Broadview®
www.broadview.com.cn

嵌入式Linux网络体系结构设计与TCP/IP协议栈

单立平 编著

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

嵌入式Linux

网络体系结构设计与TCP/IP协议栈

单立平 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书涵盖了 Linux 嵌入式系统开发中网络体系结构实现的主要内容。

全书共分 12 章，第 1 章概述 Linux 内核组件与内核技术特点，以及网络体系结构实现应用到的内核开发的基础知识。第 2～5 章在介绍了实现网络体系结构、协议栈、设备驱动程序的两个最重要的数据结构 sk_buff 和 net_device 的基础上，展示了 Linux 内核中为网络设备驱动程序设计和开发而建立的系统构架，最后以两个实例来具体说明如何着手开发网络设备驱动程序，数据在硬件设备上的接收和发送过程。第 6 章讨论了网络协议栈中数据链路层收发数据的设计和实现，以及硬件层与协议层之间的接口。第 7 章讲解了网络层 IP 协议的实现。第 8～9 章介绍传输层数据收发过程，重点介绍基于套接字的 TCP/UDP 传输实现。第 10 章讨论了 Linux 内核套接字层的实现，以及套接字层与应用层、传输层之间的接口。第 11 章介绍网络应用软件的开发技术，以及内核对网络应用的支持。第 12 章讲解在嵌入式系统开发中如何将硬件驱动程序、内核代码、应用程序集成在一起下载至芯片中，形成嵌入式可运行的系统，作为全书的总结。

本书可以作为高等院校计算机、通信专业学生学习操作系统的参考书，也可以作为从事嵌入式、计算机行业的工程技术人员的参考书。

图书在版编目（CIP）数据

嵌入式 Linux 网络体系结构设计与 TCP/IP 协议栈/单立平编著.—北京：电子工业出版社，2011.5
ISBN 978-7-121-12976-6

Ⅰ.①嵌… Ⅱ.①单… Ⅲ.①Linux 操作系统－程序设计②计算机网络－通信协议 Ⅳ.①TP316.89②TN915.04

中国版本图书馆 CIP 数据核字（2011）第 026595 号

策划编辑：张春雨
责任编辑：许　艳
特约编辑：赵树刚
印　　刷：三河市鑫金马印装有限公司
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×1092　1/16　印张：30　字数：807 千字
印　　次：2011 年 5 月第 1 次印刷
印　　数：4000 册　定价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前　言

无论现在或将来，网络都是一个热门的主题。目前几乎所有的电子产品都具备不同程度的网络功能。网络功能的强弱和灵活度与其使用的操作系统对网络的支持程度有直接的关系。Linux 操作系统从开发之初就是在 Internet 环境下实现的，网络子系统是 Linux 系统中最重要、最具特色的子系统之一，Linux 内核中网络子系统在体系结构设计上的合理与灵活，使其可以任意地在现有 Linux 内核协议栈的基础上实现新的网络协议、网络功能特色、对新网络适配器硬件的支持。

Linux 是开放源代码的系统，随着它的技术越来越完善，Linux 的应用也越来越普及，现在更多的产品都选择把 Linux 作为其嵌入的操作系统。在全世界有无以计数的计算机研发人员和爱好者在使用、测试 Linux 操作系统，为 Linux 系统开发新的应用，使 Linux 的技术日趋完美。

使用 Linux 作为研发平台，与使用别的操作系统不一样，只能通过其提供的应用编程接口（API）函数来完成，无法清楚地了解其内部的实现原理，也就无法更好地在研究过程中对性能、效率等实施控制，有时为了满足应用的需求，需要费很大的周折。特别是做嵌入式开发时，更需要依据手上的资源和成本要求对开发过程实施控制。

Linux 是开放源码的系统，可以通过学习和研究掌握其内部的实现，这无论对科研、学习还是系统开发都能带来巨大的好处，这样才能根据需要量体裁衣，定制自己所需的操作系统，去掉多余的功能，只保留最有效、最适用的部分。Linux 内核的网络功能更是如此，网络子系统具有大量的可选功能，如防火墙功能、路由功能等，不是每个设备都需要配备所有的这些功能，特别是在嵌入式系统，配置 Linux 内核组件就显得尤其重要。

Linux 网络体系结构比 Linux 内核中其他组件理解起来更困难，原因在于网络任务的实现被划分为好几个阶段，在不同的时间由不同的代码和进程来实现。如何将这些片段连在一起，各阶段之间的接口是什么，是研究 Linux 网络体系结构的一个难点。

笔者多年从事嵌入式 Linux 网络系统产品开发，在这个过程中了解到嵌入式系统的开发涉及硬件驱动、操作系统内核和系统应用 3 个层次。最终的嵌入式产品需要将以上 3 个部分集成，形成一个完整的可执行文件，下载至嵌入式芯片中。本书的目的就是以 Linux 内核的网络子系统为纵向线索，以 Linux 内核 TCP/IP 协议在网络子系统中的实现为实例，把与嵌入式网络应用开发相关的技术知识组织在一起，来讲解嵌入式 Linux 系统的应用开发技术、内核支持和硬件驱动程序开发的完整过程，以形成相关知识领域的完整体系结构，这样读者在研发过程中应用起来更得心应手。无论在做哪个部分的研发和学习：应用、内核、驱动程序，都能清楚地知道自己在做什么，上下之间如何联系。

Linux 内核网络子系统的功能特色多，代码分散，所以本书的一个重要目的就是清楚地解释 Linux 内核中网络子系统的主要功能特色、设计原理和实现流程，并告诉读者如何跟

踪分析其内部 C 源代码，如何将不同的代码实现片段串连在一起，并展示其函数功能和数据结构的相互关系，从而对具体的研究过程起到指导和帮助作用。Linux 内核 TCP/IP 协议栈的实现技术是学习和研究操作系统中网络子系统的一个很好的样本和实例。

全书共分 12 章，按照 TCP/IP 协议栈分层结构，从如何驱动硬件、网络数据在 TCP/IP 协议栈中如何发送/接收，到网络应用程序开发技术，最后如何形成嵌入式集成系统的思路来安排各章节的内容。本书由电子科技大学教授、博士生导师罗蕾担任主审，罗蕾教授对本书的编写予以积极支持和认真指导，她仔细地审阅了书稿，提出许多宝贵意见，在此谨向罗蕾教授表达衷心的感谢。在本书编写过程中，还得到了电子工业出版社张春雨老师等的支持和帮助，谨向他们表示诚挚的谢意。

本书主要由单立平编著，其他参与编写的人员有姜云舟、单根全、单宏伟、王雪梅等。

由于本人水平有限，对书中的错误和不足之处，恳请广大读者批评指正，笔者将不胜感激。

编 著 者

目　录

第 1 章　概　述

本章在介绍 Linux 内核的主要组件及功能的基础上，概要描述了作为 Linux 内核组件中的网络子系统在实现过程需要的共享系统资源、资源管理技术、内核活动类型和特点、并发访问控制、模块机制等内容。本章的主要目的是让读者了解网络子系统的实现应具备的预备知识。

TCP/IP 协议栈是使用最广泛的网络协议栈，Internet 就是建立在 TCP/IP 协议栈基础上的。本书将以 TCP/IP 协议栈在 Linux 内核中的设计和实现为实例来讲解 Linux 的网络体系结构。我们将深入到 TCP/IP 协议栈在 Linux 内核中的实现细节，包括各层协议工作的基本原理、Linux 内核中描述协议的数据结构、协议层之间的接口和实现源码分析；由此揭示 TCP/IP 协议栈在 Linux 中实现的优异性，它异常健壮、成熟和稳定。在本书中我们不仅会分析 Linux 内核 TCP/IP 协议栈与网络体系结构，同时也会带领读者深入到内核的源代码中，给出跟踪 Linux 内核源代码执行流程的方法。

Linux 内核网络体系结构的设计和实现非常高效，极易扩展和增加新的协议栈。除 TCP/IP 协议栈外，Linux 内核可以支持多个不同的协议栈。本书以 TCP/IP 协议栈的实现为基础，来讲解 Linux 内核网络子系统的体系结构。全书内容涵盖：网络数据包在 TCP/IP 协议栈各层协议的处理；网络设备驱动程序的设计和实现；Linux 网络应用程序设计和实现；应用如何与内核网络功能交互。

虽然网络子系统的功能在 Linux 内核中相对独立，但它与内核中其他子系统类似，需要使用系统资源，实现对资源的管理，如内存管理、数据对象分配和释放、系统时钟、中断资源等。网络子系统中的活动是内核中的活动，具有内核活动的共同特点，比如它有中断处理、tasklet、系统调用等。在这一章中，我们将对 Linux 内核的整体结构和网络子系统实现需要用到的系统服务、资源和活动做概要性的介绍，以便读者更好地理解网络子系统的工作过程。

本书在写作时分析的 Linux 源代码是基于 Linux-2.6.29.6 版本的，读者可以从 http://www.kernel.org 下载该版本的全部源代码。在本书中我们常会用到以下的缩写来表示协议栈的各层。

- L2 数据链路层（如以太网协议）。
- L3 网络层（如 IP 协议）。
- L4 传输层（如 UDP/TCP/ICMP 协议）。

1.1　Linux 内核组件

在这一节中，我们将描述 Linux 内核的基本体系结构、内部组件和各组件的主要功能，给读者提供对 Linux 内核中最主要部分概貌性的介绍。以下描述的每个部件都为网络操作

提供了相应服务，可以帮助理解 Linux 网络体系结构的实现。

图 1-1 给出了 Linux 内核的主要组件，Linux 内核可以划分为 5 个组件部分，每个部分的功能定义都非常明确，各组件又为内核的其他组件提供相应的服务。这种划分可以从内核的源码树形结构中看到，在内核源码根目录下每个组件都有自己的目录和子树。

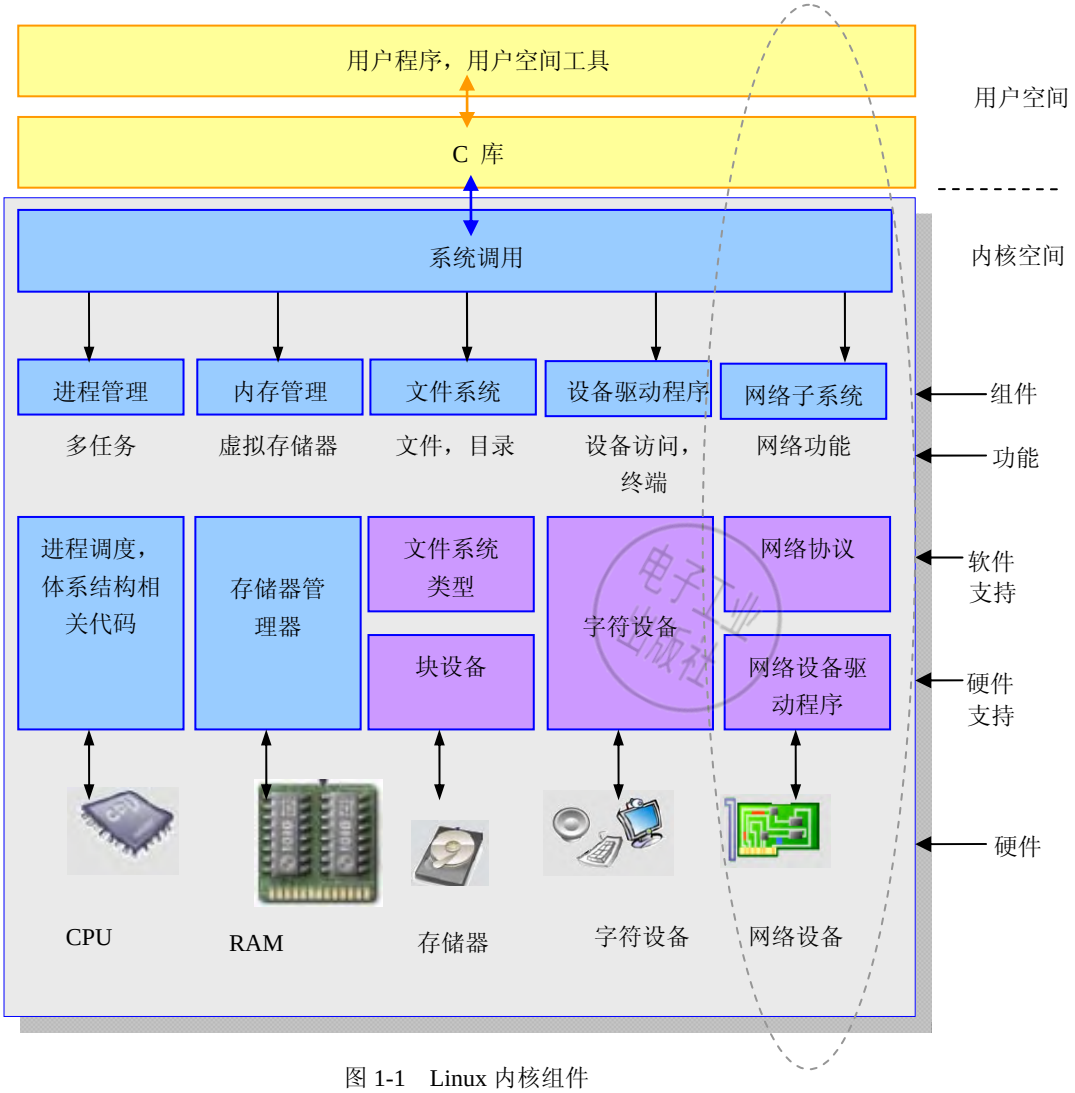


图 1-1　Linux 内核组件

各组件的主要功能介绍如下。

1．进程管理

负责创建、结束进程，管理内核的活动，如软件中断、tasklet 等，管理进程间通信，如消息（message）、管道（pipe）等，实现进程调度（schedule）。进程调度是进程管理的重要任务，它处理所有活动的、等待被执行的和被阻塞（blocking）的进程调度，使所有应用和进程合理地共享处理器的运行时间。

2．内存管理

内存是系统最主要的资源之一，计算机的性能在很大程度上与其所配备的内存有关。Linux 内核内存管理的主要功能就是给进程分配地址空间，该地址空间只允许本进程自己访问。

3．文件系统

在 Linux 操作系统中，文件系统是整个系统的中枢。Linux 与其他操作系统不同，几乎所有的操作都基于文件系统接口的处理，如设备驱动程序可以按文件方式访问设备，通过/proc 文件系统可以访问 Linux 内核的数据和参数，这两个功能在调试时非常有效。

4．设备驱动程序

在所有的操作系统中，设备驱动程序都是硬件的抽象，通过它可以访问硬件。Linux 可以用模块（模块）的方式实现设备驱动程序，提供了在系统运行时动态加载和卸载设备驱动程序的途径。

5．网络子系统

在 Linux 中所有的网络操作是由操作系统管理的。这是因为网络操作不能分配给某个进程完成。在处理收到的网络数据包时，数据包的接收是异步事件。接收数据包任务必须在进程处理这些数据包前先收集齐所有的网络包、标识数据，然后向上层传送，这就是为什么由内核的网络子系统负责处理数据包，而不是由某个进程和网络接口来处理。

在内核中还定义了大量的接口，目的是为了更方便地扩展内核功能，如虚拟文件系统接口（Virtual Filesystem Interface），可用于增加新的文件系统，现在 Linux 中能支持十多种不同的文件系统；可见 Linux 的开发人员定义这种接口所带来的优势。在 Linux 的网络体系结构中也定义了很多接口，用以支持动态增加网络协议和网络设备驱动程序。

图 1-1 中描述的部分组件都提供了接口以便动态向内核注册和注销新的功能，这种动态注册的功能可以很容易地以模块（module）的方式实现。图 1-1 中虚线中包括的部分就是 Linux 内核的网络子系统，也是本书要讲解的内容。

1.2　Linux 内核中的活动

Linux 是支持多任务的操作系统，即多个应用程序可以在系统中同时运行。在多处理器环境下，执行多个应用程序的进程可以被并行处理，但进程并不是内核中能执行的唯一活动。内核中还存在几种其他形式的活动，在系统的运行过程中起着重要作用。

1.2.1　进程和系统调用

什么是进程？进程（process）是一个正在执行的程序实例，各进程拥有自己独立的地址空间。进程通常在执行某个应用程序时启动，应用执行完成后结束。创建、控制和结束进程是操作系统内核的一项重要任务。在用户地址空间执行的进程是互斥的，它们只能访问系统分配给它们的存储空间。用户地址空间的进程也不能直接访问内核功能。

当用户进程需要访问设备或使用操作系统内核的功能时，必须通过系统调用（system call）来完成。系统调用将处理器切换到保护模式，随后访问内核的地址空间，在保护模式下，所有的设备和内存资源通过内核实现 API 访问。

除了常规的进程和系统调用外，在内核中还包含了几种其他活动，这几种活动对网络子系统而言尤其重要，因为网络功能就是在内核中处理的，在本节后面的部分，我们将较详细介绍以下几种活动：

- 内核线程（kernel thread）。
- 中断（硬件中断）。
- 软件中断。
- tasklet。
- bottom half。

在设计内核活动时，最重要的一点是使这些活动能并行执行；另一方面，在多处理器环境下也要考虑同一种活动的同一实例是否可以在不同处理器上并行运行，或同一种活动的两个不同的实例是否可以同时在不同处理器上运行。表 1-1 中列出了以上所述活动的相同实例和不同实例是否可以在不同 CPU 上同时执行的情况。

表 1-1　内核中同一活动在不同 CPU 上同时执行

活　　动	相 同 实 例	不 同 实 例
硬件中断（HW IRQ）	否	是
软件中断（Soft IRQ）	是	是
tasklet	否	是
bottom half	否	否

在这些活动中，另一个需要明确的是，哪种活动的优先级更高，即一种活动是否可以被其他的活动中断。表 1-2 列出了以上活动之间是否可被其他活动中断的关系。这些信息特别重要，因为要避免某个活动因不可预期的打断而造成系统数据的破坏。

表 1-2　活动间可被其他活动中断的关系

活　　动	HW_IRQ	Soft-IRQ	tasklet	bottom half
硬件中断（HW IRQ）	是	否	否	否
软件中断（Soft IRQ）	是	否	否	否
tasklet	是	否	否	否
bottom half	是	否	否	否
系统调用	是	是	是	是
进程	是	是	是	是

1.2.2　硬件中断

外部设备用硬件中断来通知操作系统有重要的事件发生，中断发生后，CPU 会暂时停止当前程序的执行，转去执行中断处理程序，中断处理程序结束后再恢复原来被停止程序的执行。

硬件中断是一种系统资源，当我们为设备编写中断处理程序来处理外部事件时，要向系统申请中断资源（即硬件中断信号线，通常称为中断号），并将中断处理程序与中断源相

关联。我们可以在运行时用内核提供的一对函数来申请和释放中断资源。

```
   int request_irq(unsigned int irq,irq_handler_t handler,unsigned long flags, const char *devname,
void *dev_id)
```

函数 request-irq 申请中断号为 irq 的资源，将申请到的中断号 irq 与设备名为 devname，设备索引号为 dev_id 设备的中断处理程序 handler 相连。该设备的中断类型为 flags。

```
   void free_irq(unsigned int irq, void *dev_id)
```

函数 free-irq 释放中断资源。

在这里需要对中断类型标志 flags 做进一步说明，在网络子系统中，我们主要区分两类中断类型（虽然在内核中还有其他的中断类型）。

1．快速中断

快速中断（fast interrup）的中断处理程序运行时间非常短，所以当前被打断的活动暂停时间也很短。快速中断的特点是，屏蔽当前运行中断处理程序 CPU 的其他中断，这样中断处理程序的执行就不会被别的中断打断。要将中断申请注册为快速中断，需在申请中断资源时将中断类型标志 flags 设为 SA_INTERRUPT。

2．慢速中断

慢速中断（slow interrupt）的中断处理程序在执行期间可以被别的中断打断。慢速中断处理程序执行的时间较长（相对快速中断而言），所以占用 CPU 的时间也长。

中断处理程序的执行通常可以暂停所有别的活动。不同的中断在几个 CPU 上可以同时运行，但某个中断的处理程序一次只能在一个 CPU 上执行。

如果你想查看当前 CPU 是否在中断活动中，则可以调用内核的 API。

```
   in_irq()                                    include/asm/hardirq.h
```

3．top half 和 bottom half

硬件中断可以打断当前 CPU 上所有其他的活动，并会屏蔽 CPU 其他硬件中断。硬件中断是系统中有限的资源之一，硬件中断处理程序应尽可能快地执行完。在执行中断处理程序期间，如果设备有其他的外部事件发生，由于当前中断资源和 CPU 已被占用，其他的外部事件就不能得到 CPU 的响应，所以中断处理程序的执行要尽可能的快。但不是所有事件处理都可以在很短时间内完成，例如网络数据包的接收处理，从网络适配器收到的数据包到将数据包传送到用户接收进程，需要几千个 CPU 时钟才能完成。接收网络数据包这类事件虽然是由中断触发，但对网络数据包的处理却不能全部放在中断处理程序中来做。

为了节省系统资源，使中断处理程序的运行时间尽可能短，像上述这类费时的事件处理在 Linux 内核中将其分成两个部分来完成。

（1）top half

top half 只完成中断触发后最重要的任务处理，这里 top half 就对应中断处理程序。在第 5 章我们分析网络设备驱动程序的中断处理程序时会看到，网络适配器的中断处理程序只将收到的数据包复制到内核的缓冲区后就立即结束返回，释放占有的中断和 CPU。对数据包的协议分析和处理不在中断处理程序中进行。

（2）bottom half

bottom half 完成所有非紧急部分的处理。bottom half 代码由 top half 调度，放在以后某个安全的时间运行。比如上述的网络数据包复制到内核以后的分析处理，就在这部分代码中完成。

bottom half 与 top half 最大的区别是，在 bottom half 执行期间，打开所有硬件中断。它们之间的关系是：top half 将在设备缓冲区的数据复制到内核地址空间缓冲区，调度 bottom half 后退出，这个过程非常快。bottom half 执行余下的处理任务，这样在 bottom half 工作过程中，CPU 可以响应新的外部中断请求。Linux 中有两种不同的机制来实现 bottom half 过程：tasklet 和 workqueue。

1.2.3 tasklet

tasklet 是可以被调度执行的特殊函数，在系统某个特定的安全时间运行在软件中断的执行现场。tasklet 的特点是：

- 由函数 tasklet_schedule 调度执行，一个 tasklet 只运行一次。

```
static inline void tasklet_schedule(struct tasklet_struct *t)
```

- 一个 tasklet 一次只能在一个 CPU 上执行。
- 不同的 tasklet 可以同时在不同的 CPU 上运行。

实现 tasklet 的过程可分为以下几个步骤完成：

① 编写 tasklet 的处理函数，如 func。

② 用宏 declare tasklet (name,func,data)声明一个新的 tasklet，name 为该 tasklet 的名称，func 为该 tasklet 要执行的函数，data 是传送给 tasklet 处理函数的数据。

③ 用 tasklet_schedule 调度 tasklet 执行。

除此以外，函数 tasklet_disable 和 tasklet_enable 可以分别用于禁止和允许 tasklet 的运行。下面给出一个简单的例子来了解 tasklet 的实现。

```
void my_func(unsigned long);                    /*tasklet 的处理函数*/
char tasklet_data[] = "This is a new tasklet"; /*tasklet 处理程序需要的数据*/
DECLARE_TASKLET(my_tasklet, my_func, (unsigned long) &tasklet_data);  /* 声明 tasklet */
void my_func(unsigned long data)                /*实现 tasklet 函数*/
{
    /* 完成函数体*/
}
…
/*调度新定义的 tasklet */
tasklet_schedule(&my_tasklet);
```

1.2.4 workqueue

workqueue 与 tasklet 极其类似，是可以由内核代码在将来某个时间调用执行的特殊函数。workqueue 与 tasklet 的不同在于：

- tasklet 在软件中断执行现场运行，所有的 tasklet 代码必须是原子操作。workqueue 函数在内核进程现场执行，执行起来更灵活，workqueue 函数可以休眠。
- tasklet 总是在最初调度它的处理器上执行。默认情况下 workqueue 与 tasklet 一样。

- 内核可以将 workqueue 函数推迟到一定时间以后才调用执行。

1.2.5　软件中断

软件中断是在硬件中断执行完后由内核的调度器（scheduler）调度执行的活动。软件中断和硬件中断的主要区别在于：硬件中断可以随时立刻打断 CPU 现行活动（如中断允许）；软件中断是由内核调度器调度执行的活动。软件中断必须要等到调度器调用它才能执行，软件中断的调度由内核函数 do_softirq 完成。

```
asmlinkage void do_softirq(void)                                    //kernel/softirq.c
```

软件中断的处理程序在 do_softirq 后开始执行。软件中断的执行时间只有两处：

- 系统调度结束后（在 schedule 中）被调度执行。
- 硬件中断结束后（在 do_IRQ 中）被调度执行。

Linux 内核中最多可以定义 32 个软件中断，目前使用的有：

```
enum                                                                //include/linux/interrupt.h
{
   HI_SOFTIRQ=0,
   TIMER_SOFTIRQ,
   NET_TX_SOFTIRQ,
   NET_RX_SOFTIRQ,
   BLOCK_SOFTIRQ,
   TASKLET_SOFTIRQ,
   SCHED_SOFTIRQ,
   HRTIMER_SOFTIRQ,
   RCU_SOFTIRQ
   NR_SOFTIRQS
};
```

其中网络子系统使用的软件中断有如下几种。

- NET_RX_SOFTIRQ：处理网络接收到的数据包。
- NET_TX_SOFTIRQ：处理要发送的网络数据包。

另一个软件中断 TASKLET_FOFTIRQ 用于实现 tasklet 的概念。

软件中断与 tasklet 和 bottom half 有很大区别，软件中断最重要的特性是：

- 软件中断可以同时在多个处理器上运行，所以在编写软件中断处理程序时必须要考虑重入问题。如果在软件中断处理程序中要访问共享全局变量，必须采用锁定机制执行并发访问。
- 软件中断本身不能被同类的软件中断打断。
- 软件中断在执行时只能被硬件中断打断。

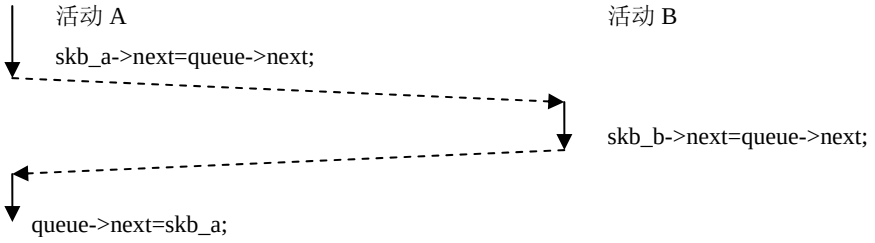
1.3　互斥机制

在 Linux 内核中几种不同的活动可能被相互打断；在多处理器环境下，不同的活动可以并行执行。在活动并行运行过程中，保持系统的稳定性和正确性是必须要解决的问题。

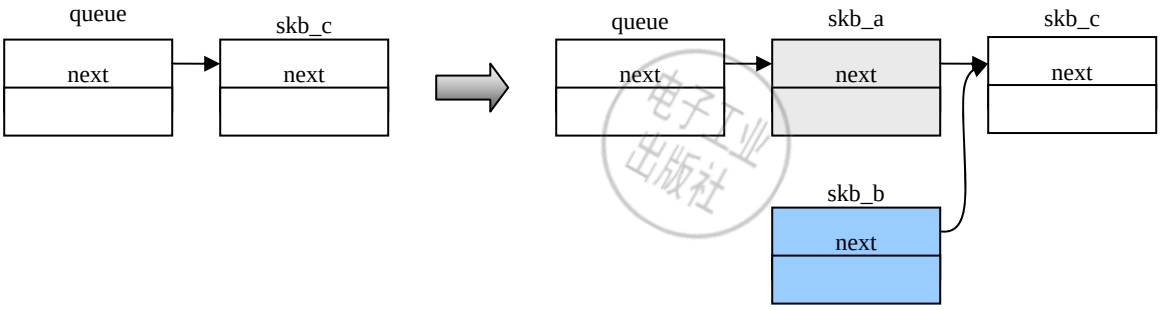
如果内核中的各活动相互独立，没有交叉访问，就不会引起任何问题。一旦几个活动

要访问同一数据结构，如果不采取相应的保护机制，即使在单 CPU 的系统中都可能会引起预想不到的结果。

下面我们用一个实例来说明这个问题，系统中有两个活动 A 和 B ，都要访问同一个链表，将自己的数据 skb_a 和 skb_b 加入到链表中。活动 A 在执行过程中被活动 B 打断，活动 B 执行一段时间后，活动 A 恢复执行。图 1-2 给出了这两个活动执行结束后的结果。将 skb_b 加入到链表中的操作结果不正确。



（a）活动 B 在活动 A 执行的关键时刻打断其执行



（b）无保护机制的操作造成操作结果错误

图 1-2　并发访问造成的问题

为了避免这类问题的发生，当几个活动需要同时操作同一个数据结构时（这段代码通常称为 critical section），这种操作必须是原子操作。所谓原子操作，即整个操作过程不能被打断，操作的几个步骤以不可分的方式执行。

在网络实现代码中，互斥机制使用得非常广泛。在 Linux 系统的发展过程中，不断创建、优化了几种机制来实现代码执行的互斥。以下几节主要总结了用于网络子系统中的互斥机制，每一种互斥机制适于某些特定的执行环境。

1.3.1　spin lock

这种锁定机制在执行时间一次只能由一个线程持有锁。如果一个线程已持有了锁，另一个执行线程想要获取锁时，就只能循环等待直到前一个线程将锁释放，也即在此期间处理器不做别的处理，一直在循环测试锁的状态。因此 spin lock 只适用于多处理器系统运行环境，而且通常用于预计锁可以在很短时间内就能获取的情况下。使用 spin lock 时，要求

持有锁的线程不能休眠，否则会造成别的线程因不能获取锁而发生系统死锁的情况。

spin lock 又称为忙等锁定机制，它适用于当一段 critical section 代码部分非常短，执行起来很快的合。这时如果我们采用的不是 spin lock 锁定机制，而是采用另一种方式，比如，当 CPU 测试到锁被别的活动持有，就用调度器将当前活动换出 CPU，调度别的进程来执行；锁有效后再把该活动调度进来执行，其中调度进程进出 CPU 所花费的时间可能比处理器忙而等待所用的时间更长，效率更低。因此，对很小的 critical section 程序段的保护采用 spin locks 更好。

在 Linux 内核中，spin lock 由 spinlock_t 类型的变量实现，它由一个整数的锁变量组成，为了使用 spin lock，首先要创建和初始化 spinlock_t 数据结构，使用 spin lock 的步骤为：

```
#include < linux/spinlock.h>
spinlock_t  my_spinlock=SPIN_LOCK_UNLOCKED;                    //初始化锁变量，或使用函数，初始化锁变量
spin_lock_init( &my_spinlock)                                  //将锁初始化为未锁定状态
```

接下来你可以用内核提供的一系列函数在需要的时候测试、获取或释放锁。获取锁的地方标志着 critical section 部分的开始，释放锁处标志着 critical section 部分的结束，这时别的活动又可以来获取锁，操作共享的数据结构。

1．获取锁

（1）spin_lock(spinlock_t *my_spinlock)

试图获取锁 my_spinlock，如锁已被别的进程持有，必须等待并不断测试锁的状态直到锁释放。锁被释放后可立刻获取。

（2）spin_lock_irqsave(spinlock_t *my_spinlock, unsigned long flags)

与 spin_lock 函数的功能类似，它自动屏蔽中断，将 CPU 的当前状态寄存器值保存到变量 flags 中。

（3）spin_lock_irq(spinlock_t *my_spinlock)

与 spin_lock_irqsave 类似，但不保存 CPU 状态寄存器的值，它假定中断已经屏蔽了。

（4）spin_lock_bh(spinlock_t *my_spinlock)

获取锁，同时阻止 bottom half 的运行。

2．释放锁

（1）spin_unlock(spinlock_t *my_spinlock)

释放一个获取的锁。

（2）spin_unlock_irqrestore(spinlock_t *my_spinlock)

释放一个 spinlock 的锁，并置中断允许。

（3）spin_unlock_irq(spinlock_t *my_spinlock)

释放锁，允许中断。

（4）spin_unlock_bh(spinlock_t *my_spinlock)

释放锁，并允许立即处理 bottom half。

以上的函数可以用于保护 critical section 代码的执行，避免多个活动同时操作共享数据结构带来不可预期的错误。比如，在图 1-2 中所示的操作可以用以下方式来保护活动 A 和活动 B 都要操作的链表。

```
spinlock_t my_spinlock = SPIN_LOCK_UNLOCKED;
// 活动A
spin_lock(&my_spinlock);
skb_a->next = queue->next;
queue->next = skb_a;
spin_unlock(&my_spinlock);
…
//活动B
spin_lock(&my_spinlock);
skb_b->next = queue->next;
queue->next = skb_b;
spin_unlock(&my_spinlock);
```

使用了锁定机制来保护链表后，活动 A 与活动 B 向链表中插入数据的操作就不会再出错了。

1.3.2 读－写 spin lock

在使用某个锁时，如果可以把操作明显地划分为只读、读写操作时，使用读-写 spin lock 会更有效。spin lock 与读－写 spin lock 的区别在于：后者可以允许多个读操作同时持有锁；一次只有一个写操作可以获取锁；当写操作持有锁时，读操作不能获取锁；通常赋予读操作的优先级比写操作的优先级高，当读操作的数量大大超过写操作数量时，这种锁定机制的执行效率更高。

例如，在网络子系统中，所有表示网络设备的数据结构 net_device 的实例都存放在以 dev_base 为头指针的链表中。在系统初始化时，当我们探测到一个有效的网络适配器时，就将该网络设备的 net_device 实例插入到链表中，这时为写操作。在系统运行过程中，我们很少改变 net_device 结构变量的值，大部分操作是在 dev_base 链表中查找设备，获取设备属性值，这属于读操作。

对这种数据结构的访问，一定要用锁定机制来保护，但对 dev_base 链表的读操作在没有写操作的情况下，不需要串行等待执行，就最适合读－写 spin lock 锁定机制。它允许多个读操作的活动同时运行其 critical section 部分的代码，但一旦锁被写操作获取后，所有的活动只能等到写操作释放锁后才能访问共享数据。

当我们要向 dev_base 链表加入一个新的网络设备的 net_device 数据结构实例时，我们需要用读－写 spin lock 的写锁来保护 dev_base 链表。以下列出的函数可以用于获取和释放读－写 spin lock。需要注意的是，我们要按照操作的类型（读/写）来区分锁的状态。同样，读－写锁也有处理中断和 bottom half 的函数，与上节所说的 spin lock 一样，在这里就不再重复叙述了。

```
read_lock …()
read_unlock…()
write_lock….()
write_unlock…()
```

1.3.3 读－复制－更新（Read-Copy-Update，RCU）

RCU 是 Linux 中提供的最新互斥机制，这种锁定机制在以下特定的条件下执行效率非常高。

- 相对于只读锁的要求，要求读－写锁的次数非常少。
- 持有锁的代码是以原子方式执行，绝不会休眠。
- 被锁保护的数据结构是通过指针访问的。

第一个条件涉及的是执行效率，其他两个条件是 RCU 工作原理的基础。如果第一个条件不满足，在需要使用锁定机制时，更好的方式就是使用读－写 spin lock。当数据需要修改时，写线程获取一个数据的拷贝，修改拷贝，随后将相关的指针改到新版本的数据结构上，当内核确定不再有对旧版本数据的引用时，旧数据就可以释放了。

1.4　内核模块（module）

Linux 内核是一个庞大、完整的操作系统，它包括了操作系统功能所有需要的组件。这样的好处是功能强大、运行效率高。但其缺点就是不够灵活，不容易扩展。当我们要为内核扩展一个新的功能时，特别是某些组件需要不断更新（如设备驱动程序，随着新设备的不断推出，要不断增加新驱动程序）时，每次小的改动，都需要编译整个操作系统，这是一个非常耗时的过程。

Linux 解决这个问题的方式是内核模块（module）。在需要的时候这些内核模块可以在运行时动态地加入到系统中。当不再需要的时候，可以将其从系统中移走。

图 1-1 给出的内核组件示例中，可以使用模块方式来扩展功能的组件有：设备驱动程序、文件系统、网络协议和网络设备驱动程序。模块的使用实际上并不仅仅局限于这些组件，它可以作为一个独立功能加入到内核中。向内核加入新功能时，内核中也需要相应的接口去通知内核的其他部件关于新功能的信息。Linux 网络体系结构中的接口和在其中扩展新功能的实现方式是本书要讨论的一个主要问题。

在本节的以下部分，我们会较详细地讨论 Linux 内核模块的结构和管理，因为模块是增强 Linux 网络体系结构最好，也是最灵活的方式。在网络子系统中大量使用了模块技术来扩展网络功能。

1.4.1　管理内核模块

内核模块由目标代码组成，它在运行时装载到内核地址空间并运行。在系统启动时，内核事先并不知道会有什么功能的模块会装载到系统中，所以模块必须自己通知内核，让相应的组件知道模块加载与否。当模块移走时，它也需要移走所有在内核地址空间对它的引用，释放占用的系统资源。这里有两个方法是用来完成以上任务的。

- init_模块：向内核注册由模块提供的所有功能。
- cleanup_模块：撤销任何由 init_模块所做的功能。

这两个方法每个模块都需要实现。在进一步深入理解模块的工作原理之前，我们首先给出一些常规的命令，以便了解如何从内核外来管理模块，用户可以用以下工具来手动插入和卸载模块。

1．插入模块

insmod 模块 name.o [arguments]：该命令用于装载一个内核模块到内核地址空间。如果成功，模块的目标代码就被链接到内核中了，这样模块就可以访问内核的符号（函数和数据结构）。发送命令 insmod 后会引起以下的系统调用执行。

- sys_create_模块：给模块在内核地址空间分配其驻留所需的内存。
- sys_get_kernel_syms：返回内核的符号表，解决模块中尚未连接的对内核符号的引用。
- sys_init_模块：复制模块的目标代码到内核地址空间，并调用模块的初始化函数（init_模块）执行模块的初始化功能。

2．给模块传参数

当我们装载模块时，也可以给模块传参数（例如设备的名称 name，设备的中断信号 irq 和 I/O 端口地址 io_addr）。在实现模块时，这些参数需要在模块中用宏 MODULE_PARM(arg,type,default)来说明装载模块时可以给模块传哪些参数。当用 insmod 命令装载模块时，这些参数可以直接传给要调度的模块：

```
[root@localhost]#  insmod mylan_cs.o eth=1 network_name="myWavlan"
```

在上例中，我们插入了一个名为 mylan_cs.o 的模块，并给它传送了两个参数，一个是设备 ID，一个是设备名。

3．移走模块

rmmod 模块 name：从内核地址空间卸载指定的模块。为此该命令会引起系统调用函数 sys_delete_模块的执行，而 sys_delete_模块系统会调用模块的清除函数 cleanup_模块。这样模块就从内核地址空间卸载了。

4．其他用户空间命令

- lsmod：列表当前所有装载了的模块以及它们相互的依赖关系和引用计数。
- modinfo：给出关于模块的信息（它的功能、参数和所有者等）。这些信息并不是自动产生的，它需要使用宏 MODULE_DESCRIPTION，MODULE_AUTHOR 在模块的源代码中定义。

1.4.2 自动装载模块

除了使用以上命令行工具外，内核模块可以在需要的时候自动装载到内核中。使用我们在前面介绍的工具装载模块和卸载模块需要用户的干预，而且出于安全原因只有根用户能使用 insmod 和 rmmod 来加载和卸载模块。虽然这样保证了安全，但却带来了不便。例如，当一个用户程序在运行过程中需要使用一个功能，但该功能的模块还没有加载到内核中，这就需要内核有能力自动调度模块到内核地址空间。

通常，在程序运行过程中如果需要的资源或某个设备驱动程序没有注册，内核会报错。你可以事先用内核函数 request_模块申请需要的组件模块。为了使用这个函数，在配置内核时需要激活选项 Kernel Module Loader。request_模块会调用 modprobe 命令自动装载需要的模块（并调度模块依赖的其他模块）。要自动调度哪些模块需要在配置文件/etc/模块 s.conf

