居表中。

339 调用 neigh_hash_free()释放旧邻居项散列表所占用的内存。

17.9 邻居项的查找

17.9.1 neigh_lookup()

邻居项的查找非常频繁：添加邻居项时需查找邻居项是否已存在；删除邻居项时需查找待删除的邻居项是否存在。neigh_lookup()用来查找邻居项，函数有三个参数：tbl 为待查找的邻居表，pkey 和 dev 是查找条件，即三层协议地址和邻居项的输出网络设备。

```
342 struct neighbour *neigh_lookup(struct neigh_table *tbl, const void *pkey,
343                   struct net_device *dev)
344 {
345     struct neighbour *n;
346     int key_len = tbl->key_len;
347     u32 hash_val = tbl->hash(pkey, dev);
348
349     NEIGH_CACHE_STAT_INC(tbl, lookups);
350
351     read_lock_bh(&tbl->lock);
352     for (n = tbl->hash_buckets[hash_val & tbl->hash_mask]; n; n = n->next) {
353         if (dev == n->dev && !memcmp(n->primary_key, pkey, key_len)) {
354             neigh_hold(n);
355             NEIGH_CACHE_STAT_INC(tbl, hits);
356             break;
357         }
358     }
359     read_unlock_bh(&tbl->lock);
360     return n;
361 }
```

遍历邻居项所在的桶，根据 pkey 和 dev 进行比较，如果查找成功则返回查找到的邻居项，否则返回 NULL。

17.9.2　neigh_lookup_nodev()

邻居子系统还提供了其他几个用于查找邻居项的函数，neigh_lookup_nodev()，顾名思义，在查找邻居项时只比较三层协议地址，原型如下：

```
struct neighbour *neigh_lookup_nodev(struct neigh_table *tbl, const void *pkey)
```

17.9.3　__neigh_lookup()和 neigh_lookup_errno()

__neigh_lookup()在查找失败并允许创建新的邻居项时，根据查找条件创建新邻居项。而__neigh_lookup_errno()则是查找失败直接创建新邻居项。原型如下：

```
static inline struct neighbour *__neigh_lookup(struct neigh_table *tbl, const
void *pkey, struct net_device *dev, int creat)

static inline struct neighbour *neigh_lookup_errno(struct neigh_table *tbl,
const void *pkey,  struct net_device *dev)
```

17.10　邻居项的更新

neigh_update()用来更新指定的邻居项，更新的内容是硬件地址和状态。在更新状态之后，会根据新状态设置相应的输出函数：如果新状态为 NUD_CONNECTED，则允许快速路径发送，更新输出函数为 neigh_connect()；否则禁止快速路径发送，更新输出函数为 neigh_suspect()。

该函数在邻居子系统和 ARP 模块中使用相当频繁，如：

- 在 arp_process()中接收到邻居的应答后需更新对应邻居项的硬件地址和状态。
- 在 arp_process()中接收到邻居的请求或转发的单播代理请求后，需将对应邻居项状态更新到 NUD_STALE 状态。
- 用 netlink 或 ioctl 命令添加或删除邻居项，也是通过该函数来更新硬件地址和状态的。

neigh_update()有四个参数，说明如下：

- neigh，待更新的邻居项。
- lladdr，新的硬件地址，虽然参数指定了硬件地址，但在处理中根据状态等条件还可能会进行调整。
- new，新的状态。
- flags，更新标志，见表 17-5。

表 17-5　邻居项更新标志

Flags	描述
NEIGH_UPDATE_F_OVERRIDE	如果存在邻居项则允许覆盖
NEIGH_UPDATE_F_WEAK_OVERRIDE	如果更新前后的硬件不同，则允许用先前处于 NUD_CONNECTED 时的硬件地址。如果硬件地址没有变化，则允许保持当前的状态

（续）

Flags	描述
NEIGH_UPDATE_F_ADMIN	当邻居项原先状态为 NUD_NOARP 或 NUD_PERMANENT 时，必须存在该标志，说明是由于用户管理而发生改变，否则不允许更新
NEIGH_UPDATE_F_OVERRIDE_ISROUTER	允许覆盖 NTF_ROUTER 标志的邻居项
NEIGH_UPDATE_F_ISROUTER	标识该邻居是个路由器

```
931  int neigh_update(struct neighbour *neigh, const u8 *lladdr, u8 new,
932           u32 flags)
933  {
934      u8 old;
935      int err;
936      int notify = 0;
937      struct net_device *dev;
938      int update_isrouter = 0;
939
940      write_lock_bh(&neigh->lock);
941
942      dev    = neigh->dev;
943      old    = neigh->nud_state;
944      err    = -EPERM;
945
946      if (!(flags & NEIGH_UPDATE_F_ADMIN) &&
947          (old & (NUD_NOARP | NUD_PERMANENT)))
948          goto out;
949
950      if (!(new & NUD_VALID)) {
951          neigh_del_timer(neigh);
952          if (old & NUD_CONNECTED)
953              neigh_suspect(neigh);
954          neigh->nud_state = new;
955          err = 0;
956          notify = old & NUD_VALID;
957          goto out;
958      }
```

946-948 当邻居项原先的状态为 NUD_NOARP 或 NUD_PERMANENT 时，必须存在 NEIGH_UPDATE_F_ADMIN 更新标志，说明是由于用户管理而发生改变，否则不允许更新。

950-958 如果新状态不在所有的有效状态之内，则停止邻居项定时器；如果原先处于 NUD_CONNECTED 状态，则调用 neigh_suspect()更新其输出函数；然后更新状态。此外如果状态是从有效转变为无效，则需要通知感兴趣的模块，因此用 notify 变量记录下来。

```
960      /* Compare new lladdr with cached one */
961      if (!dev->addr_len) {
962          /* First case: device needs no address. */
963          lladdr = neigh->ha;
964      } else if (lladdr) {
965          /* The second case: if something is already cached
966             and a new address is proposed:
967             - compare new & old
968             - if they are different, check override flag
969           */
970          if ((old & NUD_VALID) &&
```

```
971             !memcmp(lladdr, neigh->ha, dev->addr_len))
972             lladdr = neigh->ha;
973     } else {
974         /* No address is supplied; if we know something,
975            use it, otherwise discard the request.
976          */
977         err = -EINVAL;
978         if (!(old & NUD_VALID))
979             goto out;
980         lladdr = neigh->ha;
981     }
```

根据状态等条件调整硬件地址，调整过程如下：

- 如果标识的硬件地址长度为 0，则说明网络设备不需要硬件地址，因此从邻居项中获取硬件地址。
- 如果用于设置硬件地址的参数有效，并且更新前邻居项处于有效状态，同时更新的硬件地址与保存在邻居项中硬件地址不同，则使用保存在邻居项中硬件地址，否则使用通过参数传入的硬件地址。
- 如果待更新的硬件地址为空，且原先状态为有效状态，则从邻居项中获取硬件地址，否则不能继续更新邻居项。

```
983     if (new & NUD_CONNECTED)
984         neigh->confirmed = jiffies;
985     neigh->updated = jiffies;
```

983 如果新状态为 NUD_CONNECTED，说明邻居处理连接状态，可以直接根据该邻居项发送数据包，因此需更新确认时间。

985 设置更新时间。

```
987     /* If entry was valid and address is not changed,
988        do not change entry state, if new one is STALE.
989     */
990     err = 0;
991     update_isrouter = flags & NEIGH_UPDATE_F_OVERRIDE_ISROUTER;
992     if (old & NUD_VALID) {
993         if (lladdr != neigh->ha && !(flags & NEIGH_UPDATE_F_OVERRIDE)) {
994             update_isrouter = 0;
995             if ((flags & NEIGH_UPDATE_F_WEAK_OVERRIDE) &&
996                 (old & NUD_CONNECTED)) {
997                 lladdr = neigh->ha;
998                 new = NUD_STALE;
999             } else
1000                goto out;
1001        } else {
1002            if (lladdr == neigh->ha && new == NUD_STALE &&
1003                ((flags & NEIGH_UPDATE_F_WEAK_OVERRIDE) ||
1004                 (old & NUD_CONNECTED))
1005                )
1006                new = old;
1007        }
1008    }
```

处理原先为有效状态，NEIGH_UPDATE_F_OVERRIDE_ISROUTER 标志只在 IPv6 存在：

- 如果更新前后的硬件地址不同，且不存在 NEIGH_UPDATE_F_OVERRIDE 更新标志，则在存在 NEIGH_UPDATE_F_WEAK_OVERRIDE 更新标志且原状态为 NUD_CONNECTED 的情况下，更新硬件地址，并将状态转变为 NUD_STALE。
- 如果硬件地址没有变化，或者存在 NEIGH_UPDATE_F_OVERRIDE 更新标志，则在硬件地址不变，新状态为 NUD_STALE，且或者存在 NEIGH_UPDATE_F_WEAK_OVERRIDE 更新标志，或者原状态为 NUD_CONNECTED 的情况下，保持当前的状态。

```
1010     if (new != old) {
1011        neigh_del_timer(neigh);
1012        if (new & NUD_IN_TIMER) {
1013           neigh_hold(neigh);
1014           neigh_add_timer(neigh, (jiffies +
1015                     ((new & NUD_REACHABLE) ?
1016                      neigh->parms->reachable_time :
1017                      0)));
1018        }
1019        neigh->nud_state = new;
1020     }
1021
1022     if (lladdr != neigh->ha) {
1023        memcpy(&neigh->ha, lladdr, dev->addr_len);
1024        neigh_update_hhs(neigh);
1025        if (!(new & NUD_CONNECTED))
1026           neigh->confirmed = jiffies -
1027                     (neigh->parms->base_reachable_time << 1);
1028        notify = 1;
1029     }
1030     if (new == old)
1031        goto out;
1032     if (new & NUD_CONNECTED)
1033        neigh_connect(neigh);
1034     else
1035        neigh_suspect(neigh);
1036     if (!(old & NUD_VALID)) {
1037        struct sk_buff *skb;
1038
1039        /* Again: avoid dead loop if something went wrong */
1040
1041        while (neigh->nud_state & NUD_VALID &&
1042               (skb = __skb_dequeue(&neigh->arp_queue)) != NULL) {
1043           struct neighbour *n1 = neigh;
1044           write_unlock_bh(&neigh->lock);
1045           /* On shaper/eql skb->dst->neighbour != neigh :( */
1046           if (skb->dst && skb->dst->neighbour)
1047              n1 = skb->dst->neighbour;
1048           n1->output(skb);
1049           write_lock_bh(&neigh->lock);
1050        }
1051        skb_queue_purge(&neigh->arp_queue);
1052     }
```

1010-1020 在更新状态之前需先停止邻居项状态处理定时器，如果新的状态为定时状态之一，则再重新启动定时器，然后再更新状态。

1022-1029 将新硬件地址设置到邻居项的二层协议首部缓存中，调用 neigh_update_hhs()将硬件地址同时更新到邻居项对应网络设备的二层协议首部缓存中。如果更新到非 NUD_CONNECTED 状态则还需修改确认时间，最后通知感兴趣的模块。

1030-131 如果状态不变，则无需后续的操作了。

1032-1035 如果更新到 NUD_CONNECTED，则调用 neigh_connect()更新输出函数为 connected_output()，允许快速路径发送；否则调用 neigh_suspect()更新输出函数为 output()，禁止快速路径发送。

1036-1052 如果邻居项是从无效状态更新为有效状态，则遍历邻居项请求缓存队列 arp_queue，将缓存在队列中的报文逐个输出。

```
1053 out:
1054     if (update_isrouter) {
1055         neigh->flags = (flags & NEIGH_UPDATE_F_ISROUTER) ?
1056             (neigh->flags | NTF_ROUTER) :
1057             (neigh->flags & ~NTF_ROUTER);
1058     }
1059     write_unlock_bh(&neigh->lock);
1060
1061     if (notify)
1062         call_netevent_notifiers(NETEVENT_NEIGH_UPDATE, neigh);
1063 #ifdef CONFIG_ARPD
1064     if (notify && neigh->parms->app_probes)
1065         neigh_app_notify(neigh);
1066 #endif
1067     return err;
1068 }
```

1054-1058 如果是允许覆盖 NTF_ROUTER 标志的邻居项，则根据是否存在 NEIGH_UPDATE_F_ISROUTER 更新标志，来添加或删除邻居项的 NTF_ROUTER 标志。

1061-1062 向感兴趣的模块通知 NETEVENT_NEIGH_UPDATE 事件。

1064-1065 如果编译时支持 ARPD，则还需要通知 ARPD 进程。

17.11　垃圾回收

通常垃圾回收是通过周期性定时器处理函数来异步清理的，把状态为 NUD_FAILED 且引用计数为 1 的邻居项，以及闲置时间超过指定上限 gc_staletime 且引用计数为 1 的邻居项清理掉。

当在某些极端情况下，会在创建邻居项时，由于邻居项的数目达到上限，从而触发一个同步清理。同步清理时，会将引用计数为 1 且非静态的邻居项全部清除。

17.11.1　同步回收

在创建新的邻居项时，如果检测到所属邻居表的邻居项数量已达到上限，就会触发一个同步清理。尽管邻居子系统提供了定期的异步回收机制，但在紧急情况下还是有可能会触发同步回收。

```
120 static int neigh_forced_gc(struct neigh_table *tbl)
121 {
122     int shrunk = 0;
123     int i;
124
125     NEIGH_CACHE_STAT_INC(tbl, forced_gc_runs);
126
127     write_lock_bh(&tbl->lock);
128     for (i = 0; i <= tbl->hash_mask; i++) {
129         struct neighbour *n, **np;
130
131         np = &tbl->hash_buckets[i];
132         while ((n = *np) != NULL) {
133             /* Neighbour record may be discarded if:
134              * - nobody refers to it.
135              * - it is not permanent
136              */
137             write_lock(&n->lock);
138             if (atomic_read(&n->refcnt) == 1 &&
139                 !(n->nud_state & NUD_PERMANENT)) {
140                 *np    = n->next;
141                 n->dead = 1;
142                 shrunk  = 1;
143                 write_unlock(&n->lock);
144                 neigh_release(n);
145                 continue;
146             }
147             write_unlock(&n->lock);
148             np = &n->next;
149         }
150     }
151
152     tbl->last_flush = jiffies;
153
154     write_unlock_bh(&tbl->lock);
155
156     return shrunk;
157 }
```

在同步清理时，会遍历所有的邻居项（而不像异步回收时，只搜索散列表的一个桶），将引用计数为 1 且非静态的邻居项全部清除。最后返回是否执行了清理的标志，返回值为 1 表示执行了清理，0 表示没有清理邻居项。

17.11.2 异步回收

同步垃圾回收被用来处理创建新邻居项时邻居表的邻居项数量达到上限的情况。但最好是在创建新邻居项之前避免出现这种极端情况，也就是说，在创建时进行垃圾回收这种现象越少越好。这正是为什么要通过周期性定时器来异步清理实现垃圾回收的原因。

在初始化邻居子系统时，neigh_table_init_no_netlink()中启动定时器 gc_timer，该定时器的处理函数是 neigh_periodic_timer()。每次 neigh_periodic_timer()被激活，都只扫描邻居散列表的一个桶，为此保持了一个变量 hash_chain_gc，用来记住上一次函数激活时扫描过的桶，当函数再次被激活时就扫描下一个桶，完成后重新设定定时器。

如果邻居项状态为 NUD_FAILED 且引用计数为 1，或者邻居项闲置时间超过规定上限且引用计数为 1，则调用 neigh_release()进行删除。

```
636 static void neigh_periodic_timer(unsigned long arg)
637 {
638     struct neigh_table *tbl = (struct neigh_table *)arg;
639     struct neighbour *n, **np;
640     unsigned long expire, now = jiffies;
641
642     NEIGH_CACHE_STAT_INC(tbl, periodic_gc_runs);
643
644     write_lock(&tbl->lock);
645
646     /*
647      *    periodically recompute ReachableTime from random function
648      */
649
650     if (time_after(now, tbl->last_rand + 300 * HZ)) {
651         struct neigh_parms *p;
652         tbl->last_rand = now;
653         for (p = &tbl->parms; p; p = p->next)
654             p->reachable_time =
655                 neigh_rand_reach_time(p->base_reachable_time);
656     }
```

每 300s 将邻居表所有 neigh_parms 结构实例的 NUD_REACHABLE 状态超时时间 reachable_time 更新为一个新的随机值。

```
658     np = &tbl->hash_buckets[tbl->hash_chain_gc];
659     tbl->hash_chain_gc = ((tbl->hash_chain_gc + 1) & tbl->hash_mask);
660
661     while ((n = *np) != NULL) {
662         unsigned int state;
663
664         write_lock(&n->lock);
665
666         state = n->nud_state;
667         if (state & (NUD_PERMANENT | NUD_IN_TIMER)) {
668             write_unlock(&n->lock);
669             goto next_elt;
670         }
671
672         if (time_before(n->used, n->confirmed))
673             n->used = n->confirmed;
674
675         if (atomic_read(&n->refcnt) == 1 &&
676             (state == NUD_FAILED ||
677              time_after(now, n->used + n->parms->gc_staletime))) {
678             *np = n->next;
679             n->dead = 1;
680             write_unlock(&n->lock);
681             neigh_release(n);
```

```
682             continue;
683         }
684         write_unlock(&n->lock);
685
686 next_elt:
687         np = &n->next;
688     }
```

658-659 获取此次处理的桶，同时设置下次处理的桶号。

661 遍历该桶上的所有的邻居项，以便删除释放符合条件的邻居项。

667-670 对于那些静态邻居项或处于定时器状态的邻居项不处理直接跳过。

672-673 如果邻居项的最后使用时间在最后确认时间之前，则调整最后使用时间为最后确认时间。

675-683 调用 neigh_release()删除释放符合两种条件的邻居项:

- 引用计数为 1 且状态为 NUD_FAILED;
- 引用计数为 1 且闲置时间超过了指定上限 gc_staletime。

```
690     /* Cycle through all hash buckets every base_reachable_time/2 ticks.
691      * ARP entry timeouts range from 1/2 base_reachable_time to 3/2
692      * base_reachable_time.
693      */
694     expire = tbl->parms.base_reachable_time >> 1;
695     expire /= (tbl->hash_mask + 1);
696     if (!expire)
697         expire = 1;
698
699     mod_timer(&tbl->gc_timer, now + expire);
700
701     write_unlock(&tbl->lock);
702 }
```

重新设置垃圾回收定时器 gc_timer 的下次过期时间，因为完成一次扫描回收整个邻居项散列表的时间是(reachable_time/2)Hz，因此下次激活时间为 reachable_time/(2*散列表桶数)。

17.12 外部事件

邻居子系统提供了 neigh_ifdown 接口，用来响应网络设备 NETDEV_UNREGISTER 事件。如果邻居协议模块对此事件感兴趣，会调用该接口。例如，对 IPv4/ARP，arp_ifdown()被调用，通过邻居协议模块再调用到 neigh_ifdown()，使邻居子系统能及时的处理该事件。neigh_ifdown()将删除释放与禁用网络设备相关的邻居项和代理项。如果由于邻居项的引用计数大于 1 而无法删除邻居项，则会将邻居项的 output 函数指针设置为 neigh_blackhole()，这样一旦调用该邻居的输出接口就会直接丢弃要发送的报文。而对于定时器和缓存的代理请求报文，则简单地删除定时器和缓存的代理请求报文。

邻居子系统还提供了 neigh_changeaddr 接口，用来响应网络设备的 NETDEV_CHANGEADDR 事件。如果邻居协议模块对此事件感兴趣，会调用此接口删除释放与禁用网络设备相关的邻居项，参见 18.14 节。

17.13　邻居项状态处理定时器

邻居项各个状态中，有些属于定时状态，对于这些状态其转变由定时器处理函数来处理。每个邻居项都有一个定时器，该定时器在创建邻居项时被初始化，它的处理函数为 neigh_timer_handler()，处理流程参见图 17-2。

```
723 static void neigh_timer_handler(unsigned long arg)
724 {
725     unsigned long now, next;
726     struct neighbour *neigh = (struct neighbour *)arg;
727     unsigned state;
728     int notify = 0;
729
730     write_lock(&neigh->lock);
731
732     state = neigh->nud_state;
733     now = jiffies;
734     next = now + HZ;
735
736     if (!(state & NUD_IN_TIMER)) {
737 #ifndef CONFIG_SMP
738         printk(KERN_WARNING "neigh: timer & !nud_in_timer\n");
739 #endif
740         goto out;
741     }
```

不处理那些不处于定时状态的邻居项。

```
743     if (state & NUD_REACHABLE) {
744         if (time_before_eq(now,
745                 neigh->confirmed + neigh->parms->reachable_time)) {
746             NEIGH_PRINTK2("neigh %p is still alive.\n", neigh);
747             next = neigh->confirmed + neigh->parms->reachable_time;
748         } else if (time_before_eq(now,
749                 neigh->used + neigh->parms->delay_probe_time)) {
750             NEIGH_PRINTK2("neigh %p is delayed.\n", neigh);
751             neigh->nud_state = NUD_DELAY;
752             neigh->updated = jiffies;
753             neigh_suspect(neigh);
754             next = now + neigh->parms->delay_probe_time;
755         } else {
756             NEIGH_PRINTK2("neigh %p is suspected.\n", neigh);
757             neigh->nud_state = NUD_STALE;
758             neigh->updated = jiffies;
759             neigh_suspect(neigh);
760             notify = 1;
761         }
762     } else if (state & NUD_DELAY) {
763         if (time_before_eq(now,
764                 neigh->confirmed + neigh->parms->delay_probe_time)) {
765             NEIGH_PRINTK2("neigh %p is now reachable.\n", neigh);
```

```
766             neigh->nud_state = NUD_REACHABLE;
767             neigh->updated = jiffies;
768             neigh_connect(neigh);
769             notify = 1;
770             next = neigh->confirmed + neigh->parms->reachable_time;
771         } else {
772             NEIGH_PRINTK2("neigh %p is probed.\n", neigh);
773             neigh->nud_state = NUD_PROBE;
774             neigh->updated = jiffies;
775             atomic_set(&neigh->probes, 0);
776             next = now + neigh->parms->retrans_time;
777         }
778     } else {
779         /* NUD_PROBE|NUD_INCOMPLETE */
780         next = now + neigh->parms->retrans_time;
781     }
```

743-761 处理 NUD_REACHABLE 状态：

- 距离上次确认时间还未达到指定时间 reachable_time，则维持在 NUD_REACHABLE 状态，同时取得新的定时器到期时间，为上次确认时间后的 reachable_time。
- 距离上次确认时间已达到指定时间 reachable_time，但距离上次使用时间，即闲置时间，还未达到指定时间 delay_probe_time，则状态转变为 NUD_DELAY，修改更新时间，调用 neigh_suspect()重新设置输出函数，最后计算新的定时器到期时间，为当前之后 delay_probe_time。
- 距离上次确认时间已达到指定时间 reachable_time，且闲置时间也已经未达到指定时间 delay_probe_time 时，则状态转变为 NUD_STALE，同样需修改更新时间，重新设置输出函数，变量 notify 记录是否需通知 NETEVENT_NEIGH_UPDATE 事件。

762-777 处理 NUD_DELAY 状态：

- 距离上次确认时间未达到指定时间 delay_probe_time，则状态转变为 NUD_REACHABLE，修改更新时间，调用 neigh_connect()更新输出函数，计算定时器下次到期时间，为上次确认时间后 reachable_time，此外从 NUD_DELAY 状态转变为 UND_REACHABLE 状态也需要通知 NETEVENT_NEIGH_UPDATE 事件。
- 距离上次确认时间已达到指定时间 delay_probe_time，则转换到 NUD_PROBE 状态，修改更新时间，复位 probes，即尝试发送请求报文的次数，最后计算定时器下次到期时间，为当前之后 retrans_time。

778-781 NUD_PROBE 和 NUD_INCOMPLETE 这两种状态下计算定时器下次到期时间。

```
783     if ((neigh->nud_state & (NUD_INCOMPLETE | NUD_PROBE)) &&
784         atomic_read(&neigh->probes) >= neigh_max_probes(neigh)) {
785         struct sk_buff *skb;
786
787         neigh->nud_state = NUD_FAILED;
788         neigh->updated = jiffies;
789         notify = 1;
790         NEIGH_CACHE_STAT_INC(neigh->tbl, res_failed);
791         NEIGH_PRINTK2("neigh %p is failed.\n", neigh);
792
793         /* It is very thin place. report_unreachable is very complicated
```

```
794             routine. Particularly, it can hit the same neighbour entry!
795
796             So that, we try to be accurate and avoid dead loop. --ANK
797          */
798         while (neigh->nud_state == NUD_FAILED &&
799                (skb = __skb_dequeue(&neigh->arp_queue)) != NULL) {
800             write_unlock(&neigh->lock);
801             neigh->ops->error_report(neigh, skb);
802             write_lock(&neigh->lock);
803         }
804         skb_queue_purge(&neigh->arp_queue);
805     }
```

NUD_PROBE 和 NUD_INCOMPLETE 两种状态下，尝试发送请求报文的次数大于上限，则状态转变为 NUD_FAILED，然后删除释放缓存队列上的报文，并发送错误报告给报文的发送方，在 ARP 中是发送目的不可达 ICMP 消息，参见 18.7.1 节。

```
807     if (neigh->nud_state & NUD_IN_TIMER) {
808         if (time_before(next, jiffies + HZ/2))
809             next = jiffies + HZ/2;
810         if (!mod_timer(&neigh->timer, next))
811             neigh_hold(neigh);
812     }
813     if (neigh->nud_state & (NUD_INCOMPLETE | NUD_PROBE)) {
814         struct sk_buff *skb = skb_peek(&neigh->arp_queue);
815         /* keep skb alive even if arp_queue overflows */
816         if (skb)
817             skb_get(skb);
818         write_unlock(&neigh->lock);
819         neigh->ops->solicit(neigh, skb);
820         atomic_inc(&neigh->probes);
821         if (skb)
822             kfree_skb(skb);
823     } else {
824 out:
825         write_unlock(&neigh->lock);
826     }
827     if (notify)
828         call_netevent_notifiers(NETEVENT_NEIGH_UPDATE, neigh);
829
830 #ifdef CONFIG_ARPD
831     if (notify && neigh->parms->app_probes)
832         neigh_app_notify(neigh);
833 #endif
834     neigh_release(neigh);
835 }
```

807-812 设置定时器下次到期时间。

813-826 NUD_INCOMPLETE 或 NUD_PROBE 状态下尝试发送请求的次数未达到上限，则根据缓存队列中的第一个报文，向邻居发送请求报文。

827-828 向感兴趣的模块通知 NETEVENT_NEIGH_UPDATE 事件。

831-832 如果编译时支持 ARPD，则还需要通知 ARPD 进程。

17.14 代理项

对于那些代理的请求报文，通常并不是无条件转发和处理的，首先要通过系统的允许，然后还要检测代理的请求报文其输入设备和目的地址是否存在代理项中，只有在才会处理该代理的请求报文。

所有的代理项都存储在邻居表的 phash_buckets 散列表中，可通过应用层命令 ip 或 arp 进行添加和删除。

17.14.1 代理项的查找、添加和删除

pneigh_lookup()用来根据三层协议地址和输入网络设备在邻居表中查找对应的代理项。该函数不仅有查找的功能，还可以创建代理项，当参数 create 不为 0 时，一旦根据条件查找失败，就会根据查找的条件创建一个代理项并添加到 phash_buckets 散列表中。函数原型为

```
struct pneigh_entry * pneigh_lookup(struct neigh_table *tbl, const void *pkey,
struct net_device *dev, int creat)
```

pneigh_delete()用来删除指定的代理项，删除成功返回 0。函数原型为

```
int pneigh_delete(struct neigh_table *tbl, const void *pkey, struct net_device
*dev)
```

17.14.2 延时处理代理的请求报文

1. pneigh_enqueue()

