UNDERSTANDING LINUX NETWORK INTERNALS

深入理解 LINUX 网络技术内幕

CHRISTIAN BENVENUTI 著
夏安 闫江毓 黄景昌 译

O'REILLY®
中国电力出版社

GUIDED TOUR TO NETWORKING ON LINUX

深入理解LINUX网络技术内幕

Linux如此流行正是得益于它的特性丰富并且有效的网络协议栈。如果你曾经惊叹于Linux能够实现如此复杂的工作，或是你只是想通过现实中的例子学习现代网络，《深入理解Linux网络技术内幕》将会给你指导。

同其他O'Reilly的畅销书籍一样，《深入理解Linux网络技术内幕》清楚地阐述了网络的基本概念，并指导你如何用C语言实现。虽然早先的TCP/IP经验是有用的，初学者通过本书仍然可以学习到协议本身和大量的应用信息。一旦彻底掌握了这些网络工具，你就可以使用本书所附的代码，准确地指出Linux内核中最重要的部分如何工作。

网络功能的实现分布在不同的时间由不同的代码块完成，这正是理解网络和实现它的难点之一。本书的成功之一就是描述这些代码块如何集成，以及众多的函数和数据结构之间的关系。《深入理解Linux网络技术内幕》不仅描述了Linux网络的全貌，而且是理解Linux网络细节的有效指南。

作者**Christian Benvenuti**是一个专注于操作系统网络的设计者。他在书中不仅解释了Linux代码如何工作，还阐述了主要网络特性的目的，以及在不同解决方案间是如何抉择的。书中还包含了大量的流程图和其他图表以帮助读者理解。

书中的主题包含：

- 网络通信的主要问题
- 网络接口设备（NIC）驱动
- 系统初始化
- 第二层（链路层）的功能和实现
- 第三层（IP）的功能和实现
- ARP
- 桥接
- 路由
- ICMP

O'REILLY®

www.oreilly.com

O'Reilly Media, Inc.授权中国电力出版社出版

此简体中文版仅限于在中华人民共和国境内（但不允许在中国香港、澳门特别行政区和中国台湾地区）销售发行

This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-5083-7964-7

9 787508 379647 >

定价：128.00 元

深入理解 LINUX 网络技术内幕

Christian Benvenuti 著
夏安 闫江毓 黄景昌 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

O'Reilly Media, Inc. 授权中国电力出版社出版

中国电力出版社

图书在版编目（CIP）数据

深入理解Linux网络技术内幕 /（意）本（Benvenuti, C.）著；夏安，闫江毓，黄景昌译. —北京：中国电力出版社，2009

书名原文：Understanding Linux Network Internals

ISBN 978-7-5083-7964-7

I. 深 ... II. ①本 ... ②夏 ... ③闫 ... ④黄 ... III. Linux操作系统 IV. TP316.89

中国版本图书馆CIP数据核字（2008）第178408号

北京市版权局著作权合同登记

图字：01-2006-4879号

©2006 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and China Electric Power Press, 2007. Authorized translation of the English edition, 2006 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由O'Reilly Media, Inc. 出版2006。

简体中文版由中国电力出版社出版2007。英文原版的翻译得到O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

书　　名/ 深入理解Linux网络技术内幕

书　　号/ ISBN 978-7-5083-7964-7

责任编辑/ 白立军，牛贵华

封面设计/ Karen Montgomery，张健

出版发行/ 中国电力出版社（www.infopower.com.cn）

地　　址/ 北京三里河路6号（邮政编码100044）

经　　销/ 全国新华书店

印　　刷/ 北京丰源印刷厂

开　　本/ 787毫米×980毫米　16开本　63.75印张　1401千字

版　　次/ 2009年6月第1版　2009年6月第1次印刷

印　　数/ 0001 – 4000册

定　　价/ 128.00元（册）

敬告读者

本书封面贴有防伪标签，加热后中心图案消失。

本书如有印装质量问题，我社发行部负责退换。

版权所有　翻印必究

O'Reilly Media, Inc. 介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构O'Reilly Media, Inc. 授权中国电力出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc. 是世界上在 Unix、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时也是联机出版的先锋。

从最畅销的 *The Whole Internet User's Guide & Catalog*（被纽约公共图书馆评为20世纪最重要的50本书之一）到GNN（最早的Internet门户和商业网站），再到WebSite（第一个桌面PC的Web服务器软件），O'Reilly Media, Inc. 一直处于Internet发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc. 是最稳定的计算机图书出版商 —— 每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc. 具有深厚的计算机专业背景，这使得 O'Reilly Media, Inc. 形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc. 所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc. 还有许多固定的作者群体 —— 他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc. 依靠他们及时地推出图书。因为 O'Reilly Media, Inc. 紧密地与计算机业界联系着，所以 O'Reilly Media, Inc. 知道市场上真正需要什么图书。

目录

前言 1

第一部分 基础背景

第一章 简介 13

基本术语 13

常见编码模式 14

用户空间工具 28

浏览源代码 29

当功能以补丁形式提供时 30

第二章 关键数据结构 32

套接字缓冲区：sk_buff 结构 32

net_device 结构 53

本章涉及的文件 67

第三章 用户空间与内核的接口 68
概论 68
procfs 与 sysctl 70
ioctl 77
Netlink 80
配置改变串行化 81

第二部分 系统初始化

第四章 通知链 85
使用通知链的原因 85
概论 87
定义链 88
链注册 88
链上的通知事件 89
网络子系统的通知链 91
通过 /proc 文件系统的调整 92
本章涉及的函数和变量 92
本章涉及的文件和目录 93

第五章 网络设备初始化 94
系统初始化概论 94
设备注册和初始化 96
NIC 初始化的基本目标 96
设备与内核之间的交互 97
初始化选项 102
模块选项 103
设备处理层初始化：net_dev_init 104
用户空间辅助程序 106

虚拟设备 110
通过 /proc 文件系统调整 112
本章涉及的函数和变量 113
本章涉及的文件和目录 114

第六章 PCI 层和网络接口卡 115

本章涉及的数据结构 115
PCI NIC 设备驱动程序的注册 117
电源管理和网络唤醒 118
PCI NIC 驱动程序注册范例 119
大蓝图 121
通过 /proc 文件系统调整 123
本章涉及的函数和变量 123
本章涉及的文件和目录 123

第七章 组件初始化的内核基础架构 124

引导期间的内核选项 124
模块初始化代码 130
优化宏卷标 133
引导期间初始化函数 136
内存最优化 138
通过 /proc 文件系统调整 142
本章涉及的函数和变量 142
本章涉及的文件和目录 143

第八章 设备注册和初始化 144

设备注册之时 145
设备除名之时 145
分配 net_device 结构 146
NIC 注册和除名的架构 147

设备初始化 149
net_device 结构的组织 153
设备状态 155
设备的注册和除名 157
设备注册 162
设备除名 164
开启和关闭网络设备 167
更新设备队列规则状态 169
从用户空间配置设备相关信息 174
虚拟设备 177
上锁 179
通过 /proc 文件系统调整 179
本章涉及的函数和变量 179
本章涉及的文件和目录 181

第三部分 传输和接收

第九章 中断和网络驱动程序 185
决策和流量方向 186
接收到帧时通知驱动程序 186
中断处理函数 191
softnet_data 结构 213

第十章 帧的接收 217
与其他功能交互 218
设备的开启和关闭 218
队列 219
通知内核帧已接收：NAPI 和 netif_rx 219
设备驱动程序与内核间的旧接口：netif_rx 的第一部分 225
拥塞管理 231

处理 NET_RX_SOFTIRQ：net_rx_action 234

第十一章 帧的传输 245

传输的开启和关闭 247

第十二章 中断事件一般性参考数据 267

统计数据 267

通过 /proc 和 sysfs 文件系统调整 268

本部分涉及的函数和变量 269

本部分涉及的文件和目录 271

第十三章 协议处理函数 272

网络协议栈概论 272

执行正确的协议处理函数 280

协议处理函数的组织 284

协议处理函数的注册 285

Ethernet 与 IEEE 802.3 帧 287

通过 /proc 文件系统进行调整 299

本章涉及的函数和变量 299

本章涉及的文件和目录 300

第四部分 桥接

第十四章 桥接：概念 303

中继器、网桥以及路由器 303

网桥与交换机 305

主机 306

用网桥合并 LAN 306

桥接不同技术的 LAN 307

地址学习 308
多台网桥 310

第十五章 桥接：生成树协议 316
基本术语 316
阶层交换式 L2 拓扑范例 318
生成树协议的基本元素 321
网桥 ID 和端口 ID 326
网桥协议数据单元（BPDU） 328
定义活跃拓扑 332
定时器 340
拓扑变化 344
BPDU 封装 350
传输配置 BPDU 350
处理入口帧 352
收敛时间 354
新版生成树协议概况 355

第十六章 桥接：Linux 实现 360
网桥设备抽象 360
重要的数据结构 364
桥接程序的初始化 365
建立网桥设备和网桥端口 366
建立一个新网桥设备 367
网桥设备设置函数 368
删除网桥 369
给网桥添加端口 369
启动和关闭网桥设备 371
启动和关闭网桥端口 373
改变网桥端口状态 374

大蓝图 375
转发数据库 377
处理入口流量 380
网桥设备上的传输 383
生成树协议（STP） 385
netdevice 通知链 394

第十七章 桥接：其他主题 395
用户空间配置工具 395
通过 /proc 文件系统调整网桥 400
通过 /sys 文件系统调整网桥 400
统计数据 402
本部分涉及的数据结构 402
本部分涉及的函数和变量 407
本部分涉及的文件和目录 409

第五部分 因特网协议第 4 版（IPv4）

第十八章 因特网协议第 4 版（IPv4）：概念 413
IP 协议：大蓝图 413
IP 报头 415
IP 选项 418
封包的分段 / 重组 424
校验和 435

第十九章 因特网协议第 4 版（IPv4）：Linux 的原理和功能 442
主要 IPv4 数据结构 442
封包的一般性处理 446

IP 选项 456

第二十章 因特网协议第 4 版（IPv4）：转发和本地传递 469

转发 469

本地传递 475

第二十一章 因特网协议第 4 版（IPv4）：传输 476

进行传输的重要函数 477

衔接邻居子系统 512

第二十二章 因特网协议第 4 版（IPv4）：处理分段 ... 515

IP 分段 516

IP 重组 525

第二十三章 因特网协议第 4 版（IPv4）：其他主题 539

长效 IP 端点信息 539

选择 IP 报头的 ID 字段 543

IP 统计资料 544

IP 配置 548

IP-over-IP 553

IPv4：问题出在哪儿？ 553

通过 /proc 文件系统做调整 555

本部分涉及的数据结构 558

本部分涉及的函数和变量 568

本部分涉及的文件和目录 571

第二十四章 L4 协议和 Raw IP 的处理 572

可用的 L4 协议 572

L4 协议的注册 574
L3 到 L4 的传递：ip_local_deliver_finish 577
IPv4 vs IPv6 585
通过 /proc 文件系统做调整 585
本章涉及的函数和变量 585
本章涉及的文件和目录 586

第二十五章 因特网控制消息协议（ICMPv4） 587
ICMP 报头 588
ICMP 有效载荷 589
ICMP 类型 590
ICMP 协议的应用程序 597
大蓝图 598
协议的初始化 602
本章涉及的数据结构 602
传输 ICMP 消息 605
接收 ICMP 消息 613
ICMP 统计资料 619
把错误通知传给传输层 622
通过 /proc 文件系统调整 623
本章涉及的函数和变量 624
本章涉及的文件和目录 625

第六部分 邻居子系统

第二十六章 邻居子系统：概念 629
邻居是什么？ 629
需要邻居协议的原因 632
Linux 实现 637
邻居协议代理 640

何时发送和处理 Solicitation 请求 643
邻居状态和网络不可到达性探测（NUD） 645

第二十七章 邻居子系统：基础结构 654

主要的数据结构 654
L3 协议和邻居协议间的通用接口 658
邻居基础结构的一般任务 668
邻居结构的引用计数 673
创建一个邻居项 673
删除邻居 676
担任代理 682
L2 帧头缓存 686
协议初始化和清理 690
与其他子系统的交互 691
邻居协议和 L3 传输函数的交互 695
排队 696
入口队列 698

第二十八章 邻居子系统：地址解析协议（ARP） 702

ARP 封包格式 703
ARP 事务范例 705
无端 ARP 705
多个网络接口的应答 709
调整 ARP 选项 711
初始化 ARP 协议 718
初始化 neighbour 结构 720
传输和接收 ARP 封包 725
处理入口 ARP 封包 730
ARP 代理 738
实例 743

外部事件 746
ARPD 747
逆向地址解析协议（RARP） 750
ND（IPv6）对 ARP（IPv4）的改进 750

第二十九章 邻居子系统：其他问题 752
邻居的系统管理 752
使用 /proc 文件系统调整邻居性能 755
本部分涉及的数据结构 761
本部分涉及的文件和目录 778

第七部分 路由

第三十章 路由：概念 781
路由器、路由与路由表 782
路由的基本元素 786
路由表 797
查找 801
封包接收与封包传输 803

第三十一章 路由：高级路由 806
策略路由背后的概念 806
多路径路由背后的概念 812
与其他内核子系统的交互 818
路由协议守护进程 822
Verbose 监控 824
ICMP_REDIRECT 消息 825
反向路径过滤 831

第三十二章 路由：Linux的实现 833

内核选项 833

主要的数据结构 836

路由 scope 与地址 scope 841

主 IP 地址与辅助 IP 地址 843

常用的辅助程序与宏 844

全局锁 846

路由子系统初始化 847

外部事件 848

与其他子系统的交互 861

第三十三章 路由：路由缓存 864

路由缓存初始化 864

Hash 表的组织 865

主要的缓存操作 866

多路径缓存 876

DST 与调用协议间的接口 882

刷新路由缓存 888

垃圾回收 889

出口 ICMP REDIRECT 限速 900

第三十四章 路由：路由表 901

路由 Hash 表的组织 901

路由表初始化 907

路由的添加与删除 908

策略路由及其对路由表定义的影响 913

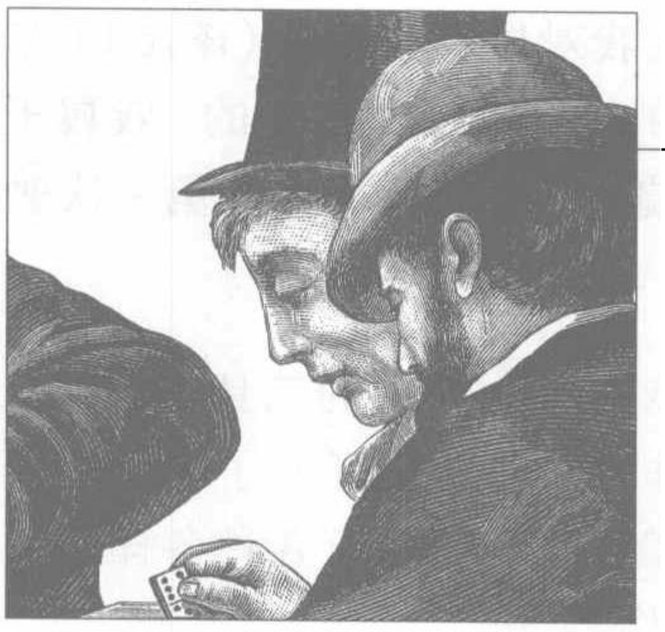
第三十五章 路由：查找 915

查找函数的高层概述 915

辅助函数 916
路由表查找：fn_hash_lookup 917
fib_lookup 函数 922
为接收和发送设置函数 923
输入和输出路由函数的一般结构 926
输入路由 928
输出路由 936
多路径对下一跳选择的影响 945
策略路由 948
源路由 951
策略路由与基于路由表的分类器 952

第三十六章 路由：相关主题 956

用户空间配置工具 956
统计数据 962
通过 /proc 文件系统调整 963
开启和关闭转发 971
本书此部分中的数据结构功能 972
本部分涉及的函数和变量 992
本部分涉及的文件和目录 995



前言

与以前相比，目前网络已成为一个热点话题，任何小的电子产品在它们最新的一代中都嵌入了某些网络功能。使用因特网的人数以及由网络产生的机遇都在继续扩大。如果出现一个健壮的、能够自由获取并有丰富功能的操作系统，如Linux操作系统，那么它被许多嵌入式设备厂商所采用并不奇怪。它的网络性能使得它对于任何种类的网络设备来说都是最佳的操作系统。其网络功能已经得以很好地实现，并且新的功能也很容易添加。如果你是一个嵌入式设备的开发者或者你是一个喜欢体验Linux的学生，本书将给你提供很好的素材。

作为一个纯粹的基于软件的产品，Linux无法与依靠专门硬件帮助的商业产品相竞争。当然这并不是软件的问题，而是由于专用硬件和通用CPU在速度上的不同而造成的。但是，Linux完全可以与基于软件的低端商业产品竞争。当然，对Linux内核进行简单的扩展使开发商可以在混合的系统上（软件和硬件）使用Linux，这只是一个编写必需的设备驱动程序的问题。

Linux也经常作为大学的项目实现以及论文研究所选用的操作系统。并不是所有的这些项目和研究成果都能成为正式的Linux内核（至少现在不是）的一部分。只有少数实现了这个目标，而另一些仅作为可在线获得的正式内核的补丁。如果看到你对Linux内核所贡献的成果被数百万潜在用户使用，这难道不是一个巨大的满足和奖赏吗？但是，如果你的贡献的确得到赏识会有一个问题：你会无力处理无数的邮件或者请求帮助的邮件。

过去的几年中，促使Linux发展的动力在不断地增长，显然这种增长还将继续。

我第一次见识Linux是大约10年前在Bologna（意大利）大学，当时我是一名计算机科学系的研究生。这真是一个奇妙的软件！在家里我可以用一台i286/486计算机进行我的图像处理项目，而不会与其他学生因访问这个大学实验室提供的数量有限的Sun工作站而发生竞争。

从那时起，我融入了充满阳光的Linux世界，甚至开始取代我对显赫的C64（译注1）年代的美好的回忆，那时我第一次使用汇编语言和各种版本的BASIC编程。是的，我属于C64年代，并且从某些程度上可以把我第一次使用C64的编程经历的喜悦与我第一次驶入Linux内核的旅程相提并论。

当我第一次走进美妙的网络世界，就开始“玩”那些Linux上能够获得的工具。我也有幸为UNESCO（联合国教科文组织）在意大利的中心工作，帮助他们开发了完全基于Linux机器的网络课程。因此，我可以使用一个极好的实验室，实验室有各种各样网络设备和文档，更有许多Linux的发烧友互相学习和协同工作。

不幸地是，为了我心灵的平静（但是我希望本书的读者是幸运的，能从中获益），我是那种喜欢了理解所有事情但是又很少想当然的人。所以在UNESCO，我开始浏览内核代码。这不仅仅是一个牢记知识的好方法，而且通过使用用户空间的配置工具也给了我更大的信心：只要一个配置工具没有提供一个特定的选项，一般我都知道是否可以添加或是否需要对内核进行重大的改变。这种学习方式走入了一条没有尽头的路：你总是想知道更多。

开发完一些Linux内核扩展工具（对2.0和2.2版本的某些修订）之后，对操作系统和网络的热衷使我来到硅谷（Cisco Systems公司）。当你学习一门语言，无论是人类的语言还是计算机编程的语言，都有同样的规则：你知道的语言越多，学习一种新的语言就会更容易。你可以鉴别每种语言的优点和缺点，领会设计折中背后的原因，等等。这个规则同样适用于操作系统的学习。

当我注意到缺少关于Linux内核网络代码的好文档和内核其他部分的好的实用书籍时，我决定尝试来补充至少其中的一部分。希望本书能作为你的启蒙文档，就像多年前我所喜爱的那样。

我相信本书，连同O'Reilly出版社其他两本内核方面的书（《深入理解Linux内核》和《Linux设备驱动程序》，其中文版均已由中国电力出版社出版），对于希望学习更多Linux内核知识的人士来说代表着一个好的开始。它们互相补充，并且它们不针对某个特定功能，如果用到，会指导读者利用其他的文档资源。

不过，我仍然建议你冲一点咖啡，打开音乐，花一些时间看源码，试着理解一个给定的功能是如何实现的。我相信你用这种方式建立起来的知识，要比通过其他任何方式记得牢固。捷径虽好，但有时路远点也有它独特的优势。

译注1： 指Commodore 64，20世纪80年代经典的8位计算机。

本书的读者对象

本书可以帮助那些已经有一些网络知识并且还想了解因特网引擎 —— 即因特网协议（Internet Protocol，IP）及其他协议 —— 如何在一个一流的操作系统上实现的人。不过，对每一个主题都有一个理论的介绍，所以新手也能够很快提高。通过充足的范例展示复杂的主题，使其更加易于理解。

Linux不仅仅支持基本的IP，它还有许多高级功能。更重要的是，这些功能的实现必须足够精巧，以很好地与其他一些内核功能配合，例如对称多处理（Symmetric multiprocessing，SMP）和内核抢占。这使得Linux内核的网络代码变成一个非常好的练习场，以训练你的网络知识并使其处于良好状态。

此外，如果你像我一样想学每样东西，你会发现本书中有足够多的细节让你感到相当满意。

背景知识

具有一定操作系统的知识将有所帮助。像操作系统的所有其他组件一样，网络代码必须遵循常用的大小写规范和与内核的其他部分共存必需的隐式规则，包括正确使用上锁；公平使用主存和CPU；着眼于模块化、代码的简洁和好的性能。即使我偶尔在这些方面花一些时间，但还是建议阅读前面提到的O'Reilly的另外两本关于内核的书，以得到在通用操作系统的服务和设计方面更深入和更详细的讨论。

具有一定的网络（尤其是IP方面）知识，也会有所帮助。不过我认为，本书中在每个实现之前的描述已经包含足够多的基本理论，既可以满足初学者的需求也可以满足有经验的读者的需求。

对于本书所讨论的主题的理论描述，并不需要具备任何编程经验。不过，相关实现的描述则需要具备中等C语言水平。第一章将说明编程中经常使用的一系列编程约定和技巧，这对于那些缺少C语言和内核编程经验的人应该有所帮助。

内容组织

网络代码的某些方面需要7章的篇幅来描述，而对于其他方面一章的篇幅就足够了。当一个主题太复杂或者太大需要跨越多章时，本书关于该主题的部分总是以介绍理解实现所必需的理论概念作为开始的一章，而具体实现在另一章描述。所有引用的和次要的材料通常位于该部分的最后一章。不管主题有多大，都使用相同的方式来组织描述。

对每个主题，实现的描述包括：

- 蓝图，用来显示所描述的内核组件位于网络协议栈的位置。
- 主要数据结构的简要描述和表示它们之间相互联系的图示。
- 描述与其他内核功能组件之间的接口，例如，利用通知链（notification chains）或数据结构的交叉引用。防火墙就是这样一个内核功能的例子，给出了在所有网络代码中的许多钩子（hooks）。
- 广泛使用流程图和图示，使得阅读代码以及从庞大且复杂的函数中得到其逻辑变得容易。

参考材料总是包括：

- 逐字段地详细描述最重要的数据结构。
- 包含所有函数、宏和数据结构的简要描述的表，可用来作快速参考。
- 本章涉及的文件列表，并标明它们在内核源码树中的位置。
- 用于配置本章主题的最通用的用户空间工具和内核接口的描述。
- 输出到*/proc*中所有文件的描述。

Linux内核的网络代码不仅仅是一个移动的目标，而且是一个快速的奔跑者。本书没有覆盖所有的网络功能。也许在你阅读本书的时候，一个新的功能正在被添加进来。许多新功能源于单个用户或者组织的需求，或者源于大学的项目，一旦他们认为该功能适用于大多数人的时候，就会设法使之进入正式内核。除了这些功能子集的实现细节，我尽力给出一个功能的一般实现。这将帮助你深入理解代码的改变以及了解新功能是如何实现的。例如，给定任何功能，你需要考虑以下几点：

- 如何设计数据结构和上锁语义？
- 是否需要一个用户空间配置工具？如果需要，它是通过一个已经存在的系统调用与内核交互，还是通过一个`ioctl`命令、一个*/proc*文件或是Netlink套接字？
- 是否需要一个新的通知链，以及是否需要注册到一个已经存在的链？
- 与防火墙是什么关系？
- 是否需要缓存、垃圾收集机制、统计数据等等？

下面是本书中所包含主题的列表：

用户空间和内核之间的接口

在第三章，将简要概述有关网络配置工具用来与内核中与之相匹配部分交互的机制。这并不是一个详细的讨论，但是有帮于你理解内核代码的某些部分。

系统初始化

第二部分描述网络代码关键组件的初始化工作，以及网络设备如何被注册和初始化。

设备驱动程序和协议处理程序之间的接口

第三部分提供了入口（ingress）封包如何被设备驱动传递给上层协议的详细描述，以及反向的过程。

桥接

第四部分描述透明桥接和生成树协议（Spanning Tree Protocol）。生成树协议是L3（3层）路由协议在L2（2层）的对应体。

IPv4

第五部分描述封包在IPv4层如何被接收、传输、转发和在本地提交。

IPv4和传输层（L4）协议之间的接口

第二十章说明了目的是本地主机的IPv4封包如何被提交到传输层协议（TCP、UDP等）。

ICMP

第二十五章描述ICMP的实现，这是本书中唯一包含的传输层（L4）协议。

邻居协议

给定IP地址后，这些协议可以查找出本地网络地址。第六部分描述了各种协议的通用基础构架和IPv4使用的ARP邻居协议的细节。

路由

第七部分是本书中篇幅最大的一部分，描述路由缓存和路由表。一些高级功能如策略路由和多路径路由也包含在其中。

未被包含的内容

由于篇幅所限，我只能选择Linux网络功能的一个子集来讨论。没有任何选择能使所有人都高兴，但是我想覆盖网络代码的核心部分，并且通过本书的学习，你会发现学习任何其他内核网络功能都变得轻松了。

在本书中，我决定重点关注网络代码，从设备驱动程序与协议处理程序之间的接口，到IPv4与L4协议间的接口。覆盖所有功能和确保质量相比，我更倾向于确保质量第一，并且选择一些具有代表性的功能子集作为内核网络实现旅途的最佳起点。

下面是一部分由于篇幅所限而没有谈到的功能的列表。

IPv6

虽然本书没有讨论IPv6，但关于IPv4的描述对你深入理解IPv6的实现有很大的帮助。这两个协议中的函数和变量的命名规则一致。其接口与Netfilter的接口也很类似。

IP安全协议

内核提供了一个通用的密码系统基础构架，包括密码（cipher）和摘要（digest）算法。密码层的第一个接口是同步的，但是最新的改进添加了一个异步接口，允许Linux利用硬件卡以减轻CPU的负载。

在内核中利用密码层实现了IPsec协议的组件认证头（Authentication Header，AH）、压缩安全有效载荷（Encapsulating Security Payload，ESP）以及IP压缩（IP Compression，IPcomp）。

IP多播和IP多播路由

多播功能的实现遵循IGMP（Internet Group Management Protocol，因特网组管理协议）的版本2和3。同时支持多播路由，遵循PIM（Protocol Independent Multicast，协议独立型多目标广播）的版本1和2。

传输层（L4）协议

Linux内核中实现了一些L4协议。除了两个众所周知的协议UDP和TCP之外，Linux还实现了更新的SCTP（Stream Control Transmission Protocol，流控制传输协议）。关于这些协议的实现，需要一本新书的篇幅才能进行详细的描述。

流量控制

这是Linux的服务质量（Quality of Service，QoS）层，也是内核网络代码中另一个有趣而强有力的组件。流量控制作为一个通用的基础构架来实现，由流量分类器和排队规则集合组成。在第十一章对此进行了简要描述，并且提供了主传送例程的接口。在 *http://lartc.org* 网站可获得大量的文档。

Netfilter

本书没有讨论防火墙代码基础构架和它的扩展（包括各种NAT版本），不过描述了它与书中讨论的其他网络功能之间的交互。在Netfilter主页（*http://www.netfilter.org*），你可以找到一些关于其内部结构的有趣的文档。

网络文件系统

在内核中实现了几种网络文件系统，包括NFS（版本2、3和4）、SMB、Coda和 Andrew。对于虚拟文件系统层的详细描述，可以参读《深入理解Linux内核》一书，然后深入研究源码以了解这些网络文件系统如何与内核交互。

虚拟设备

使用专门的虚拟设备是实现网络功能的基础。例如，802.1Q、绑定（bonding）和许多隧道协议，诸如IPIP（IP-over-IP）和GRE（Generalized Routing Encapsulation，通用路由封装协议）。虚拟设备需要遵循与实际设备一样的原则，并向其他内核组件提供相同的接口。在不同章节中，如果需要我会比较实际设备和虚拟设备的行为。只有在本书第四部分桥接接口中，对虚拟设备进行了详细的描述。

DECnet、IPX、AppleTalk 等

这些网络协议都有其历史渊源，而且现在仍然在使用，但是与 IP 相比已经很少使用了。忽略它们是为留出更多的篇幅来讨论影响更多用户的主题。

IP虚拟服务器

这是网络代码另一个有趣的部分，在 *http://www.linuxvirtualserver.org/* 中给出了描述。这个功能可以在用不同调度算法构建集群服务器（clusters of servers）时使用。

简单网络管理协议(SNMP)

本书没有单设章节来专门讨论SNMP(Simple Network Management Protocol)，但是对于每个功能，在必要的时候我会给出由内核保存的所有计数器和统计数据的描述，以及用于操作它们的例程和用于输出它们的 */proc* 文件。

帧分流器(Frame Diverter)

这个功能允许内核截获不是传给本地主机的入口帧。我会在第三部分简要介绍一下。其主页为 *http://diverter.sourceforge.net*。

许多其他网络项目也可作为独立的内核补丁（patch）来使用，但是在这里无法逐一列出。我发现了一个非常使人着迷而且很有前景的研究课题 —— 特别是其与 Linux 内核路由代码的关系 —— 就是高可配置性 Click 路由器，目前由 *http://pdos.csail.mit.edu/click/* 提供。

由于这是一本关于内核的书，因此并没有讨论用户空间配置工具。不过，对于每个主题，对其最常用的用户配置工具与内核之间的接口都进行了描述。

排版约定

下面是本书中使用的排版约定：

斜体（*Italic*）

用于文件和目录名、程序和命令名、命令行选项、URL 和新术语。

等宽字体（Constant Width）

在范例中用以显示文件内容或命令的输出，在正文中用于表示出现的C语言代码里的单词或其他文字字符串。

等宽斜体（*Constant Width Italic*）

用于表示需要用户以实际值替代的内容。

等宽黑体（**Constant Width Bold**）

用于范例中显示命令或其他应该由用户逐字输入的文本。

要特别注意正文中有以下图标的注释：

注意：这是一种技巧。它包含了关于主题的一些有益的补充信息。

警告：这是一个警告。它帮助你解决并避免讨厌的问题。

使用代码范例

本书的目的是为了帮助你完成任务。一般来说，你可以在你的程序和文档中使用本书中的代码，这些代码范例具有BSD/GPL双重授权许可。

我们虽然不要求但很赞赏在使用这些代码时注明来源。注明来源通常包括标题、作者、出版社和ISBN号，比如："Understanding Linux Network Internals by Christian Benvenuti. Copyright 2006 O'Reilly Media, Inc., ISBN 0-596-00255-6"。

建议与评论

本书的内容都经过测试，尽管我们做了最大的努力，但错误和疏忽仍然是在所难免的。如果你发现有什么错误，或者是对将来的版本有什么建议，请通过下面的地址告诉我们：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

100035 北京市西城区西直门南大街2号成铭大厦C座807室
奥莱利技术咨询（北京）有限公司

我们为本书专设了网页，列出了勘误表、实例以及所有附加信息，你可以访问如下页面：

http://www.oreilly.com/catalog/understandlni/（英文版）
http://www.oreilly.com.cn/book.php?bn=978-7-5083-7964-7（中文版）

关于本书的评论以及技术问题，请发邮件到：

bookquestions@oreilly.com
info@mail.oreilly.com.cn

要了解关于我们的图书、论坛、资源中心以及O'Reilly网络的更多信息，请登录：

http://www.oreilly.com
http://www.oreilly.com.cn

致谢

本书的出版离不开一个引人注意的主题以及众多读者的支持。这个引人注意的主题就是Linux，它是一个现代操作系统，任何人都有机会参与其中并为之贡献力量；不计其数的Linux用户成了本书的重要读者，他们不仅分享其他人的优秀成果，而且还在Linux的新发行版中贡献自己的心得。我始终乐于把自己的知识和热情与大家一起分享。通过本书，我尽自己最大的努力为同样对Linux感兴趣的人们开辟一条通往Linux内核的小径。

当然，我不可能躺在海滩的吊床上，左手雪糕，右手鼠标就能完成这些工作。在某段程序实现的背后需要做大量的调查研究工作。令人难以置信的是，在开发者邮件列表中可以挖掘出大量有价值的信息。同时，当你对朋友们的工作内容感兴趣时，他们非常乐意与你分享知识。

的确，本书的出版离不开编辑Andy Oram的极大帮助和建议。由于网络代码的频繁改变，在本书的撰写过程中少量章节的代码在不停地更新，最后在Andy Oram的帮助下才定稿。

同时还要感谢所有支持我的人，感谢Cisco Systems提供灵活的工作时间，保证了此书的撰写。

特别感谢技术评审们在短时间内对本书的评审，同时他们还给出了有价值的注释，使我得以发现错误，提高了本书素材的质量。参与评审本书的有：Jerry Cooperstein、Michael Boerner和Paul Kinzelman（按人名字母先后排列）。同时还要感谢Francois Tallet评审本书第四部分，感谢Andi Kleen在本书第五部分给予的反馈。

第一部分

基础背景

这一部分的内容是你充分理解本书其余部分必需的基本知识。如果你已熟悉Linux内核，或者你是一个有经验的软件工程师，这几章读起来会很快。对其他读者而言，建议你在继续读本书其他部分之前，先熟悉这些内容。

第一章 简介

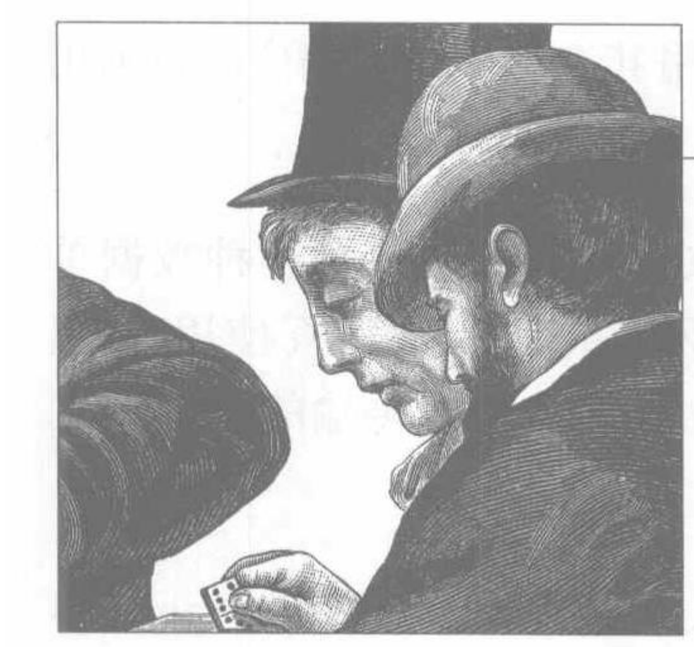
本章主要介绍在网络代码中你会时常碰到的一些常见的程序设计模式和技巧。

第二章 关键数据结构

在本章，你会看到网络代码所用的最重要数据结构的详细说明：套接字缓冲区sk_buff及网络设备net_device。

第三章 用户空间与内核的接口

对于本书所谈的每项功能，都会有一些章节说明用户空间配置工具与内核的通信。本章的信息可帮助你更为透彻地理解那些章节。



第一章

简介

对于大型项目源码的研究，就像进入一个陌生新奇的领域，有其习惯和不能言表的期待。事先学习一些主要约定，并尝试和几种组件互动而不是仅站在旁边观察，肯定会对你有所帮助。

本章主要介绍一些在网络代码中经常遇到的通用编程模式和技巧。

可能的话，希望你利用一些用户空间工具（user-space tool）尝试与内核网络代码中的特定部分交互。如果你首选的Linux发行套件中没有安装这些工具，或者你只想将其更新成最新版本，本章提示你可以去哪里下载这些工具。

我也会介绍一些工具，让你以优雅的方式探索庞大的内核代码。最后，我会简要地说明为什么某些在Linux社区中广为使用的内核特性并没有集成到正式内核版本中的原因。

基本术语

本节介绍一些本书中广为使用的术语和缩写。

在网络文献中，八个位的量通常称为八位组（octet）。然而，本书中使用更常见的术语字节（byte）。毕竟，本书描述的是内核行为，而不是某些网络抽象概念，并且内核开发人员都习惯用字节来思考。

术语向量（vector）和数组（array）是交替使用的。

当涉及TCP/IP网络协议栈的分层时，使用L2、L3和L4分别表示链路层（link layer）、网络层（network layer）以及传输层（transport layer）。这些数是基于著名的（现在已不确切）七层OSI模型。多数情况下，L2为Ethernet的同义词，L3指IPv 4或IPv6，L4

是指UDP、TCP或ICMP。当需要专指特定协议时，会使用其名称（如TCP），而不用通称的*Ln*协议的术语。

在不同的章节中，我们会说明位于网络协议栈中的各层协议如何接收和传输各种数据单元。在上下文中，术语“入口数据（ingress）”和“输入数据（input）”会交互使用，“出口数据（egress）”和“输出数据（output）”也一样。数据单元的接收或传输的行为分别用缩写为RX和TX表示。

数据单元的名称根据其所在层的不同而不同，如帧（frame）、封包（packet）、段（segment）及消息（message）等（详见第十三章）。表1-1概括了本书的主要缩写。

表1-1：本书常用的缩写

缩写	意义
L2	链路层（例如，Ethernet）
L3	网络层（例如，IP）
L4	传输层（例如，UDP/TCP/ICMP）
BH	下半部（Bottom Half）
IRQ	中断（事件）
RX	接收
TX	传输

常见编码模式

与其他内核功能类似，每种网络功能只是内核各种功能之一。因此，也必须适当而公平地 使用内存、CPU以及其他各种共享资源。多数功能在内核中并非独立的代码片段，而是根据该功能而定，或多或少会和其他内核组件交互。因此，类似的功能都会尽可能采用类似的机制实现（已经发明的东西没有必要再重新发明一次）。

有些内核组件有一些共同的需求，如必须分配同一种数据结构类型的几个实例，必须记录某种数据结构实例的引用（reference），以避免不安全的内存回收（deallocation），诸如此类。下面几小节中，我们将说明Linux处理这类需求的常见做法，还将介绍在浏览内核代码时可能碰到的常见编码技巧。

本书所谈论的子系统（subsystem）是一个较为宽松的术语，指的是实现一组主要功能[如IP或路由（routing）]的文件集合，这个文件集合的维护由同一群人完成，而且改动要步调一致。本章剩余部分会使用内核组件（kernel component）这个术语，该术语涉

及这些子系统，因为这里讨论的约定适用于内核的多数部分，而并非只限于网络相关部分。

内存缓存

内核分别使用kmalloc和kfree函数分配和释放一个内存块。这两个函数的语法，类似于另外两个来自于*libc*用户空间库的姊妹函数malloc和free的调用。有关kmalloc和kfree的细节，可参考《Linux 设备驱动程序》（O'Reilly 出版）一书。

内核组件为同一种数据结构类型分配几个实例（instance）是很常见的事。当分配和回收经常发生时，相关联的内核组件初始化函数（例如，路由表的fib_hash_init）通常会分配一块特殊的内存缓存，以作分配之用。当一个内存块被释放时，实际上是返回到当初被分配的同一个缓存区中。

内核维护的其专属内存缓存的一些网络数据结构的例子包括：

套接字缓冲区描述符

这个缓存是由*net/core/sk_buff.c*的skb_init分配的，用于分配sk_buff缓冲区描述符。sk_buff结构可能是网络子系统中分配和回收注册次数最高的。

邻居协议映射

每个邻居协议都使用一个缓存区，以分配存储L3层到L2层地址映射的数据结构。参见第二十七章。

路由表

路由代码使用两块缓存，用于定义路径的两个数据结构。参见第三十二章。

以下是用于处理内存缓存的关键内核函数：

kmem_cache_create

kmem_cache_destroy

创建和销毁一个缓存。

kmem_cache_alloc

kmem_cache_free

为缓存分配及回收一个缓冲。通常由调用包裹函数（wrapper）实现分配和回收一个缓存缓冲区，包裹函数在较高层中管理用于处理分配和回收的请求。例如，用kfree_skb要求释放一个sk_buff缓冲区的一个实例时，只有当所有对该缓冲区的引用都已释放，而且相关的子系统（例如防火墙）也把所有必要的清理工作都做完后，才会去调用kmem_cache_free。

给定缓存区（存在时）所能分配的实例数目的限制，通常由内含kmem_cache_alloc的包裹函数决定，但是有些时候可以通过*/proc*里的参数进行配置。

有关内存缓存的实现以及其和厚片分配器（slab allocator）之间的接口细节，请参考《深入理解 Linux 内核》（O'Reilly 出版）一书。

缓存和 hash 表

使用缓存以提高性能是很常见的事。在网络代码中，L3 层到L2 层之间的映射（如IPv4所使用的 ARP 缓存）以及路由表等都使用缓存。

缓存查询函数通常有一个输入参数，指出发生缓存不命中（cache miss）时，是否应该在缓存中创建并添加一个新的元素。而没有这个参数时，查询函数则随时会把不命中的元素新增进来。

缓存通常采用散列表（即 hash 表）实现。内核提供一组数据类型，如单向和双向列表，用来构建简单的 hash 表。有些输入经 hash 后得到相同的值，其标准处理方式就是将其放入一个列表中。查询这个列表所花的时间，比采用 hash 关键字（key）查询所花的时间长得多。因此，把 hash 值相同的输入值数目缩到最小一直都是很重要的事。

当 hash 表的查询时间（无论是否使用缓存）是拥有者子系统（owner subsystem）的关键参数时，可以实现一种机制，以增加hash表的大小，使得冲突列表（collision list）的平均长度得以下降，而平均查询时间得以改善。参见第三十四章“基于子网掩码的hash表容量的动态调整”一节的实例。

你也会发现有些子系统（如邻居层）在关键字（key）中添加一个随机成分（有规律地改变），使元素能分散到缓存 bucket（译注 1）中。这是为了减少拒绝服务（Denial of Service，DoS）攻击的损害，这种攻击把 hash 表的元素都集中放到一个 bucket 中。参见第二十七章“缓存”一节的实例。

引用计数

当一段代码试图访问一个已被释放的数据结构时，内核并不欢迎，而大多用户对于内核的这种反应也会感到不愉快。为了避免这类讨厌的问题，同时让垃圾收集机制更为容易而且有效率（参见本章稍后“垃圾收集”一节）多数数据结构都会保存引用计数（reference count）。对每一个数据结构的每一次存储和释放，良好的内核组件都会分别

译注 1： bucket 指 hash 表中的一项记录，后文中将直接使用 bucket。

递增和递减该数据结构的引用记数。任何数据结构若需要引用计数，拥有该结构的内核组件通常会提供两个函数，用以递增和递减引用计数值。这类函数通常分别称为xxx_hold和xxx_release。有时候，释放函数被称为xxx_put（例如，net_device结构的释放函数就是dev_put）。

虽然我们都希望内核中没有不良组件，但是开发人员也是人，因此不可能永远写出无缺陷的代码。引用计数的运用是一种简单但有效率的机制，可以避免释放仍在被引用的数据结构。然而，因其忽略了平衡递增量和递减量，因此并不能总是完全解决问题：

- 如果你释放了一个对数据结构的引用，但却忘了调用xxx_release函数，内核将永远不会让该数据结构被释放掉(除非另一段出错的代码碰巧又错误地多调用了一次释放函数！)。这样会导致内存逐渐耗尽。
- 如果你引用了一个数据结构，但却忘了调用xxx_hold，后来你碰巧又成为唯一的引用持有者，结果该结构就会提早被释放（因为你没有说明）。这种情况肯定比前一种更具灾难性：下次你试图访问该结构时，可能会破坏其他数据，或引起内核恐慌（panic，译注2），使得整个系统瞬间崩溃。

当某个数据结构因某种原因被删除时，可以明确通知引用持有者该数据结构即将消失，使其能按规则释放其引用，而这是通过通知链实现的。参见第八章“引用计数”一节的有趣实例。

数据结构的引用计数通常在下列情况下可做递增：

- 两种数据结构类型之间有很密切的关系。在这种情况下，其中的一个结构通常会维护一个被初始化为第二个结构的地址的指针。
- 一个定时器启动，而该定时器的处理程序将访问该数据结构。当定时器启动时，该结构的引用计数值就会递增，因为你最不愿意看到的是，在定时器到期前该数据结构就被释放了。
- 对列表或hash表的成功查询会返回一个指向匹配元素的指针。多数情况下，返回的结果被调用者用来执行某种任务。因此，常见的做法是，查询函数递增匹配元素引用计数值，然后在必要时再让调用者予以释放。

译注2：panic的原意为恐慌，内核恐慌指一个消息，它由操作系统发现难以修复的内部错误后显示“panic”。主要在Unix或类Unix中使用该术语，内核处理程序对于panic的处理通常是将调试信息输出到屏幕或指定的文件，然后使计算机停止运行。后文中将直接使用原文。

当一个数据结构的最后一个引用也被释放时，该数据结构就可被释放，因为再也用不到该数据结构了，不过不一定非这么做不可。

新引入的*sysfs*文件系统有助于产生内核的一部分良好代码，运用它可以更关注引用计数和一致性。

垃圾收集

内存是有限的共享资源，不应该浪费，特别是在内核中，因为内核不使用虚拟内存。多数内核子系统会实现某种垃圾收集，以回收由未使用的或无效的数据结构实例所持有的内存。根据特定功能所需而定，你会发现有两种主要的垃圾收集：

异步

这种垃圾收集类型和特定事件无关。一个定时器会定期启用一个函数，以扫描一组数据结构，然后把那些适合删除的数据结构释放掉。数据结构符合删除的条件依赖于该子系统的功能和逻辑，但是，一个常见的准则为是否存在null（空）引用计数。

同步

在内存不足时的情况下会立即触发垃圾收集，而不能等待定时器触发的异步垃圾收集。用于选取符合删除的数据结构的准则，不一定与异步清理机制所用的准则相同（例如，准则更为积极）。参见第三十三章的实例。

在第七章，你将了解内核如何管理回收由初始化函数所占用的内存，因为初始化函数执行后就不再需要了。

函数指针和虚拟函数表（VFT）

函数指针是一种方便的方式，可以写出简洁的C代码，又能利用面向对象语言的某些优点。在数据结构类型（对象）的定义中，你可以包含一组函数指针（方法）。于是，该结构的某些或全部的操作都可通过嵌入的函数完成。C语言的函数指针在数据结构中类似如下所示：

```
struct sock {
    ...
    void    (*sk_state_change)(struct sock *sk);
    void    (*sk_data_ready)(struct sock *sk, int bytes);
    ...
};
```

使用函数指针最主要的优点就是，可以根据不同准则以及该对象所扮演的角色进行初始

化。因此，调用sk_state_change时，实际上可能会为不同的sock套接字而调用不同的函数。

函数指针在网络代码中广为使用。下面只是很少的实例：

- 当入口数据封包或出口数据封包由路由子系统处理时，会对缓冲区数据结构中的两个函数做初始化。在第三十五章就会看到，参考第二章所列sk_buff数据结构中完整的函数指针列表。
- 当数据封包已准备好在网络硬件上传输时，就会交给net_device数据结构的hard_start_xmit函数指针。该函数由该设备所关联的设备驱动程序进行初始化。
- 当L3协议想传输数据封包时，会调用一组函数指针中的一个。这些函数指针由该L3协议相关联的地址解析协议（address resolution protocol）负责初始化为一组函数。根据函数指针初始化的实际函数，可以产生透明化的L3层到L2层的地址解析（例如，IPv4数据封包经过ARP）。当不需要地址解析时，就会使用另一个函数。参见第六部分对此接口的详细讨论。

通过上述几个例子可知，函数指针可作为内核组件之间的接口，或者作为一种通用的机制，根据不同子系统所做的某事的结果，在适当的时机调用适当的函数处理程序（function handler）。在其他情况下，函数指针也可作为一种简单方式，使协议、设备驱动程序或任何其他功能得以让某种动作个别化。

让我们看一个例子。当设备驱动程序在内核注册网络设备时，无论是何种类型的设备，都需要经过一系列步骤。到了某一时刻，就会对net_device数据结构调用一个函数指针，使设备驱动程序在必要时做些其他事情。设备驱动程序既可以把该函数指针初始化成其拥有的一个函数，也可以让该指针保持为NULL，因为内核默认的执行步骤已经足够了。

执行一个函数指针之前，必须始终检查其值，以避免提取NULL指针所指向单元的值的事情发生，例如，register_netdevice中的片段：

```
if (dev->init && dev->init(dev) != 0) {
    ...
}
```

函数指针有一个主要的缺点：使阅读源代码稍显困难。沿着给定的代码路径读下去，最后的关注焦点可能就是函数指针调用。在这种情况下，继续沿着代码路径读下去之前，必须搞清楚该函数指针是如何初始化的。这可能取决于一些不同的因素：

- 当选择函数分派给函数指针是基于特定数据片段时（如处理数据的协议，或者接收

的给定数据封包所来自的设备驱动程序），就比较容易推导出该函数。例如，如果给定设备是由*drivers/net/3c59x.c*设备驱动程序管理，就可以阅读该设备驱动程序提供的设备初始化函数，从而导出分派给net_device数据结构的特定函数指针的函数。

- 当函数的选择是基于更为复杂的逻辑之时，如L3层到L2层地址映射解析的状态，任何时刻所用的函数都会依赖于无法预测的外在事件。

指向一个数据结构的一组函数指针通常就称为虚拟函数表（virtual function table，VFT）。当VFT作为两个主要子系统之间的接口时，如L3与L4协议层之间，或者当VFT只是作为输出成为通用的某个内核组件（一组对象）的接口时，数据结构中的函数指针数目可能会膨胀，以包含许多不同指针，从而适应各种各样的协议或其他功能。每种功能最后可能只会用到所提供的众多函数中的一小部分而已。在第六部分可以看到一个实例。当然，如果这种用法用过头的话，VFT就会变得很笨重，从而需要大量的重新设计。

goto语句

很少有C程序员喜欢goto语句。不谈goto的历史（计算机程序设计领域历时最久、最有名的争论）我将总结一些goto通常被废弃不用，而Linux内核仍然使用它的原因。

任何使用goto语句的代码片段都能被改写成不使用该语句。使用goto语句会使代码的可读性降低，调试困难。因为对于任何goto之后的语句，你都无法明确推导出其执行的条件。

做一个类比：一棵树中无论哪个节点，你都知道从根节点到该节点的路径。但是，如果随机地把藤蔓缠绕到一些分枝上，根节点与其他节点之间就不一定只有唯一的路径了。

然而，因为C语言没有提供显式异常事件（其他语言因为性能损失和编码复杂度的缘故，通常也不提供显式异常事件）谨慎使用goto语句可以轻松地跳到处理未预料或特殊事件的代码。在内核程序设计中，特别是网络程序设计中，这类事件很常见，因此，goto语句变成了一个方便的工具。

在为内核使用goto语句辩解时，我必须指出一点，开发人员绝不是用到无法无天的地步。尽管有30 000个以上的实例，但主要都是用于处理函数内的不同返回代码，或者用于跳出一层以上的嵌套。

向量定义

在某些情况下，数据结构定义的末端会包括一个可选的区块。以下是一个实例：

```
struct abc {
    int age;
    char *name[20];
    ...
    char    placeholder[0];
}
```

可选区块从placeholder开始。注意，placeholder定义为大小为0的向量。也就是说，当abc被分配为带有可选区块时，placeholder就指向此区块的起始处。不需要可选区块时，placeholder就只是一个指向此结构尾端的指针而已，不耗用任何空间。

因此，如果abc被好几段代码所用到，每段代码都可以使用相同的基本定义（避免各自以略微不同的方式做同一件事而造成混淆），但是，又可根据其需求以不同方式将abc的定义予以个别化。

在本书中将多次看到这种数据结构定义。第十九章有一个实例。

条件指示指令（#ifdef及其系列指令）

有时候必须把一些条件指示指令传给编译器。大量使用条件指示指令，会降低代码的可读性，但是我可以说，Linux并没有滥用这些指令。条件指示指令因各种不同原因而出现，但是，我们感兴趣的是那些用于检查内核是否支持特定功能的条件指示指令。如*make xconfig*这类配置工具可以确认该功能是否被编译进来、完全不支持或者可以用模块的形式加载。

例如，#ifdef或#if defined用于指示C预处理程序完成检查功能：

从数据结构定义中引入或排除字段

```
struct sk_buff {
    ...
#ifdef CONFIG_NETFILTER_DEBUG
    unsigned int nf_debug;
#endif
    ...
}
```

在这个例子中，Netfilter调试功能需要sk_buff结构中的nf_debug字段。当内核不支持Netfilter调试时（只有少数开发人员需要这种功能）就不需要引入这个字段，引入该字段对每个网络数据封包而言只会消耗更多的内存。

从函数中引入或排除一些代码片段

```
int ip_route_input(...)
{
    ...
```

```
        if (rth->fl.fl4_dst == daddr &&
            rth->fl.fl4_src == saddr &&
            rth->fl.iif == iif &&
            rth->fl.oif == 0 &&
#ifndef CONFIG_IP_ROUTE_FWMARK
            rth->fl.fl4_fwmark == skb->nfmark &&
#endif
            rth->fl.fl4_tos == tos) {
            ...
        }

}
```

在第三十三章会说明，路由缓存查询函数 ip_route_input 只有在内核编译支持“IP: use netfilter MARK value as routing key”功能时，才会检查由防火墙所设的标记值。

为函数选择适当的原型

```
#ifdef CONFIG_IP_MULTIPLE_TABLES
struct fib_table * fib_hash_init(int id)
#else
struct fib_table * _ _init fib_hash_init(int id)
{
    ...
}
```

在这个例子中，当内核不支持策略路由（Policy Routing）时，这些指示用于把 __init 标志（注 1）加至原型。

为函数选择适当的定义

```
#ifndef CONFIG_IP_MULTIPLE_TABLES
...
static inline struct fib_table *fib_get_table(int id)
{
    if (id != RT_TABLE_LOCAL)
        return ip_fib_main_table;
    return ip_fib_local_table
}
...
#else
...
static inline struct fib_table *fib_get_table(int id)
{
    if (id == 0)
        id = RT_TABLE_MAIN;
    return fib_tables[id];
}
```

注 1：　参见第七章有关此宏的说明。

```
...
#endif
```

注意这种情况与前一种情况的不同之处。在前一种情况中，函数体位于 #ifdef/#endif 块之外，而现在这种情况中，每个块都包含一个完整的函数定义。

变量和宏的定义或初始化也可以使用条件编译。

知道某些函数或宏存在多个定义是很重要的。与前面的例子类似，这些函数或宏的定义，由预处理宏程序在编译期间决定。否则，当你查看函数、变量或宏定义时，可能会看到错误的定义。

参见第七章关于如何引入特殊宏的讨论，在某些情况下，这可以减少条件指示编译器指令的使用。

条件检查的编译期间最优化

多数情况下，当内核用某些外部值比较一个变量以了解是否满足给定条件时，其结果极有可能是可预测的。这是很常见的事，例如，那些强制健康检查（sanity check）的代码。内核分别使用 likely 和 unlikely 宏，进行返回真（1）或假（0）的包裹比较。这些宏会利用*gcc*编译器的一项功能，这个功能就是可依据该项信息使代码的编译最优化。

以下是一个实例。假设你需要调用 do_something 函数，调用失败时，你必须用 handle_error 函数进行处理：

```
err = do_something(x,y,z);
if (err)
    handle_error(err);
```

假设 do_something 很少失败，可以按下列方式重写代码：

```
err = do_something(x,y,z);
if (unlikely(err))
    handle_error(err);
```

likely 和 unlikely 宏可能做的优化实例之一就是处理 IP 报头（header）里的选项。IP 选项的使用只限于一些特定情况，因此，内核可以安全地假设多数 IP 封包不会携带 IP 选项。当内核转发 IP 封包时，需要根据第十八章所述的规则处理那些选项。转发 IP 封包的最后一个阶段由 ip_forward_finish 负责。此函数使用 unlikely 宏把检查是否要处理 IP 选项的条件包裹起来。参见第二十章的“ip_forward_finish 函数”。

互斥

网络代码中广泛使用上锁（locking），在本书中每项主题之下，你可能都会发现上锁这一议题。对众多程序设计类型而言，尤其是针对内核的程序设计，互斥（mutual exclusion）、上锁机制以及同步都是一般性主题，而且相当有趣而复杂。近年来，Linux不但引入互斥，而且对一些方法做了优化。因此，本节只总结在网络代码中所见到的上锁机制。建议参阅O'Reilly出版的《深入理解Linux内核》和《Linux设备驱动程序》中深入而详尽地讨论。

每种互斥机制都是特定环境下的最佳选择。以下是网络代码中常见的可选互斥方法的简洁摘要：

回转锁（Spin lock）

这是一种在某一时刻只能由一个执行的线程（thread）所持有的锁。若试图获得由另一个正在执行的线程拥有的锁，则要进入循环等待，直到该锁被释放为止。由于进入循环将造成浪费，回转锁只用于多处理器系统中，并且通常只用在开发人员预期该锁只会被短期持有的时候。此外，由于会引起其他执行线程的浪费，执行线程在持有回转锁时不能休眠。

读－写回转锁

当给定锁的使用可以明确分为只读和读－写时，应该先使用读－写回转锁。读回转锁和读－写回转锁的差别在于，对读－写回转锁而言，多个读取者可以同时持有该锁。然而，在同一时刻该锁的持有者只能有一个可以写入，同时，当该锁被一个写入者持有时，读取者都不能取得该锁。由于读取者的优先级高于写入者，因此当读取者的数目（或者取得只读锁的数目）远远超过写入者的数目（或者取得读－写锁的数目）时，这种类型的锁能很好地工作。

当该锁是在只读模式下取得时，不能直接提升成读－写模式：该锁必须被释放，再以读－写模式取得。

读取（拷贝）更新（Read-Copy-Update，RCU）

RCU是Linux提供互斥的最新机制之一，在下列特定条件下工作得非常好：

- 与只读锁的请求相比，读－写锁的请求很少见。
- 持有该锁的代码以原子的形式执行，而且不能休眠。
- 由该锁保护的数据结构是通过指针访问的。

第一个条件涉及性能，而另外两个条件则是RCU工作原理的基础。

注意，第一个条件似乎意味着应该使用读－写回转锁来代替RCU。为了了解当恰

当使用时，为什么RCU比读－写回转锁工作得更好，你需要考虑其他方面，如SMP（symmetric multiprocessing，对称式多重处理）系统上处理器缓存的影响。

RCU设计背后的工作原理既简单又有效。有关RCU优势的清晰描述以及其实现的简述，可以参考Paul McKenney在《Linux Journal》(*http://linuxjournal.com/article/6993*)（注2）上所发表的一篇文章。你也可以参考《深入理解Linux内核》以及《Linux设备驱动程序》。

RCU在网络代码中的使用实例就是路由子系统。在缓存中查询比更新更要频繁，因此，实现路由缓存查询的函数不会在搜索期间被堵塞。参见第三十三章。

内核也提供信号量（semaphore）但是本书所涉及的网络代码中很少使用。然而，用于把配置变更序列化（serialize）的代码就是使用信号量的例子，在第八章我们将看到这种行为。

主机和网络之间的尾端（order，译注3）转换

超过一个字节以上的数据结构可以采用两种不同格式存储于内存中：小尾端（Little Endian）和大尾端（Big Endian）。第一种格式把最低字节存储在最低内存地址，而第二种格式则刚好相反。像Linux这类操作系统所用的格式依赖于所用的处理器。例如，Intel处理器遵循的是小尾端模式，而Motorola处理器使用的是大尾端模式。

假设我们的Linux机器从一台远程主机接收到一个IP封包。由于Linux不知道远程主机初始化协议报头时所用的格式是小尾端还是大尾端，那么Linux将如何读取报头？出于这个原因，每个协议族都必须定义其所用的“尾端（endianness）”。例如，TCP/IP协议栈采用大尾端模式。

但是，这样仍然会给内核开发人员留下一个问题：他所写的代码必须能在众多支持不同尾端模式的处理器上工作。有些处理器可能与入口封包是同一尾端，但是，如果尾端不同，就必须转换成处理器所用的尾端模式。

因此，每次内核需要读取、保存或比较超过一个字节的IP报头字段时，首先就必须将网络尾端转换成主机尾端，或者相反。TCP/IP协议栈的其他协议也是如此。当协议与本地主机都是大尾端时，转换函数就是空操作的（no-op），因为没必要做任何转换。但是，转换函数还是会始终出现在代码中，为的是让代码可移植；只有转换函数本身是与平台相关的。表1-2列出了用于两字节和四字节字段的尾端转换的主要宏。

注2：　更多文件可参考作者所维护的网站：*http://www.rdrop.com/users/paulmck/rclock*。

译注3：“order”有些书籍翻译为“字节次序”，译者认为翻译为“尾端”更为贴切。

表 1-2：尾端转换函数

宏	意义（short：2 字节，long：4 字节）
htons	主机－网络尾端（short）
htonl	主机－网络尾端（long）
ntohs	网络－主机尾端（short）
ntohl	网络－主机尾端（long）

这些宏定义在一般的头文件 *include/linux/byteorder/generic.h* 中。以下是每种体系如何根据其尾端剪裁这些宏的定义：

- 就每种体系而言，每一体系目录（per-architecture directory，译注 4）*include/asm-XXX/* 中都有一个 *byteorder.h* 文件。
- 根据处理器的尾端而定，该文件会包含 *include/linux/byteorder/big_endian.h* 或 *include/linux/byteorder/little_endian.h*。
- *little_endian.h*和*big_endian.h*都会包含通用的*include/linux/byteorder/generic.h*文件。表 1-2 中定义的宏是基于其他的由 *little_endian.h* 和 *big_endian.h* 定义的不同的宏，而这也就是体系的尾端影响表 1-2 宏定义的方式。

对于表 1-2 中的每个宏 *xxx*，都会有一个姊妹宏 `__constant_xxx` 与之对应。当输入字段是常数值时，如枚举列表中的一个元素（参见第二十八章“初始化 ARP 协议”一节的实例），就会使用此姊妹宏。注意，即使输入值为常数，表 1-2 中的宏也时常用于内核代码中（参见第十三章“设置 Ethernet 协议及长度”一节的实例）。

本节前面已经谈到，当一个数据字段的范围超过一个字节时，尾端是很重要的。当一个字段所占的字节为一个或一个以上，而且又定义为位字段集（collection of bitfields）时，尾端实际上更为重要。例如，看一下第十八章中图 18-2 的 IPv4 报头的形式，以及内核在 *include/linux/ip.h* 中定义 iphdr 结构的方式。内核在前面所提到的 *little_endian.h* 和 *big_endian.h* 文件中，分别定义了 `__LITTLE_ENDIAN_BITFIELD` 和 `__BIG_ENDIAN_BITFIELD`。

捕获 bug

有一些函数假定在特定条件下被调用，或者假定在特定条件下不被调用。内核使用

译注 4：指每个体系都有的一个目录。每一 CPU（per-CPU）、每一 netmask（per-netmask）等都是相同用法。

BUG_ON 和 BUG_TRAP 宏捕获引起这类条件不满足的地方。当传给 BUG_TRAP 的输入条件为假时，内核会打印出警告消息（warning message）。对于 BUG_ON 则打印出错误消息，然后内核 panic。

统计数据

收集特定条件下发生的统计数据的功能是一种良好的习惯，如缓存查询成功和失败次数、内存分配成功和失败次数等等。对于每种收集统计数据的网络功能，本书会列出每个计数器，并给出描述。

测量时间

内核经常需要测量自某一时刻起已经过了多少时间。例如，一个进行 CPU 密集运算（CPU-intensive）任务的函数，通常会在特定时间之后释放 CPU。当其被重新调度而执行时，就会继续其工作。即使内核支持内核抢占（kernel preemption），在内核代码中，这一点也特别重要。网络代码中常见的例子就是实现垃圾收集的函数。在本书中会看到很多类似的函数。

内核空间中时间的流逝采用滴答（tick）来计算。一个滴答就是两个连续到期的定时器中断之间的时间长度。定时器会负责各种不同任务（对此我们没兴趣），而且每秒会固定到期 HZ 次。HZ 是一个变量，由体系依赖代码进行初始化。例如，在 i386 机器上，它被初始化为每秒 1000 次。也就是说，当 Linux 在 i386 系统上运行时，定时器每秒中断 1000 次。因此，两次连续到期中断之间的时间长度就是一毫秒。

每次定时器到期时，就会使一个名为 jiffies 的全局变量递增。也就是说在任何时刻，jiffies 代表的是从系统引导后所经过的滴答次数，而通用值 *n**HZ 代表的就是 *n* 秒的时间。

如果一个函数所需做的工作仅是测量经过的时间，就可以把 jiffies 的值保存在一个局部变量，稍后再拿 jiffies 和当前时间戳做对此，得到一段时间间隔（以滴答数表示），以得知从测量时刻算起经过了多少时间。

下面的例子给出了一个函数，该函数必须做某种工作，但又不想让其持有 CPU 的时间超过一个滴答。当 do_something 把 job_done 设为非零值时，表示工作已完成，则此函数就可返回：

```
unsigned long start_time = jiffies;
int job_done = 0;
```

```
do {
    do_something(&job_done);
    If (job_done)
        return;
while (jiffies - start_time < 1);
```

内核代码使用jiffies的几个实例参见第十章“积压处理：process_backlog轮询虚拟函数”一节，或者第二十七章“异步清理：neigh_periodic_timer函数”一节。

用户空间工具

有多种不同的工具可用于配置Linux众多可用的网络功能。如本章开头所述，你可以通过使用这些工具对内核巧妙地处理，以便于学习以及发现做这样的修改后的影响。

下面是在本书中将会经常涉及的工具：

iputils

除了经常使用的ping命令外，*iputils*还包括arping（用于产生ARP请求）、网络路由器发现守护函数rdisc以及其他程序。

net-tools

这是一组网络工具，其中有著名的*ifconfig*、*route*、*netstat*以及*arp*，还有*ipmaddr*、*iptunnel*、*ether-wake*和*netplugd*等。

IPROUTE2

这是新一代的网络配置套件（尽管已经存在好多年了）。通过一个多用途的命令ip，这个套件可以配置IP地址以及路由，还有其他各种高级功能，如邻居协议等等。

IPROUTE2的源代码可以从*http://linux-net.osdl.org/index.php/Iproute2*下载，而其他组件可以从大多数Linux发行版本的下载服务器上下载。

大多数Linux发行版本中都默认包含这些组件。每当你不了解内核代码如何处理来自于用户空间的命令时，鼓励你看一看那个用户空间工具的源代码，以了解用户所下达的命令是如何打包并传给内核的。

在下列URL之中，你可以发现很好的文档，说明如何使用上述工具，包括邮件列表（mailing list）（注3）：

- *http://lartc.org*。

注3： 本书不谈防火墙的基础架构设计，但是在分析各种网络协议和分层时，我通常会指出防火墙挂钩hook所在之处。

- *http://www.policyrouting.org*。
- *http://www.netfilter.org*。

如果你想追踪网络代码的最新修改，要关注下列的邮件列表：

- Linux 网络开发论坛档案库（*http://oss.sgi.com/projects/netdev/archive*）。

此外，其他特定的 URL 会在相关各章中提及。

浏览源代码

Linux 内核已经变得很大，用“老朋友”grep 浏览代码，肯定不再是好主意。现在，你可以依靠各种软件，让你的内核代码之旅更为顺心。

还不知道 *cscope* 的人，我建议他们用 *cscope*，可以从 *http://cscope.sourceforge.net* 下载。这是一个简单而功能强大的工具，例如，可以搜寻函数或变量定义之处和调用之处等等。安装这个工具的过程很简单，并且你可以在网站上找到所有必需的指导。

每个人都有最喜欢的编辑器，而且可能我们多数人都是某种形式的Emacs或*vi*的爱好者(fans)。这两种编辑器都可以使用一种特殊的叫做“tag”的文件，使用户可以在源代码之间切换(cscope也使用类似的数据库文件)。你可以在内核根树(root tree)的*makefile*中使用一个同义目标(synonymous target)轻易建立类似的文件。这三个数据库TAGS、tags以及cscope.out可以分别用*make TAGS*、*make tags*以及*make cscope*来建立（注4）。

要注意，这些文件都很大，特别是 *cscope* 所用的那一个。因此，建立该文件前，要先确保你有足够的可用磁盘空间。

如果你已经在使用其他源代码浏览导航工具，那也不错。但是，如果你还没用任何一种，而且到目前为止都是这么懒，那么此时该向 *grep* 说再见了，然后花 15 分钟学习如何使用上述工具是值得的。

死代码

内核像其他大型而动态演变的软件一样，也会包括一些不再被调用的代码片段。不幸的是，代码中的注释很少会告诉你这一点。有时候，只是因为你在看的是死代码，害得你无法理解给定函数的用法或者给定变量的初始化是怎么回事。如果运气好的话，那些代

注 4： tags和TAGS文件是通过ctags工具的协助建立的。

码没有编译，并且你也猜出那些程序代码已经太陈旧了。但其他时候，你可能就没那么幸运了。

每个内核子系统都假定指派了一位或多位维护人员。然而，有些维护人员有太多代码要看，根本没有足够的空闲时间做这件事。其他时候，他们已不再有兴趣继续维护他们的子系统，但是又找不到可以替换他们的角色。因此，看到似乎在做很奇怪的事的代码，或者代码没有遵守通用而常识性的程序设计规则时，把这件事谨记在心是很有益的。

本书中，只要有意义可言，我都尽量提醒你那些没在使用的函数、变量以及数据结构。没在使用的原因可能是因为删除一项功能时被留了下来，或者是因为一项新功能而被引入，但编码尚未完成。

当功能以补丁形式提供时

内核网络代码一直在不断的进化。不但会整合新功能，而且有时也会对现存的组件进行设计变更，以达到更为模块化和更高的性能。显然，这使得Linux非常有吸引力，以作为网络应用产品（路由器、交换机、防火墙以及负载平衡器等等）所需的嵌入式操作系统。

因为任何人都可以为Linux内核开发一项新的功能，或者拓展或重新实现一项现存的功能，对任何“开放”的开发人员而言，当看到他的作品成为正式内核版本的一部分时，都会是一件最振奋人心之事。然而，有时即使一个项目的功能很有价值而且实现得很好，也不可能整合进来，或者可能要花很长一段时间。常见的原因包括：

- 代码没有按照 *Documentation/CodingStyle* 的原则来写。
- 另一个重要项目提供相同机能已经有一段时间了，而且已经获得Linux社区和维护相关内核领域的重要内核开发人员的认同。
- 与另一个内核组件有太多重叠之处。在这种情况下，最佳做法就是把冗余的功能删除掉，可能的地方使用现存的功能，或者拓展现有组件使其能用于新的环境中。这种情况强调了模块化的重要性。
- 项目的大小以及在快速变动的内核中进行维护所需的工作量，可能导致新项目的开发人员将它保持为独立的补丁，并且仅仅偶尔才会发行新版本。
- 这种功能只会用在特定场合下，在通用操作系统中并非是必需的。在这种情况下，采用独立的补丁通常是最佳的解决方案。
- 全面的设计可能无法满足某些重要的内核开发人员。这些专家通常在心中都有大蓝

图，关心内核现在的发展以及未来的发展方向。通常他们要求在设计上进行修改，使得该项功能能适应内核的需求。

某些时候，功能间的重叠很难完全消除，例如，也许是因为一项功能太过灵活，其不同的用法只有在日后才会一目了然。例如，防火墙在网络协议栈中的几个地方都有挂钩。因此，就没有必要实现过滤或者为数据封包标明传输方向等的其他功能，只要简单地依赖防火墙就行了。当然，这会产生相关性（例如，如果路由子系统想使得流量符合特定准则，内核就必须包括对防火墙的支持）。此外，只要其他内核功能有合理的增强请求，防火墙的维护者必须随时接受。然而，这种折中通常是有益的：减少冗余代码意味着有更少的bug，代码维护就越简单，代码路径也会简化，诸如此类。

最近的功能重叠清理的实例就是在内核2.6版中，把路由代码所支持的无态NAT（stateless Network Address Translation，无态网络地址翻译）给删除掉。开发人员认识到把有态NAT的支持放在防火墙中更为灵活，因此继续维护无态NAT程序就没有价值了（尽管无态NAT比较快，耗用的内存比较少）。注意，Netfilter所需的一个新模块也随时会被写，必要时会提供无态NAT的支持。

第二章

关键数据结构

Linux 网络代码中有些关键数据结构随处可见。在阅读本书以及直接研究源代码时，必须了解这些数据结构里的字段。当然，逐一研究这些数据结构的字段比起解析函数更加乏味，但这是很重要的基础。传奇软件工程师 Frederick P. Brooks 曾说过："给我看你的数据。"

本章介绍下列数据结构，同时说明操作这些数据结构的一些函数和宏。

`struct sk_buff`

一个封包就存储在这里。所有网络分层都会使用这个结构来储存其报头、有关用户数据的信息（有效载荷），以及用来协调其工作的其他内部信息。

`struct net_device`

在Linux内核中每种网络设备都用这个数据结构表示，包括软硬件的配置信息。第八章详细描述了`net_device`数据结构分配的时机以及分配的方式。

Linux网络所用的另一个关键数据结构是`struct sock`，用于存储套接字的网络信息。因为本书不涉及套接字，故本章没有把`sock`包含在内。

套接字缓冲区：sk_buff 结构

这可能是 Linux 网络代码中最重要的数据结构，代表已接收或正要传输的数据的报头。此结构定义在 *<include/linux/skbuff.h>* 头文件中，由巨大的变量堆（heap）组成，试图满足所有人的所有需求。

在内核的进化历程中，这个结构历经多次变动，不但新增了选项，同时也重组了现存的字段，使得布局更为清晰。其字段可粗略划分为下列几种类型：

- 布局（Layout）。
- 通用（General）。
- 功能专用（Feature-specific）。
- 管理函数（Management functions）。

多个不同的网络分层（MAC或L2分层上的另一种链路层协议、L3的IP以及L4的TCP或UDP）都会使用这个结构，而且当该结构从一个分层传到另一个分层时，其不同的字段会随之发生变化。L4在传给L3之前会附加一个报头，而L3在传给L2之前又会加上其自己的报头。附加报头比起把数据从一个分层拷贝到另一个分层更有效率。由于要在一个缓冲区开端新增空间——也就是要改变指向该缓冲区的变量——是一种复杂的运算，内核提供了skb_reserve函数（本章稍后会说明）来执行这一操作。所以，当缓冲区往下传经每个分层时，每层的协议首先做的就是调用skb_reserve函数，为该协议的报头预留空间（注1）。在稍后的一节“数据预留和对齐：skb_reserve、skb_put、skb_push以及skb_pull”中，我们会看到一个实例，说明缓冲区穿越各个分层时，内核如何确保缓冲区头部有足够的空间预留下来，允许每个分层都能加上其报头。

当缓冲区往上传经各个网络分层时，每个源自于旧分层的报头就不再有用处。例如，L2报头只由处理L2协议的设备驱动程序使用，所以对L3而言并无用处。不过，并没有把L2的报头从缓冲区中删除，而是把指向有效载荷开端的指针向前移到L3报头的开端，这样做只需要很少的CPU周期。

本节剩余部分说明有关条件（可选）字段的基本原则，然后讨论刚才所列出的每种字段类型。

网络选项以及内核结构

粗略阅读TCP/IP说明书或内核配置后可知，网络代码提供了大量的选项，虽然有用，但不一定总是需要，如防火墙、多播（multicasting）以及其他功能。这些选项大多数要求在内核数据结构中附加其他字段。因此，sk_buff是被C预处理程序的#ifdef指示附加的字段。例如，在sk_buff定义底端附近可以发现：

```
struct sk_buff {
    ... ... ...
#ifdef CONFIG_NET_SCHED
    __u32    tc_index;
#ifdef CONFIG_NET_CLS_ACT
```

注1：设备驱动程序也使用skb_reserve对齐输入帧的IP报头。参见第十章。

```
    __u32     tc_verd;
    __u32     tc_classid;
#endif
#endif
}
```

这表示只有在编译期间定义了CONFIG_NET_SCHED符号，也就是管理员或自动安装工具已经使能了某种版本的make config的正确选项，字段tc_index才会是此数据结构的一部分（就此例而言，就是“Device Drivers → Networking support → Networking options → QoS and/or fair queueing”）。

前述的例子实际上是两个嵌套选项：只有“QoS and/or fair queueing”的支持存在，CONFIG_NET_CLS_ACT（packet classifier，封包分类器）所用的那些字段才会被引入。

顺便提一下，应注意到QoS选项不能编译成一个模块。其原因在于内核编译之后，开启该选项所得的多数结果为不可逆的。一般而言，任何引起内核数据结构改变的选项（如把tc_index字段添加到sk_buff结构），都不适合编译成一个模块。

通常你会想知道*make config*或其变量的哪个编译选项和特定的#ifdef符号相配，以了解一个代码区块什么时候会包含到内核中。在2.6版本内核中，找出这种关联性的最快方式，就是在那些分散在源码树中的kconfig文件中（每个目录都有一个）寻找该符号。对2.4版本内核而言，可以参考*Documentation/Configure.help*文件。

布局字段

sk_buff有些字段只是为了方便搜寻以及组织数据结构本身。内核在一个双向链表中维护所有的sk_buff结构，但是该表的组织比传统的双向链表更为复杂。

像任何双向链表一样，通过每个sk_buff结构中的next和prev字段实现联系。next字段指向前，而prev指向后。但是，这个表还有另一项必要需求：每个sk_buff结构必须能够迅速找出整个表的头。为了实现这项必要需求，在表的开端额外增加一个sk_buff_head结构作为一种哑元元素。sk_buff_head结构是：

```
struct sk_buff_head {
    /* 这两个成员必须在最前 */
    struct sk_buff    * next;
    struct sk_buff    * prev;

    __u32          qlen;
    spinlock_t     lock;
};
```

qlen代表表中元素的数目。lock是用于防止对表的并发访问，在本章稍后的“表管理函数”一节中会予以描述。

sk_buff和sk_buff_head的前两个元素是相同的：next和prev指针。尽管sk_buff_head与sk_buff相比实在太小，但还是允许两个结构共同存在于同一个表中。另外，同样的函数也可用于操作sk_buff和sk_buff_head二者。

每个sk_buff结构都包含一个指针，指向专一的sk_buff_head结构，增加了其复杂性。这个指针的字段名称为list。图2-1有助于你了解这些数据结构的关系。

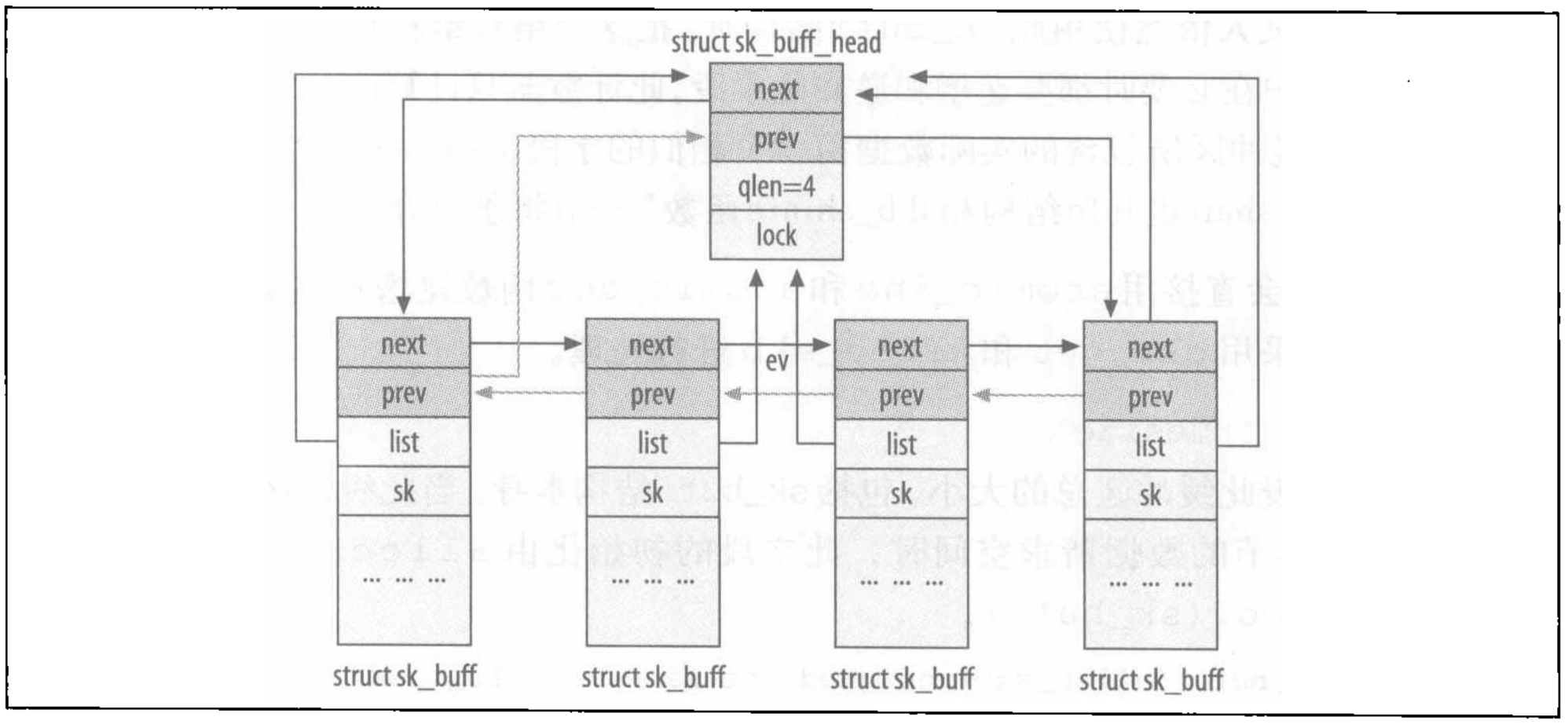


图2-1：sk_buff元素列表

sk_buff其他感兴趣的字段如下：

struct sock *sk

这是一个指针，指向拥有此缓冲区的套接字的sock数据结构。当数据在本地产生或者正由本地进程接收时，就需要这个指针，因为该数据以及套接字相关的信息会由L4（TCP或UDP）以及用户应用程序使用。当缓冲区只是被转发时（也就是说本地机器不是来源地也不是目的地），该指针就是NULL。

unsigned int len

这是指缓冲区中数据区块的大小。这个长度包括主要缓冲区（由head所指）的数据以及一些片断（fragment，注2）的数据。当缓冲区从一个网络分层移往下一个网络分层时，其值就会变化，因为在协议栈中往上移动时报头会被丢弃，但是往下

注2：参见第二十一章有关片段缓冲区（fragmented buffer）的讨论。

移动时报头就会添加进来。len也会把协议报头算在内，如“数据预留和对齐：skb_reserve、skb_put、skb_push以及skb_pull”一节中的图2-8所示。

unsigned int data_len

与len不同的是，data_len只计算片段中的数据大小。

unsigned int mac_len

MAC报头的大小。

atomic_t users

这是引用计数，或者使用这个sk_buff缓冲区的实例的数目。这个参数的主要用途是避免在某人依然使用此sk_buff结构时，把这个结构给释放掉。因此，此缓冲区的每个用户在必要时都要递增和递减此字段。此计数器只计算sk_buff数据结构的用户，此缓冲区所包含的实际数据由一个相似的字段（dataref）所包含；本章稍后的“skb_shared_info结构和skb_shinfo函数”一节将予以介绍。

users有时会直接用atomic_inc和atomic_dec函数递增和递减，但在大多数情况下，采用skb_get和kfree_skb进行处理。

unsigned int truesize

此字段代表此缓冲区总的大小，包括sk_buff结构本身。当此缓冲区得到所分配的len个字节的数据请求空间时，此字段的初始化由alloc_skb函数设置成len+sizeof(sk_buff)。

```
struct sk_buff *alloc_skb(unsigned int size,int gfp_mask)
{
    ... ... ...
    skb->truesize = size + sizeof(struct sk_buff);
    ... ... ...
}
```

每当skb->len的值增加时，此字段就会得到更新。

unsigned char *head

unsigned char *end

unsigned char *data

unsigned char *tail

这些字段代表缓冲区的边界以及其中的数据。当每一分层为其工作而准备缓冲区时，可能会为一个报头或更多的数据分配更多的空间。head和end指向已分配缓冲区空间的开端和尾端，而data和tail则指向实际数据的开端和尾端。参见图2-2所示。然后，该分层可以把head和data之间的空隙填上一个协议报头，或者以新数据填入tail和end之间的空隙。在稍后的“分配内存：alloc_skb和dev_alloc_skb”一节中就会知道，图2-2右边的缓冲区在底端包含一个附加报头。

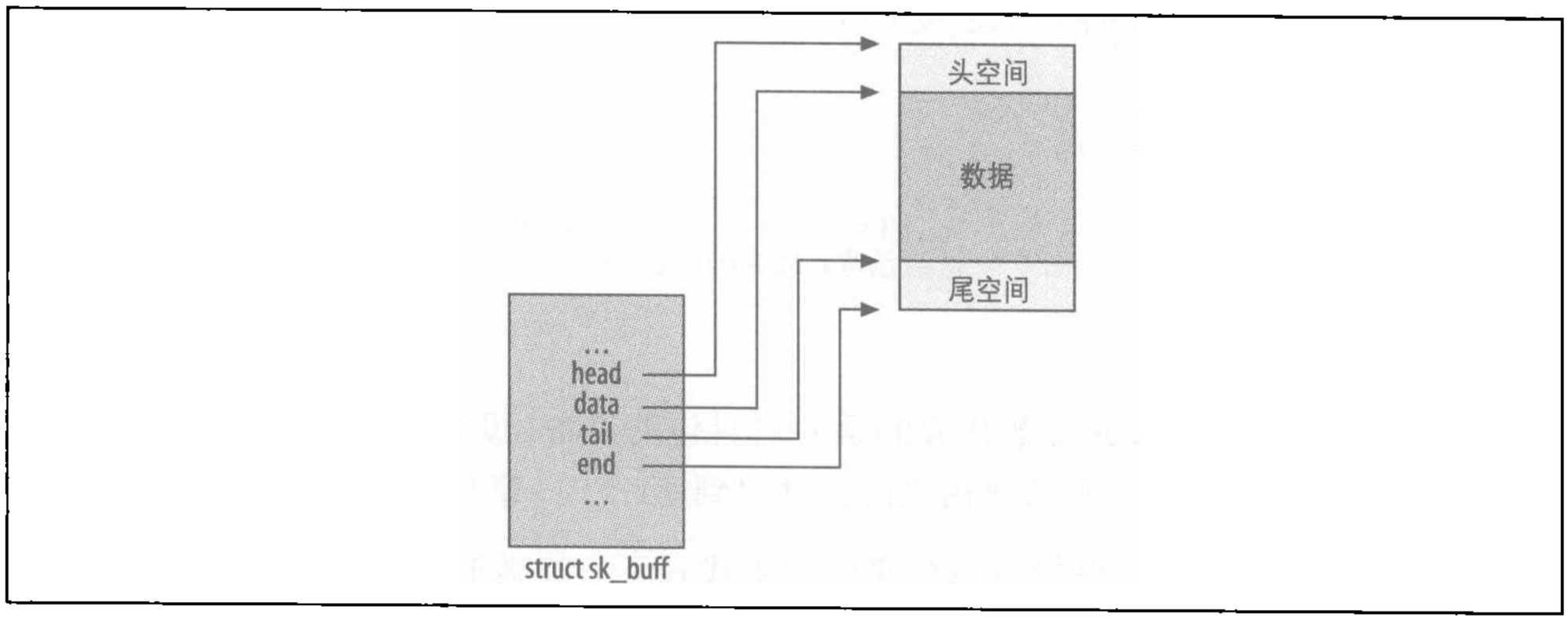


图 2-2：head/end 与 data/tail 指针

`void (*destructor)(...)`

此函数指针可以被初始化为一个函数，当此缓冲区被删除时，可完成某些工作。当此缓冲区不属于一个套接字时，destructor 通常不会被初始化。当此缓冲区属于一个套接字时，通常设成 `sock_rfree` 或 `sock_wfree`（分别由 `skb_set_owner_r` 和 `skb_set_owner_w` 初始化函数设置）。这两个 `sock_xxx` 函数可用于更新套接字队列中所持有的内存。

通用字段

这一节讨论 `sk_buff` 的主要字段，而这些字段都与特定内核功能无关：

`struct timeval stamp`

通常只对一个已接收的封包才有意义。这是一个时间戳记，用于表示封包何时被接收，或者有时用于表示封包预定传输的时间。此字段由 `netif_rx` 函数用 `net_timestamp` 设置，而该函数在接收每个封包之后由设备驱动程序调用，对此在第二十一章给出了描述。

`struct net_device *dev`

此字段描述一个网络设备，其类型（`net_device`）将在本章稍后详细说明。由 `dev` 所代表的设备的角色，依赖于存储在该缓冲区内的封包是即将传输的还是刚被接收的而定。

当接收到一个封包时，设备驱动程序会用代表此接收接口的数据结构的指针更新此字段，如下列取自 `vortex_rx` 的代码片段所示，当接收到一帧时，3c59x 系列的以太网卡的驱动程序就会调用该函数（在 *drivers/net/3c59x.c* 中）：

```
static int vortex_rx(struct net_device *dev)
{
            ... ... ...
        skb->dev = dev;
            ... ... ...
        skb->protocol = eth_type_trans(skb, dev);
        netif_rx(skb); /* 把封包传到较高的分层 */
            ... ... ...
}
```

当传输一个封包时，此参数代表的是发送封包的设备。设置此值的代码要比接收封包的代码更为复杂，所以把相关讨论推迟到第二十一章和第三十五章进行。

有些网络功能允许一些设备按组集合起来代表专一的虚拟接口（也就是没有和硬设备直接相关联），由一个虚拟设备驱动程序提供接口服务。当该设备驱动程序被调用时，dev参数会指向此虚拟设备的net_device数据结构。驱动程序会从其群组中选择一个特定设备，然后把dev参数改为指向该设备的net_device数据结构。因此，这些情况下，在封包处理期间传输设备的指针可能会变化。

struct net_device *input_dev

这是已被接收的封包所源自的设备。当封包是由本地产生时，其为NULL指针。就Ethernet设备而言，此字段在eth_type_trans中初始化（参见第十章和第十三章）。此字段主要由流量控制（Traffic Control）使用。

struct net_device *real_dev

此字段只对虚拟设备有意义，代表的是与虚拟设备所关联的真实设备。例如，Bonding和VLAN接口使用该字段，用以记下真实设备的输入流量是从什么地方接收而得的。

union {...} h
union {...} nh
union {...} mac

这些是指向TCP/IP协议栈的协议报头的指针：h针对L4、nh针对L3，而mac针对L2。每个字段都指向一个由各种结构组成的联合（union），每个协议的结构都由内核中该层来解释。例如，h是一个联合，内核所解释的每个L4协议的报头在h中都有一个字段。每个联合都有一个名为raw的成员用于初始化。而后续所有的访问都是通过协议指定的成员。

当接收一个数据封包时，负责处理第*n*层报头的函数，会从第*n* − *1*层接收一个缓冲区，而该缓冲区的skb->data指向第*n*层报头的开端。处理第*n*层的函数会为该层初始化适当的指针（例如，L3的处理函数的skb->nh），用以保存skb->data字段，因为在下一层进行处理时，skb->data会设成缓冲区内另一个不同的偏移

量，这个指针的内容就会丢失。接着，此函数完成第n层的处理，把封包传给第$n+1$层的处理函数前，先更新skb->data，使其指向第n层报头的尾端，也就是第$n+1$层报头的开始（参见图2-3）。

传送一个封包就是此过程的逆过程，而所增加的复杂度就是在每一层加上一个新的报头。

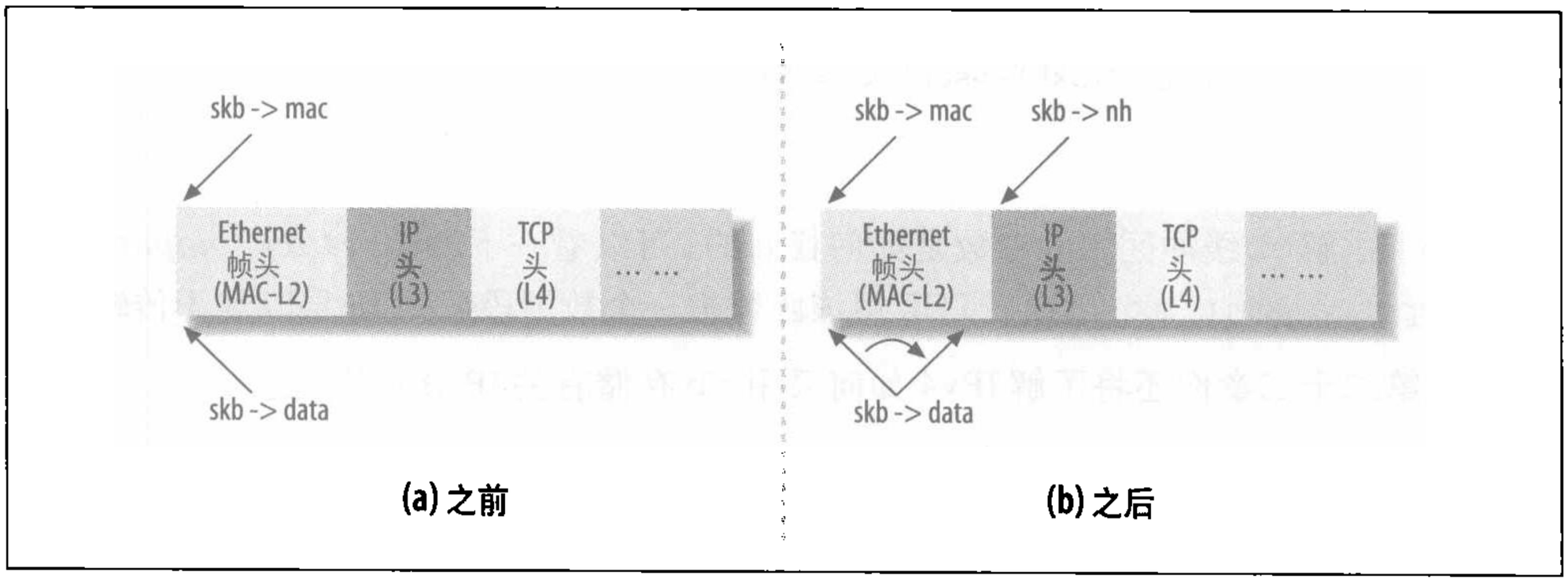


图2-3：从第二层移往第三层时报头指针的初始化

struct dst_entry dst

这个结构由路由子系统使用。由于这个数据结构相当复杂，需要了解其他子系统如何工作的知识，因此，推迟到第七部分再予以描述。

char cb[40]

这是一个“控制缓冲区（control buffer）”，或者说是私有信息的存储空间，为每一层内部使用起维护的作用。该字段在sk_buff结构内静态分配（目前的大小是40个字节），而且容量足以容纳每个层所需的私有数据。在每一层的代码中都是通过宏进行访问的，这样使代码更具有可读性。例如，TCP使用这个空间来存储一个tcp_skb_cb数据结构，该数据结构定义在*include/net/tcp.h*中：

```
struct tcp_skb_cb {
    ... ... ...
    __u32          seq;          /* 起始序列编号 */
    __u32          end_seq;      /* SEQ + FIN + SYN + datalen*/
    __u32          when;         /* 用于计算rtt's*/
    __u8           flags;        /* TCP报头标志 */
    ... ... ...
};
```

以下是TCP代码用于访问该结构的宏。该宏由一些简单的指针组成：

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

以下是TCP子系统在段的收条中填写此结构的实例：

```
int tcp_v4_rcv(struct sk_buff *skb)
{
        ... ... ...
        th = skb->h.th;
        TCP_SKB_CB(skb)->seq = ntohl(th->seq);
        TCP_SKB_CB(skb)->end_seq = (TCP_SKB_CB(skb)->seq + th->syn + th->fin +
                                    skb->len - th->doff * 4);
        TCP_SKB_CB(skb)->ack_seq = ntohl(th->ack_seq);
        TCP_SKB_CB(skb)->when = 0;
        TCP_SKB_CB(skb)->flags = skb->nh.iph->tos;
        TCP_SKB_CB(skb)->sacked = 0;
        ... ... ...
}
```

为了了解cb缓冲区中的参数是如何找回的，可以看一下*net/ipv4/tcp_output.c*中的tcp_transmit_skb函数。TCP用该函数把一个数据段压入IP层以便于传输。

在第二十二章你还将了解IPv4如何采用cb存储有关IP分片的信息。

unsigned int csum

unsigned char ip_summed

这些代表校验和(checksum)以及相关联的状态标识。其用法将在第十九章予以描述。

unsigned char cloned

当一个boolean标识置位时，表示该结构是另一个sk_buff缓冲区的克隆。参见稍后的“缓冲区的克隆和拷贝”一节。

unsigned char pkt_type

此字段会根据帧的L2目的地址进行类型划分。可能的取值列于*include/linux/if_packet.h*中。对Ethernet设备而言，此参数由函数eth_type_trans进行初始化，在第十三章中将予以描述。

可能被分派的主要值如下：

PACKET_HOST

已接收帧的目的地址，即接收接口；换句话说，封包已到达其目的地。

PACKET_MULTICAST

已接收的帧的目的地址是该接口已注册的多播地址之一。

PACKET_BROADCAST

已接收的帧的目的地址是接收接口的广播地址。

PACKET_OTHERHOST

已接收的帧的目的地址不属于与该接口相匹配的地址（单播（unicast）、多播以及广播）；因此，如果转发机制使能，该帧将不得不被转发，否则就会被丢弃。

PACKET_OUTGOING

封包正被发送。此标识的用户是Decnet协议，并且给每个网络分流器一个输出封包副本的函数（参见第十一章的dev_queue_xmit_nit）。

PACKET_LOOPBACK

封包正传送至回环设备（loopback device）。由于此标识，当处理回环设备时，内核可以跳过一些真实设备所需的操作。

PACKET_FASTROUTE

用Fastroute功能路由封包。2.6版本内核不再支持Fastroute。

第十三章详细描述了这些值如何根据L2目的地址而设置。

__u32 priority

此字段表示正被传输或转发的封包QoS（Quality of Service，服务质量）等级。如果该封包由本地产生，套接字层会定义优先级的值。相反的，如果此封包正被转发，则函数rt_tos2priority（由ip_forward函数调用）会根据IP报头本身的ToS（Type of Service，服务类型）字段的值而定义此字段的值。此参数的值和第十八章所描述的DSCP（DiffServ Code Point，DiffServ码点）无关。将在第二十章的“ip_forward函数”一节中讨论其角色。

unsigned short protocol

从L2层的设备驱动程序的角度来看，就是用在下一个较高层的协议。典型的协议有IP、IPv6以及ARP；完整的列表可以在*include/linux/if_ether.h*中找到。由于每种协议都有自己的函数处理例程用来处理输入的封包，因此，驱动程序使用这个字段通知其上层该使用哪个处理例程。每个驱动程序会调用netif_rx用来启动上面的网络分层的处理例程，所以，在该函数被调用前protocol字段必须初始化。更多的细节参见第十章和第十三章。

unsigned short security

这是封包的安全级。最初引入这个字段是为了IPsec，但是现在已不再使用了。

功能专用字段

Linux内核是模块化的，允许你选择要包含什么及要省略什么。因此，只有当内核编译为支持特定功能，如防火墙（Netfilter）或QoS，某些字段才会包含在sk_buff数据结构中：

```
unsigned long nfmark
__u32 nfcache
__u32 nfctinfo
struct nf_conntrack *nfct
unsigned int nfdebug
struct nf_bridge_info *nf_bridge
```

这些参数由Netfilter（防火墙代码）使用，更明确地讲就是指由内核选项“Device Drivers → Networking support → Networking options → Network packet filtering”及其两个子选项“Network packet filtering debugging”和“Bridged IP/ARP packets filtering”所用。

```
union {...} private
```

这个联合由HIPPI（High Performance Parallel Interface，高性能并行接口）使用。相关联的内核选项为“Device Drivers → Networking support → Network device support → HIPPI driver support”。

```
__u32 tc_index
__u32 tc_verd
__u32 tc_classid
```

这些参数由流量控制功能使用，更明确地讲就是指由内核选项“Device Drivers → Networking support → Networking options → QoS and/or fair queueing”及其子选项“Packet classifier API”使用。

```
struct sec_path *sp
```

由IPsec协议组使用，以记录转换信息。

管理函数

内核通常提供许多很短、很简单的函数，用以操作sk_buff元素或元素列表。在图2-4的协助下，将描述其中最重要的几个函数。首先，我们来看用于分配和释放缓冲区的函数，然后是用于在帧头部或尾部预留空间的操作指针（即skb->data）函数。

如果你查看一下*include/linux/skbuff.h*和*net/core/skbuff.c*文件，就会注意到几乎所有函数都有两个版本，名称类似do_something和__do_something。通常来讲，第一个是包裹函数，增加了额外的合理性检查或者在调用第二个函数前后加入上锁机制。内部所用的__do_something形式通常不会被直接调用（除非满足特定条件，例如，上锁条件或命名）。违反规则时通常都是写得很差的函数，而最终都会被修正。

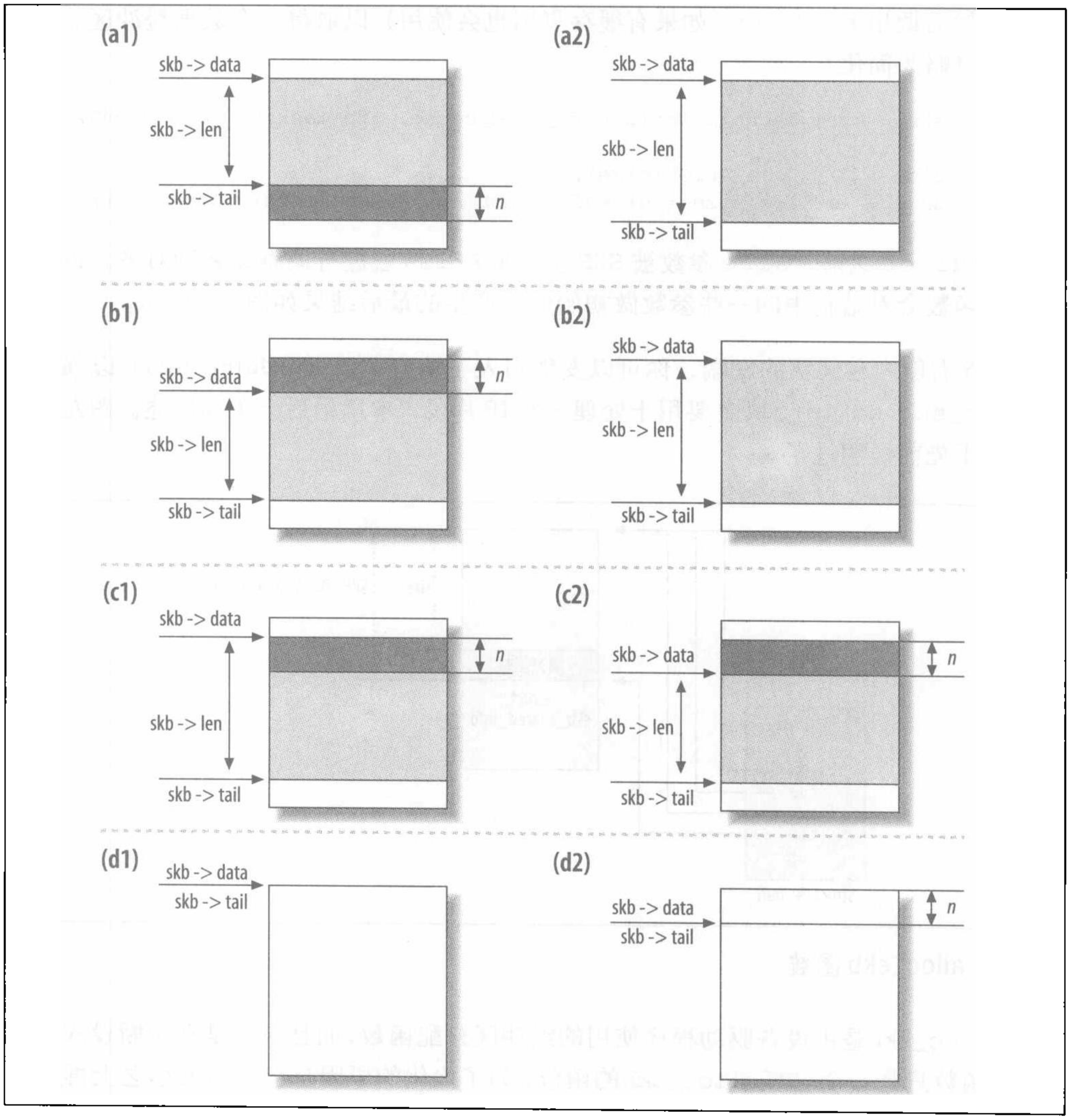


图2-4：之前和之后：（a）skb_put、（b）skb_push、（c）skb_pull及（d）skb_reserve

分配内存：alloc_skb和dev_alloc_skb

定义在*net/core/skbuff.c*中的alloc_skb是分配缓冲区的主要函数。我们已经知道，数据缓冲区和报头（sk_buff数据结构）是两种不同的实例，也就是说，建立一个缓冲区会涉及到两次内存分配（一个是分配缓冲区，而另一个是分配sk_buff结构）。

alloc_skb通过调用kmem_cache_alloc函数，从一个缓存中取得一个sk_buff数据

结构，然后调用kmalloc（如果有缓存可用也会使用）以取得一个数据缓冲区。其代码如下（略为简化）：

```
skb = kmem_cache_alloc(skbuff_head_cache, gfp_mask & ~_ _GFP_DMA);
... ... ...
size = SKB_DATA_ALIGN(size);
data = kmalloc(size + sizeof(struct skb_shared_info), gfp_mask);
```

调用kmalloc之前，size参数被SKB_DATA_ALIGN宏进行调整以强制对齐。返回之前，此函数会对结构中的一些参数做初始化，产生的最后结果如图2-5所示。

在图2-5右侧内存区块的底端，你可以发现引入了填充区域（padding area）以强制对齐。skb_shared_info块主要用于处理一些IP片段，本章稍后会予以描述。图左侧的字段已于先前说明过了。

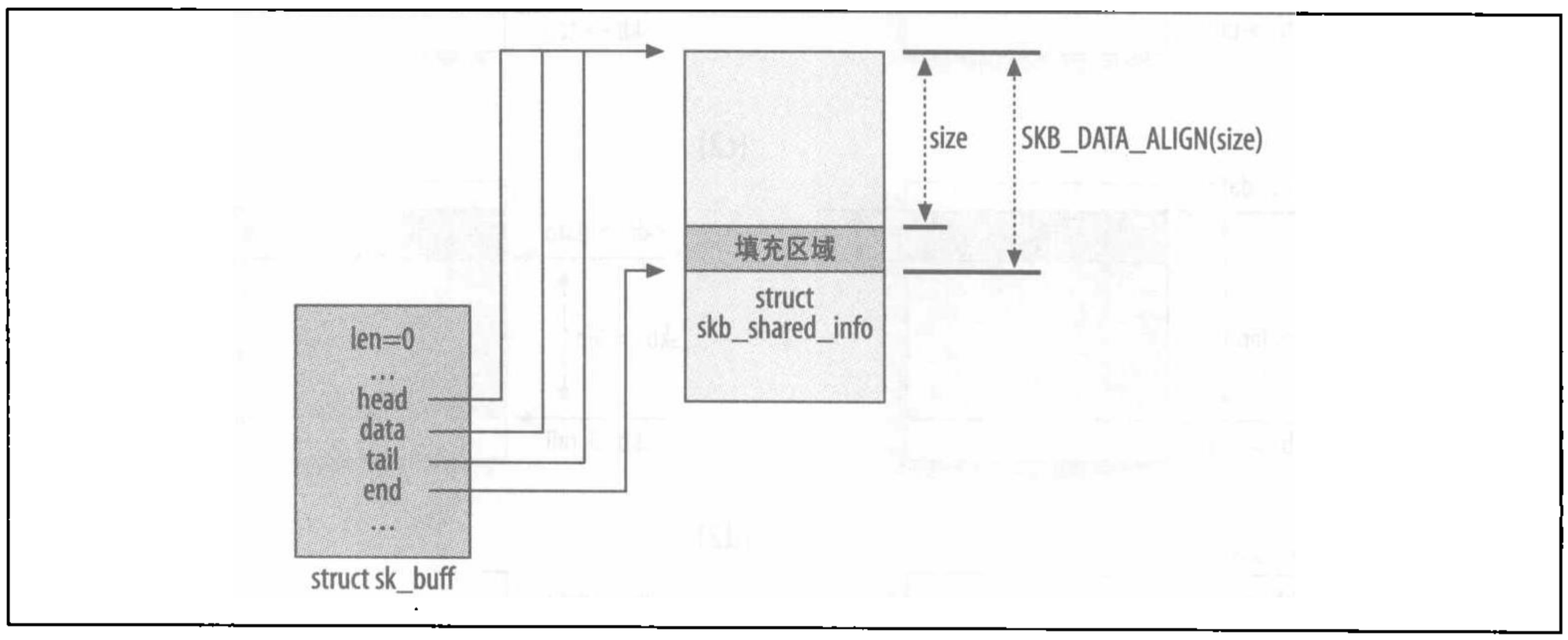


图2-5：alloc_skb函数

dev_alloc_skb是由设备驱动程序使用的缓冲区分配函数，而且应该是在中断模式中执行。此函数只是一个包裹alloc_skb的函数，为了优化的原因在申请的大小之上再加16个字节。而且因为此函数是由中断事件处理函数调用的，所以会要求原子操作(GFP_ATOMIC)：

```
static inline struct sk_buff *dev_alloc_skb(unsigned int length)
{
    return __dev_alloc_skb(length, GFP_ATOMIC);
}
static inline
struct sk_buff *__dev_alloc_skb(unsigned int length, int gfp_mask)
{
    struct sk_buff *skb = alloc_skb(length + 16, gfp_mask);
    if (likely(skb))
            skb_reserve(skb, 16);
```

```
        return skb;
    }
```

当没有体系结构说明（architecture-specific）定义时，__dev_alloc_skb的定义就是默认的。

释放内存：kfree_skb 和 dev_kfree_skb

这两个函数会释放一个缓冲区，使其返回缓冲池（缓存）。kfree_skb 是直接由dev_kfree_skb包裹函数调用并启动的。随后由设备驱动程序定义并使用，而其名称和dev_alloc_skb类似，但其组成只有一个简单的宏，这个宏不做任何事情，只是调用kfree_skb。只有当skb->users计数器为1时（该缓冲区已无任何用户时），这个基本函数才会释放一个缓冲区。否则，就只是递减该计数器。所以，如果一个缓冲区有三位用户，则只有第三次调用dev_kfree_skb或kfree_skb时才会释放内存。

图 2-6 的流程图显示出释放一个缓冲区时的所有步骤。在第三十三章会看到，一个sk_buff 结构可以持有一次对 dst_entry 数据结构的引用。因此，当 sk_buff 结构被释放时，也必须调用dst_release以递减相关的dst_entry数据结构的引用计数值。

当 destructor 函数指针已经被初始化时，就会在这里调用（参见本章稍早的“配置字段”一节）。

图2-5看起来是一个简单的场景：一个sk_buff数据结构与另一个实际存储数据的内存块相关联。然而，如图2-5所示，在该数据区块底端的skb_shared_info数据结构可以持有一些指向其他内存片段的指针，参见第二十一章的实例。kfree_skb也会释放这些片段所持有的内存（如果存在的话）。最后，sk_buff 数据结构返回 skbuff_head_cache 缓存。

数据预留及对齐：skb_reserve、skb_put、skb_push以及skb_pull

skb_reserve会在缓冲区的头部预留一些空间（头空间），通常允许插入一个报头，或者强迫数据对齐某个边界。此函数会移动标记有效载荷开端指针data和尾端指针tail（稍早的“配置字段”一节已经讨论过）。图 2-4（d）显示出调用 skb_reserve(skb,n) 之后的结果。缓冲区分配之后，通常马上就会调用此函数，此时 data 和 tail 的值仍然相同。

如果你看一下几种 Ethernet 驱动程序之一的接收函数（例如，*drivers/net/3c59x.c* 中的 vortex_rx），你会发现，它们把任何数据存储在刚分配到的缓冲区之前，都会使用下列命令：

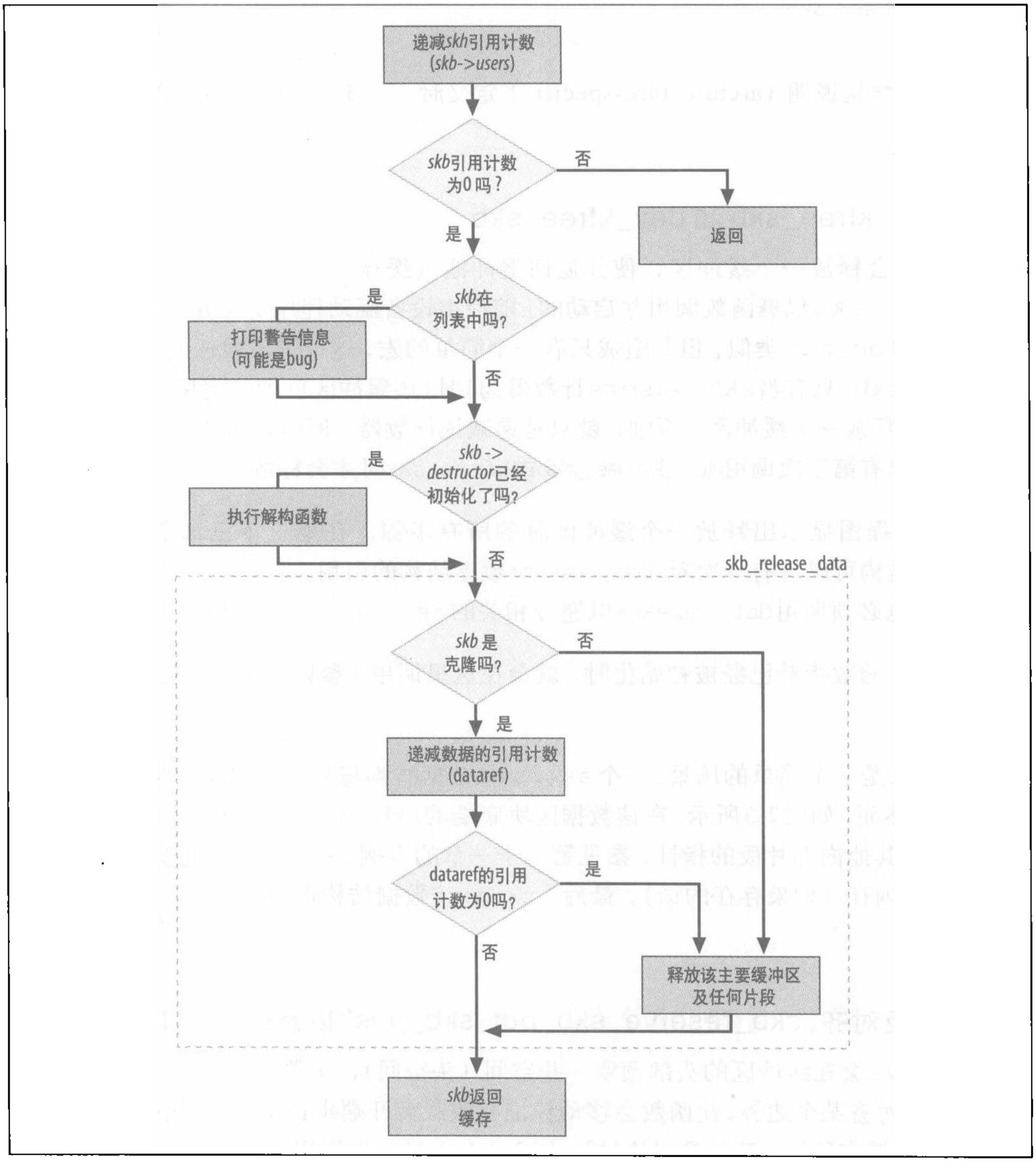


图 2-6：kfree_skb 函数

```
skb_reserve(skb, 2);      /* 把 IP 对齐在 16 字节地址边界上 */
```

由于知道要把一个带有14个字节头的Ethernet帧拷贝到缓冲区中，参数2会使缓冲区的头移动 2 个字节。这样 IP 报头就可以从缓冲区开始按照 16 字节边界对齐，并紧接在 Ethernet 报头之后，如图 2-7 所示。

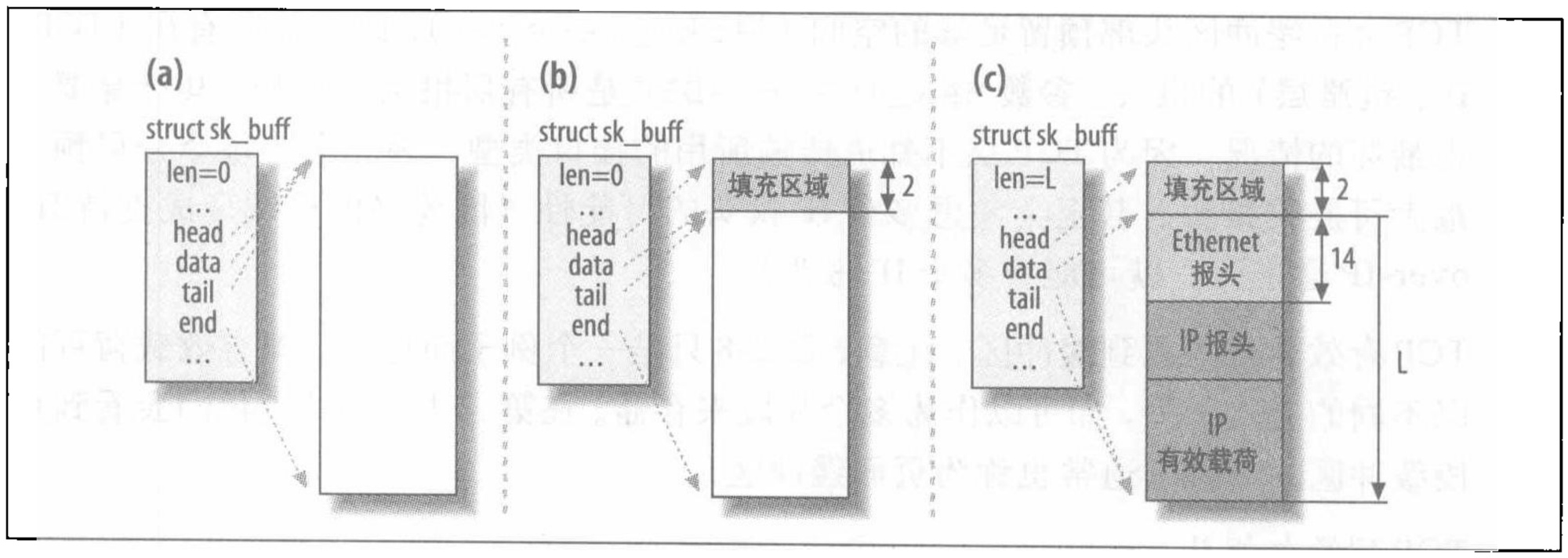


图 2-7：（a）在 skb_reserve 之前（b）在 skb_reserve 之后（c）拷贝帧到缓冲区之后

图 2-8 所示为数据在传输期间在相反方向使用 `skb_reserve` 的实例。

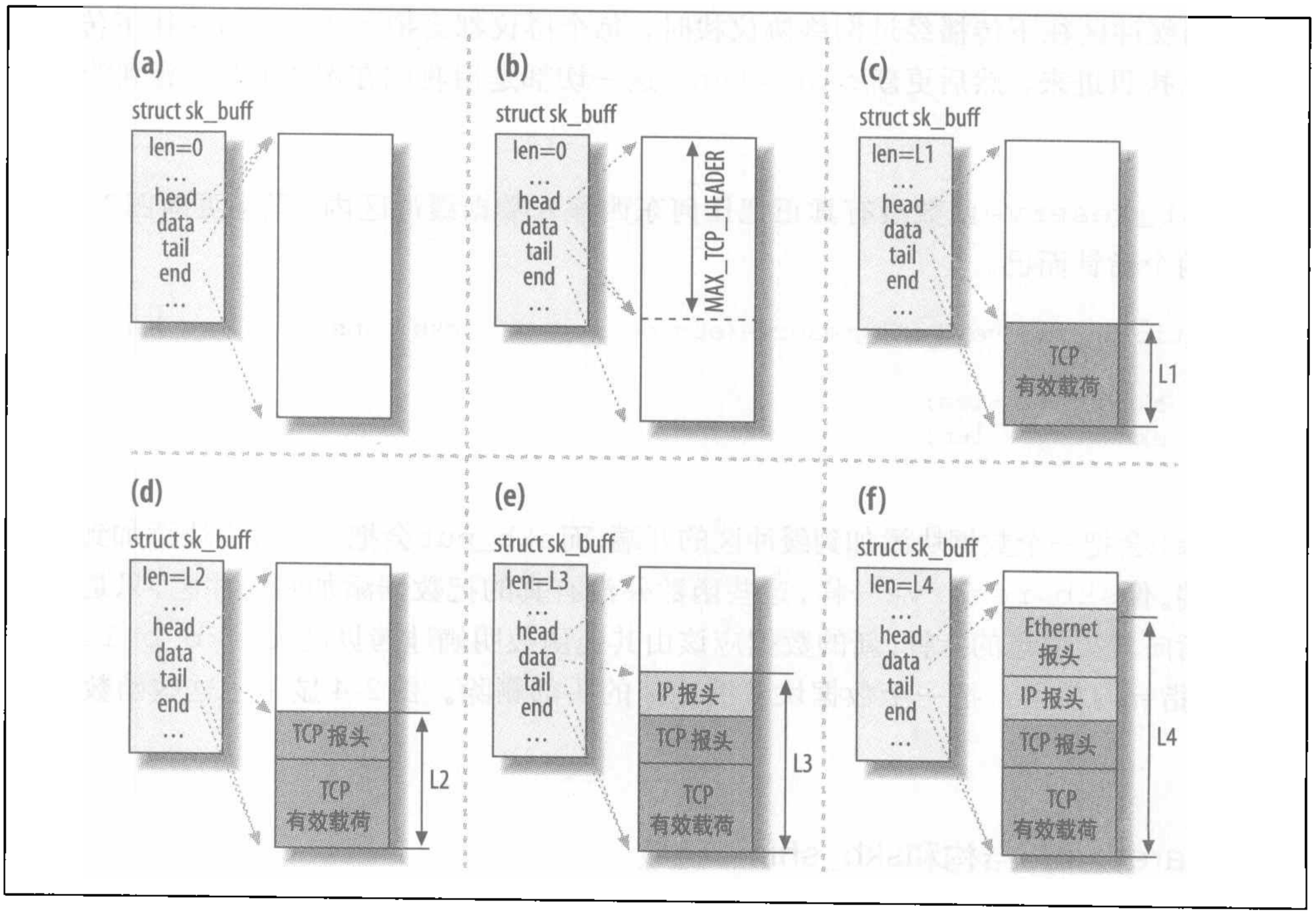


图 2-8：缓冲区穿过协议栈从 TCP 层向下到链路层

1. 当TCP被请求传输一些数据时，它会根据一些准则 [TCP mss (Maximum Segment Size，最大节段尺寸)，支持分散－聚集IO（scatter-gather IO）等等] 分配一个缓冲区。

2. TCP会在缓冲区头部预留足够的空间（用skb_reserve），以容纳所有层（TCP、IP、链路层）的报头。参数MAX_TCP_HEADER是所有层报头的总和，其计算要考虑最坏的情况：因为TCP层不知道传输所用的接口类型，因此会为每个分层预留最大可能的报头。甚至会考虑多个IP报头的可能性（因为当内核编译成支持IP-over-IP通道时，就可能有多个IP报头）。
3. TCP有效载荷拷贝到缓冲区。注意，图2-8只是一个例子而已。TCP有效载荷可能以不同的方式组织，如可以作为多个片段来存储。在第二十一章中，我们会看到片段缓冲区的形式（通常也称为页面缓冲区）。
4. TCP层添加报头。
5. TCP层把缓冲区传给IP层，IP层也同样添加其报头。
6. IP层把IP封包传给邻居层，邻居层则把链路层报头添加进来。

注意，当缓冲区往下传播经过网络协议栈时，每个协议都会把skb->data往下传，并将其报头拷贝进来，然后更新skb->len。这一切都是用我们在图2-4中所看到的函数完成的。

注意，skb_reserve函数没有真正把任何东西移入数据缓冲区内，只是更新图2-4(d)所示的两个指针而已。

```
static inline void skb_reserve(struct sk_buff *skb, unsigned int len)
{
    skb->data+=len;
    skb->tail+=len;
}
```

skb_push会把一个数据块添加到缓冲区的开端，而skb_put会把一个数据块添加到缓冲区的尾端。像skb_reserve一样，这些函数并没有真的把数据添加到缓冲区，只是简单地移动指向其头或尾的指针。新的数据应该由其他函数明确地拷贝进来。skb_pull是通过head指针向前移，把一个数据块从缓冲区的头部删除。图2-4显示了这些函数的工作方式。

skb_shared_info结构和skb_shinfo函数

如图2-5所示，数据缓冲区尾端有个名为skb_shared_info的数据结构，用以保持此数据区块的附加信息。此数据结构紧接在标记数据尾端的end指针之后。此数据结构的定义如下：

```
struct skb_shared_info {
    atomic_t        dataref;
    unsigned int    nr_frags;
```

```
    unsigned short  tso_size;
    unsigned short  tso_segs;
    struct sk_buff  *frag_list;
    skb_frag_t      frags[MAX_SKB_FRAGS];
};
```

dataref代表的是数据块的“用户”数目，下一节“缓冲区的克隆和拷贝”中会对此予以描述。nr_frags、frag_list以及frags用于处理IP片段，在第二十一章会对此予以描述。skb_is_nonlinear函数可用于检查缓冲区是否为片段，而skb_linearize（注3）可用于把几个片段合压（collapse）成一个均一的（single flat）缓冲区。合压几个片段时会引发拷贝，因而会使性能下降。

有些网络接口卡（network interface card，NIC）可以采用硬件处理一些传统上由CPU所做的任务。最常见的例子就是L3和L4层的校验和的计算。有些NIC甚至可以维护L4协议的状态机。在这里给出这段代码的原因在于，我们感兴趣的是TCP节段卸载（TCP segmentation offload，TSO）。NIC在这里实现了TCP层的子集。此功能会用到tso_size和tso_segs。

注意，sk_buff结构中没有指向skb_shared_info数据结构的字段。为了访问该结构，函数必须使用返回end指针的skb_shinfo宏：

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

例如，如下语句显示出该宏如何用于增加私有块的一个字段：

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

缓冲区的克隆和拷贝

当同一个缓冲区需要由不同消费者个别处理时，那些消费者可能需要修改sk_buff描述符的内容（指向协议报头的h和nh指针），但内核不需要完全拷贝sk_buff结构和相关联的数据缓冲区。相反，为了提高效率，内核可以克隆（clone）原始值，也就是只拷贝sk_buff结构，然后使用引用计数，以免过早释放共享的数据块。缓冲区的克隆由skb_clone函数实现。

使用克隆的情况的一个例子就是，当一个输入封包需要传递给多个接收者时，如协议处理例程和一个或多个网络分流器（参见第二十一章）。

sk_buff的克隆没有链接到任何表（list），而且也没有引用套接字的拥有者。skb->cloned字段在克隆的和原有的缓冲区内都置为1。而克隆的skb->users也置为1，使

注3：　参见第十一章“dev_queue_xmit函数”一节有关此函数的用法。

得第一次尝试删除就能成功，但是，对包含数据的缓冲区的引用数目（dataref）则会递增（因为从现在起，又有一个sk_buff数据结构指向该区）。图2-9所示是克隆缓冲区的一个实例。

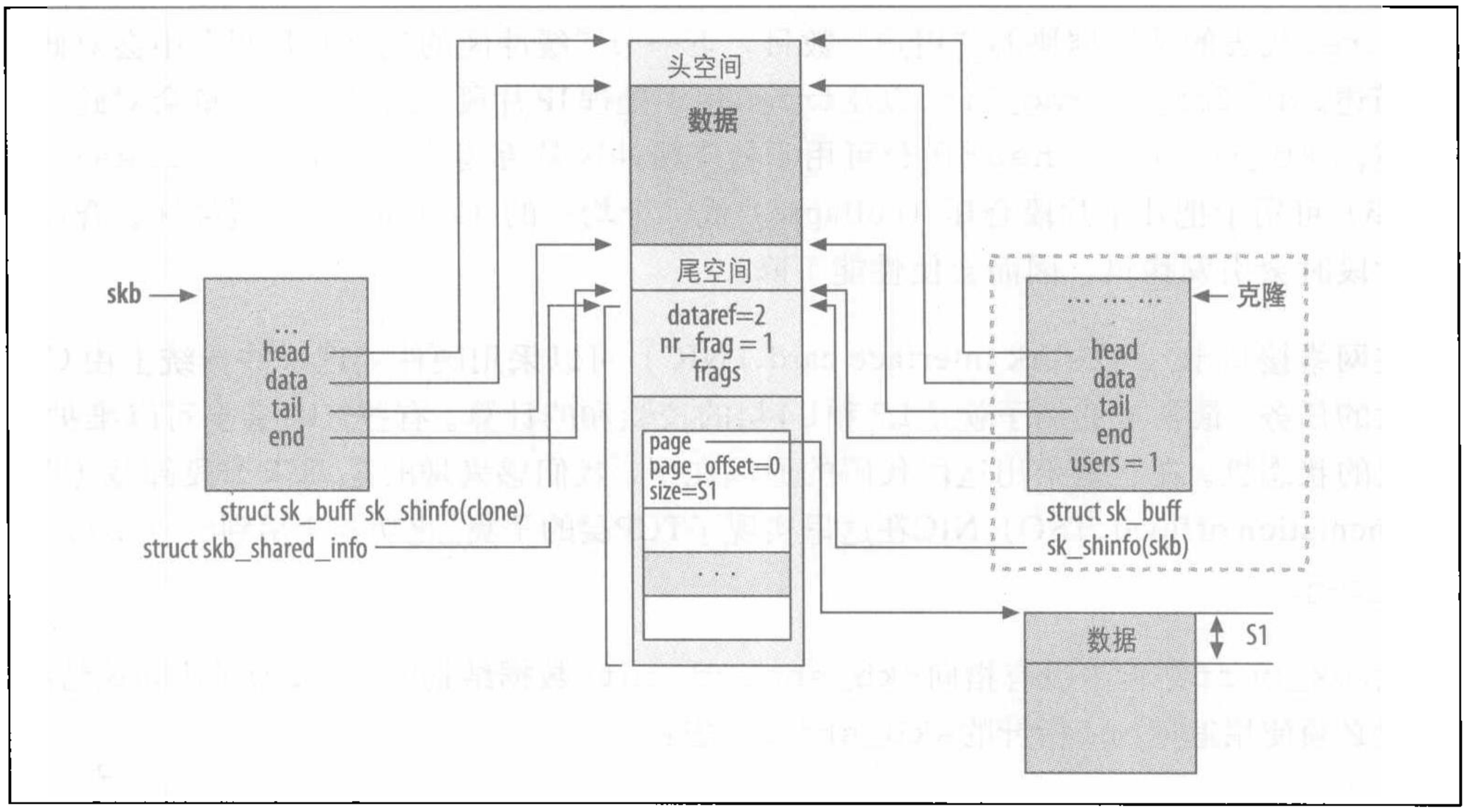


图2-9：skb_clone函数

skb_clone函数可用于检查一个skb缓冲区的克隆状态。

图2-9所示为片段缓冲区的一个实例——也就是说，这个缓冲区内有一些数据是存储在一些以frags数组链接起来的数据片段。第二十一章会说明片段缓冲区的用法，目前我们先不理会这些细节。

skb_share_check函数可用于检查引用计数skb->users，并且当users字段说明该缓冲区是共享时可以克隆该缓冲区。

当一个缓冲区被克隆时，数据区块的内容不能修改。这意味着访问该数据的代码不需要上锁机制。然而，当函数不仅需要修改sk_buff结构的内容，而且也需要修改数据时，就必须连数据区块一起克隆。在这种情况下，程序员有两种选择：当他知道他只需修改介于skb->start和skb->end的区域的数据内容时，可以使用pskb_copy只克隆该区域；当他认为他可能必须连片段数据区块的内容也跟着修改时，就必须使用skb_copy。pskb_copy和skb_copy的结果如图2-10所示。在第二十一章可以看到，skb_shared_info数据结构也可以包括一个sk_buff结构列表（链接到一个frag_list字段）。pskb_copy和

skb_copy处理该列表的方式和frags数组的处理方式相同（在图2-10中忽略了细节，使skb_copy易于理解）。

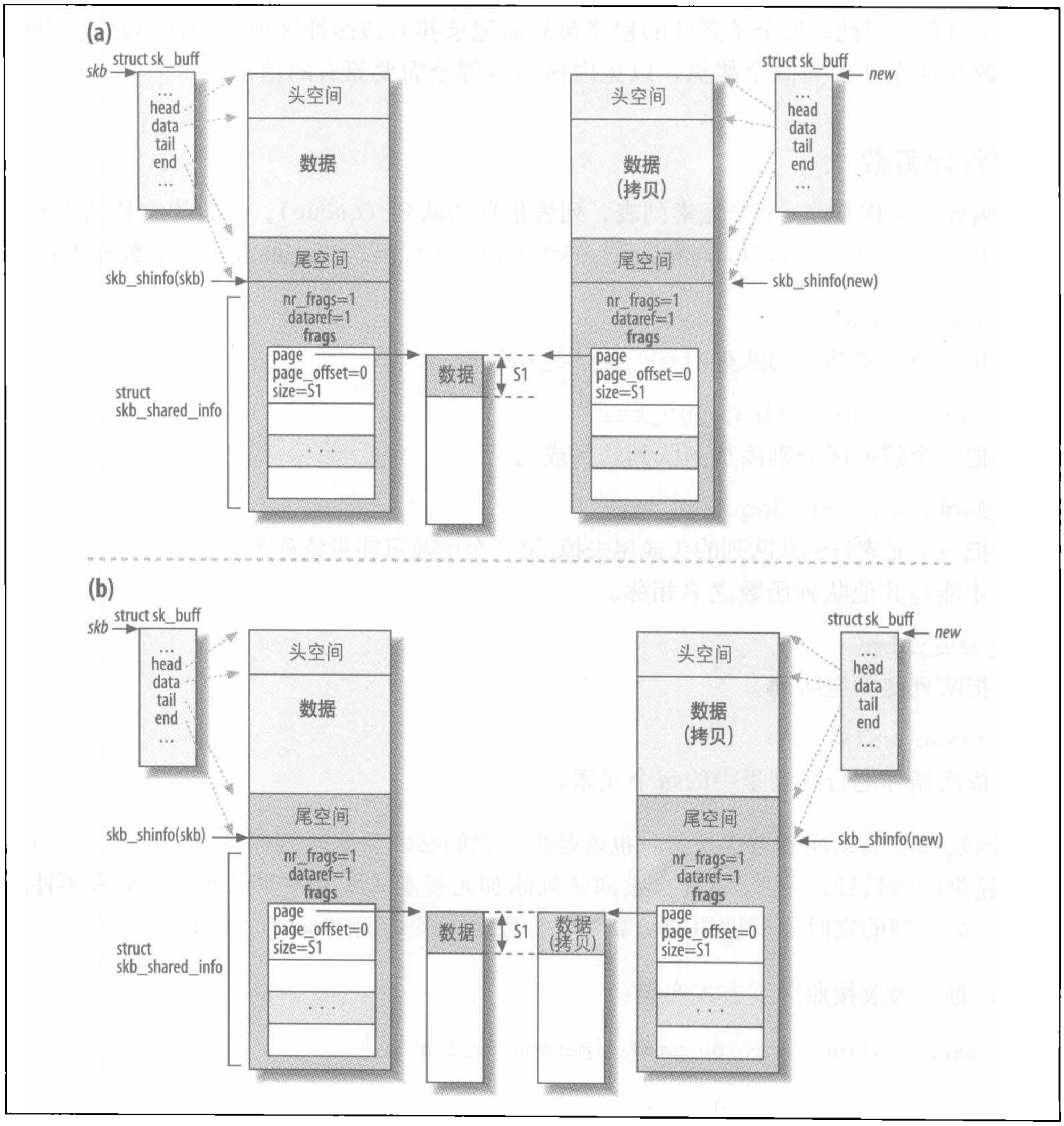


图2-10：（a）pskb_copy函数（b）skb_copy函数

此时，你可能还无法理解图2-9和图2-10的所有细节。到了本书以后的章节，特别是你读过第五部分后，一切都会变得更有意义。

讨论本书的各项主题时，有些时候我会强调给定的函数必须克隆或拷贝一个缓冲区。决

定做一个缓冲区的克隆或拷贝时，每个子系统的程序员无法预测其他内核组件（或其子系统的其他用户）是否需要该缓冲区内的原有数据。内核是非常模块化的，而且以非常动态而无法预测的方式改变，所以，每个子系统对其他子系统会对一个缓冲区做些什么都是未知的。因此，每个子系统的程序员只需记录其对该缓冲区所做的任何修改，而且在修改任何东西前先做个拷贝，以免内核其他部分需要原有的信息。

列表管理函数

这些函数会操作sk_buff元素列表，列表也称为队列（queue）。函数的完整列表参见*<include/linux/skbuff.h>*和*<net/core/skbuff.c>*。一些最常用的函数如下：

skb_queue_head_init

用一个元素为空的队列对sk_buff_head。

skb_queue_head, skb_queue_tail

把一个缓冲区分别添加到队列的头或尾。

skb_dequeue, skb_dequeue_tail

把一个元素分别从队列的头或尾去掉。第二个函数可能应该称为skb_dequeue_ head，才能与其他队列函数之名相称。

skb_queue_purge

把队列变为空队列。

skb_queue_walk

依次循环运行队列里中的每个元素。

这类函数都必须以原子方式执行；也就是说，它们必须为此队列抓住sk_buff_head结构所提供的回转锁。否则，可能会被向队列添加元素或从队列中删除元素等异步事件所中断，如到期的定时器所调用的函数，将导致竞争的情况（race condition）。

因此，每个函数按照以下方式实现：

```
static inline function_name ( parameter_list )
{
        unsigned long flags;

        spin_lock_irqsave(...);
        __function_name ( parameter_list )
        spin_unlock_irqrestore(...);
}
```

此函数内有一个包裹函数会取得回转锁，然后调用一个以两个下划线符号开头命名的函数，接着再放开该锁。

net_device结构

net_device数据结构存储着特定网络设备的所有信息。每个设备都有一个这种结构，无论是真实设备（如Ethernet NIC）或虚拟设备［如Bonding（注4）或VLAN（注5）］。本节中，接口（interface）和设备（device）两词是交替使用的，不过在其他情境下，这两个词之间的差异性是很重要的。

所有设备的net_device结构放在一个由全局变量dev_base所指的全局列表中。此数据结构定义在*include/linux/netdevice.h*中。网络设备的注册在第八章中描述。在第八章中会详细描述net_device的大多数字段如何做初始化以及初始化的时机。

如同sk_buff，这个结构相当庞大，包括很多特定功能的参数，以及其他来自很多不同分层的参数。因此，此结构的整体组织很有可能会因为优化的原因而很快做一些改变。

网络设备可以分成几种类型，如Ethernet卡和Token Ring卡。虽然对同一种类型的所有设备而言，net_device结构的某些字段会置成相同的值，但是有些字段则必须由每种设备模型做不同的设置。因此，对几乎所有类型而言，Linux都提供了一个通用函数，可以对一些参数做初始化，使其值在所有模型之间都一样。除了针对其模型设置那些具有特定值的字段外，每个设备驱动程序都会启用此函数。驱动程序还可以改写一些已由内核初始化的字段（例如，为了改善性能）。更多细节参见第八章。

net_device结构的字段可以分成以下几种类型：

- 配置（Configuration）。
- 统计数据（Statistics）。
- 设备状态（Device status）。
- 列表管理（List management）。

注4：bonding（绑定）也称为etherChannel（Cisco术语）和trunking（中继，Sun术语），允许一组接口群集起来，视为单一接口。当系统需要以高带宽支持点对点联机时，这种功能就有用处。速度的增加几乎可以达到线性，而虚拟接口的吞吐量几乎等于个别接口的吞吐量之总和。

注5：VLAN指的是Virtual LAN（虚拟LAN）。VLAN的使用是一种方便的方式，通过给Ethernet帧附加VLAN标签，就可以把使用相同L2交换的、在不同广播域中的流量隔离出来。可以在*http://www.linuxjournal.com/article/7268*中找到关于VLAN的简介以及VLAN与Linux结合的文档。

- 流量管理（Traffic management）。
- 功能专用（Feature specific）。
- 通用（Generic）。
- 函数指针（或 VFT）。

标识符

net_device结构有三个标识符，不能搞混：

int ifindex

独一无二的 ID，当设备以 dev_new_index 注册时分派给每个设备。

int iflink

这个字段主要是由（虚拟）隧道设备使用，可用于标识抵达隧道另一端的真实设备。

unsigned short dev_id

目前在 zSeries OSA NIC 上由 IPv6 所用。此字段用于区别可由不同 OS 同时共享的同一种设备的诸多虚拟实例。参见 *net/ipv6/addrconf.c* 中的注释。

配置

有些配置字段由内核根据网络设备的种类给定一个默认值，而其他字段留给驱动程序填写。如前所述，驱动程序可以改变默认值，而有些字段甚至可以在执行期间通过像 ifconfig和ip的命令予以改变。事实上，有些参数 —— base_addr、if_port、dma 以及irq —— 通常是在设备模块加载时由用户设置的。另一方面，虚拟设备不会使用这些参数。

char name[IFNAMSIZ]

设备的名称（例如，*eth0*）。

unsigned long mem_start

unsigned long mem_end

这些字段描述设备所用的共享内存，用于设备与内核沟通。其初始化和访问只会在设备驱动程序内进行，较高分层不需要关心这些字段。

unsigned long base_addr

设备自有内存映射到 I/O 内存的起始地址。

unsigned int irq

设备用于与内核对话的中断编号。此值可由多个设备共享。驱动程序使用request_irq函数分配此变量，并且使用free_irq予以释放。

unsigned char if_port

此接口所使用的端口类型。参见下一节“接口类型和端口”。

unsigned char dma

设备所使用的DMA通道（如果有的话）。为了从内核获取和释放DMA通道，文件*kernel/dma.c*定义了函数request_dma和free_dma。取得DMA通道之后，要予以开启或关闭时，可以使用各种*include/asm-architecture*文件中（例如，*include/asm-i386*）所提供的enable_dma和disable_dma。这些函数由ISA设备使用；PCI设备采用了其他的方法，因而不需要这些函数。

并非所有设备都可用DMA，因为有些总线不使用DMA。

unsigned short flags

unsigned short gflags

unsigned short priv_flags

flags字段中的某些位代表网络设备的功能(如IFF_MULTICAST)，而其他位代表状态的改变(如IFF_UP或IFF_RUNNING)。你可以在*include/linux/if.h*中发现这些标识的完整列表。此设备驱动程序通常会在初始化期间设置这些功能，而这些状态标识由内核管理，以响应外部事件。这些标识的设置可以通过熟悉的ifconfig命令查看：

```
bash# ifconfig lo
lo          Link encap:Local Loopback
            inet addr:127.0.0.1  Mask:255.0.0.0
            UP LOOPBACK RUNNING  MTU:3924  Metric:1
            RX packets:198 errors:0 dropped:0 overruns:0 frame:0
            TX packets:198 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:0
```

在这个例子中，UP LOOPBACK RUNNING就相当于标识IFF_UP、IFF_LOOPBACK以及IFF_RUNNING。

priv_flags用于存储用户空间不可见的标识。目前而言，此字段由VLAN和Bridge虚拟设备使用。gflags几乎已不再使用了，只是由于兼容性的原因而存在。这些标识可通过dev_change_flags函数进行修改。

int features

这是另一组标识位，用于存储其他一些设备功能。为此数据结构包含多个标识变量并非是多余的。features字段可报告适配卡的功能，以便与CPU通信，如适配

卡能否对高端内存做DMA或者硬件能否对所有封包做校验和工作。可能的特征功能的列表定义在net_device结构本身之内。此参数由设备驱动程序初始化。你可以在net_device数据结构定义中找到NETIF_F_*XXX*特征功能列表以及很好的注释。

unsigned mtu

MTU代表最大传输单元(Maximum Transmission Unit),表示设备能处理的帧的最大尺寸。表2-1是最常见的网络技术的MTU取值。

表2-1：不同设备类型的MTU值

设备类型	MTU	
PPP	296	
SLIP	296	
Ethernet	1,500	
ISDN	1,500	
PLIP	1,500	(ether_setup)
Wavelan	1,500	(ether_setup)
EtherChannel	2,024	
FDDI	4,352	
Token Ring 4 MB/s (IEEE 802.5)	4,464	
Token Bus (IEEE 802.4)	8,182	
Token Ring 16 MB/s (IBM)	17,914	
Hyperchannel	65,535	

Ethernet MTU值有必要澄清一下。Ethernet帧规范把最大有效载荷尺寸定为1,500个字节。有时候，你会发现Ethernet MTU定义为1,518或1,514：第一个Ethernet帧的最大尺寸包括报头在内，而第二个包括报头但不含帧检查序列（4个字节的校验和）。

在1998年，Alteon Networks(2000年时由Nortel Networks收购)提议把Ethernet帧的最大载荷增加至9 KB。后来，这项提议正式成为一份IETF因特网草案，但是IEEE没有接受。在IEEE规范中，帧的有效载荷若超过1,500个字节，通常就称为巨大帧（jumbo frame)；与Gigabit Ethernet一起使用以增加吞吐量。这是因为对于大型数据传输，帧越大则所需要的帧数目就越少，中断也就较少，因此CPU的使用也会减少，报头所需的消耗也会变少，等等。有关增加Ethernet MTU的优点以及为什么IEEE不同意让这种扩充规格标准化的讨论，可以阅读《Use of Extended Frame Sizes in Ethernet Networks》白皮书；也可以在*http://www.ietf.org/proceedings/01aug/I-D/draft-ietf-isis-ext-eth-01.txt*中找到，或者上网搜索。

`unsigned short type`

设备所属的类型（Ethernet、Frame Relay 等）。*include/linux/if_arp.h* 包含可能类型的完整列表。

`unsigned short hard_header_len`

以字节为单位的设备头的大小。例如，Ethernet 报头是 14 字节。每个设备头的长度定义在该设备的头文件中。例如，对于 Ethernet 而言，`ETH_HLEN` 就定义在 *<include/linux/if_ether.h> 中*。

`unsigned char broadcast[MAX_ADDR_LEN]`

链路层广播地址。

`unsigned char dev_addr[MAX_ADDR_LEN]`

`unsigned char addr_len`

`dev_addr`是设备链路层地址，不要将其与L3或IP地址混淆。此地址的字节长度由`addr_len`指定。`addr_len`的值依赖于设备的类型。Ethernet地址都是6个字节长。

`int promiscuity`

参见稍后的“混杂模式”一节。

接口类型和端口

有些设备有一个以上的连接器（最常见的组合是 BNC+RJ45），允许用户根据需求选择其中之一使用。这个参数用于设置设备的端口类型。当设备驱动程序没有通过配置命令强制选择特定的端口类型时，就会简单地选择默认的端口类型。有些情况下，一个设备驱动程序可以处理不同的接口模型，在这些情况下，该接口可以按特定次序尝试所有端口类型而找出其端口类型。下面显示的代码片段，是一个设备驱动程序如何根据已有的配置而设定其接口模式：

```
switch (dev->if_port) {
case IF_PORT_10BASE2:
    writeb((readb(addr) & 0xf8) | 1, addr);
    break;
case IF_PORT_10BASET:
    writeb((readb(addr) & 0xf8), addr);
    break;
}
```

混杂模式

某些网络管理任务会要求一个系统接收在一条共享缆线传播的所有帧，而不是仅限于地址直接指定给该系统的帧。一个设备如果可以接收所有封包，意味着其处于混杂模式（promiscuous mode）。例如，应用程序检查其局域网段的性能或安全缺陷时，就需要使

用这种模式。桥接代码也会用到混杂模式（参见第四部分）。最后，不幸的是，对恶意的窥探者而言，这种模式显然有其价值所在。由于这个原因，除非数据被加密，否则局域网络上其他用户的数据都不安全。

net_device结构包含一个名为promiscuity的计数器，表示一个设备处于混杂模式中。采用计数器而非简单标识的原因在于，多个客户程序可能都会要求混杂模式。因此，进入混杂模式时递增该计数器，而退出该模式时递减该计数器。除非计数器为零，否则该设备不会退出混杂模式。通常来讲，对此字段的操作是通过调用dev_set_promiscuity函数实现的。

当promiscuity为非零时(如通过调用dev_set_promiscuity),flags的IFF_PROMISC标识位也会设置，并且由配置此接口的函数进行检查。

下列取自*drivers/net/3c59x.c*驱动程序的代码片段，说明了如何根据flags字段中的标识（位）而设置不同的接收模式：

```
static void set_rx_mode(struct net_device *dev)
{
        int ioaddr = dev->base_addr;
        int new_mode;
        if (dev->flags & IFF_PROMISC) {
             if (corqscreq_debug > 3)
                  printk("%s: Setting promiscuous mode.\n", dev->name);
          new_mode = SetRxFilter | RxStation | RxMulticast | RxBroadcast
            | RxProm;
      } else if ((dev->mc_list)  ||  (dev->flags & IFF_ALLMULTI)) {
             new_mode = SetRxFilter | RxStation | RxMulticast | RxBroadcast;
        } else
             new_mode = SetRxFilter | RxStation | RxBroadcast;

        outw(new_mode, ioaddr + EL3_CMD);
}
```

当IFF_PROMISC标识被置位时，就会对new_mode变量做初始化，从而接收地址指定该网卡的数据流(RxStation)、多播数据流(RxMulticast)、广播数据流(RxBroadcast)以及其他数据流。EL3_CMD是对ioaddr内存地址的偏移量，代表与设备交互时命令应该拷贝到何处。

统计数据

net_device结构没有提供一个用来记录统计数据的收集字段，而是引入了一个由驱动程序设置的priv指针，指向一个存储有关接口信息的私有数据结构。私有数据由统计数据组成，如已收发的封包数目，以及已经发生的错误数目。

priv指向结构的格式取决于设备类型以及特定的模型：因此，不同的Ethernet卡使用不同的私有结构。然而，几乎所有结构都包括一个类型为net_device_stats（定义在*include/linux/netdevice.h*）的字段，该字段包含所有网络设备共有的统计数据，而且可以通过get_stats方法获取，稍后说明。

无线设备的行为迥异于有线设备，致使net_device_stats数据结构不适用于无线设备。因此，无线设备改用一个类型为iw_statistics的字段，可以通过调用get_wireless_stats的方法获取，稍后说明。

priv指向的数据结构有时名称会反映出其接口（例如，vortex_private是指Vortex和Boomerang系列的适配卡，也称为3c59x系列），而其他时候则只是简单地称为net_local。不过，net_local中的字段依然是由每个设备驱动程序各自定义。

私有数据结构复杂度的多少，依赖于适配卡的功能，以及设备驱动程序编写者打算采用多精密的统计数据和复杂的设计以提高性能。例如，比较3c507 Ethernet卡所采用的通用net_local结构（在*drivers/net/3c507.c*中），以及3c59x Ethernet卡所采用的非常详尽的vortex_private结构（在*drivers/net/3c59x.c*中）。然而，这两种结构都包含一个类型为net_device_stats的字段。

在第八章将会看到，私有数据结构有时会添加到net_device结构自身（两者只需一次malloc），但有时也会分配成一个独立的区块。

设备状态

为了控制与NIC之间的交互，每个设备驱动程序都必须维护一些信息，如表示接口需要哪种行为的时间戳和标识。在SMP（Symmetric multiprocessing,对称多道处理）系统中，内核也必须确保不同CPU对同一个设备并发访问的正确处理。net_device结构的以下几个字段就是专门用于这些类型的信息：

unsigned long state

由网络队列子系统所使用的一组标识。其索引值是enum netdev_state_t（*include/linux/netdevice.h*）中的常数，而为每位所定义的常数类似__LINK_STATE_XOFF。个别位的设置和清除都使用通用函数set_bit和clear_bit，而这两个函数的调用通常是通过一个包裹函数把用到的位的细节隐藏起来。例如，要停止一个设备队列时，子系统会调用netif_stop_queue，如下列所示：

```
static inline void netif_stop_queue(struct net_device *dev)
{
    ...
    set_bit(_ _LINK_STATE_XOFF, &dev->state);
}
```

第十一章简要地介绍了流量控制子系统。

`enum {...} reg_state`

设备的注册状态。参见第八章。

`unsigned long trans_start`

最近的一个帧传输启动的时间（以jiffies测量）。设备驱动程序会在传输前设置此值。如果在一段给定时间后传输没有完成，这个字段用于检测适配卡的问题。传输时间过长就意味着有地方出错，在这种情况下，驱动程序通常复位适配卡。

`unsigned long last_rx`

最后一个封包到达的时间（以jiffies测量）。此刻，不用于任何特殊的目的，但有需要时可以利用。

`struct net_device *master`

有些协议允许一组设备群集起来作为单一设备。这些协议包括EQL（Equalizer Load-balancer for serial network interfaces，串行网络接口的均衡负载平衡器）、Bonding(也称为EtherChannel和trunking),以及流量控制的TEQL(普通均衡器)队列规则。群组中的一个设备会被选定为所谓的主设备（master），扮演特殊角色。这个字段是一个指针,指向群组中主设备的net_device数据结构。如果此接口并非这类群组中的成员之一,则此指针就只是NULL。

`spinlock_t xmit_lock`

`int xmit_lock_owner`

xmit_lock锁使驱动程序函数hard_start_xmit的访问串行化（serialize）。这意味着，每个CPU一次只能对任何给定的一个设备做一次传输。xmit_lock_owner是持有该锁的CPU的ID。对单处理器系统而言，其值总是0，而在SMP系统中当该锁没被取走时，其值就为-1。当设备驱动程序支持时，也有可能做无锁的传输。参见第十一章有关有锁和无锁的情况。

`void *atalk_ptr`

`void *ip_ptr`

`void *dn_ptr`

`void *ip6_ptr`

`void *ec_ptr`

`void *ax25_ptr`

这6个字段都是指针，指向特定协议专用的数据结构，而每个数据结构都包含一些该协议私有的参数。例如，ip_ptr指向一个类型为in_device（尽管其声明为void *）的数据结构，其中包含各种不同的与IPv4相关的参数，其中有该接口上

所配置的IP地址列表（参见第十九章）。本书其他章节会描述本书中涉及协议所使用的数据结构字段。多数时候，只有其中的一个字段在使用。

列表管理

net_device数据结构均被插入到一个全局列表和两个hash表中，对此在第八章中予以描述。下列字段用于完成这些任务：

struct net_device *next

把每个net_device数据结构链接为全局列表中的下一个（next）元素。

struct hlist_node name_hlist

struct hlist_node index_hlist

把net_device结构链接至两个hash表的bucket列表。

链路层多播

多播是一种用于把数据传递给多位接收者的机制。多播可以在L3网络层（IP）以及L2链路层（Ethernet）中使用。在本节我们关心的是后者（链路层）。

链路层多播的传送，可以通过在链路层报头中使用特殊地址或者控制信息（当链路层协议不支持时，也可以模拟）。Ethernet本身就支持多播：在第十三章会看到一个Ethernet地址如何划分为单播、多播或广播。

利用一个特定位把多播地址和其他范围的地址区分开来。这意味着可能地址中有50%是多播的，而2^{48}的50%可是很大的数字！当一个接口被要求加入一大堆多播群组时（每个群组都由一个多播地址标识），则该接口只简单地监听所有多播地址，这比维护一份很长的列表，再根据该列表费时地过滤输入的L2多播帧要更有效率和快速。net_device数据结构中的flags之一就是用于表示该设备是否监听所有地址。决定何时设置或清除此标识，由本节说明的allmulti字段控制。

每个设备都会为其监听的每个链路层多播地址保存一个dev_mc_list结构的实例。链路层多播地址可以分别用dev_mc_add和dev_mc_delete函数添加或删除。net_device结构中相关的字段包括：

struct dev_mc_list *mc_list

指向此设备的dev_mc_list结构列表表头的指针。

int mc_count

此设备的多播地址数目，也就是mc_list所指的列表长度。

int allmulti

为非零值时，引起此设备监听所有多播地址。如同本章稍早所讨论的promiscuity，allmulti是引用计数，而非一个简单的布尔值。这是因为多台设备（例如，VLAN和绑定设备）可能各自都需要监听所有地址。当此变量从0变为非零时，就会调用dev_set_allmulti函数，以指示该接口监听所有多播地址。当allmulti变为0时，就会发生相反的事。

流量管理

Linux的流量控制子系统已有长足的进步，并且代表着Linux内核实力的一方面。相关联的内核选项为“Device drivers → Networking support → Networking options → QoS and/or fair queueing”。net_device结构中的相关字段包括：

struct net_device *next_sched

由软中断之一使用，对此在第十一章给出了描述。

struct Qdisc *qdisc

struct Qdisc *qdisc_sleeping

struct Qdisc *qdisc_ingress

struct list_head qdisc_list

这些字段用于管理入口和出口的封包队列，以及不同CPU对此设备的访问。

spinlock_t queue_lock

spinlock_t ingress_lock

流量控制子系统为每个网络设备定义了一个私有的出口队列。queue_lock用于避免对出口队列的并发访问（参见第十一章），ingress_lock则是针对入口流量做相同的事。

unsigned long tx_queue_len

设备的传送队列的长度。当内核支持流量控制时，可能不用tx_queue_len（仅有很少的队列规则使用）。表2-2所列是最常见设备类型所采用的值。其值可以通过*sysfs*文件系统予以调整（参见*/sys/class/net/device_name/*目录）。

表2-2：不同设备类型的tx_queue_len值

设备类型	tx_queue_len
Ethernet	1,000
Token Ring	100
EtherChannel	100
Fibre Channel	100

表2-2：不同设备类型的tx_queue_len值（续）

设备类型	tx_queue_len
FDDI	100
TEQL（true link equalizer，普通链接均衡器）[a]	100
ISDN	30
HIPPI	25
PLIP	10
SLIP	10
AX25	10
EQL（串行网络接口的均衡负载平衡器）	5
Generic PPP	3
Bonding	0
Loopback	0
Bridge	0
VLAN	0

a. TEQL是队列规则之一，可以用流量控制（QoS层）进行配置。

根据所用的队列规则决定输入和输出的封包队列化的策略，可能会用到tx_queue_len，但也可能用不到。当队列类型为FIFO（First In，First Out，先进先出）或者相当简单的队列时，通常就会使用到它。

注意，所有队列长度为0的设备都是虚拟设备：虚拟设备依赖相关联的真实设备去做所有的队列化工作（回环设备除外，因为回环设备由内核内部使用，会立刻传递所有流量，因此不需要）。

功能专用

如说明sk_buff时所见，只有当所属的功能包含在内核时，那些参数才会出现在net_device的定义中（注6）：

```
struct divert_blk *divert
```

分流器（diverter）是一种功能，允许你改变输入的封包的源和目的地址。因此，有可能以此配置所指定的特定特征重新绕送流量至不同的接口或不同的主机。此字段所指的数据结构存储了分流器功能所需的参数。相关联的内核选项为“Device drivers → Networking support → Networking options → Frame Diverter”。

注6：只有当相关联的功能为内核之一部分时，这些字段才会实际引入。范例参见br_port。

```
struct net_bridge_port *br_port
```

当此设备配置成桥接端口时，就需要额外的信息。桥接代码和STP（Spanning Tree Protocol，生成树协议）会在第四部分讨论。相关联的内核选项为“Device drivers →Networking support→Networking options→802.1d Ethernet Bridging”。

```
void (*vlan_rx_register)(...)
void (*vlan_rx_add_vid)(...)
void (*vlan_rx_kill_vid)(...)
```

这三个函数指针由VLAN代码所用，分别是指把设备注册为拥有VLAN标记（VLAN tagging）功能（参见*net/8021q/vlan.c*）、把VLAN添加至设备以及从设备中删除VLAN。相关联的内核选项为“Device drivers → Networking support → Networking options → 802.1Q VLAN Support”。

```
int netpoll_rx
void (*poll_controller)(...)
```

由选用的Netpoll功能使用，在第十章会简略论及。

通用

除了稍早讨论的net_device结构的列表管理字段外，还有一些字段用于管理一些结构，确保这些结构在不需要时能予以删除。

```
atomic_t refcnt
```

引用计数。此计数器变为零之前，设备无法除名（unregister），参见第八章。

```
int watchdog_timeo
struct timer_list watchdog_timer
```

配合先前讨论过的tx_timeout变量，这些字段实现了第十一章“看门狗定时器”一节所讨论的定时器。

```
int (*poll)(...)
struct list_head poll_list
int quota
int weight
```

由NAPI功能使用，对此在第十章予以描述。

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

其他无线设备使用的参数和函数指针。还可参见get_wireless_stats。

`struct list_head todo_list`

网络设备的注册和除名是以两步骤进行的。`todo_list`用于处理第二步骤。参见第八章。

`struct class_device class_dev`

由新的通用内核驱动程序基础架构使用。

函数指针

我们在第一章中看到，网络代码大量使用了函数指针。`net_device`数据结构中也有不少的函数指针。这类函数主要用于：

- 传输和接收帧。
- 在缓冲区上添加或解析链路层报头。
- 改变配置的一部分。
- 获取统计数据。
- 与特定功能交互。

前几节说明完成特定任务所需的字段时，已介绍了一些函数指针。以下是通用的函数指针：

`struct ethtool_ops *ethtool_ops`

指向一组函数指针的指针，用于设置或取出不同设备参数的配置。参见第八章“Ethtool”一节。

```
int (*init)(...)
void (*uninit)(...)
void (*destructor)(...)
int (*open)(...)
int (*stop)(...)
```

用于初始化、清理、销毁、开启以及关闭一个设备。不是所有函数都会被用到。参见第八章。

```
struct net_device_stats* (*get_stats)(...)
struct iw_statistics* (*get_wireless_stats)(...)
```

设备驱动程序所收集的一些统计数据可以使用用户空间应用程序予以显示，如`ifconfig`和`ip`，而其他统计数据则严格由内核使用，本章稍早的“设备状态”一节

对此已有讨论。这两种方法用于收集统计数据，get_stats针对一般设备，而get_wireless_stats针对无线设备。还可以参见稍早的“统计数据”一节。

```
int (*hard_start_xmit)(...)
```

用于传输一个帧，参见第十一章。

```
int (*hard_header)(...)
int (*rebuild_header)(...)
int (*hard_header_cache)(...)
void (*header_cache_update)(...)
int (*hard_header_parse)(...)
int (*neigh_setup)(...)
```

由邻居层使用。参见第二十七章的“设备驱动程序提供的方法”以及“邻居初始化”两节。

```
int (*do_ioctl)(...)
```

ioctl是系统调用，用于向设备发出命令（参见第三章）。调用此方法可以处理一些ioctl命令（参见第八章）。

```
void (*set_multicast_list)(...)
```

在“链路层多播”一节已知道mc_list和mc_count用于管理L2多播地址列表。这个方法用于要求设备驱动程序配置设备以监听这些地址。通常不会直接调用此方法，而是通过包裹函数，如dev_mc_upload或其无锁版本__dev_mc_upload。当一个设备无法安装多播地址列表时，就只会将其全部开启。

```
int (*set_mac_address)(...)
```

改变设备的MAC地址。当设备没有提供此功能时（如同Bridge虚拟设备的情况），就会置为NULL。

```
int (*set_config)(...)
```

配置驱动程序参数，如硬件参数irq、io_addr以及if_port。较高分层的参数（如协议地址）是由do_ioctl来处理。使用此方法的设备不是很多，特别是能实现探测函数的新设备。可以在*drivers/net/sis900.c*中的sis900_set_config找到一个好的例子及其说明。

```
int (*change_mtu)(...)
```

改变设备的MTU值（参见本章稍早“配置”一节对MTU的说明）。改变此字段对设备驱动程序没有影响，只是强制内核软件接受新的MTU，再据此处理分段。

```
void (*tx_timeout)(...)
```

在看门狗定时器到期之时调用此方法，用于确认该次传送是否花了一段很可疑的长

时间才完成。只有定义了该方法，看门狗定时器才会启动。参见第十一章“看门狗定时器”一节的更多信息。

`int (*accept_fastpath)(...)`

快速交换是一种内核功能，允许设备驱动程序在中断期间使用一个小的缓存用来路由输入的流量（跳过所有软件分层），也称为FASTROUTE。从内核2.6.8版本开始不再支持快速交换功能。这种方法用于测试此设备是否可以使用快速交换功能。

本章涉及的文件

图2-11展示了本章引用的主要文件。缺少的几个文件将在接下来的几章介绍。

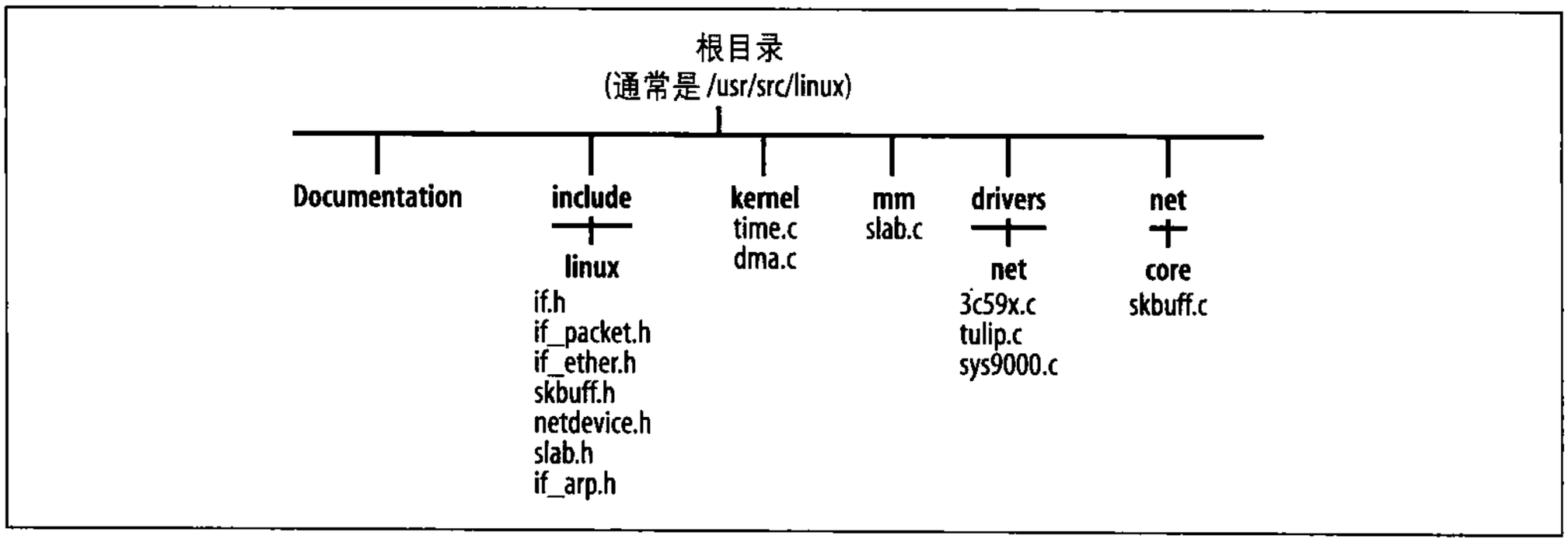
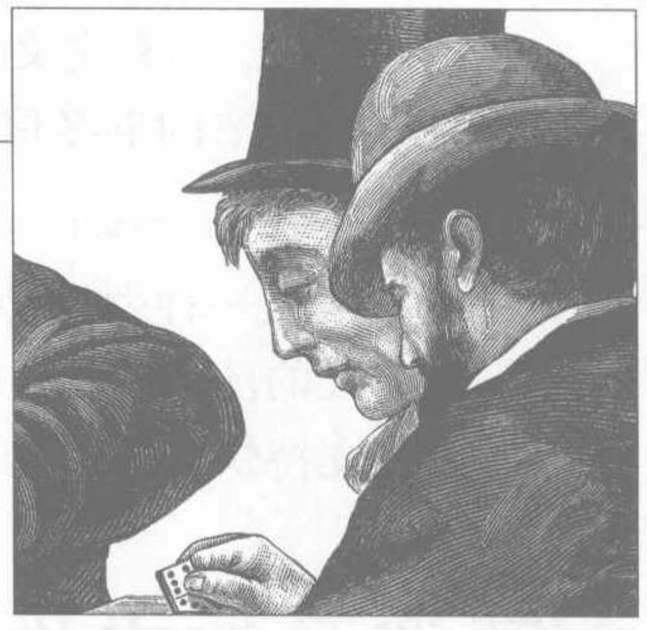


图2-11：本章引用的文件

第三章

用户空间与内核的接口



本章简要地介绍用户空间应用程序可用于与内核通信或读取内核输出信息的主要机制。我们不谈其实现细节，因为每种机制都需要一章的篇幅才能说清楚。本章的目的在于给出足够的指引，若有兴趣进一步研究该项主题，可以找程序代码和外部的文档。例如，通过本章，你可以取得所需要的信息，了解给定目录如何添加到*/proc*及其所在位置、处理给定`ioctl`命令的内核处理函数（kernel handler），以及目前用户空间网络配置最佳接口 Netlink 提供了哪些函数。

本章所关注的机制，就是在谈到用户空间配置命令（如*ifconfig*和*route*）及内核处理函数（应用所请求的配置）两者的接口时，我在本书中经常提及的那些机制。有关内核内部的通信，以及内核与用户空间之间的通信，所采用的通用消息传输系统的相关分析，可以参考《深入理解 Linux 内核》一书（O'Reilly 出版）。

本书讨论每项功能时，都会有一些章节说明用户空间配置工具与内核的通信。本章的信息就是为了帮助你对那些章节理解地更加透彻。

概论

内核通过各种不同的接口把内部信息输出到用户空间。除了程序员用于请求特定信息的经典系统调用（system call）外，还有三个特殊接口，而其中两个是虚拟文件系统：

procfs（/proc 文件系统）

这是一个虚拟文件系统，通常是挂（mount）在*/proc*，允许内核以文件的形式向用户空间输出内部信息。这些文件并没有实际存在于磁盘中，但是可以通过*cat*或*more*以及 > shell 重定向字符予以写入。这些文件甚至可以像真实文件一样指定其访问权限。因此，创建这些文件的内核组件可以说明任何一个文件可由谁读取或写

入。目录则不能被写入（也就是说，用户不能把文件或目录添加到*/proc*中的任何目录，也不能从中予以移除）。

多数Linux发行套件的默认内核都支持*procfs*。*procfs*不能编译成为一个模块。配置菜单中相关的内核选项为“Filesystems→pseudo filesystems→/proc file system support”。

sysctl（/proc/sys 目录）

此接口允许用户空间读取或修改内核变量的值。不能用此接口对每个内核变量进行操作：内核应明确指出哪些变量从此接口是可见的。从用户空间，你可以用两种方式访问*sysctl*输出的变量。一种是`sysctl`系统调用（参见*man sysctl*），而另一种是*procfs*。当内核支持*procfs*时，会在*/proc*中添加一个特殊目录（*/proc/sys*），为每个由*sysctl*所输出的内核变量引入一个文件。

*procps*包随附的*sysctl*命令可用于配置由*sysctl*接口所输出的变量。此命令通过写入*/proc/sys*与内核对话。多数Linux发行套件默认内核都支持*sysctl*。*sysctl*不能编译成为一个模块。配置菜单中相关的内核选项为“General setup → Sysctl support”。

sysfs（/sys 文件系统）

*procfs*和*sysctl*已经被滥用了多年，这就导致引入了一种新的文件系统：*sysfs*。*sysfs*以非常干净而有组织的方式输出很多信息。如你所期望的，当前由*sysctl*所输出的部分信息可以移植到*sysfs*。

内核对*sysfs*的支持只是在2.6版才开始。多数Linux发行套件默认内核都支持*sysfs*。*sysfs*不能编译成为一个模块。配置菜单中相关的内核选项为“Filesystems→Pseudo filesystems → sysfs filesystem support（NEW）”。只有先开启“General setup → Configure standard kernel features（for small systems）”选项，才能看得到上述选项。

在O'Reilly最新版的《Linux 设备驱动程序》书中可以找到对*sysfs*的详尽分析。在第十七章中，我们将看到桥接代码如何使用*sysfs*。

你也可以使用下列接口把命令传给内核，配置某事或者倾卸（dump）某事的配置内容：

`ioctl` *系统调用*

`ioctl`（输入/输出控制）系统调用操作的对象是一个文件，通常是用于实现特殊设备所需但标准文件系统没有提供的操作。你也可以把socket系统调用返回的套接字描述符（socket descriptor）传给`ioctl`，而这也是网络代码使用`ioctl`的方式。此接口也由老一代（old-generation）命令所用，如*ifconfig*和*route*等。

Netlink 套接字（socket）

这是网络应用程序与内核通信时最新的首选机制。IPROUTE2包中大多数命令都使用此接口。对Linux而言，Netlink代表的就是BSD世界中的路由套接字(routing socket)。

多数网络内核功能都可以用Netlink或`ioctl`接口进行配置，因为内核支持较新的配置工具（IPROUTE2）以及老式工具（*ifconfig*、*route* 等）。

procfs 与 sysctl

procfs 和 *sysctl* 都输出内核内部信息，但是 *prcfs* 主要是输出只读数据，而大多数 *sysctl* 信息都是可写入的，但只有超级用户（superuser）能写入。

就输出只读数据而言，*procfs* 和 *sysctl* 之间的选择依赖于有多少信息应该输出。与一个简单的内核变量或数据结构相关联的一些文件，可以用 *sysctl* 输出。其他涉及更为复杂的数据结构而且需要特殊格式时，就以 *procfs* 输出，例如缓存和统计数据。

procfs

大多数网络功能在其初始化时都会在 */proc* 中注册一个或多个文件，不是在引导时就是在模块加载时。当一位用户读取该文件时，会引起内核间接运行一组内核函数，以返回某种输出内容。网络代码所注册的文件位于 */proc/net*。

/proc 中的目录可以使用`proc_mkdir`创建。*/proc/net* 中的文件可以使用定义在 *include/linux/proc_fs.h* 中的`proc_net_fops_create`和`proc_net_remove`予以注册和除名。这两个函数都是包裹函数，其中含有通用的API：`create_proc_entry`和`remove_proc_entry`。特别地，`proc_net_fops_create`负责创建文档（用`proc_net_create`），然后初始化其文件操作处理函数。我们来看一个实例。

以下是ARP协议如何在 */proc/net* 中注册其 *arp* 文件：

```
static struct file_operations arp_seq_fops = {
    .owner      = THIS_MODULE,
    .open       = arp_seq_open,
    .read       = seq_read,
    .llseek     = seq_lseek,
    .release    = seq_release_private,
};

static int __init arp_proc_init(void)
{
  if (!proc_net_fops_create("arp", S_IRUGO, &arp_seq_fops))
```

```
        return -ENOMEM;
    return 0;
}
```

由proc_net_fops_create的三个输入参数可知，文件名为*arp*，权限必须指定为只读，而且该组文件操作处理例程是arp_seq_ops。当一个用户读取该文件时，使用file_operations数据结构，允许*procfs*返回相当多（chunk）的数据给用户。当数据由一群相同类型的对象组成时会很有用。例如，ARP缓存在返回时一次只返回一项（entry），而路由表在返回时一次只返回一条路径。

open所初始化的例程（前例中的arp_seq_open）会做另一次重要的初始化：注册一个函数指针数组，包括procfs用于遍历要传回给用户数据的所有例程：一个例程启动倾卸（dump），另一个推进到一个项目，而另一个再倾卸一个项目。这些例程内部负责保存必要的环境信息（此例中，就是ARP缓存已经倾卸了多少），这些信息会就是倾卸点以及从正确位置重新继续倾卸所必需的。

```
static struct seq_operations arp_seq_ops = {
    .start   = clip_seq_start,
    .next    = neigh_seq_next,
    .stop    = neigh_seq_stop,
    .show    = clip_seq_show,
};
static int arp_seq_open(struct inode *inode, struct file *file)
{
    ...
    rc = seq_open(file, &arp_seq_ops);
    ...
}
```

sysctl：目录 /proc/sys

用户在*/proc/sys*下看到的一个文件，实际上是一个内核变量。就每个变量而言，内核可以定义：

- 要将其放在*/proc/sys*的何处。与相同内核组件或功能相联系的变量，通常都位于同一个目录中。例如，在*/proc/sys/net/ipv4*里可以找到与IPv4相关的文件。
- 命名。多数时候，文件名只是简单地命名为与相关联的内核变量相同的名字，但是，有时候为了更友好一点，其名字也会改变。
- 访问权限。例如，一个文件可以由任何人读，但只能由超级用户修改。

输出到*/proc/sys*中的变量内容可借助于相关联的文件进行读写（只要你有必要的权限），或者直接用*sysctl*系统调用。

有些目录和文件在引导期间静态定义，而其他的则是在运行期间添加的。导致运行期间创建目录或文件的事件实例如下：

- 当一个内核模块实现一项新功能，或者有一个协议被加载或卸载（unload）时。
- 当一个新的网络设备被注册或除名时。有些配置参数（如*/proc/sys*中的文件）对每个设备而言都有一个实体。例如，*/proc/sys/net/ipv4/conf*（在第三十六章讨论）和*/proc/sys/net/ipv4/neigh*（在第二十九章讨论）目录对每个已注册的网络设备而言，其下都有一个子目录。

*/proc/sys*中的文件和目录都是以ctl_table结构定义的。ctl_table结构的注册和除名是通过在*kernel/sysctl.c*中定义的register_sysctl_table和unregister_sysctl_table函数完成。

以下是ctl_table的关键字段：

const char *procname
: 在*/proc/sys*中所用的文件名。

int maxlen
: 输出的内核变量的尺寸大小。

mode_t mode
: 分派给*/proc/sys*中相关联的文件或目录的访问权限。

ctl_table *child
: 用于建立目录与文件之间的父子（parent-child）关系。本节稍后会看到一些相关实例。

proc_handler
: 当你在*/proc/sys*中读取或写入一个文件时，完成读取或写入操作的函数。所有与文件（就是树的叶子）相关联的ctl_instances都必须由proc_handler初始化。内核会给目录分派一个默认值。

strategy
: 此函数可以选择初始化为 —— 在显示或存储之前 —— 完成数据的额外格式化工作的函数。当*/proc/sys*中的文件用sysctl系统调用访问时，此函数就会被调用。

extra1

extra2
: 两个可选的参数，通常用于定义变量的最小值和最大值。通常把这两个参数称为min/max参数。

根据与文件相关联的变量种类而定，proc_handler和strategy的初始化也会不同。例如，当内核变量由一个或多个整数值组成时，proc_dointvec就是proc_handler所用的函数。表3-1和表3-2列出了可用于初始化proc_handler和strategy的一些函数。所有函数的定义和充分的注释都在*kernel/sysctl.c*中。

表3-1：初始化proc_handler的函数

函数	描述
proc_dostring	读/写一个字符串
proc_dointvec	读/写一个包含一个或多个整数的数组
proc_dointvec_minmax	类似proc_dointvec，但是，要确定输入数据在min/max范围内。不符合该范围的值会被拒绝
proc_dointvec_jiffies	读/写一个整数数组。此内核变量以jiffies为单位表示，但是，返回给用户之前会先转成秒数（反之亦然）
proc_dointvec_ms_jiffies	读/写一个整数数组。此内核变量以jiffies为单位表示，但是，返回给用户前会先转成毫秒数（反之亦然）
proc_doulongvec_minmax	类似proc_dointvec_minmax，但其值为长整数而非整数
proc_doulongvec_ms_jiffies_minmax	读/写一个长整数数组。此内核变量以jiffies为单位表示，但是，返回给用户前，会先转成毫秒数（反之亦然）。此内核变量被指派的值必须在min/max范围内

表3-2：初始化strategy的函数

函数	描述
sysctl_string	读/写一个字符串
sysctl_intvec	读/写一个整数数组，而且确定其值符合min/max范围
sysctl_jiffies	读/写一个以jiffies表示的值，然后将其转成秒数
sysctl_ms_jiffies	读/写一个以jiffies表示的值，然后将其转成毫秒数

strategy或proc_handler函数初始化为包含表3-1或3-2的函数之一的包裹函数，这并非不常见。包裹函数根据相关内核变量的意义，附加一些逻辑或合理性检查。在下一节有一个实例。

在本书中，因任何功能配置而着眼于*procfs*接口时，为了简洁起见，将总是涉及proc_handler函数。

ctl_table 初始化的实例

我们首先看文件和目录所用的ctl_table结构的初始化是什么，然后再看其如何使用。

*/proc/sys/net/ipv4/conf/default/forwarding*文件所用的ctl_table实体的初始化定义在*net/ipv4/devinet.c* 中，如下所示，其用法在第三十六章予以描述。

```
{
    .ctl_name       = NET_IPV4_CONF_FORWARDING,
    .procname       = "forwarding",
    .data           = &ipv4_devconf.forwarding,
    .maxlen         = sizeof(int),
    .mode           = 0644,
    .proc_handler   = &devinet_sysctl_forward,
}
```

从上述的简单介绍看不出这个文件被放在*/proc/sys* 中的何处。等一下你会看到如何找到这些信息。从代码可知，该文件名称为*forwarding*，通过*forwarding* 文件输出其值的内核变量是ipv4_devconf.forwarding(此字段内有一个相当复杂的结构)，参数声明为整数，该文件的访问权是0644 (也就是任何人都可读取，但只有超级用户可以写入)，而proc_handler 函数被初始化为 devinet_sysctl_forward。

现在，我们看 *kernel/sysctl.c* 中一个目录的声明实例：

```
{
    .ctl_name     = CTL_NET,
    .procname     = "net",
    .mode         = 0555,
    .child        = net_table,
}
```

这个ctl_table实体定义了*/proc/sys/net*目录。此时，不需要proc_handler(内核提供默认函数，适用于所有目录的需要)，但却有一个child字段。child是一个指针，指向另一个ctl_table实体，这个实体只是ctl_table实体列表的头元素(对于每一个文件或子目录都会在 *net* 目录中建立一个实体)。

在 /proc/sys 中注册文件

我们知道可以分别用register_sysctl_table和unregister_sysctl_table在*/proc/sys* 中注册或除名文件。注册函数（源码中的文档很详细）需要两个输入参数：

- 指向一个 ctl_table 实体的指针
- 一个标识，指出新元素应该放在位于相同目录中ctl_table实体列表的何处：头（1）或尾（0）。

注意，register_sysctl_table的输入并不包括输入参数ctl_table应该添加到*/proc/sys*文件系统中何处的参考值。原因在于所有的插入都是针对*/proc/sys*目录进行，如果你想要把一个文件注册到*/proc/sys*的子目录，就必需建立一棵树（意味着多个由child字段链接的ctl_table实体）以提供完整路径，然后把代表你刚建的树根的ctl_table实体传给register_sysctl_table。当该树的任何节点尚未存在时，就会被创建。

我们看两个实例，先从简单的着手。下面这段代码源自*drivers/scsi/scsi_sysctl.c*，显示出文件*logging_level*的定义以及如何置放到*/proc/sys/dev/scsi/*目录：

```
static ctl_table scsi_table[] = {
    { .ctl_name      = DEV_SCSI_LOGGING_LEVEL,
      .procname      = "logging_level",
      .data          = &scsi_logging_level,
      .maxlen        = sizeof(scsi_logging_level),
      .mode          = 0644,
      .proc_handler = &proc_dointvec },
    { }
};

static ctl_table scsi_dir_table[] = {
    { .ctl_name     = DEV_SCSI,
      .procname     = "scsi",
      .mode         = 0555,
      .child        = scsi_table },
    { }
};

static ctl_table scsi_root_table[] = {
    { .ctl_name     = CTL_DEV,
      .procname     = "dev",
      .mode         = 0555,
      .child        = scsi_dir_table },
    { }
};

int __init scsi_init_sysctl(void)
{
    scsi_table_header = register_sysctl_table(scsi_root_table, 1) ;
}
```

注意，register_sysctl_table接收的是scsi_root_table，也就是代码中所定义的ctl_table树的根。其结果如图3-1所示。

还注意到，如果稍后你想把另一个文件添加到同一个目录，例如*abc*文件，你需要定义一棵类似的树（也就是*dev*和*scsi*目录是两个相同的ctl_table实体，为新文件*abc*另加一个新的ctl_table实体）。

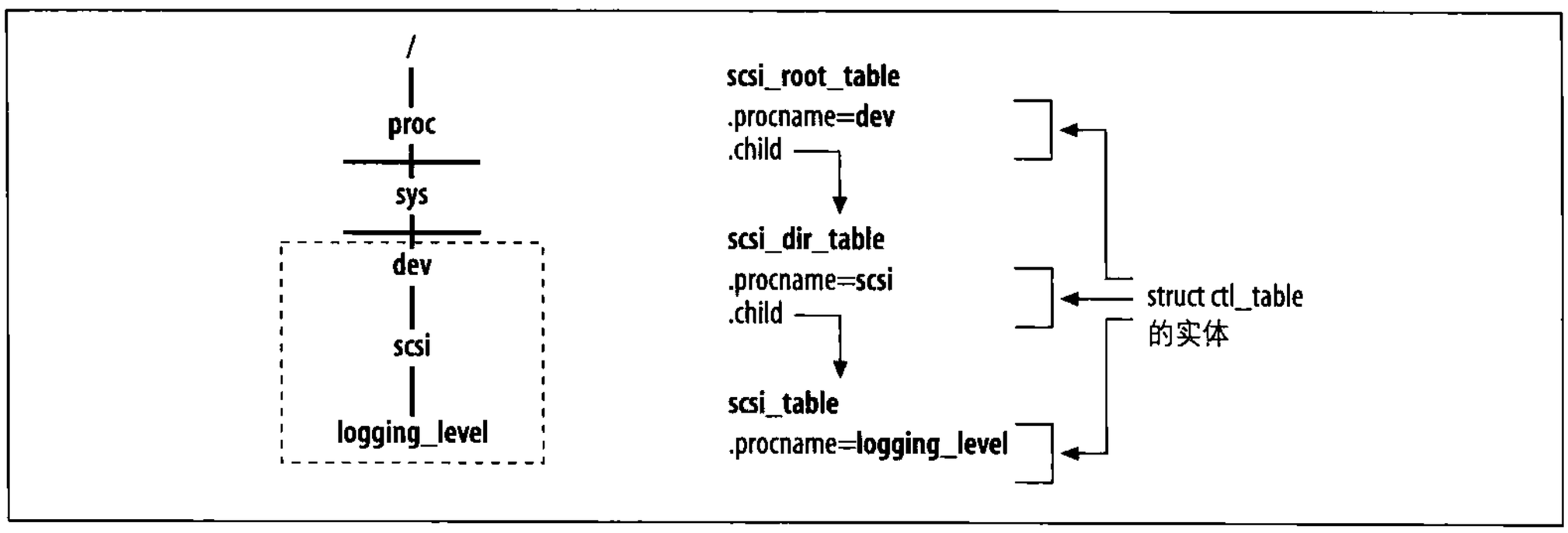


图 3-1：/proc/sys/dev/scsi/logging_level 文件的注册

有时候，开发人员为了简化把新文件添加到已存在的目录而定义一个模板，然后，每次有新文件要添加到相同目录时就予以重用。使用模板的好处是ctl_table实体只需初始化一次便可贯穿整个目录（例如，前例的scsi_root_table和scsi_dir_table）。之后，每次增加一个新文件时，只需对叶节点初始化（也就是真实的文件）。例如，你可以去看一看邻居子系统如何使用在*net/core/neighbour.c*中的neigh_sysctl_register定义neigh_sysctl_template（另参见第二十九章）。

核心网络文件和目录

图 3-2 显示了在*/proc/sys*中由网络代码所使用的主要目录。就每个目录而言，都列出了在哪一章描述其文件。

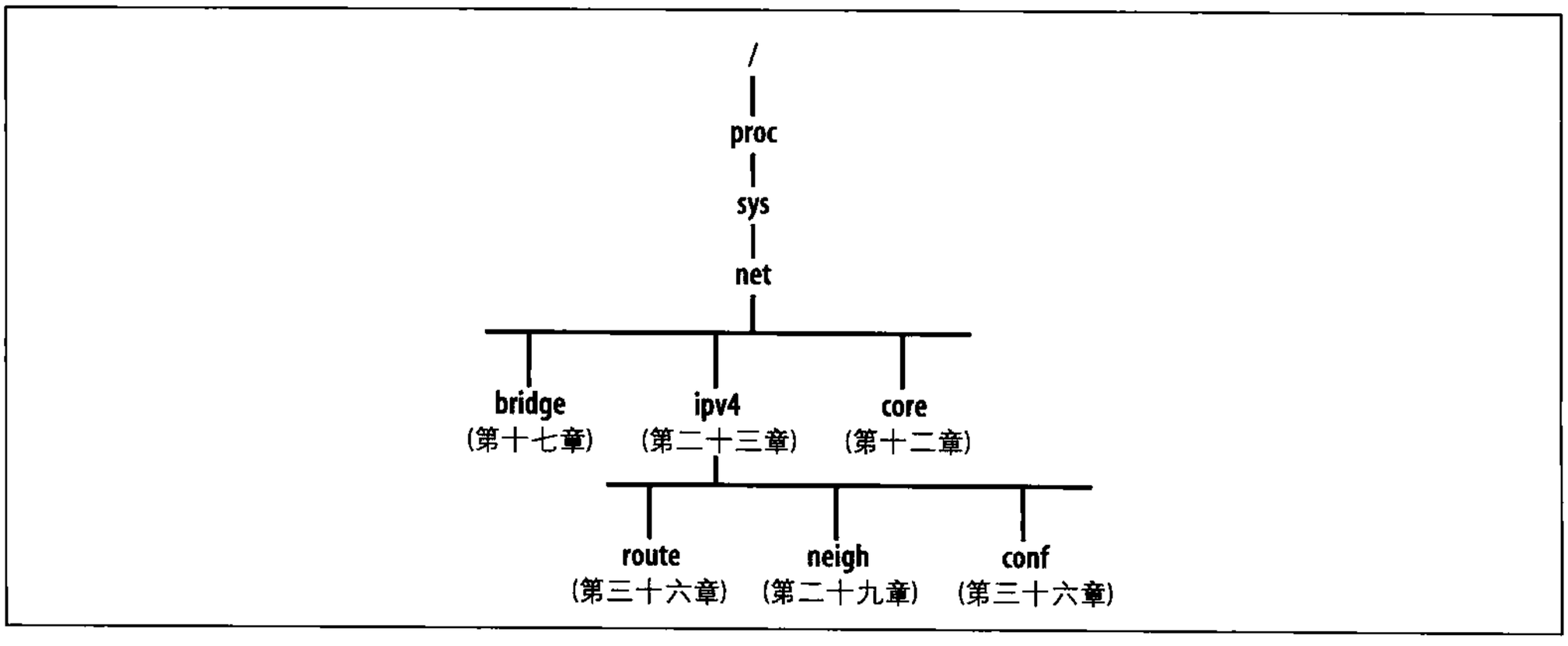


图 3-2：/proc/sys/net 中的核心目录

根据前节所述，我们来看*net*中的树根是如何定义的，以及在引导期间如何注册。

对于如图3-2所示的每个目录以及这些目录的每个文件而言，都有一个ctl_table实体。图3-3显示了图3-2所示的大多数目录的ctl_table实体在何处定义以及其父子关系是什么。为了使图具有更高的可读性，并没有包括所有目录。

图3-3中的三个方块显示了ctl_table初始化的三个实例。注意：

- *netdev_max_backlog*文件被分派了一个proc_handler例程，但没有一个strategy例程。因为netdev_max_backlog是一个整数，来自于用户的输入由proc_dointvec读取。
- *min_delay*文件被分派了proc_handler和strategy两个例程。因为内核变量ip_rt_min_delay以jiffies表示，但是用户的输入和输出都是以秒来表示，因此这两个例程可以完成把秒转换为jiffies。
- *ip_local_port_range*文件是一个有趣的案例。这个文件允许用户配置一个范围，定义两个值。这个范围必须遵循一个最小值和一个最大值的规范。因此，所选的strategy和proc_handler例程必须能够管理一个整数值的数组（此例中为两个整数值）。这些值（extra1和extra2）表示这个范围，并且用于确保用户的输入值遵循此范围。

ioctl

在图3-4顶端，可以看到ioctl调用是如何发布（issue）的。来看一个包括*ifconfig*的示例。

稍早我们曾说过，*ifconfig*命令使用ioctl与内核通信。例如，当系统管理输入像*ifconfig eth0 mtu 1250*这样的命令，用以改变接口*eth0*的MTU时，*ifconfig*会打开一个套接字，用从系统管理员那里接收的信息（例中的data）初始化一个本地数据结构，然后以ioctl调用传送给内核。SIOCSIFMTU是命令标识符。

```
struct ifreq data;
fd = socket(PF_INET, SOCK_DGRAM, 0);
< ... 对"data"初始化 ...>
err = ioctl(fd, SIOCSIFMTU, &data);
```

内核会在几个不同地方处理ioctl命令。图3-4显示了网络代码最常用的ioctl命令如何由sock_ioctl分派，并且路由至正确的函数处理例程。我们将不说明sock_ioctl是如何调用的，或者像UDP和TCP这些传输协议是如何注册其处理例程的。如果你想深入挖掘这部分的代码，可以使用此图作为起点。有关本书所谈及的例程，图3-4提供了正确章节的指引。

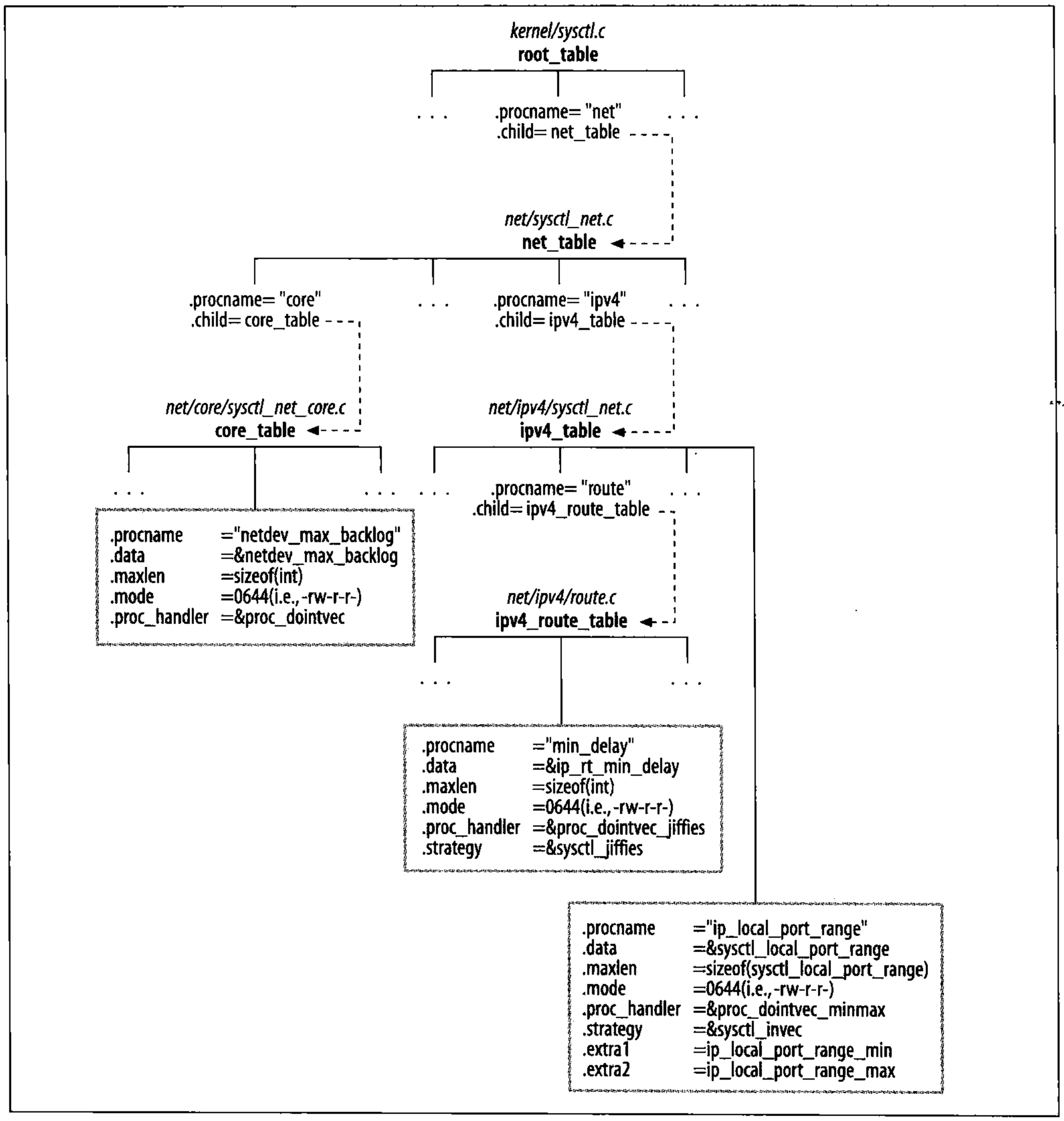


图 3-3：在 /proc/sys/net 中创建的核心目录

为了方便起见，图中 ioctl 命令的名称已做了分解（分成几个组件）。例如，用于把一条路径新增至路由表的命令 SIOCADDRT 被显示成 SIOC ADD RT，以强调两个感兴趣的组件：ADD，表示要添加什么；RT，表示要添加的是一条路由。多数命令都遵循此语法。通常，当一个对象类型可以读写时，命令名称中会多一个组件：G 是指取得，而 S 是指设置。SIOCGIFADDR 和 SIOCSIFADDR 这两个命令可以为接口新增或删除 IP 地

址，就是实际的例子。先前 *ifconfig* 范例中所见的 SIOCSIFMTU 是设定（S）接口（IF）的最大传输单位（MTU），这件事是由 dev_ioctl 所做，但并没有出现在图 3-4 中。

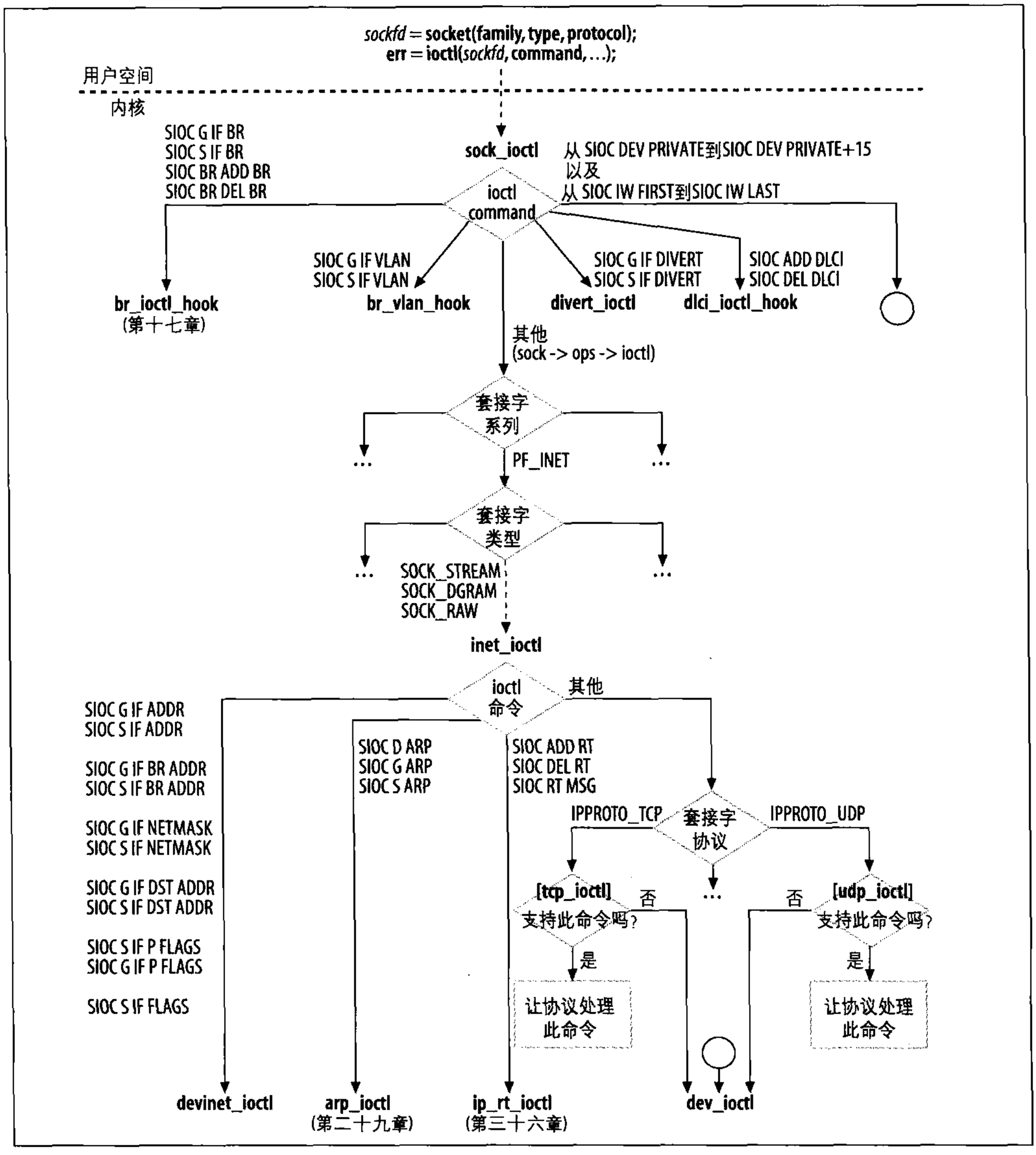


图 3-4：ioctl 命令的分派

网络用的 ioctl 命令列在 *include/linux/sockios.h* 中。设备驱动程序可以用代码定义新的（私有）命令，其范围界于 SIOCDEVPRIVATE 和 SIOCDEVPRIVATE+15 之间。例如，可

以看一看（虚拟）隧道设备使用的四个私有命令在*include/linux/if_tunnel.h*中是如何定义的。然而，使用私有的ioctl命令是被反对和不鼓励的。

各种协议也可以在SIOCPROTOPRIVATE到SIOCPROTOPRIVATE+15范围内定义私有命令。

Netlink

Netlink套接字在RFC 3549中描述得很清楚，代表用户空间与内核的IP网络配置之间的首选接口。Netlink也可作为内核内部以及多个用户空间进程之间的消息传输系统。

通过Netlink套接字，你可以使用标准套接字API打开或关闭套接字、使用套接字传输数据或者接收套接字数据。快速审视一下socket系统调用的原型：

```
int socket(int domain, int type, int protocol)
```

有关TCP/IP套接字如何对此三个自变量做初始化（也就是PF_INET域）的细节，可以使用*man socket*命令。

如同其他任何套接字，当你打开一个Netlink套接字时，必须提供domain、type以及protocol参数。Netlink使用新的PF_NETLINK协议族（域），只支持SOCK_DGRAM类型，而且定义了几种协议，每一种都用于网络协议栈的不同组件（或一组组件）。例如，NETLINK_ROUTE协议用于大多数网络功能，如路由和邻居协议，而NETLINK_FIREWALL用于防火墙（Netfilter）。Netlink的协议列在*include/linux/netlink.h*中的NETLINK_*XXX*枚举列表中。

使用Netlink套接字时，终端点通常是由打开此套接字的进程的ID（PID）标识，而特殊值0代表的就是内核。Netlink的功能之一是传送单播和多播消息：目的地终端点地址可以是一个PID、一个多播群组ID或两者的组合。内核定义Netlink多播群组的目的是传出特定种类事件的通知信息，而用户程序如果对这类通知信息感兴趣，可以向这些群组注册。这些群组列在*include/linux/rtnetlink.h*中的RTMGRP_*XXX*枚举列表中。而其中的两个RTMGRP_IPV4_ROUTE 和RTMGRP_NEIGH群组，分别用于通知有关路由表以及L3到L2的地址映射的改变。我们将在第六部分和第七部分中看到这两个群组的用法。

另一个有趣的功能是传送正面和负面的确认信息（acknowledgment）。

Netlink相对其他用户－内核接口（如ioctl）的优点之一，就是内核可以启动传输，而不只是仅限于响应用户空间请求而返回信息。

配置改变串行化

每次应用配置改变时，内核中内负责处理此事的例程都会取得一个信号量(rtnl_sem)，以确保对储存网络配置内容的数据结构的访问具有互斥性。无论该配置的改变是通过 ioctl 还是 Netlink 而施加，都是如此。

第二部分

系统初始化

此部分说明网络设备初始化以及与内核注册的方式和时机。我会把重点放在PCI（Peripheral Component Interconnect，外设部件互连）设备上，因为PCI设备已逐渐普及，而且有特殊需求。

很多和NIC（network interoce card，网络接口卡）有关的任务都必须先完成，网络才能开启并运行。首先，关键内核组件必须予以初始化。其次，设备驱动程序必须对其所负责的所有设备予以初始化和注册，然后分配一些资源（IRQ、I/O端口等）让内核能使用，以便与驱动程序通信。

区分两种注册是很重要的。首先，当设备被发现时就会向内核注册，成为通用设备。其次，NIC设备和网络协议栈注册时就成了网络设备。例如，PCI Ethernet卡会和PCI层注册成通用的PCI设备，同时和网络协议栈注册成Ethernet卡（在此设备取得*eth0*之类的名称）。第一种注册将在第六章说明，而第二种将在第八章说明。

以下是各章的概要：

第四章 通知链

内核组件用于通知特定事件给其他内核组件的机制。

第五章 网络设备初始化

网络设备如何初始化。

第六章 PCI层和网络接口卡

PCI设备驱动程序如何向内核注册，以及PCI设备如何识别以及和其驱动程序相关联。

第七章 组件初始化的内核基础架构

说明确保必要的初始化函数会在引导期间或模块加载期间启用的内核机制。我们会

了解初始化例程如何标记为特殊宏使其内存用法得以最优化，因而减少内核映像的大小。我们也会了解如何把引导选项传给内核，以及引导选项如何用于配置 NIC。

第八章 设备注册和初始化

设备如何向内核注册以及初始化。

第四章

通知链

内核的很多子系统之间具有很强的相互依赖性，因此，其中一个子系统侦测到的或者产生的事件，其他子系统可能都有兴趣。为了实现这种交互需求，Linux 使用了所谓的通知链（notification chain）。

本章我们将看到：

- 通知链如何声明以及网络代码定义了哪些链（chain）。
- 内核子系统如何向通知链注册。
- 内核子系统如何在链上产生通知信息。

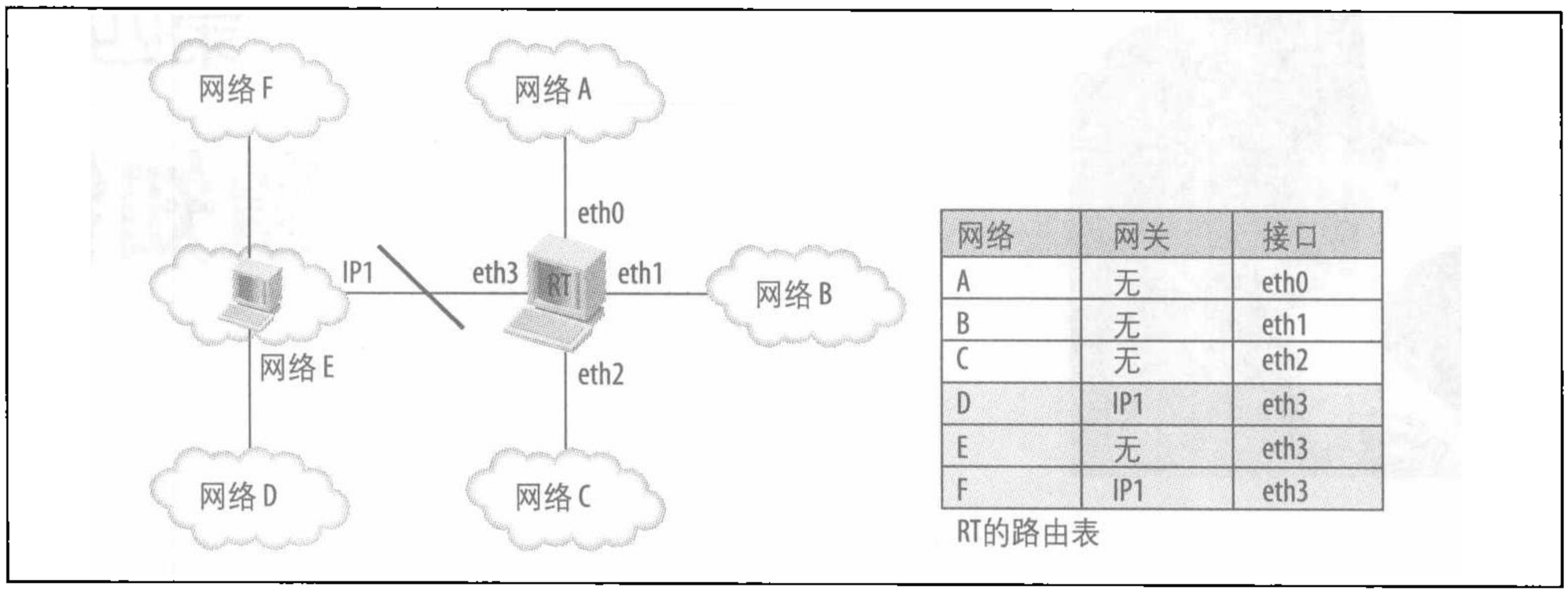
注意，通知链只在内核子系统之间使用。内核和用户空间之间的通知信息则是依赖其他机制，如第三章所介绍的那些。

使用通知链的原因

假设我们有如图 4-1 中所示的带有四个接口的 Linux 路由器。此图显示了路由器与五个网络之间的关系，还有一份简化的路由表。

让我们看看图 4-1 拓扑结构的几个实例。网络 A 直接连接至 RT 的接口 *eth0*，而网络 F 没有直接连接至 RT，但是 RT 的 *eth3* 则直接连接至另一个路由器，其接口的地址为 IP1，并且第二个路由器知道该如何联系网络 F。其他的案例都很类似。简而言之，有些网络是直接连接的，而其他的网络则需要一个或多个附加的路由器协助才能联系得到。

有关路由代码如何处理这种情况的详细描述，可参考第七部分。本章的焦点是通知链的角色。假设接口 *eth3* 由于网络断线使得管理命令（如 ifconfig eth3 down）失效或者造成硬件失败，结果 RT 无法联系网络 D、E 和 F（以及 A、B、C 网络中依赖 RT 联机的系统），因此应该从路由表中删除。谁将通知路由子系统该接口的失效？那就是通知链。



网络	网关	接口
A	无	eth0
B	无	eth1
C	无	eth2
D	IP1	eth3
E	无	eth3
F	IP1	eth3

RT的路由表

图 4-1：Linux 路由器范例

图 4-2 是稍微复杂的例子，路由子系统会与动态路由协议交互，这种协议可以调整一个或多个路由表（注 1），使其吻合网络拓扑结构，因此当拓扑结构许可时就可以处理接口失败的问题（也就是有冗余路径存在）。

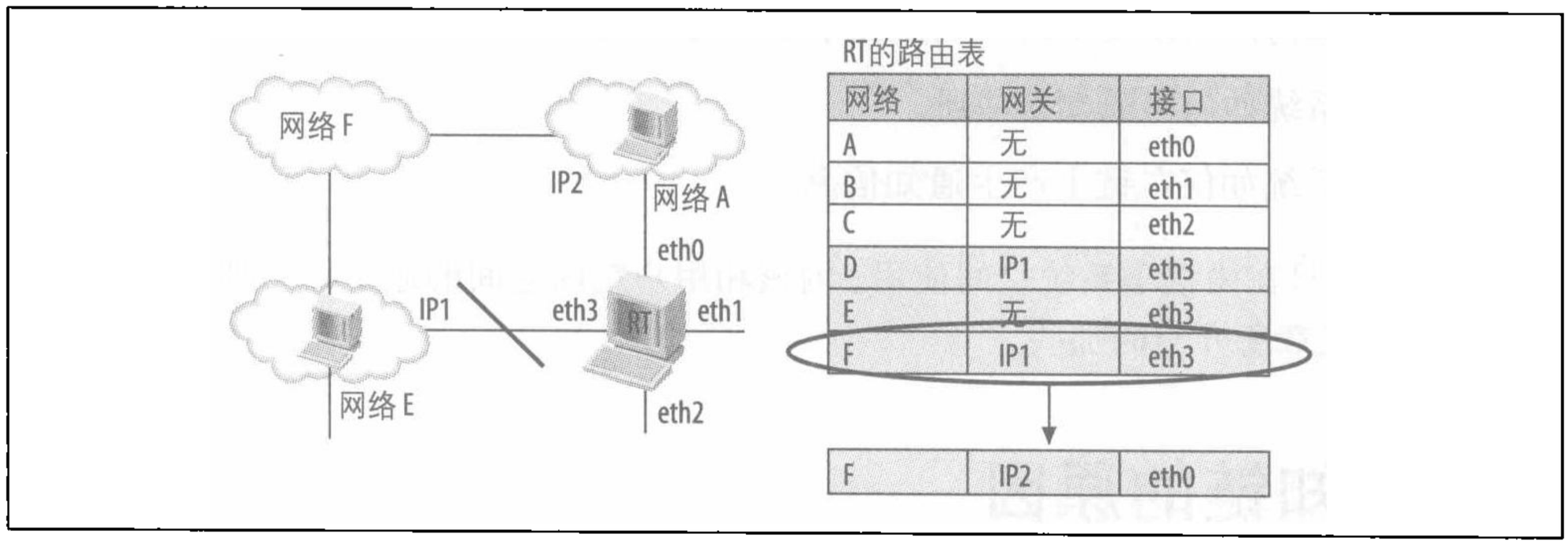


图 4-2：带有动态路由协议的 Linux 路由器范例

在图 4-2 中，RT 可通过网络 A 和网络 E 而联系网络 F。最初会选 E 是因为其成本较少（注 2），但现在 E 已无法联系，因此，路由表应该更新网络 F 的路由改走网络 A。这种决策的基础可能包括一些本地主机事件，诸如设备注册和删除，以及路由器配置中的复杂因素和所用的路由协议。在任何情况下，管理路由表的路由子系统必须从其他子系统那里收到相关信息的通知，因而产生了通知链的需求。

注 1：在同一时刻可能有多个路由表。将在第三十一章说明这一功能。

注 2：链接成本用于比较链接，并从中选择路由协议的权值之一。参见第三十章。

概论

通知链就是一份简单的函数列表，当给定事件发生时予以执行。每个函数都让另一个子系统知道，调用此函数的子系统内所发生的一个事件或者子系统所侦测到的一个事件。

因此，就每条通知链而言，都有被动端（被通知者）和主动端（通知者），也就是所谓的发布－订阅（publish-and-subscribe）模型：

- 被通知者（notified）就是要求接收某事件的子系统，而且会提供回调函数予以调用。
- 通知者（notifier）就是感受到一个事件并调用回呼函数的子系统。

所执行的函数是由被通知的子系统所选取，绝不是链条的拥有者（产生通知信息的子系统）决定该执行什么函数。拥有者只是定义这份列表而已；任何内核子系统都可以对该链条注册一个回调函数以接收通知信息。

通知链的使用使源码更易于编写和维护。想象一下，一个通用函数不使用通知链，该如何把一个事件通知给外部子系统：

```
If (subsystem_X_enabled) {
    do_something_1
}
if (subsystem_Y_enabled) {
    do_something_2
}
If (subsystem_Z_enabled) {
    do_something_3
}
... ... ...
```

换言之，必须为每个可能对一个事件感兴趣的子系统都引入一个条件子句，结果每次某人添加一个子系统给内核时，此子系统的维护者就得添加一个子句。

子系统维护者不可能去追踪每个被添加到内核的子系统。然而，每位子系统维护者应该知道：

- 自己对来自其他子系统的哪种事件感兴趣。
- 自己知道的事件有哪几种，并且其他子系统可能感兴趣的事件又是哪种。

因此，通知链允许每个子系统和其他子系统共享发生的事件，而无需知道究竟是哪些子系统产生事件，以及那些子系统为什么感兴趣。

定义链

通知链列表元素的类型是notifier_block，其定义如下：

```
struct notifier_block
{
    int (*notifier_call)(struct notifier_block *self, unsigned long, void *);
    struct notifier_block *next;
    int priority;
};
```

notifier_call是要执行的函数，next用于链接列表的元素，而priority代表的是该函数的优先级。较高优先级的函数会先被执行。但是在实际中，注册时几乎都不理会notifier_block定义中的priority，意味着获得其默认值为0。因此，执行的次序仅依赖于注册次序（也就是半随机次序）。notifier_call的返回值列在后面一节“链上的通知事件”中。

notifier_block实例的常见名称有*xxx*_chain、*xxx*_notifier_chain以及*xxx*_notifier_list。

链注册

当一个内核组件对给定通知链的事件感兴趣时，可以用通用函数notifier_chain_register予以注册。内核也提供一组内含notifier_chain_register的包裹函数，其中一部分列在表4-1中。

表4-1列出了主要的API以及相关的包裹函数，可用于向三条链inetaddr_chain、inet6addr_chain以及netdev_chain注册及删除。

表4-1：一些链条所用的主要函数和包裹函数

操作	函数原型
注册	int notifier_chain_register(struct notifier_block **list, struct notifier_block *n)
	包裹函数
	inetaddr_chain　　register_inetaddr_notifier
	inet6addr_chain　　register_inet6addr_notifier
	netdev_chain　　register_netdevice_notifier
除名	int notifier_chain_unregister(struct notifier_block **nl, struct notifier_block *n)

表 4-1：一些链条所用的主要函数和包裹函数（续）

操作	函数原型	
	包裹函数	
	inetaddr_chain	unregister_inetaddr_notifier
	inet6addr_chain	unregister_inet6addr_notifier
	netdev_chain	unregister_netdevice_notifier
通知	int notifier_call_chain(struct notifier_block **n, unsigned long val, void *v)	

对于每条链条而言，那些notifier_block实体被插入到一个按优先级排序的列表中。相同优先级的元素则按插入时间排序：新的元素排在尾端。

对通知链的访问受到notifier_lock锁的保护。所有通知链只用一个锁并不是什么大不了的限制，而且也不影响性能，因为子系统通常只是在引导或模块加载时注册其notifier_call函数，然后从那时起对列表的访问都是只读的(即共享)。

因为调用notifier_chain_register函数就是把回调函数插入到所有列表中，因此必须把列表指定为输入参数。然而，此函数很少被直接调用，通常是改用通用的包裹函数。

```
int notifier_chain_register(struct notifier_block **list, struct notifier_block *n)
{
    write_lock(&notifier_lock);
    while(*list)
    {
        if(n->priority > (*list)->priority)
            break;
        list= &((*list)->next);
    }
    n->next = *list;
    *list=n;
    write_unlock(&notifier_lock);
    return 0;
}
```

链上的通知事件

通知信息由定义于*kernel/sys.c* 中的notifier_call_chain产生，而此函数只是按优先级次序调用对此链注册的所有回调函数。注意，回调函数是在调用notifier_call_chain的进程上下文(context)中执行的。然而，回调函数也可以实现成把通知信息排进某处的队列，然后唤醒查看此通知信息的进程。

```
int notifier_call_chain(struct notifier_block **n, unsigned long val, void *v)
```

```
{
    int ret = NOTIFY_DONE;
    struct notifier_block *nb = *n;

    while (nb)
    {
        ret = nb->notifier_call(nb, val, v);
        if (ret & NOTIFY_STOP_MASK)
        {
            return ret;
        }
        nb = nb->next;
    }
    return ret;
}
```

以下是其三个输入参数的意义：

n

通知链。

val

事件类型。链本身标识的一组事件；val明确标识一种事件类型（也就是NETDEV_REGISTER）。

v

此输入参数可由各种各样的客户所注册的处理函数使用，在不同情况下可以有不同用途。例如，当一个新的网络设备在内核注册时，相关的通知信息会使用v以标识net_device数据结构。

notifier_call_chain所调用的回调函数可以返回定义在*include/linux/notifier.h*中的任何NOTIFY_*XXX*的值：

NOTIFY_OK

通知信息被正确地处理了。

NOTIFY_DONE

对通知信息不感兴趣（注3）。

NOTIFY_BAD

有些事情出错。停止调用此事件的回调函数。

NOTIFY_STOP

函数被正确调用。然而，此事件不需要进一步调用其他回调函数。

注3：　这些返回值有些时候用于替代NOTIFY_OK是不恰当的。

NOTIFY_STOP_MASK

此标识由notifier_call_chain检查，以了解是否停止调用回调函数，或者继续调用下去。NOTIFY_BAD和NOTIFY_STOP在其定义中都包括了此标识。

notifier_call_chain捕获并返回由最后一个调用的回调函数所接收的返回值。无论是否所有回调函数都已被调用，或者其中之一由于NOTIFY_BAD或NOTIFY_STOP的返回值中断了循环，都是如此。

注意，在同一时间不同CPU上的相同的通知链有可能同时调用notifier_call_chain。回调函数的责任是在必要的地方处理互斥和串行化。

网络子系统的通知链

内核定义了至少10种不同的通知链。我们感兴趣的是那些对网络代码表征特别重要事件的通知链。主要是：

inetaddr_chain

发送有关本地接口上的IPv4地址的插入、删除以及变更的通知信息。在第二十三章会描述这类通知信息何时产生。IPv6使用类似的链（inet6addr_chain）。

netdev_chain

发送有关网络设备注册状态的通知信息。在第八章会描述这类通知信息何时产生。

对于网络子系统所用的这些链以及其他链而言，其目的和使用都会在相关的关于通知者子系统的章节中描述。

网络代码也可以注册其他内核组件产生的通知信息。例如，某些NIC设备驱动程序可以用reboot_notifier_list链注册；当系统重新引导时，此链会发出警告。

包裹函数

大多数通知链都有一组包裹函数，可以用于注册和删除。例如，下列包裹函数可用于向netdev_chain册：

```
int register_netdevice_notifier(struct notifier_block *nb)
{
        return notifier_chain_register(&netdev_chain, nb);
}
```

包裹函数常见名称包括[un]register_xxx_notifier、xxx_[un]register_notifier以及xxx_[un]register。

范例

通知链注册通常发生在感兴趣的内核组件初始化时。例如，下面是从*net/ipv4/fib_frontend.c*中取出的片段；ip_fib_init就是路由代码所使用的初始化函数。第三十二章的“路由子系统初始化”一节对此进行了描述。

```
static struct notifier_block fib_inetaddr_notifier = {
    .notifier_call = fib_inetaddr_event,
};

static struct notifier_block fib_netdev_notifier = {
   .notifier_call = fib_netdev_event,
};

void _ _init ip_fib_init(void)
{
    ... ... ...
    register_netdevice_notifier(&fib_netdev_notifier);
    register_inetaddr_notifier(&fib_inetaddr_notifier);
}
```

路由代码会在本章稍早“网络子系统的通知链”一节中所提的两条链注册。改变本地IP地址配置以及改变本地设备的注册状态，都会影响路由表。

通过/proc文件系统的调整

就本章而言，与*/proc*中的文件无关。

本章涉及的函数和变量

表4-2总结了本章介绍的函数和数据结构。

表4-2：通知链所用的函数、宏以及数据结构

名称	描述
函数和宏	
notifier_chain_register + 包裹函数 notifier_chain_unregister + 包裹函数 notifier_call_chain	前两个函数是为通知链注册和删除回调处理程序。第三个函数是送出特定种类事件的所有通知信息
数据结构	
struct notifier_block	定义通知信息的处理函数，包括要调用的回调函数

本章涉及的文件和目录

图 4-3 所示为本章涉及的文件。

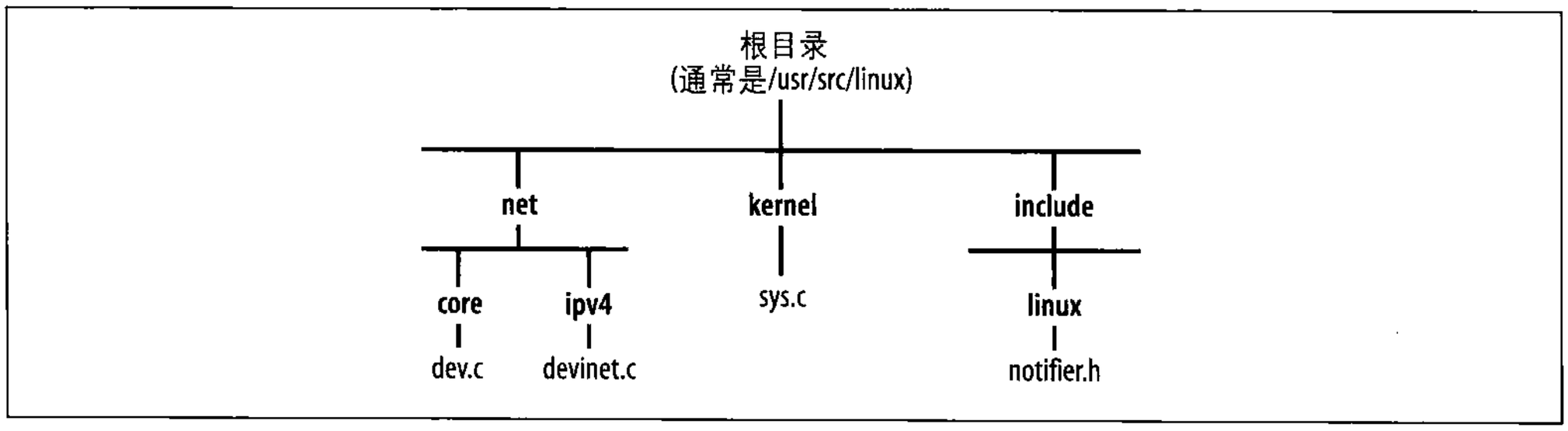


图 4-3：与通知链相关的文件

第五章

网络设备初始化

现代操作系统的适应性增加了初始化的复杂度。首先，设备驱动程序可以作为模块加载至内核，也可以作为内核的静态组件。此外，设备可以在引导期间就存在，也可以在运行期间插入（或删除）：后一种类型的设备就称为可热插拔（hot-pluggable）设备，如USB、PCI CardBus、IEEE 1394（Apple称其为FireWire）以及其他。我们将会看到热插拔对内核和用户空间有什么影响。

本章讨论下列内容：

- 一段内核网络代码的初始化。
- NIC 的初始化。
- NIC如何使用中断，以及IRQ处理函数如何分配和释放。我们也将讨论驱动程序如何共享IRQ。
- 用户如何给以模块方式加载的设备驱动程序提供配置参数。
- 设备初始化及配置期间，用户空间与内核之间的交互。我们将会看到内核如何运行一个用户空间辅助器为NIC加载正确的设备驱动程序，或者应用用户空间配置。特别地，我们将着眼于热插拔功能。
- 就配置以及与内核交互而言，虚拟设备和真实设备之间的差异何在。

系统初始化概论

了解主要的网络相关子系统在何处初始化以及其初始化的方式是很重要的，包括设备驱动程序在内。然而，因为本书只关注网络方面的初始化，因此不会讨论一般内核服务下的设备驱动程序（如内存管理）。想了解这方面的背景，建议阅读O'Reilly出版的《Linux设备驱动程序》和《深入理解Linux 内核》。

图5-1简要地显示了引导期间一些内核子系统在何处初始化以及以何种顺序初始化（参见 *init/main.c*）。

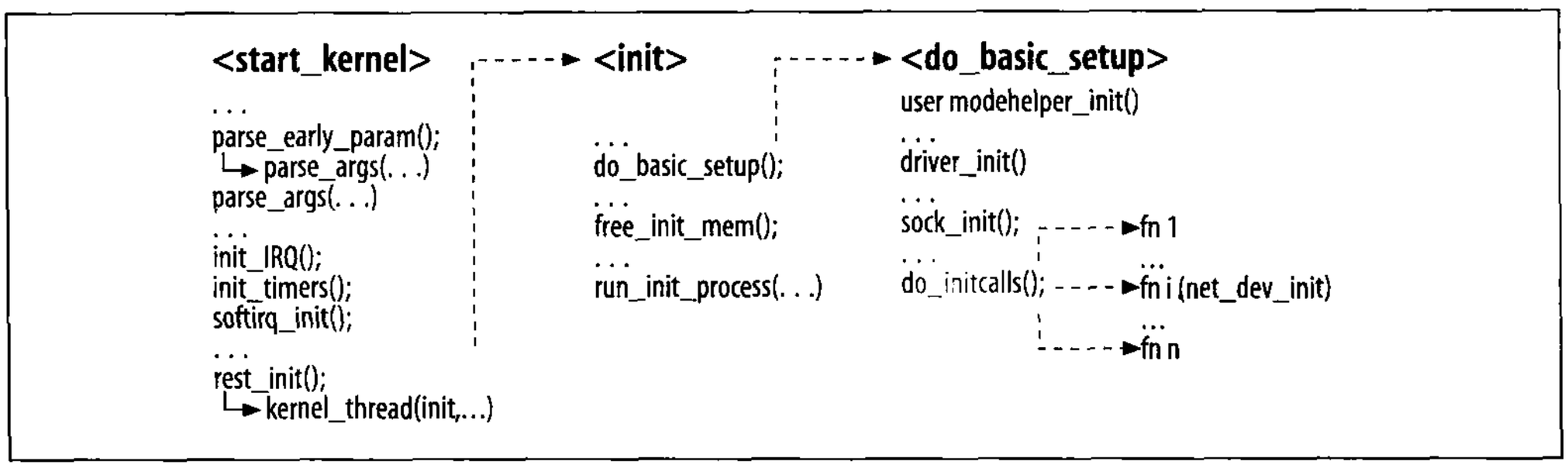


图5-1：内核初始化

当内核引导时，会执行start_kernel对一些子系统做初始化，部分显示于图5-1中。start_kernel终止前会调用init内核线程，由其负责初始化的后续工作。本章所涉及的初始化活动多数都是在do_basic_setup内发生的。

在各种初始化任务中，我们最感兴趣的有三个：

引导期间选项

调用两次parse_args（一次是直接调用，而另一次是通过parse_early_param间接调用）以处理引导加载程序（boot loader，如LILO或GRUB）在引导期间传给内核的配置参数。“引导期间内核选项”一节将说明这项任务是如何处理的。

中断和定时器

硬中断和软中断分别由init_IRQ和softirq_init做初始化。中断会在第九章谈及。本章只说明设备驱动程序如何把处理函数注册给IRQ，以及IRQ处理函数在内存中是如何组织的。定时器在引导过程中很早就被初始化，使得后续的任务可以使用。

初始化函数

内核子系统及内建的设备驱动程序由do_initcalls初始化。free_init_mem会释放一块被无用程序所持有的内存。这项优化成为可能要归功于精明函数标记法（smart routine tagging）。更多的细节参见第七章。

run_init_process确定在系统上运行的第一个进程，也就是所有其他进程的父进程；其PID为1，一直运行直到系统做完工作。正常情况下，运行的程序是*init*（属SysVinit套件的一部分）。然而，管理员可以通过init=引导期间选项指定另一个不同程序。不提供这个选项时，内核就会尝试从一组众所周知的位置去执行*init*命令，如果都找不到*init*，就会发生内核panic。用户也可以提供一些引导期间的选项传给*init*（参见“引导期间内核选项”一节）

设备注册和初始化

一个网络设备可用，就必须被内核认可，并且关联正确的驱动程序。驱动程序把驱动设备所需的所有信息存储在私有数据结构中，然后与其他需要此设备的内核组件交互。注册和初始化任务的一部分由内核负责，而其他部分由设备驱动程序负责。来仔细考察初始化的几个阶段：

硬件初始化

由设备驱动程序和通用总线层（例如，PCI或USB）合作完成。驱动程序有时独自而有时通过用户提供的参数的协助，把每个设备的这类功能配置成IRQ和I/O地址，使其能与内核交互。与较高层的协议和功能相比，这项活动更接近设备驱动程序，因此我们不会在这方面花太多的时间。我们将看一个PCI层的例子。

软件初始化

在设备能够被使用之前，依赖于所开启和配置的网络协议为何而定，用户需要提供其他一些配置参数，如IP地址等。这项任务在其他章说明。

功能初始化

Linux内核有很多网络选项。因为有些选项需要针对每个设备进行配置，因此，设备初始化引导顺序必须负责这些选项。其中一个例子是流量控制（实现QoS的子系统），它可以决定封包加入及退出设备出口队列的方式（以及封包加入及退出入口队列的方式，不过有些限制）。

在第二章我们已经知道`net_device`数据结构包含一组函数指针，内核可用它们与设备驱动程序及特殊内核功能交互。这些函数的初始化部分取决于设备类型（例如Ethernet），部分取决于厂家和型号。由于Ethernet很常见，本章的重点就在Ethernet设备的初始化（但其他设备的过程也很类似）。

在第八章将详细说明设备驱动程序如何采用网络代码使其设备注册。

NIC初始化的基本目标

Linux内核中，每个网络设备都由一个`net_device`数据结构的实例表示。在第八章将了解`net_device`数据结构如何分配以及其字段如何初始化，部分是由设备驱动程序完成，部分是由内核函数完成。本章的重点是设备驱动程序如何分配建立设备/内核通信所需的资源，如：

IRQ线

在“设备与内核之间的交互”一节将会知道，NIC必须被分派一个IRQ，然后在必要

时用它要求内核的注意。然而，虚拟设备不需要分派一个IRQ：回环设备就是一例，因其活动都在内部进行（参见稍后的“虚拟设备”一节）。

用于请求和释放IRQ线（IRQ line）的两个函数会在稍后的“硬件中断”一节介绍。在稍后的“通过/proc文件系统调整”一节将会知道，*/proc/interrupts*文件可用于观察当前分派状态。

I/O端口和内存注册

一般地，驱动程序将其设备的一个内存区域（例如，其配置寄存器）映射到系统内存，使得驱动程序的读/写操作可以通过系统内存地址直接进行，这样可以简化代码。I/O端口和内存分别使用`request_region`和`release_region`注册及释放。

设备与内核之间的交互

几乎所有设备（包括NIC）都采用以下两种方式之一与内核交互：

轮询（polling）

由内核端驱动。内核会定期检查设备状态，以了解其是否发生了什么事情。

中断

由设备端驱动。当设备需要内核注意时，会向内核发送出一个硬件信号（产生中断事件）。

在第九章会详细讨论各种NIC驱动设计方式以及软中断。你也会看到Linux如何结合使用轮询和中断以提高性能。本章我们只谈基于中断的情况。

我不打算详细讨论硬件如何报告中断事件、硬件异常和设备中断之间的差别、驱动程序和总线内核基础架构如何设计等等。这些主题可以参考《Linux设备驱动程序》以及《深入理解Linux内核》。但是，我会对中断做简要概述，以帮助你理解设备驱动程序如何初始化以及为它们所负责的设备进行注册，并且专注于网络方面。

硬件中断

你不需要知道硬件中断如何处理的底层背景。然而，有些细节值得一提，因为这些细节有助于理解NIC设备驱动程序是如何编写的以及它们如何与那些上层网络层交互。

每个中断事件都会运行一个函数，被称为中断处理例程（interrupt handler），而中断处理例程必须按照设备的所需进行裁剪，因此由设备驱动程序安装。一般而言，当设备驱动程序注册一个NIC时，会请求并分派一个IRQ。然后，用两个依赖体系结构的函数为给定的IRQ注册或删除（如果驱动程序被卸载了）处理例程。这两个函数定义在*kernel/*

*irq/manage.c*中，并由*arch/XXX/kernel/irq.c*中的体系结构描述函数改写，其中*XXX*是体系结构描述目录：

```
int request_irq(unsigned int irq, void (*handler)(int, void*, struct
pt_regs*), unsigned long irqflags, const char * devname, void *dev_id)
```

此函数会注册一个处理例程。首先确保所请求的中断是一个有效的中断，而且还没分配给另一个设备，除非这两个设备能够共享IRQ（参见稍后的“中断共享”一节）。

```
void free_irq(unsigned_int irq, void *dev_id)
```

给定的设备由dev_id标识，此函数会删除处理例程，而且如果没有其他设备注册在该IRQ线，就关闭该IRQ。注意，要识别此处理例程时，内核需要IRQ编号以及设备标识符。对共享IRQ而言，这一点尤为重要。稍后的“中断共享”一节对此有解释。

当内核接收到中断通知时，会使用IRQ编号找出该驱动程序的处理例程，然后执行该处理例程。为了找出处理例程，内核把IRQ编号和函数处理例程间的相联关系存储在一张全局表中。相联关系可以是一对一或一对多，因为Linux内核允许几台设备使用相同的IRQ，这种功能会在稍后的“中断共享”一节描述。

在随后的几节中，你会看到设备和驱动程序之间通过中断交换信息的常见示例，以及IRQ如何在某些情况下由几台设备共享。

中断类型

通过中断，NIC能够告知其驱动程序几种不同的事情。其中包括：

接收一帧

这是最常见的、标准的情况。

传输失败

这种通知信息只有当被称为二进制指数后退（exponential binary backoff）功能失败时，才由Ethernet设备产生（这种功能由NIC在硬件层次实现）。注意，驱动程序不会把这种通知信息转送到那些较高层的网络层，这些网络层会通过其他方式得知这种失败（定时器到期，拒绝接受的ACK等等）。

DMA传输已成功完成

给定一个帧传输，当帧上载至NIC的内存准备在此媒介上传输时，驱动程序就会将持有该帧的缓冲区释放掉。使用同步传输时（无DMA），当该帧已上传至NIC，驱动程序就会立刻知道。但使用DMA时，也就是使用异步传输，设备驱动程序必须等待NIC发出明确的中断事件。你可以在驱动程序*drivers/net/3c59x.c*（DMA）和

drivers/net/3c509.c（非DMA）中调用dev_kfree_skb（注1）之处找到每种情况的范例。

设备有足够内存处理新传输

当出口队列没有足够空间保存一个最大尺寸的帧时（例如，Ethernet NIC是1,536个字节），NIC设备驱动程序会停止出口队列而关闭传输，这是很常见的事。当内存可用时，该队列又会再次开启。本节的其他部分将对此情况详加说明。

上述最后一种情况包括了一种巧妙的传送节流（throttling）方式，做得恰当的话可以改善效率。在此系统中，设备驱动程序会在队列空间缺乏时关闭传输，同时要求NIC当可用内存大于给定量时（一般而言就是该设备的MTU）发出一个中断，然后当中断到来时重启传输。

设备驱动程序也可以在传输前关闭出口队列（以防止内核对该设备产生另一次传输请求），然后只有当NIC有足够可用内存时才予以重启。如果没有，该设备会要求一个中断，允许其稍后重新继续传输。下列是此逻辑的范例，取自el3_start_xmit函数，*drivers/net/3c509.c*驱动程序将其安装在net_device结构中，作为其hard_start_xmit（注2）函数：

```
static int
el3_start_xmit(struct sk_buff *skb, struct net_device *dev)
{
    ... ... ...
    netif_stop_queue (dev);
    ... ... ...
    if (inw(ioaddr + TX_FREE) > 1536)
        netif_start_queue(dev);
    else
        outw(SetTxThreshold + 1536, ioaddr + EL3_CMD);
    ... ... ...
}
```

驱动程序可以用netif_stop_queue停止设备队列，因此能禁止内核提交后续的传输请求。然后，驱动程序会检查该设备的内存是否有足够的空间容纳一个1,536个字节的包。如果有，驱动程序就会启动队列，允许内核再次提交传输请求；否则，就会指示设备（通过用一个outw调用写一个配置寄存器），当条件满足时产生一个中断。中断处理例程将使用netif_start_queue重启设备队列，使内核能够重新继续传输。

netif_xxx_queue函数在第十一章“传输的开启和关闭”一节说明。

注1：第十一章会详述此函数。

注2：hard_start_xmit虚拟函数在第十一章描述。

中断共享

IRQ线是有限的资源。增加系统能容纳设备数目的简单方式，就是允许几台设备共享同一个IRQ。正常来讲，每个设备会针对该IRQ将其自己的处理例程注册给内核。再由内核启用那些注册同一个共享IRQ的设备的所有处理例程，而不是由内核接收中断通知，寻找正确的设备，再启用其处理例程。接着由处理例程去过滤出错误的启用，如通过读取其设备上的注册信息。

一组设备共享一条IRQ线时，所有这些设备的设备驱动程序都必须有能力处理共享的IRQ。换言之，每当一个设备注册要使用一条IRQ线时，就必须明确说明其是否支持中断共享。例如，第一个设备注册了一个IRQ，就好像在说："分派IRQ *n* 给我，然后使用*fn*函数作为处理例程。"同时也必须指明其是否愿意和其他设备共享此IRQ。当另一设备试图注册同一个IRQ编号时，如果此设备或者该IRQ当前所分派的设备无法共享IRQ，就会被拒绝。

IRQ处理例程映射的组织

IRQ处理例程的映射存储在一个表向量中，每一个IRQ都对应一个处理例程列表（如图5-2所示）。只有当多台设备共享同一个IRQ时，一个列表中才会有一个以上的元素。向量的尺寸（也就是可能的IRQ编号数目）取决于具体的体系结构，可以从15（x86机器上）变化到200以上。通过刚才介绍的中断共享可知，一个系统可以同时支持更多的设备。

"硬件中断"一节已经介绍了内核提供的两个函数，分别可以对一个处理例程予以注册及删除。现在，来看用于存储此映射的数据结构。

映射用`irqaction`数据结构定义。稍早一节"硬件中断"中所介绍的`request_irq`函数其实就是包含`setup_irq`的包裹函数，用一个`irqaction`结构为其输入，然后将其插入至一个全局向量`irq_desc`。`irq_desc`向量定义在*kernel/irq/handler.c*中，可以在每个体系结构文件*arch/XXX/kernel/irq.c*中被重写。`setup_irq`定义在*kernel/irq/manage.c*中，可以在每个体系结构文件*arch/XXX/kernel/irq.c*中被重写。

处理中断并将其传给驱动程序的内核函数是依赖于体系结构的。在大多数体系结构上，都称之为`handle_IRQ_event`。

图5-2显示了`irqaction`实例的存储方式：对每个可能的IRQ都有一个`irq_desc`实例，而每个成功注册的IRQ处理例程都有一个`irqaction`实例。`irq_desc`实例向量也被称为`irq_desc`，而其尺寸是由体系结构依赖符号`NR_IRQS`指定。

注意，当给定IRQ编号（也就是irq_desc向量的给定元素）有一个以上的irqaction实例时，就需要中断共享（每个结构都必须设置SA_SHIRQ标识）。

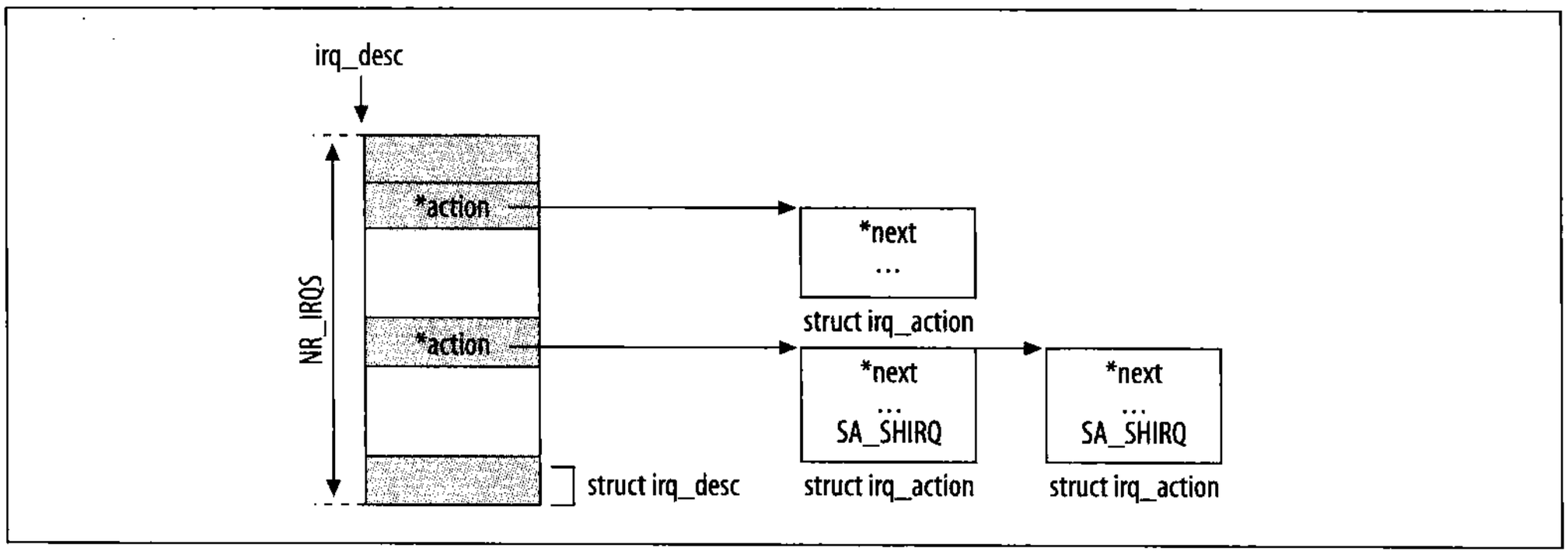


图5-2：IRQ处理例程的组织

现在，让我们看一看irqaction数据结构的字段中存储了哪些与IRQ处理例程相关的信息：

`void (*handler)(int irq, void *dev_id, struct pt_regs *regs)`

由设备驱动程序所提供的函数，用以处理中断的通知信息：每当内核接收irq在线的中断事件时，就调用此例程。以下是此函数的输入参数：

`int irq`

产生此通知信息的IRQ编号。大多数时候，NIC的设备驱动程序不用此参数就能完成其工作，有设备ID就足够了。

`void *dev_id`

设备标识符。同一个驱动程序可能同时要负责不同的设备，所以需要设备ID来正确处理通知信息。

`struct pt_regs *regs`

中断事件打断当前进程时，用于存储处理器的寄存器内容的结构。中断处理例程通常不会用到此结构。

`unsigned long flags`

一组标识。可能的取值SA_XXX定义在*include/asm-XXX/signal.h*中。以下是x86体系结构文件中主要的一些可能取值：

`SA_SHIRQ`

当置位时，设备驱动程序可以处理共享的IRQ。

`SA_SAMPLE_RANDOM`

当置位时，设备将自身变为随机事件的来源。这样有助于协助内核产生供内部使用的随机数，并且对系统熵（system entropy）起一定的作用。稍后的“设备处理层初始化：net_dev_init”一节会进一步描述。

· SA_INTERRUPT

当置位时，处理例程运行在本地处理器上，并且中断处于关闭状态。只有能快速完成处理工作的处理例程，才能指定此值。范例可参见一个handle_IRQ_event的实例（例如*/kernel/irq/handle.c*）。

还有其他值存在，但是，不是淘汰了，就是只用于特定的体系结构。

`void *dev_id`

与此设备相关联的net_device数据结构的指针。声明为void *的原因是，不仅仅只有NIC设备使用IRQ。因为各种设备类型使用不同的数据结构以标识和代表设备实例，因此，这里使用通用的类型声明。

`struct irqaction *next`

所有共享同一个IRQ编号的设备会用此指针链接成一个列表。

`const char *name`

设备名称。你可以通过倾卸*/proc/interrupts*的内容读取设备名称。

初始化选项

内核内建的组件以及作为模块加载的组件都能通过输入参数，使用户调整组件所实现的功能、重写其默认值，或者在系统引导前后有不同的值。

模块选项（module_param系列的宏）

当加载模块时可以提供这些定义选项。当组件内建在内核时，不能在内核引导期间提供这些选项的值。然而，引入*/sys*文件系统后，可以通过这些文件在运行期间配置这些选项。与*/proc*接口相比，*/sys*接口要相对新一些。稍后的“模块选项”一节会对这些选项做更进一步的细节说明。

引导期间内核选项（__setup系列的宏）

在引导加载程序引导期间可以提供这些定义选项。这些选项主要由那些用户可内建在内核的模块以及不能编译成模块的内核组件使用。在第七章的“引导期间内核选项”一节将看到这些宏。

有趣的是，一个模块可以按照两种方式定义初始化选项：当模块内建在内核时，第一种方式有效；当模块独立加载时，另一种方式就可生效。这有点令人困惑，尤其是不同模块可以在模块加载期间定义相同名称的传递参数，而不会有任何名称冲突的风险（也就

是说，这些参数只会传给正在加载的模块），但是如果你在内核引导期间传递这些参数，就必须确保不同模块选项之间没有名称冲突的问题。

我们将不再深入探讨这两种方法的细节。可以看一看*drivers/block/loop.c*驱动程序，其中有使用module_param和__setup的明确范例。

模块选项

内核模块是采用如module_param宏的方式定义其参数。参见*include/linux/moduleparam.h*中的列表。module_param需要三个输入参数，如下示例取自于*drivers/net/sis900.c*：

```
...
module_param(multicast_filter_limit, int 0444);
module_param(max_interrupt_work, int, 0444);
module_param(debug, int, 0444);
...
```

第一个输入参数是给用户使用的参数名称；第二个是参数类型（例如，整型）；第三个表示参数作为文件输出到*/sys*时，分派给该文件的权限。

列出*/sys*中该模块的目录时，会看到下列结果：

```
[root@localhost src]# ls -la /sys/module/sis900/parameters/
total 0
drwxr-xr-x    2 root root    0 Apr  9 18:31 .
drwxr-xr-x    4 root root    0 Apr  9 18:31 ..
-r--r--r--    1 root root    0 Apr  9 18:31 debug
-r--r--r--    1 root root 4096 Apr  9 18:31 max_interrupt_work
-r--r--r--    1 root root 4096 Apr  9 18:31 multicast_filter_limit
[root@localhost src]#
```

每个模块都在*/sys/modules*中被分派一个目录。子目录*/sys/modules/module/parameters*中的每个文件就是该模块所输出的每个参数。刚才从*drivers/net/sis900.c*取出的代码片段中显示出这三个选项可以由任何人读取，但是不可写入（无法改变）。

*/sys*中文件（以及*/proc*的文件）权限的定义与普通文件使用相同的语法，所以，你可以为拥有者、组以及其他所有人指定其读、写和执行权限。例如，400指的是拥有者（也就是root用户）可以读取而其他任何人都无法访问。当权限被分派为0时，表示没有任何人有任何权限，甚至不会在*/sys*中看到该文件。

如果组件程序员想让用户可以读取参数的值，则至少必须给予读取权限。也可以提供写权限，允许用户可以修改参数的值。然而，考虑到输出参数的模块对文件的任何修改并未获得任何通知，所以模块必须有一种机制可监测修改，或者能够处理修改。

有关*/sys*接口的详细描述，可参考《Linux设备驱动程序》一书。

设备处理层初始化：net_dev_init

网络代码初始化的重要部分，包括流量控制和各个CPU（per-CPU）入口队列，在引导期间由定义在*net/core/dev.c*中的net_dev_init完成：

```
static int _ _init net_dev_init(void)
{
    ...
}
subsys_initcall(net_dev_init);
```

在第七章将说明subsys_initcall宏如何确保任何NIC设备驱动程序自行注册前net_dev_init会先执行，以及为什么如此做这一点很重要。你也会知道为什么net_dev_init用__init宏标记。

让我们浏览net_dev_init的主要部分：

- 由两个网络软件中断（softirq）所使用的对应各个CPU的数据结构被初始化。在第九章，我们就将说明什么是softirg，然后详细说明网络代码如何使用softirg。
- 当内核被编译为支持*/proc*文件系统时（默认配置），有一些文件会通过dev_proc_init和dev_mcast_init添加到*/proc*。更多细节参见稍后的“通过/proc文件系统调整”一节。
- netdev_sysfs_init向*sysfs*注册为net类。这样会创建*/sys/class/net*目录，在该目录下面，每个已注册的网络设备都会有一个子目录。这些目录包括很多文件，其中有些以前是放在*/proc*中的。
- net_random_init会初始化一个对应各个CPU的种子向量，用于net_random函数产生随机数。net_random用在各种不同的环境中，本节稍后会加以描述。
- 用于协议无关的目的缓存（protocol-independent destination cache，DST）会以dst_init做初始化，对此在第三十三章描述。
- 协议处理例程向量ptype_base初始化，用于分离入口流量的多路合并传输。更多细节参见第十三章。
- 当定义OFFLINE_SAMPLE符号时，内核会建立一个定期执行的函数，以收集有关设备队列长度的统计数据。在这种情况下，net_dev_init需要建立定时器，以定期执行该函数。参见第十章的“平均队列长度及拥塞等级计算”一节。

- 把一个回调处理例程注册到发出有关CPU热插拔事件的通知信息的通知链。使用的回调函数为dev_cpu_callback。目前，唯一处理的事件就是CPU停止事件。当接收到此通知消息时，CPU入口队列里的缓冲区出列，然后传给netif_rx。参见第九章有关对应各个CPU入口队列的细节。

随机数的产生是通过一个辅助函数，内核借此函数令其自身某些活动随机化。在本书中，你将看到很多网络子系统使用随机产生的值。例如，那些网络子系统通常会对定时器的延时加入一个随机组件，使得定时器不太可能同时执行，而且通过后台处理使CPU的负载下降。随机化也可防御某些试图猜出特定数据结构组织的人所进行的DoS（Denial of Service，服务阻绝）攻击。

内核的数字可被视为真正随机的程度就称为系统熵。系统熵的改进是通过一些具有非确定性活动（nondeterministic aspect）的内核组件的作用，而网络通常都在此范畴内。目前而言，只有少数NIC设备驱动程序对系统熵有作用(参见稍早对SA_SAMPLE_RANDOM的讨论)。内核2.4版本的补丁多加了一个编译期间选项，可让你开启或关闭NIC对系统熵的作用。使用关键词“SA_SAMPLE_NET_RANDOM”在网上搜索，就会发现当前的版本。

旧代码

前一节曾提到，subsys_initcall宏要确保net_dev_init在任何设备驱动程序有机会注册其设备之前就执行。引入这种机制之前，使用一次标识（one-time flag）的旧式机制，执行次序通常被强制有所不同。

全局变量dev_boot_phase以前作为布尔标识，用来记住net_dev_init是否已被执行。其初值为1（也就是net_dev_init还没被执行），然后由net_dev_init清除。每次register_netdevice被一个设备驱动程序调用时，都会检查dev_boot_phase的值。如果该标识被设置表示该函数尚未执行，于是就执行net_dev_init。

这种机制已经不再使用了，因为如果关键的设备驱动程序的例程已经被正确标记，register_netdevice就不能在net_dev_init之前调用，对此将在第七章描述。然而，为了侦测错误的标记或者有bug的代码，net_dev_init仍然会清除dev_boot_phase的值，而register_netdevice会使用宏BUG_ON以确保当dev_boot_phase设置时，其绝不会被调用（注3）。

注3：宏BUG_ON和BUG_TRAP的使用都是采用相同的机制，确保特定码点（code point）之处必要的条件能满足，而且在从一种设计转换至另一种时也有其用处。

用户空间辅助程序

有些情况下，内核调用用户空间应用程序以处理事件也是讲得通的。有两个辅助程序特别重要：

/sbin/modprobe

当内核需要加载模块时就会被调用。这个辅助程序是*module-init-tools*套件的一部分。

/sbin/hotplug

当内核侦测到一个新设备已经插入或拔出系统时就会被调用。其主要的工作是根据设备标识符加载正确的设备驱动程序（模块）。设备的识别是通过其所插入的总线（例如，PCI）以及总线规范（注4）所定义的相关联的ID。这个辅助程序是*hotplug*套件的一部分。

内核提供一个名为`call_usermodehelper`的函数，以执行这类用户空间辅助程序。此函数允许调用者通过`arg[]`传递一些自变量，并通过`env[]`传递一些环境变量给应用程序。例如，第一个自变量`arg[0]`通知`call_usermodehelper`要启用哪个用户空间辅助程序，而`arg[1]`可以用于通知辅助程序该使用什么配置脚本（script）（通常称为用户空间代理）。稍后一节“/sbin/hotplug”会看到实例。

图5-3显示了两个内核函数`request_module`和`kobject_hotplug`如何调用`call_usermodehelper`，以调用*/sbin/modprobe*和*/sbin/hotplug*。图中也有`arg[]`和`envp[]`在两种情况下初始化的例子。后续几小节将对这两个用户空间辅助程序做更详细的说明。

kmod

`kmod`是内核模块加载程序，允许内核组件请求加载一个模块。内核提供的函数不止一个，但这里我们只谈`request_module`。此函数用要加载的模块名字初始化`arg[1]`。*/sbin/modprobe*使用配置文件*/etc/modprobe.conf*去做各式各样的事情，其中之一就是去了解从内核所接收的模块名字实际上是否为其他模块的别名（参见图5-3）。

以下是导致内核要求*/sbin/modprobe*加载一个模块的两个事件示例：

注4：参考第六章的“PCI NIC设备驱动程序的注册”一节中涉及的PCI示例。

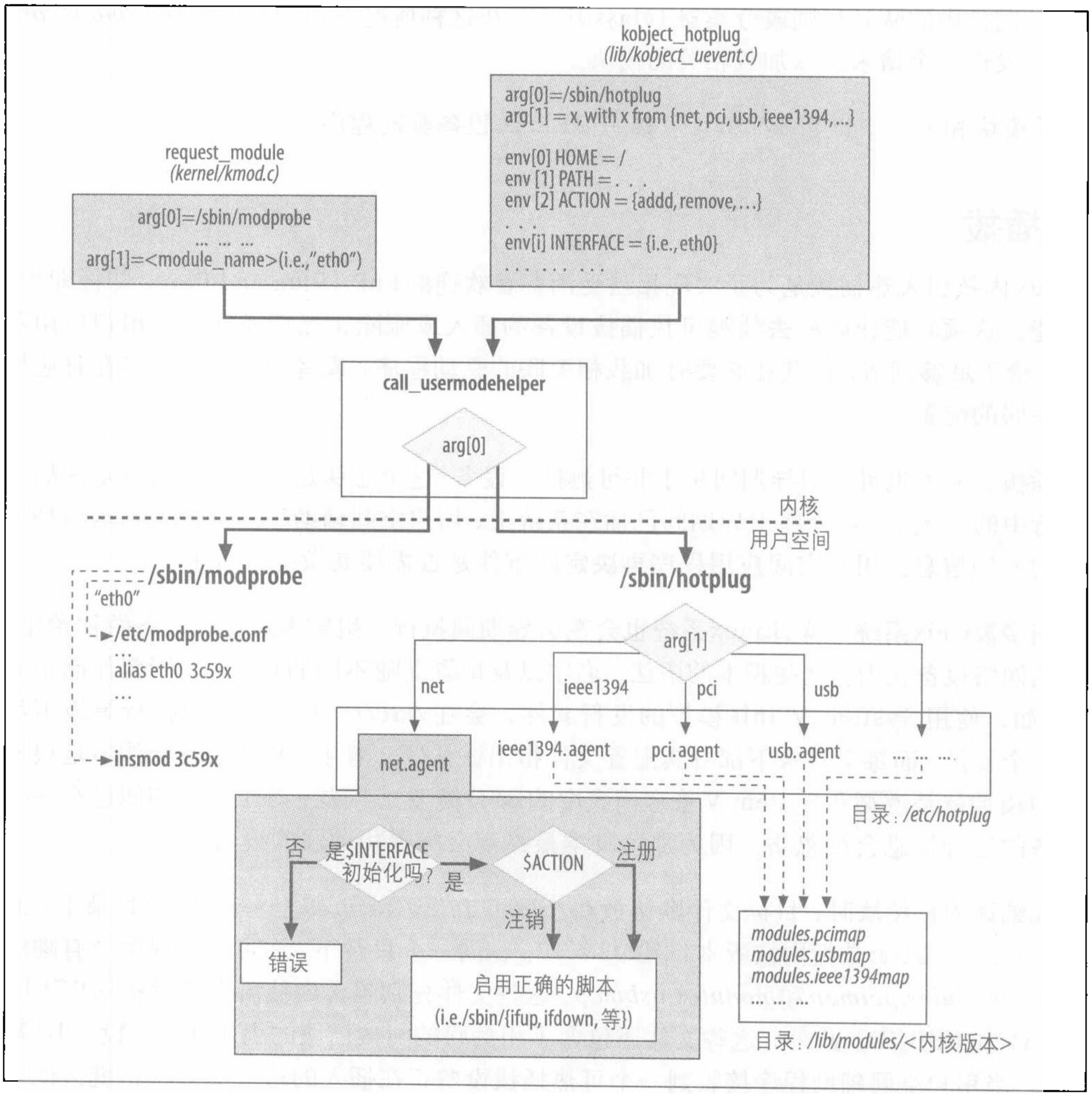


图 5-3：事件从内核到用户空间的传播

- 当管理员使用*ifconfig*配置一张设备驱动程序尚未加载的网卡 —— 例如，设备eth0（注5）时 —— 则内核向*/sbin/modprobe*送出一个请求，以加载名称为字符串"eth0"的模块。如果*/etc/prorobe.conf*包含"alias eth0 3c59x"条目，则*/sbin/modprobe*会尝试加载模块*3c59x.ko*。
- 当管理员以IPROUTE2包的*tc*命令配置一个设备的流量控制时，可能会涉及不在

注5：　注意，因为设备驱动程序尚未加载，*eth0*也还未存在。

内核内的队列规则或分类器（classifier）。在这种情况下，内核将向*/sbin/modprobe*发送一个请求，以加载相关的模块。

有关模块和kmod的更多细节，可参考《Linux设备驱动程序》一书。

热插拔

Linux内核引入热插拔是为了实现相当受消费者欢迎的PnP（Plug and Play，即插即用）功能。这项功能让内核去侦测可热插拔设备的插入或删除，然后通知一个用户应用程序，给予足够细节，使其在必要时加载相关联的驱动程序，或者当驱动程序存在时应用相关联的配置。

热插拔实际上也可在引导期间用于非可热插拔设备。这个想法是无论一个设备是热插在运行中的系统上，或者在引导期间已插在系统上，用户空间辅助程序都会收到这两种情况的通知信息。用户空间应用程序再决定此事件是否需要其做些什么事。

如同多数Unix系统一样，Linux系统也会在引导期间执行一组脚本对接口设备做初始化，包括网络设备在内。这些脚本的语法、名字以及位置会随不同的Linux发行套件而不同（例如，使用System V init模型的发行套件，会在*/etc/rc.d/*下的每个执行等级下都有一个目录，而每个目录下都有其配置文件指出该从什么着手。其他发行套件不是根据BSD模型就是遵循和System V兼容模式中的BSD模型）。因此，对于引导期间已存在的设备的通知信息会被忽略，因为那些脚本最终都会配置相关联的设备。

当你编译内核模块时，目标文件默认放在*/lib/modules/kernel_version/*目录下，而kernel_version是指内核版本，例如2.6.12。在同一个目录下，你可以发现两个有趣的文件：*modules.pcimap*和*modules.usbmap*。这些文件分别包含内核所支持设备的PCI ID（注6）和USB ID。此外，这些文件还包含了相关联的内核模块的引用（每个设备ID都有）。当用户空间辅助程序接收到一个可热插拔设备正在插入的通知信息时，就会使用这些文件找出正确的设备驱动程序。

*modules.xxxmap*文件的填写数据来自于设备驱动程序所提供的ID向量。例如，在第六章的“PCI NIC驱动程序注册范例”一节就会知道，Vortex驱动程序会对其pci_device_id实例做初始化。因为该驱动程序是针对PCI设备编写的，因此，该表的内容会加入到*modules.pcimap*文件中。

如果你对最新代码有兴趣，可以在*http://linux-hotplug.sourceforge.net*找到更多信息。

注6：第六章“PCI NIC驱动程序注册范例”一节给出了PCI设备标识符的简述。

/sbin/hotplug

Hotplug默认的用户空间辅助程序是脚本（注7）*/sbin/hotplug*，该脚本是Hotplug套件的一部分。这个套件可以通过默认目录*/etc/hotplug/*和*/etc/hotplug.d/*中的文件进行配置。

内核会调用kobject_hotplug函数以响应一个设备的插入和删除以及其他事件。kobject_hotplug会把arg[0]的初值设为*/sbin/hotplug*，把arg[1]设为要使用的代理程序：*/sbin/hotplug*是一个简单的脚本，把事件的处理委托给arg[1]指定的另一个脚本（代理程序）。

根据你想要的自动配置有多奇异而定，用户空间辅助代理程序可以很复杂或很简单。Hotplug套件提供的脚本会尝试认出Linux发行版本，然后根据其配置文件的语法和位置调整其行为。

我们以本书的主题“网络”作为热插拔的范例。当一块NIC添加到系统或从系统删除时，kobject_hotplug会把arg[1]的初值设为*net*，使得*/sbin/hotplug*去执行*net.agent*代理程序。

与图5-3所示的其他代理程序不同的是，*net.agent*并不代表一种媒体或总线类型。虽然*net*代理程序可用于配置设备，但其他代理程序可根据设备标识符而加载正确的模块（设备驱动程序）。

*net.agent*可以应用与新设备相关联的任何配置，所以内核至少必须提供设备标识符。在图5-3所示的范例中，设备标识符是由内核通过INTERFACE环境变量传入的。

为了能够配置设备，此程序必须先由内核创建并注册。这项任务通常是由相关联的设备驱动程序驱动，当然必须先加载才行。例如，添加一块PCMCIA Ethernet卡会引起调用*/sbin/hotplug*好几次，包括：

- 导致*/sbin/modprobe*（注8）执行，借此加载正确的模块设备驱动程序。就PCMCIA而言，驱动程序是由*pci.agent*代理程序加载（使用动作ADD）。
- 配置新设备。这是由*net.agent*代理程序所做的事（再度使用动作ADD）。

注7：管理员可以写自己的脚本，或者使用最常见Linux发行套件所提供的脚本。

注8：与*/sbin/hotplug*（shell脚本）不同的是，*/sbin/modprobe*是二进制可执行文件。如果你想看的话，可以下载*modutil*套件的源码。

虚拟设备

虚拟设备是建立在一个或多个真实设备之上的抽象。虚拟设备与真实设备之间的关联可以是多对多，如图5-4所示的三种模型。也有可能在其他虚拟设备之上建立虚拟设备。然而，并非所有组合都有意义，也并非所有组合都有内核支持。

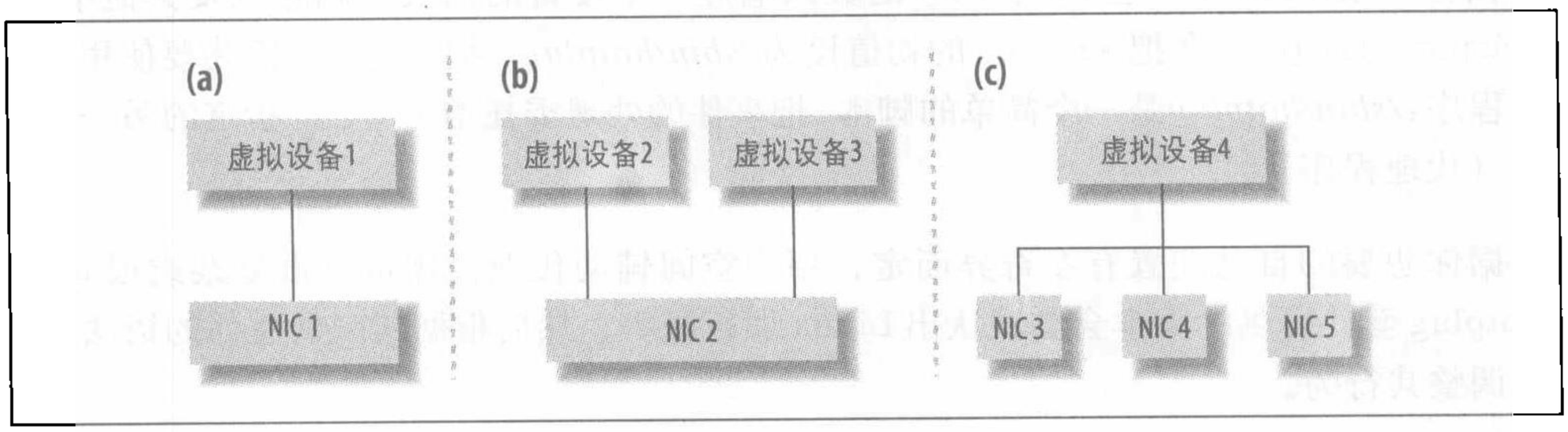


图5-4：虚拟设备与真实设备之间可能的关系

虚拟设备范例

Linux允许你定义多种不同的虚拟设备。以下是一些范例：

绑定（bonding）

利用这个功能，虚拟设备可以绑定一组物理设备，使其如同单一设备。

802.1Q

这是一种IEEE标准，以所谓的VLAN报头扩充802.3/Ethernet帧头，因而得以建立VLAN。

桥接（bridging）

桥接接口就是网桥的虚拟代表。其细节见第四部分。

别名接口（aliasing interface）

原本这项功能的主要目的是允许单一真实的Ethernet接口横跨几个虚拟接口（*eth0:0*、*eth0:1*等等），每一个接口都有其自己的IP配置。现在，由于网络代码的改进，在同一个NIC上配置多个IP地址时已经不需要定义一个新的虚拟接口。然而，有些情况下（特别是路由），让同一个NIC具有不同的虚拟NIC可能会轻松一点，也许会使得配置简化一点。详情见第三十章。

普通均衡器（true equalizer，TEQL）

这是队列规则，用于流量控制。其实现需要建立一个特殊设备。TEQL背后的想法有点类似Bonding。

隧道接口（tunnel interface）

IP-over-IP（IPIP）隧道以及GRE（Generalized Routing Encapsulation，通用路由封装）协议的实现基础是虚拟设备的建立。

这份列表并不完整。此外，由于新功能加入Linux内核的速度相当快，你可以预期会有新的虚拟设备加入内核之中。

绑定、桥接以及802.1Q设备都是图5-4（c）所示模型的范例。别名接口则是图5-4（b）所示模型的范例。图5-4（a）所示模型则可视为其他两种模型的特殊情况。

与内核网络协议栈之间的交互

虚拟设备和真实设备与内核的交互方式略有不同。例如，它们在下列几点上有差别：

初始化

大多数虚拟设备如同真实设备一样，都会被分派一个net_device数据结构。通常，大多数虚拟设备的net_device函数指针所初始化成的函数都是所谓的包裹函数，有的比较复杂，而有的比较简单，其中所包含的函数指针都是相关的真实设备所用的函数指针。

然而，并非所有虚拟设备都会被分派一个net_device实例。别名设备就是一例；别名设备作为相关联的真实设备上的简单标签（label）来实现（参见第三十章的“老版本的配置：接口别名”一节。

配置

提供特殊用户空间工具来配置虚拟设备是很常见的事，尤其是那些只应用于这些虚拟设备的高层字段，无法使用如*ifconfig*标准工具予以配置。

外部接口

每个虚拟设备通常都会输出一个文件或一个内含一些文件的目录到*/proc*文件系统内。这些文件所输出的信息有多复杂和详尽，则取决于虚拟设备的种类及其设计。在“虚拟设备”一节将列出每种虚拟设备所用的文件以及其相关的章节（本书中所提及的那些设备）。与虚拟设备相关的文件属于额外文件，不会替代与物理设备相关的文件。不过，本身没有自己的net_device实例的别名设备是一个例外。

传输

当虚拟设备与真实设备之间的关系不是一对一时，用于传输的函数可能需要选择所用的真实设备以及其他任务（注9）。因为QoS是按照每个设备的需要而应用的，虚拟设备和相关的真实设备之间有多重关系时，就意味着要配置流量控制。

注9：参考第十一章有关封包传输的一般性议题，特别是dev_queue_xmit。

接收

因为虚拟设备都是软件对象，不需要参与系统上真实资源的交互，如注册IRQ处理例程或者分配I/O端口以及I/O内存。其流量是间接获得的，来自于执行这些任务的物理设备。不同种类的虚拟设备在封包接收时各有不同的行为。例如，802.1Q接口会注册一个Ethertype，然后只让相关联的真实设备所接收的携带了正确协议ID（注10）的封包通过。相反，桥接接口则会接收来自于相关联设备的任何包（参见第十六章）。

外部通知信息

内核中发生其他内核组件对特定事件发出通知信息，对虚拟设备而言，就如同真实设备一样，依然有其利益所在（注11）。因为虚拟设备的逻辑是实现在真实设备之上的，真实设备对此逻辑毫无所知，因此无法把这些通知信息传过去。因此，通知信息必须直接面向虚拟设备。我们以Bonding为例说明：如果绑定设备群中有一台设备宕机了，则用于把流量散布在绑定设备群中成员的算法必须察觉到这一点，才不会选到已无法使用的设备。

与这些软件触发的通知信息不同的是，硬件触发的通知信息（例如，PCI电源管理）就无法直接到达虚拟设备，因为没有和虚拟设备相关的硬件。

通过/proc文件系统调整

图5-5所示为可用于调整或观察与本章主题相关的配置参数文件。

在*/proc/sys/kernel*中的文件*modprobe*和*hotplug*，可以修改稍早的“用户空间辅助程序”一节所介绍的两个程序的路径名称。

*/proc*中的一些文件会输出内部数据结构和配置参数的值，有助于记录设备驱动程序分配了哪些资源；如稍早的“NIC初始化的基本目标”一节所示。就这些数据结构的其中几个而言，提供了用户空间命令可以采用较为友善的格式打印出其内容。例如，*lsmod*以*/proc/modules*作为其信息来源，可列出当前加载的模块。

在*/proc/net*中，你可以找到由net_dev_init通过dev_proc_init和dev_mcast_init所创建的文件（参见稍早的“设备处理层初始化：net_dev_init”一节）。

注10： 第十三章会讨论根据协议标识符而分离入口流量的多路合并传输。

注11： 第四章定义了通知链，并说明通知链可用于通知哪些事情。

dev

对每个已注册至内核的网络设备而言，会显示一些有关接收和传输的统计数据，如已接收或已传输的字节数、封包数目以及错误等等。

dev_mcast

对每个已注册至内核的网络设备而言，会显示一些由 IP 多播所使用的参数值。

wireless

类似于*dev*，对每个无线设备而言，会打印出来自于dev->get_wireless_stats虚拟函数所返回的无线区块中的一些参数值。注意，dev->get_wireless_stats返回的信息只适用于无线设备，因为无线设备会分配一个数据结构以保存这些统计数据（所以，*/proc/net/wireless* 将只包含无线设备）。

softnet_stat

输出有关由网络代码所用的软件中断的统计数据，参见第十二章。

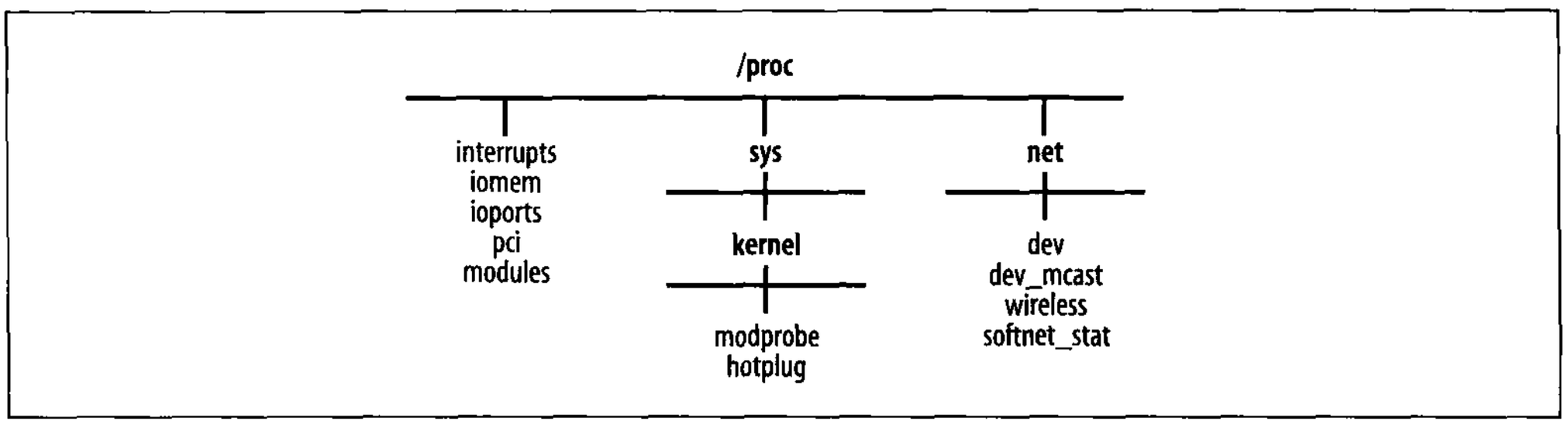


图 5-5：与路由子系统相关的 /proc 文件

还有其他有趣的目录存在，包括 */proc/drivers*、*/proc/bus* 以及 */proc/irq*，建议你去看《Linux 设备驱动程序》。此外，内核参数已逐渐从*/proc* 移出，而移进一个名为*/sys* 的目录，但因为篇幅有限，不再描述这个新系统。

本章涉及的函数和变量

表 5-1 总结本章介绍的函数、宏、变量以及数据结构。

表 5-1：与系统初始化相关的函数、宏、变量以及数据结构

名称	描述
函数和宏	
request_irq free_irq	分别为 IRQ 线注册和释放回调处理函数。注册可以独占或共享

表5-1：与系统初始化相关的函数、宏、变量以及数据结构（续）

名称	描述
request_region release_region	分配和释放I/O端口及I/O内存
call_usermodehelper	调用用户空间辅助程序应用
module_param	用于定义模块所需的配置参数的宏
net_dev_init	在引导期间对一段网络代码做初始化
全局变量	
dev_boot_phase	旧代码所用的布尔标识，以强制在NIC设备驱动程序自行注册前，执行net_dev_init
irq_desc	指向IRQ描述符向量的指针
数据结构	
struct irq_action	每条IRQ线都由此结构的实例定义。除了其他字段外，还包括一个回调函数
net_device	描述网络设备

本章涉及的文件和目录

图5-6列出了本章涉及的文件和目录。

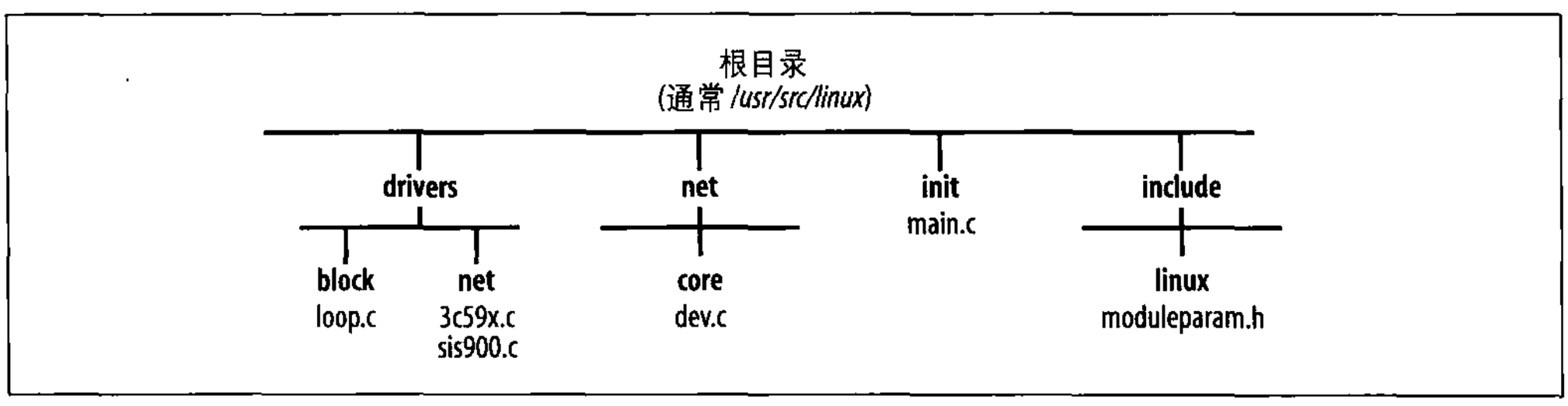


图5-6：本章涉及的文件和目录

第六章

PCI层和网络接口卡

由于PCI总线在x86和其他体系结构上都很流行，我们要花一些篇幅讨论，以便于你了解内核如何管理PCI设备，这里我们特别强调网络设备。本章将有助于你了解第八章有关设备注册代码的背景知识。你也将学到一些PCI如何处理一些很好的内核功能，如探测和电源管理的知识。至于对PCI的深入讨论，如设备驱动程序设计、PCI总线功能以及实现的细节，可以参考《Linux设备驱动程序》和《深入理解Linux内核》，以及PCI规范。

内核中的PCI子系统（也称为PCI层）提供各种PCI设备驱动程序共同的所有通用功能。这个子系统让程序员减少了很多必须对每种设备所做的事，让驱动程序能以更简明的方式编写，使得内核更易于收集和维护有关各种设备的信息，如描述信息和统计数据。

本章我们将说明PCI层所用的一些关键数据结构，以及这些结构如何由一个共同的NIC设备驱动程序予以初始化。我也将对PCI电源管理以及网络唤醒（Wake-on-LAN，WOL）功能给出一些结论。

本章涉及的数据结构

在此列出PCI层使用的一些关键数据结构类型。还有很多其他类型，但是下列几种是阅读本书时所必须知道的。第一个结构定义在*include/linux/mod_devicetable.h*中，而另外两个定义在*include/linux/pci.h*中。

`pci_device_id`

设备标识符。这不是Linux所使用的本地ID，而是根据PCI标准所定义的ID。稍后的“PCI NIC设备驱动程序的注册”一节会说明此ID的定义，而稍后的“PCI NIC驱动程序注册范例”一节会介绍一个范例。

pci_dev

每个PCI设备都会被分派一个pci_dev实例，如同网络设备都会被分派net_device实例一样。这个结构由内核使用，以引用一个PCI设备。

pci_driver

定义PCI层和设备驱动程序之间的接口。这个结构主要由函数指针组成。所有PCI设备都会使用这个结构。稍后的“PCI NIC驱动程序注册范例”一节会对其进行定义，并给出关于其初始化的一个范例。

PCI设备驱动程序由pci_driver结构的实例定义。以下是其主要字段的说明，特别注意NIC设备的情况。函数指针会由设备驱动程序初始化为该驱动程序内适当的函数。

char *name

驱动程序的名称。

const struct pci_device_id *id_table

这是一个ID向量，内核用于把一些设备关联到此驱动程序。“PCI NIC驱动程序注册范例”一节会展示一个范例。

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

当PCI层发现它正在搜寻驱动程序的设备ID与前面所提到的id_table匹配，就会调用此函数。此函数应该开启硬件、分配net_device结构、初始化并注册新设备（注1）。此函数中，驱动程序也会分配正确工作所需的所有数据结构（如传输或接收时所用的缓冲区）。

void (*remove)(struct pci_dev *dev)

当驱动程序从内核除名时，或者当可热插拔设备被删除时，PCI层就会调用此函数。此函数为probe的配对函数，用于清理任何数据结构和状态。

网络设备使用此函数来释放已分配的I/O端口和I/O内存，为设备除名，释放net_device数据结构以及其他由设备驱动程序在probe函数内所分配的辅助数据结构。

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

int (*resume)(struct pci_dev *dev)

当系统进入挂起模式以及重新继续时，PCI层就会调用这些函数。参见稍后的“电源管理和网络唤醒”一节。

int (*enable_wake)(struct pci_dev *dev, u32 state, int enable)

利用这个函数，驱动程序可以通过产生特定的电源管理事件（Power Management

注1：关于NIC注册将在第八章讨论。

Event）信号，开启或关闭设备唤醒系统的能力。参见稍后的“电源管理和网络唤醒”一节。

`struct pci_dynids dynids`

动态ID。参见下一节。

参见稍后的“PCI NIC驱动程序注册范例”一节中有关`pci_driver`实例的初始化范例。

PCI NIC设备驱动程序的注册

PCI设备独一无二的识别方式是通过一些参数的组合，包括开发商以及模型等。这些参数由内核存储在`pci_device_id`类型的数据结构中，定义如下：

```
struct pci_device_id {
    unsigned int vendor, device;
    unsigned int subvendor, subdevice;
    unsigned int class, class_mask;
    unsigned long driver_data;
};
```

这些字段大多数都是不需要加以说明的。`vendor`和`device`通常就足以识别设备。`subvendor`和`subdevice`很少用到，通常设置为通配符（`PCI_ANY_ID`）。`class`和`class_mask`代表该设备所属的类，而NETWORK就是本章所要讨论的设备所属的类。`driver_data`不是PCI ID的一部分，而是由驱动程序所使用的一个私有参数。

每个设备驱动程序都会把一个`pci_device_id`实例的向量注册到内核，这个实例向量列出了其所能处理的设备ID。

PCI设备驱动程序分别用`pci_register_driver`和`pci_unregister_driver`向内核注册和除名。这些函数定义在*drivers/pci/pci.c*中。此外，还有一个`pci_module_init`，是`pci_register_driver`的别名。引入`pci_register_driver`之前，旧版本内核中所用的函数名为`pci_module_init`，有些驱动程序依然使用它。

`pci_register_driver`需要一个`pci_driver`数据结构作为自变量。借助于`pci_driver`的`id_table`向量，内核知道该驱动程序可以处理哪些设备，此外，也因为`pci_driver`中所有虚拟函数，使内核有一种机制，可以与此驱动程序的任何相关联的设备彼此交互。

PCI的优点之一是，其支持寻找IRQ和每个设备所需的其他资源的探测方式（probing）相当优雅。模块可以在加载期间接收一些输入参数，以告知该如何配置其所负责的所有设备。但是，有些时候（特别是PCI这类总线）让驱动程序自行检查系统上的设备，然后为其负责的那些设备做配置会比较简单一点。必要时，用户依然可以退回到手动配置。

/sys 文件系统输出有关系统总线（PCI、USB 等等）的信息，包括各种设备及各种设备之间的关系。*/sys* 也允许管理员为特定的设备驱动程序定义新的ID，使得除了驱动程序通过其pci_driver结构的id_table向量注册的静态ID之外，内核还能使用由用户所配置的参数。

我们不会说明内核根据设备ID查询驱动程序所采用的探测机制。然而，值得一提的是，探测方式有两种：

静态

给定一个设备 PCI ID，内核就能根据 id_table 向量查询出正确的PCI驱动程序（也就是pci_driver实例）。这称为静态探测方式。

动态

这种查询是根据用户手动配置的ID，这种情况在实际中很少见，但是，在调试的情况下偶尔也使用。动态指的是系统管理员可以新增ID的能力，而不是指ID本身可自行变动。

由于动态ID是在运行中的系统上配置的，只有当内核被编译成支持热插拔时才能使用。

电源管理和网络唤醒

PCI电源管理事件由pci_driver数据结构的suspend和resume函数处理。除了分别负责PCI状态的保存和恢复之外，这些函数遇到NIC的情况时还需要采取特殊步骤：

- suspend主要停止设备出口队列，使得该设备无法再传输。
- resume重启出口队列，使得该设备得以再次传输。

网络唤醒（Wake-on-Lan，WOL）是一种功能，允许NIC在接收到一种特殊类型的帧时唤醒处于待命模式的系统。WOL通常默认是关闭的。此功能可以用pci_enable_wake打开或关上。

当首次引入WOL功能时，只有一种帧可以唤醒系统："魔术封包（Magic Packet）"（注2）。这类特殊帧有两个主要特征：

- 目的 MAC 地址属于正在接收的 NIC（无论该地址为单播、多播还是广播）。
- 帧中的某处（任何地方）会设置一段 48 位序列（也就是 FF:FF:FF:FF:FF:FF），后面再接 NIC MAC 地址，在一行中至少连续重复 16 次。

注 2：AMD 引入 WOL 时称之为"魔术包技术（Magic Packet Technology）"。

现在也有可能允许其他类型的帧唤醒系统。有少量设备可以根据在模块加载期间设置的一个参数开启或关闭WOL功能（参见*drivers/net/3c59x.c*为例）。*ethtool*工具允许管理员配置哪几种帧可以唤醒系统，其中一种选择是ARP包，如第二十八章的“网络唤醒事件”一节所述。*net-utils*套件中有一个命令*ether-wake*，可用于产生WOL Ethernet帧。

只要一个开启WOL功能的设备识别一个帧，其类型为可唤醒系统，就会产生一个电源管理通知信息，去做相应的工作。

有关电源管理的更多细节，可参考第八章的“与电源管理之间的交互”一节。

PCI NIC驱动程序注册范例

我们用*drivers/net/e100.c*中的Intel PRO/100 Ethernet驱动程序来说明驱动程序的注册：

```
#define INTEL_8255X_ETHERNET_DEVICE(device_id, ich) {\
        PCI_VENDOR_ID_INTEL, device_id, PCI_ANY_ID, PCI_ANY_ID, \
        PCI_CLASS_NETWORK_ETHERNET << 8, 0xFFFF00, ich }
static struct pci_device_id e100_id_table[] = {
        INTEL_8255X_ETHERNET_DEVICE(0x1029, 0),
        INTEL_8255X_ETHERNET_DEVICE(0x1030, 0),
        ...
}
```

由“PCI NIC设备驱动程序的注册”一节可知，PCI NIC设备驱动程序会把一个pci_device_id结构向量注册给内核，该结构向量列出了其所能处理的那些设备。例如，e100_id_table就是*e100.c*驱动程序使用的结构。注意：

- 第一个字段（相当于此结构定义中的vendor）为固定值PCI_VENDOR_ID_INTEL，它为指定给Intel初始化的开发商ID（注3）。
- 第三字段和第四字段（subvendor和subdevice）通常初始化为通用值PCI_ANY_ID，因为前两个字段（vendor和device）足以识别那些设备。
- 许多设备都会对那张设备表使用宏__devinitdata，用来标记初始化数据，不过e100_id_table没有。第七章将说明这个宏有什么作用。

此模块由module_init宏指定的e100_init_module初始化（注4）。当此函数在引导期间或者模块加载期间由内核执行时，就会调用“PCI NIC设备驱动程序的注册”一节

注3：更新的列表可在*http://pciids.sourceforge.net*中找到。

注4：有关模块初始化代码的更多细节，参见第七章。

所介绍的函数pci_module_init。此函数会注册该驱动程序，并间接注册所有相关的NIC，如稍后“大蓝图”一节所做的简述。

下列片段是e100驱动程序与PCI层接口相关的关键部分：

```
NAME "e100"

static int _ _devinit e100_probe(struct pci_dev *pdev,
     const struct pci_device_id *ent)
{
     ...
}
static void _ _devexit e100_remove(struct pci_dev *pdev)
{
     ...
}

#ifdef CONFIG_PM
static int e100_suspend(struct pci_dev *pdev, u32 state)
{
     ...
}
static int e100_resume(struct pci_dev *pdev)
{
     ...
}
#endif

static struct pci_driver e100_driver = {
     .name =          DRV_NAME,
     .id_table =      e100_id_table,
     .probe =         e100_probe,
     .remove =        _ _devexit_p(e100_remove),
#ifdef CONFIG_PM
     .suspend =       e100_suspend,
     .resume =        e100_resume,
#endif
};

static int _ _init e100_init_module(void)
{
     ...
     return pci_module_init(&e100_driver);
}

static void _ _exit e100_cleanup_module(void)
{
     pci_unregister_driver(&e100_driver);
}

module_init(e100_init_module);
module_exit(e100_cleanup_module);
```

此外，注意：

- 只有当内核支持电源管理时，才会对suspend和resume做初始化，所以只有当该条件为真时，e100_suspend和e100_resume这两个函数才会包括在映像中。
- pci_driver的remove字段用宏__devexit_p标记，而e100_remove用__devexit标记。
- e100_probe用__devinit标记。

在第七章将说明上述列表所提到的__devXXX宏的作用。

大蓝图

我们把前几节所了解的综合来看，采用PCI总线的系统以及一些PCI设备在引导期间会发生什么事（注5）。

当系统引导时，会建立一种数据库，把每个总线都关联一份已侦测到而使用该总线的设备列表。例如，PCI总线的描述符除了其他参数外，还包括一个已侦测PCI设备的列表。如“PCI NIC设备驱动程序的注册”一节所见，每个PCI设备都可由pci_device_id结构中的一个大的字段集合唯一地识别，不过通常只需要几个字段就足够了。我们也知道PCI设备驱动程序如何定义一个pci_driver实例，以及如何用pci_register_ driver（或者其别名pci_module_init）与PCI层注册。设备驱动程序加载时，内核已建好其数据库（注6）：我们以配有三个PCI设备的图6-1（a）为例，说明当设备驱动程序A和B加载时会发生什么事。

当设备驱动程序A被加载时，会调用pci_register_driver并提供其pci_driver实例而与PCI层注册。pci_driver结构有内含一个此驱动程序能驱动的PCI设备ID的向量。接着，PCI层使用该表去查看在已侦测的PCI设备列表中与哪些设备匹配。于是，就会建立该驱动程序的设备列表，如图6-1（b）所示。此外，对每个匹配的设备而言，PCI层会调用相匹配的驱动程序中的pci_driver结构中所提供的probe函数。probe函数会建立并注册相关联的网络设备。就此而言，设备*Dev3*需要另一个名为B的设备驱动程序。当驱动程序B终于与内核注册后，*Dev3*就会被分派给驱动程序B。图6-1（c）所示就是加载此驱动程序后的结果。

当驱动程序于稍后卸载时，该模块的module_exit函数就会调用pci_unregister_driver。接着，由于其数据库，使得PCI层能够遍历所有与该驱动程序相关联的设备，并启用该驱动程序的remove函数。此函数就会将此网络设备除名。

注5：其他总线的行为也很类似。更多信息参见《Linux设备驱动程序》一书。

注6：并非所有总线类型都可以如此。

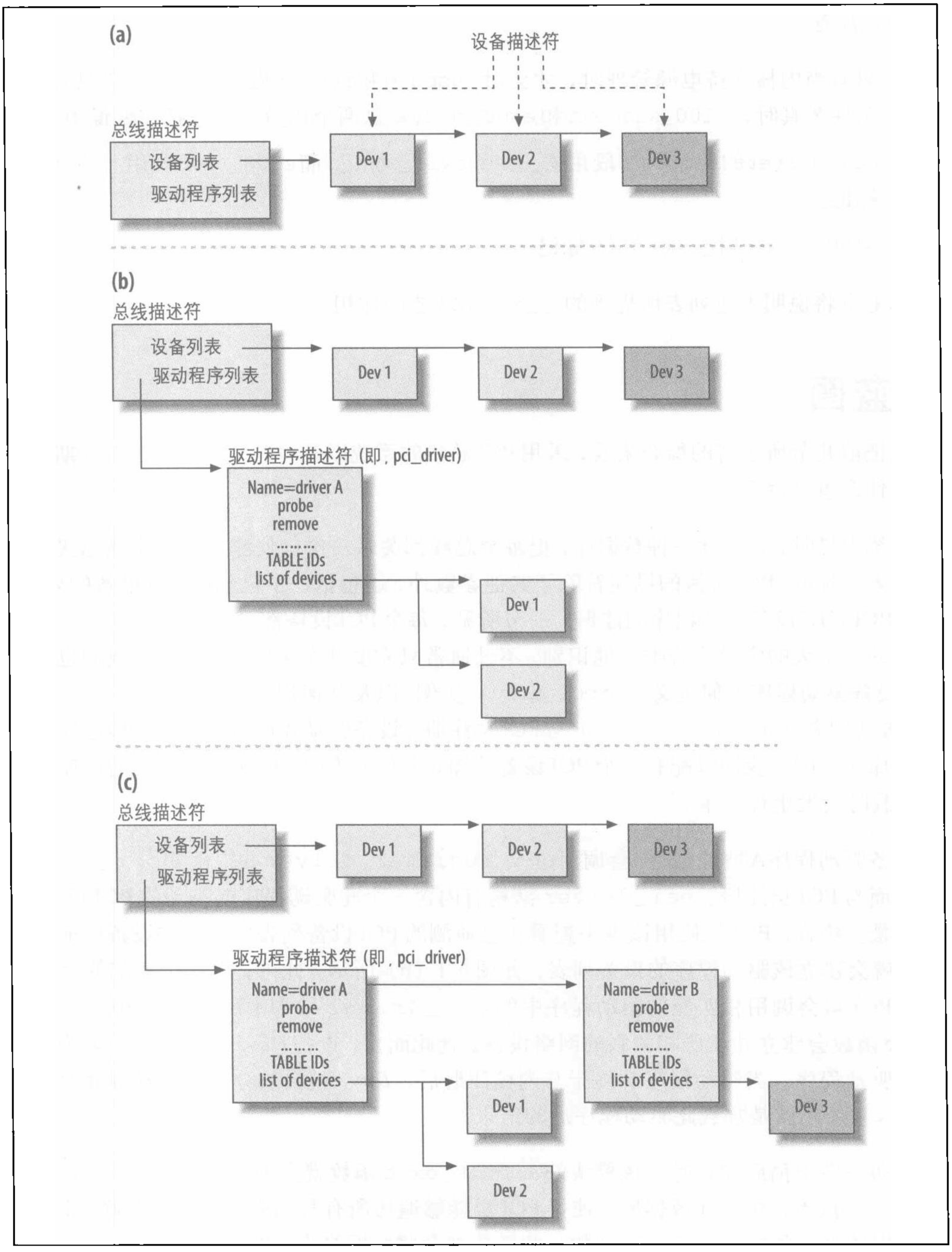


图 6-1：总线与驱动程序以及驱动程序与设备之间的绑定

第八章可以找到关于 probe 和 remove 函数的更多内部细节。

通过 /proc 文件系统调整

/proc/pci 文件可用于倾卸有关已注册的 PCI 设备的信息。*pciutils* 套件中的 *lspci* 命令也可用于打印出有关本地 PCI 设备的有用信息，但其信息取自 */sys*。

本章涉及的函数和变量

表 6-1 总结了本章介绍的函数、宏以及数据结构。

表 6-1：与 PCI 设备处理相关的函数、宏以及数据结构

名称	描述
函数和宏	
pci_register_driver	注册、除名以及初始化 PCI 驱动程序
pci_unregister_driver	
pci_module_init	
数据结构	
struct pci_driver	第一个数据结构定义 PCI 驱动程序（多数都是虚拟回调函数）。
struct pci_device_id	第二个数据结构存储 PCI 设备相关联的通用 ID。最后一个数据
struct pci_dev	结构代表内核空间中的 PCI 设备

本章涉及的文件和目录

图 6-2 列出了本章所涉及的文件和目录。这张图并没有包含本章涉及的主题所使用的所有文件。例如，*drivers/pci/* 目录下就包括好几个其他文件。

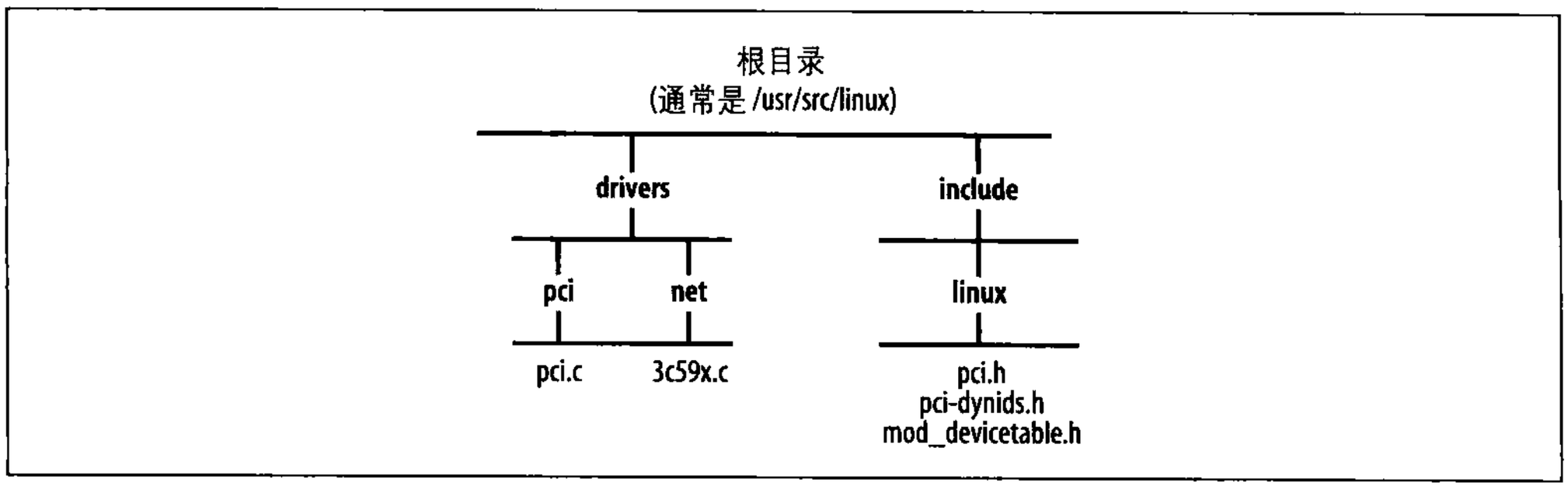


图 6-2：本章涉及的文件和目录

第七章

组件初始化的内核基础架构

要完全了解内核组件，不但要知道给定的一组函数在做什么，还要知道这些函数何时被启用以及由谁启用。子系统的初始化就是内核根据其模型所处理的基本任务之一。这种基础架构值得研究，有助于了解网络协议栈的关键组件是如何初始化的（包括NIC设备驱动程序在内）。

本章的目的在于说明内核如何处理用于对内核组件进行初始化工作的函数，这些组件包括与内核静态连接的组件以及那些以内核模块形式加载的组件，但强调的重点是网络设备。因此，我们会谈到：

- 初始化函数如何命名，以及如何识别特殊的宏。
- 这些宏如何根据内核配置来定义，使得内存的使用最优化，并确保各种初始化工作能按正确次序进行。
- 这些函数何时执行以及如何执行。

我们将不讨论初始化基础架构的所有细节，但是，你会得到足够的概论，足以轻松浏览源码。

引导期间的内核选项

Linux允许用户把内核配置选项传给引导记录，然后引导记录再把选项传给内核。有经验的用户可借此机制在引导期间微调内核（注1）。在引导阶段，如第五章的图5-1所示，对parse_args调用两次，负责引导期间配置输入数据。下一节会说明为什么要调用两次parse_args，而其细节则于稍后的“两遍分析”一节说明。

注1：可以在《Linux BootPrompt HOWTO》一书中找到引导选项的一些说明文件以及一些使用范例。

parse_args是一个函数，用于解析输入字符串，而输入的字符串内是一些参数，其形式为变量名称=值，寻找特定关键字，并启用适当的处理函数。加载模块时，也会用到parse_args，借以解析命令列参数(如果有的话)。

我们不需要知道parse_args实现解析行为的细节，但是了解内核组件如何为关键字注册处理函数以及处理函数如何启用相当有趣。为了有清晰的概念，我们需要了解：

- 内核组件如何注册关键字，而当引导字符串中出现该关键字时，相关联的处理函数会被执行。
- 内核如何解析关键字与处理函数之间的关联性。我们将高度概括内核如何解析输入字符串。
- 网络设备子系统如何使用这项功能。

所有的解析代码都在*kernel/params.c*中。我们逐一说明上述列表的要点。

注册关键字

内核组件可以利用定义在*include/linux/init.h*中的__setup宏，注册关键字和相关联的处理函数。以下是其语法：

```
__setup(string, function_handler)
```

string是关键字，而function_handler是相关联的处理函数。此例只显示了指示内核当输入的引导期间字符串中包括string时，就执行function_handler。string必须以=字符作结束，以使parse_args的解析能轻松一点。任何跟在=之后的文本都将作为输入值传给function_handler。

下面是取自*net/core/dev.c*的例子，其中netdev_boot_setup注册成了netdev=关键字的处理函数：

```
__setup("netdev=", netdev_boot_setup);
```

同一个处理函数可以与几个不同关键字相关联。例如，*net/ethernet/eth.c*为ether=关键字注册了同一个处理函数netdev_boot_setup。

当一段代码编译成模块时，__setup宏会被忽略（即定义为空操作）。可以查看一下*include/linux/init.h*中的__setup宏定义，如何根据头文件init.h的代码是否为模块而变动。

start_kernel两次调用parse_args以解析引导配置字符串的原因在于，引导期间选项实际上是分成两类，而每一次调用都是针对其中一类：

默认选项

多数选项都属于这一类。这些选项都是用__setup宏定义的，而且是由parse_args第二次被调用时处理。

初期选项

内核引导期间，有些选项必须比其他选项更早处理。内核提供了early_param宏以声明这些选项替代__setup。然后，这些选项会由parse_early_params负责。early_param和__setup的唯一差别是early_param会设置一个特殊标识，使内核能区分这两种情况。此标识是obs_kernel_param数据结构的一部分，在“.init.setup内存节区”一节将对此进行说明。

到了内核2.6版本，引导期间选项的处理有了改变，但是并非所有内核代码都据此做了更新。最新的改变之前，仅仅是用__setup宏。因此，等着更新的旧代码现在都使用_obsolete_setup宏。当用户把__obsolete_setup宏所声明的选项传给内核时，内核会输出警告信息，指出其废弃状态，并且提供声明该宏的文件和源代码的所在处。

图7-1总结了各种宏之间的关系：这些都是包裹函数，内含通用的函数__setup_param。

注意，传给__setup的输入函数会被放到.init.setup内存节区中。在“引导期间初始化函数”一节，这种行为的效果就会看得很清楚了。

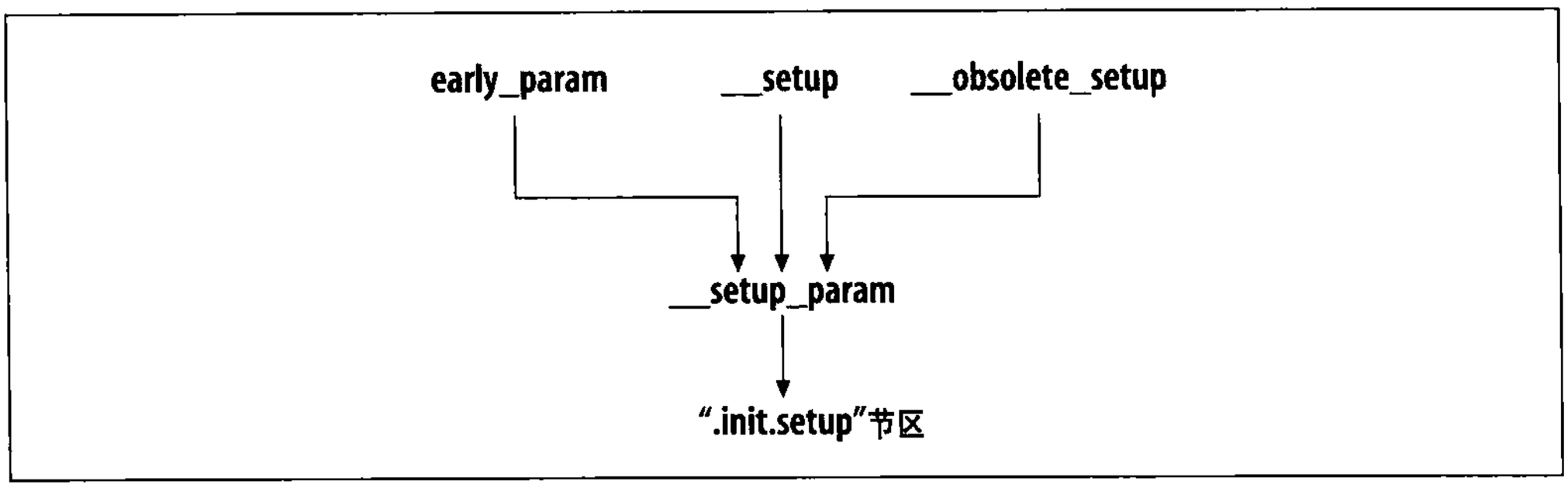


图7-1：setup_param宏及其包裹函数

两遍分析

因为早先的几种版本的内核都以不同的方式处理引导期间选项，而且并非所有选项都已转换为新的模型，所以内核需要处理两种模型。当新型基础架构无法识别某个关键字时，就会使用旧型基础架构予以处理。如果旧的基础架构也不能识别，则该关键字和其值就会传给init进程，init内核线程结束时会通过run_init_process予以启用（如

第五章图5-1所示)。该关键字及其值不是添加到arg参数列表，就是添加到envp环境变量列表。

前一节提到，为了使初期选项以必要次序处理，引导字符串的解析以及处理函数的调用是分成两遍进行的，如图7-2所示（图中是第五章介绍过的start_kernel的范例)：

1. 第一遍只看必须在初期处理的较高优先级的选项，由一个特殊的标识(early)识别。
2. 第二遍则会负责所有其他选项。多数选项都属于这一类。遵循旧模型的所有选项都会在这一遍处理。

第二遍会先检查那些选项中是否匹配根据新基础架构实现的选项。这些选项会存储在kernel_param数据结构内，由第五章“模块选项”一节所介绍的module_param宏填入。module_param宏也会确保所有这些数据结构都会放进特定的内存节区(__param)，由指针__start___param和__stop___param限定。

识别其中一个选项时，相关联的参数被引导字符串所提供的值初始化。如果没有匹配这个选项，unknown_bootoption就会试图了解该选项是否应该由旧的处理函数处理(见图7-2)。

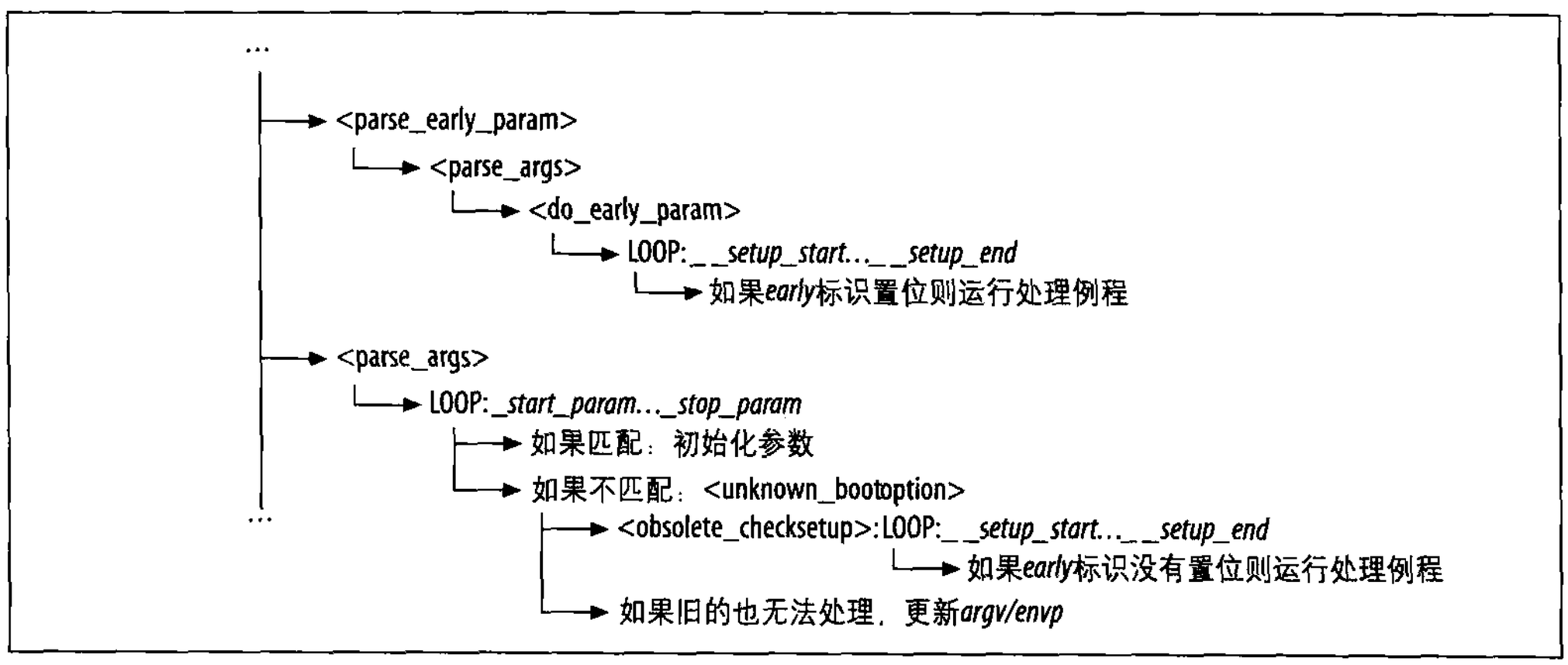


图7-2：两遍选项解析

旧的和新的选项会放在两个不同的内存区域：

__setup_start ··· __setup_end

稍后一节就会知道，这个区域会在引导阶段结束时释放掉：一旦内核引导了，这些选项就不再需要了。用户无法在运行期间看到或修改这些选项。

__start___param ··· __stop___param

这个区域不会被释放。其内容会输出到*/sys*，使得这些选项可以展露给用户。

有关模块参数的更多细节请参考第五章。

此外还应注意，所有旧的选项，无论其是否设置了early标识，都会放入__setup_start … __setup_end内存区域内。

.init.setup 内存节区

前一节介绍的__setup宏的两个输入数据会放入一个定义在*include/linux/init.h*中的obs_kernel_param类型的数据结构内。

```
struct obs_kernel_param {
    const char *str;
    int (*setup_func)(char*);
    int early;
};
```

str是关键字，setup_func是处理例程，而early标识我们在“两遍分析”一节中已经介绍过。

__setup_param宏会把所有obs_kernel_params实例放入一个专用的内存区域内。这样做主要有两个原因：

- 比较容易遍历所有实例，例如，在根据str关键字进行查询时。在进行关键字查询时，我们就会知道内核如何使用这两个指针__setup_start和__setup_end，分别指向刚才所提到的区域的开端和尾端（如稍后的图7-3所示）。
- 当这些数据结构不再有用时，内核可以快速释放所有数据结构。在“内存优化”一节会返回来讨论这一点。

使用引导选项配置网络设备

根据前几节所说的，我们来说明网络代码如何使用引导选项。

“注册关键字”一节已经提到过，ether=和netdev=关键字都注册为使用同一个处理函数netdev_boot_setup。当此处理函数被启用处理输入参数（也就是匹配关键词的字符串）时，就会把结果存储至定义在*include/linux/netdevice.h*中类型为netdev_boot_setup的数据结构内。处理函数和数据结构类型恰巧共享同一名称，因此要注意别混淆两者。

```
struct netdev_boot_setup {
```

```
    char name[IFNAMSIZ];
    struct ifmap map;
};
```

name是设备名，而定义在 *include/linux/if.h* 中的 ifmap是存储输入配置值的数据结构：

```
struct ifmap
{
    unsigned long mem_start;
    unsigned long mem_end;
    unsigned short base_addr;
    unsigned char irq;
    unsigned char dma;
    unsigned char port;
    /* 剩3个字节 */
};
```

同一个关键字可以在引导字符串中出现多次（针对不同设备），如下例所示：

LILO: linux ether=5,0x260,eth0 ether=15,0x300,eth1

然而，能以此种机制在引导期间配置设备的最大数目是NETDEV_BOOT_SETUP_MAX，也就是用于存储配置值的静态数组 dev_boot_setup 的大小：

```
static struct netdev_boot_setup dev_boot_setup[NETDEV_BOOT_SETUP_MAX];
```

netdev_boot_setup相当简单：就是从字符串中抽取输入参数，填入一个ifmap结构，然后用 netdev_boot_setup_add把ifmap结构添加到 dev_boot_setup 数组中。

引导阶段结束时，网络代码可以使用netdev_boot_setup_check函数检查给定接口是否与引导期间的配置相关联。对数组dev_boot_setup的查询基于设备名称dev->name：

```
int netdev_boot_setup_check(struct net_device *dev)
{
    struct netdev_boot_setup *s = dev_boot_setup;
    int i;

    for (i = 0; i < NETDEV_BOOT_SETUP_MAX; i++) {
        if (s[i].name[0] != '\0' && s[i].name[0] != ' ' &&
            !strncmp(dev->name, s[i].name, strlen(s[i].name))) {
            dev->irq        = s[i].map.irq;
            dev->base_addr  = s[i].map.base_addr;
            dev->mem_start  = s[i].map.mem_start;
            dev->mem_end    = s[i].map.mem_end;
            return 1;
        }
    }
    return 0;
}
```

有特殊能力、功能或限制的设备，如果除了需要ether=和netdev=这些基本关键字外还需要其他参数，也可以自行定义其自用的关键字和处理函数（PLIP就是其中一个这么做的驱动程序）。

模块初始化代码

接下来几节的范例都与模块有关，因此一些基本概念必须搞清楚才行。

内核代码可以和主要映像做静态链接，也可以在需要时以模块形式动态加载。并非所有内核组件都适合编译成模块。驱动程序以及基本功能的扩充这类内核组件，就是通常都编译成模块的典型范例。你可以参考《Linux设备驱动程序》一书，书中详细描述了模块的优缺点，以及内核在需要时将其动态加载以及不需要时将其予以卸载的机制。

每个模块都必须提供两个特殊函数，称为init_module和cleanup_module。第一个函数会在模块加载时被调用，以初始化该模块。删除模块时，内核会调用第二个函数，以释放由模块由于其用途所分配的任何资源（包括内存在内）。

内核提供两个宏module_init和module_exit，允许开发人员为这两个函数任意命名。下面的程序范例取自 *drivers/net/3c59x.c* Ethernet驱动程序：

```
module_init(vortex_init);
module_exit(vortex_cleanup);
```

在“内存最优化”一节我们将会看到这两个宏是如何定义的，以及它们的定义如何根据内核配置而改变。内核的大多数程序都会使用这两个宏，但是有些模块依然使用旧的默认名称init_module和cleanup_module。本章接下来都将使用module_init和module_exit来初始化和清除函数。

首先来看旧版本内核模块初始化代码是如何编写的，然后再了解基于一组新的宏的当前内核模型的工作方式。

旧模型：条件式代码

无论内核组件是编译成模块还是和内核静态链接，都必须做初始化工作。因此，内核组件的初始化代码必须通过条件式指令使编译器能区分这两种情况。对旧模型而言，这迫使开发人员到处使用类似#ifdef的条件式指令。

以下是取自内核2.2.14版本 *drivers/net/3c59x.c* 驱动程序的范例：注意#ifdef MODULE和#if defined (MODULE)用了多少次。

```
...
#if defined(MODULE) && LINUX_VERSION_CODE > 0x20115
MODULE_AUTHOR("Donald Becker <becker@cesdis.gsfc.nasa.gov>");
MODULE_DESCRIPTION("3Com 3c590/3c900 series Vortex/Boomerang driver");
MODULE_PARM(debug, "i");
...
#endif
...
#ifdef MODULE
...
int init_module(void)
{
    ...
}
#else
int tc59x_probe(struct device *dev)
{
    ...
}
#endif  /* 不是MODULE */
...
static int vortex_scan(struct device *dev, struct pci_id_info pci_tbl[])
{
   ...
#if defined(CONFIG_PCI) || (defined(MODULE) && !defined(NO_PCI))
   ...
#ifdef MODULE
    if (compaq_ioaddr) {
        vortex_probe1(0, 0, dev, compaq_ioaddr, compaq_irq,
                compaq_device_id, cards_found++);
        dev = 0;
    }
#endif

    return cards_found ? 0 : -ENODEV;
}
...
#ifdef MODULE
void cleanup_module(void)
{
    ... ... ...
}
#endif
```

以上范例显示了在旧模型下程序员如何根据代码是否编译成模块或静态链接至内核映像，而让一些指定的事情有不同的结果：

初始化代码的执行行为不同

上述范例表明，只有当驱动程序编译成模块时，才会定义cleanup_module函数（因此也才会使用此函数）。

一些代码片段可以引入模块或从模块排除

例如，只有当驱动程序被编译成模块时，vortex_scan才会调用vortex_probe1。

这种模型使得源代码难以领会，因此也难以调试。此外，每个模块都会重复相同的逻辑。

新模型：宏卷标

现在，我们拿2.6版本内核中相同文件的同一部分和前一节的范例做比较：

```
static char version[] _ _devinitdata = DRV_NAME " ... ";

static struct vortex_chip_info {
    ...
} vortex_info_tbl[] _ _devinitdata = {
    {"3c590 Vortex 10Mbps",
   ... ... ...
}

static int _ _init vortex_init (void)
{
    ...
}
static void _ _exit vortex_cleanup (void)
{
    ...
}

module_init(vortex_init);
module_exit(vortex_cleanup);
```

可以看到，#ifdef指令已不再必需了。

为了消除条件式代码引起的混乱，使程序具有更高的可读性，内核开发人员引入了一组宏，现在模块开发人员可以使用这组宏编写更为清晰的初始化代码（大多数驱动程序都是使用这些宏的绝佳对象）。在这个范例中只用到其中一些宏而已：__init、__exit以及__devinitdata。

稍后几节将描述一些新宏的用法和工作方式。

这些宏允许内核在后台决定每个模块要把哪些代码引入到内核映像中，由于不需要哪些代码排除在外，以及哪些代码只在初始化期间执行等等。这使得每个程序员的负担减轻，不用在每个模块内重复相同的逻辑（注2）。

注2：注意，使用这些宏并无法让条件式指令的使用完全消失。内核依然使用条件式指令安排一些选项，当编译内核时由用户配置。

显然，这些宏允许程序员替代前一节范例所示的旧式条件式指令。这些宏必须提供至少下列两项服务：

- 定义新内核组件开启时必须执行的函数，无论这些组件是因为与内核静态连接，或者是因为于执行期间以模块形式而加载。
- 定义一些初始化函数之间的次序，强制内核组件之间遵循相互依赖性。

优化宏卷标

Linux 内核使用各种不同的宏为函数和数据结构标记特殊属性：例如，标记一个初始化函数。这些宏大多数都定义在 *include/linux/init.h* 中，而有些宏会通知链接器把带有共同属性的代码或数据结构放到特定专用的内存区域（内存节区）。如此一来，内核就会用简单的方式轻易管好全部具有共同属性的对象类（函数或数据结构）。在“内存最优化”一节将看到一个范例。

图 7-3 所示为一些内核内存节区：左侧是指针名称，限定每个区域节区（有意义时）的开端和尾端。

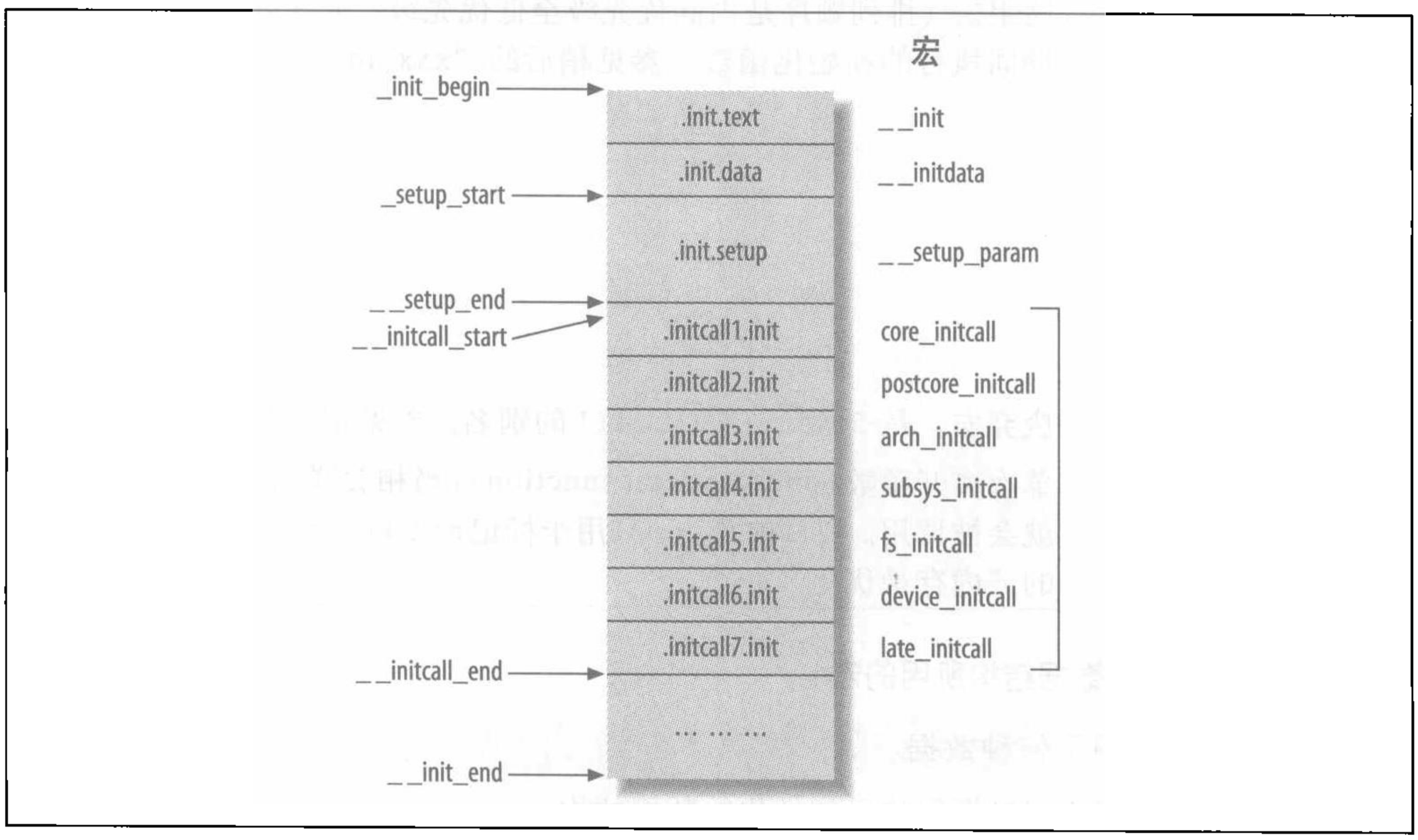


图 7-3：初始化代码所用的一些内存节区

右侧是宏的名字，用于把数据和代码放入相关的节区内。由于实在太多，这张图不便引入所有与内存节区相关的宏。

表7-1和表7-2列出了一些用于标记函数和数据结构的宏并附有简述。因为篇幅有限，我们不会全部讨论，但是我们会在“xxx_initcall宏”一节花点篇幅说明xxx_initcall宏，然后在“__init和__exit宏”一节说明__init和__exit。

本节的目的不是描述内核映像是如何建立的或者模块是如何处理的，而是给一些提示，让你知道为什么这些宏会存在，以及设备驱动程序最常用的几个宏是如何工作的。

表7-1：函数所用的宏

宏	宏用于何种函数
__init	引导期间初始化函数：针对那些到了引导阶段结束时已不需要使用的函数。 这项信息也可用于在某些条件下抛弃函数(参见稍后的“内存最优化”一节)
__exit	__init的配对宏。当相关的内核组件关闭时，就会被调用。通常是用于标记module_exit函数。 这项信息也可用于在某些条件下抛弃函数(参见稍后的“内存最优化”一节)
core_initcall postcore_initcall arch_initcall subsys_initcall fs_initcall device_initcall late_initcall	这组宏（排列顺序是由高优先级至低优先级）可标记那些必须在引导期间执行的初始化函数。参见稍后的“xxx_initcall宏”一节
__initcall	废弃宏，是device_initcall的别名。参见稍后的“旧程序”一节
__exitcall	单次离开函数(one-shot exit function)；当相关联的内核组件关闭时，就会被调用。到目前为止，只用于标记module_exit函数。参见稍后的“内存最优化”一节

表7-2：已初始化的数据结构所用的宏

宏	宏用于何种数据
__initdata	只用于引导期间的已初始化的数据结构
__exitdata	只用于标记__exitcall的函数所用的数据结构。也就是说，如果标记__exitcall的函数即将不再使用，则标记为__exitdata的数据也不再使用。因此，可以对__exitdata和__exitcall应用相同的优化手段

在我们深入说明表 7-1 和表 7-2 的一些宏之前，要先强调下列几点：

- 大多数宏都是成对出现的：一个（或一组）负责初始化，而另一个姊妹宏（或一组姊妹宏）就负责删除。例如，__exit 是__init 的姊妹宏，而 __exitcalls 是__initcall的姊妹宏，等等。
- 宏用于两种情况（非此即彼）：一种是当函数要被执行时（也就是 __initcall、__exitcall）；另一种是函数或数据结构要放入内存节区（也就是 __init、__exit）。
- 同一个函数可以标记一个以上的宏。例如，下列片段指出 pci_proc_init 必须在引导期间执行（__initcall），而且执行之后就可以释放了（__init）：

```
static int __init pci_proc_init(void)
{
...
}

__initcall(pci_proc_init)
```

设备初始化函数使用的初始化宏

表 7-3 所列的一组宏，通常用于初始化其设备的设备驱动程序标记函数，而且当内核不支持热插拔时，还能进行内存最优化工作。第六章“PCI NIC 驱动程序注册范例”一节可以找到使用这些宏的范例。稍后“其他的最优化”一节，可知表 7-3 的宏何时可促进内存的最优化。

表 7-3：设备初始化函数所用的宏

名称	描述
__devinit	用于标记初始化设备的函数。 例如，对PCI驱动程序而言，指定给pci_driver->probe的函数标记这个宏。 被另一个标记为__devinit的函数进行互斥启用的函数通常也标记为__devinit
__devexit	标记当设备删除时要被调用的函数。
__devexit_p	用于初始化标记为 __devexit 函数的指针。 如果内核支持模块和热插拔，__devexit_p(fn)会返回 fn，否则返回 NULL。参见稍后的“其他的最优化”一节
__devinitdata	标记负责设备初始化的函数（也就是标记 _devinit）使用的已初始化的数据结构，因此，彼此间属性共享
__devexitdata	同 __devinitdata，不过是与 __devexit 相关联

引导期间初始化函数

大多数初始化函数都有两个有趣的属性：

- 当所有内核组件都已初始化之时，必须在引导期间执行。
- 一旦被执行后，就不再需要这些函数了。

下一节“xxx_initcall宏”将描述引导期间运行初始化函数所用的机制，把这些属性以及模块之间的优先级考虑进来。稍后的“内存最优化”一节会说明如何使用巧妙的标记，在链接期间或运行期间使不再需要的函数和数据结构可以被释放掉。

xxx_initcall 宏

内核引导的早期阶段有下列两项主要的初始化工作：

- 各种必须以特定次序完成初始化的关键子系统和强制子系统。例如，PCI层初始化之前，内核无法对PCI设备驱动程序做初始化。参见稍后的“初始化函数之间的依赖性范例”一节的范例。
- 其他不需要以严格次序完成初始化的内核组件：一些具有相同优先级的函数可以按任何次序执行。

第一部分由第五章图5-1中do_initcalls之前的代码负责。而第二部分是调用该图中靠近do_basic_setup尾端附近的do_initcalls完成。第二部分的初始化函数根据其角色和优先级进行分类。内核会逐个执行这些初始化函数，由最高优先级类的那些函数（core_initcall）开始执行。这些函数调用时必须使用的地址都放在图7-3所示的.initcallN.init内存节区中，它们由表7-1的那些xxx_initcall宏中的其中之一来标记。

用于存储标记xxx_initcall宏函数地址的区域，由起始地址（__initcall_start）和结尾地址（__initcall_end）限定。在下列摘录自do_initcalls的程序片段中可以发现，就是把函数地址一个接一个从该区域中取出，然后执行其所指的函数：

```
static void _ _init do_initcalls(void)
{
        initcall_t *call;
        int count = preempt_count();

        for (call = __initcall_start; call < __initcall_end; call++) {
            ... ... ...
            (*call)();
            ... ... ...
        }
```

```
            flush_scheduled_work();
    }
```

由do_initcalls所调用的函数不应该改变抢占状态或关闭IRQ。因此，每个函数执行之后，do_initcalls就会检查该函数是否做了任何修改，然后必要时调整抢占和IRQ状态（前例中没有显示出来）。

xxx_initcall函数有可能会让一些工作进入调度，使其稍后发生。也就是说，由这些函数所处理的任务会以异步方式在未知时刻终止。调用flush_scheduled_work就是为了使do_initcalls在返回前等待异步任务完成。

注意，do_initcalls本身就以__init标记：因为此函数只会在引导阶段在do_basic_setup内使用一次，一旦do_basic_setup完成后，内核就可以丢弃do_initcalls。

__exitcall是__initcall的配对宏。这个宏很少直接使用，而是通过定义为其别名的其他宏，如module_exit，对此我们在“模块初始化代码”一节中已有介绍。

__initcall和__exitcall函数范例：模块

我在“模块初始化代码”一节说过，module_init和module_exit宏分别标记那些在模块初始化以及卸载时执行的函数(如果内建在内核就是引导期间，如果是个别加载就是执行期间)。

这使得模块成为__initcall和__exitcall宏的最佳对象：按我刚才所说的，下列来自于*include/linux/init.h* 中的module_init和module_exit宏的定义应该也不会令人诧异了：

```
#ifndef MODULE
... ... ...
#define module_init(x)     _ _initcall(x);
#define module_exit(x)     _ _exitcall(x);
#else
... ... ...
#endif
```

module_init针对与内核静态链接的代码，被定义为__initcall的别名：其输入函数被归类为引导期间初始化函数。

module_exit也遵循相同规则：当代码内建于内核时，module_exit就变成关闭函数。目前，当系统关机时，关闭函数并不会被调用，但代码已存在，想做就可以做（注3）。

注3：User-Mode Linux是唯一实际利用关闭函数的架构。User-Mode Linux并未使用__exitcall宏，而是定义其自用的宏__uml_exitcall。User-Mode Linux项目的首页为*http://user-mode-linux.sourceforge.net*。

初始化函数之间的依赖性范例

第五章曾介绍了net_dev_init。设备驱动程序以其module_init函数与内核注册，如第六章“大蓝图”一节所述，会将其设备注册给网络代码。那些内建驱动程序的net_dev_init和各种module_init函数会在引导期间由do_initcalls调用。因此，内核必须确保net_dev_init执行前，不会发生任何设备注册的情事。由于设备驱动程序函数以宏device_initcall（或其别名__initcall）标记，而net_dev_init以subsys_initcall标记，使得这样的确保工作得以透明化实施。在图7-3中，你可以发现subsys_initcall函数的执行早于device_initcall函数（内存节区按优先级次序排序）。

旧代码

引进xxx_initcall这组宏之前，只有一个宏可标记初始化函数：__initcall。只用一个宏引发了很严重的局限性：只通过宏标记函数，无法强制执行次序。在很多情况下，由于模块之间的依赖性以及其他因素，使得这种限制无法接受。因此，__initcall的使用无法拓展到所有初始化函数。

设备驱动程序过去时常用到__initcall。为了与那些尚未更新为新模型的代码兼容，此宏依然存在，但只定义为device_initcall的别名。

当前模型依然存在的另一项限制，就是无法向初始化函数提供参数。然而，这一限制看起来并不重要。

内存最优化

与用户空间代码及数据不同的是，内核代码及数据会永久驻留在主存中，所以，采用各种可能的方式减少内存浪费是很重要的。初始化代码就是内存最优化的理想对象。因其固有的特性，大多数初始化函数只会执行一次，或者根本不会执行，这取决于内核的配置。例如：

- 当相关联的模块加载时，module_init函数只会执行一次。当该模块与内核静态链接时，在引导期间，module_init函数执行后内核就可将其释放掉。
- 当相关联的模块与内核静态链接时，module_exit函数根本不会执行。因此，就此而言，没必要把module_exit函数引入内核映像中（也就是这类函数可以在链接期间就被丢弃掉）。

第一种情况是执行期间的最优化，而第二种是链接期间的最优化。代码及数据只在引导

期间用一次，而后就不再需要了，从那时以后就被放在图 7-3 所示的内存节区之一。一旦内核完成初始化阶段，就可丢弃整个内存区域，做法是调用 free_init_mem（注 4），如第五章图 5-1 所示。不同的宏可用于把代码放入图 7-3 所示的不同内存节区内。

如果你看一下稍早一节“新型：宏卷标”的范例，会发现 module_init 和 module_exit 两个输入函数（通常）分别标记为 __init 和 __exit：这样做正好利用了本节开头所提到的两个属性。

__init 和 __exit 宏

在内核早期阶段所执行的初始化函数都会标记为 __init 宏。

如前节所述，大多数 module_init 输入函数都会标记为此宏。例如，第五章图 5-1 中的大多数函数（调用 free_initmem 之前）都用 __init 标记。

如其定义所示，__init 宏把输入函数放到 .text.init 内存节区：

```
#define __init      __attribute__ ((__section__ (".text.init")))
```

这个节区是 free_initmem 在运行期间释放的内存节区之一。

__exit 是 __init 的配对宏。用于关闭模块的函数都放在 .text.exit 节区中。对内建于内核的模块而言，这个节区可以在链接期间直接丢弃掉。然而，有些体系结构会在运行期间予以丢弃，以应付交叉引用。注意，对个别加载的模块而言，当内核不再支持模块卸载时，同一个节区也可以在加载时予以删除（有一个内核选项可让用户无法卸载模块）。

xxx_initcall 和 __exitcall 节区

内核用于放置指向 xxx_initcall 和 __exitcall 宏所标记函数的地址存储区也将被丢弃掉：

- 图 7-3 所示的 xxx_initcall 节区在运行期间被 free_initmem 丢弃掉。
- 给 __exitcall 函数使用的 .text.exit 节区则会在链接期间被丢弃掉，因为在系统关机时内核不会立即调用 __exitcall 函数（也就是没有使用类似 do_initcalls 的机制）。

注 4：　这种内存就是这类引导信息所指的内存：“Freeing unused kernel memory: 120k freed”。

其他的最优化

其他最优化范例所包括的宏见表7-3所列。

__devinit

当内核没有编译为支持热插拔时，在引导阶段结束后就不再需要标记__devinit的函数（在所有设备都已初始化之后）。因此，在不支持热插拔时，__devinit就变成__init的别名。

__devexit

当PCI驱动程序内建于不支持热插拔的内核时，标记为__devexit初始化pci_driver->remove的函数因为不再需要就可以被丢弃掉。当模块个别加载于不支持模块卸载的内核时，该函数也可以丢弃掉。

__devinitdata

在不支持热插拔时，这种数据也只是在引导期间需要而已。正常来讲，设备驱动程序会以此宏标记标题字符串，使得pci_driver->probe函数在初始化设备时得以打印。例如，PCI驱动程序会用__devinitdata标记pci_device_id表：一旦系统完成引导，而且不支持热插拔，内核就不再需要那些表格了。

这一节只给出一些删除程序的范例而已。浏览源代码可以学到更多，例如，可以从/DISCARD/节区的体系结构依赖定义着手。

动态宏的定义

前几节中，我介绍了一些宏，如__init和各种xxx_initcall版本。我们也知道传给module_init宏的函数都用诸如__initcall之类的宏标记。因为大多数内核组件可以编译成模块或者与内核静态链接，而选择之后，会改变这些宏的定义，以应用前一节所介绍的内存最优化的工作。

特别是表7-1中宏的定义，正如你在*include/linux/init.h*所见的一样，会根据下列符号是否定义在包含*include/linux/init.h*的文件的范围内而改变：