中设置。

下面给出了/etc/模块 s.conf 配置文件的示例。在该文件中它指明当前网络设备是由模块 mylan_cs 代表的，为了装载该模块，需要给它传送特定的参数。如果 modprobe 不能找到模块，printk 会给出错误信息。

```
/etc/模块s.conf.
#Aliases – specify your hardware
alias eth0 mylan_cs
options mylan_cs eht=1 network_name="MyNet" station_name="neo"

alias char_major-4 serial
alias char-major-5 serial
alias char-major-6 lp
alias char-major-9 st

alias tty-ldisc-1  slip
alias tty-ldisc-3  ppp
```

虽然这种方式可以自动装载模块，但它也只能调度系统管理员在配置文件中指定的模块。

1.4.3　模块功能的注册和取消

应用程序的作用通常是运行后完成一定的功能，模块的主要任务是为当前内核中的其他组件提供服务。某个时候内核以模块的方式增加了新功能，运行一段时间后，当不再需要这种功能时，可能将其移去。在系统启动阶段我们无从知道将会有什么功能以模块的形式加入到系统中，所以我们需要为模块提供接口来注册。内核中各组件都有相应的模块注册的接口（如注册和取消网络驱动程序、文件系统、协议等）。

这些接口很容易从其函数名识别出来，一般是以 register_...和 unregister_...开始的函数。表 1-3 中给出了一些接口的函数示例。

表 1-3　内核组件模块注册和取消函数

功能模块	动态注册模块的接口
字符设备	(un)register_chrdev
块设备	(un)register_blkdev
二进制格式	(un)register_binfmt
文件系统	(un)register_filesystem
串行接口	(un)register_serial
网络适配器	(un)register_netdev
网络层协议（Layer-3）	dev_add_pack, dev_remove_pack
传输层协议（Layer_4，TCP/UDP）	inet_add_protocol inet_del_protocol
控制终端驱动程序	tty_((un)register_driver
符号表	(un)register_symtab
模块	init_模块, cleanup_模块

模块的注册和初始化由模块自己的 init_mudule 方法来完成。如前所述，该函数是在模块成功地集成到内核中后直接被调用。init_模块需要完成所有的模块初始化任务，如申请

内存空间，创建在 /proc 文件系统中的入口，初始化数据结构，注册函数等。

init_模块函数执行成功后，模块的功能就被内核识别了，而且它需要的所有初始化过程都应该运行结束；如果在初始化过程的某个环节出了错，所有在此之前执行的动作都应回退。因为，当 init_模块返回的是错误代码时，模块的目标代码会从内核地址空间卸载，所以任何对该模块原地址空间的访问都会导致不可预期的错误。

模块自己的方法 cleanup_模块，是用于将模块从内核地址空间卸载的。它需要清除原模块的所有运行环境（注销模块的功能，释放模块占用的内存，去掉内核中模块间和部件间的相互依赖关系）。一旦调用了 cleanup_模块，内核或其他模块就不应该再引用该模块了，否则会导致存储器访问出错，造成系统崩溃。

1.4.4 在模块装载时给模块传递参数

在本节的开始我们提到过，在内核模块装载时可以给它传递参数。这些参数可以在使用 insmod 命令时直接给出，也可以在使用 modprobe 命令时，在配置文件中给出。为了可以给模块传递参数，你必须事先在模块的代码中声明这些参数，以下的宏就是用于声明模块参数的。

1．为模块声明参数

MODULE_PARM(var,type)：声明 var 是该模块的一个参数，在装载模块期间，可以给它分配一个值，type 指定了参数的类型。模块的参数可以是以下的类型。

- b：字节（byte）。
- h：短整型 （short 两个字节）。
- i：整型数（integer）。
- l：长整型（long）。
- s：字符串（string 或指向字符串的指针）。

2．模块参数描述信息

MODULE_PARM_DESC(var,desc)：给参数加入描述信息。这些描述信息在使用 modinfo 命令时，会显示对参数的描述。

1.4.5 内核和模块的符号表

内核的模块是目标代码，在运行时加入到内核中，一旦它被嵌入到内核后，模块就在内核的地址空间了。在模块被嵌入到内核之前，有几个问题需要注意，模块中的函数可能需要调用内核的函数，也可能需要使用内核的数据结构，所以我们首先需要展开这些函数和数据结构的地址。Linux 内核中包含了一个符号表 ksym，表中包含了所有符号（函数名、变量名）与地址的对应关系。

模块只能访问内核符号表中列出的函数和变量。你可以用命令 ksyms –a 来列出内核符号表的内容。

```
c0823100 u mem_map
c047e166 u __kmalloc
f884186c t cleanup_模块
```

```
c046a0a6 u register_shrinker
c043d4eb u prepare_to_wait
c047d5e2 u kfree
c0426460 u __wake_up
…
```

定义在 kernel/ksyms.c 中的指令 EXPORT_SYMBOL(xxx) 向符号表中加入内核的函数或变量，这样模块就可以访问这些符号了。除此之外，模块也可以向符号表输出模块的函数和变量的引用指针。宏 EXPORT_SYMBOL 允许模块向符号表加入选定的函数或数据指针。

1.5　内存资源

内存分配管理是实现高性能网络协议栈的关键因素，操作系统中的网络协议栈需要连续高可靠地运行，因此它需要频繁地为接收到的数据包分配缓冲区，同时也需要频繁释放处理完的数据缓冲区。

1.5.1　高速缓冲区（memory cache）

内核使用函数 kmalloc 和 kfree 来分配和释放内存块。这两个函数的使用语法与用户空间 C 库函数的 malloc 和 free 类似。在内核的各种组件中，经常需要为同一类型数据结构分配多个实例，如存放网络数据包的 Socket Buffer。当分配内存空间和释放内存的操作发生得非常频繁时，内核组件的初始化函数通常为自己初始化一个特殊的高速缓冲区来为其数据结构分配内存空间，在数据对象使用结束释放内存时，释放的内存返回给它分配的高速缓冲区。

内核的网络子系统中维护和管理的高速缓冲区包括以下几种。

1．套接字缓冲区描述符

套接字缓冲区描述符（socket buffer descriptor）的高速缓冲区，由 skb_init（定义在 net/core/sk_buff.c 文件中）函数初始化，用于为 sk_buff 缓冲区描述符分配内存。在网络子系统中，sk_buff 数据结构是分配、释放操作最频繁的一个数据结构。在第 2 章中，我们将详细介绍 sk_buff 数据结构的各数据域。

2．邻居协议映射高速缓冲区

在网络子系统中的每个邻居协议都使用一个内存高速缓冲区分配内存，来存放网络层至数据链路层（L3 到 L2）的地址映射关系。我们将在第 8 章讲解邻居协议的实现。

3．路由表

路由子系统中使用了两个主要数据结构来定义路由，相应的路由子系统初始化了两个内存高速缓冲存储器，为这两个数据结构的实例分配内存。

内核中管理内存高速缓冲区的关键函数有：kmem_cache_create 和 kmem_cache_destroy，用于创建和释放高速缓冲区。kmem_cache_alloc 从高速缓存中分配内存给数据对象，kmem_cache_free 释放内存对象，将缓冲区返回给高速缓存。

在内核的各组件中有各自对以上高速缓存管理函数的包装函数，所以在内核组件中，常通过包装函数来为数据对象分配内存和释放数据对象占用的内存空间。例如，释放一个 sk_buff 缓冲区的实例时，网络子系统调用 kfree_skb，kfree_skb 在所有进程对的引用都释放，完成所有清除工作后，最终调用 kmem_cache_free 来将缓冲区释放。

1.5.2 高速缓存和哈希链表

为了提高系统性能，最常采用的方法就是高速缓冲存储器（caching）。在网络子系统，使用了高速缓冲存储器来缓存已解析好的网络层地址（IP 地址）到数据链路层地址（MAC 地址）的映射关系，由路由表高速缓冲存储器来保存已寻址好的路由结果等。

缓存查询例程以输入参数为关键字在高速缓冲存储器中查询，查看查询结果是命中还是需要以某种算法在高速缓冲存储器中加入新缓存对象。

高速缓冲存储器常以哈希链表的机制实现，内核实现了一系列的数据结构类型，如单向和双向链表，用于建立简单的哈希表。在网络子系统中可以看到大量使用哈希表来提高查询数据对象的速度。

1.6 时间管理

在内核代码中常需要管理任务执行的时间。例如，一个进程调度到 CPU 上执行，进程调度器需要衡量该进程在 CPU 上运行的时间，当进程运行到一个指定时间后，需要将进程调度出 CPU，更换为其他进程。被调换出的进程只有等到下一次重调度到 CPU 中后再继续其任务。在网络子系统中我们常需要记录数据到达的时间，等待处理的时间等。

在内核地址空间，时间是以 tick 来衡量的。一个 tick 是两次时钟中断之间的间隔值。时钟需要管理各种不同的任务，周期性地产生中断来管理这些任务。时钟中断产生间隔以每秒钟多少 HZ 数来衡量。HZ 是一个变量，不同的 CPU 体系结构的初始值不一样。在 i386 系列的 CPU 中，HZ 被初始化为 1000，即当 Linux 运行在 i386 系统上时，时钟中断每秒钟产生 1000 次。

每次时钟中断产生时，时钟中断处理程序对全局变量 jiffies 递增 1，也即在任何时候 jiffies 的值代表了系统从启动到当前时间的 tick 数。

如果一个函数对时间的管理只需衡量经过的时间值，可以首先将当前 jiffies 值保存在一个局部变量中，随后将保存的值与当前 jiffies 比较（以 tick 数表示），看从记录开始已经过了多长时间。

以下给出了一个示例，函数需要完成某项功能，但占用 CPU 的时间不超过 1 个 tick。如果函数 do_something 已完成其操作，将标志 job_done 设置为非零值，函数返回。

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

```
    do_something(&job_done);
    if ( job_done)
        return;
}while ( jiffies - start_time < 1 );
```

1.7　嵌入式的挑战

什么是嵌入式系统？嵌入式系统是结合了硬件和软件，针对某特定应用领域和特定功能而定制的专用系统。所谓专用系统，是按照应用需求量身定制的系统，且这种定制包含了硬件和软件两方面的定制。嵌入式系统与通用系统相对应，就是其针对性强，不会造成资源的浪费，便于成本的控制。

嵌入式系统无论在其表现形式、大小上都是多种多样的，常规的嵌入式系统具有以下一些特点：

- 包含有一个处理引擎，如常规的微处理器。
- 为特殊应用或目标而专门设计。
- 有简单的用户界面，或没有用户界面，如自动引擎点火控制器。
- 资源有限，如只有很小的存储器空间，通常没有硬盘。
- 电源的电力有限，通常只有在电池供电的情况下工作。
- 通常不会采用通用目的计算平台。
- 嵌入了应用软件在系统中，软件不是由用户选择的。
- 系统中事先集成了所有应用硬件和软件。
- 预制应用，没有用户介入。

基于以上特点，嵌入式系统开发遇到的最大挑战就是资源有限，因此在有限资源的情况下要求嵌入式软件代码尽可能小，运行速度尽可能快，可靠性尽可能高。Linux 内核的组件化构成与模块化特性，使 Linux 内核易于裁剪来适应应用需求，经过裁剪后的内核，可固化在容量只有几十万字节或几十亿字节的存储器芯片或单片机中。Linux 内核所具有的独特性能，使 Linux 成为嵌入式操作系统首选。

1．组件化结构

Linux 内核是全组件层次结构，内核完全开放。Linux 内核由很多规模小且性能高的微内核和组件组成。在内核代码完全开放的前提下，不同领域和不同层次的用户可以根据自己的应用需要对内核进行改造，在低成本的前提下，设计和开发出真正满足自己需要的嵌入式系统。

2．强大网络支持功能

Linux 诞生于 Internet 应用，并具有 UNIX 的特性，这就保证了它支持所有标准 Internet 协议；Linux 网络体系结构的可配置性和可扩展性，保证可在 Linux 内核网络协议栈基础上将其开发成为嵌入式的 TCP/IP 网络协议栈以支持嵌入式系统的网络应用。

3．完整的开发调试工具链

Linux 具备一整套工具链，容易建立嵌入式系统的开发环境和交叉运行环境，一般开

发嵌入式操作系统的程序调试和跟踪都是使用仿真器来做，而使用 Linux 系统做原型的时候，可以绕过这个障碍，直接使用内核调试器来做操作系统的内核调试和查错。

4．广泛的硬件支持

无论是 RISC 还是 CISC，32 位还是 64 位等各种处理器，Linux 都能运行。Linux 除支持最常用的微处理器 Intel X86 芯片家族外，它同样能运行于 Motorola 公司的 68K 系列 CPU 平台；IBM、Apple、Motorola 公司的 PowerPC 体系结构的 CPU 以及 Intel 公司的 Strong ARM CPU 等处理器系统，这意味着使用 Linux 作为嵌入式操作系统将具有更广泛的应用前景。

1.8 本章总结

网络子系统和 TCP/IP 协议栈是在 Linux 内核代码中实现的一个组件，除了具有本身的特性外，它还具有作为内核组件的一些共同特点。本章在深入分析 Linux 网络子系统体系结构和 TCP/IP 协议栈在内核中的实现之前，概要介绍了理解 Linux 内核代码需要的一些预备知识，本章并没有深入到这些内核特点的内部展开讨论，只是给读者一个线索，以便在需要的时候读者可对这些知识做进一步的深入研究。

网络协议规范与协议实现有很大的区别，本书目的是分析 TCP/IP 协议栈在 Linux 内核的实现，在本书中我们只对协议做基本的介绍，我们基于假设读者对 TCP/IP 协议规范有基本了解。

第 2 章　Linux 网络包传输的关键数据结构——Socket Buffer

本章主要介绍 Linux 网络体系结构中存放网络数据的套接字缓冲区（Socket Buffer）的数据结构的作用和操作方式，在分析 Socket Buffer 数据结构（sk_buff）数据域的设计和各域含义的基础上，讲解在 Linux 内核 TCP/IP 协议栈中如何操作这些数据域，最后给出当数据包被分割时，存放数据片的数据结构及其与 Socket Buffer 的关联方式。

2.1　Socket Buffer 设计概述

在这一章和下一章中，我们将会介绍 Linux 内核网络体系结构实现的两个最重要的数据结构：Socket Buffer 数据结构 sk_buff 和网络设备数据结构 net_device。这两个数据结构所包含的数据域比较多，是网络子系统中两个较复杂的数据结构。一开始就逐一学习数据结构的组成和数据域的含义是一项非常枯燥的任务，要全部理解和记住它们所有的数据域也比较困难，为什么我们仍然要在本书的开始部分就花大量的篇幅对这两个数据结构的数据域进行详细分析呢？因为数据结构是理解程序运行方式和状态的关键因素。

什么是数据结构？它是系统中各个实体，如网络数据包、网络设备在内核中的符号描述和抽象，它表示了这些实体的属性和运行时间变化过程。程序的运行就是：读取数据结构的相关数据域，获取实体的属性，根据它们的当前值来分析数据包/网络设备的状态；修改数据结构的数据域或更新某些数据域的值来对数据包/网络设备作处理。数据结构设计得是否合理在很大程度上决定了：程序体系结构是否清晰；操作是否灵活；是否具备可扩展性。所以要理解 Linux 内核网络体系结构的设计思想和实现原理，首先应掌握代表网络子系统中各个实体的重要数据结构。

2.1.1　Socket Buffer 与 TCP/IP 协议栈

协议代表了通信双方的一种约定，使双方能理解彼此要交流的内容。就有如你要发一封信到英国，如果对方不懂中文，你需要用英语写信件；如果你不懂英语，对方也不懂中文，你们双方就需要约定第三种彼此都懂的语言来通信，这样双方才能沟通。TCP/IP 协议栈就是在计算机网络上通信双方的约定，TCP/IP 协议栈把复杂的网络通信协议分步实现。TCP/IP 协议栈与 OSI 参考模型如图 2-1 所示。

Socket Buffer 与 TCP/IP 协议栈的关系就如信件与写信的语言一样。网络数据包从应用程序传到内核时是原始数据，协议栈要在原始数据中加入通信约定。由于内核中 TCP/IP 协议栈是分层实现的，数据包在内核中要经过 TCP/IP 协议栈的传输层、网络层、数据链路层各层协议的处理，最后由网络设备发送出去，所以各层协议都要对表示网络数据包的数据结构进行操作。

对 Socket Buffer 数据结构的操作会贯穿在整个 TCP/IP 协议栈的各层。

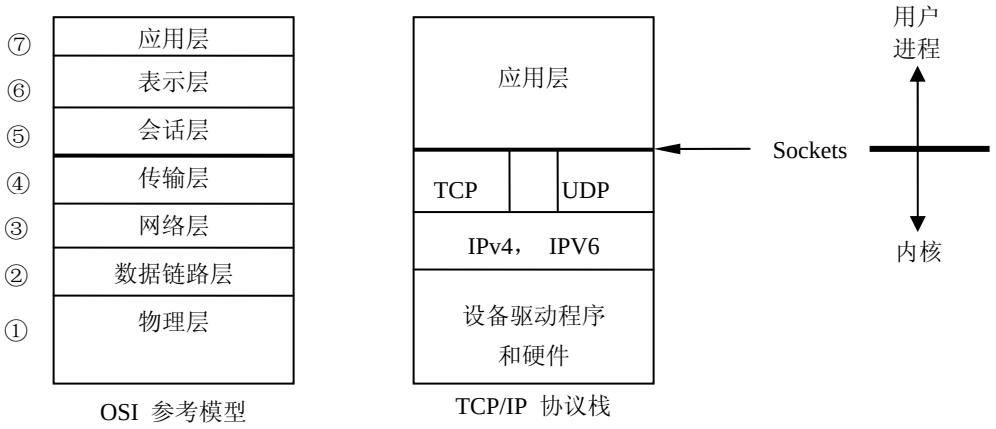


图 2-1 TCP 协议栈与 OSI 参考模型

2.1.2 Socket Buffer 的对外接口

如果各层协议处理程序都对 Socket Buffer 数据结构的数据域直接操作，会造成程序的结构混乱。一旦对 Socket Buffer 数据结构做出调整，则整个协议栈的代码都需要修改。所以内核中实现了一系列的方法来操作 Socket Buffer 的数据域，供 TCP/IP 协议栈各层处理函数调用，这些方法构成了 Socket Buffer 对外交流的统一接口。在本章中我们也会对其主要的方法和实现流程作详细的分析。这是理解应用 Linux 内核网络协议栈的重要基础。

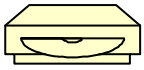
2.1.3 Socket Buffer 的特点

Linux 内核网络子系统的实现之所以灵活高效，主要在于其管理网络数据包的缓冲器-Socket Buffer 设计得高效合理。在 Linux 网络子系统中， Socket Buffer 是一个关键的数据结构，因为它代表了一个网络数据包在内核中处理的整个生命周期过程。也就是说 Socket Buffer 就是要接收或发送的网络数据包在内核中的对象实例。

2.2 Socket Buffer 的构成

Linux 内核网络子系统的设计目标是使网络子系统的实现独立于特定的网络协议，

这一设计目的不仅适用于网络层和传输层的协议（TCP/IP、IPX/SPX 等），也适用于网络适配器所采用的协议（Ethernet、token ring 等）。其他的网络协议不需做大的改动就能直接加入到 TCP/IP 协议栈的任何层次中。本章要讨论的内容的源代码都在以下目录中。



文件路径

Socket Buffer 的完整定义在 Linux 源文件的目录树： include/linux/草纲目 skbuff.h 文件中。

Socket Buffer 的构成如图 2-2 所示。

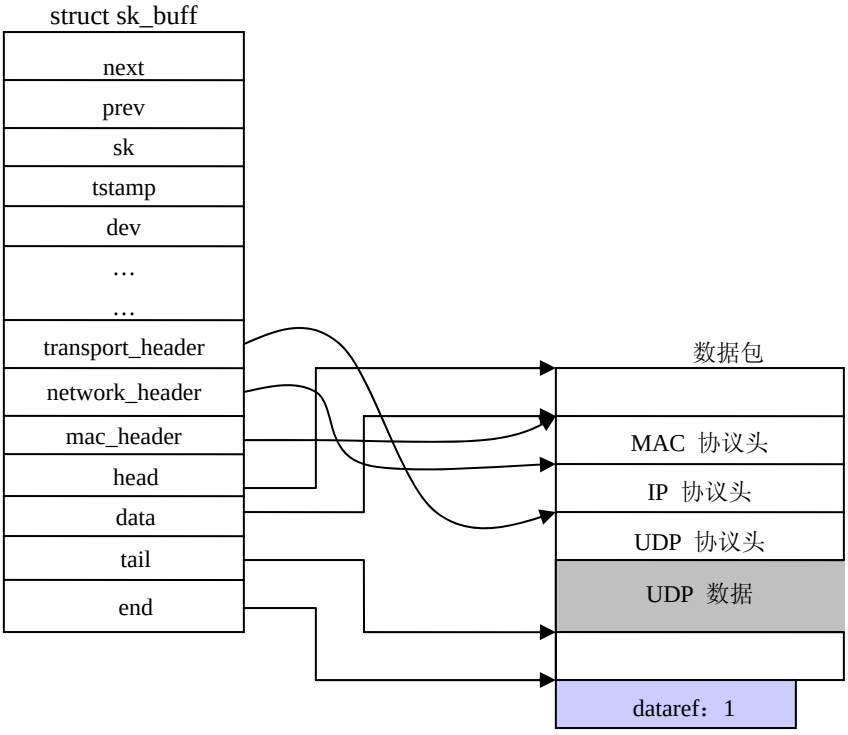


图 2-2　Socket Buffer 的构成示意图

2.2.1　Socket Buffer 的基本组成

一个完整的 Socket Buffer 主要由两个实体组成。

- 数据包：存放实际要在网络中传送的数据缓冲区。
- 管理数据结构（struct sk_buff）：当在内核中处理数据包时，内核还需要一些其他的数据来管理数据包和操作数据包，例如数据接收/发送的时间、状态等。这些信息是不需要向外发送的，所以不存放在数据包中。

数据包与 sk_buff 的关系就好比信件与信封的关系一样，信件是你要传递给他人的信息内容，信封告诉邮局如何将信件传到目的地。

在本书中，如果我们提到的是数据包和管理数据结构的整体，会用 Socket Buffer 来描述；如果只是管理数据结构，会以 sk_buff 来说明。

2.2.2 Socket Buffer 穿越 TCP/IP 协议栈

网络数据包在 TCP/IP 协议栈各层传送时，以上两个实体——数据包与管理数据结构的内容会不断发生变化。

1．发送网络数据包

例如，当网络应用数据从用户地址空间复制到内核地址空间时，在内核的套接字层会创建相应的 Socket Buffer 实例来将负载数据存放在数据包缓冲区中。数据包的地址、长度、状态等信息存放在 sk_buff 结构中。在 Socket Buffer 由上向下穿越 TCP/IP 协议栈的各层期间会发生如下事件。

- 各层协议的头信息会不断插入到数据包中。
- 相应的，在 sk_buff 结构中描述协议头信息的地址指针会被赋值。

所以，创建 Socket Buffer 时需要在数据包缓冲区前留出足够的空间来存放各层协议的头信息。

2. 接收网络数据包

网络适配器接收到发给本机的网络数据后，首先产生中断通知内核收到了网络数据帧。接着在网络适配器的中断处理程序中调用 dev_alloc_skb 函数向系统申请一个 Socket Buffer，将接收到的网络数据帧从设备硬件的缓冲区复制到 Socket Buffer 的数据包缓冲区；填写 sk_buff 结构中的地址、接收时间和协议等信息。

这时：Socket Buffer 到达了内核地址空间。当 Socket Buffer 由下向上穿越 TCP/IP 协议栈的各层时将发生如下事件。

- 各层协议的头信息会不断从数据包中去掉。
- 相应的在 sk_buff 结构中描述头信息的地址指针会被复位，并调整 sk_buff 结构中指向有效数据的 sk_buff->data 指针。

图 2-3 给出了 Socket Buffer 在传送过程中的变化示意图。

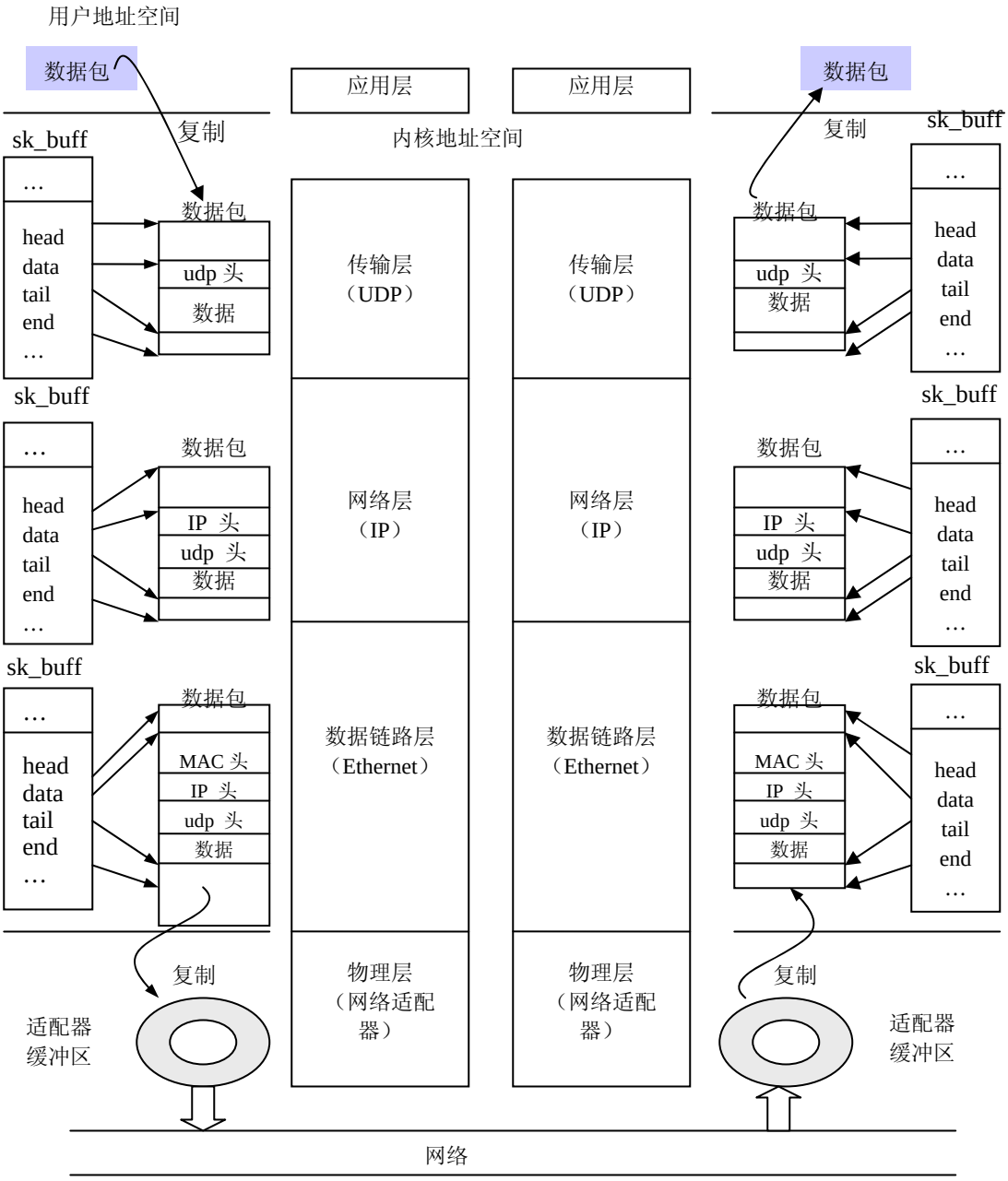


图 2-3　网络数据包在 TCP/IP 协议栈的传送示意图

3. 设计优点

Socket Buffer 这样组织的优点是避免了重复复制数据，要传送的数据只需复制两次：一次是从应用程序的用户地址空间复制到内核地址空间；一次是从内核地址空间复制到网络适配器的硬件缓冲区中。

2.3 sk_buff 数据域的设计和含义

从上一节的叙述我们可以看到，网络数据是从用户地址空间或网络适配器硬件缓冲区传送到内核地址空间的。Socket Buffer 的数据包在穿越内核地址空间的 TCP/IP 协议栈过程中，数据内容通常不会被修改，只会有数据包缓冲区中的协议头信息会发生变化。大量操作是在 sk_buff 数据结构中进行的。

1．sk_buff 设计说明

sk_buff 结构所包含的域和域的组织在内核的开发过程中做了多次添加和重新组织，在尽可能地使 sk_buff 在结构和布局上显得清晰的基础上，还考虑了数据传送的高效性。例如，按系统的 cache line 的大小对齐（cache Line 的大小与 CPU 的体系结构相关，一般为 64 个字节），这样在系统总线上以 burst 方式传送，可更快速地读取 sk_buff 的数据。另一方面随着 Linux 网络功能的增强， sk_buff 数据结构中也增加了一些新的数据域来标识对这些新功能的支持，比如对 IP 虚拟服务器的支持（ipvs_property），包过滤的支持等。

2．sk_buff 数据域分类

sk_buff 的域从功能上大致可以划分成以下几类。

- 结构管理。
- 常规数据域。
- 网络功能配置相关域。

2.3.1 sk_buff 中的结构管理域

sk_buff 中的某些域是为了方便组织和搜索数据结构本身的。在内核中，所有 Socket Buffer 根据其状态和类型（比如，接收、向外传送、前送或已处理完成的 Socket Buffer）存放在不同的队列中。队列中的 Socket Buffer 链接为双向链表，队列的首地址保存在 sk_buff_head 结构类型的指针变量中。

1．*next 和*prev

队列中将 sk_buff 链接成双向链表的前向指针和后趋指针。sk_buff 结构中的 next 指向链表中的下一个成员，prev 指针指向链表中的前一个成员。引进一个新的数据结构保存该双向链表队列的起始地址，该数据结构为：

```
struct sk_buff_head{
  struct sk_buff   *next;
  struct sk_buff   *prev;
  __u32            qlen;
  spinlock_t       lock;
}
```

其中 qlen 为链表中 sk_buff 结构实例成员的个数，lock 用于保护该双向链表的锁，以防并发访问链表。图 2-4 描述了 Socket Buffer 在内核中的组织。

内核如此管理 Socket Buffer 的优点在于：某个 Socket Buffer 的状态变化了，需要将 Socket Buffer 在各队列之间移动时，无须复制整个缓冲区，只需修改 next 和 prev 指针，就

可将缓冲区从一个队列放入另一个队列中管理。

2．struct sock *sk

指向拥有该 Socket Buffer 的套接字（socket）数据结构的指针。当数据是由本机的应用产生，将要向外发送时，或者从网络来的数据包的目标地址是本机应用程序时，这个数据域需要设置。

所谓套接字就是端口号加 IP 地址，用来唯一识别系统中的网络应用程序。sk_buff->sk 数据域表示了该网络数据包最终应传送给哪个应用程序。所以如果一个数据包是从网络上接收，经由本机继续向前传送时，这个域的值为空（NULL）。

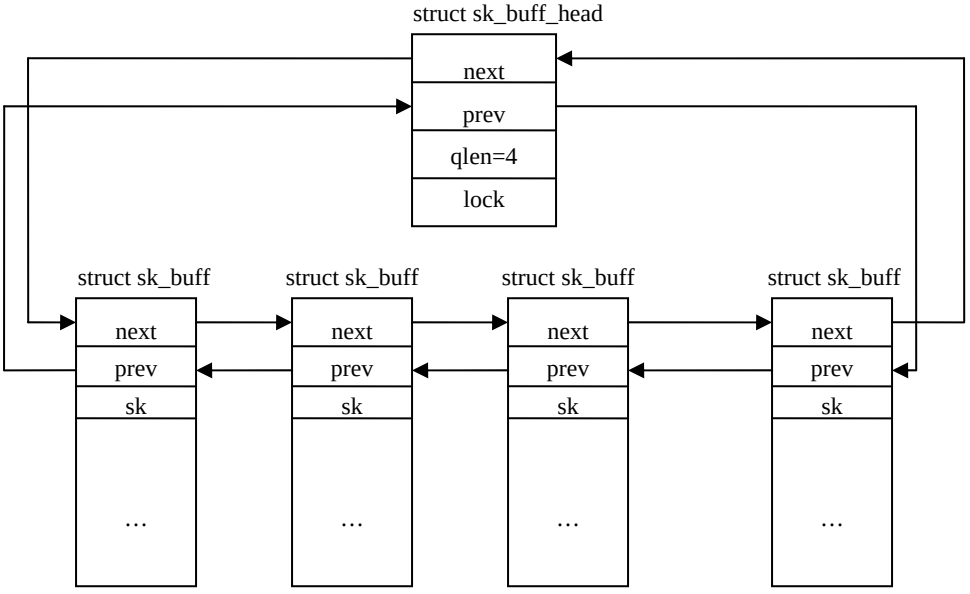


图 2-4　Socket Buffer 在内核中的组织

3．unsigned int len

该域指明 Socket Buffer 中数据包的长度。这个长度是数据总长度值，即其值包括了：①主缓冲区中的数据长度（由 sk_buff->head 指针指向的地址存放的数据）；②各分片数据的长度（当应用程序产生的数据大于网络适配器硬件一次能传送数据的最大传送单元 MTU 时，数据包会被分成更小的片段。数据包的分片在 2.5 节中介绍）。

sk_buff->len 在数据包通过协议栈的各层时其值也在发生变化，因为各层的协议头信息在不断加入或从 Socket Buffer 中去掉。由此可见，sk_buff->len 也包含协议头的长度。

4．unsigned int data_len

data_len 只精确计算被分了片的数据块长度。

5．__u16 mac_len

数据链路层（如 Ethernet）协议头的长度。

6．__u16 hdr_len

hdr_len 是针对克隆数据包时使用的，它表明克隆的数据包的头长度。当我们克隆（克隆的含义会在 2.4.3 节中讲解）Socket Buffer 时，可以只做纯数据克隆（即不克隆数据包的协议头信息），这时克隆 Socket Buffer 需要从 hdr_len 中获取头长度。

7．atomic_t users

引用计数，或称为所有正在使用该 sk_buff 缓冲区的进程计数。这个参数的主要作用是防止 sk_buff 还在使用就被释放了。任何进程要使用 sk_buff 时，都应对 sk_buff->users 域加 1，使用完成后应对 sk_buff->users 域减 1。

sk_buff->users 域记录对 sk_buff 结构的引用计数。数据包缓冲区的使用计数由 dataref 记录。用户可以用 atomic_int 和 atomic_dec 函数来直接做加 1 和减 1 操作，但更多的是使用包装函数 skb_get 和 free_skb 来操作 sk_buff->users，这样更安全。

8．unsigned int truesize

记录整个 Socket Buffer 的大小，即 sk_buff 数据结构的长度和数据包的长度和。它是由 alloc_skb 函数将其值初始化为 len+sizeof(sk_buff)的。

创建 Socket Buffer 时，我们调用 alloc_skb 函数来向系统申请内存。该函数需要一个参数为数据包预留的内存空间大小，即 sk_buff 数据结构中 len 域的值（由参数 size 给出），如：

skb->truesize 的值会随着 skb->len 的值变化而变化。

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

为 Socket Buffer 分配内存时，其大小为：sk_buff 数据结构的大小与数据包大小的总和

9．sk_buff_data_t tail;

```
    sk_buff_data_t      end;
    unsigned char *head, *data
```

Socket Buffer 中数据包缓冲区的内容包括：①TCP/IP 协议栈各层的协议头信息；②负载数据。这是最终在网络上传送的内容。以上的几个域代表了数据包缓冲区中各种信息的边界。

当每层协议处理函数为自己的活动准备空间时，在 head 和 end 之间分配自己所需的空间。head 和 end 指向整个数据包缓冲区的起始和结束地址，data 和 tail 指向实际数据的起始和结束地址，各层协议处理函数可以在 data 和 head 之间的空隙处填写头信息，在 tail 和 end 之间放新的数据，如图 2-5 所示。

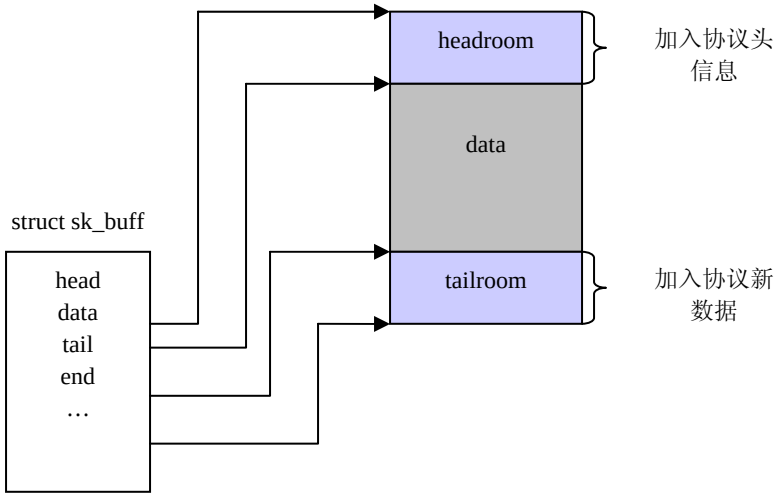


图 2-5　Socket Buffer 数据包与管理数据的指针

10．void(*destructor) (…)

destructor 函数指针可以在 sk_buff 数据结构初始化时指向 Socket Buffer 的析构函数。在释放 Socket Buffer 时，完成具体的清除工作。当 sk_buff 缓冲区不属于任何套接字时，析构函数通常不需要初始化。

这些管理域全面描述了 Socket Buffer 的各种管理信息：Socket Buffer 应放在哪个队列，有多少进程在访问 sk_buff，有多少进程在访问数据包，各种信息的大小、边界等。

2.3.2　常规数据域

这一节介绍 sk_buff 数据结构中的一些常规域，这些常规域是描述 Socket Buffer 状态的主要数据域，也是 TCP/IP 协议栈处理程序操作最频繁的数据域。

1．ktime_ttstamp

描述了接收数据包到达内核的时间。由接收数据包处理函数 netif_rx 调用 net_timestamp(skb)来对该数据域赋值。netif_rx 是由网络设备驱动程序在收到网络数据时调用的（网络设备驱动程序和 netif_rx 函数实现功能会分别在第 5 章及第 6 章中讲解）。

2．struct net_device *dev

Int iif

dev 是指向代表网络设备数据结构的指针，它表明该数据包是通过哪个网络设备接收或传送的。当网络设备从网络收到一个数据包时，设备驱动程序将该域更新为一个 net_device 类型的指针，指向接收该数据包的网络设备，以下程序示例是 cs89x0 网络设备驱动程序，其接收数据包的函数为：net_rx。

当传送数据包时，dev 代表的是数据包将通过哪个网络设备发送出去。

```
文件目录　以下两个函数分别在 drivers/net/cs89x0.h 和 net/ethernet/eth.c
static void net_rx ( struct  net_device *dev)              //网络设备驱动程序接收函数
{
  …
```

```
        skb->protocol = eth_type_trans(skb, dev);          //调用设置接收的协议和网络设备
    netif_rx(skb);
    …
}
__be16 eth_type_trans(struct sk_buff *skb, struct net_device *dev))
{
    …
    skb->dev = dev;                                        //设置接收该数据包的网络设备
}
```

iif 数据域为接收/发送数据包的网络设备索引号。在 Linux 系统中，某种类型的网络设备的命名方式是以字符开头的设备名后跟顺序索引号。例如 Ethernet 类型的设备名为 eht0、eth1 等。设置索引号便于快速查找到网络设备。

3．union

```
    {
        struct dst_entry *dst;
        struct rtable         *rtable;
    };
```

由路由子系统使用，当接收到的数据包的目标地址不是本机，只是通过本机继续向前发送时，则这个数据域包含的信息指明应如何将数据包向前发送。该域是一个联合域，它或者指向路由表中的一条路由记录，其中存放了数据包的发送目标 IP 地址；或者指明应在系统的哪个路由表中查找数据包发送地址的信息。

4．char cb[48]