当在延时处理的代理请求报文时，会调用 pneigh_enqueue()将请求报文缓存到 proxy_queue 队列中，然后设置 proxy_timer 定时器，到定时器到期时再处理该请求报文。

```
1239 void pneigh_enqueue(struct neigh_table *tbl, struct neigh_parms *p,
1240              struct sk_buff *skb)
1241 {
1242     unsigned long now = jiffies;
1243     unsigned long sched_next = now + (net_random() % p->proxy_delay);
1244
1245     if (tbl->proxy_queue.qlen > p->proxy_qlen) {
1246         kfree_skb(skb);
1247         return;
1248     }
1249
1250     NEIGH_CB(skb)->sched_next = sched_next;
1251     NEIGH_CB(skb)->flags |= LOCALLY_ENQUEUED;
1252
1253     spin_lock(&tbl->proxy_queue.lock);
1254     if (del_timer(&tbl->proxy_timer)) {
1255         if (time_before(tbl->proxy_timer.expires, sched_next))
1256             sched_next = tbl->proxy_timer.expires;
1257     }
1258     dst_release(skb->dst);
1259     skb->dst = NULL;
```

```
1260     dev_hold(skb->dev);
1261     skb_queue_tail(&tbl->proxy_queue, skb);
1262     mod_timer(&tbl->proxy_timer, sched_next);
1263     spin_unlock(&tbl->proxy_queue.lock);
1264 }
```

1242-1243 由当前系统时间、随机数和 proxy_delay 计算该请求报文的延时时间。

1245-1248 如果邻居表的代理报文缓存队列长度已达到上限，则将报文丢弃。

1250-1251 将之前计算得到的延时时间和 LOCALLY_ENQUEUED 标志保存到该请求报文的控制块中。将存放到代理报文缓存队列的报文会添加 LOCALLY_ENQUEUED 标志。

1254-1257 先去活 proxy_timer 定时器，然后在原到期时间和计算得到的延期时间之中取近者为新到期时间。

1258-1261 把 skb 的路由缓存项置空后将其添加到 proxy_queue 队列中。

1262 最后重新设置 proxy_timer 定时器下次到期时间。

2. neigh_proxy_process()

proxy_timer 定时器是在 neigh_table_init_no_netlink() 中初始化的，其处理函数为 neigh_proxy_process()。每当 proxy_timer 到期时，该函数就会从缓存队列中逐个取出并处理报文，直至全部处理完毕。

```
1205 static void neigh_proxy_process(unsigned long arg)
1206 {
1207     struct neigh_table *tbl = (struct neigh_table *)arg;
1208     long sched_next = 0;
1209     unsigned long now = jiffies;
1210     struct sk_buff *skb;
1211
1212     spin_lock(&tbl->proxy_queue.lock);
1213
1214     skb = tbl->proxy_queue.next;
1215
1216     while (skb != (struct sk_buff *)&tbl->proxy_queue) {
1217         struct sk_buff *back = skb;
1218         long tdif = NEIGH_CB(back)->sched_next - now;
1219
1220         skb = skb->next;
1221         if (tdif <= 0) {
1222             struct net_device *dev = back->dev;
1223             __skb_unlink(back, &tbl->proxy_queue);
1224             if (tbl->proxy_redo && netif_running(dev))
1225                 tbl->proxy_redo(back);
1226             else
1227                 kfree_skb(back);
1228
1229             dev_put(dev);
1230         } else if (!sched_next || tdif < sched_next)
1231             sched_next = tdif;
1232     }
1233     del_timer(&tbl->proxy_timer);
1234     if (sched_next)
1235         mod_timer(&tbl->proxy_timer, jiffies + sched_next);
1236     spin_unlock(&tbl->proxy_queue.lock);
1237 }
```

遍历 proxy_queue 队列，如果延时的时间已经超出当前请求报文的延时时间，则将该请求报文从队列中取下，然后根据邻居表 proxy_redo 接口的有效性以及输出设备是否启用来决定是调用 proxy_redo()处理之还是丢弃之；否则，重新计算并设置 proxy_timer 定时器下次到期时间。

17.15 输出函数

邻居子系统中提供了多个输出报文的函数，如 neigh_connect()、neigh_suspect()，这些函数用来初始化 neigh_ops 结构实例，如 IPv4 中的 arp_generic_ops、arp_hh_ops 等。以便在邻居项状态发生变化时，根据状态选取适当的输出函数，参见 17.13 节。

17.15.1 丢弃

调用到 neigh_blackhole()时，待输出的报文会被毫不留情地丢弃。以下是会使用 neigh_blackhole()作为邻居项输出函数的情况：

- neigh_alloc()中分配了邻居项之后，作协议无关初始化时，会使用 neigh_blackhole()来初始化邻居项的 output。
- 当接收到网络设备的 NETDEV_UNREGISTER 事件时会删除释放与禁用网络设备相关的邻居项。此时如果由于邻居项的引用计数大于 1 而无法删除之，就会用 neigh_blackhole ()作为它的 output，这样一旦调用到其输出接口就会直接丢弃待输出报文。
- 在删除释放邻居项时，同样会删除释放邻居项中的二层首部缓存。此时如果该二层首部缓存的引用计数大于 1 而无法删除，也会用 neigh_blackhole()作为二层首部缓存的输出函数 hh_output()。

17.15.2 慢速发送

1. neigh_resolve_output()

当邻居项不处于 NUD_CONNECTED 状态时，不允许快速路径发送报文。函数 neigh_resolve_output()用于慢速而安全的输出，通常用来初始化 neigh_ops 结构实例的 output 函数指针，当邻居项从 NUD_CONNECTED 转到非 NUD_CONNECTED 状态，便会调用 neigh_suspect()将邻居项的 output 设置为 neigh_resolve_output()。

```
1138 int neigh_resolve_output(struct sk_buff *skb)
1139 {
1140     struct dst_entry *dst = skb->dst;
1141     struct neighbour *neigh;
1142     int rc = 0;
1143
1144     if (!dst || !(neigh = dst->neighbour))
1145         goto discard;
1146
1147     skb_pull(skb, skb->nh.raw - skb->data);
1148
1149     if (!neigh_event_send(neigh, skb)) {
1150         int err;
```

```
1151        struct net_device *dev = neigh->dev;
1152        if (dev->hard_header_cache && !dst->hh) {
1153            write_lock_bh(&neigh->lock);
1154            if (!dst->hh)
1155                neigh_hh_init(neigh, dst, dst->ops->protocol);
1156            err = dev->hard_header(skb, dev, ntohs(skb->protocol),
1157                           neigh->ha, NULL, skb->len);
1158            write_unlock_bh(&neigh->lock);
1159        } else {
1160            read_lock_bh(&neigh->lock);
1161            err = dev->hard_header(skb, dev, ntohs(skb->protocol),
1162                           neigh->ha, NULL, skb->len);
1163            read_unlock_bh(&neigh->lock);
1164        }
1165        if (err >= 0)
1166            rc = neigh->ops->queue_xmit(skb);
1167        else
1168            goto out_kfree_skb;
1169    }
1170 out:
1171    return rc;
1172 discard:
1173    NEIGH_PRINTK1("neigh_resolve_output: dst=%p neigh=%p\n",
1174              dst, dst ? dst->neighbour : NULL);
1175 out_kfree_skb:
1176    rc = -EINVAL;
1177    kfree_skb(skb);
1178    goto out;
1179 }
```

1149 确保用于输出的邻居项状态有效，才能发送数据包。

1150-1164 如果邻居项的输出设备支持 hard_header_cache，同时路由缓存项中的二层首部缓存尚未建立，则先为该路由缓存项建立硬件首部缓存，然后在待输出的报文前添加该硬件首部，否则直接在报文前添加硬件首部。

1165-1168 如果添加硬件首部成功，则调用 queue_xmit()将报文输出到网络设备。

2. __neigh_event_send()

neigh_event_send()用于检测邻居项状态是否有效，如果邻居项状态为 NUD_CONNECTED、NUD_DELAY 或 NUD_PROBE，可以直接发送，因此返回 0 表示有效；否则调用__neigh_event_send()作进一步检测，如果无效则丢弃报文。

```
837 int __neigh_event_send(struct neighbour *neigh, struct sk_buff *skb)
838 {
839    int rc;
840    unsigned long now;
841
842    write_lock_bh(&neigh->lock);
843
844    rc = 0;
845    if (neigh->nud_state & (NUD_CONNECTED | NUD_DELAY | NUD_PROBE))
846        goto out_unlock_bh;
847
848    now = jiffies;
849
```

```
850     if (!(neigh->nud_state & (NUD_STALE | NUD_INCOMPLETE))) {
851         if (neigh->parms->mcast_probes + neigh->parms->app_probes) {
852             atomic_set(&neigh->probes, neigh->parms->ucast_probes);
853             neigh->nud_state     = NUD_INCOMPLETE;
854             neigh->updated = jiffies;
855             neigh_hold(neigh);
856             neigh_add_timer(neigh, now + 1);
857         } else {
858             neigh->nud_state = NUD_FAILED;
859             neigh->updated = jiffies;
860             write_unlock_bh(&neigh->lock);
861
862             if (skb)
863                 kfree_skb(skb);
864             return 1;
865         }
866     } else if (neigh->nud_state & NUD_STALE) {
867         NEIGH_PRINTK2("neigh %p is delayed.\n", neigh);
868         neigh_hold(neigh);
869         neigh->nud_state = NUD_DELAY;
870         neigh->updated = jiffies;
871         neigh_add_timer(neigh,
372                 jiffies + neigh->parms->delay_probe_time);
873     }
874
875     if (neigh->nud_state == NUD_INCOMPLETE) {
876         if (skb) {
877             if (skb_queue_len(&neigh->arp_queue) >=
878                 neigh->parms->queue_len) {
879                 struct sk_buff *buff;
880                 buff = neigh->arp_queue.next;
881                 __skb_unlink(buff, &neigh->arp_queue);
882                 kfree_skb(buff);
883             }
884             __skb_queue_tail(&neigh->arp_queue, skb);
885         }
886         rc = 1;
887     }
888 out_unlock_bh:
889     write_unlock_bh(&neigh->lock);
890     return rc;
891 }
```

845-846 邻居项状态处于 NUD_CONNECTED、NUD_DELAY 或 NUD_PROBE 时，表示邻居项当前有效，直接可以发送，返回 0。

850-865 到此，剩下未考察的状态是 NUD_STALE、NUD_INCOMPLETE 和 NUD_NONE，因此如果当前的状态不为 NUD_STALE 或 NUD_INCOMPLETE，则必为 NUD_NONE。

如果允许发送广播请求报文，或者允许应用程序发送请求报文来解析邻居地址，则将邻居项状态转换到 NUD_INCOMPLETE，并启动邻居项状态处理定时器；否则邻居项只能转换到 NUD_FAILED 状态，并释放待输出的报文，同时返回 1，表示该邻居项无效，不能输出。

866-873 如果邻居项当前状态为 NUD_STALE，由于有报文输出了，因此状态转变为 NUD_DELAY，并设置邻居项状态处理定时器。状态 NUD_DELAY 表示可以输出，因此也返

回 0。

875-887 如果邻居项当前状态为 NUD_INCOMPLETE，说明请求报文已经发送，但尚未收到应答。此时如果请求缓存队列长度还未达到上限，则将待输出报文缓存到该队列中，否则只能丢弃该报文。无论是哪种情况都返回 1，表示还不能发送报文。

3. neigh_hh_init()

neigh_hh_init()实现通过邻居项为指定路由缓存项建立硬件首部缓存。

```
1082 static void neigh_hh_init(struct neighbour *n, struct dst_entry *dst,
1083                __be16 protocol)
1084 {
1085     struct hh_cache   *hh;
1086     struct net_device *dev = dst->dev;
1087
1088     for (hh = n->hh; hh; hh = hh->hh_next)
1089         if (hh->hh_type == protocol)
1090             break;
1091
1092     if (!hh && (hh = kzalloc(sizeof(*hh), GFP_ATOMIC)) != NULL) {
1093         seqlock_init(&hh->hh_lock);
1094         hh->hh_type = protocol;
1095         atomic_set(&hh->hh_refcnt, 0);
1096         hh->hh_next = NULL;
1097         if (dev->hard_header_cache(n, hh)) {
1098             kfree(hh);
1099             hh = NULL;
1100         } else {
1101             atomic_inc(&hh->hh_refcnt);
1102             hh->hh_next = n->hh;
1103             n->hh       = hh;
1104             if (n->nud_state & NUD_CONNECTED)
1105                 hh->hh_output = n->ops->hh_output;
1106             else
1107                 hh->hh_output = n->ops->output;
1108         }
1109     }
1110     if (hh)    {
1111         atomic_inc(&hh->hh_refcnt);
1112         dst->hh = hh;
1113     }
1114 }
```

1088-1090 根据协议在邻居项的硬件缓存列表中查找对应的硬件首部缓存。如果查找命中，则使用该硬件首部缓存为路由缓存项建立硬件首部缓存。

1092-1109 如果查找未果，则创建新的硬件首部缓存，并将其添加到邻居项的硬件缓存列表中，同时根据状态设置合适的 hh_output 函数指针。

1110-1113 将查找命中的或是新创建的硬件首部缓存设置到路由缓存项中。

17.15.3　快速发送

当调用了 dst_output()后根据数据报的目的路由缓存项输出数据报，对于单播数据报最终会调用到 ip_finish_output2()。在 ip_finish_output2()中如下代码所示，如果目的路由缓存项缓

存了链路层的首部，则调用 neigh_hh_output()输出报文。否则，若存在对应的邻居项，则通过邻居项的输出方法输出报文，参见 11.11.1 节。

```
164  static inline int ip_finish_output2(struct sk_buff *skb)
165  {
...
185      if (dst->hh)
186          return neigh_hh_output(dst->hh, skb);
187      else if (dst->neighbour)
188          return dst->neighbour->output(skb);
...
194  }
```

当邻居项处于 NUD_CONNECTED 状态时，可以快速路径发送报文。neigh_hh_output()和 neigh_connected_output()都是用于快速输出，这两个函数都不作邻居项状态有效性的检测。不同的是前者直接复制链路层的首部到数据报中，而后者需要逐步构建链路层首部，因此前者效率会更高。

相关函数如下：

（1）neigh_hh_output()

```
312 static inline int neigh_hh_output(struct hh_cache *hh, struct sk_buff *skb)
313 {
314     unsigned seq;
315     int hh_len;
316
317     do {
318         int hh_alen;
319
320         seq = read_seqbegin(&hh->hh_lock);
321         hh_len = hh->hh_len;
322         hh_alen = HH_DATA_ALIGN(hh_len);
323         memcpy(skb->data - hh_alen, hh->hh_data, hh_alen);
324     } while (read_seqretry(&hh->hh_lock, seq));
325
326     skb_push(skb, hh_len);
327     return hh->hh_output(skb);
328 }
```

（2）neigh_connected_output()

此函数初始化 neigh_ops 结构实例的 connected_output 函数指针。当邻居项从非 NUD_CONNECTED 转到 NUD_CONNECTED 状态，便调用 neigh_connect()将邻居项的 output 设置为 neigh_connected_output()。

```
1183 int neigh_connected_output(struct sk_buff *skb)
1184 {
1185     int err;
1186     struct dst_entry *dst = skb->dst;
1187     struct neighbour *neigh = dst->neighbour;
1188     struct net_device *dev = neigh->dev;
1189
1190     __skb_pull(skb, skb->nh.raw - skb->data);
1191
```

```
1192     read_lock_bh(&neigh->lock);
1193     err = dev->hard_header(skb, dev, ntohs(skb->protocol),
1194                            neigh->ha, NULL, skb->len);
1195     read_unlock_bh(&neigh->lock);
1196     if (err >= 0)
1197        err = neigh->ops->queue_xmit(skb);
1198     else {
1199        err = -EINVAL;
1200        kfree_skb(skb);
1201     }
1202     return err;
1203 }
```

1190-1195 在待输出的报文前添加硬件首部。以太网上，则添加以太网帧首部。

1196-1201 如果添加硬件首部成功，则调用 queue_xmit()将报文输出到网络设备。

第 18 章　ARP：地址解析协议

在以太网上传输 IP 数据报时，以太网设备并不能识别 32 位 IP 地址，而是以 48 位以太网地址传输以太网数据包的。因此，IP 数据报在以太网上传输前需要分装成以太网帧，而以太网帧的目的地址正是通过 IP 数据报的目的 IP 地址查询得到的。因此 IP 地址和以太网地址之间存在着映射，通过查看 ARP 表就可以得到这两地址间的对应关系。地址解析协议（Address Resolution Protocol——ARP）就是用来确定这些对应关系的协议。

当通过 ARP 根据 IP 地址获取以太网地址时，会发送出一个含有所希望的 IP 地址的以太网广播数据包。目的地主机收到该 ARP 查询报文后，会回应一个含有 IP 和以太网地址对的数据包作为应答。发送者会将得到的以太网地址缓存在 ARP 表中，用于以后数据报的发送，以节省 ARP 通信成本。

ARP 协议的处理涉及以下文件：

- include/linux/if_arp.h，定义 ARP 报文等结构、宏和函数原型。
- net/ipv4/arp.c，ARP 协议的实现。

18.1　ARP 报文格式

图 18-1 给出了 ARP 报文的格式。当发送 ARP 请求时，发送方填入发送方以太网地址、发送方 IP 地址以及目标 IP 地址。目标主机接收到这个 ARP 广播包时，会在响应报文中填上自己的以太网地址，即目标以太网地址。

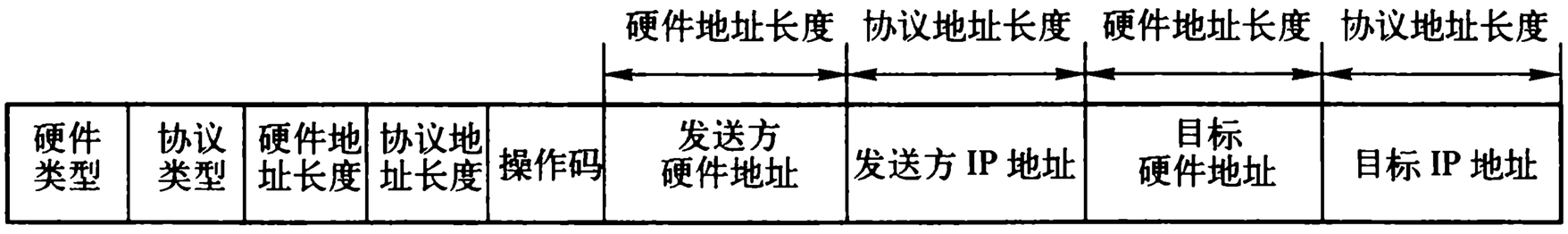

图 18-1　ARP 报文格式

具体格式如下：

（1）硬件类型

硬件地址类型。该字段的取值是一组定义在 include/linux/if_arp.h 中以 ARPHRD_打头的常量，如 ARPHRD_ETHER(1)表示以太网地址。

（2）协议类型

三层协议地址类型。该字段的取值是一组定义在 include/linux/if_ether.h 中以 ETH_P_打头的常量，如 ETH_P_IP(0x0800)即表示 IP 地址。

（3）硬件地址长度

硬件地址的长度，以字节为单位。对以太网地址来说，其值为 6，即 48 位。

（4）协议地址长度

协议地址的长度，以字节为单位。对 IP 地址来说，其值为 4，即 32 位。

（5）操作码

有四种操作类型：ARP 请求（ARPOP_REQUEST）、ARP 应答（ARPOP_REPLY）、RARP 请求（ARPOP_RREQUEST）和 RARP 应答（ARPOP_RREPLY）。ARPOP_REQUEST 和 ARPOP_REPLY 见表 18-1。

表 18-1　操作码

操作码	描述
ARPOP_REQUEST	用于发送一个通过 2 层 MAC 地址获取 3 层 IP 地址的请求。对于新的邻居项，主机会将请求报文发送到广播地址。而对于用来验证一个已存在的邻居项，主机会直接通过硬件地址发送请求报文到目标
ARPOP_REPLY	用于应答一个 ARPOP_REQUEST 操作码的 ARP 报文，通常它会直接发送到请求者，但有时也可以将其发送到广播地址

（6）发送方硬件地址与 IP 地址

（7）目标硬件地址与目标 IP 地址

系统中定义 ARP 报文的为 arphdr 结构。

```
131 struct arphdr
132 {
133     __be16          ar_hrd;         /* format of hardware address   */
134     __be16          ar_pro;         /* format of protocol address   */
135     unsigned char   ar_hln;         /* length of hardware address   */
136     unsigned char   ar_pln;         /* length of protocol address   */
137     __be16          ar_op;          /* ARP opcode (command)         */
138
139 #if 0
140      /*
141       *    Ethernet looks like this : This bit is variable sized however...
142       */
143     unsigned char       ar_sha[ETH_ALEN];   /* sender hardware address   */
144     unsigned char       ar_sip[4];          /* sender IP address         */
145     unsigned char       ar_tha[ETH_ALEN];   /* target hardware address   */
146     unsigned char       ar_tip[4];          /* target IP address         */
147 #endif
148
149 };
```

133-137 对应到图 18-1 中的硬件类型、协议类型、硬件地址长度、协议地址长度以及操作码。

143-146 由于不同网络介质的 MAC 地址长度是不同的，因此 ARP 报文的结构不能包括操作码后面的内容。这里只是列举了以太网上的 ARP 的定义。

ARP 协议并不仅仅被 IPv4 使用，在内核的网络模块代码中使用缩写 SIP 和 TIP（Source IP address 和 Target IP address）来代表发送方 IP 地址和目的 IP 地址。

18.2　系统参数

系统参数如下：

（1）arp_filter

是否允许从其他网络设备输出 ARP 应答，取值见表 18-2。

表 18-2 arp_filter 的取值

arp_filter	描述
0（默认值）	允许从其他网络设备输出 ARP 应答，这样能增大成功沟通的概率，因为 IP 地址是属于主机的而不是属于网络设备的。这种配置适合类似负载均衡复杂情况
1	允许有多个网络设备在同一个子网上，而且每个接口的被应答的 ARP 的输出是基于 ARP 中的源 IP 地址来获取路由

（2）arp_announce

从输出 ARP 请求时，从 IP 数据报中确定源 IP 地址的规则，见表 18-3。

表 18-3 arp_announce 的取值

arp_announce	描述
0（默认值）	使用配置在任意网络设备上的任意本地地址
1	尽量避免使用与目标不在同一子网的本地地址的网络设备。这个网络设备可达的目标主机要求在 ARP 请求中的源 IP 地址是配置在接收网络设备上。此时，会检查所有的子网，如果没有这样的子网，则根据第 2 种规则选择源地址
2	使用最合适的本地地址作为源地址。在这种规则下，会忽略 IP 数据报中的源地址，而尝试选择本地地址。查找所有输出网络设备且在同一子网的主 IP 地址，如果找不到合适的本地地址，则查找所有网络设备

（3）arp_ignore

接收 ARP 请求报文的过滤规则，见表 18-4。

表 18-4 arp_ ignore 的取值

arp_ignore	描述
0（默认值）	答复目标 IP 地址是配置在任意网络设备任意的本地地址
1	只答复目标 IP 地址是配置在输入网络设备上的本地地址
2	只答复目标 IP 地址是配置在输入网络设备上的本地地址，并且在这个接口上与发送方的 IP 地址在同一子网内
3	不回复本地地址类型 RT_SCOPE_HOST 的目标地址，只答复地址类型 RT_SCOPE_LINK 的目标地址
4-7	保留
8	不回复所有的本地地址

（4）arp_accept

处理非 ARP 请求而接收到的 ARP 应答，见表 18-5。

表 18-5 arp_accept 的取值

arp_accept	描述
0（默认值）	丢弃非 ARP 请求而接收到的 ARP 应答
1	接收非 ARP 请求而接收到的 ARP 应答

（5）proxy_arp

是否允许进行 ARP 代理。

18.3　注册 ARP 报文类型

ARP 报文像 IP 数据报一样，也是作为数据封装在以太网帧中发送的。ARP 报文由 arp_rcv()接收处理，ARP 模块初始化时需要在协议栈中注册 ARP 报文的类型，参见 18.4 节。

```
1234 static struct packet_type arp_packet_type = {
1235     .type =    __constant_htons(ETH_P_ARP),
1236     .func =    arp_rcv,
1237 };
```

18.4　ARP 初始化

ARP 模块的初始化是由 arp_init()完成的，该函数由 IPv4 协议栈初始化函数 inet_init()调用，首先初始化 ARP 协议的邻居表，然后在协议栈中注册 ARP 协议，最后建立 proc 对象，注册事件通知。

```
1241 void __init arp_init(void)
1242 {
1243     neigh_table_init(&arp_tbl);
1244
1245     dev_add_pack(&arp_packet_type);
1246     arp_proc_init();
1247 #ifdef CONFIG_SYSCTL
1248     neigh_sysctl_register(NULL, &arp_tbl.parms, NET_IPV4,
1249                 NET_IPV4_NEIGH, "ipv4", NULL, NULL);
1250 #endif
1251     register_netdevice_notifier(&arp_netdev_notifier);
1252 }
```

18.5　ARP 的邻居项函数指针表

在 ARP 中，根据不同的介质，提供了多种邻居项函数指针表的实例，例如通用的 arp_generic_ops，支持缓存硬件首部 arp_hh_ops，不支持 ARP 的 arp_direct_ops 以及支持业余无线电设备等的 arp_broken_ops。在初始化邻居项时，会根据访问邻居项的输出网络设备的特性使用合适的邻居项函数指针表的实例，参见 18.8 节。

除了两个输出函数指针 output 和 connected_output，这些邻居项函数指针表实例区别并不大，此外 arp_direct_ops 没有定义 solicit()和 error_report()。

```
139 static struct neigh_ops arp_generic_ops = {
140     .family =       AF_INET,
141     .solicit =       arp_solicit,
142     .error_report =       arp_error_report,
143     .output =       neigh_resolve_output,
144     .connected_output =    neigh_connected_output,
145     .hh_output =       dev_queue_xmit,
```

```
146     .queue_xmit =       dev_queue_xmit,
147 };

149 static struct neigh_ops arp_hh_ops = {
150     .family =       AF_INET,
151     .solicit =       arp_solicit,
152     .error_report =       arp_error_report,
153     .output =       neigh_resolve_output,
154     .connected_output =    neigh_resolve_output,
155     .hh_output =       dev_queue_xmit,
156     .queue_xmit =       dev_queue_xmit,
157 };
```

```
159 static struct neigh_ops arp_direct_ops = {
160     .family =       AF_INET,
161     .output =       dev_queue_xmit,
162     .connected_output =    dev_queue_xmit,
163     .hh_output =       dev_queue_xmit,
164     .queue_xmit =       dev_queue_xmit,
165 };

167 struct neigh_ops arp_broken_ops = {
168     .family =       AF_INET,
169     .solicit =       arp_solicit,
170     .error_report =       arp_error_report,
171     .output =       neigh_compat_output,
172     .connected_output =    neigh_compat_output,
173     .hh_output =       dev_queue_xmit,
174     .queue_xmit =       dev_queue_xmit,
175 };
```

18.6 ARP 表

ARP 的邻居表为 arp_tbl，其中与协议特性相关的字段有：地址族为 AF_INET；邻居项的大小为 neighbour 结构+4（IPv4 地址的长度）；哈希算法为 arp_hash()；ARP 初始化函数 arp_constructor()；延时处理代理 ARP 报文的例程为 parp_redo()，以及调整 ARP 表特性的参数。

```
177 struct neigh_table arp_tbl = {
178     .family =    AF_INET,
179     .entry_size =    sizeof(struct neighbour) + 4,
180     .key_len =    4,
181     .hash =       arp_hash,
182     .constructor =    arp_constructor,
183     .proxy_redo =    parp_redo,
184     .id =       "arp_cache",
185     .parms = {
186         .tbl =          &arp_tbl,
187         .base_reachable_time =    30 * HZ,
188         .retrans_time =    1 * HZ,
189         .gc_staletime =    60 * HZ,
190         .reachable_time =       30 * HZ,
191         .delay_probe_time =    5 * HZ,
192         .queue_len =       3,
```

```
193         .ucast_probes =    3,
194         .mcast_probes =    3,
195         .anycast_delay =    1 * HZ,
196         .proxy_delay =        (8 * HZ) / 10,
197         .proxy_qlen =       64,
198         .locktime =       1 * HZ,
199     },
200     .gc_interval =    30 * HZ,
201     .gc_thresh1 =    128,
202     .gc_thresh2 =    512,
203     .gc_thresh3 =    1024,
204 };
```