CONFIG_MODULE

当内核支持可加载模块时就会定义（“Loadable module support”配置选项）。

MODULE

当该文件所属的内核组件编译为模块时就会定义。

CONFIG_HOTPLUG

当内核编译为支持“Support for hot-pluggable devices”（“General setup”菜单中的一个选项）时就会定义。

虽然MODULE对不同文件可以有不同的值，但其他两个符号的属性为全内核通用，因此，不是有设置就是没设置，整个内核都一致。

在表 7-1 和表 7-2 的宏中，从 NIC 驱动程序初始化的角度出发，我们最感兴趣的是__init、__exit、__initcall以及__exitcall。把迄今为止所讨论的一切做个总结，图7-4显示了前述几个宏基于符号MODULE和CONFIG_HOTPLUG是否定义（假设内核支持可加载模块，也就是 CONFIG_MODULE 有定义）在节省内存的效力。由图可见，当内核不支持可加载模块和热插拔时，与两种选项都支持时相比可利用的空间就多很多：限制越多，就越能得到最优化。

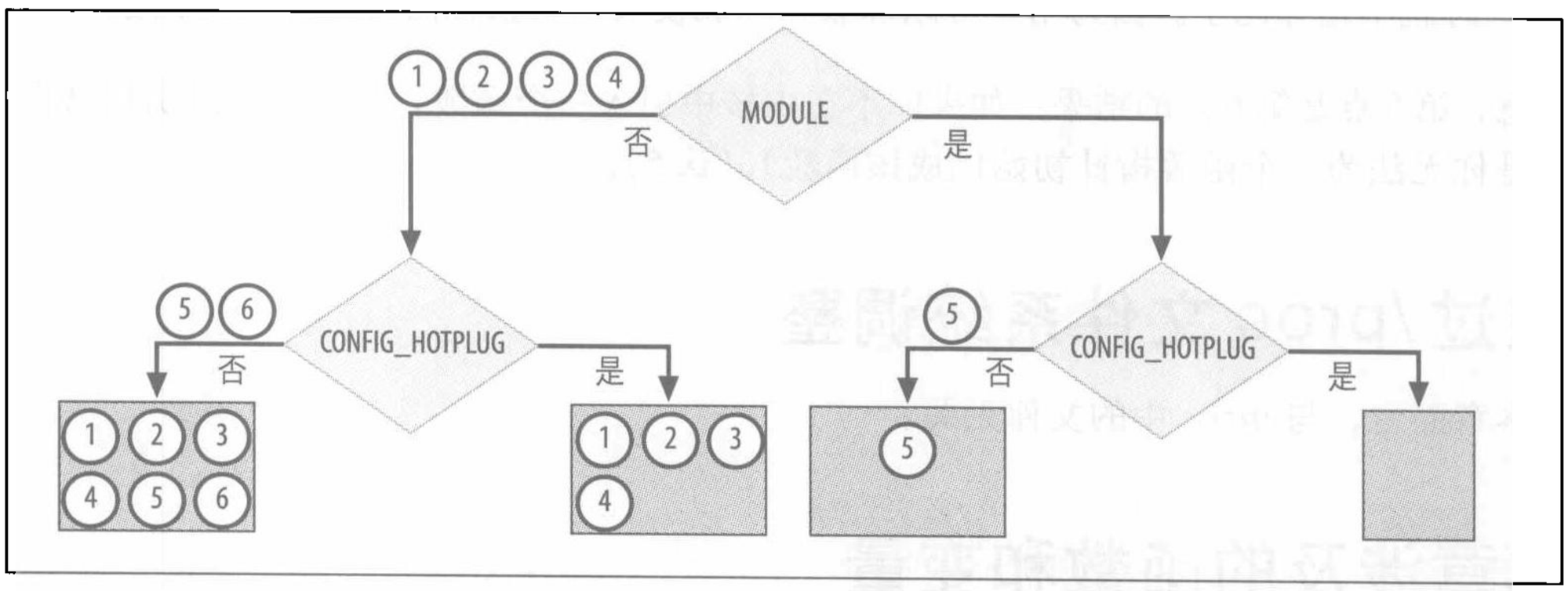


图 7-4：表 7-1 中宏的影响，按文本中所列编号为序

我们逐一说明图 7-4 中点 1~6 的意义，把前面“新型：宏卷标”一节所示的设备驱动程序的通用结构，以及稍早“内存最优化”一节所说的__initcall和__exitcall定义谨记在心。

把模块编译成内核的一部分时，就可做下列的最优化：

1. module_exit函数绝不会被用到，所以将其标记为__exit之后，程序员就可确保在链接期间这些函数将不会被引入映像中。
2. module_init函数只会在引导期间执行一次，所以将其标记为__init之后，程序员就可确保它们在执行后就会被丢弃。
3. module_init(fn)变成__initcall(fn)的别名，如此就能确保 fn 会被do_initcalls执行，如“xxx_initcall 宏”一节所见。

4. module_exit(fn)变成__exitcall(fn)的别名。这会把输入函数的地址放到.exitcall.exit内存节区中，使得内核在关机期间可以更为轻易地予以执行，但是该节区实际会在链接期间被丢弃。

我们把PCI设备作为参考，看一看缺少了对热插拔的支持时会引进哪些其他最优化。这些事与pci_driver->remove函数有关；每当模块卸载时此函数就会被调用，该模块所注册的每个设备都会调用一次（参见第六章“大蓝图”一节）。

5. 无论MODULE是否定义，当内核不支持热插拔时，设备就无法从正在运行中的系统中删除掉。因此，remove函数绝不会被PCI层调用，从而得以初始化为NULL指针。这是由__devexit_p宏指出的。

6. 当内核不支持热插拔或模块时，该模块就不再需要驱动程序用于初始化pci_driver->remove的函数。这是由__devexit宏指出的。注意，当支持模块时就不是如此了。因为用户可以加载和卸载模块，内核就需要remove函数。

注意，第5点是第6点的结果：如果你不在内核中引入一个函数，就无法予以引用（也就是你无法为一个函数指针初始化成该函数）（注5）。

通过/proc文件系统调整

就本章而言，与*/proc*中的文件无关。

本章涉及的函数和变量

表7-4总结了本章所介绍的函数、宏、结构以及变量。

表7-4：本章所介绍的函数、宏、结构以及变量

名称	描述
函数和宏	
__init、__exit、__initcall、__exitcall、__initdata、__exitdata、__devinit、__devexit、__devexit_p、__devinitdata、__devexitdata、xxx_initcall	用于标记一些代码片段的宏，使其具有特殊性质。这些标签可用于让内核映像大小最优化，例如把不需要的代码丢掉
do_initcalls	在引导期间执行所有标记为xxx_initcall宏的函数

注5： 参考第六章的“PCI NIC驱动程序注册范例”一节的程序范例。

表 7-4：本章所介绍的函数、宏、结构以及变量（续）

名称	描述
函数和宏	
init_module、cleanup_module、module_init、module_exit	前两个是函数名字，每个模块都应该提供，以初始化和删除模块。其他两个是宏，允许设备驱动程序编写者为前两个函数使用任意名称
netdev_boot、setup_check、netdev_boot_setup_add	把引导期间配置（如果有的话）应用到特定设备
module_param	定义可选的模块参数，加载模块时可以使用这些参数
数据结构	
kernel_param	存储module_param宏的输入值
obs_kernel_param	存储__setup宏的输入值
netdev_boot_setup、ifmap	netdev_boot_setup为ether=和netdev=存储引导期间的参数。ifmap是netdev_boot_setup字段之一
变量	
dev_boot_setup	netdev_boot_setup结构的数组
NETDEV_BOOT_SETUP_MAX	dev_boot_setup的大小

本章涉及的文件和目录

图 7-5 所示为本章所涉及的文件和目录。

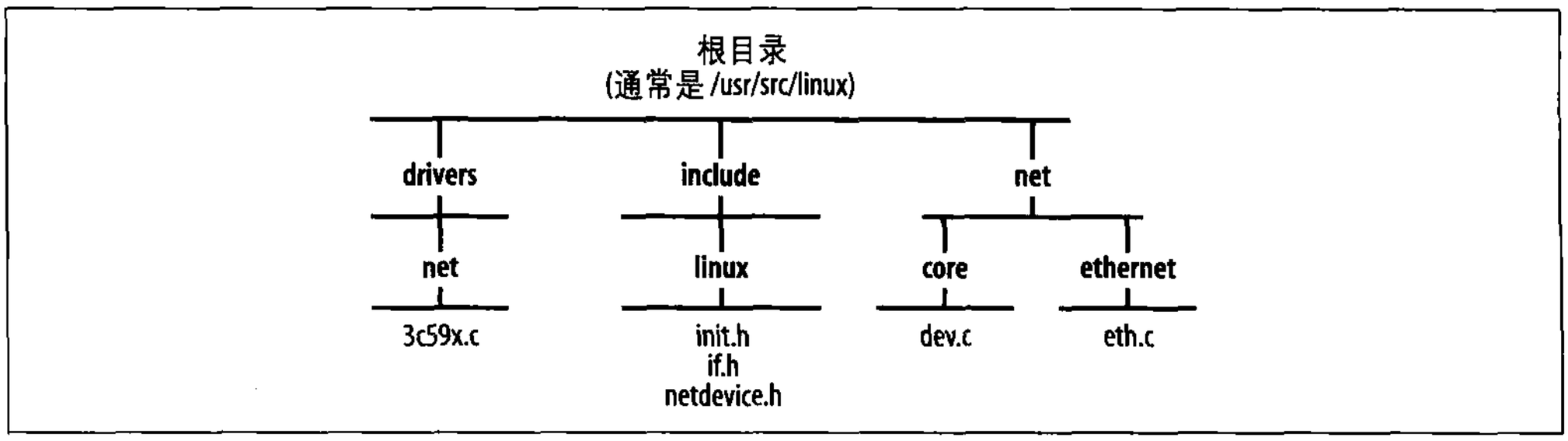


图 7-5：本章涉及的文件和目录

第八章

设备注册和初始化

在第五章和第六章，我们已经知道内核如何验证NIC，以及内核所做的初始化，使得NIC得以和其设备驱动程序对话。本章我们将讨论初始化的其他步骤：

- 网络设备何时以及如何在内核注册。
- 网络设备如何利用网络设备数据库注册，并指派一个net_device结构的实例。
- net_device结构如何组织到hash表和列表，以便于做各种查询。
- net_device实例如何初始化，一部分由内核核心函数完成，一部分由其设备驱动程序完成。
- 就注册而言，虚拟设备和真实设备有何差别。

本章不是教你如何编写NIC设备驱动程序。偶尔我们会走进NIC设备驱动程序代码的细节，但是不会讨论NIC设备驱动程序的整体设计。我们感兴趣的只是注册、设备驱动程序之间的接口，以及各种功能，如链接状态变更侦测以及电源管理。有关设备驱动程序的详尽讨论，可参考《Linux设备驱动程序》一书（O'Reilly出版）。

NIC可用之前，其相关联的net_device数据结构必须先初始化，添加至内核网络设备数据库、配置并开启。不要把注册/除名以及开启/关闭混淆是十分重要的，这是两种不同的概念：

- 如果把加载设备驱动程序模块的动作排除的话，注册和除名是独立于用户之外的，是由内核驱动的。仅仅注册了的设备还不能运转。我们会在“设备注册之时”以及“设备除名之时”两节说明设备注册和除名的时间。
- 开启和关闭设备都需要用户参与。一旦设备已由内核注册，用户就可通过用户命令看到该设备，配置并予以开启。参见稍后的“开启和关闭网络设备”一节。

我们先谈什么事件会触发网络设备的注册和除名。

设备注册之时

网络设备的注册发生在下列几种情况之下：

加载NIC设备驱动程序

如果NIC设备驱动程序内建至内核中，则在引导期间初始化；否则，如果以模块加载，就会在运行期间初始化。每当发生初始化时，该驱动程序所控制的所有NIC都会被注册。

插入可热插拔网络设备

当用户把可热插拔NIC设备插入进来时，内核会通知其驱动程序，而驱动程序再注册该设备（为简化起见，我们假设该设备驱动程序已经加载）。

在第一种情况下，应用的注册模型会在稍后的“NIC注册和除名的架构”一节描述。这种模型适用于所有总线类型，无论注册函数最后是被总线基础架构或是被模块初始化程序调用，都是一样的。例如，在第六章已经知道，加载PCI设备驱动程序如何导致pci_driver->probe（通常命名为xxx_probe）函数的执行。此函数由驱动程序提供，并由其负责设备的注册。本章我们将探讨这些Probe函数是如何实现的。

使用其他总线类型（USB、PCMCIA等等）的设备注册也共享相同架构。我们不会去谈这些总线的基础架构最后如何调用其Probe函数，如第六章PCI的情况。旧式总线可能无法自动侦测设备的存在，因而可能需要设备驱动程序自行探测特定内存地址，使用默认参数或者用户提供的引导期间参数（注1）。我们也不谈这种情况。

设备除名之时

有两种主要情况会触发设备的除名：

卸载NIC设备驱动程序

当然，这仅是针对那些以模块加载的驱动程序，不适于那些内建至内核的驱动程序。当管理员卸载NIC设备驱动程序时，所有相关联的NIC都必须除名。

例如，从第六章已知卸载PCI设备驱动程序时会导致pci_driver->remove函数的执行，通常命名为xxx_remove_one，此函数负责设备的除名。此函数会由PCI层启用，针对正在卸载的驱动程序，每个已注册设备都会执行一次。本章我们将探讨这类函数是如何实现的。

注1：例如，参考*drivers/net/Space.c*中的net_olddevs_init。此函数标记为第七章介绍的devie_initcall宏，在引导期间执行。同一个函数也负责回环设备的注册。

删除可热插拔网络设备

当用户从系统（其运行的内核支持可热插拔设备）删除可热插拔NIC时，则网络设备就会被除名。

分配 net_device 结构

网络设备利用net_device结构定义。因为在内核代码中通常将其称为dev，所以本章我们也时常以dev表示net_device。这些数据结构使用定义在*net/core/dev.c*中的alloc_ netdev分配，需要三个输入参数：

私有数据结构的大小

在“net_device结构的组织”一节中将看到，net_device数据结构可以由设备驱动程序扩充，加上一个私有数据区块，以存储驱动程序的参数。此参数指定该区块的大小。

设备名称

这可能是一部分名称，再由内核通过某种规则，将其完成，以确保独一无二的设备名称。

设置函数

这个函数用于初始化net_device的部分字段。更多细节参见“设备初始化”以及“设备类型初始化：xxx_setup 函数”两节的内容。

返回值是指向已分配的net_device结构的指针，但如果发生错误，就是NULL。

每个设备都会根据其设备类型被分派一个名称，而且为了唯一性还会包含一个数字，该数字的分派是按已注册的同类型设备数量而连续指定。例如，Ethernet设备就依序称为*eth0*、*eth1*等等。根据设备注册的次序，一台设备的名称可能会有所不同。例如，如果你有两块由两个不同模块处理的卡，则设备名称会取决于这两个模块加载的次序。可热插拔设备名称的变化更是不可预测。

因为用户空间配置工具会引用内核所分派的设备名称，因此设备注册的次序就很重要。因为这是用户空间的细节，我们不打算深入探讨，只是会提到一些工具，如*net-tools*套件的*nameif*，它允许你分派固定名称给基于MAC地址的接口。

当传给alloc_netdev的设备名称为*name*%d的形式时（例如，“eth%d”），内核就会使用函数dev_alloc_name以完成该名称。此函数会把%d换成该设备类型中头一个尚未分派的数字。

内核也提供一组内含alloc_netdev的包裹函数，其中一些如表8-1所示，可用于为一组通用设备类型提供正确参数给alloc_netdev（注2）。例如，alloc_etherdev用于Ethernet设备，因此，建立的设备名称形式就是字符串eth再接上一个唯一的数字；第二个自变量将ether_setup指定为设置函数，它可以把net_device结构的一部分初始化为所有Ethernet设备通用的值。

表8-1：alloc_netdev函数的包裹函数

网络设备类型	包裹函数名称	包裹函数定义
以太网（Ethernet）	alloc_etherdev	return alloc_netdev(sizeof_priv, "eth%d", ether_setup);
光纤分布式数据接口（FDDI）	alloc_fddidev	return alloc_netdev(sizeof_priv, "fddi%d", fddi_setup);
高性能并行接口（HPPI）	alloc_hippi_dev	return alloc_netdev(sizeof_priv, "hip%d", hippi_setup);
令牌环（Token Ring）	alloc_trdev	return alloc_netdev(sizeof_priv, "tr%d", tr_setup);
光纤通道（Fibre Channel）	alloc_fcdev	return alloc_netdev(sizeof_priv, "fc%d", fc_setup);
红外数据标准协会（InDA）	alloc_irdadev	return alloc_netdev(sizeof_priv, "irda%d", irda_device_setup);

NIC注册和除名的架构

图8-1（a）所示是NIC设备驱动程序利用网络代码注册的通用方案。图8-1（b）所示则是相反动作，负责除名。虽然这里以PCI Ethernet NIC为例，但这样的方案对其他设备类型而言也是一样的，只是所用的函数名称不同，或者函数启用方式有别，可能依赖于总线代码如何实现而定。

此函数首先使用alloc_etherdev分配net_device结构。alloc_etherdev也会为那些所有Ethernet设备都通用的参数做初始化。接着，驱动程序初始化net_device结构的另一部分，然后调用register_netdev函数为设备注册。

注2：其他一些类似的包裹函数并没有遵循alloc_xxxdev命名规则。此外，有些设备会直接调用alloc_netdev和内核注册，而不使用包裹函数。

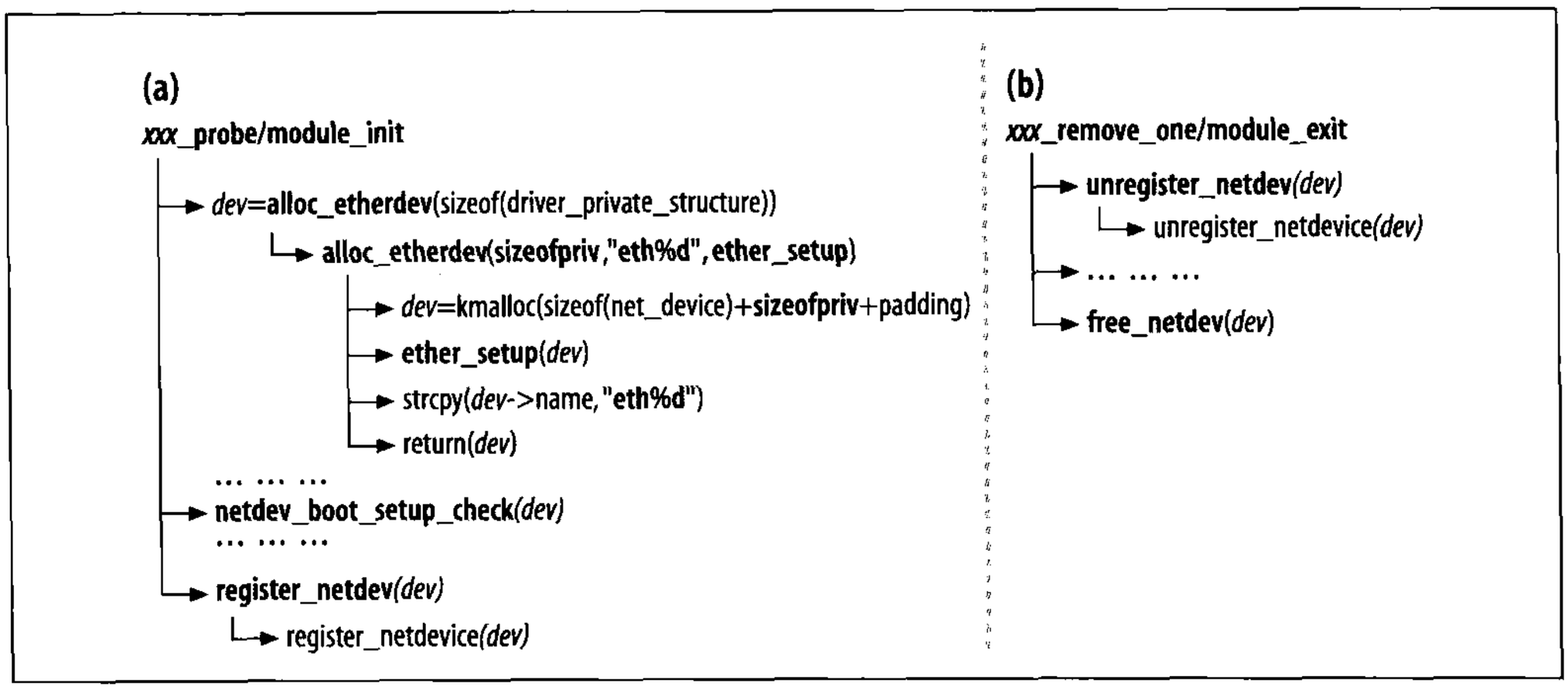


图 8-1：（a）设备注册模型；（b）设备除名模型

注意以下几点：

- 驱动程序会调用适当的内含alloc_netdev的包裹函数，而且只提供其私有数据区块的大小，此例中为alloc_etherdev。表 8-1 列出了一些包裹函数。
- 包裹函数会使用驱动程序所提供的参数以及另外的两个参数（设备名称和初始化函数）来调用alloc_netdev。
- alloc_netdev所分配的内存区块大小包括net_device结构、驱动程序的私有区块以及为了强制对齐所补的一些空白空间。参见本章稍后的图 8-2。
- 有些驱动程序会调用netdev_boot_setup_check，以检查加载内核时用户是否提供了任何引导期间参数。参见第七章的“使用引导选项配置网络设备”一节。
- 新的net_device实例会利用register_netdevice插入至设备数据库（参见稍后的“设备注册”一节）。此外，这里我用的术语是“数据库”，但是在本书其他部分里，则是指数据结构的松散组合，在内核需要时提供便利存取途径，供内核访问信息。

设备的除名的简单形式如图8-1（b）所示，除名时总是会调用unregister_netdevice和free_netdev。有时会显式地调用free_netdev，但有时则是间接通过dev->destructor函数（注 3），如稍后图 8-4 所示。设备驱动程序也必须释放设备所使用的任何资源（IRQ、内存映像等等），但是，本章对那些细节不感兴趣。

注 3：只有少数虚拟设备的设备驱动程序采用这种做法（例如，参见*net/8021q/vlan.c*）。图 8-4 所示的两个调用是彼此互斥的。

设备初始化

在“设备注册之时”一节中，我们已经知道应该初始化什么东西，内核才能与NIC通信。本章其余部分将着眼于更高层的初始化任务。

net_device结构相当大，其字段会一批一批由不同函数予以初始化，而每个函数都负责一组不同的字段子集（注4）。特别是：

设备驱动程序

如IRQ、I/O内存以及I/O端口，其值取决于硬件配置，这些参数就是由设备驱动程序负责的。参见第五章。

设备类型

对同一设备类型系列的所有设备所通用字段的初始化，由xxx_setup函数负责。例如，Ethernet设备使用的是ether_setup。参见“设备类型初始化：xxx_setup函数”一节。

各种功能

强制和可选的功能也必须初始化。例如，队列规则（也就是QoS），是在register_netdevice中初始化的，如“register_netdevice函数”一节所述。其他功能的初始化则是在相关联的模块接到通知，说明有新设备注册之时（参见“设备注册状态通知”一节）。

设备类型初始化属于设备驱动程序初始化的一部分（即xxx_setup是由xxx_ probe调用），因此，驱动程序有机会改写默认设备类型的初始化。参见“可选的初始化和特殊情况”一节的范例。

表8-2所示是由xxx_setup函数初始化的函数指针，以及留给设备驱动程序所做之事（xxx_probe）（注5）：设备类型特定的事以及设备模型特定的事。注意，并非所有设备驱动程序都遵循表8-2所做的区分。例如，有些情况下xxx_setup函数并不初始化任何函数指针（其中一例就是*net/irda/irda_device.c*中的irda_device_setup），而其他情况下则会全部予以初始化（其中一例就是*drivers/net/wireless/airo.c*中的wifi_setup）。

注4：有趣的例外是回环设备，其初始设定工作写在loopback_dev定义中（位于*drivers/net/loopback.c*）。

注5：第二章详细说明了net_device数据结构的所有参数。

表 8-2：由 xxx_setup 和 xxx_probe 初始化的 net_device 函数指针

初始化程序	函数指针名称
xxx_setup	change_mtu
	set_mac_address
	rebuild_header
	hard_header
	hard_header_cache
	header_cache_update
	hard_header_parse
设备驱动程序的探测函数	open
	stop
	hard_start_xmit
	tx_timeout
	watchdog_timeo
	get_stats
	get_wireless_stats
	set_multicast_list
	do_ioctl
	init
	uninit
	poll
	ethtool_ops（实际上是一个函数数组）

表 8-3 类似于表 8-2，不过不是函数指针，而是其他一些 net_device 字段。

表 8-3：xxx_setup 和 xxx_probe 初始化的 net_device 字段

初始化程序	变量名称
xxx_setup	type
	hard_header_len
	mtu
	addr_len
	tx_queue_len
	broadcast
	flags
设备驱动程序的探测函数	base_addr
	irq
	if_port
	priv
	features

有关表 8-2 和 8-3 中字段的具体意义，可参考第二章。

设备驱动程序初始化

由设备驱动程序所初始化的net_device字段通常是由第六章“大蓝图”一节所介绍的xxx_probe 函数负责，如图 8-1（a）所示。

有些驱动程序可以处理不同的设备模型，所以相同参数可根据设备模型和能力而初始化为不同的值。下列片段取自*drivers/net/3c59x.c*驱动程序，显示出函数hard_start_xmit（第十一章中介绍）的初始化会根据设备的能力而有不同的值（注 6）：

```
if (vp->capabilities & CapBusMaster) {
    vp->full_bus_master_tx = 1;
        ... ... ...
}
... ... ...
if (vp->full_bus_master_tx) {
    dev->hard_start_xmit = boomerang_start_xmit;
        ... ... ...
} else {
    dev->hard_start_xmit = vortex_start_xmit;
}
```

设备类型初始化：xxx_setup函数

就最常见的网络设备类型而言，xxx_setup函数可以对net_device结构中对相同类型的所有设备通用的字段做初始化（包括参数和函数指针），例如所有 Ethernet 卡。

从表8-1可知，各种alloc_ xxxdev函数如何把正确的xxx_setup函数传给alloc_ netdev（作为第三个输入参数）。以下是 ether_setup 函数，也就是 Ethernet 设备所用的xxx_setup函数：

```
void ether_setup(struct net_device *dev)
{
    dev->change_mtu             = eth_change_mtu;
    dev->hard_header            = eth_header;
    dev->rebuild_header         = eth_rebuild_header;
    dev->set_mac_address        = eth_mac_addr;
    dev->hard_header_cache      = eth_header_cache;
    dev->header_cache_update    = eth_header_cache_update;
    dev->hard_header_parse      = eth_header_parse;

    dev->type                   = ARPHRD_ETHER;
```

注 6：　能力可以硬编码写入驱动程序，或者从 NIC 的寄存器中读出。

```
    dev->hard_header_len     = ETH_HLEN;
     dev->mtu                      = 1500;
     dev->addr_len                 = ETH_ALEN;
     dev->tx_queue_len             = 1000;
     dev->flags                    = IFF_BROADCAST|IFF_MULTICAST;

     memset(dev->broadcast,0xFF, ETH_ALEN);
}
```

如你所见，此函数只对那些可由任何 Ethernet 卡共享的字段和函数指针做初始化，如 1,500 MTU、链接层广播地址 FF:FF:FF:FF:FF:FF、长度为 1,000 个封包（注 7）的出口队列等等。

使用通用的分配包裹函数以及 xxx_setup函数（如表8-1所示）是最常见的方法。然而：

- 有些类型的设备定义了 Setup 函数，但是没有提供类似表 8-1 所示的通用包裹函数。例如，ARCNET设备（注8）（参见*drivers/net/arcnet/arcnet.c*中的arcdev_setup）和IrDA设备（注9）（参见*net/irda/irda_device.c*中的irda_device_setup）就是如此。
- 通用的*xxx*_setup可能也会被不属于指定种类的设备所用。ether_setup就是一例：非Ethernet设备也会使用它。当特定*xxx*_setup函数的多数初始化工作满足某个设备驱动程序所需时，则该驱动程序也许会使用那个*xxx*_setup函数，然后改写一些不适用的初始化工作。但这种做法不常见。
- Ethernet驱动程序可以使用由 ether_setup（由 alloc_etherdev间接调用）提供的默认初始化，但是要改写掉一些初始化。例如，*3c59x.c* 驱动程序不使用 ether_setup所设置的net_device->mtu值，而是以一个局部变量予以改写。此变量的初值也是由ether_setup所设的相同默认值，但是，遇到可以处理更大值的 NIC 模型时，驱动程序就可以设成更大的值。

可选的初始化和特殊情况

有些情况下，因为对该种类的设备而言没有意义，有些net_device参数就不会初始化；相关的函数指针或值就没初始化，因此就是 NULL。

注 7：这是 Linux 实现时的选择，它并非源自于任何协议规范。根据所配置的出口队列规则的不同，可能不会使用此值。

注 8：ARCNET（Attached Resource Computer Net work，相连资源计算机网）是一种基于令牌总线（token-bus）设计（类似于 802.4）的 LAN 技术，因其确定性的性能，在工业自动化领域得到了广泛应用。Linux为ARCNET提供了一个通用层以及一些设备驱动程序。

注 9：IrDA（Infrared Data Association，红外线数据协会）是一种红外线无线通信的标准。

要避免NULL指针的引用，内核总是要在启用前先确定选用的函数指针被初始化（注10），如下面取自register_netdevice的范例所示：

```
if (dev->init && dev->init(dev) != 0) {
    ...
}
```

值得注意的是，外在因素也会影响到表 8-2 和表 8-3 的字段在何处初始化以及如何初始化。其中一例与 net_device->mtu 字段有关。虚拟设备通常会从其相关联的真实设备那里继承配置参数，然后在需要时予以调整。例如，IP-over-IP 协议所建的虚拟隧道接口，会从其相关联的真实设备那里继承 dev->mtu（这不是自动化过程，虚拟设备驱动程序要负责）。然而，由于IP-over-IP协议所需的额外IP报头的缘故，MTU也必须据此调低（参见 *net/ipv4/ipip.c* 中的 ipip_tunnel_xmit，假设底层为Ethernet设备）。

net_device 结构的组织

net_device 结构有些值得注意的方面，如下所述：

- 由“分配net_device结构”一节可知，调用alloc_netdev以分配net_device结构时，会把驱动程序的私有数据区块的大小传进去（其大小依赖驱动程序而定，有些甚至不使用私有数据）。alloc_netdev会把私有数据附加到net_device结构。图 8-1 显示了该参数是如何传递的，而图 8-2 则显示出在内存分配上的影响。
- 图 8-2 也显示出 net_device 数据结构和选用的驱动程序私有数据结构之间的关系。通常，第二部分的分配和第一部分一起完成，使得调用 kmalloc 一次就已足够，但是，有些情况下驱动程序可能宁可自行分配其私有区块（参见图 8-2 中所示的驱动程序 C）。
- 如图8-2中的范例所示，驱动程序私有区块的大小及其内容不仅会随设备类型而变（例如，Token Ring 与 Ethernet），而且对同一类型的不同设备而言也是如此（例如，两块不同的 Ethernet 卡）。
- dev_base（本节稍后介绍）和 net_device 中的 next 指针指向 net_device 结构的开头，而非已分配区块的开头。然而，开头补空白空间的大小则存储在 dev->padded，使得内核得以在时机到来时释放整个内存区块。

net_device 数据结构插入在一个全局列表（如图 8-2 所示）和两张 hash 表（如图 8-3 所示）中。这些不同的结构可让内核按需求浏览或查询net_device数据库。以下是其细节：

注 10：　第一章中详细说明了 VFT 的使用。

dev_base

内含所有net_device实例的全局列表能够让内核轻易浏览设备，例如，取得某些统计数据，因为用户命令的结果而必须改变所有设备的某项配置，或者找出吻合特定搜寻准则的设备。

因为每个驱动程序对私有数据结构都有其自己的定义，net_device结构全局列表所链接的元素可能会有不同的大小（参见图 8-2 所示）。

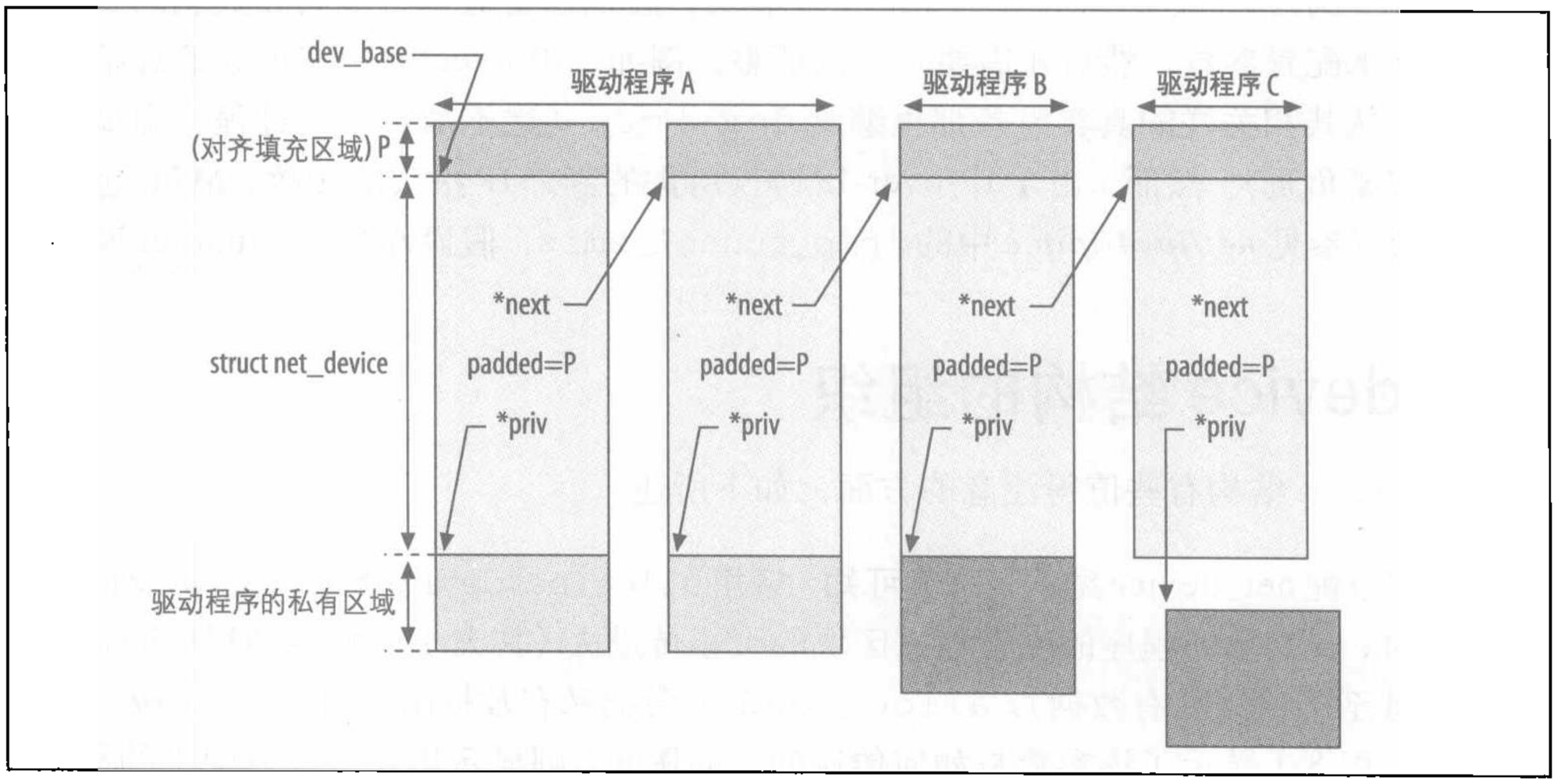


图 8-2：已注册设备全局列表

dev_name_head

这是一张hash表，以设备名称为索引。例如，要通过ioctl接口应用某项配置变更时，就有其用处。老一代配置工具通过ioctl接口与内核对话，通常会以设备名称引用设备。

dev_index_head

这是一张 hash 表，以设备 ID dev->ifindex 为索引。对 net_device 结构做交叉引用时，通常会存储设备ID或者指向net_device结构的指针。dev_index_head 存储设备 ID 供交叉引用。此外，新一代配置工具 *ip*（来自于 IPROUTE2 套件）会通过 Netlink 套接字与内核对话，通常就是以设备 ID 引用设备。

查询

最常见的查询都是通过设备名称或设备 ID 进行。这两种查询的实现是由 dev_get_by_name 和 dev_get_by_index 负责，其所用的就是前一节所讨论的两张 hash 表。也

有可能根据设备类型和MAC地址搜寻net_device实例。这种查询用的是dev_base列表。

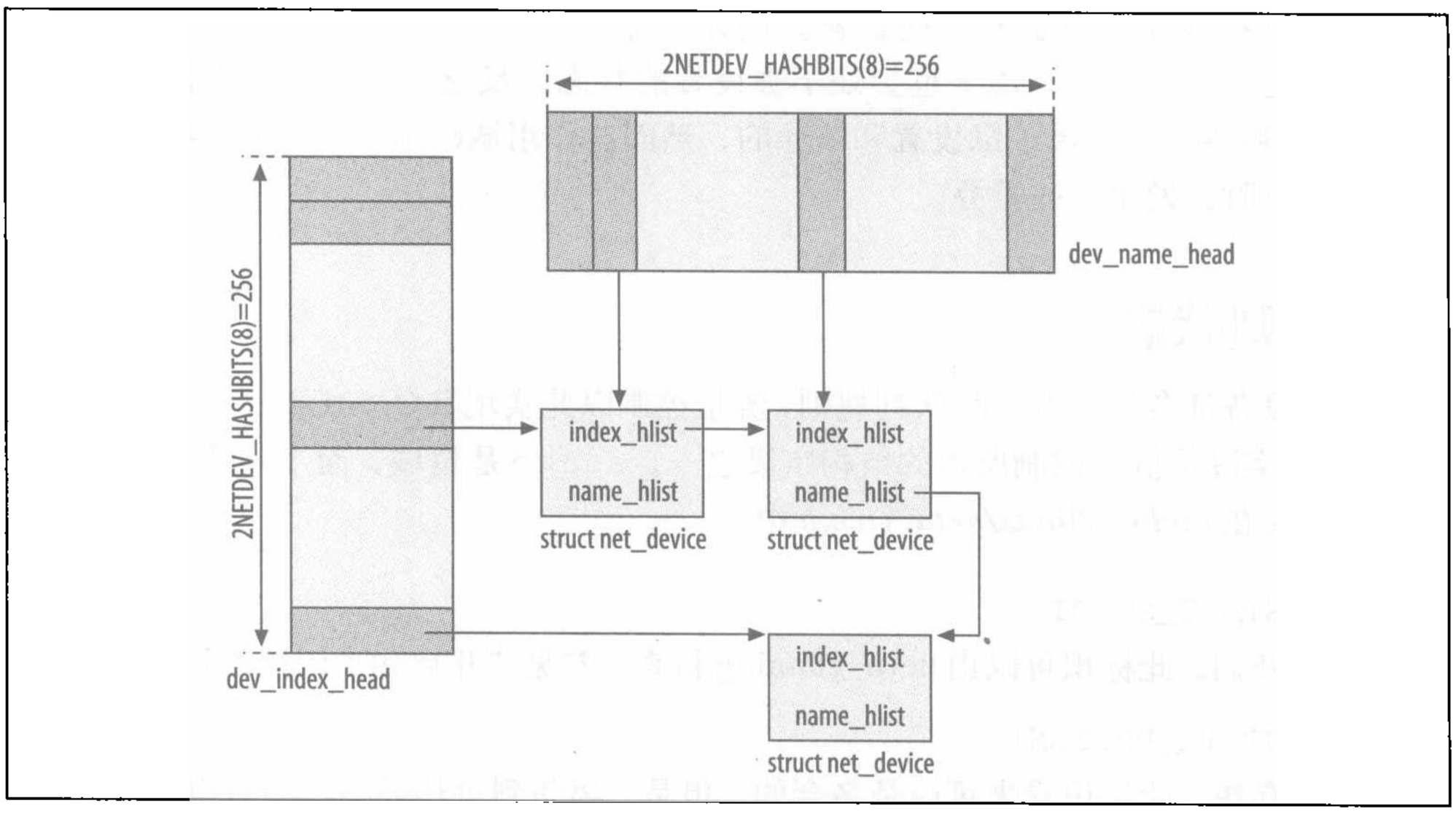


图8-3：根据设备名称和设备索引值搜寻net_device的hash表

所有查询，包括dev_base列表和那两张hash表，都会由dev_base_lock锁保护。

所有查询函数都定义在*net/core/dev.c*中。

设备状态

net_device结构有各种字段可定义设备当前状态，如下所示：

flags

用于存储各种标识的位域。多数标识都代表设备的能力。然而，其中之一的IFF_UP是用于指出该设备是开启或关闭。可以在*include/linux/if.h*中找到IFF_*XXX*标识列表。

reg_state

设备注册状态。“注册状态”一节列出可以分派给此字段的一些值，以及何时该值可以变更。

state

和其队列规则有关的设备状态。参见“队列规则状态”一节。

你会发现这些变量有时会有一些重叠。例如，每次IFF_UP在flags中被设置时，__LINK_STATE_START也会处于被设置的状态，反之亦然。这两个标识都是由dev_open和dev_close予以设置和清除的，然而其作用域则不同。此外，编写模块化程序时，有时会发生一些重叠。

队列规则状态

每个网络设备都会被分派一种队列规则，流量控制以此实现其QoS机制。net_device的state字段是流量控制所用的结构字段之一。state是位域，而下列是可以设置的标识，定义在*include/linux/netdevice.h*中。

__LINK_STATE_START

设备开启。此标识可以由netif_running检查，参见“开启和关闭网络设备”一节。

__LINK_STATE_PRESENT

设备存在。此标识看来可能是多余的，但是，要想到可热插拔设备可以暂时删除。当系统进入挂起模式然后再重新继续时，此标识也会被清除然后再取回其值。此标识可以由netif_device_present检查，参见“register_netdevice函数”一节。

__LINK_STATE_NOCARRIER

没有载波（carrier）。此标识可以由netif_carrier_ok检查，参见“链接状态变更侦测”一节。

__LINK_STATE_LINKWATCH_EVENT

设备的链接状态已变更。参见“调度并处理链接状态变更事件”一节。

__LINK_STATE_XOFF

__LINK_STATE_SHED

__LINK_STATE_RX_SCHED

这三个标识由负责管理设备的入口和出口流量的代码所使用。我们会在第三部分看到其用法。

注册状态

设备和网络协议栈之间的注册状态存储在net_device结构中的reg_state字段内。此字段所取的NETREG_*XXX*值都定义在*include/linux/netdevice.h*中（放在net_device结构定义中）。到了下一节，我们会说明它们彼此间的关系。以下予以简述：

NETREG_UNINITIALIZED

定义成0。当net_device数据结构已分配而且其内容都清为零时，此值代表的就是dev->reg_state中的0。

NETREG_REGISTERING

net_device结构已添加到稍早的“net_device结构的组织”一节中提及的那些结构内，但是，内核依然要在*/sys*文件系统中添加一个项目。

NETREG_REGISTERED

设备已完成注册。

NETREG_UNREGISTERING

net_device结构已从稍早的“net_device结构的组织”一节中提及的那些结构中删除掉了。

NETREG_UNREGISTERED

设备已完全除名（包括删除*/sys*中的项目），但是net_device结构还没释放掉。

NETREG_RELEASED

所有对net_device结构的引用都已释放。从网络代码的观点看，此数据结构已可释放。然而，由*sysfs*来决定是否这样做。参见“引用计数”一节。

设备的注册和除名

网络设备通过register_netdev和unregister_netdev在内核注册和除名。这两个只是简单的包裹函数，负责上锁，然后分别启用register_netdevice和unregister_netdevice函数。我们已在图8-1中介绍过这些函数了，它们都定义在*net/core/dev.c*中。

图8-4显示出net_device可以设成的注册状态，以及前面提过的函数在此图中的哪些位置，同时也显示出其他关键函数在何处受到调用。这些函数都会在后续几节加以说明。但特别要注意如下几点：

- 状态的改变会用到界于NETREG_UNINITIALIZED和NETREG_REGISTERED之间的中间状态。这些进程是由netdev_run_todo处理，在“切割操作：netdev_run_todo”一节加以说明。
- 为设备注册和除名时，设备驱动程序可以使用两个net_device虚拟函数init和uninit，分别对私有数据做初始化以及清理工作。参见“虚拟设备”一节。
- 对设备除名时，除非对相关联的net_device数据结构的引用都已全部释放，否则无法完成：netdev_wait_allrefs不会返回，直到条件满足。

- 设备的注册和除名都是由 netdev_run_todo 完成的。在“切割操作：netdev_run_todo”一节就会知道，register_netdevice和unregister_netdevice如何与 netdev_run_todo 交互。

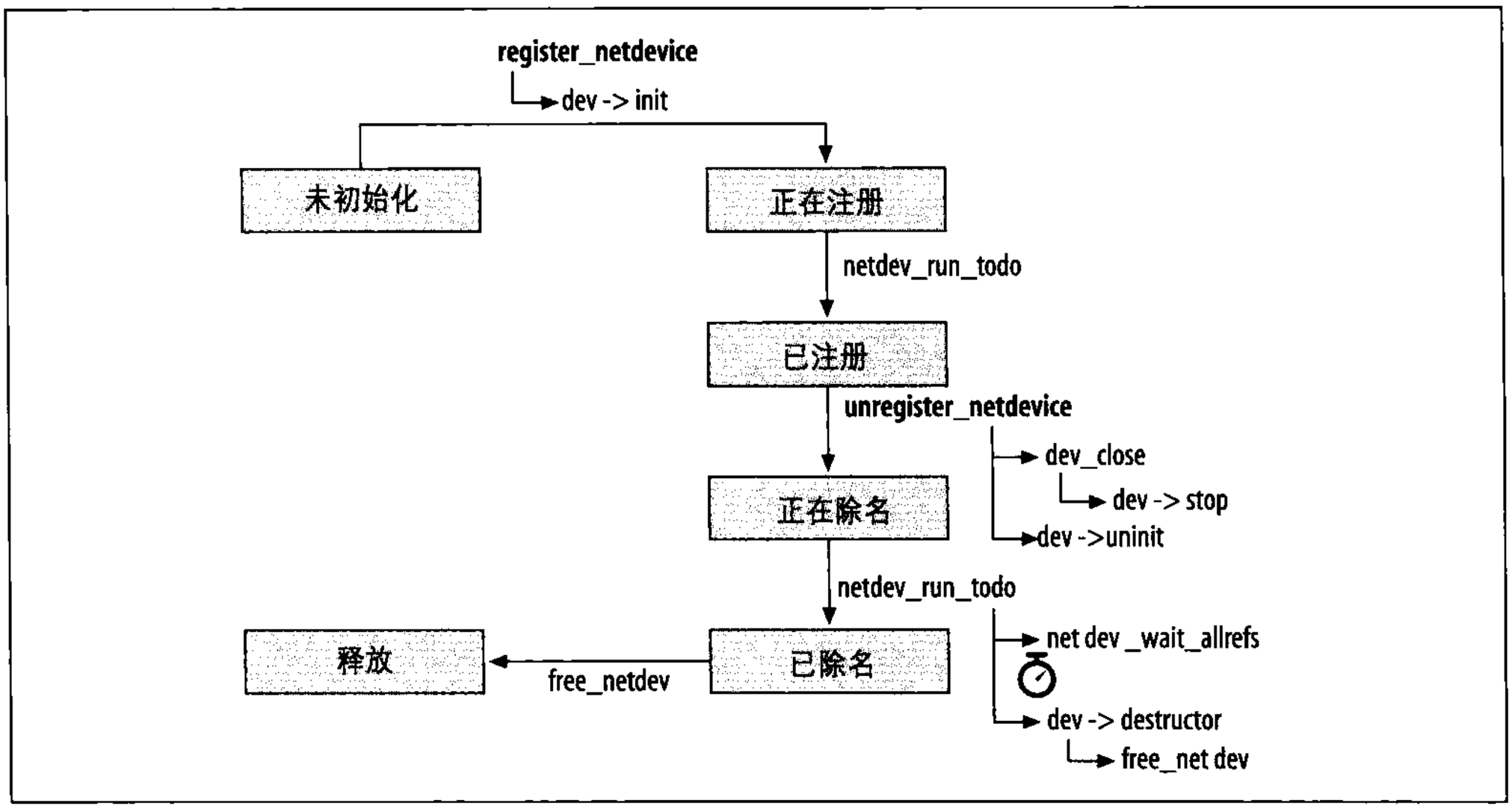


图 8-4：net_device 的注册状态机

切割操作：netdev_run_todo

register_netdevice会负责一部分的注册，然后再让netdev_run_todo予以完成。一开始，只看程序代码可能看不清到底发生了什么。通过图8-4的协助来说明其工作方式。

对net_device结构做修改时，会受到通过 rtnl_lock和rtnl_unlock操作的 Routing Netlink 信号量的保护；这也就是为什么 register_netdev 在开始之时会取得该锁（信号量），然后在返回前予以释放（细节可参考“上锁”一节）。一旦 register_netdevice 完成了其份内工作，就会以 net_set_todo 把新 net_device 结构添加到 net_todo_list。这份列表所含的设备，其注册（或除名，稍后会看到）必须完成才行。这份列表不会由另一个内核线程处理，或者通过定期定时器处理，而是由 register_netdev 在释放锁时以间接方式予以处理。

因此，rtnl_unlock 不仅释放该锁，也会调用 netdev_run_todo（注 11）。netdev_run_todo 会浏览 net_todo_list 数组，然后完成其全部的 net_device 实例注册。

注 11：rtnl_unlock是包裹函数，内含信号量原素（primitive）up。当 up 被直接调用时（如 rtnetlink_rcv 的情况），就会明确调用 netdev_run_todo。参见“上锁”一节。

任何时刻，只有一个CPU可以执行net_run_todo。串行化是通过net_todo_run_mutex互斥体强制实施的。

设备的除名处理方式也相同，参见图 8-5（b）所示。

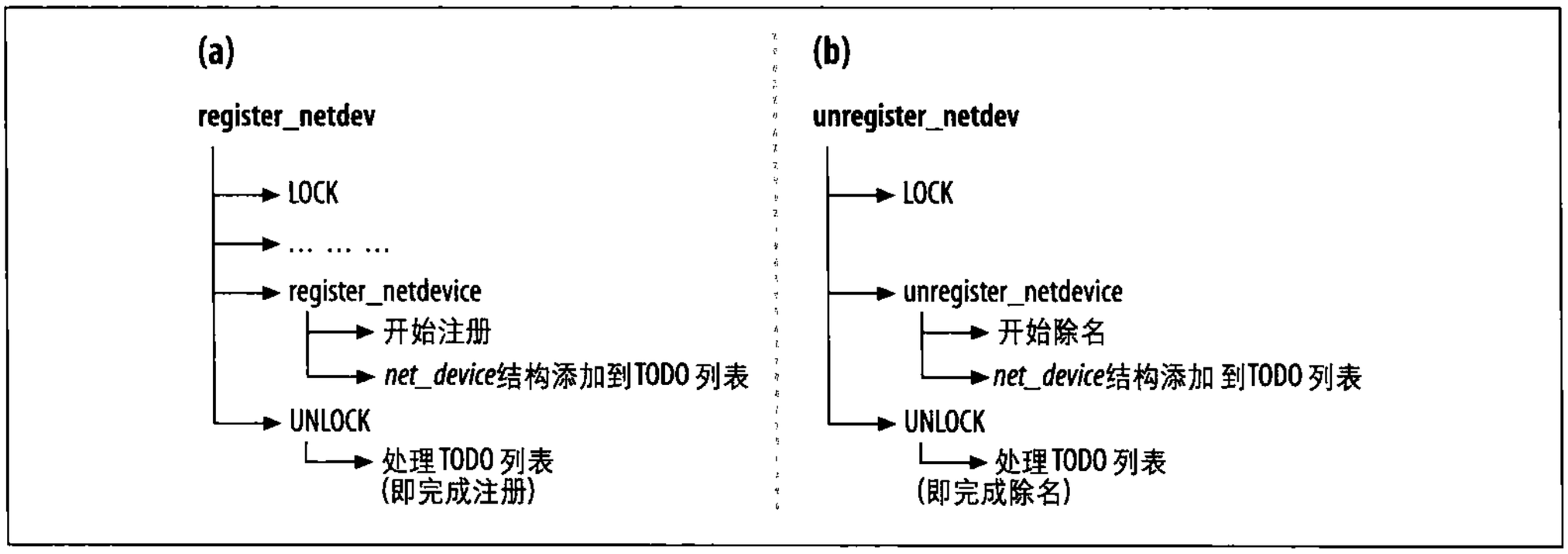


图 8-5：register_netdev 和 unregister_netdev 的结构

netdev_run_todo完成设备的注册或除名所做之事，具体会在“register_netdevice函数”和“unregister_netdevice 函数”两节的结尾处予以说明。

注意，由于netdev_run_todo 所处理的注册和除名任务不用持有该锁，此函数可以安全地进入休眠状态，使信号量可用。你会在“引用计数”一节中的范例了解到为什么这是好事。

根据图 8-5 的模型可以看出，每次调用 netdev_run_todo 时，内核都不能在 net_todo_list 中放置一个以上的 net_device 实例。如果 register_netdev 和 unregister_netdev 只添加一个 net_device 实例到该列表中，然后释放该锁时就立刻予以处理，又如何会有一个以上的元素呢？举例来说，设备驱动程序有可能使用下列的循环，一次把其所有设备都除名（例如，参见*drivers/net/tun.c*中的tun_cleanup）：

```
rtnl_lock();
循环（此驱动程序所驱动的每个设备） {
    ... ... ...
    unregister_netdevice(dev);
    ... ... ...
}
rtnl_unlock();
```

这种做法比下列做法要好，即在循环的每一轮中取得和释放该锁，然后处理 net_todo_list：

```
循环（此驱动程序所驱动的每个设备） {
    ... ... ...
    unregister_netdev(dev);
    ... ... ...
}
```

设备注册状态通知

内核组件和用户空间应用程序可能都想知道何时发生网络设备注册、除名、关闭或者开启之事。这类事件的通知是通过两种通道传送的：

netdev_chain

内核组件可以注册此通知链。参见后续一节“netdev_chain 通知链”。

Netlink 的 RTMGRP_LINK *多播群组*

用户空间应用程序可以注册Rtnetlink的RTMGRP_LINK多播群组。参见“Rtnetlink链接通知”一节。

netdev_chain 通知链

第四章已谈过通知链的定义及其用法。设备的注册和除名在各个阶段的进展都是通过netdev_chain通知链报告的。此链定义在*net/core/dev.c 中*，而对此类事件感兴趣的内核组件可以通过register_netdevice_notifier和unregister_netdevice_notifier分别对该链注册或除名。

所有由netdev_chain报告的NETDEV_*XXX*事件都列在*include/linux/notifier.h*中。以下是我们在本章看过的几种事件以及触发这些事件的条件：

NETDEV_UP

NETDEV_GOING_DOWN

NETDEV_DOWN

送出NETDEV_UP以报告设备已开启，而且此事件是由dev_open产生的。

当设备要关闭时，就会送出NETDEV_GOING_DOWN。当设备已关闭时，就会送出NETDEV_DOWN。这些事件都是由dev_close产生的。有关这三个事件的细节，可参见“开启和关闭网络设备”一节。

NETDEV_REGISTER

设备已注册。此事件是由register_netdevice产生的，参见“register_netdevice函数”一节。

NETDEV_UNREGISTER

设备已经除名。此事件是由unregister_netdevice产生的，参见“unregister_netdevice函数”一节。

此外，还有其他一些事件：

NETDEV_REBOOT

因为硬件失败，设备已重启。目前没有用。

NETDEV_CHANGEADDR

设备的硬件地址（或相关联的广播地址）已改变。

NETDEV_CHANGENAME

设备已改变其名称。

NETDEV_CHANGE

设备的状态或配置已经改变。此事件会用在NETDEV_CHANGEADDR和NETDEV_CHANGENAME没包括在内的所有情况下。目前，当dev->flags中有些改变时，就会用到此事件。

产生NETDEV_CHANGEXXX通知信息通常是为了响应用户配置变更。

注意，向链注册时，register_netdevice_notifier也会（仅对新注册者）重放当前系统已注册设备所有过去的NETDEV_REGISTER和NETDEV_UP通知信息。这样就能给新注册者有关已注册设备当前状态的清晰图像。

有不少内核组件都在netdev_chain注册。其中一些如下所述：

路由

例如，路由子系统使用此通知信息新增或删除和此设备相关联的所有路由项目。参见第三十二章。

防火墙

例如，如果防火墙之间把来自某设备的包暂存在缓冲区内，则必须根据其策略把包丢掉，或者采取另一种动作。

协议代码（也就是ARP、IP等等）

例如，当你改变一个本地设备的MAC地址时，ARP表也必须据此更新。参见相关的协议章节以了解细节。

虚拟设备

参见“虚拟设备”一节。

RTnetlink

参见后续的“RTnetlink 链接通知”一节。

RTnetlink 链接通知

当设备的状态或配置中有变更时，就会用rtmsg_ifinfo把通知信息传送给Link多播群组 RTMGRP_LINK。其中一些通知信息如下：

- 当netdev_chain通知链接收到一个通知信息时。RTnetlink会注册前一节所提及的netdev_chain,然后重放其接收到的通知信息。
- 当一个已关闭的设备开启时或者处于相反的过程(参见netdev_state_change)。
- 当 net_device->flags 中的一个标识有变动时，例如，通过用户配置命令修改(参见 dev_change_flags)。

netplugd是守护进程，属于*net-utils*套件，会监听这些通知信息，然后根据用户配置文件而反应。参见 *netplugs* 指南页（manpage）中的细节。

设备注册

设备注册不是简单地把net_device结构插入到“net_device结构的组织”一节所介绍的全局列表和hash表就行了，其基本模型如图8-1(a)所示。设备注册还涉及net_device结构中一些参数的初始化、产生广播通知信息以通知其他内核组件有关此次注册，以及其他任务。设备是以register_netdev注册，而register_netdev只是简单的包裹函数，其内含有 register_netdevice。此包裹函数主要是负责上锁以及完成名称，如稍早的“分配 net_device 结构”一节所述。该锁会保护 dev_base 已注册的设备列表。

register_netdevice 函数

如图 8-5（a）所示，register_netdevice 会着手进行设备的注册，然后调用 net_set_todo，而 net_set_todo 最终会要求 netdev_run_todo 完成注册。

以下是 register_netdevice 所进行的主要任务：

- 对 net_device 的某些字段做初始化，包括用于上锁的那些字段，具体见“上锁”一节。
- 当内核支持 Divert 功能时，就分配一个该功能所需的配置区块，然后链接至 dev->divert。这是由 alloc_divert_blk 负责完成的。

- 如果设备驱动程序已对dev->init做了初始化，那就执行该函数。参见“虚拟设备”一节。
- 用dev_new_index分派一个独一无二的识别码给设备。此识别码的产生是通过一个计数器，每次一个新设备新增至系统时就递增一次。此计数器是32位变量，所以dev_new_index包含一个if子句，用以处理环绕问题(wraparound)，此外还有另一个if子句，用以处理该变量遇到一个已经分派过的值的可能情况。
- 把net_device附加到全局列表dev_base，然后插入到那两张hash表，如“net_device结构的组织”一节所述。虽然把该结构添加到dev_base的头部会比较快，但是，内核还是有机会浏览整个列表以检查有无重复的设备名称。设备名称也会通过dev_valid_name检查是否为无效名称。
- 检查功能标识是否有无效的组合。例如：
  - 不支持L4硬件校验和时，散播/聚集－DMA无用，因此，这种情况下应该关闭。
  - TCP节段卸载（TSO）需要散播/聚集－DMA，因此，当后者不支持时就要关闭。

  参见第十九章有关L4校验和的细节。
- 设置dev->state中的__LINK_STATE_PRESENT标识，让设备能为系统所用(可见可用)。例如，当可热插拔设备拔除时，或者当支持电源管理的系统进入挂起模式时，该标识就会清除。参见“队列规则状态”一节。

  此标识的初始化不会触发任何动作，相反，其值会经过缜密的检查，以过滤出非法请求或者取得设备状态。
- 设备的队列规则通过dev_init_scheduler做初始化，由流量控制用于实现QoS。队列规则定义出口封包如何排入出口队列，以及如何退出出口队列、定义开始丢掉封包前有多少封包可以排入队列中等等。参见第十一章的“队列规则接口”一节。
- 通过netdev_chain通知链，通知所有对此设备注册感兴趣的子系统。通知链已在第四章描述过了。

当netdev_run_todo被调用完成注册时，只会更新dev->reg_state，然后在*sysfs*文件系统中注册该设备。

除了内存分配的问题外，只有当设备名称无效或重复，或者因某些原因使得dev->init失败时，设备注册才会失败。

设备除名

要把设备除名，内核和相关联的设备驱动程序必须复原注册期间所执行过的所有操作和其他事项：

- 以dev_close关闭设备，如“开启和关闭网络设备”一节所述。
- 释放所有已分配的资源（IRQ、I/O内存和I/O端口等等）。
- 从全局列表和那两张hash表中把net_device结构删除，如“net_device结构的组织”一节所介绍。
- 一旦该结构的所有引用都释放后，就释放net_device结构、该驱动程序的私有数据结构，以及其他任何相链接的内存区块（参见图8-2所示）。net_device结构的释放是通过free_netdev完成。当内核被编译成支持*sysfs*时，free_netdev就能使*sysfs*释放该结构。
- 删除曾经添加到*/proc*和*/sys*文件系统的文件。

注意，每当设备之间存在依赖性而把其中一个设备除名时，就会强制其他所有（或部分）设备除名。参见“虚拟设备”一节的范例。

net_device（由变量dev表示）中的三个函数指针在设备除名时会派上用场：

dev->stop

此函数指针将由设备驱动程序初始化为其自有函数之一。关闭设备时，就会启用dev_stop（参见“开启和关闭网络设备”一节）。这里所处理的常见任务包括以netif_stop_queue（注12）停止出口队列、释放硬件资源以及停止设备驱动程序使用的任何定时器等等。

虚拟设备不需要释放任何硬件资源，但是需要负责其他高层问题。参见“虚拟设备”一节。

dev->uninit

此函数指针也是由设备驱动程序初始化为其自有函数之一。虽然很少，但目前有些隧道虚拟设备会对此函数指针初始化：指向一个函数，而此函数主要负责引用计数。

dev->destructor

当使用该指针时，通常是初始化为free_netdev或者内含free_netdev的包裹函

注12： netif_xxx_queue函数将在第十一章描述。

数。然而，destructor通常不做初始化，只有少数虚拟设备使用。多数设备驱动程序在unregister_netdevice之后都会直接调用free_netdev。

图8-4所示为这三个函数启用的时机以及次序。

unregister_netdevice函数

unregister_netdevice接受一个参数，也就是要删除的net_device结构的指针：

```
int unregister_netdevice(struct net_device *dev)
```

第九章我们会详细说明网络代码如何使用软件中断（软式IRQ）以处理封包的传输（net_tx_action）和接收（net_rx_action）。就目前而言，你可以把这些函数视为设备驱动程序和上层协议之间的接口。两次调用synchronize_net是用于让unregister_netdevice能和接收引擎（net_rx_action）同步，使其被unregister_netdevice更新后不会访问旧数据。

其他由unregister_netdevice负责的任务包括：

- 如果设备没关闭，就需要先以dev_close予以关闭（参见“开启和关闭网络设备”一节）。
- 接着，net_device实例会从全局列表dev_base和那两张hash表中删除，在“net_device结构的组织”一节介绍。注意，这样并不足以禁止内核子系统使用该设备：内核子系统依然持有net_device数据结构的指针。这也就是为什么net_device会使用引用计数以记录对该结构的引用数目还剩多少（参见“引用计数”一节）。
- 所有和该设备相关联的队列规则实例都会使用dev_shutdown销毁。
- NETDEV_UNREGISTER通知信息会送往netdev_chain通知链，让其他内核组件知晓此事。参见“设备注册状态通知”一节。
- 用户空间也必须得知有关除名之事。例如，系统中有两块NIC，可用于访问因特网，而此通知信息就可用于启动第二个设备。参见“设备注册状态通知”一节。
- 任何链接至net_device结构的数据区块都会被释放。例如，多播数据dev->mc_list会使用dev_mc_discard删除，而Divert区块会使用free_divert_blk删除等等。没有在unregister_netdevice中明确删除的东西，都应该由上一点中提到的负责处理通知信息的函数处理例程予以删除。
- 无论在register_netdevice中的dev->init做了什么，这里dev->uninit都要予以复原。

- 诸如绑定这类功能可让你把一组设备群集起来，将其视为有特殊性质的单一虚拟设备。在这些设备中，通常会有一个设备被选为主设备，因为主设备会在此群组内扮演特殊角色。显然，要删除的设备应该释放任何对主设备的引用：此时，dev->master的值不为NULL就是bug了。再以绑定为例，由于早先的几行程序代码所送出的NETDEV_UNREGISTER通知信息，dev->master的引用会被清除。

最后，调用net_set_todo使net_run_todo完成除名，如“切割操作：netdev_run_todo”一节所述。此外，引用计数的值会以dev_put递减。net_run_todo将使设备从*sysfs*中除名，把dev->reg_state改成NETREG_UNREGISTERED，等待所有引用都消失，然后调用dev->destructor以完成除名。

引用计数

net_device结构无法释放，除非对该结构的所有引用都已释放。该结构的引用计数放在dev->refcnt中，每次以dev_hold或dev_put新增或删除引用时，其值就会更新一次。

当设备以register_netdevice注册时，dev->refcnt的初值就设为1。因此，这第一个引用就会由负责网络设备数据库的内核代码所持有。只要调用unregister_netdevice，这个引用就会被释放。也就是说，除非设备要除名了，否则dev->refcnt绝不会降到零。因此，和其他内核对象（当引用计数减到零时，就会由xxx_put函数予以释放）不同的是，除非你把设备从内核除名，否则net_device结构就不会被释放。在“设备除名之时”一节中，我们已经知道导致设备除名的条件了。

总之，在unregister_netdevice结束时调用dev_put不足以使net_device实例有删除的资格：内核仍然必须等待，直到所有引用都释放为止。但是，因为该设备除名后就不再能用，内核必须通知所有引用持有者，使其能释放其引用。其做法是送出一个NETDEV_UNREGISTER通知信息给netdev_chain通知链。也就是说，引用持有者应该对通知链注册，否则就无法接收这类通知信息而据此采取行动了。

如“切割操作：netdev_run_todo”一节所述，unregister_netdevice会着手进行除名流程，然后让netdev_run_todo予以完成。netdev_run_todo会调用netdev_wait_allrefs，一直等待下去，直到所有对net_device的引用都被释放为止。下一节会说明netdev_wait_allrefs的内部细节。

函数 netdev_wait_allrefs

如图8-6所示，netdev_wait_allrefs由一个循环组成，只有当dev->refcnt减至零时才会结束。此函数每秒都会送出一个NETDEV_UNREGISTER通知信息，而每10秒都会在

控制台上打印出一条警告。剩余时间都在休眠。此函数不会放弃，直到对输入net_device结构的所有引用都已释放为止。

有两种常见情况需要传送一个以上的通知信息：

bug

例如，有段代码持有对net_device结构的引用，但是因为没有在netdev_chain通知链注册，或者因为没有正确处理通知信息，使其无法释放引用。

未决的定时器

例如，假设当定时器到期时要执行的那个函数必须访问的数据中，包含了对net_device结构的引用。在这种情况下，你必须等待直到该定时器到期，而且其处理函数有望会释放其引用。

注意，unregister_netdevice把锁放开之后就会启用netdev_run_todo，如“切割操作：netdev_run_todo”一节所述，这意味着无论是谁启动除名（几乎都是驱动程序），都会进入休眠状态等待netdev_run_todo完成其份内工作。

当此函数发送通知信息时，也会处理未决的链接状态变更事件。链接状态变更事件会在“链接状态变更侦测”一节谈及。这里你只要知道，当设备正被除名时，内核得知设备上有一链接状态变更事件时不需要做任何事。当当前设备状态指出该设备即将被删除时，处理链接状态变更事件列表时和那个正被删除的设备相关联的事件都会被关联为空操作（no-op），所以，结果就是该事件列表会被清除，而实际上只有其他设备的事件将会得到处理。这是简单的方式，把链接状态变更队列中和即将消失的设备相关联的事件清理掉。

开启和关闭网络设备

设备一旦注册就可用了，但是，除非由用户（或用户应用程序）明确的开启，否则还是无法传输和接收数据流。开启设备的请求由定义在*net/core/dev.c* 中的dev_open负责。开启设备之时，有下列任务要做：

- 如果有定义的话，调用dev->open。并非所有设备驱动程序都初始化此函数。
- 设置dev->state中的__LINK_STATE_START标识，把设备标识为开启和运行中。
- 设置dev->flags中的IFF_UP标识，把设备标识为开启。

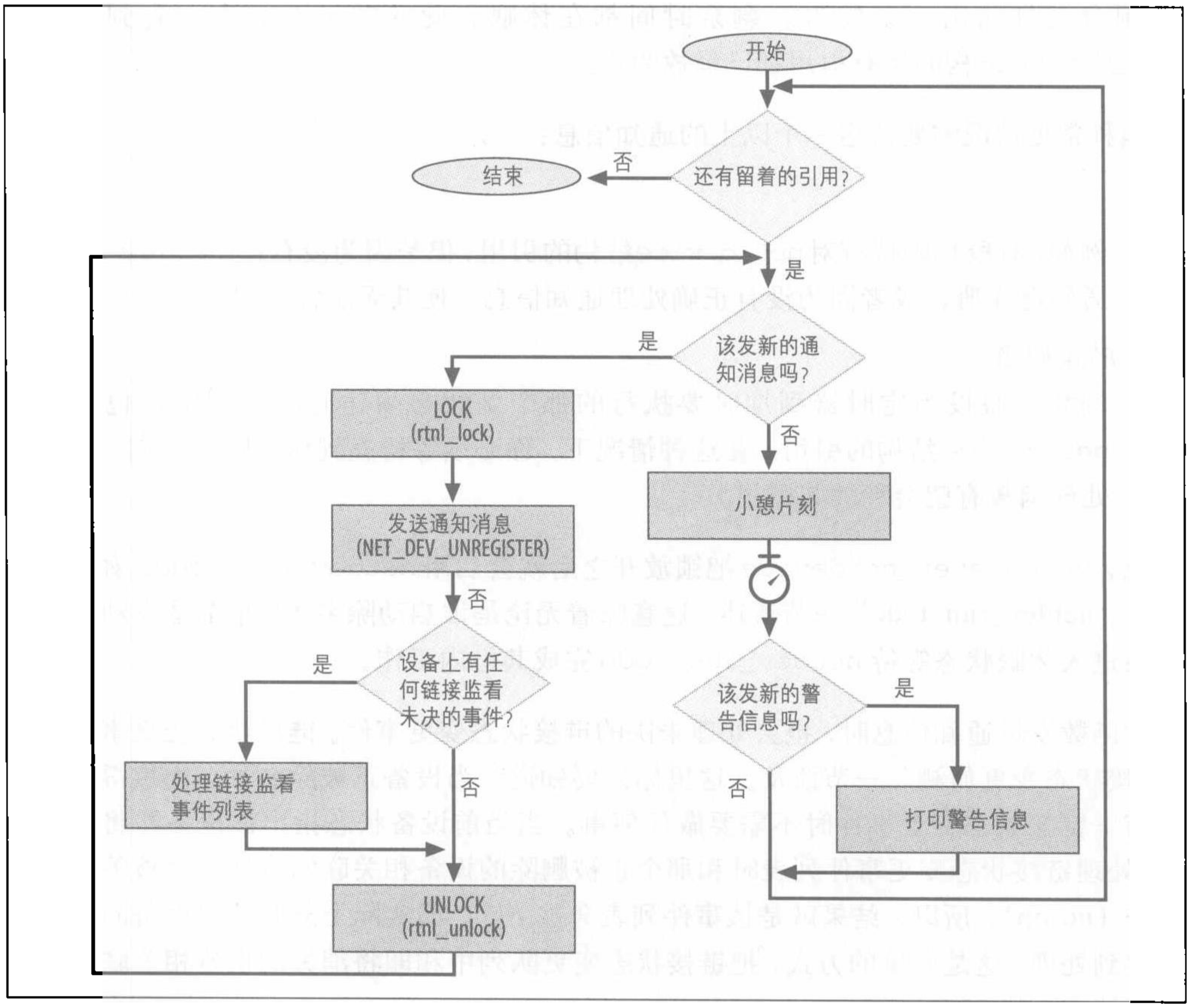


图 8-6：函数 netdev_wait_allrefs

- 调用dev_activate以初始化由流量控制使用的出口队列规则，然后启动看门狗定时器（注 13）。如果流量控制没有用户配置，就指定默认的 FIFO（First In, First Out，先进先出）队列。
- 传送NETDEV_UP通知信息给netdev_chain通知链，以通知感兴趣的内核组件，该设备现在已经开启了。

当设备必须被显式地开启时，也可以由用户命令显式地关闭或由其他事件隐式地关闭。例如，设备除名前，首先得关闭（参见“设备除名”一节）。网络设备是用dev_close关闭的。关闭设备时，有下列任务要做：

注 13： 参见第十一章有关看门狗定时器的细节。

- 传送NETDEV_GOING_DOWN通知信息给netdev_chain通知链，以通知感兴趣的内核组件该设备即将关闭。
- 调用dev_deactivate以关闭出口队列规则，使得该设备再也无法用于传输，然后因为不再需要，停止看门狗定时器。
- 清除 dev->state 中的 __LINK_STATE_START 标识，把设备标识为关闭。
- 如果一个轮询动作被调度，以读取设备上的入口封包，就要等待该动作完成。因为__LINK_STATE_START标识已被清除，该设备上已不能再为其他接收的轮询动作进行调度了，但是在该标识清除前可能有一个轮询动作未决。参见第十章了解接收的轮询的更多细节。
- 如果有定义，调用dev->stop。并非所有设备驱动程序都初始化此函数。
- 清除 dev->flags 中的 IFF_UP 标识，把设备标识为关闭。
- 传送NETDEV_DOWN通知链给netdev_chain通知链，以通知感兴趣的内核组件该设备现在已经关闭。

更新设备队列规则状态

从“队列规则状态”一节可知，dev->state中可设置一些标识，以定义设备队列规则状态。本节中，我们要说明如何用其中的两个标识处理电源管理和链接状态变更。

与电源管理之间的交互

当内核支持电源管理时，只要系统进入挂起模式或者重新继续，NIC设备驱动程序就可接到通知。由第六章“PCI NIC 驱动程序注册范例”一节已知，pci_driver 结构的suspend和resume函数指针如何根据内核是否支持电源管理而进行初始化。例如，以下是 *drivers/net/3c59x.c* 设备驱动程序初始化其 pci_driver 实例的方式：

```
static struct pci_driver vortex_driver = {
     .name          "3c59x",
    .probe       vortex_init_one,
     .remove          _ _devexit_p(vortex_remove_one),
     .id_table     vortex_pci_tbl,
#ifdef CONFIG_PM
    .suspend    vortex_suspend,
     .resume          vortex_resume,
#endif
};
```

当系统进入挂起模式时，就会执行设备驱动程序所提供的 suspend 函数，让驱动程序

据此采取动作。电源管理状态变更不会影响注册状态dev->reg_state，但是设备状态dev->state必须变更。

挂起设备

当设备挂起时，举例言之，其设备驱动程序会调用pci_driver为PCI设备准备的suspend函数，以处理此事件。除了驱动程序特定行动外，每个设备驱动程序还必须执行其他一些动作：

- 清除dev->state中的__LINK_STATE_PRESENT标识，因为该设备暂时不能操作。
- 如果设备已开启，就用netif_stop_queue（注14）关闭其出口队列，以防止该设备再被用于传输其他任何封包。注意，已注册的设备不一定必须开启：当设备能被识别时，将由内核为该设备分派设备驱动程序，因而得到注册。然而，除非用户配置明确的予以请求，否则该设备不会被开启（因此也就不能用）。

这些任务是由netif_device_detach实现的：

```
static inline void netif_device_detach(struct net_device *dev)
{
    if (test_and_clear_bit(_ _LINK_STATE_PRESENT, &dev->state) &&
        netif_running(dev)) {
        netif_stop_queue(dev);
    }
}
```

使设备重新继续

当设备重新继续时，举例言之，其设备驱动程序会调用pci_driver为PCI设备准备的resume函数以处理此事件：

- 设置dev->state中的__LINK_STATE_PRESENT标识，因为该设备现在又可用了。
- 如果该设备在挂起前已开启，就用netif_wake_queue重新开启其出口队列，然后重启流量控制所用的看门狗定时器（参见第十一章“看门狗定时器”一节）。

这些任务是由netif_device_attach实现的：

```
static inline void netif_device_attach(struct net_device *dev)
{
    if (!test_and_set_bit(_ _LINK_STATE_PRESENT, &dev->state) &&
```

注14： 参见第十一章有关用于启动、停止以及重启出口队列的函数的更多细节。

```
            netif_running(dev)) {
            netif_wake_queue(dev);
            _ _netdev_watchdog_up(dev);
        }
    }
```

链接状态变更侦测

当NIC设备驱动程序侦测载波或信号是否存在时，也许是由NIC通知，或者对NIC读取配置寄存器以明确做检查，可以分别利用netif_carrier_on和netif_carrier_off通知内核。当载波状态有变时，就可以调用这些函数；因此，当这些函数的启用不恰当时，就什么也不会做。

以下是可能导致链接状态变更的一些常见情况：

- 电缆线插入NIC，或者从NIC中拔除。
- 电缆线另一端的设备电源关掉或关闭了。这类设备有Hub、桥接器、路由器以及PC NIC等。

当设备驱动程序侦测到其设备之一上有载波而调用netif_carrier_on时，此函数会：

- 清除dev->state中的__LINK_STATE_NOCARRIER标识。
- 产生一个链接状态变更事件，并将其交付给linkwatch_fire_event进行处理。参见“调度并处理链接状态变更事件”一节。
- 如果设备已开启，就启动看门狗定时器。此定时器由流量控制使用，以侦测传输是否失败因而受阻（定时器到期）。参见第十一章“看门狗定时器”一节。

```
static inline netif_carrier_on(struct net_device *dev)
{
    if (test_and_clear_bit(__LINK_STATE_NOCARRIER, &dev->state))
        linkwatch_fire_event(dev);
  if (netif_running(dev)
        __netdev_watchdog_up(dev);
}
```

当设备驱动程序侦测到其设备之一上遗失载波而调用netif_carrier_off时，此函数会：

- 设置dev->state中的__LINK_STATE_NOCARRIER标识。
- 产生一个链接状态变更事件，并将其交付给linkwatch_fire_event进行处理。参见“调度并处理链接状态变更事件”一节。

注意，这两个函数都会产生一个链接状态变更事件，然后交付给 linkwatch_fire_event 进行处理，对此在下一节描述。

```
static inline netif_carrier_off(struct net_device *dev)
{
    if (!test_and_set_bit(_ _LINK_STATE_NOCARRIER, &dev->state))
        linkwatch_fire_event(dev);
}
```

调度并处理链接状态变更事件

链接状态变更事件用 lw_event 结构定义。此结构很简单：只包含一个指向相关联的 net_device结构的指针，以及另一个用于把此结构链接至全局列表 lweventlist 的字段，该全局列表内含未决的链接状态变更事件。

注意，lw_event 结构并没有包括任何区分侦测到载波和载波遗失这两种情况的参数，这是因为不需要区分。内核所需知道的就是链接状态有了变更，所以，拥有对该设备的引用就已足够。对任何设备而言，lweventlist列表中都不会有一个以上的lw_event 实例。因为没必要记录历史或记下前前后后的变更：该链接不是能操作，就是不能操作，因此链接状态不是开就是关。两次状态变更等于没变更，三次变更等于一次变更。以此类推，所以，当设备已有一个未决链接状态变更事件时，新事件就不需要排入队列。此情况的侦测可以通过检查dev->state中的__LINK_STATE_ LINKWATCH_PENDING 标识进行（如图 8-7 中的流程图所示）。

一旦 lw_event数据结构得到初始化，取得指向正确net_device实例的引用，而且也已添加到lweventlist列表，此外，dev->state中的__LINK_STATE_LINKWATCH_PENDING 标识也已设置，则linkwatch_fire_event就必须启用能实际处理 lweventlist 列表元素的函数。函数 linkwatch_event 并非是直接调用，而是交付请求给 keventd_wq 内核线程调度执行：有一个 work_struct 数据结构会初始化为指向 linkwatch_event 函数的引用，然后交付给 keventd_wq。

为了避免处理函数linkwatch_event太过频繁地执行，其执行频率会限制在每秒一次。

linkwatch_event用linkwatch_run_queue在rtnl锁的保护下处理 lweventlist列表的元素，在“上锁”一节已给出了描述。处理 lw_event 实例仅仅包括如下内容：

- 清除 dev->state 上的 __LINK_STATE_LINKWATCH_PENDING 标识。
- 传送一个 NETDEV_CHANGE 通知信息到 netdev_chain 通知链上。
- 传送 RTM_NEWLINK 通知信息到 RTMGRP_LINK RTnetlink 群组。参见“RTnetlink 链接通知”一节。

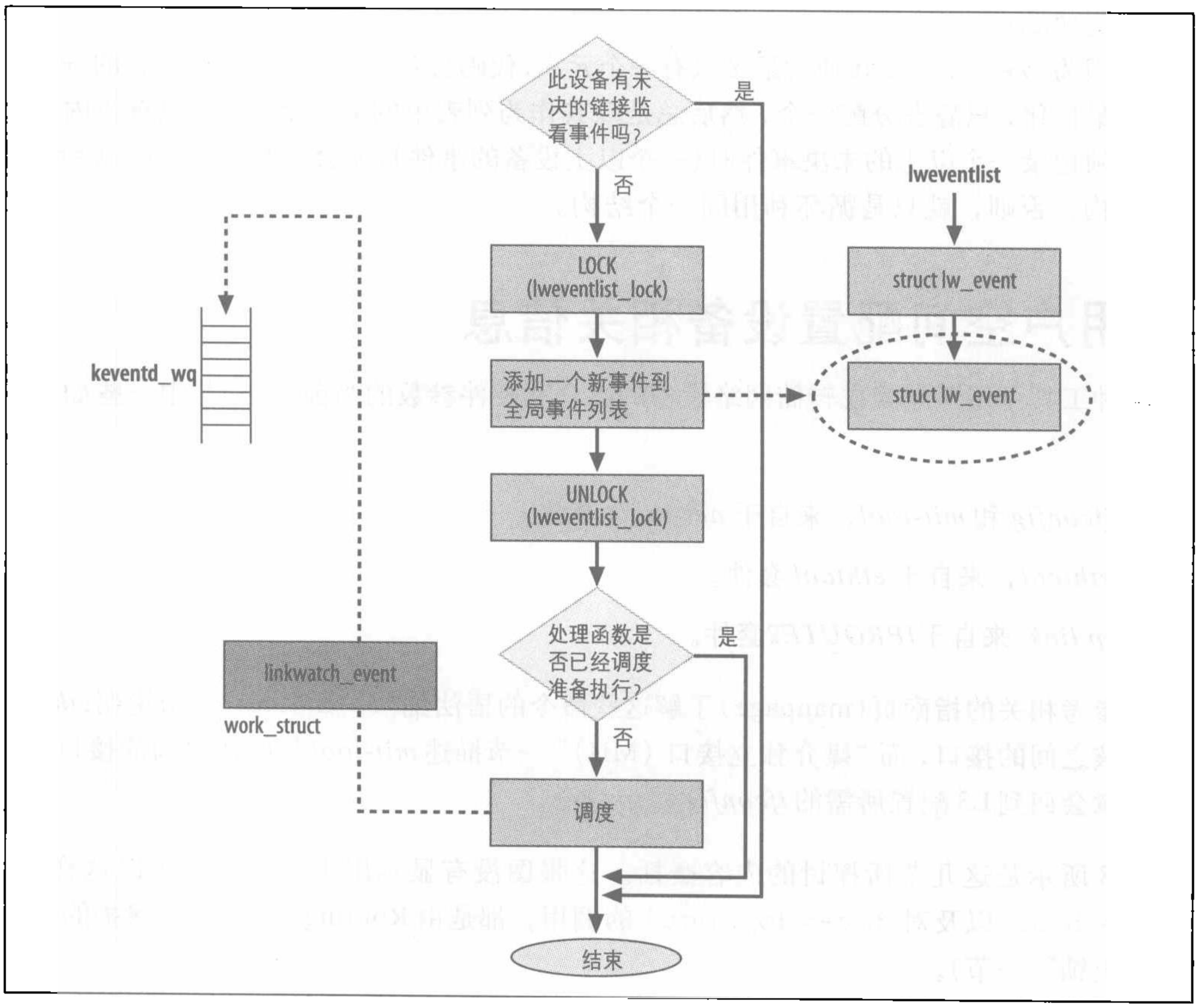


图 8-7：linkwatch_fire_event 函数

这两条通知信息用 netdev_state_change 传送，但是，只有当设备开启时才会传送 (dev->flags & IFF_UP)：没人关心已关闭设备上的链接状态变更。

链接监看标识

net/core/linkwatch.c 中的代码定义了两个标识，可以在全局变量 linkwatch_flags 中设置：

LW_RUNNING

当此标识设置时，linkwatch_event 已进入执行调度。此标识由 linkwatch_event 本身清除。

LW_SE_USED

因为lweventlist通常最多只有一个元素，代码会对lw_event数据结构的分配做最优化，只静态分配一个，然后总是以其作为列表中的第一个元素。只有当内核必须记录一个以上的未决事件时(一个以上设备的事件)，才会分配其他lw_event结构，否则，就只是循环利用同一个结构。

从用户空间配置设备相关信息

有多种工具可用于配置或转储网络设备的媒介和硬件参数的当前状态。其中一些如下所示：

- *ifconfig* 和 *mii-tool*，来自于 *net-tools* 套件。
- *ethtool*，来自于 *ethtool* 套件。
- *ip link*，来自于*IPROUTE2*套件。

可以参考相关的指南页(manpage)了解这些命令的语法细节。“Ethtool”一节说明*ethtool*与内核之间的接口，而“媒介独立接口 (MII)”一节描述*mii-tool*与内核之间的接口。稍后几章会回到L3配置所需的 *ifconfig* 和 *ip* 命令。

图 8-8 所示是这几节所探讨的内容概括。这张图没有显示出上锁细节。可以这样说，dev_ethtool以及对 dev->do_ioctl 的调用，都是由 Routing Netlink 锁保护的（参见“上锁”一节)。

Ethtool

此节对 *ethtool* 提出一个概括说明，并说明其与 *mii-tool* 以及net_device中的do_ioctl 函数指针之间的关系。

net_device数据结构中有一个指向类型为ethtool_ops的VFT的指针。ethtool_ops结构是一组函数指针，可用于读取和初始化net_device结构上的许多参数，或者用以触发一种行为（也就是重启自动协商)。

目前，并非所有设备驱动程序都支持此功能；但那些支持此功能的驱动程序不一定会支持所有函数。一般而言，dev->ethtool_ops的初始化都是在本章开头所介绍的probe函数中完成的。

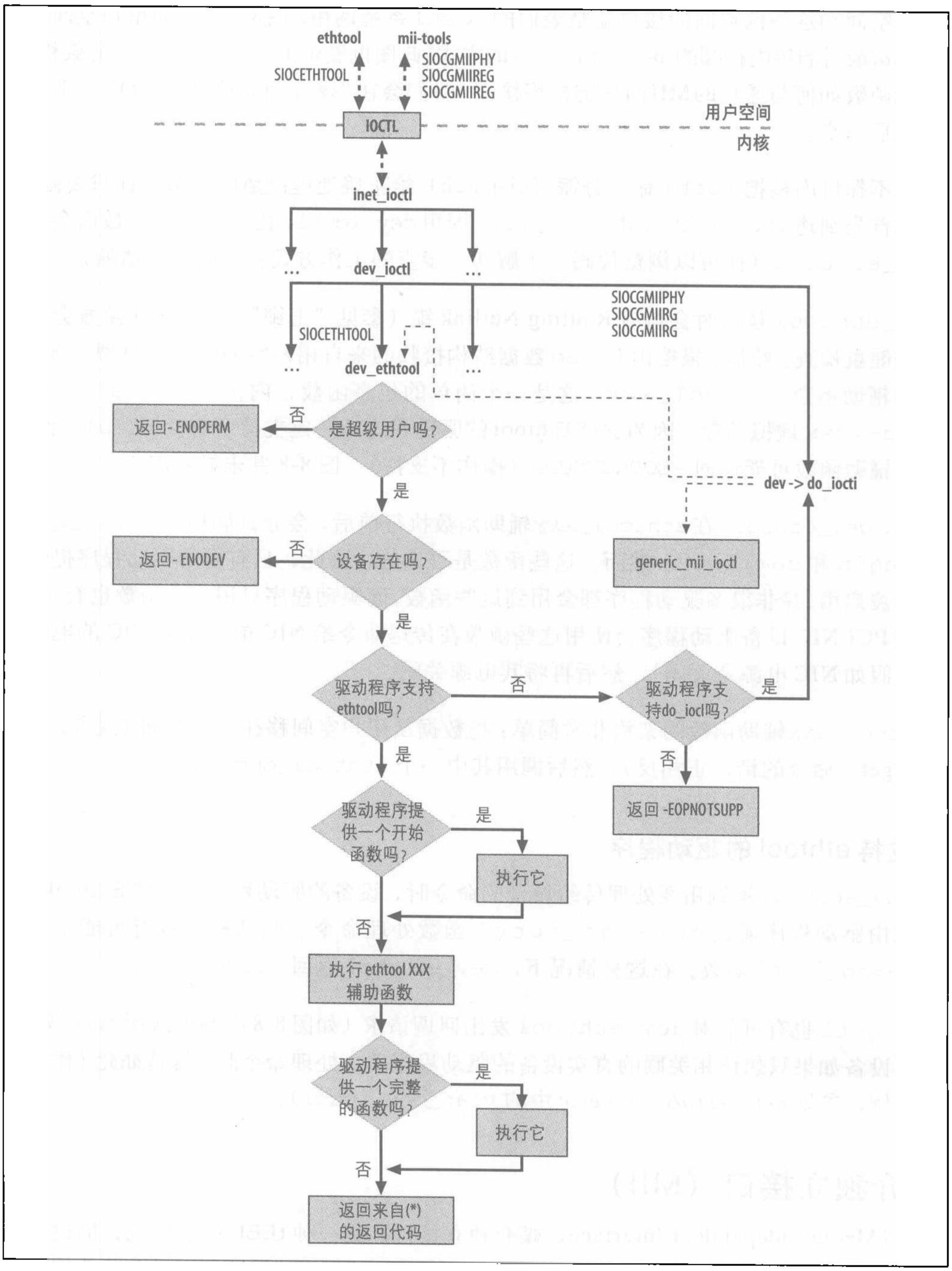


图 8-8：设备配置的 ioctl 接口

用户空间和这些函数间的接口就是老旧的ioctl系统调用。图8-8所示为用户空间命令*ethtool*最后启用内核端的dev_ethtool的方式。此图也显示出dev_ethtool的架构，以及此函数如何与通用的MII内核链接库接口。我们会在“媒介独立接口（MII）”一节说明这最后一点。

我们不探讨内核把ioctl命令分派（dispatch）给正确处理函数的细节，你只要知道，请求首先到达inet_ioctl，由inet_ioctl调用dev_ioctl，由dev_ioctl最后会调用dev_ethtool（你可以浏览代码，了解其一步步的工作方式；代码相当清晰）。

dev_ethtool执行时会持有Routing Netlink锁（参见“上锁”一节）。此函数会先做一些健康检查。然后，根据由ifreq数据结构接收的来自用户空间的命令类型，调用正确的辅助函数ethtool_*xxx*。这是一个简单的包裹函数，内含dev->ethtool_ops->*xxx*虚拟函数。因为支持Ethtool的驱动程序不一定支持所有ethtool_ops函数，辅助函数可能返回-EOPNOTSUPP（操作不支持）。图8-8并未显示出来。

注意，dev_ethtool在ethtool_*xxx*辅助函数执行前后，会分别调用ethtool_ops函数begin和complete。然而，这些函数是可选的，因此，只有设备驱动程序提供时才会被启用。并非很多驱动程序都会用到这些函数，而驱动程序只用一个函数也有可能。有些PCI NIC设备驱动程序会使用这些函数在传送命令给NIC前，先把NIC的电源打开（假如NIC电源是关闭），然后再将其电源关闭。

ethtool_*xxx*辅助函数的架构非常简单：把数据从用户空间移往内核空间（或者，如果是“get”命令的话，正相反），然后调用其中一个ethtool_ops函数。

不支持ethtool的驱动程序

当dev_ethtool被调用要处理传给设备的命令时，设备的驱动程序不支持Ethtool，就尝试由驱动程序通过dev->do_ioctl函数处理命令。驱动程序也有可能不支持dev->do_ioctl函数。在这种情况下，dev_ethtool返回-EOPNOTSUPP。

do_ioctl也有可能对dev_ethtool发出回调请求（如图8-8中的虚线所示）：例如，虚拟设备如果只想让相关联的真实设备的驱动程序负责处理命令时，就是如此（作为一个范例，参见*net/8021q/vlan_dev.c*中的vlan_dev_ioctl）。

媒介独立接口（MII）

MII（Media Independent Interface，媒介独立接口）是一种IEEE标准规范，描述网络控制芯片和实例媒介芯片之间的接口。例如，有了这种接口，用户可以开启、关闭以及配置自动协商。但并非所有NIC都支持。

Linux最常用的与MII交互的工具就是*mii-tools*。如同*ethtool*一样，也是通过ioctl与内核交互，如图8-8中所示。内核提供一组ioctl命令以处理MII。这些命令主要是读写特定的NIC寄存器。

如图8-8所示，ioctl命令传给驱动程序所提供的dev->do_ioctl函数。此函数可以采用下面两种方式之一予以处理：

- 只识别MII的三个ioctl命令，然后用设备驱动代码予以处理，这是最常见的情况。
- 依赖内核MII链接库*drivers/net/mii.c*，用generic_mii_ioctl处理输入命令。

也有可能，尤其是对虚拟设备而言，用dev->do_ioctl函数去识别和处理MII以外的其他命令。

下列是dev->do_ioclt函数，针对那些依赖内核MII链接库，而且不实现特殊命令的驱动程序的通用模型：

```
if (!netif_running(dev)) {
    return -EINVAL;
}
<lock private data structure>
err = generic_mii_ioctl(...);
<unlock private data structure>
return err;
```

注意，在图8-8中，*ethtool*命令最后可能会调用来自MII内核链接库的函数（例如，重启自动协商）。

虚拟设备

在第五章的“虚拟设备”一节中我们已经知道，就初始化而言虚拟设备和真实设备之间的差异所在。就注册而言，虚拟设备也必须像真实设备那样注册以及开启才能予以使用。然而，还是有些差别：

- 虚拟设备偶尔会调用register_netdevice和unregister_netdevice而不是其包裹函数，而且自行负责上锁。虚拟设备可能需要处理上锁，以便于持有该锁的时间能比真实设备更长。采用这种方法，此锁可能因为保护另一段代码（除了register_netdev以外；但该段程序可以用其他方式保护）而遭到误用，使得持有的时间超过所需的时间。
- 真实设备不能以用户命令除名（也就是销毁），只能被关闭。真实设备会在其驱动程序卸载（当然，是以模块形式加载时）时除名。相反，虚拟设备可以通过用户命令创建和除名，至于是否可行，依赖虚拟设备驱动程序的设计。

在“register_netdevice函数”和”设备除名”两节中我们已经知道，虚拟设备和多数真实设备在使用dev->init、dev->uninit以及dev->destructor时有所不同。因为多数虚拟设备会在真实设备上实现某种或多或少有点复杂的逻辑，因此，会使用dev->init和dev->uninit负责额外的初始化和清理工作。dev->destructor通常会初始化为free_netdev（如图8-4所示），使得驱动程序不需要在除名后明确地调用free_netdev。

在“设备初始化”一节中已经了解，net_device结构的初始化工作如何在设备驱动程序的probe函数和一些通用设置函数之间分离。因为虚拟设备没有probe函数，表8-2和表8-3的分类法就不适用了。

虚拟设备驱动程序会注册到在“设备注册状态通知”一节所描述的netdev_chain通知链上，因为多数虚拟设备都是定义在真实设备之上，所以对真实设备的变更也会影响虚拟设备。我们看两个范例：

Bonding

Bonding是虚拟设备，允许你绑定一组接口，使其看起来像单一接口。流量可以通过各种算法而分散在这组接口之间，其中一种是简单循环法(simple round robin)。我们以图8-9（a）为例。当*eth0*宕掉时，绑定接口*bond0*在真实设备间分配流量时必须知道此事，把这件事考虑进来。假设*eth1*也宕掉了，*bond0*就得关闭，因为没有任何可工作的真实设备存在。

VLAN界面

Linux支持802.1Q协议，允许定义VLAN（Virtual LAN）接口。考虑图8-9（b）中的范例，用户已在*eth0*上定义了两个VLAN接口。当*eth0*宕掉时，所有虚拟(VLAN)接口也必须宕掉。

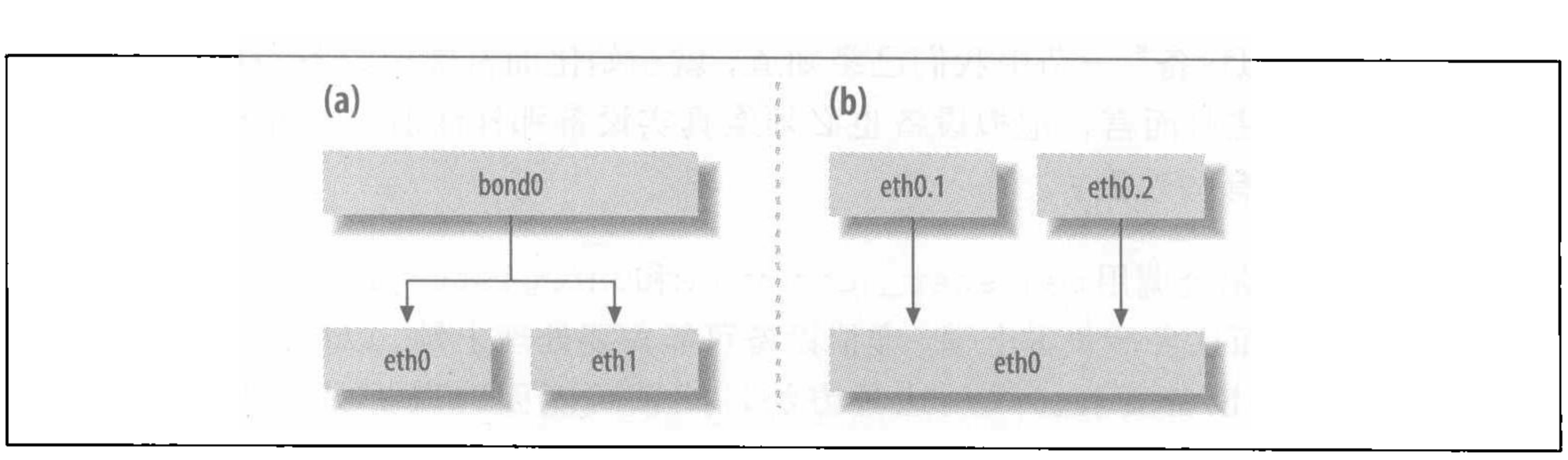


图8-9：（a）Bonding接口（a）VLAN接口

上锁

由“net_device结构的组织”一节可知，dev_base列表以及dev_name_head和dev_name_index两张hash表由dev_base_list锁保护。然而，该锁只用于对列表和hash表的访问予以串行化，而不是对net_device数据结构内容的变更予以串行化。net_device内容的变更是由Routing Netlink信号量（rtnl_sem）负责；而此锁的取得和释放分别通过rtnl_lock和rtnl_unlock（注15）完成。此信号量可用于net_device实例变更的串行化，而这些变更来自于：

运行期间事件

例如，当链接状态变更时（例如，网络电缆插入或拔除），内核也必需通过修改dev->flags改变设备状态。

配置变更

当用户应用来自*net-tools*套件的*ifconfig*和*route*命令，或者来自*IPROUTE2*套件的*ip*命令改变配置时，内核会通过*ioctl*命令和Netlink套接字接收到通知。由这些接口启用的函数都必须使用锁。

net_device数据结构包括一些可用于上锁的字段，其中一些如下所示：

ingress_lock

queue_lock

当处理入口和出口流量调度时，分别由流量控制使用。

xmit_lock

xmit_lock_owner

用于同步设备驱动程序hard_start_xmit函数的访问。有关这些锁的细节，请参考第十一章。

通过/proc文件系统调整

*/proc*里没有可用于调整设备注册和除名任务的文件。

本章涉及的函数和变量

表8-4总结了本章所介绍的函数、数据结构以及变量。

注15： 也可用其他函数取得和释放该信号量。参见*include/linux/rtnetlink.h*的细节。

表 8-4：本章所介绍的函数、数据结构以及变量

名称	描述
函数	
alloc_netdev alloc_xxxdev wrappers	分配一个 net_device 结构并做部分初始化
free_netdev	释放一个 net_device 结构
dev_alloc_name	完成设备名称
register_netdevice、 register_netdev unregister_netdevice、 unregister_netdev	注册和除名网络设备。xxx_netdev API是xxx_netdevice API的包裹函数
xxx_setup	用于对net_device结构的一部分做初始化的辅助函数。最常见的接口类型中，每种都有一个函数
dev_hold dev_put	递增和递减对net_device结构的引用计数
netif_carrier_on netif_carrier_off netif_carrier_ok	当设备上的载波被侦测到、遗失或者要被读取时，就会调用这些函数
netif_device_attach netif_device_detach	当设备插入系统或者从系统拔除时，就会调用这些函数。此外，当系统进入挂起模式然后重新继续时，也会调用这些函数
netif_start_queue netif_stop_queue netif_queue_stopped	对设备出口队列予以启动、停止以及检查其状态时，就会调用这些函数
dev_ethtool	处理来自于 *ethtool* 用户空间命令的 ioctl 命令
变量	
dev_base dev_name_head dev_index_head dev_base_lock	dev_base是一份已注册网络设备列表。dev_xxx_head是两份存储 net_device 结构的以设备名称和 ID 为索引值的 hash 表。前三个结构是由 dev_base_lock 锁保护
lweventlist lweventlist_lock	lweventlist是列表，内含一些未决的lw_event事件。此列表是由 lweventlist_lock 锁保护
数据结构	
lw_event	链接状态变更事件[173]

本章涉及的文件和目录

图 8-10 显示了本章所涉及的文件和目录在内核源码树中的位置。

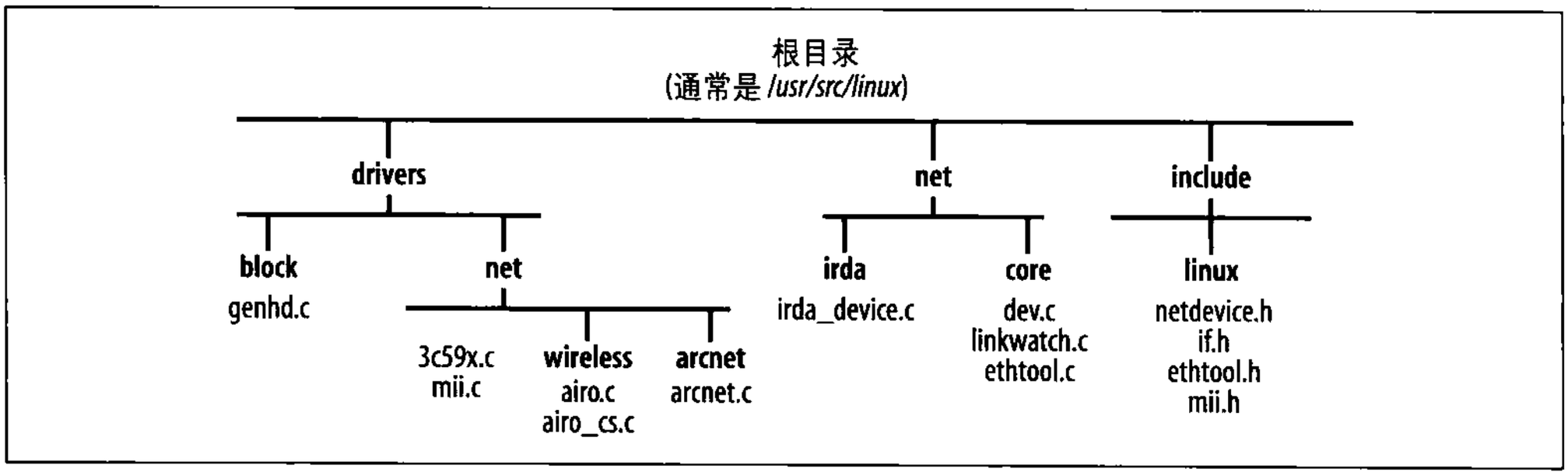


图 8-10：本章涉及的文件和目录

第三部分

传输和接收

这五章的目标是把内核内部会影响封包路径的所有功能都连接起来，让你对整个大蓝图有个概念。你会了解每个子系统在大蓝图中应该做些什么及何时出场。这几章不会涉及路由（本身就是很大一章的主题）或防火墙（不在本书讨论之列）。

就一般使用而言，术语传输通常是用于涉及任何方向的通信。但是，讨论内核时，传输只是指把帧往外传送而已，而接收是指帧传进来。在某些地方，我会以术语入口对应接收，而以出口对应传输。

转发的封包产生以及终止都是在远程系统，但是都会使用本地系统完成路由的建立，这就构成了另一类型，结合了接收和传输的元素。转发的有些方面将在第十章讨论；而完整的讨论则出现在第五部分和第七部分。

第一章已知术语帧、数据报及封包之间的区别。因为第三部分的这几章讨论 L2 和 L3 间的接口，术语帧和封包在多数情况下都是恰当的。虽然我多半使用术语帧，有时涉及数据单元但没提及特定分层时，我会使用封包。封包这个词在我们讨论的代码中最常见。

以下是第三部分各章的概述：

第九章 中断和网络驱动程序

本章会得到底层函数以及内核同步机制的概述。

第十章 帧的接收

本章会说明已接收的帧通过 L2 层的路径。

第十一章 帧的传输

第十一章所讲述的内容与第十章相同，不过，是针对已传输的帧（外出的帧）。

第十二章 中断事件一般性参考数据

本章是前几章的参考资料。

第十三章 协议处理函数

作为本部分的结论，本章将讨论入口帧如何交给正确的L3协议接收函数。

第九章

中断和网络驱动程序

前几章概述了如何初始化网络代码中的关键组件。本书接下来就是按照一个个的功能或一个个的子系统分析如何实现网络，为什么要引入某些功能，以及当功能之间有意义时彼此间是如何交互的。

本章先说明封包如何在L2层或驱动程序层，以及IP层或网络层（在本书第五部分说明）之间传送。我会引用第二章和第八章所介绍的许多数据结构，所以，需要时要回头去看这些章。

内核准备处理进出L2层的帧之前，必须先处理精妙而复杂的中断系统，设立中断系统，才可能每秒处理成千的帧。这就是本章主题。

其他会影响本章讨论内容的一般性议题如下：

- 当Linux内核编译成支持SMP而且在多处理器系统上运行时，接收和传输封包的代码就会完全利用该运算能力。所包含的数据结构在设计时就是以该目标为准。本章我们要谈SMP的一个方面，特别是新的软IRQ队列（softirq queue）和旧积压队列（backlog queue）之间的差别。
- 讨论入口路径时，我会论及多数网络驱动程序依然在使用的旧接口以及被称为NAPI的新接口，NAPI接口在中等和高等负载下可大幅提高性能。

本章会概述下半部函数（bottom half handler）以及内核同步机制。然而就细节讨论而言，可参考另外两本O'Reilly出版的书籍：《Understanding the Linux Kernel》和《Linux Device Drivers》（这本两书的中文版《深入理解Linux内核》的《Linux设备驱动程序》均已由中国电力出版社出版）。

决策和流量方向

封包经过网络协议栈所走的路径，会随封包是接收的、传输的还是转发的而不同（参见图 9-1 所示）。处理时的差异性也依赖于编译进内核的功能，以及这些功能是如何配置的而定。最后，所包括的设备也会造成差异，因为不同的设备支持不同的功能。

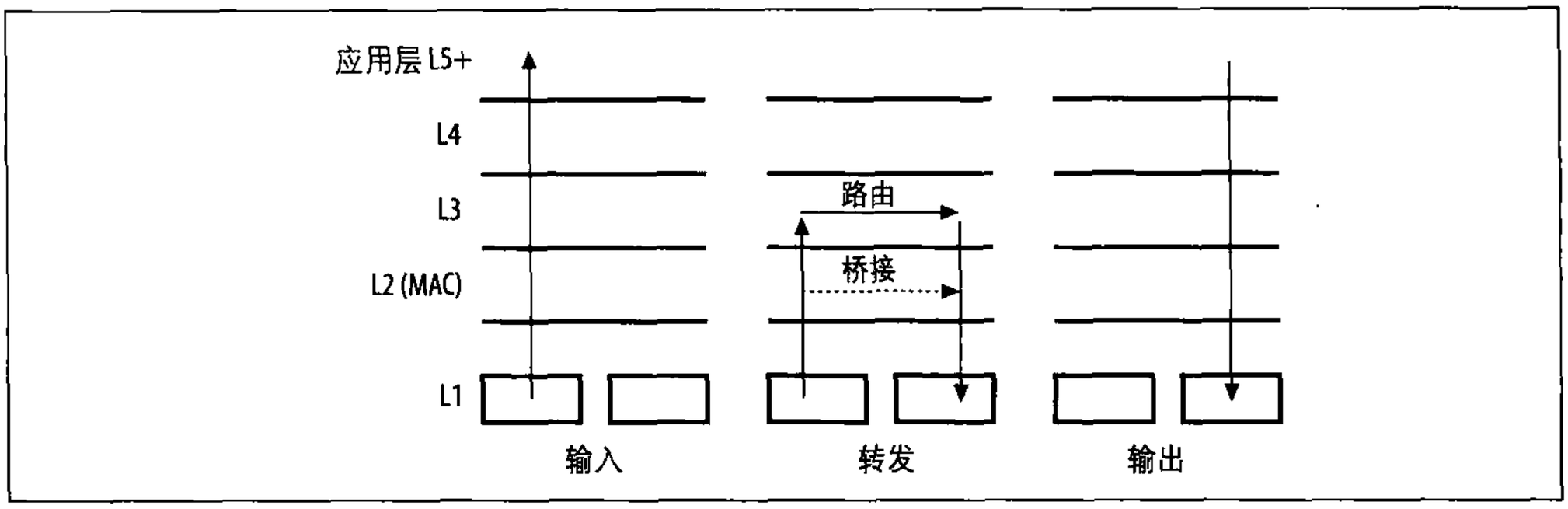


图 9-1：流量方向

虚拟设备，如我们熟悉的回环设备（lo），倾向于使用网络协议栈中的快捷方式。这些设备只是软件而已。例如，回环接口和任何硬件都无关，但是绑定接口则和一块或多块网络卡间接相关。因此，有些虚拟设备可以免除硬件上的某些限制（如 MTU），因而可以提高性能。

图 9-2 所示是一个理想的大蓝图，当然，相当简略。例如，没有显示导致丢弃帧的所有条件（注 1）。此图包括其他与输入路径有关的细节；你可以在本书第五部分、第六部分以及第七部分中找到出口路径的图示细节。本章接下来要讨论属于此图中一部分的所有链接。

接收到帧时通知驱动程序

第五章我曾提到设备与内核可以使用两种主要技术来交换数据：轮询和中断。我也曾说过，这两种技术的组合也是有效的选项。本节概述驱动程序用于通知内核接收到帧最常见的方法，以及每种方法的优缺点。有些方法依赖于设备上的特定功能是否可用（如特殊的定时器），而有些可能必须对驱动程序、操作系统或者两者做变更。

注 1：帧被丢弃的原因有很多：输入队列中没有内存、输出队列中没有内存（只针对转发的帧或已传输的帧）、找不到通往目的地的路径、防火墙策略、健康检查失败等等。

理论上，这里的讨论适用于任何设备类型，但是，把这些设备看成是产生高交互量（也就是帧的接收）的网络接口卡会比好。

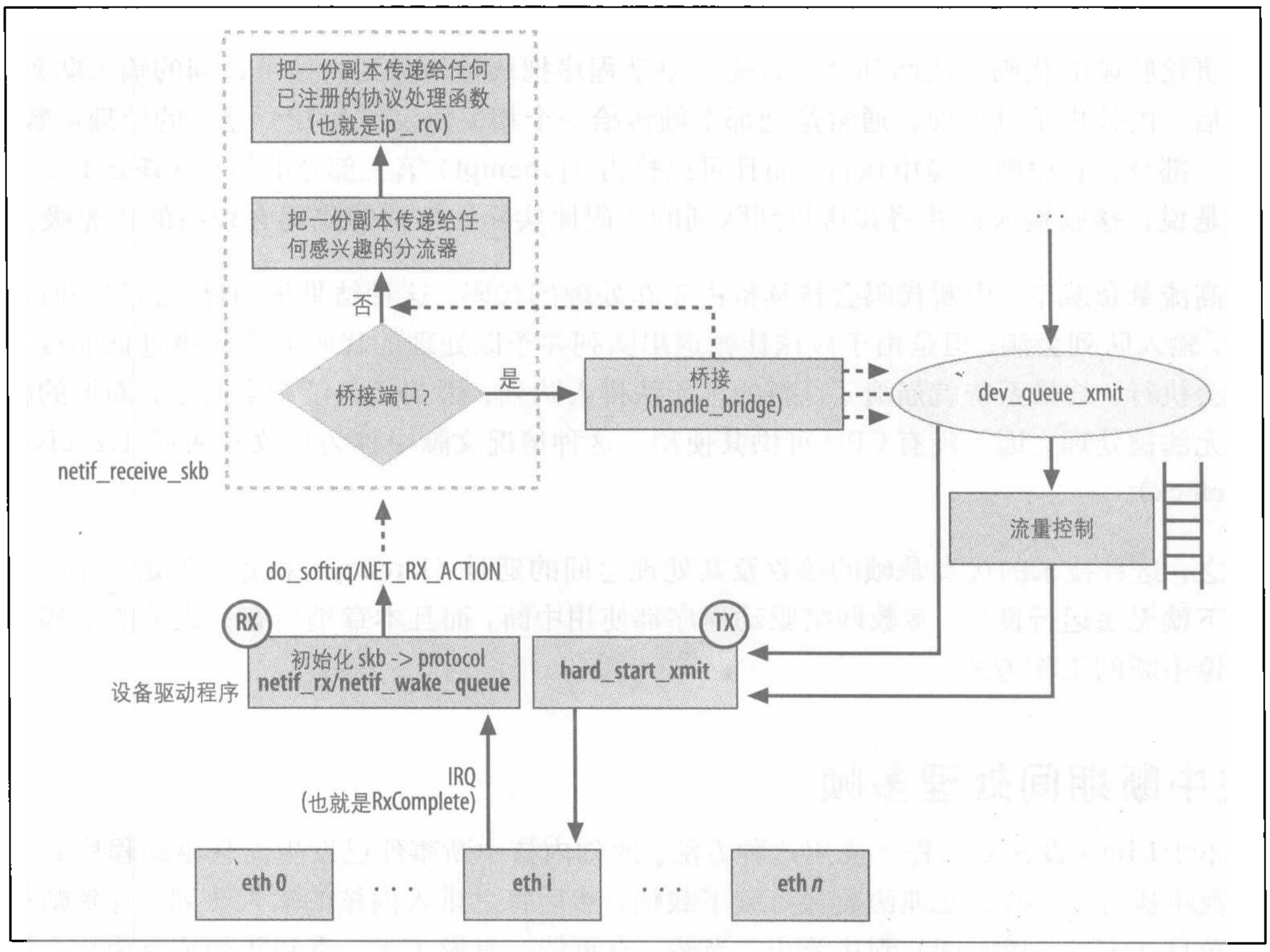


图 9-2：入口路径（帧接收）

轮询

通过这种技术，内核可不断持续检查设备是否有话要说。例如，内核可以持续读取设备上的一个内存寄存器，或者当一个定时器到期时就回头去检查那个内存寄存器。可以想象得到，这种做法会轻易浪费掉很多系统资源，因此，如果操作系统和设备可以使用其他技术（如中断），轮询就很少会被采用。不过在有些情况下，轮询是最佳方法。稍后我们会继续讨论这一点。

中断

此时，当特定事件发生时，设备驱动程序会代表内核指示设备产生硬件中断。内核将中断其他的活动，然后调用一个驱动程序所注册的处理函数，以满足设备的需要。当事件

是接收到一个帧时，处理函数就会把该帧排入队列某处，然后通知内核。这一项技术相当常见，依然是低流量负载下的最佳选择。遗憾的是，在高流量负载下就无法良好运行：每接收一个帧就强制产生中断，很快就会让CPU为处理中断事件浪费所有的时间。

负责接收帧的代码分成两部分：首先，驱动程序把该帧拷贝到内核可访问的输入队列，然后，内核再予以处理，通常是把那个帧传给一个相关协议（如IP）专用的处理函数。第一部分会在中断环境中执行，而且可以抢占（preempt）第二部分的执行（译注1）。也就是说，接收输入帧并将其拷贝到队列的代码比实际处理帧的代码有较高的优先级。

在高流量负载下，中断代码会持续抢占正在处理的代码。这种结果很明显：到某一时间点，输入队列会满，但是由于应该让帧退出队列并予以处理的代码的优先级过低而没有机会执行，结果系统就崩溃了。新的帧无法排入队列，因为已经没有空间了，而旧的帧又无法被处理，因为没有CPU可供其使用。这种情况文献中称为接收－活锁（receive-livelock）。

总之，这种技术的优点是帧的接收及其处理之间的延时（latency）很短，但是，在高负载下就无法运行良好。多数网络驱动程序都使用中断，而且本章稍后有一大节的篇幅将讨论中断的工作方式。

在中断期间处理多帧

有不少Linux设备驱动程序使用这种方法。通知内核中断事件已发生而且驱动程序处理函数也执行了，接着处理函数会持续下载帧，然后将之排入内核的输入队列，直到帧最大数目（或者一段时间）到达为止。当然，有可能一直做下去，直到队列清空为止，但是我们要记住，设备驱动程序应该扮演优良组件的角色。驱动程序必须与其他子系统共享CPU，与其他设备共享IRQ线。合于礼法的行为尤其重要，因为当驱动程序运行时中断功能会关闭。

存储限制也要适用，如同前节讲述的情况。每个设备的内存数量都有限，因此，其能存储的帧数目也有限。如果驱动程序不能以实时的方式处理帧，缓冲区就会填满，而新的帧（或旧的帧，依赖于驱动程序策略）就会被丢弃。如果承受负载的设备一直处理送进来的帧，直到其队列成空，那么饿死的情形就会发生在其他设备身上。

这种技术不需要对操作系统做任何变更，完全是在设备驱动程序内实现的。

这种方法有其他一些变种。驱动程序可以只关闭某个设备的中断事件（功能），该设备

译注1：也就是第一部分会先于第二部分执行，即第一部分的优先级高于第二部分的优先级。

的输入队列中有一些帧，然后把轮询驱动程序的队列的任务交给一个内核处理函数，不再是把所有中断事件都关闭，之后令驱动程序为内核把帧排入队列以便于处理。而这正是为什么Linux采用新接口 —— NAPI的原因。然而，与本节所述方法不同的是，NAPI需要对内核做变更。

定时器驱动的中断事件

这种技术是前几种技术的增强版。驱动程序会指示设备定期产生一个中断事件，不再是让设备以异步方式通知驱动程序有关帧的接收。然后，处理函数会检查自从上次中断事件以来是否有任何帧到达，然后一次处理所有的帧。最好是在驱动程序有话要说时才定期产生中断事件。

根据定时器粒度的不同（由设备在硬件内实现，不是内核定时器），设备所接收的帧会体现出不同等级的延时。例如，如果设备每 100 ms 产生一个中断事件，则一个帧的接收的通知信息会有平均为 50 ms 的延迟，而最大的延迟则为 100 ms。根据使用设备的网络联机上所运行的应用程序而定，这种延迟可能可以接受，也可能无法接收（注 2）。

驱动程序能用的粒度取决于该设备要提供什么，因为定时器是在硬件内实现的。目前，只有少数设备提供这种能力，所以Linux内核中的所有驱动程序不一定都能用这种解决方案。驱动程序可以把该设备的中断事件关闭，然后改用内核定时器，借此模拟这种能力。然而，驱动程序没有硬件的支持，处理定时器时CPU能花费的资源也不像设备那么多，所以，驱动程序也没办法让定时器的调度过于频繁。结果，绕到最后就变成轮询方法了。

组合

前几节说明的每种方法都有其优缺点。有些时候，有可能将它们组合起来得到更好的东西。我们说，在低负载下，纯中断模型可以保证低延时，但是在高负载下，其运行就很糟糕。另一方面，定时器驱动的中断事件在低负载下可能会引入过多的延时，而且浪费太多的CPU时间，但是在高负载下，可以大量减少CPU用量并解决接收－活锁问题。好的组合就是在低负载下使用中断技术，在高负载下切换至定时器驱动的中断事件。例如，Linux内核中的*tulip*驱动程序就可以这么做（参见*drivers/net/tulip/interrupt.c*，注3）。

注 2： 这里的讨论主要是针对Ethernet设备，因其使用的拥塞算法的缘故，传输时间（以及接收时间）已不再保证有上限。

注 3： 这不是简单的驱动程序，建议你先读完本部分的前三章。

范例

处理多个帧的平衡方法如下列代码片段所示（取自 *drivers/net/3c59x.c* Ethernet 驱动程序）。这段代码是vortex_interrupt中关键的几行代码，此函数是由驱动程序注册，作为处理来自于3Com的Vortex系列设备的中断事件的处理函数：

```
static irqreturn_t vortex_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    int work_done = max_interrupt_work;
    ioaddr = dev->base_addr;
    ... ... ...
    status = inw(ioaddr + EL3_STATUS);
    do {
        ... ... ...
        if (status & RxComplete)
            vortex_rx(dev);
        if (--work_done < 0) {
            /* 关闭所有未决的中断事件。*/
            ... ... ...
            /* 定时器会重新开启中断事件。*/
            mod_timer(&vp->timer, jiffies + 1*HZ);
            break;
        }
        ... ... ...
    } while ((status = inw(ioaddr + EL3_STATUS)) & (IntLatch | RxComplete));
    ... ... ...
}
```

其他遵循此模型的驱动程序也有类似的东西，可能把EL3_STATUS和RxComplete符号叫做其他名称，而且 *xxx*_rx 函数的实现可能也不同，但是其架构和此例所示非常接近。

在 vortex_interrupt 之中，驱动程序从设备读取发生中断事件的原因，然后存储至 status。网络设备产生一个中断事件的原因有很多种，而且好几种原因都可归结于单一中断事件。如果 RxComplete（此驱动程序特别定义的符号，意指已接收一个新的帧）是这些原因之一，程序就会启用 vortex_rx（注 4）。执行期间，该设备的中断事件会被关闭。然而，驱动程序可以读取卡上的一个硬件寄存器，同时寻找是否已有新的中断事件发布。已有新的中断事件发布时，IntLatch标识就是真（当驱动程序把该中断事件处理掉时，就会将此标识清除）。

vortex_interrupt 会持续处理送进来的帧，直到寄存器说有个中断事件处于未决中（IntLatch），而且是起因于接收一个帧（RxComplete）。这也是说，只有多次出现的 RxComplete 中断事件能以一次处理。其他类型的中断事件比较少见，就要等待。

注 4：vortex_rx 的输入参数是设备，因为设备驱动程序可以处理多个属于同一种设备类型（或系列）的设备实体。因此，当此函数启用时，必须知道其面对的是哪个设备。

最后，如果到达了可以处理的输入帧的最大数目时，循环就终止，存储在work_done中，这就是优良组件该做的事。驱动程序使用默认值32，但是可以在模块加载期间调整此值。

中断处理函数

本章讨论的帧的处理，多数都是为了响应来自于网络硬件的中断事件。中断事件触发的函数的调度是很复杂的主题，即使与网络间并没有特别利害关系，也还是值得做一些研究。因此，本节我们讨论不同网络驱动程序处理中断事件的各种方式，然后介绍下半部（bottom half，BH）和软中断（softirq）的概念。

第五章我们了解了设备驱动程序如何以IRQ编号注册其处理函数，但是，我们没说明硬件中断事件如何把帧的处理委托给软中断处理函数。这一节要说明与接收一个帧相关联的中断事件请求是如何处理的，然后再说明协议处理函数（第十三章讨论）在何处接收其封包。我们会说明硬IRQ（hardware IRQ）和软IRQ（software IRQ）之间的关系，以及为什么需要软IRQ。我们会简洁说明旧版本内核如何处理中断件事，然后比较旧方法和内核2.4版本引进的新方法。这些讨论内容会显示出新模型相对于旧模型所具有的优点，特别是在性能方面。

说明软IRQ之前，我们必须对下半部函数的概念做一点介绍。当然，我不会深入探讨，因为其他文献资源已做了非常详细的介绍，特别是《深入理解Linux内核》和《Linux设备驱动程序》。

下半部函数存在的原因

每当CPU接收一个中断通知信息时，就会调用与该中断事件相关联的处理函数，这种关联性由编号识别。在该处理函数执行期间，即内核程序处于中断环境（interrupt context）中，服务于该中断事件的CPU会被关闭其中断功能。也就是说，如果一个CPU忙于服务一个中断事件，就不能服务其他中断事件，无论是相同类型或是不同类型的中断事件（注5）。CPU也不能执行其他进程：CPU完全属于该中断处理函数，而且不能被抢占。

在最简单的情况下，这些是一个中断事件引起的主要事件：

1. 设备产生一个中断事件，而硬件通知内核。

注5：在第五章我们看到，声明为慢速处理函数的中断处理函数在本地CPU上执行时，中断功能是开启的。

2. 如果内核没有为另一个中断事件服务(而且如果中断功能没有因为其他原因而被关闭)，就会看到该通知信息。

3. 内核关闭本地CPU的中断功能，然后执行与所接收的中断事件类型相关联的处理函数。

4. 内核会离开该中断处理函数，然后重启本地CPU的中断功能。

简而言之，中断处理函数是非抢占的（nonpreemptible），而且是非可再进入的（non-reentrant)。当一个函数无法被另一个自身的调用而中断，该函数就定义为非可再进入；就中断处理函数而言，就是说其执行时中断功能会关闭。这种设计选择有助于降低竞争情况的可能性。然而，因为CPU能做的事如此有限，内核做非可抢占设计以及等待被CPU服务的进程，就会对性能有潜在的严重影响。

因此，中断处理函数所做的工作应该尽快完成。中断处理函数在中断环境期间所需的处理量依赖于事件的类型。例如，键盘可能只需在每次有一个按键被按下时传送一个中断事件，而处理此事只需很少的能力：处理函数只需把按键的编码存储于某处，而且最多每秒执行几次。在其他时间，处理一个中断事件所需要的动作并非琐碎之事，而且其执行需要花费很多CPU时间。例如，网络设备就有相当复杂的工作：必需分配一个缓冲区(sk_buff)，把接收到的数据拷贝进去，对缓冲区结构（protocol）内的一些参数做初始化以告知较高层协议的处理函数来自于驱动程序的数据是什么种类等等。

以下该由“下半部”处理程序的概念登场。尽管由一个中断事件所触发的动作需要用掉很多CPU时间，通常此动作的内容多数都可以等待。中断事件可以先对CPU抢占，这是因为如果操作系统让硬件等待太久，可能会遗失数据。对实时数据流而言，这显然是真的，但是对任何必须把送进来的数据存储在固定大小的缓冲区内的硬件而言，这也是真的。此外，如果硬件遗失数据，通常无法再找回来。

另一方面，如果内核或用户空间进程必须被延迟或被抢占，则没有数据会遗失（实时系统例外，因其必须使用完全不同的方式处理进程和中断事件)。根据这些考虑，现代中断处理函数就区分成上半部（top half）和下半部。上半部就是释放CPU之前必须执行的一切事物，以此保留数据；下半部则包含所有可以从容做完的一切事情。

你可以把下半部定义成执行特定函数的异步请求。一般而言，当想执行函数时，不用请求任何东西，只要调用就行了。当一个中断事件到达时，有很多事要做，不想立刻中断。因此，你可以把多数工作打包成一个函数，当成下半部函数。

与简单模型相比，下列模型允许内核把中断功能关闭的时间大幅减少：

1. 设备发出信号给CPU，通知有中断事件。

2. CPU会执行相关的上半部，关闭后续的中断事件通知信息，直到此处理函数完成其工作为止。
3. 一般而言，上半部会执行下列工作：
   a. 把内核稍后处理该中断事件的所有信息都保存在RAM某处。
   b. 在某处标记一个标识（或者使用另一种内核机制触发某事），以确保内核会知道该中断事件，而且会使用处理函数所保存的数据以完成事件的处理。
   c. 终止前，会重新开启本地CPU的中断事件通知信息功能。
4. 稍后某一时刻，当内核不用再做一些紧迫之事时，就会检查该中断处理函数所设的标识（指出有要求处理的数据存在），然后调用相关的下半部函数。内核也会清除该标识，稍后当中断处理函数再次设置该标识时，就能再认出来。

随着时间的流逝，Linux开发人员已尝试过各种类型的下半部，不同类型遵循不同规则。网络在新的产品开发上扮演了一个重要的角色，这是因为网络需要低延时，也就是帧接收和其传递之间的时间量要最小。与其他类型的设备相比，低延时对网络设备驱动程序的重要性更高，这是因为涉及接收和传输的任务数目很高。如稍早“中断”一节所述，让一大群帧集结等待处理会损失惨重。声卡是另一种需要快速响应的设备范例。

下半部解决方案

内核提供各种不同机制以实现各种下半部以及一般的延期性工作。这些机制的主要差别在于下列几点：

运行环境

内核把来自于其他内核代码和用户空间进程的中断事件区别看待。当下半部函数所执行的函数可以进入休眠时，就会受限于进程环境中的许可机制，有别于中断环境。

并发和上锁

当一种机制可以利用SMP时，就涉及如何强制（必要时）串行化（serialization），以及锁机制如何影响伸缩性（scalability）。

本章中，我们只讨论那些不需进程环境的机制，也就是软IRQ和微任务（tasklet）。下一节，我们会简述这些机制对于并发及上锁的意义。

当必须延期一个可能会休眠函数的执行时，必须使用一个专用的内核线程或工作队列（work queue）。工作队列只是一种队列，你可以把执行一个函数的请求排入队列，然后，一个内核线程就会去负责这件事。就此而言，此函数会在该内核线程的环境内执行，因

此，休眠是可以的。因为网络代码主要是使用软IRQ和微任务，所以我们不会谈工作队列。

并发和上锁

在说明网络驱动程序用于处理下半部的代码之前，我们需要对并发的背景做一点了解。并发指的是一些函数之间可以彼此干扰，也许是因为它们在不同的CPU中调度，或者是因为有一个函数被内核挂起，而去执行另一函数。相关的主题是那些锁以及中断功能的关闭（关于并发在《深入理解Linux内核》和《Linux设备驱动程序》两书中有详细讨论）。

本章要介绍三种不同类型的函数，分别用来处理中断事件、旧式下半部、软IRQ及微任务。这些都可用于为一个函数的执行进行调度，但是彼此间存在很大的差异。就并发而言，我们把差异性总结如下：

- 只有一个旧式下半部可以在任何时刻执行，无论CPU的数目有多少个（内核2.2版本）。
- 任何时刻，每个微任务只有一个实例可以执行。不同的微任务可以同时在不同的CPU中执行。也就是说，就所有微任务而言，不需要强制任何串行化之事，因为内核已经强制了：同一个微任务不能有多个实例同时执行。
- 任何时刻，一个CPU中每个软IRQ只能有一个实例可以执行。然而，相同的软IRQ可以在不同的CPU中同时执行。也就是说，对任何软IRQ而言，必须确保不同的CPU对共享数据的访问，都要采用正确的上锁机制。为了提高并行性（parallelization），软IRQ应该尽可能设计成只访问每个CPU的数据，以减少大量使用上锁机制的需求。

因此，这三种功能需要不同种类的上锁机制。允许并发程度越高，为了精确和性能起见，程序员在设计要执行的代码时就要越仔细。对于任何给定的环境而言，软IRQ或微任务是否代表最佳选择取决于上锁和并发的需求。多数情况下，微任务是应该选择的方法。但是，因为接收和传输这些网络任务需要及时的响应，因此，就这两种特别情况而言，软IRQ比较适合。本章稍后会说明网络代码如何使用软IRQ。

在某些情况下，程序员必须关闭硬件中断、软件中断，或者两者均关闭。要对这些环境做详细讨论，需要对SMP、Linux内核的抢占功能以及其他本书范围之外的材料有所了解。然而，要理解网络代码，必须知道用于开启和关闭中断事件功能的主要函数的意义。表9-1总结了本章所需的几个函数（可以在*kernel/softirq.c*、*include/asm-XXX/hardirq.h*、*include/asm-XXX/spinlock.h*以及*include/linux/spinlock.h*中找到很多这类函数）。其中一些函数会定义为全局的，但其他函数则依体系结构而定。

表 9-1：一些与软件中断及硬件中断相关的 API

函数 / 宏	描述
in_interrupt	如果CPU正在服务于一个硬件中断或软件中断，或者抢占功能是关闭的，in_interrupt就返回TRUE
in_softirq	如果 CPU 正在服务于一个软件中断，in_softirq就返回 TRUE
in_irq	如果 CPU 正在服务于一个硬件中断，in_irq 就返回 TRUE 在“抢占功能”一节中，通过图 9-3 的帮助，可以看到这三个函数是如何实现的
softirq_pending	如果 CPU（其 ID 当成输入自变量传递进来）至少有一个软 IRQ 在未决中（也就是调度准备执行），就返回 TRUE
local_softirq_pending	如果本地 CPU 至少有一个软 IRQ 在未决中，就返回 TRUE
__raise_softirq_irqoff	设置与输入的软IRQ类型相关联的标识，将该软IRQ标记为未决
raise_softirq_irqoff	这是一个内含 __raise_softirq_irqoff 的包裹函数，当 in_interrupt()返回 FALSE 时，也会唤醒 ksoftirqd
raise_softirq	这是一个内含 raise_softirq_irqoff 的包裹函数，而调用 raise_softirq_irqoff之前，会先关闭硬件中断，然后再恢复其原有状态
__local_bh_enable local_bh_enable local_bh_disable	__local_bh_enable会开启本地CPU的下半部（等于连同软 IRQ/ 微任务一起开启），此外，如果有任何软 IRQ 处在未决中，并且 in_interrupt()返回 FALSE，则 local_bh_enable也会启用 invoke_softirq
local_bh_disable	关闭本地 CPU 的下半部
local_irq_disable	
local_irq_enable	关闭和开启本地 CPU 的中断功能
local_irq_save local_irq_restore	local_irq_save会先把本地CPU的中断状态保存起来，然后再予以关闭 local_irq_restore 会恢副本地 CPU 的中断状态（源于 local_irq_save 先前所保存的信息）
spin_lock_bh spin_unlock_bh	分别为取得和释放回转锁（spin lock）。这两个函数在操作过程中会关闭和重新开启下半部以及抢占功能

抢占功能

在分时系统中，内核总是能按其意愿抢占用户进程，但是内核本身通常是非可抢占的，也就是说，一旦开始运行，就不能被中断，除非已准备好放弃控制权。非可抢占内核有时会阻碍已准备好可以执行的高优先级进程，因为内核正在为一个低优先级进程执行一

个系统调用。为了支持实时扩展功能以及其他原因，Linux在2.5版本内核开发周期中终于实现了完全可抢占（preemptible）。有了这个新的内核功能，系统调用和其他内核任务就可以被其他较高优先级的内核任务给抢占掉。

为了支持SMP上锁机制，必须从内核中把那些关键节区（critical section）消除掉（非可抢占代码），在这方面已经做了很多工作，因此，加上完整的抢占功能并没有对内核造成重大变革。一旦加入抢占功能，开发人员只需明确定义在何处予以关闭（硬、软件中断代码中，调度器本身的内部，以及由回转锁和读/写锁所保护的代码中等等）。

然而，有些时候抢占功能必须予以关闭，正如中断功能一样。本节中，我只讨论几个与抢占功能有关的函数，当你在浏览代码时，可以仔细地去看。此外，我会简述为了处理抢占功能，有些上锁宏做了哪些更新。

下列几个函数可控制抢占功能：

`preempt_disable`

为当前任务关闭抢占功能。可以重复调用，递增一个引用计数器。

`preempt_enable`

`preempt_enable_no_resched`

与`preempt_disable`相反，可使抢占功能再度开启。`preempt_enable_no_resched`只是递减一个引用计数器，使得当其值到达零时，可以让抢占功能再度开启。此外，`preempt_enable`会检查计数器是否为零，然后强制调用`schedule()`，让任何较高优先级的任务去执行。

`preempt_check_resched`

此函数由`preempt_enable`调用，与`preempt_enable_no_resched`不同。

网络代码没有直接面对这些函数。然而，`preempt_enable`和`preempt_disable`会被间接调用，例如由上锁原语（locking primitive）间接调用，如`rcu_read_lock`、`rcu_read_unlock`、`spin_lock`以及`spin_unlock`等等。用于访问各个CPU数据结构的函数也会在读取数据之前关闭抢占功能，如`get_cpu`和`get_cpu_var`。

每个进程都有一个计数器，名为`preempt_count`，内嵌于`thread_info`结构内，指出特定的进程是否容许抢占功能。此字段可以通过`preempt_count()`读取，而且会通过定义在*include/linux/preempt.h*中的`inc_preempt_count`和`dec_preempt_count`函数间接操作。有些情况下，内核是不应该被抢占的，如正在服务于硬件，以及其所用的一些函数调用之一显示出抢占功能已被关闭。因此，`preempt_count`被分成三个组件，每个字节都是一个计数器，各自针对一种需要非抢占功能的情况：硬件中断、软件中断以及一般非抢占情况。`preempt_count`的配置如图9-3所示。

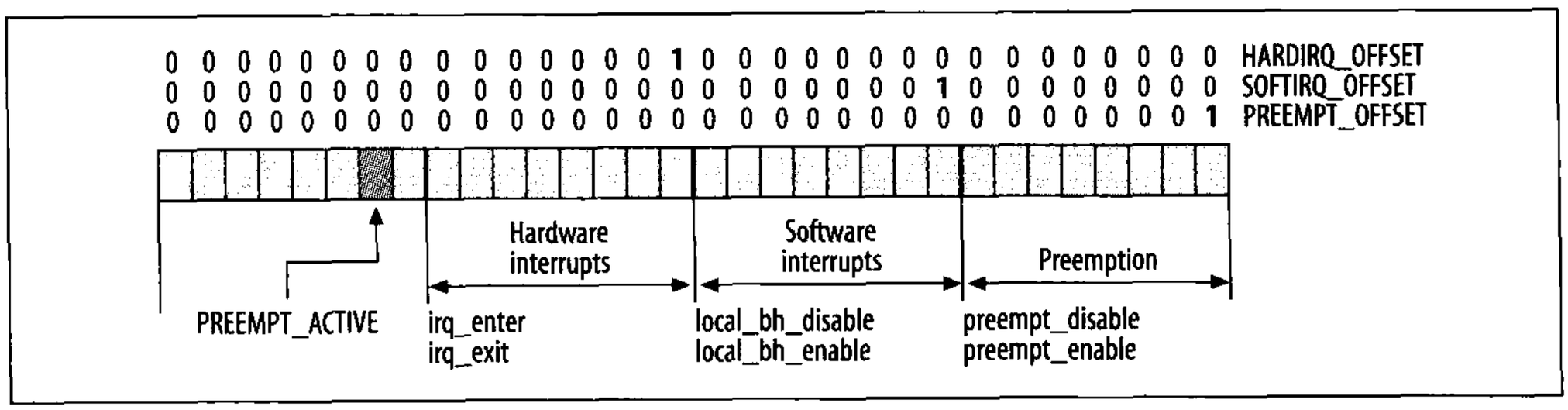


图 9-3：preempt_count 的结构

此图除了显示出每个字节的用途外，也显示出操作该字节的主要函数。此时，并没有完全用到高位字节，但是，调用schedule函数并通知该函数它已被调用而抢占了当前任务之前，其第二最低位会被设置（注 6）。在 *include/asm-xxx/hardirq.h* 中有几个宏，可让你轻易读写preempt_counter，其中一些包括图9-3所示的XXX_OFFSET变量，而且图中所列的函数会予以使用，以递增或递减正确的字节。

尽管如此复杂，每当必须对当前进程做检查以了解其是否可被抢占时，内核所必须知道的就是preempt_count是否为NULL(抢占功能为何被关闭其实不重要)。

下半部函数

下半部的基础架构必须解决下列需求：

- 把下半部分类成适当类型。
- 注册下半部类型及其处理函数间的关联关系。
- 为下半部函数调度，以准备执行。
- 通知内核有已调度的BH存在。

首先，我们看内核版本到 2.2 以来是如何处理下半部函数的，然后再说明内核 2.4 和 2.6 版本如何使用软IRQ处理下半部函数。

内核 2.2 版本的下半部函数

2.2 版本模型把下半部函数划分为一大群类型，差别在于内核何时以及以何种频度予以检查和执行。2.2 版本的列表如下所示，取自 *include/linux/interrupt.h*。本书中，我们最感兴趣的是 NET_BH。

注 6：　PREEMPT_ACTIVE 标识的定义是随体系结构而不同的。此图所示为最常见的定义。

```
enum {
        TIMER_BH = 0,
        CONSOLE_BH,
        TQUEUE_BH,
        DIGI_BH,
        SERIAL_BH,
        RISCOM8_BH,
        SPECIALIX_BH,
        AURORA_BH,
        ESP_BH,
        NET_BH,
        SCSI_BH,
        IMMEDIATE_BH,
        KEYBOARD_BH,
        CYCLADES_BH,
        CM206_BH,
        JS_BH,
        MACSERIAL_BH,
        ISICOM_BH
};
```

每种下半部类型都可通过 init_bh 而关联上一个函数处理函数。例如，网络代码把 NET_BH 下半部类型初始化为 net_dev_init 中的net_bh处理函数，如第五章所述。

```
__initfunc(int net_dev_init(void))
{
        ... ... ...
        init_bh(NET_BH, net_bh);
        ... ... ...
}
```

用于为BH处理函数除名的主要函数是remove_bh（还有其他相关函数，如enable_bh/disable_bh，但是我们不需要全都讨论）。

每当中断处理函数想触发下半部函数的执行时，就必须用 mark_bh 设置相应的标识。此函数相当简单：即在全局位图 bh_active 中设置一个位；稍后就会看到，此位图会在多处进行测试。

```
extern inline void mark_bh(int nr)
 {
        set_bit(nr, &bh_active);
};
```

例如，本章稍后会讲到，每次网络设备驱动程序成功接收一个帧后，就调用netif_rx以通知内核。内核会把新接收的帧排进入口队列backlog（被所有CPU共享），然后标记NET_BH下半部函数标识。

```
skb_queue_tail(&backlog, skb);
mark_bh(NET_BH);
return
```

在几个函数的运行过程中，内核会检查是否有任何下半部例程进入调度准备执行。如果有任何下半部例程处于等待中，内核就会执行函数do_bottom_half（目前在*kernel/softirq.c*中）以执行下半部例程。这些检查是在下列过程中进行的：

do_IRQ

每当内核接到IRQ通知说有硬件中断发生时，就会调用do_IRQ以执行相关联的处理函数。因为中断处理函数会使得很多下半部例程进入调度准备传送，要如何做才能使它们的延时少于do_IRQ结束时立刻启用另一函数的时间？因此，定期定时器中断事件，以HZ频率为到期单位，代表着两个do_bottom_half连续执行间隔的时间上限。

从中断和异常事件（包括系统调用）返回

参见*arch/XXX/kernel/entry.S*中处理此情况的汇编语言代码。

schedule

此函数决定CPU接下来要执行什么，它会检查是否有任何下半部函数处于未决中，然后给予其高于其他任务的优先级。

```
asmlinkage void schedule(void)
{
        /* 这里可以做一些“管理性”的工作，因为我们没有持有任何锁 */
        if (bh_mask & bh_active)
                goto handle_bh;
handle_bh_back:
        ... ... ...
handle_bh:
        do_bottom_half();
        goto handle_bh_back;
        ... ... ...
}
```

由do_bottom_half使用的函数run_bottom_half会执行未决的中断处理函数，如下所示：

```
        active = get_active_bhs();
        clear_active_bhs(active);
        bh = bh_base;
        do {
                if (active & 1)
                        (*bh)();
                bh++;
                active >>= 1;
        } while (active);
```

这些处于未决中的处理函数的调用次序取决于位图中相关联标识的位置，以及用于扫描这些标识的方向（由get_active_bhs返回）。换而言之，下半部的执行不是以谁先来谁先得到服务的方式进行的。此外，网络下半部可能会花掉很长时间，那些不幸到最后才从队列退出的下半部例程就会经历很长的延时。

2.2版本和早期内核中的下半部例程禁止并发。任何时刻，只能有一个下半部例程允许运行，无论CPU数目有几个。

内核2.4版本以后的下半部函数：引入软IRQ

就中断处理而言，内核2.2版本和2.4版本之间的最大改进就是引入了软件中断（软IRQ）；而软件中断可视为下半部函数的多线程版本。许多软IRQ不仅可以同时运行，而且相同的软IRQ还可以在不同的CPU上同时运行。对并发的唯一限制就是无论何时，在一个CPU上每个软IRQ都只能有一个实例在运行。

新式的软IRQ模型只有6种类型（来自于*include/linux/interrupt.h*）：

```
enum
{
        HI_SOFTIRQ=0,
        TIMER_SOFTIRQ,
        NET_TX_SOFTIRQ,
        NET_RX_SOFTIRQ,
        SCSI_SOFTIRQ,
        TASKLET_SOFTIRQ
};
```

旧模型中所有*XXX*_BH下半部类型依然可供旧驱动程序使用，但是，这些旧类型也已全都重新实现，以HI_SOFTIRQ类型（表示其优先级高于其他软IRQ类型）的软IRQ运行。网络代码所用的两种类型会在稍后的“网络代码如何使用软IRQ”一节介绍：NET_TX_SOFTIRQ和NET_RX_SOFTIRQ。下一节会介绍微任务（tasklet）。

如同旧式下半部一样，软IRQ执行时，中断功能是开启的，因此任何时候都可挂起，以处理新进来的中断事件。然而，如果软IRQ的一个实例已在一个CPU上挂起了，内核是不允许针对该软IRQ的新请求又在该CPU上运行，这样可以大幅减少所需的上锁量。每种软IRQ类型都维护着一个类型为softnet_data的数据结构数组，而每个CPU都有一个softnet_data类型的数据结构，以存储当前软IRQ的状态信息；“softnet_data结构”一节将阐述此结构的内容。因为同一种类型的软IRQ的不同实例可以同时在不同CPU上运行，因此，软IRQ所执行的函数还是必须锁住其他共享的数据结构，以避免竞争情况的发生。

为软IRQ函数注册和调度的那些函数以及其背后的逻辑，都与2.2版本的下半部函数所用的那些非常类似。

软IRQ函数是以open_softirq函数注册，而与init_bh不同的是，此函数会接收一个额外的参数，当需要时，可以传一些输入数据给函数处理函数。然而，软IRQ都还

没有使用到这个额外的参数，因此要予以删除的提案已浮出。open_softirq只是把输入参数拷贝到在*kernel/softirq.c*中声明的全局数组softirq_vec，而此数组存有类型和处理函数之间的关联。

```
static struct softirq_action softirq_vec[32] __cacheline_aligned_in_smp;

void open_softirq(int nr, void (*action)(struct softirq_action*), void *data)
{
    softirq_vec[nr].data = data;
    softirq_vec[nr].action = action;
}
```

软IRQ可通过下列函数在本地CPU上进入调度而准备执行：

__raise_softirq_irqoff

此函数在2.2版本中是mark_bh的配对函数，只是设置与要执行的软IRQ相关联的位标识。稍后，检查此标识时，相关联的处理函数就会被调用。

raise_softirq_irqoff

这是内含__cpu_raise_softirq的包裹函数，如果此函数不是从硬件或软件中断环境中调用，而且抢占功能没被关闭，就会另外再为ksoftirqd线程（本章稍后讨论）调度。如果此函数是从中断环境中调用，调用此线程就是不必要的，因为do_softirq一定会被调用（稍后就会知道）。

raise_softirq

这是内含raise_softirq_irqoff的包裹函数，而执行时，中断功能会关闭。

下列代码取自内核2.4.5版本（注7），说明软IRQ开发初期所用的模型。此模型非常类似2.2版本模型，而且会调用函数do_softirq，该函数是2.2版本中do_bottom_half函数的配对函数，这在前节已经讨论过。如果至少有一个软IRQ已进入调度准备执行，do_softirq就会被调用：

```
asmlinkage void schedule(void)
{
        /* 这里可以做一些“管理性”的工作，因为我们没有持有任何锁 */
        if (softirq_active(this_cpu) & softirq_mask(this_cpu))
                goto handle_softirq;
handle_softirq_back:
        ... ... ...
handle_softirq:
        do_softirq();
        goto handle_softirq_back;
        ... ... ...
}
```

注7：　在2.4.6版本中已被删除。

这个初期阶段的软IRQ模型和2.2版本的下半部模型的唯一差别就是，软IRQ版模型必须以每个CPU为基准检查标识，因为每个CPU都有自己的未决软IRQ位图。

do_softirq的实现非常类似于其2.2版本中的配对函数do_bottom_half。内核也会在一些相同的地点调用此函数，但并不完全都相同。主要的改变是引入了一个新的各个CPU（per-CPU）内核线程ksoftirqd。

以下是do_softirq会被调用的主要地点（注8）：

do_IRQ

定义在各个体系结构文件*arch/arch-name/kernel/irq.c*中的do_IRQ的架构如下：

```
fastcall unsigned int do_IRQ(struct pt_regs * regs)
{
    irq_enter();
    ... ... ...
    /* 使用已注册处理函数处理IRQ编号“irq” */
    ... ... ...
    irq_exit();
    return 1;
}
```

在内核2.4版本中，此函数称为do_softirq。对于2.6版本多数体系结构而言，对do_softirq的调用改在irq_exit之内进行，但有少部分还是在do_IRQ中调用。

因为可以对irq_enter做嵌套调用（nested call），所以，只有当所有通常的条件都吻合（没有软IRQ处于未决中等等）以及与中断环境相关联的引用计数已到达零时，即表示内核要离开中断环境了，irq_exit才会调用invoke_softirq。

以下是来自于*kernel/softirq.c*的irq_exit的通用定义，但是，有些体系结构会定义其自己的版本：

```
void irq_exit(void)
{
    ...
    sub_preempt_count(IRQ_EXIT_OFFSET);
    if (!in_interrupt() && local_softirq_pending())
        invoke_softirq();
    preempt_enable_no_resched();
}
```

在*arch/XXX/kernel/apic.c*中的处理SMP定时器的smp_apic_timer_interrupt，也会使用irq_enter/irq_exit。

注8：也有可能调用invoke_softirq，而非直接调用do_softirq。invoke_softirq也可能是do_softirq的别名，或者是其辅助函数__do_softirq，这要依赖于__ARCHIRQ_EXIT_IRQS_DISABLED服务是否有定义。

从中断和异常事件（包括系统调用）返回

同于内核2.2版本。

`local_bh_enable`

当一个CPU上再重新开启软IRQ时，调用`do_softirq`就可处理未决的请求（如果有的话）。

内核线程 `ksoftirqd_CPUn`

为了阻止软IRQ垄断所有CPU（这种情况在高负载的网络上很容易发生，因为`NET_TX_SOFTIRQ`和`NET_RX_SOFTIRQ`中断事件的优先级高于用户进程），开发人员引进了新的一组各个CPU线程。这些线程的名称是`ksoftirqd_CPU0`、`ksoftirqd_CPU1`等等，而且*ps*命令看得见这些线程。

我所说的是i386一般性的行为，其他体系可能使用不同的命名规则，或者有其他也会启用`do_softirq`的定时器。

另一个调用`do_softirq`的有趣地点是在`netif_rx_ni`中，对此第十章“设备驱动程序与内核间的旧接口：netif_rx的第一部分”一节会说明。内建至内核的流量产生器（traffic generator）也会调用`do_softirq`（位于*net/core/pktgen.c*）。

微任务

2.2.x版本内核的多数下半部功能都已转换为软IRQ或微任务。微任务是函数，可以延迟某个中断事件或其他任务，使其晚一点执行。微任务建立在软IRQ之上，通常是由中断处理函数发出，但是内核其他部分也会用到微任务，如本书第六部分所提到的邻居子系统（注9）。

在“内核2.4版本以后的下半部函数：引入软IRQ”一节中，我们看到了软IRQ列表。`HI_SOFTIRQ`是用于实现高优先级的微任务，而`TASKLET_SOFTIRQ`是用于实现较低优先级的微任务。每次，发出延迟执行的请求之后，`tasklet_struct`结构的一个实例就会排入一个由`HI_SOFTIRQ`所处理的列表，或者另一个由`TASKLET_SOFTIRQ`所处理的列表。

由于软IRQ是由每个CPU各自独立处理，每个CPU有两份未决`tasklet_struct`列表，一份与`HI_SOFTIRQ`关联，而另一份与`TASKLET_SOFTIRQ`关联，因此就应该没什么好惊讶的了。以下是来自于*kernel/softirq.c*的定义：

注9：内核也提供工作队列。我们不谈及此是因为网络代码已很少使用。有关工作队列的说明，可参考《深入理解Linux内核》一书。

```
static DEFINE_PER_CPU(struct tasklet_head tasklet_vec) = { NULL };
static DEFINE_PER_CPU(struct tasklet_head tasklet_hi_vec) = { NULL };
```

初看起来，微任务似乎就是旧式的下半部函数，但实际上有一些实质性的差异：

- 不同微任务的数目不受限制，但是bh_base的每个位标识只限于一种类型的下半部函数。
- 微任务提供两种等级的优先级。
- 不同微任务可以在不同的CPU上同时运行。
- 微任务与旧式下半部函数及软IRQ的不同之处就是动态，不需要在*XXX*_BH或*XXX*_SOFTIRQ枚举列表中静态声明。

tasklet_struct数据结构定义在*include/linux/interrupt.h*中，如下所示：

```
struct tasklet_struct
{
    struct tasklet_struct *next;
    unsigned long state;
    atomic_t count;
    void (*func)(unsigned long);
    unsigned long data;
};
```

下面逐一描述其字段。

struct tasklet_struct *next

是一个用于把关联到同一个CPU的未决结构链接起来的指针。新元素由函数tasklet_hi_schedule和tasklet_schedule添加到头部。

unsigned long state

位图标识，其可能的取值由TASKLET_STATE_XXX枚举列表在*include/linux/interrupt.h*中。

TASKLET_STATE_SCHED

此微任务已进入调度准备执行，而该数据结构已放在与HI_SOFTIRQ或TASKLET_SOFTIRQ（依赖于所分派的优先级）关联的列表中。相同微任务不能同时在不同的CPU上调度。当第一个微任务还没开始执行前，其他执行此微任务的请求又到来了，则这些请求会被丢弃掉。因为对任何微任务而言，都只能有一个实例在执行中，所以，没有理由再调度而又多执行一次。

TASKLET_STATE_RUN

此微任务正被执行中。此标识用于防止相同微任务的多个实例被同时执行。这

一点只对SMP系统有意义。此标识的操作通过三个上锁函数完成。tasklet_trylock、tasklet_unlock以及tasklet_unlock_wait。

atomic_t count

有些情况下，可能必须暂时关闭而后重新开启微任务。这是由计数器完成的：零值表示微任务被关闭（因此不可执行），而非零值意味着微任务已开启。其值由tasklet[_hi]_enable和tasklet[_hi]_disable函数（本节稍后说明）递增和递减。

void (*func)(unsigned long)

unsigned long data

func是要执行的函数，而data是选用的输入数据，可传给func。

下列是一些来自于*kernel/softirq.c*和*include/linux/interrupt.h*的处理微任务的重要内核函数：

tasklet_init

把自变量func和data的值填入tasklet_struct结构字段中。

tasklet_action、tasklet_hi_action

分别执行低优先级和高优先级的微任务。

tasklet_schedule、tasklet_hi_schedule

为低优先级和高优先级的微任务调度以准备执行。这些函数会把tasklet_struct结构添加到与本地CPU相关联的未决微任务列表中，然后为相关联的软IRQ（TASKLET_SOFTIRQ或HI_SOFTIRQ）调度。如果微任务已进入调度（但尚未运行），这些API就什么也不做（参见TASKLET_STATE_SCHED标识）。

tasklet_enable、tasklet_hi_enable

这两个函数都一样，用于开启微任务。

tasklet_disable、tasklet_disable_nosync

这两个函数都是用于关闭微任务，而且可以用于高优先级和低优先级的微任务。tasklet_disable是内含tasklet_disable_nosync的包裹函数。当tasklet_disable_nosync返回（异步），只有当微任务已终止其执行时，tasklet_disable才会返回（同步，以免微任务还在运行）。

tasklet_enable、tasklet_hi_enable以及tasklet_disable_nosync会操作count字段的值，以声明微任务为已开启或已关闭，但不能做嵌套调用。

软 IRQ 初始化

在内核初始化期间，softirq_init 会以两个通用软 IRQ 对软 IRQ 层做初始化：tasklet_action 和 tasklet_hi_action（分别与 TASKLET_SOFTIRQ 以及 HI_SOFTIRQ 相关联）。

```
void __init softirq_init()
{
    open_softirq(TASKLET_SOFTIRQ, tasklet_action, NULL);
    open_softirq(HI_SOFTIRQ, tasklet_hi_action, NULL);
}
```

这两个由网络代码 NET_RX_SOFTIRQ 和 NET_TX_SOFTIRQ 所用的软 IRQ 是在 net_dev_init（网络初始化函数之一；参见“网络代码如何使用软 IRQ”一节）中初始化的。

在“内核 2.4 版本以后的下半部函数：引入软 IRQ”一节中，所列的其他软 IRQ 都会在相关联的子系统内注册（SCSI_SOFTIRQ 是 *drivers/scsi/scsi.c*，TIMER_SOFTIRQ 是 *kernel/timer.c*，等等）。

HI_SOFTIRQ 主要是由声卡设备驱动程序使用的（注 10）。

TASKLET_SOFTIRQ 的使用者包括：

- 网络适配卡的驱动程序（不仅限于 Ethernet）。
- 各种其他设备驱动程序。
- 媒介层（USB、IEEE 1394 等）。
- 网络子系统（邻居、ATM qdisc 等）。

未决软 IRQ 的处理

我们在“内核 2.4 版本以后的下半部函数：引入软 IRQ”一节说过，do_softirq 何时会被启用以负责那些未决的软 IRQ。这里我们来说明该函数的内部细节。你会注意到，该函数与“内核2.2版本的下半部函数”一节中所说的2.2版本内核所用的函数非常相似。

如果CPU正在于服务硬件或软件中断，do_softirq就会停止而什么事也不做。此函数会调用 in_interrupt（相当于 in_irq() || in_softirq()）以检查此事。

注 10：在内核 2.4 版本中，内核 2.2 版本的下半部例程都已转成高优先级微任务，方法是把 mark_bh 函数定义成内含 tasklet_hi_schedule 的包裹函数。

如果do_softirq决定继续下去，就会以local_softirq_pending把未决的软IRQ存储在pending。

```
#ifndef __ARCH_HAS_DO_SOFTIRQ

asmlinkage void do_softirq(void)
{
    if (in_interrupt())
        return;
    local_irq_save(flags);
    pending = local_softirq_pending();
    if (pending)
        __do_softirq();
    local_irq_restore;
}
EXPORT_SYMBOL(do_softirq);
#endif
```

由上述程序片段来看，do_softirq执行时似乎IRQ是关闭的，但并非如此。只有在操作未决的软IRQ的位图时（也就是访问softnet_data结构），IRQ才会被关闭。等一下就会知道，执行软IRQ函数时，__do_softirq会在内部重新开启IRQ。

__do_softirq函数

在do_softirq运行时，同一种软IRQ类型有可能被调度多次。由于在执行软IRQ函数时IRQ会被开启，而服务于一个中断事件时，未决的软IRQ的位图也可以受到操作，因此，任何已被__do_softirq执行的软IRQ函数在__do_softirq本身执行期间有可能会被重新调度。

因此，__do_softirq重启IRQ之前，会把未决的软IRQ的当前位图存储至局部变量pending，然后使用local_softirq_pending()=0将其从与本地CPU相关联的softnet_data实例中清除。最后，再根据pending调用所有必需的处理函数。

一旦所有处理函数都已被调用，__do_softirq会同时检查是否有任何软IRQ又已进入调度（这个请求会关闭IRQ）。如果至少有一个未决的软IRQ，则会重复整个流程。然而，__do_softirq只会重复最多MAX_SOFTIRQ_RESTART次(实验结果发现，10次就可以工作得很好)。

使用MAX_SOFTIRQ_RESTART是一种设计决策，为地是在几个CPU中的其中之一上，网络流中的某种类型的中断事件不会使其他中断事件饿死。没有__do_softirq的限制，当服务器因网络流量而承受高负载，并且NET_RX_SOFTIRQ中断事件的数目达到最高的话，就很容易会发生饿死的事。

来看饿死是如何发生的。do_IRQ引发一个NET_RX_SOFTIRQ中断事件，使得do_softirq被执行。__do_softirq会清除NET_RX_SOFTIRQ标识，但是，结束前可能会被另一个中断事件中断，使得NET_RX_SOFTIRQ会再次被设置，结果就这样没完没了。

现在，我们来看__do_softirq的关键部分是如何启用软IRQ函数的。每一次一种软IRQ类型受到服务时，其位就会从活跃的软IRQ的本地副本pending中清掉。h会初始化为指向全局数据结构softirq_vec，该数据结构持有软IRQ类型及其函数处理函数间的关联（例如，NET_RX_SOFTIRQ是由net_rx_action处理）。当位图被清除时，此循环就结束。

最后，如果因为do_softirq已重复其工作MAX_SOFTIRQ_RESTART次，有些未决的软IRQ仍无法被处理而必须返回，则ksoftirqd线程会被唤醒，赋予稍后处理这些未决的软IRQ的责任。因为do_softirq会在内核内许多地方启用，实际上很有可能在ksoftirqd线程调度前，晚一点调用的do_softirq就会处理这些中断事件。

```
#define MAX_SOFTIRQ_RESTART 10

asmlinkage void __do_softirq(void)
{
    struct softirq_action *h;
    __u32 pending;
    int max_restart = MAX_SOFTIRQ_RESTART;
    int cpu;

    pending = local_softirq_pending();

    local_bh_disable();
    cpu = smp_processor_id();
restart:
    /* 开启irqs前先重设pending位屏蔽 */
    local_softirq_pending() = 0;

    local_irq_enable();

    h = softirq_vec;

    do {
        if (pending & 1) {
         h->action(h);
         rcu_bh_qsctr_inc(cpu);
        }
      h++;
      pending >>= 1;
} while (pending);

    local_irq_disable();
```

```
        pending = local_softirq_pending();
        if (pending && --max_restart)
            goto restart;

        if (pending)
            wakeup_softirqd();

        __local_bh_enable();
    }
```

依体系结构处理软IRQ

*kernel/softirq.c*中所提供的do_softirq函数可以被体系结构代码（最后还是调用__do_softirq）所提供的另一个函数覆写。这说明了为什么*kernel/softirq.c*中do_softirq的定义会以__ARCH_HAS_DO_SOFTIRQ包裹起来检查（参见前一节）。

有些体系结构会定义自己的do_softirq版本，包括i386（参见*arch/i386/kernel/irq.c*）。当体系结构使用4 KB堆栈（而非8 KB），而且以剩余的4 KB去实现堆栈来处理硬IRQ（hard IRQ）和软IRQ时，就会使用这类体系结构版本。请参考《Understanding the Linux Kernel》一书中的详细说明。

ksoftirqd内核线程

后台内核线程被分派的工作是检查没被前述函数执行到的软IRQ，然后，在必须把CPU交还给其他活动前，尽可能多地执行几个这些没被执行的软IRQ。每个CPU都有这么一个内核线程，名为ksoftirqd_CPU0、ksoftirqd_CPU1等。“启动线程”一节会说明这些线程在CPU引导期间是如何启动的。

和这些线程相关的ksoftirqd函数其实相当简单，就定义在同一个文件中（*softirq.c*）：

```
static int ksoftirqd(void * __bind_cpu)
{
    set_user_nice(current, 19);
    ...
    while (!kthread_should_stop()) {
        if (!local_softirq_pending())
            schedule();

        __set_current_state(TASK_RUNNING);
        while (local_softirq_pending()) {
            /* 抢占功能关闭可防止CPU离线（offline）。
               但如果已离线，表示我们在错误的CPU上：
               不要处理。*/
            preempt_disable();
            if (cpu_is_offline((long)__bind_cpu))
                goto wait_to_die;
```

```
                do_softirq();
                preempt_enable();
                cond_resched();
            }
            set_current_state(TASK_INTERRUPTIBLE);
        }
        __set_current_state(TASK_RUNNING);
        return 0;
        ...
    }
```

有几个小细节我想强调一下。进程优先级也称为仁慈优先级（nice priority），其值介于−20（最大）～+19（最小）之间。ksoftirqd线程被给定一个很低的优先级19。只有这么做，那些时常执行的软IRQ（如NET_RX_SOFTIRQ）才不会完全绑定CPU，使得其他进程几乎根本毫无资源可用。我们已经知道代码中有许多地方都会启用do_softirq，因此，这样的低优先级别并没有什么阻碍。启动之后，循环只会持续调用do_softirq（抢占功能总是关闭），直到下列条件之一成立：

- 没有未决的软IRQ需要处理了（local_softirq_pending()返回FALSE）。

  在这种情况下，此函数会把线程的状态设成TASK_INTERRUPTIBLE，然后调用schedule()以释放CPU。线程可以通过wakeup_softirqd（可以由wakeup_softirqd本身和raise_softirq_irqof调用）而被唤醒。

- 线程已执行太久了，被要求释放CPU。

  除了其他该做之事外，与定时器中断事件相关联的处理函数会设置need_resched标识，通知说当前进程/线程已用掉其时间片（time slot）。在这种情况下，ksoftirqd会释放CPU，而使其状态保持在TASK_RUNNING，因为很快就会重新继续下去。

启动线程

每个CPU都有一个ksoftirqd线程。当系统的第一个CPU上线（online）时，第一个线程就会在do_pre_smp_initcalls（注11）中启动（内核引导期间）。至于其他在引导时间运行起来的CPU，以及支持可热插拔CPU的系统上，有其他可供稍后开启的CPU，其ksoftirqd线程则是由cpu_chain通知链负责。

第四章已介绍过通知链。cpu_chain链让各种子系统知道一个CPU何时起来运行或者何时死掉。软IRQ子系统会以spawn_ksoftirqd（从刚才提到的do_pre_smp_initcalls函数内调用）对cpu_chain注册。回调函数cpu_callback会处理来自于

注11： 参见第七章有关内核如何完成引导期间的基本初始化。

cpu_chain的通知信息，而且可用于对必要的各个CPU数据结构进行初始化，以及启动CPU上的ksoftirqd线程。

CPU_*XXX*通知信息的完整列表在*include/linux/notifier.h*中，但是就本章的内容而言，我们只需要其中四个：

CPU_UP_PREPARE

当CPU将要运行起来但还没就绪时，就会产生。

CPU_ONLINE

当CPU就绪时，就会产生。

CPU_UP_CANCELLED

CPU_DEAD

只有当内核编译为支持可热插拔CPU时，这两条信息才会产生。当先前的CPU_UP_PREPARE通知信息所触发的任务之一失败时，就会使用第一个通知信息，因此，CPU就无法上线了。当CPU死掉时，就会用到第二个。

CPU_PREPARE_UP会建立线程，并将此与相关联的CPU绑定，但是没有唤醒该线程。CPU_ONLINE唤醒该线程。当CPU死掉时，其相关联的ksoftirqd实例也会被杀掉：

```
static int __devinit cpu_callback(struct notifier_block *nfb, unsigned
long action, void *hcpu)
{
    ...
    switch(action) {
        ...
    }
    return NOTIFY_OK;
}

static struct notifier_block __devinitdata cpu_nfb = {
   .notifier_call = cpu_callback
};

__init int spawn_ksoftirqd(void)
{
   void *cpu = (void *)(long)smp_processor_id();
     cpu_callback(&cpu_nfb, CPU_UP_PREPARE, cpu);
   cpu_callback(&cpu_nfb, CPU_ONLINE, cpu);
   register_cpu_notifier(&cpu_nfb);
    return 0;
}
```

注意，通过register_cpu_notifier向cpu_chain注册前，spawn_ksoftirqd会直接调用两次cpu_callback。这是必要的，因为对于第一个上线的CPU不会产生CPU通知信息。

微任务的处理

低延时微任务（TASKLET_SOFTIRQ）和高延时微任务（HI_SOFTIRQ）的两个处理函数都相同：只是处理的是两个不同的列表。因此，我们只说明其中一个tasklet_action，也就是与TASKLET_SOFTIRQ相关联的那个。

任何时刻，每个微任务都只有一个实例可以等待执行。当tasklet_schedule或tasklet_hi_schedule为一个微任务调度时，该函数会设置“微任务”一节所提到的TASKLET_STATE_SCHED位。尝试重新为相同微任务调度时则会被忽视掉，因为TASKLET_STATE_SCHED已设置。只有当微任务开始执行时，该位才会被清除；因此，在微任务执行期间或者在其执行之后，另一个实例就能被调度了。

tasklet_action函数会先把等待处理的微任务列表拷贝到一个局部变量，然后清除该全局列表（注12）。此函数以中断功能关闭而执行的地方只有这里而已。关闭中断功能是必要的，为的是避免引发竞争，即其他中断处理函数可以把新元素添加到该列表中，但tasklet_action又在访问该列表。

此时，该函数会按照列表一个微任务接一个微任务地执行。就每个元素而言，如果下列条件都为成立，就会启用处理函数：

- 微任务已经停止运行，换而言之，TASKLET_STATE_RUN被清除（此函数会执行tasklet_trylock以得知TASKLET_STATE_RUN是否已设置，如果没有，tasklet_trylock就会设置该位）。
- 微任务已开启（count是零）。

函数中实现这些活动的部分如下：

```
struct tasklet_struct *list;

local_irq_disable();
list = __get_cpu_var(tasklet_vec).list;
__get_cpu_var(tasklet_vec).list = NULL;
  local_irq_enable();

  while (list) {
    struct tasklet_struct *t = list;

    list = list->next;

    if (tasklet_trylock(t)) {
          if (!atomic_read(&t->count)) {
```

注12：有一个网络软IRQ处理函数（net_rx_action）也是做类似的事。

此时，由于微任务还没有执行，而且已从未决的微任务列表中抽取出来，因此，TASKLET_STATE_SCHED标识必须设置：

```
                if (!test_and_clear_bit(TASKLET_STATE_SCHED, &t->state))
                    BUG();
                t->func(t->data);
                tasklet_unlock(t);
                continue;
            }
            tasklet_unlock(t);
        }
```

如果处理函数不能执行，微任务就会被放回列表中，而TASKLET_SOFTIRQ会重新调度，以完成那些现在因前述两点理由之一而无法处理的所有微任务：

```
        local_irq_disable();
            t->next = __get_cpu_var(tasklet_vec).list;
            __get_cpu_var(tasklet_vec).list = t;
            __raise_softirq_irqoff(TASKLET_SOFTIRQ);
            local_irq_enable();
        }
    }
```

网络代码如何使用软IRQ

网络子系统分派两种不同的软IRQ。NET_RX_SOFTIRQ会处理进来的流量，而NET_TX_SOFTIRQ会处理出去的流量。这两个软IRQ都是通过下列几行代码在net_dev_init中注册（见第五章描述）：

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

因为同一个软IRQ函数的不同实例可以在不同的CPU上同时执行（和微任务不同），因此，网络代码不但是低延时的，而且又具有适应性。

这两个网络软IRQ的优先级都比普通微任务（TASKLET_SOFTIRQ）要高，但是，比高优先级微任务（HI_SOFTIRQ）要低。这样的优先级安排，即使系统处于高网络负载情况下，也可以保证其他高优先级任务的运行具有足够的响应力和及时的效果。

这两个处理函数的内部细节会在第十章“处理NET_RX_SOFTIRQ：net_rx_action”一节以及第十一章“处理NET_TX_SOFTIRQ：net_tx_action”一节加以说明。

softnet_data结构

到第十章就会知道，每个CPU都有其队列，用来接收进来的帧。因为每个CPU都有其

数据结构用来处理入口和出口流量，因此，不同CPU之间没必要使用上锁机制。此队列的数据结构softnet_data就定义在*include/linux/netdevice.h*中，如下所示：

```
struct softnet_data
{
    int             throttle;
    int             cng_level;
    int             avg_blog;
    struct sk_buff_head    input_pkt_queue;
    struct list_head       poll_list;
    struct net_device      *output_queue;
    struct sk_buff         *completion_queue;
    struct net_device      backlog_dev;
}
```

此结构的字段可用于接收和传输。换而言之，NET_RX_SOFTIRQ和NET_TX_SOFTIRQ软IRQ都引用此结构。入口帧会排入input_pkt_queue（注13），而出口队列会放入由流量控制（QoS层）所处理的特殊队列（而不是由软IRQ和softnet_data结构所处理），但是，稍后软IRQ依然可用于清理已传输的缓冲区，以免该任务传输缓慢。

softnet_data的字段

下面对此数据结构的主要字段逐一进行描述，其细节会在后续各章说明。有些驱动程序使用NAPI接口，而其他的则还没更新为使用NAPI。这两种类型的驱动程序都会使用此结构，但是，有些字段是保留给非NAPI驱动程序的。

throttle

avg_blog

cng_level

这三个参数由拥塞管理算法使用，此列表后面会再进一步说明，而第十章“拥塞管理”一节也会加以说明。默认地，每接收一个帧这三个字段都会被更新。

input_pkt_queue

这个队列(在net_dev_init中初始化)用来保存进来的帧(被驱动程序处理前)。非NAPI驱动程序也会使用此字段，那些还没更新为使用NAPI的则会使用其自己私有的队列。

backlog_dev

这是一个完整的嵌入式数据结构(不只是一个指针而已)，类型为net_device，代表着一个设备已在相关联的CPU上为net_rx_action调度以准备执行。这个字段

注13： 到第十章就会知道，对于使用NAPI的驱动程序而言不再是真的了。

是由非NAPI驱动程序使用。其名称指的是“backlog device”（积压设备）。在第十章“设备驱动程序和内核间的旧接口：netif_rx的第一部分”一节将看到其用法。

poll_list

这是一个双向列表，其中的设备都带有输入帧等着被处理。第十章“处理NET_RX_SOFTIRQ：net_rx_action”一节中可找到其他细节。

output_queue

completion_queue

output_queue是设备列表，其中的设备有数据要传输。而completion_queue是缓冲区列表，其中的缓冲区已成功传输，因此可以释放掉。相关细节可以参考第十一章“处理NET_TX_SOFTIRQ：net_tx_action”一节。

throttle被视为布尔变量，当CPU超负荷则为真，否则为假。其值依赖于input_pkt_queue中的帧数目。当throttle标识设置时，此CPU所接收的所有输入帧都会被丢弃，无论队列中的帧数目有多少（注14）。

avg_blog代表地是input_pkt_queue队列长度加权后的平均值，其值界于0到最大长度(由netdev_max_backlog表示)之间。avg_blog可用于计算cng_level。

cng_level代表拥塞等级，其值可以是图9-4中的任何值。当avg_blog遇到此图所示的极限之一时，cng_level就改变其值。NET_RX_XXX枚举值的定义放在*include/linux/netdevice.h*中，而拥塞等级的定义mod_cong、lo_cong以及no_cong则放在*net/core/dev.c*中（注15）。方括号中的字符串（/DROP和/HIGH）会在第十章“拥塞管理”一节说明。默认会按每个帧重新计算avg_blog和cng_level，但是，重新计算可能会被延缓，而且和定时器绑在一起以免过于耗时。

avg_blog与cng_level是与CPU相关的，因此，可用于非NAPI设备（共享每个CPU所用的队列input_pkt_queue）。

注14：使用NAPI的驱动程序在这些情况下可能不会丢弃进来的流量。

注15：这些NET_RX_XXX值也用在此环境以外的地方，而且有些NET_RX_XXX值并没有用在此处。no_cong_thresh值就没用在这里，以前是由process_backlog（第十章说明）使用，在某些条件下，从throttle（窒息）状态下删除一个队列（当内核还支持此功能时，但已经被废弃了）。

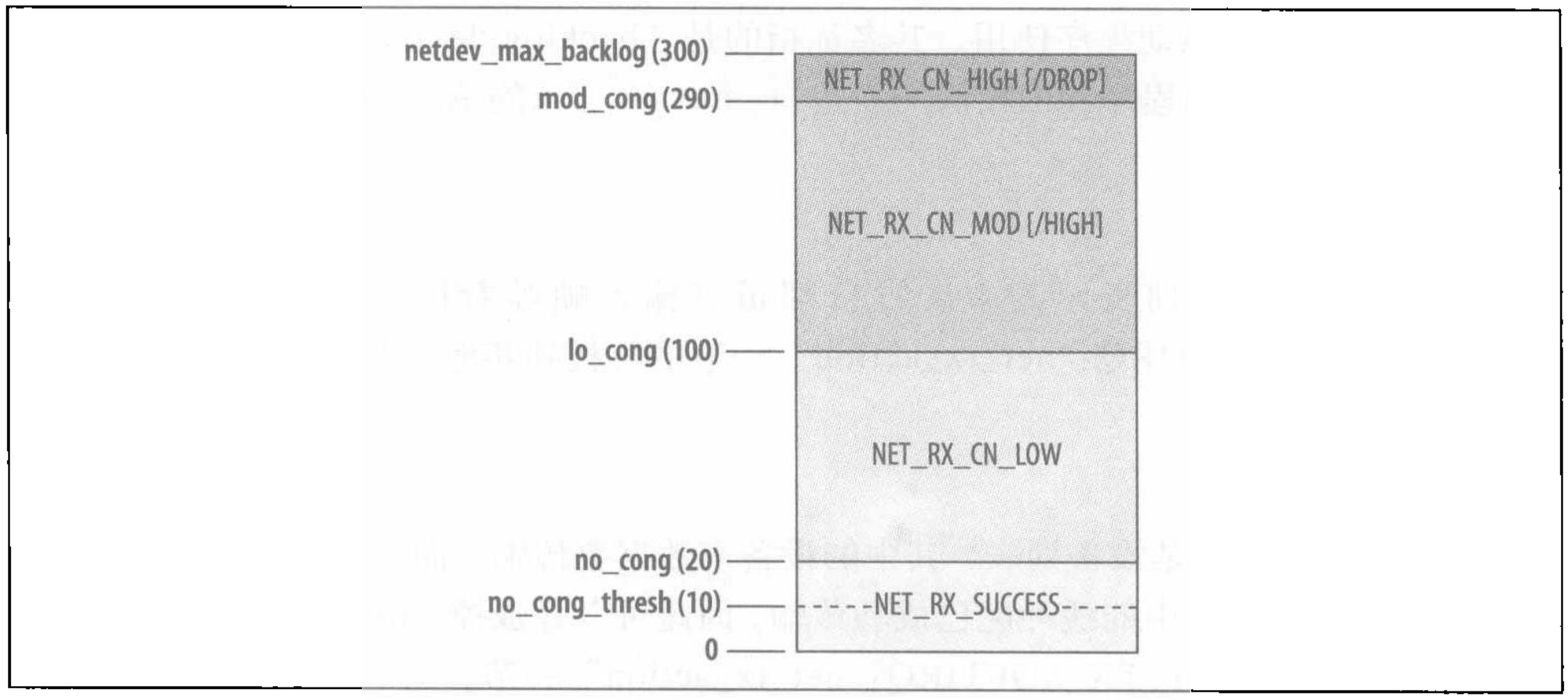


图 9-4：根据平均积压值 avg_blog 得到的拥塞等级（NET_RX_XXX）

softnet_data 的初始化

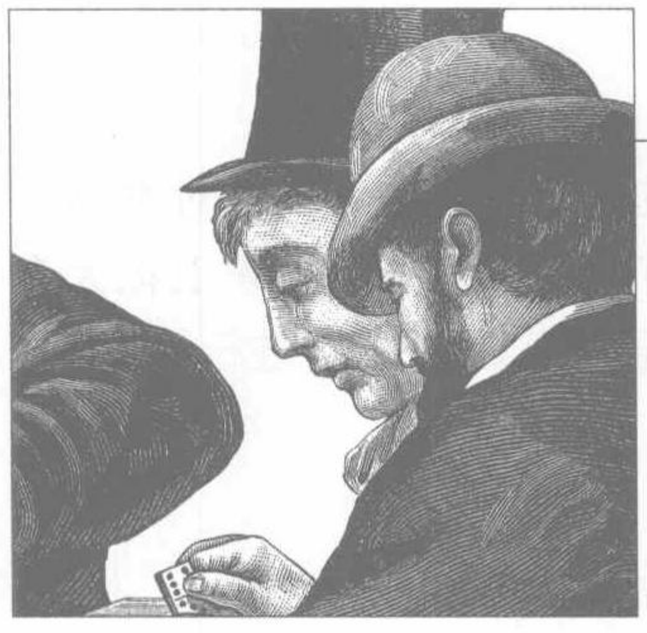
每个 CPU 的 softnet_data 结构是由 net_dev_init 在引导期间执行初始化的（第五章有说明）。其初始化代码如下：

```
for (i = 0; i < NR_CPUS; i++) {
    struct softnet_data *queue;

    queue = &per_cpu(softnet_data,i);
    skb_queue_head_init(&queue->input_pkt_queue);
    queue->throttle = 0;
    queue->cng_level = 0;
    queue->avg_blog = 10; /* 任何非零值 */
    queue->completion_queue = NULL;
    INIT_LIST_HEAD(&queue->poll_list);
    set_bit(__LINK_STATE_START, &queue->backlog_dev.state);
    queue->backlog_dev.weight = weight_p;
    queue->backlog_dev.poll = process_backlog;
    atomic_set(&queue->backlog_dev.refcnt, 1);
}
```

NR_CPUS是Linux内核所能处理的最大CPU数目，而softnet_data是一个向量，内含一些 struct softnet_data 结构。

此程序也会对softnet_data->blog_dev（此结构的类型为net_device，为一种特殊设备，代表的是非 NAPI 设备）的字段做初始化。第十章“积压的处理：process_backlog 轮询虚拟函数”一节会说明如何使用旧式 netif_rx 接口，而以透明化的方式处理非 NAPI 设备驱动程序。



第十章

帧的接收

前一章已经知道，处理L2层的帧的函数是由中断事件驱动的。本章我们着手讨论帧的接收，此时，硬件会使用中断事件通知CPU，该帧已经可用了。

如第九章中的图9-2所示，接收中断事件的CPU会执行do_IRQ函数。IRQ编号会引发正确的处理函数被启用。此处理函数通常是设备驱动程序在设备驱动程序初始化期间所注册的函数。IRQ函数处理例程会在中断模式下执行，即后续的中断事件都会暂时被关闭。

如第九章“中断处理函数”一节所述，中断处理函数会执行一些立即性的任务，然后把其他任务安排到下半部函数中以便在稍后执行。明确地讲，中断处理函数会：

1. 把帧拷贝到sk_buff数据结构（注1）。
2. 对一些sk_buff参数做初始化，以便在稍后由上面的网络层使用（显然是skb->protocol，借此表示较高层协议处理函数，而且到了第十三章将成为主要角色）。
3. 更新其他一些该设备私用的参数，但本章不予讨论，因为不会影响帧在网络协议栈内的路径。

为NET_RX_SOFTIRQ软IRQ调度以准备执行，借此通知内核新帧的事。

由于设备发出中断事件的理由各有不同（新帧已接收、帧已成功传输等等），内核的代码会配合中断通知信息，使得设备驱动程序处理例程可以按类型处理中断事件。

注1：如果该设备使用DMA（现今是相当普遍的事），驱动程序就只需初始化一个指针（不需要做拷贝）。

与其他功能交互

研究本章介绍的函数时，会时常看见一些代码片段和选用的内核功能交互。针对本书所涉及的功能，我会指出该功能所在的章节。就其他功能而言，我不会在其代码上花太多的时间。本章所示的流程图多数都会显示出这些选用功能会在函数的何处进行处理。

以下是我们会看见的选用功能及其相关联的内核符号：

802.1d Ethernet Bridging（CONFIG_BRIDGE/CONFIG_BRIDGE_MODULE）

桥接在本书第四部分描述。

Netpoll（CONFIG_NETPOLL）

Netpoll是一个通用的架构，可通过轮询网络适配卡（NIC）而传送及接收帧，把中断事件的需求删除掉。任何内核功能都可使用Netpoll，以受益于其功能。著名的案例是Netconsole，可以通过UDP把内核信息（例如，用printk打印字符串）发送到远程主机。Netconsole及其子选项可以通过*make xconfig*菜单打开："Networking support → Network console logging support"选项。要使用Netpoll时，设备必须支持才行（有不少已经支持）。

Packet Action（CONFIG_NET_CLS_ACT）

通过此功能，流量控制可以对入口流量进行分类，并且采取一些操作。可能的操作包括丢弃封包以及消化封包。要查看*make xconfig*菜单中的这个选项以及其所有子选项，必须选取"Networking support → Networking options → QoS and/or fair queueing → Packet classifier API"选项。

设备的开启和关闭

当net_device->state中的__LINK_STATE_START标识被设置时，设备就可被视为已开启。第八章"开启和关闭网络设备"一节已说明了此标识的细节。当设备打开时（dev_open），这个标识通常就会被设置，而当设备关闭时（dev_close），就会被清除。虽然有一个标识是明确用于开启或关闭设备的传输（__LINK_STATE_XOFF），却没有标识用于开启或关闭接收。此能力是通过其他方式完成的，也就是关闭设备（如第八章所述）。__LINK_STATE_START标识的状态可以由netif_running函数检查。

本章稍后显示的几个函数都是简单的包裹函数，可以检查标识，如__LINK_STATE_START的正确状态，以确保设备已准备好做其该做之事。

队列

讨论L2行为时，我通常会提到帧接收和传输时所需的队列（入口队列和出口队列）。每个队列都有一个指针指向其相关联的设备，以及一个指针指向存储输入/输出缓冲区的skb_buff数据结构。只有少数专用的设备可以不需要队列就能工作，其中一例是回环设备。回环设备可以省掉队列，因为当你从回环设备传出一个封包时，该封包会立刻被递送出去（给本地系统），而无需排入队列的中间过程。再者，因为在回环设备上的传输不会失败，也没必要让封包重新排入队列以尝试另一次传输。

出口队列和设备直接相关联；流量控制（QoS层）会为每个设备都定义一个队列。在第十一章将会知道，内核会记录等待传输帧的设备而非帧本身。我们也会知道，并非所有设备都在用流量控制。对于入口队列而言，情况比较复杂，稍后就会知道。

通知内核帧已接收：NAPI和netif_rx

Linux 2.5版本内核中（然后回头移植到2.4版本的新修正版中）引进了一套新API以处理入口帧，由于找不到更好的名称，我们称之为NAPI（New API，新型API）。因为很少有设备更新成使用NAPI，所以Linux驱动程序通知内核新帧的事有两种方式：

通过旧函数netif_rx

这种方法是由那些遵循第九章“在中断期间处理多帧”一节所提到的技术的设备所使用。多数Linux设备驱动程序依然使用这种手法。

通过NAPI机制

这种方法是由那些遵循第九章“在中断期间处理多帧”一节末尾所介绍的改进的技术的设备所使用。这是Linux内核中的新事物，但是只有少数驱动程序用到。*drivers/net/tg3.c*是第一个改用NAPI的驱动程序。

当用*make xconfig*这类工具配置内核选项时，有些设备驱动程序可让你在这两种接口间做选择。

下列代码片段来自于vortex_rx，依然使用旧函数netif_rx，因此，可以想象得到，多数尚未改用NAPI的网络设备驱动程序，其所做之事都很类似：

```
skb = dev_alloc_skb(pkt_len + 5);
    ... ... ...
if (skb != NULL) {
    skb->dev = dev;
    skb_reserve(skb, 2);     /* 把IP对齐在16字节边界上 */
            ... ... ...
            /* 把DATA拷贝到sk_buff结构 */
```

```
            ... ... ...
        skb->protocol = eth_type_trans(skb, dev);
        netif_rx(skb);
        dev->last_rx = jiffies;
            ... ... ...
    }
```

首先，以dev_alloc_skb（见第二章）分配sk_buff数据结构，然后帧会拷贝进去。注意，拷贝前，程序会保留两个字节使IP报头对齐16字节边界。每个网络设备驱动程序都被关联上给定的接口类型；例如，Vortex设备驱动程序*driver/net/3c59x.c*就关联到一个特定的Ethernet卡系列。因此，驱动程序知道链接层报头的长度，也知道该如何予以解读。假设报头长16*k+n，则驱动程序可以调用skb_reserve，用16−n作为偏移量，就可使其对齐16字节边界。Ethernet报头是14个字节，所以k=0，n=14，而程序所需的偏移量为2（参见*include/linux/sk_buff.h*中NET_IP_ALIGN的定义以及相关的批注）。

此外应注意，此时驱动程序并没有区分不同的L3协议。驱动程序把L3报头对齐到16字节边界上，而不管其类型。L3协议可能是IP，因为IP使用最广泛，但是此时还无法保证，也有可能是Netware的IPX或其他东西。无论所用的L3协议为何，对齐都有用处。

关于eth_type_trans将在第十三章描述，可用于提取协议标识符skb->protocol（注2）。

取决于驱动程序设计的复杂度，其程序区段后面可能有其他的管理工作，但本书对这些细节不作探讨。此函数最重要的部分是把帧接收的通知信息传给内核。

NAPI简介

虽然有些NIC设备驱动程序尚未改用NAPI，但是新型基础架构已经整合至内核，而且连netif_rx和内核其余部分之间的接口也都把NAPI考虑进来了。我们先谈NAPI而非先介绍旧的方法（纯netif_rx），再说明旧驱动程序如何保有其旧接口（netif_rx），同时又共享一些新型基础架构机制。

NAPI混合了中断事件和轮询，在高流量负载下其性能会比旧方法要好，因为可以大幅减少CPU的负载。内核开发人员又把这个基础架构反回来移植到2.4版内核。

在旧模型中，设备驱动程序会为其所接收的每个帧都产生一个中断事件。在高流量负载下，花在处理中断事件的时间会造成资源相当程度的浪费。

注2：不同的设备类型使用不同的函数，例如eth_type_trans是由Ethernet设备使用，而tr_type_trans是由Token Ring接口使用。

NAPI背后的主要想法很简单：混合使用中断事件和轮询，而不使用纯粹的中断事件驱动模型。如果接收到新帧时，内核还没完成处理前几个帧的工作，驱动程序就没必要产生其他中断事件：让内核一直处理设备输入队列中的数据会比较简单一点（该设备的中断功能关闭），然后当该队列为空时再重新开启中断功能。如此一来，驱动程序就能获得中断事件和轮询的优点：

- 异步事件 —— 如帧的接收 —— 是由中断事件指出，如此一来，如果设备的入口队列是空的，内核就不用一直去检查了。
- 如果内核知道设备的入口队列中有数据存在，就没必要浪费时间去处理中断事件通知信息。用简单的轮询就够了。

从内核处理的观点来看，以下是NAPI方法的一些优点：

减少了CPU的负载（因为中断事件变少了）

假设工作量（也就是每秒帧数）相同，则使用NAPI时CPU的负载比较低。这在高工作量时特别明显。在低工作量时使用NAPI，根据内核开发人员在内核邮件论坛所公布的测试报告，实际上的CPU用量还会高一点。

设备的处理更为公平

稍后就会知道，一些设备的入口队列中若有数据，就会以相当公平的循环方式予以访问。这样就能确保当其他设备的负载都很高时，低流量的设备所体验到的延时依然处于可接受的范围之内。

NAPI所用之net_device字段

查看NAPI的实现和用法之前，我必须说明一些net_device数据结构的字段，第九章“softnet_data结构”一节曾提到过。

为了处理驱动程序使用NAPI接口的设备，有四个新字段添加到此结构中，以供NET_RX_SOFTIRQ软IRQ使用。其他设备不会用到这些字段，但是它们可共享嵌入在softnet_data结构中作为backlog_dev字段的net_device结构的字段。

`poll`

这个虚拟函数可用于把缓冲区从设备的输入队列中退出。此队列是使用NAPI设备的私有队列，而softnet_data->input_pkt_queue供其他设备使用。参见“积压的处理：process_backlog轮询虚拟函数”一节。

`poll_list`

这是设备列表，其中的设备就是在入口队列中有新帧等待被处理的设备。这些设备

就是所谓的处于轮询状态。此列表的头为softnet_data->poll_list。此列表中的设备都处于中断功能关闭状态，而内核当前正在予以轮询。

quota

weight

quota（配额）是一个整数，代表的是poll虚拟函数一次可以从队列退出的缓冲区的最大数目。其值的增加以weight为单位，用于在不同设备间施加某种公平性。配额愈低，表示潜在的延时愈低，因此让其他设备饿死的风险就愈低。另一方面，低配额会增加设备间的切换量，因此整体的耗费会增加。

对配有非NAPI驱动程序的设备而言，weight的默认值为64（存储在*net/core/dev.c*顶端的weight_p变量）。weight_p之值可通过*/proc*修改。

对配有NAPI驱动程序的设备而言，默认值是由驱动程序所选。最常见的值是64，但是也有使用16和32的。其值可通过*sysfs*调整。

就*/proc*和*sysfs*接口而言，可参考第十二章“通过/proc和sysfs文件系统调整”一节。

使用NAPI的设备会对这四个字段以及其他net_device字段根据第八章所述的初始化模型做初始化。关于虚拟的backlog_dev设备，第九章“softnet_data的初始化”一节曾有介绍，而且本章稍后也会说明，其初始化工作由net_dev_init（在第五章描述）完成。

net_rx_action和NAPI

图10-1所示是每次内核轮询进来的网络流量时所发生的事。在此图中可以看到poll_list列表，其内的设备处于轮询状态，也可以看到poll虚拟函数以及软中断函数net_rx_ action之间的关系。下面几节会详细说明此图各个方面的细节，但是转入源码之前，了解这些部分如何交互是很重要的。

我们已经知道net_rx_action是与NET_RX_SOFTIRQ标识相关联的函数。为了简洁起见，我们假设在一段低活动量期间之后有些设备开始接收帧，而这些行为触发了net_rx_action的执行，至于是如何做到的，就目前而言并不重要。

net_rx_action会浏览列表中处于轮询状态的设备，然后为每个设备都调用相关联的poll虚拟函数，以处理入口队列中的帧。稍早曾说明过，该列表中的设备会按照循环方式被查阅，而且每次其poll方法启用时，能处理的帧数目都有最大值存在。如果在其时间片内无法使队列清空，就得等到下一个时间片继续下去。也就是说，net_rx_

action会持续为入口队列中有数据的设备调用其设备驱动程序所提供的poll方法，直到入口队列为空。到那时就不用再轮询了，而设备驱动程序就可重新开启该设备的中断事件通知功能。值得强调的是，中断功能关闭只针对那些在poll_list中的设备，也就是只用于那些使用NAPI而且不共享backlog_dev的设备。

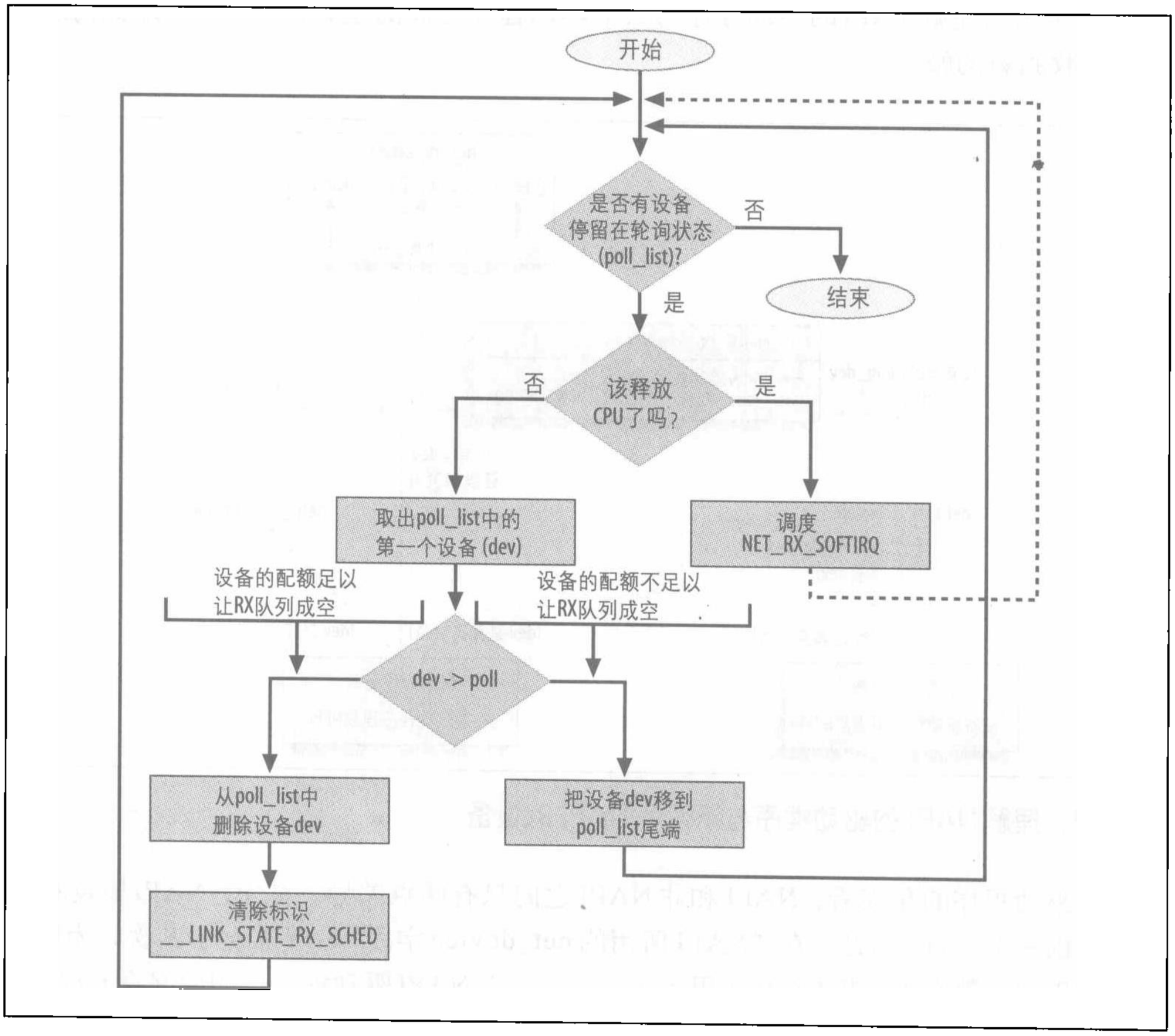


图 10-1：net_rx_action 函数和 NAPI 概述

net_rx_action会限制其执行时间，当其用完限制的执行时间或处理过一定数量的帧后，就会自行重新调度以准备执行，这样是为了强制net_rx_action能与其他内核任务彼此公平运行。同时，每个设备也会限制其poll方法每次启用时所能处理的帧数目，才能与其他设备之间彼此公平运行。当设备无法清空其入口队列时，就得等到下一次调用其poll方法的时候。

新旧驱动程序接口

现在，net_device结构和NAPI相关的字段意义，以及NAPI背后高层思想应该都已经清楚了，我们可以向源码走近一点了。

图10-2显示出理解NAPI的驱动程序与其他驱动程序之间的差异性，驱动程序如何通知内核接收到新的帧。

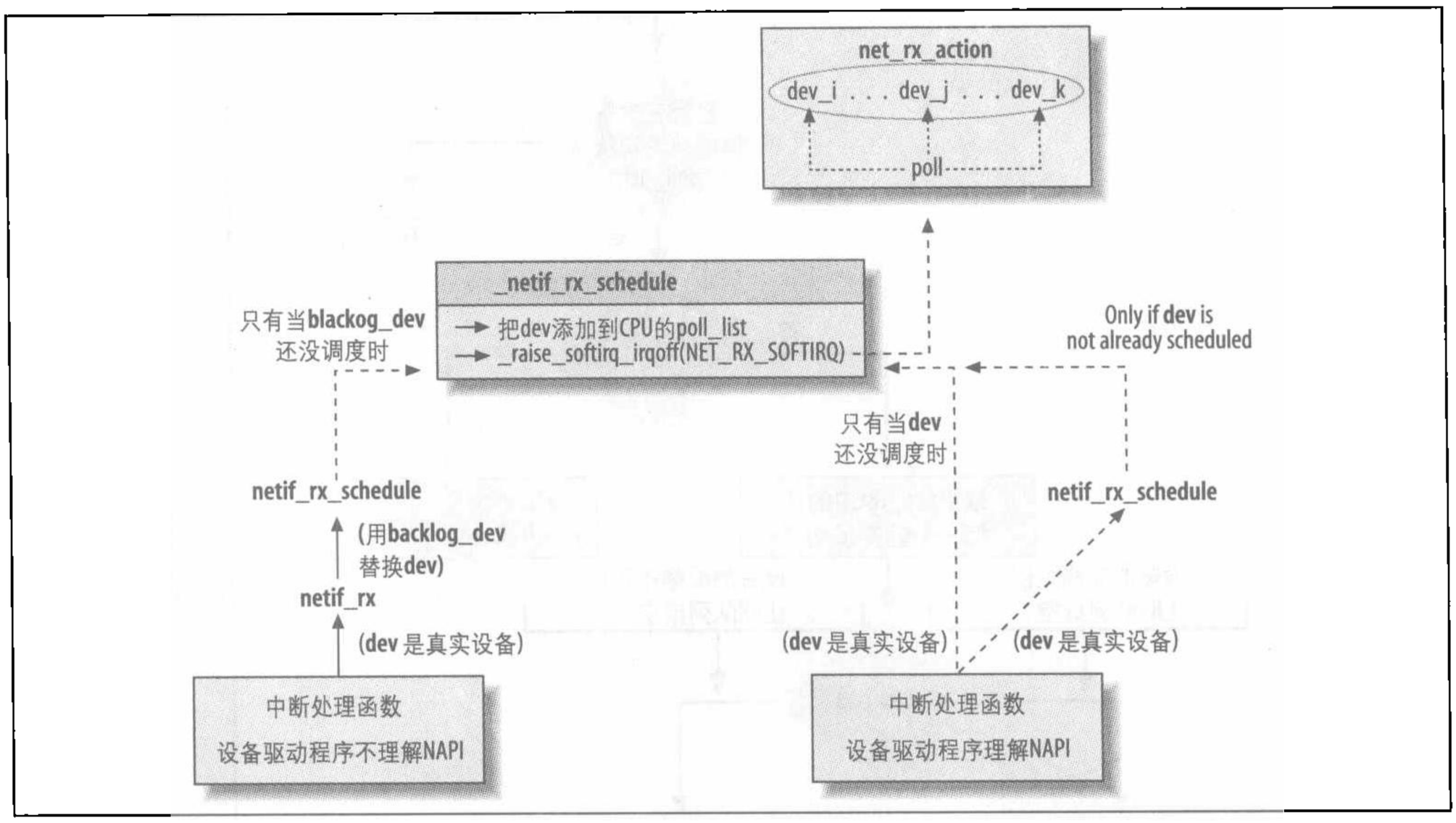


图 10-2：理解 NAPI 的驱动程序与不理解 NAPI 的设备

从设备驱动程序的角度看，NAPI和非NAPI之间只有两点差异。首先，NAPI驱动程序必须提供一个poll方法，在“NAPI所用的net_device字段”一节描述。其次，为帧调度所调用的函数有别：非NAPI调用netif_rx，而NAPI驱动程序调用定义在*include/linux/netdevice.h*中的__netif_rx_schedule。内核提供一个名为netif_rx_schedule的包裹函数，检查以确保该设备正在运行，而且该软IRQ还没调度，然后才调用__netif_rx_schedule。这些检查由以netif_rx_schedule_prep进行的。有些驱动程序会调用netif_rx_schedule，而其他的驱动程序则明确调用netif_rx_schedule_prep，而必要时再调用__netif_rx_schedule。

如图10-2所示，这两种驱动程序都会把输入设备排入轮询列表(poll_list)，为NET_RX_SOFTIRQ软中断调度以准备执行，最后再由net_rx_action予以处理。即使这两种驱动程序最后都会调用__netif_rx_schedule (非NAPI驱动程序会在netif_rx中做这件事)，NAPI设备所给予的性能会好很多，而其原因已在第九章“接收到帧时通知驱动程序”一节说明。

图10-2中的一项重要细节是，在两种情况下传给__netif_rx_schedule的net_device结构。非NAI设备使用的是内建至CPU的softnet_data结构的net_device结构，而NAPI设备使用的是涉及它们自己的net_device结构。

操作poll_list

前节已知，调用netif_rx_schedule或__netif_rx_schedule时，就可以把任何设备（包括虚拟的backlog_dev）添加到poll_list列表。

逆向操作——从列表中移除设备——是由netif_rx_complete或__netif_rx_complete所做的（第二个是假设本地CPU的中断功能已关闭）。在“处理NET_RX_SOFTIRQ：net_rx_action”一节将会看到这两个例程在何时调用。

设备可以通过netif_poll_disable和netif_poll_enable暂时关闭和重新开启轮询。这并不表示设备驱动程序已决定返回到中断模式。例如，当设备必须被设备驱动程序复位，以应用某种硬件配置变更时，对设备的轮询可能就得关闭。

我已说过，netif_rx_schedule会为poll_list中的设备（也就是__LINK_STATE_RX_SCHED标识被设置）过滤请求。因此，如果驱动程序设置该标识，但是没有设备添加到poll_list，基本上就是为该设备关闭轮询：该设备绝不会被添加到poll_list。这是netif_poll_disable的工作方式：如果_LINK_STATE_RX_SCHED没设置，就会予以设置，然后返回；否则，就等待该标识被清除，然后再予以设置。

```
static inline void netif_poll_disable(struct net_device *dev)
{
    while (test_and_set_bit(_ _LINK_STATE_RX_SCHED, &dev->state)) {
        /* 一点都不急。*/
        current->state = TASK_INTERRUPTIBLE;
        schedule_timeout(1);
    }
}
```

设备驱动程序与内核间的旧接口：netif_rx的第一部分

当新的输入帧正等待处理时（注3），设备驱动程序通常是调用定义在*net/core/dev.c*中

注3：有个有趣的异常：当SMP系统的一个CPU死掉时，dev_cpu_callback函数会清空相关联的softnet_data实例的input_pkt_queue队列。dev_cpu_callback就是net_dev_init在cpu_chain内注册的回调函数，在第九章已介绍。

的netif_rx函数，此函数的工作是为短暂执行的软IRQ（把帧退出队列然后予以处理）调度。图10-3所示是其所做之事以及其事件的流程。此图实际上比代码还长，但是有助于了解netif_rx流程中的脉络。

netif_rx通常是在中断环境下被驱动程序调用，但是，也有例外，特别是当此函数是被回环设备调用时。因此，netif_rx在其启动时会关闭本地CPU的中断事件，然而当其完成工作时会再予以重新开启（注4）。

查看代码时应该记住一点：不同的CPU可以同时执行netif_rx。这不是问题，因为每个CPU都配有一个私有的softnet_data结构以维护状态信息。此外，CPU的softnet_data结构还包括一个私有的输入队列（参见第九章的“softnet_data结构”一节）。

以下是此函数的原型：

```
int netif_rx(struct sk_buff *skb)
```

其唯一的输入参数就是该设备接收的缓冲区，而其输出值就是指出拥塞等级之值（“拥塞管理”一节可找到细节）。

netif_rx的主要任务如下，其详细的流程图如图10-3所示。

- 对sk_buff数据结构的一些字段做初始化（如接收帧的时间）。
- 把已接收的帧存储到CPU的私有输入队列，然后触发相关联的软IRQ NET_RX_SOFTIRQ以通知内核时。只有当某些条件满足时，这个步骤才会发生，而最重要的一点就是队列中是否有空间。
- 更新有关拥塞等级的统计数据。

图10-4所示是一个系统范例，其中有一些CPU和设备。每个CPU都有自己的softnet_data实例，而该实例内有私有的输入队列，netif_rx会在此处存储入口帧，此外，当缓冲区已不再需要时，就会被送往completion_queue（参见第十一章“处理NET_TX_SOFTIRQ：net_tx_action”一节）。此图显示出一个范例，CPU 1从*eth0*接收到一个RxComplete中断事件。相关联的驱动程序会把入口帧存储至CPU 1的队列。CPU m从*ethn*接收到一个DMADone中断事件，指出已传输的缓冲区已不再需要，因此可以移往completion_queue队列（注5）。

注4：netif_rx_ni是net_if_rx的姊妹函数，用于非中断环境中。使用此函数的系统之一是TUN（Universal TUN/TAP）设备驱动程序（*drivers/net/tun.c*）。

注5：input_pkt_queue和completion_queue_keep只持有指向缓冲区的指针（只不过图中看起来好像是实际存储整个缓冲区）。

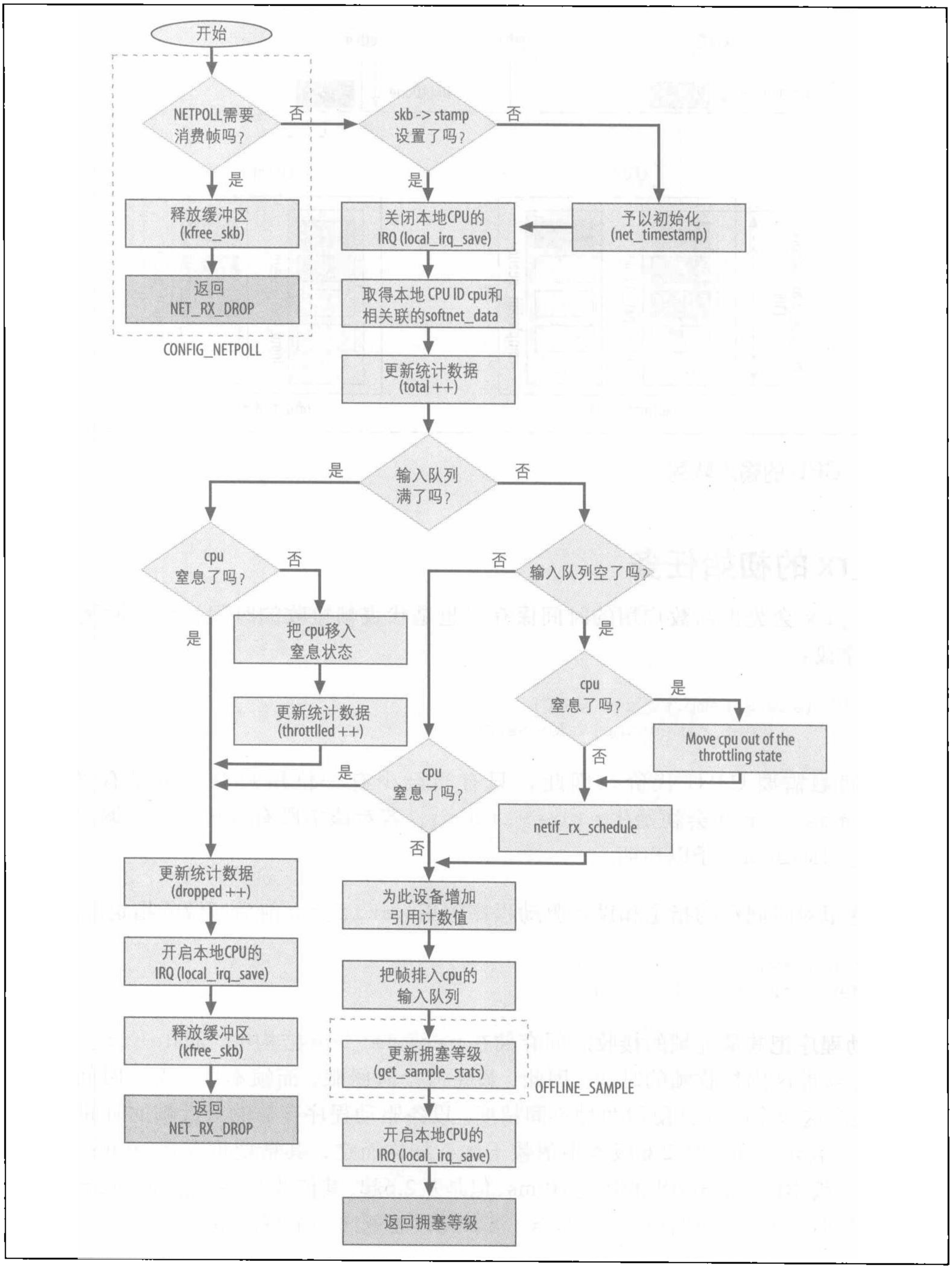


图 10-3：netif_rx 函数

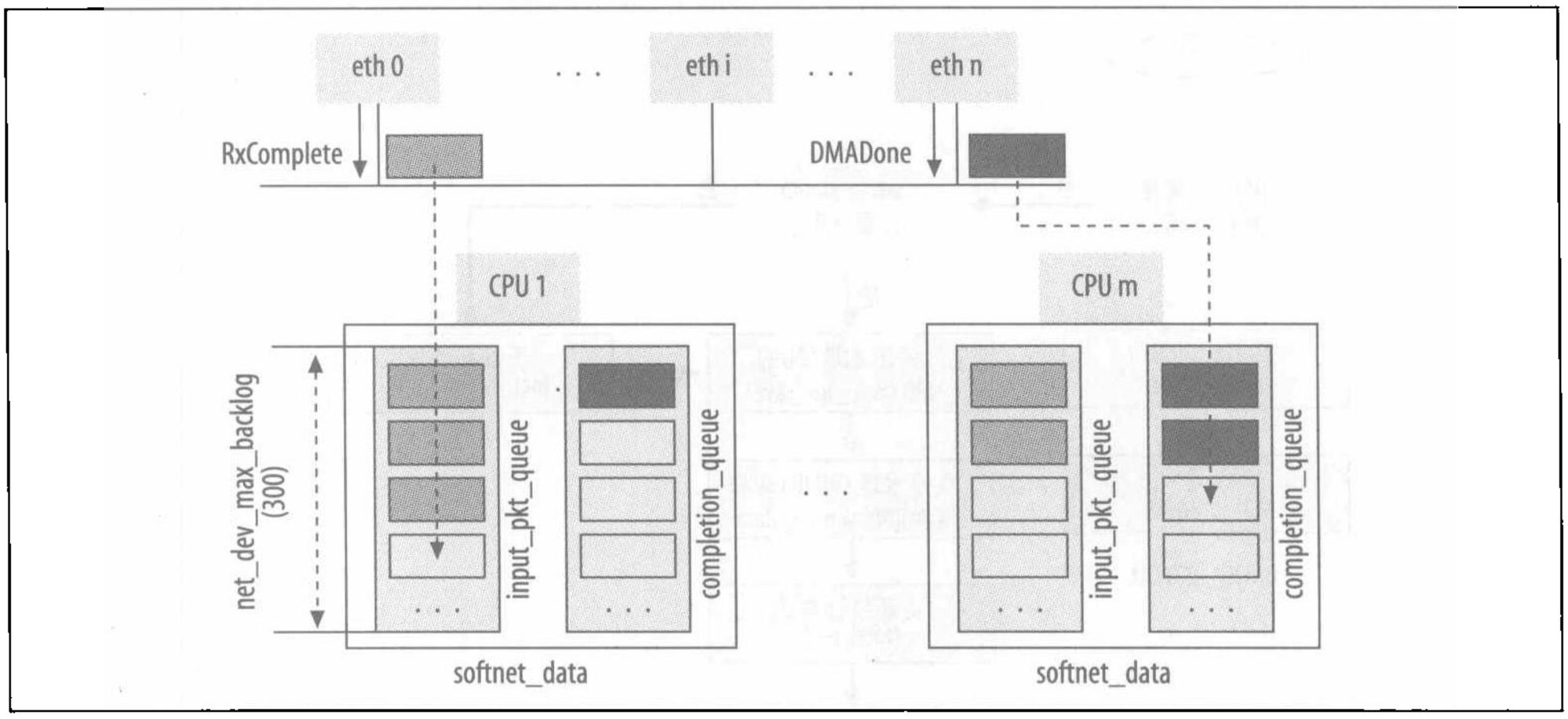


图 10-4：CPU 的输入队列

netif_rx 的初始任务

netif_rx 会先把函数启用的时间保存（也是代表帧接收的时间）到缓冲区结构的 stamp 字段：

```
if (skb->stamp.tv_sec == 0)
        net_timestamp(&skb->stamp);
```

保存时间戳需要 CPU 代价，因此，只有当至少有一位用户对该字段有兴趣时，net_timestamp 才会初始化 skb->stamp。若对该字段有兴趣，可以调用 net_enable_timestamp 予以声明。

不要把这里对时间戳的指定和设备驱动程序调用 netif_rx 前后所做的指定混淆了：

```
netif_rx(skb);
dev->last_rx = jiffies;
```

设备驱动程序把其最近帧的接收时间存储在 net_device 结构中，而 netif_rx 存储的时间是缓冲区内接收帧的时间。因此，设备有个时间戳，而帧本身也有个时间戳。再者，要注意这两个时间戳使用两种不同精度。设备驱动程序存储的最近帧的时间戳是以 jiffies 表示，在内核 2.6 版本中依赖于体系结构而定，其精度可以是 10 ms 或 1 ms（例如，2.6版本以前，i386用的值是10 ms，但是从2.6起，其值为1 ms）。然而，netif_rx 取得其时间戳的方式是调用 get_fast_time，而返回一个更高精度的值。

本地 CPU 的 ID 通过调用 smp_processor_id()获取，并存储在局部变量 this_cpu 之中：

```
this_cpu = smp_processor_id();
```

我们必须使用本地CPU ID，才能在各个CPU向量中取出与该CPU相关联的数据结构，如下列netif_rx中的代码所示：

```
queue = &__get_cpu_var(softnet_data);
```

上列代码会把指向softnet_data结构的指针存储在queue。softnet_data结构与本地CPU相关联，为调用netif_rx的设备驱动程序所触发的中断事件提供服务。

现在，netif_rx会更新CPU所接收的帧总数：包括接受的以及丢弃的（例如，因为队列中已没有空间）：

```
netdev_rx_stat[this_cpu].total++
```

每个设备驱动程序也会持有统计数据，将其存储在dev->priv所指的私有数据结构中。这些统计数据——包括已接收的帧数目、已丢弃的帧数目——都是按各个设备分别存储的（参见第二章），但是，由netif_rx更新的统计数据则是按各个CPU分别存储的。

管理队列以及下半部调度

输入队列是由softnet_data->input_pkt_queue管理。每个输入队列都有最大长度，由全局变量netdev_max_backlog指定，其值为300。也就是说，每个CPU的输入队列中最多只能有300个帧等待处理，而无论系统中的设备数目有多少（注6）。

常识告诉我们，netdev_max_backlog的值应该依赖于设备数目以及设备的速度。然而，在SMP系统中这一点很难追踪，因为中断事件都是动态分布在CPU之间，哪个设备会与哪个CPU对话很难看出来。因此，netdev_max_backlog值的选择是通过试验和出错而得到的。可以想见，将来其值会以动态方式设置，以反映出接口的类型和数目。其值已经可由系统管理员做配置调整，如第十二章“通过/proc和sysfs文件系统调整”一节所述。性能议题如下：这个值太大时没有必要的，只会浪费内存。另一方面，这个值太小就会使得设备的性能降低，因为突然爆量的流量可能会导致很多帧被丢弃。最佳值多半依赖于系统的角色（主机、服务器或路由器等等）。

在以前的内核中，当时各个CPU数据结构softnet_data还不存在，所有设备会共享一个输入队列，名为backlog，而其大小都是300个帧。使用softnet_data的主要收

注6：这一点也适用于非NAPI设备。因为NAPI设备使用私有队列，设备可以选择它们喜欢的最大长度。常见的值有16、32以及64。10 Gigabit Ethernet驱动程序*drivers/net/s2io.c*使用较大的值（90）。

获不是n个CPU使队列有n*300个帧空间，而是CPU之间不再需要上锁，因为每个CPU都有自己的队列。

下列代码能控制在什么条件下netif_rx会将其新帧插入到一个队列，以及在什么条件下会为该队列调度以准备执行：

```
        if (queue->input_pkt_queue.qlen <= netdev_max_backlog) {
            if (queue->input_pkt_queue.qlen) {
                if (queue->throttle)
                    goto drop;
enqueue:
                dev_hold(skb->dev);
                __skb_queue_tail(&queue->input_pkt_queue,skb);
#ifndef OFFLINE_SAMPLE
                get_sample_stats(this_cpu);
#endif
                        local_irq_restore(flags);
                return queue->cng_level;
            }
            if (queue->throttle)
                queue->throttle = 0;
            netif_rx_schedule(&queue->backlog_dev);
            goto enqueue;
        }
        ... ... ...
drop:
        __get_cpu_var(netdev_rx_stat).dropped++;
        local_irq_restore(flags);
        kfree_skb(skb);
        return NET_RX_DROP;
}
```

第一个if语句确认是否还有空间。如果队列已满，则语句返回false，而CPU就会被放入窒息状态（throttle state），也就是说输入流量已超载，因此后续的帧都会被丢弃。创立窒息的代码在这里没有显示出来，但是，下一节讨论拥塞管理时就会出现。

然而，即使队列有空间也不足以确保该帧会被接受。CPU可能已处于“窒息”状态（由第三个if语句确认），就此而言，该帧还是会被丢弃。

当队列成空时，窒息状态就可解除。这就是第二个if语句所做的测试。当队列上有数据而且CPU处在窒息状态时，该帧就会被丢弃。但是，当队列为空，而CPU处于窒息状态时（也就是此处所示的代码下半部的if语句所做的测试），窒息状态就会被解除（注7）。

注7：这种情况实际上很罕见，因为net_rx_action可能早已解除窒息状态“通过process_backlog间接解除”。本章稍后会予以说明。

调用dev_hold(skb->dev)会为该设备增加引用计数值，使得该设备无法被移除，直到此缓冲区已完全处理完为止。相应的递减是由dev_put完成，会在net_rx_action内发生，本章稍后会予以分析。

如果所有测试都满足，则该缓冲区就会用_skb_queue_tail(&queue->input_pkt_queue,skb)排入输入队列，而该CPU的IRQ的状态就会被恢复，然后此函数返回。

把帧排入队列是相当快的，因为不涉及任何内存拷贝，只是指针操作而已。input_pkt_queue是指针列表。__skb_queue_tail把指向新缓冲区的指针添加到列表中，而没有拷贝该缓冲区。

NET_RX_SOFTIRQ软中断可通过netif_rx_schedule调度以准备执行。注意，只有当新缓冲区添加到空队列时，netif_rx_schedule才会被调用。其原因在于如果队列不为空，则NET_RX_SOFTIRQ已经被调度了，因此，没必要再调用一次。

在第九章“未决的软IRQ的处理”一节中我们已经知道，内核会负责已调度的软件中断。接下来一节“处理NET_RX_SOFTIRQ：net_rx_action”中会说明NET_RX_SOFTIRQ软IRQ的处理例程的内部细节。

拥塞管理

拥塞管理是输入帧处理任务中的重要部分。超负荷的CPU会变得不稳定，对系统造成很大的延时。第九章“中断”一节已说明高负载所产生的中断事件为何会使得系统瘫痪，因此需要拥塞管理机制以确保系统的稳定性，使得在高网络负载下不会受到拖累。在高流量负载下降低CPU负载的常见方式包括：

尽可能减少中断事件的数目

实现方法是把驱动程序编写为在一次中断事件中处理许多帧（参见第九章“在中断期间处理多帧”一节），或者使用NAPI。

在入口径中尽早丢弃帧

如果程序知道有一个帧会被那些较高层丢弃掉，就可以迅速丢弃帧，以节省CPU时间。例如，如果设备驱动程序知道入口队列已经满了，就可以立刻丢弃帧，而不用转发给内核使其稍后再予以丢弃。

本节要谈的就是第二点。

类似的最优化也会应用在输出路径上：如果设备驱动程序没有资源可接受新的帧以传输

(也就是内存不足)，再让内核把新帧压入驱动程序以传输，那就是浪费CPU时间了。这一点会在第十一章“传输的开启和关闭”一节讨论。

就接收和传输这两种情况而言，内核都提供了一组函数，可设置、清除以及获取接收和传输队列的状态，因此，就能使设备驱动程序（在接收上）以及内核核心程序（在传输上）执行刚才所提到的优化工作。

拥塞等级的良好就是已接收的帧数目以及正等待处理的帧的数目。当设备驱动程序使用NAPI时，就是由驱动程序实现任何拥塞控制机制。这是因为入口帧会被放在NIC的内存内，或者放在驱动程序所控管的接收环（receive ring）内，因此内核无法追踪流量拥塞情况。相反，当设备驱动程序没使用NAPI时，帧会被添加到各个CPU队列(softnet_data->input_pkt_queue)，而内核就会追踪队列的拥塞等级。本节我们讨论后者的情况：

队列理论是很复杂的主题，本书不讨论这种数学细节，只要简单的要点就能令我满意：在队列中当前帧的数目并不一定代表真实的拥塞等级。平均队列长度是队列状态的较好的导向。记录平均值可以使系统避免错误地把突发的流量归类为拥塞。在Linux网络协议栈中，平均队列长度是由softnet_data结构的两个字段报告：cng_level和avg_blog，第九章“softnet_data结构”一节对此已做过介绍。

avg_blog是平均值，随时都可能比input_pkt_queue的长度更大或更小。avg_blog代表历史，而input_pkt_queue代表当前情况。因此，这两个变量用于两种不同的目的：

- 在默认情况下，每次一个帧排入input_pkt_queue时，avg_blog就会更新，而相关联的拥塞等级也会计算出来，存储至cng_level。netif_rx的返回值就是cng_level，使得调用netif_rx的设备驱动程序可以得到有关队列状态的返馈信息，然后据此改变其行为。
- input_pkt_queue中帧的数目不能超过最大值。到达最大值时，后续的帧就会被丢弃，因为CPU显然已经不能承受了。

我们回到拥塞等级的计算和使用。avg_blog和cng_level会在由netif_rx调用的get_sample_stats中得到更新。

此时，很少有设备驱动程序使用来自于netif_rx的反馈信息。这类反馈信息的最常见用法，就是用于更新设备驱动程序的本地统计数据。有关反馈信息的有趣应用可参考*drivers/net/tulip/de2104x.c*：当netif_rx返回NET_RX_DROP时，局部变量drop会置为1，使得主要循环开始丢弃接收环内的帧，而不予以处理。

只要入口队列input_pkt_queue没满，设备驱动程序就得使用来自于netif_rx的反馈信息以处理拥塞情况。当情况变得更糟而输入队列填满时，内核就得干预，使用softnet_data->throttle标识以CPU关闭帧的接收（记住，每个CPU都有一个softnet_data结构）。

netif_rx中的拥塞管理

让我们回到netif_rx，看一些前一节省略掉的代码。下面摘录的两段代码包括了先前显示过的代码以及显示出CPU何时会进入窒息状态的新代码。

```
if (queue->input_pkt_queue.qlen <= netdev_max_backlog) {
    if (queue->input_pkt_queue.qlen) {
        if (queue->throttle)
            goto drop;
        ... ... ...
        return queue->cng_level;
    }
    ... ...    ...
}

if (!queue->throttle) {
    queue->throttle = 1;
    __get_cpu_var(netdev_rx_stat).throttled++;
}
```

当队列为空时，softnet_data->throttle会被清除。正确地讲，当第一个帧排入空队列时，softnet_data->throttle就会被netif_rx清除。此事也会发生在process_backlog中，详情到了“积压的处理：process_backlog轮询虚拟函数”一节就会知道。

平均队列长度及拥塞等级计算

avg_blog和cng_level的值总是会在get_sample_stats中更新。而get_sample_stats可以用两种不同的方式启用：

- 每次接收一个新帧时（netif_rx）。这是默认方式。
- 通过周期性的定时器。要使用这种技术，必须定义OFFLINE_SAMPLE符号。这也就是为什么在netif_rx中get_sample_stats的执行依赖于OFFLINE_SAMPLE符号的定义。默认是关闭的。

在中等和高流量负载下，以第一种方法执行get_sample_stats比第二种方法更常见。

就这两种情况而言，用于计算avg_blog的公式都应该又简单又快速，因为公式时常会被调用。所使用的公式把新近的历史和当前情况都考虑进来了：

```
new_value_for_avg_blog = (old_value_of_avg_blog + current_value_of_queue_len) / 2
```

当前和过去的权重该怎么分配，不是简单的问题。上述公式可以很快就对拥塞等级有所调整，因为过去（旧值）只有50%的权重，而当前则占有另外50%。

get_sample_stats也会更新cng_level，其值源于avg_blog，再通过第九章中图9-4所示的对应关系而得知。如果RAND_LIE符号有定义，此函数会多执行一个操作，以随机决定把cng_level升高一级。这种随机调整需要更多时间去计算，但奇怪的是，这样可以使内核在一种特定场合下执行得更好。

我们花点时间说明随机的优点。但是，不要和RED（Random Early Detection，早期随机侦测）混淆了。

在只有一个接口的系统中，如果没有拥塞问题，随机丢弃帧实在没什么意义，只会让吞吐量（throughput）变低而已。但是，假设我们有多个接口共享一个输入队列，而且有一个设备的流量负载比其他的都高很多。因为贪心的设备会以比其他设备更快的速度填入共享的入口队列，因此，其他设备通常会发现入口队列没有空间，所以它们的帧就会被丢弃（注8）。贪婪的设备也会看到他的一些帧被丢弃，但是，与其负载相比并不成比例。当拥有多接口的系统遇到拥塞问题时，就应该按设备负载的比例丢弃输入帧。用在这种情境下时，RAND_LIE就能增添一些公平性：随机丢弃额外的帧，最后就应该能得到按负载比例丢弃帧的结果。

处理NET_RX_SOFTIRQ：net_rx_action

net_rx_action是用于处理进来的帧的下一个函数。其执行的触发是当驱动程序通知内核有关输入帧存在的时候。图10-5所示是此函数的控制流程。

帧可以在两个地方等待net_rx_action予以处理：

一个共享的CPU专用队列

非NAPI设备的中断处理例程，调用netif_rx能把帧放入CPU的softnet_data->input_pkt_queue，中断处理函数在此运行。

设备内存

NAPI驱动程序所用的poll方法会直接从设备（或设备驱动程序的接收环）中取出帧。

注8：共享一个队列时，用户彼此之间的行为必须公平对待，但是不一定都可能。NAPI不会遇到这种问题，因为每个使用NAPI的设备都有自己的队列。然而，非NAPI驱动程序仍然使用共享的输入队列input_pkt_queue，因此，就得面临其他设备超载的可能性。

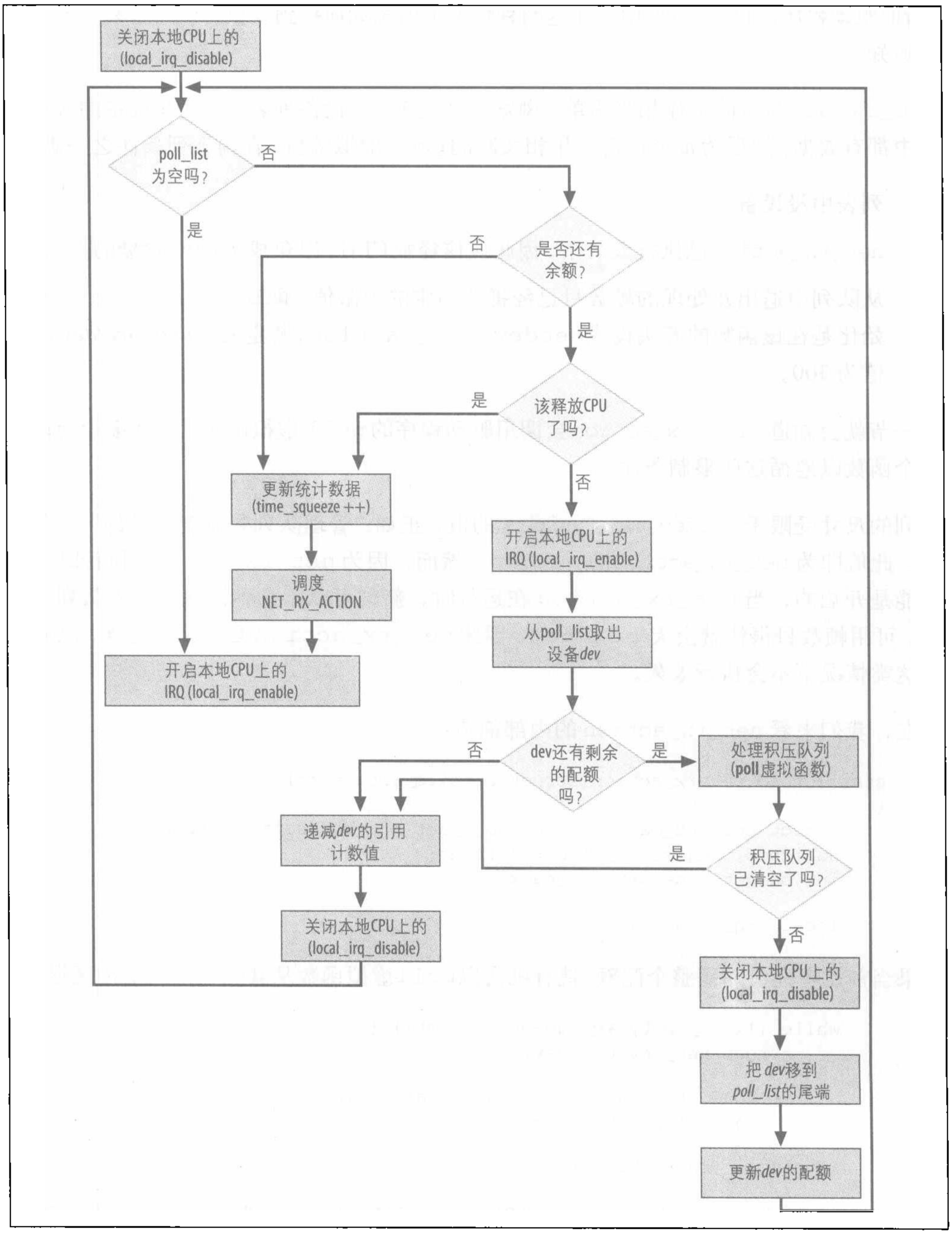


图 10-5：net_rx_action 函数

“新旧驱动程序接口”一节说明了在这两种情况下内核如何接到要执行net_rx_action的通知。

net_rx_action的工作相当简单：浏览poll_list设备列表，而这些设备的入口队列中都有数据，然后为每个设备启用相关联的poll虚拟函数，直到下列条件之一满足。

- 列表中没设备了。
- net_rx_action已执行太久了，因此应该释放CPU，以免成为CPU贪婪的占用者。
- 从队列中退出及处理的帧数目已经抵达指定的上限值，即budget。budget的初始化是在该函数的开头设成netdev_max_backlog，其定义在*net/core/dev.c*中，值为300。

下一节就会知道，net_rx_action会调用驱动程序的poll虚拟函数，而且多少会依赖这个函数以遵循这些限制条件。

队列的尺寸受限于netdev_max_backlog的值，正如“管理队列和调度下半部”一节所示。此值即为net_rx_action的budget。然而，因为net_rx_action执行时中断功能是开启的，当net_rx_action在运行时，新帧可以添加到设备的输入队列。因此，可用帧数目很快就会大于budget，因此net_rx_action必须采取行动，以确保在这类情况下不会执行太久。

现在，我们来看net_rx_action的内部细节：

```
static void net_rx_action(struct softirq_action *h)
{
    struct softnet_data *queue = &_ _get_cpu_var(softnet_data);
    unsigned long start_time = jiffies;
    int budget = netdev_max_backlog;

    local_irq_disable();
```

如果当前设备还没用到整个配额，就有机会以poll虚拟函数从其队列中将缓冲区退出：

```
    while (!list_empty(&queue->poll_list)) {
        struct net_device *dev;

        if (budget <= 0 || jiffies - start_time > 1)
            goto softnet_break;

        local_irq_enable();

        dev = list_entry(queue->poll_list.next, struct net_device, poll_list);
```

如果因为设备配额不够大，无法把入口队列中所有缓冲区都退出，使得dev->poll返回（此时，传回值为非零），则该设备会被移到poll_list的尾端：

```
if (dev->quota <= 0 || dev->poll(dev, &budget)) {
    local_irq_disable();
    list_del(&dev->poll_list);
    list_add_tail(&dev->poll_list, &queue->poll_list);
    if (dev->quota < 0)
        dev->quota += dev->weight;
    else
        dev->quota = dev->weight;
} else {
```

当代替poll设法清空设备入口队列时，net_rx_action不会从poll_list中把该设备删除掉：poll应该调用netif_rx_complete（如果本地CPU的IRQ被关闭，也可以调用__netif_rx_complete）负责此件事。这一点会在下一节介绍process_backlog函数时说明。

此外，注意到budget是以引用方式传给poll虚拟函数，这是因为该函数会返回一个新的budget，以反映已处理的帧。net_rx_action中的主要循环每一轮都会检查budget，以确保没有超过整体限制的值。换言之，budget使net_rx_action和poll函数彼此合作，使其不超出其限度。

```
                dev_put(dev);
                local_irq_disable();
            }
        }
out:
        local_irq_enable();
        return;
```

当net_rx_action被迫返回，而入口队列中仍然留有缓冲区时，就会执行最后一段代码。在这种情况下，NET_RX_SOFTIRQ软IRQ会再度被调度准备执行，稍后将启用net_rx_action，负责剩余的缓冲区：

```
softnet_break:
    __get_cpu_var(netdev_rx_stat).time_squeeze++;
    __raise_softirq_irqoff(NET_RX_SOFTIRQ);
    goto out;
}
```

注意，只有在操作设备列表poll_list做轮询时（也就是访问其softnet_data结构实体时），net_rx_action才会用local_irq_disable关闭中断功能。netpoll_poll_lock和netpoll_poll_unlock（NETPOLL功能所用）已被省略。如果你可以看到内核源码，可以参见*net/core/dev.c*中的*net_rx_action*查看其细节。

积压的处理：process_backlog 轮询虚拟函数

net_device 数据结构的 poll 虚拟函数会由 net_rx_action 执行，以处理设备的积压队列，对那些不使用 NAPI 的设备而言，其初始化值默认在 net_dev_init 中设置为 process_backlog。

内核 2.6.12 版本时，只有少数设备驱动程序使用 NAPI，而且以其自己的一个函数的指针作为 dev->poll 初始化的值：在 *drivers/net/tg3.c* 中的 Broadcom Tigon3 Ethernet 驱动程序是第一个采用 NAPI 的，因此是考察的绝佳范例。本节我们要分析定义在 *net/core/dev.c* 的默认处理函数 process_backlog，其实现方式非常类似使用 NAPI 的设备驱动程序的 poll 方法（例如，可以拿 process_backlog 和 tg3_poll 比较）。

然而，因为 process_backlog 能管理一大群共享同一个入口队列的设备，有一个重要的差异必须要考虑。当 process_backlog 运行时，硬件中断是开启的，所以此函数可以被先占。因此，访问 softnet_data 结构时总是用 local_irq_disable 关闭本地 CPU 的中断功能，使访问受到保护，特别是调用 __skb_dequeue 的时候。此锁对使用 NAPI 的设备驱动程序而言并不需要（注 9）：当其 poll 方法被启用时，会为该设备关闭硬件中断功能。此外，每个设备都有其自己的队列。

来看 process_backlog 的主要部分，如图 10-6 所示是其流程图。

此函数一开始先做一些初始化工作：

```
static int process_backlog(struct net_device *backlog_dev, int *budget)
{
    int work = 0;
    int quota = min(backlog_dev->quota, *budget);
    struct softnet_data *queue = &_ _get_cpu_var(softnet_data);
    unsigned long start_time = jiffies;
```

然后进入主循环，试图把输入队列中的所有缓冲区都退出，而且只有当下列条件之一满足时才会打断：

- 队列成空。
- 设备的配额已用尽。
- 此函数已执行太久了。

注 9：　因为每个 CPU 都有自己的 softnet_data 实例，不需要额外的上锁机制以负责 SMP。

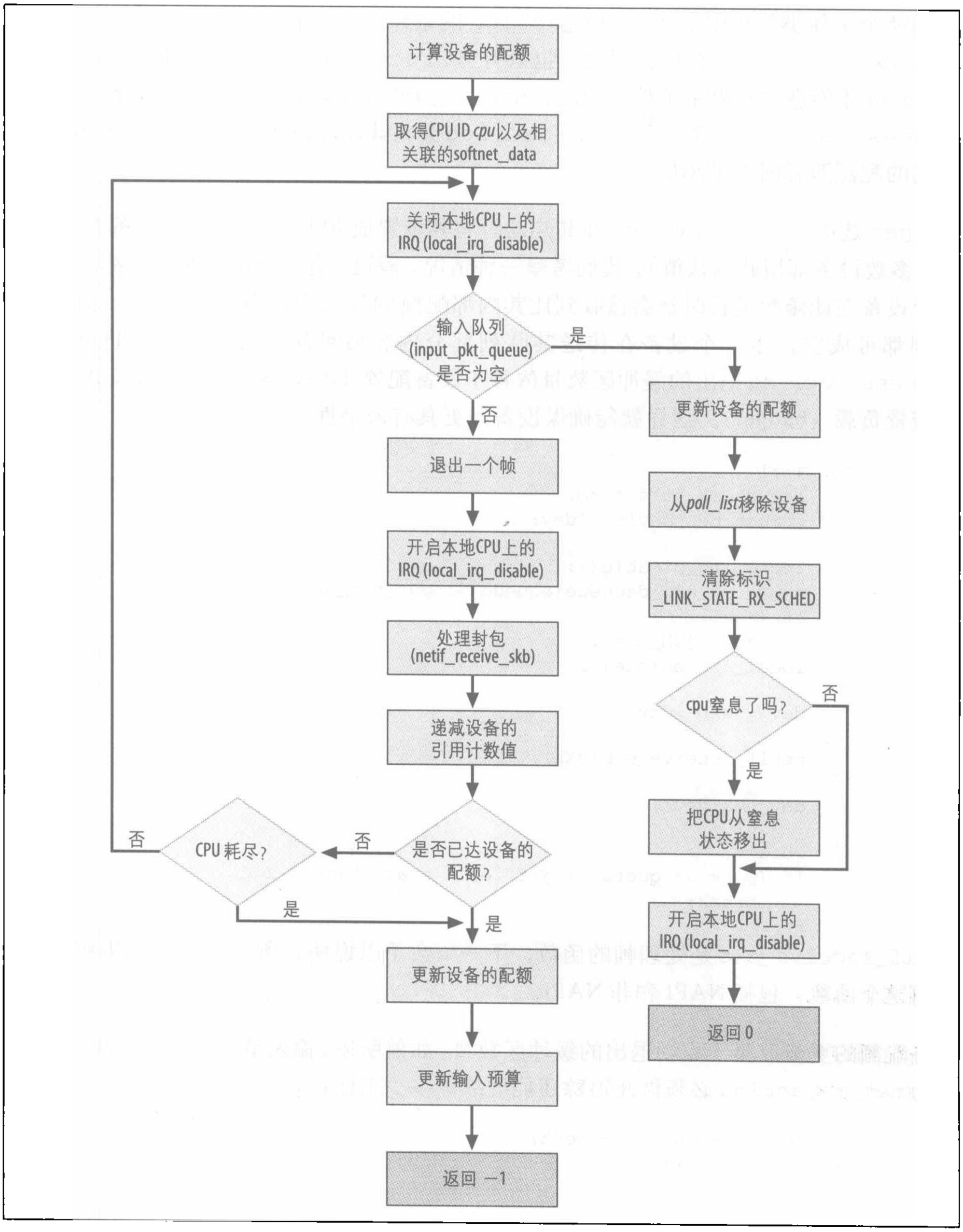


图 10-6：process_backlog 函数

后面两个条件类似于限制net_rx_action的条件。因为process_backlog是在net_rx_action中的循环调用的，也只有当process_backlog配合时，net_rx_action才能遵守其限制条件。因此，net_rx_action会把其所剩下的budget传给process_backlog，而process_backlog会将其配额设成该输入参数(budget)与自己的配额两者间的最小值。

budget是由net_rx_action在其启动时将其设置成300。dev->quota的默认值是64（多数设备都用此默认值）。我们考察一种情况，假设，有多个设备的队列都满了。前四个设备在此函数运行时都会接收到比其内部配额64还要大得多的budget，因此，其队列都可成空。下一个设备在传送其队列部分内容后可能就得停止了。也就是说，process_backlog退出的缓冲区数目依赖于设备配置（dev->quota）以及其他设备的流量负载（budget）。这样就能确保设备间更具有公平性。

```
for (;;) {
    struct sk_buff *skb;
    struct net_device *dev;

    local_irq_disable();
    skb = _ _skb_dequeue(&queue->input_pkt_queue);
    if (!skb)
        goto job_done;
    local_irq_enable();

    dev = skb->dev;

    netif_receive_skb(skb);

    dev_put(dev);

    work++;
    if (work >= quota || jiffies - start_time > 1)
        break;
```

netif_receive_skb是处理帧的函数，下一节会予以说明。所有poll虚拟函数都会用到这个函数，包括NAPI和非NAPI。

设备配额的更新取决于成功退出的缓冲区数目。如前所述，输入参数budget也会更新，因为net_rx_action必须借此追踪还能继续做多少工作：

```
backlog_dev->quota -= work;
*budget -= work;
return -1;
```

稍早所示的主循环在输入队列成空时，会跳转到标号job_done。如果此函数到达此处，窒息状态就可清除（如果被设置），而且设备也可从poll_list删除。__LINK_STATE_RX_SCHED标识也会清除，因为设备在输入队列中已无数据，因此不需要再次调度做积压的处理。

```
job_done:
    backlog_dev->quota -= work;
    *budget -= work;

    list_del(&backlog_dev->poll_list);
    smp_mb_ _before_clear_bit();
    netif_poll_enable(backlog_dev);

    if (queue->throttle)
        queue->throttle = 0;
    local_irq_enable();
    return 0;
}
```

实际上，process_backlog和NAPI驱动程序的poll方法之间有另外的一个差异。我们以 *drivers/net/tg3.c* 作为范例：

```
if (done) {
        spin_lock_irqsave(&tp->lock, flags);
    __netif_rx_complete(netdev);
    tg3_restart_ints(tp);
        spin_unlock_irqrestore(&tp->lock, flags);
}
```

这里的done是process_backlog中job_done的对称物，其意义相同，都是指队列成空。此时，在NAPI驱动程序中，__netif_rx_complete（定义在相同文件）函数会从poll_list中把设备删除，此任务是process_backlog直接所做的事。最后，NAPI驱动程序会为该设备重新开启中断功能。如本节开始所预期的，在process_backlog执行时中断功能是开启的。

入口帧的处理

如前节所述，netif_receive_skb是辅助函数，poll虚拟函数用其处理入口帧，如图 10-7 所示。

L2和L3都允许多种协议。每个设备驱动程序都会与一个特定的硬件类型相关联（例如Ethernet），所以，解读L2报头并获取信息，使其知道正在用的L3协议为何（参见第十三章）就很简单了。当net_rx_action启用时，L3协议标识符已由设备驱动程序从L2报头中提取而且存入skb->protocol中。

netif_receive_skb的三个主要任务是：

- 把帧的副本传给每个协议分流器，如果正在运行的话。
- 把帧的副本传给skb->protocol所关联的L3协议处理函数（注 10）。

注 10：　参见第十三章有关协议处理函数的细节。

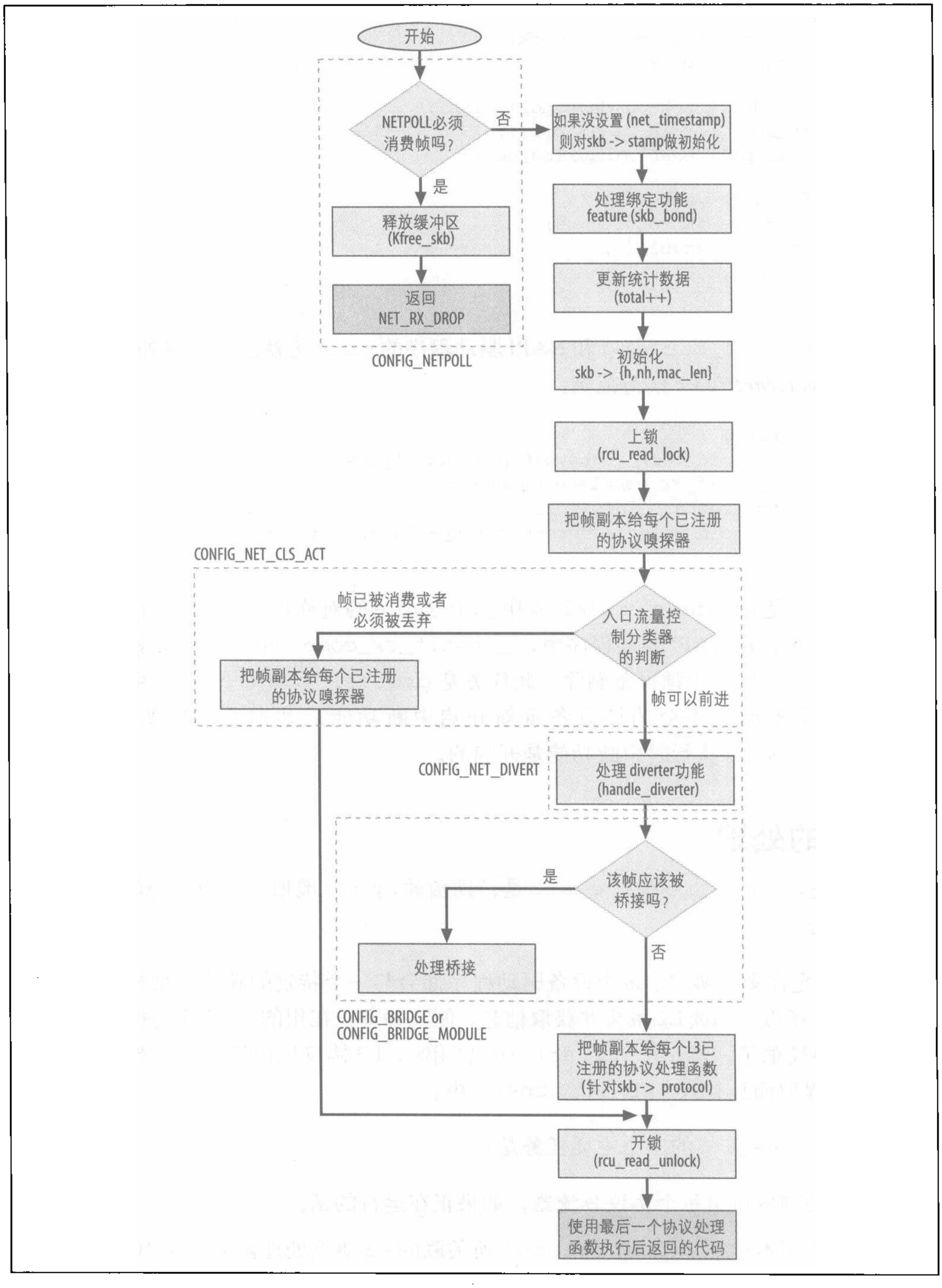


图 10-7：netif_receive_skb 函数

- 负责此层必须处理的一些功能，例如桥接（本书第四部分将描述）。

如果没有协议处理例程和skb->protocol关联，而且netif_receive_skb中的处理功能（如桥接）都没有消化该帧，该帧就会被丢弃，因为内核不知道该如何处理。

把输入帧传递给那些协议处理函数前，netif_receive_skb必须处理一些会改变帧命运的功能。

Bonding（绑定）可让一群接口组合起来，视为单一接口。如果帧来自的接口属于这类群组，则sk_buff数据结构中所引用的接收接口必须改成该群组中具有主设备角色的那个设备，netif_receive_skb才可以把封包传递给L3处理例程。这就是skb_bond的目的。

```
skb_bond(skb);
```

帧传递给嗅探器（sniffer）以及协议处理例程的细节，将在第十三章讨论。

一旦所有协议嗅探器都接收到封包副本，而且在真实协议处理函数取得副本前，Diverter、入口的流量控制以及桥接功能都必须予以处理（参见下一节）。

如果桥接代码或入口的流量控制代码都没消化该帧，则该帧会传给L3协议处理函数（通常每种协议只有一个处理函数，但是也可以注册多个）。在旧版本内核中，这就是唯一需要做的处理工作。但是，随着内核网络协议栈的增强，更多的功能添加进来（在这一层及其他层），封包经过网络协议栈的路径就变得越来越复杂。

此时，接收部分已完成，下一步就是L3协议处理函数决定要对封包如何处理：

- 传给接收工作站中正在执行的接收者（应用程序）。
- 丢弃（例如，健康检查失败）。
- 转发。

最后一项选择对路由器而言是很常见的，但是对单接口工作站就不常见了。本书第五部分和第六部分会详细讨论L3的行为。

内核可从L3的目的地地址确定该封包是否是传给其本地系统。这个过程到本书第七部分再讨论，此时就视其为理所当然。如果封包要给本地系统，封包就会传递到上层去（即TCP、UDP、ICMP等等），如果不是给本地系统，就会传给ip_forward（参见第九章图的9-2）。

以上就结束了帧接收过程的冗长讨论。下一章会说明帧的传输。这第二条路径包括本地产生的帧以及接收后需要转发的帧。

处理特殊功能

netif_receive_skb会检查是否有任何Netpoll客户想消化该帧。

流量控制一直都用于实现出口路径上的QoS。然而，就新近版本的内核而言，也可以在入口流量上配置过滤器并有所动作。根据这样的配置，ing_filter可以决定输入缓冲区是要被丢弃，还是要在某处做进一步处理（也就是消化该帧）。

diverter允许内核改变原本发往其他主机帧的L2目的地地址，使得帧可以改道发往本地主机。这种功能有很多可能的用法，如网站*http://diverter.sourceforge.net*所做的讨论。内核可以配置diverter所用的测定准则，以决定是否让帧改道。diverter所用的常见准则如下：

- 所有IP包（不管L3协议）。
- 所有TCP包。
- 特定端口号的TCP包。
- 所有UDP包。
- 特定端口号的UDP包。

调用handle_diverter可决定是否改变目的地的MAC地址。除了改变目的地的MAC地址外，skb->pkt_type也必须改成PACKET_HOST。

但是，另一个L2功能也会影响帧的命运：Bridging（桥接）。L2的桥接与L3的路由是配对功能，对此本书第四部分会予以讨论。每个net_device数据结构都有一个指向类型为net_bridge_port数据结构的指针，而此结构是用于存储表示桥接端口所需的额外信息。当接口没有开启桥接功能时，其值为NULL。当一个端口配置成桥接端口时，内核只会看L2报头。在此情况下，内核唯一用到的L3信息就是与防火墙有关的信息。

因为net_rx_action代表的是设备驱动程序和L3协议处理例程间的边界，此函数中必须处理Bridging功能，并没有错。当内核支持桥接时，handle_bridge会初始化为一个函数，而由此函数检查该帧是否要交给桥接代码。当帧要交给桥接代码，而且桥接代码也予以消化时，handle_bridge就返回1。其他情况下，handle_bridge返回0，而且netif_receive_skb会继续处理帧skb。

```
if (handle_bridge(skb, &pt_prev, &ret));
    goto out;
```

第十一章

帧的传输

“传输”这一术语用于离开系统的帧，也许是因为被系统传送出去，或者是因为被转发出去。本章我们讨论帧传输的数据通路中所涉及的主要任务：

- 为设备开启和关闭帧的传输。
- 为设备调度以准备传输。
- 为下一帧调度以准备传输，也就是在设备出口队列中等待的那些帧。
- 传输本身（我们将检查主要函数）。

传输过程和第十章所谈的接收过程大部分都是对称的：NET_TX_SOFTIRQ传输软IRQ，与NET_RX_SOFTIRQ软IRQ配对，而net_tx_action与net_rx_action配对，诸如此类。因此，如果你已读过前一章，应该会发现这一章很容易理解。图11-1比较了调度设备以准备接收，以及调度设备以准备传输两者之间背后的逻辑。以下是一些相似性：

- poll_list是设备列表，因为其接收队列不为空，其中的设备会被轮询。output_queue是设备列表，其中的设备有信息要传输。poll_list和output_queue是第九章所介绍的softnet_data结构的两个字段。
- 只有开启的设备（__LINK_STATE_START标识设置的那些设备）才能接受调度以准备接收。只有开启传输功能的设备（__LINK_STATE_XOFF标识清除的那些设备）才可以接受调度以准备传输。
- 当设备接受调度准备接收时，其__LINK_STATE_RX_SCHED标识会设置。当设备接受调度准备传输时，其__LINK_STATE_SCHED标识会设置。

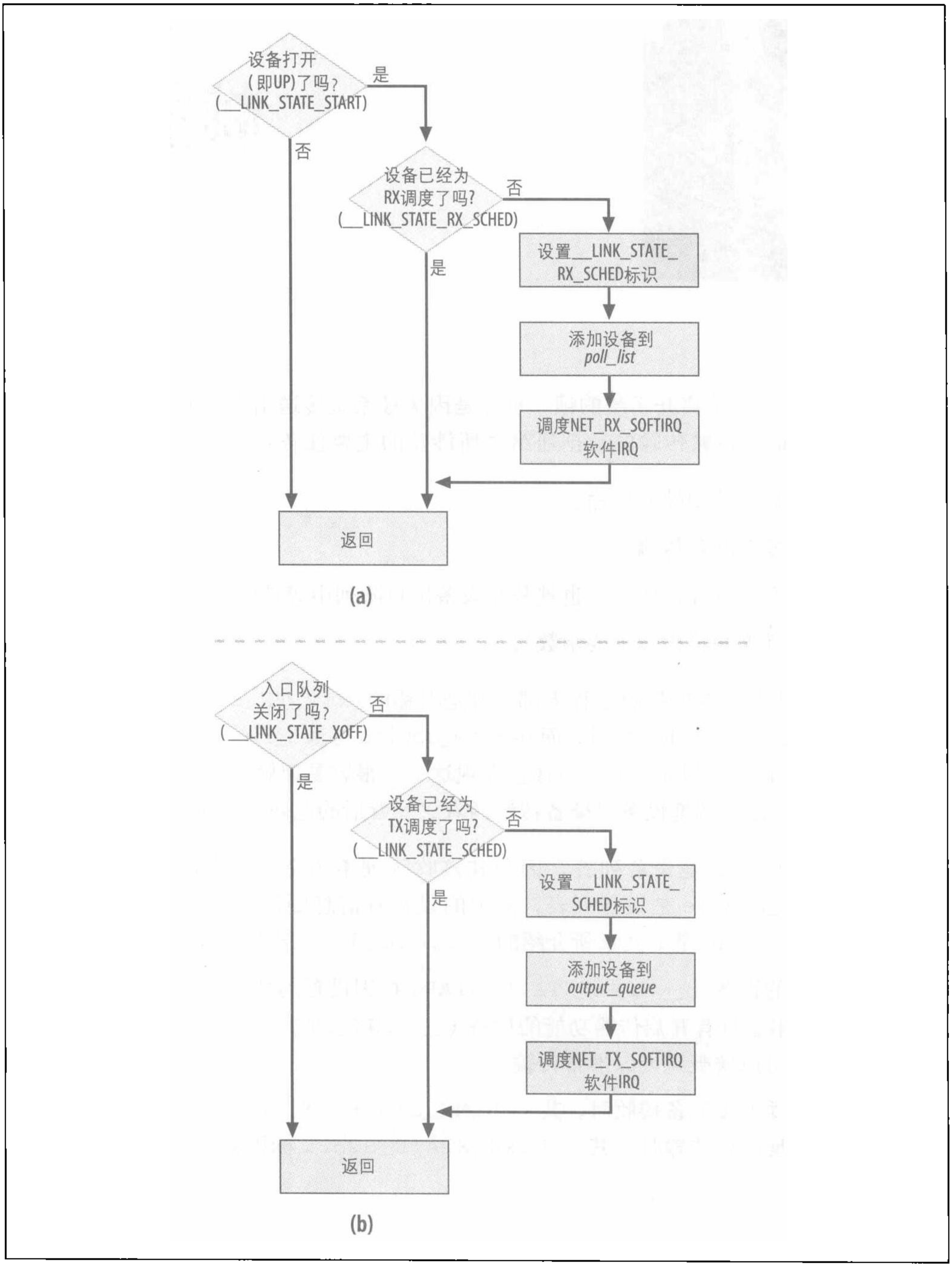


图 11-1：调度设备：(a) 准备接收 (RX)；(b) 准备传输 (TX)

dev_queue_xmit对出口路径的角色就如同netif_rx对入口路径一样，都是在驱动程序的缓冲区和内核的队列之间传输帧。当设备等待传输某些数据以及对那些不再需要的缓冲区做些清扫工作时，net_tx_action函数就会被调用。就像入口流量有队列可用一样，出口流量也有队列可用。出口队列（由流量控制处理，QoS 层），实际上比入口队列复杂许多：入口队列只是平常的FIFO队列，但出口队列是层次化的树形队列。虽然流量控制也支持入口队列操作功能，但是，使用流量控制主要是为了监督和管理的原因，而不是为了实际做队列的操作：流量控制对入口流量并没有使用真正的队列，而是只对动作进行分类，然后予以施行而已。

传输的开启和关闭

在第十章“拥塞管理”一节中我们已经知道，某些情况下帧的接收必须关闭，也许是单一设备，也许是全部的设备。对帧的传输而言，也有类似的情况。

出口队列的状态是由net_device->state中的标识__LINK_STATE_XOFF表示。其值可以通过定义在 *include/linux/netdevice.h* 中的下列函数操作和检查（注 1）：

netif_start_queue

开启设备的传输。当设备启动时通常就会调用此函数，此外，如果必须重启已停止的设备时，也可以再次调用。

netif_stop_queue

关闭设备的传输。任何企图在设备上传输信息的尝试都会被拒绝。本节稍后有一个此函数应用的常见实例。

netif_queue_stopped

返回出口队列的状态：开启或关闭。此函数很简单：

```
static inline int netif_queue_stopped(const struct net_device *dev)
{
    return test_bit(_ _LINK_STATE_XOFF, &dev->state);
}
```

只有设备驱动程序可以开启和关闭设备的传输。

设备运行时，为什么要停止和启动队列？原因之一是设备可能暂时用尽其内存，使得传输尝试失败。以前，传输函数（稍后“dev_queue_xmit 函数”一节会介绍）需要把帧再放回队列（重新排入队列）以解决这个问题。现在，由于__LINK_STATE_XOFF标识，这个额外的处理工作可以避开了。当设备驱动程序了解到没有足够空间以存储一个最大尺

注 1：　列表中的其他标识在第八章和第十章有相应的描述。

寸（MTU）的帧时，就会以netif_stop_queue停止出口队列。如此一来，就有可能避免后续的传输（内核已知会失败）而浪费资源。下面的窒息范例取自vortex_start_xmit（*drivers/net/3c59x.c*驱动程序所用的hard_start_xmit方法）：

```
outsl(ioaddr + TX_FIFO, skb->data, (skb->len + 3) >> 2);
dev_kfree_skb (skb);
if (inw(ioaddr + TxFree) > 1536) {
    netif_start_queue (dev);    /* AKPM: 多余的? */
} else {
    /* 当FTFO有空间可容纳最大尺寸的封包时,就打断我们。*/
    netif_stop_queue(dev);
    outw(SetTxThreshold + (1536>>2), ioaddr + EL3_CMD);
}
```

outsl传输之后，程序马上就会检查是否有空间容纳一个最大尺寸（1536）的帧，然后如果没有足够空间，就使用netif_stop_queue以停止设备的出口队列。这是相当粗糙的技术，用于避免因为内存不足造成的传输失败。当然，只要有300个字节以上的空间存在，就可传输一个300个字节的帧，因此，检查是否有1,536个字节可能就会造成不必要的传输关闭。程序也可以使用较低的值，如500，以作为折中，但是到头来并没有获得很大的利益，而且当传输开启却碰到较大帧来到时，反而会造成失败。

为了包含所有可能性，当设备上有足够内存时，程序会调用netif_start_queue。代码注释中的“多余的？”指的是两种中断事件而重启队列的事。当设备指出其已完成传输时，以及当设备指出其内存有足够空间可容纳一个帧时，驱动程序就会发出队列重启请求。如果驱动程序只因这些中断事件之一而发出重启请求时，队列可能会立刻重启，但是，并无法保证。所以，重启该队列的请求都是在这两种情况下都会发出。

代码也会传送SetTxThreshold命令给设备，指示该设备在特定内存量（此例就是MTU的量）可用时产生一个中断事件。

你可能会想，在前述情景中队列何时以及如何重启。就Vortex驱动程序而言，当特定量的内存可用时（就此而言就是MTU的量），就会要求设备产生一个中断事件。下面就是处理这种中断事件的代码：

```
static void vortex_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
        ... ... ...
    if (status & TxAvailable) {
        if (vortex_debug > 5)
            printk(KERN_DEBUG "    TX room bit was handled.\n");
        /* FTFO中有空间可容纳一个全尺寸的封包。*/
        outw(AckIntr | TxAvailable, ioaddr + EL3_CMD);
        netif_wake_queue (dev);
    }
        ... ... ...
}
```

status变量的那些位代表的是适配卡产生中断事件的原因。TxAvailable位指出空间足够，因此，唤醒设备是安全的（也就是唤醒队列，而且是由netif_wake_queue来做）。如EL3_CMD之类的值只是从ioaddr起算的偏移量，设备驱动程序借此在正确位置读写网卡寄存器。

注意，出口队列是用netif_wake_queue而不是用netif_start_queue重启。那个新函数（稍后会详谈）不仅开启出口队列，也会要求内核去检查该队列中是否有任何信息等待传输。原因在于，当队列被关闭时可能已有传输的尝试。就此而言，都会失败，而那些无法被传送的帧都会被放回出口队列。

设备调度以准备传输

说明入口路径时，我们知道当设备接收一个帧时，其驱动程序会启用一个内核函数（至于启用哪一个则依赖于驱动程序是否使用NAPI），把该设备添加到轮询列表中，然后调度NET_RX_SOFTIRQ以准备执行。

出口路径上所发生的事也很类似。要传输帧时，内核会提供dev_queue_xmit函数（稍后在其所在一节中将予以说明）。此函数会从设备的出口队列中退出一个帧，然后将该帧传递给设备的hard_start_xmit方法。然而，dev_queue_xmit可能因各种原因而无法传输，例如，因为该设备的出口队列已关闭（如前节所述），或者该设备队列的锁已被取走。为了处理后一种情况，内核提供了一个函数，名为__netif_schedule，可为设备调度以准备传输（有点类似netif_rx_schedule对接收路径所做之事）。此函数绝不会被直接调用，而是会通过两个包裹函数（本节稍后会谈及）。

以下是此函数的定义（来自于 *include/linux/netdevice.h*）：

```
static inline void _ _netif_schedule(struct net_device *dev)
{
    if (!test_and_set_bit(_ _LINK_STATE_SCHED, &dev->state)) {
        unsigned long flags;
        struct softnet_data *sd;

        local_irq_save(flags);
           sd = &_ _get_cpu_var(softnet_data);
        dev->next_sched = sd->output_queue;
        sd->output_queue = dev;
        raise_softirq_irqoff(cpu, NET_TX_SOFTIRQ);
        local_irq_restore(flags);
    }
}
```

__netif_schedule完成两项主要任务：

- 把设备添加到output_queue列表的头部。此列表是接收时所用的与poll_list配对的列表。每个CPU都有一个output_queue，就如同每个CPU都有一个poll_list。然而，output_queue会由NAPI及非NAPI设备所用，而poll_list只用于处理非NAPI设备。output_queue中的设备是以net_device->next_sched指针链接在一起的。在“处理NET_TX_SOFTIRQ：net_tx_action”一节将会知道这个列表如何使用。

  由第九章“softnet_data结构”一节已知，output_queue代表的是一个设备列表，其中的设备有数据要传送（因为先前的尝试失败了，如“队列规则接口”一节所述），或者其出口队列在被关闭一段时间之后又已重新开启。因为__netif_schedule可在中断环境内外被调用，把输入设备添加到output_queue列表时，就会关闭中断功能。

- 为NET_TX_SOFTIRQ软IRQ调度以准备执行。__LINK_STATE_SCHED用来标记位于output_queue列表中的设备，因为那些设备有数据要传输（__LINK_STATE_SCHED是接收路径的__LINK_STATE_RX_SCHED的配对标识）。注意，如果该设备已进入调度准备传输，__netif_schedule不做任何事情。

如果设备的传输功能已关闭，为设备调度以准备传输就没意义，因此，内核另外提供了两个函数以供使用，而这两个函数都是__netif_schedule的包裹函数：

netif_schedule（注2）

确保为设备调度进行传输之前该设备的传输功能已开启：

```
static inline void netif_schedule(struct net_device *dev)
{
    if (!test_bit(__LINK_STATE_XOFF, &dev->state))
        __netif_schedule(dev);
}
```

netif_wake_queue

开启该设备的传输，而且如果先前传输已经关闭，就为设备调度以准备传输。这种调度是必需的，因为该设备队列被关闭时可能已有传输的尝试。前一节已看到此函数的使用范例。

```
static inline void netif_wake_queue(struct net_device *dev)
{
    ...
    if (test_and_clear_bit(__LINK_STATE_XOFF, &dev->state))
        __netif_schedule(dev);
}
```

注2：为了一致性起见，netif_tx_schedule可能是比较好的名称。

test_and_clear_bit会清掉__LINK_STATE_XOFF标识（如果有设置），然后返回旧值。

注意，调用netif_wake_queue就相当于调用netif_start_queue和netif_schedule。在“传输的开启和关闭”一节已说过，驱动程序的责任（不是较高层的函数）是关闭和开启设备的传输功能。通常来讲，高层函数会为设备的传输进行调度，而设备驱动程序则在必要时关闭和重启队列，如为了应付内存不足的情况。因此，netif_wake_queue是由设备驱动程序使用，而netif_schedule则是用在其他地方——例如，由net_tx_action（注3）和流量控制使用——应该就不会令人诧异了。

驱动程序会在下列情况使用netif_wake_queue：

- 到了“看门狗定时器”一节就会知道，设备驱动程序会使用一个看门狗定时器，令挂起的传输得以恢复。在这种情况下，虚拟函数net_device->tx_timeout通常会重设适配卡。在设备不可用的那段黑洞期间内，可能有其他传输尝试，所以驱动程序必须先开启设备队列，然后为设备调度以准备传输。对那些通知错误情况的中断事件而言也是如此（参见*drivers/net/3c59x.c*中的范例）。
- 当设备通知驱动程序（驱动程序之前的请求）已有足够内存可处理一个特定尺寸的帧的传输时，该设备就可被唤醒。我们已在前一节谈到TxAvailable中断事件时看过这种范例。使用此函数的原因依然是在驱动程序关闭该队列的期间可能已有一些传输尝试。类似的考虑也用在一种中断事件类型上（当驱动程序对适配卡的DMA传输已完成时，就通知驱动程序）。

队列规则接口

几乎所有设备都会使用队列调度出口流量，而内核可以使用名为队列规则（queuing discipline）的算法安排帧，使其以最有效率的次序传输。虽然流量控制及其队列规则的详细讨论不在本书范围之内，但是，本节还是要对设备驱动程序以及本章讨论的传输层之间的接口做一些简介。

每种流量控制队列规则都提供各种函数指针，可由较高层调用，以完成各种任务。其中最重要的函数如下：

enqueue

把一个元素添加到队列中。

注3：net_tx_action无法取得设备出口队列的dev->queue_lock锁时，就无法传输，因此会为设备调度以准备传输。

dequeue

从队列中提取一个元素。

requeue

把一个原先已经提取的元素放回队列上（例如，因为传输失败）。

每当设备进入调度以准备传输时，qdisc_run 函数就会选出下一个要传输的帧，而该函数会间接调用相关联的队列规则的 dequeue 虚拟函数。

同样，真实的工作是由另一个函数所做，qdisc_restart。qdisc_run 函数（定义在 *include/linux/pkt_sched.h*）只是一个包裹函数，为那些输出队列已关闭的设备过滤请求而已：

```
static inline void qdisc_run(struct net_device *dev)
{
    while (!netif_queue_stopped(dev) && qdisc_restart(dev) < 0)
        /* 什么也没有 */;
```

qdisc_restart函数

前面已知设备进入调度以准备传输的常见情况。有时是因为出口队列中有数据等待传输，但其他时候，设备会被调度是因为其队列已被关闭了一段时间，因此有可能因先前失败的传输尝试使得队列中有数据在等待。驱动程序不知道是否真的已有任何数据抵达，但必须对设备进行调度，以免有数据在等待中。如果事实上并没有数据在等待，则后续对dequeue方法的调用就会失败。即使有数据在等待，调用也可能失败，因为复杂的队列规则可能决定不传输任何数据。因此，qdisc_restart（定义在 *net/sched/sch_generic.c*）会根据 dequeue 方法的返回值而采取各种不同的行动。

```
int qdisc_restart(struct net_device *dev)
{
    struct Qdisc *q = dev->qdisc;
    struct sk_buff *skb;

    if ((skb = q->dequeue(q)) != NULL) {
```

一开始 dequeue 函数就会被调用。我们假设调用成功。传输一个帧需要取得两个锁：

- 保护队列的锁（dev->queue_lock）。这是由 qdisc_restart 的调用者（dev_queue_xmit）取得的。
- 驱动程序传输函数 hard_start_xmit 的锁（dev->xmit_lock）。此锁是由此函数管理。当设备驱动程序已实现其自己的上锁机制时，就会设置 dev->features 中的NETIF_F_LLTX标识（无锁传输功能）指出这一点，以通知那些上层没有必

要取得dev->xmit_lock锁。使用NETIF_F_LLTX可让内核在不需要取得dev->xmit_lock锁时就不用取得，借此让传输数据路境得以最优化。当然，如果队列是空的，也就没必要取得该锁。

注意，从队列退出一个缓冲区之后，qdisc_restart并没有立刻释放queue_lock，因为如果无法取得该驱动程序锁时，就得立刻将该缓冲区重新排入队列。当该函数取得驱动程序锁后，就会释放queue_lock，然后再返回到前面，会再次重新取得queue_lock。最后，dev_queue_xmit会负责予以释放。

当驱动程序不支持NETIF_F_LLTX，而且驱动程序的锁已经被取走时（也就是spin_trylock返回0），传输就会失败。如果qdisc_restart无法获取驱动程序锁，就表示另一个CPU正通过同一个设备进行传输。此时，因为qdisc_restart不想等待，所以唯一能做的事就是把帧放回队列，再为设备重新调度以准备传输。如果此函数运行的CPU和持有该锁的CPU是一同个，这表示侦测到了一个循环（也就是代码中有bug），那么帧就会被丢弃，否则就只是冲突而已。

```
            if (!spin_trylock(&dev->xmit_lock)) {
            collision:
                ...
                goto requeue;
            }
            ...
requeue:
        q->ops->requeue(skb, q);
    netif_schedule(dev);
```

一旦成功取得驱动程序锁后，就会放开队列锁，使得其他CPU可以访问队列。有时，没必要取得驱动程序锁，因为NETIF_F_LLTX有设置。无论哪种情况，qdisc_restart都可着手进行其真实的工作。

```
if (!netif_queue_stopped(dev)) {
    int ret;
    if (netdev_nit)
        dev_queue_xmit_nit(skb, dev);

    ret = dev->hard_start_xmit(skb, dev);
    if (ret == NETDEV_TX_OK) {
        if (!nolock) {
            dev->xmit_lock_owner = -1;
            spin_unlock(&dev->xmit_lock);
        }
        spin_lock(&dev->queue_lock);
        return -1;
    }
    if (ret == NETDEV_TX_LOCKED && nolock) {
        spin_lock(&dev->queue_lock);
```

```
                goto collision;
            }
        }
```

由前节已知，qdisc_run早已利用netif_queue_stopped检查过出口队列的状态，但是，这里qdisc_restart又检查了一次。第二次检查并非多余的。考虑这种情景：当qdisc_run调用netif_queue_stopped时，驱动程序的锁尚未被取走。到了该锁被取走时，另一个CPU可能传送了某种数据，而该适配卡可能已经没有缓冲区空间可用了。因此，netif_queue_stopped先前返回FALSE，但现在则会返回TRUE。

netdev_nit代表的是已注册的协议探嗅器数目。如果任何协议探嗅器已经注册，就会用dev_queue_xmit_nit把帧的副本递送给每个协议探嗅器(在第十章netif_receive_skb中做接收时也有类似的情况)。

最后，我们得启用设备驱动程序的虚拟函数以进行帧的传输。设备驱动程序提供的函数是dev->hard_start_xmit（每个设备在初始设置期间都会定义，参见第八章)。hard_start_xmit函数返回的NETDEV_TX_XXX值列在*include/linux/netdevice.h*。以下是qdisc_restart处理的结果：

NETDEV_TX_OK（注4)

传输成功。缓冲区尚未被释放（还没发出kfree_skb)。到“处理NET_TX_SOFTIRQ：net_tx_action”一节就会知道，驱动程序不会自行释放缓冲区，而是通过NET_TX_SOFTIRQ软IRQ要求内核做此事。比起每个驱动程序都自己来释放，这样的内存处理会比较有效率。

NETDEV_TX_BUSY

驱动程序发现NIC的传输缓冲池内没有足够的空间。侦测到这种情况时，驱动程序通常也会调用netif_stop_queue（参见“传输的开启和关闭”一节)。

NETDEV_TX_LOCKED

驱动程序被锁住了。此返回值只由支持NETIF_F_LLTX的驱动程序使用。

总之，当下列条件之一为真时，传输失败，而帧就必须被放回队列：

注4：NETDEV_TX_*XXX*这些值是内核2.6.9版本才刚刚引入的。引入这些值之前，hard_start_xmit函数在成功时返回0，而在错误时返回1（例如，如果NIC的内存中没有空间)。到目前为止，只有少数设备驱动程序更新成使用NETDEV_TX_*XXX*这些值(主要是那些支持NETIF_F_LLTX)，其他驱动程序还是直接使用0和1。

- 队列已关闭（netif_queue_stopped(dev)为真）。
- 另一个CPU持有驱动程序锁。
- 驱动程序失败（hard_start_xmit没有传回NETDEV_TX_OK）。

参见图11-2了解有关disc_restart函数的细节。

dev_queue_xmit函数

此函数是设备驱动程序执行传输时的接口。如第九章中图9-2所示，dev_queue_xmit会导致驱动程序传输函数hard_start_xmit通过以下两种路径之一执行：

衔接至流量控制（QoS层）

这是通过上一节已说明过的qdisc_run函数实现的。

直接启用hard_start_xmit

这是为那些不使用流量控制基础架构的设备（也就是虚拟设备）所做的。

我们很快就会看到这些案例，但先从这两者都常见的检查和任务着手。

当dev_queue_xmit被调用时，传输帧所需的所有信息都已准备就绪，如外出设备，下一个跳点，以及其链接层地址。

dev_queue_xmit只接收一个sk_buff结构作为输入值。此结构包含此函数所需的一切信息。例如，skb->dev是外出设备，而skb->data指向有效载荷的开头，而其长度为
skb->len。

```
int dev_queue_xmit(struct sk_buff *skb)
```

dev_queue_xmit的主要任务是：

- 检查帧是否由一些片段（fragment）组成，以及设备是否能通过散播/聚集DMA的方式处理这些片段。如果设备无法这么做，就把这些片段结合起来。参见第二十一章有关片段缓冲区（fragmented buffer）的讨论。
- 计算以确认L4校验和（也就是TCP/UDP），除非设备在硬件内计算此校验和。参见第十八章有关校验和的细节。
- 选择要传输哪个帧（由input sk_buff所指的帧不一定就是要传输的帧，因为有一个队列要尊重）。

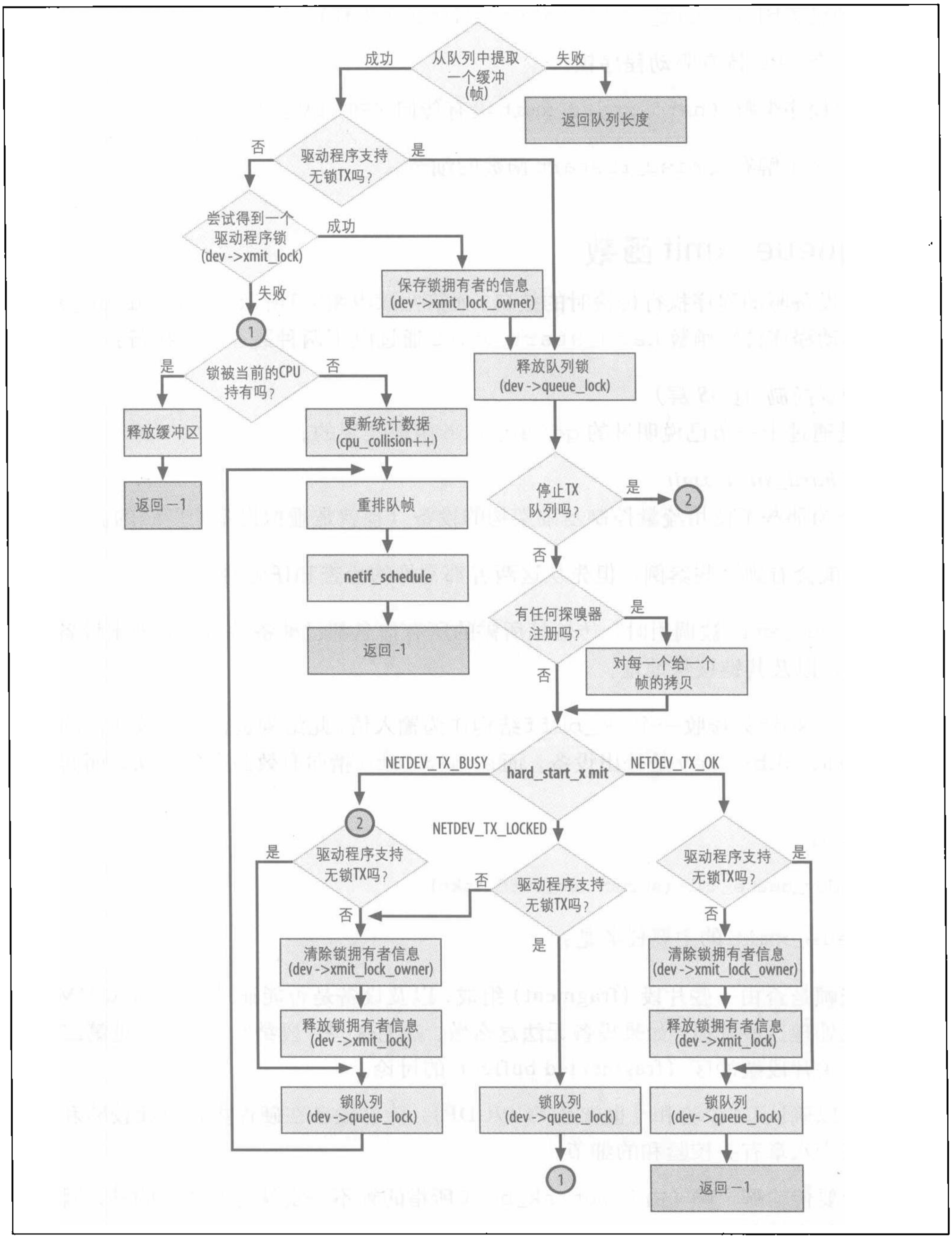


图 11-2：qdisc_restart 函数

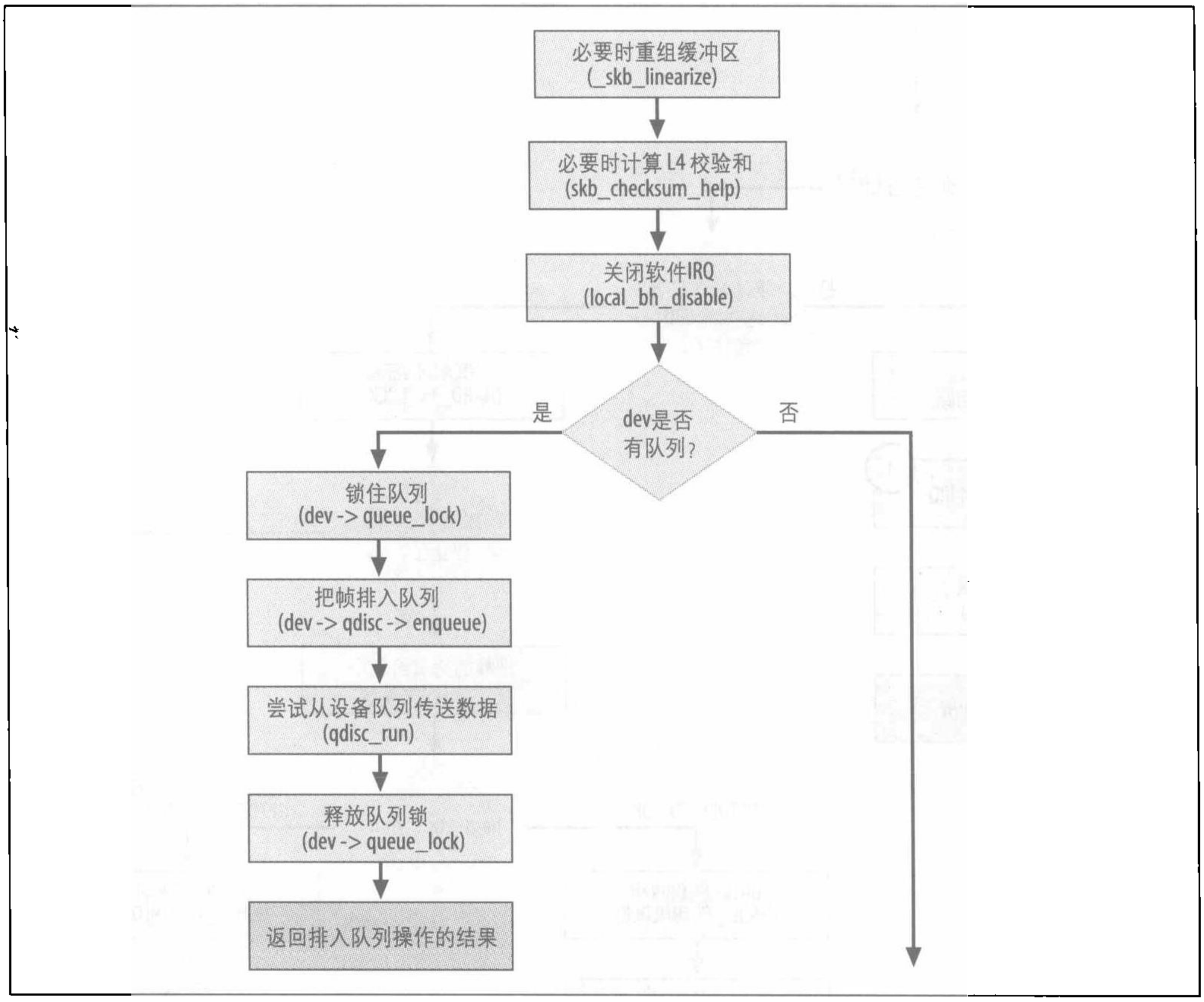


图 11-3（a）说明 dev_queue_xmit。

下列程序中，当 skb_shinfo(skb)->frag_list 不为 NULL 时，数据有效载荷就是一个片段列表；否则，有效载荷就是一个区块。如果有一些片段，则程序会检查设备是否支持散播/聚集DMA功能，如果不支持，就把那些片段结合成单一缓冲区。如果片段之中有任何片段存储所在的内存区域其地址过大，使得设备无法寻址（即 illegal_highdma(dev, skb) 为真），此函数也得把那些片段结合起来（注 5）。

```
if (skb_shinfo(skb)->frag_list &&
    !(dev->features&NETIF_F_FRAGLIST) &&
    __skb_linearize(skb, GFP_ATOMIC)) {
    goto out_kfree_skb;
}
```

注 5：　有些设备只能使用 16 位地址，因此，可寻址的内存会受限。

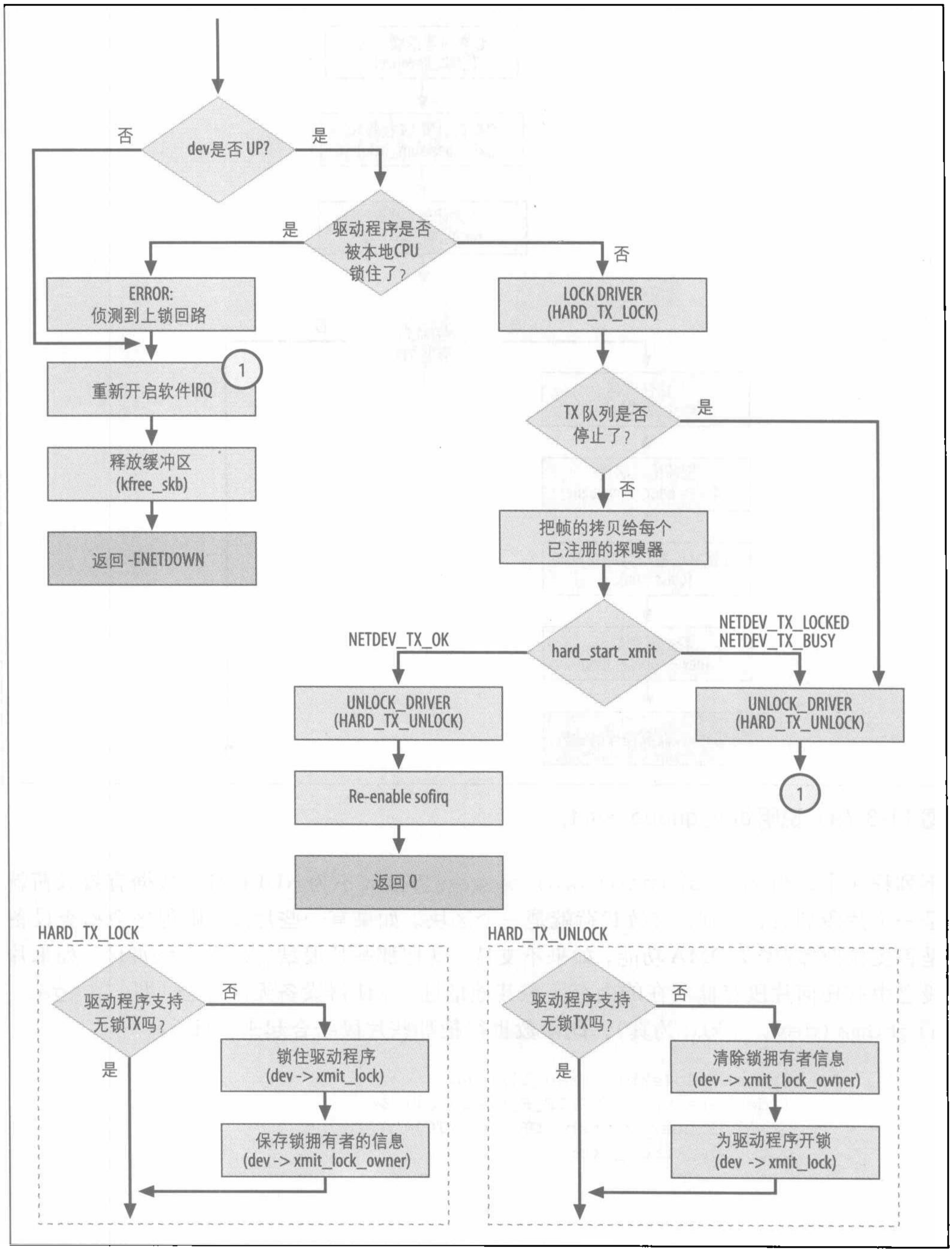


图 11-3（b）：说明 dev_queue_xmit

```
if (skb_shinfo(skb)->nr_frags &&
    (!(dev->features&NETIF_F_SG) || illegal_highdma(dev, skb)) &&
    __skb_linearize(skb, GFP_ATOMIC)) {
    goto out_kfree_skb;
}
```

片段的重组（defragmentation）是由__skb_linearize所做，但是，会因为下列原因之一而失败：

- 用于存储那些联结后的片段的新缓冲区无法得到分配。
- sk_buff缓冲区和其他子系统共享（也就是引用计数大于1）。就此而言，此函数并非实质失败，但是会调用BUG()以产生一条警示信息。

L4校验和可在软件和硬件中计算（注6）。并非所有网络卡都可在硬件内计算校验和。那些可以如此做的网络卡会在设备初始化期间设置net_device->features中相关的位标识。这样会通知较高层的网络层，它们不需要担心校验和计算的问题。但是，如果是下列情况之一，校验和就必须在软件内计算：

- 硬件不支持校验和计算。
- 适配卡只能针对底层是IP的TCP/UDP封包使用硬件校验和计算，但是，正在传输的封包不使用IP，或者使用其他底层是IP的L4协议。

软件校验和是由skb_checksum_help计算的：

```
if (skb->ip_summed == CHECKSUM_HW &&
      (!(dev->features & (NETIF_F_HW_CSUM | NETIF_F_NO_CSUM)) &&
       (!(dev->features & NETIF_F_IP_CSUM) ||
        skb->protocol != htons(ETH_P_IP))))
      if (skb_checksum_help(skb, 0))
          goto out_kfree_skb;
```

一旦校验和被处理后，所有报头也都准备就绪，下一步就是决定该传送哪个帧了。

此时，其行为依赖于设备是否使用流量控制基础架构，以及是否获派队列规则。是的，这有点令人惊讶。此函数只处理一个缓冲区（必要时，予以重组以及计算校验和），但是依赖于是否使用队列规则、使用哪种队列规则以及外出队列的状态，此缓冲区可能不是下一个实际传送的缓冲区。

有队列设备

当设备的队列规则存在时，就可通过dev->qdisc予以访问。输入帧会通过enqueue

注6：　每种协议用于计算校验和的算法将在相关各章中说明。

虚拟函数而排入队列，接着，一个帧会通过qdisc_run而退出队列并传输，其细节在“队列规则接口”一节已经描述。

```
local_bh_disable();
q = rcu_dereference(dev->qdisc);
...
if (q->enqueue) {
    spin_lock(&dev->queue_lock);

    rc = q->enqueue(skb, q);

    qdisc_run(dev);

    spin_unlock_bh(&dev->queue_lock);
    rc = rc == NET_XMIT_BYPASS ? NET_XMIT_SUCCESS : rc;
    goto out;
}
```

注意，排入队列以及退出队列都会由队列锁queue_lock保护。软IRQ也会用local_bh_disable关闭，作为RCU的需要local_bh_disable也会把先占功能关闭。

无队列设备

有些设备没有队列，如回环设备：每当一个帧传输时，就会立刻被传递出去（但是，因为没有队列可以让帧重新排入，如果有地方出错，帧就会被丢弃，没有第二次机会）。如果看一下*drivers/net/loopback.c*中的loopback_xmit，会发现在程序末尾直接调用netif_rx，跳过所有队列事务。由第十章已知，netif_rx是由非NAPI设备驱动程序所调用的API，可将送进来的帧放到输入队列，然后通知较高层有关此事件。由于回环设备没有输入队列，传输函数要完成两个任务：一端传输，而另一端接收，如图11-4所示。

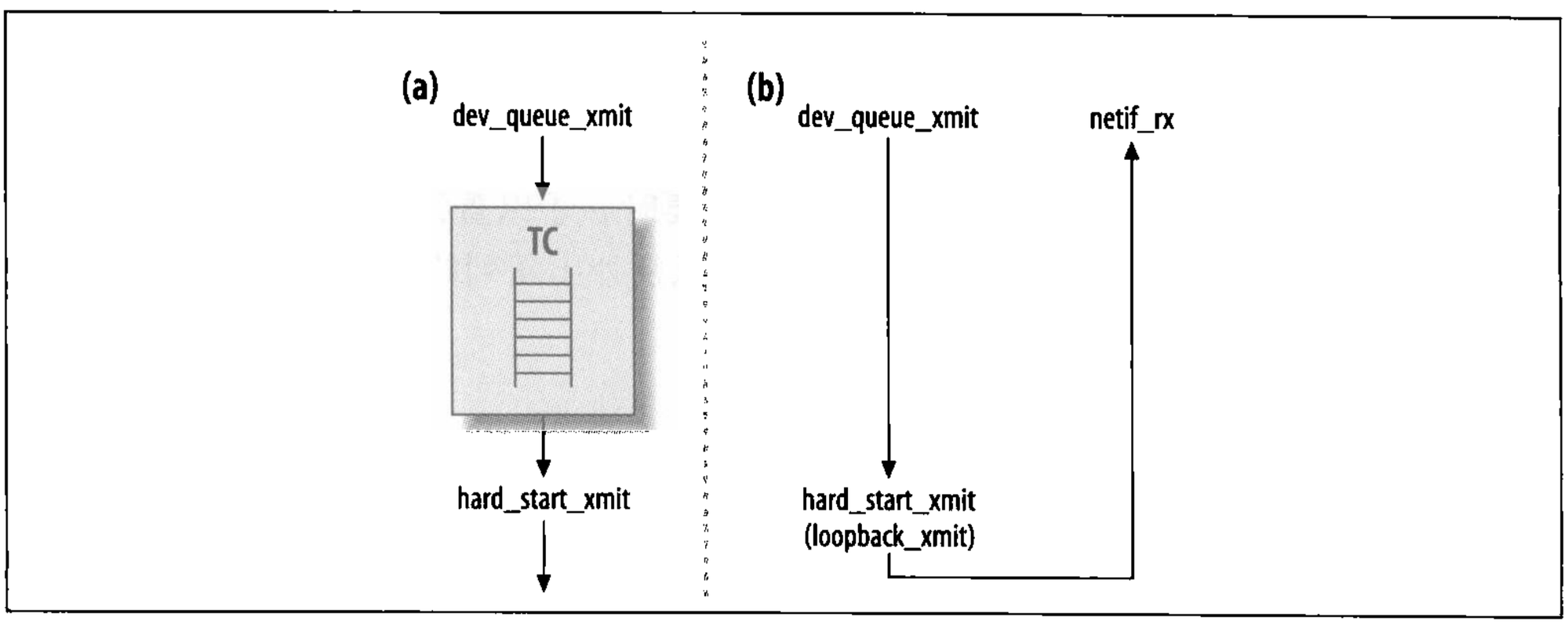


图11-4：（a）有队列设备传输；（b）回环传输

dev_queue_xmit的最后一部分用于处理没有队列规则的设备(因此也没有出口队列)。这部分相当类似“队列规则接口”一节所提到的qdisc_run的行为。然而，不使用队列时有两点差异在：

- 当传输失败时，驱动程序无法把缓冲区放回任何队列中，因为根本没有队列，所以dev_queue_xmit会丢弃该缓冲区。如果较高层使用的是可靠的协议，如TCP，则数据终究会被重新传输；否则，数据就会遗失。
- “qdisc_restart函数”一节介绍的NETIF_F_LLTX是由HARD_TX_LOCK和HARD_TX_UNLOCK这两个宏负责。HARD_TX_LOCK使用spin_lock而不是spin_trylock，当驱动程序锁被取走时，dev_queue_xmit就处于旋转状态，等待该锁被释放。

处理NET_TX_SOFTIRQ：net_tx_action

由第十章已知，net_rx_action函数是与NET_RX_SOFTIRQ软件中断相关联的处理函数。此函数由设备驱动程序触发（某些特定情况下会自行触发），负责设备驱动程序延期至“中断事件处理阶段之后”才进行的那一部分的输入帧的处理。如此一来，驱动程序在中断环境内所执行的程序就只会做绝对必要之事（把数据拷贝到内存，然后产生一个软件中断事件通知内核此事），因而不会强制系统其余部分等待太久，稍后，此软件中断会负责处理那部分可以等待的帧的处理。

net_tx_action的工作方式类似，此函数可由设备在两种情境下以raise_softirq_irqoff(NET_TX_SOFTIRQ)予以触发，以完成两项主要任务：

- 当设备的传输功能通过netif_wake_queue开启时，在这种情况下，此函数要确保当所有必需条件都吻合时（例如，当设备有足够内存时），等待被传送的帧实际上都被传送出去。
- 当传输已完成而且设备驱动程序通过dev_kfree_skb_irq通知相关联的缓冲区可释放时，在这种情况下，此函数要收回（deallocate）那些已成功传输的缓冲区的sk_buff结构。

第二项任务的原因如下。我们知道当来自设备驱动程序的程序在中断环境下运行时，就必需尽快执行。释放一个缓冲区会耗时间，所以会受到拖延，而要求net_tx_action软IRQ予以负责。设备驱动程序不使用dev_kfree_skb，而是使用dev_kfree_skb_irq。dev_kfree_skb收回sk_buff（实际上就是缓冲区，回到每CPU缓存内），但是，dev_kfree_skb_irq只会把要被释放的缓冲区的指针添加到与CPU相关联的softnet_data结构的completion_queue列表中，然后让net_tx_action稍后去做实际的工作。

让我们来看net_tx_action如何完成其两项任务。

首先，收回所有因设备驱动程序调用 dev_kfree_skb_irq 而被添加到 completion_queue列表的缓冲区。因为net_tx_action是在中断环境外运行的，设备驱动程序可以在任何时刻添加元素，所以，net_tx_action在访问softnet_data结构时必须关闭中断功能。为了尽可能让中断功能关闭时间短一点，net_tx_action 会把completion_queue设成NULL以清除该列表，然后把指向该列表的指针存储在一个局部变量clist（没人可以访问，此外每个CPU都有其自己的列表）。如此一来，net_tx_action就能遍历此列表，用__kfree_skb释放每个元素，同时驱动程序又能持续把新元素添加到 completion_queue。

```
if (sd->completion_queue) {
    struct sk_buff *clist;

    local_irq_disable();
    clist = sd->completion_queue;
    sd->completion_queue = NULL;
    local_irq_enable();

    while (clist != NULL) {
        struct sk_buff *skb = clist;
        clist = clist->next;

        BUG_TRAP(!atomic_read(&skb->users));
        __kfree_skb(skb);
    }
}
```

函数的另一半是传输帧，其工作方式也相似：使用一个局部变量，不让硬件中断受影响。注意，对每个设备而言，传输任何数据之前此函数都必须获取输出设备的队列锁(dev->queue_lock)。如果函数无法获取该锁（因为另一个CPU持有该锁），就只好以netif_schedule为设备重新调度以准备传输。

```
if (sd->output_queue) {
    struct net_device *head;

    local_irq_disable();
    head = sd->output_queue;
    sd->output_queue = NULL;
    local_irq_enable();

    while (head) {
        struct net_device *dev = head;
        head = head->next_sched;

        smp_mb_ _before_clear_bit();
        clear_bit(_ _LINK_STATE_SCHED, &dev->state);
```

```
            if (spin_trylock(&dev->queue_lock)) {
                qdisc_run(dev);
                spin_unlock(&dev->queue_lock);
            } else {
                netif_schedule(dev);
            }
        }
    }
```

在“队列规则接口”一节已知qdisc_run如何工作。设备是用循序从列表的头开始依次接收处理。因为netif_schedule函数(内部调用__netif_schedule)会把元素添加到列表的头，设备接收服务的次序就是FIFO，在某些情况下这可能不公平。

以上就是net_tx_action函数的说明。我们来看一些启用此函数以释放缓冲区的情况。有些想释放缓冲区的函数可以在各种不同的情况下启用（在中断环境内外）。有一个包裹函数可以用优雅的方式处理这些情况：

```
static inline void dev_kfree_skb_any(struct sk_buff *skb)
{
    if (in_irq() || irqs_disabled())
        dev_kfree_skb_irq(skb);
    else
        dev_kfree_skb(skb);
}
```

当调用函数处于中断环境下时，dev_kfree_skb_irq函数会执行，如下所示：

```
static inline void dev_kfree_skb_irq(struct sk_buff *skb)
{
    if (atomic_dec_and_test(&skb->users)) {
        struct softnet_data *sd;
        unsigned long flags;

        local_irq_save(flags);
           sd = &_ _get_cpu_var(softnet_data);
        skb->next = sd->completion_queue;
        sd->completion_queue = skb;
        raise_softirq_irqoff(NET_TX_SOFTIRQ);
        local_irq_restore(flags);
    }
}
```

只有当没有任何引用时，缓冲区才可被释放（也就是skb->users为0)。

我们看一个实例：设备驱动程序间接调用cpu_raise_softirq(cpu, NET_TX_SOFTIRQ)而触发net_tx_action的执行的过程（另一个实例可在“传输的开启和关闭”一节找到)。

前面介绍过的vortex_interrupt函数(*drivers/net/3c59x.c*)所处理的中断类型中，有一

种中断事件是由设备启用以通知驱动程序，CPU对设备的DMA传输已完成(DMADone)。由于缓冲区已传输给设备，sk_buff结构现在可以释放了。因为中断处理函数是在中断环境内工作的，驱动程序会调用dev_kfree_skb_irq。

```
if (status & DMADone) {
    if (inw(ioaddr + Wn7_MasterStatus) & 0x1000) {
        outw(0x1000, ioaddr + Wn7_MasterStatus); /* 通知收到此事件 */
        pci_unmap_single(VORTEX_PCI(vp), vp->tx_skb_dma,
                (vp->tx_skb->len + 3) & ~3, PCI_DMA_TODEVICE);
        dev_kfree_skb_irq(vp->tx_skb); /* 释放已传输的缓冲区 */
        if (inw(ioaddr + TxFree) > 1536) {
            netif_wake_queue(dev);
        } else { /* 当FIFO有空间容纳最大尺寸封包时就打断 */
            outw(SetTxThreshold + (1536>>2), ioaddr + EL3_CMD);
            netif_stop_queue(dev);
        }
    }
}
```

看门狗定时器

在“传输的开启和关闭”一节已知，当某些条件吻合时，设备驱动程序可以关闭传输。关闭传输应该是暂时的，所以当传输没有在合理时间内重启时，内核就假定该设备遇到了一些问题而应该重新启动。

这是由各个设备定时器(当设备以dev_activate启动时，由dev_watchdog_up启动)所完成的。此定时器会定期到期，以确保该设备一切没问题，然后自行重新启动。当侦测到问题(因为设备的输出队列已关闭；netif_queue_stopped返回TRUE)以及上次帧传输后已经过太多时间时，定时器的处理函数就会启用设备驱动程序所注册的函数，以复位NIC。

以下是用于实现此机制的net_device字段：

trans_start

这是由设备驱动程序设置的时间戳，记录上次帧传输启动的时间。

watchdog_timer

这是由流量控制启动的定时器。当定时器到期时，所执行的处理函数是dev_watchdog(定义在*net/sched/sch_generic.c*)

watchdog_timeo

这是要等待的时间量。此值由设备驱动程序初始化。当其设为0时，watchdog_timer就不会启动。

tx_timeout

这是设备驱动程序提供的函数，dev_watchdog将被启动以复位设备。

当定时器到期时，内核处理函数dev_watchdog会调用tx_timeout所指的函数而采取行动。tx_timeout会复位网络卡，然后以netif_wake_queue重启接口调度器。

watchdog_timeo的适当值依赖于接口而定。如果驱动程序没有设置，其默认值为5秒。定义此值时需要考虑的参数如下：

传输冲突的可能性

对点对点链接而言为零，但是对插入Hub的共享并超载的Ethernet链接而言，就可能很高。

接口速率

接口愈慢，时限就应该愈大。

watchdog_timeo的值通常定义为变量HZ（代表1秒）的倍数。HZ是全局变量，其值依赖于平台而定（定义在体系结构依赖的文件*include/asm-XXX/param.h*中）。如表11-1所示，每种相同类型的设备也可能采用不同的值作为时限。此表只列出一些范例，并不是完整的列表。

表11-1：常见网卡所用的传输时限

设备驱动程序	watchdog_timeo（时限）
3c501	HZ
3c505	10*HZ
3c509	(400*HZ)/1000
3c515	(400*HZ)/1000
3c523	HZ
3c527	5*HZ
3c59x	5*HZ
Dl2k	4*HZ
Natsemi	2*HZ
Aironet 4500	8*HZ
s2io (10Gbit)	5*HZ
8390	(20*HZ)/100
8139too	6*HZ
b44	5*HZ
Tg3	5*HZ
e100	2*HZ

表 11-1 常见网卡所用的传输时限（续）

设备驱动程序	watchdog_timeo（时限）
e1000	5*HZ
SIS 900	4*HZ
Tulip系列	4*HZ
Intel EtherExpress 16	2*HZ
SLIP	20*HZ

看门狗定时器机制是由流量控制程序所提供的。然而，高层设备驱动程序也可以实现自己的看门狗定时器。参见 *drivers/net/e1000_main.c* 中的范例。

第十二章

中断事件一般性参考数据

本章讨论几种一般性信息，适用于前面三章所涉及的中断事件和帧的处理。

统计数据

有关帧接收的统计数据保存在各个CPU数组netdev_rx_stat中，其元素为netif_rx_stats类型（参见 *include/linux/netdevice.h*）：

```
struct netif_rx_stats netdev_rx_stat[NR_CPUS];

struct netif_rx_stats
{
        unsigned total;
        unsigned dropped;
        unsigned time_squeeze;
        unsigned throttled;
        unsigned fastroute_hit;
        unsigned fastroute_success;
        unsigned fastroute_defer;
        unsigned fastroute_deferred_out;
        unsigned fastroute_latency_reduction;
        unsigned cpu_collision;
} ____cacheline_aligned;
```

netif_rx_stats的元素有：

total

已处理的入口帧总数，包括任何可能被丢弃的。此值会在netif_rx和netif_receive_skb中更新，也就是说，当驱动程序不使用NAPI时（也就是使用netif_rx接口，参见第十章的图10-2），同一个帧被计算两次（错误）。

dropped

因为接收时CPU处于窒息状态而被丢弃的帧的数目。

time_squeeze

当帧依然在CPU入口队列中，因此没有变成CPU的贪婪者时，net_rx_action返回的次数。参见第十章“处理NET_RX_SOFTIRQ：net_rx_action”一节。

throttled

CPU进入窒息状态的次数。此值由netif_rx递增。

fastroute_hit

fastroute_success

fastroute_defer

fastroute_latency_reduction

fastroute_deferred_out

由Fastroute功能所用的一些字段。内核2.6.8版本已丢弃此功能。

cpu_collision

CPU无法获得设备驱动程序锁（明确讲就是dev->xmit_lock）的次数，因为该锁已被另一个CPU取走。这个计数器会在qdisc_restart中更新，而qdisc_restart只处理帧传输，而无接收。cpu_collision是此结构中唯一与传输有关的统计数据。

前述有些计数器目前都由netif_rx（由非NAPI驱动程序所用）更新，这表示使用NAPI驱动程序时，其值是不正确的。

netdev_rx_stat向量的内容可通过*/proc*接口查看。参见下一节。

其他统计数据则由驱动程序、较高层协议以及流量控制队列规则保存在私有数据结构中（参见第二章）。这些值有些可以通过用户空间应用程序读取，如*ifconfig*、*tc*、*ip*或*netstat*，而其他的则可通过*/proc*输出。

通过/proc和sysfs文件系统调整

表12-1列出了的所有在*/proc/sys/net/core*中的文件，都定义在*net/core/sysctl_net_core.c*，你可以找出文件与内核变量之间的关联性。

表 12-1：可用于调整帧的接收的 /proc/sys/net/core/ 文件

文件名	内核变量[a]	默认值
netdev_max_backlog	netdev_max_backlog	300
mod_cong	mod_cong	290
lo_cong	lo_cong	100
no_cong	no_cong	20
no_cong_thresh	no_cong_thresh	10
dev_weight	weight_p	64

a. 所有这些变量都定义在 *net/core/dev.c* 中。

我要强调的是，NAPI 驱动程序不需要表 12-1 中的任何字段。NAPI 驱动程序应该以驱动程序的局部变量值设置 net_device->weight，而不使用 weight_p。但是，如果 NAPI驱动程序想用weight_p，也可以使用，特别是它们通常都使用相同的默认值64。从内核2.6.12版本起，net_device的weight字段值可以在运行期间以 *sysfs* 通过各个设备文件*/sys/class/net/device_name/weight*进行调整。*Weight*文件是在*net/core/net-sysfs.c* 中创建的。

“统计数据”一节所涉及的 netdev_rx_stats 结构所收集的统计数据可通过文件 */proc/net/softnet_stat* 读取（其输出是十六进制格式）。

本部分涉及的函数和变量

表12-2所示是前三章介绍或涉及的主要函数、变量以及数据结构。其他的可以在第九章的表 9-1 中找到。

表 12-2：与中断事件及帧处理相关的函数、变量以及数据结构

名称	描述
函数	
netif_rx	把输入帧排入 CPU 队列。参见第十章“新旧驱动程序接口”一节
netif_rx_schedule __netif_rx_schedule	调度 NET_RX_SOFTIRQ 软件中断以准备执行。参见第十章“新旧驱动程序接口”一节
netif_rx_complete	当net_device->poll虚拟函数清除队列时就会调用此函数
netif_start_queue netif_stop_queue	分别开启和关闭设备的传输。参见第十一章的“传输的开启和关闭”一节
netif_queue_stopped	检查设备是否已开启传输功能

表 12-2：与中断事件及帧处理相关的函数、变量以及数据结构（续）

名称	描述
netif_schedule[a]	netif_schedule用以调度设备以准备传输。netif_wake_queue
netif_wake_queue	开启设备传输功能，并且设备调度以准备传输。参见第十一章的“为设备调度以准备传输”一节
qdisc_run	从设备出口队列中退出帧，将其压入设备驱动程序以进行传输。参见第十一章的“队列规则接口”一节
process_backlog	非NAPI设备驱动程序所用的poll虚拟函数。参见第十章的“积压的处理：process_backlog轮询虚拟函数”一节
netif_receive_skb	处理输入帧传给较高层协议处理函数。参见第十章的“输入帧的处理”一节
dev_queue_xmit	帧传输的主要函数。参见第十一章的“dev_queue_xmit函数”一节
dev_kfree_skb	释放sk_buff结构。参见第十一章的“处理
dev_kfree_skb_irq	NET_TX_SOFTIRQ：net_tx_action”一节
dev_kfree_skb_any	
do_IRQ	启用相关联的处理函数以负责硬件中断通知信息
open_softirq	为软件中断注册以及调度。参见第九章的“内核2.4版本以后的
raise_softirq、	下半部函数：引入软IRQ”一节
raise_softirq_irqoff	
do_softirq	启用相关联的处理函数以负责未决的软件中断。参见第九章的
invoke_softirq	“未决软IRQ的处理”一节
net_rx_action	分别是NET_RX_SOFTIRQ和NET_TX_SOFTIRQ的处理函数。参见
net_tx_action	第九章的“网络代码如何使用软IRQ”一节。
tasklet_init	初始化tasklet_struct结构。
tasklet_action	分别是TASKLET_SOFTIRQ和HI_SOFTIRQ软件中断的处理函
tasklet_hi_action	数。参见第九章的“微任务”一节。
tasklet_enable、	分别是开启和关闭微任务。参见第九章的“微任务”一节。
tasklet_hi_enable	
tasklet_disable、	
tasklet_disable_nosync	
tasklet_schedule	为微任务调度以准备执行。参见第九章的“微任务”一节。
tasklet_hi_schedule	
变量	
mod_cong	输入队列的拥塞等级（由非NAPI设备所用）。参见第九章的
lo_cong	“softnet_data的字段”一节

表 12-2：与中断事件及帧处理相关的函数、变量以及数据结构（续）

名称	描述
no_cong	
no_cong_thresh	
netdev_max_backlog	CPU 输入队列的最大尺寸。参见第九章的图 9-4
数据结构	
softnet_data	那两个NET_XXX_SOFTIRQ软件中断会为每个CPU都使用这么一个结构。参见第九章的“softnet_data 结构”一节
tasklet_struct	代表一个微任务。参见第九章的“微任务”一节。

a. 为了与接收函数名称一致，可能应该叫作netif_tx_schedule。

本部分涉及的文件和目录

图12-1所示是第三部分前四章所涉及的文件和目录。图中，*xxx*关键词代表一种体系结构（例如i386）（注 1）。有些体系结构不需要特殊的体系结构专用文件，因为一个通用文件有时也可以由好几种体系结构使用。

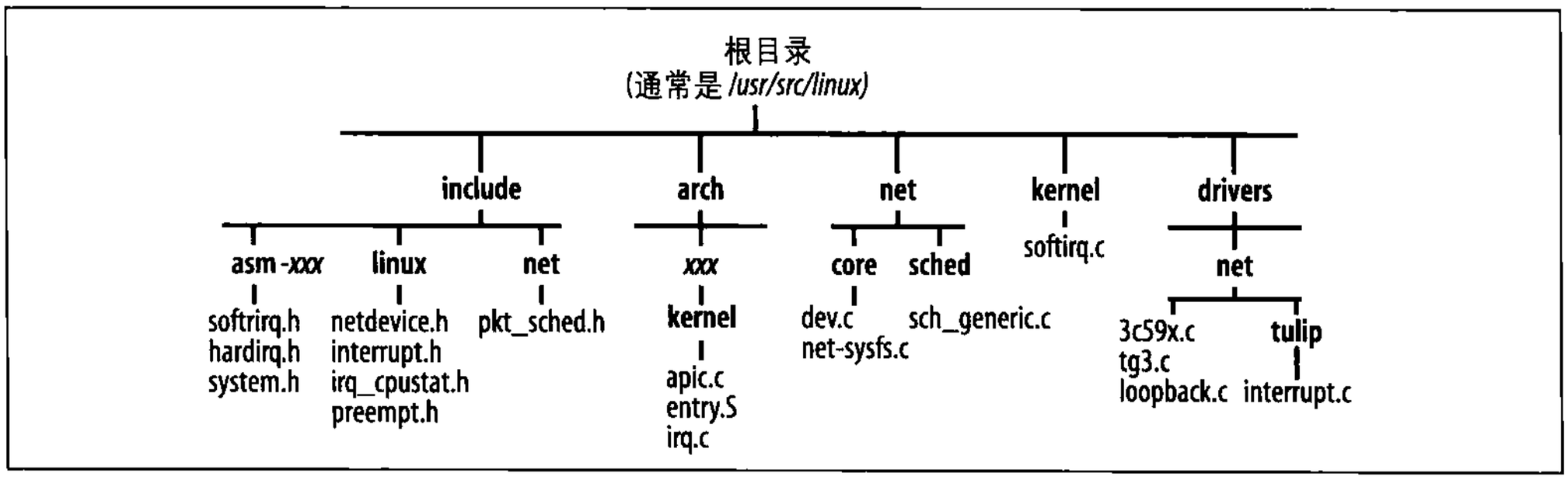
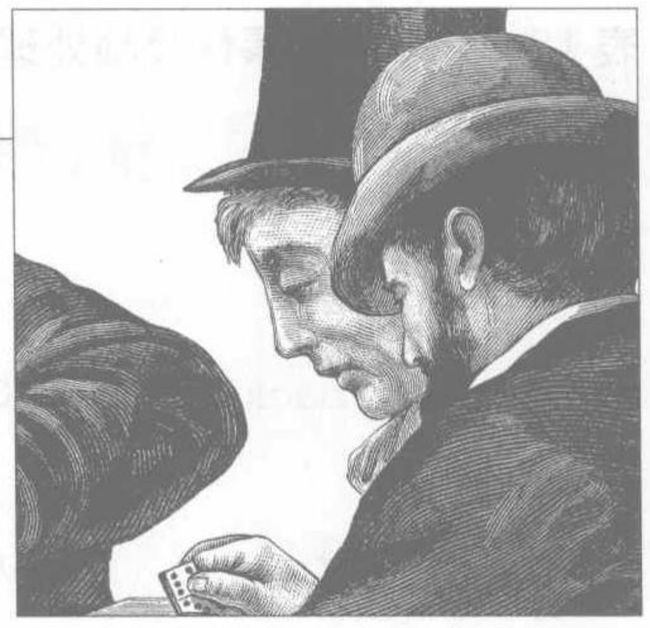


图 12-1：本部分涉及的文件和目录

注 1： *irq.c* 文件不一定在 *kernel* 目录下。

第十三章

协议处理函数



协议是所有通信的架构：指示每位通信者如何理解通话的另一端。在Linux中，对通信的理解是通过网络中各层的协议处理函数。本章说明这些处理函数如何安装、如何在执行期间被选中以及如何启用。

为了了解通信层和协议之间的关系，可以想象真实生活中的可能情况，例如与陌生人交谈。我应该使用什么语言？如果我人在意大利，就应该讲意大利语，但是如果我人在美国，就应该先讲英语。如果行不通，可以使用其他语言协商。

在基本协议之上，还有其他东西。例如，写信时，我与通信方的关系会决定开头是“亲爱的夫人”或“嗨，小姐”。这类选择在真实生活沟通的许多层面都会发生。网络也有分层，而协议的选择在网络代码中也已经定形了。

网络协议栈概论

本书读者应该熟悉基本的TCP/IP协议，然而还有一些协议也很常见，如LLC（Logical Link Control，逻辑链接控制）以及SNAP（Sub network Access Protocol，子网访问协议），但你可能还不懂。本节要介绍一些关键协议及其彼此间的关系。 网络协议最著名的两个模型就是七层的OSI模型以及五层的TCP/IP模型，如图13-1所示。即使因各种原因并未实际实践过，OSI模型一直是讨论网络链接时很重要的参考点。TCP/IP模型包含了当今计算机所用的多数协议（注1）。

注1：　有关这两种模型的其他信息，建议阅读《Computer Networks》第二版（由Prentice Hall出版）。

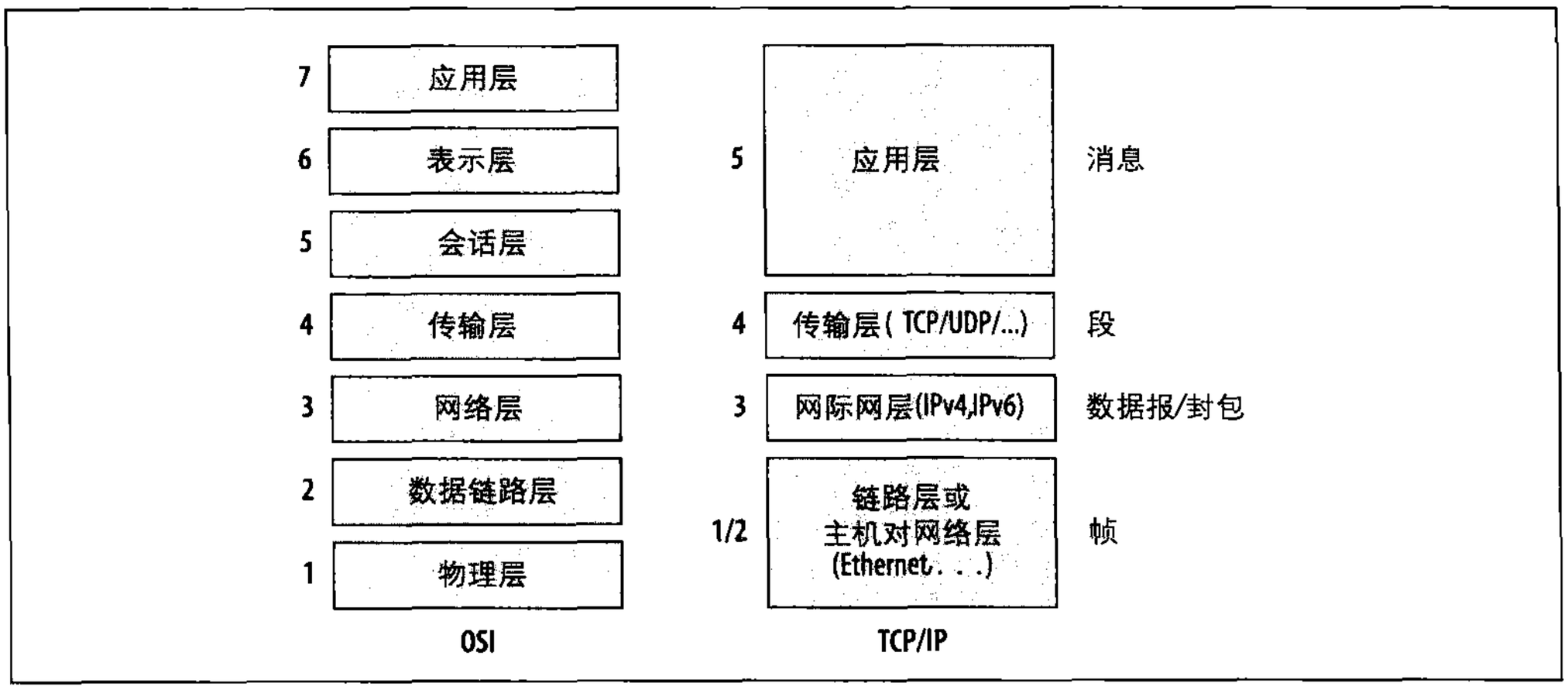


图 13-1：OSI 和 TCP/IP 模型

每一层中都有很多种协议可用。在最低层的接口交换数据，而所用的协议是预先决定的。协议的驱动程序被关联至该接口，而来到该接口的所有数据都假定会遵循此协议（也就是 Ethernet）。如果没有，就会回报错误，不会有任何通信发生。

但是，一旦驱动程序必须把数据交给较高层时，就发生了协议选择的问题了。L3 的数据应该由 IPv4、IPv6、IPX（Novell NetWare 协议）、DECnet 或其他网络层协议中的哪个协议处理呢？此外，从 L3 到 L4 时也需要做类似的选择；L4 有 TCP、DP、ICMP 以及其他协议可用。

本章涉及的是最低的三层，而第四层会简要提一下。

单独一个封包传输数据通常在链接层被称为帧（frame），在网络层则称为封包（packet），在传输层称为段（segment），而在应用层则称为消息（message）。

这些分层通常称为网络协议栈，因为通信会往下传播通过各个分层，直到实际上经过线路（或无线频道）传输，然后再返回来。报头也会以 LIFO 的方式添加和删除掉。

大蓝图

图 13-2 建立在图 13-1 所示的 TCP/IP 模型之上。图 13-2 显示出相邻分层间的每个接口会由哪一章说明。有些接口会涉及由协议栈上方向下的通信，而其他的则是由协议栈下方向上。

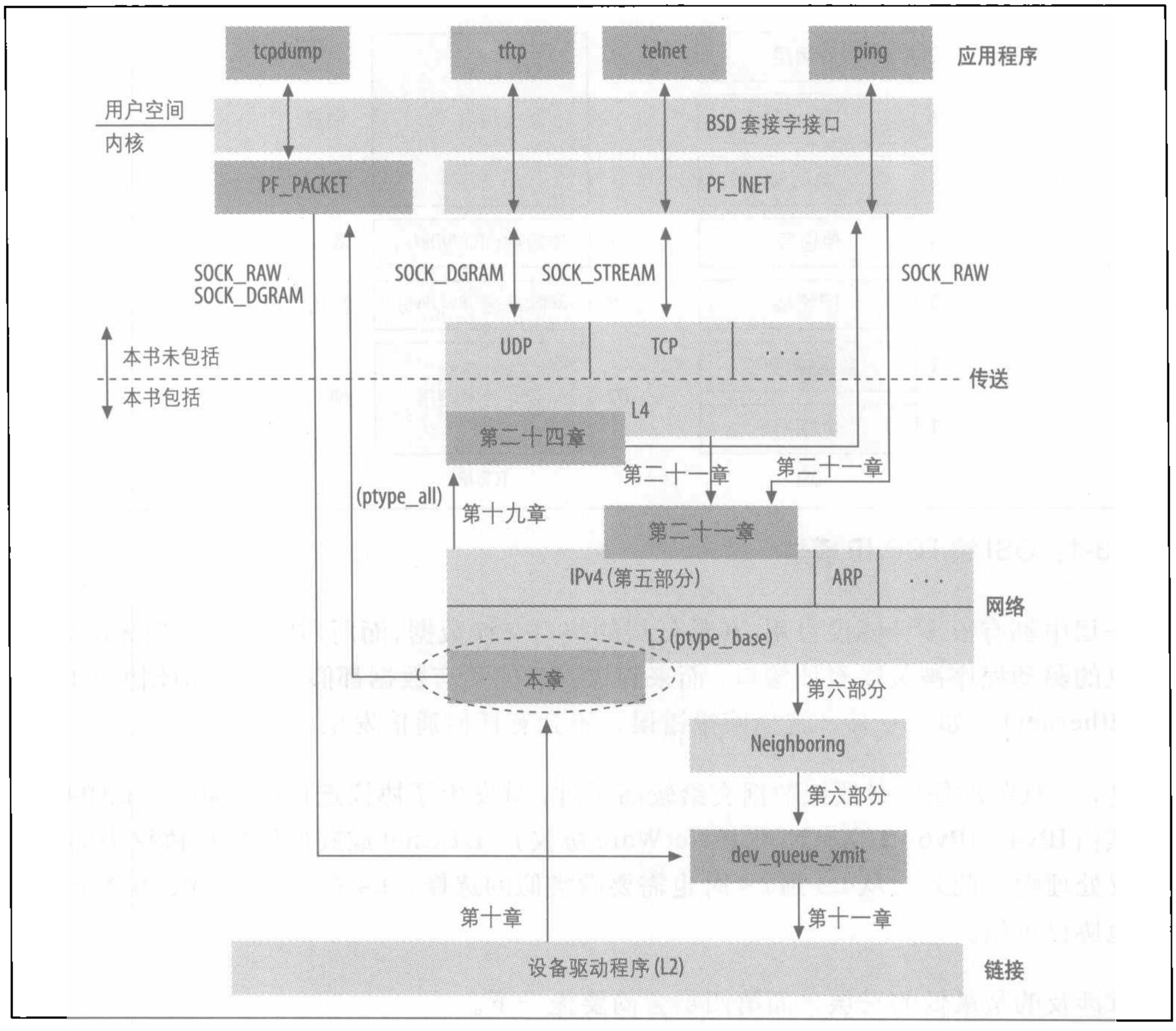


图 13-2：大蓝图

在协议栈中向上传递（用于接收信息）

本章说明入口流量如何交给正确的协议处理函数（在“协议处理函数的组织”一节中，ptype_base和ptype_all的意义就会清晰了）。

第十章描述设备驱动程序如何通知内核接收到入口帧之事。

第二十四章说明IPv4协议如何把入口IPv4封包传递给正确的L4协议（IPv4是本书中唯一讨论的网络层协议）。IPv4接收函数会在第十九章予以说明。

在协议栈中向下传递（用于传送信息）

第二十一章说明IPv4层所提供的传输函数。

本书第六部分会说明邻居层如何衔接L3协议和第十一章讨论过的传输函数dev_queue_xmit。

如图13-2所示，本书并不讨论套接字接口（socket interface）。然而，对于AF_PACKET套接字类型而言，有一点值得一提，即Linux在链路层捕捉帧并将其注入至链路层的方式：直接跳过所有中间的协议层。网络探嗅器（如*tcpdump*和*Ethereal*）是AF_SOCKET套接字的最普遍的用户。从图中可知，AF_PACKET套接字直接把帧交给dev_queue_xmit，并直接从网络协议分派函数接收入口帧（第十章已说明）。

图13-2只显示了两种协议系列（PF_INET、PF_PACKET），但是Linux内核还实现了其他多种协议。例如：

PF_NETLINK

网络配置较好的接口。参见第三章。

PF_KEY

网络安全服务的关键管理接口。IPsec是这些服务之一。

PF_LLC

参见“逻辑链接控制（LLC）”一节。

Ethernet的链路层的选择(LLC和SNAP)

虽然链路层协议会被所用的硬件绑定，但是，Ethernet标准可让我们在协议之间做某种选择。首次尝试将这种选择标准化的协议称为LLC（逻辑链接控制）。因为LLC提供的选项很有限，所以很少使用。随后，IEEE 802委员会又制定了SNAP（子网络访问协议）协议标准，现在已相当常见。这些子协议的实现会在本章稍后描述。

在LLC中，报头内含有一个字段，指出SSAP（Source Service Access Point，来源地服务访问点）的协议以及DSAP（Destination Service Access Point，目的地服务访问点）的协议。然而，每个字段只有8位，其中一个位指出是否使用了多播的标识，而另一个位指出该地址是一个网络的区域地址还是全球都识别的地址。因此，只剩6位可用于指明协议，因而LLC最多只能支持64种协议，使得此项技术难以普及。

鉴于以上原因，IEEE 802委员会在SSAP和DSAP字段中提供了一个特殊值，以此扩充LLC。此特殊值指出来源地或目的地所用的协议是由报头中另外5个字节标识。采用这项扩充功能（改称为SNAP），就有40位可以分派给各种协议了。

网络协议栈的操作方式

来粗略考察一个通信范例，以了解通信点之间如何做选择。

如图13-3中所示，假设有一个用户在主机X，想使用网页浏览器从服务器Y上的网站服务器下载一页HTML网页。要回答如下一些问题：

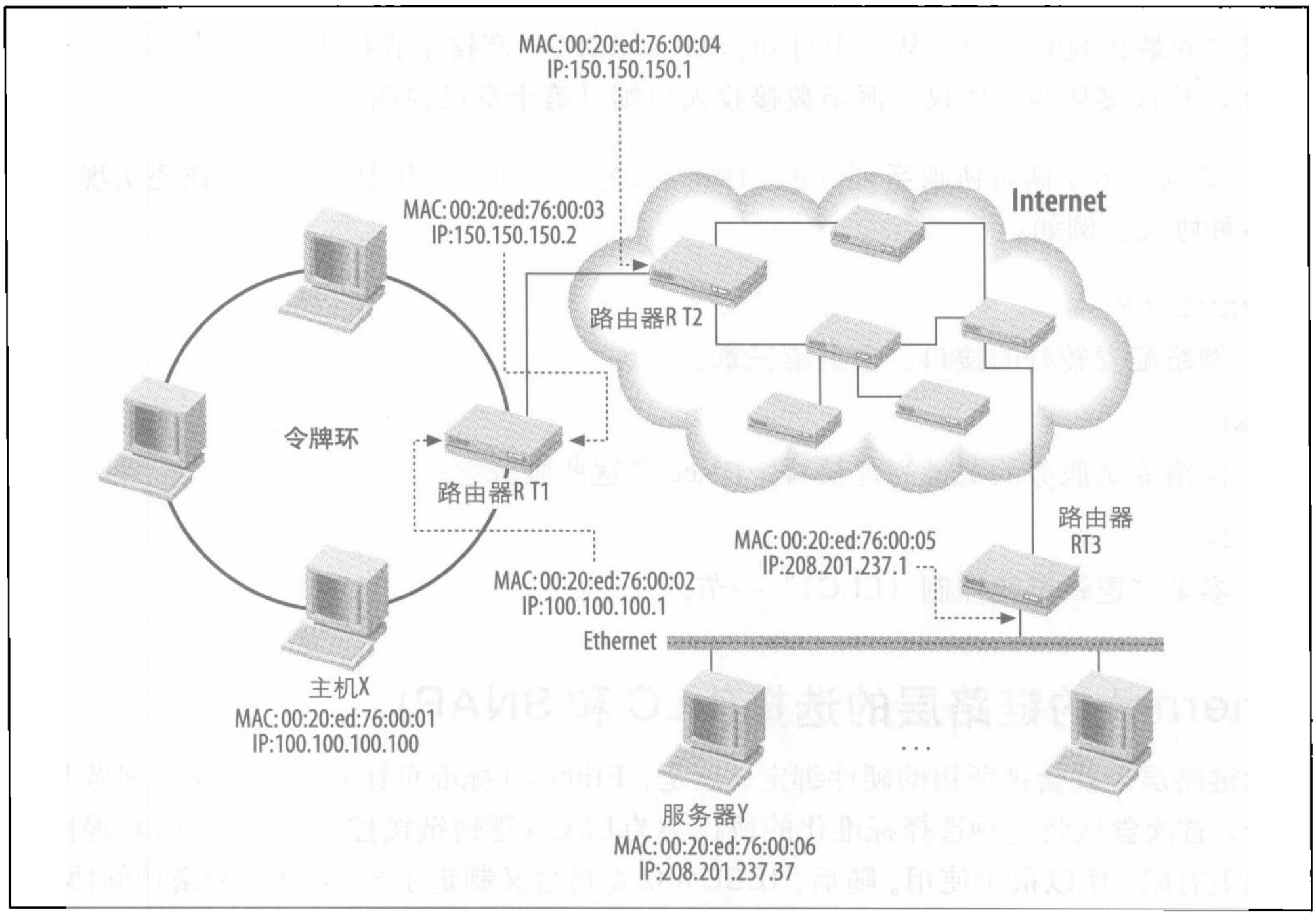


图13-3：两个远程站点间通信实例（主机X和服务器Y）

- 主机X和服务器Y位于不同的局域网络中，它们如何相互对话？
- 主机X不知道服务器Y实际位于何处，主机X如何知道该把请求传送至何处？
- 如果服务器Y上运行的应用程序不止一个（不只是网站服务器），其操作系统如何确定该由哪个应用程序处理来自于主机X的请求？
- 如果主机X上运行的应用程序不止一个（不只是浏览器），其操作系统如何确定该由哪个应用程序接收返回的数据？

让我们跟随网页的请求，通过网络协议栈看一看这些问题如何解答。我们要以图13-3（注2）和图13-4作为参考图。

注2：　图中只显示我们讨论所需的细节。

应用层，主机 X

浏览器读取用户请求的URL，假设是 *http://www.oreilly.com*。浏览器使用域名系统（Domain Name System，本书不作讨论）把网址 *www.oreilly.com* 解析成IP地址（假设是208.201.239.37）。至于怎么找出主机X和使用此地址的服务器Y之间的路径，就是IP协议（L3，网络层）的事了。

现在，浏览器在应用层对208.201.239.37启动了一个HTTP会话（session），然后，浏览器启用TCP把流量带往远程网站服务器（使用TCP而非UDP是因为HTTP需要可靠的信道来传递大量数据，而不会造成损毁）。现在，请求从网络协议栈上方向下传送。

传输层，主机 X

TCP层在必要时会把HTTP消息请求分成几个节段，然后为每个节段加上一个TCP报头。此外，TCP会加入来源地和目的地端口。端口号使操作系统可以把请求导向至正确的应用程序。服务器Y的网站服务器会监听默认的HTTP端口80，除非明确配置成使用另一个端口号，然后在那里获取所有流量。服务器Y会把响应导向至主机X的端口5000，也就是来源地端口号，而服务器就所接到的请求就是从主机的这个端口发出的。

端口号是一种L4的概念，所以，TCP和UDP各有一组端口可用。

主机X的TCP层知道目的地端口是80，因为浏览器使用分派给HTTP协议的默认端口（除非URL中提供另一个不同的端口）。分派给浏览器的来源地端口（处理入口流量时用于指认目的应用程序）是由OS分派的（除非应用程序要求特定的端口）。我们假设该端口为5000。会话两端可以使用不同端口。NAT(Network Address Translation，网络地址转换）以及代理防火墙（proxying firewall）使得这项议题更为复杂，但是，从这里的讨论应该可以看清楚应用程序彼此间如何联系的轮廓。