这是一个控制缓冲区（Control Buffer），是各层协议在处理数据包时存放私有信息或变量的地方。各层协议可以自由使用该控制缓冲区。在当前的内核版本中，它的大小是 48 个字节。例如在传输层，UDP 协议用控制缓冲区来存放它的 udp_skb_cb 数据结构。

```
文件目录  以下数据结构定义在 include/net/udp.h 文件中。
struct udp_skb_cb {
  union {
        struct inet_skb_parm   h4;
#if defined(CONFIG_IPV6) || defined (CONFIG_IPV6_MODULE)
        struct inet6_skb_parm  h6;
#endif
  } header;
  __u16      cscov;
  __u8       partial_cov;
};
```

为了使代码更易理解，在各层中通常都定义了宏来访问控制缓冲区，以下代码是 UDP 访问其私有数据的宏：

```
#define UDP_SKB_CB(__skb)((struct udp_skb)_cb *)((__skb)->cb)
```

以下是在初始化过程中对 UDP 数据包做校验和时，填写控制缓冲区的代码：

```
文件目录  net/ipv4/udp.c
static inline int udp4_csum_init(struct sk_buff  *skb, struct udphdr  *uh, int proto)
{
  …
  UDP_SKB_CB(skb)->partial_cov = 0;
  UDP_SKB_CB(skb)->cscov = skb->len;      // 对整个 Socket Buffer 做校验和
```

```
    …
}
```

如果你需要让控制缓冲区的信息跨协议层传送，则必须克隆 sk_buff。

5．csum

```
__u8    ip_summed
```

csum 域存放数据包的校验和。校验和是检验网络数据包在接收/发送过程中是否损坏的手段。发送数据包时，我们将数据从用户地址空间复制到内核地址空间，同时以相应的算法计算数据包的校验和存放于该域。接收数据包时，csum 中存放的是网络设备计算的校验和。其中必须包含 csum_start/csum_offset 对。

- csum_start：以 skb->head 为起始地址的偏移量，指出校验和从数据的什么位置开始计算。
- csum_offset：以 csum_start 为起始地址的偏移量，指明校验和存放的地址。

ip_summed 描述的是网络设备是否可用硬件来对 IP 数据进行校验编码或解码。ip_summed 用两位编码来描述网络设备硬件对校验和的支持，它是由设备驱动程序反馈的信息，它的合法值在表 2-1 中给出。

表 2-1　ip_summed 数据域的值定义

ip_summed 值的符号	描　　述
CHECKSUM_NONE	网络设备不具有计算校验和的功能
CHECKSUM_UNNECESSARY	不需要对数据包检验校验和，这个值一般用于 loopback 设备
CHECKSUM_COMPLETE	网络设备硬件具有计算校验和的功能
CHECKSUM_PARITAL	针对输出数据包，要求设备对由 hard_start_xmit 函数发送的数据包做校验和，校验和的范围从 csum_start 指明的地址起始到数据包结束处，校验和存放在 csum_start + csum_offset 的地址

6．__u32 priority

priority 数据域是用来实现质量服务（Quality of Service）QoS 功能特性的。QoS 描述了数据包传送的优先级别。例如网络视频数据、语音数据需要使用 QoS 以保证视频、语音的流畅。

该域的值说明要传送或前送的数据包在传送队列中的优先级。如果数据包是本地产生的，则由套接字（socket）层填 priority 的值；如果数据包是要前送的，则由路由子系统的函数根据数据包中 IP 协议头信息的 Tos（Type of Service，服务类型）域来填写 priority 的值。

1	1	2	1	2
local_df	cloned	ip_summed	nohdr	nfctinfo

priority 将一个字节划分成不同的位域，标记 sk_buff 的状态以及 sk_buff 中与网络特性相关的状态。ip_summed 在该字节中占两位，编码网络设备对数据帧做的校验和的类型，其值我们已在前面介绍了。其余几个位域的作用如表 2-2 所示。

表 2-2　位域定义说明

域　名	说　明
local_df	由 IP 协议使用的，标明是否允许对已经分割的数据片段在本地进一步分割成更小的片段，这种情况在用 IPSEC 时会用到
cloned	为了共享 Socket Buffer 中的数据，节省内存空间（如某个数据包有多个协议处理函数对其进行处理），Linux 中使用了克隆 sk_buff 数据结构的概念。当克隆一个 sk_buff 数据结构时，原 sk_buff 数据结构的 cloned 和克隆的 sk_buff 数据结构的 cloned 域都要设置为 1。而数据包为这几个 sk_buff 结构共享
nohdr	数据包克隆时，是否克隆协议头
nfctinfo	如果当前本机有网络连接，该标志说明该 Socket Buffer 是否与这些网络连接相关

3	2	2	1	1
pkt_type	fclone	ipvs_property	peeked	nf_trace

这是另一个划分成不同位域以标明 Socket Buffer 特性的字节。

- pkt_type：数据包的类型。数据包的类型是按照数据链路层中设置的目标地址类型来划分的，如表 2-3 所示。

表 2-3　数据包类型值定义

文件目录　include/linux/if_packet.h	
符 号 定 义	描　述
PACKET_HOST	数据包的目标地址就是本机
PACKET_MULTICAST	数据包的目标地址为组传送地址，该组传送地址是本机的网络接口注册了的组传送地址之一
PACKET_BROADCAST	广播报文数据包
PACKET_OTHERHOST	数据包的目标地址不是本机，如果该数据包允许继续发送，则向外发送，否则扔掉
PACKET_OUTGOING	向外发送的数据包
PACKET_LOOPBACK	loopback 设备发送的数据包
PACKET_FASTROUTE	指定的数据包要在两个网络适配器间快速传送

- fclone：sk_buff 克隆类别，它的值与含义如下。
  - SKB_FCLONE_UNAVAILABLE：sk_buff 未克隆。
  - SKB_FCLONE_ORIG：被克隆的 sk_buff。
  - SKB_FCLONE_CLONE:克隆的 sk_buff。
- ipvs_property：该 Socket Buffer 是否是 IP 虚拟服务器拥有的数据包。
- peeked：为 UDP 协议使用的，在 Linux 以前的版本中，裸的 UDP 数据包的状态都是独立处理的。这样的问题是可能存在重复设置 UDP 数据包状态。使用 peeked 位后，说明该数据包在前面的处理过程已见到了，其状态已设置好，不需再设备。
- nf_trace：该标志是为网络过滤功能（Netfilter）使用的，是包过滤的跟踪标志。当该位被置 1，说明需要对数据包进行过滤检查。

7．__be16 protocol

接收数据包的网络层协议（如 IP 协议）。它标志了网络数据包应传给 TCP/IP 协议栈网络层的哪个协议处理函数。

protocol 域协议的完整定义在 include/linux/if_ether.h 头文件中，该域是由网络适配器的驱动程序调用相关函数来填写的，可能的协议如：

#define ETH_P_802_3、#define ETH_P_802_2　等。

8．__u16 queue_mapping

具有多个缓冲队列的网络设备的队列映射关系。早期的网络设备只有一个数据发送缓冲区，现在很多的网络设备都有多个发送队列来存放待发送的网络数据包，queue_mapping 描述了发送网络数据包所在队列与设备硬件发送队列的映射关系。

9．__u32 mark

数据包为常规数据包的标志。

10．__u16 vlan_tci

虚拟局域网的标记控制信息（Vlan Tag Control Information）。

11．struct sec_path　*sp

安全路径（Security Path），用于发送帧。它是用于 IPsec 协议跟踪网络数据包的传送路径。

虚拟局域网与 IPsec 在本书不会讲解到，我们就不对以上三个数据域做更详细的分析了。

12．sk_buff_data_t transport_header

sk_buff_data_t network_header

sk_buff_data_t mac_header

以上 3 个域是 sk_buff 结构中描述 Linux 内核网络协议栈中各层协议头在网络数据包的位置信息，它们各自的含义如下。

- transport_header：传输层协议头在网络数据包中的地址。
- network_header：网络层协议头在网络数据包中的地址。
- mac_header：数据链路层协议头在网络数据包中的地址。

如果定义了使用偏移量（针对 64 位 CPU 体系结构），这几个值是指相应协议头在网络数据包中的地址是以 skb->head 为起始的偏移量，否则 sk_buff_data_t 的类型就为指针（用于 32 位 CPU 体系结构），存放各层协议头在网络数据包中的起始地址。

```
文件目录 sk_buff_data_t 定义在 include/linux/skbuff.h 中。
#ifdef  NET_SKBUFF_DATA_USES_OFFSET
typedef unsigned int sk_buff_data_t;
#else
typedef unsigned char *sk_buff_data_t;
#endif
```

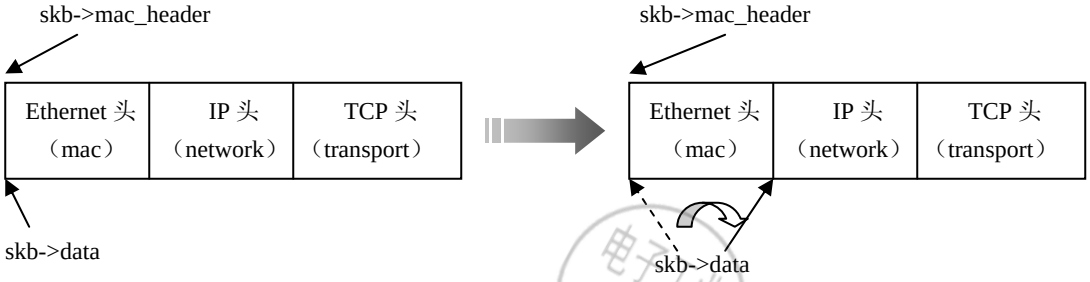
以上面这种方式定义，其优越性表现在以下几个方面。

- 使程序实现更灵活。因为在内核中 TCP/IP 协议栈的各层都存在着多个网络协议实例，每种协议的头信息长度不一样，将协议头在数据包中的地址信息以偏移量或指针的方式来描述，我们就不必关心具体协议头的长度。

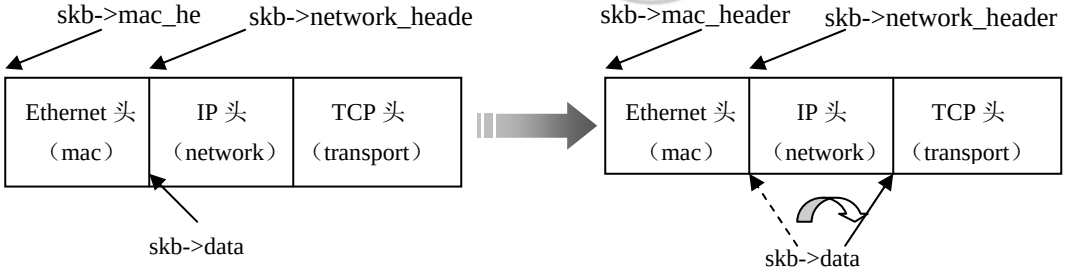
- TCP/IP 协议栈各层协议头信息独立操作。在接收数据包的过程中，TCP/IP 协议栈中负责处理第 n 层的处理函数接收从第 n-1 层传来的 sk_buff。这时 sk_buff 的 data 指针指向的是第 n 层协议头的起始位置，如图 2-6（a）所示。

第 n 层协议处理函数在处理数据包之前需要解析协议头的信息，才能确定应对数据做何操作。第 n 层的协议处理函数应初始化该层的头指针指向其头信息来读取协议头数据。因为第 n 层的协议头在数据包传到第 n+1 层后就不再需要了，所以第 n 层的处理程序在把数据包传到第 n+1 层之前，要把 skb->data 指针的位置调整到第 n 层头信息结束的地址，也就是第 n+1 层协议头信息的起始位置。这时如果没用头指针保存头信息的地址，头信息就不再可访问。其次，解析协议头信息时不直接操作 skb->data，可保证数据包的完整性和协议头操作的独立性。

发送数据包的过程正好与此相反，是在每层加入头的信息。头指针和 skb->data 指针的操作如图 2-6 所示。



（a）数据包从 L2 层传到 L3 层时头指针的调整



（b）数据包从 L3 层传到 L4 层时头指针的调整

图 2-6　Socket Buffer 头指针的操作

2.3.3　sk_buff 的网络功能配置域

Linux 内核的网络子系统实现了大量功能，这些功能的实现是模块化的，用户可根据实际需要配置自己的系统，决定内核包含什么功能，不包含什么功能。例如，你可以将系统配置为路由器，或将系统配置为桥设备。根据内核配置，sk_buff 数据结构中包含的数据域也随之变化。有的域只有在配置内核时选择了这些功能支持，编译器才会将这些数据域

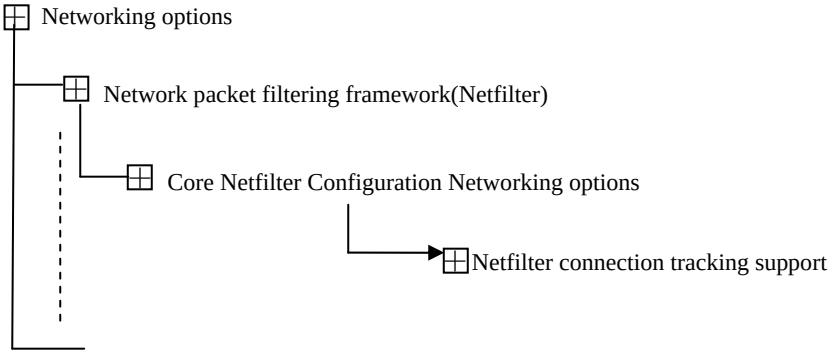
根据条件编译包含在 sk_buff 数据结构中。这些域的定义形式通常如下。

```
#if defined ( CONFIG_NF_CONNTRACK ) ||
    defined ( CONFIG_NF_CONNTRACK_MODULE )
  struct nf_conntrack        *nfct;
  struct sk_buff       *nfct_reasm;
#endif
```

1．连接跟踪

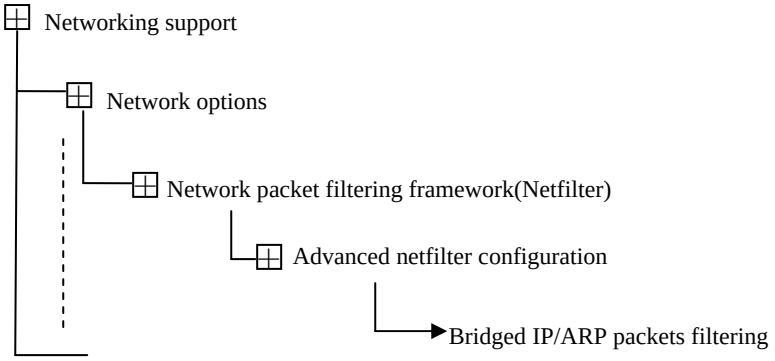
连接跟踪可以记录有什么数据包经过了你的主机以及它们是如何进入网络连接的。要支持连接跟踪功能，可以在 Linux 的源码树目录下运行以下几条命令。

- make xconfig：进入内核配置的图形界面，在内核配置的树形结构中做以下选择。该功能也可配置成模块，在运行时动态加载到系统中。
- nfct：用于连接跟踪计数。
- nfct_reasm：用于连接跟踪时数据包的重组装。



2. 桥防火墙（CONFIG_BRIDGE_NETFILTER）

为桥设备配置防火墙功能，该功能是通过以下内核配置加入系统的：



nfct_bridge：存放桥数据包的数据结构。

3. 流量控制（CONFIG_NET_SCHED）

当内核有多个数据包要通过网络设备向外发送时，内核必须决定哪个先送，哪个后送，哪个数据包扔掉，内核实现了多种算法来选择数据包。如果没有选择该功能，内核发送数据包时就选用最常规的先进先出（FIFO）策略。

流量控制功能通过以下内核配置加入系统。

- tc_inde：存放数据包发送选择算法的索引号。
- tc_verd：存放选择流量控制后对数据包所做处理行动（Action）的索引号，如排序、优先级设置等。

其他的条件编译与此类似，在 sk_buff 数据结构中与网络功能配置相关的其他数据域如表 2-4 所示。

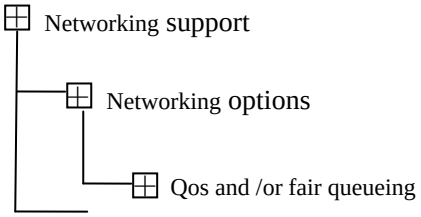


表 2-4　sk_buff 中与网络功能选择配置相关的数据域

数　据　域	描　　述
ndisc_nodetype	路由器的类型
do_not_encrypt	不封装该数据帧
dma_cookie	在多个 DMA 操作中，由 sk_buff 的 DMA 函数完成的 DMA 请求
secmark	允许有安全标记的数据包通过主机

以功能配置的方式来决定 sk_buff 中数据域的构成，用户可以根据自己的系统需要量体裁衣，特别是在嵌入式系统中，系统资源极其有限的情况下，依据系统资源配置内核组件尤其重要。配置内核时只保留必需的功能，以最大限度地节省系统开销。

2.4　操作 sk_buff 的函数

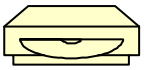
Linux 内核中实现了大量简单实用的函数来操作 sk_buff 的数据域和管理 sk_buff 的存放队列。内核提供这些操作函数的目的是使对 sk_buff 的操作与 TCP/IP 协议栈的实现相独立。以后无论是对 sk_buff 进行扩展还是在 TCP/IP 协议栈中实现了新的网络协议实例，原来的代码无须修改，只需扩展 sk_buff 的操作 API 就可实现前后操作的一致性。

在下面几种情况下需要应用 sk_buff 的操作函数：开发网络设备驱动程序时；在 TCP/IP 协议栈基础上开发新的网络协议实例时。

这些函数从其功能上可以划分为以下几类。

- 创建、释放和复制 Socket Buffer。
- 操作 sk_buff 结构中的参数和指针。

- 管理 Socket Buffer 的队列。


文件路径

sk_buff 操作函数的实现集中在两个源文件中：
net/core/skbuff.c 和 include/linux/skbuff.h。

说明：在阅读 Linux 的源代码时，你可以看到很多同名函数有两个版本，一个是不带下画线的如 do_something，一个是带双下画线的如__do_something。带双下画线的函数是实际的功能实现裸函数，不带下画线的函数通常是带双下画线函数的包装函数。包装函数在裸函数基础上增加了安全检查、实现防止并发访问的锁定机制等功能，或向实现函数传送正确的参数，建议一般不要直接调用带双下画线函数。大部分包装函数定义在 include/linux/skbuff.h 头文件中，而裸函数定义在 net/core/skbuff.c 源文件中。

2.4.1　创建和释放 Socket Buffer

创建 Socket Buffer 比常规内存分配复杂。在高速网络的环境下，每秒有几千个网络数据包接收/发送，需要频繁地创建/释放 Socket Buffer。如果设计 Socket Buffer 的创建/释放过程不合理，将大大降低整个系统的性能。尤其内存分配是系统最耗时的一个操作。

为此内核在系统初始化时已创建了两个 sk_buff 的内存对象池。

- skbuff_head_cache
- skbuff_fclone_cache

这两个对象池是由 skb_init 创建的（net/core/skbuff.c）。每当需要为 sk_buff 数据结构分配内存时，根据所需的 sk_buff 是克隆的还是非克隆的，分别从以上两个 cache 中获取内存对象，释放 sk_buff 时也就将对象放回以上两个 cache。

1．创建 Socket Buffer

Linux 内核中实现的各种创建 Socket Buffer 的函数如表 2-5 所示。

表 2-5　创建 Socket Buffer 的函数列表

创建 Socket Buffer 函数原型	所 在 文 件
__alloc_skb (unsigned int size,　gfp_t　gfp_mask, int fclone,　int node)	skbuff.c
alloc_skb　(unsigned int size, gfp_t　priority)	skbuff.h
alloc_skb_fclone (unsigned int size, gfp_t　priority)	skbuff.h
dev_alloc_skb (unsigned int length)	skbuff.c
__dev_alloc_skb (unsigned int length , gfp_t　gfp_mask)	skbuff.h
__netdev_alloc_skb(struct net_device *dev,　unsigned int length, gfp_t gfp_mask)	skbuff.c
netdev_alloc_skb (struct net_device *dev,　unsigned int length)	skbuff.h

（1）__alloc_skb

__alloc_skb 是为 Socket Buffer 分配内存的主要函数。创建 Socket Buffer 需考虑的主要因素有：　应从哪个 cache 中获取内存对象；　__alloc_skb 会在哪里调用；　Socket Buffer 的数据结构两个实体单独分配。对以上因素的处理都体现在__alloc_skb 的传入参数上，如下。

- size：存放数据包需要的内存空间的大小。
- gfp_mask：做内存分配时的标志。该标志描述了__alloc_skb 函数的运行特点，

即可否被别的进程中断。

- GFP_ATOMIC：在中断处理程序中申请内存，gfp-mask 标志必须为该值，因为中断不能休眠。
- GFP_KERNEL：表示常规内核函数申请内存分配。
- 其他值。

- fclone：是否对该 sk_buff 克隆。其值决定了从哪一个 sk_buff 内存对象池中获取 sk_buff 数据结构所需的内存空间。
  - 0：skbuff_head_cache
  - 1：skbuff_fclone_cache

__alloc_skb 完成内存分配的流程如图 2-7 所示。

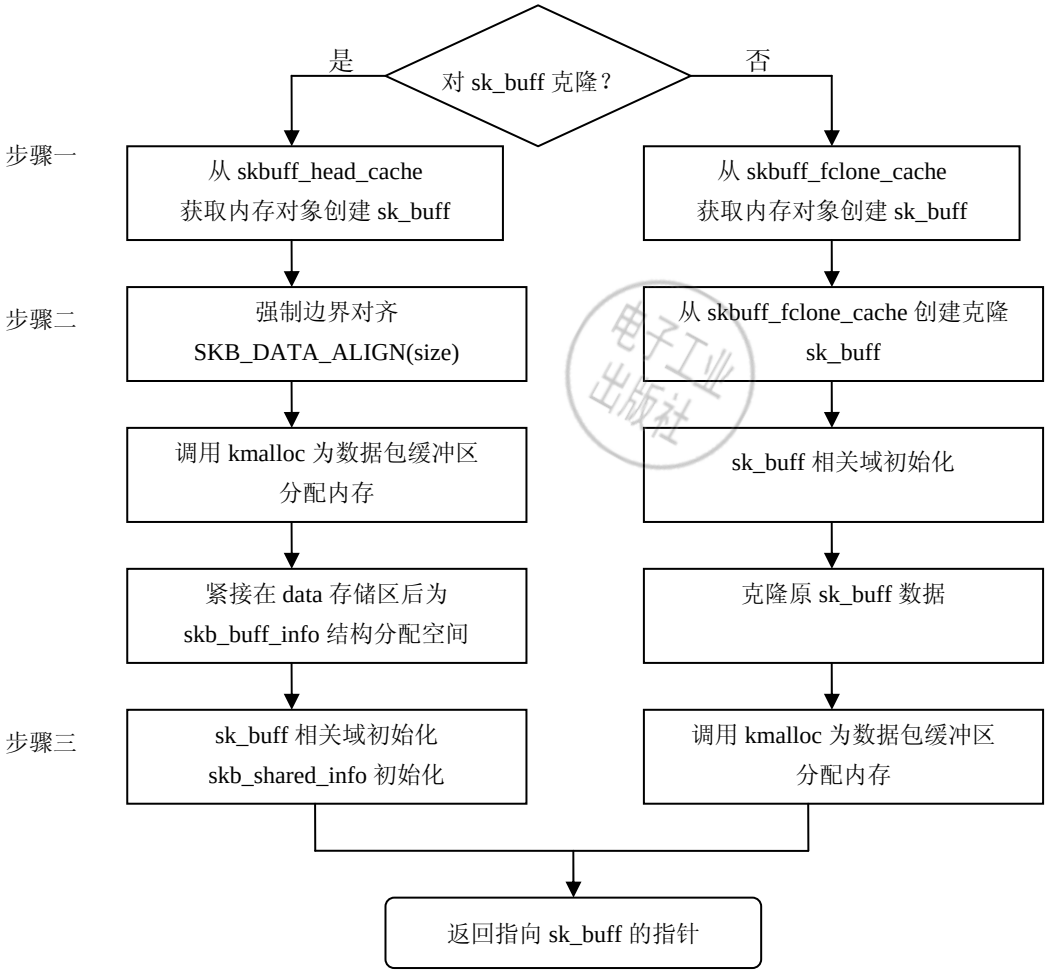


图 2-7　__alloc_skb 创建 Socket Buffer 的实现流程

Socket Buffer 创建结果如图 2-8 所示。

（2）alloc_skb

alloc_skb_fclone

我们在为 Socket Buffer 分配内存时，应调用它的两个包装函数，它们将为__alloc_skb 传递正确的参数。

```
// 为非克隆 Socket Buffer 分配内存
static inline struct sk_buff *alloc_skb(unsigned int size, gfp_t priority)
{
  return __alloc_skb(size, priority, 0, -1);
}
// 为克隆 Socket Buffer 分配内存
static inline struct sk_buff *alloc_skb_fclone(unsigned int size,gfp_t priority)
{
  return __alloc_skb(size, priority, 1, -1);
}
```

fclone=0

fclone=1

（3）_dev_alloc_skb

__dev_alloc_skb

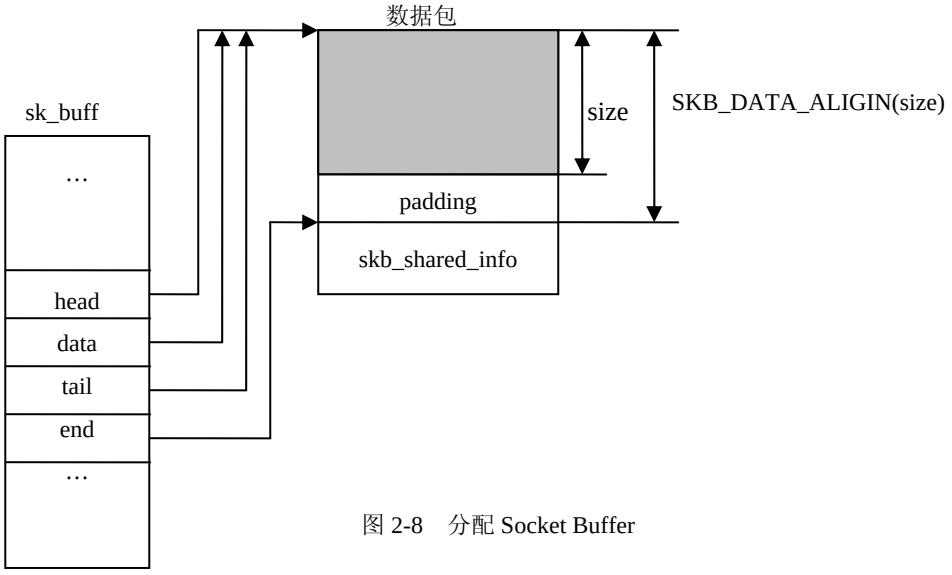


图 2-8　分配 Socket Buffer

__dev_alloc_skb 是给网络设备驱动程序使用的函数，当网络设备从网络上收到一个数据包时，它调用该函数向系统申请缓冲区来存放数据包，它是 alloc_skb 的包装函数，在其中调用了 alloc_skb 来分配 Socket Buffer。

为了提高效率，__dev_alloc_skb 为数据链路层在数据包缓冲区前预留了 16 个字节的 headroom，（NET_SKB_PAD，其值为 16），以避免头信息增长时原空间不够而重新分配内存空间造成额外系统开销。

现代网络设备数据链路层大多数使用的是以太网协议，协议头的长度为 14 个字节，预留 16 个字节是为了保证数据在 2^n 的边界对齐。

dev_alloc_skb 是__dev_alloc_skb 的包装函数。由于网络数据包是在中断处理程序中接收的，中断不能休眠，dev_alloc_skb 用 GFP_ATOMIC 标志传给__dev_alloc_skb 来分配内存。

```
static inline struct sk_buff *__dev_alloc_skb(unsigned int length,gfp_t gfp_mask)
{
   struct sk_buff *skb = alloc_skb(length + NET_SKB_PAD, gfp_mask);

   if ( likely( skb ) )
       skb_reserve(skb, NET_SKB_PAD);
   return skb;
}

struct sk_buff *dev_alloc_skb(unsigned int length)
{
   return __dev_alloc_skb(length,GFP_ATOMIC);
}
```

为 Socket Buffer 分配内存

返回 skb 之前，在前面预留 16 个字节

在中断中分配内存标志

（4）__netdev_alloc_skb

netdev_alloc_skb

```
struct sk_buff *__netdev_alloc_skb(struct net_device *dev,
        unsigned int length, gfp_t gfp_mask)
  {
   …
   struct sk_buff *skb;
   skb = __alloc_skb(length + NET_SKB_PAD, gfp_mask, 0, node);

   if (likely(skb)) {
         skb_reserve(skb, NET_SKB_PAD);
         skb->dev = dev;
   }
   return skb;
}
static inline struct sk_buff *netdev_alloc_skb(struct net_device *dev,
        unsigned int length)
{
      return __netdev_alloc_skb(dev, length, GFP_ATOMIC);
}
```

初始化接收数据的设备

设置在中断中分配内存的标志

__netdev_alloc_skb 与__dev_alloc_skb 函数类似，它为指定了某个接收数据包的网络设备分配 Socket Buffer 的内存空间。因此，在返回分配的 sk_buff 之前，初始化了 sk_buff 的设备指针*dev 域。同样 netdev_alloc_skb 包装函数也是在接收数据包的中断处理程序中调用的，要用 GFP_ATOMIC 选项来申请内存分配。

2．释放 Socket Buffer

- kfree_skb 函数用于释放 Socket Buffer 的内存。先前分配的 sk_buff 会返回到内存对象池（cache）中。对 Socket Buffer 内存的释放操作比常规内存对象释放操作更复杂，需要考虑以下情形：
- kfree_skb 只有在 sk_buff 的引用计数（skb->users）为 1 时才会完全释放 sk_buff 数据结构本身（即没有进程使用该数据结构了），否则该函数只做对 skb->users 数据域的减 1 操作。

例如，如果当前有 3 个进程在使用该 sk_buff，只有第 3 次调用 kfree_skb 的时候才会最后释放该 sk_buff。图 2-9 给出了释放 Socket Buffer 的所有步骤。

- sk_buff 数据结构中还包含指向其他数据结构的指针，如路由表、连接跟踪等，

所以它可能持有对其他数据结构的引用计数，比如对 struct dst_entry *dst, nfct 的引用计数。既然该 sk_buff 要释放，也要调用相应的函数对与其连接的数据结构的引用计数减 1。

- 如果 sk_buff 的 destructor 函数指针初始化了，在释放 sk_buff 时，需要调用该函数做清理工作。
- 紧接在 sk_buff 底部有另一个数据结构实体 skb_shared_info，该数据结构描述的是数据包被分片后的相关信息。skb_shared_info 内部的指针还可能指向别的数据片所在内存，这时 kfree_skb 也要释放数据片占用的内存以及管理数据片内存的队列。
- 最后 sk_buff 数据结构将放回到 sk_buff 的 cache 中。如果该 sk_buff 没有克隆(skb->fclone 为 SKB_FCLONE_UNAVAILABLE)，它将返回到 skbuff_head_cache 缓冲区中。如果该 sk_buff 有克隆，该 sk_buff 数据结构将返回到 skbuff_fclone_cache 缓冲区中。完成 Socket Buffer 释放的主要函数如表 2-6 所示。

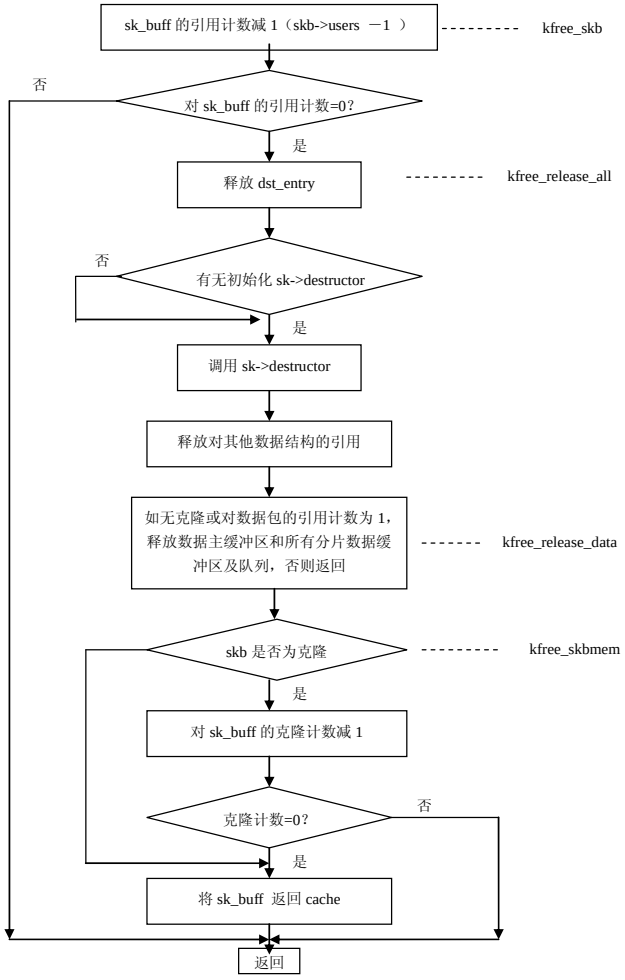


图 2-9　Socket Buffer 的释放操作

表 2-6　释放 Socket Buffer 的操作函数

函　数　名	函 数 功 能
kfree_skb	释放 Socket Buffer 的包装函数
kfree_release_all	释放与 sk_buff 相连的其他数据结构的引用计数
kfree_release_data	释放数据包的主缓冲区和数据片缓冲区
kfree_skbmem	将 sk_buff 数据结构返回内存对象池
dst_release	释放对路由表的引用

2.4.2　数据空间的预留和对齐

表 2-7 列出了在内核中实现的完成在 Socket Buffer 数据包存放缓冲区数据空间的预留和对齐操作的主要函数。这些函数提供的是操作 sk_buff 数据结构的 data、tail 指针的方法。

表 2-7　在 Socket Buffer 中预留空间和对齐操作函数

函数原型	函数功能
skb_reserve(struct sk_buff *skb, int len)	在 skb 的头和尾预留指定长度的空间
skb_put(struct sk_buff *skb,unsigned int len)	在 skb 的尾部预留指定长度的空间
skb_push(struct sk_buff *skb, unsigned int len)	在 skb 的头部预留指定长度的空间
skb_pull(struct sk_buff *skb,unsigned int len)	从 skb 的头部移走指定长度的数据
skb_trim(struct sk_buff *skb, unsigned int len)	从 skb 的尾部移走指定长度的数据

1．skb_reserve

在存放数据包的缓冲区的头部预留指定的空间，通常用于在数据包存储区插入网络协议头或强制数据边界对齐。该函数同时将 skb->data、skb->tail 指针向后移动。即:

skb->data += len;

skb->tail += len;

看一看以太网设备驱动程序的接收函数，你会看到，在将数据存入刚分配的 Socket Buffer 之前，都要使用以下命令：

```
文件目录 drivers/net/cs89x0.c 中的 net_rx 函数
static void net_rx(struct net_device *dev)
{
skb = dev_alloc_skb(length + 2 );
        …
        skb_reserver(skb,2);
      }
```

因为 cs89x0 是以太网类的网络适配器，它的驱动程序知道其接收的是以太网协议的网络数据包，以太网的协议头长度为 14 个字节，所以驱动程序在数据包缓冲区头部预留 2 个字节，这样网络层的协议头在插入时可以按照 16 字节的边界对齐。

2．skb_put

将 skb->tail 指针向后移动 len 的位置，在 Socket Buffer 存放数据包的缓冲区尾部为数据预留长度为 len 的空间。

3．skb_push

将 skb->data 指针向前移动 len 的位置，在 Socket Buffer 存放数据包的缓冲区的头部为将要写入的数据预留长度为 len 的空间。

4．skb_pull

从 Socket Buffer 存放数据包缓冲区的头部移走长度为 len 的数据，即 skb->data 指针向后移动 skb->data + len。

5．skb_trim

从缓冲区尾部移走长度为 len 的数据，即 skb->tail 指针向前移动 skb->tail – len。

以上各项操作的示意图如图 2-10 与图 2-11 所示。

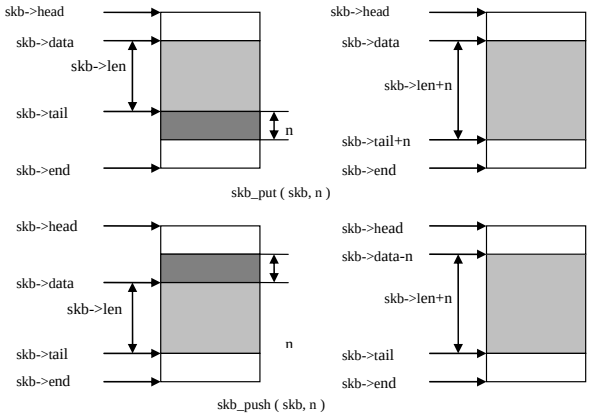
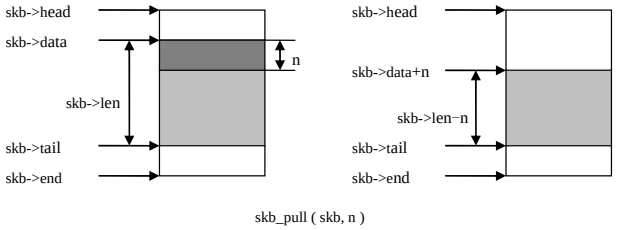


图 2-10　skb_put 与 skb_push 的操作示意图



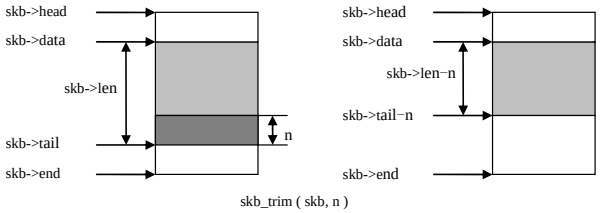


图 2-11　skb_pull 与 skb_trim 的操作示意图

skb_put 和 skb_push 并不做实际的数据复制，只是为将要复制到数据缓冲区的数据预留空间，实际的数据复制在相应的协议层由该层自己的函数来完成。

2.4.3　复制和克隆

从网络上接收的数据包有时有多个进程要对其进行处理，这些处理过程有时需要修改

数据包的内容，有时只需修改 sk_buff 数据结构的内容。这时我们需要对 Socket Buffer 进行克隆和复制。

1．Socket Buffer 的克隆

在什么时候需要对 Socket Buffer 进行克隆？某些时候，同一个 Socket Buffer 会由不同的进程独立处理，但这些进程所需要操作的只是 sk_buff 数据结构描述符，不需要对数据包本身做改动。这时为了提高处理性能，内核不需对整个 Socket Buffer（sk_buff 数据结构和数据包缓冲区）做完全的复制，只对 sk_buff 数据结构做完全的复制，并将数据包的引用计数（dataref）加 1，以此来防止在还有进程使用该 Socket Buffer 的数据包情况下，缓冲区被释放。这就是 sk_buff 克隆函数的功能。这时，原 sk_buff 和克隆的 sk_buff 共享同一个数据包。克隆操作过程如图 2-12 所示。

实现克隆操作的函数为：

```
struct sk_buff *skb_clone(struct sk_buff *skb, gfp_t gtp_mask)
```

skb_clone 产生一个对 skb 的克隆数据结构，返回指向克隆出来的 sk_buff 数据结构的指针。克隆的 sk_buff 都具有以下特点：

- 不放入任何 sk_buff 的管理队列。
- 不属于任何套接字，即没有任何套接字拥有克隆的 sk_buff。
- 两个 sk_buff 结构的 skb->cloned 域都需置为 1。克隆出来的 sk_buff 的 sk->users 域应设为 1，这样当要释放克隆出来的 sk_buff 数据结构时，第一次对它的释放就能成功了。
- 当一个 sk_buff 被克隆后，它的数据包中的值就不能再修改，这时我们访问数据包时可以不加锁，因为只能对数据包作读操作，也就不存在并发访问的问题。