18.7　函数

18.7.1　arp_error_report()

arp_error_report()调用 dst_link_failure()向三层报告错误。用来初始化除 arp_direct_ops 之外的三个 neigh_ops 结构实例的 error_report 函数指针。当邻居项缓存中还存在有未发送的报文，而该邻居却无法访问时被调用。

```
325 static void arp_error_report(struct neighbour *neigh, struct sk_buff *skb)
326 {
327     dst_link_failure(skb);
328     kfree_skb(skb);
329 }
```

18.7.2　arp_solicit()

arp_solicit()用来发送 ARP 请求，用来初始化除 arp_direct_ops 之外的三个 neigh_ops 结构实例的 solicit 函数指针，在邻居项状态定时器处理函数中被调用。参数说明如下：

- neigh，ARP 请求的目的邻居项。
- skb，缓存在该邻居项中的待发送报文，用来获取该 skb 的源 IP 地址。

```
331 static void arp_solicit(struct neighbour *neigh, struct sk_buff *skb)
332 {
333     __be32 saddr = 0;
334     u8  *dst_ha = NULL;
335     struct net_device *dev = neigh->dev;
336     __be32 target = *(__be32*)neigh->primary_key;
337     int probes = atomic_read(&neigh->probes);
338     struct in_device *in_dev = in_dev_get(dev);
339
340     if (!in_dev)
341         return;
342
343     switch (IN_DEV_ARP_ANNOUNCE(in_dev)) {
344     default:
345     case 0:       /* By default announce any local IP */
346         if (skb && inet_addr_type(skb->nh.iph->saddr) == RTN_LOCAL)
```

```
347             saddr = skb->nh.iph->saddr;
348         break;
349     case 1:         /* Restrict announcements of saddr in same subnet */
350         if (!skb)
351             break;
352         saddr = skb->nh.iph->saddr;
353         if (inet_addr_type(saddr) == RTN_LOCAL) {
354             /* saddr should be known to target */
355             if (inet_addr_onlink(in_dev, target, saddr))
356                 break;
357         }
358         saddr = 0;
359         break;
360     case 2:         /* Avoid secondary IPs, get a primary/preferred one */
361         break;
362     }
363
364     if (in_dev)
365         in_dev_put(in_dev);
366     if (!saddr)
367         saddr = inet_select_addr(dev, target, RT_SCOPE_LINK);
368
369     if ((probes -= neigh->parms->ucast_probes) < 0) {
370         if (!(neigh->nud_state&NUD_VALID))
371             printk(KERN_DEBUG "trying to ucast probe in NUD_INVALID\n");
372         dst_ha = neigh->ha;
373         read_lock_bh(&neigh->lock);
374     } else if ((probes -= neigh->parms->app_probes) < 0) {
375 #ifdef CONFIG_ARPD
376         neigh_app_ns(neigh);
377 #endif
378         return;
379     }
380
381     arp_send(ARPOP_REQUEST, ETH_P_ARP, target, dev, saddr,
382          dst_ha, dev->dev_addr, NULL);
383     if (dst_ha)
384         read_unlock_bh(&neigh->lock);
385 }
```

340-341 检测邻居项网络设备的 IP 配置块是否有效。

343-362 根据 arp_announce 系统参数，选择源 IP 地址(规则 0 或 1)，参见 18.2 节中 arp_announce 系统参数。

366-367 根据 arp_announce 系统参数，选择源 IP 地址(规则 2)。

369-379 检测 ARP 请求报文重传次数是否达到上限。如果是，则停止发送。

381 将得到的硬件源、目标地址和 IP 源、目标地址等作为参数，调用 arp_send()创建一个 ARP 报文并将其输出。

18.7.3 arp_ignore()

arp_ignore()用来根据过滤规则对输入 ARP 报文中的源、目标 IP 地址进行确认，返回值非 0 表示需要过滤。函数中首先根据规则获取 sip 和 scope，然后将这二者作为参数调用 inet_confirm_addr()对源、目的 IP 地址进行确认。参数说明如下：

- in_dev，输入 ARP 请求报文网络设备的 IP 控制块。
- dev，输入 ARP 请求报文的网络设备。
- sip，发送方 IP 地址。
- tip，ARP 请求报文的目标 IP 地址。

```
387 static int arp_ignore(struct in_device *in_dev, struct net_device *dev,
388                         __be32 sip, __be32 tip)
389 {
390     int scope;
391
392     switch (IN_DEV_ARP_IGNORE(in_dev)) {
393     case 0:    /* Reply, the tip is already validated */
394         return 0;
395     case 1:    /* Reply only if tip is configured on the incoming interface */
396         sip = 0;
397         scope = RT_SCOPE_HOST;
398         break;
399     case 2:    /*
400          * Reply only if tip is configured on the incoming interface
401          * and is in same subnet as sip
402          */
403         scope = RT_SCOPE_HOST;
404         break;
405     case 3:    /* Do not reply for scope host addresses */
406         sip = 0;
407         scope = RT_SCOPE_LINK;
408         dev = NULL;
409         break;
410     case 4:    /* Reserved */
411     case 5:
412     case 6:
413     case 7:
414         return 0;
415     case 8:    /* Do not reply */
416         return 1;
417     default:
418         return 0;
419     }
420     return !inet_confirm_addr(dev, sip, tip, scope);
421 }
```

392 获取系统配置的过滤规则，根据规则做相应处理。

393 规则 0，应答所有的 ARP 请求报文时，直接返回 0，表示不过滤需应答该请求报文。

395-398 规则 1，应答目标 IP 地址是配置在输入接口上的本地地址的 ARP 请求报文，由于无需检测发送方 IP 地址，因此将 sip 设为 0，并且目的 IP 地址类型为 RT_SCOPE_HOST。

399-404 规则 2，应答目标 IP 地址是配置在输入接口上的本地地址，并且与源 IP 地址在同一个子网上的 ARP 请求报文，由于需检测源、目的 IP 地址，因此将目的 IP 地址类型为 RT_SCOPE_HOST 即可。

405-409 规则 3，不应答源地址类型为 RT_SCOPE_HOST 的 ARP 请求报文，因此不检测发送方 IP 地址，将目的 IP 地址类型设置为 RT_SCOPE_LINK，并且需遍历所有的网络设备。

410-414 规则 4～7 保留未使用，直接返回 0，应答。

415-416 规则 8，直接返回 1，不应答。

417-418 无效规则号，直接返回 0，应答。

420 调用 inet_confirm_addr()对源、目的 IP 地址进行确认。

18.7.4 arp_filter()

arp_filter()根据 ARP 请求报文中的发送方 IP 地址和目的 IP 地址，查找输出到 ARP 请求报文发送方的路由，过滤掉那些查找路由失败，或是查找到的路由输出设备与输入 ARP 请求报文的设备不同的 ARP 请求报文。

18.8 IPv4 中邻居项的初始化

arp_constructor()是 ARP 的邻居初始化函数，用来初始化新的 neighbour 结构实例，在邻居表创建函数 neigh_create()中被调用。

```
235 static int arp_constructor(struct neighbour *neigh)
236 {
237     __be32 addr = *(__be32*)neigh->primary_key;
238     struct net_device *dev = neigh->dev;
239     struct in_device *in_dev;
240     struct neigh_parms *parms;
241
242     neigh->type = inet_addr_type(addr);
243
244     rcu_read_lock();
245     in_dev = __in_dev_get_rcu(dev);
246     if (in_dev == NULL) {
247         rcu_read_unlock();
248         return -EINVAL;
249     }
250
251     parms = in_dev->arp_parms;
252     neigh_parms_put(neigh->parms);
253     neigh->parms = neigh_parms_clone(parms);
254     rcu_read_unlock();
255
256     if (dev->hard_header == NULL) {
257         neigh->nud_state = NUD_NOARP;
258         neigh->ops = &arp_direct_ops;
259         neigh->output = neigh->ops->queue_xmit;
260     } else {
261         /* Good devices (checked by reading texts, but only Ethernet is
262            tested)
263
264            ARPHRD_ETHER: (ethernet, apfddi)
265            ARPHRD_FDDI: (fddi)
266            ARPHRD_IEEE802: (tr)
267            ARPHRD_METRICOM: (strip)
268            ARPHRD_ARCNET:
269            etc. etc. etc.
270
271            ARPHRD_IPDDP will also work, if author repairs it.
```

```
272           I did not it, because this driver does not work even
273           in old paradigm.
274         */
275
276 #if 1
277         /* So... these "amateur" devices are hopeless.
278           The only thing, that I can say now:
279           It is very sad that we need to keep ugly obsolete
280           code to make them happy.
281
282           They should be moved to more reasonable state, now
283           they use rebuild_header INSTEAD OF hard_start_xmit!!!
284           Besides that, they are sort of out of date
285           (a lot of redundant clones/copies, useless in 2.1),
286           I wonder why people believe that they work.
287         */
288         switch (dev->type) {
289         default:
290            break;
291         case ARPHRD_ROSE:
292 #if defined(CONFIG_AX25) || defined(CONFIG_AX25_MODULE)
293         case ARPHRD_AX25:
294 #if defined(CONFIG_NETROM) || defined(CONFIG_NETROM_MODULE)
295         case ARPHRD_NETROM:
296 #endif
297            neigh->ops = &arp_broken_ops;
298            neigh->output = neigh->ops->output;
299            return 0;
300 #endif
301         ;}
302 #endif
303         if (neigh->type == RTN_MULTICAST) {
304            neigh->nud_state = NUD_NOARP;
305            arp_mc_map(addr, neigh->ha, dev, 1);
306         } else if (dev->flags&(IFF_NOARP|IFF_LOOPBACK)) {
307            neigh->nud_state = NUD_NOARP;
308            memcpy(neigh->ha, dev->dev_addr, dev->addr_len);
309         } else if (neigh->type == RTN_BROADCAST || dev->flags&IFF_POINTOPOINT) {
310            neigh->nud_state = NUD_NOARP;
311            memcpy(neigh->ha, dev->broadcast, dev->addr_len);
312         }
313         if (dev->hard_header_cache)
314            neigh->ops = &arp_hh_ops;
315         else
316            neigh->ops = &arp_generic_ops;
317         if (neigh->nud_state&NUD_VALID)
318            neigh->output = neigh->ops->connected_output;
319         else
320            neigh->output = neigh->ops->output;
321      }
322      return 0;
323 }
```

242 根据邻居地址获取邻居的类型。

244-254 检测邻居输出网络设备的 IP 配置块是否有效。如果有效，则从 IP 配置块中克

隆一份邻居配置块给邻居项；否则初始化失败，返回错误码。

256-259 如果无需支持 ARP，则设置该邻居项的状态为 NUD_NOARP，同时用 arp_direct_ops()作为邻居项的函数指针表，并初始化邻居项的输出接口 output。

288-302 需要 ARP 支持的情况，硬件接口类型为 ROSE、AX.25、NETROM 这三种情况，使用 arp_broken_ops()作为邻居项的函数指针表。

303-305 如果邻居项地址是组播类型，也无需 ARP 支持，调用 arp_mc_map()解析组播地址，把获取的组播地址存储到邻居项中。

306-308 如果邻居项的输出网络设备无需 ARP 支持或者是回环设备，则设置邻居项状态为 NUD NOARP，并从该网络设备中获取硬件地址存储到邻居项中。

309-312 如果邻居项地址是广播地址，或是点对点地址，则同样设置邻居项状态为 NUD_NOARP，并将广播地址作为硬件地址存储到邻居项中。

313-316 根据邻居项输出网络设备支持缓存硬件首部接口 hard_header_cache()与否，来设置邻居项的函数指针表为 arp_hh_ops 或是 arp_generic_ops。

317-321 根据邻居项的状态设置输出接口 output。

18.9 ARP 报文的创建

arp_create()用来创建一个完整的 ARP 类型二层报文，创建过程如下：

1）分配 SKB，并设置 SKB 的输出网络设备和协议。

2）通过网络设备的 hard header 接口填充 ARP 报文的硬件首部，以太网的该接口为 eth_header()。

3）根据网络设备的类型，设置 ARP 报文的硬件地址类型和协议类型。如果未提供源硬件地址则使用输出网络设备的硬件地址；如果未提供目标硬件地址则使用广播地址。

4）继续设置 ARP 报文首部以及其他字段。ARP 报文的格式见图 18-1。

arp_create()原型如下：

```
struct sk_buff *arp_create(int type, int ptype, __be32 dest_ip, struct
net_device *dev, __be32 src_ip, unsigned char *dest_hw, unsigned char *src_hw,
unsigned char *target_hw)
```

其中有多个涉及二层帧和 ARP 报文首部的参数，参数说明如下：

- type，ARP 协议的操作码，如 ARPOP_REPLY、ARPOP_REQUEST 等。
- ptype，三层协议类型，如以太网上 ARP 协议类型编码为 ETH_P_ARP（0x0806）。
- dest_ip，src_ip，输出 APR 报文的目的 IP 地址和发送方 IP 地址，填充到 ARP 报文中。
- dev，输出 ARP 报文的网络设备。
- dest_hw，target_hw，输出 ARP 报文的目的硬件地址，dest_hw 填充到二层帧首部，当为 NULL 时，会创建一个广播报文；target_hw 填充到 ARP 报文。
- src_hw，src_hw 位输出 ARP 报文的源硬件地址，填充到以太网帧首部和 ARP 报文。

18.10 ARP 的输出

arp_send()先创建一个 ARP 报文，如果创建成功就将其发送出。该函数的参数与 arp_create()

的参数相同，参见 arp_create()的参数说明。

```
678 void arp_send(int type, int ptype, __be32 dest_ip,
679         struct net_device *dev, __be32 src_ip,
680         unsigned char *dest_hw, unsigned char *src_hw,
681         unsigned char *target_hw)
682 {
683     struct sk_buff *skb;
684
685     /*
686      *    No arp on this interface.
687      */
688
689     if (dev->flags&IFF_NOARP)
690         return;
691
692     skb = arp_create(type, ptype, dest_ip, dev, src_ip,
693             dest_hw, src_hw, target_hw);
694     if (skb == NULL) {
695         return;
696     }
697
698     arp_xmit(skb);
699 }
```

689-690 如果网络设备无需 ARP 支持，则不需要发送 ARP 报文直接返回。

692-696 调用 arp_create()创建 ARP 报文。

698 如果创建成功，则再调用 arp_xmit()将其发送出去。arp_xmit()通过 NF_HOOK 封装了 dev_queue_xmit()，在 netfilter 处理之后调用 dev_queue_xmit()输出报文。

18.11 ARP 的输入

18.11.1 arp_rcv()

arp_rcv()用来从二层接收并处理一个 ARP 报文。参数说明如下：

- skb，ARP 报文的 SKB。
- dev，接收 ARP 报文的网络设备，可能与 orig_dev 不是同一个设备。
- pt，packet_type 结构实例，对 ARP 协议来说是 arp_packet_type，在其中定义了 ARP 协议接收函数为 arp_rcv()。该参数 arp_rcv()中并未使用。
- orig_dev，接收到 ARP 报文的原始网络设备，arp_rcv()中未使用。

```
930 static int arp_rcv(struct sk_buff *skb, struct net_device *dev,
931         struct packet_type *pt, struct net_device *orig_dev)
932 {
933     struct arphdr *arp;
934
935     /* ARP header, plus 2 device addresses, plus 2 IP addresses.  */
936     if (!pskb_may_pull(skb, (sizeof(struct arphdr) +
937                 (2 * dev->addr_len) +
938                 (2 * sizeof(u32)))))
```

```
939         goto freeskb;
940
941     arp = skb->nh.arph;
942     if (arp->ar_hln != dev->addr_len ||
943         dev->flags & IFF_NOARP ||
944         skb->pkt_type == PACKET_OTHERHOST ||
945         skb->pkt_type == PACKET_LOOPBACK ||
946         arp->ar_pln != 4)
947         goto freeskb;
948
949     if ((skb = skb_share_check(skb, GFP_ATOMIC)) == NULL)
950         goto out_of_mem;
951
952     memset(NEIGH_CB(skb), 0, sizeof(struct neighbour_cb));
953
954     return NF_HOOK(NF_ARP, NF_ARP_IN, skb, dev, NULL, arp_process);
955
956 freeskb:
957     kfree_skb(skb);
958 out_of_mem:
959     return 0;
960 }
```

936-939 检测 ARP 报文的完整性：其长度是否等于一个 ARP 头部长度，加两个硬件地址长度，再加两个 IP 地址长度。

941-947 检测报文和网络设备的标志。ARP 报文的硬件地址长度与网络设备的硬件地址长度是否匹配；网络设备是否支持 ARP 协议；ARP 报文是否是转发的包；ARP 报文是否来之回环接口等。

949-950 如果待处理的 ARP 报文是克隆的，则需复制一份新的报文。

954 通过 netfiter 处理之后，转到 arp_process()中处理。

18.11.2 arp_process()

ARP 的输入流程如图 18-2 所示，但其中不包括 ARP 代理。

```
705 static int arp_process(struct sk_buff *skb)
706 {
707     struct net_device *dev = skb->dev;
708     struct in_device *in_dev = in_dev_get(dev);
709     struct arphdr *arp;
710     unsigned char *arp_ptr;
711     struct rtable *rt;
712     unsigned char *sha, *tha;
713     __be32 sip, tip;
714     u16 dev_type = dev->type;
715     int addr_type;
716     struct neighbour *n;
717
718     /* arp_rcv below verifies the ARP header and verifies the device
719      * is ARP'able.
720      */
721
722     if (in_dev == NULL)
```

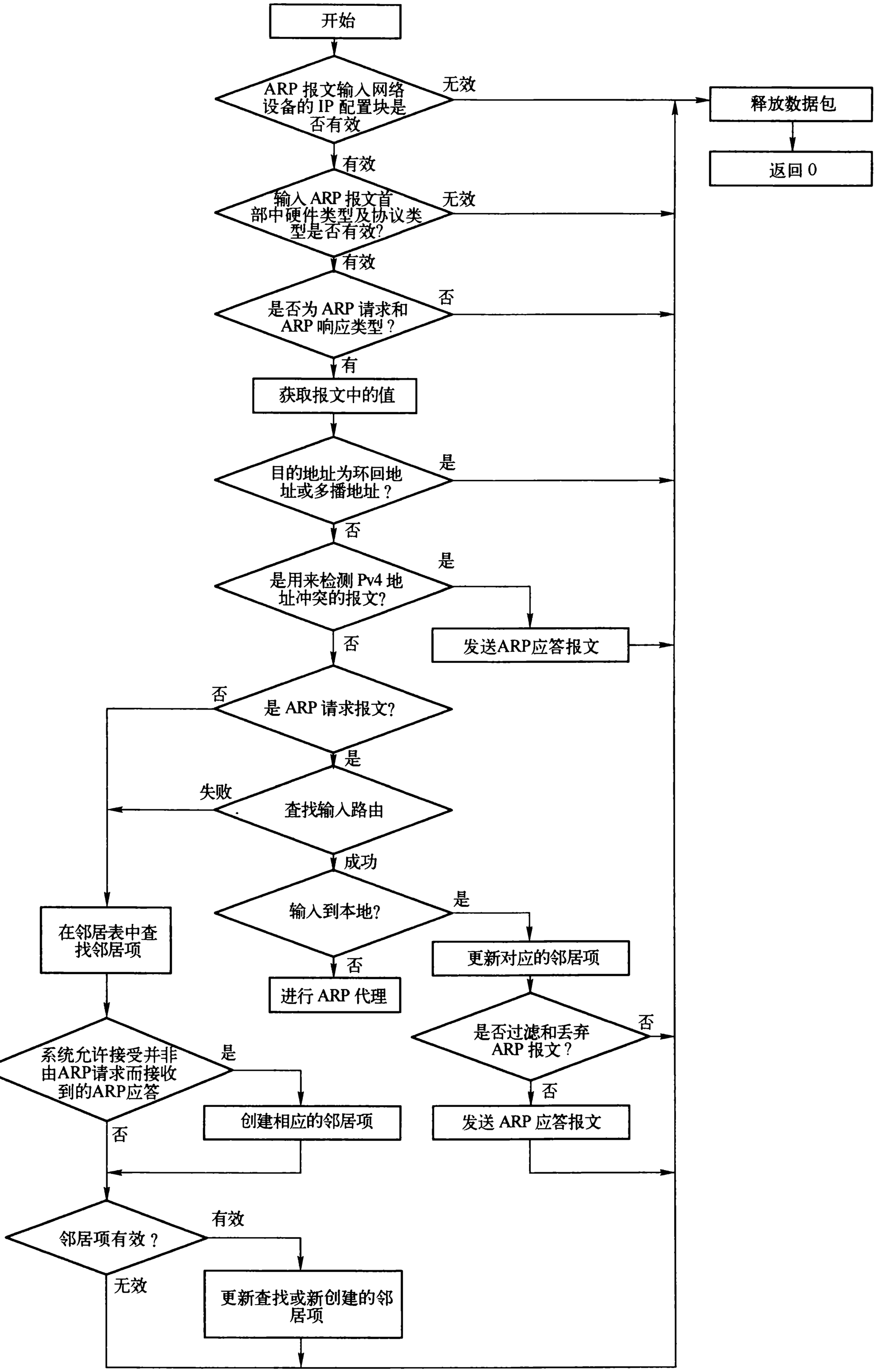

图 18-2　arp_process()流程图（不包括 ARP 代理）

```
723         goto out;
724
725     arp = skb->nh.arph;
726
727     switch (dev_type) {
728     default:
729         if (arp->ar_pro != htons(ETH_P_IP) ||
730             htons(dev_type) != arp->ar_hrd)
731             goto out;
732         break;
733 #ifdef CONFIG_NET_ETHERNET
734     case ARPHRD_ETHER:
735 #endif
736 #ifdef CONFIG_TR
737     case ARPHRD_IEEE802_TR:
738 #endif
739 #ifdef CONFIG_FDDI
740     case ARPHRD_FDDI:
741 #endif
742 #ifdef CONFIG_NET_FC
743     case ARPHRD_IEEE802:
744 #endif
745 #if defined(CONFIG_NET_ETHERNET) || defined(CONFIG_TR) || \
746     defined(CONFIG_FDDI)    || defined(CONFIG_NET_FC)
747         /*
748          * ETHERNET, Token Ring and Fibre Channel (which are IEEE 802
749          * devices, according to RFC 2625) devices will accept ARP
750          * hardware types of either 1 (Ethernet) or 6 (IEEE 802.2).
751          * This is the case also of FDDI, where the RFC 1390 says that
752          * FDDI devices should accept ARP hardware of (1) Ethernet,
753          * however, to be more robust, we'll accept both 1 (Ethernet)
754          * or 6 (IEEE 802.2)
755          */
756         if ((arp->ar_hrd != htons(ARPHRD_ETHER) &&
757              arp->ar_hrd != htons(ARPHRD_IEEE802)) ||
758             arp->ar_pro != htons(ETH_P_IP))
759             goto out;
760         break;
761 #endif
762 #if defined(CONFIG_AX25) || defined(CONFIG_AX25_MODULE)
763     case ARPHRD_AX25:
764         if (arp->ar_pro != htons(AX25_P_IP) ||
765             arp->ar_hrd != htons(ARPHRD_AX25))
766             goto out;
767         break;
768 #if defined(CONFIG_NETROM) || defined(CONFIG_NETROM_MODULE)
769     case ARPHRD_NETROM:
770         if (arp->ar_pro != htons(AX25_P_IP) ||
771             arp->ar_hrd != htons(ARPHRD_NETROM))
772             goto out;
773         break;
774 #endif
775 #endif
776     }
```

722-723 检测 ARP 报文输入网络设备的 IP 配置块是否有效。

727-776 根据网络设备类型，检测输入 ARP 报文首部中硬件类型及协议类型域的有效性。

```
778     /* Understand only these message types */
779
780     if (arp->ar_op != htons(ARPOP_REPLY) &&
781         arp->ar_op != htons(ARPOP_REQUEST))
782         goto out;
783
784 /*
785  *    Extract fields
786  */
787     arp_ptr= (unsigned char *)(arp+1);
788     sha    = arp_ptr;
789     arp_ptr += dev->addr_len;
790     memcpy(&sip, arp_ptr, 4);
791     arp_ptr += 4;
792     tha    = arp_ptr;
793     arp_ptr += dev->addr_len;
794     memcpy(&tip, arp_ptr, 4);
795 /*
796  *    Check for bad requests for 127.x.x.x and requests for multicast
797  *    addresses.  If this is one such, delete it.
798  */
799     if (LOOPBACK(tip) || MULTICAST(tip))
800         goto out;
801
802 /*
803  *    Special case: We must set Frame Relay source Q.922 address
804  */
805     if (dev_type == ARPHRD_DLCI)
806         sha = dev->broadcast;
```

780-782 目前 ARP 接收处理只支持 ARP 请求（ARPOP_REPLY）和 ARP 响应（ARPOP_REQUEST），其他类型的 ARP 报文均丢弃。

787-800 获取 ARP 报文中发送方硬件地址（sha）、发送方 IP 地址（sip）、目的硬件地址（tha）和目的 IP 地址（tip），丢弃目的 IP 地址为环回地址或多播地址的报文。

805-806 如果硬件类型为 Q.922，则发送方硬件地址，即 ARP 应答报文的目标硬件地址，设置为网络设备的广播地址。

```
808 /*
809  *  Process entry.  The idea here is we want to send a reply if it is a
810  *  request for us or if it is a request for someone else that we hold
811  *  a proxy for.  We want to add an entry to our cache if it is a reply
812  *  to us or if it is a request for our address.
813  *  (The assumption for this last is that if someone is requesting our
814  *  address, they are probably intending to talk to us, so it saves time
815  *  if we cache their address.  Their address is also probably not in
816  *  our cache, since ours is not in their cache.)
817  *
818  *  Putting this another way, we only care about replies if they are to
819  *  us, in which case we add them to the cache.  For requests, we care
820  *  about those for us and those for our proxies.  We reply to both,
821  *  and in the case of requests for us we add the requester to the arp
822  *  cache.
```

```
823  */
824
825     /* Special case: IPv4 duplicate address detection packet (RFC2131) */
826     if (sip == 0) {
827        if (arp->ar_op == htons(ARPOP_REQUEST) &&
828           inet_addr_type(tip) == RTN_LOCAL &&
829           !arp_ignore(in_dev,dev,sip,tip))
830         arp_send(ARPOP_REPLY,ETH_P_ARP,tip,dev,tip,sha,dev->dev_addr,dev->dev_
           addr);
831        goto out;
832     }
833
834     if (arp->ar_op == htons(ARPOP_REQUEST) &&
835        ip_route_input(skb, tip, sip, 0, dev) == 0) {
836
837        rt = (struct rtable*)skb->dst;
838        addr_type = rt->rt_type;
839
840        if (addr_type == RTN_LOCAL) {
841           n = neigh_event_ns(&arp_tbl, sha, &sip, dev);
842           if (n) {
843              int dont_send = 0;
844
845              if (!dont_send)
846                 dont_send |= arp_ignore(in_dev,dev,sip,tip);
847              if (!dont_send && IN_DEV_ARPFILTER(in_dev))
848                 dont_send |= arp_filter(sip,tip,dev);
849              if (!dont_send)
850                  arp_send(ARPOP_REPLY,ETH_P_ARP,sip,dev,tip,sha,dev->dev_addr,
                  sha);
851
852              neigh_release(n);
853           }
854           goto out;
855        } else if (IN_DEV_FORWARD(in_dev)) {
... ...
874        }
875     }
```

826-832 如果请求报文的源 IP 地址为 0，则该 ARP 报文是用来检测 IPv4 地址冲突的(RFC2131)，因此在确定请求报文的目标 IP 地址为本地 IP 地址后，以该 IP 地址为源地址及目标地址发送 ARP 应答报文。

834-875 如果是 ARP 请求报文，则调用 ip_route_input()找根目的 IP 地址 tip 对应的路由。

840-854 处理发送给本机的 ARP 请求报文，首先调用 neigh_event_ns()更新对应的邻居项，然后根据系统设置来决定是否过滤和丢弃 ARP 报文，最后如果没有被过滤或丢弃掉，则发送 ARP 应答报文。