TCP层不知道该如何把那些节段送往目的系统。因此TCP层会启用IP层，在每个传输请求中都传递目的地IP地址。

网络层，主机 X

IP层不关心应用程序或端口。其所做的事就是检查封包的IP地址以及和IP相关的网络选项。其重责大任就是查询路由表（很复杂的流程，本书第七部分会讨论），以找出该封包应该经过路由器RT1。IPv4协议会在本书第五部分说明。

封包要往下传到另一层以便于被传送至该路由器，但是，IP层必须在此层找出路由器的正确地址。因为L2涉及到相邻主机间的通信（如主机共享LAN或者点对点链接），因此，IP层用于找出特定IP地址相关联的L2地址的流程就称为邻居协议（neighbor protocol）。对此本书第六部分会讨论。

链路层，主机 X 和路由器 RT1

这一层的一部分是由设备驱动程序实现的。在LAN上，Ethernet是最常见的协议，但是还有 ATM、Token Ring、FDDI以及其他协议存在。长距离的链接会使用专门的铜缆线或光纤缆线；最简单的链接就是几百万家庭和小办公室用户通过其ISP所建立的拨号连线。LAN使其自己的（L2）寻址方案与TCP/IP无关。对Ethernet（以及一般的IEEE 802网络）而言，地址是6个八位字节构成，而且通常称为MAC地址。在专线上（如拨号链接）并不需要L2寻址机制，因为每一边都是直接传给另一边。

不同的链接可能使用不同类型的报头，因为每一种都和硬件有关。这些报头所携带的信息对应用层的浏览器和服务器而言都没有意义。

路由器 RT1、RT2 等等

路径中的每个路由器（最后一个除外）都会经过下列流程把封包转发至其最终目的地：

- 删除链路层报头。
- 由于链路层报头中的特定字段（本章稍后讨论），可以看到 L3 协议就是 IP。
- 确定本地系统不是封包目的地，因为 IP 报头中的目的地 IP 地址不在其自己的 IP 地址之列。
- 把IP包转发至通往服务器Y路径上的下一个路由器。为此，路由器会查询其路由表，以选出下一个跳点路由器，然后建立新的链接层报头，如图 13-4（e）中所示。最后一步会在第三十五章详细描述。

正常来讲，封包在系统间传送时，L3（IP 报头）上的信息并不会改变（注 3）。而每个链接上都会使用不同的L2 报头。

当封包终于到达路由器 RT3 时，RT3 会发现服务器 Y 和其直接连接，因此没必要再把封包绕送至另一个跳点。

一旦消息到达目的地服务器时，就会再由下而上穿越网络协议栈：

连路层，服务器 Y

剥离 L2 报头时，此层会检查一个字段，以了解该由哪个协议处理 L3 层。得知L3是由IP处理后，链路层会启用适当的函数以继续处理L3包（也就是L2有效载荷）。本章多数内容都是讨论协议注册的方式，以及处理指出该用哪个协议的关键字段。

注 3： 第五部分就会知道，只有一些会变动的字段需要更新，如 TTL（Time to Live，存活时间）以及校验和。

网络层，服务器 Y

此层识别出其自己系统的IP地址（208.201.239.37）就是封包中的目的地地址，因此，此封包应该在本地接收处理。网络层会剥离L3报头，再一次检查一个字段，以了解由哪一个协议处理L4。第二十四章会深入讨论入口流量的L3和L4之间的接口。

图13-4显示出每一分层从较高层取得数据后增加报头的过程。最后一步（从图13-4（d）到图13-4（e））显示出由主机X传送至路由器RT1的原有帧，以及路由器RT1和路由器RT2之间的帧的差异性。

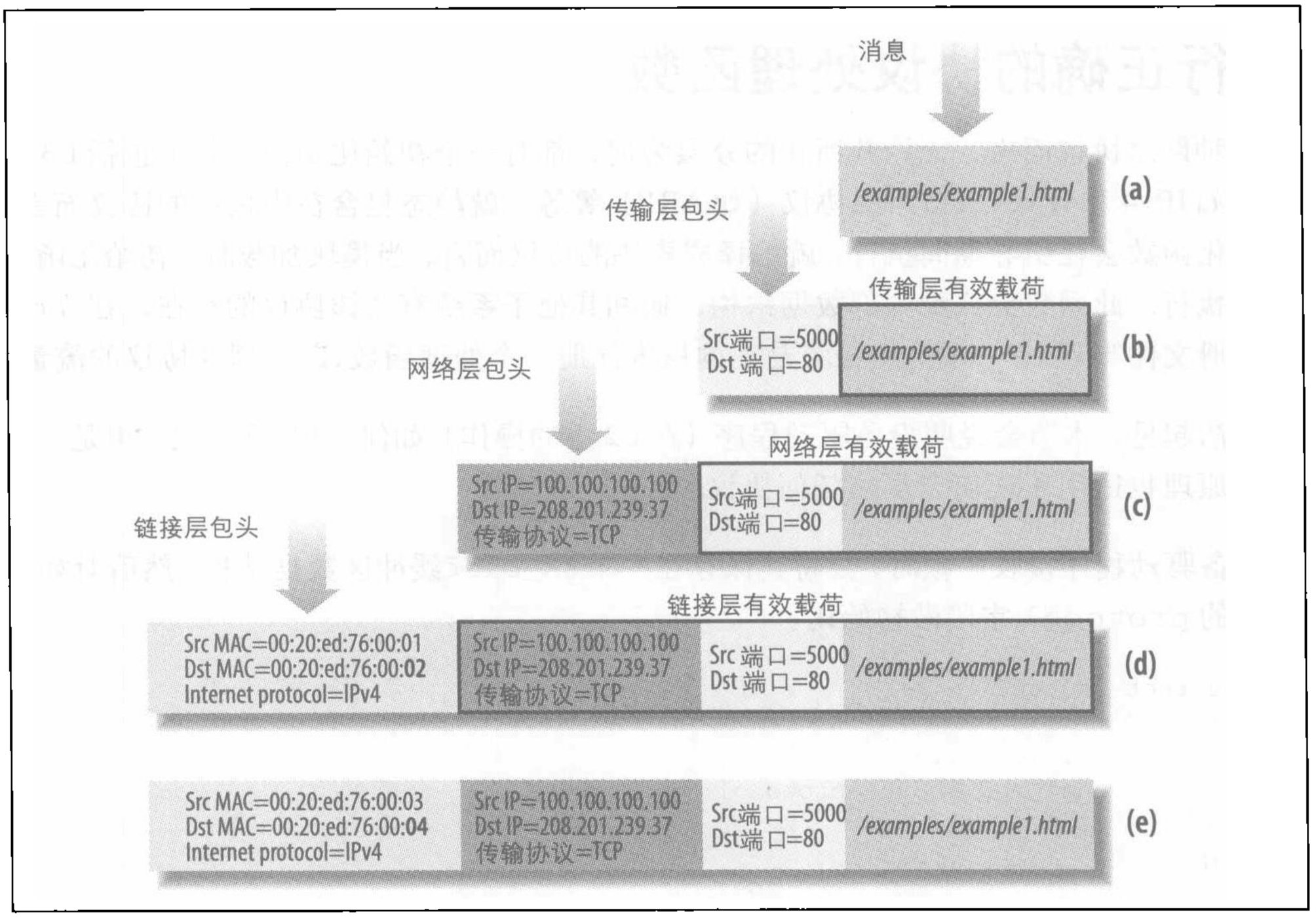


图13-4：各个分层所编辑的报头；（a-d）在主机X上向协议栈下方传送；（e）在路由器RT1上

我们已经知道，每个分层都提供各式各样的协议，而每一种协议都由一组不同的内核函数处理。因此，当封包回头向协议栈上方传送时，每一种协议都必须搞清楚下一较高层所用的协议是哪一种，然后启用适当的内核函数以处理该封包。

在最低的软件层L2上，所用的硬件决定所用的协议。如果帧是在Ethernet接口上被接收，接收者知道其包含了一个Ethernet报头，而Token Ring接口则知道其包含了一个

Token Ring报头等等。这里不能模棱两可，除非指定LLC或SNAP。LLC和SNAP会在本章稍后讨论。

当封包在网络协议栈中向上传送时，每一个协议在其报头中都需要一个字段，指出下一阶段的处理该使用哪个协议。此进程就如同图13-5所示。因此，从图13-5（a）中的L2转至图13-5（b）中的L3时，就依赖于L2检查L2报头中的“上层协议”字段。同样的，L3层也会检查其报头内的一个字段，以促成转至L4，如图13-5（b）和图13-5（c）所示。最后，L4使用封包的“目的地端口”字段把封包从内核中取出，然后找出在本地主机处理该封包的进程（如网站服务器）。

执行正确的协议处理函数

就每种网络协议而言，无论其所在的分层为何，都有一个初始化函数。其中包括L3协议（如IPv4和IPv6）、链路层协议（如ARP）等等。就静态包含在内核中的协议而言，初始化函数会在引导期间执行；就编译成模块的协议而言，当模块加载时，初始化函数就会执行。此函数会分配内部数据结构，通知其他子系统有关该协议的存在，在*/proc*中注册文件等等。有一项关键任务是在内核内注册一个处理函数，以处理该协议的流量。

为简洁起见，本节会说明设备驱动程序（在L2上的操作）如何启用L3协议，但是，同样的原理也适用于任何分层的任何协议。

当设备驱动程序接收一帧时，会将其保存在一个sk_buff缓冲区数据结构，然后对如下所示的protocol字段做初始化：

```
struct sk_buff
{
        ... ... ...
        unsigned short  protocol;
        ... ... ...
};
```

此字段的值可以是内核所用的任何值，以识别特定的协议，或者进来的帧中一个MAC报头中的一个字段。此字段会由netif_receive_skb内核函数（第十章已描述）查询，以确定该由哪个函数来执行，以处理L3的封包，参见图13-6所示。

由内核用在protocol字段中涉及协议的值大多都列在了*include/linux/if_ether.h*中，名为ETH_P_*XXX*。尽管有ETH前缀，但并非所有名称都涉及Ethernet硬件。如表13-1所示，其中包含了各种各样的活动。表13-1列出内核内部所用的值，而这些值由设备驱动程序直接指定给skb->protocol，而不是从帧报头中剥离出来（表中省略的值并没有分派函数处理函数）。例如，表中第一列指出，内核处理函数ipx_rcv用于处理一个输入封包，而且其skb->protocol字段为ETH_P_802_3。

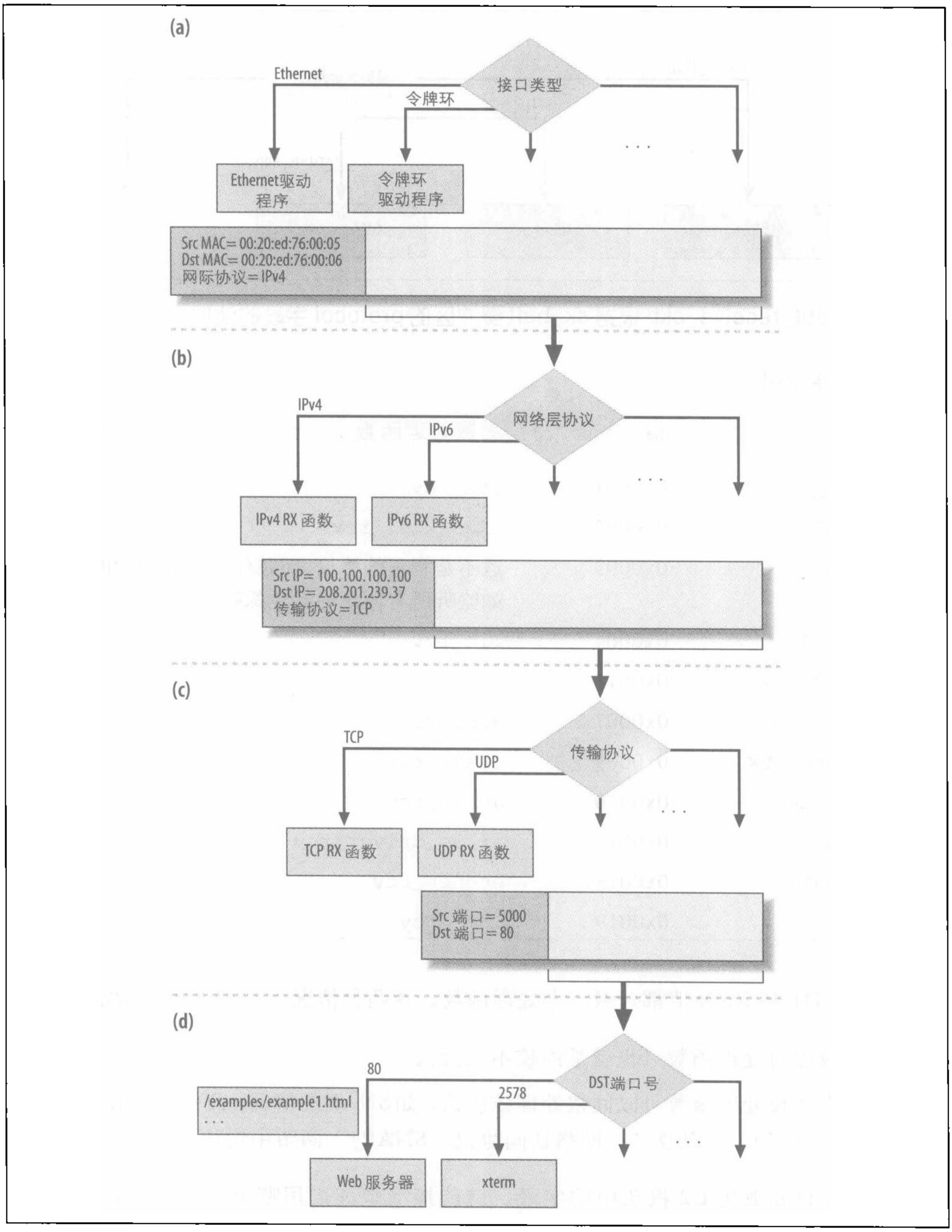


图 13-5：帧在服务器 Y 上逐层拆封

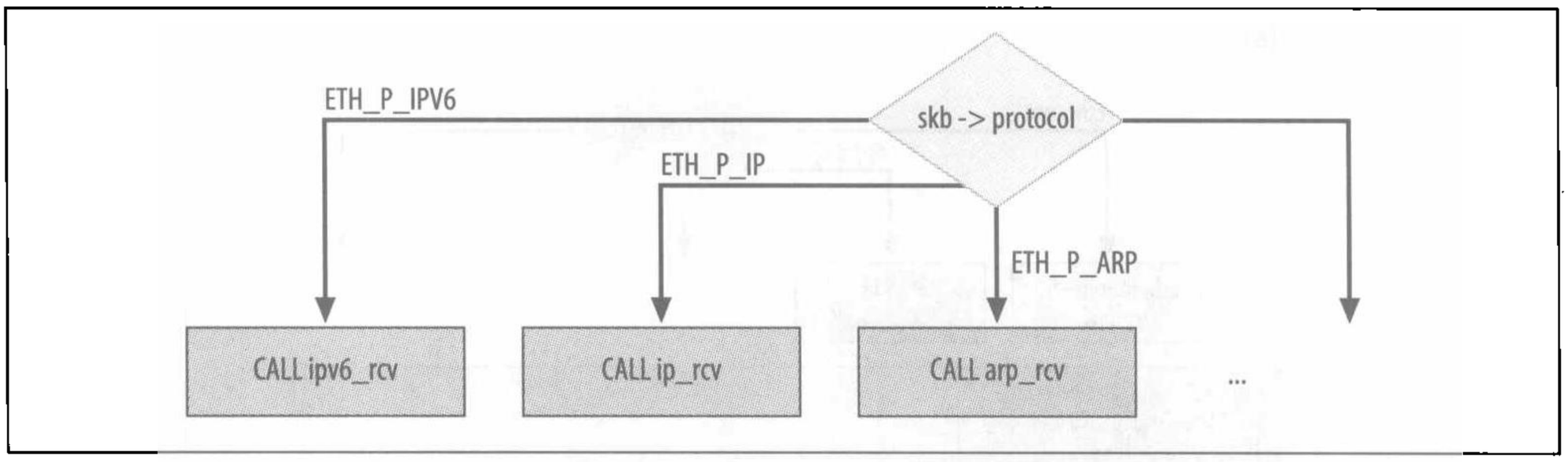


图 13-6：netif_receive_skb 根据 sk_buff 缓冲区的 protocol 字段的处理

表 13-1：内部协议

符号	值	函数处理函数
ETH_P_802_3	0x0001	ipx_rcv
ETH_P_AX25	0x0002	ax25_kiss_rcv
ETH_P_ALL	0x0003	这不是真正的协议，而是作为处理函数的通配符，如监听所有协议的封包探嗅器
ETH_P_802_2	0x0004	llc_rcv
ETH_P_TR_802_2	0x0011	
ETH_P_WAN_PPP	0x0007	sppp_rcv
ETH_P_LOCALTALK	0x0009	ltalk_rcv
ETH_P_PPPTALK	0x0010	atalk_rcv
ETH_P_IRDA	0x0017	irlap_driver_rcv
ETH_P_ECONET	0x0018	econet_rcv
ETH_P_HDLC	0x0019	hdlc_rcv

并非所有 ETH_P_XXX 值都分派一个处理函数。在两种情况下，可以不予指定：

- 该协议没有处理函数（也就是内核不支持）。
- 另一个协议处理函数可以间接处理该协议，如SNAP的情况。此情况会在“逻辑链接控制（LLC）”以及“子网络访问协议（SNAP）”两节中讨论。

可惜，并非只要取出 L2 报头中的字段，就能搞清楚该启用哪个处理函数；skb->protocol和处理帧的协议处理函数间的关联关系不一定都是一对一的。有些情况下，特定 ETH_P_XXX 的协议处理函数实际上只会从帧报头中读取其他参数（没有处理帧），然后把帧交给另一个会处理该帧的协议处理函数。其中一例就是 ETH_P_802_2 处理函数。

如第十章所述，netif_receive_skb是把入口帧分派给正确的协议处理函数的函数。当特定协议没有处理函数可用时，该帧就会被丢弃。

在特殊情况下，单一封包可以传递给多个处理函数。例如，当封包探嗅器在运行时就是这种情况。这种操作模式有时称为混杂模式，也就是表 13-1 中的 ETH_P_ALL。这类处理函数通常不用于为接收者处理封包，而只是窥探特定设备或一组设备，其目的是纠错或收集统计数据。

特殊媒介封装

到目前为止，Ethernet是实现共享和点对点网络联机最常见的机制。本书中讨论L2时，始终是指Ethernet设备驱动程序。然而，Linux允许你使用现代PC上由任何最常见的媒介（media）携带IP流量（以及有时任何网络协议流量）。可用于传输IP的媒介范例包括串行和并行端口(SLIP/PLIP/PPP)、FireWire(eth1394)、USB、蓝牙(Bluetooth)以及IrDA等等。

这类媒介会把网络设备定义成通用端口的抽象层，通常是通过对通用媒介设备驱动程序做扩充。这类虚拟设备对那些上层而言，看起来就像是真实的NIC。

以下是接收和传输在这些虚拟网络设备上的实现方式：

传输

net_device的hard_start_xmit虚拟设备函数指针，由设备驱动程序初始化为一个能根据媒介所用的协议而封装IP封包（假设原本就是IP封包）的函数。

接收

当通用驱动程序从其端口之一接收数据时，会把媒介报头剥离（如同Ethernet设备会剥离Ethernet报头一样），对skb->protocol初始化，然后调用netif_rx通知其上层。当这些媒介只用于点对点联机时，就没有必要使用链接层报头，所以skb->protocol会静态初始化成ETH_P_IP；在其他情况下，媒介封装可能也会包括一个假设的Ethernet报头，所以skb->protocol会由eth_type_trans函数做初始化（如同真实的Ethernet驱动程序所做的那样）。

如何把给定媒介类型的通用设备驱动程序与虚拟网络设备接口，是实现细节的问题。具体依赖于媒介而定，也许会提供同步或异步接口，或在接收和传输路径上使用缓冲机制等。

协议处理函数的组织

图13-7所示是各种协议处理函数在内核内的组织方式。每种协议都由packet_type数据结构描述。

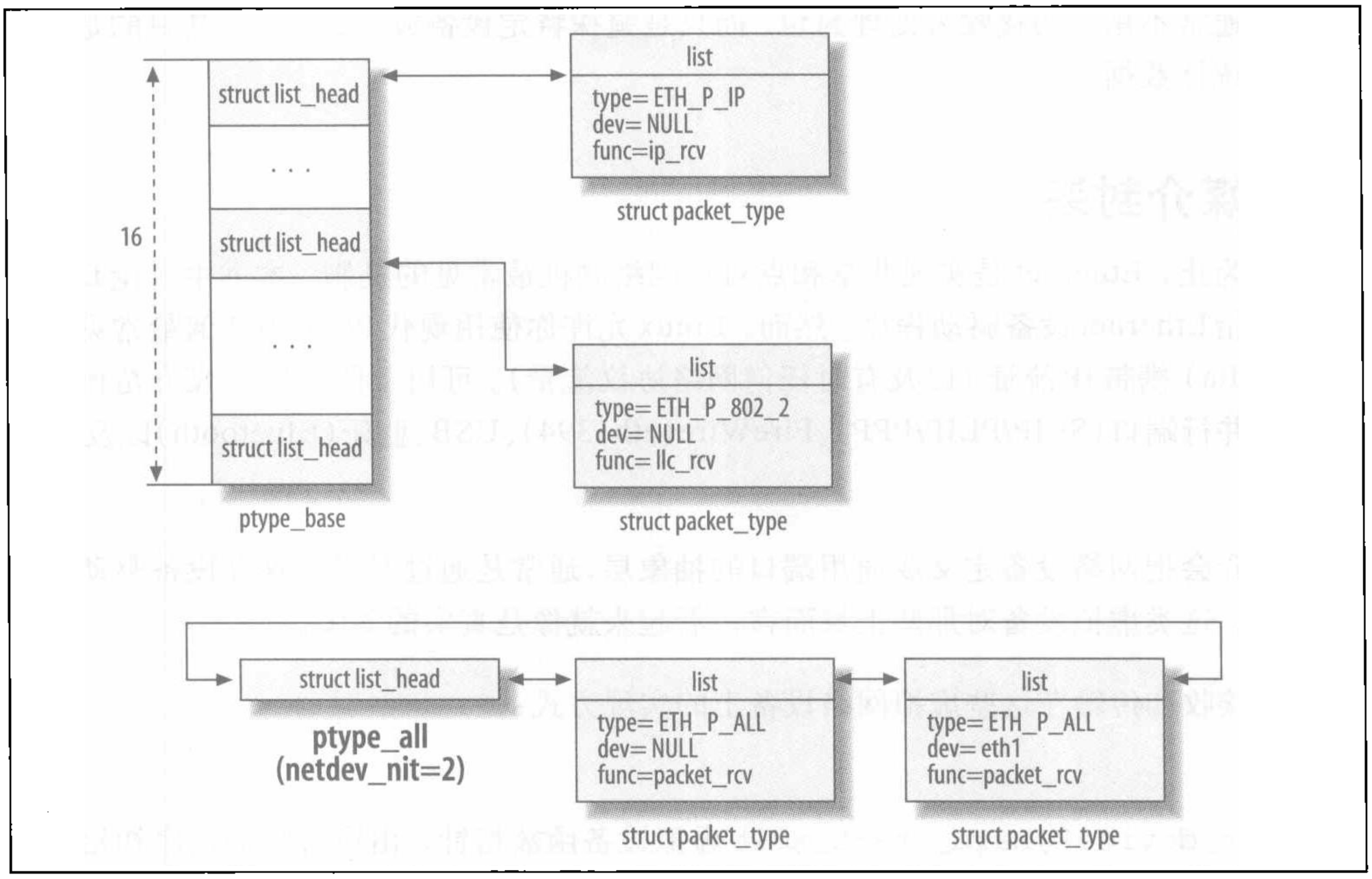


图13-7：用于存储已注册协议处理函数的数据结构

为了访问快一点，多数协议都用到一个很简单的hash函数。16个列表组织成一个数组，也就是全局变量ptype_base所指的数组。当一个协议使用dev_add_pack函数注册时（下一节会说明），此函数就会对协议类型执行hash函数，然后把packet_type结构分派至16个列表之一。稍后，要找出一个packet_type结构时，内核只要重新运行hash函数，并且遍历匹配列表。

那些ETH_P_ALL协议（参见表13-1）被组织在它们所属的列表中，也就是全局变量ptype_all所指的列表（注4）。此列表中的协议数目存储在netdev_nit。而dev_queue_xmit和qdisc_restart会使用netdev_nit检查某个PF_PACKET套接字是否开启（也就是监听中的探嗅器），使其得以传递入口帧的副本（参见第十章）。

注4：图中显示ETH_P_ALL协议类型使用packet_rcv函数。func是由在*net/packet/af_packet.c*中的packet_create根据内核配置初始化的。

协议处理函数的注册

无论是系统启动或其他时刻，当一个协议注册时，内核就会调用dev_add_pack，把一个定义在 *include/linux/netdevice.h* 类型为packet_type的数据结构传进去：

```
struct packet_type
{
    unsigned short       type;
     struct net_device *dev;
     int              (*func) (struct sk_buff *, struct net_device *,
                       struct packet_type *);
     void              *af_packet_priv;
     struct list_head    *list;
};
```

这些字段的意义如下：

type

协议代码。此值可以是表13-1和表13-4第一栏（如ETH_P_IP）所列任何值之一。不同表内的协议间的差异性会在后续几节逐渐阐述。

dev

设备的指针(如*eth0*)，该协议为此设备而开启。置为NULL时，指的是“所有设备”。由于此参数有可能使不同的设备有不同的处理函数，或者把特定处理函数关联给特定设备。通常不会这样做，不过有助于测试。PF_PACKET套接字时常以此监听特定设备。例如，像 *tcpdump -i eth0* 这样的命令会通过一个 PF_PACKET 套接字建立一个packet_type实例，并将dev初始化为与*eth0*相关联的net_device实例。

func

当netif_receive_skb（参见第十章）必须处理一个skb->protocol=type（如ip_rcv）的帧时，就会调用此函数处理函数。注意，此func的输入参数之一是一个指向packet_type结构的指针：此结构由 PF_PACKET 套接字使用，以访问af_packet_priv字段。

af_packet_priv

由PF_PACKET套接字使用。这是一个指针，指向与packet_type结构的建立者相关联的sock数据结构。此字段允许dev_queue_xmit_nit（第十章讨论过）不把缓冲区传递给传送者，而且通过PF_PACKET接收函数把输入数据传递给正确的套接字。

list

用于把这个数据结构和其他在同一个bucket的列表中相冲突的实例链接起来。参见图 13-7 所示。

当你有多个packet_type实例都关联于有相同的type的协议，则匹配type的输入帧就会通过为这些实例启用 func，转交给所有这些协议处理函数实例。

为每个协议注册时，内核会对packet_type结构做初始化，然后调用dev_add_pack。这里有个取自 *net/ipv4/ip_output.c* 的范例，显示出IPv4关键代码如何注册IPv4协议处理函数。

当IPv4协议在引导期间初始化时，ip_init 函数会被执行。其中一种结果是，IPv4 packet_type结构中的函数ip_rcv会注册成此协议的函数处理函数。所有Ethernet帧接收时，若 ETH_P_IP的值为Protocol Above，就会由函数 ip_rcv处理。

```
static struct packet_type ip_packet_type =
{
    .type = _ _constant_htons(ETH_P_IP),
    .func = ip_rcv,
}
...
void _ _init ip_init(void)
{
    dev_add_pack(&ip_packet_type);
    ...
}
```

dev_add_pack 相当简单：检查要新增的处理函数是否为协议探嗅器（pt->type==htons(ETH_P_ALL)）。如果是，则此函数会将其添加到ptype_all所指的列表，然后递增已注册的协议探嗅器的数目（netdev_nit++）。如果处理函数不是探嗅器，就会被插入至ptype_base所指的16个列表之一（依赖于hash编码的值）。ptype_base和ptype_all 所指的数据结构都会由 ptype_lock 回转锁保护。

```
void dev_add_pack(struct packet_type *pt)
{
    int hash;

    spin_lock_bh(&ptype_lock);
    if (pt->type == htons(ETH_P_ALL)) {
        netdev_nit++;
        list_add_rcu(&pt->list, &ptype_all);
    } else {
        hash = ntohs(pt->type) & 15;
     list_add_rcu(&pt->list, &ptype_base[hash]);
    }
    spin_unlock(&ptype_lock);
}
```

函数 dev_remove_pack，如其名称所暗示与 dev_add_pack 互补。

```
void dev_remove_pack(struct packet_type *pt)
{
```

```
        _ _dev_remove_pack(pt);

    synchronize_net();
}
```

dev_remove_pack会从ptype_all或ptype_base中把packet_type结构删除掉，而synchronize_net是用于确保dev_remove_pack返回时没有谁还持有已删除的packet_type实例的引用（例如，参见第十章netif_receive_skb中所用的RCU上锁机制）。

如果dev_add_pack在负责模块初始化的函数init_module之内调用，则dev_remove_pack最可能在cleanup_module（当模块要被删除时，由内核调用）之内被调用（可以在*net/ax25/af_ax25.c*中找到范例）。另一方面，如果该协议静态包含在内核内，就会在引导期间自动被注册，而且只有当系统关机时才会被删除。IPv4协议是绝不会在运行期间被删除的协议。

Ethernet与IEEE 802.3帧

有很多协议都归属于广义的术语*Ethernet*之下。802.2和802.3标准分别由协议ETH_P_802_2和ETH_P_802_3表示，但是，还有很多其他Ethernet协议（如表13-2中所列），以及LLC和SNAP扩充协议。这些标准构成了一些足以支持所有变化版本协议的手段（关于h_proto会在下一节讨论）。

表13-2：有效的Ethernet类型（当h_proto > 1536）

协议	Ethernet类型	函数处理
ETH_P_IP	0x0800	ip_rcv
		ic_bootp_recv[a]
ETH_P_X25	0x0805	X25_lap_receive_frame
ETH_P_ARP	0x0806	arp_rcv
ETH_P_BPQ	0x08FF	bpq_rcv
ETH_P_DNA_RT	0x6003	dn_route_rcv
ETH_P_RARP	0x8035	ic_rarp_recv
ETH_P_8021Q	0x8100	vlan_skb_rcv
ETH_P_IPX	0x8137	ipx_rcv
ETH_P_IPV6	0x86DD	ipv6_rcv

表 13-2：有效的 Ethernet 类型（当 h_proto > 1536）（续）

协议	Ethernet 类型	函数处理
ETH_P_PPP_DISC	0x8863	Pppoe_disc_rcv
ETH_P_PPP_SES	0x8864	Pppoe_rcv

a. IP 之所以有两个处理函数，是考虑到内核有可能以 RARP/BOOTP 这类协议获取 IP 配置。ic_bootp_recv 函数只用于引导期间负责动态 IP 配置，而一旦配置获取后就会被卸除。参见 *net/ipv4/ipconfig.c*。

Ethernet 是在 IEEE 建立其 802.2 和 802.3 标准前设计的。802.2 和 802.3 标准并非纯粹的 Ethernet，只不过通常都称为 Ethernet 标准。所幸，IEEE 802 委员会决定让协议可兼容。每块 Ethernet 卡都可接收 802 标准帧类型以及旧式 Ethernet 帧，而内核可以提供一个函数（本节稍后讨论）使设备驱动程序可以识别这些帧（由于本章所说的解决办法）。

以下是 Ethernet 报头的定义：

```
struct ethhdr
{
        unsigned char   h_dest[ETH_ALEN];       /* 目的地 eth 地址    */
        unsigned char   h_source[ETH_ALEN];     /* 来源地 eth 地址    */
        unsigned short  h_proto;                /* 封包类型 ID 字段   */
} _ _ATTRIBUTE_ _ ((packed));
```

后面两节讨论 LLC 和 SNAP 时就会知道，其他字段也会关注 ethhdr 结构。这里我们关注的焦点是协议字段 h_proto。尽管其名称如此，但实际上可以存储正在使用的协议或者帧的长度。这是因为此字段的大小是 2 个八位字节（2 个字节），但是，一个 Ethernet 帧的最大尺寸为 1,500 个字节（实际上，如果包含 SA、DA、校验和字段以及开头内容的话，尺寸可以多达 1,518 个字节。使用 802.1q 的帧有四个额外的封装字节，因此可以有的尺寸是 1,522 个字节）。

为了节省空间，IEEE 决定使用大于 1,536 的值以表示 Ethernet 协议。有些早已存在而加上标识符后依然低于 1,536 的协议（0x600 十六进制值），也已更新为满足这项准则。然而，802.2 和 802.3 协议使用这个字段存储帧的长度（注 5）。界于 1,501 和 1,535 之间的值，在此字段中是非法的。

注 5：这种安排的原因说来话长。好奇的话，我建议你阅读《Interconnections, Second Edition: Bridges, Routers, Switches》以及《Internetworking Protocols》（Addison Wesley 出版），作者进行解释时充满了讽刺意味。

图 13-8 所示是 Ethernet 报头可能的几种变种。图 13-8（a）所示是简单的 Ethernet。图 13-8（b）所示是 802.2 和 802.3 变种。如你所见，对前者而言，那个字段是作为协议字段，而就后者而言，该字段又成了长度字段。此外，802 变种也可支持 LLC［如图 13-8（c）所示］和 SNAP［如图 13-8（d）所示］。

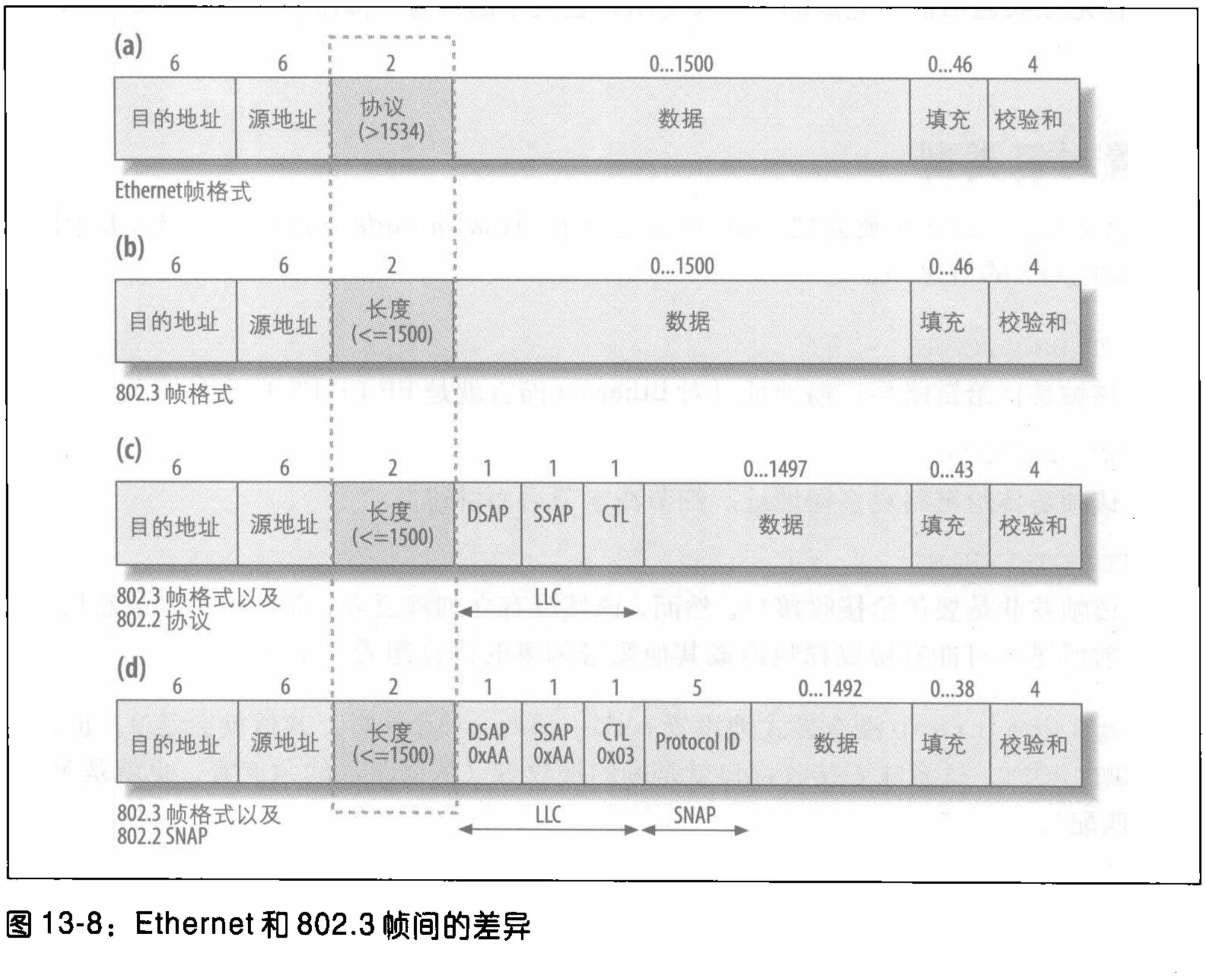


图 13-8：Ethernet 和 802.3 帧间的差异

Linux 在 eth_type_trans 函数中完成协议和长度这种特殊的区分。下列代码片段代表的是典型的环境，当 *drivers/net/3c509.c* Ethernet 驱动程序接收一个帧时就会用到。netif_rx 函数负责把帧拷贝到输入队列中，然后设置 NET_RX_SOFTIRQ 标识，让内核知道队列中有新帧［在第十章已描述了这一点（注 6）］。调用 netif_rx 之前，调用者要先调用 eth_type_trans 做一些重要的初始化工作。

```
el3_rx(struct device *dev)
{
        ... ... ...
```

注 6：　netif_rx 仅是设备驱动程序可用的两种接口之一，以此通知内核接收到帧之事。这两种接口都已在第十章做过描述。

```
            skb->protocol = eth_type_trans(skb,dev);
            netif_rx(skb);
            ... ... ...
    }
```

eth_type_trans执行两项主要任务：设置封包类型（注7）以及设置协议。设置协议的工作是在其返回值中完成的。先来说明封包类型的设置，再回到本节的主要议题：协议。

设置封包类型

eth_type_trans函数会把skb->pkt_type设成*include/linux/if_packet.h*中所列PACKET_XXX的值之一。

PACKET_BROADCAST

该帧是传给链路层广播地址（对Ethernet而言就是FF:FF:FF:FF:FF:FF）。

PACKET_MULTICAST

该帧是传给链路层多播地址。细节在本节后面讲述。

PACKET_OTHERHOST

该帧并非是要传给接收接口。然而，该帧没有立刻被丢弃，而是传给次最高层。如前所述，可能有协议探嗅器或其他爱管闲事的协议想看一下这个帧。

当eth_type_trans没有显式地设置skb->pkt_type时，其值就会是0，也就是PACKET_HOST。这意味着接收接口就是帧的接收者（从链路层的角度看，也就是MAC地址匹配）。

设置正确封包类型所需的信息，大多数都会在报头中明确指定。Ethernet地址是48位或6字节长。第一个字节（网络字节尾端）的最高两个位有特殊意义（参见图13-9）：

- 位0区分多播地址和单播地址。广播地址是多播的特殊情况。设为1时，此位指的是多播；设为0时，指的是单播。通过if(*eth->h_dest&1)检查此位后，此函数会继续通过memcmp(eth->h_dest,dev->broadcast, ETH_ALEN)比较此地址和设备的广播地址，以了解该帧是否为广播帧。
- 位1区分局部地址和全局地址。全局地址是全球唯一的，而局部地址就不是了：系统管理员可自行决定要如何适当分派局部地址（注8）。设为1时，此位指全局地址；为0时，指的是局部地址。

注7：虽然程序称其为封包类型，但实际上是帧类型，因为是从链接层地址派生而来的。

注8：本地MAC地址和不可路由IP地址（192.168.*x.x*等）之间并没有关系：它们在概念上很类似，但是用在协议栈中的不同分层。

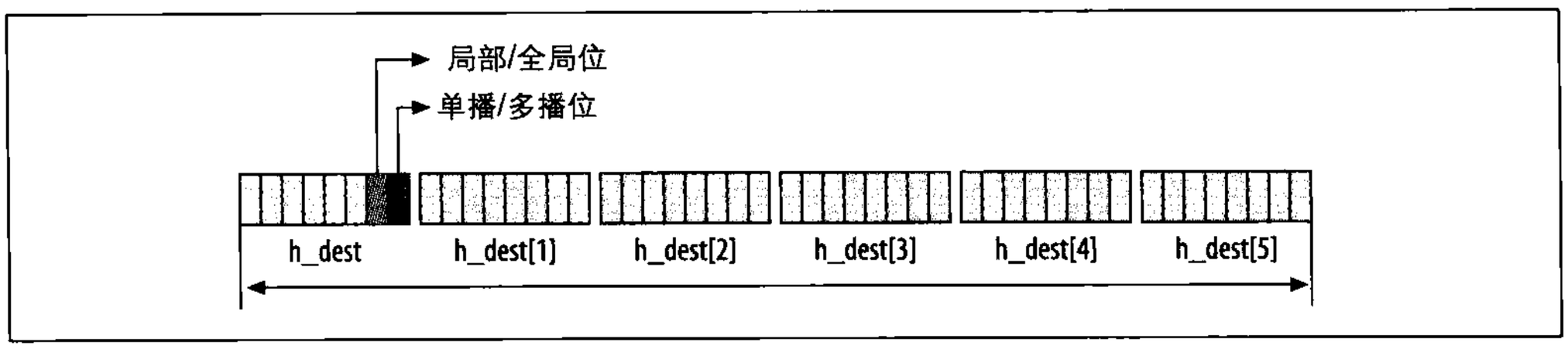


图 13-9：MAC 地址中的单播 / 多播以及局部 / 全局位

因此，eth_type_trans 的开头部分如下：

```
unsigned short eth_type_trans(struct sk_buff *skb, struct net_device *dev)
{
    struct ethhdr *eth;
    unsigned char *rawp;

    skb->mac.raw=skb->data;
    skb_pull(skb,ETH_HLEN);
    eth= eth_hdr(skb);
    skb->input_dev = dev;

    if(*eth->h_dest&1)
    {
        if(memcmp(eth->h_dest,dev->broadcast, ETH_ALEN)==0)
            skb->pkt_type=PACKET_BROADCAST;
        else
            skb->pkt_type=PACKET_MULTICAST;
    }

    else if(1 /*dev->flags&IFF_PROMISC*/)
    {
        if(memcmp(eth->h_dest,dev->dev_addr, ETH_ALEN))
            skb->pkt_type=PACKET_OTHERHOST;
    }
```

当接口进入混杂模式时，dev->flags 中的 IFF_PROMISC 标识就会设置。如前述代码片段所示，当目的地 MAC 地址与接收接口的地址不匹配时（无论 IFF_PROMISC 是否设置），eth_type_trans 都会把 skb->pkt_type 设为 PACKET_OTHERHOST。这样将允许 PF_SOCKETS 处理函数接收帧副本（参见第十章 netif_receive_skb），但是上层协议处理函数必须丢弃 PACKET_OTHERHOST 类型的缓冲区（例如，参见 arp_rcv 和 ip_rcv）。

设置 Ethernet 协议及长度

eth_type_trans 的第二部分是获取在较高层所用协议的标识符。协议值也称为 Ethertype，而有效类型的列表会随时保持最新，位于 *http://standards.ieee.org/regauth/ethertype*。高于 1,536 的旧 Ethernet 协议与 802 协议间的区别由下列代码片段完成：

```
        if (ntohs(eth->h_proto) >= 1536)
            return eth->h_proto;

        rawp = skb->data;

        if (*(unsigned short *)rawp == 0xFFFF)
            return htons(ETH_P_802_3);

       /*
        *    真实802.2 LLC
        */
      return htons(ETH_P_802_2);
}
```

如果大于1,536的值会被解读成协议ID，那么，设备驱动程序该如何找出其接收的帧的大小？当协议/长度值小于1,500或大于1,536时，设备本身要把帧的大小存储至其寄存器之一，使得设备驱动程序可以读取。得益于为此目的而设的著名的位模式，设备可以搞清楚每个帧的大小。下列代码片段取自 *drivers/net/3c59x.c* 中的 vortex_rx，显示出驱动程序如何先从设备读取大小，然后再据此分配缓冲区的过程：

```
/* 封包长度：多达4.5K!  */
int pkt_len = rx_status & 0x1fff;
struct sk_buff *skb;

skb = dev_alloc_skb(pkt_len + 5);
```

不要被上述代码的注释搞糊涂了。这个特殊的设备可以接收的帧的尺寸可以多达4.5 K，是因为此设备也处理 FDDI NIC。

第一章已知何谓主机及网络字节尾端。由eth_type_trans返回的值，以及分派给skb->protocol的值采用网络字节尾端：从Ethernet报头中剥离出来时，已经处于网络字节尾端，此外，当 eth_type_trans 使用一个局部符号 ETH_P_*XXX*时，就必须以 htons 宏将其从主机字节尾端转换至网络字节尾端。也就是说，当内核稍后访问 skb->protocol，并且用来与 ETH_P_*XXX*值比较时，必须把 ETH_P_*XXX*转成网络字节尾端，或者把 skb->protocol 转成主机字节尾端：使用什么尾端并不重要，重要的是比较时两边都是以相同尾端表达。换而言之，下面两行代码是对等的：

```
ntohs(skb->protocol) == ETH_P_802_2
skb->protocol == htons(ETH_P_802_2)
```

因为 eth_type_trans 只会因 Ethernet 帧而被调用，其他媒介类型也有类似函数存在，有些名称的结尾是 _type_trans，而有些有不同的名称。例如，下面的范例取自 IBM Token Ring 驱动程序（*drivers/net/tokenring/ibmtr.c*）的一些代码，在大家熟悉的 netif_rx调用前，skb->protocol会被tr_type_trans设置，如同eth_type_trans为 Ethernet 设备所做的一样：

```
static void tr_rx(struct device *dev)
{
    ...
    skb->protocol=tr_type_trans(skb, dev);
    ...
    netif_rx(skb);
    ...
}
```

如果你看*net/802/tr.c*中的tr_type_trans，就会看到与eth_type_trans相似的逻辑，只不过对象是Token Ring设备。

还有些媒介类型会直接设置skb->protocol，而不用任何_type_trans辅助函数，因为这些媒介类型只能携带一种协议（如IrDA、AX25等）。

逻辑链接控制（LLC）

LLC层是由IEEE 802委员会在为LAN做标准化规格时设计的。其想法是，不要只用一个较高层协议标识符，而是要更有灵活性，可以指定来源地的协议标识符（SSAP）以及目的地的协议标识符（DSAP）。在大多数情况下，对于任何给定连接而言，SSAP和DSAP都是相同的——事实上，当全局标识被设置时，SSAP和DSAP都是相同的——但是，有两个值就使系统具有灵活弹性，可以使用不同的协议。

LLC可以为其上层提供各种不同服务类型：

类型I

无连接（如数据报文协议），不支持应答机制、流量控制以及错误恢复。

类型II

面向连接，支持应答机制、流量控制以及错误恢复。

类型III

无连接，但有类型II的一些优点。

图13-8（c）所示是使用LLC帧的报头格式。如你所见，其中有三个新字段：

SSAP

DSAP

这些是8位字段，用于指定所用的协议。

控制（CTL）

此字段的大小依赖于所用的LLC的类型（类型I或类型II）。我不计划深入探讨LLC层，但是会假设此字段有1个字节长，而且其值为0x03（类型I，CTL=UI）。这样就足以了解本章后续内容了。

LLC报头无法流行的原因有多个。也许主要原因在于SSAP和DSAP标识符的8位限制，更糟糕的是这些位还保留了两个给单播/多播以及局部/全局标识（注9）。只有64个协议可以在剩余的6个位内指定，这样实在太过限制。

使用局部SAP时（由协议字段中的局部/全局标识指定），网络管理员必须确保所有系统都同意其所使用的局部SAP，因而使事情更为复杂而且可用性也减少了。全局SAP不可能模棱两可，但是新协议都没有使用全局SAP。下一节你会看到这种限制如何因为使用了SNAP的概念扩充了报头而获得解决。

表13-3是和Linux内核注册的SAP。与表13-2中所列并且以dev_add_pack注册的协议相比，LLC使得内核在获取处理函数时使用了一个额外的层。

表13-3：内核的802.2 SAP客户程序

协议	SAP	函数处理
SNAP	0xAA	snap_rcv
IPX	0xE0	ipx_rcv

IPX的情况

你可能会想，能否使用纯粹的802.3帧格式，因为图13-8（b）中没有表示协议ID的地方。事实上，通常不会使用纯粹的802.3帧。唯一众所周知的例外涉及IPX。IPX封包可以通过未加工的802.3帧（也就是没有LLC报头的帧）而传送。接收者会通过一种手段识别这些封包。IPX报头的第一个字段是一个16位的校验和字段，正常来讲通过简单地设成0xFFFF予以关闭。因为0xFF/0xFF（注10）是无效的SSAP/DSAP组合，而且没有Ethertype有该值，因此，使用未加工的802.3的IPX封包就可轻易被识别。当这些封包被侦测出来时，skb->protocol就会被设成ETH_P_802_3，而其处理函数就是IPX处理函数（参见表13-1）。

Linux的LLC实现方式

在2.5版本开发周期中，802.2 LLC层不仅做了拓展而且还被重新改写。内核的LLC实现支持类型Ⅰ和类型Ⅱ，由下列主要组件组成：

注9：这两个标识的意义与先前讨论MAC地址时的相同，不过，这里是用于协议而非地址。

注10：检查0xFF/0xFF以识别出IPX封包在Linux内核中广为使用。eth_type_trans只是其中一例。

- 两个状态机。用于记录局部SAP以及建立在其上的连接的状态。
- LLC接收函数，根据其所接收的输入帧把正确的输入数据反馈到两个状态机。
- AF_LLC套接字接口。可用于在用户空间的LLC层的顶部建立协议或服务。

因为本书所说明的协议都没有使用LLC层，所以我不打算详细讨论LLC服务的定义（可以参考《IEEE 802.2 Logical Link Control》规范（注11）），此外，我也不会深入探讨Linux内核的LLC实现的细节。这里我将只谈定义局部SAP所用的数据结构，以及简单说明一些输入帧是如何处理的。

使用定义在*include/net/llc.h*中的数据结构llc_sap定义局部SAP。其中一些字段如下：

struct llc_addr laddr

SAP标识符。

int (*rcv_func)(struct sk_buff *, struct net_device *, struct packet_type *)

函数处理。当一个SAP通过PF_LLC套接字开启时，此字段为NULL。当SAP由内核开启时，此字段指向内核所提供的函数（参见表13-3）。

局部SAP由llc_sap_open建立，并被插入到llc_sap_list列表中。调用llc_sap_open时，可建立两种类型的SAP：

- 由内核本身所安装的SAP，安装内核层次的处理程序（注12）（参见表13-3）。
- 由PF_LLC套接字所管理的SAP（例如，当服务器在PF_LLC套接字上使用bind系统调用以将其绑定至给定的SAP）。

处理入口的LLC帧

每当送进来的帧被eth_type_trans归类为使用LLC报头时（因为其类型/长度字段小于1,536，而且没有侦测到特殊的IPX情况），skb->protocol初始化为ETH_P_802_2之后就会导致llc_rcv处理程序被选中（参见表13-1）。此处理程序会根据LLC报头内的DSAP字段选出正确的协议处理程序：为此，它会为那些由内核所开启的SAP调用以llc_sap_open注册的rcv_func处理程序，然后当SAP由PF_LLC套接字开启时（参见图13-10），就把正确的输入数据输入到正确的状态机。

注11： 如同大多数IEEE文档一样，有关LLC设计的文件也相当大，且读起来乏味。然而，手中有这样的文档后就比较容易看懂LLC代码，尤其是研究那些无聊的状态机细节。

注12： 也可以通过register_8022_client函数间接完成。

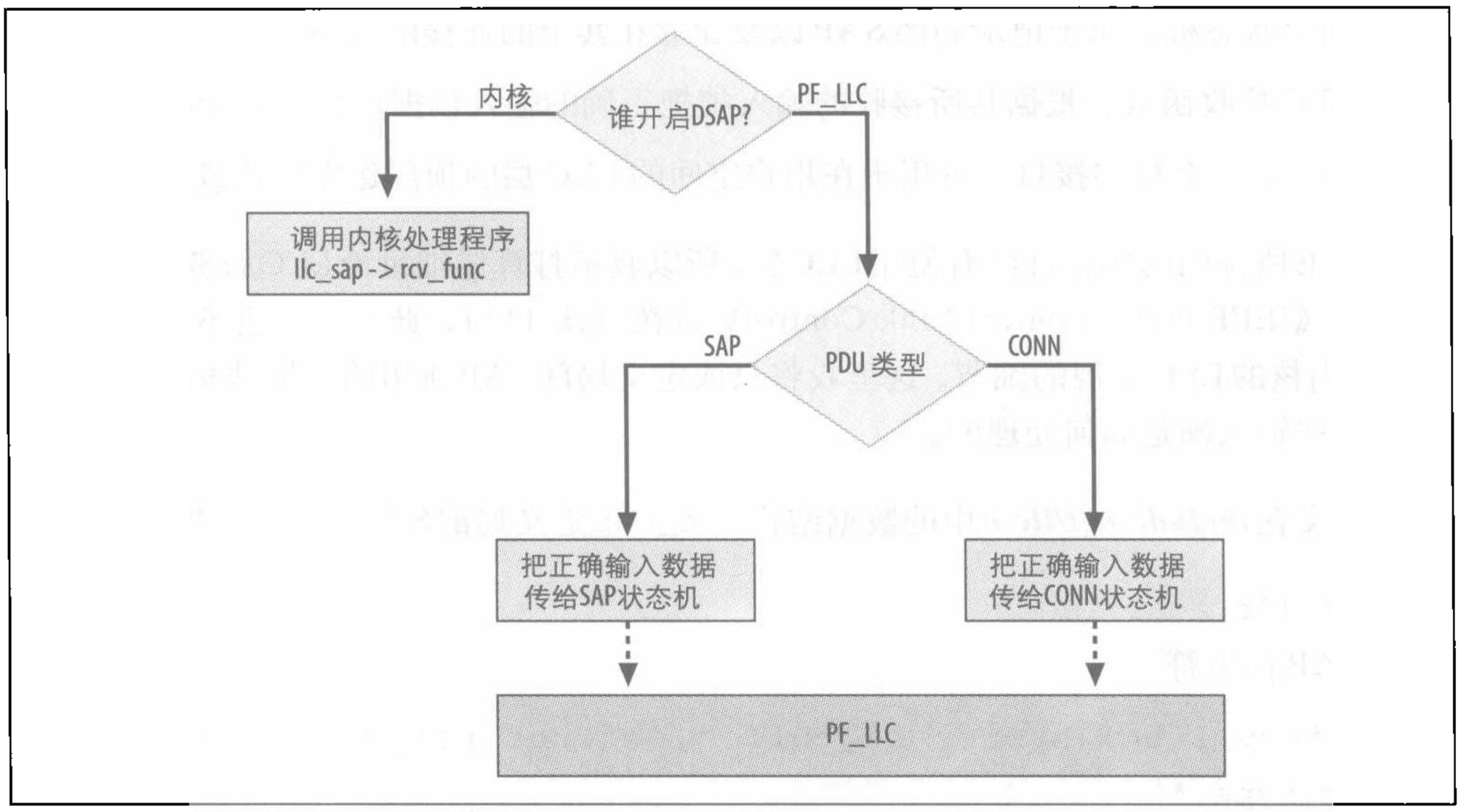


图 13-10：llc_rcv 函数

当两个状态机之一需要时（例如，用以通知接收到一个帧），就会由帧送出给定的SAP。PF_LLC 套接字可以使用标准接口（如 sendmsg）进行传输。在这两种情况下，适当的链路层报头已正确初始化后，帧就会直接输入到 dev_queue_xmit。

子网访问协议（SNAP）

由于LLC报头的限制，802委员会又把数据链路报头进一步拓展。为了让协议的范围更大，还引入了 SNAP 的概念。基本上，当 SSAP/DSAP 分派了 0xAA/0xAA 值时，就有了特殊意义：跟在LLC报头的CTL字段之后的5个字节表示一个协议标识符。单播/多播以及局部/全局位已不再使用。因此，协议标识符的大小从 8 位跳变到 40 位。委员会决定使用5个字节的原因与MAC地址可以推导出多少协议编号有关（注 13）。与SSAP/DSAP 不同的是，SNAP 编码的使用相当普遍。

因为 SNAP 标识符 0xAA/0xAA 是 SSAP/DSAP 的特例（如表 13-3 所示），因此，SNAP 是使用 `llc_sap_open` 的客户之一（参见 *net/802/psnap.c* 中的 `snap_init`）。这意味着使用 SNAP 编码的协议将多一层间接性，即有三层！

注 13： SNAP编码被定义成MAC地址的一个子集，而IEEE一批批地将MAC地址销售出去。因此，每位MAC地址拥有者都有一些SNAP编码连同那些MAC地址分派给他。更多细节建议阅读《Interconnections, Second Edition》（Addison Wesley 出版）一书。

在了解SNAP客户如何和内核注册之前，我们先简述如何定义SNAP协议ID。你可能已经知道，MAC地址由IEEE管理，而IEEE分2^{24}批将其卖出去。因为一个MAC地址是48位长（6个字节），因此，IEEE给予每个客户一个24位长的数字（也就是MAC地址的前3个字节），然后让客户可以把任何值放进剩下的24位。假设因为我想销售网卡，所以想买一批MAC地址。我们称分派给我的数字为XX:YY:ZZ。此时，我会成为XX:YY:ZZ:00:00:00与XX:YY:ZZ:FF:FF:FF之间的所有地址的拥有者。除了这些2^{24}个MAC地址，我还将被分派界于XX:YY:ZZ:00:00与XX:YY:ZZ:FF:FF之间的2^{16}个SNAP编码。

当你从IEEE得到一个24位长的数字时，由于局部/全局以及单播/多播位的四种可能组合，实际上是给了你4个24位长的数字（参见图13-9所示）。

采用与SAP协议类似的注册和除名方式，SNAP层也提供register_snap_client和unregister_snap_client函数，也使用一个全局列表(snap_list)把所有和内核注册的SNAP协议链接。表13-4所示是利用Linux内核注册的客户。

表13-4：SNAP客户

协议	Snap ID	函数处理例程
AppleTalk地址解析协议	00:00:00:80:F3	aarp_rcv
AppleTalk数据段传递协议	08:00:07:80:9B	atalk_rcv
IPX	00:00:00:81:37	ipx_rcv

定义在*include/net/datalink.h*中的数据结构datalink_proto用于定义SNAP协议。其中一些字段如下：

unsigned short header_length

这是数据链路报头的长度。在register_snap_client中，其初始值设为8，参见图13-8（d）所示。

unsigned_char type[8]

协议标识符。只使用了5个字节（SNAP协议ID；参见表13-4）。

void (*request)(struct datalink_proto *, struct sk_buff *, unsigned char *)

在register_snap_client中，其初始值设为snap_request。此字段会对SNAP报头做初始化（只有协议ID），并把帧传给802.2代码。在传输用于填写数据链路报头的数据之前，此字段的函数就会被启用。

```
void (*rcvfunc)(struct sk_buff *, struct net_device *, struct
packet_type *)
```

入口流量所需的函数处理函数。参见表 13-4。

我只粗略地探讨IPX。值得指出的是，该协议在三种不同场合会以相同的处理函数向内核注册：

- 有 Ethertype 时（见表 13-2）。
- 作为 802.3 SSAP/DSAP 协议（见表 13-3）。
- 作为 SNAP 协议（见表 13-4）。

图 13-11 总结了内核如何识别并处理 Ethernet、802.2、802.3 以及 SNAP 帧。

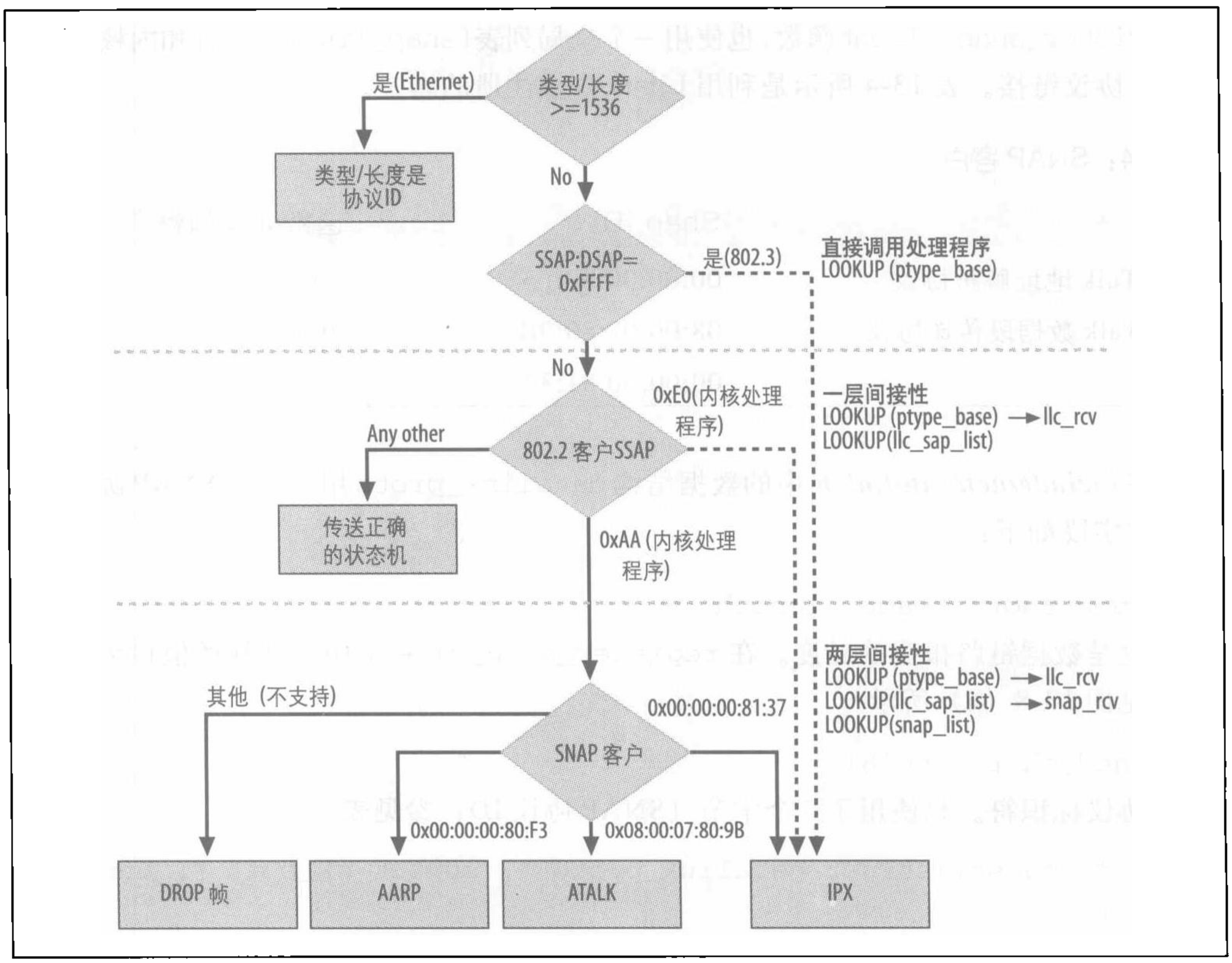


图 13-11：协议侦测 Ethernet/802.3/802.2/SNAP 帧

通过/proc文件系统进行调整

就Ethernet和802而言，*/proc/sys/net*中都有一个目录，*/proc/sys/net/ethernet/*（空的）和*/proc/sys/net/token-ring/*（只有一个文件）分别在*net/core/sysctl_net_ethernet.c*和*net/802/sysclt_net_802.c*中注册。只有当内核编译成支持Ethernet和Token Ring时，这两个目录才会包含进来。

本章涉及的函数和变量

表13-5是本章介绍的主要函数及最重要的全局变量和数据结构。

表13-5：涉及协议处理程序管理的函数和数据

名称	说明
函数	
dev_add_pack dev_remove_pack	添加/移除协议处理函数
register_8022_client unregister_8022_client	为802.2协议注册/除名。这些函数被定义成包裹函数，内含llc_sap_open和llc_sap_close
register_snap_client unregister_snap_client	为SNAP客户注册/除名
llc_sap_open llc_sap_close	创建/移除SAP
eth_type_trans	由Ethernet设备使用以剥离较高层协议标识符，然后把帧归类为单播/多播/广播
变量	
netdev_nit	已注册的协议探嗅器的数目
ptype_base	指向内含已注册协议处理例程的数据结构的指针
ptype_all	同ptype_base，但是用于协议探嗅器
snap_list	SNAP客户列表
数据结构类型	
struct packet_type	用于存储有关ETH_P_*XXX*协议处理函数的信息
struct datalink_proto	用于表示SNAP协议

本章涉及的文件和目录

图 13-12 所示是本章所涉及的文件的位置。在 *include/linux* 目录中，可以找到其他媒介类型的 *if_xxx.h* 头文件。*net/llc* 目录中包含多个文件。

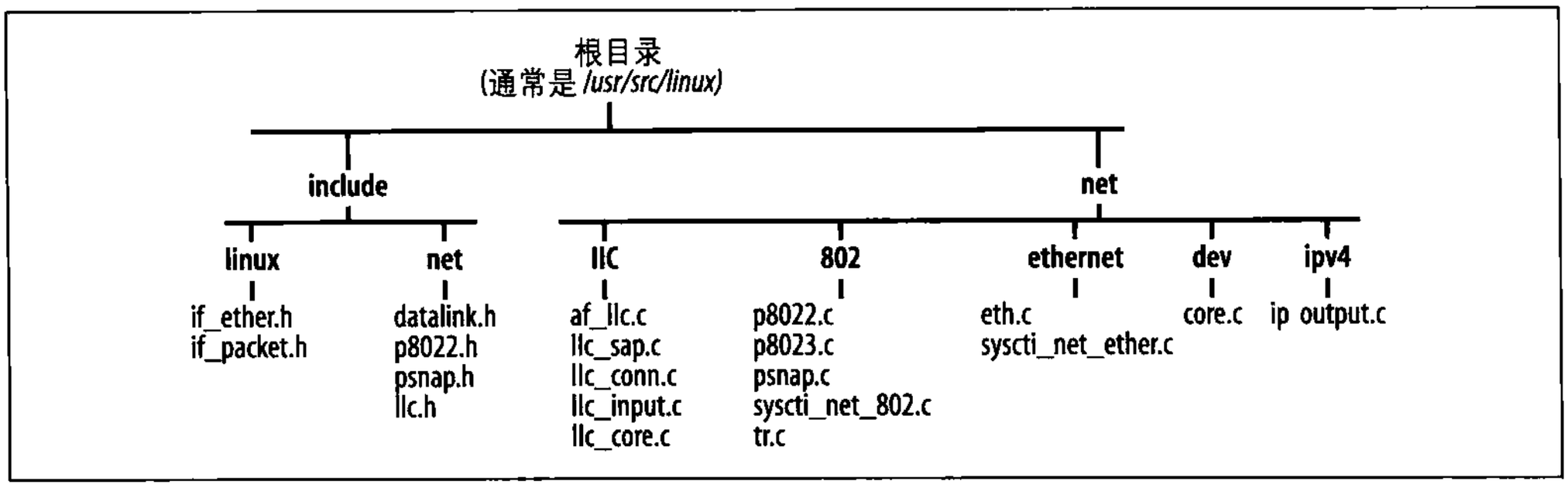


图 13-12：本章涉及的文件和目录

第四部分

桥接

在第三层，各种协议（例如，IPv4）通过第七部分讲的路由子系统连接到不同的网络。本书这一部分，我们将看到链路层或路由在L2层的对应体：桥接。这部分包括：

第十四章 桥接：概念

介绍透明地址学习（transparent learning）和选择性转发（selective forwarding）的概念。

第十五章 桥接：生成树协议（Spanning Tree Protocol）

介绍生成树协议（STP）如果解决网桥局限性，以及新版STP协议的增强功能（Linux还没有实现）。

第十六章 桥接：Linux 实现

介绍Linux如何实现桥接和STP。

第十七章 桥接：其他主题

讨论桥接程序如何与其他网络子系统交互，以及详细描述桥接程序使用的数据结构。

第十四章

桥接：概念

本章是讨论桥接的第一章，主要说明什么是网桥设备、网桥设备如何使用以及网桥设备有何限制。尤其要说明透明化桥接（transparent bridging）、地址学习（address learning）以及转发数据库（forwarding database）的使用。本章最后会说明为什么网桥不能在有环路的网络拓扑上运行，并在下一章介绍STP（Spanning Tree Protocol，生成树协议）如何解决这一限制。也存在其他的桥接形式，但它们很少被使用，而且Linux内核也不支持。

本章使用的网络拓扑并不一定代表真实场景；选择这些网络拓扑只是为了讲解基本原理。

中继器、网桥以及路由器

在介绍网桥之前，我将先澄清一下可以进行封包转发的不同网络设备间的区别：中继器、网桥以及路由器。其区别如图14-1中所示。

- 中继器（repeater）通常配有两个端口，只是简单地把在一个端口所接收的东西复制到另一个端口，反之亦然。它复制数据时是按位复制，并且对协议毫不关心，因此它无法区分不同的帧或封包。如今，中继器已很少使用，因为网桥已经变得很便宜，而且能提供更好的性能。多端口的中继器又称为集线器（*Hub*）。
- 网桥（bridge）和中继器不同，它了解链路层协议，因此可以按帧复制数据，而非按位复制。也就是说，网桥在每个端口上至少可以缓存一个帧。大多数LAN都用网桥 —— 也常称为交换机（switch）；参见“网桥与交换机”一节 —— 实现。该设备是本章的主角。
- 路由器（router）是了解L3网络层协议（例如IP）的设备，它可以根据路由表转发入口封包。在使用路由器 之前，通常使用术语“网关（gateway）”指这类设备。本书第七部分会深入探讨Linux中的路由实现机制。

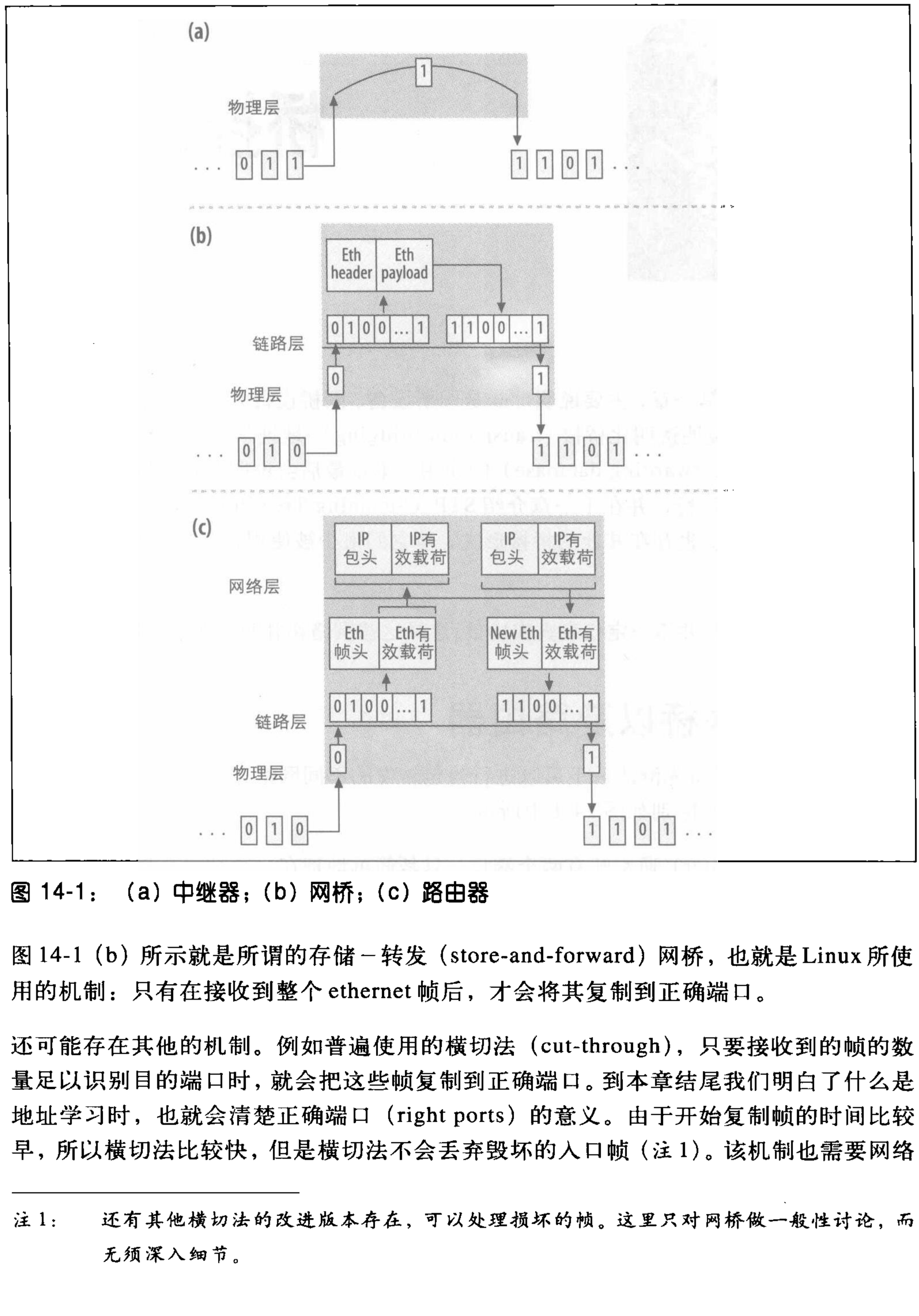


图 14-1：（a）中继器；（b）网桥；（c）路由器

图 14-1（b）所示就是所谓的存储－转发（store-and-forward）网桥，也就是Linux所使用的机制：只有在接收到整个 ethernet 帧后，才会将其复制到正确端口。

还可能存在其他的机制。例如普遍使用的横切法（cut-through），只要接收到的帧的数量足以识别目的端口时，就会把这些帧复制到正确端口。到本章结尾我们明白了什么是地址学习时，也就会清楚正确端口（right ports）的意义。由于开始复制帧的时间比较早，所以横切法比较快，但是横切法不会丢弃毁坏的入口帧（注 1）。该机制也需要网络

注 1：　还有其他横切法的改进版本存在，可以处理损坏的帧。这里只对网桥做一般性讨论，而无须深入细节。

接口卡（network interface card，NIC）硬件的配合。在当前模型下，NIC会把整个帧传给设备驱动程序。

网桥为它的每个接口指定一个链路层地址，然后，把所有不是给自己的东西都转发出去（路由器在L3层上的功能与此类似，本书第七部分会讲述）。但是，难道没有任何帧的目的地就是该网桥的接口么？毕竟，网桥的目的是协助帧通往其他目的地。因此，在下面两种情况下网桥也会消费（consume）一些入口帧：

将帧传往上层协议（也就是L3）

当网桥也实现L3功能（也就是该设备除了是台网桥，也是路由器或者主机），并且入口帧的目的地就是该接收接口所配置的L2地址时，才有可能出现这种情况。

将帧传给某个协议处理程序

在第十五章介绍生成树协议时，将会看到这种情况。

网桥与交换机

术语网桥（bridge）和交换机（switch）可以指相同的设备。但是，在讨论网桥的行为以及STP（后面的章节会看到）的运行方式的文档中（例如本章末尾提到的IEEE规范），主要使用术语“网桥”。相反的，当指那些真实的物理设备时，通常都是使用术语“交换机”。

我曾见过别人以术语“网桥”表示过网桥设备的唯一情况是，这种设备只配备两个端口（而今配备两个端口的网桥已经不常见）。这也就是为什么我自己时常把交换机称为多端口网桥的原因了。除非你对正式的IEEE规范很熟悉，你可能会使用术语“交换机”。我本人做网桥软件已经好几年了，在我记忆里，只有在撰写说明文件时我才会使用术语“网桥”，而从来没有在任何实际网络安装中用其指代一个设备。

一般而言，我可以这么说，网桥和交换机没有什么实质差异。

现在网桥已经很常见。你可以发现各种不同端口数量的网桥（价格与端口数量相关）。如今，Ethernet网桥代表着实施LAN时最常见的方式。

在一台运行Linux的PC上，只要多加一块NIC，就可以当作一台网桥。你也可以在市场上购买一块多接口的PCI接口的NIC，让你的网络接口数目超过PCI插槽数目。例如，在一块PCI NIC上，可以有四个Ethernet端口。

在本章后续部分及后面几章中，我都会使用术语“网桥”。但是，现在你应该知道办公室中用于连接PC和网络打印机的交换机只不过是个多端口网桥而已，不过，为了提供其他功能，这个网桥上跑的软件可能比纯粹的网桥还要多。

主机

在本章以及后续几章（包括路由器）讨论网桥的背景下，任何运行在高于网桥所属层（也就是链路层或L2）的网络层上的设备，就视其为主机。一台主机（比如Linux系统）只要经过适当配置，就可以作为网桥或者独立主机（用户可将其作为标准工作站使用）。但在本章以及后续几章中，我们不会考虑这种情况；我们假定主机就是没有运行任何网桥程序的设备。因此，除非文中特别说明，图中所有PC都没有运行网桥程序。

用网桥合并LAN

我们以图14-2所示的环境为例，说明网桥如何将两个LAN合并在一起，使其看起来就像一个LAN。我们假设这两个LAN里的主机都属于同一个IP子网。在图中没有包含IP地址，因为我们的重点在于链路层所发生的事。

你应该注意到，LAN其实就是一种多端口网桥。

在LAN中的任何主机发出的任何帧都会被其他主机接收。因此，当主机A发出一个帧时，LAN1的其他主机和网桥都会接收到该帧。网桥将入口帧复制到其他所有端口上（本例中就是另一个端口）。因此，最后LAN1和LAN2上的所有主机都会接收到主机A所产生的帧的副本。也就是说，有了网桥，从LAN1和LAN2的主机的角度来看，就只有一个大LAN而已。网桥通常用于合并物理LAN，其主机都配置为相同的IP子网，因为网桥可以使那些主机产生单一LAN的幻觉。

注意，网桥会转发收到的入口帧。它不会对其做增加、删除、或修改内容的行为：LAN2里的主机接收的就是主机A所产生的原帧的精确副本。

你可能认为，主机A要传给主机B的封包并不需要转发至LAN2，这样会浪费LAN2的带宽和LAN2主机的CPU的能力（因为那些主机最后都会丢弃不是传给它们的帧）。如果把两个LAN的主机设置成两个不同的IP子网，然后用路由器替换网桥，这种浪费就消除了，如图14-3所示。这是因为，路由器不会把那些要传给LAN1中的主机的封包转发至LAN2。

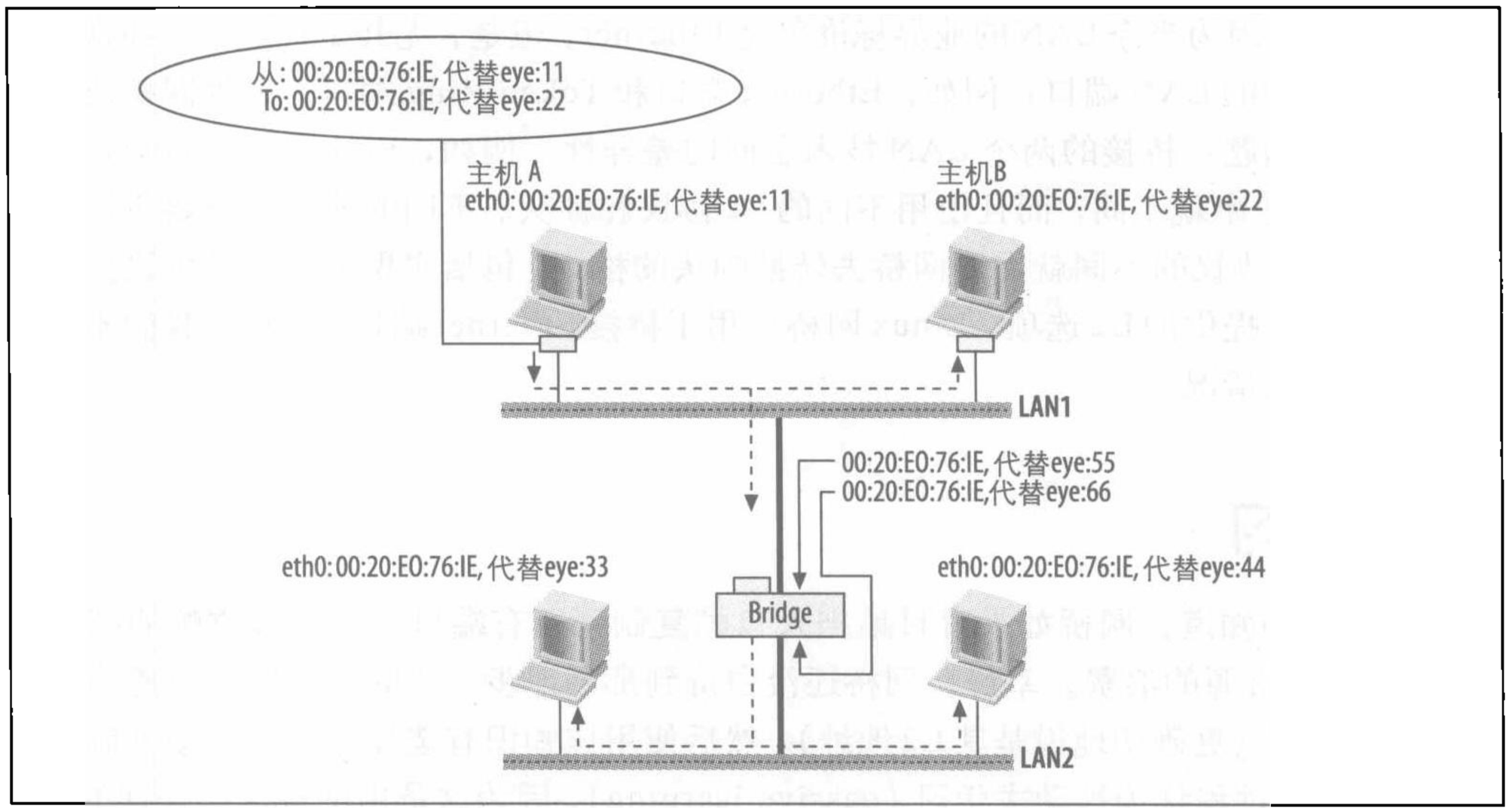


图 14-2：用网桥合并两个 LAN

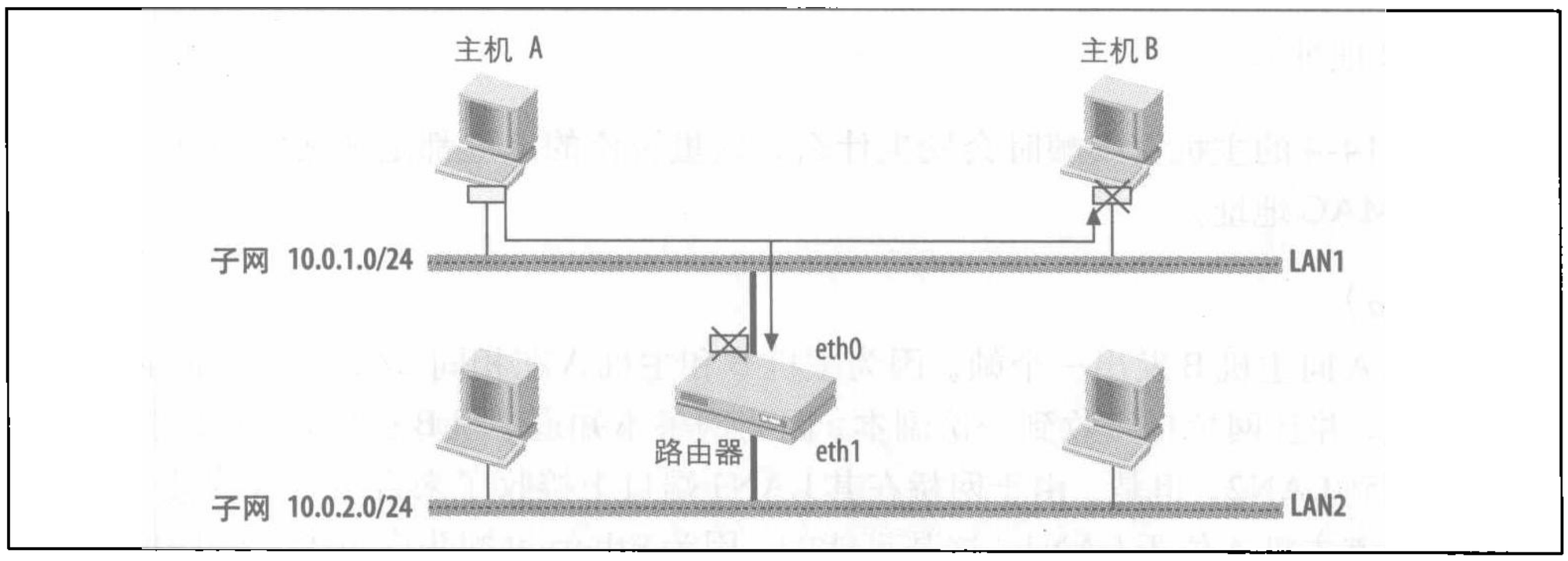


图 14-3：用路由器连接的两个 LAN

图 14-2 和图 14-3 的拓扑分别用在不同情况下。第一种拓扑就是让一些处在不同 LAN 中的主机共享相同的L2以及相同的IP（L3）子网。第二种拓扑可以把主机隔离在不同的子网内，也许是基于管理方面的原因。

注意，图 14-2 的主机还是需要路由器才能访问子网外的 IP 地址。

桥接不同技术的 LAN

在前面几个例子中，网桥的两个端口都连接到 Ethernet LAN 上。这种网桥类型是最常

见的，主要是因为当今LAN的业界标准就是Ethernet。但是，尤其是在过去，经常是用网桥连接不同的LAN端口；例如，Ethernet端口和Token Ring端口。这种网桥还得再多考虑一个问题：桥接的两个LAN技术之间的差异性。例如，Ethernet和Token Ring LAN的运行速率就不同，而且使用不同的L2协议和帧头。不同的速率就需要应用一些缓冲机制，而协议的不同就需要网桥去转换帧头的格式，包括实现那些一种协议提供而另一种协议不提供的L2选项。Linux网桥只用于桥接Ethernet端口，所以，我们不考虑其他更复杂的情况。

地址学习

从前几节我们知道，网桥如果盲目地把入口帧复制到所有端口（那个接收帧的端口除外），会导致资源的浪费。幸好，网桥还没白痴到那种地步。实际上，网桥有能力去学习主机的位置（更确切地说是其L2地址），然后使用该知识有选择地把入口帧复制到正确端口。这个过程称为被动式学习（passive learning），因为这是由网桥自行处理的，而不需要用户配置或协议的帮助。通过图14-4，我们来说明被动式学习的运行方式。为了让该图易于理解，图中使用符号“Host N”表示L2地址（即没有像图14-2那样显示出真正的L2地址）。

来看当图14-4的主机交换帧时会发生什么。这里讨论的地址都是链路层地址（也就是Ethernet MAC地址）：

图14-4（a）

主机A向主机B发出一个帧。因为主机B和主机A在相同LAN中，因而它接收到该帧，并且网桥也接收到一份副本。因为网桥不知道主机B在何处，所以把该帧也复制到LAN2。但是，由于网桥在其LAN1端口上接收了来自主机A的帧，现在网桥知道主机A位于LAN1。这是可能的，因为Ethernet帧头内包含了源地址和目的地址。

图14-4（b）

主机B向主机A发出一个帧。主机A和网桥都接收到该帧。因为网桥已知道主机A在LAN1内，而其接收的帧也来自于相同LAN，所以网桥不会把帧复制到LAN2。

图14-4（c）

主机A向主机C发出一个帧。主机B和网桥都会接收到帧。主机B丢弃该帧，因为主机B不是接收者，而网桥会把该帧复制到LAN2，因为网桥不知道主机C在何处。但网桥已知道主机A位于LAN1，因此，网桥不需要在LAN1端口的可到达地址列表中添加任何项目。

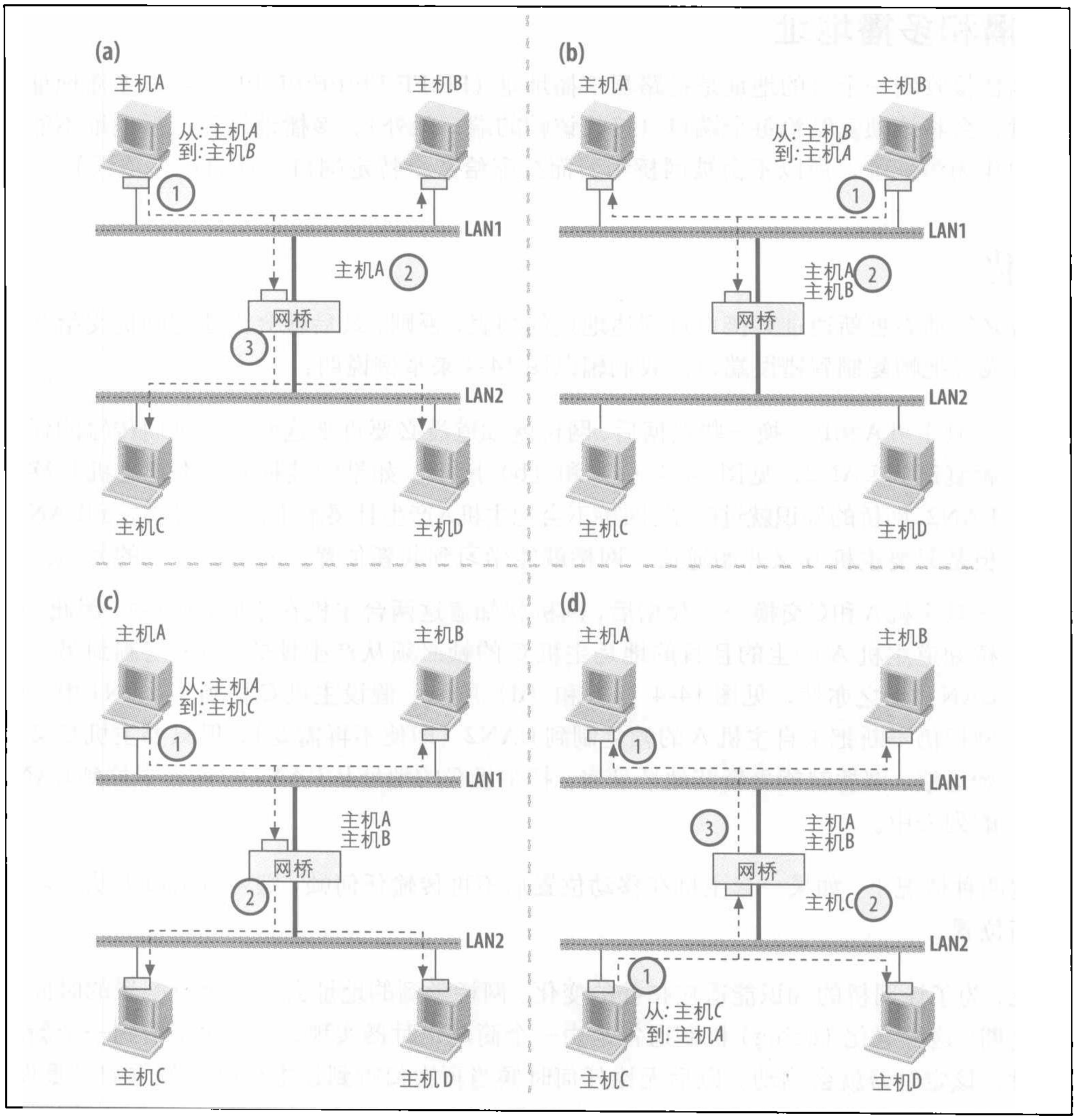


图 14-4：地址学习范例

图 14-4（d）

主机 C 向主机 A 发出一个帧。主机 D 和网桥都会接收到其副本。主机 D 会丢弃该帧，因为它不是接收者，而网桥会将其复制到LAN1，因为网桥知道主机A在LAN1内。

当网桥不知道该用哪个接口来连接特定的L2地址时，就会把帧复制到所有接口上（接收该帧的接口除外），这种行为就称为扩散（flooding）。

广播和多播地址

当网桥接收到一个目的地址是链路层广播地址（FF:FF:FF:FF:FF:FF）或L2多播地址的帧时，会将该帧复制给每个端口（接收该帧的端口除外）。多播地址和广播地址不能在帧中作为源地址，所以不会被网桥学习而分配给某个特定端口（这将产生错误）。

老化

网桥必须动态更新通过其接口可到达地址的列表；否则，最后就会无法把帧提交给其接收者或者把帧复制到错误端口。我们辅以图 14-4 来举例说明：

- 一旦主机A和B交换一些数据后，网桥就知道没必要再把这两台主机间传输的任何帧复制到 LAN2，见图 14-4（a）和（b）所示。如果因某种原因你把主机 B 移到LAN2，网桥的知识就过时了：网桥不会把主机A产生且要给主机B的帧转发到LAN2。但是只要主机 B 又开始通信，网桥就能学习到其新位置，而更新自己的知识。
- 一旦主机A和C交换一些数据后，网桥就知道这两台主机在不同LAN中。因此，网桥知道主机 A 产生的且目的地是主机 C 的帧必须从产生帧的 LAN 复制到另一个LAN；反之亦然，见图 14-4（c）和（d）所示。假设主机 C 移到了 LAN1 中，则网桥仍不断把来自主机 A 的帧复制到 LAN2（即使不再需要）。但只要主机 C 又开始通信，网桥就能更新其地址列表，把主机C的地址从LAN2的列表中移到LAN1的列表中。

在这两种情况下，如果一台主机在移动位置后不再传输任何帧，则网桥就没办法学习到其新位置。

因此，为了让网桥的知识能适应拓扑的变化，网桥学到的地址会在一个可配置的时间量后过期。这种老化（aging）机制通常是由一个简单定时器实现。当首次学习到一个新地址时，该定时器就会启动。以后无论任何时候当再次侦听到该主机时，就重启（重设）该定时器，以便确认或更新其地址。这个流程如图 14-5 所示。定时器时间越短，网桥对地址变化的学习速度就越快。但是，这也表示网桥会时常发现自己不知道某个主机位于何处，这时必须使用扩散。默认的老化时间是 5 分钟。第十五章就会知道，在特殊情况下，老化时间会被STP缩短。此外，第十七章会说明系统管理员如何改变默认老化时间。

多台网桥

到目前为止，我们说明了只有一个网桥时的简单情景。由于透明网桥彼此之间以及和主机和路由器之间都是透明的，所以可以采用多台网桥以建立较大的 L2 网络（也就是较大的 LAN），如图 14-6 所示。

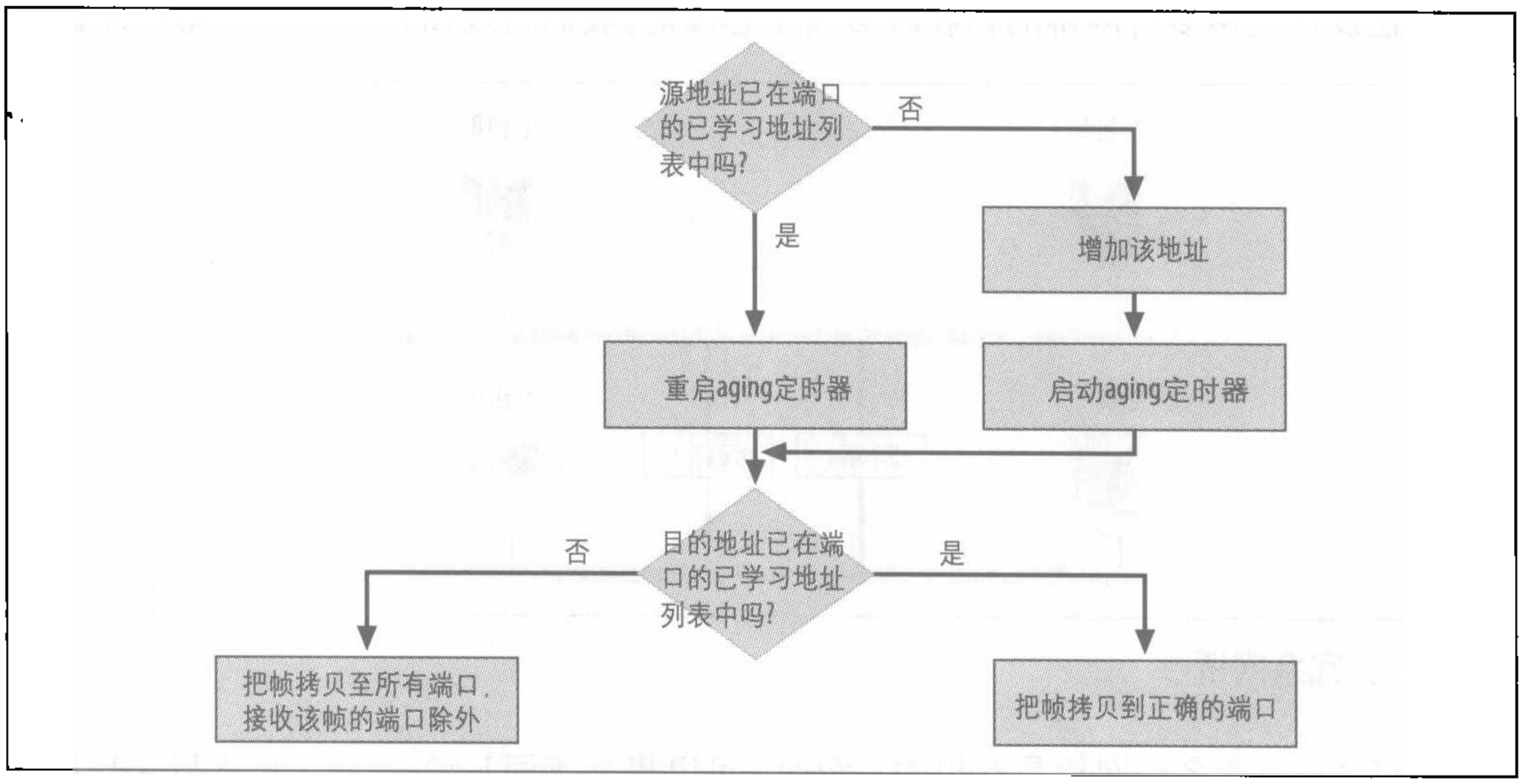


图 14-5：地址学习和老化

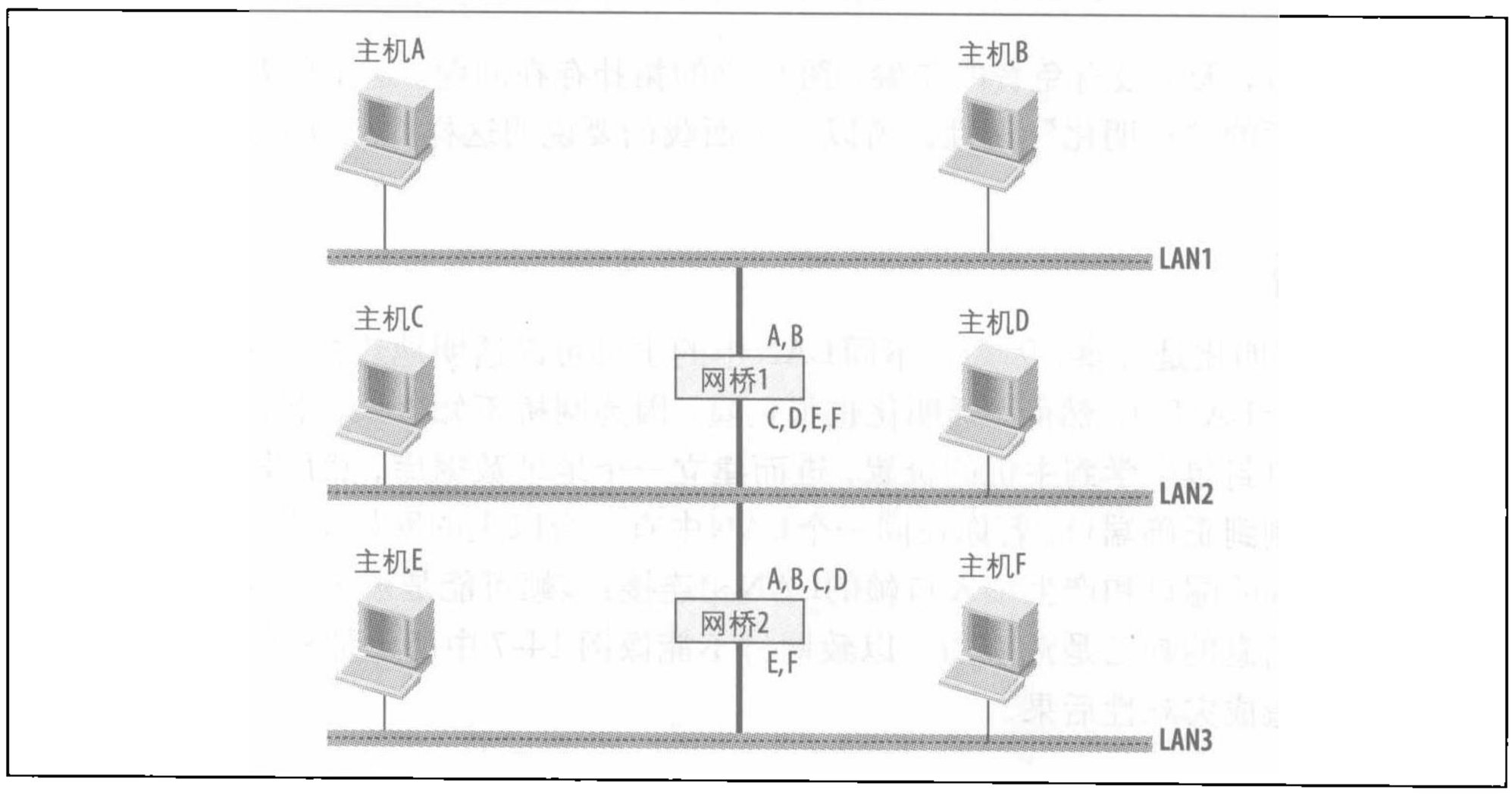


图 14-6：两台网桥的拓扑

图中还给出这两台网桥的每个接口学到的地址列表，假设每台主机至少通信一次，因此网桥就有机会知道其位置。例如，从网桥1的角度看，主机A、B、C以及D都位于LAN2中，或者换句话讲，都可通过网桥1在LAN2的接口访问。毕竟，网桥并不关心主机究竟在何处，它只需知道该用哪个端口连接该主机。

不过，在设计使用多台网桥的拓扑时必须小心谨慎。我们通过图14-7的范例说明其原因。

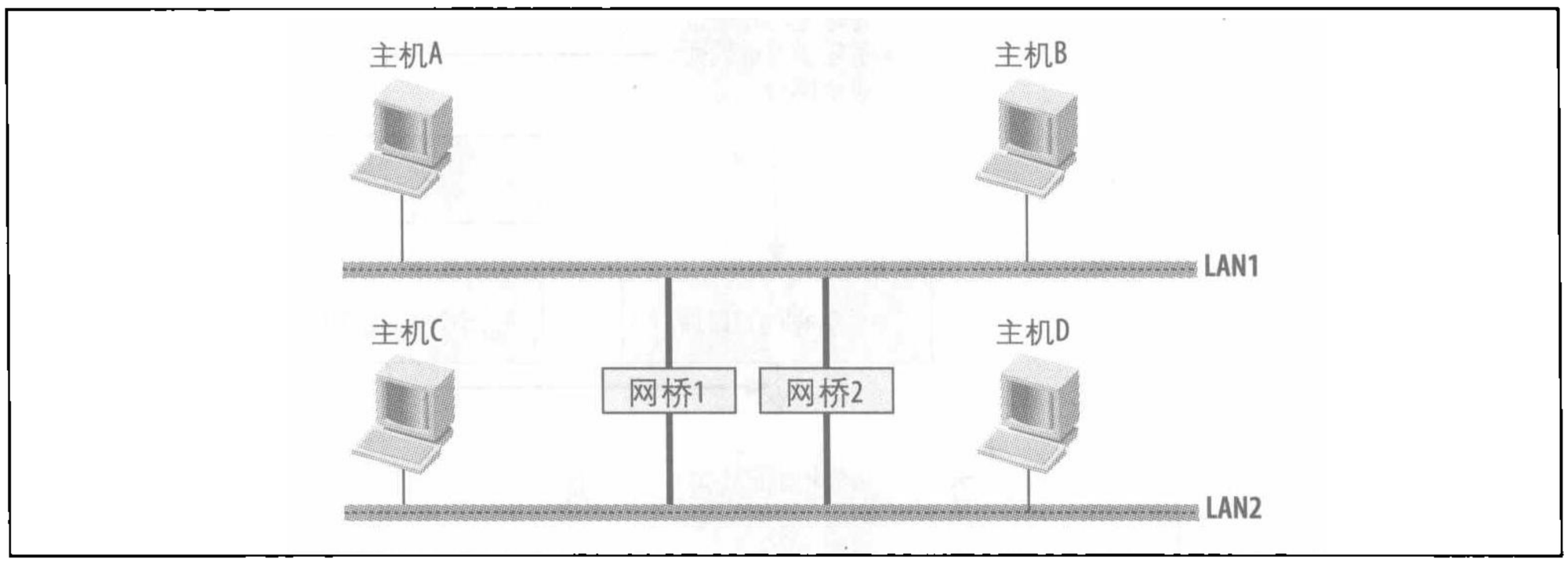


图 14-7：冗余网桥

同一个LAN中有多台网桥是有用的，例如，可以提高不同LAN之间（网桥与这些LAN上有接口）的连通能力。如果一台网桥因某种原因无法使用，其他网桥就可以维持这种连通性（connectivity）。该图给出的拓扑只有两台网桥，但是要再多几台也行。

不过你也知道，天下没有免费的午餐，图14-7的拓扑存在问题。这个问题来自于以前正面看待的网桥的“透明化”属性。所以，下面我们要说明这样的配置问题出在哪里。

网桥环路

总地来说，透明化是好事，因为在不同LAN中的主机可以透明地连接起来，仿佛这些主机都在同一个LAN中。然而，透明化也很危险，因为网桥不知道入口帧的来源。网桥的工作是从入口封包中学到主机的位置，进而建立一个地址数据库，然后根据这个数据库把入口帧复制到正确端口。若你在同一个LAN中有一台以上的网桥工作，就再也不能假设接收一个帧的端口和产生该入口帧的LAN相连接：该帧可能是被另一台网桥复制到那个端口的。信息的缺乏是危险的，以致网桥不能像图14-7中所示那样使用，因为这种安装方式会造成灾难性后果。

例如，来看看图14-8中的场景会发生什么事，其中，主机A发出一个帧，且网桥1和网桥2的数据库都是空的（也就是还没学到任何地址）。

这两台网桥都会接收到该帧，并了解到主机A在LAN1中，然后把帧复制到LAN2上。哪台网桥会先做此事是无法确定的，这取决于这两台网桥的负载情况。我们假设它们几乎是在相同的时间做复制。因此，这两台网桥会在其LAN2的接口上接收到帧的副本，之后便认为主机A已移到LAN2中（别忘了，它们在LAN2上所接收的帧就是LAN1的主机A所传输的那个帧）。此时，这两台网桥又将把帧复制到LAN1（我们假设目的主机

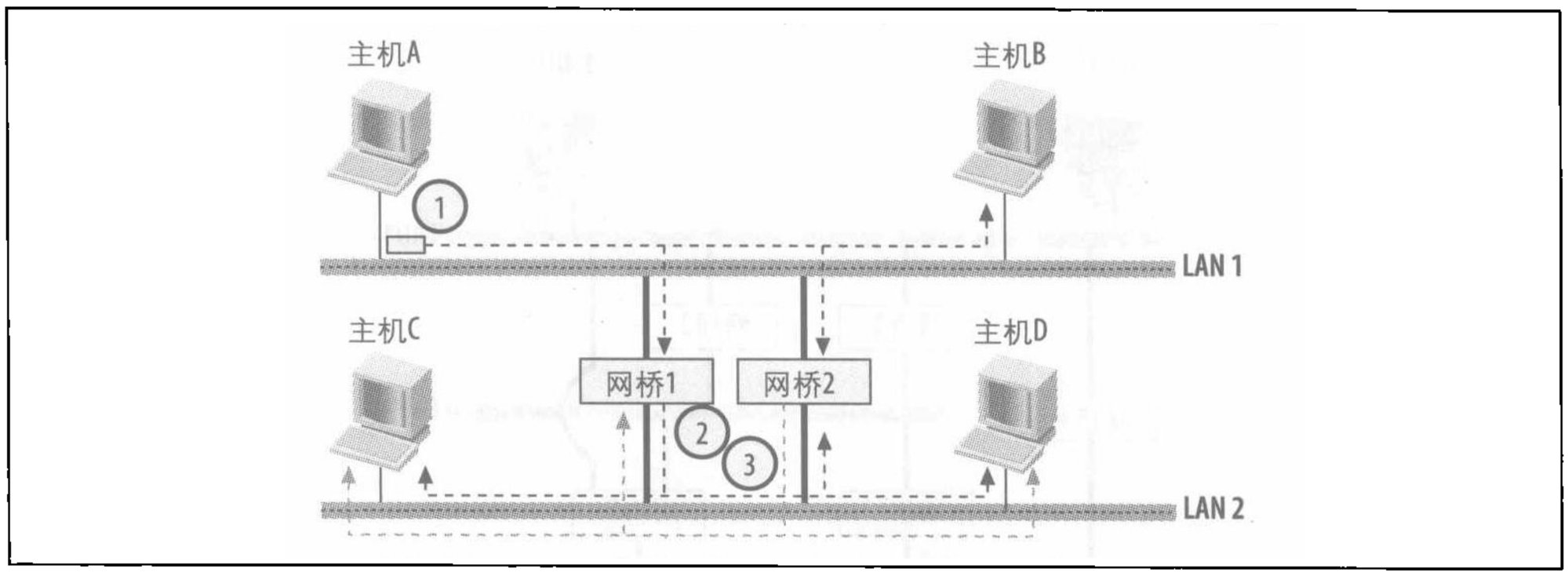


图 14-8：网桥环路

尚未应答，因此网桥还不知道该主机在何处）。接着，这两台网桥又会再接收彼此的副本，以为主机 A 的位置又变了，然后又把帧复制到 LAN2。

这样的环路会使得这两个LAN因同一个帧无限循环而导致扩散，使这两个LAN无法再进行其他传输。LAN中的其他主机的CPU也会忙于接收和丢弃同一个帧的大量副本，并且如果主机的接口层没有速率限制保护的话，那些主机的 CPU 终究会崩溃。

这个简单的场景告诉我们一条重要规则：透明网桥不能用在环路拓扑上。

无环路拓扑

让图 14-8 所示的拓扑得以运行的简单方法就是关闭网桥 2，并且只有当网桥 1 停止运行时才能够启动。但是这种解决方法无法给我们任何冗余度，因为这需要人工干预。常用的是另一种方法，也就是让网桥彼此都看见，不过依然让学习和复制任务保持透明，如前所述。

图 14-9 所示是稍微复杂的场景（注 2）。注意，网桥 5 在三个 LAN 上都有接口。显然，所有网桥都必须合作，因此，你不能只是简单地把网桥打开或关闭（注 3），必须用更为细致的手法定义出无回路的拓扑。所以，不是关闭网桥，而是关闭网桥的端口。图 14-9 中的拓扑并不表示常见的或建议采用的情景。然而，在类似这种混乱的情况下，网桥必须能够运行并提供无环路的连通性。

注 2：此图略去 LAN2 和 LAN3 中的主机，因为我想让你把重点放在网络拓扑上。在图中的所有 LAN 中的所有主机都会因环路拓扑的后果而受到影响。

注 3：如果你愿意，可以关闭网桥，但是会降低本来能达到的冗余度。

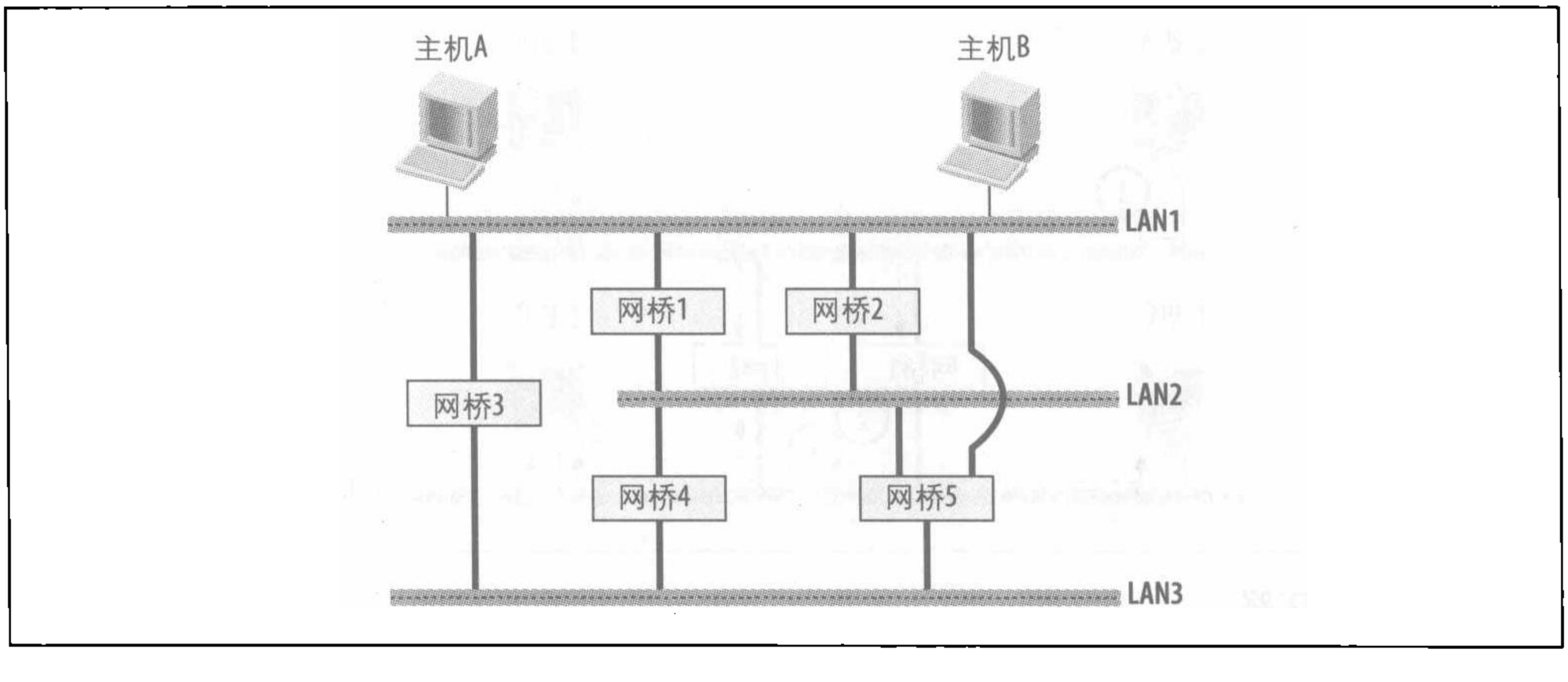


图 14-9：多环路的拓扑

我们回到图 14-8 中的简单例子，找出使其安全的桥接协议的特点。如果你画一张状态图，其中网桥和LAN都是状态，而网桥和LAN的连接（也就是网桥端口）都是双向的链路，那么，就会得到如图 14-10 所示结果。

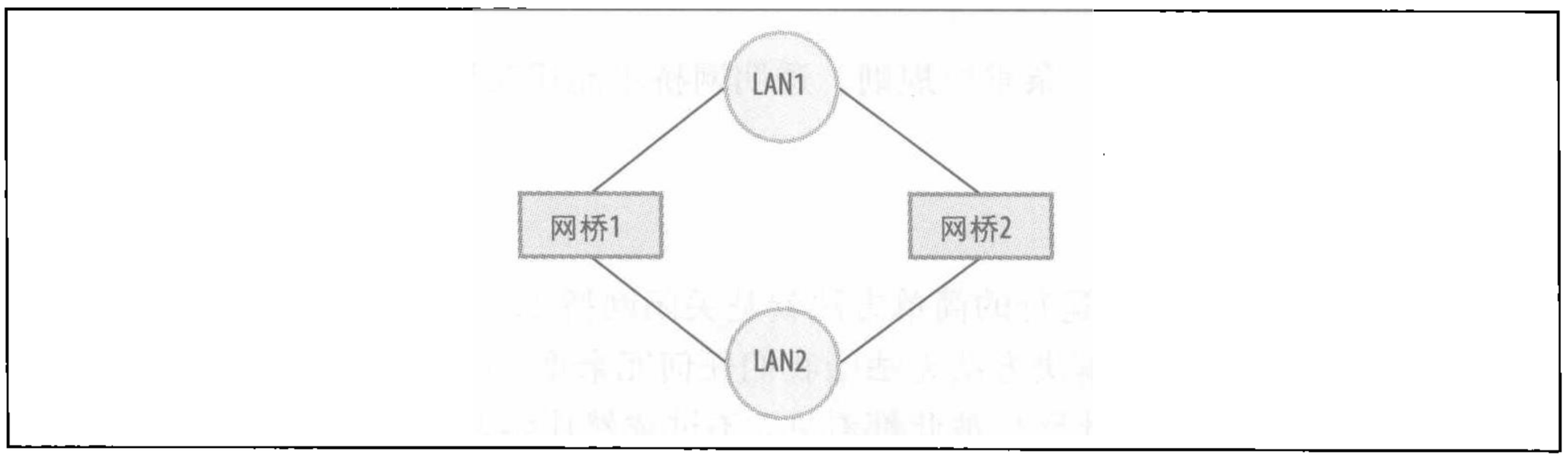


图 14-10：图 14-8 对应的拓扑图

要让图 14-10 的图形无环路，所需要做的就是关闭一个网桥端口并删除一条连接线路。图 14-9 所示拓扑的图形会更为复杂，而且包含多个环路（我算了一下，至少有 5 个）。

注意，有多种方式可让环路拓扑变成无环路拓扑。例如，把图 14-10 中的环路切断，就有 4 种不同的方法。

定义无环路拓扑

如果你熟悉图论，就知道已知一个图形（有连接开销）而要找出最佳无环路拓扑是个经典问题，且已经有一系列不同的很优雅的解决算法。然而，所有那些算法都是集中式的

(centralized)：算法依靠所有必要信息只运行一次。对我们的情况来说，网桥需要的是分布式算法。网桥通过交换某种信息关闭适当的端口后，必须能够收敛成无环路拓扑。

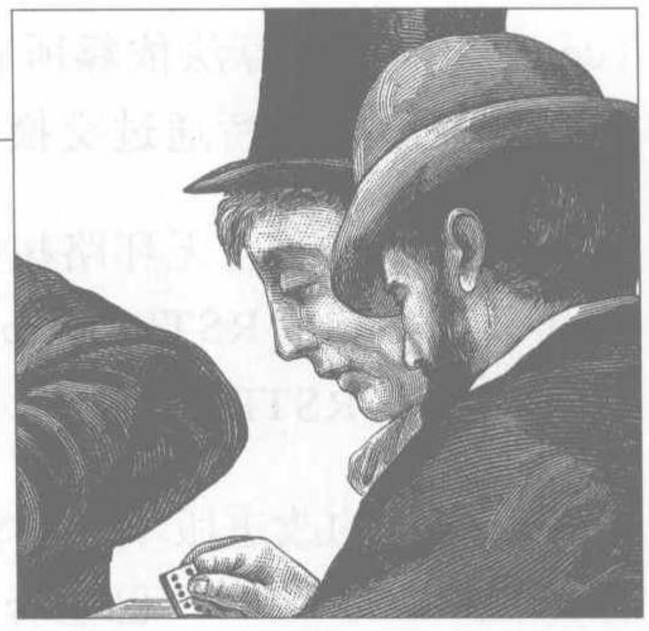
网桥用于找出最佳无环路拓扑的算法就是802.1D-1998 IEEE标准所定义的生成树协议，由其改进得到的RSTP（Rapid Spanning Tree Protocol，快速生成树协议）成为802.1D-2004标准。RSTP有时也称为802.1w。

另一个有趣的改进版本是MSTP(Multiple Spanning Tree Protocol,多重生成树协议),也称为802.1s。MSTP已整合到802.1Q-1998中，后来802.1Q-1998又演变为802.1Q-2002（后又成为802.1Q-2005，增加了新的变化）。

为了简化起见，我们把这些协议称为STP（原来的生成树协议）、RSTP以及MSTP。第十五章会介绍STP，同时简单说明对RSTP和MSTP所做的改进。第十六章会深入说明STP的实现细节。

第十五章

桥接：生成树协议



由第十四章可知，透明桥接是一种合并LAN的简单方式，但是它只能用在无环路拓扑上。这种限制使得透明网桥无法用于为了增加整体可用性而使用了冗余连接的网络上。

本章我们将看到生成树协议（Spanning Tree Protocol，STP）如何让拓扑消除环路，也因此使得网络管理员可以使用有冗余链路的拓扑。我们要特别说明的是：

- STP所用的分布式算法如何通过关闭适当的冗余链路而得到无环路拓扑。STP所选择的无环路拓扑是一种树结构（按其定义就是无环路）。由网桥连接的LAN中的主机之间的所有流量都会沿着这棵树的链路传输。
- STP如何动态更新拓扑，以应付配置变化以及网桥或链路失效。
- 当检测到拓扑变化时，STP如何动态更新转发数据库（即在网桥端口所学到的那些地址）。

因为篇幅有限，我们无法详细讨论STP。本章的目标是给你足够详细的概要介绍，让你在阅读第十六章、讨论STP在内核中的实现时能轻松应对。有关802.1D STP及其改进的全面讨论，请参考IEEE规范。

本章所提供的范例并不能够作为配置STP的指南：这些范例大多数只是用于演示会出现哪些特殊情况，以及STP如何对其做出相应处理。

基本术语

先来定义一些关键术语，本章及后续一章都会用到：

LAN

这个术语不需要多做介绍。我们不仅以术语LAN指类Ethernet网络，也用于指点对点连接。

L2 网络

由网桥所合并的一个或一组LAN。如第十四章所述，使用网桥可以合并多个LAN，看起来就像是一个比较大的LAN（从路由器和主机的角度看，也就是从运行在较高的网络层设备的角度看）。

桥接式（或交换式）网络

一种用网桥实现的L2网络。

网桥端口

接口

在一台网桥设备上，每个网络接口都是一个网桥端口。在一个通用系统上，比如运行着Linux的PC上，网络接口并不需要总是作为网桥端口。在本章的中，术语“网桥端口”和“接口”可以交换使用，但是在第十六章中，我们必须区分桥接接口和非桥接接口。我将使用术语“接口”仅指非桥接的NIC。网桥的每个端口都可以用于连接主机和其他网桥。本章和后续几章会看到几个这样的例子。

链路

两台设备间的连接。本章中，我以这个术语特指两台网桥间的连接。

稳定网络

指L2网络，其STP已收敛成最终的无环路拓扑。

图15-1所示是本章所用之术语及其和日常网络术语之间的关系。

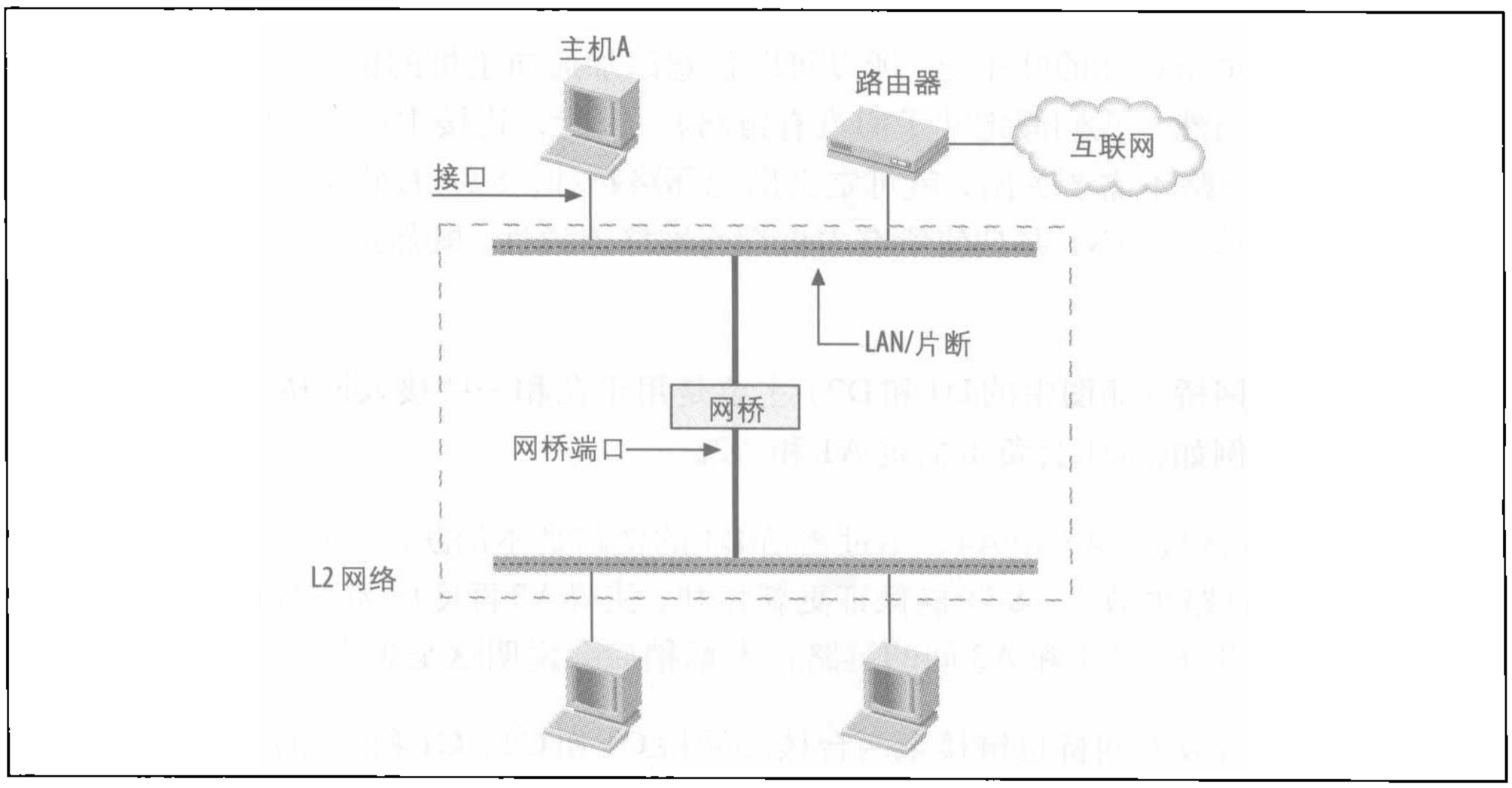


图15-1：基本术语

阶层交换式 L2 拓扑范例

我们知道，两台主机可以通过一条交叉网线彼此连接；不必要使用集线器（Hub）或网桥这类设备。网桥和路由器之间也可用相同的方式连接。在本章的范例中，你会时常看到网桥间以交叉网线连接的情况。

真正的桥接式网络与第十四章的模式(简单的双端口网桥直接和位于其所连接LAN中的主机相连）不同，它的拓扑通常很像一棵树，主机只（或主要）位于叶子节点上。

当有类似第十四章所述的简单情形时，通常不需要开启STP；实际上，大多数站点使用单一的平坦LAN，而完全不需采用网桥。为了更好地理解STP，我们需要知道现实生活场景是个什么样子。我们以图 15-2（a）所示的经典的阶层桥接式和冗余拓扑为例做说明，这种模式也是多数网桥生产企业主打和推广的模式。

在本章后半部分会说明该图中的一些细节，诸如网桥ID和优先级、端口开销以及优先级值，这样安排的目的是让你把重点集中在拓扑和活跃链路的选择上。本章后续的图形中，我们会多次使用位于该图底端的图例说明中的符号定义。

这棵树的叶子（此图底端）就是主机。主机链接到所谓的接入网桥（access bridge，通常也称为接入交换机）上：能把网络连通能力给予主机的网桥。接入网桥主要是用于在连到同一个网桥上的主机间转发流量，但是，接入网桥和上层网桥间也有一条或多条链路。图 15-2 中所示的接入网桥用 A1、A2、A3 以及 A4 标识。

由于主机总是位于拓扑的叶子上，所以可以任意添加通向主机的链路。这些链路不会造成任何环路（当然，我们假定叶子间没有链路）。因此，连接主机的链路不会受到 STP 的影响：这类链路不需要关闭，就可定义出无环路拓扑。STP 的最终目标就是让网络看起来像大型的单一LAN，并且使所有主机都有网络连通性。既然这样，为何要让任一主机断线呢？

处于分发层的网桥（即图中的D1 和D2）主要是用于在和一些接入网桥直接相连的主机间桥接流量。例如，D1 会负责搞定 A1 和 A2。

注意到，D1 也链接至 A3 和 A4，不过当前 D1 的链路是不活跃的（图中的虚线）。一旦 D2 和A3 间的链路失效了，STP能保证更新拓扑，使得A3 再度成为这棵树的一部分。例如，该网络可以开启 D1 和 A3 间的链路；本章稍后会说明这是怎么实现的。

D1 和D2 这两台分发网桥也链接至两台核心网桥 C1 和C2。C1 和C2 的主要工作是：将 D1 的子树和 D2 的子树相连（另一种做法是用一台功能更强大的核心网桥）。在分发层和核心层间也有冗余链路，比如，如果C1-D1 链路失效，C2-D1 就会取而代之。网桥所在的层越高，所处理的流量就越大（因为子树会越大）。

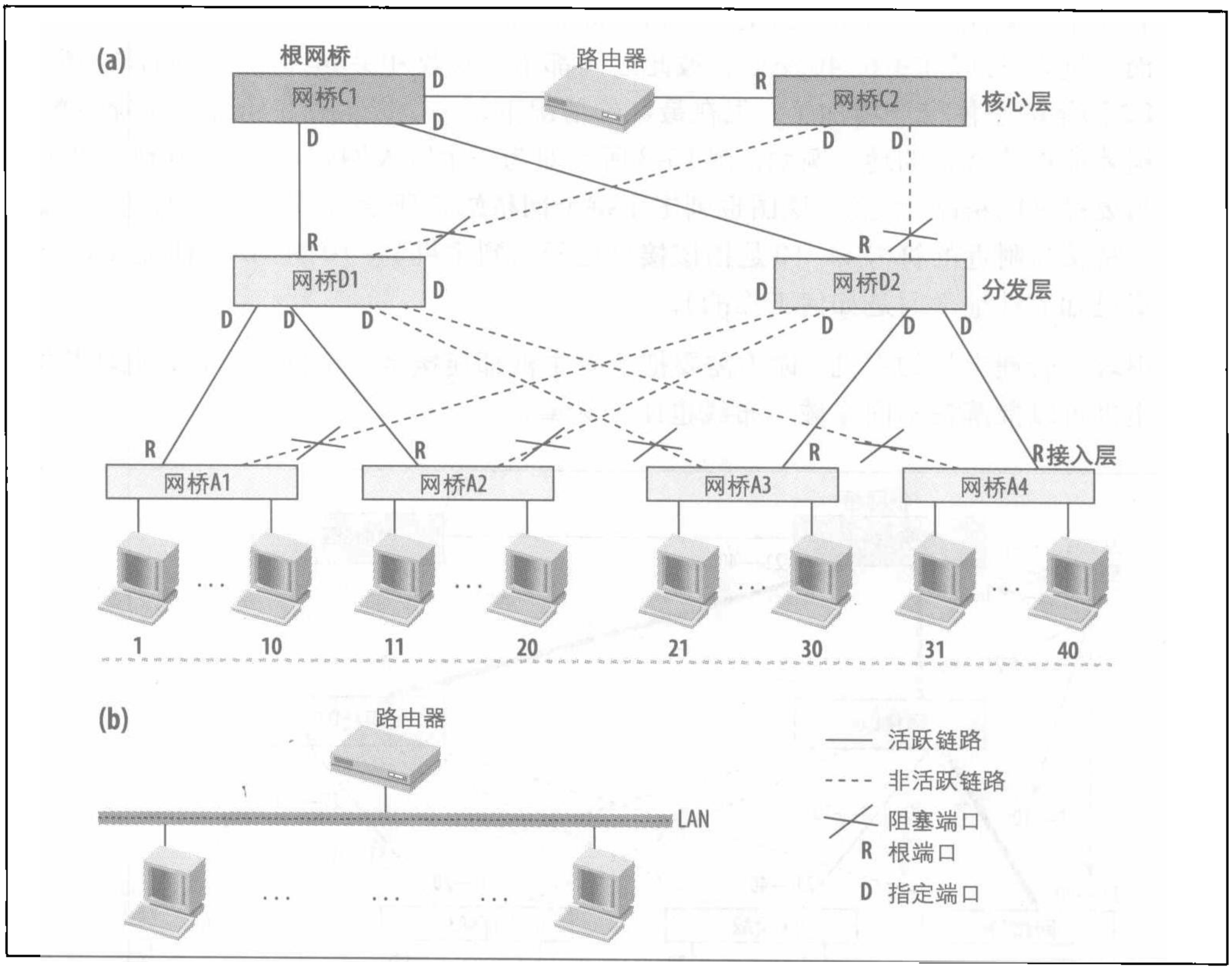


图 15-2：阶层桥接式 L2 拓扑

该图给出了STP选择的、用于定义无环路拓扑的链路，和被分配给指定的角色或根角色的端口。在本章中，我们会说明指定角色和根角色的用处、这些角色是如何分配的以及分配的原因。

注意，图15-2（a）中的L2网络内任何两台主机间交换的流量都是用L2协议传输的（即，Ethernet）。路由选择可以在核心层实施，或者在核心层之上运行。从主机的角度看，该图在链路层没有层次等级，只是一个扁平的LAN；整体的拓扑就如图15-2（b）所示（注1）。

使用多台网桥有以下优点：

注1：　用简单的2位二进制替代12位的MAC地址（例如11:22:33:44:55:66），目的是为了让该图更具可读性。

- 有助于分离流量。例如，当主机 1 和主机 10 交谈时，主机 11 可以和主机 20 交谈，而主机 21 可以和主机 40 交谈，彼此之间都不会接收和丢弃其他以太网帧。所以，L2 网络的整体带宽增加了。但在最坏的情况下，一个以太网帧可能必须穿越整棵树才能抵达其目的地。例如，图 15-3 所示即为一个以太网帧从主机 40 到主机 1 必须要经过的链路。注意，该图也列出了每个网桥端口所学习到的MAC地址：例如，网桥接口附近的符号 1～10 是指该接口已学习到主机 1～10 的MAC地址（第十四章已知道地址学习是如何工作的）。
- 很容易管理大量的主机。你不需要把所有主机都连接至一个巨型网桥，也就是说，主机可以散落在不同区域，布线也比较简单。

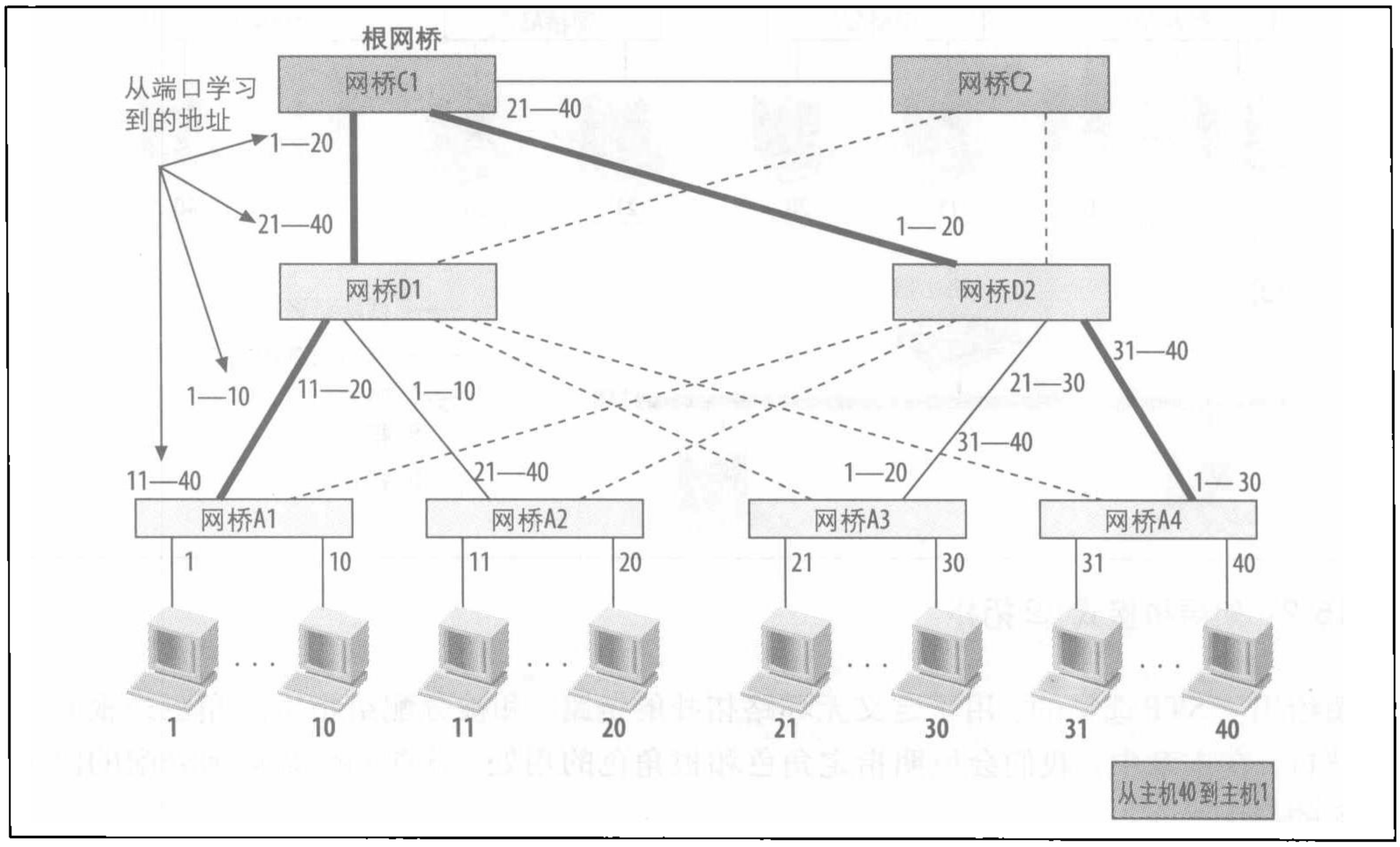


图 15-3：桥接式网络范例

L2桥接式拓扑的概括介绍就到此为止。桥接协议和STP的细节需要一整本书才谈得完，所以，下面我们将简单介绍 STP 使用的算法。

本章接下来会使用较简单的拓扑来说明该协议。但是，我们所看到的较简单的拓扑，其运作方式和同图 15-2 中那个较大又较复杂的网络是相同的。

生成树协议的基本元素

寻找最佳无根生成树（best nonrooted spanning tree）或最佳有根生成树（best rooted spanning tree）的搜寻是运筹学的经典问题。相关文献提供了各式各样的算法，各有不同的计算复杂度。这些算法在现实生活中有很多的应用场合。

本章介绍的STP协议也有类似的目标：已知一个图形以及一个根节点R，要定义出根是R的最佳生成树。但是，一个重要的区别是：该算法不是在单一主机上执行，然后再把结果散布至所有其他主机；恰恰相反，它是一种分布式协议。网络中的所有网桥都必须执行这个协议。执行此协议后，网桥就可以开启它们的某些端口并关闭其他端口，此时所得到的整体拓扑结构就是最佳有根生成树。根节点的选择也是该协议的一部分：主机都同意谁是根节点，然后决定要开启和关闭哪些链路。

下面我们来详细说明什么是“最佳生成树”。有了一个图形以及你想作为根的节点之后，所谓的最佳生成树就是一个无环路拓扑（树），每个节点到根节点的距离为最小。对于某些图形，可能有一棵以上的树可以作为最佳生成树（注2）（图15-4给出的例子就有两种好的解决方法）。

当链路没有指定开销，或者都设置成相同的开销（称为对等）时，则一个节点到根的距离用链路数目（也就是网络跃点）来测量。但是，当你把开销指派给链路后，跃点数目就不一定是路径长短的指标了。

例如，如果我们给图15-4中的拓扑结构加上开销，则与之相配的最佳有根生成树和图15-4所示的两种就不同，如图15-5所示。

我们会在“网桥ID和端口ID”一节讲述网络管理员如何以其他值覆盖默认开销值，例如根据诸如货币成本（monetary cost）以及连接的可靠性等管理性参数来选择开销。

STP通过让各个网桥之间交换称为网桥协议数据单元(bridge protocol data unit,BPDU)的特殊帧来达到其目标。用BPDU交换的信息可让网桥：

- 为每个网桥端口指定一个明确的状态，例如转发或阻塞，以定义该端口可否接收数据流量。
- 通过对端口状态的设定，从环路拓扑中选择或丢弃适当的链路，以这种方式最终生成无环路拓扑。

注2：以后就会知道，STP定义了一种机制，如果有一种以上的最佳化结果时，每次绝对都可以选出的是同一棵树。

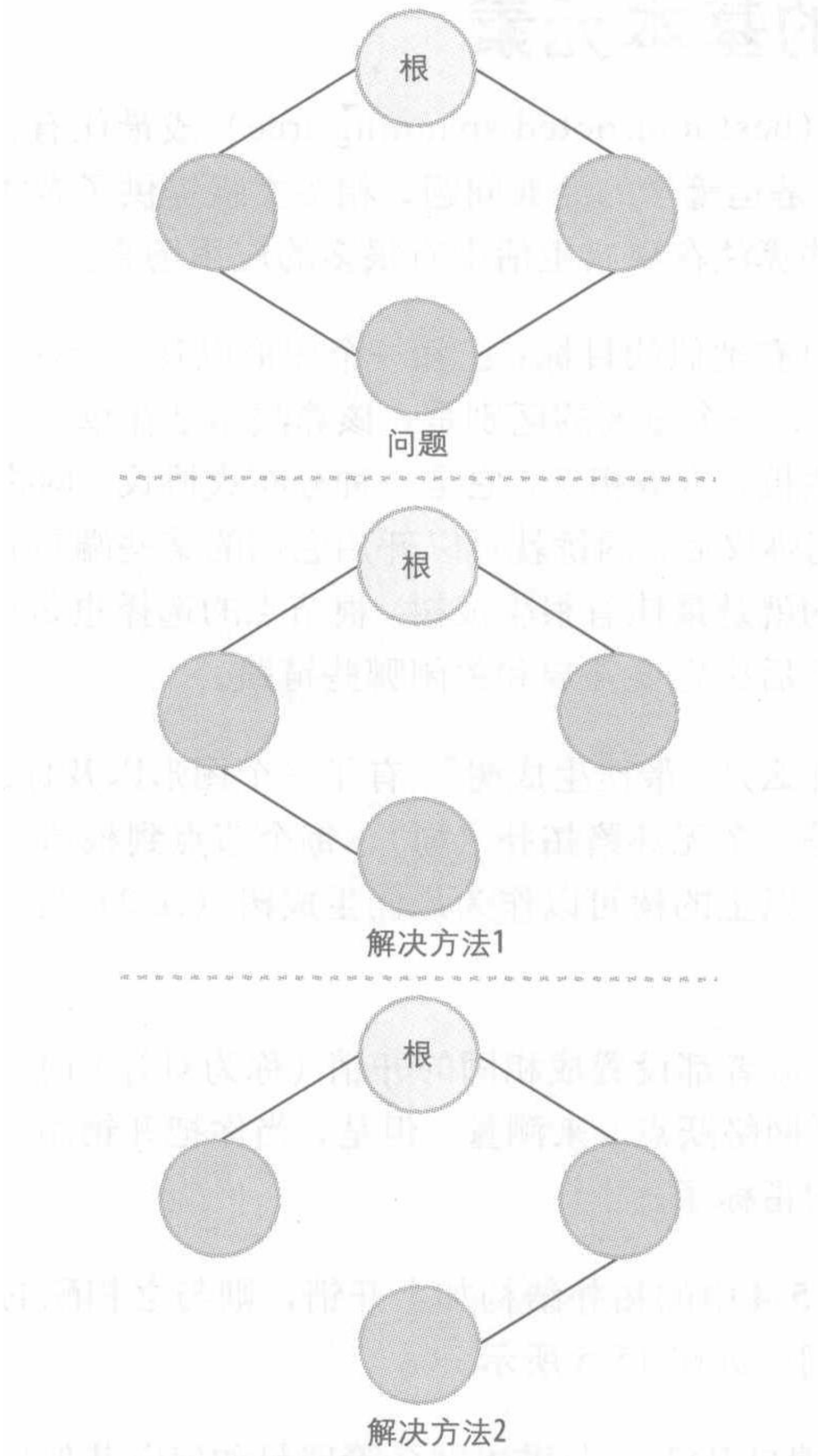


图 15-4：没有给链路指定开销的拓扑图

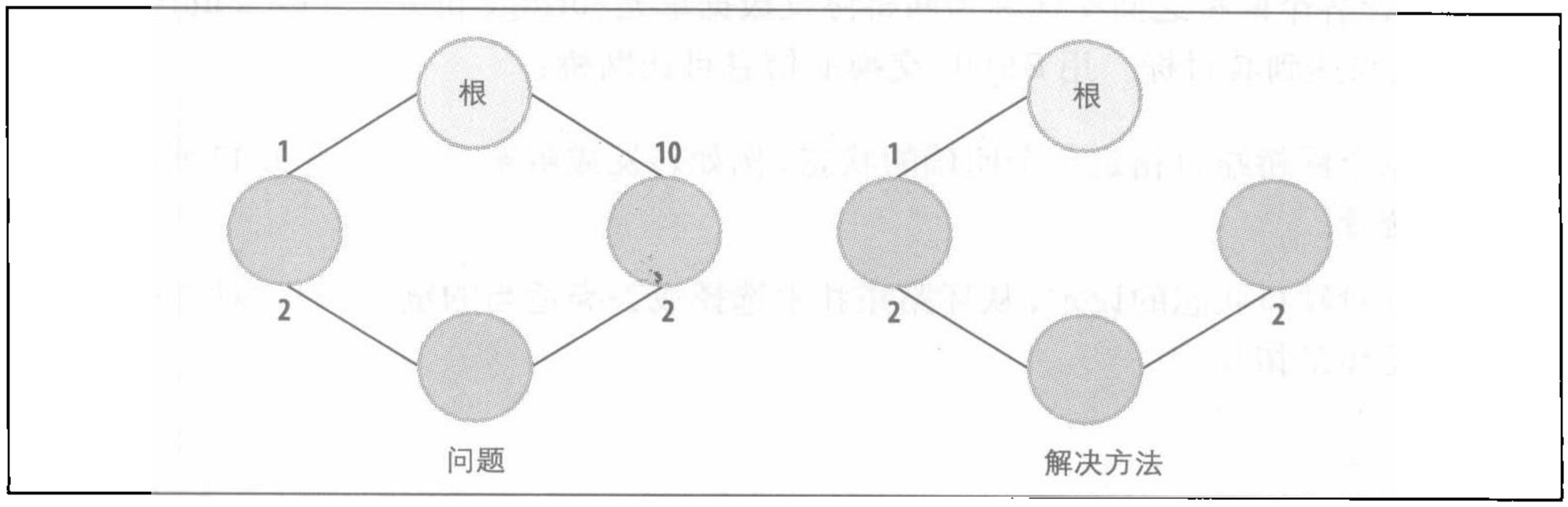


图 15-5：给链路指定开销的拓扑图

有些网桥端口会被赋予特殊角色，例如，依据其是否可到达该树的根（所谓的根网桥）或者该树的叶子节点来赋予不同的角色。

既然图论中有精确的术语，STP使用定义明确的术语以区分节点（网桥）和链路（网桥端口）也就不足为怪了。在研究该算法前，我有必要说明一下几点：

- 什么是根网桥，以及什么是指定网桥
- 网桥的端口可以被赋予哪些状态和角色
- 根端口和指定端口的任务

根网桥

根网桥不仅仅只是在拓扑中占据一个位置而已，它在整个算法中扮演关键角色。例如，在下几节中你会知道：

- 根网桥是唯一能产生BPDU的网桥。其他网桥只有在其接收到BPUD时才会传输BPUD（也就是它们修改所接收到的信息，更新其中一些字段）。
- 当拓扑发生改变时，根网桥确保网络中的每台网桥都知道此事（参见“拓扑变化”一节）

注意，端口状态和角色的选择（哪些链路应该开启或关闭）取决于根网桥在拓扑中的位置：这是因为当你先选出根网桥后，才能据此建立最佳生成树。

指定网桥

虽然每棵树只能有一个根网桥，但是，每个LAN都可以有一个指定网桥，它成为该LAN中所有主机和网桥到达根节点的网桥。指定网桥的选择是根据LAN上哪台网桥通往根网桥的路径开销最低而定。

以图15-2为例（注3）：

- 在A3-D2 LAN中，D2是指定网桥。
- 在D2-C2 LAN中，C2是指定网桥。

注3：图中的链路并未指定开销。你可以假设它们的开销为1，因此路径开销就是跃点数目。这只是示例；1不是指派给链路的默认开销。

生成树端口

我们在前几节提到了根网桥和指定网桥的角色。我们这里说明可以指派给网桥端口的状态和角色有哪些。

端口状态

STP 端口就是运行 STP 协议的网桥中的端口。这个端口会根据“何时传输配置 BPDU”一节的规则处理入口 BPDU 并传输 BPDU 帧。

STP 端口可以处于下列几种状态中的任何一种：

关闭（Disabled）

该端口已关闭（通过管理性操作）；它不再接收或传输任何流量。

阻塞（Blocking）

该端口已开启，但 STP 将其阻塞住了。该端口不能转发任何数据流量。

监听（Listening）

该端口已开启，但是不能用于转发任何数据流量。

学习（Learning）

该端口已开启，但是不能用于转发任何数据流量；然而，该网桥的地址学习进程是激活的。

转发（Forwarding）

此端口已开启，地址学习是激活的，并且可以转发数据流量。

使用中间的学习态可让网桥减少地址扩散次数，当转发数据库为空时就需要扩散。

除了处于关闭状态的端口外，无论其他端口状态为何，都会处理入口 BPDU。处于某一状态的端口是否可以接收入口 BPDU 或传输 BPDU，取决于该端口的角色，“端口角色”一节会介绍。

图 15-6 给出了如何改变端口的状态。从图中可以清楚地看到其变化的过程，从阻塞到监听到学习再到最活跃的状态：转发。阻塞态和转发态之间的转换是由协议根据各种因素决定的（参见稍后“定义活跃拓扑”一节）。注意到：

- 一个端口往转发状态转换时，可以在进入转发状态前退回到阻塞状态。例如，当拓扑尚未稳定而其状态会在短时间内不断改变时，就有可能发生这种情况。
- 从阻塞态到转发态过渡的中间状态是由定时器驱动的（参见“定时器”一节），为了必须避免产生临时环路（参见“消除临时环路”一节），这是必要的。

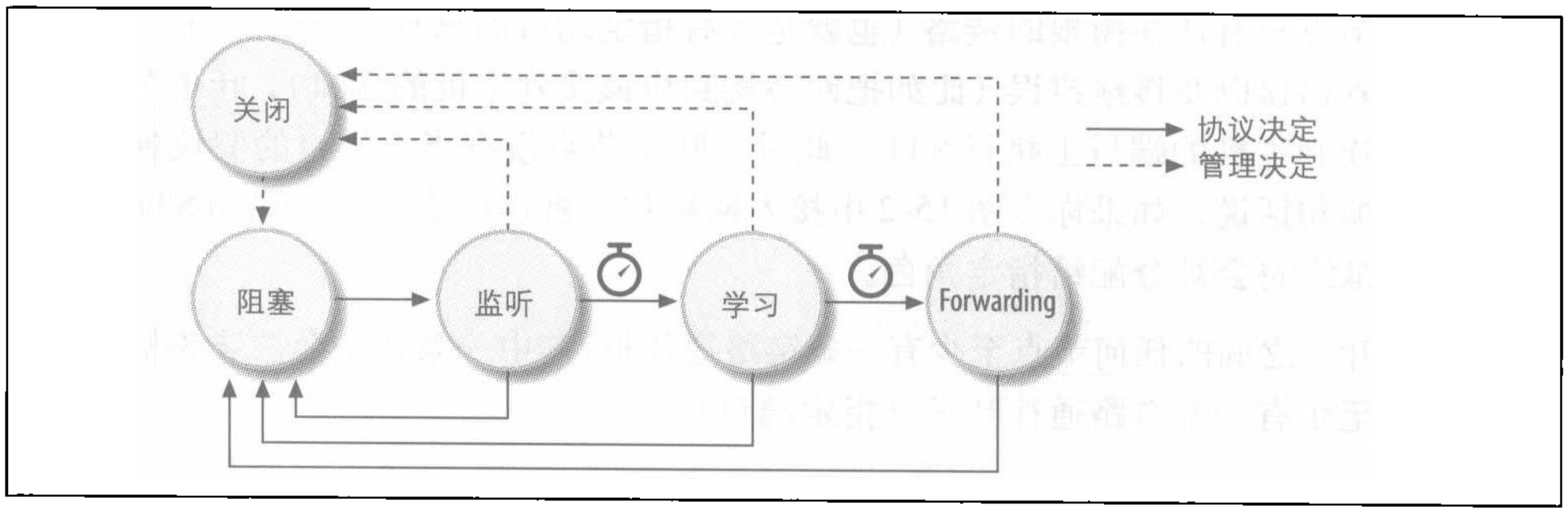


图 15-6：端口状态转换

此外，管理员可以手动地把端口从任何状态中移出并将其关闭。当一个端口经由管理手段关闭时，则只有再度以管理手段干预才能重新启用；STP 没有超越人工管理手段的能力。然而，网桥可以在协议定义的基本功能之外实现一些可选功能，以开启和关闭端口，而无需管理手段干预。

端口角色

STP 端口可以被赋予下列两种角色之一：

根

对每台网桥来说，根网桥除外，通往根网桥的路径开销最低的端口就会被选为根端口。

指定

在每个 LAN 上，通往根网桥的路径开销最小的端口就会被选为指定端口。指定端口所属的网桥就称为该 LAN 的指定网桥。需要注意的是，一台网桥上的端口连接不同的 LAN 时，可以有一个以上的指定端口，如图 15-2 所示。用于选择指定端口的规则会在“选择指定端口”一节说明。

根端口可通往生成树根（也就是根网桥），而指定端口可通往叶子。在图 15-2 中，你可以看见根端口和指定端口间的关系。

从树的观点看，这两种角色可以按以下方式看待：

- 该树的根有只通往叶子的链路（也就是只通往指定端口（注 4））。

注 4：如果你使用共享接口（诸如 Hub）连接网桥，如图 15-11（c）所示，则根网桥也可以有非指定端口。新版的 RSTP 协定称该端口为备用端口（backup port），参见“快速生成树协定（RSTP）”一节。

- 叶子节点只有通往树根的链路（也就是没有指定端口而只有一个根端口）。为了防止配置错误以及接线错误（比如把网桥接到应该接到主机的端口），叶子节点也可以在连到主机的端口上执行 STP。此时，叶子节点没有指定端口的假设就不再成立。换句话说，如果你在图 15-2 中接入网桥和主机连接的端口上启动 STP，这些端口最终将会被分配给指定角色。
- 根和叶子之间的任何节点至少有一条链路通往根（其中一条链路会被选为根端口），而且至少有一条链路通往叶子（指定端口）。

有些STP端口既不是根端口也不是指定端口；当在网桥间有冗余连接时，就有可能发生这种情况。在图 15-2 中，通往 D2 的 A1 端口就是一例。新版的 STP 协议定义了新角色，我在“新版生成树协议概况”一节会简要介绍，使得每个STP端口都会被赋予一种角色。

“选择根端口”和“选择指定端口”两节会分别说明根端口和指定端口角色是怎么分配的。

网桥 ID 和端口 ID

根网桥和端口状态及角色的选择取决于一组参数。每个参数都会有个默认值，而且可以由用户改变设置。以下是主要的参数：

网桥 ID（Bridge ID）

每个网桥都会分配一个 ID，称为网桥 ID，此 ID 是一个 8 字节的数，分成两部分。最低的六个字节是某个桥接端口的 Ethernet MAC 地址（参见第十六章），而最高的两个字节是一个可以配置的优先级，称为网桥优先级（bridge priority）。在根网桥选择算法中要用到网桥 ID（参见“选择根网桥”一节）

端口 ID（Port ID）

每个端口都会被指定一个 ID。这个 ID 的一部分代表一个唯一的识别码，称为端口号。设置端口号的方式和具体的协议实现有关，且其值只在网桥本地有意义。例如，端口号可以反映出端口建立的顺序：第一个端口指定为 1，第二个指定为 2，等等。另一种方法是使用端口的实际位置：例如，总线上的第一个端口指定为 1，等等。设备即使重启后，设定的端口号也应该具有确定性和一致性，这样系统管理员就不必在重起网桥后修改网桥的配置来反映出端口号的变化。

端口 ID 的另一部分称为端口优先级（port priority），用于给端口指定优先级（值较低表示较有高优先级）。如图 15-7（b）所示。

参考“选择根端口”一节有关此参数的使用范例。

除了网桥和端口优先级外，用户还可以配置下列参数：

端口开销

每个端口都被设定一个开销。开销值越低，表示该端口越应当优先使用。如果没有进行明确的设置，设备会根据该端口的速率而设置一个默认开销。例如，一个Fast Ethernet端口的速率是100 Mbit/s，那么其开销值会比速率为10 Mbit/s的Ethernet端口的低。如果生成树中点和点之间的开销测量方式是用延迟（latency）来计算，那么默认开销的设置在很多情况下是有意义的。在特殊情况下，管理员宁愿根据其他因素明确地设置一个开销值。

定时器

STP为每个端口和每个网桥都设有定时器。这些定时器都有默认设置，并且用户可以对其调整，参见“定时器”一节。定时器配置不会影响根网桥和端口状态及角色的选择。

在本章后面会讲解这些参数的设置（定时器除外）如何影响拓扑结构的选择。

2001年时，IEEE颁布了802.1t规范——即802.1D的维护文档——改变了网桥ID和端口ID的定义方式。其格式变化如图15-7所示。

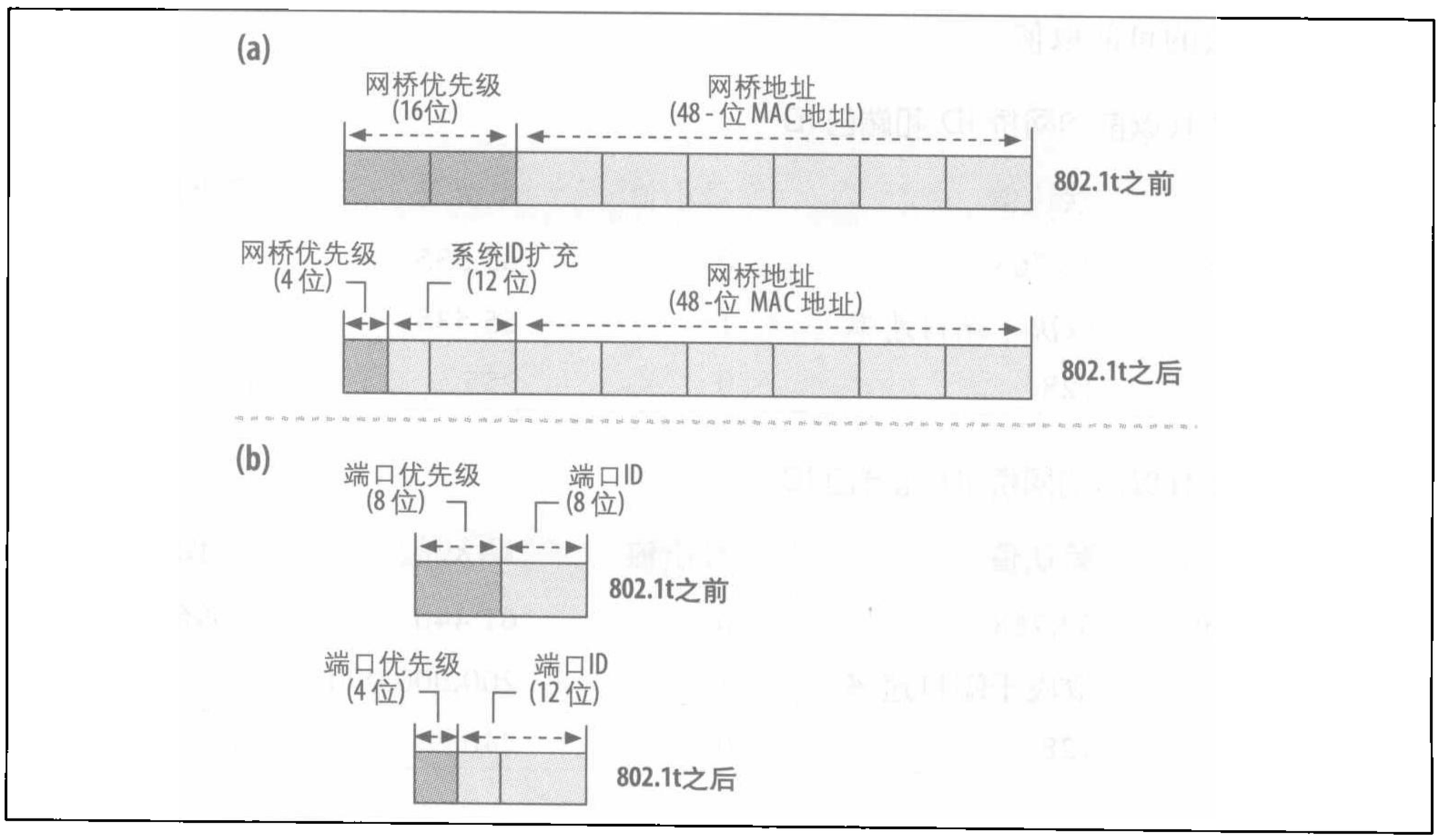


图15-7：802.1t引入的网桥ID和端口ID的变化

注意：

- 网桥优先级现在只用4位二进制表示。为了和旧版本兼容，网桥优先级的范围依然是0～64 K，但是因为只有4位可以设置，所以优先级的增量就变成了4,096（2^{12}）。

- 网桥ID中多了一个新元素，称为系统ID扩展（system ID extension）。此元素可以设4,096个不同值，可以让4,096个不同的网桥ID共享同单一MAC地址。之前，则需要有4,096个不同的MAC地址。注意，每个MAC地址不是由管理员随机乱选的数字，而是由IEEE管理的且是全世界独一无二的数字（因此是有限资源）。
- 端口号现在是一个12位二进制数，可以让网桥有多达4,096个端口。以前只能有256个（当初觉得这已经很奢侈了）。端口优先级现在是4位二进制数。为了和旧版本兼容，优先级范围依然是1～256，因此，优先级现在的增量是16。

要理解802.1t变化的原因，需要从高端商业设备的角度考虑，而非从只配备几块NIC的普通PC考虑。普通PC的限制是256个网桥端口或每个MAC地址只能有一个网桥ID。然而，对大型网络而言，配置上百个端口并且运行着好几百个网桥的情况是很常见的

注意：4,096并不是一个随机数：该值代表802.1Q协议中所允许的VLAN的最大数目。

802.1t的变化对STP没有任何影响。从STP的角度看，网桥ID是一个8字节二进制数，而端口ID是一个2字节二进制数。用户所设置参数的字节数和设置的目的根本不重要。也就是说，802.1t的变动只会影响配置工具，而不影响协议的行为。表15-1和表15-2总结了每个参数的可能取值。

表15-1：802.1t以前的网桥ID和端口ID

	默认值	最小值	最大值	最小增量
Bridge priority	32,768	0	65,535	1
Port cost	取决于端口速率	1	65,535	1
Port priority	128	0	255	1

表15-2：802.1t以后的网桥ID和端口ID

	默认值	最小值	最大值	最小增量
Bridge priority	32,768	0	61,440	4,096
Port cost	取决于端口速率	1	200,000,000	1
Port priority	128	0	240	16

网桥协议数据单元（BPDU）

网桥进行协议交换的帧称为BPDU，其内部包含许多信息，足以让这些网桥彼此同意谁才是根网桥，以及决定其本地端口的角色和状态。BPDU有两种：

配置 BPDU

用于定义无环路拓扑。到了“何时传输配置BPDU”一节就会知道，哪些情况会触发这类BPDU的传输。

TCN（Topology Change Notification，拓扑变化通知）BPDU

当检测到拓扑变化时，网桥用于通知根网桥。参见“拓扑变化”一节。

图15-8所示是这两种BPDU的格式。注意，这两种类型共享前三个字段，可以通过BPDU类型参数进行区别。

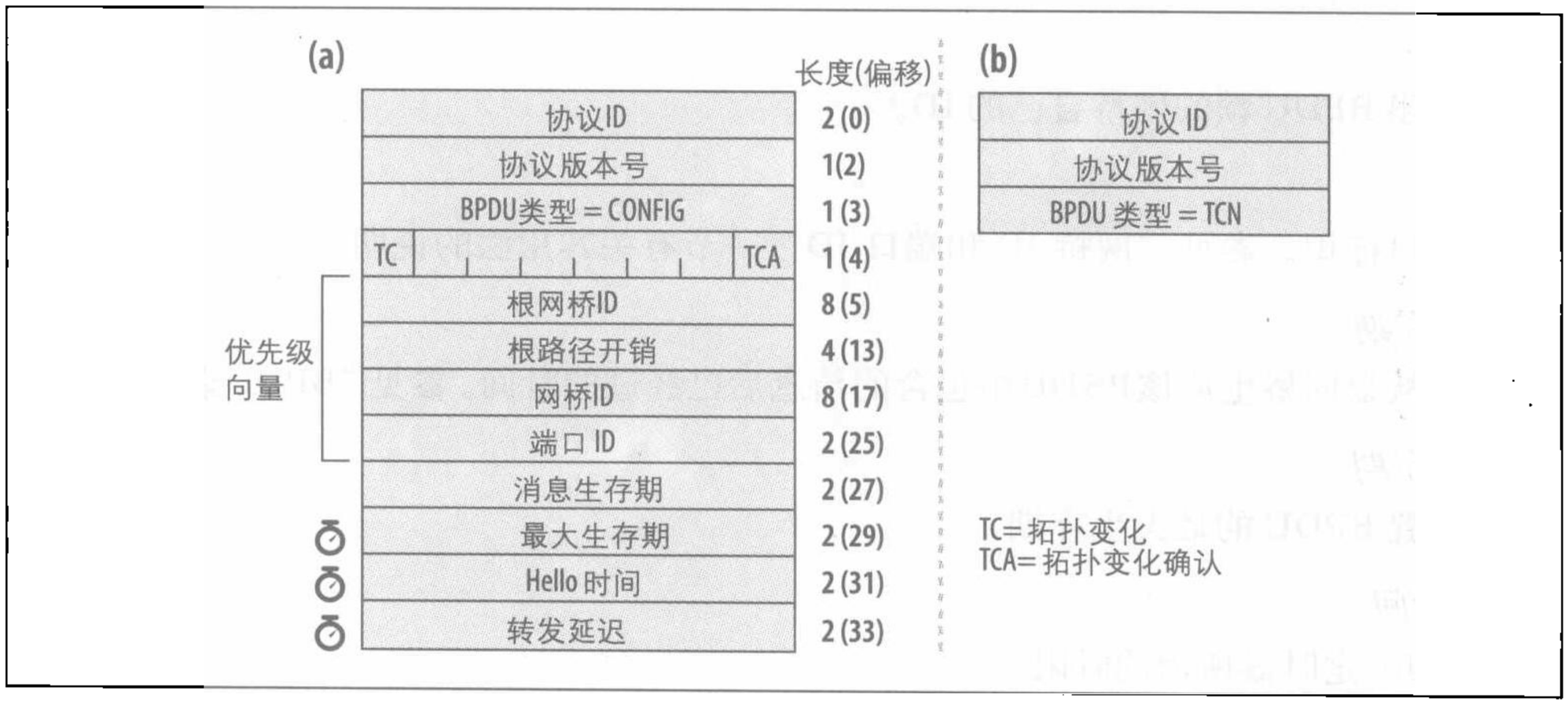


图 15-8：（a）配置 BPDU；（b）BPDU

表15-3列出了三种IEEE STP标准所用的协议ID和协议版本号。本章只谈基本的802.1D协议，至于另外两种则会在“新版生成树协议概况”一节介绍。

表 15-3：BPDU 版本

协议名称	协议 ID	协议版本号
STP (802.1D-1998)	0	0
RSTP (802.1D-2002 或 802.1w)	0	2
MSTP (802.1Q-2002 或 802.1s)	0	3

配置 BPDU

下面是配置 BPDU 中的各个字段的含义：

标识

只使用两个标识：TC（Topology Change，拓扑变化）和TCA（Topology Change Acknowledgment，拓扑变化确认）。这两个标识的使用会在“拓扑变化”一节说明。

根网桥ID

根网桥的ID。也就是发出BPDU帧的网桥所认定的当前根网桥。

根路径开销

从发出BPDU帧的网桥到根网桥的最短路径开销。当进行传输的网桥是（或者认为其即将成为）根网桥时，其值为0。

网桥ID

发出BPDU帧的网桥自己的ID。

端口ID

端口标识。参见“网桥ID和端口ID”一节有关其用法的说明。

消息生存期

自从根网桥生成该PBDU中包含的信息后已经过的时间。参见“BPDU老化”一节。

最大生存期

配置BPDU的最大生存期。

Hello时间

Hello定时器所用的时限。

转发延迟

Forward Delay定时器所用的时限。见图15-6。

Max Age、Hello以及Forward Delay这三个定时器的值都不是在网桥本地配置的，而是由根网桥公布（参见“传输配置BPDU”一节）。所有这些值都用滴答（tick）表示（1/256s）。参见“定时器”一节。

优先级向量

配置BPDU的四个元素——根网桥ID、根路径开销、网桥ID以及端口ID——构成了优先级向量（Priority Vector），如图15-8所示。因为这四个元素是按顺序排列的，所以这个向量可以视为一个独立的22字节的数。此数愈小，网桥在拓扑中的重要性愈高；换句话说，优先级向量会决定谁赢得根网桥和指定网桥这类竞争性角色的叫牌权。本章接下来将以[*BR-Root*, Cost, *BR-ID*, *Port-ID*]这个符号表示优先级向量。

在本章稍后的示例中的图形只给出已经完成传输的配置BPDU的优先级元素，因为优先级元素是网桥用于选择其端口的角色和状态时所用的配置 BPDU 的一部分。

已知有两个优先级向量 PV1 和 PV2，PV1=[*BR-Root1*, *Cost1*, *BR-ID-1*, *Port-ID1*]，而 PV2=[*BR-Root2*, *Cost2*, *BR-ID-2*, *Port-ID2*]。当 PV1 的值小于 PV2 的值时，就说 PV1 的优先级较高；当 PV1 的值大于 PV2 的值时，就说 PV1 的优先级较低。也就是说，如果 *BR-Root1* < *BR-Root2*，则 PV1 的优先级高于 PV2。这两个值相等时，如果 *Cost1* < *Cost2*，则PV1 的优先级高于PV2；如果Cost1 和Cost2 也相同时，若 *BR-ID1* < *BR-ID2*，则 PV1 的优先级高于 PV2；最后当两个网桥 ID 也相吻合时，如果 *Port-ID1*<*Port-ID2*，那么 PV1 的优先级就高于 PV2。

何时传输配置 BPDU

在下列情况下，一台网桥会从其指定端口传输出配置 BPDU：

- 根网桥会运行一个定时器（Hello定时器），该定时器到期后就触发配置BPDU的传输。同一个 BPDU 从根网桥的每一个指定端口传输出去。只有根网桥能产生新的 BPDU，但是当一个网桥加电启动时，它会以为自己就是根网桥（因为没有其他优先级向量可以比较）。所以，该网桥会将其所有端口都设为指定角色，启动Hello定时器，然后开始产生 BPDU（参见“选择根网桥”一节）。
- 非根网桥只有在应答其根端口收到的BPDU时才会产生BPDU；也就是说，非根网桥会传递BPDU。由非根网桥传输的BPDU，其所携带之信息和它们所接收的BPDU除下列字段之外都是相同的，非根网桥会更新这些字段（参见图 15-9）：
  - 传输者的网桥 ID 和端口 ID 会被该网桥自己的信息替换。
  - 该网桥会把开销更新为其所接收的开销加上其本地网桥中接收 BPDU 的端口（根端口）的开销之和。
  - 消息生存期会按照“BPDU 老化”一节所讲述的逻辑进行更新。那一节会说明如何定义 DT 量（DT quantity）。

无论网桥是否为根网桥，在下列情况下也会传输一个配置 BPDU：

- 当一台网桥接收到一个BPDU，如果其优先级向量低于该网桥在其端口上所用的，那么，网桥就会以其自己（较高级）的信息应答。参见“STP 应用实例”一节。
- 当一台网桥接收到一个 TCN BPDU 时，会立刻发出一个设有特殊标识的配置 PBDU 作为应答。这样有利于快速传播拓扑中的变化（参见“让所有网桥知道拓扑变化”一节）。

不管是因何种原因从某一给定的端口发送出一个配置BPDU，STP都会对其速率进行限制：网桥从其任何端口中每秒只能最多传输一个配置 BPDU（参见“传输配置 BPDU”一节）。

BPDU 老化

因为BPDU只由根网桥生成，而且其他网桥只有在其根端口接收到一个BPDU时才会重新产生一个，所以，根网桥从其生成BPDU所用的信息到BPDU抵达叶子网桥的时间是实时变化的。在一个稳定的网络环境中，这个时间主要取决于网桥的负载情况以及网桥处理 BPDU 有多快。

携带过期信息的BPDU不应该用来建立无环路拓扑。因此，配置BPDU有个称为消息生存期的字段，接收到该 BPDU 的网桥会拿该 BPDU 中的另一个字段 —— 最大生存期 —— 和其比较。如果该BPDU的生存时间太长且无法相信其优先级向量的话，网桥就会把这些 BPDU 丢弃掉。

消息生存期字段最初会由根网桥初始化为0，而每个非根网桥在对其转发前都会更新该字段（参见图15-9所示的DT）。假定消息生存期表示自原根网桥产生该BPDU后所经过的时间。然而，要计算这段时间并不容易。例如，是否应该把传输延迟和处理时间也考虑进来：也就是说，帧从一台网桥的端口走到另一台的端口在传输介质中所花的时间，以及每个网桥处理该帧并对重新生成一个BPDU时在网桥内存中花的时间。但是，商业网桥的常见做法是只把消息生存期字段视为跃点计数，如同IP包头中的TTL（Time To Live，存留时间）字段：入口 BPDU 的消息生存期字段会增加256个滴答（也就是1 s），然后复制到输出的 BPDU 中。这意味着，最多经过 20 个跃点（20 s 是最大生存期的默认值），该 BPDU 就会被丢弃。Linux 没有把消息生存期视为跃点计数，而是尽量遵循“传输配置 BPDU”一节所描述的原有规则。

当网桥在其端口上所接收到的BPDU尚未过期时（也就是消息生存期小于最大生存期），网桥会启动Message Age定时器，该定时器会在由最大生存期和消息生存期的时间差确定的一段时间后到期。参见“定时器”一节，可以了解Message Age定时器到期后所触发的行为。这样可以确保 BPDU 在其产生之后经过最大生存期（秒）数后就会被丢弃，除非到时候得到确认。

定义活跃拓扑

每台网桥按照本地配置信息和接收到的入口配置 BPDU 中的信息，可完成下列工作：

- 选出根网桥。

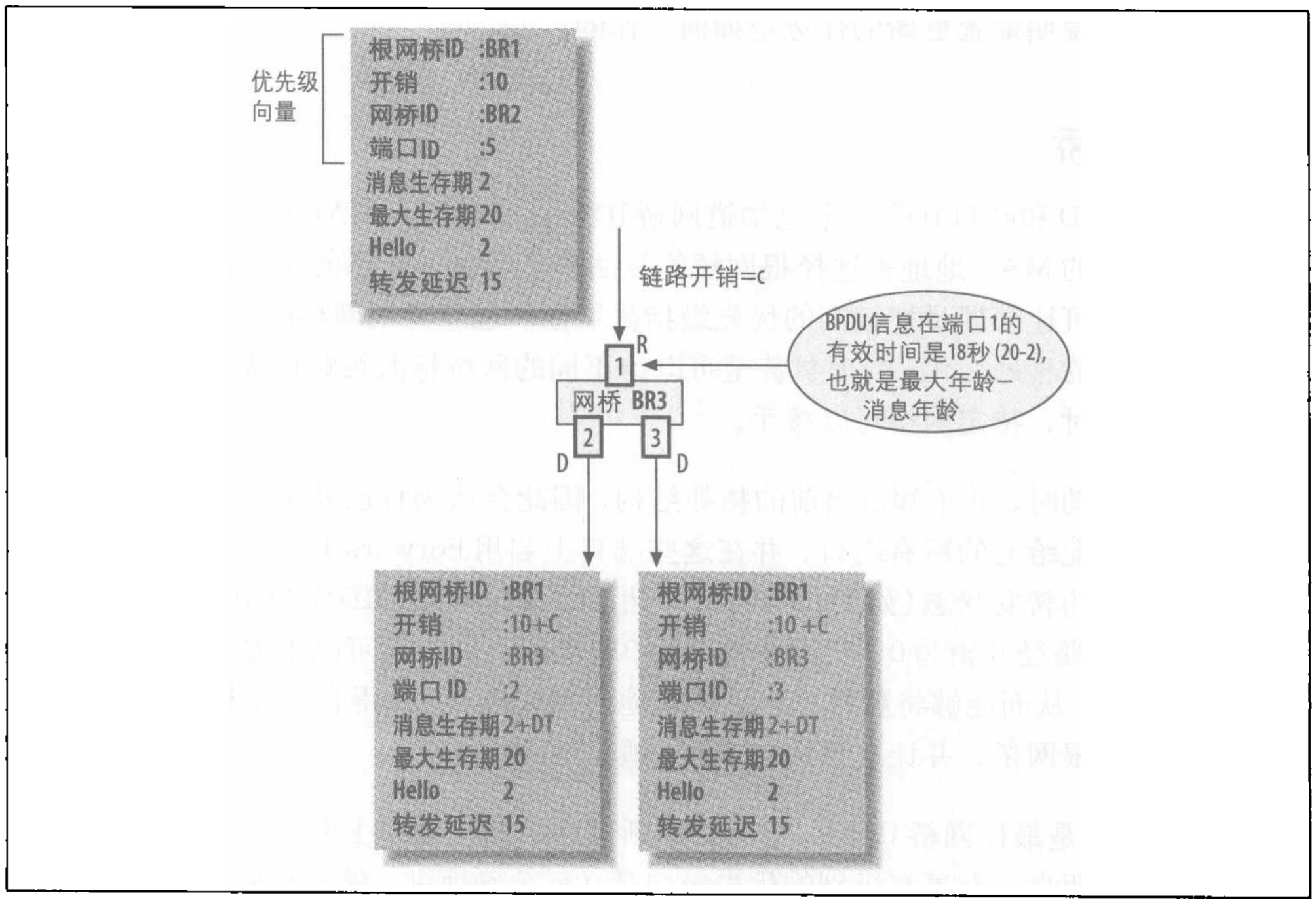


图 15-9：BPDU 通过非根网桥进行传递

- 从其几个端口中选出一个作为根端口。
- 对每个端口，为该端口所属的 LAN 分配指定网桥和指定端口。

我将上面那些任务称为配置更新（configuration update），每当网络有变动而使得拓扑需变化时，就需执行这些更新。例如：

- 开启或关闭一个端口。
- 如果一个端口的 Message Age 定时器到期了。此端口会重新启动（重新分配指定角色）。
- 一台网桥的本地配置信息发生变化。
- 一个网桥端口接收的一个配置BPDU，其优先级向量比该端口先前所接收的BPDU的优先级向量高级。

注意，触发配置信息更新的是配置有变化的网桥，或者出于管理目的改变了某个端口状态的网桥。其他网桥（如果有必要）接下来就会看见这些改变，通过它所接收的BPDU中携带的信息可以反映出来。

下面我们逐一说明配置更新的任务是如何工作的。

选择根网桥

我们在“网桥ID和端口ID”一节已知道网桥ID的定义。因为MAC地址是全球唯一的，所以根据所用的MAC地址来选择根网桥的算法可足够保证选择的确定性。然而，优先级元素的使用可让管理员把较高的优先级指派给他们想选为根网桥的网桥，来迫使网络拓扑按照他们的喜好变化。管理员甚至可以给不同的网桥指派策略性优先级，使得在当前根网桥失效时，特定网桥可以接手。

当网桥首次启动时，并不知道当前的拓扑结构，因此会认为自己就是根网桥。该网桥会把指定角色分配给它的所有端口，并在这些端口上启用Forward Delay定时器，因此它们最终都会设为转发状态(如图15-6所示)，然后以自己的网桥ID作为BPDU中根Bridge ID字段并且根路径开销为0，从而开始传输BPDU。这种方式可以很方便地将该BPDU变成广播数据，从而能够将数据尽可能快速地向四处散布，使得自己和其他网桥都可发现真正最佳的根网桥，并让这棵树能重新平衡。

当该网桥ID就是最佳网桥ID时，它就会不断地在其指定端口上发送BPDU，因为其他网桥都无法声明自己有更高级别的优先级向量（更精确地讲，就是更好的网桥ID），因此，该网桥就会担当起根角色的职务。

如果该网桥ID不是最佳网桥ID，最后它就会收到一个配置BPDU，里面包含一个比较高级的根网桥ID（也就是更高优先级的BPDU），然后：

- 接受并记录这些级别较高的信息（包括根网桥ID和定时器）。
- 相应地更新其端口的状态和角色。这就是所谓的配置更新。

选择根端口

按照“端口角色”一节所述，每台网桥都必须选择自己的根端口，根端口就是具有通往根网桥最短路径（或最低开销）的端口。根网桥是唯一没有根端口的网桥；非根网桥则有唯一的一个根端口。

对每个端口来说，通过管理手段关闭的端口除外，网桥都会保留一份接收自入口BPDU的最佳优先级向量的复本。这样一来，网桥就知道每个端口到达根网桥的最佳路径（最低开销）是多少。

根端口的选择就是遍历所有端口，然后选出拥有最佳优先级向量的那个端口。如果有一

个以上的端口恰好有相同的最佳优先级向量，则选择拥有最低端口ID的本地端口作为根端口，如图 15-10 所示（注意，接收端口 ID 不是 BPDU 的一部分）。

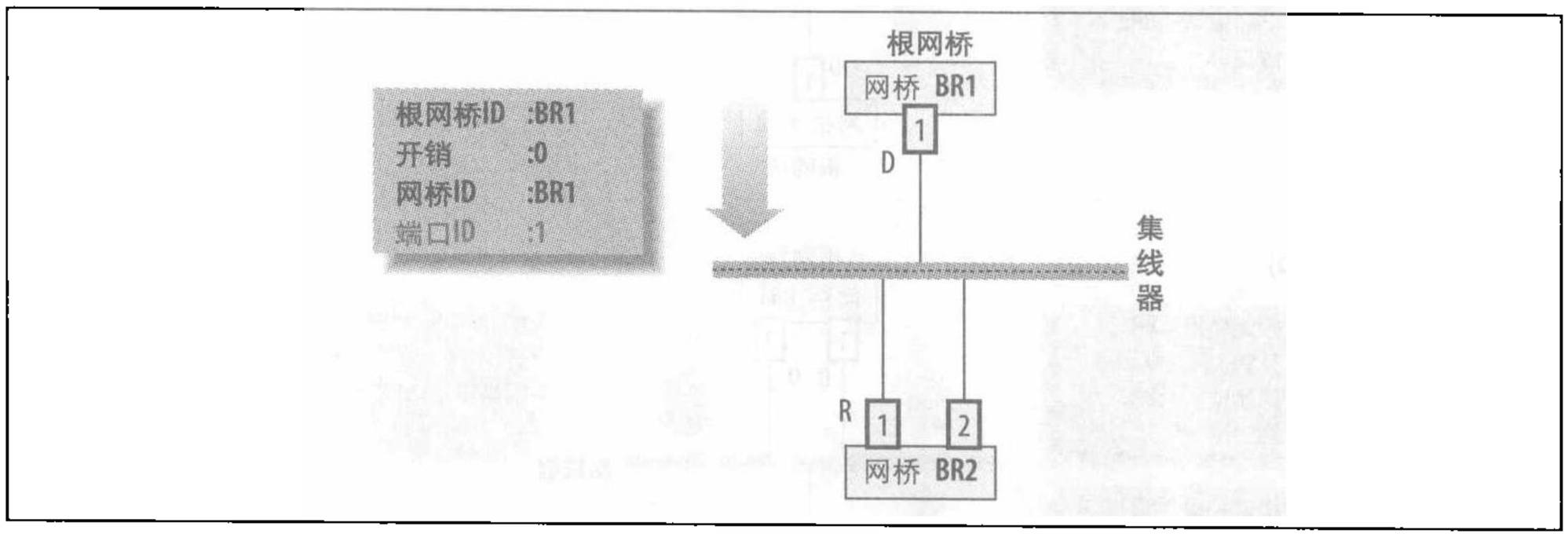


图 15-10：根端口的多个候选对象

选择指定端口

尽管每台网桥都有一个根端口，每个LAN也只有一个指定端口。STP要确保每台网桥选的都是同样的指定端口。指定端口应该是具有通往根网桥最低路径开销的那个端口。因此，也就是拥有最佳优先级向量的端口。

每台网桥通常都连接一个以上的 LAN，所以，它必须知道每个 LAN 的指定端口。

若两台网桥之间是点对点的连接，那么只有两个端口。那个送出拥有最佳优先级向量的BPDU的端口会被选中作为指定端口。相应地，在具有 Ethernet Hub 这种共享介质的网络中可能有两台以上的网桥。这种情况下，每台网桥都会接收到其他网桥的BPDU，然后通过检查优先级向量，选出正确的指定端口。

图15-11所示就是当你使用共享介质连接网桥时的情况。一开始，只有BR2连接至Hub，因此，BR2 会被选为根网桥。当管理员稍后添加BR1时，BR1 也认为它是根网桥，图 15-11（b）中可见它传输了BPDU。让我们假设BR2的ID高于（大于）BR1的ID，因此，这两台网桥最后都会同意BR1是根网桥。

由于这些网桥端口全都连接在同一Hub上，当BR1的端口1传输一个BPDU时，BR1会在其端口 2 上接收自己的BPDU，反之亦然。不过，在这种情况下，基于最佳优先级向量选择指定端口的算法仍然适用：优先级向量的第四个字段是端口 ID，使得端口 1 的BPDU 优先级向量有最高的级别。

但是，由于一些原因，这种包含共享介质的网络很少见，所以本章后续部分只说明点对点的情况。

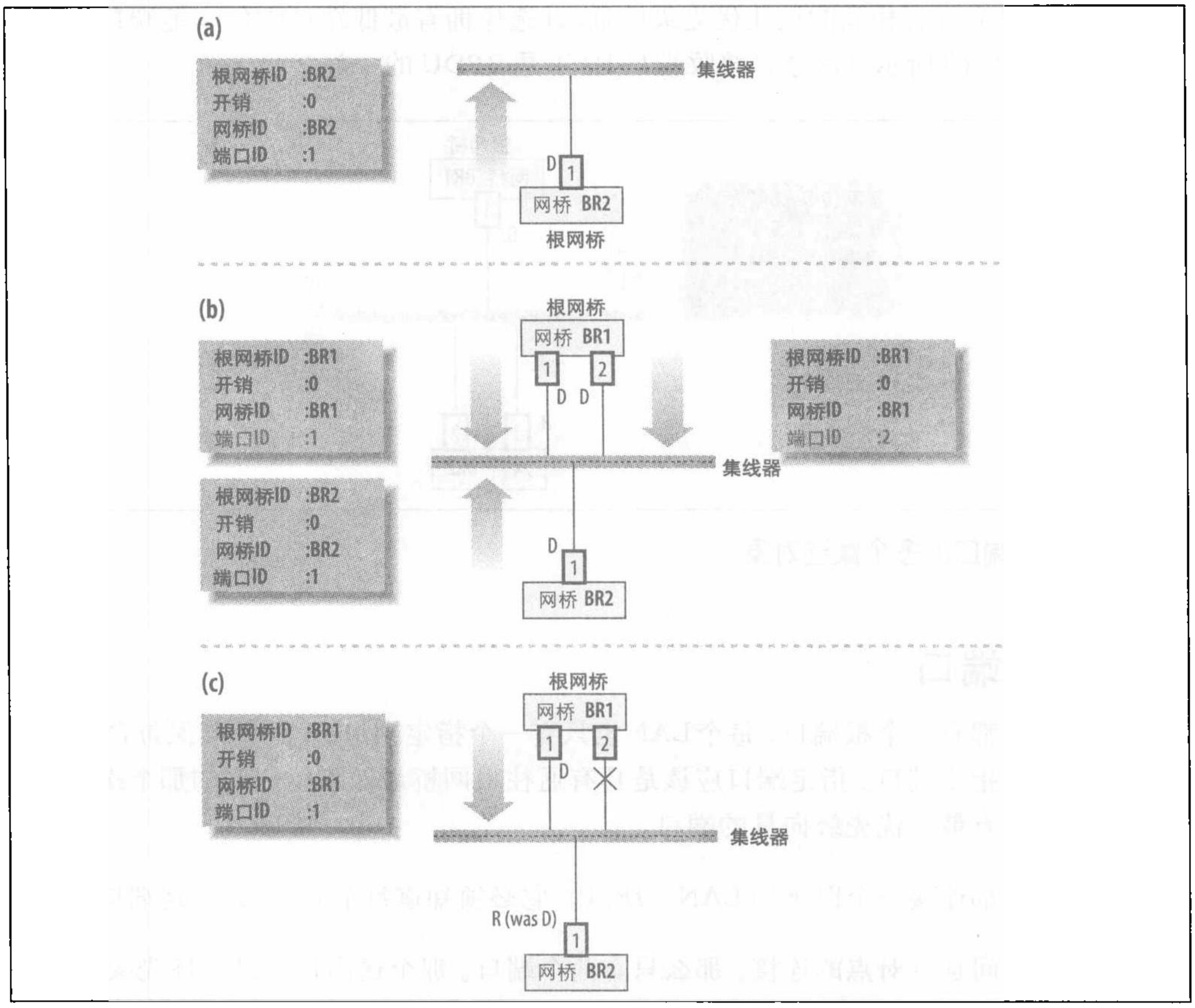


图 15-11：选择指定端口

STP 应用实例

假设我们有如图15-12所示的拓扑结构。注意，图中因为没有冗余链路，所以不需要STP。假设：

- *Bridge ID BR1 < Bridge ID BR2 < Bridge ID BR4*（因此 BR1 是根网桥）。
- 每台网桥可以自行配置其本地接口的开销，而和其他网桥无关。为了简化并让图易于阅读，我们假定所有路径开销都是匀称的（注 5）（每条链路的两端的开销都相同）。

注 5： 要记住，任何链路的路径开销都是网桥本地配置信息的参数，不是由配置BPDU携带的。

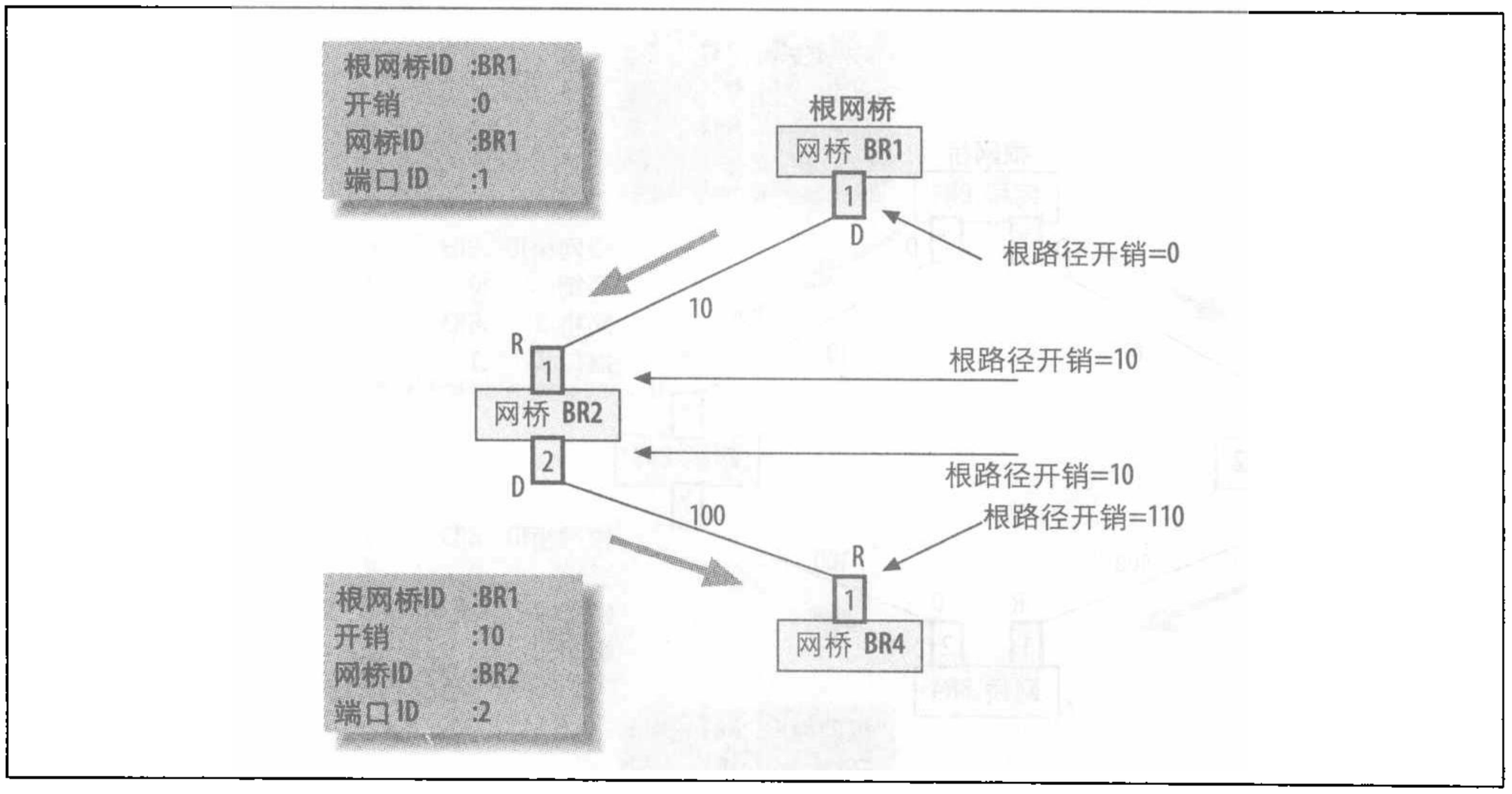


图 15-12：更新根路径开销

注意：

- 每次Hello定时器到期时，BR1的指定端口就会定期发出一个配置BPDU。因为BR2每隔一段 Hello 时间就会在其端口 1 上收到接收配置 BPDU，因此，差不多也是每隔一段 Hello 时间就会在其端口 2 上重新产生（转发）一个配置 BPDU。
- 公布网桥 1 的配置 BPDU：
  - BR1 为根网桥
  - 根路径开销为 0
  - 其自身的网桥 ID 为 BR1
  - 端口 ID 为 1
- 公布网桥 2 的配置 BPDU：
  - BR1 为根网桥
  - 根路径开销为 10（自己的开销加上 BR1 所传出的开销）
  - 其自身的网桥 ID 为 BR2
  - 端口 ID 为 2

现在，我们加入一台名为 BR3 的新网桥，然后假定 *Bridge ID BR3 < Bridge ID BR4*，如图 15-13 所示。

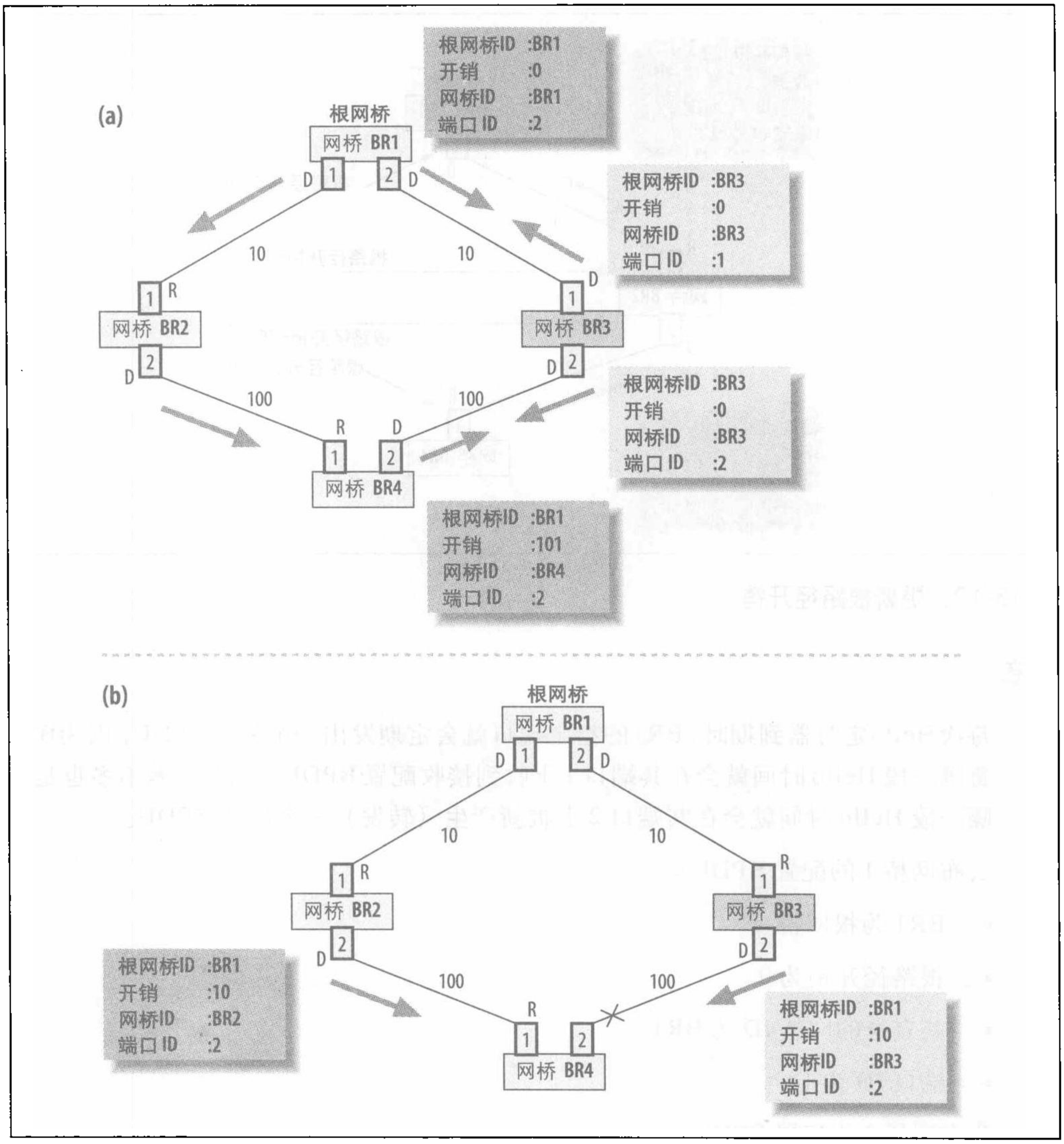


图 15-13：在一个稳定拓扑中添加一个新网桥

在“选择根网桥”一节曾讲过，当BR4首次启动时，会以为自己就是根网桥，因此会把指定角色指派给它的两个端口。BR4会在每个端口上发出一个配置BPDU，公布其为根网桥，并且在BPDU中使用为0的根路径开销。

如果我们假设BR3是以点对点方式连接到BR1和BR4（如图15-13所示），则当BR3加电时，BR1和BR4就会启动其和BR3连接的端口，分配这些端口为指定角色，然后开始发出配置BPDU。

我们来看BR1、BR3以及BR4接收到其他网桥的配置BPDU时如何响应：

- BR1和BR4的配置BPDU中各自的优先级向量是：*[BR1, 0, BR1, 2]*和*[BR1, 110, BR4, 2]*。
- 由于BR1从BR3接收到的BPDU的优先级向量级别较低，BR1会让其端口2保持指定角色不变，并且维持其根网桥的地位。另一方面，当BR3收到来自BR1的BPDU时，会了解到BR1有较高级的网桥ID（以及较高级的优先级向量），因此会更新其端口1的优先级向量，并选择端口1为其根端口，选择BR1为根网桥。
- BR3从BR4接收到的BPDU的优先级向量比BR3送给BR4的要高，但是，并不比BR3接收自BR1的那个更高。因此，BR3不会改变其当前根端口以及根网桥的信息：端口1依然是根端口，而BR1依然是根网桥。
- 当BR3发出一个新的BPDU给BR4时如图15-13（b）所示，会使用新的优先级向量，来反映从BR1取得的新信息。BR4接收该BPDU时，识别出那个较高级的优先级向量，于是阻塞其端口2。BR3的优先级向量优于BR4优先级向量，因为其路径开销较低（也就是BR3的端口2会被选为LAN的指定端口，因为和BR4的端口2相比，其离根网桥更近些）。
- BR4选择端口1为其根端口，因为其接收了较高级的优先级向量（我们已假设BR2的网桥ID比BR3的网桥ID还要低）。

如果比较一下BR4从BR2接收的配置BPDU和从BR3接收的配置BPDU，可以发现这两个BPDU都有相同的根网桥ID（BR1）和相同的根路径开销（10），但优先级向量的第三个元素则是BR2的更高级，因为BR2的网桥ID小于BR3的。因此，BR4会选择端口1作为其根端口。如果管理员偏爱BR4到BR3的链路胜过BR4到BR2的链路时，只要在端口2配置较低的开销即可（参见“网桥ID和端口ID”一节）。

在本例中，优先级向量的前三个元素就足够选出根端口和指定端口。下面我们通过图15-14所示的例子，说明何时需要第四个元素（端口ID）来做最后的决定。

现在BR4从BR2收到两个BPDU，它们的优先级向量中的前三个字段都有相同的值。然而，第四个参数（端口ID）可让BR4将其端口1选为根端口。在“选择根端口”一节中我们也说过，当入口BPDU的优先级向量的四个元素都不足以确认出获胜的BPDU时，网桥如何使用其本地端口ID（相对于作为优先级向量的一部分的远程端口ID）作为决定因素。

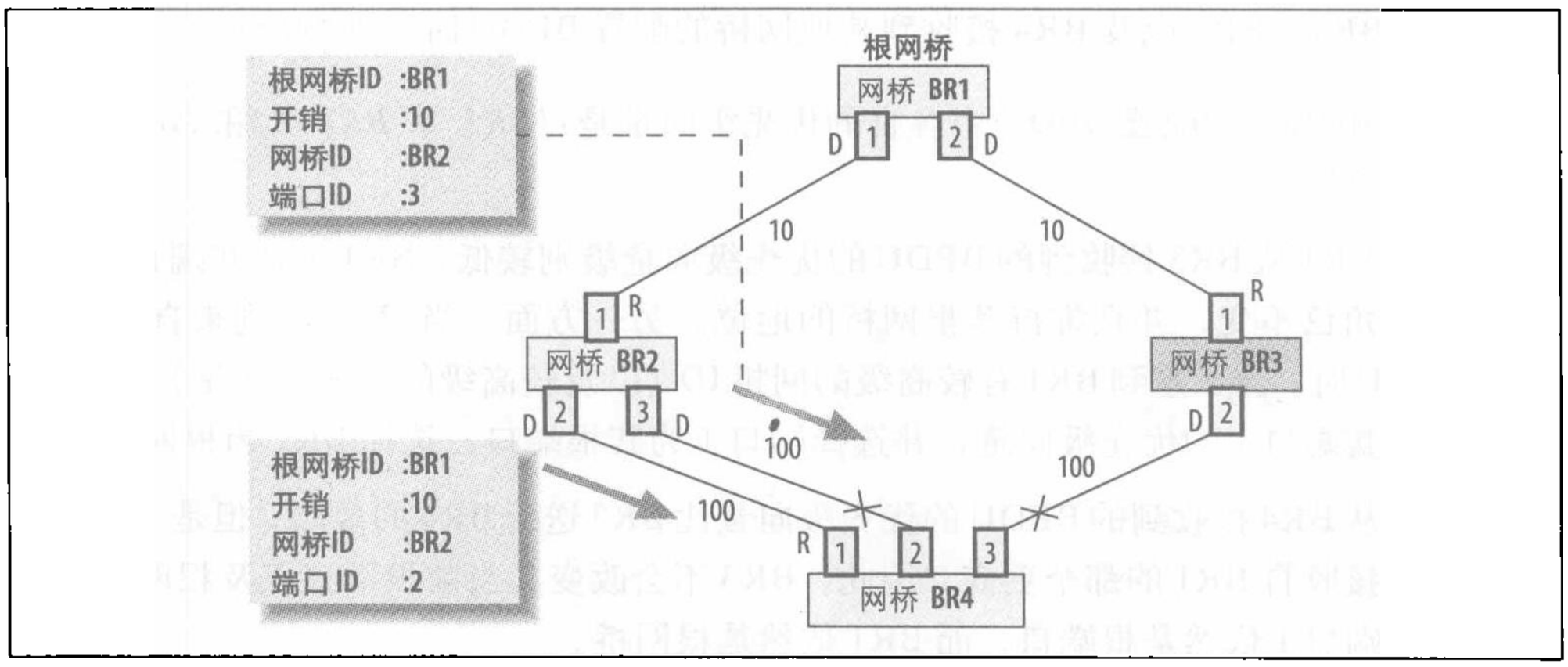


图 15-14：端口 ID 作为决定因素

定时器

STP 为每个网桥和每个端口都设有定时器。在表 15-4 和表 15-5 中，可以看到这些定时器的默认时限以及其取值范围（注 6）。

表 15-4：网桥定时器

定时器	默认值（s）	取值范围
Hello	2	1～10
Topology Change	转发延迟 + 最大生存期	未定义
TCN	Hello 时间	未定义
Addresses Aging	300 或 转发延迟[a]	未定义

a： 参见“拓扑变化”一节

表 15-5：端口定时器

定时器	默认值（s）	取值范围
Message Age	20	6～40
Forward Delay	15	4～30
Hold	1	未定义

注 6： 如果你想知道这些定时器是如何定义的，可以阅读 IEEE 802.1d 规范，或者上网（*http://www.cisco.com/warp/public/473/122.html*）查看。

注意并不是所有的定时器都可由用户配置。还要注意，有些定时器使用相同的配置（例如，TCN 和 Hello 定时器），因此若更新一个定时器的配置，也会影响其他定时器。

以下这些是网桥使用的定时器：

Hello

用于定期产生配置 BPDU。只有根网桥可以使用该定时器。

TCN

由检测到拓扑变化而且必须通知根网桥的网桥使用。参见“拓扑变化”一节。

Topology change（拓扑变化，TC）

由根网桥使用，以便记住要在其配置BPDU中设定一个特殊标识。此标识用于将拓扑变化通知其他网桥。参见“拓扑变化”一节。

Aging 定时器

用于从转发数据库中清理无效的地址。该定时器由网桥使用，而无论其是否启用 STP。参见“Short Aging 定时器”一节。

每个网桥都会保存其定时器配置的两个副本：一个来自管理员的本地配置信息，一个是接收自根网桥的配置信息。

根网桥是唯一使用其自身定时器配置的网桥；根网桥让所有其他网桥都采纳它的配置。非根网桥使用它们在其根端口所接收的BPDU中所携带的定时器配置。在图15-8中可以看到定时器配置信息。

以下是端口定时器：

Message Age

由“BPDU老化”一节已经知道，BPDU所携带的信息的生存期是有限的。Message Age定时器用于强制执行这个生存期。每当端口上接收了一个BPDU之后，该定时器就会重新启动。只要接收到一个BPDU，就会对它的消息生存期和网络允许的最大生存期做一比较，如果前者大于后者，该帧就会被丢弃。Message Age定时器在非指定端口上运行（也就是收到较高级 BPDU 的端口）。

在没有任何问题的稳定的网络环境下，该定时器绝不会到期。但当根网桥无法产生BPDU，或者接收到的BPDU已过期，或者因某种原因BPDU被丢弃时，此定时器就会到期。当此定时器到期时，该端口就会重新启动，被分配为指定角色。

Forward Delay

该定时器负责状态转移：从监听到学习以及从学习到转发。图15-15所示为Forward Delay 定时器到期时的典型处理方式和其如何遵循图 15-6 的模型。

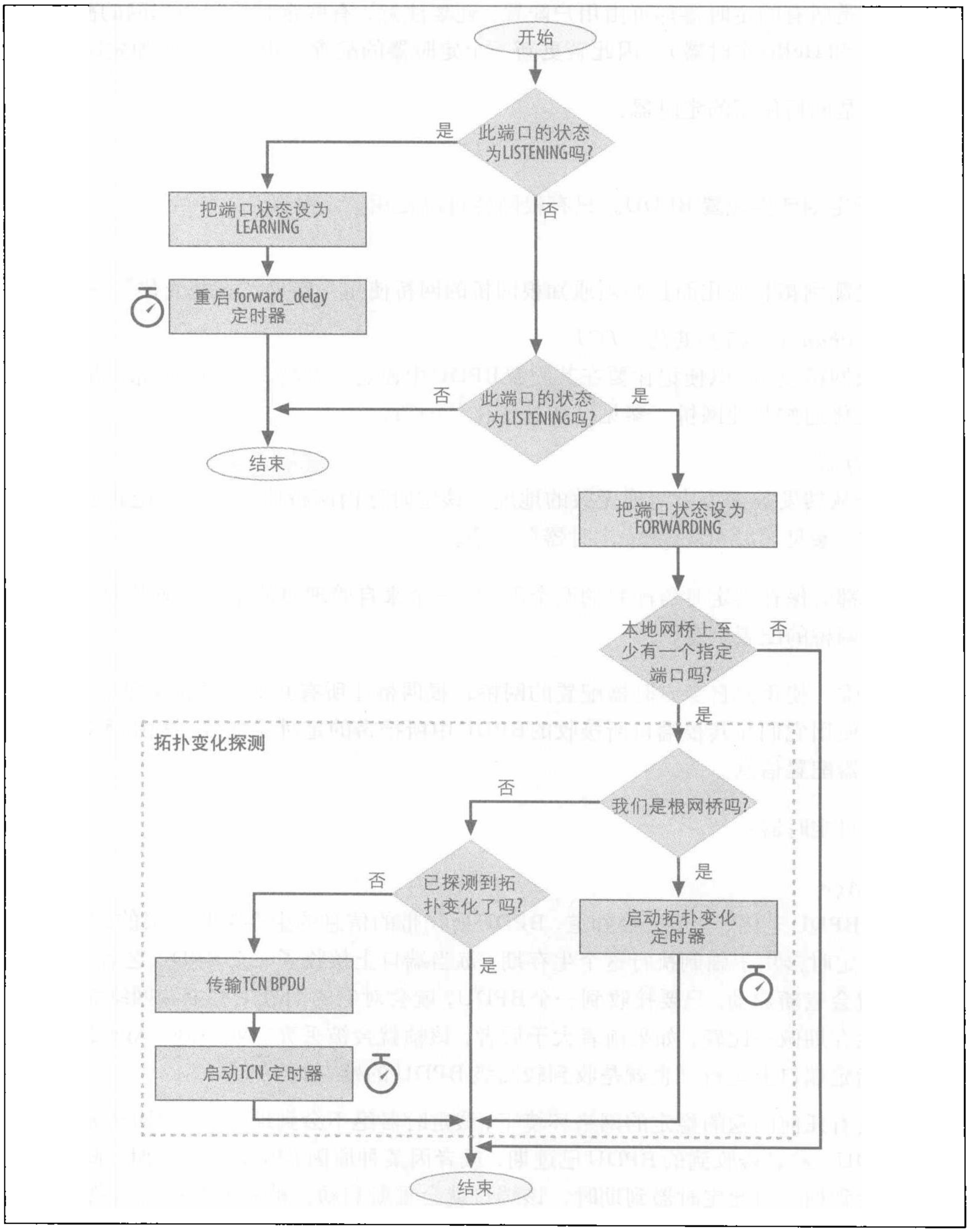


图15-15：Forward Delay定时器的处理流程

Hold

配置BPDU的传输是有速率限制的：每个端口每秒一个。在稳定的网络中（也就是STP收敛的网络），每个指定端口每间隔Hello时间就会发出一个BPDU。然而，在一个大而复杂的网络中，当拓扑发生变化时，由于STP算法的分布式特性，使得拓扑收敛至新拓扑的时间可能要花上几分钟。根据“何时传输配置BPDU”一节的规则，此时传送的BPDU帧数量很容易就会变得很大，这种情况下就需要速率限制来发挥作用。

必要时，Hold 定时器会在指定端口（传输配置 BPDU 的那个端口）上运行。

每个端口的定时器使用同样的配置。例如，两个不同的端口不能设置两个不同的 Max Age 定时器。

消除临时环路

根端口和指定端口是唯一处于转发状态的端口。当一个端口被分配为根角色或指定角色时，不会立刻设为转发状态：首先必须得经过两个中间状态，如图 15-6 所示。这些中间状态会在网络收敛为一个稳定的无环路拓扑时，减少形成临时性环路的风险。下面我们以图 15-16（a）所示的简单拓扑为例来加以说明。

该拓扑有两台网桥，它们由两条链路连接起来。这两条链路其中一个必须关闭，否则就会形成一个环路。

前面已经提到，当一台网桥的端口首次启动时，都会设为指定角色并进入阻塞状态。在“选择根网桥”一节中我们也曾说过，当一台网桥首次启动时，它对其相邻网桥一无所知，因此会以为自己就是根网桥。图 15-16（a）中给出的两台网桥刚刚启动，因此：

- 它们都认为自身就是根网桥。
- 这两台网桥的端口都设为指定角色和阻塞状态。
- 每个端口的状态会变成监听态，接着启动 Forward Delay 定时器，发出一个配置 BPDU。这些 BPDU 的优先级向量会反映出它们的网桥为根网桥的假设。

注意，此时这些端口都还没处于转发状态。这些端口还不能接收或传输数据流量，只可以传输和接收 BPDU。

当 BR2 在其端口上接收到 BR1 的 BPDU 时，会发现 BR1 的优先级向量较高级（确切地说就是较好的网桥 ID）。此时，BR2 会着手进行配置更新：选出根网桥、根端口和指定端口，然后更新其所有端口的状态。特别地，它选出BR1为根网桥，端口1为根端口（因为此端口就是接收具有最佳优先级向量的 BPDU 之处）。端口 2 既不是根端口，也不是

指定端口，因此会被阻塞（也就是不在此树内）。当端口 1 分配到新角色时，其 Forward Delay 定时器会重启。当端口 2 被阻塞时，其 Forward Delay 定时器就会停止。

假设这些动作都发生得很快，那么可以认为 Forward Delay 定时器在所有端口上差不多将会同时到期，因而可以得到如图 15-16（b）所示的新配置。现在，请读者注意：

- 三个 Forward Delay 定时器已到期的端口会进入学习状态（还没开始转发数据流量）。
- 这三个端口上的 Forward Delay 定时器会重新启动。
- BR2 不再传输配置 BPDU（因为它没有指定端口）。

当 Forward Delay 定时器于 15s 后再次到期时，BR1 的端口 1 和端口 2 以及 BR2 的端口 1 都会设为转发状态。此时，该拓扑就稳定了。在这个简单应用中，该拓扑收敛得非常快，但在比较复杂的环境中可能会花上很长的时间，阻塞态和转发态之间的中间状态可以确保避开临时环路。

注意，在图 15-16（c）中，BR1 的端口 2 处于转发状态。只要链路的另一端是阻塞状态（图 15-16（c）中 BR2 的端口 2），就不存在形成环路的危险。尽管 BR2 的端口 2 是关闭的，BR1 的端口 2 也会一直转发流量：BR1 的端口 2 和 BR2 的端口 2 可能与其他主机连接到 hub 上，这样通过 BR2 的端口 2 就可以连通其他主机。

拓扑变化

拓扑变化（topology change）就是指在 L2 网络上的系统发生的变化，或者它们连接的端口发生的变化。当拓扑发生变化时，要到达一个地址不再按照通常所走的链路，而是通过一条不同的链路。因此拓扑变化必须被正确处理，才能让网络免除环路，并且及时更新转发数据库。从图形和树的观点看，当增加或删除一条链路，或者选择其他节点作为该树的根时，拓扑就发生了变化（生成树的判断是基于对树的根的选择）。

我们先看看会触发拓扑变化的事件，以及这些事件是如何处理的：

非转发态的网桥端口变成转发态（或者反之）

这种情况包括关闭的端口被启动，和因为协议的裁决仅改变了端口状态。从树的角度看，相当于给树增加一条链路或从树上删除一条链路。

根网桥 ID 变化

这种情况的发生可能是因为当前根网桥已被关闭（因此另一台网桥承担起该角色），或者是因为有更高优先级的网桥被开启了，还有可能是因为当前根网桥或另一台网

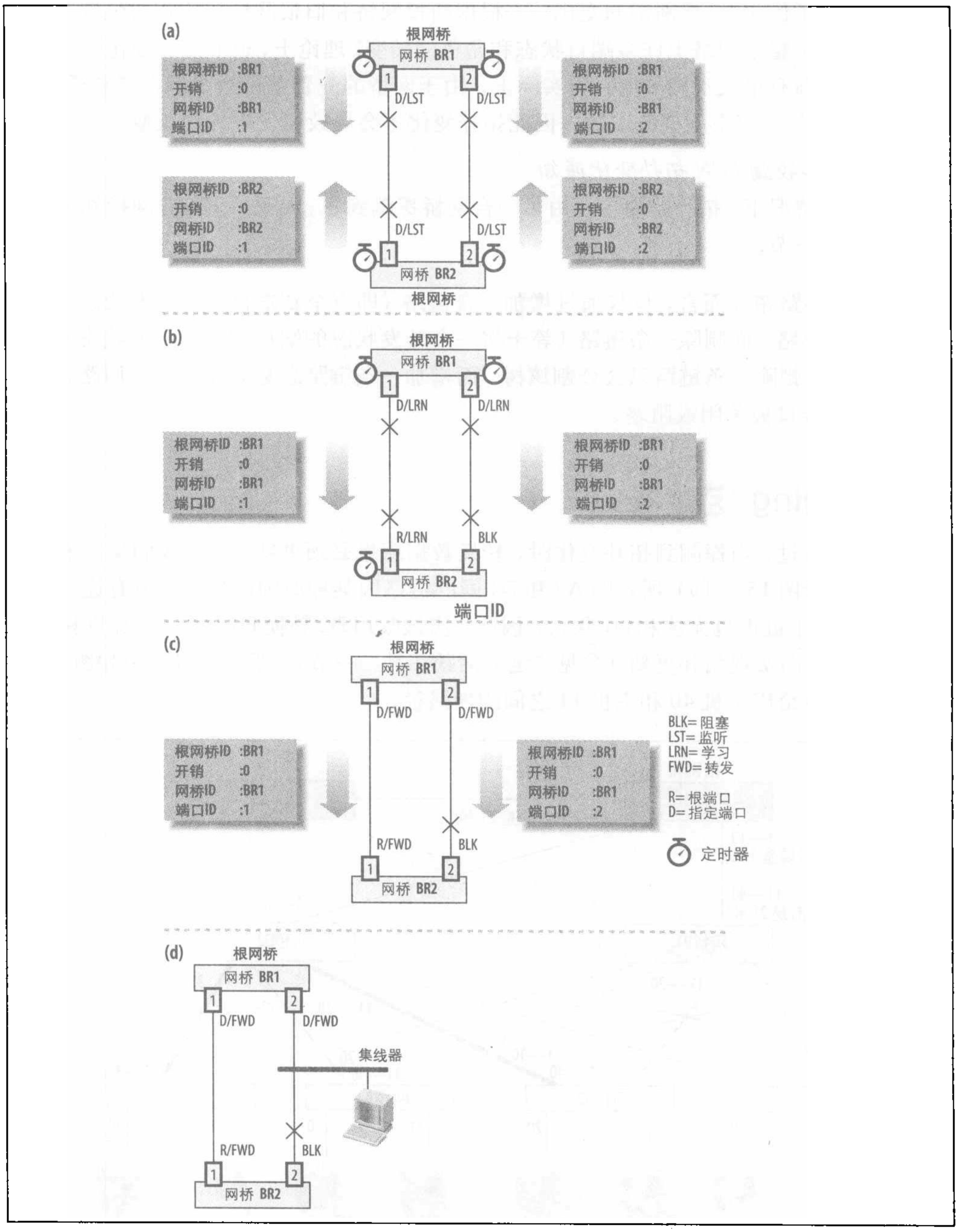


图 15-16：转移到转发状态

桥改变了优先级。根网桥的变化——根据新根网桥和旧根网桥间的相对位置——可能会触发整个网络上许多端口状态和角色的转变。理论上，根节点的变化会生成一棵和以前有很大不同的树，但实际上，由于网桥的配置是使用我们在“网桥 ID 和端口 ID”一节中所说的参数，因此拓扑变化不会导致树发生大的改变。

网桥端口上接收到 TCN 拓扑变化通知

在这种情况下，拓扑的变化已由另一台网桥探测到了。参见“让所有网桥知道拓扑变化”一节。

注意，对无环路拓扑而言，仅仅通过增加一条链路（即有个新端口进入转发状态）就可以建立一个环路，而删除一条链路（等于将一个转发状态的端口切换到阻塞状态）并不会建立环路。删除一条链路只会分割该树，而增加一条链路总是会建立一个回路，除非同时另一个端口被关闭或阻塞。

Short Aging 定时器

我在前面曾讲过，当探测到拓扑变化时，转发数据库也必须更新才行。我们举例来说明其原因。假设图 15-2（a）所示的 A2 和 D1 间的链路因某种原因而失效。所有连接至接入网桥 A2 的主机再也无法和 D1 联系，因此，应该改用 D2 替换 D1。STP 会让网桥经过配置更新，从而实现拓扑更新（参见“定义活跃拓扑”一节）。新拓扑看起来如图 15-17 所示。图中也给出主机 40 和主机 11 之间的新路径。

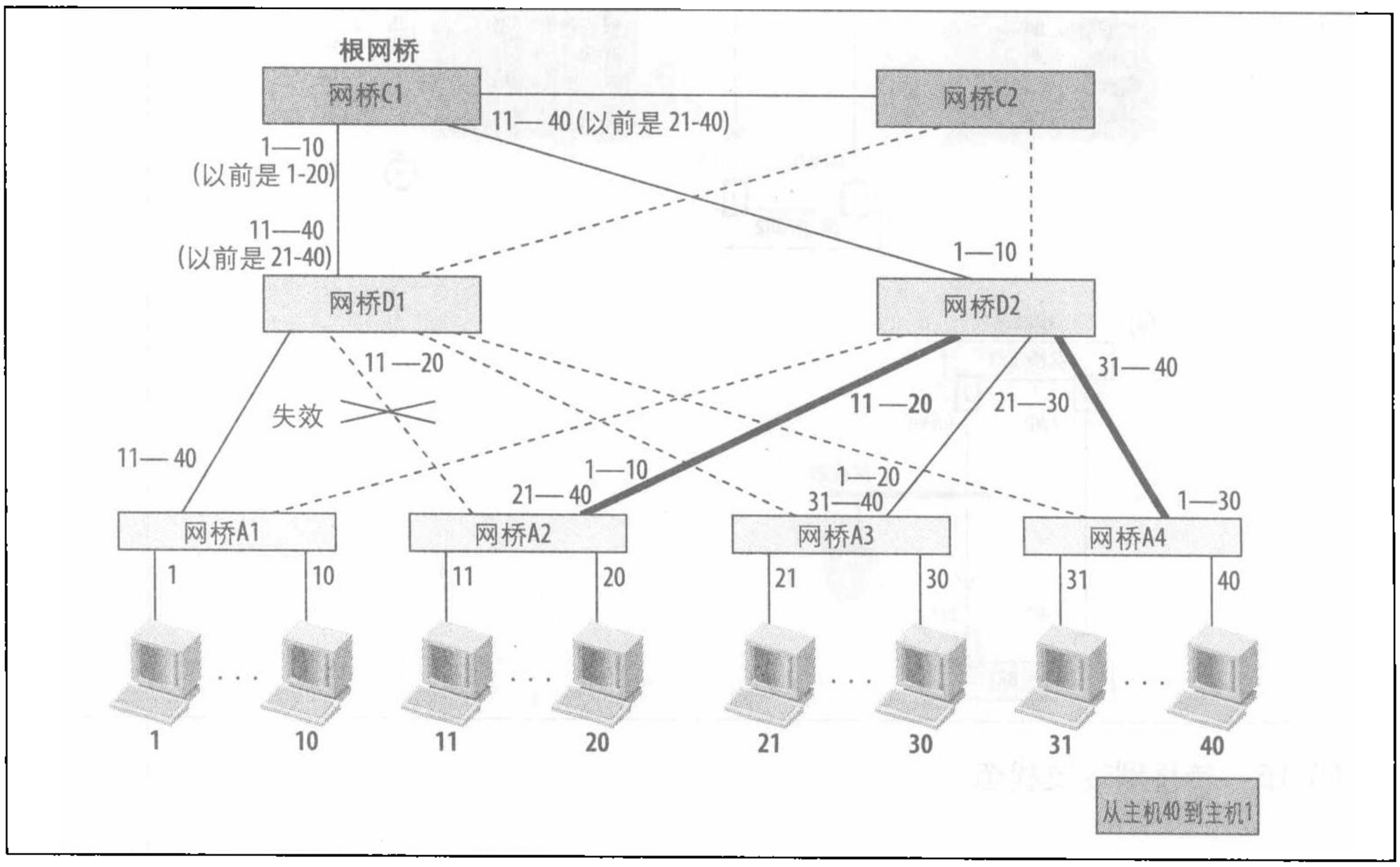


图 15-17：A2 根端口失效时的处理

STP 也会确保 D2 中的过期信息得到更新（例如，主机 11～20 可以通过 D2 的端口连到 C1）。过期无效信息实际上不仅只在该网桥上；其他网桥的转发数据库也必须清理。此外，当拓扑有变化时，STP 必须收敛到一个新的无环路拓扑。在收敛期间，网桥端口可能会数次改变其角色和状态，甚至转发数据库的内容也会更新数次。

转发数据库中的地址若不再被使用，过一段时间才被删除。数据库中的过期数据的清理就是通过缩短这段时间来实现的。其具体做法就是当网桥接收到拓扑变化通知后，就把 Aging定时器缩短（默认值是5min）为转发延迟的时间（默认为15s）。通过在配置BPDU 中设定一个特殊标识来通知其他网桥拓扑发生变化（参见下一节）。

让所有网桥知道拓扑变化

当网桥探测到有拓扑变化，必须通知所有网桥，以使其可以用缩短的 aging 时间来清理它们转发数据库内的过期数据项。下面我们来说明这个过程：

1. 探测到拓扑变化的网桥通知根网桥此变化。
2. 根网桥通知所有网桥拓扑发生变化。

第一步是利用 TCN BPDU 实现的。探测到拓扑变化的网桥会从根端口送出一个 TCN BPDU 至其指定网桥。该网桥每隔 Hello 时间就会发出一个 TCN BPDU，直到指定网桥确认说已收到该 TCN BPDU。指定网桥确认收到 TCN BPDU 的方式是，在其发出的下一个配置 BPDU 中设定 TCA 标识。此时，指定网桥会重复相同过程，通过根端口发出一个 TCN BPDU 至其指定网桥。当 TCN 最终抵达根网桥时，这个过程就结束。若拓扑变化是由根网桥自己探测到的，就不需要使用 TCN BPDU（因为根网桥不需要通知自己）。

第二步是由根网桥发出配置 BPDU，并在该配置 BPDU 中设定一个特殊标识（TC）。其他非根网桥重新产生的BPDU中都会附带该标识，使得网络上所有网桥最后都会接收到拓扑变化通知。当一个网桥看到该标识被设置时，就会启动 Short Aging 定时器。

拓扑变化实例

如果以图 15-2（a）的网络拓扑为例，并设想一下，若 A2 和 D1 之间的链路（也就是根端口）关闭了，则 A2 会选出其他端口作为根端口，因而该端口的状态会从阻塞转为转发。这样就得到了如图 15-18（a）所示的新拓扑。

A2 启动了 TCN 定时器，然后从其（新）根端口发出一个 TCN BPDU。当 D2 接收到 TCN BPDU 时，会传回一个设有 TCA 标识的配置 BPDU 作为应答，然后启动 TCN 定时器，

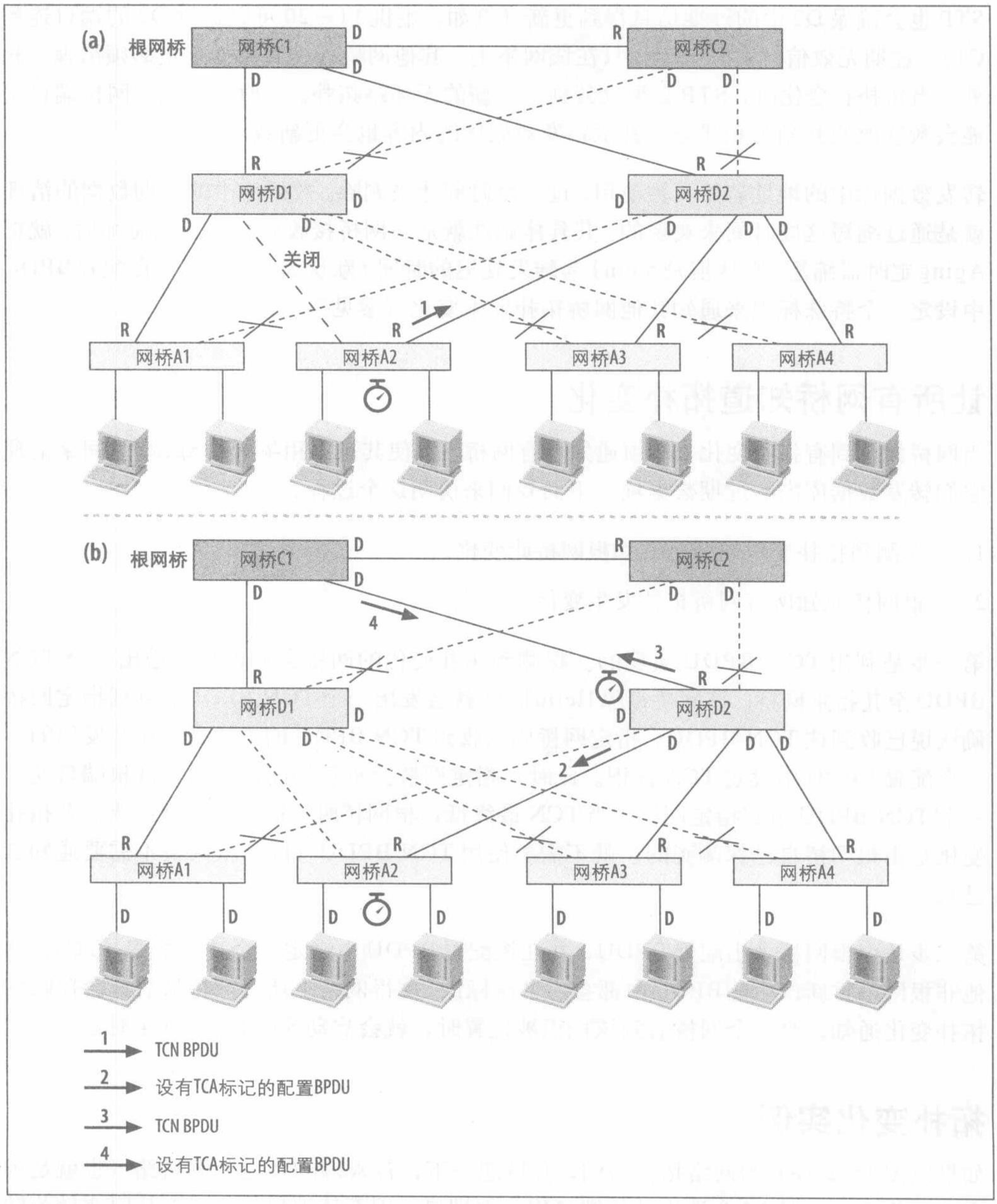


图 15-18：TCN BPDU 的使用

再从其根端口传出一个 TCN BPDU。当 A2 接收来自于 D2 的应答时，就会停止其 TCN 定时器。当 D2 接收来自于 C1 的应答后，也会做同样的事。

当C1收到TCN BPDU时，会启动Topology Change定时器（会持续35s），然后在该定时器未决的期间，在所有发送出去的BPDU上设定TC标识（见图15-19）。Topology Change定时器所用的35s并不是随机值：它等于转发延迟加最大生存期（见表15-4）。

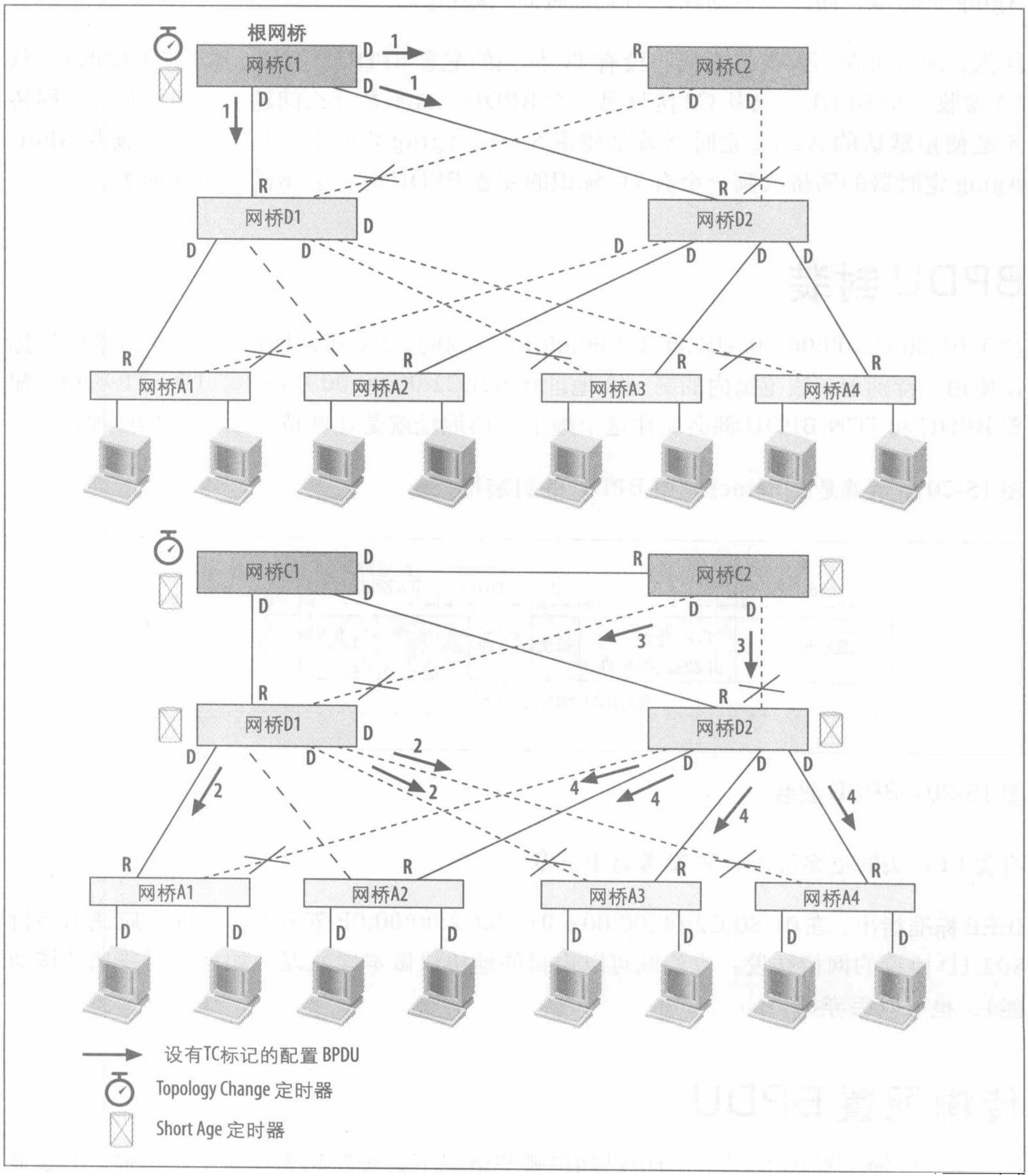


图15-19：将拓扑变化通知所有网桥

TC标识会往下传播到整个树上，因为所有网桥都会转发接收自根网桥的标识。当一台网

桥看见一个入口 BPDU 上设有此标识时，就会启动 Short Aging 定时器（如果还没启动的话）。一旦根网桥的 Topology Change 定时器到期了，根网桥就会停止在其 BPDU 中设置 TC 标识。其他网桥接收到一个清除掉 TC 标识的 BPDU 时，就会停止使用 Short Aging 定时器，而开始启动默认的老化时间（aging）。

注意，网桥可在不同端口上接收设有 TC 标识的配置 BPDU。例如，图 15-19 中的 D2 从 C1 接收一个 BPDU，又从 C2 接收另一个 BPDU。这不是什么问题：在任何时刻，网桥不是使用默认的 Aging 定时器就是使用 Short Aging 定时器，所以当正在使用 Short Aging 定时器的网桥收到一个有 TC 标识的配置 BPDU 时，它不需要做任何事。

BPDU 封装

位于 01:80:C2:00:00:00～01:80:C2:00:00:FF 之间的 L2 多播地址由 IEEE 保留给标准协议使用。特别地，该范围内的第一个地址 01:80:C2:00:00:00 是由 802.1D STP 所用：配置 BPDU 和 TCN BPDU 都会发往这个地址。该地址就是让网桥识别出 BPDU 帧。

图 15-20 所示就是 Ethernet 帧中 BPDU 的封装格式。

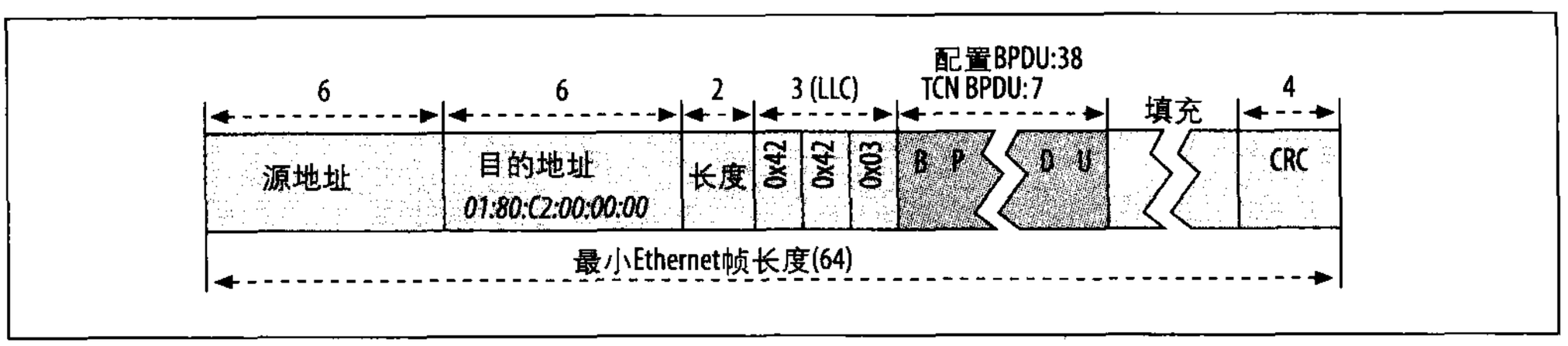


图 15-20：BPDU 封装

有关 LLC 头的更多细节，可参考第十三章。

IEEE 标准指出，在 01:80:C2:00:00:00～01:80:C2:00:00:0F 范围内的地址不应该由运行 802.1D 协议的网桥转发：它们既可以由目的地协议做本地处理（如果实现并启动该功能），也可以丢弃掉。

传输配置 BPDU

在“何时传输配置 BPDU”一节中已知道哪些情况下会触发配置 BPDU 的传输。不管是出于什么原因来传输配置 BPDU，都可适用如图 15-21 所示的逻辑。

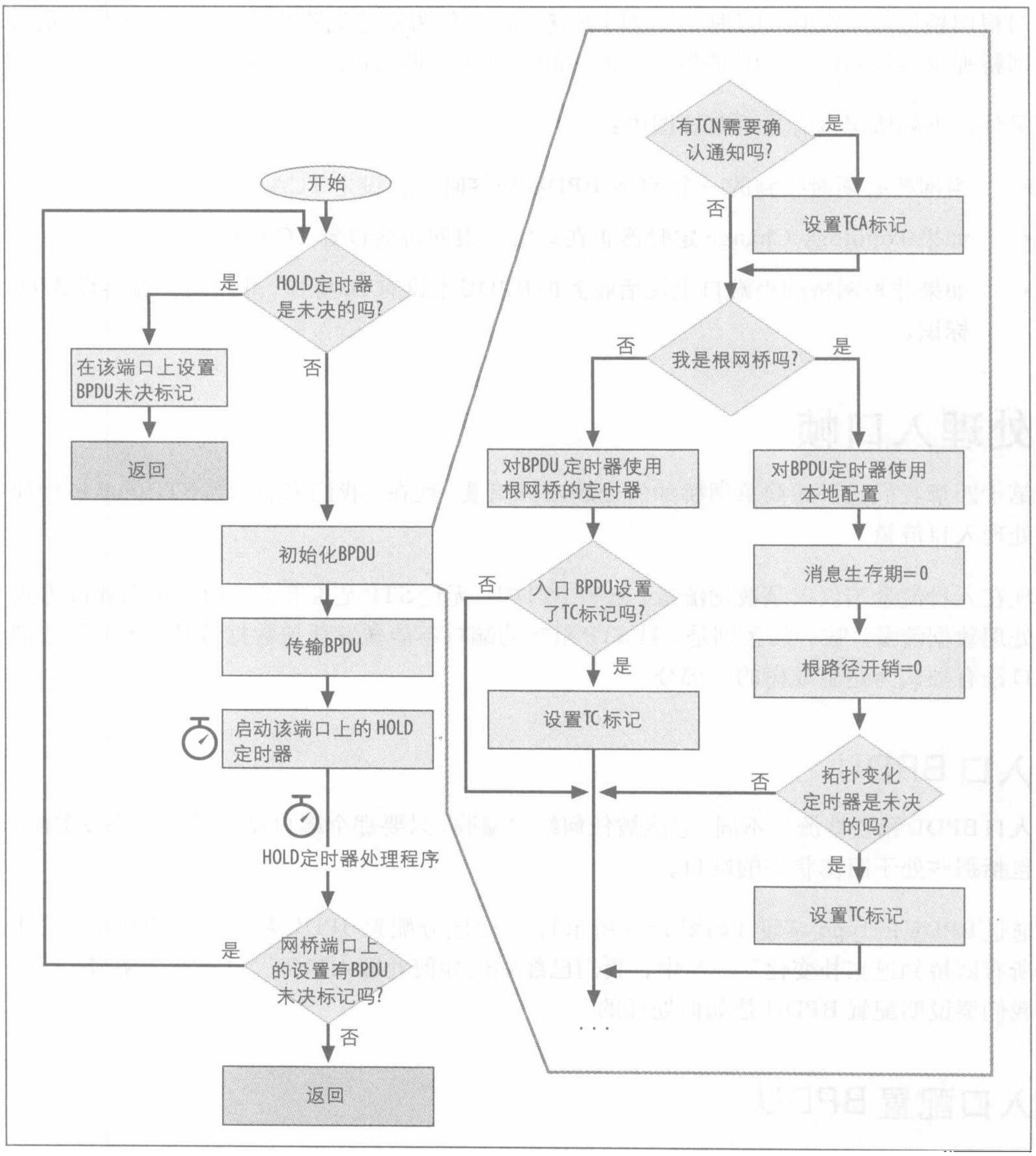


图 15-21：配置 BPDU 传输逻辑

每个端口的 HOLD 定时器会对速率进行限制：每秒一个 BPDU。当一个 BPDU 传输后，该定时器就会启动。如果再想尝试另一次传输，而此定时器正处于未决状态，则该BPDU不会被传输，并且网桥端口配置块中会设置一个标识。当该定时器到期时，就会检查该标识，然后，如果发现设置了该标识就传输一个配置 BPDU。

当根网桥传输一个BPDU时，其定时器就被初始化为本地配置的值；否则，就使用从根网桥那里接收到的BPDU的值。根网桥的消息生存期和根路径开销都是0。

另外，下列情况没有显示在该图中：

- 当网桥必须对收到的一个TCN BPDU应答时，会设定TCA标识。
- 如果Topology Change定时器正在运行，根网桥会设置TC标识。
- 如果非根网桥的根端口上最后收到的BPDU上设有TC标识，非根网桥就会设置TC标识。

处理入口帧

第十四章我们已讲述简单网桥如何处理入口流量。现在，我们来看运行STP的网桥如何处理入口流量。

现在入口流量不仅仅是数据流量，还有BPDU。无论STP是否开启，网桥都以相同方式处理数据流量。唯一的区别是，被STP阻塞的端口不能转发任何数据流量，因为那些端口没有被认为是生成树的一部分。

入口BPDU

入口BPDU和数据流量不同，它能被任何端口接收，只要那个端口没通过管理手段关闭，包括那些处于阻塞状态的端口。

通过BPDU的type字段（如图15-7所示），可以区分配置BPDU和TCN BPDU。在“让所有网桥知道拓扑变化”一节中，我们已经知道如何处理入口TCN BPDU。在下一节，我们要说明配置BPDU是如何处理的。

入口配置BPDU

图15-22所示是入口配置BPDU的处理过程。

入口BPDU的处理取决于其优先级向量是否：

比进行接收的网桥端口当前知道的优先级向量还要更好

这种情况下，BPDU会引起配置更新，包括新的根端口、指定端口以及所有端口的新状态。

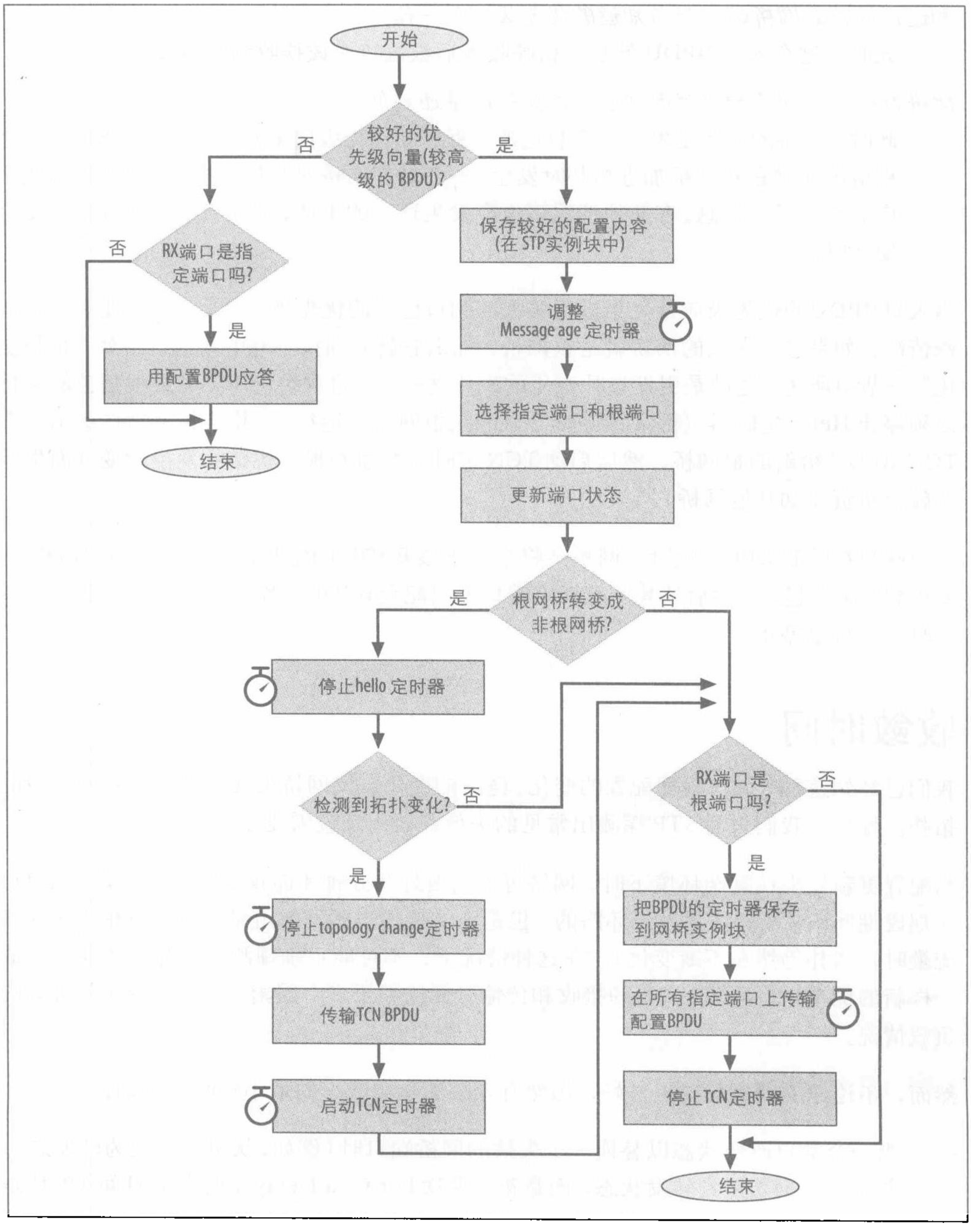


图 15-22：处理入口配置 BPDU

和进行接收的网桥端口当前知道的优先级向量一样

此时，这个入口 BPDU 就是当拓扑收敛后根端口应该接收到的 BPDU。

比进行接收的网桥端口当前知道的优先级向量还要低

此时，网桥的应答是发出带有自己的（更好的优先级向量）信息的配置BPDU。这种情况通常在新网桥加进拓扑时发生：一开始新网桥对其他网桥一无所知，因此会广播它自己的信息。在其他情况下也会发生这样的事情，例如当某台网桥配置发生变化时。

当入口BPDU的优先级向量优于进行接收的端口已知的优先级向量时，需要处理一种特殊情况：如果进行接收的网桥就是根网桥，那么它就必须摘下它的皇冠。正如“拓扑变化”一节中所述，这就是引发拓扑变化的事件之一。在这种情况下，失去根角色的网桥必须停止 Hello 定时器（因为该定时器只能在根网桥上运行），并从其根端口送出一个 TCN BPDU 给新的根网桥，然后启动 TCN 定时器并通知根网桥拓扑发生改变（而根网桥就会负责通知其他网桥）。

当根端口接收到 BPDU 帧时，网桥会将来自于该BPDU 的定时器（在其出口 BPDU 中要用到）保存起来，然后从其所有指定端口发出配置BPDU。若设置了TCA标识，TCN 定时器就可以停止。

收敛时间

我们已经知道STP如何根据配置的变化、链路的变化以及网桥失效来动态更新网络树的拓扑。现在，我们来看 STP 探测出常见的失效并做出反应需要多长时间。

当配置更新发生在复杂环境下时，网络可能需要好几分钟才能重新收敛并稳定（注 7）。在那段期间内，拓扑依然是无环路的，但是无法传输正常网络流量（因为当在传送网络流量时，拓扑仍然在不断变化）。在这种情况下，不可能正确预测出拓扑会如何演化成一棵新的稳定树，因为BPDU的接收和传输时间依赖于多个因素，比如网桥在某时刻的负载情况。

然而，不论你的网桥配置得多好，仍然有无法消除或减少的最小延时。例如：

- 当一个端口改变状态以替换一个失效的网桥端口时（例如，从阻塞态变为转发态），并不是立刻切换到转发状态，而是花了两次Forward Delay定时器的时间（默认是 30s），如图 15-6 所示。该端口在这段期间内无法转发任何数据流量。

注 7： 可以人工配置网桥来减少失效和配置更新的影响，这样也能有助于拓扑的快速收敛。

- 根端口和非指定端口（即接收BPDU的端口）只有在其消息生存期定时器超时后，才会发现它们和相应的指定网桥失去连接（因此，除了在该网桥指定端口以下的部分外，其他部分和整棵树失去了连接（注 8））。例如，在图 15-2（a）中，如果通往 D1 的 C1 端口因某种原因而失效，只有在 Max Age 定时器 20s 后到期时 D1 才会知道此事。

这两种情况都是由定时器驱动的。当然，你也可以配置Forward Delay和Max Age这两个定时器，使其早一点到期，从而得以缩短收敛时间。但是，依赖于网络的复杂度，你不可能让这些定时器太过于积极。

我们以图 15-2 为例，来看看当D1网桥因某种原因失效时会发生什么事。因为A1和A2都是通过 D1 去访问网络其他部分，一旦 D1 失效，所有连接到 A1 和 A2 的主机就和网络的其他部分隔离开来了，一直到STP设法替A1和A2选出新的根端口。那么，STP要花多少时间才能完成这个变化呢？在最坏的情况下，会发生下列事情：

- D1 无法正常工作。
- 20s 后，A1 和 A2 的根端口的 Message Age 定时器都到期。
- A1 和 A2 选择通往 D2 的端口作为新的根端口。
- 又过了 30s 后，这些新的根端口会进入转发状态。

也就是说，连接至 A1 和 A2 的主机有 50s 的潜在黑洞时期。在复杂网络中，拓扑收敛所需时间可能比这个还要长，甚至是好几分钟。

新版生成树协议概况

当 802.1D-1998 STP 在几年前由 IEEE 委员会首次定义完成时，其收敛时间是可以接受的。然而，这几年来，由于新的网络应用程序，例如，交互式多媒体（IP 电话、视频会议等）有更高的可用性需求，那个收敛时间已被证明太慢了，更别提不断增长的用户期望了。

为了解决这个问题，各种商业网桥生产商私下里对STP做了改进。可惜的是，这种改进通常都无法用在采用了多家厂商设备的异构网络上。

最近，IEEE 提出两个较新的协议，RSTP（Rapid Spanning Tree Protocol，快速生成树协议）和 MSTP（Multiple Spanning Tree Protocol，多重生成树协议），这两个协议解

注 8：　因此，如果网桥是树的叶子（如接入网桥），就完全与树隔离了。

决了老一代协议802.1D目前已知的所有重大缺陷。限于篇幅，我们在这里就不讨论这两种协议。不过，目前Linux还不支持这两种协议。接下来的两小节中，我们只探讨这些新协议所提供的一些主要改进。更详细的说明可参考下列文档：

- 快速生成树协议 (802.1w) 参见 *http://www.cisco.com/warp/public/473/146.html*。
- 多重生成树 (802.1s) 参见 *http://www.cisco.com/warp/public/473/147.html*。

快速生成树协议（RSTP）

RSTP和旧的802.1D向下兼容，因此，运行旧协议的网桥和新协议的网桥彼此互连是没有问题的。但并非RSTP的所有改进都可以在这类异构网络中使用。下面是RSTP中的一些改进：

- 每个网桥端口现在都会分派一个角色。那些既不是指定角色也不是根角色的端口，就会分派替代（alternate）角色或备用（backup）角色。替代是用于表示通往根网桥的替代路径的端口（是当前根路径的潜在替换路径），而备用则表示通往子树的替代路径的端口（指定端口的潜在替换端口）。例如：
  - 在图15-10中，BR2的端口2就会是替代端口，因其提供了通往根网桥的替代路径。
  - 在图15-11（c）中，BR1的端口2是备用端口，因其代表指定端口的候选端口，但是，BR1上的另一个端口有更好的优先级向量。
- 可能分派给端口的状态经过简化：新的丢弃状态包括老旧、关闭、阻塞以及监听状态。
- RSTP把端口切换到转发状态的速度快很多，通过端口间的握手和称为sync的机制，sync机制能确保够避开环路。RSTP的这项新的有趣的改进只对点对点链接有效。
- 当根端口被替换，它可立即进入转发状态（也就是不必等Forward Delay定时器到期两次，如图15-6所示）。因为该协议有一个机制可确保实时转移到转发状态时不会造成任何环路。
- 前两项改进意味着收敛时间大幅加快，也许在一秒内（取决于拓扑的复杂度）。
- 所有网桥都运行Hello定时器，而且独立产生配置BPDU。这样可以更快地探测连通性问题。
- 探测连通性问题不再依靠Max Age定时器。当一个应该接收BPDU的端口连续三个Hello时间内都没接收到BPDU时，就会启动恢复机制。旧的Max Age定时器依然在运行，但只有当先前涉及的新的RSTP进程不可用时，才会用到该定时器。

- 拓扑变化的处理也有所不同，不再需要TCN BPDU了。现在当网桥检测到拓扑变化时，只需从其根端口和指定端口发出有设置TC标识的BPDU就行。其他接收到这个BPDU的网桥则会重复这个过程：把设有TC标识的BPDU从其每个转发端口传送出去（那个接收原有BPDU的端口除外）。这个简单机制可以向四面八方广播拓扑变化。当一台网桥接收到设有TC标识的BPDU时，不会启动Short Aging定时器，相反，会清除其所有端口上所学习到的MAC地址（除了那个接收BPDU的端口）。
- 和STP相比，RSTP所用的BPDU的结构只有一点变化。标识字段使用全部的8位以适应新的需求。

写作本章之时，还没有Linux下可用的开源RSTP实现。但可以在SourceForge找到一个用户空间的RSTP仿真器（*http://rstplib.sourceforge.net*）。

多重生成树协议（MSTP）

定义RSTP时，MSTP差不多也设计出来了。MSTP引入主要的改进是可以定义多棵独立的生成树。每棵生成树都有其可传送数据流量的子集。

根据每个数据封包来自的VLAN的不同，为该封包选择所用的生成树。图15-23显示的主机分属两组不同的VLAN，而且对每一组VLAN，MSTP都会建立了一棵独自的生成树。

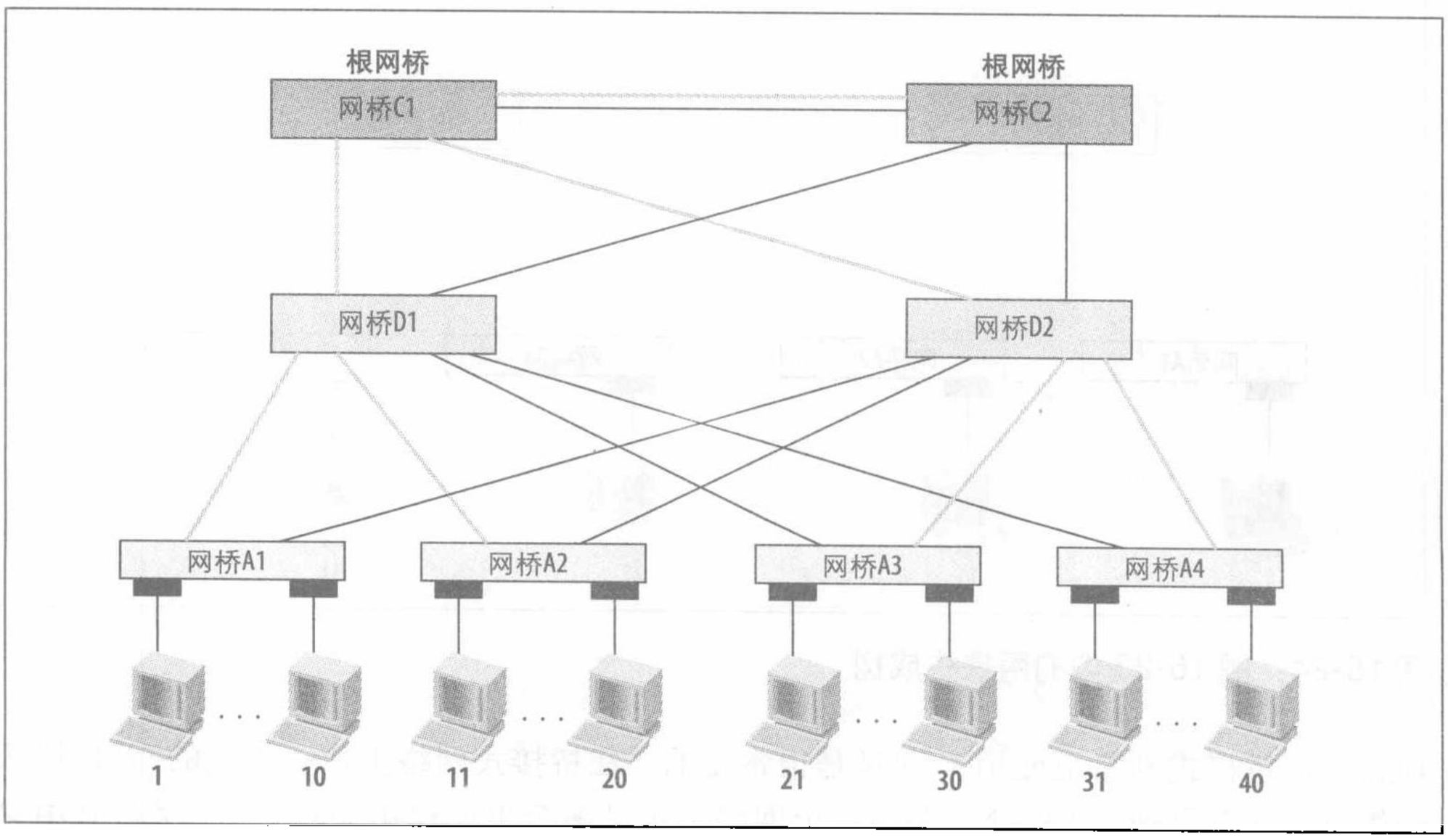


图15-23：定义两个生成树实体的桥接式网络实例

从主机的角度看，其结果如图 15-24 所示。

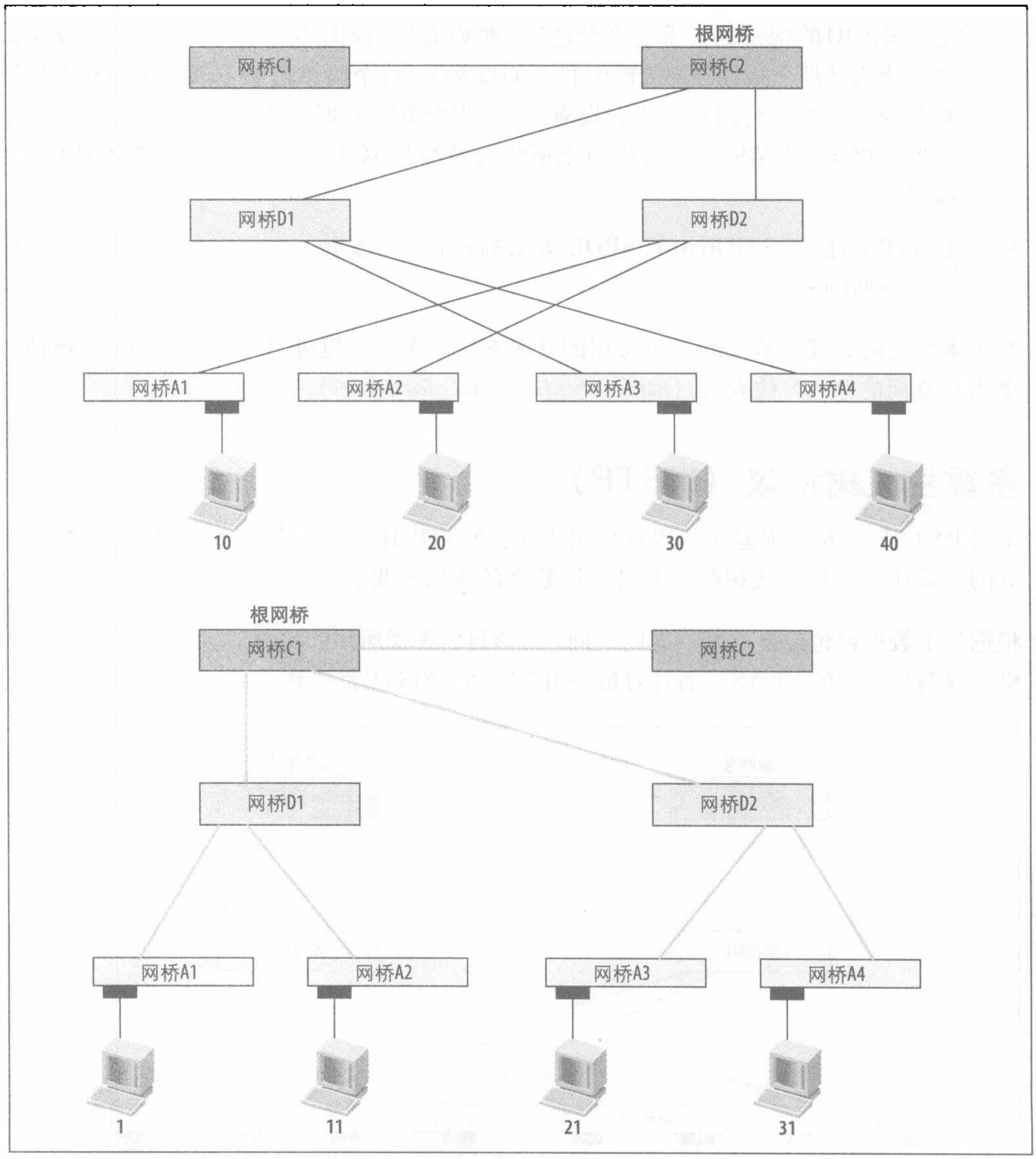


图 15-24：图 15-23 中的两棵生成树

现在，在桥接式网络上使用 VLAN 是很常见的。在桥接式网络上建立不同的 L2 广播域的简单方式就是通过 VLAN。在同一个网络上定义多个生成树拓扑有几个优点。其中之

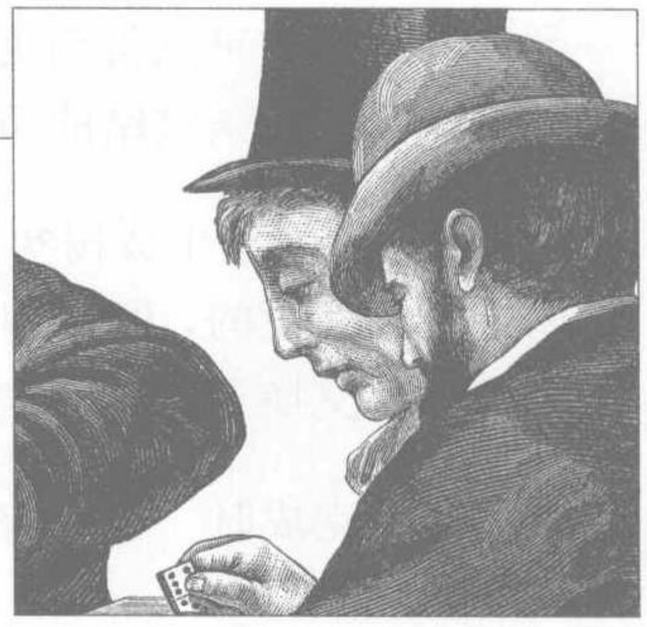
一就是网络带宽可以得到更好地利用（也就是对冗余链路做更好地运用，即负载平衡）。这样可以降低每条链路的负载。

MSTP让它的每个生成树实体都使用RSTP。该协议实际上比看起来更复杂：不同的STP实体是彼此独立的，但MSTP协议中有一特殊实体扮演中枢的角色，这个角色尤其要处理有关如何交换 BPDU 和怎样兼容以前版本的协议的信息。

到本书写作完成时，也没有 MSTP 的开源实现。

第十六章

桥接：Linux 实现



上一章讨论了网桥协议，本章说明Linux 下这些协议的实现。

第10章已了解了桥接程序如何在netif_receive_skb中捕获入口封包。本章我们将看到如何正确处理这些数据包。我们会了解到，在STP 启动和关闭的情况下，桥接程序如何维护设备状态和处理入口流量。

有关桥接程序的性能评价，请参考James T. Yu的文章《Performance Evaluation of Linux Bridge》（上网搜索一下即可找到）。

网桥设备抽象

对Linux而言，网桥是虚拟设备。因此，除非把一个或几个真实设备绑定到网桥设备上，否则它就无法接收或传输任何东西。我们以术语“绑定（enslave）”来指把真实设备绑定到（虚拟的）网桥设备的过程。

假设我们要实现如图 16-1 所示的拓扑。图中有如下几点需要强调：

- 两个LAN通过网桥连在一起。LAN1和LAN2的主机都配置成相同的子网：10.0.1.0/24。
- 网桥连接到路由器上，使得 LAN1 和 LAN2 的主机可以和 LAN3 的主机通信。
- 从路由器的角度看，在 *eth0* 上只有一个 LAN。

因为Linux 实现了桥接和路由，我们可以将这两个设备合并到一个Linux 系统中，从而获得类似如图 16-2（a）所示的拓扑。网桥和路由器间的网络连接是内核的事。

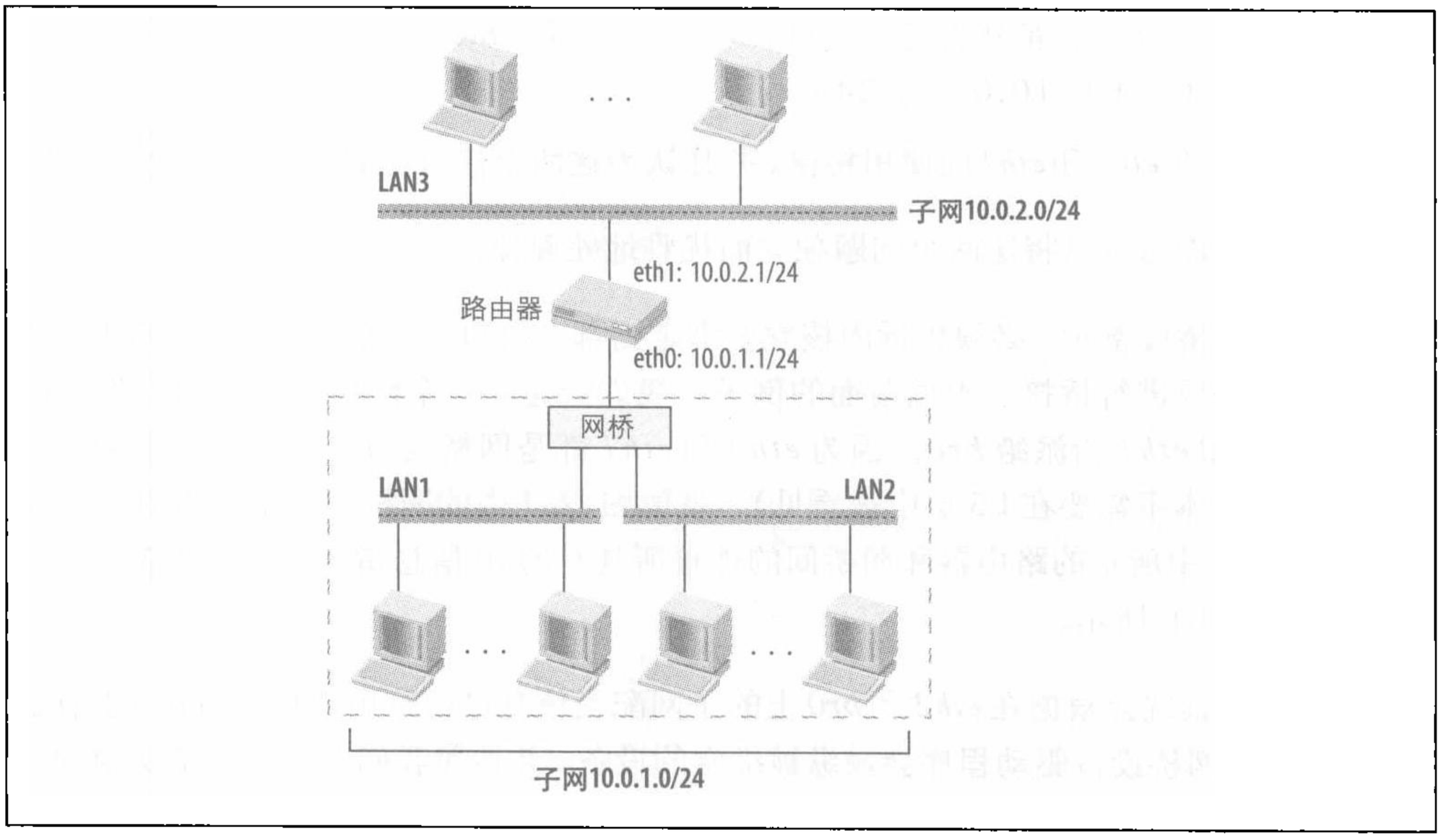


图 16-1：网桥使用的范例

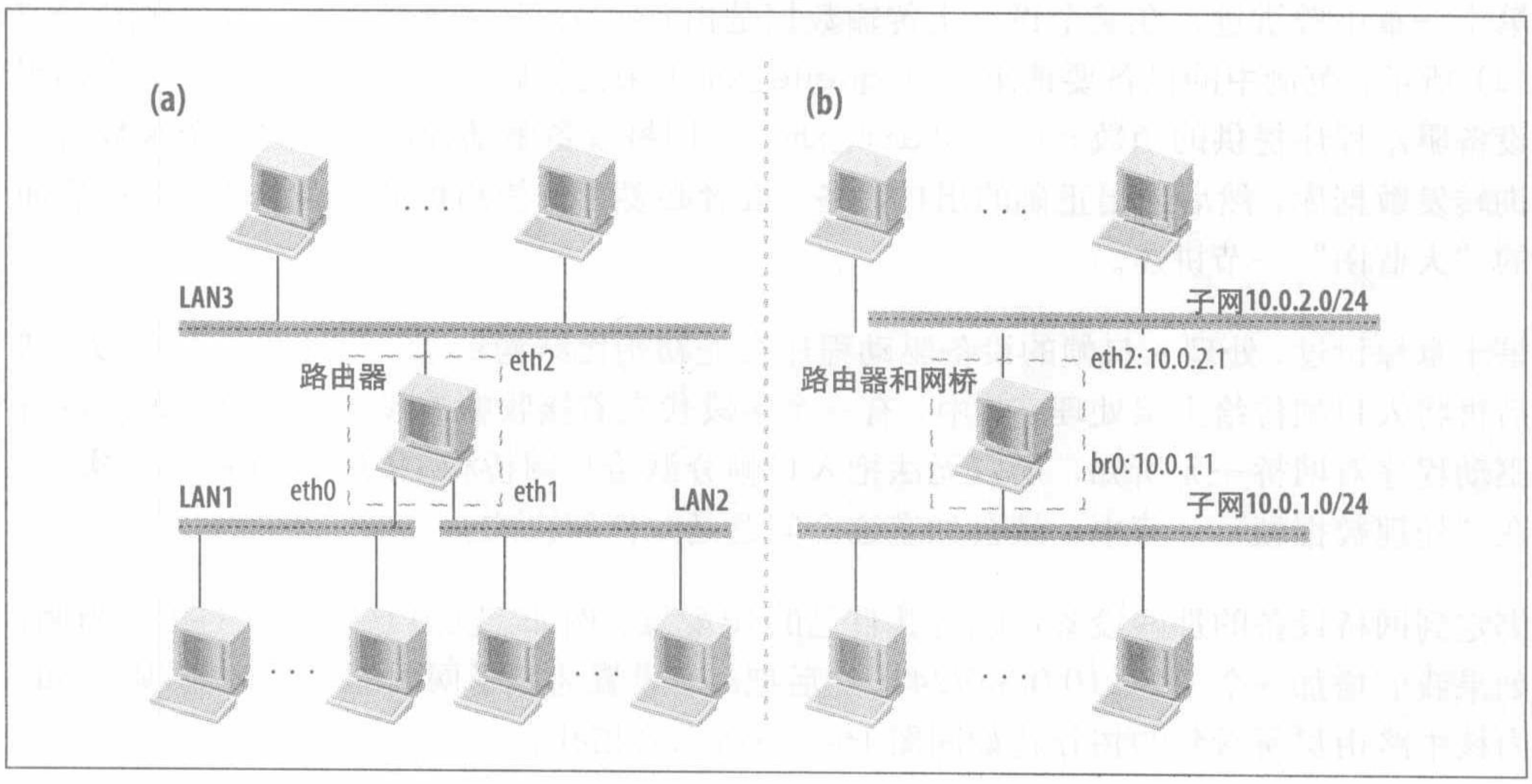


图 16-2：网桥设备抽象

现在，内核必须能够处理下面两个问题：

- 在路由器层次上，虽然有三个接口（*eth0*、*eth1* 和 *eth2*），但内核只看见两个子网（10.0.1.0/24 和 10.0.2.0/24）。
- 内核只会在*eth0*和*eth1*间使用桥接，并且认为这两个接口配置在相同的IP子网内。

抽象化的网桥设备可以将这两个问题独立而优雅地处理掉。

当创建一个网桥设备时，必须告诉内核它要绑定到哪个接口上。换言之，你必须告诉内核哪些接口需要进行桥接。根据前面的例子，我们会建立一个网桥设备，称之为 *br0*，然后把 *eth0* 和 *eth1* 指派给 *br0*。因为 *eth0* 和 *eth1* 都是网桥接口，它们不需要配置任何 IP（它们根本不需要在 L3 层中被看见），就像图 16-1 中的网桥接口都没有 IP。你也可以把图 16-1 中所示的路由器和网桥间的连接所具有的 IP 信息指派给网桥设备。其结果如图 16-2（b）所示。

此时，路由子系统会根据在 *eth2* 和 *br0* 上的子网配置信息进行路由选择。当 *br0* 上有数据要传输时，网桥设备驱动程序会操纵被绑定的设备，并按照我们在第十四章所说的逻辑传输数据。如果转发数据库知道目的MAC地址，以太网帧就只会从正确的网桥端口传输；否则，就会扩散到网桥设备的所有端口。

第十一章中曾讲过，在某个设备上传输数据是由 `dev_queue_xmit` 执行的。如图 16-3（a）所示，范例中的设备要请求 `dev_queue_xmit` 来传输帧。`dev_queue_xmit` 会调用设备驱动程序提供的函数 `hard_start_xmit`。网桥设备驱动程序使用的这个函数会查询转发数据库，然后选出正确的出口设备，或者必要时会使用扩散。详细过程会在后面的“大蓝图”一节讲述。

第十章探讨过，处理入口帧的设备驱动程序会先初始化结构 `sk_buff` 中的一些字段，然后再将入口帧传给上层处理。其中，有一个字段代表着接收帧的设备。但由于NIC设备驱动程序对网桥一无所知，所以无法把入口帧分派给与网桥相关的 `net_device` 实例。在“处理数据帧”一节中，就会知道这个问题是如何解决的。

绑定到网桥设备的那些设备可以有其自己的IP配置。例如，以图 16-2 所示的拓扑为例，如果我们增加一个子网（10.0.3.0/24），然后把*eth0*设置为该子网内的一个地址，则Linux内核中路由层所看到的拓扑就如同图 16-4 中所示的拓扑。

因此，*eth0* 可以接收要传给 *br0* 网桥设备或其本身的网络数据。也就是，如图 16-3 所示的模型变成了如图 16-5 所示的模型。

虽然传输部分在程序代码中不需费神，但是，接收部分就另当别论了。默认情况下，被绑定的设备所接收的网络流量会分配给它的指定网桥设备。例如，在图 16-5 中，*eth1*

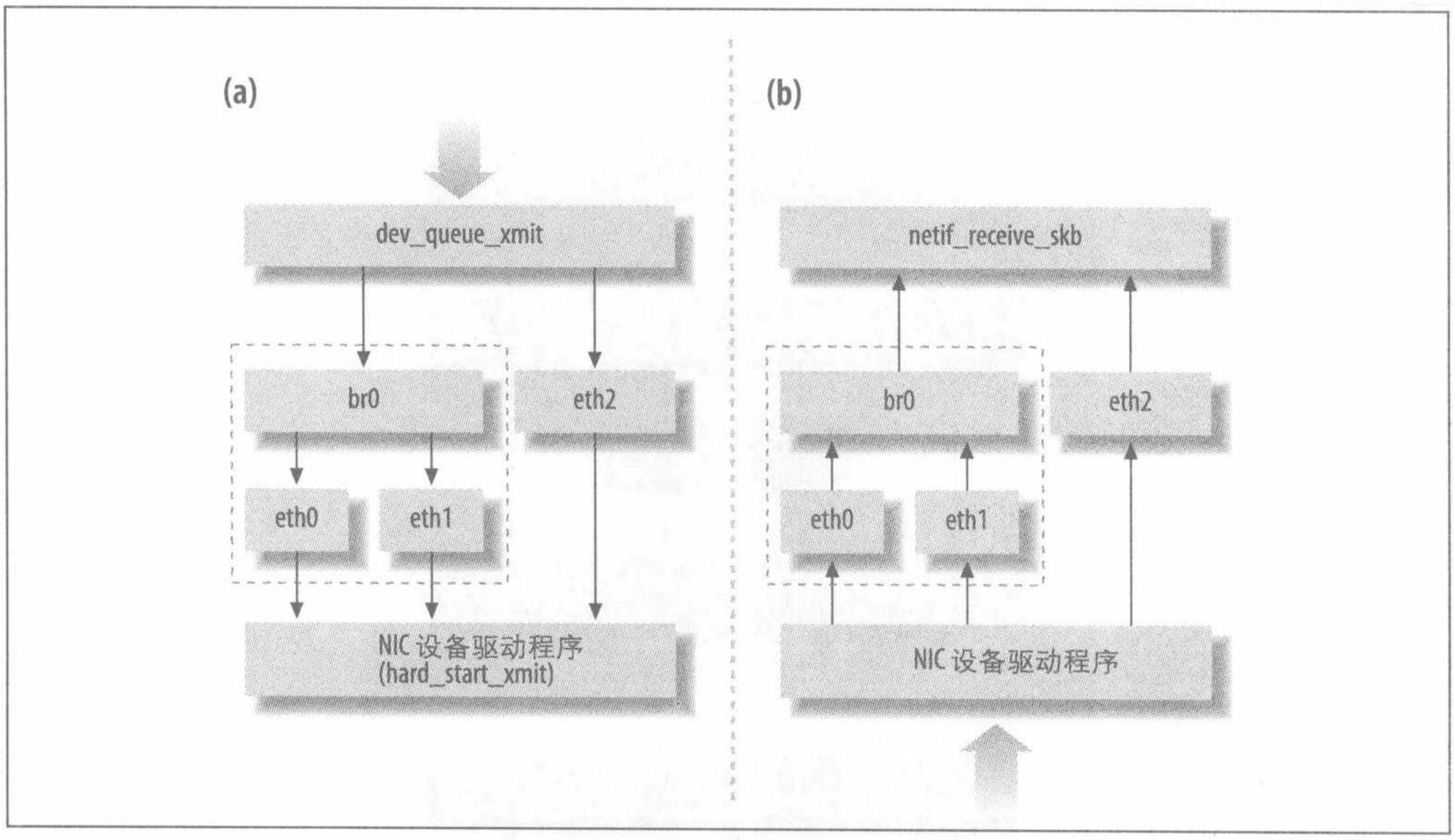


图 16-3：（a）在网桥设备上的传输；（b）在网桥设备上的接收

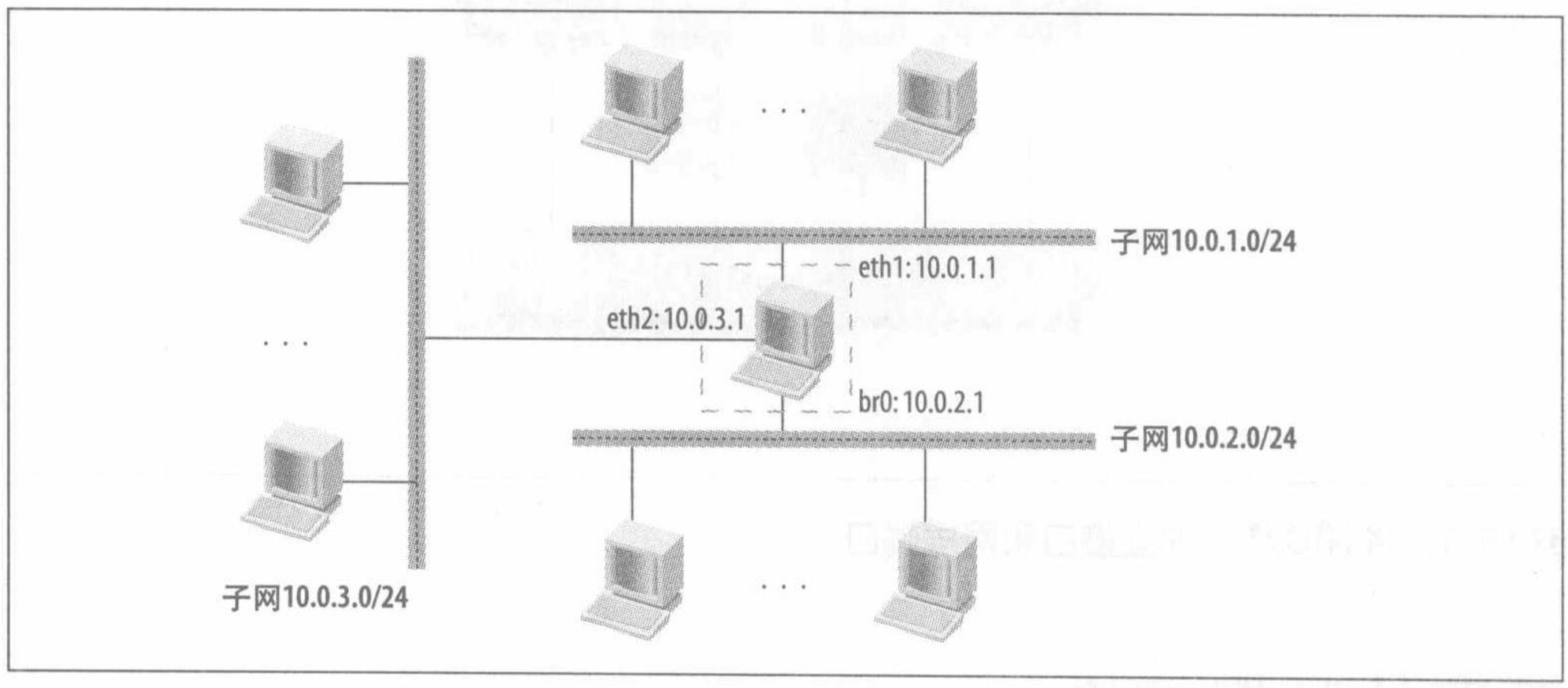


图 16-4：给被绑定的 NIC 指定 L3 配置信息

接收帧会分配给 *br0*。一个入口帧究竟是要进行桥接还是路由（也就是前一例中决定要把入口帧交给 *eth0* 还是 *br0*），可以通过 ebtables 进行配置（参见“数据帧和 BPDU”一节）。

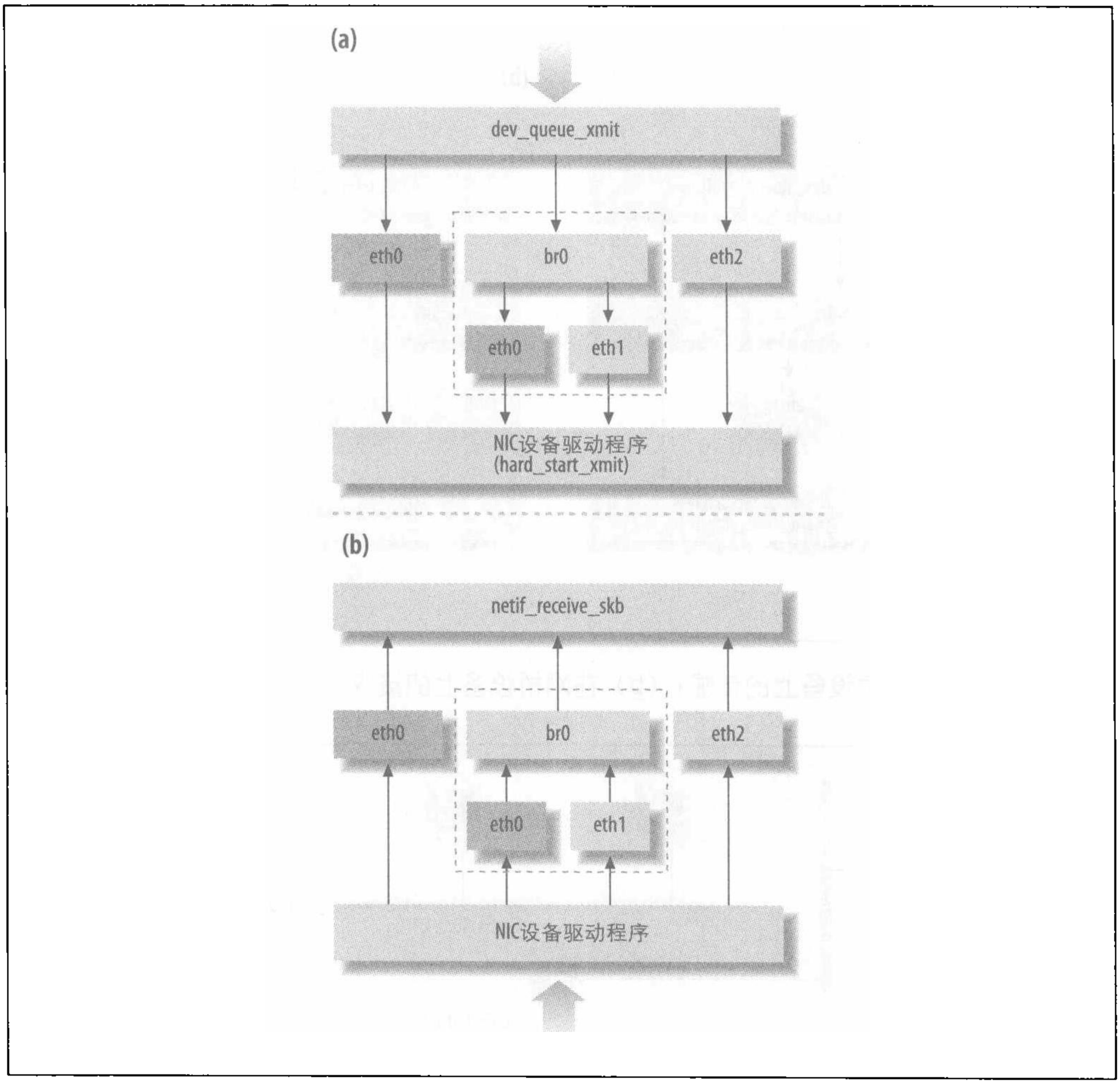


图 16-5：将 NIC 作为独立接口和网桥端口

重要的数据结构

下面列出了内核桥接模块代码所定义和使用的主要数据结构。第十七章中的部分小节将专门介绍这些数据结构及其字段。

`mac_addr`

MAC 地址。

bridge_id

网桥 ID（第十五章中有其定义）。

net_bridge_fdb_entry

转发数据库的记录项。网桥所学到的每个 MAC 地址都有这样一个记录。

net_bridge_port

网桥端口。

net_bridge

应用到单个网桥的信息。如图 16-6 所示，该结构会附加到 net_device 数据结构之上。和大多数虚拟设备一样，其内部包括只有虚拟设备程序（这里就是指桥接代码）才能理解的私有信息。

br_config_bpdu

入口配置 BPDU 的一些关键字段会复制到该数据结构中，此结构会将这些字段传递给处理配置 BPDU 的函数，而不是由原 BPDU 来传递。

所有数据结构都定义在 *net/bridge/br_private.h* 中 —— 除了 br_config_bpdu，它定义在 *net/bridge/br_private_stp.h* 中。图 16-6 所示为部分数据结构之间的关系。此图并没有反映出上一节所讲的配置范例。

age_listt 列表已不再使用；将其引入图中只是作为参考。参见第十七章“net_bridge 结构”一节。

桥接程序的初始化

桥接程序既可以集成在内核中，也可以编译成独立模块。初始化函数 br_init 和清理函数 br_uninit 的定义在 */net/bridge/br.c* 中。

初始化工作包括：

- 转发数据库初始化，就是在内存中建立一块 slab cache，以存放 net_bridge_fdb_entry 结构（br_fdb_init）。
- 初始化函数指针 br_ioctl_hook 为处理 ioctl 命令的函数。ioctl 命令会在第十七章说明。
- 初始化函数指针 br_handle_frame_hook，使其指向处理入口的 BPDU 的函数。参见“处理入口流量”一节。
- 用 netdev_chain 通知链注册一个回调函数。见“netdevice 通知链”一节。

当内核被编译成支持Bridging-Firewalling时，br_netfilter_init函数就会在此时初始化Bridging-Firewalling。稍后，在“大蓝图”一节的图16-11中，你会知道所有Netfilter钩子（hook）在桥接程序用于处理入口和出口网络流量的主要函数中的位置。

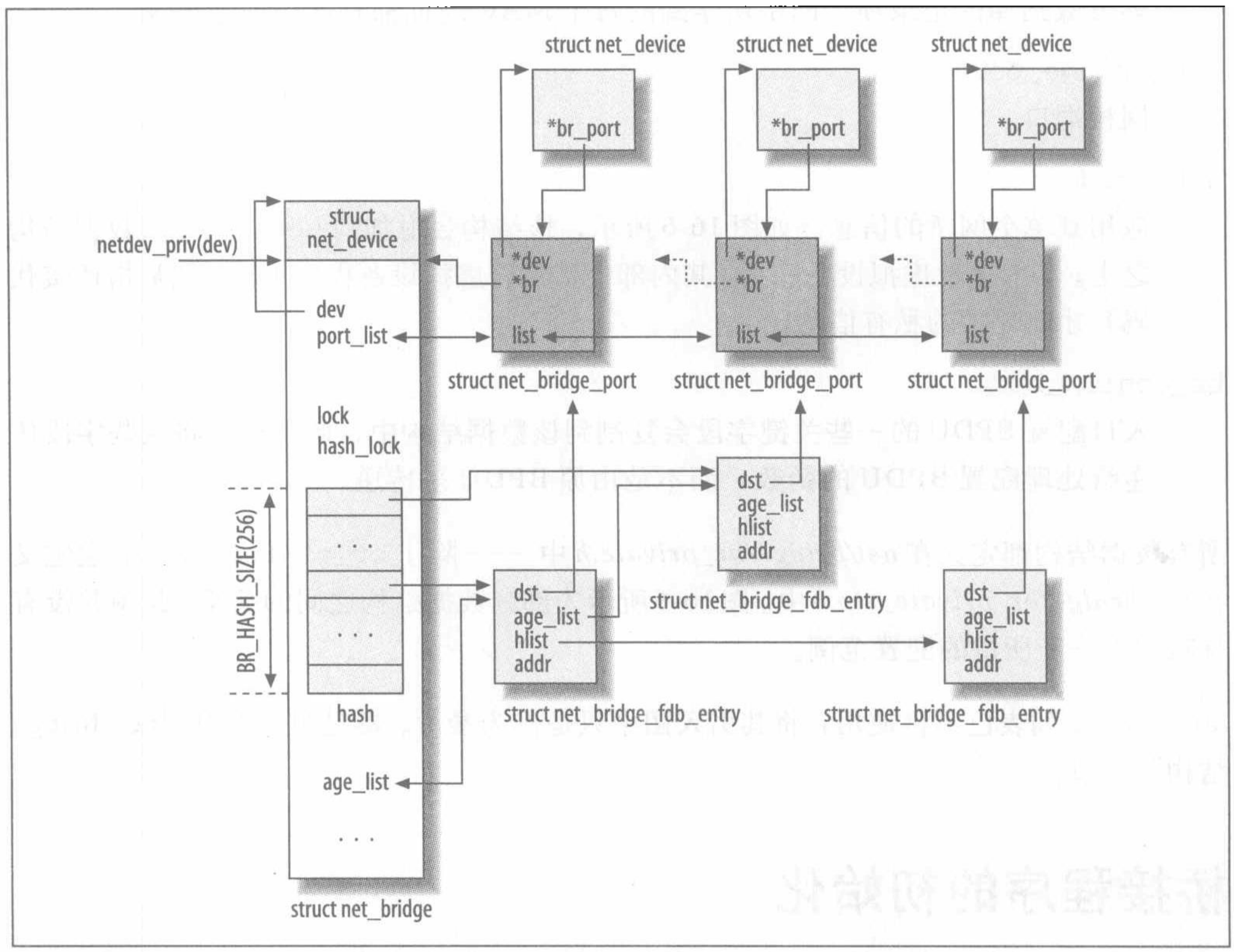


图16-6：主要数据结构之间的联系

在编译内核时，选中Networking support → Networking options → Network packet filtering （replaces ipchains） → Bridged IP/ARP packet filtering后，内核就支持Bridging-Firewalling。Ethernel-Bridging-Tables选项（也就是ebtables）是在其他地方进行初始化（参见“数据帧和BPDU”一节）。

清理函数br_deinit只是把br_init所做的事撤销而已。

建立网桥设备和网桥端口

Linux并没有明确地限制管理员能够创建的网桥设备数目。每个网桥设备最多可以有BR_MAX_PORTS（1,024）个端口。

网桥设备的建立和删除分别是通过br_add_bridge函数和br_del_bridge函数进行。

给网桥设备添加端口或删除端口是通过 br_add_if 函数和 br_del_if 函数进行。

这四个函数运行时，都必须持有 Netlink routing 锁。该锁是通过 rtnl_lock 函数取得的，并使用 rtnl_unlock 函数释放。br_add_bridge 和 br_del_bridge 各自自动锁定，而 br_add_if 和 br_del_if 则是由 dev_ioctl 函数控制锁定（见第十七章）。

这四个以 br_ 开头的函数在 *net/bridge/br_if.c* 中定义。到了第十七章，就会知道这几个函数在响应用户空间的配置命令时是如何调用的。

建立一个新网桥设备

虽然网桥设备是虚拟设备，但第八章中讲解的有关设备的启动、关闭、注册以及注销的内容依然适用。阅读这一节时，可以参考该章。

网桥设备的建立和注册遵循第八章讲述的模型。唯一的区别在于网桥设备是虚拟设备，因此网桥需要在其私有区域内（也就是图 16-6 的底端的 net_bridge 数据结构）多一些额外的初始化工作。这个任务由 new_bridge_dev 函数完成，包括：

- 分配一个 net_device 数据结构并初始化（用 br_dev_setup 作为设置函数），见“网桥设备设置函数”一节。
- 初始化私有数据结构，如图 16-6 所示。
- 初始化网桥优先权为默认值：32,768（0x8000）。
- 初始化指定网桥 ID（设为其识别码）、根路径开销（设为 0）以及根端口（设为 0，也就是没有根端口）。这样的设置是因为当一台网桥首次加电时，会认为它就是根网桥。参见第十五章“选择根网桥”一节。
- 设老化时间的初值为默认的 5min。
- 用 br_stp_timer_init 函数初始化每个网桥的定时器。

无论该网桥是否启动 STP，都会对生成树的参数初始化。

可以给网桥设备指定任何名称，常用的有 *brN*（当 STP 关闭时）和 *stpN*（当 STP 开启时）。如果你定义了两个不使用 STP 的网桥，按惯例，名称为 *br1* 和 *br2*。但是，用你家的狗的名字也是可以的。

同其他任何网络设备类似，网桥会在 */sys/class/net/* 里设定一个目录。见第十七章“通过 /sys 文件系统调整网桥”一节。

网桥设备设置函数

有关设备驱动程序在初始化设备时该如何调用相关设置函数，可以参考第八章“设备类型初始化：xxx_setup函数”一节的内容。网桥设备使用br_dev_setup设置函数。以下代码显示的就是该函数最有趣的部分：

```
void br_dev_setup(struct net_device *dev)
{
    memset(dev->dev_addr, 0 , ETH_ALEN);
    ether_setup(dev);
    ...
    dev->do_ioctl = br_dev_ioctl;
    dev->hard_start_xmit = br_dev_xmit;
    dev->open = br_dev_open;
    dev->change_mtu = br_change_mtu;
    dev->stop = br_dev_stop;
    dev->tx_queue_len = 0 ;
    dev->set_mac_addr = NULL ;
    dev->priv_flags = IFF_EBRIDGE;
}
```

网桥设备默认是没有实现队列机制，而是让被绑定的设备负责实现，这一点正好说明为什么tx_queue_len值被初始化为0。但是，管理员可以通过 *ifconfig* 或 *ip link* 命令配置tx_queue_len。

当网桥设备上的MTU改变时，内核必须确保新MTU值不会大于那些被绑定的设备中最小的MTU值。这一点由br_change_mtu来确保。

网桥MAC地址dev_addr会被清除掉，因为这个地址将由br_stp_recalculate_bridge_id函数从其绑定的设备上配置的MAC地址获得（参见“网桥ID和端口ID”一节）。基于同样的理由，驱动程序没有提供set_mac_addr函数。

必要时，还会设定IFF_EBRIDGE标记，使得内核可以区别网桥设备和其他类型的设备。

在网桥设备上发出的ioctl命令使用br_dev_ioctl函数处理，见第十七章。

第十一章讲过驱动程序会把hard_start_xmit函数指针初始化为设备用于传输数据的函数。网桥驱动程序将该指针初始化为br_dev_xmit。该函数负责实现“网桥设备抽象”一节所讲的网桥设备抽象层。本章后面的图16-11会显示出该抽象层是如何实现的。

当网桥设备因管理手段被开启或关闭时，内核会分别通过dev_open和dev_close调用br_dev_open和br_dev_close来启动和关闭网桥。见“启动和关闭网桥设备”一节。

删除网桥

删除网桥设备前，必须先将其关闭。如果没关闭，br_del_bridge返回 -EBUSY，并拒绝删除该设备。在删除时，br_del_bridge会调用 del_br 去做如下工作：

- 删除网桥设备的全部端口。对于每个网桥端口来说，也会删除*/sys*目录中相关的链接（这些链接是目录）。见“删除网桥端口”一节。
- br_fdb_delete_by_port函数会删除每个端口在转发数据库中所有相关的数据项，停止该端口的所有定时器，然后将promiscuity 计数器减 1（promiscuity 计数器会在后一节“给网桥添加端口”中说明）。
- 停止垃圾收集定时器br->gc_timer。
- 用br_sysfs_delbr删除*/sys/class/net*目录中的网桥设备子目录。
- 用 unregister_netdevice从内核注销该设备。该函数在第八章曾讲过。

给网桥添加端口

在当前的桥接实现中，NIC和网桥端口是一一对应的，如图 16-6所示。有些商用网桥允许管理员让多个网桥设备共享一块NIC，然后根据用户所选的规则把网络流量指派给特定网桥，但是，Linux 并没有这样做。

br_add_if 函数可以给一台网桥设备添加端口。该函数内部结构如图 16-7 所示。它并不关心网桥设备上是否开启了 STP。

br_add_if 函数一开始会先做合理性检查。如果下列任一条件满足，程序就中断执行：

- 要添加端口的设备不是 Ethernet 设备（或回环设备）。
- 若要添加端口的设备是网桥，如图 16-6 所示，网桥端口必须指定到真实设备（或者不是网桥设备的虚拟设备）上。
- 该网桥端口已指派给一个设备（即 dev->br_port 不为 NULL）。

当这些检查通过后，就可分配新网桥端口，并用 new_nbp 做部分初始化。初始化包括：

- 给该网桥端口指派一个端口号。见“网桥 ID 和端口 ID”一节。
- 给该端口指派默认优先权。
- br_make_port_id 函数结合端口号和优先权计算端口 ID。16 位的端口 ID 中，10

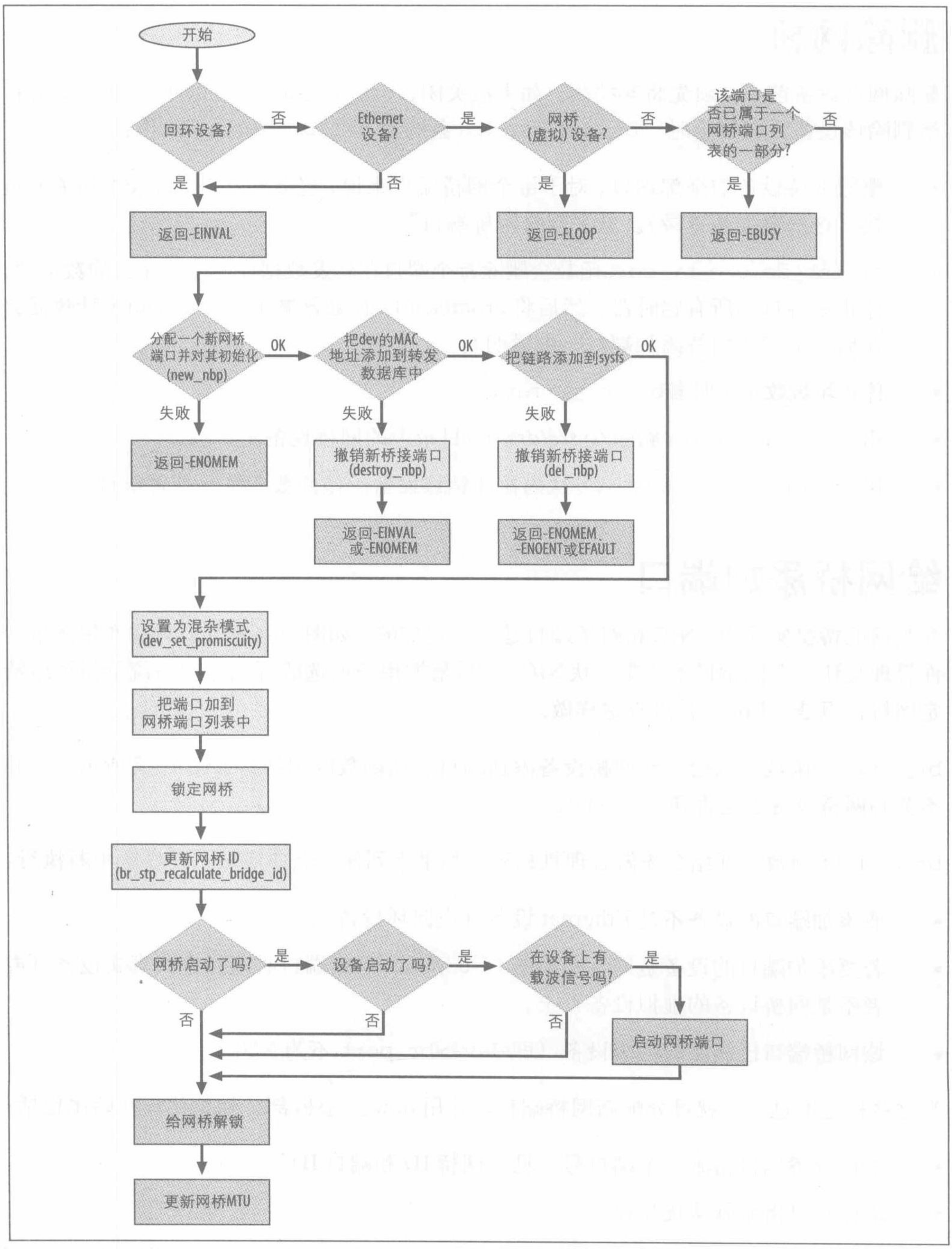


图 16-7：br_add_if 函数

(BR_PORT_BITS) 位由端口号使用，6 位由端口优先权使用。这一点和第十五章“网桥和端口 ID”一节所描述的标准中的表述并不相符。

- 根据被绑定设备的传输速率给端口设置默认开销。由 br_initial_port_cost 选择开销（看一看 new_npb 是如何被调用的），也就是通过第八章介绍的 ethtool 接口读取设备的传输速率，然后将其转变为开销值。当被绑定的设备的驱动程序不支持 ethtool 接口时，就无法从该接口导出默认开销，因此，开销的选择是基于被绑定的设备是 Ethernet 10 Mbit/s 设备的假设。在 IEEE 802.1D 协议规范中规定了端口传输速率和默认端口开销间的联系。
- 指定 BR_STATE_DISABLED 初始态。
- 将网桥端口连接到被绑定的设备和网桥设备，如图 16-6 所示。

与这个新端口相关的设备的 MAC 地址会被 br_fdb_insert 添加到转发数据库中。

br_sysfs_addif 函数会在 */sys* 目录中添加些必要的链接，如第十七章“通过 /sys 文件系统调整网桥”一节所述。

与该网桥端口相关的 NIC 会通过 dev_set_promiscuity 函数进入混杂模式。混杂模式可以捕获所有 LAN 网络数据，而且当网桥转发帧时，也需要进入混杂模式。对每个网桥端口，都有一个计数器而不是一个布尔标志来保存该模式，这样内核就能够处理进入混杂模式的嵌套请求（不断进入混杂模式）。当一个网桥端口上进入混杂模式时（由 dev_set_promiscuity 实现），在相关绑定的设备上的计数器就会递增；当混杂模式关闭时，该计数器就会递减。

最后，这个新的端口被加入到网桥的端口列表中，如图 16-6 所示。此外，网桥 ID 和 MTU 也会根据第十五章“网桥 ID 和端口 ID”和本章的“网桥设备设置函数”两节所述的规则分别用 br_stp_recalculate_bridge_id 函数和 dev_set_mtu 函数进行更新。

删除网桥端口

删除网桥端口时，主要是把建立端口时所做的事撤销。图 16-8 所示为 br_del_if 函数的内部流程。

启动和关闭网桥设备

由“网桥设备设置函数”一节已知道 dev->open 和 dev->stop 如何对网桥设备初始化，此外，第八章“开启和关闭网络设备”一节也讲述了如何用 dev_open 和 dev_close 处理启动和关闭网络设备的管理命令。

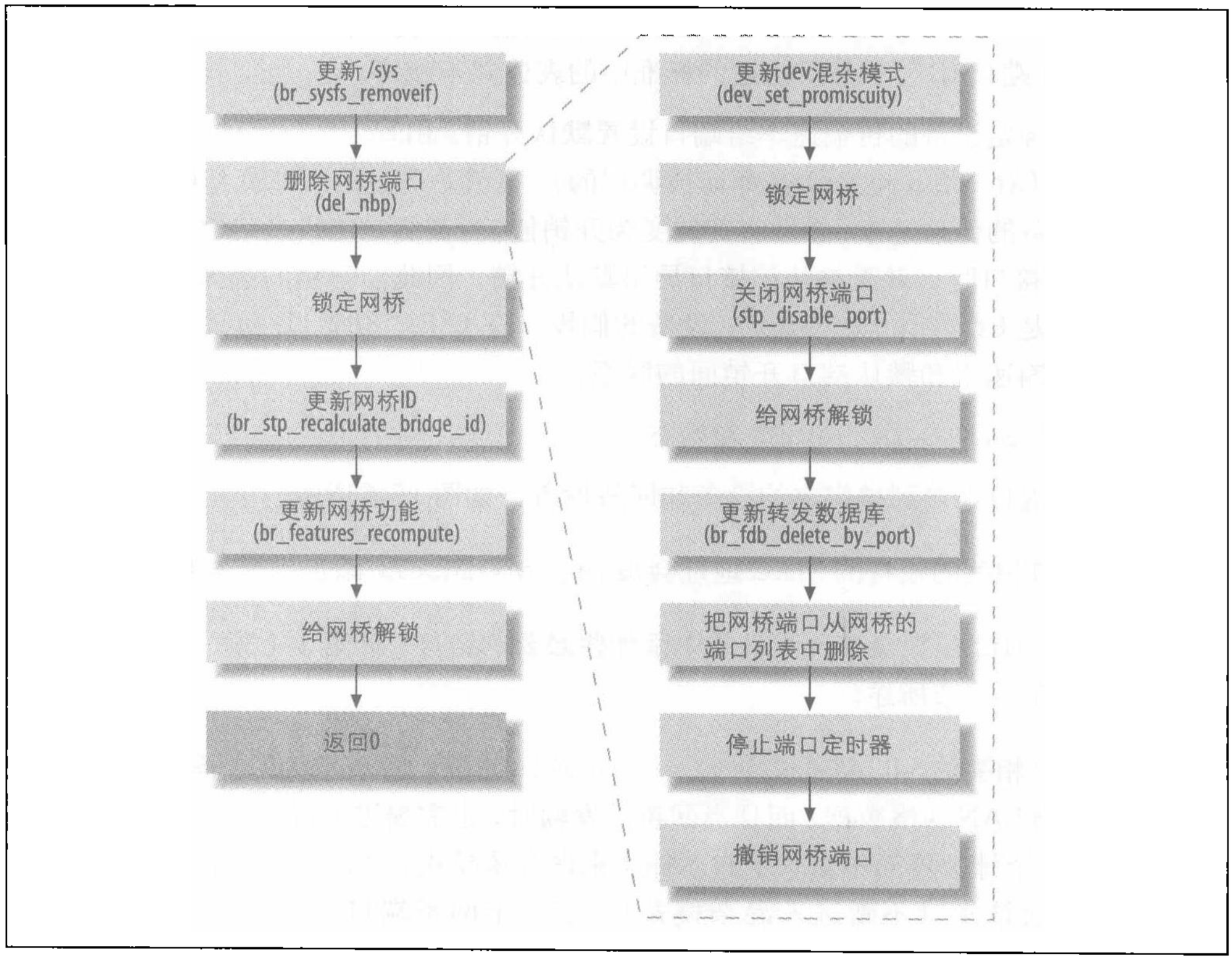


图 16-8：br_del_if 函数

br_dev_open 函数启动网桥的步骤如下：

1. br_features_recomputed将网桥设备的基本特征初始化为其绑定的设备所支持的功能的最小常用子集。
2. 用netif_start_queue函数启动设备进行数据传输（见第十一章“传输的开启和关闭”一节）。
3. 用 br_stp_enable_bridge 函数启动网桥设备。

当启动网桥设备时，先前绑定到该设备上的端口也会跟着启动。

如图 16-9 所示，br_dev_stop 只是 br_dev_open 的镜像（mirror image）而已。

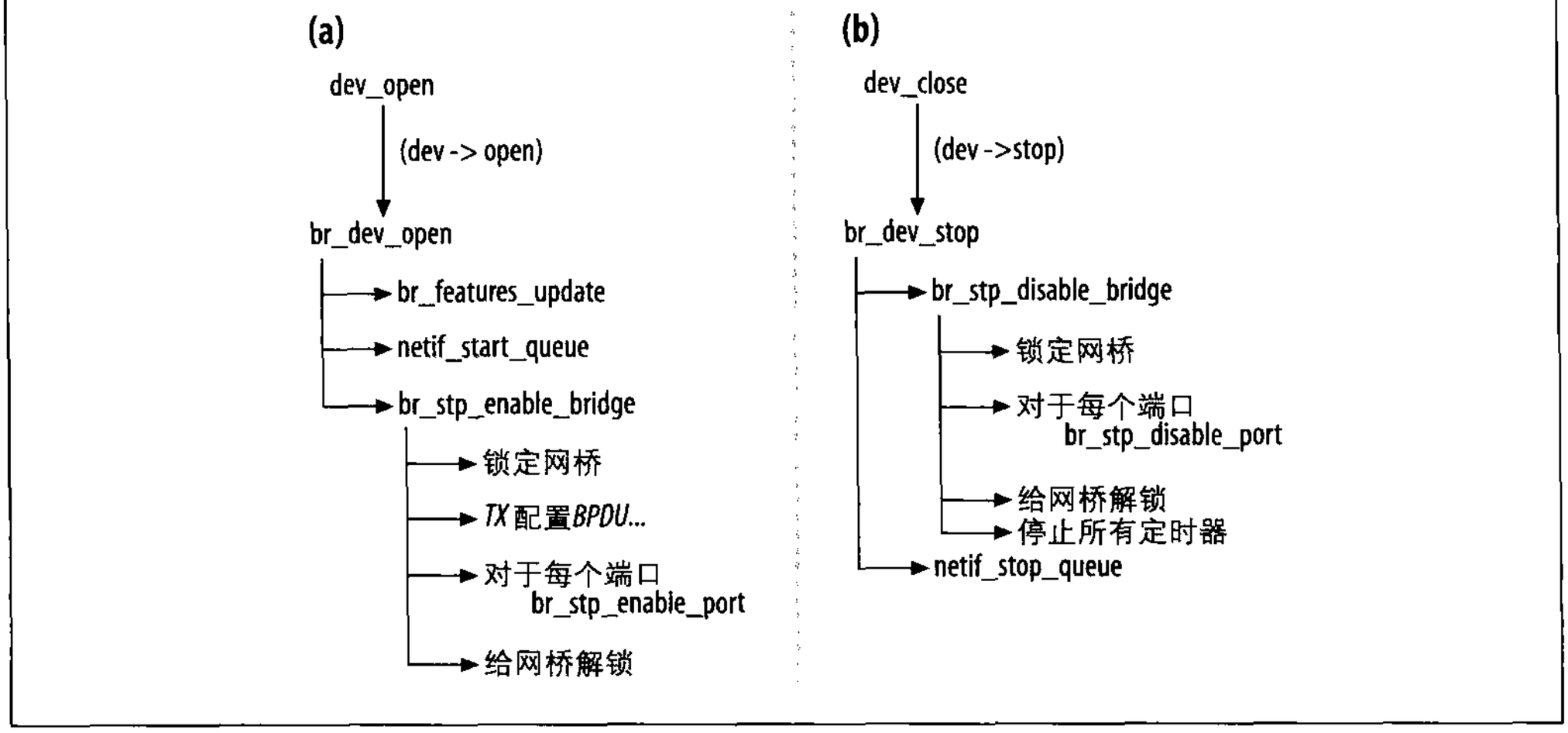


图 16-9：(a) 启动网桥；(b) 关闭网桥

启动和关闭网桥端口

网桥端口分别用 br_stp_enable_port 和 br_stp_disable_port 来启动和关闭。

要启用网桥端口，必须满足下列所有条件：

- 被绑定的相关设备已用管理手段启动。
- 被绑定的相关设备有载波状态。参见第八章“链接状态变更侦测”一节有关 Linux 如何检测载波信号状态变化的描述。
- 相关的网桥设备已用管理手段启动。

注意：网桥设备上没有载波状态，因为网桥是虚拟设备。

当网桥端口是以用户空间命令建立并且先前三项条件都满足时，该网桥端口就可立即启用了。参见“给网桥添加端口”一节。

假设当端口建立时，由于上述三项必要条件中至少有一项不满足，因而无法启用端口。下面是每项条件最终满足时启用端口的场合：

- 当被关闭的网桥设备重新启动时，其所有关闭的端口就会启用。
- 当被绑定的设备检测到载波状态时，桥接程序会收到 NETDEV_CHANGE 通知消息。参见“netdevice 通知链”一节。

- 当被关掉的绑定设备重新启动时，桥接程序会收到NETDEV_UP通知消息。参见“netdevice通知链”一节。

当本节开头所列三项条件之一不再满足时，网桥端口就会被关闭。

图16-10总结了启动和关闭网桥端口的步骤和相关函数。注意，当网桥端口关闭时，非根网桥可能会变成根网桥。这种转变在“成为根网桥”一节会加以说明。

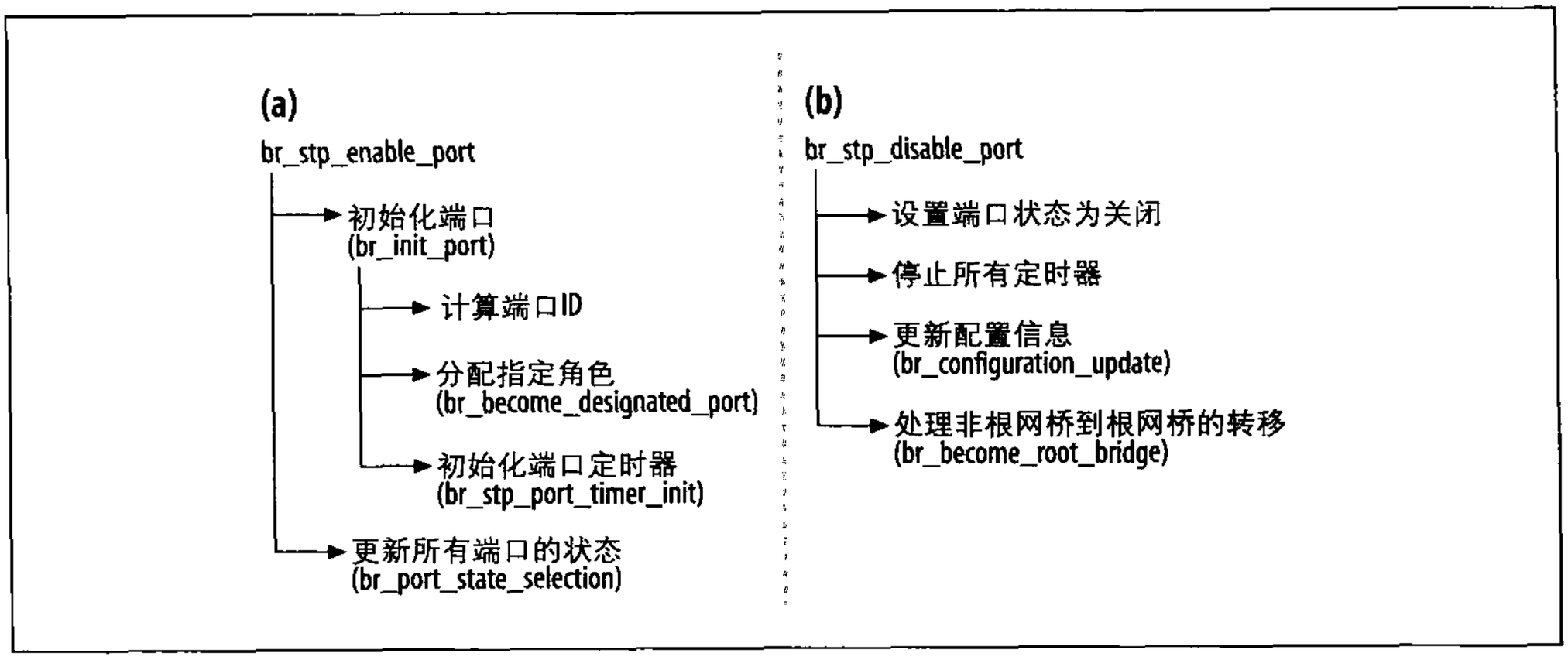


图16-10：（a）启动端口；（b）关闭端口

注意，当一个端口启动时，首先要做初始化，然后用br_port_state_selection函数为其指定合适的状态。该函数会遍历所有网桥端口，把适当状态应用到每个端口上。对于一台没有运行STP的网桥，该函数实际上就是把新端口置为BR_STATE_FORWARDING态。这是因为该端口被指派为指定角色（尽管不运行STP的网桥不在意端口角色）。应谨记，大多数函数无法区分STP是启动还是关闭的。例如，br_state_port_selection会遍历所有端口，因为当STP启动且经过配置更新时，有可能改变很多端口的角色和状态（见“配置更新”一节）。

改变网桥端口状态

网桥端口状态不是处于活跃（active）就是处于不活跃（inactive）：相关的状态是BR_STATE_FORWARDING或BR_STATE_BLOCKING。不过，虽然BR_STATE_ BLOCKING态可以被立刻指派给端口，但是，只有在经历第十五章“端口状态”一节所介绍的中间状态后，才能转变为BR_STATE_FORWARDING态。

BR_STATE_FORWARDING态和BR_STATE_BLOCKING态的指派分别是通过

br_make_forwarding 函数和 br_make_blocking 函数进行的。使用这两个函数时不用管端口所属的网桥设备是否运行 STP。

```
static void br_make_blocking(struct net_bridge_port *p)
{
    if (p->state != BR_STATE_DISABLED &&
        p->state != BR_STATE_BLOCKING) {
        if (p->state == BR_STATE_FORWARDING ||
            p->state == BR_STATE_LEARNING)
            br_topology_change_detection(p->br);
        p->state = BR_STATE_BLOCKING;
        br_log_state(p);
        del_timer(&p->forward_delay_timer);
    }
}
static void br_make_forwarding(struct net_bridge_port *p)
{
    if (p->state == BR_STATE_BLOCKING) {
        if (p->br->stp_enabled) {
            p->state = BR_STATE_LISTENING;
        } else {
            p->state = BR_STATE_LEARNING;
        }
        br_log_state(p);
        mod_timer(&p->forward_delay_timer, jiffies + p->br->forward_delay);
    }
}
```

不能把 BR_STATE_BLOCKING 和 BR_STATE_FORWARDING 间的任何中间状态指派给端口，这也是为什么如果要将不是处于 BR_STATE_BLOCKING 态的端口改变为 BR_STATE_FORWARDING 态时，br_make_forwarding 函数就不执行任何动作的原因。中间状态就预示着该端口已在前往 BR_STATE_FORWARDING 态的路上，且在适当的定时器到期时就会到达那个状态。

在 br_make_forwarding 函数中，当没启用 STP 时，网桥端口就会跳过 BR_STATE_LISTENING 态。当没启用 STP 时，所有网桥端口都会指派为转发状态；因此，也可以跳过 BR_STATE_LEARNING 态。但是，使用中间状态 BR_STATE_LEARNING 还可以让网桥学习一些MAC地址，并减少扩散的数量（转发数据库为空时就需要扩散）。

大蓝图

图 16-11 所示是桥接程序用于处理进入口帧和出口帧（包括数据帧和 BPDU）的一些重要函数。

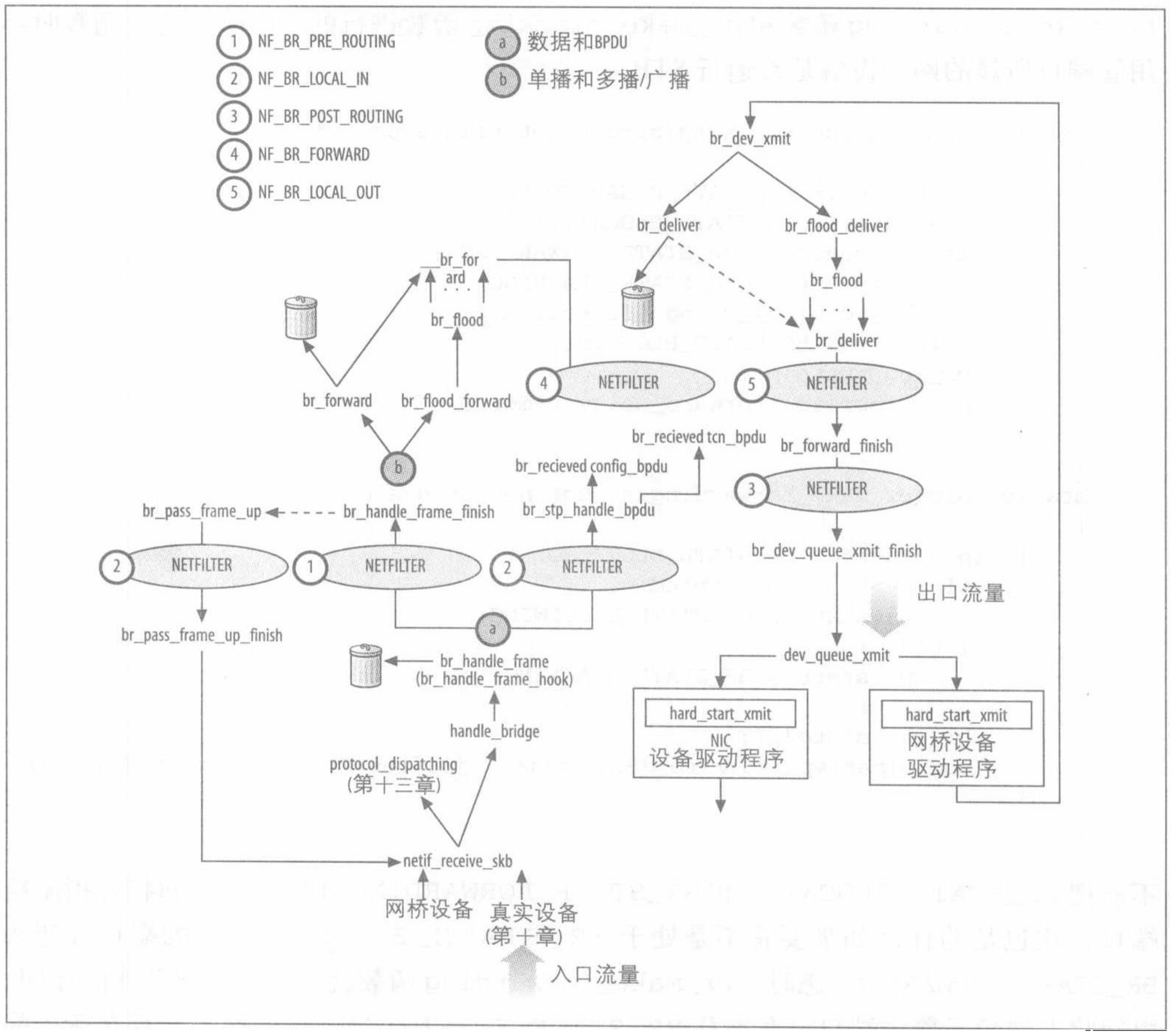


图 16-11：大蓝图

特别要注意的是：

- IP层里也有很多钩子函数（参见第十八章的图18-1所示）。在br_handle_frame(NF_BR_BROUTING)函数有多个钩子是由ebtables命令使用的，图中没有给出，参见“数据帧和BPDU”一节说明。
- 入口数据帧可能会两次通过netif_receive_skb函数。关于netif_receive_skb函数在第十章曾介绍过。也可参考本章后面的“处理数据帧”一节。
- 当对handle_bridge的调用表明入口帧必须传递给桥接子系统时，netif_receive_skb函数就负责传递；否则，该函数就把帧传给上层协议处理函数（如第十三章所述）。

- 入口帧也可能会被桥接程序丢弃，例如，当端口已关闭时。
- 若进行接收的端口已被STP阻塞，入口数据帧就会被br_forward丢弃。若出口帧的目的地址是主机本地地址，br_deliver就不必把它从任何端口传输出去。在这两种情况下，should_deliver会过滤掉不必要的传输。
- 出口数据帧会两次通过函数dev_queue_xmit。关于dev_queue_xmit函数已在第十一章描述过。另外也可参考本章稍后的“网桥设备上的传输”一节。
- br_flood函数能在一台网桥的端口上扩散一个帧。扩散对入口帧和出口帧都可能是必要的。无论一个帧来自何处，当其要发给一个多播或广播地址时，或者要传给一个不在转发数据库内的地址时，就必须进行扩散。br_flood函数可从其最终的输入参数得知其所处理的是入口帧还是出口帧，并多次调用相关函数在每个网桥端口上送出这些帧(入口帧用_ _br_forward函数，出口帧用_ _br_deliver函数)。

在阅读后续几节的程序时要记住：无论 STP 是否启动，桥接程序都使用同一组核心函数。一些重要差异在子任务的执行中才会显示出来。

当开启 STP 时：

- 处理入口 BPDU。
- BPDU 也可能是本地产生的（取决于本地网桥端口的角色）。
- 入口网络数据不是转发到正确端口就是扩散到所有端口（根据第十四章所述的规则）。
- STP 阻塞的端口不能用于接收和传输数据流量。

当 STP 关闭时：

- 入口 BPDU 会被认为是数据流量。
- 不会产生本地 BPDU。
- 入口数据流量依然转发至正确端口或者扩散到所有端口（根据第十四章所述的规则）。
- 所有网桥端口（除非用管理手段关闭）都可用于接收和传输数据流量。

转发数据库

每个网桥实例都有自己的转发数据库，无论STP是开启或关闭都会用到。我们在本章后面就会知道，何时该数据库会被查询或更新。首先来看一下它的实现方式以及操纵该数据库的核心函数。用于管理转发数据库的所有函数都在 *net/bridge/br_fdb.c* 中。

该数据库嵌入在net_bridge数据结构中，并被定义成一个hash表，如图16-6所示。对于网桥的任何端口上所学到的每个MAC地址，会为其在该数据库中插入一个net_bridge_fdb_entry数据结构的实例。

网桥转发数据库子系统是由br_fdb_init初始化的，它简单地创建br_fdb_cache缓存，用于分配net_bridge_fdb_entry实例。

分配工作是由fdb_create完成的，此外，它也会根据其输入参数初始化net_bridge_fdb_ entry的一些字段。

查询

转发数据库中的记录是通过MAC地址来标识的。查询这个表包括用br_mac_hash选出正确的hash表bucket，以及浏览bucket中的net_bridge_fdb_entry实例列表，找出一个和指定MAC地址相匹配的数据项。

主要的查询例程有两个：

fdb_find
: 就是对给定的MAC地址简单搜索net_bridge_fdb_entry，且它不能用于转发数据流量。它主要是被桥接管理函数使用。

_ _br_fdb_get
: 与fdb_find类似，由桥接程序调用来转发流量。它不考虑过期的数据项（见“老化”一节）。

对这两个函数来说，调用者必须在适当的时候上锁。

外部子系统想查询转发数据库时，可以使用br_fdb_get函数；而br_fdb_get函数是一个包裹函数，能够执行上锁和引用计数及调用_ _br_fdb_get的工作。但是，并不能直接调用br_fdb_get，而是通过br_fdb_get_hook函数调用，这个函数在br_init中初始化为指向br_fdb_get的指针。

引用计数

由于用br_fdb_get查询转发数据库的外部子系统很有可能会把结果先缓存起来，因此，引用计数是用于记录转发数据库中的数据项到何时仍然需要，以及何时可以被释放。每个数据项都会指定一个引用计数。当查询成功时，br_fdb_get都会将其增1。当调用函数不再需要引用查询结果时，就应该用br_fdb_put将其减1。当引用计数为0时，br_fdb_put就会释放net_bridge_fdb_entry。

增加、更新及删除数据项

转发数据库的数据添加和更新通过不同的函数实现（取决于MAC地址是和本地接口匹配还是和入口帧匹配）。

在创建一个网桥端口时，br_add_if通过br_fdb_insert函数把被绑定的设备的MAC地址添加到转发数据库。br_fdb_insert会忽略不应加到转发数据库的MAC地址，例如多播和广播地址。当新地址碰巧已在数据库中时，除非它对应于另一个本地接口（此时无需做任何更新），否则就会被替换掉。转发数据库中的本地MAC地址允许桥接程序把要提交给本地接口的入口帧传至本地，因此桥接程序并不关心本地MAC和哪个接口相匹配。重要的是数据库中至少要有一个数据项可告知桥接程序哪些网络流量要提交至本地。

对于可以添加到转发数据库的数据项的数目并没有硬性限制。因此，这可能会让系统遭遇DOS攻击，我们期望开发人员在未来会增加一个明确的限制。

当与一个网桥端口相关联的本地设备（因此其MAC地址会在转发数据库中）改变其MAC地址时（注1），对应的数据项会由br_fdb_change_addr进行更新（参见“netdevice通知链”一节）。因为可能有多个本地接口配置为相同的MAC地址（这种情况不常见），所以，br_fdb_change_addr在删除net_bridge_fdb_entry数据项前会检查同一台网桥的另一个端口是否也有相同的MAC地址：如果发现另一个端口也有相同的MAC地址，就会把那个数据项绑定到该端口对应的接口上。

通过入口帧学习到的MAC地址（如第十四章所述）会由br_fdb_update添加到数据库。当该地址已在数据库中时，如果需要则更新对入口端口（dst）的引用，并且更新最后一次更新（ageing_timer）的时间戳。

net_bridge_fdb_entry实例是由fdb_delete负责删除的。该函数绝不会被直接调用，而是通过诸如br_fdb_cleanup（下一节说明）和br_fdb_delete_by_port这类包裹函数调用。

老化

对于每个网桥实例来说，都有一个垃圾收集定时器（gc_timer）来定期扫描转发数据库，然后把过期的数据项删除。在网桥实例初始化时，该定时器会在br_stp_timer_init中被初始化，之后br_stp_enable_bridge启动网桥时，该定时器也就会启动。

注1：只有系统管理员可以改变一个接口的MAC地址，这需要明确的命令，诸如*ip link set eth0 address 00:20:ED:76:1E:12* 或 *ifconfig eth0 hw ether 00:20:ED:76:1E:12*。不过，改变接口的MAC地址的情况相当少见。

此定时器每十分之一秒到期一次，并调用br_fdb_cleanup做清理工作。该函数还会扫描数据库，用 fdb_delete删除过期的数据项。

如果一个数据项至少5分钟没被用过，通常就算过期了。但是当网桥运行STP时，一旦检测到拓扑变化，这个老化时间就会使用较短的forward_delay（以秒记）计时；参见第十五章“Short Aging定时器”一节。通过调用hold_time函数，就可以透明地使用正确的老化时间（该函数会按这里所述的流程返回正确的老化时间）。

处理入口流量

第十章已知道netif_receive_skb如何处理入口流量。我们也知道，在把每个入口帧传给上层协议处理函数前，该函数如何调用handle_bridge（定义在*net/core/dev.c*中）函数。

若内核不支持桥接，handle_bridge就被定义成NULL指针，并且netif_receive_skb会把入口帧交给其他协议处理。若内核支持桥接，当网桥端口上接收到帧时，handle_bridge就会用br_handle_frame_hook处理该帧。当初始化桥接模块时，br_handle_frame_hook的指针就初始化为br_handle_bridge。

```
#if defined(CONFIG_BRIDGE) || defined (CONFIG_BRIDGE_MODULE)
...
static _ _inline_ _ int handle_bridge(struct sk_buff **pskb,
                      struct packet_type **pt_prev, int *ret)
{
    struct net_bridge_port *port;

    if ((*pskb)->pkt_type == PACKET_LOOPBACK ||
        (port = rcu_dereference((*pskb)->dev->br_port)) == NULL)
        return 0;

    if (*pt_prev) {
        *ret = deliver_skb(*pskb, *pt_prev);
        *pt_prev = NULL;
    }

    return br_handle_frame_hook(port, pskb);
}
#else
#define handle_bridge(skb, pt_prev, ret)     (0)
#endif
```

下面几个小节中，我们会说明handle_bridge如何处理入口帧，区分数据帧和STP BPDU帧（图16-11中的圈a）。对于数据帧，该函数也会区分单播帧、多播帧以及广播帧（图16-11中的圈b）。

数据帧和 BPDU

在支持桥接的Linux 系统上，并非所有NIC都要配置为网桥端口。当一个NIC配置成网桥端口时，其net_device中的br_port指针会指向相关联的网桥端口。由于每个网桥端口都包含指向其所属网桥实例的指针，所以，可以轻易从任何真实设备中找出其所属的网桥实例（如果有的话），然后读取net_bridge数据结构中的一个标识，检查该设备是否启动 STP，见图 16-6 所示。

STP 所产生的 BPDU 和其他所有的入口帧有所不同，而且它是由 STP 接收函数来处理。但是，也只有当包含入口端口的网桥上启动了 STP 之后才是这样。

图 16-12 给出 br_handle_frame 函数如何把入口帧交给函数 br_handle_frame_finish 或 br_stp_handle_bpdu（取决于 STP 是否开启）。

已关闭的端口上接收的任何帧都会被丢弃。

只有处于BR_STATE_FORWARDING 态的端口才可接受数据帧，而只要 STP 启动（否则，BPDU 只会被视为普通数据帧），任何启用的端口都可接受 BPDU。