与克隆相关的帮助函数 skb_cloned(skb)可用于查看某个 skb 是否为克隆的缓冲区。

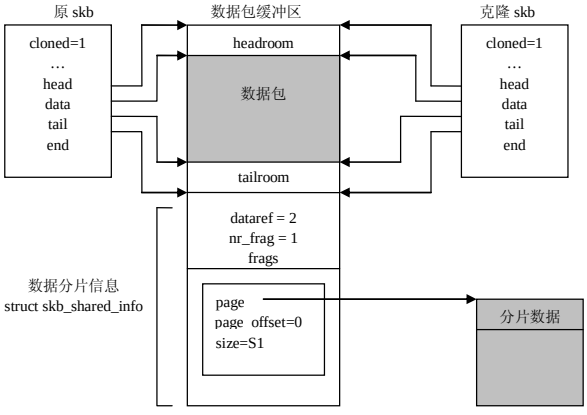


图 2-12　克隆 Socket Buffer

2．Socket Buffer 的复制

与克隆不同，当多个进程对同一个 Socket Buffer 的操作为既要修改 sk_buff 数据结构中

的内容，也要修改数据包内容时，必须对 Socket Buffer 进行复制。这时可以有两个选择：

- 如果既要修改主数据包中的内容，又要操作分片数据段中的值，函数 struct sk_buff *skb_copy(struct sk_buff *skb) 可以完成该功能。
- 如果只修改主数据包中的内容，而不需读/写分片数据段中的值，可以使用函数：struct sk_buff *pskb_copy(struct sk_buff *skb)来完成操作。

图 2-13 给出了这两个函数操作的差别。

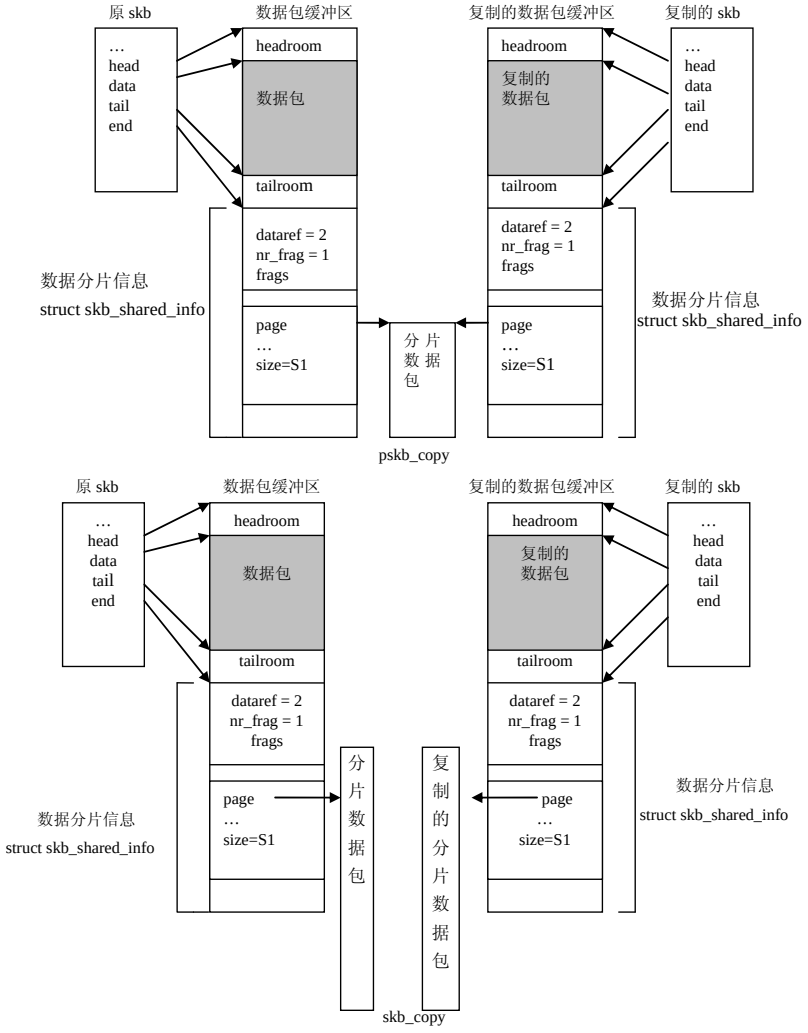


图 2-13　Socket Buffer 的复制和部分复制

2.4.4　操作队列的函数

Socket Buffer 在内核中组织在不同的队列里，相应的内核也提供了一系列的函数来管理队列操作，比如将 sk_buff 数据结构插入队列，从队列中取出 sk_buff 等。这些函数如表 2-8 所示。

表 2-8　内核中操作队列的函数

函 数 原 型	函 数 功 能
skb_queue_head_init	初始化 sk_buff 的等待队列，这时队列为一个空队列
skb_dequeue	从 list 队列的头读取一个 Socket Buffer，返回值为读取的 Socket Buffer 的地址
skb_dequeue_tail	从 list 队列的结尾读取一个 Socket Buffer
skb_queue_purge	清空 list 队列，依次从 list 队列中读取 Socket Buffer 并释放缓冲区，直到 list 队列为空
skb_queue_head	将一个新的 Socket Buffer 放入 list 队列的头
skb_queue_tail	将一个 Socket Buffer 放入队列尾
skb_unlink	从指定的队列中去掉一个 Socket Buffer
skb_append	将 newsk 放入队列指定的某个成员后
skb_insert	将 newsk 插入队列指定的成员前
skb_drop_list	释放一个队列中的所有 sk_buff，最后释放队列
skb_queue_walk	在队列中遍历每一个成员
skb_drop_fraglist	释放分片数据缓冲区和队列

这类函数的执行必须是原子操作，在操作队列前它们必须首先获取 sk_buff_head 结构中的 spinlock，否则，操作可能被异步事件（如中断操作）打断。

2.4.5　引用计数的操作

1．sk_buff 引用计数加 1

对 Socket Buffer 引用计数的操作由 skb_get 函数完成。

```
static inline struct sk_buff *skb_get( struct sk_buff *skb)
{
  atomic_inc(&skb->users);
  return skb;
}
```

2．分片数据引用计数加 1

```
static void skb_clone_fraglist(struct sk_buff *skb)
{
  struct sk_buff *list;
  for (list = skb_shinfo(skb)->frag_list; list; list = list->next)
        skb_get(list);
}
```

对数据包的引用计数不在 sk_buff 数据结构中，在 2.5 节将介绍数据包的引用计数在描述数据分片的数据结构变量中。

2.4.6　协议头指针操作

从 Linux-2.6.26 以后，sk_buff 中描述各层协议头信息在数据包中地址的数据域发生了重大的变化。原来的版本中，传输层、网络层、数据链路层的协议头在数据包中的起始地址是由联合（union）变量表示的。各协议层都可以有多个协议，例如在传输层最常用的就有 TCP 和 UDP。不同的协议头的长度是不一样的。在 Linux-2.6.26 以后将其改为指针（32

位体系结构）或头数据相对于数据包起始地址的偏移量（64 位体系结构）。在 include/linux/skbuff.h 中也增加了一系列的函数来操作这些指针，其函数原型及功能如表 2-9 所示。

表 2-9　各层协议头指针操作函数

传　输　层	函 数 原 型	函 数 功 能
	skb_transport_header	返回传输层协议头数据的地址
	skb_reset_transport_header	复位传输层协议头数据地址指针
	skb_set_transport_header	将传输层协议头信息地址指针设置到相应位置
网络层	skb_network_head	返回网络层协议头数据的地址
	skb_reset_network_heade	复位网络层协议头数据地址指针
	skb_set_network_header	将网络协议头信息地址指针设置到相应位置
数据链路层	skb_mac_heade	返回数据链路层协议头数据的地址
	skb_reset_mac_header	复位数据链路层协议头数据地址指针
	skb_set_mac_header	将数据链路层协议头信息地址指针设置到相应位置

2.5　数据分片和分段

如图 2-7 所示，在 sk_buff 数据结构后紧跟了另一数据结构 struct skb_shared_info。在为 Socket Buffer 分配内存时，我们也会同时为 skb_shared_info 分配相应大小的内存空间，并顺序地初始化该数据结构。struct skb_shared_info 是用于支持 IP 数据分片（fragmentation）和 TCP 数据分段（segmentation）的数据结构。

2.5.1　为什么要分割数据包

当一个数据包的长度大于网络适配器硬件能传送的最大传送单元 MTU 时，需要对数据包进行分割，将其分成更小的数据片，这些数据片存放在由 struct skb_shared_info 数据结构管理的 Socket Buffer 组成的单向链表中。管理数据片的 struct skb_shared_info 数据结构的定义为：

```
struct skb_shared_info
{
  atomic_t         dataref;
  unsigned short   nr_frags;
  unsigned short   gso_size;
  unsigned short   gso_segs;
  unsigned short   gso_type;
  __be32                ip6_frag_id;
  struct sk_buff   *frag_list;
  skb_frag_t       frags[MAX_SKB_FRAGS];
};
```

其中各数据域的含义如下。

- Dataref：描述了对主数据包缓冲区的引用计数，例如当每做一次 skb_buff 克隆时，该计数加 1。
- nf_frags：是这个数据包被分割的数据片的计数，描述了一个数据包最终被分成

了多少个数据片。这个域是支持 IP 分片使用的。

- frag_list：如果数据包被分成片段，该域是指向存放分片数据链表起始地址的指针。
- frags：这是一个页表入口数组，每一个入口就是一个 TCP 的段。
- gso_size、gso_type：这两个域用于说明网络设备是否具有在硬件上实现对 TCP 分段的能力，是网络设备的功能特点 NET_F_TSO（我们在介绍网络设备数据结构时会介绍）描述的。
- gso_size：给出 TCP 数据包被分段的数量。

目前，有的网卡在硬件上可以完成对 TCP 层数据的分段（segment），这种功能原来是由 CPU 来完成的，通过将对大的数据包（如 64KB）的分割任务交由网络适配器硬件完成，可以减轻 CPU 的负载，从而提高网络的速度。这种技术称为 TSO（TCP Segmentation Offload），或 GSO（Generic Segmentation Offload）。gso_size、gso_type 和 gso_segs 就是用于这类技术的。GSO 可以用于各种协议。

2.5.2 设计 skb_shared_info 数据结构的目的

struct skb_shared_info 数据结构紧接在数据包缓冲区之后，由 sk_buff 的 end 指针来寻址。使用 struct skb_shared_info 数据结构有几个目的：支持 IP 数据分片；支持 TCP 数据分段；跟踪数据包的引用计数。

当用于 IP 数据分片时，struct skb_shared_info 数据结构中有指针指向包含 IP 数据片的 Socket Buffer 的链表。当用于 TCP 数据分段时，该结构中包含了一个相关的页面数据，其中存放了分段的数据。

frags 数组中的每个元素都是一个管理存放 TCP 数据段的 skb_frag_t 的结构，frags 数组的大小的总和为 64KB。

```
#define       MAX_SKB_FRAGS( 65536/PAGE_SIZE + 2 )
typedef struct skb_frag_struct skb_frag_t;
struct skb_frag_struct{
   struct page *page;            //指向存放 TCP 数据段存放的页面
   __u16       page_offset;      //TCP 数据段在页面中相对于页面的偏移量
   __u16       size;             //段的大小
}
```

2.5.3 操作 skb_shared_info 的函数

操作 skb_shared_info 的函数有如下几种。

- skb_is_nonlinear：查看 Socket Buffer 的数据包是否被分了片段
- skb_linearize：将分了片的小数据包组装成一个完整的大数据包。
- skb_shinfo：在 sk_buff 结构中，并没有指针指向 struct skb_shared_info 数据结构，要访问该数据结构，需要用到宏，skb_sh_info 返回 sk_buff 的 end 指针。

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

例如，以下的语句显示了如何对该数据结构中的 dataref 域做加 1 操作。

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

2.6　本章总结

Socket Buffer 数据结构设计的高效、合理主要体现在：

- 管理数据结构 sk_buff 与数据包缓冲区分为两个实体，可使操作灵活，减少数据复制，传送处理高效。
- 内核实现了统一访问 Socket Buffer 数据结构的函数，使 Socket Buffer 的实现独立于网络协议与网络硬件设备。系统可扩展性强，可随意为内核扩展新的网络协议实现。
- 数据包分割后的分片数据与主数据包的 Socket Buffer 由不同的数据结构管理，但又密切相关。在需要的时候可任意访问网络数据、管理数据、分片数据的不同部分，互相不受影响。
- 以内核标准的队列头数据结构管理 Socket Buffer 队列，可应用内核中标准的队列操作函数，一方面避免了重复开发，代码重利用率高；另一方面以队列的方式管理 Socket Buffer，数据在 TCP/IP 协议栈中传递简单、高效。

第 3 章　网络设备在内核中的抽象——struct net_device 数据结构

本章主要介绍 Linux 内核网络体系结构中描述网络设备在内核中的表现实体 struct net_device 数据结构。在详细描述 struct net_device 数据结构数据域的设计和各域含义的基础上，对 net_device 的功能作了分类总结。最后给出在网络设备驱动程序中实现的常用功能函数与 struct net_device 数据结构中函数指针的关联。

网络设备是最终将数据发送到网络上的终点设备，也是从网络上接收数据包的源头；网络设备是内核、应用程序与网络之间的接口。目前网络设备硬件种类繁多，实现的传输协议，使用的数据传送技术也各有不同，网络设备硬件的技术也在不断更新。Linux 内核在实现对各种网络设备硬件的广泛支持的同时，又要屏蔽网络设备硬件与驱动程序的实现细节，不至于因设备的更换而修改内核上层协议代码的实现以及数据传送方法。为此在 Linux 网络体系结构中将网络设备抽象为 struct net_device 数据结构，struct net_device 数据结构的数据域可以根据不同的网络设备初始化为不同的属性。struct net_device 结构中的函数指针，如图 3-1 所示，dev->open、dev->hard_start_xmit 等可以依照使用的网络设备的不同，而初始化为不同的设备驱动程序的函数实例。这样 struct net_device 数据结构就成为网络设备硬件与上层协议之间的接口，无论网络设备硬件如何更换，都不需要修改网络体系结构上层协议代码的实现。

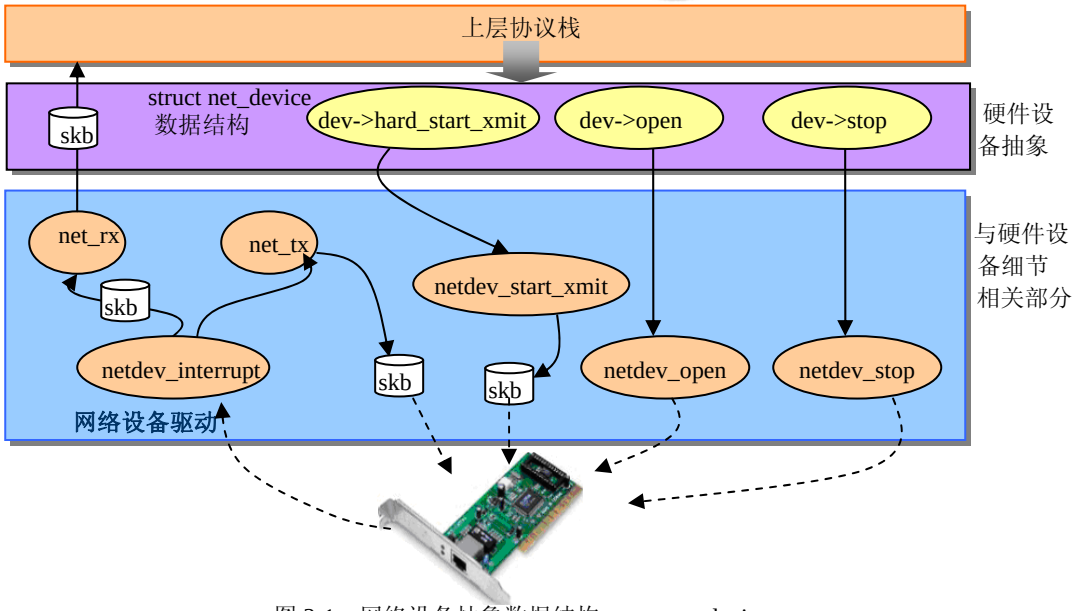


图 3-1　网络设备抽象数据结构 struct net_device

本章我们将介绍 Linux 内核中描述网络设备的关键数据结构 struct net_device，以及操作 struct net_device 数据结构相关域的常用函数，在此基础上讨论网络设备驱动程序的实现。

3.1　协议栈与网络设备

任何网络数据的传送都需要通过一定的物理介质，网络适配器访问传输数据的物理介质，将数据通过物理介质传送出去或从物理介质上接收进来。网络传输介质与网络适配器一起构成了数据传输的桥梁，使数据可以在两个或更多的系统之间交换。网络适配器的功能涵盖了 ISO/OSI 七层协议模型（参见图 2-1 左端）的第一层（物理层）和第二层（数据链路层）的一部分，为硬件实现部分。上层协议栈（如 TCP/IP 协议栈）的软件，负责实现操作系统内核使用的各种网络协议。

3.1.1　协议栈软件与网络设备硬件之间的接口

在网络适配器硬件和软件协议栈之间需要一个接口，共同完成操作系统内核中协议栈数据处理与异步收发的功能。在 Linux 网络体系结构中，这个接口要满足以下要求：

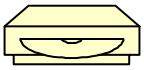
1．抽象出网络适配器的硬件特性

不同生产厂商的网络适配器在物理层和数据链路层的实现可能会有很大不同，这样每个网络设备都需要一个软件与硬件通信，这就是网络适配器驱动程序。

2．为协议栈提供统一的调用接口

在 Linux 的网络体系结构中，可以有多个协议栈，每个协议栈的各层也可以有多个协议实例使用网络适配器提供的服务；各协议实例的代码实现应独立于网络设备的硬件特性，也即无论系统使用什么类型、什么生产厂家的网络适配器，上层协议的代码实现都不需作任何变化，就可以使用网络适配器的服务，这意味着网络设备与上层协议栈之间需要一个统一的接口。

以上两个要求在 Linux 内核的网络体系结构中分别由两个软件和一个主要的数据结构 struct net_device 实现，将图 3-1 进一步细化，协议栈软件与网络设备之间的关系如图 3-2 所示，分为 3 个层次。



文件路径

- dev.c 文件在 Linux 源文件的目录树：net/core/目录下。
- net_device 数据结构的完整定义包含在 Linux 源文件的目录树：include/linux/netdevice.h 文件中。

- 网络设备驱动程序。
- struct net_device 数据结构。
- dev.c 文件中的接口函数。

本章讨论的数据结构与代码实现集中在 Linux 源码目录结构的以下文件中：

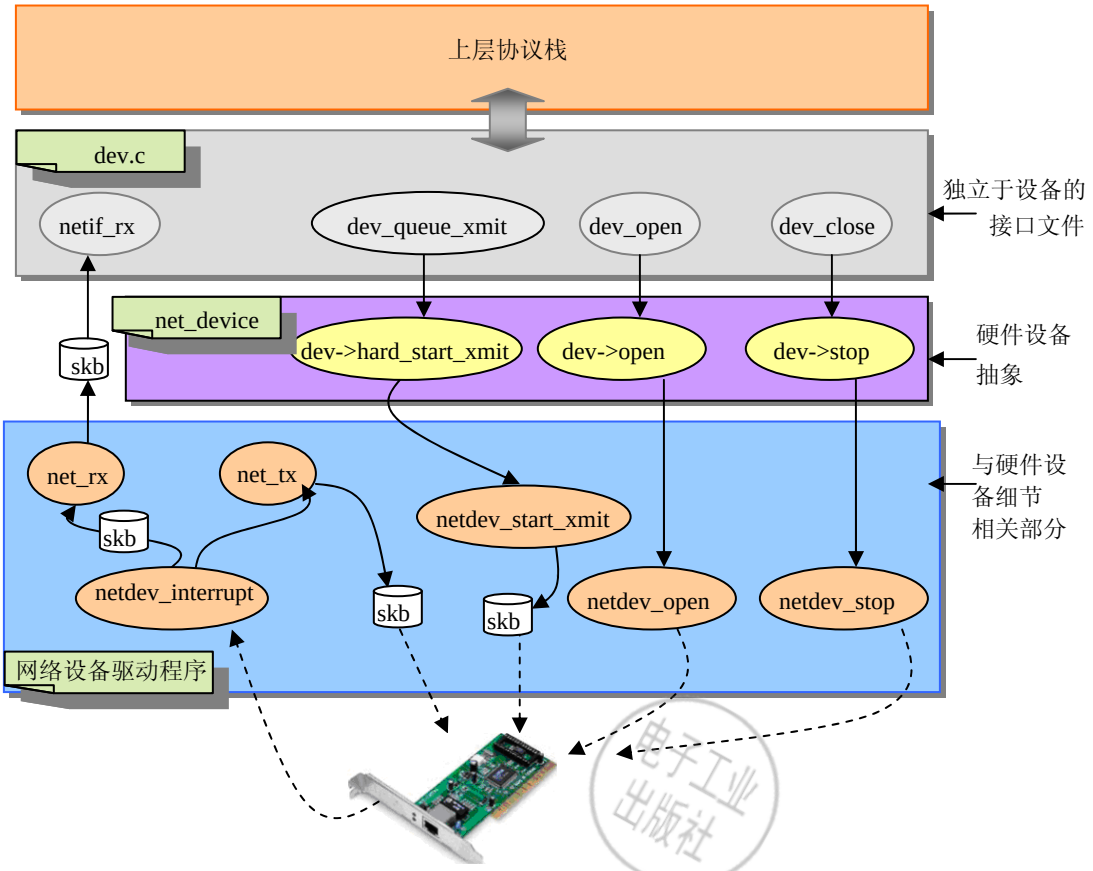


图 3-2　网络设备抽象数据结构 struct net_device

3.1.2　设备独立接口文件 dev.c

dev.c 文件中实现了对上层协议的统一调用接口，dev.c 文件中的函数实现了以下主要功能。

1．协议调用与驱动程序函数对应

dev.c 文件中的函数查看数据包由哪个网络设备（由 struct sk_buff 结构中*dev 数据域指明该数据包由哪个网络设备 net_device 实例接收/发送）传送，根据系统中注册的设备实例，调用网络设备驱动程序函数，实现硬件的收发。

2．对 struct net_device 数据结构的数据域统一初始化

struct net_device 数据结构中某些数据域，例如网络设备类型 dev->type（网络设备类型有 Ethernet、Token、Ring 等），只要是同一类型的网卡，不管它是由什么厂家生产，dev->type 的值都相同。struct net_device 数据结构的某些数据域，对每个网络设备（即便是同一类型的）也需设成不同的值。dev.c 提供了一些常规函数，来初始化 struct net_device 结构中的这样一些数据域：它们的值对所有类型的设备都一样，驱动程序可以调用这些函数来设置其设备实例的默认值，也可以重写由内核初始化的值。

3.1.3　设备驱动程序

由以上的描述可以看到，每一个网络设备都必须有一个驱动程序，并提供一个初始化函数供内核启动时调用，或在装载网络驱动程序模块时调用。不管网络设备内部有什么不同，有一件事是所有网络设备驱动程序必须首先完成的任务：初始化一个 struct net_device 数据结构的实例作为网络设备在内核中的实体，并将 struct net_device 数据结构实例的各数据域初始化为可工作的状态，然后将设备实例注册到内核中，为协议栈提供传送服务。

3.1.4　struct net_device 数据结构

struct net_device 数据结构从以下两个方面描述了网络设备的硬件特性在内核中的表示。

1．描述设备属性

struct net_device 数据结构实例是网络设备在内核中的表示，它是每个网络设备在内核中的基础数据结构，它包含的信息不仅仅是网络设备的硬件属性（中断、端口地址、驱动程序函数等），还包括网络中与设备有关的上层协议栈的配置信息（如 IP 地址、子网掩码等）。它跟踪连接到 TCP/IP 协议栈上的所有设备的状态信息。

2．实现设备驱动程序接口

struct net_device 数据结构代表了上层的网络协议和硬件之间的一个通用接口，使我们可以将网络协议层的实现从具体的网络硬件部件中抽象出来，独立于硬件设备。为了有效地实现这种抽象，struct net_device 中使用了大量函数指针，这样相对于上层的协议栈，它们在做数据收发操作时调用的函数的名字是相同的，但具体的函数实现细节可以根据不同的网络适配器而不同，由设备驱动程序提供，对网络协议栈透明。

例如当一个数据包已由协议栈处理完毕，准备从网络层（L3）传给数据链路层（L2）的硬件向外发送时，无论当前系统中使用的是什么网络设备，网络层的协议实例都调用 hard_start_xmit 函数指针所指定的功能函数来发送数据包；如果目前使用的是 3Com 的 3c509 网络适配器，hard_start_xmit 函数指针调用的就是 el3_start_xmit，3c509 的网络设备驱动程序的传送函数；如果这时我们使用的是 CS8900 网络适配器，hard_start_xmit 函数指针调用的就是 CS8900 网络设备驱动程序的传送函数：net_send_packet，但对上层的协议栈都一样，它要传送数据包时都只需调用 hard_start_xmit。

struct net_device 数据结构的数据域根据其功能，可以划分为以下各类。

- 配置（configuration）。
- 统计（statistics）。
- 设备状态（device status）。
- 链表管理（list management）。
- 流量管理（traffic management）。
- 常规域（generic）。
- 函数指针（function pointer）。

struct net_device 数据结构包含的信息非常多，所以我们不按照数据域的功能分类来描

述数据域的含义，相关功能域有时在 struct net_device 数据结构中并不在邻近位置，为了便于查看，我们按照数据域在 struct net_device 数据结构中的排放顺序来描述。

3.2 struct net_device 数据结构

struct net_device 数据结构应用最多的场合是网络设备驱动程序的开发，所以在着手网络设备驱动程序的开发之前，首先需要了解 struct net_device 数据结构中各数据域的设置，为什么要设置这些数据域，各数据域描述了网络设备的什么硬件特性，以及这些特性在系统运行过程中会如何变化？除了这些之外，上层 TCP/IP 协议栈需要获取网络设备的哪些物理属性和运行状态才能正确处理和维护数据的发送和接收。在这一节我们首先会对 struct net_device 数据结构的数据域的含义和设置目的做较详细的分析，再归纳支持同一功能的数据域。图 3-3 描述了网络设备硬件实例化，从注册到内核的过程。

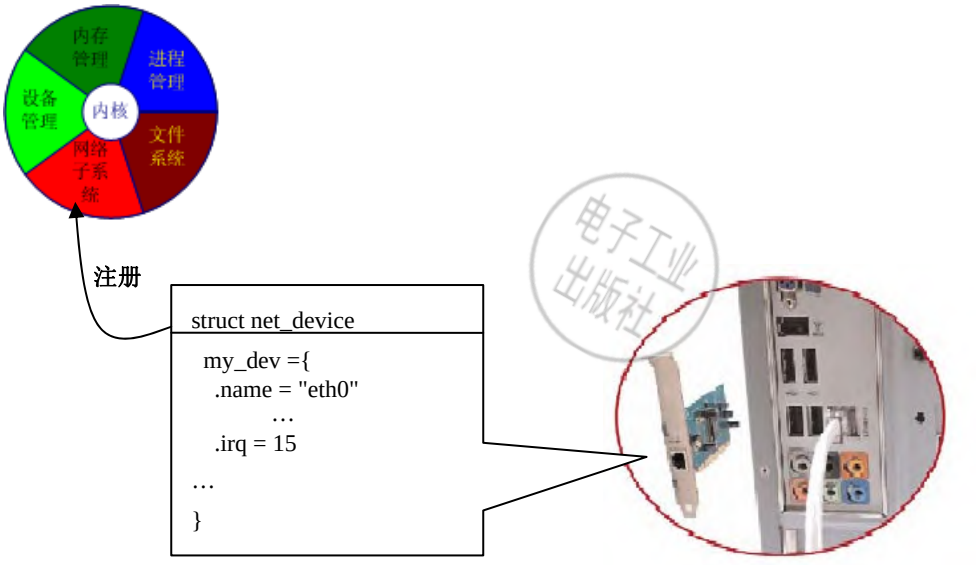


图 3-3 网络设备硬件、struct net_device 与内核的关系

3.2.1 struct net_device 数据结构的数据域

struct net_device 数据结构的数据域第一部分描述如下：

1．name[IFNAMSIZ]

网络设备名，系统中每个网络设备都有一个唯一的设备名。一方面，在 Linux 内核代码中网络设备名是内核查找设备的关键字；另一方面，在用户地址空间，设备名反映了当前系统中有哪些网络设备。

网络设备名的取名规则为几个字母后跟一个数字如 abc0、abc1 等。前面的字母描述了网络设备的硬件类型，后面的数字描述了系统中有多少个这类的设备。例如，对于以太网

类型设备，网络设备名为：eth0、eth1……，依次类推。loopback 网络设备，其设备名依次为 lo0、lo1 等。

在向 Linux 系统注册一个网络设备时，可以向系统传送一个设备名，也可以让系统自动为设备分配设备名。

2．name_hlist

以网络设备名为关键字建立的哈希链表，支持对网络设备的快速搜索。

3．*ifalias

网络设备的别名，主要用于简单邮件传输协议（SMTP）访问网络设备的别名。

4．mem_end

mem_start

设备和内核之间通信的共享存储区，只在网络设备驱动程序中初始化和访问这段区域，上层协议不需关心。这段缓冲区的大小指明了网卡上的有效缓冲存储区，当用 ifconfig 来初始化网络适配器时，可以指定这段共享存储区在内存中的位置。

5．base_addr

设备上的存储区映射到内存地址范围的 I/O 起始地址。

6．irq

表示网络设备使用的中断号，中断号也可以由多个设备共享。设备驱动程序在向内核注册自己的设备时可以调用 request_irq 函数申请所需的中断号，设备卸载时用 free_irq 函数来释放设备占用的中断。

7．if_port

当前网络设备连接的传输介质可以为：同轴电缆、双绞线等；最常用连接头是屏蔽双绞线 RJ45 头。有的设备不止一个连接器，这样用户可以根据需要选择一个。该参数用于设置设备的接口类型，如果设备驱动程序没有强行规定一定要选择什么类型的接口，就用默认值。也有一个设备驱动程序处理多个不同接口的情况，这时，只需简单地测试所有接口的类型就可以确定当前接口的种类。以下的代码片段展示了设备驱动程序如何根据配置来确定设备的接口类型。

```
    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;
}
```

Linux 支持的全部接口类型定义也在 include/linux/netdevice.h 头文件中：

```
enum{
  IF_PORT_UNKNOWN = 0,
  IF_PORT_10BASE2,
  IF_PORT_10BASET,
  IF_PORT_AUI,
```

```
    IF_PORT_100BASET,
    IF_PROT_100BASETX,
    IF_PROT_100BASEFX
}
```

8．dma

表示设备使用的 DMA（Direct Memory Access，直接存储器访问）通道号，如果设备支持 DMA 传送方式的话。Linux 内核实现了 API，让设备可以从内核中获取或释放 DMA 通道，它们分别是 request_dma 和 free_dma。在获取 DMA 通道后，可以使用函数 enable_dma 和 disable_dma 来允许或禁止使用 DMA 传送方式。根据 CPU 体系结构不同，request_dma 和 free_dma 这两个函数的实现也不同，也即这两个函数是依赖于 CPU 体系结构的。所以它们的原型包含在 include/asm-architecture 文件中（如 include/asm-i386.h），这些例程主要由 ISA 设备使用，PCI 设备不需要。DMA 方式不是所有的设备都可以使用的，要看总线是否支持。

9．state

表示网络设备状态。这个数据域包括几个标志位，驱动程序通常不直接操作这些标志位，Linux 内核提供了一系列实用的函数来设置和清除这些标志。这组状态标志由网络中收发数据包队列使用，其值由 enum 数据类型的 netdev_state_t 常量来索引。netdev_state_t 也定义在 include/linux/netdevice.h 头文件中。

对于独立的标志位设置和清除操作可以使用常规的函数：set_bit 和 clear_bit。但一般是通过包装函数来设置和清除这些位，这样可以隐藏各标志位的使用细节。这些设备状态标志位的含义如表 3-1 所示。


```
enum netdev_state_t
{
  __LINK_STATE_START,
  __LINK_STATE_PRESENT,
  __LINK_STATE_NOCARRIER,
  __LINK_STATE_LINKWATCH_PENDING,
  __LINK_STATE_DORMANT,
};
```

表 3-1　网络设备状态值

netdev_state_t 状态值	含　义
__LINK_STATE_START	网络设备是否打开，即使网络设备打开并不表示可以传送数据包，这时网卡上的所有缓冲区可能都被占用了
__LINK_STATE_PRESENT	网络设备已连接到传输介质
__LINK_STATE_NOCARRIE R	网络设备没有连接到传输介质
__LINK_STATE_LINKWATCH_PENDING	查看网络设备连接传输介质挂起
__LINK_STATE_DORMANT	网络设备暂停

例如，当我们要停止网络设备的收发队列时，子系统就调用 netif_tx_stop_queue 函数，此函数为 set_bit 的包装函数。

```
  static inline void netif_stop_queue( struct net_device *dev)
  {
        netif_tx_stop_queue(netdev_get_tx_queue(dev, 0));
  }
  static inline int netif_tx_stop_queue(struct netdev_queue *dev_queue)
  {
        set_bit(__QUEUE_STATE_XOFF, &dev_queue->state);
}
```

10．dev_list

网络设备实例组成的链表，系统中所有网络设备的 struct net_device 数据结构变量实例都会放入这个全局链表中，其基地址存放在 dev_base_head 指针变量中。

11．napi_list

支持 NAPI（New Access Protocol Interface）功能的网络设备组成的链表。我们将在第 6 章讲解数据包在数据链路层的接收过程时介绍 NAPI。

12．features

表示网络设备的硬件功能特点。features 的值由网络设备驱动程序设置，用于告诉内核网络设备具有的硬件特性。struct net_device 数据结构中有多个数据域来描述设备功能，每个数据域描述的是网络设备在不同方面的功能特性，features 描述了设备和 CPU 之间通信的能力，例如设备是否可以在存储区高端做 DMA 操作，网络设备硬件是否可以对所有的数据包做校验和检查等。对 features 值的定义就紧跟在 struct net_device 数据结构中 features 数据域定义之后，包含在 struct net_device 数据结构内。

features 值的定义形式为：#define NETIF_F_XXX，部分值的含义如表 3-2 所示。

表 3-2　struct net_device 数据结构中 features 数据域的常量值定义

符　　号	值	含　　义
NETIF_F_SG	1	这两个标志都用于表明设备硬件是否具有 scatter/gather 输入/输出能力，如果你的网络设备可以传输被分割成几个存储数据段的数据包，应该设置 NETIF_F_SG。NETIF_F_FRAGLIST 表明网络接口可以处理被分段的数据列表
NETIF_F_FRAGLIST	64	
NETIF_F_IP_CSUM	2	硬件可以为基于 IPv4 协议的 TCP/UDP 传送数据包做校验和，设置该标志
NETIF_F_HW_CSUM	8	可由硬件完成对所有数据包的校验和。设置这个标志位
NETIF_F_NO_CSUM	4	不需做校验和，比如 loopback 设备
NETIF_F_HIGHDMA	32	如果接口可以在存储器高端做 DMA 操作

13．ifindex

iflink

ifindex 是唯一标识网络设备的索引号。当向内核注册网络设备时调用 dev_new_index 函数为每个设备分配设备索引号。iflink 标识符由虚拟设备使用，标识在虚拟设备中实际完成数据传送的网络设备。

14．stats

表示网络设备工作的状态统计信息。数据结构 struct net_device_stats 也定义在 include/linux/netdevice.h 头文件中，它包含的是网络设备工作过程中的常规统计信息，其中一些统计信息如表 3-3 所示。

表 3-3　网络设备工作过程中的统计信息

名　　称	含　　义
rx_packets	网络设备从网络接收到的总数据包数
tx_packets	网络设备发送的总数据包数
rx_bytes	网络设备从网络收到的总字节数
tx_bytes	网络设备发送的总字包数

15．*wireless_handlers

*wireless_data

*wireless_handlers 是函数指针，指向一组函数，这组函数是无线网络设备用以处理设备扩展功能的。在 include/net/iw_handler.h 头文件中有这组函数的详细信息，使用这些功能的另一种方法是通过 ioctl 系统调用。*wireless_data 是无线网络处理函数使用数据。

16．*netdev_ops

在 struct net_device_ops 数据结构中定义了网络设备驱动程序实现的功能函数。struct net_device_ops 数据结构的定义也包含在 include/linux/netdevice.h 头文件中。

17．*ethtool_ops

ethtool 是为系统管理员提供的用于控制网络接口的工具函数，使用 ethtool 能够控制包括速度、介质类型、双工操作、DMA 设置、硬件校验、LAN 唤醒等功能，前提是在网络驱动程序中实现了 ethtool 工具的接口。struct ethtool_ops 结构中包含了一组函数指针，这组函数是用于修改和报告网络设备设置的函数，其详细的定义在 include/linux/ethtool.h 头文件中，如下代码为其中的片段：

```
struct ethtool_ops {
  int   (*get_settings)(struct net_device *, struct ethtool_cmd *);
  int   (*set_settings)( struct net_device *, struct ethtool_cmd *);
  void  (*get_drvinfo)(struct net_device *, struct ethtool_drvinfo *);
  int   (*get_regs_len)(struct net_device *);
…
};
```

18．*get_settings

用于获取网络设备的设置，并将网络设备的设置返回到 struct ethtool_cmd 指向的变量中。

19．*set_settings

将网络设备支持的所有设置放在 struct ethtool_cmd 指向的变量中传给网络设备，以改变网络设备的设置。

到这里，struct net_device 数据结构中可见部分就讲解完了。后面几节介绍的所有数据域在内部使用，可能在内核以后的版本中会做修改，所以设备驱动程序的实现最好不要依赖于以下的数据域。

3.2.2 struct net_device 数据结构的其他数据域

1．header_ops

struct header_ops 数据结构中包含了一组函数指针，指向操作数据链路层协议头的功能函数组，如协议头的创建、解析、重建协议头等。struct header_ops 数据结构也定义在 include/linux/netdevice.h 头文件中。

```
struct header_ops {
  int   (*create) (struct sk_buff *skb, struct net_device *dev,
                   unsigned short type, const void *daddr,
                   const void *saddr, unsigned len);
  int   (*parse)(const struct sk_buff *skb, unsigned char *haddr);
  int   (*rebuild)(struct sk_buff *skb);
#define HAVE_HEADER_CACHE
  int   (*cache)(const struct neighbour *neigh, struct hh_cache *hh);
```

```
    void  (*cache_update)(struct hh_cache *hh,
                          const struct net_device *dev,
                          const unsigned char *haddr);
};
```

2．flags

gflags

priv_flags

flags 中的某些位表示网络设备的工作模式，如 IFF_MULTICAST，表示设备设置为组传送工作模式。其他的位表示接口状态的改变，如 IFF_UP 或 IFF_RUNNING。可以在 include/linux/if.h 文件中看到其值的完全定义。驱动程序通常在初始化阶段设置设备的工作模式数据域，而状态位由 Linux 内核根据外部发生的事件来管理，其设置可以用 ifconfig 来查看。

struct net_device 数据结构中 flags 部分位的含义如表 3-4 所示。

表 3-4　struct net_device 数据结构中 flags 部分位的含义

位	描　　述
IFF_UP	网络设备激活了，可以接收和发送数据包
IFF_BROADCAST	该设备支持广播工作模式，广播地址有效
IFF_DEBUG	调试标志打开
IFF_LOOPBACK	表明该网络设备是 LOOPBACK 设备
IFF_POINTOPOINT	该设备是点到点的连接
IFF_RUNNING	设备正在运行相关操作
IFF_NOARP	设备不支持 ARP 协议
IFF_PROMISC	该标志使设备工作于混杂模式（Promiscuous），接收所有的数据包
IFF_ALLMULTI	接收所有的组传送（Multicast）数据包
IFF_MULTICAST	该设备支持组传送，组传送地址有效

priv_flags 存放的是对用户空间不可见的标志，目前这个数据域主要是由 VLAN 和桥虚拟设备使用，例如，IFF_802.1Q_VLAN 表示 802.1Q 的 VLAN 设备，IFF_EBRIDGE 表示以太网的桥设备。

gflags 几乎没有使用过，该数据域的存在是为了兼容。

对 flags 的修改可以使用函数 dev_change_flags 来完成。

3．padded

该数据域表示用 alloc_netdev 函数为网络设备分配内存空间时，应在最后补多少个零。

4．operstate

表示网络的操作状态。在 Linux 中区分了接口的管理状态和设备操作状态，管理状态是指接口是否呈现，反映的是系统管理员是否将使用该设备来交换数据；而设备操作状态给出的是接口传送数据的能力。操作状态分成两个部分，其一是只能由设备驱动程序设置的两个标志；其二是由驱动程序设置的标志驱动的 RFC2863 兼容的状态、策略。operstate 中包含的就是 RFC2863 兼容的状态，反映的是访问管理信息的能力，其值如下：