855-874 对于不是发送给本机的 ARP 请求报文，根据系统参数确定是否进行 ARP 代理，参见 18.12 节。

```
879     n = __neigh_lookup(&arp_tbl, &sip, dev, 0);
880
881     if (ipv4_devconf.arp_accept) {
882        /* Unsolicited ARP is not accepted by default.
```

```
883             It is possible, that this option should be enabled for some
884             devices (strip is candidate)
885          */
886         if (n == NULL &&
887             arp->ar_op == htons(ARPOP_REPLY) &&
888             inet_addr_type(sip) == RTN_UNICAST)
889             n = __neigh_lookup(&arp_tbl, &sip, dev, -1);
890     }
891
892     if (n) {
893         int state = NUD_REACHABLE;
894         int override;
895
901         override = time_after(jiffies, n->updated + n->parms->locktime);
902
906         if (arp->ar_op != htons(ARPOP_REPLY) ||
907             skb->pkt_type != PACKET_HOST)
908             state = NUD_STALE;
909         neigh_update(n, sha, state, override ? NEIGH_UPDATE_F_OVERRIDE : 0);
910         neigh_release(n);
911     }
912
913 out:
914     if (in_dev)
915         in_dev_put(in_dev);
916     kfree_skb(skb);
917     return 0;
918 }
```

879 对于 ARP 的应答消息或未处理代理请求，则需要更新邻居表，因此先在邻居表中根据 sip 查找对应的邻居项。

881-890 对于那些并非由 ARP 请求而接收到的 ARP 应答，在系统允许接收的情况下，创建相应的邻居项。

892-911 更新查找或新创建的邻居项，首先确定邻居项的新状态，如果是发送给本机的 ARP 应答报文，则对应邻居项状态应转变为 NUD_REACHABLE；否则转到 NUD_STALE。然后调用 neigh_update()更新邻居项，如果其更新时间已超过 locktime，则应用覆盖的方式进行更新。

18.12　ARP 代理

代理 ARP（PROXY ARP），通常像路由器这样的设备才使用，用来代替处于另一个网段的主机回答本网段主机的 ARP 请求。

在路由器启用了 ARP 代理及报文转发后，每当收到的非本网段的 ARP 请求，该网络设备就会做出 ARP 应答。这样一来，即使内部主机没有设置网关，在访问外部地址时会发出查找外部 IP 地址对应 MAC 地址的 ARP 请求，路由器应答自己的 MAC 地址，从而是内部主机能通过网关访问外部。当然前提是目标主机和内部主机在同一网段内。

如图 18-3 所示，主机 PC1（192.168.20.66/24）需要向主机 .PC2（192.168.20.20/24）发送报文，因为主机 PC1 不清楚子网的存在且和目标主机 PC2 在同一主网络网段，所以主机 PC1

将发送 ARP 请求广播报文请求 192.168.20.20 的 MAC 地址。这时，路由器将识别出报文的目标地址属于另一个子网（注意，路由器的接口 IP 地址配置的是 28 位的掩码），因此向请求主机回复自己的硬件地址（0004.dd9e.cca0）。之后，PC1 将发往 PC2 的数据包都发往 MAC 地址 0004.dd9e.cca0（路由器的接口 E0/0），由路由器将数据包转发到目标主机 PC2。接下来路由器将为 PC2 做同样的代理发送数据包的工作。这种 ARP 代理使得子网化的网络拓扑对于主机来说是透明的。

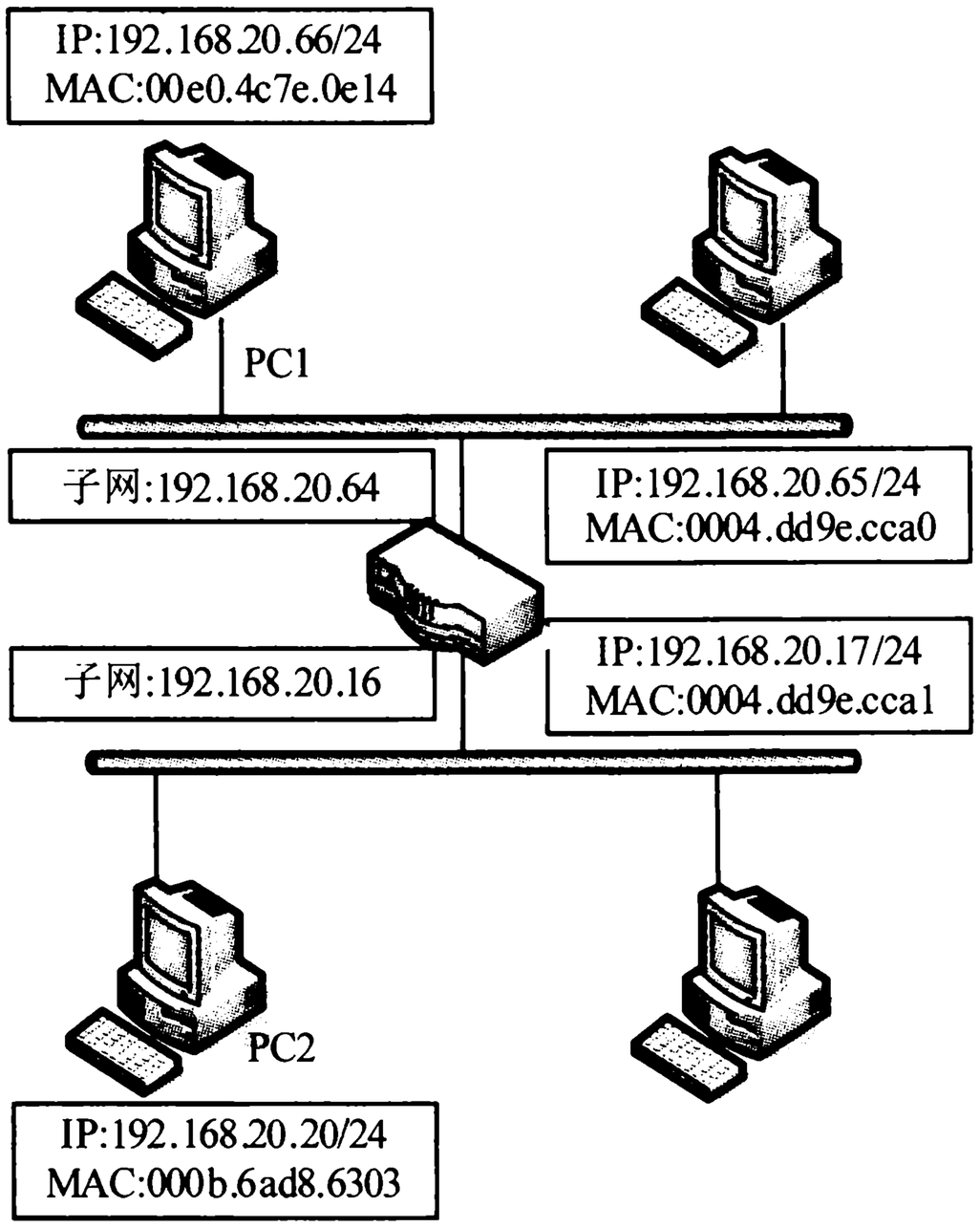

图 18-3　ARP 代理

18.12.1　arp_process()

如果 ARP 请求是从一个网络的主机发往另一个网络上的主机，那么连接这两个网络的路由器就可以回答该请求，这个过程称作 ARP 代理。这样可以欺骗发起 ARP 请求的发送端，使它误以为路由器就是目的主机，而事实上目的主机是在路由器的“另一边”。路由器的功能相当于目的主机的代理，把分组从其他主机转发给它。

以下代码是 arp_process()中对 ARP 代理的处理：

```
705 static int arp_process(struct sk_buff *skb)
706 {
... ...
834     if (arp->ar_op == htons(ARPOP_REQUEST) &&
... ...
855         } else if (IN_DEV_FORWARD(in_dev)) {
856             if ((rt->rt_flags&RTCF_DNAT) ||
857                 (addr_type == RTN_UNICAST  && rt->u.dst.dev != dev &&
858                  (arp_fwd_proxy(in_dev, rt) || pneigh_lookup(&arp_tbl, &tip, dev,
                     0)))){
859                 n = neigh_event_ns(&arp_tbl, sha, &sip, dev);
```

```
860             if (n)
861                 neigh_release(n);
862
863             if (NEIGH_CB(skb)->flags & LOCALLY_ENQUEUED ||
864                 skb->pkt_type == PACKET_HOST ||
865                 in_dev->arp_parms->proxy_delay == 0) {
866                         arp_send(ARPOP_REPLY,ETH_P_ARP,sip,dev,tip,sha,dev->dev
                            _addr,sha);
867             } else {
868                 pneigh_enqueue(&arp_tbl, in_dev->arp_parms, skb);
869                 in_dev_put(in_dev);
870                 return 0;
871                 }
872                 goto out;
873             }
874         }
875     }
... ...
918 }
```

855 对接收到目的不是本机的 ARP 请求报文，要代理首先需系统允许转发。

856 其次还需要至少满足以下两个条件之中的一个：

- 路由进行 DNAT。
- 目的地址是单播，ARP 的输入与输出不是同一个网络设备，同时或者允许网络设备 ARP 代理或者在 ARP 表中有相关代理项。

859-861 如果邻居表中有该邻居项，则将其删除释放。

863-866 如果该 ARP 报文来自代理报文缓冲队列，或者该 ARP 报文是发送给本机的，又或者代理 ARP 报文不需要延时处理，则调用 arp_send()发送应答报文。

867-871 对于需要延时处理的代理请求报文，将其缓存到 proxy_queue 队列中。

18.12.2　arp_fwd_proxy()

arp_fwd_proxy()用于判定是否允许进行 ARP 代理，参数说明如下：

- in_dev，ARP 请求报文输入的网络设备。
- rt，ARP 请求报文的目的 IP 对应的路由表项。

```
533 static inline int arp_fwd_proxy(struct in_device *in_dev, struct rtable *rt)
534 {
535     struct in_device *out_dev;
536     int imi, omi = -1;
537
538     if (!IN_DEV_PROXY_ARP(in_dev))
539         return 0;
540
541     if ((imi = IN_DEV_MEDIUM_ID(in_dev)) == 0)
542         return 1;
543     if (imi == -1)
544         return 0;
545
546     /* place to check for proxy_arp for routes */
547
548     if ((out_dev = in_dev_get(rt->u.dst.dev)) != NULL) {
```

```
549         omi = IN_DEV_MEDIUM_ID(out_dev);
550         in_dev_put(out_dev);
551     }
552     return (omi != imi && omi != -1);
553 }
```

538-539 检测系统和输入网络设备是否允许进行 ARP 代理。

541-544 检测输入网络设备 IP 配置块的 medium_id，根据其值做如下处理：

- 为 0 时表示不限制介质类型，可以进行 ARP 代理。
- 为-1 时表示未知介质类型，禁止 ARP 代理。
- 大于 0 时，还需检测代理 ARP 请求报文的输出网络设备 IP 配置块的 medium_id。

548-552 获取代理 ARP 请求报文输出网络设备 IP 配置块的 medium_id，只有当其值大于 0，且与输入网络设备 IP 配置块的 medium_id 不等时，才可以进行 ARP 代理。

18.12.3 parp_redo()

parp_redo()是缓存代理 ARP 报文的处理函数，当邻居表中缓存了代理的 ARP 报文之后，便会启动邻居表的 proxy_timer 定时器，然后在定时器到期时激活 proxy_redo 接口，该接口在 ARP 中即为 parp_redo()。

```
920 static void parp_redo(struct sk_buff *skb)
921 {
922     arp_process(skb);
923 }
```

18.13 ARP 的 ioctl

arp_ioctl()实现了 ARP 模块的 ioctl 调用，由 inet_ioctl()调用，包括三个命令：SIOCDARP、SIOCSARP 和 SIOCGARP。

无论是删除、添加（替换）还是获取，都需要用到一个 arpreq 结构类型的参数。

```
103 struct arpreq {
104     struct sockaddr    arp_pa;       /* protocol address       */
105     struct sockaddr    arp_ha;       /* hardware address       */
106     int            arp_flags;    /* flags                */
107     struct sockaddr      arp_netmask;    /* netmask (only for proxy arps) */
108     char           arp_dev[16];
109 };
```

struct sockaddr arp_pa

三层协议地址，IPv4 中为 32 位的 IPv4 地址。

struct sockaddr arp_ha

硬件地址，例如 48 位的以太网地址。

int arp_flags

添加和删除邻居项时使用到的一些标志，见表 18-6。

表 18-6　arp_flags 的取值

arp_flags	描述
ATF_COM	表示该邻居项的硬件地址是有效的，由内核根据 ATF_PERM 或邻居项的状态设置
ATF_PERM	标识邻居项的硬件地址静态配置，通常由应用层命令设定
ATF_PUBL	标识配置的是一个代理项
ATF_USETRAILERS	未使用
ATF_NETMASK	标识使用了子网掩码，arp_netmask 中的掩码有效，依赖于 ATF_PUBL
ATF_DONTPUB	不应答来自该代理项的请求，依赖于 ATF_PUBL，目前内核暂不支持该该功能

`struct sockaddr arp_netmask`

子网掩码，只用于配置代理项，配置普通邻居项时设置为 0xFFFFFFFF。

`char arp_dev[16]`

邻居项输出网络设备名。

ARP 中的一些 ioctl 命令如下：

- SIOCDARP，此命令是通过调用 arp_req_delete()删除 ARP 表中的一个指定的邻居项，调用者指定要删除表项的 IP 地址。
- SIOCSARP，此命令是通过调用 arp_req_set()把一个新的表项加入到 ARP 表，或者修改其中已经存在的一个表项。
- SIOCGARP，此命令是通过调用 arp_req_get()从 ARP 表中获取一个指定的表项。

18.14　外部事件

当一个设备的 IP 地址发生变化，ARP 模块通过注册到通知链中 arp_netdev_notifier 收到通知，调用 arp_netdev_event()处理该 NETDEV_CHANGEADDR 事件，删除释放与禁用网络设备相关的邻居项，同时刷新路由缓存。如果该邻居项引用计数大于 1 不能删除，会使用 neigh_blackhole()作为邻居项的 output，这样一旦调用到该邻居的输出接口会毫不留情地丢弃报文。

```
1200 static int arp_netdev_event(struct notifier_block *this, unsigned long
event, void *ptr)
1201 {
1202     struct net_device *dev = ptr;
1203
1204     switch (event) {
1205     case NETDEV_CHANGEADDR:
1206         neigh_changeaddr(&arp_tbl, dev);
1207         rt_cache_flush(0);
1208         break;
1209     default:
1210         break;
1211     }
1212
1213     return NOTIFY_DONE;
1214 }
```

ARP 模块的 arp_ifdown()，用来处理网络设备状态变为 NETDEV_DOWN 的情况。该函数调用了邻居子系统的 neigh_ifdown()。

```
1224 void arp_ifdown(struct net_device *dev)
1225 {
1226     neigh_ifdown(&arp_tbl, dev);
1227 }
```

18.15 路由表项与邻居项的绑定

在路由模块中，每当添加一条输出路由或是单播转发路由时，会尝试将该路由与该路由目的地址相对应的邻居项绑定。arp_bind_neighbour()实现了路由表项与邻居绑定的功能，在绑定过程中，如果对应的邻居项不存在，则会新建一个邻居项然后将路由项与之绑定。绑定之后，再输出报文时就能通过路由缓存找到输出函数，参见 20.5 节。

第19章　路　由　表

19.1　什么是路由表

路由子系统的核心是转发信息库（Forwarding Information Base，FIB），即路由表。路由表是用来存储这样一些信息的：一是用以确定输入数据报是应该上传给本机的上层协议还是继续转发的信息；二是如果需要转发，正确转发数据报所需要的信息；三是输出数据报应从哪个具体的网络设备输出的信息。图 19-1 中的灰色框体现了路由子系统在网络协议栈中的位置。

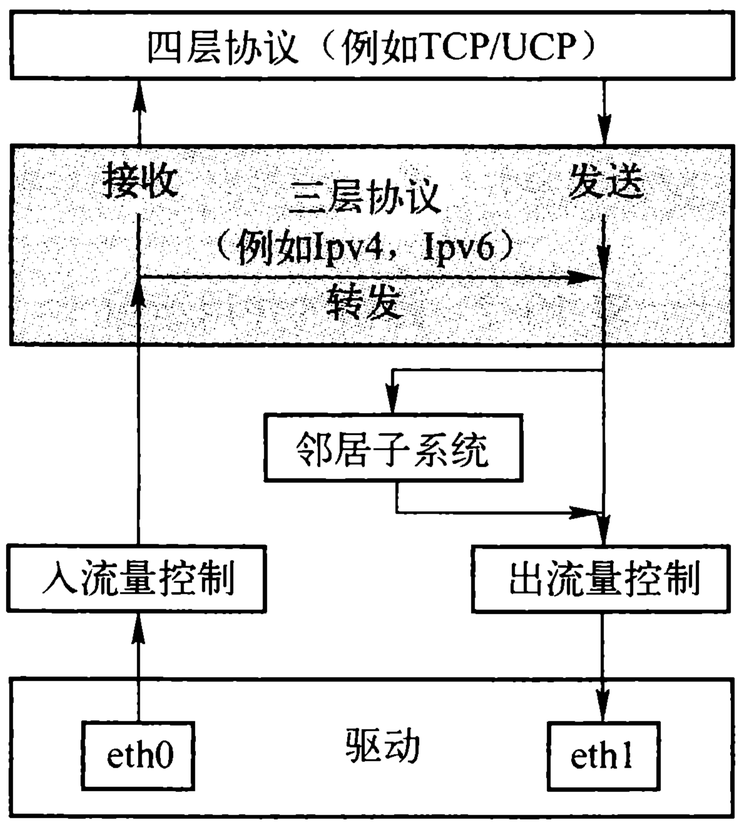

图 19-1　路由子系统与其他主要网络子系统之间的关系

路由表项的维护以及查找涉及以下文件：

- include/net/ip_fib.h，定义路由表等结构、宏和函数原型。
- net/ipv4/fib_lookup.h，定义路由查找的相关函数原型。
- net/ipv4/fib_hash.c，实现路由表的查找和维护。
- net/ipv4/fib_frontend.c，实现操作路由表的接口函数和通知。
- net/ipv4/route.c，实现路由缓存项的操作函数。

19.1.1　路由的要素

接下来介绍路由中的一些基本要素，清晰地理解这些基本要素是非常重要的。

（1）路由表

路由表是一个由路由表项组成的数据库，并为诸如 IPv4 等其他子系统提供了多种接口，其中最重要的接口就是路由查找。

（2）metrics

metrics 是在一条路由上配置的相关度量值。注意：此 metric 不同于路由协议使用的 metric，后者用于判断路由的好坏，如端对端延迟、跳跃数、配置权值或 cost 等。当用

IPROUTE2 来配置一条路由时，可以提供更多的 metric 参数，其中包括路径最大传输单元（PMTU），还有一些 TCP 使用的参数——窗口、往返时间、往返时间方差、慢启动门限值、拥塞窗口、最大段长度和 Reordering 等，在后面会看到如何使用这些度量值。

（3）作用范围

IP 地址和路由都有作用范围，用于说明它们在哪些情况下是有意义并可以被使用的。IP 地址的作用范围表示该 IP 地址距离本地主机有多远，而路由的作用范围表示到目的网络的距离。表 19-1 和表 19-2 分别是对 IP 地址和路由作用范围值的描述。

表 19-1 IP 地址的作用范围

IP 地址的作用范围	描述
Host	当一个地址只用于主机自身内部通信时，作用范围为 Host，该地址在主机以外不可知并且不能被使用。例如回环地址 127.0.0.1
Link	当一个地址只在一个局域网（即每一台计算机通过链路层互联的一个网络）内有意义且只在局域网内使用时，该地址的作用范围为 Link。例如子网的广播地址。子网内一台主机发送到子网广播地址的数据报被送给同一子网内的其他主机
Universe	当一个地址可以在任何地方使用时作用范围为 universe，这是大多数地址的默认 scope。注意：scope 不能反映不可路由（私有）地址与可路由（公开）地址之间的区别

表 19-2 路由的作用范围

路由的作用范围	描述
Host	当一条路由使目的地址为本地主机时，作用范围为 host
Link	当一条路由使目的地址为本地网络时，作用范围为 link
Universe	当一条路由使目的地址超过一跳时，作用范围为 universe

地址作用范围和路由作用范围被路由代码和内核其他模块广泛使用。

由于接口上配置 IP 地址是属于主机而不属于接口，因此在本地发送一个数据报时，内核会根据目的地址，在某个设备上选择一个有特定作用范围的源 IP 地址，参见 6.2.3 节的 inet_select_addr()。

路由代码中用作用范围对配置进行简单而又强有力的合理性检查。假定需要向与本地主机配置的任何子网都不直达的远端主机发送一个数据报，那么路由查找返回的将是网关地址，因此要使数据报到达该远端主机，只需将其发送给路由器，路由器会负责转发该数据报。为了避免出现环路，路由器必须比本地主机更靠近目的地，换句话说，到达目的主机的路由作用范围必须比到达路由器的路由作用范围更大。

以图 19-2 为例，从主机 A 到达主机 B，主机 A 进行路由查找，由图中所示的 A 路由表可知，通过 A 的 eth0 接口有两条路由：第一条可以直达网关地址 10.0.1.1，其作用范围为 RT_SCOPE_UNIVERSE；第二条的作用范围为 RT_SCOPE_LINK，比前一条路由的作用范围窄，因而返回经过 10.0.1.1 的默认路由，以便发送数据报给作用范围更广的地址 B。

（4）默认网关

默认网关通常是指 0.0.0.0/0 路由，当到一个目的地址不存在明确的路由项时使用该路由。与因特网相连的主机通常配置有到本地网络的一条路由（根据 NIC 的配置间接得出）和访问因特网的一条默认路由。

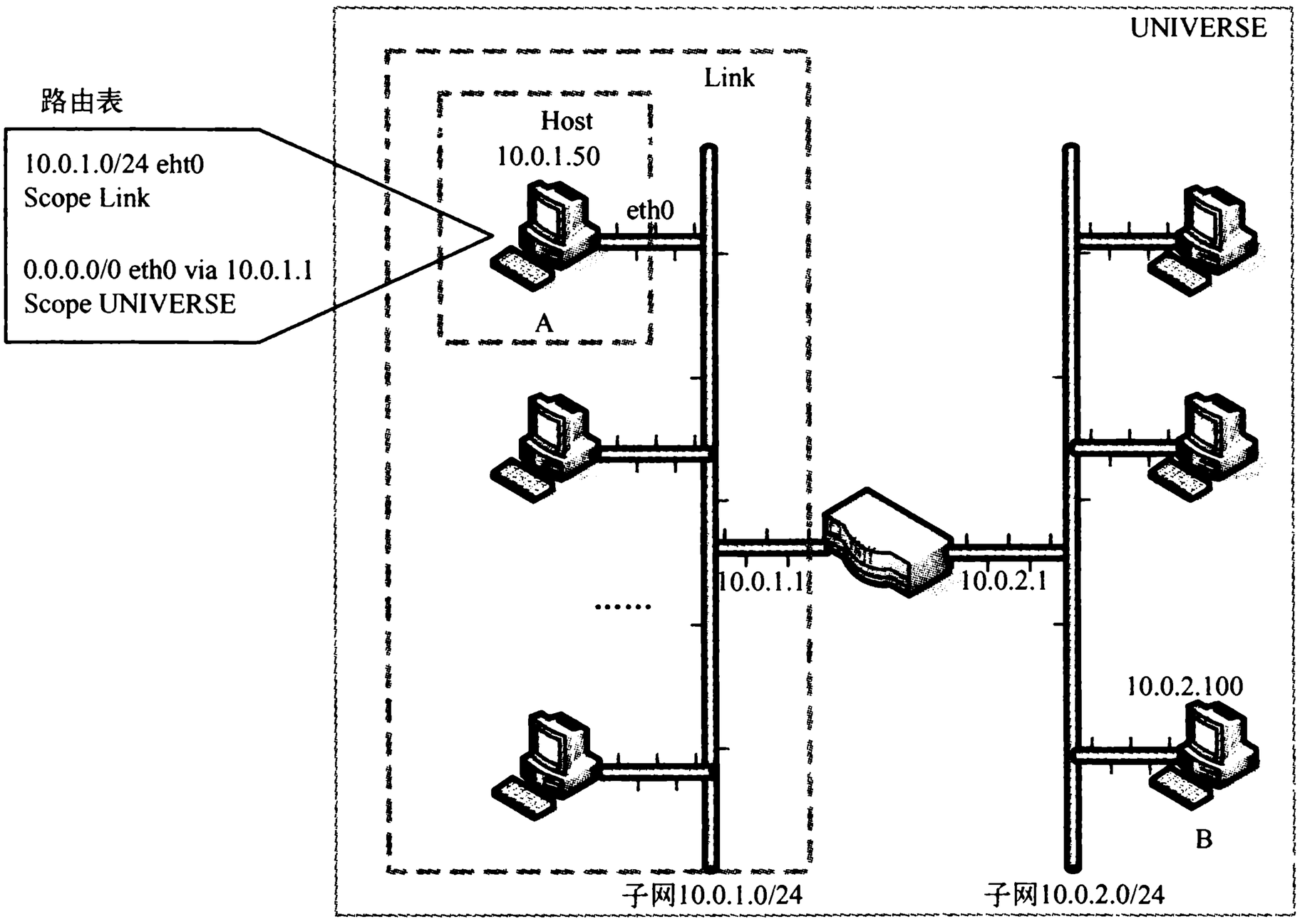

图 19-2 网络拓扑

19.1.2 特殊路由

当主机收到一个数据报后，路由子系统需要决定将它上传给本地上层协议还是继续转发出去。因此在路由子系统中，有两张特殊的路由表：

- 一张表用于本地地址，存储了所有的本地地址，如果在该表中能查找到匹配表项，则表明数据报是发送给本机的。
- 一张表用于所有其他的路由，路由表项由用户手工静态配置或由路由协议动态配置。

在路由查找时先扫描本地路由表，只有当查找该表未果的情况下，才会去查找另一路由表，以确定是否可以转发。

19.1.3 路由缓存

一个路由表中的路由项数量，在一般的主机中可能只是几条而已，而在路由器中，这个数目可以达到数十万条。因此很显然，在这种情况下维护一张更小的表来缓存路由查找结果是非常有必要的。

路由缓存分为两部分：一部分是与协议（如 IPv4 等三层协议）相关的缓存，这是缓存框架部分，每个元素被定义为一个由具体协议字段组成的集合；另一部分是与协议无关的缓存，通常被称为 DST，嵌套在缓存框架中，只存储与协议无关的信息，参见第 20 章。

当启用策略路由时，系统中可能会有多张独立的路由表，尽管如此，也仍只使用一个路由缓存，各路由表在缓存中占用的缓存表项数不均等。

路由表和路由缓存除了容量和结构不同之外，对象粒度也不同。路由表使用连续地址的集合，即子网，而缓存项与单个 IP 地址相关联。因此，路由表和路由缓存使用的查找算法也不同。

19.2 系统参数

系统参数如下：

（1）gc_timeout

gc_timeout 用于设置一个路由表项的生存期时长。

（2）gc_interval

gc_interval 用于标识路由表垃圾回收的时间间隔。

（3）error_burst

（4）error_cost

在输入过程中，由于目的不可达而选路失败时，以上参数用于对发送 ICMP_DEST_UNREACH 消息进行限速，参见 19.10.1 节。

error_burst 用来控制何时丢弃目的不可达和错误信息。默认的配置设置为每秒钟 5 个消息。

error_cost 作用见 error_burst 描述，此值设置得越高，允许通过的错误消息越少。

（5）redirect_load

（6）redirect_number

（7）redirect_silence

以上几个参数用于实现对 ICMP_REDIRECT 消息的限速。

限速所用的算法是一个很简单的指数回退算法（exponential backoff algorithm）。如果目的地持续忽略 ICMP 重定向消息，内核就持续发送 ICMP 重定向消息给它，直到发送数目到达 ip_rt_redirect_number，每发送一个消息就翻倍间隔时间。当发送的 ICMP 重定向数目达到 ip_rt_redirect_number 时，内核停止发送，直到 ip_rt_redirect_silence 秒过后还没有输入数据报并能够触发内核生成 ICMP 重定向消息。一旦 ip_rt_redirect_silence 秒过后有输入数据报触发内核生成 ICMP 重定向消息，内核即重新开始发送 ICMP 重定向消息。

ip_rt_redirect_load 为指数回退算法的初始延迟时间，参见 19.11 节。

19.3 路由表组成结构

2.6.20 版本的内核支持两种查找路由表项的算法——FIB_HASH 和 FIB_TRIE，前者是系统默认算法，而后者在超大路由表的情况下可以提高查找效率，当然复杂度和内存消耗也大为增加，具体启用哪种算法在编译期确定。现在以 FIB_HASH 算法为例进行讲解。