图 16-12 的左边用以识别 BPDU 的流程是按照第十五章“BPDU 封装”一节所介绍的规则实施的。

图16-12底端的两个函数只有在Netfilter因其他原因没有丢弃或消化该帧时才会被调用。

ebtables也能查看帧。ebtables是一个架构，能提供一些Netfilter所没有提供的额外功能。

尤其是，ebtables 可以过滤和修改任何类型的帧，而非仅限于那些携带 IP 封包的帧。就这里的讨论而言，我只涉及 ebtables 的两个功能：

- 允许定义规则 —— 根据一些因素，例如网络协议（如 IPv4）或目的 IP 地址 —— 然后通知内核，哪些网络流量要通过网桥，而哪些网络流量要通过路由选择。也就是说，被网桥端口绑定的NIC不仅作为网桥端口，也是以独立的L3 接口的形式存在，可以有其自己的 L3 配置信息。“网桥设备抽象”一节已见过这样的例子。
- 可以修改目的MAC地址，例如，把一个帧重定向至另一台主机，或者实现某种网络地址的翻译。这也就是为何在 ebtables 完成任务后，用 br_handle_frame 检查目的 MAC 地址的原因。

要让内核支持 ebtables，可在编译内核时选中选项 Networking support → Networking options→Network packet filtering (replaces ipchains)→Bridge: Netfilter Configuration

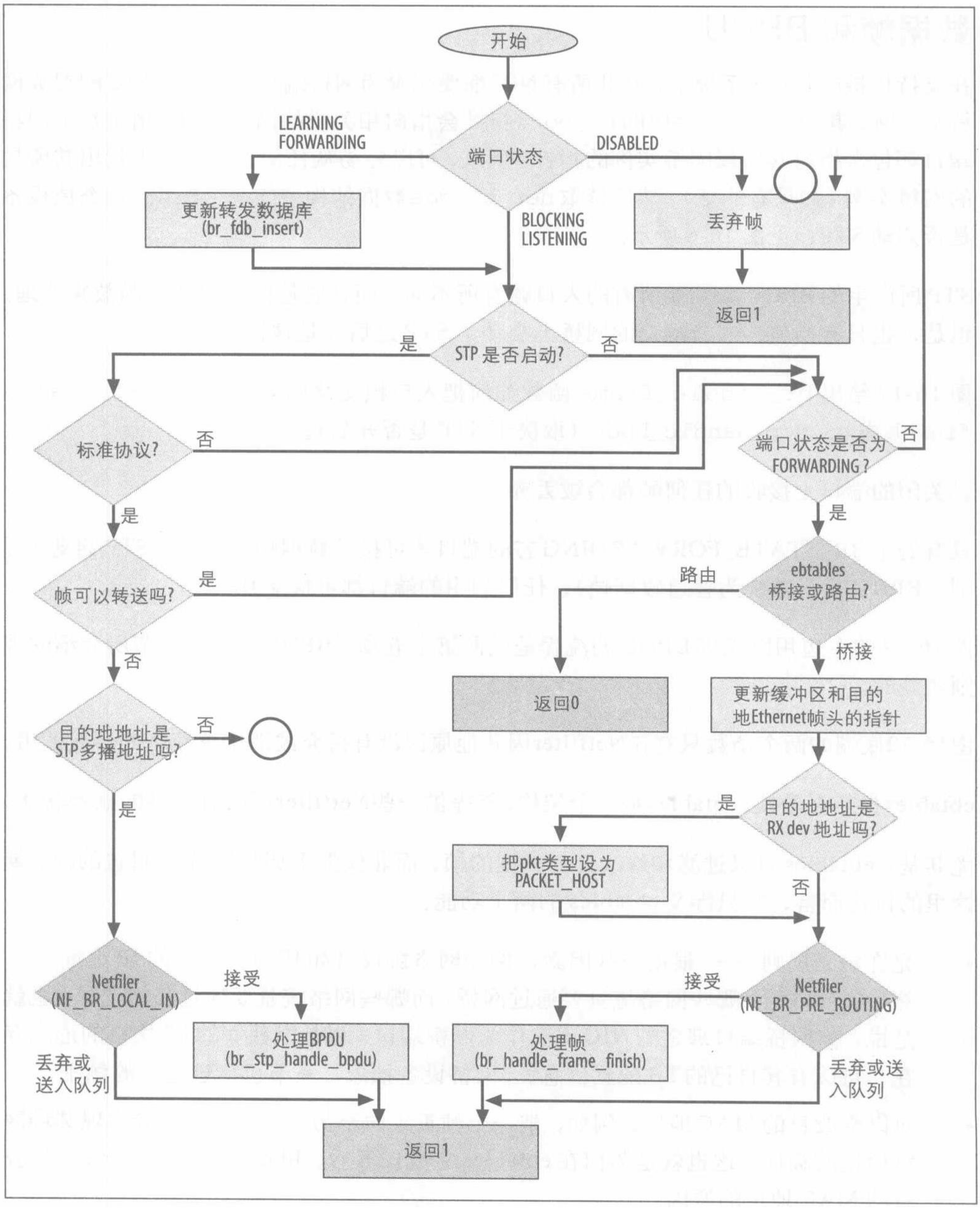


图 16-12：br_handle_frame 函数

Ethernet Bridge tables（ebtables）support。在 ebtables 主页上，可以找到有关该选项的很详细的说明文档（包括用户空间方面和内核方面），也可以找到说明如何使用

ebtables所提供的每项功能的清楚的例子。ebtables项目的主页在*http://ebtables.sourceforge.net*站点上。

处理数据帧

入口数据帧是由br_handle_frame_finish函数处理的，如图16-13所示。

首先，帧的源MAC地址会由br_fdb_update加入到转发数据库。然后，会在转发数据库中搜索目的MAC地址。如果找到该地址，帧就会由br_forward转发到正确网桥端口；否则，就会让br_flood_frame将帧扩散到网桥的所有转发端口。目的地是广播或多播链路层地址的帧总是会被扩散。

若下列条件之一满足时，帧的一个副本会由br_pass_frame_up（即传送帧到上层协议的函数）提交至本地：

- 网桥接口处于混杂模式。注意，所有网桥端口绑定的设备都会处于混杂模式，因为网桥运行必需此模式。但除非明确地对其进行配置，否则网桥自己是不会处在混杂模式的。
- 因前述理由之一而扩散帧。
- 转发数据库显示，目的MAC地址属于一个本地接口。

了解本地传送的原理也是很有趣的。图16-11的子集图16-14给出了入口帧最终传递至上层协议的正确路径。

当该帧由NIC的设备驱动程序接收到时，skb->dev就被初始化并指向真实设备。然后，该帧被向上送到网络堆栈，最终会传递到br_pass_frame_up函数。该函数越过另一个Netfilter钩子，然后调用br_pass_frame_up_finish。此时，skb->dev的值会被该入口端口所属的网桥设备替换掉，并且再次调用netif_receive_skb。这一次，handle_bridge看到该设备已不再是被绑定的设备（也就是br_port为NULL），因此会把该帧交给正确的协议处理函数，如同第十三章所述。

网桥设备上的传输

我们在“网桥设备抽象”一节已经知道，网桥设备抽象层要求把一台网桥设备上的传输转变成在一个或所有网桥端口上的传输。图16-11给出了相关的主要函数。

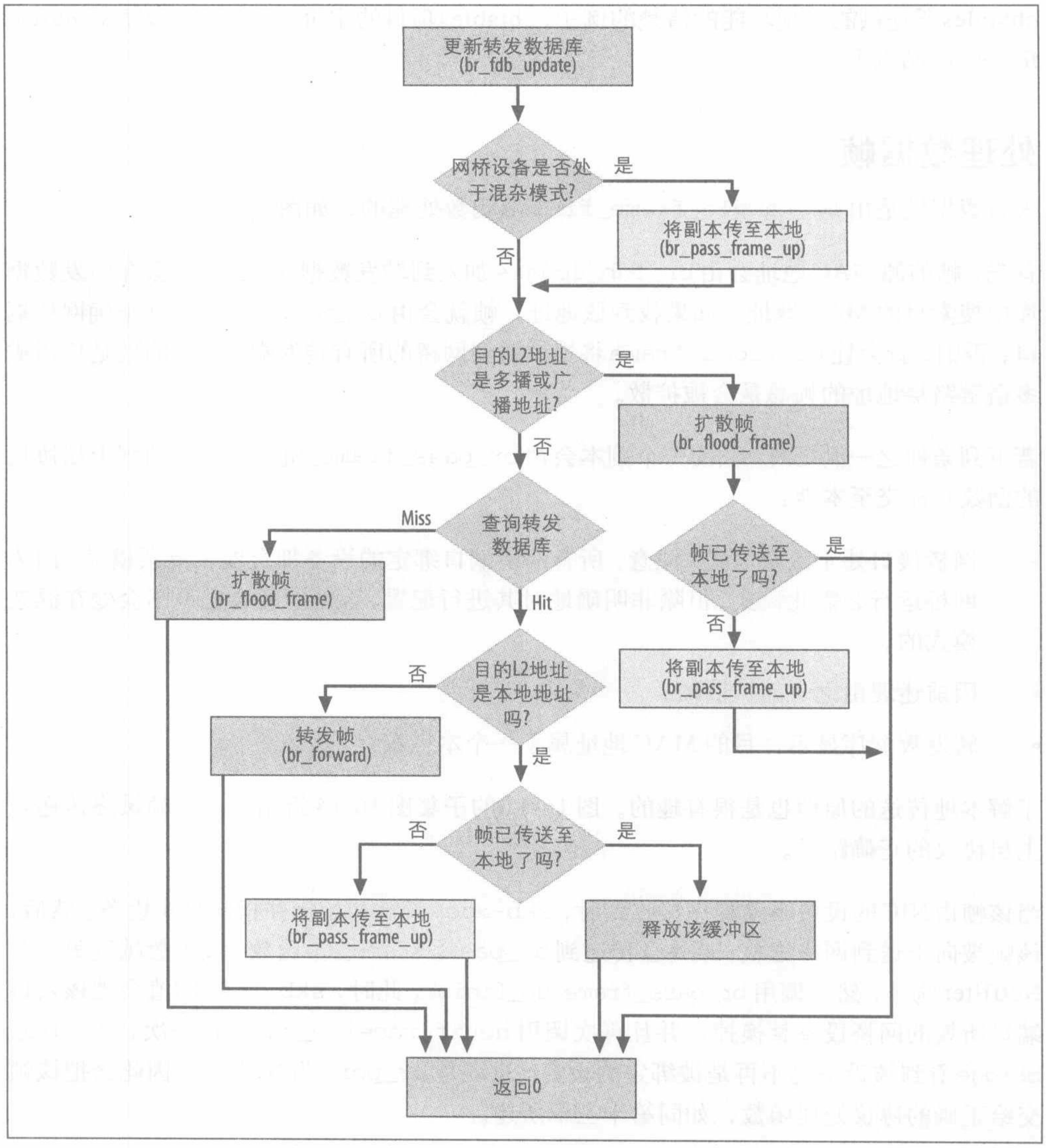


图 16-13：br_handle_frame_finish 函数

网桥驱动程序实现hard_start_xmit的函数是br_dev_xmit，而后者只是简单实现了网桥传输所需的基本逻辑。当网桥转发数据库查询成功时，br_dev_xmit就会从正确的网桥端口将该帧的拷贝发送出去。另一方面，当查询失败时，或者当目的MAC地址是L2多播或L2广播地址时，就会在所有符合条件的网桥端口上扩散该帧。

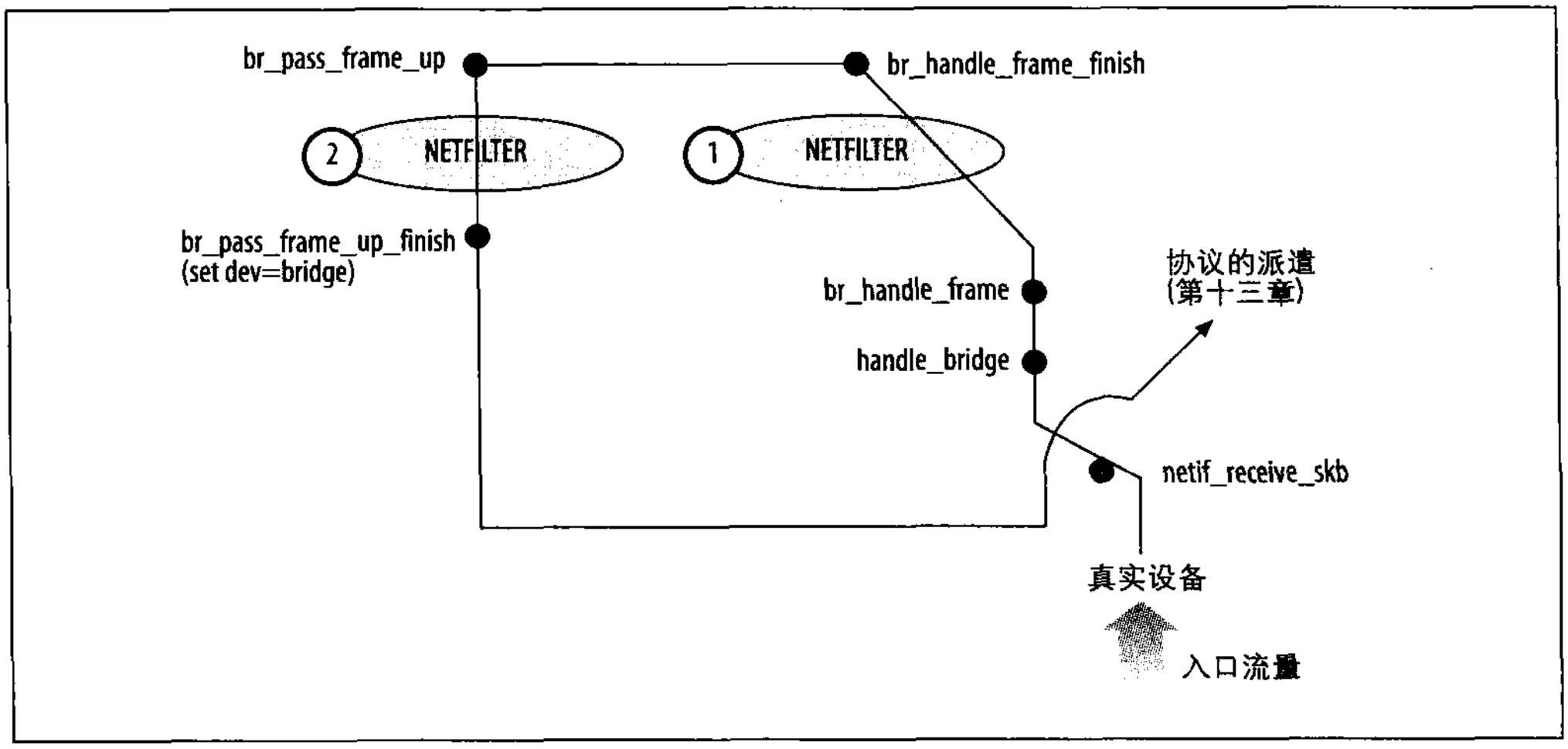


图 16-14：入口数据帧传送至本地

生成树协议（STP）

由第十五章已知 STP 如何运行。本章主要讲述的是：

- 如何处理入口 BPDU。
- 如何发出出口 BPDU。
- 如何处理定时器。

主要生成树函数

下面列出的是生成树程序用于实现第十五章描述的逻辑的一些主要函数：

br_become_root_bridge

br_is_root_bridge

br_become_root_bridge会把非根网桥变成根网桥。这项工作包括停止TCN定时器（因为该定时器不应该在根网桥上运行），以及启动Hello定时器（该定时器只在根网桥上运行）。这个函数也会更新其他定时器，使其采用本地配置的值，然后启动拓扑变化。当输入网桥为根网桥时，br_is_root_bridge返回1；否则，返回0。

br_should_become_designated_port

br_designated_port_selection

br_become_designated_port

br_is_designated_port

br_is_designated_for_some_port

如果输入端口应获得指定角色，则br_should_become_designated_port返回1；否则，返回0。br_designated_port_selection会遍历所有网桥端口，然后给应该获得指定角色的端口指派为指定角色（见第十五章“选择指定端口”一节）。

br_become_designated_port会把指定角色指派给一个网桥端口。当输入端口为指定端口时，br_is_designated_port就返回1；否则，返回0。给定一台网桥，如果该网桥至少有一个端口是指定角色，则br_is_designated_for_some_port返回1；否则，返回0。

br_supersedes_port_info

给定一个网桥端口和一个该端口接收的输入配置BPDU，若这个BPDU比该端口所知的优先权向量更高级，则返回1；否则，返回0。

br_should_become_root_port

br_root_selection

给定一个网桥端口和当前根端口，br_should_become_root_port会比较二者的优先权向量，如果前者的优先权向量较高级，则返回1（因此，给定的端口比当前根端口更适合作为根端口）；否则，会返回0。给定一台网桥，br_root_selection会按照第十五章“选择根端口”一节所述的规则选出根端口。

br_configuration_update

给定一台网桥，该函数可测定根端口和指定端口，并返回它们的信息。

br_port_state_selection

给定一台网桥，该函数可为每个网桥端口选出正确的端口状态。

br_topology_change_detection

br_topology_change_acknowledge

br_topology_change_acknowledged

br_topology_change_detection负责处理拓扑变化的探测，并区分根网桥探测到的拓扑变化和非根网桥探测到的拓扑变化。br_topology_change_acknowledge通过送出一个设有TCA标识的配置BPDU作为对接收到TCN的应答。br_topology_change_acknowledged还会停止TCN定时器。

br_record_config_information

br_record_config_timeout_values

给定一个网桥端口和一个输入配置BPDU，br_record_config_information就会把该BPDU的优先权向量记录在该端口的net_bridge_port数据结构中，然后重启消息生存期定时器，并且 br_record_config_ timeout_values 会记录 BPDU 内的定时器配置信息（见第十五章的图 15-8 所示）。

br_get_port

给定一个网桥设备和一个端口号，该函数返回相关的net_bridge_port结构。

网桥 ID 和端口 ID

由第十五章“网桥 ID 和端口 ID”一节已知道如何定义网桥 ID 和端口 ID。这两种 ID 的优先权元素都会被指定默认值，并且系统管理员可以对其重写覆盖，不过网桥ID的MAC地址部分以及端口 ID 的端口号部分则是由内核按照下列方式初始化：

网桥 MAC 地址

从被绑定的设备上所配置的 MAC 地址中，选出最低的 MAC 地址。这个选择工作都是由函数br_stp_recalculate_bridge_id来完成。在创建或删除网桥端口时，以及当被绑定的设备改变其MAC地址时，都会进行这一选择过程（参见“netdevice 通知链”一节）。

端口号

从 1 到 BR_MAX_PORTS 的范围中的第一个尚未被使用的号会被选中，作为端口号。建立网桥端口时，这个选择工作由 find_portno 来做（参见“给网桥添加端口”一节）。

在网桥设备上启动生成树协议

第17章中我们会看到如何在一个网桥设备上启动或关闭STP。端口的net_bridge结构中的 stp_enabled 字段会指出该网桥设备是否启动了 STP。

当没有使用 STP 时，“重要数据结构”一节所列的多数数据结构就会包含一些不需要的字段，比如会包含许多定时器。此外，不会有 BPDU 传输，也不会处理入口 BPDU。

你可能会认为，当创建一个新网桥时其 STP 是关闭着的，只有必要的字段和定时器会被初始化，稍后设置stp_enabled来开启STP时，其他相关字段和定时器才会被初始化并启动。但是，Linux 的并不是这样做的。

当一个网桥设备或网桥端口被初始化时，其所有字段（包括那些由STP所用的）都会被初始化，无论STP是否开启。用于传输BPDU的Hello定时器（由STP在根网桥上使用）也会启动。这样的话，如果稍后STP启动了，则所有数据结构都已就绪可用。

每次Hello定时器到期时，根据STP的规则，网桥就应该从其指定端口送出BPDU。由于网桥的定时器无论STP是否启动都会运行，因此，传输函数总是会检查stp_enabled的值，若该字段显示STP已关闭，就会立刻返回。一旦STP启动了（通过设定stp_enabled），BPDU的传输就会立刻启动。在一个网桥较少的系统中，让一个定时器定期到期且什么事也不做，并不是浪费资源的行为，但是应该避免出现这种情况。因为在一个拥有多个网桥实例的系统上，当不需要Hello定时器时若继续让其运行，那就很浪费CPU时间了。

处理入口BPDU

入口BPDU帧会传给br_stp_handle_bpdu（见图16-15所示），该函数更新转发数据库，并根据输入的BPDU的类型交给合适的处理函数，或者当满足下列条件之一时干脆将其丢弃：

- 该帧被截断。
- 接收该帧的网桥或网桥端口是关闭的。
- 网桥设备上的STP是关闭的。这种情况比较少见，因为当网桥上关闭STP时，br_handle_frame不会把BPDU交给br_stp_handle_bpdu（见“数据帧和BPDU”一节）。
- 网桥不知道如何解读BPDU信息。因为Linux内核只实现了IEEE 802.1D STP，因此只接收配置BPDU和TCN BPDU。其他任何类型的BPDU都会被丢弃。

br_received_config_bpdu函数会根据第十五章“传输配置BPDU”一节所述的逻辑处理配置BPDU。

br_received_tcn_bpdu函数会根据第十五章“让所有网桥知道拓扑变化”一节所述的逻辑处理TCN BPDU。

注意，网桥的BPDU的处理是用网桥锁（bridge lock）控制的串行化操作：不同的CPU不能同时处理同一个网桥上的BPDU。

传输BPDU

我们由第十五章“传输配置BPDU”和“让所有网桥知道拓扑变化”两节已知，何时要传输配置BPDU和TCN BPDU。以下是用到的一些传输函数：

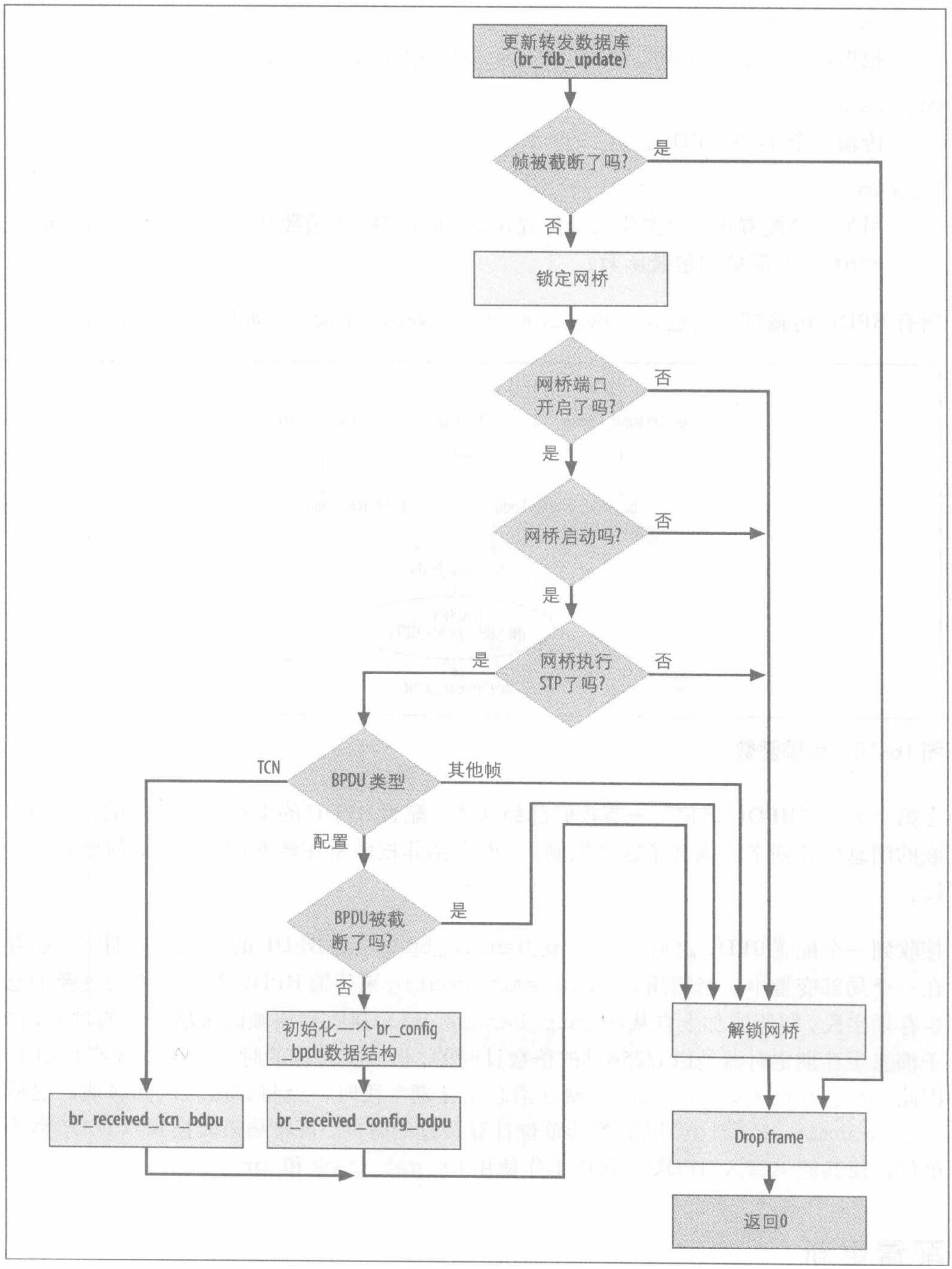


图 16-15：br_stp_handle_bpdu 函数

br_transmit_config

根据第十五章“传输配置 BPDU”一节所述的逻辑传输配置 BPDU。

br_transmit_tcn

传输一个 TCN BPDU。

br_reply

用另一个配置BPDU作为入口配置BPDU的应答。该函数是一个对 br_transmit_config的简单的包装函数。

所有 BPDU 传输都会通过 NF_BR_LOCAL_OUT Netfilter 钩子，如图 16-16 所示。

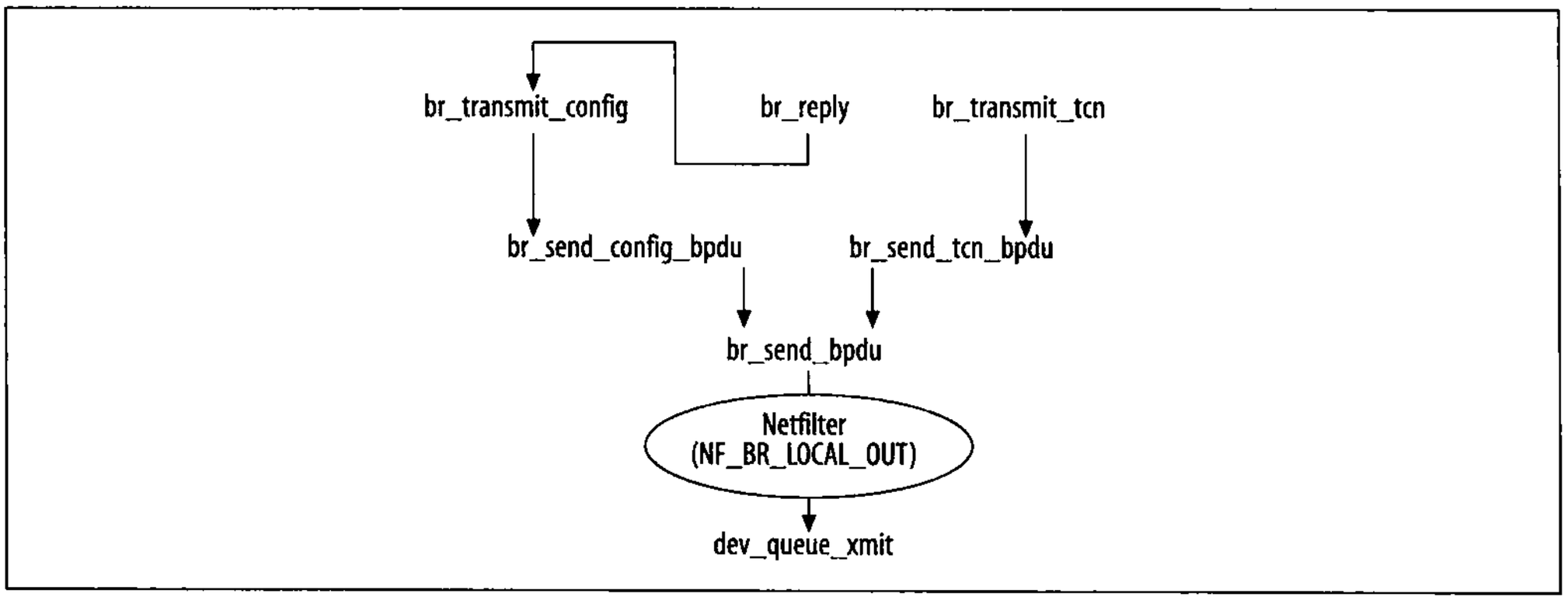


图 16-16：传输函数

在第十五章“BPDU 老化”一节我们已经知道，配置 BPDU 的生存期是有限的，因为内嵌的消息生存期字段强加了这个限制。下面介绍非根网桥在转发BPDU前如何更新该字段。

接收到一个配置BPDU之后，br_stp_handle_bpdu会把BPDU的消息生存期字段储存在一个局部变量中。当调用 br_transmit_config 来传输 BPDU 时，它就会更新消息生存期字段，即原值加上自从br_stp_handle_bpdu接收到原帧以来所经过的时间。由于消息生存期定时器是以1/256秒的倍数计时的，但内核管理的时间单位是滴答(tick)，因此，br_stp_handle_bpdu 储存消息生存期字段时，会将其值转成滴答值。之后br_transmit_config会以滴答为单位计算经过的时间，再将结果转换到以1/256秒为单位，使其能够写入 BPDU。转换工作使用 br_get_tick 和 br_set_tick。

配置更新

在第十五章我们已知道，系统管理员可以使用配置参数来影响STP收敛后形成的拓扑。我们也知道网桥端口的角色和状态的选择取决于网桥当前的信息和通过配置BPDU所接

收到的信息，特别是优先权向量（见第十五章的图 15-8 所示）。最后，我们在第十五章的“定义活跃拓扑”一节也知道有哪些事件会触发网桥配置更新，以及配置更新的有些什么内容。

负责配置更新的函数是br_configuration_update。表16-1所示是该函数被调用的场合和时机。

表 16-1：触发配置更新的函数

何处	何时
br_received_config_bpdu	在一个网桥端口上收到一个有更好优先权向量的 BPDU
br_message_age_timer_expired	一个网桥端口知道的信息已经过期。参见第十五章“BPDU 老化”一节
br_stp_disable_port	一个网桥端口已被关闭
br_stp_change_bridge_id	网桥 ID 的 MAC 地址部分已经被改变。参见“Bridge ID 和 Port ID”一节
br_stp_set_bridge_priority	网桥 ID 的优先权部分已经被改变
br_stp_set_path_cost	端口路径开销已被改变

每次调用 br_configuration_update 后，总是会接着调用 br_port_state_selection，它根据端口所分派的角色负责更新每个网桥端口的状态。状态变化是通过“改变网桥端口状态”一节所介绍的那些函数实现的。

在第十七章“处理配置变化”一节中，你能够找到致使表16-1所列函数执行的用户命令。

选择根网桥

在第十五章“选择根网桥”一节，我们已明白如何选出根网桥。当网桥首次启动时，会以为自己就是根网桥。接下来，根据所接收的入口配置BPDU的信息以及系统管理员做出的配置，根网桥的地位可能会有所变化。

图 16-17 给出了能改变网桥根地位的事件如何改变网桥角色。

成为根网桥

能使非根网桥成为根网桥的事件的处理函数是按照如图 16-17（a）所示的流程工作的：首先保存根状况，然后更新端口角色和状态。如果这样的更新会把网桥变成根网桥，就会展开必要的动作，例如启动和停止正确的定时器。

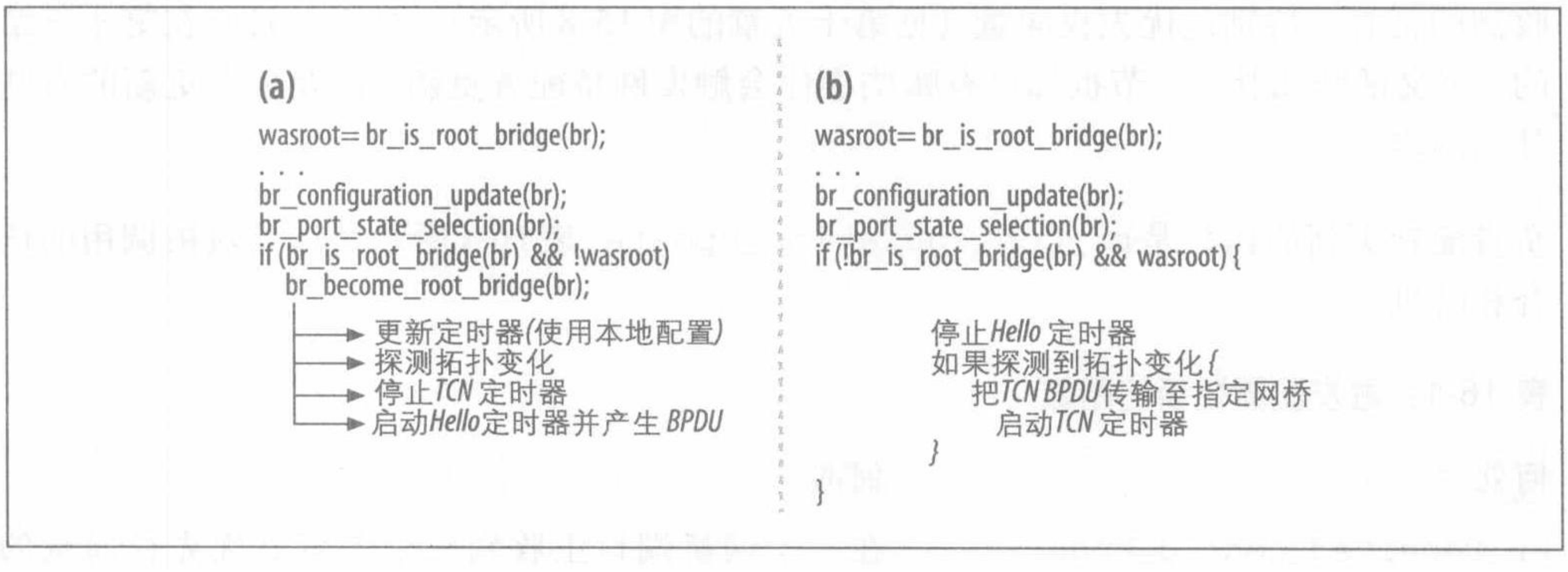


图 16-17：（a）成为根网桥；（b）放弃根网桥角色

下面是触发配置更新和能选择一个非根网桥作为新的根网桥的函数：

`br_stp_change_bridge_id`
`br_stp_set_bridge_priority`

当网桥的MAC地址以及网桥的优先权变化时，就会分别调用这两个函数。因为这两项就是构成网桥ID的两个字段，而且根网桥的选择是基于网桥ID的，因此，这两个字段的任何变化都可能会改变根网桥。

`br_stp_disable_port`

当你把某个网桥中唯一能到达当前根网桥的端口关闭时，生成树就等于被割裂了，当前网桥所属的那部分就必须选出新的根网桥。这也就是为什么第十五章中的多数例子都有冗余链路的原因。

`br_message_age_timer_expired`

当某个端口所接收的信息（由指定网桥送出）过期时（有可能是因为指定网桥不再是指定网桥，或者是因为指定网桥坏了），该端口就会获派指定角色。因为这是拓扑发生变化，所以有可能也会改变根网桥。

放弃根网桥角色

当根网桥接收到的BPDU拥有更高的优先权向量时，根网桥就会让出其根角色。这种情况会由 `br_received_config_bpdu` 探测，其处理方式就如图 16-17（b）所示。

定时器

在第十五章“定时器”一节已说过 STP 程序所用的端口和网桥定时器。

端口和网桥定时器可分别用br_stp_port_timer_init和br_stp_timer_init进行初始化。

表16.2和表16.3列出了定时器到期时要执行的函数。这些函数和那两个初始化函数都是在 *net/bridge/br_stp_timer.c* 里定义的。所有定时器处理函数执行时，都必须持有网桥锁。

表 16-2：STP 网桥定时器处理函数

定时器	处理函数
Hello	br_hello_timer_expired
Topology Change Notification	br_tcn_timer_expired
Topology Change	br_topology_change_timer_expired

表 16-3：STP 端口定时器处理函数

定时器	处理函数
Max Age	br_message_age_timer_expired
Forward Delay	br_forward_delay_timer_expired
Hold	br_hold_timer_expired

处理拓扑变化

由第十五章“拓扑变化”一节已知哪些事件会被认为是拓扑变化。这些事件是由下列函数进行探测：

br_make_blocking

当 STP 决定阻塞一个转发端口时就会调用该函数。

br_forward_delay_timer_expired

当处于 BR_STATE_LEARNING 态（也就是尚未至转发态）的某一端口要转换到 BR_STATE_FORWARDING 态时，就会调用该函数。

br_become_root_bridge

当非根网桥变成根网桥时就调用该函数。参见“变成根网桥”一节可以知道什么时候调用该函数。

br_received_tcn

当某个网桥端口接收到一个TCN BPDU时就调用该函数。参见“处理入口BPDU”一节可以知道何时调用该函数。

netdevice 通知链

由于虚拟网桥设备是定义在真实（被绑定）设备之上的一个抽象，当其任一被绑定的设备改变其状态时，网桥设备就有可能受到影响。因此，桥接子系统的初始化函数 —— 在“桥接程序的初始化”一节介绍过 —— 会用netdevice通知链向内核注册br_device_event回调函数。桥接程序只对被绑定的设备感兴趣，所以，任何有关非绑定设备的通知消息对桥接程序没有用，根本不需要注意。

下面是每个收到的事件通知消息（event notification）的处理方式：

NETDEV_CHANGEMTU

网桥设备的MTU会被更新，以反映出被绑定的设备中配置得最小MTU。

NETDEV_CHANGEADDR

当一个被绑定的设备改变其MAC地址时，其在转发数据库内的对应数据项会通过br_fdb_changeaddr更新，而网桥ID会通过br_stp_recalculate_bridge_id更新，以反映出本章“网桥ID和端口ID”一节所述的规则。

NETDEV_CHANGE

这个通知消息可用于各种目的。桥接子系统只对载波状态变化感兴趣。

当被绑定的设备失去或检测到其载波状态，相关的桥接端口被br_stp_enable_port和br_stp_disable_port分别启动和关闭。当该设备相关的网桥设备还没有被管理员启动（即IFF_UP没有设置）时，该通知消息会被忽略。

NETDEV_FEATCHANGE

当一个被绑定设备的功能发生变化时，网桥设备的功能集合会用br_features_recompute进行更新，以反映出它的所有真实设备共同的功能集合。

NETDEV_DOWN

当被绑定的设备被管理员关闭时，相关的网桥端口也必须被关闭；这个工作由br_stp_disable_port来做。若该端口对应的网桥已关掉，就没再必要关闭端口，因为那意味着网桥端口也已经关闭了。

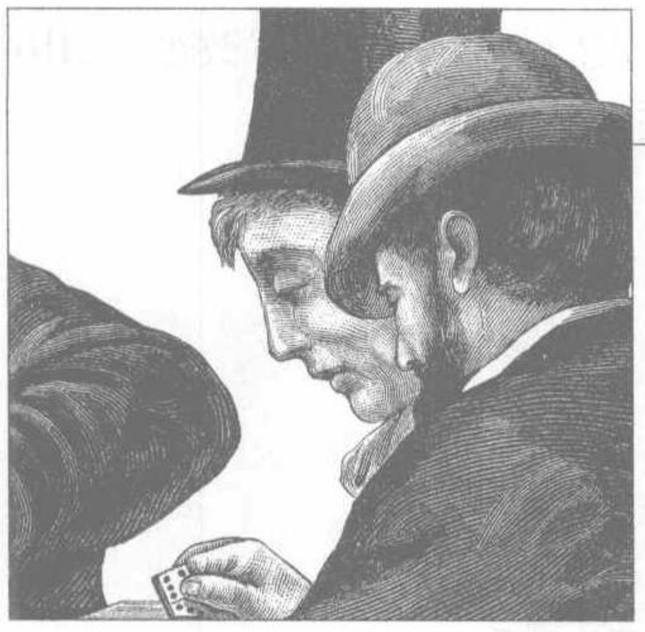
NETDEV_UP

当被绑定的设备被管理员启动时（即设置了IFF_UP标记），则相关的网桥端口会通过br_stp_enabled_port启动（如果有载波状态），而且相关的网桥设备也会启动。

NETDEV_UNREGISTER

当被绑定的设备被注销时，br_del_if函数会将相关的桥接端口删除。

除NETDEV_UNREGISTER之外，所有事件在处理时都必须持有网桥锁。



第十七章

桥接：其他主题

前几章中，我们已经知道桥接和STP是如何实现的，以及它们如何与网络堆栈融合。本章作为本书网桥部分的总结，主要说明网桥子系统如何与配置桥接的用户空间命令交互。我不会具体介绍那些命令本身，因为管理问题不在本书讨论之列。

我们还将探讨*/sys*目录中可以用于调整桥接的各种文件。本章结尾将详细讨论第十六章介绍过的数据结构。

用户空间配置工具

可以使用*brctl*工具配置桥接，该工具可从*http://bridge.sourceforge.net/*站点下载。利用*brctl*，可以创建网桥设备、将NIC绑定到网桥设备，以及为STP配置网桥参数和网桥端口参数。

除非安装了*libsysfs*库，否则*brctl*程序就使用`ioctl`接口与内核通信，前一种情况下则会使用*sysfs*接口。*libsysfs*库可以从*http://linux-diag.sourceforge.net/Sysfsutils.html*下载，它提供了用以访问和修改在*/sys*目录中的变量的全部原语。参见“通过/sys文件系统调整网桥”一节。

在第十六章“数据帧和BPDU”一节中，我们曾提到了ebtables工具。这个用户空间配置工具可以从*http://ebtables.sourceforge.net*下载。在本章将不讨论这个工具，读者可以在其首页上找到很不错的说明文档。

处理配置变化

表17-1列出了*brctl*命令以及内核桥接代码（配置层调用通知桥接变化）的回调函数。例如，当用命令*brctl addbr br0*创建一个网桥*br0*时，内核最后会调用`br_add_bridge`函数，该函数在第十六章“建立网桥设备和网桥端口”一节已介绍过。

注意，有些命令不需要调用任何回调函数。例如，若用命令*brctl sethello br0 3*改变Hello时间，则桥接程序会立刻看见新设置的值：STP不需要再采取任何动作。

表17-1：brctl命令和相关的内核处理程序

brctl命令	描述	桥接回调函数
addbr	建立一个网桥设备	br_add_bridge
delbr	删除一个网桥设备	br_del_bridge
addif	建立一个网桥端口	br_add_if
delif	删除一个网桥端口	br_del_if
setageing	为转发数据库中的地址设置老化时间	N/A
setbridgeprio	设置网桥优先权	br_stp_set_bridge_priority
setfd	设置Forward Delay定时器	N/A
sethello	设置Hello定时器	N/A
setmaxage	设置Max Age定时器	N/A
setpathcost	设置端口路径开销	br_stp_set_path_cost
setportprio	设置端口优先权	br_stp_set_port_priority
show	显示网桥设备	N/A
showmacs	显示一个网桥的转发数据库信息	N/A
showstp	显示一个网桥的生成树信息	N/A
stp	开启或关闭网桥上的STP	N/A

无论*brctl*是用ioctl 命令还是用*sysfs*接口来与内核交流，都会用表17-1中列出的函数。无论某个给定命令是否需要调用桥接回调函数，始终都会调用一个内核函数去处理*brctl*命令。

新旧接口

由于内核代码对新旧接口都支持，因此必须能正确两个不同版本的接口。不幸的是，这使得处理桥接配置命令的ioctl 代码有点混乱。旧接口完全基于ioctl命令，而新接口只有部分命令使用 ioctl ，而其他的使用*sysfs*。

图17-1（a）和图17-1（b）所示分别为这两种接口使用的ioctl命令如何被转送到正确的函数进行处理（当然，这并非你所说的简洁的代码）的情形。

最上面的那个菱形框就是sock_ioctl（在*net/socket.c*里）所做的初始分发工作。注意，

该图只给出发出桥接命令的细节，尽管一些函数也被其他功能的命令共享。浅色框中的命令就是由新接口所使用的。

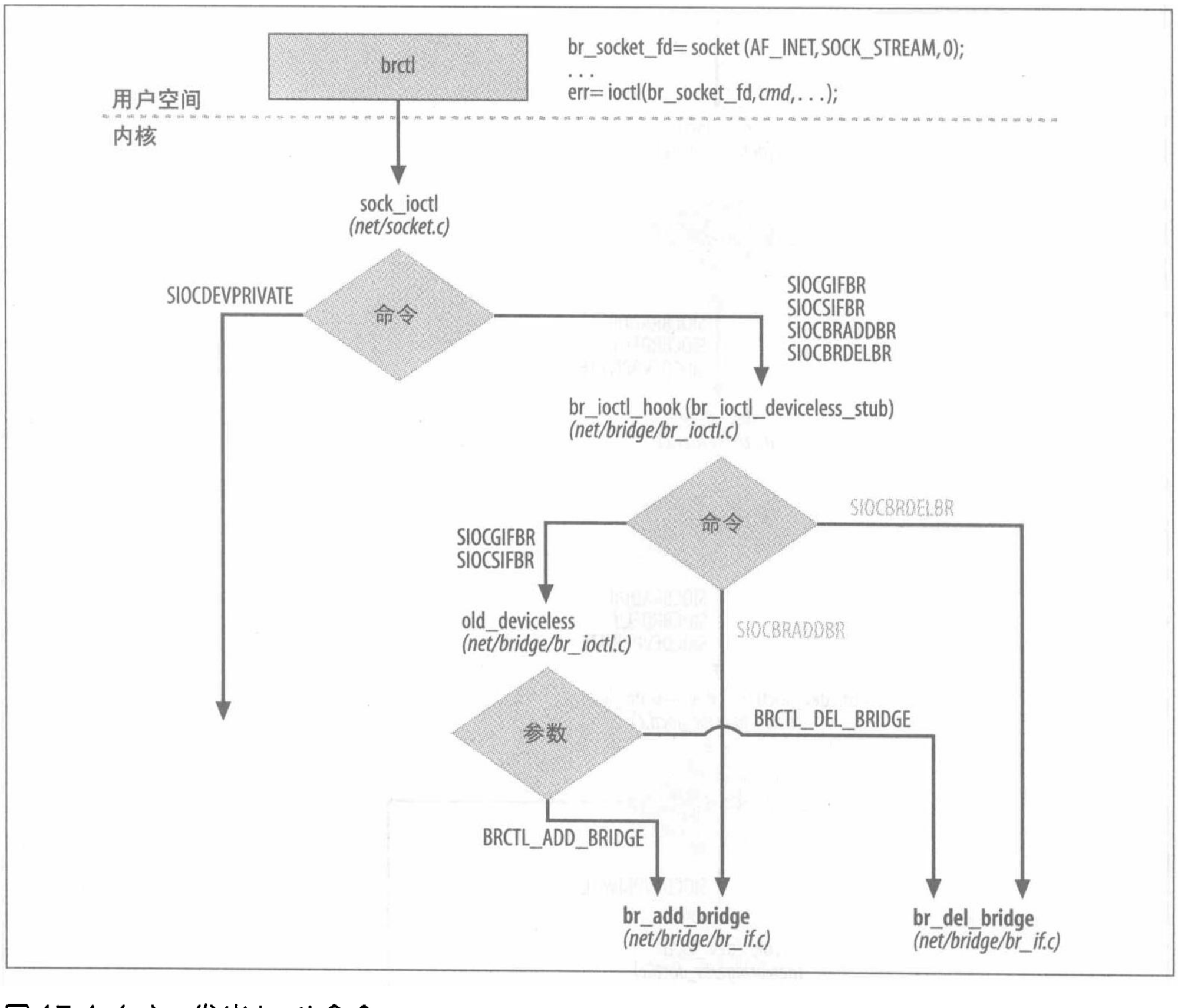


图 17-1（a）：发出 ioctl 命令

有个细节要注意，如果网桥内核模块还没在内存中，`br_ioctl_deviceless_stub`就会试图加载它。

下面两节将介绍这两种接口的更多细节。

建立网桥设备和网桥端口

我把*brctl*命令分成两类：用于建立与删除网桥设备及网桥端口的命令，以及用于配置或转储网桥设备及网桥端口（包括 STP 的细节）的配置信息的命令。

新旧接口都使用ioctl命令来实现第一类命令。新旧接口所用的具体ioctl命令程序如表17-2所示。

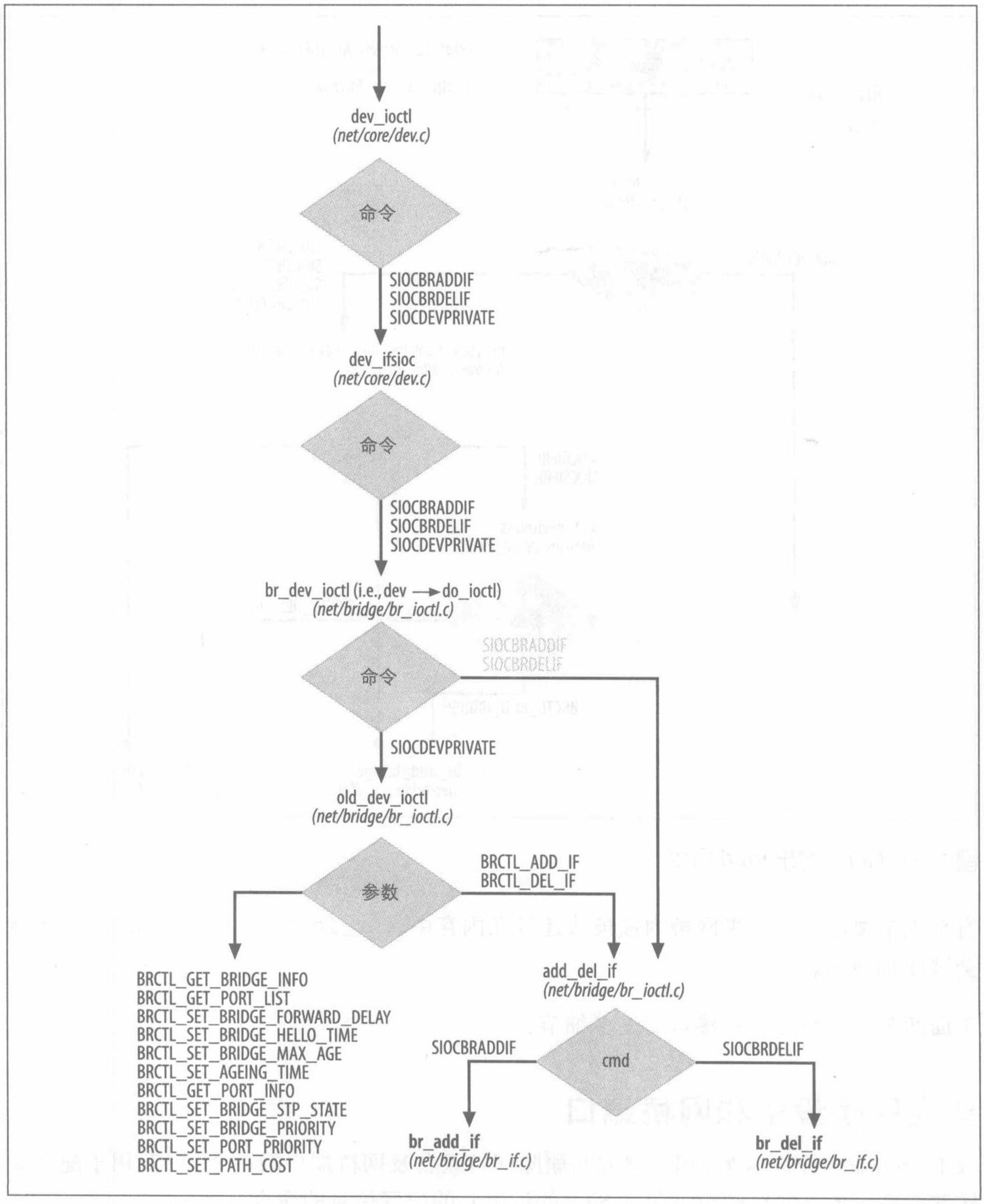


图17-1（b）：发出ioctl命令

表 17-2：用于建立网桥设备和端口的 ioctl 命令

brctl 命令	旧接口（ioctl 命令，参数）	新接口（ioctl 命令）
addbr	SIOCSIFBR, BRCTL_ADD_BRIDGE	SIOCBRADDBR
delbr	SIOCSIFBR, BRCTL_DEL_BRIDGE	SIOCBRDELBR
addif	SIOCDEVPRIVATE, BRCTL_ADD_IF	SIOCBRADDIF
delif	SIOCDEVPRIVATE, BRCTL_DEL_IF	SIOCBRDELIF

注意，为了识别具体的 *brctl* 命令，旧接口需用给 `ioctl` 命令传递一个参数，而新接口只需 `ioctl` 命令就足够了。

配置网桥设备和网桥端口

第二类命令的实现方式在新旧接口间有所不同：旧接口使用 `ioctl` 命令，而新接口使用 *sysfs* 命令。

旧接口所用的具体 `ioctl` 命令如表 17-3 所示。

表 17-3：旧接口用于配置网桥设备和端口的 ioctl 命令

brctl 命令	ioctl 命令，参数
setageing	SIOCDEVPRIVATE, BRCTL_SET_AGEING_TIME
setbridgeprio	SIOCDEVPRIVATE, BRCTL_SET_BRIDGE_PRIORITY
setfd	SIOCDEVPRIVATE, BRCTL_SET_FORWARD_DELAY
sethello	SIOCDEVPRIVATE, BRCTL_SET_HELLO_TIME
setmaxage	SIOCDEVPRIVATE, BRCTL_SET_MAX_AGE
setpathcost	SIOCDEVPRIVATE, BRCTL_SET_PATH_COST
setportprio	SIOCDEVPRIVATE, BRCTL_SET_PORT_PRIORITY[a]
show	SIOCDEVPRIVATE, BRCTL_GET_BRIDGE_INFO
showmacs	SIOCDEVPRIVATE, BRCTL_GET_FDB_ENTRIES
showstp	SIOCDEVPRIVATE, BRCTL_SET_BRIDGE_INFO
stp	SIOCDEVPRIVATE, BRCTL_SET_BRIDGE_STP_STATE

a：1.0.6 版本的 *brctl* 使用 BRCTL_SET_PATH_COST 而非 BRCTL_SET_PORT_ PRIORITY。这可能是剪切、粘贴时弄错了。

注意，所有*brctl*命令都使用SIOCDEVPRIVATE命令（尽管在Linux中强烈反对使用该命令）以及一个参数来识别正确的操作。

基于*sysfs*的配置只简单地识别出*/sys*里的正确文件，然后使用*libsysfs*库把配置内容写入该文件。表17-2中所列的操作无法通过*sysfs*实现，因为在该层次中没有相关文件可供使用。

通过/proc文件系统调整网桥

普通的桥接程序不会在*/proc*文件系统中建立任何文件。但是，网桥的防火墙扩展程序会在*/proc/sys/net/bridge/*里建立一些文件，使得*net/bridge/br_netfilter.c*里的主要函数不去处理它们所接收的缓冲区就返回。这些文件由br_netfilter_init建立，而br_netfilter_init是当桥接程序初始化时由br_init调用的（参见*net/bridge/br_netfilter.c*）。

通过/sys文件系统调整网桥

就像我在“用户空间配置工具”一节所说，*brctl*可以通过*sysfs*接口配置网桥和STP参数。在说明内核如何处理来自*brctl*的命令前，先看一看*/sys*里的信息是如何组织的。

内核会在*/sys/class/net*中为每个已注册的网络设备建立一个目录。这个目录用于输出应用在网络设备上只读的和可读写的一般性参数。网桥设备（和其他网络设备那样，会分配给一个目录）在其目录中包含两个特殊子目录：*bridge*和*brif*。第一个子目录用于输出网桥的配置参数，而第二个子目录包含一些软（符号）链接，每个链接都指向一个绑定设备（也就是一个网桥端口）的目录。图17-2中给出的范例是拥有两个Ethernet设备（*eth0*和*eth1*）的系统，其中管理员已经建立了一个网桥设备*br0*，同时让*eth0*和*eth1*绑定到*br0*上。

*br0*目录包含另一个网桥专用文件：*brforward*。这个文件用于输出网桥转发数据库（以二进制格式）。可以用*brctl showmacs*命令将其转储为文件。

*bridge*目录中的文件对应于net_bridge数据结构的字段，而*brport*目录中的文件对应于net_bridge_port数据结构的字段（注1）。

注1：change_ack是topology_change_ack的简写。

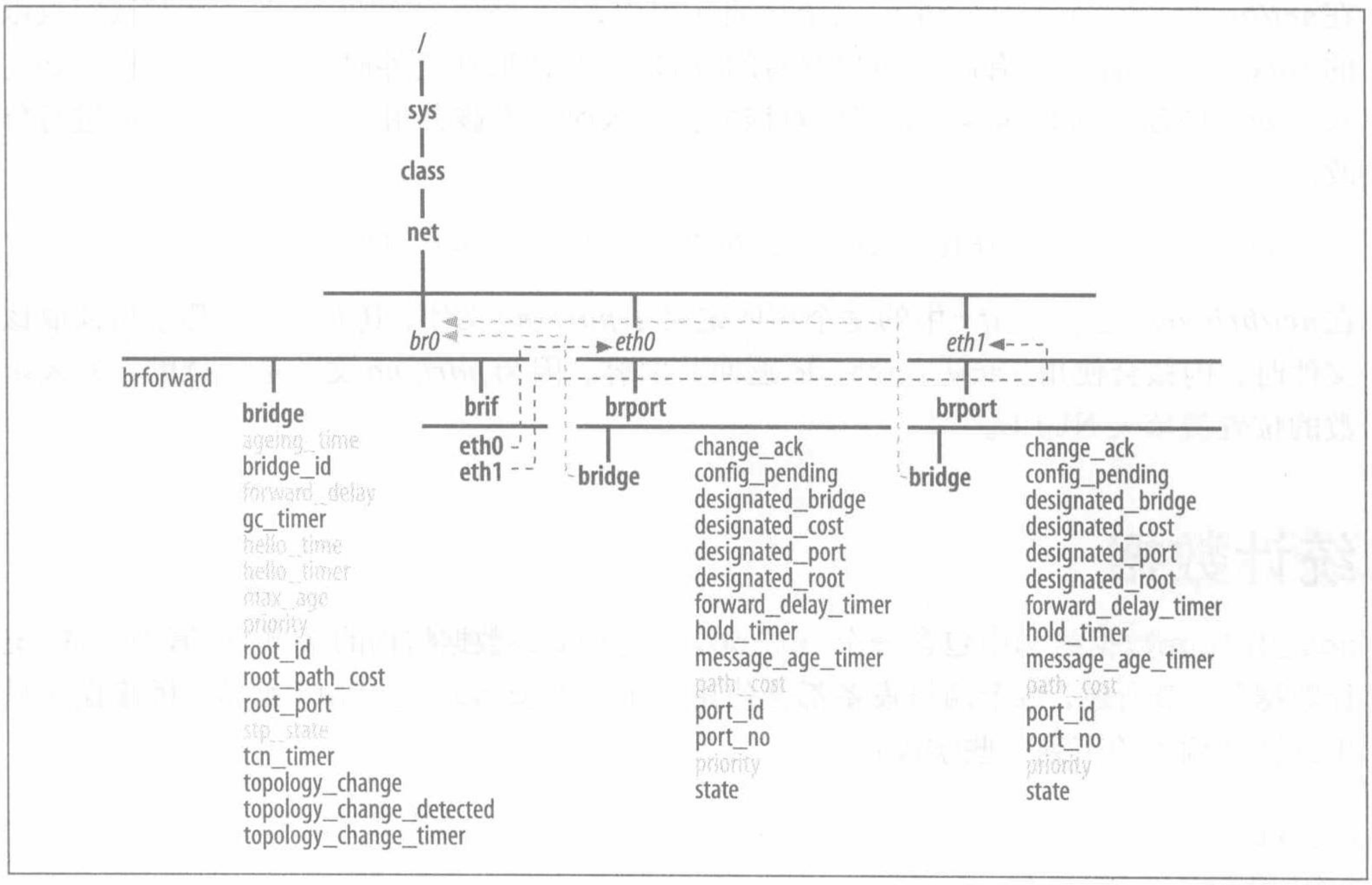


图 17-2：用 sysfs 输出的网桥信息的范例

建立网桥设备时（见第十六章“建立一个新网桥设备”一节），br_sysfs_addbr 会在网桥设备目录（例如，前例的*br0*）中建立网桥参数文件和目录。当一个设备被绑定时，其目录（前例中即为*eth0*和*eth1*）会由br_sys_addif建立。br_sys_addif函数也会建立网桥的*brif*目录。

图17-2中的所有文件都是只读的，但那些浅颜色标识的例外，它们也可以被改写。例如，*brctl*用于配置网桥和网桥端口参数的那些文件就是可以写入的（通过*libsysfs*）。

与网桥设备目录中的文件交互的内核代码在*net/bridge/br_sysfs_br.c*中，而和网桥端口设备（也就是被绑定的设备）目录中的文件交互的代码在*net/bridge/br_sysfs_if.c*中。

乍一看，这些代码好像很复杂，但实际上它们相当简单而且组织得很好。对于*bridge*和*brport*目录中创建的每个文件（每个网桥或端口的属性），代码用一个特殊宏实例定义了对该文件发出读写请求时应该要调用哪些函数。我们跳过这些宏如何置入一张表内以及如何由先前介绍的br_sysfs_addxxx函数来使用的细节，直接来看一组它们的应用示例。

```
static CLASS_DEV_ATTR(max_age, S_IURGO | S_IWUSR, show_max_age, store_max_age)
```

在*net/bridge/br_sysfs_br.c*中的这个声明使用CLASS_DEV_ATTR宏定义了具有读写权限的*max_age*文件（只有超级用户有写的权限）。当读取该文件时，内核会使用show_max_age函数返回其内容；而若要对该文件写入时，内核会用store_max_age进行修改。

```
static BRPORT_ATTR(port_no, S_IURGO, show_port_no, NULL)
```

在*net/bridge/br_sysfs_if.c*中的这个声明定义了*port_no*文件，具有只读权限。当读取该文件时，内核会使用show_port_no返回其内容。因为*port_no*文件是只读的，写入函数的位置就填入NULL。

统计数据

net_bridge数据结构中包含一个net_device_stats数据结构的实例。如第十二章“统计数据”一节所述，每个网络设备都会使用一个net_device_stats结构。桥接程序只用到该结构中的下列一些字段：

tx_packets

tx_bytes

: tx_packets表示本地所产生并经过网桥设备发出的帧的数量。该字段由br_dev_xmit更新。注意，虽然扩散帧从所有开启的端口出去，但只算一次。tx_bytes（已发出的tx_packets个帧的字节总和）也是由br_dev_xmit更新。

tx_dropped

: 由于扩散函数br_flood无法分配到缓冲区导致无法发出的帧的数量。

rx_packets

rx_bytes

: rx_packets由br_pass_frame_up在网桥设备每次把接收入口帧提交到本地时递增。rx_bytes为tx_bytes的副本。

这里所涉及的所有函数在第十六章都介绍过。

STP不保存任何统计信息。

本部分涉及的数据结构

在第十六章“重要的数据结构”一节曾对桥接程序所用的数据结构做了概述。本节对那些数据结构的字段逐一说明。一些不重要的字段没必要特别介绍，例如mac_addr和br_config_bpdu。

bridge_id 结构

由第十五章“网桥 ID 和端口 ID”一节已知道，网桥 ID 有两个元素，即优先权和地址：

`unsigned char prio[2]`

网桥优先权。

`unsigned char addr[6]`

网桥 MAC 地址。

注意，该数据结构的定义没有反映出 802.1t 引入的新变化。

net_bridge_fdb_entry 结构

下面这些字段用于定义转发数据库内的每个数据项。

`struct hlist_node list`

用于把该数据结构链接至 hash 表 bucket 的冲突元素列表的指针。

`struct net_bridge_port *dst`

网桥端口。

`struct rcu_head rcu`

当删除使用读—复制—更新（read-copy-update，RCU）机制的数据结构时会用到（参见 *net/bridge/br_fdb.c* 里的 `br_fdb_put`）。

`atomic_t use_count`

引用计数。参见第三十章“查找”一节。

`unsigned long ageing_timer`

Aging 定时器。内核的其他部分将它拼写为 *aging* 或 *ageing*。参见第十六章“老化”一节。

`mac_addr addr`

MAC 地址。这是查询函数所用的关键字段。

`unsigned char is_local`

当该标识为 1 时，表示 MAC 地址 `addr` 被配置在一个本地设备上。

`unsigned char is_static`

当该标识为 1 时，表示 MAC 地址 `addr` 是静态的，且不会过期。所有本地地址（即那些 `is_local` 为 1 的地址）也都是静态的。

net_bridge_port 结构

第一部分字段的使用不论是否启用了 STP：

`struct net_bridge *br`

`struct net_device *dev`

br 是网桥设备，而 dev 是被绑定的设备。参见第 16 章的图 16-6 所示。

`struct list_head list`

用于把该数据结构链接至 hash 表 bucket 的冲突元素列表的指针。

`u8 state`

端口状态。有效值列在 *include/linux/if_bridge.h* 的 BR_STATE_*XXX* 枚举列表中。

`struct kobject kobj`

由通用设备基础结构使用。该字段的作用是使“通过/sys 文件系统调整网桥”一节所讲的一切为可能。

`struct rcu_head rcu`

用于安全地销毁使用 RCU 机制的数据结构（见 *net/bridge/br_if.c* 中的 del_nbp）。

第二部分的字段只有当 STP 开启时才会用到：

`u8 priority`

端口优先权。

`u16 port_no`

端口号。

`port_id port_id`

端口 ID。由 br_make_port_id 使用 priority 和 port_no 的组合计算得到。

`unsigned char topology_change_ack`

当设定该标识后，经过这个端口传出的配置 BPDU 必须设定 TCA 标记。

`unsigned char config_pending`

当一个配置 BPDU 由于被 Hold 定时器阻塞而等待传输时，该标识为 1。

`port_id designated_port`

`bridge_id designated_root`

`bridge_id designated_bridge`

`u32 designated_cost`

在端口上最后接受的配置 BPDU 的优先权向量的四个元素（见第十六章的图 16-8

所示）。每接收到一个配置 BPDU，这四个字段就会由 br_record_config_configuration 进行更新。

u32 path_cost

端口路径开销。

struct timer_list forward_delay_timer

struct timer_list hold_timer

struct timer_list message_age_timer

端口定时器。参见第十五章“定时器”一节。

net_bridge 结构

第一部分的字段的使用不论是否启用了 STP：

spinlock_t lock

这个锁使改变net_bridge结构或改变port_list中的某个端口的操作串行化。使用 rcu_read_lock 和 rcu_read_unlock 原语只能对该锁进行只读访问。

struct list_head port_list

网桥端口列表。

struct net_device *dev

网桥设备（参见第十六章的图 16-6 所示）。

struct net_device_stats statistics

统计。参见“统计数据”一节。

spinlock_t hash_lock

struct hlist_head hash[BR_HASH_SIZE]

hash是转发数据库。hash_lock是用于对其数据项读写访问串行化的锁。只读访问使用 rcu_read_lock 和 rcu_read_unlock 原语。

struct list_head age_list

未使用。这个列表用于把转发数据库内的所有数据项，根据最后被使用的时间按递增的顺序链接在一起（参见第十六章的图16-6所示）。老化算法用该列表扫描数据库里的过期数据项。

unsigned long ageing_time

一个数据项没被使用时可以待在转发数据库里的最大时间。参见第十六章“老化”一节。

struct kobject ifobj

由通用设备基础结构使用。该字段的作用是使“通过/sys文件系统调整网桥”一节所讲的一切为可能。

unsigned char stp_enabled

当设定该标识后，表示网桥开启了STP。

下面介绍的字段只有当STP开启时才会用到。但forward_delay是个例外，不管STP是否开启都会用到。当STP开启时，网桥的端口就不会被指定为转发状态，它们使用forward_delay定时器度过中间状态。

bridge_id designated_root

根网桥的ID。

bridge_id bridge_id

网桥ID。

u32 root_path_cost

到根网桥最佳路径的开销。

unsigned long max_age
unsigned long hello_time
unsigned long forward_delay

网桥定时器。定时器的值在根网桥上进行配置，并且当根端口上收到配置BPDU时，会由br_record_config_timeout_values函数将该定时器的值保存在本地。

unsigned long bridge_max_age
unsigned long bridge_hello_time
unsigned long bridge_forward_delay

本地配置的网桥定时器。这只能由根网桥使用。

u16 root_port

根端口的端口号。

unsigned char topology_change

当根端口所接收的最新的配置BPDU中设定TC标识时，该标识就会被设定。当设定了topology_change标识时，该网桥所传输的任何配置BPDU都必须设定TC标识。参见第十五章“拓扑变化实例”一节。

unsigned char topology_change_detected

当探测到拓扑变化时，就会设定该标识。对于可能被认为是拓扑变化的条件，参见第十五章“拓扑变化”一节。

```
struct timer_list hello_timer
struct timer_list tcn_timer
struct timer_list topology_change_timer
```

网桥定时器。参见第十五章“定时器”一节。

```
struct timer_list gc_timer
```

转发数据库所用的垃圾收集定时器。参见第十六章“老化”一节。

本部分涉及的函数和变量

表17-4总结了第四部分介绍过的主要的函数、变量和数据结构。在第十六章“重生成树协议的主要函数”以及“定时器”两节中还可以找到其他的函数和变量。

表17-4：第四部分引入的函数、变量和数据结构

名称	描述
函数	
br_init br_deinit	初始化和清除内核桥接模块。参见第十六章的“桥接程序的初始化”一节
br_fdb_init	初始化转发数据库
br_netfilter_init	初始化桥接程序所使用的 Netfilter 钩子
br_stp_timer_init br_stp_port_timer_init	初始化网桥和网桥端口定时器
br_sysfs_addbr br_sysfs_delbr	通过 sysfs 接口处理与网桥设备有关的外部文件，参见“通过 /sys 文件系统调整网桥”一节
br_sysfs_addif br_sysfs_removeif	通过sysfs 接口处理和网桥端口有关的外部文件，参见“通过 /sys 文件系统调整网桥”一节
br_add_bridge br_del_bridge	建立和删除一个网桥设备，参见第十六章的“建立网桥设备和网桥端口”一节
br_add_if br_del_if	建立和删除一个网桥端口，参见第十六章的“建立网桥设备和网桥端口”一节
br_stp_recalculate_bridge	给定一个桥，选择一个在该网桥端口（即绑定设备）上配置的数值最小的MAC地址，并用它计算该网桥的ID
br_min_mtu	给定一个网桥，找到配置在该网桥端口上的最低 MTU 值
br_stp_enable_bridge br_stp_disable_bridge	启动和关闭一个网桥设备，参见第十六章的“启动和关闭网桥实例”一节

表 17-4：第四部分引入的函数、变量和数据结构（续）

名称	描述
br_stp_enable_port br_stp_disable_port	启动和关闭一个网桥端口，参见第十六章的“启动和关闭网桥端口”一节
_ _br_fdb_get br_fdb_get	在转发数据库中查找一个数据项。参见第三十章的“查找”一节
fdb_create br_fdb_insert bf_fdb_change_addr br_fdb_update br_fdb_cleanup	维护转发数据库的各种函数。参见第十六章的“转发数据库”一节和其小节中的内容
handle_bridge br_handle_frame br_handle_frame_finish br_stp_handle_bpdu br_forward br_flood br_pass_frame_up br_pass_frame_up_finish	用于处理入口帧的各种函数。参见第十六章的“处理入口流量”一节
br_received_config_bpdu br_received_tcn_bpdu	分别处理入口配置 BPDU 和 TCN BPDU。参见第十六章的“处理入口 BPDU”一节
br_transmit_config br_transmit_tcn br_reply br_send_bpdu	各种传输函数。参见第十六章的“传输 BPDU”一节
br_make_blocking br_make_forwarding	br_make_blocking阻塞一个端口，而br_make_forwardin将一个端口指定为转发状态，允许该端口发送和接受数据流量
br_get_tick br_set_tick	读写时间间隔，在 1/256 秒（配置 BPDU 使用）与时钟滴答（Linux 使用）间转换时要小心
变量	
BR_MAX_PORTS	可以添加到一个网桥设备上的网桥端口的最大数目
br_handle_frame_hook	初始化为桥接子系统处理入口帧所用函数的指针。见第十六章的图 16-11 所示
br_fdb_cache	用于转发数据库中元素分配的缓存。
数据结构	
struct mac_addr struct bridge_id	桥接程序所使用的主要数据结构。参见第十六章的“重要的数据结构”一节。

表 17-4：第四卷引入的函数、变量和数据结构（续）

名称	描述
struct bridge_fdb_entry	
struct net_bridge_port	
struct net_bridge	
struct br_config_bpdu	

本部分涉及的文件和目录

图 17-4 所示为本书第四部分中各章所涉及的文件和目录。

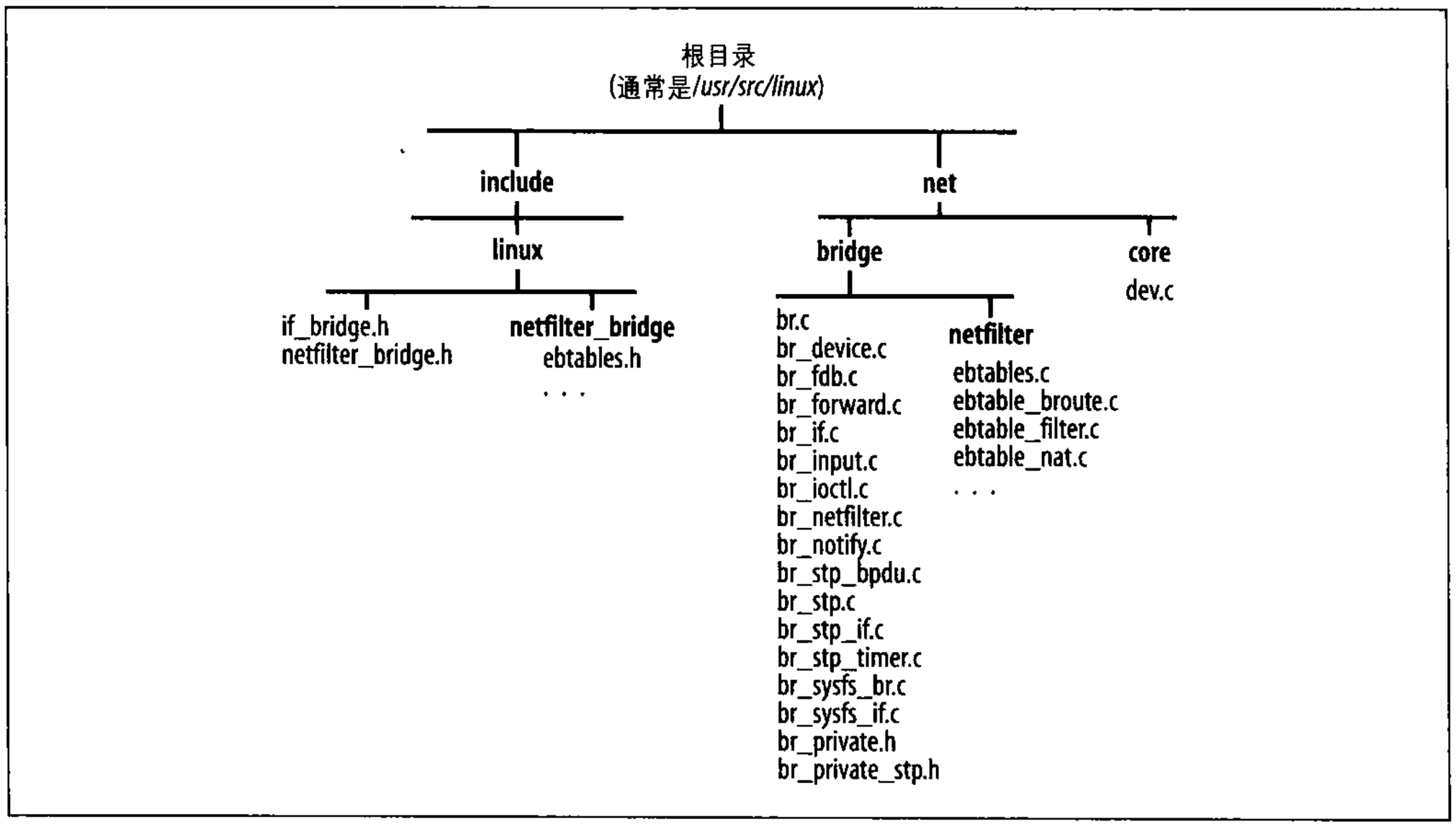


图 17-4：本部分涉及的文件和目录

第五部分

因特网协议第4版（IPv4）

Linux内核支持很多L3协议，如AppleTalk、DECnet、以及IPX，但是本书只谈一种主导现代网络联机的协议：IP。虽然IPv4会详细讨论，但IPv6只会在必要时简略论及而已。我不会花太多时间说明这些协议背后的理论，因为你应该多少都懂一点，所以，我会讲的是Linux的实现细节。我把焦点摆在那些不显眼的设计层面，或者和其他操作系统有实质上的差异处。我也会说明IPv4的主要缺点，以及IPv6如何试着给于解决。因此，虽然要有理论背景以及一些程序基础，但是，我也认为读者要熟悉IP协议基本行为。以下是各章概要：

第十八章 因特网协议第4版（IPv4）：概念

介绍IP层的主要任务及所用策略。

第十九章 因特网协议第4版（IPv4）：Linux的原理和功能

说明IP层接收函数如何处理入口封包，以及IP选项如何处理。

第二十章 因特网协议第4版（IPv4）：转发和本地传递

说明入口IP封包如何传递至本地的L4协议处理函数，或者当目的地IP地址不属于本地主机，而该主机有开启转发功能时，该如何给以转发。

第二十一章 因特网协议第4版（IPv4）：传输

说明L4协议如何衔接IP层来请求传输。

第二十二章 因特网协议第4版（IPv4）：处理分段

说明分段和重组是怎么处理的。

第二十三章 因特网协议第4版（IPv4）：其他主题

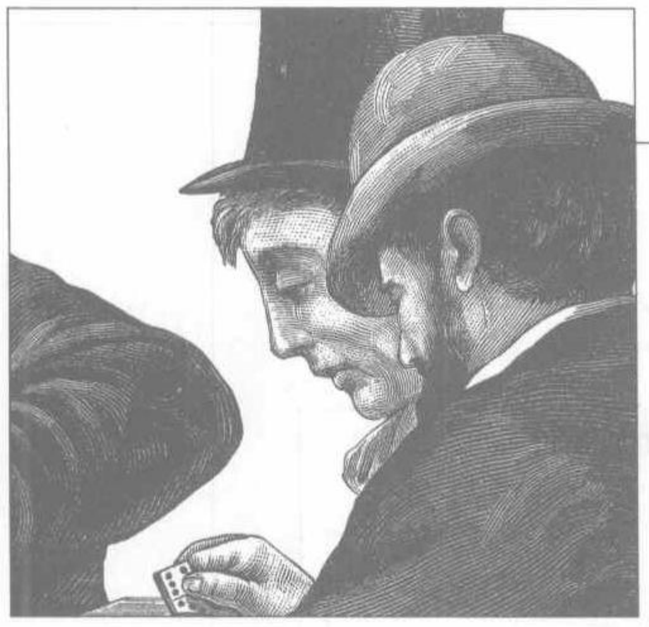
说明配置工具（如IPROUTE2套件的那些工具）如何衔接核心，出口封包上的IP报头的ID字段如何初始化，以及详细说明IP层所用的数据结构。

第二十四章 L4 协议和Raw IP 的处理

说明L4协议如何为入口流量注册处理函数。

第二十五章 因特网控制消息协议（ICMPv4）

说明ICMP协议的实现细节。



第十八章

因特网协议第4版（IPv4）：概念

本章说明IP协议的作用，然后讨论支持这些作用的IP报头字段以及这些作用对一些可能的实现的影响。虽然本章讨论的是Linux所做的一些选择，但是，实现细节会在后续各章说明。

了解IPsec安全套件协议与IP协议的整合方式也很有趣，但是，因为篇幅有限，我不会谈论这个话题。然而，偶尔我们会看见IPsec转换对关键函数的实现有何影响。

IP协议：大蓝图

图18-1所示是处理IPv4的一些Linux组件间的重要关系。主要函数之间的数据流向由箭头表示。我们会在后续几章分析所有这些函数。该图包含两个会在其他章节讨论的子系统：Neighboring子系统（邻居子系统）和Traffic Control子系统（流量控制子系统）。此外，图中还包含有许多Netfilter防火墙系统使用到的钩子（hooks）函数（注1）。

当你阅读网络代码，猜想某个特定函数是用于输入还是输出、是否在转发时被调用以及由谁调用该函数时，图18-1就是很有用的参考图。

由于IP层没有直接和Traffic Control子系统互动，因此，Traffic Control子系统留到本书第六部分再谈。然而，第二十一章“衔接邻居子系统”一节将会看到IP如何和Neighboring子系统互动。

IP协议的一些任务如下：

注1：图18.1中不含用于处理多播流量流量的函数(除了ip_mc_output之外)。此图包含的是主要的API；不过，还有其他API是用于特定情况。参见第二十一章。

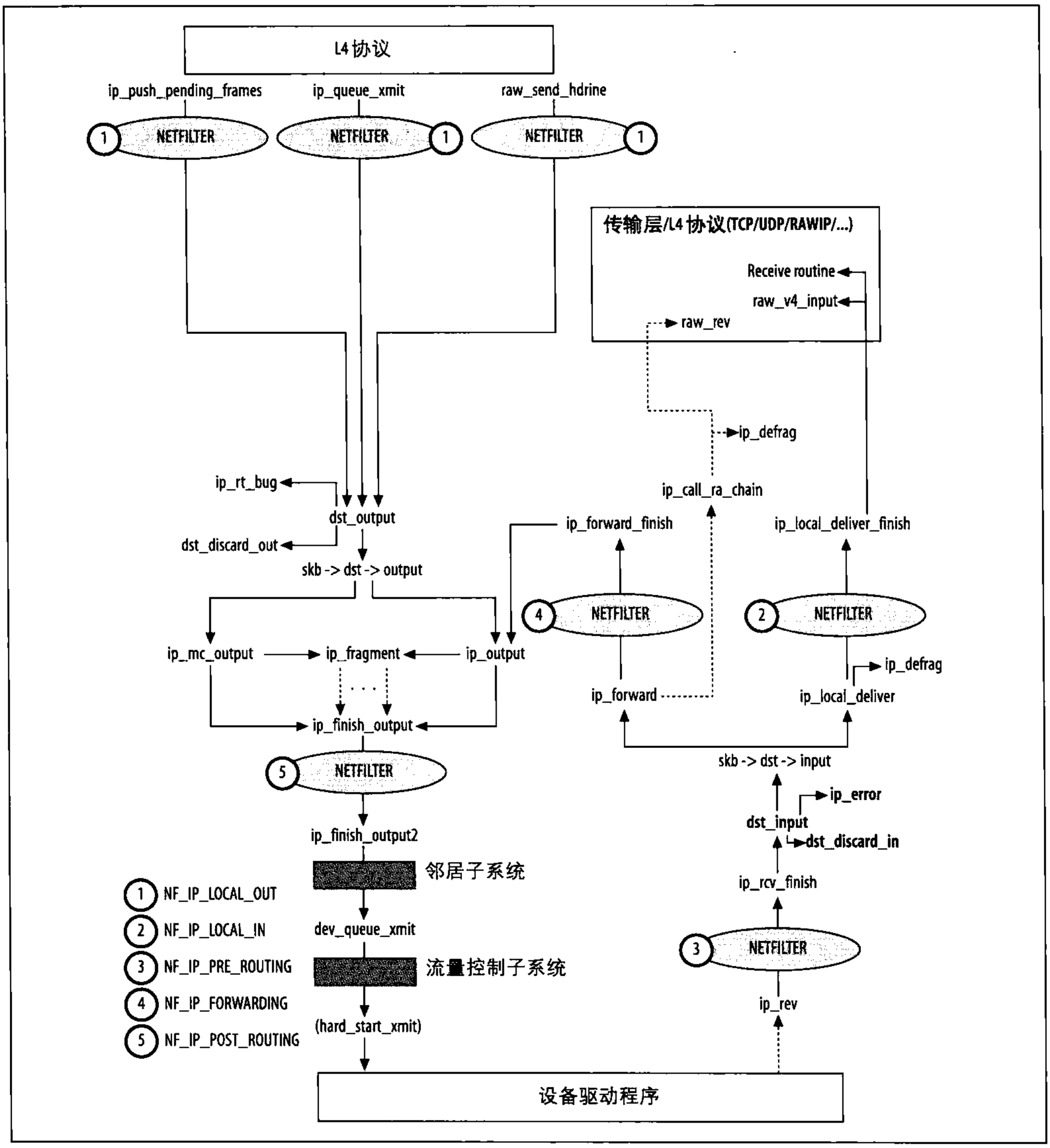


图 18-1：IP 内核协议栈的关键函数

健康检查

IP数据报一进入系统，就可能立刻被丢弃，原因是校验和不正确（传输时被损毁）、一个报头字段的值超出范围或者其他因素。

防火墙

如图 18-1 所示，Netfilter防火墙子系统（由客户端以*iptables*命令控制）在封包生

存期中可以在很多点上被调用，从而改变它的命运。我们在本书第五部分将会知道，Netfilter也可作用于L2层。

处理选项

IP协议包含一些应用程序可以使用的选项。即使原本的IP RFC（791）文件说主机和路由器都有实现这些选项的义务，但并非所有选项都有实现。有些选项已经认为是过时的，而有些选项则用于特殊情况。

分段/重组

IP报头的`len`字段够长，足以容纳多达64 KB的数据段，但是数据段很少达到这个极限值。事实上，MTU会随网络不同部分而变化，而这一点取决于用于传输的媒介（注2），所以，很有可能一个封包对传输沿路上的某一跳点而言过大。在这类情况下，封包必须分割成一些较小的单位才能成功传输。抵达目的地之前，每个片段可以再分割成子片段，而目的地必须重新组合这些片段。现在，并不鼓励使用分段，因为这会产生一些问题。具体在“封包的分段/重组”一节就会知道。

接收、传输以及转发操作

输入封包由接收函数处理，而输出封包则由传输函数处理。转发和传输有关，但是，处理的对象是从其他主机接收的封包（而非本地系统较高网络分层所产生的封包）。

我会在第二十四章将简要介绍Raw IP协议，而在第二十三章将简要介绍IP-over-IP（也称为IP隧道）。

IP报头

读者可能对IP报头的基本字段都很熟悉，但是，有些参数并非众所皆知，而且有些可能随时间连意义都转变了。图18-2所示是IP报头，而图后面的文字则用于简要说明它的用途：

版本（Version）

协议的版本。当前仅实现第4版和第6版，本章描述的是第4版。本书并没有讨论第6版；不过，当对上下文有用时，我们偶尔也会提及IPv6和IPv4之间的差异。

报头长度（Header Length，IHL）

报头的长度，以32位为单位。

注2：第二章中可以找到一份多数常见接口所用的MTU表。

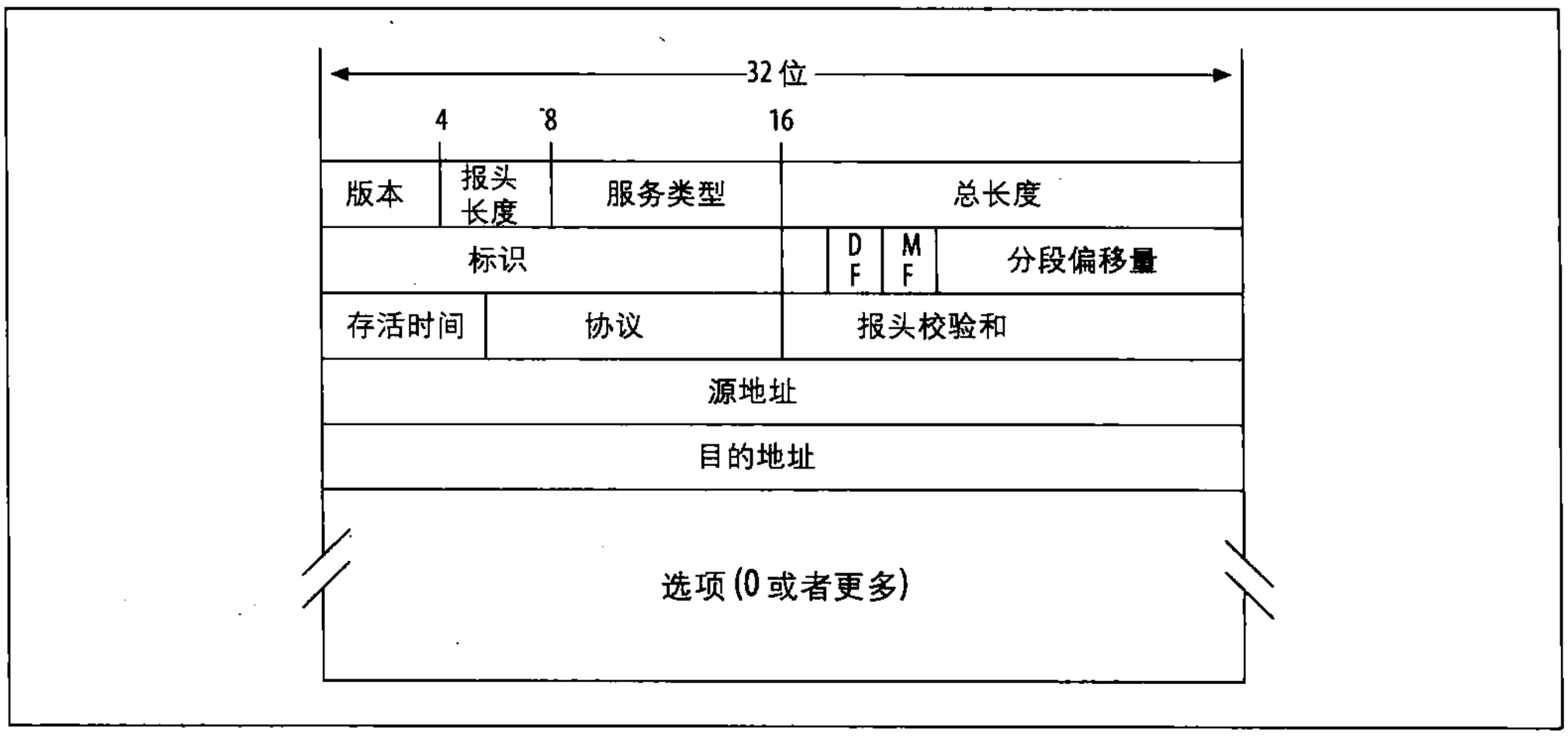


图 18-2：IP 报头

服务类型（Type of Service，TOS）

这个8位字段由3个子字段组成。我不会谈论其中的细节，但因为许多原因使得这几个字段的用途有限。最初，这个字段是为了协助 QoS（Quality of Service，服务质量）功能，也就是通知路由器此封包的传送者把哪些准则视为最重要：最小延迟、最大吞吐量等等。TOS 字段依然可以照此使用，但是，互联网研究人员认为这样太含糊，因此决定采用不同方式实现 QoS。所以，他们引进了 diffserv（Differentiated Services，差别服务）模型（注 3），结果改变了这个字段的结构和意义。和 diffserv 模型相配的新意义如图 18-3（b）所示。DSCP 指的是 DiffServ Code Point（DiffServ 码点）。每个可能的值都有唯一而特定的意义，指出应该如何对待该封包。TOS 字段先前两个未用的位现在已由 ECN（Explicit Congestion Notification，显式挤满通知）功能使用，如图 18-3（c）所示。多数用于读取和操作IP与TCP报头中的ECN标志的程序都位于*include/net/inet_ecn.h*和*include/net/tcp_ecn.h* 中。参考图 18-3 所示的 RFC 文件以了解其细节。

总长度（Total Length）

封包长度（包括报头），以字节为单位。

识别（Identification）

封包标识符。本章稍后就会知道，此字段在处理那些片段时扮演关键角色。

注 3： 可以在IETF网站（*http://www.ietf.org/html.charters/OLD/diffserv-charter.html*）找到其他有关diffserv的信息（因某些原因，有OLD的URL比没有OLD的URL更新）。

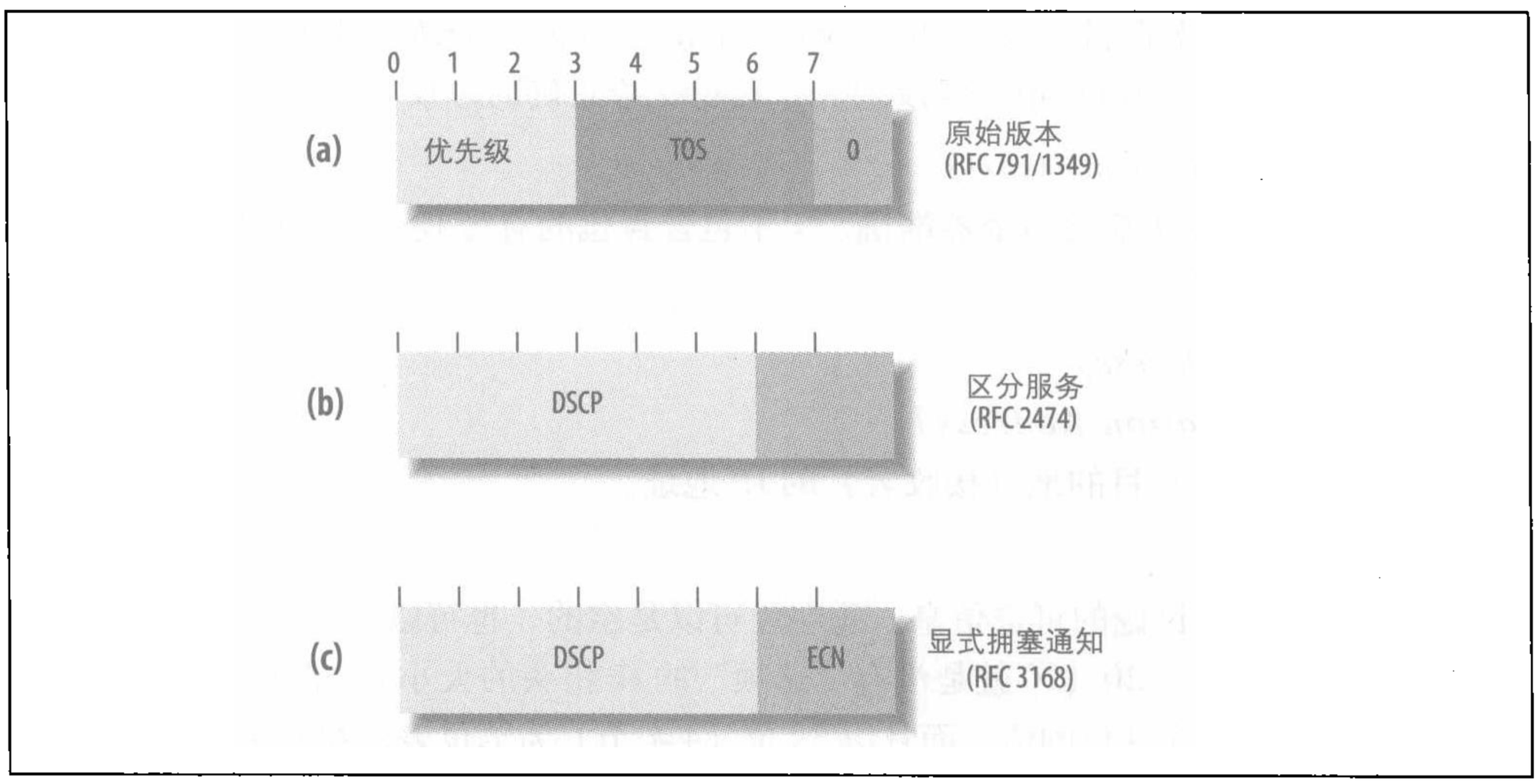


图 18-3：IP 报头的 TOS 字段的新旧意义

DF（Don't Fragment，不分段）

MF（More Fragments，还有其他片段）

片段偏移量（Fragment Offset）

这三个字段以及识别字段会由 IP 协议的分段 / 重组功能使用。参见“封包的分段 / 重组”一节。

存活时间（Time To Live，TTL）

此字段假定是代表自 IP 封包传输后多少秒数就要被丢弃（如果还没抵达最终目的地）。然而，由于路由器会将其递减，而不管其花了多少时间转发该IP封包，因此此字段实际上代表的是简单的跳点计数。假定每个路由器在转发封包时都应该递减此字段，当TTL为零时，该封包就应该被丢弃。理论上，其初值（由传送者设定）取决于所携带的有效载荷的类型。有效载荷对终端对终端的延迟愈敏感，TTL的值就应该愈小。然而，多数时候都是使用默认值64（参见*include/linux/ip.h*）（注4）。封包不会无声无息地被丢弃：发送方会收到一条ICMP（Internet Control Message Protocol，因特网控制消息协议）消息作为警示。

协议（Protocol）

此字段代表较高层（L4）的协议标识符。文件*/etc/protocols*（注5）包含了一份

注4：实际上，默认值取决于封包是否为多播。多播IP封包的TTL默认值为1（可以用setsockopt系统调用进行修改）。

注5：注意，这个文件不是内核的一部分，而是包含在所有Linux发行套件中。

部分列表。更多资料可以在*http://www.iana.org/numbers.html*找到。第二十四章我们会说明IP层如何利用此字段把入口封包交给正确的协议处理函数。

报头校验和（Header Checksum）

确保IP报头在传输之后依然准确。但不包含封包的有效载荷；必要时，由L4协议负责检查内容。

源地址（Source Address）

目的地址（Destination Address）

源（发送者）和目的地（接收者）的IP地址。

选项（Options）

包含下一节要讨论的可选信息。此字段可以是空的，也可以长达40字节。其大小是报头长度减去20（20就是没有“选项”的IP报头的大小）。最大值是40，因为报头长度是一个4位的值，而且以32位（4字节）为单位表示报头大小。4位中可以表示的最高值是15，而15乘上4字节是60字节。因为20字节已被基本IP报头用掉了，因此，“选项”可用的就只有40字节了。

IP选项

如本章稍早所述，网络协堆栈必须实现一些IP选项，让应用程序想用时可以使用。为了能容纳与那些选项相关的信息，基本的20字节IP报头又另外扩充了40字节。

大多数IP选项都很少被用到，而且只在特定情形下会被用到。各种不同选项可以结合至相同IP封包。然而，除了End of Option List和No Operation选项之外，每个报头中每个选项最多只能有一个实例。这些选项的存在也会影响分段/重组流程，这到了“封包的分段/重组”一节就会知道。

有些选项很简单，可以用单一字节指定；比较复杂的选项就需要更为灵活的格式，称为多字节选项。

图18-4所示是这两种选项的格式。注意，多字节选项内的`option_data`并没有从32位边界起算。

每个选项都有一个8位字段，名为`type`，可以将其进一步拆成三个子字段，如图18-5所示。`type`字段最常见的值如表18-1所示（注6）。此表显示出Linux内核为这些选项所用的符号，以及符号的值如何拆成图18-5所示的三个字段。

注6：　更多详细内容参见*http://www.iana.org/assignments/ip_paramemts*。

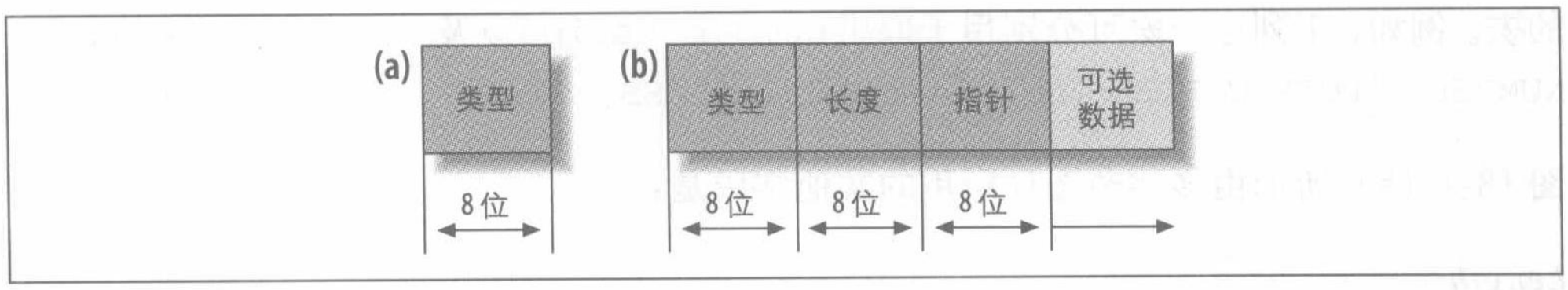


图18-4：（a）单字节IP选项格式；（b）多字节IP选项格式

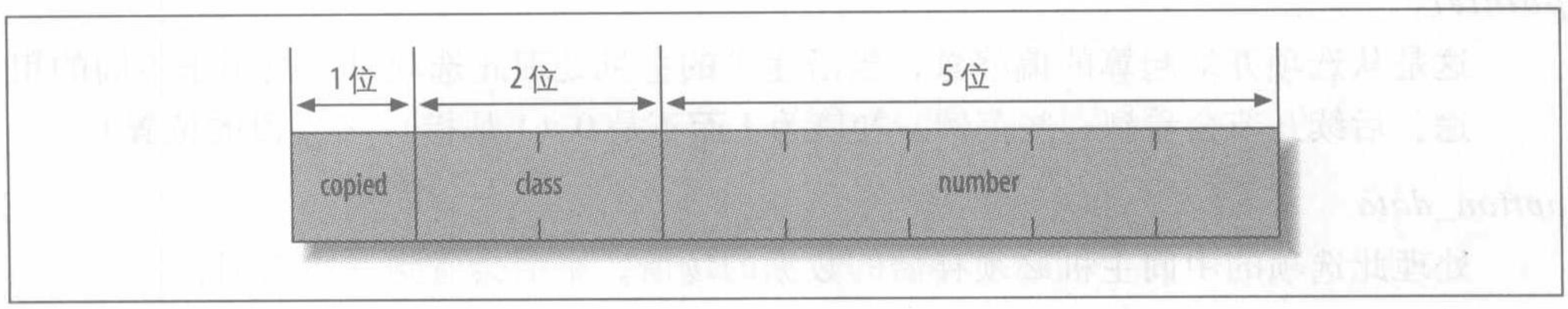


图18-5：IP选项的type字段的格式

当copied被设定时，若封包需要分段，IP层就必须把该选项拷贝至每个片段。class会根据四条准则对此选项分类；这些字段可根据IP选项过滤封包，或者把不同的QoS参数施加至这些封包。

表18-1：IP选项type字段的常见代码值

选项	内核源码中所用符号	number	copied	class（Control(00)/Reserved(01)/Measurement(10)/Reserved(11)）
End of Options List	IPOPT_END	0	0	Control
No Operation	IPOPT_NOOP	1	0	Control
Security	IPOPT_SEC	2	1	Control
Loose Source and Record Route	IPOPT_LSRR	3	1	Control
Timestamp	IPOPT_TIMESTAMP	4	0	Measurement
Record Route	IPOPT_RR	7	0	Control
Stream ID	IPOPT_SID	8	1	Control
Strict Source and Record Route	IPOPT_SSRR	9	1	Control
Router Alert	IPOPT_RA	20	1	Control

在*include/linux/ip.h*里可以找到这些选项类型的定义，以及一些可以用于存取其子字段

的宏。例如，下列三个宏可分别用于取出number、copied以及class子部分：IPOPT_NUMBER、IPOTP_COPIED（注7）以及IPOPT_CLASS。

图18-4（b）所示由多字节选项所用的其他字段是：

Length

选项的长度（单位是八字节），包括type和length。

pointer

这是从选项开端起算的偏移量，当沿途上的主机处理此选项时，被用于不同的用途。后续几节会看到一些范例。初值为1而不是0（1是指type字段的位置）。

option_data

处理此选项的中间主机必须存储的数据的场所。稍后会看到一些范例。

后续几个小节中，我们将会说明由Linux所处理的表18-1中的那些选项是怎么运作的。

End of Option List和No Operation选项

不含选项的IP报头的大小是20字节。当IP选项的大小不是4字节的倍数时，传送者会以IPOPT_END选项为IP报头补白，使其对齐4字节边界。这是必要的，因为IP报头的“Header Length”字段是以4字节的倍数表示的。

IPOPT_NOOP选项可用于填补选项之间的空白，例如，为了让后续的IP选项对齐特定边界。第十九章就会知道，Linux如何利用此选项作为从IP报头中删除选项的简便方式。

“Source Route”选项

来源地路由（Source routing）可让传送者指定封包传至其接收者时所走的路径。有一种来源地路由可以在L2和L3层上使用；这里我要谈L3层的实现细节。

Source Routing是一个多字节选项，在此选项中，发送节点会列出后续跳点所用的IP地址。当然，如果列表中的路由器之一宕机，由来源地路由的封包就无法受益于路由协议所做的动态路由。通常，当路由器宕机时，较高层协议会计算出新的来源地路由，然后重送该封包。但有时无法指定新路径，这也许是为了安全上的考虑。

来源地路由有两种类型：严格的（strict）和松散的（loose）。在严格来源地路由中，传送者必须列出路径上每台路由器的IP地址，而且沿途都不能做修改。在松散来源地路由

注7： 第十九章“IP选项”一节将探讨ip_forward_options如何使用IPOPT_COPIED。

中，一台中间路由器可以使用另一台路由器（不在列表之中），作为通向列表中下一个路由器的路径。然而，传送者所指定的所有路由器还是必须按其指定的次序使用。

例如，以图18-6所示的网络和路由器为例。假设主机X想使用Strict Source Routing选项把封包传给主机Y。这时，主机X必须指明所有居中路由器的地址。严格来源地路由的范例之一就是R_1 R_2 R_3 主机Y。以松散来源地路由而言，像R_1 R_3这样的路径就够了。使用非相邻路由器（如本例中R_1和R_3）是允许的，而且有些优点：如果R_2失败了，就可改用R_2b，反之亦然。

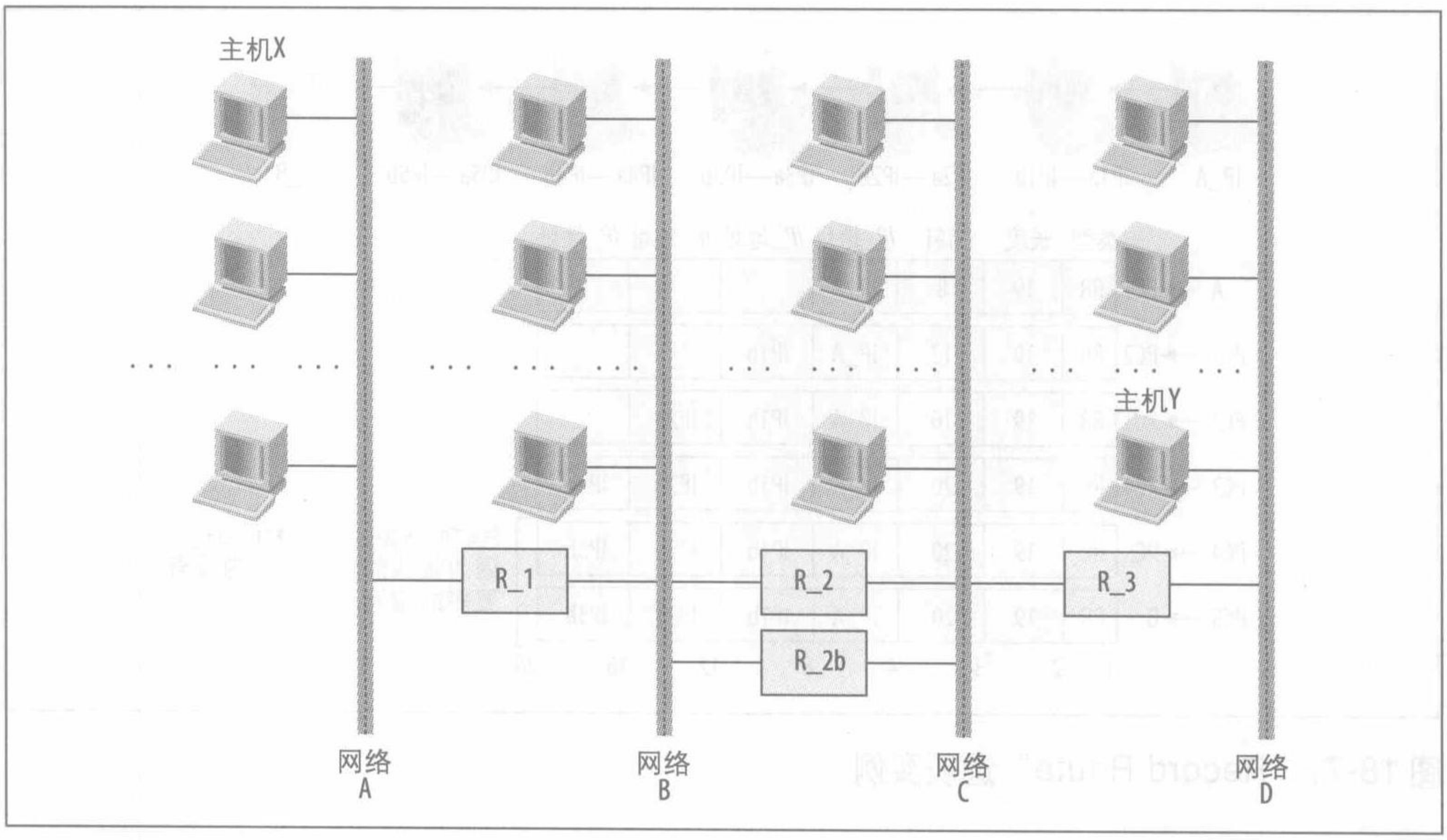


图18-6：IP来源地路由实例

Record Route选项

这个选项的目的是请求发送方和目的地之间的路由器存储它们用于转发封包时所用的外出接口的IP地址。因为报头的空间有限，因此最多只能存储9个地址（如果报头中包含其他选项，能存的更少）。因此，封包抵达时，会在该选项中存储前9个（注8）地址；接收者无法得知在那之后的路由器用的是什么了。因为这个选项会让报头（因此，连同IP封包）沿路增长，而且因为其他选项可能也会存在于报头中，所以传送者应该保留即

注8：(40 − 3)/4=9。其中，40是IP选项的最大尺寸，3是选项报头的尺寸，而4是一个IPv4地址的尺寸。

将用于存储这些地址的空间。如果封包抵达其目的地之前保留空间就已经满了，即使IP报头的最大尺寸可以容纳其他地址，其他地址也无法再加进来。当没有空间存储新地址时，不会产生错误（ICMP消息）。显然，传送者应该保留的空间量是4字节（一个IP地址的大小）的倍数（注9）。

图18-7所示是此选项的IP报头部分随着跳点而改变的情况。当每个路由器填写其地址时，也会更新pointer字段，使其指向选项中数据的尾端。图18-9底端的那些偏移量是从1开始，而你可以拿这些偏移量和pointer字段的值互相比较。

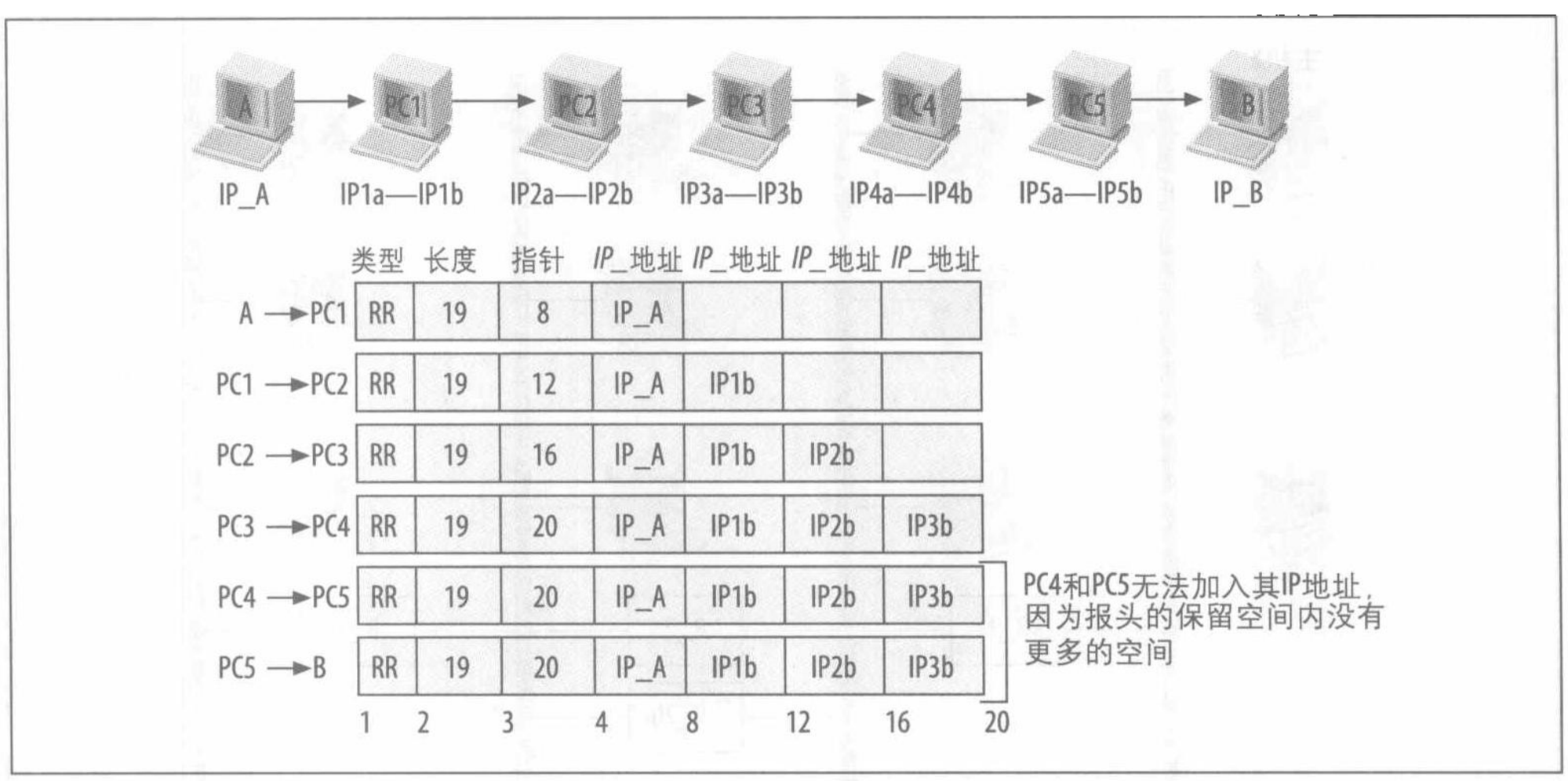


图18-7：“Record Route”选项实例

“Timestamp”选项

这个选项是最复杂的一个选项，因为它包含了子选项，此外和Record Route选项不同的是，它还处理溢出的问题。为了管理这两个多出来的概念，这个选项在其报头中还需要其他字节，如图18-8所示。

前三个字节的意义和其他选项的都相同：type、length以及pointer。第四个字节实际上分成两个4位的字段。最右边的4位（低4位）代表的是可以改变此选项效应的子命令。其可能的值如下：

注9： length的值并非刚好是4的倍数，因为选项报头（type、length以及pointer）是3字节长。也就是说，32位IP地址会在32位字（word）边界上不自然地断开。

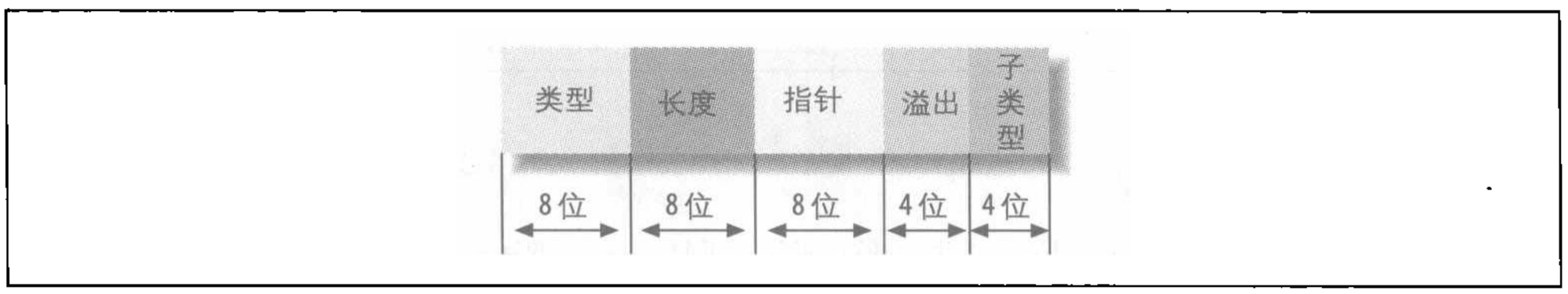


图18-8：IP“Timestamp”选项报头

RECORD TIMESTAMPS

每个路由器会记录其收到封包时的时间。

RECORD ADDRESSES AND TIMESTAMPS

类似于前一个子命令，但是也会存储接收接口的IP地址。

RECORD TIMESTAMPS ONLY AT THE PRESPECIFIED SYSTEMS

每个路由器在接收到封包时都会记录接收时间（如同RECORD TIMESTAMPS），但是只针对传送者选择的一些特定IP地址。

在这三种情况下，时间是以毫秒计，从当天午夜算起（UTC时间），而且是存储在一个32位变量内（注10）。

另外4位代表的是所谓的overflow字段。因为TIMESTAMP选项用于记录路径沿途上的信息，并且因为IP报头中可供此用的空间只限于40字节，因此，会发生因空间不足而路由器无法记录信息的情况。尽管Record Route选项的处理方式就是直接忽略那种情况，使得接收者对其发生次数毫无所知，但是对TIMESTAMP选项而言，每次发生这种事时就递增overflow字段。可惜，overflow只是4位字段，因此最大值为15：在现代网络上，这个字段本身就很容易溢出。当发生这种情况时，产生溢出的路由器就必须返回一个ICMP参数错误消息给原来的传送者。

前两个子选项很类似（差别只在于每个跳点上要存储什么），而第三个子选项就略有不同，值得多说几句。封包原本的发送者会列出其感兴趣的那些IP地址，而每个IP地址都有4字节的空间。在每个跳点上，该选项的pointer字段指的就是下个4字节空间的偏移量。每个出现在此地址列表中的路由器会在适当空间中填入时间戳，然后更新pointer字段，参见图18-9所示。此图顶端序列中有下划线的就是加入了时间戳的主机。此图底端的偏移量从1起算，而你可以拿这些偏移量和pointer字段的值互相比较。

注10： UTC代表Universal Time Clock（全球时钟），也称为GMT（Greenwich Mean Time，格林威治标准时间）。

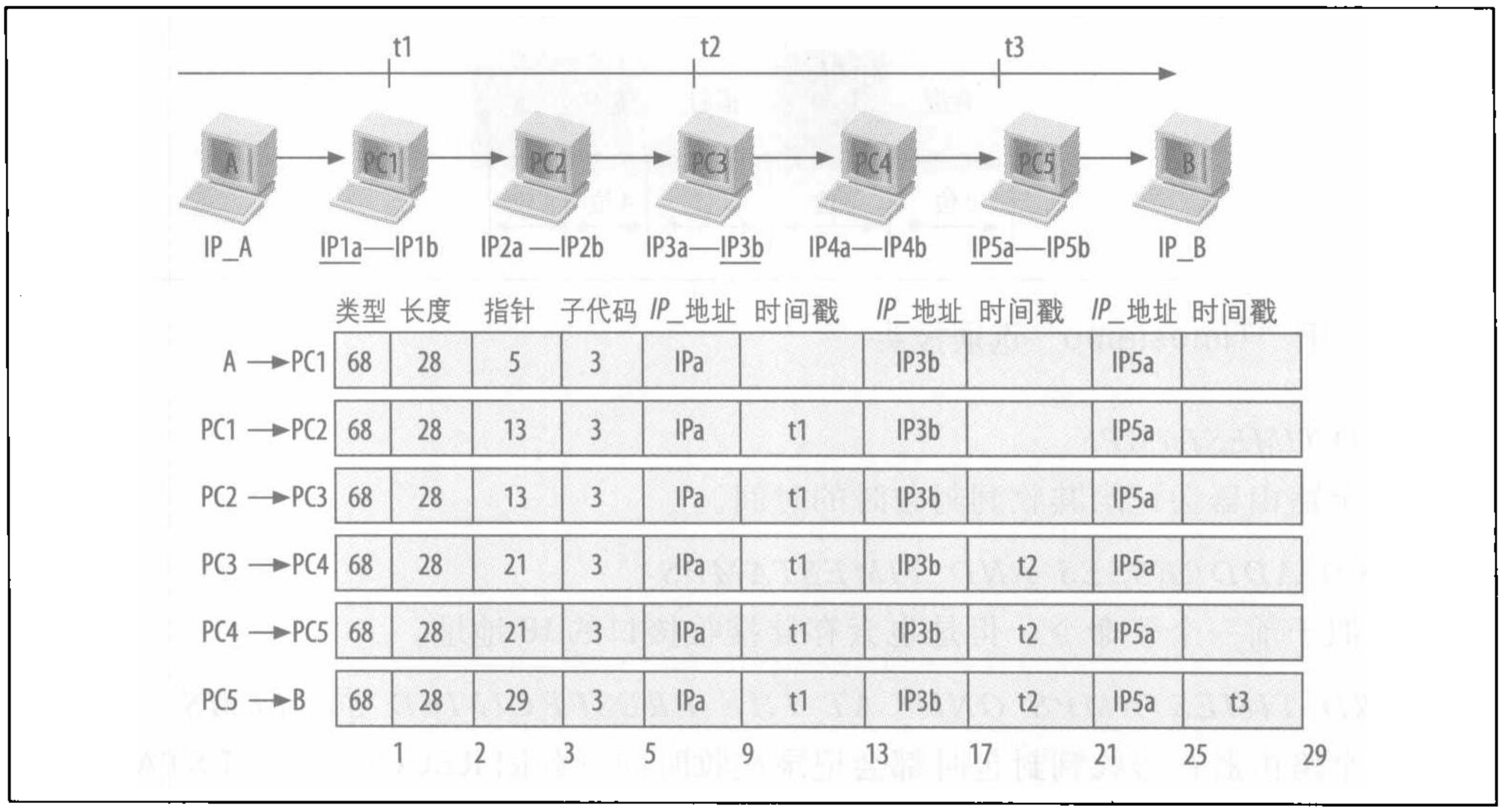


图 18-9：针对“PRESPECIFIED SYSTEMS”存储“Timestamp”选项的范例

Router Alert 选项

这个选项是在 1995 年加入 IP 协议的定义，而且在 RFC 2113 中有所说明。这个选项会把封包标记成需要特殊处理（除了查看目的地地址和转发封包之外）。例如，RSVP（Resource Reservation Protocol，资源保留协议）—— 它试图为封包数据流建立更好的 QoS —— 会使用此选项告诉路由器，必须以特殊方式对待该串流里的封包。现在，最后两个字节只有一个指定值，也就是零。这仅意味着路由器应该检查该封包。携带其他值的封包是非法的，而且应该被丢弃，同时产生 ICMP 错误消息返回给产生这种封包的发送方。

封包的分段 / 重组

封包的分段（fragmentation）和重组（defragmentation）是 IP 协议的主要工作之一。IP 协议把封包的最大尺寸定义成64 KB，而这样的事实是来自于报头中表示封包尺寸的len 字段是 16 位值（以字节为单位表示）。然而，没有多少接口类型可以传送高达 64 KB 大小的封包。也就是说，当 IP 层必须传输尺寸大于出口接口的 MTU 的封包时，必须把封包切成几块较小的单位。本章稍后就会知道，所用的 MTU 不见得就是和出口设备相关的 MTU；例如，有可能是和用于路由封包的路由表项目相关的 MTU。路由表项目所用的 MTU 取决于多个因素，而其中之一就是出口设备的 MTU。

无论MTU是怎么计算的，分段流程会建立一系列尺寸相等的片段（fragment），如图18-10所示。图中所示的MF和OFFSET字段会在本节稍后说明。如果MTU无法刚好除尽封包尺寸，最后的片段就会比其他的都小。

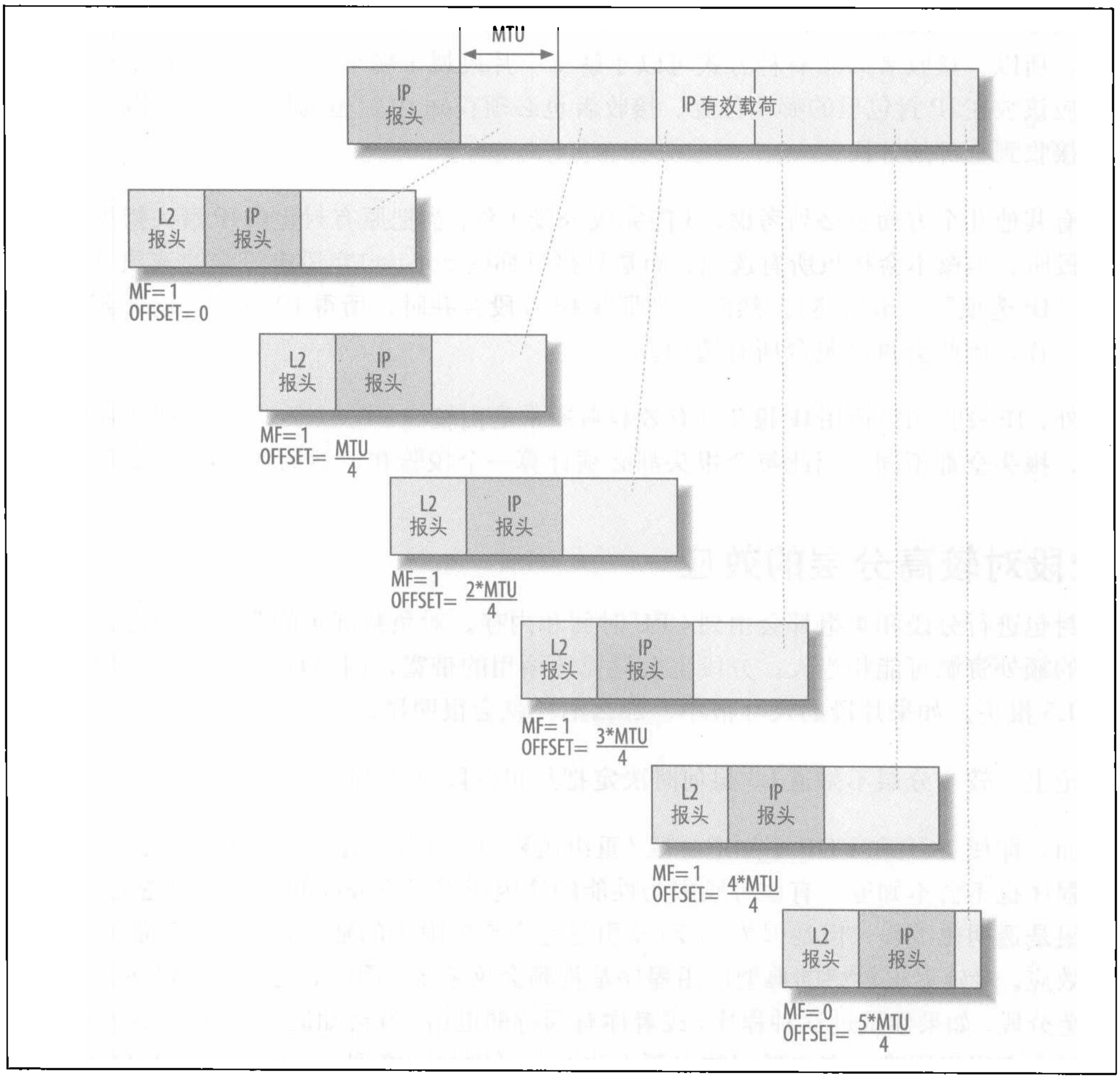


图18-10：IP封包分段

分段后的IP封包通常会由目的地主机重组，但是，必须查看整个IP封包的中间设备也得重组。这类设备的两个例子是防火墙和NAT（Network Address Translation，网络地址转换）路由器。

以前，有个可以接受的方案是接收者以原IP封包的尺寸分配一个缓冲区，然后把那些到达的片段放在那里。事实上，接收者可能只会分配一个最大可能尺寸的缓冲区，因为原

始IP封包的尺寸只有在接收到最后一个片段时才会知道。现在，这种简单做法已经不再使用了，因为这会浪费内存，此外，恶意的攻击会让路由器不能正常工作（传送大量的极小片段，并谎报其原始尺寸）。

因为每个IP封包都可以被分段，而且因为每个片段在路径上也会因相同原因再度被分段，所以，接收者必须有种方式可以了解每个片段属于哪个IP封包，以及每个片段原本应该放在IP封包里的哪个位置。接收者也必须获知IP封包的原始尺寸，以了解它何时接收到全部的片段。

还有其他几个方面也必须考虑，才能完成分段工作。当把原有封包的IP报头拷贝到它的片段时，内核不会拷贝所有选项，而是只拷贝那些copied字段有设定的选项（如稍早于“IP选项”一节所述）。然而，当那些IP片段合并时，所得IP封包看起来会和原有的一样，因此会再度包含所有选项。

此外，IP校验和只适用IP报头（有效载荷通常是由较高层协议负责）。当那些片段建立时，报头全都不同，因此每个报头都必须计算一个校验和，然后在接收端校验。

分段对较高分层的效应

对封包进行分段和重组都会用到CPU时间和内存。对负载沉重的服务器而言，牵涉其中的额外资源可能相当大。分段也会耗掉传输用的带宽，因为每个片段都必须包含L2和L3报头。如果片段的尺寸很小，那么消耗就会很明显。

理论上，较高分层不知道L3层何时决定把封包分段（注11）。

然而，即使TCP和UDP不知道分段/重组流程（注12），建立在这两个协议之上的应用程序也不会不知道。有些时候因为性能的原因还得考虑分段的问题。理论上，分段/重组是透明化流程，但是因为分段/重组总是会产生附加的延迟，所以对性能总是有负面效应。对延迟很敏感的典型应用程序是视频会议系统，因此，这类应用程序会尽可能避免分段。如果你试过这种程序，或者你有因特网电话，就会知道延迟太长是指什么了：交谈会变得很困难。有些延迟的来源无法避免（例如网络拥挤，缺乏可靠的QoS），但是如果有办法可以减少延迟，应用程序就会多采取几个步骤去做那些事。许多应用程序都很聪明，会试着通过考虑下列因素去避免分段：

注11：第二十一章“ip_append_data函数”一节将说明L3和L4间的接口如何演进，以对本地产生的封包的分段工作做最优化处理。

注12：到了第二十一章“整合传输函数”一节就会知道，L4协议实际上提供了一些可以影响分段的选项。

- 首先，内核不见得只能使用出口接口的MTU，也可以使用一个名为路径MTU发现（path MTU discovery）的功能，以发现可以使用的最大封包尺寸，以避免在特定路径上的分段（参见“路径MTU发现”一节）。
- MTU可以设成一个相当安全的小值576。这反映出RFC 791中的规范：每台主机必须足以接受多达576个8字节大小的封包。对封包尺寸做这样的限制，就可大为减少分段的可能性。很多应用程序最后的默认行为都是使用此MTU（如果没有明确配置成使用不同的值）。

当传送者决定使用比其可用的MTU更小的封包尺寸以避免分段时，还是必须承担像分段的额外报头所需的处理相同的问题。然而，沿路上让路由器避免分段就能大量减少路径上的处理，因此，避免分段对改善响应时间而言就成了关键。

分段/重组所用的IP报头字段

以下是IP报头中用于处理分段/重组流程的字段。第二十二章将会探讨这些字段如何使用。

DF（Doilt Fragment 不分段）

有些情况下，分段对那些上层是坏事。例如，交互式流媒体如果受到分段，就会导致相当糟糕的性能。而且某些时候，传输者知道接收者只有简单、轻量级的IP协议实现，因此无法处理重组。就此而言，IP封包报头中提供了一个字段以指出是否容许分段。如果路径上的封包超出某个连结的MTU，就会被丢弃掉。“路径MTU发现”一节会提到这个标志和路径MTU发现有关的用法。

MF（More Fragmerts, 更多片段）

当一个节点把一个封包分段时，在每个片段中将此标志设为TRUE（最后一个除外）。当接收者接收由此封包所建的最后一个片段时，就会知道原本没分段的封包的尺寸（即使有些片段尚未接收到）。

片段偏移量

这是代表该片段要放在原本IP封包中的位置偏移量。这是一个13位字段。因为len是16位字段，所以，片段的建立始终要在8字节边界上，而且此字段的值会读成8字节的倍数（也就是左移3位）。偏移量0是指该片段是此封包内第一个片段；该项信息很重要，因为第一个片段内包含和整个原有封包相关的报头信息。

ID

即IP封包ID，它对一个IP封包的所有片段而言都是相同的。根据此参数，接收者知道该重新组合哪些片段。第二十三章“长效IP端点信息”一节将会知道此字段的值是如何选择的。Linux会把最近一个所用的ID存储在一个名为`inet_peer`的结构中；而且在此结构中，Linux存储了其正在通信的远程主机的信息。

分段/重组的问题实例

分段是很简单的流程：节点只需选出适当的值以匹配MTU。大多数议题都和重组有关，这并不令人讶异。在下面两小节中，我们要谈及两种最常见的议题：正确处理重新传输和重新组合封包，以及NAT的特殊问题。

另一个不用分段的原因是和阻塞控制算法不兼容。

重新传输

稍早我曾说过，IP封包彻底重组之前，它将无法传递给下一个较高分层。然而，这不表示那些片段会毫无限制地放在主机内存内。否则，只要通过简单的拒绝服务（Denial of Service，DoS）攻击，就可以让主机停止响应。一个片段没被接收到的原因有好几个：例如，路上被一个因挤满而没有内存可供存储的路由器给丢弃了，或者可能损毁了，然后做CRC（错误检查）时就被丢弃了，或者被防火墙挡着，因为防火墙在转发任何片段前想先看一看第一个片段里的报头。因此，每个路由器和主机都有一个定时器，如果某些片段在特定时间内没接收到，就会清理对应的那个IP封包的那些片段所用的资源。

如果传送者知道有个片段丢失了或者路上被丢弃了，如果传送者可以重传丢失的片段，那就太棒了。但是，这是完全不可行的。传送者甚至无法知道路上会有一个路由器把其封包给分段了，更别提他会知道是哪些片段了。所以，每个传送者只会等待，直到较高分层通知其重传整个封包。

重传的封包不会重用原先的ID。然而，主机还是有可能接收到好几个具有相同封包ID的同一个IP片段的副本，所以主机必须能处理这种情况。注意，即使没有重新传输，同一个片段也可能会被接收数次：常见的范例是当L2层有个回路时。我们会在本书第六部分说明这种情况。这种浪费就是另一个避免在发送方使用分段的良好原因，而且如果延迟对应用程序（例如视讯会议软件）不是好事，就要试着使用能把路上被分段的可能性降到最低的封包尺寸。

因为内核无法将其数据交换到磁盘上（只交换用户空间数据），为了处理片段而造成的内存浪费对路由器的性能有很大的冲击。Linux对片段能用的内存量可设限制，如第二十三章“通过/proc文件系统做调整”一节所述。

由于IP是一种无连接协议，因此没有所谓的流量控制，而且完全由上层协议（或应用程序）负责数据的丢失。当然，有些应用程序不在乎数据丢失，但有的会。

我们假设上层以某些方式（例如，因为没收到确认通知消息，一个定时器到期了）检测到某些数据丢失了，然后试着重新传输。因为不可能选择性地只重发丢失的片段，L4

协议得重传整个IP封包。每次重新传输都可能引发一些特殊情况，使得接收者这一端必须处理（当接收者实现了某种防火墙机制使得封包必须重组时，偶尔会由中间路由器负责处理）。以下是一些例子：

重叠

一个片段内含的某些数据可能已经在前一个封包内抵达了。重传的封包有不同的ID，因此，其片段不应该和前一次传输的片段混在一起。然而，有问题的操作系统可能没为重传的封包使用不同的ID，或者下一节会介绍的卷绕问题（wraparound problem）都使得重叠有可能发生。

重复

这可以视为重叠的特殊情况，其中两个片段完全相同。如果两个片段的开始偏移量相同，而且有相同的长度，就视为重复。但是，对实际的有效载荷内容并未作检查。除非你陷入安全攻击之中，否则，同一个封包重传时，没有理由有效载荷内容会变动。先前所提的L2回路也可能是重复的来源。

重组完成后又接收到

就此而言，IP层会认为此片段为新IP封包的第一个片段。如果所有新片段都没接收到，IP层就会在其垃圾收集流程中清掉重复片段；否则，就会重建整个封包，不过是否将此封包视为重复就是上层协议的事了。

如果考虑到片段还可以再分段的话，事情就更复杂了。

把片段和其IP封包相关联

因为那些片段抵达的次序可能很乱，所以，重组是很复杂的流程，因为必须认出每个封包，而且当封包抵达时将其放在适当地点。插入、删除以及合并操作都必须简单而快速。

要识别一个片段所属的IP封包，内核会考虑下列参数：

- 发送方和目的地IP地址
- IP封包ID
- L4协议

可惜，不同封包可能会共享所有这些参数。例如，两个不同的传送者可能碰巧为封包选择了相同的封包ID，而且封包又碰巧同时抵达。你可能以为发送方IP地址可以区分那些封包，但是，如果两台主机都位在NAT路由器之后，而NAT路由器又将其自己的IP地址放到封包上，那该怎么办？接收端的IP层无法在这些情况下区分那些片段。你也不能仰赖IP ID字段，因为这是一个16位字段，因此在一个高速网络上很快就会卷绕。

因为IP ID字段在重组流程中扮演关键角色，来看看IP片段在内存中的组织方式以及IP ID是如何生成的。IP ID生成器最简明的实现方式就是递增一个全局计数器，然后，每次IP层要传送一个封包时就以它作为ID。这样可以确保ID值循序渐近而且容易实现。然而，这种简单模型有些问题：

- 就所有可能共享一个全局ID的较高层协议而言，需要某种上锁机制（特别是在多处理器机器上），以防止出现竞争（race condition）。然而，使用这种锁会限制SMP的可伸缩性。
- ID可预测的话，会使得一些常见手法得以攻击机器。
- ID值很快就会卷绕，结果造成重复的ID。因为ID字段是16位值，只能有65,535个独一无二的数字，而拥有高流量和快速联机的节点可能会发现旧封包抵达其目的地前，就得让新封包使用相同ID了。例如，以平均512字节的封包尺寸而言，GB速率的接口会在半秒内传送65,535个封包。高负载的服务器在1s内就会卷绕一个全局IP ID计数器。

因此，我们必须接受IP层偶尔会把来自完全不同的封包的数据混在一起的可能性。这样有一些问题。只有较高层可以解决这个问题——通常是通过错误检查。

下面一节介绍Linux减少（但未解决）卷绕问题和ID可预测性的可能性的一种方式。在第二十三章“选择IP报头的ID字段”一节会介绍精确的算法和程序。

IP ID生成范例

通过数个同时运行的全局定时器，卷绕问题得以部分解决。Linux内核不是使用一个全局IP ID，而是为每个目的地IP地址都持有一个不同的计数器（直到可能的IP目的地达到的最大数目）。注意，通过使用数个IP ID，可以让ID拖久一点才卷绕，但是它们终究还是会卷绕。

图18-11所示是一个范例。假设我们有数据要传给地址IP1和IP2的两台服务器。还假设对每个IP地址而言，我们有不同而独立的数据流，例如HTTP、Telnet以及FTP。因为通往相同目的地的所有数据流都共享IP ID，如果你把通往目的地的数据看成一个整体，封包就有连续的ID，但是每个应用程序的数据流则不会有连续的ID。例如，由Telnet会话所产生的通往目的地IP1的IP封包就不是连续的。注意，这只是Linux所采用的解法，并非一种标准。其他替代方案也可用。

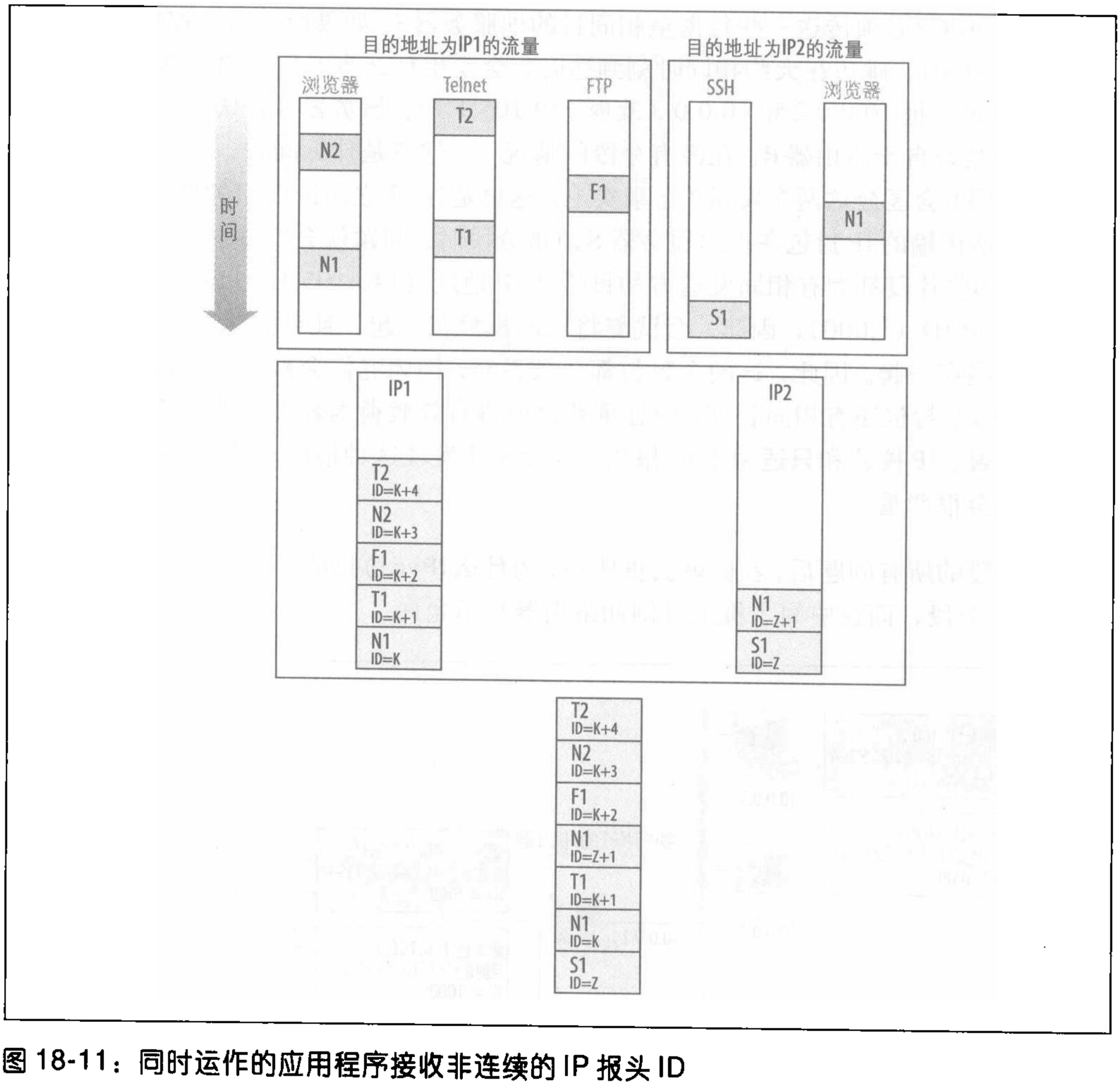


图 18-11：同时运作的应用程序接收非连续的 IP 报头 ID

无法解决的重组问题范例：NAT

尽管IP层尽了该尽的本份，分段规则还是有可能引发IP层无法解决的情况。图18-12所示是其中一例。假设R是路由器，为其网络上所有主机做NAT。更精确地讲，假设R做了伪装（注13）：如果要传至因特网时，内部网络中的主机所产生的IP封包的报头中的发送方IP地址就会被路由器R的IP地址140.105.1.1给替换掉（注14）。

注 13：Linux 所谓的伪装，也时常称为端口地址转译（Port Address Translation，PAT）。

注 14：注意，对于从因特网传回、要给内部网络主机的流量，其目的地IP地址都是140.105.1.1，而R会使用目的地UDP/TCP端口号码找出正确的主机，然后把输入流量传送过去。就我们的范例而言，我们不会去谈这种端口的事是怎么做到的。

假设PC1和PC2必须传送一些数据至相同目的地服务器S。如果两个封包拥有相同IP ID（此例为1,000），碰巧在大约相同时刻到达时，会发生什么事？因为路由器R重写了发送方IP地址（把10.0.0.2和10.0.0.3改成140.105.1.1），服务器S会认为其所接收的两个IP封包是来自于路由器R。在没有分段的情况下，这不是什么问题，因为L4信息（例如端口号码）会区分这两个发送方。事实上，这也是NAT之所以能用的原因。但是，当两个由R所传输的IP封包在抵达服务器S前被分段时，问题就会产生。本例中，服务器S接收的那些片段都拥有相同发送方和目的地IP地址（140.105.1.1 151.41.21.194）以及相同的IP ID（1,000），因此，会试着将它们都放在一起，且可能把两个不同的IP封包的片段混在一起。因此，这两个封包都会被丢弃，因为它们会被视为损毁。最糟的情况下，这两个封包还有相同长度，而且重叠会使得有效载荷内容被损毁，但是，L4报头却没被损毁。IP校验和只适用于IP报头，因此无法发现这种情况。对于某些应用程序，结果可能会很严重。

列举出分段的所有问题后，我们就会更明白，为什么IPv6协议的设计决定只在源头主机上允许IP分段，而在中间主机上（例如路由器）不允许。

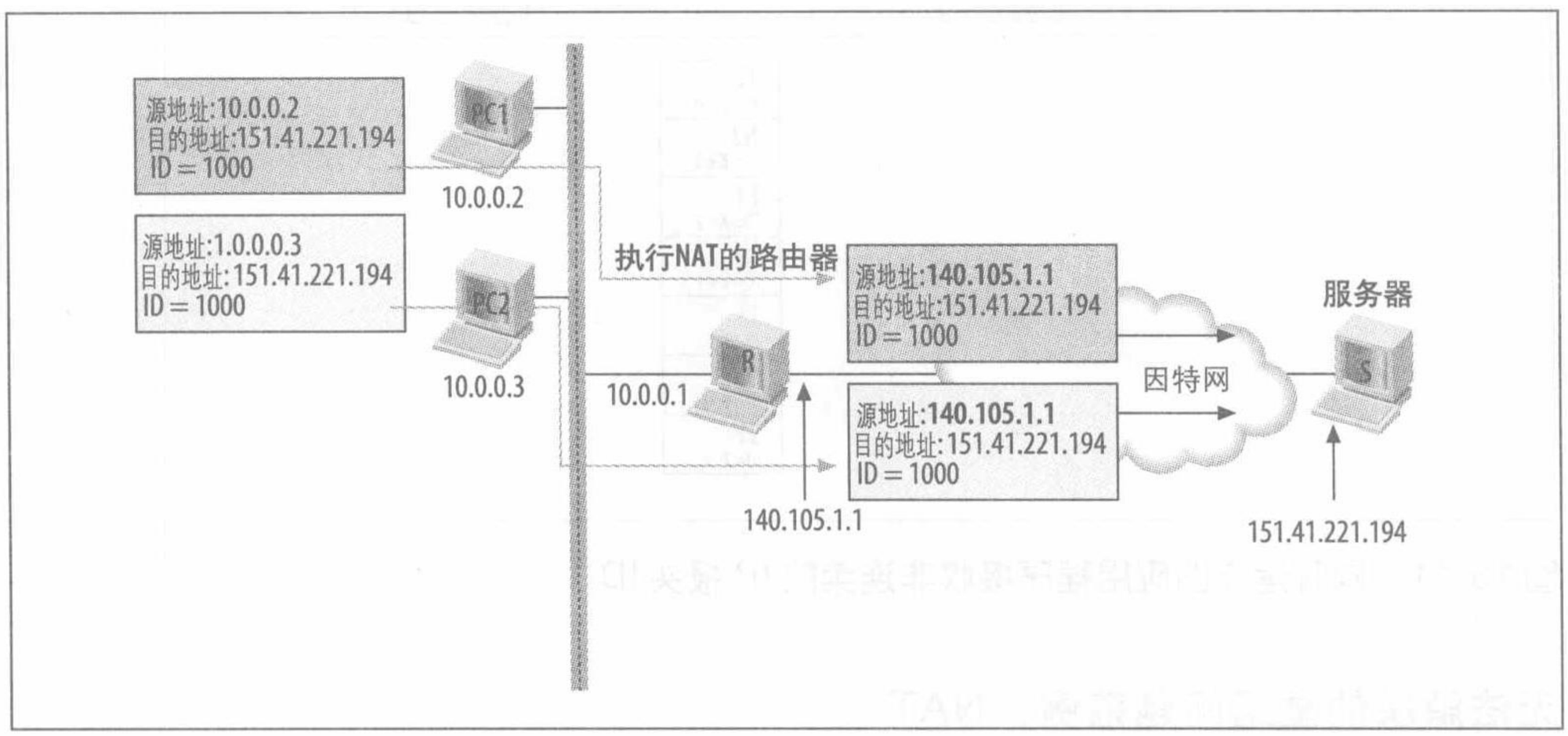


图18-12：NAT和IP分段惹麻烦的范例

路径MTU发现

对封包分段的缺点做了冗长的讨论之后，读者就可以欣赏我要讨论的下一个IP层功能：路径MTU发现。

当我在第二章讨论net_device数据结构时，曾列出了最常见接口类型的MTU。MTU的范围就是该网络接口所连接的LAN。如果你把IP封包传输给同一个LAN（以此作为

传输接口）的另一台主机，当封包的尺寸大于LAN的MTU时，IP封包就必须被分段。然而，如果你选的尺寸适合MTU，就可以确保不会发生分段。当目的地主机不在直接相连的LAN上，就无法依据LAN的MTU推导出是否会发生分段。这时就轮到路径MTU发现功能出场了。

路径MTU发现用于发现封包传输至指定目的地地址而不用被分段的最大尺寸。该参数称为路径MTU（Path MTU，PMTU）。基本上，PMTU就是从一台主机到另一台主机间的路径上沿途所碰到的所有连接中最小的MTU。

因为两个终端间的路径可能是不对称的，因此，对任何一对给定主机而言，都可能有两个不同的PMTU。每台主机会计算并使用适当的PMTU，以传送封包至另一台主机。此外，路径的改变也会导致PMTU的改变。

由于每个目的地IP地址会使用不同的PMTU，其值会暂存在相关的路由表缓存项目中。本书第七部分就会知道，路由表中的路径可以聚集好几个IP地址；例如，你可以有一条路径，说网络10.0.1.0/24可通过网关10.0.2.1连接。另一方面，路由表缓存区中会为最近以来该主机所连过的每个目的地IP地址都记录一个项目（注15）。因此，你可能会有主机10.0.1.2的一个项目，10.0.1.3也有另一个项目（虽然这两台主机可通过相同网关连接）。这些项目每一个都包含一个唯一的PMTU。你可能会说，如果属于两台主机的两个地址位于相同的LAN之中，第三台主机可能会使用相同路径连接这两台主机，因此，可共享相同的PMTU。路由表中只保留一个PMTU就足够了。可惜，这是不可能的。只是因为一条路径可用于连接一群地址，并不见得意味着这些地址都属于相同LAN。路由是复杂的主题，我们会在本书第七部分讨论其中的几个方面。

每个路由表项目都会和一个出口设备相关（注16）：也就是用于把数据传输给路径上下一个跳点的设备。如果该设备直接连接至其通信对象，而且PMTU发现功能有开启，则默认，PMTU就会设成该出口设备的MTU。

直接连接的设备包括电缆线的两个终端点或者一个Ethernet LAN上的一些设备。LAN上的所有设备（彼此间没有路由器）共享相同MTU以正确运作，这一点特别重要。

如果设备没有直接连接（也就是如果设备间至少有一台路由器），或者如果PMTU功能被关闭了，则PMTU默认是设成576。这不是随机值，而是定义在最初的IP RFC 791

注15：更准确地讲，一个路由表缓存项目会与几个参数的组合相关联，包括源IP地址、目的IP地址以及IP TOS。

注16：第三十一章就会知道，如果让内核支持多路径路由，就可以用好几个下个跳点来定义路径，而每个跳点都可能由不同接口联系。

中（注17）。无论默认值为何，管理员可以通过用户空间配置程序（例如*ifconfig*）设定初始PMTU。

我们来看PMTU发现功能是如何运作的。其算法只是利用了IP报头中用于处理分段/重组的字段以及相关的ICMP消息。

如果你传输一个报头中DF标志有设定的IP封包，而且没人抱怨，就表示沿着路径到目的地都没有发生分段，因此你所用的PMTU就没问题。但这并不表示你用的是最佳尺寸。你可能可以增加PMTU的值，但依然不会有分段发生。一个简单例子就是两个由一台路由器所连接的Ethernet LAN。在网络的两端上，MTU是1,500，但是每个LAN的主机都以576的MTU值和另一个LAN的主机对话，因为这些主机并非直接连接。这就不是优化。

如果增加封包尺寸以探测其最佳尺寸，当你越过真正的PMTU时，就会收到ICMP消息。ICMP消息中将会包括有所抱怨的设备的MTU，使得内核能以此更新本地PMTU。

Linux可以配置成以下列方式之一来处理路径MTU发现功能：

`IP_PMTUDISC_DONT`

绝不传送报头中设定了DF标志的IP封包；因此，不使用路径MTU发现功能。

`IP_PMTUDISC_DO`

总是在本地节点所产生的封包的报头中设定DF标志（不是转发的封包），试图在每次传输时都找出最佳的PMTU。

`IP_PMTUDISC_WANT`

按每条路径决定是否使用路径MTU发现功能。这是默认行为。

当路径MTU发现功能开启时，和一条路径相关联的PMTU可以随时改变，以包括具有较小最大尺寸的路由器，使得发送方会接收到ICMP FRAGMENTATION NEEDED消息（参见第二十五章有关`icmp_unreach`的讨论）。就此而言，路由缓存中所有具备相同目的地的项目的PMTU都会受到更新（注18）。参考第三十三章“过期标准”一节，了解关于路由表如何处理ICMP FRAGMENTATION NEEDED消息的接收的细节。应该注意的是，这个算法只会让PMTU缩水，不会让PMTU增大。然而，路由缓存的项目——它的PMTU如果是从入口ICMP FRAGMENTATION NEEDED消息推导得出的话——一段时间后就会过期，也就是会回到（较大的）默认PMTU值。参考本节刚才所述的细节。

注17： 更多细节建议读RFC 791、1191及2923。

注18： 通往相同目的地的路径可能不只一条，这是为了冗余或负载平衡。

当通过 *ip route* 命令新增路径时，路径的PMTU也可以手动设定。

即使开启了路径MTU发现功能，依然可能锁定当前PMTU，使其无法被更改。这在下列两个主要情况下发生：

- 当使用 *ip route* 设定PMTU时，有可能以 *lock* 关键词将它锁定。如下示例增加了一条经下个跳点网关100.100.100.1通往10.10.1.0/24网络的路径，而且把PMTU锁定在750个字节：

  ip route add 10.10.1.0/24 via 100.100.100.1 mtu lock 750

- 如果因接收到ICMP FRAGMENTATION NEEDED消息使得你要用的PMTU小于最小容许值，则PMTU会设成该最小值，然后锁住。最小值可以通过 */proc/sys/net/ipv4/route/min_pmtu* 文件进行配置（参见第三十六章“/proc/sys/net/ipv4/route目录”一节）。无论如何，PMTU不能设成低于68的值，因为这是RFC 1191第3.0节的要求（RFC 791“Fragmentation and reassembly”一节也有间接提及）。参见第三十三章“过期标准”一节。

对Linux而言，`ip_dont_fragment` 函数（第二十二章会谈）会以这里所描述的思路决定当封包超过PMTU时是否应该分段。

特定传输的PMTU的值可也会受到下列因素的影响：

- 用户空间中是否对该设备的MTU做了明确的配置。
- 应用程序是否改变了特定TCP套接字上所用的最大节段尺寸（mss）。

校验和

校验和（checksum）字段是一种冗余字段，网络协议用它发现传输错误。有些校验和不仅能检测错误，而且还可以自动修复特定类型的错误。

校验和的思路很简单。传输封包前，传送者会计算一个小而长度固定的字段（校验和），其中包含数据的某种hash值。如果数据的一些位在传输时有了改变，很有可能损毁的数据会产生不同的校验和。取决你用于产生校验和的函数而定，可靠度也有不同的等级。IP协议所用的校验和算法很简单，只是涉及总和以及一的补码而已，因此太过于贫弱而不可靠。想做更可靠的健康检查，需要依靠L2 CRC或者SSL/IPSec MACs（Message Authentication Codes，消息鉴定码）。

不同的协议会使用不同的校验和算法。IP协议的校验和只包括IP报头。多数L4协议的校验和则包括其报头和数据。

在L2（如Ethernet）做校验，在L3（如IP）做另一次校验，然后在L4（如TCP）又做另一次校验，这样似乎是多余的，因为这些校验工作通常都会用到数据的重叠部分，但是这些检查还是很有价值。错误不仅仅是在传输期间会发生，在分层间移动数据时也可能发生。此外，每个协议都要负责确保其自身的传输正确，而不能假设其上或其下的分层会把这件任务处理好。

以可能发生的复杂场景为例说明，想象一下LAN1里的PC A经因特网把数据传送至LAN2的PC B。再假设LAN1所用的L2协议使用了校验和，但是，LAN2的L2协议没有使用。至少有一个较高分层有提供某种校验形式以减少接收损毁数据的可能性，这一点很重要。

在每个协议的定义中，都会建议使用校验和，虽然这不是必要的。不过，你得承认，那些相关协议的设计如果再好一点，就可以把不同分层的协议中重叠的功能所强加的消耗给去除掉。因为大多数L2和L4协议都会提供校验工作，让L3也做这件事就不见得非做不可；基于相同的理由，IPv6已把校验工作删除。

在IPv4中，IP校验和字段是一个16位字段，包括整个IP报头（含选项在内）。校验和首先由封包的发送方计算出来，然后在前往其目的地的沿路上再由一个一个的跳点更新，以反映出每个路由器所施加的报头。更新校验和前，每个跳点必须先检查封包的健康状态（比较封包里的校验和以及本地所计算的校验和）。如果健康检查失败，封包就会被丢弃，但是不会产生ICMP：L4协议会处理此事（例如，利用定时器，如果特定时间内没接收到确认通知消息，就强迫重新传输）。

以下是一些触发校验和更新需求的情况：

递减TTL

路由器转发封包前，必须递减其IP报头中的TTL。因为IP校验和也包括该字段，原本的校验和就不再有效。第二十章“ip_forward函数”一节就会知道，TTL是由ip_decrease_ttl递减，而ip_decrease_ttl也会处理校验和之事。

封包调整（包括NAT）

牵涉到改变一或数个IP报头字段的所有功能都会强迫校验和重新计算。NAT可能是最有名的实例。

IP选项的处理

由于选项是报头的一部分，故也被包括在校验和内。因此，每次选项的处理方式使得IP报头必须有所新增（如新增时间戳）或修改时，就会迫使校验和重新计算。

分段

当封包被分段时，每个片段都有不同的报头。大多数字段依然不变，但是和分段有关的字段就不同了，例如offset。因此，校验和也必须重算。

因为IP协议所用的校验和的计算方式也是使用TCP、UDP以及ICMP所用的相同简单算法，因此，有一组通用函数可由这些协议使用。此外，也有专用函数为IP校验和做了优化。根据IP校验和算法的定义，其报头会切成一些16位字，再求其总和及一的补码。图18-13为了简洁起见只以两个16位字计段校验和作为范例。Linux没有计算那些16位字的总和，但是，确实把那些32位字甚至于64位长的字段的总和求了出来，以获取较快的计算结果（需要在总和及一的补码的计算间多一个步骤；参见下一节关于`csum_fold`的说明）。实现此算法的函数称为`ip_fast_csum`，在大多数架构上，该函数是直接以汇编语言写成的。

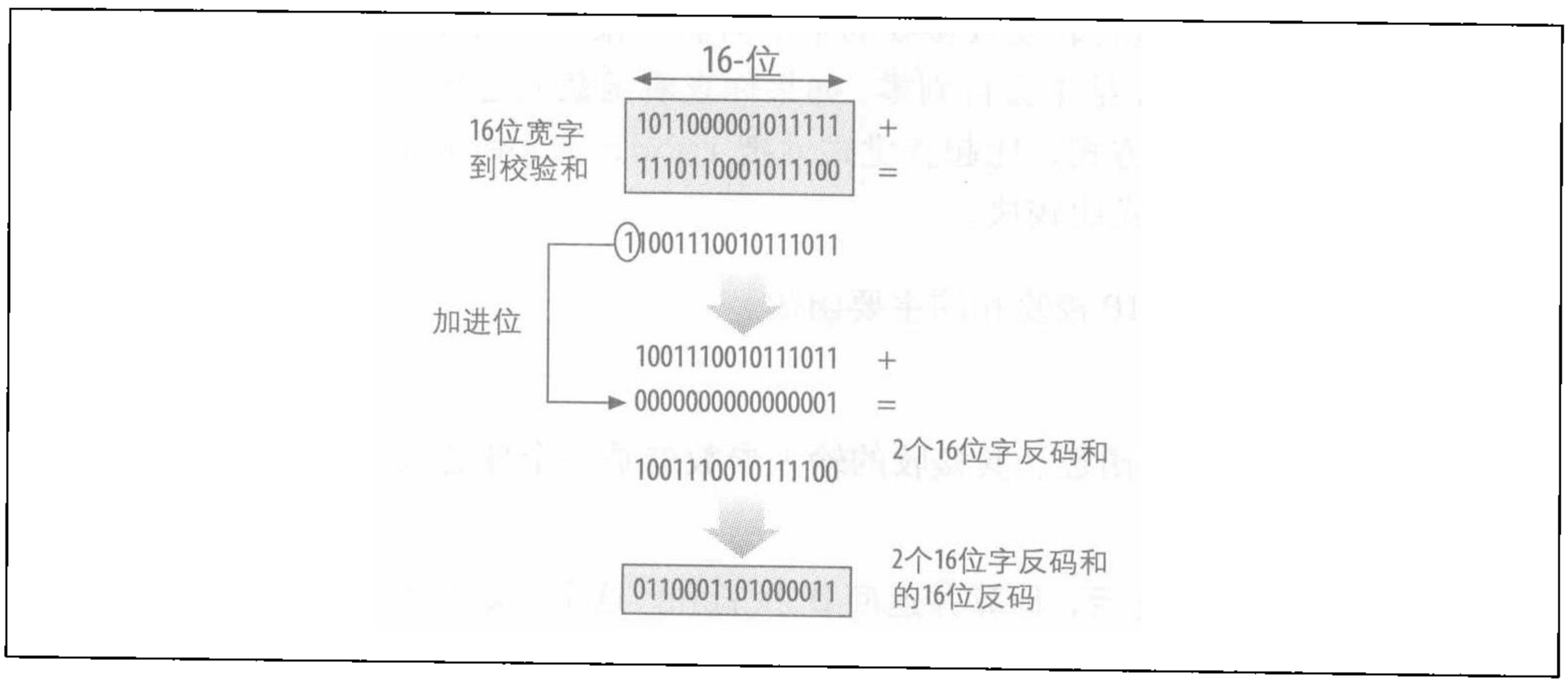


图18-13：IP校验和计算

用于校验和计算的API

L3（IP）校验和的计算比L4校验和要快很多，因为它只包含IP报头。由于这是简单的运算，所以时常用软件计算。

用于计算校验和的那一组通用函数放在体系结构相关文件*include/asm-xxx/checksum.h*里（例如，对i386平台而言就是*include/asm-i386/checksum.h*）。每个协议会使用正确的输入参数直接调用通用函数，或者定义包裹函数以调用通用函数。当先前已算过校验和的数据（例如IP报头）被修改时，校验和计算算法可以让协议去更新校验和，而非从头重新计算。

一个IP专用函数ip_fast_csum的原型（*checksum.h*里）如下所示。该函数的参数是IP报头的指针（iph）及其长度（ihl）。长度会因为IP选项而有所变化。返回值就是校验和。此函数会利用IP报头的长度始终是4字节的倍数这一点，让一些处理能流线化（streamline）。

```
static inline
unsigned short ip_fast_csum(unsigned char * iph, unsigned int ihl)
```

对要被传输的封包计算IP报头的校验和时，iphdr->check的值应该先变为零，因为校验和不应该反映出校验和本身。在此算法中，因为使用的是简单的求和法，零值字段就能有效排除在所得校验和结果之外。这就是为什么程序中的许多地方都可看见在调用ip_fast_csum之前该字段会先变为零的原因。

校验和算法有个有趣的特性，阅读封包转发和接收的源代码的人一开始可能会混淆。如果校验和是正确的，而进行转发或接收的节点对整个报头执行了此算法（原本的iphdr->check字段留着不动），结果会得到零。如果你查看函数ip_rcv，就会发现这就是输入封包以校验和做验证的方式。比起直觉式地把iphdr->check字段变零再重新计算而言，这种检查损毁的方式比较快。

以下是用于计算或更新IP校验和的主要函数：

ip_compute_csum
: 计算校验和的通用函数。其接收的输入参数就是一个任意大小的缓冲区。

ip_fast_csum
: 指定IP报头和长度后，计算并返回IP校验和。这个函数可用于验证输入封包以及计算外出封包的校验和。

  你可以把ip_fast_csum视为ip_compute_csum的变种，针对IP报头进行优化。

ip_send_check
: 计算外出封包的IP校验和。这是一个内含ip_fast_csum的包裹函数，事先会把iphdr->check变为零。

ip_decrease_ttl
: 当改变IP报头的一个字段时，对IP校验和做增值更新比从头计算要快一点。有了用于计算校验和的简单算法，才使此事成为可能。常见例子就是要转发的封包，因此其iphdr->ttl字段会被递减。ip_decrease_ttl会在ip_forward里被调用。

在前述*checksum.h*文件中还有好几个通用辅助函数，不过，几乎都是由L4协议所用。例如：

skb_checkum

这是由好几个包裹函数（包括前述几个函数）所使用的通用校验和计算函数，定义在*net/core/skbuff.c*中，几乎都是由L4协议在特定情况下使用。

csum_fold

把32位值的最高16位对折到最低16位，然后求出输出值的补码。此运算通常是校验和计算的最后阶段。

csum_partial[_xxx]

这个函数系列所计算的校验和会缺少csum_fold所做的最后对折步骤。L4协议可以调用csum_partial函数之一对L4数据做校验和计算，再启用一个诸如csum_tcpudp_magic的函数对伪报头(将在下一节说明)计算校验和，最后求出这两部分的校验和，再把结果合并。

csum_partial及其一些变种在多数架构上都是以汇编语言编写而成的。

csum_block_add

csum_block_sub

分别求出两个校验和的和及差。当一个数据区块的校验和是逐一增值计算时，第一个函数就有用处。当一块数据从校验和已算出的数据区块中删除时，就需要第二个函数。其他很多函数在其内部都使用这两个函数。

skb_checksum_help

此函数有两种不同行为，取决于其所得的参数是入口IP封包还是出口IP封包。

对入口封包而言，它会使其L4硬件校验和失效。

对出口封包而言，它会计算L4校验和。例如，当出口设备的硬件校验和计算能力无法使用时（参见第十一章的dev_queue_xmit），或者当L4硬件校验和已失效并因此必须重新计算之时，就会派上用场。例如，校验和可以因Netfilter的NAT运算而失效，或者当IPsec组集的转算协议调整了L4有效载荷（在原有IP报头和L4报头间安插其他报头）时失效。注意，如果一个设备可以在硬件中计算L4校验和，然后将之存储在L4报头中，则最后会修改到L3有效载荷，但是当L3有效载荷已被消化（digest）或者被IPsec套件加密时，就不可能修改了，因为这会使得数据失效。

csum_tcpudp_magic

对TCP和UDP伪报头计算校验和（参见图18-14所示）。

较新的NIC在硬件中提供了IP和L4校验和计算能力。虽然Linux利用了多数现代NIC的L4硬件校验和计算能力，但是没有利用IP硬件校验和计算能力，因为不值得多花这种复杂度（也就是说软件计算已经够快，因为IP报头的尺寸有限）。硬件校验和计算

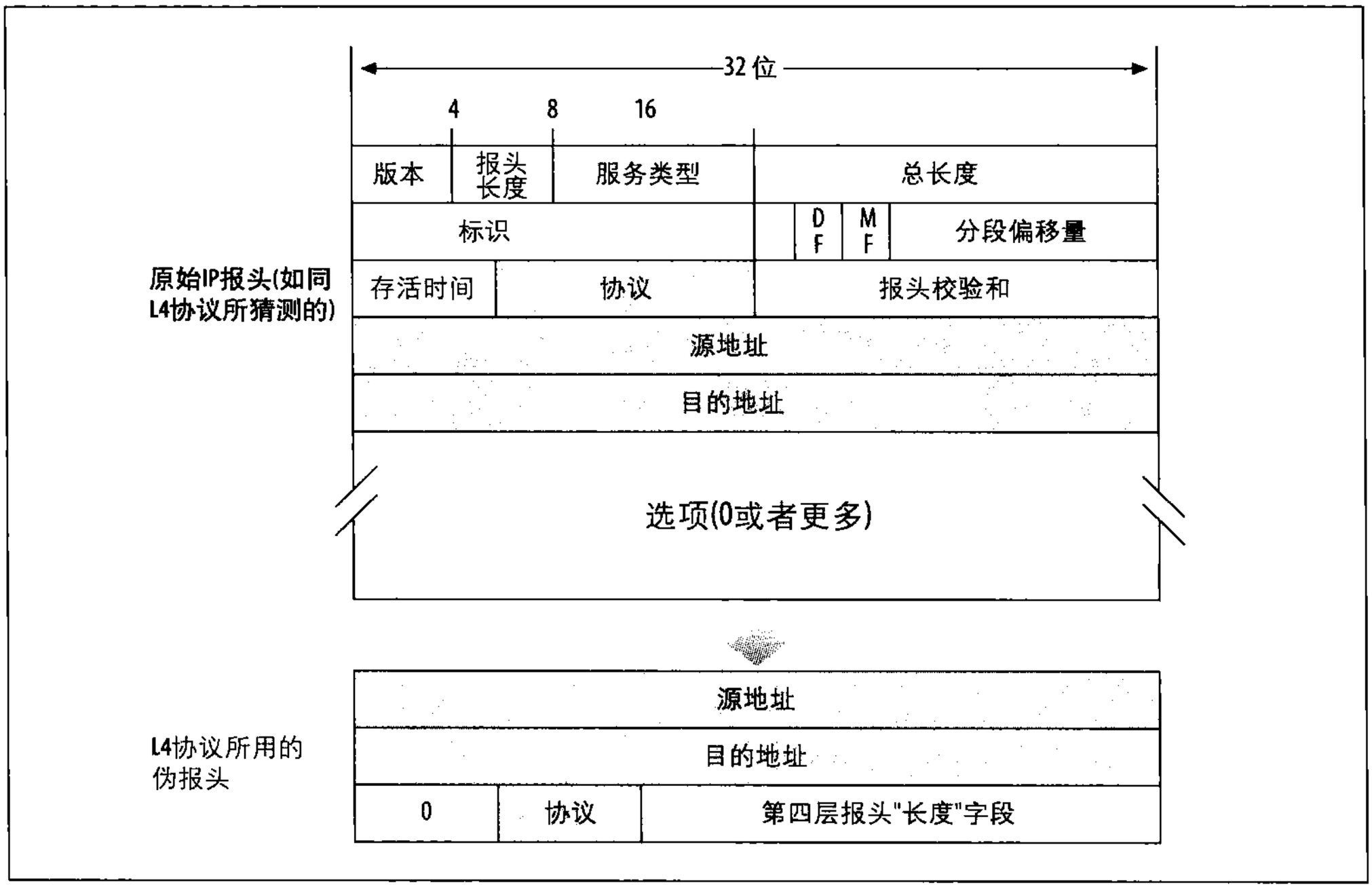


图 18-14：计算校验和时 TCP 和 UDP 所用的伪报头

只是降低 CPU 负载，因而得以让内核处理封包时能比较快；大多数现代 NIC 也都提供了在 L4（主要是 TCP）层降低 CPU 负载的能力卸载。硬件校验和计算会在第十九章简介。

对 L4 校验和所做的修改

TCP和UDP协议所计算的校验和会包括其报头、有效载荷以及所谓的伪报头。伪报头基本上就是一个区块，为了方便起见，其中的字段是从IP报头中取来（参见图18-14所示）。换而言之，IP 报头中出现的一些信息最后会整合至 L4 校验和内。注意，伪报头只是为了计算校验和而定义；伪报头并不存在于在网络中传送的封包内。

可惜，为了 NAT 或其他活动，IP 层偶尔必须修改一些 IP 报头字段，而这些字段又会被 TCP 和 UDP 用在其伪报头中。对 IP 层次所做的修改会使得 L4 校验和失效。如果保留校验和，IP 层的节点都不会检测到任何错误，因为它们只验证 IP 校验和。然而，目的地主机的 TCP 层会认为封包已损毁。因此，这种情况必须由内核处理。

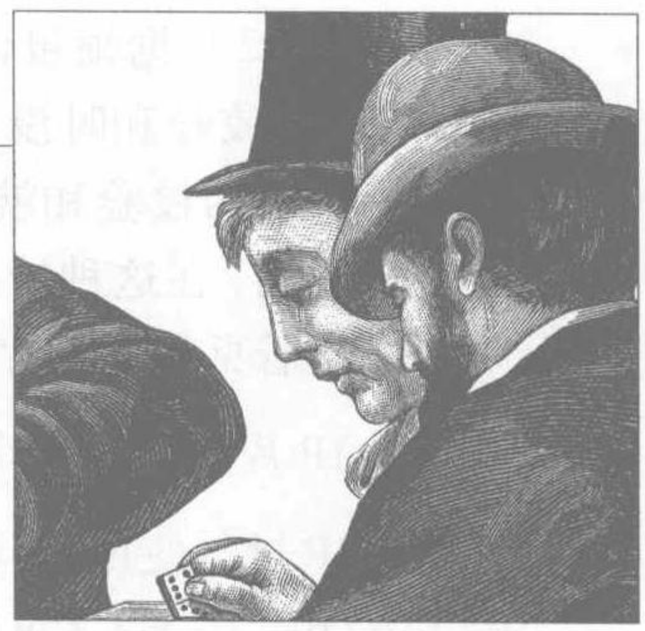
此外，有些固定情况下，已接收数据帧中的硬件计算的 L4 校验和会失效。以下是最常见的几种情况：

- 当输入之L2数据帧包含一些填充字节以满足最小帧尺寸的要求，但是NIC不够聪明，在计算校验和时没有把填充字节略掉。就此而言，硬件校验和和进行接收的L4分层所计算的校验和就会不吻合。第十九章“处理输入IP封包”一节将会知道，为了安全起见，在这种情况下ip_rcv函数总是会使得校验和失效。在本书第四部分中，也会看见桥接程序做类似事情。
- 当输入IP片段和前一个已接收片段重叠时。参见第二十二章。
- 当输入IP封包使用IPsec套件的协议之一时。就此而言，NIC再也无法计算正确的L4校验和，因为L4报头和有效载荷不是被压缩、消化就是加密。例如，参见*net/ipv4/esp4.c*中的esp_input。
- 因为IP层发生NAT或类似干扰，校验和必须重算。例如，参见*net/ipv4/netfilter/ip_nat_standalone.c*中的ip_nat_fn。

虽然名称令人困惑，但是字段skb->ip_summed必须处理L4校验和有关（更多细节参见第十九章）。当IP层知道有东西使得L4校验和失效时（例如伪报头中的一个字段受到修改），就会操作此字段的值。

我不会讨论本地产生的封包是怎么计算校验和的。但是，在第二十一章“把数据拷贝到片段：getfrag”一节，将会简单介绍当建立片段时校验和如何增值地计算出来。

第十九章



因特网协议第4版（IPv4）：Linux的原理和功能

前一章铺陈操作系统支持IP协议所需做的事情，而本章将介绍Linux支持IP的数据结构和基本活动，诸如入口IP封包如何传递至IP接收函数、校验和如何验证，以及IP选项如何处理。

主要IPv4数据结构

本节介绍IPv4协议所用的主要数据结构。可以参考第二十三章来了解其字段的详细说明。

我没有画出数据结构间的关系图，因为大多数数据结构都是独立的，不需要交叉参考。

`iphdr structure`

IP报头。其字段的意义已在第十八章“IP报头”一节谈过了。

`ip_options structure`

此结构（定义在*include/linux/ip.h*中）代表必须被传输或转发的封包的选项。那些选项都存储在此结构中，因为比起从IP报头相对应的部分读取而言，这样会比较容易。

`ipcm_cookie structure`

此结构结合了传输封包所需的各种信息。

`ipq structure`

IP封包的片段集。参见第二十二章“IP片段hash表的组织”一节。

inet_peer structure

内核会为最近连接过的每个远程主机都保留一个这一结构的实例。在第二十三章“长效IP端点信息”一节中就会知道这个结构如何使用。所有inet_peer结构的实例都会放在一棵AVL树中，而该树是一种对频繁的查询做了优化的结构。

ipstats_mib structure

SNMP（Simple Network Management Protocol，简单网络管理协议）采用一种，名为MIB（Management Information Base，管理信息库）的对象来收集系统的相关统计数据。名为ipstats_mib的数据结构会保存关于IP层的统计资料。第二十三章“IP统计资料”一节会详细说明此结构。

in device structure

in_device结构存储了一个网络设备所有与IPv4相关的配置内容，诸如用户以*ifconfig*或*ip*命令所做的变更。该结构通过net_device→ip_ptr链接到net_device结构，而且可以用in_dev_get和__in_dev_get取出。这两个函数间的差别是，第一个函数会负责所有必要的锁定，而第二个函数则假定调用者已把事情完成。

因为in_dev_get事情办成功后就会对in_dev结构增加引用计数值（当该设备的配置是支持IPv4时），而其调用者完成此结构相关事情时就应该用in_dev_put递减引用计数。

此结构的分配以及链接到设备是由inetdev_init来做（当设备上配置第一个IPv4地址时，此函数就会被调用）。

in_ifaddr structure

当在接口上配置一个IPv4地址时，内核会建立一个in_ifaddr结构，其中包含4字节长的地址以及其他几个字段。

ipv4_devconf structure

ipv4_devconf数据结构（其字段是通过*/proc in /proc/sys/net/ipv4/conf/*输出）用于调整网络设备的行为。每个设备都有一个实例，另外还有一个是存储默认值（ipv4_devconf_dflt）。其字段的意义会在第二十八章和三十六章讨论。

ipv4_config structure

尽管ipv4_devconf结构是用于存储每个设备的配置，ipv4_config存储主机的配置。

cork structure

cork结构是用于处理套接字CORK选项。第二十一章将会看到其字段如何用于在ip_append_data和ip_append_page连续启用时（处理数据分段）维护一些环境信息。

sk_buff 和 net_device 结构里与校验和相关的字段

第十八章“校验和”一节已看过用于计算IP和L4校验和的函数。本节我们要说明sk_buff结构的哪些字段是用于存储校验和相关信息、设备如何通知内核有关其硬件校验和计算能力，以及L4协议如何使用这类信息来决定是否替入口和出口封包计算校验和或者让网络适配卡（NIC）做这件事。

因为IP校验和始终是由内核在软件中计算和验证，接下来几个小节将重点关注L4校验工作的处理和相关议题。

net_device 结构

net_device->features字段会表明设备的能力。除了各种可以设定的标志外，有些标志可用于定义设备的硬件校验和计算能力。可能的功能列表列在*include/linux/netdevice.h*中，也就是在net_device本身定义之内。下面是用于控制校验和计算的一些标志：

NETIF_F_NO_CSUM

此设备很可靠，不需要使用任何L4校验和。例如，回绕设备就开启了此功能。

NETIF_F_IP_CSUM

此设备可以在硬件中计算L4校验和，但是只针对使用IPv4的TCP和UDP。

NETIF_F_HW_CSUM

此设备可以为任何协议在硬件中计算L4校验和。和NETIF_F_IP_CSUM相比，此功能较少见。

sk_buff 结构

取决于skb是指向已接收封包还是已传输出的封包而定，skb->csum和skb->ip_summed这两个字段的意义会不同。

当一个封包被接收时，skb->csum可能包含其L4校验和。名称很怪的skb->ip_summed字段则会记录L4校验和的状态。状态是由下列值指定（定义在*include/linux/skbuff.h*中）。下列定义代表着设备驱动程序要告诉L4层的事。一旦L4接收函数接收到一些缓冲区时，就可能改变skb->ip_summed的初始值。

CHECKSUM_NONE

csum中的校验和无效，究其原因可能有如下几种：

- 设备不提供硬件校验和计算。

- 设备计算了硬件校验和，并发现该数据帧已损坏。此时，设备驱动程序可以直接丢弃该数据帧。但是有些设备驱动程序宁愿把ip_summed设成CHECKSUM_NONE，然后让软件再次计算并验证校验和。这也实在可惜，因为做了接收该封包所有该做的事后，内核最后能做的就是重新检查那个校验和，然后丢弃该封包（参见*drivers/net/e1000/e1000_main.c*中的e1000_rx_checksum）。注意，如果输入数据帧要被转发，路由器不能因为L4校验和错误就将它丢弃（路由器不应该查看L4校验和）。这件事是由目的地主机来做。这是设备驱动程序不丢弃L4校验和失败的数据帧，而是让L4接收函数验证的另一个原因。
- 校验和必须重算并重新验证。参见第十八章“对L4校验和所做的修改”一节了解最常见的原因。

CHECKSUM_HW

NIC以L4报头和有效载荷计算了校验和，然后把校验和拷贝到skb→csum字段。软件（如L4接收函数）不仅必须把伪报头的校验和加至skb→csum，还必须验证最后所得的校验和。此标志也可视为下一个标志的特例。

CHECKSUM_UNNECESSARY

NIC已经计算并验证了L4报头以及伪报头的校验和（伪报头的校验和可以由设备驱动程序在软件中计算），所以，软件无需对L4校验与验证工作再做任何事。

例如，当错误的发生概率很低，不值得浪费时间和CPU运算力去计算及验证L4校验和时，也可设定CHECKSUM_UNNECESSARY。实例之一就是回环设备：因为通过此虚拟设备传出的封包绝不会离开本地主机，唯一可能的错误就是因为RAM出错或者操作系统中有bug。因此，此选项可用于这类特殊设备，但是接收端的标准行为是计算每个已收到的封包的校验和，然后丢弃已损坏的封包。

当封包传输后，csum代表指向缓冲区内的地点的指针，不再是校验和本身；而此地点也就是硬件适配卡要把它即将计算的校验和存放的地方。因此，在封包传输期间，只有当校验和是在硬件中计算时，才会用到此字段。这种L4和L2之间的互动（跳过L3）会引出其他一些必须解决的问题。诸如NAT这种功能会操作L4层所用的IP报头字段来计算伪报头的校验和，而此举会使该结构失效（参见第十八章“对L4校验和所做的修改”一节）。

如同接收的情况一样，ip_summed代表的是L4校验和的状态。此字段是由L4协议所用，以通知设备是否必须处理校验和计算工作。特别地，以下就是ip_summed在传输期间的意义：

CHECKSUM_NONE

协议已处理了校验和；设备不需做任何事情。当你转发一个出口数据帧时，L4校

验和已经准备好，因为已经由传送端主机计算好了；因此，没必要再计算它。参见第二十章的 ip_forward。当 ip_summed 被设成 CHECKSUM_NONE 时，csum 就没有意义。

CHECKSUM_HW

协议只把伪报头的校验和存储至其报头内；假定设备应该通过添加L4报头和有效载荷的校验和来以完成此工作。

在传输封包时，ip_summed 没有使用 CHECKSUM_UNNECESSARY（相当于 CHECKSUM_NONE）。

尽管当NIC开启时，功能标志 NETIF_F_XXX_CSUM 是由设备驱动程序初始化的，但是每个已接收或被传输出去的 sk_buff 缓冲区都必须设定 CHECKSUM_XXX 标志。在接收时，设备驱动程序会根据 NETIF_F_XXX_CSUM 设备能力对 ip_summed 做正确的初始化。

传输时，L3传输API会根据出口设备的校验和计算能力（可以从路由表推导出来：吻合目的地的路由表缓存项目会包括出口设备相关信息，因此可以取得其校验和计算能力；参见 ip_append_data 作为范例）对 ip_summed 进行初始化。

了解先前所述的 skb->csum 和 skb->ip_summed 字段以及 CHECKSUM_HW 标志的意义之后，就可以研究（例如）TCPv4 如何处理对入口节段的查验工作（在 *tcp_v4_checksum_init* 中），以及 TCPv4 如何处理出口节段的查验工作（在 *tcp_v4_send_check* 中）。

封包的一般性处理

本章接下来要谈的是处理入口IP封包时内核必须注意的一般性考虑，诸如校验和计算及选项。后续各章将详谈如何转发、传输以及分段 / 重组封包。

协议初始化

IPv4 协议是由 ip_init（定义在 *net/ipv4/ip_output.c* 中）初始化的。由于 IPv4 功能无法从内核中删除（也就是不能编译成模块），也就没有所谓的 ip_uninit 函数。

以下是由 ip_init 完成的主要任务：

- 用 dev_add_pack 函数为IP封包注册处理程序（参见第十三章）。此处理程序为函数，名为 ip_rcv。
- 初始化路由子系统，包括与协议无关的缓存（参见第三十二章）。

- 初始化用于管理IP端点的基础架构（参见第二十三章“长效IP端点信息”一节）。

开机期间，ip_init会由inet_init调用，而inet_init会处理所有与IPv4有关的子系统的初始化，包括L4协议在内。

和Netfilter互动

本书不谈Netfilter防火墙子系统，但是，我们现在可以谈一谈主要的工作原理，特别是和本书这一部分所讨论的IPv4实现的关系。

基本上，防火墙在网络堆栈程序中的某些地方都有钩子函数，当封包或内核吻合某些条件时，封包总是会通过那些钩子函数。在这些地方，防火墙允许网络管理员操作流量的内容或配置。内核中的这些位置（如第十八章的图18-1所示）包括：

- 封包接收
- 封包转发（路由决策前）
- 封包转发（路由决策后）
- 封包传输

区分路由前和路由后之所以有用的原因，到了本书第七部分就会清楚了。

就刚才所列的每种情况而言，负责其运算的函数会分成两部分，通常称为*do_something*和*do_something*_finish（少数情况下，名称会是*do_something*和*do_something2*）。*do_something*只包含一些健康检查，也许还有一些处理工作。做实事的程序位于*do_something*_finish或*do_something2*之内。*do_something*最后会调用Netfilter函数NF_HOOK，而且把调用来自于哪个点位传进去（例如，封包接收），然后，如果用户以*iptables*命令配置的过滤规则决定不把该封包丢弃或拒绝，则此函数就会执行。如果没有规则可用，或者只是指出“向前走”，则函数*do_something*_finish就会被执行。通用的调用形式如下所示：

```
NF_HOOK(PROTOCOL, HOOK_POSITION_IN_THE_STACK, SKB_BUFFER, IN_DEVICE, OUT_DEVICE,
do_something_finish)
```

NF_HOOK的输出值可以是下列值之一：

- 当*do_something*_finish稍后被执行时，就是其输出值。
- 如果因为某个过滤器的缘故使得SKB_BUFFER被丢弃，就返回-EPERM。
- 如果没有足够的内存执行过滤运算，就返回-ENOMEM。

本章中，我们不需要担心这些细节。我们将假设没有配置任何过滤器，因此，在*do_something*结尾处，对Netfilter函数调用时只会执行*do_something*_finish。在ip_rcv函数结束处将看到第一个示例。

与路由子系统的交互

IP层在好几个地方都必须和路由表交互，诸如接收和传输封包时。我会在本书第七部分说明路由子系统时谈论路由细节。现在，我只简单谈谈IP层用于查询路由表的三个函数：

ip_route_input

决定输入封包的命运。如第十八章图18-1中所示，封包可以被本地传递、转发或丢弃。

ip_route_output_flow

传输封包前使用。此函数会返回下个跳点网关以及要使用的出口设备。

dst_pmtu

给定一个路由表缓存项目，就可返回相关的PMTU（Path Maxirnum Transmission Unit，路径最大传输单元）。

ip_route_*xxx*函数（第三十三章和第三十五章会说明）会查询路由表，并根据一组字段做出其决策：

- 目的IP地址
- 源IP地址
- 服务类型（ToS）
- 进行接收的设备（接收时才有）
- 可进行传输的设备列表

影响这些函数返回结果的因素中较为复杂的两个，是路由策略的存在以及防火墙的存在。

这些函数会把路由表查询的结果存储在skb->dst中。此结构包含好几个字段，包括完成封包的接收或传输所需调用的输入和输出函数指针（参见第十八章图18-1中有关使用这两个函数指针的地点）。如果查询失败，ip_route_*xxx*函数会返回负值。

这两个函数也会使用一个缓存，让通往同一个地点的封包流能快一点。目的地IP地址是做决策最重要的准则，而且用作缓存的搜索键。但是，每个缓存项目也包括其他一些参数，可区分所用的路径是哪一条。例如，缓存记录每条路径的PMTU，这在第十八章“路径MTU发现”一节曾讨论过。

处理输入IP封包

第十三章指出，内核会在每层上都把流量传送至正确的协议，其做法就是调用该协议所注册的处理程序函数。在第十三章的“协议处理函数的注册”一节中，我们了解了IP协议如何通过内核注册其协议处理函数ip_rcv（定义在*net/ipv4/ip_input.c*中）。现在，我们可以开始分析内核网络协议栈内IP封包的路径（从ip_rcv函数着手）。

ip_rcv是“与Netfilter交互”一节所述两阶段处理的典型案例。其工作就是对封包做健康检查，然后调用Netfilter钩子。大多数处理将在ip_rcv_finish（由Netfilter钩子调用）里发生。

以下是ip_rcv的原型。第三个输入参数没用到。

```
int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt)
```

netif_receive_skb函数（第十章中有说明）会把指向L3协议的指针（skb→nh）设在L2报头尾端。因此，IP层函数可以安全地将它转换成iphdr结构。

在调用ip_rcv之前，sk_buff的多数字段就会在一系列事件（从NIC发出的中断通知到L3协议处理程序的调用）发生期间而设定（如前几章所说明的）。图19-1所示是ip_rcv启动时一些sk_buff字段的值。注意，此时，skb->data（通常是用于指向有效载荷）指向L3报头。

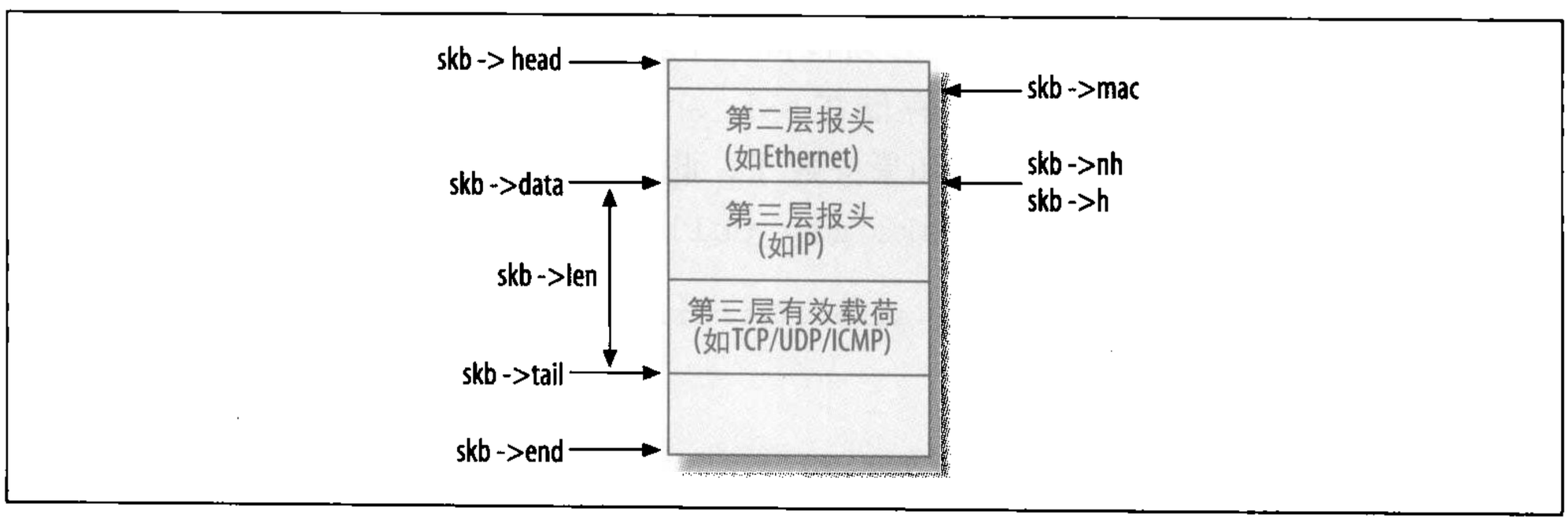


图19-1：ip_rcv开端处sk_buff数据结构的一部分

由第十章和第十三章已经知道，NIC设备驱动程序如何设定L3协议标识符skb->protocol以及封包类型skb->pkt_type。例如，Ethernet驱动程序通过eth_type_trans函数进行设定。

当数据帧的L2目的地址和接收接口的地址不同时，skb->pkt_type就会被设成PACKET_OTHERHOST。通常这些封包会被NIC本身丢弃掉。然而，如果该接口已进入混

杂模式，无论目的地L2地址为何，都会接收所有封包并将其传给较高层。内核会调用那些要求要存取所有封包的嗅探器，如第十章所述。但是ip_rcv和传给其他地址的封包无关，而只会简单地丢弃它们：

```
if (skb->pkt_type == PACKET_OTHERHOST)
    goto drop;
```

注意，接收用于别的L2地址的封包和接收应该路由到另一个系统的封包并不相同。对于后者而言，封包拥有该接口的L2地址，但是，L3层的地址和当前接收者的L3地址并不相同。路由器的配置就是为了接受这类封包，然后路由（本书第七部分会加以说明）。

skb_share_check会检查封包的引用计数是否大于1，大于1则意味着内核的其他部分拥有对该缓冲区的引用。如前几章所述，嗅探器和其他用户可能对封包感兴趣，所以每个封包都包含一个引用计数。netif_receive_skb函数（也就是调用ip_rcv的函数）会在调用协议处理程序前递增引用计数值。如果处理程序看见引用计数大于1，就会自己建立一份缓冲区副本，使其可以修改该封包。后续所有处理程序都将接收到原本无修改的缓冲区。如果需要一个副本，但内存分配失败，该封包就会被丢弃。

```
if ((skb = skb_share_check(skb, GFP_ATOMIC)) == NULL) {
        IP_INC_STATS_BH(IPSTATS_MIB_INDISCARDS);
        goto out;
}
```

pskb_may_pull的工作是确保skb→data所指区域包含的数据区块至少和IP报头一样大，因为每个IP封包（包括片段）必须包含一个完整的IP报头。如果条件吻合，就无事可做。否则，缺漏的部分就会从存储在skb_shinfo(skb)->frags[]里的数据片段（如果有的话）拷贝过来（注1）。如果失败了，此函数会以错误终止。如果成功了，此函数必须再次初始化iph，因为pskb_may_pull可以改变缓冲区结构。

```
if (!pskb_may_pull(skb, sizeof(struct iphdr)))
    goto inhdr_error;
iph = skb->nh.iph;
```

接着对IP报头做一些健康检查。基本IP报头的尺寸是20字节，因为存储在报头内的尺寸是以32位（4字节）的倍数表示，如果其值小于5，则表示有错误。if语句中的第二项检查就大有讲究。目前IP协议有两个版本：IPv4和IPv6。这个if语句确保该封包为IPv4封包。但是，由于这两个协议是由不同函数处理的，ip_rcv函数绝不会因IPv6而首先被调用。

```
if (iph->ihl < 5 || iph->version != 4)
    goto inhdr_error;
```

注1：　不要把数据片段和IP分段混在一起。参见第二章有关skb_info宏的用法。

现在，我们重复先前做过的相同检查，只不过这一次使用的是完整的IP报头尺寸（包括选项）。如果IP报头声明了iph->ihl的尺寸，则封包应该至少和iph->ihl一样长。这项检查会拖到现在才做，是因为此函数必须先确定基本报头（即不含选项的报头）没被截断，而且从中（本例中为ihl）读取东西前已通过基本健康检查。

```
if (!pskb_may_pull(skb, iph->ihl*4))
    goto inhdr_error;
iph = skb->nh.iph;
```

在这两个协议一致性检查已执行过后，此函数必须计算校验和，然后看看是否和报头中所携带的吻合。如果不吻合，该封包就会被丢弃。ip_fast_csum函数在第十八章“用于校验和计算的API”一节已介绍过。

```
if (ip_fast_csum((u8 *)iph, iph→ihl) != 0)
    goto inhdr_error;
```

校验和计算后，还有另外两个健康检查：

- 确保缓冲区（即已接收的封包）长度大于或等于IP报头中记录的长度。
- 确保封包的尺寸至少和IP报头的尺寸一样大。

```
{
_ _u32 len = ntohs(iph->tot_len);
if (skb->len < len || len < (iph->ihl<<2))
    goto inhdr_error;
```

这里我们必须说明为什么需要这两项检查。第一项是源自于L2协议（如Ethernet）会填补有效载荷（注2），所以，在IP有效载荷之后可能有多余的字节（例如，当数据帧的L2尺寸小于协议所要求的最小值时就会发生此事。Ethernet数据帧的最小数据帧长度为64字节）。这种情况下，封包看起来会比IP报头中记录的长度大一点。不同的尺寸和填充如图19-2所示。

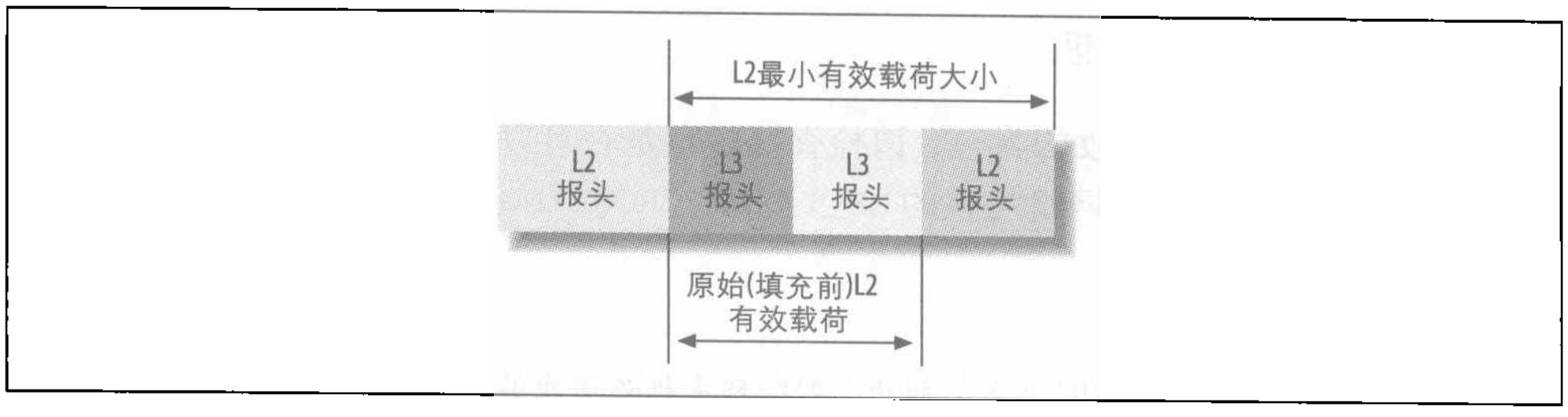


图19-2：L2填充以满足最小有效载荷尺寸

注2：从L2的角度看，有效载荷就是IP报头及其后的任何东西。

第二项检查源自于IP报头不能分段的事实，因此，每个IP片段必须至少包含一个IP报头（注3）。条件式中<<2的原因是报头的尺寸（iph->ihl）是以32位为单位计算的。只有很罕见的情况下，这项检查才会失败。也就是说，用损坏的封包计算而得的校验和碰巧产生与最初的封包相同的校验和（即查验时没有检查出错误）。

和一条路径相关的最小MTU事实上是68（来自于RFC 791文件）。由于IP报头最多可以有60字节长（20+40），而最小片段长度（最后一个除外）是8字节，因此，每个IP路由器都必须能够转发一个68字节的IP封包，而不再做进一步的分段。

如你所料，迄今为止我们所见以及稍后即将看见的所有健康检查，对系统的稳定性而言都非常重要。如果sk_buff结构碰巧没有正确初始化，或者如果IP报头本身已损坏，内核就会以错误方式处理封包，或者存取无效的内存位置，这将间接造成系统崩溃。

我们说过，L2协议可能会补满封包以达到特定最小长度。函数pskb_trim_rcsum会检查是否有此事发生，然后，如果真的有，就会用__pskb_trim把封包剪成正确尺寸，再让L4校验和失效，以免进行接收的NIC已计算出此值。__pskb_trim有点复杂，因为它也可能必须应付一些片段缓冲区（注4）。

当L4校验和由网卡在硬件中计算时，如果网卡不够聪明没把L2填充省去，就会把L2填充包括在内。因为此时无从得知是否为这种情况，为了安全起见，pskb_trim_rcsum只会使该校验和失效，并强迫L4协议重新计算。更多细节参见第十八章“校验和”一节。

```
        if (pskb_trim_rcsum(skb, len)) {
            IP_INC_STATS_BH(IPSTATS_MIB_INDISCARDS);
            goto drop;
        }
    }
```

最后，我们来到此函数的尾端。注意，到目前为止还没有做任何路由决策或选项处理；那是ip_rcv_finish的工作。如本章稍早所预期的，此函数最后会调用Netfilter子系统，而且大致上可以这么想：

“skb是从设备dev所接收的封包；请检查该封包是否可以继续其旅程，或者它是否需要改变。别忘了我们是从网络堆栈的NF_IP_PRE_ROUTING位置向你要求这件事，也就

注3： IP协议规范（RFC791文件）指出，因特网主机必须能够转发一个68字节的数据段，而无须进一步将其分段：换而言之，L2协议必须能够传输一个有效载荷至少68字节的数据帧。

注4： 参见第二十一章有关片段缓冲区的范例。

是说，封包已被接收但是还没做路由决策。如果你决定不丢弃该封包，就执行ip_rcv_finish”。

```
return NF_HOOK(PF_INET, NF_IP_PRE_ROUTING, skb, dev, NULL,
               ip_rcv_finish);
```

参见稍早“和Netfilter互动”一节所述的背景信息。

ip_rcv_finish函数

ip_rcv所做的仅是封包的基本健康检查。所以，当ip_rcv_finish被调用时，就会处理主要处理工作，例如：

- 决定封包是否必须本地传递或者转发。就第二种情况而言，则必须找到出口设备和下个跳点。
- 分析和处理一些IP选项。然而，并非所有IP选项都在此处理，这说明转发情况时就会知道。

以下是ip_rcv_finish函数的原型（和ip_rcv定义在相同的*net/ipv4/ip_input.c*文件中）。

```
static inline int ip_rcv_finish(struct sk_buff *skb)
```

skb->nh字段是在netif_receive_skb（在接收路径中很早就出现）里初始化的。当时，还不知道L3协议，所以会使用nh.raw做初始化。现在，此函数可以取得指向IP报头的指针了。

```
struct net_device *dev = skb->dev;
    struct iphdr *iph = skb->nh.iph;
```

skb→dst可能包含封包通往其目的地的路由信息。如果尚未得知该信息，此函数会询问路由子系统该把封包传送到哪儿，而且，如果路由子系统说目的地无法抵达，则该封包就会丢弃。参见第二十章“本地传递”一节以了解skb->dst不为NULL的范例。

```
    if (skb->dst == NULL) {
        if (ip_route_input(skb, iph->daddr, iph->saddr, iph->tos, dev))
            goto drop;
    }
```

接着，此函数会更新一些Traffic Control（QoS层）所用的统计数据。

```
#ifdef CONFIG_NET_CLS_ROUTE
    if (skb->dst->tclassid) {
        struct ip_rt_acct *st = ip_rt_acct + 256*smp_processor_id();
```

```
            u32 idx = skb->dst->tclassid;
            st[idx&0xFF].o_packets++;
            st[idx&0xFF].o_bytes+=skb->len;
            st[(idx>>16)&0xFF].i_packets++;
            st[(idx>>16)&0xFF].i_bytes+=skb->len;
        }
    #endif
```

当IP报头的长度大于20字节（注5）（5×32位），表示有一些选项要处理。此时，skb_cow——其名称来自于著名的“Copy on Write”（写入时才拷贝）——会被调用，如果缓冲区和别人共享，就会做出缓冲区的副本。对缓冲区具有排他拥有权是必要的，因为我们要处理那些选项，而且可能必须修改IP报头。

```
if (iph->ihl > 5) {
    struct ip_options *opt;
    if (skb_cow(skb, skb_headroom(skb))) {
                IP_INC_STATS_BH(IPSTATS_MIB_INDISCARDS);
        goto drop;
    }
    iph = skb->nh.iph;
```

ip_option_compile用于解读报头中所携带的IP选项。下一节会详细说明其实现内容。现在，我们感兴趣的是该函数的输出。第二章已知skb包含一个名为cb的字段，可以由任何管理sk_buff缓冲区的程序存放私有数据。就此而言，IP层可用此字段存储IP报头选项分析结果以及其他一些数据（例如分段相关信息）。此结果存储在一个struct inet_skb_parm类型的数据结构（定义在*include/net/ip.h*）中，而且可以由宏IPCB存取（参见第二十三章“ipq结构”一节）。

如果有任何错误的选项，封包就会被丢弃，而一条特殊的ICMP（Internet Comtrol Message Protocol,因特网控制消息协议）消息就会送回给传送者来告知所发生的问题。第二十五章就会知道，ICMP消息包含的信息是在报头中的哪里发现了错误，而这种信息有助于传送者了解发生了何事。

下一节就会知道，当ip_options_compile的第一个输入参数为NULL时，分析流程的输出结果就会存储在IPCB(skb)->opt，而这就说明了为什么已分析的选项都是以IPCB取出的。

```
if (ip_options_compile(NULL, skb))
        goto inhdr_error;
```

注意，ip_options_compile只会检查那些选项是否正确，然后将那些选项存储在skb

注5： 20字节是没有选项的IP报头的长度。

->cb所指的私有数据域字段的ip_option结构内。此函数不会处理任何选项。相反的，接下来的程序片段会处理部分工作。

在封包是由来源地路由的情况下，内核必须检查该设备的配置是否允许使用该选项（如果你对IP源路由不熟，可以参考“选项：严格和松散源路由”一节）。

我会在第二十三章“in_device结构”小节中简述in_device结构和相关的API。如果没有明确配置可以使用IP源路由，则该选项默认为允许使用。另一方面，如果该选项被关闭，则该封包就会被丢弃（但是不会产生ICMP消息）。NIPQUAD是简单宏（定义在*include/linux/kernel.h*），会把一个32位变量分割成4个8位的元素。

```
                if (opt->srr) {
                    struct in_device *in_dev = in_dev_get(dev);
                    if (in_dev) {
                        if (!IN_DEV_SOURCE_ROUTE(in_dev)) {
                            if (IN_DEV_LOG_MARTIANS(in_dev) && net_ratelimit())
                                printk(KERN_INFO "source route option %u.%u.%u.%u ->
%u.%u.%u.%u\n",
                                       NIPQUAD(iph->saddr), NIPQUAD(iph->
daddr));
                            in_dev_put(in_dev);
                            goto drop;
                        }
                        in_dev_put(in_dev);
                    }
                    if (ip_options_rcv_srr(skb))
                        goto drop;
                }
            }
```

当设备允许IP源路由时，程序就会调用ip_options_rcv_srr来设定skb->dst，然后决定要如何处理封包，也就是说，要决定使用哪个设备把该封包转发至来源地路由列表中的下一个跳点。正常来讲，所请求的下一个跳点指的就是另一台主机，因此，此函数就只需设定opt->srr_is_hit来指出所找到的地址。然而，ip_options_rcv_srr函数还得考虑“下个跳点”是本地主机的一个接口的可能性。如果发生这种事，此函数会把该IP地址写入IP报头的目的地IP地址，然后继续检查来源地路由列表中的下一个地址（如果有的话）；在程序中，这就称为“superfast loopback forward”（超快循环转发）。ip_options_rcv_srr会持续浏览IP报头来源地路由选项区块中的下一个跳点列表，直到其找到一个不是主机本地的IP地址。正常来讲，该列表中不会有一个以上的本地IP地址。然而，有一个以上也是合法的。如果有一个以上，从下个跳点再到下个跳点等于无运算，只是在ip_options_rcv_srr里多绕一圈循环而已。当ip_options_rcv_srr找到的最后一个下个跳点不是本地IP地址时，srr_is_hit标志就会设定，而这就表示该封包还没抵达其目的地而需要转发。

如果该封包要被转发（如第二十章“ip_forward_finish函数”一节所述），则srr_is_hit的初始化值会通知ip_forward_options把必要数据加进IP报头，以处理来源地路由选项。如果该封包正被传输（也就是如果它源自于此主机），则会改用opt->faddr，而不再使用opt->srr_is_hit标志。

上列程序中以术语MARTIANS决定一个参数值是否错误。此术语不是Linux开发人员耍嘴皮子的选择，而是来自于RFC文件本身。

ip_rcv_finish最后会调用dst_input，而dst_input实际上会调用存储于skb缓冲区的dst字段的函数。skb->dst的初始化不是在ip_rcv_finish的开端，就是在ip_options_rcv_srr（如果报头里存在着IP来源地路由选项时就会被调用）的尾端。skb->dst->input会设成ip_local_deliver或ip_forward，这取决于封包的目的地地址。因此，调用dst_input时，就可完成封包的处理（参见第十八章的图18-1所示以及稍早“和路由子系统互动”一节）。

此外，还可以参见第三十五章“源路由”一节，以了解在ip_rcv_finish中以及在ip_options_rcv_srr中调用ip_route_input时这两者间的关系。

IP选项

因为处理IP选项相当耗时，因此IP选项一直用得不多。在下面几节中，我们会逐一说明Linux内核所处理的IP选项以及这些IP选项是怎么处理的。

以下是涉及IP选项管理的主要API，而所有这些API都定义在*net/ipv4/ip_options.c*中。为了了解其中一些API，要记住，并非一个封包的所有IP选项都必须在其所有片段中重复。

ip_options_compile

分析IP报头中的一群选项，然后相应地对一个ip_options结构的实例做初始化。此结构稍后会用于处理那些选项；它还包括一些标志和指针，用于告知路由子系统中负责处理转发的部分，IP报头选项空间中该写入什么以及写入何处。“选项的分析”一节会详细探讨ip_options_compile。

ip_options_build

对IP报头中专属那些选项的部分做初始化（根据输入的ip_options结构）。传输本地产生的封包时，就会用到此函数。根据输入参数，该函数可以区分出那些片段并相应地进行处理：把那些不用拷贝到每个片段的选项从该片段的报头中删除（参见第十八章“IP选项”一节），然后改以空选项覆盖。此外，也会清掉ip_options

结构的标志（诸如opt->rr_needaddr），用于通知必须新增一个时间戳或一个地址至那些选项中。

ip_options_fragment

因为第一个片段是唯一继承原有封包的所有选项的片段，所以，其报头的尺寸应该大于或等于后续片段的尺寸。然而，Linux简化了此条规则。Linux让所有片段都保持相同的报头尺寸，让分段流程更为简单且有效。其做法是拷贝原有报头及其所有选项，然后，除了第一个片段以外，对其他所有片段都以空选项覆写那些不需要重复的选项（也就是IPOPT_COPY没设定的选项），然后清掉和其相关的ip_options结构的标志（如ts_needaddr）。

最后一项运算正是ip_options_fragment的目的。当我们在第二十二章讨论ip_fragment时，就会知道当第一个IP片段被传送出去后，内核会调用ip_options_fragment去修改IP报头，然后为后续所有片段重复利用新调整过后的报头。

ip_forward_options

转发一个封包时，有些选项可能必须被处理。ip_options_compile会解析一些选项，然后对用于存储解析结果的ip_options结构的一组标志做初始化。稍后，ip_forward就会处理那些选项。

ip_options_get

此函数会接收一群选项，用ip_options_compile解析，然后把结果存储在其分配的一个ip_options结构中。此函数也可从内核空间或用户空间那儿接收输入选项；有一个输入参数可用于指定来源。此函数的使用范例是通过ip_setsockopt函数（由诸如TCP和UDP之类的L4协议所使用）在指定的套接字上设定IP选项，参见系统调用setsockopt。ip_options_get会处理第十八章“End of Option List和No Operation选项”一节所述的填充工作。

ip_options_echo

指定入口IP封包及其IP选项后，此函数就可建立用于回复传送者的IP选项。例如，源路由选项必须在回复封包中反过来。参考RFC 1122（因特网主机的需求）第3.2.1.8节、4.1.3.2节以及4.2.3.8节还有RFC 1812（IPv4路由器的需求）。

此函数受到调用的一些地方包括：

- 回复入口ICMP请求的icmp_reply。
- 当入口IP封包吻合产生ICMP消息需求时的icmp_send。
- IP层所提供的通用函数ip_send_reply（以回复入口IP封包）。
- 存储入口SYN节段的选项的TCP。

现在，让我们看一看这些函数的实务应用。因为你还没看过第十八章图18-1中所有函数

的内部细节，所以，此时你可能无法理解所有事情。当你对其他函数都熟悉后，就可以回到本节的第二部分。

如你在第十八章图18-1所见，有不同路径可以传输封包，而且以略微不同的方式处理IP选项。我会谈及两种情况，而其他的情况请自行练习。

选项的处理

入口IP封包的选项首先会由ip_options_compile函数（下一节说明）解析。如前节所述，这些选项之后会由不同函数于不同时间进行处理（取决于封包是否要被转发、分段等）。图19-3总结了前节所介绍的重要函数（颜色较淡的部分）会于何处因入口封包和本地产生的封包而被调用。

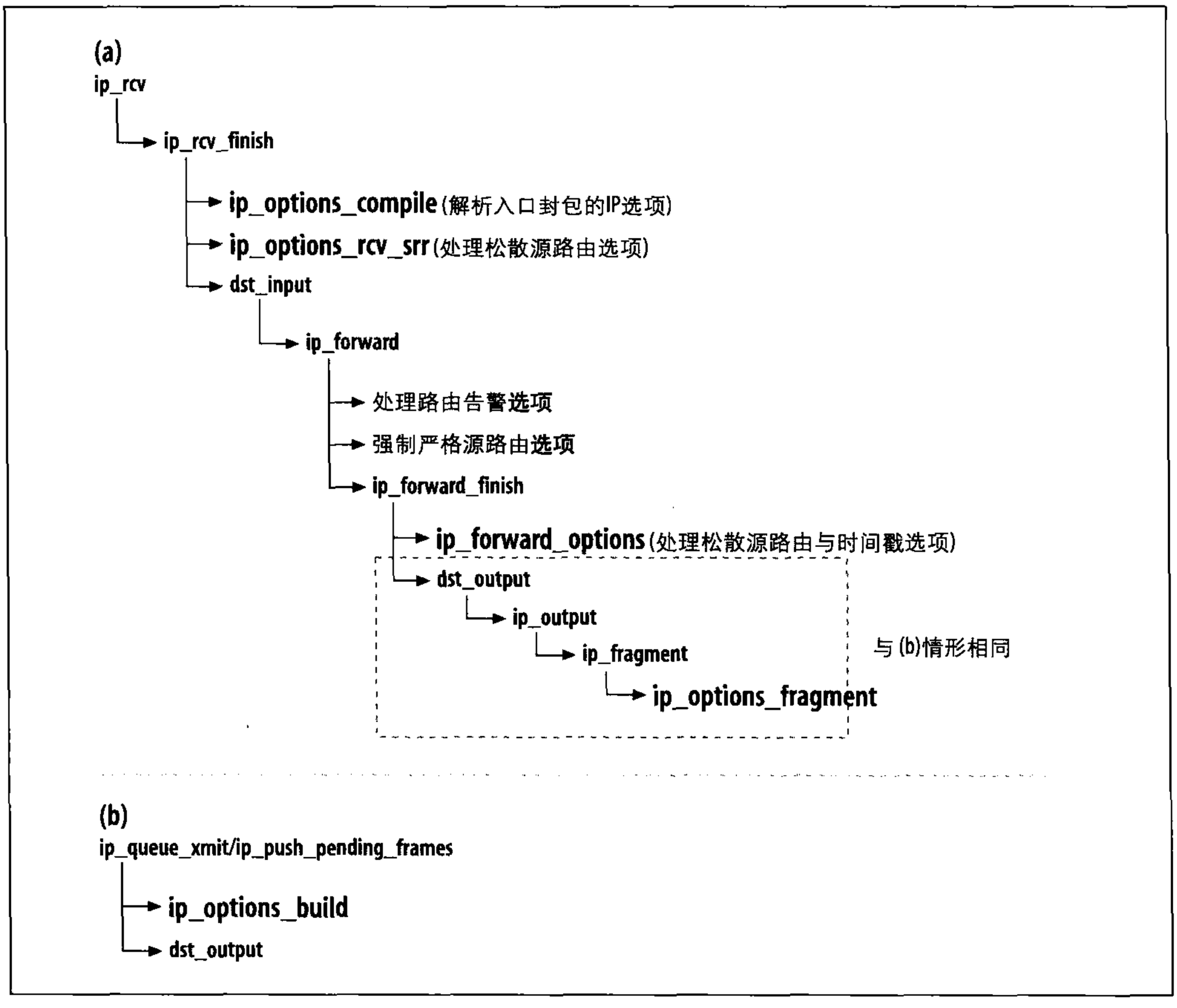


图19-3：（a）入口封包；（b）本地产生的封包

当入口封包要被转发时，ip_rcv_finish会调用ip_forward（通过dst_input）以处理转发流程。ip_forward会处理Router Alert选项（如果存在），而且会确保严格源路由选项没有问题。然后，ip_forward会请求ip_forward_finish以完成转发工作。ip_ forward_finish的行为会根据报头中是否包含选项而有所不同。

我们假设封包有选项。本例中，ip_forward_finish会调用ip_forward_options以处理这些选项（在转发封包时就应该被处理），然后调用dst_output以执行实际的传输。如第十八章图18-1所示，当入口IP封包必须被转发时，dst_output最后会调用ip_output。

此时，IP报头已就绪可用，因为所有选项都已被处理过。如果没有分段，选项的处理就算完成了。然而，如果封包必须分段，ip_output就必须确保只有第一个片段包含所有选项；而其他的则只有选项子集（根据第十八章表18-1所示）。本例中，ip_output会调用ip_fragment。一旦第一个片段做出来后，ip_fragment会使用ip_options_fragment去清掉后续片段不需要的选项。如此一来，ip_fragment就能一直从原有封包拷贝IP报头，并使得所有选项都是正确的。

在本地产生的封包里，选项是由ip_options_build处理的。第二十一章就会知道，ip_queue_xmit和ip_push_pending_frames如何使用这个函数。

选项的解析

在此，解析是指把IP选项从存储在IP封包报头里的格式中抽取出来，然后将其存储在一个名为ip_options的结构中，使得程序代码更方便处理。将那些选项存储在一个专用的数据结构中是有帮助的，因为不同选项是由IP程序中的不同部分处理的。ip_options_compile只解析那些选项，并不处理那些选项。前节已知选项会在何处处理。

ip_options_compile会在两种不同情况被调用：

- 由ip_rcv_finish调用以解析和验证输入封包的IP选项。如第十八章图18-1所示，ip_rcv_finish会因所有入口封包而被调用，无论封包是本地传递或者被转发。当我在此节提及入口封包时，包括必须要被转发的入口封包，因为这种封包不是要传给本地系统的。
- 由ip_options_get调用，例如以替AF_INET套接字解析要传给setsockopt系统调用的输入数据。

现在，让我们分析ip_options_compile如何解析IP封包报头的选项。以下是此函数的原型：

```
int ip_options_compile(struct ip_options * opt, struct sk_buff * skb)
```

两个输入参数的值会让函数知道自身是在什么情境下被调用的：

- 入口封包：skb不为NULL（本例中，opt为NULL）。
- 封包正被传输：skb等于NULL（本例中，opt不为NULL）。

这意味着取决于函数的上下文而定，IP报头会存储在不同的地方。在传输一个本地产生的封包时，opt不为NULL，而且opt->data包含一个指向IP报头（先前由调用者产生一部分）的指针。反过来，如果此函数正在处理一个入口封包，则报头包含在skb输入缓冲区中，而且opt为NULL。就第二种情况而言，ip_options结构是存储在skb->cb中的。ip_options_compile会根据IP报头位在何处（也就是skb->nh或opt->__data）而对一些本地变量做初始化，诸如optptr。ip_options_compile也时常使用skb的值来区分前两种情况。

在这两种情况下（传输和转发），都必须填写opt。唯一要做的抉择就是到哪里去取得做解析的输入IP报头，以及把结果存储于何处。

```
if (!opt) {
    opt = &(IPCB(skb)→opt);
    memset(opt, 0, sizeof(struct ip_options));
    iph = skb->nh.raw;
    opt->optlen = ((struct iphdr *)iph)->ihl*4 - sizeof(struct iphdr);
    optptr = iph + sizeof(struct iphdr);
    opt->is_data = 0;
} else {
    optptr = opt->is_data ? opt->__data : (unsigned char*)&(skb->nh.iph[1]);
    iph = optptr - sizeof(struct iphdr);
}
```

如果解析失败，ip_options_compile会立刻返回。调用者会以下列方式之一处理该事件，这取决于那些选项是否由已接收或已传输的封包使用：

已接收封包中有损坏的选项

ICMP消息会传回给来源地。

已传输封包中有损坏的选项

应用程序会接到通知（也就是由用于传输封包的函数所返回的错误值）。

以下是一些可能导致解析失败的原因：

- 单一选项不能出现在报头中一次以上。唯一的例外是空选项IPOPT_NOOP。空选项可出现任意次数，而且通常是用于施加某种对齐，不是对个别选项，就是对选项之后的有效载荷（空选项不需要做处理）。

- 一个报头字段的值为无效值，或者其值为当前用户不准使用的值。这种情况适用于本地产生的流量。只有超级用户可以产生带有内核不懂的选项或子选项代码的IP封包。对超级用户特权的检查是由capable函数来进行的。

  最初的IP RFC提到，当接收到一个不理解的选项时，路由器应该将其忽略。Linux只有对本地产生的封包有不同的行为（参见刚才提到的capable）。

目前，只有两个单字节选项：

- 选项结尾（IPOPT_END）
- 空选项（IPOPT_NOOP）

主要的for循环就是绕行一个一个的选项，然后把解析结果存储在输出的ip_options结构opt中。循环内的代码看起来可能很复杂，但实际上很容易阅读，只要你考虑下列几点：

- 代表尚未解析的选项区块的大小（注6）。
- optptr指向正被分析的选项区块的当前位置。optptr[1]是选项的长度，而optptr[2]是选项指针（也就是选项起始之处）。图19-4所示是此数组元素所指之处。处理每个选项的代码总是根据这些参数而先从两项健康检查做起。
- optlen的初值会设定当前选项的长度。不要把optlen和opt->optlen搞混了。注意，当opt不为NULL时，optlen就不会再设初值，因为这早已在ip_options_get里做了。
- 标志is_changed用于记录报头何时已修改（这需要重算校验和）。

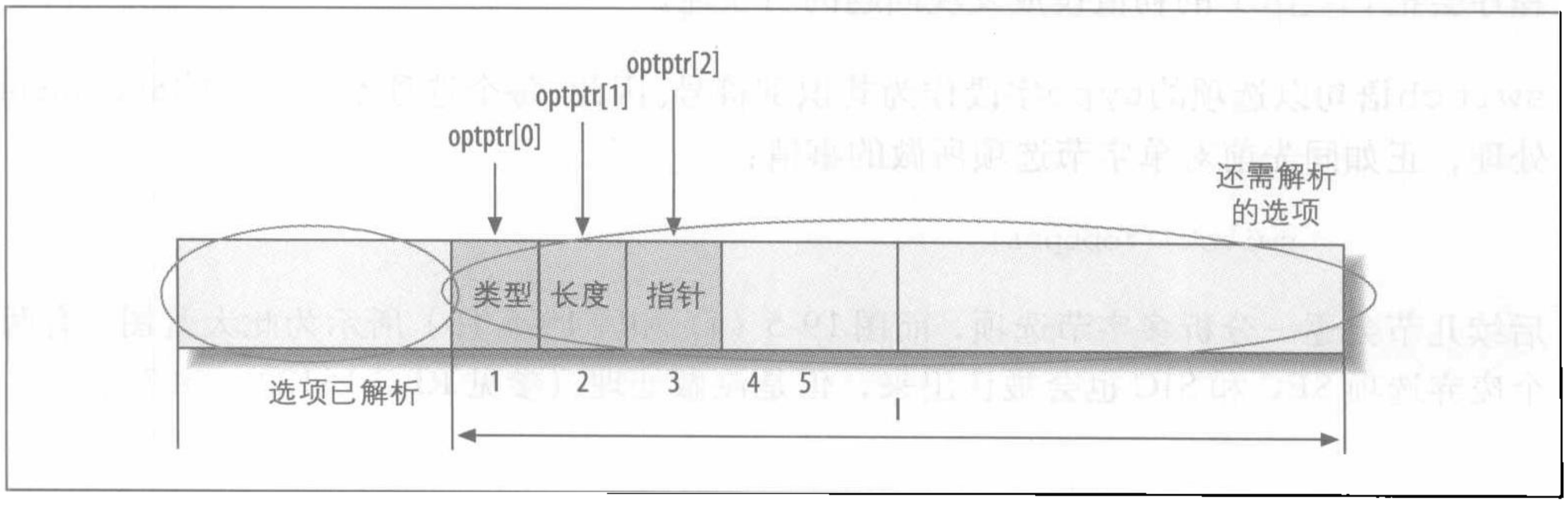


图19-4：执行期间ip_options_compile的本地变量的值

注6：　阅读代码时，要确定你没被变量1（用作for循环的索引）和整数1搞混。它们看起来很像，如果你搞混了，可能会花上一个小时都还没看懂代码。

IPOPT_END选项之后不能有其他选项。因此，一旦找到一个，无论后面是什么，都会以更多的IPOPT_END选项覆盖。

多字节选项的基本健康检查包括：

- 该选项必须至少4个字节长。因为选项的报头是3个字节长，字段pointer就不能小于4。例如，时间戳选项至少需要5个八位组长度，而4个八字节仅是由报头所用（参见第十八章图18-8）。
- 在报头中保留空间的那些选项（因为假定它们应该由后面几个跳点或者目的地主机填写）必须尊重该选项所需的尺寸。例如，假定时间戳选项应该保留的空间是4字节（一个IPv4地址的尺寸）的倍数。

由于每个选项的长度包括前两个字节（type和length），而且由于是从1起算（不是0），如果length小于2或者大于还没解析的选项区块，就表示有错误：

```
if (optlen<2 || optlen>l) {
    pp_ptr = optptr;
    goto error;
}
```

注意，有些选项（诸如TIMESTAMP）的最小长度大于2，因此，刚才所示的一般性检查虽然必要，但不见得永远充分。每选项处理程序中还有更为明确的检查。当在选项中发现错误时，特殊的ICMP消息必须传回给传送者。此ICMP封包中包括原有的IP报头、8字节的IP有效载荷，以及一个指向错误于何处发现的偏移量。8字节的IP有效载荷中包括L4报头的开头部分，而且通常会包括L4端口号；这样可以让ICMP错误消息的接收者发现和出差错的IP封包相关的套接字（细节详见第二十五章）。在传回错误消息前，程序会把pp_ptr的初值设成发现问题的所在地。

switch语句以选项的type字段作为其识别符号。因此，每个选项都由不同的Statement处理，正如同先前对单字节选项所做的事情：

```
switch (*optptr)
```

后续几节会逐一分析多字节选项，而图19-5（a）和图19-5（b）所示为此大蓝图。有两个废弃选项SEC和SIC也会被认出来，但是没做处理（参见RFC 1812）（注7）。

注7：还有共他IP选项，诸如IP MTU Discovery Option(RFC1063)，虽然这个选项有定义，但几乎没有真正用到过，或者是在过去几年中觉得有用而已，因此也已经废弃了。特别地是，IP MTU Discovery已被路由MTU发现功能（在RFC1191中，第十八章“路径MTU发现”一节曾讨论）替代掉了。

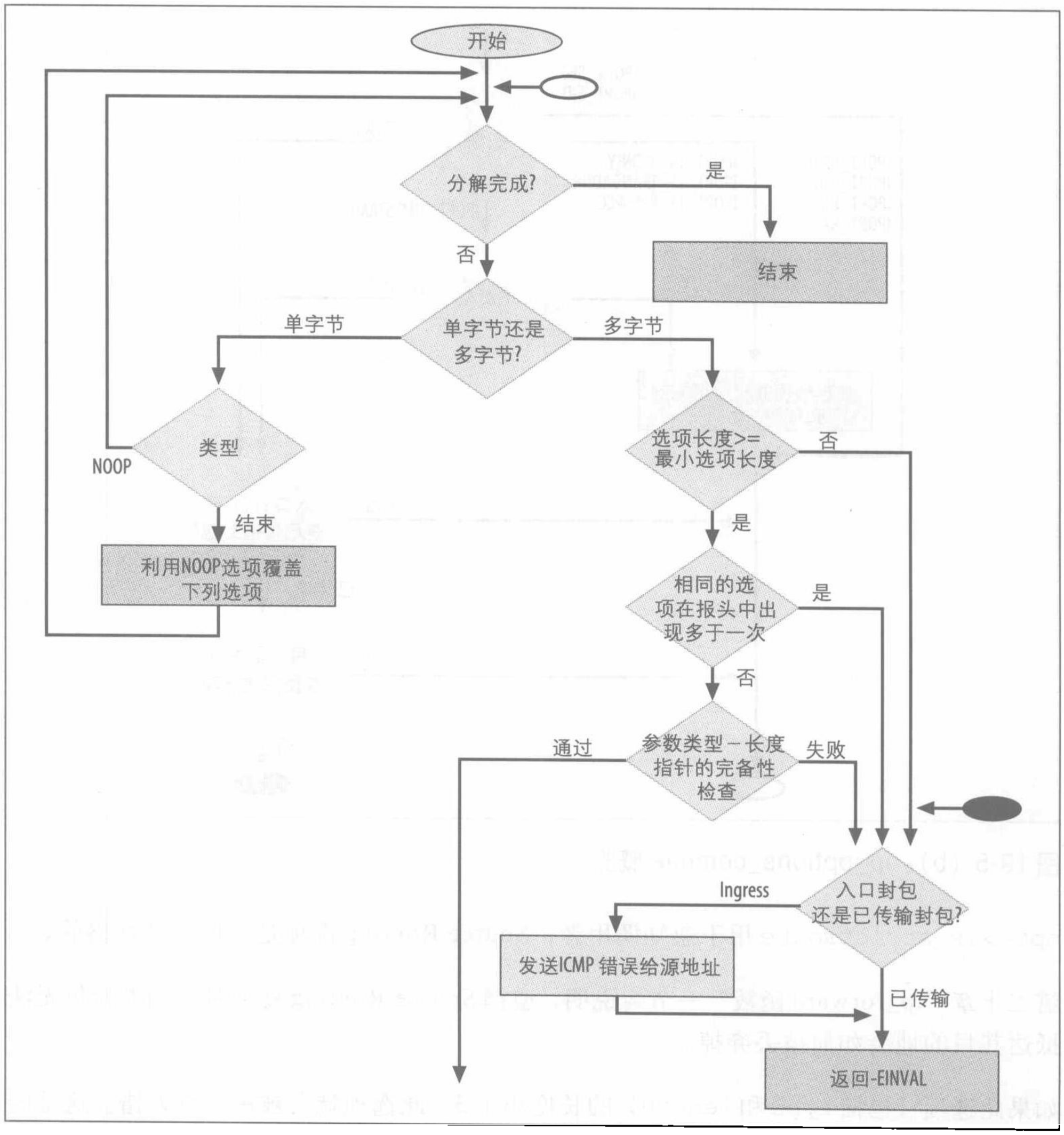


图19-5（a）：ip_options_compile概览

选项：严格和松散源路由

一个报头中只能出现一个Source Routing选项。标志opt->srr用于检测该条件：如果后续代码没发现此选项中有任何错误，就设定该标志。如果同一类型的另一个选项出现在此报头中的后面，则错误就会被检测到。

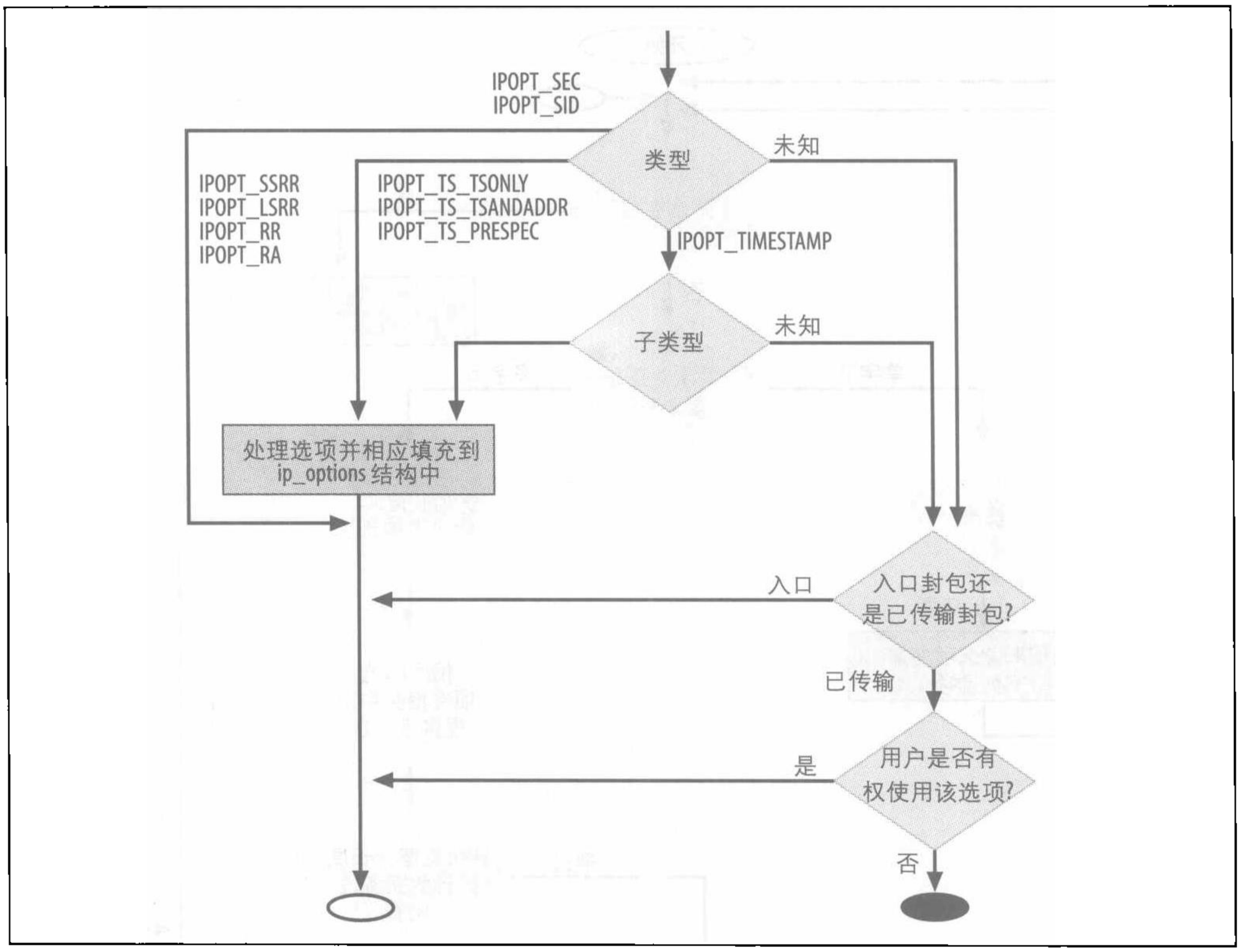


图 19-5（b）：ip_options_compile 概览

opt->is_strictroute 用于通知调用者，Source Routing 选项是松散还是严格的。

第二十章“ip_forward 函数”一节会说明，遵循 Source Routing 规则时，如果封包无法抵达其目的地会如何被丢弃掉。

如果此选项（包括 type 和 length）的长度小于 3，此选项就会被视为有差错。这是因为其值必须包含 type、length 以及 pointer 字段。同时，pointer 的值不能小于 4，因为此选项的前三个字节已用于 type、length 以及 pointer 字段。

当输入的 skb 参数为 NULL 时，表示 ip_options_compile 已被调用，以解析一个外出封包（由本地产生，非转发）的选项。就此而言，用户空间所提供的地址数组中的第一个 IP 地址会存储在 opt->faddr 中，然后以 memmove 运算把数组的其他元素往回移一个位置，将该地址从数组中删除。此地址稍后会由第二十一章所述函数（ip_queue_xmit 和 ip_append_data 的用户）取出，如此，那些函数才会知道目的地 IP 地址为何。opt->faddr 的用法的简单范例可在函数 udp_sendmsg 中找到。

```
if (!skb) {
    if (optptr[2] != 4 || optlen < 7 || ((optlen-3) & 3)) {
        pp_ptr = optptr + 1;
        goto error;
    }
    memcpy(&opt->faddr, &optptr[3], 4);
    if (optlen > 7)
        memmove(&optptr[3], &optptr[7], optlen-7);
}
opt->is_strictroute = (optptr[0] == IPOPT_SSRR);
opt->srr = optptr - iph;
break;
```

选项：Record Route

就Record Route选项（以及Timestamp选项）而言，传送者要事先在报头中保留所需部分。因此，处理此选项时，只有当还有空间时，新元素才能加进报头中。如果有空间，ip_options_compile函数会设定rr_needaddr标志，以通知路由子系统，当做出路由决策后（注8），把外出接口的IP地址写入IP报头。注意，如果那些选项属于本地产生的封包，则IP地址列表中会包含进行传输的接口的地址。

只有当正在处理入口封包的选项时skb才不是NULL，所以这段程序就是把首选源IP地址拷贝到记录在报头中的地址列表，然后更新标志is_changed来强迫更新IP校验和。参见第三十五章“首选源地址选择”一节，以了解为何使用rt_spec_dst IP地址的原因。

该地址如果不是在此处所示的程序区段内被写入（因为封包正被转发），就是稍后在标志rr_needaddr设定时才会被写入，而该选项的pointer字段都会往前移4个字节（IP地址的尺寸）。这就说明了为什么ip_forward_options要往回走4个字节，以把IP写入正确位置。

```
if (optptr[2] <= optlen) {
    if (optptr[2]+3 > optlen) {
        pp_ptr = optptr + 2;
        goto error;
    }
    if (skb) {
        memcpy(&optptr[optptr[2]-1], &rt->rt_spec_dst, 4);
        opt->is_changed = 1;
    }
    optptr[2] += 4;
    opt->rr_needaddr = 1;
}
opt->rr = optptr - iph;
break;
```

注8：　其做法是调用ip_options_build。参见第二十一章。

选项：Timestamp

因为optlen代表正在解析的选项的长度，所以，if语句只是检查是否有任何空间可容纳新信息。本例中，此选项的长度代表的就是传输者所保留的空间（而不是迄今为止所用掉的空间）。

```
if (optptr[2] <= optlen) {
        _ _u32 * timeptr = NULL;
```

此选项的处理取决于第十八章图18-8中subtype字段所指定的子选项，但是这三个子选项的处理方式大致相同。无论子类型为何，要处理该选项者都需要两项信息（将被存储在ip_option结构中）：

- 是否必须记录一个地址、一个时间戳，或者两者都记录。
- 该项信息应该写入IP报头中的何处（偏移量）。

如果必须记录时间戳（对TS_ONLY和TS_TSANDADDR的情况而言为真），timeptr的初值就会设成应该写入到IP报头中的正确地点。此外还要注意，仅当skb不为NULL时（当该选项属于入口封包时，和本地产生的封包相反），timeptr才会初始化。

由“选项的分析”一节已知道，在处理本地产生的封包时也可以调用ip_options_compile。就此而言，skb为NULL，所以timeptr不会被初始化（也就是依然为NULL），因此，报头中不会记录任何时间戳。这里并没有任何事情出错，因为时间戳会由ip_options_build放在那儿。该函数会存储时间戳，因为opt->ts_needtime等于1。

处理要转发的入口封包以及本地产生的封包的唯一差别就是，在转发的情况下，时间戳会加进IP报头中，而校验和必须重算（同时opt->is_changed也必须设定）。

当子代码为IPOPT_TS_PRESPEC时，只有当下一个要比对的IP地址是系统本地地址时，时间戳才会加进去。用于做此项检查的函数是inet_addr_type，而其主要返回值如下：

RTN_LOCAL

该IP地址属于一个本地接口。

RTN_UNICAST

根据路由表，该IP地址可以抵达，而且是单播地址。

RTN_MULTICAST

该地址是多播地址。

RTN_BROADCAST

该地址是广播地址。

因为本地广播地址以及已注册多播地址可视为本地地址（也就是系统监听的地址），下列检查RTN_UNICAST的程序所做的事正是我们想要做的事——判断该地址是否为本地地址：

```
{
    u32 addr;
    memcpy(&addr, &optptr[optptr[2]-1], 4);
    if (inet_addr_type(addr) == RTN_UNICAST)
        break;
    if (skb)
        timeptr = (__u32*)&optptr[optptr[2]+3];
}
opt->ts_needtime = 1;
```

取决于正在处理的子选项而定，时间戳必须写入IP报头中的不同偏移量之处。第一部分是根据情况对timeptr初始化，而第二部分是把时间戳拷贝到正确位置。根据子选项而定，ts_needtime和tr_needaddr也会被初始化。

```
if (timeptr) {
    struct timeval tv;
    _ _u32  midtime;
    do_gettimeofday(&tv);
    midtime = htonl((tv.tv_sec % 86400) * 1000 + tv.tv_usec / 1000);
    memcpy(timeptr, &midtime, sizeof(_ _u32));
    opt->is_changed = 1;
}
```

最后一部分是处理第十八章“Timestamp选项”一节所讨论的计数器溢出。

```
unsigned overflow = optptr[3]>>4;
if (overflow == 15) {
    pp_ptr = optptr + 3;
    goto error;
}
opt->ts = optptr - iph;
if (skb) {
    optptr[3] = (optptr[3]&0xF)|((overflow+1)<<4);
    opt->is_changed = 1;
}
```

选项：Router Alert

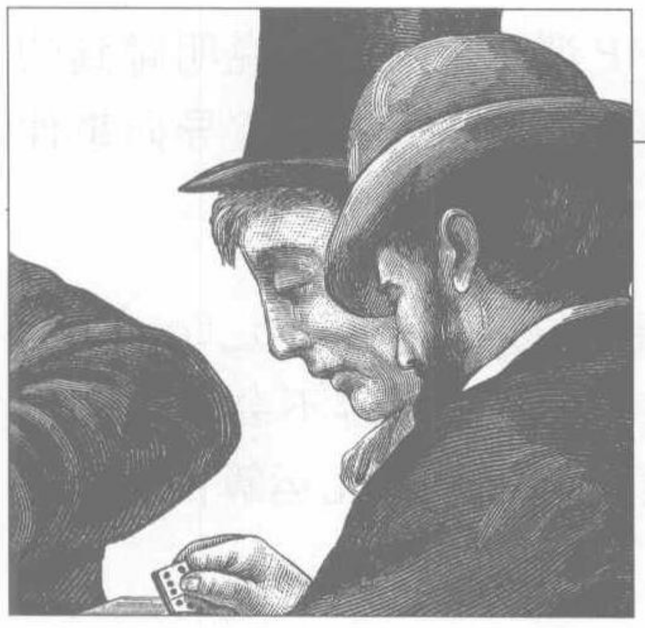
如第十八章“Router Alert选项”一节所解释的，此选项的最后两个字节必须为零。如果此选项通过了健康检查，ip_options_compile会对router_alert标志做初始化，使得稍后ip_forward可据此处理（opt->router_alert会被视为布尔值、零或非零）。

```
if (optptr[2] == 0 && optptr[3] == 0)
    opt->router_alert = optptr - iph;
```

处理解析错误

如果在本地产生的封包（skb==NULL）中发现了错误，此函数只会返回错误（必须由调用者处理）。但是，如果错误是在已接收的IP封包中发现的，就得把ICMP错误消息传回给来源地：

```
error:
    if (skb) {
        icmp_send(skb, ICMP_PARAMETERPROB, 0, htonl((pp_ptr-iph)<<24));
    }
    return -EINVAL;
}
```



第二十章

因特网协议第4版（IPv4）：转发和本地传递

在ip_rcv_finish函数的尾端，如果目的地址和本地接口不同，内核就必须把封包转发至适当的主机。另一方面，如果目的地址为本地地址，内核就必须把封包准备好，以便较高层使用。如第十九章“ip_rcv_finish 函数”一节所述，正确的选择会通过调用dst_input而从skb缓冲区得知。现在，我们来看这两项任务（转发和本地传递）是怎么完成的。

转发

如同前章所述的众多网络活动一样，转发也是分割成两个函数：ip_forward和ip_forward_finish。第二个函数会在第一个函数尾端调用（只要Netfilter允许）。这两个函数都定义在 *net/ipv4/ip_forward.c* 中。

此时，ip_rcv_finish里调用了ip_route_input（第十九章所述），sk_buff缓冲区内包含了转发封包所需的所有信息。转发由下列步骤组成：

1. 处理IP选项。这可能会涉及记录本地IP地址以及时间戳（如果IP报头中的选项需要这两项信息）。
2. 确定封包可以被转发（根据IP报头字段）。
3. 递减IP报头的TTL字段，然后，如果TTL字段变为零，就丢弃该封包。
4. 根据路径相关MTU，必要时处理分段工作。
5. 把封包传送至外出设备。

如果封包因为某种原因而无法转发，源主机就必须收到ICMP消息通知，来说明碰到的问题。即使封包会被转发，但是封包绕送时使用的是次佳路径，因而触发了重导向事件时，ICMP消息也会送出，来作为通知。

和IPsec交互是转发工作的主要部分，而且是由xfrm4_xxx函数实现的（在ip_forward里），而这些函数都是进入IPsec基础架构的钩子。因篇幅有限，这些内容不会在本书中讨论。这些提及的行为是当IPsec没配置时（此时，那些函数调用就只是无运算而已）转发如何运作。

ICMP重导向

当主机系统（通常是路由器）被要求做某件另一台路由器比较适合去做的事情时，主机系统就会送出一条ICMP重导向消息（更多细节参见第二十五章和第三十一章）。

当封包为源路由时，路由器会假设传送者请求该路径有很好的理由，因此不会去质疑。路由器会尊重所请求的路径，不会传送ICMP重导向消息。这种特殊情况会在“ip_forward函数”一节涉及。

ip_forward函数

我们已经知道，ip_forward是由ip_rcv_finish调用（参见第十八章图18-1）来处理所有不是传给本地系统的输入封包。此函数接收的一个输入参数是和该封包相关的skb缓冲区；所有必需信息都在该结构内。skb->dst（路由信息）是由调用ip_route_input而初始化的（在ip_rcv_finish里的程序路径的开端）；请参见第三十三章了解更多细节。

图20-1总结了此函数的内部逻辑：

```
int ip_forward(struct sk_buff *skb)
```

此函数的中心是skb和本地变量iph（代表封包的IP报头，而其初值设定会重复地来自于skb的iph字段；此字段必须重新设定初值，因为报头可能因为ip_forward里的一些函数调用而被修改）的操作。

如果IP报头中发现了Router Alert选项，则现在就会被处理（注1）。此选项的函数处理程序为ip_call_ra_chain，而此处理程序依赖一个全局列表（ip_ra_chain），该列表中包含了本地套接字列表（其中的套接字都设定了IP_ROUTER_ALERT选项，

注1：参见第十八章“Router Alert”选项一节。

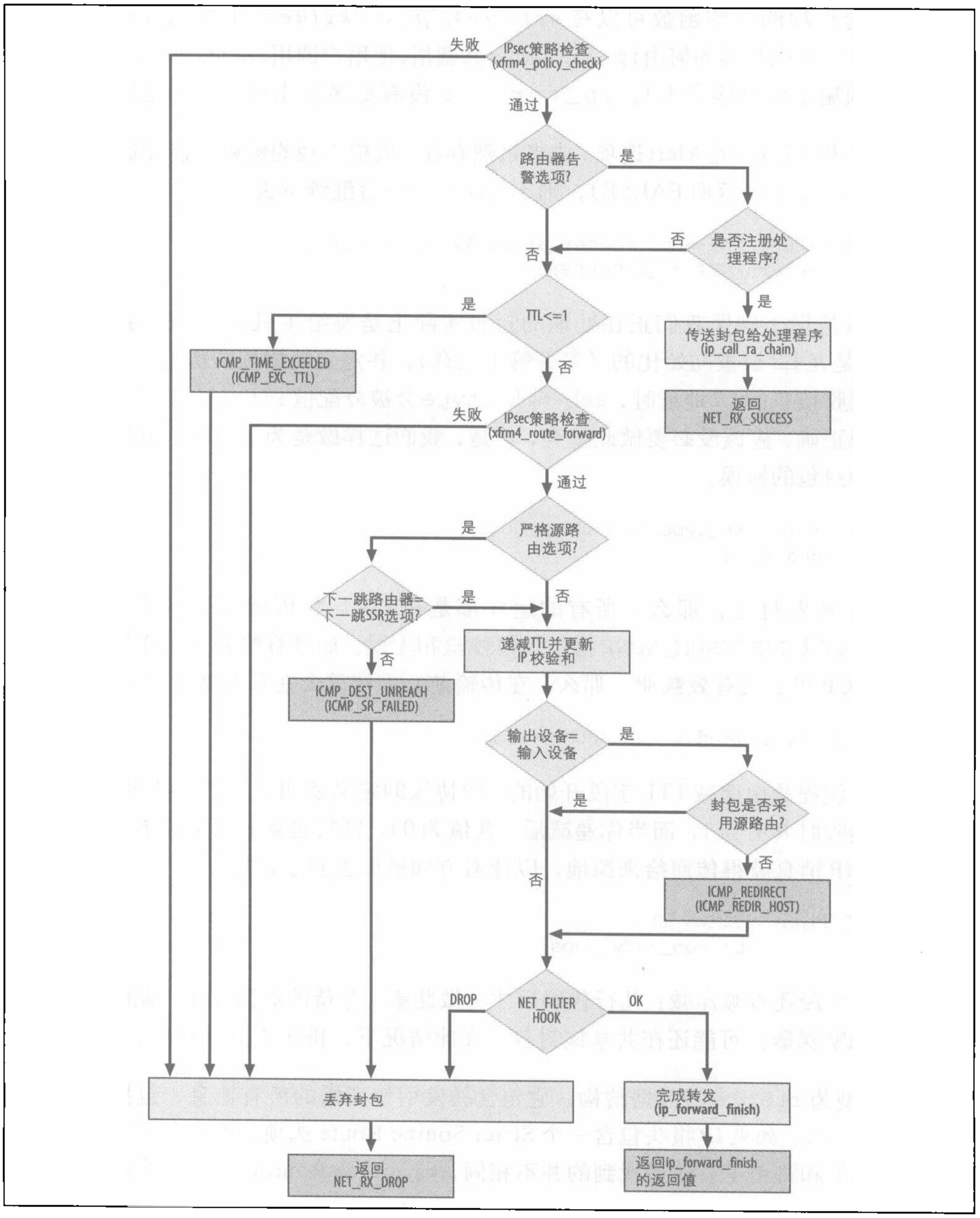


图20-1：ip_forward函数

因为这些套接字对携带Router AlertIP选项的封包感兴趣）。当一个入口IP封包被分段时，ip_call_ra_chain会先重组整个IP封包，然后才会将该封包传给ip_ra_chain的那些Raw套接字，如第十八章图18-1所示。

管理该告警选项的一些函数可以在*net/ipv4/ip_sockglue.c*里找到（例如，参见ip_ra_control以及其如何由ip_setsockopt调用，把用户调用setsockopt系统调用所请求的选项施加至套接字上）。ip_forward没有更多工作可做，于是返回成功。

如果报头中没有Router_Alert选项，或者虽然存在，但感兴趣的流程都没在运作（此时，ip_call_ra_chain返回FALSE），则ip_forward会继续下去：

```
if (IPCB(skb)->opt.router_alert && ip_call_ra_chain(skb))
    return NET_RX_SUCCESS;
```

下列检查只是用于确保我们正在处理的封包实际上是要给主机的L2层的。skb->pkt_type是在L2层被初始化的（参见第十三章），并定义数据帧的类型。当数据帧的目的地是接收接口的L2地址时，skb->pkt_type会被分配值PACKET_HOST。如果较低层函数处理正确，应该没必要做此检查，但是，我们这样做是为了避免一开始就接收到不该接收的封包的错误。

```
if (skb->pkt_type != PACKET_HOST)
   goto drop;
```

既然我们在转发封包，那么，所有的运作都是在L3层，因此我们不用担心L4查验工作；我们以CHECKSUM_NONE指出当前校验和无误。如果有些处理工作稍后修改了IP报头、TCP报头或有效载荷，那么，在传输前，内核就会在那儿重算校验和。

```
skb->ip_summed = CHECKSUM_NONE;
```

真实的转发流程是由递减TTL字段开始的。IP协议的定义表明，当TTL抵达0值时（这意味着你接收时其值为1，而当你递减后，其值为0），则封包就必须被丢弃，然后一条特殊的ICMP消息就得传回给来源地，以让对方知道你丢弃了封包。

```
if (iph->ttl <= 1)
    goto too_many_hops;
```

注意，TTL字段还没被递减；几行代码后才会做此事。等待的原因在于，此时，其他子系统（例如嗅探器）可能还在共享该封包；在此情况下，报头不能被修改。

rt指向类型为rtable的数据结构，它包含转发引擎所需的所有信息，包括下个跳点（rt_gateway）。如果IP报头包含一个Strict Source Route选项，而且下个跳点（由该选项中取出）和路由子系统所找到的并不相同，则Source Routing选项就失败了，同时该封包会被丢弃。

```
rt = (struct rtable*)skb->dst;
if (opt->is_strictroute && rt->rt_dst != rt->rt_gateway)
        goto sr_failed;
```

这种情况下，另一条ICMP消息会传输给传送者。

多数健康检查都做完后，此函数会略微更新封包报头，然后把封包交给ip_forward_finish。因为我们即将修改缓冲区内容，所以，必须替我们自己做个本地拷贝。只有当该封包为共享时（如果封包没被共享，就可以修改，安全无虞），或者封包报头的可用空间不足以存储L2报头时，这个拷贝才会由skb_cow生成。

```
if (skb_cow(skb, LL_RESERVED_SPACE(rt->u.dst.dev)+rt->u.dst.header_len))
    goto drop;
```

现在，TTL会由ip_decrease_ttl递减，而且ip_decrease_ttl也会更新IP校验和。

```
ip_decrease_ttl(iph);
```

如果有比请求的还要更好的下一个跳点存在，现在，原来的主机就会接获一条ICMP REDIRECT消息（但是，只有在原来的主机没有请求源路由之时）。opt->srr字段指出源路由已被请求，因此，原来的主机不在乎是否可能找到更佳路径。第三十五章就会知道，何时会对缓存的路径设定RTCF_DOREDIRECT标志，以指出封包的来源应该收到一条ICMP REDIRECT消息。

```
if (rt->rt_flags&RTCF_DOREDIRECT && !opt->srr)
 ip_rt_send_redirect(skb);
```

这里，priority字段是使用IP报头的TOS字段设定的。此优先权稍后会由Traffic Control（QoS层）使用。

```
skb->priority = rt_tos2priority(iph->tos);
```

此函数终止时会要求Netfilter执行ip_forward_finish（如果没有禁止转发的过滤规则）。

```
return NF_HOOK(PF_INET, NF_IP_FORWARD, skb, skb->dev, rt->u.dst.dev,
               ip_forward_finish);
```

ip_forward_finish函数

如果看到了这个函数，就表示封包已通过所有会使其停止的检查，而且真的已经就绪，可以传出到另一个系统了。

到目前为止，IP报头中有两个可能的选项已经处理过了，也就是我们在“ip_forward函数”一节所见的：Router Alert和Strict Source Routing。现在，我们把该封包传给函数ip_forward_options，以处理那些选项所需的最后工作。只要检查稍早由ip_options_compile（由ip_rcv_finish启用）做初始化设定的标志（例如opt->

rr_needaddr）和偏移量（例如opt->rr），就可以找出该做些什么。ip_forward_options也会重算IP校验和（之前可能必须更新所有IP报头字段）。参见第十九章“选项的处理”一节。

最后，封包是由dst_output（下一节会谈及）传输出去：

```
static inline int ip_forward_finish(struct sk_buff *skb)
{
    struct ip_options * opt = &(IPCB(skb)->opt);

    IP_INC_STATS_BH(IPSTATS_MIB_OUTFORWDDATAGRAMS);

    if (unlikely(opt->optlen) {
            ip_forward_options(skb);
    return dst_output(skb);
}
```

我们似乎已经很接近线路了，但是，在让设备驱动程序去做传输前，还有些任务要做。

dst_output函数

所有传输（无论是本地产生或是从其他主机转发而来）都会通过dst_output上路，以抵达目的地主机，如第十八章图18-1所示。此时，IP报头已完成：内含传输所需信息以及本地系统要负责添加的其他任何信息。

```
static inline int dst_output(struct sk_buff *skb)
{
        int err;

        for (;;) {
                err = skb->dst->output(&skb);
                if (likely(err = 0))
                        return err;
                if (unlikely(err != NET_XMIT_BYPASS))
                        return err;
        }
}
```

dst_output会调用虚拟函数output(如果目的地址是单播的，会初始化为ip_output；如果是多播的，则会初始化为ip_mc_output)。分段也是在该函数内处理的。最后，会调用ip_finish_output来衔接邻居子系统（参见第十八章图18-1所示)。只有当Netfilter放行时，ip_finish_output才会被调用(参见第二十一章“衔接邻居子系统”一节)，否则，封包会被丢弃。

注意，如果output函数返回NET_XMIT_BYPASS值，就可能会被调用数次。这可谓是一种连续调用output函数的简单机制。IPsec协议套件藉此实施转换，然后才做真实的传输。

本地传递

第三十五章会说明转发（路由）引擎如何得知本地主机是封包的目的地。由第十九章“ip_rcv_finish函数”一节的结尾已知，当封包已抵达其目的地主机时，在ip_rcv_finish顶端调用ip_route_input，就会把skb->dst->input初始化为ip_local_deliver（和ip_forward相反，此时封包必须被转发）。此外，Netfilter有最后的权力，可决定通用do_something函数（诸如ip_local_deliver）是否可以调用相应的do_something_finish函数（此例中为ip_local_deliver_finish）来完成工作。

```
intip_local_deliver(structsk_buff*skb)
{
    if (skb->nh.iph->frag_off & htons(IP_MF|IP_OFFSET)) {
        skb = ip_defrag(skb, IP_DEFRAG_LOCAL_DELIVER);
        if (!skb)
            return 0;
    }
    return NF_HOOK(PF_INET, NF_IP_LOCAL_IN, skb, skb->dev, NULL,
                   ip_local_deliver_finish);
}
```

和转发（重组几乎可以忽略）相反的是，本地传递必须做很多工作来处理重组工作。除了特殊情况外（例如当Netfilter必须重组封包来查看其内容），每个片段都可做转发，而无须试着重新合并它们。相反的，对本地传递而言，原有IP封包一定要始终重组以整体传送才行，因为较高的L4层应该幸福到对IP层的分段需求毫无所知。

重组工作是在ip_defrag函数内进行的；当ip_defrag完成重组工作时，会返回一个指向原有封包的指针，但是，如果封包还不完整，就返回NULL。ip_local_deliver里的程序调用ip_defrag，检查本地变量skb里的返回值，然后如果封包不完整就返回。ip_defrag的第二个输入参数会在第二十三章“ipq结构”一节中介绍。

只有当封包重组后，此函数才能传递。Netfilter必须去查询其配置，然后，如果该封包可接受，就执行ip_local_deliver_finish。我们会在第二十四章详细探讨ip_local_deliver_finish的细节。第十八章“封包的分段/重组”一节曾介绍过重组，而第二十二章会再详谈。

第二十一章

因特网协议第4版（IPv4）：传输

本章我们讨论L3层的封包传输，也就是第十八章图18-1的左上角所示。传输（transmission）指的是封包离开本地主机而去往另一台主机；传输可以由L4层进行，也可以作为转发的最后阶段而被调用。如第十八章图18-1所示，中央传递封包的函数是dst_output；本章所探讨的函数都是在此函数之前执行的，也就是替该函数把封包准备好。在此阶段，内核的任务包括：

查询下一个跳点

IP层必须知道外出设备以及用作下一个跳点的下一个路由器。路径是通过函数ip_route_output_flow发现的（在L3或L4层调用）。本章不讨论路由，因为该项主题够大，应独立讨论，因此会在本书第七部分谈及。

初始化IP报头

几个字段会在此阶段填入（诸如封包ID）。如果封包是要转发的那个，稍早之前就已对报头做了一些工作（诸如更新TTL、校验和以及选项字段）。但是，此时若要开启传输，还有更多事情必须完成。

处理选项

软件必须尊重需要把一个地址或时间戳加进报头里的那些选项。

分段

如果IP封包太大，无法在外出设备上传输，就必须分段（除非分段被显式地禁止）。

校验和

所有对报头的其他工作都做完后，还必须计算IP校验和。我们会看到，IP层除了处理L3校验和外，也可能处理L4校验和。就这两种情况而言，校验和可以单次就算出来，也可以逐一增值计算出来。虽然需要校验和，但L3层不见得总是必须去计算，因为有些设备的硬件会干这件事（由CHECKSUM_HW标志所指示）。

由Netfilter检查

如第十八章图18-1所示，Linux防火墙系统有机会在处理的各个阶段（包括传输）把每个封包丢弃或调整。

更新统计数据

取决于传输结果（成功或失败）以及一些诸如分段的行动而定，相关的SNMP定时器可能必须被更新。

到目前为止，选项处理和分段是最麻烦的任务；分段会在第二十二章说明，而选项已在第十九章谈过了。过去，有两种不同函数用于传输：一种是给可以快速传输的封包使用，因为封包不需要分段或者处理IP选项；而另一种则提供的所有的服务。现在，内核不再刻意区分这两种情况了。

进行传输的重要函数

第十八章图18-1左上角所列的两个函数出现在图21-1中，而调用这两个函数的L4协议则为其做了分类。分成两组函数的原因在于右边的L4协议——TCP和SCTP（Stream Control Transmission Protocol，流控制传输协议）——做了很多工作来准备分段，而这使得IP层的工作变少了。相反的，列在左边的Raw IP和其他协议则把所有分段工作都留给了IP层。

图21-1显示出L4的传输和L3的最后步骤间的主要函数，而L3会调用第二十七章讨论的邻居系统。在此图顶端，最常见的L4协议都列出来了。UDP和ICMP会调用一组L3函数来进行传输，而TCP和SCTP则会调用另一组。当本章所述的L3函数完成其工作时，就会把封包传给`dst_output`。至于Raw IP，当其使用`IP_HDRINCL`选项时，就得完全负责把IP报头准备好，所以会跳过本章所述函数，而直接调用`dst_output`。更多细节参见“Raw套接字”一节所述。IGMP（Internet Group Management Protocol，因特网群组管理协议）也会直接调用`dst_output`（对其IP报头初始化后）。

于是，分段是由两组函数处理的，如下所示：

`ip_queue_xmit`

L4协议把第十八章讨论的PMTU列入考虑后，已经把数据切成一些大小适合分段（需要的话）的块了。IP层的工作就是把IP报头加入已经创建好的那些数据片段中。

`ip_push_pending_frames`及相关函数

调用此函数的那些L4协议不会考虑分段或协助分段。此外，出于提高大效率的原因，那些L4协议把封包中的数据直接往下传给IP层，但引入了处理的复杂性。取

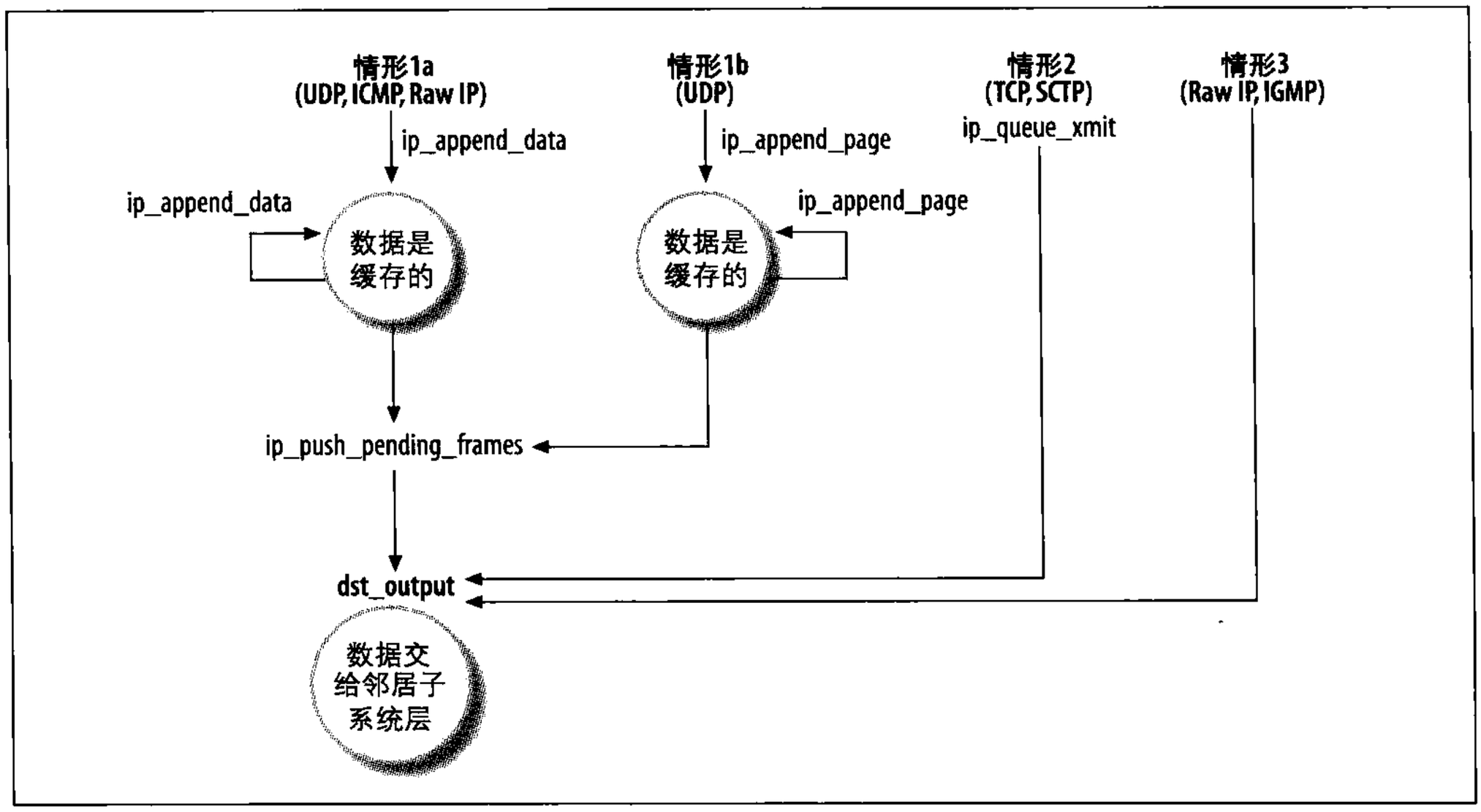


图 21-1：不同协议以不同方式调用 dst_output

决于本章稍后所述的几个因素，L4 协议可以多次调用 ip_append_data 来存储好几个传输请求，而不实际传输任何东西。

ip_append_data不仅是将传输请求暂存于缓冲区而已，同时也以透明方式产生一些最佳大小的数据片段，使得 IP 层稍后更易于处理分段。这样可以让 IP 层不用在做分段时还得把数据从一个缓冲区拷贝到另一个缓冲区，因而获得很大的性能改善。

当 L4 协议必须刷新 ip_append_data 所创建的输出队列时，该协议会调用 ip_push_ pending_frames，通过此函数做所有必要的分段，然后把所得的封包往下推给 dst_output。

ip_append_data的变种名为 ip_append_page，目前由 UDP 使用。我们稍后会简述此函数。

传输期间，在特定情境下也会用到其他函数：

ip_build_and_send_pkt

由 TCP 使用，来传输 SYN ACK 消息。

ip_send_reply

由 TCP 使用，来传输 ACK 和 Reset 消息。图 21-1 中的分类只包含最常见情况：因为ip_send_reply会使用 ip_append_data和ip_push_pending_frame，因此，TCP 不会只用 ip_queue_xmit。

当你了解图21-1中的函数如何运作之后，这些就很容易了解了。L4协议也有可能直接调用dst_output；IGMP和RawIP是完成此事的两个协议（参见“Raw套接字”一节）。

本章我只会简述ip_queue_xmit，而把比较多的时间花在ip_append_data/ip_push_pending_frames上头，因为这两个函数是分段这个复杂任务的关键部分。

多播流量

如第十八章图18-1所示，已传输的多播和单播流量的出口路径是很相似的，比入口路径更为相似。我不打算在本书中详谈多播，但是，本章我要指出传输期间单播和多播间的一些差异。例如，在“建置IP报头”一节就会看到，对于多播流量，TTL的初始化有所不同。转发封包时也是如此。

本地流量相关的套接字数据结构

Linux中用socket结构的实例表示BSD套接字。此结构包含一个指向sock数据结构的指针，而网络层信息都存储在sock结构内。sock数据结构相当庞大，但是，在*include/net/sock.h*里有相当完善的说明资料。实际上，sock数据结构是被分配成一个更大结构的一部分，而这个更大结构也就是协议簇专用的结构；就PF_INET套接字而言，此结构是inet_sock（定义在*include/linux/ip.h*中）。inet_sock的第一个字段是一个sock实例，而剩余的则存储PF_INET私有信息，诸如来源和目的IP地址、IP选项、封包ID、TTL以及cork（马上就会讨论）。

```
struct inet_sock {
    struct sock sk;
    ...
    struct {
        ...
    } cork;
}
```

给定一个指向sock数据结构的指针后，IP层使用inet_sk宏把该指针强制转换成指向外层inet_sock数据结构的指针。换而言之，inet_sock和sock结构的基地址是一样的，这是C程序中常用的一个功能，用于处理复杂而嵌套的结构。

inet_sock的cork字段在ip_append_data和ip_append_page中扮演着重要角色：存储由这两个函数正确做数据分段所必需的背景信息。在各种信息中，包含了IP报头里的选项（如果有的话）以及片段长度。

只要传输是由本地产生（只有少数例外），每个sk_buff缓冲区都会和其sock实例相关联，而且以skb->sk链接起来。

有各种不同函数可用于设定和读取sock和inet_sock结构的字段值。而其中有些函数是由图21-1中的函数所调用的。本章中，我们只需了解其中几个函数的意义：

sk_dst_set *和* _ _sk_dst_set

一旦连接套接字之后，这些函数可以把用于抵达目的地的路径存储在sock结构中。sk_dst_set是内含_ _sk_dst_set的简单包裹函数，用来处理上锁。如果不需要上锁（因为先前已经处理了），就可以直接调用_ _sk_dst_set。

sk_dst_check *和* _ _sk_dst_check

顾名思义，路径的有效性可以由这两个API测试。然而，如果路径有效，这些函数就会将其作为返回值而返回。也就是说，这些函数可用于取出路径，而不仅是用于测试有效性（无效路径会使其传回NULL）。这两个函数很相像；略微的差别在于，如果它们发现路径不再有效时，清理暂存于缓存内的路径的方式不同。

skb_set_owner_w

把一个sk_buff缓冲区指派给给定的sock结构。这有助于查核。

sock_alloc_send_skb *和* sock_wmalloc

这些函数会分配sk_buff缓冲区。sock_alloc_send_skb可用于分配单一缓冲区或者一系列片段的第一个片段（参见ip_append_data的讨论）；sock_wmalloc会处理后续的片段。这两个函数最后都会调用alloc_skb，但是第一个函数比较复杂，而且造成失败的原因比第二个函数还要多。这是因为，如果第一个缓冲区分配成功的话，后续的分配会失败的原因就比较少了。

另一个出现在本章众多函数中的数据结构就是和封包相关联的路由表缓存项目：rtable。很多函数都通过变量rt指向它。rtable里含有一些信息，诸如外出设备、外出设备的MTU，以及下一个跳点网关。此结构是由ip_route_output_flow初始化的，而且会在第三十六章加以说明。

ip_queue_xmit 函数

ip_queue_xmit是TCP和SCTP当前所用的函数。此函数只接收两个输入参数，而处理封包所需的所有信息都可通过skb存取（直接或间接）。

```
int ip_queue_xmit(struct sk_buff *skb, int ipfragok)
```

以下是其参数的意义：

skb

要传输的封包的缓冲区描述符。此数据结构有填入IP报头以及传输封包所需的所有参数（例如，下一个跳点网关）。记住，ip_queue_xmit用于处理本地产生的封包；转发封包没有相关的套接字。

ipfragok

主要由SCTP便用的标志，用来指出是否允许分段。

和skb相关的套接字包含有一个名为opt的指针，指向第十九章“选项的解析”一节所说过的一个结构。此结构包含IP报头中的选项，而其存储格式使得IP层的函数更易于存取。此结构是放在socket结构中的，因为此结构对每个要通过该套接字传输的封包而言都相同；替每个封包重建此信息就太浪费了。

```
struct sock *sk = skb->sk;
struct inet_sock *inet = inet_sk(sk);
struct ip_options *opt = inet->opt;
```

opt结构里有一些字段是偏移量，指出函数可以在报头中的哪些位置存储IP选项所需的时间戳和IP地址。注意，该结构并不缓存IP报头本身，而是只含一些数据，告诉我们要把什么写入报头，以及写入到何处。

设定路径

如果缓冲区已被设定正确路由信息（skb->dst），就没必要查询路由表。当缓冲区由SCTP协议处理时，在某些情况下这是有可能的：

```
rt = (struct rtable *) skb->dst;
if (rt != NULL)
    goto packet_routed;
```

在其他情况下，ip_queue_xmit会检查套接字结构中是否已缓存了一条路径。如果有的话，就会确定该路径依然有效（由_ _sk_dst_check检查）：

```
rt = (struct rtable *)_ _sk_dst_check(sk, 0);
```

如果套接字还没有一条缓存的路径可供封包使用，或者如果IP层迄今一直在用的路径在同一时间也失效了（例如来自于路由协议的更新），则ip_queue_xmit就必须利用ip_route_ output_flow寻找一条新路径，然后将结果存储到sk数据结构中。目的地是由daddr变量表示。首先，此变量设成封包的最终目的地（inet->daddr）；如果IP报头不含Source Route选项，此值即为正确值。然而，ip_queue_xmit会接着检查Source Route选项，如果存在，就把daddr变量设成源路径中的下个跳点（inet->faddr）。有Strict Source Route选项时，ip_route_output_flow找到的下个跳点必须和源路径列表中的下个跳点相吻合。

```
if (rt == NULL) {
    u32 daddr;

    daddr = inet->daddr;
    if(opt && opt->srr)
```

```
        daddr = opt->faddr;

    {
        struct flowi fl = { .oif = sk->sk_bound_dev_if,
                            .nl_u = { .ip4_u =
                                      { .daddr = daddr,
                                        .saddr = inet->saddr,
                                        .tos = RT_CONN_FLAGS(sk) } },
                            .proto = sk->sk_protocol,
                            .uli_u = { .ports =
                                       { .sport = inet->sport,
                                         .dport = inet->dport } } };

        if (ip_route_output_flow(&rt, &fl, sk, 0))
            goto no_route;
        }
        _ _sk_dst_set(sk, &rt->u.dst);
        tcp_v4_setup_caps(sk, &rt->u.dst);
    }
```

flowi数据结构的相关细节请参考第三十六章，而ip_route_output_flow函数的相关细节请参考第三十三章。

调用tcp_v4_setup_caps会把出口设备所提供的一些功能存储在套接字sk中；我们在讨论时，可以忽略此调用。

如果ip_route_output_flow失败了，封包会被丢弃。如果找到了路径，就会以_ _sk_dst_set存储在sk数据结构中，使其下次可以直接被使用，而路由表就不必再次被查询。如果因为某种原因该路径再次失效，就得再次调用ip_queue_xmit来通过ip_route_output_flow再次找出新路径。

如下列程序所示，如果IP报头携带Strict Source Routing选项，而且该选项所提供的下个跳点和路由表所传回的下个跳点不吻合，该封包就会被丢弃（注1）：

```
        skb->dst = dst_clone(&rt->u.dst);

    packet_routed:
        if (opt && opt->is_strictroute && rt->rt_dst != rt->rt_gateway)
            goto no_route;
```

调用dst_clone时，会对指派给skb->dst的数据结构上的引用计数值有所递增。

当封包被丢弃时，错误代码会返回给上层，而相关的SNMP统计数据也会被更新。注意，本例中，该函数不需要传送任何ICMP给来源地（我们就是来源地）。

注1： 第二十章“ip_forward函数”一节也看过类似的事情。

相反的，如果万事俱备，我们就拥有了传输封包所必需的所有信息了，接下来就该构建IP报头了。

构建IP报头

到目前为止，skb只含IP有效载荷——一般就是来自于L4层（不是TCP就是SCTP）的报头和有效载荷。这些协议总是会分配缓冲区，而缓冲区的大小都能够处理较低层报头增加时最糟糕的场景。如此一来，就能减少IP或其他较低层将必须去做的内存拷贝或缓冲区重新分配（以处理新增的报头不符合可用空间大小的情况）的机会。

当ip_queue_xmit接收skb时，skb->data会指向L3有效载荷（L4协议在此写入其数据）的开端。L3报头就在此指针之前。所以，这里会使用skb_push把skb->data往回移动，使其指向L3或IP报头的开端；此结果已在第十九章图19-2说明过了。iph也会初始化成该位置的指针。

```
iph = (struct iphdr *) skb_push(skb, sizeof(struct iphdr) +
                                (opt ? opt->optlen : 0));
```

下段代码是对IP报头中的一组字段做初始化。第一个赋值语句是同时设定三个字段的值（version、ihl以及tos），因为它们共享同一个16位。因此，这一语句把报头中的“版本”设为4，把“报头长度”设为5，并把TOS设为inet->tos。

有些用于对IP报头做初始化的值取自sk，而其他一些值则取自rt，这些都已在稍早的“本地流量相关的套接字数据结构”一节中说明过。

```
*((_ _u16 *)iph) = htons((4 << 12) | (5 << 8) | (inet->tos & 0xff));
iph->tot_len = htons(skb->len);
if (ip_dont_fragment(sk, &rt->u.dst) && !ipfragok)
    iph->frag_off = htons(IP_DF);
else
    iph->frag_off = 0;
iph->ttl      = ip_select_ttl(inet, &rt->u.dst);
iph->protocol = sk->sk_protocol;
iph->saddr    = rt->rt_src;
iph->daddr    = rt->rt_dst;
skb->nh.iph   = iph;
```

如果IP报头中包含了一些选项，此函数必须更新“报头长度”字段iph->length（它已被预先初始化为其默认值），然后调用ip_options_build处理这些选项。ip_options_ build会使用opt变量（它已被预先初始化为inet->opt）把所需的选项字段（诸如时间戳）添加至IP报头。注意，ip_options_build的最后一个参数被设为零，以指出该报头不属于片段（参见第十九章“IP选项”一节）。

```
if(opt && opt->optlen) {
    iph->ihl += opt->optlen >> 2;
    ip_options_build(skb, opt, inet->daddr, rt, 0);
}

mtu = dst_pmtu(&rt->u.dst);
```

接着，ip_select_ident_more会根据该封包是否可能被分段而在报头中设定IP ID(参见第二十三章“选择IP报头的ID字段”一节)，然后，ip_send_check会对IP报头计算校验和。

流量控制使用skb->priority来决定要把封包排入哪一个外出队列；而这一点有助于决定该封包多快会被传输。此函数中的值是来自sock结构的，而在ip_forward中(它管理非本地流量，并因此没有本地套接字)，其值是根据IP TOS的值，从一张转换表推导而得（参见第二十章“ip_forward函数”一节)。

```
ip_select_ident_more(iph, &rt->u.dst, sk, skb_shinfo(skb)->tso_segs);
ip_send_check(iph);
skb->priority = sk->sk_priority;
```

最后，会调用Netfilter来了解该封包是否有权跳至后续步骤（dst_output）并得以继续传输：

```
return NF_HOOK(PF_INET, NF_IP_LOCAL_OUT, skb, NULL, rt->u.dst.dev,
               dst_output);
```

ip_append_data函数

这是由那些想把传输的数据暂存于缓冲区的L4协议所使用的函数。如本章稍早所述，此函数并不传输数据，而是将数据放在大小合适的一些缓冲区中，让后续的函数可以借此构成一些片段（必要的话）并进行传输。因此，此函数并不建立或操作任何IP报头。要把由ip_append_data缓冲的数据刷新并传输，L4层必须显式地调用ip_push_pending_frames（也会处理IP报头）才行。

如果L4层想要快速的响应时间，每次调用ip_append_data之后，可能会再调用ip_push_pending_frames。但是，这两个函数是用于为了让L4层可以尽可能把多一点的数据暂存于缓冲区内（直到PMTU的尺寸)，然后一次传送，以获取效率。

由于具有准备封包的角色的缘故，ip_append_data暂存于缓冲区的数据只会到IP封包的最大尺寸。如第十八章“封包的分段/重组”一节所述，这一最大尺寸值为64 KB。

ip_append_data的主要任务是：

- 把来自于L4层的输入数据组成一些缓冲区，而这些缓冲区的尺寸又使其易于处理

IP分段工作（必要时）。此外，把那些数据片段放进那些缓冲区时，要还安排成让L3和L2层稍后能够轻易新增较低层协议头。

- 优化内存分配，把来自于上层的信息以及出口设备的能力考虑进来。特别地是：
  - 如果上层指出马上就有更多其他传输请求（通过MSG_MORE标志），分配大一点的缓冲区才合理。
  - 如果出口设备支持分散/聚集I/O（NETIF_F_SG），片段可以安排让内存的处理得以优化。
- 处理L4校验和。由第十九章“net_device结构”一节已知，skb->ip_summed是如何根据出口设备能力和其他因素而进行初始化的。

和ip_queue_xmit相比，ip_append_data的工作比较复杂，因此其原型较为复杂也并不令人惊讶：

```
int ip_append_data(struct sock *sk,
                   int getfrag(void *from, char *to, int offset, int len,
                               int odd, struct sk_buff *skb),
                   void *from, int length, int transhdrlen,
                   struct ipcm_cookie *ipc, struct rtable *rt,
                   unsigned int flags)
```

以下是输入参数的意义：

sk

此封包传输背后的套接字。此数据结构包含一些参数（诸如IP选项），稍后必须用于填写IP报头（由ip_push_pending_frames函数）。

from

指向L4层正试着传输的数据（有效载荷）的指针。这不是内核指针，就是用户空间指针。getfrag函数（接着说明）的工作就是正确处理这一指针。

getfrag

用于把接收自L4层的有效载荷拷贝到即将建立的一些数据片段中。更多细节参见“把数据拷贝到片段：getfrag”一节。

length

要传输的数据量（包括L4报头和L4有效载荷）。

transhdrlen

传输（L4）报头的尺寸。

ipc

正确转发封包所必需的信息。参见第二十三章“ipcm_cookie结构”一节。

rt

与此封包相关的路由表缓存项目（将在第三十六章说明）。当ip_queue_xmit自己接收此信息时，ip_append_data会依赖调用者通过ip_route_output_flow来收集该项信息。

flags

此变量可包含任何一个MSG_XXX标志（定义在*include/linux/socket.h*中）。此函数会用到其中三个标志：

MSG_MORE

此标志是由应用程序使用，来告知L4层马上就有更多其他传输。如我们所见，此标志会传播至L3层。稍后我们就会看到，在分配缓冲区时此项信息有何用处。

MSG_DONTWAIT

当此标志设定时，对ip_append_data的调用一定不能受到堵塞。ip_append_data可能必须为套接字sk分配一个缓冲区（利用sock_alloc_send_skb）。当sock_alloc_send_skb用掉其限额时，不是堵塞住（通过定时器）期望定时器到期前可以有些空间可用，不然就是失败。此标志可用于在前两个选项中做选择。

MSG_PROBE

当此标志设定时，用户其实不想传输任何东西，而只是在探测路径。例如，此标志可用于测试通往给定IP地址的路径上的PMTU（注2）。如果此标志已设定，ip_append_data只会立刻传回一个代表成功的返回代码。

ip_append_data是冗长复杂的函数。由于定义了很多局部变量（名称通常很相似），使其难以看懂。因此，我们将其分成几个主要步骤。因为可能的输出有很多不同组合（取决于本节开头所列举的一些考虑因素），我们只专注于少数常见情况。如此一来，你应该就可以自行推演其他情况。

接下来几节将说明ip_append_data的结果为何。之后的几节将说明该函数的初始任务，最后再说明其主循环。

注2：只看ip_append_data，MSG_PROBE是怎么用于测试PMTU的可能显得不太清楚。可参考*net/ipv4/raw.c*里的raw_sed_hdrinc作为范例。

图21-2～图21-7中所示的标签hh_len、exthdrlen、fraghdrlen、trailer_len、copy以及length不是ip_append_data的输入参数，就是ip_append_data所用的局部变量（特别地是，图中所示copy的值就是传给getfrag的值）。这些标签的值都以字节为单位表示。标签X、Y、S、S1以及S2代表一个数据区块的尺寸（单位为字节）。

针对ip_append_data的基本内存分配和缓冲区组织

了解ip_append_data的输出（片段要变成IP封包）在内存中是如何组织的，这一点很重要。本节以及后续两节将谈及组织输出数据的数据结构及其用法。这里的说明也适用于由L4层格式化并传给ip_queue_xmit的数据：例如，这是由TCP所做的事，而非使用ip_append_data完成的。在各种情况下，缓冲区最后都会交给dst_output（出现在第十八章图18-1中心附近）。我们来看一些范例。

ip_append_data可以建立一个或多个sk_buff实例，而每一个实例代表一个独立的IP封包（或IP片段）。无论数据在sk_buff里如何存储的（也就是无论其是否分段），都是如此。

假设我们想传输的数据量小于PMTU（也就是不必分段）。还假设因为我们的主机的配置的原因，必须至少使用IPsec套件中的协议之一。最后，为了简洁起见，假设我们分配缓冲区时不用达到内存最优化。因此，ip_append_data的结果（如图21-2所示）如下：

- 因为不需要分段，我们只分配一个缓冲区。
- IPsec套件的协议可能需要一个报头和一个报尾（trailer），包住原有的缓冲区（包括其原有的IP报头）。分配缓冲区时以及把L4层的数据拷贝到缓冲区时，必须将此考虑进来。
- 我们也会替L2层的报头预先分配空间（注3）。

L4层之后所有协议和分层所需的空间保留下来，就把稍后要耗费时间去做的内存操作工作消除掉了。此外还要注意，指向某些报头的指针（例如h.raw和nh.raw）会得到初始化；稍后，相关协议也可以将其所有的部分填入。封包中唯一由ip_append_data填入的部分是L4有效载荷。其他部分也会被填入，如下所示：

- L4报头会由调用ip_push_pending_frames的函数所填写。该函数可以直接被调用，或者通过包裹函数（例如，UDP使用udp_push_pending_frames）。
- L3报头（包括IP选项）会由ip_push_pending_frames填写。

注3：缓冲区左边的指针是sk_buff的字段，而右边则是ip_append_data的本地变量。

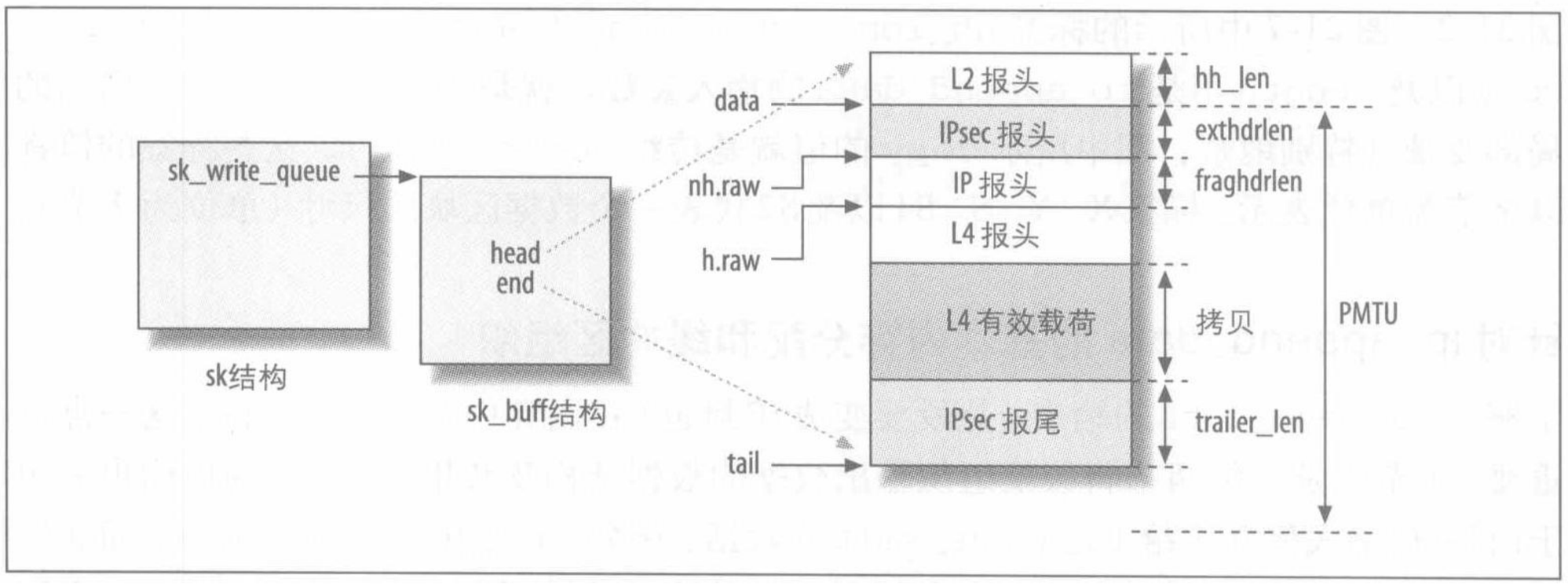


图 21-2：不需要分段的 IP 封包（使用 IPsec）

本书第六部分会讨论报头的 L2 部分。

现在，让我们考虑稍微复杂一点的例子，也就是需要分段。我们把前例的IPsec拿掉，然后增加有效载荷尺寸，使其超过 PMTU。图 21-3 所示是其输出结果（注 4）。

左下方的对象是ip_append_data从输入中所接收的缓冲区，而length是ip_append_data的另一个输入参数。此函数所建的两个缓冲区位在右边。注意，第一个包含了一个具有最大尺寸（PMTU）的片段，而第二个缓冲区包含了剩余的资料。ip_append_data会根据 PMTU 建立所需数量的缓冲区；这里碰巧第二个缓冲区持有所有剩余的有效载荷，而且尺寸小于 PMTU。

先前说过，ip_append_data将不会传输任何东西，只是建立一些稍后给封包片段使用的缓冲区。也就是说，L4 层可能会替前几例再次启用 ip_append_data，并新增更多数据。

我们以第二个例子说明会发生何事。因为第二个缓冲区已满，我们被迫分配一个新的缓冲区。这可能最后会得到次佳的分段；让每个片段（除了最后一个）都达到PMTU的尺寸是比较好的。

此时，实现最佳分段的简单解决办法就是分配另一个最大尺寸的缓冲区，从第二个缓冲区把数据拷贝到那儿，删除第二个缓冲区，再把新数据合并到新缓冲区。如果空间不够，我们可以分配第三个缓冲区。但是，这种做法无法提供良好的性能。这样就使得调用 ip_fragment 前要做数据分段的理由失效了（如第十八章图 18-1 所示），而 ip_fragment 可以避免额外的内存拷贝。

注 4：在“缓冲区分配”一节将会知道，如果 PMTU 不是 8 字节的倍数，所有片段（最后一个除外）的尺寸就会缩减至最近的 8 字节边界。

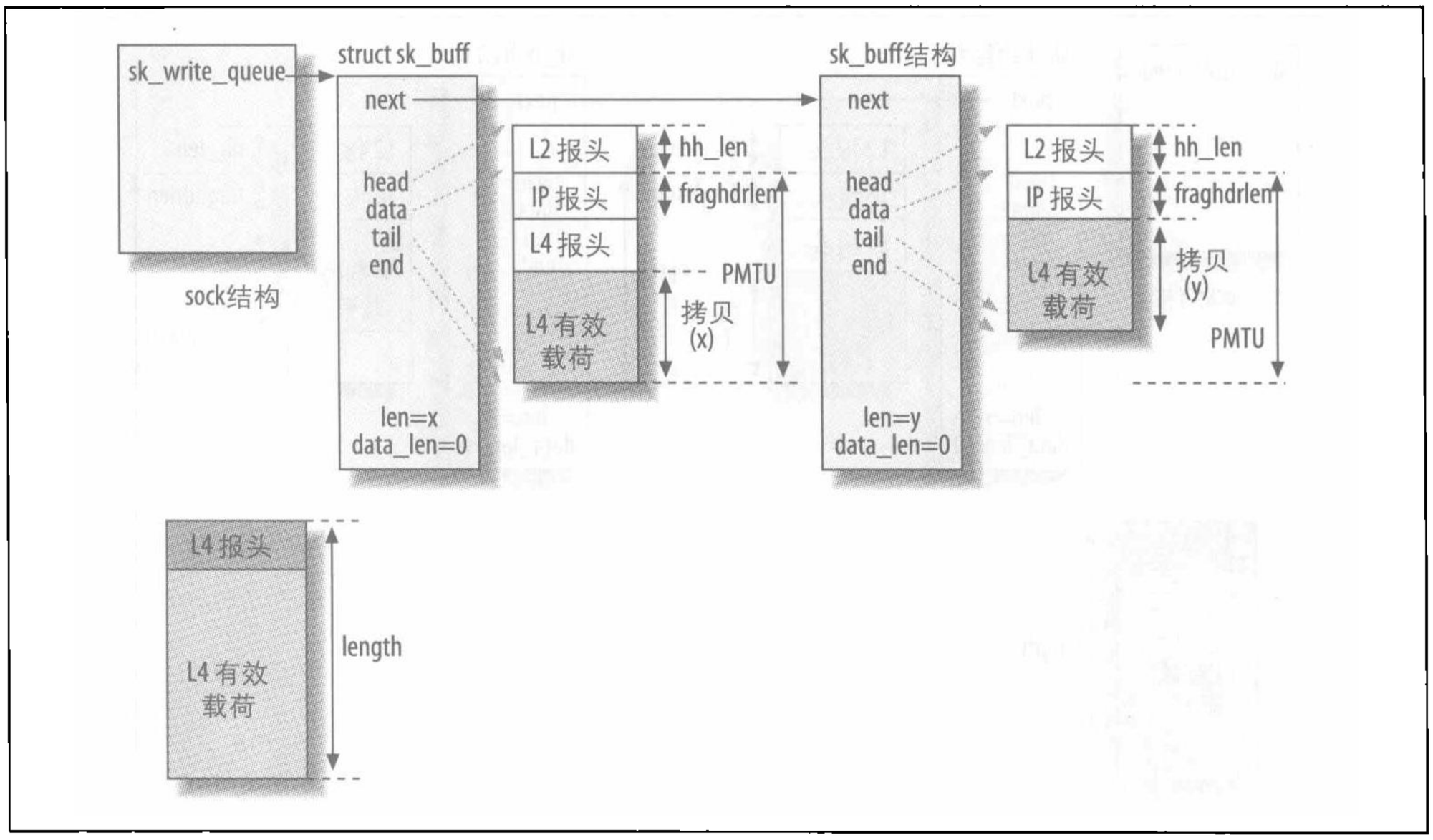


图 21-3：没有分散 / 聚集 I/O 的分段（无 MSG_MORE）

现在应该清楚为什么“ip_append_data 函数”一节所介绍的 MSG_MORE 标志会有用了。例如，如果在第二例中我们知道第二次调用就快来了，就可以直接分配拥有最大尺寸的第二个缓冲区，因而获得如图 21-4 所示的输出结果（注意，L2 报头的尺寸 hh_len 不算在 PMTU 之中）。

如果在 ip_push_pending_frames 调用之前，ip_append_data 又再次被调用，在分配第三个缓冲区之前，就会先试着在图21-4中所示的第二个缓冲区内的空白空间内填写数据。

针对 ip_append_data（有分散 / 聚集 I/O）的内存分配和缓冲区组织

偶尔，即使片段没有以最大尺寸分配，实际上还是有可能把数据添加进去。当设备支持分散/聚集I/O时，就有可能。这仅意味着L3层不去管L4层所放数据的那些缓冲区，让设备把这些缓冲区结合起来去做传输。分散/聚集I/O的优点是，它减少了分配内存和拷贝数据的消耗。

考虑一下：上层协议以连续的操作产生很多小数据项，而 L4 层将其存储在内核内存的不同缓冲区内。然后，L3 层被要求用一个 IP 封包传输所有这些项目。没有分散 / 聚集 I/O 时，L3 层就得把数据拷贝到新缓冲区来组成单一的封包。如果设备支持分散 / 聚集 I/O，数据可以一直留在其原地，直到离开主机。

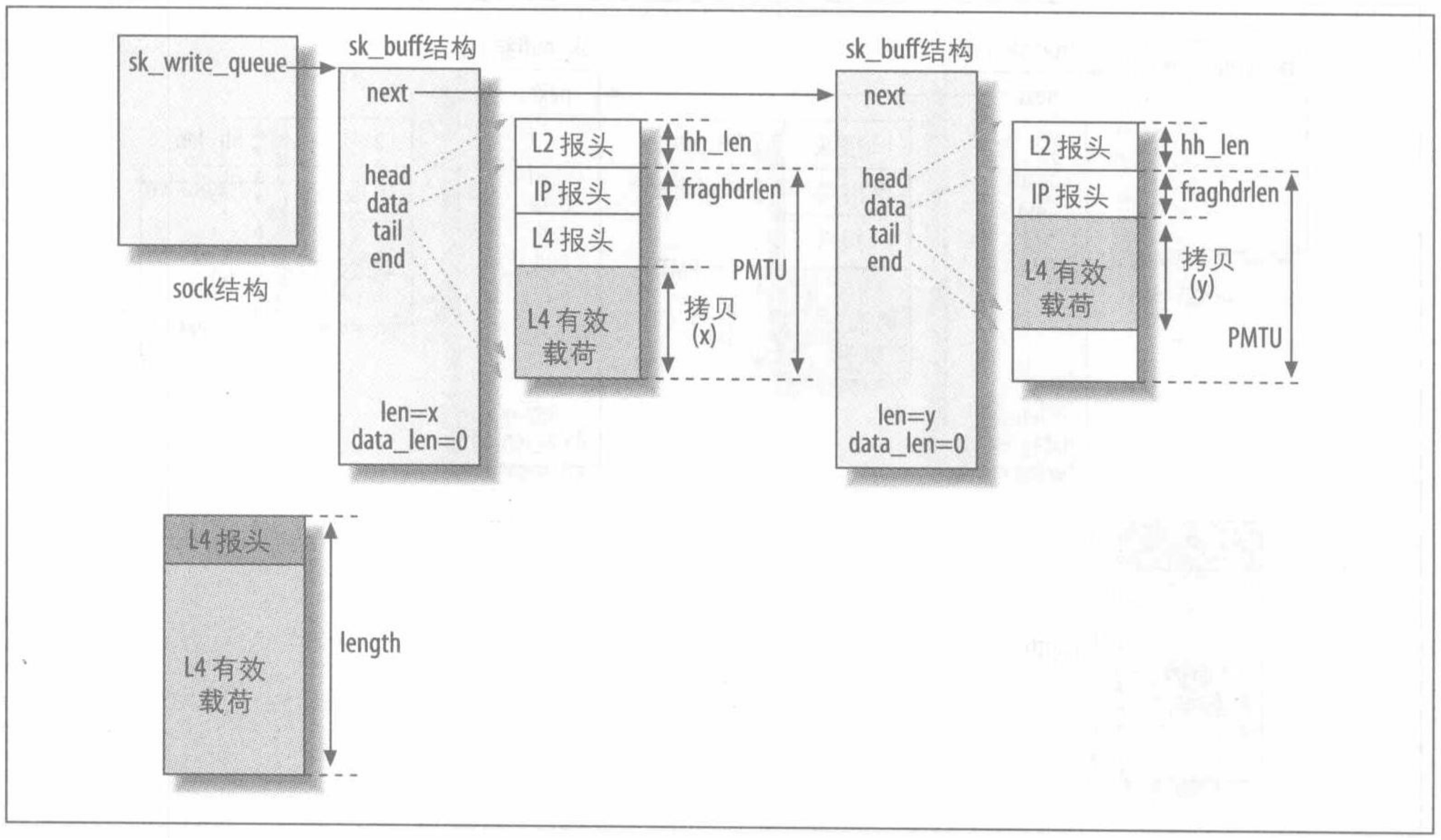


图 21-4：没有分散 / 聚集 I/O 的分段（MSG_MORE）

使用分散 / 聚集 I/O 时，skb->data 所指内存区域只会用那一次。接着的数据块会拷贝到特别为此而分配的内存页面中。图 21-5 和图 21-6 比较了当分散 / 聚集 I/O 开启和关闭时，ip_append_data 第二次调用时，数据是如何接收的。

- 图 21-5（a）所示是第一次调用后的内存使用情况，而图 21-5（b）所示是第二次调用之后的结果（分散 / 聚集 I/O 开启）。使用 frags 的缓冲区就称为页面缓冲区（paged buffer）。注意，图 21-5（b）所示的数据片段不需要任何报头：记住，一个 sk_buff 实例的所有数据片段都是和同一个 IP 封包相关。这也意味着 X+S1 依然小于 PMTU。
- 图 21-6（a）所示是第一次调用后的内存使用情况，而图 21-6（b）所示是第二次调用后的结果（分散 / 聚集 I/O 关闭）。

有些辅助数据结构可支持分散 / 聚集 I/O。除第一个（不支持分散 / 聚集 I/O 时，其分配方式也相同）之外的每个缓冲区都存储在 skb_shinfo(skb)->frags 中。这些缓冲区可以通过令人熟悉的 sk_buff 结构里的指针而找到。如第二章所说，每个 sk_buff 结构都包括一个类型为 skb_shared_info 的字段，该字段可以通过宏 skb_shinfo 存取。这个结构可用于增加缓冲区的尺寸 —— 新增内存区域（可位于任何地方，不见得要彼此相邻）。nr_frags 字段可协助 IP 层记住此封包挂着多少个分散 / 聚集 I/O 缓冲区。注意，此字段算的是分散 / 聚集 I/O 缓冲区，而不是 IP 片段（顾名思义）。

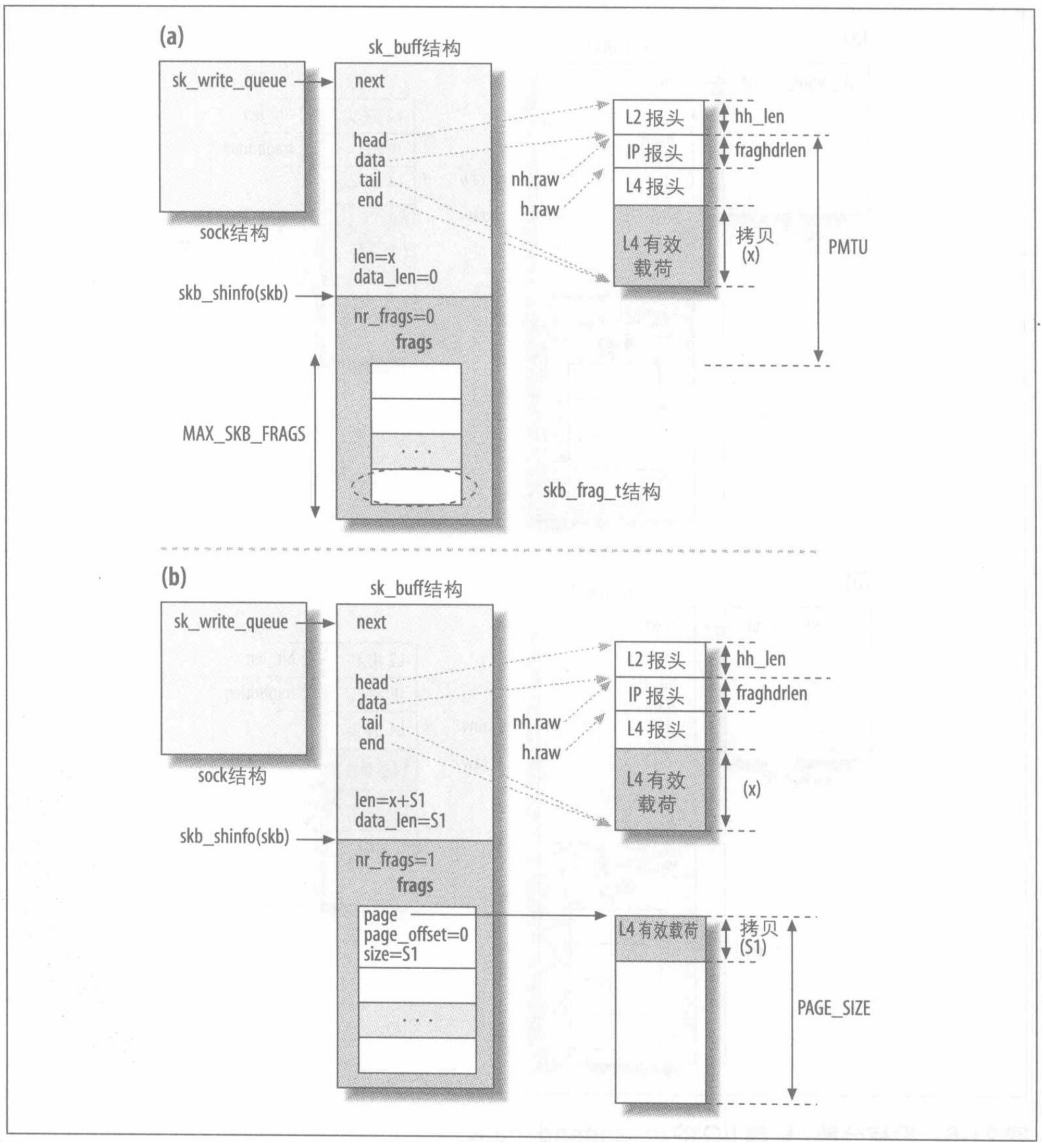


图21-5：有分散/聚集I/O的ip_append_data

现在我们可以谈一谈，为什么内核需要设备端这边有特殊支持才能使用这种缓冲区表达形式：为了能够引用一些不连续但是其内容假定代表一个连续的数据片段的内存区域，设备必须能够处理那种缓冲区表达形式。注意，图21-7所示是简单范例，其中有一个页面，内含两个相邻内存区域。但是，片段很容易就不相邻，不是在单一页面中，就是在不同页面中。

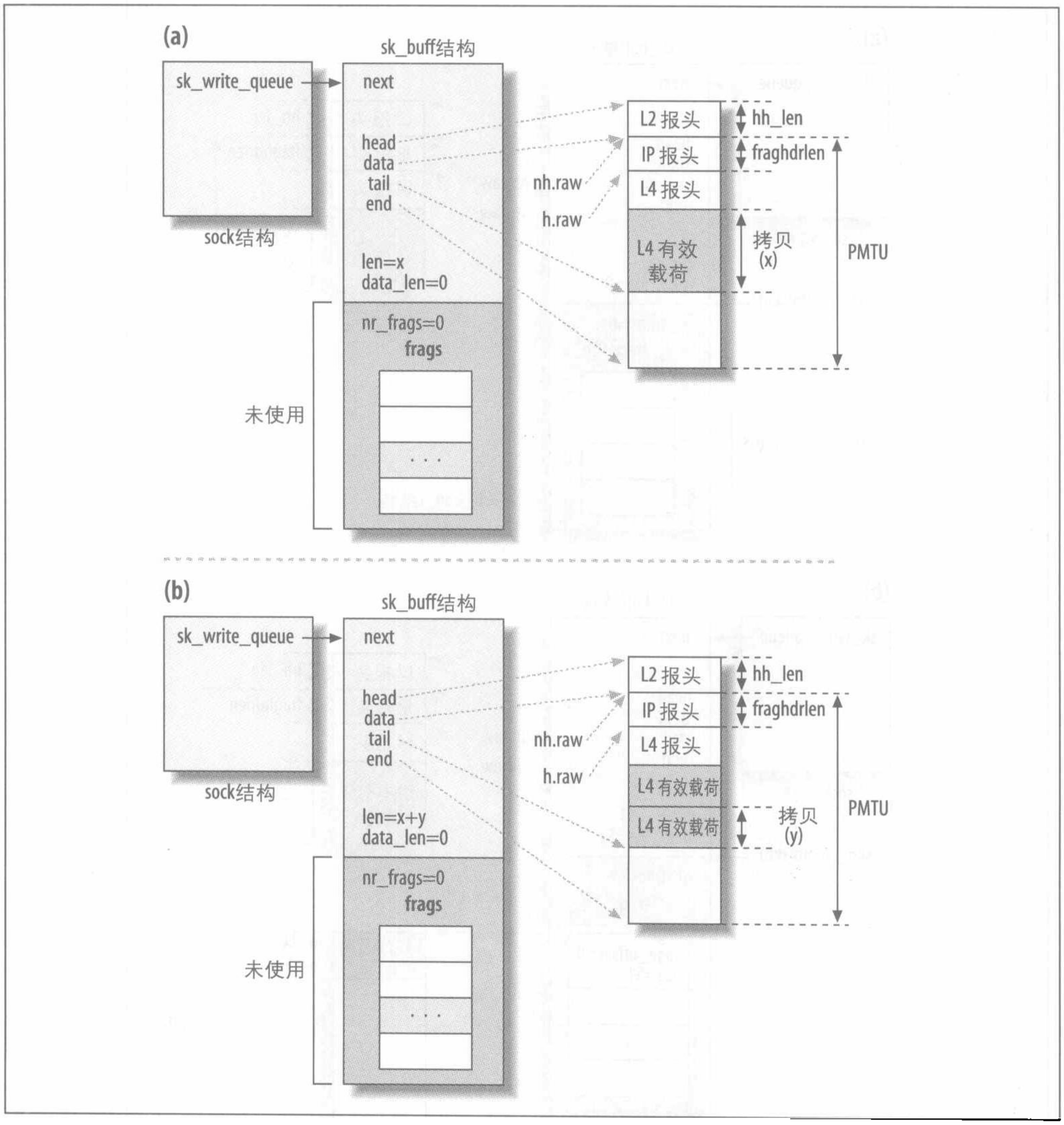


图 21-6：没有分散 / 聚集 I/O 的 ip_append_data

frags数组的每个元素都是由一个skb_frag_t结构表示，而此结构包括一个指向内存页面的指针、一个和页面开端相对的偏移量，以及该片段的尺寸。注意，因为图 21-7 中的两个片段位于同一个内存页面内，其页面指针都指向相同的内存页面。片段的最大数量是 MAX_SKB_FRAGS，而其定义基于一个 IP 封包的最大尺寸（64 KB）以及一个内存页面的尺寸（其定义因每种架构而异，对 i386 而言，其默认值为 4 KB）。

你可以在 *include/linux/sk_buff.h* 里找到前述所有结构的定义。

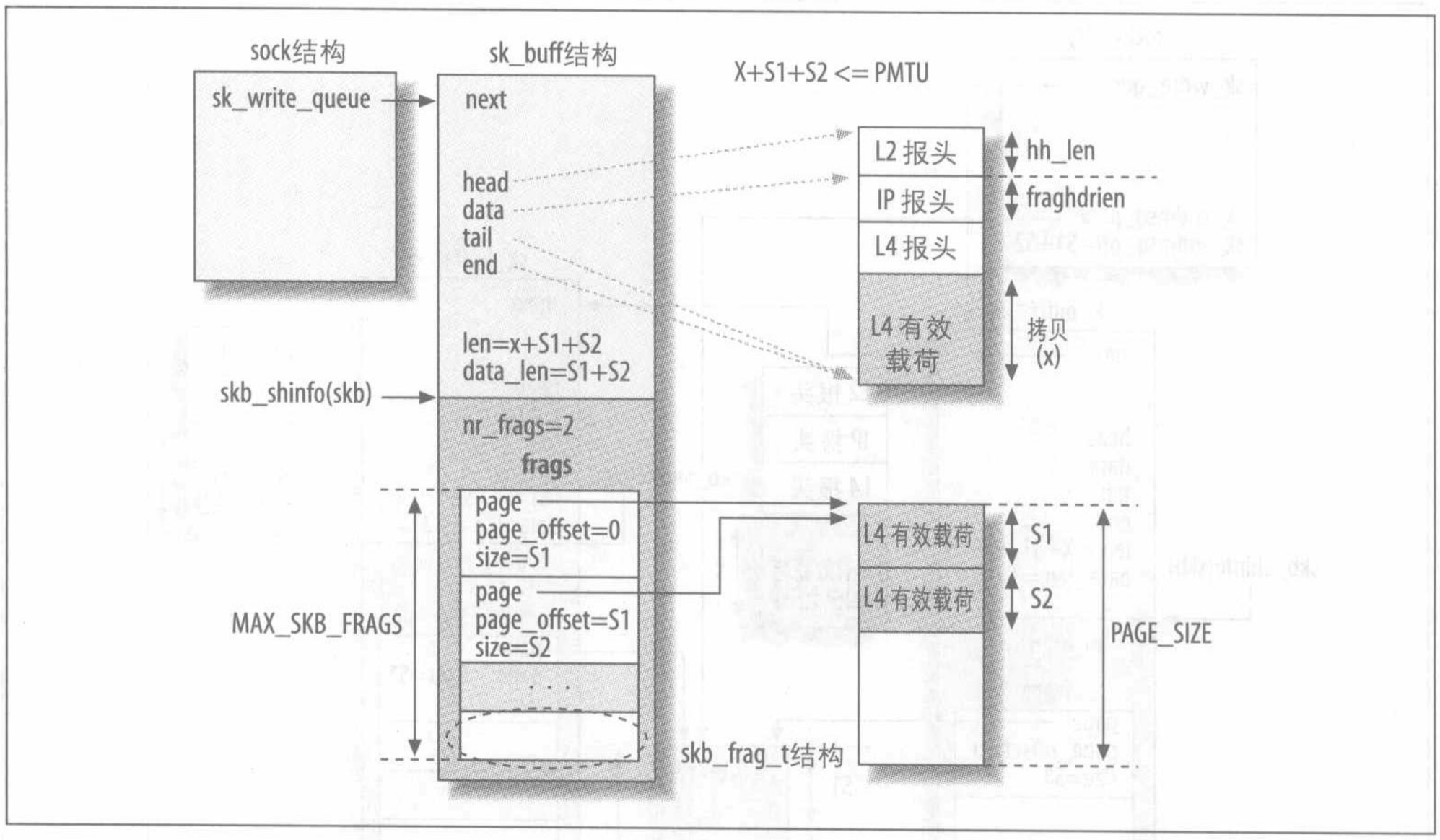


图21-7：好几个片段（有分散/聚集I/O）

图21-7所示是只有一个页面的情况，但是，因为有可能有好几个页面，frags数组的元素会包含指向正确页面的页面指针。一个片段不能横跨两个页面。当一个新片段的尺寸大于当前页面的可用空间量时，该片段就会切分成两部分：一部分到已存在的页面并填写进去，而第二部分就到一个新页面中去。

要谨记在心的一项重要细节是，分散/聚集I/O与IP数据分段是各自独立的。分散/聚集I/O只是让程序和硬件可以使用非相邻内存区域，就好像它们是相邻的那样。然而，每个片段依然必须尊重其最大尺寸（PMTU）的限制。也就是说，即使PAGE_SIZE大于PMTU，当sk_buff里的数据（由skb->data所指）加上由frags所引用的数据来到PMTU时，就需要建立一个新的sk_buff。

此外还要注意，同一个页面可以持有不同IP片段的数据片段，如图21-8所示。每个加入内存页面的数据片段都会递增该页面的引用计数。当IP片段最终被传送出去后，页面里的数据片段就会被释放，而引用计数值也会据以递减，而且该内存页面也会被释放（参见skb_release_data，它由kfree_skb间接调用）。

图21-8左上角的sock结构包括一个指向最后页面的指针（sk_sndmsg_page），以及一个位于该页面内的偏移量（sk_sndmsg_off），该页面也就是下一个数据片段要放的位置。

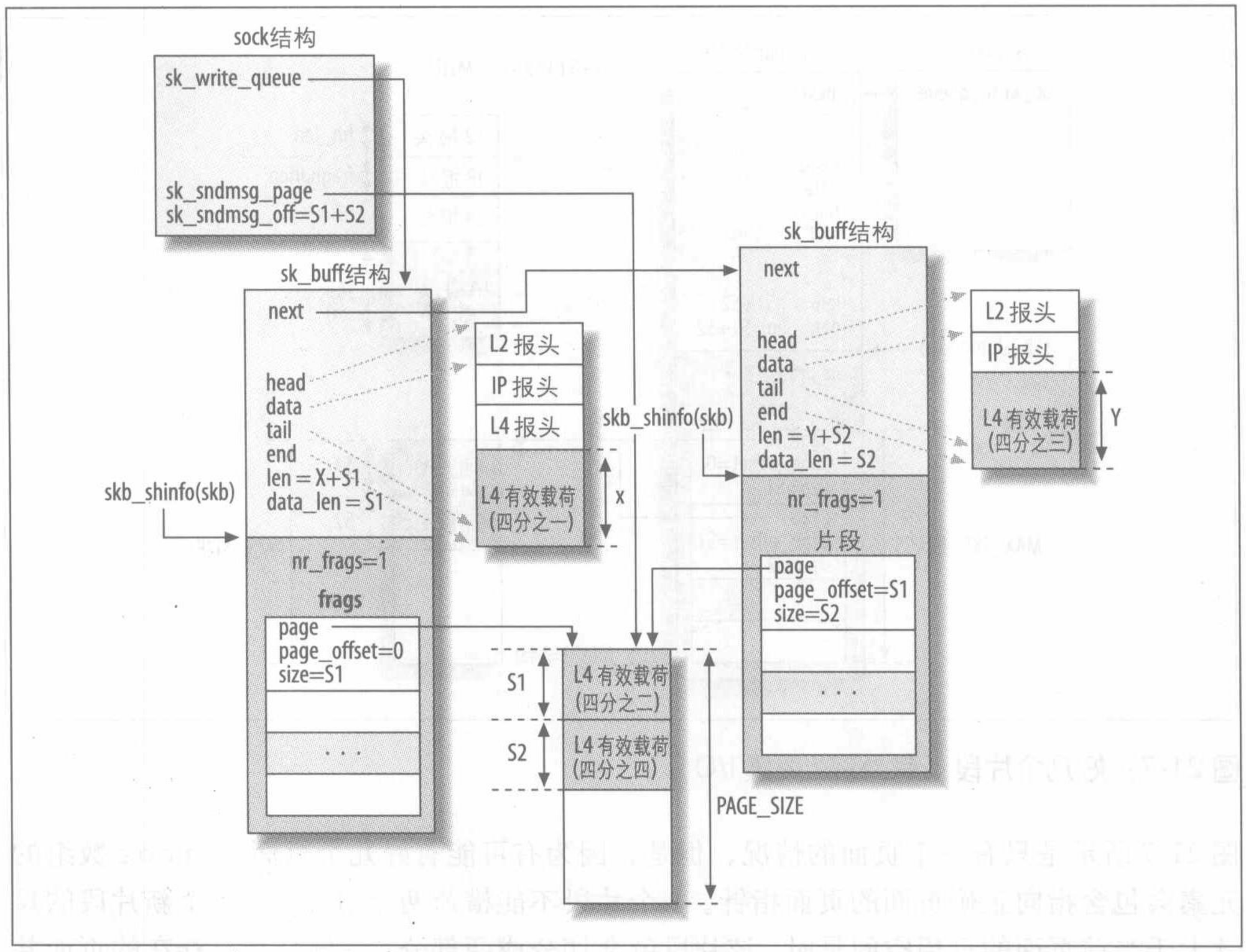


图 21-8：一些 IP 片段所共享的内存页面

处理片段缓冲区的重要函数

要理解本章及第二十二章所述的函数，必须对第二章介绍的几个重要的缓冲区操作函数以及下列几个函数都很熟悉才行：

`skb_is_nonlinear`

当缓冲区被分段时就返回 true（即 `skb->data_len` 不为 NULL）。

`skb_headlen`

给定一个分段缓冲区，就会返回主要缓冲区中的数据量（即不计算 `frags` 片段，也不考虑 `frag_list` 列表）。别把 `skb_headlen` 误认为 `skb_headroom`：`skb_headroom` 返回的是介于 `skb->head` 和 `skb->data` 之间的可用空间。

`skb_pagelen`

分段缓冲区的尺寸；会把主缓冲区中的数据以及 `frags` 片段中的数据计算进来，但是，不考虑任何链接至 `frag_list` 列表的缓冲区。

图21-9所示是2个范例。注意，skb->len包括frags（在ip_append_data里更新）和frag_list（在ip_push_pending_frames里更新）里的数据片段。我省略掉了关于协议报头的细节，因为就我们的讨论而言，用不到那些细节。

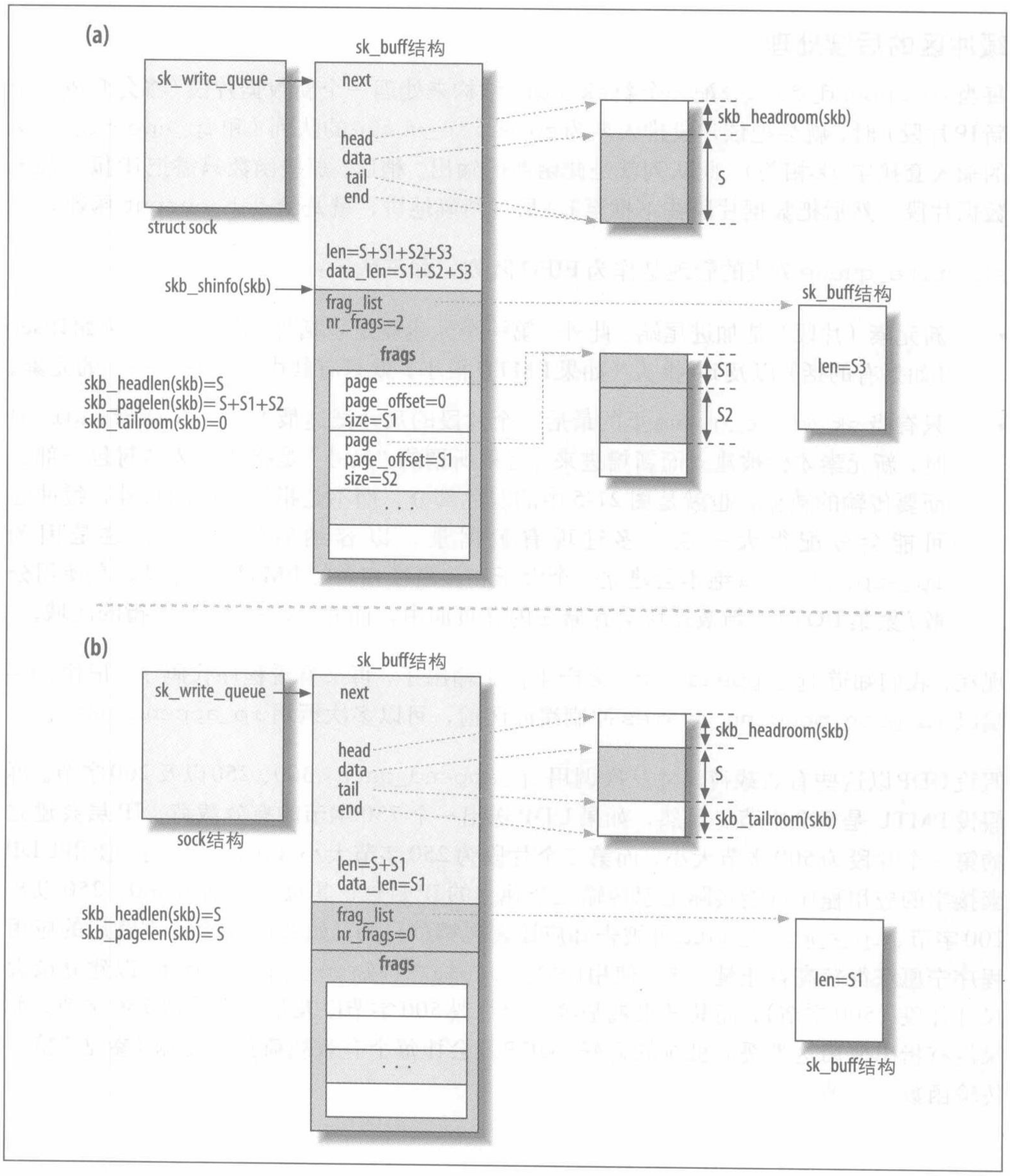


图21-9：针对片段缓冲区的重要函数：（a）散播/聚集；（b）无散播/聚集

我也要再次强调一点：frags向量里的数据是主缓冲区中数据的扩展，而frags_ list里的数据代表的是独立缓冲区（也就是每一个缓冲区都必须作为单独的IP片段而独立传输）。

缓冲区的后续处理

每当ip_append_data分配一个新sk_buff结构来处理一个新数据片段（将会变成一个新IP片段）时，就会把该片段排入名为sw_write_queue的队列（和ip_append_data的输入套接字sk相关）。此队列就是此函数的输出。稍后，那些函数只需把IP报头加入数据片段，然后把数据片段往下推至L2层（精确地讲，就是给dst_output函数）。

sk_write_queue列表的管理是作为FIFO队列，如下所示：

- 新元素（片段）是加进尾端。此外，第一个元素就是包括外部报头——例如IPsec（如果有的话）以及L4报头（如果PMTU很小，就只有其中一部分）——的元素。
- 只有当sk_write_queue里的最后一个片段的尺寸抵达最大尺寸（maxfraglen）时，新元素才会被建立而新增进来（这里所谓的“尺寸”是指要作为该封包一部分而要传输的数据，也就是图21-5中的灰色部分，而不是指缓冲区的尺寸，缓冲区可能会分配得大一点，多过现有数据量，以容纳后续数据）。这是因为ip_append_data绝不会建立一个大于和该路径相关的PMTU的片段。在使用分散/聚集I/O时，新数据块会存储在内存页面中，而非skb->data所指的区域。

现在，我们知道ip_append_data会产生什么输出了，再来看看程序代码了。记住，L4层以ip_push_pending_frames冲刷缓冲区前，可以多次调用ip_append_data。

假设UDP以这些有效载荷尺寸三次调用ip_append_data：300、250以及200字节。再假设PMTU是500字节。显然，如果UDP送出一个750字节的有效载荷，IP层会建立的第一个片段为500字节大小，而第二个片段为250字节大小（注5）。然而，使用UDP套接字的应用程序可能实际上想传输三个独立的IP封包，其尺寸分别为300、250以及200字节。ip_append_data可被告知应该表现哪种行为。如果UDP套接字背后的应用程序宁愿获得较高吞吐量，就会使用MSG_MORE标志告知ip_append_data以建立最大尺寸片段（500字节），而其结果就是第一个片段500字节以及第二个片段250字节。如果没有指出使用这类缓冲机制的喜好，UDP就会让每个有效载荷独自传输（参见“整合传输函数”一节）。

注5：为了简洁起见，我忽略了报头所需消耗。

设定上下文

ip_append_data函数的第一个代码块是对一些本地变量做初始化，而且可能会修改一些输入参数。实际所做的工作取决于该函数是否建立封包内的第一个IP片段（此时，sk_write_queue队列应为空）或者是后续片段。就第一个元素而言，ip_append_data会利用一些字段对inet->cork和inet做初始化，ip_append_data后续调用时（以及ip_push_pending_frames）都会用到这些字段。

所存储的信息中有IP选项以及路由表缓存选项。将其缓存，可以在后续因相同封包而数次调用ip_append_data期间节省时间，但是因为ip_append_data的调用者会在后续所有调用中把数据再次传进去，所以，也不见得非得这样。

```
if (skb_queue_empty(&sk->sk_write_queue)) {
    opt = ipc->opt;
    if (opt) {
        if (inet->cork.opt == NULL) {
            inet->cork.opt = kmalloc(sizeof(struct ip_options) + 40,
                                     sk->sk_allocation);
                    if (unlikely(inet->cork.opt == NULL))
                        return -ENOBUFF;
        }
        memcpy(inet->cork.opt, opt,
               sizeof(struct ip_options)+opt->optlen);
        inet->cork.flags |= IPCORK_OPT;
        inet->cork.addr = ipc->addr;
    }
    dst_hold(&rt->u.dst);
    inet->cork.fragsize = mtu = dst_pmtu(&rt->u.dst);
    inet->cork.rt = rt;
    inet->cork.length = 0;
    sk->sk_sndmsg_page = NULL;
    sk->sk_sndmsg_off = 0;
    if ((exthdrlen = rt->u.dst.header_len) != 0) {
        length += exthdrlen;
        transhdrlen += exthdrlen;
    }
} else {
    rt = inet->cork.rt;
    if (inet->cork.flags & IPCORK_OPT)
        opt = inet->cork.opt;
    transhdrlen = 0;
    exthdrlen = 0;
    mtu = inet->cork.fragsize;
}
```

要理解该函数后续部分，必须了解下列重要变量的意义。其中有些变量是由ip_append_data从输入中接收而得的；参考“ip_append_data函数”一节的说明。回顾图21-2～图21-8也会有帮助。

rt

用于传输此IP数据段的路由表缓存项。此结构包括一些字段，诸如下个跳点网关、出口设备以及 PMTU 等。

mtu

和 rt 相关的 PMTU。

opt

要加入 IP 报头的 IP 选项。当此变量为 NULL 时，就没有选项。

exthdrlen（*外部报头* len）
transhdrlen（*传输报头* len）

当L4层调用 ip_append_data 时，会传入这两个参数，因为分配缓冲区时必须考虑这两个函数。transhdrlen 是直接传入的，而 exthdrlen 是由 rt 间接取得的。外部报头的范例是那些由 IPsec 套件里的协议所使用的报头，例如鉴定报头（Authentication Header，AH）以及封装安全有效载荷报头（Encapsulation Security Payload，ESP）。传输报头的范例就是那些常见的 TCP、UDP 以及 ICMP 协议的报头。

初始化 length、exthdrlen 以及 transhdrlen 的方式可能令人困惑。我将会说明为何在某些情况下其值会改变。

如我们所见，只有第一个片段必须包含传输报头和选用的外部报头。因此，建立第一个片段之后，transhdrlen 和 exthdrlen 都会变零。后面就会看到，如果 sk_write_queue 不为空，此函数于开头处就可做此事，或者在大 while 循环内做此事（第二轮循环开始之前）。

因为有此初始化，此函数会使用 transhdrlen 的值以区分第一个片段和后续的片段：

- transhdrlen ! = 0 表示 ip_append_data 工作在第一个片段。
- transdhrlen = 0 表示 ip_append_data 未工作在第一个片段。

这种逻辑不适用于 exthdrlen，因为每个 IP 封包都需要 L4 报头，但是很多都没有外部报头，因为没用到诸如 IPsec 的特殊功能。

此处所初始化的一些变量，稍后会有几项重要用途：

- 在决定要把多少数据拷贝到每个数据片段时，此函数必须把第一个片段包含 L4 报头和选用的外部报头的情况考虑进来，因此，有效载荷的可用空间就比较少（参见图 21-2 所示）。

- 在决定缓冲区要分配到多大时，此函数必须考虑外部报头（如果有的话）所需的额外空间。
- 在对nh.raw和h.raw指针做初始化时，此函数必须知道是否有外部报头以及其位在何处，才能正确计算出于封包内的偏移量。

准备产生片段

稍后就会知道，拷贝到每个已产生的片段的数据量会随片段而异。然而，每个片段总是有固定的L2和L3报头部分。图21-2～图21-8都显示出此保留部分。

继续之前，此函数会定义下列三个本地变量：

```
hh_len = LL_RESERVED_SPACE(rt->u.dst.dev);
fragheaderlen = sizeof(struct iphdr) + (opt ? opt->optlen : 0);
maxfraglen = ((mtu - fragheaderlen) & ~7) + fragheaderlen;
```

hh_len是L2报头的长度。在替IP之前的所有报头在缓冲区内保留空间时，ip_append_data必须知道L2报头需要多少空间。如此一来，当设备驱动程序对其报头做初始化时，就不需要重新分配空间，或者移动缓冲区内的数据来腾出空间给L2报头。

fraghdrlen是IP报头（包括IP选项）的尺寸，而maxfraglen是IP片段的最大尺寸（基于路径的PMTU）。

如第十八章“封包的分段/重组”一节所述，IP封包（报头加有效载荷）的最大尺寸是64 KB。这一点不仅适用于个别片段，也适用于整个封包（这些片段最终会重组成此封包）。于是，ip_append_data会记录为特定封包所接收的所有数据，并拒绝超过64 KB（0xFFFF）的限制。

```
if (inet->cork.length + length > 0xFFFF - fragheaderlen) {
    ip_local_error(sk, EMSGSIZE, rt->rt_dst, inet->dport, mtu-exthdrlen);
    return -EMSGSIZE;
}
inet->cork.length += length;
```

最后一项初始化是校验模式（其值存储在skb->ip_summed中）。参见“L4校验和”一节。

把数据拷贝到片段：getfrag

任何L4协议都可能用到ip_append_data。此函数任务之一就是把输入数据拷贝到其所创建的片段中。不同的协议可能必须对所拷贝的数据实行不同的操作。这类专用操作实例之一就是L4校验和的计算（对某些L4协议而言无强制性）。另一个区别是数据的

源头。也就是本地产生的封包是指用户空间，而转发封包或者由内核产生的封包（例如ICMP消息）则是内核空间。

利用适合各个协议需求的多个小函数，比让一个共享的函数处理各种协议以及要施加的可选操作的所有可能组合，会更容易而明确。为了让ip_append_data尽可能通用，此函数允许每协议通过输入参数getfrag指定要用于拷贝数据的函数。换而言之，ip_append_data使用getfrag把输入数据拷贝到缓冲区内；此拷贝的结果就是图21-2～图21-9中标有“L4有效载荷”的内存区域。

表21-1列出了最常见L4协议（调用ip_append_data）所用的函数。ip_send_reply会用到另一个函数ip_reply_glue_bits（参见“进行传输的重要函数”一节）。

getfrag接收四个输入参数（from、to、offset以及len），而且只会从from拷贝len个字节到to+offset，同时把from可能是指向用户空间内存的指针考虑进来，因此，必须相应地加以处理（可能需要从用户内存转移至内核内存）。此外，此函数也会处理L4校验和：把数据拷贝进内核缓冲区时，会根据skb->ip_summed的配置更新skb->csum。

表21-1：一些getfrag函数

协议	API
ICMP	icmp_glue_bits
UDP	ip_generic_getfrag
RAW IP	ip_generic_getfrag
TCP（通过ip_send_reply）	ip_reply_glue_bits

当getfrag函数的输入数据的源头（用户空间与内核）一直都相同时，此函数就不用区分这两种情况。例如：

- 传输消息时，ICMP协议会用到icmp_glue_bits。因为ICMP消息不是由内核所建，就是从先前已接收的另一条ICMP消息推演而得（因此也是在内核内存内），则icmp_glue_bits就知道数据是在内核空间中。
- 当应用程序对UDP或Raw IP套接字发出一个sendmsg系统调用时，内核最后会调用ip_append_data，把ip_generic_getfrag当成getfrag函数传入。这种情况下，已经知道输入数据始终来自用户空间。

我们来更近地看一下通用函数ip_generic_getfrag：

```
int
ip_generic_getfrag(void *from, char *to, int offset, int len, int odd,
                   struct sk_buff *skb)
{
    struct iovec *iov = from;
    if (skb->ip_summed == CHECKSUM_HW) {
        if (memcpy_fromiovecend(to, iov, offset, len) < 0)
            return -EFAULT;
    } else {
        unsigned int csum = 0;
        if (csum_partial_copy_fromiovecend(to, iov, offset, len, &csum) < 0)
            return -EFAULT;
        skb->csum = csum_block_add(skb->csum, csum, odd);
    }
    return 0;
}
```

第十九章“sk_buff结构”一节解释过CHECKSUM_HW的意义，以及skb->csum和skb->ip_summed是如何使用的。在“L4校验和”一节中，我们将会说明ip_append_data如何决定L4校验和应该在硬件中还是软件中计算（或者根本不计算）。在上述代码片段中，你可以看见ip_generic_getfrag使用两个不同函数去拷贝数据（memcpy_fromiovecend和csum_partial_copy_fromiovecend）；取决于L4校验和将在硬件中计算还是必须在软件中计算。

缓冲区分配

ip_append_data是根据下列因素选择要分配的缓冲区尺寸：

单一传输与多次传输

如果ip_append_data获知很快就会有其他传输请求（如果设定了MSG_MORE），就有理由分配大一点的缓冲区，使得后续传输的数据可以合并至同一个缓冲区。参见稍早“针对ip_append_data的基本内存分配和缓冲区组织”一节的说明。

分散/聚集I/O

如果设备可以处理分散/聚集I/O，片段就可以更有效率的方式存储至内存页面。参见稍早“针对ip_append_data（有分散/聚集I/O）的内存分配和缓冲区组织”一节的说明。

下列程序片段会根据刚才所提两点决定出要分配的缓冲区的大小（alloclen）。如果预期还有更多数据，而且如果设备无法处理分散/聚集I/O，此缓冲区就会以最大尺寸建立（根据PMTU）。如果这些条件都不满足，则缓冲区的大小只要能够容纳当前数据就行了。

```
if ((flags & MSG_MORE) &&
    !(rt->u.dst.dev->features&NETIF_F_SG))
```

```
        alloclen = mtu;
    else
        alloclen = datalen + fragheaderlen;

    if (datalen == length)
            alloclen += rt->u.dst.trailer_len;
```

注意，当ip_append_data产生最后片段时，就必须考虑一些报尾（例如IPsec的报尾）是否存在。

datalen是要拷贝到我们所分配的缓冲区的数据量。其值会根据三个因素预先初始化：剩余数据量（length），一个片段所能容纳的最大数据量（fraghdrlen），以及一个可有可无、来自于前一个缓冲区的间隙（fraggap）。

关于最后一个元素fraggap，需要做点解释。除了最后一个缓冲区（持有最后一个IP片段）以外，所有片段都必须尊循IP片段的有效载荷的尺寸必须是8字节的倍数这一规则。因此，当内核分配的一个新缓冲区不是给最后片段使用时，可能必须从前一个缓冲区的尾端移动一段数据（其尺寸为0～7个字节）到新分配缓冲区的头部。换而言之，除非下列所有条件都为真，否则 fraggap 为零：