- IF_OPER_UNKNOWN:接口处于不可知状态，无论驱动程序层或用户空间都没有设置操作状态。
- IF_OPER_NOPRESENT：在当前内核不可用。
- IF_OPER_DOWN：接口在 L1 层不能传送数据，例如在以太网中接口没有插入或

管理员卸载了接口等。

5．link_mode

映射到 RFC2863 兼容状态的策略。内核中的网络设备驱动程序操作的两个标志分别映射到 IFF_LOWER_UP 和 IFF_DORMANT。这些标志可以从任何地方设置，甚至可以从中断中设置，但保证只有设备驱动程序具备写操作的权限。

- __LINK_STATE_NOCARRIER：映射到 !IFF_LOWER_UP。
- __LINK_STATE_DORMANT：映射到 IFF_DORMANT。

6．mtu

表示网络接口的最大传送单元（Maximum Transmission Unit），它代表了网络设备一次可以处理的最大字节数，表 3-5 给出了最常使用的网络数据链路层技术上的 MTU 值。

表 3-5　常用网络链路层技术上的最大传送单元

设 备 类 型	MTU
PPP	296
SLIP	296
Ethernet	1500
SISDN	1500
PLIP	1500
Wavelan	1500
EhterChannel	2024
FDDI	4352
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

对于以太网的 MTU 需要说明的是，以太网数据帧规范中定义的最大数据负载是 1500 字节，有时你会看到以太网的 MTU 定义的是 1518 或 1514。第一个长度是在以太网的数据帧中包含了协议头和校验和，第二个长度是包含了协议头但不包含校验和（占 4 个字节）。

7．type

网络设备的类型（如以太网、帧中继网络设备等）。在 include/linux/if_arp.h 文件中定义了所有 Linux 内核支持的网络设备类型。表 3-6 为部分在 include/linux/if_arp.h 中定义的网络设备类型的常量。

表 3-6　网络设备类型的常量

符　　号	值	网络设备类型
ARPHRD_NETROM	0	NET/ROM 虚拟
ARPHRD_ETHER	1	10Mbps 以太网卡
ARPHRD_EETHER	2	实验以太网
ARPHRD_AX25	3	AX .25 Level2
ARPHRD_PRONET	4	PROnet 令牌环
ARPHRD_CHAOS	5	Chaosnet 混沌网络协议
ARPHRD_IEEE802	6	IEEE 802.2 以太网/TR/TB
ARPHRD_ARCNET	7	ARCnet

（续表）

符　　号	值	网络设备类型
ARPHRD_APPLETLK	8	APPLEtalk
ARPHRD_DLCI	15	帧中继 DLCI
ARPHRD_ATM	19	ATM

8．hard_header_len

表示某种类型的网络设备可传送的数据帧头信息长度，例如以太网类的设备数据帧头信息的长度为 14 个字节。各类网络设备的数据帧头信息的长度定义在该类设备的头文件中；以太网的数据帧头信息长度定义在 include/linux/if_ether.h 头文件中。例如：

```
#define ETH_HLEN   14
```

9．needed_headroom

needed_tailroom

表示硬件填写信息时可能需要在 Socket Buffer 中预留的额外 headroom 和 tailroom 的空间。

10．*master

有的协议允许一系列的网络设备组成一个组，当成一个设备使用。组中有一个设备需设置成主设备，起特殊的作用。master 数据域就是指向组中的主设备的 struct net_device 数据结构实例的指针。如果网络设备不属于任何组，master 设为 NULL。

11．perm_addr[MAX_ADDR_LEN]

addr_len

表示网络设备的硬件地址（即 MAC 地址），其长度由 addr_len 确定，不同类型的网络设备硬件地址长度不一样。

12．dev_id

用于共享网卡设备号。目前是由 IPv6 的 zSeries OSA 网络控制器使用。在 Linux 上的 zSeries OSA 网络适配器可以在各种 Linux 上共享，OSA 网卡只有一个 MAC 地址，如多个 Linux 内核使用同一个网卡时，会导致多地址冲突。但网卡的设备驱动程序可传递一个 16 位的标识符，使每个 Linux 主机共享一个网卡。

13．addr_list_lock

表示访问地址列表时防止并发访问控制的锁，用于防止对主机网络地址（unicast）列表和组地址（multicast）列表的并发访问。

14．*uc_list

uc_count

网络设备的主机网络地址列表。uc_list 列表用于支持在同一个网络设备上配置多个主机网络地址。uc_count 为网络设备配置的主机网络地址（unicast）的个数。也即 uc_list 列表中成员的个数。

15．uc_promisc

工作于混杂模式（promiscuity）的主机网络地址的个数。

16．*mc_list

mc_count

mc_list 表示网络设备配置的组传送 MAC 地址列表，即该网络设备的地址属于哪些组传送地址。mc_count 为网络设备配置的组传送（multicast）地址的个数，也就是 mc_list 链表中的成员数量。

17．promiscuity

表示混杂模式的计数器。某些网络管理任务要求网络设备接收所有经过网络设备的数据包，而不仅仅是目的地址与该网络设备匹配的数据包，这种工作模式称为混杂模式。

这种模式可以用于在局域网段上检查网络性能或安全性，它也用于桥接器代码中。struct net_device 数据结构中的计数器 promiscuity，用于指明当前设备是否处于混杂模式。之所以使用计数器而不是用一个标志变量来表明混杂模式，是因为某一时刻可以有多个任务（task）需要使用混杂模式，每个任务在进入该模式时，对 promiscuity 计数器加 1，在退出混杂模式时对计数器减 1，直到计数器的值为 0 时，设备才退出混杂模式，我们可以调用函数 dev_set_promiscuity 函数来操作 promiscuity 数据域。

无论什么时候当 promiscuity 的值为非零值时，struct net_device 数据结构中 flags 域的 IFF_PROMISC 标志位都应置位，查看接口配置的函数需要使用 flags 的位值。

以下的代码片段，来自 drivers/net/3c59x.c 设备驱动程序，此段代码的功能是根据 flags 的各标志位的值确定当前网络设备的接收工作模式。

```
static void set_rx_mode( struct net_device *dev)
{
  int ioaddr = dev->base_addr;
  int new_mode;

  if( dev->flags & IFF_PROMISC ) {                              //如果设备接收模式为混杂模式
    …
    new_mode = SetRxFilter|RxStation | RxMulticast | RxBroadcast | RxProm;
  }
  else if (( dev->mc_list ) || ( dev->flags & IFF_ALLMULTI)     //组传送模式
    new_mode = SetRxFilter | RxStation | RxMulticast |RxBroadcast;
  else
     new_mode = SetRxFilter | RxStation | RxBroadcast;
  outw(new_mode, ioaddr + EL3_CMD);                             //将工作模式写入网卡控制寄存器
}
```

设备标志 dev->flags 设置了 IFF_PROMIC 标志位时，new_mode 变量就初始化为接收目标地址为该网络设备的数据包（RxStation）、组传送数据包（RxMulticast）、广播数据包（RxBroadcast）和所有的数据包（RxPorm）。EL3_CMD 是网卡控制寄存器相对于设备 I/O 起始地址 ioaddr 的偏移量，将 new_mode 写入网卡的控制寄存器，就将网络设备设置成了新的工作模式。

18．allmulti

当这个数据域为非零值时，设备监听所有组传送地址。此数据域不是简单的布尔

（boolean）类型变量，它是一个计数器，因为可能有多个应用程序都需要监听所有组传送地址，在 allmulti 数据域从 0 变为非零值时，dev_set_allmulti 函数向设备发出命令，让网络设备监听所有的组传送地址；当 allmulti 的值变为 0 时，做相反的操作。

19．*dsa_ptr /*atalk_ptr /*ip_ptr /*dn_ptr /*ip6_ptr /*ec_ptr /*ax25_pt /*ieee80211_ptr

这 8 个数据域是指向特定网络层协议实例的数据结构指针，每个网络层协议数据结构中都包含协议自身使用的私有参数信息。例如 ip_ptr 指向 in_device 数据结构，其中包含了 IPv4 协议使用的各种参数和为网络接口配置的 IP 地址。这 8 个数据结构指针及其含义如表 3-7 所示。

表 3-7　数据结构指针及其含义

数据结构指针	描述
*dsa_ptr	dsa 规范数据
*atalk_ptr	AppleTalk 协议配置的数据
*ip_ptr	IPv4 网络层协议数据
*dn_ptr	DECnet 网络层协议数据
*ip6_ptr	IPv6 网络层协议数据
*ec_ptr	Econet 协议数据
*ax25_ptr	AX.25 协议数据
*ieee80211_ptr	IEEE 802.11 协议数据，在注册前分配

20．last_rx

表示网络设备最近一次接收到数据包的时间。

21．dev_addr[MAX_ADDR_LEN]

表示数据链路层地址，也即 MAC 地址或硬件地址。dev_addr 数据域是专门为以太网类网络设备存放硬件地址设置的，因为我们使用的网络设备大多数都为以太网卡，地址长度为 8 个字节，是 eth_type_trans（确定数据包的链路层协议 ID，net/ethernet/eth.c）函数使用的地址。

22．broadcast[MAX_ADDR_LEN]

表示加入到广播地址中的硬件地址。

23．rx_queue

```
*_tx ____cacheline_aligned_in_smp
num_tx_queues
real_num_tx_queues
tx_queue_len
```

rx_queue 为网络设备接收数据包的队列和发送数据包队列。num_tx_queues 指明发送队列个数，有的网络设备会有多个发送队列。real_num_tx_queues 描述了当前可用发送队列数。比如网络设备总的发送队列可以为 num_tx_queues，但有的发送队列可能已满，未满的发送队列为由 real_num_tx_queues 表示。

tx_queue_len 为发送队列的长度，也即每个发送队列中允许存放的最大数据帧数。表 3-8 给出了常用网络设备发送队列的长度值，其值可使用 sysfs 文件系统得到（见 sys/class/net/device_name/目录）。

表 3-8　常见网络设备发送队列的长度

设备类型	tx_queue_len 的值
Ethernet	1000
Token Ring	100
Etherchannel	100
Fibrechannel	100
FDDI	100

24．tx_global_lock

发送队列的访问控制锁，防止在对称多处理器环境下对网络设备发送队列的并发访问。

25．trans_start

表示最后一次传送数据包的时间（用 jiffies 来衡量），网络设备驱动程序在开始传送数据包之前设置 trans_start 的值。trans_start 是用于网络设备检查数据传送错误的数据域。网络设备开始数据传送后，经过一段时间内如果传送过程一直没有结束，即超过了预先设定的传送超时仍没结束，说明传送过程中发生了某种错误，产生传送超时错，这时网络设备驱动程序需要对设备进行复位。

26．watchdog_timeo

watchdog_timer

这两个数据域用于 watchdog 的时间计数。watchdog_timer 是网络层设置传送数据包超时的时钟（以 jiffies 表示），如传送超时，则认为数据包的发送过程出错，调用设备驱动程序的 tx_timeout 函数对传送超时做处理。

watchdog_timeo 是发生传送超时时，设置的标志。

27．refcnt

对网络设备的引用计数。

28．todo_list

网络设备注册到 Linux 内核和取消网络设备注册通常需要两个步骤来完成，第一个步骤完成后，会将设备放入 todo 链表中，由系统调用函数来完成余下的工作。设备的注册过程我们将在下一章中详细讲解。

29．index_hlist

该数据域为以设备索引号为关键字建立的哈希链表，以便于按设备索引号快速查找系统中的网络设备。

30．*link_watch_next

链路查看机制（link watch mechanism）是网络子系统内部管理传输介质的一个部分，捕获传输介质的所有事件。链路查看机制需要的内存空间在创建 struct net_device 实例时分配，不是在需要的时候才分配，是为了防止因为内存分配失败，造成丢失某些事件。

31．reg_state

设备注册和取消注册的状态机。设备注册与取消注册过程可在以下状态之间发生变化，如表 3-9 所示。

表 3-9　reg_state 的值及含义

状态描述符	值	含　　义
NETREG_UNINITIALIZED	0	设备未初始化
NETREG_REGISTERED	1	设备注册已完成，即函数 register_netdevice 已执行完
NETREG_UNREGISTERING	2	调用了 unregister_netdevice 函数，该函数正在执行，但取消注册的过程可能还没有完全结束
NETREG_UNREGISTERED	3	卸载设备过程执行完
NETREG_RELEASED	4	调用了 free_netdev 函数来释放 struct net_device 数据结构

32．*destructor

函数指针，指向 struct net_device 数据结构的析构函数，当卸载网络设备时调用。

33．*npinfo

如果内核配置支持 netpoll 功能，该指针指向存放 netpoll 信息的数据结构 struct netpoll_info。netpoll 是后增加到网络协议栈中的，它的目的是在网络和 I/O 子系统不可用的情况下，使内核能够发送和接收报文，用于远程网络控制台和远程内核调试。驱动程序不一定要支持 netpoll 功能，但如果支持，可使驱动程序在某些情况下功能更强大。

34．*nd_net

该网络设备所属的网络名字空间。从 Linux-2.6 以后，很多内核的子系统中加入了对名字空间的支持，名字空间作用是使不同进程的系统视图不同。

35．*ml_priv

中间层的私有数据结构。

36．*br_port

*macvlan_port

*garp_port

以上 3 个数据域分别为：在数据链路层实现桥的功能、VLAN 和 GARP 协议的实现。

37．struct device dev

表示网络设备在 /sys 文件系统下 class/net/name 目录下的入口。每一个注册了的网络设备，在 /sys 文件系统的 class/net 目录下都有一个与设备名相同的目录，例如：/sys/class/net/eth0，输出了网络设备的配置信息。这些配置信息包括：

- Address：网络设备的硬件地址。
- addr_len：地址长度。
- broadcast：广播地址等。

3.3　struct net_device 数据结构中数据域的功能分类

基于 3.2 节对 struct net_device 数据结构各数据域的介绍，在这一节中我们对 struct net_device 数据结构中数据域的功能特性做出归纳和总结，struct net_device 中的某些数据域从名称来看非常相似，如描述网络设备状态的几个字段，其实它们存在细微的差别，描述

了网络设备不同方面的特性、状态和功能。

3.3.1 设备管理域

Linux 内核从几个不同角度来识别、管理以及查询网络设备，这主要是为了便于在不同功能模块中能快速地查找到要操作的网络设备。

1．设备名

在 struct net_device 数据结构中有两个标识网络设备名的数据域：name 和 ifalias，用于不同场合。

2．设备标识符

在 struct net_device 数据结构中有 3 个标识符，它们的作用是不同的，不要混淆了，它们是：ifindex、iflink、dev_id。在 3.2 节中我们介绍了：ifindex 是系统分配给网络设备的标识符；iflink 用于虚拟网络设备；dev_id 用于共享网络设备。

3.3.2 设备配置管理域

struct net_device 数据结构中包含了大量关于网络设备的配置信息供内核访问，其中有关于硬件的配置、数据链路层的配置、网络层相关协议的配置等。对于 struct net_device 中的某些配置域，Linux 内核会根据网络设备的类型给出一些默认值，而有的数据域需由网络设备驱动程序来填写，网络设备驱动程序也可以在初始化时改变默认值。除此以外，有的配置数据可以在程序运行时用 ifconfig 或 ip 等应用工具来设置或修改。

几个特别的网络设备硬件参数：base_addr、if_port、dma、mem_start、mem_end 和 irq，通常是在网络设备驱动程序模块装载时由用户给出的。需要注意的是，虚拟设备不使用这些参数。struct net_device 数据结构中存储配置信息的相关数据域有如下几类。

1．硬件配置信息

- I/O 操作相关配置。实现网络设备输入/输出操作需要的信息包括：网络设备上的数据缓冲区、状态/控制寄存器映射在存储器中的区域（Memory mapping）：mem_start、mem_end，或设备按 I/O 端口操作时 I/O 端口的基地址 base_addr，以及 irq、dma 等。
- 硬件连接接口信息：if_port。
- 硬件功能特性：features。

2．网络设备物理层特性

网络设备的物理层特性对于不同类型 type 的网络设备其配置值不同，但同一类网络设备对于不同的生产厂商其值都是相同的。根据 type 值的不同，相应的 mtu、hard_header_len 、tx_queue_len 值也不同，其值我们在 3.2 节已给出了相应的列表。

3．数据链路层配置

在数据链路层的配置信息有数据链路层协议相关的配置信息，如协议头长度：needed_headroom、needed_tailroom。根据网络设备的工作模式配置不同，其数据链路层地

址配置信息包括以下几类。

- 硬件地址相关配置信息：perm_addr、addr_len、*uc_list、uc_count、add_list_lock。
- 组传送地址：*mc_list、mc_count。
- 混杂模式地址数：uc_promisc。

4．网络层特性配置

*dsa_ptr、*atalk_ptr、*ip_ptr、*dn_ptr、*ip6_ptr、*ec_ptr、*ax25_ptr、*ieee80211_ptr，以上几个域中，可以同时有几个数据域都不为空，因为网络设备可以同时为多个网络层协议服务，但大多数情况下只有一个数据域在使用。

3.3.3　设备状态

Linux 内核为了控制与网络设备之间的交互，每个设备驱动程序必须维护一些设备的状态信息，比如，网络设备各项操作的时间戳，网络设备最后接收与发送数据包的时间，使用了哪些标志来指明接口需要完成的操作。在对称多处理器（SMP）系统中，内核还必须保证不同的 CPU 对同一网络设备操作的正确性，在 struct net_device 数据结构中，以下数据域用于这类信息描述。

- 设备收发数据包时间信息：trans_start、last_rx。
- 设备物理工作状态：state。
- 网络设备通信模式状态：flags、gflags、priv_flags。
- 设备的注册状态：reg_state。
- 设备的分组状态：*master。

在这里需要对描述网络设备状态和特性的数据域作一些说明，它们分别是：state、flags 和 feature。

1．设备物理工作状态 state

state 主要用于描述网络设备在物理上的工作状态，比如网卡是否已连接，网络设备的传送队列是否可以开始传送新的数据包或队列是否已满，传输介质是否已连接等。

2．设备通信模式状态 flags

flags 中有的标志位描述的是网络设备的状态，如 IFF_UP、IFF_RUNNING，但 flags 主要描述的是网络设备的通信能力，如 IFF_BROADCAST、IFF_ALLMULTI 等。

3．设备硬件功能特性 feature

feature 描述的是网络设备在硬件上具备的其他功能特性，例如设备硬件是否具有对数据包做校验和的功能，设备硬件是否可完成数据包的分割和重组等。

3.3.4　统计

struct net_device 数据结构中 stats 数据域包含了网络设备在数据包的收发过程中最常规的详细统计数据。这些统计信息用户可使用用户空间工具查询，如 ifconfig。

3.3.5 设备链表

系统中所有的网络设备都会初始化产生一个 struct net_device 数据结构的实例，struct net_device 数据结构实例代表了在系统运行状态下网络设备在内核中的实体，这些 struct net_device 的实例向内核注册后，系统会将其插入到一个全局链表和两个哈希链表中。此外在 Linux 的网络子系统中增加了一种新的接收数据包的方式：NAPI（NAPI 的实现细节我们将在第 6 章中介绍）。目前只有少数网络设备支持 NAPI 功能，为了便于管理，Linux 支持 NAPI 工作方式的网络设备插入到另一个链表中。以下的数据域用于对这些任务的管理。

- dev_list：链接每个 struct net_device 实例的全局链表。
- napi_list：支持 NAPI 传送的网络设备的 struct net_device 实例的链表。
- todo_list：网络设备推迟注册/注销设备实例链表。
- name_hlist：以网络设备名为关键字建立的哈希链表。
- index_hlist：以网络设备接口索引号为关键字建立的哈希链表。

3.3.6 链路层组传送

组传送（multicast）是将数据传送给多个接收主机的机制，组传送可以在 L3（网络层）和 L2（数据链路层）实现。这里我们描述的是为了支持在数据链路层的组传送功能，struct net_device 数据结构中需要的数据域。

数据链路层的组传送实现可以通过使用特定的地址或数据链路层协议头的控制信息来描述。组传送 MAC 地址是通过 MAC 地址中的一个特定位的设置与其他地址范围区分的，也即有 50%的 MAC 地址可能是组传送的地址。2^{48} 的 50%是一个相当大的数，如果把网络接口加入到组传送地址组中，数据传送会更快更有效，这时只需监听所有的组传送地址，而不需维护大量的地址列表，struct net_device 数据结构中的 flags 有一个标志位（IFF_ALLMULTI）说明设备是否需要监听所有组传送地址，什么时候设置和清除该位由 struct net_device 中的 allmulti 数据域控制。

每个 struct net_device 结构中都有一个 struct dev_addr_list *mc_list 数据对象指针，是数据链路层组传送地址的列表。数据链路层的组传送的地址可以使用函数：dev_mc_add 和 dev_mc_delete 将网络设备的地址加入到组传送地址列表中或从组传送地址列表中移走。在 struct net_device 数据结构中与数据链路层组传送功能相关的数据域有以下几种。

- struct dev_addr_list *mc_list：指向网络设备的组传送地址列表的首地址的指针。
- int mc_count：网络设备侦听的组传送地址个数，也等于由指针 mc_list 指向的链表的长度。
- int allmulti：需要监听所有组传送的应用计数。

3.3.7 流量管理

Linux 内核的流量控制子系统增加了很多新功能，代表了对 Linux 内核的扩展。网络设备处理数据包的速度与网络设备的接收数据包队列和发送数据包队列的数量、队列大小有极大的关系。在 struct net_device 数据结构中与数据包收/发队列管理相关的数据域和队列配

置如下：

- rx_queue。
- _tx。
- tx_queue_len。
- num_tx_queue。
- real_num_tx_queue。
- tx_global_lock。

关于流量管理在这里需要做一些说明，在 Linux 内核的网络子系统中，另一个基础数据结构就是与每个网络设备相连的发送队列。网络协议栈的代码将数据包放入设备的发送队列，随后调用网络设备驱动程序的函数 hard_start_xmit 来通知网络设备数据包已准备好，可以开始发送了，接下来就由网络设备驱动程序把发送队列中的数据包放入硬件的缓冲区中，其结果类似图 3-4 所示。

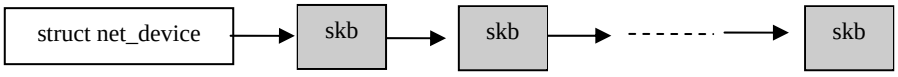


图 3-4　将数据包的 Socket Buffer 放入设备发送队列

早期的 Linux 版本一直以上述方式工作，大多数网络设备只有一个数据包发送队列。但随着网络设备的发展，上述工作方式有很大的局限性，这种工作方式不能很好地反映有多个发送队列的网络设备的工作模式，目前有多个数据包发送队列的网络设备越来越多，特别是无线网络设备。

例如实现无线多媒体扩展的无线网络设备可以有四个不同类型的服务：图像、声音、最好效果和后台服务。图像和声音数据在设备中通常具有较高优先级，它们需要优先传送，而且这些数据包的传送会占用设备更长时间的无线通道。另外，存放图像、声音数据的传送队列比较短，对图像、声音数据包的处理方式是：如果图像数据队列不快速传送它的数据包，接收方就不再接收它的信息。所以一般对图像、声音数据的处理如果延迟太长，就扔掉数据包。

后台服务级别的数据包只有在没有别的数据要传送时才传送它们，它非常适合某些低优先级的数据包，如 bittorrent 下载或电子邮件的传送。后台数据处理队列相对较长。

在有多个发送数据包队列的网络设备中，每种服务有它们自己的发送队列，这种区分流量的方式使硬件更容易选择下一时间发送什么，也可以独立地定制各个队列的大小。因为每个发送队列需要独立调度，所以 Linux 内核提供了一个新的数据结构 struct netdev_queue 来管理队列，其中封装了所有管理队列需要的信息。有多个发送队列的网络设备驱动程序可以定义一个 struct netdev_queue 数据结构的数组，数组的首地址由 struct net_device 数据结构中的*_tx 数据域保存。

多发送队列的管理模式如图 3-5 所示。

struct netdev_queue 数据结构同样定义在 include/linux/netdevice.h 头文件中。

```
struct netdev_queue {
  struct net_device     *dev;                  //队列所属的网络设备
  struct Qdisc          *qdisc;                //队列操作函数与 Socket Buffer 链表
```

```
    unsigned long    state;                  //队列状态
    spinlock_t       _xmit_lock;             //防止对队列并发访问的控制锁
    int              xmit_lock_owner;
    struct Qdisc     *qdisc_sleeping;        //休眠队列
} ____cacheline_aligned_in_smp;
//队列状态值
enum netdev_queue_state_t
{
    __QUEUE_STATE_XOFF,
    __QUEUE_STATE_FROZEN,
};
```

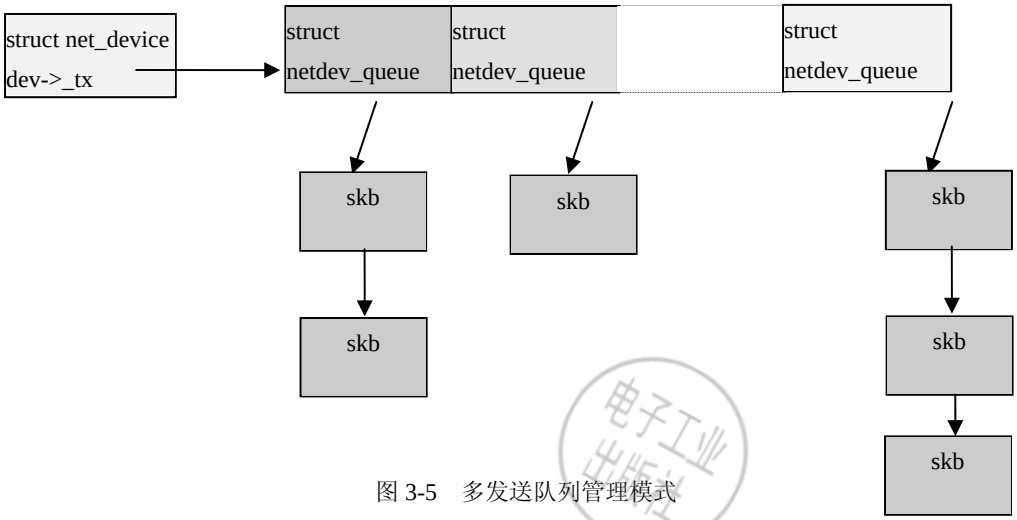


图 3-5　多发送队列管理模式

发送数据包的队列实际上是以特定的数据结构格式存放于设备的缓冲区中。一旦网络设备的这些发送队列建立起来以后，就可以制定与每种服务相关的发送策略。每个发送队列的管理是独立的，这样单队列的设备驱动程序并不需要改变，其他的支持多队列的设备驱动程序首先需要用以下几个新函数来初始化设备的内存分配。

1．为设备分配内存空间

```
struct net_device *alloc_etherdev_mq(int sizeof_priv, unsigned int queue_count)
```

用上述函数来代替原来的 alloc_etherdev 函数来为设备分配内存空间。新的函数中 queue_count 参数描述了设备支持的最大发送队列数。实际的发送队列数存放在 dev->real_num_tx_queue 数据域中，注意这个值只有在设备卸载后才能改变。

2．管理发送队列

对于支持多个发送队列的网络设备，其设备驱动程序通过函数 ndo_start_xmit 从任意的指定队列得到发送的数据包，为了区分当前哪个队列为数据包发送队列可用以下函数来查看。

- 该函数的返回值是当前传送数据包队列的索引号。

```
u16 skb_get_queue_mapping(struct sk_buff *skb)
```

- 选择发送队列，由定义在 struct net_device 数据结构的回调函数实现。

```
u16 (*select_queue)(struct net_device *dev, struct sk_buff *skb)
```

3．发送队列状态

支持多发送队列的网络设备的驱动程序还需要完成的改进就是要通知 Linux 内核网络设备某个发送队列的状态，为了实现这一功能，可以使用如下一组新的函数。

- 获取当前发送队列。

```
struct netdev_queue *netdev_get_tx_queue(struct net_device *dev, U16 index);
```

- 启动发送队列。

```
void netif_tx_start_queue(struct netdev_queue *dev_queue);
```

- 唤醒发送队列。

```
void netif_tx_wake_queue(struct netdev_queue *dev_queue);
```

- 停止发送队列。

```
void netif_tx_stop_queue(struct netdev_queue *dev_queue);
```

4．操作所有发送队列

如果需要一次性操作网络设备的所有发送队列，可由下面的一组函数来实现。

```
void netif_tx_start_all_queues(struct net_device *dev);
void netif_tx_wake_all_queues(struct net_device *dev);
void netif_tx_stop_all_queues(struct net_device *dev);
```

3.3.8　常规域

除了前面提到的链表管理域外，struct net_device 中还有一些其他数据域是用于管理该数据结构本身的，这些数据域描述网络设备自身的一些常规信息。比如：数据域 refcnt 表示对网络设备的引用计数，只有对设备的引用计数为 0 时，才能取消设备在 Linux 内核中的注册。

其他的一些常规数据域还有：padded、watchdog_timer、 watchdog_timo 等。

3.3.9　操作函数结构

在 Linux 内核现在的版本中，将操作网络设备的函数进行了分类，从 struct net_device 数据结构中提取出来，放在 3 个函数结构块中。

- net_dev_ops：该数据结构中包含了一组函数指针，是网络设备驱动程序需要实现的函数集合。
- ethtool_ops：该数据结构中包含了一组函数指针，驱动程序可以实现这组函数，支持修改和报告网络设备设置的函数集合。
- header_ops：操作数据链路层协议头的函数集合。

3.4　函数指针

在 Linux 内核代码中，包括网络子系统的代码在内，大量使用了函数指针来隐藏硬件细节。网络设备与网络协议栈之间的接口最重要的任务就是抽象硬件特性，它必须让网络设备驱动程序的函数映射到一个统一的接口上，供上层协议栈访问。这个任务是通过 struct

net_device 数据结构中的函数指针来实现的。struct net_device 数据结构中的这些函数指针让网络设备驱动程序编写人员在实现自己的网络设备驱动程序时，可以做到不同 struct net_device 数据结构实例有不同的函数名。

有的函数与网络设备的硬件特性相关，必须在网络设备驱动程序的初始化函数中建立；有的函数专门用于实现网络适配器的数据链路层（MAC 层）协议，可以由特殊的方法来建立，在网络设备驱动程序中不需要实现的函数指针可以初始化为空*（NULL）。

接下来我们将介绍网络设备驱动程序中的方法，在这里主要描述这些方法实现的功能，方法的代码实现与网络设备硬件有关，我们将在第 5 章分析网络设备驱动程序实现时详细介绍。

网络设备驱动程序的方法可以分成两大类：基本方法和可选方法。基本方法是那些保证网络设备正常工作的函数；可选方法实现一些更高级的功能特性，但不是要求所有网络设备驱动程序必须实现的方法。

为了使 struct net_device 数据结构的组织更加清晰，在 Linux-2.26.29.6 版本中，将网络设备驱动程序要实现的方法从 struct net_device 数据结构中提取出来，集中安排在 struct net_dev_ops 数据结构中。为了保持对旧版本的向后兼容，原来定义在 struct net_device 数据结构的驱动程序中的方法，加上#ifdef CONFIG_COMPAT_NET_DEV_OPS 条件编译后，旧版本的网络设备驱动程序函数实现仍与原定义的函数指针相关联。为了加以区分，定义在 struct net_dev_ops 数据结构中的新函数指针名前加了前缀“ndo_”。在以下对函数方法的说明中，我们仍以原函数名来讲解。

3.4.1 设备初始化

```
int(*init)(struct net_device *dev)/ndo_init
```

init 函数是网络设备驱动程序提供的设备初始化程序，完成网络设备搜索和初始化的任务。该函数完成的主要功能包括以下几种。

- 创建网络设备的 struct net_device 数据结构实例。
- 初始化 struct net_device 的相关数据域，如设备名（name）、I/O 端口地址（base_addr）、中断号（irq）等。
- 向 Linux 内核注册设备，使设备处于可工作状态。

```
void (*uninit)(struct net_devie *dev)/ ndo_uninit
```

在注销设备时调用该函数，用于执行卸载设备时网络设备驱动程序需要完成的一些特定的操作。

```
int (*open)(struct net_device *dev);/ndo_open
```

此函数打开网络设备接口，不过只能打开已注册的网络设备。无论什么时候用 ifconfig 命名来激活网络设备接口都会调用 open 函数，在 open 函数中注册所有接口需要的资源（I/O 端口、中断号、DMA 通道等）。open 函数的功能就是打开硬件，执行所有网络设备需要的初始化工作。

```
int (*stop)(struct net_device *dev);/ ndo_stop
```

停止网络设备。当卸载网络设备后应停止设备，这个函数执行所有与打开网络设备操作相反的过程，如释放资源、关闭硬件等。

3.4.2　传送

```
int (*hard_start_xmit)(struct sk_buff *skb, struct net_device *dev); /ndo_start_xmit
```

这是初始化数据包发送过程的方法。要发送的数据包（包括协栈各层协议头信息和数据负载）存放在 Socket Buffer 中。如果 hard_start_xmit（ndo_start_xmit）函数执行成功，发送数据包就放入了网络适配器的发送数据缓冲区中。

```
void (*tx_timeout) ( struct net_device *dev);/ ndo_tx_timeout
```

数据包发送超时错误处理函数。如果经过一定的时间周期，数据包的传送还没有完成，调用该函数来处理数据包传送的超时错误，恢复发送过程通常需要对网络设备进行复位。

```
int (*poll) ( struct net_device *dev, int  *quota);
```

这是由 NAPI 兼容的网络设备驱动程序提供的函数。在禁止中断的情况下，用轮询的模式来操作网络接口，读取网络设备接收到的数据包。

```
void (*poll_controller) ( struct net_device *dev);/ ndo_poll_controller
```

该函数用于让驱动程序在中断被禁止的情况下查看网络接口上发生的事件。它主要用于一些特殊的内核网络任务，如远程控制终端，通过网络跟踪调试内核代码等。

```
u16 (*select_queue) ( struct net_device *dev, struct sk_buff  *skb);/ ndo_select_queue
```

该函数在支持多个发送队列的网络设备中选择网络设备的发送队列，将包含发送数据包的 Socket Buffer 放入网络设备的某个发送队列中。

```
void (*set_multicast_list) ( struct net_device *dev);/ ndo_set_multicast_list
void (*set_rx_mode) ( struct net_device *dev);/ ndo_set_rx_mode
```

根据 struct net_device 数据结构中标志域 flags 的 IFF_PROMISC、IFF_MULTICAST 和 IFF_ALLMULTI 标志位来设置网络设备接收模式（set_rx_mode 函数），将网络设备地址放入组传送地址列表 mc_list 中（set_multicast_list 函数）。

在 3.2 节中描述 struct net_device 的区域的含义时，我们曾经介绍，在 struct net_device 数据结构中 mc_list 和 mc_count 是用于管理数据链路层组传送地址列表的数据域，set_multicast_list 函数就用于通知网络设备驱动程序对设备做相关的配置（将设备的地址加入到组传送地址链表中），监听在这些地址的组传送数据包。

```
void (*change_rx_flags) ( struct net_device *dev, int flags);/ ndo_change_rx_flags
```

修改网络设备的接收工作模式标志。当 struct net_device 数据结构的标志域 flags 中任何会影响接收数据包过滤的标志位发生变化时，调用该函数，使设备驱动程序立即同步由 set_multicast_list/set_rx_mode 设置的地址列表。

3.4.3　硬件协议头

在以前的 Linux 版本中 struct net_device 数据结构的函数指针中包含了一系列创建、更

新、解析网络硬件协议头信息的函数指针，它们主要是：

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

hard_header 函数根据数据包中的数据链路层的源地址和目的地址构建硬件协议头数据（该函数应在调用 hard_start_xmit/ndo_start_xmit 之前调用），它将输入参数组织成特定硬件协议头信息需要的数据。对于以太网类型的网络设备，该函数指针默认的初始化为指向 eth_header 函数（在 net/ethernet/eth.c 文件中）。

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

hard_header_parse 函数用于解析包含在 skb（Socket Buffer）中网络数据包的源地址，并将源地址复制到 haddr 指向的缓冲区，向该函数的调用程序返回数据包源地址的长度。以太网类型的网络设备使用 eth_header_parse 函数作为该函数的默认实现。

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

在发送数据包之前，地址解析协议 （Address Resolution Protocol，ARP，根据 IP 地址发现相应的网络设备的硬件 MAC 地址）将 IP 地址转换成硬件地址后，调用该函数来重新构建硬件协议头数据。

```
int (*header_cache) ( struct neighbour *neigh, struct hh_cache *hh);
```

函数 ARP 协议实例调用 header_cache 来填写 struct hh_cache 数据结构，struct hh_cache 数据结构实例中缓存由邻居协议（ARP）子系统转换好的硬件地址（由 IP 地址转换成对应的 MAC 地址）、L2 层协议头。几乎所有的以太网设备都可以使用默认的 eth_header_cache 函数（在 net/ethernet/eth.c 文件中）来实现这个功能。

```
int (*header_cache_update) (struct hh_cache *hh, struct net_device *dev, unsigned char  * haddr);
```

当发送数据包的目标地址发生变化时，该方法用于对缓存在 struct hh_cache 结构中的目标硬件地址、L2 协议头进行更新。以太网卡可使用函数 eth_header_cache_update（在 net/ethernet/eth.c 文件中）作为其默认实现。

在目前的 Linux 内核的 struct net_device 数据结构中，这一系列操作硬件协议头信息的函数指针被移到了 struct header_ops 数据结构中，struct header_ops 包含在 struct net_device 结构中。这样使 struct net_device 结构变得更加清晰。header_ops 结构定义做了如下更新：

```
struct header_ops {
  int   (*create) (struct sk_buff *skb, struct net_device *dev,
                   unsigned short type, const void *daddr,
                   const void *saddr, unsigned len);
  int   (*parse)(const struct sk_buff *skb, unsigned char *haddr);
  int   (*rebuild)(struct sk_buff *skb);
#define HAVE_HEADER_CACHE
  int   (*cache)(const struct neighbour *neigh, struct hh_cache *hh);
  void  (*cache_update)(struct hh_cache *hh,
                        const struct net_device *dev,
                        const unsigned char *haddr);
};
```

目前使用的网络设备大部分都是以太网类型的网卡，在 Linux 内核网络子系统代码中的 net/ethernet/eth.c 文件中，为以太网类的网络设备实现了大部分具有共性的函数，如

eth_header（创建 ethernet 头）、eth_rebuild_header、eth_header_parse 等。我们在前面已经提到，系统初始化时会将 header_ops 数据结构中的各函数指针初始化为指向以太网类的默认函数实现。这样我们在编写设备驱动程序时就可以把注意力集中在必须要实现的方法上。

```
const struct header_ops eth_header_ops ____cacheline_aligned = {
   .create= eth_header,
   .parse= eth_header_parse,
   .rebuild= eth_rebuild_header,
   .cache= eth_header_cache,
   .cache_update= eth_header_cache_update,
};
```

3.4.4　网络统计状态

```
struct net_device_state* (*get_stats) ( struct net_device *dev);
```

通过此函数能获取网络设备的统计状态信息。当应用程序需要获取网络设备的统计信息时，通过用户地址空间应用工具调用该函数。例如用 ifconfig 命令或 netstat　–i 命令显示网络统计状态信息时，就是由这个函数获取的数据。

3.4.5　修改配置

```
int* (*set_config) ( struct net_device *dev, struct ifmap *map,);/ ndo_set_config
```

配置网络设备的参数，例如硬件参数、irq、base_addr 等。这个函数是驱动程序配置设备的接口，设备的 I/O 地址和中断号可以在运行时用函数 set_config 来修改。这个功能可以帮助系统管理员在没有探测到接口时手动调用该方法来设置网络设备接口。

```
void* (*set_mac_address) ( struct net_device *dev, void *addr,);/ ndo_set_mac_address
```