路由表是由 fib_table 结构来描述的，所有的 fib_table 结构链接在全局散列表 fib_table_hash 中，见图 19-3。在路由表初始化函数 fib_hash_init()中会看到，一个 fib_table 结构是和一个 fn_hash 结构一起分配的，fib_table 结构主要包含了路由表 ID 和一些管理路由表的函数，此外值得注意的一个字段是 tb_data，又是一个零长数组，实际上该字段的地址就等同于后接的 fn_hash 结构地址，而 fn_hash 结构包含一个由 33 个 fn_zone 结构指针组成的向量，用来把路由表项按目标地址掩码的长度分成 33 个区，所有非空的 fn_zone 结构又通过其 fz_next 字段和 fn_hash 结构的 fn_zone_list 字段链接成一个循环单链表，fn_zone_list 指向掩码最短的那个 fn_zone 结构，而掩码地址最长的 fn_zone 结构 fz_next 字段指向 fn_zone_list，见

图 19-4。

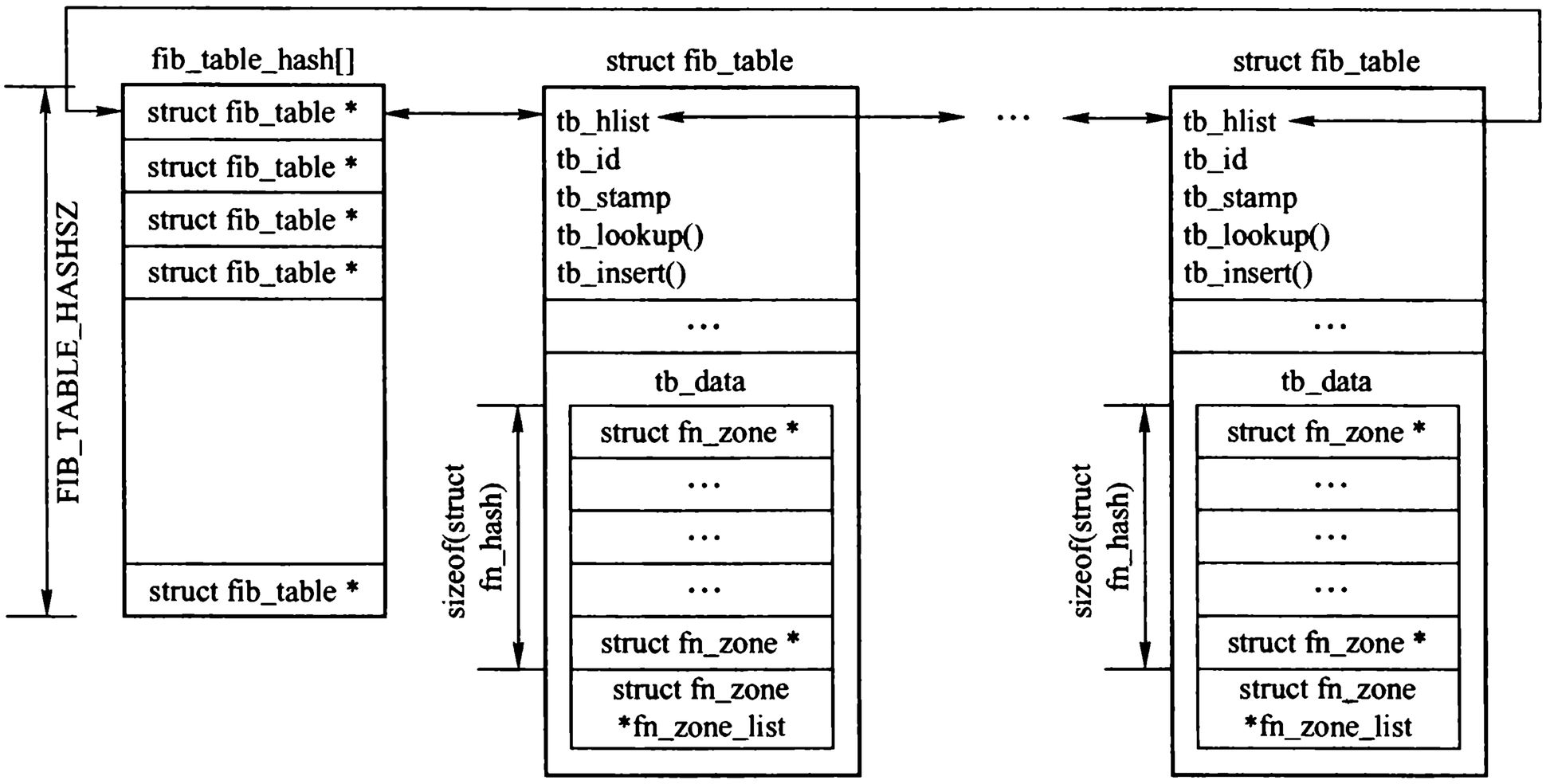

图 19-3　fib_table_hash 散列表

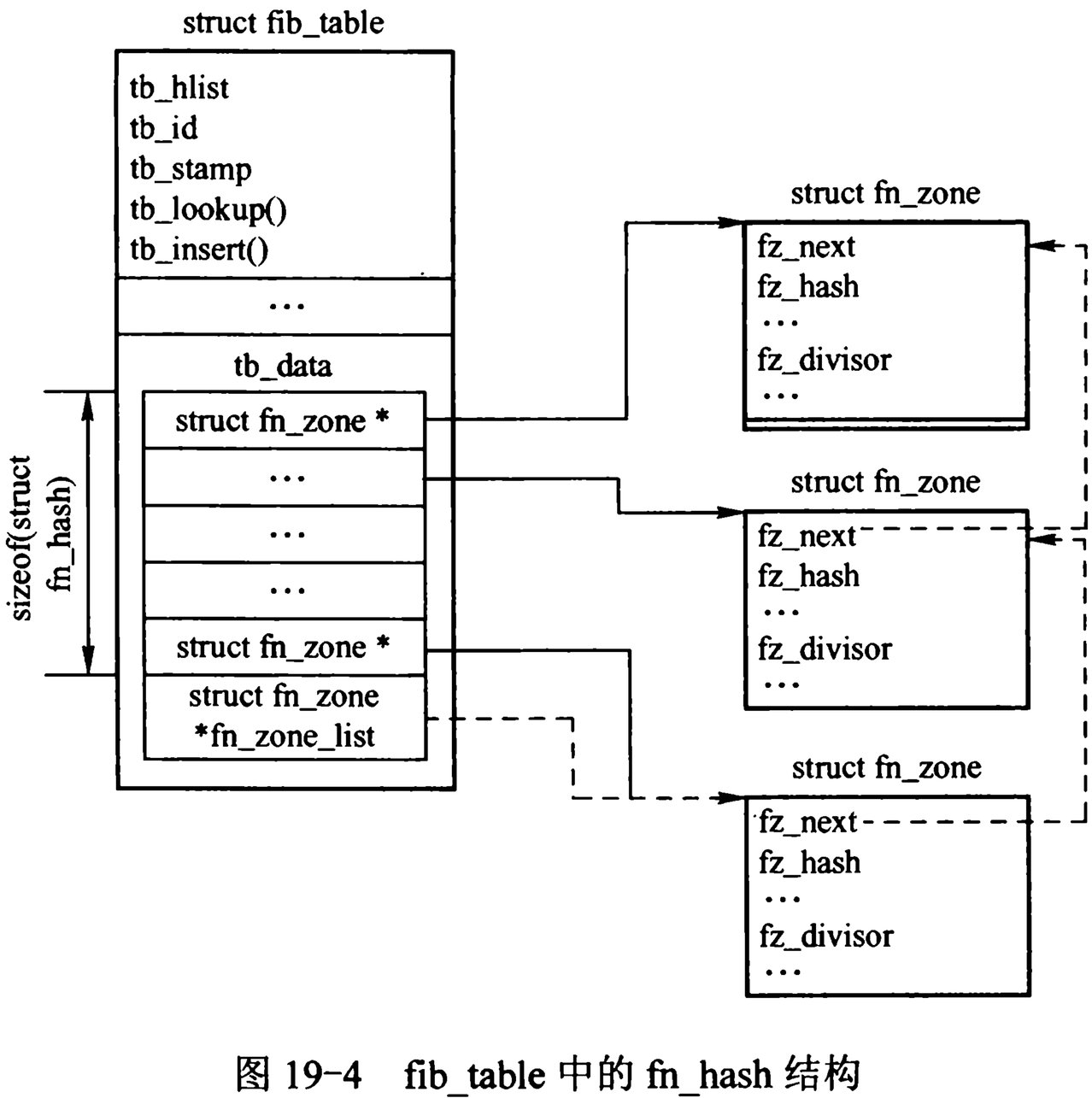

图 19-4　fib_table 中的 fn_hash 结构

每个 fn_zone 结构代表了路由表中所有同一掩码长度表项的集合，为了进一步划分这个集合，fn_zone 的 fz_hash 字段指向一个长度为 fz_divisor 的 HASH 表，代表不同子网的结构 fib_node 被根据其路由键值 fn_key，HASH 到该表中，冲突的 fib_node 结构由其 fn_hash 字段链接成双向链表。

具体的路由表项是由 fib_alias 和 fib_info 这两个数据结构构成的，其中 fib_info 中是一些可能被多个 fib_alias 共享的信息。

路由表项是由多个不同的数据结构来描述的，每个数据结构描述不同片段的路由信息。因

为只通过部分字段可以区分多条路由表项，因而一条路由信息被分散到多个数据结构内。这样，不是维护一个庞大而臃肿的结构，而是将路由表项分散为多个片段，路由子系统更容易在相似的路由表项之间共享公共片段信息，可以独立出不同的功能，并在这些功能之间定义更加清晰的接口。

每个网段对应一个 fib_node 实例，用变量 fn_key 来表示，它的值为网段。例如对子网 10.1.1.0/24 而言，fn_key 为 10.1.1。注意：fib_node 结构（即该结构中的 fn_key 变量）与一个网段相关，而不是与一条路由表项相关；一定要记住该细节。这个细节的重要性是由于多条路由表项可能都有相同的网段。

有相同网段的路由表项（即 fn_key 相同）共享同一个 fib_node。每一条路由表项有各自的 fib_alias 结构。例如：可能有一些路由表项，它们的网段相同而只是 TOS 值不同：每一个 fib_alias 实例因而有不同的 TOS 值。每个 fib_alias 实例与一个存储真正路由信息（即如何到达目的地）的 fib_info 结构相关联。

对于一个 fib_node 实例，相关的 fib_alias 实例链表按照 IP 的 TOS（即 fa_tos 字段）的递增顺序排列。fa_tos 值相同的 fib_alias 实例按照相关的 fib_info 中 fib_protocol 字段的递增顺序排列。

本章前面提到路由子系统划分为多个数据结构，目的是优化它们的使用并使逻辑更加清晰。所以，fib_alias 与 fib_info 之间的关联不是一对一，多个 fib_alias 结构可能共享一个 fib_info 结构。当多条不同的路由表项碰巧与一个已存在的 fib_info 结构中的参数值相同时，它们就指向这同一个 fib_info 实例。通过 fib_info 结构中的一个引用计数来记住共享数目。

例如，有五条到不同目的网络的路由表项碰巧使用相同的下一跳网关，那么对所有这五个表项而言下一跳信息都是相同的，因而就可以共享同样的下一跳信息。这种情况下就有五个 fib_node 结构和五个 fib_alias 结构，但只有一个 fib_info 结构。

在图 19-5 中给出的一个配置例子中，展示出本章描述各种散列表的不同数据结构之间的一些关系。在这个图中：

- 存在四条路由表项（即四个 fib_alias 实例）。
- 这四条路由表项是到三个不同的网段（即三个 fib_node 实例），这是由于有两个 fib_alias 实例共享一个 fib_node 实例。
- 四条路由表项中有两条共享同一个下一跳路由器。因而，这两个 fib_alias 结构中的 fa_info 字段指向同一个 fib_info 结构，见图的右下角。

19.3.1 fib_table 结构

对每个路由表实例创建一个 fib_table 结构，这个结构主要由一个路由表标识和管理该路由表的一组函数指针组成。

```
156 struct fib_table {
157     struct hlist_node tb_hlist;
158     u32         tb_id;
159     unsigned    tb_stamp;
160     int         (*tb_lookup)(struct fib_table *tb, const struct flowi *flp, struct
        fib_result *res);
161     int         (*tb_insert)(struct fib_table *, struct fib_config *);
162     int         (*tb_delete)(struct fib_table *, struct fib_config *);
163     int         (*tb_dump)(struct fib_table *table, struct sk_buff *skb,
```

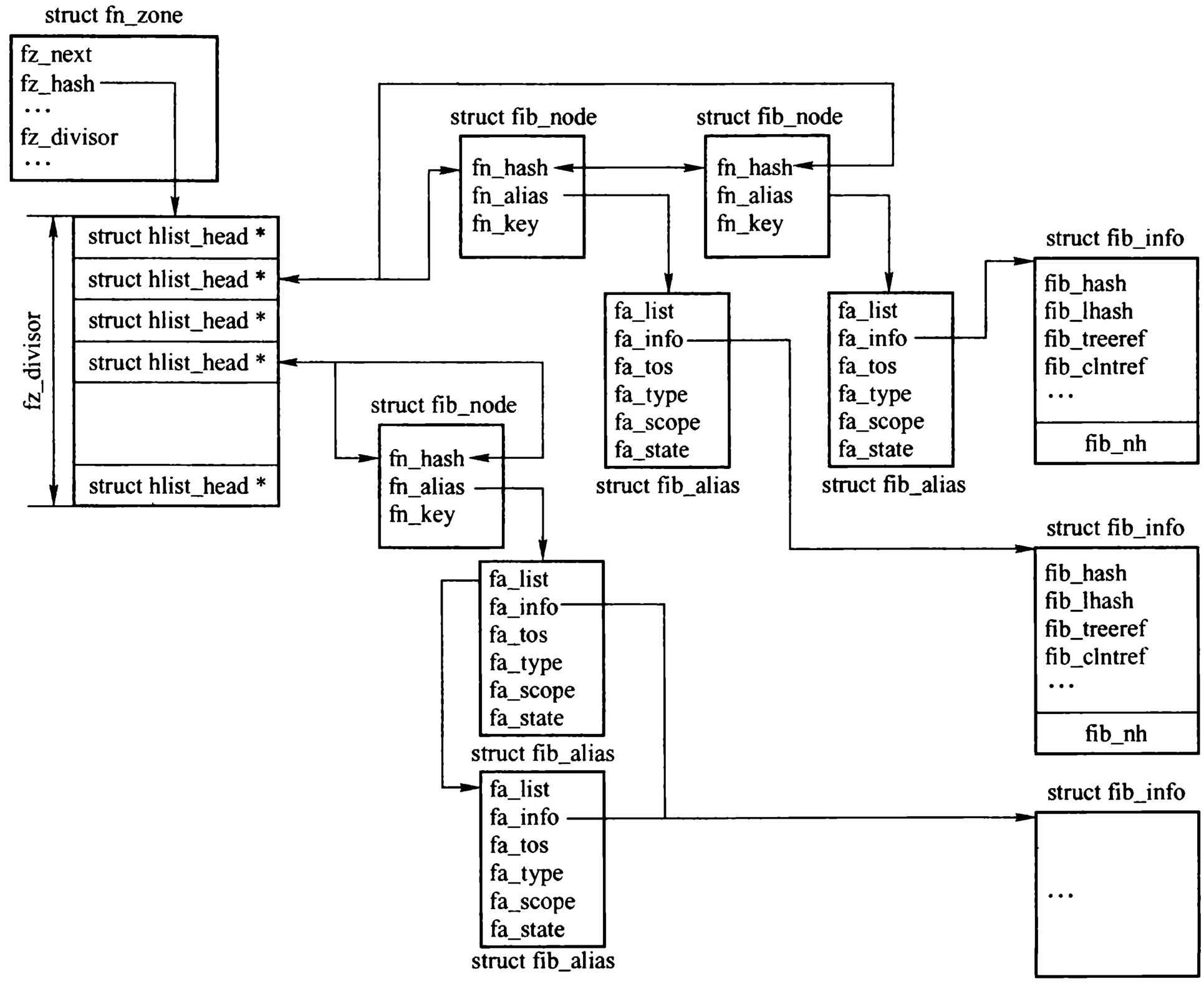

图 19-5 fn_zone 结构

```
164                     struct netlink_callback *cb);
165     int         (*tb_flush)(struct fib_table *table);
166     void         (*tb_select_default)(struct fib_table *table,
167                         const struct flowi *flp, struct fib_result *res);
168
169     unsigned char    tb_data[0];
170 };
```

157 struct hlist_node tb_hlist

用来将各个路由表链接成一个双向链表。

158 u32 tb_id

路由表标识。在支持策略路由的情况下，系统中最多可以有 256 个路由表，枚举类型 rt_class_t 中定义了保留的路由表 ID，如 RT_TABLE_MAIN、RT_TABLE_LOCAL，除此之外从 1 到（RT_TABLE_DEFAULT-1）都是可以由用户定义的。

159 unsigned tb_stamp

未使用。

160 int (*tb_lookup)(struct fib_table *tb, const struct flowi *flp, struct fib_result *res)

用于在当前路由表搜索符合条件的路由表项。在 FIB_HASH 算法中为 fn_hash_lookup()，此接口被 fib_lookup 调用。

161 int (*tb_insert)(struct fib_table *, struct fib_config *)

用于在当前路由表中插入给定的路由表项。在 FIB_HASH 算法中为 fn_hash_insert()，此接口被 inet_rtm_newroute 和 ip_rt_ioctl 调用。通常在处理 ip route add 命令和 route add 命令时被激活。该接口也被 fib_magic()调用，参见 19.9.6 节。

```
162 int (*tb_delete)(struct fib_table *, struct fib_config *)
```

用于在当前路由表中删除符合条件的路由表项。在 FIB_HASH 算法中为 fn_hash_delete()，此接口被 inet_rtm_delroute 和 ip_rt_ioctl 调用。通常在处理 ip route del 命令和 route add 命令时被激活。该接口也被 fib_magic()调用，参见 19.9.6 节。

```
163 int (*tb_dump)(struct fib_table *table, struct sk_buff *skb,
struct netlink_callback *cb)
```

dump 出路由表的内容。在 FIB_HASH 算法中为 fn_hash_dump()，此接口被 inet_rtm_getroute 调用。通常在执行 ip route get 命令时被激活。

```
165 int (*tb_flush)(struct fib_table *table)
```

删除设置有 RTNH_F_DEAD 标志的 fib_info 结构实例，在 FIB_HASH 算法中为 fn_hash_flush()。

```
166 void (*tb_select_default)(struct fib_table *table, const struct
flowi *flp, struct fib_result *res)
```

选择一条默认路由，在 FIB_HASH 算法中为 fn_hash_select_default()。

```
169 unsigned char tb_data[0]
```

路由表项的散列表起始地址。在 FIB_HASH 算法中指向 fn_hash 结构，而在 FIB_HASH 结构则指向 trie 结构，在后面的讨论中也以 FIB HASH 算法为主。

19.3.2 fn_zone 结构

一个 zone 是一组有着相同目的地址掩码长度的路由表项的散列表。

```
57 struct fn_zone {
58     struct fn_zone       *fz_next;    /* Next not empty zone    */
59     struct hlist_head    *fz_hash;    /* Hash table pointer     */
60     int          fz_nent;    /* Number of entries    */
61
62     int          fz_divisor;    /* Hash divisor         */
63     u32          fz_hashmask;    /* (fz_divisor - 1)    */
64 #define FZ_HASHMASK(fz)       ((fz)->fz_hashmask)
65
66     int          fz_order;    /* Zone order         */
67     __be32           fz_mask;
68 #define FZ_MASK(fz)       ((fz)->fz_mask)
69 };
```

```
58 struct fn_zone *fz_next
```

将活动（路由表项不为空）的 zone 链接在一起的指针，该链表的头部存储在 fn_hash 数据结构的 fn_zone_list 字段中。

```
59 struct hlist_head *fz_hash
```

指向存储该 zone 中路由项的散列表。

```
60 int fz_nent
```

在该 zone 的散列表中 fib_node 实例的数目，用于检查是否需要改变该散列表的容量。

62 int fz_divisor

表示散列表 fz_hash 的容量，及散列表桶的数目。

63 u32 fz_hashmask

其值为(fz_divisor-1)，用来计算散列表的关键值。

66 int fz_order

在网络掩码 fz_mask 的长度，在代码中有些地方也用 prefixlen 来表示。例如，网络掩码 255.255.255.0 所对应的 fz_order 为 24。

67 be32 fz_mask

用 fz_order 构造得到的网络掩码。例如设 fz_order 值取 3，则生成的 fz_mask 其十进制表示为 224.0.0.0。

19.3.3 fib_node 结构

fib_node 实例代表每一个唯一的目的网络的路由表项，即同一个子网中所有路由表项所共享的信息。目的网络相同但其他配置参数不同的路由表项共享同一个 fib_node 实例，因此一个 fib_node 实例上存在着一个或多个路由表项。

```
51 struct fib_node {
52     struct hlist_node    fn_hash;
53     struct list_head    fn_alias;
54     __be32              fn_key;
55 };
```

52 struct hlist_node fn_hash

用于将散列表中同一个桶内所有的 fib_node 实例链接成一个双向链表。

53 struct list_head fn_alias

fn_alias 指向一个或多个 fib_alias 结构实例构成的链表。

54 be32 fn_key

由 IP 地址和路由项的 netmask 与操作后得到，被用作查找路由表时的搜索条件。

19.3.4 fib_alias 结构

fib_alias 实例代表一条路由表项，目的地址相同但其他配置参数不同的表项共享 fib_node 实例。

```
8  struct fib_alias {
9      struct list_head    fa_list;
10     struct rcu_head rcu;
11     struct fib_info       *fa_info;
12     u8              fa_tos;
13     u8              fa_type;
14     u8              fa_scope;
15     u8              fa_state;
16 };
```

9 struct list_head fa_list

将与共享同一个 fib_node 实例的所有 fib_alias 实例链接在一起。

```
11 struct fib_info *fa_info
```

指针指向一个 fib_info 实例，该实例存储着如何处理与该路由相匹配数据报的信息。

```
12 u8 fa_tos
```

路由的服务类型比特位字段。当该值为零时表示还没有配置 TOS，所以在路由查找时任何值都可以匹配。fa_tos 用户对每一条路由表项配置的 TOS，区别于 fib_rule4 结构中的 tos。

```
13 u8 fa_type
```

路由表项的类型，它间接定义了当路由查找匹配时应采取的动作。该字段可能的取值如表 19-3 所示。

表 19-3 路由类型

路由类型	描述
RTN_UNSPEC	定义一个未初始化的值。例如，当从路由表中删除一个表项时使用该值，这是因为删除操作不需要指定路由表项的类型
RTN_LOCAL	目的地址被配置为一个本地接口的地址
RTN_UNICAST	该路由是一条到单播地址的直连或非直连（通过一个网关）路由。当通过 ip route 命令添加路由但没有指定其他路由类型时，路由类型默认被设置为 RTN_UNICAST
RTN_MULTICAST	目的地址是一个多播地址
RTN_BROADCAST	目的地址是一个广播地址。匹配的输入的数据报以广播方式送往本地，匹配的输出的数据报以广播方式发送出去
RTN_ANYCAST	在 IPv4 中不使用
RTN_BLACKHOLE RTN_UNREACHABLE RTN_PROHIBIT RTN_THROW	这些值与特定的管理配置相关联，根据类型来执行对应的动作，而与目的地址无关
RTN_NAT	已废弃
RTN_XRESOLVE	由一个外部解析器来处理该路由。目前尚未实现该功能

```
14 u8 fa_scope
```

路由表项的作用范围，见表 19-2。

```
15 u8 fa_state
```

一些标志的位图。目前只有一个标志，即 FA_S_ACCESSED，表示该表项已经被访问过，参见 20.9 节。

19.3.5 fib_info 结构

fib_node 结构和 fib_alias 结构的组合用于标识一条路由表项，同时存储相关信息，而更多的信息，比如下一跳网关等重要的路由信息则存储在 fib_info 结构中。

```
70 struct fib_info {
71     struct hlist_node    fib_hash;
72     struct hlist_node    fib_lhash;
73     int             fib_treeref;
74     atomic_t          fib_clntref;
75     int             fib_dead;
76     unsigned          fib_flags;
77     int             fib_protocol;
78     __be32             fib_prefsrc;
```

```
79     u32             fib_priority;
80     u32             fib_metrics[RTAX_MAX];
81 #define fib_mtu fib_metrics[RTAX_MTU-1]
82 #define fib_window fib_metrics[RTAX_WINDOW-1]
83 #define fib_rtt fib_metrics[RTAX_RTT-1]
84 #define fib_advmss fib_metrics[RTAX_ADVMSS-1]
85     int             fib_nhs;
86 #ifdef CONFIG_IP_ROUTE_MULTIPATH
87     int             fib_power;
88 #endif
89 #ifdef CONFIG_IP_ROUTE_MULTIPATH_CACHED
90     u32             fib_mp_alg;
91 #endif
92     struct fib_nh       fib_nh[0];
93 #define fib_dev       fib_nh[0].nh_dev
94 };
```

71 struct hlist_node fib_hash

通过 fib_hash 将 fib_info 实例插入到 fib_info_hash 散列表中，并且所有的 fib_info 实例都会插入到 fib_info_hash 散列表中。

72 struct hlist_node fib_lhash

将 fib_info 实例插入到 fib_info_laddrhash 散列表中。在路由表项有一个首选源地址时，才将 fib_info 结构插入到 fib_info_laddrhash 散列表中。

73 int fib_treeref

74 atomic_t fib_clntref

引用计数。fib_treeref 是持有该 fib_info 实例引用的 fib_node 数据结构的数目，fib_clntref 是由于路由查找成功而被持有的引用计数。

75 int fib_dead

标记路由项正在被删除的标志。当该标志被设置为 1 时，警告该数据结构将被删除而不能再使用。

76 unsigned fib_flags

当前使用的唯一标志是 RTNH_F_DEAD，表示下一跳已无效，在支持多路径条件下使用。

77 int fib_protocol

设置路由的协议，见表 19-4。

表 19-4 fib_protocol 的取值

fib_protocol	描述
RTPROT_UNSPEC	表示该字段无效
RTPROT_REDIRECT	由 ICMP 重定向设置的路由，当前的 IPv4 不使用该标志
RTPROT_KERNEL	由内核设置的路由。参见 19.9.6 节
RTPROT_BOOT	由诸如 ip route 和 route 等用户空间命令设置的路由
RTPROT_STATIC	由管理员设置的路由。目前已废弃不用
RTPROT_GATED	由 GateD 添加的路由
RTPROT_RA	由 rdisc 通告添加的路由。在 RFC 1256 中定义的 ICMP 路由器发现协议（Router Discovery Protocol）可以使主机找到相邻的路由器
RTPROT_MRT	由多线程路由工具包（Multi-Threaded Routing Toolkit，MRT）添加的路由

（续）

fib_protocol	描述
RTPROT_ZEBRA	由 Zebra 添加的路由
RTPROT_BIRD	由 BIRD 添加的路由
RTPROT_DNROUTED	由 DECnet 路由守护进程添加的路由
RTPROT_XORP	由 XORP 路由守护进程添加的路由

fib_protocol 取值大于 RTPROT_STATIC 的路由项不是由内核生成，由用户空间路由协议生成，比如路由守护进程在与内核通信时，操作只能局限于它们自己生成的路由项。

```
78 be32 fib_prefsrc
```