- PMTU 不是 8 字节的倍数。
- 当前 IP 片段的尺寸还不到 PMTU。
- 当前 IP 片段的尺寸已超过 8 字节的最高倍数（此最高倍数小于或等于 PMTU）。

图21-10所示的范例中fraggap不为零，而alloclen已初始化为mtu。注意，当内核将数据从当前缓冲区skb_prev移到新缓冲区skb时，也必须调整skb_prev和skb的L4校验和（参见“L4校验和”一节）。为了简洁起见，图21-10把缓冲区表示成两个平坦的内存区域，但是，它们可以有分页（如图21-5所示）或无页面（如图21-6所示）：用于移动fraggap区域的函数skb_copy_and_csum_bits可以处理这两种格式。而同一个函数也会更新L4校验和。

主要循环

while循环可能会建立额外缓冲区，看起来可能比实际还要复杂。图21-11中总结了其工作。

最初，length的值代表的是ip_append_data的调用者想传输的数据量。然而，一旦进入循环，其值就代表剩余要处理的数据量。这说明了为什么其值会在每轮循环结尾被更新，以及ip_append_data为何一直循环，直到length变为零。

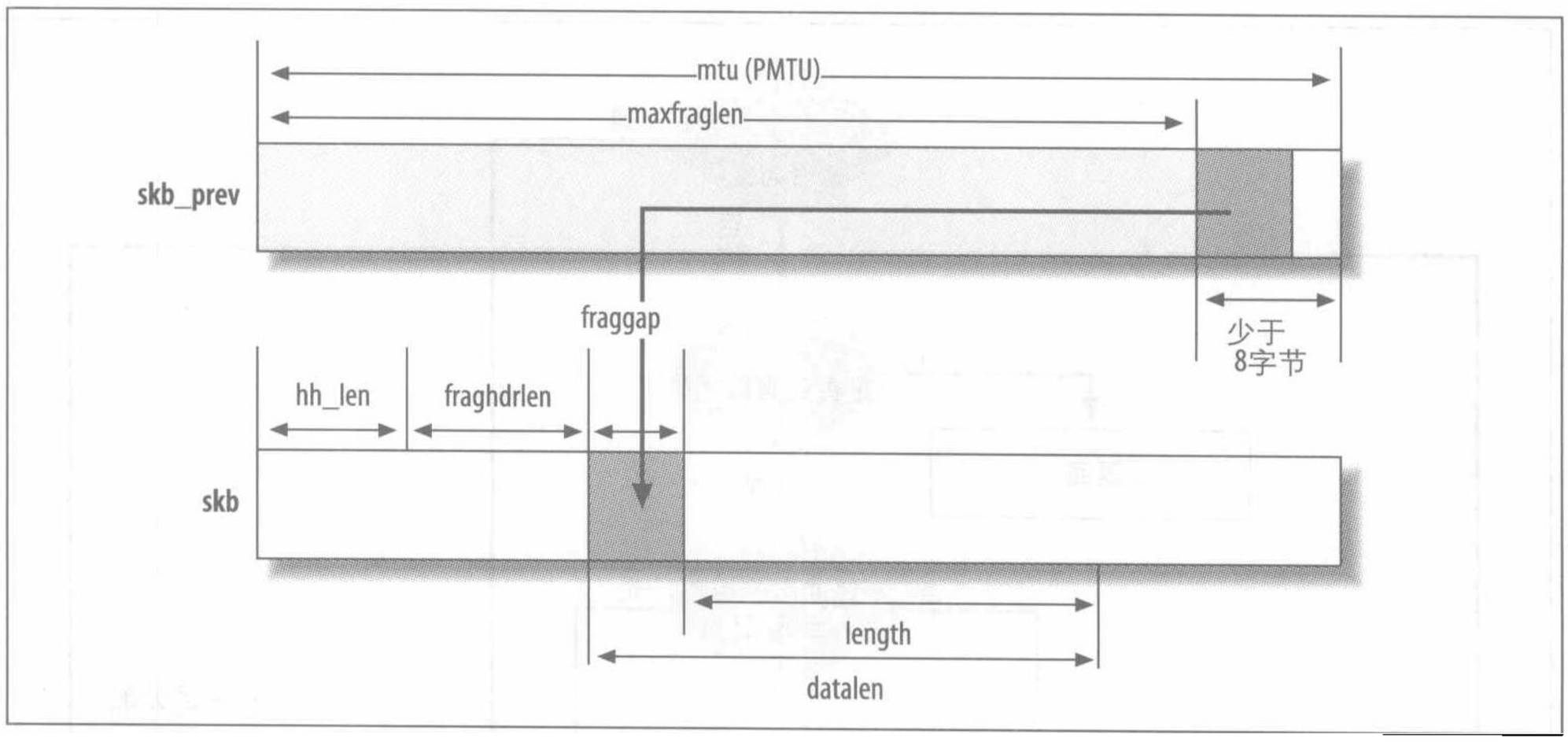


图21-10：尊重IP片段的8字节边界

我们已知MSG_MORE指出了L4层是否预期还有更多数据，而NETIF_F_SG指出了设备是否支持分散/聚集I/O。这些设定值对循环里的的第一项任务（也就是循环内第一个if区段中分配及初始化sk_buff结构）并无影响。此外，第一个数据片段总是会拷贝到sk_buff区域，参见图21-5（a）～图21-6（a）所示。

每当下列情况之一发生时，ip_append_data就会分配一个新的sk_buff结构，并将它排入sk_write_queue队列：

- sk_write_queue是空的（也就是第一个片段）。
- sk_write_queue的最后一个元素已完全填满。

循环前的那段程序会处理第一种情况，当队列为空时就强迫分配：

```
if ((skb = skb_peek_tail(&sk->sk_write_queue)) == NULL)
    goto alloc_new_skb;
```

循环里的第一部分会处理第二种情况。首先，把copy初始化为当前IP片段中所剩的空间量：mtu - skb->len。如果要加进来的剩余数据量（length）大于可用空间量copy，就需要另一个IP片段。这种情况下，copy会受到更新。为了实行8字节边界规则，copy会缩减到最近的8字节边界。此时，内核会决定是否必须分配一个新的缓冲区（也就是新的IP片段）。以下是利用if条件式比较copy和0的相关逻辑：

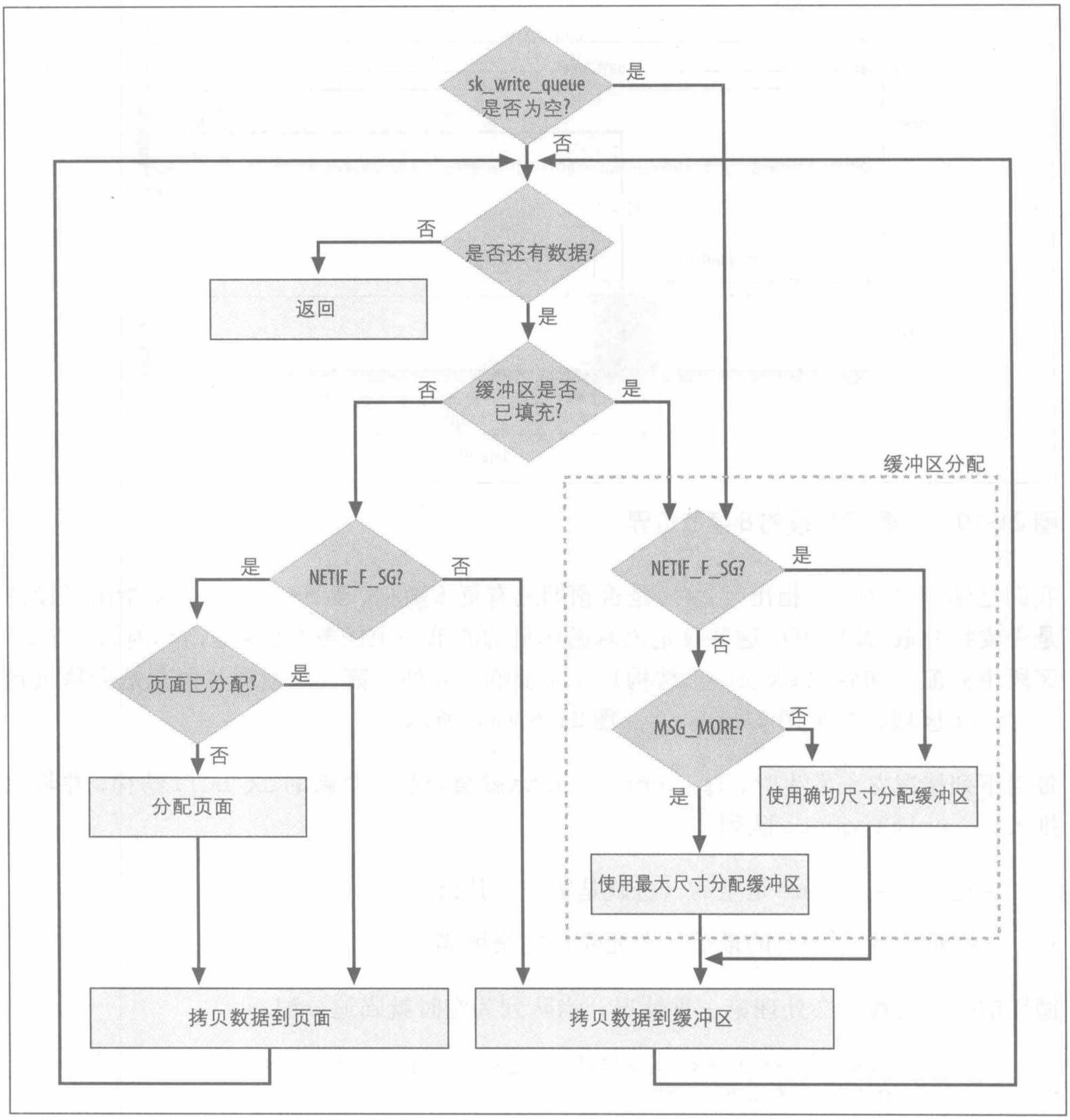


图 21-11：ip_append_data 函数：主要循环

copy > 0

这意味着skb(sk_write_queue的最后一个元素)有一些可用空间。ip_append_data会先用那些空间。如果所剩空间不够（也就是length大于该可用空间），则循环会再次迭代，而这一次会进入下一情况，参见图 21-3～图 21-4 所示。

copy = 0

这意味着此时该分配一个新的sk_buff，因为最后一个已被完全填满。这种情况下，if区段内的程序会分配一个新缓冲区，把输入数据拷贝到该缓冲区，然后把新片段排入sk_write_queue队列。后续片段不是和前一个片段合并，就是拷贝到专为分散/聚集I/O所分配的内存页面中去。

copy < 0

这是前一种情况的特例。当copy为负时，表示有些数据必须从当前IP片段中删除，并移动至新片段。更多细节参见稍早“缓冲区分配”一节。

每次一轮循环结束时，此函数必须把指向要拷贝的数据的指针（offset）向前移，然后更新剩余要拷贝的数据量(length)。一旦片段被_ _skb_queue_tail排入队列后，如果还剩下任何数据，此函数就必须重启循环。

L4校验和

由第十九章“net_device结构”一节已知，当出口NIC的设备驱动程序通过在dev->features中设定正确标志而主张其能力时，L3和L4校验和就可由出口NIC计算。特别地是，必须初始化skb->ip_summed（以及最后skb->csum），来指出出口设备是否提供L4硬件校验和计算功能。更多细节参考前面几节。

是否可以使用硬件校验和计算，取决于何时调用ip_append_data处理第一个片段（也就是transhdrlen为非零）。只有当下列所有条件都吻合时，才可用硬件校验和计算：

- 由ip_append_data所构建的IP封包不会被分段（也就是送给ip_append_data的总数据不会超过PMTU）。
- 出口设备支持硬件校验和计算。
- 没有转换报头（也就是IPsec集组的协议）。举例言之，这类转换可以在计算校验和时，把NIC应该读取的数据压缩或加密。这些转换也会把其他报头插入在IP报头和L4报头之间。这意味着，L4硬件校验和计算和IPsec转换不能同时存在。

在其他情况下，硬件校验和计算可能也必须关闭。

对于前述列表的第一项，需要一点说明。当IP封包被分段时，硬件校验和计算就行不通（如图21-3中的范例）。然而，因为实际传输发生前（也就是ip_push_pending_frames被调用前），ip_append_data可以被调用多次，当ip_append_data第一次被调用时，IP层不会知道是否需要分段，因此，最初决策只能基于输入数据（length）：如果根据length确定需要分段，就不会使用硬件校验和计算。

```
if (transhdrlen &&
    length + fragheaderlen <= mtu &&
    rt->u.dst.dev->features&(NETIF_F_IP_CSUM|NETIF_F_NO_CSUM|NETIF_F_HW_CSUM) &&
    !exthdrlen)
        csummode = CHECKSUM_HW;
```

这里初始化的局部变量csummode会被指派给第一缓冲区的skb->ip_summed。如果需要分段，而且ip_append_data据此分配更多的缓冲区（每个IP片段一个缓冲区），则后续缓冲区的skb->ip_summed就会被设成CHECKSUM_NONE。在调用getfrag把数据拷贝到缓冲区时，如果传入的缓冲区的skb->ip_summed被初始化为CHECKSUM_NONE，那么，getfrag也会处理L4校验和（参见“把数据拷贝到片段：getfrag”一节）。

注意，ip_append_data只对L4有效载荷计算校验和。由第十八章“对L4校验和所做的修改”一节已知，L4校验和必须包括L4报头和所谓的伪报头。当sk_write_queue只有一个IP片段而且出口设备支持硬件校验和计算时，如果L4层调用了ip_push_pending_ frames，则L4协议只需把skb->csum初始化成正确的偏移量，并用伪报头校验和设定L4报头的校验字段（如第十九章“sk_buff结构”一节所见）。相反，如果出口设备不支持硬件校验和计算，或者虽然支持，但是因为sk_write_queue有一个以上的IP片段而无法使用，则L4校验和就必须在软件中计算。这种情况下，getfrag把数据拷贝到缓冲区时，会对L4有效载荷计算部分校验和，然后，L4协议稍后会将这些值结合起来得到要放到L4报头内的值。参见“整合传输函数”一节，来了解在调用ip_push_pending_ frames前UDP如何处理L4校验和。

有关设备驱动程序如何指示NIC在必要时计算L4硬件校验和的范例，可参考*drivers/net/3c59x.c*里的boomerang_start_xmit函数以及*drivers/net/8139cp.c*里的cp_start_xmit。就此两者而言，你也可以了解一下，建立DMA传输时已分页的skb是如何处理的。

ip_append_page函数

由“把数据拷贝到片段：getfrag”一节已知，来自于用户空间的传输请求（有类似sendmsg之类的调用）需要做拷贝，把要传输的数据从用户空间移往内核空间。这样的拷贝是由作为输入参数而传给ip_append_data的getfrag函数完成的。

内核提供另一个接口给用户空间应用程序：sendfile；此接口允许应用程序优化传输并拷贝数据。通常称呼这个接口为“零拷贝”TCP/UDP。

只有当出口设备支持分散/聚集I/O时，才可以使用sendfile接口。从这种情况下，由ip_append_data所实现的逻辑就可以简化，不需要任何拷贝（也就是用户请求传输的数据可以留在那儿）。内核只需用输入数据中所接收的数据缓冲区位置对frag向量做

初始化，然后必要时处理L4校验和就行了。这种简化的逻辑就是ip_append_page所提供的东西。尽管ip_append_data接收到的数据位置带有void*指针，ip_append_page会接收的位置是指向一个内存页面的指针，此外，还有位于该内存页面内的偏移量，因此要对frag的一个项目做初始化就相当简单。

就分散/聚集I/O而言，和ip_append_data唯一不同的代码片段如下：

```
i = skb_shinfo(skb)->nr_frags;
if (len > size)
    len = size;
if (skb_can_coalesce(skb, i, page, offset)) {
    skb_shinfo(skb)->frags[i-1].size += len;
} else if (i < MAX_SKB_FRAGS) {
    get_page(page);
    skb_fill_page_desc(skb, i, page, offset, len);
} else {
    err = -EMSGSIZE;
    goto error;
}

if (skb->ip_summed == CHECKSUM_NONE) {
    unsigned int csum;
    csum = csum_page(page, offset, len);
    skb->csum = csum_block_add(skb->csum, csum, skb->len);
}
```

在把一个新片段加入一个页面时，ip_append_page会首先试着把新片段和已存在于页面中的前一个片段合并起来。为此，它会首先检查新片段应该加入的位置是否和最后一个片段结尾的位置相吻合（通过skb_can_coalesce）。如果可能合并，它需要做的就是更新页面中已存在的前一个片段的长度来包含新数据。

当合并不可能时，此函数会先用skb_fill_page_desc对新片段做初始化。从这种情况下，它也会以get_page递增对该页面的引用计数值。该引用计数一定要被递增，因为ip_append_page会使用作为输入接收的那个页面，而且该页面也可能由他人使用。

ip_append_page当前只由UDP使用。我们说过，TCP不使用ip_append_data和ip_push_pending_frames函数，因为它在tcp_sendmsg里实现相同的逻辑。这个零拷贝接口也是如此：TCP不使用ip_append_page，而是在do_tcp_sendpage里实现相同逻辑。和UDP不同的是，只有当出口设备支持L4硬件校验和计算时，TCP才会让应用程序使用零拷贝接口（注6）。

注6：如果设备不需要L4校验和，也可以使用零拷贝。参见第十九章有关NETIF_F_NO_CSUM的描述。

ip_push_pending_frames 函数

如本章开头所述，ip_push_pending_frames 的运作和 ip_append_data 及 ip_append_page 有前后关系。当 L4 层决定是时候把排在 sw_write_queue 里的那些片段（通过 ip_append_data 或 ip_append_page）打包起来传输时（也许是因为某个协议特定的准则，或者因为较高层应用程序通知说要传送数据），就会调用 ip_push_pending_ frames：

```
int ip_push_pending_frames(struct sock *sk)
```

这个函数会接收一个 sock 结构。这个函数必须访问一些字段，主要是通过指向该套接字的 sk_write_queue 结构的指针。

由“针对 ip_append_data（有分散/聚集 I/O）的内存分配和缓冲区组织”一节已知，根据是否使用分散/聚集 I/O，封包中数据的组织和在 sk_buff 结构中的组织会有所不同。

这部分的程序会把第一个缓冲区之后的所有缓冲区都排入一个 frag_list 链表（第一个元素的一部分），如图 21-12 所示，并更新该链表头部的缓冲区的 len 和 data_len 字段以便于查核所有片段。进行最后一项操作是因为对程序路径后头的 ip_fragment 函数有所帮助（参见第十八章图 18-1 以及第二十二章）。因为缓冲区是排入 frag_list 的，所以会被 sk_write_queue 清出。建立新链表所需时间很短（没有拷贝数据，只修改指针），而结果就是释放 sk_write_queue 链表，使得 L4 层认定数据已传输。现在，数据已流出 L4 层之手，完全在 IP 层的掌控下了。

记住，当你查看图 21-12 时，nr_frags 反映出的是分散/聚集 I/O 缓冲区的数目，而非 IP 片段的数目。关于图 21-12 有两点值得一提：

- 图 21-12（a）所示范例中到 ip_push_pending_frames 的输入数据反映的是没有散播/聚集的情况（也就是没有使用 frags 向量）。有散播/聚集时，应该会有一个如图 21-7 所示的缓冲区链表。
- 图 21-12（b）中只有缓冲区使用了 skb_shinfo 时才会显示其区块，不过，其他所有 sk_buff 结构都有 skb_shinfo。

接下来，该填写 IP 报头了。如果有多个片段，则只有第一个片段的 IP 报头是由 ip_push_pending_frames 填写；其余片段则是在稍后才处理（第二十二章会说明）。

IP 报头的 TTL 字段（iph->ttl）的设定取决于目的地地址是否为多播。通常，多播流量所用的值较小，因为多播最常用于传递流（以及偶尔交互）数据，例如音频和视频（如

果接收得太晚，就没用了）。就多播和单播封包而言，指派给TTL字段的默认值分别为1和64（注7）。

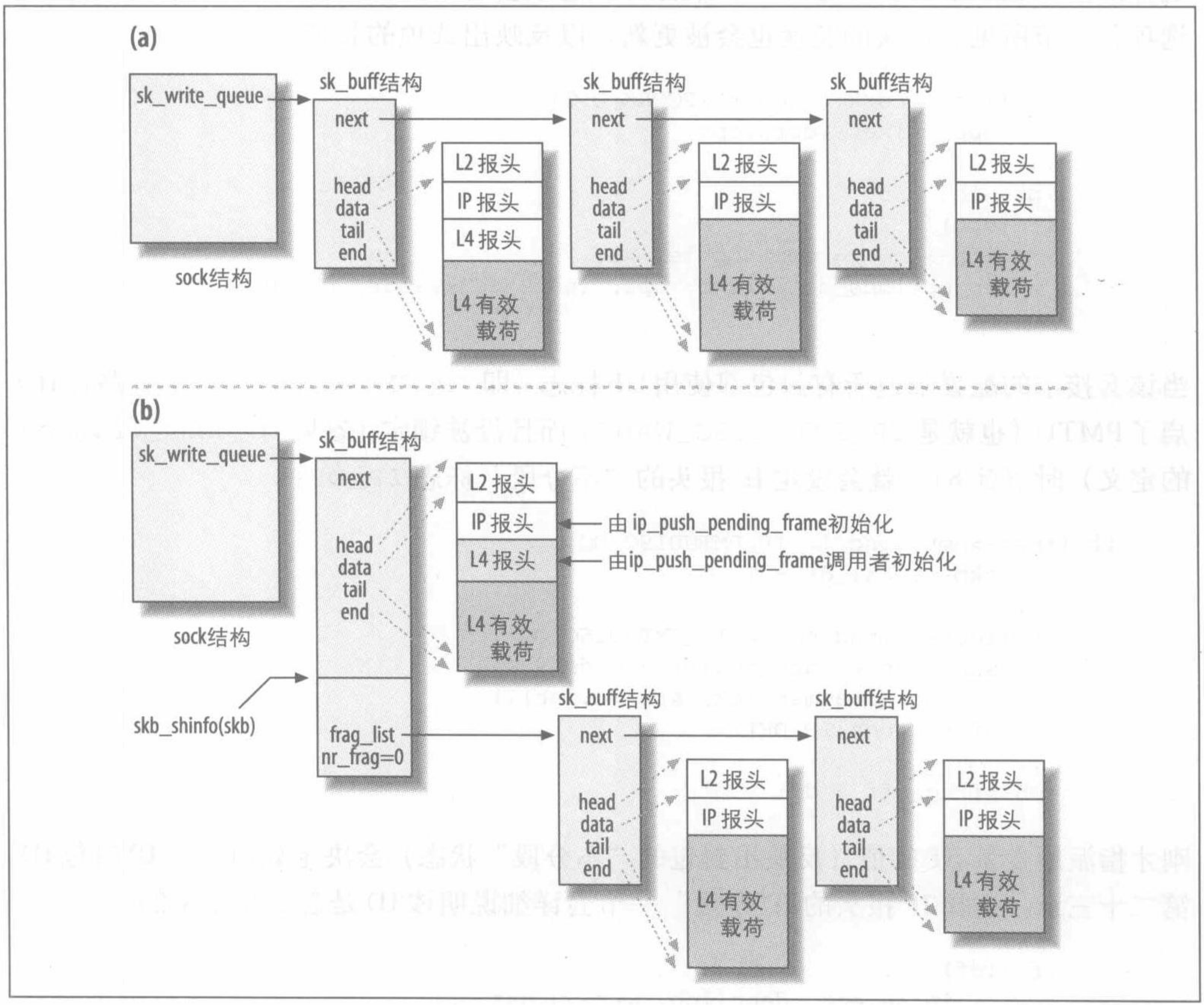


图21-12：（a）从sk_write_queue队列删除缓冲区之前；（b）删除之后

```
if (rt->rt_type == RTN_MULTICAST)
    ttl = inet->mc_ttl;
else
    ttl = ip_select_ttl(inet, &rt->u.dst);
...
iph->ttl = ttl;
```

注7：这两个都可以用ip_setsocketopt修改，但是，只有单播的值可以用*/proc*接口设定（参见第二十三章“通过/proc文件系统做调整”一节）。

如果报头中有IP选项，则会用ip_options_build处理那些选项。给ip_options_build的最后一个输入参数是零，以告诉该API，它正在填写的是第一个片段的选项。这样的区别是必要的，因为第一个片段的IP选项会有不同的对待方式，如第十八章“IP选项”一节所见。报头的长度也会被更新，以反映出选项的长度。

```
if (inet->cork.flags & IPCORK_OPT)
    opt = inet->cork.opt;
...
iph->ihl = 5;
if (opt) {
    iph->ihl += opt->optlen>>2;
    ip_options_build(skb, opt, inet->cork.addr, rt, 0);
}
```

当该套接字的配置强迫所有封包都使用DF标志（即IP_PMTUDISC_DO），且路径rt开启了PMTU（也就是IP_PMTUDISC_WANT）而且没被锁定（参见ip_dont_fragment的定义）时（注8），就会设定IP报头的“不分段”标志IP_DF：

```
if (inet->pmtudisc != IP_PMTUDISC_DO)
        skb->local_df = 1
...
if (inet->pmtudisc == IP_PMTUDISC_DO ||
    skb->len <= dst_mtu(&rt->u.dst) &&
    ip_dont_fragment(sk, &rt->u.dst)))
    df = htons(IP_DF);
...
iph->frag_off = df;
```

刚才指派给变量df的值（反映出封包的“不分段”状态）会决定如何设定IP封包ID。第二十三章“选择IP报头的ID字段”一节会详细说明该ID是怎么算出来的。

```
if (!df) {
    _ _ip_select_ident(iph, &rt->u.dst, 0);
} else {
    iph->id = htons(inet->id++);
}
```

Traffic Control会使用skb->priority来决定要把该封包排入哪一个外出队列。参见“构建IP报头”一节中ip_queue_xmit所做的类似的初始化。

```
iph->version = 4;
iph->tos = inet->tos;
iph->tot_len = htons(skb->len);
iph->protocol = sk->sk_protocol;
```

注8：PMTU是可以指派给路径的量度（metrics）之一。当一个量度被锁住时，协议事件就无法修改。第三十章和第三十六章会介绍量度。

```
iph->saddr = rt->rt_src;
iph->daddr = rt->rt_dst;
ip_send_check(iph);
skb->priority = sk->sk_priority;
skb->dst = dst_clone(&rt->u.dst);
```

把缓冲区传给dst_output以完成传输之前，此函数必须取得Netfilter的权限才能做这件事。注意，只会为一个封包中的所有片段查询一次Netfilter。稍早以前的内核版本（2.4）中，每个片段都会查询Netfilter。这样可以让Netfilter有机会以较高的细度去过滤IP封包，但是也强迫Netfilter重组封包并重新对封包分段，以免有些过滤器要检查L4或较高层次的内容。这样会得不偿失。

注意，当把一个sk_buff缓冲区链表（和单一缓冲区相反）传给dst_input时，如图21-12（b）所示，只有第一个缓冲区的IP报头会被初始化。第二十二章就会看到，ip_fragment是怎么处理这种链表的。

```
err = NF_HOOK(PF_INET, NF_IP_LOCAL_OUT, skb, NULL,
              skb->dst->dev, dst_output);
```

返回之前，此函数会清掉IPCORK_OPT字段，使得cork结构的内容失效。这是因为后续传往相同目的地的封包会重复使用cork结构，因此，IP层必须知道何时该把旧数据丢掉。

整合传输函数

要了解我们所见的几个函数（ip_append_data和ip_push_pending_frames）如何合作，先来关注一个由UDP层所调用的函数udp_sendmsg，了解它如何调用那些函数。

```
int udp_sendmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
    size_t len)
{
    ... ... ...
    struct udp_opt *up = udp_sk(sk);
    ... ... ...
    int corkreq = up->corkflag || msg->msg_flags&MSG_MORE;
    ... ... ...
    err = ip_append_data(sk, ip_generic_getfrag, msg->msg_iov, ulen,
        sizeof(struct udphdr), &ipc, rt,
        corkreq ? msg->msg_flags|MSG_MORE : msg->msg_flags);
    if (err)
        udp_flush_pending_frames(sk);
    else if (!corkreq)
        err = udp_push_pending_frames(sk, up);
```

局部标志corkreq的初始化取决于多个因素，而此标志会传给ip_append_data，用于指出是否应该使用缓冲区机制。其中一些因素如下：

MSG_MORE

此标志可以在每次传输请求时单独设定或清除。

corkflag (UDP_CORK)

这个标志只会对套接字设定一次，然后一直保持有效，直到显式地被关闭为止。

这两个标志有比较的目的。讨论过哪个最佳之后，这两个最终都在内核内实现了。

udp_sendmsg 会首先调用 ip_append_data，然后只有 corkreq 为 false 时以 udp_push_pending_frames 强迫立即进行数据传输。如果 ip_append_data 因某个原因失败了，udp_sendmsg 会用 udp_flush_pending_frames（内含 IP 函数 ip_flush_pending_frames 的包裹函数）刷新该队列。

图 21-13 所示是 udp_push_pending_frames 的内部逻辑。注意 L4 校验和是如何根据“L4 校验和”一节所述的逻辑被处理的。

有关如何使用 ip_append_page 的范例，可以参考 udp_sendpage。

raw 套接字

raw 套接字（使用 raw IP 的套接字）把 IP 报头包含在传给 IP 层的数据内，是有可能的。也就是说，可以要求 IP 层传送一段数据，而该段数据内已包含一个已初始化的 IP 报头。为此，raw IP 使用 IP_HDRINCL（包含报头）选项；举例言之，可以用 setsockopt 系统调用设定（参见 ip_setsockopt 函数）。

当设定了此选项时，既不会使用 ip_push_pending_frames，也不会使用 ip_queue_xmit。相反的，raw IP 会直接调用 dst_output。参考 raw_sendmsg 和 raw_send_hdrinc 函数。

衔接邻居子系统

如第十八章图 18-1 所示，传输会以调用 ip_finish_output 结束。ip_finish_output 只是一个简单的包裹函数，内含一个 Netfilter 钩子。注意，ip_finish_output 的命名规则与 *do_something* 及 *do_something*_finish 不同，而是遵循了 *do_something* 及 *do_something*2 的规则。ip_finish_output2 会在第二十七章“邻居协议和 L3 传输函数的交互”一节说明。

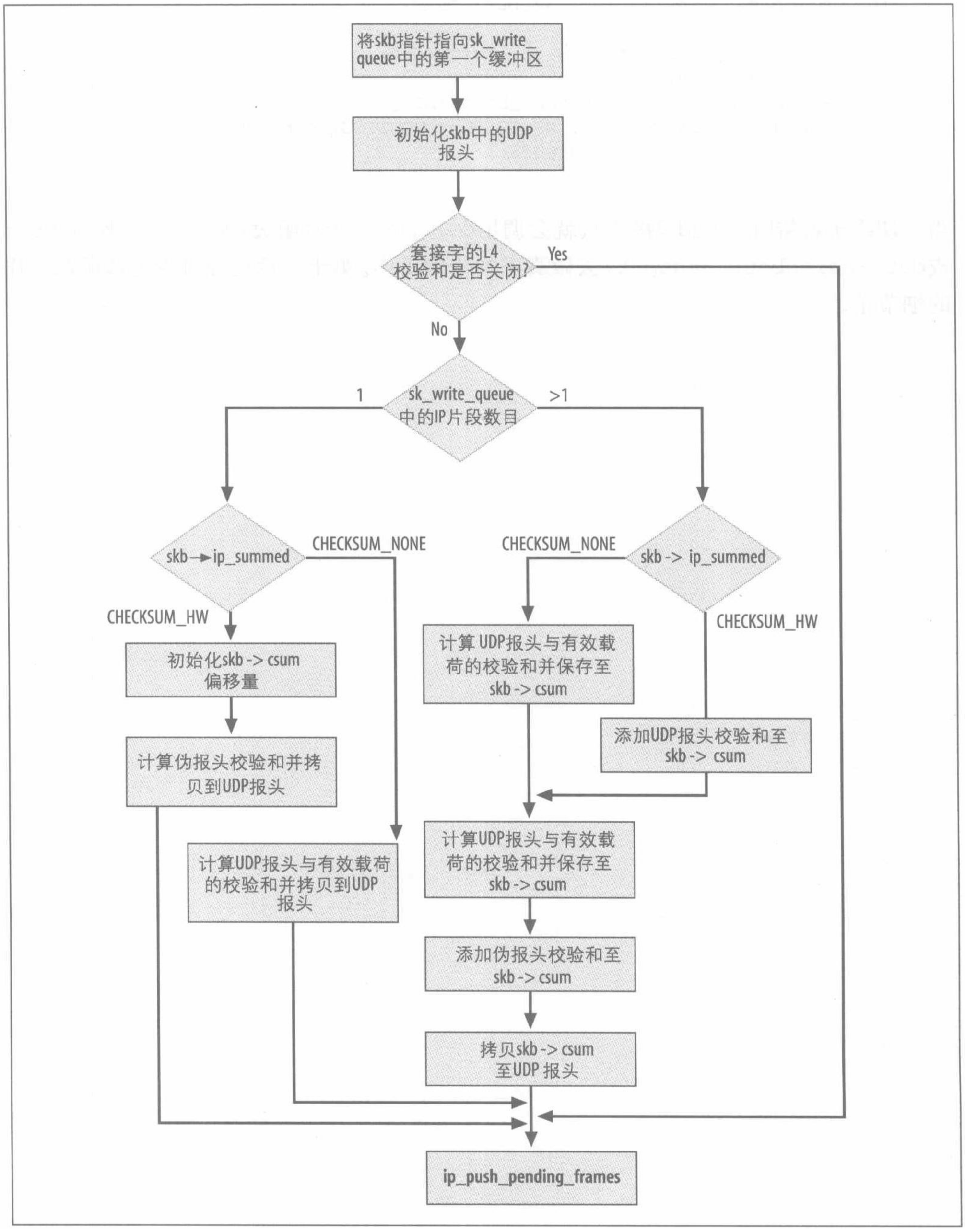


图21-13：udp_push_pending_frames函数

```
int ip_finish_output(struct sk_buff *skb)
{
    struct net_device *dev = skb->dst->dev;
    skb->dev = dev;
    skb->protocol = _ _constant_htons(ETH_P_IP);
    return NF_HOOK(PF_INET, NF_IP_POST_ROUTING, skb, NULL, dev,
                ip_finish_output2);
}
```

当一切终于就绪时(包括L2报头),就会调用dev_queue_xmit函数(通过hh->hh_output或dst->neighbour->output)去做真正的传输工作。第十一章已经讨论过该函数工作的细节了。

第二十二章

因特网协议第4版（IPv4）：处理分段

分段工作和重组工作是复杂的任务，因为在分段和重组封包时，主机接收的IP层输入数据变化多端。从前几章讨论IPv4时所提到的一些函数可知分段工作有很多事情要做。本章说明ip_fragment函数（定义在*net/ipv4/ip_output.c*中）；到该函数中，所有的努力会到达顶峰，并产生一些独立的封包准备传输。本章也会说明相对应的ip_defrag函数（定义在*net/ipv4/ip_fragment.c*中）；这里，进来的片段会重新组合成单一封包，再通过ip_local_deliver传给L4层。同时，每一节也会提到一些辅助函数。

除了IPv4外，这两个函数也可以由其他子系统使用。例如，当Netfilter被迫把一个IP封包重组（并重新分段）以便于存取L3层以上的报头字段时，就会使用这两个函数。这对转发封包而言几乎是不可避免的，而且在第二十一章“ip_push_pending_frames函数”一节也曾做了讨论。

IP层如何认出一个封包是另一个较大封包的一个片段？根据第十七章所述，我们需要IP报头的Offset（偏移量）和MF字段才能分辨。如果封包没被分段，Offset=0且MF=0。相反的，如果我们手上有一个片段，则满足下列条件：

- 第一个片段有Offset=0且MF=1。
- 界于第一个片段和最后的片段之间的所有片段，这两个字段都为非零。
- 最后一个片段有MF=0且Offset为非零。

稍早曾说过，ip_local_deliver是发生重组的地方之一。以下是来自该函数的代码片段，说明如何根据刚才所列的条件而分辨出片段并传给ip_defrag：

```
if (skb->nh.iph->frag_off & htons(IP_MF|IP_OFFSET)) {
    skb = ip_defrag(skb);
    if (!skb)
```

```
                return 0;
        }
```

在分段程序中也可找到类似的逻辑，来为片段正确贴上标签。

分段/重组子系统是由ipfrag_init（由inet_init在开机期间调用）初始化的。这个初始化函数没做什么；主要是启动一个定时器，并把一个变量的初值设成随机数值。这两项任务是必需的，为的是处理一项新增的最优化工作（保护内核免受可能的DoS攻击）；更多细节参见“hash表重新组织”一节。

IP分段

如第十八章图18-1所示，无论是本地产生的封包还是转发封包都会调用dst_output函数，所以，在dst_output以下区域的ip_fragment函数也可在这两种情况下执行。因此，ip_fragment的输入数据可以是：

- 完整的已转发封包。
- 原来的主机或者路上的路由器已分段的已转发封包。
- 由本地函数（已启动分段流程，但尚未把作为封包传输所需的报头加进去）所建的缓冲区（如前章所述）。

特别地是，ip_fragment必须能够处理下列两种情况：

必须切成小块的大数据块

要切割大缓冲区，需要分配新缓冲区，并在大缓冲区和小缓冲区间做内存拷贝。当然，这样会影响性能。

不需要再分段的数据片段链表或数组

如果分配的缓冲区有空间可以新增较低层的L3和L2报头，则ip_fragment处理这些缓冲区时就不需要做内存拷贝。IP层所需做的就是为每个片段加一个IP报头，并处理校验和。

前几版的内核通常在IP层整个地处理IP分段。用于传输封包的IP函数可以接收的有效载荷尺寸为0～64 KB，因此，当封包尺寸大于PMTU时，就必须把有效载荷分割成多个IP片段。我们已经在第十八章“封包的分段/重组”一节讨论过这一内容。

较新版内核所采取的做法是让L4协议事先协助分段任务：L4协议不再把一个会被分段的缓冲区传给IP层，而是传递一组和PMTU相配的缓冲区。如此一来，IP层所处理的IP分段工作就只是为每个已形成的数据片段建立IP报头而已。这并不意味着L4协议实

现了IP分段工作，只意味着因为L4协议也懂IP分段，所以L4协议会试着合作，让IP层的日子轻松一点。L4协议不会接触IP报头。

第二十一章讨论ip_append_data/ip_append_page函数之前，通常认为IP分段比IP重组简单。现在，这两个流程都一样复杂。

当前，分段可以用两种方式进行：所谓的快速法（或效率法）以及慢速法（或旧式法）。这两种方式都是由ip_fragment处理。在了解这两种方式的差异前，我们先回顾一下对IP封包分段所需做的主要任务：

1. 把L3有效载荷分割成一些较小段的数据，使它与传送此封包的路径所用的MTU（PMTU）相匹配。稍后就会看到，这项任务可能会也可能不会牵涉内存拷贝。如果IP有效载荷的尺寸并非刚刚好是片段尺寸的倍数，则最后一个片段会小于其他片段。此外，因为IP报头的“片段偏移量”字段是以8字节为单位，因此，该值会以8字节边界对齐。每个片段都是这个大小（最后一个可能例外）。参见第十八章图18-10所示。
2. 为每个片段的IP报头做初始化（考虑并非所有选项都必须在所有片段内重复）。第十八章“IP选项”一节介绍的ip_options_fragment就是在做这件事。
3. 计算IP校验和。每个片段都有一个不同的IP报头，所以每次都得重新计算校验和。
4. 向Netfilter（Linux过滤系统）请求完成传输的权限。
5. 更新所有内核和SNMP必要的统计数据（诸如IPSTATS_MIB_FRAGCREATES、IPSTATS_MIB_FRAGOKS以及IPSTATS_MIB_FRAGFAILS）。

2.4版以前的内核有个名为ip_build_xmit_slow的函数，会以倒序方式为本地产生的封包建立并传输IP片段（从最后一个到第一个）。这种做法有一些优点：

- 最后一个片段是唯一可以通知接收者原有未分段封包尺寸的片段。尽可能早地知道这件事，有助于重组器对其内存妥善运用。
- 重组器有可能更快建立起一个封包。如“IP重组”一节所述，片段会以偏移量递增次序加入一个链表（ipq）。如果每个片段都比其后片段晚到，则片段就可迅速加入链表头部。

虽然当接收者是Linux机器时这种优化工作行得通，但是，如果接收者使用一些其他采用不同假设的操作系统时（注1），可能就无效，甚至还会成为缺陷。因此，从2.4版开始，Linux就以顺向次序传输片段了。

注1：例如，Cisco Systems的PIX防火墙有个选项可让管理员阻止IP片段通过，除非所接收IP片段的次序是从第一个到最后一个。

牵涉IP分段的函数

前一章讨论在IP层传输数据的函数时，提到了ip_append_data/ip_append_page这组函数，它们为分段做了很多基础工作。本节接下来的焦点是ip_fragment，而ip_fragment会把等待传输的缓冲区转变成实际的封包。

以下是分段程序所用的两个辅助函数：

ip_dont_fragment

决定IP封包是否可以分段（根据路径MTU发现功能的配置）；参见第十八章“路径MTU发现”一节。

ip_options_fragment

修改第一个片段的IP报头，使其可以被后续片段循环利用。参见第十九章“IP选项”一节。

ip_dont_fragment和ip_options_fragment分别定义在*include/net/ip.h*和*net/ipv4/ip_options.c*中。

ip_fragment函数

前一节曾提过，ip_fragment可以处理两种分段方式。我们先看公共部分做些什么。在后续两节中，我们会分别说明这两种情况。

```
int ip_fragment(struct sk_buff *skb, int (*output)(struct sk_buff*))
```

以下是此函数的输入参数的意义：

skb

包含要被分段的IP封包的缓冲区。此封包含有一个已初始化的IP报头，而这个IP报头将会被调整，用于拷贝到所有片段内。参见第二十一章图21-12(b)中有关skb概况的范例。

output

用于传输片段的函数。在第十八章图18-1中，可以看见ip_fragment被调用的一些地点。可以查看一下这些地点，以了解哪些函数被用作output（例如，ip_output使用ip_finish_output）。

ip_fragment一开始会对一些稍后会用到的重要变量做初始化：从设备和IP报头结构（通过skb输入参数取得）取出这些值。出口设备dev和PMTU mtu是从用于传输封包的路径项目（rt）取出的。第三十六章就会知道，该数据结构中还有其他一些参数。

如果输入的IP封包因为来源地设有DF标志而无法被分段，则ip_fragment会传回一条ICMP封包给来源地告知此问题，然后丢弃该封包。if条件式里的local_df标志主要是由Virtual Server程序设定（当其不想让方才所述的情况产生一条ICMP信息时）。

```
dev = rt->u.dst.dev;
    iph = skb->nh.iph;

    if (unlikely((iph->frag_off & htons(IP_DF)) && !skb->local_df)) {
        icmp_send(skb, ICMP_DEST_UNREACH, ICMP_FRAG_NEEDED,
                  htonl(dst_pmtu(&rt->u.dst)));
        kfree_skb(skb);
        return -EMSGSIZE;
    }
```

```
 hlen = iph->ihl * 4;
mtu = dst_mtu(&rt->u.dst) - hlen;
```

当ip_fragment接收了一个其数据已被分段的sk_buff时，就会使用快速分段。例如，对于使用ip_append_data和ip_push_pending_frames函数的L4协议在本地所产生的封包，就有可能发生这种情况。对于使用ip_queue_xmit函数的L4协议所产生的封包，也有可能发生这种情况，因为L4协议会自行处理片段的建立，参见第二十一章。

慢速法则用在所有其他情况，其中一些如下：

- 被转发的封包。
- 抵达dst_output之前，尚未被分段的本地产生的流量。
- 由于对缓冲区所做的健康检查使得快速分段法被关闭的所有情况（参见ip_fragment的开头部分）。

即使输入ip_fragment的缓冲区已是分段后的大小，还是有可能因为片段的组织的错误而无法使用快速法。错误可能是因为一项坏掉的功能执行了有差错的缓冲区操作造成的，或者是因为IPsec协议所用的转换器所造成的。

就这两种情况而言（慢速和快速），如果任何片段传输失败，ip_fragment会立刻返回错误代码，而后续片段就无法传输了。当这一情况发生时，目的主机只会接收到IP片段子集，因此无法把所有IP片段重新组合起来。

慢速分段

与快速分段在ip_append_page/ip_append_data的协助下所做的不同，慢速分段不需要记录任何状态信息（诸如片段链表等）。慢速分段的流程仅包括把IP封包分割成一

些片段，而片段尺寸是根据外出接口的MTU而定，或者如果开启路径MTU发现功能，就由路径相关的MTU决定。

进入循环前，该函数必须对一些局部变量做初始化。

ptr就是要被分段的封包里的偏移量；它的值会随着分段工作的进行而移动。left的初值设为IP封包的长度。计算left时，ip_fragment函数会减去hlen（L2报头长度），因为这个元素不是IP有效载荷的一部分，而且该函数必须为其保留空间，因为它将会被拷贝到每个片段。

IP报头把片段偏移量以及DF和MF标志都放在一个16位字段内。下列代码中的公式会从中抽取出offset字段。

如果该封包中当前片段之后还有更多其他数据，则局部变量not_last_frag（顾名思义指不是最后的分段）就为真。这一数据位很重要，因为封包中最后一个片段会指出该封包的尺寸，而这是有效分配内存的宝贵信息；该函数稍后会使用此项信息。然而，对封包（该封包指原有的封包；例如，如果一个封包断成两段，而第二段稍后又分段，则第二段里的所有分段都会设有not_last_frag变量）中第一个片段而言，不会设定not_last_frag变量。

```
left = skb->len - hlen;
ptr = raw + hlen;

offset = (ntohs(iph->frag_off) & IP_OFFSET) << 3;
not_last_frag = iph->frag_off & htons(IP_MF);
```

接着，ip_fragment会进入循环，为每个片段建立一个新缓冲区（skb2）。输入参数skb包含原有的IP封包。

```
while(left > 0) {
  len = left;
```

对每个片段而言，其长度设成稍早由PMTU字段所定义的MTU值。片段的尺寸也要对齐8字节边界，这是IP RFC强加的条件。下面代码中的条件不满足的唯一情况是当我们在传输最后一片段之时，或者当不需要分段之时。但是，第二种情况应该绝不会发生，因为如果不需要分段，此函数一开始就不会执行。

```
if (len > mtu)
    len = mtu;

if (len < left) {
    len &= ~7;
}
```

保存一个片段的缓冲区的尺寸是下列各项之和：

- IP有效载荷的尺寸
- IP报头的尺寸
- L2报头的尺寸

最后一项值正是在while循环之前初始化的，而且是从路由表缓存中抽取出的。IP层可以从路由表知道用于传输片段的L2设备是什么。ip_fragment函数可以从相关的net_device数据结构取出和该设备协议相关的报头尺寸。此值会由LL_RESERVED_SPACE[_EXTRA]宏对齐16字节边界，而且会存储在局部变量ll_rs（Link Layer Reserved Space，链路层保留空间）。这里的对齐和刚才对有效载荷所做的对齐无关。当内核被编译成支持L2防火墙机制时（即CONFIG_BRIDGE_NETFILTER内核选项），ll_rs和mtu也会相应地更新，以容纳一个可能的802.1Q报头。

```
if ((skb2 = alloc_skb(len+hlen+ll_rs,
                      GFP_ATOMIC)) == NULL) {
    NETDEBUG(printk(KERN_INFO "IP: frag: no memory for new fragment!\n"));
    err = -ENOMEM;
    goto fail;
}
```

现在，该函数必须把一些sk_buff结构（原有IP封包，正被拷贝）字段的值拷贝到新分配的缓冲区skb2。有些字段是在这里拷贝，但其他的字段则由ip_copy_metadata处理，也会根据内核是否会包含一些特定功能（诸如Traffic Control和Netfilter）至内核而拷贝一些字段。指向L3（nh.raw）和L4（n.raw）报头的指针也会被初始化。

```
ip_copy_metadata(sk2, skb);
skb_reserve(skb2, ll_rs);
skb_put(skb2, len + hlen);
skb2->nh.raw = skb2->data;
skb2->h.raw = skb2->data + hlen;
```

新分配的缓冲区与试着做传输的套接字（如果有的话）关联。例如，当第十八章图18-1中左边的函数请求传输时，就会如此。

```
if (skb->sk)
  skb_set_owner_w(skb2, skb->sk);
```

现在，该把一些真实数据填入新缓冲区skb2了（到目前为止，该函数只处理一些skb_buff结构的管理字段而已）。这要分成两部分来做：

- 利用简单memcpy拷贝IP报头。
- 然后，将原有封包里的一段有效载荷拷贝至片段内。

第二项任务无法使用简单memcpy，因为skb中数据的存储方式有很多种，使用片段链表或者内存页面扩展（参见第二十一章）。当封包内的所有数据在skb->data所指内存区域时（参见第二十一章图21-2所示），或者当数据抵达ip_fragment前早已分段，但是因先前所述其中一项健康检查的缘故而无法使用快速法时，就可以使用慢速法。

```
        memcpy(skb2->nh.raw, skb->data, hlen);

        if (skb_copy_bits(skb, ptr, skb2->h.raw, len))
            BUG();

        left -= len;
        iph = skb2->nh.iph;
    iph->frag_off = htons((offset >> 3));
```

从IP选项的角度看，第一个片段（其中offset为0）很特殊，因为这个片段是唯一包含来自原有IP封包的选项的完整拷贝的片段。并非所有选项都必须拷贝到所有片段；只有第一个片段将包含所有选项。

```
        if (offset == 0)
            ip_options_fragment(skb);
```

ip_options_fragment（第十九章曾探讨过）会清掉与原有IP封包相关的ip_opt结构的内容，使得第一个片段之后的片段不会包含不需要的选项。因此，只有在处理第一个片段期间（offset=0的片段），才会调用ip_options_fragment。

如果下列条件之一成立，就会设定MF（More Fragments）标志：

- 被分段的封包本身不是片段，而此循环内所创建的此片段也不是最后片段（left>0）。
- 被分段的封包本身是片段，但不是最后一个片段，因此，其所有片段都必须设有MF（not_last_frag=1）。

```
        if (left > 0 || not_last_frag)
            iph->frag_off |= htons(IP_MF);
```

下列两条语句更新这两个变量。这两个变量很容易搞混。持有offset是因为正被分段的封包可能是另一个较大封包的一个片段；如果是的话，offset代表的就是当前片段在原有封包内的偏移量（否则就是0）。ptr是我们正在分段的封包内的偏移量，而且会随着循环进程而改变。这两个变量在两种情况下的值会相同：我们分段的封包本身不是片段，或者该片段就是第一片段。

```
        ptr += len;
    offset += len;
```

最后，慢速法必须更新报头长度（把选项的尺寸考虑进来），用ip_send_check计算校

验和，然后使用output函数（作为输入参数）传输该片段。IPv4所用的output函数就是ip_finish_output（参见第十八章图18-1所示）。

```
iph->tot_len = htons(len + hlen);
ip_send_check(iph);

err = output(skb2);
```

快速分段

当ip_fragment看见输入的skb缓冲区的frag_list指针不为NULL时，就会尝试快速法。然而，如本章稍早所述，它必须确保片段适用快速法。以下是协议所规定的健康检查：

- 每个片段的尺寸不应该超过PMTU。
- 只有最后一个片段的L3有效载荷的尺寸可以不是8字节的倍数。
- 每个片段在头部必须有足够空间来容纳稍后L2报头的新增。

此外，还有其他一些缓冲区管理工作要检查：

- 该片段不能被共享，因为这会使得ip_fragment无法对其修改（无法新增IP报头）。使用慢速法时，ip_fragment接收一个共享的缓冲区，这是可以接受的，因为该缓冲区会被拷贝到其他很多新缓冲区，但是对快速法而言就是不可接受的。

```
if (skb_shinfo(skb)->frag_list) {
    struct sk_buff *frag;
    int first_len = skb_pagelen(skb);

    if (first_len - hlen > mtu ||
        ((first_len - hlen) & 7) ||
        (iph->frag_off & htons(IP_MF|IP_OFFSET)) ||
     skb_cloned(skb))
      goto slow_path;

for (frag = skb_shinfo(skb)->frag_list; frag; frag = frag->next) {
    if (frag->len > mtu ||
        ((frag->len & 7) && frag->next) ||
        skb_headroom(frag) < hlen)
        goto slow_path;

    if (skb_shared(frag))
        goto slow_path;
    ...
}
```

第一个片段的IP报头的初始化会在循环外面完成，因为可以稍微对其进行优化。例如，当此函数运行时，它知道至少有两个片段，因此不需要对第一个片段检查frag->next

就可以对iph->frag_off做初始化：作为第一个片段，此片段必须设定了IP_MF标志，而偏移量部分则设为0（iph->frag_off = IP_MF）。其他封包必须在其frag_off里设定IP_MF位，而且不能干扰该值的其余部分（iph->frag_off |= IP_MF）。

我们假设可以使用快速法。程序的剩余部分相当简单，某种程度上和慢速法的程序相当类似。第一个片段传送之后（也就是for区段的第一轮循环之后），IP报头会由ip_options_fragment修改，使得可被后续片段循环利用。如果排除特殊情况，传输片段时所需做的就是：

- 从第一个IP片段中把（修改过的）报头拷贝到当前片段。
- 对那些可能有所不同的IP报头字段做初始化。其中一些是偏移量和IP校验和（由ip_send_check计算）。此外，如果该片段不是最后片段，就设定MF标志。
- 从第一片段中把sk_buff的剩余字段拷贝到当前片段(使用ip_copy_metadata)。这些字段是管理参数；和IP片段的内容无关。
- 用output函数（作为输入参数）传输片段。

万一发生错误，frag_list里所有后续片段的内存都会被释放（下列程序片段中没有显示出来）。注意，在if (frag) {...}区段内的程序会把要在下一轮循环中传输的片段准备好，然后调用output来传输当前片段。

```
        skb->data_len = first_len - skb_headlen(skb);
        skb->len = first_len;
        iph->tot_len = htons(first_len);
        iph->frag_off = htons(IP_MF);
        ip_send_check(iph);

    for (;;) {
        if (frag) {
            frag->ip_summed = CHECKSUM_NONE;
            frag->h.raw = frag->data;
            frag->nh.raw = _  _skb_push(frag, hlen);
          memcpy(frag->nh.raw, iph, hlen);
            iph = frag->nh.iph;
            iph->tot_len = htons(frag->len);

            ip_copy_metadata(frag, skb);
            if (offset == 0)
                ip_options_fragment(frag);
            offset += skb->len - hlen;
            iph->frag_off = htons(offset>>3);
            if (frag->next != NULL)
                iph->frag_off |= htons(IP_MF);
            ip_send_check(iph);
        }
```

```
        err = output(skb);

    if (err || !frag)
            break;

        skb = frag;
        frag = skb->next;
        skb->next = NULL;
    }
```

IP重组

显然，当封包已抵达其最终目的地，而且必须传给上面的网络层时（在Linux中是由ip_local_deliver函数处理的），就需要进行重组。相反的，路由器通常只是让封包通过，而不在乎封包是否为另一个较大封包的片段。但是，路由器偶尔也需要重组：一般而言，每当一台主机必须对整个封包做点处理工作时，就需要重组。路由器的两种这类情况如下：

- IP报头中包含Router Alert选项，强迫路由器去处理该封包，参见ip_call_ra_chain（由ip_forward调用）以及第十八章图18-1所示。
- Netfilter必须查看封包，才能决定该对封包做什么事。以第十八章图18-1为例，Netfilter会强迫进行重组的钩子函数是NF_IP_PRE_ROUTING和NF_IP_LOCAL_OUT。

但是，重组工作的运作方式并非取决于触发重组的环境，所以，我将从高层的角点说明它的实现方式。

IP片段hash表的组织

接收IP片段之后，会将其组织成一张hash表（内含struct ipq元素）。图22-1所示是此数据结构的组织方式以及其用法的范例。

```
#define IPQ_HASHSZ     64
static struct ipq *ipq_hash[IPQ_HASHSZ];
```

每个正被重组的IP封包都会由一个ipq实例表示，其内由一个片段链表组成。图22-1所示是一个ID为1234的IP封包范例，到目前为止只收到两个片段。在该图的底端，可以看见这两个片段位于原有IP封包（长度为1,250个字节）的哪个位置。此图显示出此数据结构内一些最重要字段的角色。

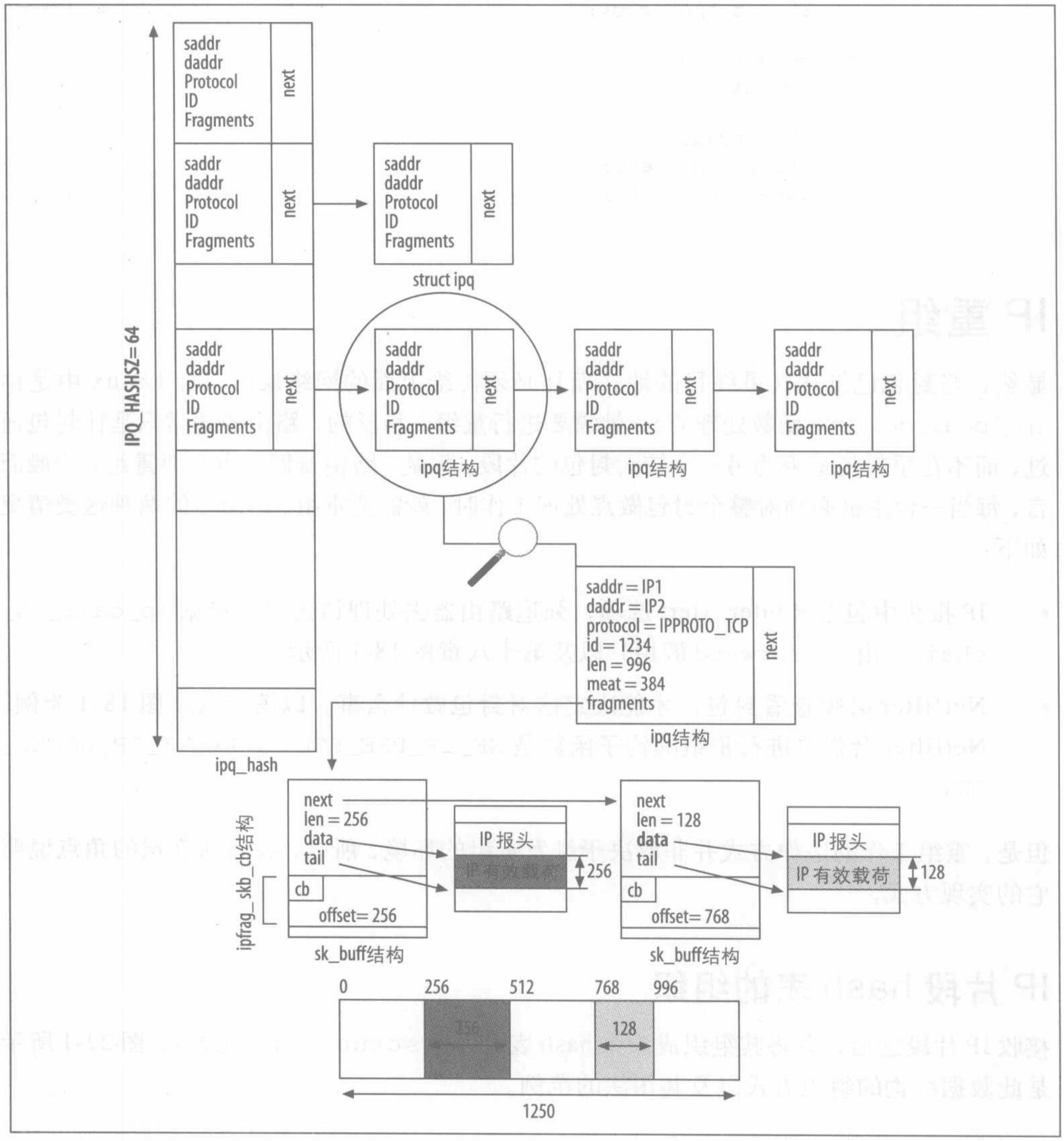


图 22.1：用于存储 IP 片段的结构

在该图底端附近，可以看见每个片段的偏移量是存储在每个 sk_buff 里名为 cb 的字段内。由第二章已知，该字段是一个缓冲区，可以由各种网络分层用以存储私有信息。存储在那个缓冲区内的数据会改变，这取决于该缓冲区是在接收还是传输而定。

在 IP 重组的上下文中，IP 会使用 sk_buff->cb 字段来存储一个 ipfrag_skb_cb 结构，该结构只是内含 inet_skb_parm 的简单包裹结构；inet_skb_parm 结构用于存储 IP

选项和标志（同一个结构通常也由较高层使用）。加入inet_skb_parm的新字段就是该片段位于原有IP封包内的偏移量。该数据结构可以由宏FRAG_CB（定义在*net/ipv4/ip_fragment.c*中）存取。于是，IP层会使用FRAG_CB用以片段重组；此外，为了其他事情，IP层会使用IPCB（定义在*include/net/ip.h*）来存取选项。这些宏以不同名称指向那些数据结构，但是，最后都是指向内存中相同的位置。

ipq_hash表是由ipfrag_lock保护的，而ipfrag_lock可以在共享（只读）或独占（读/写）模式下被取得。不要把这个锁和内嵌在每个ipq元素里的锁搞混了。

重组的重要议题

当你读完这一节的其余部分时，将有助于你把使得重组复杂化的制约条件谨记在心：

- 片段必须存储于内核内存中，直到它完全被网络子系统处理完，但是内存很昂贵。因此，一定要有种方式去限制内存的使用。
- 存储大量信息（只要想到路由器每秒传送好几百万个封包）最有效的结构就是hash表。然而，hash表也会失衡，特别是如果恶意攻击者摸清hash算法后，精打细算试着拉高hash表中特定元素的权重而使得处理速度慢下来。在“hash表重新组织”一节，我们将会看Linux如何让hash算法使用一个额外的随机元素来使得特定输入数据所产生的输出不可预测。
- 网络联机时常使用不可靠的媒介，所以，片段可能会遗失。如果一个封包里的不同片段以不同路径传输时，这一点尤其可能发生。因此，IP层必须为每个封包持有一个定时器，并在某个点上放弃，并把所有接收到的片段抛弃掉。此外，也必须采用校验，把检测到损毁的机会拉到最大。
- 如果源主机在一段时间后没收到某些数据的确认通知信息，而且传输协议实现了流控制，该主机就会重传数据。因此，对单个IP封包而言，目的地可能会接收到好几个重复片段。让这一问题更复杂的是，第二个IP封包所走路径可能和第一个不同，因此，其分段也不同，所以片段间的边界可能也不相吻合。由第十八章已知，当一个IP封包重新传输时，会得到一个新IP ID，这有助于减少发生这种问题的可能性。可惜，如第十八章所说，在高速网络上，IP ID很快就会用尽而环绕，所以不同IP数据段间的片段混杂在一起的问题依然存在。有关IP协议把IP片段和IP报文关联的规则，请参考第十八章“把片段和其所属的IP封包相关联”一节。

于是，这些需求就造成接下来要说明的实现细节了。片段存储在一个会定期改变（在hash函数的输入参数中多一个随机元素）的hash表内（更多细节可参考“hash表重新组织”一节）。每个封包都会关联一个定时器，如果定时器到期了，该封包就会被删除。每个片段都会检查是否损毁，以及是否和先前接收的片段彼此重叠。

涉及重组的函数

如前所述，用于处理重组的主要函数是ip_defrag。此函数每次调用时都会接收一个片段作为输入参数，然后试着将该片段加至正确的封包。当最后一个片段已找到，而该封包已完成时，此函数才会返回成功。下一节会说明其实现细节。此函数也会接收第二个输入参数user，以指出请求重组的原因。参见第二十三章“ipq结构”一节有关user的说明。

图22-1所示是用于存储已接收IP片段的数据结构，由数据结构的hash表所组成，每个完整的封包都有一个hash表，依次指向该封包的那些片段。

以下是ip_defrag所用（直接或间接）的一些辅助函数；都定义在*net/ipv4/ip_fragment.c*中：

ip_evictor

逐一删除不完整封包的ipq结构，从最旧的着手，直到片段所用的内存降到sysctl_ipfrag_low_thresh阈值以下。

为了让ip_evictor正确运作，一个LRU（Last Recently Used，最近最后使用）链表必须不断更新。其做法就是把新ipq结构加到一个全局链表（ipq_lru_list）尾端，然后每次有新片段加至一个ipq结构时，就把该结构移往该链表的尾端。也就是说，最久没被碰过的元素会摆在ipq_lru_list的头部；于是，没希望完成的封包（例如，因为进行传输的主机挂掉了）就会站在队列前端。

ip_find

找出和正在被处理的片段相关的封包（片段链表）。查询是根据IP报头的4个字段：ID、源IP地址、目的IP地址以及L4协议。这样就可以相当确信（但非绝对确信）选中的是正确的封包（参见第十八章“无法解决的重组问题范例：NAT”一节）。查询关键字实际上也包含一个本地参数：user。这一参数用于指出重组的原因（参见第二十三章“ipq结构”一节）。

ip_frag_queue

把指定的片段插入和同一个IP封包相关的片段链表（ipq结构）中。参见图22-1所示以及“ip_frag_queue函数”一节所述。

ip_frag_reasm

一旦所有片段都被接收到之后，就从这些片段构建原有的IP封包。

以下是一些用于处理ipq删除的辅助函数：

ip_frag_destroy

删除传进来的ipq结构以及其相关的所有IP片段，然后更新全局计数器ip_frag_mem（参见“ip_defrag函数”一节）。该函数是通过其包裹函数ipq_put调用的，而非被直接调用。

ipq_put

递减传进来的ipq结构的引用计数值，如果没有其他用户对该结构进行引用，就用ip_frag_destroy将该结构及其片段删除掉。

```
static _ _inline_ _ void ipq_put(struct ipq *ipq, int *work)
{
        if (atomic_dec_and_test(&ipq->refcnt))
                ip_frag_destroy(ipq, work);
}
```

当输入参数work不是NULL指针时，在ipq_put返回时，work的初值会设成由ip_frag_destroy所释放的内存数量。例如，这一信息对ip_evictor的调用（以释放特定量的内存）有所帮助，因此必须知道每次对ipq_put的调用需要释放多少内存。

ipq_kill

把一个ipq结构标记为可删除，因为有些片段没有准时到达。更多细节参见“垃圾收集”一节。

接下来两节中，我们将进一步说明ip_defrag和ip_frag_queue。首先，来看看如何建立一个新片段链表（ipq实例）。

新ipq实例的初始化

ip_defrag的首项任务就是搜寻封包，也就是其于输入参数所接收的片段应该加入的那个封包。为了找出封包，此函数会调用ip_find。如果该片段碰巧是封包的第一个片段（以其抵达时间算，不见得是其在封包内的位置），ip_find就会失败。这一情况下，会使用ip_frag_create函数建立一个新的ipq实例。无论是找到一个结构还是新建一个结构，ip_find都会返回指向该结构的指针。ip_defrag使用这个指针把新片段插入到正确封包的ipq结构。ip_find会失败（返回NULL）的唯一情况是在试着建立一个新ipq元素时发生错误。

尽管新片段的插入由ip_defrag处理，但是新ipq实例的插入是由ip_frag_create通过ip_frag_intern处理的。除了在此新结构中对一群参数做初始化外，此函数也会启动一个垃圾收集定时器，如果相关的重组无法在特定时间量内完成时，就会清理新ipq结构（及其所有片段）。该超时时限默认为30秒，但是可以通过*/proc*配置（参见

第二十三章“通过/proc档案系统做调整”一节)。做垃圾收集的函数ip_expire也会产生ICMP消息，以通知源主机重组尝试失败。

ip_defrag 函数

ip_defrag函数实际上相当简单,因为所有的复杂度都存在于其内部所使用的四个函数中：ip_find、ip_frag_queue、ip_frag_reasm以及ip_evictor。

```
struct sk_buff *ip_defrag(struct sk_buff *skb, u32 user)
```

ip_defrag于输入参数所接收的片段skb包含稍早讨论过的所有信息,足以指认出其所属的ipq实例（如果已建立的话)。

该函数会先检查IP片段所用掉的内存，然后，如果已经抵达可配置的阈值时，就以ip_evictor触发垃圾收集。参见“垃圾收集”一节。

如果这是一个新IP封包的第一个片段，ip_find会创建一个新ipq结构；否则，仅将其找到的结构返回。如果是新建结构,则该结构会通过ip_frag_queue加入ipq_hash。

```
if ((qp = ip_find(iph)) != NULL) {
    struct sk_buff *ret = NULL;
```

最后，该片段会排入队列中。ip_frag_queue是相当复杂的函数，我们会在下一节详细加以说明。片段的链表qp会由一个锁保护，以确保无法同时对该链表做不兼容的存取：

```
spin_lock(&qp->lock);
ip_frag_queue(qp, skb);
```

如果第一个和最后一个片段都已接收,而且片段的总尺寸等于原有IP封包的尺寸,那么就该将这些片段联结起来获得原有封包,并将封包传给较高层。ip_frag_reasm会停止和qp元素相关的定时器，把那些片段连接起来，更新一些全局变量——诸如代表片段所用内存的变量（ip_frag_mem）——然后处理L4硬件校验和（参见“L4校验和”一节)。我们不会详细说明此函数，因为这个函数大部分都是寻常的低级指令。

```
        if (qp->last_in == (FIRST_IN|LAST_IN) &&
            qp->meat == qp->len)
            ret = ip_frag_reasm(qp, dev);

        spin_unlock(&qp->lock);
        ipq_put(qp, NULL);
        return ret;
    }

    IP_INC_STATS_BH(IPSTATS_MIB_REASMFAILS);
    kfree_skb(skb);
    return NULL;
}
```

ip_frag_queue 函数

把一个新片段加入一个 ipq 结构（链表内的片段都属于同一个 IP 封包）的任务是复杂的，因为用于存储片段的数据结构并非普通的数组，其使用 offset 字段拷贝片段。这种做法有个主要问题：因为原有IP封包的尺寸直到接收最后一个片段前都是未知的，这会迫使 IP 层去分配一个缓冲区，使它的尺寸等于最大 IP 封包尺寸。可以预料，这会浪费很多内存。此外，虽然实现简单，这类做法不会运作得很好，因为只要做 DoS 攻击，很快就会让路由器不能正常工作。

使用链表处理片段，可以对所用的内存进行优化，但是会使得片段的处理稍微复杂一点。我们总结 ip_frag_queue 所做的主要任务如下：

- 弄清输入片段处于原有封包何处（根据其偏移量和长度）。
- 根据本章开头所述的考虑要点，求出其是否为封包的最后片段，如果是的话，就从中取出 IP 封包长度。
- 把该片段插入到链表中，而该链表内的片段都关联同一个 IP 封包，此外还要处理可能的重叠问题（如本章稍早所述，如果封包被主机认为已遗失而由主机重新传输时，那些片段就可能重叠；可能是来自于不同路径，而且使用不同的 PMTU）。
- 更新ipq结构中由垃圾收集任务所用的那些字段（也就是时间戳以及所用的内存）。
- 必要时，让在硬件中计算的 L4 校验和失效（例如，当一个片段必须被 ip_frag_queue 截断时）。

这是函数原型：

```
static void ip_frag_queue(struct ipq *qp, struct sk_buff *skb)
```

其中qp是该片段所属的IP封包（由调用者利用ip_find函数找到），而skb是新片段。

该函数会先从IP报头把数据抽出，然后作一些一般性检查，以确保该片段有效。第一项一般性检查是确保此函数不是在IP封包已完全接收时而被错误地调用。COMPLETE标志通常是在所有片段都接收后设定，但是在其他不寻常的情况下也会被设定 —— 例如，当 ipq_kill 把一个 ipq 元素标记为死掉时。

```
if (qp->last_in & COMPLETE)
    goto err;
```

偏移量存储在IP报头中16位“偏移量”字段里的低13位内。三个最高位中的两个位是由两个标志使用：DF 和 MF（注 2）。而另一位没用到。

注 2：参见文件 *include/net/io.h* 以及第十八章中的图 18-2。

因为偏移量是以8字节为单位，此字段值在使用前一定要乘上8。报头长度ihl是以4字节为单位，因此使用前必须乘上4。IP_OFFSET只是个屏蔽值，用于从16位字段中取出低13位。

```
offset = ntohs(skb->nh.iph->frag_off);
flags = offset & ~IP_OFFSET;
offset &= IP_OFFSET;
offset <<= 3;
ihl = skb->nh.iph->ihl * 4;
```

由于IP片段会携带其偏移量和其长度，我们可轻易算出片段于原有IP封包内的何处结束。skb->len - ihl是IP有效载荷的尺寸，而且由于该有效载荷处在原有IP封包内offset之处，因此，其和就是此片段在原有IP封包内终止的偏移量。

```
end = offset + skb->len - ihl;
```

如果未设定MF标志，意味着该片段是最后片段；因此，我们可以抽取出原有IP封包的总长度，将此长度存储在qp->len中，然后设定标志LAST_IN。如果从这一最后片段所推得的原有封包尺寸（end）和我们稍早定义的值（如果有的话）不吻合，就表示这一片段或者先前片段之一损毁了，因此，该片段会被丢弃。

```
if ((flags & IP_MF) == 0) {
    if (end < qp->len ||
        ((qp->last_in & LAST_IN) && end != qp->len))
        goto err;
    qp->last_in |= LAST_IN;
    qp->len = end;
} else {
```

每个片段（最后一个除外）都必须是8字节的倍数。因此，如果当前片段不是最后一个（MF未设定），而且其尺寸不是8字节的倍数，则内核会将它截断，以使其尺寸成为8字节的倍数（这里是希望另一个片段会带着被截断的信息抵达，使得重建的封包得以正确）。由于此运算会改变L4有效载荷（通过截断数据），此函数也必须使校验和失效，以防校验和已经计算（注3）。进行接收的L4层必须进行重算。

```
if (end&7) {
    end &= ~7;
      if (skb->ip_summed != CHECKSUM_UNNECESSARY)
          skb->ip_summed = CHECKSUM_NONE;
}
```

如果该片段的结束点（offset+len）大于qp->len的当前值，qp->len就会被更新。注意，只有当最后一个片段已被接收时，qp->len才会代表原有已重组封包的长度。因

注3： 参见ip_rcv以及ip_rcv_finish中的注释以了解类似情况。

此，当最后一个片段已被接收，而有个片段的结束点超过qp->len时，这意味着某处出错了，因此该片段会被丢弃（注4）。

```
        if (end > qp->len) {
            if (qp->last_in & LAST_IN)
                goto err;
            qp->len = end;
        }
    }
```

根据IP协议的定义，IP报头不能被分段。也就是说，如果一个封包被分段了，有效载荷一定不为空。因此，没有有效载荷的片段（也就是片段的起始点就是其结束点）就没有意义；因此，如果有片段符合此情况，就会被视为已损毁。

```
    if (end == offset)
        goto err;
```

现在，该函数删除IP报头（通过把skb->data偏移量移往IP有效载荷，并更新skb->len；其做法是调用pskb_pull）。然后，该函数会调用pskb_trim，把缓冲区数据部分的长度设成IP有效载荷的长度（end-offset）。注意，第二项运算实际上只在下列两种情况下才需要：

- 缓冲区依然含有一些L2填充。这种情况绝不会发生，因为如果还有L2填充存在，在ip_rcv的程序路径中早就被删除了。
- IP片段的尺寸不是8字节的倍数。这种情况下，该函数会把长度缩减成8字节的倍数，因此，缓冲区的尾端会留下一些垃圾。

```
    if (pskb_pull(skb, ihl) == NULL)
        goto err;
    if (pskb_trim(skb, end-offset))
        goto err;
```

包含在qp输入参数里的片段链表（参见图22-1所示）会保持排序结果，使得最低偏移量的片段放在链表头部。因此，该函数现在必须找出应该把新片段加至链表的何处。

```
    prev = NULL;
    for(next = qp->fragments; next != NULL; next = next->next) {
        if (FRAG_CB(next)->offset >= offset)
            break;
        prev = next;
    }
```

注4：　理论上，已损毁的封包是那个MF=0而且先前设定qp->len的封包，而非我们正在丢弃的封包。

处理重迭

现在，该处理先前接收的数据帧可能的重迭了。这是通过两个步骤进行的：首先，该函数处理那些起始偏移量较小的片段间的冲突，然后处理其他起始偏移量较大的片段间的冲突。next 和 prev 变量（指向前节所述 qp 链表内）管理旧片段链表。

如果新片段不必放到该链表头部（prev!=NULL），也就是我们至少已经接收到一个偏移量较小的片段，那么，进行插入时就得把公共部分（如果有的话）从重迭片段之一中删除。

为此，此函数只需求出重迭部分的尺寸，然后从新片段头部删除该尺寸大小的区块。注意，下列程序中，重迭的存在是由 i 标记（为正数）的：

```
if (prev) {
    int i = (FRAG_CB(prev)->offset + prev->len) - offset;

    if (i > 0) {
        offset += i;
        if (end <= offset)
            goto err;
        if (!pskb_pull(skb, i))
            goto err;
        if (skb->ip_summed != CHECKSUM_UNNECESSARY)
            skb->ip_summed = CHECKSUM_NONE;
    }
}
```

在和链表中前一个片段确实有重迭发生时，此函数会使用pskb_pull从新片段中把余赘部分删除，进而更新稍早从报头中取出的 offset 字段，然后让硬件中所计算的 L4 校验和失效。如果把偏移量往前移意味着片段的起始点比结束点还要高，就表示新片段已完全包含在先前已收到的那些片段中，所以，此函数只要返回即可。

处理过前面的片段后，此函数现在可以处理后续片段（偏移量较高的）可能的重迭了。有两种情况：

- 一个或多个后续片段完全包含在新片段内。
- 一个后续片段部分地和新片段重迭。

这两种情况都在图 22-2 中加以展示，其中 P 指新片段，而 F 指旧片段。P1 只和 F2（完全包含在 P1 之中）重迭，而 P2 则和 F3 及 F4（完全包含 P2）重迭。

此时，next 所指向的就是其 offset 值最先大于新片段偏移量的片段。此函数会逐一遍历片段，直到发现重迭；其做法就是比较新片段的结束点以及链表中那些片段的结束点（记住，已存在于链表里的片段是以 offset 递增次序排序的）。

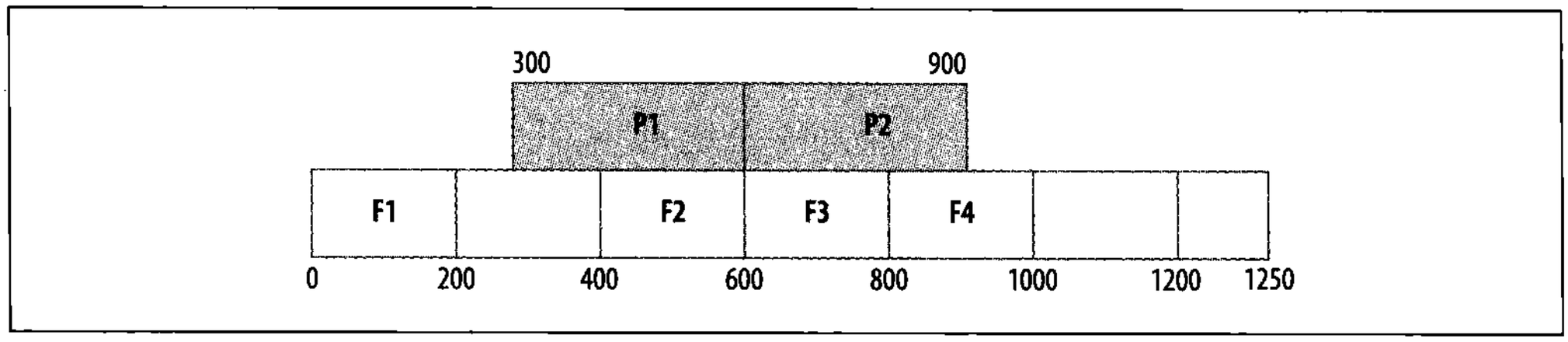


图22-2：片段间有一个或多个重迭的范例

```
while (next && FRAG_CB(next)->offset < end) {
    int i = end - FRAG_CB(next)->offset;
```

如果重迭部分的尺寸比新片段尺寸小，这表示该函数已抵达该链表的最后一个重迭片段，因此只需从已存在于链表内的片段中删除重迭部分。新片段则将于稍后被加入。由于此函数会把已存在于链表内的片段的一部分截断，被截断的片段的qp->meat和offset字段都必须被更新，而硬件校验和也必须失效。注意，当是和前一个片段重迭时，此函数会把新片段的数据删除；但是，当是和后一个片段重迭时，此函数会做相反的事情。

```
if (i < next->len) {
    if (!pskb_pull(next, i))
        goto err;
    FRAG_CB(next)->offset += i;
    qp->meat -= i;
    if (next->ip_summed != CHECKSUM_UNNECESSARY)
        next->ip_summed = CHECKSUM_NONE;
    break;
} else {
```

相反的，如果片段完全包含在新片段中，则此函数会将它从链表中删除（再次更新qp->meat）（注5）。

如果被删除的片段是链表的头部，则头部指针必须被更新：

```
            struct sk_buff *free_it = next;
            next = next->next;

            if (prev)
                prev->next = next;
            else
                qp->fragments = next;
            qp->meat -= free_it->len;
            frag_kfree_skb(free_it, NULL);
        }
    }
```

注5：frag_kfree_skb也会同时更新ip_frag_mem。

最后，解决了所有可能重迭后，此函数就可以把新片段插入到链表中，并更新qp结构的一些参数，诸如meat、stamp以及last_in。skb->truesize以及片段（ip_frag_mem）当前所用的内存也会被更新。qp也会被移到ipq_lru_list链表尾端。

```
qp->stamp = skb->stamp;
qp->meat += skb->len;
atomic_add(skb->truesize, &ip_frag_mem);

if (offset == 0)
    qp->last_in |= FIRST_IN;
```

L4校验和

如果入口设备支持L4硬件校验和计算，入口IP片段的L4校验和可能已经算好了。当片段由ip_frag_reasm重新组合时，会用csum_add把个别片段的校验和结合起来，然后把结果存储在已重新组合缓冲区。然而，当下列条件之一满足时，已重新组合缓冲区的硬件校验和会失效（也就是skb->ip_summed被设为CHECKSUM_NONE）：

- 一个片段（最后一个片段例外）在ip_defrag里已被截断，因为它的尺寸不是8字节的倍数。
- 一个片段和至少一个先前已接收的片段重迭。因为处理重迭的方式就是删除余赘部分，因此，校验和（也包含余赘部分）必须失效。

垃圾收集

内核为IP片段实现两种垃圾收集：

- 系统内存使用限度。
- 重组定时器。

为了防止IP重组子系统滥用内存，对该内存的使用有强加限度，而且存储在sysctl_ ipfrag_high_thresh变量中，该变量的值可以在运行期间通过*/proc*文件系统而修改。全局变量ip_frag_mem代表当前由片段使用的内存。每次一个新片段加入ipq_hashhash表结构或从中删除时，其值就会被更新。当抵达系统限度时，就会调用ip_evictor来释放一些内存。

```
if (atomic_read(&ip_frag_mem) > sysctl_ipfrag_high_thresh)
    ip_evictor();
```

对内存限度的检查是通过ip_defrag实现的（参见“ip_defrag函数”一节）。

当一个新IP封包的第一个片段加入ipq_hashhash表时（即新ipq实例已建立时），内核会启动一个重组定时器。此定时器是用于丢弃不完整封包的所有片段，以避免不完整封包停留在ipq_hash里太久（参见第二十三章“通过/proc文件系统做调整”一节有关sysctl_ipfrag_time的讨论）。如果一个片段遗失或延迟得够久，而定时器到期了，则其处理函数ip_expire会被调用以进行清理工作；而其清理工作由下列运算组成：

1. 从ipq_hashhash表以及lru_list链表解除与ipq结构的链接。
2. 如果ipq包含IP封包的第一个片段，就传回一个ICMP TIME EXCEEDED消息给源主机。本地主机必须接收了第一个片段，才可以传输ICMP消息，因为该消息必须在它的有效载荷中包含原有IP封包的一部分，而且只有第一个片段才有包含原有的IP报头（未分段封包的所有选项）以及所有或部分L4报头（参见第二十一章图21-4所示）。只有当接收最后一个片段的设备依然开启并运行，才会传送ICMP消息，因为该设备将极有可能用于传输该ICMP消息。
3. 由于失败的重组事件而更新SNMP计数器。

第一项运算是由ipq_kill完成的（通过调用ipq_unlink）。因为该函数也会在其他上下文中被调用（而非仅由ip_expire），因此，其停掉ipq定时器的尝试并非无用。由ip_expire调用时，它不会停止任何定时器，但是，在其他情况下可能会停掉一个定时器。如果为封包运行着一个定时器，则当该定时器启动时，ipq的引用计数值会递增。因此，为了让引用计数保持正确，删除该定时器后，ipq_kill会递减引用计数值。

除了ip_expire之外，还有另外两种情况会引发对ipq_kill的调用：

- 当接收到最后一个遗漏的片段时，ip_frag_reasm会调用ipg_kill。
- ip_evictor(本节开始曾介绍过)会调用ipg_kill来杀掉它选中要删除的ipq结构。

无论ipq_kill被调用的原因为何，都会设定COMPLETE标志，而ipq结构会从其所在的所有链表上解除。也就是说，COMPLETE标志不见得一定是指向已完全重组的IP封包。

hash表重新组织

由图22-1已知，送进来的片段等着被重组时在内存中的组织方式。在顶端，所有封包的所有片段都是通过名为ipq_hash的hash表存取的。当hash函数可以尽可能均匀地散播各个元素时，hash表就运作得最好。如果允许大量冲突存在，会使得IP片段群集在附属于ipq_hash的一些元素的一些链表内，使得性能下降，甚至于让DoS攻击之门洞开。为了避免这些冲突，Linux内核会定期使用不同hash函数来重新组织表中的所有IP片段。只有时常重新组织，而且每次选中的新函数无法从前一个函数猜中时，这种机制才有效。

片段的重新组织是用一个定时器着手进行的，而该定时器由 ipfrag_init 启动，默认地，每 10 分钟会到期一次（到期时间可以通过 */proc* 接口配置，参见第二十三章“通过 /proc 文件系统做调整”一节）。

该定时器到期时将执行的函数 ipfrag_secret_rebuild，该函数其实相当简单。每次执行时，会以 get_random_bytes 产生一个随机数，然后将此值存储在全局变量 ipfrag_hash_rnd 中（由 ipqhashfnhash 函数使用）。然后，hash 表内的每个元素会逐一解除，而它的 hash 值会用 ipqhashfn 重算（现在使用 ipfrag_hash_rnd 的新值），最终重新插入至表中。

ipfrag_hash_rnd 会先在 ipfrag_init 里初始化（没有使用 get_random_bytes），因为 get_random_bytes 函数依赖系统的一个量（名为“熵”，其值通过检查传统上发生次数不定的系统事件而设定，并随着时间推）。计算机启动时，可能没有足够的熵能让 get_random_bytes 借以求出随机数。

iqp 结构的重新组织不会影响 ipq_lru_list 链表。

第二十三章

因特网协议第4版(IPv4)：其他主题

本章依旧围绕在网络程序中对IPv4层的讨论上，说明一些一般性议题。例如，内核对IPv4层中信息的管理、统计数据和用户接口（通过*/proc*）。本章也会描述IPv4协议的局限，也因此才发展了IPv6。

长效IP端点信息

在IP层上，没有所谓的有状态连接的概念。因为IP是无状态协议，所以，除了统计资料外，没有保留任何参数或者和连接相关的数据结构（这些是可选的，但是，协议本身并不需要）。然而，为了改善性能，内核会按每个目的地IP地址，记录一些参数的信息。等一下会看一个关于这方面的范例。

任何主机只要最近和Linux机器交换过数据，就会被视为一个IP端点。内核会为每个端点分配一个数据结构，来保留一些长效信息（long-living information）。此时，该结构中并没有持有很多参数。最重要的参数是IP封包ID。由第十八章可知，每个IP封包都是通过一个名为ID的16位字段识别的。这种ID不是一种共享ID，每个IP端点都有一个唯一的实例，无论目的地是哪里，每个IP封包的ID值都会递增（这种解法只是实现上的选择，并非任何标准所要求的）。第十八章已对封包ID做过一些讨论了。

端点是由`inet_peer`结构表示；而该结构定义在*include/net/inetpeer.h*中，在“inet_peer结构”一节会说明。这些`inet_peer`结构会组织成一棵AVL树（这种著名的数据结构专为查询而做了最优化处理）。我不打算详谈AVL数据结构；你可以在任何程序语言书籍中找到该结构的说明（注1）。然而，值得一提的是AVL树所牵涉的代价。

基本上，由于插入和删除运算的定义方式，该树能够保持平衡。因为该树平衡，搜寻总

注1：*net/ipv4/inetpeer.c*顶端的批注就相当明确以及足以自我说明了。

是会花上 O（lg n）时间；此处的 n 是此树中的元素数目。一般而言，要让树保持平衡是要付出成本的，因此，这种数据结构通常是用在和插入/删除/修改运算相关的查询有很多的时候，以及这些查询的速度特别重要的时候。

整棵AVL树及相关的全局变量（例如，peer_total）都是由peer_pool_lock锁保护的。该锁的取得可以是在共享和独占模式中。查询只需读取权，因此，可以采用共享模式取得此锁，而插入/删除运算就必须以独占模式取得该锁。

初始化

端点子系统是由 inet_initpeers（定义在 *net/ipv4/inetpeer.c*；当 IPv4 协议于开机期间做初始化时，会由 ip_init 调用）做初始化。

该函数会完成三项主要任务：

- 分配缓存（用于持有 inet_peer 结构，当端点被认出时，就会分配出去）。
- 定义门阈值（inet_peer_threshold），用于限制inet_peer结构能用的内存量。其值的计算是根据系统中 RAM 的数量。当一个新项目建立时，全局计数器 peer_total 会被递增；而当一个元素被删除时，其值当然也会被递减。如果 peer_total 变得大于该门阈值，最久使用过的那个元素会被删除（参见 inet_getpeer）。
- 启动垃圾收集定时器。我们会在“垃圾收集”一节说明此任务。

查询

搜寻关键字是目的地 IP 地址，有两个主要函数：

lookup

这是 *net/ipv4/inetpeer.c* 里的宏，在 AVL 树中实现简单的搜寻。

inet_getpeer

此函数可以由其他子系统使用。例如，TCP和路由，用来搜寻指定项目。此函数建立在 lookup 之上。

inet_getpeer会接收搜寻关键字（端点的IP地址）以及一个标志（create），该标志可用于在搜寻失败时建立一个新项目。当一个新项目建立时，初始 IP 封包 ID 会由 secure_ip_id设成一个随机值。

图 23-1 是 inet_getpeer的内部逻辑。这个函数相当简单，不需要多做说明。然而，有

一点值得澄清：为何要有两次查询来了解正在请求者是否已有一个项目具有相同目的地地址。第二次检查并非多余的，因为类似项目可能在读取锁放开以及写入锁被取得之间的时间建立而加入该树中。

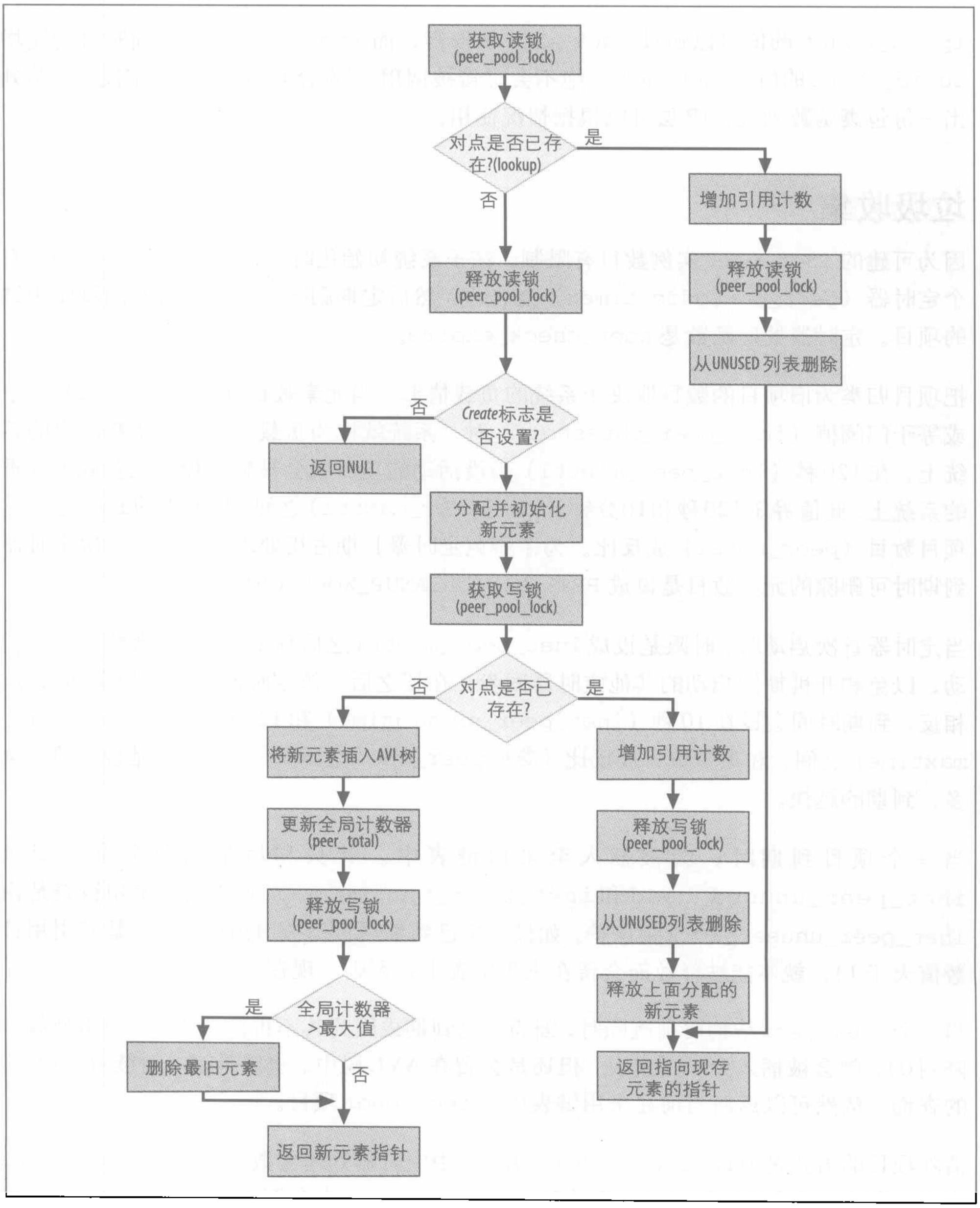


图 23-1：inet_getpeer 函数

IP 层使用 inet_peer 结构的方式

在 inet_peer 结构的那些字段中，当前只有两个由 IP 层使用：v4addr（用户识别端点）和 ip_id_count。

ip_id_count 的值可以通过 inet_getid 取得，而 inet_getid 也会同时自动递增 ip_id_count 的值。inet_getid绝不会直接被调用。“选择IP报头的ID字段”一节列出一份包裹函数列表，IP 层可以根据情况使用。

垃圾收集

因为可建的 inet_peer 实例数目有限制，在子系统初始化时（inet_initpeers），有个定时器（peer_periodic_timer）会启动，然后定期删除一些给定时间内没使用过的项目。定时器处理函数是 peer_check_expire。

把项目归类为旧项目的数目取决于系统的负载情形。当元素数目（peer_total）大于或等于门阈值（inet_peer_threshold）时，系统就视为负载沉重。在负载沉重的系统上，在 120 秒（inet_peer_minttl）后没活动的项目就会被删除掉。在负载不沉重的系统上，此值界于120秒和10分钟（inet_peer_maxttl）之间，和未决的inet_peer项目数目（peer_total）成反比。为了避免定时器长期占用处理器资源，每次定时器到期时可删除的元素数目是设成 PEER_MAX_CLEANUP_WORK（30）。

当定时器首次启动时，时限是设成inet_peer_minttl之后到期，不过，会略微有点扰动，以免和开机期间启动的其他定时器同步。在那之后，该定时器并非真的定期执行。相反，到期时间会设在 10 秒（inet_peer_gc_mintime）和 120 秒（inet_peer_gc_maxtime）之间，和项目数目成反比（参见 peer_check_expire），也就是说，项目越多，到期的越快。

当一个项目到期时，会被插入至未用链表中。其头和尾是由两个全局变量inet_peer_unused_head和inet_peer_unused_tailp所指。未用链表是由inet_peer_unused_lock锁保护。如果一个已到期项目仍被引用中（也就是其引用计数值大于 1），就不能被释放而会留在未用链表中；否则，现在就可释放。

当一个inet_peer结构要被删除时，因为其已到期或因为其不再被使用（其引用计数值降到0），就会被插入到未用链表，但还是会留在AVL树中。也就是说，后续对AVL树的查询，依然可以返回当前在未用链表中的 inet_peer 项目。

清除项目的方式是通过 cleanup_once 函数（由定时器处理函数peer_check_expire调用；此外，当项目数量超过容许值时，inet_getpeer也会调用）。cleanup_once的

输入参数指出inet_peer实例必须在未用链表中待过多久时间才可以被删除。其值为0时（由inet_getpeer所用），表示任何实例都适合。

当访问未用链表中的项目时（也就是对AVL树查询而被选中），就会从该链表中删除。因此，一个项目在其生命期间，可以进出未用链表数次（参见inet_getpeer）。

选择IP报头的ID字段

IP封包ID初始化的主要函数是__ip_select_ident。该函数可直接或间接由ip_select_ident或ip_select_ident_more调用。这两个包裹函数会区分可分段以及不可分段的封包（根据MF标志）。有两种定义情况：

不能分段的封包（DF=1）

加入这种情况是为了处理Windows系统IP协议栈中的bug（注2）。ID会间接从sock数据结构中取出（inet_sk(sk)->sk），每次包裹函数访问该结构时，其值都会递增。这可以确保每次传输时IP ID都会改变。

可以分段的封包（DF=0）

ip_select_ident会处理ID。

ip_select_ident_more（由TCP使用，参见ip_queue_xmit）会多接收一个输入参数（more），用于设备支持TCP卸载（TCP offloading）的情况。

我们回到__ip_select_ident：

```
void __ip_select_ident(struct iphdr *iph, struct dst_entry *dst, int more)
{
    struct rtable *rt = (struct rtable *) dst;

    if (rt) {
        if (rt->peer == NULL)
            rt_bind_peer(rt, 1);
        if (rt->peer) {
            iph->id = htons(inet_getid(rt->peer, more));
            return;
        }
    } else
        printk(KERN_DEBUG "rt_bind_peer(0) @%p\n",
               __builtin_return_address(0));
        ip_select_fb_ident(iph);
}
```

注2：　根据源代码的批注，这个议题和RFC 1144文件所说的TCP/IP报头压缩算法的实现有关。

从“长效IP端点信息”一节可知，就每个IP端点而言，都有一个inet_peer数据结构，除了其他信息外，还有一个计数器，可用于设定IP封包ID（iph->id）。当此ID可用时，__ip_select_ident就会使用此ID，不然，就会返回去使用ip_select_fb_ident。

如果inet_peer结构尚未在路由缓存项目rt中做好初始化，则rt_bind_peer会先寻找和该端点相匹配的inet_peer结构，然后，如果不存在，此函数就会试着建立（因为给rt_bind_peer的最后一个输入参数是设为1）。这种建立的尝试会在负载沉重而耗尽内存的系统上失败（无法分配新的inet_peer结构）。就此而言，作为最后手段，__ip_select_ident会以ip_select_fb_ident产生一个ID。

ip_select_fb_ident [fb是指*fallback*（退守）] 的工作方式很简单：持有一个静态变量ip_fallback_id，再和端点的目的地IP地址结合，然后传给我们在“查询”一节看过的secure_ip_id函数。这种做法的唯一缺点是因为该函数可能会因被好几个端点使用，所以，无法保证在合理的时间内，指派给传送到任何已知端点的连续IP封包的ID会不一样。让传给相同目的地的不同IP封包有不同的ID是很重要的，因为IP ID是用于处理重组的字段之一。于是，如果拥有相同ID的不同IP封包被分段了，而那些片段又混在一起，接收者就无法区分出哪些片段各自属于不同的IP封包（参见第十八章“把片段和其所属的IP封包相配”一节）。

IP统计资料

Linux内核记录好几组有关不同事件和情况的统计资料，这些资料有助于查核、除错、或者确认和标准的兼容性。本章我们只简述IP协议层所记录的统计数据（不谈SNMP基础架构）以及其更新方式。前几章中，特别是在说明各种函数时，我们有看见一些情况，使用例如IP_INC_STATS这类的宏来更新某些计数器的值。

我们先从包含和IP协议相关的所有计数器的数据结构着手。其名为ip_statistics，定义在*net/ipv4/ip_input.c*。该结构为向量（有两个指针），每个CPU都有一个这种结构，而每个结构都指向一个ipstats_mib（注3）结构（定义在*include/net/snmp.h*）向量。这类向量的分配是在init_ipv4_mibs（位在*net/ipv4/af_inet.c*）里做的。

```
static int __init init_ipv4_mibs(void)
{
        ...
        ip_statistics[0] = alloc_percpu(struct ipstats_mib);
        ip_statistics[1] = alloc_percpu(struct ipstats_mib);
```

注3：如前所述，MIB指的是Management Information Base（管理信息基库），用于描述一群对象（通常是计数器）。

```
        ...
    }
```

ipstats_mib结构就是声明成一个数组（里面是一些unsigned long字段）；它的尺寸是__IPSTATS_MIB_MAX，碰巧就是 *include/linux/snmp.h* 里的IPSTATS_MIB_*XXX*列举的大小。

以下是那些IPSTATS_MIB_*XXX*值的意义（分成四组）。详细的说明，可以参考IPv4的RFC 2001文件以及IPv6的RFC 2465文件。IPv4没用的那些IPSTATS_MIX_*XXX*计数器也没在RFC 2011文件中定义（IPSTATS_MIB_INADDRERRORS例外）。

和已接收封包有关的字段

IPSTATS_MIB_INRECEIVES

已接收封包数目。此字段不区分完整IP封包和片段，也包含会被接收及因任何原因会被丢弃的封包数（但是，处于混杂模式下的接口传给ip_rcv的数据帧如果不是要传给进行接收的接口因而被丢弃时，就不算在内）。在ip_rcv的开头处，其值就会被更新。

IPSTATS_MIB_INHDRERRORS

因为IP报头损坏而被丢弃的封包数目（片段以及非分段封包）。此字段可在ip_rcv和ip_rcv_finish里因为各种理由而被更新。

IPSTATS_MIB_INTOOBIGERRORS

在IPv4没用此字段。IPv6以此计算那些因为必须被分段（和IPv4不同，IPv6的路由器不可做分段运算）而无法转发的入口IP封包数目。

IPSTATS_MIB_INNOROUTES

当前没有使用。此字段应该计算那些因为本地主机没有有效路径而无法转发的入口封包数目。

IPSTATS_MIB_INADDRERRORS

IPv4当前没有使用。IPv6以此计算那些已接收到但具有错误地址类型的封包数目。

IPSTATS_MIB_INUNKNOWNPROTOS

已接收到但L4协议为未知协议（也就是该协议没有已注册的处理函数）的封包数目。此字段会在ip_local_deliver_finish里更新。

IPSTATS_MIB_INTRUNCATEDPKTS

封包被截断了（也就是不包含完整的IP报头）。这是由IPv6使用，但IPv4没用。

IPSTATS_MIB_INDISCARDS

被丢弃的封包数目。关于这方面的计数器不包含因为报头错误而被丢弃的封包数，

其主要是包含内存分配问题而丢弃的封包数。此字段是在ip_rcv和ip_rcv_finish里被更新。

IPSTATS_MIB_INDELIVERS

成功传递至L4协议处理函数的封包数目。此字段是在ip_local_deliver_finish里被更新。

IPSTATS_MIB_INMCASTPKTS

已接收的多播封包数目。此字段由IPv6使用，但IPv4没用。

和已传输封包有关的字段

IPSTATS_MIB_OUTFORWDATAGRAMS

必须被转发的入口封包数目。实际上，在封包传输前，以及理论上封包可能会因某种因素而被丢弃时，关于统计入口封包的计数器就会被递增。其值会在ip_forward_finish里被更新（对多播而言是在ipmr_forward_finish里）。

IPSTATS_MIB_OUTREQUESTS

系统试着传输的封包数目（成功或失败），但不包括转发封包。此字段是在ip_ouput里被更新（对多播而言是在ip_mc_output里）。

IPSTATS_MIB_OUTDISCARDS

传输失败的封包数目。此字段会在包括ip_append_data、ip_push_pending_frames、raw_send_hdrinc等的几个被更新。

IPSTATS_MIB_OUTNOROUTES

因为无路径传输而被丢弃的本地产生的封包数目。正常来讲，此字段是在ip_route_output_flow失败后而被更新。另外，ip_queue_xmit也是其中一个会更新此字段的函数。

IPSTATS_MIB_OUTMCASTPKTS

已传输多播封包的数目。目前，IPv4没有用此字段。

和重组有关的字段

IPSTATS_MIB_REASMTIMEOUT

重组失败的封包数目（因为有些片段没有实时收到）。此值是完整封包的数目，而非片段数目。此字段会在ip_expire里更新；而ip_expire是在IP片段因超时而被丢弃时所执行的定时器函数。注意，关于统计重组失败的封包数目的计数器的用法和本节开头所提的两份RFC文件里的定义并不相同。

IPSTATS_MIB_REASMREQDS

已接收片段的数目（也就是试着重组的数目）。此字段会在ip_defrag里被更新。

IPSTATS_MIB_REASMFAILS

重组失败的封包数目。此字段会在几个地方因不同原因而被更新(__ip_evictor、ip_expire、ip_frag_reasm以及ip_defrag)。

IPSTATS_MIB_REASMOKS

成功重组的封包数目。此字段是在ip_frag_reasm里被更新。

和分段有关的字段

IPSTATS_MIB_FRAGFAILS

分段尝试失败次数。此字段是在ip_fragment里被更新（对多播而言是在ipmr_queue_xmit)。

IPSTATS_MIB_FRAGOKS

已传输的片段数目。此字段是在ip_fragment里被更新。

IPSTATS_MIB_FRAGCREATES

已创建的片段数目。此字段是在ip_fragment里被更新。

这些计数器的值会在*/proc/net/snmp*文件里输出。

每个CPU都会保持其所处理的封包的查核信息。此外，CPU还会持有两个计数器：一个是针对中断环境下的事件，另一个是针对中断环境外的事件。因此，ip_statistics数组包含两个元素（每个CPU都有），一个是针对中断环境，而另一个是针对非中断环境。并非所有事件都可在这两种环境下发生，但是，为了让事情简单而清楚一点，此向量在定义时就是双倍大小；那些在这两种环境之一无意义的元素只是简单地没被使用。

因为有些程序片段可以在中断环境内和中断环境外执行，内核提供三个不同宏可以把事件加进IP统计数据向量中：

```
#define IP_INC_STATS      (field)     SNMP_INC_STATS      (ip_statistics, field)
#define IP_INC_STATS_BH   (field)     SNMP_INC_STATS_BH   (ip_statistics, field)
#define IP_INC_STATS_USER(field)      SNMP_INC_STATS_USER(ip_statistics, field)
```

第一个宏可以用在任一种环境内，因为其内部会检查它是否在中断环境中被调用，然后据以更新正确元素。第二个宏和第三个宏分别用于发生在中断环境内和中断环境外的事件。宏IP_INC_STATS、IP_INC_STATS_BH以及IP_INC_STATS_USER是定义在*include/net/ip.h*内，而那三个SNMP_INC_*XXX*宏则定义在include/net/snmp.h。

IP 配置

系统管理员可以用各种方式对Linux IP协议进行调整和配置。这种调整包括对协议本身以及对设备配置进行变更。所用到的四个主要接口如下：

通过 ifconfig 发出 ioctl 调用

ifconfig 是古老的 Unix 旧式工具，可以用于为网络设备配置 IP。

通过 ip 使用 RTNetlink

ip 是 IPROUTE2 套件的一部分，是 Linux 所提供的可用于配置网络设备 IP 的较新的工具。

/proc 文件系统

协议行为可以通过目录 */proc/sys/net/ipv4* 下的一些文件调整。

RARP/BOOTP/DHCP

这三个协议可用于动态分配 IP 配置给主机及其接口。

上述列表的最后一组协议有个有趣的缠绕。这组协议通常是在用户空间中实现的，但是，Linux也提供一个简单的内核空间实现，可以配合nfsroot开机选项一起使用。nfsroot可以让内核通过 NFS 把根目录（/）挂载进来。为此，系统对用户空间的 IP 配置（可能是存储在远程分区内，当内核挂进根目录时，系统还无法使用该分区）进行初始化前，内核就必须先于开机期间取得 IP 配置。通过内核开机选项，就有可能给 nfsroot 静态配置，或者指定要使用哪些协议（可以同时使用一个以上的协议）来取得该配置。IP配置程序位在 *net/ipv4/ipconfig.c*，而 nfsroot 所用的程序放在 *fs/nfs/nfsroot.c*。这两个文件会交叉引用变量和函数，但是，它们实际上很容易阅读。不过我们不会说明，因为网络文件系统和用户空间客户程序不在本书讨论范围内。一旦你知道如何阅读__setup宏（第七章有说明），阅读这些程序应该也算不了什么。程序很明确，注释也很清楚。

列表中的第三项 */proc* 会在稍后“通过 /proc 文件系统做调整”一节说明。

本节中，我会多谈一点支持前两项 *ifconfig* 和 *ip* 行为的内核接口。这里的目的不是讨论用户空间命令或处理配置请求的相关内核部分的内部细节，而是显示用户空间和内核空间是如何沟通的，以及如何启用内核函数以响应用户空间命令。

操作 IP 地址和配置的主要函数

在 *net/ipv4/devinet.c* 里，你可以发现好几个函数，可以用于把IP地址加给网络接口、从接口删除地址、修改地址、取出设备的IP配置（指定设备索引值或net_device数据结

构）等等。这里我只介绍一些将要用到的函数，它们帮助你了解稍后讨论 *ip* 和 *ifconfig* 用户空间工具。

阅读这些函数的说明前，再看一看IP层所用到的重要数据结构（第十九章介绍的，以及本章稍后详细说明的）应该有价值。例如，IP地址是由 `in_ifaddr` 结构表示，而设备的完整IPv4配置是由 `in_device` 结构表示。

`inetdev_init` 和 `inetdev_destroy`

当第一个IP配置施加至一台设备时，`inetdev_init` 就会被启用。该函数会分配 `in_device` 结构，并将它连接至相关的 `net_device` 实例。此外，该函数也会在 */proc/sys/net/ipv4/conf/* 里建立一个目录（参见“通过/proc文件系统做调整”一节）。

可以通过 `inetdev_destroy` 把IP配置删除，`inetdev_destroy` 就是把 `inetdev_init` 里所做的事复原而已，另外把所有连接的 `in_ifaddr` 结构都删除掉。`in_ifaddr` 结构是随着 `inet_free_ifa` 删除掉的，此外，`inet_free_ifa` 也会使用 `in_dev_put` 把对 `in_device` 结构的引用计数值递减掉。当最后一个引用释放时（可能是最后一次调用 `inet_free_ifa`），`in_device` 实例就会在 `in_dev_finish_destroy` 里被释放掉。

`inet_alloc_ifa` 和 `inet_free_ifa`

这两个函数分别会分配和释放一个 `in_ifaddr` 数据结构。当用户新增一个新地址至一个接口时，就会分配一个新结构。删除单一地址时，或者一起删除所有设备的IP配置时，就可以触发删除。这两个函数都会使用RCU机制作为互相排除的手法。

`inet_insert_ifa` 和 `inet_del_ifa`

`inet_insert_ifa` 会在 `in_device` 里的链表中增加一个新的 `in_ifaddr` 结构。此函数会检测重复部分，然后，如果它发现有个地址落在另一个地址的子网内，就为将其标示为次要。例如，假设 *eth0* 已经有地址 10.0.0.1/24；当新地址 0.0.0.2/24 加入时，就第一个地址而言，此地址会被视为次要。主要地址也会用于喂送内核随机数产生器的熵（以 `net_srandom`）。有关主要地址和次要地址的信息，可以在第三十章找到。

`inet_del_ifa` 就是从相关 `in_device` 实例删除一个 `in_ifaddr` 结构，同时确保如果该地址为主要地址，所有相关的次要地址也会删除掉，除非管理员刻意通过该设备的 */proc/sys/net/ipv4/conf/dev_name/promote_secondaries* 文件来配置该设备，不要删除次要地址。相反的，当相配的主要地址被删除时，一个次要地址可以提升成主要地址。指定 `in_device` 实例后，此项配置就可通过 `IN_DEV_PROMOTE_SECONDARIES` 宏存取。`inet_del_ifa` 函数会接收另一个输入参数，可用于辨别当

最后一个in_ifaddr实例删除时，是否要将in_device结构删除掉。虽然把空的in_device结构删除掉是很正常的，但偶尔调用者可能不会做这件事。例如，当它知道很快就会新增一个新的in_ifaddr时。

对新增和删除这两种情况而言，成功完成后，会引发rtmsg_ifa做Netlink广播通知（参见“变更通知：rtmsg_ifa”一节），以及通过inetaddr_chain通知链把通知消息传给其他内核子系统（参见第四章）。

inet_set_ifa

这是内含inet_insert_ifa的包裹函数，如果相关设备没有存在in_device结构，就建立一个结构，然后把127.*x.x.x*这类地址的地址Scope设成本地（RT_SCOPE_HOST）。参考第三十章“Scope”一节有关Scope的细节。

还有很多其他较小的函数可以让程序更具可读性。以下是其中的一些函数：

inet_select_addr

该函数用于在指定设备上所配置的那些IP地址中选出一个IP地址。此函数接受另一个可选参数是Scope，可用于缩小查询范围。我们会在第三十五章看见此函数的用处。

inet_make_mask和inet_mask_len

给定网络掩码（netmask）所组成的1的数目，inet_make_mask就可建立相配的网络掩码。例如，输入值24会产生的网络掩码就是十进制表达形式255.255.255.0。

inet_mask_len则是相反，会返回十进制网络掩码中1的数目。例如，255.255.0.0会返回16。

inet_ifa_match

给定一个IP地址和一个网络掩码后，inet_ifa_match会检查指定的第二个IP地址是否落在相同子网内。此函数通常也用于分类出次要地址，以及检查指定的IP地址是否属于那些本地配置子网之一。可参见inet_del_ifa。

for_primary_ifa和for_ifa

这两个函数是宏，可用于浏览给定的in_device结构内相配的所有in_ifaddr实例。for_primary_ifa只考虑主要地址，而for_ifa则是考虑所有地址。

变更通知：rtmsg_ifa

Netlink提供RTMGRP_IPV4_IFADDR多播群组给那些对本地配置IP地址的变更感兴趣的用户空间应用程序使用。当本地IP地址有任何变更发生时，内核就会使用rtmsg_ifa函数去通知那些对该群组有注册的应用程序。

RTM_NEWADDR

一个设备上已配置了一个新地址。

RTM_DELADDR

一个设备上已删除了一个地址。

所产生的消息是由 inet_fill_ifaddr 做设定，而 inet_fill_ifaddr 也可用于处理来自于用户空间的显示请求（类似 *ip addr list* 的命令）。消息中包括要被新增或删除的地址以及相关的设备。

那么，谁对这种通知消息感兴趣？路由协议是主要的例子。如果你在用 Zebra，你已配置的路由协议会想把所有和已消失的地址直接或间接相依的路径删除掉。第三十一章中，你会进一步了解路由协议和内核路由子系统互动的方式。

inetaddr_chain 通知链

IP 子系统使用 inetaddr_chain 通知链来通知其他内核子系统有关本地设备 IP 配置的变更。内核子系统可以通过 register_inetaddr_notifier 和 unregister_inetaddr_notifier 函数对 inetaddr_chain 注册或注销。以下是使用此通知链的两个范例：

路由

参见第三十二章“外部事件”一节。

Netfilter 伪装

当一个本地 IP 地址由 Netfilter 伪装功能使用时，而该地址又消失了，那么所有使用该地址的连接都必须被丢弃（参见 *net/ipv4/netfilter/ipt_MASQUERADE.c*）。

当一个 IP 地址被删除了，以及当一个 IP 地址加入一个本地设备了，就会发出 NETDEV_DOWN 和 NETDEV_UP 这两个事件的通知消息。这类通知消息是由 inet_del_ifa 和 inet_insert_ifa 函数产生的（在“操作 IP 地址和配置的主要函数”一节有介绍）。

IP 配置（通过 ip）

传统上，Unix 系统管理员会使用 *ifconfig*、*route*、以及其他命令手动配置接口和路径。目前，Linux 提供一个大的 *ip* 命令以处理 IP 配置（有很多子命令）。

本节我们会说明 IPROUTE2 如何处理主要的地址运算，例如，新增和删除地址。一旦你熟悉这些操作，就可轻易读通其他操作的程序。

图23-2是IPROUTE2套件中涉及IP地址配置活动的文件和主要函数。线条上的标签是*ip*关键词，而节点是显示出所启用的函数及其所属的文件。例如，命令*ip address add*是由ipaddr_modify处理。

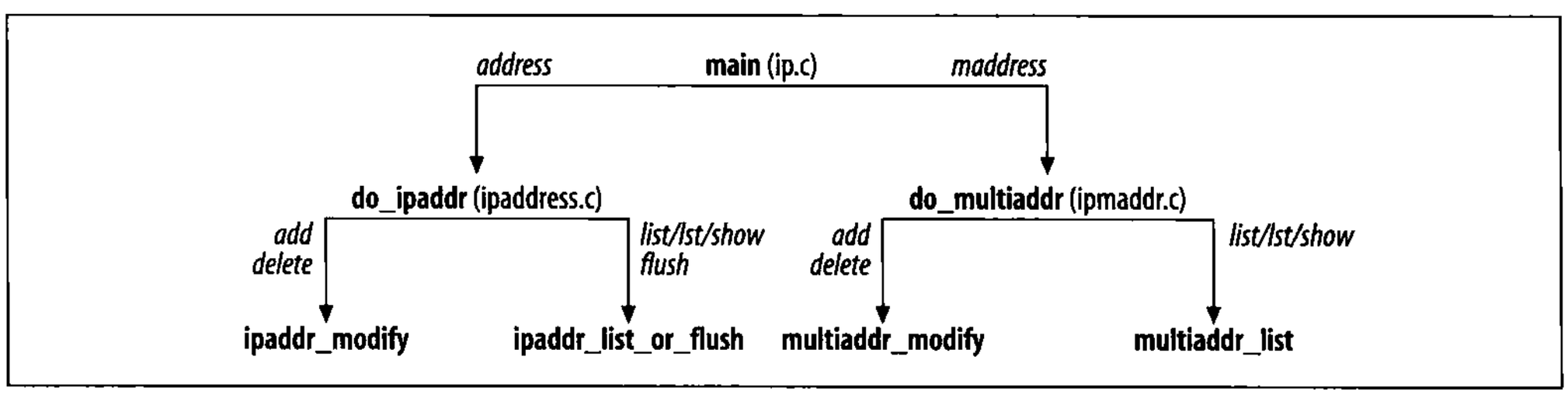


图23-2：IPROUTE2和地址配置相关的文件和函数

表23-1是命令行关键词（如*add*）指定的操作以及内核所执行的内核处理函数间的关联。例如，当内核接收到RTM_NEWADDR操作的请求时，就知道其和add命令相配，因此，会启用inet_rtm_newaddr。有些内核操作是超载式的，因此，就这类操作而言，内核需要其他标志才能搞清楚用户空间命令究竟要做什么，可参见第三十六章的范例。这样的关联是定义在inet_rtnetlink_table结构内（位在*net/ipv4/devinet.c*）。有关RTNetlink的介绍，参考第三章。

表23-1：ip route命令和相关内核操作

CLI关键词	操作	内核处理函数
add	RTM_NEWADDR	inet_rtm_newaddr
delete	RTM_DELADDR	inet_rtm_deladdr
list, lst, show	RTM_GETADDR	inet_dumpifaddr
flush	RTM_GETADDR	inet_dumpifaddr

*list*和*flush*命令需要一些说明。*list*命令只是对内核请求转储信息，例如，转储特定设备信息，而*flush*则是请求清除设备的整个IP配置。

inet_rtm_newaddr和inet_rtm_deladdr是两个内含通用函数inet_insert_ifa和inet_del_ifa（我们在“操作IP地址和配置的主要函数”一节有介绍）的包裹函数。这些包裹函数所做的事就是把来自于用户空间的请求转换成这两个更一般性的函数能懂的输入数据。他们也会过滤掉要给不存在设备的坏请求。

IP配置（通过ifconfig）

*ifconfig*是在*ifconfig.c*（*net-tools*套件的一部分）用户空间文件中实现的。和ip不同的是，*ifconfig*使用ioctl调用与内核衔接。然而，*ip*和*ifconfig*处理函数都使用同一组函数。第三章中，我们简要介绍了内核如何处理ioctl调用。这里我们所需知道的就是和IPv4配置相关的请求是由inet_ioctl函数（位在*net/ipv4/af_inet.c*）处理。根据ioctl程序，你可以看见inet_ioctl使用哪些辅助函数处理用户空间命令（如devinet_ ioctl）。

至于IPROUTE2，来自于*ifconfig*的用户空间请求是由一些包裹函数（最后会调用“操作IP地址和配置的主要函数”一节里提到的函数）在内核这一边处理。

IP-over-IP

IP-over-IP（也称为IP隧道或IPIP）就是传输在其他IP封包内的IP封包。该协议在一些有趣情况下有其用处，包括在VPN（Virtual Private Network，虚拟私有网络）中。当然，天下没有白吃的午餐，你可以想象得到这种协议加倍的沉重负担：因为每个IP封包内有两个IP报头，对小封包而言，消耗就变得很重，实现上也有一些微妙的复杂度。例如，两个报头的IP选项间的关系是怎么样的？

如果你只考虑IPv4和IPv6协议，就有四种可能的隧道组合。但是，并非所有组合都可能被使用。

让事情更复杂的是（我应该说是“灵活性”），别忘了，隧道内递归次数不限（注4）。

在Linux中创建的各种不同隧道接口不在本书讨论之列。然而，有了本书此卷有关IP实现的背景知识后，你可以研究*net/ipv4/ipip.c*和*include/net/ipip.h*里的程序以了解实现细节。

IPv4：问题出在哪儿？

从第十八章“IP协议：大蓝图”一节可知IP协议的主要任务是什么。IPv4是大约在26年前设计的（1981年），而从那时起，因特网和网络服务的速度已有很大变革，此协议也展现了它的历史。因为IPv4原本的设计就没办法考虑到当今大型网络拓朴结构以及商业应用，因此，随着岁月渐增，它的一些限制也显露出来。但这些问题只有部分解决而已，有些时候是通过对协议的扩充（例如，无类别网域间路由（classless interdomain

注4：IPv6定义了“隧道封装限度”作为嵌层封装的最大数目。参见RFC 2473的6.6节。

routing)、以DSCP替换ToS、阻塞通知等)，而其他时候则是定义专用的外部协议，例如IPsec。

感谢从IPv4所获取的经验，此协议的IPv6新版的设计就是为了解决IPv4的已知缺点，设计时考虑了如下方面：

- 功能。
- 易于配置。
- 性能。
- 从IPv4网络到IPv6网络的转换。
- 安全。

当然了，委员会设计新协议时试着尽可能让IPv4和IPv6可以兼容，并且从一种转换到另一种可以尽可能的顺利。这种兼容性和互动性不仅必须在应用程序层面处理，也得在内核层面处理。

分析IPv4封包传输时，我们知道，分段和选项的处理是两项最为艰难的任务。因此，IPv6解决了这两个问题，这种做法无不令人叹为惊奇。

- IPv6中限制了分段。IP封包只能在来源地分段。
- IP选项的存在偶尔会无法使用快速处理法。对软件路由器（如PC上的Linux）及商业硬件IP实现而言都是如此。就商业产品而言，就是说在硬件中转发没有选项的IP封包可以用较高的速率，而那些有选项的IP封包就得在软件中处理。IPv6处理选项的方式也不同：IPv6使用扩展的概念，它的主要优点是并非所有路由器都得处理选项。

IPv4的另一项大限制是它的地址为32位大小，以及因此带来的有限的层次结构。NAT是唯一的短期解决方案，可以部分解决此问题。但NAT有些限制，如下所示。

- 每个协议都得特殊对待，所以，有些协议不一定都能通过NAT路由器（如H323）。
- NAT路由器会成为单一失败点。因为它必须为所有通过它自身的所有连接都保持状态信息，因此，要设计出考虑有冗余或安全性的网络并不容易。
- 必须支持那些设计时就不考虑NAT支持的复杂协议时，NAT路由器的任务相当复杂，而且计算的负载沉重（这些协议会被视为“和NAT不友好”）（注5）。

注5： 如果你想知道什么可视为NAT友好协议或NAT友好应用程序，可以阅读RFC 3235。

IPv4的地址的有限数目（因其阶层有限）也造成必须建立巨大的路由表。核心路由器可以拥有多达几十万条路径。这种趋势并不好，主要原因有两个：

- 路径需要很多内存。
- 查询比较慢。

虽然无类别网域间路由可协助减少路由表的大小，但是，它无法解决IPv4有限地址空间的问题。

对IPv6而言，地址尺寸已大了四倍，这不是说地址量大了四倍，而是大了2^{96}倍！这样有可能可以让系统摆脱NAT路由器，使它成为因特网上成熟的成员，这也意味着新型态的应用程序会诞生。

IPv4设计时并没有考虑安全问题。因此，开发出了几种细致度有所不同的方法：应用程序点对点方法（如SSL）、主机点对点方法（如IPsec）等。每种方法各有其优缺点。SSL需到应用程序使用它的安全层（位于TCP之上），而IPsec（多数人所谓的VPN）就不是这样：IPsec位于L3层之上，因此，对于应用程序而言是看不见的。虽然IPv4和IPv6都可以使用IPsec，但是，IPsec更适合IPv6。

随着IPv6的产生，邻居系统也跟着变了。此时，改称邻居发现机制（neighbor discovery），而对IPv4而言，就是ARP的配对物。另外，QoS组件也有了扩展。

就IPv4网络而言，已经有可能进行自动化主机配置，这得感谢像DHCP这类协议；然而，有些限制使得这种做法没有达到应该有的PnP效果。这个问题也被IPv6解决了，也就是使用所谓的自动化配置功能（autoconfiguration feature）。

通过/proc文件系统做调整

第三章介绍了*/proc*文件系统；该文件系统提供简单接口，可让用户察看和修改内核参数，而且是较新*sysfs*目录的模型。该文件系统内有很多文件（或者说是虚拟数据结构，只是用户看起来像文件）对应到内核内的变量和函数，通过这些文件可以调整内核网络组件的行为。

调整IPv4的文件主要位在两个目录：

/proc/sys/net/ipv4/

表23-2列出此目录中由IPv4所用的一些文件。和这些文件相关的内核变量是在*net/ipv4/sysctl_net_ipv4.c*中声明的，而且会在开机期间静态注册（参见第三章）。注

意，此目录所包含的文件远超过表23-2所列。其他文件多数和L4协议有关，尤其是TCP。

/proc/sys/net/ipv4/conf/

此目录下，内核所识别的每个网络设备都有一个目录，此外，还有其他特殊目录（参见第三十六章图36-4）。这些子目录包含设备专用的配置参数；其中一些是accept_redirects、send_redirects、accept_source_route、以及forwarding。这些会在第三十六章讨论，但是promote_secondaries则是在“操作IP地址和配置的主要函数”一节中做了说明。

表23-2：/proc/sys/net/ipv4中和IPv4相关的文件

/proc文件名	相关的内核变量	默认值
ip_forward	ipv4_devconf.forwarding	0
ip_no_pmtu_disc	<C>ipv4_config.no_pmtu_disc	0
ip_autoconfig	ipv4_config.autoconfig	0
ip_default_ttl	sysctl_ip_default_ttl	IPDEFTTL (64)
ip_nonlocal_bind	sysctl_ip_nonlocal_bind	0
ip_local_port_range	sysctl_ip_local_port_range[0]	1
	sysctl_ip_local_port_range[1]	65535[a]
ipfrag_high_tresh	sysctl_ipfrag_high_thresh	256K
ipfrag_low_tresh	sysctl_ipfrag_low_thresh	192K
ipfrag_time	sysctl_ipfrag_time	IP_FRAG_TIME (30 * HZ)
ipfrag_secret_interval	sysctl_ipfrag_secret_interval	10 * 60 * HZ
ip_dynaddr	sysctl_ip_dynaddr	0
inet_peer_gc_maxtime	inet_peer_gc_maxtime	120 * HZ
inet_peer_gc_mintime	inet_peer_gc_mintime	10 * HZ
inet_peer_maxttl	inet_peer_maxttl	10 * 60 * HZ
inet_peer_minttl	inet_peer_minttl	120 * HZ
inet_peer_threshold	inet_peer_threshold	65536 + 128[b]

a：这此值是由tcp_init在开机期间更新（根据系统可用内存量而定）。即使是由TCP更新，任何使用端口的L4协议都可以使用这些值。

b：此值是由inet_initpeer在开机期间更新（根据系统可用内存量而定）。

表23-2的前三个元素是类型为ipv4_devconf和ipv4_config的两个数据结构（分别位于*include/linux/inetdevice.h*和*include/net/ip.h*中，本章稍后说明）的成员。这些结

构的其他成员不是在其他地方输出，就是根本不输出（我们会在相关章节说明）。这些文件和内核变量的含义如下：

ip_forward

设为非零值时，就可让设备转发流量。参见第三十六章“开启和关闭转发”一节。

ip_no_pmtu_disc

当为零时，路径MTU发现功能就会开启。

ip_autoconfig

当主机的IP配置是通过像DHCP这类协议完成时，就会设为1。参见“IP配置”一节。

ip_default_ttl

这是IP TTL字段的默认值（用于单播流量）。多播流量的默认值为1，而且没有对等的sysctl变量可供设定。

ip_nonlocal_bind

非零时，应用程序就有可能绑定至非本地主机地址。例如，这样可以让套接字绑定至一个地址（即使与之关联的接口关掉了）。

ip_local_port_range

可用于向外连接的端口范围。

ipfrag_high_thresh

ipfrag_low_thresh

用于限制传送进来的IP片段所用的内存量的门阈值。当片段所用的内存到达ipfrag_high_thresh时，旧项目就会删除，直到所用的内存下降至ipfrag_low_thresh。可参见“垃圾收集”一节。

ipfrag_time

传送进来的IP片段保留在内存内的最大时间值（在到期前）。

ipfrag_secret_interval

传送进来的IP片段从hash表中被抽出而又以不同hash函数重新插入的时间间隔。参见第二十二章“hash表重新组织”一节。

ip_dynaddr

该变量用于处理的情况是：绑定至按需拨号接口（dial-on-demand interface）的地址的套接字没接收到任何回复，直到该接口打开为止。如果ip_dynaddr被设定，套接字就会重新试着绑定。

inet_peer_threshold

可以分配的inet_peer结构的最大数目。

inet_peer_gc_maxtime

inet_peer_gc_mintime

两次定期垃圾收集间所经过的时间量。因为由那些inet_peer结构所用的内存量有限（inet_peer_threshold），有个定期定时器会根据两个变量让未用项目到期。当系统负载不沉重时，就使用inet_peer_gc_maxtime，而系统负载沉重时，就使用inet_peer_gc_mintime。于是，项目越多，此定时器就越常到期。

inet_peer_maxttl

inet_peer_minttl

inet_peer项目的最大和最小TTL。显然，它的值应该比sysctl_ipfrag_time还要大。

本部分涉及的数据结构

第十九章“主要IPv4数据结构”一节已经对主要结构做了简要介绍。本节对每种数据结构类型做详细说明。所定义的每种数据结构的文件如图23-3所示。

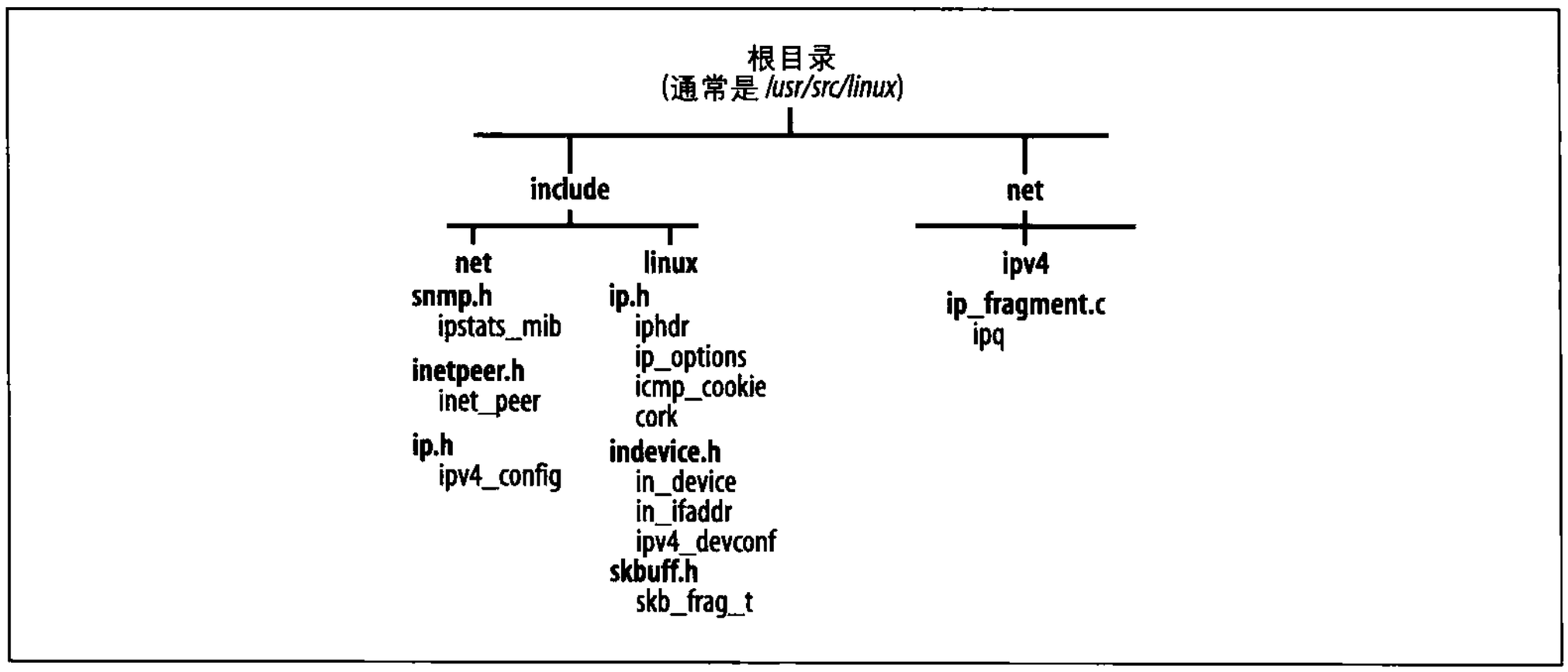


图23-3：这些数据结构在内核文件中的分布情形

iphdr结构

它的字段的意义已在第十八章“IP报头”一节谈过了。

ip_options结构

该结构代表的是必须被传输或转发的封包的选项。选项存储在此结构中是因为比起从IP报头中相应部分读取而言，读取此结构比较容易一些。

我们逐一说明它的字段。如果你已经读过第十八章“IP选项”一节，就应该很容易理解。读过之后，你应该更能理解分析工作是怎么做到的，以及IP层子系统如何使用其结果，例如处理送进来的IP封包的程序。有些位字段会组成一个unsigned char字段；这些字段的声明会以:1为结尾。

unsigned char optlen

这组选项的长度。如第十八章所述，根据IP报头的定义，其值最大为40字节。

unsigned char is_changed:1

如果IP报头已被修改（例如，IP地址或时间戳），就会设定。知道这件事非常有用处，因为如果封包要被转发，该字段会指出IP校验和必须重新计算。

__u32 faddr
unsigned char is_strictroute:1
unsigned char srr
unsigned char srr_is_hit:1

faddr只对已传输封包（也就是那些本地产生的封包）及使用来源地路由的封包有意义。faddr的值会设成提供来源地路由使用的IP地址列表中的第一个地址。参见第十九章“选项：严格和松散Source Routing”一节。当“Strict Source Route”为选项之一时，is_strictroute标志就会设定。

srr内含有报头中“Source Route”选项的偏移量。如果没用此选项，则其值为零。

如果封包使用源路由，而进行接收的接口的IP地址是来源地路由表中的地址之一，则srr_is_hit为真（参见*net/ipv4/ip_options.c*中的ip_options_rcv_ srr）。

unsigned char rr

当rr为非零时，“Record Route”就是IP选项之一，而该字段的值就代表该选项在IP报头中起始的偏移量。此字段和rr_needaddr一起合用。

unsigned char rr_needaddr:1

当rr_needaddr为真时，“Record Route”是IP选项之一，而且报头中还有空间可容纳另一条路径；因此，当前节点应该把外出接口的IP地址拷贝到rr在IP报头中所指的偏移量。

unsigned char ts

当ts为非零时，“Timestamp”是IP选项之一，而该字段的值就代表该选项在IP报头中起始的偏移量。此字段和ts_needaddr及ts_needtime一起合用。

unsigned char is_setbyuser:1

此字段只对已传输封包有意义，当选项从用户空间以setsockopt系统调用传递时，就会设定。然而，当前都没用到。

unsigned char is_data:1

unsigned char _data[0]

这些字段用在两种情况：当本地节点传输一个本地产生的封包时，以及当本地节点回复一条ICMP回应请求时。就这些情况而言，is_data为真，而_data会指向一个区域，此区域包含要附加到IP报头的选项。[0]的定义方式是很常见的惯例，用于替指针保留空间。

转发封包时，选项会放在相配的skb缓冲区内（参见*net/ipv4/ip_options.c*文件中的ip_options_get选项。）

unsigned char ts_needtime:1

当此选项为真时，“Timestamp”就是IP选项之一，而且报头中依然有空间可容纳另一个时间戳；因此，当前节点应该把传输时间加入IP报头，而位置就是ts所指定的偏移量。

unsigned char ts_needaddr:1

与ts和ts_needtime一起使用，以指出出口设备的IP地址也应该拷贝到IP报头。

unsigned char router_alert

当此选项为真时，“Router Alert”就是IP选项之一。

unsigned char __pad1, __pad2

因为当位置对齐32位边界时，内存的存取会比较快，Linux内核数据结构通常会采用无用字段（__padn）做填充，以便使其尺寸为32位的倍数。这就是__pad1和__pad2的用途，除此别无他用。

分析选项时，为了检查出存在次数超过一次以上的选项（为非法），标志srr、rr、以及ts也派得上用场；参见第十九章“选项的分析”一节。

ipcm_cookie结构

该结构结合了各种传输封包所需的信息。

```
struct ipcm_cookie
{
    u32                 addr;
    int                 oif;
    struct ip_options   *opt;
};
```

目的地IP地址是addr，出口设备是oif（有定义的话），而IP选项是ip_options结

构。注意，addr是唯一一定会设定的字段。如果没有对使用哪个设备做限制，oif就是0。

ipq结构

以下是ipq结构的字段的说明。为了简洁起见，第二十二章图22-1并没有显示所有字段。

struct ipq *next

当片段放进ipq_hash hash表时，冲突的元素（有相同hash值的元素）会用该字段连接起来。注意，该字段没有指出片段在封包内的次序；只是作为标准方式来组织hash表而已。封包内片段的次序是由fragments字段控制（参见第二十二章图22-1）。

struct ipq **pprev

这个指针是指回链表头部，而该链表里的IP封包都有相同hash值。

struct list_head lru_list

所有ipq结构都会在放一个全局链表ipq_lru_list中，而排序的条件是根据最近最少使用（LRU）的原则。执行垃圾收集时，这个链表就有用。该字段可用于把ipq结构连接至这样的一个链表。

u32 user

这是IP封包为什么重组的原因，等于间接说出是哪个内核子系统要求重组。IP_DEFRAG_*XXX*容许值列表的在*include/net/ip*.h中。最常见的是IP_DEFRAG_LOCAL_DELIVER，当重组的入口封包要传递给本地时，就会使用。

u32 saddr

u32 daddr

u16 id

u8 protocol

这些参数代表来源地IP地址、目的地IP地址、IP封包ID、以及L4协议标识符。如第十八章所述，这四个参数会指出片段所属的原有IP封包。因此，hash函数也会使用这些参数让元素能够在hash表中做最佳的分布。

u8 last_in

存储三个标志，可能的值是：

COMPLETE

所有片段都已接收，因此，可以连接起来获取原有IP封包。该标志也可用于标

示那些被选中要删除的ipq结构（参见*net/ipv4/ip_fragment.c*中的ipq_kill）。

FIRST_IN

片段中的第一个片段（offset=0的片段）已接收到了。第一个片段就是唯一一个携带原有IP封包中所有选项的片段。

LAST_IN

片段中最后一个片段（MF=0的片段）已经接收到了。最后一个片段很重要，因为这个片段会告知我们原有IP封包的尺寸。

struct sk_buff *fragments

是一个链表，内容是目前已接收的片段。

int len

偏移量最大的片段的结尾处的偏移量。当最后一个片段（MF=0的片段）已接收时，len会指出原有IP封包的尺寸。

int meat

代表我们到目前为止接收了原有封包多少字节。当前值和len相同时，表示该封包已完全接收了。

spinlock_t lock

保护此结构以避免处于竞争条件下。例如，不同IP片段由不同NIC同时接收，而且由不同CPU处理时，就会发生竞争情形。

atomic_t refcnt

计数器用于记录对该封包的外部引用。它的用途的例子如，timer定时器递增refcnt，来确保当定时器依然处于工作中时，没人可以释放ipq结构；否则，该定时器可能会在过期时试着存取已不存在的数据结构。你可以想象得到后果。

struct timer_list timer

第十八章说明了为何IP片段不能永久存在内存中，而且当重组不可能时，一段时间后，就应该删除。该字段就是处理此事的定时器。

int iif

传送最后一个片段的设备的ID。当一个片段链表到期时，该字段可用于决定由哪一个设备传输FRAGMENTATION REASSEMBLY TIMEOUT ICMP消息（参见*net/ipv4/ip_fragment.c*文件中的ip_expire）。

struct timeval stamp

最后一个片段接收的时间（参见*net/ipv4/ip_fragment.c*中的ip_frag_queue）。

ipq_hash表是由ipfrag_lock保护，而该锁可以在共享模式（只读）或独占模式（读写）下取得。不要把此锁和每个ipq元素内的锁搞混了。

inet_peer结构

内核会为最近联系过的每台远程主机都维护一个此结构的实例。在“长效IP端点信息”一节中，你已经知道该结构如何使用。所有inet_peer结构实例都放在一棵AVL树中，而该树是针对频繁查询做了最优化的结构。用于操作inet_peer实例的函数都放在*net/ipv4/inetpeer.c*中。

struct inet_peer *avl_left
struct inet_peer *avl_right

它们是指向两棵子树的左右指针。

__u16 avl_height

AVL树的高度。

struct inet_peer *unused_next
struct inet_peer **unused_prevp

用于把此节点连接到一个内含过期元素的链表。而unused_prevp是用于检查此节点是否在该链表中。

一个节点可以数次放进该链表中再从中取出，完全不用删除。参见“垃圾收集”一节。

unsigned long dtime

当此元素通过inet_putpeer加入未用链表inet_peer_unused_head的时间。

atomic_t refcnt

该元素的引用计数值。使用此结构者有路由子系统及TCP层等。

__u32 v4daddr

远程端点的IP地址。

__u16 ip_id_count

此端点下一个可用的IP封包ID（参见*include/net/inetpeer.h*中的inet_getid）。

__u32 tcp_ts
unsigned long tcp_ts_stamp

由TCP使用，来管理时间戳。

ipstats_mib 结构

SNMP 协议采用一种名为 MIB 的对象，借以收集有关系统的统计数据。一个名为ipstats_mib的数据结构包含IP层的统计资料。“IP统计资料”一节详细谈过这个结构。

in_device 结构

in_device结构是为网络设备存储所有IPv4相关配置。例如，用户用*ifconfig*或*ip*命令所做的变更。此结构会通过net_device->ip_ptr连接到net_device结构，而且可以用in_dev_get和__in_dev_get取出。这两个函数的差别在于前者会处理所有必要的上锁问题，而后者会假定调用者已处理上锁问题。

因为in_dev_get成功时（如设备配置成支持IPv4），其内部会对in_dev结构的引用计数递增，而其调用者用完该结构后，就应该以 in_dev_put 递减引用计数值。

该结构的分配及连接至设备的事是通过inetdev_init（其调用时机是当第一个IPv4地址在设备上配置时）。以下是其字段的含义：

struct net_device *dev

指回相配的net_device结构的指针。

atomic_t refcnt

引用计数值。此结构无法被释放，除非此字段为0。

int dead

此字段设定时就是把设备标示成已死。这可用于检查一些情况，例如，项目无法被销毁，因其引用计数值不为零，但是，销毁动作早就启动了。触发删除in_device结构的两个最常见事件是：

- 设备的注销（参见第八章）。
- 从设备上把最后一个配置的 IP 地址删除掉（参见 inet_del_ifa in *net/ipv4/devinet.c*）

struct in_ifaddr *ifa_list

设备上所配置的IPv4地址列表。in_ifaddr实例会按范围排序（范围大者排前面），而相同范围的元素则按地址类型排序（主要地址排前面）。in_ifaddr数据结构会在“in_ifaddr 结构”一节进一步说明。

struct neigh_parms *arp_parms

此字段的含义会在第六部分中详细说明。

`struct ipv4_devconf cnf`

参见“ipv4_devconf结构”一节。

`struct rcu_head rcu_head`

由RCU机制使用实现互斥。其完成之工作就如同锁一般。

剩余的字段由多播程序使用。例如，`mc_list`存储设备的多播配置，而且是`ifa_list`.`mr_v1_seen`的多播配对物；此外，`mr_v2_seen`是由IGMP协议所用的时间戳以记录1版和2版IGMP封包的接收。

in_ifaddr结构

在接口上配置一个IPv4地址时，内核会建立一个`in_ifaddr`结构，该结构中会包含一个4字节的地址及其他几个字段。以下是它们的含义：

`struct in_ifaddr *ifa_next`

指向链表中下一个元素的指针。此链表包含了设备上所配置的所有地址。

`struct in_device *ifa_dev`

指回相配的`in_device`结构的指针。

`u32 ifa_local`

`u32 ifa_address`

这两个字段的值取决于该地址是否被指派给一个隧道接口。如果是，`ifa_local`和`ifa_address`就是隧道的本地和远程地址。如果不是，则两者所含都是本地接口的地址。

`u32 ifa_mask`

`unsigned char ifa_prefixlen`

`ifa_mask`就是和该地址相配的网络掩码。`ifa_prefixlen`是构成此网络掩码的1的数目。因为这两者是以不同方式表达相同的信息，一般而言，两者之一可从另一者计算而得。例如，*ip*和*ifconfig*用户空间配置工具（“IP配置”一节有说明）就是这么做的。*ip*把`ifa_prefixlen`传给内核，然后，让`ifa_prefixlen`计算出`ifa_mask`；而*ifconfig*所做的刚好相反。内核提供一些函数可以把网络掩码转成前缀长度（prefix length），或者反过来转换。

`u32 ifa_broadcast`

广播地址。

`u32 ifa_anycast`

选播地址。

unsigned char ifa_scope

地址的范围。默认是RT_SCOPE_UNIVERSE(相当于0),而此字段通常设成*ifconfig/ip*所设的值，不过，也可以选不同的值。主要的例外是位于范围127.*x.x.x*里的地址，其范围为RT_SCOPE_HOST。参见第三十章的细节。

unsigned char ifa_flags

可能的IFA_F_XXX位标志都列在*include/linux/rtnetlink.h*中。以下是IPv4所用的一个标志：

IFA_F_SECONDARY

当一个新地址加至一台设备时，如果该设备已有另一个地址具有相同子网络，则该地址就会被视为次要地址。

其他标志则由IPv6使用。

char ifa_label[IFNAMSIZ]

使用此字符串主要是为和2.0.x内核兼容（允许别名接口（其名称类似*eth0:1*））。

struct rcu_head rcu_head

由RCU机制使用以实现互斥。它完成的工作和锁一样。

ipv4_devconf结构

ipv4_devconf数据结构（它的字段是通过*/proc in /proc/sys/net/ipv4/conf/*输出）是用于调整网络设备的行为。每个设备都有一个这种实例，而另外一个实例则用于存储默认值（ipv4_devconf_dflt)。其字段的意义会在第二十九章和第三十六章说明，不过，promote_secondaries例外，因其已在“操作IP地址和配置的主要函数”一节说明过。

ipv4_config结构

ipv4_devconf结构是用于存储每个设备的配置，而ipv4_config存储的配置则是用于主机身上。

以下是其字段的简要描述：

int log_martians

该参数也存在于ipv4_devconf结构内。当特定错误发生时，此参数可用于决定是否打印警告信息到控制台上。它的值不会被直接检查，而是通过宏IN_DEV_LOG_MARTIANS（可以把较高优先权给予这个设备实例)。

int autoconfig

没有使用。

int no_pmtu_disc

用于对变量inet_sock->pmtudisc（替套接字存储PMTU配置）做初始化。第十八章有详细描述路径MTU发现功能。

cork结构

cork结构（定义在inet_sock的定义内，位于*include/linux/ip.h*中）是用于处理套接字cork选项，如UDP的UDP_CORK及TCP的TCP_CORK（注6）。第二十一章已经知道它的字段如何用于维护一些环境信息（通过ip_append_data和ip_append_ page的连续调用，以处理数据分段）。

以下是其字段的简要描述：

unsigned int flags

目前IPv4只有一个标志可以设定：IPCORK_OPT。当此标志设定时，意味着opt中有选项。

unsigned int fragsize

产生的数据片段的尺寸。此尺寸包括有效载荷和L3报头，而且通常就是PMTU。

struct ip_options *opt

要用的IP选项。

struct rtable *rt

用于传输IP封包的路由表缓存项目。

int length

IP封包的尺寸（所有数据片段的和，不含IP报头）。

u32 addr

目的地IP地址。

struct flowi fl

有关连接两端点的信息集。细节可参考第三十六章。

skb_frag_t结构

第二十一章我们已经知道页面缓冲区是什么样子了（例如，参考该章的图21-5）。要指认一个内存页面中的一个数据块，就需要skb_frag_t的字段。

注6： IPv6定义了自己的cork版本（位于*include/linux/ipv6.h*中）。

`struct page *page`

指向内存页面的指针。就i386而言，页面尺寸大小为4 KB。要找出任何架构 *xxx* 的页面尺寸，可以找 *include/asm-xxx/page.h* 中的 `PAGE_SIZE`。

`__u16 page_offset`

偏移量，相对于页面的开端，也就是此片段起始处。

`__u16 size`

此片段的尺寸。

本部分涉及的函数和变量

表23-3总结了本书讨论IPv4协议各章中所介绍或引用的主要函数、变量以及数据结构。

表23-3：IPv4子系统中的函数、变量、以及数据结构

/proc文件名	相关的内核变量
ip_init	对IPv4协议进行初始化。参见第十九章“IP选项”一节
ip_rcv	处理入口IP封包。参见第十九章“处理输入IP封包”一节
ip_forward ip_forward_finish	转发入口IP封包或片段。参见第二十章“转发”一节
ip_local_deliver ip_local_deliver_finish	把入口IP封包传递给本地主机。参见第二十章“本地传递”一节
ipfrag_init ip_defrag ip_find ip_frag_queue ip_frag_reasm ip_frag_destroy ip_expire ip_evictor	对IP分段/重组子系统进行初始化 处理IP重组。参见第二十二章“IP重组”一节
ip_fragment ip_dont_fragment getfrag	处理IP分段。参见第二十二章“IP分段”一节

表23-3：IPv4子系统中的函数、变量、以及数据结构（续）

/proc文件名	相关的内核变量
ip_options_compile ip_options_parse ip_options_build ip_forward_options	处理IP选项。参见第十九章“IP选项”一节
ip_queue_xmit, ip_append_data, ip_push_pending_frames	由L4协议使用以传输IP封包。参见第二十一章“进行传输的重要函数”一节
dst_output	根据前次路由查询结果而启用正确的传输函数。参见第十八章图18-1
ip_finish_output ip_finish_output2	IP层传输函数和邻居子系统间的接口。参见第二十一章“衔接邻居子系统”一节
ip_decrease_ttl	递减IP报头的TTL字段，然后更新IP校验和
ip_fast_csum ip_send_check, ...	计算或更新IP校验和。第十八章“校验和计算的API”一节列出了其他很多这类函数
in_dev_get	返回网络设备的IP配置区块in_device，然后递增其引用计数值
inet_initpeers	对IP端点子系统做初始化
inet_getpeer	使用IPv4地址为关键字搜寻inet_peer结构
ip_select_ident ip_select_ident_more secure_ip_id	选择出口IP封包要用的IP ID
ip_call_ra_chain	把带有“Router Alert”选项的入口IP封包交给感兴趣的本地Raw套接字。参见第二十章“ip_forward函数”一节
IP_INC_STATS IP_INC_STATS_BH IP_INC_STATS_USER	递增用于记录IP流量统计数据的计数器。参见“IP统计资料”一节
inet_rtm_newaddr inet_rtm_deladdr inet_dump_ifaddr	处理来自于用户空间IPROUTE2套件的*ip addr*命令

表 23-3：IPv4 子系统中的函数、变量、以及数据结构（续）

/proc 文件名	相关的内核变量
inet_alloc_ifa inet_free_ifa inet_insert_ifa inet_del_ifa inet_set_ifa inet_select_addr inet_make_mask inet_mask_len inet_ifa_match	新增、删除、以及操作本地设备上配置的 IP 地址。参见“操作 IP 地址和配置的主要函数”一节
for_primary_ifa for_ifa	浏览网络设备上所配置的 IP 地址
rtmsg_ifa	产生有关本地设备 IP 配置变更的通知消息。参见“变更通知：rtmsg_ifa”一节
变量	
ipv4_devconf ipv4_devconf_dflt	存储一组可按每个设备做调整的参数（通过 */proc* 文件系统）。参见“通过 /proc 文件系统做调整”一节
ip_frag_mem	入口 IP 片段所持有的内存量。参见第三十三章“垃圾回收”一节
ipfrag_lock	用于 ipq 实例表的锁。参见第二十二章“IP 片段 hash 表的组织”一节
peer_total	peer_total 是未决的 inet_peer 结构的数目，而 inet_peer_threshold 是可用于分配 inet_peer 实例的最大内存量
inet_peer_threshold	
peer_pool_lock	用于 AVL 树的锁，而 inet_peer 结构会插入至此棵树
inet_peer_unused_lock	用于链表的锁，未用 inet_peer 结构会插入至此链表
ip_statistics	存储有关 IP 流量的统计资料。参见“IP 统计资料”一节

表23.3：IPv4子系统中的函数、变量、以及数据结构（续）

/proc文件名	相关的内核变量
数据结构	
struct iphdr	IPv4所用的主要数据结构。第十九章介绍过，而本章则详细描述
struct ip_options	
struct ipcm_cookie	
struct ipq	
struct ip_mib	
struct inet_peer	
struct in_device	
struct ipv4_devconf	
struct ipv4_config	
struct in_ifaddr	
struct cork	

本部分涉及的文件和目录

*net/ipv4*目录所包含的文件超过图23-4所列，但是，都有在其他章节讨论，包括第六部分和第七部分里的那几章。

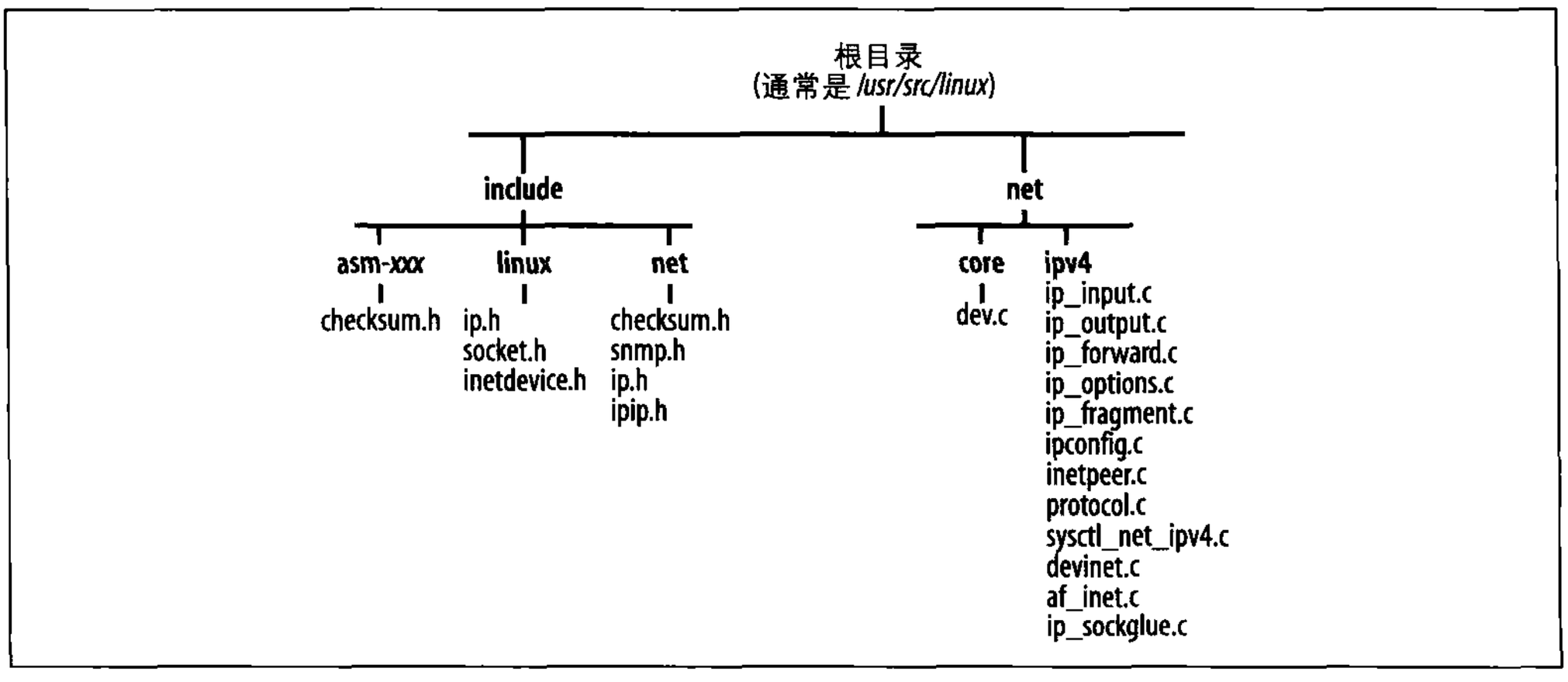


图23-4：本部分涉及的文件和目录

第二十四章

L4 协议和 Raw IP 的处理

本章描述 L3 和 L4 协议间的接口。此处唯一考虑的 L3 协议是 IP。L4 协议包括比较熟的 TCP、UDP、ICMP、以及其他几种协议。因为篇幅有限以及复杂度的缘故，本书没有谈论 L4 协议。然而，本章会说明当应用程序通过 Raw IP 处理其 L4（偶尔 L3）时会发生什么事情。

特别的是，本章解释以下问题：

- L4 协议如何和内核注册并告知内核它们感兴趣的流量是哪一种。
- 入口封包如何传给正确的 L4 协议处理函数。
- 应用程序如何告知内核让某个应用程序处理报头。

第二十一章已经知道 L4 协议用于传输 IP 数据报的那些函数。因为本书的焦点是 IP，本章只谈论那些位于 IP 之上的 L4 协议。本章会说明 IPv4 接口，然后简单说明 IPv6 与它的不同之处。

可用的 L4 协议

有些重要的L4协议是静态编译至内核中的。有几种比较少见的协议可以编译成模块。表 24-1 是静态编译进来的协议。

表 24-1：静态编译至内核的协议

协议	RFC#（年份）
UDP	768(1980)
ICMP	792(1981)
TCP	793(1981)

表24-2是属于第二类的一些协议。这些协议可以在内核配置中从“Networking Support”/“Networking Options”这一节加入内核内。

表 24-2：以模块实现的协议

协议	RFC#（年份）
Internet Group Management Protocol (IGMP)	1 版: 1112(1989) 2 版: 2236(1997) 3 版: 3376(2002)
Stream Control Transmission Protocol (SCTP)	2960(2000)
Protocol Independent Multicast，1 版(PIMv1)和 2 版(PIMv2)	2362(1998)
IPsec suite: IP Authentication Header Protocol (AH), IP Encapsulating Security Payload Protocol (ESP), IP Payload Compression Protocol (IPcomp)	AH: 2402(1998) ESP: 2406(1998) IPcomp: 3173(2001)
Generic Routing Encapsulation (GRE)	2784(2000)
IPv4-over-IPv4 tunnels (IPIP)	1853(1995)
IPv6 over IPv6	2473(1998)
Simple Internet Transition (IPv6-over-IPv4 tunnel, SIT)	1933(1996)

还有其他协议可供Linux内核使用，但是，这类协议不是在用户空间中实现（路由协议就是一例），就是以内核补丁形式存在（因为尚未整合到内核中）。

图 24-1 是 L4 协议在 L3 协议之上的情形。三个主要协议（ICMP、UDP 以及 TCP）、以及 IPsec 组都有 IPv6 上对应的协议。图 24-1 中没有 IGMPv6，因为它的功能已实现成 ICMPv6 的一部分。

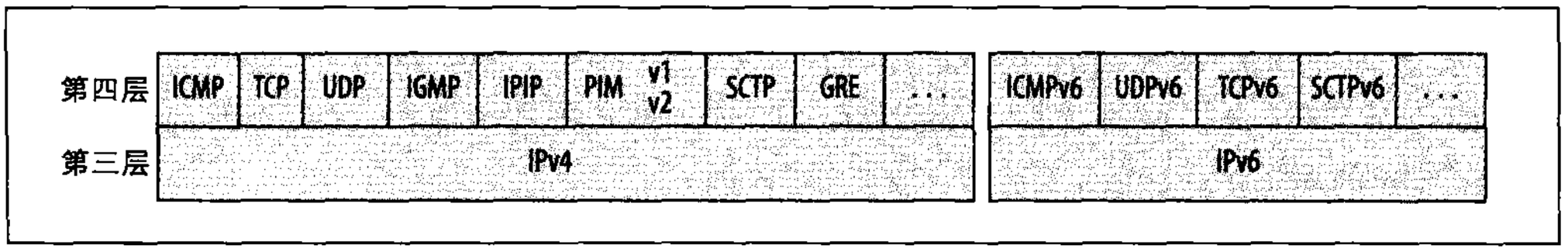


图 24-1：Linux 内核所实现的 L4 协议（位于 IPv4 和 IPv6 之上）

注意，表 24-2 的最后四个项目是隧道协议。它的 ID 指出它是 L3 协议。例如，IPIP 协议是用于传输 IPv4 数据报内的 IPv4 数据报。注意，封装一个 IP 数据报时，指派给 IPv4 报头的“协议”字段的值以及当 Ethernet 有效载荷为 IP 数据报时用于对 Ethernet 报头的“协议”字段做初始化的值并没有关联。即使两个字段都是指相同协议（IPv4），但却是属于两个不同范围：一个是 L3 协议标识符，而另一个是 L4 协议标识符。

L4 协议的注册

位于IPv4之上的L4协议是由net_protocol（定义在*include/net/protocol.h*中）数据结构定义，其中有下列三个字段：

`int (*handler)(struct sk_buff *skb)`

由此协议注册的函数，来作为送进来的封包的处理函数。在“L3到L4的传递：ip_local_deliver_finish”一节会进一步讨论。有可能让IPv4和IPv6共享相同处理函数（例如，SCTP）。

`void (*err_handler)(struct sk_buff *skb, u32 info)`

由ICMP协议处理函数所用的函数，用于通知L4协议有关接收到ICMP UNREACHABLE消息的事情。到第三十五章就会知道，Linux系统何时产生ICMP UNREACHABLE消息，而到第二十五章也会知道，ICMP协议如何使用err_handler。

`int no_policy`

此字段在网络协议栈中的某些关键点都会被查询，用于使协议免于IPsec策略检查：1是指没有必要针对此协议检查IPsec策略。不要把net_protocol结构的no_policy字段和ipv4_devconf结构中拥有相同名称的字段搞混了：前者是用于协议身上，而后者是用于设备身上。参见“L3到L4的传递：ip_local_deliver_finish”及“IPsec”两节了解no_policy的用法。

*include/linux/in.h*文件中包含一份L4协议列表，定义成一系列IPPROTO_*XXX*符号（有关的完整列表，可参考*/etc/protocols*文件，或者RFC 1700以及后继的RFC）。L4协议标识符的最大值是$2^8 - 1$或255，因为该字段（位于IP报头里）指定L4协议的大小是8位。最高号码255保留给Raw IP（IPPROTO_RAW）

那些符号所定义的协议并非全都是在内核层次处理的，有些通常是在用户空间中处理（例如，RSVP及各种路由协议）。这就是为什么RSVP及路由协议（如OSPF）没有包含在前节所示内核所支持的L4协议列表中的原因。

注册：inet_add_protocol和inet_del_protocol

协议会以inet_add_protocol函数自行注册，此外，实现成模块时，则是以inet_del_protocol自行除名。这两个函数都定义在*net/ipv4/protocol.c*中。

所有和内核注册的L4协议的inet_protocol结构都会插入到一张名为inet_protos的表内，如图24-2所示。在内核早期版本中，这是一张hash表，而“hash”这个字依

然出现在处理该张表的程序中，但是，现在只是一个寻常的数组；而可能的256种协议中的每一种都有一个项目。来自于*/etc/protocols*的协议号码就是表中协议所对应的位置。如果你想知道2.4版内核中这张表如何当成hash表处理，可以看一看2.4版源代码中的`ip_run_ipprot`函数。图24-2是最常见协议的号码和几个起首字母。例如，ICMP是协议1，占着`inet_protos`表中的位置1。

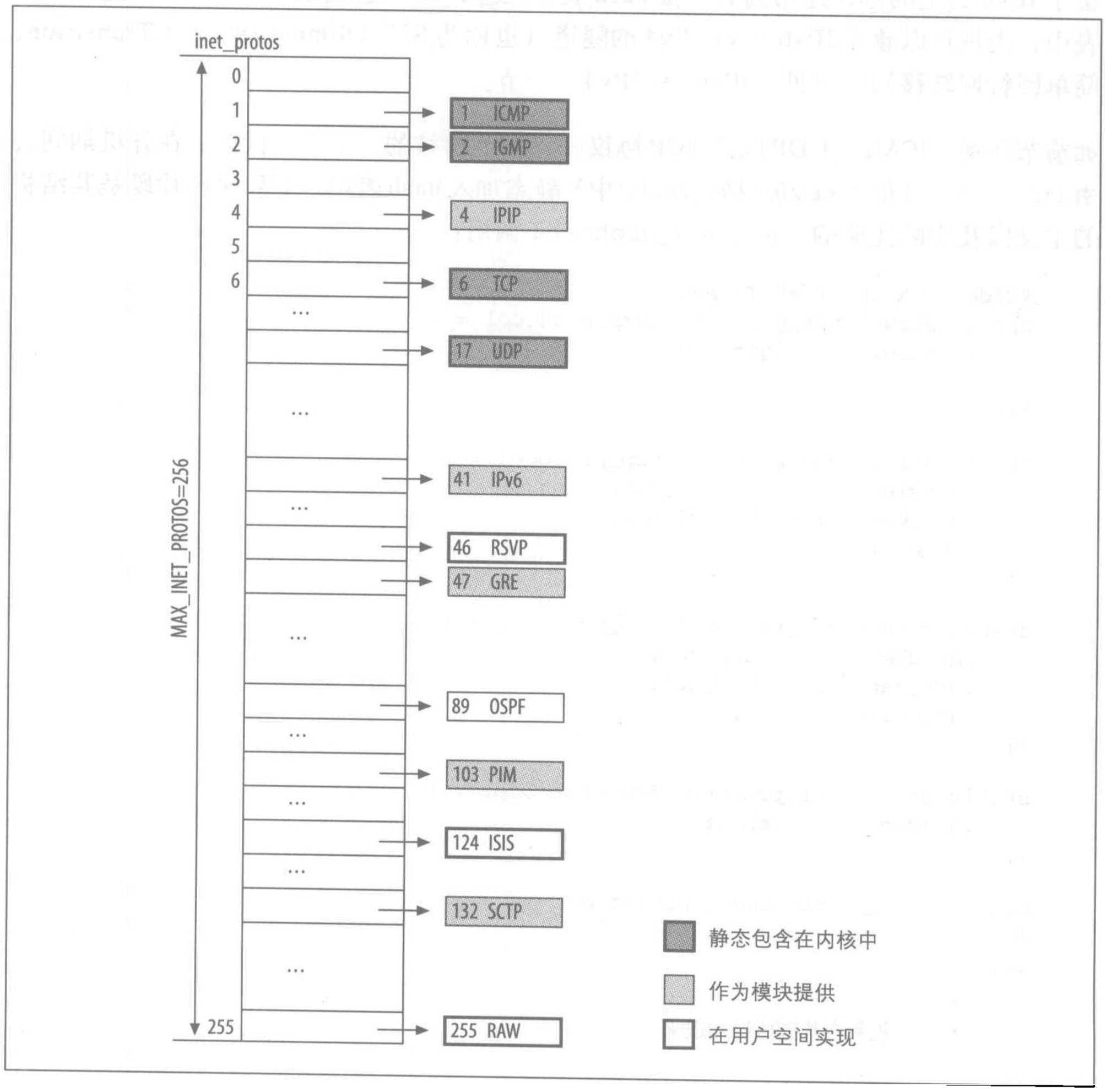


图24-2：IPv4协议表

对`inet_protos`表的同时存取是以下列方式管理的：

- 读写存取（如`inet_add_protocol`和`inet_del_protocol`）是用`inet_proto_lock`回转锁来串行化处理。

- 只读存取（如ip_local_deliver_finish，参见下一节）是以rcu_read_lock/rcu_read_unlock保护。

inet_del_protocol（删除当前由一个RCU读取者持有的表的一个项目）在返回前会调用synchronize_net来等待当前所有正在执行的RCU读取者以完成他们的关键片段。位于IPv6之上的协议还用到另一张hash表。注意，IPv6也出现在IPv4 inet_protos表中：内核可以建立IPv6-over-IPv4的隧道（也称为SIT（Simple Internet Transition，简单因特网转移））。参见“IPv6 vs. IPv4”一节。

如前节所述，ICMP、UDP以及TCP协议一直都是内核的一部分，因此，在开机期间会由inet_init（位于*net/ipv4/af_inet.c*中）静态加入hash表中。下列程序片段是其结构的定义以及实际注册的inet_add_protocol调用：

```
#ifdef CONFIG_IP_MULTICAST
static struct net_protocol igmp_protocol = {
    .handler =    igmp_rcv,
};
#endif

static struct net_protocol tcp_protocol = {
    .handler =      tcp_v4_rcv,
    .err_handler =  tcp_v4_err,
    .no_policy =    1,
};

static struct net_protocol udp_protocol = {
    .handler =      udp_rcv,
    .err_handler =  udp_err,
    .no_policy =    1,
};

static struct net_protocol icmp_protocol = {
    .handler =    icmp_rcv,
};

static int __init inet_init(void)
{
...
    /*
     *    把所有基础协议加进来
     */

    if (inet_add_protocol(&icmp_protocol, IPPROTO_ICMP) < 0)
        printk(KERN_CRIT "inet_init: Cannot add ICMP protocol\n");
    if (inet_add_protocol(&udp_protocol, IPPROTO_UDP) < 0)
        printk(KERN_CRIT "inet_init: Cannot add UDP protocol\n");
    if (inet_add_protocol(&tcp_protocol, IPPROTO_TCP) < 0)
        printk(KERN_CRIT "inet_init: Cannot add TCP protocol\n");
#ifdef CONFIG_IP_MULTICAST
```

```
    if (inet_add_protocol(&igmp_protocol, IPPROTO_IGMP) < 0)
        printk(KERN_CRIT "inet_init: Cannot add IGMP protocol\n");
#endif
...
}
```

只有当内核编译成支持 IP 多播时，IGMP 处理函数才会注册。

下列程序片段取自Zebra用户空间路由常驻程序所实现的OSPFIGP（Open Shortest Path First IGP）协议，用于示范其他协议如何动态注册。这个程序来自于 Zebra 套件中的 *ospfd/ospf_network.c* 文件。socket 调用有效的把用户空间常驻程序注册进内核，使得内核有地方可以传送给使用该协议的入口封包（在第三自变量指定）。该协议为 IPPROTO_OSPFIGP，而该符号相当于89，也就是 */etc/protocols* 中指派给OSPFIGP的号码。另外，注意，套接字类型为 SOCK_RAW，这是因为封包为私有格式，而 OSPFIGP 协议知道该怎么处理。稍后“Raw 套接字和 Raw IP”一节会说明 Raw 套接字的使用。

```
int
ospf_serv_sock (struct interface *ifp, int family)
{
  int ospf_sock;
  int ret, tos;
  struct ospf_interface *oi;

  ospf_sock = socket (family, SOCK_RAW, IPPROTO_OSPFIGP);
  if (ospf_sock < 0)
  {
    zlog_warn ("ospf_serv_sock: socket: %s", strerror (errno));
    return ospf_sock;
  }
  ...
}
```

对于每个 L4 协议而言，内核空间中只能有一个处理函数（但是，用户空间中可以存在多个处理函数，如稍后“Raw 套接字和 Raw IP”一节所述）。当 inet_add_protocol 被调用来为已有处理函数的 L4 协议安装处理函数时，就会返回错误（返回 −1）。

L3 到 L4 的传递：ip_local_deliver_finish

ip_local_deliver_finish（定义在 *net/ipv4/ip_input.c* 中，第二十章有简要叙述）的主要工作是根据输入IP封包报头的“协议”字段找出正确的协议处理函数，然后把该封包交给该处理函数。同时，ip_local_deliver_finish 必须处理 Raw IP，此外，如果有配置安全策略，该函数也要施加安全检查。后两项任务会在稍后节次中说明。

当然，多数封包都会配一种 L4 协议。ip_local_deliver_finish 会从 IP 报头的 8 位字段中取出该协议的号码，如图 24-3 的深色部分所示。如果 inet_protos 表不含该号

码的处理函数（也就是说内核接收到的封包所需的L4协议本身并未采用前节所述的方式注册），而且也没有raw套接字对该封包感兴趣，则该封包会被丢弃，而一条ICMP无法抵达消息就会传回给发送者。

然而，除了由内核处理外，应用程序也可以处理封包。这种处理方式可以是代替内核处理，或者是内核处理过后再进一步处理。因此，无论是否有注册内核处理函数，ip_local_deliver_finish函数总是会检查是否有应用程序已设立一个raw套接字要处理该协议，而且如果有的话，就会拷贝该封包的副本，将它交给应用程序；如图24-4所示。最后，无论封包是由已注册L4协议或Raw IP处理，其他协议可能还是得被启用，例如IPsec集组里的那些协议。

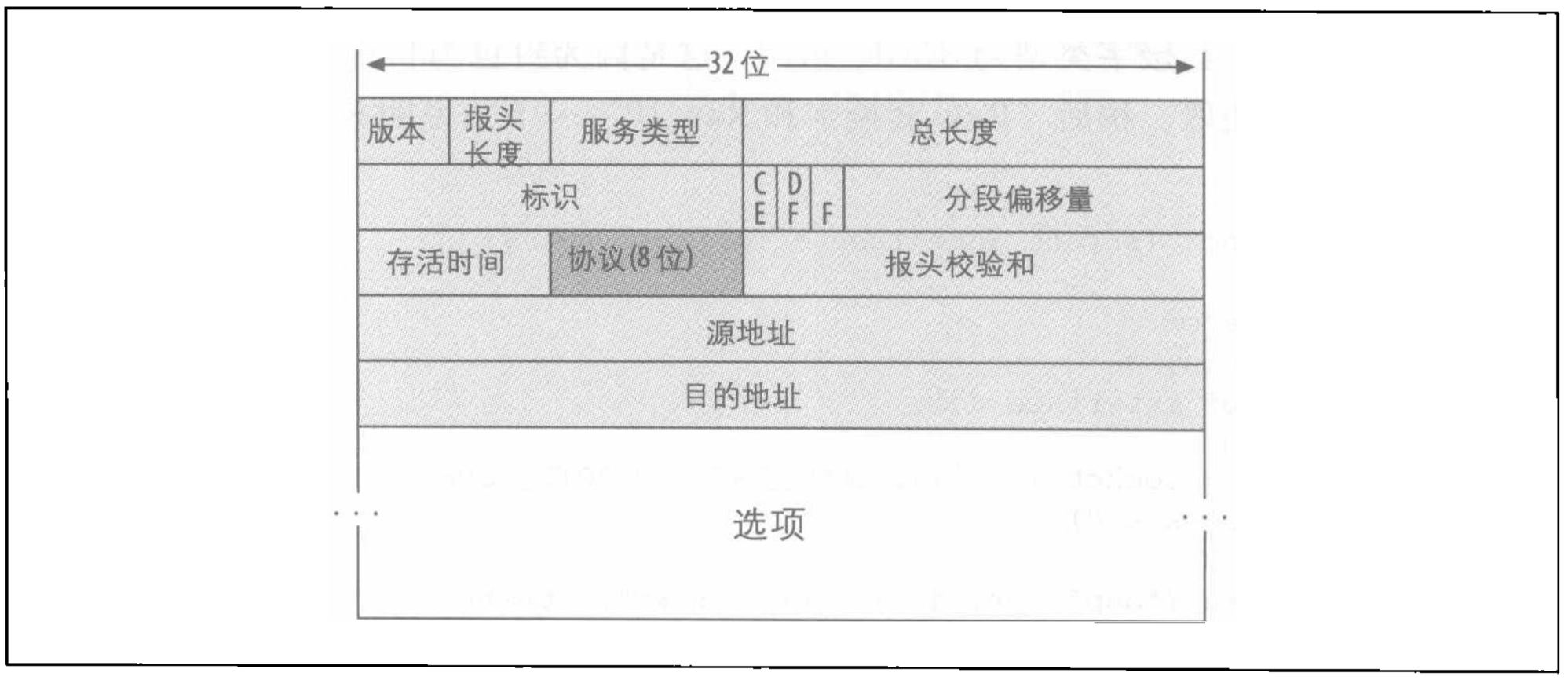


图24-3：IPv4报头和“协议标识符”字段

图24-4是此函数的基本运算。

该函数的开头如下所示：

```
static inline int ip_local_deliver_finish(struct sk_buff *skb)
{
    int ihl = skb->nh.iph->ihl*4;
    __skb_pull(skb, ihl);
    skb->h.raw = skb->data;
```

skb->h是在netif_receive_skb中初始化（如第十章所述），来指向IP报头的开端。此时，内核不再需要IP报头了，因为IP层的事已经做完，而且封包也要传给下一个较高层了。因此，这里所示的__skb_pull调用会把封包的数据部分缩小来忽略L3报头，而后续的叙述则是更新skb里的指针为skb->data指针的值，也就是指向L4报头的开端。

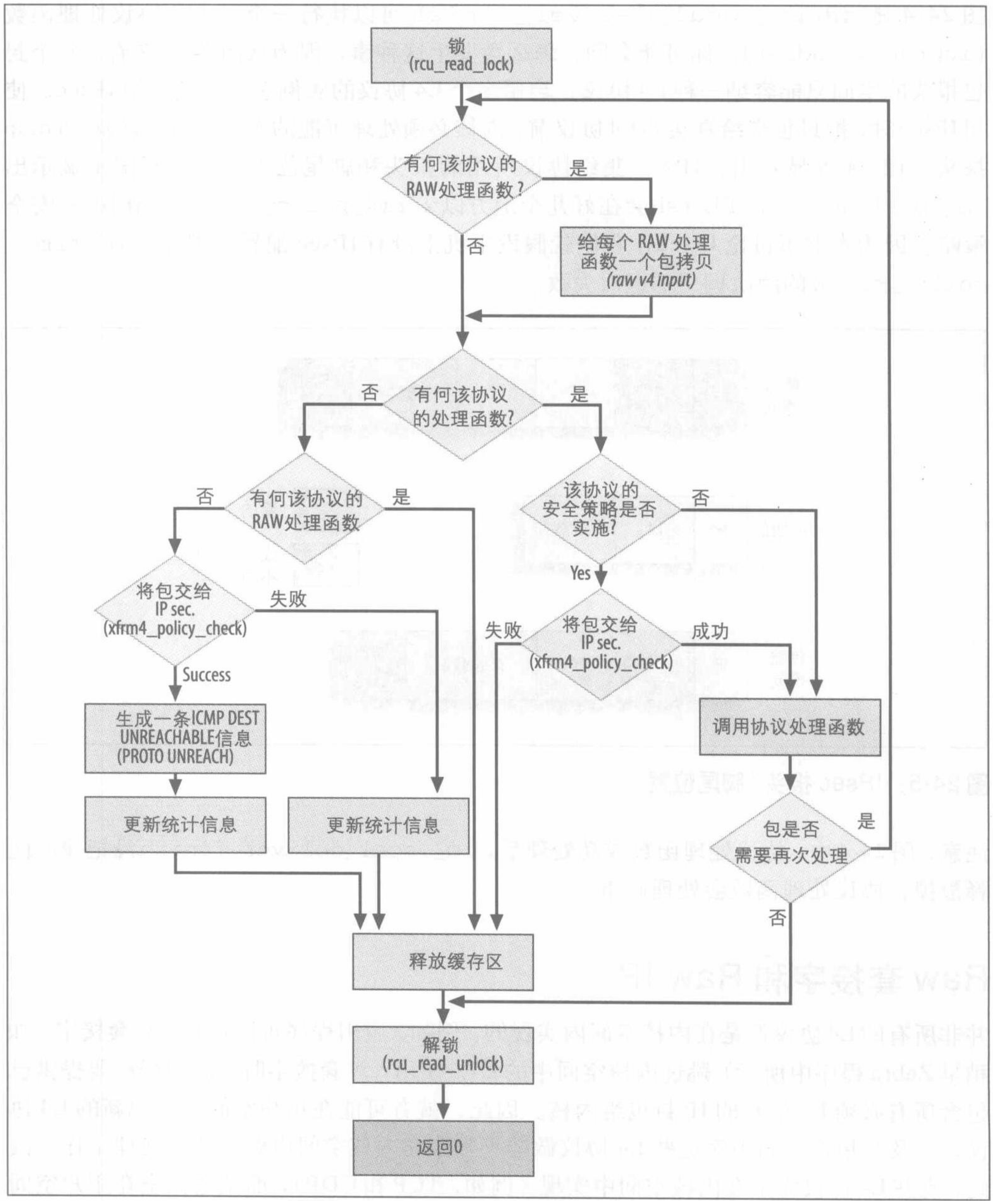


图 24.4：ip_local_deliver_finish 函数

协议 ID 是从 `skb->nh.iph->protocol` 变量（指向 IP 报头的“协议”字段，图 24-3 的深色部分）取出的。

图 24-4 显示出 ip_local_deliver_finish 可以执行一个以上的协议处理函数(ipprot->handler)。你可能会问，怎么会发生这种事，因为从图 24-3 来看，每个封包报头的空间只能容纳一种 L4 协议。启用多个 L4 协议的实例之一就是使用 IPsec。使用IPsec时，把封包交给真实的L4协议前，内核必须处理可能的AH、ESP、以及IPcomp报头。图 24-5 显示出，IPsec 集组协议所用的报头和脚尾位于何处。此图也显示出ip_local_deliver_finish会在好几个地方以xfrm4_policy_check检查IPsec安全策略。因为本书不讨论 IPsec，我们就假设主机上没有 IPsec 配置，因此，对 xfrm4_policy_check 的两次调用都返回失败。

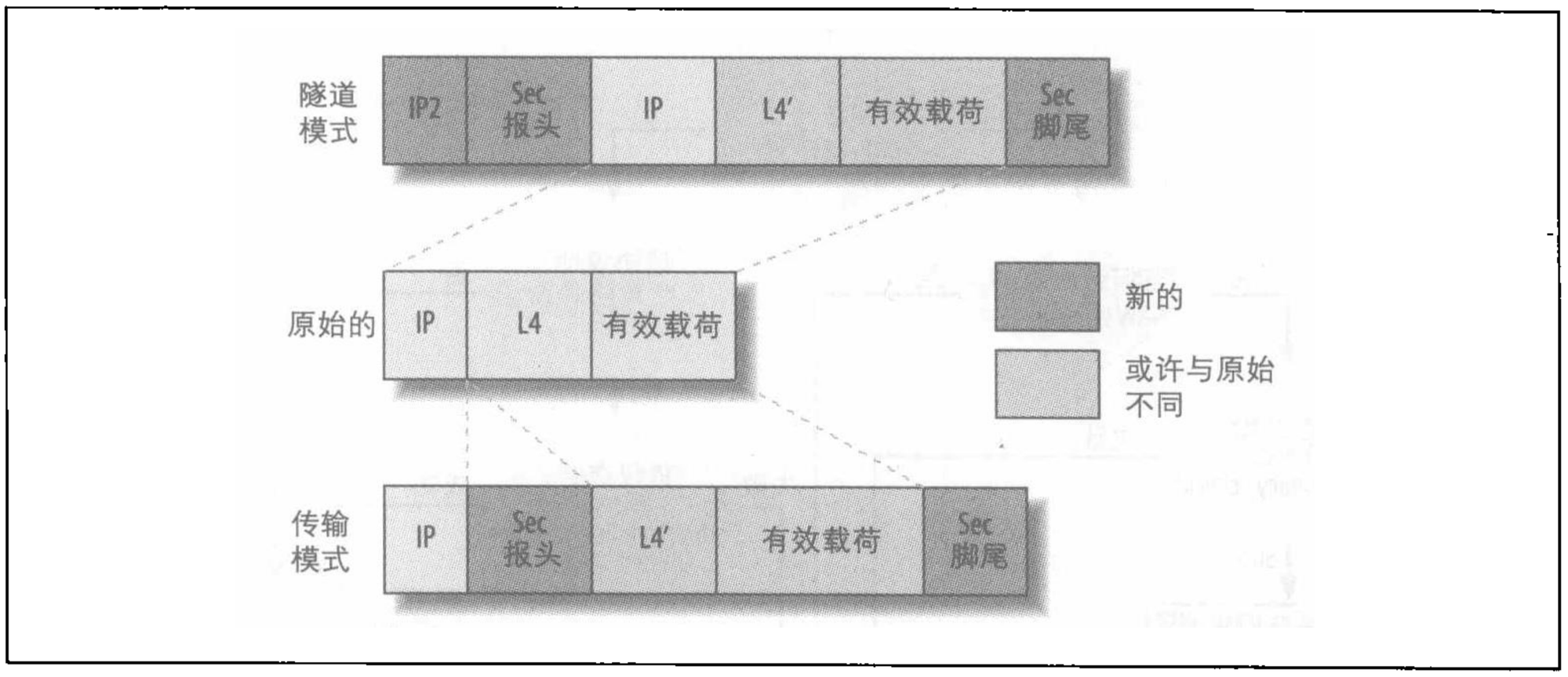


图 24-5：IPsec 报头 / 脚尾位置

注意，图 24-4 中，协议处理函数成功处理后，ip_local_deliver_finish没把缓冲区释放掉：协议处理函数会处理此事。

Raw 套接字和 Raw IP

并非所有的L4协议都是在内核空间内实现的。例如，应用程序可以使用raw套接字（如稍早Zebra程序中所示）跳过内核空间中的L4。使用raw套接字时，应用程序要提供已包含所有必须 L4 信息的 IP 封包给内核。因此，就有可能在用户空间中实现新的 L4 协议，以及在用户空间中对这些 L4 协议做些平常会在内核空间中做的其他处理工作。因此，有些L4协议完全在内核空间中实现（例如，TCP和UDP），而某些完全在用户空间实现（例如OSPF），此外，有些则是部分在内核空间中实现，而部分在用户空间中实现（如ICMP）。图 24-6（a）、图 24-6（b）、图 24-6（c）显示了这三种情况，而图 24-6（d）则是图 24-6（b）的特殊情况。以下是对此图做的说明：

- (a) 网页浏览器和远程网站服务器通信。就此而言，通信是通过一个或多个TCP套接字完成的。TCP 是在内核空间中实现的：浏览器和网站服务器只传递 TCP 有效载荷给内核，而内核则会处理 TCP 和 IP 报头。
- (b) 两台执行OSPF常驻程序的路由器彼此对话。OSPF协议是在用户空间实现的，而且会把L4 报头传给内核（注 1）。这是使用 raw 套接字的实例之一。参见第十三章来了解有关 raw 套接字和协议栈如何搭配。
- (c) 一台主机 ping 另一台主机。请求组件是在用户空间中实现，回复组件是在内核空间中实现。
- (d) 一台执行 *traceroute* 的主机在执行网络除错工作。L3 和L4 报头由应用程序处理。应用程序将它的 L4 协议指定为 RAW IP，然后对套接字设定 IP_HDRINCL 选项（包含报头）（注 2）。参见第二十一章有关 IP 如何处理 Raw IP。

当来自于用户空间方块的箭头跳过内核空间中的“L4”方块时，就是指这是 raw 传输。

ICMP是部分在用户空间实现而部分在内核空间实现的协议范例。当你ping一台主机时，*ping* 应用程序会产生 ICMP 封包，然后将它当作 IP 封包传给内核。内核不会去碰 ICMP 报头。然而，进行接收的主机会在内核空间中处理 ICMP_ECHO_REQUEST，从而回复一条 ICMP_ECHO_REPLY 消息。

把 Raw 输入数据段传递给进行接收的应用程序

学习程序语言时，你可能已看过socket调用。我们这里再提出来是为了显示出它和raw协议间的关系。当应用程序打开一个套接字时，调用时必须指定家族、套接字类型以及协议标识符。套接字和协议的类型都可以是 raw。我们来看这两者间的关系。以下是 socket 系统调用的原型：

```
socket(int family, int type, int protocol)
```

family是地址家族；允许值 AF_*XXX* 列在 *include/linux/socket.h*（TCP/IP 所用之值为 AF_INET）中。type 是套接字类型，允许值 SOCK_*XXX* 列在 *include/linux/net.h* 中。protocol是L4协议标识符，IP协议的允许值 IPPROTO_*XXX* 列在 *include/linux/in.h* 中。

当你打开一个类型为 SOCK_RAW 的套接字（所选协议号码为整数 P），则你的应用程序会接获所有满足下列规则的入口封包：

注 1： 多数 OSPF 实现也会传递 IP 报头。参见图 24-6 (d)。

注 2： IP封面是由dst_output函数传输（第二十一章说明）。必要时，dst_output会处理L3对 L2 的地址对应；因此，(d) 并非是直接调用 L2 的情况。

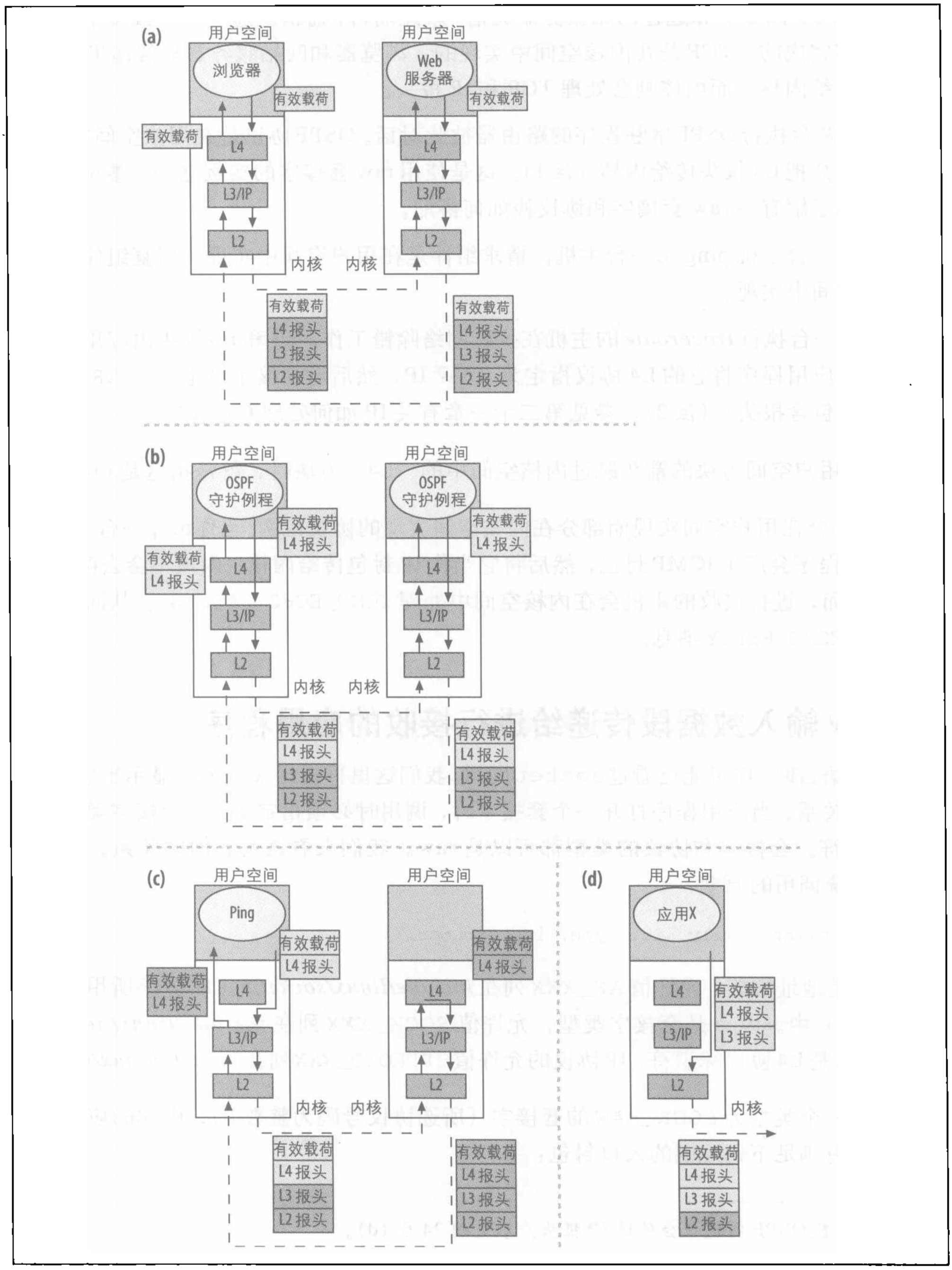


图 24-6：协议的实现（内核 vs 用户空间）

- IP 报头中的 L4 协议标识符为 *P*。
- 当套接字绑定至目的 IP 地址时，封包中源地 IP 地址必须吻合。
- 当套接字绑定至本地 IP 地址时，封包中目的 IP 地址必须吻合。

不是只有一个套接字可以满足这些规则，所以，一个 Raw IP 封包可以传递给多个应用程序。例如，想一想从两个不同终端机 ping 相同远程 IP 地址，如图 24-7 所示。

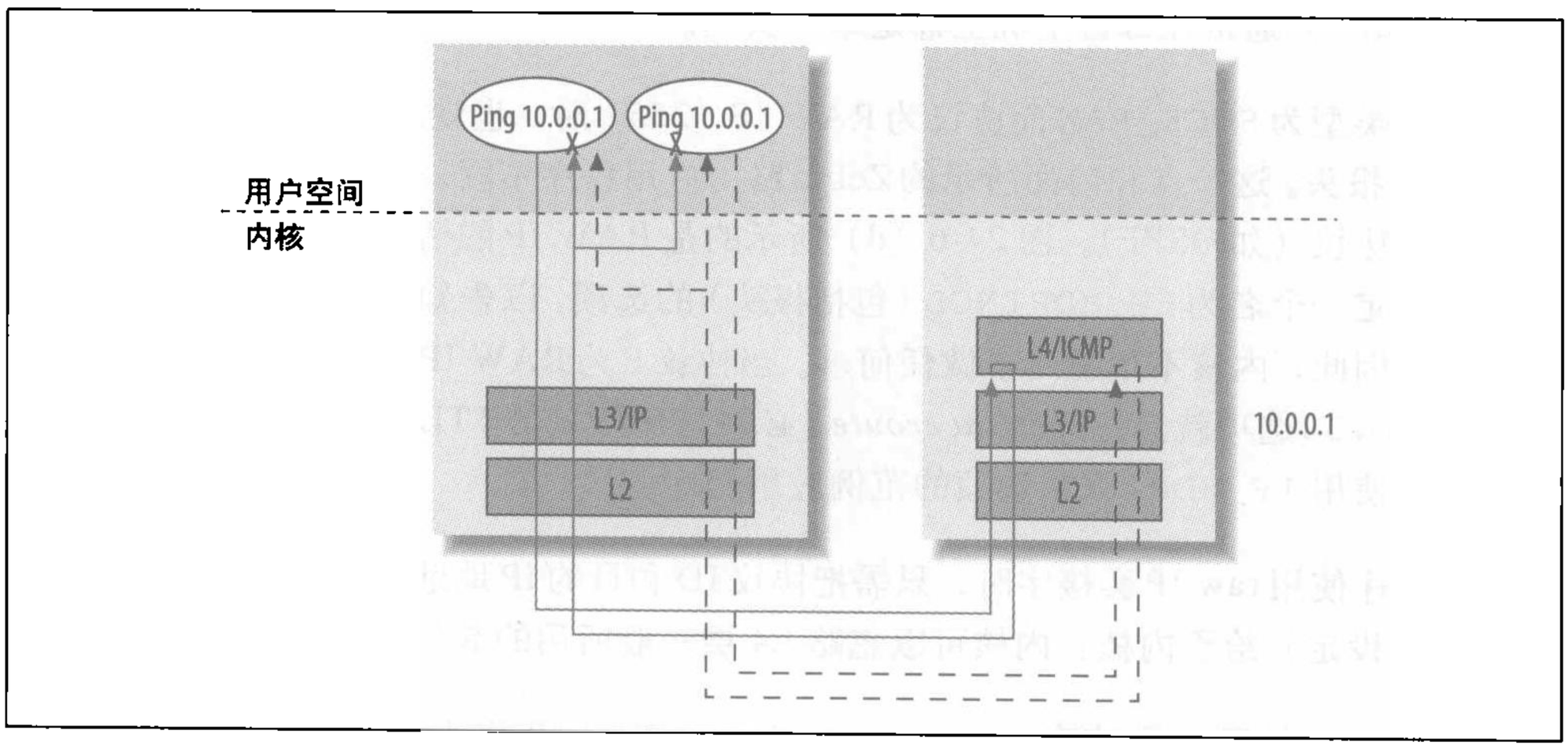


图 24-7：同时 ping 相同的目的 IP 地址

这两个 *ping* 实例要怎么区分回复数据，才不会被另一个实例的流量给搞混了？L4 协议必须在它的报头或有效载荷中包含足以区分应用程序的信息。例如，由 *ping* 命令送出的 ICMP ECHO REQUEST 消息会使其 ICMP 报头的 `identifier` 字段设成发送者的进程 ID（pid）。这个字段就可以让 *ping* 应用程序认出将会由接收者所送回的输入 ECHO REPLY ICMP 消息。ICMP 报头的“序列号码”字段会设成一个计数器（每次传输后，*ping* 都会递增它）。此计数器可让 *ping* 比对入口 ICMP ECHO REQUEST 消息及其相配的 ICMP ECHO REPLY 消息。在下例中，该计数器是输出成 `icmp_seq` 字段。

```
# ping www.oreilly.com
PING www.oreilly.com (208.201.239.36) 56(84) bytes of data
64 bytes from www.oreillynet.com (208.201.239.36): icmp_seq=0 ttl=50 time=245 ms
64 bytes from www.oreillynet.com (208.201.239.36): icmp_seq=1 ttl=50 time=244 ms
...
```

有关 ICMP 的细节，可以参考第二十五章。注意，把封包传递给数个应用程序会涉及额外的开销，而且应用程序必须把不想要的封包筛掉，而不是让内核通过端口做筛选。因为有此开销，新协议如果需要繁重的多路复用 / 多路复用分解时，通常就不会在用户空间中使用 Raw IP 实现。

简而言之，每次内核收到一个包含内核无法处理的L4协议的封包时，所有为该协议所注册的套接字都会接收到一份该封包副本。至于要接收或丢弃该封包，就由那些套接字决定了。也就是说，应用程序必须有种方式了解它所接收的封包是否就是要传给自己的，而这种任务没必要由端口系统在TCP和UDP中提供。

Raw IP适用于*ping*，因为同一台机器有可能一次执行好几个*ping*实例，而且通常不是针对同一个目的地IP地址，此外，每个实例通常只传送几个封包而已。同样的，路由协议（如OSPF）通常在每台主机上都是单一实例。

当套接字类型为SOCK_RAW而协议为RAW IP（255）时，也就是说应用程序要处理L4报头和IP报头。这一点和前面所示的Zebra路由应用程序不同，因为协议是RAWIP，不再是著名协议（如OSPF）。图24-6（d）所示的是RAW IP的情况。这类应用程序会对套接字设定一个名为IP_HDRINCL（包括报头）的选项，以告知内核该应用程序会处理IP报头，因此，内核不用就此事做任何事。当协议P为RAW IP时，默认的套接字上的IP_HDRINCL选项就会打开。*traceroute*（必须与IP报头的TTL字段一同才能完成其工作）就是使用IP_HDRINCL选项的范例应用程序。

当应用程序使用raw IP套接字时，只需把协议ID和目的IP地址（将会在内核所产生的IP报头上设定）给予内核：内核可以忽略L4层一般所用的其他所有参数和选项。

用于存储raw处理函数的表（raw_v4_htable）及用于存储协议处理函数的表（inet_protos）具有相同尺寸，所以，ip_local_deliver_finish使用相同的hash值来存取这两张表（如前所述，该值不再是一个实际hash）。Raw封包会传给raw_v4_input。该函数不会直接操作输入缓冲区，因为该封包属于调用者（ip_local_deliver_finish），而且可能被很多应用程序共享。因此，raw_v4_input会做本地拷贝（克隆），然后把它们交给主要处理函数raw_rcv。

IPsec

ip_local_deliver_finish把封包传递给正确协议处理函数前，必须先以IPsec检查该封包是否准许处理。当内核必须产生ICMP错误消息时（由于内核中缺乏正确协议处理函数），也会做相同的事情。IPsec包含安全策略数据库，分成入口和出口策略。因为ip_local_deliver_finish会处理送进来的流量，而IPsec函数xfrm4_policy_check启用时会带有方向标志XFRM_POLICY_IN。如果封包可准许处理，此函数的返回值为1，若不准处理，则为零。当传输协议的net_protocol实例的no_policy设为1时，安全策略就不会被查询。

因篇幅有限，IPsec集组的协议的实现不在本书讨论之列。

IPv4 vs IPv6

就L3对L4协议的接口而言，IPv6和Pv4很相似。L4协议可通过inet6_add_protocol注册，以及通过inet6_del_protocol除名（这两个函数都定义在*net/ipv6/protocol.c*中）。处理函数存储在一张名为inet6_protos的表内；表的尺寸和IPv4所用的相同（MAX_INET_PROTOS）。在IPv6上头执行的L4协议是由inet6_protocol数据结构（定义在*include/net/protocol.h*中）表示，而它的定义几乎和IPv4所用的结构相同。唯一的差别是处理函数的原型及err_handler函数指针，另外，使用标志（而非整数）存储安全策略是否存在的信息。

IPv6报头中，IPv6用于识别上层协议的字段叫做next_header（下一个报头），而此字段是8位值，类似于IPv4所用的字段。参见图24-8了解此字段在报头中的位置。

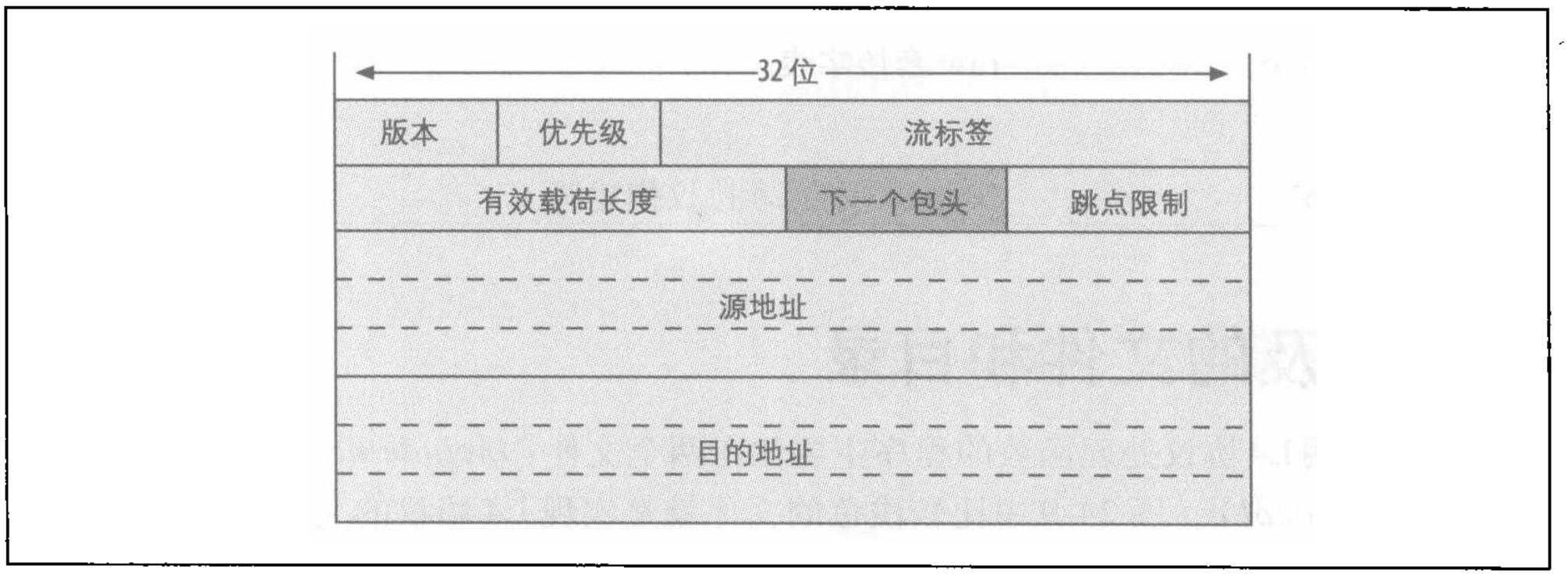


图 24-8：IPv6 报头和 next_header 协议标识符

通过/proc 文件系统做调整

/proc 里没有文件可用于调整L3和L4间的接口。

本章涉及的函数和变量

表24-3总结本章所介绍的函数、变量以及数据结构。

表 24.3：本章涉及的函数、变量以及数据结构

名称	说明
函数	
inet_add_protocol inet_del_protocol	为 IP 协议栈注册和除名 L4 协议处理函数
inet_init	AF_INET 协议家族的初始化函数。最常用的 L4 协议都在此注册
ip_local_deliver_finish raw_v4_input	ip_local_deliver_finish会把入口IP流量传递给正确的L4协议处理函数，而且会使用raw_v4_input把副本交给适合的 RAW IP 套接字
变量	
inet_protos	IP 协议栈的 L4 协议处理函数表
raw_v4_htable	raw 套接字表
数据结构	
net_protocol	IP 协议栈的 L4 协议描述符

本章涉及的文件和目录

内核用于调用L4协议处理函数的程序主要位在两个文件：*include/net/protocol.h*和*net/ipv{4,6}/protocol.c*。图 24-9 中比较浅色的文件就是实现 L4 协议的文件。

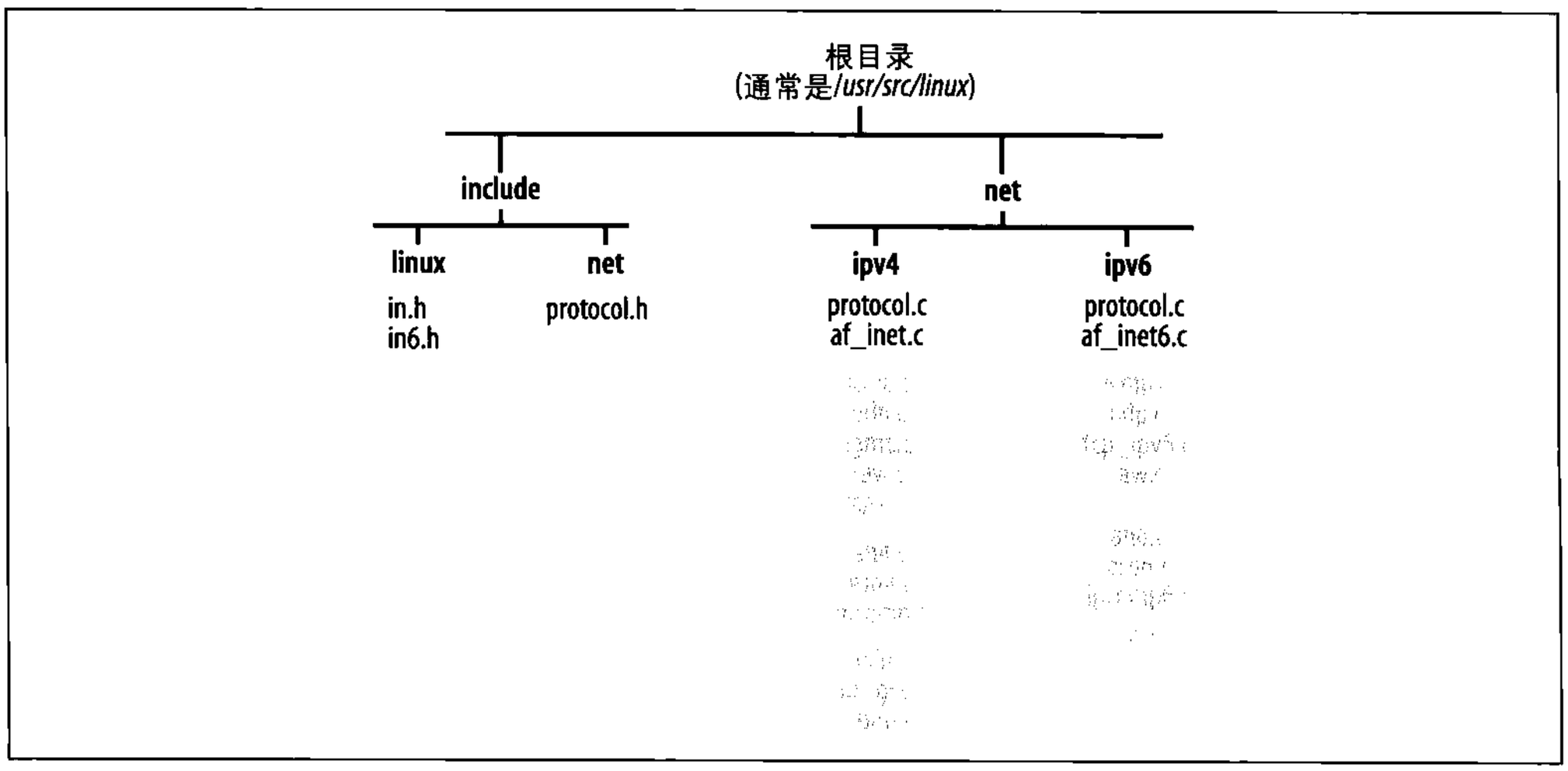


图 24-9：本章涉及的文件和目录

第二十五章

因特网控制消息协议（ICMPv4）

ICMP（Internet Control Message Protocol，因特网控制消息协议）是一种传输协议，它由因特网主机使用，用于交换控制消息（主要是错误通知及信息请求）。本章我们讨论ICMPv4，也就是IPv4所使用的版本。IPv6使用ICMPv6协议（此协议包含ICMPv4以外的其他功能）。

近些年来，ICMP协议已逐渐成为开发监控和测量应用程序的基础。不幸的是，ICMP协议也时常被用作安全攻击的基础，例如，DoS或者远程指纹收集（remote fingerprint collection）。因此，网络管理员时常把路由器和防火墙配置成过滤掉多数ICMP消息类型。偶尔，网络管理员过滤掉太多了，因而违反了RFC建议案。无论消息是否被过滤掉，消息通常有速率限制。因此，建立在ICMP之上的任何应用程序就测量和监控目的而言，不见得都可靠。然而，因为测量本来就不是原本设计目标，ICMP通常无法让监控应用程序收集到其所需的所有信息。相反的，为了收集完整信息，另外又开发了专用应用程序，而其基础通常是TCP或UDP。

读者若对ICMP安全方面的议题感兴趣，我建议你读以色列籍安全顾问Ofir Arkin所写的文章《ICMP Usage in Scanning》（*http://www.sys-security.com/archive/papers/ICMP_Scanning_v3.0.zip*）。文中说明了ICMP消息如何用于网络扫描等用途，以及为何多数ICMP消息都会被网络管理员过滤掉。文章中有包含一份有关ICMP的主要RFC文件的详尽摘要。

本章中，我们要说明Linux如何实现ICMP协议。针对每种ICMP消息类型，我们会简要说明内核何时产生消息，以及当内核接收消息时又该如何处理。针对其他细节，可以参考下列RFC文件：

- RFC 792，《Internet Control Message Protocol》
- RFC 950，《Internet Standard Subnetting Procedure, Appendix I》

- RFC 1016，《Something a Host Could Do with Source Quench》
- RFC 1191，《Path MTU Discovery》
- RFC 1122，《Requirements for Internet Hosts-Communication Layers》
- RFC 1812，《Requirements for IP Version 4 Routers》
- RFC 1256，《ICMP Router Discovery Messages》
- RFC 1349，《Type of Service in the Internet Protocol Suite》

特别的是，RFC 792 说明了多数 ICMP 类型的报头的配置，而 RFC 1122 及 1812 则说明主机和路由器是否应该产生及处理每种ICMP类型；这部分信息有些也会在本章提及。

有关 ICMP 消息详尽的 RFC 列表，可以参考此 URL：*http://www.iana.org/assignments/icmp-parameters*。

ICMP 报头

图 25-1 为 ICMP 报头的结构。

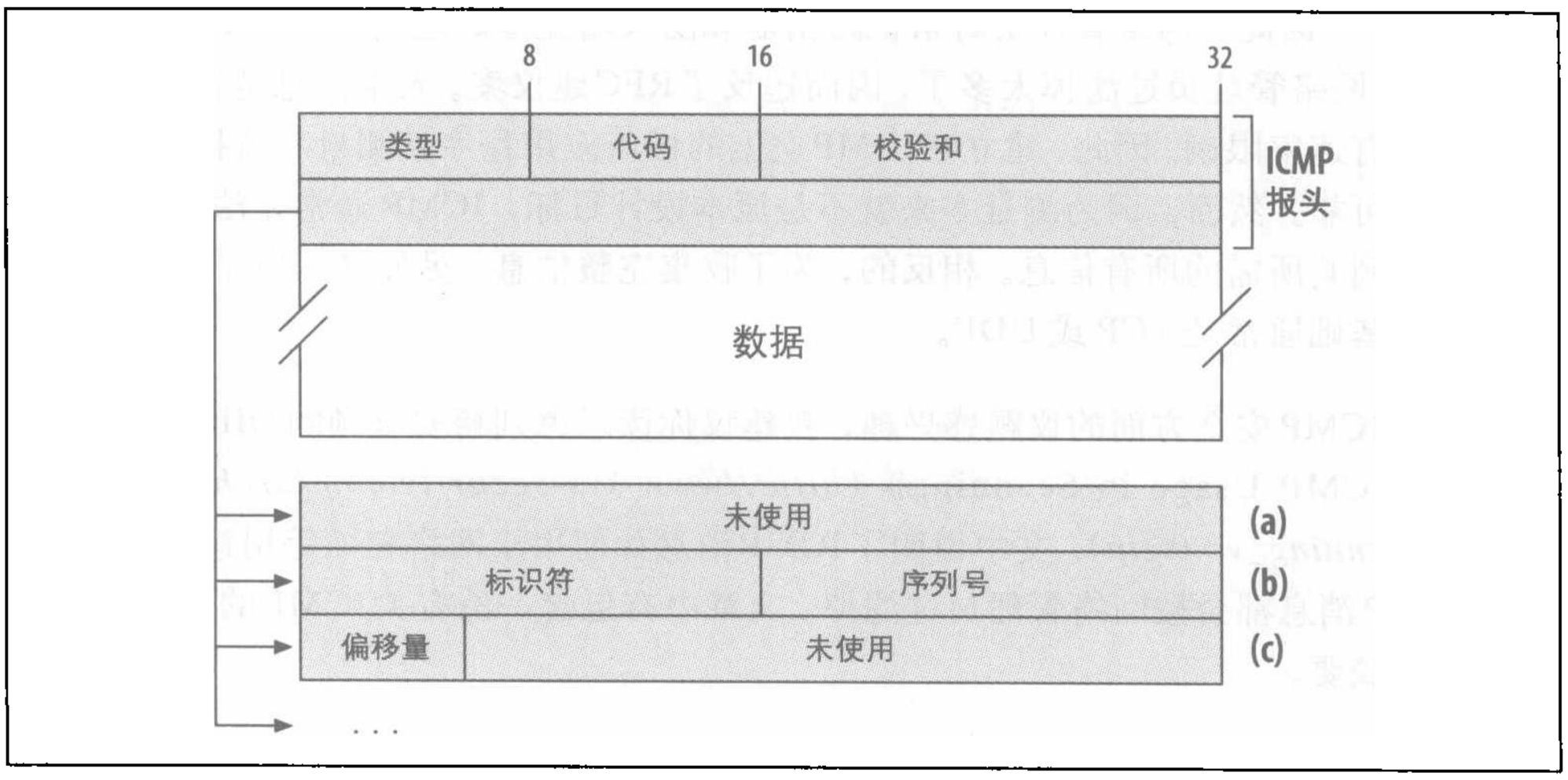


图 25-1：ICMP 报头

前三个字段对所有 ICMP 消息类型而言都相同：

类型

代码

这一组字段可以分辨出ICMP消息的类型。有些时候，类型就足以明确分辨消息，而其他时候则需要代码来区分同一种消息类型的不同变种。参见“ICMP类型”一节了解细节。

校验和

校验和包括ICMP报头和有效载荷。它所用算法和其他主要IP协议（IP、UDP、TCP以及IGMP）相同：也就是IP封包的16位字段的1的补码和。参见第十八章“校验和”一节的细节。

ICMP报头后半段的结构取决于消息类型。根据类型和代码的值，接收者可以识别出消息类型，并以此读取剩余的报头。后续的32位可以是未用、完全使用或者部分使用，这得根据消息类型而定；这三种不同配置的范例如图25-1底端所示。

ICMP消息被分成两种：错误和查询（请求/响应）。表25-1中，你可以看见各种ICMP类型属于哪一种。查询消息会使用报头的另外32位以定义两个字段：identifier（标识符）和sequence_number（序列号码）；如图25-1（b）所示。接收者不会碰这两个字段（也就是从请求消息中拷贝到响应消息中），而这两个字段可让来源地对比出响应及其原有的请求。

ICMP错误消息包含有效载荷（内容会在下一节说明）。

在RFC 792文件中，你可以找到多数ICMP消息类型的报头的配置。

ICMP有效载荷

当内核在处理入口IP封包而检测到错误时，就会传送ICMP错误消息。所有ICMP错误消息在ICMP有效载荷中都包含相同信息：触发ICMP消息传输的IP封包的IP报头，外加一部分IP有效载荷。所得IP封包必须不能超过576个字节，包括外面的IP报头以及ICMP报头［后一条规则是在RFC 1812的4.3.2.3节中陈述的（更新了RFC 792的报头的定义）。根据旧版RFC 792所述，ICMP有效载荷只需包括原有的IP报头，外加原有传输报头（64位）］。

图25-2是ICMP_FRAG_NEEDED错误消息的范例（根据RFC 792）。图25-2（a）就是触发ICMP消息传输的片段，而图25-2（b）是ICMP消息。注意，ICMP有效载荷也包括原有IP报头及一段传输报头。Linux和RFC 1812兼容，因此，包括图25-2（a）中所示的额外的区块（尺寸至多576字节）。

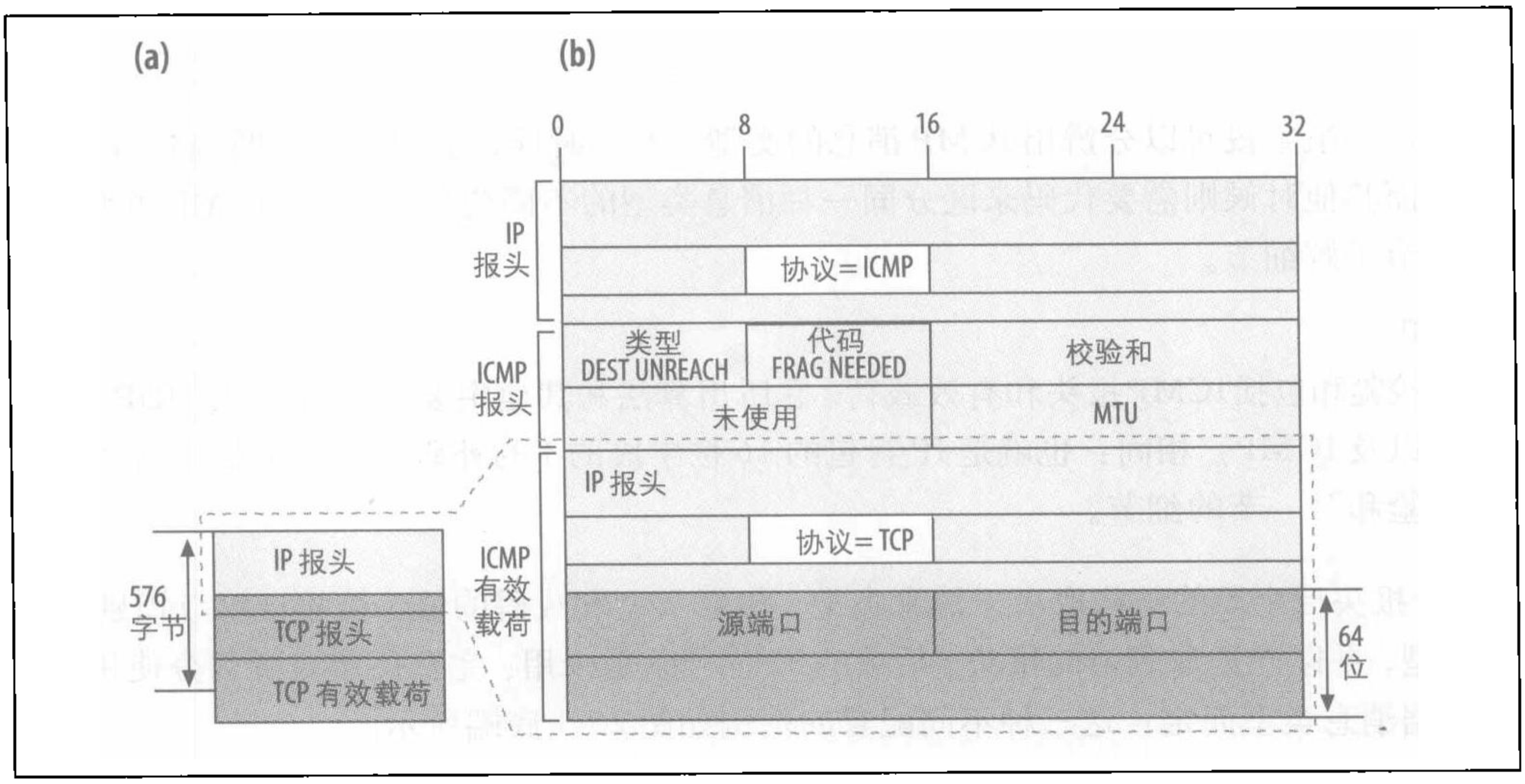


图 25-2：ICMP_DEST_UNREACH 错误消息的 ICMP 有效载荷范例

原有IP报头的“协议”字段会由ICMP消息的目的主机使用，用于分辨出正确的传输协议（此例为TCP）；此外，ICMP有效载荷中一部分的传输报头（包含源端口和目的端口号）可以让同一个目的主机用于分辨出本地套接字。因此，目的主机就可多少追查出其造成错误的原因。

ICMP 类型

表 25-1 列出 ICMP 类型及其定义所在的 RFC 文件，表明其通常是在内核空间或用户空中传输及处理，分为错误或查询消息。内核符号列在 *include/linux/icmp.h* 中。这张表只列出Linux内核关心的ICMP消息类型（无论是否实现）。你可以参考本章简介时所提到的URL以找到最新列表。

表 25-1：ICMP 类型

类型	名称	TX	RX	RFC	错误/查询
0	Echo Reply	由内核	由内核	792	查询
8	Echo Request	由用户	由内核	792	查询
1	没指定				
2	没指定				
3	Destination Unreachable	由内核	由内核	792	错误

表 25-1：ICMP 类型（续）

类型	名称	TX	RX	RFC	错误/查询
4	Source Quench （已废弃；参见 RFC 1812 第 4.3.3.3 节）				
5	Redirect	由内核	由内核	792	错误
6	Alternate Host Address（已废弃[a]）				
7	没指定				
9	Router Advertisement	由用户	由用户	1256	查询
10	Router Solicitation	由用户	由用户		
11	Time Exceeded	由内核	由内核	792	错误
12	Parameter Problem	由内核	由内核	792	错误
13	Timestamp Request	由用户	由内核	792	查询
14	Timestamp Reply	由内核	由用户	792	查询
15	Information Request				
16	Information Reply （已废弃；参见 RFC 1122 第 3.2.2.7 节以及 RFC 1812 第 4.3.3.7 节）				
17	Address Mask Request	由内核	由内核	950	查询
18	Address Mask Reply	由内核	由内核		

a：这个选项被同一位作者在 RFC 792 中定义了，但是并没有在所有 RFC 中定义。

ICMP 类型 1、2 以及 7 没指定，而类型 6 已不再在任何 RFC 定义。

类型 9 和 10 没在内核空间内处理（因此也没在内核空间中定义）；路由器发现消息是由实现 RFC 1256 的应用程序在用户空间中处理。就 Linux 而言，你可以参考 *rdisc*（此应用程序属于 *iputils* 套件）。

RFC 1122 和 RFC 1812 说明每种 ICMP 消息类型的实现是可选的或强制的（对主机和路由器而言）。表 25-2 总结这些需求。有关必须、应该以及可以这些字眼的正确理解，可以参考 RFC 2119。此表没有包含已废弃选项。

表 25-2：主机和路由器需求

类型	名称	主机（RFC 1122）	路由器（RFC 1812）	Linux 兼容
0	Echo Reply	必须实现	必须实现	是
8	Echo Request	响应服务器	响应服务器	
3	Destination Unreachable	应该传输必须接收	必须传输	是

表 25-2：主机和路由器需求（续）

类型	名称	主机（RFC 1122）	路由器（RFC 1812）	Linux 兼容
5	Redirect	不应该传输必须接收	必须传输可以接收	是
9	Router Advertisement	N/A	必须接收	否
10	Router Solicitation			（用户空间有支持）
11	Time Exceeded	必须接收	必须传输	是
12	Parameter Problem	应该传输必须接收	必须传输	是
13	Timestamp Reques	可以接收	可以接收 / 传输	是
14	Timestamp Reply	可以接收		
17	Address Mask Request	可以接收 / 传输	必须接收 / 传输	否
18	Address Mask Reply			

当主机是触发ICMP消息传输的那个IP封包的产生者时，路由器就必须尊重该主机需求。例如，Destination Unreachable ICMP 消息会传给 IP 封包无法被传递的主机。当路由器产生错误的封包时，路由器就必须根据表25-2的主机需求而处理ICMP错误消息。注意，路由器不应该接收不是它产生的IP封包所引发的Destination Unreachable消息（而这正好说明了为何表 25-2 没有指明当路由器接收此类消息时该做些什么）。

类似的说明也适用其他消息类型。

ICMP_ECHO 和 ICMP_ECHOREPLY

这些可能是最常见、最为人所知的ICMP 类型了。各种应用程序都会用到，最著名的就是 *ping*。

ICMP_ECHO消息类型是用于测试远程主机是否可连接。当主机接收到ICMP_ECHO消息时，会以 ICMP_ECHOREPLY 消息回复。参见“ping”一节。

ICMP_DEST_UNREACH

当 IP 封包无法传递到它的目的地时，或者当 IP 有效载荷无法传递到远程主机上的目的应用程序时，该ICMP 类型就可用于通知传送者传递失败及其原因。此 ICMP 类型有一些不同的子类型（代码值），全都列在表 25-3 中。并非所有子类型都由 Linux 使用。

此消息所用的报头包括那个 32 位字段，如图 25-1（a）所示。

表 25-3：ICMP 类型 ICMP_UNREACH 的 ICMP 代码

代码	内核符号	说明
0	ICMP_NET_UNREACH	网络无法连接
1	ICMP_HOST_UNREACH	主机无法连接
2	ICMP_PROT_UNREACH	协议无法连接。IP 上头所用的传输协议没在目的主机上实现
3	ICMP_PORT_UNREACH	端口无法连接。没有应用程序在监听传输报头中目的端口所指定的端口号
4	ICMP_FRAG_NEEDED	需要分段。IP 封包必须被分段，但是，IP 报头中的 DF 标志有设定
5	ICMP_SR_FAILED	源路由失败
6	ICMP_NET_UNKNOWN	目的网络未知
7	ICMP_HOST_UNKNOWN	目的主机未知
8	ICMP_HOST_ISOLATED	源主机被隔离
9	ICMP_NET_ANO	和目的网络通信遭到管理手段禁止
10	ICMP_HOST_ANO	和目的主机通信遭到管理手段禁止
11	ICMP_NET_UNR_TOS	目的网络无法连接（ToS）
12	ICMP_HOST_UNR_TOS	目的主机无法连接（ToS）
13	ICMP_PKT_FILTERED	通信遭到管理手段禁止
14	ICMP_PREC_VIOLATION	违反主机优先权
15	ICMP_PREC_CUTOFF	优先权低于最低指定值

ICMP_SOURCE_QUENCH

此消息类型最初是定义成路由器的机制，用于通知端点阻塞相关事情。然而，产生更多流量来协助阻塞恢复并不算有效，因此，RFC 1812 将此 ICMP 消息类型废弃了。

此 ICMP 类型原本的目标（阻塞控制）现在已由 RFC 3168 所描述的 ECN（Early Congestion Notification，提前阻塞通知）机制负责了。

ICMP_REDIRECT

ICMP REDIRECT 消息类型只由路由器传送，而且会由主机处理，不过路由器也可以选择处理（注 1）。Linux 在 */proc* 提供一个文件，让你可以开启和关闭 ICMP_REDIRECT

注 1：参见 RFC1812 第 4.3.3.2 和 5.2.7.2 节。

消息的处理。当路由器检测到相邻主机正在使用次佳路由时（也就是比起产生此消息的路由器而言，目的地可以通过更佳网关而抵达），就会产生此类消息。

产生ICMP_REDIRECT消息最基本而最常见的原因就是入口封包必须从其所来的同一台设备上转发出去。本节稍后会看一实例。

此ICMP消息类型有四种子类型，如表25-4所示。RFC1812指出应该只产生ICMP_REDIR_HOST和ICMP_REDIR_HOSTTOS消息，因为有些情况下，使用子网络时，会使另外两个ICMP代码难以处理。Linux遵循此建议。

表25-4：ICMP类型ICMP_REDIRECT的ICMP代码

代码	内核符号	说明
0	ICMP_REDIR_NET（废弃）	网络地址的重定向
1	ICMP_REDIR_HOST	主机地址的重定向
2	ICMP_REDIR_NETTOS（废弃）	网络地址及ToS的重定向
3	ICMP_REDIR_HOSTTOS	主机地址及ToS的重定向

图25-3提供的场景是路由器产生了ICMP_REDIRECT消息。从其拓朴结构可知，主机X应该使用路由器RT2以连接主机Y。但是，假设主机X的配置是只能使用默认网关RT1，因此，主机X传给其局域网络外的任何流量都会经过RT1。

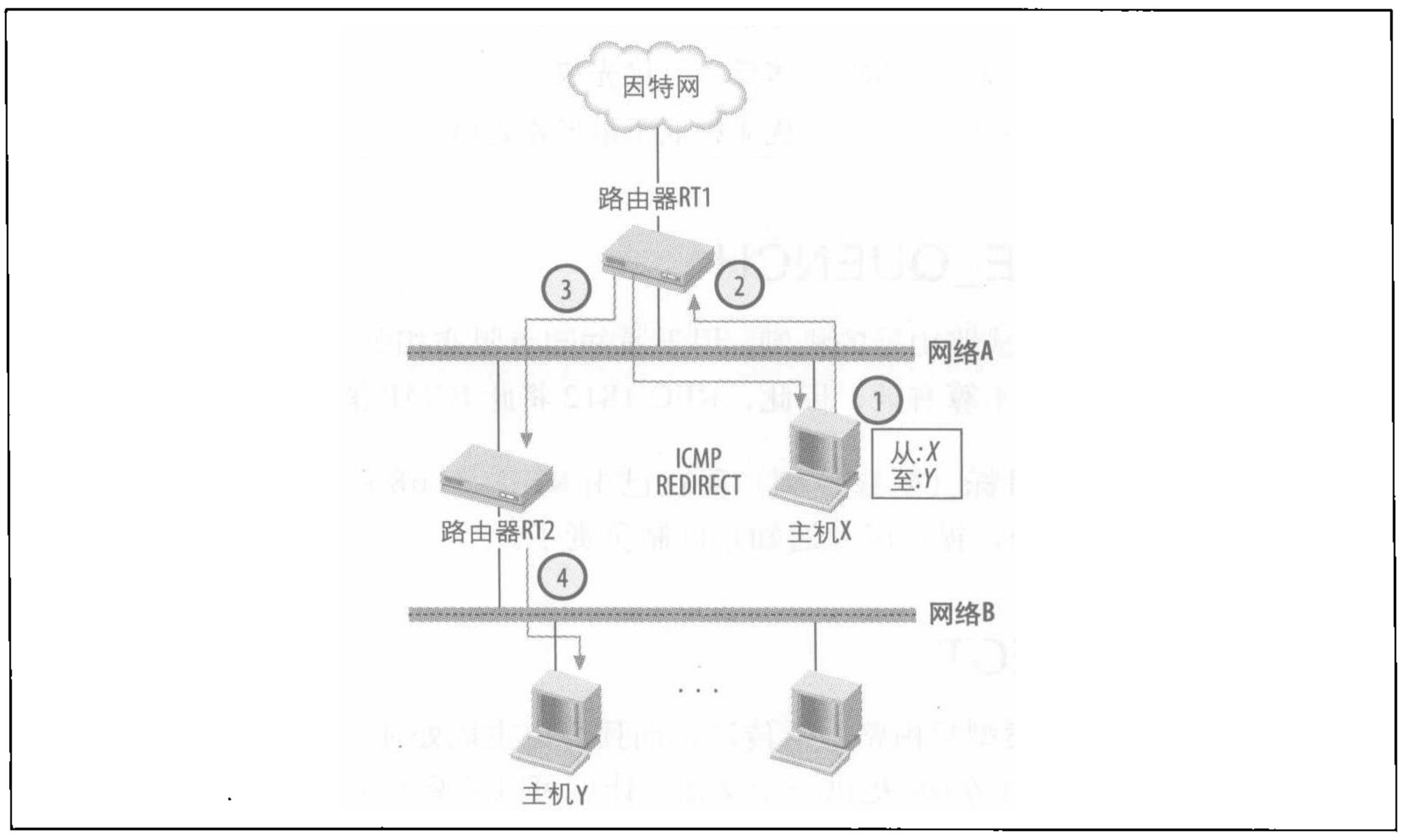


图25-3：ICMP_REDIRECT范例

当主机 X 传输 IP 封包到主机 Y 时，会发生下列事情：

1. 主机 X 传送一个要给主机 Y 的封包到路由器 RT1。
2. 路由器 RT1 会查询它的路由表，发现下一个跳点是路由器 RT2。RT1 也了解到因为路由器 RT2 和主机 X 位于相同子网上，所以主机 X 可以直接把封包传给路由器 RT2。
3. 路由器 RT1 传送一条 ICMP_REDIR ECT 消息给主机 X，通知主机 X 有更好的路径。主机 X 就会存储该路径，在下次采用。
4. 路由器 RT1 把封包转发至 RT2。

正常来讲，当路由器检测到它正被要求沿着次佳路径绕送封包时，就会以 ICMP_REDIRECT 消息回复传送者，来描述正确路径。然而，为了安全缘故，现在这类建议通常都会被回绝：你可以想象得到，要搞出问题很简单，只要说："嘿，要到那个网络，你应该使用路由器 XYZ，而不是你已经配置的那个"。

在第三十一章"发送 ICMP_REDIRECT 消息"一节中，你可以找到触发 ICMP_REDIRECT 消息传输的适当情况。此外，也可参考第二十章"ICMP 重定向"一节中此 ICMP 类型和"Source Route" IP 选项间的相互影响。在第三十一章"处理入口 ICMP_REDIRECT 消息"一节中，你可以找到是否处理入口 ICMP_REDIRECT 消息的细节。

ICMP_TIME_EXCEEDED

此消息类型有两个子类型，如表 25-5 所示。

表 25-5：ICMP 类型 ICMP_TIME_EXCEEDED 的 ICMP 代码

代码	内核符号	说明
0	ICMP_EXC_TTL	超过 TTL
1	ICMP_EXC_FRAGTIME	超过片段重组时间

IP报头包含一个TTL字段，每次经过源地和目的地间的中间跳点时，其值都会递减。如果封包抵达目的主机前，TTL变为0，则IP封包会被丢弃。丢弃该封包的中间主机会传送一条ICMP_EXEC_TTL消息给传送者，来通知传送者它的封包被丢弃了。"traceroute"一节会说明著名的命令 *traceroute* 如何使用此消息。

当 IP 封包的重组花的时间太长而夭折时，就会产生 ICMP_EXC_FRAGTIME 消息。

ICMP_PARAMETERPROB

处理入口 IP 封包的 IP 报头时，若发现问题，则检测出该问题的主机就会传送一条此类 ICMP 消息给来源地。ICMP 报头中（参见图 25-1（c））包括一个偏移量，指出在 IP 报头中的何处发现此问题。

ICMP_TIMESTAMP 和 ICMP_TIMESTAMPREPLY

ICMP_TIMESTAMP 消息类型可用于要求远程主机提供一个时间戳（实际上是两个），然后利用此时间戳使那些主机的时钟同步。接收 ICMP_TIMESTAMP 消息的主机会用 ICMP_TIMESTAMPREPLY 消息回复，它的结构如图 25-4 所示。

类型(8 位) ICMP_TIMESTAMP_REPLY	代码(8位) (未使用)	校验和(16位)
标识符		序列号
时间戳1: 由ICMP_TIMEST AMP 发送者设置		
时间戳2: ICMP_TIMESTAMP收到的时间		
时间戳3: ICMP_TIMESTAMPREPLY发送的时间		

图 25-4：ICMP_TIMESTAMPREPLY 结构

第一个时间戳是由 ICMP_TIMESTAMP 传送者做初始化，而另外两个则是由 ICMP_TIMESTAMPREPLY 传送者做初始化。第二个和第三个时间戳应该反应出接收 ICMP_TIMESTAMP 消息的时间以及相关 ICMP_TIMESTAMPREPLY 消息传输的时间。

这些 ICMP 类型用的不多，实现相同的目的采用其他协议更合适（如 NTP）。

ICMP_INFO_REQUEST 和 ICMP_INFO_REPLY

根据 RFC 1122，这两种 ICMP 消息类型已废弃，因为其他协议（例如，DHCP 及其他旧的 BOOTP 和 RARP）可做相同事情，甚至更多。

ICMP_ADDRESS 和 ICMP_ADDRESSREPLY

这些ICMP类型的目的是让主机在它所连接的网络上广播一条查询，以发现可在其接口上使用的网络掩码。接收到ICMP_ADDRESS消息的路由器会以ICMP_ADDRESSREPLY

消息回复。此回复通常是对传送者的单播消息，但是，当传送者使用的来源地IP地址为0时（也就是没配置），则会以广播方式传送。

现在，这两种消息类型的目标已由其他方式实现，例如DHCP。

Linux内核不会对入口ICMP_ADDRESS消息做回复，但是会监听入口ICMP_ADDRESS-REPLY消息，以检测出错误配置（例如，错误的网络掩码配置）。

Linux不处理ICMP_ADDRESS消息的原因之一是相同接口可以使用多个IP地址，因此，传回给任何特定请求的，也许就不是唯一的网络掩码。

根据RFC 1812，ICMP_ADDRESS和ICMP_ADDRESSREPLY的实现在路由器上是强制的，所以，Linux与此并不兼容。然而，因为这些ICMP消息类型并不常用，所以，缺少这种兼容性，对Linux而言，并不代表真的有任何兼容问题。

ICMP协议的应用程序

ICMP消息可以由内核及用户空间应用程序传输。用户空间应用程序会使用raw IP套接字界面（第十三章已介绍过）。有两个著名的网络除错工具是*traceroute*和*ping*，都是使用ICMP协议。其他使用raw IP套接字接口来进行传输或监听ICMP消息的是各种路由协议。

ping

*ping*不需要再做介绍。就多数人而言，接触网络领域时，*ping*就是第一个学会的命令。指定一个输入IP地址及一组选项，*ping*就会传输ICMP_ECHO消息到输入IP地址，然后接收到相关的ICMP_ECHOREPLY消息，显示出来回时间及其他信息。你可以在作者的网页上找到*ping*的历史（*http://ftp.arl.mil/~mike/ping.html*）。

traceroute

*traceroute*可能是继*ping*之后学会的另一个命令。*traceroute*可用于寻找发出此命令的主机与特定目的地IP地址间的路由线路。该路由线路就是一份途经路由器的IP地址列表。

*traceroute*可以使用UDP或ICMP来达到它的目标（注2）。默认情况下，*traceroute*使

注2：还有第三种选择，也就是基于IP选项的使用（RFC 1393），但是Linux不支持。最常Linux发行套件所含的*traceroute*并不支持RFC 1393。

用UDP，但是，你可以用*-I*切换选项迫使它使用ICMP。后面就会知道，UDP法也得依赖ICMP消息才能成功。这两种做法背后都有相当多的窍门。

我们来说明这种基于ICMP的技术如何运作。如第二十章所见，当入口IP封包的IP报头的TTL字段为1而且需要做转发时，接收者会丢弃该封包而传送一条类型为ICMP_TIME_EXCEEDED而代码为ICMP_EXC_TTL的ICMP消息给来源地。*traceroute*就是利用这条规则，一次找出一个居中跳点：传送ICMP_ECHO消息给目的地IP地址，而TTL字段的值会递增（从1起算），如此就可确保所有居中主机都会产生ICMP_TIME_EXCEEDED消息，而最后一个主机（也就是目的主机）则会以ICMP_ECHOREPLY消息回复。图25-5是其范例。

图中没有包含ICMP回复消息的TTL字段值，因为不同操作系统会使用不同的值（64和255是最常见者）。

基于UDP协议的技术也有点相似：依然是利用TTL字段的处理方式，只不过不是用ICMP_ECHO消息，而是使用UDP封包和很高的目的地端口号（终端主机不太可能使用）。当IP封包来到终端主机时，终端主机会返回类型为ICMP_DEST_UNREACH而代码为ICMP_PORT_UNREACH的ICMP消息。图25-6是这样的范例。

就ICMP和UDP这两种情况而言，居中主机是通过独立的“探测”封包逐一被发现出来的。这样有两种结果值得一提：

- 和居中路由器相关的来回时间会在不同时间点上反应出网络拥塞状态。因此，第n个居中路由器的来回时间通常会比第$n-1$个居中路由器的来回时间还要大，但不见得都是如此。
- 用于连接第n个跳点的那些居中路由器不见得和用于连接第$n-1$个跳点的那些居中路由器相同。通往指定目的地所选的路径牵涉到好几项不同因素，例如，动态路由变更、负载均衡器等。*traceroute*命令的源代码（你可以从最常见Linux发行套件的下载服务器那儿下载）有一些例子值得一读。

大蓝图

图25-7是一些和ICMP协议互动的内核子系统。此图只显示出TCP和UDP这两个常见的传输协议，但是，还有其他很多协议也会和ICMP互动，例如，各种隧道协议（IPIP、GRE）、IPsec集组的协议（AH、ESP、IPcomp）等。

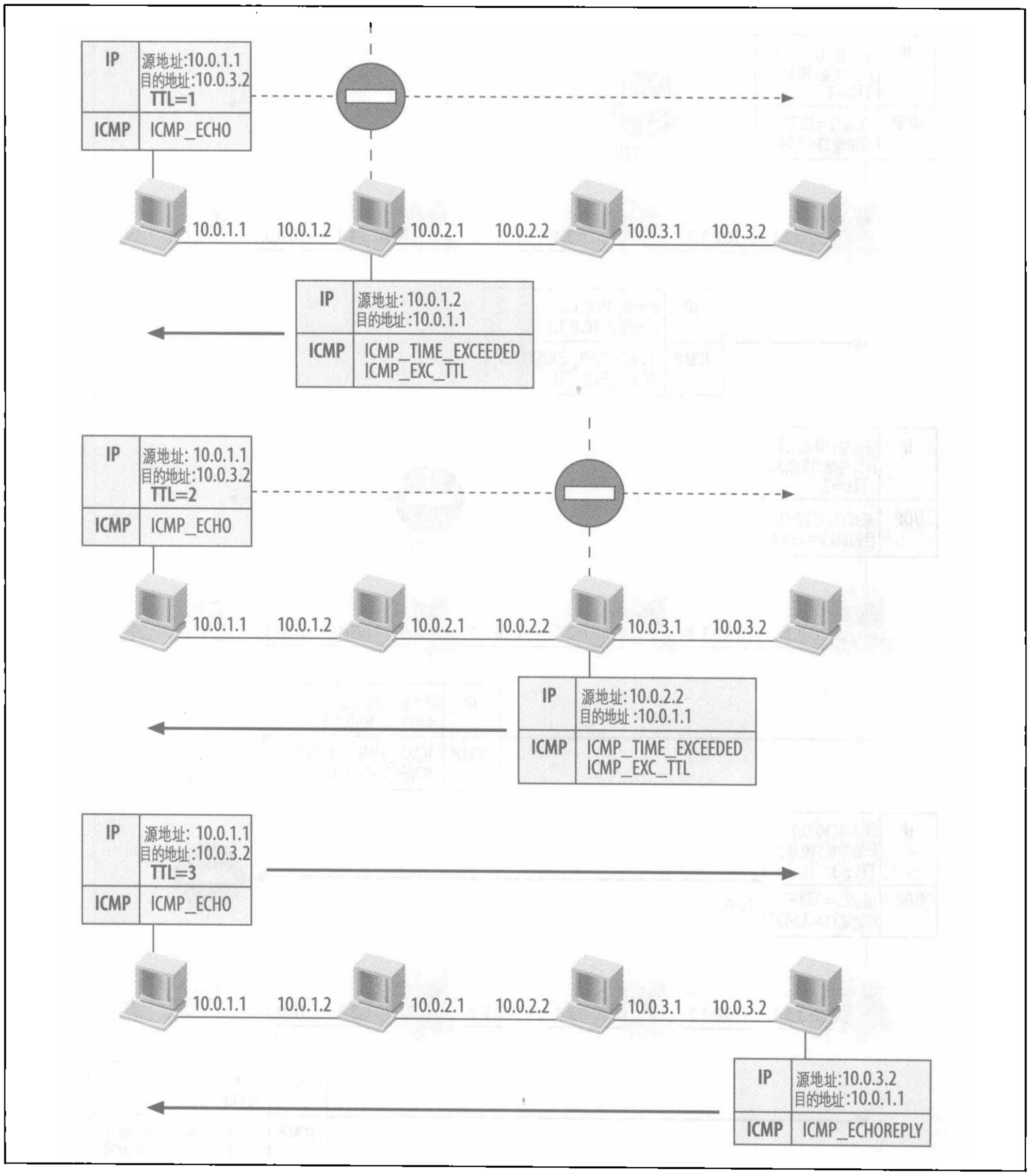


图 25-5：traceroute 使用 ICMP 的范例

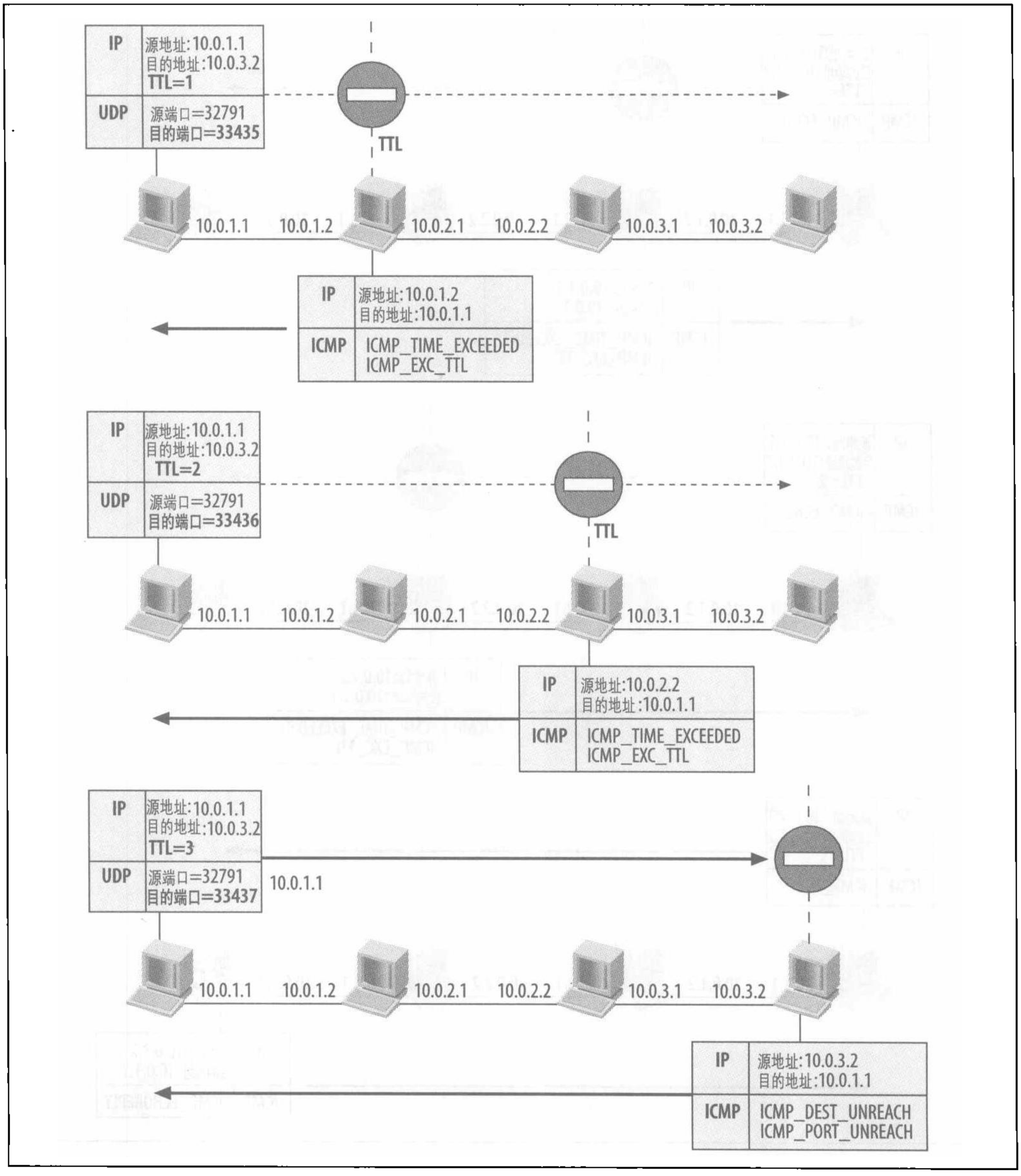


图 25-6：traceroute 使用 UDP 的示例

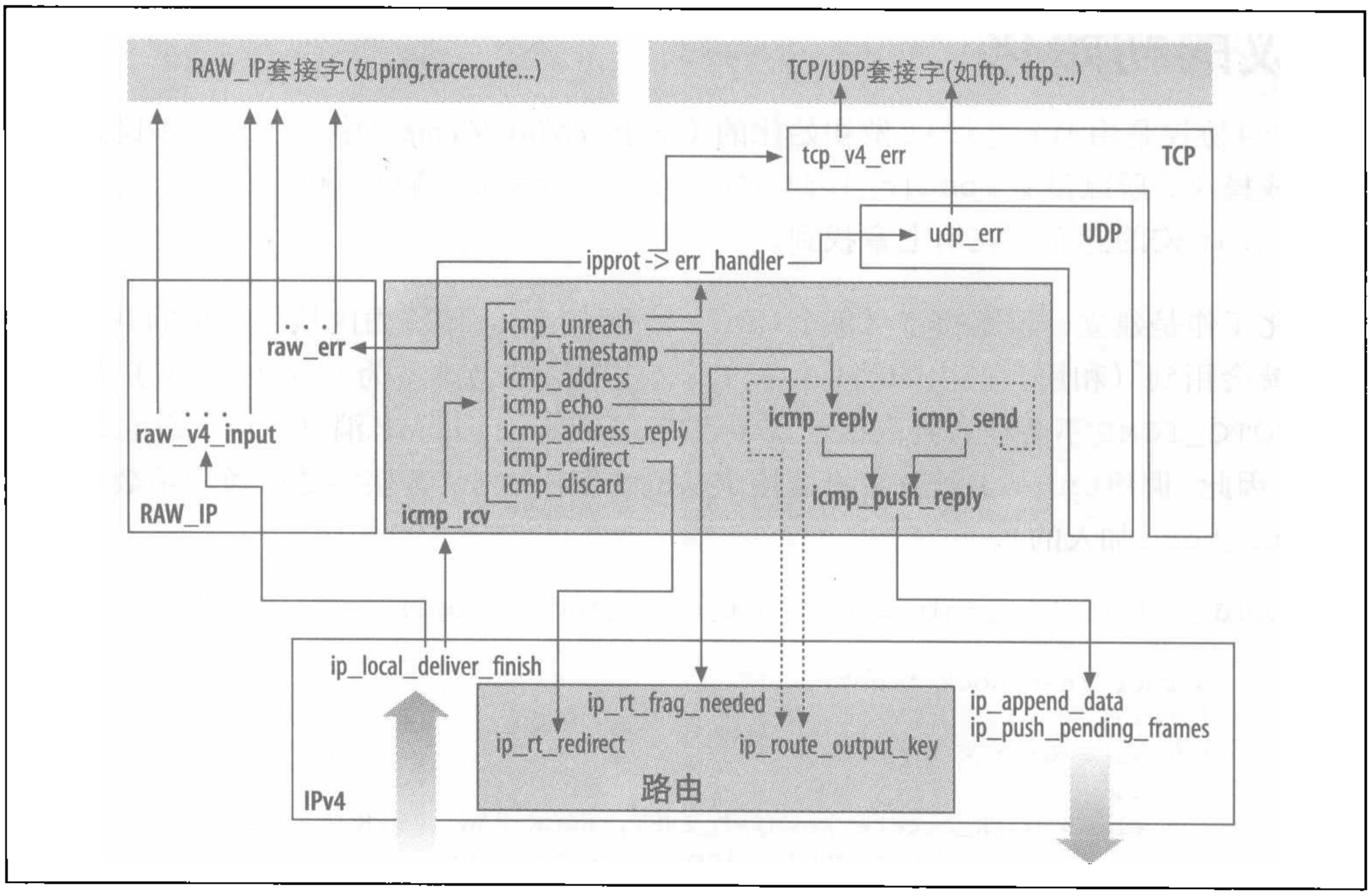


图 25-7：大蓝图

以下是协议间彼此互动的范例：

IP 协议

ip_local_deliver_finish函数（第二十四章有说明）会把入口ICMP消息传递给 ICMP 协议注册的接收函数 icmp_rcv，不过，该函数也会把入口 ICMP 消息传递给针对 ICMP 协议注册的 raw IP 套接字（raw_v4_input）。传输请求会通过ip_append_data和ip_push_pending_frames函数(第二十一章有详细说明)交付给 IP 层。此图没有显示出 IP 协议或路由子系统在哪里调用 icmp_send。

路由子系统

ICMP 消息是通过 icmp_reply 和 icmp_send（在“传输 ICMP 消息”一节说明）传输的。这些函数会以 ip_route_output_key 函数（第三十三章说明）查询路由表。此外，处理入口 ICMP 消息的函数必须和路由子系统互动（例如，通过使用 ip_rt_redirect 和 ip_rt_frag_needed）来处理由 ICMP 消息所接收的信息。

套接字层

当入口ICMP消息携带着错误指示时，则套接字层会接收通知（通过启用和出差错 IP 封包相关联的传输协议所注册的 err_handler 函数指针）。

入口 ICMP 消息会被分发到正确的处理函数（根据 ICMP 类型）。

协议的初始化

ICMPv4协议是由icmp_init做初始化的（位于*net/ipv4/icmp.c*中）。ICMP协议无法被编译成模块，所以没有module_init或module_cleanup函数。附在icmp_init前面的__init宏的意义可在第七章找到。

初始化工作是建立一群套接字（每个CPU一个套接字），传输由内核所产生的ICMP消息时就会用到（和用户产生的消息不同）。这些套接字（类型为SOCK_RAW）及协议IPPROTO_ICMP不会插入到内核套接字表中，因为入口ICMP消息不应该以它为目的对象。因此，调用unhash函数会将套接字从hash表中取出（套接字是由通用函数sock_create_kern加入的）。

```
void __init icmp_init(struct net_proto_family *ops)
{
    struct inet_sock *inet;
    int i;
    for (i = 0; i < NR_CPUS; i++) {
        ...
        err = sock_create_kern(PF_INET, SOCK_RAW, IPPROTO_ICMP,
                    &per_cpu(__icmp_socket, i));
        if (err < 0)
            panic("Failed to create the ICMP control socket.\n");
        ...
        inet = inet_sk(per_cpu(__icmp_socket, i)->sk);
        inet->uc_ttl = -1;
        inet->pmtudisc = IP_PMTUDISC_DONT;
        per_cpu(__icmp_socket, i)->sk->sk_prot->unhash(per_cpu(__icmp_socket, i)->sk);
    }
```

uc_ttl（送往单播地址的IP封包所用的TTL值）的初值设为－1，以告知内核使用默认单播TTL（sysctl_ip_default_ttl）。IP_PMTUDISC_DONT的设定会关掉套接字上的PMTU发现功能。

这些每个CPU套接字可以由icmp_socket宏（定义在*net/ipv4/icmp.c*中）存取，而该宏会以透明化方式选出正确套接字（根据区域CPU ID）。

```
static DEFINE_PER_CPU(struct socket *, __icmp_socket) = NULL;
#define icmp_socket    __get_cpu_var(__icmp_socket)
```

本章涉及的数据结构

ICMP程序所用的三个主要数据结构为：

icmphdr

ICMP报头。

icmp_control

ICMP消息类型描述符。其字段之一是用于处理入口消息的函数。

icmp_bxm

要作为参数传给“传输ICMP消息”一节所述两个传输函数的输入结构。该结构内包含传输ICMP消息所需的所有必要信息。

icmphdr结构

图25-1已知ICMP消息的结构。下列数据结构（来自*include/linux/icmp.h*中）用于定义ICMP报头：

```
struct icmphdr {
  __u8    type;
  __u8    code;
  __u16    checksum;
  union {
        struct {
            __u16    id;
            __u16    sequence;
        } echo;
        __u32    gateway;
        struct {
            __u16    __unused;
            __u16    mtu;
        } frag;
  } un;
};
```

前三个字段对所有ICMP类型而言都相同，接着一个联合（union）类型的数据会根据消息类型而提供各种不同字段。例如，un.frag是由ICMP_FRAG_NEEDED消息所用，而un.echo是由查询消息（如ICMP_ECHO、ICMP_ECHOREPLY等）所用。

icmp_control结构

就每种ICMP类型而言，都有一个icmp_control数据结构实例（定义在net/ipv4/icmp.c中）。其中有一个字段，指向一个函数的指针；而该函数会被调用处理入口ICMP消息。下面是它的字段：

int output_entry

int input_entry

这是接收函数icmp_rcv和“传输ICMP消息”一节中的传输函数所用的索引值，用于更新数组内正确的SNMP计数器。参见“ICMP统计资料”一节。

void (*handler)(struct sk_buff *skb)

由接收函数 icmp_rcv 启用的函数，处理送进来的 ICMP 消息。

short error

当 ICMP 类型被分类为错误（和查询相反）时，此标志就会被设定。参见表 25-1。

以下是 error 字段有用的两个例子，如“传输 ICMP 错误消息”一节所述：

- 内核可以检查，来确定不是采用另一个ICMP错误消息回复入口ICMP错误消息（这是不允许的）。
- 被分类为错误的 ICMP 类型会得到较佳 TOS（IPTOS_PREC_INTERNET-CONTROL），因为这些 ICMP 类型被视为比较重要（参考 icmp_send）（注 3）。

请参考“接收 ICMP 消息”一节了解 icmp_control 数据结构是如何组织的。

icmp_bxm 结构

icmp_bxm 定义在 *net/ipv4/icmp.c* 中。以下说明其字段：

struct sk_buff *skb

就那些以 icmp_send 传送的 ICMP 消息而言，代表的是触发传输的入口 IP 封包；就那些以 icmp_reply 传送的 ICMP 消息而言，代表的是入口 ICMP 消息请求。

int offset

skb->data 和 skb->nh 之间的偏移量（也就是 IP 报头的尺寸）。评估那些需要有效载荷的 ICMP 消息的 ICMP 有效载荷能放多少数据进去时，此偏移量就有用处。

int data_len

ICMP 有效载荷尺寸。

```
struct {
    struct icmphdr icmph;
    __u32 times[3];
} data
```

icmph 就是要传输的 ICMP 消息的报头。times 是由 ICMP_TIMESTAMPREPLY 消息类型所用。

int head_len

ICMP 报头的尺寸。

struct ip_options replyopts

unsigned char optbuf

replyopts存储在IP层所用的IP选项。它的值是由ip_options_echo初始化（根

据skb的IP选项）的。optbuf是replyopts的扩充，由ip_options_echo存取（通过ip_options的__data字段）。参见第十九章。

传输ICMP消息

“ICMP报头”一节所介绍的两种ICMP消息（错误和查询）是由不同函数传输的：

icmp_send

当检测到特定情况时，由内核使用，传输ICMP错误消息。

icmp_reply

由ICMP协议使用，回复需要做响应的入口ICMP请求消息。

这两个函数都会接收一个skb缓冲区作为输入。然而，给icmp_send作为输入的缓冲区是代表触发ICMP消息传输的入口IP封包；而给icmp_reply作为输入的缓冲区代表的是需要响应的入口ICMP请求消息。

*net/core/icmp.c*中的程序会处理送进来的ICMP消息，因此，总是会使用icmp_reply传输ICMP消息来作为另一个于输入参数中所接收的ICMP消息的响应。其他内核子系统（如路由、IP等）需要产生ICMP消息时，会使用icmp_send，如图25-8所示。

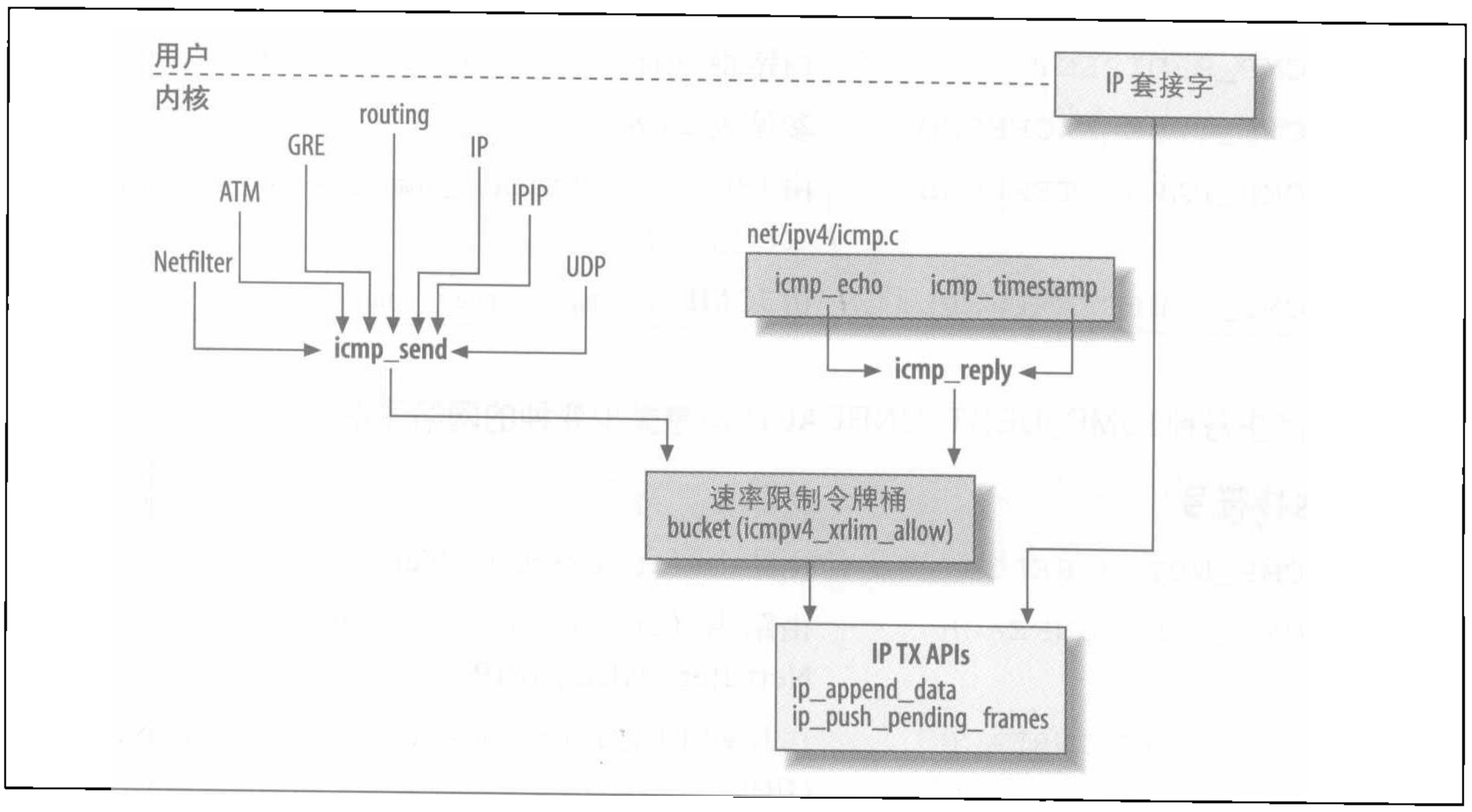


图25-8：使用icmp_send/icmp_reply的子系统

在这两种情况下：

- ip_route_output_key用于找出前往目的地的路径（参见第三十三章）。
- ip_append_data和ip_push_pending_frames这两个函数用于请求对IP层的传输。这些函数已在第二十一章说明过。
- 内核空间中产生的ICMP消息以icmpv4_xrlim_allow（参见“速率限制”一节）作速率限制（如果内核有通过*/proc*做此设定）。
- 这些传输会以每个CPU回转锁进行串行化（通过icmp_xmit_lock和icmp_xmit_unlock）。每个CPU回转锁的存取是通过每个CPU ICMP套接字（参见“协议的初始化”一节）。当回转锁已被持有而无法取得时，传输就会失败（但是这些函数都不会返回错误代码）。

表25-6～表25-8说明出表25-1的ICMP类型是由内核于何处产生的。就本书所谈的这些子系统而言，也包括ICMP消息于哪些函数内产生。

表25-6：产生ICMP消息的网络子系统

类型	名称	产生
0	ICMP_ECHOREPLY	由ICMP（icmp_echo）
3	ICMP_DEST_UNREACH	参见表25-7
5	ICMP_REDIRECT	由路由（ip_rt_send_redirect）
11	ICMP_TIME_EXCEEDED	参见表25-8
12	ICMP_PARAMETERPROB	由IPv4（ip_options_compile,ip_options_rcv_srr）
14	ICMP_TIMESTAMPREPLY	由ICMP（icmp_timestamp）

表25-7：产生各种ICMP_DEST_UNREACH消息类型变种的网络子系统

代码	内核符号	产生
0	ICMP_NET_UNREACH	由路由（ip_error）、Netfilter
1	ICMP_HOST_UNREACH	由路由（ip_error,ipv4_link_failure）、Netfilter、GRE、IPIP
2	ICMP_PROT_UNREACH	由IPv4（ip_local_deliver_finish）、Netfilter、GRE
3	ICMP_PORT_UNREACH	由Netfilter、GRE、IPIP、UDP
4	ICMP_FRAG_NEEDED	由IPv4（ip_fragment）、GRE、IPIP、虚拟服务器
5	ICMP_SR_FAILED	由IPv4（ip_forward）

表25-7：产生各种ICMP_DEST_UNREACH消息类型变种的网络子系统（续）

代码	内核符号	产生
9	ICMP_NET_ANO	由Netfilter
10	ICMP_HOST_ANO	由Netfilter
13	ICMP_PKT_FILTERED	由路由（ip_error）、Netfilter

Netfilter根据所采用的配置（例如，以*iptables*）而丢弃入口IP封包时，就会产生ICMP_DEST_UNREACH消息。REJECT目的对象所用的-reject-with选项可让用户在回绝符合特定规则的入口IP封包时选择要使用哪一种ICMP消息类型。

隧道协议（例如，IPIP和GRE；分别定义在*net/ipv4/ipip.c*中和*net/ipv4/ip_gre.c*中）必须根据RFC 2003第4节来处理ICMP消息。

表25-8：产生各种ICMP_TIME_EXCEEDED消息类型变种的网络子系统

代码	内核符号	产生
0	ICMP_EXC_TTL	由IPv4（ip_forward）
1	ICMP_EXC_FRAGTIME	由IPv4（ip_expire）

传输ICMP错误消息

图25-9（a）和25-9（b）说明了icmp_send的内部逻辑。以下是它的输入参数：

skb_in

和此错误相配的输入IP封包。

type

code

ICMP报头中所用的“类型”和“代码”字段。

info

其他信息：ICMP_FRAG_NEEDED消息的MTU、ICMP_REDIRECT消息所用的网关地址及ICMP_PARAMETERPROB消息的偏移量。

icmp_send一开始先做一些健康检查，把非法请求过滤掉。下列情况会使其放弃：

- 接收的IP数据报为广播或多播。这种情况的检测是检查和skb_in相关联的路由缓存项目的RTCF_BROADCAST和RTCF_MULTICAST标志。

- 接收的IP数据段封装在广播链路层数据帧内。这种情况的判断是比较封包类型skb_in->pkt_type和PACKET_HOST。
- IP数据报是片段，而且不是原有封包的第一个片段。这种情况的判断是读取IP报头的“偏移量”字段（参见第二十二章）。
- IP数据报携带ICMP错误消息。你不能使用错误消息去回复错误消息。

对IP报头做初始化并非ICMP层的责任。然而，IP层会根据ICMP的需求，对一些IP报头字段做初始化。特别的是：

源IP地址

当ICMP消息的目的对象并非本地配置的IP地址时（也就是RTCF_LOCAL），则放入封装报头内的源IP地址的选择就是根据sysctl_icmp_errors_use_inbound_ifaddr配置而定（参见“通过/proc文件系统调整”一节）。

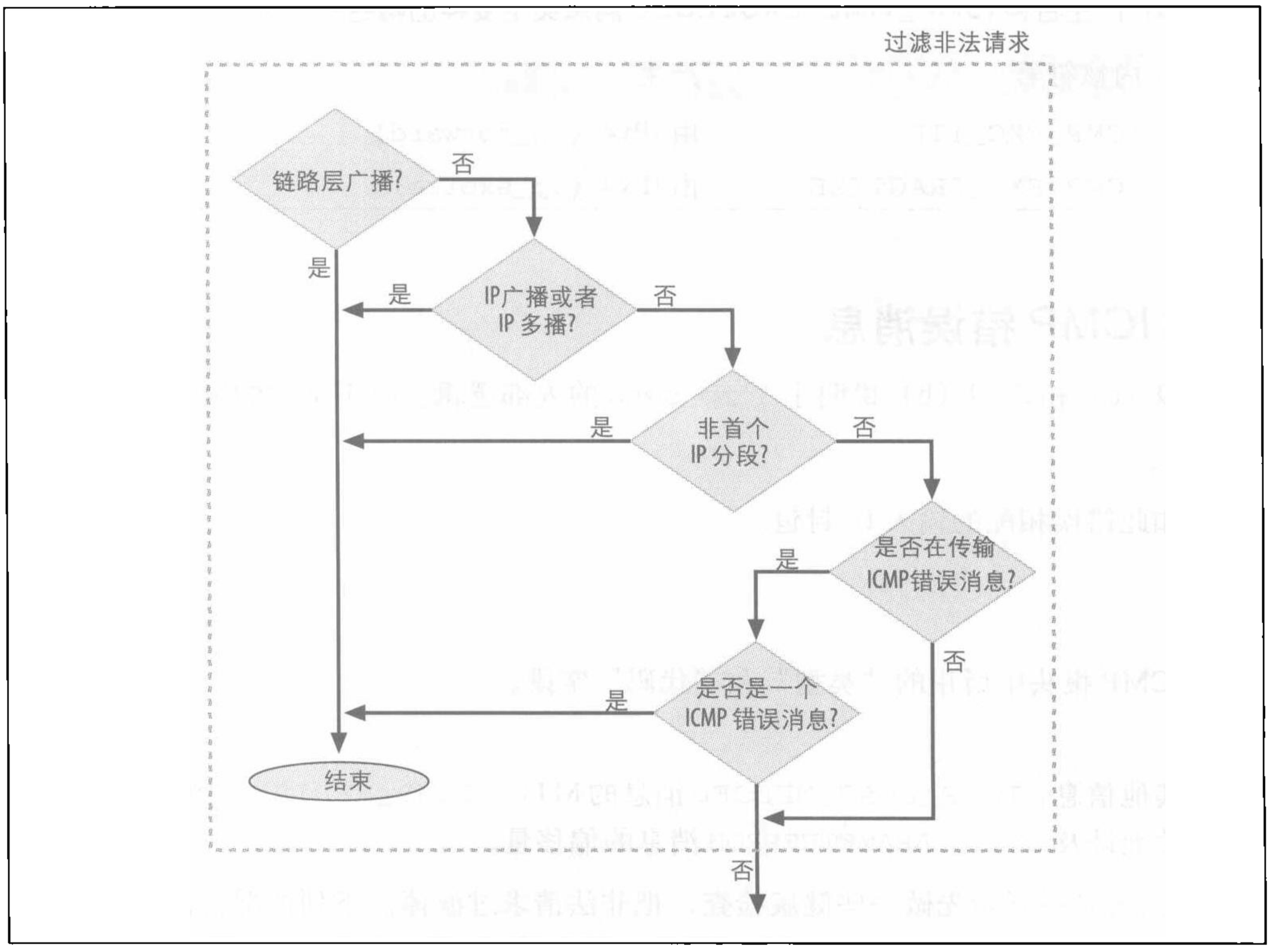


图25-9 (a)：icmp_send函数

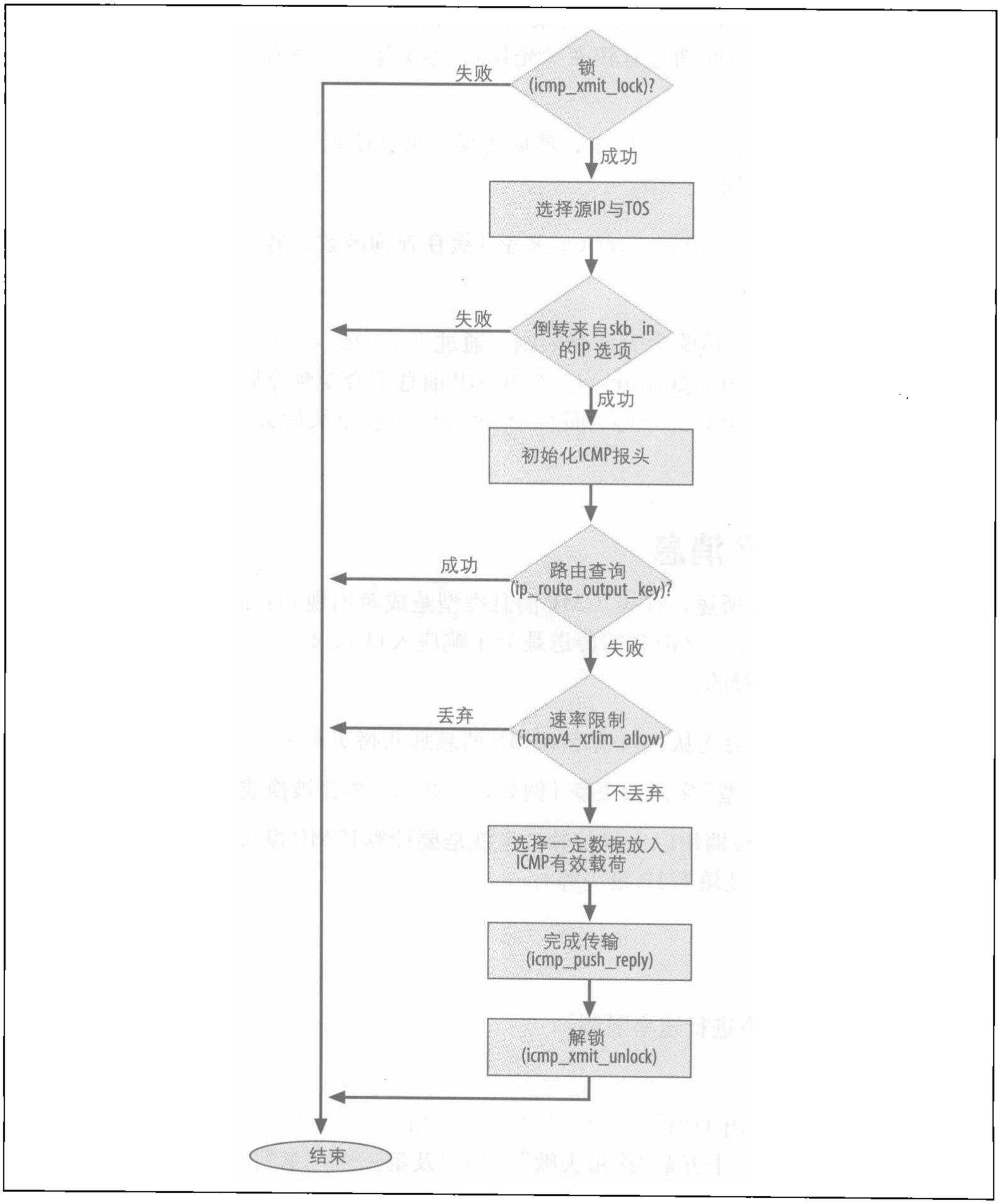


图 25-9 (b)：icmp_send 函数

服务类型（TOS）

TOS 是从 skb_in 的 TOS 拷贝而来的。此外，当 ICMP 消息为错误消息时（参见表

25-1 所示），则 TOS 的优先权元素的初值就会设成 IPTOS_PREC_INTERNET-CONTROL（也就是此消息有较高优先权）。参见第十八章有关 TOS 的其他信息。

IP 选项

IP 选项是从 skb_in 拷贝出来，然后再反过来（使用 ip_options_echo）。参见第十九章“IP 选项”一节。

接着，该函数会以 ip_route_output_key（缓存查询函数，第三十三章会介绍）找出通往目的地的路径。

注意，按图 25-8 所示，传输会受速率限制（通过 icmpv4_xrlim_allow 函数，使用令牌桶算法（token bucket algorithm））。当 ICMP 消息不会受到令牌：桶算法压制时，传输最后会调用 icmp_push_reply，而 icmp_push_reply 最后会调用图 25- 8 所示的两个 IP 函数。

回复入口 ICMP 消息

如“ICMP 报头”一节所述，有些 ICMP 消息类型是成对出现的：请求消息及响应消息。例如，ICMP_ECHOREPLY 消息的传送是为了响应入口 ICMP_ECHO 消息。响应消息的传输是按下列步骤进行的：

1. 响应消息的报头会先从入口请求 ICMP 消息那儿拷贝出来。
2. ICMP 报头的“类型”字段会更新（例如，ICMP_ECHO 会被换成 ICMP_ECHOREPLY）。
3. icmp_reply 会被调用以完成传输（也就是要计算 ICMP 报头的校验和、找出通往目的地的路径以及填写 IP 报头等）。

速率限制

ICMP 消息在两个地方进行速率限制：

由路由程序

路由程序只对外出 ICMP_DEST_UNREACH 和 ICMP_REDIRECT 消息类型进行速率限制。参见第三十五章“路由失败”一节以及第三十三章“出口 ICMP REDIRECT 限速”一节。

由 ICMP 程序

ICMP 程序可以对所有外出 ICMP 消息类型（只有本节稍后所列的少数例外）进行速率限制，包括那些由路由程序进行速率限制的类型。这两种速率限制有一点重要的差别：路由程序是根据每个目的 IP 地址对 ICMP 消息进行速率限制的，但是，

ICMP程序是按每个源IP地址进行速率限制。也就是说，那些被ICMP和路由程序做速率限制的类型会被限制速率两次。

我来说明这一点。内核会保存所需的速率限制信息，用于对路由表缓存的dst_entry项目施加令牌桶算法。每个dst_entry实例都会配置一个目的IP地址（详情在第三十三章）。单这一点就可让我们知道，速率限制的施加是以每个IP地址为基准，而非以每种ICMP消息类型为基准，但是，我们来看下基于源和基于目的地址速率限制的差异在何处：

- 当内核子系统（例如，IPv4协议）处理的入口IP封包符合特定错误情况时，就会传送ICMP错误消息回该入口IP封包的源地址。ICMP程序会查询路由表，路由查询后会返回一个缓存项目，该缓存项目用于存储速率限制信息。此缓存项目和那一条从本地主机至出错IP封包的源地址（也就是出错IP封包的源IP地址）的路径相关联。这就是所谓的基于源IP地址速率限制。
- 当路由程序无法传送入口IP封包时，就会产生ICMP_HOST_UNREACH消息，但是，当入口IP封包的目的IP地址最好通过另一网关连接时，就会产生ICMP_REDIRECT消息。就这两种情况而言，路由程序会把一个项目加进缓存内，其相配的目的IP地址就是入口IP封包的目的IP地址。因此，这种方式就被称为基于目的IP地址速率限制。第三十五章会说明这类缓存项目如何由后续相符合的IP封包使用。

速率限制的实现

现在，我们来看看ICMP程序如何应用速率限制。如图25-10所示，当内核配置速率限制时，任何时候，ICMP消息传输时，icmpv4_xrlim_allow函数都将被调用来施加速率限制。要做速率限制的ICMP消息类型（sysctl_icmp_ratemask）及速率限制的速率（sysctl_icmp_ratelimit）都可以通过*/proc*配置（参见“通过/proc文件系统做调整”一节）。

在下列情况下，icmpv4_xrlim_allow不会施加速率限制：

- ICMP消息的类型是内核所未知的（这类消息可能很重要）。
- PMTU协议所用的ICMP消息（RFC 1191），也就是类型为ICMP_DEST_UNREACH而代码为ICMP_FRAG_NEEDED（注4）。PMTU已在第十八章介绍过。

注4：注意，内核所用策略和防火墙所用的无关。例如，防火墙只让少数ICMP消息通过，而其他的全都丢弃，这是很常见的。偶尔，PMTU所用的ICMP消息也会被丢弃，只不过这样和RFC建议案相违背而已。

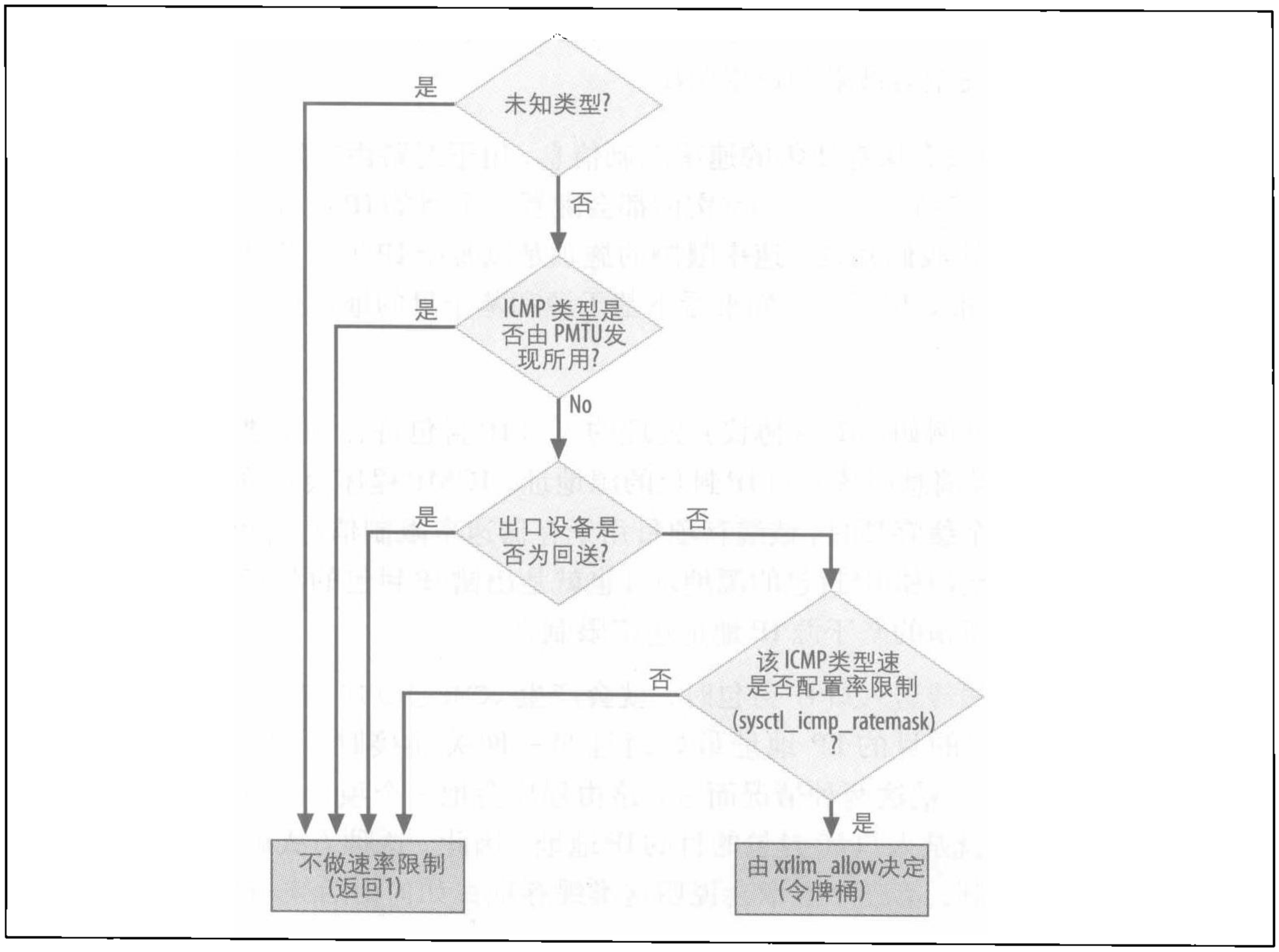


图 25-10：icmpv4_xrlim_allow 函数

- ICMP 是从回环设备传送出去的。

icmpv4_xrlim_allow是包裹函数，内含另一个更通用的函数xlim_allow，而xlim_allow才是干正事的函数。如果根据sysctl_icmp_ratemask掩码，ICMP消息必须做速率限制，就会调用此函数。

```
#define XRLIM_BURST_FACTOR 6
int xrlim_allow(struct dst_entry *dst, int timeout)
{
    unsigned long now;
    int rc = 0;

    now = jiffies;
    dst->rate_tokens += now - dst->rate_last;
    dst->rate_last = now;
    if (dst->rate_tokens > XRLIM_BURST_FACTOR * timeout)
            dst->rate_tokens = XRLIM_BURST_FACTOR * timeout;
    if (dst->rate_tokens >= timeout) {
        dst->rate_tokens -= timeout;
        return 1;
```

```
        }
        return rc;
    }
```

xrlim_allow会施加简单的令牌桶算法。每当它被调用时，就会更新可用的dst->rate_tokens令牌（以jiffies为单位测量），来确保累积的令牌不会多过预定的最大值（XRLIM_BURST_FACTOR），而且如果可用令牌足够的话，就可让ICMP消息传输。输入参数timeout代表的是要施加的速率，以Hz为单位表示（例如，1*HZ指的速率限制是每秒一条ICMP消息）。

注意，因为xrlim_allow是通用函数，各种协议都会用到，因此，它所操作的是独立于协议之外的路由缓存项目（dst_entry结构）。但是，icmpv4_xrlim_allow是IPv4函数，因此操作的是rtable数据结构。有关dst_entry和rtable数据结构的细节，请参考第三十六章。

接收ICMP消息

icmp_rcv函数由ip_local_deliver_finish所调用，来处理入口ICMP消息。

ICMP协议会注册它的接收函数icmp_rcv（位于*net/ipv4/protocol.c*中），如第二十四章所述。参见第二十章有关入口IP封包本地传递的细节。

首先，会验证ICMP消息的校验。注意，即使进行接收的NIC可以使用硬件计算L4校验和（此例为ICMP校验和），而该校验和表明ICMP消息已损毁，icmp_rcv还是会在软件中再次查证该校验和。你可以参考第十九章“sk_buff结构”一节有关NIC所支持的L4校验和计算细节。

并非所有ICMP消息类型都可传送到多播IP地址：只有ICMP_ECHO、ICMP_TIMESTAMP、ICMP_ADDRESS以及IMCP_ADDRESSREPLY才行。icmp_rcv会把那些不合此规则的消息过滤掉。特别的是，如果系统配置成要把入口广播ICMP_ECHO消息丢弃掉，则该类消息就会被丢弃掉。参见“通过/proc文件系统做调整”一节。

当所有健康检查都满足时，icmp_rcv就会把入口ICMP消息传给正确的辅助函数。辅助函数是通过icmp_pointers向量存取的，而该向量是在*net/ipv4/icmp.c*末尾初始化的。icmp_pointers是一个数组，包含一些icmp_control数据结构。表25-9总结icmp_pointers部分的初始化。参见“icmp_control结构”一节有关处理函数和错误字段的意义。任何不在表中的类型就是已废弃、不支持、或者不应该在内核空间中处理。就这些类型而言，处理函数会设成icmp_discard。

表 25-9：处理函数和错误的初始化

类型	内核符号	处理函数	错误
3	ICMP_DEST_UNREACH	icmp_unreach	1
4	ICMP_SOURCE_QUENCH	icmp_unreach	1
5	ICMP_REDIRECT	icmp_redirect	1
8	ICMP_ECHO	icmp_echo	0
11	ICMP_TIME_EXCEEDED	icmp_unreach	1
12	ICMP_PARAMETERPROB	icmp_unreach	1
13	ICMP_TIMESTAMP	icmp_timestamp	0
17	ICMP_ADDRESS	icmp_address	0
18	ICMP_ADDRESSREPLY	icmp_address_reply	0

图 25-11 表明了 icmp_rcv 内部逻辑。

注意，ICMP_ADDRESS 和 ICMP_ADDRESSREPLY 都不支持；为其所注册的处理函数只是占个位置或者提供某种记录功能而已。

另外，icmp_unreach 处理函数会处理各种 ICMP 消息类型，不仅是 ICMP_DEST_UNREACH。

图 25-12（a）说明当 icmp_rcv 调用时，skb 的一些指针是怎么初始化的，而图 25-12（b）则说明表 25-9 的处理函数被调用时，那些指针是怎么初始化的。分析表 25-9 的函数时，该图就有用处，特别是 icmp_unreach。

处理 ICMP_ECHO 和 ICMP_ECHOREPLY 消息

ICMP_ECHO 消息的处理是根据“回复入口 ICMP 消息”一节所述的通用模型：

```
static void icmp_echo(struct sk_buff *skb)
{
    if (!sysctl_icmp_echo_ignore_all) {
        struct icmp_bxm icmp_param;

        icmp_param.data.icmph              = *skb->h.icmph;
        icmp_param.data.icmph.type         = ICMP_ECHOREPLY;
        icmp_param.skb                     = skb;
        icmp_param.offset                   = 0;
        icmp_param.data_len                    = skb->len;
        icmp_param.head_len          = sizeof(struct icmphdr);
        icmp_reply(&icmp_param, skb);
    }
}
```

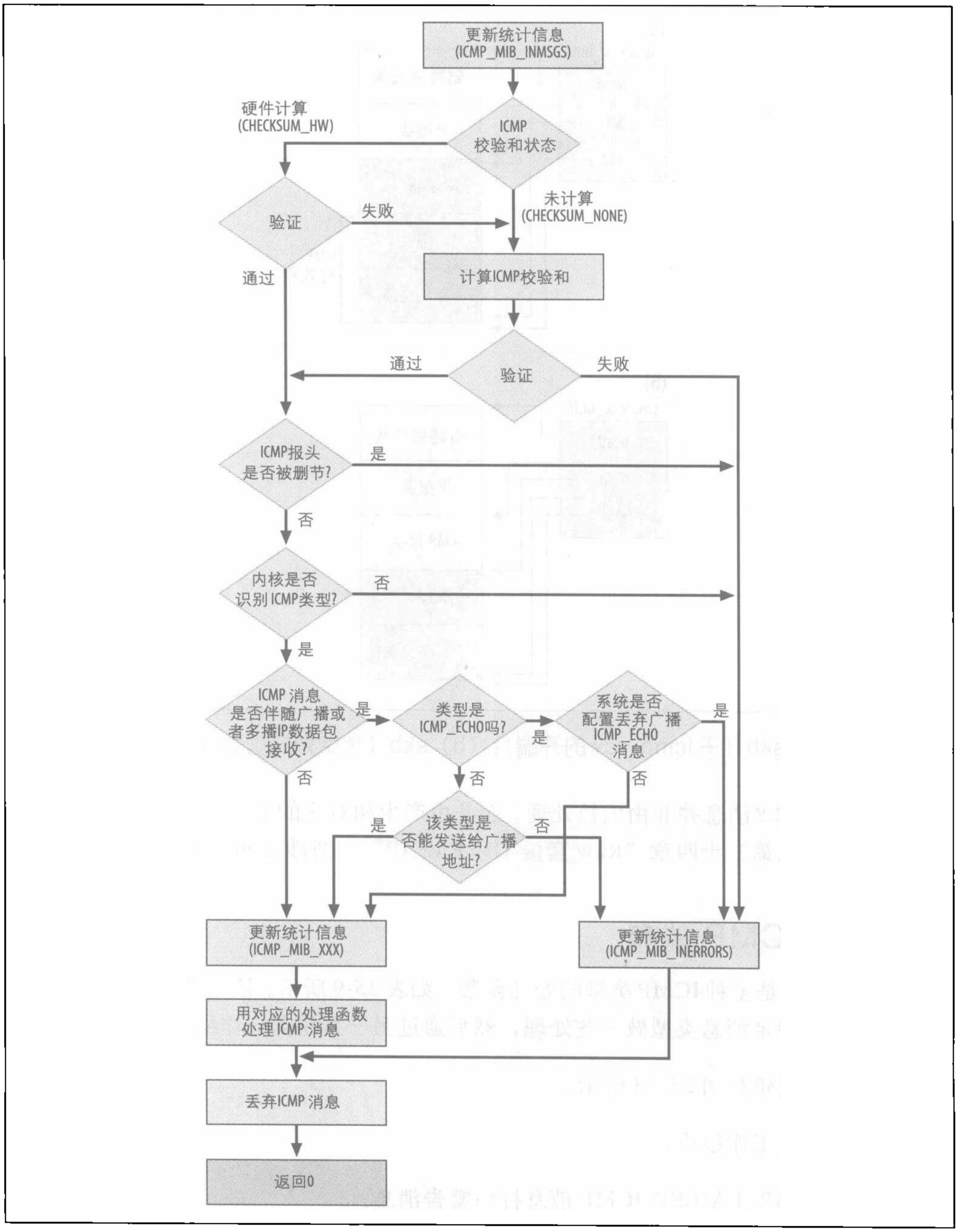


图 25-11：icmp_rcv 函数

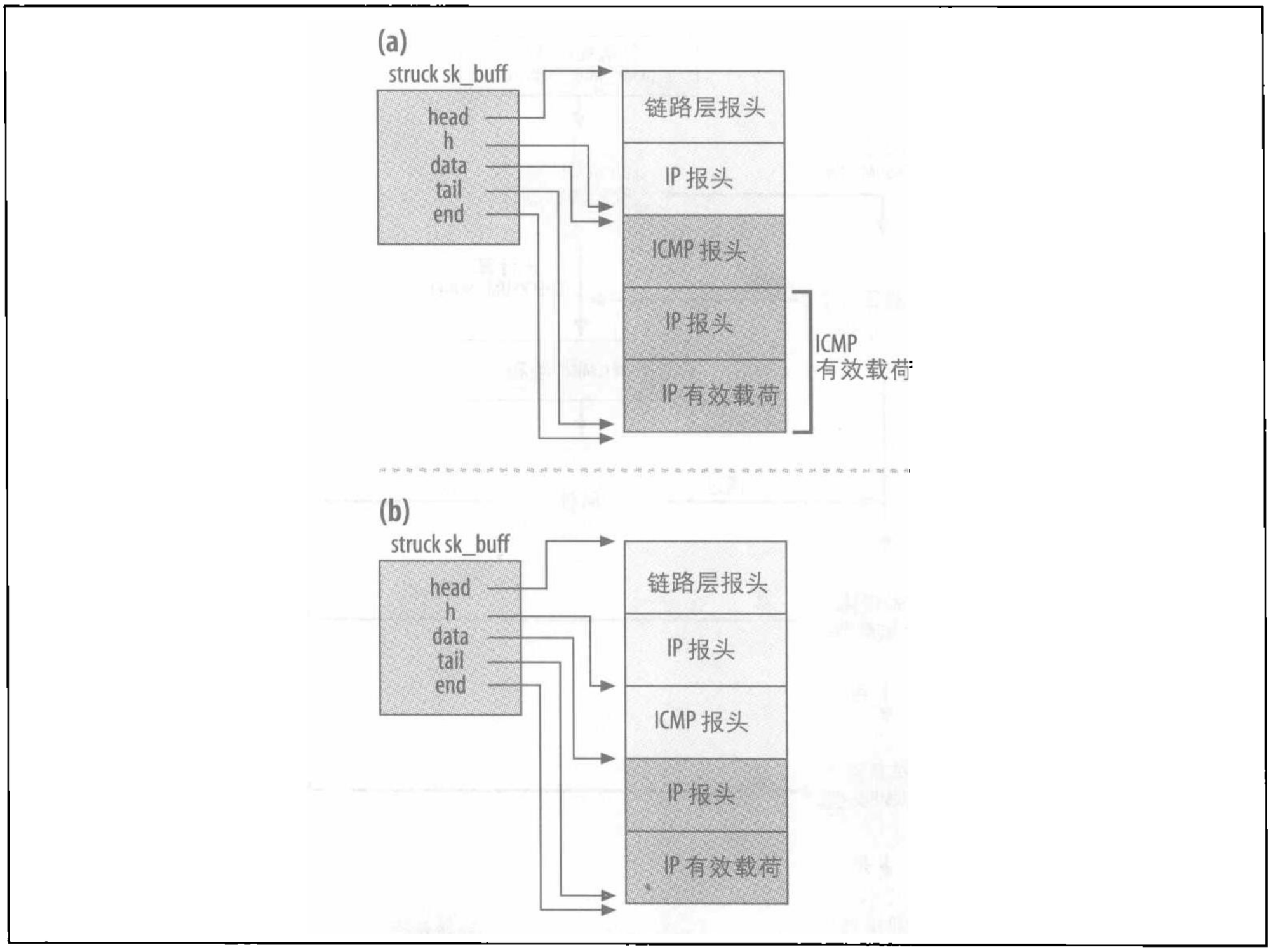


图 25-12：（a）skb（于 icmp_rcv 的开端）；（b）skb（传给处理函数时）

ICMP_ECHOREPLY消息并非由内核处理，而是由产生相对应的ICMP_ECHO消息的应用程序处理。参见第二十四章“Raw 套接字和 Raw IP”一节涉及 *ping* 的范例。

处理常见 ICMP 消息

icmp_unreach是多种ICMP类型的处理函数，如表25-9所示。该函数会先做一些健康检查，再根据特定消息类型做一些处理，然后通过另一共用部分作结论。

此函数的内部逻辑如图 25-13 所示。

每种类型的处理工作很少：

- 为 ICMP_SR_FAILED ICMP 消息打印警告消息。
- 当它接收类型为ICMP_DEST_UNREACH及代码为ICMP_FRAG_NEEDED的ICMP消息时，就会更新路由缓存。缓存是以 ip_rt_frag_needed更新，但是，只有在 PMTU 发现功能开启时（也就是如果 ipv4_config.no_pmtu_disc为非零）。当 PMTU 发现没开启时，内核只会记录警告消息。

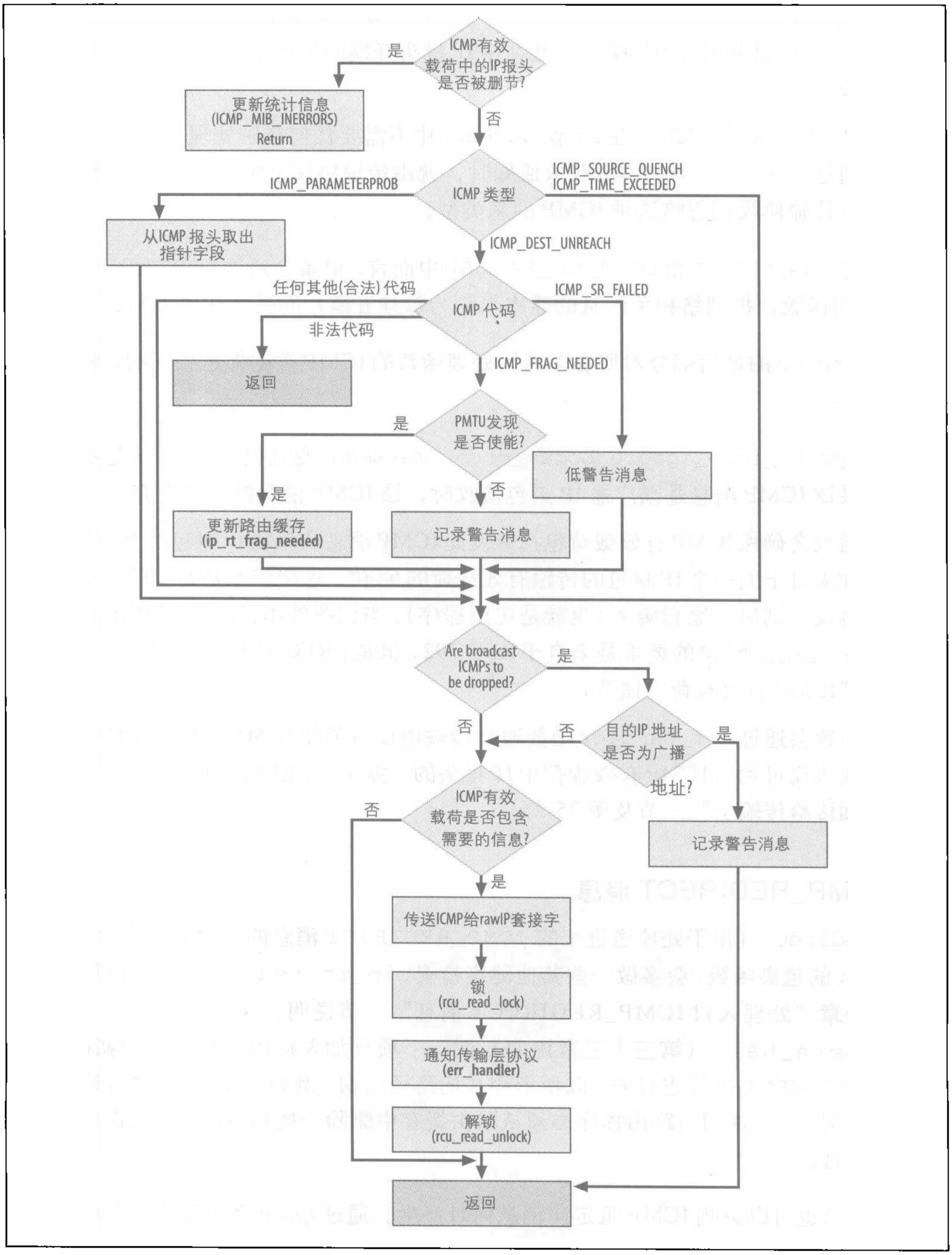


图 25-13：icmp_unreach 函数

- 当消息类型为ICMP_PARAMETERPROB时，就从ICMP报头取出pointer字段。pointer是相对于ICMP有效载荷中IP报头开端的偏移量。此字段也会传给传输协议。
- ICMP_SOURCE_QUENCH在icmp_unreach中不需要任何特别处理，所以，当传输协议通过err_handler函数获取通知时，就由传输协议决定如何处理。目前而言，所有传输协议都忽略这种ICMP消息类型。

就ICMP_FRAG_NEEDED和ICMP_SR_FAILED中而言，记录会通过LIMIT_NETDEBUG（这是通用函数，把网络相关消息的速率限制为每秒五道）而受到速率限制。

icmp_unreach的最后部分对所有以其为处理函数的ICMP类型都是相同的，相关任务如下所示：

- 当sysctl_icmp_ignore_bogus_error_messages变量设定时(默认是不设定)，如果该ICMP消息是以广播IP封包接收时，该ICMP消息就会被丢弃。
- 该函数会确保ICMP有效载荷包括触发此ICMP消息产生的IP封包的整个IP报头，外加来自于同一个IP封包的传输有效载荷的64位。这个信息是必要的，才能让传输协议识别出本地套接字(也就是应用程序)。当该条件不符时，ICMP消息就会被丢弃。注意，64位的要求是来自于RFC 792，但是，RFC 1812改变了这个要求（参见“ICMP有效载荷”该节)。
- 该函数会通过err_handler函数通知传输协议有关此ICMP消息的事情。正确的传输协议可利用ICMP有效载荷中IP报头的“协议”字段识别出来。参见“把错误通知传给传输层”一节及图25-2。

处理ICMP_REDIRECT消息

icmp_redirect（用于处理送进来的ICMP_REDIRECT消息的函数）是内含ip_rt_redirect的包裹函数，会多做一些其他健康检查。ip_rt_redirect函数所用逻辑在第三十一章“处理入口ICMP_REDIRECT消息”一节说明。ip_rt_redirect用rt_intern_hash（第三十三章说明）把一个项目加入路由缓存。路径初始化时，RTCF_REDIRECTED标志打开，以便于和其他路径区别。例如，第三十章“合格的缓存受害者范例”一节说明当路由程序必须从路由缓存中删除一些项目时，路由程序如何使用这项信息。

系统管理员也可以影响ICMP重定向消息何时产生。通过*/proc*文件系统，有可能为每个接口指定是否传送和接受ICMP重定向消息(参见第三十六章“/proc/sys/net/ipv4/conf

目录”一节）。使用防火墙的话，管理员可以指定接受谁传来的特定ICMP封包类型，因此，也就能决定要信任谁的 ICMP_REDIRECT 消息。

处理 ICMP_TIMESTAMP 和 ICMP_TIMESTAMPREPLY 消息

入口 ICMP_TIMESTAMP 消息的处理方式就是回复 ICMP_TIMESTAMPREPLY 消息；使用的手法就如“回复入口 ICMP 消息”一节所讨论的。第二和第三个时间戳不会根据“ICMP_TIMESTAMP和ICMP_TIMESTAMPREPLY”一节所见规则做初始化，而是以 do_gettimeofday 设成相同的时间戳。

注意，head_len的初始化不仅包括默认的ICMP报头长度，也包括那三个32位时间戳。

处理 ICMP_ADDRESS 和 ICMP_ADDRESSREPLY 消息

因为 Linux 内核不产生 ICMP_ADDRESS 消息，对本地产生的查询而言（至少是不在内核空间内的查询），ICMP_ADDRESSREPLY消息不会回复。然而，入口设备上有开启转发和记录火星地址（Martian address）（注 5）的功能时，Linux 就会以 icmp_address_reply 监听 ICMP_ADDRESSREPLY（注 6）。icmp_address_reply 检查该消息所主张的掩码对进行接收的接口上所配置的IP地址而言是否正确：如果进行接收的接口没有任何IP地址的配置和ICMP消息传送者所用的源IP地址是位于相同子网时（也意谓着相同的网络掩码），内核就会记录警告消息。

当路由缓存设定RTCF_DIRECTSRC标志时，回复消息在接收后不再做健康检查。只有当本地主机可通过具有本地范围的下个跳点（也就是只存在于Linux机器内部）连接目的地址时，此标志才会设定。

ICMP 统计资料

ICMP 协议保持 RFC 2011 所定义的统计数据，也就是将其存储在 icmp_mib 数据结构内。内核按每个 CPU 存储统计资料，此外，就每个 CPU 而言，内核也会区分在软件中断环境内外所更新的统计资料。换而言之，就每个计数器而言，每个 CPU 都有两个实例：这两个实例之一是由在软件中断环境内执行的程序使用，而另一个则是给不在软件中断环境内执行的程序使用。所有这些 icmp_mib 实例的分配都是通过 init_ipv4_

注 5： 即无效的 IP 地址

注 6： 参见第三十六章“文件描述”一节的 *log_martians* 的定义。

mibs（位于*net/ipv4/af_inet.c*中）。icmp_statistics是一个两元素数组，第一个元素代表的是内含一些icmp_mib实例的每CPU数组（这些实例是由在软件中断环境内执行的程序所使用的），第二个元素则代表另一个每CPU数组。

```
static int __init init_ipv4_mibs(void)
{
    ...
    icmp_statistics[0] = alloc_percpu(struct icmp_mib);
    icmp_statistics[1] = alloc_percpu(struct icmp_mib);
    ...
}
```

icmp_mib结构由一个带有unsigned long成员的数组组成，每个成员都是计数器（根据RFC 2011针对ICMP协议所定义的）：

```
#define SNMP_MIB_DUMMY  __ICMP_MIB_MAX
#define ICMP_MIB_MAX    (ICMP_MIB_MAX + 1)
struct icmp_mib {
    unsigned long mibs[ICMP_MIB_MAX];
} __SNMP_MIB_ALIGN__;
```

计数器的识别是通过枚举列表ICMP_MIB_*XXX*（定义在*include/linux/snmp.h*）来实现的：

```
enum
{
    ICMP_MIB_NUM = 0,
    ICMP_MIB_INMSG,
    ...
    ICMP_MIB_OUTADDRMASKREPS,
    __ICMP_MIB_MAX
}
```

注意，icmp_mib数组的尺寸只比ICMP_MIB_*XXX*列举列表的尺寸大一个单位。多出来的元素用于内核不认识的ICMP消息类型。

无论何时，当内核必须更新特定计数器时，会根据中断环境选出icmp_statistics的正确元素，然后根据当前CPU选出正确的icmp_mib实例。内核提供一组宏（位于*include/net/icmp.h*中），只需以计数器标识符（也就是ICMP_MIB_*XXX*）作为输入参数，即可以透明化方式处理刚才所提的两个选中项目。

ICMP_INC_STATS

该宏可用在软件中断环境内外。

ICMP_INC_STATS_BH

当程序必须更新一个总是在软件中断环境内执行的计数器时，就可以使用该宏。

ICMP_INC_STATS_USER

当程序必须更新一个绝不会在软件中断环境内执行的计数器时，就可以使用该宏。

这三个宏是定义成包裹宏，内含SNMP子系统所提供的通用宏：

```
#define ICMP_INC_STATS(field)       SNMP_INC_STATS(icmp_statistics, field)
#define ICMP_INC_STATS_BH(field)    SNMP_INC_STATS_BH(icmp_statistics, field)
#define ICMP_INC_STATS_USER(field) SNMP_INC_STATS_USER(icmp_statistics, field)
```

以下是ICMP_MIB_*XXX*值的意义。有关的详细说明，可参考RFC 2011。

和已接收ICMP消息相关字段

ICMP_MIB_INMSG

已接收ICMP消息的数目。包括ICMP_MIB_INERRORS所记的那些消息。

ICMP_MIB_INERRORS

因某种问题而丢弃的ICMP消息数目。icmp_rcv及表25-9的处理函数碰到截断的ICMP报头时，就会丢弃入口消息。“把错误通知传给传输层”一节所述的L4层err_handler函数在碰到截断的ICMP有效载荷时就会丢弃入口消息。

ICMP_MIB_IN*XXX*

除了刚才所列的两个通用计数器外，还有一种每ICMP消息类型的计数器。ICMP_MIB_IN*XXX*会计算已接收的*XXX*类型的ICMP消息数目。

*每个*ICMP_MIB_OUT*XXX*计数器的ICMP_MIB_IN*XXX*配对物：

ICMP_MIB_OUTMSG

已传输ICMP消息的数目。

ICMP_MIB_OUTERRORS

出错的ICMP传输的数目。没有使用。

ICMP_MIB_OUT*XXX*

除了刚才所列两种通用的计数器外，还有一种每ICMP消息类型的计数器。ICMP_MIB_OUT*XXX*会计算已传输的*XXX*类型的ICMP消息的数目。

ICMP_MIB_IN*XXX*计数器会在icmp_rcv中被更新。

ICMP_MIB_OUT*XXX*计数器会在icmp_reply和icmp_send中被更新（通过启用icmp_out_count）：

```
static void icmp_out_count(int type)
{
    if (type <= NR_ICMP_TYPES) {
        ICMP_INC_STATS(icmp_pointers[type].output_entry);
        ICMP_INC_STATS(ICMP_MIB_OUTMSGS);
```

```
        }
    }
```

在这两种情况下，对任何ICMP类型*t*而言，要递增的正确计数器的识别是通过和*t*相配的icmp_control数据结构的input_entry和output_entry字段。这些计数器的值会在*/proc/net/snmp*文件中输出。你可以用*netstat -s*来读取（当然，也可以用一些SNMP代理程序）。

把错误通知传给传输层

第二十四章“L4协议的注册”一节已知，当传输协议在内核注册时，会提供一个inet_protocol数据结构实例。该结构内包含一个函数指针err_handler，它会由ICMP协议调用，把通过入口ICMP消息所接收的错误通知传播给传输层。RFC 1122和1256指出，对主机和路由器而言，每种ICMP消息类型是否应该传播至传输层。需要把通知消息传给传输层的所有错误消息类型都是由icmp_unreach处理。在该函数尾端，会以err_handler通知传输层。

当传输层处理通知消息时，会使用icmp_err_convert数组（定义在*net/ipv4/icmp.c*中）把ICMP_DEST_UNREACH代码转换成套接字层比较容易理解的错误代码（参见*net/ipv4/udp.c*中的udp_err作为范例）。传输层会把错误代码传给和该错误相关联的套接字（因为ICMP有效载荷才得以识别出来）如“ICMP有效载荷”一节所述）。Raw IP套接字也会接到通知（通过raw_err）。表25-10列出icmp_err_convert所做的转换。注意，由隧道协议（例如，IPIP和GRE）所注册的err_handler函数可能会产生新ICMP消息（参见*net/ipv4/ipip.c*中的ipip_err作为范例）。

表25-10：icmp_err_convert的初始化

代码	内核符号	错误代码	致命（0=否，1=是）
0	ICMP_NET_UNREACH	ENETUNREACH	0
1	ICMP_HOST_UNREACH	EHOSTUNREACH	0
2	ICMP_PROT_UNREACH	ENOPROTOOPT	1
3	ICMP_PORT_UNREACH	ECONNREFUSED	1
4	ICMP_FRAG_NEEDED	EMSGSIZE	0
5	ICMP_SR_FAILED	EOPNOTSUPP	0
6	ICMP_NET_UNKNOWN	ENETUNREACH	1
7	ICMP_HOST_UNKNOWN	EHOSTDOWN	1
8	ICMP_HOST_ISOLATED	ENONET	1

表25-10：icmp_err_convert的初始化（续）

代码	内核符号	错误代码	致命（0=否,1=是）
9	ICMP_NET_ANOE	ENTUNREACH	1
10	ICMP_HOST_ANO	EHOSTUNREACH	1
11	ICMP_NET_UNR_TOS	ENETUNREACH	0
12	ICMP_HOST_UNR_TOS	EHOSTUNREACH	0
13	ICMP_PKT_FILTERED	EHOSTUNREACH	1
14	ICMP_PREC_VIOLATION	EHOSTUNREACH	1
15	ICMP_PREC_CUTOFF	EHOSTUNREACH	1

通过/proc文件系统调整

ICMP协议没有内核的编译选项可用，所有调整参数都定义在*net/ipv4/sysctl_net_ipv4.c*中，并通过*/proc*文件系统在目录*/proc/sys/net/ipv4*下输出：

icmp_echo_ignore_all

该标志由icmp_echo（处理送进来的ICMP_ECHO ICMP消息的处理函数）使用，以决定是否回复。这种过滤机制通常是为了安全缘故而由防火墙实施；然而，ICMP子系统也提供这种能力。

icmp_echo_ignore_broadcasts

当该标志设定时，传给广播地址的ICMP_ECHO消息会被忽略。参见第三十章“定向广播”一节的范例。该字段的值会由icmp_rcv检查。

icmp_ignore_bogus_error_responses

当此标志清掉时，带有广播目的IP地址的ICMP错误消息类型会被忽略掉。icmp_unreach会处理此标志。

icmp_errors_use_inbound_ifaddr

当本地主机传输ICMP错误消息时，该标志可用于改变源IP地址如何选取。当此标志没设定时，Linux从即将用于传输ICMP消息的接口选择源IP地址（参见第七部分）。当此标志设定时，Linux从接收处发ICMP消息传输的IP封包的接口选择源IP地址。

多数情况下，这两个接口都一样；但是，也有可能不同，例如，当两台主机可以非对称路径连接时（参见第三十章“路由基本元素”一节）。

icmp_ratelimit

icmp_ratemask

这两个变量是由ICMP使用，以对外出ICMP消息做速率限制（参见“速率限制”一节）。sysctl_icmp_ratemask只是位图掩码，它的每个位（从最低位起算）代表一种ICMP类型：如果相应于*XXX*类型的位有设定，则类型*XXX*的外出ICMP消息就会被施以速率限制。

表25-11总结变量和相关文件。

表25-11：/proc/sys/net/ipv4可用于调整ICMP子系统

内核变量	文件名	默认值
sysctl_icmp_echo_ignore_all	*icmp_echo_ignore_all*	0
sysctl_icmp_echo_ignore_broadcasts	*icmp_echo_ignore_broadcasts*	0
sysctl_icmp_ignore_bogus_error_responses	*icmp_ignore_bogus_error_responses*	0
sysctl_icmp_errors_use_inbound_ifaddr	*icmp_errors_use_inbound_ifaddr*	0
sysctl_icmp_ratelimit	icmp_ratelimit	1 * HZ
sysctl_icmp_ratemask	icmp_ratemask	0×1818[a]

a. 所给的每一位表示了ICMP的类型，在表25-1中给出了其类型，位域包括下列类型：ICMP_DEST_UNREACH、ICMP_SOURCE_OUENCH、ICMP_TIME_EXCEEDED和ICMP_PARAMETERPROB。

本章涉及的函数和变量

表25-12总结本章所介绍的主要函数、变量以及数据结构。

表25-12：本章所介绍的函数、变量、以及数据结构

名称	说明
函数	
icmp_init	对ICMPv4协议做初始化。参见“协议的初始化”一节
icmp_rcv	处理入口ICMP消息。参见“接收ICMP消息”一节
icmp_send	传输ICMP消息。参见“传输ICMP消息”一节
icmp_reply	
icmp_xmit_lock	取得和释放每CPU ICMP套接字的传输锁
icmp_xmit_unlock	

表25-12：本章所介绍的函数、变量、以及数据结构（续）

名称	说明
icmpv4_xrlim_allow xrlim_allow	对ICMP消息的传输做速率限制。参见“速率限制”一节
icmp_out_count	为已传输ICMP消息更新SNMP计数器
icmp_err_convert	把ICMP错误代码转成套接字错误代码。参见“把错误通知传给传输层”一节
ICMP_INC_STATS ICMP_INC_STATS_BH ICMP_INC_STATS_USER	递增用于持有ICMP消息的统计数据的计数器。参见“ICMP统计资料”一节
变量	
icmp_statistics	SNMP计数器。参见“ICMP统计资料”一节
数据结构	
struct icmphdr	ICMPv4所用的主要数据结构。参见“本章所提数据结构”一节
struct icmp_control	
struct icmp_bxm	
icmp_mib	计数器数组。参见“ICMP统计资料”一节

本章涉及的文件和目录

ICMP子系统只用到五个文件（两个是IPv4，两个是IPv6，而另一个则由这两个IP版本共享），如图25-14所示。

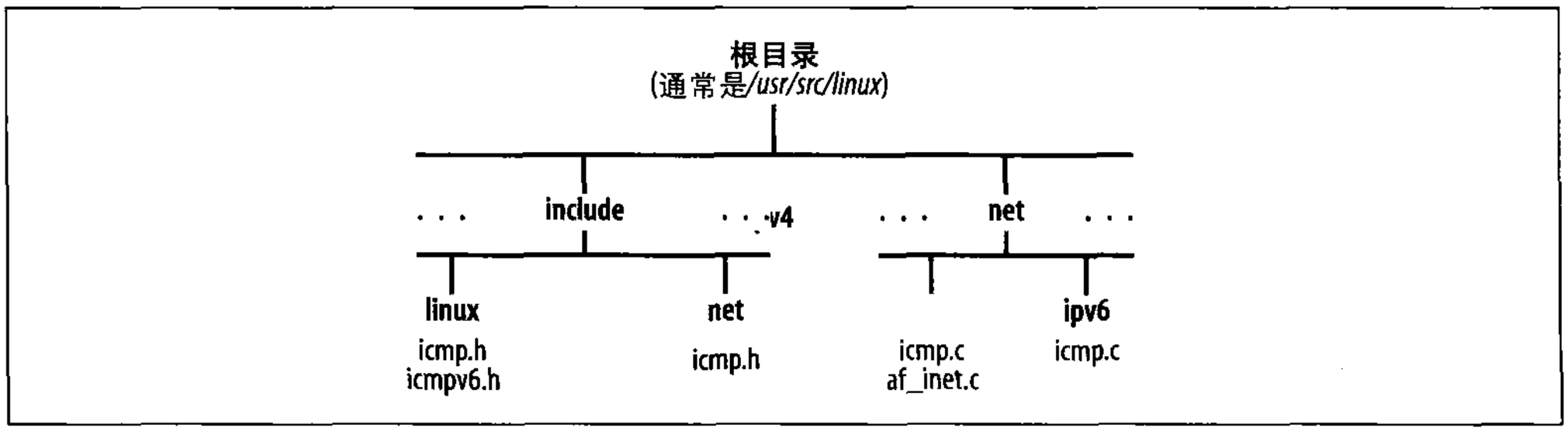


图25-14：本章涉及的文件和目录

第六部分

邻居子系统

第三层协议(例如,IP协议)将封包送到一个LAN中,然后利用第二层协议(例如,Ethernet协议)通过LAN的路由器到达终端应用程序所在的主机。在这个过程中,忽略了一个步骤。路由器和应用程序主机彼此是如何知道的呢? 若用专业术语讲,一台主机是如何由一个IP地址找到对应的L2地址(例如,一个MAC地址)呢? 给一个L3地址,找到其对应的L2地址的行为称为“L3地址的解析”。这个步骤是由邻居协议(*neighboring protocol*)完成的。

最常用的邻居协议是地址解析协议(ARP),在第二十八章中将概括介绍该协议。IPv6中相应的协议是Neighbor Discovery(ND)协议。但邻居协议的主要任务和原理,以及操作系统内部的邻居子系统都是通用的。

下面列出了每章主要讨论的内容:

第二十六章 邻居子系统:概念

本章描述为何要使用邻居协议和什么时候使用,以及它的主要任务。

第二十七章 邻居子系统:基础结构

本章讨论所有邻居协议共同的基础结构。

第二十八章 邻居子系统:地址解析协议(ARP)

本章介绍最常用的邻居协议:ARP协议如何使用基础结构,该协议也可能是读者最感兴趣的。

第二十九章 邻居子系统:其他问题

本章涉及命令行和用户空间接口(在*/proc*文件系统中的邻居子系统目录)

第二十六章

邻居子系统：概念

本章描述应该在什么时候使用邻居协议、使用该协议的原因，以及它的主要任务。只有慎重地了解整个邻居协议概况才能更好地理解特定的邻居协议（如ARP）。本章涉及以下一些问题：

- 通用邻居基础结构承担的任务。
- 缓存的重要性。
- 缓存中的邻居项所处的状态。
- 可到达检测和网络不可到达的检测（Network Unreachability Detection，NUD）。
- 代理的职责。

Linux 内核源码中使用的一些专业术语是遵循RFC 2461中描述的IPv6邻居发现模式定义的，参见“邻居协议”一节，但我们讨论邻居子系统时，尽量不涉及具体的协议实现。

通常所提到的一些术语，比如L2地址（L2 address）、二层地址（Layer two address）、硬件地址（hardware address）、MAC地址（MAC address）及链路层地址（link layer address）都是一个概念。本章中，我们主要用第一个术语。

邻居是什么？

如果一台主机和你的计算机连接在同一个LAN上（也就是说，你和这台主机通过一个共享介质相连或点对点直接相连），那么它就是你的邻居（neighbor），而且它有相同的L3网络配置。例如，在一个IP网络上，如果两台主机都连到同一个LAN中，而且各自至少有一个接口有相同的IP子网设置，那么就认为这两台主机是邻居。这两台主机可以通过与连接两者的物理介质（如Ethernet）相关的协议直接通信。定义邻居的另一种方式

是，一台主机到其邻居必须有且仅有一个L3跃点，并且它的L3路由表必须提供可以直接和其邻居通信的一条路径。不是邻居的主机间的通信必须通过网关或路由器。

如果两台主机被L2上的某个设备（如网桥）隔离，它们仍然是邻居。第四部分将对这个问题详细介绍，下面我们看一个基于IP网络的简单例子，如图26-1所示。

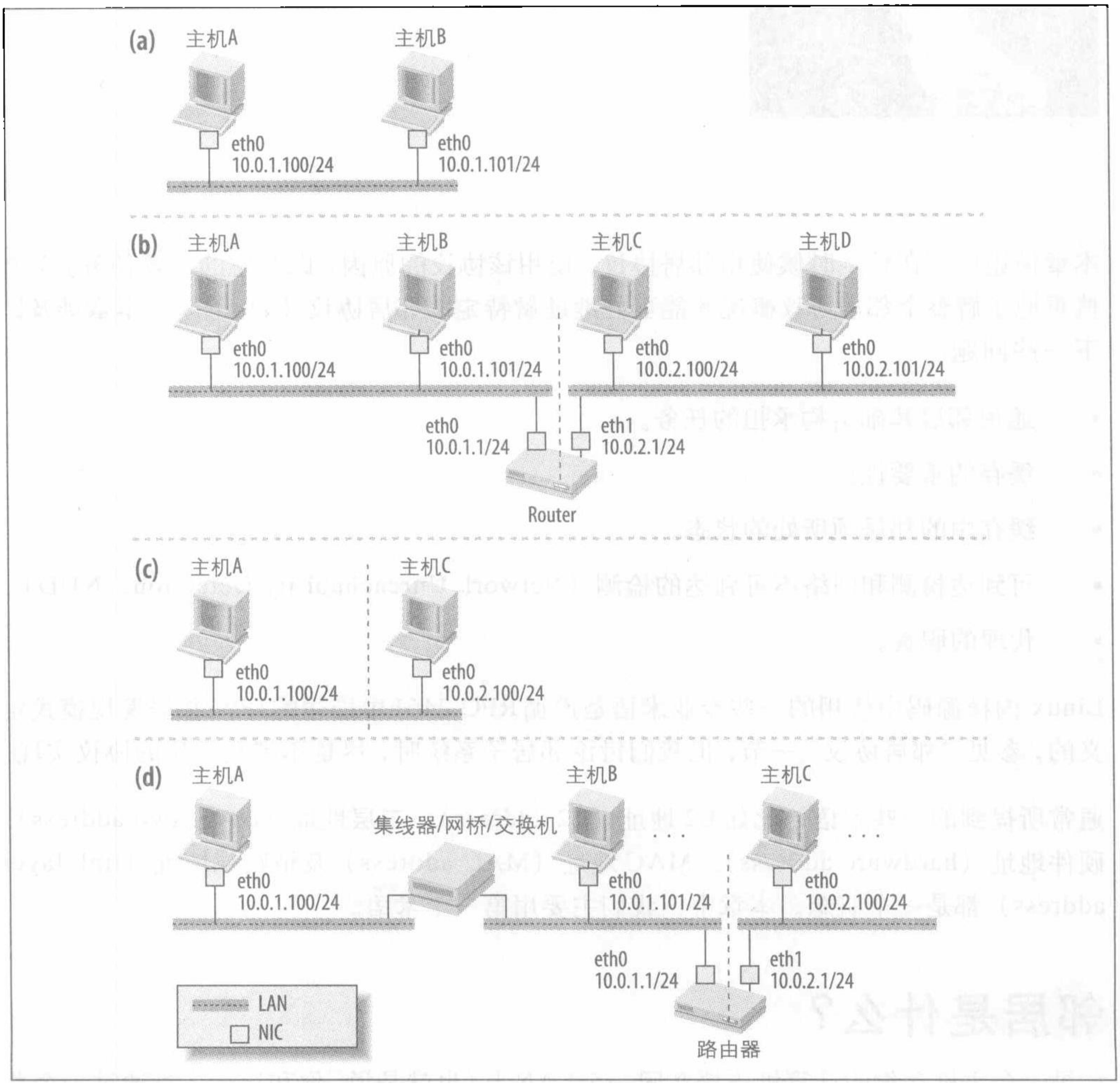


图26-1：邻居主机和非邻居主机

图26-1中的各种拓扑展示的是L3地址和L2地址之间的不同联系，每种拓扑访问邻居的方式不同：

图 26-1(a)

主机 A 和主机 B 是邻居，因为它们属于同一个 IP 子网 10.0.1.0/24，所以它们可以直接通信，彼此之间只有一个 L3 跃点。

图 26-1(b)

这个例子稍微复杂一些。主机 A 和主机 B 仍在同一个子网内，彼此之间可以直接通信。但是主机 A 和主机 C 分属于两个不同的 IP 子网，因此它们之间的通信要依赖路由器（假定它们都做了适当的配置）。这种情况下，主机 A 和主机 C 之间有两个 L3 跃点，因此它们不是邻居。

图 26-1(c)

图中所示的两台主机连接到同一个集线器上，但它们之间不能通信。尽管彼此会收到对方转发的数据，但由于它们被配置成不同的IP子网，因此无法在L3实现通信。主机 A 认为它只能访问子网 10.0.1.0/24 内的主机。它甚至不会尝试往不是该子网内的目的地址发送任何数据。要解决两者互访的问题有多种方式，在下一节我们将看到其中的一种解决方法，在其他章节有更多的方法介绍。

图 26-1(d)

图中所示的子网10.0.1.0/24实际上由两个LAN通过一个集线器或网桥组合到一起。在第十四章已经讲过集线器和网桥的区别，但从本章的角度看它们是等同的。注意到：用于合并两个 LAN 的接口上没有 IP 地址，这是因为这三个设备（集线器 / 网桥 / 交换机）都工作在 IP 层下面。

当两个主机彼此都只有一个 L3 跃点时，它们通常也只有一个 L2 跃点，如图 26-1(a)、26-1(b) 和 26-1(c)所示。但这并不是绝对的，如图 26-1(d)所示，主机 A 和路由器之间分别有一个 L3 跃点（因此两者是邻居），L2 跃点有两个。

进一步讲，在物理子网（LAN）和逻辑子网（如 IP 子网）之间不总是一对一的，如图 26-2(a)所示。在一个 LAN 上可以存在多个 IP 子网，或者多个 LAN 位于同一个 IP 子网内。图 26-1(c)所示的就是同一个 LAN 上有两个 IP 子网的例子。图 26-1(d)所示的例子则是通过集线器相连的两个 LAN 位于同一个 IP 子网（图中左半部份）内。前一个例子不常见，后一个例子在配置 ARP 代理或网桥的情况下，经常被使用。在第二十八章的“普通处理的结尾”一节中可以看到配置 ARP 代理的范例，在第四卷则有网桥的范例。

图26-2(b)所示的两组主机位于不同的IP子网内。即使各个组内的主机共享同一个LAN，但它们彼此之间要直接通信仍必须通过路由器，它同时监听两个LAN。路由器可以有两块不同的网络接口卡（NIC，如图 26-2(b)所示），或者是配有多个 IP 信息的单块 NIC。下面这种情况很少用到，但也有可能使用。例如，要临时访问某个设备或者某个设备坏了。比如，你使用图 26-2(b)所示的情景，并且 LAN1 无法运行，你可以把 LAN1 中的

主机移到LAN2中［包括路由器的*eth0*接口（注1）］。这样，不必改变原来在LAN1中的主机的IP子网设置，一切就能正常运转。原来位于LAN2中的主机仍然可以通过路由器访问其他主机。尽管这种情景不常出现，但内核必须能够合理地处理。该情景中涉及的一些设置，尤其当路由器使用单接口来访问两个子网时（也就是说，没有*eth0*并且其地址被设置到*eth1*上），将在第二十八章“调整ARP选项”一节中讲述。

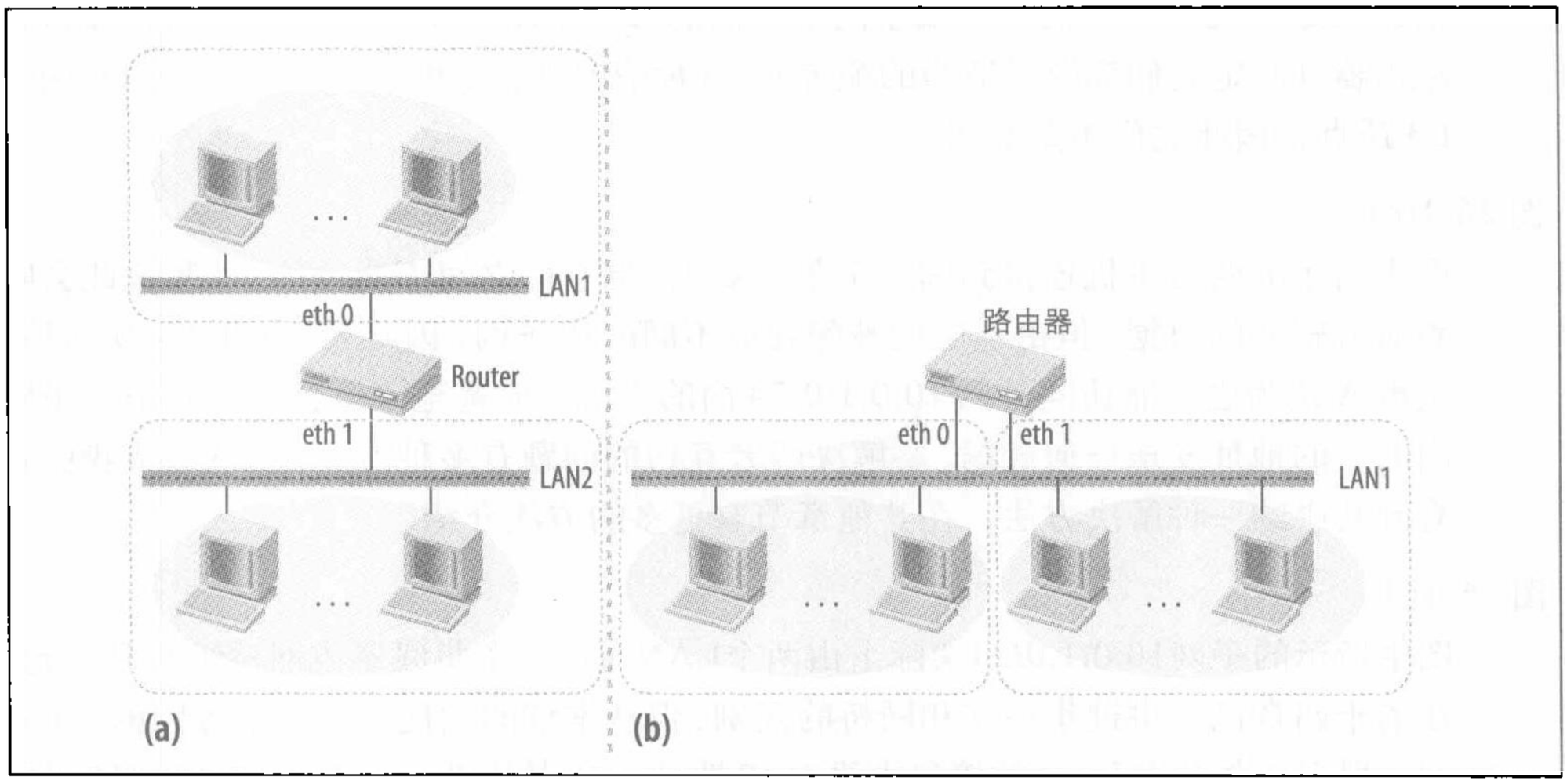


图26-2：(a)IP_子网-LAN 1:1; (b) IP_子网-LAN *n*:1

在本章剩余部分我们将不涉及图26-2(b)所示的情形，但你应该记住配置一个类似的网络是可能的，并且也是可行的，是符合规定的。

需要邻居协议的原因

在本节中，我们将看到需要邻居子系统的根本原因。其根源就是网络体系的层次描述和共享介质的存在，比如Ethernet。

L3地址何时需要被转换为L2地址

二层网络协议（Ethernet、802.11无线、令牌环Token Ring、点对点等）和三层网络协议（IP或专用协议）的区别是，为了邻居间可以交换数据存在着许多不同的二层协议，而三层路由协议传输数据时不必关心使用哪种物理介质。在两个系统之间的高层协议应该能够使用同样的软件来发送封包，而不必关系它们是Ethernet连接还是点对点连接。