如果网络设备支持对硬件地址的修改功能，可以实现这个方法。大多数网络设备都不具有这个功能。另外一些网络设备默认使用 eth_mac_addr 函数（在 net /ethernet/eth.c 文件中）来实现硬件地址修改功能。eth_mac_addr 函数只是将新的硬件地址复制到 dev->dev_addr 数据域中，而且在网络设备还没运行时执行这个操作。如果驱动程序使用了 eth_mac_addr 函数，应在打开设备时（open）用 dev->dev_addr 数据域来初始化网络设备的硬件地址。

```
int* (*change_mtu) ( struct net_device *dev,  int  new_mtu,);/ ndo_change_mtu
```

修改网络设备的最大传输单元（MTU）值。如果用户修改 MTU 时，设备驱动程序需要做一些特殊的处理，则网络设备驱动程序就需要实现自己的 change_mtu 函数，如无须做特殊处理，内核提供的默认函数可以完成正确的修改操作。

```
int* (*validate_addr) ( struct net_device *dev);/ ndo_validate_addr
```

检查 dev 中的地址域 dev_addr 中的值是否为有效地址。

```
int* (*neigh_setup) ( struct net_device *dev, struct neigh_parm *);/ ndo_neigh_setup
```

用于建立与相邻协议（ARP）的连接。

3.4.6 ioctl 命令

```
int* (*do_ioctl) ( struct net_device *dev, struct  ifreq * ifr, int cmd);/ ndo_do_ioctl
```

执行网络接口特定的 ioctl 命令。ioctl 是应用程序给设备发送命令的系统调用，do_ioctl 方法用于处理某些 ioctl 命令。

3.5 本章总结

struct net_device 数据结构是网络设备在 Linux 内核运行过程中的实体，是网络控制器与 TCP/IP 协议栈通信的标准接口。掌握 struct net_device 数据结构是网络设备驱动程序开发的基础，也是理解网络数据如何从网络介质传入内核地址空间的基本要求。struct net_device 数据结构具有以下特点。

- 数据结构庞大，数据域覆盖了不同类型、不同功能特点、支持不同物理协议网络设备的所有属性。
- 使用了大量函数指针，使用各种网络设备驱动程序可以具有自己的函数名和不同实现代码，但不影响 TCP/IP 协议栈的实现。
- 对上层 TCP/IP 协议栈隐藏了网络设备硬件特性和差异，与 dev.c 中的通用设备功能函数结合形成了 TCP/IP 协议栈与网络设备硬件之间的统一接口。
- 对支持多种数据接收模式，使 Linux 操作系统可支持最广泛的网络设备，网络数据传送快速高效。

基于 struct net_device 数据结构的以上特点，在我们开发网络设备驱动程序与实现数据发送时，不是 net_device 所有的数据域都会使用，也不是所有函数指针都需要实现。对于本章，读者可以作为手册一样使用，在开发网络设备驱动程序时，查阅需要的数据域。

第 4 章　网络设备在 Linux 内核中识别

本章在讲解 Linux 内核启动、组件初始化、命令行参数传递和解析实现的一般过程基础上，分析了网络子系统、网络设备和启动初始化过程，重点讲解网络设备在整个内核启动过程中何时被内核识别，网络设备实例在内核与驱动程序中初始化，注册到内核的实现。

在系统运行过程中，物理网络设备在 Linux 内核代码中的实体是 struct net_device 数据结构的实例。struct net_device 数据结构的实例要成为能被内核识别、代表正常工作网络设备的代码描述，与内核代码融为一体，需要经过实例创建、初始化、设备注册等一系列过程。这个过程涉及以下几个方面的问题：

1．Linux 内核识别网络设备的步骤

① struct net_device 数据结构的实例由谁在何时创建。

② struct net_device 数据结构实例的各数据域与函数指针由谁在何时赋值初始化。

③ struct net_device 数据结构的实例何时注册到内核中。

网络设备编译到内核的方式

除此之外，网络设备驱动程序模块可以直接编译到 Linux 内核代码中，也可以在运行时以模块（module）的形式装载到内核，不需要时从内核中卸载。在第一种情况下，当内核发现了一个网络硬件设备时，就调用探测函数 probe 寻找与设备匹配的驱动程序，初始化 struct net_device 数据结构的实例。在第二种情况中，会在装载网络设备驱动程序模块时初始化 struct net_device 数据结构的实例。由此可见，struct net_device 数据结构实例的创建、初始化、注册都是由网络设备驱动程序中的 API 实现的，只是设备驱动程序装载到内核中的方式不同，调用者与调用时机也不同，本章中我们会介绍以上两种情况下网络设备的初始化过程。

2．网络设备初始化的一般过程

网络设备驱动程序初始化函数的首要任务就是分配网络设备数据结构实例的内存空间；其次是初始化数据结构，初始化设备的收发队列，建立函数指针；一旦初始化完成，就注册设备。从这以后，设备就挂接到了 TCP/IP 协议栈上。如果设备本身除了一般的初始化过程外还需要某些特殊操作，则初始化过程继续进行。

要使网络设备处于一个可工作的状态，网络设备实例初始化是一项重要工作。网络设备实例及设备驱动程序是支持网络子系统协议栈的重要部分，需在网络子系统初始化完成的前提下，才能完成网络设备实例的创建与初始化。

作为 Linux 内核组件之一的网络子系统，网络协议栈和网络设备驱动程序都运行在内核的地址空间，它们的初始化是整个内核初始化的一个部分，具有内核启动、初始化过程的一般特点。本章我们首先会分析内核启动初始化的过程与特点，在此基础上描述网络子系统的初始化任务，最后介绍网络设备的初始化与注册过程。

4.1 内核初始化的特点

内核启动时的重要任务之一是首先为内核的各子系统建立运行的共享资源，如内存管理、中断、时钟子系统等。内存正常初始化是整个系统运行的首要条件。随后内核逐一启动各个子系统，使它们正常工作，让各子系统的服务能为内核别的组件所使用。其中各种输入/输出设备的初始化是应用—内核—硬件设备交换信息的重要渠道。

1．内核子系统初始化的共同特点

- 需要在系统启动时执行，这时是内核所用组件初始化例程执行的时间。
- 初始化例程执行完以后，这部分代码在系统后面的运行过程中就不再需要了，系统可以释放这部分代码和它们使用的数据对象所占用的内存空间。

2．内核初始化程序的组成

在内核启动的早期阶段，初始化程序可分为两大部分。

- 各种典型的和必需的子系统初始化。这些子系统的初始化需要按照一定的顺序完成，例如内核不能在网络子系统的初始化完成之前就初始化网络设备。
- 内核的其他组件初始化。这些组件的初始化过程没要严格的顺序要求，同一优先级的代码可以按照任何顺序执行。

3．为内核启动提供参数

内核启动时的另一个特性是允许用户通过 bootloader（如 grub、LILO 或 U-boot 等，主要是嵌入式系统使用的启动程序）给内核传送各种配置参数，称为命令行参数。有经验的用户可以利用这种机制让内核的启动更有效。例如，我们知道自己的网卡使用的 I/O 端口地址（base_addr）、中断号（irq），以及共享内存区域时，就可以在系统启动时以命令行参数的形式将这些配置直接传送给内核，此后内核就可以使用这些配置参数来快速初始化网络设备的数据结构实例了。在内核代码中提供了一套机制来实现命令行参数的定义和传递。

4.1.1 命令行参数

在 bootloader 中为内核提供命令行参数的格式是：

```
参数名=值
```

如果你查看调度内核的启动程序 grub，可以看到这样的命令行参数：root=/dev/sda。该命令行参数为内核指定 Linux 根文件系统所在的设备。当你打开安装了 Linux 的计算机进入到启动程序界面时（如果台式机的 bootloader 为 grub），输入“a”可以在 bootloader 环境中加入新的命令行参数，例如以下的命令行参数：

```
netdev= 5,0x260,eth0  netdev=15,0x300,eth1
```

向内核分别指定了网络设备 eth0 和 eth1 的中断号和 I/O 端口的起始地址。

如果是嵌入式系统，大多使用的是 U-boot 作为 bootloader。在 U-boot 中同样可以通过设置环境变量指定传给内核的命令行参数：

```
#define CONFIG_EXTRA_ENV_SETTINGS
 “netdev = eth0 5 0x260\0”
…
```

用户可以向内核传递什么命令行参数，需要在内核代码中事先定义。

1．如何定义命令行参数

内核的各子系统可以使用_ _setup 宏来指定自己在系统启动过程中可接收的命令行参数，并将这些命令行参数注册到系统中。_ _setup 宏的格式为：

```
_ _setup( string, function )
```

其中传给_ _setup 宏的参数有两个：string 是命令行参数的名字符串，比如 netdev；function 是解析与处理该命令行参数的函数，即如果用户在系统启动过程中，通过 bootloader 命令行给 Linux 内核的某个子系统定义的命令行参数传了值，其值由 function 函数负责处理。在内核子系统中实现注册命令参数的过程如下。

① 编写命令行参数的处理函数，处理函数前以关键字_ _init 说明。

```
static int _ _ init my_function(char *str)
{
  …
}
_ _ setup( “netdev=” , my_function);
```

② 紧跟在处理函数后用，宏_ _setup 注册命令行参数和处理命令行参数的函数（_ _setup 宏定义在 include/linux/init.h 中）。

2．命令行参数的解析

内核启动时，命令行参数的解析在 init/main.c 文件中的函数 start_kernel 中完成，通过函数 parse_early_param 调用 parse_args 或在 start_kernel 中直接调用函数 parse_args 来解析。

parse_args 函数的功能是：以重启动程序（boot loader）处接收到的命令行参数名作为关键字，用此关键字在内核中注册的命令行参数中查找，一旦找到匹配的命令行参数，就执行该命令行参数注册的处理函数。赋给命令行参数的“值”是命令行参数处理函数的输入参数。如果没找到匹配的命令行参数，在内核启动结束时，最终将该关键字传给第一个用户进程 init 来处理。

从以上所述可以看到，start_kernel 分两次调用函数 parse_args，一次是间接调用，一次是直接调用，也即命令行参数的解析分为两次完成。之所以要分两次解析命令行参数，是因为内核启动时的配置选项划分成了以下两大类。

（1）早期选项

在内核启动期间有的选项必须要先于其他的选项处理。内核用 early_param 宏来定义这类选项，而不是_ _setup。这些选项就由 parse_early_param 函数调用函数 parse_args 来解析。用宏 early_param 注册的命令行参数与用宏_ _setup 注册的命令行参数，唯一不同的是前者有一个特殊的标志“early”，这样内核就能够将两者区分开。

（2）默认选项

大部分命令行参数都是默认选项。这些选项由宏__setup 定义并且在第二次调用 parse_args 函数时解析。

3．命令行参数的存放和管理

无论是 early_param 宏还是_ _setup 宏，都是__setup_param 宏的包装宏（它们都定义在 include/linux/init.h 文件中），命令行参数与其处理函数由 obs_kernel_param 数据结构描述。

```
struct obs_kernel_param{
  const char *str;                          //指向存放命令行参数名的字符串
  int ( *setup_func)(char*);                //函数指针，指向命令行参数的处理函数
  int early;                                //标志 early_param 注册的命令行参数
}
```

随后，__setup_param 宏将所有 obs_kernel_param 实例放入内存中由.init.setup 标志的段内，也即所有描述命令行参数的 obs_kernel_param 实例集中存放在由.init.setup 标志的内存段中。这样做有以下两个目的。

- 便于遍历以关键字“str”为索引的所有命令行参数。内核中使用了两个 obs_kernel_param 类型的指针__setup_start 和__setup_end 来标记.init.setup 内存区域的起始和结束位置。
- 内核在初始化结束后可以快速释放这个区域中数据结构所占用的内存空间。

由此可见，用 early_param 和__setup 宏注册的所有命令行参数对象都会放在以指针__setup_start 和__setup_end 指定的内存区域中（也即.init.setup 段）。该区域在内核初始化结束后将释放，随后其他进程可以使用这段内存区域。

调用 parse_args 函数做第二遍解析的命令行参数会复制到__start_param 和__stop_param（在 start_kernel 函数中定义）之间的区域中。这部分区域在内核初始化结束后不会释放，这些参数值会通过/sys 文件系统向用户空间输出，可以供用户应用程序使用。以上提到的函数、宏和数据结构所在的文件如下图 4-1 所示。读者可以阅读源代码，以便更深刻地理解内核和网络子系统的命令行参数设置。

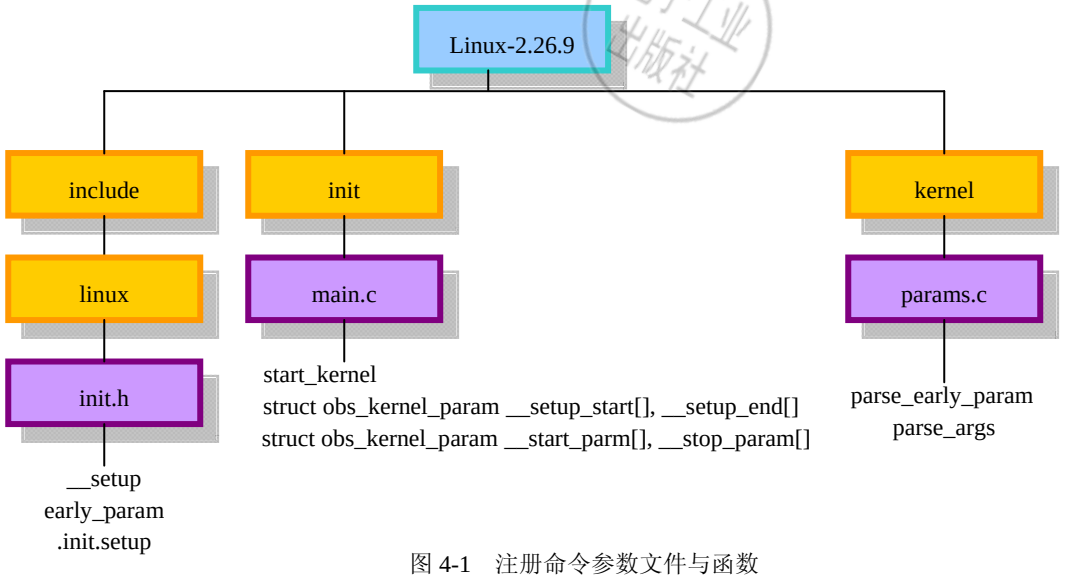


图 4-1　注册命令参数文件与函数

4.1.2　网络子系统的命令行参数

接下来，我们介绍网络子系统如何注册命令行参数和命令行参数处理函数，以及在网络子系统中这些命令行参数的作用。

1．网络子系统定义的命令行参数

在 Linux 网络子系统中定义了命令行参数"netdev="，用户在内核启动时或在运行时加载网络设备驱动程序模块时使用该函数给系统提供网络设备的硬件配置参数。用户可以给网络子系统传递的硬件配置参数有：

- 网络设备端口地址：(port address)。
- 中断号：(irq)。
- 网络设备名：(name)。
- 设备数据缓冲区起始地址与结束地址：(mem_start、mem_end)。
- DMA 通道号：(dma)。
- 网络设备连接端口类型：(port)。

在内核启动时，可以在 boot loader 中用以下格式通过命令行参数向网络子系统传递网络设备的硬件配置参数：

```
netdev=5,0x260,eth0  netdev=15,0x300,eth1
```

同一个命令行参数的关键字在系统启动的命令行参数字符串中可以出现多次，以上命令行参数给系统中的两个网络设备提供了硬件配置参数。

- eth0：中断号为 5，端口地址为 0x260。
- eth1：中断号为 15，端口地址为 0x300。

2．处理网络子系统命令行参数的函数

在 net/core/dev.c 文件中实现的 netdev_boot_set 函数是网络子系统命令行参数的解析函数。

```
int __init netdev_boot_setup(char *str)
{
  int ints[5];
  struct ifmap map;
  str = get_options(str, ARRAY_SIZE(ints), ints);    //函数从系统中获取参数
  if (!str || !*str)
       return 0;
  memset(&map, 0, sizeof(map));
  if (ints[0] > 0)                                   //如果命令行参数有赋值
       map.irq = ints[1];                            //将命令行参数值存放在
  if (ints[0] > 1)                                   //ifmap 数据结构中
       map.base_addr = ints[2];
  if (ints[0] > 2)
       map.mem_start = ints[3];
  if (ints[0] > 3)
       map.mem_end = ints[4];
  return netdev_boot_setup_add(str, &map);           //将所有的命令行参数值存放在
}                                                    //netdev_boot_setup 结构数组中
__setup("netdev=", netdev_boot_setup);
```

3. 网络子系统命令行参数的保存

如果在系统启动时，用户给"netdev="命令行参数赋了值，则其值由函数 netdev_boot_setup 处理。netdev_boot_setup 处理函数的功能非常简单，它查看输入的字符串，将字符串后跟着的值解析到 ifmap 结构的数据域中。

```
struct ifmap
{
  unsigned long mem_start;
  unsigned long mem_end;
  unsigned short base_addr;
  unsigned char irq;
  unsigned char dma;
  unsigned char port;
};
```

最后，解析函数 netdev_boot_setup 调用 netdev_boot_setup_add，将解析好的所有命令行参数以设备名为索引保存在 struct netdev_boot_setup 数据结构类型的数组 dev_boot_setup[NETDEV_BOOT_SETUP_MAX]中。系统启动时，网络子系统可配置的最大设备数为 8。

```
struct netdev_boot_setup {
  char name[IFNAMSIZ];
  struct ifmap map;
};
static struct netdev_boot_setup dev_boot_setup[NETDEV_BOOT_SETUP_MAX];
#define NETDEV_BOOT_SETUP_MAX  8
```

4．网络子系统命令行参数的作用

在内核启动的后期，网络子系统代码会通过函数 netdev_boot_setup_check 来查看数组 dev_boot_setup[NETDEV_BOOT_SETUP_MAX]，看用户是否在系统启动时给出了网络设备的硬件配置参数。如果数组不为空，函数 netdev_boot_setup_check 就将这些参数值填入到网络设备的 struct net_device dev 实例的数据域中。

```
int netdev_boot_setup_check(struct net_device *dev)
{
  struct netdev_boot_setup *s = dev_boot_setup;
  int i;
  //将 dev_boot_setup 数据中的网络设备名与网络设备实例 dev 中的设备名比较，如命令行参数中用户为该设备提供了配件
  //配置参数，则命令行参数初始化网络设备
for (i = 0; i < NETDEV_BOOT_SETUP_MAX; i++) {
        if (s[i].name[0] != '\0' && s[i].name[0] != ' ' &&
           !strcmp(dev->name, 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;
}
```

4.2 内核启动过程

Linux 系统从上电到进入初始化要经过如图 4-2 所示的几个步骤。

bootloader 中的指令是系统上电后执行的第一条指令，完成系统基本的硬件初始化，然

后将控制权交给 Linux 内核，并将命令行参数传给内核，进入 secondloader。这部分代码与 CPU 体系结构密切相关，由汇编语言实现。例如，32 位 PowerPC 系列的 CPU 的 secondloader 代码在 Linux 源码目录结构的 arch/powerpc\kernel\head_32.S 文件中。secondloader 的主要任务是完成 CPU 硬件的初始化，特别是初始化 MMU 单元（Memory Management Unit，内存管理单元），建立内存页面管理结构（页表）；实现物理地址到虚拟地址的转换；随后跳入 start_kernel 执行，开始 Linux 内核启动过程。例如在 arch/powerpc/kernel/head_32.S 文件中我们可以看到以下代码：

```
…
li    r4,MSR_KERNEL
FIX_SRR1(r4,r5)
lis   r3,start_kernel@h
ori   r3,r3,start_kernel@l
mtspr SPRN_SRR0,r3
mtspr SPRN_SRR1,r4
SYNC
RFI
```

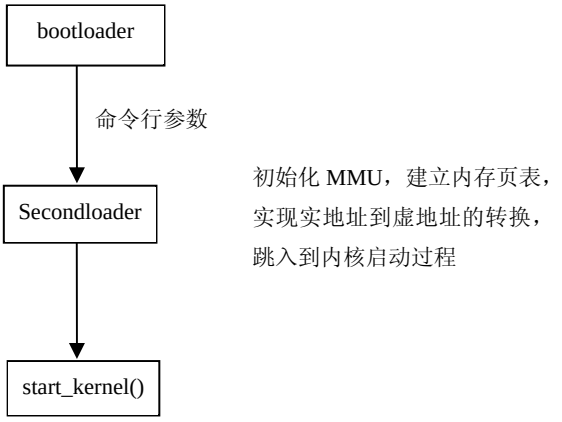


图 4-2　Linux 内核启动过程

内核启动过程从 init/main.c 文件中的 start_kernel 函数开始。start_kernel 函数的首要任务是建立内核运行的基线，即基础子系统的初始化，包括：

- 建立多处理 CPU 体系结构的运行环境。
- 进程调度。
- 初始化系统时钟。
- 初始化中断系统。
- 初始化内存。
- 解析命令行参数。
- 建立文件系统等。

在 start_kernel 函数结束处调用 rest_init 函数创建第一个内核进程，执行 kernel_init 函数，kernel_init 函数执行 do_basic_setup 函数转而调用 do_initcalls 函数，按照常规子系统注册初始

化函数的优先级顺序调用它们提供的初始化函数，完成整个内核的启动、初始化过程。

内核的初始化序列如图 4-3 所示，系统共享资源的基础子系统初始化在调用 do_initcalls()函数之前完成。

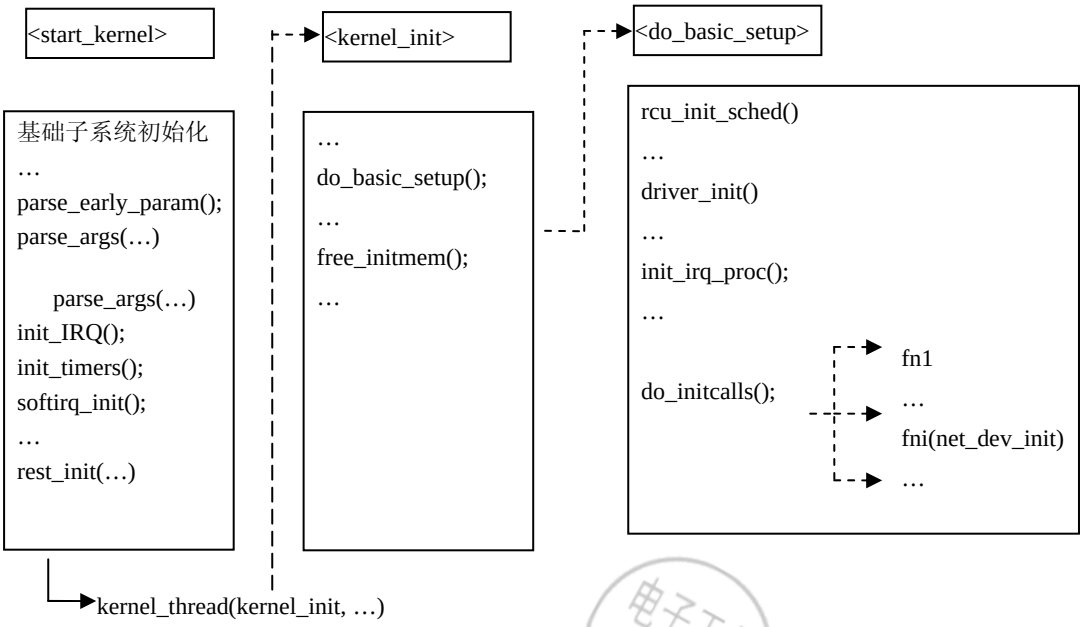


图 4-3　内核的初始化过程

```
asmlinkage void init start_kernel(void)
{
   char * command_line;
   extern struct kernel_param_start_param[],_stop_param[];

   smp_setup_processor_id();
…
sched_init()
…
parse_early_param();                    //分两次解析命令行参数
   parse_args("Booting kernel", static_command_line, _start_param,
            __stop_param -_start_param,
            &unknown_bootoption);
…
rest_init()
……


static noinline void _init_refok_rest_init(void)
   _releases(kernel_lock)
{ …
                                        //创建第一个内核进程执行，内核各子系统初始化过程
kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);


static int_init kernel_init(void * unused)
{
   …
   do_basic_setup();
   …


static void_init do_basic_setup(void)
{
   …
   do_initcalls();
}
```

4.2.1　用 do_initcall 函数完成的初始化

Linux 内核第二部分的初始化（其他子系统的初始化）由 do_initcalls 函数完成（在 do_basic_setup 函数的底部）。在这一部分中，对各子系统的初始化函数的调用顺序按照内核中各子系统的角色和优先级来划分。内核一个接一个地执行这些子系统的初始化函数，从优先级最高的类型（由宏 core_initcall 注册的初始化函数）开始。所有这部分初始化程序存放在内存特定的段中，由.initcallN.init 来标记。优先级不同的子系统初始化程序使用不同的宏来标记，注册到内核中，这些宏的格式为 xxx_initcall，这样在注册后，内核将各优先级的初始化函数放入相应的 .initcallN.init 段中。所有的初始程序所在内存段都在以__initcall_start 为起始地址，以__initcall_end 为结束地址内存范围内。内核启动结束后，这部分内存就可以释放返回给系统使用。由初始代码使用的存储区分布如图 4-4 所示。

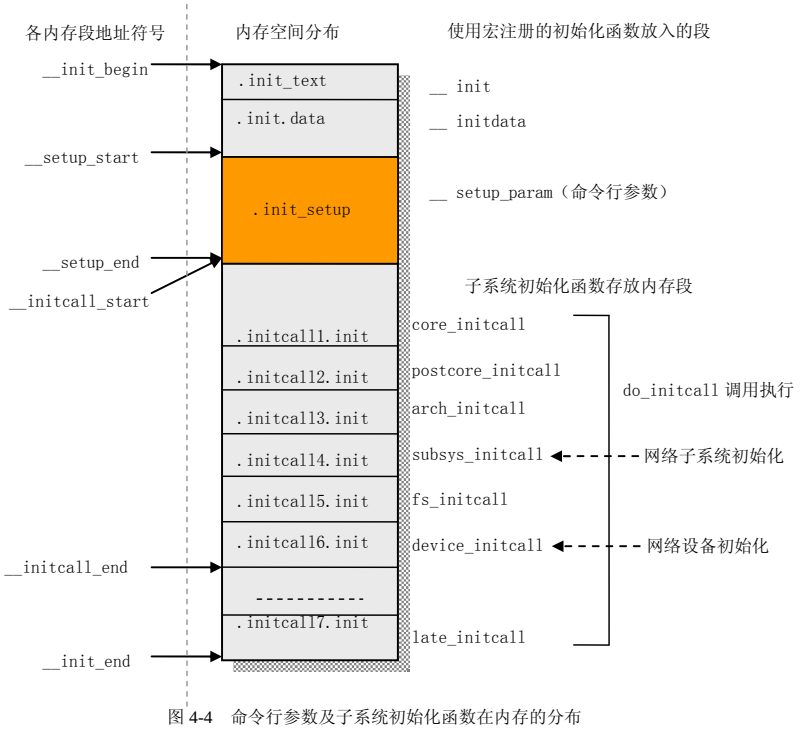


图 4-4　命令行参数及子系统初始化函数在内存的分布

左边是每一个内存段起始和结束指针名，右边是把数据代码和数据放入相应内存段中的宏，中间是内存段的名字。

在 do_initcalls 函数中，就逐一地执行__initcall_start 与__initcall_end 之间的函数。

```
extern initcall_t __initcall_start[], __initcall_end[], __early_initcall_end[];
static void __init do_initcalls(void)
{
  initcall_t *call;

  for (call = __early_initcall_end; call < __initcall_end; call++)
       do_one_initcall(*call);
  flush_scheduled_work();
}
```

到此，系统初始化已完成，CPU 的各子系统都建立起来，内存和进程管理开始工作了。内核初始化完成后，在 kernel_init 函数最后调用 init_post 函数，中间会调用 free_initmem 函数将内核初始化时使用的命令参数、子系统初始化函数占用的内存空间释放返回给系统使用。

4.2.2 标记初始化函数的宏

Linux 内核中定义了大量的宏来标记和注册应把代码和数据放在存储区的什么段中，这些宏主要定义在 include/linux/init.h 文件中，在内核启动过程中会用到表 4-1 和表 4-2 列出的宏。

表 4-1 注册初始化代码的宏

宏	宏用于定义的例程
_ _init	内核启动时的初始化例程；标识启动阶段结束后不再需要的例程
_ _ exit	与_ _init 成对，用于 kernel 组件退出的例程。常用于标记模块清除函数 module_exit
core_initcall postcore_initcall arch_initcall subsys_initcall fs_initcall device_initcall late_initcall	一组宏，用于注册系统启动时由 do_initcall 调用的初始化函数，宏在表中所列的顺序规定了初始化函数的优先级顺序
__ initcall	旧的宏，定义为 device_initcall 的别名
_ _ exitcall	一次性退出函数，当 kernel 组件卸载时调用。目前它只用于标记 module_exit 例程

表 4-2 定义数据结构的宏

宏	标记的数据结构
_ _ initdata	只在启动时使用的数据结构
_ _exitdata	由_ _exitcall 标记的程序使用的数据结构。如果由_ _exitcall 标记的代码不再需要时，由_ _exitdata 标记的数据结构也不再需要，它们占用的内存都可以释放

对标记内核初始化函数与初始化函数使用的数据结构的宏，需要说明以下几点。

- 大多数宏都是成对出现的，其中一个（或一组）完成初始化功能，而与其成对的另一个宏完成清除功能。例如，__exit 与__init 成对，__exitcall 与__initcall 成对。
- 宏描述了两个特性，一个是代码什么时候执行（如__initcall、__exitcall），一个是说明代码或数据结构在内存中的存放位置（如__init、__exit）。
- 同一个例程可以由多个宏标记。例如以下代码说明 pci_proc_init 是在系统启动时被执行（__initcall）的，它一旦被执行后，其占用的内存空间就可以释放（__init）。

```
static int __init pci_proc_init(void)
{
…
}

__initcall(pci_proc_init)
```

4.2.3　网络子系统初始化

网络子系统的初始化例程是 net_dev_init，它是使网络代码能正常工作的重要函数，其功能包括初始化流量控制和为每个 CPU 建立传送队列。net_dev_init 函数在系统启动时由 do_initcall 调用执行，完成网络子系统的初始化。网络子系统不能编译成模块的形式，只能直接编成 kernel 的一个组件。

net_dev_init 在系统启动期间会遍历整个网络设备链表（dev_list），将初始化失败的设备数据结构实例从设备链表中移走（通常都是没安装的硬件），将有效设备数据结构实例留在列表中。

```
static int __init net_dev_init (void)                          //net/core/dev.c
{
  int i, rc = -ENOMEM;
  //不是内核启动阶段不调用该函数
  BUG_ON(!dev_boot_phase);
  //初始化网络设备在/proc 文件系统下的入口
  if (dev_proc_init())
        goto out;
  …
  //注册需处理来自网络设备数据的上层协议实例的处理函数
  INIT_LIST_HEAD(&ptype_all);
  for (i = 0; i < PTYPE_HASH_SIZE; i++)
        INIT_LIST_HEAD(&ptype_base[i]);
  …
  //初始化每个 CPU 的数据包输入/输出队列
for_each_possible_cpu(i) {
        struct softnet_data *queue;
        queue = &per_cpu(softnet_data, i);
        skb_queue_head_init(&queue->input_pkt_queue);          //初始化输入数据包队列
        queue->completion_queue = NULL;                        //完成队列为空
        INIT_LIST_HEAD(&queue->poll_list);                     //建立轮询设备队列
  …
  }
  dev_boot_phase = 0;
  //初始化 loopback 网络设备，loopback 设备比较特殊，如果网络名字空间呈现了任何网络设备，loopbakc 设备就必须呈现
  if (register_pernet_device(&loopback_net_ops))
        goto out;
  //遍历网络设备链表，保留有效设备
if (register_pernet_device(&default_device_ops))
        goto out;
  //注册网络子系统的接收/发送网络数据包软件中断处理函数
  open_softirq(NET_TX_SOFTIRQ, net_tx_action);
  open_softirq(NET_RX_SOFTIRQ, net_rx_action);
  //注册接收 CPU 事件通知的处理函数到 CPU 事件通知链中
  hotcpu_notifier(dev_cpu_callback, 0);
  //初始化路由表
  dst_init();
  //初始化组传送设备
  dev_mcast_init();
  rc = 0;
out:
  return rc;
}
  //用 subsys_initcall 宏将网络子系统初始化函数注册放入。initcall 内存段中
subsys_initcall(net_dev_init);
```

归纳起来，网络子系统的初始化函数的流程与完成的任务为：

1．创建在/proc 文件系统下的入口

首先如果内核编译为支持 /proc 文件系统（通常默认的配置为支持/proc 文件系统）时，在/proc 文件系统目录下会加入一些文件，来描述网络子系统参数配置，用户程序可以读取这些文件来获取网络子系统的配置信息。这些文件会由函数 dev_proc_init 和 dev_mcast_init 加入到/proc 文件系统中。

其次，在 / sys 文件系统下注册 net 类，它在 / sys 文件系统下创建目录/sys/class/net，在该目录下你会看到每个注册了的网络设备都有一个子目录，这些目录下包含了一系列关于网络设备配置参数的文件。

2．流量管理初始化

网络子系统的流量管理初始化过程包括：为每个 CPU 建立网络数据包的接收/发送队列；向系统注册接收网络设备数据包的上层协议栈的接收处理函数，这些协议处理函数在向量表 ptype_base 中进行管理；向系统注册网络子系统中接收/发送数据包的两个软件中断处理函数（NET_TX_SOFTIRQ，NET_RX_SOFTIRQ）。

3．设备与事件初始化

接着网络子系统初始化函数 net_dev_init 遍历网络设备实例列表，将初始化失败的设备从设备链表中移走，保留有效网络设备的实例在设备管理链表中。

向 CPU 热插拔的事件通知链注册网络子系统的回调函数，一旦 CPU 热插拔事件产生，该回调函数 dev_cpu_callback 处理 CPU 事件，当前唯一的事件就是 CPU 暂停，当接到该事件的通知时，相应 CPU 的网络数据包接收队列停止接收网络包，并通知数据链路层接收函数 netif_rx。

4．初始化路由与其他初始化过程

独立于协议栈的目的路由缓冲存储器 cache(DST)，由 dst_init 函数初始化。

最后宏 subsys_initcall 将 net_dev_init 函数放入内核的系统初始化内存段中。在内核启动时，do_initcall 就会顺序地调用初始内存段中的子系统初始化例程完成初始化过程。而 subsys_initcall 也保证了网络子系统会在任何设备驱动程序注册其设备之前执行 net_dev_init 函数（见表 4-1 中各子系统初始化函数的优先级顺序）。

全局变量 dev_boot_phase 是一个布尔类型变量，用于标记 net_dev_init 是否已被执行过。其初值为 1（即 net_dev_init 还没被执行过），在 net_dev_init 中被清零。

网络子系统初始化函数为网络组件建立了基本的执行条件，比如网络数据包队列，软件中断处理函数，CPU 事件处理函数，注册协议栈处理函数等网络子系统的初始化不仅服务于网络设备，而且为整个协议栈的正常工作建立基础，当然，以后在分析 TCP/IP 协议栈各层协议的工作原理时我们会看到，在网络子系统初始化的基础上，各层协议实例自身也要完成初始化任务。

4.2.4 网络设备的初始化

内核的代码既可以静态地连接（在编译时就连接）到整个内核的目标代码中，也可以

作为模块在系统运行时动态地加载到内核。但并不是所有的内核组件都适于编译成模块，一般设备驱动程序或对内核功能扩展的代码常编译成模块。

每个模块必须提供两个特殊的函数：init_module 和 cleanup_module。第一个函数是在装载模块时的初始化函数，第二个函数是在把模块从内核中卸载时调用的模块清除函数，释放分配给模块的所有资源。

内核提供了两个宏定义：module_init 和 module_exit，允许模块命名自己的模块初始化函数和模块清除函数，通过宏 module_init 和 module_exit 标记了的函数就成为 init_module 和 cleanup_module 的别名。例如，3c59x 的以太网驱动程序中（drivers/net/3c59x.c）命名的模块初始化函数与模块清除函数为：

```
module_init(vortex_init);                    //drivers/net/3c59x.c
module_exit(vortex_cleanup);
```

大多数模块都使用这两个宏来声明自己的模块初始化函数，但也有少部分模块仍使用默认的初始化函数定义：init_module 和 cleanup_module。例如以下 cs89x0 的驱动程序（driver/net/cs89x0.c）模块初始化函数与清除函数分别为：

```
int __init init_module(void)                 // drivers/net/cs89x0.c
void __exit cleanup_module(void)
```

在本章的其余部分，将使用 mudule_init 和 module_exit 来表示模块的初始化和清除函数。我们会介绍这两个宏是如何定义的，以及它们的定义如何按照内核的配置而发生变化。

一个网络设备的驱动程序可以直接编译成内核的一个组件，也可以编译成模块，在系统运行期间插入内核。两种模式都是通过调用宏 module_init 和 module_exit 定义的初始化函数及清除函数来完成网络设备的初始化（如果静态编译到内核中就在系统启动时执行，如果编译成模块就在运行时装载模块时执行）和卸载。

1. 静态编译到内核

当网络设备驱动程序是静态连接到内核中时，以下在 include/linux/init.h 文件中对宏 module_init 和 module_exit 的声明保证了由 module_init 定义的设备初始化函数会在系统启动时执行。

```
…                                    //include/linux/init.h
#ifndef MODULE
#define pure_initcall(fn)            __define_initcall("0",fn,0)
…
#define subsys_initcall(fn)          __define_initcall("4",fn,4)
…
#define device_initcall(fn)          __define_initcall("6",fn,6)
….
#define __initcall(fn)                 device_initcall(fn)
…
#define module_init(x)               __initcall(x)
#define module_exit(x)               __exitcall(x)
#else
```

所以当设备驱动程序静态编译到内核时（即不为模块），宏 module_init 定义为__initcall 的别名，module_init 描述的函数就分类为内核启动时的初始化例程。在上面的程序代码中我们还可以看到，__initcall 宏是 device_initcall 宏的别名，网络设备的初始化例程由 do_initcall

调用 device_initcall 宏指定的优先级初始化段中的函数来执行。网络子系统的初始化函数 net_dev_init 是由宏 subsys_initcall 声明的，subsys_initcall 宏指定的初始化函数的调用顺序在 device_initcall 宏之前，这就保证了在网络子系统初始化完成前不会有任何网络设备注册。

2．设备驱动程序编译为模块

网络设备驱动程序编译成模块时，module_init 宏声明的函数会在系统运行期间用户发出 insmod 或 modprobe 命令时被调用执行，以初始化网络设备。module_exit 宏声明的函数会在用户发出命令 rmmod/modprobe -r 时被调用来卸载网络设备驱动程序模块。

4.3 网络设备的注册和 struct net_device 数据结构实例的初始化

在上一节中我们介绍了网络子系统在何时被初始化；网络子系统初始化函数 net_dev_init 完成的功能；网络设备的初始化函数在什么时候执行。在这一节中我们主要介绍网络设备初始化函数完成的任务和设备的注册过程。

在网络设备硬件可以被内核识别使用之前，描述网络设备的 struct net_device 数据结构实例必须被创建、初始化，并加入到内核的网络设备数据库中，然后配置设备参数、激活设备，允许设备开始收发网络数据。在这里需要区分注册/注销设备与允许/禁止设备的概念。

1. 注册/注销

如果排除驱动程序以模块的形式装载到内核的方式，这个过程独立于用户，由内核驱动，设备仅仅是已注册，但还不能开始操作。

2. 允许/禁止

这个操作是与用户互动的，一个设备被内核注册后，用户可以通过命令看到它，配置它，允许它开始操作。

一个网络设备在什么时候注册到内核中？是在执行网络设备驱动程序提供的模块初始化函数（由宏 module_init 指定的函数或 init_module 函数本身）的过程中注册到系统的，也即在内核启动过程中或系统运行时间装载模块时注册。当卸载网络设备驱动程序时（只有当驱动程序编译成模块时才能卸载），所有与该网络设备驱动程序相关的网络设备都从系统中注销。本节我们就讲解网络设备初始化、设备注册/注销的实现过程。

4.3.1 初始化函数的任务

无论什么初始化函数（网络子系统的或网络设备的初始化函数），主要的任务就是要让被初始化的对象能被 Linux 内核识别，处于正常工作状态。因此，网络设备驱动程序的初始化任务就需要完成以下几方面的任务。