首选源 IP 地址。

```
79 u32 fib_priority
```

路由优先级，值越小则优先级越高。添加路由表项时，当没有明确设定时，它的值初始化为默认值 0。

```
80 u32 fib_metrics[RTAX_MAX]
```

与路由相关的一组度量值，见表 19-5。当配置路由时，可以通过 ip route 命令设定，默认值为 0。

表 19-5 Metric

Metric	描述
RTAX_LOCK	不是度量值，而是以比特位图位标识各种度量值的上锁状态，当位置 n 的比特位被设置时，表示已经对值为 n 的度量值上锁，不允许修改。目前只针对 RTAX_MTU 有效
RTAX_MTU	路径 MTU
RTAX_WINDOW	最大通告窗口
RTAX_RTT	往返时间（RTT）
RTAX_RTTVAR	RTT 方差
RTAX_SSTHRESH	慢启动门限值
RTAX_CWND	拥塞窗口
RTAX_ADVMSS	最大段长度
RTAX_REORDERING	当前的 reordering
RTAX_HOPLIMIT	默认生存时间
RTAX_INITCWND	初始拥塞窗口

```
87 int fib_power
```

当内核编译支持多路径路由时，用于实现加权随机轮转算法。

```
90 u32 fib_mp_alg
```

当内核编译支持多路径路由时，标识多路径缓存算法。

```
85 int fib_nhs
```

可用下一跳的数量。通常为 1，只有当内核支持多路径时，fib_nhs 才可能大于 1。

```
92 struct fib_nh fib_nh[0]
```

在支持多路径路由时的下一跳散列表。

19.3.6 fib_nh 结构

fib_nh 结构存放着下一跳路由的地址（nh_gw）。通常情况下一个路由会有一个该结构，然而当支持多路由路径时，一个路由（fib_alias）可能有多个 fib_nh 结构，说明这个路由有多个下一跳地址。下一跳地址的选择也有多种算法，这些算法都是基于 nh_weight、nh_power 成员的。

```
49 struct fib_nh {
50     struct net_device    *nh_dev;
51     struct hlist_node    nh_hash;
52     struct fib_info      *nh_parent;
53     unsigned       nh_flags;
54     unsigned char       nh_scope;
55 #ifdef CONFIG_IP_ROUTE_MULTIPATH
56     int          nh_weight;
57     int          nh_power;
58 #endif
59 #ifdef CONFIG_NET_CLS_ROUTE
60     __u32          nh_tclassid;
61 #endif
62     int          nh_oif;
63     __be32          nh_gw;
64 };
```

50 struct net_device *nh_dev

该路由表项输出网络设备。

51 struct hlist_node nh_hash

用于将 nh_hash 链入散列表。

52 struct fib_info *nh_parent

指向所属的路由表项的 fib_info 结构。

53 unsigned nh_flags

一些标志，见表 19-8。

54 unsigned char nh_scope

路由范围。

56 int nh_weight

57 int nh_power

当内核编译支持多路径路由时，用于实现加权随机轮转算法。

60 __u32 nh_tclassid

基于策略路由的分类标签。

62 int nh_oif

该路由表项的输出网络设备索引。

63 __be32 nh_gw

路由项的网关地址。

19.4 路由表的初始化

路由表的初始化是由 fib_hash_init()实现的。ip_fib_main_table 和 ip_fib_local_table 表的创建是由初始化 IP 路由子系统的 ip_fib_init()调用 fib_hash_init()实现的。在支持策略路由时，在创建新的路由表后，也会调用此函数进行初始化。

参数 id 为进行初始化路由表的 id，ip_fib_main_table 和 ip_fib_local_table 表的 id 分别为 RT_TABLE_LOCAL 和 RT_TABLE_MAIN。

```
761 #ifdef CONFIG_IP_MULTIPLE_TABLES
762 struct fib_table * fib_hash_init(u32 id)
763 #else
764 struct fib_table * __init fib_hash_init(u32 id)
765 #endif
766 {
767     struct fib_table *tb;
768
769     if (fn_hash_kmem == NULL)
770         fn_hash_kmem = kmem_cache_create("ip_fib_hash",
771                                          sizeof(struct fib_node),
772                                          0, SLAB_HWCACHE_ALIGN,
773                                          NULL, NULL);
774
775     if (fn_alias_kmem == NULL)
776         fn_alias_kmem = kmem_cache_create("ip_fib_alias",
777                                          sizeof(struct fib_alias),
778                                          0, SLAB_HWCACHE_ALIGN,
779                                          NULL, NULL);
780
781     tb = kmalloc(sizeof(struct fib_table) + sizeof(struct fn_hash),
782             GFP_KERNEL);
783     if (tb == NULL)
784         return NULL;
785
786     tb->tb_id = id;
787     tb->tb_lookup = fn_hash_lookup;
788     tb->tb_insert = fn_hash_insert;
789     tb->tb_delete = fn_hash_delete;
790     tb->tb_flush = fn_hash_flush;
791     tb->tb_select_default = fn_hash_select_default;
792     tb->tb_dump = fn_hash_dump;
793     memset(tb->tb_data, 0, sizeof(struct fn_hash));
794     return tb;
795 }
```

761-765 在没有配置策略路由的情况下，该函数只在系统启动时调用，因而带有__init 宏。但在支持策略路由的情况下，可以在任意时刻创建一个新路由表，所以 fib_hash_init 不带有__init 宏。

769-779 当 fib_hash_init()被首次调用时，创建用于分配 fib_node 数据结构的内存池 fn_hash_kmem，以及用于分配 fib_alias 数据结构的内存池 fn_alias_kmem。

786-793 然后分配一个 fib_table 数据结构，注意这同时分配一个 fn_hash 结构，这在前

面已经提到，然后初始化路由表结构中的钩子函数，最后将 fn_hash 的内容清空。

19.5 netlink 接口

Linux 兼容 UNIX，因此提供了 net-tools 包中的有关命令。例如 route，它通过 ioctl 接口对路由进行相应的操作和配置。然而 Linux 提供了功能更强大的配置工具，那就是 IPROUTE2 包，它通过 Linux 特有的 netlink 接口对路由进行相应的操作和配置。

图 19-6 展示了通过 netlink 接口来操作路由表的主要函数。

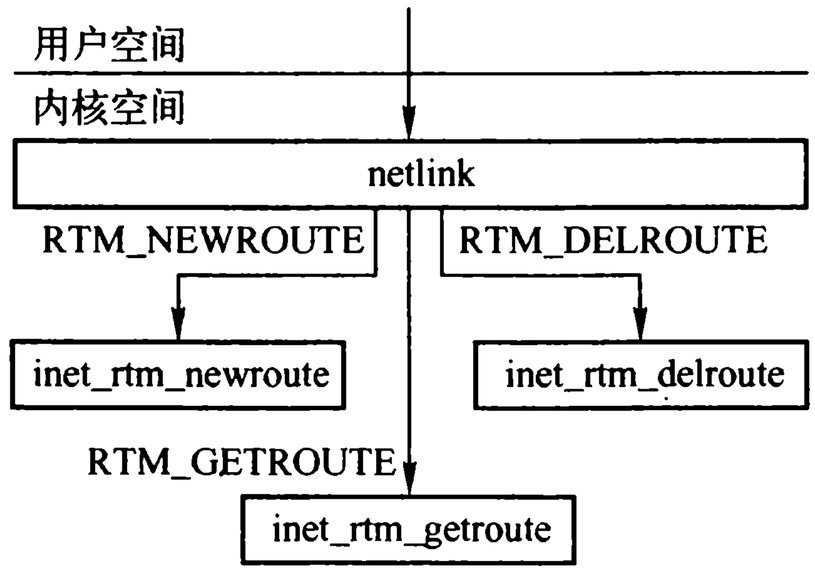

图 19-6 netlink 接口

19.5.1 netlink 路由表项消息结构

配置路由的 netlink 消息格式如图 19-7 所示。

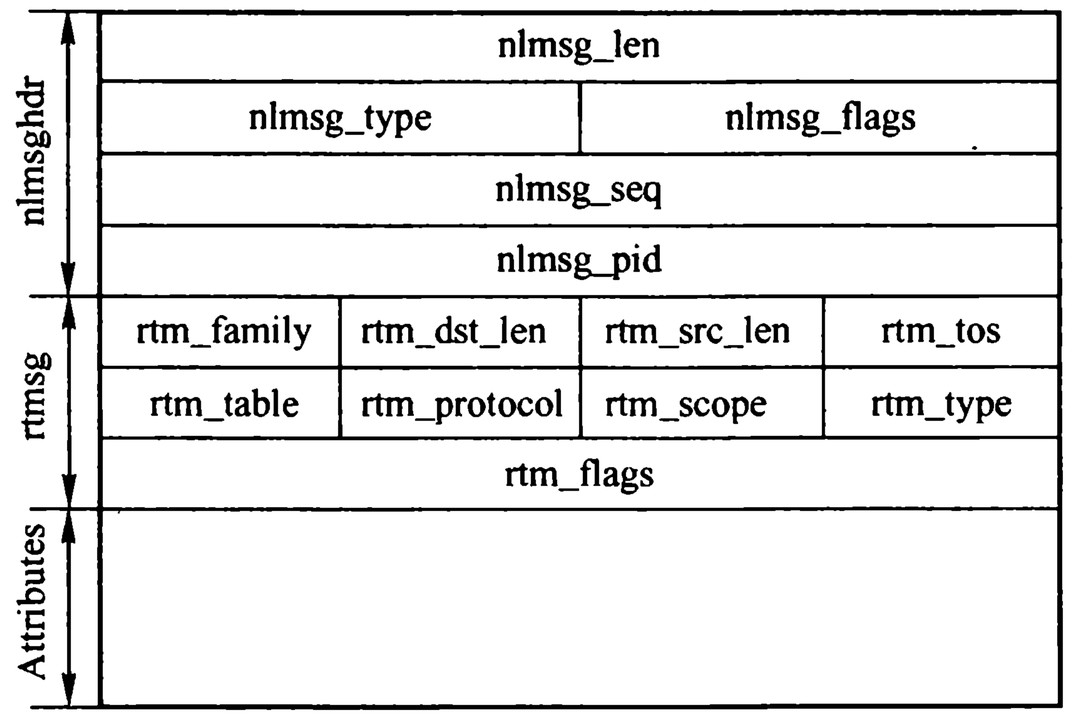

图 19-7 相关操作路由表项消息

nlmsghdr 结构是 netlink 消息的首部，rtmsg 结构是路由消息，这两个结构是紧接在一起的，当然 nlmsghdr 结构须按四字节对齐，这在 nlmsg_data()中可以很清楚地看到，函数的参数是一个 nlmsghdr 结构指针，返回一个 void 指针，其实就是 rtmsg 结构指针，函数中只是简单地将 nlmsghdr 结构的地址加上其对齐后的长度 NLMSG_HDRLEN，因此通过 nlmsghdr 结构可以获得 rtmsg 结构。其中各字段的意义如下：

（1）rtm_family

8 位，标识操作的路由表项所属的地址族，如 AF_INET 和 AF_INET6 等。

（2）rtm_dst_len

8 位，目的地址掩码长度。

（3）rtm_src_len

8 位，源地址掩码长度。

（4）rtm_tos

8 位，路由的服务类型（TOS）比特位字段。

（5）rtm_table

8 位，路由表 ID，支持 255 个路由表，但 RT_TABLE_LOCA 表只能有内核操作。用户可以指定操作 RT_TABLE_UNSPEC 和 RT_TABLE_DEFAULT 之间的路由表，见表 19-6。

表 19-6　rtm_table 的取值

rtm_table	描述
RT_TABLE_UNSPEC	未指定的路由表
RT_TABLE_DEFAULT	用于策略路由的默认路由表
RT_TABLE_MAIN	主路由表
RT_TABLE_LOCAL	本地路由表

（6）rtm_protocol

8 位，标明该路由的特性或标志，见表 19-7。

表 19-7　rtm_protocol 的取值

rtm_protocol	描述
RTPROT_UNSPEC	未知
RTPROT_REDIRECT	通过 ICMP 重定向消息添加
RTPROT_KERNEL	通过内核添加
RTPROT_BOOT	通过 bootup 添加
RTPROT_STATIC	由管理员通过命令添加

（7）rtm_scope

8 位，路由寻址范围，见表 6-1。

（8）rtm_type

8 位，路由表项的类型，见表 19-3。

（9）rtm_flags

32 位，一些标志，见表 19-8。

表 19-8　rtm_flags 的取值

rtm_flags	描述
RTM_F_NOTIFY	如果这条路由项发生修改，需要通知用户进程
RTM_F_CLONED	这条路由项是由其他路由项克隆而来的
RTM_F_EQUALIZE	在启用多路径路由时，允许随机取下一跳，目前尚未使用

（10）Attribute

属性，标识配置的各种值，见表 19-9。

表 19-9　Attribute 的取值

属性	描述
RTA_UNSPEC	忽略
RTA_DST	路由项的目的地址
RTA_SRC	路由项的源地址
RTA_IIF	路由项的输入网络设备索引
RTA_OIF	路由项的输出网络设备索引
RTA_GATEWAY	路由项的网关地址
RTA_PRIORITY	路由项的优先级
RTA_PREFSRC	首选源地址
RTA_METRICS	路由和协议相关的度量值（比如 RTT，PMTU 等）
RTA_MULTIPATH	多路径路由下一跳的属性值
RTA_PROTOINFO	基于策略路由的防火墙属性值
RTA_FLOW	基于策略路由的分类标签
RTA_CACHEINFO	缓存的路由信息

19.5.2　inet_rtm_newroute()

当通过 netlink，操作类型为 RTM_NEWROUTE 对路由进行配置时，inet_rtm_newroute() 被调用。

```
560 int inet_rtm_newroute(struct sk_buff *skb, struct nlmsghdr* nlh, void *arg)
561 {
562     struct fib_config cfg;
563     struct fib_table *tb;
564     int err;
565
566     err = rtm_to_fib_config(skb, nlh, &cfg);
567     if (err < 0)
568         goto errout;
569
570     tb = fib_new_table(cfg.fc_table);
571     if (tb == NULL) {
572         err = -ENOBUFS;
573         goto errout;
574     }
575
576     err = tb->tb_insert(tb, &cfg);
577 errout:
578     return err;
579 }
```

566-568 从 netlink 消息格式的参数中获取用于配置的路由表项的信息到 fib_config 结构实例。

570-574 根据给定的路由表 ID 获取路由表，不支持策略路由时返回的是 local 和 main 两个路由表中的一个，而支持策略路由时查找散列表 fib_table_hash 获取，参见 19.6 节。

576 获取路由表之后，通过路由表中的 tb_insert 接口创建路由表项并添加到路由表中，

参见 19.7 节。

19.5.3 inet_rtm_delroute()

inet_rtm_delroute()与 inet_rtm_newroute()几乎完全一样，只是在最后调用的是路由表的 tb_delete 接口来删除相应的路由表项。

19.6 获取指定的路由表

fib_new_table()用于获取指定的路由表。

fib_new_table()有两个版本，在不支持策略路由的情况下，由于在初始化时已经创建了 ip_fib_main_table 和 ip_fib_local_table 路由表，因此直接获取指定 id 的路由表即可。

```
184 static inline struct fib_table *fib_new_table(u32 id)
185 {
186     return fib_get_table(id);
187 }
```

在支持策略路由时，路由表最多可多达 255 个，随时都有可能创建新的路由表，因此通过 fib_get_table()在 fib_table_hash 散列表中查找指定的路由表。如果不存在，则通过 fib_hash_init()创建并初始化新创建的路由表，然后添加到 fib_table_hash 散列表中并返回。

```
66 struct fib_table *fib_new_table(u32 id)
67 {
68     struct fib_table *tb;
69     unsigned int h;
70
71     if (id == 0)
72         id = RT_TABLE_MAIN;
73     tb = fib_get_table(id);
74     if (tb)
75         return tb;
76     tb = fib_hash_init(id);
77     if (!tb)
78         return NULL;
79     h = id & (FIB_TABLE_HASHSZ - 1);
80     hlist_add_head_rcu(&tb->tb_hlist, &fib_table_hash[h]);
81     return tb;
82 }
```

19.7 路由表项的添加

添加一条新的路由表项是通过 fn_hash_insert()来实现的。根据 netlink 消息首部中的 nlmsg_flags 字段，除了插入新的路由项以外，包括在尾部追加、首部追加和替换等都会调用该函数实现对应的功能，因此该函数的实现比较复杂。

当添加一条新路由表项时，如果路由表中已经存在目的地与 TOS 都相同的路由表项，添加的标志为 NLM_F_EXCL 时则函数返回错误，而添加的标志为 NLM_F_REPLACE 时才能进

行处理。

插入一条路由表项后，如果 fz_hash 散列表容量发生了变化，则通过 fn_rehash_zone 重建散列表。

参数说明如下：

- tb，待添加的路由项。
- cfg，添加路由项的信息。

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

24 u8 fc_dst_len

目的地址掩码长度。

25 u8 fc_tos

路由的服务类型（TOS）位字段。

26 u8 fc_protocol

标明该路由的特性。

27 u8 fc_scope

路由范围。

28 u8 fc_type

路由表项的类型。

30 u32 fc_table

路由表 ID。

31 __be32 fc_dst

路由项的目的地址。

32 __be32 fc_gw

路由项的网关地址。

33 int fc_oif

路由项的输出网络设备索引。

34 u32 fc_flags

一些标志，见表 19-8。

35 u32 fc_priority

路由项的优先级。

36 __be32 fc_prefsrc

首选源地址。

37 struct nlattr *fc_mx

39 int fc_mx_len

路由和协议相关的度量值（比如 RTT，PMTU 等）。

38 struct rtnexthop *fc_mp

40 int fc_mp_len

多路径路由下一跳的属性值。

41 u32 fc_flow

基于策略路由的分类标签。

42 u32 fc_mp_alg

多路径缓存算法。

43 u32 fc_nlflags

操作模式，见表 19-10。

表 19-10　fc_nlflags 的取值

fc_nlflags	描述
NLM_F_REPLACE	如果存在则替换
NLM_F_EXCL	如果存在则不添加
NLM_F_CREATE	如果不存在则创建
NLM_F_APPEND	添加到最后

44 struct nl_info fc_nlinfo

配置路由的 netlink 数据包信息。

```
382 static int fn_hash_insert(struct fib_table *tb, struct fib_config *cfg)
383 {
384     struct fn_hash *table = (struct fn_hash *) tb->tb_data;
385     struct fib_node *new_f, *f;
386     struct fib_alias *fa, *new_fa;
387     struct fn_zone *fz;
388     struct fib_info *fi;
389     u8 tos = cfg->fc_tos;
390     __be32 key;
391     int err;
392
393     if (cfg->fc_dst_len > 32)
394         return -EINVAL;
395
```

```
396     fz = table->fn_zones[cfg->fc_dst_len];
397     if (!fz && !(fz = fn_new_zone(table, cfg->fc_dst_len)))
398         return -ENOBUFS;
399
400     key = 0;
401     if (cfg->fc_dst) {
402         if (cfg->fc_dst & ~FZ_MASK(fz))
403             return -EINVAL;
404         key = fz_key(cfg->fc_dst, fz);
405     }
406
407     fi = fib_create_info(cfg);
408     if (IS_ERR(fi))
409         return PTR_ERR(fi);
410
411     if (fz->fz_nent > (fz->fz_divisor<<1) &&
412         fz->fz_divisor < FZ_MAX_DIVISOR &&
413         (cfg->fc_dst_len == 32 ||
414          (1 << cfg->fc_dst_len) > fz->fz_divisor))
415         fn_rehash_zone(fz);
416
417     f = fib_find_node(fz, key);
418
419     if (!f)
420         fa = NULL;
421     else
422         fa = fib_find_alias(&f->fn_alias, tos, fi->fib_priority);
```

393-394 检测掩码长度是否有效。

396-398 获取指定掩码长度的 zone。如果该 zone 不存在，则调用 fn_new_zone()创建一个新的 zone 结构，并将其链接到活动 zone 结构的循环单链表中，且由 fn_hash 的 fn_zones 数组相应元素指向它。

400-405 获取目的网络对应 fib_node 实例的 key，通过目标地址和目标地址掩码得到目标地址网络键值，用于之后查找相应的 fib_node 实例。

407-409 根据路由项信息创建 fib_info 结构实例。

411-415 fz_hash 散列表容量可能发生变化，需要重建散列表。

417-422 根据 key 获取目的网络对应 fib_node 实例，然后进一步根据 tos 和优先级匹配对应的 fib_alias 实例。

```
424     /* Now fa, if non-NULL, points to the first fib alias
425      * with the same keys [prefix,tos,priority], if such key already
426      * exists or to the node before which we will insert new one.
427      *
428      * If fa is NULL, we will need to allocate a new one and
429      * insert to the head of f.
430      *
431      * If f is NULL, no fib node matched the destination key
432      * and we need to allocate a new one of those as well.
433      */
434
435     if (fa && fa->fa_tos == tos &&
436         fa->fa_info->fib_priority == fi->fib_priority) {
```

```
437         struct fib_alias *fa_orig;
438
439         err = -EEXIST;
440         if (cfg->fc_nlflags & NLM_F_EXCL)
441             goto out;
442
443         if (cfg->fc_nlflags & NLM_F_REPLACE) {
444             struct fib_info *fi_drop;
445             u8 state;
446
447             write_lock_bh(&fib_hash_lock);
448             fi_drop = fa->fa_info;
449             fa->fa_info = fi;
450             fa->fa_type = cfg->fc_type;
451             fa->fa_scope = cfg->fc_scope;
452             state = fa->fa_state;
453             fa->fa_state &= ~FA_S_ACCESSED;
454             fib_hash_genid++;
455             write_unlock_bh(&fib_hash_lock);
456
457             fib_release_info(fi_drop);
458             if (state & FA_S_ACCESSED)
459                 rt_cache_flush(-1);
460             return 0;
461         }
462
463         /* Error if we find a perfect match which
464          * uses the same scope, type, and nexthop
465          * information.
466          */
467         fa_orig = fa;
468         fa = list_entry(fa->fa_list.prev, struct fib_alias, fa_list);
469         list_for_each_entry_continue(fa, &f->fn_alias, fa_list) {
470             if (fa->fa_tos != tos)
471                 break;
472             if (fa->fa_info->fib_priority != fi->fib_priority)
473                 break;
474             if (fa->fa_type == cfg->fc_type &&
475                 fa->fa_scope == cfg->fc_scope &&
476                 fa->fa_info == fi)
477                 goto out;
478         }
479         if (!(cfg->fc_nlflags & NLM_F_APPEND))
480             fa = fa_orig;
481     }
```

435-481 处理存在 tos 和优先级完全相同的 fib_alias 实例的情况。

440-441 添加路由项的标志中存在 NLM_F_EXCL，表示如果存在就不要创建，直接返回。

443-461 添加路由项的标志中存在 NLM_F_REPLACE，表示如果存在就进行替换，因此将相关的值替换到该路由项中，并刷新路由缓存表，完成后返回。

467-480 如果通过 NLM_F_EXCL 和 NLM_F_REPLACE 匹配不到添加标志，那就需要通过 scope，type 和 nexthop 进行全匹配，匹配成功则表明存在相同的路由项，不需进行任何操作。否则，如果存在 NLM_F_APPEND 标志，则进行替换。

```
483     err = -ENOENT;
484     if (!(cfg->fc_nlflags & NLM_F_CREATE))
485         goto out;
486
487     err = -ENOBUFS;
488     new_fa = kmem_cache_alloc(fn_alias_kmem, GFP_KERNEL);
489     if (new_fa == NULL)
490         goto out;
491
492     new_f = NULL;
493     if (!f) {
494         new_f = kmem_cache_alloc(fn_hash_kmem, GFP_KERNEL);
495         if (new_f == NULL)
496             goto out_free_new_fa;
497
498         INIT_HLIST_NODE(&new_f->fn_hash);
499         INIT_LIST_HEAD(&new_f->fn_alias);
500         new_f->fn_key = key;
501         f = new_f;
502     }
503
504     new_fa->fa_info = fi;
505     new_fa->fa_tos = tos;
506     new_fa->fa_type = cfg->fc_type;
507     new_fa->fa_scope = cfg->fc_scope;
508     new_fa->fa_state = 0;
```

484-485 不存在 tos 和优先级完全匹配的 fib_alias 实例，且添加路由项的标志中不存在 NLM_F_CREATE，不能添加新的路由表项，则退出。

487-508 为新的路由项分配 fib_alias 实例和与它目的网络对应 fib_node 实例。如果分配任何个实例失败，都会直接返回。分配成功后，将相关值设置到新路由项中。

487-508 为新的路由项分配 fib_alias 实例，如果之前未找到匹配的 fib_node 实例，也需要分配。分配任何实例失败，都会直接返回。如果分配成功，则将相关值设置到新的结构中。

```
510     /*
511      * Insert new entry to the list.
512      */
513
514     write_lock_bh(&fib_hash_lock);
515     if (new_f)
516         fib_insert_node(fz, new_f);
517     list_add_tail(&new_fa->fa_list,
518          (fa ? &fa->fa_list : &f->fn_alias));
519     fib_hash_genid++;
520     write_unlock_bh(&fib_hash_lock);
521
522     if (new_f)
523         fz->fz_nent++;
524     rt_cache_flush(-1);
525
526     rtmsg_fib(RTM_NEWROUTE, key, new_fa, cfg->fc_dst_len, tb->tb_id,
527          &cfg->fc_nlinfo);
528     return 0;
```

```
529
530 out_free_new_fa:
531     kmem_cache_free(fn_alias_kmem, new_fa);
532 out:
533     fib_release_info(fi);
534     return err;
535 }
```

514-520 将路由项对应的 fib_alias 实例和 fib_node 实例插入到对应的链表中。

522-524 刷新该路由项对应 zone 中的路由项数目，同时刷新路由缓存表。

526 将添加新路由项的消息通过 netlink 通知感兴趣的进程。

19.8 路由表项的删除

fn_hash_delete()实现删除一条路由表项。先构造搜索条件，然后用它来查找待删除的表项是否存在。当查找到符合条件的 fib_alias 实例时，就删除它并通过 netlink 广播通知感兴趣的子系统，如果该路由表项已经被使用则刷新路由缓存。

```
538 static int fn_hash_delete(struct fib_table *tb, struct fib_config *cfg)
539 {
540     struct fn_hash *table = (struct fn_hash*)tb->tb_data;
541     struct fib_node *f;
542     struct fib_alias *fa, *fa_to_delete;
543     struct fn_zone *fz;
544     __be32 key;
545
546     if (cfg->fc_dst_len > 32)
547         return -EINVAL;
548
549     if ((fz  = table->fn_zones[cfg->fc_dst_len]) == NULL)
550         return -ESRCH;
551
552     key = 0;
553     if (cfg->fc_dst) {
554         if (cfg->fc_dst & ~FZ_MASK(fz))
555             return -EINVAL;
556         key = fz_key(cfg->fc_dst, fz);
557     }
558
559     f = fib_find_node(fz, key);
560
561     if (!f)
562         fa = NULL;
563     else
564         fa = fib_find_alias(&f->fn_alias, cfg->fc_tos, 0);
565     if (!fa)
566         return -ESRCH;
```

546-547 检测掩码长度是否有效。

549-550 获取指定掩码长度的 zone，如果获取不到，则肯定没有需删除的路由项，则可直接返回了。

552-559 获取目的网络对应的网络键值key，并根据key获取目的网络对应fib_node实例。

561-566 如果没有对应的fib_node实例，则删除失败。否则，根据tos和优先级匹配对应的路由项和fib_alias实例。

```
568     fa_to_delete = NULL;
569     fa = list_entry(fa->fa_list.prev, struct fib_alias, fa_list);
570     list_for_each_entry_continue(fa, &f->fn_alias, fa_list) {
571        struct fib_info *fi = fa->fa_info;
572
573        if (fa->fa_tos != cfg->fc_tos)
574           break;
575
576        if ((!cfg->fc_type ||
577             fa->fa_type == cfg->fc_type) &&
578            (cfg->fc_scope == RT_SCOPE_NOWHERE ||
579             fa->fa_scope == cfg->fc_scope) &&
580            (!cfg->fc_protocol ||
581             fi->fib_protocol == cfg->fc_protocol) &&
582            fib_nh_match(cfg, fi) == 0) {
583            fa_to_delete = fa;
584            break;
585        }
586     }
```