注1：　另一种方法是不移动路由器上面的接口，只是简单地将IP添加到下面的接口。

图 26-3 给出了在不同情况下邻居子系统所做出的不同反应。

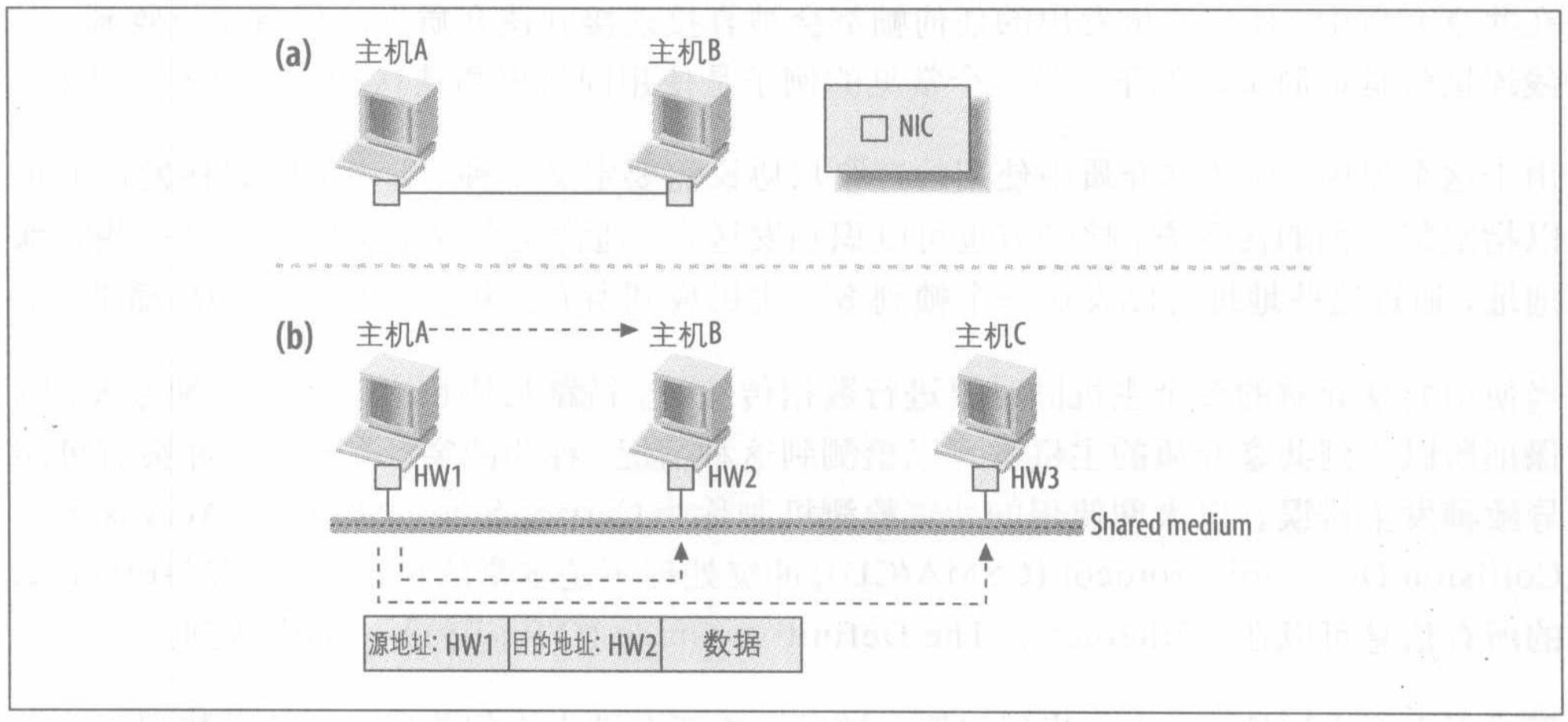


图 26-3：点对点连接和共享介质连接

图 26-3(a)所示的是点对点连接，例如拨号方式。若该连接运行在一条半双工的介质上，那么其二层协议相当简单，只处理类似错误检测和通信切换。邻居协议也是最简的，因为它到只需简单地调用二层协议。也没有必要选择要封包要到达的邻居。

图 26-3(b)所示的场景比较复杂：主机位于以太网中或其他使用广播包来运作的共享介质网络上。如果主机 A 要给主机 B 发送数据，A 必须把数据放到线缆（若是无线网络，则是无线电波）上，让所有在这个共享介质上的系统都可以收到该数据。A 必须在数据中指明一个 L2 地址，以便某个主机知道这个数据是给它的。其他的主机会检查这个地址并忽略这个数据。邻居协议负责为封包选择与 L3 地址相配的 L2 地址。

如果主机 A 和主机 B 被网桥分割，网桥根据封包中的 L2 地址直接将其送到正确的主机（注 2）。邻居子系统根本无需理会。实际上，网桥对邻居子系统是透明的。

通常在 L3 地址和它相应的 L2 帧之间是一对一的关系。有多个 L3 地址的系统（通常是路由器）具有多个物理接口以便保持 L3 地址和 L2 地址间的一一对应。但在后面的“特殊情况”一节中，可以看到 L3 的多个多播地址可以映射到同一个 L2 地址上。同样，一个物理接口配置多个 IP 地址也是可能的。

注 2：我们在第四部分看到网桥和交换机尽可能地直接将帧送到正确的主机，这样可以减少帧在 LAN 中的每个主机间无用的转发。

共享介质

在共享介质中，任一主机发出的任何帧都会被直接连接到该介质上的所有主机收到。无线连接就是最简单的例子，另一个常见的例子是使用同轴电缆连接的10-base2以太网。

由于这个原因，在共享介质中使用的链路层协议需要定义一种寻址机制，以便发送方可以指定每个帧的接受方，接受方也可以识别发送方。通常这个寻址机制也定义一些特殊地址，通过这些地址可以发送一个帧到多个主机或到所有主机：多播地址和广播地址。

当使用共享介质的多个主机同时要进行数据传输时，链路层协议必须提供一种方式，以保证所以连到共享介质的主机都可以检测到这种情况（称为冲突*collision*），冲突有可能导致帧发生错误。以太网使用的冲突检测机制称为Carrier Sense Multiple Access with Collision Detection protocol (CSMA/CD)。冲突处理不是本章的讨论内容。有关Ethernet的所有信息可以在《Ethernet:， The Definitive Guide (O'Reilly)》一书中找到。

而对于点对点连接，例如，串行线路，被设计成两个端点之间的通信。这种情况下，不需要链路层地址来标识源端点和目的端点。两个端点可以用全双工或半双工的方式进行通信，这取决于二者是共享同一个线缆还是各自分别有自己的线缆。这种连接也不需要冲突检测：两个端点彼此各自有条线缆（全双工）或者有一种机制保证每一端都使用共享线缆时是独占的。因此，通过点对点介质连接的两个主机不需要邻居协议。

Ethernet 最初的设计就是在共享介质上工作，允许多个主机共享同样的介质，并依靠CSMA/CD解决冲突。最初是使用同轴电缆。但随着时间的推移，同轴电缆由于多种原因被非屏蔽双绞线（UTP），或称RJ-45线取代。UTP允许Ethernet接口配置为全双工或者半双工模式，因为UTP线缆中有足够多的线允许两端同时通信。只有在两个以太网接口间是点对点连接时，才能使用全双工模式。在这种情况下，连接的两端都可以指定一条线用作传输，另一条用作接收，因此不需要CSMA/CD。

如今，实施Ethernet局域网主要使用交换机（switch）（注3）：每个主机都用一根UTP线缆连到交换机上。这种情况下，你可以将以太网接口配置为半双工模式，此时CMSA/CD协议将处理交换机端口和主机Ethernet适配器之间的通信冲突；或者将这两个接口配置为全双工模式，允许主机和交换机同时进行数据传输。通信的两个端点必须采用同样的双工配置。大多数情况下，并不需要对通信的两端进行明确的双工配置，工作模式检测机制将自动进行处理。

注3： 在本书中，网桥和交换机被看作是同一类型的设备。参见第四部分的详细介绍。

注意，由主机产生的帧永远不会以交换机为目的地（当然也有例外）；一台主机要访问连接到同一个交换机上的另一台主机时才会用到交换机。因此，尽管接口在全双工模式下并不需要CSMA/CD，你仍需要源地址和目的地址，以及邻居协议。这也说明实际的共享介质（例如同轴电缆）能提供的多播和广播功能，在交换机上通过其他方式也能提供：当交换机收到一个目的地址是链路层多播或广播地址时，它将其复制到除接收到该帧的端口外的其余全部端口。在第四部分我们能看到交换机实际上还能完成更多工作。

现代的局域网主要是用Ethernet 交换机实现的，主机通过点对点链路（UTP）连接到交换机上。在新设计的以太网标准中CSMA/CD协议的重要性已降到次要地位。新的以太网标准速度将更快，因为CSMA/CD已不是必要的，或者干脆将其去掉。

表26-1给出了支持CSMA/CD协议的Ethernet种类。注意到，尽管千兆以太网（Gigabit Ethernet）主要用于全双工的点对点连接，但其仍然支持CSMA/CD（共享方式）。10 Gigabit Ethernet主要作为城域网（WAN，和LAN相对）的标准，完全不支持CSMA/CD，并且只能使用光纤组成点对点链路。表26-1中的每个元素实际都要多个不同的实现，因为它们不在本书的讨论范围，所以这里没有涉及。

表26-1：以太网种类和点对点/共享介质能力

Ethernet 种类	仅支持点对点	支持共享（也就是说支持CSMA/CD）
Ethernet (10 Mbit/s)		×
快速 Ethernet (100 Mbit/s)		×
千兆 Ethernet		×
10 Gigabit Ethernet	×	

为何没有足够的静态地址

在第十三章我们已经知道了L2和L3地址及协议所担任的角色。L3地址是逻辑地址，例如IP地址，这意味着任一合法L3地址可以分配给任一物理接口。而L2地址是被绑定到NIC上，是不可改变的：它们由生产该接口的厂商指定，并且是全球唯一的。但是，大多数NIC可以通过类似*ifconfig*的工具设置为任意的L2地址。当处理本地IEEE 地址时，在第十三章有详细介绍，这种方式也许有用。但当你把一个NIC的L2地址变为一个不属于你的地址时，可能会有些风险：你无法确保这个地址是唯一的，因此也无法保证在使用L2地址识别NIC的共享介质上正常工作。一般来说，只有有经验的高级管理员出于特殊目的才会这么做，例如，配置虚拟服务器或者设置网络的高可用性（high-availability）。

因为L3地址是逻辑地址，所以它们可以随意改变。下面是一些需要改变L3地址的情况。同样，这些情况下L3地址到相应的L2地址的映射也发生变化。

动态配置

在IP网络中，通过协议的方式（例如，DHCP）可以给一台主机指定动态IP地址。同一个主机每次请求得到的IP地址可能会不同，但是硬件地址已经固化在Ethernet网卡或无线网卡中，因此L3到L2的映射也必须相应地更新。

故障接口的重新配置

一旦替换了主机的NIC，那么它的L2地址也就改变了，但是管理员可能更愿意在网络中保持同样的逻辑配置，因此L3地址是不变的。

L3地址的移动

某台服务器可能无法工作，这时就要求另外一台不同的服务器处理同样的流量。这意味着旧的L3地址应该被设置到新服务器的新接口上。如果该主机换了一个网络接口，管理员想保持其L3地址不动，也会发生这种情况。

为了将上述变化从L2和L3隔离出来，因为它们有大量的工作要做，无法处理所有可能出现的情况并缓存相关数据，因此需要一个协议来管理L3地址到L2地址的关联。这些任务都是由本章讨论的邻居协议完成的。

特殊情况

有时候L3地址到L2地址的映射不需要任何协议就能完成，例如下面一些情况：

- 仅有一台主机能把数据发送到点对点介质上。例如，拨号连接或者管理员临时将一个系统连接到要监控的主机上。此时，在L2根本没有地址映射机制。（但是，在某些环境中点对点连接也使用L2地址）。
- 可能有一些特殊的L3地址，它们与L2地址的关联通过一种简单的规则获得；因为这些地址不会产生歧义，没有动态的地址分配，也不需要任何协议。
- 多播地址可以不需要任何协议实现静态转换。在IPv4/ARP网络中，`arp_mc_map`函数负责解析多播地址，若接口设备是Ethernet NIC，该函数会依次调用非常简单的函数`ip_eth_mc_map`。在函数`ip_eth_mc_map`中的映射不需要任何协议，只是根据图26-4所示和下面介绍的规则来实现：
  - 最高24位指定由IANA分配的静态地址01:00:5E。
  - 第23位（低24位的最高位）设为0。
  - 剩余的低23位直接复制自相应IP地址的低23位。

注意：相同的Ethernet 多播地址可以设定多个IP地址（因为IP地址的最高9位没有使用）。

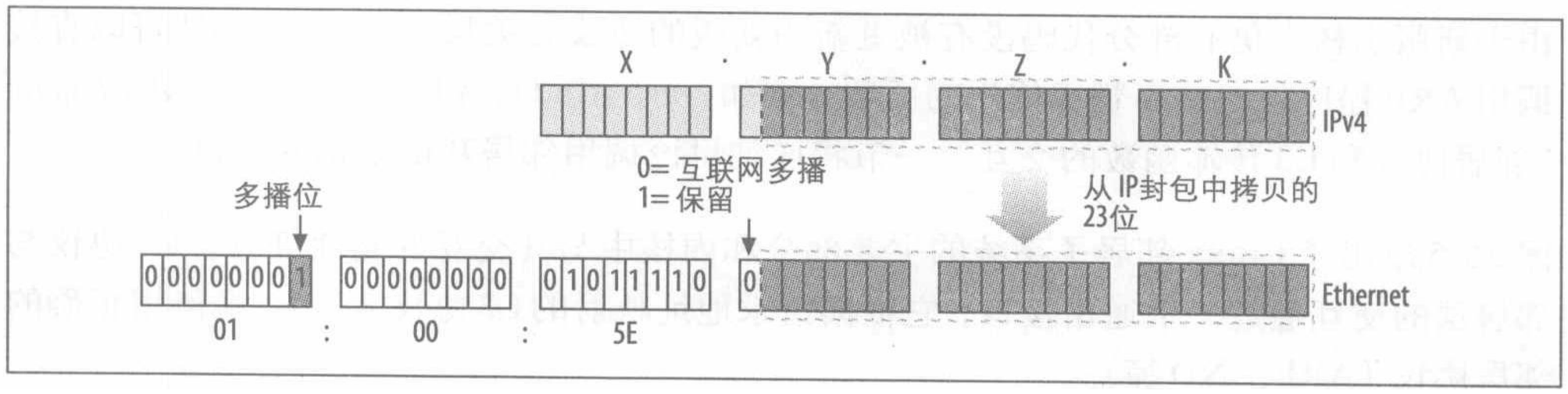


图 26-4：一个IPv4 的多播地址到以太网多播地址的映射

- 广播地址（IP子网广播）静态解析为链路层广播地址（以太网是FF:FF:FF:FF:FF:FF）。如果需要，你可以为每个设备的L2广播地址设置一个确定的值。

Solicitation 请求和应答

当无法使用上节提到的静态翻译完成L3到L2的地址映射时，就需要邻居协议来完成这个映射。不同的邻居协议有不同的工作机制。但是，所有的邻居协议都使用下面的两个术语，本书就对这两个术语进行讨论。

Solicitation 请求（也叫做邻居 Solicitation）

这个术语表示在网络中发出一个封包并询问所有的主机，是否有主机知道这个L3地址对应的L2地址。根据不同的协议和实际情况，该请求可以以单播、多播或广播的方式发送出去。

Solicitation 应答（也叫做邻居 advertisement）

一般来说，这个术语指发出一个封包作为solicitation请求的应答。但该封包也可以独立生成（参见第二十八章“免费ARP”一节中的例子）。在正常条件下，与目的L3地址关联主机会产生这个应答，但也有可能有另一台主机替它产生应答（参见“邻居协议代理”一节）。该封包通常使用单播方式发送，特殊情况时，也可以使用广播。

Linux 实现

早期的Linux 内核让L3协议直接调用邻居协议提供的函数，因此，IPv4子系统直接与

ARP程序互相作用。在新版本的内核中，开发商分析了不同的协议的共同的要求，并将其抽象为一个新层，称作邻居基础结构（neighboring infrastructure）。

由于新版内核中仍有部分代码没有被更新为新版的协议无关层，所以，你仍然可以直接调用ARP程序中一些不赞成使用的函数（例如，arp_find），但它们是例外。第27章的“邻居协议和L3传输函数的交互”一节将详细讨论调用邻居基础结构的接口。

图26-5给出了Linux邻居子系统的主要部分和内核中与其交互的其他部分。L3协议与邻居层的交互通过一个通用接口，它根据请求地址映射的L3协议（注4）来使用正确的邻居协议（ARP、ND等）。

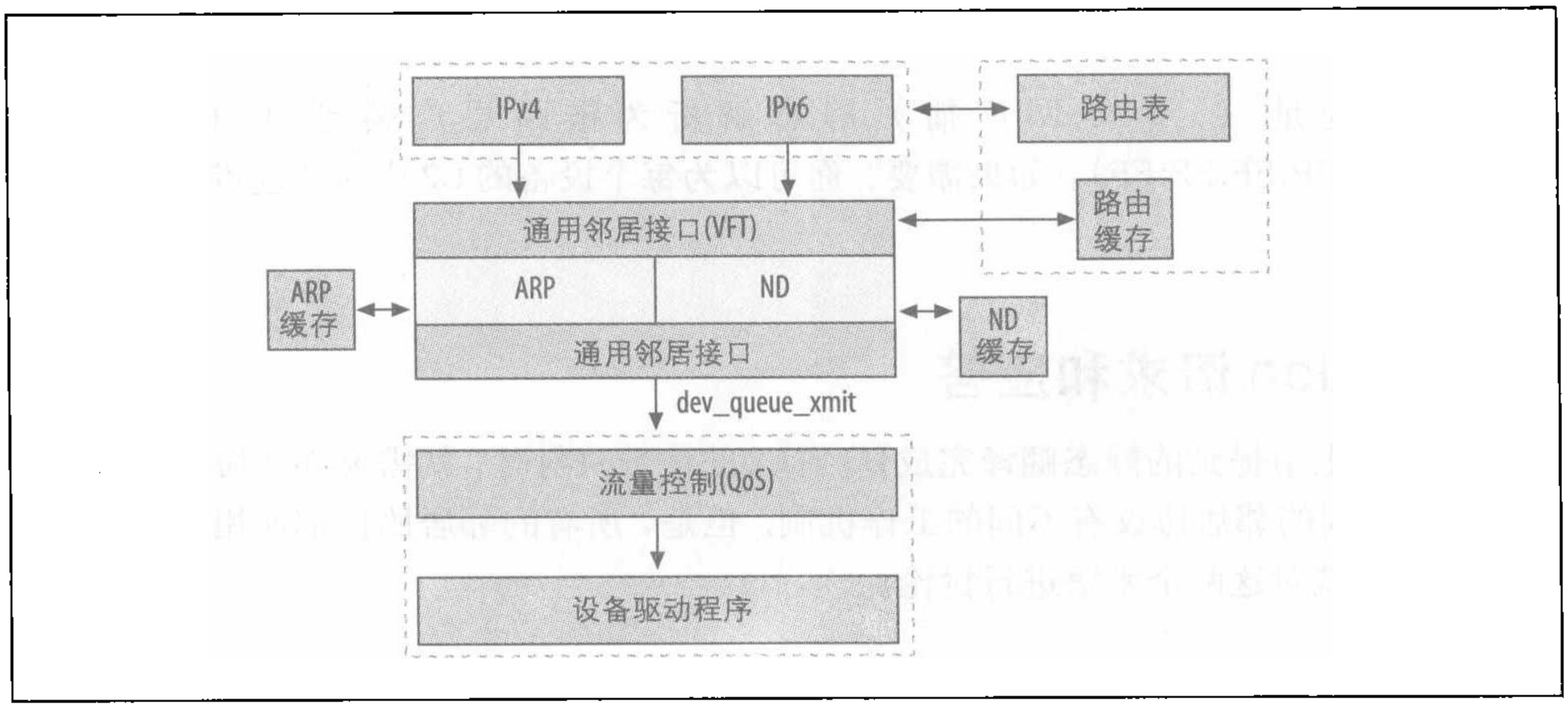


图 26-5：大蓝图

当传送一个封包时，执行下面的步骤：

1. 本地主机的路由子系统选择L3目的地址（下一个跃点）。
2. 根据路由表，如果下一个跃点在同一个网络中（也就是说下一个跃点是邻居），邻居层就把目的L3地址解析为跃点的L2地址。这个关联被放入缓存以便将来再次使用。这样的话，如果某个应用程序短期内发送多个封包给另一个应用程序，那么只在发送第一个封包的时候使用一次邻居协议。
3. 最后，一个函数，比如dev_queue_xmit（第十一章讲过）负责完成发送，将封包传递给流量控制（Traffic Control）或者QoS（Quality of Service）层。尽管图

注4： 图中没有包括DECnet和ATM，因为它们不属于本书的讨论范围。

26-5 中只给出了 dev_queue_xmit，但邻居层实际也能调用其他函数（大部分用 dev_queue_xmit 进行封装），本章的后面将会讲到该内容。

注意到，dev_queue_xmit函数是在当要传送的封包准备发送出去时才被调用的。因此，如果需要L2头部信息的话，邻居协议必须在调用该函数前加到封包上。点对点连接、广播、多播都不需要 L2 头部信息，因而也不需要 L3 到 L2 的映射。上述几种类型的传输在“特殊情况”一节有介绍。其他的传输使用共享介质，因此就需要一个 L2 头部信息。该信息可能来自邻居子系统的缓存，或是来自邻居系统向网络发出的请求。

邻居协议

在现在的 IP 网络中使用两个邻居协议。绝大多数系统使用 IPv4 中的 ARP 协议。另一个更通用的协议是邻居发现协议（Neighbor Discovery，ND），它是为 IPv6 开发的。为了能在专有网络使用，Linux 内核中支持其他的邻居协议，比如 DECnet 使用的邻居协议。但由于这些协议使用范围比较窄，在本书中对其不予讨论。

尽管 ARP 被认为是一个 L3 协议，但其任务已被 IPv6 的设计者们移到了 L4。如图 26-6 所示，ND 协议在 IPv6 中是作为 Internet 消息控制协议（ICMP）的一部分。这样安排是根据多年来使用IPv4的经验。ND协议有许多优点，包括利用L3的一些特征，例如，IPsec 加密。在第二十八章的“ND（IPv6）对 ARP（IPv4）的改进”一节将介绍 ND 和 ARP 的主要区别。

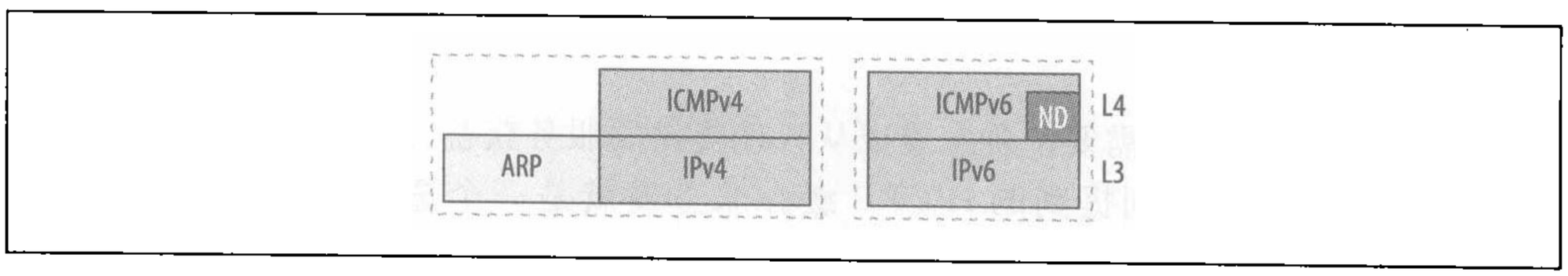


图 26-6：ARP/ND 协议在网络栈中的位置

前面已经提过，为了减少系统开销和所有邻居协议使用的相似服务间的代码重复，Linux 提供了一个通用基础结构。通用邻居基础结构提供的服务可由不同协议根据其需要适当裁剪。下面是通用基础结构提供给这些协议的部分服务：

- 为每个协议提供一个缓存来存放 L3 到 L2 转换的结果。
- 提供在缓存中添加、删除、改变和查找一个特定映射项的函数。查找函数必须要快，因为它会影响整个系统的性能。
- 为每个协议缓存的数据项提供一种老化机制。

- 当缓存已满时并且正好要创建新的映射项时，提供选择替换策略。
- 为每个邻居提供一个请求队列。当准备要发送一个封包且其L2地址还不在地址缓存中时，就必须把这个封包放到缓冲区中，直到发出 solicitation 请求，并收到应答。参见第二十七章的“排队”一节。

为了让每种协议都适应邻居子系统的行为，Linux中定义了一组占位符（placeholder）或虚函数，这样每个协议都可以在其中插入它想用的函数。这与Linux允许用户定制内核的方式有些相似，邻居层也提供了一组可调参数对其进行配置。配置这些参数可以通过用户空间命令、/proc文件系统或协议本身。最后，所有协议访问缓存的函数都是相同的，但不同的协议可能会用不同字节长度的关键字（地址）。因此，基础结构提供了一种通用方式来定义关键字使用的类型。后面的章节我们会详细讲述刚才提到的这些要点。

每个协议的运行和配置都是独立的，与其他协议无关。在二十七章的“协议初始化和清理”给出了一个邻居协议如何向内核注册及注销。

邻居协议代理

若一台主机将发往另一台主机的流量拦截下来，并且代替后者处理这些流量，那么这台主机就是一个代理。这里的代理并不包括发起中间人攻击（man-in-the-middle）的主机。相反，代理最常见的例子就是一台HTTP缓存服务器，它将发往受访问的web服务器的请求截断，然后把保存自己的缓存中的那些服务器上的网页提供给请求者，这种方式可以减少网络流量。

如果主机和应用程序不需要精确配置成从代理提供的服务获益，这种代理称为透明代理（transparent）。刚刚提到的HTTP缓存服务器就是一个透明代理的例子。如图26-7所示，同样的服务可以由透明代理提供，也可以由非透明代理提供。图26-7给出了两个使用HTTP代理的例子：

- (a) 一个局域网可以通过安装在路由器上的代理来访问 Internet。来自该局域网内主机的所有浏览器请求都通过路由器，因此管理员可以让路由器来截断并代理所有的HTTP请求。这就是透明代理，因为在主机B上不需要进行任何配置或者使用任何特殊的浏览器。
- (b)主机 B 的浏览器被设置为使用代理服务器（Proxy ）来访问 Internet。代理服务器只有在其自身缓存不命中时，才会使用路由器访问 Internet。

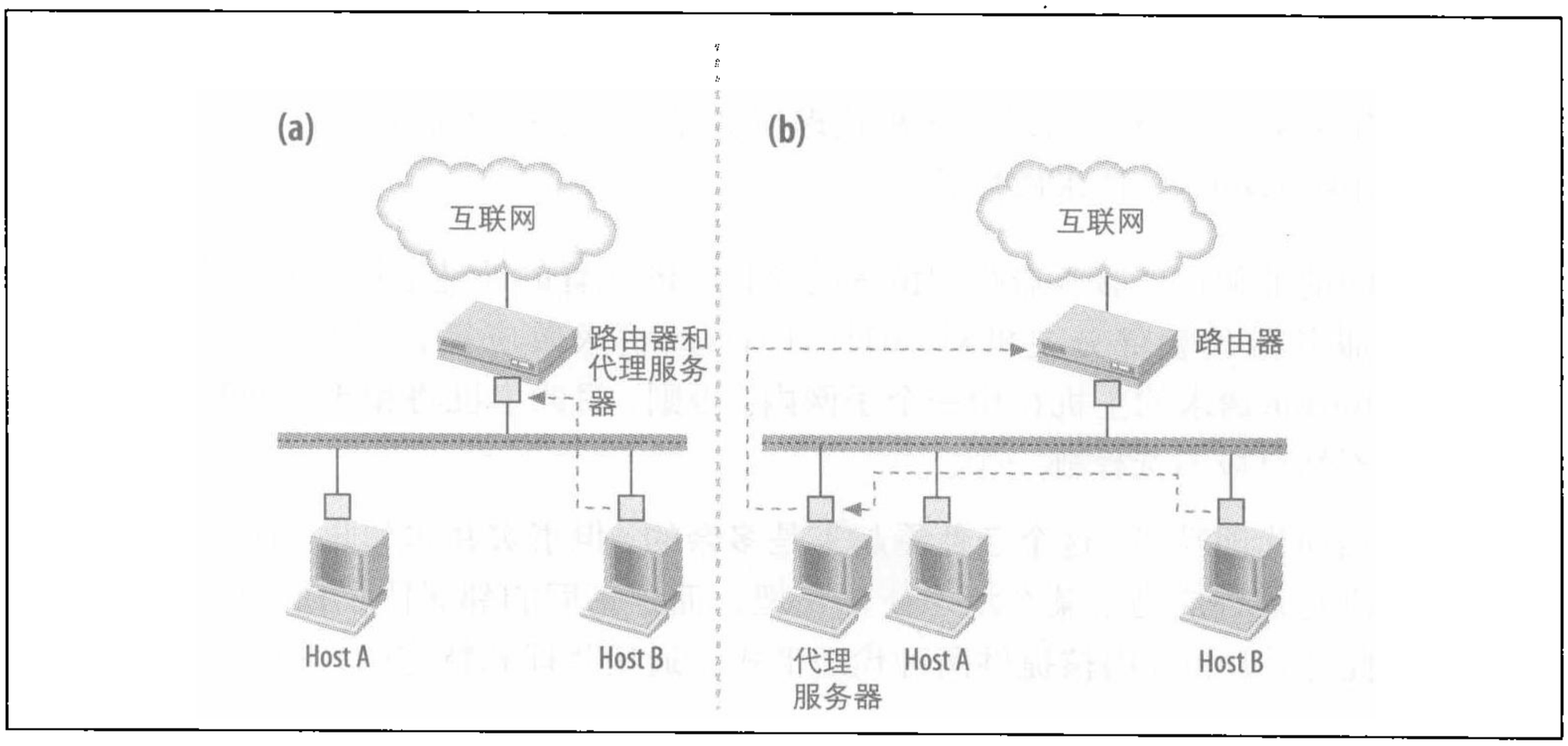


图 26-7：(a) 透明代理；(b) 非透明代理

当然，也存在其他的选择。例如，代理有可能是一台单独的机器，而路由器被设置成将HTTP 请求转发到该代理。因为代理是个很大的主题且超出了本书范围，所以在此不予详细讨论。邻居协议使用的代理通常是透明的。

前面的例子展示的是一种普遍的代理类型：HTTP 代理或 Web 代理。现在考虑代理与本章内容的关系。邻居协议的代理服务器是一台主机，对请求不属于自己的地址的solicitation 请求，它能够代替拥有这些地址的主机做出应答。因为有了代理，位于不同LAN 中的主机可以互相通信，就好像它们在同一个 LAN 中。

例如，ARP 代理通常用在 IPv4 网络中，实现从扁平型网络到子网型网络的地址转换。通信主机不需要特殊的协议，也不需要特别配置，因为代理对它们来说是透明的。但是，如果代理服务器崩溃了，通过代理连接的主机间的连通性也就消失了。如果使用多个代理服务器，就能够减轻上述影响。此时，一台主机的 solicitation 请求可能会收到多个solicitation 应答。只要选择最先到达的应答，就能够确定最快且负载最小的代理服务器。

代理的使用也简化了被代理服务器接管的主机的配置；在第二十八章中的“ARP 代理服务器作为路由器”一节有一范例。

在 Linux 内核支持的邻居协议中，只有 IPv4 和 IPv6 可以使用代理。通用邻居基础结构由两个协议共享使用，每个协议都可以根据自己的需求来适应代理。

在第二十七章的“担任代理”一节，我们将看到协议无关的代理模块的详细实现（计时器、队列等）。在第二十八章的“ARP 代理”一节，我们将看到 IPv4 和 ARP 的使用代理的特殊情况。

使用代理的条件

并不是所有的 solicitation 请求都会被代理服务器处理。代理服务器只有在满足下列条件时才会对 solicitation 请求做应答：

- 请求的地址和代理服务器收到请求的接口上的配置的地址不属于同一个子网。因为代理服务器代替部分主机对 solicitation 请求做应答，这些主机不能和发出 solicitation 请求的主机在用一个子网内。否则，目的主机将和代理同时应答，发送方将不知道应当选择哪一个。
- 必须启动代理功能。这个条件看起来是多余的，但事实并非如此。有几个判断条件可以决定是否应当对某个请求使用代理，而且不同的邻居协议使用的判断条件不同。此外，Linux 内核提供两种代理形式：通用代理和特定的代理。

基于设备（Device based）

该设备接收的所有有效请求都会被处理。这是IPv4中最常用的情况，IPv6中不使用。

基于目的地址（Destination based）

在决定是否要进行代理时，同时考虑目的地址和设备两个属性。也就是说代理可以对选定IP地址的请求做出应答。基于目的地址的代理是IPv6网络的标准，但在IPv4中也是可以使用的。

图26-8给出了两种代理方式之间的优先次序。当一个主机收到一个对本地子网外某个地址的solicitation请求时，这台主机若是启动了代理功能就可以处理它。首先邻居子系统检查设备上的代理功能是否全局使能，如果没有，再检查该设备是否被配置为可以代理那个特定地址。

- 在接收请求的代理服务器上已开启转发功能。

  因为代理服务器是插入到主机之间的，所以它必须接收两个端点间转发的流量（注5）。

ARP solicitation 请求总是发往L2广播地址。这就保证了所有共享介质上的主机都可以收到它。这样，代理服务器不需要把其任何接口配置成混杂模式，就可以截取地址是它代理的主机的请求。当进行可到达性确认时（参见“可到达性确认”一节），ARP使用单播而不是广播。

ND 协议使用 L3 广播地址处理 solicitation 请求和应答。当路由器想代理特定的一个 IP 地址时，它必须接收对应的 L3 多播地址。

注 5：在代理服务器上启动代理功能并不表示自动启动转发功能。这两个功能是独立配置的，但代理需要转发工作正常。

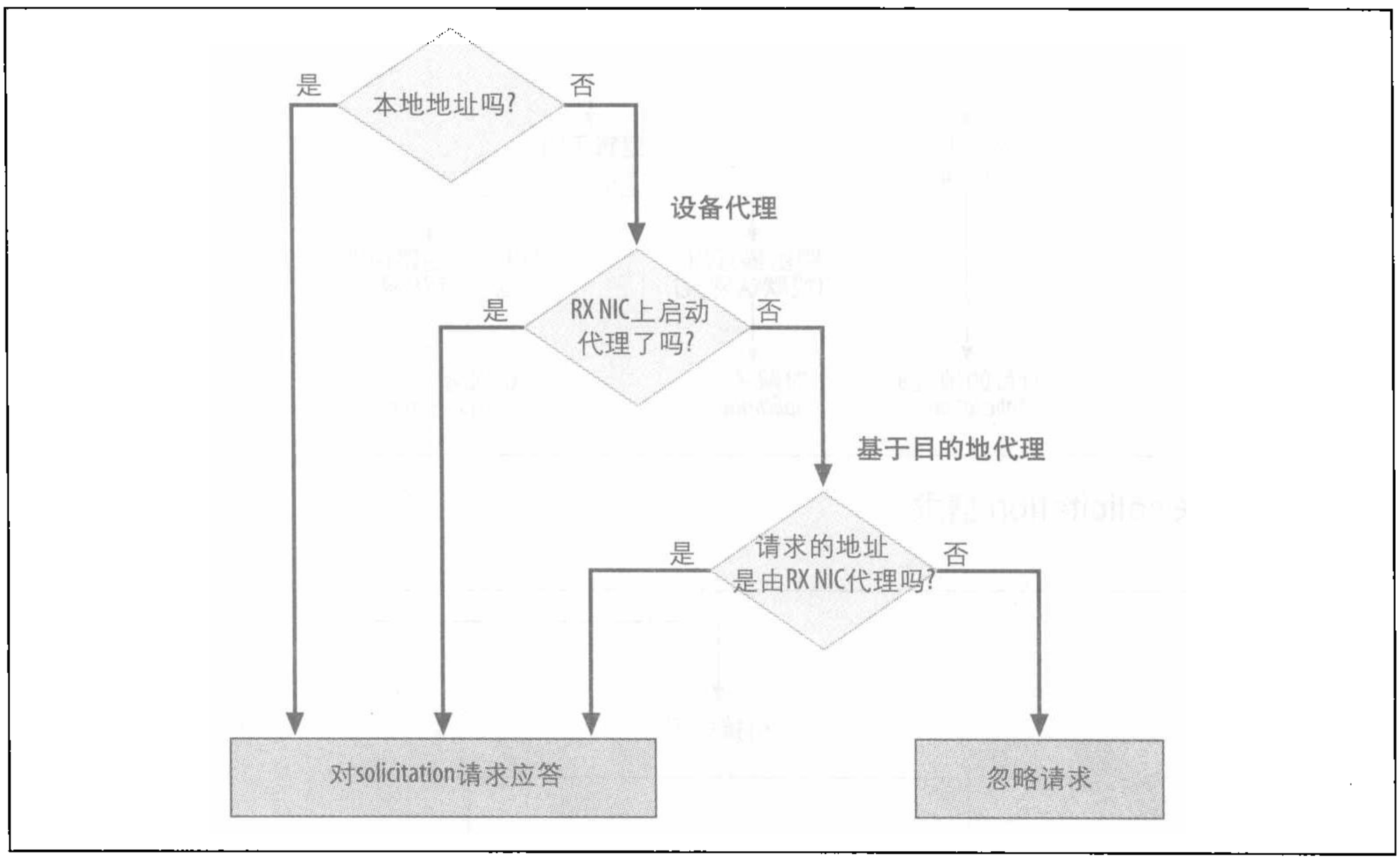


图 26-8：设备代理和目的地址代理间的优先级

何时发送和处理 Solicitation 请求

本节中我们将看到，根据接收请求的主机的配置情况和网络的物理拓扑结构，什么时候应该处理 solicitation 请求。图 26-9 给出了导致主机发出 solicitation 请求的一些因素，图 26-10 给出了接收到该请求的 Linux 主机用于决定是否对其处理的最常见的因素。为了能够说明接收者决定过程的潜在复杂性，图 26-10 假定接收主机同时具有代理和桥接功能（注 6）。去掉任何一个功能都会简化处理流程的复杂度。图 26-10 也假定代理功能是基于设备的。基于目的地址的代理决定过程与此类似，只是少一个步骤。图 26-10 中给出了代理服务器的情况和没有实施任何代理的普通主机的情况："代理开启"指一台代理服务器，而"代理关闭"表示一台普通主机。

上述分析不涉及具体协议，ARP 相关的细节可参见第二十八章。

当接收到 solicitation 请求的主机启动桥接功能时，该主机对其不进行处理，而是按照网桥配置，将其从合适的接口上转发出去。桥接功能在邻居协议之前起作用，因此它能看

注 6：在图 26-10 中增加桥接是为了显示在邻居协议之前是桥接在处理请求，因此邻居协议并不总是能看到入口 Solicitation 请求。网桥的详细情况见第四部分。

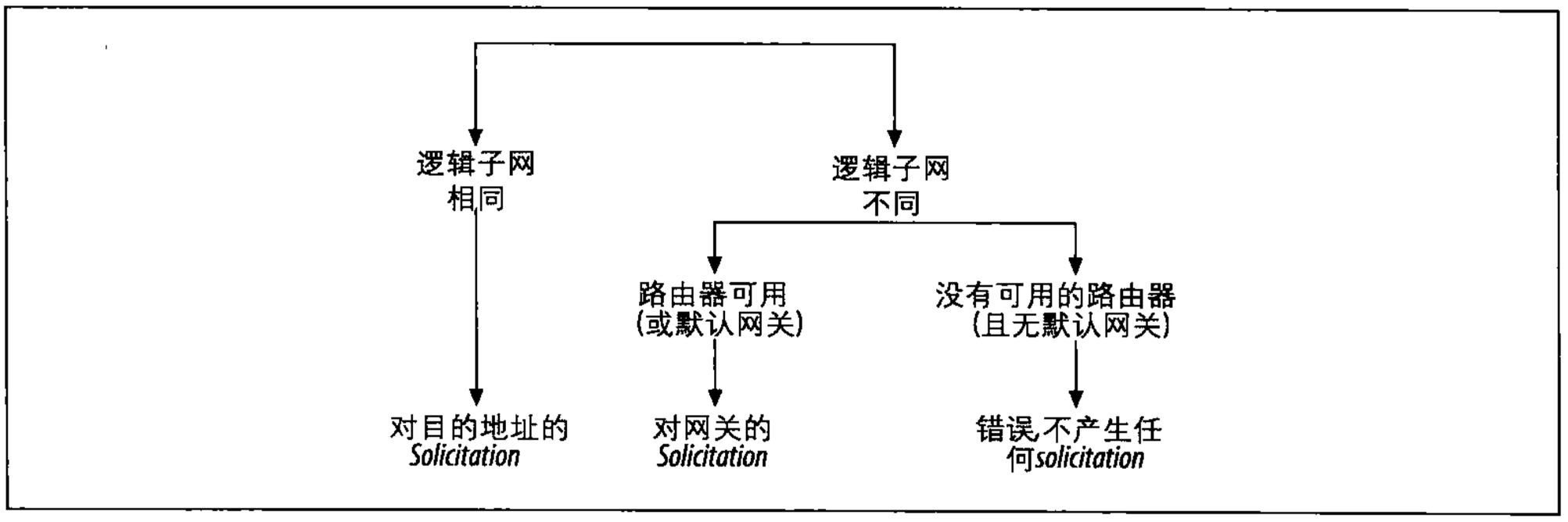


图 26-9：发送 solicitation 请求

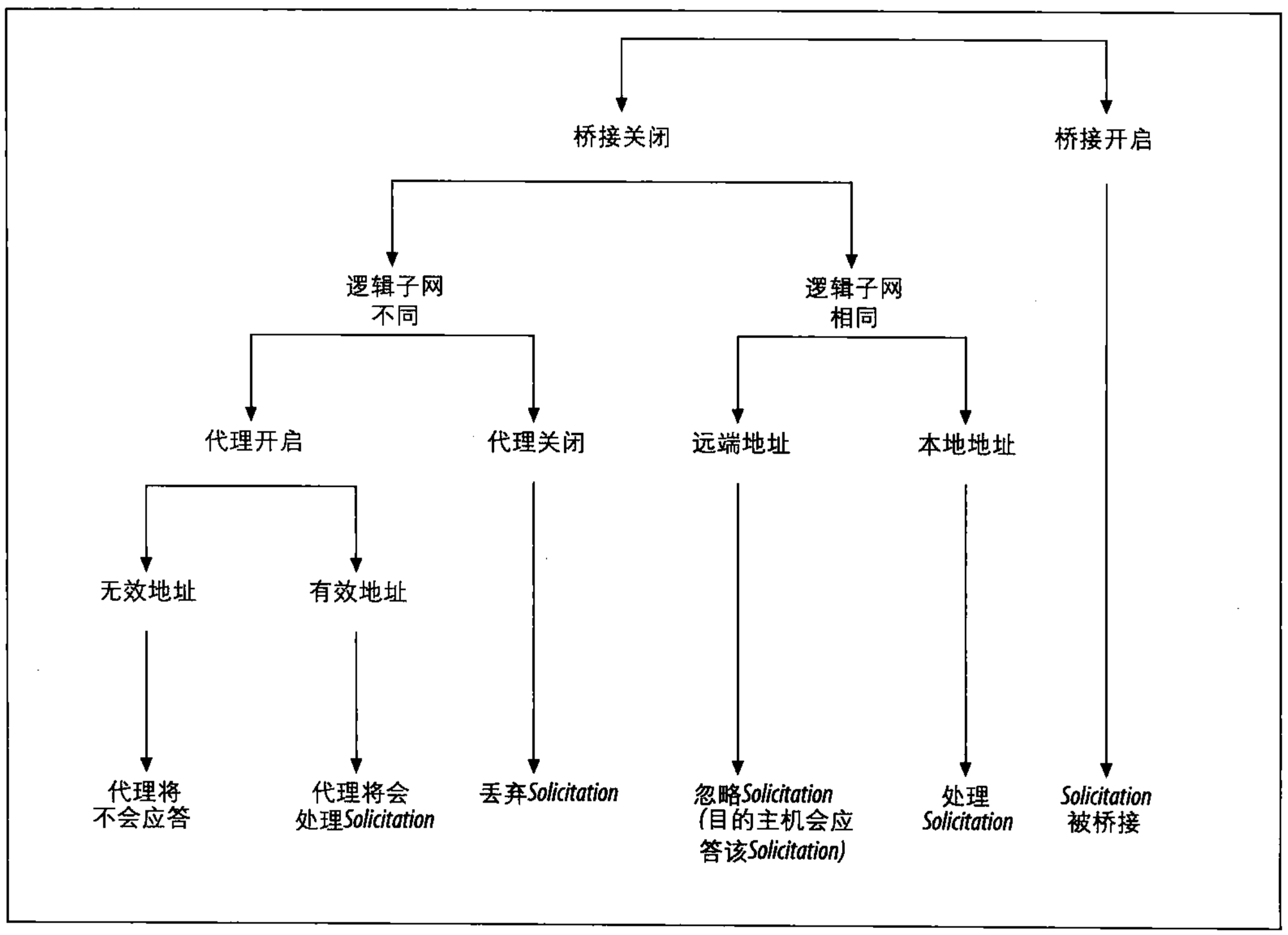


图 26-10：处理入口 solicitation 请求

到入口封包。换句话说，如图 26-10 所示，在 Linux 处理 solicitation 请求的过程中，桥接模块先于代理模块处理请求。第四部分有关于桥接的详细介绍。

现在假定关闭桥接功能。在一个共享介质中的一台主机能够接收到目的地址是其他主机

的 solicitation 请求。下面是能够影响 Linux 决定是否应答入口 solicitation 请求的一些因素：

逻辑子网（Logical subnet，也就是 IP 子网）

当请求解析的地址和配置在接收该solicitation 请求的NIC上的L3地址属于同一个逻辑子网（按照接收 solicitation 请求的主机配置）时，就出现 26-10 所示的“逻辑子网相同”。我们以 IPv4 为例，10.0.0.1（作为请求解析的地址）和 10.0.0.2/24（配置在接收请求的 NIC 上的地址）是属于同一个 10.0.0.0/24 IP 子网。

当两台主机属于同一个逻辑子网时，它们可以直接通信。否则，需要有路由器的帮助才能完成通信。

注意到，一个接口上可能配置了属于同一个逻辑子网的多个地址（其中一个是主地址，其他是辅助地址），或多个地址属于不同的子网，再或者两种情况皆而有之。如果接收请求的NIC上配置了多个不同子网的地址，那么请求解析的地址必须属于其中的一个子网。

物理子网（Physical subnet，LAN）

当两台主机属于同一个 LAN 时，它们理论上可以直接通信，但它们实际上能否可以还要根据 LAN 中的逻辑（L3）配置。例如，在图 26-1(c)中，位于同一个 LAN 中的主机却属于不同的 IP 子网。

一台主机不会试图去解析属于不同的逻辑子网的另一台主机的地址。相反，它会解析路由器的地址，因为路由器是该主机要访问远端主机所必需的，如图 26-9 所示。

一台主机将永远不会在其 NIC 上收到这样一个 solicitation 请求（如果我们排除特殊情况和发生错误）：请求的地址是另一块 NIC 上的 L3 地址，除非使用代理。由于图 26-10 是从接收者的角度给出的，所以图中在“逻辑子网不同”节点下面没有区分“无论子网相同”和“物理子网不同”。因为它们之间没有任何差异：只有代理的状态才是最重要的。

代理服务器的要求

代理服务器并不是处理所有收到的solicitation请求，参见“使用代理的条件”一节。

第二十八章的“处理入口 ARP 封包”一节给出了 ARP 协议是如何处理图 26-10 中的各种情况的。

邻居状态和网络不可到达性探测（NUD）

图 26-11 给出了传送一个封包到某个 L3 地址时，内核工作的简化过程。

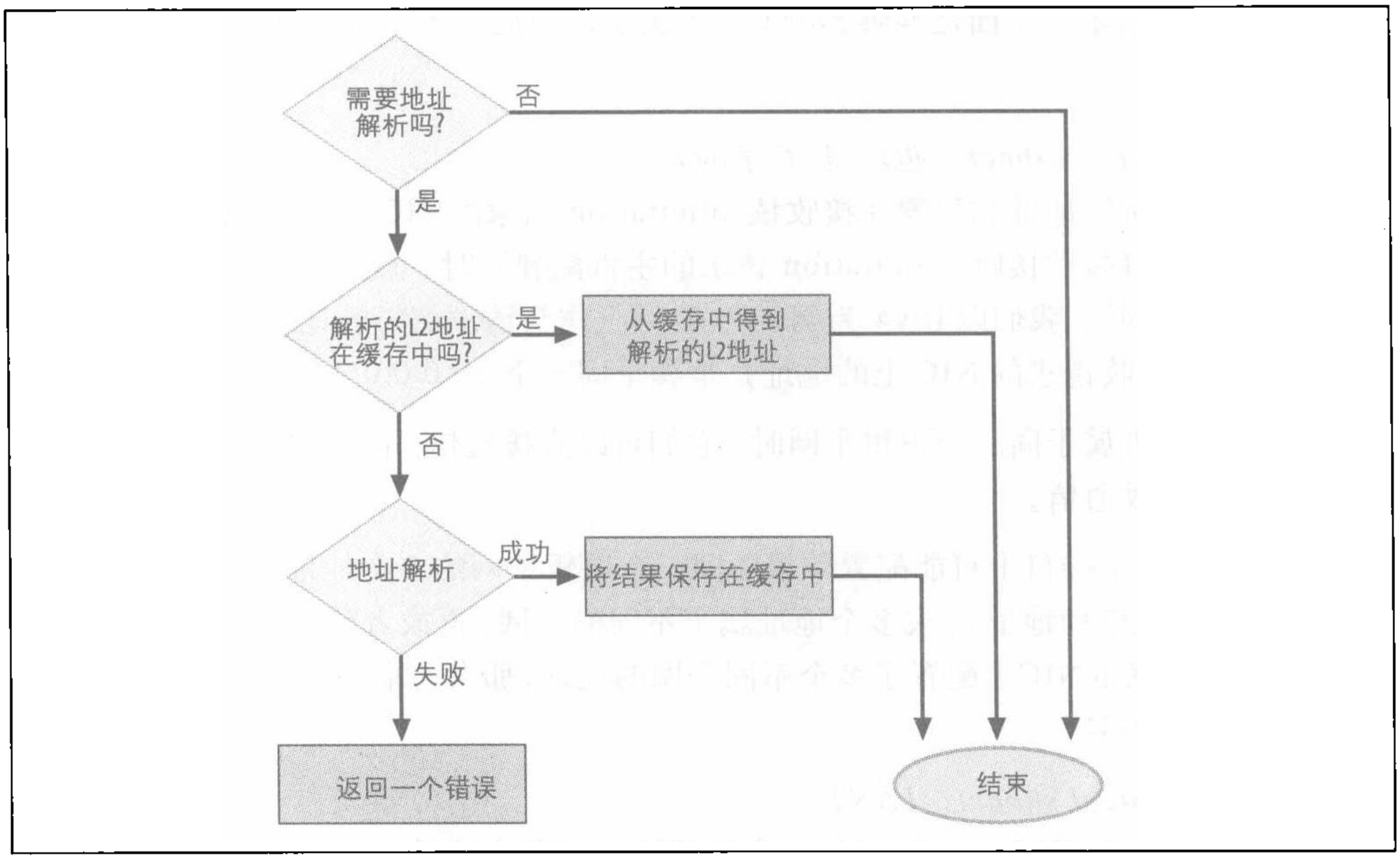


图 26-11：L3 到 L2 地址的解析步骤

图 26-12 说明邻居状态变化的简化模型。

图 26-11 和图 26-12 的这两个简化模型代表了大多数情况。但是，Linux 内核使用更为复杂的模型来处理所有可能的状态。下一节将扩展图 26-12 的模型，后续章节将详细讨论图 26-11 中的情形。

正如你将看到的，管理邻居最重要的工作是要知道它们是否可到达（reachable）。

可到达性

从邻居子系统的角度看，可到达性可以通过实际生活的一个实例来表述。假设你处于一间漆黑的屋子，屋内还有其他人，包括我。如果你说：“每个人都出去！”其余人就会都离开这个房间，因为他们都能听到你的声音。但如果你只想我出去，那么你需要知道一个信息：我的名字。

类似地，一个目的地是广播地址的solicitation应答和一个目的是单播地址的应答所携带的信息量是不同的：任何主机都可以接收广播，但如果你想与特定的接收者通信，必须知道它的确切地址。

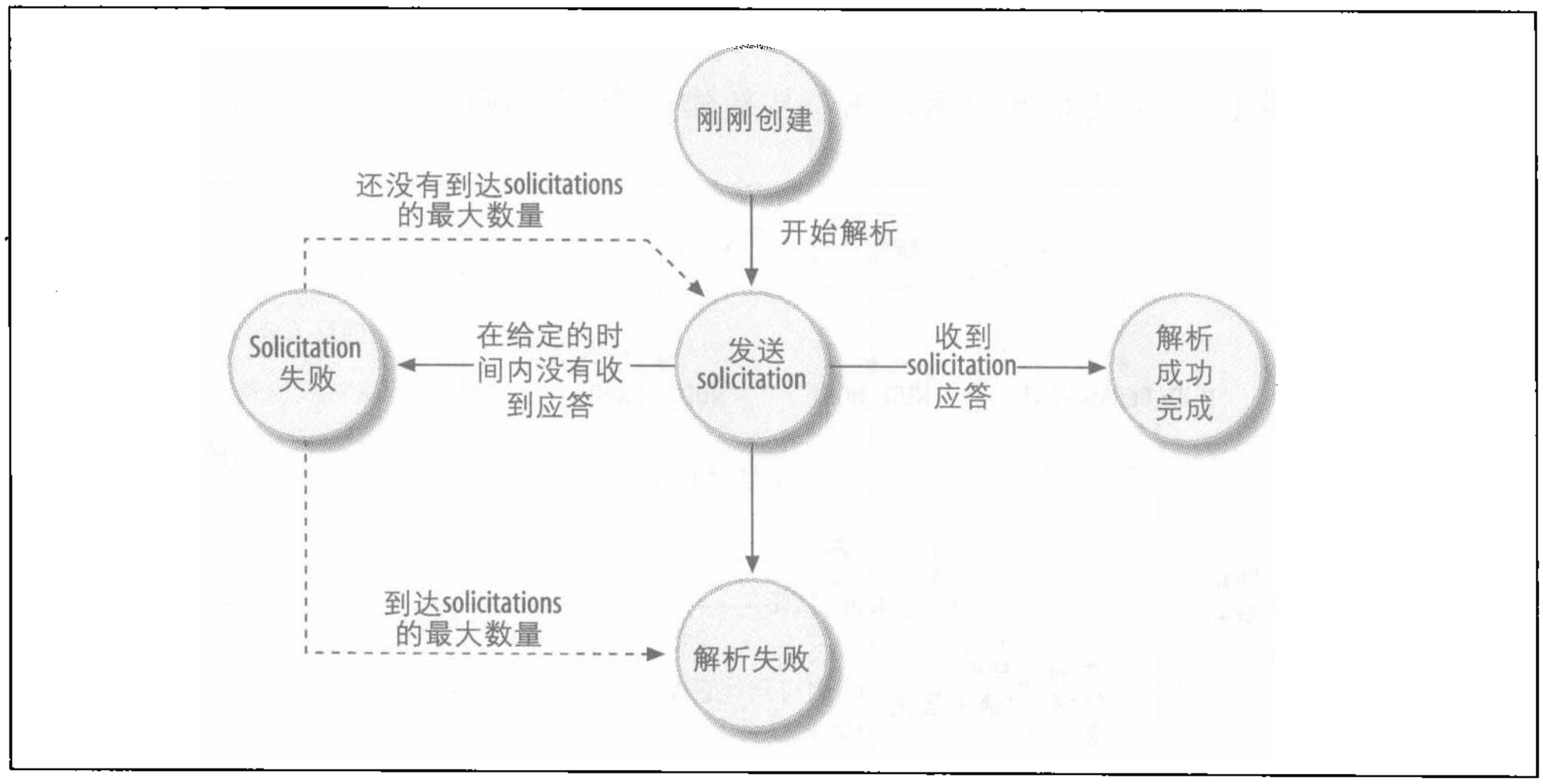


图 26-12：L3 到 L2 的映射状态图

从邻居的角度来说，如果内核有证据表明接收者可以正确地接收地址是其单播地址的帧，那么这台主机就认为是可到达主机，反之亦然。换句话说，内核认为一个邻居可到达时，表示的是双向可到达性。在本章剩余部分，我们将使用的术语可到达（reachable）就意味着双向可到达性。在“可到达性确认”一节中，我们会看到有两种方式来证实可到达性：L4 确认和 solicitation 应答。

NUD 状态之间的转换

IPv6 定义了一种 NUD 机制，它能快速判断邻居是否断线或死机。Linux 内核对 IPv4 和 IPv6 使用同样的机制。本书没有提到的其他协议，例如，DECnet 也使用类似的机制。

图 26-13 总结出了邻居可能处于的状态和触发状态转换的条件。每个邻居项可以由多个事件创建，包括传送数据封包到邻居的请求，或者收到来自邻居的 solicitation 请求。

一个邻居项的状态在其生存期内可能会改变多次，并且同一个状态可以多次被同一个邻居项使用。不同的协议为了利用特殊条件，也许执行不同的状态转换，包括 26-3 中一些没有给出状态转换。例如，IPv4 可以把一个新邻居项直接设为 NUD_STALE 态，但 IPv6 不行。

下面介绍图 26-13 中所示的状态。根据不同状态值一些共性将状态按组分类。下面按照图中的对状态转移进行介绍，尤其是对 NUD 机制。

基本状态

图 26-13 所示的状态定义如下所示。我们从新建一个邻居项时的默认状态开始：

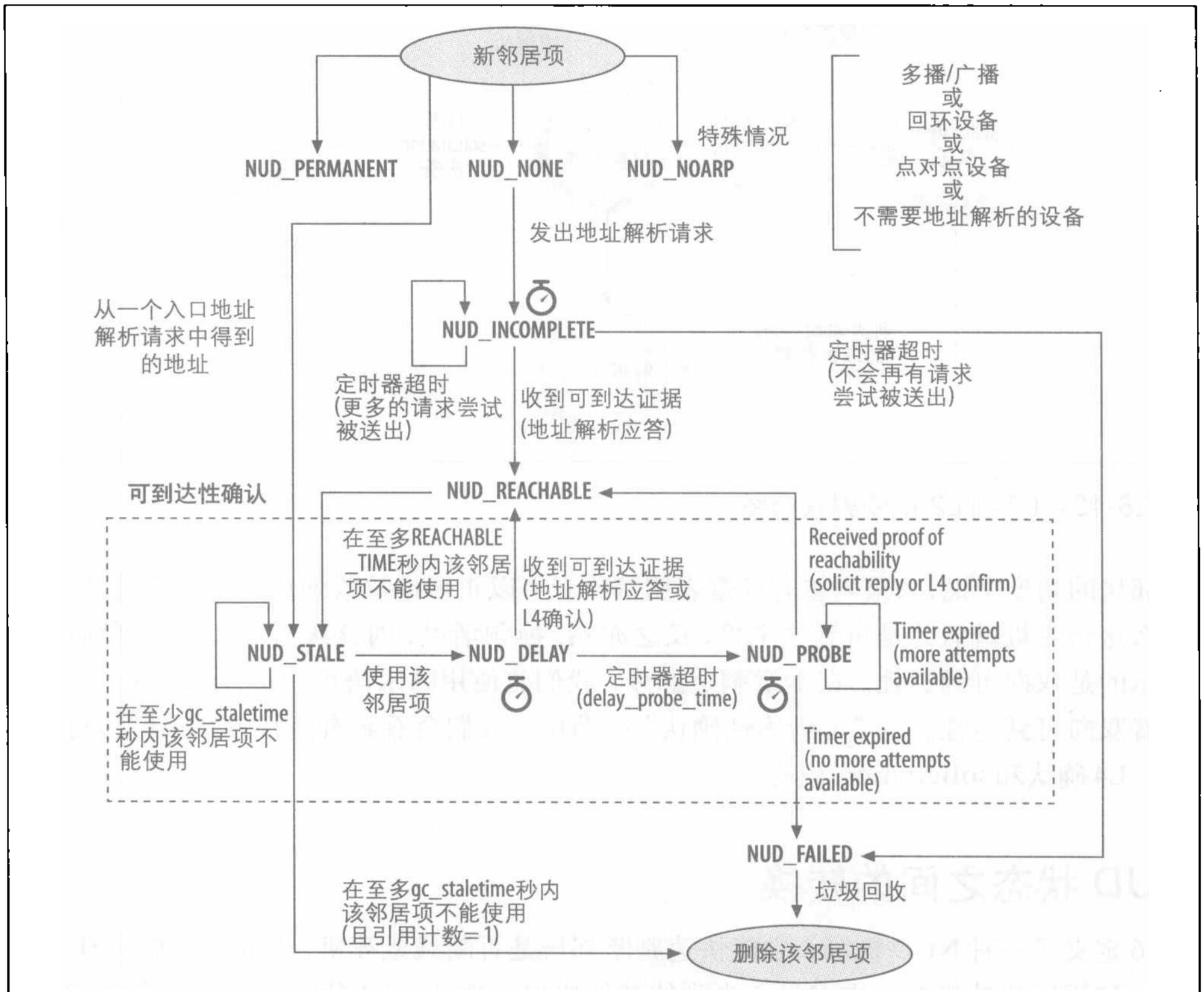


图 26-13：NUD 状态间的转换

NUD_NONE

邻居项刚被创建，还没有状态可用。

下面几组来自 IPv6 的邻居定义，在最新版 Linux 的 ARP/IPv4 实现中也被采用：

NUD_INCOMPLETE

一个 solicitation 请求已经被送出，但还没有收到应答。在这个状态，不使用任何硬件地址（甚至旧地址也不使用，那是 NUD_STALE 状态）。

NUD_REACHABLE

邻居的地址被放入缓存，并且知道该邻居是可到达的（已经有了可到达性证据）。

NUD_FAILED

由于 solicitation 请求失败，将邻居标记为不可到达。当创建一个新邻居项时可能进入该状态，或者 NUD_PROBE 状态也会触发到该状态。

NUD_STALE

NUD_DELAY

NUD_PROBE

状态转换阶段；当本地主机确定邻居是否可到达时，状态会改变。参见“可到达性确认”一节。

下面的一些值表示一组特定的状态，这些状态一旦被指定永远不会改变：

NUD_NOARP

该状态用于标记不要任何协议进行 L3 到 L2 的地址映射的邻居（参见“特殊情况”一节）。第二十八章的“arp_constructor 函数的启动”一节给出了在 IPv4/ARP 中使用该状态的原因和方法。从名字上看该状态好像只应用到 ARP 中，但它实际上可以被任何邻居协议使用。

NUD_PERMANENT

邻居的L2地址是静态配置（也就是说使用用户空间命令），因此不需要邻居协议进行地址解析。参见第二十九章的“邻居的系统管理”一节。

派生状态（Derived states）

除了上一节列出的基本状态外，还定义了下面的一些派生状态。在涉及多个有共同点的状态时，这些派生状态可以使代码更清晰。

NUD_VALID

如果某邻居项的状态不是下列状态（这些状态代表邻居相信有一个可用的地址）之一，那么它被认为是 NUD_VALID 态。

NUD_PERMANENT

NUD_NOARP

NUD_REACHABLENUD_PROBE

NUD_STALE

NUD_DELAY

NUD_CONNECTED

这个状态是NUD_VALID状态的子状态，没有未决的确认要处理：

NUD_PERMANENT

NUD_NOARP

NUD_REACHABLE

NUD_IN_TIMER

当某一邻居项的状态不是很清晰时，邻居子系统就为其运行一个定时器。与此相关的基本状态是：

NUD_INCOMPLETE

NUD_DELAY

NUD_PROBE

下面用一个例子来说明内核代码中派生状态的作用。当某个邻居实例被删除时，需要停止所有的与它的数据结构关联的未决的定时器。执行这个任务以前需要将邻居状态分别和与它关联的三个有未决的计时器的状态比较，定义NUD_IN_TIMER态后，这个过程就很简单，只要对邻居的状态与NUD_IN_TIMER的比较就可以，而这个比较通过位操作符 & 就能完成。

初态

当创建一个邻居实例时，其默认状态为NUD_NONE。若是通过明确的用户命令创建的，那么其初态就可以明确设置为其他任何态（参见第二十九章）。

在第二十七章的“邻居初始化”一节中会提到，邻居协议的constructor方法可以根据相关设备（例如，点对点连接）的特性和L3地址（例如，广播地址）改变邻居状态。

可到达性确认

我们在“为何没有足够的静态地址”一节提到过，L3地址到L2地址的映射是可以改变的。基于此原因，有必要对保存在缓存中的信息定期进行确认，看是否有信息在一定时间内没有使用。这个过程就叫做可到达性确认（reachability confirmation）。

注意，可到达性状态的改变不只是由于“需要邻居协议的原因”一节所列出的原因；路由器、网桥或者其他网络设备出了故障，也会导致这个问题。当正在进行可到达性确认时，缓存中的信息仍然可以使用，因为假定其仍然是有效的。

NUD_STALE、NUD_DELAY和NUD_PROBE三个NUD状态，都支持可到达性确认。使用这些状态主要的原因是，只有一个封包要发送到相关的邻居时，才启动可到达性确认。

我们再一次对这三个NUD状态进行确切的定义，然后看看确认映射正确的两种方式：

NUD_STALE

缓存中包含一个邻居的地址，但是该地址已经有一段时间（参见第二十九章的“neigh_parms 结构”一节的内容）没有进行确认了。当下一次有封包要到达这个邻居时，要启动可到达性认证进程。

NUD_DELAY

这个状态与NUD_STALE状态关系很密切，表示一种优化方式，可以减少solicitation请求发送的次数。

当发送一个封包到一个邻居，且该邻居相关的缓存项处于NUD_STALE时，就进入这个状态。NUD_DELAY状态表示一个时间窗，通过窗口可以通过外部主机来证实该邻居的可到达性。最简单的外部确认方式是，若被怀疑的邻居发出一个封包，则表明它仍在运行且能够访问。

为了进行可到达性确认，该状态给网络层一个时间期限，这个期间内内核不发出solicitation请求，这样节省了网络带宽、降低了CPU的利用率。看上去，它只是进行了很小的一点优化，但如果你放在一个大网络环境下考虑，你就能想象到它带来的好处。

如果没有得到确认，该缓存项就进入下一个状态：NUD_PROBE。这个状态通过明确的solicitation请求，或者协议使用的其他机制能够判定指定邻居的状况。

NUD_PROBE

在分配的某个时间段，当邻居已处于NUD_DELAY并且没有收到可到达性验证时，它的状态就转变为NUD_PROBE，同时开始solicitation处理。

邻居的可到达性状况可以通过两种方式进行确认。这两种方式各自的权限等价不同，它们是：

来自单播的solicitation应答认证

当你的主机收到一个solicitation应答时，该应答是作为它以前发出的solicitation请求的回应，这就意味着邻居收到了那个请求并且做了回答；也就是说邻居已经有了你的L2地址，或者从你的请求中得到了你的地址（见第二十七章的“创建一个邻居项”一节）。同时还表示存在一条双向的有效路径。但是，上述一切只有在solicitation应答作为单播发送出来时才是成立的。广播应答的接收将时缓存项的状态转移到NUD_STALE态而不是NUD_REACHABLE态（在第二十八章的“处理入口ARP封包”一节，你可以找到从ARP的角度对这个问题的详细解释）。

外部认证

如果你的主机确定自己收到了一个来自邻居的封包，该封包是对以前发送的某个封包的回答，那么就可以假定邻居仍然是可到达的。图26-14给出了这样一个例子，当图中主机A收到来自主机B的SYN/ACK封包时，该封包是对主机A的SYN封包的应答，就证实主机B是可以到达的。注意，如果主机B不是A的邻居，来自B的SYN/ACK应答将证明主机A到达主机B时使用的下一跳网关是可到达的。

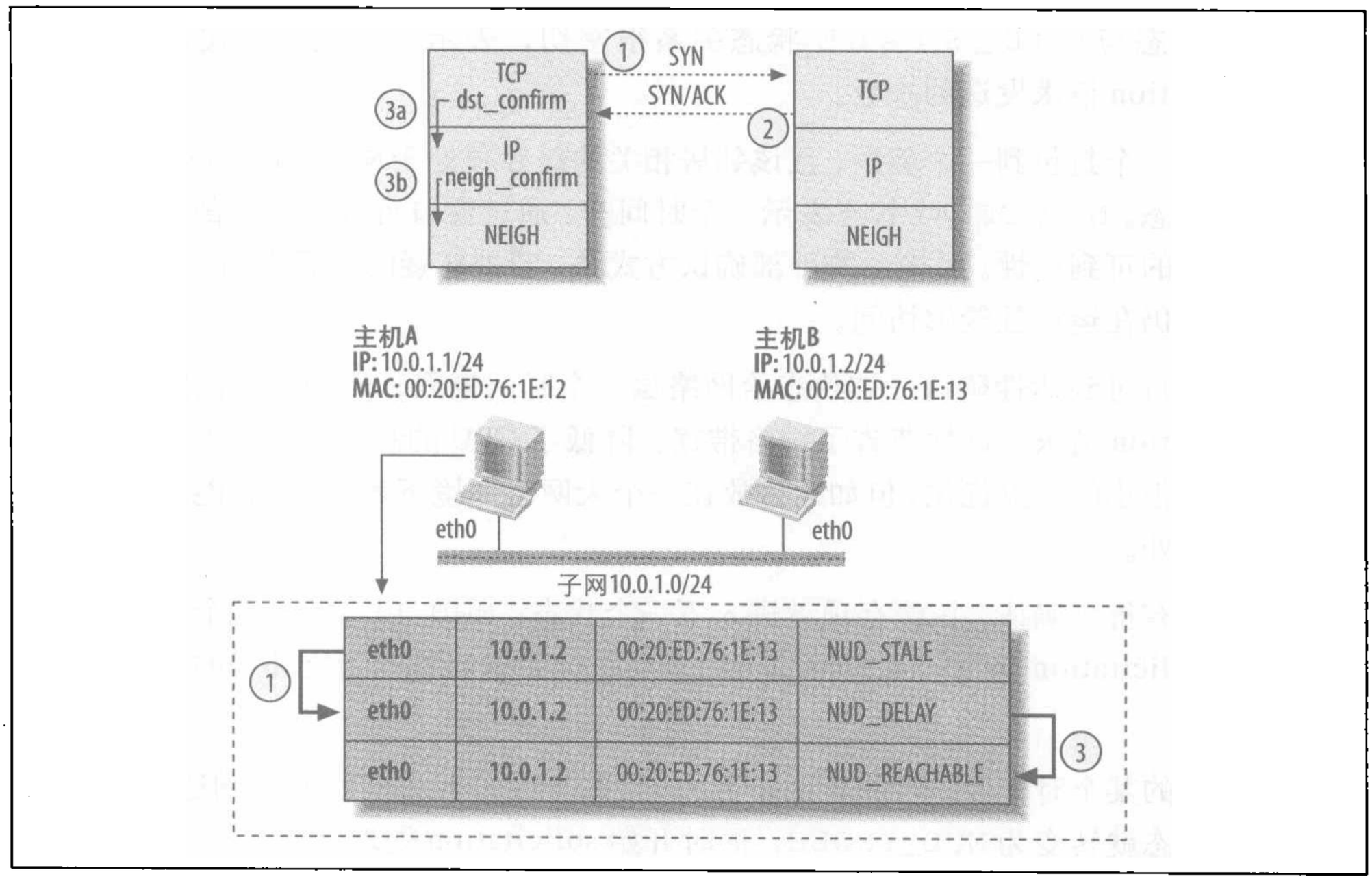


图26-14：外部邻居可到达性确认的例子

确认是通过dst_confirm函数完成的，该函数证实了用于向主机B路由SYN封包的路由表缓存项的有效性。dst_confirm是对neigh_confirm函数的简单封装，后者完成我们前面描述过的任务：确认邻居的可到达性并且完成L3到L2的映射。注意，neigh_confirm只更新neigh→confirmed时间戳；neigh_periodic_timer函数（当邻居进入NUD_DELAY态时，启动的定时器到期后执行该函数）负责将邻居项状态更新为NUD_REACHABLE态（注7）。

注7：在从L4收到确认应答和设置状态到NUD_REACHABLE态之际的这段延时内不影响任何网络流量。

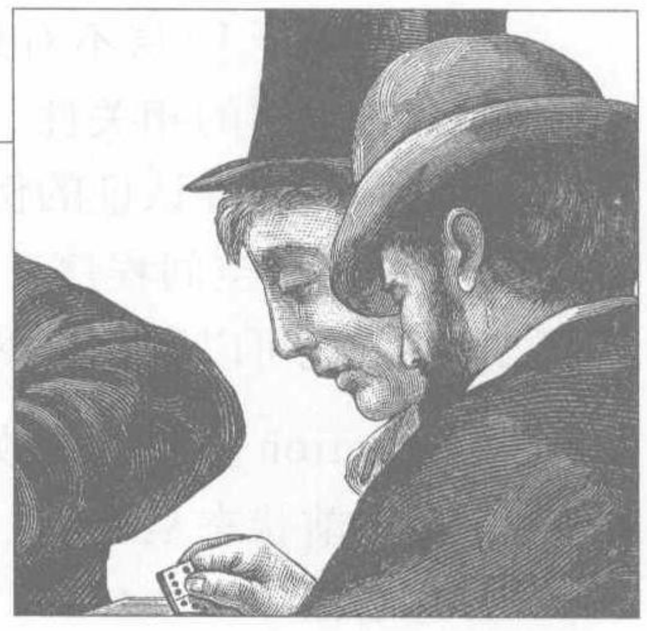
注意，由于IP层不对封包中数据流做任何处理，所以它无法检测在图26-14中的两个封包间的相关性。这也是由L4来处理封包的原因。TCP SYN/ACK封包是L4协议提供额外认证的例子。给定一个套接字，以及相关的路由缓存项和下一跳网关地址，用户空间程序在类似send和sendmsg的传输函数中使用MSG_CONFIRM选项，就可以确认该网关的可到达性。

solicitation应答的接收使状态转换到NUD_REACHABL态，而不管当前状态为何。只有当前状态是NUD_STALE态时，才能使用外部认证。如果缓存项刚刚创建，那么它处在NUD_INCOMPLETE态，此时不允许通过外部认证来确认邻居的可到达性（参见图26-13）。

注意，NUD_DELAY/NUD_PROBE和NUD_NONE态可以转换到NUD_REACHABLE态，如图26-13所示。但是，从NUN_NONE态到NUD_REACHABLE态，需要完整的可到达性验证。而从NUD_DELAY/NUD_PROBE态转换过去，满足任何验证都可以。

第二十七章

邻居子系统：基础结构



在第二十六章中，我们看到了邻居协议要解决的主要问题。你也知道了Linux内核将部分解决方法抽象为一个通用基础结构（infrastructure），并且这个通用基础结构由各种邻居协议共享使用。本章中，我们将看到这个基础结构是如何设计的。尤其是邻居协议如何与这个通用基础结构连接，代理和缓存是如何实现的，以及外部子系统（例如，高层协议）如何将其关心的事件通知邻居协议。我还会说明L3协议，例如，IPv4，如何与它们的邻居协议相连接，以及等待地址解析的缓冲区队列是如何实现的。

主要的数据结构

为了理解邻居基础结构的代码，我们首先对邻居子系统中使用较多的一些数据结构做一介绍，并且解释它们彼此之间是如何交互的。

这些数据结构的定义大部分在*include/net/neighbour.h*文件中。Linux内核代码中，邻居子系统的数据结构和函数使用英式拼写*neighbour*来表示邻居。本书中使用更常见的美式拼写neighbor，这种拼写方式在RFC和其他官方文档中也会看到。

`struct neighbour`

存储邻居的有关信息，例如，L2和L3地址、NUD状态、访问该邻居经过的设备等。注意，一个`neighbour`项不是与一台主机有关，而是与一个L3地址相关。因为一台主机可能有多个L3地址。例如，路由器就有多个接口，因此也有多个L3地址。

`struct neigh_table`

描述一种邻居协议的参数和所用函数。每个邻居协议都有该结构的一个实例。所有实例都插入到一个由静态变量`neigh_tables`指向的一个全局表中，并由`neigh_tbl_lock`来加锁保护。该锁只保护全局表的完整性，并不对表中每个条目的内容进行保护。

struct neigh_parms

对每个设备上邻居协议行为进行调整的一组参数。由于在大部分接口上可以启动多个协议（例如，IPv4 和IPv6），所以一个net_device结构可以关联多个neigh_parms结构。

struct neigh_ops

一组函数，用来表示L3协议（如IP）和dev_queue_xmit之间的接口。在第十一章介绍过API dev_queue_xmi，在下面的一节“L3协议和邻居协议间的通用接口”中还有对其详细介绍。这些虚拟函数可以根据它们使用的上下文环境来改变（也就是说，根据第二十六章介绍的邻居的状态）。

struct hh_cache

缓存链路层头部信息用于加快传输速度。一次将一个缓存的头部信息复制到发送缓冲区中比按位填充头部信息要快的多。并不是所有的网络设备驱动都支持缓存头部信息。参考“L2帧头缓存”一节。

struct rtable

struct dst_entry

当主机需要路由一个封包时，首先会查询自己的路由缓存中的目的主机信息，在缓存不命中时，接着才会查询路由表。每次查询路由表时，都会将结果保存在路由缓存中。IPv4路由缓存由结构rtable组成。每一个rtable实例对应于一个不同的IP地址。在rtable结构的字段中包括目的地址、下一跳地址和一个dst_entry类型的结构，用于存储与协议无关的信息。dst_entry包含一个指向neighbour结构的指针，该neighbour结构和下一跳地址有关。在第三十六章我会详细介绍dst_entry结构。在本章的剩余部分，我会经常提到dst_entry结构作为路由表缓存中的一个元素的dst_entry结构。实际上dst_entry只是rtable结构的一个字段。

图27-1给出了dst_entry结构如何链接到hh_cache结构和neighbour结构。

邻居代码也使用一些小的数据结构。例如，pneigh_entry结构用于基于目的地址的代理；neigh_statistics结构用于收集邻居协议的统计数据。第一个结构在“担任代理”一节会介绍，第二个结构在第二十九章的“统计数据”一节介绍。图27-2中也包括下面一些数据结构，这些在第二十二章和第二十三章中曾详细介绍过：

in_device、inet6_dev

分别用于保存某个设备上的IPv4和IPv6配置信息。

net_device

对内核识别的每个网络设备都有一个net_device 结构。参见第八章。

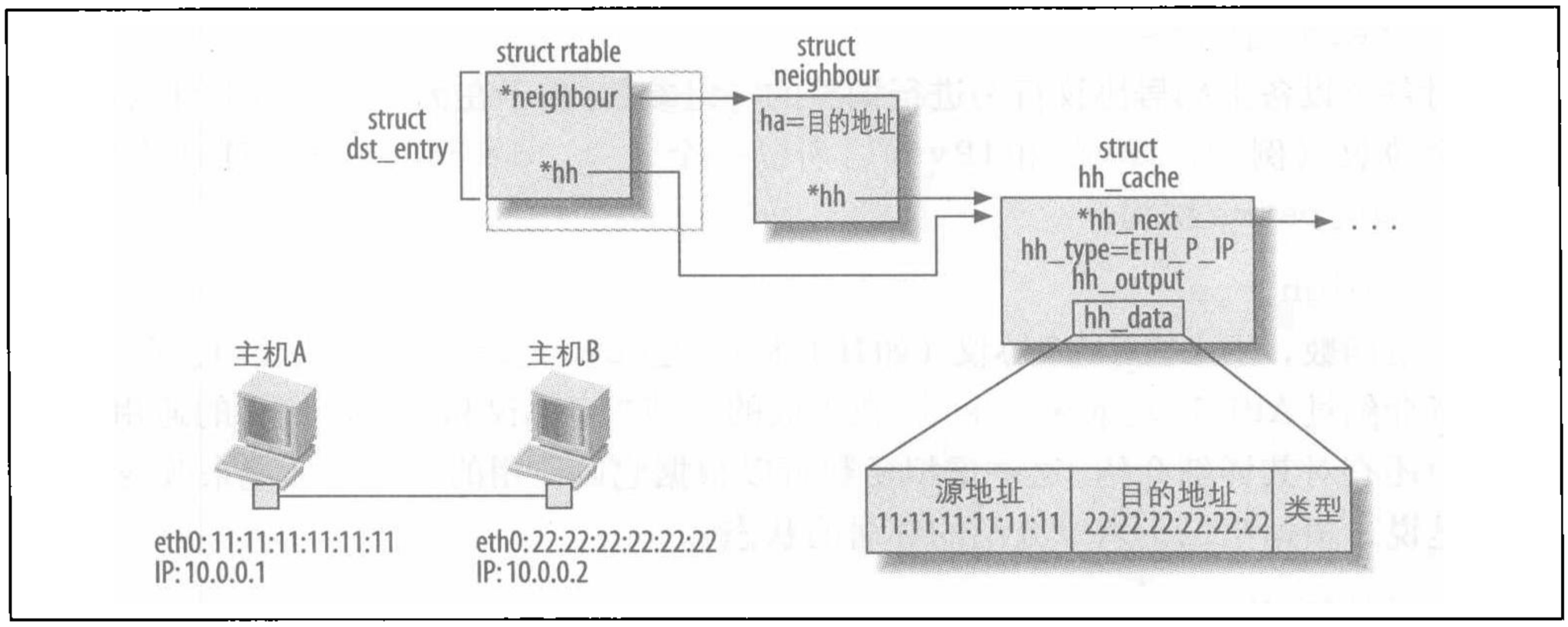


图 27-1：dst_entry、neighbour 和 hh_cache 结构之间的联系

图 27-2 给出了最重要的几个数据结构之间的关系。现在你是不是感到有点头大，但到本章结尾你就会很清楚了。

下面是对图 27-2 的几点说明：

- 在图的中间部分，你可以看到每个网络设备都有一个，指向一个数据结构的指针，该结构中保存着配置在这个设备上的一个 L3 协议的配置信息。如图中所示，IPv6 和 IPv4 都在同一个设备上。结构 `in_device`（IPv4 配置信息）和结构 `inet6_device`（IPv6 配置信息）中各包含一个指针，分别指向它们对应的邻居协议，即 ARP 和 ND。

  每个协议所用的全部 `neigh_parms` 结构都链接到一个单向列表中，其根结点保存在该协议的 `neigh_table` 结构中。

- 图中上部和底部显示每个协议中有两个 hash 表。第一个是 `hash_buckets` ，用于缓存协议解析过对 L3 到 L2 的映射或静态配置。第二个是 `phash_bucket`，保存着需要被代理的 IP 地址，见“按设备代理和按目的地代理”一节。应该注意的是，`phash_bucket` 不是缓存，因此它的元素不会过期，也不需要定期确认。每个 `pneigh_entry` 结构包含一个指向对应的 `net_device` 结构的指针（图 27-2 中没有显示）。图 27-6 给出了缓存结构 `hash_buckets` 的详细细节。
- 如果网络设备支持缓存链路层头部信息，那么每个 `neighbour` 实例就与一个或多个 `hh_cache` 结构相关联。“L2 帧头缓存”一节和图 27-1、图 27-10 中有对 `neighbour` 结构和 `hh_cache` 结构之间关系的详细介绍。

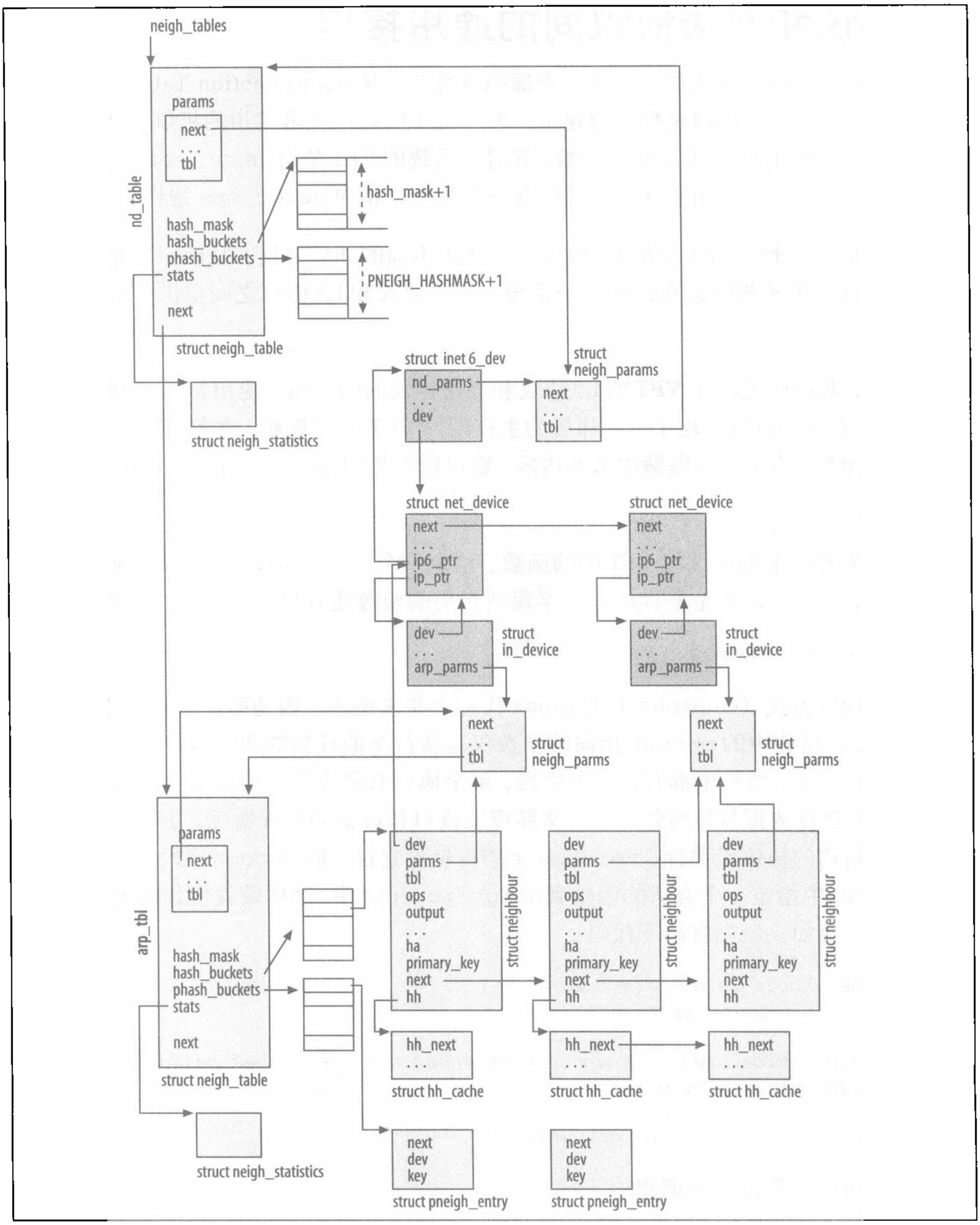


图 27-2：数据结构之间的关系

L3 协议和邻居协议间的通用接口

Linux 内核有个通用邻居层，通过一个虚拟函数表（Virtual Function Table，VFT）将 L3 协议和主要的L2 传输函数连接起来。VFT 是 Linux 内核最常用的机制，可以使各个子系统在不同的时间使用不同的函数。邻居子系统的VFT是由neigh_ops结构实现的。在每个 neighbour 结构的 ops 字段中有一个指针，指向 neigh_ops 结构。

VFT 接口的灵活性使得不同的 L3 协议可以使用不同的邻居协议。从而，即允许不同的邻居协议行为很不相同又允许邻居子系统在邻居协议和 L3 协议之间提供一个公共的接口。

在本节中，我们研究基于 VFT 的 L3 协议和邻居协议间的接口、使用 VFT 的优点、什么时候 VFT 第一次初始化及在一个邻居的生存期内 VFT 如何更新。并简要介绍控制 VFT 初始化的函数。为了更好理解本节的内容，建议你首先阅读第二十九章的“neigh_ops结构”一节。

让我们首先看一下如何调用 VFT 中的函数。给定一个 neighbour 实例和它内嵌的 VFT neighbour->ops，理论上 output 字段所指向的函数能直接以如下方式调用：

```
neigh->ops->output
```

但是这种构造方式（construct ）在 Linux 代码中并未使用，因为即使这样也不够通用。neigh_ops 结构中的 output 指向的函数仅是执行类似任务的四个函数之一，每个函数在neigh_ ops结构中都有自己的字段。每个协议必须决定它要使用四个函数中的哪个。函数的选择依据具体事件、上下文环境及接口和设备的配置情况。因此，为了保持邻居基础结构的协议无关性，neighbour结构包含它自己的output字段。每个协议从neigh->ops中指定一个合适的函数到neigh->output中。这样做就使代码更加简单而且更清晰。例如，原先有如下代码：

```
if (neighbour is not reachable)
    neigh->ops->output(skb)
else
if (the device used to reach the neighbor can use cached headers)
  neigh->ops->hh_output(skb)
else
  neigh->ops->connected_output(skb)
```

而邻居基础结构只需如下调用：

```
neigh->output
```

neigh->output被协议初始化为正确的neigh_ops方法。当然，每个邻居协议使用它自己的逻辑初始化 neigh->output；不必遵照上述规则。

当创建一个邻居时，其neighbour->ops字段被初始化为适当的neigh_ops结构，如图27-3(a)所示。这种设定关系在邻居生存期内是不会改变的。但是，正如图27-3(b)中描述的那样，neigh->output在邻居结构的生存期内可以多次改变为不同的函数。这种改变可以由在协议运行期间发生的事件触发，也可能（情况比较少）是被用户命令改变。下面的章节将详细介绍图27-3所示的两个初始化过程。

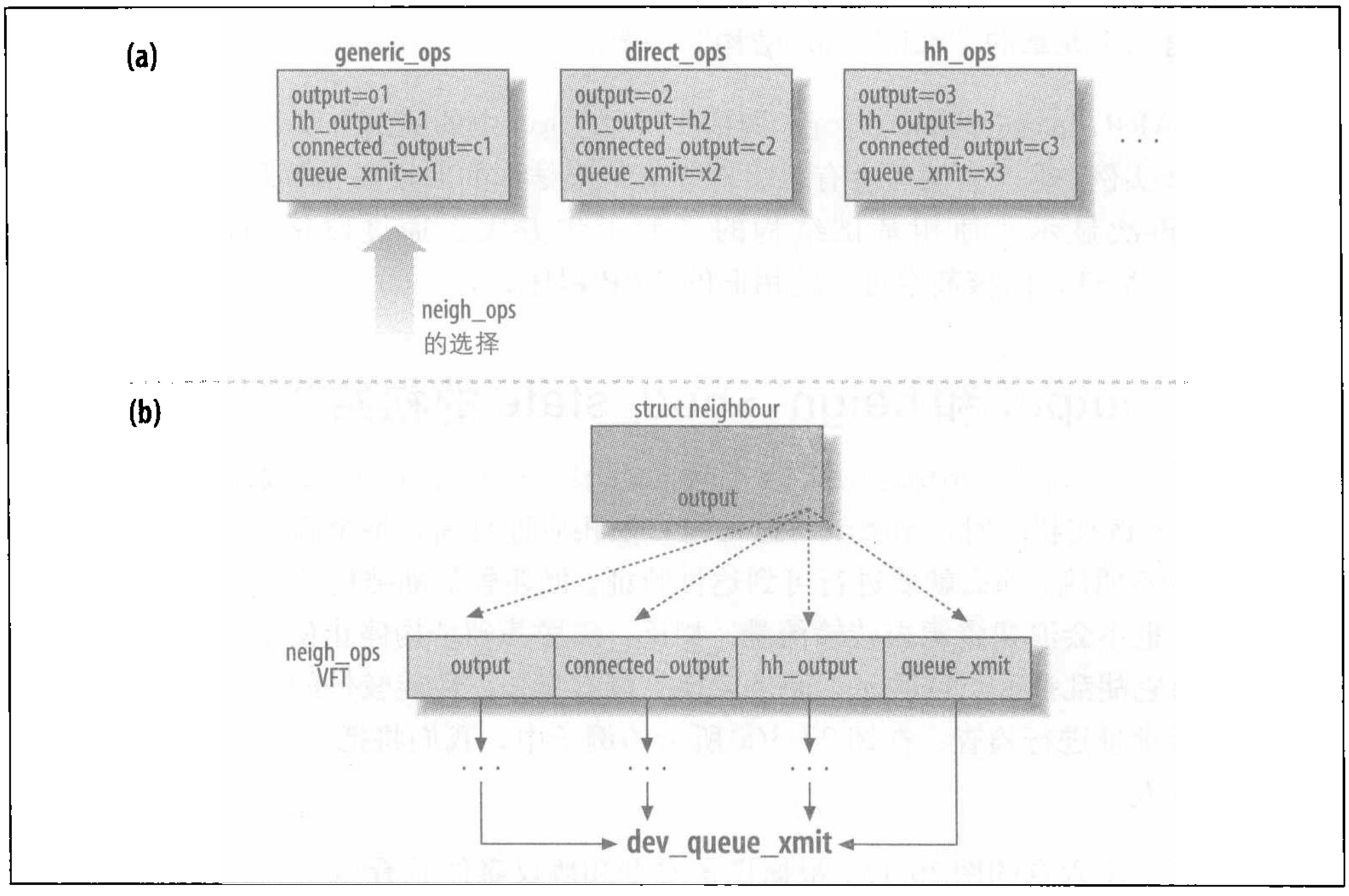


图27-3：(a) neigh->ops的初始化；(b)neigh->output 的初始化

neigh->ops的初始化

对于特殊类型的设备，图27-3（b）中列出的函数初始化过程可以进一步优化，以加快传输速度。这些优化包括，例如，对于第二十六章的“特殊情况”一节中描述的情况，是不需要进行L3地址到L2地址的映射。在那些情况下，几乎可以绕过邻居子系统，仅仅需要第十一章描述的queue_xmit函数。协议代码需要知道这种细节，但是通用邻居基础结构无法提供，因此协议只能初始化neigh->output为neigh->ops->queue_xmit，并且一切对上层协议而言仍是透明的。这种方式真是简单明了！

基于这个原因，每个协议都提供三个不同的neigh_ops VFT实例：

- 一个可以在任何上下文中使用的通用表（*xxx*_generic_ops）。这个函数经常用于处理那些需要对L2地址进行解析的邻居。

- 当设备驱动程序提供它自己的一组函数来处理L2帧头时，就会使用一组经过优化的函数。这些函数能够加快缓存的帧头（xxx_hh_ops）的使用。
- 当设备不需要对L3地址到L2地址的映射时，就要用到一个表(xxx_direct_ops)。ISDN用原始IP封装就是一个例子。

当创建一个邻居实例时，协议根据几个条件将neigh_ops VFT初始化到正确的实例函数。参见第二十九章的“neigh_ops结构”一节。

若是IPv4/ARP，arp_broken_ops调用neigh_ops中的第四个实例通常用于初始化neighbour实例，这些实例与没有采用新邻居基础结构的旧设备相关联，否则它们无法工作。这再次显示了通用基础结构的一般工作方式：通过以适当的方式初始化neigh_ops VFT，内核甚至可以使用旧的ARP程序。

neigh->output 和 neigh->nud_state 的初始化

邻居的状态（neigh->nud_state）和neigh->output函数彼此互相依赖。当nud_state改变状态时，output通常也必须相应地更新。举个简单例子，如果邻居的状态变得不明确，那么就要进行可到达性验证。但邻居基础结构并不是立刻进行可到达性验证，也不会浪费资源去传输流量。相反，邻居基础结构停止使用优化的output函数，因为它胡乱插入当前地址，而是切换到没有优化、性能较慢的output函数，这个函数会对地址进行检查。在图27-3(a)所示的例子中，我们将把connected_output从*c1*改为*o1*。

查看一下第二十六章的图26-13，根据设备类型和协议事件看看neigh->nud_state可能呈现什么状态，将有助于理解本节的内容。

邻居子系统提供一个通用（generic）函数——neigh_update，它能将邻居的状态改变为其输入参数中提供的状态。本章后面的章节会详细描述neigh_update函数，在这里首先看一下最常用的状态变化和为了管理这些状态需要调用的辅助函数，这些函数可以直接调用，也可以通过neigh_update调用。

让我们从最常见的情形开始：一台需要邻居协议的设备、一个不属于第二十六章介绍的特殊情况的地址和由传输引起的状态的改变（也就是说，这里排除了创建状态和删除状态）（注1）。第二十六章的图26-12能被简化为图27-4。图中也给出了内核中处理传输

注1：首次对neigh->output初始化时，要检查comstructor函数的源代码（例如，对于ARP/ND协议是arp_constructor/ndisc_constructor）。对于ARP，参见第二十八章的“初始化neighbour结构”一节。

封包的函数。但是，图中并没有显示全部neigh_update调用引起的传输，因为大部分的传输都是普通的，只会给图中增加无关的值。只有收到solicitation应答引起的传输才在图中显示。

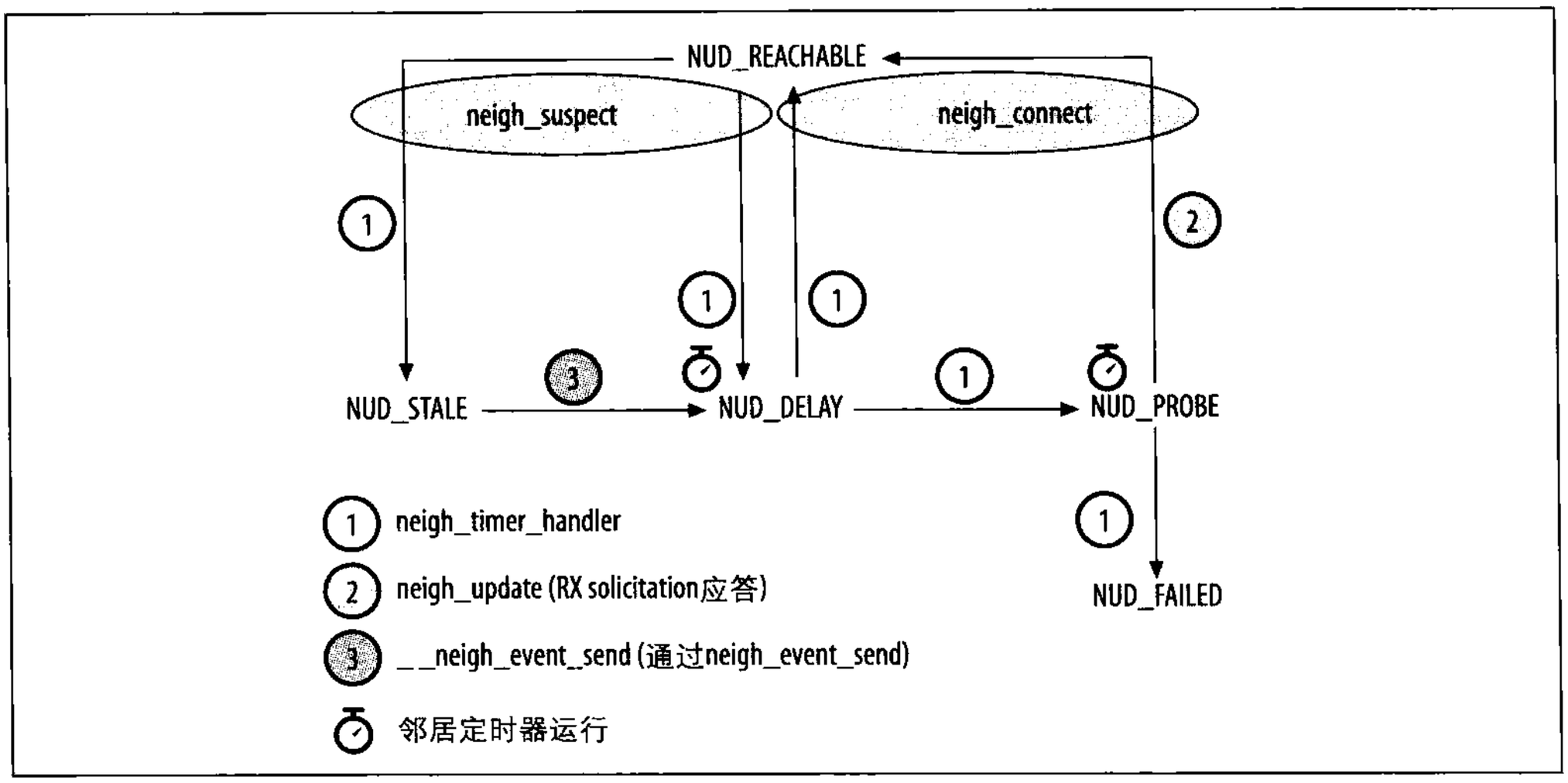


图 27-4：已经至少被解析过一次的邻居的可能的状态改变

注意，图27-4所示的部分状态的转移是异步的：这些转变由一个定时器管理，通过比较时间戳进行触发（注2）。其他的状态转移由协议负责同步［例如，neigh_event_send（注3）］。

普通状态改变函数：neigh_connect 和 neigh_suspect

邻居进入NUD_REACHABLE态（见第二十六章）的主要的方式是：

收到一个solicitation应答

当收到一个solicitation应答时，可能是由于第一次解析某个地址，也可能由于确认邻居是否处于NUD_PROBE状态，协议就会通过neigh_update更新neigh→nud_state。这个更新立即发生并且是同步的。

注2：比较时间戳的函数，比如time_after_eq和time_before_eq，定义在*include/linux/jiffies.h*文件中。

注3：图27-13中也有对neigh_event_send函数的部分介绍，作为扩展后的neigh_resolve_output函数流程图的一部分。

L4 认证

当收到一个 L4 可到达性确认后，就会第一次执行 neigh_timer_handler 函数，邻居状态就改变为NUD_REACHABLE态（参见第二十六章的“可到达性确认”一节）。L4 的确认是异步的并且可能会有些延迟。

人工配置

当用户使用系统管理命令创建一个新的 neighbour 结构时，在该命令中可以指定邻居状态、且 NUD_REACHABLE 态是合法态。这种情况下，通过 neigh_update 来调用 neigh_connect。

无论什么时候进入 NUD_REACHABLE 态，邻居基础结构就调用 neigh_connect 函数，将 neigh->output 函数指向 neigh_ops->connected_output。

当某个邻居由NUD_REACHABLE态转移到NUD_STALE态或NUD_DELAY态，或者仅仅初始化到与 NUD_CONNECTED 中的任一状态不同的状态（例如，通过调用 neigh_update 可以实现）时，内核就请求 neigh_suspect 执行可到达性确认（参见第二十六章“可到达性确认”一节）。这个工作由 neigh_suspect 完成，它将 neighbour->output 设置到 neigh_ops->output。

neigh_connect 和 neigh_suspect 也会更新链接到输入的 neighbour 实例的 hh_cache 结构中的所有 our->output 函数和 neighbour->hh_output 函数。但是没有函数会更新 neighbour 实例的 NUD 状态，因为上述函数的调用者会负责更新。本章的后面，我将会用两个词组，“connect the neighbor”和“suspect the neighbor”来分别表示对 neigh_connect 的调用和对 neigh_suspect 的调用。

某些状态转移（NUD 状态的改变）可能会在任何时候发生，并且在 neighbour 实例的生存期内转换多次。而另一些状态转移可能只发生一次。根据一些组网知识，不难看懂第二十六章的图 26-13，并且能够区分两类不同属性的转换。对于被初始化到固定状态（例如，NUD_NOARP 状态）的 neighbour 实例，neigh->output 可以被立刻初始化为 neigh_ops->connected 并且永远不会改变。

neigh->output 使用的函数

在前面的几节中提到过，neigh->output 由邻居的 constructor 函数初始化，之后由函数 neigh_connect 和 neigh_suspect 根据协议事件的结果对其操作。neigh->output 总是被设置成 neigh_ops 的一个虚拟函数。本节列出的函数可以被指定为 neigh_ops 的虚拟函数。定义在 *net/core/dev.c* 的函数 dev_queue_xmit 并不是真正作为邻居子系统的一部分。其他的函数在 *net/core/neighbour.c* 中进行定义。

dev_queue_xmit

当要传送一个封包时，L3总是调用这个函数，而不管接口设备的种类或使用什么样的L2和L3协议。当出口设备上传输需要的所有信息都准备好，并且邻居子系统没有其他工作时，邻居协议会将neigh_ops的函数指针初始化为dev_queue_xmit。如果你看看第二十八章的arp_direct_ops，就能发现全部四个传输虚函数都被设置为dev_queue_xmit。在第十一章有对函数dev_queue_xmit的详细介绍。

neigh_connected_output

该函数只是填充L2帧头，然后调用neigh_ops->queue_xmit。因此，它希望L2地址被解析。neighbour结构在NUD_CONNECTED状态会用到这个函数。

neigh_resolve_output

该函数在数据传输前将L3地址解析为L2地址。因此，当L3地址和L2地址的对应关系还没有建立或者需要对其确认时，就会用到该函数。如果创建一个neighbour新结构并且需要对其L3地址进行解析时，除了在第二十六章“特殊情况”一节讲到的情形外，neigh_resolve_output是作为默认函数使用的。

neigh_compat_output

该函数是为了保证向下兼容。在引入邻居基础结构以前，由它负责调用dev_queue_xmit函数，即使L2地址还没准备好。

neigh_blackhole

该函数用于处理neighbour结构不能删除的临时情况，因为有人仍然要调用这个neighbour结构。函数neigh_blackhole会丢弃在输入接口上接收的任何封包。为了确保任何试图给邻居传送封包的行为不会发生，这样处理是必需的。因为邻居的数据结构将要被删除。参见“删除邻居”一节。

第二十八章的“初始化neighbour结构”一节给出了ARP如何使用上述函数来初始化neigh_ops VFT的不同实例。图27-13所示的流程图中也给出了如何选择相应的函数。

邻居信息的更新：neigh_update

neigh_update定义在文件*net/core/neighbour.c*中，是用于更新neighbour结构链路层地址的通用函数。下面简单介绍一下带有输入参数的它的原型：

```
int neigh_update(struct neighbour *neigh, const u8 *lladdr, u8 new,
                 u32 flags)
```

neigh

指向要更新的neighbour结构。

lladdr

新的链路层（L2）地址。lladdr并不总是初始化为一个新值。例如，当调用neigh_update来删除一个neighbour结构时（设置其状态为NUD_FAILED，参见“删除邻居”一节），会给lladdr传递一个NULL值。

new

新的NUD状态。

flags

用于传达信息，例如，是否要覆盖一个已有的链路层地址等。下面是来自*include/net/neighbour.h*文件中一些有效的flags：

NEIGH_UPDATE_F_ADMIN

管理性改变。意思是说改变来自用户空间命令（参见第二十九章的“邻居的系统管理”一节）。

NEIGH_UPDATE_F_OVERRIDE

指当前的L2地址可以被lladdr覆盖。管理性改变使用这个标识来区分*replace*和*add*命令（参见第二十九章的图29-1）。协议代码可以使用这个标识来给一个L2地址设定一个最小生存期（参见第二十八章的“普通处理的结尾”一节）。

下面三个标识只能在IPv6中使用：

NEIGH_UPDATE_ISROUTER

表示这个邻居是个路由器。这个标识用于初始化neighbour→flags中的IPv6标识NTF_ROUTER。

NEIGH_UPDATE_F_OVERRIDE_ISROUTER

表示IPv6 NTF_ROUTER标识可以被覆盖。

NEIGH_UPDATE_F_WEAK_OVERRIDE

如果输入参数中提供的链路层地址lladdr与当前已知的邻居neigh→ha的链路层地址不同，那么这个地址就是可疑的（也就是说，邻居的状态会转移到NUD_STALE，以便触发可到达性认证）。

IPv6的ND协议在协议头中使用的标识可以影响刚刚列出的NEIGH_UPDATE_F_*XXX*标识的设置。下面的讨论中略去了neigh_update中处理IPv6的标识的部分内容。

所有用于改变neighbour结构的链路层地址的管理接口都使用neigh_update函数，如第二十九章的图29-1所示。邻居协议自身也使用该函数，但是它不是改变邻居状态的唯一一个函数。

图 27-5(a)和图 27-5(b)给出了 neigh_update 函数内部的深层描述。流程图被分为不同区域，每个区域负责不同的任务：

- 合理性检查。
- 当前状态不是 NUD_VALID 态的邻居发生的变化。
- 选择 L2 地址，供状态不是 NUD_VALID 态的邻居使用。
- 设置一个新的链路层地址。
- 改变 NUD 状态。
- 处理 arp_queue 队列。

下面的几个小节详细解释相关的代码。

neigh_update 优化

在改变一个邻居的状态前，neigh_update 函数会首先查看是否可以避免状态的改变。如果满足下列的条件（参见图 27-5(a)中 C 区域），则放弃改变状态：

- 链路层地址还没有被修改（也就是说，输入的 lladdr 和当前的 neigh->ha 是相同的）。
- 新状态是 NUD_STALE 而当前状态是 NUD_CONNECTED，这就表示当前状态比新状态更合适。

neigh_update 初始化

在本节中，我们跟踪 neigh_update 函数的判断过程，看它如何处理当前状态（neighbour->nud_state）和请求的状态（new 参数的值）的各种值。

只有管理命令（NEIGH_UPDATE_F_ADMIN）可以改变当前状态是 NUD_NOARP 态或 NUD_PERMANENT 态的邻居的状态。以 neigh_update 函数开头的会做合理性检查，如果约束条件不满足，程序就会退出。

当新状态 new 不是一个合法状态时，如果它是 NUD_NONE 态或 NUD_INCOMPLETE 态，就要停止启动了的邻居定时器；如果旧状态是 NUD_CONNECTED 态，则使用 neigh_suspect 函数将该邻居项标记为可疑的（也就是说，要求进行可到达性认证）。参考“neigh->output 和 neigh->nud_state 的初始化”一节。当新状态是一个合法态时，若其要求定时器（NUD_IN_TIMER），则邻居定时器就会重新启动。

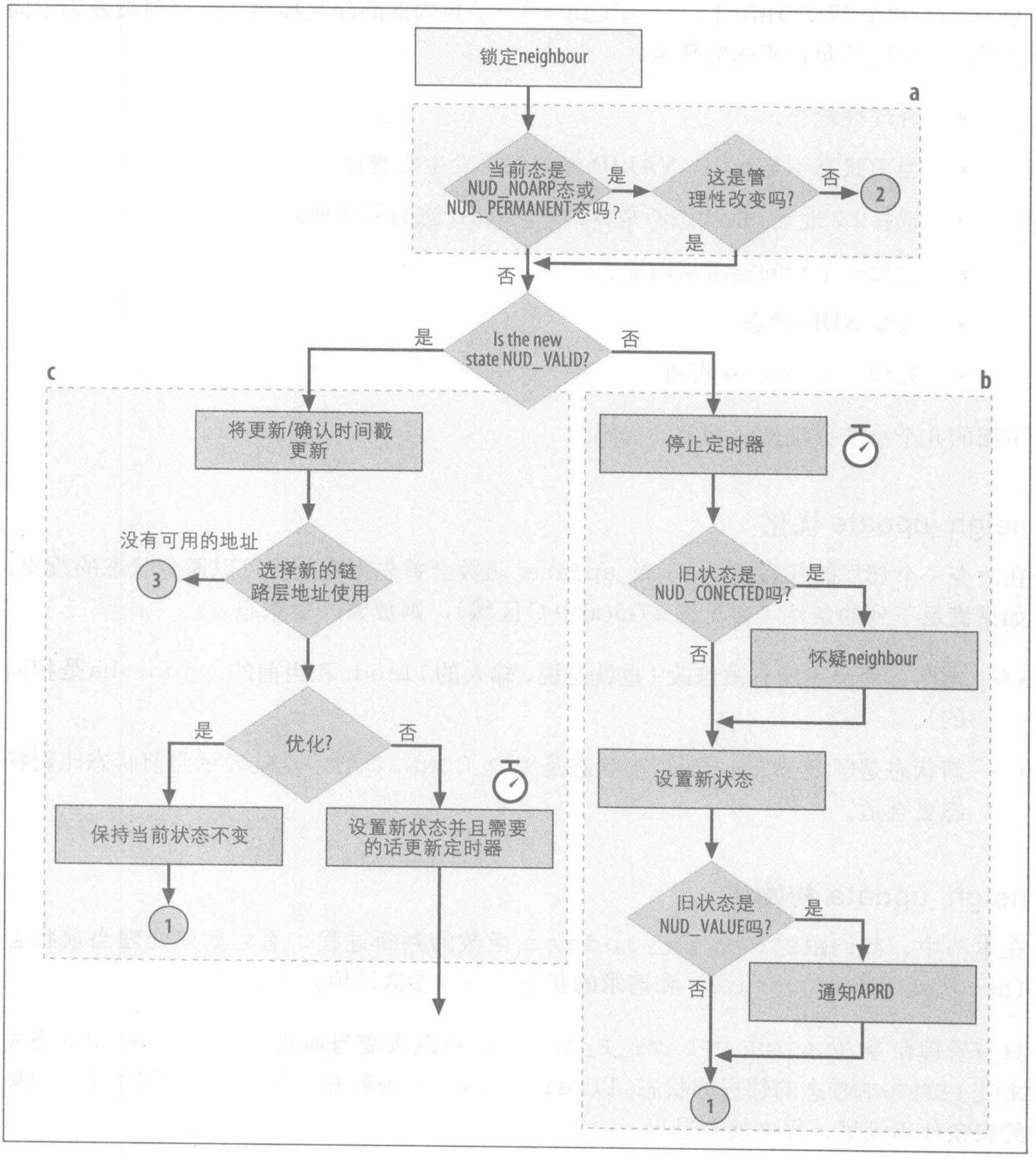


图 27-5 (a)：neigh_update 函数

当要求用 neigh_update 函数来将 NUD 状态转移到一个与当前态不同的值时，这种情况是最常见的，neigh_update 函数就需要检查该值是否是包含在 NUD_VALID 中的值，还是不在NUD_VALID中的其他值（NUD_VALID是派生态，其中包含了多个NUD_*XXX*值）。尤其当旧状态不是NUD_VALID态并且新状态是NUD_VALID态时，主机必须将在

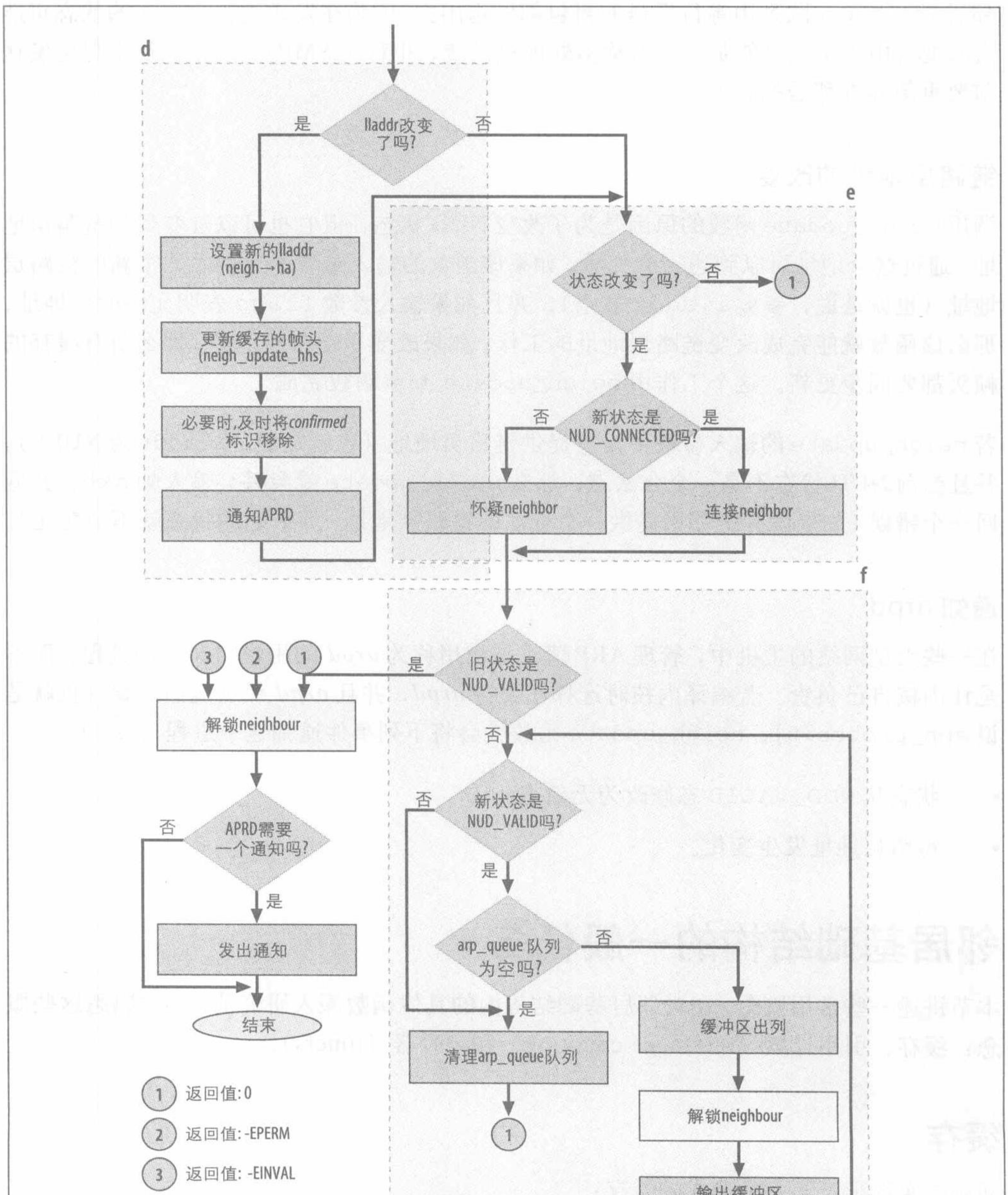


图 27-5（b）：neigh_update 函数

邻居arp_queue队列中等待发送的封包都发送出去。因为在发送过程中邻居的状态可能会改变（由于主机可能是一个对称多处理机系统，也称为SMP），所有在每个封包发送前要重新检查邻居的状态。

链路层地址的改变

调用neigh_update函数的原因是为了改变NUD状态，但它也可以改变目的链路层地址，通过这个地址可以到达一个邻居。如果该函数的输入参数中提供了一个新的链路层地址（也就是说，参数lladdr非空），并且如果输入参数flags表明允许改变地址，那么该函数就能完成改变链路层地址的工作。如果改变了链路层地址，那么所有缓存的帧头都要同步更新。这个工作由neigh_update_hhs函数完成。

若neigh_update的输入参数中没有提供链路层地址（也就是说，lladdr为NULL），并且当前NUD状态不是一个合法态，那么neigh_update就会丢弃输入帧skb，并返回一个错误（如果没有给邻居提供一个合法的链路层地址，那么它的状态就不会变化）。

通知arpd

在一些大型网络的主机中，管理ARP请求是使用称为*arpd*的用户空间守护进程，而不是让内核自己负责。若编译内核时选择了支持*arpd*，并且*arpd*经过适当配置（也就是说app_probes>0），neigh_update函数就会将下列事件通知这个进程（注4）：

- 状态从NUD_VALID态修改为无效态。
- 链路层地址发生变化。

邻居基础结构的一般任务

本节讲述一些常用概念。在对邻居基础结构内的具体函数深入研究前，必须熟悉这些概念：缓存、引用计数（reference counting）和定时器（timers）。

缓存

邻居层实现了下面两种形式的缓存：

邻居映射

由于任何数据都有可能被多次使用，那么缓存L3到L2的映射结果就是有意义的。

注4： 参见第二十八章的“ARPD”一节和第二十九章的“neigh_parms结构”一节。

不缓存错误结果（失败的地址解析）。但是会将映射失败的neighbour结构设置为NUD_FAILED态，以便垃圾回收定时器能将其清除（参见“垃圾回收”一节）。

L2 帧头

邻居基础结构会缓存 L2 帧头，这样可以缩短 L3 封包到 L2 帧的封装时间。否则，基础结构将要挨个初始化封包的 L2 帧头的每个字段。

因为缓存邻居映射对邻居子系统的操作是很重要的，本节将对此进行详细介绍。（后面的一节“L2 帧头缓存”介绍 L2 帧头缓存）。neighbour 结构的内容在第二十九章的“neighbour 结构”一节描述，在该章中的后面部分还会介绍 neighbour 结构的创建和删除。下面我们从更高的层次，描述一下这些结构如何被组织及如何被邻居基础结构访问的。

邻居基础结构将neighbour结构放到缓存中，每个协议一个缓存，其结构是典型的hash表结构，同一个bucket中冲突的元素被链接到一个单链表中。新元素被增加到链表的开头（参见“函数 neigh_create 的参数”一节的 neigh_create 函数）。Hash 函数负责把元素分发到 bucket 中，它的输入参数是 L3 地址、关联的设备和一个为了降低假设的拒绝服务攻击（DoS）的影响而定期重新计算的随机数。图 27-6 给出了缓存的结构。在图 27-2 中，你可以看到它和其他主要数据结构的关系，例如，和每个协议的neigh_table结构的关系。

Hash 表是由neigh_hash_alloc函数和neigh_hash_free函数来进行分配和释放。在每个协议初始化时（参见 neigh_table_init 函数），都会建立一个 hash 表，其初始容量为两个元素大小。如果表中的元素数目大于 bucket 的数目，hash 表就会按照下面的方式重新组织。首先，表的容量扩大一倍（hash 表的容量总是 2 的次方）。

然后，重新计算 hashing 所用的随机值。最后，表中的元素按照前面提到的变量重新分配：L3 地址、设备和随机数。Hash 表的扩充由 neigh_hash_grow 函数实现，必要时，neigh_create 函数会调用该函数。

注意，很容易就会引起 hash 表的扩充。因此，每个 bucket 中的数据结构很少多于两个。

Hash 表中所含元素的最大数目由 gc_threshX 变量控制，见“垃圾回收”一节。为了防止可能的 Dos 攻击，需要这样一个限制。

当“邻居系统”需要在hash表中查找一个邻居时，查找关键字是目的L3地址（primary_key）和能访问该邻居的设备（dev）。因为不同的协议使用的关键字长度不同，通用的查找 API 函数需要考虑关键字的长度。因此，关键字长度保存在 neigh_table 结构中。

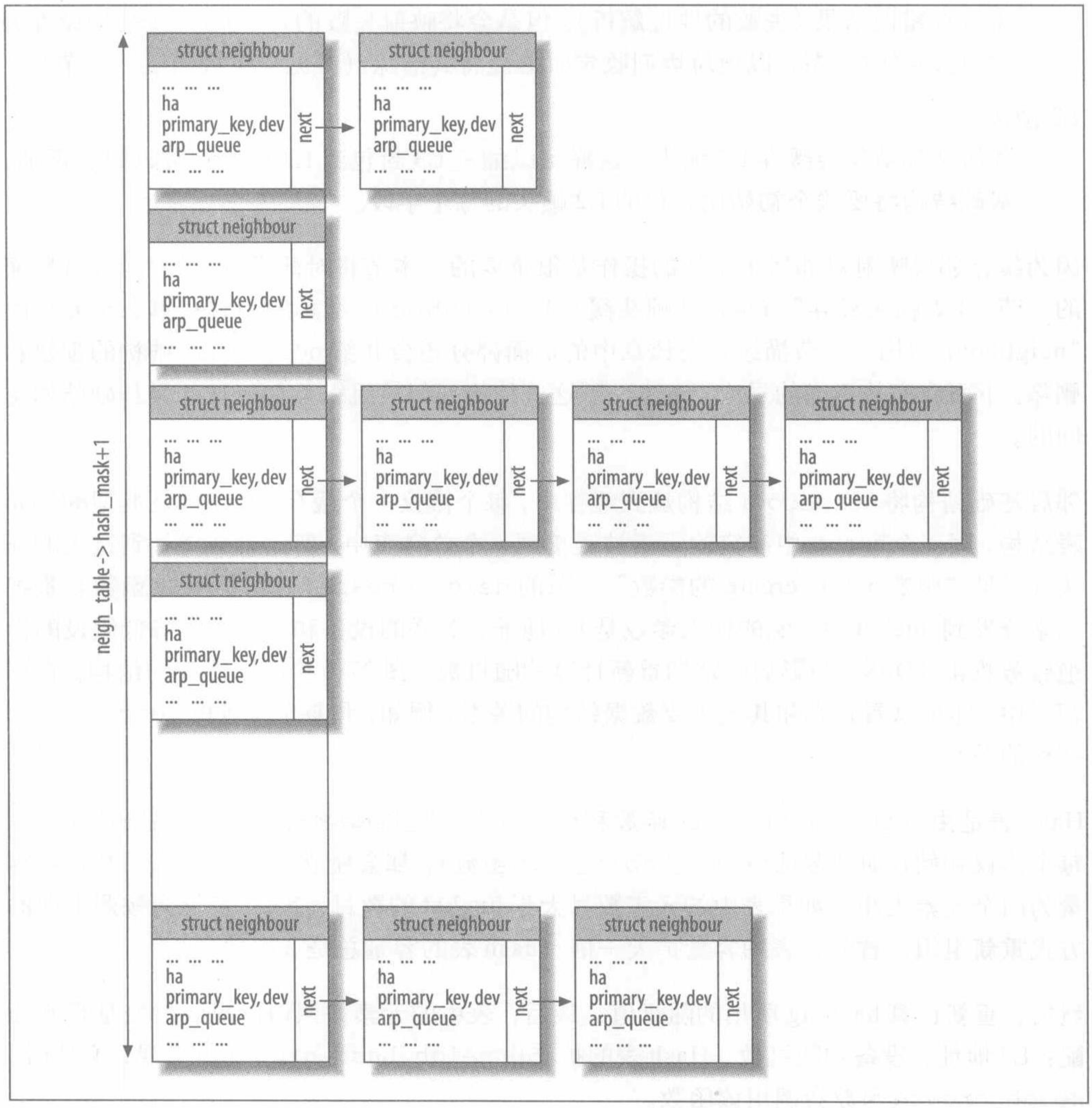


图 27-6：邻居的缓存

查询某个邻居协议的缓存的主要函数是neigh_lookup。除此之外，还有其他两个函数，都是用 neigh_lookup 包裹的，当查找失败时，它们既可以在缓存中强制创建一个 neighbour 项；也可以决定是否按照输入的参数来创建neighbour项。下面是这三个函数的简要介绍：

neigh_lookup

该函数会检查要查找的元素是否存在，并且在查找成功时返回指向该元素的指针。

```
struct neighbour *neigh_lookup(struct neigh_table *tbl, const void *pkey,
                               struct net_device *dev)
{
    struct neighbour *n;
    int key_len = tbl->key_len;
    u32 hash_val = tbl->hash(pkey, dev) & tbl->hash_mask;

    read_lock_bh(&tbl->lock);
    for (n = tbl->hash_buckets[hash_val]; n; n = n->next) {
        if (dev == n->dev &&
            !memcmp(n->primary_key, pkey, key_len)) {
            neigh_hold(n);
            NEIGH_CACHE_STAT_INC(tbl, hits);
            break;
        }
    }
    read_unlock_bh(&tbl->lock);
    return n;
}
```

_ _neigh_lookup

该函数用neigh_lookup包裹。当查找失败和该函数的输入参数中设置了creat标识时，该函数就使用 neigh_create 函数来建立一个 neighbour 项。

_ _neigh_lookup_errno

该函数使用neigh_lookup函数来查看要查找的邻居项是否存在，并且当查找失败时，总是创建一个新的neighbour实例。除了不需要输入 creat 标识外，该函数基本上和 _ _neigh_lookup 函数相同。

第二十八章介绍了另一个函数：arp_find。它用 _ _neigh_lookup包裹，并且向下兼容，主要是被旧代码来使用。除此之外，还有一个函数：neigh_lookup_nodev，它只在 DECnet 协议中使用。

每个协议也有一个独立的缓存和一组相关的 API 查找函数，这些函数用于目的地代理。你可以在“担任代理”一节找到相关的详细介绍。

定时器

邻居子系统使用了几个定时器。有些定时器是全局的，而另外一些定时器为每个邻居单独创建。某些定时器的运行是周期性的，也有一些只是在需要的时候才启动。下面是这些定时器的简单描述，在后面的章节我会对它们进行更详细的介绍；

状态转移时的定时器（neighbour->timer）

一些NUD状态的转移是由时间的推移而触发的，而不是由系统事件触发。这些状态转移包括：

从 NUD_REACHABLE *态到* NUD_DELAY *态或* NUD_STALE *态*

当在一段特定时间内没有送往或接收自某一邻居的流量，那么就发生这种状态转移；并且邻居子系统自动认为该邻居是不可到达的。

从 NUD_DELAY *态到* NUD_PROBE *态或* NUD_REACHABLE *态*

在邻居的可到达性被怀疑后，这就是该邻居的下一个状态；此时，如果可到达性没有被外部事件所证实，那么邻居子系统必须进行一次明确的探测。该定时器只是简单地检测状态改变所要求的条件，并且处理状态的转变。例如，在第二十六章的图 26-14 中，我们能看到当 TCP 提供可到达性认证时，如何调用neigh_confirm函数。neigh_confirm函数会更新neighbour结构中的一个时间戳，但并不改变该邻居的状态。但是当定时器检测到一个新的时间戳时，它就会改变相关的邻居状态。

每个 neighbour 结构中的定时器可以控制上述三个状态转移中的两个。当用 neigh_alloc 函数创建一个 neighbour 项时，该定时器的回调函数被初始化为 neigh_timer_handler 函数。在图 27-4 和第二十六章的“可到达性确认”一节中，你可以找到更多的相关信息。

失败的 solicitation 请求的定时器

如果在一个给定的时间段内没有收到solicitation请求的应答，那么就再发送一个新的 solicitation 请求。发送 solicitation 请求的最大次数在 neigh_parms 结构的 *XXX*_probes 字段中设定，相关内容参见第二十九章的“neigh_parms 结构”一节

当最后一次发送尝试失败后，相关的邻居项就被转移到 NUD_FAILED 状态（如图 27-13 所示），之后就会启动垃圾回收定时器来清除该邻居项。

垃圾回收定时器（neigh_table->gc_timer）

这是一个周期性定时器，用于确保内存不会浪费在没用的数据结构上。其回调处理函数是 neigh_periodic_timer。“垃圾回收”一节将详细描述垃圾回收机制。

neigh_periodic_timer 函数每隔 300 秒会将 neighbour 结构中的 reachable_time 更新为一个随机数（注 5）。之所以是个随机数，而不是一个固定值，是因为这样可以避免太多的邻居项同时到期：在特大型网络中，这会导致网络流量和CPU利用率的突然增加。

Proxy 定时器（neigh_table->proxy_timer）

对一个会接收到大量solicitation请求的代理来说，推迟请求的处理是有好处的。这个定时器负责延迟处理的时间。参见“延迟处理 Solicitation 请求”一节。

注 5：随机值由函数neigh_rand_reach_time计算，为了使其有较高的精确度，取值范围为 base_reachable_time/2 到(3xbase_reachable_time)/2。

邻居结构的引用计数

许多与创建新邻居有关的内核子系统都会在一些数据结构中保存对 neighbour 结构引用；例如，路由子系统就这样做。因此，neighbour结构中包含一个名为refcnt的引用计数器。分别使用neigh_hold函数和neigh_release函数对该计数器进行递增（加1）和递减（减1）操作。

引起邻居引用计数器递增的事件中，最普通的就是封包的传输。当发出一个封包时，由于相应的 sk_buff 缓存区中就保留着对一个 neighbour 结构的引用，所以计数器 neighbour->refcnt 就要加1，以保证封包的传送完成且没出问题。一旦封包传输完成，该计数器就减1。

这是一个短期引用的例子，其他的引用可以持续很长时间。例如，路由表缓存保持的引用［包括 IPv4 和 IPv6（注 6）］，如图 27-10 所示。

每个邻居的定时器的每次启动都会导致该邻居的引用计数器加1，如下面neigh_update函数的代码片断所示：

```
if (new & NUD_IN_TIMER) {
        neigh_hold(neigh);
        neigh->timer.expires = jiffies +
                               ((new & NUD_REACHABLE) ?
                               neigh->parms->reachable_time : 0);
        add_timer(&neigh->timer);
}
```

当一个邻居项由于某些原因需要被删除时（参见“与其他子系统的交互”一节的neigh_ifdown函数），并不是立刻释放其所占的内存，因为也许某个结构仍然引用它，而是将neighbour→dead字段设为1，表示该邻居已停用（dead）。之后不久，垃圾回收定时器就会负责清理，见“垃圾回收”一节中的解释。

创建一个邻居项

和大多数缓存项一样，neighbour 缓存项的创建是事件驱动的：当系统需要一个邻居，并且缓存不命中时，就需要创建一个邻居实例。特别是当下列的一个事件发生时，就创建一个新邻居项：

注 6：IPv4 的 rt_intern_hash（见第三十三章）和 IPv6 的 ip6_route_add 都是调用 _neigh_lookup_errno 来结束。

传输请求

当有一个向一台L2地址未知的主机的传输请求的，就需要对该地址进行解析。这是最常见的情况，图27-13(a)中也有描述。当目的主机不是与发送方直接相连时，解析的L2地址就是下一跳网关的地址，而不是目的主机的地址。

收到solicitation请求

由于发送请求的主机在请求封包中有自己的识别信息，收到该请求的主机会假定即将有两个系统间的通信产生，于是自动创建一个缓存项。(涉及ARP的详细内容可以参见第二十八章的图28-2)。然而，以这种方式（被动方式）学习到的信息，和用明确的solicitation请求和应答获得的信息并不具有相同的权威性（详细内容清参见第二十六章的“NUD状态之间的转换”一节)。

手工添加

管理员可以通过*ip neigh add*命令创建一个邻居缓存项，详情参见第二十九章的“邻居的系统管理”一节。

当发生上述事件之一，以及查询邻居子系统的缓存不命中时，邻居协议就试图解析地址(通常情况是发出一个solicitation请求)，并且在每个邻居协议缓存中保存解析到的neighbour项。

函数neigh_create的参数

现在我们知道了什么情况会创建新的neighbour结构，下面我们看看创建时涉及的主要函数。

创建数据结构本身是用neigh_create函数，它的返回值是指向创建的neighbour结构的指针。下面是函数的原型和它的三个输入参数的介绍：

```
struct neighbour * neigh_create(struct neigh_table *tbl, const void *pkey,
                                struct net_device *dev)
```

tbl

表示使用的邻居协议。设置这个参数的方式很简单：如果调用者来自IPv4程序(也就是说，来自arp_rcv)，就设为arp_tbl等等。

pkey

表示L3地址。之所以称为pkey 是因为它在缓存查找中被用作查找关键字。

dev

与要创建的邻居项相关的设备。因为每个neighbour项都与一个L3地址相关联，并且后者总是与一个设备相关联，所以neighbour实例就与一个设备相关联。

新的neighbour数据结构是用neigh_alloc函数来分配存储空间，该函数也用于初始化一些参数，例如，嵌入的定时器、引用计数器、指向关联的neigh_table（邻居协议）结构的指针和对分配的neighbour结构数目的整体统计。

neigh_alloc函数使用邻居子系统初始化时建立的内存池（参见“协议初始化和清理”一节）。如果当前分配的邻居结构数目大于配置的阈值，并且接下来的垃圾回收器试图释放某块内存失败了，该函数就无法完成分配（参见“同步清理：the neigh_forced_gc函数”）一节。

在key_len的帮助下，pkey被复制到邻居数据结构中。key_len提供被复制的数据的长度。这是必要的，因为neighbour结构是被与协议无关的缓存查找函数使用，并且各种邻居协议表示地址的字节长度不同。

```
memcpy(n->primary_key, pkey, key_len);
```

而且，由于neighbour项中包括了对net_device结构中dev的引用，内核会使用dev_hold来对后者的引用计数器加1，以此保证该设备在neighbour结构存在时不会被删除。

邻居初始化

neighbour结构有两种初始化方式：一种由邻居协议执行，一种由设备执行。

```
if (tbl->constructor && (error = tbl->constructor(n)) < 0) {
    rc = ERR_PTR(error);
    goto out_neigh_release;
}
```

协议执行的初始化靠调用neigh_table->constructor函数完成，正如上面代码显示的那样，从函数的tbl参数中调用。第二十八章会解释ARP constructor如何完成这个工作。

设备执行的初始化工作由neigh_setup虚函数完成：

```
if (n->parms->neigh_setup &&
      (error = n->parms->neigh_setup(n)) < 0) {
      rc = ERR_PTR(error);
      goto out_neigh_release;
}
```

实际上只有少数设备定义了这个函数。例如，shaper虚拟设备（*drivers/net/shaper.c*文件中的一些旧代码已经被流量控制子系统废弃了，但保留这些代码是为了向下兼容）使用这个设置函数来保证它与ARP提供的一个特殊neigh_ops结构的实例相关联（参见“neigh->ops的初始化”一节）。一些WAN设备出于类似的原因使用一个设置函数。

最后neigh_create函数要设置创建的邻居项的confirmed字段，表示该邻居是可到达的。正常情况下，该字段由可到达性证明来更新，并且其值设为用jiffies表示的当前时间。但在这里，从新建的角度来说，neigh_create函数会把confirmed值减去一小段时间（reachable_time值的一半），这样就使得邻居状态能比平常和要求有可到达性证据时，稍快点转移到NUD_STALE态。

```
n->confirmed = jiffies - (n->parms->base_reachable_time<<1);
```

一旦邻居项被初始化后，就使用邻居协议提供的hash函数将它增加到主要缓存中。

删除邻居

删除neighbour结构的原因主要有以下三个：

- 内核企图向一个不可到达的主机发送封包。发生这种情况的原因有很多：该主机已关机、或者它的网线没有插好、或者它是个无线设备但是移动到了信号范围之外、或者它的网络配置出现问题、或许是有人为一台不存在的主机手工建立了一个缓存项。不管是什么原因，邻居子系统觉察到了传输不成功，就将相关的neighbour结构转移到NUD_FAILED状态。这样就可以使异步垃圾回收机制清除掉该neighbour结构，参见“异步清理：neigh_periodic_timer函数”一节。
- 与该邻居结构关联的主机的L2地址改变了（可能是由于它换了NIC），但它的L3地址还是原来的。这样一来，该neighbour结构的L2地址就过时了。一台有过时的邻居项的主机必须进入NUD_FAILED态，然后重新创建一个新的邻居项（注7）。
- 该邻居结构存在时间太长，且内核需要它所占用的内存。因此使用垃圾回收将其删除，参见“同步清理：the neigh_forced_gc函数”一节。

在第二十六章的“NUD状态之间的转换”一节中介绍的NUD算法负责将该邻居状态转移到NUD_FAILED态。异步垃圾回收由neigh_periodic_timer函数执行，该函数与neigh_table->gc_timer定时器有关（详细内容参见“定时器”和“垃圾回收”小节）。

只有当一个结构的引用计数器为0时，才会删除该结构。只有neigh_release函数才调用实际执行删除任务的函数neigh_destroy。每次对一个结构的引用释放后，就调用一次neigh_release函数，它会将该结构的引用计数器减1。当引用计数器变为0时，neigh_release就调用neigh_destroy函数来真正删除该结构。

注7：一些设备驱动程序可以让管理员临时改变MAC地址（就是说，在一定时间后MAC地址会变为原来的值），或者永久性改变。MAC地址变化限制在特殊情况下使用，一般应用是不需要的。

```
static inline void neigh_release(struct neighbour *neigh)
{
        if (atomic_dec_and_test(&neigh->refcnt))
                neigh_destroy(neigh);
}
```

neigh_destroy 完成下列的任务：

- 停止所有未决的定时器。这是一种皮带与吊带式（belt-and-suspenders）预防措施。理论上，在执行 neigh_destroy 函数时，不应该有任何定时器是未决的，因为neigh_release调用neigh_destroy的条件是引用计数器为0，而且定时器运行时总是保持对要删除的数据结构的引用。
- 释放所有对外部数据结构的引用，例如，关联的设备及缓存的 L2 帧头。参见图 27-1 和图 27-10。

  本章后面的“L2 帧头缓存”一节会解释这种缓存的目的，并给出了neighbour结构和包含L2帧头信息的hh_cache结构之间的关系。每个hh_cache结构直接和一个neighbour项连在一起。因此一旦neighbour项被删除或进入NUD_FAILED态，相关的hh_cache结构也不应该再使用了。当删除一个neighbour项时，要解除缓存中与之相关的任何 hh_cache结构的链接，并且如果这些hh_cache结构的引用计数允许的话，就释放其空间。同时 neigh_destroy 函数会设置缓存的L2 帧头中的 hh_cache->hh_output 字段为 neigh_blackhole（对这个函数，参见“neigh ->output使用的函数”一节）。之后，使用该neighbour项的所有传输都会悄悄失败，并会丢弃相关封包。在L3 层，丢弃封包的结果可以在“邻居协议和L3 传输函数的交互”一节中看到。
- 如果一个邻居协议提供了 destructor 方法，该邻居协议就会执行这个方法自己清理邻居项。
- 如果 arp_queue队列非空，就要将其清空（删除其所有元素）。“出口队列”一节中有对 arp_queue 队列的描述。
- 将表示主机使用的 neighbour 项总数的全局计数器减 1。
- 释放该 neighbour 结构（将其占用的内存空间返还给内存池）。

垃圾回收

垃圾回收涉及清除不再使用的资源的系统进程。和许多 Linux 内核子系统（网络子系统或其他子系统）类似，邻居子系统也维护着一个定时器。该定时器周期性地运行，当其过期时，就执行某个函数来清理不用的数据结构。

邻居基础结构使用的垃圾回收算法有两个主要组成部分：

同步清理

当邻居基础结构需要分配一个新的neighbour基础结构 ，并且这种结构专用的内存池已经用完时，内核就立即执行同步清理。

异步清理

异步清理是周期性执行的，目的是为了删除某段时间内没有用过的neighbour结构。该时间段的长短是可以配置的，其值保存在gc_staletime变量中。“定时器”一节中讲到的函数neigh_periodic_timer实施这个规则。

在邻居子系统中使用这种相对复杂的回收机制，是因为设计者认为，两种方式比单一方式（例如，当一个结构的引用计数为0时，就删除该结构）效率更高。异步清理会释放没有价值的结构，而同步清理是牺牲一些较少使用的邻居项来释放部分内存。因此，两种清理方式选择符合清理条件的结构的标准是不同的。

注意，异步清理可以由其他外部子系统触发。例如，当路由子系统无法在其缓存中插入一个新的路由项时，它就会删除没有用的缓存项（参见第三十三章中的对rt_intern_hash函数的介绍），这个过程也直接导致相应的neighbour结构也被释放。

调整垃圾回收行为的参数有：

- neigh_table中的参数有：
  - gc_interval
  - gc_thresh1, gc_thresh2, gc_thresh3
  - last_flush
  - gc_timer
- neigh_parms中的参数有：
  - gc_staletime

下面两个小节会解释这些参数的含义和用法。关于这些变量的相关信息可以查看“neigh_table 结构”一节、“neigh_parms 结构”一节及第二十九章的表29-3。

图27-7和图27-8给出了neigh_periodic_timer函数和neigh_forced_gc函数的内部结构，下面介绍这两个函数。

同步清理：the neigh_forced_gc 函数

图27-7给出了neigh_forced_gc函数的内部结构。

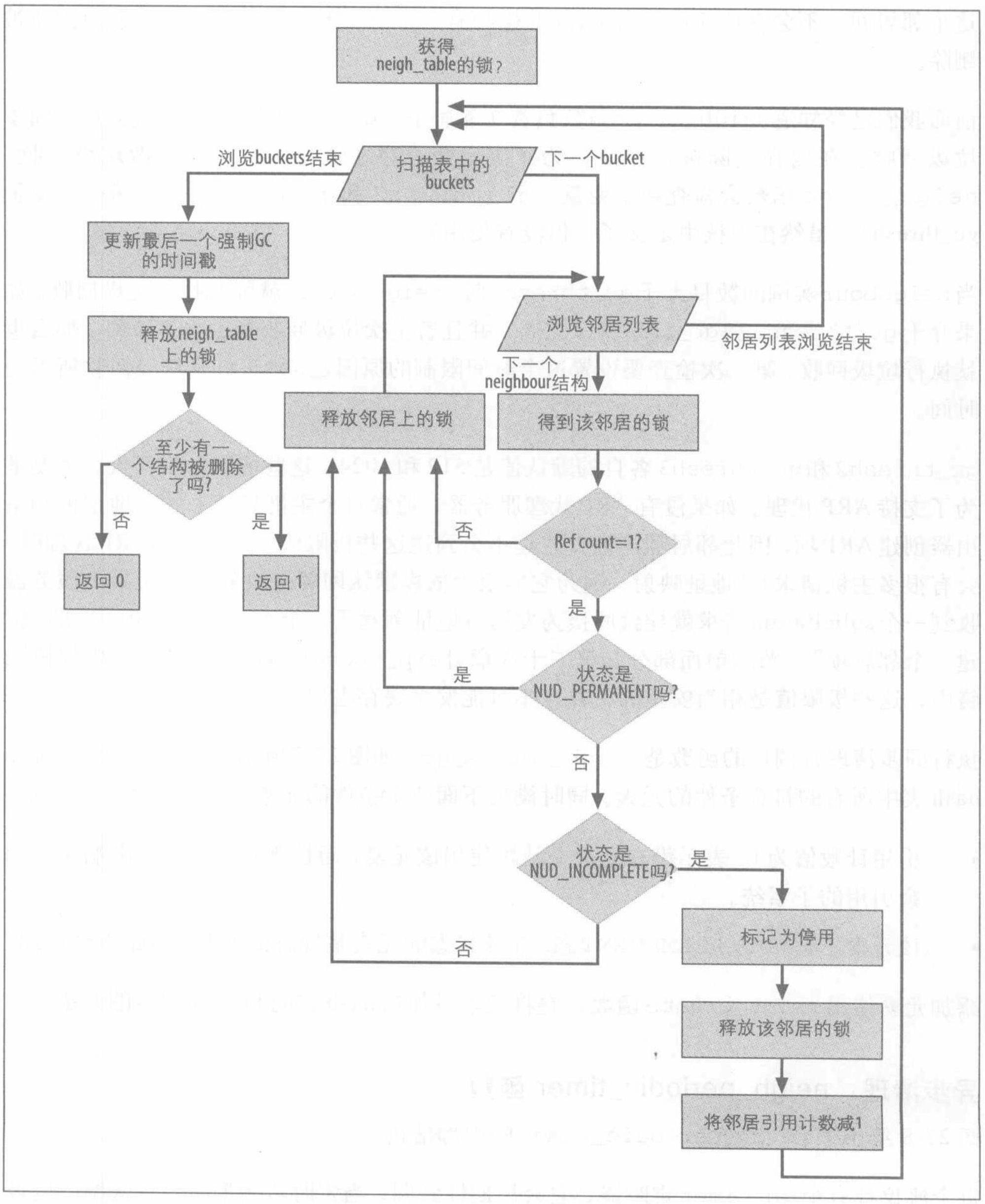


图 27-7：neigh_forced_gc 函数

如果没有内存分配给新的neighbour实例，主机就无法向该邻居传输任何封包，因为在缓存中没有相应的neighbour结构。若对此没有相应的对策，那么后果是很严重的：和

这个邻居间将不会有任何通信发生，除非碰巧有一个neighbour结构由于某些原因要被删除。

前面我们已经知道neigh_alloc函数负责在邻居子系统中分配内存，自然就会启动同步垃圾回收。在内存实际被耗尽前，为了决定是否存在危险，是否需要做垃圾回收，neigh_alloc函数会检查两个变量：gc_thresh2和gc_thresh3。（另一个变量gc_thresh1，虽然在内核中定义了，但没有使用）。

当neighbour实例的数目大于gc_thresh3时，neigh_alloc就强制执行垃圾回收。如果介于gc_thresh2和gc_thresh3之间，并且若上次垃圾回收已过去5秒钟，那么也就执行垃圾回收。第二次检查要设置这个时间限制的原因是5秒正好是垃圾回收所需的时间。

gc_thresh2和gc_thresh3各自的默认值是512和1024。这些值看起来很大，主要是为了支持ARP代理。如果没有ARP代理服务器，通常每个主机只为一些本地主机和路由器创建ARP项，因此邻居项的数目永远不会到达这些极限值。但当使用ARP代理时，会有很多主机请求L3地址映射，因为它们很少依靠默认网关来映射。代理ARP服务器收到一个solicitation请求就导致间接为发送方地址创建了一个neighbour项。参见“创建一个邻居项”一节的前面部分和第二十八章对arp_process的介绍。在一个中规模网络中，这些极限值是相当安全的，并且不可能发生缓存溢出。

执行同步清理时调用的函数是neigh_forced_gc，如图27-7所示。该函数会删除缓存hash表中所有的符合条件的元素。同时满足下面两个条件的元素就是符合条件的元素：

- 引用计数值为1，表示没有函数或结构使用该元素，而且删除该元素不影响保持剩余引用的子系统。
- 该元素不是NUD_PERMANENT态。在该状态的元素是静态配置的，因此不会过期。

添加元素使用neigh_create函数，它将元素添加在hash表的bucket列表的开头。

异步清理：neigh_periodic_timer 函数

图27-8给出了neigh_periodic_timer的内部结构。

每个协议都有个gc_timer定时器，它会周期性到期。当定时器到期时，它就调用垃圾回收函数neigh_periodic_timer。内核实际上调用的是neigh_table结构（每个邻居协议都有这样一个结构）中的一个字段指定的函数。因此，理论上每个协议都有它自己的垃圾回收处理函数，但是实际上，在neigh_table_init函数中将所有协议的该字段都初始化为同一个函数。

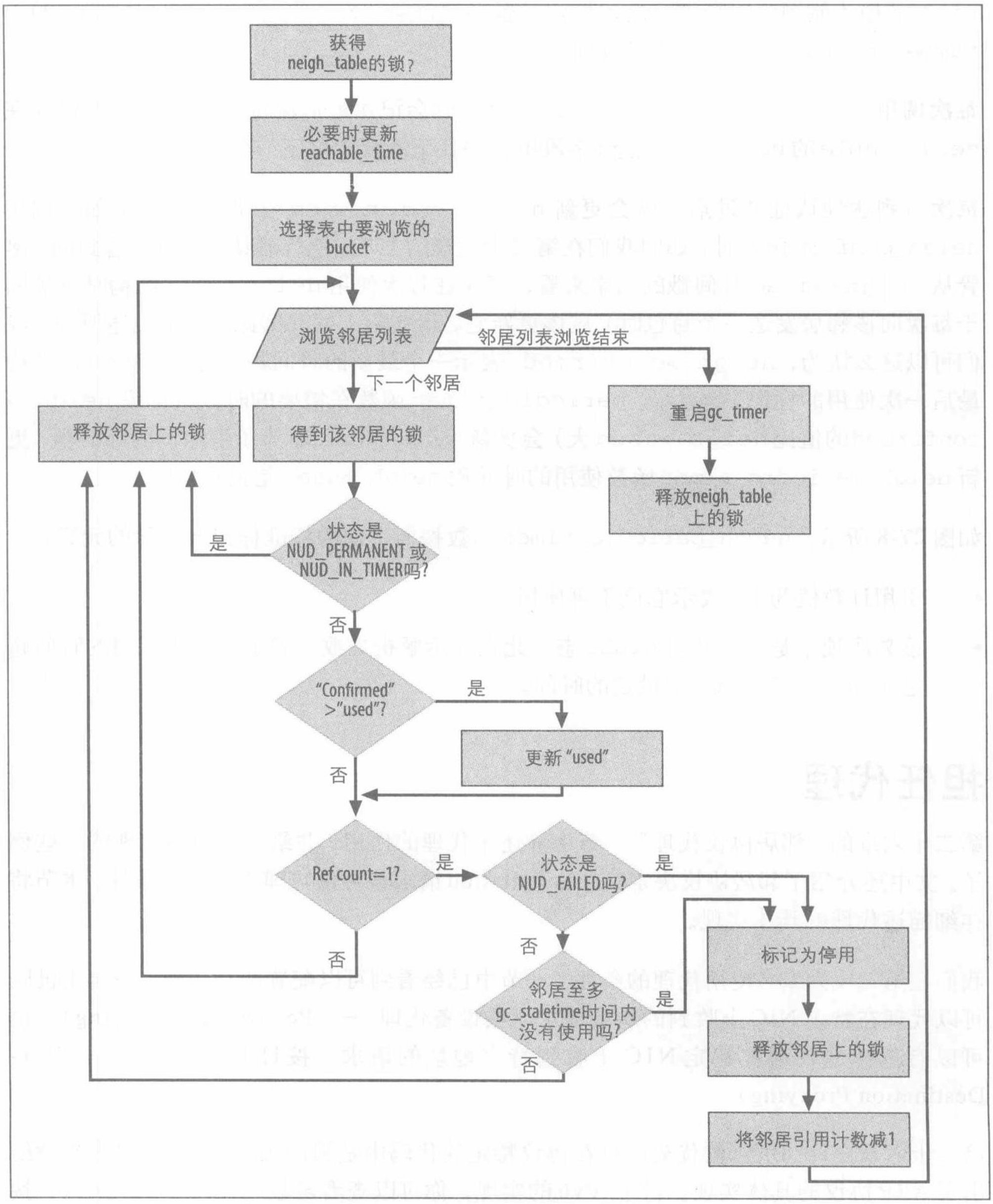


图 27-8：neigh_periodic_timer 函数

gc_timer 定时器的过期频率根据 hash_buckets 表的大小决定：由于调用 neigh_periodic_timer 时，它每次仅浏览表中的一个 bucket，因此每隔 base_reachable_

time/2秒才能浏览整个表一次，该定时器必须每隔（base_reachable_time/2）/ *number_of_buckets*秒后才能过期。

每次调用neigh_periodic_timer 时，它会记住最后浏览的bucket，并保存在neigh_table的hash_chain_gc字段中，并且下次会浏览下一个bucket。

每次可到达性认证通过后，就会更新neigh->confirmed时间戳。例如，调用neigh_confirm函数时，如同我们在第二十六章的“可到达性确认”一节中看到的。尽管从neigh->used时间戳的名字来看，好像在每次使用neighbour 结构时（等同于每次向该邻居发送一个封包时）应该更新它，但实际上并非如此。基于上述原因，我们可以这么认为，neigh->confirmed 表示一个最新的时间戳，是neighbour结构最后一次使用的标记。neigh_periodic_timer函数在需要的时候（如果neigh->confirmed的值比neigh->used大）会更新neigh->used。为了消除旧的邻居项，更新neigh_periodic_timer函数使用的时间戳neigh->used是很重要的。

如图27-8所示，neigh_periodic_timer函数按照下面的标准标记要删除的元素：

- 引用计数值为1，表示它已不再使用。
- 该邻居项不是为NUD_FAILED态，此态表示解析失败，就是由于未使用的时间超过了gc_staletime中设定的时间。

担任代理

第二十六章的“邻居协议代理”一节中描述了代理的作用，并给出了使用代理的一些例子。文中还介绍了邻居协议决定某个solicitation请求是否由代理来处理的条件。本节将详细描述代理的具体实现。

我们在第二十六章“使用代理的条件”一节中已经看到可以配置两种代理：一台主机既可以代理在特定NIC上收到的所有请求（按设备代理 —— Per-Device Proxying），也可以有选择地代理在特定NIC上收到特定地址的请求（按目的地代理 —— Per-Destination Proxying）。

第二十六章图26-8所示的优先顺序在协议特定的代码中是强制实施的。在第二十八章给出了ARP协议的具体实现。对于IPv6的实现，你可以查看函数neigh_recv_ns。“按设备代理和按目的地代理”一节会给出这两种代理类型的详细描述。

在研究代码之前，先介绍一下代码中广泛使用的命名约定。邻居子系统包含许多成对的函数和数据结构，唯一区别是它们的名字是否以字母*p*开头（例如，neigh_lookup和

pneigh_lookup)。这里，*p* 代表 *proxy*。因为代理截获的地址是用不同的方式处理，有专门的一组函数来负责处理这些地址。

延迟处理 Solicitation 请求

由代理负责的Solicitation请求既可以立刻处理，也可以在一段可配置的延迟后处理。引入延迟的主要原因是为了使代理的邻居项的优先级比其他授权的主机（例如，实际拥有 solicited L3 地址的主机）的低。发出请求的主机将收到的第一个应答锁定一小段时间，然后等待，一旦另一个应答到来，就能提高该主机的优先级；详细描述见第二十八章的“普通处理的结尾”一节。

延迟的时间是一个介于0和一个可配置的值 proxy_delay（参见 pneigh_enqueue 函数）之间的随机数。随机数的使用可以降低多个主机同时发出请求，并导致拥塞的可能性。例如，如果某个网络掉电了，当其恢复后，网络内的数百台主机会同时启动，此时，所有主机会请求同一组服务器或默认网关的地址。随机数延迟可以消除可能导致的流量峰值。

为了使用延迟，邻居子系统会创建一个保存入口 solicitation 请求的队列和一个定时器。在经过一段设定的延迟后，定时器就会到期，并且触发一个特殊的函数将队列中的相应元素出队列。然后就处理出列的元素（solicitation 请求）好像刚从网络上接收到它们一样。

图 27-9 描述出了刚才介绍的模型。

与处理代理延时相关的主要变量和虚函数有：

- 来自 neigh_table（协议参数）

  proxy_queue

  临时缓冲入口 solicitation 请求的队列。元素被添加到队尾。当 proxy_queue 队列达到它的最大长度 proxy_qlen（后面会介绍）时，就会丢弃新元素；该队列不会替换掉旧元素。

  proxy_timer

  用于执行延时的定时器。定时器由 neigh_table_init 初始化，默认处理函数是 neigh_proxy_process。

  proxy_redo

  处理出列请求的函数。如图 27-9 所示，它只包括一个函数调用，调用的函数就是处理最新收到的封包的函数。

- 来自 neigh_parms（设备参数）

  proxy_delay

  用于启动定时器的可配置延迟。

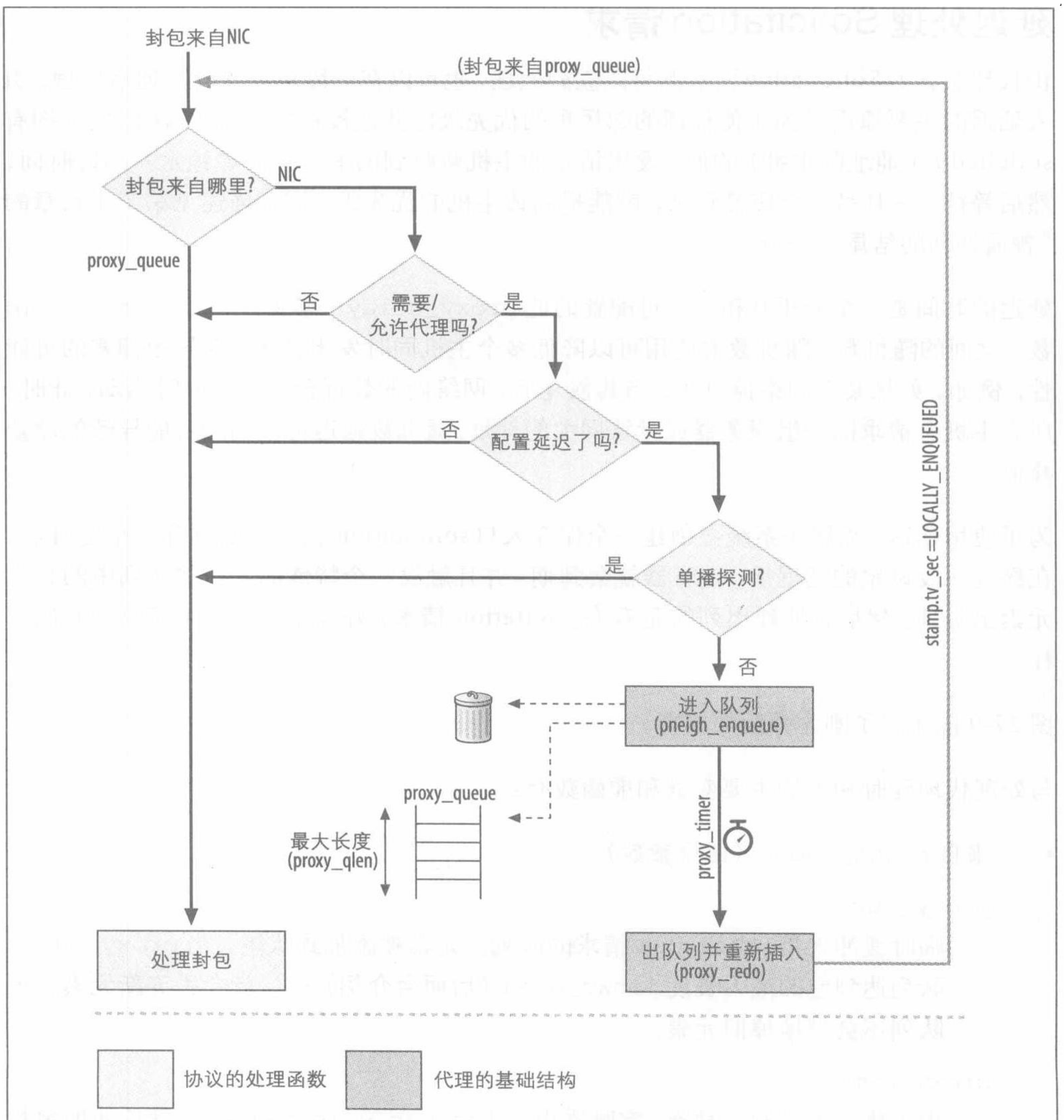


图 27-9：协议代理处理函数的一般模型

  proxy_len

  临时存储队列的最大长度。

对相关重要数据结构的详细描述可以参考第二十九章。

每个协议还有一个私有队列（neigh_table→proxy_queue），该队列被所有使用该协议的NIC共享。使用pneigh_enqueue函数将新元素添加到proxy_queue队列中。proxy_ queue是一个经过排序的双向链表，可以让定时器很容易地按照入列的时间顺序处理未决的请求。当pneigh_enqueue要增加一个新请求时，如果proxy_queue队列中已经包含了该请求，该函数会重启定时器。它既可以在当前安排的超时期后到期，也可以在新元素要求的超时期后到期，哪个先到算哪个。

对于从网络设备上收到的新solicitation请求和从代理队列中出来的solicitation请求，Linux都使用相同的函数来处理，如图27-9所示（IPv4和IPv6都是这样处理）。由于这个原因，需要有一个函数来区分下面两类封包：

- 刚刚收到的封包，需要被加入到proxy_queue队列中。
- 一段延迟后从代理队列中出来的封包。

Linux通过在sk_buff缓冲区结构中的一个字段设定一个特殊值来区分这两类封包：skb->stamp.tv_sec。该字段是个时间戳，当首次收到封包时，被netif_rx（见第十章）初始化为本地接收时间。之后才调用邻居协议处理函数（见第十三章），因此，协议处理函数访问skb->stamp.tv_sec时，得到的值通常是非负的。但是，当一个封包元素被放进proxy_queue时，它的stamp.tv_sec被设为LOCALLY_ENQUEUED，其值等于-2。这样，当该封包被代理定时器出列，并被送到邻居协议处理函数时，处理函数就知道该封包来自代理队列，如图27-9所示。

若proxy_delay值为0，就不使用缓冲区，而是立刻处理请求。若proxy_delay值为非零，请求就被送到proxy_queue队列中。前面已经提到，一个介于0和proxy_delay之间的随机数作为延迟时间，为了避免来自不同主机的同时请求造成的扩散（注8）。由于这个原因，请求的处理顺序也许会和它们的接收顺序不同，但这并不是问题。

按设备代理和按目的地代理

当在一个设备上完全启动代理时，那么状态信息就很简单：该设备只需要关联一个标识，这个标志表明是否启动了代理。另一方面，按目的地代理需要保存被代理的地址。这些L3地址（本应该侦听这些solicitation 请求的主机的地址）保存在hash表neigh_table->phash_buckets中（如图27-2所示），用pneigh_lookup函数（neigh_lookup函数在代理程序中的相应函数）查找。

注8：随机延迟是RFC2461讨论的主要问题之一。该文档主要处理IPv6/ND协议，但Linux使用同样的方式实现IPv4/ARP协议。

如同neigh_lookup函数（在“缓存”一节中有介绍），pneigh_lookup函数有一个输入参数，可以在查找失败时，强制建立一个neighbour结构。与phash_buckets和hash_buckets不同，它们没有容量限制，而且也没有垃圾回收机制，因为给phash_buckets表中的元素设置不同的属性没有意义：这些元素的地址是明确配置为要被代理的，因此它们一直是有效的，直到再次被明确指定不用代理。在IPv4中，这些地址只能手动配置。在IPv6中，在一定条件下可以使用协议配置这些地址。

使用邻居协议可以动态往hash表中添加新邻居项，或者使用管理命令静态添加（参见第二十九章的“邻居的系统管理”一节的内容）。用pneigh_delete函数可以删除表中的项。

L2帧头缓存

从一台主机发往另一台主机的所有封包的L2帧头都是相同的。L3包头可能不同，因为不同的封包有不同的ID；如果产生碎片，也会有不同的碎片偏移量（fragment offsets）；并且连续两个封包也有一些不同。因此，内核不必为缓存L3包头而费心，但是它要缓存L2帧头。复杂的L2协议的帧头也可能并不一致，但大部分协议是一样的，例如，Ethernet的帧头（见第十三章对Ethernet的详细介绍）。当使用L2帧头缓存后，出口设备的设备驱动程序必须对其支持。

当向一个目的地发送第一个封包后，驱动程序就将其L2帧头保存在名为hh_cache的专用结构中。如果下一次若有封包发往同一个邻居，发送者就不需要在挨个填充其L2帧头的每个字段，只需要从缓存中拷贝一个帧头就可以。hh_cache结构与其他邻居协议结构之间的联系在“主要的数据结构”一节中有介绍，见图27-2。该结构更详细的描述在第二十九章的“hh_cache结构”一节。

L2帧头缓存和在L3的路由子系统缓存有紧密的联系（见第三十三章）。如图27-1所示，IPv4路由缓存中的每个dst_entry元素都包含一个指向与下一跳相关的neighbour结构的指针及一个hh_cache帧头缓存列表。每个邻居可以有多个帧头被缓存，但通常只缓存一个。图27-1给出了一个Ethernet帧头的例子（注9）。

在图27-10中给出了不同缓存中使用的数据结构之间的关系。图中还给出一种简单场景：两个LAN由路由器连在一起。从主机C的角度看，通过下一跳路由器可以到达主机A、主机B和路由器。如果主机C已经与上述三个主机进行了数据交换，那么它的路由缓存

注9： Ethernet帧头不包括前同步码（preamble）和校检和（checksum），因为它们由NIC负责处理。

中为每个主机保持一个dst_entry结构（也就是路由缓存元素）。正如前面所讲，每个dst_entry结构包含一个到相关的下一跳地址（本例中，就是路由器）的neighbour结构的链接。neighbour和dst_entry结构都有一个指向hh_cache项的链接。由于这三个主机（主机A、主机B和路由器）通过同一个下一跳地址都可以到达，所以缓存一个L2帧头就足够了。

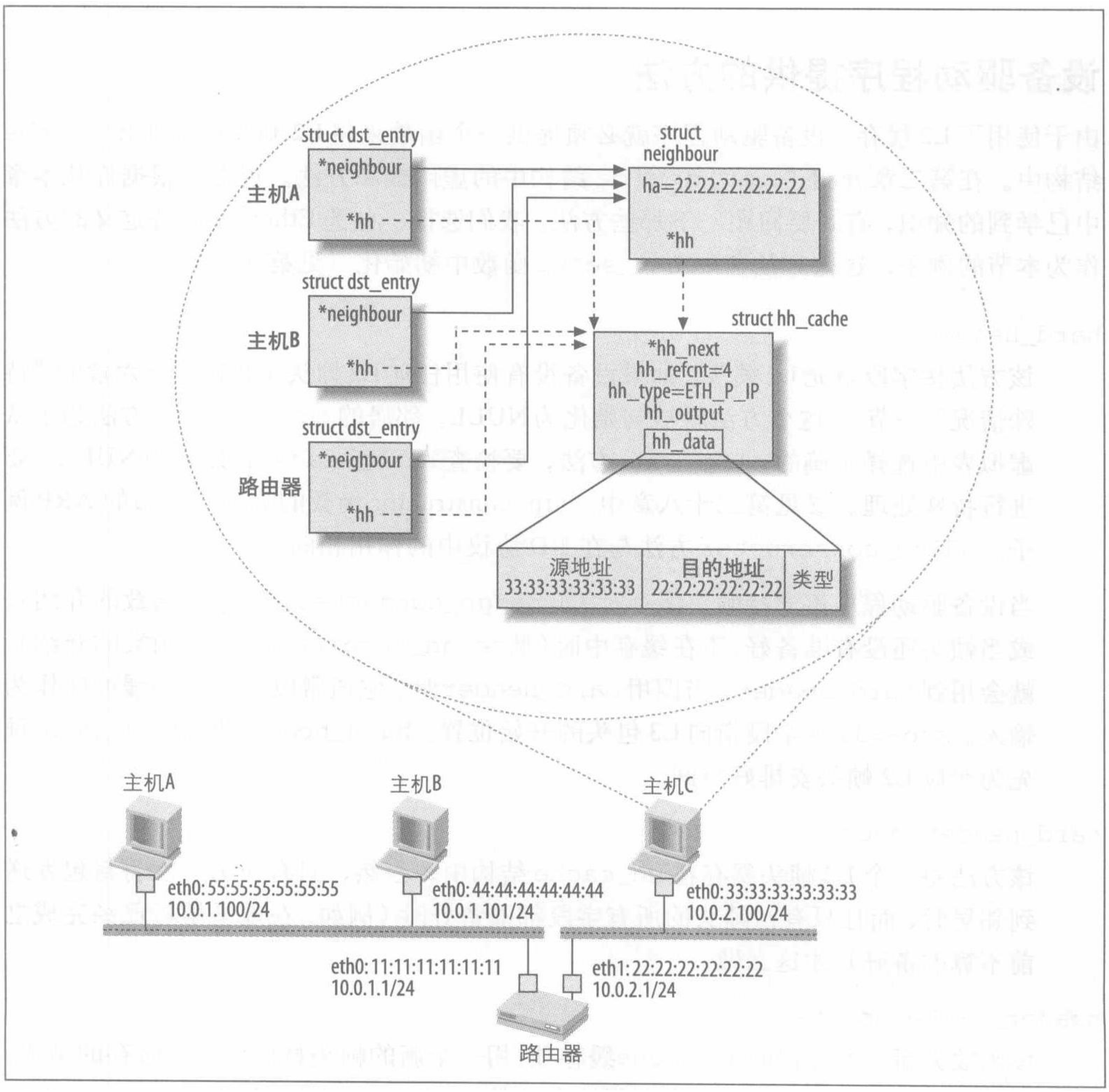


图27-10：路由使用的缓存例子

图27-10中hh_cache结构的引用计数（hh_refcnt）的值是4，这也是图中虚线的数目。dst_entry结构中保持的引用和neighbour结构中保持的引用都是通过neigh_hh_

init函数设置的，“路由缓存和L2帧头缓存之间的联系”一节中有相关描述。hh_cache结构引用计数通过直接调用atomic_inc来递增。正如“邻居结构的引用计数”一节中描述的，内核为neighbour结构提供了一个特殊的包裹函数。

L2帧头缓存的使用对L2协议是透明的，在后面的“邻居协议和L3传输函数的交互”一节中有介绍。

设备驱动程序提供的方法

由于使用了L2缓存，设备驱动程序就必须提供一个函数来将L2帧头保存在hh_cache结构中。在第二章介绍了net_device结构中的虚函数和方法。现在，根据你从本章中已学到的知识，有必要回顾一下那些方法。我们选择一些为Ethernet设备定义的方法作为本节的例子，这些方法在ether_setup函数中初始化（见第八章）。

hard_header

该方法按字段填充L2帧头。如果设备没有使用任何L2帧头（见第二十六章的“特殊情况”一节），这个方法就被初始化为NULL。邻居的constructor方法为了从虚拟表中选择正确的neigh_ops方法，要检查hard_header；如果为NULL，要进行特殊处理。参见第二十八章中“arp_constructor函数的启动”一节的ARP例子。ndisc_constructor方法与在ND协议中的作用相同。

当设备驱动程序不支持帧头缓存时（见neigh_connected_output函数的介绍），或当帧头还没有准备好，不在缓存中时（见neigh_resolve_output函数的介绍），就会用到hard_header。当调用hard_header时，它通常以一个skb缓冲区作为输入。skb->data字段指向L3包头的开始位置。hard_header使用skb_push预先为对应L2帧头安排好空间。

hard_header_cache

该方法将一个L2帧头缓存在hh_cache结构中。当然，只有在第一次将封包发送到邻居时，而且只有当帧头的所有字段都准备好时（例如，在地址解析已经完成之前不算准备好）才这么做。

header_cache_update

该函数更新一个存在的hh_cache缓存项，用一个新的帧头替换该项中缓存的帧头。通常从neigh_update_hhs中调用这个函数。neigh_update使用neigh_update_hhs对邻居项进行更新（参见“邻居信息的更新：neigh_update”一节）。

hard_header_parse

从一个缓冲区中取出源L2地址，然后返回该地址的长度。

rebuild_header

该函数只是为了兼容2.2版以前的内核设备驱动程序。使用这个函数的设备不能使用dst_entry->neigh中的缓存的已经解析的地址。

路由缓存和L2帧头缓存之间的联系

当刚刚创建一个邻居项时，neigh->output指向neigh_resolve_output函数，该函数负责将该邻居与一个L2帧头关联。这样，L3层的传输函数（在第二十一章和“邻居协议和L3传输函数的交互”一节中有介绍）可以透明地进行地址解析。

下面是来自neigh_resolve_output函数的代码片断，其中dst是“主要数据结构”一节介绍的路由表缓存项。

```
if (dev->hard_header_cache && !dst->hh) {
        write_lock_bh(&neigh->lock);
        if (!dst->hh)
                neigh_hh_init(neigh, dst, dst->ops->protocol);
        err = dev->hard_header(skb, dev, ntohs(skb->protocol),
                               neigh->ha, NULL, skb->len);
        write_unlock_bh(&neigh->lock);
} else {
        read_lock_bh(&neigh->lock);
        err = dev->hard_header(skb, dev, ntohs(skb->protocol),
                               neigh->ha, NULL, skb->len);
        read_unlock_bh(&neigh->lock);
}
```

如果设备可以使用帧头缓存（也就说，设置了hard_header_cache），但是帧头还没有准备好（!dst->hh），则neigh_resolve_output必须初始化并缓存L2帧头。通过调用neigh_hh_init函数来进行这个过程，该函数也创建hh_cache缓存项，并将其链接到dst->hh路由表缓存项上（图27-10中的虚线就是这个操作）。

相反，如果设备不支持帧头缓存，就使用hard_header来填充L2帧头。

在上述两种情况下，都会在一个锁的保护下访问neighbour结构。但是，第一种情况是独占性访问，因为要写帧头；第二种情况是共享式访问的。注意，在第一种情况中，一旦neigh_resolve_output函数获得锁，它就要检查dst->hh的状态。这是用锁来避免竞争条件的标准方式；在这种情况下，由于在上一次检查和这次获得锁之间，dst->hh可能会被另一CPU初始化，所有要检查dst->hh的状态。

有关neigh_resolve_output函数的内部结构，可以参考图27-13。

缓存失效和更新

一个缓存的帧头可能包含许多不同的字段，但是有两个字段是最有可能发生变化的并且使该缓存项失效。这两个字段就是源地址字段和目的地址字段。

当一个本地设备改变了自己的L2地址，所有与此地址关联的缓存帧头就过期了。当邻居子系统知道L2地址变化后（NETDEV_CHANGEADDR，在“使用neigh_changeaddr（netdevice通知链）更新”一节中有介绍），它就会刷新与该地址相关的所有neighbour项，从而使所有相关的L2帧头缓存项失效。

当系统探测到邻居的L2地址发生变化，它就调用neigh_update_hhs函数来处理。这个函数会依次更新那个邻居使用的所有的缓存帧头。neigh_update_hhs函数调用设备驱动程序提供的header_cache_update函数（见“设备驱动程序提供的方法”一节）来完成更新。参见“邻居信息的更新：neigh_update”一节。

协议初始化和清理

每个邻居协议都有一个初始化函数。如果邻居协议包含在内核中，就在系统启动时执行那个初始化函数；如果协议是被编译为模块，则在载入模块时执行初始化函数。和其他内核子系统一样，初始化函数会将该子系统需要的全部资源分配给合适的函数。表27-1中列出了在Linux内核中实现的四个邻居协议的初始化函数。

表27-1：邻居协议初始化/清理函数

协议	初始化函数	清理函数	所在文件
ARP	arp_init[a]	None	*net/ipv4/arp.c*
Neighbor Discovery (ND)	ndisc_init	ndisc_cleanup	*net/ipv6/ndisc.c*
DECnet	dn_neigh_init	dn_neigh_cleanup	*net/decnet/dn_neigh.c*
ARP over IP (clip)[b]	atm_clip_init	atm_clip_exit	*net/atm/clip.c*

a: 在第二十八章的“初始化ARP协议”一节中有对arp_init孩子数的介绍。

b: clip表示ARP中的特殊情况，并不是一个独立的协议。因此，它不像其他三个协议，不需要使用neigh_table_init函数向内核注册，但是它通过自己完成初始化（例如，内存池的分配）。基本上，它会初始化它的neigh_table结构，并且让ARP协议（arp_bind_neighbour）负责处理这个结构。

下面是这些函数要完成的一些共同的任务：

- 用neigh_table_init函数初始化neigh_table结构。
- 在需要的时候（通常由管理员设置），在*/proc*文件系统中注册一组变量。
- 注册一个协议处理函数。IPv4为ARP协议注册arp_rcv函数（见第十三章）。IPv6中的邻居协议是通用协议ICMPv6的一部分，因此IPv6注册ICMPv6协议的处理函数，这个函数调用arp_rcv在IPv6中的对应函数（ndisc_rcv）来处理那些必须使用IPv6的邻居协议（ND）的ICMPv6消息。

neigh_table_init完成下列工作：

- 为neighbour结构分配预备的内存池。
- 分配一个neigh_statistics结构来收集协议的统计信息。参考第二十九章的“neigh_statistics结构”一节。
- 分配两个hash表：hash_buckets和phash_buckets。这两个表分别作为解析过的地址关联缓存和代理的地址数据库，如图27-2所示。
- 在*/proc/net*中建立一个文件，用于转储缓存的内容。文件名来自neigh_table->id。
- 启动gc_timer垃圾回收定时器，参考“垃圾回收”一节。
- 初始化（但是不起动）proxy_timer代理定时器和相关的proxy_queue队列。参考“延迟处理Solicitation请求”一节。
- 添加neigh_table结构到neigh_tables全局列表中。后者由一个锁保护，如图27-2所示。
- 初始化其他一些参数，例如reachable_time。

当一个协议使用模块方式运行，并且模块被卸载时，会调用neigh_table_clear函数来撤消neigh_table_init在初始化时做的工作；并且会清理在协议生存期内分配给该协议的任何资源，例如，定时器和队列。

图27-1给出了协议清理函数使用neigh_table_clear来清理协议资源的流程。IPv4是唯一不能编译为模块的协议，因此ARP不需要清理函数。

与其他子系统的交互

邻居子系统与其他子系统是互相影响的，当发生特定事件时，都会产生并接收相关通知。下面列出了一些与邻居子系统有联系的其他子系统：

路由

路由层的缓存和邻居层的帧头缓存之间的联系在“L2帧头缓存”一节中有详细介绍。

流量均衡（TEQL – Traffic equalizer）

TEQL是一种流量控制的排队规则，可以用IPROUTE2包中的*tc*命令对其配置。这个特征将一些L3的链路组合在一起，在当给一个目的地传送封包时，循环使用这些链路。对邻居协议的影响是，对单个IP地址（主地址）的解析事实上可能引起多个附属IP地址的解析。

由于这个组中的每个链路必须对L3到L2地址的映射进行绑定，因而当第一个循环时，就需要对组中的地址一个接一个地进行解析、绑定。

IPsec

Ipsec定义了一系列应用到一个封包（当它被传输时）上的变换（transformation），这样就能对其加密传输。由于这个原因，如果在图27-1中加入Ipsec的影响，那么在一个链表中将有多个dst_entry结构，但只有最后一个才是指向neighbour结构的指针（见第三十三章的图33-5）。

Netfilter（iptables）

Netfilter钩子被放在能影响入口封包、出口封包和转发封包的多个地方；和这些在所有流量上的潜在影响一样，Netfilter钩子也会影响邻居层的solicitation请求和应答。在Netfilter和邻居协议间的交互操作是独立于邻居基础结构的，部分原因是因为不同邻居协议在网络栈中所处的层不同。

第二十八章的图28-13给出了Netfilter和ARP之间交互时使用的三个专用钩子点：NF_ARP_IN、NF_ARP_OUT和NF_ARP_FORWARD。ND与ARP不同，它位于IPv6的上层，因此它可以使用IPv6流量使用的默认钩子来过滤封包，这些钩子有：NF_IP6_PRE_ROUTING、NF_IP6_POST_ROUTING、NF_IP6_LOCAL_IN和NF_IP6_LOCAL_OUT。要了解这些钩子在IPv6网络栈中的位置，可以参考第十八章的图18-1中的相应描述。

在下面的几个小节中，我们将从邻居子系统的角度看看这些交互的实现。

邻居层产生的事件

当一个邻居被声明是不可到达时，就会进入NUD_FAILED态，邻居层就要执行neigh_ops→error_report函数。该函数会将这个事件通知上层协议。例如，在ARP失败的情况下，就要通知IPv4层。所有通知动作都是由neigh_timer_handler负责的，相关的定时器处理函数见“定时器”一节。

邻居层接收的事件

正如我们所知道的，邻居子系统维护的邻居项中，不管什么时候只要邻居项中有一个主要元素（L3 地址、L2 地址或接口设备）变化了，那么该项也就失效了。此时，内核必须确保邻居协议能够知道这些信息是否发生变化。这个工作是由邻居子系统提供的两个函数完成的：

`neigh_ifdown`

其他的内核子系统要调用该函数，以通知邻居子系统有关设备和L3 地址的变化。L3 地址改变的通知由 L3 协议送出。

`neigh_changeaddr`

当一个本地设备的L2 地址发生变化时，邻居协议调用该函数来更新协议的缓存。为了得到这些事件的通知，每个协议可以向内核注册。参考第二十八章的“收到的事件”一节中有关 ARP 的例子。当使用用户命令改变一个设备的硬件地址时，L2 地址改变的通知是由内核发出的。

使用 neigh_ifdown 更新

图27-11 中总结了能生成邻居协议感兴趣的外部事件的行为和函数。其中的主要事件是：

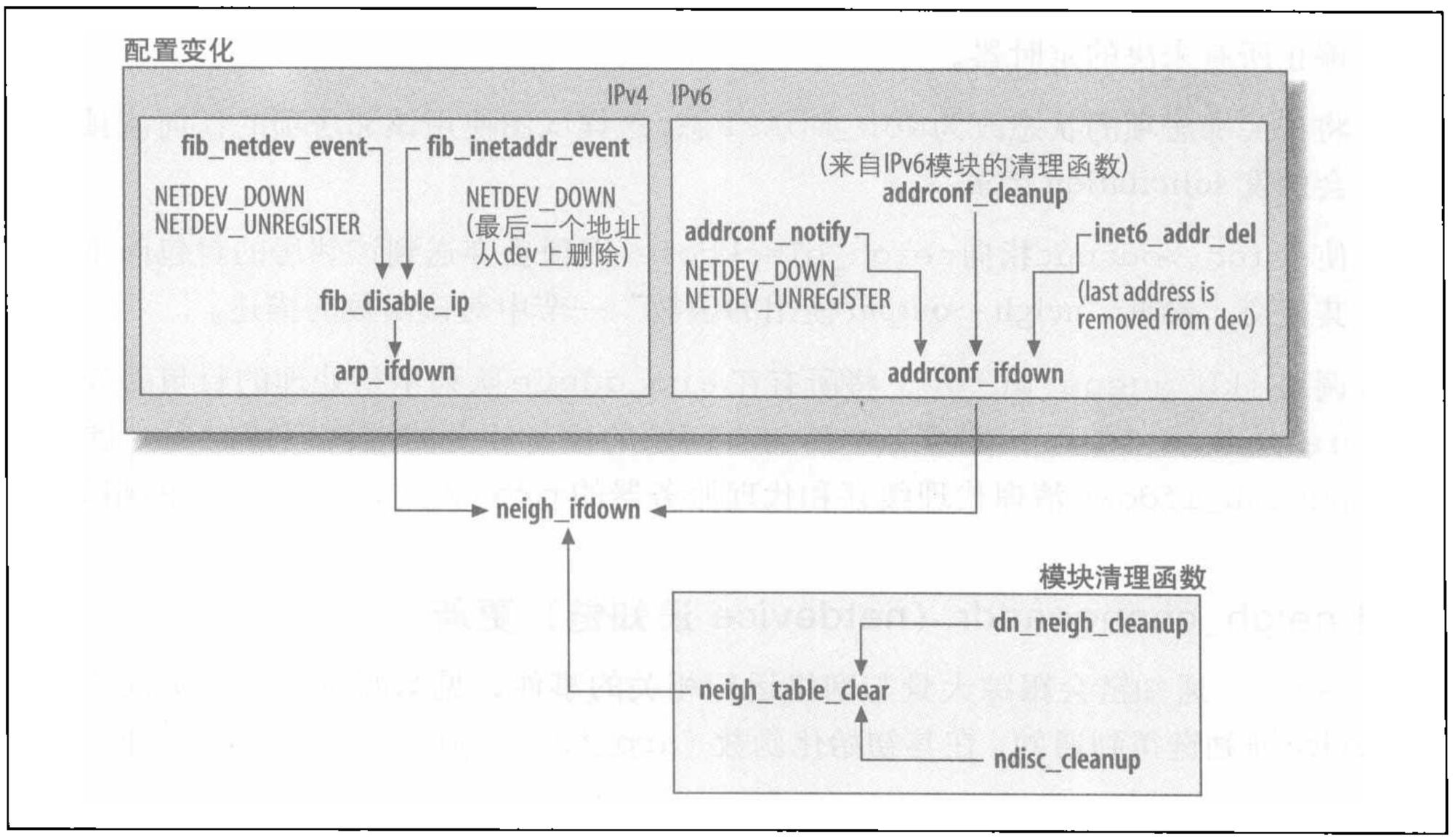


图 27-11：调用 neigh_ifdown 时的上下文

设备关闭

每个邻居项都与一个设备相关联。因此，如果该设备停止运行了，那么所有与之相关的邻居项都要被删除。更确切地讲，该事件不仅表示这个设备停止运行，而且还表示要清除该设备上的L3配置，以及要报告该L3地址到L2地址的映射是无效的。

相反情况，如果系统中添加一个设备，邻居子系统对其并不关心。

L3 地址改变

如果管理员改变了接口的配置，以前通过该接口可到达的主机有可能通过它已无法到达。因此，改变接口的L3地址会触发调用neigh_ifdown函数。

协议关闭

如果作为模块安装的L3协议从内核中卸载了，那么所有相关的邻居项将不再有用，必须要删除。图27-11给出了做这种清理工作的两个函数：DECnet协议中的dn_neigh_cleanup和IPv6中的ndisc_cleanup。由于IPv4不是模块实现，所以它不会从内核中删除，也就没有相应的清理函数。

neigh_ifdown函数是很简单的。它会浏览所有的neighbour结构，找到与触发事件的设备相关的结构，然后使其不再可用。（这些不用的结构不会被立刻删除，因为在邻居子系统内可能仍然有对它们的引用）。下面列出了neigh_ifdown在每个受影响的neighbour结构上要执行的主要动作：

- 停止所有未决的定时器。
- 将相关邻居项的状态改为NUD_NOARP态，这样试图使用该邻居项的任何流量不再会触发solicitation请求。
- 使neigh->output指向neigh_blackhole，以便丢弃送到该邻居的封包而不是将其提交。参见“neigh->output使用的函数”一节中对该函数的描述。
- 调用skb_queue_purge，将所有在arp_queue队列中待处理的封包丢弃。在neigh_ifdown把缓存中的与有问题的设备相关的项清理掉之后，就调用pneigh_ifdown清理代理缓存和代理服务器的proxy_queue队列中的相关项。

使用 neigh_changeaddr（netdevice 通知链）更新

netdevice通知链会跟踪大量与网络运行相关的事件，见第四章。邻居协议为了从netdevice通知链得到通知，在其初始化函数（arp_init、ndisc_init等）中会向内核注册。

对邻居子系统来说最重要的事件是NETDEV_CHANGEADDR，在使用命令改变一个设备的L2地址时，由do_setlink 函数产生。类似的命令如下：

```
ip link set eth0 lladdr 01:02:03:04:05:06
```

当该命令调用neigh_changeaddr 函数改变地址时，这个该函数会扫描协议缓存中的所有项，并将与要改变地址的设备相关的项标记为停用(dead)。垃圾回收进程会负责处理这些停用项。

邻居协议和L3传输函数的交互

在第二十一章中，我们已经看到IPv4子系统中的封包传输过程的结尾要调用ip_finish_output2函数，该函数将封包传到L2。本节中，我们将看到这个函数如何与邻居子系统相互影响。在IPv6中有个名称与这个函数类似的函数，执行的任务也类似，只是它要调用ND协议而不是ARP协议。

ip_finish_output2的输入包含有封包数据(但是没有L2帧头)，以及其他一些信息，比如传输使用的设备，内核用于判断转发路由的路由表缓存项（dst），把这些输入到skb缓冲区中。如图27-1所示，dst 缓存项中包含一个指针，指向和下一跳地址（可以是路由器，也可能是目的主机）相关的neighbour项。在本章中我们感兴趣的是ip_finish_ output2的判断过程，如27-12所示。

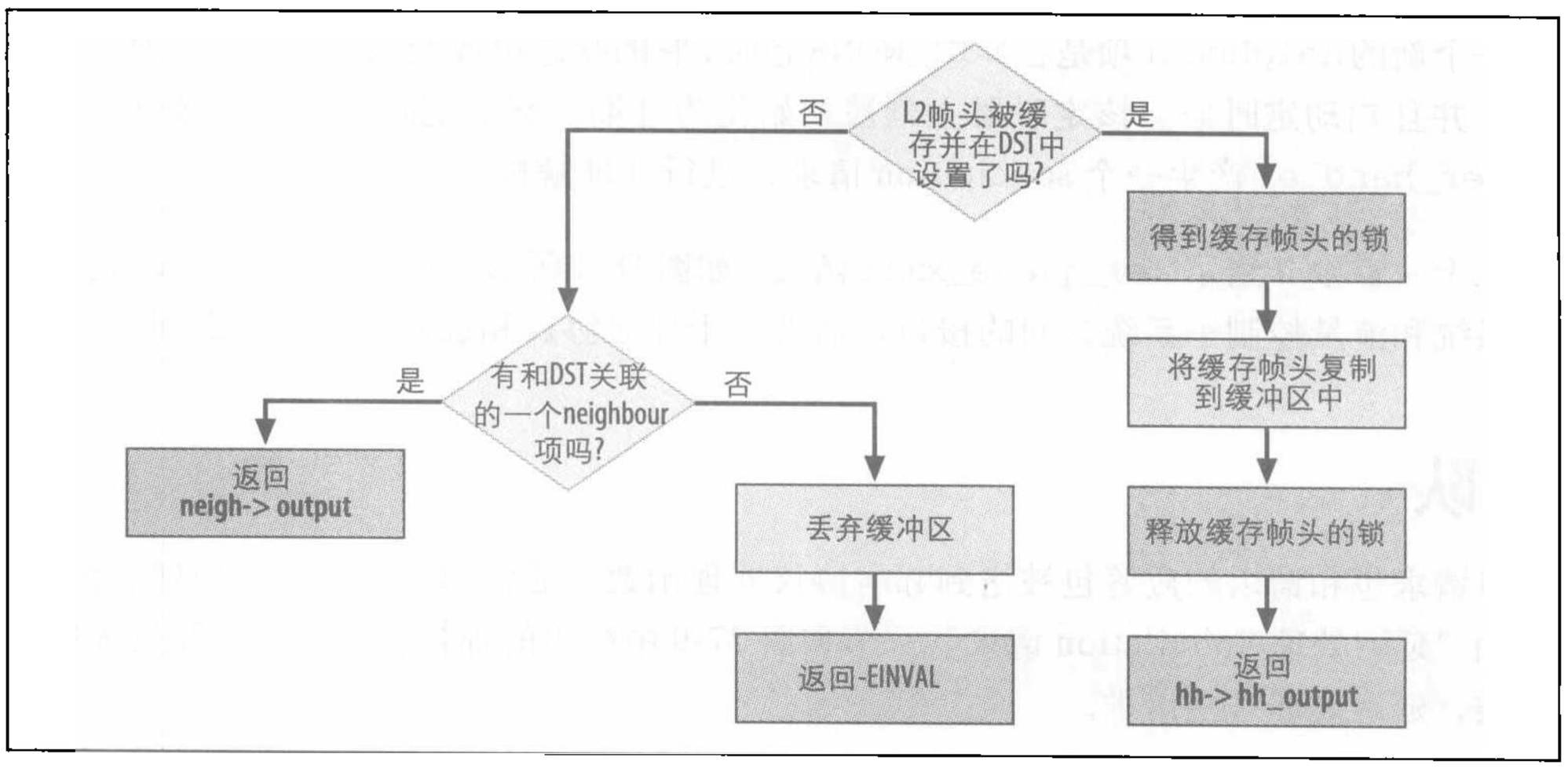


图 27-12：ip_finish_output2 函数：简洁形式

如果一个缓存的L2帧头是有效的(hh不是NULL)，就把它拷贝到skb缓冲区中。(skb->data指向用户数据的开头，这里也是放置L2帧头的位置。)最后，要调用hh_output。

如果缓存的L2帧头是无效的，ip_finish_output2就调用neigh->output方法。在

本章的前面已经解释过，与neigh->output相关的准确函数是根据neighbour项的状态决定的。如果L2地址准备好了，这个函数就可能是neigh_connected，这样就可以正确填充帧头，然后传输封包。否则，neigh->output就有可能被初始化为neigh_resolve_output，这个函数将封包放到arp_queue队列中，然后试着发送solicitation请求来解析L3地址，直到solicitation应答到达，于是封包就被传输出去。不论封包是立即发送还是先放到队列中，ip_finish_output2函数的返回值都一样，表示发送成功。从这之后，封包就不是由IP子系统来负责；当收到solicitation应答时，邻居子系统从arp_queue队列中取出封包，然后通过网络设备将其发送出去。

在“删除邻居”和“使用neigh_ifdown更新”两个小节中已经讲过，当调用ip_finish_output2时，如果必需的neighbour项（与传输dst->neighbour封包的路由表缓存元素相关联）不存在，就将封包丢弃。这个条件应该是不可能存在的，但是如果确实发生这个异常，邻居程序要能处理。

图27-13(a)和 图27-13(b)是图27-12所示函数的更详细描述，图中给出了根据哪个函数被指定给neigh->output和neighbour项的状态会出现的情况。如果你看一下源码，就能发现流程图中代表neigh_resolve_output的部分大部分是neigh_event_send函数的展开，但除了用虚线框中的部分。

当一个新的neighbour项是在NUD_NONE态时，它的状态就改变为NUD_INCOMPLETE态，并且启动定时器。该定时器立刻被初始化为过期。然后定时器处理函数neigh_timer_handler产生一个solicitation请求，进行地址解析。

第二十一章中介绍了dev_queue_xmit函数。如图18-1所示，dev_queue_xmit是邻居子系统和流量控制子系统之间的接口，后者位于邻居协议和设备驱动程序之间。

排队

入口请求包和请求的应答包被送到邻居协议处理函数，通常都是立即进行处理。但是，正如“延迟处理Solicitation请求”一节和图27-9中给出的那样，代理能够配置成排队等候，延迟处理这些请求。

如果L3发出的封包的目的地L2地址还没有被解析，那么它们就被邻居层临时放到队列中等待地址解析，参见“邻居协议和L3传输函数的交互”一节（与此相反，邻居协议自己产生的请求和应答立即就传输）。

下面的几节将详细介绍入口队列和出口队列。

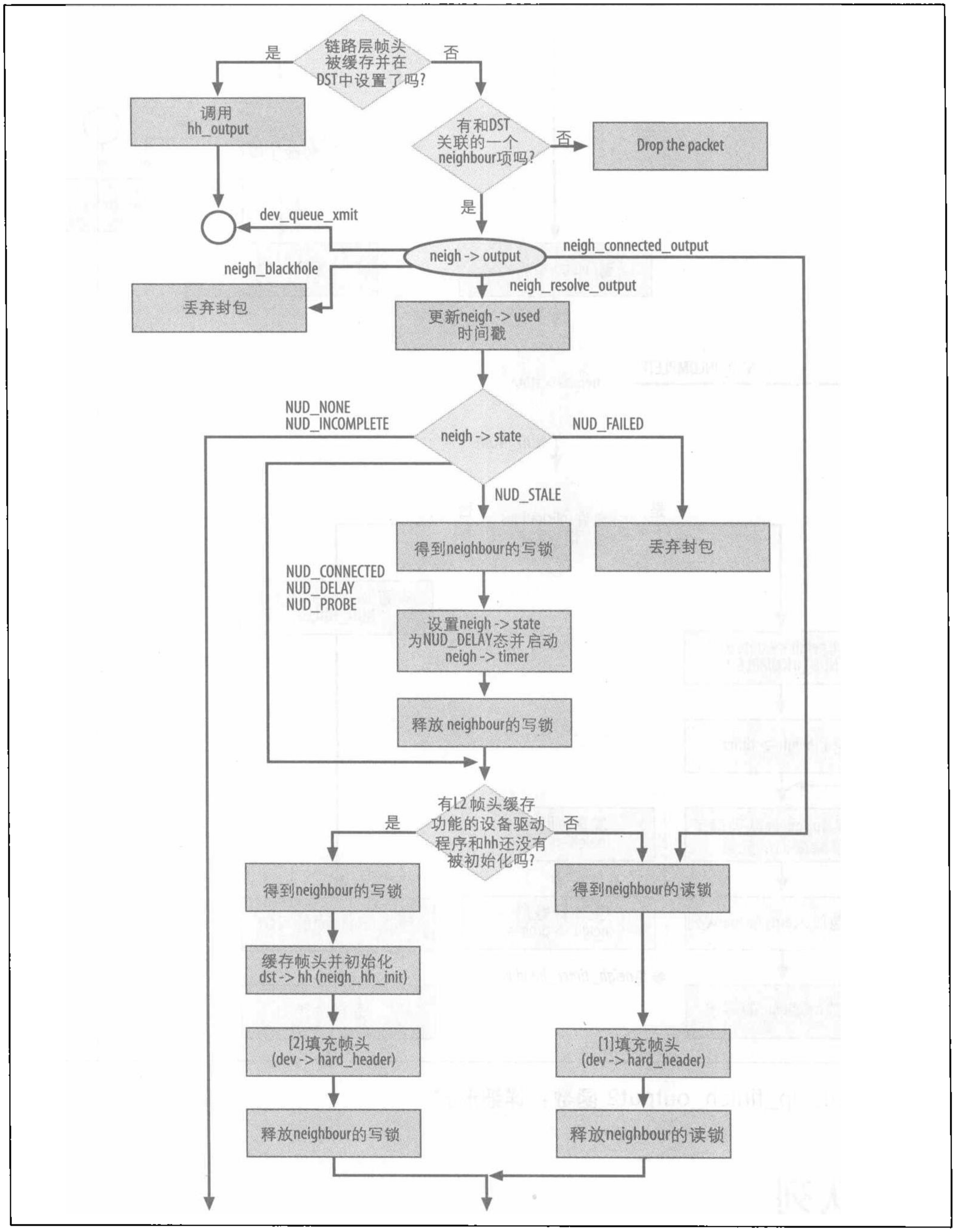


图 27-13(a)：ip_finish_output2 函数：详细形式

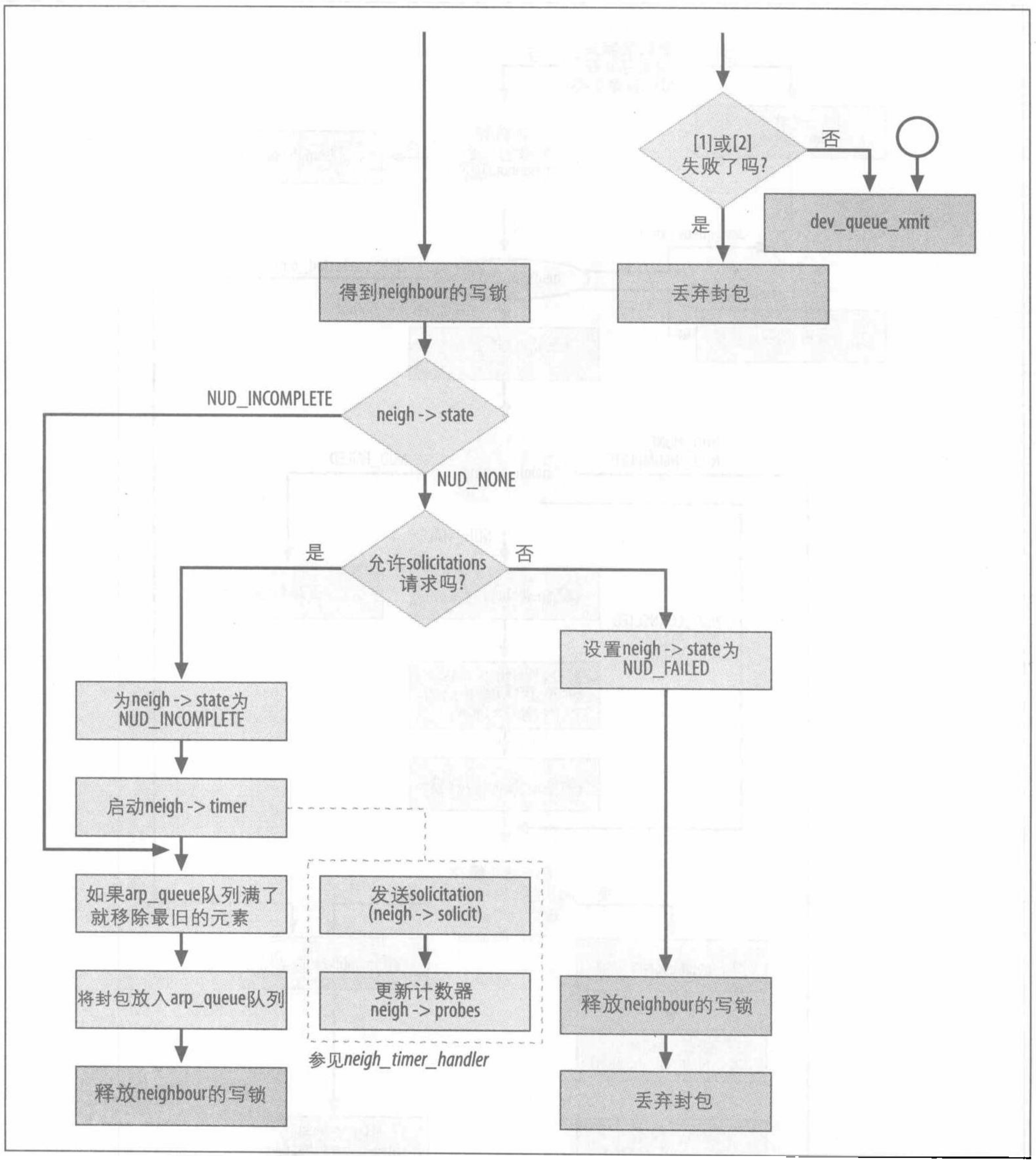


图 27-13(b)：ip_finish_output2 函数：详细形式

入口队列

当入口封包被放进队列时，所有邻居协议都要执行一些特定的任务。这些任务包括往缓存中增加封包，当收到 solicitation 应答时刷新（flush）arp_queue 及使用代理的

proxy_queue。也有一些个别邻居协议的特定任务。关于ARP协议执行的任务在第二十八章有详细介绍。

出口队列

当发送一个数据封包时，如果目的L3地址和L2地址之间的关联还没有建立，邻居协议将该封包临时插到arp_queue队列中（每个邻居都有一个名为arp_queue的队列，并不只是ARP协议有）。如果这个关联被及时建立，该封包就出队列并被发送出去；否则，就将其丢弃。图27-13给出了当L2地址还没有准备好时，IPv4封包是如何进入ARP的arp_queue队列。

每个neighbour项都有它自己的一个小的、私有的arp_queue队列；默认情况下，它包含三个元素，但这个队列也可以通过*/proc*配置为基于设备的队列（参考第二十九章）。若队列是私有的，而不是被所有邻居共享时，当邻居协议收到一个solicitation应答时，搜索队列的速度会更快，而且也可保证更高的正确性。当新元素要被增加到私有队列时，如果没有多余空间，就将新元素简单替换掉比较旧的元素（参考_ _neigh_event_send函数）。

图27-13给出了三个常用情况，为了简化问题，图中没有考虑代理：

(a) 空缓存

处理步骤如下：

1. L3提交要传送封包的请求，该封包要发往L3地址是192.168.1.1的目的地。
2. 查询缓存，但缓存未命中。
3. 临时将封包插入到队列中。
4. 发出solicitation请求。
5. 收到solicitation应答。
6. 缓存中增加相应的邻居项。
7. 队列中的等待的封包被发送出去。

(b) 地址解析待定

处理步骤如下：

1. L3提交要传送封包的请求，该封包要发往L3地址是192.168.1.1的目的地。
2. 查询缓存。

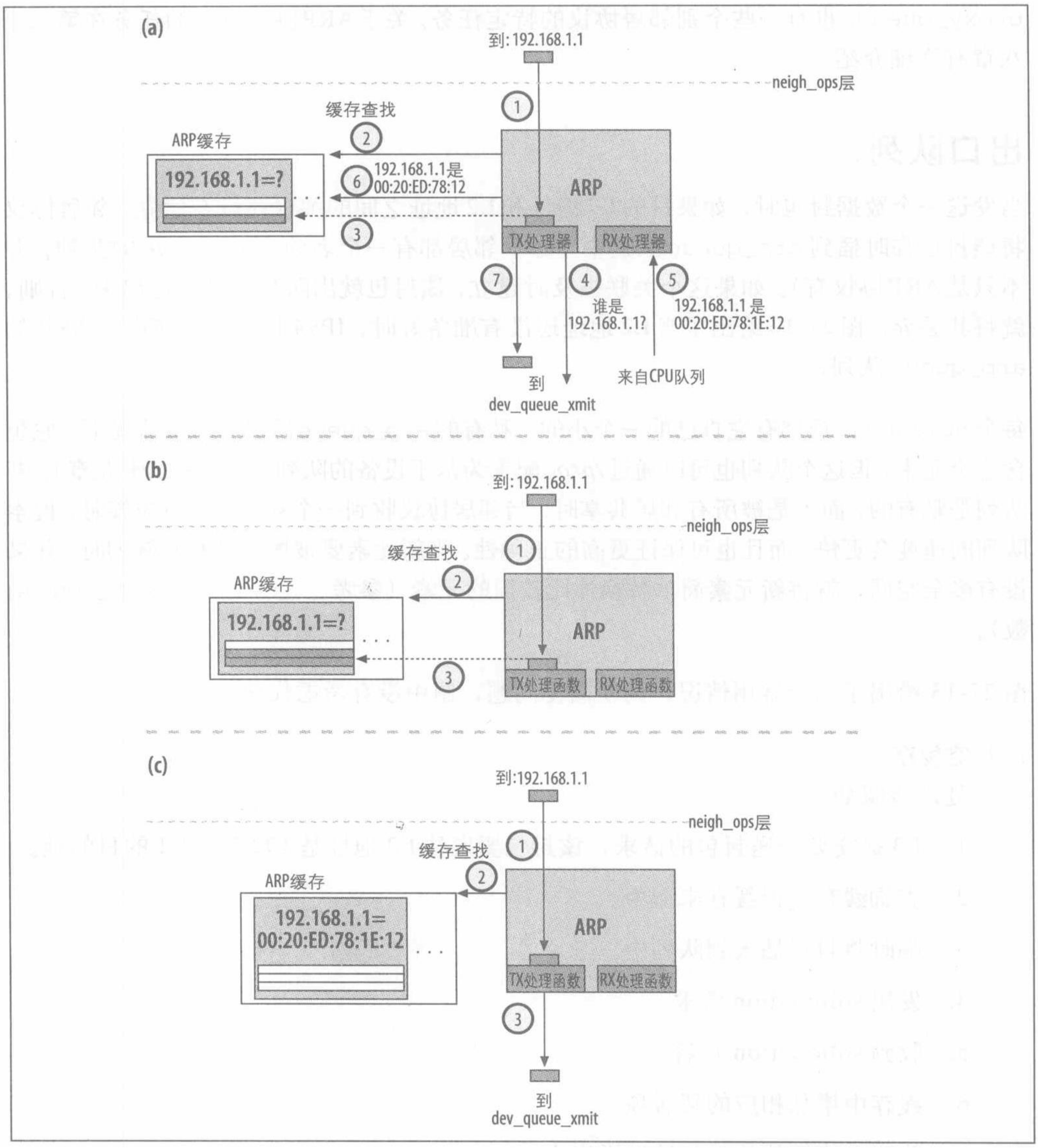


图 27-13：处理封包的三种常见情况

3. 地址不在缓存中，但是内核已开始解析地址的工作，因此封包被临时插入队列中，等待相关的应答。

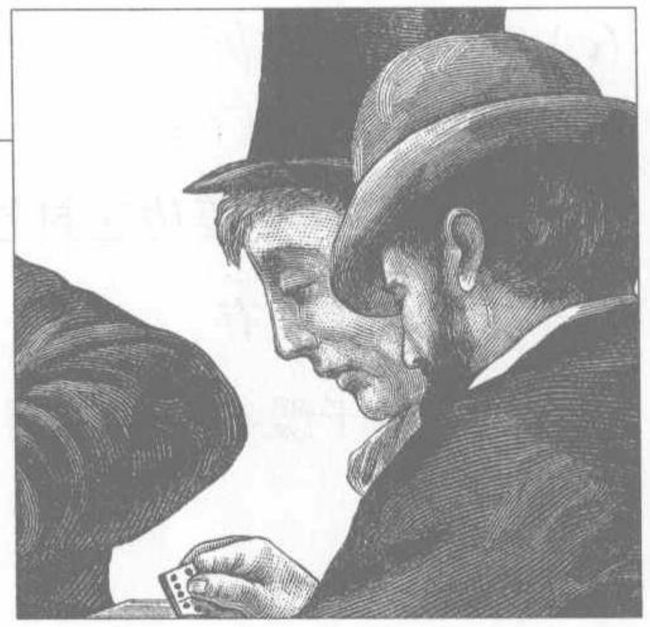
   只有另一个封包正在（a）中的第五步等待时，这种情况才会发生。

(c) 完成地址解析

处理步骤如下：

1. L3 提交要传送封包的请求，该封包要发往 L3 地址是 192.168.1.1 的目的地。
2. 查询缓存。
3. 由于缓存命中，封包就立刻发出去。

第二十八章



邻居子系统：地址解析协议（ARP）

第二十七章介绍了邻居基础结构对所有邻居协议提供的通用服务。本章中，我们将看到IPv4 使用的ARP协议如何适应邻居基础结构的模块化设计。读者若熟悉ARP协议，在前面几章对邻居子系统的描述中，尽管其中使用的术语多是来自 IPv6 的ND 协议，而不是ARP 协议，就可以看出 ARP 运行时的大致过程。

通用基础结构的出现，使得 ARP的设计和实现变得相对简单。本章主要讨论 ARP，我们将着重说明以下几点：

- 如何初始化 `neigh_table` 结构 `arp_tbl`，使其调整通用基础结构的行为来适应ARP。
- 如何初始化`neigh_parms`结构，使其调整通用基础结构的行为来适应ARP（例如，设置定时器超时阈值）。
- ARP封包（即`ARPOP_REQUEST/ARPOP_REPLY`）的接收如何与邻居子系统配合，以及 `solicit` 方法如何运作。
- 如何根据设备类型和L3地址的类型（单播、多播或广播）初始化`neigh_ops`结构。
- ARP 代理如何使用通用基础结构。
- 如何根据编译选项和对某些特性的明确配置来进一步调节 ARP 的行为。
- 为了处理特别繁重的负荷，内核如何将一些工作交给用户空间守护进程 *arpd*。
- ARP 和逆向 ARP（RARP）之间的关系。
- ARP 会将哪些事件通知其他的内核子系统，反之亦然。

本章还简要介绍一下IPv6的ND协议对ARP做了哪些改进。

ARP 封包格式

图28-1给出了ARP封包在Ethernet帧中的封装格式。

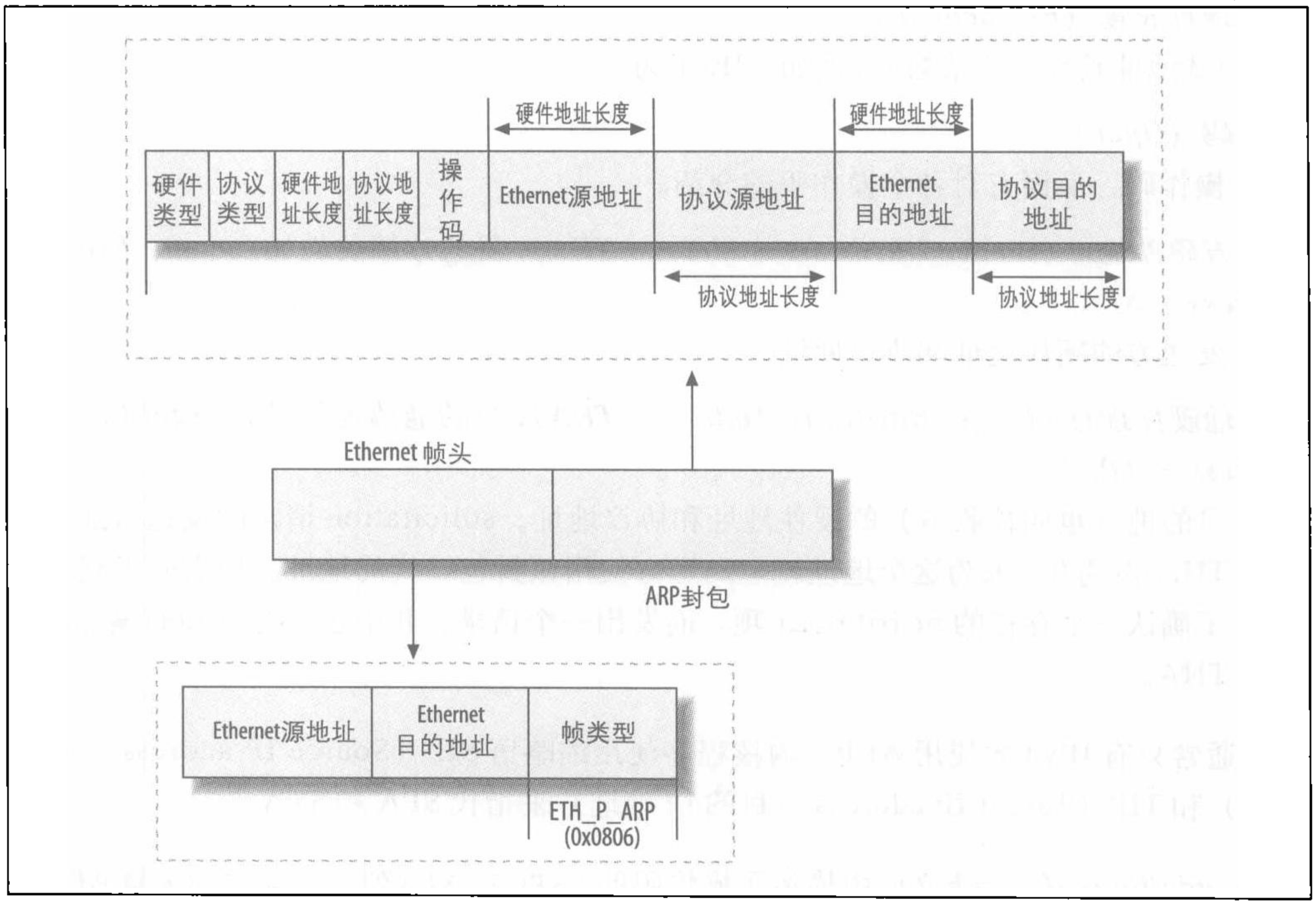


图28-1：ARP封包在Ethernet帧中的封装

下面介绍ARP封包中每个字段的含义，在Linux中用arphdr结构来表示ARP封包（注1）：

硬件类型（Hardware type）

硬件类型标识符（例如，Ethernet）。参见*include/linux/if_arp.h*文件中的ARPHDR_*XXX*值。

注1：arphdr结构中没有包含ARP帧中最后四个字段（表示地址）使用的占位符；它们可以通过读取Oper字段的末尾提取得到，HS字段和PS字段使之成为可能。

协议类型（Protocol type）

L3协议标识符（例如，IPv4）。参见*include/linux/if_ether.h*文件中的ETH_P_*XXX*值。

硬件地址长度（Hardware size）

L2地址长度（字节数）（例如，Ethernet为6）。

协议地址长度（Protocol size）

L3地址长度（字节数）（例如，IPv4为4）。

操作码（Oper）

操作码，后面有对各个操作码的介绍。

发送方硬件地址（Sender Hardware Address－SHA），发送方协议地址（Sender Protocol Address－SPA）

发送方的硬件地址和协议地址。

目的地硬件地址（Target Hardware Address－THA），目的地协议地址（Target Protocol Address－TPA）

目的地（也叫接收方）的硬件地址和协议地址。solicitation请求的发送方通常将THA设为0，因为这个地址就是发送方使用该封包寻找的地址。但有时发送方为了确认一个存在的neighbour项，而发出一个请求，其中包含有已知的该邻居的THA。

由于通常只有IPv4才使用ARP，内核程序使用缩略语SIP（Source IP address－源IP地址）和TIP（Target IP address－目的IP地址）来指代SPA和STA

在*include/linux/if_arp.h*文件中描述了操作码的ARPOP_XXX列表中提供的大量ARP消息类型。RARP使用的两个消息类型在后面“逆向地址解析协议（RARP）”一节中会介绍。有几个类型是用在InARP中，它是个相对较新的协议（RFC 2390），是对ARP的扩展，主要用在帧中继和ATM网络中，本书对其不作介绍。下面是ARP使用的两个基本消息类型：

ARPOP_REQUEST

在试图将一个L3地址解析为L2地址时，这个消息用于发出一个solicitation。对于一个新邻居项来说，主机会将该消息发到与设备硬件有关的广播地址。如果要确认一个存在的neighbour项，主机会将该消息直接发送到该邻居的L2地址。request等同于IPv6调用*neighbor solicitation*。

使用Solicitations也可能出于其他原因，例如“无端ARP”一节所讨论的情况。

ARPOP_REPLY

这个消息是ARPOP_REQUEST的应答。它通常直接发到发出请求的主机。但有时，它也被发到广播地址；在一台主机改变其配置后，该主机就会这么做，来更新其邻居的缓存。广播应答（reply）等同于IPv6调用*neighbor advertisement*。

ARP封包中的目的地址类型

L3地址的地址类型（address type）有单播、广播和多播三种。类型保存在neighbour结构中（neigh->type字段），见“初始化neighbour结构”一节，通过调用函数inet_addr_type可以确定地址类型。ARP对每个类型的处理如下：

单播（Unicast）

这是最常见的情况，并且其解析是用ARP的普通solicitation方法。

广播（Broadcast）

ARP简单地将L3广播地址直接映射为与具体设备有关的L2广播地址。

多播（Multicast）

ARP使用函数arp_mc_map从L3多播地址导出L2多播地址。ARP不需要生成solicitation请求，因为L2地址可以用一个根据硬件类型（Ethernet、Token Ring等）的公式推算得到。参见第二十六章的“特殊情况”一节。

ARP事务范例

图28-2给出了一个ARP事务的简单例子，图中一台主机请求与IP地址10.0.0.4相关的L2地址，并且拥有该地址的主机做出了应答。请求封包中目的MAC地址（要解析的地址）字段为0，表示这个地址应该由应答者（可能是目的IP地址的主机）填充。

如果你仔细观察图28-2，你能发现发送方硬件地址出现了两次：一次是在Ethernet帧头中，一次是在ARP封包中。这两个地址通常是一样的，但也有例外（参见第二十七章的“担任代理”一节）。

在“实例”一节中有更详细的ARP范例。

无端ARP

一般来说，发出一个ARPOP_REQUEST是由于发送方想和一个IP地址通信，需要找到其对应的L2地址。但有时候，发送方生成一个ARPOP_REQUEST是为了通知接收方一些信息，而不是请求信息。这种封包就称为无端ARP，它通常用在下面列出的情况中：

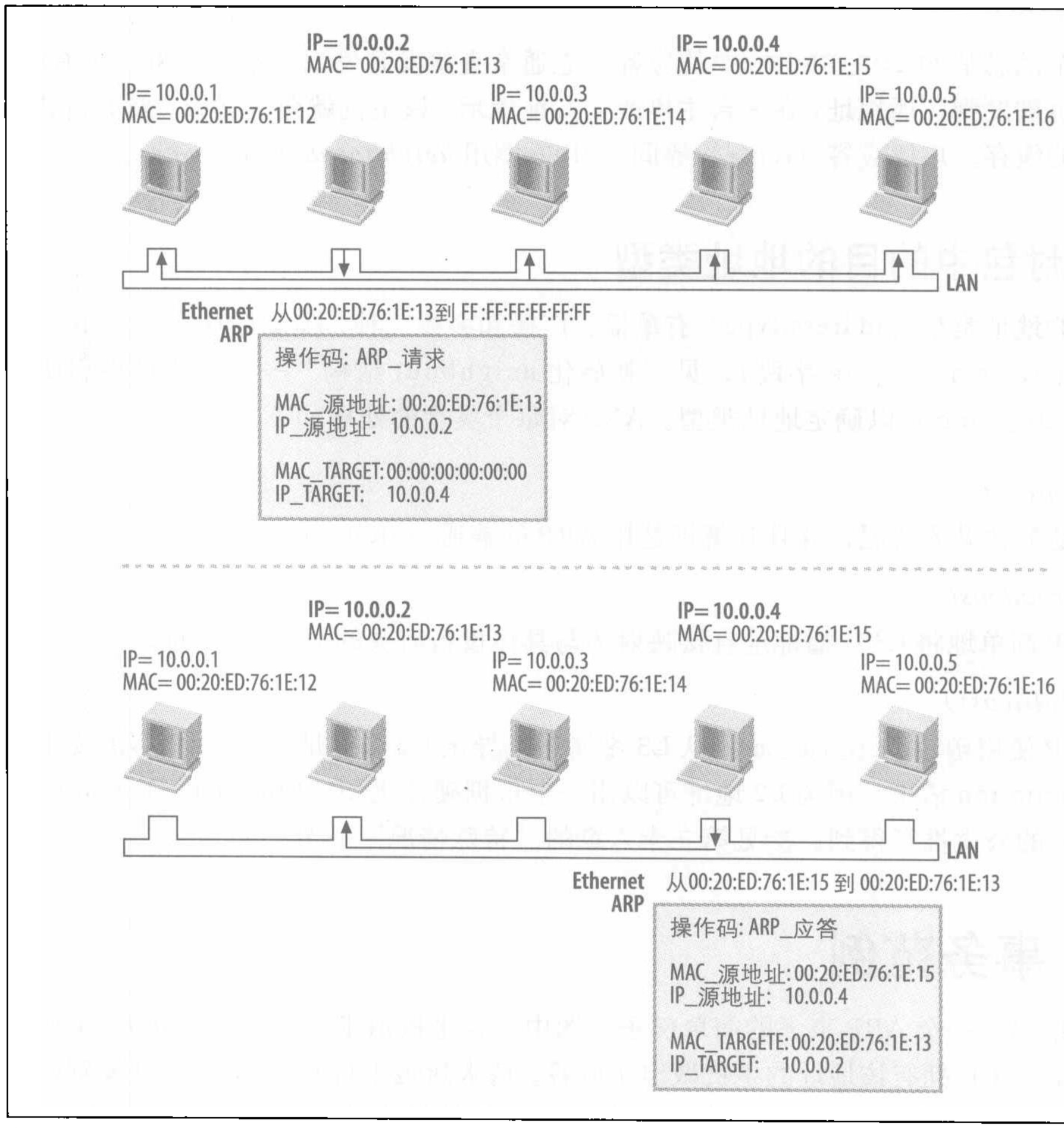


图 28-2：应用 ARP 的例子

- L2 地址发生变化。
- 重复地址探测。
- 虚拟 IP。

下面分别对每种情况进行介绍。

L2 地址发生变化

在第二十六章的“需要邻居协议的原因”一节中我们已经看到，L2 地址的变化（这会使

网络中其他节点的neighbour项失效）没有协议的帮助内核是无法检测到的。当L2地址变化后，如果等待旧的L3和L2间的关联过期，然后强制每个节点启动新的协议事务（因此要忍受一段短暂黑洞时期），因为提前触发关联的更新是有意义的。改变了地址的节点通过无端ARP来完成这类的更新。*net/irda/irlan/irlan_eth.c*文件中代码可以作为这个过程的一个例子。

重复地址检测

同一个局域网中不应该出现有两台主机有相同的L3地址，但这种情况总会发生，尤其在一个同时包括静态IP配置和动态IP配置（基于DHCP）的大型网络中最有可能。地址重复的最常见的原因是存在多个地址池重叠的DHCP服务器，以及错误的手工配置。

主机可以使用无端ARP来检测重复地址的存在。如果你发出一个目的地是你自己地址的ARP请求，那么只有当存在一台与你有相同IP配置的主机时，才会收到应答。如果不存在重复地址，就收不到任何应答。

让我们用图28-3中的拓扑结构看一个例子。当主机A启动后，它一将IP地址10.0.0.4配置在自己的*eth0*接口上，然后就发出一个请求，询问谁的地址是10.0.0.4（该主机自己的地址）。如果在同一子网内没有其他主机配置该地址，主机A就不会收到应答。但若是主机Bad_guy 配置了和主机A相同的IP地址10.0.0.4，那么它就对刚才的ARPOP_REQUEST做出应答，这样就告诉主机A存在重复的地址。

当然，在大型网络中，允许主机随机发出ARP封包对整个网络的性能是不利的。相反，正如“零地址的请求”一节所述，DHCP服务器在将一个地址授权给客户端前，通常会发出请求，检测这个地址是否会重复，相对来说，这是一种比较有弹性的解决方式。

当你在本地接口上配置了一个IP地址时，Linux内核不会生成任何无端ARP。但是，大部分Linux发行版都安装有*iputils*软件包，该软件中包含*arping*命令。*arping*可以产生ARP_REQUEST帧。当你使用*/sbin/ifup*命令（initscripts软件包的一部分）启动一个网络接口时，*它就会用arping*来检查重复的地址。

虚拟IP

无端ARP另一常用的地方是允许服务器池中的故障迁移。通常情况下，为了使一个站点有冗余度，该站点除了有一台活跃服务器外，还有一定数量的配置类似的主机处于备用模式。当活跃服务器由于某些原因宕机后，与*heartbeat*定时器相关一个机制（通过服务器池使用的协议实现）就会检测到这个故障，并启动新活跃服务器的选择。这个新服务

器会生成一个无端ARP封包来更新网络中的所有其他主机上的ARP缓存。因为新服务器继承了旧服务器的IP地址，发出的ARPOP_REQUEST不会被应答，但所有的接收者会更新它们的缓存。

注意，在这种方式中，IP层及其上层仍然可以继续通信，甚至不知道这种变化。当然由于heartbeat是以规定的间隔发出的，在旧服务器失效后到新服务器接管这段期间就存在一个小的时间窗口，这段时间内网络流量是不会被提交的。因此某些节点可能会发现这个失效，并将其邻居项也失效，直到新的ARPOP_REQUEST到达。

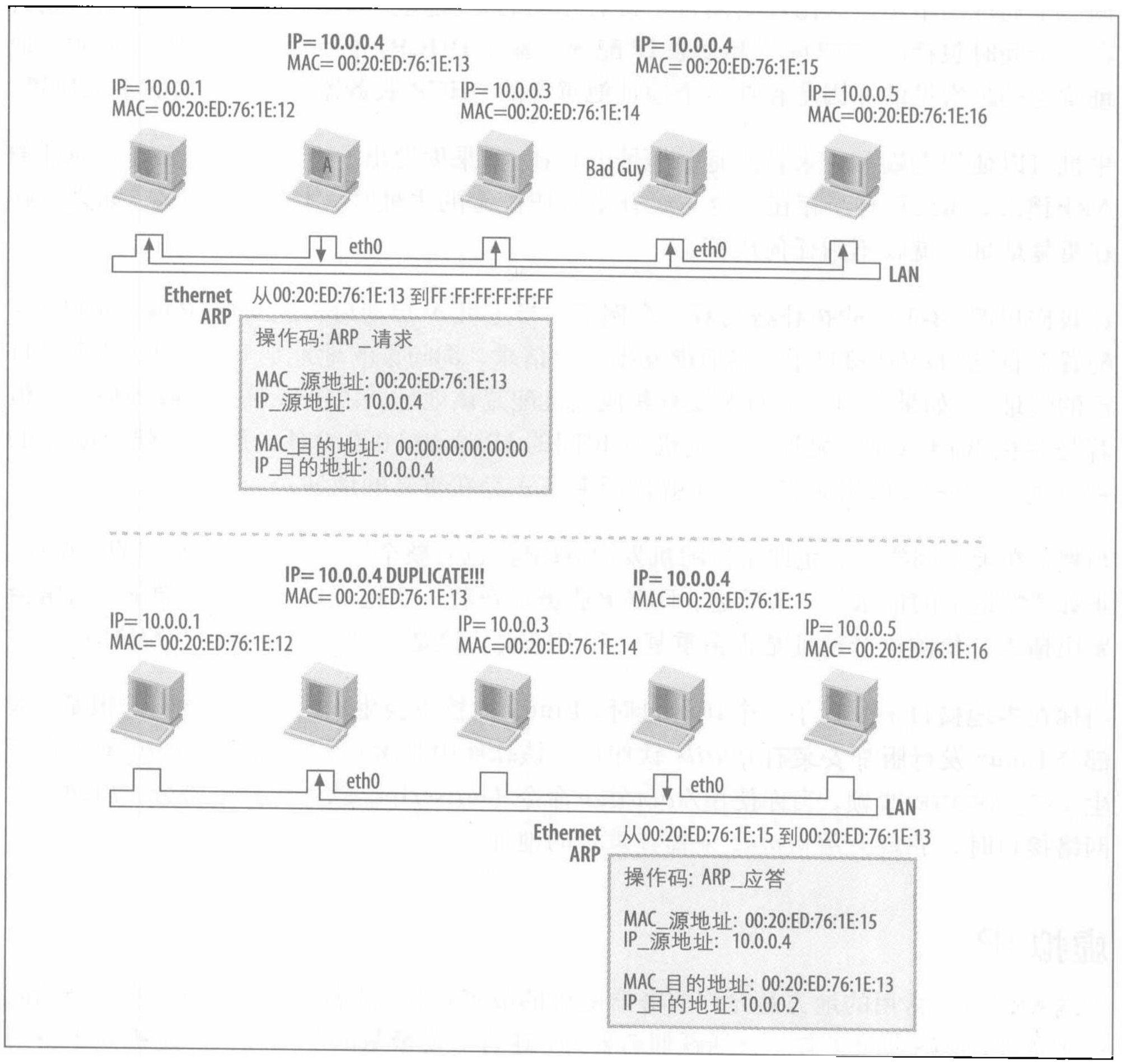


图 28-3：检测重复地址的例子

图28-4所示的例子（注2）中有两个路由器，一个是活跃角色，另一个是备份角色（a）。标为“活跃主机”的路由器的IP地址是10.0.0.1。LAN2中的主机使用这个路由器与LAN1中的主机进行通信，反之亦然。

一个故障迁移系统在运行着，以便当活跃路由器失效时，备份路由器就接管IP地址10.0.0.1，而成为活跃路由器（b）。当备份路由器成为新的活跃路由器后，就发出无端ARP请求，要求改变所有本地主机的邻居项（c），因为10.0.0.1现在与新活跃路由器的L2地址关联。接下来，来自LAN2的IP流量就会通过这个路由器。新活跃路由器也会向LAN1发出无端ARP请求，但图中没有画出。图中也没有给出实际中管理员应该配置的另一个细节：每个路由器在它的每个接口上都有一个备份IP地址，主要在当前角色不是活跃时，提供额外的连通性。

多个网络接口的应答

Linux有个不同寻常的设计：它认为一个IP地址属于一个主机，而不是属于一个接口，尽管管理员总是将IP地址设置在特定的接口上（注3）。于是一些管理员总是有如下的抱怨：

- 一台Linux主机会对目的地址是配置在该主机任一接口上的某一IP的任何ARP solicitation请求做出应答，而不管这个请求是在其哪个接口上收到的。为了使Linux的行为好像是IP地址是配置在接口上，管理员可以使用在后面的“/proc选项”一节描述的ARP_IGNORE特性。
- 主机可能会遇到*ARP flux*问题，在这种情况下，与L3地址关联的可能是错误的接口。在下面的内容中会对这个问题进行描述。

假设在同一个LAN中，有一台主机有两块NIC，并且另一台主机向其中一个地址发出了一个ARP请求。在两个接口上都会收到该请求，如图28-5所示，并且这两个接口都会做出应答。

注2：为了表述方便，将图中的MAC地址被截短了。例如00.....03表示00：00：00：00：00：03。用类似的简单MAC地址可以简化图形。

注3：使用在“调整ARP选项”一节介绍的一些选项，就可以使Linux表现成IP是属于相应接口的。若对这个设计特性有兴趣，包括它的优缺点，你可以参见*netdev*邮件列表中的“ARP responds on all devices”邮件（会很长），可以从网址*http://oss.sgi.com/archives/netdev*获得。

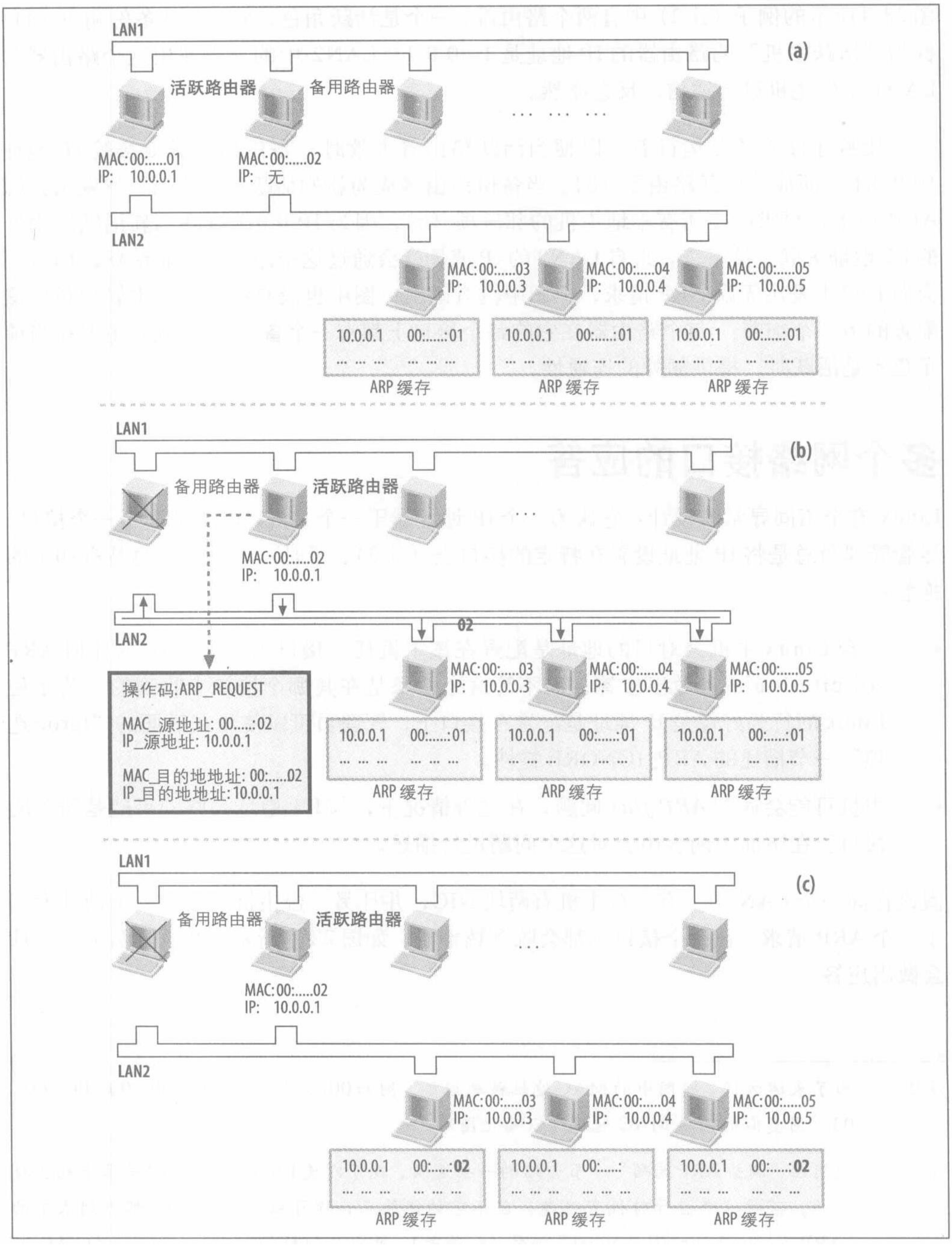


图 28-4：无端 ARP 的例子

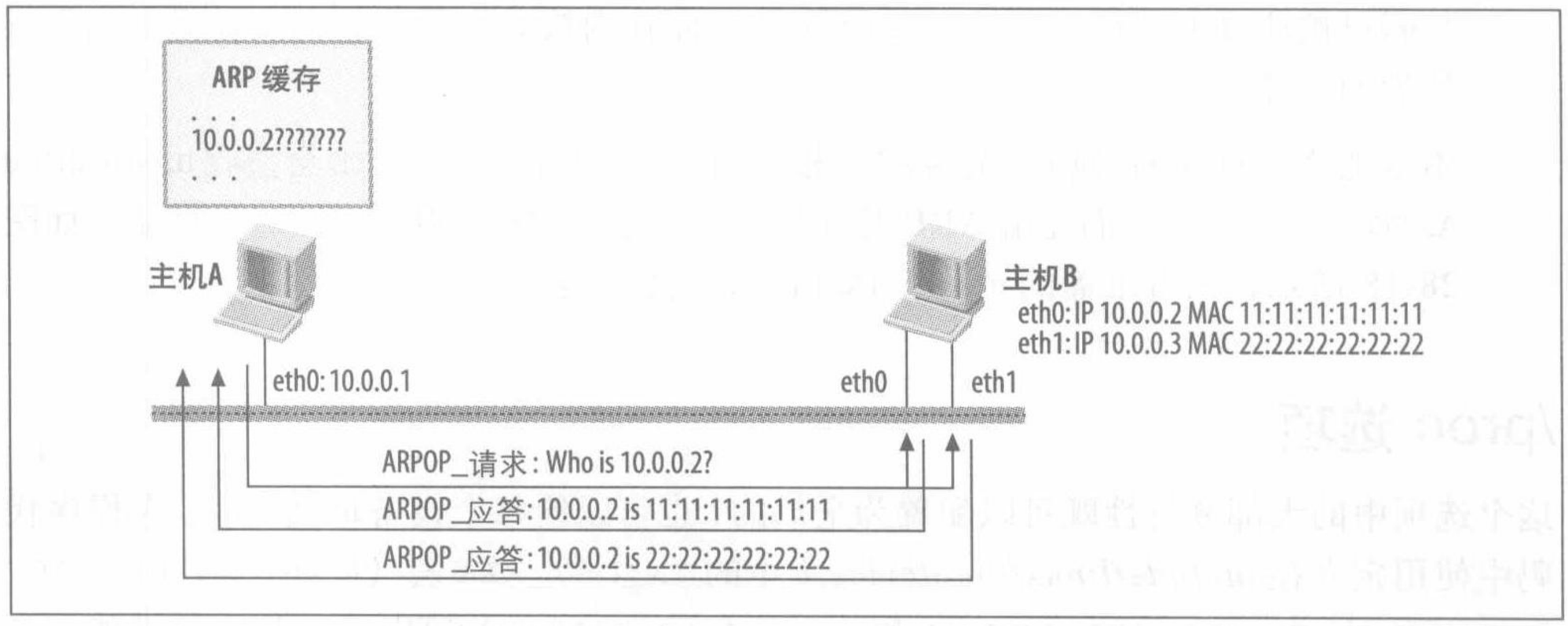


图 28-5：ARP flux 问题

因此，发出请求的主机会收到两个应答。一个来自有正确L2地址的NIC（*eth0*），而另一个是属于另一个NIC地址（*eth1*）。至于哪个地址会进入该主机的ARP缓存是不确定的，与该主机收到应答的先后顺序，以及处理多重应答的方式有关。

ARP flux 问题可以通过“调整 ARP 选项”一节描述的特性解决。

调整 ARP 选项

内核允许用户通过 */proc* 文件系统和编译时间选项来调整 ARP 的行为。在第二十九章的“使用 /proc 文件系统调整邻居性能”一节中，我们会看到如何配置这些特性的详细介绍，以及它们允许的设置和默认值。下面介绍其中主要的几个特性。

编译选项

在编译时，可以设置两个选项：

ARPD (CONFIG_ARPD)

这个选项允许用户空间的守护进程处理ARP包，这样做在超大型且流量比较大的网络中，可以提高传输性能。参见“ARPD”一节。

UNSOLICITED ARP (CONFIG_IP_ACCEPT_UNSOLICITED_ARP)

默认情况下，如果一台主机收到一个 ARPOP_REPLY，但它没有对应的未决的 ARPOP_REQUEST，那么它就会将该应答丢弃。然后，有时候收下该应答也可能会有用。对于这个特性，确定接收unsolicited应答，在Linux中已不再支持：相关的

代码已被注释（在arp_process函数中），而且内核的配置菜单也没有提供启用该特性的途径。

不要把这个特征的影响与无端ARP相混淆。ARP_UNSOLICITED会接收unsolicited ARPOP_REPLY包，而无端ARP是通过ARPOP_REQUEST引起“推送”更新。如图28-18所示，只有单播的unsolicited请求才会被接受。

/proc 选项

这个选项中的大部分特性既可以配置为全局的，也可以给每个设备单独设置。在程序代码中使用定义在 *include/linux/inetdevice.h* 中的 IN_DEV_*XXX*宏（例如，IN_DEV_ARP_ANNOUNCE、IN_DEV_ARP_IGNORE和IN_DEV_ARPFILTER）可以启动相应的功能。参考这些宏的定义，就能知道哪些特性是全局的，哪些是局部的（可以为设备单独设置的）。所有这些宏都已设备的IP配置块（net_device->ip_ptr）作为输入参数，IP配置块可以通过 in_dev_get 函数得到。

在向Linux虚拟服务（Linux Virtual Servers，LVS）发布地址时会用到一些选项。在LVS HOWTO文档的“LVS: the ARP problem”一节中，你可以找到关于这些选项的作用及如何配置的详细信息。你也能找到对以前的解决方法的介绍。

ARP_ANNOUNCE

当产生solicitation请求的主机有多个IP地址时，这个选项控制哪个地址应被放到solicitation请求的ARP头中。表28-1列出了该选项允许的级别，以及如何从本地系统的多个IP中选择一个。“Solicitations”一节中会给出ARP协议如何使用这个选项。ARP_ANNOUNCE选项在 arp_solicit 函数中处理。

表28-1：ARP_ANNOUNCE取值

级别值	含义
0（默认值）	任何本地IP地址都可以。
1	如果可能，选择和目的地址位于同一个子网内的地址。否则，使用级别2的结果。
2	优先使用主地址。

ARP_IGNORE

这个选项控制判断是否处理ARPOP_REQUEST封包的条件。

正常情况下，主机会处理所有的ARP请求。在“多个网络接口的应答”一节中曾经提到，Linux中IP地址是属于主机的，而不是属于接口。由于这个原因，只要ARPOP_REQUEST包中目的IP地址是配置在主机任一接口，包括回环接口（loopback）（注4），该主机就会处理这个请求。在某些情况下，例如，使用LVS时，这个特性就成了一个问题。但通过适当地配置ARP_IGNORE选项，管理员能够解决此问题。参见LVS HOWTO文档来获得对此问题及其解决方法的详细介绍。

图28-6给出了虚拟服务器配置的例子。该服务器的地址是全世界都知道的，图中称为VIP,该地址被配置在虚拟服务器的一个NIC上,但在其他两个真实服务器上是作为回环地址。所有对VIP地址的请求的应答只应该来自虚拟服务器。但当虚拟服务器收到对它提供的服务的请求时,它会使用定义好的选择算法,将其转发到两个真实服务器之一上。接收主机会接受这个请求服务的封包，因为它在本地配置上有VIP地址。两个真实服务器都在它们的*eth0*接口上设置了ARP_IGNORE选项，这样它们就不对VIP地址的ARPOP_REQUEST响应。

ARP_IGNORE选项在arp_process函数中处理。表28-2列出了该选项可能的取值。

表28-2：ARP_IGNORE取值

值	含义
0（默认）	对任何本地地址的ARP请求都应答。
1	如果目的IP配置在收到ARP请求的接口上，才应答。
2	和值为1的情况类似，但是源IP（发送方地址）必须和目的IP属于同一个子网。
3	如果目的IP的scope（译注1）不是本地主机（例如，那个地址不用于和其他主机通信），才应答。
4~7	保留。
8	不应答。
>8	未知的值，接受请求。

ARP_FILTER

当一台主机有多个NIC连接到同一LAN，且配置在同一个IP子网上时，这个选项可以控制一个接口是否对入口ARPOP_REQUEST做出应答。上述情况下，每个NIC会收到一个ARPOP_REQUEST封包的拷贝，通过这个选项你可以只选择一个接口来应答（确定性选择，不是随机选择）。这个选项在使用了IP源路由选项的网络中很有用。

注4： 地址127.*x.x.x*是个例外，对这些地址的ARP请求永远不会处理。

译注1： 关于范围的概念和详细讨论，参考第三十章的“范围”一节。

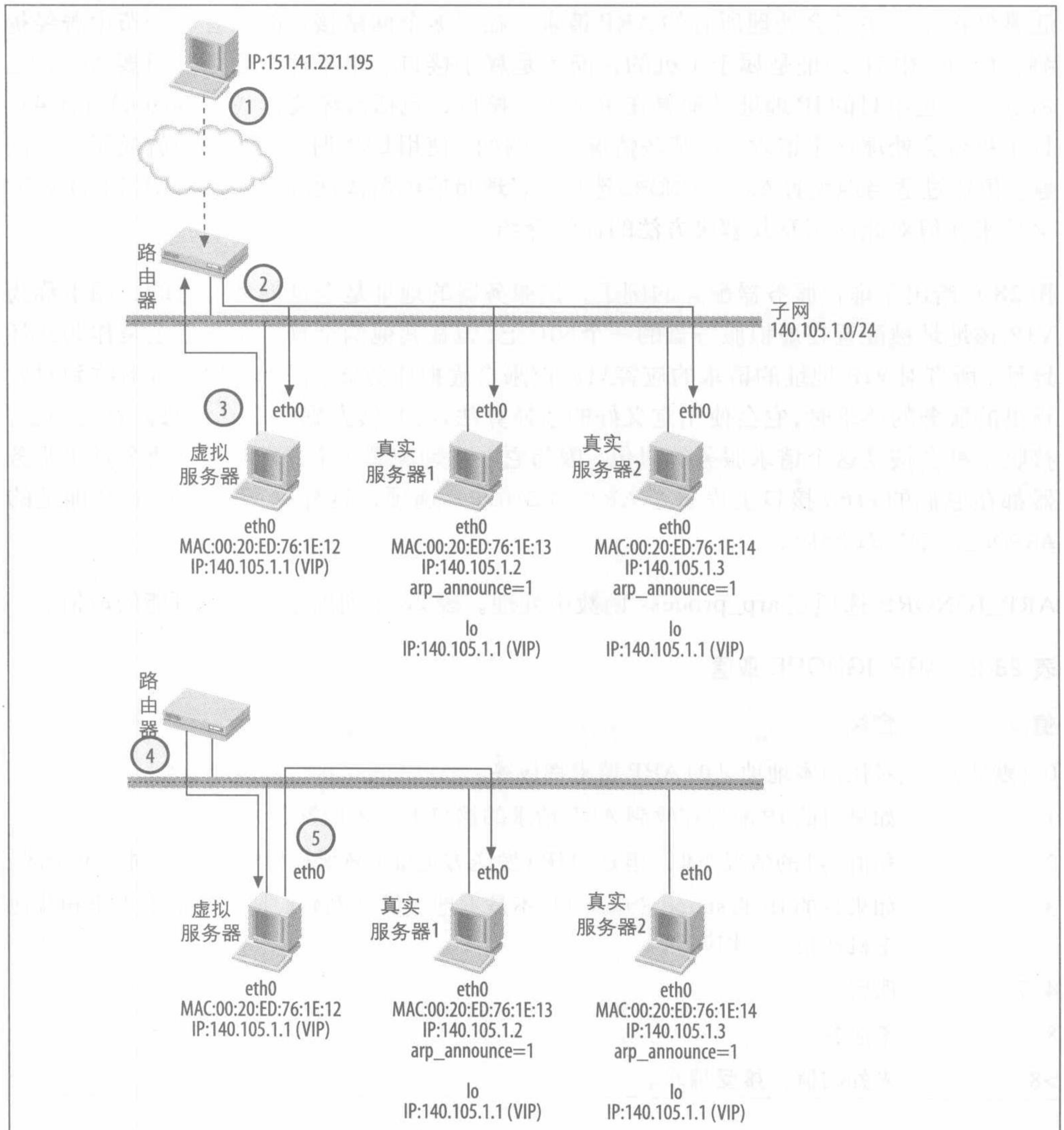


图 28-6：使用 ARP_IGNORE 选项的例子

下面以图28-7所示的网络看一个例子。当主机A试图解析IP地址10.0.0.1时，主机B的两个接口都会收到ARPOP_REQUEST。由于有两个请求，主机B需要查询路由表，然后只应答其中的一个请求，也就是主机B用于访问发送方IP地址（10.0.0.3）时用的接口上收到的请求。主机B的路由表显示出地址10.0.0.3通过*eth0*和*eth1*都是可到达的。但是，在第七部分就会看到，当到一个IP地址有多条路由可用时，路由表查找程序总是返回同一条（注5）（也就是说，匹配的第一条）。

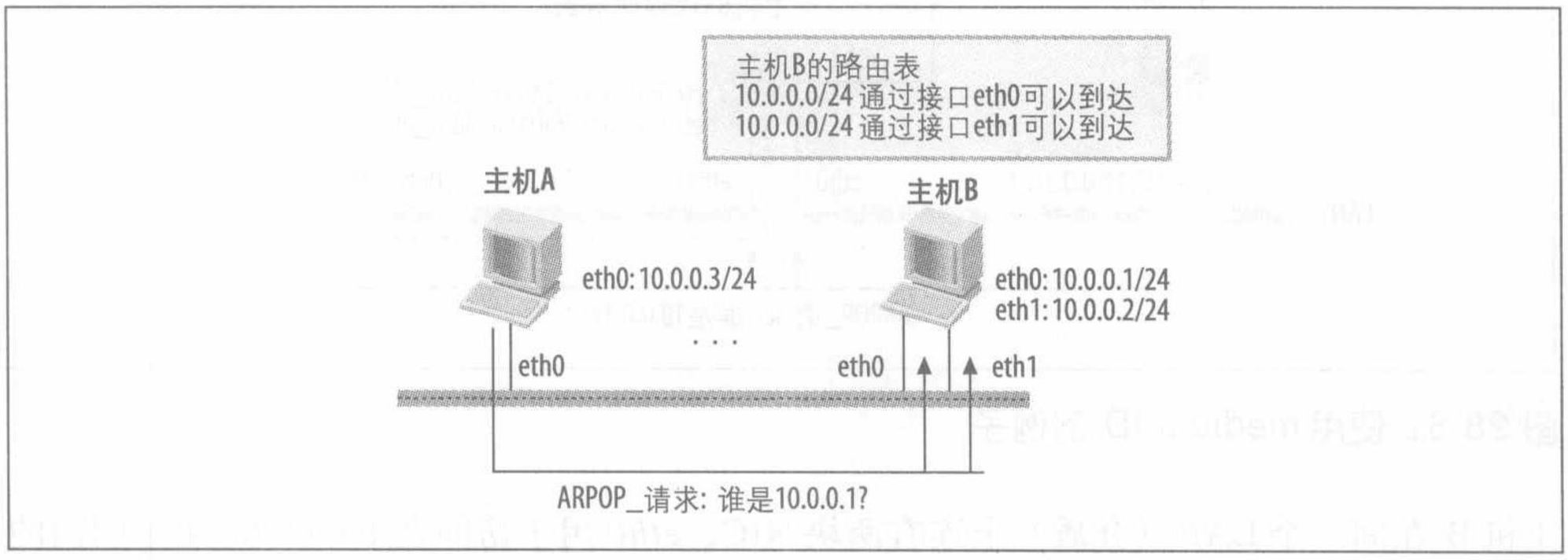


图28-7：使用ARP_FILTER选项的例子

当设置该选项后，只有在内核知道如何到达发送方的IP地址，并且只有到达发送方IP地址的设备是接收入口ARPOP_REQUEST封包的设备时，内核才会处理这个请求。

注意，ARP过滤和Netfilter过滤是没有关系的。这两种过滤方式可以单独配置和使用。

ARP_FILTER选项和前面两个选项不同，它只能开启或关闭；没有任何中间状态。它由arp_process函数处理。

Medium ID

这个特性能用于处理某些很少见的情况，IP子网横跨不同的LAN，且该子网中提供ARP代理的主机有多块NIC。术语*介质*指有唯一的广播地址的网络。如果使用集线器或交换机将这样两个网络连在一起，就会出现一种情况，这台ARP代理主机就成了一个不适当的代理，它替另一个LAN中的主机响应请求，但那台主机自己能处理该请求。

在第二十六章中，我们已经看到，若收到ARPOP_REQUEST请求的设备，但通过该设备

注5：　除非内核支持多路径缓存。该特征在第三十三章会介绍。

能到达这个请求解析的IP地址时，ARP代理服务器就不会应答该请求。但是，主机若有多块NIC 连到同一个LAN中，上述条件就无法满足。下面看图28-8所示的例子（注6）。

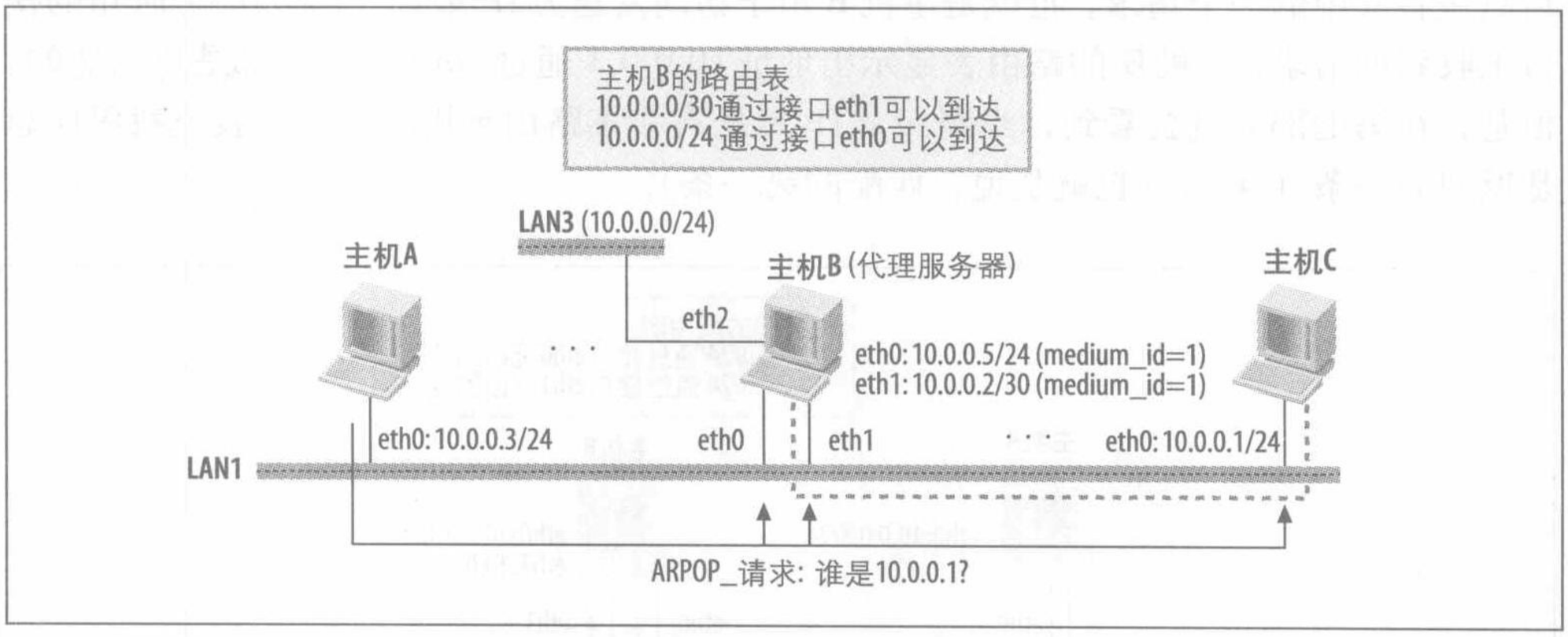


图 28-8：使用 medium ID 的例子

主机 B 在同一个 LAN（介质）上连有两块 NIC。*eth0* 用于访问在 10.0.0.0/24 子网内的所有 IP 地址，*eth1* 只用于和主机 C 通信，因为它们使用 /30 子网掩码。对于 LAN1 和 LAN3，主机 B 都是作为代理。

假设主机 A 现在需要向主机 C 传输一些东西，但是没有其L2 地址。主机 A 就会发出一个 ARPOP_REQUEST 广播包，主机 B 的 *eth0* 和 *eth1* 接口，以及主机 C 都会收到该包。主机 B 不应该对这个 ARPOP_REQUEST 作出应答，因为主机 C 自己可以应答。

假设主机是一个 ARP 代理服务器，并且在其所有接口上都启动了代理。让我们看看当该主机在其两个接口上收到 ARPOP_REQUEST 包时如何进行处理：

eth0 接口上收到的请求

根据路由表，请求解析的地址 10.0.0.1 通过另一个接口（*eth1*）可到达。因此，主机 B 会处理该请求。注意，主机 B 中有两条路由与目的地址 10.0.0.1 相匹配，但子网掩码是 /30 的那条路由更具体些，因此就选中这条。

eth1 接口上收到的请求

在这种情况下，接收接口和用于到达 10.0.0.1 接口是一致的，因此主机 B 不处理这个请求。

注 6：这个例子是在 Julian Anastasov 和 Alexey Kuznetsov 提供的范例的基础上做了扩充。上述范例在文档 *http://www.ssi.bg/~ja/medium_id.txt* 中，其中也描述了使用这个选项的通常情况。

正如你所看到的，需要一种方式来告诉ARP代理服务器，它的两个接口都在同一个广播域内，因此，两个ARPOP_REQUEST包都不应该被处理。通过给连在同一个LAN上的接口指定一个称为*medium ID*的ID就可以解决这个问题。在上述情况中，主机B上的*eth0*和*eth1*都设为同一个medium ID。对于入口solicitation请求，只有当请求解析的地址通过一个设备可以到达，但这个设备的Medium ID和入口设备的Medium ID不同时，主机才会应答该请求。Medium ID是一个正数；表28-3中列出了其取值所表示的含义。

表28-3：medium ID的取值

值	含义
-1	ARP代理已关闭。
0（默认值）	Medium ID特性已关闭。
>0	合法的medium ID。

在图28-9所示的拓扑中不需要配置medium ID，因为主机B的两个接口连在两个不同的LAN中。关于medium ID的使用方法和用途的详细情况，参见文档*http://www.ssi.bg/~ja/medium_id.txt*。

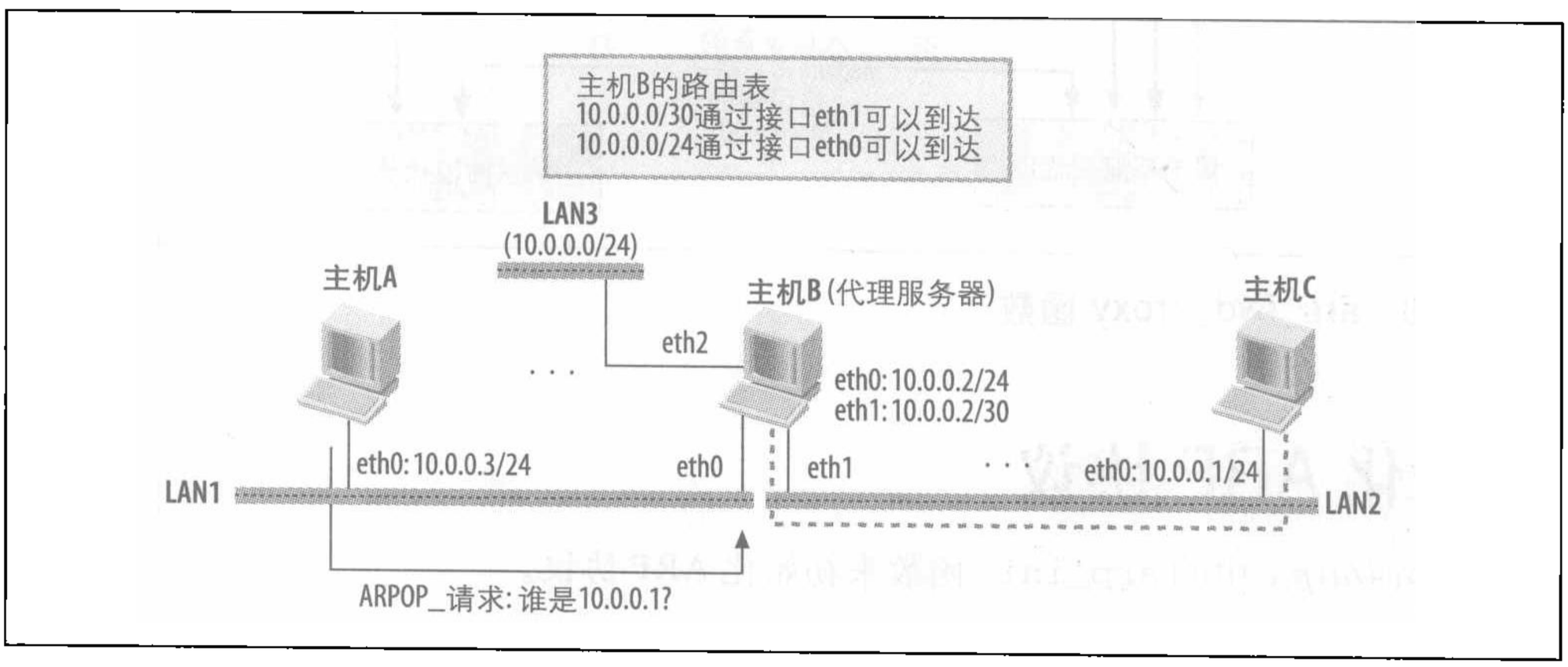


图28-9：重新设计的图28-8中的拓扑结构，不需要配置medium ID

图28-10给出了arp_fwd_proxy函数对上述过程的逻辑实现。arp_process会调用这个函数，根据ARP代理和Medium ID的配置情况，检查一个ARP请求是否可以被代理。

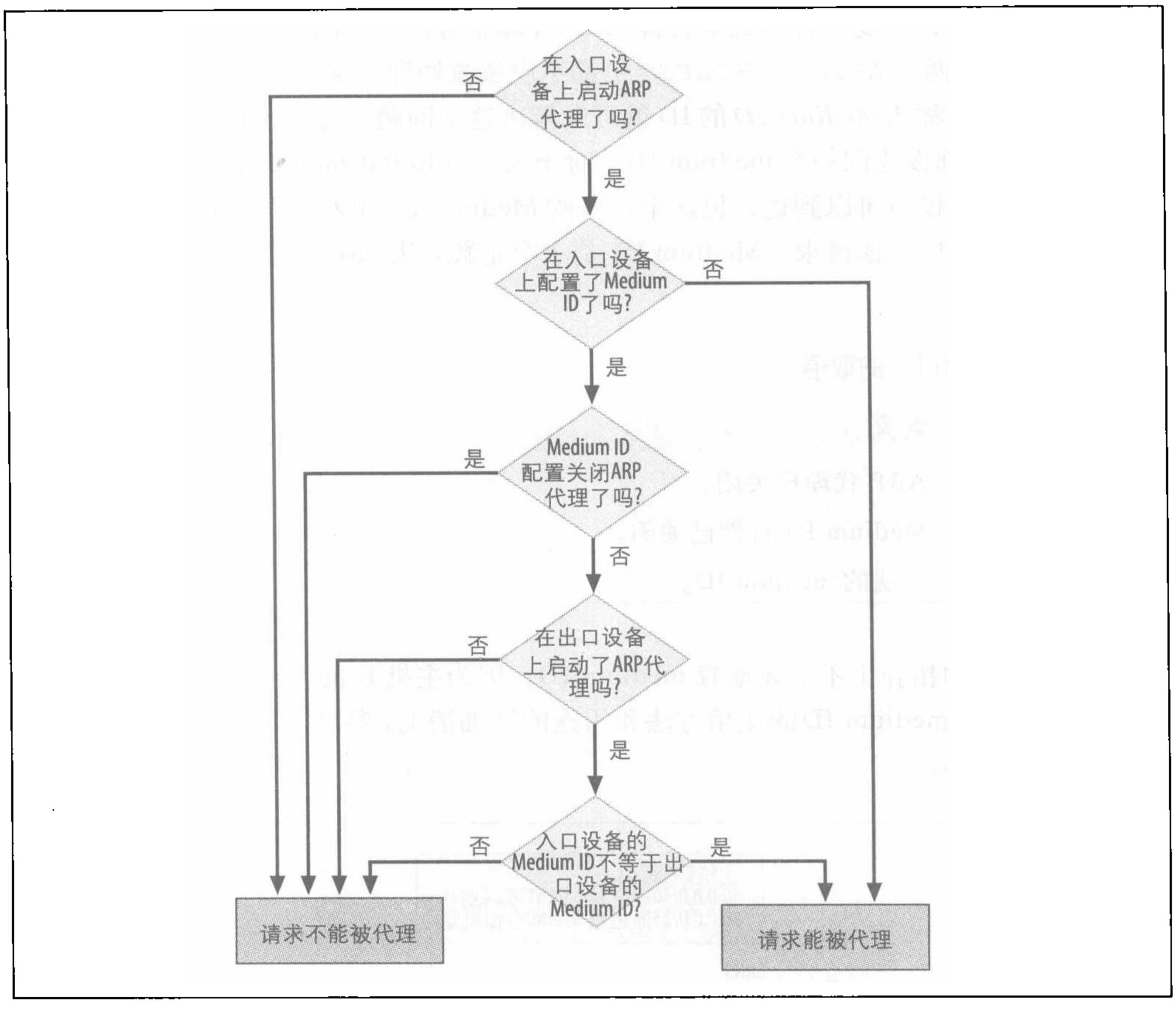


图 28-10：arp_fwd_proxy 函数

初始化 ARP 协议

用 *net/ipv4/arp.c* 中的 arp_init 函数来初始化 ARP 协议。

第二十七章的“协议初始化和清理”一节中给出了一个常用协议初始化函数的框架结构。本章中，我们具体分析 ARP 协议是被如何初始化的。

初始化函数首先是注册一个虚函数表和 ARP 协议使用的其他常用参数；这个工作由 neigh_table_init 函数完成。下面一节会介绍这个虚函数表（arp_tbl）的内容。

在第十三章我们曾看到dev_add_pack函数是如何安装一个协议处理函数的。你若还记

得该函数的用法，那么从下面的 arp_packet_type定义中就能明白 arp_rcv函数（也是在 *net/ipv4/arp.c* 文件中定义）是如何处理 ARP 封包的。

```
static struct packet_type arp_packet_type = {
    .type:    _ _constant_htons(ETH_P_ARP),
    .func:    arp_rcv,
};
```

arp_proc_init 函数会建立 */proc/net/arp* 文件，读取该文件就可以看到 ARP 缓存的内容（包括 ARP 代理的地址）。

若内核支持 sysctl，就能依靠函数 neigh_sysctl_register 创建一个目录 */proc/sys/net/ipv4/neigh*，用于输出 neigh_parms 结构的默认调节参数。注意，neigh_sysctl_register 函数的第一个输入参数为 NULL，在第二十九章的“建立目录”一节我们会看到，NULL 表示调用函数（arp_init）想注册为默认目录。

register_netdevice_notifier向内核注册一个回调函数，用于接收设备状态和配置变化的通知。详细内容请参见“外部事件”一节。

arp_tbl 表

这个表是个基本的数据结构，其中含有 ARP 协议涉及的关键变量。这个结构的类型是 neigh_table。在第二十七章的“主要的数据结构”一节有对该结构的详细讲述。ARP 按照如下流程初始化这个表：

```
struct neigh_table arp_tbl = {
    .family:        AF_INET,
    .entry_size:    sizeof(struct neighbour) + 4,
    .key_len:       4,
    .hash:          arp_hash,
    .constructor:   arp_constructor,
    .proxy_redo:    parp_redo,
    .id:            "arp_cache",
    .parms: {
        .tbl:               &arp_tbl,
        .base_reachable_time: 30 * HZ,
        .retrans_time:        1 * HZ,
        .gc_staletime:        60 * HZ,
        .reachable_time:      30 * HZ,
        .delay_probe_time:    5 * HZ,
        .queue_len:           3,
        .ucast_probes:        3,
        .mcast_probes:        3,
        .anycast_delay:       1 * HZ,
        .proxy_delay:         (8 * HZ) / 10,
        .proxy_qlen:          64,
        .locktime:            1 * HZ,
```

```
        },
        .gc_interval:    30 * HZ,
        .gc_thresh1:     128,
        .gc_thresh2:     512,
        .gc_thresh3:     1024,
    };
```

以base_reachable_time字段（见第二十九章的“neigh_parms 结构”一节）为例说明一下上述各字段在ARP中的重要性。该字段的值表示ARP只有在最后一个可到达性证据是在最后30 秒内收到时， 才认为一个邻居项是NUD_REACHABLE态。同样，retrans_time字段（见第二十九章的“neigh_parms Structure”一节）表示如果没有收到一个 solicitation 请求的应答，就在1秒后在发送一个新的请求。

下面几节中，我们将研究hash、constructor和proxy_redo方法。我们也会看到ARP协议是如何处理入口 ARP 封包的。

初始化 neighbour 结构

在前面的章节中已经知道，neighbour结构中保存着执行邻居协议任务（对一个L3到L2的地址映射，将L3地址翻译为L2地址）的全部信息。每个协议在它的neigh_table->constructor虚函数中指定一个函数来创建neighbour结构。在前面的arp_tbl结构的定义中已经看到，ARP 的初始化函数是arp_constructor。

基本的初始化顺序

图 28-11 给出了创建一个邻居项的基本步骤。

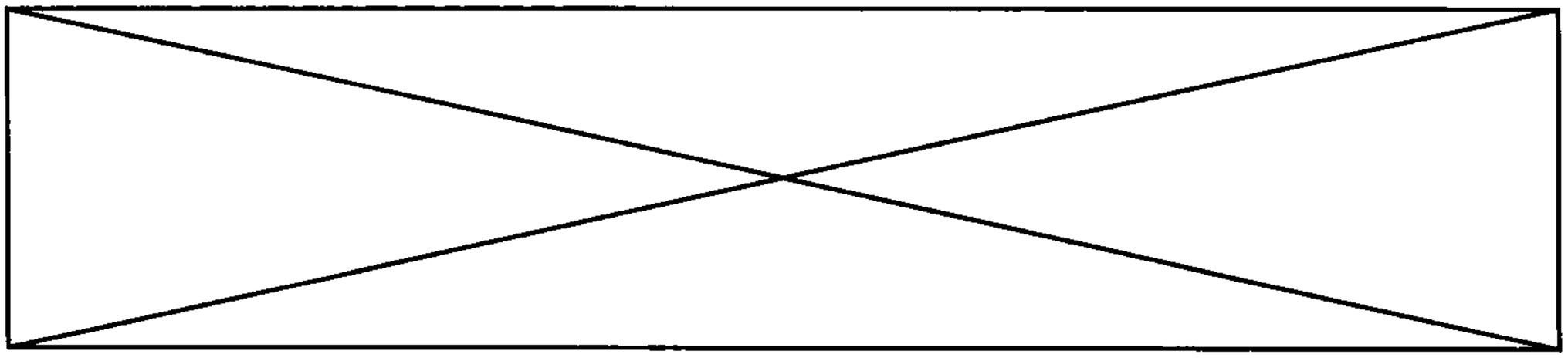


图 28-11：新 neighbour 结构的初始化顺序

在第二十七章的“创建一个邻居项”一节我们曾看到，创建邻居的原因不同，并且邻居的初态也可以不同。所以，neighbour结构中各个字段设置的默认值能被调用者设置的值覆盖。例如，当由于向一个邻居的传输请求而创建对应的neighbour结构时，neigh->nud_state的值就设为NUD_NONE。但是如果用命令行命令创建邻居项，该值也可能

是NUD_PERMANENT或NUD_STALE。广播和多播的IP地址不需要ARP来将其转换为相应的L2广播或多播地址，这些情况下，nud_state的值就是NUD_NOARP。

下面这些字段的初始化尤其重要：

nud_state

neighbour结构的初始态，根据L3地址的类型和创建该邻居项的原因取值。

output

根据指定给nud_state的值来初始化output。

ha

这个字段表示L2地址，也就是ARP协议要发现的东西。第二十六章“特殊情况”一节中介绍的地址是不需要ARP的，其L2地址可以直接从其L3地址推导出来。

ops

在第二十七章中讲过，这是个虚函数的集合，它决定了IP子系统要调用的动作。图28-12总结了ARP（确切地说，是arp_constructor）选择neigh_ops实例的标准，从*net/ipv4/arp.c*中定义的四个neigh_ops中选一个。

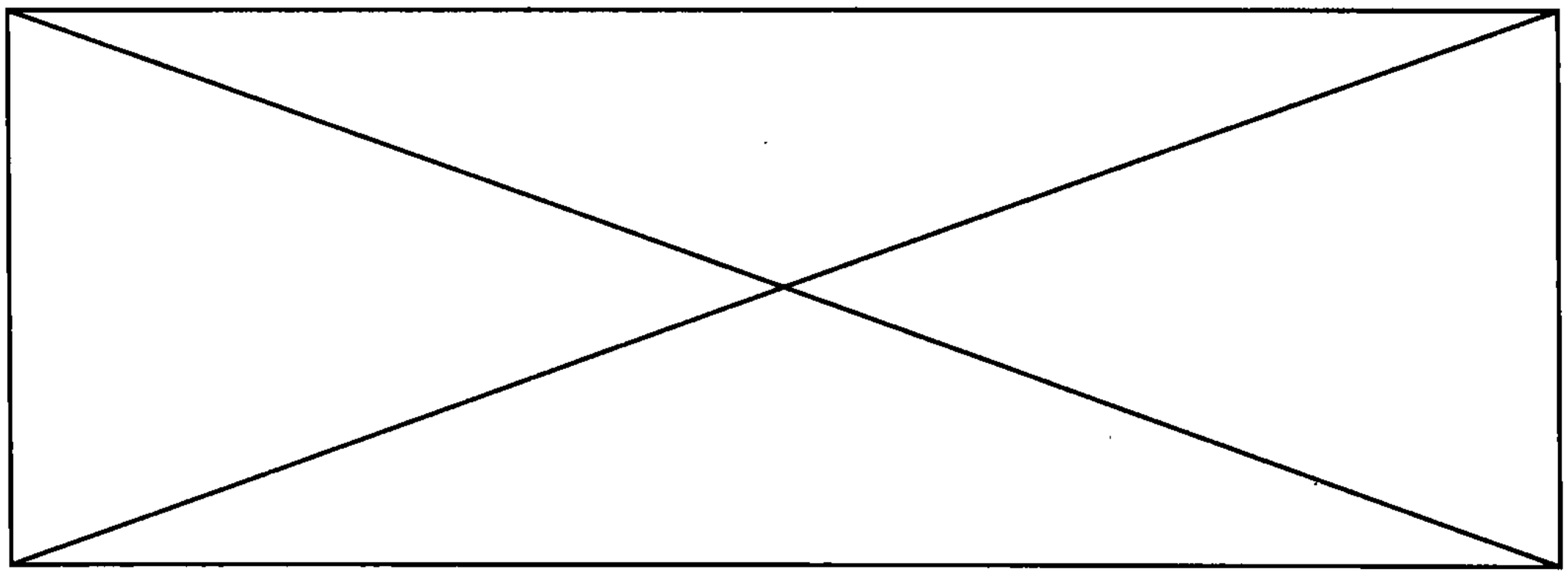


图28-12：在arp_constructor中的neigh->ops初始化

当neigh_create没有为刚才描述的这些字段指定明确的值时，它们就会继承neigh_alloc函数中指定的值。在neigh_create调用constructor虚函数前会调用neigh_alloc函数。

ops字段中的虚函数

在第二十七章的“L3协议和邻居协议间的通用接口”一节中，我们曾简要介绍了邻居基础结构在*net/core/neighbour.c*文件中提供的函数，那些函数由邻居协议使用output、

connected_output、hh_output和queue_xmit方法调用。本章中，我们重点讨论上述函数中与ARP协议相关的部分。根据邻居的状态，可以被指定给neigh->ops的四组方法是：

```
static struct neigh_ops arp_generic_ops = {
    .family:            AF_INET,
    .solicit:           arp_solicit,
    .error_report:      arp_error_report,
    .output:            neigh_resolve_output,
    .connected_output:  neigh_connected_output,
    .hh_output:         dev_queue_xmit,
    .queue_xmit:        dev_queue_xmit,
};

static struct neigh_ops arp_hh_ops = {
    .family:            AF_INET,
    .solicit:           arp_solicit,
    .error_report:      arp_error_report,
    .output:            neigh_resolve_output,
    .connected_output:  neigh_resolve_output,
    .hh_output:         dev_queue_xmit,
    .queue_xmit:        dev_queue_xmit,
};

static struct neigh_ops arp_direct_ops = {
    .family:            AF_INET,
    .output:            dev_queue_xmit,
    .connected_output:  dev_queue_xmit,
    .hh_output:         dev_queue_xmit,
    .queue_xmit:        dev_queue_xmit,
};

struct neigh_ops arp_broken_ops = {
    .family:            AF_INET,
    .solicit:           arp_solicit,
    .error_report:      arp_error_report,
    .output:            neigh_compat_output,
    .connected_output:  neigh_compat_output,
    .hh_output:         dev_queue_xmit,
    .queue_xmit:        dev_queue_xmit,
};
```

在四个neigh_ops实例（只是arp_direct_ops不需要某些字段，因此省略了它们的定义）中有三个字段在初始化ARP时都被设成相同的值。

family

AF_INET表示ARP和IPv4一起工作。

solicit

调用arp_solicit来生成一个solicitation请求，可能是首次解析一个地址，也可

能是要证实缓存中的地址。在后一种情况中，会由一个定时器的超时触发这个函数，见第二十七章的“定时器”一节。

ARP封包的传输是由arp_send函数实现的，在“发送ARP封包：arp_send的介绍”一节会对其介绍。

error_report

当一个ARP事务中发生错误时，arp_error_report函数就通知上面的网络层。见“产生的事件”一节。

arp_constructor函数的启动

arp_constructor的首要任务是从与邻居相关的网络设备中取出一个in_dev结构。这个结构中保存着该网络设备的IP层配置信息，其中也包括ARP的配置信息。如果没有ARP配置信息，该设备也就没有IP配置信息，因此在其上使用ARP也就没有意义；arp_constructor就终止执行，并返回一个错误。

如果设备中有IP配置信息，arp_constructor经由neigh->parms指针将ARP配置信息保存在相关的neighbour项中。

```
static int arp_constructor(struct neighbour *neigh)
{
    u32 addr = *(u32*)neigh->primary_key;
    struct net_device *dev = neigh->dev;
    struct in_device *in_dev;
    struct neigh_parms *parms

    neigh->type = inet_addr_type(addr);
    rcu_read_lock();
    in_dev = rcu_dereference(_ _in_dev_get(dev));
    if (in_dev == NULL) {
        rcu_read_unlock();
        return -EINVAL;
    }

    parms = in_dev->arp_parms;
    __neigh_parms_put(neigh->parms);
    neigh->parms = neigh_parms_clone(parms);

    rcu_read_unlock();

    if (dev->hard_header == NULL) {
        /* 情况1：设备没有使用L2*/
    } else {
        /*情况2：设备使用L2*/
    }
```

下面的步骤取决于网络设备驱动程序是否提供L2协议头：dev->hard_header。

不需要 ARP 的设备

当没有设置 dev->hard_header 时，表示设备驱动程序没有提供填充 L2 帧头的函数。也就是说设备不需要 L2 帧头，因此 neighbour 项的状态应该被设为 NUD_NOARP。此外，neigh_ops 应被初始化为 arp_direct_ops。arp_direct_ops 由一个 neigh_ops 结构体组成，该结构中的所有函数都被初始化为 dev_queue_xmit：因为不需要邻居协议，arp_direct_ops 直接访问底层函数。

```
neigh->nud_state = NUD_NOARP;
neigh->ops = &arp_direct_ops;
neigh->output = neigh->ops->queue_xmit;
```

注意，不必在每次将 state 设为 NUD_NOARP 态时，也将 neigh->ops 设为 arp_direct_ops。有些情况，例如，使用 IP 广播地址时，L2 层会使用一个帧头（dev->hard_header 非 NULL），但是不需要 ARP。

还有要注意的是，因为不需要 neigh->ha，所有它没被初始化。

需要 ARP 的设备

如图 28-12 所示，一旦 arp_constructor 确定某个设备需要 ARP，它就必须进一步区分两种设备：驱动程序更新到新的邻居基础结构的设备和仍使用旧驱动程序的设备（参见 *net/ipv4/arp.c* 文件中的注释掉的函数）。

通过 ARP 封包头中硬件类型字段可以识别设备的类型，*include/linux/if_arp.h* 文件中列出了 ARPHDR_*XXX* 的值。arp_constructor 使用这些类型值来区分旧驱动程序和新驱动程序。

现在，只有业余无线电设备和一些 WAN 接口卡仍然使用旧驱动程序。对于这些设备，neigh->ops 会初始化为 arp_broken_ops，后者内部包含了基于这些旧程序的虚函数。

```
        switch (dev->type) {
        default:
            break;
        case ARPHRD_ROSE:
#if defined(CONFIG_AX25) || defined(CONFIG_AX25_MODULE)
        case ARPHRD_AX25:
#if defined(CONFIG_NETROM) || defined(CONFIG_NETROM_MODULE)
        case ARPHRD_NETROM:
#endif
            neigh->ops = &arp_broken_ops;
            neigh->output = neigh->ops->output;
            return 0;
#endif
```

```
        ;}
#endif
```

对于其他设备，内核根据设备驱动程序的性能来初始化neigh->ops。如果驱动程序提供了管理L2 帧头缓存（dev->hard_header_cache）的函数，就使用arp_hh_ops。否则，选择通用函数arp_generic_ops。想要知道一个设备是否提供L2 帧头缓存，可以查看与该设备相关的xxx_setup函数（例如，对于Ethernet 网卡，就是ether_setup函数，参见第八章的内容）。

```
if (dev->hard_header_cache)
    neigh->ops = &arp_hh_ops;
else
    neigh->ops = &arp_generic_ops;
```

在前面的“基本的初始化顺序”一节中介绍过，neigh->output的初始化是根据nud_state决定的。例如，当neighbour结构即将被使用时（NUD_VALID态），output函数就直接初始化为connected_output。关于neigh->output的详细内容可以参见第二十七章的“neigh->output 使用的函数”一节。

```
if (neigh->nud_state&NUD_VALID)
    neigh->output = neigh->ops->connected_output;
else
    neigh->output = neigh->ops->output;
```

回环设备（lo）和设置IFF_NOARP标识的设备不需要使用ARP解析地址。但是，邻居子系统仍需要一个地址来放到L2 帧头中，这些函数就指定一个与该设备相关的地址。

```
neigh->type = inet_addr_type(addr);
.....
if (neigh->type == RTN_MULTICAST) {
    neigh->nud_state = NUD_NOARP;
    arp_mc_map(addr, neigh->ha, dev, 1);
} else if (dev->flags&(IFF_NOARP|IFF_LOOPBACK)) {
    neigh->nud_state = NUD_NOARP;
    memcpy(neigh->ha, dev->dev_addr, dev->addr_len);
} else if (neigh->type == RTN_BROADCAST || dev->flags&IFF_POINTOPOINT) {
    neigh->nud_state = NUD_NOARP;
    memcpy(neigh->ha, dev->broadcast, dev->addr_len);
}
```

传输和接收 ARP 封包

用于发送和接收 ARP封包的函数分别是：

arp_send

邻居子系统调用neigh_ops->solicit来发出solicitation 请求。在ARP 中，

solicit 函数（arp_solicit）只是对 arp_send 的简单包裹。arp_send 负责填充 ARP 包头和封包内容，并调用 dev_queue_xmit 函数传送这个 ARP 请求。

arp_rcv

由于 ARP 是一个完整功能的协议（和 IPv6 中的 ND 不同），因而它要在 arp_init 中注册一个处理函数。下一节将详细描述 arp_rcv 函数，以及如何处理两种主要的 ARP 封包。

如图 28-13 所示，ARP 封包的传输和接收可以被 Netfilter 控制。

图28-13：(a) arp_rcv; (b) arp_send

图中 arp_rcv 和 arp_send 之间的虚线表明，在某些情况下收到一个 ARP 封包会导致发出至少一个其他的 ARP 封包。这些情况是：

- 配置了网桥。收到 ARP 封包的网桥可能不对其做任何处理，只是转发到其他网桥的接口上。
- 入口的 ARP 封包是 ARPOP_REQUEST 类型，且邻居子系统根据自身的配置决定作出应答。然后邻居子系统就发出 ARPOP_REPLY 类型的封包。

如图 28-13 所示，arp_send 可以由外部事件和少数内核功能（比如，bonding 功能）触发；后面的小节对此有详细介绍。

发送 ARP 封包：arp_send 的介绍

arp_send 是 ARP 用于传输 solicitation 请求和应答的程序，如图 28-14 所示。第二十七章从协议无关的角度，解释了邻居基础结构如何管理 solicitation 的传输和传播。下面我们看 arp_send 函数如何完成它的工作。

如图 28-13(b)所示，arp_send 分为两个部分：arp_create 初始化 ARP 封包、arp_xmit 在 Netfilter 中设置钩子，然后调用 dev_queue_xmit 函数。

arp_send 之所有分为这么两部分，是为了使需要操作封包的驱动程序（比如，在封包中插入附加的头信息），可以分别调用 arp_create 和 arp_xmit。这样，驱动程序就可以可在两个函数中间执行用户定制的任务。例如，如果需要，可让 bonding 程序在封包中添加 802.1Q 标记（rlb_update_client 函数在 *drivers/net/bonding/bond_alb.c* 文件中）。

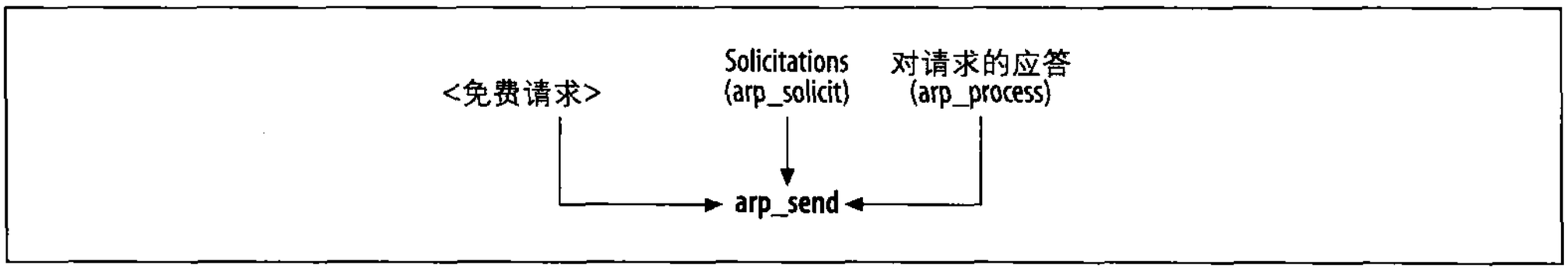


图 28-14：使用 arp_send 时的上下文环境的例子

Solicitations

在第二十七章的“创建一个邻居项”一节，我们已经知道内核会在什么时候生成 solicitation 请求。在本章中，我们分析 ARP 完成这个任务的函数：arp_solicit。

arp_solicit的调用者负责计算探测（发送solicitation的尝试）的次数，并保证还未到达最大值。因此，arp_solicit对此并不关心。

下面是这个函数的原型和其输入参数的含义：

```
static void arp_solicit(struct neighbour * neigh, struct sk_buff *skb)
```

neigh

L3 地址需要被解析的邻居。

skb

保存数据封包的缓冲区，该封包的传输由 solicitation 请求的产生触发。

要理解 arp_solicit 函数的实现，理解下面两组参数的关系和区别是很重要的：

- skb 缓冲区中 IP 头的源 IP 地址，和 arp_solicit 选择的要放到 ARP 头（参见图 28-1 中的 ARP 头格式）中的源 IP 地址。

  如果封包是本地网络生成的，IP 包头中源 IP 地址就是本地主机的。若封包是要转发的，源 IP 地址就是最初发送方的。

- skb 缓冲区 IP 包头中的目的 IP 地址，和 arp_solicit 请求解析（neigh->primary_key）的目的 IP 地址。

  请求 ARP 解析的地址是路由 skb 的下一跳地址。只有下一跳就是最终目的地时，IP 包头内的目的 IP 地址才会和请求 ARP 解析的地址一致。

arp_solicit 的主要任务是：

- 选择要放到 ARP 头中的源 IP 地址。“/proc 选项”一节中提到的 ARP_ANNOUNCE 选项会影响这个选择过程。图 28-15 给出了 arp_solicit 函数的内部结构，并解释了源 IP 地址是如何选择的。
- 更新生成的 solicitation 请求的数目。
- 使用 arp_send 函数传输请求。

下一节将详细解释如何选择源IP 地址。我们先简单说一下其他两个任务是如何实现的。

arp_solicit要区分内核生成的请求和用户空间产生的请求。当ARP 守护进程*arpd*运行时，会发生后一种情况；这也要求内核编译的时候选择支持 ARPD，详细的论述在“ARPD ”一节中。两种情况按照下面的方式处理：

- 对于内核生成的请求，使用 arp_send 传输 solicitation 请求。

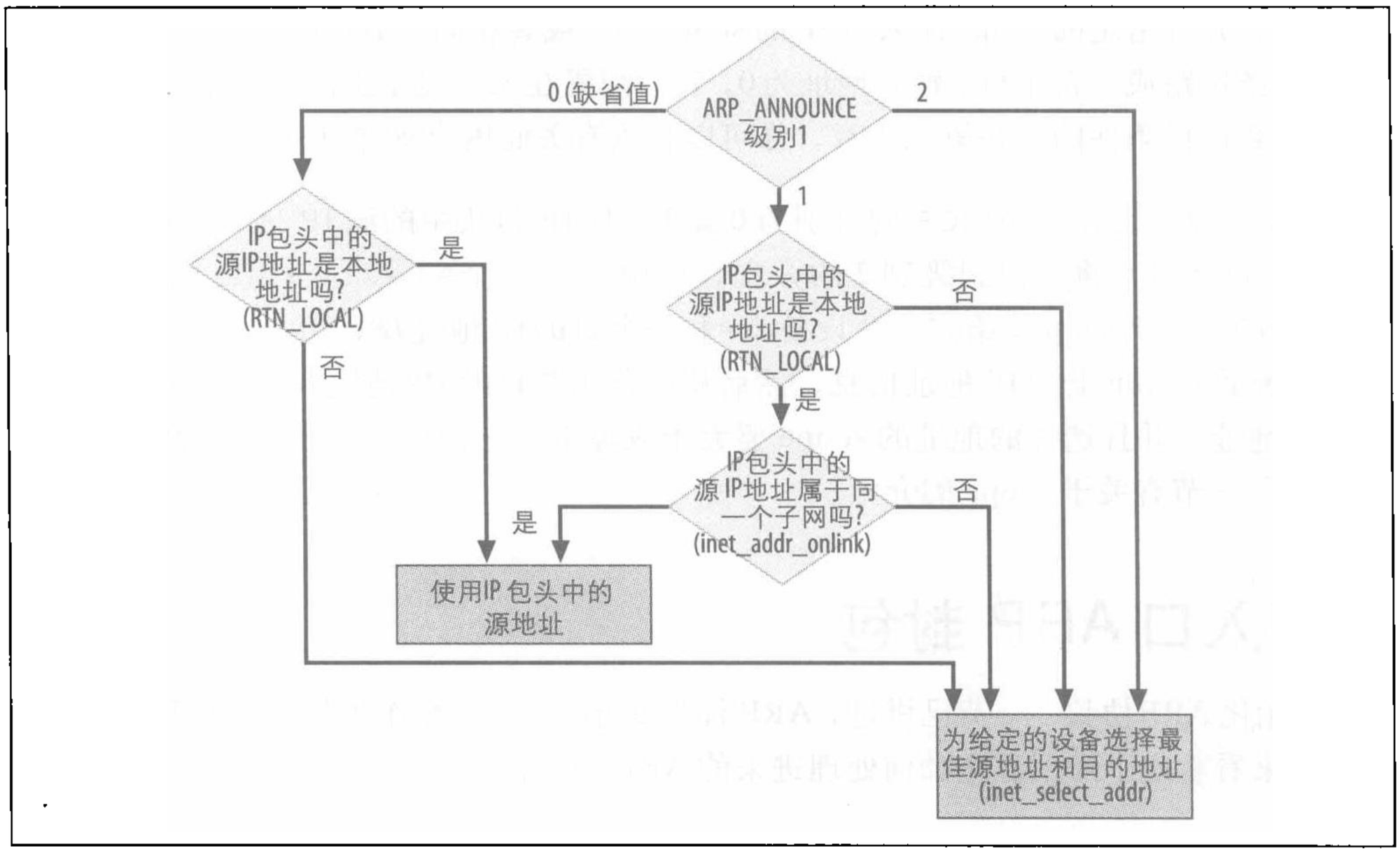


图 28-15：arp_solicit 中源 IP 的选择

- 对于用户空间产生的请求，arp_solicit 调用 neigh_app_ns 来通知相应的用户空间程序：需要生成一个solicitation请求。如果内核不支持ARPD，arp_solicit 就不做 solicitation 请求，直接返回。

ARP_ANNOUNCE 和源 IP 地址的选择

大部分主机只有一个IP地址，因此将其直接拷贝到ARP头中。若一个主机有多个IP 地址，ARP_ANNOUNCE的设置就会影响IP 的选择。arp_solicit 根据表 28-1 和图 28-15 中的描述进行选择。为了完成这个工作，它要利用路由和配置子系统提供的三个函数：

inet_addr_type

在该函数的输入参数中给一个IP 地址，它就返回该地址的类型。在本章的内容中，我们感兴趣的类型是 RTN_LOCAL，它表示这个地址属于本地主机。

inet_addr_onlink

输入一个设备和两个 IP 地址，该函数会检查这两个地址是否属于同一个子网。

inet_select_addr

输入一个设备、一个 IP 地址（通常不是本地主机的地址）和一个 scope，该函数会在设备配置信息中查找这个IP地址，该地址应该和入口IP地址位于同一个子网

中，并且给定的scope比入口IP的scope小，或者相同。典型的范围是一个站点、一条链路或一台主机。输入地址为0，表示配置在入口设备上的任一主要地址都是符合选择条件的。在第三十章，你可以找到有关此内容更详细的介绍。

注意到，当ARP_ANNOUNCE的级别为0或1，且IP包头中的源IP地址不能使用时，arp_solicit将退回到级别2的分支。调用inet_select_addr函数要用到RT_SCOPE_LINK scope。给定一个设备*dev*和一个目的IP地址*IP*，inet_select_addr要浏览配置在*dev*上的IP地址信息，然后从中选出与目的IP地址*IP*的子网相匹配的第一个地址，并且选择的地址的scope要大于或等于RT_SCOPE_LINK。第三十章的“Scope”一节有关于scope的介绍。

处理入口ARP封包

在“初始化ARP协议”一节已讲过，ARP注册arp_rcv函数作为其协议处理程序。下面我们来看看这个处理程序如何处理进来的ARP封包。

ARP封包可以从skb缓冲区中得到，skb也是函数arp_rcv的输入参数；ARP包头就是skb->nh.arph。该函数的首要任务是确保收到的ARP封包不是成碎片的；也就是说，内存中的该封包可以线性访问。这个任务是必需的，因为有时内存中的skb缓冲区是些碎片（注7）。如果不是碎片，arp_rcv就调用通用函数pskb_may_pull来保证在主缓冲区中有足够的空间，这些用于存放ARP包头和负载。

```
int arp_rcv(struct sk_buff *skb, struct net_device *dev, struct
packet_type *pt)
{
    struct arphdr *arp;

    /* ARP header, plus 2 device addresses, plus 2 IP addresses.  */
    if (!pskb_may_pull(skb, (sizeof(struct arphdr) +
                (2 * dev->addr_len) +
                (2 * sizeof(u32)))))
        goto freeskb;
```

如果满足下面的条件，输入的ARP封包就会被arp_rcv丢弃。

- 收到封包的设备是没有使用ARP协议的设备（也就是说，设备有IFF_NOARP标识）。

  回环接口是一种特殊情况。其发送和接收的封包用PACKET_LOOPBACK类型识别。因为这个接口是虚拟的，并没有实际的硬件地址，因此也不需要使用ARP协议。

注7：这种情况与IP包碎片无关。第二章有详细介绍。

- 封包的目的地不是收到封包的接口（目的地址不是接收接口的地址或广播地址）。

当缓冲区被共享时（其他程序可以对其引用），arp_rcv将用skb_share_check函数克隆一个缓冲区。克隆是很有必要的，因为当处理ARP封包时，要保证没有人会改变skb的内容（特别是skb的头指针）。详细内容可以参见第2章的“缓冲区的克隆和拷贝”一节。

参见“ARP 封包格式”一节中有关 SIP 和 TIP 的含义。一旦入口 ARP 封包已经准备进行处理，假设 Netfilter 没有拦截它，arp_process 函数就会负责处理，如图 28-13 所示。

图 28-16 给出了 arp_process 函数的结构。该函数首先会对它能理解的所有 ARP 封包类型进行合理性检查，然后根据封包类型执行相应的操作。函数的最后一部分是另一些常用代码，负责更新 ARP 缓存（如果对应的缓存项没有被锁定），参见“普通处理的结尾”一节。由于对多播IP地址的请求会被丢弃，因为它们是不合规定的：在第二十六章的“特殊情况”一节我们曾看到，多播 IP 地址不需要使用 ARP 来转换成链路层地址。

普通处理的初始化

ARPOP_REQUEST类型和ARPOP_REPLY类型的封包都由arp_process进行处理的。其他任何类型的 ARP 封包都会被丢弃。目的地址是多播或广播地址的封包［可以通过 LOOPBACK 宏和 MULTICAST 宏检测到（注 8）］也被丢弃，因为它们不需要 ARP。可以参见前面的“ARP 封包中的目的地址类型”一节和第二十六章的“特殊情况”一节。

某些设备类型，只有编译内核时明确指定支持它们，内核才会支持。它们不属于默认设备，因为不会经常使用，因此内核开发者决定将对它们的支持设为可选项，以减少内核的大小。下面给出的switch语句对设备类型挨个判断（使用#ifdef语句保证每个设备已编译进内核），然后检查对于该设备类型，ARP封包中指定的协议是否正确。这部分代码很长而且有部分重复。

```
switch (dev_type) {
default:
    if (arp->ar_pro != htons(ETH_P_IP)) ||
        htons(dev_type) != arp->ar_hrd)
        goto out;
    break;
```

注 8： LOOPBACK 宏可识出地址 127.*x*.*x*.*x* 而 MULTICAST 宏可识别出地址 224.*x*.*x*.*x*（class D）。

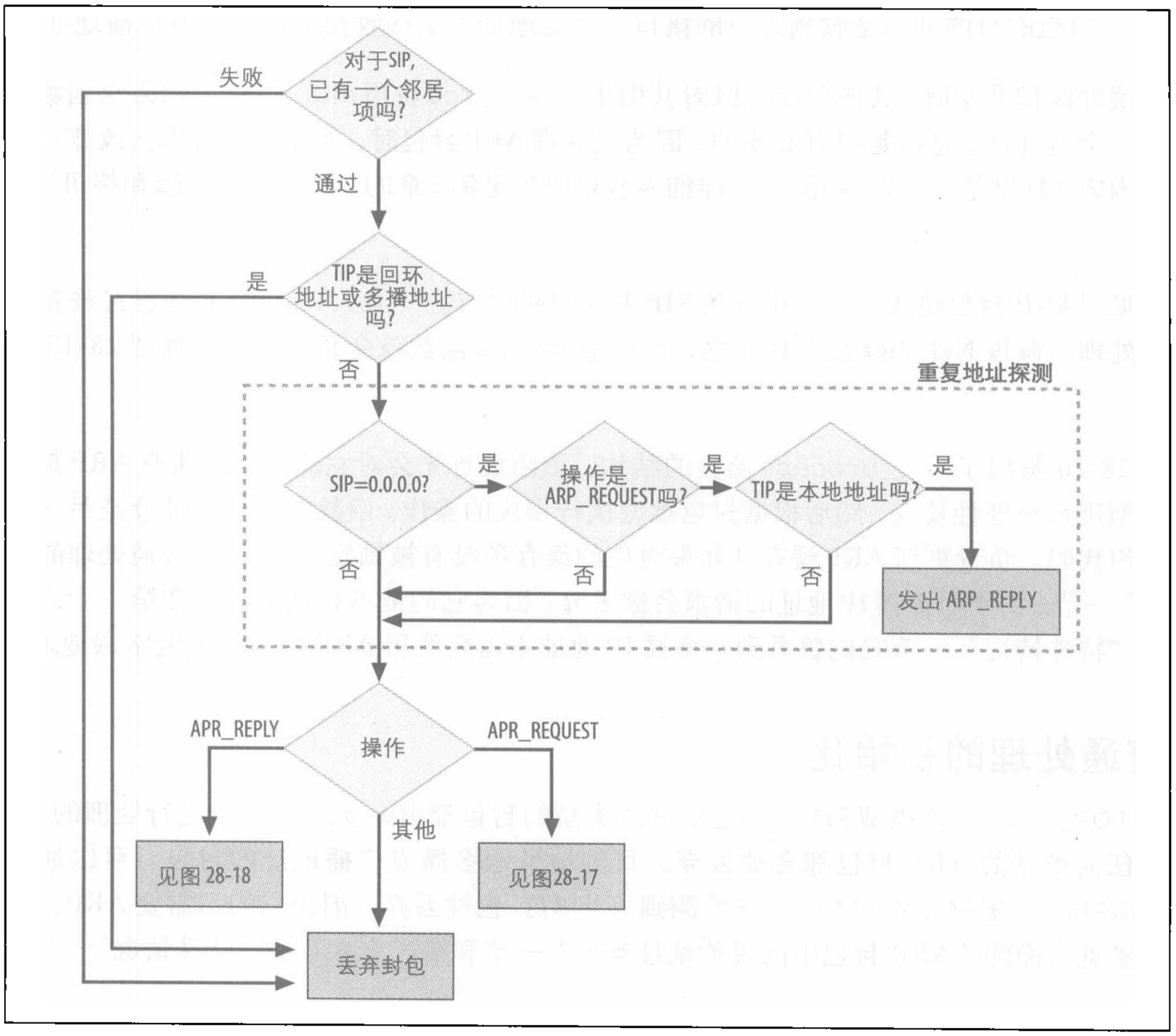


图 28-16：arp_process 函数

```
#ifdef CONFIG_NET_ETHERNET
    case ARPHRD_ETHER:
    ......
        if (arp->ar_hrd != htons(ARPHRD_ETHER) &&
            arp->ar_hrd != htons(ARPHRD_IEEE802)) ||
            arp->ar_pro != htons(ETH_P_IP))
            goto out;
        break;
#endif
#ifdef CONFIG_TR
    case ARPHRD_IEEE802_TR:
            ......
#endif
            ......
#endif
    }
```

arp_process 函数的最后一个任务是初始化一些局部变量，从 ARP 头中的字段变量到可以使后面的代码更清晰的变量。这里没有给出这部分代码，但通过图 28-1 很容易理解。arp_ptr 指向硬件头的末尾。

处理 ARPOP_REQUEST 封包

图 28-17 是对 arp_process 如何处理 ARPOP_REQUEST 封包的更高级描述。arp_process 对本地地址的请求和对非本地地址的请求都可以处理。图中的左半部分给出的就是对非本地地址的请求处理，见“ARP 代理”一节。表 28-4 给出了 SIP 和 TIP 的含义。

表 28-4：ARP 包中提取的参数

ARP 包中的字段	局部变量名
发送方 Ethernet 地址	sha
发送方 IP 地址	sip
目的 Ethernet 地址	tha
目的 IP 地址	tip

只有下面条件都满足，才会处理 ARPOP_REQUEST 封包：

- 内核知道如何访问发送方请求的地址（在路由表中，有一条到该地址的有效路由）。

```
if (arp->ar_op == htons(ARPOP_REQUEST) &&
    ip_route_input(skb, tip, sip, 0, dev) == 0) {
    /* 处理包 */
}
```

这是一种对 IP 地址请求过滤的简单方式，本地系统并不知道这些地址。若本地系统是一台主机，它只对配置在本地接口上的 IP 地址的请求做出应答。若本地系统是台 ARP 代理服务器，它会对属于配置在其本地接口上子网内的 IP 地址的请求作出应答（也是说，IP 地址属于邻居主机）。

在第七部分我们将看到，路由子系统会为本地主机的每个 IP 地址添加一个路由项，为这些 IP 地址所属的每个子网也添加一个路由项。因此，这两种情况下，路由查找程序会过滤掉对本地主机不应该应答的请求。

- 请求的地址既可以属于本地系统，也可以是这个作为代理服务器的主机能够处理的远程地址。本节中，我们使用 RTN_LOCAL 标识表示请求的地址属于本地地址。“ARP 代理”一节中我们介绍远程地址的情况。

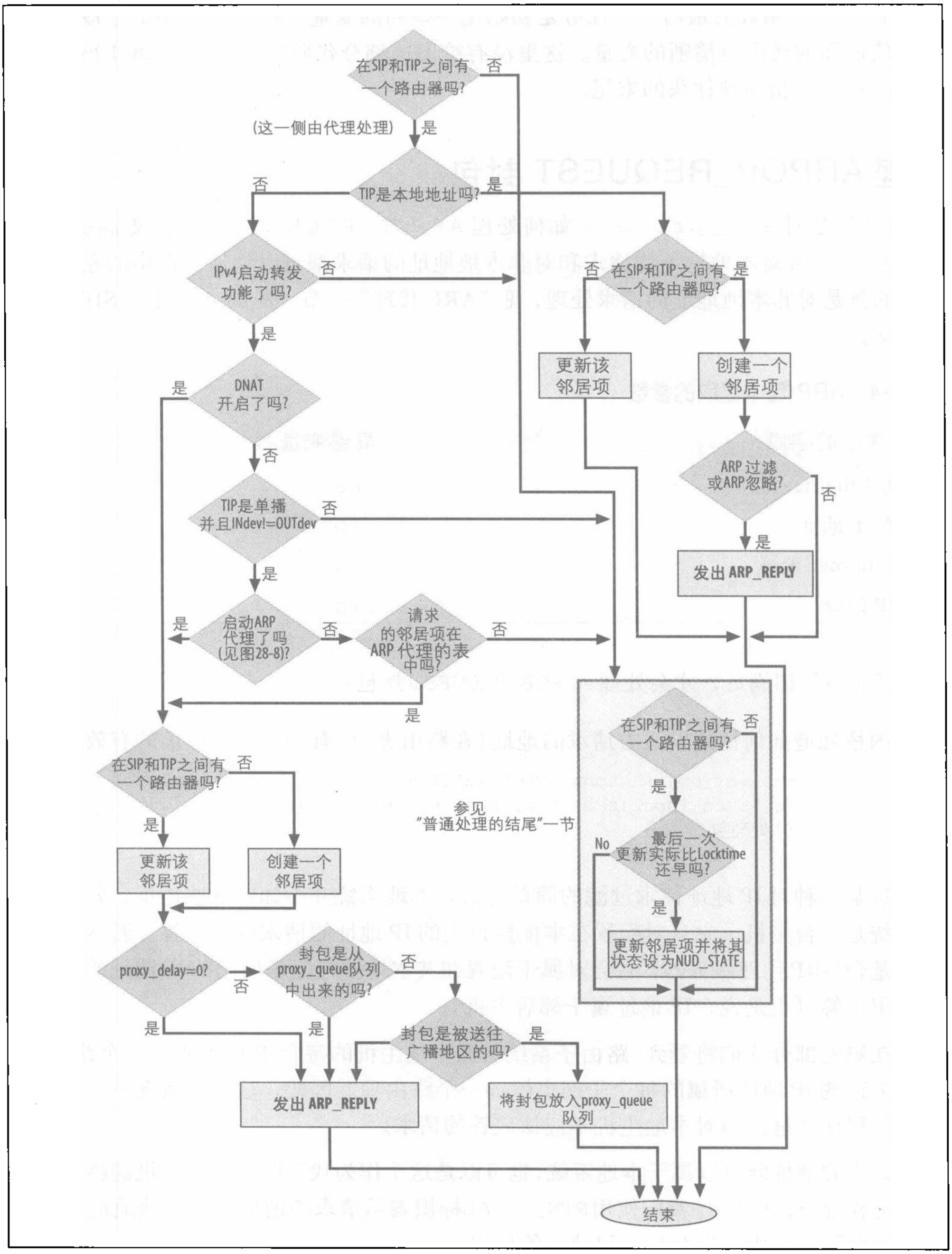


图 28-17：arp_process 处理 ARPOP_REQUEST 封包

- 没有明确配置来禁止ARPOP_REPLY的传输（参见前面的“ARP_IGNORE”和“ARP_FILTER”两小节）。

如果所有条件都满足，arp_process就调用arp_send，并给其正确的输入参数。在“发送ARP封包：arp_send的介绍”一节有对arp_send的描述。

```
rt = (struct rtable*)skb->dst;
addr_type = rt->rt_type;

if (addr_type == RTN_LOCAL) {
    n = neigh_event_ns(&arp_tbl, sha, &sip, dev);
    if (n) {
        int dont_send = 0;
        if (!dont_send)
            dont_send |= arp_ignore(in_dev,dev,sip,tip);
        if (!dont_send && IN_DEV_ARPFILTER(in_dev))
            dont_send |= arp_filter(sip,tip,dev);
        if (!dont_send)
            arp_send(ARPOP_REPLY,ETH_P_ARP,sip,dev,
                                    tip,sha,dev->dev_addr,sha);
        neigh_release(n);
    }
    goto out;
} else {

        /* 遇到所有的请求情况时处理ARP代理,
        参见“ARP 代理”一节 */
```

被动学习和优化ARP

在第二十七章的“创建一个邻居项”一节曾提到，在一个ARP事务结束时，请求方和应答方都会学到一些东西。请求方达到了从ARPOP_REPLY封包中其学到目的地址的主要目标；这个过程就称为主动学习（active learning）。而收到ARPOP_REQUEST封包的目的主机从请求封包中也知道了发送方的地址；这个过程称为被动学习（passive learning）。这两个过程可使邻居协议的性能达到最优化。

被动学习由neigh_event_ns负责。该函数会检查是否有一个邻居项和请求者相关联；然后就会更新这个存在的邻居项，或者当不存在时，创建一个新的邻居项。

不管是更新存在的邻居项，还是创建新的邻居项，该函数都要将邻居状态设为NUD_STALE态。ARP不会采取乐观的措施，将邻居设为NUD_REACHABLE态，因为这个状态是主机已经提供可到达性证明后才能设置，见第二十七章中描述的设置该状态的严格条件。

当neigh_event_ns不能创建邻居项时（通常是因为内存不够、缓存中没有足够的空间），它就返回NULL。当发生这种情况时，应答也就不会发给请求者。这个策略有点

保守；比较积极的方式根据上述情况做出应答，即使我们暂时无法在我们系统上为该邻居创建一个邻居缓存项，但我们不应该剥夺它向我们传输数据的能力。

neigh_event_ns要调用第二十七章“缓存”一节介绍的查找函数中的一个。因为当查找成功时，总是会使邻居项的引用计数器加1，而neigh_event_ns需要将该计数器相应递减。

零地址的请求

当一个ARP请求中的源IP地址被设为0时（标准格式为0.0.0.0），它就可能是个被破坏的封包，因为0.0.0.0不是一个有效的IP地址。但是这也可能是DHCP用于检测重复地址的特殊封包。参见前面的“重复地址检测”一节中关于发送这些封包的条件，以及RFC 2131中第2.2节关于0地址使用的介绍。

DHCP服务器或客户端可以随意地对DHCP分配的一个IP地址发出一个ARPOP_REQUEST，用于双向检测是否相同的地址被错误地用在另一台主机上。这种特殊的ARPOP_REQUEST送出时，其源IP地址是0.0.0.0，以便它不会给子网内的其他主机引来任何麻烦。

当源IP地址（sip）为0时，下面的arp_process部分代码就会运行，并在当封包的发送者生成这种请求时，让本地主机申请一个地址。

```
if (sip == 0) {
    if (arp->ar_op == htons(ARPOP_REQUEST) &&
        inet_addr_type(tip) == RTN_LOCAL &&
           !arp_ignore(in_dev, dev, sip, tip))
        arp_send(ARPOP_REPLY,ETH_P_ARP,tip,dev,tip,sha,
                              dev->dev_addr,dev->dev_addr);
    goto out;
}
```

处理ARPOP_REPLY封包

如果有下列任一条件满足，输入的ARPOP_REPLY封包就会被处理：

- 存在一个与接收的ARPOP_REPLY相匹配的未决的ARPOP_REQUEST。换句话说，这个ARPOP_REPLY是对内核先前生成的ARPOP_REQUEST的应答，这是最普通的情况。
- 没有未决的ARPOP_REQUEST，但内核支持UNSOLICITED-ARP（见“Compile-Time Options”一节）。在这种情况时，通过调用_ _neigh_lookup函数创建一个新邻居项，且该函数的最后一个参数不能是NULL。

```
#ifdef CONFIG_IP_ACCEPT_UNSOLICITED_ARP
    if (n == NULL &&
        arp->ar_op == htons(ARPOP_REPLY) &&
        inet_addr_type(sip) == RTN_UNICAST)
        n = _ _neigh_lookup(&arp_tbl, &sip, dev, -1);
#endif
```

图 28-18 的左右两边分别表示这两种情况的处理过程。

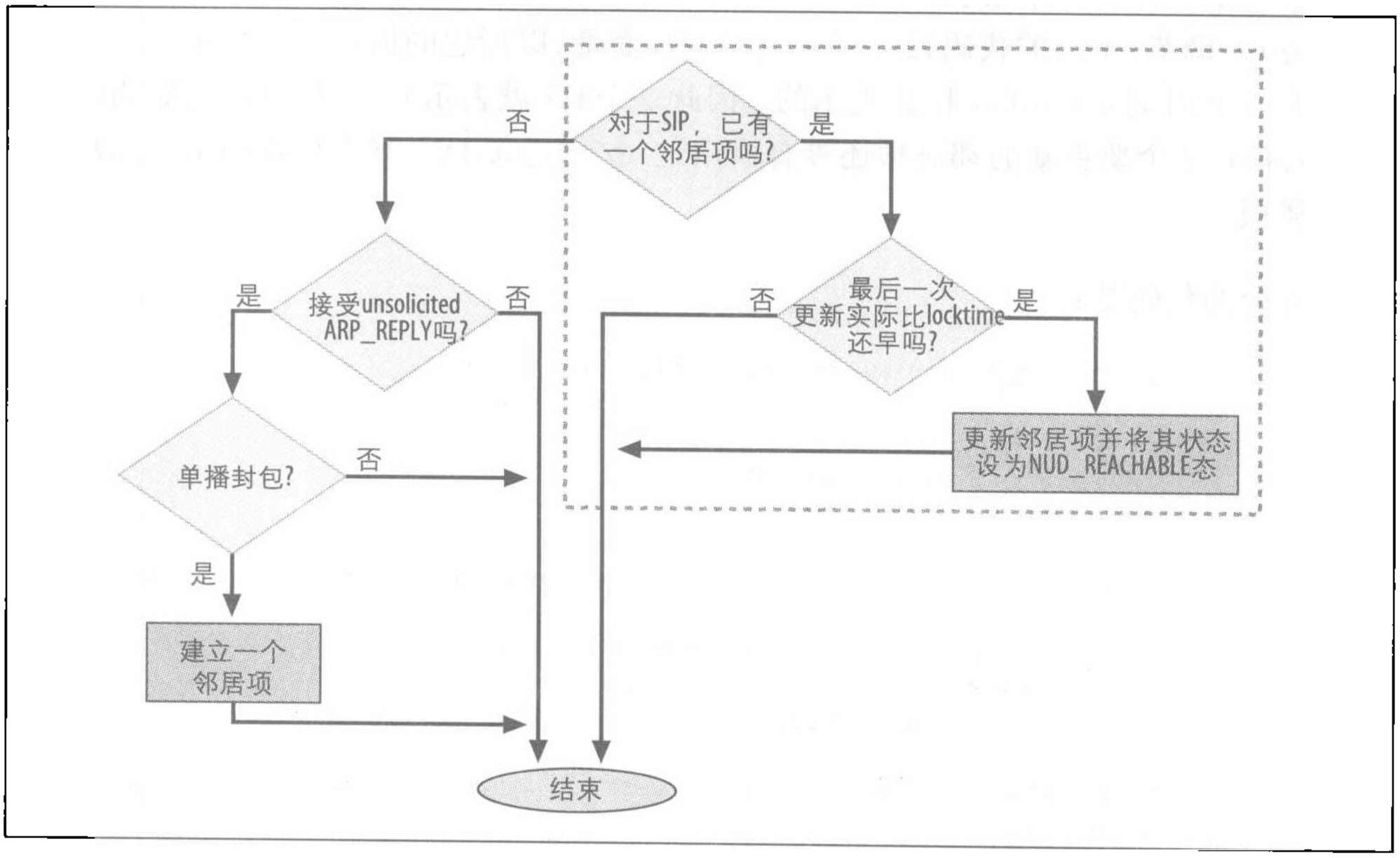


图 28-18：arp_process 处理 ARPOP_REPLY

撇开封包被接受的原因，已有的neighbour项会被下一节要介绍的通用代码（上图中的虚线框部分）更新，以便反映出 ARPOP_REPLY 封包中的信息。

普通处理的结尾

对于所有的 ARPOP_REPLY 封包和还没有被处理的 ARPOP_REQUEST 封包（因为它们不满足“处理 ARPOP_REQUEST 封包”一节中列出的条件），arp_process 函数都要执行它最后的那部分代码。

记住，当一个主机对一个 ARPOP_REQUEST 做出应答时，它会交换 ARP 头中的源地址和目的地址字段，对于空字段也会填入相应的内容。

在读这些代码时，另一个需要理解的概念是：锁定期（locktime）。这个概念与内核经常用的加锁的信号量类型没有关系。它只是一种简单的超时机制，负责处理对同一个ARPOP_REQUEST封包，主机可能收到多个ARPOP_REPLY封包的偶然事件。如果某种配置错误或在同一个LAN中有多个ARP代理服务器，那么就可能发生这种情况；对此，arp_process函数只会对第一个应答做出反应，而拒绝后面的应答。

处理机制如下：邻居子系统在neigh_table结构中引入locktime参数；该参数也可以通过*/proc*调节。下面的代码设置了override变量，以后它的值就能反映出locktime（locktime时间是用jiffie格式表示的，因此一个HZ就表示1s）。如果在前面的锁定期内，只有对某个要更新的邻居项还没有调用过neigh_update函数，才调用该函数更新该邻居项。

最后部分的代码是：

```
        n = _ _neigh_lookup(&arp_tbl, &sip, dev, 0);
        ...
        if (n) {
            int state = NUD_REACHABLE;
            int override;

            override = time_after(jiffies, n->updated + n->parms->locktime);

            if (arp->ar_op != htons(ARPOP_REPLY) ||
                skb->pkt_type != PACKET_HOST)
                state = NUD_STALE;

            neigh_update(n, sha, state, override ? NEIGH_UPDATE_F_OVERRIDE : 0);
            neigh_release(n);
    }
```

这些代码必须选择出正确的状态，指定给要更新的neighbour项。在第26章的“可到达性”一节中已讲过，单播应答和广播应答的权限级别不同。单播应答（PACKET_HOST）会设置邻居为UD_REACHABLE态，而广播应答是设为NUD_STALE态。ARPOP_REQUEST封包触发的更新总是使相应邻居的状态转为NUD_STALE态。

ARP 代理

在“处理入口ARP封包”一节中，我们知道了arp_process如何处理对本地地址的请求。现在我们来看arp_process对远端地址的请求是如何处理的。

我们在第二十六章的“使用代理的条件”和第二十七章的“按设备代理和按目的地代理”两节中已经知道，内核支持两种类型的代理：基于设备代理和基于目的地（全局）代理。在一台主机上，其于设备代理默认是关闭的。通过*/proc* 接口既可以启动按设备代理，也

可以启动全局代理。内核通过检查在 *include/linux/inetdevice.h* 中定义的 IN_DEV_PROXY_ARP 宏，来确定在某个设备上是否启动了 ARP 代理。其于目的地代理可以通过 *arp* 命令配置，也可以通过 *ip neigh* 命令配置（见第二十九章的“邻居的系统管理”一节）。

ARP 中增加了一个执行代理的条件：目的地网络地址转换（Destination Network Address Translation，DNAT）。在“目的NAT (DNAT)”一节将知道，若配置了DNAT，内核为何要代理 ARP 请求。

代理服务器能够处理的 ARPOP_REQUEST 请求必须满足下列条件：

- 在接收封包的设备、或代理服务器所有设备上启动了转发功能。
- 目标IP 地址是单播地址（在第二十六章的“特殊情况”一节中已知道，其他类型的地址不需要被解析）。在程序代码中，就是 addr_type==RTN_UNICAST。
- 收到这个请求的设备不是到达目的IP 地址要使用的设备（如果是后者，就不需要代理服务器：目的主机自己会应答）。在程序代码中，就是 rt->u.dst.dev!=dev。

下面的代码来自 arp_process 函数，展示了 arp_process 函数如何检查上述几个条件：

```
if (addr_type == RTN_LOCAL) {
                ......
} else if (IN_DEV_FORWARD(in_dev)) {
    if ((rt->rt_flags&RTCF_DNAT) ||
        (addr_type == RTN_UNICAST  && rt->u.dst.dev != dev &&
         (arp_fwd_proxy(in_dev, rt) ||
                pneigh_lookup(&arp_tbl, &tip, dev, 0)))) {
```

如果上述基本条件都满足，代理程序就要检查基于设备代理和基于目的地代理的配置信息。第二十六章的图 26-8 给出了这个检查过程。下面的条件决定了代理服务器是否对请求作出反应。

- 在某个设备上或所有设备上都启动了 ARP 代理。
- “Medium ID”一节已经讲过，入口接口和出口接口不是同一个介质。
- 代理过的地址数据库中有目的地址。这个数据库是按照目的地址组织的，并且通过 pneigh_lookup 函数进行查询操作。

这里假设 arp_process 函数对 ARPOP_REQUEST 请求的处理开了绿灯。

首先，用 neigh_event_ns 来创建（或更新）发送方 IP 对应的 neighbour 结构，参见“被动学习和优化 ARP”一节中描写的 ARP 对本地地址的请求时处理。

为了防止网络流量的突然增加，ARP代理可以被延迟处理，见第二十七章“延迟处理Solicitation请求”一节的描述。这样的话，如果封包是直接来自另一个主机，并且配置了延迟处理，那么该封包就被送到代理队列中。如果封包是来自代理队列（也就说该封包以前进入了队列，现在该对其进行处理了），或者没有配置延迟处理，那么就立即处理该封包。

```
n = neigh_event_ns(&arp_tbl, sha, &sip, dev);
if (n)
    neigh_release(n);

if (skb->stamp.tv_sec == LOCALLY_ENQUEUED ||
    skb->pkt_type == PACKET_HOST ||
    in_dev->arp_parms->proxy_delay == 0) {
        arp_send(ARPOP_REPLY,ETH_P_ARP,
                                    sip,dev,tip,sha,dev->dev_addr,sha);
    } else {
        pneigh_enqueue(&arp_tbl, in_dev->arp_parms, skb);
        in_dev_put(in_dev);
        return 0;
    }
    goto out;
  }
}
```

目的NAT（DNAT）

IPROUTE2软件包中，目的NAT也称为路由NAT（Route NAT），允许主机定义伪地址（NAT）：主机检测到目的地是这些伪地址的入口封包后，将其转发到另一些地址。主要是路由器使用DNAT，与Netfilter实现的Destination NAT没有任何关系（注9）。

尽管Linux中的ARP程序要处理DNAT，但路由程序中似乎不再支持它。因此，在2.6版的内核中已取消了该特性。

图28-19给出了DNAT的一个例子。路由器RT配置了一个伪NAT地址10.0.0.5。无论RT什么时候收到目的地址是10.0.0.5的流量，它都将封包的目的地址改为10.0.1.10，并转发到该地址对应的主机。当然，这个配置要保证发向流量也可以处理。

所有这一切都是由ARP代理实现的。在10.0.0.0/24子网中，没有主机配置了地址10.0.0.5。但是，这个地址是作为某台特定主机的地址（例如，一台web服务器）向外公布的。子网10.0.0.0/24内的主机无论什么时候想与10.0.0.5通信，它就发出一个对

注9：Linux支持所有的NAT。SNAT、DNAT、伪装（Maspuerading）等都是用Netfilter实现的。由于本书不涉及Netfilter的内部结构，所以我也没有在本书中论述NAT。有关各种NAT间的区别，可以参见Netfiter项目的网页：*http://www.netfilter.org*。

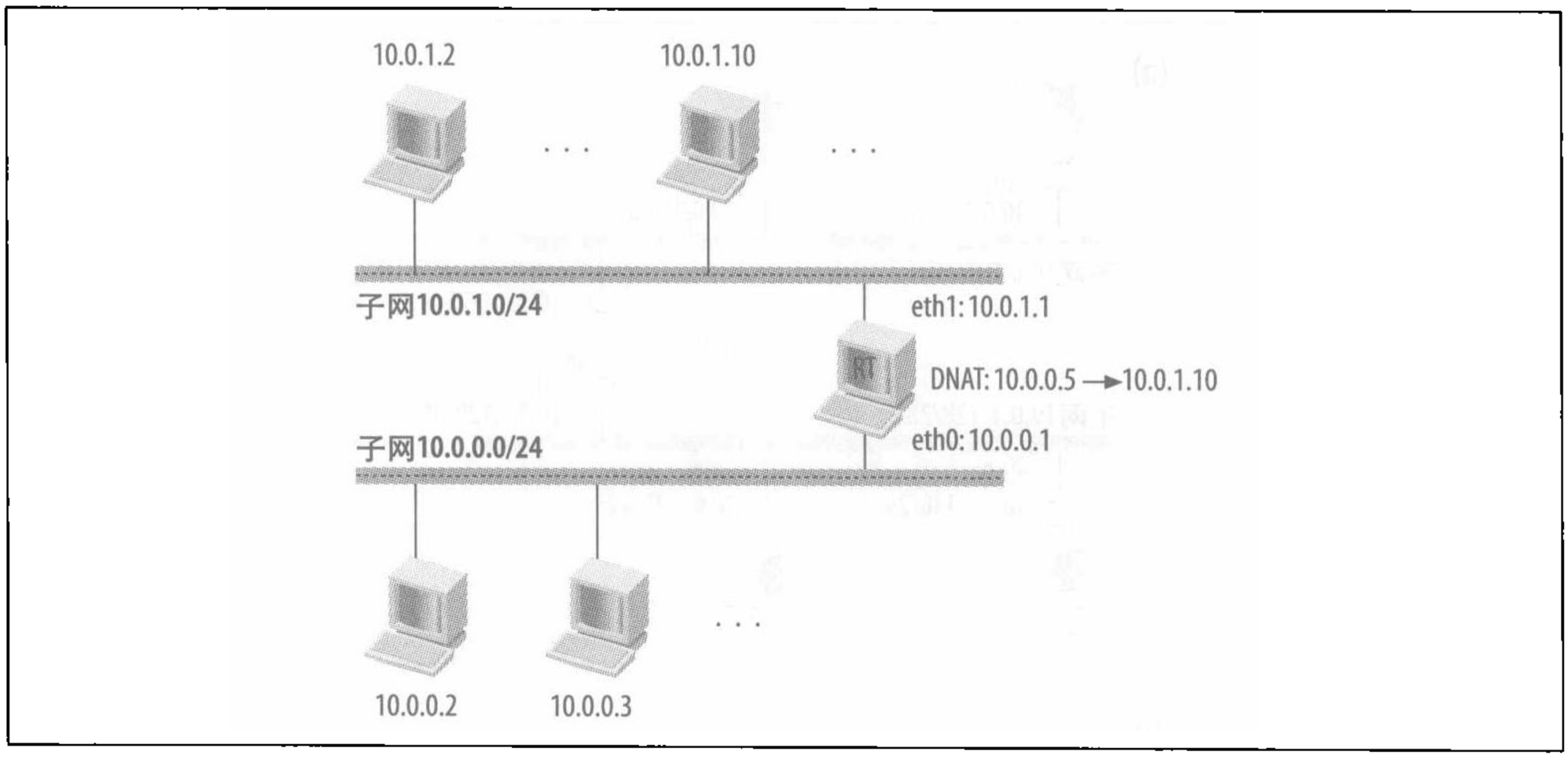


图 28-19：DNAT 的例子

10.0.0.5 的 ARP 请求。由于该 ARP 请求是发往 Ethernet 广播地址，所有 RT 就会收到，然后对其做出应答，也就是提供自己的 *eth0* 接口的 L2 地址。从此，RT 就会代理请求者和 10.0.1.10 之间的流量。

若是一台主机配置了一个伪NAT地址，就要创建一个特殊的路由项，并设立RTCF_DNAT标识，这样 ARP 就可以检查并代理该地址。

ARP 代理服务器作为路由器

代理和路由器都是对某功能的扩展，从这个角度上讲，它们看起来是相似的。实际上，IPv4 下路由器通常就是处理 ARP 代理的主机，IPv6 下只允许使用路由器（见 RFC 2461 文档）。但是，代理和路由在下面的问题上还是有区别的：ARP 代理服务器对其服务对象来说通常是透明的，而路由器并不是这样。每个主机需要明确配置来使用路由器。在大多数情况下，代理服务器担当着透明路由器的角色，其服务的主机位于不同的 LAN 中，但属于相同的 IP 子网，如图 28-20 所示。

图 28-20(a)给出了一个简单的拓扑结构，两个子网掩码都是 /25 的子网通过一个路由器进行通信。图28-20(b) 给出了同样的拓扑中通过代理而不是路由器连接两个子网，只是这两个子网的掩码改为 /24；这个变化增加了两个子网 10.0.1.0/25 和 10.0.1.128/25。图 28-20(c)给出是两个子网中的主机用图 28-20(b)的配置后的等价拓扑结构。

注意，图 28-20 中的例子没有建议在 ARP 代理服务器和路由器间优先选择哪个。这两种设备完成的任务是不同的：路由器将子网分为多个LAN，而 ARP 代理服务器将不同

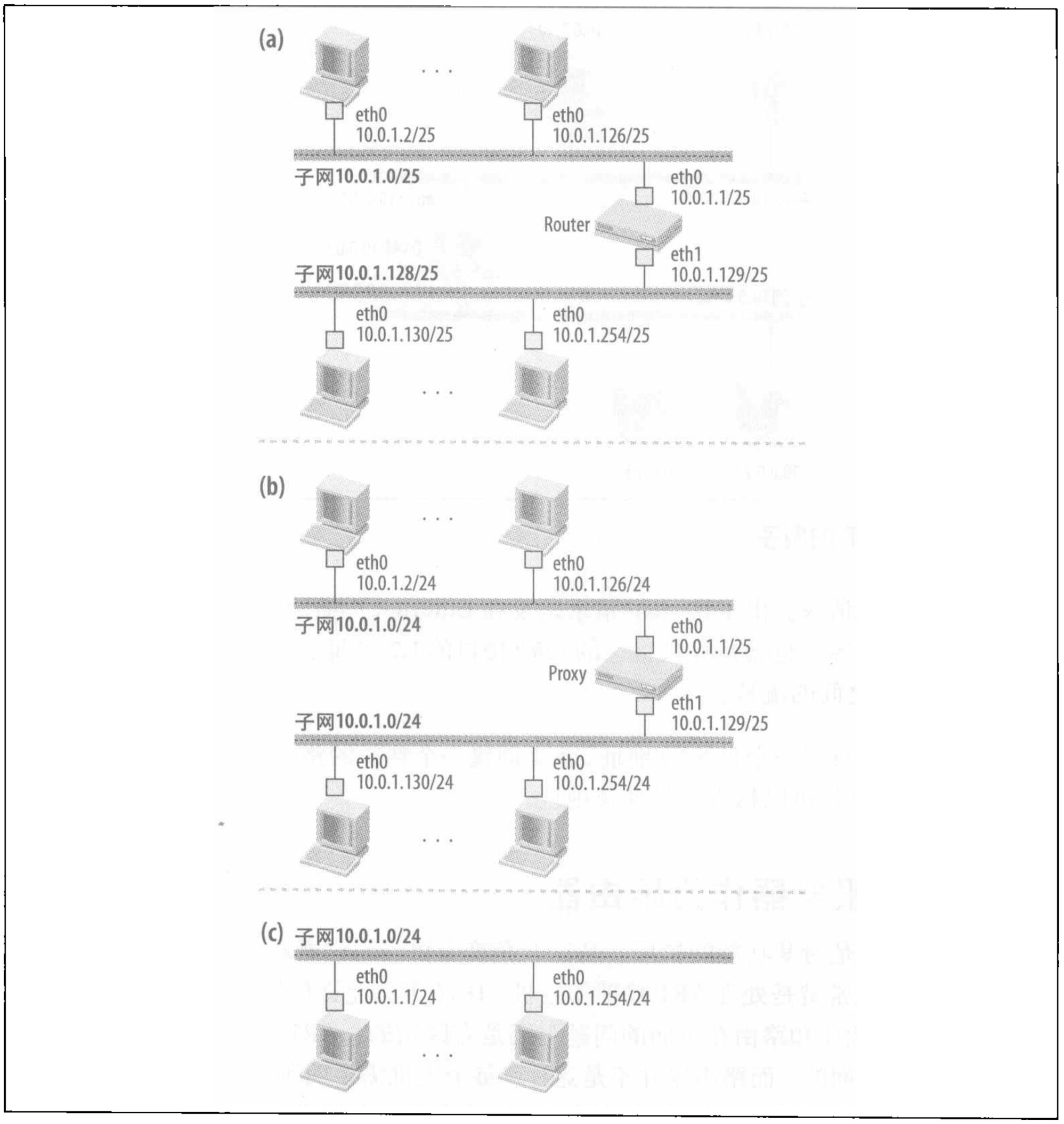


图 28-20：路由器和代理

的LAN 合并为单一子网。图中提供的例子仅给出了如何改变主机配置，才能使两个子网内的主机通过路由器或者 ARP 代理服务器通信。当然，你可以将所有主机都放在一个LAN中，这样就不需要任何路由器和ARP 代理服务器。但既然我们这里讨论APR 代理，我就需要提供一个使用它的例子。

有一个特殊情况要特别注意：一台ARP代理服务器可以配置成一台透明的默认网关。换句话说，管理员可以让LAN中的主机使用ARP代理服务器来访问默认路由，而不必在

每台主机上配置一条默认路由。若要实现这个功能，管理员要将主机子网掩码设为/0，在定义默认网关路由时也是使用同样的子网掩码。在这种方式下，ARP代理服务器会处理所有流向未知地址的网络流量，实际上就成了默认网关。ARP代理服务器甚至可以改变自己的地址，而不会对主机产生任何影响，只需更新主机缓存中所有陈旧的neighbour项即可（详细描述见“无端ARP”一节）。但是，这种看似聪明的方案不是很有效的，下面我会解释其原因。

包含一台ARP代理服务器的网络拓扑结构（类似图28-20（b）所示）要记录大量的solicitation请求和应答。当要代理的主机数量很多时，solicitations占用的网络带宽会相当大。

假定有个和图28-20（a）相似的网络，最坏的情况是/25子网包含了全部126台主机（主机数量占7位，减去默认的网络地址和广播地址），每台主机需要对其他每个主机的地址进行解析。这就会产生（126−1）×（126−1）个不同的solicitation请求。但是，像这样的最坏情况是很少出现的，因为一台主机通常只访问少数机器，例如，服务器。大部分主机间的流量要经过路由器，因此所有的主机都需要这个网关路由器的L2地址和L3地址。

如果我们保持网络拓扑结构不变，但将子网掩码由/24改为/0，最坏情况迅速爆发，平均情况开始接近最坏情况。任何时候，主机若想与其他的主机通信，不管后者是本地主机还是远端主机，总是会有一个单独的solicitation请求。该主机不是生成一个单一的向默认网关（用于访问路由器外的其他主机）的solicitation请求，而是必须为每个主机生成一个独立的solicitation请求，因为它没有任何路由器的信息。

总之，ARP代理服务器作为路由器使用可以简化主机和子网的配置，而且这些主机可以使用一个轻量级的TCP/IP栈，因为没有路由功能。但是，网络的负载和代理服务器上的CPU的占用将会是很高，因为有大量的solicitations封包。

实例

下面我们看图28-21所示的例子。

我们首先做如下假定：

- 所有主机都使用Ethernet接口卡。
- LAN1和LAN2中的所有主机配置为相同的子网掩码255.255.255.0 (/24)。在它们的路由表中没有任何路由，也没有配置默认网关。换句话说，在LAN1和LAN2中的主机只能和同在一个逻辑子网内的主机通信。