- 创建设备的代码实例，即 struct net_device 数据结构类型变量。
- 初始化网络设备实例，即 struct net_device 数据结构类型变量的数据域。网络设

备实例的数据域，有的由内核代码赋值，有的由网络设备驱动程序赋值。

- 将网络设备实例注册到内核中。

1．为设备创建 net_device 数据结构实例

网络设备在内核代码中由 struct net_device 数据结构实例表示，创建 net_device 实例首先需要向系统申请内存空间，将变量驻留在申请到的内存空间上，这个功能由 alloc_netdev_mq 函数完成，函数原型在 net/core/dev.c 文件中。下面我们对该函数完成的功能和实现流程做进一步的分析。

```
struct net_device *alloc_netdev_mq(int sizeof_priv, const char *name,
        void (*setup)(struct net_device *), unsigned int queue_count)
```

（1）输入参数

该函数需要 4 个参数。

- sizeof_priv：私有数据结构需占用的内存空间。struct net_device 数据结构可以由设备驱动程序通过私有数据结构块来扩展，其中存放设备驱动程序的参数决定了私有数据结构的大小。
- name: 设备名。name 参数可以给出网络设备名的一部分，内核会根据一定的机制为设备分配一个在内核中唯一的设备名。
- *setup：初始化例程。该例程用于初始化 struct net_device 数据结构的部分数据域。
- queue_count：网络设备的发送队列数。

（2）函数返回值

alloc_netdev_mq 函数执行成功后会返回指向分配好的 net_device 结构实例的指针，若分配不成功则返回 NULL 指针。

（3）网络设备实例需要的内存空间

由 alloc_netdev_mq 为网络设备分配的空间由三个部分组成： net_device 数据结构所需的内存空间+私有数据结构占用的内存空间+设备发送队列所需内存空间。

（4）为网络设备分配设备名

每个网络设备根据其类型内核都会分配一个网络设备名，以字符串开头后跟一个数字为网络设备名。前缀字符串描述了网络设备的类型，而数字是按同类网络设备注册到系统中的顺序产生的序列号。以以太网设备为例，以太网设备按其在内核中的注册顺序，设备名分别为：eth0、eth1 等。一个设备注册的顺序不同，其设备名可能不同。当设备名以 name%d（eth%d）的形式传给 alloc_netdev_mq 函数时，该函数会调用 dev_alloc_name 函数完成设备名的分配，将%d 替换成该类网络设备中第一个没有被占用的序列号。

（5）网络设备发送队列的建立

alloc_netdev_mq 函数根据输入参数，设备发送队列数 queue_count，为设备分配发送队列数据结构 netdev_queue 所需的内存空间。随后初始化 struct net_device 数据结构的发送队列、发送队列数量、实际可用发送队列数等数据域（dev->tx，dev->num_tx_queue，dev->real_num_tx_queues）。

内核提供了一系列 alloc_netdev_mq 函数的包装函数为不同类型的网络设备创建网络设备实例。alloc_netdev_mq 的包装函数的主要作用是向 alloc_netdev_mq 函数传递正确的参

数。由包装函数传递的参数主要是设备名 name 和初始化例程 setup。

例如，以太网网络设备对 alloc_netdev_mq 函数的包装函数是 alloc_etherdev_mq。alloc_etherdev_mq 向 alloc_netdev_mq 函数传送格式为 eth%d 的设备名，ether_setup 为初始例程的参数，ether_setup 例程是内核为以太网类设备统一初始化 net_device 部分数据域而提供的函数。

```
struct net_device *alloc_etherdev_mq(int sizeof_priv, unsigned ing queue_count)
{
  return alloc_netdev_mq(sizeof_priv, “eth%d”, ether_setup,queue_count);
}
```

最后 alloc_netdev_mq 函数调用传送过来的 setup 函数指针指向的设备初始化函数，完成内核代码对 struct net_device 数据结构部分数据域的初始化。

2. 内核代码 xxx_setup 对网络设备实例的初始化

大多数常规类型的网络设备都有一个统一的 xxx_setup 例程来初始化 struct net_device 数据结构实例的某些数据域（参数和函数指针），这些数据域对同类型的网络设备而言为通用数据域，无论网络设备的生产厂商如何，它们的值都相同。如所有以太网卡的通用初始化函数是 ether_setup，ether_setup 初始化所有对以太网类网络设备而言通用的数据域，建立以太网的网络设备实例。表 4-3 列出了各种类型网络设备对 alloc_netdev_mq 函数的包装函数，各种 alloc_xxxdev_mq 包装函数传给 alloc_netdev_mq 函数的设备名，以及正确的 xxx_setup 函数指针。

表 4-3 alloc_netdev_mq 函数的包装函数

网络设备类型	包装函数名	包装函数定义
以太网	alloc_etherdev_mq	return alloc_netdev_mq(sizeof_priv，"eth%d"，ether_setup)；
光纤分布式数据接口（FDDI）	alloc_fddidev_mq	return alloc_netdev_mq(sizeof_priv，"fddi%d"，fddi_setup)；
高性能并行接口（HIPPI）	alloc_hippi_dev_mq	return alloc_netdev_mq(sizeof_priv，"hip%d"，hippi_setup)；
令牌环（Token Ring）	alloc_trdev_mq	return alloc_netdev_mq(sizeof_priv，"tr%d"，tr_setup)；
光纤通道	alloc_fcdev_mq	return alloc_netdev_mq(sizeof_priv，"fc%d"，fc_setup)；
IrDa	alloc_irdadev_mq	return alloc_netdev_mq(sizeof_priv，"irda%d"，irda_device_setup)；

以下是以太网的 ether_setup 例程初始化的网络设备实例的部分数据域示例：

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

从以上代码可以看到，ether_setup 例程初始化的是所有以太网卡共享的数据域和函数指针，如设备最大传送单元 MTU：（1500），数据链路层的广播地址（FF：FF：FF：FF：FF：FF），发送队列的长度：（1000 个数据包）等。

有时 struct net_device 数据结构中某些数据域在以上过程中（xxx_setup 和网络设备驱

动程序的初始化例程 xxx_probe）都没有初始化，是因为这些数据域对这种类型的网络设备而言无意义，所以保留它们为空。

在访问 struct net_device 数据结构的数据域时，特别是函数指针，为了保证操作正确，内核在调用函数之前总要确认可选函数指针是否被初始化了，否则可能会访问错误的内存区域，造成系统崩溃。例如，以下代码片段为在 register_netdevice 函数中调用 init 函数时所做正确性检查：

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

3. 驱动程序初始化 struct net_device 数据结构的其他数据域

由内核代码 xxx_setup 初始化 struct net_device 数据结构的通用数据域后，接着由网络设备驱动程序初始化 struct net_device 数据结构的其他部分，最后调用 register_netdev 注册设备。

在网络设备驱动程序中，初始化 net_device 数据结构的任务通常由 xxx_probe 命名的函数（或 module_init 声明的函数本身）来完成。在 xxx_probe 函数中完成的初始化是对该网络设备而言特定的数据域的初始化，与具体的网络设备硬件特性相关，例如，xxx_probe 函数初始化网络设备向系统申请的中断号 dev->irq 数据域（通过 irq_request 函数获取）、I/O 端口地址 dev->base_addr、网络设备 DMA 通道号 dev->dma 等；建立驱动程序的函数指针，如 open、stop、hard_start_xmit 等虚函数的实例化。表 4-4 列出了由 ether_setup 函数与由 xxx_probe 函数初始化的 net_device 数据结构域的对比。

表 4-4　由内核代码与设备驱动程序初始化的 net_device 数据域的对比

初始化例程	net_device 的数据域	net_device 的函数指针
xxx_setup 函数	type hard_header_len mtu addr_len tx_queue_len flags broadcast	header_ops change_mtu set_mac_address validate_addr
设备驱动程序的 xxx_probe 函数	base_addr irq if_port features priv	open stop hard_start_xmit tx_timeout watchdog_timeo get_stats get_wireless_stats set_multicast_list do_ioctl init uninit poll ethtool_ops

网络设备从创建设备实例到设备注册到内核的过程如图 4-5 所示。开始由函数 alloc_etherdev_mq 来分配 struct net_device 数据结构所需的内存，实际的分配在 kzalloc 函数中完成。kzalloc 函数分配 struct net_device 和私有数据结构的内存空间，kcalloc 函数分配发

送队列的数据结构空间。接着 setup 例程和网络设备驱动程序初始化 struct net_device 数据结构的部分数据域，最后调用 register_netdev 例程注册设备。

有的驱动程序会调用 netdev_boot_setup_check 来查看在系统启动时用户是否将配置参数传递给内核。

图中还给出了注销一个设备的简单过程。注销一个设备总是包含对函数 unregister_netdevice 和 free_netdev 的调用。对 free_netdev 有时是直接调用，有时是通过 dev->destructor 函数来调用的，在下一节介绍网络设备的注册和注销时我们可以看到，设备驱动程序还需要释放所有设备占用的资源。

总结起来，struct net_device 数据结构是一个非常庞大的数据结构，它的数据域的初始化分散在几个不同的函数中进行，每个函数负责对其部分不同数据域（或函数指针）进行赋值。

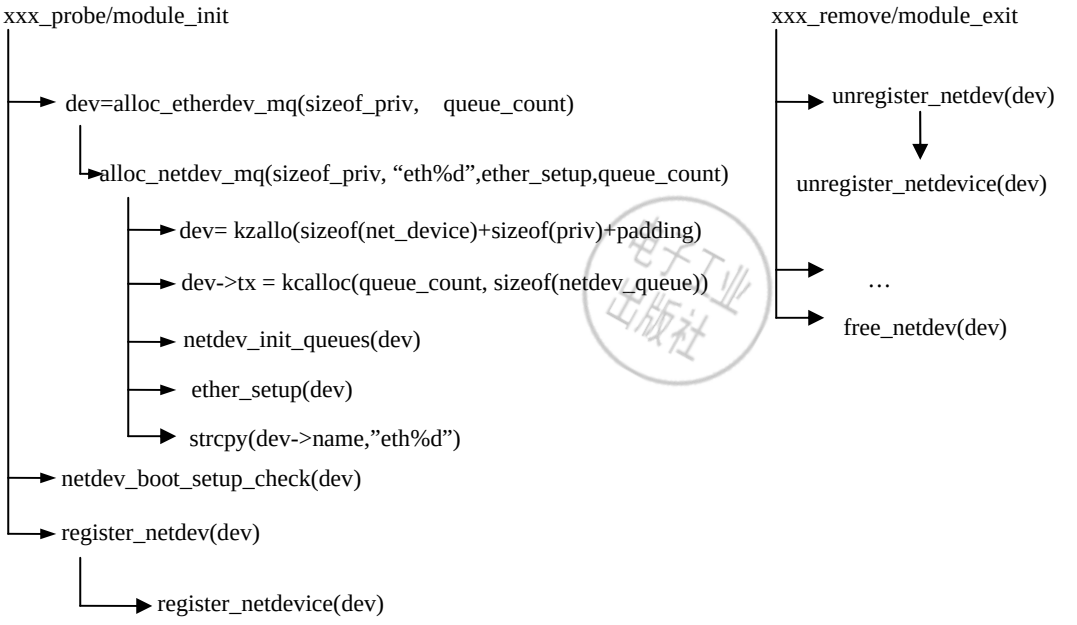


图 4-5　创建网络设备实例与初始化的过程

4.3.2　网络设备的注册和注销

网络设备的注册和注销分别由函数 register_netdev 和 unregister_netdev 函数完成。这两个函数是包装函数，它们获取防止并发访问的锁，然后调用 register_netdevice 和 unregister_netdevice。在图 4-5 中我们已经简单给出了这些函数，图 4-6 描述了注册网络设备实例时需要设置的状态和状态变化，也给出了在注册过程中还有哪些例程被调用了。

1．网络设备的注册和注销概述

网络设备实例需经过注册才能被内核识别，内核需要了解网络设备实例注册是否成功。

如果注册成功，设备就加入到了内核管理网络设备的数据库中，协议栈即可使用网络设备提供的服务；如果注册不成功，内核需要清除网络设备实例分配时占用的所有系统资源。网络设备注册过程需要维护一系列的状态来反映当前设备注册进程。网络设备的撤销与注册类似，也需要维护相应的状态。

- 注册过程中的状态变化可以是 NETREG_UNINITIALIZED 和 NETREG_REGISTERED 之间的中间状态，如图 4-6 所示。这些过程由 netdev_run_todo 函数来处理，我们会在事件通知一节中介绍。
- 网络设备驱动程序可以使用 struct net_device 数据结构的两个虚函数 init 和 uninit 在注册和撤销网络设备时来初始化和清除私有数据，它们主要由虚拟设备使用。

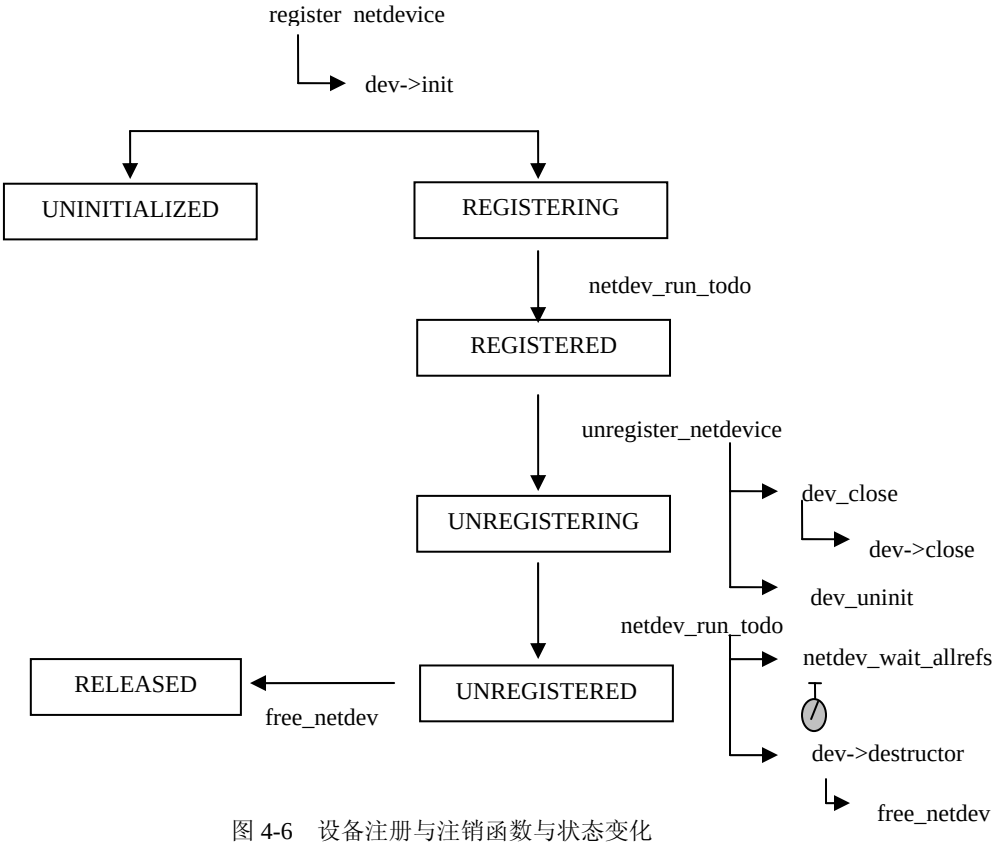


图 4-6　设备注册与注销函数与状态变化

- 只有在所有引用 struct net_device 数据结构实例的进程都释放了对设备的引用时，设备才最后注销，因此 netdev_wait_allrefs 会等条件满足时才会返回。
- 注册设备和注销设备都由 netdev_run_todo 最后完成。

注册网络设备的 register_netdevice 函数只完成部分设备注册功能，余下的部分由 netdev_run_todo 来做。如果只从代码本身来看，并不容易看出这个过程是如何进行的，我们借助于图 4-7 来看看其工作过程。

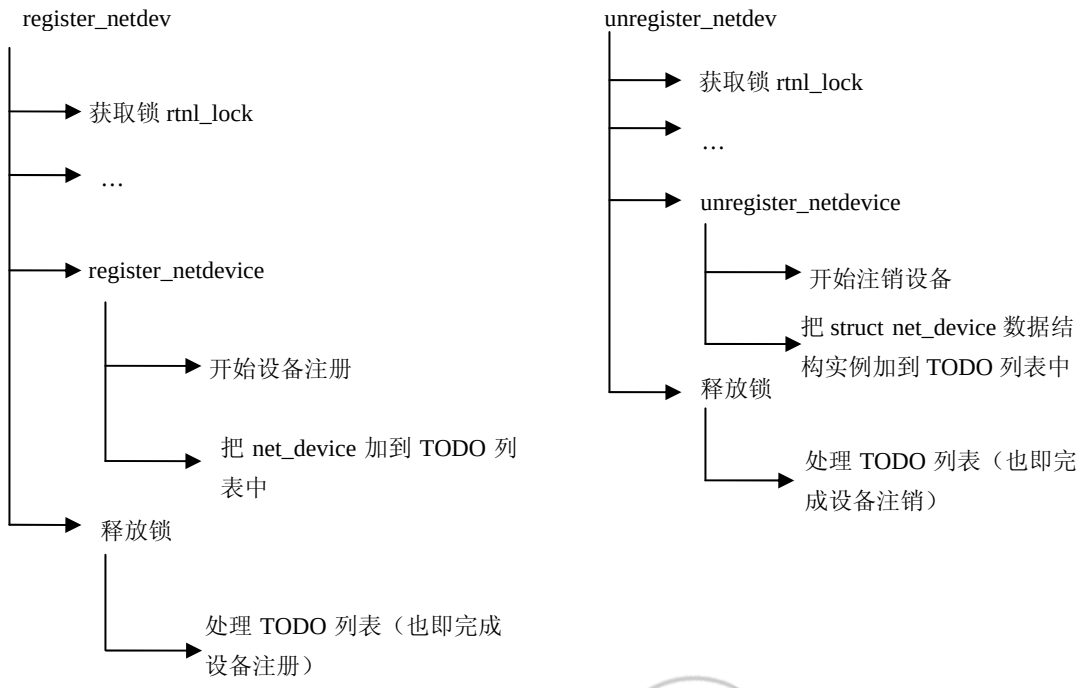


图 4-7　网络设备注册与注销过程

在注册网络设备时，要访问 struct net_device 数据结构实例的数据域，因为所有网络设备对应的 struct net_device 数据结构实例都保存在一个全局链表中。在对称多处理器环境中，为了防止并发访问造成的错误，修改 struct net_device 数据结构实例首先需要通过 rtnl_lock 获取 Routing Netlink semaphore 锁，register_netdevice 的包装函数 register_netdev 在最开始处就去获取锁，在结束处释放锁。

一旦 register_netdevice 完成了它的处理过程后，就把网络设备对应的 struct net_device 数据结构实例用 net_set_todo 加到 net_todo_list（由 dev->todo_list 指针指向）列表中，这个列表包含了所有需要完成注册（或注销）的设备。net_todo_list 列表不由独立的内核线程或周期性函数来处理，它会在 register_netdev 函数释放锁时间接被调用。

因此，rtnl_unlock 函数不仅释放锁，还会调用函数 netdev_run_todo。netdev_run_todo 遍历 net_todo_list 列表，完成所有网络设备 net_device 数据结构实例的注册（或注销）。任何时刻只有一个 CPU 可以运行 net_run_todo 函数， net_todo_run_mutex 保证不会有多个 CPU 同时访问该函数。注销一个设备的过程与上述过程完全相同。

注意，因为网络设备的注册和注销任务由 netdev_run_todo 函数在处理，netdev_run_todo 函数没有持有任何锁，所以 netdev_run_todo 函数可以安全地处于休眠状态，并保持 semaphore 有效。

2．网络设备的注册

网络设备注册过程从在网络设备实例初始例程（xxx_probe/module_init）中调用 register_netdevice 函数开始。如前所述，设备注册函数 register_netdevice 只是开始一个设备

的注册，它调用 net_set_todo，后者最终调用 netdev_run_todo 来完成设备的注册任务。对照定义在 net/core/dev.c 文件中的 register_netdevice 函数源代码，可以看到 register_netdevice 的主要任务如下。

- 初始化网络设备的 struct net_device 数据结构实例的某些数据域。
- 如果 net-device 实例的 init 虚拟函数不为空，则调用 init 函数来初始化设备驱动程序的私有数据结构。

```
if (dev->netdev_ops->ndo_init) {
        ret = dev->netdev_ops->ndo_init(dev);
```

- 用函数 dev_new_index 为设备分配一个唯一的标识符。标识符由计数器生成，每次当有新设备加入到系统中时，计数器的值加 1，计数器是一个 32 位的变量，所以 dev_new_index 要检查标识符是否已分配。
- 查看 struct net_device 数据结构实例的 features 标志是否为无效的组合。例如，Scather/Gather_DMA。如果没有硬件对传输层的校验和的支持，其功能就无效，因此应该禁止该功能。TSO 功能特性要求有 Scather/Gather-DMA 的支持，如后者无效，也应禁止 TSO 的功能设置。
- 设置 dev->state 的__LINK_STATE_PRESENT 位，标志设备有效（对系统而言可见也可以使用）。当从系统中拔掉一个热插拔的设备时，或系统的电源管理支持挂起模式，且设备挂起，__LINK_STATE_PRESENT 标志位会被清除。初始化这个标志不会触发任何活动。
- 用 dev_init_scheduler 函数初始化设备的队列策略。队列策略定义了发送数据包如何放入队列中，以及如何从队列中取出，它还定义了有多少数据包可以放入队列中。
- 用 list_netdevice 将网络设备的 struct net_device 数据结构实例加入到全局链表 dev_base_head 和两个哈希链表中。在加入到全局链表中时需检查设备名是否有重复，查看设备名是否为有效设备名（dev_valid_name）。
- 用事件通知链 netdev_chain 的 NETDEV_REGISTER 事件（在后面的章节中会介绍）通知所有需要获知网络设备已注册的子系统。

当调用 netdev_run_todo 来完成设备注册时，只需更新 dev->reg_state 状态，并将设备注册到/sysfs 文件系统中。

除了分配内存失败以外，网络设备注册失败的唯一原因是设备名无效或有重复的设备名，或者是由于某种原因 dev->init 调用失败。

3．网络设备的注销

（1）网络设备注销的任务

注销一个网络设备时，内核和网络设备的驱动程序需要取消所有在网络设备注册时所做的工作，除此之外，还要完成以下任务：

- 调用 dev_close 函数禁止网络设备操作。
- 释放所有为网络设备分配的资源（中断号、I/O 内存空间、I/O 端口地址等）。
- 从全局链表 dev_base_head 和两个哈希链表中移走网络设备的 struct net_device 数据结构实例。

- 所有持有网络设备 struct net_device 数据结构实例引用的进程，都释放了对网络设备的引用后，才最后释放网络设备的 net_device 数据结构实例、设备驱动程序的私有数据结构及其他与网络设备实例相连的存储块。struct net_device 数据结构实例由 free_netdev 函数释放。
- 清除网络设备加入到/proc 和/sys 文件系统中的所有文件。

struct net_device 数据结构中有 3 个函数指针指向的函数与注销设备过程相关。

- dev->stop：该函数指针由网络设备驱动程序初始化，指向一个本地函数，它由 net/core/dev.c 中的 dev_stop 函数在停止网络设备时调用。其功能通常为调用 netif_stop_queue 停止网络设备传送队列，释放硬件资源，停止由设备驱动程序使用的所有时钟计数器。
- dev->uninit：这个函数指针也是由设备驱动程序初始化，指向一个本地函数，有时（但很少）由虚拟设备初始化它，它主要负责递减对网络设备实例的引用计数。
- dev->destructor：当使用这个函数指针时，它通常初始化为指向 free_netdev 或设备释放函数 free_netdev 的一个包装函数。通常 destructor 都不初始化，只有一些虚拟设备使用它，大多数设备驱动程序都是在 unregister_netdevice 后直接调用 free_netdev。

（2）网络设备注销函数的流程

unregister_netdevice 函数接收一个参数，就是指向需要释放的网络设备数据结构 net_device 的指针。

```
int unregister_netdevice(struct net_device *dev)
```

由 unregister_netdevice 函数完成的其他任务还包括：

- 如果网络设备没有被禁止，它首先调用函数 dev_close 来禁止网络设备活动，使其处于不活动状态。
- 接着将网络设备的 struct net_device 数据结构实例从全局链表 dev_base_head 和两个哈希链表中移走，注意，这样还不足以禁止内核的其他子系统使用设备，它们可能仍持有网络设备的 struct net_device 数据结构实例的引用指针，这就是为什么 net_device 使用引用计数来跟踪有多少用户在使用网络设备的 struct net_device 数据结构实例的原因。
- 所有与这个设备相关的队列策略实例都由 dev_shutdown 取消。
- 网络子系统的事件通知链 netdev_chain 用 NETDEV_UNREGISTER 事件通知内核的其他组件网络设备已卸载。
- 用户空间必须要获取设备注销的通知，因为在一个系统中可以有多个网络设备用于访问 Internet，这个通知可以告诉用户一个网络设备卸载了，可以激活另一个网络设备。
- 任何与网络设备的 struct net_device 数据结构实例链接的数据块都应释放，例如网络设备的组传送地址列表 dev->mc_list 由 dev_mc_discard 函数将其移走。
- 所有在 register_netdevice 中由 dev->init 完成的工作需由 dev->uninit 函数取消。

最后，unregister_netdevice 函数调用 net_set_todo 让 netdev_run_todo 来完成网络设备的

最终注销任务，对网络设备的引用计数由 dev_put 函数来递减。netdev_run_todo 将网络设备从/sys 文件系统中注销，改变网络设备状态 dev->reg_state 为 NETREG_UNREGISTERED。等到所有的引用都释放后，用 dev->destructor 完成设备的注销。

4.3.3　网络设备的引用计数（reference count）

对网络设备 struct net_device 数据结构实例引用没有完全释放前，网络设备不能释放。对网络设备的引用计数存放在 dev->refcnt 数据域中，该值每次在有新的用户引用或释放设备引用时分别由函数 dev_hold 和 dev_put 做递增或递减操作。

当一个网络设备由 register_netdevice 函数向内核注册时，dev->refcnt 的值初始化为 1。第一个引用计数由内核代码保存，负责维护网络设备数据库，而且第一个引用计数只有在调用了 unregister_netdevice 时才释放，也即 dev->refcnt 数据域的值不会为 0，除非设备被注销。网络设备与别的内核对象不一样，当引用计数为“0”时由 xxx_put 函数来释放对象，网络设备的 net_device 数据结构实例只有在将网络设备从内核中注销后才会释放。

总结起来，在 unregister_netdevice 最后调用 dev_put 函数后还不足以释放网络设备的 struct net_device 数据结构实例，内核还需要等到所有引用该设备的对象都释放对网络设备的引用后，设备才能完全释放。但网络设备在注销后就不再可用，内核需要通知持有网络设备引用的组件释放它们对网络设备的引用。内核组件通过网络子系统通知链 netdev_chain 传送的 NETDEV_UNREGISTER 事件获取消息，然后释放对网络设备的引用。这也意味着，网络设备引用的持有者应该将自己的处理函数注册到网络子系统的通知链中，它们才能知道网络设备什么时候已注销。

在前面我们提到 unregister_netdevice 函数开始设备的注销过程，并让 netdev_run_todo 来完成网络设备的最后注销，netdev_run_todo 调用 netdev_wait_allrefs 来等待所有对网络设备的 struct net_device 数据结构实例的引用都释放。下面我们就来看看 netdev_wait_allrefs 的实现细节，如图 4-8 所示。

netdev_wait_allrefs 函数由一个循环组成，只有当 dev->refcnt 的值为 1 时才结束循环，每秒钟它都送出一个 NETDEV_UNREGISTER 事件的通知，每 10 秒在控制终端上打印一次警告信息，其他时候 netdev_wait_allrefs 函数处于休眠状态。netdev_wait_allrefs 函数不会停止，直到所有的引用都释放。下面的两种情况会需要多次发送 NETDEV_UNREGISTER 事件通知。

1．程序有 bug

一段代码持有网络设备的 struct net_device 数据结构实例的引用，但它可能没有注册到网络子系统的 netdev_chain 的通知链中，所以没有收到网络设备注销的事件通知，也没有释放对网络设备的引用，或它处理通知链的事件出错。

2．挂起时间

假设一个例程在访问持有网络设备的 struct net_device 数据结构实例的引用对象时，该例程在 CPU 上的执行时间超时，则需要等到下一个时间片时才能释放对网络设备实例的引用。

当网络子系统的事件通知链 netdev_chain 发送网络设备注销事件通知时，它也将与该

设备相关的链路状态改变（网络是否连接在网络设备上），则事件挂起。因为该设备将要注销，所以内核不需再处理任何关于设备链路状态改变的事件，也即该设备的事件列表被清除，只有别的活动设备的事件才会被处理。

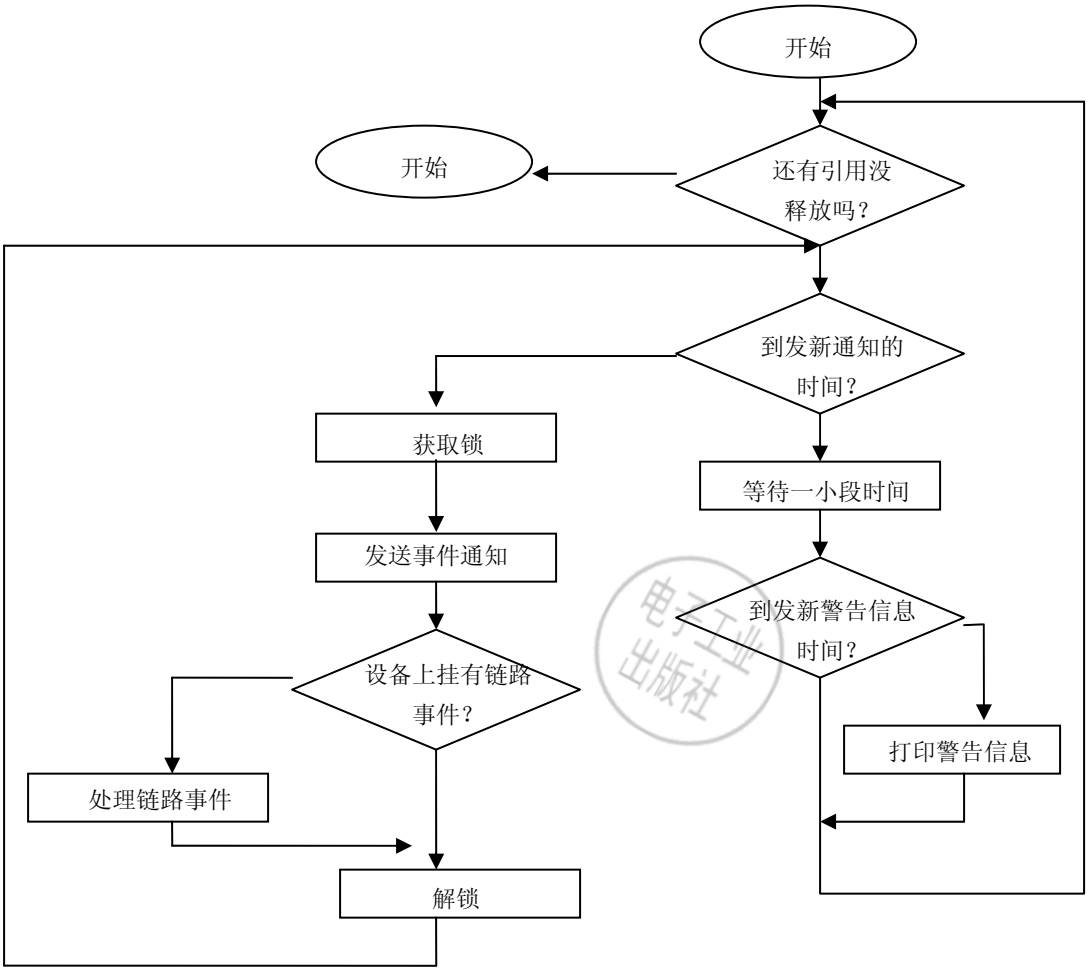


图 4-8　函数 netdev_wait_allrefs 的流程

4.3.4　允许和禁止网络设备

一旦一个网络设备注册到内核后，该网络设备就可以使用了。但它还不能进行数据包的接收和传送操作，直到用户允许该网络设备后，网络设备才被激活，随后才能开始数据收发操作。

1．激活网络设备

激活网络设备由 dev_open 函数完成。激活一个设备，要完成的任务包括：

- 调用 dev->open，不过网络设备驱动程序必须实现了 open 函数，并初始化了设备的 open 函数指针。

- 将 dev->state 的标志设为__LINK_STATE_START，标志设备开始运行。
- 设置 dev->flags 的标志为 IFF_UP，表明设备已经可以工作。
- 调用 dev_activate 函数初始化网络设备的数据包传送队列策略，该队列策略由流量控制系统使用，启动 watchdog 时钟。如果用户没有配置流量控制，队列策略默认为先进先出（FIFO）。
- 通过网络子系统的事件通知链 netdev_chain 发送 NETDEV_UP 事件通知，通知内核的其他组件，此设备可用。

2．禁止网络设备

既然网络设备可以显示被允许，它也可以由用户以命令的方式或通过其他事件禁止网络设备工作。禁止网络设备的使用是通过 dev_close 函数来完成的，有如下步骤。

- 通过网络子系统的事件通知链发送 NETDEV_GOING_DOWN 事件，通知内核的组件，网络设备将停止工作。
- 调用 dev_deactivate 函数停止网络设备的传送队列，保证网络设备不能再用于发送数据包。停止 watchdog 的时钟，因为它已不再有用。
- 清除 dev->state 状态标志的__LINK_STATE_START 位，表明网络设备已下线。
- 如果轮询（polling）被调度来读网络设备的输入队列，等到网络设备的轮询活动结束。因为__LINK_STATE_START 标志位已经被清除，所以不会再有别的数据包被轮询进该设备，但在该标志被清除前可能已有数据包进入并挂在设备的接收队列上。
- 调用 dev->stop 函数（如果驱动程序定义了该函数）。
- 清除 dev->flags 的 IFF_UP 标志。
- 通过网络子系统的事件通知链 netdev_chain 传送 NETDEV_GOIHG_DOWN 事件，通知内核的其他组件网络设备已停止工作。

4.4　网络设备的管理

网络设备的 struct net_device 数据结构实例在创建后会插入到一个全局链表 dev_base_head 和两个哈希链表中，这些数据结构使内核查找设备更容易。

4.4.1　管理网络设备的链表

1．网络设备实例全局链表

dev_base_head 全局链表中包含了所有网络设备的 struct net_device 数据结构实例，使内核管理网络设备更容易。例如，内核可以得到所有网络设备的统计信息，按照用户的命令修改所有网络设备的配置，或搜索符合给定条件的网络设备。

因为每个网络设备驱动程序都可能有自己的私有数据结构，所以链表中网络设备的 net_device 数据结构实例大小可能不一样。

（1）网络设备实例什么时候加入到全局链表中

所有的网络设备注册后就放入由 dev_base_head 指针指定起始地址的全局链表中，net_device->dev_list 数据域的*next 和*prev 指针将各网络设备链接在全局链表中，如图 4-9 所示。

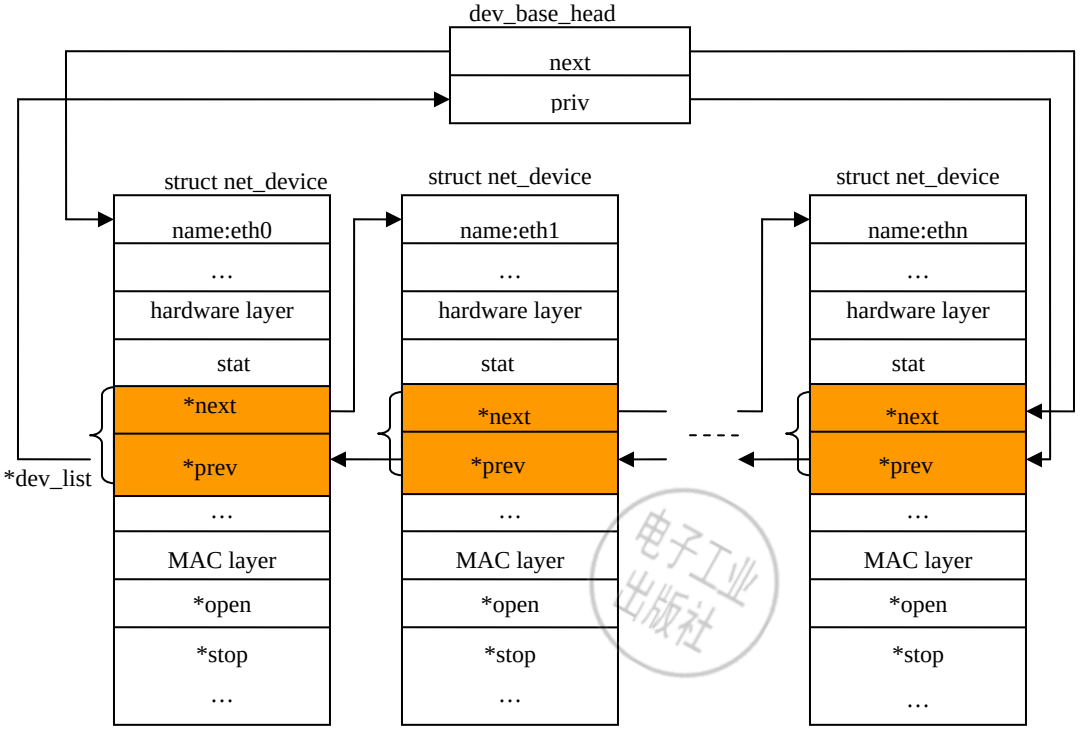


图 4-9　管理网络设备实例的全局链表 dev_base_head

（2）操作全局链表的函数

内核实现了以下两个函数，将网络设备的 net_device 数据结构实例插入或移出全局链表和两个哈希链表。

```
static int list_netdevice(struct net_device *dev)
static void unlist_netdevice(struct net_device *dev)
```

（3）防止对全局链表的并发访问

全局链表 dev_base_head 由 dev_base_lock sprin_lock 和 rtnl semaphore 保护，在对链表进行写操作时，必须首先获取 rtnl semaphore，开始在全局链表中查找要操作的网络设备；找到匹配的网络设备后，进行实际的写操作前，必须获取 dev_base_lock 写操作锁。这样做的目的是对全局链表的只读操作在写操作查找网络设备期仍能访问全局链表。

2．哈希链表

为了实现对系统中网络设备的快速查找，网络子系统还建立了两个哈希链表，这两个哈希链表分别以网络设备名和网络设备索引号为关键字而建立，可方便进程从不同角度搜

索网络设备。

（1）dev_name_head 哈希链表

以设备名（dev->name）为关键字的哈希链表，在使用 ioctl 接口改变网络设备配置时非常有用，网络配置工具 ifconfig 通常使用 ioctl 与内核通信，用网络设备名做索引。

（2）dev_index_head 哈希链表

以设备索引号为关键字的哈希链表(dev->ifindex)。对网络设备的 struct net_device 数据结构实例的交叉引用通常既存放指向网络设备的 struct net_device 数据结构实例的指针，也存放网络设备的索引号。新的网络配置工具 ip，使用 Netlink socket 与内核通信，一般使用设备索引号来引用网络设备。

以下两个函数用于建立网络设备的哈希链表：

```
static inline struct hlist_head *dev_name_hash(struct net *net, const char *name)
{
  unsigned hash = full_name_hash(name, strnlen(name, IFNAMSIZ));
  return &net->dev_name_head[hash & ((1 << NETDEV_HASHBITS) - 1)];
}

static inline struct hlist_head *dev_index_hash(struct net *net, int ifindex)
{
  return &net->dev_index_head[ifindex & ((1 << NETDEV_HASHBITS) - 1)];
}
```

4.4.2　网络设备的搜索函数

在处理网络协议栈的数据包收发过程时常需要搜索网络设备，例如协议栈上层传来的数据包指定由哪个网络设备向外发送，用户通过 ifconfig、ip 等工具配置网络设备参数时都需要搜索网络设备。为此内核基于前面所述的管理网络设备的全局链表与两个哈希链表实现了一系列的网络设备搜索函数，以满足各方面的查询需求。