查找匹配待删除的路由项。

```
588     if (fa_to_delete) {
589        int kill_fn;
590
591        fa = fa_to_delete;
592        rtmsg_fib(RTM_DELROUTE, key, fa, cfg->fc_dst_len,
593              tb->tb_id, &cfg->fc_nlinfo);
594
595        kill_fn = 0;
596        write_lock_bh(&fib_hash_lock);
597        list_del(&fa->fa_list);
598        if (list_empty(&f->fn_alias)) {
599           hlist_del(&f->fn_hash);
600           kill_fn = 1;
601        }
602        fib_hash_genid++;
603        write_unlock_bh(&fib_hash_lock);
604
605        if (fa->fa_state & FA_S_ACCESSED)
606           rt_cache_flush(-1);
607        fn_free_alias(fa);
608        if (kill_fn) {
609           fn_free_node(f);
610           fz->fz_nent--;
611        }
612
613        return 0;
614     }
615     return -ESRCH;
616 }
```

如果查找命中待删除的路由项，则将其删除，否则返回失败。

19.9 外部事件

路由子系统需要知道网络设备的状态和 IP 地址的变化，这样当路由子系统接收到这样的事件后，可以根据事件做出相应的操作。

19.9.1 网络设备状态变化事件

当网络设备的状态发生变化，路由子系统通过 fib netdev notifier 收到通知注册到 netdev_chain 通知链，然后调用 fib_netdev_event()来处理相关事件。参数说明如下：

- event，通知的事件，见表 5-6。
- ptr，指向状态发生变化的网络设备。

```
858 static int fib_netdev_event(struct notifier_block *this, unsigned long event,
    void *ptr)
859 {
860     struct net_device *dev = ptr;
861     struct in_device *in_dev = __in_dev_get_rtnl(dev);
862
863     if (event == NETDEV_UNREGISTER) {
864         fib_disable_ip(dev, 2);
865         return NOTIFY_DONE;
866     }
867
868     if (!in_dev)
869         return NOTIFY_DONE;
870
871     switch (event) {
872     case NETDEV_UP:
873         for_ifa(in_dev) {
874             fib_add_ifaddr(ifa);
875         } endfor_ifa(in_dev);
876 #ifdef CONFIG_IP_ROUTE_MULTIPATH
877         fib_sync_up(dev);
878 #endif
879         rt_cache_flush(-1);
880         break;
881     case NETDEV_DOWN:
882         fib_disable_ip(dev, 0);
883         break;
884     case NETDEV_CHANGEMTU:
885     case NETDEV_CHANGE:
886         rt_cache_flush(0);
887         break;
888     }
889     return NOTIFY_DONE;
890 }
```

863-866 如果网络设备注销，则清除该网络设备的网络功能信息和相关功能。

868-869 如果网络设备的 IP 配置块无效，则不做处理。

872-880 当激活网络设备时，则根据配置在该设备上的本地地址添加路由表项到RT_TABLE_LOCAL 路由表中，然后延时刷新路由缓存。

881-883 当关闭网络设备时，则清除该网络设备的网络功能信息和相关功能。

884-887 当网络设备修改了 MTU 或状态配置发生变化，则立刻刷新路由缓存。

19.9.2 IP 地址变化事件

当一个网络设备的 IP 地址发生变化，路由子系统通过 fib_inetaddr_notifier 收到通知注册到 inetaddr_chain 通知链，然后调用 fib_inetaddr_event()处理添加或删除 IP 地址的事件。参数说明如下：

- event，通知的事件，见表 6-7。
- ptr，指向 IP 地址发生变化的 IP 地址块。

```
831 static int fib_inetaddr_event(struct notifier_block *this, unsigned long
    event, void *ptr)
832 {
833     struct in_ifaddr *ifa = (struct in_ifaddr*)ptr;
834
835     switch (event) {
836     case NETDEV_UP:
837         fib_add_ifaddr(ifa);
838 #ifdef CONFIG_IP_ROUTE_MULTIPATH
839         fib_sync_up(ifa->ifa_dev->dev);
840 #endif
841         rt_cache_flush(-1);
842         break;
843     case NETDEV_DOWN:
844         fib_del_ifaddr(ifa);
845         if (ifa->ifa_dev->ifa_list == NULL) {
846             /* Last address was deleted from this interface.
847               Disable IP.
848              */
849             fib_disable_ip(ifa->ifa_dev->dev, 1);
850         } else {
851             rt_cache_flush(-1);
852         }
853         break;
854     }
855     return NOTIFY_DONE;
856 }
```

836-842 添加了一个新的本地地址之后，根据该本地地址添加路由表项到RT_TABLE_LOCAL 路由表中，然后延时刷新路由缓存。

843-853 删除了一个本地地址之后，将该地址从 RT_TABLE_LOCAL 路由表中删除。如果该设备的 IP 地址被全部删除，则立刻刷新路由缓存和该网络设备 ARP 表，并停止 ARP 功能。

19.9.3 fib_add_ifaddr()

当网络设备上添加了一个新地址之后，便会调用 fib_add_ifaddr()函数进行路由表项的操作。该设备可能处于禁用状态，因此需要检测，只有启用了设备后，才可能配置 RT_TABLE_MAIN 或 RT_TABLE_LOCAL 路由表，见图 19-8。

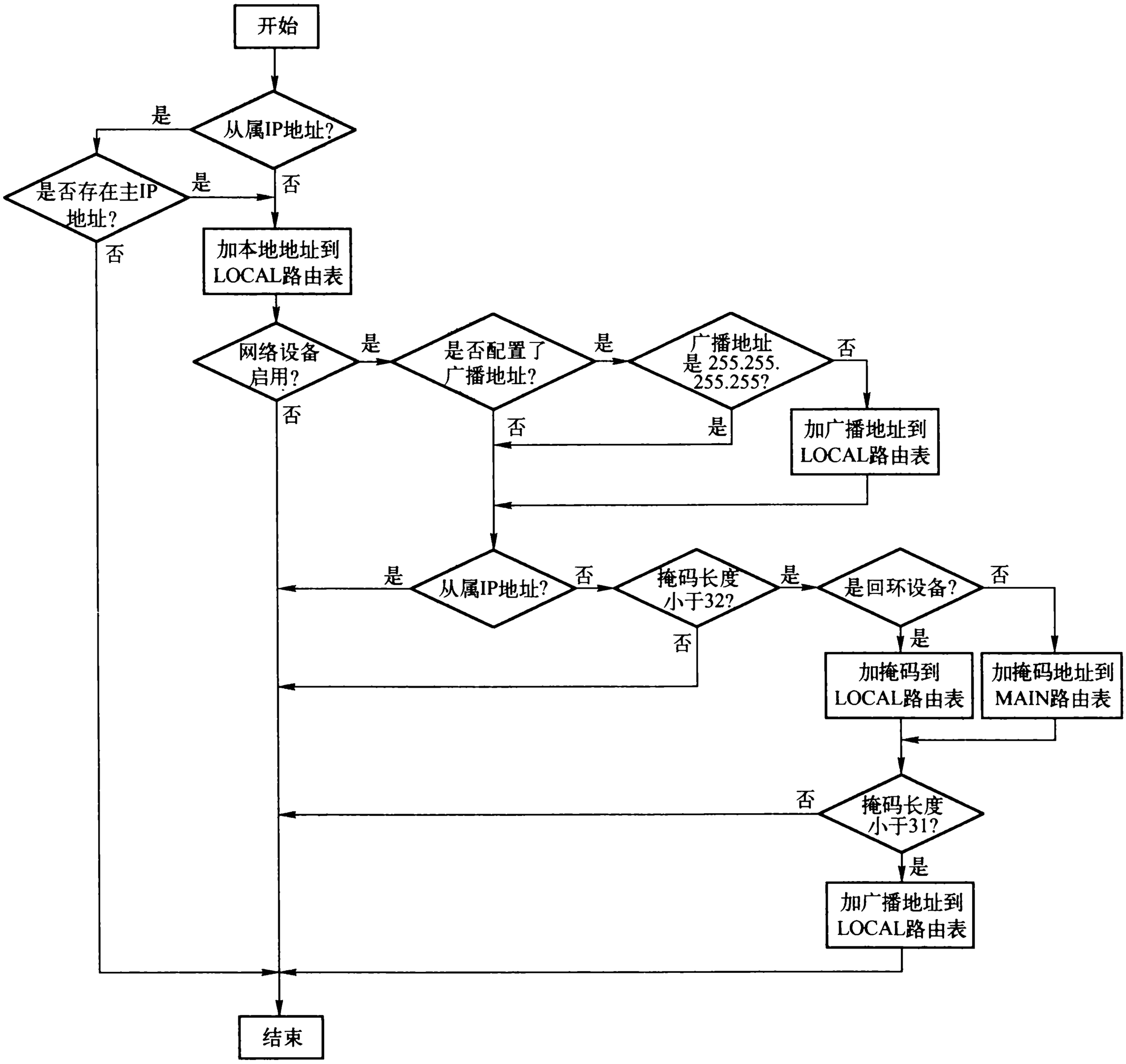

图 19-8　fib_add_ifaddr()流程图

参数 ifa 为添加的新地址。

```
659 void fib_add_ifaddr(struct in_ifaddr *ifa)
660 {
661     struct in_device *in_dev = ifa->ifa_dev;
662     struct net_device *dev = in_dev->dev;
663     struct in_ifaddr *prim = ifa;
664     __be32 mask = ifa->ifa_mask;
665     __be32 addr = ifa->ifa_local;
666     __be32 prefix = ifa->ifa_address&mask;
667
668     if (ifa->ifa_flags&IFA_F_SECONDARY) {
669         prim = inet_ifa_byprefix(in_dev, prefix, mask);
670         if (prim == NULL) {
671             printk(KERN_DEBUG "fib_add_ifaddr: bug: prim == NULL\n");
672             return;
673         }
674     }
```

```
675
676     fib_magic(RTM_NEWROUTE, RTN_LOCAL, addr, 32, prim);
677
678     if (!(dev->flags&IFF_UP))
679         return;
680
681     /* Add broadcast address, if it is explicitly assigned. */
682     if (ifa->ifa_broadcast && ifa->ifa_broadcast != htonl(0xFFFFFFFF))
683         fib_magic(RTM_NEWROUTE, RTN_BROADCAST, ifa->ifa_broadcast, 32, prim);
684
685     if (!ZERONET(prefix) && !(ifa->ifa_flags&IFA_F_SECONDARY) &&
686         (prefix != addr || ifa->ifa_prefixlen < 32)) {
687         fib_magic(RTM_NEWROUTE, dev->flags&IFF_LOOPBACK ? RTN_LOCAL :
688             RTN_UNICAST, prefix, ifa->ifa_prefixlen, prim);
689
690         /* Add network specific broadcasts, when it takes a sense */
691         if (ifa->ifa_prefixlen < 31) {
692             fib_magic(RTM_NEWROUTE, RTN_BROADCAST, prefix, 32, prim);
693             fib_magic(RTM_NEWROUTE, RTN_BROADCAST, prefix|~mask, 32, prim);
694         }
695     }
696 }
```

668-674 如果添加的是从属 IP 地址，则先校验添加的从属 IP 地址是否存在主 IP 地址。

676 在 RT_TABLE_LOCAL 路由表添加一条输入到本地表项。

678-679 检测添加 IP 地址的网络设备是否处在启用状态。如果启用，则还需要添加其他类型的路由。

682-683 如果配置了广播地址并且不为 255.255.255.255，则添加广播地址为目的地址路由表项。

685-688 如果添加的是主 IP 地址并且网络掩码长度小于 32，则根据添加地址的网络设备添加路由表项。添加地址的是回环网络设备，则在 RT_TABLE_LOCAL 路由表添加表项，否则在 RT_TABLE_MAIN 路由表中添加表项。

691-694 如果网络掩码长度小于 31，则在 RT_TABLE_LOCAL 路由表中添加两条广播类型的表项。

19.9.4 fib_del_ifaddr()

当网络设备上删除了一个地址之后，便会调用 fib_del_ifaddr()函数进行路由表项的操作。

如果删除的是从属 IP 地址，则需要进行校验。该从属地址必须有处于同一个子网的主地址，否则出错。由于广播地址和掩码并不是总随着主 IP 地址添加而添加，因此需要需要检测那些广播地址确实已经删除了。如果有主 IP 地址或其他从属 IP 地址还在使用广播地址和掩码，则不能删除对应的路由表项，见图 19-9。

参数 ifa 为删除的地址。

```
698 static void fib_del_ifaddr(struct in_ifaddr *ifa)
699 {
700     struct in_device *in_dev = ifa->ifa_dev;
701     struct net_device *dev = in_dev->dev;
702     struct in_ifaddr *ifa1;
703     struct in_ifaddr *prim = ifa;
```

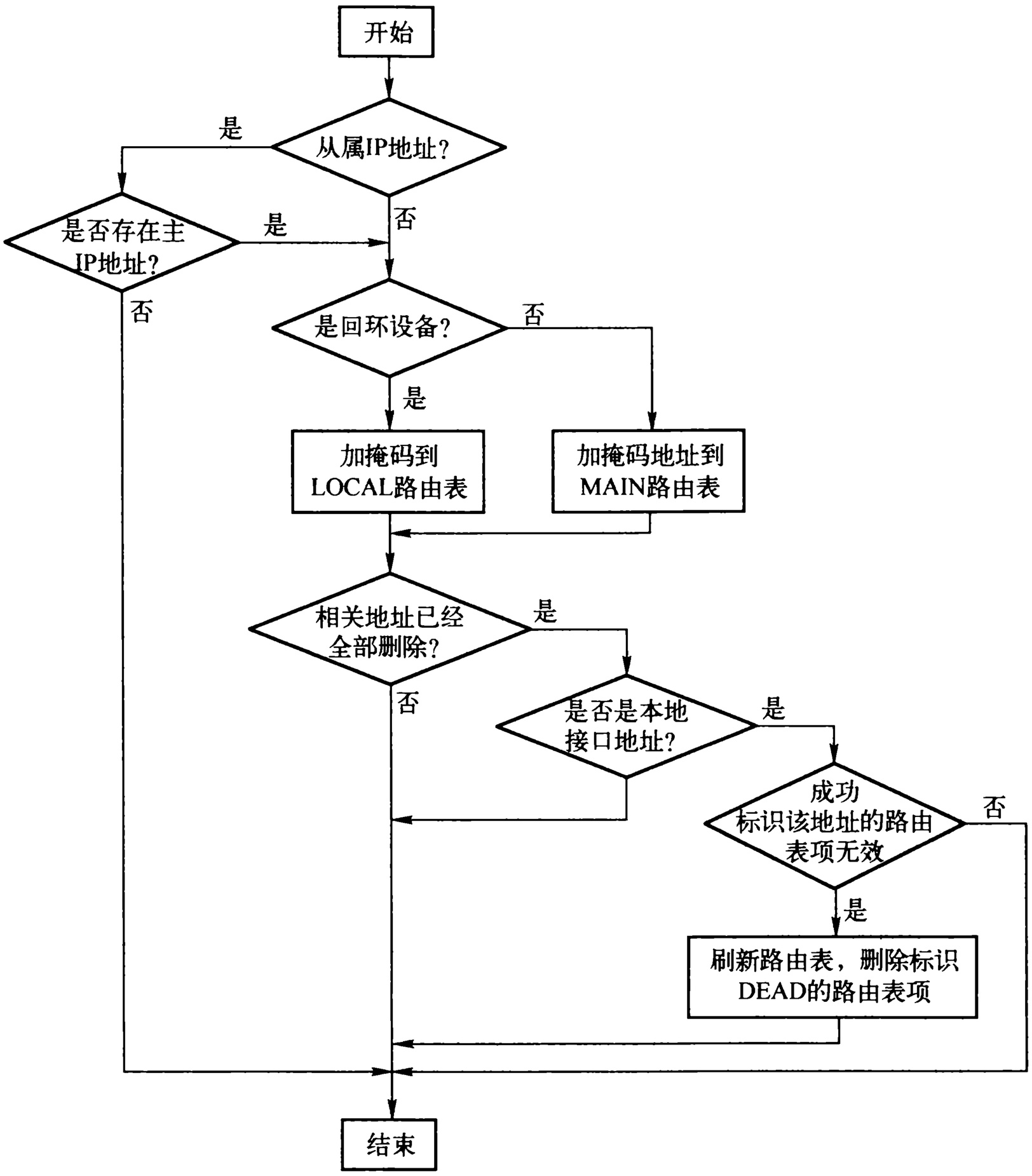

图 19-9　fib_del_ifaddr()流程图

```
704     __be32 brd = ifa->ifa_address|~ifa->ifa_mask;
705     __be32 any = ifa->ifa_address&ifa->ifa_mask;
706 #define LOCAL_OK    1
707 #define BRD_OK      2
708 #define BRD0_OK     4
709 #define BRD1_OK     8
710     unsigned ok = 0;
711
712     if (!(ifa->ifa_flags&IFA_F_SECONDARY))
713         fib_magic(RTM_DELROUTE, dev->flags&IFF_LOOPBACK ? RTN_LOCAL :
714                   RTN_UNICAST, any, ifa->ifa_prefixlen, prim);
715     else {
716         prim = inet_ifa_byprefix(in_dev, any, ifa->ifa_mask);
717         if (prim == NULL) {
718             printk(KERN_DEBUG "fib_del_ifaddr: bug: prim == NULL\n");
719             return;
720         }
721     }
722
723     /* Deletion is more complicated than add.
724        We should take care of not to delete too much :-)
```

```
725
726       Scan address list to be sure that addresses are really gone.
727     */
728
729     for (ifa1 = in_dev->ifa_list; ifa1; ifa1 = ifa1->ifa_next) {
730        if (ifa->ifa_local == ifa1->ifa_local)
731           ok |= LOCAL_OK;
732        if (ifa->ifa_broadcast == ifa1->ifa_broadcast)
733           ok |= BRD_OK;
734        if (brd == ifa1->ifa_broadcast)
735           ok |= BRD1_OK;
736        if (any == ifa1->ifa_broadcast)
737           ok |= BRD0_OK;
738     }
739
740     if (!(ok&BRD_OK))
741        fib_magic(RTM_DELROUTE, RTN_BROADCAST, ifa->ifa_broadcast, 32, prim);
742     if (!(ok&BRD1_OK))
743        fib_magic(RTM_DELROUTE, RTN_BROADCAST, brd, 32, prim);
744     if (!(ok&BRD0_OK))
745        fib_magic(RTM_DELROUTE, RTN_BROADCAST, any, 32, prim);
746     if (!(ok&LOCAL_OK)) {
747        fib_magic(RTM_DELROUTE, RTN_LOCAL, ifa->ifa_local, 32, prim);
748
749        /* Check, that this local address finally disappeared. */
750        if (inet_addr_type(ifa->ifa_local) != RTN_LOCAL) {
751           /* And the last, but not the least thing.
752              We must flush stray FIB entries.
753
754              First of all, we scan fib_info list searching
755              for stray nexthop entries, then ignite fib_flush.
756           */
757           if (fib_sync_down(ifa->ifa_local, NULL, 0))
758              fib_flush();
759        }
760     }
761 #undef LOCAL_OK
762 #undef BRD_OK
763 #undef BRD0_OK
764 #undef BRD1_OK
765 }
```

712-714 如果删除的是主 IP 地址，则根据删除地址的网络设备删除路由表项。删除地址的是回环网络设备，则在 RT_TABLE_LOCAL 路由表删除表项，否则在 RT_TABLE_MAIN 路由表中删除表项。

715-721 如果删除的是从属 IP 地址，则先校验删除的从属 IP 地址是否存在主 IP 地址。

729-738 扫描地址列表，确信该地址已经真正删除了，包括广播地址、本地地址等。

740-745 如果地址列表中已经不存在相同的广播地址了，则删除目的地址是一个广播地址类型的路由表项。

746-760 如果本地地址确实删除了，则删除 RT_TABLE_LOCAL 路由表中的表项。并且，删除的地址不是本地接口的地址，则标识为该地址的路由表项全部无效，然后刷新路由表。

19.9.5 fib_disable_ip()

fib_disable_ip()清除网络设备的网络功能信息和相关功能，比如强制删除通过此网络设备所有路由表项并刷新路由缓存，删除该网络设备的 ARP 表，并停止 ARP 功能。

```
823 static void fib_disable_ip(struct net_device *dev, int force)
824 {
825     if (fib_sync_down(0, dev, force))
826         fib_flush();
827     rt_cache_flush(0);
828     arp_ifdown(dev);
829 }
```

19.9.6 fib_magic()

fib_magic()在本地地址发生了变化后，用于修改 RT_TABLE_MAIN 或 RT_TABLE_LOCAL 路由表，在 fib_add_ifaddr()和 fib_del_ifaddr()被调用。参数说明如下：

（1）cmd，添加或删除路由表项的命令，见表 19-11。

表 19-11 cmd 取值

cmd	描述
RTM_NEWROUTE	添加路由表项
RTM_DELROUTE	删除路由表项

（2）type，用来确定操作的路由表。当 type 为 RTN_UNICAST 操作 RT_TABLE_MAIN 路由表，其他值则操作 RT_TABLE_LOCAL 路由表。

（3）dst，路由表项的目的地址。

（4）dst_len，路由表项的目的地址的长度。

（5）ifa，添加路由项的相关信息，包括首选源地址和输出网络设备索引。

```
625 static void fib_magic(int cmd, int type, __be32 dst, int dst_len, struct
    in_ifaddr *ifa)
626 {
627     struct fib_table *tb;
628     struct fib_config cfg = {
629         .fc_protocol = RTPROT_KERNEL,
630         .fc_type = type,
631         .fc_dst = dst,
632         .fc_dst_len = dst_len,
633         .fc_prefsrc = ifa->ifa_local,
634         .fc_oif = ifa->ifa_dev->dev->ifindex,
635         .fc_nlflags = NLM_F_CREATE | NLM_F_APPEND,
636     };
637
638     if (type == RTN_UNICAST)
639         tb = fib_new_table(RT_TABLE_MAIN);
640     else
641         tb = fib_new_table(RT_TABLE_LOCAL);
642
```

```
643     if (tb == NULL)
644         return;
645
646     cfg.fc_table = tb->tb_id;
647
648     if (type != RTN_LOCAL)
649         cfg.fc_scope = RT_SCOPE_LINK;
650     else
651         cfg.fc_scope = RT_SCOPE_HOST;
652
653     if (cmd == RTM_NEWROUTE)
654         tb->tb_insert(tb, &cfg);
655     else
656         tb->tb_delete(tb, &cfg);
657 }
```

628-636 构成用于添加路由表项的信息。

638-643 根据路由类型确定操作的路由表，当路由类型为一条到单播地址的直连或非直连，则操作 RT_TABLE_MAIN 路由表，其他类型操作 RT_TABLE_LOCAL 路由表。

646-651 设置操作的路由表 ID 和路由范围。

653-656 根据命令添加或删除路由表项。

19.10 选路

19.10.1 输入选路：ip_route_input_slow()

对从网络设备输入的数据报进行路由，会调用 ip_route_input()进行选路。如果缓存中没有查找到匹配表项时，会调用 ip_route_input_slow()在路由表中进行查找，查找命中后，则将该表项添加到缓存中。

```
1908 static int ip_route_input_slow(struct sk_buff *skb, __be32 daddr, __be32
     saddr,
1909                 u8 tos, struct net_device *dev)
1910 {
1911     struct fib_result res;
1912     struct in_device *in_dev = in_dev_get(dev);
1913     struct flowi fl = { .nl_u = { .ip4_u =
1914                   { .daddr = daddr,
1915                     .saddr = saddr,
1916                     .tos = tos,
1917                     .scope = RT_SCOPE_UNIVERSE,
1918                   } },
1919             .mark = skb->mark,
1920             .iif = dev->ifindex };
1921     unsigned    flags = 0;
1922     u32         itag = 0;
1923     struct rtable * rth;
1924     unsigned    hash;
1925     __be32      spec_dst;
1926     int         err = -EINVAL;
1927     int         free_res = 0;
```

```
1928
1929    /* IP on this device is disabled. */
1930
1931    if (!in_dev)
1932        goto out;
1933
1934    /* Check for the most weird martians, which can be not detected
1935       by fib_lookup.
1936     */
1937
1938    if (MULTICAST(saddr) || BADCLASS(saddr) || LOOPBACK(saddr))
1939        goto martian_source;
1940
1941    if (daddr == htonl(0xFFFFFFFF) || (saddr == 0 && daddr == 0))
1942        goto brd_input;
1943
1944    /* Accept zero addresses only to limited broadcast;
1945     * I even do not know to fix it or not. Waiting for complains :-)
1946     */
1947    if (ZERONET(saddr))
1948        goto martian_source;
1949
1950    if (BADCLASS(daddr) || ZERONET(daddr) || LOOPBACK(daddr))
1951        goto martian_destination;
```

1913-1920 组织查找路由表项的条件。

1913-1920 根据源地址和目的地址以及 tos 构造 flowi 实例，用于查找路由项的条件。

1931-1932 校验与网络设备与 IP 特性的配置块的有效性。

1938-1951 校验源地址和目的地址的有效性，如目的地址不能为广播地址或回环地址等。

```
1953    /*
1954     *    Now we are ready to route packet.
1955     */
1956    if ((err = fib_lookup(&fl, &res)) != 0) {
1957        if (!IN_DEV_FORWARD(in_dev))
1958            goto e_hostunreach;
1959        goto no_route;
1960    }
1961    free_res = 1;
1962
1963    RT_CACHE_STAT_INC(in_slow_tot);
1964
1965    if (res.type == RTN_BROADCAST)
1966        goto brd_input;
1967
1968    if (res.type == RTN_LOCAL) {
1969        int result;
1970        result = fib_validate_source(saddr, daddr, tos,
1971                     loopback_dev.ifindex,
1972                     dev, &spec_dst, &itag);
1973        if (result < 0)
1974            goto martian_source;
1975        if (result)
1976            flags |= RTCF_DIRECTSRC;
1977        spec_dst = daddr;
```

```
1978        goto local_input;
1979    }
1980
1981    if (!IN_DEV_FORWARD(in_dev))
1982        goto e_hostunreach;
1983    if (res.type != RTN_UNICAST)
1984        goto martian_destination;
1985
1986    err = ip_mkroute_input(skb, &res, &fl, in_dev, daddr, saddr, tos);
1987    if (err == -ENOBUFS)
1988        goto e_nobufs;
1989    if (err == -EINVAL)
1990        goto e_inval;
```

1956-1961 通过 fib_lookup()在路由表中根据查找条件查找合适的表项。如果查找失败，并且禁止转发，返回 EHOSTUNREACH 发送主机没找到的错误，否则跳转到 no_route 处理。

1965-1966 目的地址为广播地址由 brd_input 处处理。

1968-1979 如果目的地址为本地接口的地址，需要检测源地址的有效性。检测通过后由 local_input()处处理。

1981-1982 如果系统禁止转发，且查找命中的表项的目的地址不为本地接口的地址，则返回 EHOSTUNREACH 发送主机没找到的错误。

1983-1984 此函数处理的目的地址为单播地址，因此查找的路由目的地址不为单播地址，则返回无效。

1986-1990 在对转发的数据报完成校验查找到的路由后，调用 ip_mkroute_input()创建路由缓存表项并添加到缓存中。

```
1992 done:
1993    in_dev_put(in_dev);
1994    if (free_res)
1995        fib_res_put(&res);
1996 out:    return err;
```

当正常查询结束后会从此处返回。

```
1998 brd_input:
1999    if (skb->protocol != htons(ETH_P_IP))
2000        goto e_inval;
2001
2002    if (ZERONET(saddr))
2003        spec_dst = inet_select_addr(dev, 0, RT_SCOPE_LINK);
2004    else {
2005        err = fib_validate_source(saddr, 0, tos, 0, dev, &spec_dst,
2006                    &itag);
2007        if (err < 0)
2008            goto martian_source;
2009        if (err)
2010            flags |= RTCF_DIRECTSRC;
2011    }
2012    flags |= RTCF_BROADCAST;
2013    res.type = RTN_BROADCAST;
2014    RT_CACHE_STAT_INC(in_brd);
```