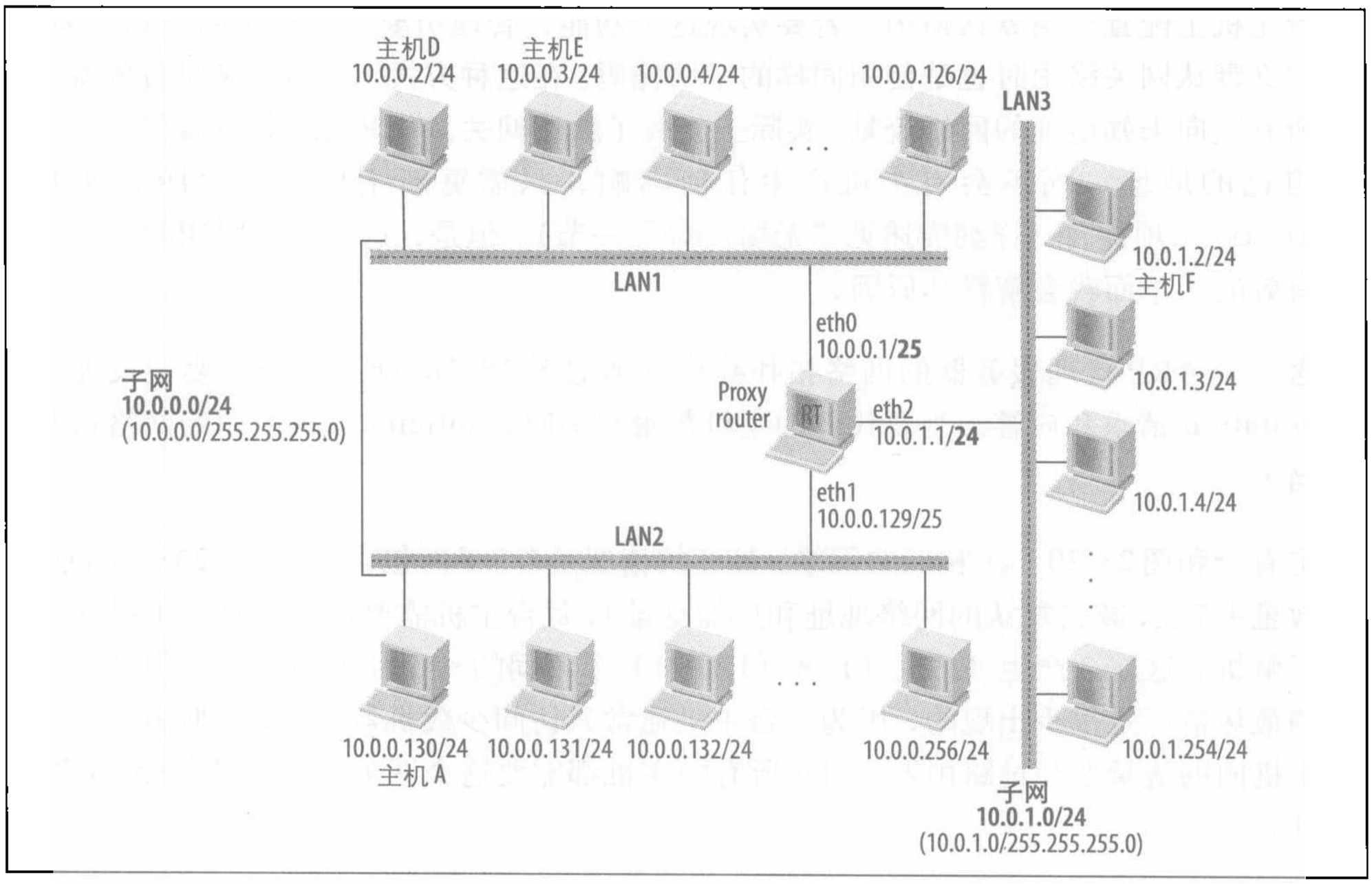


图 28-21：在 RT 上配置 ARP 代理的网络实例

- 所有邻居缓存都是空的，也就是说，任何主机都不知道其他主机的链路层地址。
- 网桥功能都关闭。这就排除了第二十六章中图26-10 中右上角所示的情况。如果你对这条假定的含义不是很明白，应该阅读第四部分。

注意，尽管LAN1 和LAN2 中的主机属于不同的LAN，但它们都可以有相同的逻辑子网（网络地址10.0.0.0/24，子网掩码255.255.255.0）配置。这种配置的意思就是，不同LAN中的所有主机共享同一个子网，并且不需要路由器的帮助就可以彼此通信。但是，就图中的网络拓扑结构来说，没有 RT 的帮助它们是无法交流的。

对于 LAN1 和 LAN2 中的主机来说，RT 给它们造成了属于同一子网的幻觉，因此 RT 需要其他的信息：它要合并的 LAN 的真实子网掩码应该是 255.255.255.128 或 /25。如果 RT 不是一台 ARP 代理服务器，那么 RT 或者 LAN1 和 LAN2 中的主机都属于配置错误（注 10）。

注 10： 如果我们排除了使用一些特殊功能，比如桥接，这种说法就是正确的。

RT为了让LAN1和LAN2中的主机可以透明地通信，它需要有关此网络拓扑结构的更多信息，以及要确切知道每台主机的位置。比较简单的解决方式是，RT将其在一个LAN中收到的任何封包，都转发到另一个LAN中，这种方式和代理无关。第四部分会涉及这种情况。对于来自LAN2的请求，如果RT想代表LAN1中的主机（例如，代替其应答），那么它就需要知道每个主机位于哪个LAN中。例如，RT不应对LAN1中产生的，且地址是LAN1中的其他主机的请求做出应答（因为LAN1中的主机已经属于同一个子网，10.0.0.0/24）。由于在RT的*eth1*和*eth2*接口上有正确的子网掩码，所以它就知道：

- 在*eth0*上，主机的地址范围是从10.0.0.1到10.0.0.126（10.0.0.127是广播地址，10.0.0.0是网络地址）。
- 在*eth1*上，主机的地址范围是从10.0.0.129到10.0.0.254（10.0.0.255是广播地址，10.0.0.128是网络地址）。

让我先提醒一下你，当从一个子网到另一个子网转发封包时需要路由器（也就是说，发送方和接收方不在用一个子网内）。注意，图中的RT路由器如果没有启动ARP代理功能，那么它的两块能到达LAN1和LAN2的NIC是配置错误。

现在开始分析一些ARP请求的常见情况。对期望的行为，你可以参见第二十六章的图26-9和图26-10：

- （a）从LAN1到LAN1（例如，从主机D到主机E）

  由于主机D（10.0.0.2）是和主机E在同一个子网内（10.0.0.0/24），它就可以发出一个对IP地址10.0.0.3的solicitation请求（ARPOP_REQUEST）。在LAN1中的所有主机都会收到这个请求，但只有主机E会向主机D应答，表明自己的L2地址。注意，尽管在*eth0*上已启动了代理功能，但RT不会应答。原因是RT在*eth0*（LAN1）上收到solicitation封包，但它知道10.0.0.3位于发送方的子网内，因此不需要拦截该请求：主机E所处的网络，正是solicitation请求生成的网络，因此主机E自己可以应答。

- （b）从LAN1到LAN1中的非法IP地址（例如，从主机D到10.0.0.128）

  从A2的角度看，10.0.0.128是一个合法的主机地址；从RT的角度看，它不是（是一个网络地址）。没有人会做出应答这个请求，不管RT是否是代理。

  这里不好理解的是即使LAN1和LAN2中的主机配置有相同的子网掩码10.0.0.0/24，但它们仍然按照RT的设置物理上分为两个区域。RT不会应答这种请求，因为它认为10.0.0.128是个网络地址。

- （c）从LAN1到LAN2（例如，从主机D到主机A）

由于10.0.0.130所属的主机位于另一个LAN中，它不会收到这个请求并应答。但是，由于RT在*eth0*上配置了代理功能，它会用*eth0*接口的地址应答。这表示当主机D要往主机A发送数据时，它实际上是发到了RT，然后RT再转发到主机A。如果主机A请求主机D的地址，就是相反的过程。

- (d) 从LAN1到LAN3（例如，从主机D到主机F）

  由于主机F和主机D不在同一个子网内（10.0.1.2不属于子网10.0.0.0/24），且在主机D中没有定义访问LAN3（10.0.1.0/24）的路由，于是其内核的IP层会用一个消息作为应答，该消息表明没有到达主机F的可用路由。主机D甚至不会产生solicitation请求（注11）。

外部事件

当满足一定条件时，ARP就可以接收和生成通知（notifications）。第二十七章的“邻居协议和L3传输函数的交互”一节中，曾简要介绍了邻居协议如何与内核的其他部分进行交互。下面我们看具体到ARP协议是如何管理这些通知。

收到的事件

在“初始化ARP协议”一节已看到，ARP为了能得到设备事件的通知，要向内核注册一个函数。这个函数就是arp_netdev_event，它负责处理这些事件。在该函数接收到的各种事件类型中，ARP只对NETDEV_CHANGEADDR类型的事件感兴趣。当一个设备的L2地址发生变化时（例如，手工修改地址），内核就会产生该事件。内核中处理用户空间请求改变设备链路层地址的函数是do_setlink，也是它发出NETDEV_CHANGEADDR通知，该函数定义在*net/core/rtnetlink.c*文件中。

```
static int
arp_netdev_event(struct notifier_block *this, unsigned long event, void
*ptr)
{
    struct net_device *dev = ptr;

    switch (event) {
    case NETDEV_CHANGEADDR:
        neigh_changeaddr(&arp_tbl, dev);
        rt_cache_flush(0);
        break;
    default:
```

注11：注意，即使主机F物理上位于LANI中，但仍然有LAN3的地址，上述结论也是成立的[大多数情况下这属于错误的配置，如第二十六章的图26-1（c）所示]。

```
            break;
        }

        return NOTIFY_DONE;
    }
```

在第二十七章的“邻居层接收的事件”一节中，有对neigh_changeaddr函数的介绍。

为了强制让IP层使用新的L2地址，rt_cache_flush函数会刷新IPv4的路由缓存。该函数不是有选择地删除与产生通知的设备关联的缓存项，而是完全清除整个缓存。第三十三章会介绍该函数输入参数的含义，以及路由缓存的细节。

产生的事件

在第二十七章的“邻居层产生的事件”中提到了虚函数error_report，它是neigh_ops结构的一部分。在ARP中，该函数由arp_error_report实现。当一个ARP事务失败时，ARP子系统就调用该函数。它的两个主要任务是：

- 从路由表缓存中（注12）删除与不可到达的邻居相关的缓存项。
- 用一个ICMP UNREACHABLE消息通知发送方邻居不可到达。

网络唤醒（Wake-on-LAN）事件

一些高级的NIC设备能提供称为网络唤醒（Wake-on-LAN，WOL）的功能。

在第六章中做过简单介绍，WOL通过给一个NIC发送一个特殊类型的帧，就能使它唤醒一个处于待机模式的系统。在各种帧的类型中，有唤醒功能的只有ARP帧。该功能是在硬件级实现的，因为当系统还在待机模式，CPU上没有运行能处理输入封包的设备驱动程序。带有WOL功能的NIC需要有其自己的电源输入，以便能够扫描到这些特殊的帧。我不会深入描述这个功能，因为它完全由NIC的驱动程序处理，而不是由ARP模块处理。详细的代码可以用WAKE_ARP关键字搜索得到。

ARPD

在一个网络中，邻居的数量可能会是几个，也会是上千个。因此，在一个大型网络中，neighbour数据结构需要的内存也会相当大，并且会影响到系统的性能。增大

注12：确切地说，是从与协议无关的缓存中删除相应的项，详情情况可以参见第三十三章的内容。

neigh_table结构中的配置参数gc_threshn的值只会简单改变可以创建的邻居项的最大数目，而不会解决过量消耗内核内存带来的性能问题。

*Arpd*是一个用户空间守护进程，可以将ARP负载从内核中转移到它自己的（更大些）缓存中。ARP用户空间的实现的执行速度不可能和内核空间实现的速度一样快，但在大多数情况下，这种速度差别是可以接受的。

要使用*arpd*，内核编译时必须要支持APRD功能。虽然内核文档中称ARPD是一个试验性特征，但它实际上已经应用了很长时间。

现有两个*arpd*守护程序可以从网上下载得到。一个是旧版本，它根本无法正确运行；另一个是IPROUTE2程序包的一部分，可以运行。本节中我主要谈第二个。

Arpd守护进程负责拦截来自其他系统的ARP请求，并且维护自己的数据库，以代替内核缓存。本章中我们不过多讨论该程序的内部结构，我们的重点在该程序和内核的交互上。*arpd*自己维护与网络的联系。内核也可以继续处理ARP请求，并且负责通知*arpd*自己知道的一些事件。它们两者之间的通信使用Netlink套接字，2.6版的内核中默认支持Netlink。

图28-22中给出了邻居子系统中ARP和*arpd*之间的交互关系。本质上，邻居子系统给守护进程发出通知，守护程序会监听这些通知。下面两个小节将详细介绍这种交互过程。

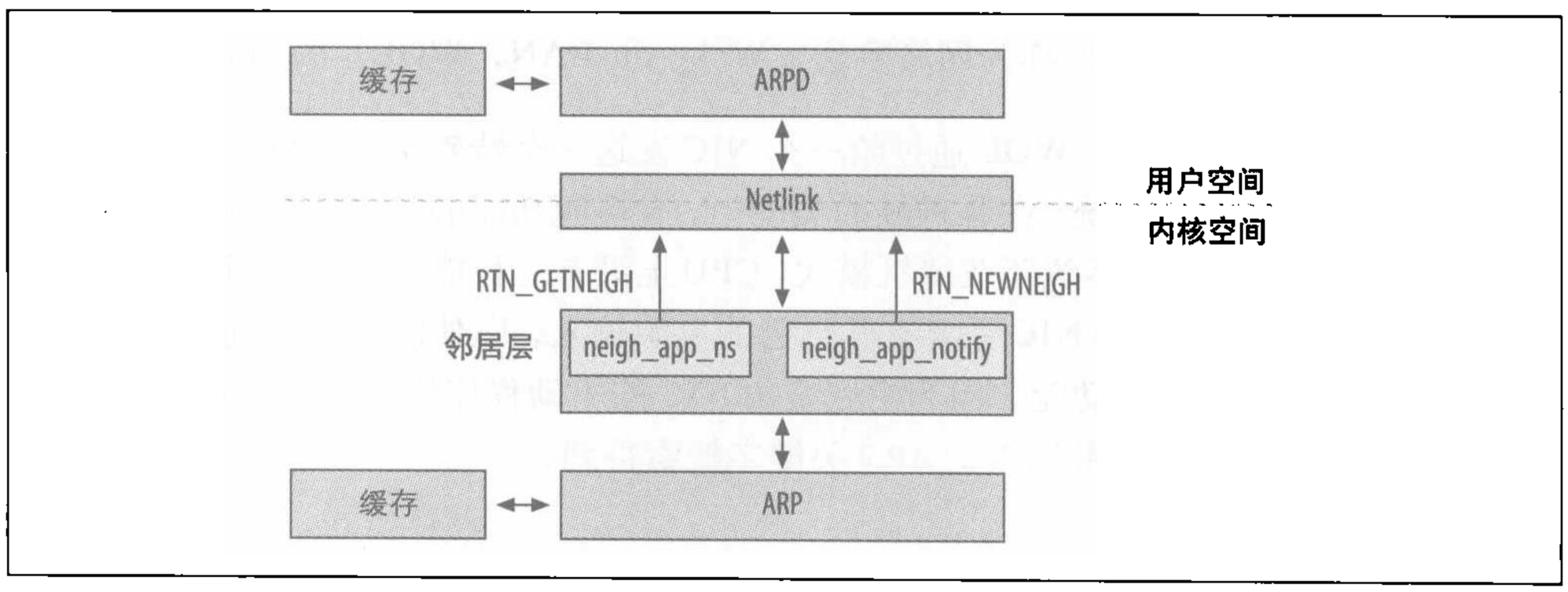


图28-22：ARP和arpd守护程序间的交互

内核方面

当启动ARPD功能后，邻居子系统就会给用户空间的守护进程发送消息。下面我们回顾一下用于发送这些消息的相关函数，以及调用这些函数的条件：

neigh_apps_ns

当内核发出的solicitations封包的数目（地址探测的次数）达到内核允许的最大值，而用户空间生成的solicitations封包的数目还未达到用户空间允许的最大值时，来自协议的solicit函数（arp_solicit）就调用这个函数。使用*arpd*程序的规则是内核在调用该守护进程前，必须用完对某一邻居进行地址探测的尝试次数。但是，管理员可以通过配置APRD，使内核根本不产生任何探测，立即调用*arpd*。

neigh_app_ns生成的消息类型是RTM_GETNEIGH。

neigh_app_notify

这个函数用于发出送下面两种通知：

- 一个neighbour已被移到NUD_FAILED态，不久之后就会被垃圾回收器删除。这种情况下，这个状态的变化和对neigh_app_notify的调用是由neigh_periodic_timer（见第二十七章）处理的。
- 一个邻居的状态从一个有效态（派生态NUD_VALID）变为一个无效态，或者邻居的L2地址发生变化时，由neigh_update函数来处理状态的变化和调用neigh_app_notify。

neigh_app_notify函数生成的消息类型是RTM_NEWNEIGH。

用户空间方面

从前面一节我们知道了什么时候内核会向*arpd*发出通知。现在我们来看*arpd*如何处理通知。下面是这个守护进程结构框架（main函数）：

```
1. Parse command-line options
2. Open database
3. Load database from file if option present
    (3.1) Open socket for reception and transmission of ARP packets
    (3.2) Open socket with kernel for ARPD notifications
4. Infinite loop
    (4.1) Poll the two sockets
    (4.2) If events appear on socket (1), process input ARP packet
    (4.3) If events appear on socket (2), process input kernel message
```

图28-23中给出了上述行为的一个简单模型（图28-23(a)代表4-2，28-23(b)代表4-3）。图中也清楚地给出了上一节中介绍的内核的相应行为。

守护进程有一些命令行选项，用以调整其动作。例如，管理员可以通过这些选项指定：

- 在发出多少个探测封包后，放弃探测该地址。
- 内核是否应该发出探测封包，还是只有守护程序可以产生探测封包。

- 从一个文件中加载邻居项到缓存。

当前的*arpd*守护进程的ARP缓存实现是用通用的Berkeley DB数据库，这也是为何管理员在安装IPROUTE2包时，要解决Berkeley DB包依赖性的原因。

这里需要提一下*arpd*程序和内核的ARP子系统之间的一个区别：*arpd*缓存和内核的ARP缓存不同，*arpd*缓存中保存否定的结果（译者注：失败的解析）。当解析一个地址的尝试失败时，*arpd*守护程序就将其信息保存在它的缓存中，并且在一定时间内不在试图对其进行解析。

逆向地址解析协议（RARP）

RARP是一个旧协议，用于自动配置一台动态主机。它的功能后来被*bootp*和DHCP替代。尽管RARP的目的和ARP的不同，但RARP也使用ARP包（ARPOP_REQUEST和ARPOP_REPLY中的操作码不同），并使用相同的传输函数arp_send。Linux内核默认并不包括RARP协议，在编译内核是必须要明确指定才支持该协议。

ND（IPv6）对ARP（IPv4）的改进

第二十六章已经提到过，IPv6邻居协议ND和ARP是完全不同的设计。下面列出了ND中的一些改进：

- ND是ICMPv6提供的一个函数。ICMPv6是一个强大的覆盖了ICMPv4、ARP和其他功能的协议。如同在第二十六章的“邻居协议”一节中讲的，讲ND集成在ICMP中使得ND可以充分利用任何L3的特性，特别是加密功能。
- ND使用多播solicitations而不是广播。使用的多播地址来源于要请求的目的地址，表示只有注册到给定的IP多播地址的主机才接收相应的solicitation请求。在一个大型网络中，这种方式可以显著减少主机接收和丢弃的solicitations数量，丢弃的原因是主机不是收到的solicitations的目的地。
- ND使用邻居不可到达检测算法来检测停用的邻居。并不是每个ARP实现都使用该算法，但我们在第二十七章已经看到，Linux的ARP实现使用了该算法。

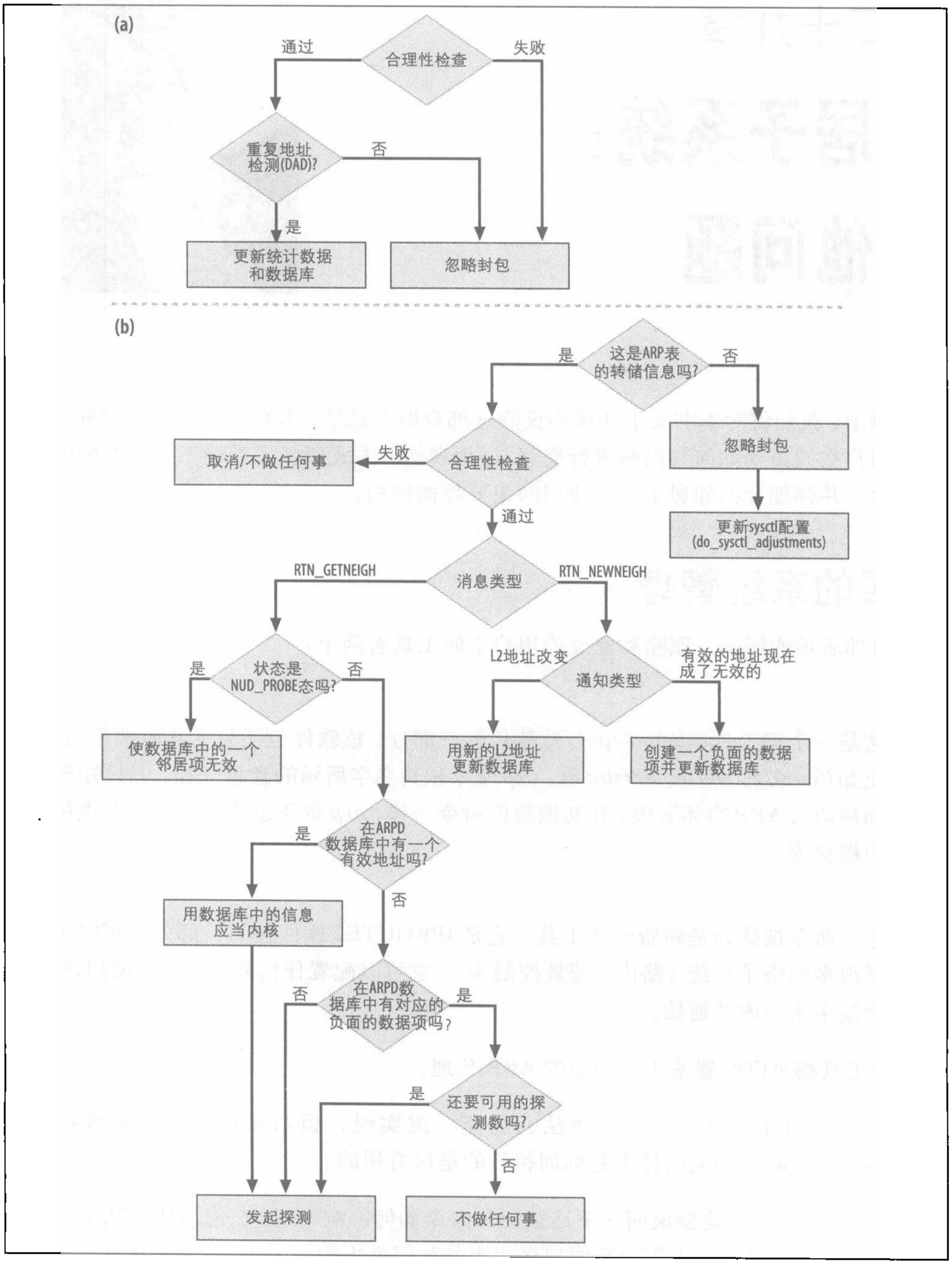


图 28-23：(a) 处理 ARP 封包；(b) 处理内核消息

第二十九章

邻居子系统：其他问题

这一章中，我们将对本书关于邻居协议的这部分做个总结。本章会给出用于配置邻居协议的用户空间命令如何与内核进行交互，还用表格的方式概括了前面三章中介绍的函数和变量，并详细介绍邻居子系统使用的主要数据结构。

邻居的系统管理

可以对邻居项的增加、删除和修改的用户空间工具有两个：

arp

这是一个旧工具，是*net-tools*软件包的一部分。该软件包还包括其他常用的命令，比如*ifconfig*、*route*、*netstat*等。*arp*命令和其名字所示的意思一样，它只处理IPv4邻居协议ARP的邻居项。和其他类似命令一样，*arp*命令也是通过`ioctl`调用来和内核交流。

ip

这个命令被认为是新版管理工具。它是IPROUTE2软件包的一部分，可以用于配置很多网络子系统（路由、流量控制等）。它可以配置任何邻居协议，使用Netlink套接字来与内核通信。

这两个工具都可以配置基于目的地的ARP 代理。

本章不会详细介绍这些命令的语法、功能、或实现，但当使用这些命令维护一个`neighbour`项时，知道内核中是如何执行的是很有用的。

下面的三个小节，会简要说明一下这些配置命令如何影响内核。在使用IPROUTE2包的情况中，我会简单说一下用户空间程序代码是如何组织的。

命令函数

尽管*ip*和*arp*使用不同的机制和内核交互，在下面两节我们会看到，虽然使用不同的内核处理函数处理配置命令，但最后它们都要通过相同一组函数与邻居层对话：

查找函数

在改变一个已有的邻居项或新增一个邻居项前，内核需要在缓存中进行一次查找。这些查询工作使用第二十七章“缓存”一节介绍的一些函数。

`neigh_update`

`neigh_update`是个通用函数，根据其输入参数，可以完成各种不同的操作。该函数的介绍在第二十七章的“邻居信息的更新：neigh_update”一节中。

`pneigh_update`

`pneigh_update`是按目的地址代理中用于代替`neigh_update`的函数。见第二十七章的“担任代理”一节。

在必要的时候，查询函数会创建或删除`neighbour`项，所用的`neigh_add`和`neigh_destroy`函数在第二十七章中有介绍。

图29-1总结了本节内容和前面几章内容的联系。

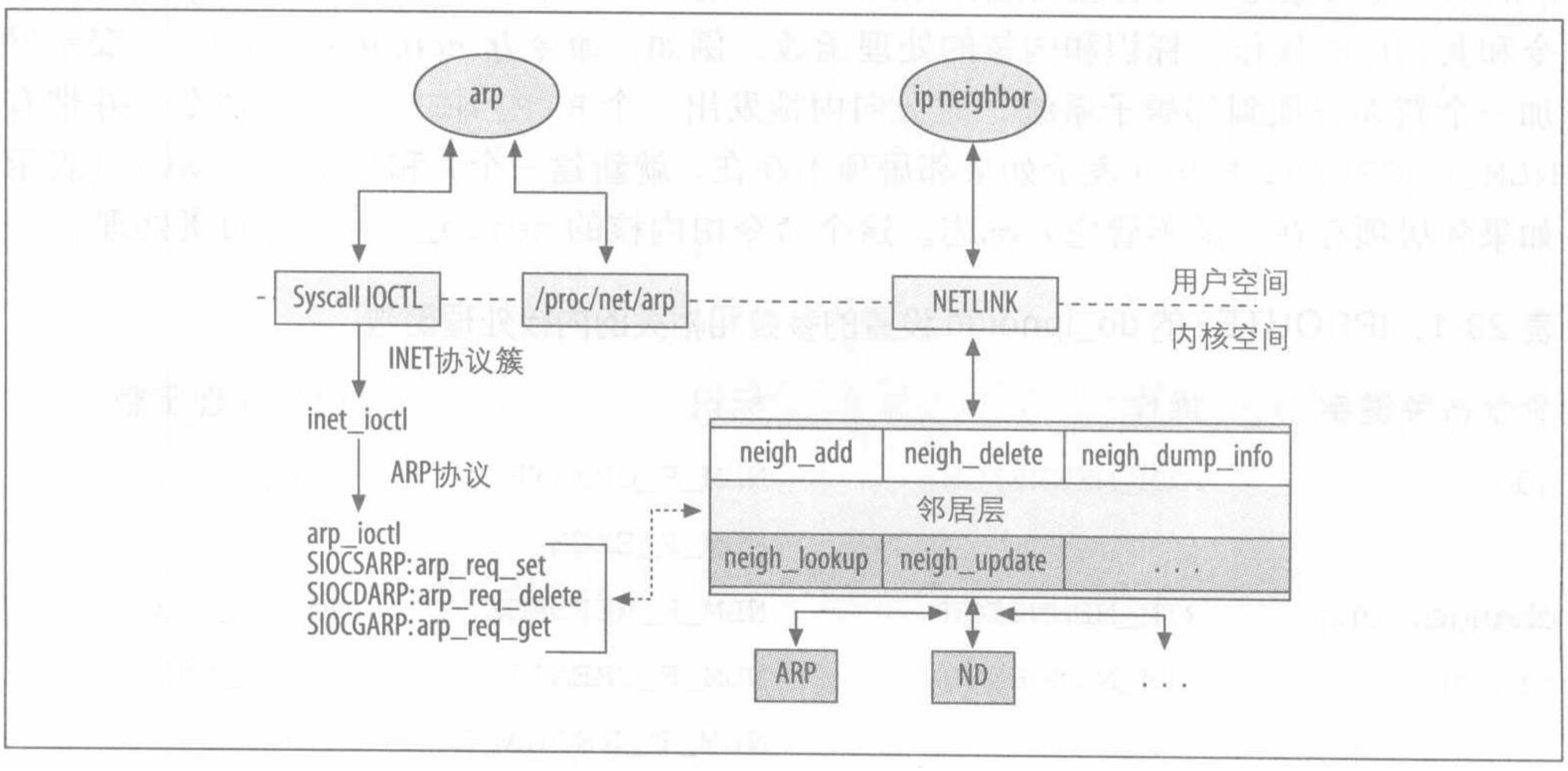


图29-1：arp命令和ip neighbour命令在用户空间和内核空间的接口

新一代工具：IPROUTE2包中的ip命令

*ip*是个通用命令，用于代替一些传统Unix命令，例如*ifconfig*、*route*和*arp*。*ip*命令的

第一个参数可以是 *address*、*route* 或 *neighbour* 等，表示该命令要操作的对象，这样它就可以执行 *ifconfig*、*route*、*arp* 等的工作。对内核来说，*ip* 操作的对象就决定了该命令要与哪个子系统交互。

用于配置邻居协议的命令是以 *ip neighbour* 开头。图 29-2 给出了 IPROUTE2 软件包中实现邻居协议配置的主要文件和函数。

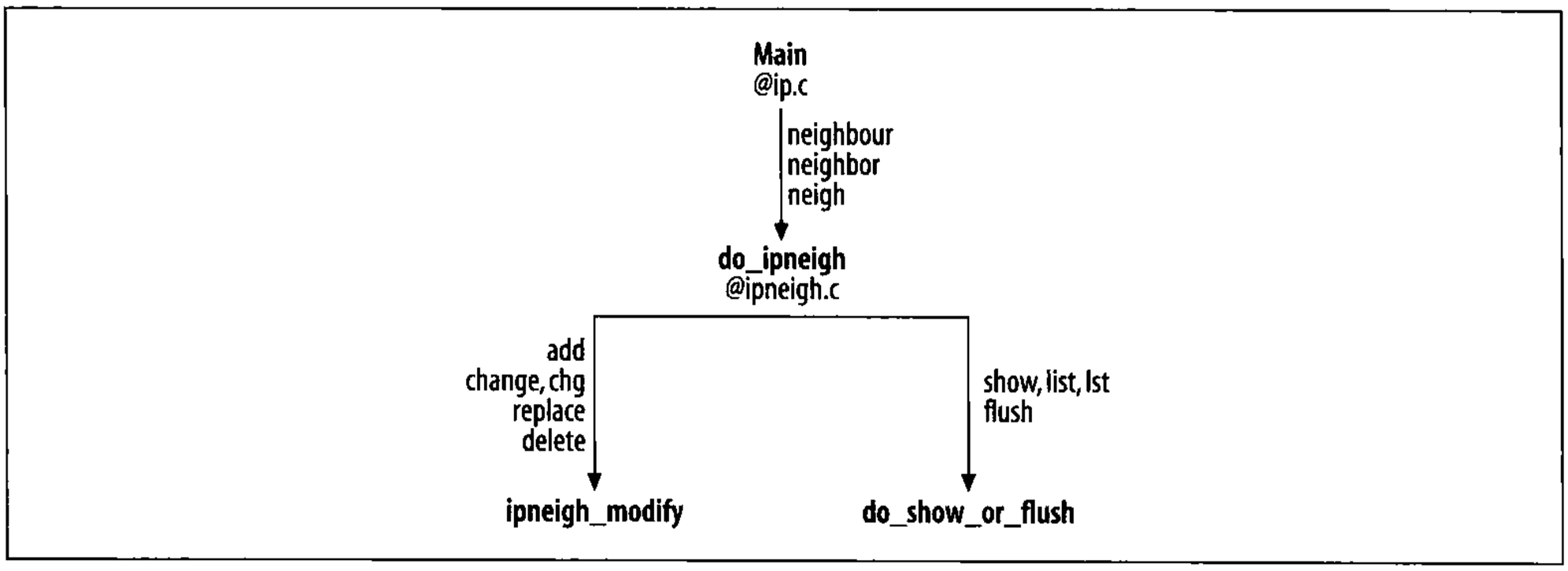


图 29-2：IPROUTE2 软件包中有关邻居的文件和函数的结构

*ip*的第二个参数是表示管理员想让相应子系统做什么工作的命令。表29-1概括了这些命令和其相应的操作、标识和内核的处理函数。例如，命令 *ip neighbour add*…，表示增加一个新邻居项到邻居子系统，它会向内核发出一个 RTM_NEWNEIGH 命令，并带有 NLM_F_CREATE 标识（表示如果邻居项不存在，就新建一个）和 NLM_F_EXCL（表示如果邻居项存在，就不管它）标志。这个命令由内核的 neigh_add 函数负责处理。

表 29-1：IPROUTE2 的 do_ipneigh 设置的参数和相关的内核处理函数

命令行关键字	操作	标识	内核处理函数
add	RTM_NEWNEIGH	NLM_F_CREATE	neigh_add
		NLM_F_EXCL	
change, chg	RTM_NEWNEIGH	NLM_F_REPLACE	neigh_add
replace	RTM_NEWNEIGH	NLM_F_CREATE	neigh_add
		NLM_F_REPLACE	
delete	RTM_DELNEIGH	None	neigh_delete
show, list, lst	RTM_GETNEIGH	NLM_F	neigh_dump_info
flush	RTM_GETNEIGH	NLM_F	neigh_dump_info

如果你看一眼上面列出的内核处理函数中的一个，例如，neigh_add，再根据表29-1，你应该能够确定每个函数要完成什么工作。当然，还必须具有少量的Netlink层知识，例如，如何对输入的参数进行语法分析。在第三章有介绍Netlink的内容；但是由于篇幅所限没有涉及其内部结构。

上一代工具：net-tools包中的arp命令

对于更愿意使用旧Unix命令而不用IPROUTE2程序的人来说，很少会用到*arp*命令来手工维护一台主机的ARP列表（当然，该命令对其他的邻居协议不支持）。表29-2列出了主要的*arp*命令和处理这些命令的内核处理函数。表中也给出了完成同样功能的*ip neigh*命令。注意到，对*ip neigh change*命令或*ip neigh_replace*命令没有对应的*arp*命令（可以用一个删除命令和紧接着的添加命令代替）。

表29-2：arp命令、对应的ip命令和调用的内核函数

net-tools	用户空间命令IPROUTE2	net-tool调用的内核函数
arp -s ...	ip neigh add ...	arp_req_set
arp -d ...	ip neigh del ...	arp_req_delete
arp	ip neigh show ...	*/proc/net/arp file*

arp_req_xxx函数的定义在*net/ipv4/arp.c*文件中。在该文件中，你还能发现维护虚拟的*/proc/net/arp*文件的函数。*arp*为了获取有关信息，要读取这个文件，而不是发出ioctl调用，尽管内核也提供了一个用于处理这类调用请求的函数arp_req_get。参考*net/ipv4/arp.c*文件中关于arp_seq_ops结构的定义，可以找到更多关于使用*/proc*文件的内容。

使用/proc文件系统调整邻居性能

在前面几章我们已经知道，邻居协议按照内核通常的惯例，在*/proc*目录中提供一个方便的接口，以便让管理员可以调整邻居子系统的参数。邻居子系统的参数保留在4个目录中，其中两个给IPv4，两个给IPv6。

/proc/sys/net/ipv4/neigh

/proc/sys/net/ipv6/neigh

里面存放着邻居子系统的普通参数，例如，用于控制什么时候发生缓存操作的定时器。

/proc/sys/net/ipv4/conf
/proc/sys/net/ipv6/conf

保存邻居协议的一些特殊行为，例如，第二十八章“调整 ARP 选项”一节中讲过的一些行为。

上面每个目录中包含多个设备子目录（每个子目录对应系统中的一个NIC 设备）、一个*default*子目录和一个*all*子目录。*all*子目录用于将一个变化同时应用到所有设备上。在*conf*目录中，*default*子目录中给出了邻居协议每个特性的全局状态，而*neigh*中的*default*子目录则给出了每个特性的默认设置（也就是配置参数）。每当创建一个设备子目录时，就用 *default* 子目录中的值初始化这个目录。

每个设备对应的目录中的参数优先于普通目录中的参数。但并不是所有设备会使用目录中的全部参数；如果某个参数和一个设备无关，虽然对应于该设备的目录中有包含该参数的文件，但内核不会使用。例如，任何协议都不会使用参数 `gc_thresh1`，以及只有 IPv4 使用 `locktime` 参数。

图 29-3 给出了文件和要向内核注册的函数的结构安排。

在图 29-3 右上角的三个文件 *arp*、*arp_cache* 和 *ndisc_cache* 不用于配置任何参数，只用于输出只读数据。这三个文件位于 */proc/net* 目录，而不是 */proc/sys* 目录。*arp* 命令使用文件 */proc/net/arp* 转储 ARP 缓存的内容（ND 协议没有对应的文件），参见“上一代工具：net-tools 包中的 *arp* 命令”一节。*/proc/net/stat/xxx_cache* 文件中保存着对应协议缓存的统计数据。大部分这些文件都对应于`neigh_statistics`结构的一些字段，详细介绍参见“neigh_statistics 结构”一节。

/proc/sys/net/ipv4/neigh 目录

这个目录中包含来自`neigh_parms`结构的参数，在第二十七章中介绍过这个结构。那一章还提到每个设备为每个和它交互的邻居协议保持一个`neigh_parms`结构(见第二十七章的图 27-2)。我们也已看到在`neigh_table`结构中包含一个 `neigh_parms` 结构的实例，用于存储参数的默认值。

但是，并不是 `neigh_parms` 结构中所有的字段都导出到 */proc* 目录。例如，导出字段`reachable_time`的参数值由`base_reachable_time`变量间接计算得到，因此不能被用户改变，也就不会保存在*/proc*目录中。此外，内核用参数`tbl`和`neigh_setup`组织其数据结构，它们与协议没有任何联系，因此也不会被导出。

邻居子系统除了会将`neigh_parms`结构中的大部分参数输出到*/proc*目录外，还会输出`neigh_table`结构的一些字段。

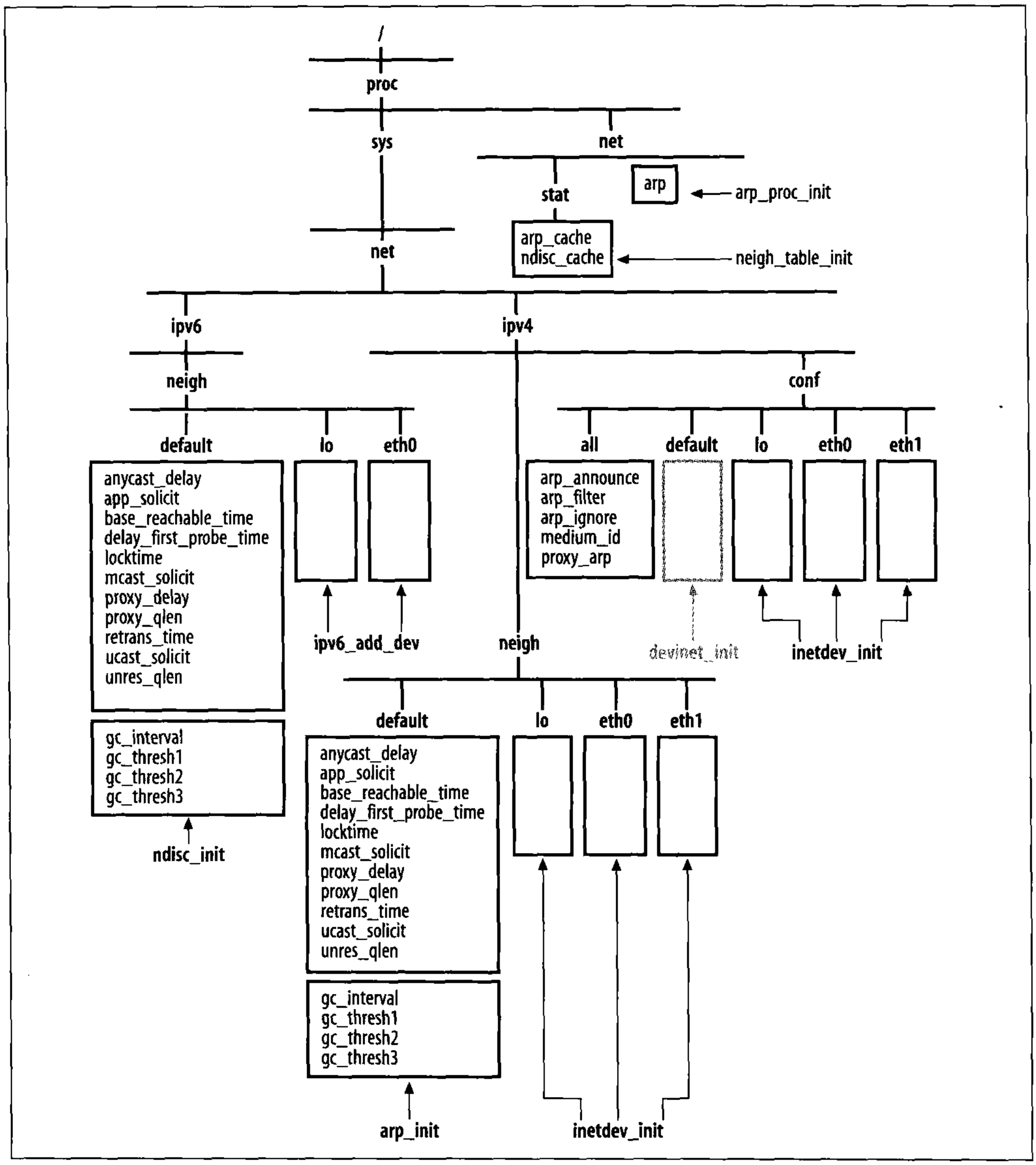


图 29-3：邻居子系统注册的 /proc/sys 文件例子

global 目录和设备目录的初始化

由于默认值是协议自己提供的，所以在协议初始化的时候（见 `arp_init` 函数和 `ndisc_init` 函数），就建立 *default* 子目录，并且根据 `neigh_parms` 结构中的相关字段建立对

应的文件。你能从表 29-3 中找到 xxx_tbl 表初始化过程中，这些参数的默认值；第二十八章中则给出了一个 ARP 的例子。

内核变量和 */proc/sys/net/ipv4/neigh/xxx/* 下文件名称之间的关系见表 29-3。参见 *net/core/neighbour.c* 文件中 neigh_sysctl_template 的初始化过程，也可以参考第三章介绍的一些模板。

表 29-3：/proc/sys/net/ipv4/neigh 子目录中的内核变量和相关文件

内核变量名	文件名	IPv4/IPv6 的默认值
mcast_probes	*mcast_solicit*	3
ucast_probes	*ucast_solicit*	3
app_probes	*app_solicit*	0
retrans_time	*retrans_time*	100 * HZ
base_reachable_time	*base_reachable_time*	30 * HZ
delay_probe_time	*delay_first_probe_time*	5 * HZ
gc_staletime	*gc_stale_time*	60 * HZ
queue_len	*unres_qlen*	3
proxy_qlen	*proxy_qlen*	64
anycast_delay	*anycast_delay*	1 * HZ
proxy_delay	*proxy_delay*	(8*HZ)/10
locktime	*locktime*	1 * HZ
gc_interval	*gc_interval*	30 * HZ
gc_thresh1	*gc_thresh1*	128
gc_thresh2	*gc_thresh2*	512
gc_thresh3	*gc_thresh3*	1,024

每当首次配置一个设备时，就会为其建立一个子目录。比如，第一次给设备D配置一个地址时，那么在 */proc/sys/net/ipv4/neigh* 目录下就会创建一个名为D的目录。由于所有的参数都是应用到一个设备上，而不是应用到一个特定的地址，因此每个设备只有一个目录，尽管它可以配置多个地址。

从图 29-3 给出的目录树，我们可以看到，图中的主机有三个设备：*eth0*、*eth1* 和 *eth2*；*eth0* 和 *eth1* 设置 IPv4 地址；*eth0* 还设置了一个 IPv6 地址；*eth2* 还没有配置。

用于配置IPv4和IPv6设备的函数分别是 inetdev_init 和 ip6_add_dev。在下面的一

节我们会知道，每次对 neigh_sysctl_register 的调用都会在 */proc* 下建立设备的子目录。

建立目录

在 */proc/sys/net/ipv4/neigh* 中的 *default* 子目录和设备的子目录都是用 neigh_sysctl_register函数创建的。该函数会依据其输入参数dev值的不同建立不同的目录。以IPv4为例，你可以比较一下 arp_init（协议初始化函数）和 inetdev_init（设备的配置块初始化函数）调用neigh_sysctl_register方式的区别。neigh_sysctl_register需要区分上述两种情况：

- 选择要建立的目录的名称。如果参数 dev 是 NULL，则建立的是 *default* 目录；否则目录名取自对应的设备（dev->name）。
- 决定哪些参数可以作为文件添加到目录中；*default* 目录下包括的参数比其他目录中的多（确切地讲，是多四个）。从 neigh_parms 中提取的参数，在根据每个设备进行配置时都是有含义的，但是从 neigh_table 得到的参数并不是这样。于是，来自 neigh_table 中的四个参数只会放到 *default* 目录中（见表 29-3 的结尾）。这四个参数与垃圾回收处理有关：

  _gc_interval

  _gc_thresh1, gc_thresh2, gc_thresh3

下面列出了函数 neigh_sysctl_register 输入参数的含义：

struct net_device *dev

与要建立的目录有关的设备。如果 dev 是 NULL，就表示调用这个函数是来创建 *default* 目录。

struct neigh_parms *p

需要将其参数导出的数据结构。例如，对于使用 ARP 的设备，就是 in_dev->arp_parms。若 dev 是 NULL，这个参数就嵌入到邻居的 neigh_table 结构的 neigh_parms实例(neigh_table->neigh_parms)，里面保存有协议的默认参数值。

int p_id

协议标识符。参考 *include/linux/sysctl.h* 文件中的 NET_*XXX* 值。对于 ARP 来说，该参数就是 NET_IPV4。

int pdev_id

要导出的参数的类别标识符。参考 *include/linux/sysctl.h* 文件中的 NET_IPV4_*XXX* 值。对于 ARP 来说，该参数就是 NET_IPV4_NEIGH。

char *p_name

一个字符串，表示与该邻居协议有关的L3 协议。对于ARP来说，参数值为“ipv4”。

proc_handler *handler

当用户要修改某个导出的字段的值时，内核要调用的函数。只有IPv6下，该值才不是NULL，它提供函数是对内核在其他情况下应该安装的默认处理函数的简单包裹。参见 *net/ipv6/ndisc.c* 文件中的 ndisc_ifinfo_sysctl_change 函数作为例子。

neigh_sysctl_register 函数中最复杂的部分是如何将四个 gc_*xxx* 参数从 neigh_table 结构中提取出来。实现过程巧妙地利用了neigh_table结构在内存中的安排：与垃圾回收相关的四个参数保存在neigh_table 结构中，且位于neigh_parms 结构之后，如下所示：

```
struct neigh_table
        ...
        struct neigh_parms parms;
        int gc_interval;
        int gc_thresh1;
        int gc_thresh2;
        int gc_thresh3;
        ...
```

这样，所有需要获得neigh_table中值的函数首先要越过neigh_parms结构，然后将指针设为整数格式，之后就可以连续取得这四个整数值：

```
    if (dev) {
        dev_name_source = dev->name;
        t->neigh_dev[0].ctl_name = dev->ifindex;
        memset(&t->neigh_vars[12], 0, sizeof(ctl_table));
    } else {
        t->neigh_vars[12].data = (int *)(p + 1);
        t->neigh_vars[13].data = (int *)(p + 1) + 1;
        t->neigh_vars[14].data = (int *)(p + 1) + 2;
        t->neigh_vars[15].data = (int *)(p + 1) + 3;
}
```

/proc/sys/net/ipv4/conf 目录

*/proc/sys/net/ipv4/conf*子目录中的文件与ipv4_devconf 结构中的一些字段有关，这个结构在*include/linux/inetdevice.h*文件中定义。邻居协议不会使用它的所有字段（见第二十三章和三十六章）。表29-4 列出了与邻居协议相关的参数，这些参数的含义见第二十八章的“调整 ARP 选项”一节。

表 29-4：/proc/sys/net/ipv4/conf 子目录中的内核变量和相关文件

内核变量名	文件名	IPv4/IPv6 的默认值
ipv4_devconf.arp_announce	*arp_announce*	0
ipv4_devconf.arp_filter	*arp_filter*	0
ipv4_devconf.arp_ignore	*arp_ignore*	0
ipv4_devconf.medium_id	*medium_id*	0
ipv4_devconf.proxy_arp	proxy_arp	0

如图29-3 所示，除了每个设备对应的子目录外，还有两个特殊的子目录：*default*和*all*。详细内容见第三十六章。

本部分涉及的数据结构

第二十七章的“主要的数据结构”一节中，我们对邻居子系统使用的主要数据结构做了个简单描述。本节将详细介绍每个结构的各个字段。

图29-4给出了定义每个数据结构的文件。浅色的数据结构不属于邻居子系统，但在本书的这一部分曾提到过。

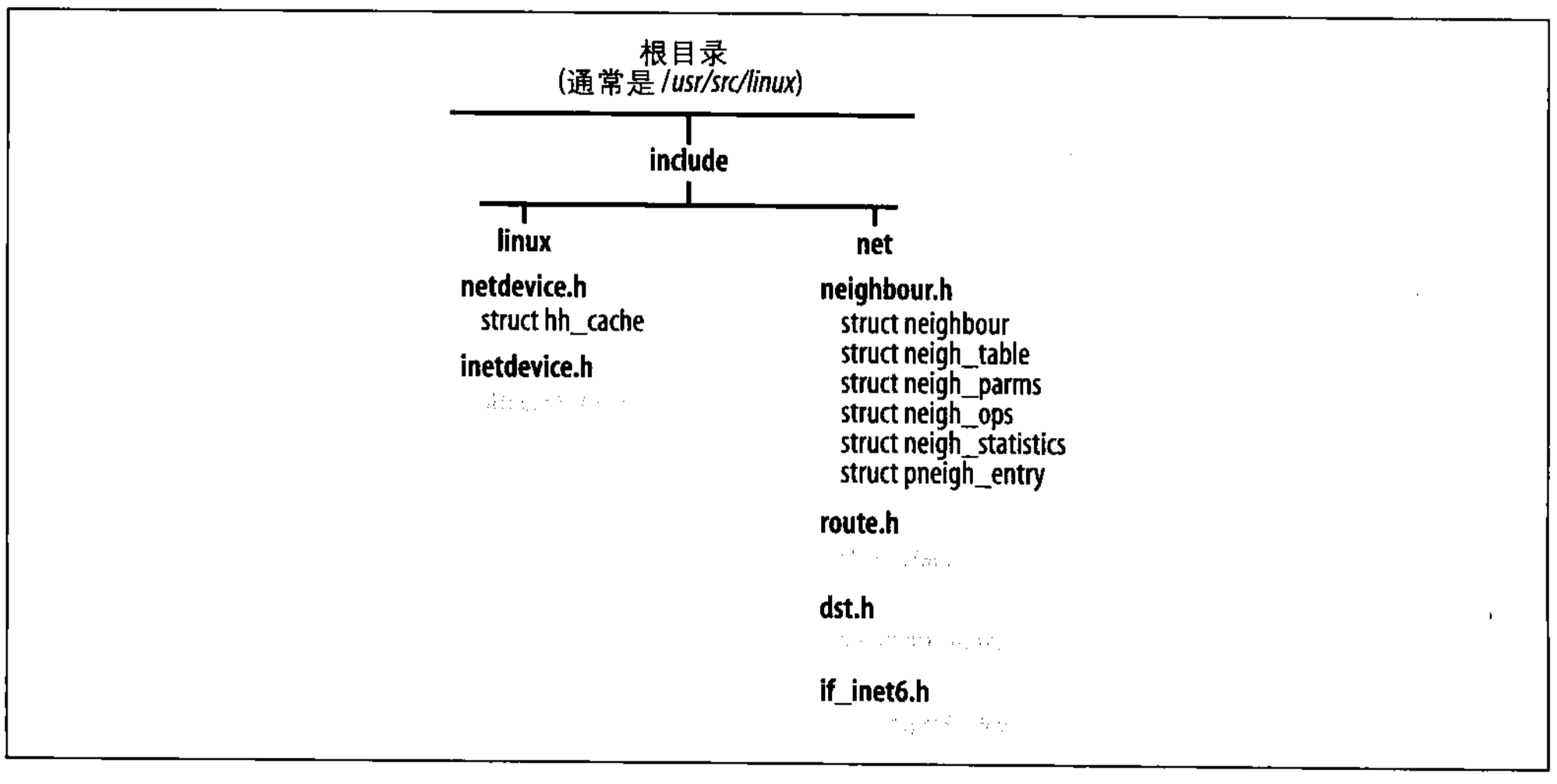


图 29-4：数据结构在内核文件中的分布

neighbour 结构

邻居是用struct neighbour结构定义的。该结构比较复杂，包括状态字段、与L3 协议的接口相关虚函数、定时器和缓存的L2 帧头。

下面是对每个字段的描述：

struct neighbour *next

每个neighbour项都插入到一个hash表中。next指向另一个neighbour结构，这个结构与当前结构冲突且共享同一个bucket。新元素总是插入到bucket列表的表头（见第二十七章的“创建一个邻居”一节和图27-2）。

struct neigh_table *tbl

指向neigh_table结构的指针，这个结构定义了与当前邻居项有关的协议。例如，如果邻居使用一个IPv4 地址，tbl 就指向arp_tbl结构。

struct neigh_parms *parms

用于调整邻居协议行为的参数。当建立一个neighbour结构时，用嵌入到和协议相关的neigh_table结构中的neigh_parms结构的默认值初始化parms。当neigh_create函数调用协议的constructor方法时（对ARP来说是arp_constructor），用相关设备的配置信息取代初始化时的默认值。而大部分设备使用的都是系统的默认值，本章前面讲过，可以用不同的参数启动设备，或者管理员以后可以用不同的参数配置设备。

struct net_device *dev

通过这个设备，可以访问该邻居。每个邻居只能通过一个设备来访问。该值不能为NULL，因为在其他内核子系统中，NULL值是个通配符，表示所有设备。

unsigned long confirmed

时间戳（用jiffies表示）表示该邻居的可到达性最后验证过的时间。L4协议用neigh_confirm函数（见第二十六章的图26-14）更新这个值。邻居基础结构用neigh_update函数更新它。

unsigned long updated

一个时间戳，表示neigh_update函数最近一次更新该邻居的时间（首次初始化时是由neigh_alloc函数设置）。不要将updated和confirmed混淆，这两个字段表示不同的事件。当邻居的状态改变时，要设置updated字段；而confirmed 字段只记录邻居特殊的一次状态改变：当邻居最近一次证明是有效时，发生的状态转变。

unsigned long used

邻居项最近一次被使用的时间。这个值不会随着数据传输而同步更新。当该邻居项还没有到NUD_CONNECTED态时，这个字段由neigh_resolve_output函数调用neigh_event_send来更新。相应地，当邻居项进入NUD_CONNECTED态时，它的值由neigh_periodic_timer更新为该邻居项的可到达性最近被证实的时间。

__u8 flags

这个字段的可选值在 *include/linux/rtnetlink.h* 和 *include/net/neighbour.h* 文件中：

#define NTF_PROXY 0x08

当使用用户空间命令 *ip neigh* 添加邻居项到代理表中时（例如，使用命令 *ip neigh add proxy 10.0.0.2 dev eth0*），传送给内核的数据结构中会设置这个标识，内核的处理函数neigh_add就会知道，有新邻居项要加到代理表中（见“邻居的系统管理”一节）。

#define NTF_ROUTER 0x80

这个标识由IPv6使用。当设置后，它表示邻居是一个路由器。这个标识和NTF_PROXY标志不同，它不能由用户空间工具设置。当IPv6邻居发现程序收到来自邻居的信息时，就会更新这个标志的值。

__u8 nud_state

指示邻居项的状态。可能的取值以NUD_*XXX*形式命名，它定义在 *include/net/neighbour.h* 和 *include/linux/rtnetlink.h* 中。在第二十六章的“NUD状态之间的转换”一节介绍了每个状态的作用。第二十六章的图26-13还给出了如何根据不同的事件转换邻居的状态。

__u8 type

当neigh_create函数调用协议的constructor方法（对于ARP是arp_constructor）创建邻居项时，就会设置这个字段。它的值可以用于各种场合，例如，决定哪些值可以赋给nud_state。type设为第三十六章的表13-12列出的值，文件 *include/linux/rtnetlink.h* 中也列出了这些值。

在本章的内容中，不会用到表中所有的值：我们只对这些值有兴趣：RTN_UNICAST、RTN_LOCAL、RTN_BROADCAST、RTN_ANYCAST 和RTN_MULTICAST。

假定有个IPv4地址（例如，与一个neighbour项相关的L3地址），inet_addr_type函数就会找到相应的RTN_XXX值（见第二十八章）。对于IPv6来说，有个类似的函数：ipv6_addr_type。

__u8 dead

若dead被设为1，表示该结构将被删除，不能再使用了。参见第三十二章“外部

事件”一节的neigh_ifdown函数，以及neigh_forced_gc和neigh_periodic_timer的使用范例。

atomic_t probes

失败的solicitation尝试的次数。它的值由neigh_timer_handler定时器检测。当尝试次数到达允许的最大值时，这个定时器就将该neighbour项转移到NUD_FAILED态。

rwlock_t lock

用于在出现竞争时对neighbour结构进行保护。

unsigned char ha[]

与primary_key（见下面）表示的L3地址关联的L2地址（例如，Ethernet NIC的Ethernet MAC地址）。这个地址是二进制格式。向量ha的长度是MAX_ADDR_LEN（在*include/linux/netdevice.h*文件中定义的是32），向上舍入到C语言long类型的一倍。一个Ethernet地址只有6个字节（48位），但是其他的链路层协议可能会需要更多位数。对于每种硬件地址类型，内核都定义一个符号，用于指定地址的长度。大部分符号都以*XXX*_ALEN或*XXX*_ADDR_LEN的形式命名。例如，Ethernet在*include/linux/if_ether.h*文件中定义了ETH_ALEN符号。

struct hh_cache *hh

缓存的L2帧头列表。参见第二十七章的“L2帧头缓存”一节。

atomic_t refcnt

引用计数。参见第二十七章的“缓存”和“邻居结构的引用计数”两节。

int (*output)(struct sk_buff *skb)

用于向邻居发送帧的函数。根据一些因素，这个函数指针实际指向的函数在该结构的生存期内可以改变多次。它的初始化是用neigh_table结构的constructor方法（见第二十八章的“初始化neighbour结构”一节）。当邻居项的状态为NUD_REACHABLE态或NUD_STALE态时，可以分别通过调用neigh_connect和neigh_suspect函数更新该字段的值。

struct sk_buff_head arp_queue

目的L3地址还没有被解析的封包被临时放到这个队列中。不要管这个队列的名称，它能被所有的邻居协议使用，不只是ARP。参见第二十七章的“出口队列”一节。

struct timer_list timer

用于处理几个任务的定时器。参见第十五章的“定时器”一节。

struct neigh_ops *ops

VFT中包含的各种方法，它们用于维护neighbour项。例如，这些方法中，有几

个是用于传送封包的，每个方法都为不同状态优化或相关的设备类型进行了优化。每个协议会提供三个或四个不同的 VFT；根据 L3 地址的类型、相关设备的类型、以及链路的类型（例如，点对点链路），选择合适的 VFT 用于特殊的 neighbour 项。参考上面一节“neigh_ops 结构”和第二十七章的“neigh->ops 的初始化”一节中的内容。

`u8 primary_key[0];`

该邻居的 L3 地址。它作为缓存查找函数的查找关键字。对 ARP 项来说，它是一个 IPv4 地址；对邻居发现项来说，它是一个 IPv6 地址。

neigh_table 结构

这个结构用于调整邻居协议的行为。在内核中有许多neigh_table结构的实例，每个不同的协议有一个：

`arp_tbl`

IPv4 使用的 ARP 协议（参见 *net/ipv4/arp.c*）

`nd_tbl`

IPv6 使用的邻居发现协议（参见 *net/ipv6/ndisc.c*）。

`dn_neigh_table`

DECnet 使用的邻居发现协议（参见 *net/decnet/dn_neigh.c*）。

`clip_tbl`

ATM over IP 协议（参见 *net/atm/clip.c*）。

当内核中相关的邻居子系统初始化时，这些neigh_table结构也被初始化，并插入到一个由 neigh_tables 指向的一个全局列表中，参见第二十七章的图 27-2。

这些数据结构包含了大部分（不是全部）邻居协议所需的信息。因此，每个 neighbour 项会有一个指针 neigh->tbl，指向与其关联的 neigh_table；例如，与 IPv4 地址相关的一个 neighbour 项中的指针是指向 arp_tbl 结构，而一个 IPv6 邻居项则指向 nd_tbl。

为了更容易地理解每个字段的含义，可以参考四个表的初始化过程。尤其是作为范围的 arp_tbl 结构，第二十八章“arp_tbl 表”一节也对其进行了讨论。

`struct neigh_table *next`

将所有的协议表链接到一个链表中。

rwlock_t lock

用于在出现竞争时保护这个表的锁。对于只需要读权限的函数，例如，neigh_lookup，该锁用于只读模式；对于其他函数，例如，neigh_periodic_timer，它可以处于读/写模式。

注意，整张表由单独一个锁保护，和常见的情况正好相反，例如，缓存表中的每个bucket都有一个不同的锁。

char *id

这只是一个用于标识协议的字符串。在分配内存池时（参见neigh_table_init），这个内存池用于分配neighbour结构，该字段主要作为一个ID 。

struct proc_dir_entry *pde

注册到*/proc/net/stat/*中的文件，用于输出协议的统计信息。例如，ARP协议会建立*/proc/net/stat/arp_cache*文件。当协议初始化时，由neigh_table_init函数建立这个文件。

int family

邻居协议所表示的邻居项的地址簇。它的可能取值位于*include/linux/socket.h*文件中，名称都是AF_*XXX*的形式。对于IPv4和IPv6，对应的值分别是AF_INET和AF_INET6。

int entry_size

插入到缓存中的数据结构的长度。由于一个neighbour结构包含一个字段，它的长度与具体的协议有关（primary_key）。entry_size字段的值就是一个neighbour结构的字节数与协议提供的primary_key字段的字节数之和。例如，在IPv4/ARP下，该字段被初始化为sizeof(struct neighbour)+4，这里4就表示IPv4地址占4个字节。例如，当neigh_alloc函数要清理从缓存中取回的邻居项的内容时，就要用到该字段（注1）。

int key_len

查找函数（参见第二十七章的“缓存”一节）使用的查找关键字的长度。由于查找关键字是一个L3地址，对IPv4来说，该字段值就是4；对IPv6来说，就是8；对DECnet来说，就是2。

__u32 (*hash)(const void *pkey, const struct net_device *)

Hash函数。在查找一个邻居项时，该函数用搜索关键字从hash表中选择正确的bucket。

注1： 当neigh_destroy把一个neighbour结构放回内存池时，它的内容不会被清除。

`int (*constructor)(struct neighbour *)`

当建立一个新的邻居项时，neigh_create函数调用的constructor方法。该方法会初始化新neighbour项中协议指定的一些字段。例如，在第二十八章的“初始化neighbour 结构”一节中，详细介绍了ARP协议使用的方法（arp_constructor）。

`struct neigh_parms parms`

这个数据结构包含了一些用于调整邻居协议行为的参数，例如，在没有收到应答后，重新发送一个solicitation请求前要等待多长时间；等待一个应答时，队列中保存多少个要发送的封包。参考“neigh_parms 结构”一节。

`struct neigh_parms *parms_list`

没有使用。

`kmem_cache_t *kmem_cachep`

分配neighbour结构时需要的内存池。这个内存池在协议初始化时，由neigh_table_init分配和初始化。你可以通过将其内容转储到*/proc/slabinfo file*文件来查看其状态。

`atomic_t entries`

在协议缓冲中当前neighbour结构实例的数目。每当用neigh_alloc分配一个新的邻居项，它的值就加1；用neigh_destroy释放一个邻居项，它的值就减1。参考本节后面对gc_thresh1、gc_thresh2和gc_thresh3的描述。

`unsigned long last_rand`

与一个表（每个设备有这样一个表）关联的neigh_parms结构中，变量reachable_time最近被更新的时间（用jiffies表示）。

`struct neigh_statistics *stats`

缓存中的neighbour实例的各种统计信息。参见“neigh_statistics 结构”一节。

`struct neighbour **hash_buckets`

存储neighbour项的hash表。

`unsigned int hash_mask`

Hash表的长度。参见第二十七章的图27-6。

`__u32 hash_rnd`

当缓存长度要增加时，用于分发neighbour项的随机数值。参见第二十七章的“缓存”一节。

下面的变量和函数用于第二十七章“垃圾回收”一节中介绍的垃圾回收算法：

int gc_interval

这个变量用来控制gc_timer 定时器多久会超时，并启动垃圾回收。过去它的值是30秒，现在可能更短。定时器每次只会触发哈希表中一个bucket中的垃圾回收。更多信息可以参见第二十七章的“垃圾回收”一节。

int gc_thresh1

int gc_thresh2

int gc_thresh3

这三个阈值定义了三个不同级别的内存状态，邻居协议可将这些状态赋给当前缓存中的 neighbour 项。

unsigned long last_flush

这个变量表示neigh_forced_gc函数最近一次执行的时间，用jiffies测量。换句话说，它表示由于内存不足，最近一次垃圾回收程序执行时的时间。

struct timer_list gc_timer

垃圾回收定时器。参见第二十七章的“垃圾回收”一节。

unsigned int hash_chain_gc

记录定期运行的垃圾回收定时器要扫描的 hash 表中的下一个 bucket。这些 bucket 是按顺序扫描的。

下面的字段在系统作为一个代理时要用到。参见第二十七章的“担任代理”一节。

struct pneigh_entry **phash_buckets

存储要被代理的 L3 地址的表。

int (*pconstructor)(struct pneigh_entry *)

void (*pdestructor)(struct pneigh_entry *)

pconstructor 方法是 constructor 方法的对应体。现在，只有 IPv6 使用 pconstructor；当首次配置关联的设备时，这个方法会注册一个特殊的多播地址。

当释放一个代理的邻居项时，要调用 pdestructor 方法。它只能由 IPv6 中使用，撤销 pconstructor 方法所做的工作。

struct sk_buff_head proxy_queue

当启动代理并且配置了非空的 proxy_delay 延迟时，收到的 solicit 请求（例如，ARP 下收到 ARPOP_REQUEST 封包）就放到这个队列中。新元素被加到队尾。

void (*proxy_redo)(struct sk_buff *skb)

当 solicit 请求（例如，ARP 下的 ARPOP_REQUEST 封包）从代理队列 neigh_table ->proxy_queue 中取出后，处理该请求的函数。参见第二十七章的“延迟处理 Solicitation 请求”一节。

struct timer_list proxy_timer

当proxy_queue队列中至少有一个元素时，就会启动这个定时器。若定时器超时，执行的处理函数是neigh_proxy_process。由neigh_table_init函数在协议初始化时对这个定时器初始化。它与neigh_table->gc_timer定时器不同，不会周期性启动，只会在需要的时候启动（例如，在往proxy_queue中首次增加一个元素时，协议就会启动它）。第二十七章的“担任代理”一节介绍了一个元素什么时候及为何要放到proxy_queue队列，另外还介绍了proxy_timer 定时器如何对其处理。

neigh_parms 结构

neigh_parms数据结构存储着邻居协议的配置参数。对于每个使用邻居协议的配置好的L3 协议，有多个 neigh_parms 实例：每个设备（注 2）对应一个，再加上一个保存默认参数值。

下面是对每个字段的描述：

struct neigh_parms *next

链接到和同一个协议簇关联的 neigh_parms 实例的指针。意思就是说每个neigh_table结构有它自己的neigh_parms结构列表，每个实例对应于一个配置的设备（参见第二十七章的图 27-2）。

int (*neigh_setup)(struct neighbour *)

对于仍然使用旧邻居基础结构的设备，主要使用这个函数来初始化。通常，该函数只用于将 neighbour->ops 初始化为 arp_broken_ops 实例（见本章后面的“neigh_ops 结构”一节，以及第二十七章的“neigh->ops 的初始化”一节）。也可参见文件 *drivers/net/shaper.c* 中的 shaper_neigh_setup 函数。要想明白在新 neighbour 结构初始化过程中什么时候调用这个函数，可以参见第二十八章的图 28-11。

不要将这个虚函数与net_device->neigh_setup函数混淆。当第一次给一个设备配置 L3 地址，和初始化 eigh_parms->neigh_setup时，要调用后者。每个设备只调用 net_device->neigh_setup一次。每个与该设备关联的neighbour结构也只调用 neigh_parms->neigh_setup一次。

注 2：　这个说法并不是百分之百正确。因为一个neigh_parms结构用来调整邻居协议的行为，系统中至少有一个设备的 L3 配置使用了邻居子系统，这个结构才是需要的。

struct neigh_table *tbl

回溯指针，指向持有该结构的neigh_table结构。

int entries

void *priv

暂不使用。

void *sysctl_table

这个表在*net/ipv4/neighbour.c*文件的结尾的程序中完成初始化。它与允许用户修改neigh_parms结构中某些参数的值有关，这些参数可以导出到*/proc*中，参见“使用/proc文件系统调整邻居性能”一节的描述。

int base_reachable_time

int reachable_time

base_reachable_time是一个时间间隔（用jiffies表示），表示自从最近一次收到可到达性证明后经过的时间。注意，这个间隔是用于计算实际时间的一个基本值，实际时间保存在reachable_time（注3）中。实际时间是base_reachable_time和3/2 base_reachable_time之间的一个随机值（均匀分布）。这个随机值每300秒由neigh_periodic_timer 更新一次，但它也可以由其他事件更新（尤其在IPv6中）。

int retrans_time

当一台主机在retrans_time时间内没有收到solicitation请求的应答时，就会发出一个新的solicitation 请求，一直到尝试次数到达最大值。retrans_time也是用jiffies表示。

int gc_staletime

如果一个neighbour结构在gc_staletime时间内还没有被使用过，并且没有程序引用它，那么它就会被删除。gc_staletime是用jiffies表示的。

int delay_probe_time

这个变量表明在一个邻居在进入NUD_PROBE态前，在NUD_DELAY态等待了多长时间。参见第二十六章的图26-13。

int queue_len

arp_queue队列中能容纳的元素的最大数目。

int proxy_qlen

proxy_queue队列中能容纳的元素的最大数目。

注3： 在ND/IPv6中，reachable_time的值也可以通过协议头的特定字段在路由器和主机间交换。

int ucast_probes

int app_probes

int mcast_probes

ucast_probes表示为了证实一个地址的可到达性，能发送的单播solicitations数量。

app_probes是用户空间应用程序在解析一个地址时（对于IPv4/ARP的情况，参见第二十八章的“ARPD”一节），可以发送的solicitations的数量。

mcast_probes表示为了解析一个邻居地址，可以发出的多播solicitations的数量。对ARP/IPv4来说，这实际就是广播solicitations的数目，因为ARP不使用多播solicitations，但IPv6使用。

注意，mcast_probes和app_probes是互斥的（只能有一个非空）。

int anycast_delay

暂不使用。

int proxy_delay

邻居协议封包在代理处理前，在代理队列中应该保存的时间（用jiffies表示）。参见第二十七章的“延迟处理Solicitation请求”一节。

int locktime

相邻两次对一个neighbour项中的字段（代表性的是nud_state字段和ha字段）更新时，必须经过的最小时间（用jiffies表示）。这个时间窗口可以避免令人讨厌的乒乓作用发生。例如，当在一个网络中有多个ARP代理服务器时，它们都会对同一个solicitations做出应答，这样应答的地址就会冲突。第二十八章的“普通处理的结尾”一节有对这个问题的详细论述。

int dead

这是个布尔标识，设置后表示该邻居实例可以“被删除”。参见neigh_parms_release。

atomic_t refcnt

引用计数。

struct rcu_head rcu_head

负责管理互斥。

引用计数refcnt的使用应该多说一些。在下面的论述中请参见第二十七章的图27-2。由于每个设备在每个协议中都有一个neigh_parms结构的实例，还有一个实例是嵌在neigh_table结构中，用于保存默认参数值，再加上在每个neighbour结构中的

neigh_parms指针，就容易混淆：这个指针是指向哪个neigh_parms实例，这个实例又是属于谁，下面我们对其进行仔细分析。

每个neigh_table表中，也就是每个协议，都有自己的neigh_parms实例。这个实例中保存着协议提供的默认参数。每个设备的net_device结构可以配置多个L3协议。对每个配置在该设备上的L3 协议，net_device都有个指针，指向保存该L3 协议配置信息的数据结构（例如，IPv4 协议的是in_device）。这个协议指定的结构中也包含一个指针，指向一个neigh_parms实例，这个实例中保存着该L3 协议所用的邻居协议（对于IPv4 是ARP）的配置参数。

表29-5 列出了主要的协议初始化函数，这些函数给neigh_parms结构分配存储空间。从图29-3 中，你可以看到两个IP 协议分配的结果。

表29-5：L3 协议初始化函数

协议	函数	所在文件
IPv4	inetdev_init	*net/ipv4/devinet.c*
IPv6	ipv6_add_dev	*net/ipv6/addrconf.c*
DECnet	dn_dev_create	*net/decnet/dn_dev.v*

在剩下的讨论中，我们只考虑IPv4。当在一个设备上第一次配置IPv4 信息时，IPv4 的相关函数就会调用inetdev_init函数，来分配ARP 协议使用的neigh_parms实例。这个新的neigh_parms实例的内容是拷贝自neigh_table->parms，而ARP 下neigh_table实例是arp_tbl。当创建一个neighbour实例时，neigh->parms就被初始化为相关联设备的neigh_parms实例。我们在“使用/proc文件系统调整邻居性能”一节已经看到，全局默认配置（neigh_table->parms）和每个设备的独立配置信息都可以由管理员改变。

因为每个设备的neigh_parms结构要被所有的与该设备关联的neighbour实例引用，因此neigh_parms->refcnt就保存着这些引用计数。可以直接或间接更新这个引用计数的函数有：

neigh_parms alloc

neigh_parms_destroy

分配和释放一个neigh_parms实例。只有当这个结构的引用计数是0时，neigh_parms_destroy函数才被调用。

__neigh_parms_put

neigh_parms_put

__neigh_parms_put 函数只会对引用计数减 1。如果引用计数为 0，neigh_parms_put 函数也可以调用 neigh_parms_destroy 函数。

neigh_parms_release

将 neigh_parms 实例标记为停用（dead），并间接调用 neigh_parms_put 函数。

neigh_parms_clone

对一个结构的引用计数加 1，返回指向该结构的指针。

neigh_rcu_free_parms

由 neigh_parms_release 调用这个函数来实际删除结构（这里就需要用到 neigh_parms->rcu_head 来处理互斥）。

neigh_ops 结构

neigh_ops 结构由一些函数指针组成，这些指针指向在一个 neighbour 项生存期内的不同时间要调用的函数。这些函数大部分是虚函数，作为 L3 协议和第十一章介绍的 dev_queue_xmit API 间的接口。还有一些函数是由邻居基础结构（neigh_xxx 函数）提供的，剩下的是由各个邻居协议提供的（ARP 协议提供的 arp_xxx 函数）。参考第二十八章的“初始化 neighbour 结构”一节的内容。

这些函数间最主要的区别是它们使用的上下文环境不同。第二十六章的“特殊情况”一节包括两种最常用的情况。

下面是该结构中每个字段的描述：

int family

在描述 neigh_table 结构的类似的 family 字段时，我们已经知道它的含义。

void (*destructor)(struct neighbour *)

当一个 neighbour 项要被 neigh_destroy 删除时，就执行该函数。它基本上是 neigh_table->constructor 方法的互补方法。但由于某些原因，constructor 位于 neigh_table 结构中，而 destructor 位于 neigh_ops 结构中。

void (*solicit)(struct neighbour *, struct sk_buff*)

发送 solicitation 请求的函数。

void (*error_report)(struct neighbour *, struct sk_buff*)

当一个邻居被认为不可到达时，要调用这个函数。参见第二十七章的“邻居层产生的事件”一节。

下面的四个方法传送的是数据封包，不是邻居协议封包。这些方法间的区别在于它们使用的环境不同。参见第二十七章的“L3 协议和邻居协议间的通用接口”一节。

`int (*output)(struct sk_buff*)`

这个是最普通的函数，用于所有的情况下。它会检查地址是否已经被解析过；在没有被解析的情况下，它会启动解析程序。如果地址还未准备好，它会把封包保存在一个临时队列中，并启动解析程序。由于该函数为了保证接收方是可到达的，它会做每一件必要的事情，因此它相对来说需要的操作比较多。不要将 `neigh_ops->output` 和 `neighbour->output` 相混淆。

`int (*connected_output)(struct sk_buff*)`

当已经知道邻居是可到达时（邻居状态为NUD_CONNECTED态），使用该函数。因为所有需要的信息都是满足的，该函数只要简单填充一下L2 帧头，因此它比output速度要快。

`int (*hh_output)(struct sk_buff*)`

当地址已被解析过，并且整个封包头已经根据上一次传输结果放入帧头缓存时，就使用这个函数。参见第二十七章的“邻居协议和 L3 传输函数的交互”一节。

`int (*queue_xmit)(struct sk_buff*)`

前面的函数中，除了 `hh_output` 外，都不会实际传输封包。它们所做的工作都是确保封包帧头是编写好的，然后当缓冲区准备好时，调用 `queue_xmit` 方法来传输。参见第二十七章的图 27-3(b)。

hh_cache 结构

用于保存缓存的L2 帧头的数据结构是 `struct hh_cache`，在 *include/linux/netdevice.h* 文件中定义（这个结构名称来自“hardware header”的首字母）。下面是对其内部各个字段的描述，第二十七章的“L2 帧头缓存”一节介绍了如何使用这个结构。

`unsigned short hh_type`

与 L3 地址相关的协议（见 *include/linux/if_ether.h* 文件中的 `ETH_P_XXX` 值）。

`struct hh_cache *hh_next`

和同一个 `neighbour` 项关联的缓存的 L2 帧头可以有多个。但是，给定一个 `hh_type` 值只能有一个与该邻居关联（参见 `neigh_hh_init`）。

`atomic_t hh_refcnt`

引用计数。

`int hh_len`

以字节数表示的缓存的帧头长度。

`int (*hh_output)(struct sk_buff *skb)`

传输封包的函数。和neigh->output一样，这个方法被初始化为neigh->ops VFT中的一个方法。

`rwlock_t hh_lock`

用于在竞争条件下保护hh_cache结构的锁。例如，一个IP函数想传送一个封包（参见第二十七章的“邻居协议和L3传输函数的交互”一节），它在将帧头从hh_cache 结构复制到封包的skb缓冲区前，需要获得只读锁。当这个结构的某个字段需要更新时，这个锁就进入互斥模式：例如，当hh_output需要被初始化为另一个函数（注4）时，或者在由于目的主机链路层地址已改变，需要更新hh_cache->hh_data 帧头时，就需要获得这个锁。

`unsigned long hh_data[HH_DATA_ALIGN(LL_MAX_HEADER) / sizeof(long)]`

缓存的帧头。

neigh_statistics 结构

这个结构中存储着对邻居协议的统计信息，用户可以仔细查看这些信息。每个邻居协议都有它自己的该结构实例。这个结构定义在*include/net/neighbour.h*文件中。下面是对该结构各个字段的描述：

`unsigned long allocs`

邻居协议分配的neighbour结构的总数。包括那些已经被删除的neighbour结构。

`unsigned long destroys`

删除的neighbour项的数目。由neigh_destroy函数负责更新。

`unsigned long hash_grows`

hash表容量增加的次数，由neigh_hash_grow函数负责更新（参见第二十七章的“缓存”一节）。

`unsigned long res_failed`

解析一个邻居地址识别后尝试的次数。每次送出一个新的solicitation封包，不会增加这个值；只有当所有的尝试都失败后，才会由neigh_timer_handler函数将该值递增。

注4：在*net/core/neighbour.c*文件的neigh_destroy函数中可以找到使用hh_lock字段的更好的例子。这个函数中，锁是用于处理无法删除一个邻居项的情况，因为它的引用计数是非零的。

unsigned long lookups

调用 neigh_lookup 函数的次数。

unsigned long hits

neigh_lookup 函数查询成功的次数。

unsigned long rcv_probes_mcast

unsigned long rcv_probes_ucast

这两个字段只有 IPv6 使用，表示收到的 solicitation 请求的数量。两个字段分别表示多播地址请求和单播地址请求。

unsigned long periodic_gc_runs

unsigned long forced_gc_runs

neigh_periodic_timer和neigh_forced_gc各自被调用的次数。参见第二十七章的“垃圾回收”一节。

内核中为每个 CPU 保存这些计数器的一个实例。这些计数器是由 NEIGH_CACHE_STAT_INC 宏更新，该宏定义在 *include/net/neighbour.h* 文件中。注意这个宏会更新当前 CPU 的计数器。

每个协议的 neigh_statistic 结构的各个字段可以输出到对应的 */proc/net/stat/{protocol_name}_cache* 文件中。

本书此部分中重要的数据结构

表29-6 总结了在本书中与邻居子系统有关的章节中，曾介绍过或涉及到的主要的函数、变量和数据结构。

表 29-6：邻居子系统中的函数、变量和数据结构

函数	说明
dev_queue_xmit	封包传输的主要函数。参见第二十七章的“neigh->output 使用的函数”一节。
neigh_compat_output	
neigh_resolve_output	
neigh_connected_output	
neigh_blackhole	
neigh_update	更新存储在 neighbour结构中的信息。参见第二十七章的“邻居信息的更新：neigh_update”一节。
neigh_update_hhs	
neigh_sync	
neigh_confirm	证实一个邻居的可到达性。

表 29-6：邻居子系统中的函数、变量和数据结构（续）

函数	说明
neigh_create *neigh_destroy*	根据协议事件的处理结果创建和删除一个neighbour结构。参见第二十七章的“创建一个邻居项”和“删除邻居”两小节。
neigh_add *neigh_delete*	根据用户空间命令创建和删除一个 neighbour 结构。参见“邻居的系统管理”一节。
neigh_alloc	为一个 neighbour 结构分配内存。
neigh_connect *neigh_suspect*	用于实施邻居间的可到达性。参见第二十七章的“neigh->output 和 neigh->nud_state 的初始化”一节。
neigh_table_init	向内核注册一个邻居协议。
neigh_ifdown	当收到外部子系统的通知时，处理L3 地址状态的转变。参见第二十七章的“使用 neigh_ifdown 更新”一节。
neigh_proxy_process	当代理定时器超时后，需要执行的处理函数。参见第二十七章的“延迟处理 Solicitation 请”一节。
neigh_timer_handler	参见第十五章的“定时器”一节。
neigh_periodic_timer *neigh_forced_gc*	垃圾回收算法使用的函数。参见第二十七章的“垃圾回收”一节。
neigh_lookup *_ _neigh_lookup* *_ _neigh_lookup_errno* *arp_find*	检查缓存中的邻居项。参见第二十七章的“缓存”一节。
neigh_hold *neigh_release*	递增 / 递减一个 neighbour 结构的引用计数。
pneigh_enqueue *pneigh_lookup*	由基于目的地的代理使用。参见第二十七章的“延迟处理 Solicitation 请求”和“按设备代理和按目的地代理”两节，以及第二十八章的“ARP 代理”一节。
arp_rcv *ndisc_rcv*	ARP 和 ND 封包各自的协议处理函数。
ip_finish_output2 *ip6_output_finish*	IPv4 和 IPv6 各自的传输函数。参见第二十七章的“邻居协议和 L3 传输函数的交互”一节。
neigh_hh_init	用一个 L2 地址初始化一个 hh_cache 结构，并将其绑定到相关的路由表缓存项上。参见第二十七章的“路由缓存和 L2 帧头缓存之间的联系”一节。

表 29-6：邻居子系统中的函数、变量和数据结构（续）

函数	说明
变量	
neigh_tables	向内核注册过的协议列表。
arp_tbl nd_tbl dn_neigh_table clip_tbl	这四个neigh_table结构定义了内核中实现的四种邻居协议。
数据结构	
struct neighbour struct neigh_table struct neigh_parms struct neigh_ops struct hh_cache struct neigh_ statistics	第二十七章中描述的主要的数据结构，它们的详细的引用方式参见 “本书邻居部分中重要的数据结构”一节。

本部分涉及的文件和目录

图 29-5 给出了与本章介绍的邻居子系统相关的重要文件和目录。

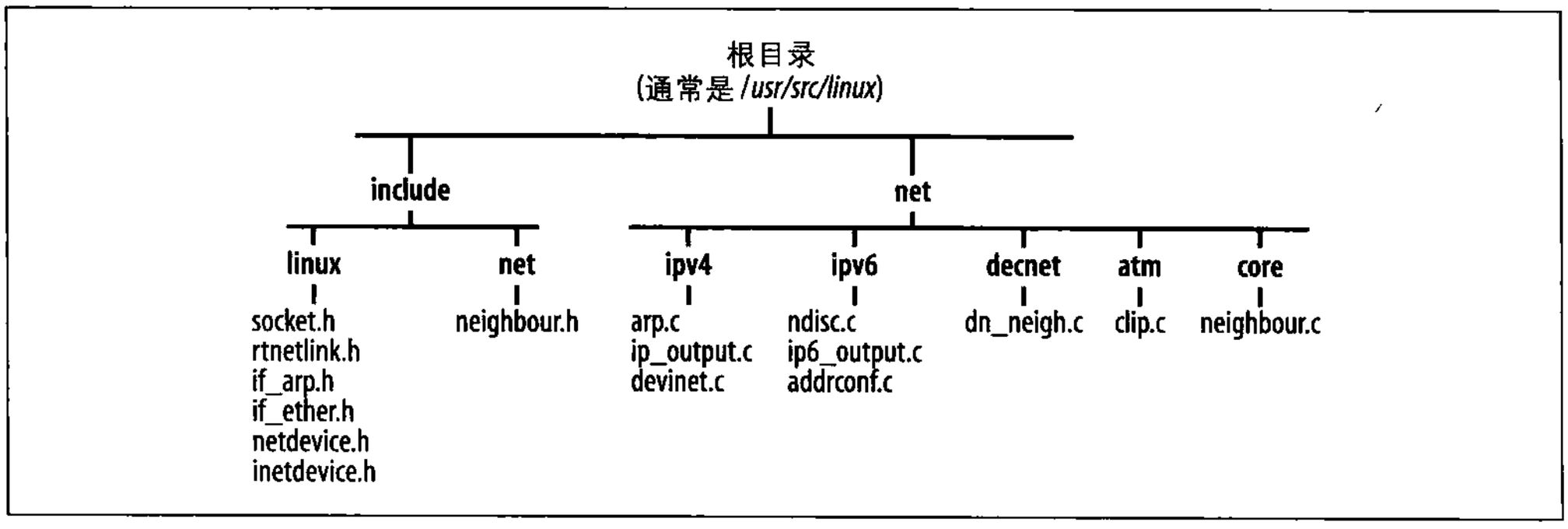


图 29-5：本部分涉及的文件和目录

第七部分

路由

第三层协议（L3）必须知道如何到达预期接收每个封包的系统，例如，IP协议。接收方可能在隔壁小房间，也可能在世界的遥远处。当涉及网络比较多时，L3协议负责计算最有效路径（如果能够找到的话），然后将消息沿着该路径发送给下一个系统，也就是送给下一跳（next hop），这一过程被称为路由（routing）。路由在Linux网络代码中扮演着核心角色。下面给出路由部分各章所涉及的内容：

第三十章 路由：概念

本章介绍一个基本路由器的功能，这也是Linux内核必须提供的。

第三十一章 路由：高级路由

本章介绍在更为复杂的环境下，通过使用一些选项来配置路由。在这些选项中，我们将看到策略路由（policy routing）与多路径路由（multipath routing），也将看到与路由系统交互的其他子系统。

第三十二章 路由：Linux的实现

本章描述Linux中路由代码用到的主要数据结构，以及路由子系统的初始化，并给出路由子系统与其他内核子系统之间的交互方式。

第三十三章 路由：路由缓存

本章描述路由缓存，包括协议无关缓存（目的缓存或DST）。主要内容包括如何将在路由缓存中插入和删除元素，以及垃圾回收和查找算法。

第三十四章 路由：路由表

本章主要描述路由表的结构，以及在路由表中如何添加和删除路由项。

第三十五章 路由：查找

本章描述针对入口流量和出口流量，在配置或没有配置策略路由的情况下路由表的查找。

第三十六章 路由：相关主题

本书结尾的这一章详细描述第三十二章中介绍的数据结构，讨论用户空间与内核之间的接口。还描述旧的路由管理工具 *net-tools* 和新的管理工具 IPROUTE2 软件。

第三十章

路由：概念

图 30-1 给出了在网络协议栈中路由子系统的位置(灰色框)，这张图没有包含所有的细节（Netfilter、网桥等），但给出了横跨路由的其他主要的内核子系统。

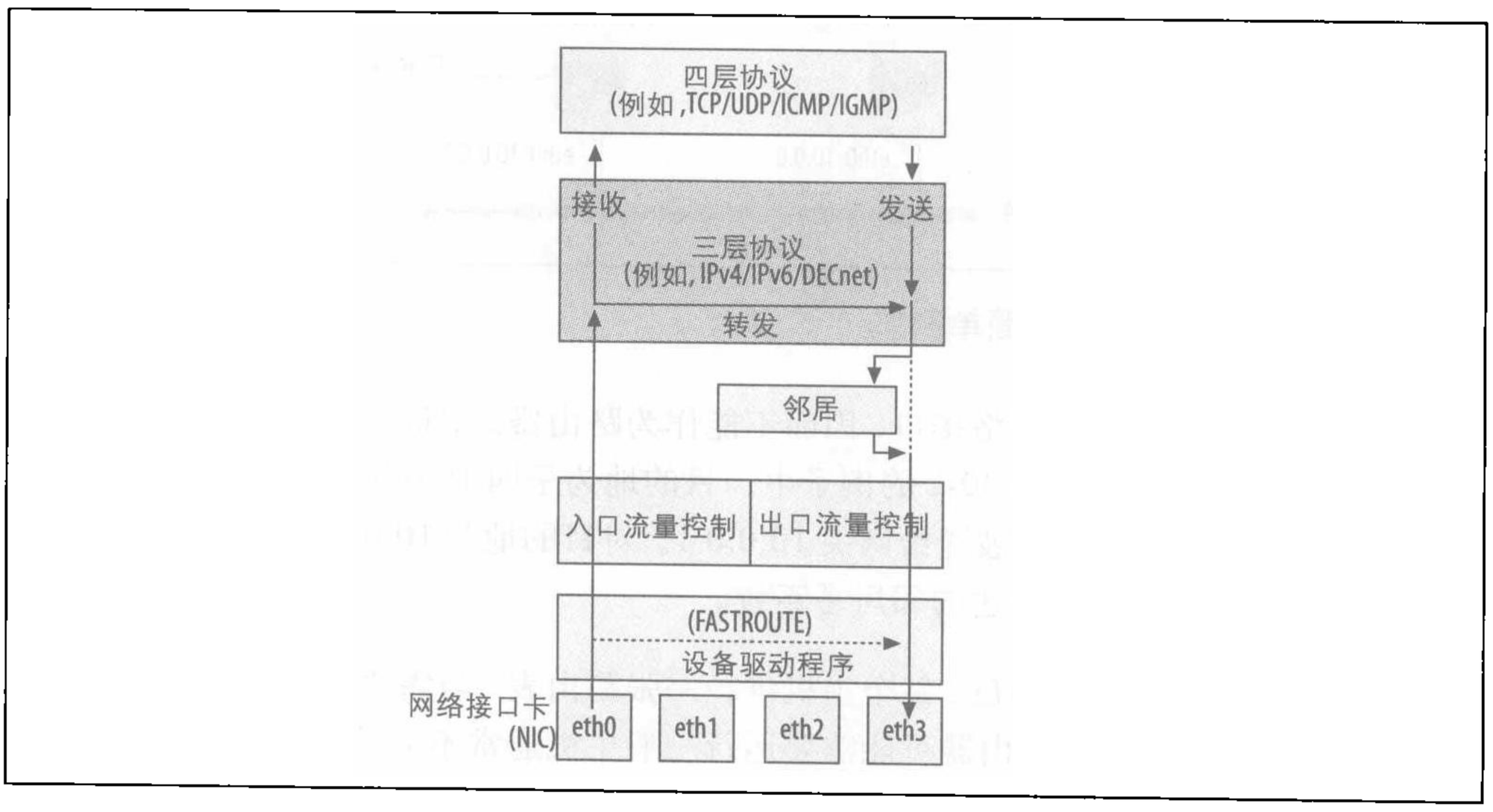


图 30-1：路由子系统和其他主要网络子系统间的关系

为了形象地解释一些特性或实现细节，我经常会给出用户空间配置的一些截图。如果你需要进一步学习在这些演示例子中用到的用户空间工具，可以参见第三十六章。

对路由讨论将针对 IPv4 网络，但我也会说明 IPv6 中与 IPv4 有明显区别的地方。

路由器、路由与路由表

简单地讲，路由器就是一台网络设备，它配备多个网络接口卡（Network Interface Card，NIC）、能利用它的网络知识正确转发入口流量（注 1）。

决定一个入口封包应当送给本地主机还是转发所需要的信息，以及在转发时正确转发封包所需要的信息，都存储在一个被称为转发信息库（Forwarding Information Base，FIB）的数据库中，它通常被简称为路由表（routing table）。

图 30-2 给出了一个简单的网络环境：一台子网为 10.0.0.0/24 的局域网内的主机和一台路由器 RT，局域网内的主机使用该路由器来访问互联网。

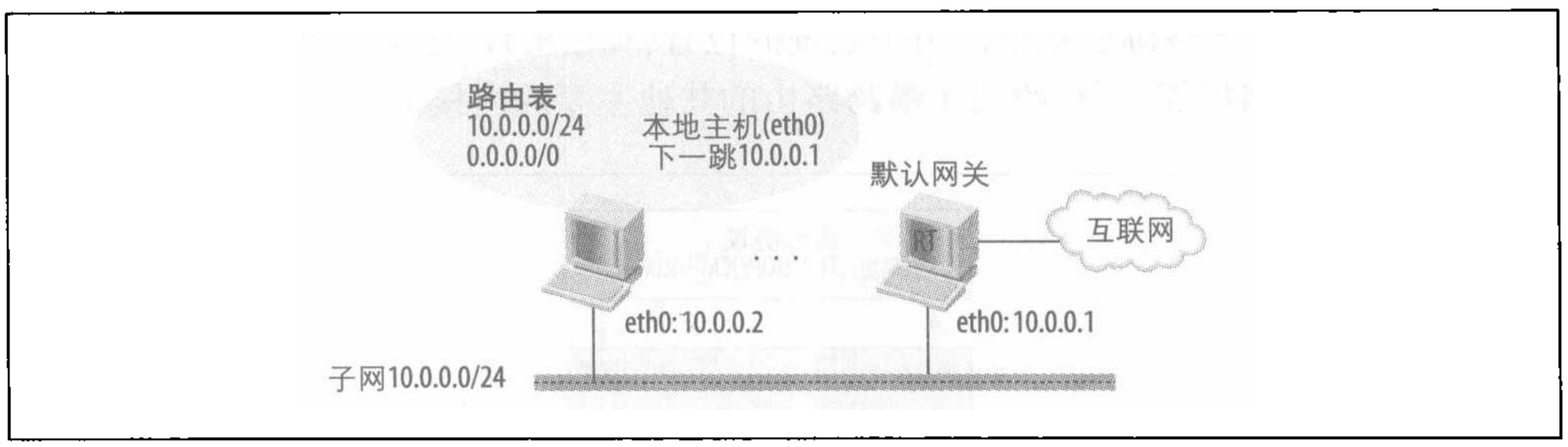


图 30-2：路由器和路由表的简单例子

大多数主机通常只有一个网络接口，因而不能作为路由器。主机被配置为使用一个默认网关来访问外部地址。在图 30-2 的例子中，目的地为子网 10.0.0.0/24 之外任何主机的流量（用 0.0.0.0/0 指定）都被送给网关 10.0.0.1。对目的地为 10.0.0.0/24 网络内的主机的流量，则使用第六部分描述的邻居子系统。

不考虑网络内主机扮演的角色，每个主机维护一张路由表，当需要处理网络流量的发送和接收时查询该路由表。路由器可能需要运行一般主机通常不会使用的特殊软件，称为路由协议（routing protocols）。毕竟，路由器需要更多关于如何到达远程网络的知识，非路由器主机就是依赖这些知识来访问远程网络的。路由协议的介绍超出了本书范围。

在特定环境下，主机所需要的路由能力可以进一步降低，例如，在第二十八章“ARP 代理服务器作为路由器”一节中所描述的。但在本章中，我们主要说明常见情况。

路由表无非就是许多路由的一个集合。一条路由（route）就是一组参数，这些参数存储了往一个给定目的地转发流量所需要的信息。在第三十二章中，我们将看到在 Linux 中

注 1：IPv6 和 IPv4 不同，它通过在 IP 头中使用一个特殊标识来明确定义路由器的角色。

如何来定义路由的细节，但这里我们可以假定一条路由所需要的最少的参数集合，以图30-2作为参考。

目的网络

路由表被用于往目的地转发流量。那么目的地是路由查找程序使用的最重要字段一点也不惊奇。图30-2给出了一张包含两条路由的路由表：一条是到本地子网10.0.0.0/24，另一条是到其他任何地方。后一条路由被称为默认路由（default route），在路由表中被记录为全零网络（见“默认网关”一节）。

出口设备

与路由项匹配的封包应当从该设备发送出去。例如，送往10.0.0.100的封包通过*eth0*送出。

下一跳网关

当目的网络和本地主机不是直连时，需要通过其他的路由器来转发。例如，在图30-2中的主机需要通过路由器RT才能到达10.0.0.0/24子网外的其他主机。下一跳网关是指该路由器的地址。

非路由多接口主机（Nonrouting Multihomed Hosts）

前面提到，一个路由器通常有多个NIC，假定它的主要任务是把从一个接口收到的数据通过另一个接口转发出去。但是，有些非路由主机，尤其是服务器也拥有多个NIC的，实际上它们不转发任何封包。由于以下的一个或多个原因，拥有多个NIC的大型服务器很常见：

高可用性

如果一个接口发生故障或停止工作，可以由另一个接口接管流量（该接口也可以与另一个局域网相连）。

更强的路由能力

服务器可以配置有多条路由而不只是一条默认路由。例如，出于特殊原因（例如，便于记录系统日志），可以用静态路由或多个NIC来到达特定的主机或子网。图30-3给出了一个例子，一台多接口主机的第二个NIC与另一个局域网相连以便能到达主机A。注意这台多接口主机并不在两个局域网之间转发流量，这样的话按照定义它应当是一台路由器。

多路传输（Channeling）

可以将多个接口捆绑在一起，使它们看上去像路由子系统的一个接口。对一个给定的连接，这个额外层（对路由子系统透明）可以使整个带宽增加，这个特性对于高负荷的服务器是很有价值。

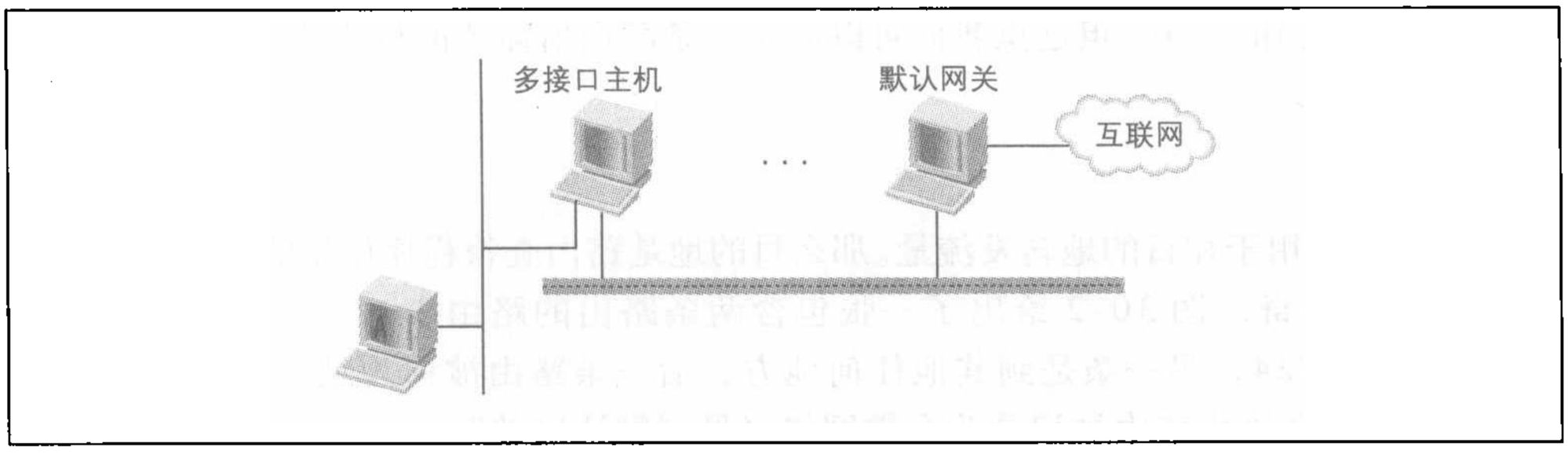


图 30-3：一台多接口主机例子

如果主机不是前面提到的这几种情况，那么可以被视为一台路由器，因为这些情况下主机不从一个接口向另一个接口转发流量。也就是说，这样一台主机除了在错误或非常特殊的条件（代理、混杂接口等）下，永远不接收目的为非本地主机的流量（可接受的流量包含广播与多播流量）。可以将多播和广播流量视为目的地为主机的流量。

各种路由配置

路由是一个复杂主题，我们不可能分析所有的环境、问题和解决方案。最重要的是通过阅读源代码时可以了解这个主题的一部分，理解为什么要考虑和特殊处理一些看似多余的情况。

图30-4给出了有助于理解路由子系统设计的三种配置。在这些配置中路由器以R*n*命名。我们看看这三个配置有哪些特殊之处：

- (a) 这是最常见的情形，不同的接口配置为不同的子网，每个子网连接的局域网都不同。
- (b) 路由器 RT 在同一个局域网中有两个接口（图中路由器下面所示），但这两个接口配置为不同的子网。
- (c) 路由器RT在子网10.0.2.0/24和10.0.3.0/24上各有一个地址，但这两个地址是配置在同一个NIC上。这可以用两种不同方法来实现：使用IPROUTE2软件包可以增加配置多个IP地址的能力，或通过创建旧的接口别名。本章后面我们将对这两种方法进行简单比较。

(b) 和 (c) 的情况不常见，但它们是完全合理的，并体现了Linux与IP的灵活性。它们所隐含的意义对你来说可能还不是很清晰。我们将在本章后面讨论它们并证明它们的合理性，我们首先从两个简单的隐含意义开始。

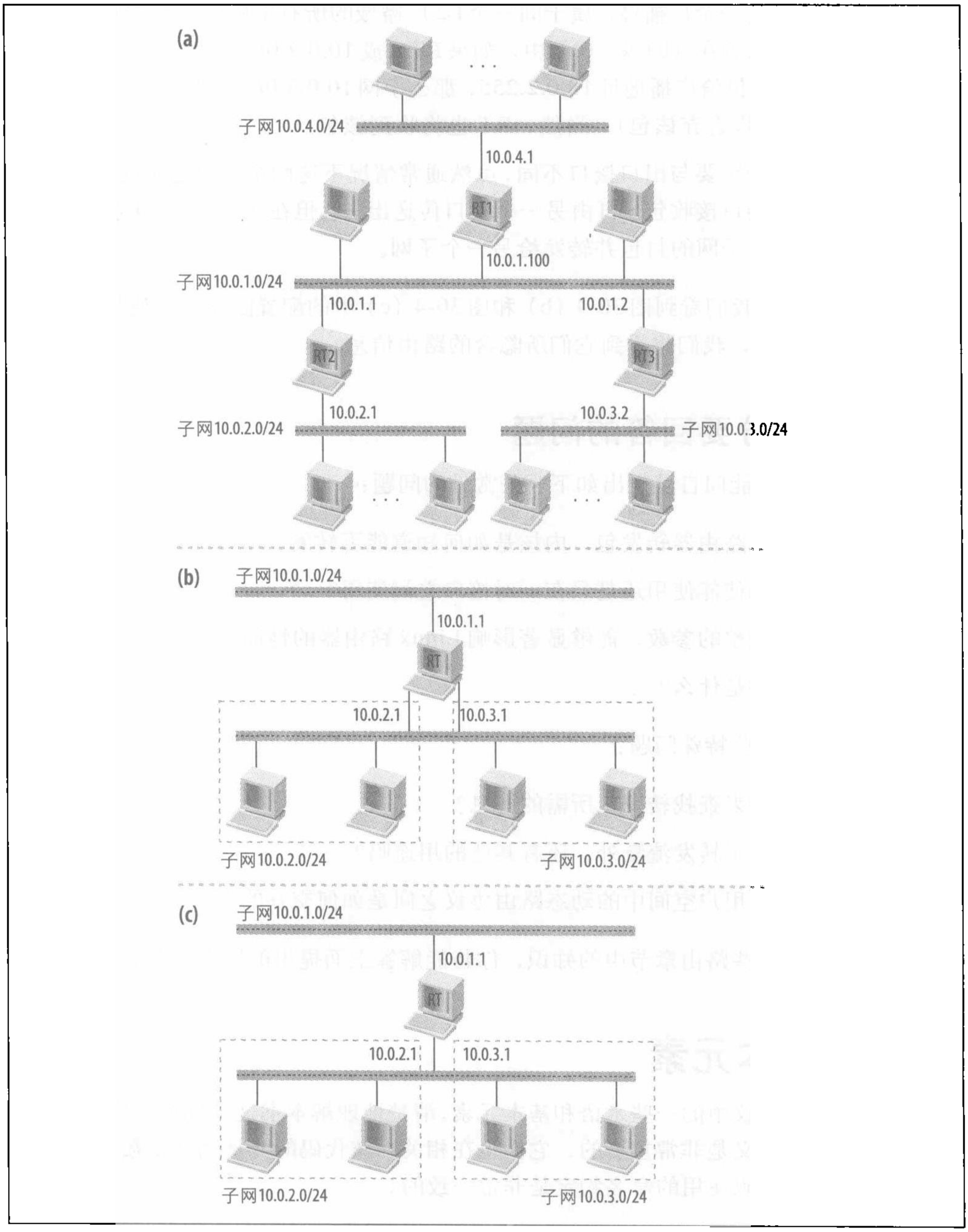


图 30-4：网络拓扑结构的例子

- 一个局域网是一个广播域。属于同一个L2广播域的所有主机会收到相互发送的广播包。这意味着在（b）和（c）中，如果RT（或10.0.2.0/24网络内的任何其他主机）发出一个包给广播地址10.0.2.255，那么子网10.0.3.0/24内的所有主机都将收到该包(虽然将丢弃该包)，当然，RT也将收到该包。
- 入口接口不一定要与出口接口不同，虽然通常情况下这两个接口是不同的。转发通常是由一个接口接收包，再由另一个接口传送出去。但在（c）中，RT利用同一个NIC接收一个子网的封包并转发给另一个子网。

在第二十六章中，我们看到图30-4（b）和图30-4（c）中的配置隐含了要使用低层的邻居协议。在本章中，我们将看到它们所隐含的路由信息。

本书这一部分要回答的问题

当你看到这时，可能向自己提出如下一些常见的问题：

- 如果假定一台路由器转发包，内核是如何知道能否转发？
- 路由是在全局使能使用还是只在一对接口之间使用？
- 是否存在可调整的参数，能够显著影响Linux路由器的性能？
- 路由表的结构是什么？

或者是诸如以下一些特殊问题：

- 利用什么算法来查找转发包所需的信息？
- 路由表除了用于转发流量外，还有其他的用途吗？
- 内核与运行在用户空间中的动态路由协议之间是如何交互的？

利用本章和下面一些路由章节中的知识，你将能解答上面提出的所有问题。

路由的基本元素

本节将介绍路由协议中的一些术语和基本元素。清楚地理解本书这一部分中广泛使用的一些关键术语的含义是非常重要的，它们也在相关内核代码的变量名和函数名中出现。幸运的是，路由代码使用的命名约定是非常一致的。

下面列出了一些经过简化的术语定义，其他的概念则会出现在各自的小节中。

互联网服务提供商 （ISP）

提供访问互联网的公司或组织。

转发信息库

简称为路由表，见前面的“路由器、路由与路由表”一节。

对称路由（Symmetric route）与非对称路由（asymmetric route）

一般来说，如果从主机A到主机B的路由与从主机B返回主机A的路由相同，这种路由被称为对称路由。但在复杂配置情况下，返回的路由可能不同，这被称为非对称路由。

Metrics

metric是在一条路由上配置的一个可选参数。不要将此处解释的metric与路由协议中使用的metric相混淆：后者用于衡量一条路由的好坏。路由协议metric的例子有端到端延迟（end-to-end delay）、跃点数（the number of hops）、配置权值（weight）或开销（cost）等。

当利用IPROUTE2来配置一条路由时，它可以提供更多的metric参数，这些参数在“路由的基本元素”一节中定义。其中包括在第十八章中描述的路径最大传输单元、或称为Path MTU。其他的metric被TCP协议用作内部变量的初值，这些值可以在以后被TCP协议改变。你可以参考任意一本关于TCP的书籍来查看下面的这些metric的含义和用途：

- 窗口（Window）。
- 往返时间（Round trip）。
- 往返时间差（Round-trip time variation）。
- 慢启动阈值（Slow-start threshold）。
- 拥塞窗口（Congestion window）。
- 通告的最大段长度（Maximum segment size）。
- 重新排序（Reordering）。

Realm

一个用数值表示的域标识。见第三十一章“基于路由表的分类器”一节。

地址分类

IP地址被划分为不同的类，如表30-1所示。表30-2则列出了每一类IP地址中，网络部分和主机部分各自的长度（注意D类地址和E类地址是C类地址的特殊情况）。

表 30-1：按类划分 IPv4 地址

类	第一个地址	最后一个地址	地址最左边的二进制位
A	0.0.0.0	127.255.255.255	0
B	128.0.0.0	192.255.255.255	10
C	192.0.0.0	223.255.255.255	110
D（多播）	224.0.0.0	239.255.255.255	1110
E（保留）	240.0.0.0	255.255.255.255	11111

表 30-2：网络与主机部分

类	网络地址部分所占位数	主机地址部分所占位数	主机数（包括网络地址与广播地址）
A	8	24	16,777,216 (2^{24})
B	16	16	65,535 (2^{16})
C	24	8	256 (2^{8})

可路由地址与不可路由地址

IP 规范中已经将一些特定范围内的地址（表 30-3 所示）规定为不可路由地址，这意味着这些地址只能保留在一个局域网内使用。可路由地址必须由中心体处理且是全球唯一的。相应的，任何人都可以配置不可路由地址，但这些人大多数都是在路由器后面的系统内使用这些地址。不可路由地址不能被用于提供任何因特网服务，因为它们不是全球唯一的，且因特网路由器不能将流量传送给它们。

子网 127.0.0.0/8 中的地址比较特殊，这些地址的 scope（注 2）就是配置它们的主机。无论这些地址是作为封包的源地址还是目的地址，该封包都不能离开主机。

表 30-3：IPv4 各类地址中的不可路由地址和回环地址

地址	类
10.0.0.0/8	1 × 类 A
172.16.0.0/16～172.31.0.0/16	16 × 类 B
192.168.0.0/16	256 × 类 C
127.0.0.0/8（回环地址）	1 × Class A

注 2：　在“Scope”一节中将介绍这些地址的确切含义。

图 30-5 中的拓扑结构包含两个使用相同的不可路由 IP 子网 10.0.1.0/24；还包含一个可路由的子网 100.0.1.0/24。如果 10.0.1.0/24 子网内的任何一个主机要与子网外部的主机通信，那么路由器必须使用某种网络地址转换（NAT）来隐藏本地不可路由的子网。而且注意每台主机都配置有默认地址127.0.0.1。在三个路由器与各自ISP的接口上配置了由 ISP 分配的可路由的 IP 地址。

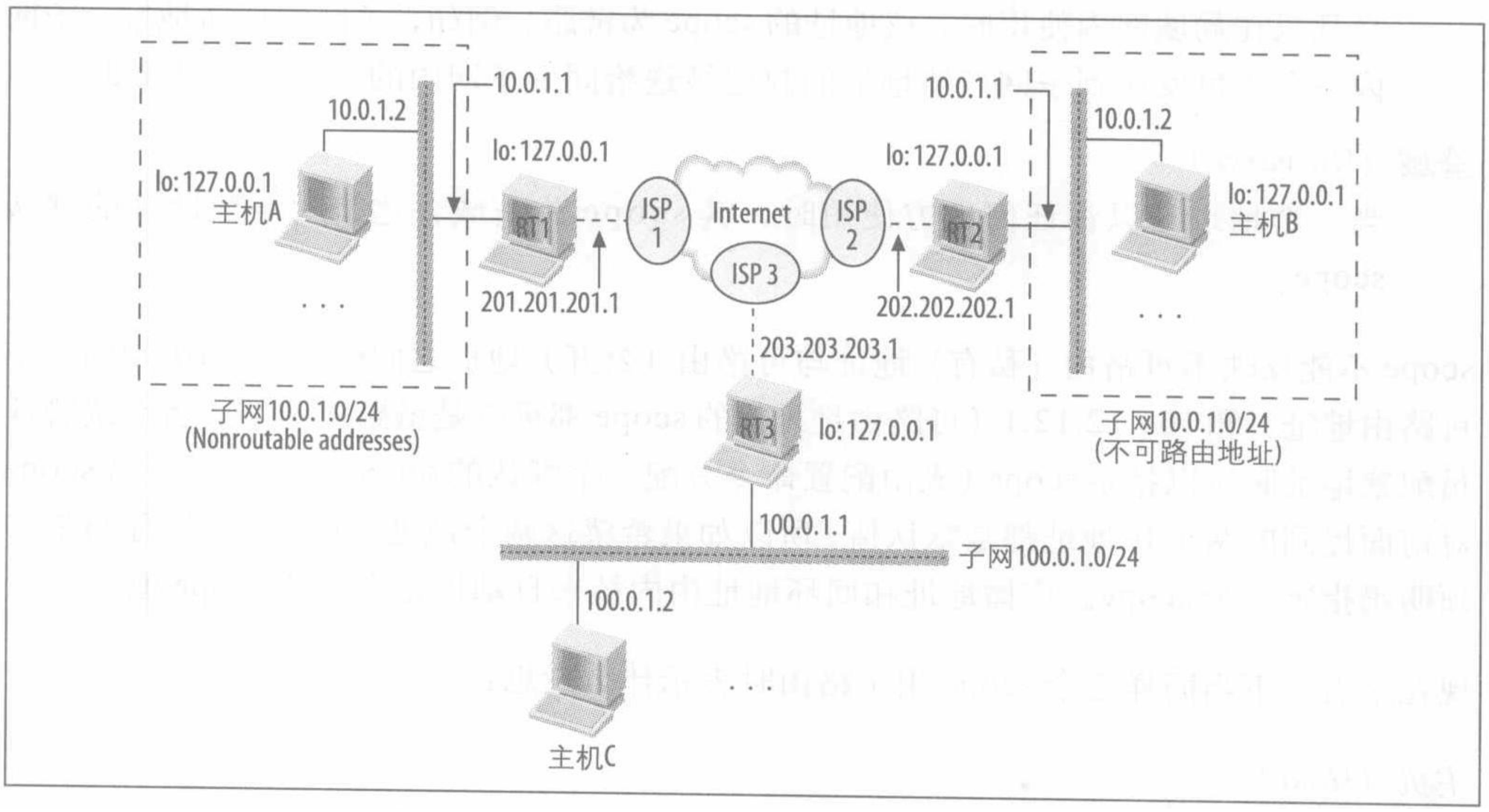


图 30-5：可路由和不可路由地址

Scope

路由和IP地址都可以指定一个 *scope*，用来告诉内核它们在哪些情况下是有意义的，是可以被使用的。如果理解了 scope 的概念，就会更容易理解路由代码中的各种合理性检查，以及各种不同 scope 的路由和不同 scope 的 IP 地址之间的区别。

在 Linux 中，路由的 scope 表示到目的网络的距离。IP 地址的 scope 表示该 IP 地址距离本地主机有多远，某种程度上也告诉你该地址的所有者距离本地主机有多远。

第三十二章提供了更详细的 scope 列表，在这里我们先来看一些例子，其中使用的术语与路由代码中的非常类似，可以更容易将代码与这些概念相关联。

我们从 IP 地址常用的 scope 开始：

主机（Host）

当一个地址只用于主机自身的内部通信时，其scope为主机。在这个主机以外不知道该地址，并且其不能被用于外部通信。例如，回环地址127.0.0.1。

链路（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scope为链路，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封包被送给同一子网内的其他主机（注3）。

全域（Universe）

当一个地址可以在任何地方使用时，其scope为全域，这是大多数地址的默认scope。

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。10.0.0.1（不可路由地址）和165.12.12.1（可路由地址）的scope都可以是链路或全域。当系统管理员配置地址时可以指定scope（或由配置命令分配一个默认的scope值）。由于全域scope对前面提到的两个IP地址都是默认值，所以如果希望这两个地址有些区别，管理员就必须明确指定一个scope。广播地址和回环地址由内核来自动设定合适的scope值。

现在来看一下当同样三个scope用于路由时表示什么意思：

主机（Host）

当一条路由表示的目的地址为本地主机时，其scope为主机。

链路（Link）

当一条路由表示的目的地址为本地网络时，其scope为链路。

全域（Universe）

当一条路由表示的目的地址超过一个下一跳时，其scope为全域。

我们将在第三十二章“添加一个IP地址”一节中看到，Linux为配置的每个本地地址都创建了一条路由，并且为每个配置子网的广播地址添加一条路由。那一节将有助于读者对IP地址scope与路由scope之间关系的理解。

scope的使用

在路由代码和内核其他部分中广泛使用IP地址scope和路由scope。

首先，记住在Linux中，虽然管理员是在接口上配置了IP地址，但这些地址属于主机而不属于接口，见第二十八章“多个网络接口的应答”一节中更详细的描述。

注3： 当然也有些例外情况，参见“定向广播”一节中的例子。

经常会见到在一台主机或是在一个接口上，或在多个接口上配置多个IP地址。当本地系统发送一个封包时，内核需要选择一个源IP地址。若主机只有一个配置了一个IP地址的NIC时，情况很简单，但是当一个复杂配置中存在不同scope的多个地址时就不简单了。根据目的地址的位置，你可能更愿意选择一个有特定scope的源IP地址，目的地可以用它来返回流量或远方站点将它用于其他目的。

路由代码也使用scopes来加强对配置进行简单而又有效的合理性检查。假定需要发送一个封包给远端主机B，它与本地主机配置的任何子网都不是直达的。路由查找将返回网关RT使用的地址。现在我们知道为了到达主机B，需要将封包发送给RT，RT将负责转发该封包。为了避免出现环路，RT必须比本地主机更靠近目的地。换句话说，到达主机B的路由scope必须比到达RT的路由scope更大(在特殊配置条件时也存在例外情况)。

下面我们看图30-6所示的例子。若主机A要访问主机B，前面提到的路由查找表返回默认的路由是通过10.0.1.1地址，其scope是RT_SCOPE_UNIVERSE。按照图中显示的主机A中的第二条路由，A通过*eth0*可以直接到达网关10.0.1.1。第二条路由的scope是RT_SCOPE_LINK，它比上一条路由的scope更窄，因此能保证接口发送的封包到达更大的scope。

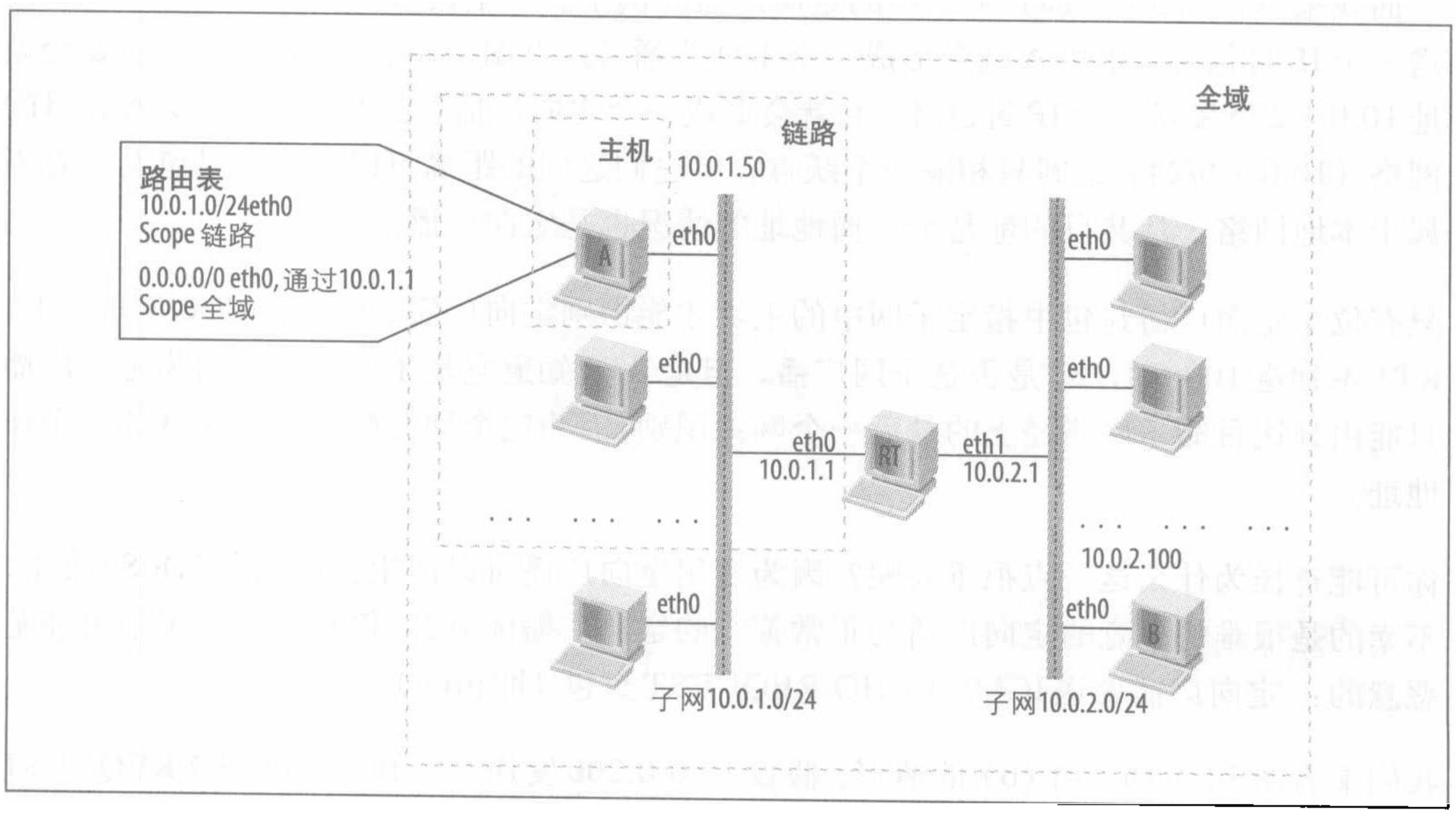


图30-6：简单的网络拓扑结构

在第三十三章的“出口查找”一节中，你可以发现ARP中使用scope的例子。

默认网关

默认网关经常作为0.0.0.0/0路由，就是当没有其他到达目的地的确定路由时使用的路由（注4）。一个连接到Internet的主机通常被配置到达本地局域网的路由器（间接源自对NIC的配置），并且也是用于访问Internet的默认路由器（通常由ISP指定）。另一方面，路由器可以配置为默认路由器，也可不是默认路由器。主要依据该路由器在网络拓扑结构中的位置和其所扮演的角色。

Linux内核没有对可配置的默认网关的数目进行任何限制，详细内容见第三十五章。

定向广播（Directed Broadcasts）

广播封包是一个发往子网广播地址的简单封包。子网广播通常由位于同一个子网中的主机发起。这就意味着广播封包的目的地是自己子网内的所有主机。

另一方面，定向广播的目的地是远端子网的广播地址。SAMBA服务器在远端子网上用于公告资源（打印机或文件夹）的远程通知功能就是定向广播的一个例子。

下面我来解释图30-7（a）中给出的定向广播的例子。当主机A要给地址10.0.0.255发送一个IP封包时，主机A就会形成一个本地广播（这点图中没有显示）。而当A要给地址10.0.1.255发送一个IP封包时，它就会形成一个定向广播。在本例中，主机A和目的网络（10.0.1.0/24）之间只相隔一个跃点，但它们之间的距离可以更大。封包发送者不属于本地网络，且其目的地是个广播地址的情况也是定向广播。

只有位于定向广播封包中指定子网中的主机才能识别定向广播。例如，图30-7（c）中，RT1不知道100.0.1.127是否是子网广播，但是RT2知道它是子网广播。因为定向广播只能由到达目的子网路径上的最后一个网关识别，因为这个网关配置有该子网的一个IP地址。

你可能奇怪为什么这一点很重要呢？因为滥用定向广播可以产生拒绝服务（DoS）攻击，不幸的是很难将恶意的定向广播与正常善意的定向广播区分开。但有一种情况很可能是恶意的：定向广播发送ICMP ECHO REQUEST封包（即ping）。

我们来看图30-7（a）和（b）的情形，假设10.0.0.200发送一个ICMP ECHO REQUEST封包到广播地址10.0.1.255（该地址不是10.0.0.200所在的子网），使用的源IP地址为10.0.0.100（该地址不是10.0.0.200自己的地址）。这将使远程子网10.0.1.0/24中的每一台主机都对该ICMP ECHO REQUEST封包做出回应，分别发送一个ICMP ECHO

注4：有些拓扑结构中默认网关是不需要的。见第二十八章中“ARP代理服务器作为路由器”一节。

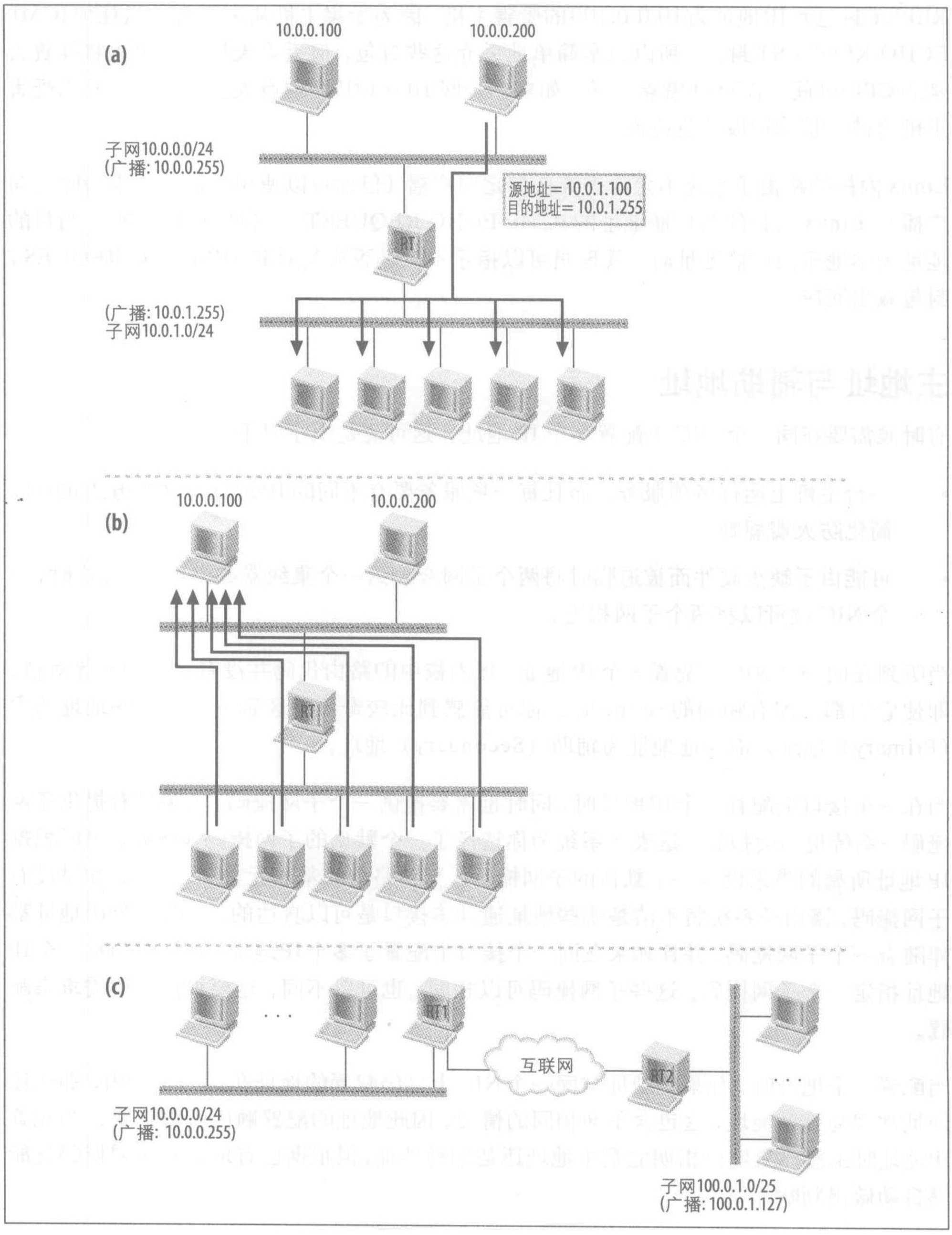


图 30-7：恶意定向广播的例子

REPLY封包到IP地址为10.0.0.100的受害主机。因为受害主机从来没有发送任何ICMP ECHO REQUEST封包，所以只是简单地丢弃这些封包，但丢弃大量的封包也将耗费大量的CPU时间。你可以想象一下，如果在子网10.0.1.0/24内有大量的主机，那么受害主机将被大量的垃圾流量淹没。

Linux内核的路由子系统不允许丢弃任何定向广播（但你可以使用过滤系统来清除定向广播）。Linux对目的为广播地址的ICMP ECHO REQUESTS封包做特殊处理：当目的地址为本地子网广播地址时，管理员可以指示主机是否应当对ICMP ECHO REQUEST封包做出回应。

主地址与辅助地址

有时候需要在同一个NIC上配置多个IP地址，这可能是出于以下一些需求：

- 一台主机上运行多项服务，而且每一项服务要有不同的IP地址。这种方式也可以简化防火墙规则。
- 可能由于缺少硬件而被迫临时将两个子网合并到一个集线器或交换机上。这时，一个NIC就可以将两个子网相连。

当听到在同一个NIC上配置多个IP地址，但内核中的路由代码并没有将它们同等对待，即使它们都分配有相同的scope时，你可能感到比较奇怪。区别就在于某些地址为主（Primary）地址，而其他地址为辅助（Secondary）地址。

当在一个接口上配置一个IP地址时，同时也需要提供一个子网掩码。如果没有提供子网掩码，系统也不会抱怨，这表示系统为你选择了一个默认的子网掩码（例如，可以根据IP地址所属的类来设置一个默认的子网掩码，见“路由的基本元素”一节）。如果没有子网掩码，路由子系统将不清楚哪些地址通过该接口是可以直达的，所以每个IP地址都伴随有一个子网掩码，并且如果在同一个接口上配置了多个IP地址，则需要为每一个IP地址指定一个子网掩码。这些子网掩码可以相同，也可以不同，这要根据具体需求来配置。

当配置一个地址时，如果该地址与同一个NIC上已经配置的地址在同一子网内，那么该地址被视为辅助地址，这包含子网相同的情况。因此地址的配置顺序就很重要：当配置IP地址时虽然没有明确指明它是主地址还是辅助地址，但根据已有地址及子网掩码就能够自动做出判断。

我们来看下面一组例子。

下面是在名为 *eth0* 的一个NIC上的配置，配置了两个子网掩码相同的地址。第一个是

10.0.0.1/24，第二个是10.0.0.2/24。这样的话两个地址落在同一个子网10.0.0.0/24内，因此第一个为主地址，另一个为辅助地址。

```
[root@router kernel]# ip address add 10.0.0.1/24 broadcast 10.0.0.255 dev eth0
[root@router kernel]# ip address list dev eth0
4: eth0: <BROADCAST, MULTICAST, UP> mtu 1500 qdisc pfifo_fast qlen 100
    link/ether 00:60:97:77:d1:8c brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.1/24 brd 10.0.0.255 scope global eth0

[root@router kernel]# ip address add 10.0.0.2/24 broadcast 10.0.0.255 dev eth0
[root@router kernel]# ip address list dev eth0
4: eth0: <BROADCAST, MULTICAST, UP> mtu 1500 qdisc pfifo_fast qlen 100
    link/ether 00:60:97:77:d1:8c brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.1/24 brd 10.0.0.255 scope global eth0
    inet 10.0.0.2/24 brd 10.0.0.255 scope global secondary eth0
```

只要你愿意，你就可以在每一个接口上配置多个主地址和多个辅助地址。对一个特定的子网掩码（本例中的子网掩码为/24），只能有一个主地址。如果我们接下来添加第三个地址10.0.0.3/24，那么它将被划分为辅助地址，与主地址10.0.0.1/24相关联。

但另一方面，10.0.0.1/24和10.0.0.3/25位于不同的子网（因为子网掩码不同），尽管两个子网覆盖的地址有部分重叠。所以，如果在前面例子中接着添加10.0.0.3/25地址，该地址将成为*eth0*的另一个主地址。下面给出*ip address list*命令的输出信息：

```
[root@router kernel]# ip address add 10.0.0.3/25 broadcast 10.0.0.127 dev eth0
[root@router kernel]# ip address list dev eth0
4: eth0: <BROADCAST, MULTICAST, UP> mtu 1500 qdisc pfifo_fast qlen 100
    link/ether 00:60:97:77:d1:8c brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.1/24 brd 10.0.0.255 scope global eth0
    inet 10.0.0.2/24 brd 10.0.0.255 scope global secondary eth0
    inet 10.0.0.3/25 brd 10.0.0.127 scope global eth0
```

在第三十五章“辅助函数”一节中，我们将看到当一个接口中存在子网重叠的多个主IP地址时，内核如何从中选择一个IP地址。

总之，决定是主地址还是辅助地址不只是考虑IP地址，还需要考虑标识子网的子网掩码。当在一个接口上配置多个IP地址时，重要的是要理解主地址与辅助地址之间的区别，这一点在阅读路由代码时也很重要。我们将在第三十二章中看到对许多事件和条件做出的响应都依赖于IP地址为主地址还是辅助地址。下面给一些例子：

- 当CPU正好运行使用IP配置的程序时，主地址可以降低其开销。
- 当删除一个主地址时，所有相关的辅助地址也被删除。但通过*/proc*可以配置一个选项，使得在当前主地址被删除时可以将辅助地址提升为主地址（参见第十八章）。
- 当主机为本地生成的流量选择源IP地址时，只考虑主地址。

老版本的配置：接口别名（aliasing interfaces）

在前面几节中你可能已经注意到，我总是使用*ip address*命令来配置IP地址。这是因为*ifconfig*是旧版的接口配置命令，它是Unix系统管理员最常使用的工具。它不区分主地址与辅助地址，甚至不显示辅助地址。所以在前面几节给出的例子中，*ifconfig*和*ip address list*命令的输出不一致。第三十六章对Linux net-tools工具包提供的*ifconfig*和由IPROUTE2软件包提供的新版的*ip address*进行了深度对比。

在引入IPROUTE2软件包及其高级路由能力之前，Linux使用接口别名（aliasing）的概念，为了保持向后兼容，在新内核中仍然可以看到接口别名。利用*ifconfig*在一个NIC上配置多个地址的唯一的方法，就是定义类似*eth0:0*，*eth0:1*等虚拟设备。每个虚拟设备都可以当作一个真正的NIC使用：你可以在它上面配置一个地址，若配置为路由则可作为一个设备等。

别名设备与主/辅助IP地址间的关系

因为内核既支持IPROUTE2的高级路由能力，也支持老式的接口别名，所以这两种模式需要共存。我们将在第三十二章中研究内核的内部细节，这里我们可以从用户空间的角度来看看这两种模式是如何共存的。

当配置一个别名设备时，仍然是根据在“主地址与辅助地址”一节中介绍的规则来设定主地址与辅助地址。但*ip address list*命令的输出添加了一个到别名设备的引用。下面的截图显示了这样一个例子。我们从一个已经配置了地址的接口*eth1*开始，在别名设备*eth1:1*上添加在同一个子网内的一个地址，然后再在另一个别名设备*eth1:2*上添加不同子网内的另一个地址。因为子网不同，所以*eth1:1*设备上的地址为辅助地址，而*eth1:2*设备上的地址为主地址。

```
[root@router kernel]# ip address list
...
11: eth1: <BROADCAST, MULTICAST, UP> mtu 1500 qdisc pfifo_fast qlen 1000
    link/ether 00:0a:41:04:bd:16 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.101/24 brd 192.168.1.255 scope global eth1
...
[root@router kernel]# ifconfig eth1:1 192.168.1.102 netmask 255.255.255.0
[root@router kernel]# ifconfig eth1:2 192.168.1.103 netmask 255.255.255.128
[root@router kernel]# ip address list
...
11: eth1: <BROADCAST, MULTICAST, UP> mtu 1500 qdisc pfifo_fast qlen 1000
    link/ether 00:0a:41:04:bd:16 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.101/24 brd 192.168.1.255 scope global eth1
    inet 192.168.1.103/25 brd 192.168.1.255 scope global eth1:2
    inet 192.168.1.102/24 brd 192.168.1.255 scope global secondary eth1:1
...
```

一个明显的问题是是否可以用IPROUTE2在一个别名设备上配置多个地址。这是不可能的，因为IPROUTE2并不像 *ifconfig* 那样将别名设备看作真正的、独立的设备：对IPROUTE2来说，别名设备只是IP地址的一个标签。

```
[root@router kernel]# ip address add 192.168.1.104/24 dev eth1:1
Cannot find device "eth1:1"
```

路由表

路由表是路由子系统的核心。简单地说，它是由一个路由数据库组成，IPv4的其他子系统通过各种函数可以使用该数据库，其中最重要的函数是路由查找。

也许你已经想到，路由并不只是由在“路由器、路由与路由表”一节中给出的基本信息组成。随着时间的推移，由于代码优化和新特性的引入，路由表中一条表项所包含的信息已经增加了许多。我们将在第三十四章看到这些细节。

在下面的几节，我们将主要看到如下内容:

- Linux是如何路由目的为本地地址的封包。
- 使用什么算法在路由表中查找地址。
- 对与匹配路由的流量，除了使用默认的转发动作外还有哪些管理动作(action)。
- 上层协议为了自己的便利，在一条路由中还保存了哪些额外信息。

特殊路由

当路由器收到一个封包时，需要决定将它提交给本地协议栈中相邻的更高层(因为本地主机是最终目的地)还是转发。一种简单方法是将所有的本地地址存储在一个列表内，对每个封包扫描该列表，作为路由查找的一部分。当然，列表不是最佳的选择，还有许多更好的数据结构可以提供更快的查找。Linux中使用一张独立的、基于Hash的路由表，该表只存储本地地址。更准确地说，在这张路由表中存储所有监听到的地址，既包括本地配置的地址，也包括子网广播地址。

这就意味着，在默认情况下，Linux使用两张路由表：

- 一张表用于本地地址。从该表中查找成功表明封包要提交给主机自己。在第三十二章中将看到内核是向该表添加数据。
- 一张表用于所有其他的路由，其数据项可由用户手工配置或由路由协议动态插入。

路由类型与动作

我们在“路由器、路由与路由表”一节中知道了一条路由的基本组成。当一个封包与路由表中的某条路由相匹配时，默认动作是根据从路由表得到的该路由的转发信息，即下一跳路由器地址和出口设备来转发该封包。

但Linux同时允许用户根据需要来定义其他动作（action）（注5），下面给出一些主要动作：

黑洞（Black hole）

与这一类型的路由相匹配的封包将被悄悄丢弃。

不可到达（Unreachable）

与这一类型的路由相匹配的封包被丢弃，并生成一个互联网控制消息协议（ICMP）主机不可达的消息。

禁止（Prohibit）

与这一类型的路由相匹配的封包被丢弃，并生成一个ICMP封包被过滤的消息。

放弃（Throw）

这一类型的路由与策略路由联合使用，策略路由特性参见第三十一章中的描述。当配置了策略路由时，若封包与这一类型的路由匹配，路由器将放弃查找当前路由表，如果还存在其他路由表时继续查找下一张表。

路由缓存

根据路由器所扮演的角色，路由表中的路由数量可以从几条到数十万条。因此，很明显应该维护一张更小的表来缓存路由查找结果，从正反两方面来说都是有益的。Linux将路由缓存分为两部分（这里所说的协议是指Ipv4和Ipv6等L3协议）：

- 一个是与协议相关的缓存。
- 一个是与协议无关的缓存，通常被称为DST。

第一部分表示缓存框架部分，其中每个元素被定义为一个由具体协议字段组成的集合。第二部分嵌套在第一部分内，只存储与协议无关的信息。与协议相关的缓存和其中与协议无关的部分都在第三十三章中描述。

注5：我们将在第三十六章中看到，只有使用新一代的配置工具IPROUTE2，才能够配置这些可选的路由类型。

我们将在第三十一章中看到，在一个Linux系统中可能创建多张独立的路由表来支持策略路由功能。但Linux中无论路由表的数量有多少，只使用一个路由缓存。如果支持策略路由，那么缓存就不能够提供任何公平性，可能有的路由表要比其他路由表使用更多的缓存项（即各个路由表在缓存中占用的空间是不均等的）。但这种方法可以确保整个系统有更大的路由吞吐量。

路由表与路由缓存

路由表和路由缓存除了在容量和结构上不同外，对象粒度（granularity）也不同。路由表使用子网即连续地址的集合，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，我们将在“查找”一节中描述。

我们来看一个例子。假定路由表中包含一条表30-4中给出的路由，它是仅有的一条到子网 10.0.1.0/24 的路由。

表 30-4：路由表项例子

目的地	下一跳地址	使用的设备
10.0.1.0/24	10.0.0.1	*eth0*

我们假定内核需要发送两个封包，分别要到地址 10.0.1.100 和 10.0.1.101。因为表 30-4 中的路由表项匹配这两种情况，因而内核使用它来路由这两个封包，这样将在路由缓存中加入两个表项，如表 30-5 所示。

表 30-5：路由缓存项例子

目的地	下一跳地址	使用的设备
10.0.1.100	10.0.0.1	*eth0*
10.0.1.101	10.0.0.1	*eth0*

当然，表30-5中给出的元素是简化版本。在第三十三章中，我们也将看到包含源地址的路由缓存项。

路由缓存的垃圾回收

垃圾回收负责清理路由子系统中不再被使用的数据结构。但是，被使用的数据结构也可以被删除。例如，在为了存储更为重要的信息而需要释放内存。垃圾回收所做的删除操作不会导致任何数据的丢失，因为所有被清理的信息可以被重建。在最坏情况下，从缓存中删除一个元素只会导致缓存查找时不命中。

垃圾回收机制有两种：

同步回收

当路由子系统认为需要释放内存时，立即进行清理。路由代码在下列两种情况下强制垃圾回收而无需等待定时器到期：

- 当一条新项需要被添加到路由缓存中，而缓存中当前的表项数量已经到达用户配置的一个阈值。
- 当邻居子系统缓存需要内存时。我们在第二十七章中知道路由缓存和邻居子系统缓存相互引用对方。创建一条新的路由缓存项将可能触发创建一条新的邻居缓存项。如果邻居协议，例如ARP，无法分配到所需的内存，那么路由子系统将强制垃圾回收来间接释放邻居协议所占用的数据结构，以帮助后者得到所需要的内存。

异步回收

为了保持缓存的容量在一个合理的范围内，使用一个周期性定时器来触发定期的清理操作。默认情况下，路由缓存项不会过期。但可能有些外部子系统会告诉路由缓存，在一定时间后过期一些缓存项。路由子系统就运行一个定时器来周期性地扫描缓存，查找以下缓存项：

- 过期且应当被删除的缓存项。
- 没有过期，但如果内核需要释放内存的可以被牺牲的缓存项。

使路由缓存项过期的事件范例

路由缓存项只有在特殊情况下才被设置为过期，这些情况包括：

- 当本地系统接收到一条ICMP UNREACHABLE 或 ICMP FRAGMENTATION NEEDED消息后，转交给ICMP层处理。这些消息告诉本地主机，它以前发送的一个封包，其长度超过了在到达目的地的途中的一台路由器的MTU。ICMP处理函数将扫描路由缓存，更新所有受影响表项的PMTU字段，并且设置这些受影响表项在一段可配置时间之后过期，默认为10分钟。

  ICMP也通知与触发该ICMP消息的封包关联的L4协议。例如，TCP协议会将这些通知用于Path MTU 发现算法，见第十八章和第二十五章中关于Path MTU 发现的更多细节。
- 当邻居协议无法解析L3地址到L2地址的映射（见第二十七章）时，或者当本地主机处于IP隧道的一端而另一端由于某种原因不可达时（例如，由于路由问题或配置错误），则认为目的IP地址是不可达的。

当一个目的IP地址被认为是不可达时，路由缓存中与该地址相关的所有项需要被刷新，因而立即被设置为过期。

合适的缓存牺牲品范例

在一些情况下，内核需要释放一些缓存表项来为新表项腾出空间，周期性定时器本身不能够保证缓存中总存在可用空间（即保持容量在某个阈值以下）。此时，主机必须删除周期性定时器没有处理的表项，虽然这些表项仍然是有效的。于是垃圾回收系统要从这些有效表项中选择牺牲品。通过选择那些只要很小的开销就可以被快速重建的表项作为牺牲品，就能减少伤害。

要被删除的表项的好的候选者是到广播地址和多播地址的路由。一般来讲，当路由子系统删除一条路由缓存表项时，可能间接删除了L3地址到L2地址的关联。在这种情况下，如果主机下一次发送数据给L3地址，邻居子系统就需要再次解析L3到L2的关联。但广播地址和多播地址解析的代价很小，因为它们不需要任何solicitation请求（见第二十六章中“特殊情况”一节）。

被删除的表项的特别差的（代价高的）候选者包括：

REDIRECT路由

这种路由是通过一条ICMP REDIRECT消息学到的。如果它被删除，主机将使用次优路由来转发封包。删除这种表项也可能浪费时间，因为主机将很有可能收到另一条ICMP REDIRECT消息来将该路由项重新插入。

管理员手工配置的路由

这些路由项是用户通过诸如*ip route get 10.0.0.1 monitor*等命令，要求内核在改变路由状态时（通过NetLink套接字）发送通知。用户可能由于某种原因认为该路由项很重要，见第三十六章中表36-11提供的更多信息。

在任何情况下，不会将引用计数非零的表项视为合适的牺牲品而删除。

查找

在“路由缓存”一节中提到，Linux同时使用路由缓存和路由表。图30-8总结了路由表查找步骤。为了简化“路由/提交（deliver）封包”过程，图中没有给出在前面“特殊路由”一节中描述的各种路由。

路由缓存查找是从一张简单hash表中寻找完全匹配项。而在容量更大、更为复杂的路由表中查找是基于最长前缀匹配（Longest Prefix Match, LPM）算法，该算法会在下面

一节中介绍。我们将在第三十四章看到，一张路由表是多种数据结构互相交织而成的。这种组织方式使LPM算法速度更快、实现更容易、处理大量路由时伸缩性更好，而且减少了共享数据结构实例的复制。

最长前缀匹配

如果到每个目的地只有一条路由，那么路由查找将很简单。只要找到一条路由，其目的子网包含要到达的目的地址，那么就完成了查找的主要部分。然而路由是一个复杂的问题，如果深入了解网络拓扑结构细节或各种特殊情况的复杂性，就会明白常见的情况是到达同一目的地存在多条路由。这些路由之间的重叠可能小到一个地址，或大到整个子网。

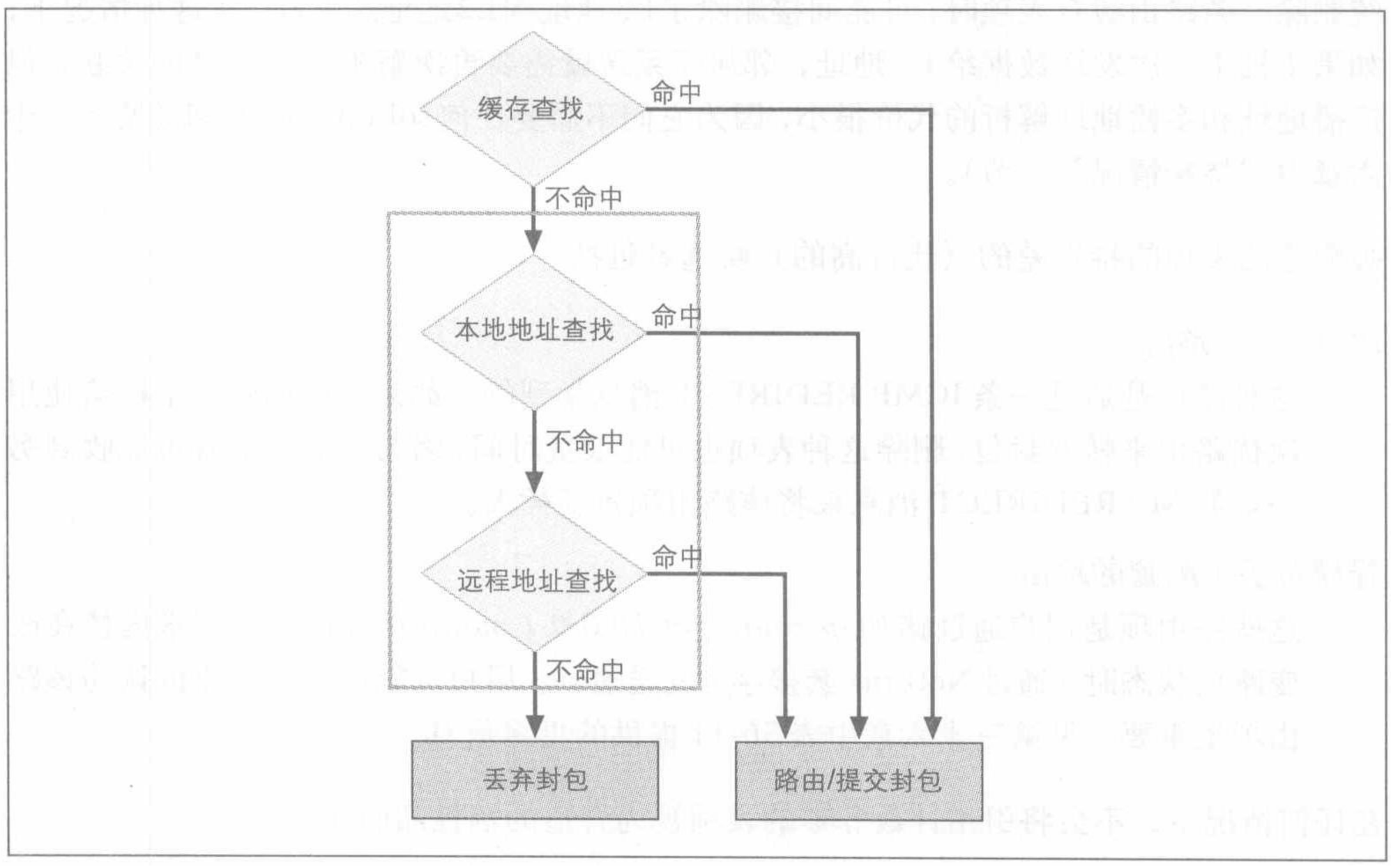


图 30-8：路由查找

在有多条路由相匹配的情况下，路由算法需要一条规则来决定应当选择哪一条作为最佳路由。这就需要用LPM算法来专门解决该问题：最具体的路由是最佳路由，也就是子网最小或子网掩码最长的路由。

我们来看一个例子，假定路由表中有两条表项与目的地址10.0.0.100匹配，如表30-6所示。

表 30-6：路由表例 1

目的地	下一跳地址	使用的设备
10.0.0.0/16	10.0.1.1	*eth0*
10.0.0.0/24	10.0.0.1	*eth1*

因为第二条路由与目的地址有24个二进制位（最多32位）相同，而第一条路由只有16个二进制位相同，那么就说第二条路由是最长前缀匹配。与表30-6给出的路由相似的情况并不是不常见，即一条路由可到达的目的地是另一条路由的一个子集。例如，由于管理或安全原因，可能需要将目的地址为某个特定子网的路由与目的是该网络中的其他部分的路由分开，这也是配置路由最简单的方法。另一种方法是将子网10.0.0.0/16划分为多个 /24 子网（即从 10.0.0.0/24 到 10.0.255.0/24），这样在路由表中有 255 条路由，其中 254 条路由的下一跳相同。但这种方式会减慢路由查找速度，耗费更多的CPU资源。

但是当某个给定的目的地存在多条匹配路由时，仅靠 LPM 算法不足以决定选择哪条路由。我们来看表 30-7 中给出的例子。

表 30-7：路由表例 2

目的地	下一跳地址	使用的设备
10.0.0.0/16	10.0.1.1	*eth0*
10.0.0.0/24	10.0.0.1	*eth1*
10.0.0.0/24	10.0.0.2	*eth1*

在这种情况下，存在两条路由有着相同的前缀长度。

我们将在第三十五章中看到路由查找搜索关键字中包含服务类型（Type of Service，TOS）：这意味着在有相同的前缀长度时可以使用TOS来区分获胜者。

当根据 TOS 也无法选出路由时，优先级更高（优先级值较低）的路由被选中。

如果根据优先级也无法选出路由，内核就简单选择第一条。这意味着到同一目的地的路由，还与有相同前缀长度的路由被添加到路由表中的顺序有关。

封包接收与封包传输

因为路由类型可以是本地提交或转发，所以路由要发出的封包和要接收的封包时都需要路由表。路由表除了这两个显而易见的应用外，还有其他用处。

图 30-9 给出了一些路由表使用的例子。由接收数据触发的路由查找（图中左边）和由于传输数据触发的（图中右边）是有区别的。当系统第一次收到一个输入封包时，要收集转发该封包所需的路由信息，这也是图30-9中转发框没有到路由框的箭头的原因。该图中也包括一些章节号，指出了可以去相应的章节去找对应的内核组件的细节。下面是图中一些相关的行为的细节介绍：

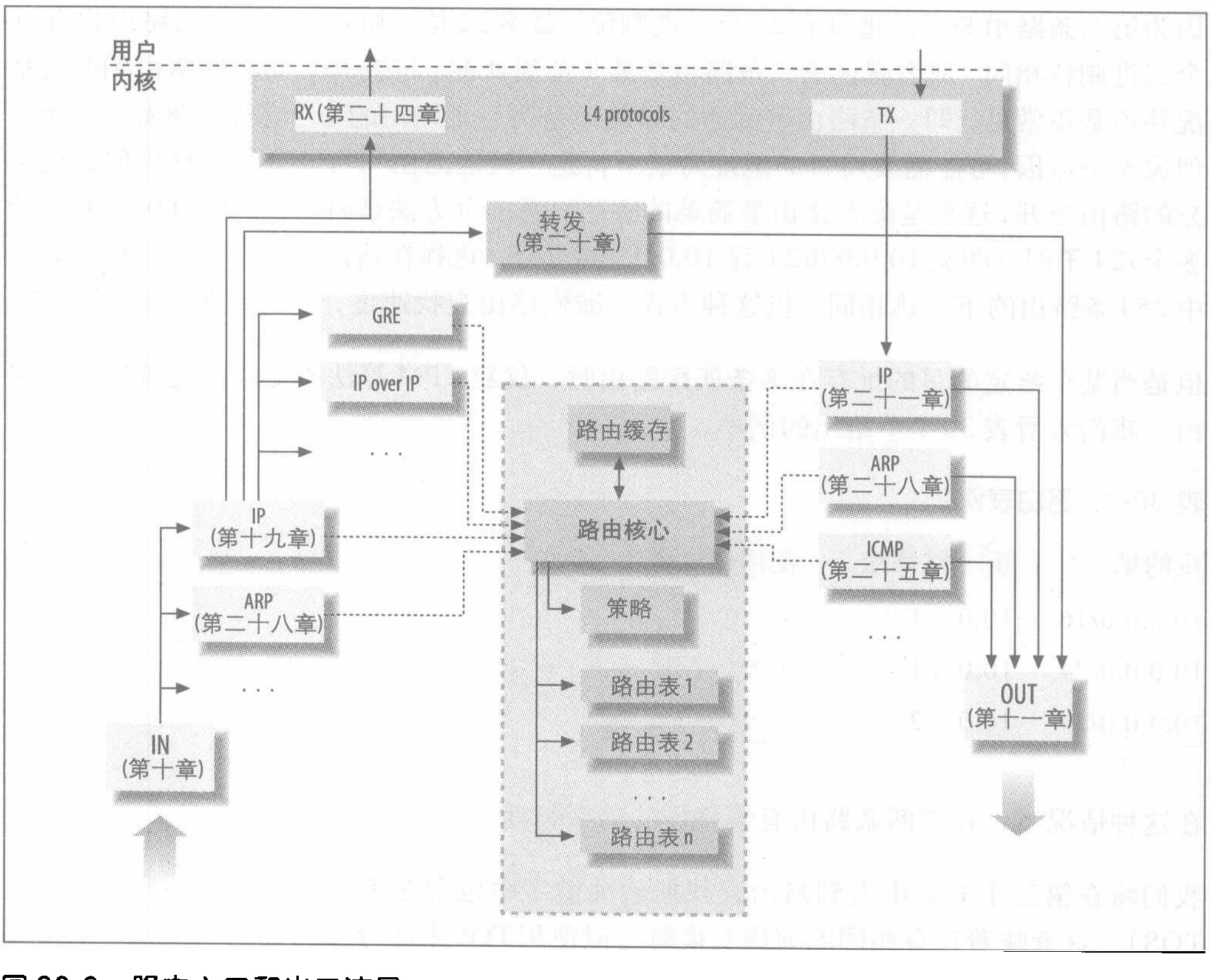


图 30-9：路由入口和出口流量

- 地址解析协议（ARP）封包不进行路由，但是 ARP 需要查找路由表来强化合理性检查，见第二十八章。
- IP-over-IP 是一个简单的隧道协议，它可以将普通 IP 封包封装到一个更大的 IP 封包中。当IP处理部件对一个入口的IP-over-IP封包处理时，它会取出其中的有效数据送给 IP 层。内部的普通 IP 封包可以像其他的入口封包一样进行路由，这种情况下，路由子系统就多做一次路由查找。

- 通常路由一个封包仅需要进行一次查找，而不管这个封包来自何处（本地主机或外部网络）。路由查找能返回路由该封包所需的所有信息，包括实现路由功能的内核函数。一些例外情况可能要进行多次路由查找：就像刚才IP-over-IP协议描述的那样，有时出于某些原因要做额外的路由查找。
- 路由系统核心首先会检查路由缓存，看其中是否包含了需要的信息，如果没有，就要返回到路由表进行查找。

第三十一章

路由：高级路由

前一章简要介绍了路由的基础知识，本章介绍在更为复杂的环境下，使用策略路由与多路径路由等特性来配置路由。本章也将介绍路由是如何与由 QoS 负责的流量控制子系统、以及防火墙程序（Netfilter）相互作用。在本章末尾介绍两个小特性：ICMP 重定向与反向路径过滤（reverse path filtering）。

策略路由背后的概念

我们在第三十章“特殊路由”一节中看到，Linux内核在默认情况下使用两张路由表：一张表用于本地路由，另一张可以由管理员来配置。如果内核编译时支持策略路由，那么可以有多达255张不同的、相互独立的路由表。在本章中，我们将看到如何使用策略路由。在第三十五章中将看到它对路由子系统设计产生的影响。

策略路由背后的主要思想是允许用户除了可以根据目的IP地址配置路由外，还可以根据其他多个参数来配置路由。

在因特网繁荣稳定地发展的这些年中，大多数路由器被配置为只能够根据封包内的目的IP地址来路由封包。（为了简化，这里不考虑跨越多个ISP或多个国家等因素）。只基于目的地址（以及一些额外的配置参数）的路由，使路由表在面临大多数情况时都是最优的。

但商业界还需要考虑更多的因素，例如，出于安全或计费考虑需要分开流量，或者通过单独的路由发送实时数据流。这就出现了策略路由，由于路由可以有各种各样的标准，在本章中，我们将除基于目的地址外的路由都认为是策略路由。

使用策略路由的一个例子是，一家ISP需要根据原有客户或基于服务质量（QoS）需求来路由流量。通常根据ISP路由器上流量的进入端口、源IP地址、或二者的结合，很容

易识别出原有的客户。路由器也可以基于源地址与目的地址的组合来标识一组流量或统计从某个源地址来的总流量。QoS需求可以根据IP头中的DiffServ Code Point (DSCP)字段和更高层头部多个字段的组合来识别（这些可以识别出上层应用）。

因为本书主要是讨论内核的内部细节，我们想知道这些策略是如何传送给内核，它们在路由表中是如何实现，以及其对于路由查找的影响。我们将学习到所有这些内容，首先来看一个例子，参考图31-1中给出的拓扑结构。

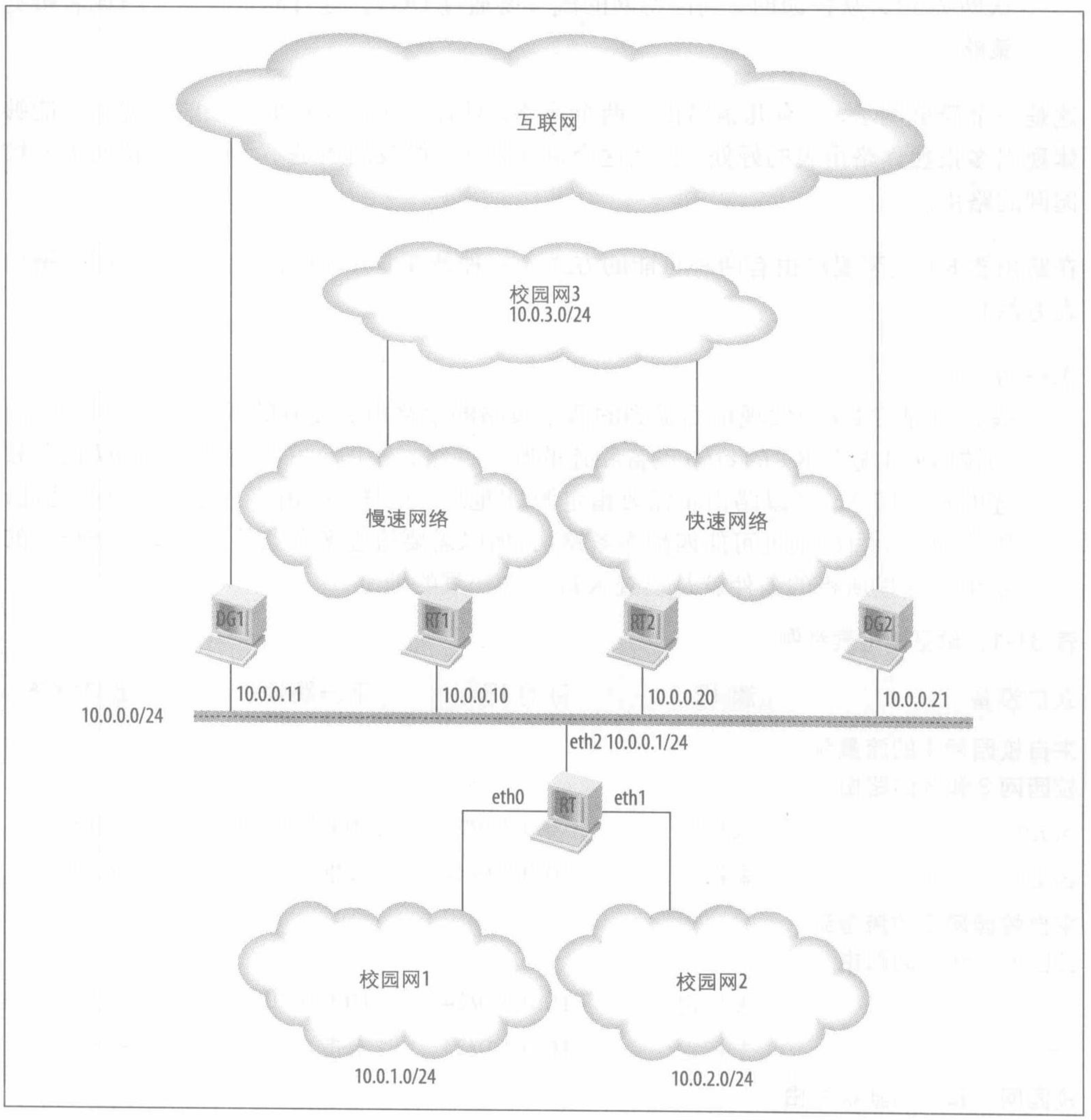


图31-1：可能需要策略路由的拓扑结构范例

注意路由器RT的配置，它用于将校园网1和2连接到校园网3和因特网（这里仅是一个例子，不关心路由器是如何来转换不可路由地址10.0.*x*.*x*）。假设需要执行以下两个策略：

- 从校园网1到校园网3的流量要通过路由器RT1，从校园网2到校园网3的流量要通过RT2。这样做的原因可能是校园网2的管理方愿意多花点钱，从而可以使用更快速的网络来连接RT与校园网3。
- 从校园网1到因特网的流量（例如，目的地址不是这三所校园网）要经过DG1（默认网关1），从校园网2到因特网的流量要通过DG2。这可能是根据安全或者带宽策略。

这是一个简单例子，只有几条路由和两个策略。只有在非常大且更复杂的环境中才能够体现出多张独立路由表的好处。虽然这个例子较小，但我们还是不考虑从因特网进入校园网的路由。

在路由器RT上配置路由有两种可能的方式，一种是在Linux中使用的方法（即多路由表方法）。

单路由表法

表31-1是在RT中实现前面提到的两个策略的单路由表配置的简化版本。注意，因为校园网1是与RT的*eth0*设备相连的唯一网络，校园网2是与RT的*eth1*设备相连的唯一网络，所以路由不需要指定源IP地址。这样，路由不只是基于目的地址，因为同一个目的地址可能匹配多条路由，所以需要检查多个条件来选择一条唯一的路由。这里所指的条件就是检查入口设备和目的地址。

表31-1：单张路由表举例

入口设备	源IP	目的IP	下一跳地址	出口设备
来自校园网1的流量到校园网2和3的路由				
eth0	未指定	10.0.3.0/24	10.0.0.10 (RT1)	*eth2*
eth0	未指定	10.0.2.0/24	未指定	*eth1*
来自校园网2的流量到校园网1和3的路由				
eth1	未指定	10.0.3.0/24	10.0.0.20 (RT2)	*eth2*
eth1	未指定	10.0.1.0/24	未指定	*eth0*
校园网1和2的默认路由				
eth0	未指定	0.0.0.0/0	10.0.0.11 (DG1)	*eth2*
eth1	未指定	0.0.0.0/0	10.0.0.21 (DG2)	*eth2*

多路由表方法

因为在每次路由查找时都可能要检查多个条件，所以主机维护多张独立的路由表，根据特定的条件选择正确的路由表，可以更加快速便捷地查找出路由。例如，可以利用源 IP 地址或入口设备来选择一张路由表，该表中可能包含更多的条件来帮助选择最终的路由。

这样，当使用多路由表时，内核在路由查找之前需要选择出正确的路由表，这里就要利用策略（policies）来选择路由表。因而在我们例子中，可以根据以下规则，以及表 31-2 和表 31-3 来确定路由器 RT 上的路由选择：

- 从 eth0 进入的流量检查路由表 1（表 31-2）。
- 从 eth1 进入的流量检查路由表 2（表 31-3）。

表 31-2：RT1 被用于路由来自校园网 1 的流量

目的 IP	下一跳	出口设备
10.0.2.0/24	None	***eth1***
10.0.3.0/24	10.0.0.10 (RT1)	***eth2***
0.0.0.0/0	10.0.0.11 (DG1)	***eth2***

表 31-3：RT2 被用于路由来自校园网 2 的流量

目的 IP	下一跳	出口设备
10.0.1.0/24	None	***eth0***
10.0.3.0/24	10.0.0.20 (RT2)	***eth2***
0.0.0.0/0	10.0.0.21 (DG2)	***eth2***

在表 31-2 和表 31-3 中的第一条路由表项不需要用户直接配置，因为它们分别是由内核根据接口*eth0*和*eth1*上的配置派生出的。我们将在第三十二章看到这些表项是如何产生的。

我们将在第三十三章中看到，Linux 只维护一张路由缓存，该缓存由所有的路由表来更新。这些路由表也共享用于构建这些表而分配的内存池。Linux 并不强加任何公平机制，要求各个路由表必须平等地共享这些公共资源。除了可以简化实现，这种做法实际上使整个路由的吞吐量最大化，因为路由表的需求越高，所分配的系统资源也就越多。但是，这可能使外部明显感觉到差异：即当一台 Linux 主机使用不同的路由表管理来源不同的流量时，客户体验的感觉可能不同于由独立路由器所带来的感觉，甚至不同于由一台平等分配资源给路由表的主机所带来的感觉。

策略路由查找

当使用策略路由时，为目的地查找路由可分为两个步骤：

1. 根据配置策略选择要用的路由表，这个额外任务不可避免地增加了路由查找时间。
2. 从选择的路由表中查找路由。

当然，内核在执行这两步骤之前，总是先在路由缓存内查找。

策略可以像路由一样指定一个管理类型（参见第三十章“路由类型与动作”一节）。这可以使内核根据策略类型来快速决策，而不必等着查找路由。例如，当匹配的策略配置的类型为UNREACHABLE时，内核生成一个ICMP HOST UNREACHABLE消息，而不用等待去查找与UNREACHABLE类型相匹配的路由项。

图31-2是第三十章中图30-9的一个修订版本，添加了策略路由的支持，以及与各种策略类型相关的细节。

选择路由表

内核可以基于以下参数作为在选择路由表时所用的策略：

源IP地址与/或目的IP地址

可以既指定源IP地址，也指定目的IP地址，每个地址有相应的子网掩码。

入口设备

在某些情况下，接收设备可能是比源IP地址更适合的路由策略标准。虽然存在同一个源IP地址可能从多个接口进入的情况，但我们还是愿意配置为根据接收设备来选择。例如，从某一个设备进入的流量被认为是实时，有更高优先级，这时源IP地址就无法发挥作用。在以下这些情况下使用设备而不是源IP地址更为可取：

- 当同一个设备上有多个、范围不连续的源IP地址，我们希望将这些地址与同一张路由表相关联。这时并不对每个不同范围的IP地址添加一条规则，而是基于设备使用一条规则来简化配置。
- 当选择路由表时，针对物理网络拓扑要比流量来源做更多处理时。

TOS

与基于流量的源地址与目的地址参数相比，使用TOS参数有助于划分流量类型（例如，大批量数据、交互式等）。

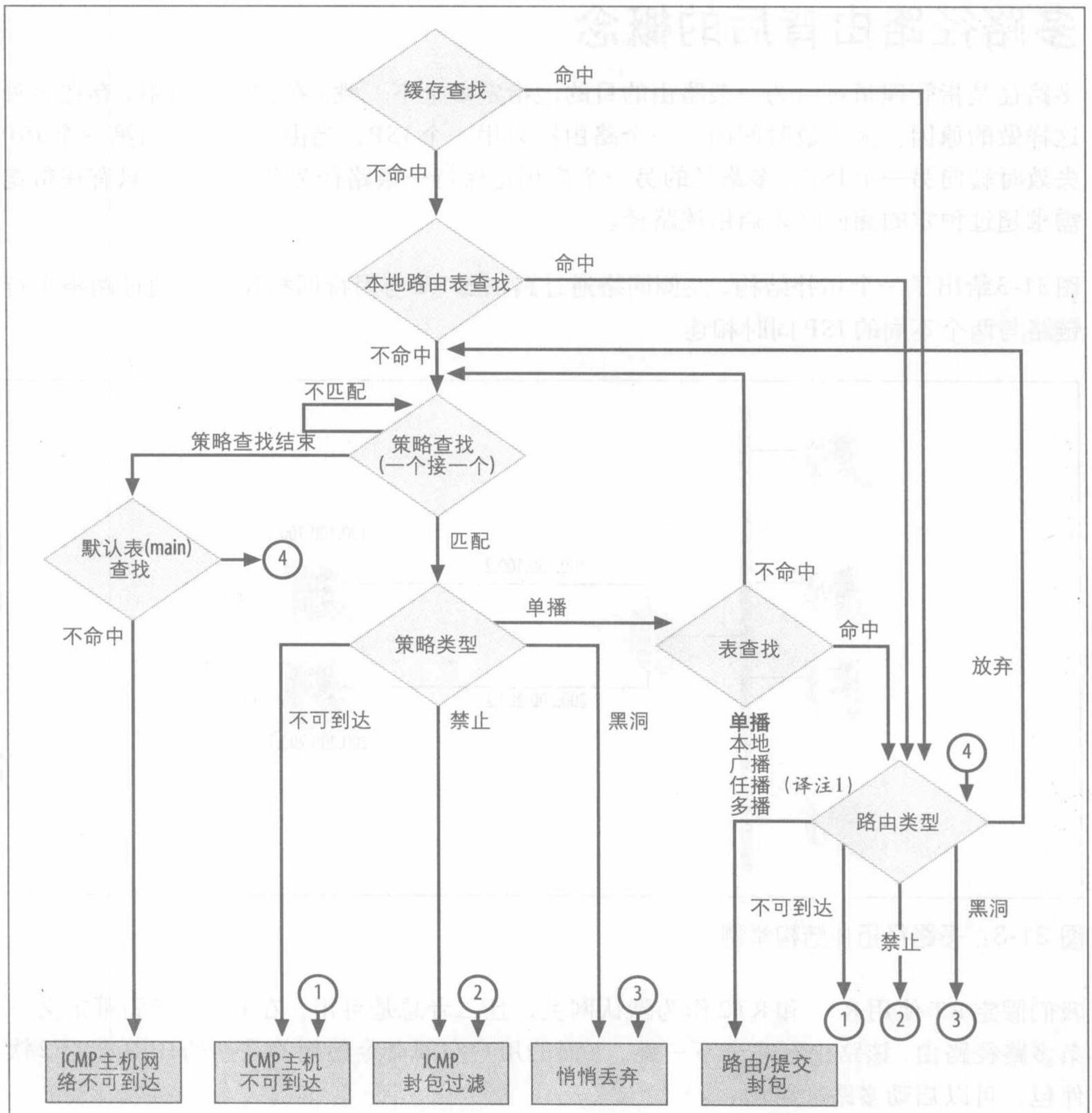


图 31-2：策略路由查找

Fwmark

这是一个表示Linux防火墙威力的一个特性。可以根据防火墙分类来定义策略路由规则。当然，要使之成为可能，防火墙就要在流量进入路由系统之前进行分类。参见“策略路由与基于防火墙的分类器”一节。

前面提到的这些参数可以任意组合来确定路由策略。

译注 1： anycast 是 IPv6 的一种新的网络服务，有些文中译为“任播”、“泛播”、“选播”等。

多路径路由背后的概念

多路径是指管理员可以为一条路由的目的地指定多个下一跳。在实际环境中，存在多种这样做的原因。大多数时间内，一个路由器只用一个ISP，当由于某种原因第一个ISP失效时转向另一个ISP。多路径的另一个应用是保持一条路径为备用状态，只有在带宽需求超过预定的阈值时才启用该路径。

图31-3给出了一个拓扑结构，左侧网络通过路由器RT与因特网相连，RT通过两条上行链路与两个不同的ISP同时相连。

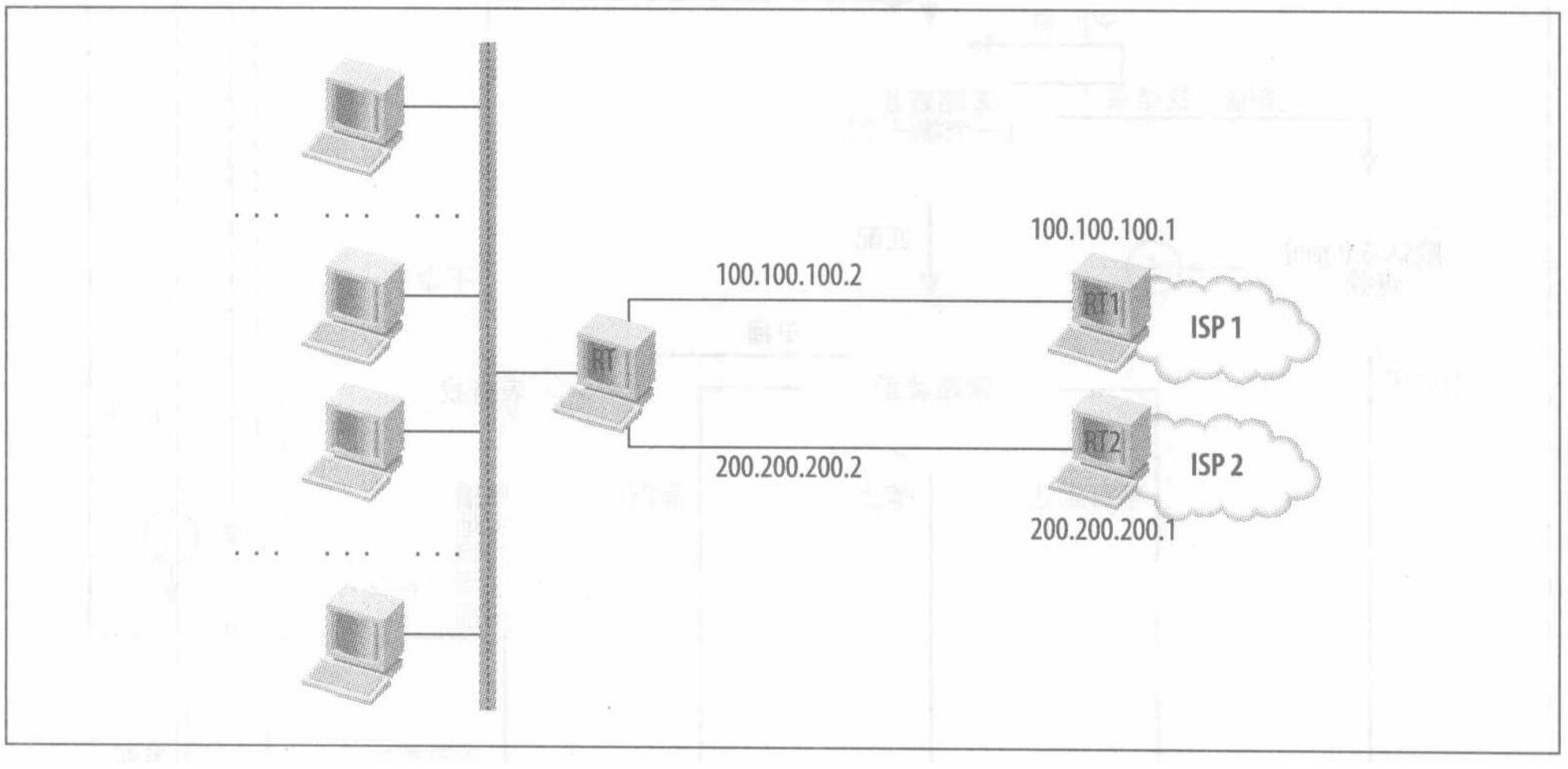


图 31-3：多路径拓扑结构举例

我们假定RT使用RT1和RT2作为默认网关，且二者总是可用。在RT上我们可定义一条多路径路由，该路由有多个下一跳。下面的用户空间命令使用的是新的IPROUTE2软件包，可以启动多路径路由：

```
ip route add default scope global nexthop via 100.100.100.1 weight 1 nexthop via
200.200.200.1 weight 2
```

注意，虽然这条路由有多个下一跳，但仍然被视为一条路由。因此，给定了一条有多个下一跳的路由（在我们例子中，默认路由为0.0.0.0/0），在每次路由查找与该路由匹配时，内核需要一种机制来选择下一跳。这有多种方法来实现，但每一种方法有各自的优缺点。对多路径路由常用算法分析感兴趣的读者，建议阅读RFC 2991和2992。

Linux允许管理员通过weight关键字为每个下一跳分配一个权值，从而为选择算法提供了很大的灵活性。下一跳被选择的次数与其权值对所有其他下一跳的权值比例成正比。

如果所有的下一跳分配的权值相同，那么，该算法就成为所谓的等价多路径算法（equal cost multipath）。

但要注意，在下一跳之间分配流量的粒度不是以封包数来衡量，而是以路由缓存表项的数量来衡量。这是因为一旦选择了一个下一跳，就向缓存内添加一条表项。因为路由子系统在查找路由表之前总是先查找缓存，所以属于该流（相同流量的集合）的后续封包将直接从缓存中处理。就像在第三十六章所解释的那样，一个流（flow）是符合一组标准的封包集合。这些标准主要由源地址或目的地址、入口设备或出口设备、IP TOS 字段等组成。在“基于流、基于连接和基于封包的流量分配”一节中，你将看到当缓存启用多路径时，流量也可能基于每个连接而不是基于每个流来分配。

从纯粹的吞吐量来看，这种粒度可能不是最理想的，因为不同的流可能有不同的带宽需求，因而当所有的下一跳配置为相同的权值时，内核可能造成不公正，更为糟糕的是，这种不公正可能是不确定的。所以Linux提供了一个选项，允许以每个封包为粒度，而不是以每个流为粒度（参见“均衡算法”一节）。但大多数情况下，由于通常有大量的流经过一个路由器，平均来说，下一跳分到的负载与其权值成比例关系。

选择下一跳

根据加权循环算法（weighted round-robin algorithm ）来选择下一跳。

在前一节的例子中，我们看到使用一个用户空间命令可以为每个下一跳指定一个权值。管理员通常为每一条路径指定一个权值来表示其优先程度。该权值在循环算法中使用。管理员根据带宽、开销等因素确定权值，所以这里不详细讨论。

选择下一跳最容易的方法是，根据它们的权重比例，每个下一跳一个接一个消耗掉它所占用的权重值，然后重新开始新的循环。例如，如果有两个下一跳，权值分别为3和5，我们将选择第一个下一跳三次，然后是第二个下一跳五次，接下来又是第一个下一跳三次等。但采用这种方法来分配流量可能导致流量的突然增大。

因此，Linux 添加了一个随机因子来选择下一跳。假定第 i 个下一跳的权值为 Wi，所有下一跳的权值总和为 W，Linux 每 W 次随机选择一个下一跳，每个下一跳被选择次数等于其权值 Wi。这种随机性的引入不是很准确，但它是一种可接受的近似方法。若所有下一跳分配的权值都为 1，就相当于简单的连续选择（从第一个到最后一个下一跳）。

下面给出加权循环算法的实现。内核定义循环预算为所有下一跳的总和。每个下一跳的预算初始值为它的权值。在每个循环，内核在0到总循环预算之间生成一个随机数。然后它遍历下一跳列表，直到找到某个下一跳的预算大于或等于该随机值。在一个下一跳被选择后，将循环预算和被选中下一跳的预算都递减。

注意，在第一次循环时可能找不到匹配的下一跳。假设有三个下一跳，权值分别为1、2、3，那么总预算为6。有效的随机值的范围为0～5，但随机值4和5将不会选出下一跳，因为每个下一跳的预算值都没有这么大。当出现这种情况时，内核从总预算中减去每个不匹配下一跳的权值，然后再次检查。

我们继续前面的例子来看该算法如何运行。假定生成的随机数为5。我们开始遍历下一跳列表。第一个下一跳的预算为1，该值不够大，因此不被选中。我们降低我们的需求，将随机值降为为5−1，即4。接下来的下一跳预算为2，该值又不够大。因此我们再次降低随机值为4−2，即2。最后一个下一跳的预算为3，比2大，因而被选中。这从性能上讲是最坏情况：最后一个下一跳被选中。

一条多路径路由的某些下一跳可能会暂时不可用（参见第三十五章“多路径对下一跳选择的影响”一节）。当然，这些节点不会被下一跳选择算法考虑。

支持多路径缓存

默认情况下，路由缓存不支持多路径。所以，我们在“多路径路由背后的概念”一节中曾看到，一旦使用“选择下一跳”一节中的算法选中一个下一跳，该下一跳将被随后与该相同查找关键字匹配的所有流量使用，因为一条引用该下一跳的路由已经被添加到缓存内。

Linux内核从2.6.12版本开始，有一个选项可以使用户激活缓存支持多路径，而且允许系统管理员针对一条多路径路由，选择一个算法在此路由的不同下一跳之间分配流量。

下面给出有用的算法：

随机算法（Random）

随机选择使用的下一跳。该算法速度很快，因为无需保持任何状态信息。平均来讲，该算法在每一个下一跳间分配的流量相同。

加权随机算法（Weighted random）

下一跳被指定一个权值，该算法根据这些权值比例随机把流量分配到所有的下一跳。

循环法（Round robin）

标准的循环算法，按照顺序将每个下一跳分配给下一条路由。

设备循环算法（Device round robin）

不是基于路由来分配流量，而是在接口上按照循环方式来分配流量。共享同一个设备的多个下一跳被视为一个单元。

当利用 IPROUTE2 中的 *ip route* 命令来配置路由时，可以用新的 mpath 关键字来选择一个算法。下面给出的是使用循环算法配置路由的例子：

```
ip route add 10.0.1.0/24 mpath rr nexthop via 192.168.1.1 weight 1
                                 nexthop via 192.168.1.2 weight 2
```

当没有指定 mpath 关键字时，该路由不支持多路径缓存。

在下面一节中，将详细讨论加权随机算法和设备循环算法。

加权随机算法

假定我们有一条多路径路由，它有四个下一跳，权值分别为 1、1、2、4。我们在图 31-4 中将四个权值沿一条线排列。权值总和为 8，所以如果产生一个在范围 0～8 内的随机数时，你就可以毫不犹豫地从该线段中标识出下一跳，值 2.8 将选择第三个下一跳。很显然下一跳被选择是与其权值成比例的。

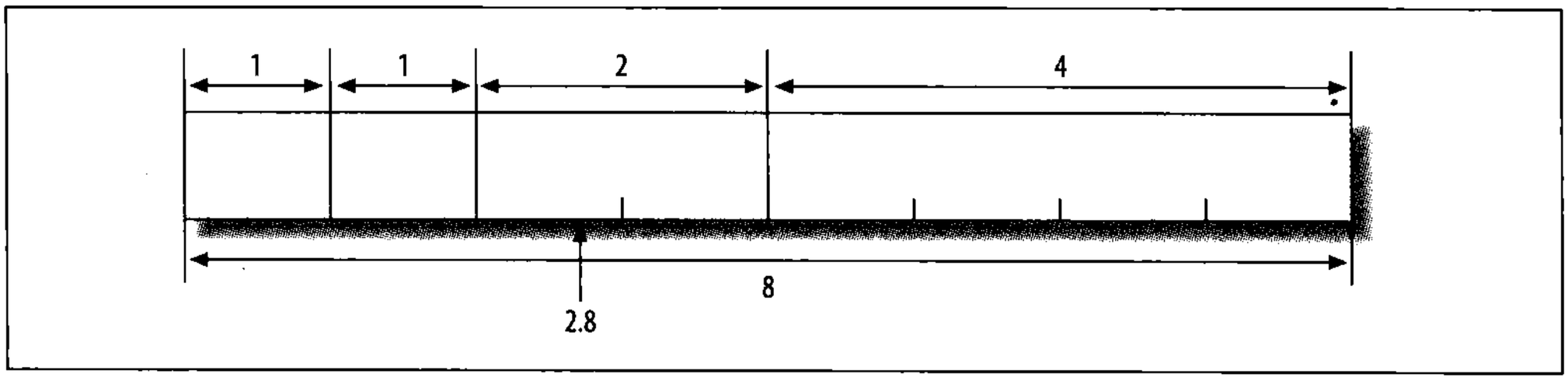


图 31-4：加权随机选择例子

设备循环算法

一条多路径路由的所有下一跳可以通过一个设备到达，每个下一跳也可以是通过一个不同的设备到达，也可以是两种情况的复合。这三种情况在图 31-5 中给出。

一个纯粹的循环算法将流量平均分配给下一跳，但对与这些下一跳相关联的设备则不一定是平均的。例如，在图 31-5（c）中，一条多路径路由有三个下一跳，其中两个下一跳共享一个出口设备，这将使两个出口设备中的一个设备分配的流量是另一个的两倍。

因而，设备循环算法就是要将流量在一组设备之间平均分配，而不是基于每条多路径路由来分配流量。对配置了使用该算法的路由项，与这些路由项匹配的流量被视为单个流量集，它们在设备间被平均分配。所以，对于一条给定的多路径路由，决定使用哪个设备不仅依赖于该路由项流量以前所用的设备，而且也依赖于其他多路径路由项流量所用的设备。

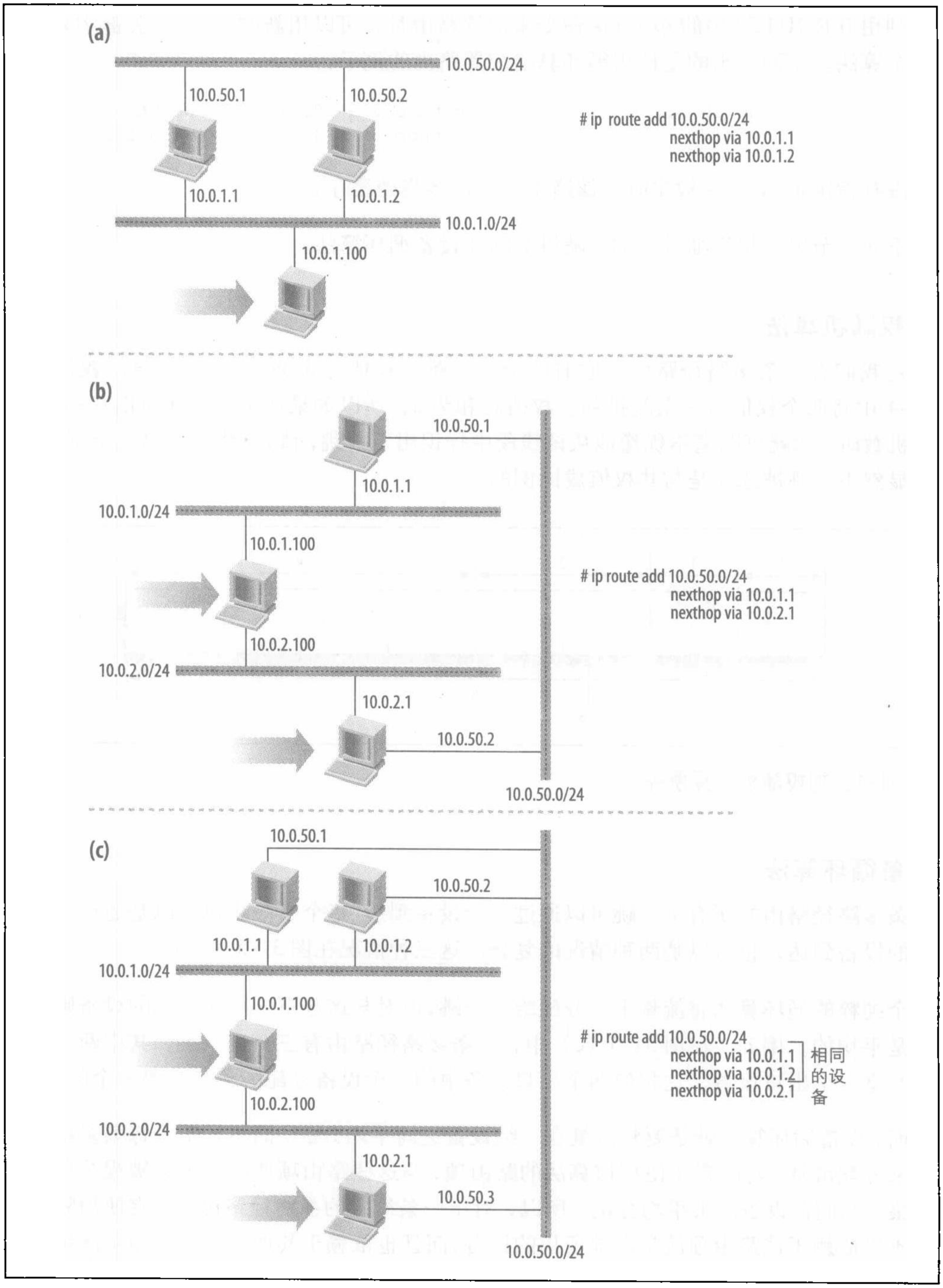


图 31-5：给接口分配下一跳的几种不同方式

注意，虽然一个纯粹的循环算法假定转发路径中的瓶颈是目标路由器的CPU，但设备循环算法目标是优化设备带宽的使用，以及降低目标CPU的重要性。

基于流、基于连接和基于封包的流量分配

给定一条多路径路由，对于与该路由项匹配的流量，在下一跳之间分配时可以基于流、基于连接或基于封包：

基于流 Per flow

根据每个源IP地址和目的IP地址的唯一组合来选择下一跳。所以，在主机之间存在多个连接时，将只能选择一个下一跳。

基于连接 Per connection

每当启动一个新连接时选择一个下一跳。这意味着在主机之间的多个连接可以分配给多个下一跳。标识一个连接一般需要5个元素：源IP、目的IP、L4协议、源L4端口和目的L4端口。

基于封包 Per packet

根据每个封包来选择下一跳。属于相同连接的封包可以分配给多个下一跳。

为了使像TCP等面向连接的协议工作正确，需要基于连接和基于流来分配流量。而基于封包的流量分配能与类似用户数据报协议（UDP）等无连接协议配合很好。

当不支持多路径缓存时，Linux在多路径路由的不同下一跳之间，总是基于流，并按照流的权重比例来将分配流量，参见“多路径路由背后的概念”一节。

当启用多路径缓存时，根据流量来源不同来分配：

本地生成的流量

本地生成的流量利用在“支持多路径缓存”一节中给出的算法，基于连接来分配。

需要转发的入口流量

这些流量的分配就像不支持多路径缓存一样：总是使用第一条匹配的缓存路由项。在任意给定连接上的IP封包到达目的主机时的顺序可能出错，这种方式对于减少的这个可能性是很有必要的。

均衡算法

Linux内核曾经提供基于封包的分配，称为均衡（equalization）。下面给出的例子中，一个命令（当前的Linux内核还没有实现该选项）可通过`eql`选项来使路由均衡化：

```
# ip route add eql 100.100.100.0/24 nexthop via 10.0.0.2 nexthop via 10.0.0.3
```

```
# ip route list
100.100.100.0/24 equalize
    nexthop via 10.0.0.2  dev eth0 weight 1
    nexthop via 10.0.0.3  dev eth0 weight 1
...
```

自从发布支持该选项以来已经过去了很长时间，但可能因为没有需求，因而该选项以后不大可能被添加到内核中。

与其他内核子系统的交互

从一个封包出现在系统中，这时它是从某个接口收到或是本地生成，到它被提交到下一跳（如果转发）或是提交到本地（如果目的地为本地主机）这一段时间内，可能有多个网络子系统要处理该封包。这其中就包括防火墙子系统和流量控制子系统。这两个子系统可以根据各种信息数据库将流量分类，并将分类结果存储到缓冲描述符的一个字段。路由子系统也可以将流量分类并将结果存储到缓冲描述符。

图 31-6 给出的是一张简化图，描述了路由、防火墙和流量控制三个子系统之间如何交互，以及一个子系统何时出现在图中。这张图给出了一个入封包是如何通过各种子系统，以及如何初始化其防火墙标签（tag）和路由标签的。

在接下来的几个一节中，我们将进一步描述策略路由和防火墙是如何计算各自的标签，并将它们提供给其他内核子系统使用。

基于路由表的分类器

在流量控制子系统可用的许多分类器（classifier）中，有一种称为基于路由表的分类器，它能够基于 *realm* 来将路由分类。Realm 是可以在策略和路由上指定的数字标签。每一条路由和每一个策略最多可以指定两个 realm ：一个是入口 realm，一个是出口 realm。

在接下来的一节中，为了更加熟悉 realm 特性，我们首先来看如何配置 realm。然后描述路由代码根据 realm 配置得出分类标签（该标签会被流量控制子系统使用）所使用的逻辑。

在 IPROUTE2 软件包包含的文件 *ip-cref.ps* 中，给出了一些例子来描述 realm 的意图和使用。在本书中，我们只考虑如何通过 IPROUTE2 命令来配置 realm。

使用 *ip* 命令既可以配置路由，也可以配置策略。*ip* 接下来的第一个关键字确定配置的对象类型。*route* 关键字表示配置路由，而 *rule* 关键字表示配置策略。

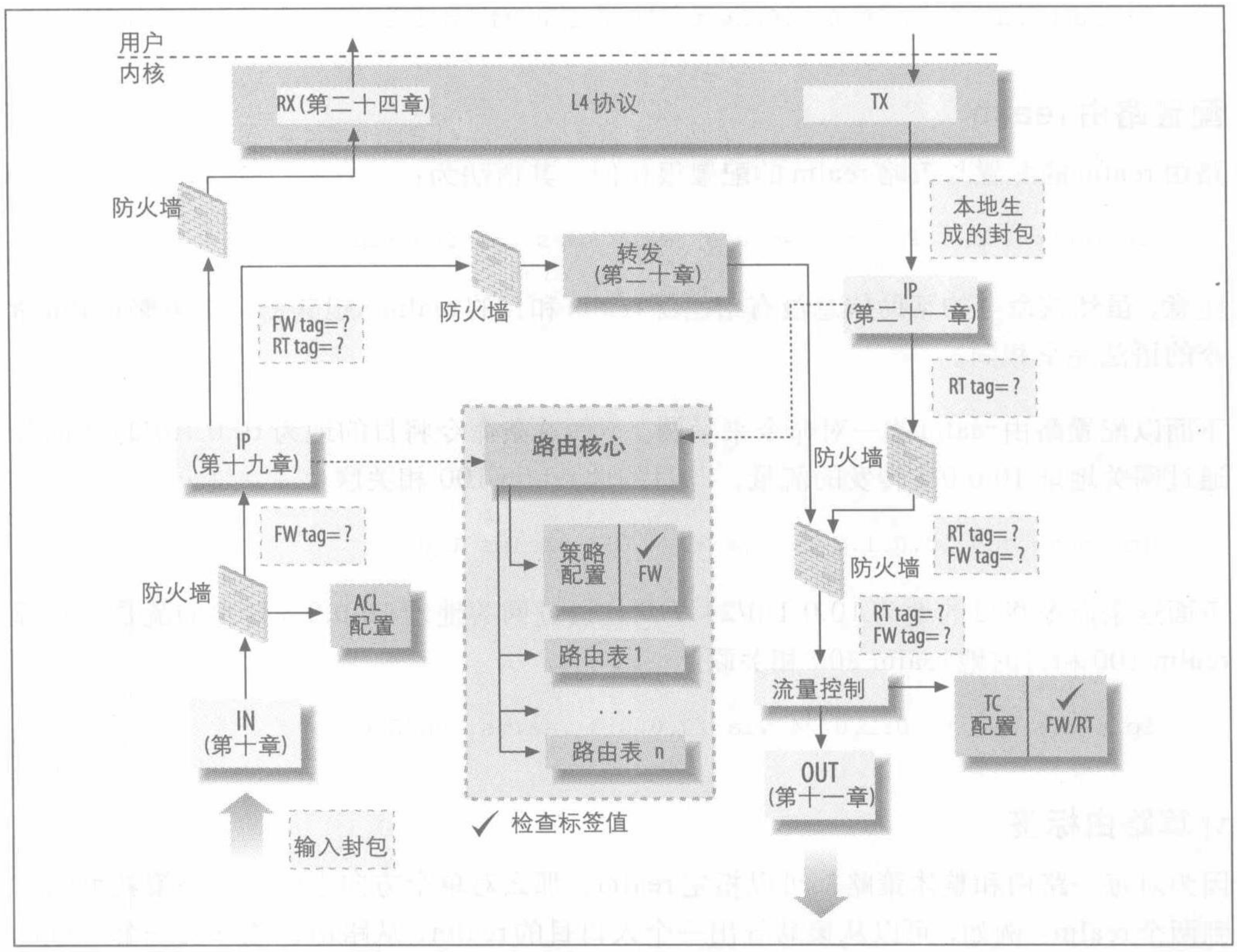


图 31-6：路由、流量控制和防火墙（Netfilter）之间的交互

配置策略 realm

利用 IPROUTE2 软件包中的 *ip rule* 命令来配置路由策略，其语法为：

```
ip rule add...realms [source_realm/]destination_realm
```

从上面的语法可以看出，源 realm 是可选的，而目的 realm 是不可选的。这表示如果只提供了一个值，那么配置的是目的 realm。

下面用配置策略realm的一对命令来举例。下面这条命令将源于 10.0.1.0/24 子网的所有流量，与策略目的地 realm 128 相关联：

```
ip rule add from 10.0.1.0/24 realms 128
```

下面这条命令将源于 0.0.1.0/24 子网且目的为 10.0.2.0/24 子网的所有流量，与策略源 realm 64 和策略目的地 realm 128 相关联：

```
ip rule add from 10.0.1.0/24 to 10.0.2.0/24 realms 64/128
```

配置路由 realm

路由 realm 的配置与策略 realm 的配置很相似，其语法为：

```
ip route add...realms [source_realm/]destination_realm
```

注意，虽然该命令的帮助信息没有给出源 realm 和目的 realm，但其语法与策略 realm 命令的语法完全相同。

下面以配置路由 realm 的一对命令来举例。下面这条命令将目的地为 10.0.1.0/24 子网且通过网关地址 10.0.0.3 转发的流量，与目的地 realm 100 相关联：

```
ip route add 10.0.1.0/24 via 10.0.0.3 realms 100
```

下面这条命令将目的地为 10.0.1.0/24 子网且通过网关地址 10.0.0.3 转发的流量，与源 realm 100 和目的地 realm 200 相关联：

```
ip route add 10.0.1.0/24 via 10.0.0.3 realms 100/200
```

计算路由标签

因为对每一路由和整体策略都可以指定 realm，那么对单个方向上的路由决策就可以得到两个 realm：例如，可以从策略导出一个入口目的 realm，从路由也能导出一个入口目的 realm。这种情况下，从路由得出的 realm 的优先级更高。通常，这种决策只对目的 realm 才有必要，管理员几乎不会根据路由来确定源 realm。

如果路由或策略中都没有提供 realm，内核将计算反向路由（从本地主机回到被分类的封包的源主机的路由），并检查是否可以使用与该反向路由相关的 realm。例如，如果内核从入口流量不能得出源realm，那么就根据反向路径上的出口流量来计算目的realm并使用它。这种推算方法是假定在入口和出口方向上的 realm 配置应当是对称的。

我们使用图 31-7 中的拓扑结构来看一个简单例子，图中的两个网络之间有一台路由器。在该路由器上的策略路由配置表明，从子网 10.0.1.0/24 来的流量都属于 Realm A，从子网 10.0.2.0/24 来的流量都属于 Realm B。假设没有配置路由 realm，只有图 31-7 中给出的两个策略realm配置。这两个策略都只提供了源realm，所以当转发流量时，可以为入口流量指定 realm，而出口流量没有 realm。我们现在假设该路由器收到一个来自主机 10.0.1.100（Realm A）且目的地址为 10.0.2.200（Realm B）的封包。当路由子系统为该封包查找路由时，也要计算路由标签。下面给出具体过程：

1. 路由查找返回的是路由R2和策略P1。因为在路由R2上没有配置realm，所以使用策略P1上的源realm A。
2. 因为没有初始化目的realm，内核计算从10.0.2.200到10.0.1.100的反向路由。这一次路由查找返回的是路由R1和策略P2。在匹配的路由R1上还是没有配置realm，所以内核依赖于策略realm，即realm B。但因为这是在反向查找时发现的，所以反向路径上的源realm B被用做转发路径上的目的realm。

最终，路由标签被初始化为源realm A和目的realm B。当流量稍后通过QoS层时，QoS层可以使用这两个realm来将封包正确地分类。

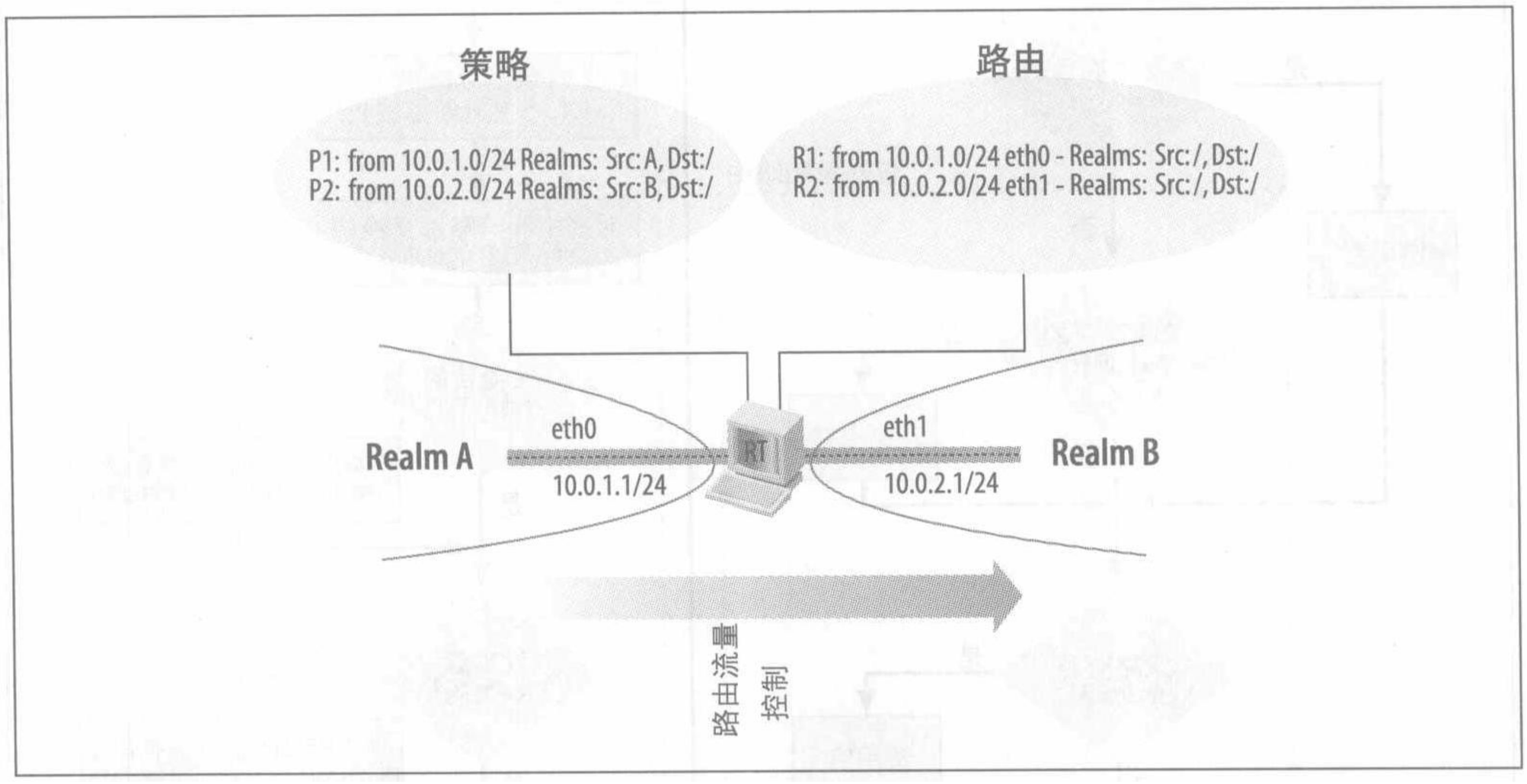


图31-7：realm配置例子

图31-8总结了用于计算路由标签的逻辑。

策略路由与基于防火墙的分类器

Netfilter防火墙软件可以根据自己的过滤标准将流量分类，判断是否需要将封包丢弃或毁坏。防火墙也可以配置为使用其功能强大的分类引擎将封包简单分类，只是为了其他内核子系统提供一个服务。防火墙在网络协议栈中有多个钩子。如果其中一个钩子放置了一个标签后，路由子系统或流量控制子系统正在运行，这些子系统可以看到并使用该标签。在本章前面的图31-6中，给出了各个子系统及防火墙钩子访问封包的顺序。

路由协议守护进程

可以根据下面三种方式将路由插入到内核的路由表中：

- 通过用户命令静态配置（如 *ip route*、*route*）。
- 通过路由协议，例如边界网关协议（BGP）、外部网关协议（RGP）、开放式最短路径优先（OSPF）协议等动态配置，这些协议的实现是作为用户空间的路由守护进程。

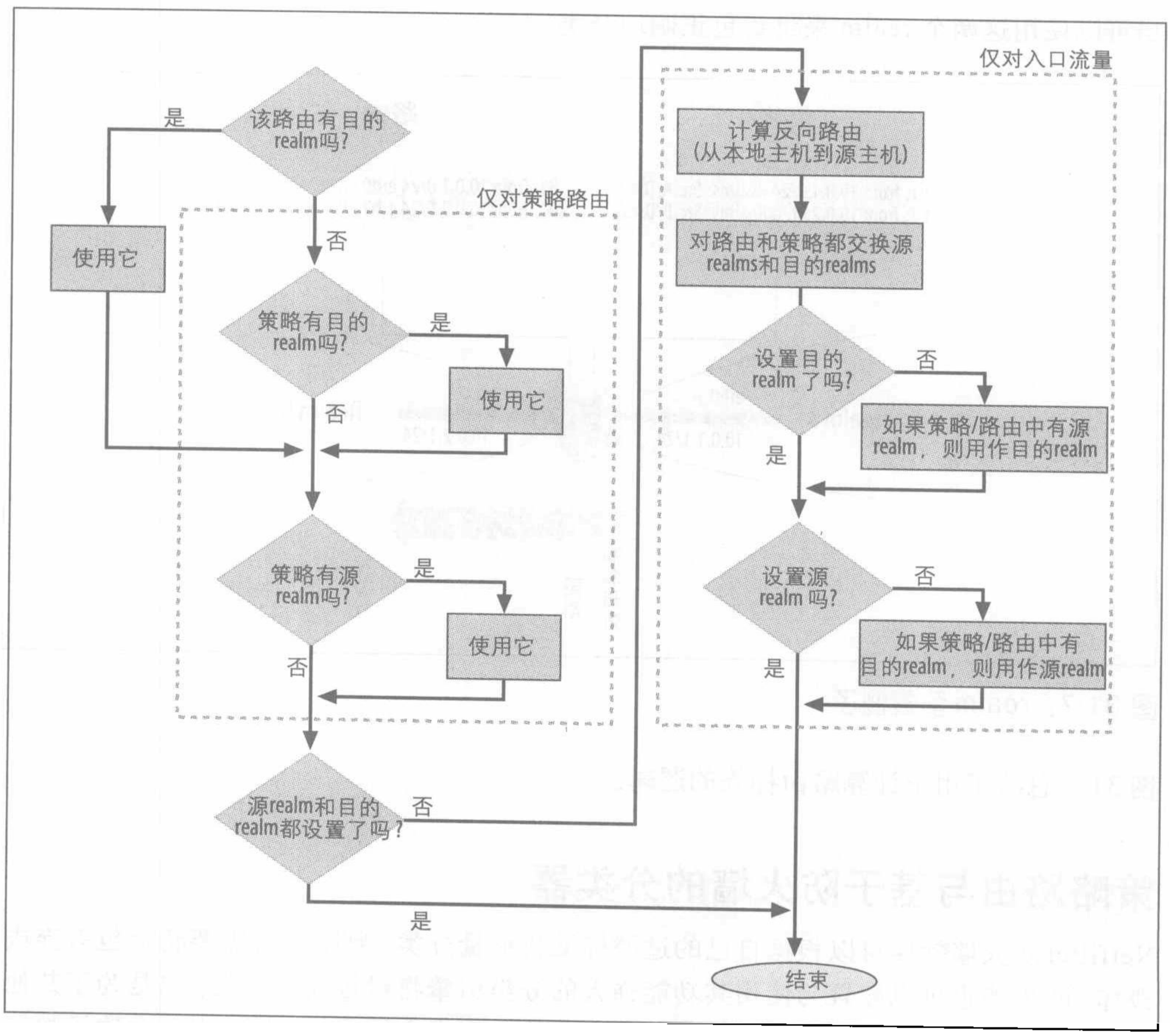


图 31-8：计算路由标签所用的逻辑

- 由于配置不理想而由内核接收和处理的 ICMP 重定向消息。

我们将在第三十六章讨论第一种方式，在本章后面讨论第三种方式，现在我们先来看第二种方式，尤其是可用于Linux 系统的路由守护进程。这些路由守护进程与内核的交互

细节将在第三十二章中讨论。下面列出的一些路由守护程序项目已不再被维护但仍然会令人感兴趣：

Routed

这是最老的 Unix 路由协议守护进程。它只包含 RIP 协议的版本 1 和版本 2（参见 RFC 2453）。

GateD (http://www.gated.org)

GateD 包含大部分路由协议。这最初是由 Merit GateD 联盟发起的一个研究项目，但其版权后来属于 NextHop 。该研究版本已不再被维护。

BIRD (http://bird.network.cz)

BIRD 是由布拉格 Charles 大学发起的一个项目，支持大部分常用的路由协议。

下面列出的路由协议套件仍然被维护并且被运用：

Zebra (http://www.zebra.org)

Zebra 包含大部分路由协议，已经被广泛应用且其邮件列表很活跃。但其版本更新周期已经变慢，这导致了 Quagga 的诞生 。

Quagga (http://www.quagga.net)

创建于 2003 年的 Quagga 是 Zebra 的一个分支，它提供用户社区，这样就使其开发周期和 BUG 修复更快，并且有更多的文档。

XORP (http://www.xorp.org)

由加利福尼亚伯克利大学国际计算科学学院（International Computer Science Institute）发起的一个新项目。

参考括号内的 URL 地址来查找每个软件包提供的具体协议及扩展。

本书不涉及路由守护进程的实现，因为它们不属于内核，但我们简单看看它们如何与内核通信。重要的是要知道这些问题，比如路由守护进程是如何将它们从其同僚或用户配置学习到的路由插入到路由表中，以及如何删除已失效的路由。

每一个守护进程是在用户空间维护自己的路由表。选择路由并不是直接使用这些路由表，而是用内核中存储的路由表。但就像在本节前面提到的那样，这些守护进程是内核路由表中添加路由项的一种方法。前面介绍的大部分守护进程都实现了多种路由协议。每一种路由协议在运行时维护各自的路由表。依据守护进程的设计，每个协议可以将自己路由表中的路由项插入到内核路由表中（见图 31-9 中的左边的部分），或是几个协议共享守护进程内的一个公共层来与内核通信（见图 31-9 中的右边部分）。用户空间内的路由协议选用哪种方法来设计不在本书的讨论范围。

路由协议和内核之间的通信是双向的：

- 路由协议将路由插入到内核的路由表中，并判定过期的或不再有效的路由项，将其删除。
- 内核将插入路由项、删除路由项，以及本地设备链路状态改变（这当然间接影响到相关的所有路由）通知路由协议。这只有当路由守护进程通过Netlink套接字与内核通信时才有可能，因为Netlink套接字是一个双向通道。

IPROUTE2软件包不仅允许用户配置路由，而且可以监听前面提到的由内核和路由守护进程生成的通知。这样，管理员可以将它们通过日志记录下来，或在屏幕上输出以用于调试。

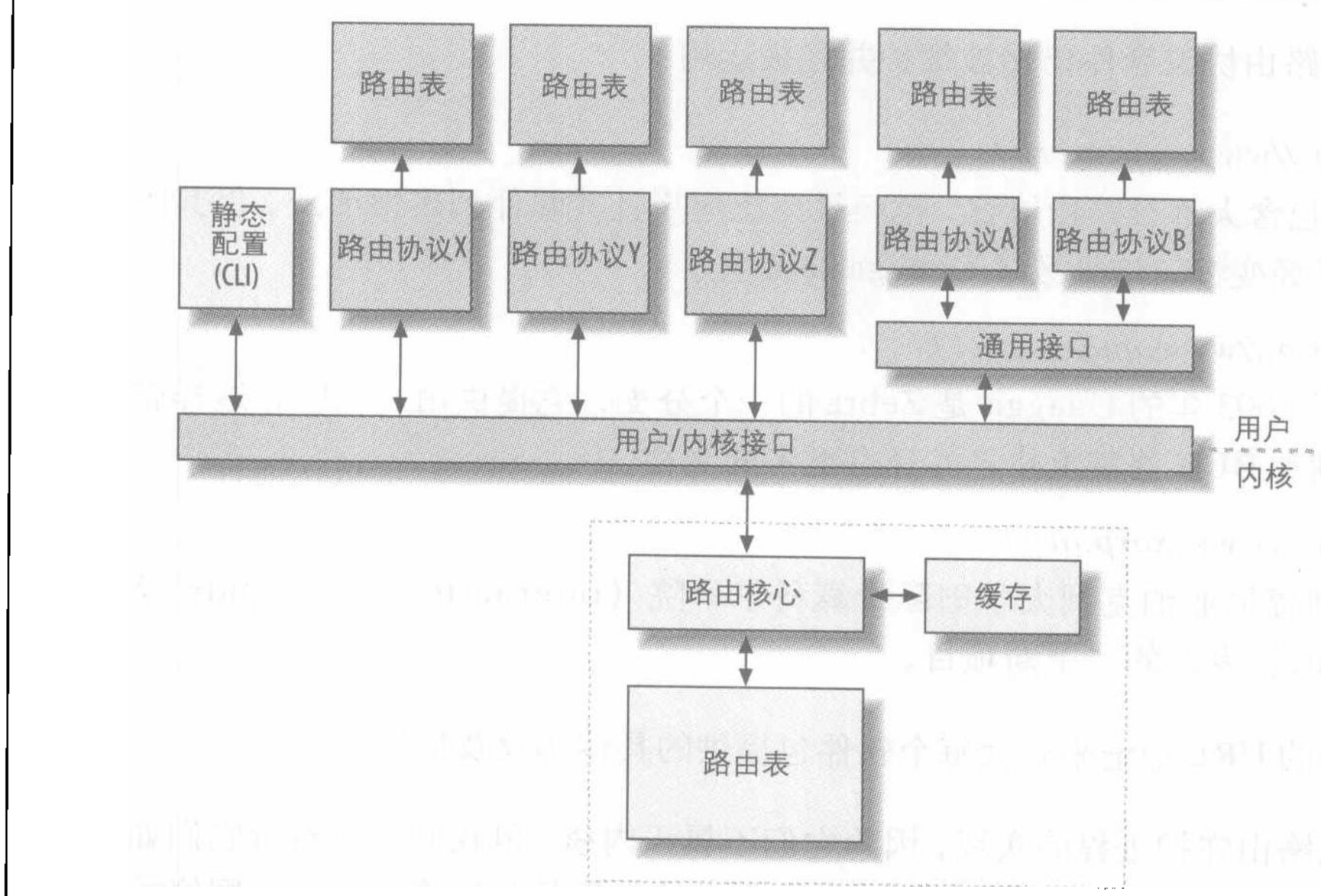


图 31-9：用户空间与内核间的接口

Verbose 监控

当内核支持verbose监控选项且激活该选项（默认为关闭）时，在输入封包的源IP地址或目的IP地址可疑或无效时，内核将在控制台上输出警告信息。这些信息的输出速度被限制为每5秒钟输出一条，以避免潜在的DoS攻击。

当入口封包由于源地址或目的地址错误而被路由子系统中的合理性检查丢弃时，将触发一个警告信息。内核可以使用第三十章中表 30-1 和表 30-2 中列出的地址分类很容易做这些的检查。总之，这些分类是：

- 源地址：多播地址、回环地址、保留地址、无效地址（zeronet）。
- 目的地址：回环地址、保留地址、无效地址（zeronet）。

内核还根据路由表对入口封包做额外的合理性检查，尤其是：

- 当启动反向路径过滤（一种反 IP 欺骗检查）时，源 IP 地址必须通过接收该封包的接口可到达，参见“反向路径过滤”一节。
- 源 IP 地址不能是子网广播地址或接收接口上配置的地址。这检查可以防止 IP 欺骗的企图（例如，另一台主机声称拥有接收接口上配置的 IP 地址），并且可以检测出可能由于 DHCP 错误配置而引发的地址重复情况。

当启动 Verbose 监控特性后，ICMP 层在一些特殊条件下也能生成警告信息：

发送 ICMP REDIRECT 消息

当内核向一个远端主机发送一定数量的 ICMP REDIRECT（重定向）消息，而该主机忽略时，则内核输出一个警告。发送的数量可以配置，参见“发送 ICMP_REDIRECT 消息”一节。

接收 ICMP REDIRECT 消息

一旦入口 ICMP 重定向消息被拒绝，内核就输出一个警告。入口 ICMP REDIRECT 消息的处理要比发送较为复杂，因为内核可能由于多种原因而拒绝入口 ICMP REDIRECT消息，其中一些原因可以由用户配置。参见“处理入口 ICMP_REDIRECT 消息”一节。

ICMP_REDIRECT 消息

ICMP 协议定义了许多不同消息来控制流量及把网络问题通知给主机。其中的一个消息为 REDIRECT［重定向消息（注 2）］，它被用于向封包源地址主机通知使用的路由不是最理想路由。参见第二十五章中对 ICMP 消息的详细描述。本章中，我们将焦点集中在由路由子系统提供的调整参数上，路由子系统使用这些参数来判断是否处理入口 ICMP REDIRECT 消息，以及当所需要的默认条件满足时是否发送一个 ICMP 重定向消息。

注 2：ICMP REDIRECT消息有四类。本章只讨论将流量重定向到一个特定IP地址的REDIRECT HOST 消息。其他类别可参见第二十五章。

是否应当发送一个ICMP REDIRECT 消息，以及是否应当处理一个入口ICMP REDIRECT消息的决定，都会受到用户配置的影响。尤其是，用户可以认为：

- 是否接收入口ICMP REDIRECT 消息或生成出口ICMP REDIRECT 消息。我们将在第三十六章看到，每一个设备上都能进行类似的配置。
- 对于入口ICMP封包，也可能指定主机是否只接收安全的重定向消息。当一个ICMP REDIRECT 消息声明的新网关已经被本地主机知道且作为一个网关时，该消息就被认为安全的。这可以通过查找路由表判断，比如，检查建议的网关是否已作为任意已配置路由项的下一跳。
- 每个设备可以配置有一个标识，表示该设备是否与一个共享介质相关联。

在下一节中会提供关于最后一点的更多描述。

共享介质

在19世纪90年代初期，IP协议设计者开始看到了在诸如以太网等介质上建设LAN，以及将这些LAN与其他网络相连的趋势（这在当时是新的趋势，而现在已经很普遍了）。管理员有时候将配置在不同IP子网的多组主机连到一个LAN内，同时使用路由器将它们隔离。这样做与历史上的多种原因有关或是出于方便，而现在几乎没有任何理由要这样做，而且也几乎没有人这样做。然而，路由子系统在设计时必须处理它。

当配置为不同子网的主机被连到同一个LAN时，IP路由文档中称之为共享介质（shared media）。这个术语是指网络配置而不是设备能力。换句话说，在常见的共享一个L2连接的所有主机也共享一个IP子网情况下，该术语不适用。这里也不关心这些话题；在本章中，我们所关心的是更有可能生成ICMP REDIRECT 的情况。

图31-10给出了一个共享介质的例子。在连到同一个LAN的主机被配置为三个不同的子网。两个路由器RT1和RT2被用于连接IP子网：每一个路由器将是两个子网的一部分，在每一侧每个NIC配有一个地址。

接下来，描述图31-10中配置路由的典型方式（虽然这不是唯一的配置方法）。

- 子网10.0.0.0/24内的主机将RT1定为自己的默认网关，通过该网关将流量送到其他子网。通过RT1可以到达其他两个子网，以及访问因特网的网关。
- 子网10.0.1.0/24内的主机有类似的配置，将RT2定为自己的默认网关。但它们需要另外一条通过RT1到达子网10.0.0.0/24的路由。

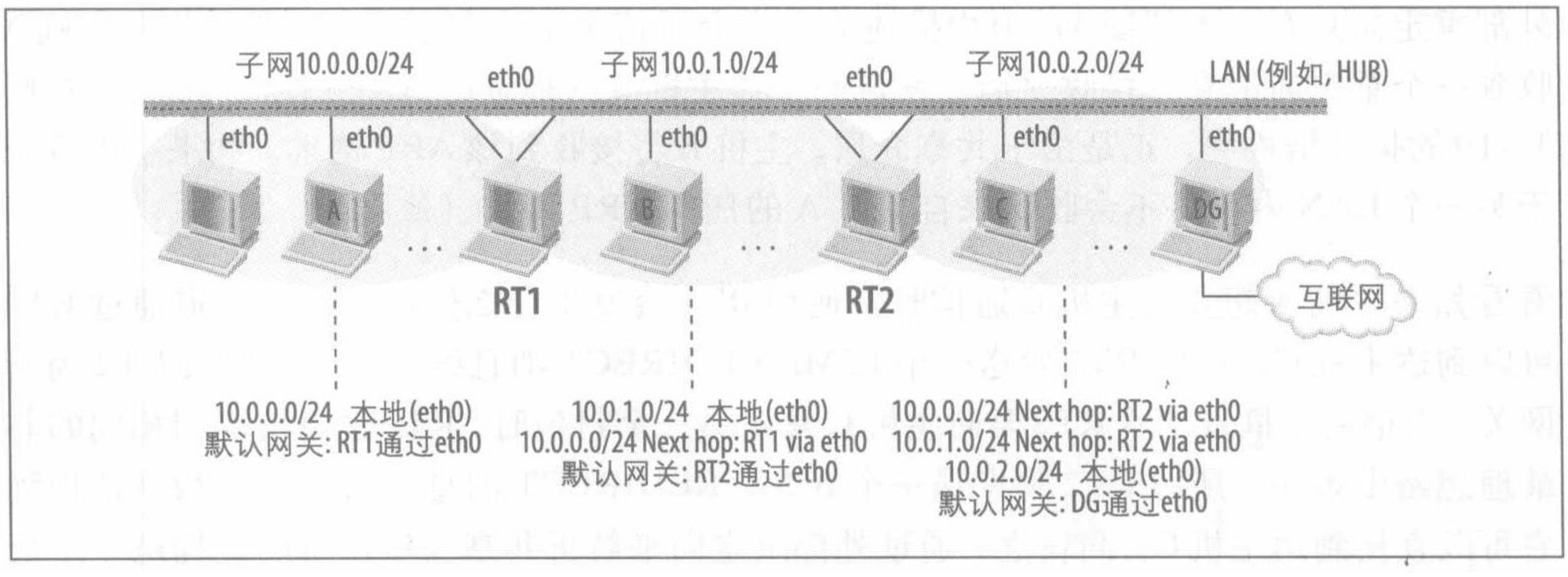


图 31-10：共享介质拓扑结构上配置例子

- 子网10.0.2.0/24的主机将DG定为自己的默认网关，并配置有两条路由，用于通过RT2到达其他两个子网。

这个路由场景的关键之处在于指定的路由效率很差。例如，子网10.0.1.0/24和10.0.0.2/24内的主机可以在L2层交换封包而不需要任何路由，但配置信息要求它们使用两个路由器。一个更为聪明的配置是只在主机上配置默认网关，由默认网关来处理到LAN中到其他子网的路由。

幸运的是，路由子系统有了克服自己这种低效率的办法，可以逐步找到直连主机。这种机制就是ICMP REDIRECT消息。我们将忽略DG及因特网连接的存在。

假定主机A想要与主机B通信。主机A根据自己的路由表，看到通过路由器RT1可以到达主机B。然而，当RT1收到主机A发给主机B的封包时，它会意识到主机A可以直接发送封包给主机B，因为它们（主机A和主机B）之间通过同一个设备*eth0*就可到达。这看上去就像触发生成ICMP REDIRECT消息的经典条件：路由不理想。但是有一点要注意：虽然主机A和主机B都连到同一个共享介质，并且因此能够从链路层角度（即Ethernet）相互通信，但是从IP层来看是不可能的。主机A不知道通过*eth0*可以到达10.0.1.0/24子网的主机，主机A所知道的就是通过RT1可以到达子网10.0.1.0/24。

为了理解其中的原因，如果必要的话可返回参看第二十六章中的“何时发送和处理Solicitation请求”一节。一台主机要根据L3地址与另一台主机通信，首先必须进行L3地址到L2地址的解析。一个发送方只能对与它相连的属于同一子网的主机做这种解析；对于主机，找到它们则是路由器的事情。在我们这个例子中，主机B不属于主机A的子网。这意味着如果一个ICMP REDIRECT消息告诉主机A可以与主机B直接通信，这些主机必须能够接收所谓的外部重定向（foreign redirect）：一个重定向建议新的下一跳不属于该消息接收者已知的本地子网。

外部重定向只有在类似图31-10中描述的共享介质情况下才有意义。这是因为在主机A收到一个重定向消息并接收之后，发送一个为主机B的地址的ARP请求（注3）。在图31-10的拓扑结构中，正是由于共享介质，主机B将接收到该ARP请求。如果主机B位于另一个LAN内，将不会收到来自主机A的任何ARP请求（注4）。

看看如果主机A想要与主机C通信时，使用RT1会发生什么情况。RT1知道通过RT2可以到达主机C，所以RT1发送一个ICMP REDIRECT消息给主机A，建议RT2为新网关。当稍后主机A请求RT2要向主机C发送另一个封包时，RT2也将检测到相同的非最理想路由条件。所以RT2将发送一个ICMP REDIRECT消息，主机A将最终认识到它可以直接到达主机C。简言之，通过外部重定向来解析共享介质连接的过程是一个反复过程。

在这一节中，你已经看到对连到同一共享介质属于不同子网的主机进行配置使所隐含的细节。RFC 1620给出了对此问题更为详尽的探讨，值得阅读。在接下来的两节中，你将看到用户对连到一个共享介质的接口进行明确配置，以便影响到入口ICMP REDIRECT消息的发送和处理，这样与本一节中类似的环境都可以正确地处理。

发送ICMP_REDIRECT消息

当路由子系统知道入口设备和出口设备已经匹配时，仍然检查共享介质配置似乎是多余的。实际上，这种检查并不多余。在前一节描述的，使用ICMP REDIRECT消息而造成的转发捷径的使用，可能并非人们所期望的。这些捷径使主机绕过某些路由器，但系统管理员可能在这些路由器上要对所有流量应用策略，一个例子就是在RT1和RT2为防火墙（注5）或代理主机时的情况。

图31-11给出了路由程序在路由一个需要转发的入口封包时所遵从的逻辑。该流程图中只有一个部分还没有描述：当设备不是配置为共享介质时是如何检查的。在这种情况下，

注3：我们前面提到一个IP主机永远不会对不属于本地配置子网的IP地址发送ARP请求。注意这里描述的行为（即一个主机向位于外部子网的IP地址发送ARP请求）是可能的，因为在路由缓存中包含有重定向路由，所以可以绕过路由表（该表不可能向外网IP发送ARP请求）。

注4：有意思的是注意IPv6有优化的ICMP重定向。在IPv6情况下，重定向消息包含了建议的新网关的L2地址。所以一个收到ICMP REDIRECT的主机不需要再做从L3地址到L2地址的解析，就知道新网关的地址。

注5：在图31-10中，所有主机共享一个L2广播域，在类似这样的环境中使用防火墙无论如何来讲都是一个很差的选择。但这可以使你理解为什么人们并不总是期望有路由捷径。

发送方（通过源IP地址来标识发送方）只有在和通过查找路由表到的下一跳属于同一子网时，才发送ICMP REDIRECT消息。当上述条件不满足时，发送方将不能够（依据路由器的知识）到达新的下一跳。这就是为什么在检查网关是否属于相同子网之前先要检查共享介质。

注意在内核2.6中已经不再支持快速NAT（参见第三十二章“最近去掉的选项”一节），所以对NAT/MASQ将检查将永远不成功。

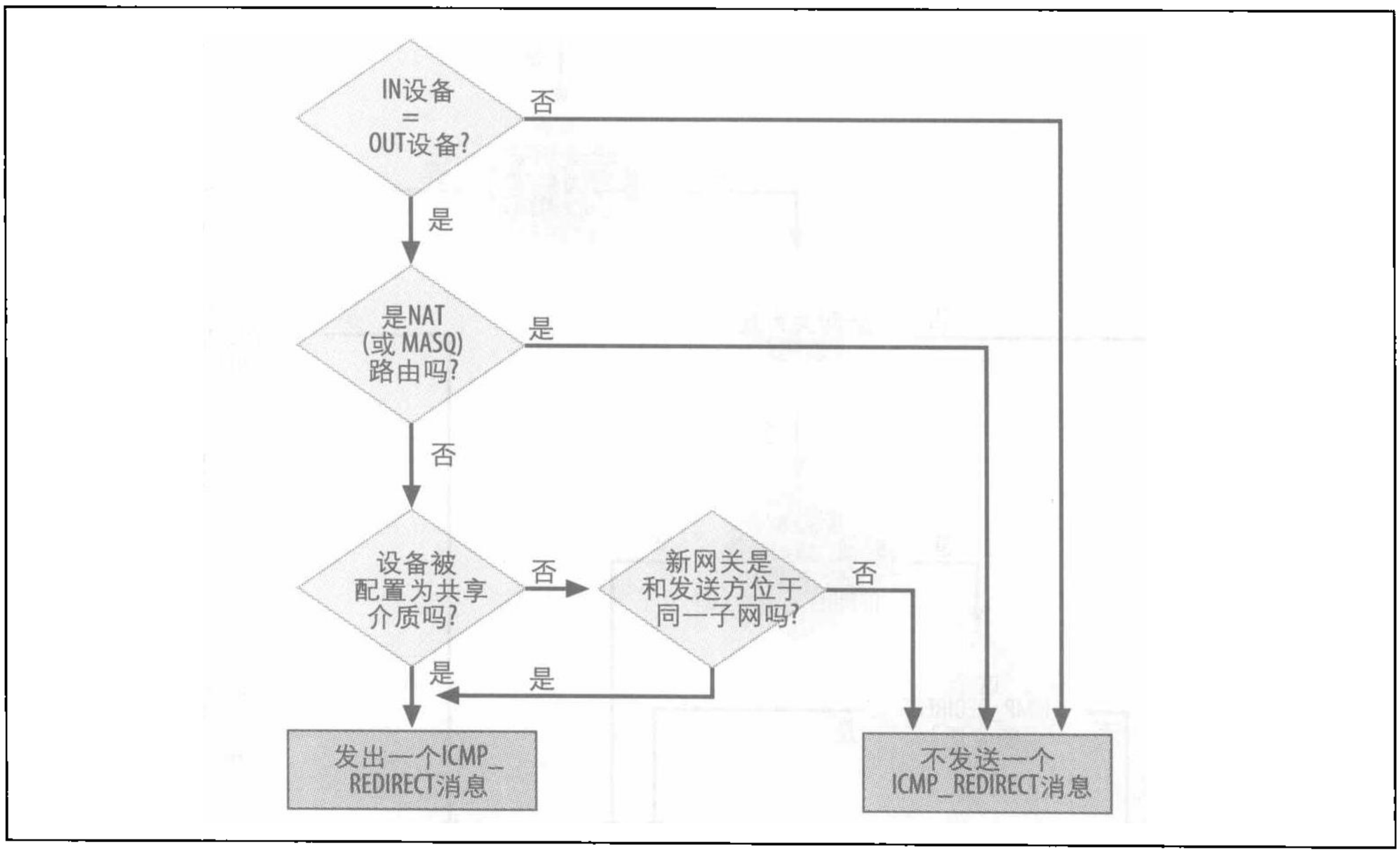


图 31-11：生成一个 ICMP REDIRECT 需要的条件

参见第二十章和第三十五章“转发”一节中关于ip_forward函数如何确定是否发送ICMP REDIRECT消息的细节。发送ICMP消息的更多细节也可以参见第二十五章。

处理入口 ICMP_REDIRECT 消息

对于一个要被接受的入口ICMP REDIRECT消息，它首先要经过一些合理性检查并且要遵守用户的配置。图31-12用一个流程图给出了所有相关的逻辑。下面我们按照图中的顺序仔细分析。

首先，需要通过一些基本的合理性检查并且遵守用户的配置：

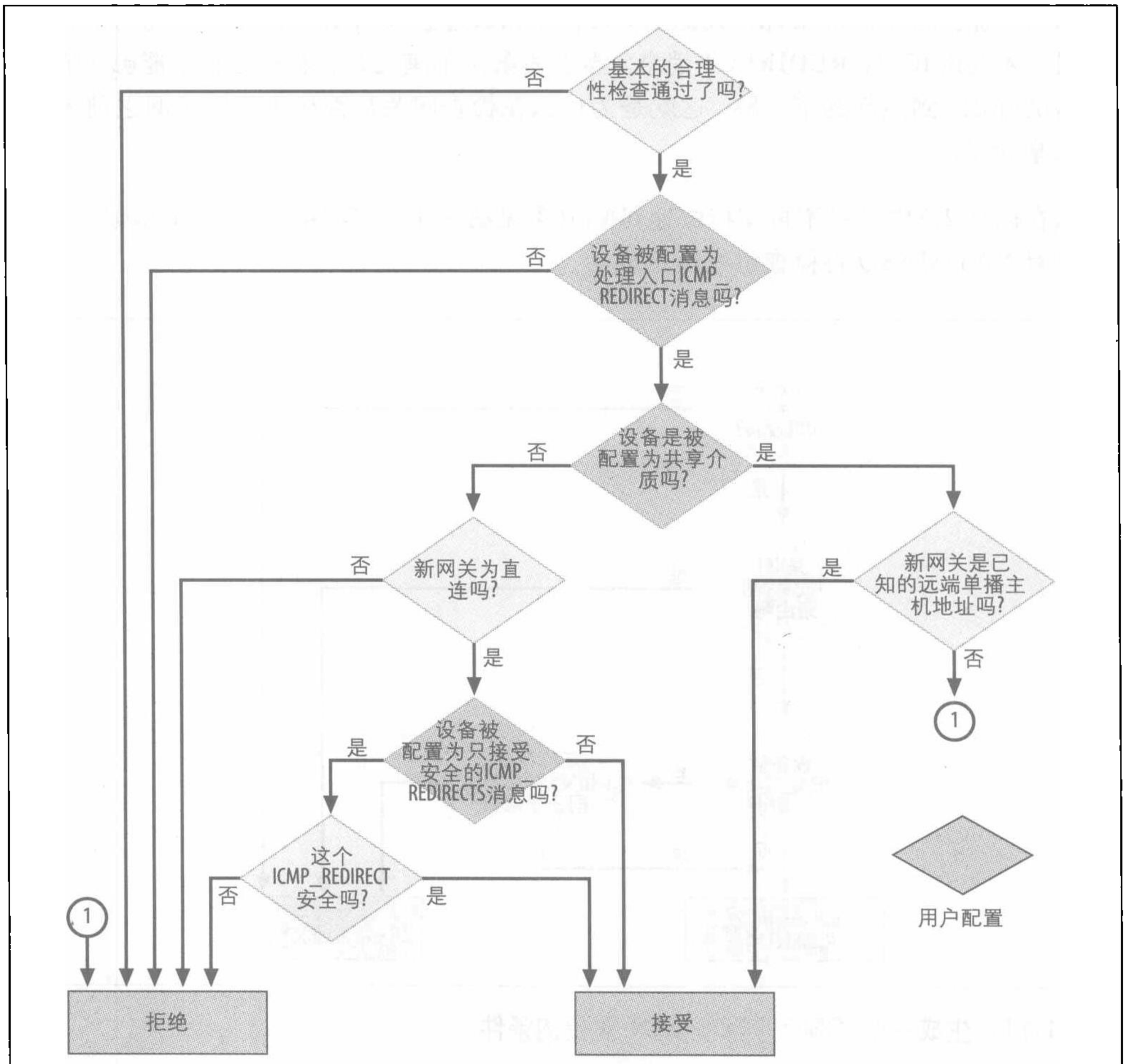


图 31-12：处理入口 ICMP REDIRECT 需要的条件

- ICMP REDIRECT 消息中宣告的新网关应当和当前网关不同（否则不需要 ICMP REDIRECT）。
- 新网关的 IP 地址不能是多播地址、无效地址（零网地址）或保留地址。
- 接收接口必须被配置为可以接受入口 ICMP REDIRECT 消息。

其次，路由子系统必须考虑共享介质配置情况：

如果设备没有被配置为共享介质

这时，只有当主机根据路由表知道新网关与旧网关位于同一子网时（即它与该主机直接相连），才能够接受这个新网关。

如果设备被配置为共享介质

只有当主机根据路由表知道如何到达该新网关时，才能够接受这个新网关。同时执行另外两项合理性检查：该网关地址一定不能是主机的本地地址，且不是广播地址。

就像在“共享介质”一节中已经提到的，可以配置一个设备，以便只有当它已经知道这个新网关是一个网关时，才可以接受重定向。

`ip_rt_redirect`函数根据本节描述的逻辑来处理入口ICMP REDIRECT消息，详细分析参见第二十五章。

反向路径过滤

我们在第三十章“路由的基本要素”一节中曾看到什么是非对称路由。非对称路由并不常见，但在某些特定情况下可能是必需的。Linux的默认行为认为非对称路由是可疑的，因而将根据路由表丢弃那些源IP地址通过接收接口为不可到达的封包。然而，可以通过*/proc*来调整每一个设备的这种行为，我们将在第三十六章讨论这一点。也可以参见第三十五章“输入路由”一节。

在第三十章的图30-7（a）和30-7（b）中，我们看到一个恶意用户利用其同一子网内另一台主机地址作为源IP地址来发送ICMP ECHO REQUEST消息的例子。图31-13给出了另一个恶意用户，这次他使用的是目的子网内的一个地址（也就是他的受害者）作为源IP地址来发送ICMP ECHO REQUEST消息。如图31-13（a）所示，Linux默认可以检测出这种企图，并将这种封包丢弃。图31-13（b）给出了如果路由器没有丢弃RTICMP ECHO REQUEST消息时所发生的情况。

上述例子使用的是定向广播ICMP封包，但反向路径过滤可以适用于任意类型的流量。

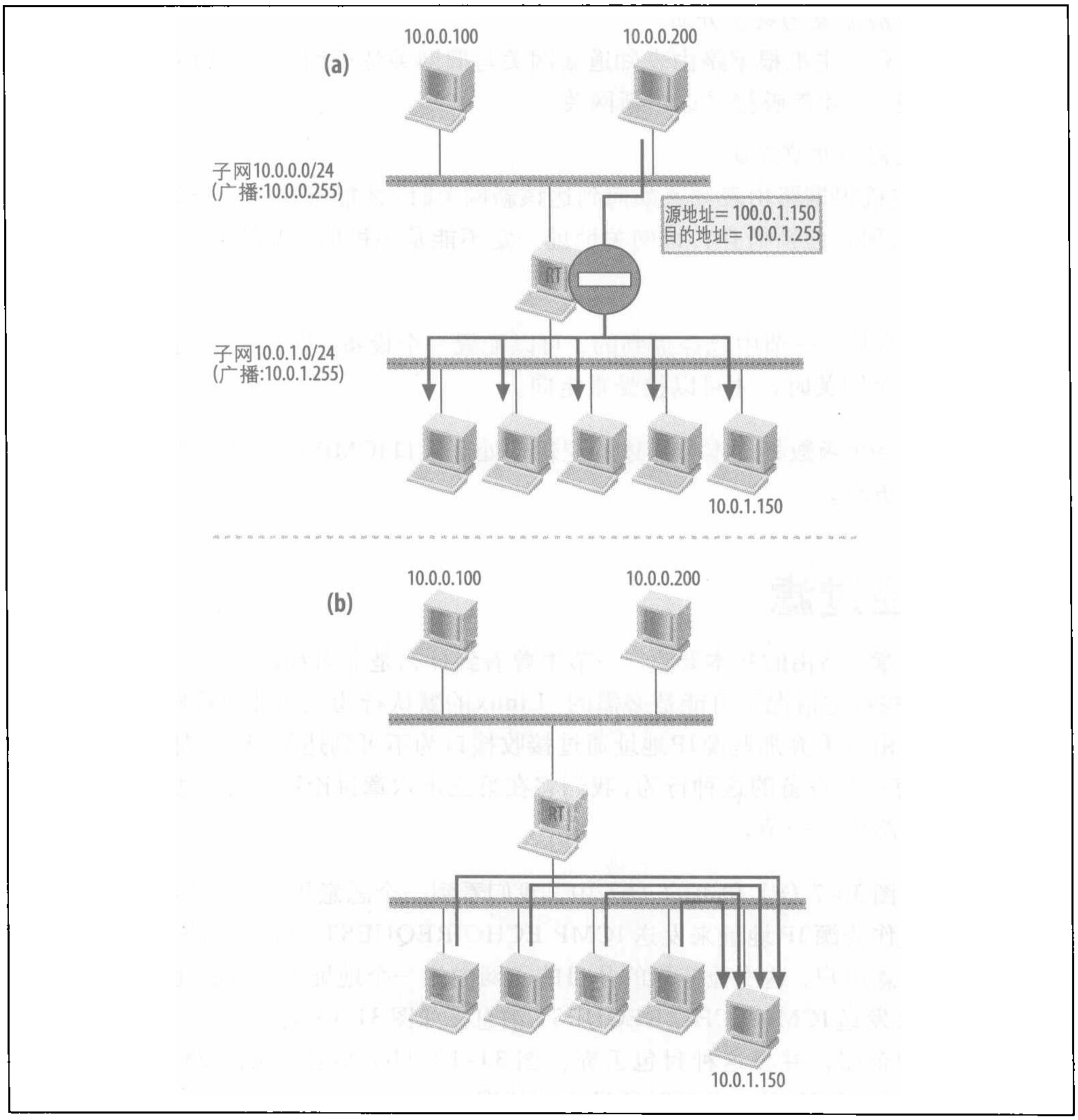


图 31-13：反向路径过滤例子

第三十二章

路由：Linux的实现

第三十章简要介绍了路由子系统的主要任务，第三十一章介绍了IP在基本路由功能上实现的一些特性，比如策略路由和多路径路由等。在本章，将首先介绍路由代码用到的主要数据结构，然后会描述：

- 如何定义路由 scope 和 IP 地址 scope。
- 如何初始化路由子系统。
- 哪些外部事件需要通知路由子系统，以便更新路由信息。

后面的章节我将详细描述路由缓存、路由表和路由查找。

内核选项

就像我们将在本章后面看到的那样，路由行为并不仅仅涉及从一个接口接收到一个封包后，查找路由表，将该封包从正确的出接口转发出去，同时还需要考虑其他一些任务。在Linux内核中已经实现了与路由相关的许多有意义的特性。而且我们在本章后面还将看到，还有许多特性正在等待 Linux 或其他子系统开发者的放行，以便集成到内核中。

下面，我简要介绍一些影响路由代码行为的Linux内核特性，让读者在仔细阅读源代码时不会混淆。在本章和接下来的章节中将会进一步描述每一个特性。

路由选项可以划分为两类：

- 一类是永远支持的，只需要用户通过比如 */proc* 来配置或开启的选项。
- 另一类是重新编译内核时可以添加或删除的选项。可以将CONFIG_WAN_ ROUTER选项和CONFIG_IP_ROUTE_MULTIPATH_CACHED菜单下的选项编译为模块，而其他的选项必须被编译进主内核。

我们将在本章和下面的章节中看到这两类选项，但本节内我们首先来看编译时选项。这些选项可以利用“Networking Options”菜单来配置，如图 32-1 所示为在执行 make *xconfig* 时的一个截图。

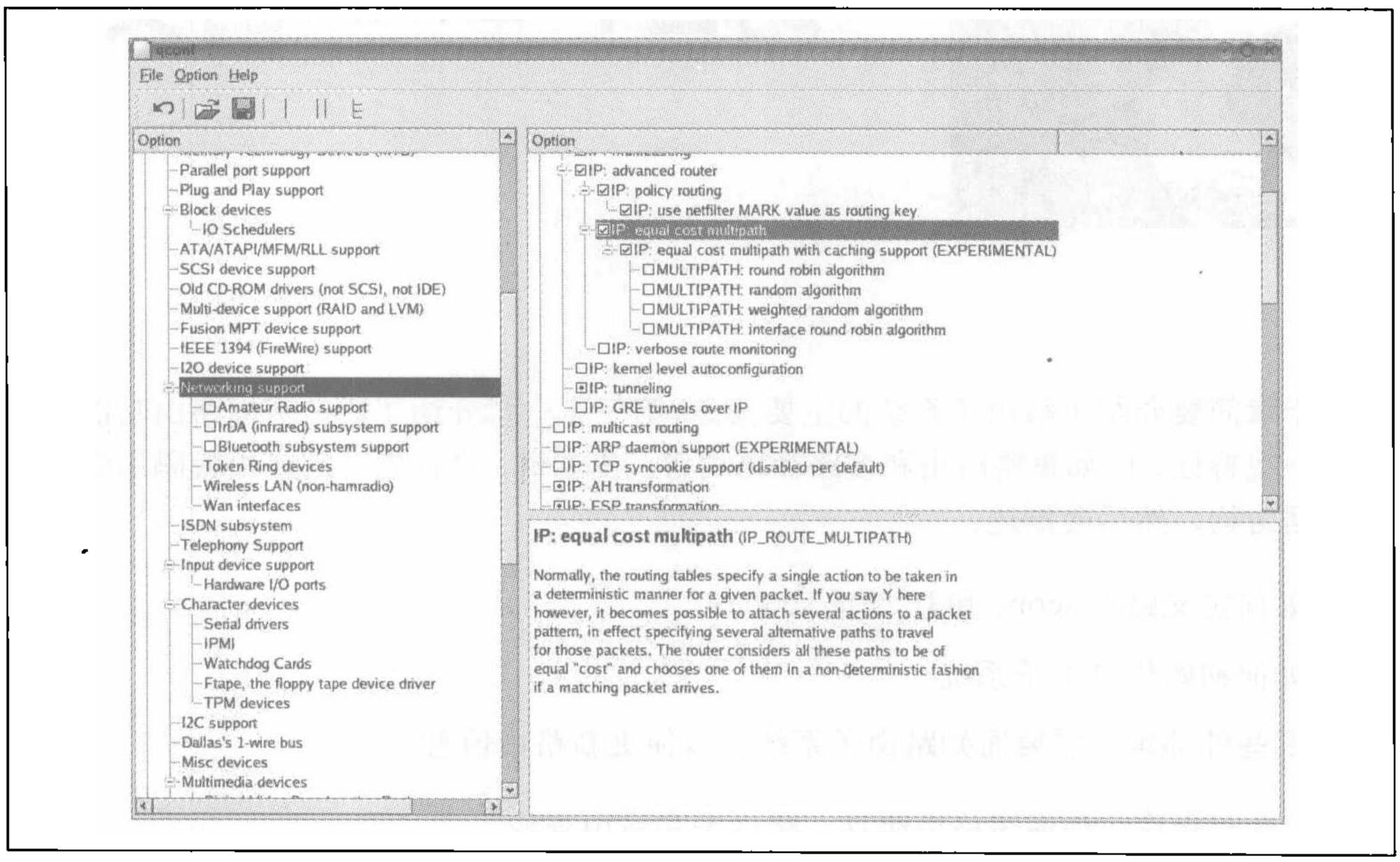


图 32-1：配置内核（make xconfig）

下面两节在括号内给出与每个选项相关的 CONFIG_*XXX* 符号表示的内核配置，可以利用这些符号来识别那些只有在内核支持该特性条件时才执行的代码。但是当某个特性使用一个文件时，在该文件内找不到 CONFIG_*XXX* 符号。

基本选项

下面给出一些基本路由选项，这些选项在本章都不涉及。

IP：多播路由（CONFIG_IP_MROUTE）

IP：支持 PIM-SM 版本 1（CONFIG_IP_PIMSM_V1）

IP：支持 PIM-SM 版本 2（CONFIG_IP_PIMSM_V2）

如果你想使用 IP 多播路由，可以选择激活内核支持的 PIM（Protocol Independent Multicast）协议的两个版本中的一个。本书不涉及多播路由。

WAN 路由器（CONFIG_WAN_ROUTER）
该选项可以在广域网设备上配置 X.25 、帧中继、HDLC 及其他非 IP 协议来进行路由。在内核配置菜单中，可以在“Network device support → Wan interfaces”下方看到可用的驱动列表。为了配置广域网设备，需要下载部分代码，这些代码在常见的 Linux 版本发布中默认是不包含的（注 1）。本书不涉及广域网路由。

高级选项

如果在网络选项菜单中激活“IP: Advanced router”选项，则可以激活更多的选项。在第三十一章中已经介绍了每一个选项，下面再简要概述一下：

IP：策略路由（CONFIG_IP_MULTIPLE_TABLES）
在一些情况下，流量处理除了依据目的IP地址以外，还必须依据其他一些条件。策略路由是回答该限定条件的一种答案。在这些情况下，必须增强路由代码以考虑更多的参数。参见第三十一章“策略路由背后的概念”一节。

IP：使用 netfilter MARK 值作为路由查找关键字（CONFIG_IP_ROUTE_FWMARK）
当激活该选项时，路由表查找时可以考虑由防火墙设置的一个标签。参见第三十一章“策略路由与基于防火墙的分类器”一节。只有先激活“IP：策略路由”选项才可以看到该选项。

IP：等价多路径（CONFIG_IP_ROUTE_MULTIPATH）
有时候，一条路由可以有多个下一跳，这时在所有路由上都发送流量将消耗大量带宽，这正是该特性需要处理的。在第三十一章“多路径路由背后的概念”一节中，我们看到实现该特性并不像它字面意思那么简单。

IP：支持缓存的等价多路径（CONFIG_IP_ROUTE_MULTIPATH_CACHED）
支持将多路径路由添加到路由缓存内，只有在前一选项激活时才可以选择该选项。当选择这个选项时，将看到一个子菜单，列出各种从缓存路由内选择下一跳的算法。参见第三十一章“支持多路径缓存”。

IP：详细的路由监控（CONFIG_IP_ROUTE_VERBOSE）
代码中会有一些地方检测到一些稀奇古怪的条件，例如，在流量处理过程中进行合理性检查时。在这些情况下输出一些告警信息是非常有用的，这正是该特性的目的。为了避免 DoS 攻击，将这些告警信息的输出限速为每 5 秒钟输出一条。

注 1：参见文件 *documentation/networking/wan-router.txt*。

最近去掉的选项

下面给出在2.4系列中内核支持，而在2.6系列中被去掉的一些选项：

IP：快速网络转换（CONFIG_IP_ROUTE_NAT）

NAT是一个在路由器上配置的典型特性，用于根据特定的配置修改转发的IP封包的源或目的IP地址。由路由代码实现的NAT与由防火墙代码实现的NAT无关，因而该选项被认为是多余，它在2.6.9版本的内核中被完全删除。

快速交换（CONFIG_NET_FASTROUTE）

该特性允许在设备驱动层直接转发与NIC之间的数据流量。从出接口转发的封包不必经过更高层（IP层），因而不需要查找路由表。在第三十章的图30-1中驱动框中的点线就表示该特性。目前只有一种NIC系列，即Tulip卡（注2）支持该特性。

该特性在2.6.8版本的内核中被删除，它与其他重要特性不兼容的简单原因在于，它属于低层交换而将绕过这些重要特性。这些重要特性的例子有Netfilter防火墙、高级路由、虚拟设备：即bonding。

高速接口间转发（CONFIG_NET_HW_FLOWCONTROL）

该特性可以使网卡根据其内存中可用的缓冲空间大小，决定开始或停止发送由内核向它送来的封包。但并不是所有的网卡都支持该特性，支持该特性的网卡例子为Tulip系列网卡（*drivers/net/tulip/**）。由于第十章中描述的NAPI的引入，这个有意思的但几乎不被使用的特性已经从2.6.10版本的内核中被删除。

主要的数据结构

路由代码使用了大量的彼此互相引用的各种数据结构。为了理解当前的路由代码及将来的改进，清楚地了解这些数据结构之间的关系是很重要的。

任何代码性能都会由于数据结构和总体代码设计而受到明显影响，这对于内核代码来说尤其如此。因此内核子系统，比如，处于网络协议栈核心的路由子系统，要确保它不仅提供功能的健壮性，而且要考虑性能。我们将在下面的章节看到本节中列出的这些数据结构是如何相互协作，以从CPU和RAM的消耗及缓存的角度来更容易实现算法的优化。

下面列出并介绍了路由代码中定义和使用的主要数据结构。在第三十六章中对那些最重

注2：你可以下载一个补丁应用于内核，使Tulip 8390卡支持2.4内核中的“快速交换”特性。如果要使用该特性（例如，用make xconfig），可以点击帮助窗口内提供的到该补丁的链接。

要的数据结构则会详细地按字段描述。在数据结构名称中的rt、fib与fn前缀分别表示路由（route），转发信息库（Forwarding Information Base）和功能（function）。

struct ip_rt_acct

该结构被基于路由表的分类器使用（参见第三十一章“基于路由表的分类器”一节），用于跟踪与一个标签（tag）关联的路由上的流量的统计信息，统计信息包括封包个数和字节数。这个结构包含一个计数器数组，每个处理器有256 个元素（注3）。数组大小为256是因为路由标签的取值范围为0～255。IPv4中是由ip_rt_init函数为该数组分配空间，但IPv6中不为该数组分配空间。ip_rt_acct结构中的四个字段由ip_rcv_finish更新，详见第三十五章“策略路由与基于路由表的分类器”一节。

struct rt_cache_stat

存储路由查找的统计信息。对每个处理器有该数据结构的一个实例。虽然这个结构名称好像是只统计与路由表缓存相关的计数器，但它的一些实例被用于更通用的路由查找的统计，参见第三十六章“统计数据”一节。

struct inet_peer

维护远程IP 端点（peer）的长期信息。该数据结构在第二十三章“长期IP端点信息”一节中描述。

struct fib_result

对路由表查找后返回该结构。它的内容并不是简单地包含下一跳信息，而且包含有其他特性，例如，策略路由所需的更多参数。

struct fib_rule

表示由策略路由在路由流量时选择路由表的规则，参见第三十一章“策略路由背后的概念”一节。

struct flowi

flowi从某种程度上讲类似于访问控制列表（ACL）：它是根据从L3与L4包头中选择的字段值，例如，IP 地址、L4 端口号等，来定义一个流量集。可以将它用来路由查找时的搜索关键字。

下面给出构造路由表的数据结构，这些结构之间的关系在第三十四章中有更详细的阐述。

struct fib_node

一条路由表项。例如，该数据结构用于存储由 *route add* 或 *ip route add* 命令添加一条路由时生成的信息。

注 3：　不要将名称相同的数组和结构混淆。

struct fn_zone

一个zone表示子网掩码长度相同的一组路由。因为子网掩码占用32位（对IPv4来说），因而每个路由表有33个zone。这样，到子网10.0.1.0/24与10.0.2.0/24的路由都将放在24位的zone链表（即第25个zone）内，到子网10.0.3.128/25的路由将放在25位的zone链表内。

struct fib_table

表示一张路由表。不要将它与路由表缓存混淆。

struct fib_info

不同路由表项之间可以共享一些参数，这些参数被存储在fib_info数据结构内。当一个新的路由表项所用的一组参数与一个已存在的路由项所用的参数匹配时，则重复使用已存在的fib_info结构。一个引用计数用于跟踪该结构重复使用的次数，参见第三十四章中图34-1所示的例子。

struct fib_alias

到达相同的网络的路由，因为其他一些参数而不同，比如TOS，不同的路由是通过fib_alias实例来区分的。

struct fib_nh

表示下一跳。如果使用命令定义了一条路由，比如*ip route add 10.0.0.0/24 scope global nexthop via 192.168.1.1*，那么下一跳为192.168.1.1。一条路由通常只有一个下一跳，但当内核支持多路径特性时，那么，就可以对一条路由配置多个下一跳。参见第三十一章“多路径路由背后的概念”一节。

struct fn_hash

该结构内包含指向33个fn_zone链表头的指针，以及一个链表。这个链表将活动zone（active zone，即那些至少有一个元素的zone）链接在一起。链表中的元素按照子网掩码长度的降序排列，参见第三十四章中图34-1。

下面这些数据结构用于与协议有关的路由缓存代码和与协议无关的路由缓存代码（DST），更详细的描述可参见第三十三章。

struct dst_entry

表示路由表缓存项中与协议无关的部分（DST）。适用于任何L3协议（例如，IPv4、IPv6、DECnet）的路由表缓存项字段被放在该结构内。在L3协议所用到的数据结构内，通常嵌入该结构来表示路由表缓存项。

struct dst_ops

表示DST核心代码使用虚函数表（VFT），该表用于向三层协议通知特定的事件（例如，链路失效）。每个三层协议各自提供一组函数，按照自己的方式来处理这些事

件。并不是所有的协议都使用 VFT 的每一个字段，参见第三十三章中“DST 与调用协议间的接口”一节。

`struct rtable`

在 IPv4 中表示一条路由表缓存项的数据结构（注 4）。

下面的数据结构通常是用于路由配置：

`struct kern_rta`

当内核接收到添加或删除一条路由的请求时，这个请求来自用户空间的一条 IPROUTE2 命令，内核就解析请求并将其存储到 `kern_rta` 结构内，参见第三十六章“inet_rtm_newroute 与 inet_rtm_delroute 函数”一节。

`struct rtentry`

当使用 *route* 命令向内核发送添加或删除路由表项的请求时所用的数据结构。IPROUTE2 中的 *ip route* 命令使用的是另一个不同的数据结构。

`struct fib_iter_state`

当遍历组成一张路由表的数据结构实例时，用该结构存储上下文信息。处理 /proc 接口的代码中使用该数据结构。

下面的数据结构用于多路径缓存特性，这些结构在第三十三章中描述：

`struct ip_mp_alg_ops`

表示多路径缓存算法。该结构由一些函数指针组成。

`struct multipath_device`

在设备循环缓存算法中用于跟踪设备信息。

`struct multipath_candidate`

`struct multipath_dest`

`struct multipath_bucket`

`struct multipath_route`

在加权随机缓存算法中用于算法所需的状态信息。

链表与 hash 表

路由代码还用到了其他两个通用的数据结构。在本书的这一部分将会经常看到它们，所以这里简单介绍一下。

注 4：IPv6 中使用 `struct rt6_info`。

hlist_head
hlist_node

Hash 表是由这两种数据结构实现的。hash 表的 bucket 定义为 hlist_head 类型，添加到 hash 表中的实际元素中要嵌入一个类型为 hlist_node 的元素，以便将实际元素链接到 hash 表中。这两种类型的唯一区别在于 hlist_head 只包含一个前向指针，而 hlist_node 包含前向和反向指针。

因而，在 hash 表 bucket 中的链表是双向的。由于 head 没有后向指针，链表不能成环，所以扫描到链表尾部代价很高，但这对hash表而言并不是问题。在hash表bucket的head结构中没有了反向指针，这种实现方法将bucket占用的空间缩小了50%，因而同样的内存可以存储的 bucket 容量就能够翻一倍。

图32-2给出了一个利用hlist_head和hlist_node数据结构构造的hash表例子。注意hlist_node结构不必是它所链接结构的第一个字段。

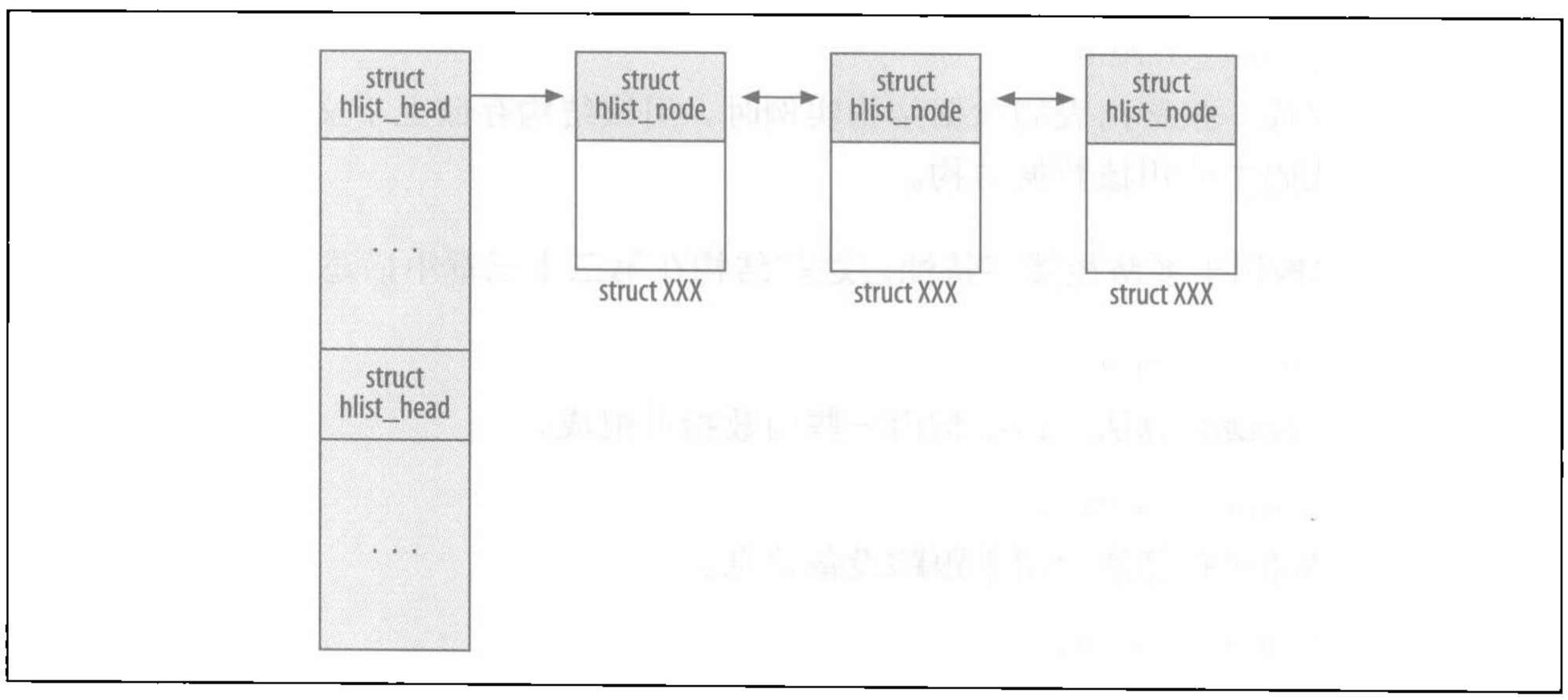


图 32-2：常见的 hash 表及列表的使用

路由代码中定义的 hash 表并不是都与图 32-2 中的结构相同：在第三十四章中将看到路由表及其数据结构组织中涉及hash表的一些例子，在第三十三章中将看到路由缓存使用了自己定义的 hash 表。

在 head 元素中也可以用 list_head 结构来实现包含两个指针的链表，可以从 *include/linux/list.h* 文件中找到有关这两种链表的更多细节，包括最常见的操作程序（添加、删除、遍历等）的定义。

路由 scope 与地址 scope

在第三十章“Scope”一节中我们已经介绍了 scope，这里来看在那一节中描述的 scope 在内核中是如何描述的，以及使用它的几个例子。

在内核 *include/linux/rtnetlink.h* 文件内，定义了一个 rt_scope_t 枚举变量，给出了可能的scope。取值范围是0～255，其中的0（RT_SCOPE_UNIVERSE）表示broadest scope。内核实际上只使用了其中的几个值，其他的值留给用户使用；目前这些留给用户的值还没有实际的应用。

路由 scope

路由的scope被保存在 fib_alias数据结构内的 fa_scope字段（参见第三十四章中的图 34-1）。下面按照 scope 的递增顺序给出 IPv4 路由代码中使用的主要 scope：

RT_SCOPE_NOWHERE

在第三十章并没有给出这个值，它被代码视为非法 scope。它的字面含义是路由项不通往任何地方，这基本上就意味着没有到达目的地的路由。

RT_SCOPE_HOST

scope 为 RT_SCOPE_HOST 的路由项的例子：为本地接口配置 IP 地址时自动创建的路由表项（参见“添加一个 IP 地址”一节）。

RT_SCOPE_LINK

为本地接口配置地址时，派生的目的地为本地网络地址（由子网掩码定义）和子网广播地址的路由表项的scope就是 RT_SCOPE_LINK（参见“添加一个IP地址”一节）。

RT_SCOPE_UNIVERSE

该scope被用于所有的通往远程非直连目的地的路由表项（也就是需要一个下一跳网关的路由项）。

地址 scope

地址的 scope 被保存在 in_ifaddr 结构内的 ifa_scope 字段。对设备上配置的每一个IP地址对应一个 in_ifaddr实例（参见第十九章）。我们在第三十章给出了与主要scope对应的一些地址例子。

路由表项中的下一跳网关是另一个具有 scope 的对象类型。每一条路由项可以有零个、一个或多个下一跳，每个下一跳是由一个 fib_nh 结构表示（参见第三十四章中的图

34-1)。fib_nh结构中有nh_gw和nh_scope两个字段：nh_gw是下一跳网关的IP地址，nh_scope是该地址的scope(这两个字段表示了从本地主机到这个下一跳网关的路由表项的 scope)。

路由 scope 与下一跳 scope 之间的关系

尽管路由 scope 和本地配置地址 scope 或者是由用户通过命令来设定，或是由内核分配一个默认值，但路由下一跳（fib_bh 结构）的 scope 总是由内核来设定。

在第三十四章“添加路由”一节中，将看到 fn_hash_insert 是如何将一条新路由表项插入到路由表中。这里可以将 fn_hash_insert 使用 fib_create_info 接口为新路由表项分配必要的数据结构，并初始化下一跳的 scope。后一个步骤是由 fib_create_info 在 fib_check_nh 接口的帮助下实现的。下一跳的 nh_scope 范围是根据配置的路由表项的 scope 计算出来的：一般来讲，给了一条路由及其下一跳，那么分配给 nh_scope 的值就是通往该下一跳的路由 scope。但也有一些特殊情况规则不同，例如，在到达本地配置地址及直连的路由表项中没有下一跳。

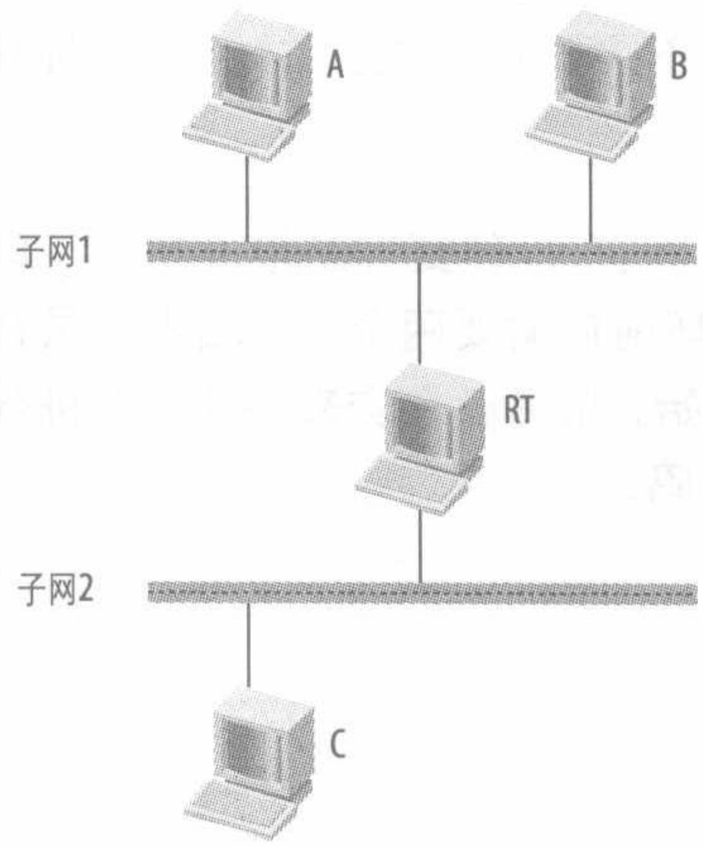
现在我们知道 nh_scope 是如何被初始化的，接下来看看如何利用它来加强对路由表项的合理性检查。

路由代码对路由 scope 和下一跳 scope 的合理性检查是在不同地方实现的。这些合理性检查主要是基于路由 scope 及其下一跳 scope 之间很有意思的关系。当主机转发一个 IP 包时，假定更接近最终的目的地（注 5）。根据这一简单性质，可以得出路由 scope 总是应当大于或等于其下一跳的 scope。

我们使用图 32-3 中的拓扑结构来看两个例子。记住对每一个路由，nh_scope 是下一跳的 scope，nh_gw 是下一跳的 IP 地址。

- 当主机 A 向主机 C 发送一个包时，匹配的路由 scope 为 RT_SCOPE_UNIVERSE，下一跳为 RT。为了使路由汇聚，到 RT 的路由 scope 必须比 RT_SCOPE_UNIVERSE 范围更小。因而，从路由表中查找主机 A 到主机 RT 的路由将返回 scope 为 RT_SCOPE_LINK 的路由项，它比 RT_SCOPE_UNIVERSE 的范围更小，因而是正确的。因为一条 RT_SCOPE_LINK 路由项不需要网关（事实上从主机 A 到主机 RT 的

注 5：这并不一定意味着物理上更接近。有时候，复杂的路由环境可能迫使封包穿越多个专门的设备，这可能需要多条非最优的路由表项。然而，假定给出了包从源到目的地的路径，那么转发包的每个系统都要使包朝着最终目的地跳跃步数超过 1。



1) A ⟶ C
路由scope: RT_SCOPE_UNIVERSE
nh_gw=RT
nh_scope= RT_SCOPE_LINK(<RT_SCOPE_UNIVERSE)
A ⟶ RT
路由scope:RT_SCOPE_LINK
nh_gw=0
nh_scope= RT_SCOPE_HOST(<RT_SCOPE_LINK)

2) A ⟶ A
路由scope: RT_SCOPE_HOST
nh_gw=0
nh_scope= RT_SCOPE_NOWHERE(<RT_SCOPE_HOST)

图 32-3：下一跳的 scope 初始化例子

路由项不需要网关），内核初始化 nh_gw 为 0，nh_scope 范围为一个比 RT_SCOPE_LINK 范围更小的值（例如，RT_SCOPE_HOST）。

当主机 A 向自己发送一个包时，匹配的路由 scope 为 RT_SCOPE_HOST。此时不需要网关，所以 nh_gw 被设置为 0，nh_scope 被设置为比 RT_SCOPE_HOST 更小的值：RT_SCOPE_NOWHERE。

当路由查找结果为一条直连路由（即不需要下一跳网关）时，上述递归过程终止。下面给出两种可能的情况：

路由查找返回 scope 为 RT_SCOPE_HOST *的路由项。*

此时，目的地是本地配置的地址，所以主机可以本地提交该封包。

路由查找返回 scope 为 RT_SCOPE_LINK *的路由项。*

因为目的地为直连，所以不需要网关，主机可以使用二层协议直接向目的地发送包（注 6）。

主 IP 地址与辅助 IP 地址

我们在第三十章“主地址与辅助地址”一节中看到，在设备上可以将一个 IP 地址配置为主 IP 或辅助 IP。内核常常需要遍历一个设备上配置的所有地址，来查找与某个条件相匹配的地址。我们下面来看如何区分这两种类型地址及如何来遍历地址。

注 6：　这包括第三十三章中描述的直连路由。

IPv4的辅助地址在其in_ifaddr数据结构内带有IFA_F_SECONDARY标识（参见第十九章）。因为只有主IP与辅助IP这两种配置，所以不需要有IFA_F_PRIMARY标识：如果一个地址不是辅助IP，那么被视为主IP。

内核在*include/linux/inetdevice.h*文件中提供了一些宏，以便更容易地遍历接口以满足指定的标准。对选择的每一个标准，通常在一个循环前后需要两个宏：编程人员在处理单个地址的代码放到一个块中，该块用一个宏来开始，用另一个宏来终止。对符合选择标准的每个地址，运行一个循环就可以使用这段代码。

下面是一个使用宏的例子：

```
for_ifa {
        do something with ifa
} endfor_ifa
```

for_ifa宏开始一个循环，变量ifa表示选择的每一个地址。两个宏之间的代码不需要放在花括号内部，但通常都放在花括号内部，这是为了使循环内使用的诸如ifa等变量只在该花括号内使用。

这些宏包括：

for_ifa
endfor_ifa

给定一个设备，用这两个宏来遍历其in_device数据结构上的所有地址（注7）。

for_primary_ifa
endfor_ifa

给定一个设备，用这两个宏来遍历只与主IP地址相关的in_device实例。

常用的辅助程序与宏

路由代码使用了许多小程序和宏，这使得代码的可读性更好。本节列出其中常用的一部分，更特殊的部分将在后面的“辅助程序”一节中介绍。重要的是必须牢记同一个函数或宏可能有不同的定义，这主要是依赖于以下这些因素：

- 内核对策略路由的支持。

注7： 在第十九章我们看到in_device数据结构被用于存储一个网络设备上的IP配置。

- 内核对多路径路由的支持。
- 三层协议（例如，IPv4 和 DECnet）

常用的程序和宏有：

FIB_RES_*XXX*

这些宏从一个给定的fib_result结构中提取特定的字段，例如，FIB_RES_DEV提取出 nh_dev 字段。这些宏定义在 *include/net/ip_fib.h* 文件中。

change_nexthops
for_nexthops
endfor_nexthops

这些宏用于对给定的一个 fib_info 实例遍历所有的 fib_nh 结构。change_nexthops 宏开始一个针对这些 fib_nh 结构的循环，用局部变量 nh 表示每个 fib_nh结构。就像从宏名中所给出的，该宏可以用于改变这些结构。for_nexthops宏与change_nexthops宏类似，它们都是用相同的endfor_nexthops宏来结束循环。唯一的区别在于 for_nexthops 将局部变量 nh 定义为一个常量指针，因而循环内的代码不能改变所遍历的任何 fib_nh 实例的内容。

对于IPv4，这些宏被定义在 *net/ipv4/fib_semantics.c* 文件中。注意对每一个宏都有两个版本：一个在内核支持策略路由时使用，一个是不支持策略路由时使用。由于考虑到没有策略路由时，每个 fib_info 实例总是最多只有一个 fib_nh 实例（即每条路由最多只有一个下一跳），因而后一个版本的宏经过优化。

inet_ifa_byprefix

给定一个设备、一个 IP 地址前缀与一个子网掩码，该函数会遍历配置在该入口设备上的所有主IP地址，来查找一个与输入的地址前缀和子网掩码相匹配的IP地址。查找成功时，它返回匹配的地址。

fib_get_table

给定一个路由表标识符（0～255 间的一个数），该函数从 fib_tables数组返回相应的 fib_info 结构，如第三十四章中图 34-1 所示。该函数定义在 *include/net/ip_fib.h* 文件中。

fib_new_table

这个函数创建并初始化一个新路由表，并将其链接到 fib_tables数组内（参见第三十四章中的图 34-1）。

LOOPBACK

ZERONET

MULTICAST

LOCAL_MCAST/BADCLASS

定义在 *include/linux/in.h* 文件中的这些宏用于将 IP 地址快速地划分到众所周知的类别中，参见第十章中的表 30-1 与表 30-2。

LOOPBACK 识别 127.*x.x.x* 地址。

ZERONET 识别 0.*x.x.x*/8 地址，这些地址大部分情况下不是合法地址。

MULTICAST 识别 D 类地址。

LOCAL_MCAST 识别 D 类中用于本地多播的地址子集：224.0.0.0/24。

BADCLASS 识别 E 类地址。

全局锁

路由代码使用了一些锁来保护竞争条件。下面只列出了一些全局锁，那些嵌入在数据结构内（即用于单个表项）的锁将在描述相关数据结构时讨论。

fib_hash_lock

这个读写旋转锁（rwlock）保护所有的路由表。例如，插入一个新的 fib_node 实例要求在互斥模式下获得该锁，而路由表查找只需要在共享模式下获得该锁。因为对所有的路由表只有这一个锁，也就意味着不可能将两条路由表项同时添加到两个不同的路由表内。但这并不是一个瓶颈，因为配置改变是小概率事件，用户主要使用共享锁，因而不会对路由器性能产生任何大的影响。

fib_info_lock

这个读写旋转锁保护所有的 fib_info 数据结构。例如，在第三十四章“fib_info 结构的组织”一节中描述的通过 hash 表来访问 fib_info 结构时，就使用该锁。

fib_rules_lock

这个读写旋转锁保护 fib_rule 数据结构的 fib_rules 全局链表。

rt_flush_lock

rt_cache_flush 函数使用这个旋转锁保护对 rt_deadline 全局变量和 rt_flush_timer 定时器的操作。缓存是由每个 bucket 自己的锁来保护，参见第三十三章中的图 33-1 及“刷新路由缓存”一节。

fib_multipath_lock

当修改 fib_info 结构中由多路径特性使用的字段时，使用这个旋转锁。

alg_table_lock

这个旋转锁用于串行化访问 ip_mp_alg_table 向量。它用于 multipath_alg_register 和 multipath_alg_unregister 函数中。参见第三十三章“注册一个缓存算法”一节。

路由子系统初始化

Ipv4 路由代码的初始化从 *net/ipv4/route.c* 文件中的 ip_rt_init 开始，它在系统启动时被 *net/ipv4/ip_output.c* 文件中初始化 IP 子系统的 ip_init 调用。ip_init 在第十九章中介绍过，这里我们来看看 ip_rt_init（注 8）。图 32-4 给出了如何调用主要的路由初始化函数。

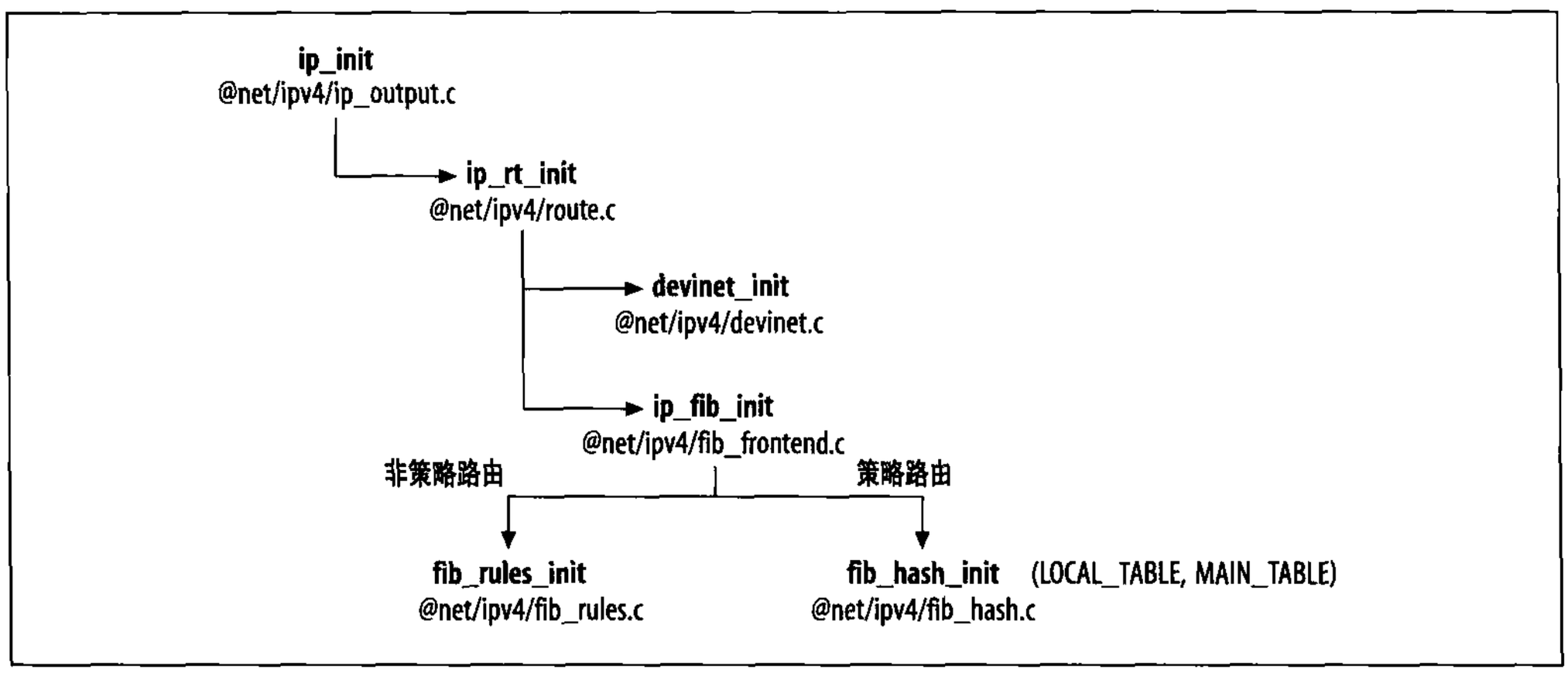


图 32-4：主要的路由初始化函数的调用顺序

在 ip_rt_init 中，IPv4 路由代码初始化其数据结构和全局变量。另外，该函数还有下列功能：

- 根据可用的内存确定路由缓存的容量。
- 创建用于分配路由缓存元素的内存池。
- 初始化路由缓存。

注 8：IPv6 做法类似，是在 inet6_init 中调用 ip6_route_init。

- 确定由垃圾回收算法使用的gc_thresh阀值(参见第三十三章“rt_garbage_collect函数”一节)。
- 启动rt_periodic_timer 定时器（参见第三十三章“垃圾回收”一节)。
- 启动rt_secret_timer 定时器（参见第三十三章“刷新路由缓存”一节)。
- 添加一些文件到*/proc*文件系统(参见第三十六章“通过/proc文件系统调整”一节)。

下面给出特别感兴趣的两个函数：

ip_fib_init

初始化默认路由表，用两个通知链（注9）netdev_chain与inetaddr_chain(参见“外部事件”一节）注册两个处理函数。

devinet_init

使用通知链netdev_chain注册另一个处理函数，用Netlink套接字为地址和路由命令（即 *ip addr ...* 与 *ip route ..* 命令）注册处理函数，并创建 */proc/sys/net/conf* 和 */proc/sys/net/conf/default* 目录。参见第三十六章是如何来实现后两个任务。

当内核编译时支持IPsec，那么ip_rt_init也调用一对IPsec初始化函数（xfrm_init与xfrm4_init)。

关于如何用 ip_rt_init 初始化 rt_hash_*xxx*全局变量的详细描述可参见第三十三章“路由缓存初始化”一节。

策略路由的初始化用*net/ipv4/fib_rules.c*文件中定义的fib_rules_init函数。其初始化就是简单用netdev_chain 通知链注册一个处理函数。注册的处理函数为fib_rules_event，它的描述可参见“对策略数据库的影响”一节。

外部事件

在网络协议栈中路由子系统处于中心地位。因此，需要知道发生的网络变化何时会影响到路由表和路由缓存。网络拓扑结构变化是由运行在用户空间的可选的路由协议来处理，而另一方面，本地主机的配置变化也需要让内核知道。

路由子系统尤其对两类事件感兴趣：

- 网络设备状态的变化。
- 网络设备上IP配置的变化。

注9： 通知链在第四章中描述过。

为了在这两种变化发生时收到通知，路由子系统分别用 netdev_chain 和 inetaddr_chain 通知链向内核注册。在“设备状态改变”和“IP 配置改变”两节中，详细讨论了这两类事件向内核注册的处理函数。

图 32-5 从高层描述了在 ip_rt_init 函数中注册的两个处理函数，它们在“对路由表的影响”和“对 IP 配置的影响”两节中描述。一些事件通过调用一些不同输入参数的辅助程序来处理，其中的一些辅助程序在下面的“辅助程序”一节中描述，对 fib_sync_down 的描述可参见这一节，force 参数的含义在图 32-5 中给出。

我们将看到许多事件都会刷新路由缓存。参见第三十三章“刷新路由缓存”一节来查看描述这些事件的完整列表。

辅助程序

在下面一节中，我们将看到 fib_netdev_event 和 fib_inetaddr_event 的实现细节。本节也简要概述这两个处理函数会调用的一些其他函数，你在阅读这两个处理函数的代码时可以参考本节。

void rt_cache_flush(int delay)

在一段由输入参数指定的时间后，安排刷新一次路由缓存的。参见第三十三章“刷新路由缓存”一节。

int fib_sync_down(u32 local, struct net_device *dev, into force)

当一个设备被关闭或删除一个本地地址时更新路由表。下面给出其输入参数的含义：

local

被删除的本地 IP 地址。

dev

被关闭的设备。

force

决定何时执行相应的行为。参考图 32-5 来理解下面每个值在何时使用。它们的含义如下：

0：一个 IP 地址已经被删除。

1：一个设备已经被关闭。

2：一个设备已经被注销。

这个 force 参数用于确定两类事情：

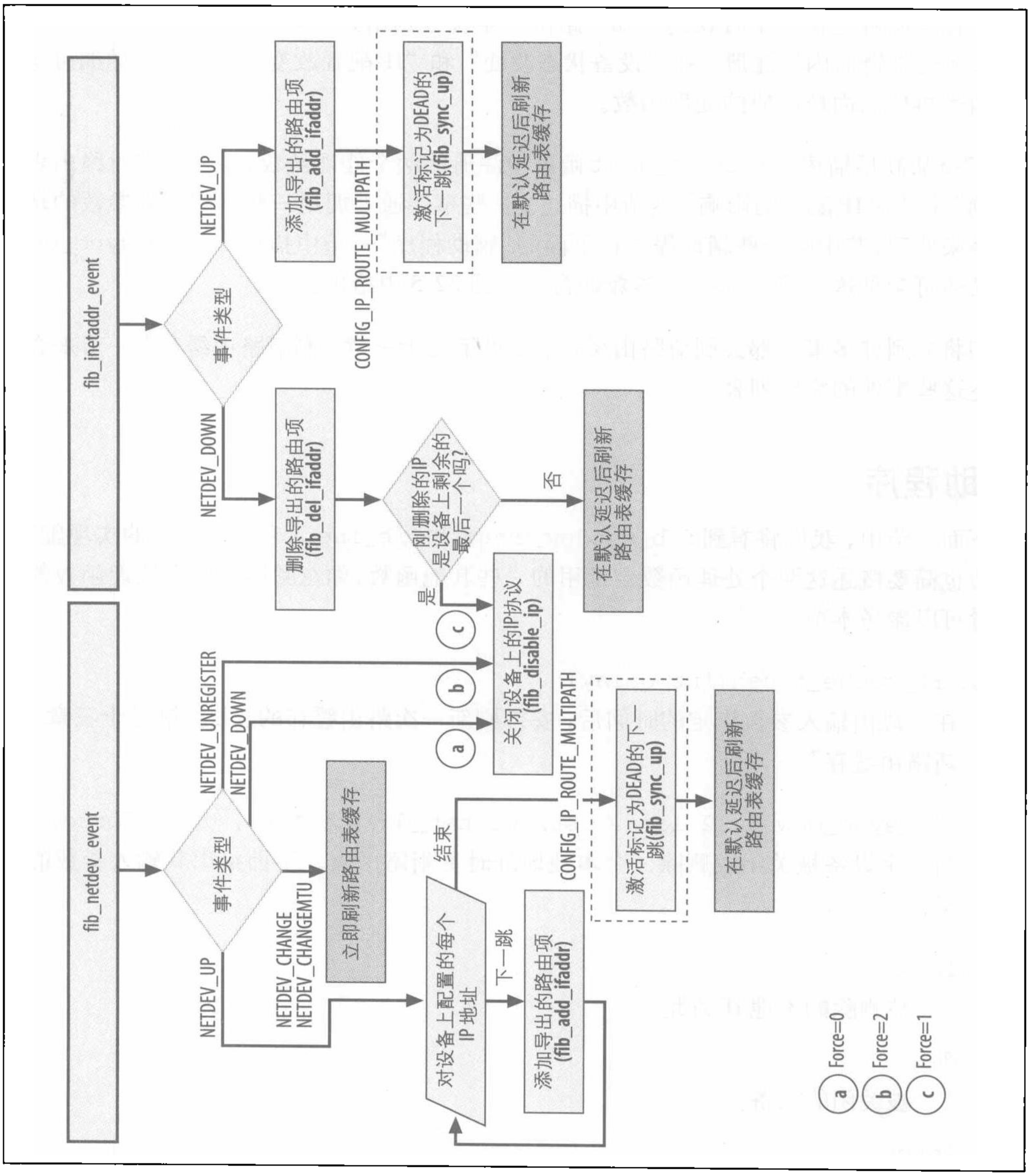


图 32-5：fib_netdev_event 与 fib_inetaddr_event 函数

被删除路由表项的 scope

当 force 为 0 时，处理函数 fib_sync_down 就删除所有合法的路由表项，除了通向本地配置地址的路由表项（即路由项 scope 为 RT_SCOPE_HOST）。当 force 为 1 或 2 时，处理函数删除所有合法的路由表项，而不管路由 scope。

如何处理多路径路由

当force为2时，如果一条多路径路由的下一跳中至少有一个使用了输入参数dev，处理函数就删除该多路径路由项。当force为0或1时，只有当一条多路径路由的所有下一跳都为dead时，处理函数才删除该路由项。

通常调用fib_sync_down时，它一次只处理其中一类事件，所以或者设置dev参数，或者设置local参数，而另一个参数为空。

当设置local参数时，fib_sync_down 删除所有以local为首选源地址的路由表项。记住路由项可以被指定一个首选源地址，该地址不必是该路由项出口设备上配置的地址。

当设置dev参数时，fib_sync_down删除所有的下一跳是通过dev到达的路由表项。

在这两种情况下都不是直接删除路由表项，而是通过设置RTNH_F_DEAD标识将它们标记为dead(不可用)。只有当一条多路径路由的所有下一跳都被标记为dead时，该路由项才被标记为dead。而且，当一条多路径路由的一个下一跳被标记为dead时，也必须更新fib_power与nh_power参数，以反映当前下一跳的状态（参见第三十五章“多路径对下一跳选择的影响”一节)。

fib_sync_down返回值是标记为dead的fib_info结构数量。例如，该函数的调用者(比如本节后面描述的fib_disable_ip)可利用该返回值来决定是否刷新路由缓存。

int fib_sync_up(struct net_device *dev)

只有在内核支持多路径路由时才使用该函数。它的主要工作是在路由的某些下一跳为alive时，更新fib_info结构内该路由的一些参数。它的返回值为RTNH_F_DEAD，标识被清除的fib_info结构的数量。

void fib_flush(void)

扫描ip_fib_main_table 与ip_fib_local_table路由表，删除所有的设置有RTNH_F_DEAD标识的fib_info结构。它既删除fib_info结构，也删除相关联的fib_alias结构。当一个fib_node实例不再有关联的fib_alias结构时，就删除该fib_node实例。fib_flush调用的默认函数可参见第三十四章中的表34-1，上述数据结构之间的关系可参见第三十四章中的图34-1。

当内核支持多路径路由时，fib_flush就扫描所有的路由表。

当至少有一个fib_info实例被删除时，用rt_cache_flush刷新路由缓存。

fib_flush的返回值是被删除的fib_info实例的数目。

```
static void fib_disable_ip(struct net_device *dev, int force)
```

该函数调用fib_sync_down来禁止输入参数dev设备上的IP协议。当删除的路由项数量为正数时（通过fib_sync_down的返回值判断），该函数也立即刷新路由表：fib_sync_down将路由项标记为dead，实际上由fib_flush执行删除操作。fib_disable_ip也立即刷新路由缓存，并要求ARP从自己的缓存中清除一些表现，这些表项引用了被关闭IP协议的设备。

注意输入参数force被传递给fib_sync_down。图32-6给出了fib_disable_ip函数的内部流程。

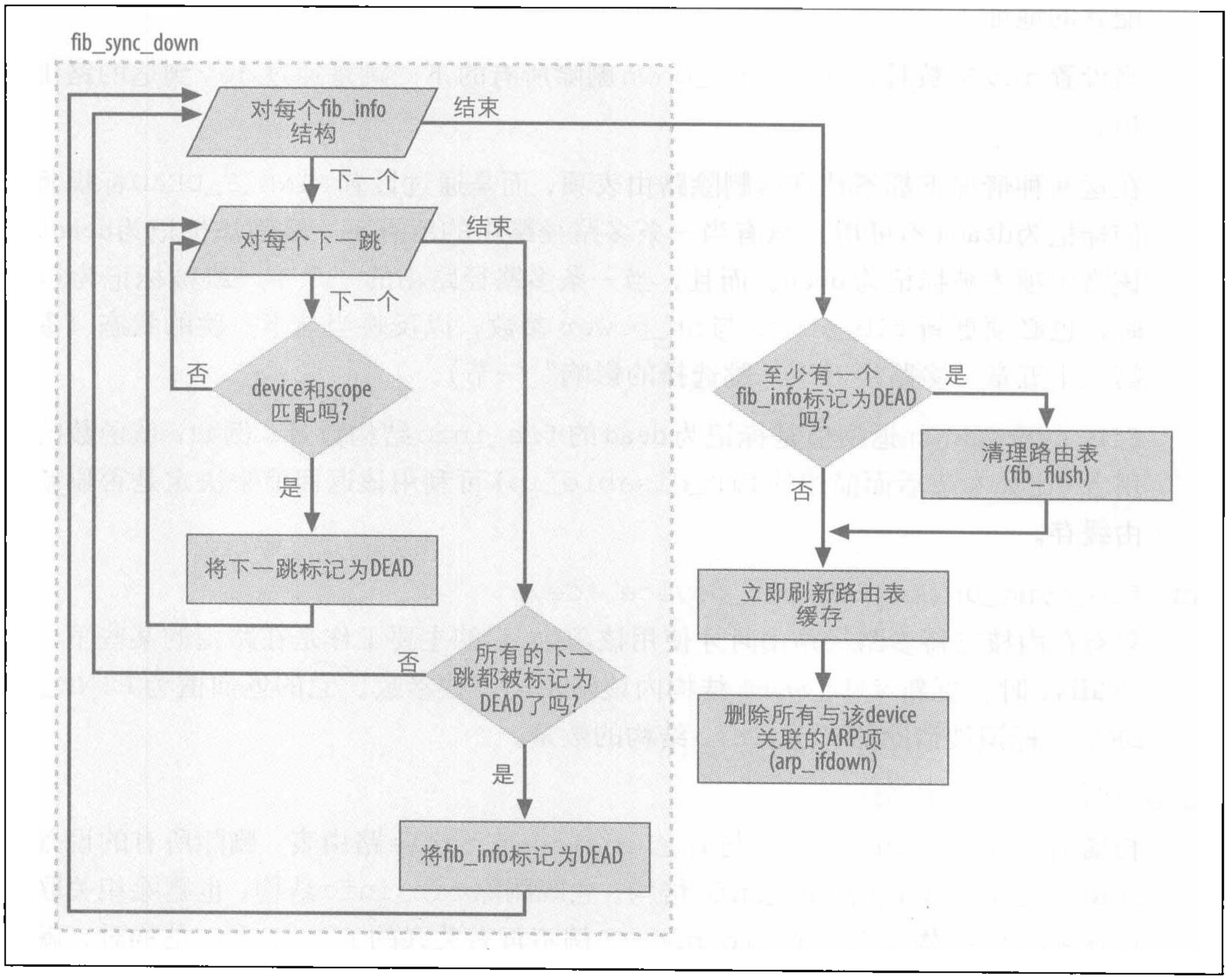


图 32-6：fib_disable_ip 函数

IP 配置改变

只要设备的IP配置发生变化，路由子系统就收到一个通知，并运行fib_inetaddr_event来处理该通知。图32-5总结了由这个通知链传送的所有可能事件所触发的活动，这里给出如何对主要的事件进行处理：

NETDEV_UP

本地设备上已经配置了一个新的IP 地址。处理函数必须将必要的路由项添加到local_table路由表中。完成这个任务的处理函数是 fib_add_ifaddr。

NETDEV_DOWN

本地设备上已经删除了一个IP地址。处理函数必须将以前由NETDEV_UP事件添加的与该地址有关的路由项删除，完成这个任务的处理函数是 fib_del_ifaddr。

在第三十章“特殊路由”一节中提到，每当一个接口上配置了一个IP地址时，内核就将一组特殊的路由项添加到一个独立的名为ip_fib_local_table的路由表中。负责添加这些特殊路由项的程序是fib_add_ifaddr，它对每一条新路由表项调用fib_magic来完成添加，fib_magic 在第三十六章中介绍。

在事件处理期间调用的大部分函数已经在前面的“辅助程序”一节中描述过，下面一节描述 fib_add_ifaddr 和 fib_del_ifaddr。

添加一个 IP 地址

fib_add_ifaddr的逻辑流程见图32-7。当这个函数被通知一个设备上有了一个新地址时，该设备可能还没有被启动。是否添加到该设备的路由项依赖于该设备是否被启动。我们首先来看从一个IP地址可以导出什么路由，然后来看在设备为启动或关闭时要添加哪些路由项。下面给出与 IP 地址 10.0.1.1/24 有关的可能的路由作为例子：

到地址 10.0.1.1/32 的路由

简单讲，就是到特定主机地址的路由。

到网络地址 10.0.1.0/24 的路由

这是从IP 地址和其子网掩码导出的路由。在我们例子中，为 10.0.1.1 和 255.255.255.0 进行与运算的结果。

到广播地址 10.0.1.255/32 和 10.0.1.0/32 的路由

这代表了协议规范要求和实际情况之间的一个折中方法。

Linux 能够处理不同的需求：它能将到两类的广播地址（译者注：网络地址和受限广播地址）的路由添加到 ip_fib_local_table路由表中。注意：到网络地址路由和受限广播地址的路由间的区别是，它们的子网掩码不同（10.0.1.0/24 对10.0.1.0/32）。此外，用户还可以配置一个特定的广播地址。此时，fib_add_ifaddr程序除了添加上述到两种广播地址的路由外，还要添加一条到该特定广播地址的路由。

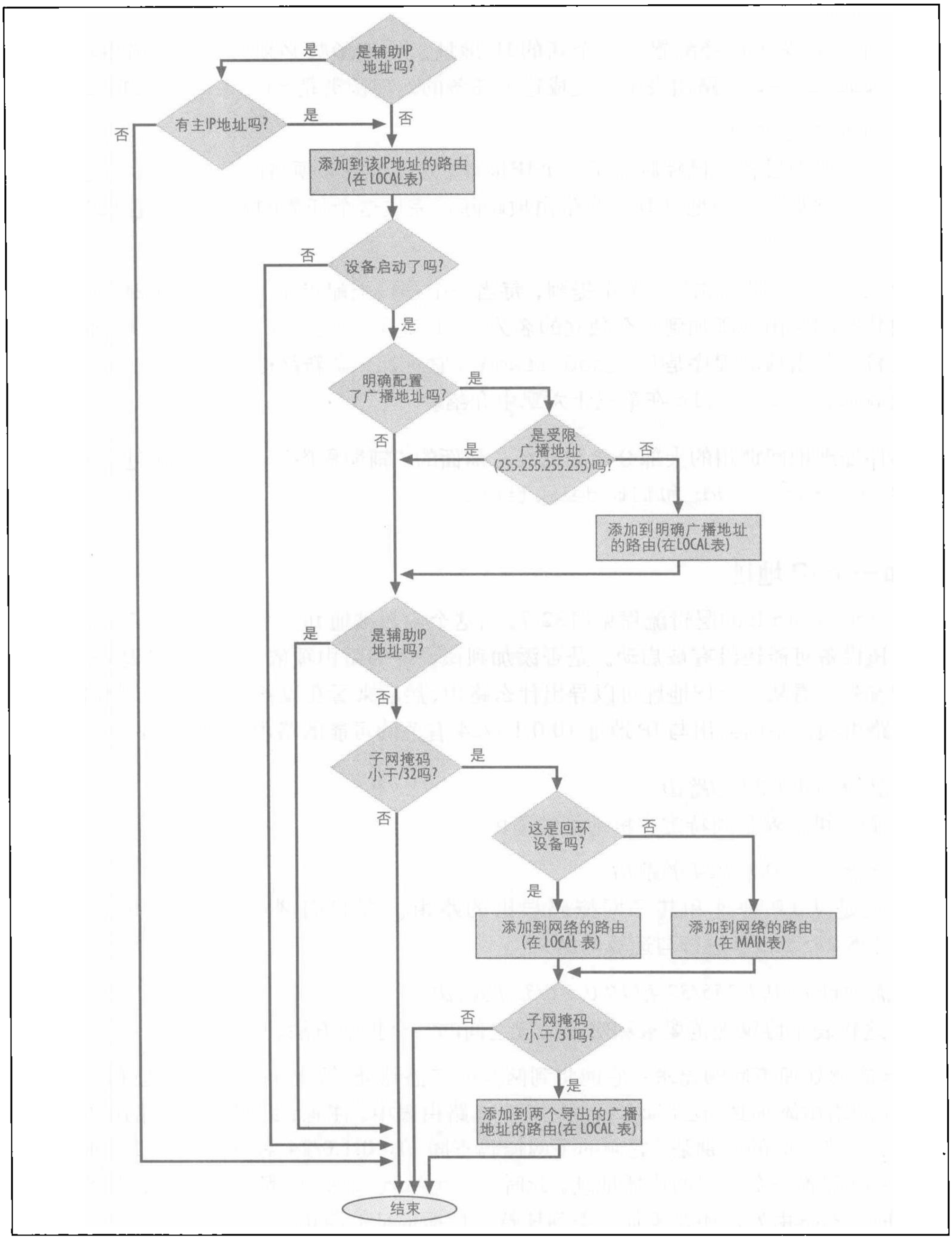


图 32-7：fib_add_ifaddr 函数

对广播地址的处理表明理论与实际间存在差距。从理论上讲，广播地址应该可以从地址类别导出。在我们的 10.0.1.1/24 地址中，它产生一个地址 10.255.255.255，这就是用 *ifconfig* 来配置地址时得到默认广播地址。

然而，通常你需要一个根据子网掩码导出的广播地址。这个地址是通过将网络地址的主机部分所有位都设置为 1 而得到的，在我们例子中将得到广播地址是 10.0.1.255。如果通过 *ip addr* 命令来配置地址，那么所生成的默认广播地址就是这个更为有用的广播地址。

两种方法出现差异是由于 10.0.0.1 是 A 类网络 10.0.0.0/8 的一个地址。而 A 类网络通常又被细分为更小的网络。例如，10.0.0.0/24 是 A 类网络 10.0.0.0/8 的一个 C 类子网。如果我们配置地址为 192.168.1.1，那么通过 *ifconfig* 和 *ip addr* 命令将得到相同的广播地址 192.168.1.255，因为 192.168.1.1 是 C 类网络 192.168.1.0/24 的一个地址。参见第三十章“路由的基本要素”一节中更详细的描述。

我们在第二十八章“多个网络接口的应答”一节已知道，IP 地址是属于系统，而不是属于所配置的那个接口。因此，到 IP 地址的路由总是被添加到 `ip_fib_local_table` 路由表内，而与设备状态无关。但是，到网络地址和广播地址的路由并非如此：当设备关闭时，网络地址和广播地址都不是可到达的，所以为它们创建两条路由是不正确的。`fib_add_ifaddr` 中使用设备的 `IFF_UP` 标识来判断该设备状态。

```
fib_magic(RTM_NEWROUTE, RTN_LOCAL, addr, 32, prim);
if (!(dev->flags&IFF_UP))
    return;
```

因此在一个关闭的设备上配置一个 IP 地址时，你只能添加到 IP 地址的路由。当该设备后来启动时，将再次调用 `fib_add_ifaddr` 来添加所有的路由。此时该函数将再次添加到该 IP 地址的路由，但因为路由表中不能够存在重复路由，所以这种情况不会发生路由重复添加。

注意：有时使用命令在设备上配置一个IP地址时，也可以启动设备。例如，通过 *ifconfig* 命令来给一个设备配置 IP 地址时，就可以启动该设备，在 IPROUTE2 中则分为两个步骤：首先使用 *ip addr add* 来配置一个 IP 地址，然后使用 *ip link set* 来启动或关闭设备。你在阅读源代码和试图领悟在内核代码行为如何响应输入的用户空间命令时，理解这些区别是很重要的。在第三十六章中有更详细的用户空间命令的描述。

表 32-1 列出了一些命令模板和创建的路由表项。

表32-1：IP配置和相关路由表的例子

命令	Main路由表	Local路由表
ip addr add 10.0.1.1/24 dev eth0	10.0.1.0/24	10.0.1.1/32 (address) 10.0.1.0/32 (broadcast) 10.0.1.255/32 (broadcast)
ip addr add 10.0.1.1/24 broadcast 10.0.1.100 dev eth0	10.0.1.0/24	10.0.1.1/32 (address) 10.0.1.100/32 (broadcast) 10.0.1.0/32 (broadcast) 10.0.1.255/32 (broadcast)
ip addr add 10.0.1.1/32 dev eth0		10.0.1.1/32 (address)
ip addr add 10.0.1.1/32 broadcast 10.0.1.255 dev eth0		10.0.1.1/32 (address) 10.0.1.255/32 (broadcast)

如果明确配置的广播地址正好与受限广播地址255.255.255.255一致时，就不会添加到该明确配置的广播地址的路由，因为路由查找程序要检查受限广播地址255.255.255.255，参见第三十五章“输入路由”与“输出路由”一节。

```
if (ifa->ifa_broadcast && ifa->ifa_broadcast != 0xFFFFFFFF)
    fib_magic(RTM_NEWROUTE, RTN_BROADCAST, ifa->ifa_broadcast, 32, prim);
```

明确配置广播地址甚至可以允许你定义一个属于/32子网的广播地址，该子网理论上只能有一个IP地址，参见表32-1中第四个例子（注意广播地址10.0.1.255不在10.0.1.1/32子网内）。

在某些条件下，函数fib_add_ifaddr不需要添加到广播地址的路由。这依赖于子网掩码长度，它保存为局部变量prefixlen中：

- 当prefixlen为32时，在子网内只有一个有效地址，所以不需要导出的广播路由或网络路由。
- 当prefixlen为31时，只有一个二进制位参与地址分配，所以在子网内只有两个地址。如果该位为0就表示网络地址，为1则表示主机地址（fib_add_ifaddr函数正在配置的地址）。这种情况下需要到这两个地址的路由，而不需要到导出的广播地址的路由。
- 当prefixlen小于31时，子网内包含的地址数大于或等于四个，但本地地址、网络地址和广播地址只占用其中三个。此时内核就添加到导出的广播地址的路由和到网络地址的路由。

下面的代码给出了对主地址出现上述情况时的处理：

```
        if (!ZERONET(prefix) && !(ifa→ifa_flags&IFA_F_SECONDARY) &&
            (prefix != addr || ifa→ifa_prefixlen < 32)) {
            fib_magic(RTM_NEWROUTE, dev→flags&IFF_LOOPBACK ? RTN_LOCAL :
                  RTN_UNICAST, prefix, ifa→ifa_prefixlen, prim);
            if (ifa→ifa_prefixlen < 31) {
                fib_magic(RTM_NEWROUTE, RTN_BROADCAST, prefix, 32, prim);
                fib_magic(RTM_NEWROUTE, RTN_BROADCAST, prefix|~mask, 32, prim);
            }
        }
    }
```

辅助 IP 地址不处理这些情况。当添加一个辅助 IP 地址时，在该设备上必须存在属于同一子网（IP 地址网络部分相同）内的一个主 IP 地址。如果没有这样的主 IP 地址，那么将出现错误，配置的辅助 IP 地址不能够生效。所以，辅助 IP 地址不需要到网络地址的路由，也不需要到导出的广播地址的路由：配置相关的主地址时已经添加了这些路由项。

删除一个 IP 地址

当从一个接口删除一个IP地址时，路由子系统得到通知以便清理路由表和路由缓存。负责这个任务的函数是 `fib_del_ifaddr`，其逻辑见图 32-8。

该函数首先进行合理性检查。若删除一个辅助 IP 地址，那么必须有一个主 IP 地址与它在同一子网。如果这个条件不满足，则前面某个地方一定出错了，因而返回一个错误。

当调用`fib_del_ifaddr`时，如果和被删除的IP地址关联的路由表项正被删除，则表明该地址已经从受影响设备上配置的地址列表中删除了（例如，当`inet_del_ifa`触发了`NETDEV_DOWN`事件通知时）。

在上一小节中提到，到广播地址和到网络地址的路由并不一定添加主 IP 地址而添加，`fib_del_ifaddr` 扫描设备上配置的所有地址，检查哪些需要删除。读者可以从表 32-1 中看出在配置一个本地 IP 地址时添加了哪些路由表项。

多数情况下，当删除一个辅助IP地址时，路由子系统只需要删除到该IP地址的路由，而不删除到网络地址和广播地址的路由，因为主IP地址（及其他可能存在的辅助IP地址）仍然需要它们。但还可能在删除一个辅助 IP 地址时，甚至不需要删除到该 IP 地址的路由：例如，管理员配置了一个具有两个不同子网掩码的IP地址就属于这种情况，见下面的例子：

```
# ip addr add dev eth0 192.168.0.1/24
# ip addr add dev eth0 192.168.0.1/16
```

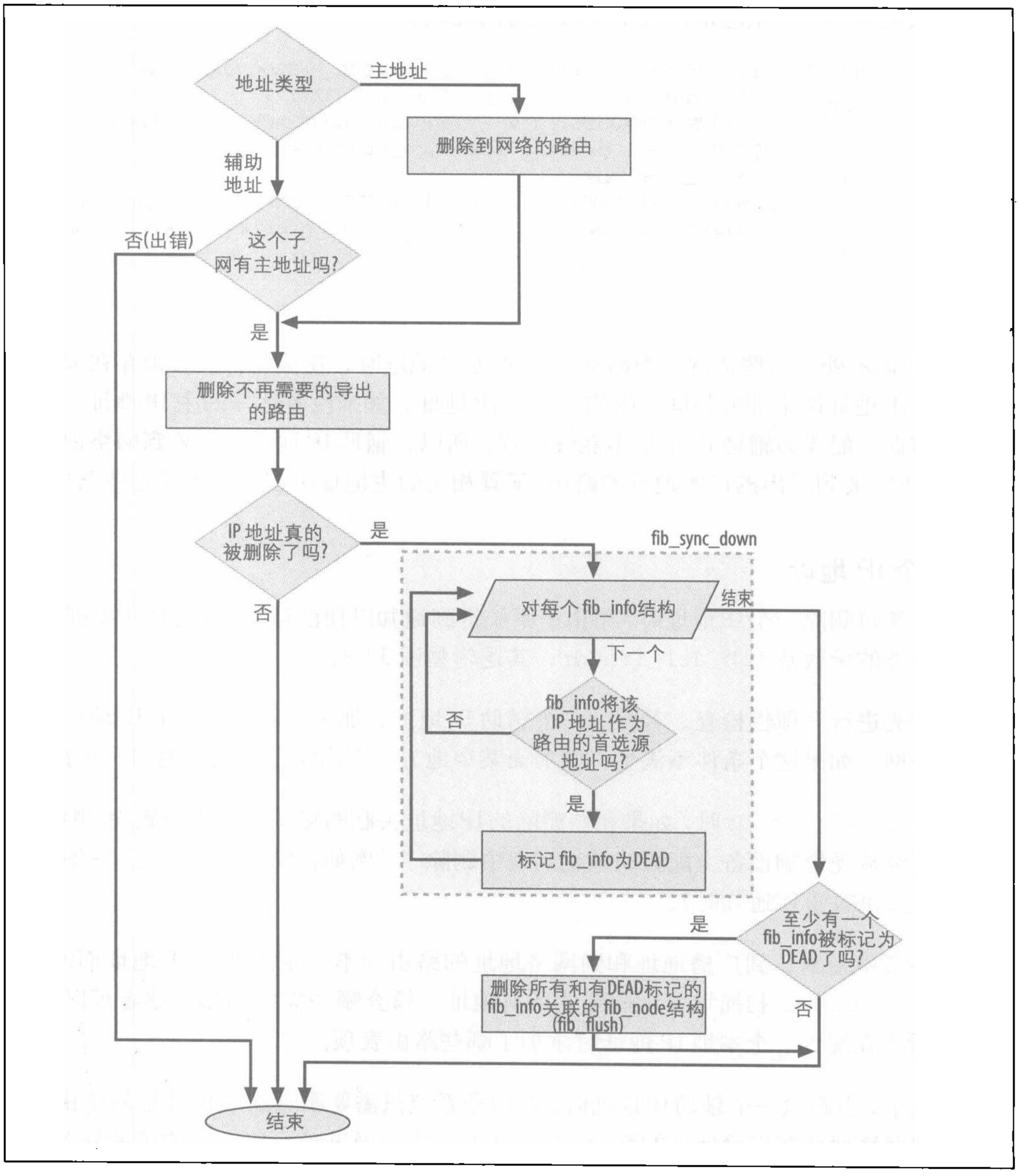


图 32-8：fib_del_ifaddr 函数

这个例子并不代表普通情况，但代码必须能够处理它。

通过这两个命令导出的到网络地址和到广播地址的路由是不同的（参见“添加一个IP地址”一节），但它们得到的到 IP 地址的路由是相同的。

fib_del_ifaddr在删除了相应的路由后，要调用 fib_sync_down和fib_flush来清理路由表。

当 fib_del_ifaddr从一个设备上删除最后一个 IP 地址时，fib_inetaddr_event 函数调用 fib_disable_ip 来关闭该设备上的 IP 协议（见图 32-5）。

设备状态改变

路由子系统向 netdev_chain 通知链注册三个不同的处理函数来处理设备状态的变化：

- fib_netdev_event 更新路由表。
- 当使用策略性路由时，fib_rules_event 更新策略数据库。
- ip_netdev_event 更新设备的 IP 配置（注 10）。

在下面的三小节中，描述这些程序是如何处理接收到的事件通知。

对路由表的影响

当一个设备的状态或其某些配置（除了IP配置，它是由另一个通知链处理）发生变化时，路由子系统将收到通知，调用 fib_netdev_event 来处理该事件。

图32-5总结了由这个通知链通知的所有可能事件所触发的行为，下面给出如何来处理这些主要事件：

NETDEV_UNREGISTER

当一个设备注销时，从路由表（包括路由缓存）中删除使用该设备的所有路由。如果多路径路由项的下一跳中至少有一个使用该设备，则该路由项也被删除。

NETDEV_DOWN

当关闭一个设备时，用fib_disable_ip从路由表（包括路由缓存）删除使用该设备的所有路由。

NETDEV_UP

当启动一个设备时，必须将与该设备上所有 IP 地址有关的路由表项添加到本地路由表 ip_fib_local_table 中。这是通过对配置在该设备上的每一个 IP 地址，都调用fib_add_ifaddr函数来完成的。fib_add_ifaddr的描述可参见“添加一个 IP 地址”一节。

注 10：这个处理函数是通过 ip_rt_init 函数注册的，但它实际上属于 IP 子系统。

NETDEV_CHANGEMTU

NETDEV_CHANGE

当一个设备改变其配置时，要刷新路由表缓存。其中最常见的配置变化是修改MTU值或PROMISCUITY状态。

注意：路由子系统对NETDEV_REGISTER事件不感兴趣。NETDEV_UP事件足以触发系统对一个新激活设备的处理。

注销一个设备与关闭一个设备对路由表产生不同的影响。注销设备的原因包括用户从内核中删除该设备的驱动程序，或拔出一块热插拔设备，比如PCMCIA以太网卡。关闭设备的原因包括用户拔出网线或发出一个管理命令关闭设备。在这两种情况下，从路由表中删除的路由是不同的。

我们来看一个例子。在表32-2中的第一列给出了使用下面两个命令添加到路由表中的路由，后面两列分别给出了当设备*eth0*关闭或注销时应当删除的路由。

```
# ip addr add dev eth0 192.168.1.100
# ip route add  10.0.1.0/24 via 192.168.1.111
```

表32-2：当设备关闭和注销时删除的路由

路由	路由表	关闭	注销
192.168.1.0/24	Main	是	是
192.168.1.0/32	Local	是	是
192.168.1.255/32	Local	是	是
192.168.1.100/32	Local	否	是
10.0.1.0/24	Main	是	是

当设备被关闭时到IP地址的路由没有被删除，这是因为它的IP地址属于主机而不是属于接口。只要与该地址相关联的设备存在，那么该地址就一直存在，参见第二十八章“多个网络接口的应答”一节。

对策略数据库的影响

一条策略（即一条规则）可以与一个设备相关联。例如，你可以为从设备*eth0*接收的流量和到子网10.0.1.0/24的流量指定一个特定的优先级。所以，当设备被注销时，与其相关的所有策略（即fib_rule数据结构）要标记为不可用，即通过fib_rules_detach来设置设备ID（fib_rule数据结构中的r_ifindex字段）为无效值−1。

另一方面，当设备注册时，如果有与该设备相关联的未激活的策略，则通过fib_rules_attach来重新激活。因为未激活的策略的设备ID为 -1，所以内核使用保存在fib_rule结构中的 r_ifname 字段里的设备名来识别与该策略相关联的设备。

对 IP 配置的影响

下面来看 inetdev_event 函数是如何处理从 netdev_chain 通知链接收到的通知：

NETDEV_UNREGISTER

禁用该设备上的 IP 协议。

NETDEV_UP

用 ip_mc_up 激活多播配置（如果存在的话）。当启动的设备为回环设备时，在其上配置地址 127.0.0.1/8。

如果启动的设备上配置的MTU小于启用IP协议的最小值68，则忽略该事件。这只是一项合理性检查。

NETDEV_DOWN

利用 ip_mc_down 来关闭多播配置（如果存在的话）。

NETDEV_CHANGEMTU

检查该设备上已设置的MTU是否低于运行IP协议的最小值68，如果是，则禁用该设备上的 IP 协议。

NETDEV_CHANGENAME

将目录 */proc/sys/net/ipv4/conf/*devname 和 */proc/sys/net/ipv4/neigh/*devname 的名称更新为新的设备名。这两个目录分别在第二十三章和第二十九章描述。

对于NETDEV_UNREGISTER与NETDEV_CHANGEMTU这两个事件，使用inetdev_destroy来禁用设备上的 IP 协议。该函数会删除该设备上所有的 IP 配置并调用 arp_ifdown 来清除对应的 ARP 缓存。

与其他子系统的交互

在第三十一章“与其他内核子系统的交互”一节中，给出了路由子系统与其他子系统，例如，流量控制和防火墙等子系统的交互。在下面的小节中，我们将看到更多的细节。路由子系统与基于路由表的分类器间的交互被延缓到第35章讨论，因为需要具备路由表结构与如何实现路由查找的一些背景知识。

Netlink 通知

当添加或删除一条路由时，利用函数 rtmsg_fib 向 Netlink 组 RTMGRP_IPV4_ROUTE 发送一个通知。创建路由和删除路由的通知分别是由 fn_hash_insert 和 fn_hash_delete 来生成，这是我们将在第三十四章看到的两个重要函数，也可以参见第三十六章“路由变化通知”一节。

策略路由与基于防火墙的分类器

在第三十一章“与其他内核子系统的交互”一节中提到，策略路由可以使用一个由防火墙代码初始化的标签（tag）来决定对入流量和出流量应当使用哪一张路由表。基于防火墙标签的路由需要内核在编译时要有特定的支持。当防火墙标签可用时，它就成为缓存和路由表查找关键字（用 flowi 结构来表示）的一部分。防火墙子系统将该标签拷贝到缓冲区字段 skb->nfmark 中，策略路由把该字段当作一个鉴别器，以确定对入流量和出流量的路由应当使用哪一张路由表。

在第三十三章，你将看到两个缓存查找函数 ip_route_input 和 __ip_route_output_key 是如何检查这个防火墙标签的。在第三十四章，给出了两个路由表查找程序 ip_route_input_slow 和 ip_route_output_slow 是如何利用防火墙标签 skb->nfmark 来初始化路由查找关键字 flowi 的 nfmark 字段。skb->nfmark 值为 0 表示没有标签。

在第三十一章中的图31-4给出了在网络协议栈中，防火墙何时根据它的配置对缓冲区添加标签，策略路由引擎何时使用该标签用于策略规则查找。

路由协议守护进程

路由可以由用户通过诸如 *ip route* 或 *route* 等命令，以及由用户空间中运行的诸如BGP、IGRP 与 OSPF 等路由协议来添加。我们在第三十一章“路由协议守护进程”一节的大图中已经看到这一点。在这一节，我们来详细描述用户 / 内核接口，而在第三十六章将更为详细地描述这些工具。但是，我们不会讨论任何路由协议的内部细节，因为这超出了本书的范畴。

路由协议运行在用户空间，但它们需要将自己的路由知识注入到内核中，以便将它们的路由合并到内核的路由表中。尽管路由协议代码与底层的操作系统无关，但在这些协议将路由注入到内核中时，必须采用由底层操作系统提供的用户 / 内核接口。

如果你喜欢阅读源代码，那么我劝你看看平台无关的代码（基本上就是指路由协议）是

如何与平台相关的代码交互，以便路由注入到内核的路由表中。例如，你将看到不同的操作系统要求有不同的接口，甚至同一个操作系统的不同版本可能要求或提供的接口也不同。

对 Linux 来说，老式的 ioctl 接口仍然可用，但内核首选的是新型的 Netlink 接口，因为它的功能更强大。尽管 ioctl 对所有的 Unix 爱好者来说是非常通用的接口，但 Netlink 是 Linux 特有的，它在 Linux 世界中扮演着与路由套接字在 BSD 世界中同样的角色。重要的是要注意当 Netlink 被编译到内核中时，它要优于 ioctl，因为它具有更好控制和双向通信能力（例如，使用 Netlink 就和在 BSD 中使用路由套接字一样，当内核检测到一个 NIC 的变化时，内核可以利用 Netlink 套接字将该变化通知给用户空间的应用程序，以便应用程序采取相应的行动）。

表32-3列出了最常见的路由协议守护进程，并给出了是否可以处理ioctl接口和Netlink接口。第三十六章将更为细致地讨论这些接口。

表 32-3：路由守护进程和 Linux 内核间最常用的接口

守护进程	ioctl	Netlink
ROUTED (v 0.17)	是	否
GATED (v3.6)	是	是
BIRD (v1.0.9)	是	是
ZEBRA (v0.94)	是	是
QUAGGA (v 0.98.0)	是	是
MRT (v2.2.0)	是	是
XORP (v1.0)	否[a]	是

a：XORP 使用 ioctl 时不能插入和删除路由。XORP 的用途和操作在文档 *http://www.xorp.org/releases/current/docs/fea/fea.pdf* 中有描述。

第三十三章

路由：路由缓存

路由缓存用于减少路由表查找的时间。路由缓存的核心是与协议无关的目的缓存(Protocol Independent Destination Cache DST)。尽管采用策略路由可有效地创建多张路由表，但所有这些路由表都共享一个路由缓存。

缓存的主要工作是存储使路由子系统能够找到封包目的地的信息，并将这些信息通过一组函数提供给上层协议。路由缓存还提供了一些函数用于缓存的清理。路由缓存存储的信息是可应用在所有三层协议的路由表缓存表项，所以，表示路由表缓存表项的任何数据结构都包含这些信息。

在本章中，我们将看到：

- 如何实现缓存。
- 如何插入新元素，以及如何删除已有的元素。
- 如何实现入口查找与出口查找，以及它们之间的差异。
- 外部子系统如何通过 DST 提供的接口与路由缓存交互。
- 不同的垃圾回收算法各自如何控制缓存容量。
- DST 是如何为出口 ICMP REDIRECT 消息提供限速机制。

路由缓存初始化

路由缓存是用 hash 表来实现的。它在路由子系统的初始化函数 `ip_rt_init` 中被初始化，参见第三十二章“路由子系统初始化”一节中的描述。

缓存容量取决于主机中可用的物理内存的大小。在你的系统中，你可以从系统启动时输出的信息或是在系统启动后从 *dmesg* 命令输出的信息中，找到由 `ip_rt_init` 函数输出

的“IP: routing cache hash table of ...”字符串，就得到hash表的容量。该值存储在rt_hash_mask中，其以2为对数的值被保存在rt_hash_log中（即$2^{rt_hash_log}$=rt_hash_mask）。内核指定的默认容量可以被用户启动选项*rhash_entries*所覆盖，该选项保存的是hash表的容量，变量rhash_entries要用到这个选项的值。

ip_rt_init主要是初始化以下变量：

rt_hash_table

用hash表来定义的路由缓存。

rt_hash_mask

rt_hash_log

分别表示hash表rt_hash_table的容量（即bucket的数目）和该容量以2为对数所得的值，当一个值必须通过二进制位数来移位时后一个值通常很有用。

rt_hash_rnd

当每次利用rt_run_flush来刷新路由缓存时，生成一个新的随机值，它保存在rt_hash_rnd参数中。该参数被用于防止DoS攻击，它是路由缓存中元素分布算法的一部分，使元素分布没有确定性。该变量首先由ip_rt_init根据与可用内存数量和当前jiffies值相关的参数来初始化。当系统已经运行了一段时间后并且有机会建立好的熵时，就使用get_random_bytes函数来重新设置该变量。

Hash表的组织

本节描述的数据结构与L3协议中的结构稍有不同。在IPv4中，hash表bucket的类型是rt_hash_bucket，该结构只包含一个指向冲突元素链表的指针和一个锁，锁的使用在“缓存锁”一节中描述。

缓存中元素的类型为rtable。这个结构包含一些与具体协议有关的字段（参见第三十六章“rtable结构”一节中的描述）和一个与协议无关的、类型为dst_entry的数据结构，如图33-1所示。dst_entry结构包含了与邻居层和邻居缓存的接口、协议变换（transformer，比如IPsec）及路由缓存管理。该数据结构的详细描述可参考第三十六章“dst_entry结构”一节，第二十七章中覆盖了与邻居层的接口。

rtable结构的第一个字段是一个联合，这使得rtable和dst_entry结构很容易共享一些数据，例如，指向下一个冲突的hash表项的指针。虽然指针的名称不同（dst_entry中的指针名为next，而rtable中的指针名是rt_next），但它们所指向的内存位置是相同的。

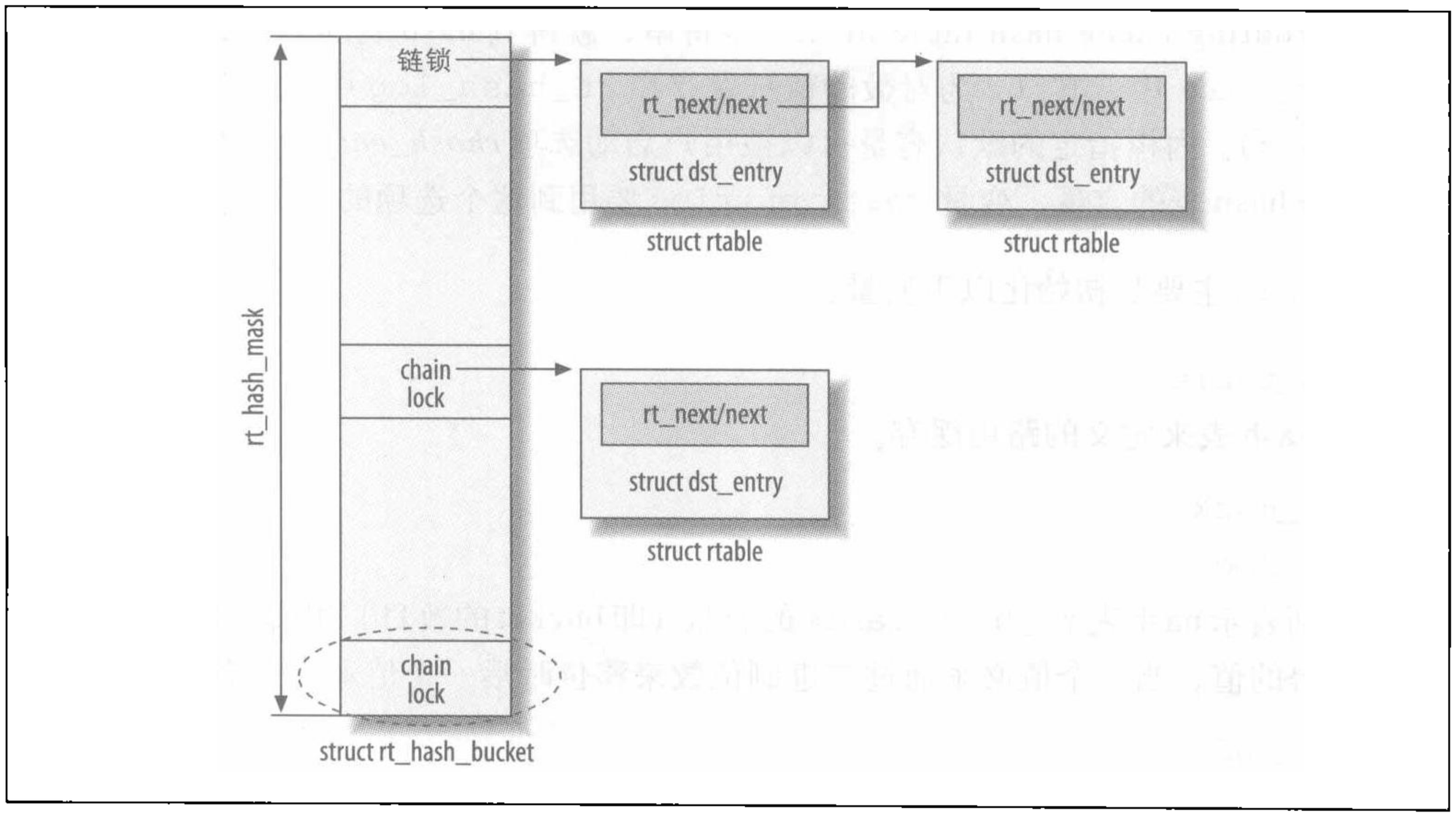


图 33-1：路由缓存结构

```
struct rtable
{
        union
        {
                struct dst_entry dst;
                struct rtable *rt_next;
        } u;
        ......
}
```

一个指向rtable结构的指针可以通过类型转换安全地转换为一个指向dst_entry的指针，反之亦然。

当对缓存表进行插入、删除或查找等操作时，路由子系统根据源IP地址、目的IP地址、TOS字段、入口设备或出口设备的组合来选择缓存表中的一个bucket。在为入口流量选择路由时使用入口设备ID，对本地生成的出口流量选择路由时使用出口设备ID。虽然对入口流量总是能够知道对应的入口设备，但对于出口流量可能还不知道相应的出口设备。除非路由查找关键字中已经包含了出口设备（本地生成的流量可能就属于这种情况，但这不是必然的），否则只有在路由查找后才能知道出口设备。

主要的缓存操作

缓存中与协议无关的（DST）部分是一组dst_entry数据结构。本章中讲述的大部分行

为都会涉及 dst_entry 结构。IPv4 和 IPv6 的数据结构 rtable 与 rt6_info 中都包含一个 dst_entry 数据结构。

dst_entry结构在名为dst_ops的字段中提供了一组虚函数，这些函数能使更高层协议调用具体协议中操作缓存表项的函数。DST 代码位于 *net/core/dst.c* 文件和 *include/net/dst.h* 文件中。

操作 dst_entry 结构的所有函数名都是以 dst_ 为前缀。注意它们操作的虽然是 dst_entry 结构，但它们实际也会影响外部的 rtable 结构。

在系统启动时，由 net_dev_init 调用 dst_init 完成 DST 的初始化（参见第五章）。

缓存锁

像查找等只读操作使用的锁机制与像插入和删除之类的读写操作使用的锁机制不同，但它们必须协同工作。下面给出如何来使用这些锁：

只读操作

在“缓存查找”一节中给出了使用只读操作的函数，该操作由一个读—复制－更新（read-copy-update，RCU）读锁来保护，如下面的代码片断所示：

```
rcu_read_lock();
...
perform lookup
...
rcu_read_unlock();
```

这个代码实际上没有锁定，因为读操作可以互不干扰地同时进行。

读写操作

插入一个缓存表项（参见“添加元素到缓存中”一节）和删除一个缓存表项（参见“删除 DST 表项”一节）都用嵌在每个 bucket 的元素内的旋转锁（spin lock），如图 33-1 所示。注意每个 bucket 都有自己的一个旋转锁，这样就可以使不同的处理器同时写不同的 bucket。

第一章解释了在路由表缓存中实现锁功能的 RCU 算法，以及读—写旋转锁如何与 RCU 共存。

缓存表项的内存分配和引用计数

在系统启动时，ip_rt_init 会创建一个用于分配新缓存表项的内存池。缓存表项的分配使用 dst_alloc 函数，它返回一个 void 型指针，该指针可以被创建者映射为正确的

数据类型。从该函数名看好像分配的是dst_entry结构，但实际并非如此，它实际分配的是包含dst_entry结构的更大的表项：对IPv4分配的是rtable结构（如图33-1所示），对IPv6分配的是rt6_info结构。这个函数可以为不同的协议分配不同大小的数据结构，所分配结构的大小通过一个entry_size虚函数来表示，这个虚函数在“DST与调用协议间的接口”一节中描述过。

添加元素到缓存中

每次为一个入口封包或出口封包选择路由时，如果缓存查找失败，则内核就查找路由表并将结果保存到路由缓存内。内核利用dst_alloc分配一个新的缓存表项，根据路由表查找结果来初始化该表项的一些字段，最后调用rt_intern_hash将这个新表项插入到缓存表的bucket链表的首部。当接收到一个ICMP REDIRECT消息时，一个新的路由表项也会被添加到缓存内（参见第二十五章）。图33-2（a）和33-2（b）给出了rt_intern_hash函数的内部逻辑。如果内核编译时选择支持多路径缓存，那么一次缓存查找失败可能导致多个路由表项被添加到缓存内，这在“多路径缓存”一节中描述。

rt_intern_hash函数首先要通过一次简单的缓存查找来检查新路由表项是否已经存在。虽然该函数是在缓存查找失败后才被调用，但该路由项可能同时被另一个CPU添加到缓存内。如果这个简单查找成功，则查找到的已有的缓存路由表项被移动到bucket链表的首部（这里假定该路由与多路径路由没有关联，即它没有设置DST_BALANCED标识）；如果查找失败，则将这个新路由项添加到缓存内。

rt_intern_hash在每次添加一个新缓存表项时都试图删除一个缓存表项，这种简单的方式可以控制缓存容量。当遍历bucket链表时，rt_intern_hash会跟踪那些最符合删除条件的路由项，并计算bucket链表的长度。一个路由项只有在符合被删除条件（即其引用计数为0）并且bucket链表的长度超过配置参数ip_rt_gc_elasticity时，才被删除。如果这些条件满足，rt_intern_hash调用rt_score函数来选择应被删除的最佳路由项。rt_score又根据许多标准将这些符合的路由项分为如下三类，按从最有价值（最不适合被删除）路由到价值最少（最适合被删除）路由的顺序（注1）：

- 通过ICMP重定向插入的且正被用户空间命令监控的路由项，或者是由于过期将被调度的路由项。
- 出口路由（用于路由本地生成的封包的路由项）、广播路由、多播路由、到本地地址（主机自己生成的封包目的地是本地）路由。

注1： 参见第三十章“合适的缓存牺牲品范例”一节。

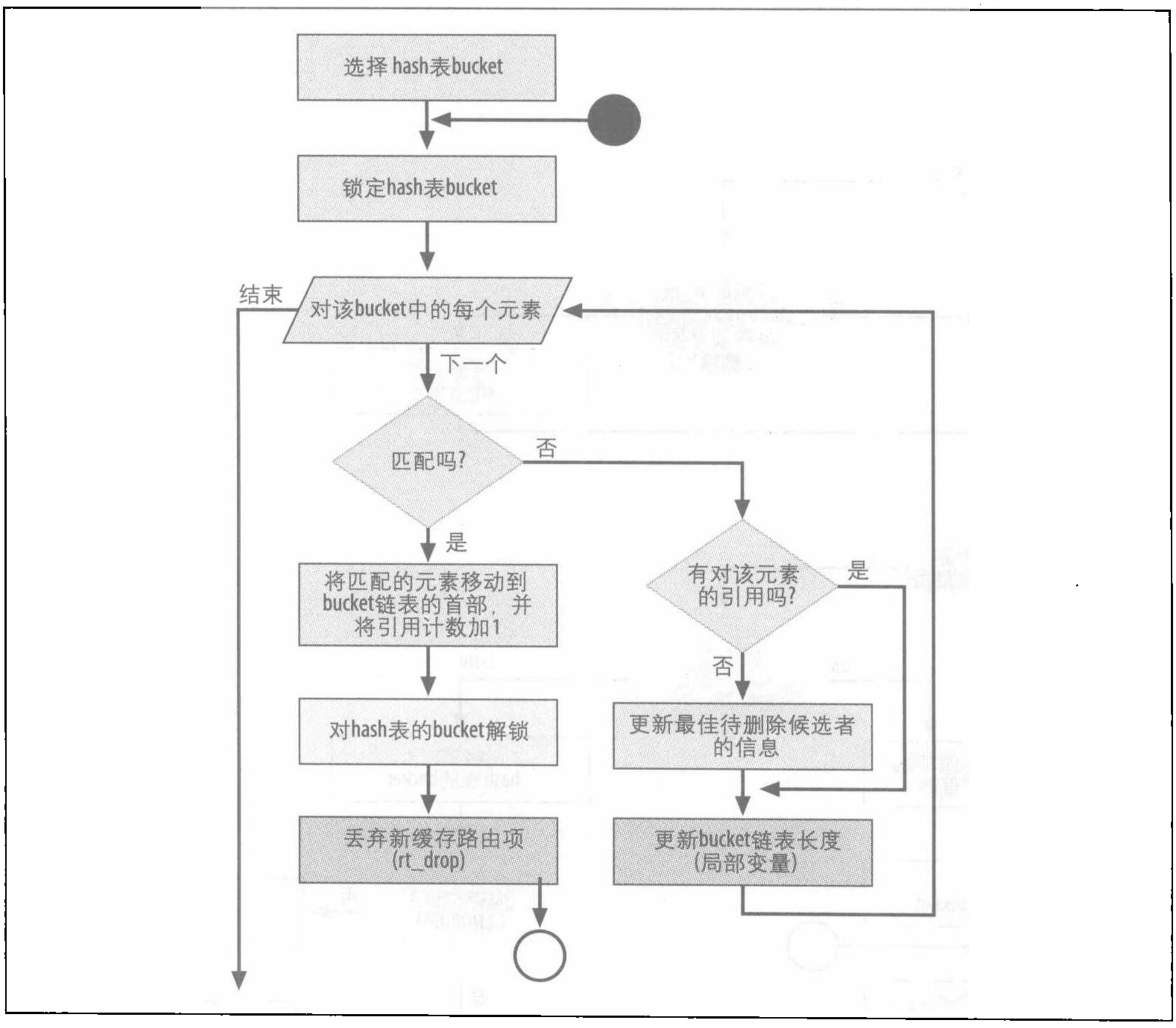


图 33-2(a)：rt_intern_hash 函数

- 按照最后一次使用的时间戳降序排列的所有其他路由项：即最近最少被使用的路由项先被删除。

rt_score将路由表项最近没有被使用的时间保存在一个32位变量的低30位中，并对第一类路由项设置第31位，对第二类路由项设置第32位。最终的值就是表示该路由项重要性的分数：得分越低的路由项，越有可能被rt_intern_hash选择为要删除的受害者。

将路由缓存绑定到 ARP 缓存

大多数路由缓存表项要被绑定到该路由下一跳地址对应的ARP缓存表项。这意味着一条路由缓存表项需要一个已有的 ARP 缓存表项，或者对同一个下一跳地址在进行 ARP 查找时能够成功。对于本地生成的封包（其入口设备 ID 为 NULL）的输出路由和单播转发

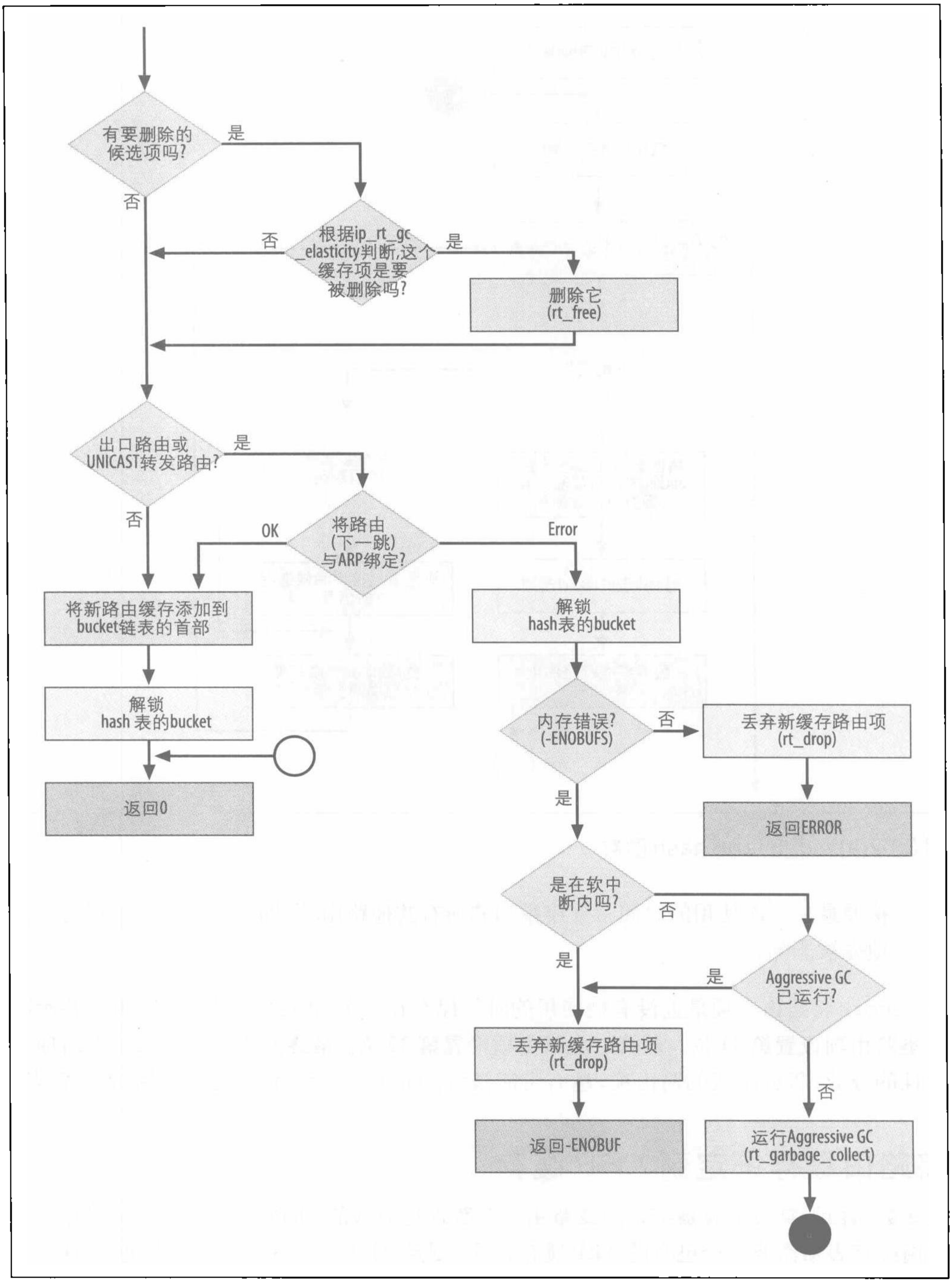


图 33-2(b)：rt_intern_hash 函数

路由尤其需要进行绑定。在这两种情况下，需要ARP来解析下一跳的L2地址。而转发目的地为广播地址、多播地址和本机地址则不需要ARP解析，因为使用其他方法可以得到这个地址。

目的地为广播地址和多播地址的出口路由不需要相关联的ARP表项，因为其相关的L2地址可通过L3地址导出（参见第二十六章“特殊情况”一节）。到本地地址的路由也不需要ARP解析，因为与这类路由匹配的封包被提交到本地。

使用arp_bind_neighbour来为路由创建与ARP的绑定。当该函数由于内存不足而失败时，rt_intern_hash通过调用rt_garbage_collect来强行对路由缓存执行一次主动的垃圾回收操作（参见“垃圾回收”一节）。主动垃圾回收是通过降低阈值ip_rt_gc_elasticity和ip_rt_gc_min_interval，然后调用rt_garbage_collect来完成的。这种垃圾回收只做一次，并且只有当rt_intern_hash的调用不是在软中断上下文中时才进行，否则将耗费大量的CPU时间。一旦完成了垃圾回收，就重新从缓存查找开始来插入新的缓存表项。

缓存查找

无论何时要找到一条路由，内核首先查找路由缓存，如果缓存没有命中则查找路由表。路由表查找过程在第三十五章描述，本章只讨论缓存查找。

路由子系统提供了两个不同路由查找函数，一个用于入口路由，另一个用于出口路由：

ip_route_input

用于入口流量的路由查找，这些流量可能是本地提交或要被转发。这个函数决定如何来处理常规封包（本地提交、还是转发、丢弃等），但它也被其他子系统用于确定如何来处理它们的入口流量。例如，ARP使用该函数来判断是否应当对一个ARPOP_REQUEST进行应答（参见第二十八章）。

ip_route_output_key

用于出口流量的路由查找，这些流量是由本地生成，可能被本地提交或被发送出去。

这两个函数的返回值包括：

0

表示路由查找成功。这种情况也包括当缓存查找失败而触发路由表查找，并且成功。

-ENOBUF

由于内存问题而导致查找失败。

-ENODEV

查找关键字包含了一个无效的设备 ID。

-EINVAL

常规查找失败。

为了用于特殊情况，内核针为这两个基本函数提供了一组包裹函数。例如，可以参见TCP如何使用 ip_route_connect 和 ip_route_newports。

图 33-3 给出了这两个主要的路由缓存查找函数的内部流程。图中给出的出口函数是 __ip_route_output_key，它被 ip_route_output_key 间接调用。

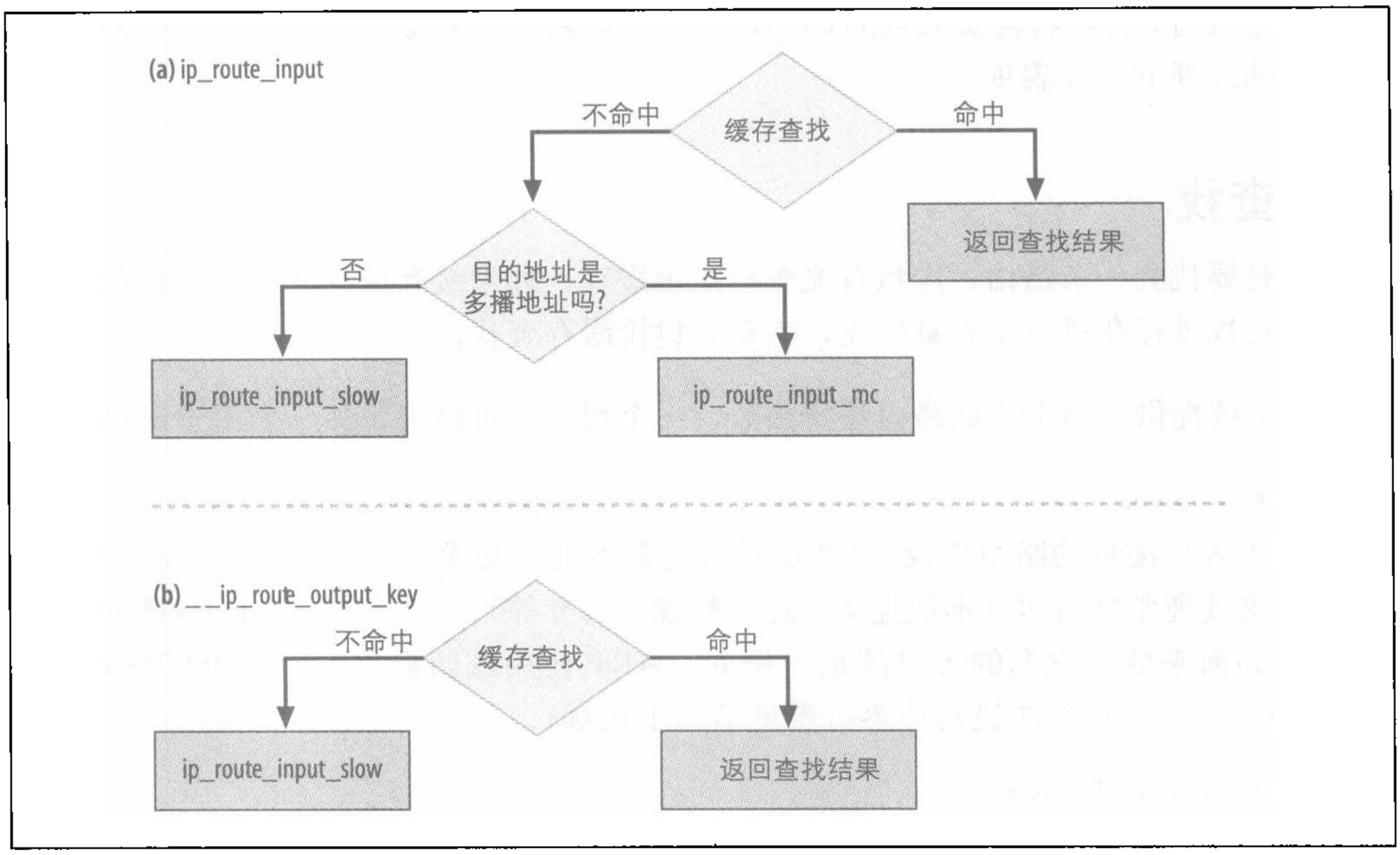


图 33-3：(a) ip_route_input_key 函数； (b) __ip_route_output_key 函数

路由缓存对入口路由和出口路由都进行保存，所以在缓存查找包含这两种情况。如果缓存查找失败，查找函数就调用 ip_route_input_slow或ip_route_output_slow，它们通过 fib_lookup 函数（我们将在第三十五章中介绍该函数）来查找路由表。函数名以_slow结尾是为了强调缓存查找和路由表查找在速度上的差异，这两种查找途径也被称为快速途径（fast）和慢速途径（slow）。

一旦决定进行路由，那么无论是通过缓存查找命中，还是通过查找路由表成功或失败，查找函数都会返回输入的缓冲区skb，但其中skb->dst->input和skb->dst->output虚函数已经被初始化，skb→dst为满足路由请求的缓存表项；当缓存查找失败时，一个新的缓存表项被创建并链接到skb→dst。

接着，调用skb->dst->input（通过一个名为dst_input的简单的包裹函数调用）和skb->dst->output（通过一个名为dst_output的包装函数调用）这两个虚函数中的一个或全部，来对封包进一步处理。第十八章中图18-1给出了这两个虚函数在IP协议栈中被调用的位置，以及根据流量方向被初始化为什么函数。

第三十五章会详细讨论路由表查找使用的慢速函数。接下来的两个小节将描述图33-3中的两个缓存查找函数的内部细节。它们的代码很相似，仅有的区别是：

- 对入口路由，入口路由项的设备需要与入口设备匹配，而出口设备还不知道，因此简单与空设备（0）比较。相反的条件应用于出口路由。
- 当缓存查找命中时，函数使用RT_CACHE_STAT_INC宏分别更新in_hit和out_hit计数器。第三十六章中描述路由缓存和路由表相关的统计数据。
- 出口查找需要考虑RTO_ONLINK标识（参见“出口查找”一节）。
- 出口查找支持多路径缓存，该特性在第三十一章“支持多路径缓存”一节中描述。

入口查找

ip_route_input用于路由入口封包。下面给出函数原型和其输入参数的含义：

```
int ip_route_input(struct sk_buff *skb, u32 daddr, u32 saddr,
                   u8 tos, struct net_device *dev)
```

skb

触发路由查找的封包。这个封包本身可能不需要被路由。例如，ARP出于其他原因使用ip_route_input来查询本地路由表，这时的skb可能是一个入口ARP请求。

saddr

daddr

用于查找的源地址和目的地址。

tos

IP包头中的TOS字段。

dev

接收该封包的设备。

ip_route_input首先根据输入条件从路由hash 表中选择应当包含该路由的bucket，然后挨个遍历bucket链表中的路由项，比较所有必要的字段，直到查找到匹配项或到链表尾部时还没有找到匹配项。

作为ip_route_input函数输入参数传递进来的查找字段与路由缓存表项rtable中存储的fl字段（注2）相比较，如下面的代码片段所示。通过输入参数的组合来选择合适的bucket（hash变量），路由缓存表项用rth变量来表示。

```
hash = rt_hash_code(daddr, saddr ^ (iif << 5), tos);
rcu_read_lock();
for (rth = rcu_dereference(rt_hash_table[hash].chain; rth;
     rth = rcu_dereference(rth->u.rt_next)) {
    if (rth->fl.fl4_dst == daddr &&
            rth->fl.fl4_src == saddr &&
            rth->fl.iif == iif &&
            rth->fl.oif == 0 &&
#ifdef CONFIG_IP_ROUTE_FWMARK
            rth->fl.fl4_fwmark == skb->nfmark &&
#endif
            rth->fl.fl4_tos == tos) {
            rth->u.dst.lastuse = jiffies;
            dst_hold(&rth->u.dst);
            rth->u.dst._ _use++;
            RT_CACHE_STAT_INC(in_hit);
            rcu_read_unlock();
            skb->dst = (struct dst_entry*)rth;
            return 0;
        }
        RT_CACHE_STAT_INC(in_hlist_search);
    }
    rcu_read_unlock();
```

当缓存查找不命中是由于封包的地址是多播地址时，那么以下两个条件只要有一个满足，该封包就被送给多播处理函数ip_route_input_mc，否则就被丢弃：

- 目的地址是本地配置的多播地址，这时通过ip_check_mc来检查。
- 目的地址不是本地配置的多播地址，但内核支持多播路由（CONFIG_IP_MROUTE）。

下面代码给出了这个判断过程：

```
if (MULTICAST(daddr)) {
    struct in_device *in_dev;
    rcu_read_lock();
    if ((in_dev = _ _in_dev_get(dev)) != NULL) {
        int our = ip_check_mc(in_dev, daddr, saddr,
                              skb→nh.iph→protocol);
```

注2： flowi结构在第三十二章“主要的数据结构”小节中描述。

```
            if (our
#ifdef CONFIG_IP_MROUTE
                || (!LOCAL_MCAST(daddr) && IN_DEV_MFORWARD(in_dev))
#endif
                ) {
                rcu_read_unlock();
                return ip_route_input_mc(skb, daddr, saddr,
                                tos, dev, our);
            }
        }
        rcu_read_unlock();
        return -EINVAL;
    }
```

最后，如果在目的地址不是多播情况下缓存查找不命中时，ip_route_input 调用 ip_route_input_slow 来查找路由表：

```
    return ip_route_input_slow(skb, daddr, saddr, tos, dev);
}
```

出口查找

__ip_route_output_key 用于路由本地生成的封包，这与 ip_route_input 很相似：首先检查缓存，在缓存查找失败情况下调用 ip_route_output_slow。当缓存支持多路径时，缓存命中需要一些更多的工作：在缓存内可能选择出多个符合条件的路由缓存表项，然后根据所使用的缓存算法选出一个正确的表项。选择过程是由 multipath_ select_ route 函数来实现，更多的细节可参见“多路径缓存”一节。

下面是该函数原型和其输入参数的含义：

```
int _ _ip_route_output_key(struct rtable **rp, const struct flowi *flp)
```

rp

当函数返回成功时，*rp 指向与搜索关键字 flp 相匹配的缓存表项。

flp

搜索关键字。

出口缓存查找成功需要匹配 RTO_ONLINK 标识，检查该标识是否设置：

```
!((rth->fl.fl4.tos ^ flp->fl4_tos) &
    (IPTOS_RT_MASK | RTO_ONLINK))
```

上面的条件只有在以下两个条件都满足时才为真：

- 路由缓存表项的TOS与搜索关键字中的TOS匹配。注意这个TOS字段占用8位tos变量的2、3、4、5位（参见第十八章中的图18-3）（注3）。
- 当路由缓存表项和搜索关键字都设置了RTO_ONLINK标识，或者都没有设置。

在第三十五章“搜索关键字初始化”一节中，你将看到RTO_ONLINK 标识。该标识通过TOS变量被传递，但该标识与IP包头中的TOS字段无关，它只用TOS字段的一个无用的位（参见第十八章的图18-1）。当该标识被设置时，表示目的地位于本地子网所以无需路由查找（或者换句话说，路由查找可能失败，但这并不是一个问题）。该标识不是管理员在配置路由时所设置，但在路由查找时要使用该标识来指定搜索的路由类型的scope必须为RT_SCOPE_LINK，这表示目的地是直接相连。然后，在创建相关的路由缓存表项时将该标识保存在这个缓存项中。查找时的RTO_ONLINK标识可通过下面协议来设置：

ARP

当管理员手工配置一个ARP映射时，内核确保该IP地址是属于本地配置子网。例如，命令 *arp -s 10.0.0.1 11:22:33:44:55:66* 将10.0.0.1到11:22:33:44:55:66的映射添加到ARP缓存内。如果内核根据它的路由表判断，IP地址10.0.0.1不属于本地配置子网内的地址，那么内核就拒绝执行该命令（参见arp_req_set和第二十六章）。

Raw IP 和 UDP

当通过一个套接字发送数据时，用户可以设置MSG_DONTROUTE标识。当应用程序从一个已知接口向直连的目的地（不需要网关）发送封包时使用该标识，这样内核就不需要确定出口设备。在路由协议和诊断性应用程序中使用这种发送方式。

多路径缓存

在第三十一章“支持多路径缓存”一节中介绍了多路径缓存背后的概念。如果内核在编译时支持多路径缓存，查找代码会将多条路由添加到缓存内，见第三十五章“多路径缓存”一节中的描述。在本节，我们来讨论实现该特性的关键函数，以及缓存算法提供的接口。

注3：在第十八章图18-3中所示的TOS字段占用8个二进制位，忽略第0位，使用1～7位。但路由程序只用1、2、3、4位。对出口路由不考虑TOS字段的高位部分（5、6、7位），通过RT_TOS宏将这些位屏蔽掉。

注册一个缓存算法

缓存算法被定义为ip_mp_alg_ops数据结构的一个实例，该结构由一些函数指针组成。根据缓存算法的需要，这些函数指针并不是全都要被初始化，但其中的mp_alg_select_route 函数指针是必须要初始化的。

向内核注册和注销缓存算法分别使用multipath_alg_register和multipath_alg_unregister。所有的缓存算法都是实现为模块，在*net/ipv4/*目录中。

路由缓存与多路径之间的接口

对ip_mp_alg_ops 数据结构的每一个函数指针，内核都定义了一个包裹函数(warpper)，它位于*include/net/ip_mp_alg.h*文件中。下面分别介绍什么时候会调用这些函数：

multipath_select_route

这是最重要的函数，它从缓存内满足一个查找条件的多个路由项(因为查询条件与同一个多路径路由相关）中选择出正确的一项。这个函数被我们在前面看到的__ip_route_output_key查找函数调用。

multipath_flush

当缓存被刷新时清空缓存算法保持的任何状态。它由rt_cache_flush调用(参见“刷新路由缓存”一节)。

multipath_set_nhinfo

当一个新的多路径路由被缓存时，更新缓存算法保持的状态信息。

multipath_remove

当一个多路径路由被删除时(例如，通过rt_free函数)，删除缓存内的正确路由。

并不是所有的缓存算法都支持multipath_remove，只有加权随机算法使用multipath_flush和multipath_set_nhinfo。

在后面的几节中，我们将看到各种缓存算法需要保存的状态信息，以及它们是如何实现mp_alg_select_route函数的。

辅助函数

下面有两个多路径代码所使用的函数：

multipath_comparekeys

对两个路由选择关键字进行比较。它主要被mp_alg_select_route算法中的函数使用，来查找和另一条缓存路由项都与一个多路径路由关联的其他缓存路由项。

rt_remove_balanced_routes

给定一个输入的缓存路由，该函数可以删除这条路由，并删除与该路由链接在一个bucket中的与同一个多路径路由关联的所有其他的缓存路由项。rt_remove_balanced_routes 的最后一个输入参数返回被删除的缓存路由项数目。这个函数的返回值为bucket链表中由输入路由项所指向的下一个rtable实例。这个函数的调用方使用该返回值，从适当的位置继续开始扫描缓存表。当rt_remove_balanced_routes 删除了bucket链表中最后一个相关的rtable实例后，就返回NULL。

几种路由算法之间的共性

记住以下三个要点将有助于你理解多路径缓存代码，尤其是缓存算法提供的mp_alg_select_route 函数的实现：

- 与多路径路由相关的路由缓存表项的识别需要归功于DST_BALANCED标识，该标识是在这些路由插入到缓存前被设置（参见第三十六章“dst_entry结构”一节）。我们将在第三十五章看到如何及何时来设置该标识。该标识通常是在路由缓存代码中，根据缓存表项是否与一条多路径路由关联来采取不同的动作。
- dst_entry结构被用于定义缓存路由项，其中包含一个上次使用的时间戳（dst→lastuse）。每当缓存查找返回一个缓存路由项时，更新该缓存路由项的时间戳。与多路径路由相关的缓存项需要特殊处理。当缓存查找返回与一个多路径路由相关的缓存项时，与该多路径路由相关的所有其他缓存项也必须更新其时间戳，为了避免垃圾回收算法将这些路由缓存项清除，必须这么做。
- 输入mp_alg_select_route函数的路由项是与查找关键字相匹配的第一个缓存项。元素被添加到路由缓存内的方式是：与同一个多路径路由相关的所有其他缓存项都位于同一个bucket内。由于这个原因，mp_alg_select_route将从输入的缓存元素开始遍历bucket链表，根据DST_BALANCED标识和multipath_comparekeys函数，识别出其他的路由缓存项。

随机算法

该算法不需要保存任何状态信息，因此既不需要分配任何内存，也不会占用很多的CPU时间来决策。该算法所做的就是遍历输入路由缓存表的bucket链表，计算符合选择条件

的缓存项数目，再利用本地函数 random 生成一个随机数，根据该随机数来选择相应的缓存项。

该算法的定义在 *net/ipv4/multipath_random.c* 文件中。

加权随机算法

这是实现最复杂的算法。一条多路径路由的每一个下一跳都指定有一个权重。该算法按照权重比例来随机选择正确的下一跳（即缓存内正确的路由项）。

对一条多路径路由的每一个下一跳，都有一个 fib_nh 数据结构实例来存储权重和其他参数。我们将在第三十四章看到这些数据结构在路由表中位于什么地方，可以参考该章中的图 34-1。

在第三十一章“加权随机算法”一节中，解释了该算法背后的基本思想。为了能够尽快决策，该算法构建了一个本地信息数据库，用于访问 fib_nh 实例和读取下一跳的权重。图 33-4 给出了在配置了以下两条多路径路由后，数据库的可能形式：

```
#ip route add 10.0.1.0/24 mpath wrandom nexthop via 192.168.1.1 weight 1
                                        nexthop via 192.168.2.1 weight 2
#ip route add 10.0.2.0/24 mpath wrandom nexthop via 192.168.1.1 weight 5
                                        nexthop via 192.168.2.1 weight 1
```

实际上，该数据库不是在定义多路径路由时立刻创建，而是在路由查找时才创建。

记住 mp_alg_select_route 函数（此时为 wrandom_select_route）的输入是与搜索关键字相匹配的第一个缓存路由项。所有其他匹配的缓存路由项都位于同一个路由缓存 bucket 内。

通过 mp_alg_select_route 来选择路由分为两个步骤：

1. mp_alg_select_route 首先要遍历输入路由项所在的 bucket 链表，对链表中的每一个路由缓存项，用 multipath_comparekeys 函数来检查它是否符合选择条件。同时，mp_alg_select_route 创建一个局部链表保存符合条件的缓存路由项，主要目标是定义一条与第三十一章中的图 31-4 相类似的线。图 33-5 给出了本章所举的例子中链表的可能样子。添加到该链表的每一个路由使用图 33-4 中的数据库得到其权重，并初始化相应的 power 字段。
2. mp_alg_select_route 生成一个随机值，并根据符合条件的路由项的链表，按照第三十一章“加权随机算法”一节中描述的机制从中选择一条路由。

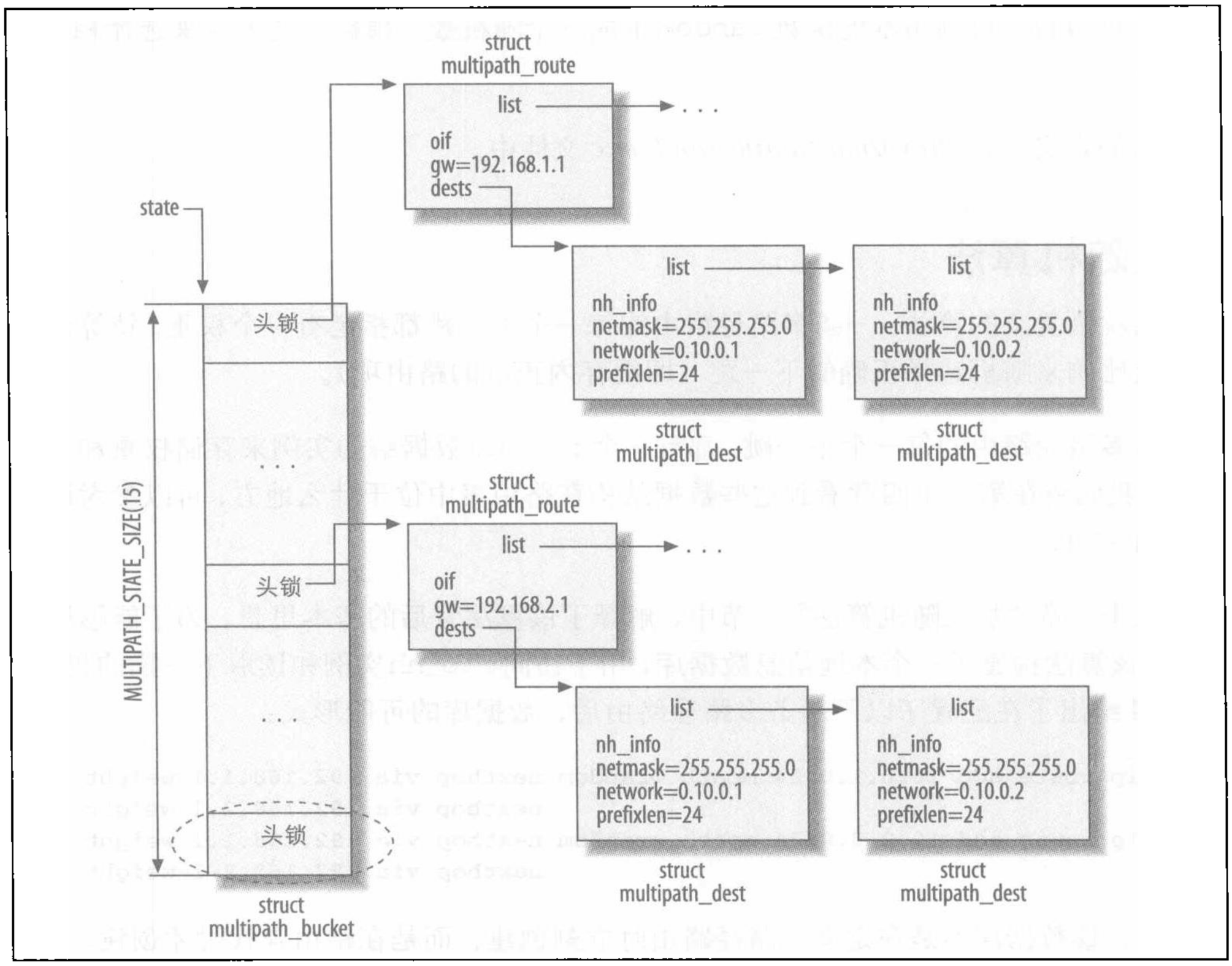


图 33-4：加权随机算法创建的下一跳数据库

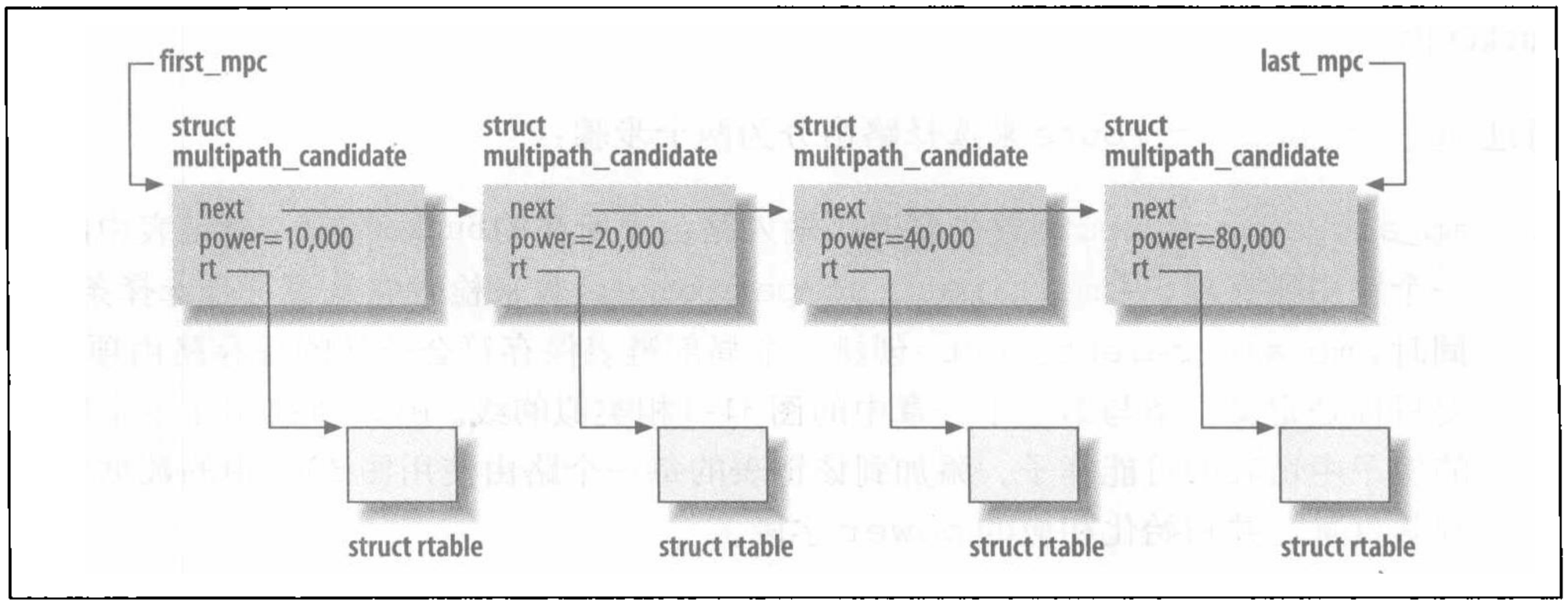


图 33-5：为选择下一跳而创建的临时链表例子

我们来看如何从这个 `state` 数据库来查找。要记住缓存路由项（即 `rtable` 实例）包含

了下一跳路由器和出口设备。给定一条缓存路由，__multipath_lookup_weight首先根据出口设备从state中选择出相应的bucket：state是以设备为索引。一旦选择出state的一个bucket，就开始扫描multipath_route元素链表，查找与网关和设备字段相匹配的项。一旦得到正确的multipath_route实例，就开始扫描相关的multipath_dest结构链表，查找与输入查找关键字fl中的目的IP地址相匹配的表项。函数可以从匹配的multipath_dest实例中，通过nh_info指针来读取下一跳权重，该指针指向相应的 fib_nh 实例。

state数据库是由multipath_set_nhinfo函数来构建的，该函数在“路由缓存与多路径之间的接口”一节中描述。

该算法的定义是在 *net/ipv4/multipath_random.c* 文件中。

循环算法

循环算法不需要额外的数据结构来保存它所需要的状态信息。所需要的全部信息可以从dst_entry结构的dst→__use字段得到，该字段表示缓存查找返回该路由项的次数。因此选择合适的路由项就是对输入的bucket链表中的路由项遍历，从合法路由项中选择__use值最小的项。

该算法的定义是在 *net/ipv4/multipath_rr.c* 文件中。

设备循环算法

设备循环算法的目的和效果在第三十一章“设备循环算法”一节中解释。该算法选择合适的出口设备，因而就是对一个给定的多路径路由得到合适的缓存表项，drr_select_route函数的流程如下：

1. 全局向量state对每个设备保持一个计数器来表示该设备已经被选择的次数。
2. 对每个多路径路由，只考虑给定设备上的第一个下一跳。这加快了决策速度，但也隐含了在共享同一个出口设备的多个下一跳之间不会负载共享：对每一个设备，任何一个多路径路由项中只用一个下一跳。
3. 当遍历路由（也就是下一跳）计算最小使用次数时，那些还没被使用的设备相关的路由项被给予更高的优先级。当选择了一个新设备时，一个新表项被添加到 state 中。
4. 分析得到使用次数值最小的设备，该设备对应的第一条路由被选中。

该算法的定义是在 *net/ipv4/multipath_drr.c* 文件中。

DST与调用协议间的接口

DST缓存是一个独立的子系统，例如，它有自己的垃圾回收机制。作为一个子系统，它提供了一组函数，各种协议可以使用这些函数来改变或调整其行为。当外部子系统需要与路由缓存交互时，例如，通知它一个事件或是读取某一个参数值时，是通过定义在*net/core/dst.c*和*include/net/dst.h*文件中的一组DST函数来进行的。这些DST函数被包裹为另外一组函数，拥有自己的路由缓存的L3协议通过初始化一个dst_ops VFT实例可以使用这组函数，如图33-6所示。

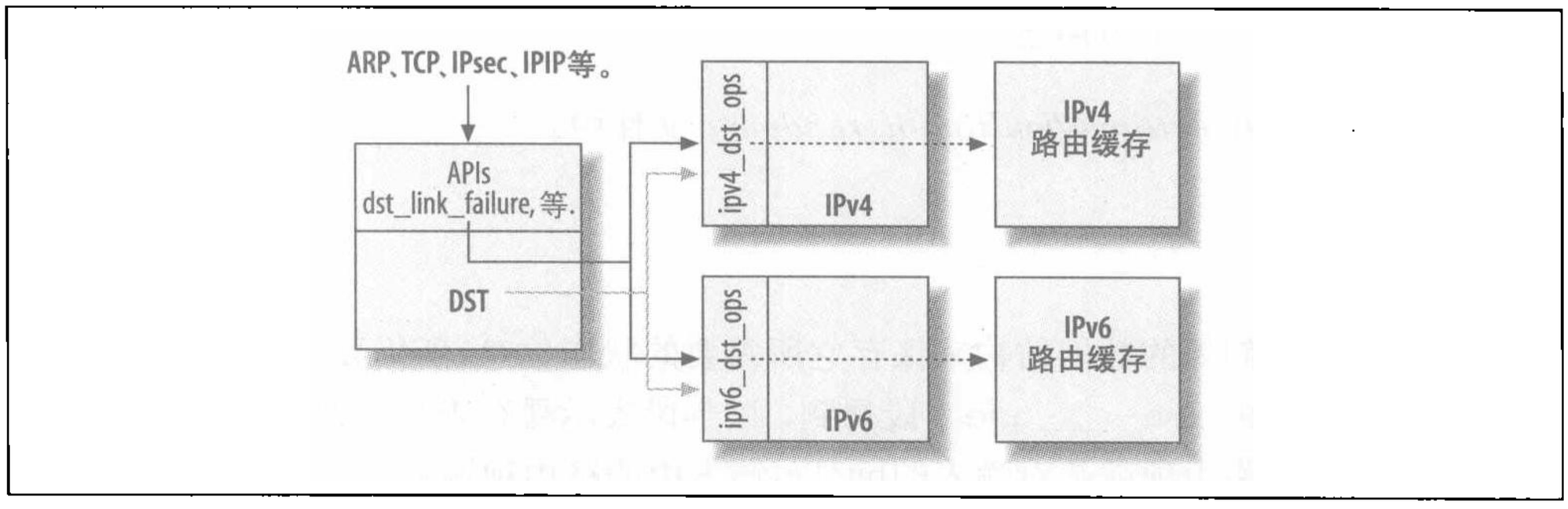


图33-6：dst_ops接口

DST向更高层提供的主要数据结构是dst_entry，与协议相关的结构，比如rtable等，只是对这个结构的包裹。IP协议有自己的路由缓存，但其他协议通常保持对这些路由缓存元素的引用。所有这些引用都涉及dst_entry，而不是协议自己的包裹结构rtable。sk_buff缓冲区也保持对dst_entry结构的一个引用，而不是对rtable结构的引用，这个引用用于存储路由查找结果。

dst_entry和dst_ops结构在第三十六章相关节中有详细描述。针对每个协议有一个dst_ops实例，例如，IPv4使用的实例为ipv4_dst_ops，它在*net/ipv4/route.c*文件中被初始化：

```
struct dst_ops ipv4_dst_ops = {
    .family =           AF_INET,
    .protocol =         _ _constant_htons(ETH_P_IP),
    .gc =               rt_garbage_collect,
    .check =            ipv4_dst_check,
    .destroy =          ipv4_dst_destroy,
    .ifdown =           ipv4_dst_ifdown,
    .negative_advice =  ipv4_negative_advice,
    .link_failure =     ipv4_link_failure,
    .update_pmtu =      ip_rt_update_pmtu,
    .entry_size =       sizeof(struct rtable),
};
```

只要DST子系统收到一个事件通知，或通过其中一个DST接口函数发起请求，与受影响的dst_entry实例相关的协议就得到通知，通过其dst_ops VFT实例来调用由dst_entry提供的适当函数。例如，如果ARP要向上层协议通知一个已有的IPv4地址不可到达，则对相关的dst_entry结构调用dst_link_failure（记住缓存路由项是与IP地址相关联，而不是与网络相关联），这将调用由IPv4通过ipv4_dst_ops注册的ipv4_link_failure函数。

调用协议也可能直接干预DST的行为。例如，当IPv4请求DST分配一个新的缓存表项时，DST可能意识到需要开始垃圾回收并调用IPv4自己提供的rt_garbage_collect函数。

当对某个给定的通知类型要求所有的协议都对其进行相同的处理时，这个相同逻辑可能直接在DST API中实现，而不是在每个协议的处理函数中都重复一次。

通过更高层的包裹函数来调用DST的dst_ops结构中的一些虚函数；没有被包裹的函数通过dst->ops->function语法来直接调用。这里给出dst_ops虚函数的含义，并简要描述IPv4子系统中给这些虚函数赋值的函数（在前面代码片段中列出）：

gc

负责垃圾回收。当子系统通过dst_alloc分配一个新的缓存表项，且该函数发现内存不够时就进行垃圾回收。IPv4的函数rt_garbage_collect在“同步清理”一节中描述。

check

dst_entry被标记为dead的缓存路由项通常不再被使用，但当使用IPsec时该结论并不一定成立。该函数检查一个废弃的dst_entry是否还有用。例如，可以参考ipv4_dst_check函数，它在删除dst_entry结构之前并不对其检查；将ipv4_dst_check函数与IPsec中做“xfrm”变换的相应函数xfrm_dst_check比较一下。也可以参考诸如sk_dst_check（在第二十一章介绍）等函数如何检查缓存路由项的状态。没有针对该函数的包裹函数。

destroy

该函数被dst_destroy调用，DST用dst_destroy函数来删除一个dst_entry结构，并将删除通知调用协议，以便调用协议先做一些必要的清理工作。例如，IPv4函数ipv4_dst_destroy使用该通知来释放对其他数据结构的引用。dst_destroy在“删除DST表项”一节中描述。

ifdown

该函数被dst_ifdown 调用，当一个设备被关闭或注销时，DST子系统自己会调用dst_ifdown。对每一个受影响的缓存路由项都要调用该函数一次（参见“外部

事件”一节)。IPv4 函数 ipv4_dst_ifdown 用一个指向回环设备的指针来替换 rtable 中指向设备 IP 配置的 idev 指针，这是因为回环设备总是存在的。

negative_advice

该函数被DST函数dst_negative_advice调用，dst_negative_advice用于向DST 通知某个 dst_entry 实例出现问题。例如，当 TCP 检测到一次写操作超时，就使用 dst_negative_advice。

IPv4 函数 ipv4_negative_advice 使用该通知来删除缓存路由项。当这个 dst_entry 已经被标记为 dead（通过其 dst → obsolete 标识，我们将在“删除DST表项”一节中看到），ipv4_negative_advice就释放rtable对该dst_entry的引用。

link_failure

该函数被DST 函数dst_link_failure调用，由于目的地不可到达而出现封包传输问题时，就调用 dst_link_failure。

下面举例描述该函数的使用。在 IPv4 和 IPv6 中使用的邻居协议分别是 ARP 和 ND，当激活该函数时表示它们从来没有接收到它们生成的 solicitation 请求的应答，该请求被用于解析L3 地址到L2 地址的关联（这通常是由于超时，例如，可以参看 *net/ipv4/arp.c* 文件中处理 ARP 协议行为的 arp_error_report）。对其他高层协议，诸如各种隧道（IP over IP 等）协议，当到达隧道另一端出现问题时做相同的处理。注意距离隧道另一端可能有多跳。例如，可以参看在 *net/ipv4/ipip.c* 文件中用于 IP-over-IP 隧道协议的 ipip_tunnel_xmit。

update_pmtu

更新缓存路由项的 PMTU。通常是在处理所接收到的 ICMP 分片需求消息时调用，参见第三十一章“处理入口 ICMP_REDIRECT 消息”一节。没有针对该函数的包裹函数。

get_mss

返回该路由使用的TCP最大段（MSS）。IPv4不初始化该函数，所以没有针对该函数的包裹函数。参见“IPsec 变换与 dst_entry 的使用”一节。

除了上面列出的这些虚函数的包裹函数以外，DST也通过一些不与其他子系统交互的函数来操作 dst_entry 实例。例如，“异步清理”一节中给出的dst_set_expires，第二十六章给出的如何使用dst_confirm来确认一个邻居是否可达。更多的细节可参见 *net/core/dst.c* 文件和 *include/net/dst.h* 文件。

IPsec 变换与 dst_entry 的使用

在前面的一节中，我们看到dst_entry结构的常见用途：对一条缓存路由存储与协议无关的信息，包括处理接收到的封包的input方法和在路由查找之后发送封包的output方法。

dst_entry 结构的另一个用途是用于 IPsec。IPsec 是一组可以在 IP 层之上提供诸如认证和加密等安全服务的协议。IPsec 使用 dst_entry 结构来构造它所称的变换束（transformation bundles）。一个变换（transformation）是针对一个封包的一个操作，例如，加密。一个束（bundle）就是被定义为一连串操作的一组变换。一旦 IPsec 协议决定将所有的变换应用在和一个给定路由相匹配的流量上，那么在路由缓存中存储的信息是一个 dst_entry 结构链表。

一般情况下，一个路由项与一个dst_entry结构相关联，该结构的input和output字段描述了如何来处理匹配封包（转发、本地提交等，参见第十八章中的图18-1）。但IPsec创建一个dst_entry实例链表，只有最后一个实例中的input和output方法被实际应用于路由决策；前面实例中的 input 和 output 方法被应用于所要求的变换，见图 33-7 所示（图中给出的是一个简化模型）。

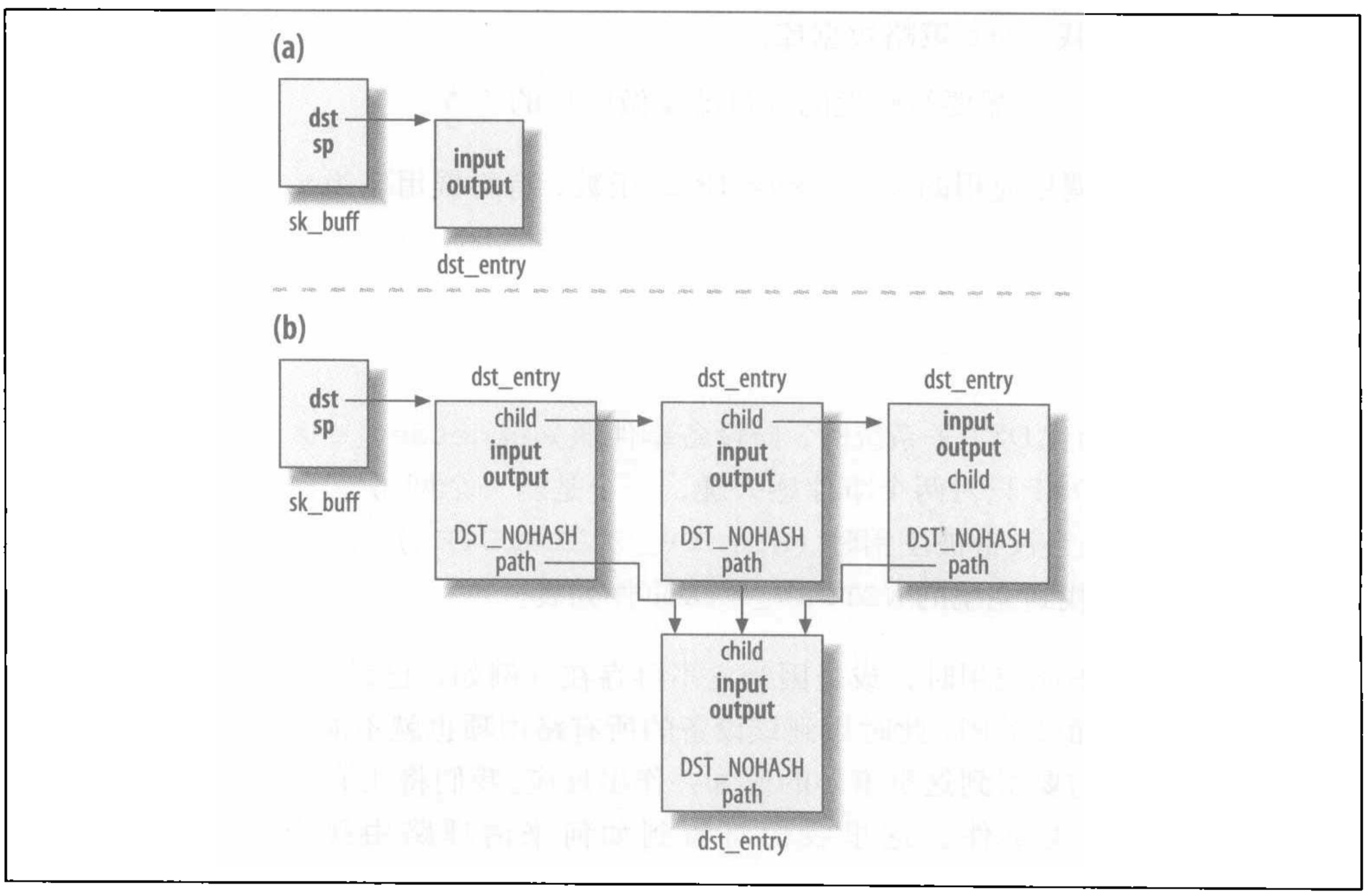


图 33-7：dst_entry 的使用 (a) 没有 IPsec；(b) 有 IPsec

在dst_entry结构中使用child指针来构造dst_entry链表。IPsec还使用另一个指针path，它指向链表的最后一个元素（即使不使用IPsec也将创建该元素）。

在链表中的其他dst_entry元素，即除了最后一个元素以外的每一个元素都要实现一个IPsec变换。每个元素设置其path字段指向最后一个元素。而且，每个元素都设置其DST_NOHASH标识，以使DST子系统知道该元素不在路由缓存hash表内，而由另一个子系统来处理它。

IPsec对路由查找的影响如下：输入和输出路由查找受到该数据结构布局的影响，见图33-7（b）中IPsec配置示意图。查找返回结果是一个指针，它指向第一个执行变换的dst_entry，而不是最后一个表示实际路由信息的dst_entry。这是因为第一个dst_entry实例表示第一个要执行的变换，且必须按照顺序来执行变换。

你可以在好几个地方找到IP或路由层与IPsec之间的交互：

- 对出口流量，ip_route_output_flow函数（被在“缓存查找”一节中介绍的ip_route_output_key调用）内包含额外的代码（即调用xfrm_lookup）来与IPsec交互。
- 对提交到本地的入口流量，ip_local_deliver_finish调用xfrm4_policy_check来查找IPsec策略数据库。
- ip_forward 对需要被转发的入口流量做同样的检查。

IP代码有时直接调用通用的xfrm_*xxx* IPsec函数，有时调用名为xfrm4_*xxx*的IPv4包裹函数。

外部事件

当dst_init初始化DST子系统时，向设备事件通知链netdev_chain注册，该通知链在第四章介绍。DST只对两个事件感兴趣，一个是当一个网络设备关闭时（NETDEV_DOWN），另一个是当设备被注销时（NETDEV_UNREGISTER）。你可以在*include/linux/notifier.h*文件中找到完整的NETDEV_*XXX*事件列表。

当一个设备变为不能使用时，或是因为它不再存在（例如，已经从内核中被注销），或是因为管理原因而被关闭，此时用到该设备的所有路由项也就不能被使用。这意味着路由表和路由缓存需要得到这类事件的通知并作出反应。我们将在第三十四章中看到路由表如何来处理这类事件。这里我们将看到如何来清理路由缓存。缓存路由项的dst_entry结构可以被插入到以下两个地方中的一个地方：

- 路由缓存。
- dst_garbage_list链表。只有当待删除的路由项的引用都被释放，才能够被垃圾回收过程视为可以被删除。

缓存内的表项被事件通知处理函数fib_netdev_event（在第三十二章“对路由表的影响”一节中描述）接管，它刷新缓存。在dst_garbage_list链表中的表项被DST向netdev_chain通知链注册的函数接管。下面的代码片断来自*net/core/dst.c*文件，DST处理事件通知的处理函数为dst_dev_event：

```
static struct notifier_block dst_dev_notifier = {
    .notifier_call = dst_dev_event,
};

void __init dst_init(void)
{
    register_netdevice_notifier(&dst_dev_notifier);
}
```

dst_dev_event遍历由dead dst_entry结构组成的dst_garbage_list链表，对每一项调用dst_ifdown。dst_ifdown的最后一个输入参数为要处理的事件。这里给出它如何来处理这两类事件：

NETDEV_UNREGISTER

当设备被注销时，对该设备的所有引用都必须被删除。dst_ifdown将dst_entry结构和相关的neighbour实例中到该设备的引用都替换为到回环设备的引用（注4）。

NETDEV_DOWN

因为设备为关闭，所以不再能够向该设备发送流量。因而，dst_entry中的input和output函数被分别设置为dst_discard_in和dst_discard_out。这两个函数将送来的任何输入buffer（即它们被要求处理的任意帧）简单丢弃掉。

我们在“IPsec变换与dst_entry的使用”一节中看到，一个dst_entry结构能够与其他的dst_entry结构通过child 指针链接在一起。dst_ifdown沿着每一个child指针来更新所有的dst_entry项。只对最后一项更新input和output函数，因为只有最后一项使用这些函数来接收或发送。

我们在第八章中看到注销一个设备不仅触发一个NETDEV_UNREGISTER通知，而且还触发一个NETDEV_DOWN通知，因为设备只有被关闭，才能够被注销。这意味着当一个

注4： 参见第二十七章“L2帧头缓存”小节。

设备被注销时，dst_dev_event要处理两类事件。这解释了为什么dst_ifdown检查其unregister参数，当该参数被设置时特意跳过部分代码而仅执行其他的部分。

刷新路由缓存

一旦系统中发生变化，使缓存中的一些信息变为过期，内核就刷新路由缓存。在许多情况下只有一些表项过期，但内核为了简化而删除所有的表项。触发刷新的主要事件包括：

设备启动或关闭

通过一个给定设备在过去可达的一些地址，可能不再可达或者是由于有更好的路由而通过另一个设备可达。

向设备添加或删除一个IP地址

我们在第三十二章“添加一个IP地址”和“删除一个IP地址”一节中看到，Linux对每个本地配置的IP地址创建一个具体路由。当一个地址被删除时，缓存内相关的所有路由项也要被删除。被删除地址最有可能配置的网络掩码不是/32，所以在同一子网内与该地址相关的所有缓存项也应当被删除（注5）。最后，如果同一子网内有一个地址被用作其他非直连路由的网关，与这些地址相关的所有缓存项也应当被删除。而刷新整个缓存要比跟踪所有这些情况更为简单。

全局转发状态或设备的转发状态发生变化

如果禁止转发，就需要删除可以转发流量的所有缓存路由项。参见第三十六章“开启和关闭转发”一节。

一条路由被删除

与被删除路由项相关的所有缓存表项需要被删除。

通过/proc接口请求进行管理性刷新

这在第三十六章“/proc/sys/net/ipv4/route 目录”一节中描述。

用于刷新缓存的函数是rt_run_flush，但它从来不被直接调用。请求对缓存刷新是通过rt_cache_flush函数来实现，它根据调用方提供的输入参数delay的取值，决定是立即刷新缓存还是启动一个定时器。当delay的取值为以下几种情况时：

小于0

缓存在内核参数ip_rt_min_delay指定的时间后被刷新，该参数可以通过*/proc*来调整，参看第三十六章“/proc/sys/net/ipv4/route 目录”一节中的描述。

注5： 当删除一个第二地址时这并不为真，参见第三十二章“删除一个IP地址”小节。

等于0

缓存立刻被刷新。

大于0

缓存在指定的时间后被刷新。

一旦递交了一个刷新请求，在ip_rt_max_delay秒之内肯定进行一次刷新，这个参数默认值为8秒。当一个刷新请求被递交且已经存在一个刷新请求准备执行的情况下，重启定时器来反映这个新请求。但这个新请求不能让定时器迟于ip_rt_max_delay秒之后才过期，因为那时前一个定时器已经执行了。这是通过使用全局变量rt_deadline来完成的。

除此之外，缓存还被一个周期性定时器rt_secret_timer定期刷新，该定时器每隔ip_rt_secret_interval秒（该参数默认值见第三十六章“/proc/sys/net/ipv4/route 目录”一节）之后到期。当该定时器到期后，处理函数rt_secret_rebuild对缓存刷新并重启该定时器。可以通过*/proc*来配置ip_rt_secret_interval参数。

垃圾回收

在第三十章“路由缓存的垃圾回收”一节中曾讲到，有两种垃圾回收：

- 当检测到内存不够时需要释放内存。这实际上分为两个任务，一个同步任务和一个异步任务。同步任务是由特殊条件在无规律的时间点触发，异步任务是在一个定时器到期时或多或少有规律地运行。
- 清理内核要求删除的dst_entry结构，它们可能因为还被某些程序引用而不能被立刻删除。

本节讨论第一种垃圾回收中的同步任务和异步任务，第二种垃圾回收在“删除DST表项”一节中详细描述。

同步和异步垃圾回收使用一个公共函数来判断一个给定的dst_entry实例是否符合删除条件：rt_may_expire。这个函数接受两个参数（tmo1，tmo2）来表示候选者在符合删除条件之前必须在缓存中度过的最短时间。tmo2用于那些被认为特别适合删除的候选项，tmo1用于其他所有候选项，在第三十章“合适的缓存牺牲品范例”一节描述过。ip_rt_gc_timeout 参数指定缓存中其他表项的时间。

这两个值越低，对应的缓存表项就越有可能被删除。这就是为什么每次删除一个表项时，rt_check_expire就将局部变量tmo减半，参见“异步清理”一节中的描述。在

“rt_garbage_collect 函数”一节中我们还将看到，rt_garbage_collect 对这两个阈值也是进行同样的处理。

同步清理

当 DST 子系统检测到内存不足时，就触发一个同步清理。尽管是由 DST 来决定何时触发垃圾回收，但处理函数是由拥有缓存的协议提供的。所有事情都通过在“DST 与调用协议间的接口”一节中有描述过的 dst_ops 结构中的虚函数来控制。在那一节中我们知道 dst_ops 有一个 gc 函数，IPv4 将它初始化为 rt_garbage_collect。在以下两种情况下调用 gc：

- 当一个新表项要添加到路由缓存中但发现内存不够时。当添加一个路由表项时，rt_intern_hash 必须将该路由和与下一跳相关的 neighbour 数据结构绑定（参见“将路由缓存绑定到 ARP 缓存”一节）。如果没有足够的内存来分配一个新的 neighbour数据结构，则扫描路由缓存试图释放部分内存。这样做的原因是可能有一些缓存项暂时没有被使用，删除这些缓存项也可能删除相关的 neighbour 表项（这里用的是“可能”，因为一个数据结构只有在所有对它的引用都被删除后，该结构才能够被删除）。
- 当一个新表项要添加到路由缓存中，但表项总数将超过阈值 gc_thresh 时。分配表项的 dst_alloc 函数就会触发内存清理，将缓存占用的内存限制在一个固定值以下。可以通过 */proc* 来配置 gc_thresh（参见第三十六章“通过 /proc 文件系统调整”一节中的描述）。

下面一个一节讨论 rt_garbage_collect 的内部细节。

rt_garbage_collect 函数

rt_garbage_collect 的逻辑流程如图 33-8(a)和 33-8(b)所示。

rt_garbage_collect 函数所做的垃圾回收需要花费大量的 CPU 时间。因此，如果上次调用该函数的时间到现在的间隔小于 ip_rt_gc_min_interval 秒，则不做任何事而立即返回。除非缓存内的表项数目达到最大值 ip_rt_max_size，这时要求立刻执行该函数。

ip_rt_max_size 是一个硬性限制。一旦达到这个阈值，dst_alloc 函数就会失败，除非 rt_garbage_collect 设法释放部分内存。

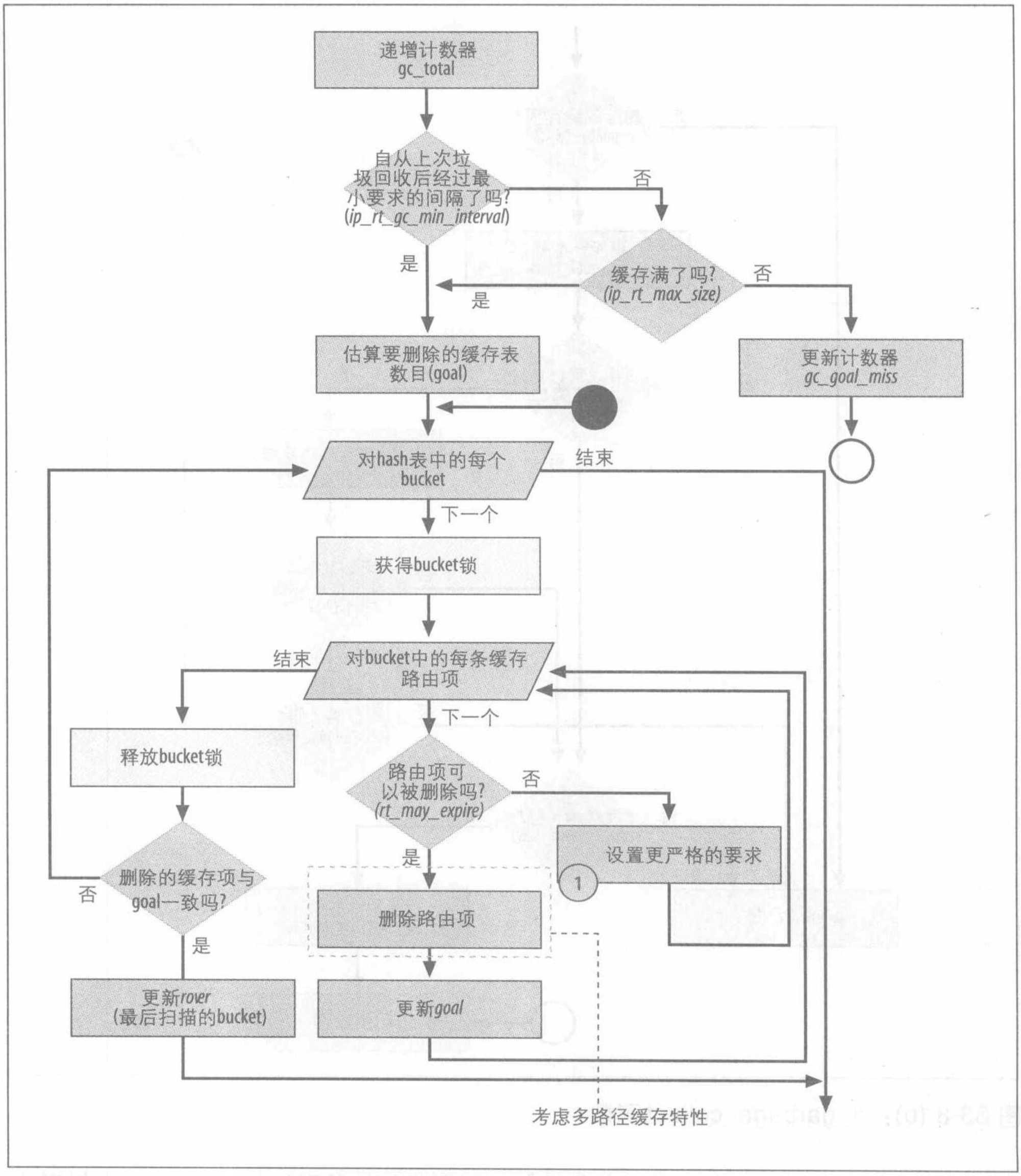


图 33-8 (a)：rt_garbage_collect 函数

下面给出 `rt_garbage_collect` 的逻辑结构：

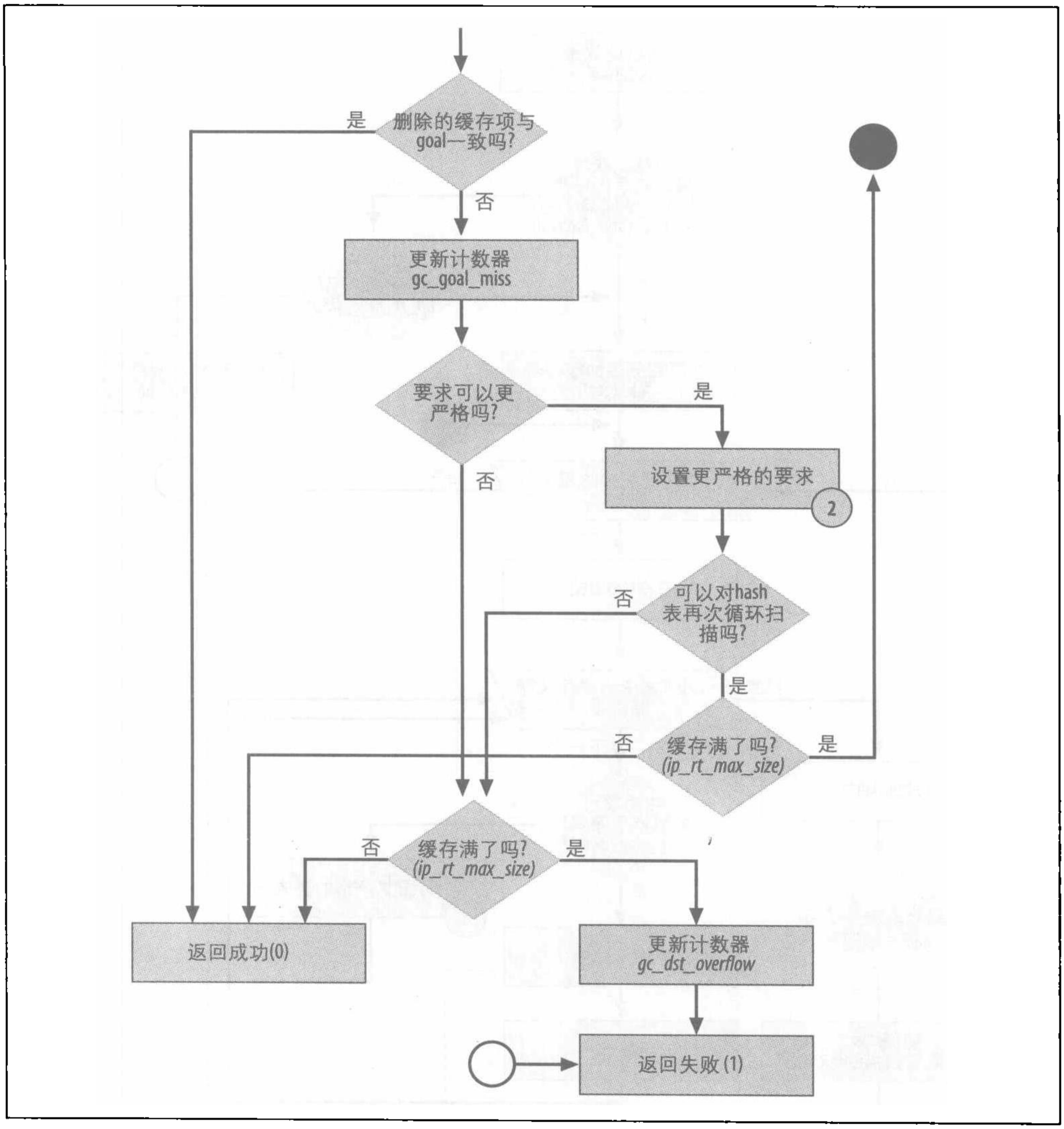


图 33-8 (b)：rt_garbage_collect 函数

- rt_garbage_collects 函数首先计算可能删除的缓存表项数目（goal）。根据这个值及缓存内当前的表项数目（ipv4_dst_ops.entries），它计算出一旦 goal 个表项被删除后，剩余的表项数目，这个数目被保存到 equilibrium 中。
- 接着它遍历 hash 表，尝试使最符合条件的表项过期，利用 rt_may_expire 检查它们是否符合过期条件。将符合删除条件的表项调用 rt_free 直接删除，当它们与

多路径路由相关时，则调用rt_remove_balanced_route来删除（参见“辅助函数”一节）。

- 一旦缓存hash表被完全扫描过，rt_garbage_collect就检查删除的表项数是否已经达到goal个。如果没有，则用更为严格选择标准来重新扫描hash表。

被删除的表项数（goal）取决于hash表的负载情况。目的是当hash表负载越大时，其表项过期也就越快。

利用图33-9的帮助，我们可以澄清一下rt_garbage_collect中定义goal时用的一些阈值：

- hash表的大小为rt_hash_mask+1，或$2^{rt_hash_log}$。当缓存内的表项数目超过gc_thresh时，就调用rt_garbage_collect，gc_thresh的默认值为hash表的大小。
- 缓存内能够容纳的表项最大数目是ip_rt_max_size，其默认值为hash表的大小乘以16。
- 当缓存内的表项数目超过ip_rt_gc_elasticity*（$2^{rt_hash_log}$）时，其默认值为hash表的大小的8倍，就认为缓存容量太大，存在风险，垃圾回收开始设置goal更为严格。

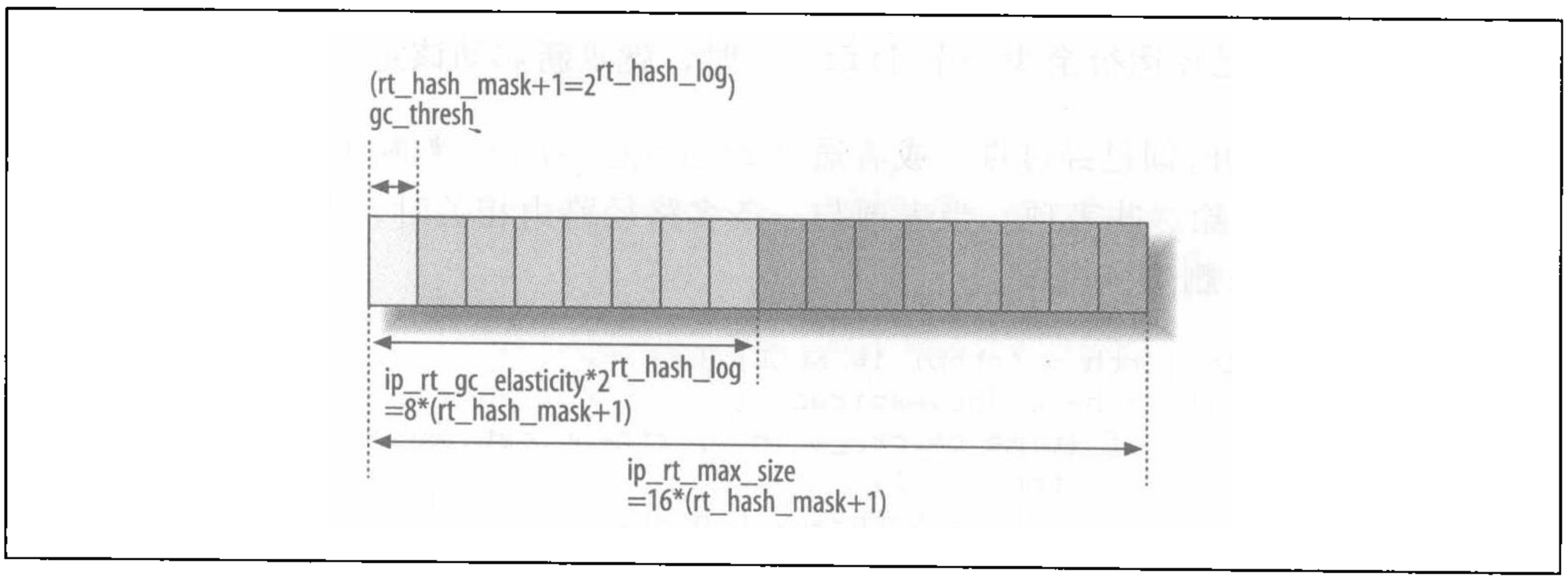


图33-9：垃圾回收阈值

一旦确定好了各个阈值，rt_garbage_collect就开始遍历hash表来寻找牺牲品。但hash表的遍历并不是简单地从第一个bucket开始到最后一个bucket。rt_garbage_collect用一个静态变量rover，来记住前一次调用自己时扫描到的最后一个bucket。这是因为每次没必要完整地扫描一遍hash表。通过记住上次扫描的bucket，rt_garbage_collect就可以公平对待所有的bucket，而不总是从第一个bucket开始来选择受害者。通过rt_may_expire函数来识别牺牲品。这个函数在“垃圾回收”一节中描述

过，它需要传递两个时间阈值来确定符合删除条件的两类表项。当扫描一个bucket的元素时，每当一个元素不符合删除条件，则降低（减半）第一个时间阈值。当扫描到每个bucket链表的末尾时，函数将再次检查被删除表项的数目是否和函数开始时设置的数目goal一致。如果不是，函数将继续扫描下一个bucket。该过程一直继续直到整张表扫描完。这时，rt_garbage_collect函数将传递给rt_max_expire的第二个时间阈值降低（减半），以便更有可能寻找到符合条件的受害者。如果重新扫描hash表不会花费太多时间的话，则重新开始扫描。如果rt_garbage_collect函数是在软中断上下文中被调用，或者前一次扫描花费的时间超过一个jiffies（在x86平台上即为1/1000秒），则认为重新扫描hash表将花费太多的时间，因而跳过不做。

异步清理

同步垃圾回收被用于处理内存不够时的特殊情况。但在采取行动之前避免出现极端条件更为可取：换句话说，极端条件出现的概率越低越好。这正是通过周期性定时器来异步清理所要做的。

当路由子系统初始化时，ip_rt_init启动定时器rt_periodic_timer，当它到期时激活处理函数rt_check_expire。每当rt_check_expire被激活时，它只扫描一部分缓存。它保持一个静态变量（rover）来记住前一次函数激活时扫描到的最后一个bucket，每当该函数激活时就从下一个bucket开始扫描。当rt_check_expire完成扫描整张hash表或已经运行至少一个jiffies时，就重新启动该定时器，函数返回。

如果缓存内表项的时间已经过期，或者通过rt_may_expire判断出符合删除条件，则调用rt_free删除这些表项。当表项与一条多路径路由相关时，调用rt_remove_balanced_route删除该表项。

```
        while ((rth = *rthp) != NULL) {
            if (rth->u.dst.expires) {
                if (time_before_eq(now, rth->u.dst.expires)) {
                    tmo >>= 1;
                    rthp = &rth->u.rt_next;
                    continue;
                }
            } else if (!rt_may_expire(rth, tmo, ip_rt_gc_timeout)) {
                tmo >>= 1;
                rthp = &rth->u.rt_next;
                continue;
            }
            /* Cleanup aged off entries. */
#ifdef CONFIG_IP_ROUTE_MULTIPATH_CACHED
            /* remove all related balanced entries if necessary */
            if (rth->u.dst.flags & DST_BALANCED) {
                rthp = rt_remove_balanced_route(
```

```
                        &rt_hash_table[i].chain,
                        rth, NULL);
                    if (!rthp)
                        break;
                } else {
                    *rthp = rth->u.rt_next;
                    rt_free(rth);
                }
#else /* CONFIG_IP_ROUTE_MULTIPATH_CACHED */
                *rthp = rth->u.rt_next;
                rt_free(rth);
#endif /* CONFIG_IP_ROUTE_MULTIPATH_CACHED */
            }
            ...
            if (time_after(jiffies, now))
                break;
```

该定时器默认每隔ip_rt_gc_interval秒到期，该参数默认值为60，但可以通过*/proc/sys/net/ipv4/route/gc_interval*文件来调整（参见第四章“通过/proc文件系统的调整”一节）。当该定时器第一次到期时，设置下一次到期是在 ip_rt_gc_interval 和 2×ip_rt_gc_interval （见 ip_rt_init）之间的一个随机时间之后。使用随机值的原因是为了避免不同内核子系统的定时器可能在同一时间到期而花费大量CPU的可能性。如果在启动过程中许多子系统同时启动，并且都是按照相同的间隔时间来调度，那么就可能出现这种情况。

过期标准

默认情况下，路由缓存表项永远不会过期，因为dst_entry→expires为0（注6）。当能够使缓存表项过期的事件发生时（参见第三十章“使路由缓存项过期的事件范例”一节），利用dst_set_expires函数（注7）设置表项的dst_entry→expires时间戳字段为一个非零值来使表项能够过期：

- 当接收到一个ICMP UNREACHABLE或FRAGMENTATION NEEDED消息时，所有相关路由项（缓存中目的地址与ICMP消息payload中携带有IP头的目的IP地址相同的路由缓存项）的PMTU必须被更新为ICMP头中指定的MTU。ICMP核心代码调用ip_rt_frag_needed来更新路由缓存。这些受影响的表项在可配置的时间 ip_rt_mtu_expires 之后被设置为过期，这个时间值默认为10分钟，可以用*/proc/sys/net/route/mtu_expires*来改变。更多细节可参见第二十五章。

注6：在dst_alloc中调用memset来设置dst_entry→expires字段。

注7：注意当使一个表项立即过期而调用dst_set_expires时，用1来代替输入值0，以便将这种情况与表示永远不过期的0区别开来。

- 当TCP代码用路径MTU发现算法来更新一条路由的MTU时，调用ip_rt_update_pmtu函数，该函数将调用dst_set_expires。参见第十八章对路径MTU发现算法的更多细节描述。
- 当一个目的IP地址被认为不可达时，通过直接或间接调用dst_ops数据结构中的link_failure方法，将缓存内相关的dst_entry结构标记为不可达（参见“DST与调用协议间的接口”一节）。

删除DST表项

在前面的一节中，我们看到rtable缓存表项是如何被同步清除或异步清除，以及被后台的垃圾回收所删除。在本一节内，我们将看到如何来处理rtable内嵌入的dst_entry结构，删除dst_entry的函数是dst_free。

dst_hold和dst_release函数分别用于递增或递减一个dst_entry的引用计数。但是当调用dst_release释放最后一个引用时，该表项并不被自动删除。实际上，dst_entry是在rt_free和rt_drop删除相关的rtable结构时被间接删除。这两个函数通过dst_rcu_free调度执行dst_free，这涉及到RCU机制（参见“缓存锁”一节）。

我们在“IPsec变换与dst_entry的使用”一节中看到，dst_entry结构并不总是嵌入在rtable结构内。孤立的dst_entry实例可通过调用dst_free来直接删除。

一个dst_entry的删除并不复杂，但是要理解dst_free及其辅助函数是如何工作，就需要了解以下要点：

- 当一个表项仍然被引用时不能被删除时，设置其obsolete标识为2来标记它为dead（dst→obsolete的默认值为0）。试图删除一个已经标记为dead的表项将失败。
- 我们在“IPsec变换与dst_entry的使用”一节中看到，一个dst_entry实例可能有children。当删除一个链表内的第一个dst_entry时，路由子系统也必须删除所有其他的表项。但与此同时，需要记住只要表项还有引用就不能被删除。

清楚了上面两点，我们现在来看dst_free是如何工作的。

当调用dst_free删除一个引用计数为0的表项，则调用dst_destroy立即删除该表项。dst_destroy函数也尝试去删除链接到该结构的任何children。当有一个children由于仍然被引用而不能被删除时，dst_destroy返回指向该child的一个指针，由dst_free来处理它。

当调用dst_free删除一个引用计数非0的表项，这包括刚才描述的dst_destroy不能删除一个child时的情况，则进行以下处理：

- 通过设置其obsolete标识来标记该表项为dead。
- 用两个钩子函数dst_discard_in和dst_discard_out来替换该表项原来的input和output函数，以此来确保相关的路由不能够接收和发送封包（参见第三十六章“dst_entry结构”一节内对input和output的描述）。这种处理方式是在设备还没有处于运行或处于down状态（没有设置IFF_UP标识）时的典型做法。

我们在“外部事件”一节内看到当dst_dev_event处理的两类事件发生时，调用dst_ifdown来处理dst_garbage_list中的dst_entry结构。只有在特殊情况下才用dst_discard_in和dst_discard_out来替换该结构内当前的input和output方法。这种做法并非多余，因为只有当释放dst_entry是与设备被关闭相关时dst_free才这样做，而dst_dev_event处理的事件并不一定总是设备被关闭事件。

- 将dst_entry结构添加到dst_garbage_list全局链表内，该链表将所有应当被删除的，但由于引用计数非0而还没有被删除的表项链接在一起。
- 调整dst_gc_timer定时器在可配置的最小延迟时间（DST_GC_MIN）后到期，在该定时器还没有运行时激活它。

dst_gc_timer定时器周期性地遍历dst_garbage_list链表，利用dst_destroy来删除引用计数为0的表项。当定时处理函数dst_run_gc没有删除链表内所有的表项时，就再次启动该定时器但使它在稍后一点时间到期。更准确地说，该定时器的到期时间再延迟DST_GC_INC秒，最长延迟时间为DST_GC_MAX。但是当每次dst_free将一个新表项添加到dst_garbage_list链表中时，重新设置到期延迟时间为默认最小值DST_GC_MIN 。

图33-10（a）和33-10（b）给出了dst_free的逻辑流程图。

调整和控制垃圾回收的变量

下面总结了控制DST垃圾回收任务所用到的全局变量和参数的含义：

dst_garbage_list

dst_entry结构组成的链表等待被删除。当dst_gc_timer定时器到期时，定时器处理函数就会处理这个链表。只有引用计数__refcnt大于0的待删除缓存表项才被放入该链表内（而不是直接删除），以避免被直接删除。新表项被插入到链表首部。

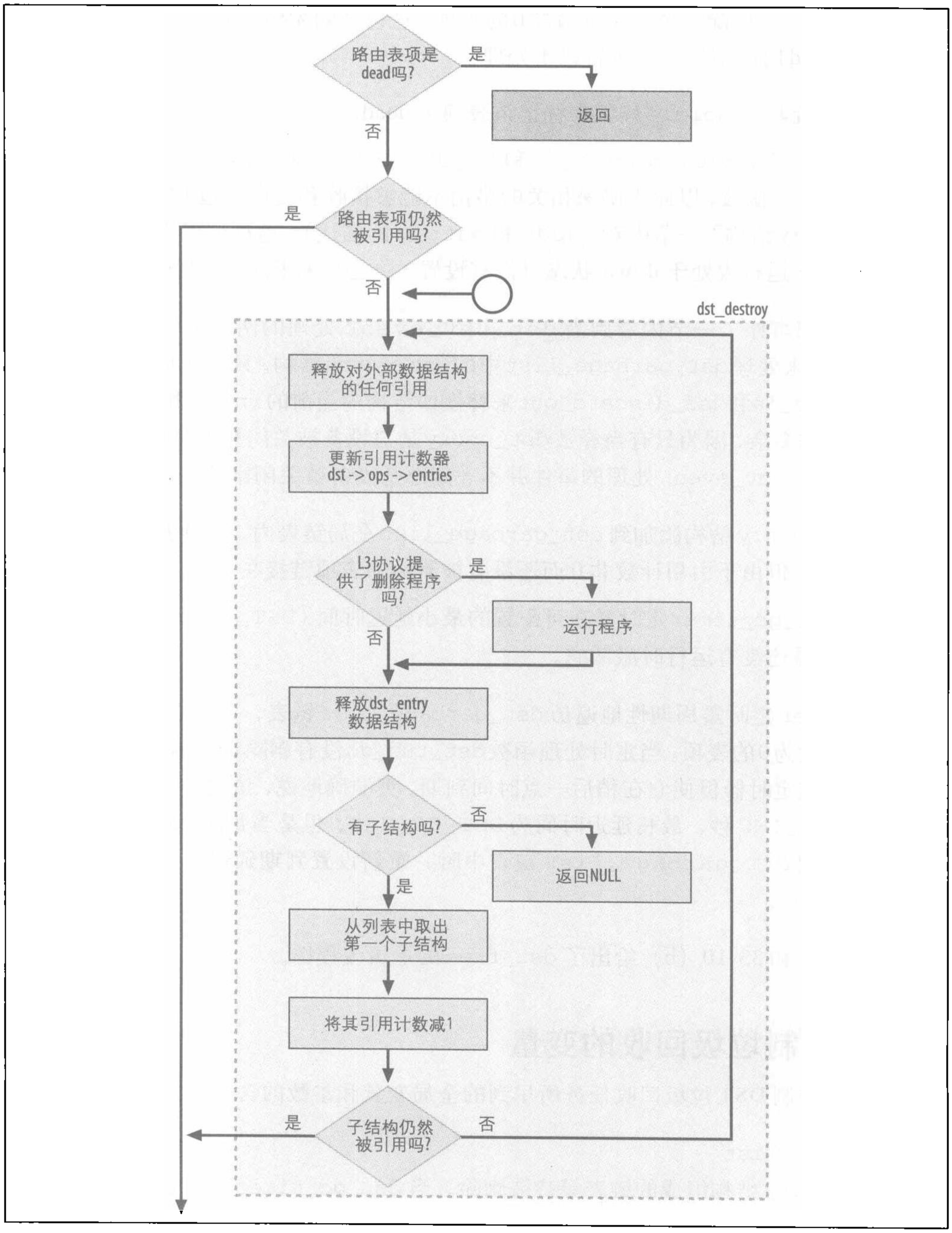


图 33-10 (a)：dst_free 函数

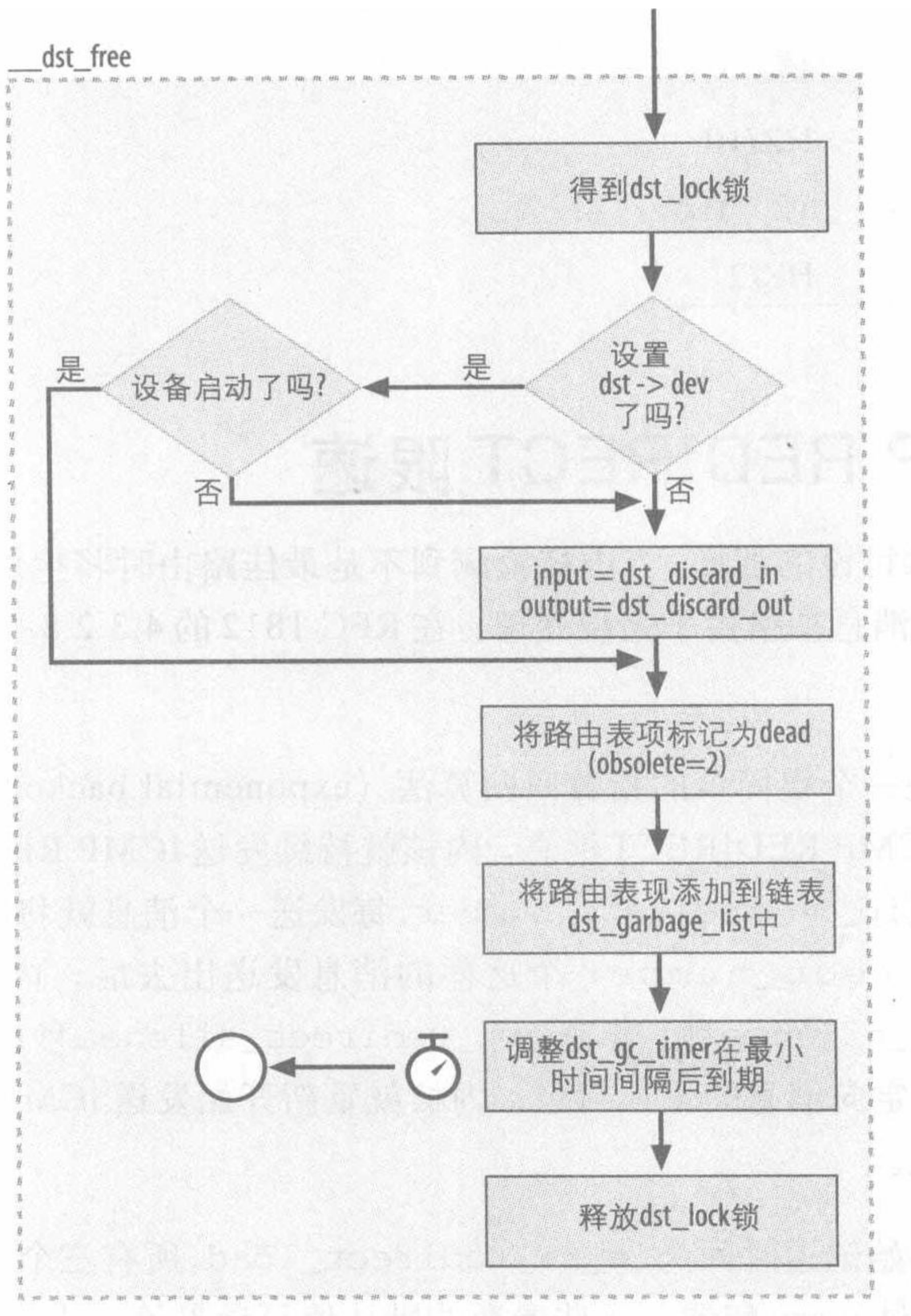


图 33-10 (b)：dst_free 函数

dst_gc_timer_expires

dst_gc_timer_inc

dst_gc_timer_expires 是定时器在到期之前等待的秒数，取值范围在DST_GC_MIN和DST_GC_MAX之间，当定时器处理函数dst_run_gc运行但没能清空dst_garbage_list 链表时，dst_run_gc 将这个等待时间增加dst_gc_timer_inc 个单位。但 dst_gc_timer_inc 必须在从 DST_GC_MIN～DST_GC_MAX 的范围内。

在表 33-1 中列出了前面提到的三个常量的取值，这些常量在 *include/net/dst.h* 文件中定义。

表 33-1：DST_GC_XXX 常量

名称	值
DST_GC_MIN	HZ/10
DST_GC_MAX	120*HZ
DST_GC_INC	HZ/2

出口 ICMP REDIRECT 限速

就像在第二十五章讨论的那样，当内核检测到不是最佳路由时将生成ICMP REDIRECT消息。这些ICMP消息被路由子系统处理，在RFC 1812的4.3.2.8一节中建议对它们进行限速。

限速所用的算法是一个很简单的指数回归算法（exponential backoff algorithm）。如果目的地持续忽略ICMP REDIRECT消息，内核就持续发送ICMP REDIRECT消息给它，直到发送数目达到ip_rt_redirect_number，每发送一个消息就将间隔时间增加一倍。当 ip_rt_redirect_number 个这样的消息发送出去后，内核停止发送，等待ip_rt_redirect_silence秒。若ip_rt_redirect_silence秒后，收到了能够触发内核生成ICMP重定向消息的输入封包，内核就重新开始发送ICMP REDIRECT消息；否则，就一直停止。

指数回归算法的初始延迟时间为ip_rt_redirect_load，所有三个ip_rt_redirect_xxx参数都可以通过/proc配置，这些参数的默认值可参见第三十六章。

出口 REDIRECT 消息的所有逻辑是在 ip_rt_send_redirect 中实现。当内核检测到需要发送一个ICMP REDIRECT消息时，就调用该函数来处理（参见第二十章）。

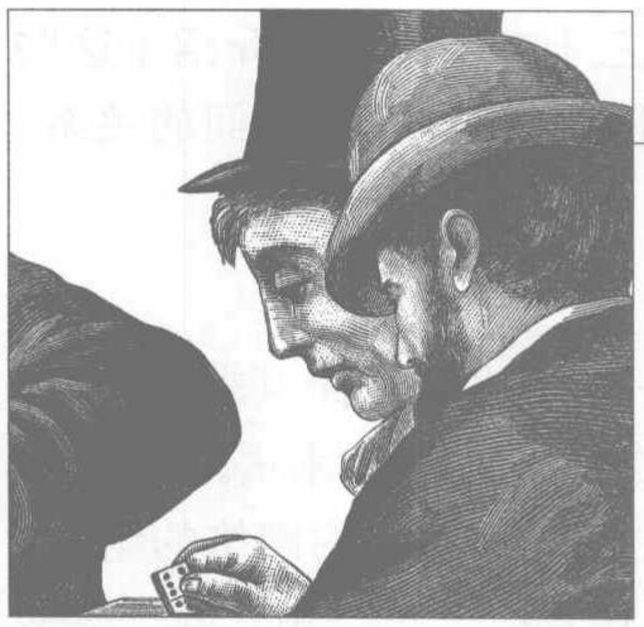
dst_entry结构中有两个字段用于实现ICMP REDIRECT消息限速：

rate_last

上一个IMCP REDIRECT消息送出的时间戳。

rate_tokens

已经向与该dst_entry实例相关的目的地发送的ICMP REDIRECT消息的数目。所以，rate_tokens-1表示目的地连续忽略的ICMP REDIRECT消息的数目。



第三十四章

路由：路由表

由于路由在网络栈中的核心地位及路由表的容量可以很大，因而设计高效的路由表来加快各种操作，尤其是路由查找操作就很重要。本章描述了 Linux 是如何来组织路由表，如何利用各种 hash 表来访问组成路由表的各个数据结构，每一种 hash 表都是根据不同的查找方法而专门设计的。

路由 Hash 表的组织

为了能使各种操作快速查找到相关信息，Linux 中采用了几种不同的 Hash 表，这些表都指向用于描述路由的相同的数据结构：

- 一组基于网络掩码长度来访问路由的 hash 表（在“基于网络掩码的 hash 表的组织”一节中描述）。
- 一组直接搜索 `fib_info` 结构的 hash 表（在“fib_info 结构的组织”一节中描述）。
- 一个以设备为索引，可以快速搜索配置路由的下一跳的 hash 表（在“下一跳路由器结构的组织”一节中描述）。
- 一个以设备和一条路由为索引，能够快速识别该路由下一跳所用网关的 hash 表。

基于网络掩码的 Hash 表的组织

在最高层，路由基于它们的网络掩码长度被组织成不同的 hash 表。因为 IPv4 使用 32 位地址，所以 IPv4 中有 33 个不同的网络掩码长度（从 /0 到 /32，其中 /0 表示默认路由），可对应于一个 IP 地址。路由子系统为每一个网络掩码长度维护一个不同的 hash 表，然后这些 hash 表被放进下面将提到的其他表中。

图34-1给出了一个路由表中主要数据结构之间的关系。在第三十二章中简要介绍了这些主要数据结构，详细介绍可参见第三十六章，本章主要描述这些数据结构之间的关系。

Hash 表组织的基本结构

fib_table数据结构来描述路由表。fib_table结构包含一个由33个指针组成的向量，每一个指针对应一个网络掩码并指向一个类型为fn_zone的数据结构。(术语*zone*是指网络掩码相同的网络)。fn_zone结构将路由组织成hash表，因此通向目的网络的路由，如果网络掩码长度相同，就被放在同一个hash表内。因此，任意给一条路由，可以根据其网络掩码长度快速得到其对应的hash表。非空的fn_zone bucket被链接在一起，该链表的头保存在fn_zone_list 中。我们将在第三十五章中看到该链表的使用。

对于这些基于网络掩码的普通hash表来说，有一个例外：那就是默认路由/0 zone对应的hash表，仅由包含一个bucket的组成，因此就被放进一个简单链表中。这样设计的原因是因为一台主机很少维护多个默认路由。

路由是由多个不同的数据结构的组合来描述的，每个数据结构代表着路由信息的不同部分。定义一条路由所需的信息被分成多个数据结构。这样做的原因是只需通过部分字段可以区分多条路由。路由子系统不是维护一个庞大而臃肿的结构而是将路由分散为多个片段，这样更容易在相似的路由间共享通用的信息，因而就可以分离出不同的函数，并在这些函数之间定义更加清晰的接口。

每个单独的子网对应一个fib_node实例，用变量fn_key识别，它的值就表明该子网。例如，对于子网10.1.1.0/24，fn_key为10.1.1。注意：fib_node结构（即该结构中的fn_key变量）与一个子网相关联，而不是与一条路由关联；一定要记住这一点以免后面混淆。这个细节的重要性是由于到达同一个子网，可能有多条不同的路由。

目的子网相同的不同路由（即fn_key相同）共享同一个fib_node。每条路由有自己的fib_alias结构。例如，你可能有一些通向同一子网的不同路由，它们只是TOS值不同：因此，每一个fib_alias实例就被指定一个不同的TOS值。每个fib_alias实例都与一个fib_info结构相关联，该结构中保存着真实路由信息（即如何到达目的地）。

对于一个fib_node实例，fib_alias实例的相关链表按照IP TOS（即fa_tos字段）的递增顺序排列。fa_tos值相同的fib_alias实例按照关联的fib_info中fib_protocol字段的递增顺序排列。

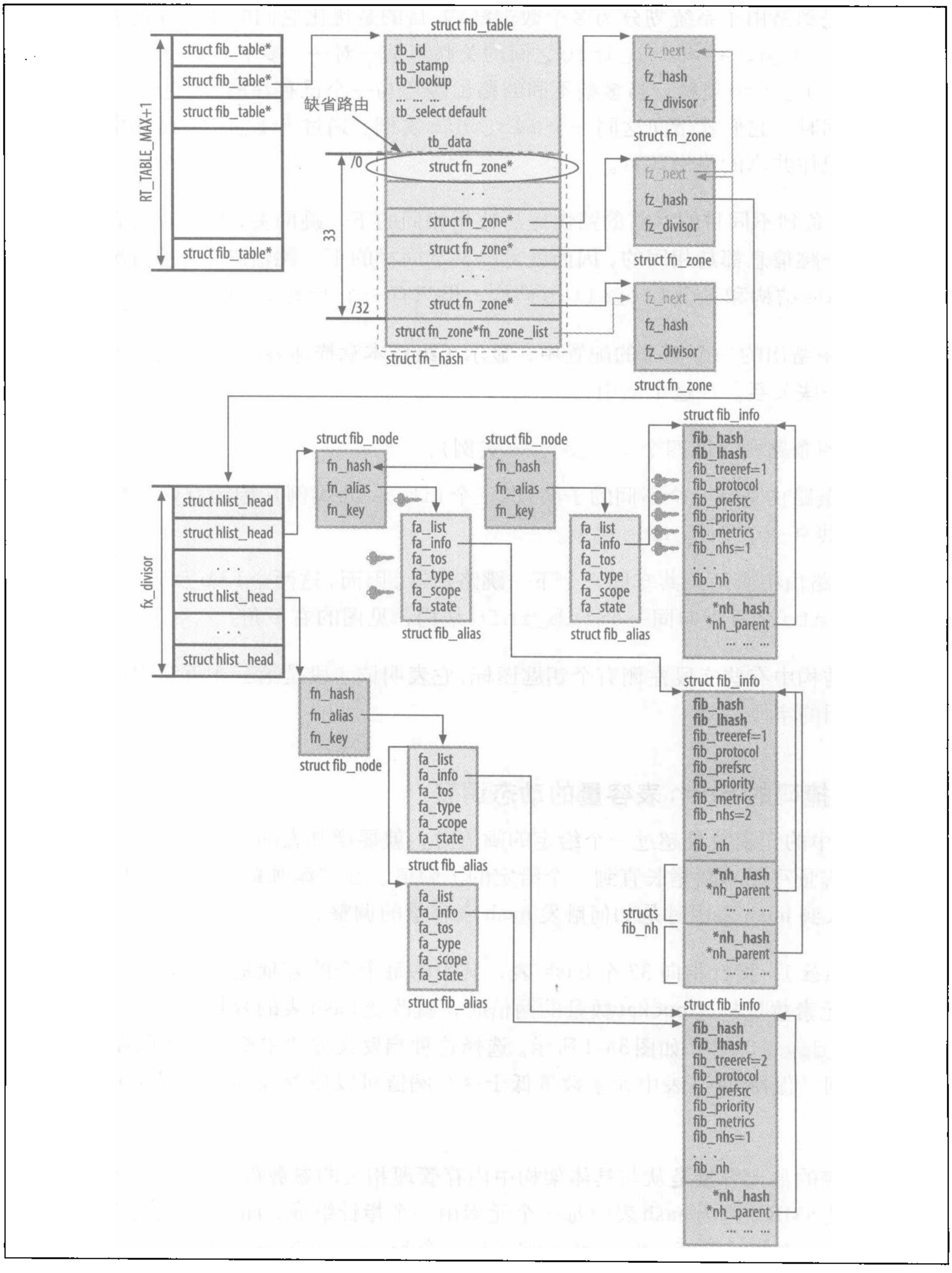

struct fib_table
tb_id
tb_stamp
tb_lookup
... ...
tb_select default
tb_data
缺省路由
struct fib_table*
RT_TABLE_MAX+1
/0
/32
33
struct fn_zone*
struct fn_zone*fn_zone_list
struct fn_hash
fz_next
fz_hash
fz_divisor
struct fn_zone
struct hlist_head
fx_divisor
struct fib_node
fn_hash
fn_alias
fn_key
fa_list
fa_info
fa_tos
fa_type
fa_scope
fa_state
struct fib_alias
struct fib_info
fib_hash
fib_lhash
fib_treeref=1
fib_protocol
fib_prefsrc
fib_priority
fib_metrics
fib_nhs=1
fib_nh
*nh_hash
*nh_parent
fib_nhs=2
structs
fib_nh

图 34-1：路由表的组织

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的路由数目。

例如，有五条到不同目的网络的路由碰巧使用相同的下一跳网关，那么对所有这五条路由来说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34-1中给出的一个例子的配置中，显示了组成本章描述各种hash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（即四个fib_alias实例）。
- 这四条路由是到三个不同的子网(即三个fib_node实例)，因为有两个fib_alias实例共享一个fib_node实例。
- 四条路由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。

图中数据结构中有些字段左侧有个钥匙图标，它表明该字段是第三十五章中介绍的查找程序将使用的字段。

基于网络掩码的Hash表容量的动态调整

当Hash表中的元素数量超过一个给定的阈值时，就要增加表的容量fz_hash。一个hash表的容量可以不断增长直到一个给定的上限值。在“添加路由”一节中将解释新元素被插入到hash表中时是如何触发hash表容量的调整。

fn_hash（注1）指针指向33个hash表，其中的每个表的容量是可以独立调整的。当hash表中元素数量超过bucket数量的两倍时，就改变hash表的容量。bucket数量存储在变量fz_divisor中，如图34-1所示。选择这种启发式方法主要是为了限制对hash表的查找时间。保持hash表中元素数量低于这个阈值可以使查找加快（假定元素分布均匀）。

一个hash表的最大容量是从与具体架构中内存管理相关的参数得出的。在i386架构中，最大容量是8MB。因为hash表中每一个元素由一个指针组成，而在32位处理器上每个指针占用4个字节，因而i386系统能够支持一个hash表的容量超过200万个bucket。

注1：　不要将fn_hash与fz_hash混淆。

当通过 fn_new_zone 第一次创建一个 hash 表时，该表的默认容量为 16 个 bucket。（如前面的章节所述，唯一的例外是 /0 zone 用于默认路由）。在 Hash 表容量扩展的最初两次时，容量首先被增大到 256，然后是 1024。随后的容量扩展总是在当前容量基础上翻一倍。

目前还没有容量缩减机制，所以如果一个 zone 的 hash 表中元素数量从 280 减少为 10，hash 表的容量不会从 1024 减小为 16。

fib_info 结构的组织

如图 34-1 所示，每一个 fib_info 结构包含两个字段：fib_hash 与 fib_lhash，它们用于将结构插入到两个更为复杂的 hash 表中，如图 34-2 所示。这两个 hash 表是：

fib_info_hash

所有的 fib_info 结构被插入到这个 hash 表中，用 fib_find_info 函数来查找该表。

fib_info_laddrhash

只有在路由表项有一个首选源地址时，才将 fib_info 结构插入到这个表中。在第三十五章“首选源地址选择”一节中描述了首选源地址（preferred source address）的使用。这个地址通常是根据设备配置自动得到，但也可以手工配置它。

这个 hash 表主要是为了方便地删除由于删除了本地配置 IP 地址而影响的路由（见 fib_sync_down 函数）。

在这两个 hash 表中，都是通过 fib_create_info 将新元素添加到一个 bucket 链表的首部。

全局 Hash 表容量的动态调整

在所有路由表中，fib_info 结构的总数是存储在计数器 fib_info_cnt 中。当创建一个 fib_info 实例，fib_create_info 将该计数器递增；当删除一个 fib_info 实例，free_fib_info 函数会递减这个计数器。

当创建一个新的 fib_info 实例时，fib_create_info 会检查 fib_info_cnt 是否已经超过 hash 表的容量 fib_hash_size，如图 34-2 所示。当超过 hash 表容量时，将 fib_info_hash 与 fib_info_laddrhash 的容量都要增加一倍。然后用 fib_hash_free 函数删除旧的 hash 表，利用 fib_hash_alloc 分配新的 hash 表，用 fib_hash_move 函数将旧 hash 表中的所有 fib_info 实例转移到新的 hash 表中。

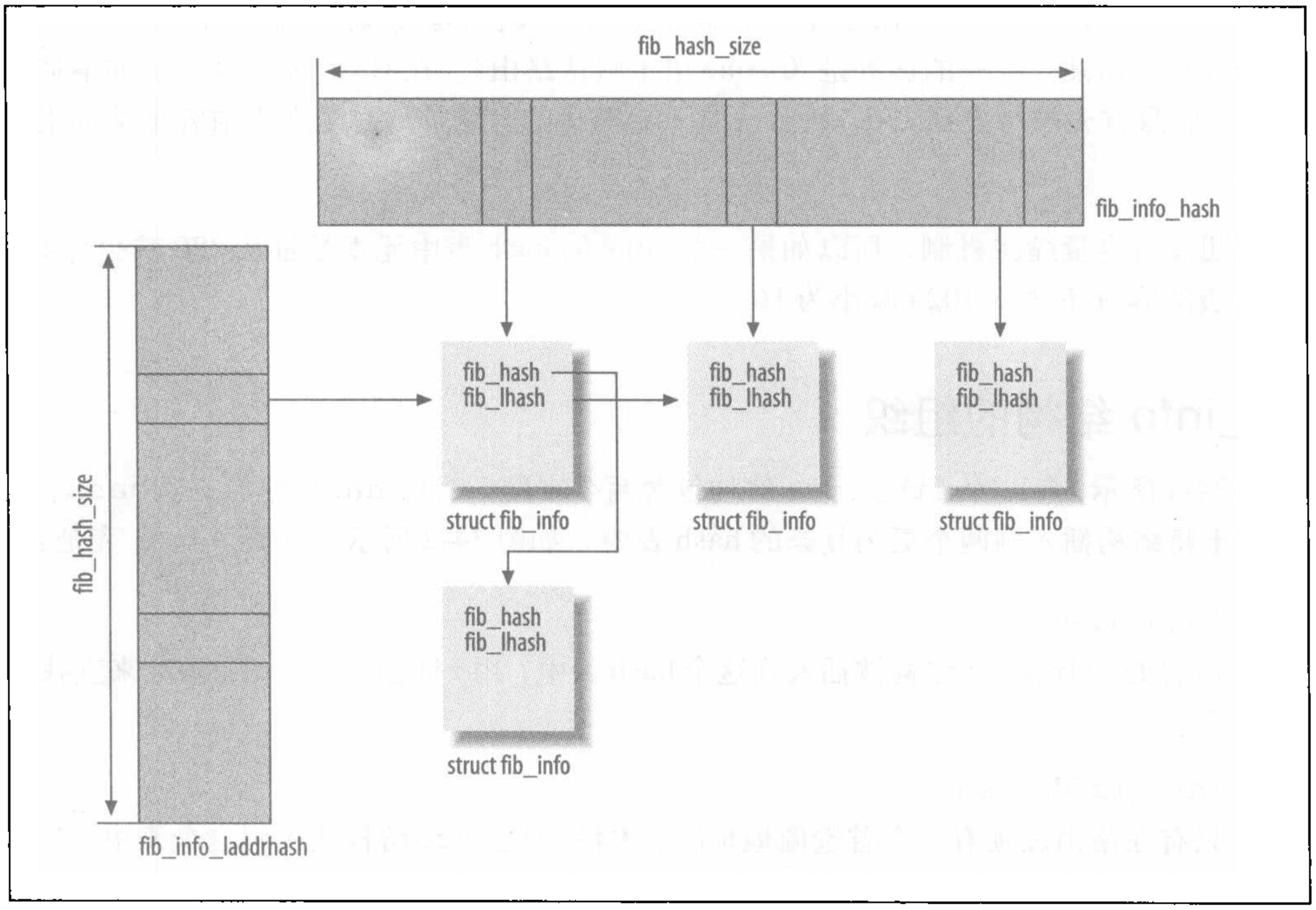


图 34-2：fib_info 结构的组织

注意，本节中讨论的hash表容量的调整和“基于网络掩码的hash表容量的动态调整”一节中讨论的 hash 表容量调整没有任何关系。

下一跳路由器结构的组织

如图34-1所示，每个fib_info结构可以包含一个或多个fib_nh结构，每个fib_nh表示一个下一跳路由器。一个下一跳路由器的信息中包括通过哪个设备能到达该路由器。因而，当已知这样一个路由器时就很容易得到对应的能到达该路由器的设备，但这个结构并没有提供当已知这样一个设备时如何快速查找到路由器的方法，而这种查找能力从两个方面讲是很重要的：

当设备被关闭时

网络子系统必须关闭与该设备相关的所有路由。这个工作通过fib_sync_down来完成，详细描述见第三十二章。

当设备被启动或重新启动时

网络子系统必须启动或重新启动所有与通过该设备可达的下一跳路由器相关的路由。这个工作通过 fib_sync_up 函数来完成的，详细描述见第三十二章。

还有另外一个小情况是与 ICMP_REDIRECT 消息有关。在第三十一章“处理入口 ICMP_REDIRECT 消息”一节中，我们看到只有当ICMP重定向消息中新建议的网关是本地已知的一个路由器时，内核才可能接受该消息。为了检查这个条件是否满足，内核只需要遍历与接收ICMP消息的设备相关的所有路由，查找是否有路由使用新建议的网关作为其下一跳路由器。这个逻辑是由 ip_fib_check_default 函数实现，该函数被 ip_rt_redirect 调用。后者又被 icmp_redirect 调用，icmp_redirect 正是入口 ICMP 重定向消息的处理函数。

这项需求是通过创建另一个以设备标识为索引的hash表来解决的，这样就可以非常快速地查找到下一跳路由器。在图34-1中所示的nh_hash字段是用来将 fib_nh 结构插入 fib_info_devhash hash 表的。该表是在 *net/ipv4/fib_semantics.c* 中静态分配的，容量是 DEVINDEX_HASH SIZE（256）个 bucket。通过 fib_create_info 函数将新元素插入到 fib_info_devhash 表 bucket 链表的首部。

两个默认路由表：ip_fib_main_table 与 ip_fib_local_table

在系统启动时总是创建下面两个路由表，它们与内核配置选项无关：

ip_fib_local_table

内核将到本地地址的路由放在该表中，包括到相关的子网网络地址及子网广播地址的路由，详见第三十六章“内核插入路由项：fib_magic 函数”一节。用户不能够直接配置该路由表。

ip_fib_main_table

所有其他的路由表项（包括用户配置的静态路由，路由协议生成的动态路由）都放在该表内。

在第三十章“特殊路由”一节中，解释了这两个路由表之间的关系。在第三十五章中，还将看路由查找程序如何使用它们。

路由表初始化

路由表是由 *net/ipv4/fib_hash.c* 文件中定义的函数初始化的。IP 路由子系统的初始化函

数 ip_fib_init 调用 fib_hash_init 来创建 ip_fib_main_table 表和 ip_fib_local_table 表（参见第三十二章“路由子系统初始化”一节）。

当首次 fib_hash_init 调用时，它就创建用于分配 fib_node 数据结构的内存池 fn_hash_kmem。

fib_hash_init 首先分配一个 fib_table 数据结构，然后用表 34-1 列出的函数初始化其虚函数。接下来将该结构下半部分（fn_hash，如图 34-1 所示）的内容清空，以便根据子网掩码长度将路由表项分布到不同的 hash 表内。

表 34-1：fib_table 结构中虚函数的初始化

方法	使用的函数
tb_lookup	fn_hash_lookup
tb_insert	fn_hash_insert
tb_delete	fn_hash_delete
tb_flush	fn_hash_flush
tb_select_default	fn_hash_select_default
tb_dump	fn_hash_dump

路由的添加与删除

在第三十六章中，我们将看到用户命令和路由守护进程如何添加、删除、修改路由。这两种方式都是通过内核的路由子系统中的一组程序来实现的。本章我们将看到内核如何从一个路由表中添加和删除路由。如表 34-1 所示，fn_hash_insert 与 fn_hash_delete 用于添加和删除路由，我们将在下面的“添加路由”和“删除路由”一节中对此加以分析。fn_hash_insert 有好几个相关的使用，都与路由的变化有关。

这里给出上述两个函数都用到的一些共同操作：

- 当在路由表中添加或删除一条路由时，构造搜索关键字，用它来查找一个 fib_node 和一个 fib_alias。这些查找与路由数据封包所走的流程类似，但目的不同：这里的目的是为了检查待添加的路由是否已经存在，或检查待删除的路由是否还存在。
- 生成（插入时）及清理（删除时）相应的 hash 表。
- 必要时刷新路由缓存表。

- 生成一条Netlink广播通知，告诉感兴趣的模块：从一个路由表中添加或删除了一条路由（参见第三十二章“Netlink通知”一节）。

添加路由

向路由表中插入一条新的路由是通过fn_hash_insert函数实现的，其逻辑可见图34-3(a)与34-3(b)(注2)。实际上，许多操作要调用这个函数：除了插入新的路由以外，还处理路由的尾部追加(append)、首部追加(prepend)、改变(change)和替换(replace)操作。这些不同的操作是由传入的NLM_F_XXX标识参数来区分的。在第三十六章的表36-1中列出了与每一种操作相关联的标识组合。

不同操作的不同需求使得这个函数的逻辑变得复杂。例如，在“路由hash表的组织”一节中提到，TOS值不同的多个路由可能通向同一个目的地。当内核添加一条新路由时，如果路由表中已经存在目的地与TOS都相同的路由，函数就返回错误。然而，在替换一条路由时也要求满足这个条件。所以，由fn_hash_insert函数所做的路由查找要根据命令类型返回不同的结果。

正如在“基于网络掩码的hash表容量的动态调整”一节中所描述的，插入一条新路由可能触发一个zone的hash表容量的动态调整，这是通过fn_rehash_zone函数实现。正如在“下一跳路由器结构的组织”一节中所描述的，当新路由指定了一个首选源地址时，新的fib_info结构就被添加到fib_info_devhash Hash表内。表示该路由表项下一跳的每一个fib_nh结构也被添加到fib_info_devhash Hash表内。

当一个替换操作用一条新的路由来替换一条已存在的路由时，内核就刷新路由缓存，以便不再使用旧的路由。

不管是哪一种操作类型，都要生成一个Netlink通知来告诉对此感兴趣的子系统。

删除路由

路由的删除由fn_hash_delete函数负责，其逻辑如图34-4所示。删除路由要比添加路由简单，比如，删除路由只有一种操作。

首先，fn_hash_delete确定出搜索关键字，然后用它来查找待删除的表项是否还存在。当查找到fib_alias结构时，就删除它并通过Netlink广播通知感兴趣的子系统。万一该路由表项已经被使用（即设置了FA_S_ACCESSED标识），则需要刷新路由缓存。

注2： 该流程图与源代码并不是完全一致，但逻辑上是一致的。

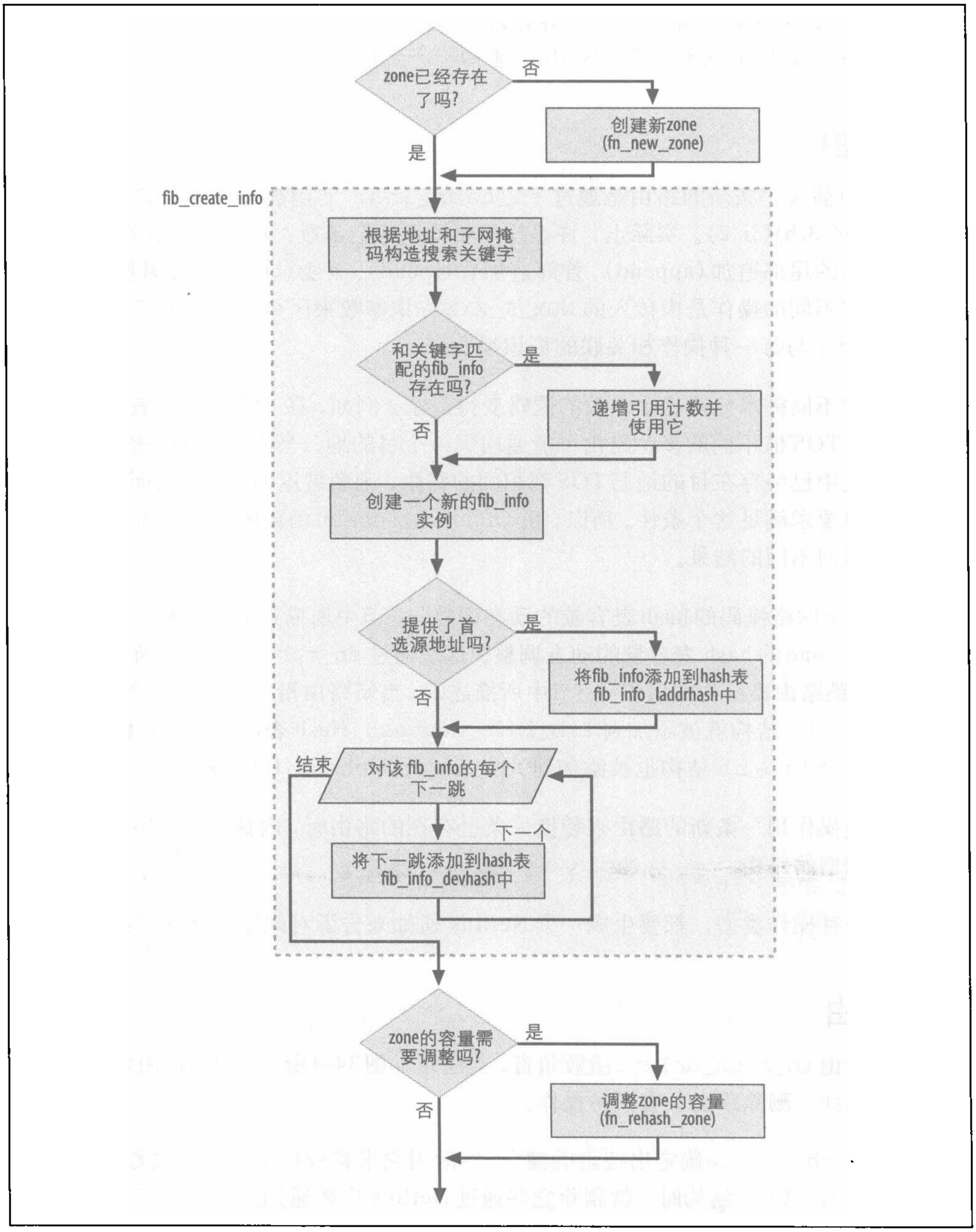


图 34-3 (a)：fn_hash_insert 函数

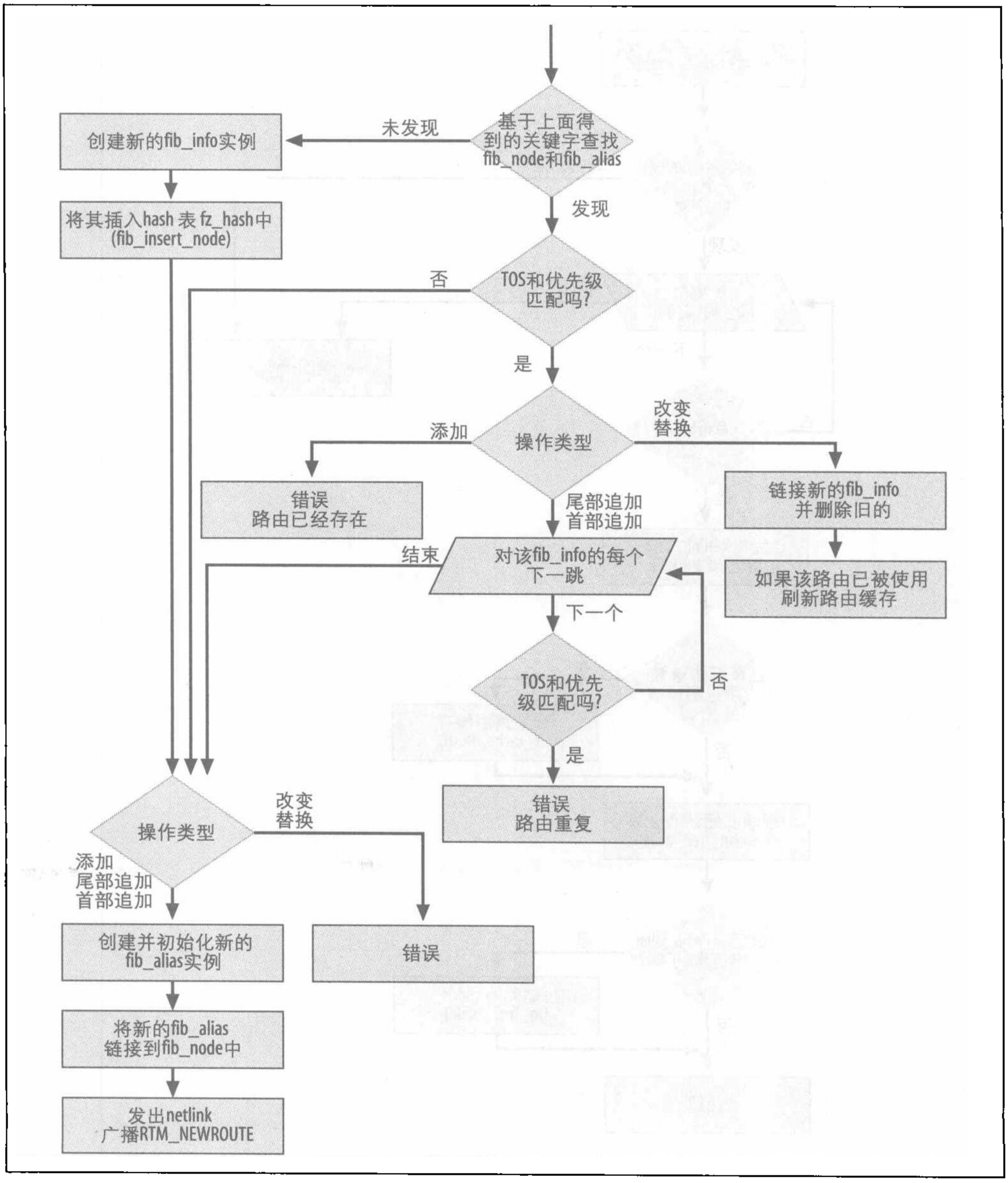


图 34-3 (b)：fn_hash_insert 函数

删除一个 fib_alias 实例可能删除一个 fib_info 实例和一个 fib_node 实例（见图 34-1）：

- 当一个删除的fib_alias为相关联的fib_node实例的最后一个实例时，也删除该 fib_node 实例。

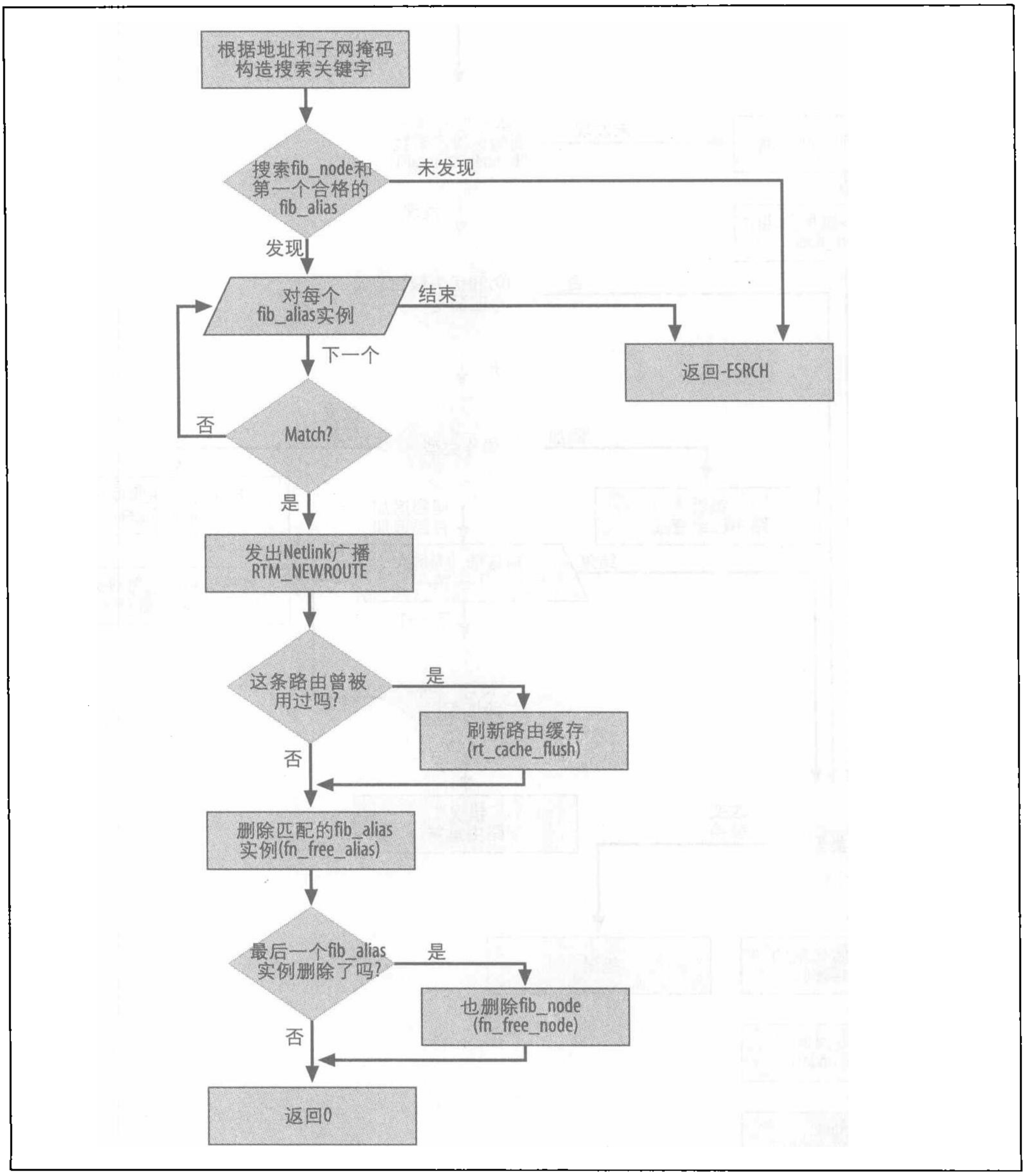


图 34-4：fn_hash_delete 函数

- 当与要删除的路由相关联的 fib_info 实例的引用计数 fib_treeref 为空时，表明不再需要这个实例，因而它要被释放。特别是，fn_free_alias 立刻释放匹配的 fib_alias 实例，并调用 fib_release_info 函数将与之相关的 fib_info 实例中的引用计数 fib_treeref 减 1。当该引用计数降为 0 时，从所有包含 fib_

info实例的hash表中除去该实例，设置fib_dead标识标记它的状态为停用，并在首次调用fib_info_put时用free_fib_info函数释放它。与该fib_info实例相关联的下一跳信息也从hash表中除去，可参见“下一跳路由器结构的组织”一节。

对fa_list和fn_alias链表的操作用fib_hash_lock锁进行保护（见图34-1）。

垃圾回收

由于配置改变或本地设备状态改变而使一些路由变为无效时，它们应当被删除。在路由子系统中提供了多个能够遍历全部路由表或部分路由表的函数。在一定的条件下，其中一个函数，fib_sync_down，设置RTNH_F_DEAD标识来标记满足删除条件的路由项。之后，调用fib_flush再次遍历路由表并删除设置了该标识的路由项。没有周期性的函数来清理路由表，而在路由缓存中有类似的函数。

在第三十二章“常用的辅助程序与宏”一节中有对fib_sync_down 函数的描述。

策略路由及其对路由表定义的影响

若内核编译时配置为支持策略路由，那么管理员可以配置最多255个独立的路由表。为了支持这个可选功能，而又能在不使用策略路由的情况下保持路由子系统程序的精简，Linux开发人员对源代码添加了一些复杂处理，在阅读这些文件之前应当理解这部分代码的复杂性。

变量与结构定义

在支持策略路由情况下，指向255个路由表的指针被存储在fib_tables数组内，该数组在*net/ipv4/fib_frontend.c*文件中定义，如图34-1所示。

```
struct fib_table *fib_tables[RT_TABLE_MAX+1];
```

注意：ip_fib_main_table与ip_fib_local_table这两个路由表分别被定义为fib_tables数组的元素，见*include/net/ip_fib.h*：

```
#ifndef CONFIG_IP_MULTIPLE_TABLES
extern struct fib_table *ip_fib_local_table;
extern struct fib_table *ip_fib_main_table;
.........
#else
#define ip_fib_local_table (fib_tables[RT_TABLE_LOCAL])
#define ip_fib_main_table (fib_tables[RT_TABLE_MAIN])
.........
#endif
```

当向一个新路由表中添加第一条路由时，利用 fib_hash_init 来初始化该表。若没有配置策略路由，该函数只在系统启动时调用，因而它加了 __init 宏标签（注 3）。但是，若配置了策略路由，新路由表可以在任意时刻被创建。所以 fib_hash_init 没有设置 __init 宏标签。这可以解释函数原型定义在不同条件下存在差异的情况：

```
#ifdef CONFIG_IP_MULTIPLE_TABLES
struct fib_table * fib_hash_init(int id)
#else
struct fib_table * __init fib_hash_init(int id)
#endif
{
    .........
}
```

即使在支持策略路由时，除非给配置的路由明确指定了一个路由表的ID，否则所有都被添加到 ip_fib_main_table 表中。路由表 ID 只在新的 *ip* 命令中提供，而在传统的 *route* 命令中则没有提供。

函数的双重定义

策略路由特性并不是完全透明集成到路由代码内。例如，内核要支持策略路由，必须用预处理条件变量 CONFIG_IP_MULTIPLE_TABLES 来保护相应的变量、函数或代码片段（注 4）。

有一些全局变量和函数都有双重定义，一个用于不支持策略路由的内核，另一个用于支持策略路由的内核。其中两个重要的函数是（*在 net/ipv4/fib_rules.c 与 include/net/ip_fib.h* 文件中定义）：

fib_lookup

用于路由表查找，在第三十五章中描述。

fib_select_default

在转发封包时，若没有到达目的地的明确路由，用该函数选择一个默认路由。

除了这两个函数以外，还有其他一些函数也有双重定义，例如，fib_get_table（根据路由表 ID 返回路由表）和 fib_new_table（创建一个新的路由表）。

在阅读源代码时，尤其是利用 TAGS 和 *cscope* 等工具阅读时，知道这些函数存在双重定义是很重要的。否则，在分析代码关系时可能得到错误的实例。

注 3： 第七章描述了 __init 宏的使用和含义。

注 4： 不要将多路由表与多路径混淆，这两个特性是完全不同的、相互独立的。

第三十五章

路由：查找

在第三十三章，我们看到了入口和出口流量是如何触发查找的。总是首先搜索路由缓存，当路由缓存没有查找到时，通过ip_route_input_slow和ip_route_output_slow函数来查找路由表。在本章，我们将分析这些函数，我们尤其将涉及：

- 入口与出口路由有什么样的区别。
- fib_lookup是如何搜索路由表的。
- 策略路由查找与一般查找有什么区别。
- 多路径路由是何时及如何处理的。
- 如何选择一个默认网关。

查找函数的高层概述

不论是哪个方向的流量，都是利用fib_lookup来查找路由表。但是，正如在第三十四章“函数的双重定义”一节中提到的，fib_lookup有两个版本，一个是当内核支持策略路由时使用（*net/ipv4/fib_rules.c*），另一个是当内核不支持策略路由时使用（*include/net/ip_fib.h*）。选择哪一个函数由编译时决定的，所以当ip_route_input_slow和ip_route_output_slow调用fib_lookup时，它们透明地启用正确的函数。

我们简要地看看在路由查找时用到的一些关键函数，在讨论期间，你会发现参考第三十四章中的图34-1是非常有帮助的。

fib_lookup函数是对每一个路由表所提供的一个包含查找函数的包裹函数。当不支持策略路由时，查找函数版本是针对local表和main表。当支持策略路由时，逻辑更为复杂，需要查找由策略路由提供的路由表。

正如图35-1所示，fib_lookup启用查找函数fn_hash_lookup，它正是fib_table的函数指针tb_lookup的初始化函数（参见第三十四章“路由表初始化”一节），这个函数识别相应的fib_node实例，该实例的关键字与目的地址相匹配。然后，fn_hash_lookup调用fib_semantic_match以查找与正在匹配的fib_node相关联的fib_alias实例。如果找到，在多路径的配置情况下fib_semantic_match还需要选择出正确的下一跳。

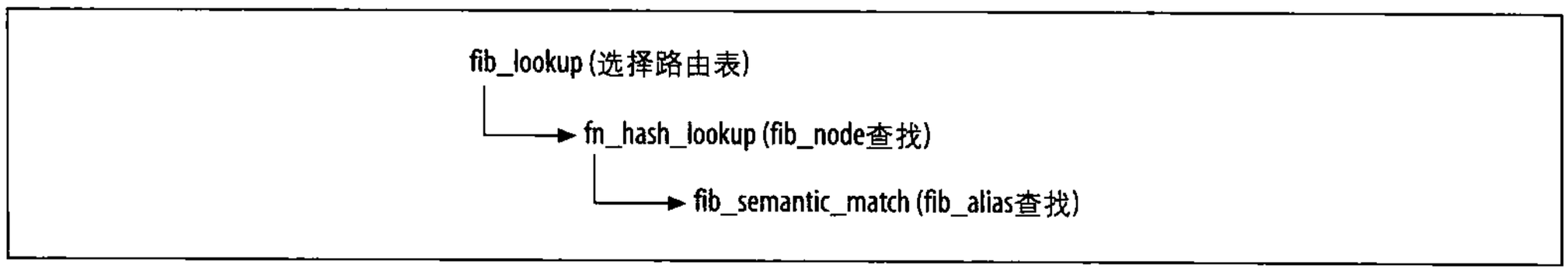


图 35-1：主要的路由查找函数之间的关系

这里介绍的这些函数将在后面的小节中详细描述，主要的内容是：

- 对入口流量和出口流量，如何使用fib_lookup来查找路由（“输入路由”一节和“输出路由”一节）。
- 如何实现fn_hash_lookup函数（“路由表查找：fn_hash_lookup”一节）。
- fib_semantic_match 函数（“语义匹配补充标准”一节）。
- 支持策略路由的fib_lookup版本与基本查找函数之间的区别是什么(“fib_lookup函数”一节)。

辅助函数

下面给出本章中一些函数所调用的函数：

fib_validate_source

对从一个给定设备接收到的封包的源IP地址检验，检测企图的IP欺骗。而且还要在开启非对称路由情况下，确保封包的源IP地址通过该封包接收接口是可达的(参见第三十一章“反向路径过滤”一节)。另外，该函数也返回用于反方向的首选源地址spec_dst，详见下面“首选源地址选择”一节，并初始化路由标签（routing tag)，参见第三十一章“基于路由表的分类器”一节。

inet_select_addr

给定一个设备dev，一个IP地址dst和一个scope，返回为scope的第一个主地址，用于从设备dev向地址dst发送封包。

因为一个设备可能配置了多个地址，而且每个地址有各自的scope，所以需要该函数。

提供dst参数的原因在于，如果在设备dev上配置的不同IP地址属于不同子网，dst允许这个函数返回一个与dst在同一子网的IP地址。

在第三十章“scope”一节中，我们看到每个设备可能配置有多个主IP地址和多个辅地址。inet_select_addr只返回主地址。

如果在dev上配置的地址都不满足由scope和dst限定的条件，该函数则尝试其他设备，检验是否存在一个IP地址，配置有所要求的scope。因为loopback_dev是dev_baselist列表中所插入的第一个设备，所以首先检查的就是它。

rt_set_nexthop

给定一个路由缓存项rtable和一个路由表查找结果res，完成rtable的各字段的初始化，如rt_gateway、嵌入dst_entry结构的metrics向量等。这个函数还对在第三十一章“基于路由表的分类器”一节内描述的路由标签初始化。

路由表查找：fn_hash_lookup

所有的路由表查找，不论路由表是否由策略路由提供，也不论流量方向如何，都是利用fn_hash_lookup来查找。在fib_hash_init（参见第三十四章“路由表初始化”一节）中，fn_hash_lookup被注册为fib_table结构中的tb_lookup函数指针所对应的处理函数。

在第三十章介绍了该函数在查找时所使用的LPM算法，通过为每一个网络掩码组织hash表，使该算法能有效地执行，参见第三十四章中的图34-1。fn_hash_lookup搜索一个能够将封包路由到特定目的地的fib_node实例。

fn_hash_lookup函数的原型为：

```
static int
fn_hash_lookup(struct fib_table *tb, const struct flowi *flp, struct fib_result *res)
```

下面给出各输入参数的含义：

tb

搜索的路由表。因为fn_hash_lookup是一个通用的查找函数，每次只能够查找一张表。调用者依赖是否支持策略路由和相关的因素，来决定搜索哪些路由表。

flp

搜索关键字。

res

查找成功时，用路由信息初始化 res。

下面给出可能的返回值：

0：成功

已经根据转发信息初始化 res（通过 fib_semantic_match 函数）。

1：失败

没有与搜索关键字匹配的路由项。

小于0：管理失败（Administrative failure）

这意味着查找不成功，因为查找到的路由没有价值：例如，相关的主机可能被标记为不可达。

LPM算法从表示最长网络掩码的区域开始循环遍历路由。这是因为网络掩码越长，就意味着路由越具体，也就意味着封包可能得到更接近最终的目的地（例如，一个 /27 的网络掩码只能涵盖 30 台主机，而 /24 的网络掩码中可以涵盖 254 台主机，所以首先检查 /27 的网络掩码）。因而，搜索时遍历所有的活动区域，且从网络掩码最长的区域开始。我们在第三十四章“路由hash表的组织”一节中看到，所有的活动区域按照网络掩码长度排序组成一张列表，fn_zone_list 存储该列表的首部。

```
struct fn_hash *t = (struct fn_hash*)tb->tb_data;
read_lock(&fib_hash_lock);
for (fz = t->fn_zone_list; fz; fz = fz->fz_next) {
    struct hlist_head *head;
    struct hlist_node *node;
    struct fib_node *f;
```

在 fn_hash_lookup 函数中，将目的IP 地址与被检查的活动区域的网络掩码相与(AND)，操作的结果作为搜索关键字。例如，如果正在检查 /24 区域，目的地址 flp->fl4_dst 为10.0.1.2，则搜索关键字 k 为10.0.1.2 & 255.255.255.0，结果为10.0.1.0。这意味着接下来的代码要搜索到子网 10.0.1.0/24 的路由：

```
u32 k = fz_key(flp->fl4_dst, fz);
```

因为路由被存储在 hash 表（fz_hash）内，通过对关键子 k 应用一个 hash 函数，head 选择表的一个正确的 bucket。下一步是遍历与该 bucket 相关联的路由列表（fib_node 结构），查找与 k 匹配的路由项。

```
head = &fz->fz_hash[fn_hash(k, fz)];
hlish_for_each_entry(f, node, head, fn_hash) {
    if (f->fn_key != k)) {
        continue;
```

```
                err = fib_semantic_match(&f->fn_alias,
                                         flp, res,
                                         f->fn_key, fz->fz_mask,
                                         fz->fz_order);

                if (err < 0)
                    goto out;
            }
        }
        err = 1;
out:
        read_unlock(&fib_hash_lock);
        return err;
}
```

我们在第三十四章“路由hash表的组织”一节中看到，一个fib_node包括了同一子网内的所有路由项，但这些路由项在诸如TOS等其他字段上可能不同。现在，如果fn_hash_lookup函数找到了与搜索关键字k匹配的fib_node，它还需要检查每一个潜在的路由项，来查找与输入参数flp中其他搜索字段相匹配的路由。这个详细检查是由fib_semantic_match来完成的，这将在下一小节描述。

fib_semantic_match也用查找结果初始化输入参数res，如果它返回成功，fn_hash_lookup就将该结果返回给调用者。fn_hash_lookup遍历路由表中所有的活动区， 直到fib_semantic_match或者返回一个成功结果，或者发现该路由表中没有可用的路由（即这些路由项都不匹配）。

语义匹配补充标准（Subsidiary Criteria）

fib_semantic_match被调用以发现在与给定的fib_node相关的路由（fib_alias结构）中，是否存在与搜索关键字所有字段都匹配的路由项。我们在前一小节看到主要的字段，即封包被路由到的目的IP地址，在调用fib_semantic_match之前已经在fn_hash_lookup中查找到匹配项。现在由fib_semantic_match来检查其他字段是否匹配。

一旦fib_semantic_match查找到相匹配的fib_alias实例，就从相关联的fib_node中提提取出路由信息。需要进行的额外工作就是选择下一跳，只有当匹配路由项使用多路径时才进行最后一项工作，可以用两种方法来处理：

- 当搜索关键字中有出口设备时由fib_semantic_match来处理。
- 当搜索关键字中没有出口设备时由fib_select_multipath来处理，fib_select_multipath被ip_route_input_slow或ip_route_output_slow函数调用。

fib_semantic_match的逻辑参见图35-2所示。

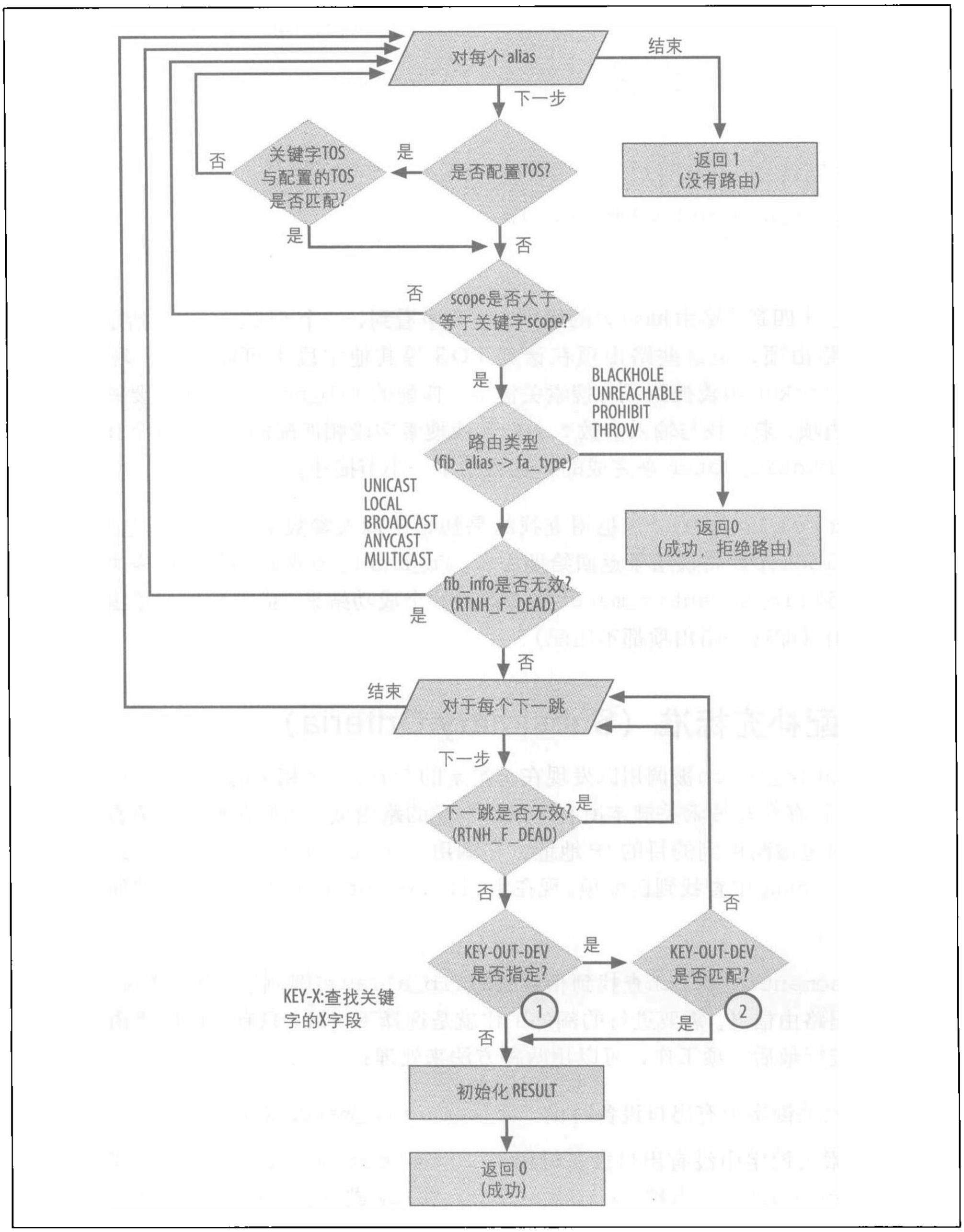


图 35-2：fib_semantic_match 函数

拒绝路由的标准

当遍历 fib_alias 结构时，fib_semantic_match 拒绝以下路由：

- TOS 不匹配。注意，当路由项没有配置 TOS 值时，可以路由任意 TOS 值的封包。
- scope 比搜索关键字中指定的更窄的路由项。例如，如果路由子系统查找 scope 为 RT_SCOPE_UNIVERSE 的路由，则不能使用 scope 为 RT_SCOPE_LINK 的路由项。

而且，该函数必须检查路由项或所期望的下一跳是否无效，在这种情况下，路由子系统已经设置 RTNH_F_DEAD 标识以标记该路由项应当被删除。在第三十二章“辅助程序”一节中，描述了如何对一个路由或路由的下一跳来设置 RTNH_F_DEAD 标志。

一旦标识出一个合法的 fib_alias 实例，并假定相关联的 fib_info 结构可用（即没有标记 RTNH_F_DEAD 标识），如果搜索关键字中给定了设备，则 fib_semantic_match 必须遍历所有下一跳的 fib_nh 实例，以查找与搜索关键字的设备相匹配的实例。但实际中可能没有一个下一跳能被使用，这可能是由于两种原因引起的：

- 所有的下一跳都不可用（即它们都设置了 RTNH_F_DEAD 标识）。
- 搜索关键字指定的出口设备与任何一个下一跳的配置都不匹配。

当不支持多路径时，则只能有一个下一跳。

当遍历 fib_alias 实例时，fib_semantic_match 对满足本小节前面提到的 scope 和 TOS 需求的实例设置 FA_S_ACCESSED 标识。该标识的设置与 fib_alias 是否被选中无关。当 fib_alias 实例被删除时，根据该标识来决定是否应当刷新缓存。

fib_semantic_match 函数的返回值

前面提到，fib_semantic_match 的返回值可能为以下三个含义之一：

- 1 意味着没有匹配的路由项。
- 0 意味着成功。在这种情况下，查找结果被存储在输入参数 res 中，这个查找结果包含一个指向匹配的 fib_info 实例的指针。
- 负值表示管理性失效。

fib_semantic_match 返回 0 值和负值二者都是由查找到的匹配路由（fa）的类型（fa->fa_type）决定的。路由类型值的例子有 RTN_UNICAST 和 RTN_LOCAL。fib_semantic_match 从这些类型中可以判断查找是成功还是失败，当失败时传回一个错误编码以使内核对这种失败采取相应的动作。

例如，一个RTN_UNREACHABLE的路由类型，引起fib_semantic_match返回错误编码EHOSTUNREACH，将导致内核产生一个ICMP不可达消息。一个RTN_THROW路由类型，引起fib_semantic_match返回错误编码-EAGAIN，将指示在*net/ipv4/fib_rules.c*文件中fib_lookup的策略路由版本尝试查找下一张路由表。

根据fa->fa_type类型字段得出返回值，返回的错误编码加入到fib_props数组中，该数组的定义和初始化在*net/ipv4/fib_semantics.c*文件中完成（参见第三十六章“rtable结构”一节）。该数组的每一个元素对应一种可能的路由类型，每个元素包含一个相关联的错误编码和一个scope是RT_SCOPE_*XXX*。以fa->fa_type为索引，就可以从数组fib_props得出该路由类型对应的错误编码和scope。

表35-1给出了如何初始化fib_props数组。

表35-1：fib_props的初始化

路由类型	错误	scope
RTN_UNSPEC	0	RT_SCOPE_NOWHERE
RTN_UNICAST	0	RT_SCOPE_UNIVERSE
RTN_LOCAL	0	RT_SCOPE_HOST
RTN_BROADCAST	0	RT_SCOPE_LINK
RTN_ANYCAST	0	RT_SCOPE_LINK
RTN_MULTICAST	0	RT_SCOPE_UNIVERSE
RTN_BLACKHOLE	-EINVAL	RT_SCOPE_UNIVERSE
RTN_UNREACHABLE	-EHOSTUNREACH	RT_SCOPE_UNIVERSE
RTN_PROHIBIT	-EACCES	RT_SCOPE_UNIVERSE
RTN_THROW	-EAGAIN	RT_SCOPE_UNIVERSE
RTN_NAT	-EAGAIN	RT_SCOPE_NOWHERE
RTN_XRESOLVE	-EINVAL	RT_SCOPE_NOWHERE

注意，前面几个元素的错误编码error为0，在这些情况下fib_semantic_match返回成功。其他的错误编码被路由代码用于正确地处理路由失败。

fib_lookup函数

在第三十章“特殊路由”一节中提到，当不支持策略路由时内核默认使用两张路由表。

对路由表的查找就是对这两张表的查找（调用两次fn_hash_lookup），所以在*include/net/ip_fib.h*文件中定义的fib_lookup函数就很简单：

```
static inline int fib_lookup(const struct flowi *flp, struct fib_result *res)
{
        if (ip_fib_local_table->tb_lookup(ip_fib_local_table, flp, res) &&
            ip_fib_main_table->tb_lookup(ip_fib_main_table, flp, res))
                return -ENETUNREACH;
        return 0;
}
```

搜索的关键字是flp。函数首先检查ip_fib_local_table路由表，如果失败则检查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）。

为接收和发送设置函数

接收到的封包和本地产生的封包二者都需要被路由：在接收到封包的情况下，要查找是否应当送交本地还是转发；在本地产生成封包的情况下，要查找是否应当送交本地还是发送出去。

在这两种情况下，给定一个需要被路由的封包skb，路由查找结果被保存在skb->dst中。该数据结构的类型为dst_entry，这在第三十六章“dst_entry结构”一节中有详细描述。该数据结构包含多个字段：其中两个字段名为input和output，它们是根据路由查找结果来处理封包的函数指针。下一小节将详细描述这两个函数指针的初始化。

接下来，在“输入路由”和“输出路由”一节中，将详细描述ip_route_input_slow和ip_route_output_slow函数。这两个函数是在缓存查找失败后，分别针对入口封包和出口封包来继续查找路由。由于这两个函数代码都比较长且使用了大量的宏、条件代码（如#ifdef），以及对某些特性的处理（例如，多路径），这可能令人有一些恐惧，但实际上要比看上去的简单。而且，那些不喜欢在源代码中使用goto语句的程序员可能由于goto语句的大量使用而感到失望。我已经对每个函数都画出流程图，在阅读这些函数之前你可能想先看看这些高层描述（high-level descriptions）。

在本小节的其余部分，我们将看到这两个虚拟函数每次被调用（dst->input和dst->output）如何初始化，并更多了解它们何时被调用。这两个函数被设为什么依赖于以下一些要素：

- 封包是正在被传送、接收还是转发。
- 目的地址是单播地址还是多播地址。

- 在路由查找期间是否检测到错误。

在表 35-2 和表 35-3 中，列出了 dst->input 和 dst->output 可以被初始化的例程。

表 35-2：dst->input 使用的函数

函数	描述
ip_local_deliver	向本地传递的封包，参见第二十章。
ip_forward	转发一个单播封包，参见第二十章。
ip_mr_input	转发一个多播封包。
ip_error	处理一个不可达目的，参见“路由失败”一节。
dst_discard_in	简单地丢弃任何输入封包。

表 35-3：dst->output 使用的函数

函数	描述
ip_output	包含 ip_finish_ouput 的包裹函数，参见第二十一章。
ip_mc_output	处理带有多播地址的出口封包。
ip_rt_bug	因为不支持调用，打印一个警告消息。
dst_discard_out	简单地丢弃任何输入封包。

在表 35-2 和表 35-3 中，并不是所有的函数都可以组合。图 35-3 总结了有意义的一些组合。这些组合不包括 dst_discard_xxx 程序，因为它们只在与路由查找无关的特殊情况下才出现（参见“特殊情况”一节）。

图 35-3 显示了对入口流量和出口流量来说，如何初始化 dst->input 和 dst->output。让我们一个一个地看。

初始化入口流量的函数指针

我们在第十九章看到入口 IP 流量是由 ip_rcv_finish 处理。这个函数参照路由表来决定将封包送往本地还是丢弃。决策是由 ip_route_input 函数来完成，它首先检查路由缓存，当缓存不命中（miss）时检查路由表（ip_route_input_slow）。ip_route_input_slow 可以创建 dst->input 和 dst->ouput 的三种主要的组合：

- 如果封包被转发，函数将 dst->input 初始化为 ip_forward，将 dst->output 初始化为 ip_output。所以 dst_input 将调用 ip_forward，而在 ip_forward 的结尾处间接调用 dst_output，即 ip_output。这是图 35-3 中的情况①。

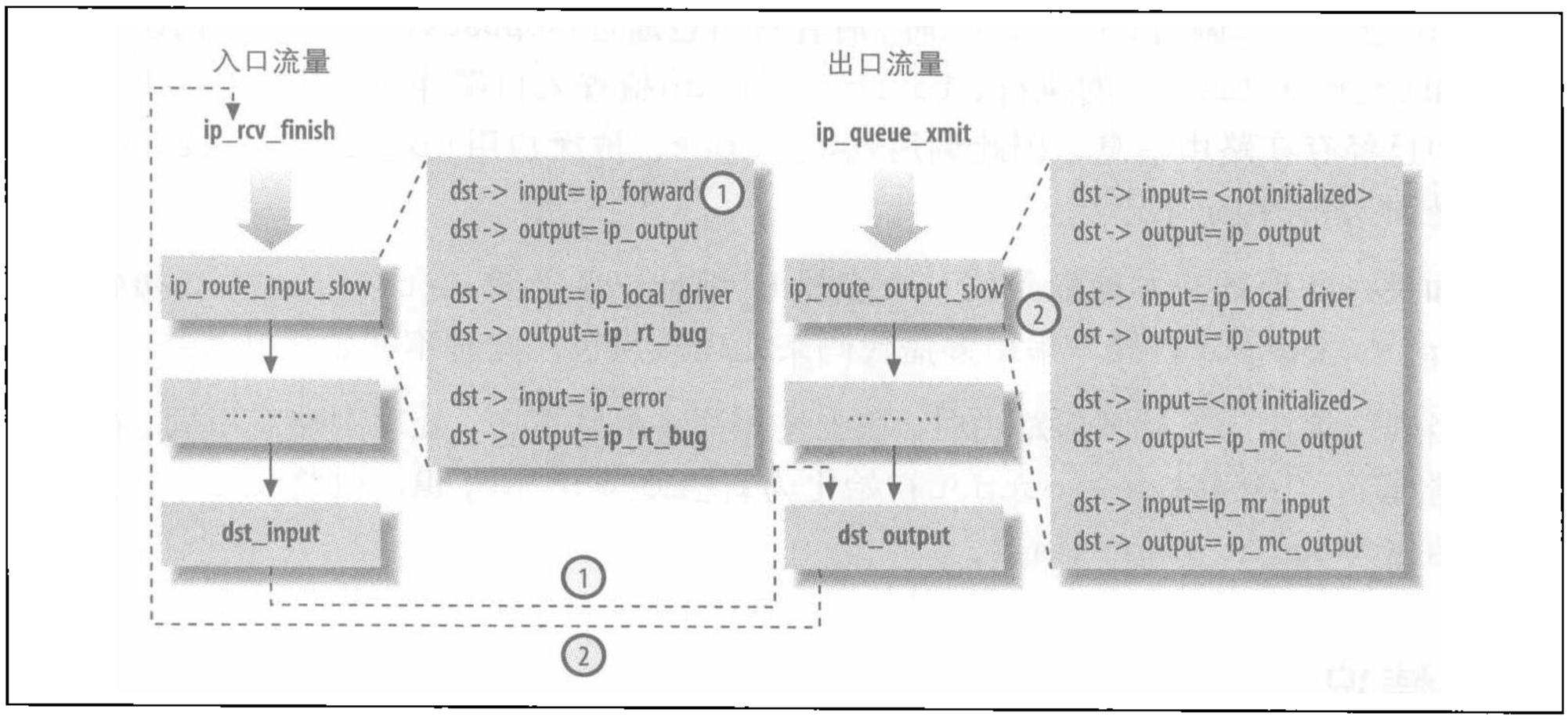


图 35-3：dst->input 和 dst->output 的初始化

- 如果封包被送往本地，函数将dst->input初始化为ip_local_deliver。此时不需要初始化dst->output，但它还是被初始化为ip_rt_error，这是一个被调用时打印出一条错误消息的函数。这样做有助于检测在处理送往本地封包时dst->output是否被错误调用。
- 如果根据路由表得出目的地址不可达，dst->input被初始化为ip_error，这将产生一个ICMP消息，消息类型依赖于路由查找返回的准确结果。因为ip_error将skb缓冲区释放，所以不需要初始化dst->output，因为即使在犯错情况下它也不会被调用。

初始化出口流量的函数指针

我们在第二十一章看到在IP层有多个不同的发送函数。图35-3使用ip_queue_xmit作为一个例子，但是不管该函数在什么情况下被启用，最终结果都是用__ip_route_output_key来查找路由，当缓存不命中时依赖ip_route_output_slow。ip_route_output_slow函数可以创建dst->input和dst->ouput的四种主要的组合：

- 如果目的地是远程主机，函数将dst->output初始化为ip_output。这里不需要初始化dst->input。但是，我们可以用一个类似ip_rt_error的函数来伪初始化，以便捕获bug，这种做法我们在“初始化入口流量的函数指针”一节中已经看到。
- 如果目的地是本地系统，函数将dst->output初始化为ip_output，将dst->input初始化为ip_local_deliver。这是一个可以形成循环的有趣组合。当

dst_output调用ip_output时，后者将封包通过loopback设备送出，导致ip_rcv和ip_rcv_finish的执行。ip_rcv_finish检查入口缓冲区，看到在skb->dst中已经存在路由信息，因此调用 dst_input，依次启用 ip_local_deliver。这是图 35-3 中的情况②。

- 如果目的地址是本地配置的一个多播 IP 地址，函数将 dst->output 初始化为 ip_mc_output，接下来由多播代码来处理该封包。此时不初始化 dst->input。
- 当内核编译为支持多播路由时，在上述相同多播情况下，处理则略有不同。在这种情况下，仍然将dst->output初始化为 ip_mc_output，但同时将dst->input初始化为 ip_mr_input 函数。

特殊情况

当一个缓存路由 dst 不能被使用时，dst->output 被初始化为 dst_discard_out，dst->input被初始化为dst_discard_in。这两个函数都是简单地将传入的封包丢弃。这样初始化的一个例子就是，当一条缓存路由应当被删除，但由于引用计数不为0还不能够被销毁时（参见第三十三章“删除 DST 表项”一节）。

当一条新路由表项已经分配空间，但还没有准备好被使用时，也是用上述两个程序来初始化，因为该路由项此时还没有完全被初始化（参见 dst_alloc）。

输入和输出路由函数的一般结构

我们在第三十三章“缓存查找”一节中看到，当入口路由和出口路由通过缓存而没有查找到路由时，分别调用 ip_route_input_slow 和 ip_route_output_slow。

这两个函数都很长。为了使它们更具有可读性，部分代码被移到两个内联函数（注1）内，分别是 ip_mkroute_input 和 ip_mkroute_output。这两个内联函数在内核支持多路径缓存特性和不支持的两种情况下，处理有所不同。当内核不支持多路径缓存时，这两个内联函数分别成为 ip_mkroute_input_def 和 ip_mkroute_output_def 的别名。无论是否支持多路径缓存，路由缓存表项的分配和初始化都是由 __mkroute_ input 和 __mkroute_output 来完成。无论 ip_route_input_slow 或 ip_route_output_slow触发将新的路由表项插入到缓存内的操作时，都是调用rt_intern_hash函数来执行。

注 1： 注意因为它们是内联函数，所以可以使用goto语句跳转到各自slow函数中定义的label。

图 35-4 总结了本小节提到的内容，并且给出了这两个 slow 函数的流程架构，它们看起来是多么对称。

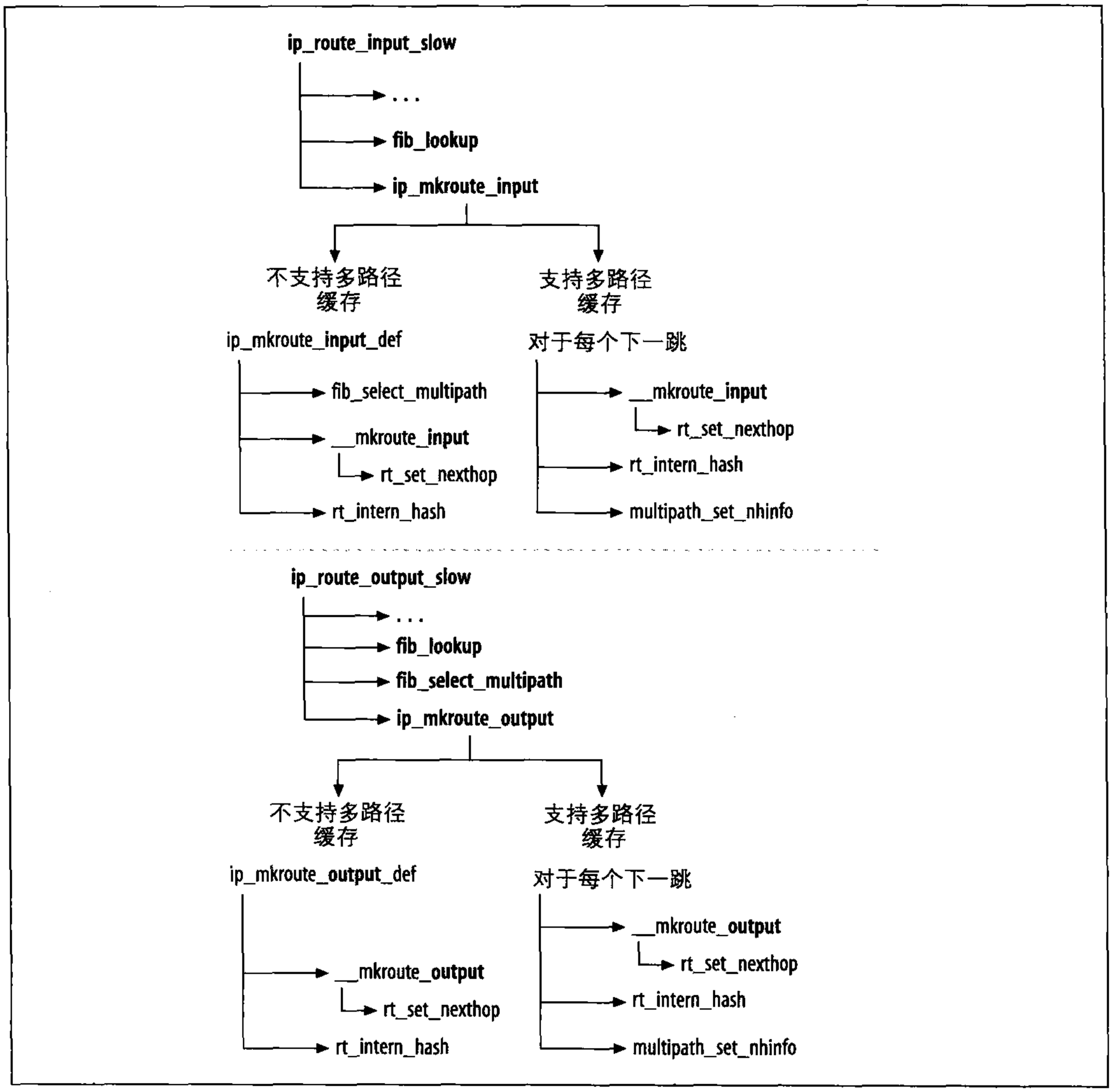


图 35-4：ip_route_input_slow 和 ip_route_output_slow 的架构

对于多路径时存在一些差异，像在 ip_mkroute_input_def 调用 fib_select_multipath，而在 ip_mkroute_output_def 中没有，其中的原因将在“多路径缓存”一节内解释。

输入路由

对还没有路由的入口 IP 封包在缓存中利用 ip_route_input 查找路由，而检查路由表是利用 ip_route_input_slow，该函数在 *net/ipv4/route.c* 文件中定义，其逻辑流程见图 35-5(a)和 35-5(b)。在这一小节，我们详细描述这个函数的内部细节。

这个函数首先对源地址和目的地址进行了一些合理性检查，例如，源 IP 地址一定不能为多播地址。我在第三十一章的“Verbose 监控”一节中已经列出了大部分的检查。下面还要进行更多的合理性检查。

查找路由表是由 fib_lookup 来完成，在“fib_lookup 函数”一节中已经介绍过这个函数。如果 fib_lookup 没有查找到一条匹配的路由，则封包被丢弃。而且，如果接收接口被配置为开启转发，则向源地址回送一条 ICMP_UNREACHABLE 消息。注意，该 ICMP 消息不是由 ip_route_input_slow 送出的，而是由调用者送出，调用者看到返回值为 RTN_UNREACHABLE 时送出该 ICMP 消息。

在成功的情况下，ip_route_input_slow 需要区分以下三种情况：

- 封包目的地址为一个广播地址。
- 封包目的地址为一个本地地址。
- 封包目的地址为一个远端地址。

在前两种情况下封包被送往本地，而在第三种情况下封包需要被转发。如何将封包送给本地及如何转发封包的细节可以在“送往本地”和“转发”一节中找到。这里给出无论是送往本地还是转发都需要处理的一些任务：

合理性检查，特别是源地址

再次检查源地址是否为非法值，并且通过 fib_validate_source 检查欺骗企图。

创建和初始化一条新缓存表项（局部变量 *rth*）。

参见下面的“创建一个缓存表项”一节。

创建一个缓存表项

我在第三十三章“缓存查找”一节中曾经说过，只要是查找路由表就可以调用 ip_route_ input（因此当缓存不命中时将调用 ip_route_input_slow），而没有必要路由一个入口封包。所以 ip_route_input_slow 并不总是需要创建新的缓存表项。当从 IP 或 L4 协议（例如，IP over IP）启用时，则总是创建一个缓存表项。当前，其他

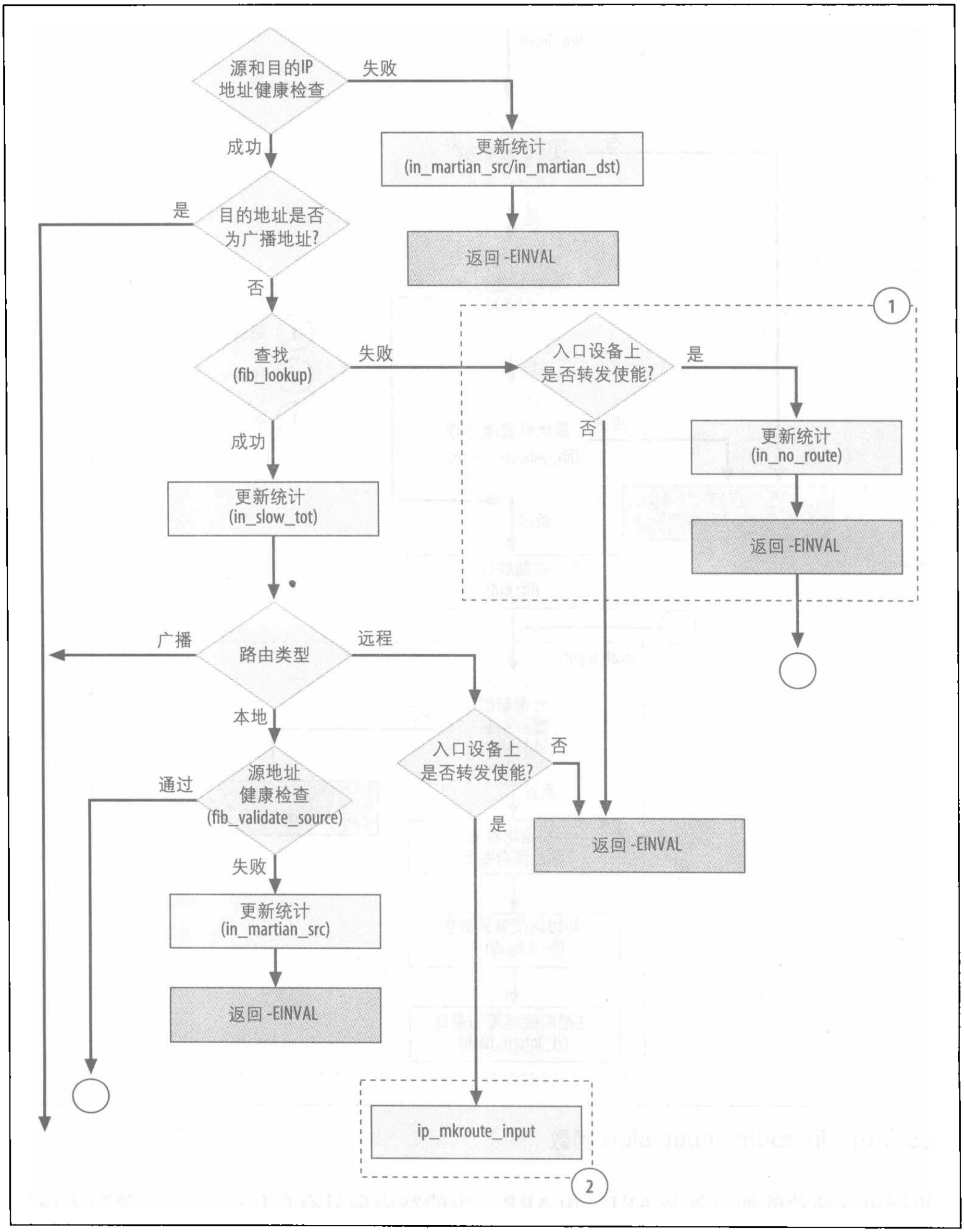


图 35-5(a)：ip_route_input_slow 函数

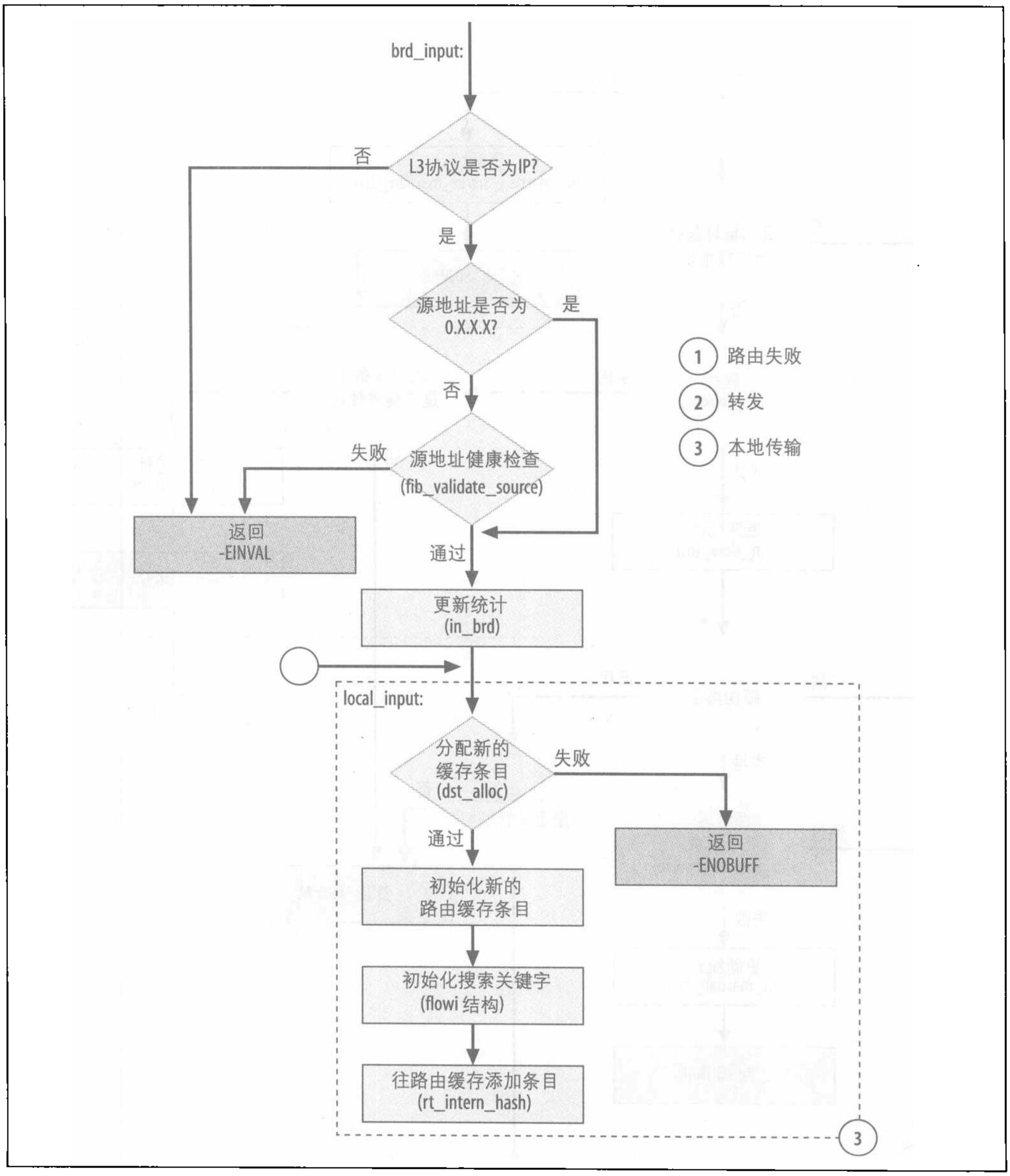


图 35-5(b)：ip_route_input_slow 函数

可能调用该函数的地方就是 ARP。由 ARP 产生的路由项只有在代理 ARP 有效时才被缓存，参见第二十八章“处理 ARPOP_REQUEST 封包”一节。

dst_alloc 为新缓存表项分配空间，特别重要的是对新缓存表项进行下面的初始化：

`rth->u.dst.input`

`rth->u.dst.output`

这两个虚拟函数分别被dst_input和dst_output启用，来完成入口封包和出口封包的处理，参见第十八章中的图18-1。我们也在“为接收和发送设置函数”一节中看到，根据封包是否应当被转发、送往本地或丢弃等情况，来决定如何初始化这两个函数。

`rth->fl`

这个flowi结构被用作缓存查找的搜索关键字。重要的是需要注意rth->fl的字段是利用ip_route_input_slow的输入参数来初始化的：这样做可以确保当下一次利用相同参数来查找时，ip_route_input查找缓存就可以得到满意的结果。

`rth->rt_spec_dst`

这是首选源地址。参见下面的“首选源地址选择”一节。

首选源地址选择

添加到路由缓存的路由项是单向的，意味着将用于被路由封包源IP地址的反方向上的流量，不能使用该路由项来进行路由。然而在某些情况下，接收到封包可能触发一个动作，要求本地主机选择一个源IP地址，以便在向发送方回送封包时使用（注2）。这个地址，即首选（preferred）源IP地址（注3），必须与路由该入口封包的路由缓存表项保存在一起。首选源IP地址被保存在rt_spec_dst字段内，以后会有用处，下面是使用该地址的两种情况：

ICMP

当一个主机接收到一个ICMP ECHO REQUEST消息时（众所周知的ping命令通常产生这种消息），如果主机没有明确配置为不作出回应，则该主机返回一个ICMP ECHO REPLY消息。对该入口ICMP ECHO REQUEST消息选择路由，路由项的rt_spec_dst被用作路由ICMP ECHO REPLY消息而进行路由查找的源地址。参见*net/ipv4/icmp.c*文件中的icmp_reply及第二十五章。在*net/ipv4/ip_output.c*文件中的ip_send_reply函数做法与之类似。

IP选项

一些IP选项要求源地址和目的地址中间的主机将它们接收该封包接口的IP地址写

注2：首选源IP地址在用于本地产生的流量时（即向本地发送封包不是由于收到另一个封包而触发或受到影响）可能不同。参见“选择源IP地址”一节。

注3：RFC 1122称之为“特殊目的地”。

入IP包头内，Linux写入的地址是rt_spec_dst。参见第十九章中对ip_options_compile 的描述。

首选源地址是通过在“辅助函数”一节中提到的fib_validate_source函数来选择的，并且被 ip_route_input_slow所调用。

ip_route_input_slow根据被路由封包的目的地址初始化首选源IP地址rt_spec_dst：

封包目的地址为本地地址

在这种情况下，封包被送往的本地地址为首选源地址（前面提到的ICMP例子就是这种情况）。

广播封包

广播地址不能被用作出口封包的源地址。因此在这种情况下，ip_route_input_slow需要另外两个函数的协助：inet_select_addr和fib_validate_source（参见“辅助函数”一节）。

当接收封包内没有设置源IP地址（即为全零）时，inet_select_addr在接收该封包的设备上选择scope为RT_SCOPE_LINK的第一个地址。这是因为封包源地址为空时，目的地址为受限广播地址，而广播地址scope为RT_SCOPE_LINK。其中一个例子就是DHCP发现消息。

当源地址不是全零时，由 fib_validate_source 负责处理。

被转发的封包

这种情况下，由fib_validate_source来处理。（前面提到的IP选项例子就是这种情况）。

用户可以使用下面的类似命令，配置一条具体路由的首选源IP，当有封包匹配该路由项时使用配置的首选源 IP：

```
ip route add 10.0.1.0/24 via 10.0.0.1 src 10.0.3.100
```

在这个例子中，当向10.0.1.0/24子网的主机发送封包时，内核将使用10.0.3.100作为源IP 地址。当然，只有本地配置的地址才能生效：这意味着要使前面的命令生效，10.0.3.100就必须已经在本地某个接口上被配置，不需要一定是在到达10.0.1.0/24子网的设备上配置。（记住在Linux中，地址属于主机而不是属于设备。参见第二十八章“多个网络接口的应答”一节）。当管理员不想使用根据出口设备默认选择的源地址时，管理员通常可以提供一个源地址。

图 35-6 总结了如何选择 rt_spec_dst。

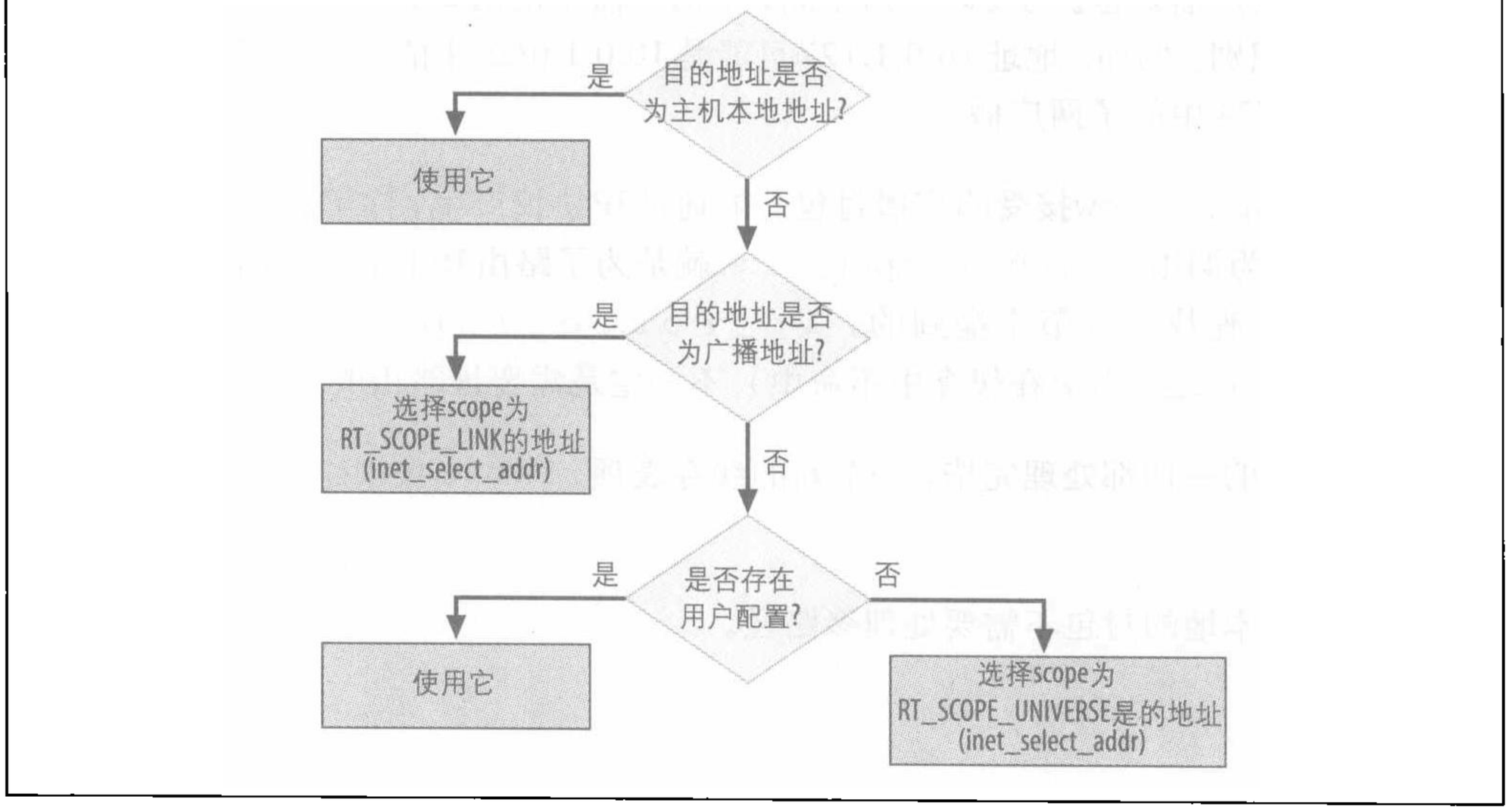


图 35-6：rt_spec_dst 的选择

送往本地

我们在“初始化入口流量的函数指针”一节中已经看到，通过正确地初始化 `dst->input`，以下类型的封包被送往本地：

- 封包目的地为本地配置的地址，包括多播地址。
- 封包目的地为广播地址。

`ip_route_input_slow` 识别两种广播：

受限广播（Limited broadcasts）

这是一个由全 1：255.255.255.255（注 4）组成的地址。不调用 fib_lookup 也可以很容易识别出来，受限广播被传递给链路上的所有主机，与主机所配置的子网无关，不需要查找路由表。

子网广播

这些广播被定向传送到配置有特定子网的所有主机。如果主机配置为通过同一个设备可以到达不同的子网 [参见第三十章中的图 30-4(c)]，则只有正确的子网才能够

注 4：已废弃的一种受限广播由全零组成：0.0.0.0。

接收子网广播封包。与受限广播不同，子网广播不使用fib_lookup查找路由表就不能够识别。例如，地址10.0.1.127可能是10.0.1.0/25中的一个子网广播，而不是10.0.1.0/24中的子网广播。

ip_route_input_slow接受的广播封包只能通过IP协议产生，你可能认为这是一项多余的检查，因为调用ip_route_input_slow就是为了路由IP封包。然而，就像在第三十三章“缓存查找”一节中提到的，ip_route_input的输入缓冲区（当ip_route_input_ slow在缓存中不命中）不一定是需要被路由的封包。

如果上述所有的一切都处理完毕，一个新的缓存表项rtable被创建、初始化并插入到路由缓存内。

注意，对送往本地的封包不需要处理多路径。

转发

如果封包需要被转发，但入口设备配置为禁止转发，则该封包不能被发送而必须被丢弃。IN_DEV_FORWARD可以用来检查设备的转发状态。图35-7显示了ip_mkroute_input的内部细节，特别是，显示了当不支持多路径缓存时函数的外部特征（即在ip_mkroute_input的结尾处调用了ip_mkroute_input_def）。在“多路径缓存”一节中，你将看到这两种情况的区别。

如果fib_lookup返回的匹配路由项包括多个下一跳，调用fib_select_multipath 以选择其中之一。当支持多路径缓存特性时，有多种不同的选择方法。在“多路径对下一跳选择的影响”一节中描述了选择算法。

调用fib_validate_source来验证源地址。然后根据我们在第三十一章“发送ICMP_REDIRECT消息”一节中看到的各种因素，内核可能决定向源发送一个ICMP_REDIRECT消息。在这种情况下，不是直接通过ip_route_input_slow发送该ICMP消息，而是当ip_forward看到RTCF_DOREDIRECT标识后，发送该ICMP消息。

正如我们在“创建一个缓存表项”一节中看到的，路由查找结果并不总是需要被缓存。

路由失败

当一个封包由于主机配置或是由于没有路由匹配而不能被路由时，一个新的路由项就被添加到缓存内，该路由的dst->input被初始化为ip_error。这意味着所有匹配该路由的入口封包将由ip_error来处理。当ip_error被dst_input启用时，将根据封包不

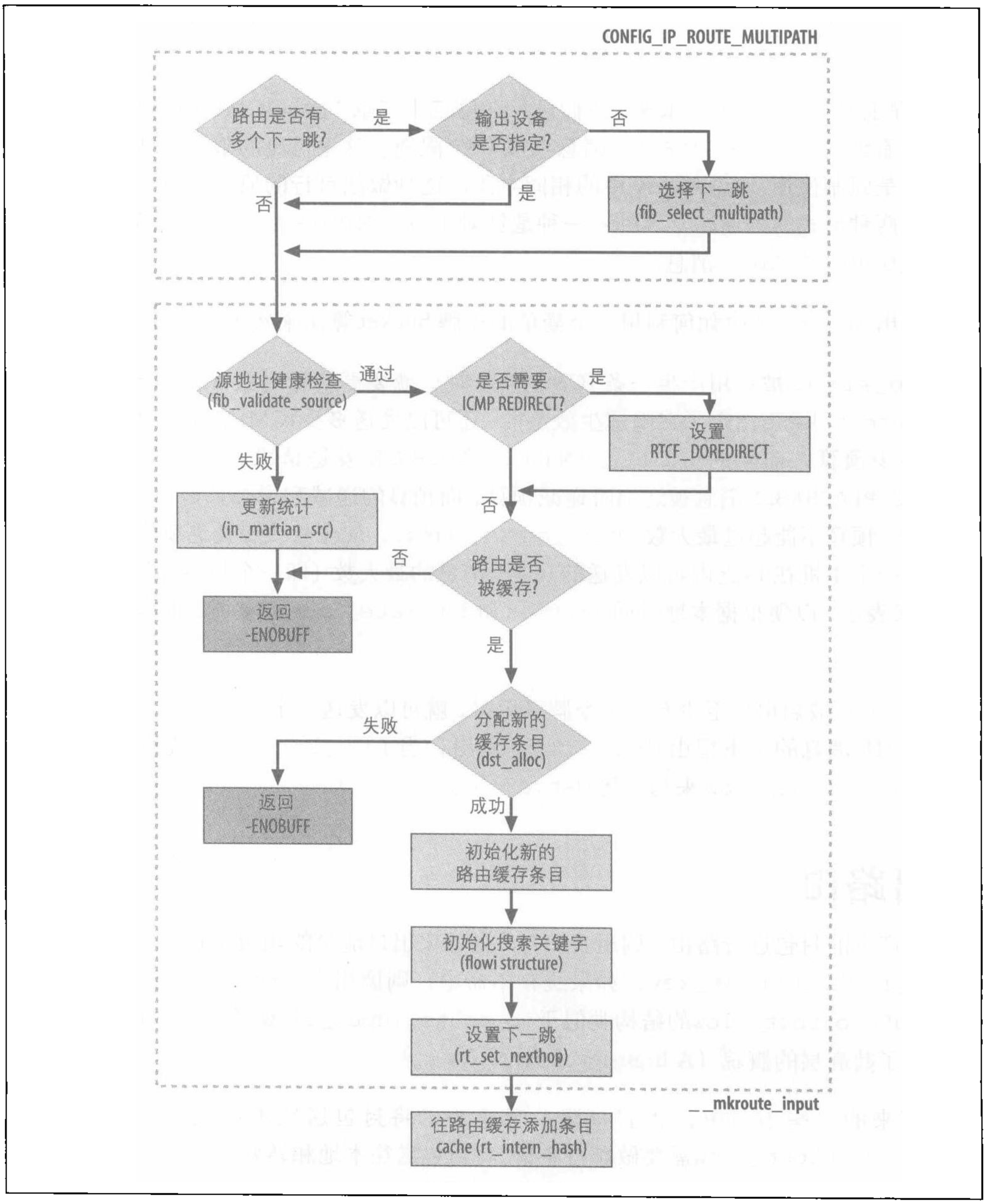


图 35-7：ip_mkroute_input 函数

能被路由的原因产生相应的ICMP_UNREACHABLE消息，然后丢弃该封包。将错误路由添

加到缓存内是有用的，因为这样可以加速对后续发送给同样不正确地址的封包错误进行处理。

ICMP消息通过ip_error限速。我们已经在第三十三章“出口ICMP REDIRECT限速”一节中看到，ICMP_REDIRECT消息也被DST限速。这里讨论的限速与其他的情况无关，只是强制使用dst_entry中的相同字段。这种做法可行的原因是对任意给定的路由，这两种形式的限速是互斥的：一种是针对ICMP_REDIRECT消息，另一种是针对ICMP_UNREACHABLE消息。

下面给出ip_error中如何利用一个简单的令牌bucket算法来实现限速。

每次ip_error被启用产生一条ICMP消息时，就要更新时间戳dst.rate_last。dst.rate_tokens指定了在限速生效之前，还可以发送多少ICMP消息，也即令牌的数目或多少预算，而新的ICMP_UNREACHABLE发送请求将被忽略。每次一个ICMP_UNREACHABLE消息被送出时递减预算，而预算的递减和增加都是在ip_error中实现的。预算不能超过最大数ip_rt_error_burst，就像从该变量名所看到的，该变量表示一个主机在1s之内可以发送的ICMP消息的最大数（即一个burst值）。这个数值用Hz来表示，以便根据本地时间jiffies和dst.rate_last的差值，很容易地添加到令牌。

当ip_error被启用且至少有一个令牌可用时，就可以发送一个ICMP_UNREACHABLE消息。ICMP消息的子类型由dst.error得出，当fib_lookup查找路由失败时由ip_route_input_slow来初始化dst.error。

输出路由

对本地产生的封包进行路由，调用我们在“初始化出口流量的函数指针”一节中介绍的__ip_route_output_key，如果缓存不命中，则调用ip_route_output_slow。ip_route_output_slow的结构类似于ip_route_input_slow，在图35-8(a)和35-8(b)中给出了其高层的概貌（A high-level overview）。

在接下来的一些小节中，我们将深入考察为了将封包送往本地或将它发送出去，ip_route_output_slow需要做哪些事情。将封包送往本地和转发出去二者都需要执行以下任务，虽然这些任务可能以不同的方式来执行：

- 从匹配的路由项中选择出使用的出口设备。
- 根据被搜索路由项的scope选择出将要使用的源IP地址。

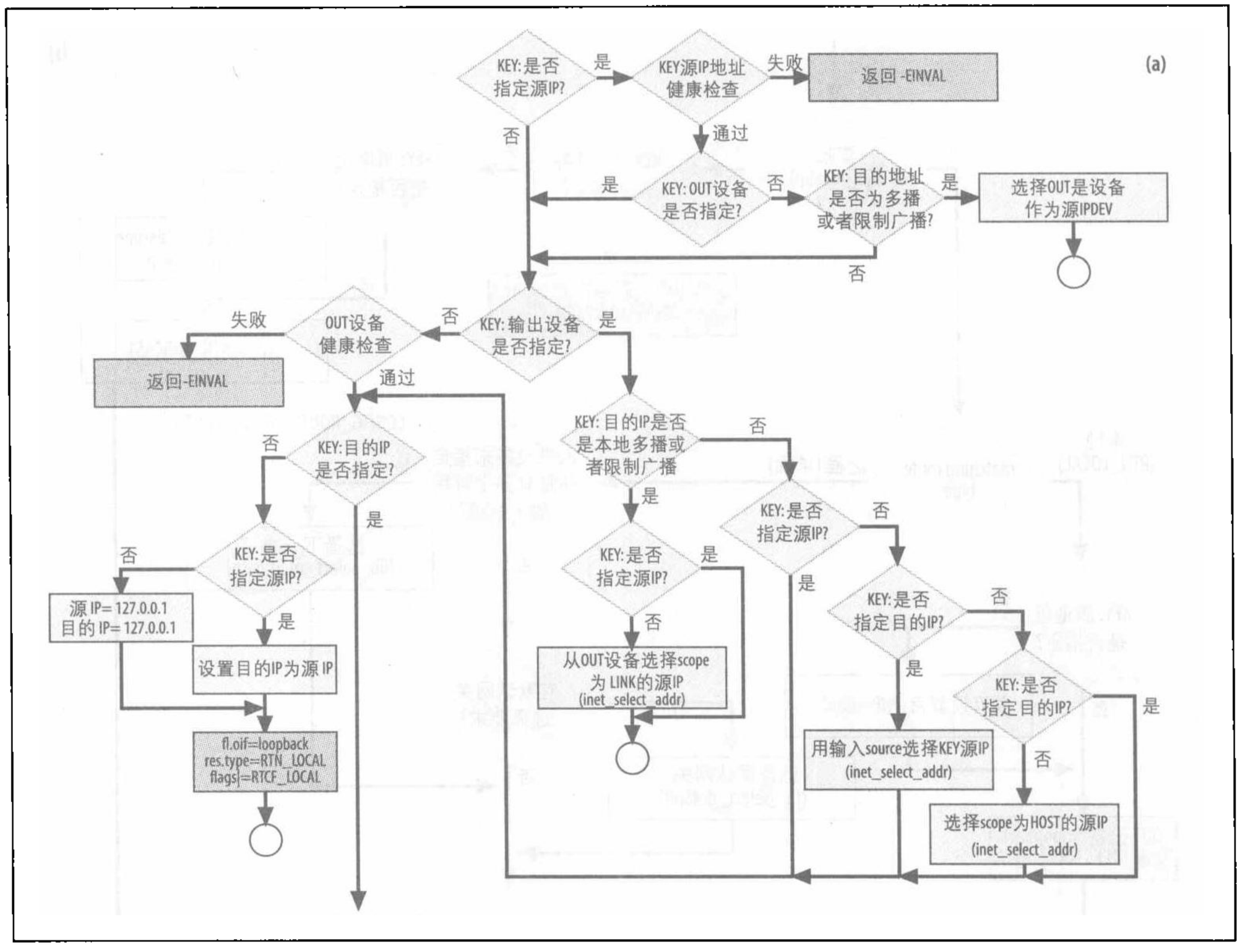


图 35-8 (a)：ip_route_output_slow 函数

- 创建及初始化一个新缓存表项，并将它插入到缓存内。

图 35-8 用点线分隔出三部分。上面部分，即 *a* 部分，利用函数输入参数来填充没有初始化的搜索关键字的字段。中间部分，即 *b* 部分，查找路由表，当需要时要从一个多路径路由项中选择出下一跳或需要选择出默认网关。下面部分，即 *c* 部分，创建新缓存表项。下面部分还根据函数中采用的转发决策结果来初始化`dst->input`和`dst->output`，函数主要通过一个局部标识变量来跟踪转发决策结果。

在一些情况下，不需要任何路由查找就可以路由一个封包（即无需调用图中中间部分的`fib_lookup`）。这主要有以下三种情况，在图 35-8 中都有描述：

当出口设备没有提供搜索关键字时，封包目的地为一个多播地址或受限的广播地址

这种情况是由于使用如 *vic* 和 *vat* 等多媒体工具而引发的一个问题，函数代码中有一段注释解释了这个问题，参见第二十六章“特殊情况”一节。

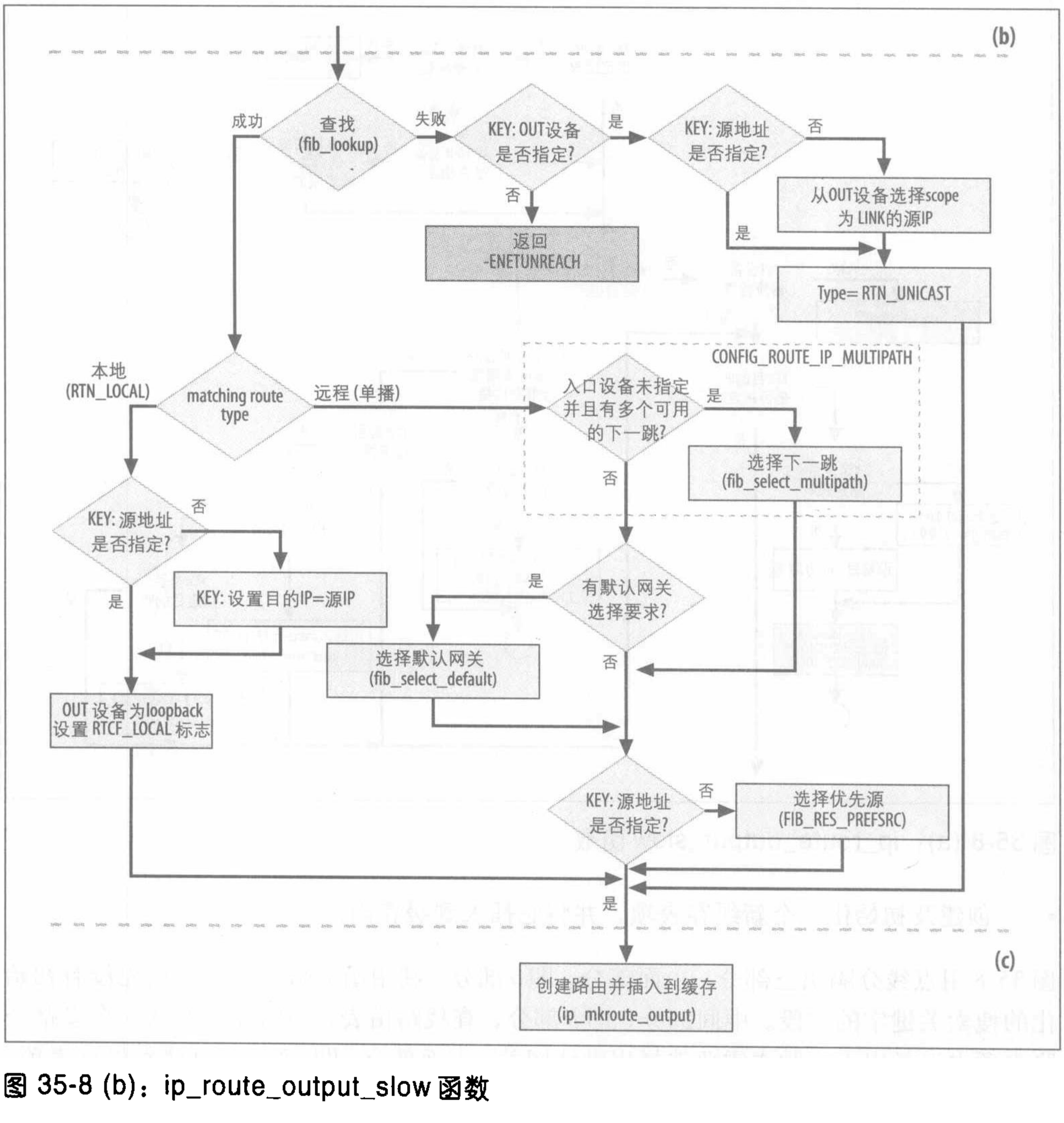


图 35-8 (b)：ip_route_output_slow 函数

封包目的地为本地多播地址（即224.0.0.X）或受限的广播地址（即255.255.255.255），从一个给定的设备发送出去

因为调用者提供了出口设备及搜索关键字，而且因为多播或受限的广播是scope为RT_SCOPE_LINK的地址，下一跳就是目的地址本身。所以路由子系统已经拥有路由该封包的所有信息，不再需要查找路由（注5）。参见第三十章“路由的基本元素”一节中对多播地址的讨论。

注5： 注意，L3-to-L2地址映射也是自动进行的，参见第二十六章“特殊情况”一节。

封包目的地为未知地址（0.0.0.0+）

这些封包被送往本地，而不是被发送出去。

搜索关键字初始化

下面是ip_route_input_slow如何来初始化搜索关键字，该搜索关键字传给fib_lookup用于路由表的查找。该搜索关键字将与新的被缓存的路由项一起被保存起来，以用于后续的缓存查找。

```
u32 tos     = RT_FL_TOS(oldflp);
struct flowi fl = { .nl_u = { .ip4_u =
                    { .daddr = oldflp->fl4_dst,
                      .saddr = oldflp->fl4_src,
                      .tos = tos & IPTOS_RT_MASK,
                      .scope = ((tos & RTO_ONLINK) ?
                            RT_SCOPE_LINK :
                            RT_SCOPE_UNIVERSE),
#ifdef CONFIG_IP_ROUTE_FWMARK
                      .fwmark = oldflp->fl4_fwmark
#endif
                        } },
                  .iif = loopback_dev.ifindex,
                  .oif = oldflp->oif };
```

源IP地址、目的IP地址和防火墙标记是直接从函数的输入参数拷贝而来。但TOS和scope的设置需要一些解释：

TOS

调用方可以把fl4_tos字段的两个最低有效位（two least significant bits）用于存储标识，ip_route_output_slow可以使用该标识来确定搜索路由的范围。因为TOS字段不需要占用整个八位，所以这种方法是可行的。参见第三十三章“出口查找”一节及第十八章中的图18-3。

如下所示，在*net/ipv4/route.c*文件中定义了RF_FL_TOS宏：

```
#define RF_FL_TOS(oldflp) \
((u32)(oldflp->fl4_tos & (IPTOS_RT_MASK | RTO_ONLINK)
```

Scope

当RTO_ONLINK标识被设置时，设置待搜索的路由项的scope为RT_SCOPE_LINK；否则设置为RT_SCOPE_UNIVERSE。参见第三十三章“出口查找”一节中涉及ARP的例子。

因为调用ip_route_output_slow只是为了路由本地产生的流量，所以搜索关键字fl

中的源设备被初始化为回环设备。我们将看到，如果目的地址也是本地地址，那么出口设备也被初始化为回环设备。

图 35-8(a)给出了当搜索关键字的一些基本字段没有提供输入关键字时，如何初始化。

选择源 IP 地址

搜索关键字中的源 IP 地址也是发送封包 IP 报头中的源 IP 地址。所以在 ip_route_output_slow的开始部分，如果搜索关键字的 fl.fl4_src存在，则选择它为源 IP 地址，后面代码中将 rth->rt_src 初始化为相同的值。

当搜索关键字没有提供源IP地址时（注6），则以目的地址类型作为inet_select_addr的输入参数来选择一个源IP地址（注7）。ip_route_output_slow还专门使用以下的scope作为 inet_select_addr的输入参数来处理特殊情况：

- RT_SCOPE_HOST当封包被送往本地（参见“送往本地”一节）。
- RT_SCOPE_LINK当封包被送到只在本地链路上有意义的地址，如广播地址、受限广播地址和本地多播地址。当 fib_lookup查找失败但封包还要送出时也使用该scope，因为搜索关键字提供了出口设备，因而假定目的地址为链路地址（参见“发送给其他主机”一节）。

当需要被路由的封包不属于上面列出的两种特殊情况时，ip_route_output_slow将fib_lookup查找到的路由结果res作为输入参数，调用FIB_RES_PREFSRC来选择源IP地址。FIB_RES_PREFSRC使用多种方法来选择首选源IP地址：如果用户为该路由项明确配置了首选源地址则返回它，否则用匹配路由的scope（res->scope）为输入参数，调用 inet_select_addr来得到首选源IP地址。

ip_route_output_slow将出口设备上配置的地址给予更高的优先级（如果该设备已知），将该地址作为inet_select_addr的第一个输入参数。但是也可以选择其他设备的地址。

图 35-9 总结了选择源 IP 地址的逻辑。

注 6：　参见“首选源地址选择”一节中当搜索关键字提供了源 IP 地址时的例子。

注 7：　该函数在“辅助函数”一节中有介绍。

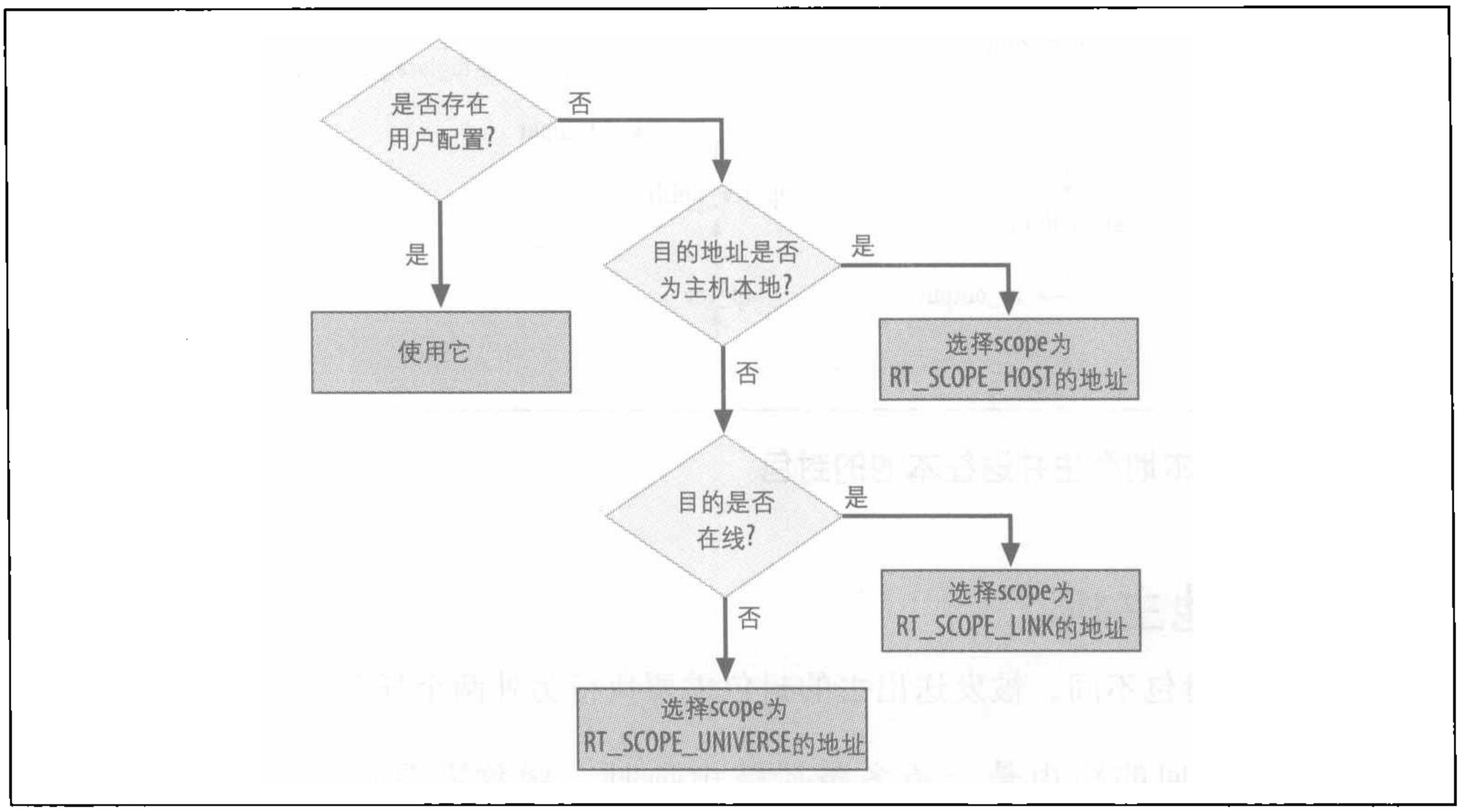


图 35-9：源 IP 地址选择

送往本地

当 fib_lookup 看到封包的目的地址是本地配置地址，或者当封包中没有提供目的地址时（即搜索包含了未知地址 0.0.0.0），该封包被送往本地。在这种情况下：

- 出口设备被设置为回环地址。这意味着该封包不会离开本地主机，该封包被发送出后，将重新注入到 IP 输入栈。
- dst->input 被初始化为 ip_local_deliver，这在第二十章“送往本地”一节中描述。正是由于这一操作，当封包重新注入到 IP 输入栈时，ip_rcv_finish 将启用 dst_input，即启用 ip_local_deliver 函数来处理该封包。

图 35-10 显示了这两个操作的效果：在内核网络代码中，封包从输出函数被转移到输入函数。

当搜索关键字中源 IP 地址和目的 IP 地址都没有被设置时，封包被送往本地，源地址和目的地址被设置为默认回环地址 127.0.0.1（INADDR_LOOPBACK），该地址的 scope 为 RT_SCOPE_HOST。

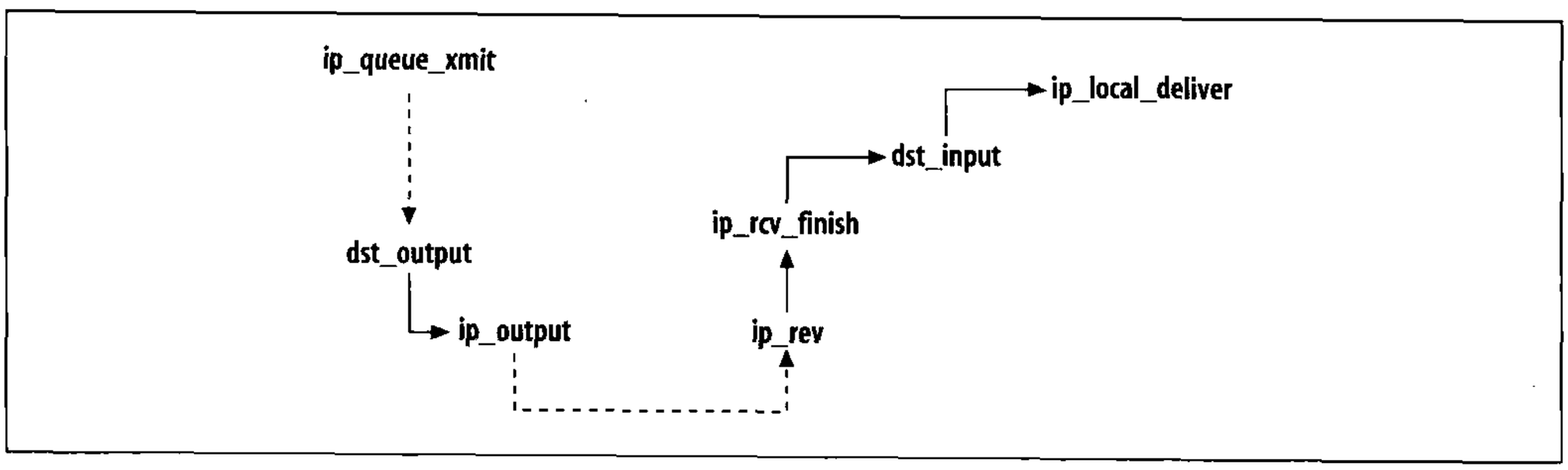


图 35-10：处理本地产生并送往本地的封包

发送给其他主机

与送往本地的封包不同，被发送出去的封包需要执行另外两个任务：

- 当查找返回的路由是一条多路径路由项时，函数需要选择下一跳。这是由 fib_select_multipath 来完成的。
- 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。这是由fib_select_default来完成的。(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)。

这两个任务在接下来的小节中讨论。

即使fib_lookup查找路由失败，但还是有可能成功地将封包发送出去。当搜索关键字提供了出口设备，ip_route_output_slow假定通过该出口设备可以直接到达目的地。在这种情况下，如果还没有源IP地址，则还需要设置一个scope为RT_SCOPE_LINK的源IP地址，可能的情况下用的是该出口设备上的一个地址。

多路径选择与默认网关选择之间的交互

从ip_route_output_slow中摘录的下面代码表明：当两个关键函数fib_select_multipath和fib_select_default分别用于处理多路径选择和默认网关选择时，res是fib_lookup返回的路由结果。

```
#ifdef  CONFIG_IP_ROUTE_MULTIPATH
    if (res.fi->fib_nhs > 1 && fl.oif == 0)
        fib_select_multipath(&fl, &res);
    else
#endif
    if (!res.prefixlen && res.type == RTN_UNICAST && !fl.oif)
        fib_select_default(&fl, &res);
```

注意，当搜索关键字指定了要使用的出口设备（fl.oif）时，不需要这两个函数。在这种情况下，res 已经包含了最终的转发决策。所以，执行 fib_lookup 及接下来调用 fib_semantic_match 函数（参见图 35-1）的主要任务就是选择：

- 当匹配路由是多路径路由时选择出下一跳。fib_semantic_match选择匹配出口设备的第一个下一跳路由器（参见“语义匹配补充标准”一节）。只有当内核在编译时支持多路径，才执行该条件代码。
- 当匹配路由是默认路由时选择出默认路由。fib_semantic_match选择匹配出口设备的第一个默认路由。fib_semantic_match不区分子网掩码长度不同的路由项，这意味着它并不特殊对待默认路由项，这种情况由 fib_semantic_match透明地处理。

多路径在“多路径对下一跳选择的影响”一节中描述，默认网关选择在“默认网关选择”一节中描述。

在本小节开始处给出的代码片段，如果不考虑使用环境的话，可能会引起以下两方面的误解：

- 它暗示多路径不能用于默认路由，因为在代码片段中的逻辑表明 fib_select_multipath 的执行不会调用接下来的 fib_select_default 函数。

  但是，多路径实际上可以用于默认路由。IPROUTE2 软件包提供的 *ip* 命令（需要配置多路径）允许对默认路由项配置多个下一跳。所以对路由项调用fib_select_multipath 足以完成路由决策。

  net-tools 的 route 工具允许管理员配置多条默认路由，每一条默认路由有一个下一跳。这种情况下就不存在多路径（fib_nhs 总是为 1）。所以 fib_select_default 足以完成路由决策。
- 它暗示管理员不能在一个出口设备上配置多个下一跳，因为只有在出口设备为空时才调用 fib_select_multipath。

  但是，使用相同的出口设备，配置一个有多个下一跳的多路径路由项是可能的。用出口设备（fl.oif）非空的搜索关键字来查找路由表由 fib_semantic_match处理，它简单返回匹配该设备的第一个可用的下一跳。fib_select_multipath 不处理这种选择。

默认网关选择

选择正确的默认网关由 fib_select_default 来完成，当以下两个条件都满足时它被 ip_route_output_slow 函数启用：

*fib_lookup返回的路由项子网掩码为/0（*res.prefixlen*为0）*

默认路由匹配任意目的地址，但它最后被检查的原因是由于子网掩码为/0，这是最短的子网掩码。如果配置的路由项没有一项匹配目的地址，那么只有匹配默认路由。但是因为所有的默认路由项都匹配，所以fib_lookup总是返回检查的第一个默认路由项。这就是为什么调用fib_select_default在多个可用的默认路由项中作出最佳选择。

fib_lookup*返回的路由项的类型为*RTN_UNICAST

本地路由、广播路由和多播路由都不需要网关，对这些路由项使用网关是没有意义的。

就像我们在第三十四章“函数的双重定义”一节中提到的，fib_select_default函数有两个版本。下面是不支持策略路由时使用的版本（定义在*include/net/ip_fib.h*文件（注8）中）：

```
static inline
void fib_select_default(const struct flowi *flp, struct fib_result *res)
{
    if (FIB_RES_GW(*res) && FIB_RES_NH(*res).nh_scope == RT_SCOPE_LINK)
        ip_fib_main_table->tb_select_default(ip_fib_main_table, flp, res);
}
```

flp是搜索关键字，res是在ip_route_output_slow中调用fib_lookup而返回的查找结果。

注意，当执行tb_select_default所需要的条件不满足时，调用者不会收到任何错误或警告信息，fib_select_default只是简单地将作为输入参数的fib_result实例返回。

tb_select_default被初始化为fn_hash_select_default，它在*net/ipv4/fib_hash.c*文件中定义，在下一小节对其描述。注意，只有当res路由项的下一跳网关的scope为RT_SCOPE_LINK时，fib_select_default才查找ip_fib_main_table表，其原因在第三十章“scope的使用”一节中已经描述。

fn_hash_select_default函数

fn_hash_select_default函数接收一个结构为fib_result的res作为输入参数，该参数是前面启用fib_lookup保存的路由查找结果。fn_hash_select_default使用该结构作为默认路由搜索的起点。

注8： 另一个版本参见“策略路由时选择默认网关”一节。

为了被选中，默认路由必须有与 res->scope 相同的 scope，优先级要小于或等于 res->fi->fib_priority，并且下一跳的 scope 为 RT_SCOPE_LINK（即必须直连）。

选择路由项也要考虑下一跳的状态是否可达。fib_detect_death 将路由项中 L3 地址已经被解析为 L2 地址的下一跳（即状态为 NUD_REACHABLE）给予更高的优先级。该检查可以确保当前所用的默认路由的下一跳网关不可达致使该路由项不可用时，则需要选择新的路由项。

前面选择出的默认路由被保存在全局变量 fn_hash_last_dflt 中。

整个函数运行时需要持有 fib_hash_lock 锁。

多路径对下一跳选择的影响

在ip_route_input_slow和ip_route_output_slow二者中，只有在以下情况下才调用 fib_select_multipath：

- 内核支持多路径（CONFIG_IP_ROUTE_MULTIPATH）。
- 利用 fib_lookup 查找路由表返回的路由项有多个下一跳（fib_nhs>1）。
- 搜索关键字没有提供出口接口。
- 目的地址不是本地地址、广播地址或多播地址。

下面代码给出了如何调用 fib_select_multipath 来选择下一跳：

```
#ifdef  CONFIG_IP_ROUTE_MULTIPATH
    if (res.fi->fib_nhs > 1 && fl.oif == 0)
        fib_select_multipath(&key, &res);
#endif
```

我们已经在第三十一章“选择下一跳”一节中看到，当一个路由项有多个下一跳可用时，Linux 是如何从中进行选择。我们现在来看看该算法是如何实现的。

我们在第三十四章“路由 hash 表的组织”一节中看到，一个路由项是由密切相关的数据结构 fib_node 和 fib_info 来表示，每个 fib_info 包含一个结构类型为 fib_nh（每个 fib_nh 表示路由项中的一个下一跳）的一个数组。

首先，我们需要清楚fib_info和fib_nh数据结构中的哪些字段被用于决定在一组可用的下一跳中，是否必须选择某个下一跳，如果是，那么选择哪一个。

下面是用于保存多路径配置的一些字段：

fib_info->fib_nhs

路由项定义的下一跳的个数。

fib_info->fib_nh

fib_nh结构数组，数组的大小为fib_info->fib_nhs。

下面字段被用于实现加权随机循环算法：

fib_info->fib_power

该字段被初始化为fib_info实例的所有下一跳权值（fib_nh->nh_weight）的总和，但不包含由于某些原因而不能使用的下一跳（带有RTNH_F_DEAD标识）。每当调用fib_select_multipath来选择一个下一跳时，fib_power的值递减。当该值递减为零时被重新初始化。

fib_nh->nh_weight

下一跳的权值。当用户没有明确配置时被设置为默认值1。我们将看到，这个值可以使fib_select_multipath对每个下一跳的选择机会，与该下一跳的权值成比例（相对于fib_info->fib_power而言）。

fib_nh->nh_power

使该下一跳被选中的令牌。这个值是在初始化fib_info->fib_power时，首先被初始化为fib_nh->nh_weight。每次fib_select_multipath选中该下一跳时就递减该值，当该值递减为零时，不再选中该下一跳，直到nh_power被重新初始化为fib_nh->nh_weight(这是在重新初始化fib_info->fib_power值时进行的)。

现在，我们将看到如何实现的工作机理。

当选择算法开始运行时，所有下一跳的tokens数（nh_power）与其权值相同。我们在前面看到，下一跳的权值默认为1。change_nexthops循环设置下一跳的nh_power字段，同时将所有下一跳的权值累加到函数的局部变量power上。

```
spin_lock_bh(&fib_multipath_lock);
if (fi->fib_power <= 0) {
    int power = 0;
    change_nexthops(fi) {
        if (!(nh->nh_flags&RTNH_F_DEAD)) {
            power += nh->nh_weight;
            nh->nh_power = nh->nh_weight;
        }
    } endfor_nexthops(fi);
```

fib_info->fib_power被初始化为所有下一跳权值的总和。因为这个值在每次fib_select_multipath作出决策时都要被递减（见本小节稍后的代码），所以每个下一跳被选中的次数等于该下一跳的权值，直到fib_power递减为0。这也保证了当

fib_info->fib_power被递减为0时，每个下一跳已经被选中的次数是与其权值成比例。

```
            fi->fib_power = power;
            if (power <= 0) {
                spin_unlock_bh(&fib_multipath_lock);
                res->nh_sel = 0;
                return;
            }
        }
```

通过fib_select_multipath选中一个下一跳是伪随机的：每次fib_select_multipath被调用时，它产生一个取值范围为零到fib_info->fib_power-1的随机数w，然后遍历所有的下一跳，直到发现某个下一跳的令牌数（fib_nh->nh_power）大于或等于w。注意，每次循环时递减w，以使每一次循环都更有可能找到匹配条件的下一跳。

```
        w = jiffies % fi->fib_power;
        change_nexthops(fi) {
            if (!(nh->nh_flags&RTNH_F_DEAD) && nh->nh_power) {
                if ((w -= nh->nh_power) <= 0) {
                    nh->nh_power--;
                    fi->fib_power--;
                    res->nh_sel = nhsel;
                    spin_unlock_bh(&fib_multipath_lock);
                    return;
                }
            }
        } endfor_nexthops(fi);
        res->nh_sel = 0;
        spin_unlock_bh(&fib_multipath_lock);
```

多路径缓存

图35-4显示了对入口流量和出口流量，何时调用前一小节描述的fib_select_multipath函数，以及对多路径缓存的支持是如何影响到通过ip_mkroute_input和ip_mkroute_output向路由缓存内加入表项的方式。我们对入口和出口情况分别进行分析。

入口流量

当内核不支持多路径缓存时，并且当前一小节列出的条件满足时ip_mkroute_input调用fib_select_multipath，根据前面描述的逻辑选出一个下一跳。

当内核支持多路径缓存时，不用fib_select_multipath来选择下一跳。相反，

它对多路径路由项的所有下一跳循环，对每个下一跳都向缓存内添加一个表项。对每个路由项还调用multipath_set_nhinfo，该函数在第三十三章“路由缓存与多路径之间的接口”一节中描述。缓存算法使用该函数来更新本地信息以选择下一跳。例如，加权随机算法使用该函数来组织下一跳的数据库（参见第三十三章“加权随机算法”一节）。

出口流量

如图35-4所示，出口情况与入口情况非常类似。唯一的区别在于即使当内核支持多路径缓存时也调用fib_select_multipath，代码后面启用ip_mkroute_output来覆盖由fib_select_multipath选出的下一跳。

在上面两种情况下，下一跳选择结果res->nh_sel被初始化为多路径路由项的最后一个下一跳。对后续封包而言，在缓存查找时就可以选择出下一跳。参见第三十三章“多路径缓存”一节。

策略路由

当内核支持策略路由时，路由查找需要考虑到可能存在多个路由表。下面两个小节描述fib_lookup和fib_select_default的策略路由版本与本章前面介绍的基本版本之间有何区别。

策略路由时的 fib_lookup

当（内核）配置了策略路由时，fib_lookup函数包含一个额外步骤：需要根据配置的策略找出使用哪一个路由表。

我们在第三十二章“主要的数据结构”一节中看到，路由策略用fib_rule数据结构来定义。所有的fib_rule实例被全局链表fib_rules链接到一起。这个链表按照priority字段的递增顺序保存，允许配置定义路由规则检查的顺序，从而缩短查找时间。更常用的匹配规则或最重要的规则（由管理员根据具体情况来定义）更靠近链表头部。priority字段占32位，这意味着一台主机理论上可以配置多达2^{32}个策略。当然，因为策略存储在一个有序的、均匀的链表内，策略数目过多将显著降低路由查找性能。

即使用户没有配置任何规则，fib_rules也包含三个默认的实例，这三个实例定义在*net/ipv4/fib_rules.c*文件中；如图35-11所示：

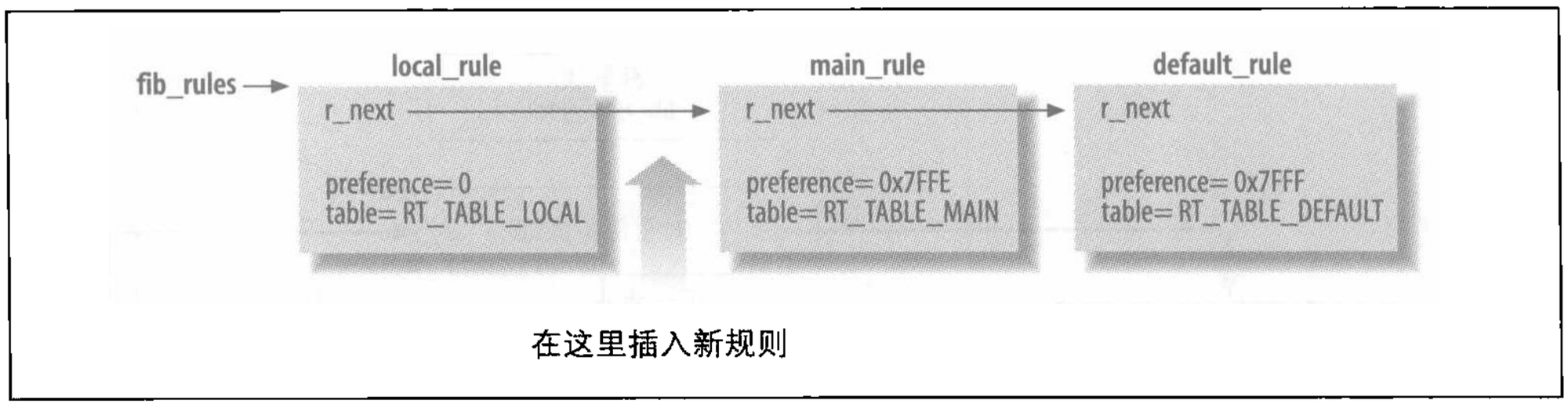


图 35-11：默认规则

local_rule

这是优先级最高的规则，因而处于链表头部。该规则总是匹配，其目的是强制第一个查找的是 ip_fib_local_table 路由表。这是因为目的地址为本地系统的封包只需要查找该表，而不需要其他更进一步的路由决策。

main_rule

这是检查的第二个路由表（除非管理员在中间插入一些用户定义的路由表）。该规则同样总是匹配，它对主路由表 ip_fib_main_table 进行搜索。

default_rule

这是默认规则，位于链表的尾部。

图 35-12 给出了 fib_lookup 的实现逻辑。它一个接一个地遍历路由策略，直到查找到与路由封包匹配的策略或到达策略链表尾部还没有找到任何匹配。当查找到匹配策略时，接下来的动作依赖于策略类型（参见第三十一章“策略路由查找”一节）。特别地，策略动作 RTN_UNREACHABLE、RTN_BLACKHOLE 和 RTN_PROHIBIT 将返回一个错误，其值作为 fib_lookup 的调用者产生 ICMP 消息的依据。策略动作 RTN_UNICAST 将调用 tb_lookup 来查找，它实际上是调用在“路由表查找：fn_hash_lookup”一节中描述的 fn_hash_lookup 函数，该函数可以返回多种结果。除了在专门章节中已经描述的错误以外，还需要注意到：

- 当查找成功时，res->r 被初始化为匹配策略。
- 当查找失败时，如果错误类型为 -EAGAIN 则 fib_lookup 继续在循环内查找策略。返回该错误值的原因是因为与通过 fn_hash_lookup 查找到的匹配路由相关的动作类型为 RTN_THROW（参见第三十章“路由类型与动作”一节）。

策略路由时选择默认网关

不支持策略路由时与支持策略路由时选择默认路由的工作原理相同，唯一的区别在于定

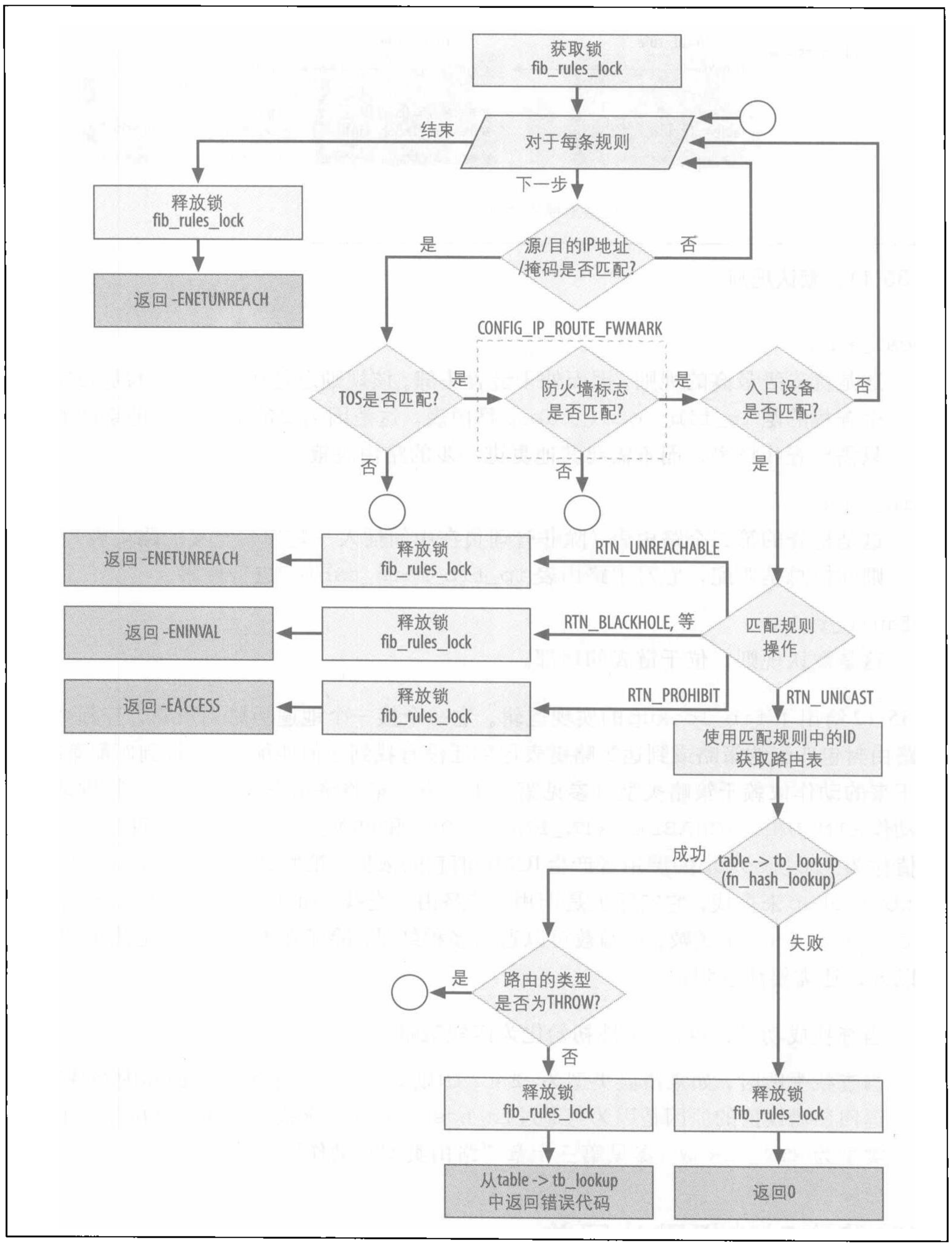


图 35-12：fib_lookup 函数的策略路由版本

义在*net/ipv4/fib_rules.c* 文件中的fib_select_default，使用匹配策略（res->r）来确定使用的路由表。

```
void fib_select_default(const struct flowi *flp, struct fib_result *res)
{
    if (res->r && res->r->r_action == RTN_UNICAST &&
        FIB_RES_GW(*res) && FIB_RES_NH(*res).nh_scope == RT_SCOPE_LINK) {
        struct fib_table *tb;
        if ((tb = fib_get_table(res->r->r_table)) != NULL)
            tb->tb_select_default(tb, flp, res);
    }
}
```

源路由

我们在第十八章中看到IP封包可以被源路由。因为它是由IP协议直接处理而不涉及路由子系统，所以是在本书的IP部分讨论。这里我们感兴趣的是源路由（source routing）所要涉及的路由查找。

我们用第十八章中的图18-1作为一个参考。当一个入口封包到达ip_rcv_finish时，触发第一次路由查找。在没有源路由的情况下，这是唯一的一次路由查找。但是在ip_rcv_finish调用dst_input之前，它检查IP报头是否指定了源路由，如果是则进行处理。

源路由在这里是通过ip_options_rcv_srr来处理。它从IP报头提取出要使用的下一跳，并调用ip_route_input来进行第二次路由查找。第二次路由查找使用更新的查找结果替代原来的skb->dst。参见图35-13中的调用顺序。

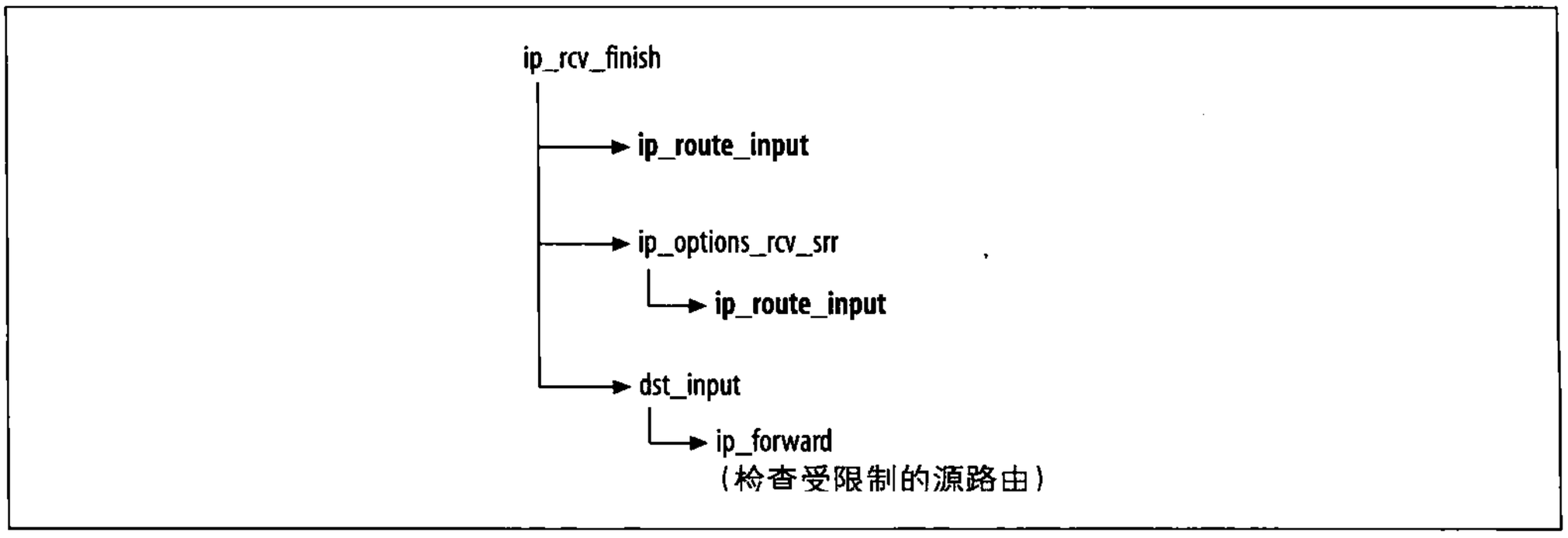


图35-13：入口流量的源路由

当本地产生的流量携带有源路由IP选项时，只触发一次路由查找，因为在此次路由查找之前已经选出正确的下一跳（参见 ip_queue_xmit）。

策略路由与基于路由表的分类器

我们在第三十一章“基于路由表的分类器”一节中看到，实现网络 QoS 层的流量控制子系统，可以根据路由子系统计算的一个标签（tag）来分类封包。同样是在这一节中，我们看到如何配置 realm，以及根据这些配置得出路由标签的逻辑。在本小节中，我们将看到如何将 realm 配置存储在路由表中，以及如何根据路由代码来计算路由标签。因为流量控制超出了本书的范围，所以这里我们不讨论如何使用路由标签。

存储 realm

内核将策略 realm 和路由 realm 分别存储在 fib_rule->r_tclassid 和 fib_nh->nh_tclassid 字段内。

fib_rule->r_tclassid

源 realm 和目的 realm 都是 8 位的值（范围为 0～255），但在 r_tclassid 中，它们都占用 16 位。当配置源 realm 时它被保存在高 16 位，当配置目的 realm 时它被保存在低 16 位，参见图 35-14。

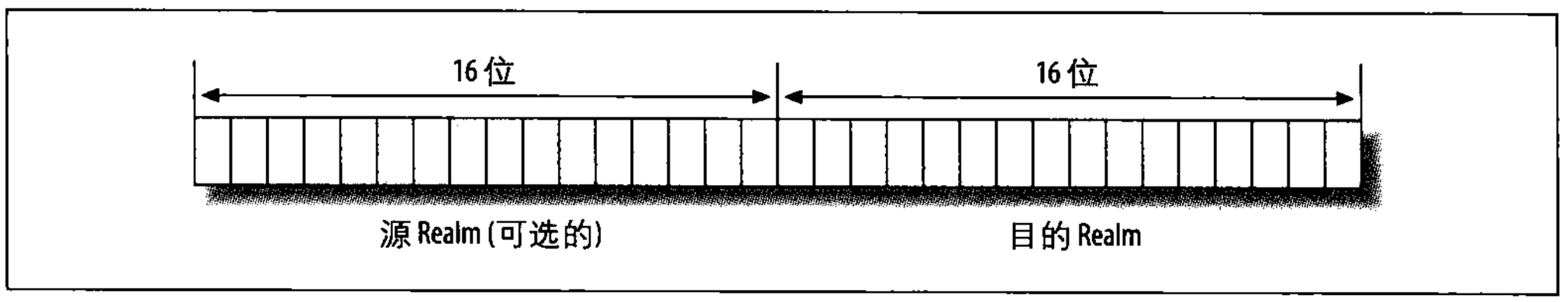


图 35-14：r_tclassid 字段结构

fib_nh->nh_tclassid

一般情况下，只使用目的 realm 来计算路由标签，根据目的地址来选择匹配路由。但是，就像我们在第三十一章“计算路由标签”一节中看到的，内核有时侯需要反向路径查找。当这种情况发生时，路由项的目的 realm 是从反向路由的源 realm 得出的。nh_tclassid 是一个 32 位的变量。

辅助函数

在查看如何初始化 dst.tclassid 之前，我们首先来看为了完成这一任务而用到的一些辅助函数：

fib_rules_tclass

它用于从一个fib_rule数据结构内提取r_tclassid字段。因为fib_lookup返回的结果内包含一个指向匹配的fib_rule实例的指针，所以在查找之后使用fib_rules_tclass来提取匹配规则。注意，只有当内核支持策略路由时才调用该函数，此时fib_rule结构才有意义。

fib_combine_itag

图35-15给出了这个函数的逻辑，它是在进行反向路径查找时来辅助查找realms。

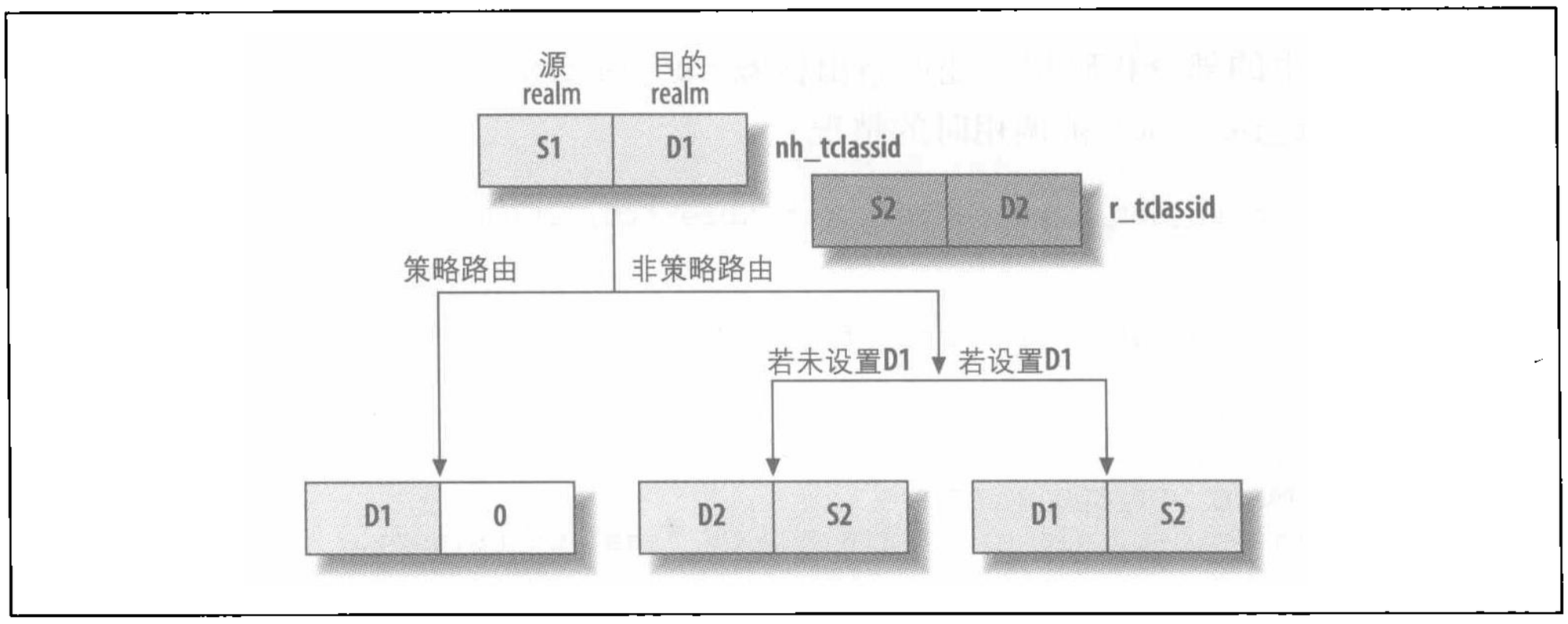


图35-15：fib_combine_itag函数

当内核关闭策略路由时，它简单地将源路由realm和目的路由realm交换。

当内核开启策略路由时，这个函数将策略源realm（图35-15中的S2）作为目的realm。而且，如果目的路由realm（D1）存在则将D1作为源realm，否则将目的策略realm（D2）作为源realm。

结果是用输入参数itag来返回，当调用者启用rt_set_nexthop时使用该返回结果（参见“计算路由标签”一节）。

在反向路径查找之后，这个函数被fib_validate_source调用。fib_validate_source接收源IP地址和目的IP地址作为输入并将它们交换，然后调用fib_lookup进行反向路径查找。因而，fib_lookup返回的结果中源realm和目的realm也已经被交换。因为这两个realm字段都为16位宽，而且fib_lookup返回的两个realm已经被交换，所以fib_combine_itag 中使用16位移位来进行调整。

set_class_tag

给定一个路由（因而也就可以得到相关的dst_entry.tclassid）和一个已经被调用者初始化了的标签，set_class_tag使用第二个参数来填充dst_entry.tclassid中还没有被初始化的realm。

计算路由标签

路由标签必须由我们在本章前面讨论的ip_route_input_slow和ip_route_output_slow函数来计算。采用的逻辑在第三十一章“计算路由标签”一节中描述。

计算一个路由标签所需要的信息是要路由的skb封包和路由查找结果skb->dst。路由标签保存在skb->dst.tclassid中。一旦ip_route_input_slow和ip_route_output_slow成功找到转发信息，它们就创建并初始化一个新的路由缓存表项，包括对路由标签，然后将该表项添加到缓存内。缓存表项的部分初始化是用rt_set_nexthop来完成的，其中的部分代码用于处理路由标签dst_entry.tclassid。图35-4准确地显示了rt_set_nexthop被调用时的情况。

```
static void rt_set_nexthop(struct rtable *rt, struct fib_result *res, u32
itag)
{
    struct fib_info *fi = res->fi;

    if (fi) {
        ......
#ifdef CONFIG_NET_CLS_ROUTE
        rt->u.dst.tclassid = FIB_RES_NH(*res).nh_tclassid;
#endif
    }
    ......

#ifdef CONFIG_NET_CLS_ROUTE
#ifdef CONFIG_IP_MULTIPLE_TABLES
    set_class_tag(rt, fib_rules_tclass(res));
#endif
    set_class_tag(rt, itag);
#endif
    ......
}
```

前面的代码片段显示了当内核支持基于路由表的分类器时，首先用目的路由的realm来初始化tclassid（否则不需要做）。注意，根据内核是否支持策略路由，调用set_class_tag时所使用的输入参数不同：

支持策略路由

根据策略realms来填充dst.tclassid中还没有被初始化的字段。

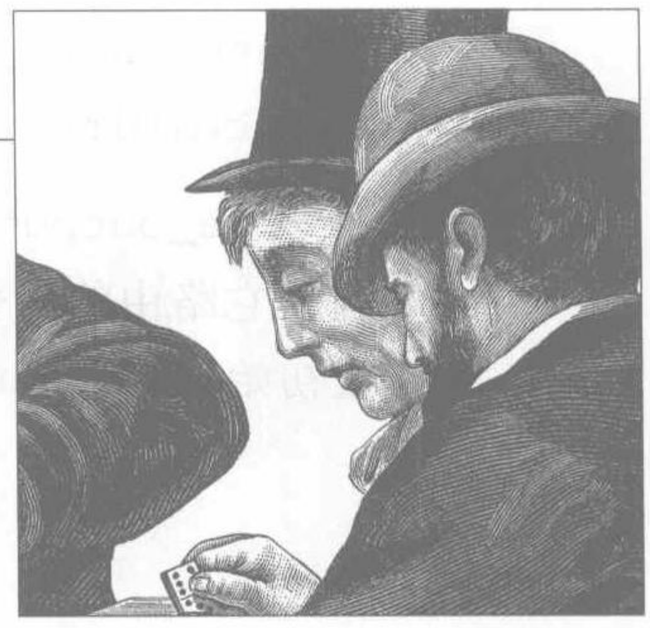
不支持策略路由

使用调用者先前计算的itag输入参数来填充dst.tclassid中还没有被初始化的字段：

- ip_route_input_slow（通过调用__mkroute_input）传递的itag值，是用fib_combine_itag计算的。
- ip_route_output_slow（通过调用__mkroute_output）传递的itag值为0，因为它路由的封包是本地产生的，因而内核不用做任何反向查找来填充还没有被初始化的realms。

第三十六章

路由：相关主题



在前面的章节中，我们看到各种路由功能是如何工作、它们之间如何相互作用及它们如何与其他内核子系统进行交互。在本章，我们通过描述路由子系统是如何与配置路由信息的用户空间命令相互作用，来结束本书对路由部分的讨论。因为路由管理问题超出了本书讨论的范围，这里我不会详细描述这些命令本身。我们还将看到输出到*/proc*目录下可用于调整路由的各种文件。在本章最后，我们将详细描述在第三十二章中已经介绍过的数据结构。

用户空间配置工具

路由的配置可以使用*net-tools*软件包和IPROUTE2软件包，它们分别使用`ioctl`和Netlink接口。下面的几小节将对这两个包详细介绍，但主要集中在IPROUTE2软件包中的*ip*命令，它是在Linux上更新、功能更强大的路由配置方法。

如果你了解了这两组工具的局限性和使用依据，那么它们可以共存是没有问题的。*net-tools*不能够配置任何高级路由功能，例如，多路径和策略路由。通过*net-tools*包中的各个实用工具显示的结果中也看不到这些高级路由功能。但IPROUTE2包提供的路由配置命令向后与*net-tools*兼容。

图36-1总结了我们将在下面几节看到的内容。该图说明了通过这两个内核接口来操作路由表的主要函数，以及*net-tools*使用的`ioctl`命令（IPROUTE2还允许你配置如策略规则等其他对象，但图中为了简洁而没有显示）。

以下几点值得注意：

- 这两种工具添加路由和删除路由最终都是调用相同的函数：`fn_hash_insert`和`fn_hash_delete`，我们在第三十四章已经看到。

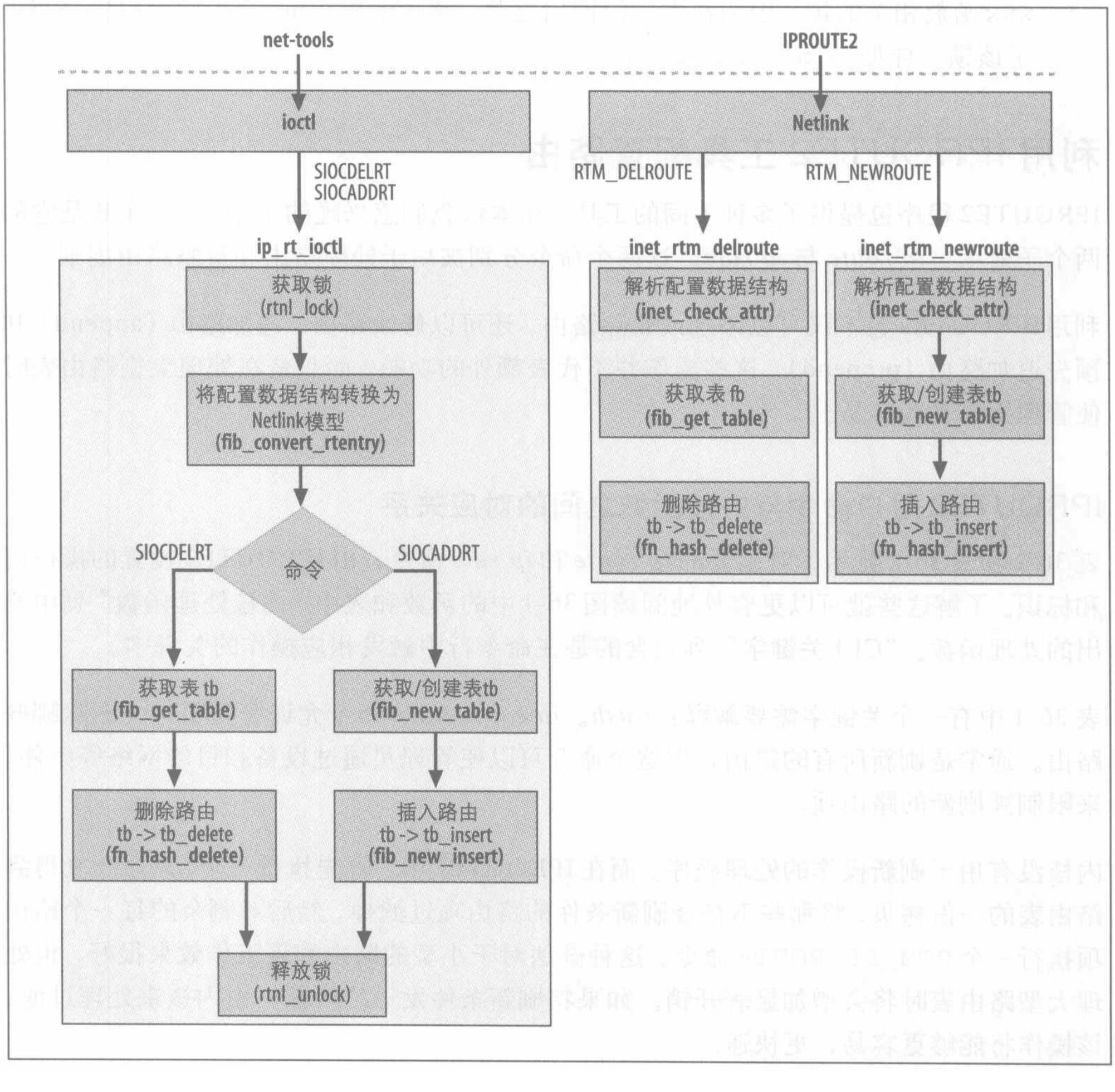


图 36-1：基于 ioctl 和基于 Netlink 的路由表操作

- 由上一点可知，在调用通用函数 fn_hash_xxx之前，必须将从这两个用户空间工具接收到的输入信息保存到相同的数据结构内。因为这两个工具与内核之间是通过不同的消息类型来通信，并且由于 Netlink 是新的、首选的接口，所以利用 fib_convert_rtentry 将 ioctl 命令的输入转换为 Netlink 格式。对于把用户输入的字符串转换为本章后面描述的内核数据结构的请求，也由这个转换函数来负责分析，因此不需要明确调用分析函数 inet_check_attr（它被 inet_rtm_xxx 函数所替代）。
- 为了使路由配置按次序变化而使用了一个锁。图36-1没有给出与这两个net_rtm_

xxx 函数相关的锁，因为在它们被调用之前，路由的 Netlink 套接字代码已经获得了该锁（详见 rtnetlink_rcv）。

利用 IPROUTE2 工具配置路由

IPROUTE2 程序包提供了多种不同的工具。在本章我们感兴趣的是 *ip* 命令，尤其是它的两个子命令：*ip route* 与 *ip rule*，这两个命令分别被用于处理路由和策略路由规则。

利用 IPROUTE2 包不仅可以添加或删除路由，还可以修改路由、追加路由（append）和预先追加路由（prepend）。这些操作并不代表额外的功能，而只是在处理大型路由表时使管理操作更为容易。

IPROUTE2 用户命令与内核函数之间的对应关系

表 36-1 和表 36-2 显示了对主要的 *ip route* 和 *ip rule* 命令，由 IPROUTE2 设置的操作码和标识。了解这些就可以更容易地阅读图 36-1 中的函数和表中“内核处理函数”列中给出的处理函数。“CLI 关键字”列包含的是在命令行中触发相应操作的关键字。

表 36-1 中有一个关键字需要解释：*flush*。*ip route flush* 命令允许管理员定义删除哪些路由。通常是刷新所有的路由，但这个命令可以使管理员通过设备和目的网络等条件，来限制被刷新的路由项。

内核没有用于刷新操作的处理程序。而在 IPROUTE2 中，首先执行一个 *list* 命令来得到路由表的一份拷贝，将那些不符合刷新条件的路由项过滤掉，然后对剩余的每一个路由项执行一个 RTM_DELROUTE 命令。这种做法对于小型的路由配置工作效果很好，但处理大型路由表时将会增加显著开销。如果将刷新条件发送给内核，由内核来处理过滤，该操作将能够更容易，更快速。

表 36-1：通过 IPROUTE2 iproute.c 文件设置的参数

CLI 关键字	操作	标识	内核处理程序
add	RTM_NEWROUTE	NLM_F_EXCL NLM_F_CREATE	inet_rtm_newroute
change	RTM_NEWROUTE	NLM_F_REPLACE	inet_rtm_newroute
replace	RTM_NEWROUTE	NLM_F_CREATE NLM_F_REPLACE	inet_rtm_newroute
prepend	RTM_NEWROUTE	NLM_F_CREATE	inet_rtm_newroute
append	RTM_NEWROUTE	NLM_F_CREATE NLM_F_APPEND	inet_rtm_newroute

表 36-1：通过 IPROUTE2 iproute.c 文件设置的参数（续）

CLI 关键字	操作	标识	内核处理程序
test	RTM_NEWROUTE	NLM_F_EXCL	inet_rtm_newroute
delete	RTM_DELROUTE	None	inet_rtm_delroute
list/lst/show	RTM_GETROUTE	None	inet_dump_fib
get	RTM_GETROUTE	NLM_F_REQUEST	inet_rtm_getroute
flush	RTM_GETROUTE	None	None

表 36-2:通过 IPROUTE2 的 iprule.c 文件设置的参数

CLI 关键字	操作	标识	内核处理程序
add	RTM_NEWRULE	None	inet_rtm_newrule
delete	RTM_DELRULE	None	inet_rtm_delrule
list/lst/show	RTM_GETRULE	None	inet_dump_rules

注意，从“CLI关键字”列中可以看出，一些内核处理程序需要处理多个用户命令类型，内核通过操作和标识参数的组合来区分不同的命令。

如图 36-1 所示，处理路由的内核处理程序是 inet_rtm_newroute 和 inet_rtm_delroute，它们在下一节中描述。关于表 36-1 中的这两个处理程序如何实现，我将留作一个练习（可以参考第三十五章“策略路由”一节）。

若读者对于研究 IPROUTE2 工具本身的代码有兴趣，图 36-2 说明了 IPROUTE2 包中，对各种 *ip route* 和 *ip rule* 命令请求进行解析，并将请求发送给内核的相关文件和函数。例如，如果你输入 *ip route add ...* 命令，在 *ip.c* 中的函数 main 将调用 *iproute.c* 文件中定义的 do_iproute 来处理该命令。因为该操作为 *add*，所以 do_iproute 将用 iproute_modify 来处理该命令。

inet_rtm_newroute 函数和 inet_rtm_delroute 函数

当内核收到用户利用IPROUTE2工具发送的路由请求时，就调用这两个函数，它们分别处理添加路由和删除路由，如图 36-1 和表 36-1 所示。

这两个函数都调用 inet_check_attr 来填充一个 kern_rta 结构，该结构保存了对用户命令解析后的结果。kern_rta 结构中的所有字段都是指针：它们直接指向从用户空间接收到的数据结构内的字段。空指针意味着相关的字段没有被配置。

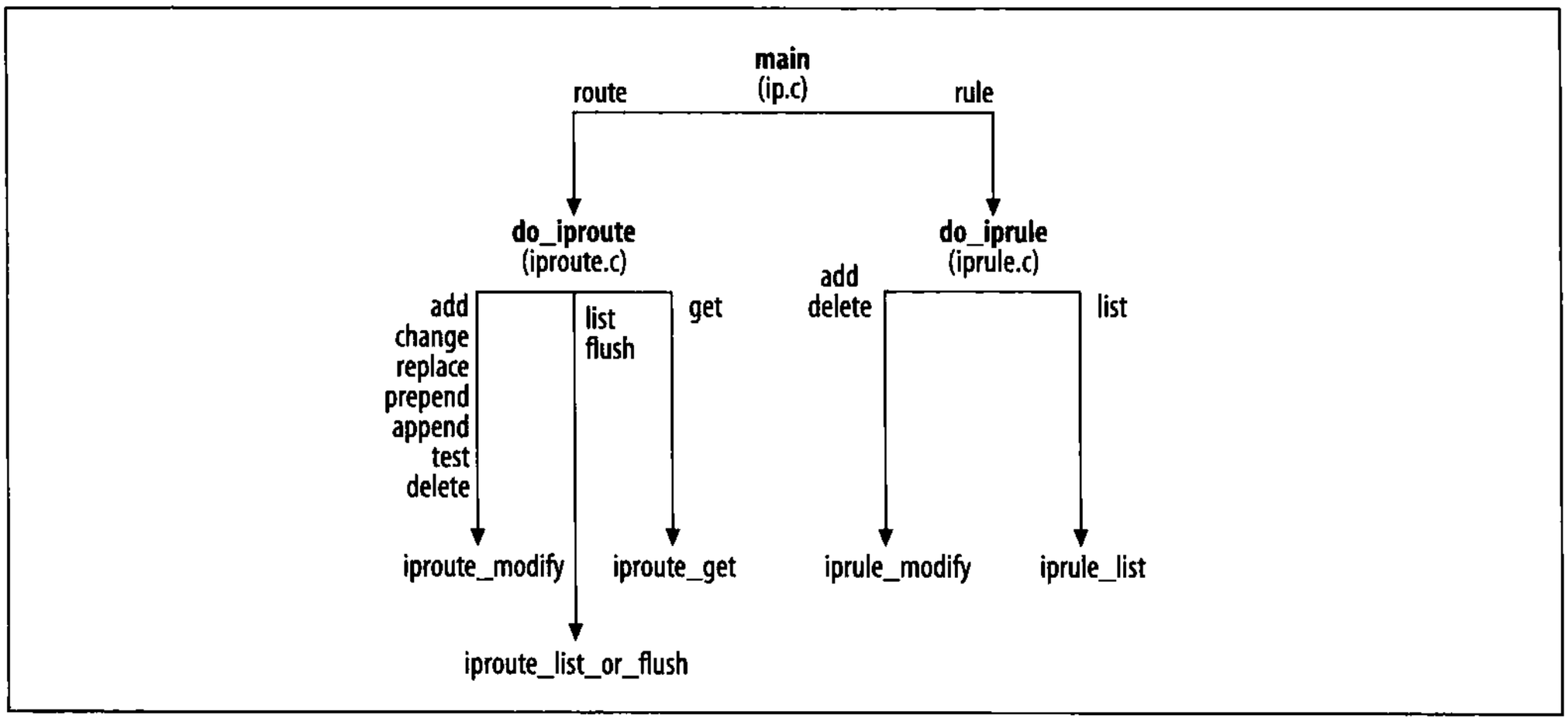


图 36-2：IPROUTE2 包用于路由的文件和函数

在本节我们考察inet_rtm_newroute函数，inet_rtm_delroute的操作与其相对应。

```
int inet_rtm_newroute(struct sk_buff *skb, struct nlmsghdr* nlh, void
*arg)
{
    struct fib_table * tb;
    struct rtattr **rta = arg;
    struct rtmsg *r = NLMSG_DATA(nlh);

    if (inet_check_attr(r, rta))
        return -EINVAL;

    tb = fib_new_table(r->rtm_table);

    if (tb)
        return tb->tb_insert(tb, r, (struct kern_rta*)rta, nlh, &NETLINK_CB(skb));
    return -ENOBUFS;
}
```

首先该函数用inet_check_attr分析输入的消息nlh，并将结果保存到rta中。当需要添加一个路由时，用户可以指定将该路由项添加到哪一个路由表中。多路由表的概念已经在第三十一章“策略路由背后的概念”一节中给出了非常详细的描述。如果指定的路由表不存在，则利用fib_new_table创建并初始化一个路由表。现在有了对该路由表的引用，该函数就调用其虚拟函数tb_insert来执行插入操作。我们在第三十四章“路由的添加和删除”一节中看到，tb_insert调用fh_hash_insert，其内部结构已经在第三十四章“添加路由”一节中给出了描述。

利用 net-tools 包配置路由

在大多数 Unix 系统中都可见到 *net-tools* 包中的 *route* 命令，这是最常见的配置路由和转储路由表及路由缓存内容的方法。

route add 和 *route del* 命令分别向内核发送 ioctl 命令 SIOCADDRT 和 SIOCDELRT，用来添加路由和删除路由。但转储路由表和路由缓存采用用另一种方式：*route*命令会简单地转储 */proc/net/route* 和 */proc/net/rt_cache* 文件的内容（注 1）。

内核处理程序 ip_rt_ioctl 负责这两个 ioctl 命令的处理，ip_rt_ioctl 定义在 *net/ipv4/fib_frontend.c* 文件中。图 36-1 说明了其部分内部结构。

只有具有网络管理员权限（CAP_NET_ADMIN）的用户才可以使用 *route* 命令。内核会调用 capable 来强制这一规则（注 2）。因为携带着路由删除或添加信息的数据结构属于用户空间，所以需要利用 copy_from_user 将其拷贝到内核空间的某一地址处。

```
......
if (!capable(CAP_NET_ADMIN))
    return -EPERM;
if (copy_from_user(&r, arg, sizeof(struct rtentry)))
    return -EFAULT;
   ......
```

变换通知

我们在第三章中看到，Netlink定义了目的为送出关于特定类型事件通知的多播组，用户程序可以注册成为这些组的一部分。在这些多播组中有一个RTMGRP_IPV4_ROUTE组，用于在 IPv4 路由表发生变化时向外通知。这些变化将使用 rtmsg_fib 函数发送给 RTGRM_IPV4_ROUTE 多播组。

当通过其他守护进程或用户手工配置对路由项添加或删除时，对这些事件感兴趣的监听者，例如，各路由守护进程需要得到通知。用户也可以使用 IPROUTE2 包的 *ip monitor route* 命令来测试这一功能。图 36-3 说明了这样一个例子：每次当一个终端上的路由表变化应用时，事件通知就被输出到另一个正在执行 *ip monitor route* 命令的终端上（注 3）。终端和内核通过一个 Netlink 套接字通信。

注 1：参见 *net-tools* 包中的文件 *lib/inet_gr.c*。

注 2：关于用户权限和处理能力的更多细节，可参见 O'Reilly 的《深入理解 Linux 内核》一书。

注 3：当前并不是所有的路由变化都产生通知。例如，当一个设备关闭而删除相关的 IPv4 路由时，就不会通知 IPv4 协议。但这种行为可能会发生变化，例如，IPv6 中同样的操作已经发送通知。

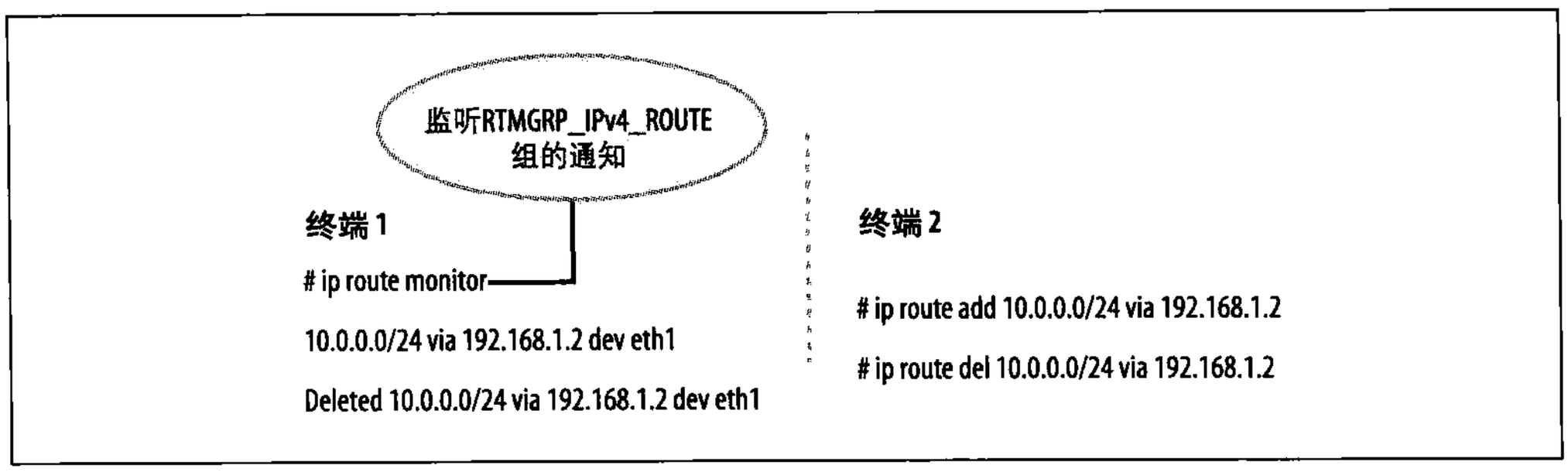


图 36-3：ip monitor route 命令使用范例

内核插入路由项：fib_magic 函数

我们在图 36-1 中看到 Netlink 套接字可以用于用户空间与内核之间的消息交换。但在某些情况下，内核中的不同部分也使用 Netlink 消息相互通信。这样事情变得简单，例如，利用通常对用户产生事件作出响应的代码，就可以同样用于对内核产生的事件作出响应。

例如，我们在第三十二章“添加一个 IP 地址”一节中看到，当对一个接口配置一个新的 IP 地址时，可以产生一组路由表项。添加这些路由项的一个简单方法是模拟接收到来自用户空间请求插入新路由项的命令。这是由 `fib_magic` 函数实现，该函数将创建一条 Netlink 消息，该消息与通过 *route add* 或 *ip route add* 命令而添加路由所产生的消息相同。

`fib_add_ifaddr` 和 `fib_del_ifaddr` 是使用 `fib_magic` 的两个很好的例子，关于这两个函数的更多细节参见第三十二章“IP 配置改变”一节。

统计数据

路由代码对其不同的方面都进行统计，例如，路由查找和垃圾回收。统计数据都是针对每个处理器的。在第三十二章“路由子系统初始化”一节中描述的 `ip_rt_init` 函数，为每个 CPU 分配一个 `rt_cache_stat` 数据结构的拷贝，每个 CPU 各自维护自己的统计数据。`rt_cache_stat` 字段使用 `RT_CACHE_STAT_INC` 宏来递增，这个宏透明地更新相应 CPU 的计数值。“rt_cache_stat 结构”一节中详细描述了 `rt_cache_stat` 中的字段。

这些统计内容可以通过转储 */proc/net/stat/rt_cache* 文件的内容来读取（参见“/proc/net 和 /proc/net/stat 目录”一节）。但得到的输出信息没有被格式化，因而不容易阅读。为了得到格式化的输出信息，可以使用 IPROUTE2 包中的 *lnstat* 工具。

通过/proc文件系统调整

IPv4路由子系统使用*/proc*文件系统以只读方式输出一些内部数据结构（例如，路由缓存），而将其他数据结构以读写方式输出以便调整。

图36-4说明了这些文件的位置及注册这些文件的函数。没有对应创建函数的文件则是在系统引导时，由`sysctl_init`函数静态定义。

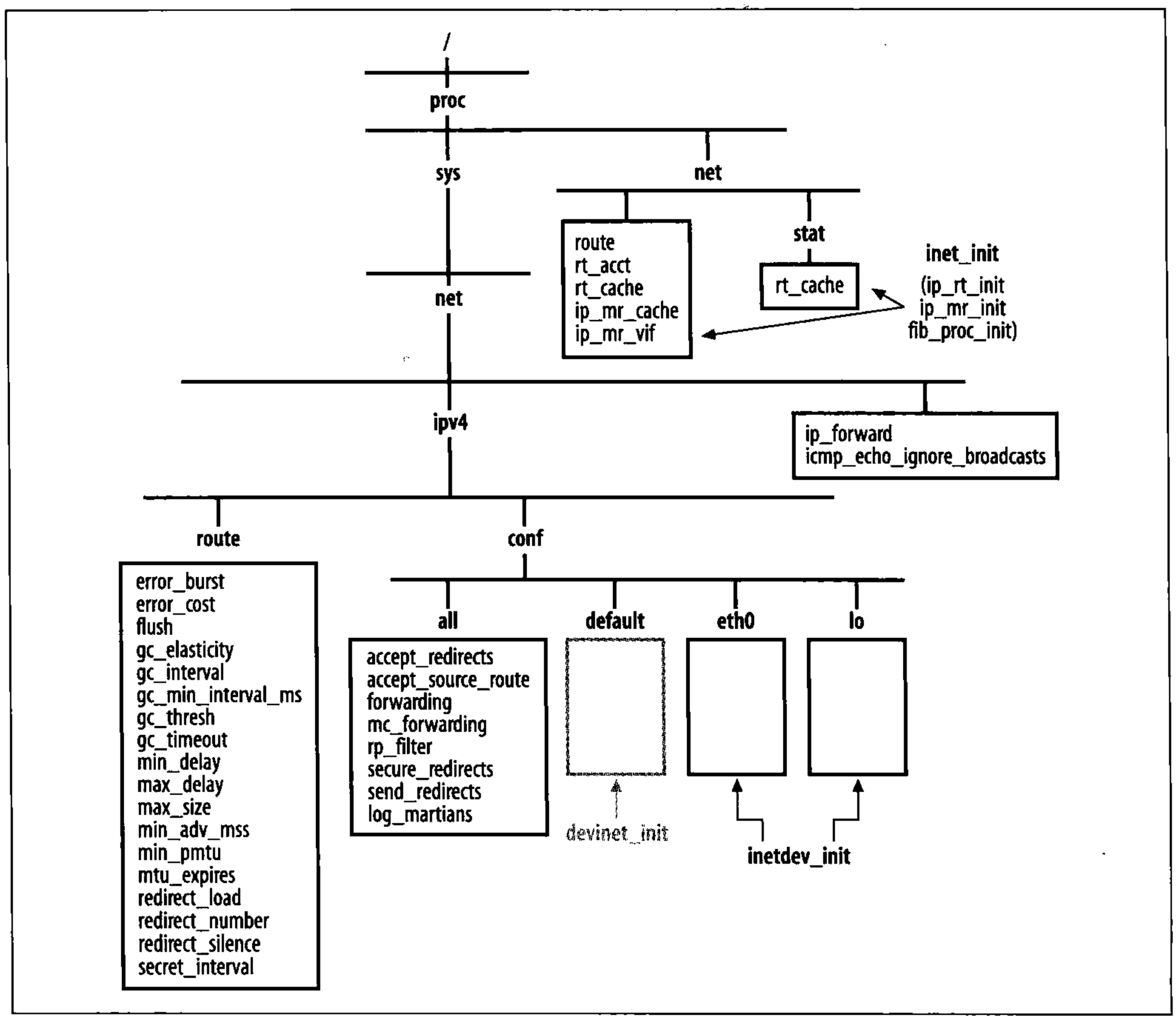


图36-4：IPv4路由子系统使用的/proc文件

/proc/sys/net/ipv4/

/proc/sys/net/ipv4/conf

/proc/sys/net/ipv4/route

这些目录下的文件输出内部数据结构用于调整路由，所以这些目录下的文件是可写

的。后面的章节列出这些目录下的文件，相关的内核变量以及这些变量的默认值（注 4）。

/proc/net/

/proc/net/stat

这些目录下的文件并不是用于调整路由，而是执行内核函数以得到某种信息。参见“/proc/net 目录和 /proc/net/stat 目录”一节。

/proc/sys/net/ipv4 目录

这个目录包含了许多文件，但仅有一些用于路由子系统：

ip_forward

包含了一个用于全局开启和关闭IP转发的布尔类型标识，这个全局的IP转发标识的值，可以被每个设备上的IP 转发标识值所覆盖（参见“开启和关闭转发”一节）。

icmp_echo_ignore_broadcasts

一个ICMP调整参数。在第三十章“定向广播”一节中介绍过该参数，它被路由代码用于确定如何来处理定向广播。广播过滤只能在这里开启和关闭，并且只能够是全局生效（不是针对每个设备）。

参见表 36-3 对这些文件的总结。

表 36-3：用于调整路由子系统的 /proc/sys/net/ipv4/ 文件

内核变量名	文件名	默认值
ipv4_devconf.forwarding[a]	*ip_forward*	0
sysctl_icmp_echo_ignore_broadcasts	*icmp_echo_ignore_broadcasts*	0

a： 参见第十九章对 ipv4_devconf 数据结构的描述。

/proc/sys/net/ipv4/route 目录

IPv4 路由子系统使用这个目录下的所有文件。下面按照功能分组来描述这些文件：

注 4： 注意由内核设置的默认值可能与 Linux 系统引导时得到的默认值不同，原因在于每一个 Linux 发行版本，可以在 Linux 系统引导时，利用初始化文件和脚本来自由改变每个 sysctl 变量的默认值。例如，可以参见 */etc/sysctl.conf* 文件。而且，不同版本的内核可能使用不同的默认值。

error_burst

error_cost

用于实现对 ICMP_UNREACHABLE 消息的限速。参见第三十五章“路由失败”一节。

max_size

gc_thresh

gc_min_interval

gc_timeout

gc_elasticity

gc_interval

用于路由缓存垃圾收集算法，参见第三十三章中的描述。

flush

min_delay

max_delay

用于控制路由缓存的刷新。

与该目录下的其他文件不同，只有 flush 是可写的（注 5）并触发一个动作；它不是一个简单的调整参数。当用户将 *n* 写入该文件时，函数 ipv4_sysctl_rtcache_flush 被启用，将在 *n* 秒后对路由表缓存进行一次刷新。当一个负值写入 *flush* 中时，内核调度在默认延迟 min_delay 之后进行一次刷新。max_delay 是在用户调度进行刷新和内核实际刷新缓存之间的最长时间间隔。参见第三十三章“刷新路由缓存”一节。

min_adv_mss

这个值与 TCP 最大报文段长度（MSS）参数相关。每个路由有一个相关的 MSS 值。当一个 dst_entry 的下一跳（使用 rt_set_nexthop）被初始化时，在调用 rt_intern_hash 将它添加到路由表缓存之前，MSS 被初始化为流出设备的 MTU 和 min_adv_mss 二者的最大值，参见 tcp_advertise_mss 中的注释和 rt_set_nexthop 中对该变量的初始化。

min_pmtu

mtu_expires

当与一个路由缓存表项相关的 PMTU 变化时，路由缓存的调度在 mtu_expires 秒之后到期。参见第三十章“使路由缓存项过期的事件范例”一节。

min_pmtu 是路径 MTU 发现协议可以为路由项设置的最小 PMTU 值。

注 5：　这个文件实际上具有读取权限，但如果试图读取文件内容，内核将发出告警信息。

redirect_load

redirect_number

redirect_silence

用于实现对ICMP_REDIRECT消息的限速。参见第三十三章“出口ICMP REDIRECT限速”一节。

secret_interval

路由缓存每隔secret_interval/HZ秒被定期冲洗。参见第三十三章“刷新路由缓存”一节。

表36-4列出了内核变量和默认值（注6）。

表36-4：用于调整路由子系统的/proc/sys/net/ipv4/route文件

内核变量名	文件名	默认值
ip_rt_error_burst	*error_burst*	5 * HZ
ip_rt_error_cost	*error_cost*	HZ
flush_delay	*flush*	N/A[a]
ip_rt_gc_elasticity	*gc_elasticity*	8
ip_rt_gc_interval	*gc_interval*	60 * HZ
ip_rt_gc_min_interval	*gc_min_interval_msb*[b]	HZ / 2
ipv4_dst_ops.gc_thresh	*gc_thresh*	依赖RAM[c]
ip_rt_gc_timeout	*gc_timeout*	RT_GC_TIMEOUT (300 * HZ)
ip_rt_min_delay	*min_delay*	2 * HZ
ip_rt_max_delay	*max_delay*	10 * HZ
ip_rt_max_size	*max_size*	依赖RAM
ip_rt_min_advmss	*min_adv_mss*	256
ip_rt_min_pmtu	*min_pmtu*	512+20+20[d]
ip_rt_mtu_expires	*mtu_expires*	10 * 60 * HZ

注6：大多数表示周期性时间的参数，在配置时是以秒为单位，但存储时以jiffies（秒数×HZ）为单位。当通过转储文件内容来读取这些值时，你得到的值可能是以秒为单位，也可能以jiffies为单位，这依赖于内核是用什么函数来转储这些值（例如，proc_handler函数）。例如，proc_dointvec输出的就是内核中的值（假设为整数值），而proc_ dointvec_jiffies把以jiffies为单位的值（即滴答）转换为秒来显示。

表 36-4：用于调整路由子系统的 /proc/sys/net/ipv4/route 文件（续）

内核变量名	文件名	默认值
ip_rt_redirect_load	*redirect_load*	HZ / 50
ip_rt_redirect_number	*redirect_number*	9
ip_rt_redirect_silence	*redirect_silence*	((HZ/50)<<(9+1))
ip_rt_secret_interval	*secret_interval*	10 * 60 * HZ

a：参见本节前面对 flush 的描述。

b：有另外一个文件 *gc_min_interval* 与同一内核变量相关，这个文件被废弃并将删除。

c：根据引导期间 hash 表的大小初始化，其值依赖于安装 RAM 的数量。

d：TCP 依照 RFC 793 和 RFC 1112 使用 512 作为 MSS 的默认值，当没有选择 IP 和 TCP 时，附加的 20+20 是 IP 报头和 TCP 表头的大小。

/proc/sys/net/ipv4/conf 目录

这个目录包含的文件可用于调整IPv4、IPsec和ARP协议，以及控制每一个设备的路由。与协议相关的参数在相关章节中已经讨论过，所以在本章我们只讨论用于调整路由参数的文件。

在 */proc/sys/net/ipv4/conf* 目录下，每个注册的网络设备（包括loopback设备）都有各自的子目录，在这子些子目录下按次序包含有每个调整参数对应的文件。这使用户可以对每个设备配置各种协议的路由参数。每个目录下包含的文件相同。所有这些参数在内核中被组合到一个类型为 ipv4_devconf 的数据结构内，该结构定义在 *include/linux/inetdevice.c* 文件中，并在表 36-5 中列出。各个参数的默认值输出到 */proc/sys/net/ipv4/conf/default* 目录下对应的文件中。

表 36-5：用于调整路由子系统的 /proc/sys/net/ipv4/conf 子目录文件

内核变量（ipv4_devconf 的字段）	文件名	默认值
accept_redirects	*accept_redirects*	1
accept_source_route	*accept_source_route*	1
forwarding	*forwarding*	0
mc_forwarding	*mc_forwarding*	0
Rp_filter	*rp_filter*	0
secure_redirects	*secure_redirects*	1

表 36-5：用于调整路由子系统的 /proc/sys/net/ipv4/conf 子目录文件（续）

内核变量（ipv4_devconf 的字段）	文件名	默认值
shared_media	*shared_media*	1
send_redirects	*send_redirects*	1
log_martians	*log_martians*	0
tag（没有使用）	*tag*	0

特殊的子目录

在*/proc/sys/net/ipv4/conf*目录下，除了每个设备对应一个目录以外，还有两个特殊的子目录：

default

用户没有明确配置的所有参数是用这个目录下的默认值来初始化，这些值由内核通过 ipv4_devconf 的一个独立实例：ipv4_devconf_dflt 来维护（参见表 36-5）。

all

这个目录用于全局配置（即用户在这里所配置的值应用于所有设备）。这些值的维护也是由内核通过一个独立的数据结构 ipv4_devconf 维护，该变量名称与其结构类型相同。

*default*目录和每个设备对应的目录是在调用 devinet_sysctl_register 时创建的。*all*目录是静态定义的（参见*net/ipv4/devinet.c*文件中 devinet_sysctl_table 的定义）。

当系统引导时，路由代码调用 devinet_init 来初始化路由，devinet_init 又调用 devinet_sysctl_register 来注册*default*目录（参见第三十二章“路由子系统初始化”一节）。因为这个函数被 inetdev_init 调用，所以当每个设备初始化时都要调用一次该函数（当设备上配置第一个 IPv4 地址时）。

特殊子目录的使用

将每个设备上的配置值和全局配置值进行组合时，以及将 *all* 目录下变量的改变传播给所有设备时，不同的功能呈现出不同的行为。例如：

- 对某一些字段，将设备上配置的值和全局配置值做与（AND）操作。在这种情况下，只有当全局配置和设备配置的值都被开启时，该功能才开启。
- 对某一些字段，将设备上配置的值和全局配置值做或（OR）操作。这时，只要一个文件中的值为开启则该功能就开启。

- 对某一些字段，不考虑全局值。

那么，要使某一个功能开启，是如何来利用这些文件呢？

对每个参数有一个对应的宏IN_DEV_*XXX*，该宏定义在*include/linux/inetdevice.h*文件中，使用该宏可得出给定设备的当前操作状态。这些宏将设备上的IPv4配置块作为输入参数，这里指的设备是结构类型为in_device的实例。可以查看这些宏得出每个参数是用什么标准（相与、相或或没有操作）来组合设备配置和全局配置。这里针对三种情况分别给出一个例子：

```
#define IN_DEV_RPFILTER(in_dev) \
(ipv4_devconf.rp_filter && (in_dev)->cnf.rp_filter)

#define IN_DEV_PROXY_ARP(in_dev) \
(ipv4_devconf.proxy_arp || (in_dev)->cnf.proxy_arp)

#define IN_DEV_MEDIUM_ID(in_dev) ((in_dev)->cnf.medium_id)
```

前面例子中使用的逻辑并不是IN_DEV_*XXX*宏实现的唯一方式。例如，IN_DEV_RX_REDIRECTS宏就更为复杂，它是对多个参数的封装，而不只是两个值的相与或相或条件。

还有一个情况需要考虑。对一些参数，*all*目录下的文件变化立即传播到所有设备对应的子目录（不是由于调用IN_DEV_*XXX*宏引起）。在这种情况下，相关的IN_DEV_*XXX*宏不需要检查全局配置值。参见“开启和关闭转发”一节中的例子。

文件描述

这里简要描述在表36-5中列出的文件：

accept_redirects

send_redirects

在第三十一章描述了ICMP重定向消息，该消息由路由器发送给主机，通知它们路由不是最优。accept_redirects是一个布尔类型标识，可用于在一个接口上开启或关闭ICMP重定向处理（注7）。send_redirects被用于反方向：当值为真，且检测到非最优路由的必要条件时，允许系统产生ICMP重定向消息。

accept_source_route

可以利用这个标识来开启和关闭IP源路由（Source Routing）选项。当为关闭时，ip_rcv_finish丢弃所有携带有该选项的IP封包。IP选项已经在第十八章中讨论。

注7：也可以参见“开启和关闭转发”一节。

forwarding

mc_forwarding

这些是布尔类型标识，分别用于开启和关闭单播与多播的转发。当使用mc_forwarding时，内核编译时必须使用必要的多播选项。

rp_filter

当这个标识为真时，如果某个入口封包的源IP地址通过非对称路由（asymmetric route）可达（依据本地主机的路由表），则丢弃该封包。参见第三十一章“反向路径过滤”一节。

secure_redirects

shared media

当secure_redirects被设置时，只有当消息中建议的网关在本地是一个已知网关时，ICMP_REDIRECT消息才被接收。

通常，ICMP_REDIRECT消息中建议使用一个新的下一跳，如果该下一跳的IP地址与当前的下一跳不在同一网段内则被拒绝，参见RFC 1122中的描述。然而，有些情况下接受不在同一网段内的新下一跳是有意义的。当shared_media为真时就接受这些ICMP_REDIRECT消息。RFC 1620清晰地解释了为什么在某些情况下这种选项是有意义的。

该功能的更多信息参见第三十一章“ICMP_REDIRECT消息”一节。

log_martians

当该标识被设置时，内核接收到的封包中含有非法的IP地址时产生日志消息。参见第三十一章“Verbose监控”一节。

/proc/net和/proc/net/stat目录

*/proc/net*目录下提供一些文件，当试图转储这些文件内容时执行内核处理程序。下面逐一描述这些文件：

route

rt_cache

可以读取这两个文件的内容，分别转储路由表（ip_fib_main_table）和路由缓存。它们并不显示用户定义的路由表的内容，这些路由表是当内核支持策略路由时才创建的。IP地址以十六进制格式输出。

stat/rt_cache

一组统计数据。参见“统计数据”和“rt_cache_stat结构”一节。

rt_acct

由在第三十一章介绍的基于路由表的分类器所收集的计费信息。要想得到更好格式的输出信息，可使用 IPROUTE2 包中的 *rtacct* 命令。

ip_mr_cache

ip_mr_vif

由多播路由使用（本书中不讨论）。

表 36-6 汇总了这些文件和相关内核处理程序之间的关联。

表 36-6：内核处理程序通过路由子系统使用的 /proc/net 目录中的文件

文件名	内核文件定义的位置
route	*net/ipv4/fib_hash.c (fib_proc_init)*
rt_cache	*net/ipv4/route.c (ip_rt_init)*
rt_acct	*net/ipv4/route.c (ip_rt_init)*
ip_mr_cache	*net/ipv4/ipmr.c (ip_mr_init)*
ip_mr_vif	*net/ipv4/ipmr.c (ip_mr_init)*
stat/rt_cache	*net/ipv4/route.c (ip_rt_init)*

如图 36-4 所示，*/proc/net* 和 */proc/net/stat* 目录下的文件都是由 inet_init 在如 ipv4_proc_init 和 ip_init 等函数的帮助下间接创建的。inet_init 标记为 module_init 宏，所以它是在系统引导时被执行（参见第七章）。

开启和关闭转发

本章前面提到，内核通过 */proc* 输出用于开启和关闭 IP 转发的参数，这些参数不仅有全局参数，而且对每个设备也有一个参数。我们在本章中只讨论 IPv4 转发。

虽然管理员可以改变全局的转发状态，但实际上并没有全局转发状态。路由代码只使用每个设备上的转发状态：改变全局配置只是一次性将该改变应用到所有设备的一种简便方法。尤其是当内核接收到一个目的地址不属于本地系统的 IP 封包时，内核根据接收接口所配置的转发状态，决定该报文是转发还是丢弃。这不是一个依赖于全局转发状态的决策，也不依赖向着其目的地发送封包的设备转发状态。

理解每个设备上的配置和全局配置之间的关系是很重要的，以便在你改变这些配置时能够了解系统的行为。下面列出 */proc* 目录下相关的文件：

/proc/sys/net/ipv4/conf/device_name/forwarding

对 device_name 设备开启和关闭转发。值为零时表示关闭，其他值表示开启。

/proc/sys/net/ipv4/conf/all/forwarding

对这个文件的改变将应用到所有的网络设备（包括不是为UP状态的设备），但不影响以后注册的设备的转发状态。

/proc/sys/net/ipv4/conf/default/forwarding

对没有明确配置的设备，这是其默认转发状态。与前面的文件不同，其值只影响以后注册的设备的转发状态（而不是已经存在的设备）。

/proc/sys/net/ipv4/ip_forward

改变这个文件与改变*/proc/sys/net/ipv4/conf/all/forwarding*文件有相同的效果。可以把前者面看作是后者的别名。

*forwarding*文件的变化是由devinet_sysctl_forward来处理，其内部区分三种情况。*ip_forward*文件的变化是由 ipv4_sysctl_forward来处理。每当至少有一个设备的转发状态发生变化时，用 rt_cache_flush来刷新路由缓存。

改变*/proc/sys/net/ipv4/conf/all/forwarding*文件或是*/proc/sys/net/ipv4/ip_forward*文件，都将触发执行 inet_forward_change，其流程为：

1. 更新 ipv4_devconf.accept_redirect 配置参数

   这样做是为了强制规则：只有主机才能接收ICMP重定向消息，而路由器不能够接收。如果全局转发状态变为开启，意味着系统现在被视为一台路由器，因而必须禁止可敬的（honouring）ICMP重定向的默认配置（管理员可以根据需求重新开启该配置参数）。

2. 更新默认转发状态

   注意，改变全局转发配置将强行改变默认转发配置，但反之不成立。

3. 更新所有设备的转发状态

本书此部分中的数据结构功能

在第三十二章“主要的数据结构”一节中，简要介绍了主要的数据结构，第三十四章中的图 34-1 有助于读者理解它们之间的关系。本节详细描述每一种数据结构类型。图 36-5 显示了定义每种数据结构的文件。

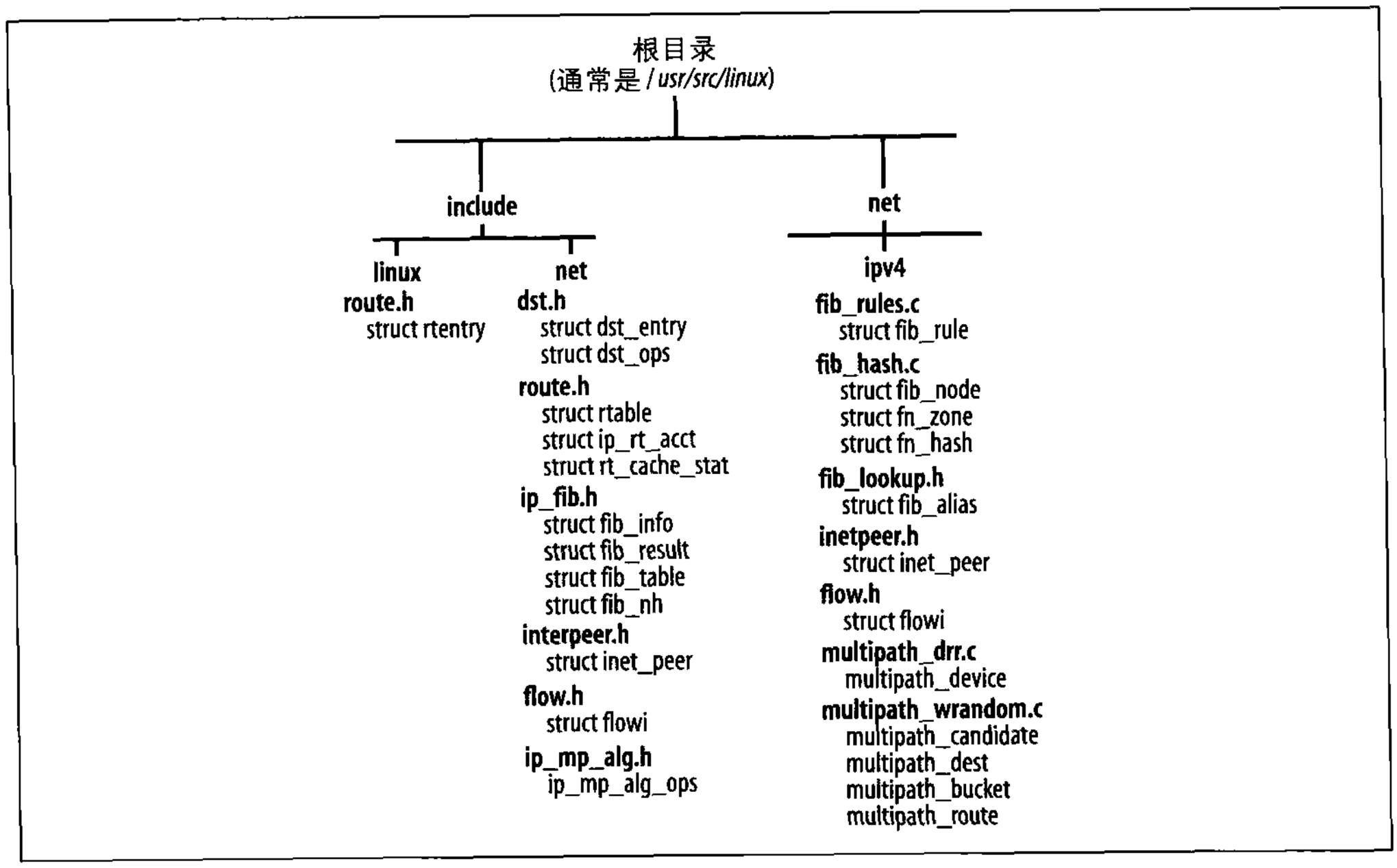


图 36-5：内核文件中的数据结构分布

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：

```
unsigned char tb_id
```

路由表标识符。在 *include/linux/rtnetlink.h* 文件中可以找到预先定义的值的列表 rt_class_t，例如 RT_TABLE_LOCAL。

```
unsigned tb_stamp
```

未被使用。

```
int (*tb_lookup)(struct fib_table *tb, const struct flowi *flp, struct
fib_result *res)
```

这个函数被第三十五章中描述的 fib_lookup 函数调用。

```
int (*tb_insert)(struct fib_table *table, struct rtmsg *r,struct
kern_rta *rta, struct nlmsghdr *n, struct netlink_skb_parms *req)
int (*tb_delete)(struct fib_table *table, struct rtmsg *r, struct
kern_rta *rta, struct nlmsghdr *n, struct netlink_skb_parms *req);
```

tb_insert 被 inet_rtm_newroute 和 ip_rt_ioctl 调用，以处理用户空间的 *ip*

route add/change/replace/prepend/append/test 和 *route add* 命令。类似地，tb_delete 被 inet_rtm_delroute（对 *ip route del* ... 命令作出的响应）和 ip_rt_ioctl（对 *route del* ... 命令作出的响应）调用，用于从路由表中删除一个路由。两者也被 fib_magic（参见“内核插入路由项：fib_magic 函数”一节）调用。

```
int (*tb_dump)(struct fib_table *table, struct sk_buff *skb,
  struct netlink_callback *cb)
```

转储路由表的内容。在处理如“ip route get...”等用户命令时被启用。

```
int (*tb_flush)(struct fib_table *table)
```

将设置有 RTNH_F_DEAD 标识的 fib_info 结构删除，参见第三十三章“垃圾回收”一节。

```
void (*tb_select_default)(struct fib_table *table, const struct flowi
*flp, struct fib_result *res)
```

选择一个默认路由，参见第三十五章“默认网关选择”一节。

```
unsigned char tb_data[0]
```

指向该结构的尾部。当该结构被分配用作一个更大结构的一部分时，这种做法是很有用的，因为代码可以在该结构之后紧接着指向外部数据结构的一部分，参见第三十四章的图 34-1。

fn_zone 结构

一个 *zone*（区域）是一组有着相同子网掩码长度的路由项。路由表中的路由项按照区域来组织，参见第三十二章中的描述。区域是由 fn_zone 结构来定义的，该结构包含以下字段：

```
struct fn_zone *fz_next
```

将活动区域（active zones）（即至少有一个路由项的区域）链接在一起的指针。该链表的头部保存在 fn_zone_list 中，fn_zone_list 是 fn_hash 数据结构的一个字段。

```
struct hlist_head *fz_hash
```

指向存储该区域中路由项的 hash 表。

```
int fz_nent
```

在该区域中路由项的数目（即在该区域的 hash 表中 fib_node 实例的数目）。这个值可以用于检查是否需要改变该 hash 表的容量（参见第三十四章“基于网络掩码的 Hash 表容量的动态调整”一节）。

int fz_divisor

hash表fz_hash的容量（bucket的数目），参见第三十四章“基于网络掩码的hash表容量的动态调整”一节。

u32 fz_hashmask

这是fz_divisor-1，提供该字段可以用廉价的二进制按位与操作，替代昂贵的模操作以计算一个值对 fz_divisor 取模操作。因为 *n*%fz_divisor 的结果与 *n*&fz_hashmask的结果相同（例如100%16 = 100&15），而二进制按位与操作所花费的CPU时间要少。

int fz_order

在网络掩码fz_mask中设置的位数目（所有连续），在代码中也用prefixlen来表示。例如，给定的网络掩码255.255.255.0所对应的 fz_order 为24。

u32 fz_mask

用fz_order构建的子网掩码。例如，fz_order的值取3，则生成一个二进制表示的fz_mask 11100000.00000000.00000000.00000000，其十进制表示为224.0.0.0。

在该结构中有两个宏，分别用于访问 fz_hashmask 和 fz_mask 字段：

```
#define FZ_HASHMASK(fz) ((fz)→fz_hashmask)
#define FZ_MASK(fz) ((fz)→fz_mask)
```

fib_node 结构

内核路由项中每一个唯一的目的网络对应一个 fib_node实例。目的网络相同但其他配置参数不同的路由项共享同一个 fib_node 实例。下面逐一描述该结构中的字段：

struct hlist_node fn_hash

fib_node元素是用hash表来组织的。这个指针用于把hash表中的一个bucket内所有发生冲突的元素链接在一起。

struct list_head fn_alias

每个fib_node结构与包含一个或多个fib_alias结构的链表相关联。fn_alias是一个指向该链表头部的指针。

su32 fn_key

这是路由的前缀（或网络地址，用路由的网络掩码字段来表示）。该字段被用作查找路由表时的搜索关键字。参见第三十四章“hash表组织的基本结构”一节。

fib_alias 结构

fib_alias实例是用来区分目的网络相同但其他配置参数（除了目的地址以外）不同的路由项。下面逐一描述该结构中的字段：

struct list_head fa_list

用于链接与同一个 fib_node 结构相关联的所有 fib_alias 实例。

struct fib_info *fa_info

该指针指向一个fib_info实例，该实例存储着如何处理与该路由相匹配封包的信息。

u8 fa_tos

路由的服务类型（TOS）位字段（bitfield）。当该值为零时意味着还没有配置TOS，所以在路由查找时任何值都可以匹配。不要将 fa_tos 字段与 fib_rule结构中的 r_tos字段相混淆。fa_tos字段允许用户在TOS中对个别路由表项指定条件，而 fib_rule 结构中的 r_tos 字段是用户在 TOS 中指定策略规则条件。

u8 fa_type

参见“rtable 结构”一节中对 rt_type 字段的描述。

u8 fa_scope

路由表项的 scope。参见第三十章“scope”一节。

u8 fa_state

一些标识的位图（bitmap of flags）。迄今为止只定义了下面这一个标识：

FA_S_ACCESSED

无论何时查找访问到 fib_alias实例，就用该标识来标记。当一个 fib_node 数据结构改变并应用时这个标识很有用：它用于决定是否应当刷新路由缓存。如果 fib_node已经被访问，可能就意味着在该路由发生变化时需要清除路由缓存内的表项，从而触发一次刷新。

fib_info 结构

在前面几节已经描述过，定义一个路由项的参数包含在fib_node结构和fib_alias结构的组合体中。如下一跳网关等重要的路由信息则存储在一个 fib_info结构内。下面逐一描述该结构中的字段：

struct hlist_node fib_hash
struct hlist_node fib_lhash

用于将fib_info数据结构插入到第三十四章“fib_info结构的组织”一节中描述的两个hash表中。

int fib_treeref
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。

int fib_dead

标记路由项正在被删除的标识。当该标识置1时，警告该数据结构将被删除而不能再使用。参见第三十四章“删除路由”一节。

unsigned fib_flags

RTNH_F_*XXX*标识的设置在表36-7中列出。当前使用的唯一标识是RTNH_F_DEAD。当与一个多路径路由项相关联的所有fib_nh结构都设置了RTNH_F_DEAD标识时，设置该标识（参见第三十二章“常用的辅助程序与宏”一节）。

表36-7：fib_nh的nh_flags字段值

标识	描述
RTNH_F_DEAD	主要被多路径代码用于追踪失效的下一跳，参见第三十二章的“常用的辅助程序与宏”一节中对fib_sync_down的描述
RTNH_F_PERVASIVE	该标识期望用来标记需要递归查找的表项，但是目前还未被使用。最新的IPROUTE2版本不接受任何pervasive关键字
RTNH_F_ONLINK	当该标识被设置时，要求内核不检查下一跳地址是否相连（即不检查下一跳地址通过流出设备是否可达）。例如，当定义隧道虚拟路由设备时，onlink关键字用于设置和使用该标识

int fib_protocol

已安装的路由协议。该字段的可能取值RTPROT_*XXX*定义在*include/linux/rtnetlink.h*文件中，并在表36-8和表36-9中列出了这些值（这两个表中缺少了ROUTED，因为它不使用Netlink接口与内核通信）。参见第三十一章“路由协议守护进程”一节中对这些协议的概述。

fib_protocol大于RTPROT_STATIC的值只用于不是由内核产生的路由项（即这些协议由用户空间的路由守护进程生成）。

使用该字段的一个例子是，当涉及内核时允许路由守护进程的操作被限定在它们自己产生的路由项。更多细节可参见第三十一章。

表 36-8：内核使用的 fib_protocol 值

值	描述
RTPROT_UNSPEC	无效的字段
RTPROT_REDIRECT	由 ICMP 重定向安装的路由，当前的 IPv4 不使用
RTPROT_KERNEL	由内核安装的路由，参见“内核插入路由：fib_magic 函数
RTPROT_BOOT	由用户空间命令如 *ip route* 和 *route* 安装的路由
RTPROT_STATIC	由管理员安装的路由，没有使用

表 36-9：用户空间使用的 fib_protocol 值

值	描述
RTPROT_GATED	由 GateD 添加路由
RTPROT_RA	由 RDISC（IPv4）和 ND（IPv6）路由器公告的路由。这是一种 RFC 1256 定义的 ICMP 路由发现协议的机制，使主机发现相邻的路由器。*rdisc* 是 *iputils* 软件包的一部分，用于实现用户空间的 ICMP 路由器发现消息
RTPROT_MRT	由多线程路由工具包（Multi-Threaded Routing Toolkit, MRT）添加的路由
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

u32 fib_prefsrc

首选源 IP 地址。参见第三十五章“选择源 IP 地址”一节。

u32 fib_priority

路由的优先级。值越小则优先级越高。它的值可以用 IPROUTE2 包中的 metric/priority/preference 关键字来配置。当没有明确设置时，内核将它的值初始化为默认值 0。

u32 fib_metrics[RTAX_MAX]

当配置路由时，*ip route* 命令允许指定一组规格（metric）。fib_metrics 是存储这组规格的向量。没有明确设置的规格在初始化时被设置为 0。参见第三十章“路由的基本要素”一节中列出的可用的规格，表 36-10 列出该小节中列出的规格与在 *include/linux/rtnetlink.h* 文件中定义的相关内核符号 RTAX_*XXX* 的之间的关系。

表 36-10：路由规格

规格	内核符号
没有规格	RTAX_LOCK
路径 MTU	RTAX_MTU
最大的公告窗口	RTAX_WINDOW
往返时间（RTT）	RTAX_RTT
RTT 变量	RTAX_RTTVAR
慢启动门限值	RTAX_SSTHRESH
拥塞窗口	RTAX_CWND
最大的段大小	RTAX_ADVMSS
最大的重定序（Reordering）	RTAX_REORDERING
默认的存活时间（TTL）	RTAX_HOPLIMIT
初始的拥塞窗口	RTAX_INITCWND
不是一种规格	RTAX_FEATURES

`int fib_power`

仅当内核编译为支持多路径时，该字段才是 fib_info 数据结构的一部分。参见第三十一章“多路径路由背后的概念”一节。

`struct fib_nh fib_nh[0]`

`int fib_nhs`

fib_nh 是结构类型为 fib_nh 的一个可变长向量，fib_nhs 是该向量的大小。仅当内核支持多路径功能时，fib_nhs 才可能大于 1。参见第三十一章“多路径路由背后的概念”一节和第三十四章中的图 34-1。

`u32 fib_mp_alg`

多路径缓存算法。在第三十一章“支持多路径缓存”一节介绍的多路径缓存算法的标识符 IP_MP_ALG_*XXX*，列在 *include/linux/ip_mp_alg.h* 文件中。仅当内核编译为支持多路径缓存时，该字段才是 fib_info 数据结构的一部分。

```
#define fib_dev fib_nh[0].nh_dev
```

这个宏用于访问 fib_nh 向量中第一个 fib_nh 实例的 nh_dev 字段。参见第三十四章中的图 34-1。

```
#define fib_mtu fib_metrics[RTAX_MTU-1]
#define fib_window fib_metrics[RTAX_WINDOW-1]
```

```
#define fib_rtt fib_metrics[RTAX_RTT-1]
#define fib_advmss fib_metrics[RTAX_ADVMSS-1]
```

这些宏用于访问 fib_metrics 向量中指定的元素。

fib_nh 结构

对每一个下一跳，内核需要跟踪的信息与IP地址相比更多。fib_nh结构在下面字段中存储这些额外的信息：

struct net_device *nh_dev

这是与设备标识nh_oif（后面描述）相关联的net_device数据结构。因为设备标识符和指向net_device结构的指针都需要利用（在不同的环境下），虽然利用其中任何一项就可以得到另一项，但这两项都保存在fib_nh结构中。

struct hlist_node nh_hash

用于将fib_nh数据结构插入到在第三十四章“下一跳路由器结构的组织”一节中描述的hash表中。

struct fib_info *nh_parent

该指针指向包含该fib_nh实例的fib_info结构，参见第三十四章中的图34-1。

unsigned nh_flags

在 *include/linux/rtnetlink.h* 文件中定义的一组RTNH_F_*XXX*标识，在本章前面的表36-7中列出。

unsigned char nh_scope

用于获取下一跳的路由范围。在大多数情况下为RT_SCOPE_LINK。该字段由fib_check_nh初始化。

int nh_weight

int nh_power

仅当内核编译为支持多路径时，这两个字段才是fib_nh数据结构的一部分，在第三十一章“多路径路由背后的概念”一节对它们有详细描述。nh_power是由内核初始化，nh_weight是由用户利用关键字weight来设置。

__u32 nh_tclassid

仅当内核编译为支持基于路由表的分类器时，该字段才是fib_nh数据结构的一部分。它的值是利用realms关键字来设置。参见第三十五章“策略路由与基于路由表的分类器”一节。

int nh_oif

出口设备标识符。它是利用关键字oif和dev来设置的。

u32 nh_gw .

下一跳网关的IP地址，由关键字via提供。注意在NAT情况下，它表示NAT路由器向外部世界公告的地址，以及在NAT路由器向内部网中的主机发送回应之前而向外部世界回应的地址。例如，命令*ip route add nat 10.1.1.253/32 via 151.41.196.1*将设置nh_gw为151.41.196.1。注意在2.6内核中路由代码已经不再支持NAT，即原来众所周知的FastNAT。

fib_rule结构

利用*ip rule*命令来配置策略路由规则（也称为策略）。如果你的Linux系统上安装了IPROUTE2软件包，就可以输入*ip rule help*来查看该命令的语法。策略存储在fib_rule结构中，下面逐一描述该结构的字段：

struct fib_rule *r_next

将这些fib_rule结构链接到一个包含所有fib_rule实例的全局链表内（参见第三十五章中的图35-8）。

atomic_t r_clntref

引用计数。该引用计数的递增由fib_lookup（只在策略路由版本中）完成，这就解释了为什么在每次路由查找成功后总是需要调用fib_res_put（递减该引用计数）。

u32 r_preference

路由规则的优先级。当管理员利用IPROUTE2软件包添加一个策略时，可以使用关键字priority、preference和order来配置。如果没有明确配置，内核为其分配一个优先级，该值比用户添加的最后一个规则的优先级小一个级别（参见inet_rtm_newrule）。优先级0，0x7FFE和0x7FFF是预留给由内核添加的特定规则（参见第三十五章“策略路由时的fib_lookup”一节，以及在*net/ipv4/fib_rules.c*文件中定义的三个默认规则（local_rule、main_rule 和default_rule）。

unsigned char r_table

路由表标识符，范围从0～255。当用户没有指定时，IPROUTE2采用以下的默认原则：当用户命令是添加一条规则时使用RT_TABLE_MAIN，其他情况使用RT_TABLE_UNSPEC（例如，删除一条规则）。

unsigned char r_action

该字段允许的取值是定义在*include/linux/rtnetlink.h*文件中的rtm_type枚举值（RTN_UNICAST等）。这些值的含义在“rtable结构”一节中描述。

当配置一条规则时，用户可以明确使用type关键字来设定该字段。如果用户没有明确配置，IPROUTE2在添加规则时设置该字段的值为RTN_UNICAST，在其他情况下设置为RTN_UNSPEC（例如，当删除规则时）。

unsigned char r_dst_len

unsigned char r_src_len

以位为单位的目的IP地址与源IP地址的长度。它们用于计算r_srcmask和r_dstmask。如果这两个字段未初始化则设置为0。

u32 r_src

u32 r_srcmask

分别表示只有从该IP地址和网络掩码组成的源网络发送的封包才能被接受。

u32 r_dst

u32 r_dstmask

分别表示只能向该IP地址和网络掩码组成的目的网络发送封包。

u32 r_srcmap

该字段用用户空间关键字nat和map-to来设置，在路由NAT实现中使用。由于不再支持路由NAT，所以该字段也不再被使用。参见第三十二章“最近去掉的选项”一节。

u8 r_flags

一组标识。当前没有使用。

u8 r_tos

IP报头中的TOS字段。包含该字段的原因是规则的定义中可以包含一个条件，该条件放在IP报头的TOS字段。

u32 r_fwmark

当内核编译为支持“使用Netfilter MARK值作为路由关键字”功能时，可以根据防火墙标签来定义规则。该字段是管理员定义一条策略规则时用fwmark关键字指定的标签。

int r_ifindex

char r_ifname[IFNAMSIZ]

r_ifname是策略应用的设备的名称。给定r_ifname，内核可以发现相关的net_device实例，并将该实例的ifindex字段拷贝到r_ifindex中。r_ifindex值取－1表示禁止该规则（参见第三十二章“对策略数据库的影响”一节）。

__u32 r_tclassid;

当内核编译为支持基于路由表的分类器时，该字段才包含在fib_rule数据结构中。其含义在第三十五章“策略路由与基于路由表的分类器”一节中描述。

int r_dead

当一个规则可用时该字段为0。当用inet_rtm_delrule删除规则时该字段被设置为1。每次调用fib_rule_put删除fib_rule数据结构的一个引用时，引用计数递减。当引用计数为0时就要释放该结构。但此时如果r_dead没有设置，则表示发生了某种错误（例如，代码对引用计数的设置不正确）。

fib_result 结构

fib_result结构被fib_semantic_match初始化为路由查找结果，更多细节可参见第三十三章和第三十五章（尤其是“语义匹配附加标准”一节）。该结构中的字段为：

unsigned char prefixlen

匹配路由的前缀长度。参见“fn_zone结构”一节中对fz_order的描述。

unsigned char nh_sel

多路径路由由多个下一跳定义。该字段标识已经被选中的下一跳。

unsigned char type

unsigned char scope

这两个字段被初始化为相匹配的fib_alias实例的fa_type和fa_scope字段的取值。

__u32 network

__u32 netmask

只有当内核编译为支持多路径缓存时，这两个字段才包含在fib_result数据结构中。参见第三十三章“加权随机算法”一节中，加权随机多路径缓存算法是如何使用这两个字段。

struct fib_info *fi

与匹配的fib_alias实例相关联的fib_info实例。

struct fib_rule *r

与前面字段不同的是，该字段由fib_lookup初始化。只有当内核编译为支持策略路由时，该字段才包含在fib_result数据结构中。

rtable 结构

IPv4使用rtable数据结构来存储缓存内的路由表项（注8）。为了转储路由缓存的内

注8：IPv6使用rt6_info，DECnet（本书中不讨论）使用dn_route。

容，可以查看*/proc/net/rt_cache*文件（参见“通过/proc文件系统调整”一节），或是发布一个*ip route list cache*或*route -C*命令。下面逐一描述该结构的字段：

`union {...} u`

这个联合用于将一个dst_entry结构嵌入到rtable结构中（参见第三十三章“hash表的组织”一节）。其中的rt_next字段用于链接分布在同一个hash表的bucket的rtable实例。

`struct in_device *idev`

该指针指向出口设备的IP配置块。注意，对送往本地的入口封包的路由，设置的出口设备为回环设备。

`unsigned rt_flags`

在该位图中可以设置的标识为在*include/linux/in_route.h*文件中定义的RTCF_XXX值，表36-11中列出了这些值。

表36-11：rt_flags的可能值

标识	描述
RTCF_NOTIFY	路由表项的任何变化，通过Netlink通知有感趣的用户空间应用程序，该选项还没有完全实现。该标识用如*ip route get 10.0.1.0/24 notify*的命令设置
RTCF_REDIRECTED	表项已经被添加以响应接收到的ICMP_REDIRECT消息。（参见ip_rt_redirect及其调用者）
RTCF_DOREDIRECT	当ICMP_REDIRECT消息必须被送回源地址时，该标识由ip_route_input_slow设置。ip_forward已在第二十章详细描述，ip_forward基于该标识和其他信息，决定是否真正送ICMP重定向消息。例如，如果该封包是被源路由的，将不产生ICMP重定向消息
RTCF_DIRECTSRC	该标识主要用于通知ICMP代码，不应当响应地址掩码请求消息。每次调用fib_validate_source检查到接收封包的源地址通过一个本地scope（RT_SCOPE_HOST）的下一跳是可达时，就设置该标识。更多细节可参见第二十五章和第三十五章
RTCF_SNAT RTCF_DNAT RTCF_NAT	IPv4不再使用这些标识。它们以前被FastNAT功能使用，该功能在2.6内核中已经被删除（参见第三十二章“最近去掉的选项”一节）
RTCF_BROADCAST	路由的目的地址是一个广播地址
RTCF_MULTICAST	路由的目的地址是一个多播地址

表 36-11：rt_flags 的可能值（续）

标识	描述
RTCF_LOCAL	路由的目的地址是一个本地地址（即本地接口上配置的某个地址）。对本地广播地址和多播地址也设置该标识（参见 ip_route_input_mc）
RTCF_REJECT	没有使用。依据 IPROUTE2 软件包的 *ip rule* 命令的语法，在该命令中有一个关键字 *reject*，但该关键字还未被接受
RTCF_TPROXY	没有使用
RTCF_DIRECTDST	没有使用
RTCF_FAST	没有使用。该标识已经被废弃，设置该标识用于标记一个路由对快速交换（Fast Switching）合法。快速交换功能已经在 2.6 内核中被废弃
RTCF_MASQ	IPv4不再使用。该标识用于标记封包来自于伪装的（masqueraded）源地址

unsigned rt_type

路由类型。它间接定义了当路由查找匹配时应采取的动作。该字段可能的取值是在 *include/linux/rtnetlink.h* 文件中定义的 RTN_*XXX* 宏，表 36-12 中列出了这些值。

表 36-12：rt_type 的可能值

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
RTN_UNICAST	该路由是一个到直接或间接（通过一个网关）的单播地址路由。当用户通过 *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。匹配的入口封包以广播方式送往本地，并且匹配的出口封包以广播方式发送出去
RTN_ANYCAST	匹配的入口封包以广播方式送往本地，匹配的出口封包以单播发送出去。IPv4 不使用
RTN_BLACKHOLE RTN_UNREACHABLE RTN_PROHIBIT RTN_THROW	这些值与特定的管理配置而不是与目的地址类型相关联。参见第三十章“路由类型与动作”小节

表 36-12：rt_type 的可能值（续）

路由类型	描述
RTN_NAT	源 IP 地址和 / 或目的 IP 地址必须被转换。因为相关的功能 FastNAT 已经在 2.6 内核中被废弃，所以该类型不再被使用
RTN_XRESOLVE	有一个外部解析器来处理该路由。该功能目前尚未实现

`__u16 rt_multipath_alg`

多路径缓存算法。该字段根据相关路由项上配置的算法来初始化（参见“fib_info 结构”一节中的 `fib_mp_alg` ）。

`__u32 rt_dst`

`__u32 rt_src`

目的 IP 地址和源 IP 地址。

`int rt_iif`

入口设备标识符。该值是从入口设备的`net_device`数据结构中提取。对本地产生的流量（因此不是从任何接口上接收到的），该字段被设置为流出设备的`ifindex`字段。不要将该字段与本章后面描述的`flowi`数据结构`fl`中的`iif`字段混淆，对本地产生的流量，`fl` 中的 `iif` 字段被设置为零（`loopback_dev`）。

`__u32 rt_gateway`

当目的主机为直连时（即在同一链路上），`rt_gateway` 匹配目的地址。当需要通过一个网关到达目的地时，`rt_gateway` 设置为被路由项识别的下一跳网关。

`struct flowi fl`

用于缓存查找的搜索关键字，参见“flowi 结构”一节中的描述。

`__u32 rt_spec_dst`

RFC 1122 中指定的目的地址，参见第三十五章“首选源地址选择”一节中的说明。

`struct inet_peer *peer`

在第十九章介绍的 `inet_peer` 结构存储有关 IP peer 的长期信息，IP peer 是该缓存路由项的目的 IP 地址对应的主机。与本地主机在最近一段时间通信的每个远端 IP 地址都有一个 `inet_peer` 结构。

dst_entry 结构

数据结构`dst_entry`用于存储缓存路由项中独立于协议的信息。L3 层协议在单独的结构中存储本协议的更多的私有信息（例如，IPv4 使用`rtable`结构）。下面逐一描述该结构的字段：

struct dst_entry *next

用于将分布在同一个hash表的bucket中具有冲突的dst_entry实例链接在一起。参见第三十三章中的图33-1。

struct dst_entry *child

unsigned short header_len

unsigned short trailer_len

struct dst_entry *path

struct xfrm_state *xfrm

IPsec代码使用这些字段。

atomic_t __refcnt

引用计数。参见第三十三章“删除DST表项”一节。

int __use

该表项已经被使用的次数（即缓存查找返回该表项的次数）。不要将这个值与rt_cache_stat[smp_processor_id()].in_hit混淆：后者（在“统计数据”一节中描述）表示针对某个CPU的全局缓存命中次数。

struct net_device *dev

出口设备（即将封包送达目的地的发送设备）。

short obsolete

用于定义该dst_entry实例的可用状态：0（默认值）表示该结构有效而且可以被使用，2表示该结构将被删除因而不能被使用，-1被IPsec和IPv6使用但不被IPv4使用。

int flags

标识集合。DST_HOST被TCP使用，表示主机路由（即它不是到网络或到一个广播/多播地址的路由）。DST_NOXFRM，DST_NOPOLICY和DST_NOHASH只用于IPsec。

unsigned long lastuse

用于记录该表项上次被使用的时间戳。当缓存查找成功时更新该时间戳，垃圾回收函数使用该时间戳来选择最合适被释放的结构。

unsigned long expires

表示该表项将过期的时间戳。参见第三十三章“过期标准”一节。

u32 metrics[RTAX_MAX]

规格向量，主要被TCP使用。该向量用fib_info->fib_metrics向量的一份拷贝来初始化（如果fib_metrics向量被定义），当需要时使用默认值。参见函数rt_set_nexthop和第三十五章，参见表36-10中列出的该向量所有可能取值的描述。

RTAX_LOCK 值需要解释一下。RTAX_LOCK 不是一个规格，而是一个位图(bitmap)：当第 *n* 位被设置时，表示已经利 *lock* 选项/关键字配置了值为枚举值 *n* 的规格。换句话说，一个类似于 *ip route add ... advmss lock ...* 的命令将设置 1«RTAX_ADVMSS 位。当一个规格被锁定时不能被协议事件改变。

unsigned long rate_last

unsigned long rate_tokens

这两个字段用于对两种类型的 ICMP 消息限速。参见第三十三章“出口 ICMP REDIRECT 限速”一节和第三十五章“路由失败”一节。

short error

当 fib_lookup API（只被 IPv4 使用）失败时，错误值被保存在 error（用一个正值）中，由后面的 ip_error 使用该值来决定如何处理本次路由查找失败（即决定产生哪一类 ICMP 消息）。

struct neighbour *neighbour

struct hh_cache *hh

neighbour 是包含下一跳 L3-to-L2 地址映射的结构，hh 是缓存的 L2 报头。详见第六部分的各章节。

int (*input)(struct sk_buff*)

int (*output)(struct sk_buff**)

分别表示处理入口封包和处理出口封包的函数，参见第三十三章“缓存查找”一节。

__u32 tclassid

基于路由表的分类器的标识。参见第三十五章“策略路由和基于路由表的分类器”一节。

struct dst_ops *ops

用于处理 dst_entry 结构的 VFT。

struct rcu_head rcu_head

处理互斥。

char info[0]

该字段用作 dst_entry 数据结构尾部的指针很有用。它只是用于占位符（a placeholder）。

dst_ops 结构

dst_ops 结构是独立于协议的缓存与使用 L3 层协议路由缓存之间的接口。参见第三十三章“DST 与调用协议间的接口”一节。下面逐一描述该结构的字段：

unsigned short family

地址簇。参见 *include/linux/socket.h* 文件中的 AF_*XXX*。

unsigned short protocol

协议标识符。参见 *include/linux/if_ether.h* 文件中的 ETH_P_*XXX*。

unsigned gc_thresh

该字段用于垃圾回收算法，指定了路由缓存的大小（即bucket的数目）。其初始化是由 ip_rt_init（IPv4路由子系统初始化函数）完成的。

int (*gc)(void)

atomic_t entries

当协议已经分配的dst_entry实例数目（entries）大于或等于门限值gc_thresh时，dst_alloc启用垃圾回收函数gc。

struct dst_entry * (*check)(struct dst_entry *, _ _u32 cookie)

void (*destroy)(struct dst_entry *)

void (*ifdown)(struct dst_entry *, struct net_device *dev, int how)

struct dst_entry * (*negative_advice)(struct dst_entry *)

void (*link_failure)(struct sk_buff *)

void (*update_pmtu)(struct dst_entry *dst, u32 mtu)

int (*get_mss)(struct dst_entry *dst, u32 mtu)

参见第三十三章“DST与调用协议间的接口”一节。

int entry_size

L3层路由缓存结构（例如，针对IPv4的结构为rtable）的大小。

kmem_cache_t *kmem_cachep

分配路由缓存元素的内存池。

flowi 结构

使用flowi数据结构，可以根据如入口设备和出口设备、L3和L4层协议报头中的参数等字段的组合对流量进行分类。它通常被用作路由查找的搜索关键字，IPsec策略的流量选择器及其他高级用途。下面简要描述该结构的字段：

int oif

int iif

出口设备标识符和入口设备标识符。

union {...} nl_u

该联合的各个字段可用于指定L3层参数取值的结构。目前支持的协议为IPv4、IPv6和DECnet。

__u8 proto

L4层协议。

__u8 flags

该变量只定义了一个标识，FLOWI_FLAG_MULTIPATHOLDROUTE，它最初用于多路径代码，但已不再使用了。

union {...} uli_u

该联合的各个字段是可用于指定L4层参数取值的主要结构。目前支持的协议为TCP、UDP、ICMP、DECnet和IPsec协议套件（suite）。

由于flowi数据结构不是扁平式的（flat），而是包含了许多联合和结构，所以内核提供了一组宏用于访问某些字段。

rt_cache_stat结构

rt_cache_stat结构存储了在“统计数据”一节中介绍的统计信息的计数器。下面描述这些计数器：

in_hit

out_hit

分别表示已经查找路由缓存成功而被路由的接收封包和本地产生封包的数目（参见ip_route_input和ip_route_output_key）。

in_slow_tot

in_slow_mc

in_slow_tot是由于缓存查找失败而需要查找路由表的封包数目（参见ip_route_input_slow），只对查找路由表成功的封包计数。该计数器被称为慢的（slow）是因为查找路由表与查找路由缓存相比非常慢。该计数器也对广播封包计数，但不对多播流量计数。多播流量用in_slow_mc变量来计数。

out_slow_tot

out_slow_mc

out_slow_tot和out_slow_mc与in_slow_tot和in_slow_mc扮演相同的角色，但它们用于出口流量的计数。

in_no_route

由于路由表不知道如何到达目的IP地址（只可能发生在默认网关没有配置或不可用的情况下）而不能被转发的入口封包的数目。参见ip_route_input_slow。没有计数器用于跟踪由于查找路由失败而不能被送出的本地产生封包的数目。

in_brd

被正确接收（合理性检查没有失败）的广播封包的数目。没有计数器用于送出的广播封包数目。

in_martian_dst

in_martian_src

这两个计数器分别表示由于目的IP地址或源IP地址没有通过合理性检查而被丢弃的封包数目。合理性检查的例子包括源IP地址不能为多播或广播，目的地址不能属于所谓的零网段，即地址不能为0.*n*.*n*.*n*。

gc_total

gc_ignored

gc_goal_miss

gc_dst_overflow

这四个字段被第三十三章“rt_garbage_collect函数”一节中描述的rt_garbage_collect更新。

gc_total跟踪rt_garbage_collect函数被启用的次数。

gc_ignored是rt_garbage_collect由于被调用不久而立即返回的次数。

gc_goal_miss是缓存被rt_garbage_collect扫描但没能满足函数开始时所设定的目标的次数。

gc_dst_overflow是gc_garbage_collect函数由于没有将缓存表项数目减少到ip_rt_max_size门限值以下而失败的次数。

in_hlist_search

out_hlist_search

这两个字段分别由缓存查找函数ip_route_input和__ip_route_output_key更新。它们表示已经测试但没有找到匹配的缓存元素数目（不是缓存查找失败次数）。

ip_mp_alg_ops结构

ip_mp_alg_ops表示路由缓存与多路径缓存功能之间的接口，该结构由以下函数指针组成：

```
void (*mp_alg_select_route) (const struct flowi *flp, struct rtable
*rth, struct rtable **rp)
void (*mp_alg_flush) (void)
void (*mp_alg_set_nhinfo) (_ _u32 network, _ _u32 netmask, unsigned
char prefixlen, const struct fib_nh *nh)
void (*mp_alg_remove) (struct rtable *rth)
```

这些函数在第三十三章“路由缓存与多路径之间的接口”一节中描述的独立于算法的包裹函数启用。

本部分涉及的函数和变量

表36-13汇总了在本书描述的路由子系统的章节中，介绍或引用的主要函数、变量和数据结构。更多的函数和变量可以参见第三十二章中“常用的辅助程序与宏”一节和“辅助函数”一节，以及在第三十五章中的两个“辅助程序”小节。

表36-13：路由子系统中的函数、变量和数据结构

函数	描述
for_ifa, endfor_ifa for_primary_ifa,endfor_ifa	这些宏用于遍历一个网络设备上配置的IPv4地址。参见第三十二章“主IP地址与辅助IP地址”一节
FIB_RES_*XXX*	这组宏用于访问fib_result结构的各个字段。参见第三十二章“常用的辅助程序与宏”一节
LOOPBACK ZERONET MULTICAST LOCAL_MCAST/BADCLASS	这些宏用于识别特殊的IP地址。参见第三十二章“常用的辅助函数与宏”一节
fib_hash_lock fib_info_lock fib_rules_lock rt_flush_lock fib_multipath_lock alg_table_lock	用于保护各种数据片段的锁。参见第三十二章“全局锁”一节
ip_rt_init ip_fib_init devinet_init fib_rules_init fib_hash_init dst_init	初始化函数。参见第三十二章“路由子系统初始化”一节

表 36-13：路由子系统中的函数、变量和数据结构（续）

函数	描述
dst_alloc	为路由缓存分配一个表项。参见第三十三章“缓存表项的内存分配与引用计数”一节
rt_periodic_timer rt_secret_timer	定时器。参见第三十三章“垃圾回收”和“刷新路由缓存”小节
fib_netdev_event fib_inetaddr_event	netdev_chain 和 inetaddr_chain 通知链的处理程序参见第三十二章“外部事件”一节
fib_add_ifaddr fib_del_ifaddr	在本地网络设备上通过配置添加或删除一个IP地址时，分别使用这两个函数来更新路由表。参见第三十二章“添加一个 IP 地址”和“删除一个 IP 地址”小节
fib_magic	内核用于在特定条件下插入路由表项。参见本章“内核插入路由项：fib_magic 函数”一节
fib_rules_detach fib_rules_attach	当网络设备注册和注销时分别用于开启和关闭路由策略。参见第三十二章“对策略数据库的影响”一节
rtmsg_fib	当路由已经被添加或删除时向一个特定的Netlink多播组发送通知。参见第三十二章“Netlink 通知”一节
ip_route_input _ _ip_route_output_key ip_route_output_flow ip_route_output_key ip_route_connect ip_route_newports	前两个函数是路由缓存查找函数，其他函数都是针对它们的包裹函数。参见第三十二章“缓存查找”一节
ip_route_input_slow ip_route_output_slow	路由表查找函数。参见第三十五章
ip_route_input_mc	用于多播目的地址的查找函数
ip_mkroute_input ip_mkroute_input_def ip_mkroute_output ip_mkroute_output_def fib_select_default fib_select_multipath	ip_route_input_slow和ip_route_output_slow使用到的各种支持函数。参见第三十五章
fib_lookup fn_hash_lookup fib_semantic_match	在路由表查找的不同阶段调用的函数。参见第三十五章“查找函数的高层见解”一节
fn_hash_insert	向路由表添加一个新路由表项。参见第三十四章“添加路由”一节

表 36-13：路由子系统中的函数、变量和数据结构（续）

函数	描述
fn_hash_delete	从路由表删除一个路由表项。参见第三十四章“删除路由”一节
rt_intern_hash	向路由缓存添加一条表项。参见第三十三章“添加元素到缓存中”一节
multipath_alg_register multipath_alg_unregister	注册和注销一个多路径缓存算法。参见第三十三章“注册一个缓存算法”一节
multipath_select_route multipath_flush multipath_set_nhinfo multipath_remove	用于管理与多路径路由相关的缓存表项的各种函数。参见第三十三章“路由缓存与多路径之间的接口”一节，更多的程序在同一章“辅助程序”一节中列出
rt_free dst_free	分别释放一个 rtable 和一个 dst_entry 结构
rt_garbage_collect rt_may_expire	用于路由缓存的垃圾收集函数。参见第三十三章“rt_garbage_collect 函数”一节
dst_input dst_output	分别完成封包的接收和发送。参见第三十三章“缓存查找”一节，也可以参见第三十三章“为接收和发送设置函数”一节
rt_garbage_collect dst_destroy dst_ifdown dst_negative_advice dst_link_failure dst_set_expires	这些程序用于初始化与 IPv4 协议相关的 dst_ops 实例。参见第三十三章“DST 与调用协议间的接口”一节
dst_dev_event	DST子系统用于处理来自netdev_chain通知链事件的函数。参见第三十二章“外部事件”一节
RT_CACHE_STAT_INC	更新各个 CPU 的统计数据。参见本章“统计数据”一节
变量	
ip_fib_local_table ip_fib_main_table	路由表。参见第三十四章“两个默认路由表：ip_fib_main_table 与 ip_fib_local_table”一节
rt_hash_table	路由缓存。参见第三十三章
rt_hash_mask	路由缓存大小（即 hash 表的 bucket 数目）
dst_garbage_list	由于仍然被引用而不能被删除的dst_entry实例链表。参见第三十三章
fib_tables	fib_table实例链表。参见第三十四章中的图 34-1

表 36-13：路由子系统中的函数、变量和数据结构（续）

函数	描述
fib_rules	路由策略链表。参见第三十五章“策略路由时的fib_lookup”一节
fib_info_cnt	所有fib_info实例数目。参见第三十四章“全局hash表容量的动态调整”一节。
fib_info_hash fib_info_laddrhash	用于搜索fib_info实例的hash表。参见第三十四章“fib_info结构的组织”一节
fib_info_devhash	用于搜索fib_nh实例的hash表。参见第三十四章“下一跳路由器结构的组织”一节
fib_props	查找函数fib_semantic_match使用该向量的元素将路由类型映射为返回值。参见第三十五“fib_semantic_match函数的返回值”一节
数据结构	
fib_table structure fn_zone structure fib_node structure fib_alias structure fib_info structure fib_nh structure fib_rule structure rtable structure dst_entry structure dst_ops structure flowi structure rt_cache_stat structure ip_mp_alg_ops structure	路由代码使用的关键数据结构。详细描述可以参见本章“本书此部分中文件和目录的特征”一节

本部分涉及的文件和目录

图 36-6 列出了在第七部分章节中涉及的文件和目录。

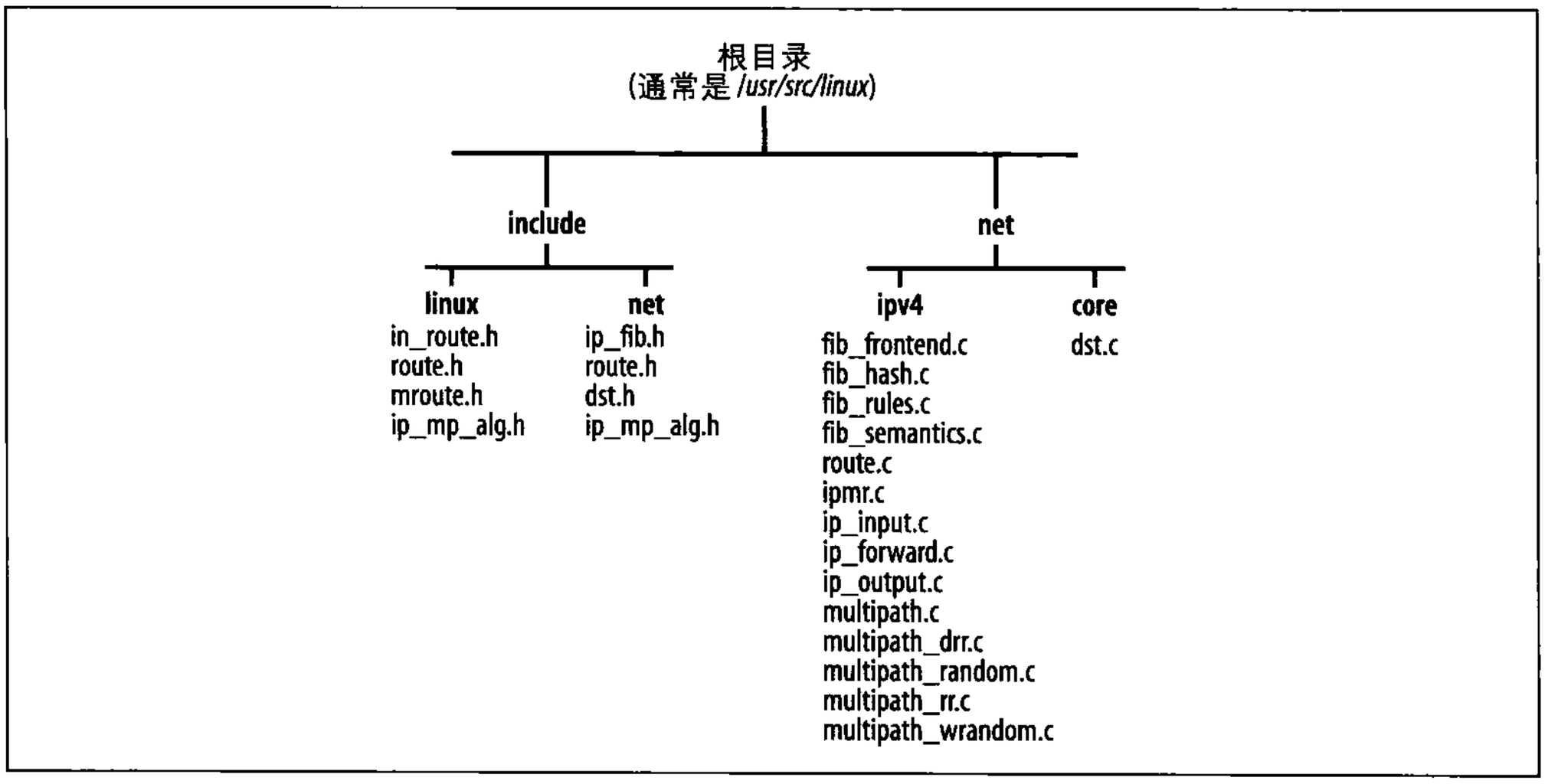


图 36-6：本部分涉及的文件和目录

作者简介

Christian Benvenuti 在意大利的Bologna大学取得了他的计算机科学硕士学位。他与位于Trieste的国际理论物理中心（International Center for Theoretical Physics，ICTP）合作多年，开发了基于Linux内核的特定软件，他是一个远程协作项目的科学顾问，并且是多个网络培训课程的讲师。这些培训主要在欧洲、非洲与南美洲举行，均是基于Linux系统，面向发展中国家的科学家，在这些国家，ICTP已经推广Linux多年。他有时也与由ICTP成员建立的一个非营利组织（Collaborium.org）合作，继续在发展中国家推广Linux。

在过去的几年里，他作为软件工程师为硅谷的Cisco Systems工作，致力于第二层交换、高可用性和网络安全。

[General Information]
书名=深入理解LINUX网络技术内幕
作者=夏安，闫江毓，黄景昌译
页数=997
SS号=12307639
出版日期=2009.06