1998-2014 处理目的地址为受限的广播地址 255.255.255.255，或者目的地址和源地址都为 0 的情况。

1999-2000 校验输入报文是否为 IP 数据报。

2002-2011 如果源地址为 0，则通过 inet_select_addr()选择合适的源地址，否则校验该源地址是否有效。根据校验结果，返回 EHOSTUNREACH 或设置该路由表项 RTCF_DIRECTSRC 标志。

2012-2013 给路由表项设置 RTCF_BROADCAST 标志，说明路由的目的地址是一个广播地址。

```
2016 local_input:
2017     rth = dst_alloc(&ipv4_dst_ops);
2018     if (!rth)
2019         goto e_nobufs;
2020
2021     rth->u.dst.output= ip_rt_bug;
2022
2023     atomic_set(&rth->u.dst.__refcnt, 1);
2024     rth->u.dst.flags= DST_HOST;
2025     if (in_dev->cnf.no_policy)
2026         rth->u.dst.flags |= DST_NOPOLICY;
2027     rth->fl.fl4_dst    = daddr;
2028     rth->rt_dst    = daddr;
2029     rth->fl.fl4_tos    = tos;
2030     rth->fl.mark    = skb->mark;
2031     rth->fl.fl4_src    = saddr;
2032     rth->rt_src    = saddr;
2033 #ifdef CONFIG_NET_CLS_ROUTE
2034     rth->u.dst.tclassid = itag;
2035 #endif
2036     rth->rt_iif    =
2037     rth->fl.iif    = dev->ifindex;
2038     rth->u.dst.dev    = &loopback_dev;
2039     dev_hold(rth->u.dst.dev);
2040     rth->idev·   = in_dev_get(rth->u.dst.dev);
2041     rth->rt_gateway    = daddr;
2042     rth->rt_spec_dst= spec_dst;
2043     rth->u.dst.input= ip_local_deliver;
2044     rth->rt_flags    = flags|RTCF_LOCAL;
2045     if (res.type == RTN_UNREACHABLE) {
2046         rth->u.dst.input= ip_error;
2047         rth->u.dst.error= -err;
2048         rth->rt_flags    &= ~RTCF_LOCAL;
2049     }
2050     rth->rt_type    = res.type;
2051     hash = rt_hash(daddr, saddr, fl.iif);
2052     err = rt_intern_hash(hash, rth, (struct rtable**)&skb->dst);
2053     goto done;
```

选路的目的地址为本地接口的地址创建路由缓存表项并添加到路由缓存中，参见 20.5 节。

```
2055 no_route:
2056     RT_CACHE_STAT_INC(in_no_route);
2057     spec_dst = inet_select_addr(dev, 0, RT_SCOPE_UNIVERSE);
```

```
2058     res.type = RTN_UNREACHABLE;
2059     goto local_input;
2088 }
```

根据 RT_SCOPE_UNIVERSE 范围选择地址作为路由的目的地址，然后转到 local_input 处，创建路由缓存表项。

```
2061     /*
2062      *    Do not cache martian addresses: they should be logged (RFC1812)
2063      */
2064 martian_destination:
2065     RT_CACHE_STAT_INC(in_martian_dst);
2066 #ifdef CONFIG_IP_ROUTE_VERBOSE
2067     if (IN_DEV_LOG_MARTIANS(in_dev) && net_ratelimit())
2068         printk(KERN_WARNING "martian destination %u.%u.%u.%u from "
2069             "%u.%u.%u.%u, dev %s\n",
2070             NIPQUAD(daddr), NIPQUAD(saddr), dev->name);
2071 #endif
2072
2073 e_hostunreach:
2074         err = -EHOSTUNREACH;
2075         goto done;
2076
2077 e_inval:
2078     err = -EINVAL;
2079     goto done;
2080
2081 e_nobufs:
2082     err = -ENOBUFS;
2083     goto done;
2084
2085 martian_source:
2086     ip_handle_martian_source(dev, in_dev, skb, daddr, saddr);
2087      goto e_inval;
```

选路失败时，返回相应错误码。

19.10.2 组播输入选路：ip_route_input_mc()

当输入的组播 IP 数据报在路由缓存中查找不到路由项时，会调用此函数进行组播数据报的选路。参数说明如下：

- skb，进行选路的组播数据报。
- daddr，组播数据报的目的地址。
- saddr，组播数据报的源地址。
- tos，组播数据报的 TOS。
- dev，输入组播数据报的网络设备。
- our，标识是否输入到本地。

```
1606 static int ip_route_input_mc(struct sk_buff *skb, __be32 daddr, __be32 saddr,
1607                u8 tos, struct net_device *dev, int our)
1608 {
1609     unsigned hash;
```

```
1610    struct rtable *rth;
1611    __be32 spec_dst;
1612    struct in_device *in_dev = in_dev_get(dev);
1613    u32 itag = 0;
1614
1615    /* Primary sanity checks. */
1616
1617    if (in_dev == NULL)
1618       return -EINVAL;
1619
1620    if (MULTICAST(saddr) || BADCLASS(saddr) || LOOPBACK(saddr) ||
1621       skb->protocol != htons(ETH_P_IP))
1622       goto e_inval;
1623
1624    if (ZERONET(saddr)) {
1625       if (!LOCAL_MCAST(daddr))
1626          goto e_inval;
1627       spec_dst = inet_select_addr(dev, 0, RT_SCOPE_LINK);
1628    } else if (fib_validate_source(saddr, 0, tos, 0,
1629                   dev, &spec_dst, &itag) < 0)
1630       goto e_inval;
1631
1632    rth = dst_alloc(&ipv4_dst_ops);
1633    if (!rth)
1634       goto e_nobufs;
1635
1636    rth->u.dst.output= ip_rt_bug;
1637
1638    atomic_set(&rth->u.dst.__refcnt, 1);
1639    rth->u.dst.flags= DST_HOST;
1640    if (in_dev->cnf.no_policy)
1641       rth->u.dst.flags |= DST_NOPOLICY;
1642    rth->fl.fl4_dst    = daddr;
1643    rth->rt_dst    = daddr;
1644    rth->fl.fl4_tos    = tos;
1645    rth->fl.mark    = skb->mark;
1646    rth->fl.fl4_src    = saddr;
1647    rth->rt_src    = saddr;
1648 #ifdef CONFIG_NET_CLS_ROUTE
1649    rth->u.dst.tclassid = itag;
1650 #endif
1651    rth->rt_iif    =
1652    rth->fl.iif    = dev->ifindex;
1653    rth->u.dst.dev    = &loopback_dev;
1654    dev_hold(rth->u.dst.dev);
1655    rth->idev    = in_dev_get(rth->u.dst.dev);
1656    rth->fl.oif    = 0;
1657    rth->rt_gateway    = daddr;
1658    rth->rt_spec_dst= spec_dst;
1659    rth->rt_type    = RTN_MULTICAST;
1660    rth->rt_flags    = RTCF_MULTICAST;
1661    if (our) {
1662       rth->u.dst.input= ip_local_deliver;
1663       rth->rt_flags |= RTCF_LOCAL;
1664    }
1665
```

```
1666 #ifdef CONFIG_IP_MROUTE
1667     if (!LOCAL_MCAST(daddr) && IN_DEV_MFORWARD(in_dev))
1668         rth->u.dst.input = ip_mr_input;
1669 #endif
1670     RT_CACHE_STAT_INC(in_slow_mc);
1671
1672     in_dev_put(in_dev);
1673     hash = rt_hash(daddr, saddr, dev->ifindex);
1674     return rt_intern_hash(hash, rth, (struct rtable**) &skb->dst);
1675
1676 e_nobufs:
1677     in_dev_put(in_dev);
1678     return -ENOBUFS;
1679
1680 e_inval:
1681     in_dev_put(in_dev);
1682     return -EINVAL;
1683 }
```

1617-1618 校验输入网络设备 IP 配置块是否有效。

1620-1622 输入组播数据报的源地址不能是组播地址、广播地址和回环地址，并且数据报必须是 IP 数据报。

1624-1627 校验源地址和目的地址，对于网络号为 0 的源地址，目的地址不能是本地组播地址。校验通过后，选择首选源地址作为待生成路由项的目的地址。

1628-1630 对于网络号不为 0 的源地址，则通过 fib_validate_source()来验证源地址是否有效。

1632-1660 通过对源地址的校验后，创建新的路由表项并设置相应的值。

1661-1664 如果是输入到本地的组播地址，则将数据报输入处理函数设置为 ip_local_deliver()。

1666-1667 启用了 IP 组播之后，如果目的地址不是本地组播地址且系统运行转发组播数据报，则将数据报输入处理函数设置为 ip_mr_input()。

1672-1674 新创建的路由表项值设置完成后，便将其添加到路由缓存中。

1676-1682 分配路由表项失败或校验源地址失败在此处处理。

19.10.3 输出选路：ip_route_output_slow()

对本地生成数据报进行路由，会调用 ip_route_output_slow()进行选路，然后又调用到了__ip_route_output_key()，如果缓存中没有查找到匹配表项时；会调用 ip_route_output_slow()在路由表中进行查找，并在查找到匹配的表项后，将表项添加到缓存中。

```
2365 static int ip_route_output_slow(struct rtable **rp, const struct flowi
     *oldflp)
2366 {
2367     u32 tos    = RT_FL_TOS(oldflp);
2368     struct flowi fl = { .nl_u = { .ip4_u =
2369                   { .daddr = oldflp->fl4_dst,
2370                 .saddr = oldflp->fl4_src,
2371                 .tos = tos & IPTOS_RT_MASK,
2372                 .scope = ((tos & RTO_ONLINK) ?
2373                       RT_SCOPE_LINK :
```

```
2374                        RT_SCOPE_UNIVERSE),
2375                    } },
2376                .mark = oldflp->mark,
2377                .iif = loopback_dev.ifindex,
2378                .oif = oldflp->oif };
2379    struct fib_result res;
2380    unsigned flags = 0;
2381    struct net_device *dev_out = NULL;
2382    int free_res = 0;
2383    int err;
2384
2385
2386    res.fi       = NULL;
2387 #ifdef CONFIG_IP_MULTIPLE_TABLES
2388    res.r        = NULL;
2389 #endif
```

2367-2378 根据在缓存中查找的条件 oldflp 初始化在路由表中查找的条件 fl，作为之后调用 fib_lookup()的参数。

源地址、目的地址和防火墙标记是直接从函数的输入参数复制而来的，而源设备则被初始化为回环设备，因为 ip_route_output_slow()只是路由本地生成的包。

由于 TOS 字段不需要占用整个八位，因此可将 flags 存储到 fl4_tos 字段的两个最低位中，这样 ip_route_output_slow()可以使用该 flags 来确定待搜索路由项的范围，参见 RF_FL_TOS 宏。

```
#define RT_FL_TOS(oldflp) \
    ((u32)(oldflp->fl4_tos & (IPTOS_RT_MASK | RTO_ONLINK)))
```

当 RTO_ONLINK 标志被设置时，设置待搜索的路由项的范围为 RT_SCOPE_LINK，否则设置为 RT_SCOPE_UNIVERSE。

```
2391    if (oldflp->fl4_src) {
2392        err = -EINVAL;
2393        if (MULTICAST(oldflp->fl4_src) ||
2394            BADCLASS(oldflp->fl4_src) ||
2395            ZERONET(oldflp->fl4_src))
2396            goto out;
2397
2398        /* It is equivalent to inet_addr_type(saddr) == RTN_LOCAL */
2399        dev_out = ip_dev_find(oldflp->fl4_src);
2400        if (dev_out == NULL)
2401            goto out;
2402
2403        /* I removed check for oif == dev_out->oif here.
2404           It was wrong for two reasons:
2405           1. ip_dev_find(saddr) can return wrong iface, if saddr is
2406              assigned to multiple interfaces.
2407           2. Moreover, we are allowed to send packets with saddr
2408              of another iface. --ANK
2409         */
2410
2411        if (oldflp->oif == 0
2412                && (MULTICAST(oldflp->fl4_dst) || oldflp->fl4_dst == htonl
```

```
                (0xFFFFFFFF))) {
2413            /* Special hack: user can direct multicasts
2414               and limited broadcast via necessary interface
2415               without fiddling with IP_MULTICAST_IF or IP_PKTINFO.
2416               This hack is not just for fun, it allows
2417               vic,vat and friends to work.
2418               They bind socket to loopback, set ttl to zero
2419               and expect that it will work.
2420               From the viewpoint of routing cache they are broken,
2421               because we are not allowed to build multicast path
2422               with loopback source addr (look, routing cache
2423               cannot know, that ttl is zero, so that packet
2424               will not leave this host and route is valid).
2425               Luckily, this hack is good workaround.
2426             */
2427
2428            fl.oif = dev_out->ifindex;
2429            goto make_route;
2430        }
2431        if (dev_out)
2432            dev_put(dev_out);
2433        dev_out = NULL;
2434    }
```

2391-2434 在源地址已知的情况下，对该源地址以及与该源地址对应网络设备进行校验。

2393-2396 源地址不能为广播地址、组播地址或为0。

2399-2401 检测根据该源地址获取对应设备是否有效。

2411-2430 如果key中没有设定输出网络设备，并且目的地址为组播地址或广播地址，则将根据源地址获取到的设备作为输出网络设备。在这种情况下，可以进行创建路由缓存了。

```
2437    if (oldflp->oif) {
2438        dev_out = dev_get_by_index(oldflp->oif);
2439        err = -ENODEV;
2440        if (dev_out == NULL)
2441            goto out;
2442
2443        /* RACE: Check return value of inet_select_addr instead. */
2444        if (__in_dev_get_rtnl(dev_out) == NULL) {
2445            dev_put(dev_out);
2446            goto out;    /* Wrong error code */
2447        }
2448
2449        if(LOCAL_MCAST(oldflp->fl4_dst)||oldflp->fl4_dst==htonl(0xFFFFFFFF)){
2450            if (!fl.fl4_src)
2451                fl.fl4_src = inet_select_addr(dev_out, 0,
2452                                              RT_SCOPE_LINK);
2453            goto make_route;
2454        }
2455        if (!fl.fl4_src) {
2456            if (MULTICAST(oldflp->fl4_dst))
2457                fl.fl4_src = inet_select_addr(dev_out, 0,
2458                                              fl.fl4_scope);
2459            else if (!oldflp->fl4_dst)
2460                fl.fl4_src = inet_select_addr(dev_out, 0,
```

```
2461                                    RT_SCOPE_HOST);
2462            }
2463        }
```

2437-2463 在输出网络设备已知的情况下，对该网络设备进行校验，并获取源地址。

2438-2441 根据给定的输出网络设备 ID 获取网络设备。

2444-2447 检测该输出网络设备的 IPv4 配置块是否有效。

2449-2462 当搜索的条件 fl 没有提供源地址时，则通过 inet_select_addr()的输入参数来选择一个源 IP 地址。但是需要根据目的地址不同的类型，inet_select_addr()获取源地址的范围也不同，比如目的地址为本地组播地址或广播地址，则类型为 RT_SCOPE_LINK，而且完成后创建路由缓存。

```
2465        if (!fl.fl4_dst) {
2466            fl.fl4_dst = fl.fl4_src;
2467            if (!fl.fl4_dst)
2468                fl.fl4_dst = fl.fl4_src = htonl(INADDR_LOOPBACK);
2469            if (dev_out)
2470                dev_put(dev_out);
2471            dev_out = &loopback_dev;
2472            dev_hold(dev_out);
2473            fl.oif = loopback_dev.ifindex;
2474            res.type = RTN_LOCAL;
2475            flags |= RTCF_LOCAL;
2476            goto make_route;
2477        }
```

目的地址未知的情况下将源地址设置为目的地址。如果目的地址和源地址都未设置，则使用回环地址作为目的地址和源地址。输出网络设备设置为回环网络设备，同时设置路由表项类型为 RTN_LOCAL，完成设置后进行创建路由缓存。

```
2479        if (fib_lookup(&fl, &res)) {
2480            res.fi = NULL;
2481            if (oldflp->oif) {
2482                /* Apparently, routing tables are wrong. Assume,
2483                   that the destination is on link.
2484
2485                   WHY? DW.
2486                   Because we are allowed to send to iface
2487                   even if it has NO routes and NO assigned
2488                   addresses. When oif is specified, routing
2489                   tables are looked up with only one purpose:
2490                   to catch if destination is gatewayed, rather than
2491                   direct. Moreover, if MSG_DONTROUTE is set,
2492                   we send packet, ignoring both routing tables
2493                   and ifaddr state. --ANK
2494
2495
2496                   We could make it even if oif is unknown,
2497                   likely IPv6, but we do not.
2498                 */
2499
2500                if (fl.fl4_src == 0)
```

```
2501                fl.fl4_src = inet_select_addr(dev_out, 0,
2502                                              RT_SCOPE_LINK);
2503            res.type = RTN_UNICAST;
2504            goto make_route;
2505        }
2506        if (dev_out)
2507            dev_put(dev_out);
2508        err = -ENETUNREACH;
2509        goto out;
2510    }
2511    free_res = 1;
```

通过 fib_lookup()在路由表中查找合适的路由表项。

查找失败，但输出的数据报确定了输出网络设备，在这种情况下，即使路由查找失败，但确信目的地址是有效的，因为当指定了输出网络设备 ID，查找路由只有一个目的，那就是确定目的地址是经过网关的，而不是直连的。另外，如果设置 MSG_DONTROUTE，在发送数据报时会忽略路由表。

```
2513    if (res.type == RTN_LOCAL) {
2514        if (!fl.fl4_src)
2515            fl.fl4_src = fl.fl4_dst;
2516        if (dev_out)
2517            dev_put(dev_out);
2518        dev_out = &loopback_dev;
2519        dev_hold(dev_out);
2520        fl.oif = dev_out->ifindex;
2521        if (res.fi)
2522            fib_info_put(res.fi);
2523        res.fi = NULL;
2524        flags |= RTCF_LOCAL;
2525        goto make_route;
2526    }
```

当 fib_lookup 查找的数据报的目的地址是本地地址，或者当数据报中没有提供目的地址时（即搜索包含了未知地址 0.0.0.0），该数据报被送往本地。

在这种情况下，输出网络设备被设置为回环设备。这表示该数据报不会离开本地主机，该数据报被发送出后，将重新回到 IP 输入栈。而 dst->input 被初始化为 ip_local_deliver()。正是由于这一操作，当数据报重新回到 IP 协议栈时，ip_rcv_finish()调用 dst_input()，即调用 ip_local_deliver()来处理该数据报。

当搜索 key 中源 IP 地址和目的 IP 地址都没有被设置时，数据报被送往本地，源地址和目的地址被设置为默认回环地址 127.0.0.1（INADDR_LOOPBACK），该地址 scope 为 RT_SCOPE_HOST。

```
2528 #ifdef CONFIG_IP_ROUTE_MULTIPATH
2529    if (res.fi->fib_nhs > 1 && fl.oif == 0)
2530        fib_select_multipath(&fl, &res);
2531    else
2532 #endif
```

在启用了多路径路由时，选择路由，本书不作论述。

```
2533     if (!res.prefixlen && res.type == RTN_UNICAST && !fl.oif)
2534        fib_select_default(&fl, &res);
2535
2536     if (!fl.fl4_src)
2537        fl.fl4_src = FIB_RES_PREFSRC(res);
2538
2539     if (dev_out)
2540        dev_put(dev_out);
2541     dev_out = FIB_RES_DEV(res);
2542     dev_hold(dev_out);
2543     fl.oif = dev_out->ifindex;
```

2533-2534 当查找返回的路由是默认路由时，需要选择使用的默认网关。这由 fib_select_default()来执行（当 res.prefixlen 字段为 0 时表示是默认路由，这表示“前缀长度”，即与该地址相关的网络掩码长度为 0）。

2539-2543 即使 fib_lookup()查找路由失败，但还是有可能成功地将数据报发送出去。当搜索 key 提供了输出网络设备，ip_route_output_slow()假定通过该输出网络设备可以直接到达目的地。这时，如果还没有源 IP 地址，则还需要设置一个作用范围为 RT_SCOPE_LINK 的源 IP 地址，可能的情况下用的是该输出网络设备上的一个地址。

```
2546 make_route:
2547     err = ip_mkroute_output(rp, &res, &fl, oldflp, dev_out, flags);
2548
2549
2550     if (free_res)
2551        fib_res_put(&res);
2552     if (dev_out)
2553        dev_put(dev_out);
2554 out:    return err;
2555 }
```

最后创建缓存表项并添加到路由缓存中，这由 ip_mkroute_output()来执行。

19.10.4 fib_lookup()

fib_lookup()用于搜索路由表，但这个函数有两个版本，一个是当内核支持策略路由时使用，另一个则在不支持时使用，在编译时便决定了选择哪一个函数，因此当 ip_route_input_slow()和 ip_route_output_slow()调用 fib_lookup()时，可以调用到确定的查找函数。

先介绍不支持策略路由时使用的 fib_lookup()。参数说明如下：

- flp，搜索路由表项的条件，参见 20.2.2 节。
- rest，如果搜索成功，将返回的结果保存到参数 rest 中。

```
101 struct fib_result {
102     unsigned char    prefixlen;
103     unsigned char    nh_sel;
104     unsigned char    type;
105     unsigned char    scope;
106 #ifdef CONFIG_IP_ROUTE_MULTIPATH_CACHED
107     __be32           network;
108     __be32           netmask;
109 #endif
```

```
110     struct fib_info *fi;
111 #ifdef CONFIG_IP_MULTIPLE_TABLES
112     struct fib_rule    *r;
113 #endif
114 };
```

102 unsigned char prefixlen

返回路由表项的网络掩码长度。

103 unsigned char nh_sel

返回选择路径的序号，通常为0。当支持多路径路由时，才可能大于0。

104 unsigned char type

返回路由表项的类型。

105 unsigned char scope

返回路由表项的作用范围。

107 __be32 network

108 __be32 netmask

用于支持多路径路由，本书不作论述。

110 struct fib_info *fi

返回查找到的路由信息。

112 struct fib_rule *r

支持策略路由时，查找到的路由策略。

```
189 static inline int fib_lookup(const struct flowi *flp, struct fib_result *res)
190 {
191     if (ip_fib_local_table->tb_lookup(ip_fib_local_table, flp, res) &&
192         ip_fib_main_table->tb_lookup(ip_fib_main_table, flp, res))
193         return -ENETUNREACH;
194     return 0;
195 }
```

搜索 ip_fib_local_table 路由表，如果失败则再搜索 ip_fib_main_table 路由表。如果两张表都没有查找到，fib_loopkup 返回-ENETUNREACH（目的网络不可达）。

以下 fib_lookup()在支持策略路由时使用，具体由 fib_rules_lookup 来实现，参见 21.6 节。

```
88 int fib_lookup(struct flowi *flp, struct fib_result *res)
89 {
90     struct fib_lookup_arg arg = {
91         .result = res,
92     };
93     int err;
94
95     err = fib_rules_lookup(&fib4_rules_ops, flp, 0, &arg);
96     res->r = arg.rule;
97
98     return err;
99 }
```

19.10.5 fn_hash_lookup()

根据查找条件在指定的路由表中查找符合条件的路由表项，然后返回查找结果。参数说明如下：

- tb，待查找的路由表；
- flp，查找条件，参见 20.2.2 节；
- res，查找路由表项的返回结果。

```
245 static int
246 fn_hash_lookup(struct fib_table *tb, const struct flowi *flp, struct fib_result
    *res)
247 {
248     int err;
249     struct fn_zone *fz;
250     struct fn_hash *t = (struct fn_hash*)tb->tb_data;
251
252     read_lock(&fib_hash_lock);
253     for (fz = t->fn_zone_list; fz; fz = fz->fz_next) {
254         struct hlist_head *head;
255         struct hlist_node *node;
256         struct fib_node *f;
257         __be32 k = fz_key(flp->fl4_dst, fz);
258
259         head = &fz->fz_hash[fn_hash(k, fz)];
260         hlist_for_each_entry(f, node, head, fn_hash) {
261             if (f->fn_key != k)
262                 continue;
263
264             err = fib_semantic_match(&f->fn_alias,
265                         flp, res,
266                         f->fn_key, fz->fz_mask,
267                         fz->fz_order);
268             if (err <= 0)
269                 goto out;
270         }
271     }
272     err = 1;
273 out:
274     read_unlock(&fib_hash_lock);
275     return err;
276 }
```

250 根据路由表得到路由表项散列表(tb->tb_data)。

253-271 遍历所有的路由表项的散列表查找相同网段的路由表项。然后调用 fib_semantic_match()将这些路由表项与查找条件进行匹配，最后生成查询结果。

19.11 ICMP 重定向消息的发送

在为转发的数据报查询路由时，当发现该路由不是最优，则会在该路由表项上加上 RTCF_DOREDIRECT 标志，然后在转发该数据报时会调用 ip_rt_send_redirect()，向该数据报

的发送方发送 ICMP 重定向消息。有关 RTCF_DOREDIRECT 标志的添加和转发数据报参见 11.10.2 节。

```
1300 void ip_rt_send_redirect(struct sk_buff *skb)
1301 {
1302     struct rtable *rt = (struct rtable*)skb->dst;
1303     struct in_device *in_dev = in_dev_get(rt->u.dst.dev);
1304
1305     if (!in_dev)
1306         return;
1307
1308     if (!IN_DEV_TX_REDIRECTS(in_dev))
1309         goto out;
1310
1311     /* No redirected packets during ip_rt_redirect_silence;
1312      * reset the algorithm.
1313      */
1314     if (time_after(jiffies, rt->u.dst.rate_last + ip_rt_redirect_silence))
1315         rt->u.dst.rate_tokens = 0;
1316
1317     /* Too many ignored redirects; do not send anything
1318      * set u.dst.rate_last to the last seen redirected packet.
1319      */
1320     if (rt->u.dst.rate_tokens >= ip_rt_redirect_number) {
1321         rt->u.dst.rate_last = jiffies;
1322         goto out;
1323     }
1324
1325     /* Check for load limit; set rate_last to the latest sent
1326      * redirect.
1327      */
1328     if (rt->u.dst.rate_tokens == 0 ||
1329         time_after(jiffies,
1330               (rt->u.dst.rate_last +
1331             (ip_rt_redirect_load << rt->u.dst.rate_tokens))) {
1332         icmp_send(skb, ICMP_REDIRECT, ICMP_REDIR_HOST, rt->rt_gateway);
1333         rt->u.dst.rate_last = jiffies;
1334         ++rt->u.dst.rate_tokens;
1335 #ifdef CONFIG_IP_ROUTE_VERBOSE
1336         if (IN_DEV_LOG_MARTIANS(in_dev) &&
1337             rt->u.dst.rate_tokens == ip_rt_redirect_number &&
1338             net_ratelimit())
1339             printk(KERN_WARNING "host %u.%u.%u.%u/if%d ignores "
1340                 "redirects for %u.%u.%u.%u to %u.%u.%u.%u.\n",
1341                 NIPQUAD(rt->rt_src), rt->rt_iif,
1342                 NIPQUAD(rt->rt_dst), NIPQUAD(rt->rt_gateway));
1343 #endif
1344     }
1345 out:
1346         in_dev_put(in_dev);
1347 }
```

1308-1309 如果系统禁止发 ICMP 重定向消息，则不再继续处理。

1314-1315 自从上次发送 ICMP 重定向消息到此次输入数据报触发内核生成 ICMP 重定向消息的时间间隔超过了 ip_rt_redirect_silence 秒，则需要对 rate_tokens 清零。

1320-1323 如果由于目的地持续忽略 ICMP 重定向消息，从而持续（间隔小于 ip_rt_redirect_silence 秒）发送 ICMP 重定向消息数目到达 ip_rt_redirect_number，从而取消此次的发送，实现对 ICMP 重定向消息发送的限速。

1328-1344 如果还未发送过 ICMP 重定向消息，或者与上次发送 ICMP 重定向消息的间隔达到规定时间，则允许继续发送 ICMP 重定向消息。关于这个规定时间，用了一个很简单的指数回退算法，即每发送一个消息就翻倍间隔时间。

1335-1343 有条件地记录 ICMP 重定向信息。