最常用的搜索都是基于网络设备名和网络设备索引号的，它们由函数 dev_get_by_name 和函数 dev_get_by_index 实现。两个函数基于前面介绍的两个哈希链表，当然搜索函数也可以基于设备类型、MAC 地址、设备状态标志等，这些搜索方法就基于设备全局链表 dev_base_head。

1．基于哈希链表的网络设备搜索函数

```
struct net_device *__dev_get_by_name(struct net *net, const char *name)
struct net_device *dev_get_by_name(struct net *net, const char *name)
```

以上两个函数在以网络设备名为关键字建立的哈希链表中查找网络设备。查找网络设备是只读操作，只需获取 rtnl semaphore 锁。dev_get_by_name 函数是__dev_get_by_name 的包装函数。dev_get_by_name 获取 rtnl semaphore 锁后调用__dev_get_by_name 函数在哈希链表中展开查询。如果查找到匹配的网络设备，函数返回指向网络设备的 struct net_device 数据结构实例的地址，否则返回 NULL。

```
__dev_get_by_index() / dev_get_by_index()
```

以上两个函数在以网络设备索引号为关键字的哈希链表中查找网络设备。如果没找到匹配的网络设备，函数返回空指针 NULL。同样 dev_get_by_index 是__dev_get_by_index 的包装函数，目的是获取保护链表的读操作锁。

2．基于网络设备全局链表的搜索

（1）按网络设备硬件地址搜索

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

给出网络设备的硬件地址（MAC 地址），此函数在管理网络设备的全局链表中搜索与给定类型匹配的网络设备。如果没找到，函数返回空指针，否则返回网络设备的 struct net_device 数据结构实例地址。

（2）按网络设备类型搜索

```
struct net_device * __dev_getfirstbyhwtype(struct net *net , unsigned short type )
struct net_device * dev_getfirstbyhwtype(struct net *net , unsigned short type )
```

以网络设备硬件类型（以太网卡，令牌环网络设备等）为关键字在全局链表中查找第一个与指定类型相符的网络设备。dev_getfirstbyhwtype 是__dev_getfirstbyhwtype 的包装函数，获取全局链表读操作保护锁。

（3）按网络设备标志搜索

```
struct net_device * dev_get_by_flags (struct net *net , unsigned short if_flags, unsigned short
mask )
```

在全局链表中查找第一个与给定 flags 值相符的网络设备。

4.5 事件通知链

内核中的许多子系统都相互依赖，其中某个子系统的状态发生了改变或发生了某个特定事件，别的子系统都需要做出相应的处理。例如网络设备可以动态注册到内核，从内核中注销，即网络设备会在系统运行时期呈现在内核中，也会在系统运行时从内核中消失，网络设备的状态在运行过程中也在随时发生变化。此外，网络设备还可以改变其硬件地址、网络设备名、硬件功能特性等。

如果网络设备状态的改变只影响设备和设备驱动程序这一级，这不会引起任何问题。但在网络子系统协议栈中，为了实现网络传输的高性能，简化协议处理，协议栈实例往往将它们使用的设备引用（reference）固定存放在自己的数据结构中，当网络设备的状态发生变化时（如一个设备注销了），在协议栈实例中存储的信息就无效了，这时协议栈实例不应再通过无效的网络设备收发数据包。因此协议栈实例需要获取网络设备状态发生变化的消息，以便按照网络设备状态的变化进行处理。

这里存在的问题是网络设备自身并不知道有哪些协议实例在使用它的服务，存储了对它的引用。为了满足让协议栈实例获取网络设备状态信息的需求，Linux 内核中实现了事件通知链（notification chain）机制。

事件通知链是一个非常奇妙的设计，它使 Linux 内核子系统之间的事件信息交互灵活、高效。在这一节中，我们将会介绍事件通知链机制在内核中的设计原理，以及网络子系统创建了什么事件通知链。有以下几个方面的内容。

- 通知链如何声明，网络子系统都定义了什么类型的通知链。

- 内核的子系统如何将自己注册到一个通知链中，以获取事件通知。
- 内核的子系统如何在通知链上发送事件消息。

4.5.1　事件通知链构成

1. 什么是事件通知链

事件通知链是一个事件处理函数的列表，每个通知链都与某个或某些事件相关，当特定的事件发生时，列在通知链中的函数就依次被执行，通知事件处理函数所属的子系统某个事件发生了，子系统接到通知后做相应的处理。每个通知链中都存在被通知方和通知方。

- 被通知方：内核中的某个子系统，提供事件处理的回调函数。
- 通知方：内核中发生事件的子系统，通知链的创建者，事件发生后通知方依次调用事件通知链上的事件处理函数，通知其他子系统某个事件发生，对方应做出相应处理。

通知方是事件通知链创建方，它定义事件通知链，需要获取通知方事件消息的其他子系统编写自己的事件处理函数，并向通知方的事件通知链中注册函数。

2. 通知链中的成员

事件通知链中是一个一个的 struct notifier_block 数据结构类型的成员列表，struct notifier_block 数据结构描述了通知链的成员由哪些属性组成，这样当事件通知链中的成员接收到事件通知后，才能做出正确反应。struct notifier_block 数据结构的定义如下。

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

struct notifier_block 数据结构中包含了 3 个数据成员。

（1）notifier_call 函数指针

notifier_call 函数指针指向事件处理函数。事件处理函数应由事件被通知方实现，是事件被通知方接收到事件通知后要完成的处理。例如在网络协议栈的各层中，每个保留了网络设备状态的协议实例都应实现一个事件处理的回调函数，放到网络子系统的事件通知链上。

事件处理函数需要 3 个输入参数。

- struct notifier_block *self：指明事件通知来自系统的哪个通知链上。
- unsigned long：当前发生的是什么事件，该参数是发生事件的编码。
- void*：传给事件处理回调函数的参数。该参数是一个无类型指针，如果把这个把参数放在数组中，可以给事件处理函数传多个参数。

（2）struct notifier_block *next

将事件通知链中成员连接成列表的指针。

（3）priority

表示事件处理函数的优先级。具有高优先级的函数先执行。但在实际应用中，几乎所有注册函数都没有设置 priority 的值，也即 priority 为默认值，而事件函数的执行顺序按函

数注册到事件通知链中的先后顺序进行。

在内核中，你可以看到 struct notifier_block 结构实例的名称常常为：xxx_chain、xxx_notifier_chain 和 xxx_notifier_list。

图 4-10 是网络子系统事件通知链 netdev_chain 的示意图，图中的方法如 ipmr_device_event 是桥接器的事件处理函数，它将该事件处理函数注册到网络子系统的事件通知链中。当某个事件发生时，该函数会被调用完成桥接器在事件发生时应做的回应。

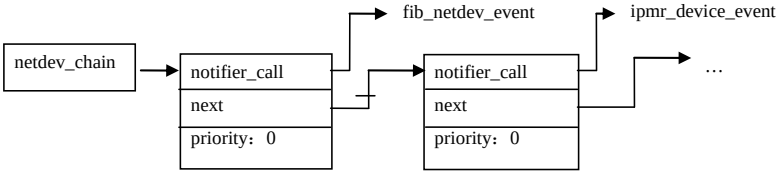


图 4-10　网络子系统事件通知链 netdev_chain

4.5.2　注册回调函数到事件通知链

Linux 内核定义了一系列的事件通知链，用于通知内核其他组件有事件发生。内核中某个组件需要对一个已存在的事件通知链上所产生的事件做处理时，该组件需要向这个事件通知链注册自己的事件处理回调函数。这个过程可以使用通用注册函数 notifier_chain_register 完成。实际上大多数拥有事件通知链的子系统，都实现了向自己的事件通知链注册的专用函数，这些专用函数是 notifier_chain_register 注册函数的包装函数，调用相应的事件通知链包装函数即向某个特定事件通知链注册了自己的事件处理回调函数。

1．向通知链注册事件处理函数

调用 notifier_chain_register (定义在 kernel/notifier.c 文件中) 函数可以向 Linux 内核的各个事件通知链注册事件处理回调函数，所以调用 notifier_chain_register 函数时需要向其传送参数指明向哪个事件通知链注册回调函数。这个函数很少直接使用，通常我们都是使用它的包装函数。

```
int notifier_chain_register(struct notifier_block **list, struct notifier_block *n)
{
while ((*nl) != NULL )
{
if(n->priority> (*nl)->priority)
         break;
      nl=&((*nl)->next);
   }
   n->next=*nl;
   rcu_assign_pointer(*nl,n);
   return 0;
}
```

调用事件通知链注册函数时需要向 notifier_chain_register 传送两个参数。

- struct notifier_block **list：注册到哪个事件通知链上。

- struct notifier_block *n：注册到事件通知链上的成员，为 struct notifier_block 类型变量，其中包含事件处理回调函数。

notifier_chain_register 遍历指定的事件通知链，将新成员的优先级与链表中现有成员优先级比较，如果发现新插入成员的优先级大于链表中某个成员的优先级，就把新成员插入到成员之前。如果新成员没有设置优先级，或链表中现有成员的优先级都高于新成员，将新成员追加到链表末尾。

2．网络子系统中创建的事件通知链

网络子系统共创建了 3 个事件的通知链：inetaddr_chain、inet6addr_chain 和 netdev_chain。其中 netdev_chain 为网络设备状态发生变化时的事件通知链，该事件通知链定义在 net/core/dev.c 文件中：

```
static RAW_NOTIFIER_HEAD(netdev_chain);
```

inetaddr_chain 和 inet6addr_chain 分别为网络设备的 IPv4 地址或 IPv6 地址发生变化时的事件通知链。表 4-5 中列出了向这 3 个通知链注册事件处理函数的包装函数。

表 4-5　向网络子系统事件通知链注册的包装函数

操　作	函 数 原 型	
注册	int notifier_chain_register(struct notifier_block **list, struct notifier_block *n)	
	包装函数	操作的通知链
	register_inetaddr_notifier	inetaddr_chain
	register_inet6addr_notifier	inet6addr_chain
	register_netdevice_notifier	netdev_chain
取消注册	int notifier_chain_unregister(struct notifier_block **list, struct notifier_block *n)	
	包装函数	操作的通知链
	unregister_inetaddr_notifier	inetaddr_chain
	unregister_inet6addr_notifier	inet6addr_chain
	unregister_netdevice_notifier	netdev_chain
传送事件通知	int notifier_call_chain(struct notifier_block **n, unsigned long val, void *v)	

调用以上函数后，notifier_block 结构实例就按照优先级顺序插入到相应的事件通知链表中。相同优先级的成员按照插入时间顺序排列。

notifier_lock 是保护事件通知链的锁。对所有的事件通知链使用一个锁不会对性能产生太大的影响。通常各子系统都是在系统启动或运行期间装载模块时向事件通知链注册自己的处理回调函数，以后对事件通知链的访问都是只读操作。

3．向事件通知链注册的步骤

从上述描述我们可以看到，如果内核子系统需要获取某个事件通知链传送的事件通知，对发生的事件做处理，要完成以下步骤来完成事件处理函数的注册。

① 声明 struct notifier_block 数据结构实例。

② 编写事件处理回调函数。

③ 将事件处理回调函数的地址赋给 struct notifier_block 数据结构实例的*notifier_call 成员。

④ 调用特定事件通知链的注册函数，将 struct notifier_block 数据结构实例注册到通知链中。

4.5.3 通知子系统有事件发生

拥有事件通知链的事件通知方在事件发生时，调用 notifier_call_chain 函数（ 定义在 kernel/notifier.c 文件中）通知其他子系统有事件发生。

notifier_call_chain 函数会按照事件通知链上各成员的优先级顺序调用注册在通知链中的所有回调函数。在这里需要注意的是，事件处理回调函数的执行现场在 notifier_call_chain 进程的地址空间。

```
static int __kprobes notifier_call_chain(struct notifier_block **nl,
                        unsigned long val, void *v,
                        int nr_to_call,  int *nr_calls)
{
//声明局部变量，ret 为事件通知函数的返回值，nb 与 next_nb 分别指向当前正在调用的事件处理函数与要调用的事件处理函数
int ret = NOTIFY_DONE;
  struct notifier_block *nb, *next_nb;
//获取事件通知链的首地址
  nb = rcu_dereference(*nl);
//事件通知链不为空，且成员数不为 0，遍历事件通知链，将通知链中下一个成员赋给 next_nb
while (nb && nr_to_call) {
        next_nb = rcu_dereference(nb->next);
//调用事件处理回调函数，调用成员计数加 1
        …
        ret = nb->notifier_call(nb, val, v);

        if (nr_calls)
             (*nr_calls)++;
//如果事件处理函数返回值为 NOTIFY_STOP_MASK，停止事件通知过程，跳出遍历通知链

        if ((ret & NOTIFY_STOP_MASK) == NOTIFY_STOP_MASK)
             break;
        nb = next_nb;
        nr_to_call--;               //要调用的通知链数减 1
  }
  return ret;
}
```


1．输入参数

对 notifier_call_chain 函数的调用，我们需要关心 3 个参数。

- struct notifier_block **nl ：表明哪个通知链发出的事件通知在调用事件处理回调函数。
- unsigned long val：发生的事件类型。Linux 系统定义的所有事件类型包含在 include/linux/notifier.h 文件中。
- void *v：传给事件处理回调函数的参数。

notifier_call_chain 的其他输入参数对我们并不重要。nr_to_call 是要调用的事件处理回调函数的个数，该值我们不必关心，调用时传−1 给它。nr_calls 是已调用了的事件处理回调函数的个数，这是 notifier_call_chain 函数自己计数的值。

2．函数返回值

notifier_call_chain 函数的返回值都为 NOTIFY_XXX 的形式，定义在 include/linux/notifier.h 文件中，表 4-6 给出了函数的返回值及其含义。

表 4-6　notifier_call_chain 函数的返回值

函数返回值	含　义
NOTIFY_OK	事件处理正确
NOTIFY_DONE	对该事件无须做处理
NOTIFY_BAD	事件处理出错，不再继续调用该事件的回调函数
NOTIFY_STOP	例程调用不正确，不再继续调用该事件的回调函数
NOTIFY_STOP_MASK	该标志由 notifier_call_chain 函数检查，看是继续调回调函数，还是停止。NOTIFY_BAD 和 NOTIFY_STOP 的定义中都包含了该标志

notifier_call_chain 捕获并返回最后一个事件处理函数的返回值。需要注意的是，对同一通知链，notifier_call_chain 函数可以同时被不同的 CPU 调用，事件处理函数要负责保证操作的互斥。

4.5.4　网络子系统中的事件通知链

Linux 内核中定义了大约 10 种不同的通知链，这里我们主要介绍对网络子系统特别重要的通知链。在网络子系统中一共创建了 3 个事件通知链。

1．inetaddr_chain

本地网络接口的 IPv4 地址发生了变化（如插入了一个新地址），或网卡的 IP 地址移走、改变时，在该事件通知链上发出事件通知。

2．inet6addr_chain

本地网络接口的 IPv6 地址变化时，在此事件通知链上发生事件通知。

3．netdev_chain

用于发送关于网络设备注册、状态变化的通知。

网络子系统的代码也会向内核中其他组件的事件通知链注册自己的事件处理函数。比如某些网络接口控制器的设备驱动程序会向 reboot_notifier_list 事件通知链注册事件处理回调函数，reboot_notifier_list 事件通知链在系统将要重启时发出事件通知。

如果内核组件需要处理某个事件通知链上发出的事件通知，该组件应在其初始化时在事件通知链上注册自己的事件处理回调函数。以下的代码片段是路由子系统的初始化函数 ip_fib_init，在该函数中路由子系统向 inetaddr_chain 和 netdev_chain 两个事件通知链上注册了事件处理回调函数（源文件在 net/ipv4/fib_frontedn.c 中）。

```
//定义处理网络接口 IPv4 地址变化事件的 notifier_block 数据结构实例
static struct notifier_block fib_inetaddr_notifier = {
   .notifier_call = fib_inetaddr_event,
};
//定义处理网络设备注册事件的 notifier_block 数据结构实例
static struct notifier_block fib_netdev_notifier = {
   .notifier_call = fib_netdev_event,
};
//在路由子系统的初始化函数中向 netdev_chain 和 inetaddr_chain 通知链注册回调函数
void __init ip_fib_init(void)
{
   …
   register_netdevice_notifier(&fib_netdev_notifier);
```

```
    register_inetaddr_notifier(&fib_inetaddr_notifier);
    …
}
```

4.5.5 网络子系统传送的事件

网络子系统在 net/core/dev.c 文件中定义了网络设备注册状态的事件通知链：STATIC RAW_NOTIFIER_HEAD(netdev_chain)。当网络设备的相关状态和功能特性发生变化时，网络子系统会调用注册在 netdev_chain 事件通知链上的所用事件处理回调函数。在 struct net_device 数据结构中有几个数据域从不同的方面描述了网络设备的状态（dev->flags）、传送队列的状态（dev->state）、设备注册状态（dev->reg_state），以及设备的硬件功能特性（dev->features）。当以上数据域的值发生变化时，意味着网络设备的状态发生改变，网络子系统就在 netdev_chain 通知链上发出通知。

网络子系统的事件格式为 NETDEV_XXX，定义在 include/linux/notifier.h 文件中。表 4-7 描述了从网络子系统发出的事件通知类型。

表 4-7 网络子系统的事件描述

事件符号名	事 件 描 述
NETDEV_UP	激活一个网络设备（由 dev_open 函数报告）
NETDEV_DOWN	停止一个网络设备，当该事件通知发出时，所有对该设备实例的引用都应释放
NETDEV_REBOOT	检查到网络设备接口硬件崩溃，硬件重启
NETDEV_CHANGE	网络设备的数据包队列状态发生了变化
NETDEV_REGISTER	一个网络设备实例注册到系统中（插入到全局链表和两个哈希链表中），但还没有激活
NETDEV_UNREGISTER	网络设备的驱动程序已卸载
NETDEV_CHANGEMTU	网络设备的最大传送单元（MTU）发生了变化
NETDEV_CHANGEADDR	网络设备的硬件地址改变
NETDEV_CHANGENAME	网络设备名改变
NETDEV_GOING_DOWN	网络设备即将注销，由 dev->close 报告，通知相关子系统处理
NETDEV_FEAT_CHANGE	网络设备的硬件功能特性改变

4.6 本章总结

Linux 操作系统的内核是组件化，内核各组件功能明确，既相互独立又相互作用。内核中的各个组件在能正常工作、为其他组件提供服务之前，都需要经过一定的初始化过程，才能被内核识别。组件的初始化过程既具有内核启动时的一般特点，又具有各自的特性，网络子系统也不例外。

本章首先分析了内核启动、初始化的一般特点和过程，系统如何定义与管理内核启动时的初始化函数与命令行参数，在此基础上介绍了网络子系统与网络设备的创建、初始化和注册在内核中的实现。

最后介绍了在内核中识别网络设备后，内核如何有效地组织管理网络设备，以及网络设备状态发生变化时的事件通知链机制。

第 5 章　网络设备驱动程序

本章重点介绍网络设备硬件与内核通讯的机制中断与软件中断在 Linux 中的实现，给出了 Linux 中网络设备驱动程序的架构。在此基础上，以实例代码讲解网络设备驱动程序的实现。最后以一个具体的网络设备控制器 CS8900A 为实例，讲解网络设备驱动程序的设计、实现方法与过程。

到目前为止，我们已经介绍了网络子系统的两个实体在内核中的数据结构：存放网络数据包的 Sk-buff 和表示网络设备的 struct net_device 数据结构；还介绍了数据包和网络设备在内核中的组织，网络子系统与网络设备的初始化过程，以及网络设备与上层协议栈之间的事件通知机制（netdev_chain），可以进一步深入分析网络体系结构、TCP/IP 协议栈在 Linux 内核中的实现。

在本书的其余部分，我们将从 TCP/IP 协议栈的底层开始，自下而上地分析 Linux TCP/IP 协议栈各层协议在内核的实现，网络数据包如何在 TCP/IP 协议栈接收/发送。我们会逐个分析网络体系结构中各层网络协议的特性，以及这些特性在 Linux 内核中的代码描述。

这一章我们将从 TCP/IP 协议栈的最底层物理层开始，分析 Linux 网络子系统中网络设备驱动程序的架构和开发实例。

5.1　网络设备驱动程序概述

网络设备驱动程序是网络物理设备与 Linux 内核之间的一个桥梁，它是将网络设备从主机外界：计算机网上收到的数据，从网络设备的数据空间传到内核空间的软件（发送过程与此相反），也即网络设备驱动程序是网络设备硬件与 Linux 内核间的接口。

那么在网络应用中，网络设备驱动程序需要完成哪些最常规的功能才具备成为网络设备与内核之间接口的能力，网络设备又是以什么方式来与内核通信，让内核能及时获取或发送网络数据的呢？

本章首先会给出网络设备驱动程序的基本构架，接着以两个实例来展示网络设备驱动程序的开发过程。第一个实例是 Linux 中网络设备驱动程序的示例程序 isa-skeletion，用这个示例程序来解释驱动程序的工作原理。它不涉及任何具体的硬件操作，如操作网络设备的寄存器、I/O 存储空间等，但这个示例程序给出了网络设备驱动程序的总体结构和功能特点，任何编写网络设备驱动程序的人员都可以以此为蓝本获取驱动程序的框架，照此框架完成基本的开发。最后会针对一个实际的网络接口控制器 cs8900 来讲解网络设备驱动程序的开发实现。

5.1.1 网络设备驱动程序的任务

网络接口是 Linux 系统中第三类标准设备，网络接口在系统中的角色是内核与外界（通过网络连接）交换数据的通道。网络接口必须以特定的数据结构类型（struct net_device）向内核注册自己，以便在需要的时候内核调用它与外界通信。

网络设备驱动程序要完成两大基本功能：其一，最重要也是最基本的功能，是响应内核的要求，向外发送数据包，并异步地从计算机网络上接收数据包，将收到的网络数据包推送给内核；其二，网络设备驱动程序还需要支持一系列的管理任务，如设置网络设备的地址，修改发送参数，管理流量和错误统计等。网络驱动程序的 API 显示了对这些功能的支持。

5.1.2 网络设备驱动程序的构成

按照上节所描述的网络设备驱动程序的任务，我们以网络设备实例从创建、探测、正常工作、完成收发任务与管理任务为线索，来看看网络设备驱动程序的构成。

1．模块初始化/清除功能

在 4.2.4 节和 4.3 节介绍网络设备初始化和注册过程中，已说明任何网络设备实例在进入正常工作状态前都必须初始化，所以网络设备驱动程序首先需要一个模块的初始化函数，该函数由 module_init 宏来标记或直接命名为 init_module，当网络设备驱动程序以模块的方式装载到内核中，该函数在装载网络设备驱动程序模块时被执行。它创建网络设备实体对应的 struct net_device 数据结构实例，初始化部分网络设备实例的数据域，然后调用 Net-device 驱动程序的 xxx_probe 函数，完成 Net-device 驱动程序对网络设备实例的特殊数据域的初始化。

2．网络硬件设备探测函数

网络设备驱动程序直接编译到内核时，如果在内核配置时选择了某个网卡作为系统网络通信的设备，内核在启动过程中会首先探测指定的网络设备硬件；一旦内核探测到安装的网络设备硬件后，就调用网络设备驱动程序的初始化函数来初始化一个网络设备实例。

内核如何才能探测到硬件呢？实际的网络硬件探测方法也需要由网络设备驱动程序提供。一般该方法的函数名为 xxx_probe，将 xxx_probe 函数放到 drivers/net/Space.c 文件中，系统启动时就会自动调用配置硬件的探测函数来探测网络设备硬件。例如 CS8900A 网络控制器的探测函数为 cs89x0_probe。至于内核如何自动探测网络设备硬件，下面将详细描述这部分功能在 drivers/net/Space.c 文件中的实现。

首先网络设备驱动程序必须实现自身的硬件设备自动探测函数，一般命名为 xxx_probe。自动探测函数与设备硬件特性密切相关，它需要按照硬件呈现在系统中的一些特征来判断设备是否已连接到系统中。例如：每一种网络控制器上的寄存器、数据缓冲区映射到内存的区域，都可以作为判断的条件之一。

然后，将网络设备的自动探测函数放入 drivers/net/Space.c 文件中，其格式如下：

```
extern struct net_device *xxx_probe(int unit)
```

根据网络控制器连入系统的总线类型，如 ISA 总线、EISA 总线、微通道 MCA 等，将网络设备的自动探测函数加入到 struct devprobe2 数据结构类型数组中。

```
struct devprobe2 isa_probes[] = {
…
#ifdef CONFIG_CS89x0
        {cs89x0_probe, 0},
#endif
…
}
```

上述代码片段说明，如果在内核配置时，选择了 CS8900X 系列的网络设备，将 CS8900X 网络设备自动探测函数放入 ISA 总线数据结构数组中，CS8900x 系列的网卡与系统通过 ISA 总线连接。

最后，内核的以太网设备自动探测过程会根据内核配置依次调用 ISA、EISA、MCA 等总线类网络设备的探测函数，发现设备后就初始化网络设备实例，并注册到内核中。

```
//声明 Linux 支持所有网络设备的探测函数
…
extern struct net_device *apne_probe(int unit);
extern struct net_device *cs89x0_probe(int unit);
extern struct net_device *hplance_probe(int unit);
//描述网络设备探测的数据结构，probe 函数指针指向网络设备的自动探测函数，如果自动探测失败，status 为非零值
struct devprobe2 {
  struct net_device *(*probe)(int unit);
  int status;
};
//如果在内核配置时选择了某个网卡，将该网卡的自动探测函数加入 devprob2 结构数组中，比如 cs89x0_probe
static struct devprobe2 isa_probes[] __initdata = {
#if defined(CONFIG_HP100) && defined(CONFIG_ISA)
  {hp100_probe, 0},
#endif
#ifdef CONFIG_3C515
  {tc515_probe, 0},
#endif
…
#ifdef CONFIG_CS89x0
  {cs89x0_probe, 0},
#endif
//在 probe_list2 函数中依次执行传入参数 struct devprob2 *p 数组中的网络设备自动探测函数，设置自动探测状态 status
static int __init probe_list2(int unit, struct devprobe2 *p, int autoprobe)
{
  struct net_device *dev;
  for (; p->probe; p++) {
        if (autoprobe && p->status)             //如探测状态表明该设备已探测过，则继续
              continue;
        dev = p->probe(unit);                   //执行自动探测函数
…
        if (autoprobe)
              p->status = PTR_ERR(dev);         //设置自动探测执行结果状态
  }
  return -ENODEV;
}
```

函数 probe_list2 前面用宏__init 标识，表明此函数会在内核启动过程中被调用执行。probe_list2 是在以太网类网络设备的自动探测过程中被调用的。

```
//执行所有以太网类网络接口和设备自动探测函数
static void __init ethif_probe2(int unit)
{
  unsigned long base_addr = netdev_boot_base("eth", unit);

  if (base_addr = = 1)
       return;

  (void)(    probe_list2(unit, m68k_probes, base_addr == 0) &&
             probe_list2(unit, eisa_probes, base_addr == 0) &&
             probe_list2(unit, mca_probes, base_addr == 0) &&
             probe_list2(unit, isa_probes, base_addr == 0) &&
             probe_list2(unit, parport_probes, base_addr == 0));
}
```

同样，ethif_probe2 函数的前面以宏__init 标识，表明 ethif_probe2 函数为内核启动时执行的函数。

在网络设备自动探测函数 xxx_probe 中（如 cs8900A 网络设备的自动探测函数是 cs89x0_probe）会创建网络设备实例，随后调用 xxx_probe1 完成网络设备的特殊初始化。

3．设备活动

网络设备经创建、初始化后注册到内核，由用户激活，就可以开始网络数据包的收发操作了。由于网络设备驱动程序本身并不知道什么时候数据会从网络或内核发送给它，所以网络数据包的收发是异步进行的。大多数网络设备驱动程序都以中断的方式来完成这种异步操作。描述网络设备活动的功能函数通常包括以下几类：

- 激活/停止设备（xxx _open/ xxx _close）。
- 收发网络数据（xxx_tx/xxx_rx）。
- 中断处理程序（xxx_interrupt ）。

4．管理任务

在网络数据包的接收与发送过程中可能会出错，设备驱动程序需要对出现的错误做处理；同时内核及用户应用程序有时需要获取数据收发的各项统计数据，此如正确接收/发送的数据包数、字节数，接收/发送出错的数据包数与字节数等。除此之外，驱动程序还应按用户要求改变网络设备配置参数，比如：硬件地址、设备最大发送单元 MTU、设备接收功能配置（flags）等。这部分功能函数通常包括以下几类：

- 错误处理/状态统计（xxx_tx_timeout / xxx_get_state）。
- 支持组发送功能函数（set_muticast_list/set_muticast_address ）。
- 改变设备配置（change_xxx 等）。

图 5-1 给出了网络设备驱动程序的构架框图。

在整个网络设备驱动程序的架构中，最重要的无非是实现网络设备活动功能函数。要实现这部分功能首先要解决的问题是，当网络设备收到数据包时，如何通知内核；内核要向网络发送数据包时如何调度网络设备的活动。下一节我们就介绍网络设备与内核交互的工作方式。

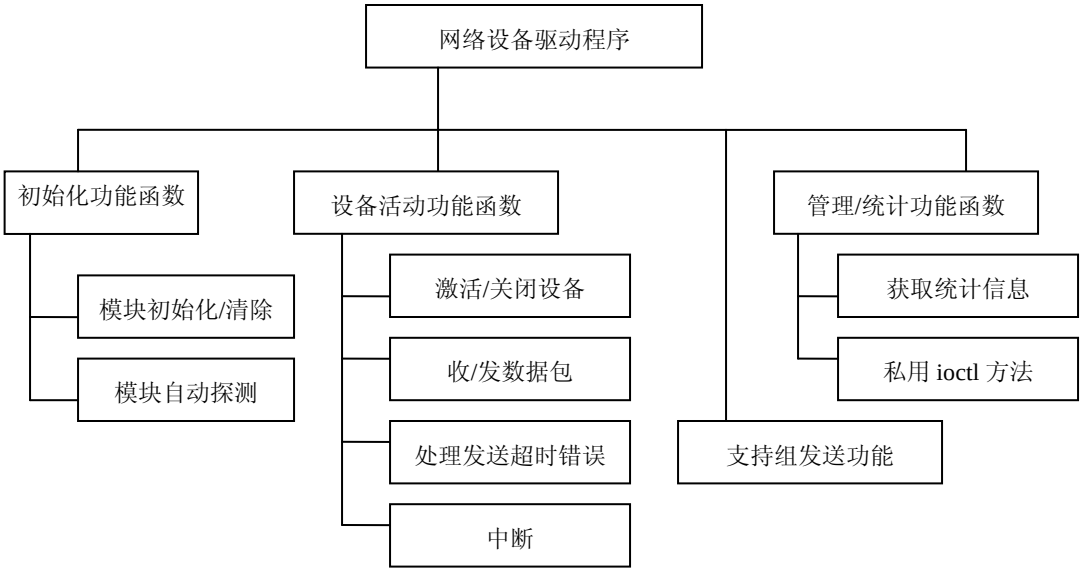


图 5-1　网络设备驱动程序架构框图

5.2　网络设备与内核的交互

网络设备与内核的交互主要发生在网络设备接收到数据包后：如何向内核推送本机产生的数据包；经 TCP/IP 协议栈处理后，如何由网络层向数据链路层传递。从现在开始我们将会大量应用到第 2 章到第 4 章介绍的数据结构和工作机制，在需要时读者可以回顾前面章节的一些内容。

在内核准备好处理网络数据包之前，它必须建立起系统的中断机制，使系统可以快速地处理网络数据包。连接到网络上的主机每秒钟可能会收到几千个数据包，系统应以什么策略来加速数据的接收过程，这就是本节要介绍的内容。除此之外，本节还将阐述以下内容：

- 当内核配置为支持对称多处理器系统（SMP），并运行在一个多处理器环境下时，网络数据包的接收和发送处理程序如何发挥多处理器的性能。
- 在介绍网络数据包的接收处理时，会覆盖网络设备驱动程序与内核接口的两种模式：旧模式目前大多数网络设备还在使用；新的接口 NAPI，在高负载网络通信的情况下可以大大提升网络数据吞吐量。

5.2.1　设备与内核的交互方式

几乎所有的网络设备与内核通信都使用轮询（polling）或者中断模式，或者两者的结合。我们首先给出轮询与中断工作方式的含义，再描述它们在内核中的实现。

1．轮询

在轮询工作方式中，内核每隔一段时间就查询设备的状态，看设备是否有数据需要与内核交换，这是通过读设备的状态/控制寄存器来判断的。这种方式耗费 CPU 的时间较多，数据交换速度慢，目前大多数设备都不再采用这种工作方式。轮询方式下设备与系统之间的连接如图 5-2 所示。

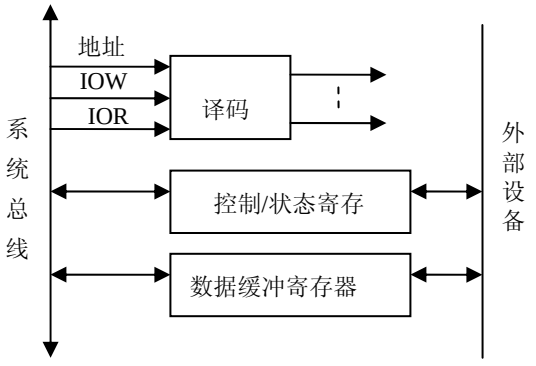


图 5-2　轮询工作方式下设备与系统的连接

系统通过地址译码和控制命令（IOW/IOR）选择访问设备的控制/状态寄存器，获取外部设备控制/状态寄存器的当前值，判断设备是否已有数据等待读取，如果外部设备中已有数据，则主机从设备的数据缓冲寄存器中获取数据。

2．中断方式

所谓中断方式是指在系统运行过程中，如果发生了某种随机事件，CPU 将暂停现行程序的执行，转去执行为该随机事件服务的中断处理程序，中断处理程序执行完成后自动恢复原程序的执行。由此可见，中断包含了随机性与程序的切换两个重要概念。

引起中断的随机事件是设备向内核发送的一个硬件信号，通知内核该设备有某类事件发生。CPU 在接到中断信号后，暂时停止现在正在执行的程序，转去执行与外部事件相关的处理程序，称为响应设备的中断请求。

例如，当引起中断的随机事件是网络设备接收到一个网络数据包时，中断处理程序就将网络数据包从设备的数据缓冲区复制到内核的接收队列中，通知内核有网络数据包到达。相反，如果引起中断的随机事件是一个网络数据包已发送完毕时，中断处理程序的任务就是通知内核网络设备可以开始接收新的发送数据包。

中断方式的好处就是它满足了网络数据包的发送与接收可以异步进行的要求，使 CPU 与网络设备之间可以并行工作。中断方式在低负载网络的情况下可以工作得很好。它的不足是当网络负载量很高时，网络设备每接到一个数据包就会产生一个中断请求，CPU 就需要从现行程序切换到中断处理程序，随后再切换回原程序。可以想象这种频繁的程序切换会使 CPU 耗费很多时间。

另一个问题是网络数据包的接收分成两个阶段：第一个阶段是网络设备驱动程序将数据包从硬件缓冲区复制到内核接收队列；第二个阶段是由内核 TCP/IP 协议栈来处理数据包。

第一个阶段的优先级比第二个阶段的优先级高，它可以抢占数据处理过程的 CPU 时间优先执行。在网络流量高峰时可能出现这样的情况：网络数据包持续进入系统，复制到 CPU 接收队列上，CPU 的接收队列容量终究有限，到某个时候接收队列满；但第二阶段的过程，将数据包从接收队列移走并处理的程序一直没有机会执行，因为它的优先级低。这样可能导致系统崩溃：CPU 接收队列满后，内核已没有空间可以容纳新的数据包，而旧的数据包又无法处理，因此 CPU 没有时间运行数据包处理程序。

综上所述，中断方式在数据流量较低的情况适合网络数据的收发和数据包处理的并行工作，但在网络流量很大时会出现一些问题。Linux 网络体系结构在处理接收网络数据包的中断方式中做了进一步的扩展。

3．在中断期间处理多个数据包

不少的网络设备使用这种方式：当中断产生并开始执行网络设备驱动程序的中断服务程序后，中断处理程序就不断地从网络上接收数据包放入内核的接收队列，直到接收到的数据包数已达到事先设定的一次可以接收的最大门限值。这里需要注意的是，在中断处理程序中需要禁止所有其他的硬件中断，所以某个中断处理程序不能执行太长的时间，以免其他设备的硬件中断得不到响应，丢失信息。

另一方面，任何接收队列都有一定的容量限制，所以中断处理程序中接收的数据包数也应受到限制，网络设备驱动程序需要对此做适当处理。这种技术不需对内核做任何改动，全部是在网络设备驱动程序中实现的。

对上述方式还可以做其他的优化：内核不必禁止所有的硬件中断，只需禁止当前有数据包进入，CPU 正在执行中断处理程序的网络设备的硬件中断。这正是 Linux 内核网络子系统中实现的新式接口 API 的工作方式（NAPI）。

5.2.2　硬件中断

通常网络数据的接收与发送都是以中断方式完成的。中断触发及其在硬件、软件上的实现细节是一个较复杂的过程，简单了解中断的过程，对编写网络设备驱动程序的中断服务程序有一定的帮助。如前所述，中断触发是由设备硬件引起的，设备的硬件中断信号与系统的连接关系如图 5-3 所示。

1．中断控制的硬件逻辑

在现代嵌入式处理器中通常都包含一个中断控制器，它接收外部设备的各路中断请求信号 IRQ_0~IRQ_i，将它们存放于中断请求寄存器中。目前中断控制器大多可以接收 16~32 个中断请求信号。未被 CPU 屏蔽的中断请求会送入优先级分析电路参加判优，由中断控制器产生一个公共的中断请求信号 INT，送至 CPU。当 CPU 响应中断请求时，发出中断响应信号 INTA，送往中断控制器；中断控制器将优先级最高的中断请求信号的硬件中断源类型码（即我们常说的中断号）经系统数据总线送至 CPU，作为 CPU 寻找该中断服务程序入口地址的依据。

从图 5-3 我们可以看到，一旦硬件设计完成，任何中断控制器对外的引脚数都是有限的，所以我们常说中断是有限的系统资源。有时几个硬件设备可以共享一个硬件中断信号。在一个中断请求被 CPU 响应时，会屏蔽其他硬件中断信号，这时如果其他设备有中断请求

产生，不能得到 CPU 的响应，所以中断处理程序的执行时间要尽可能的短，只处理最紧急的事务。

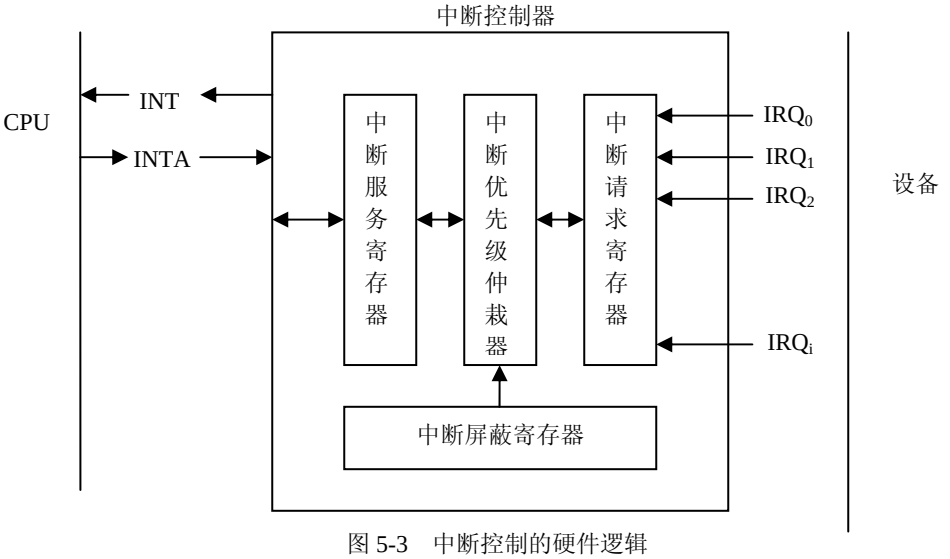


图 5-3　中断控制的硬件逻辑

2．中断控制方式的实现

要实现中断控制工作方式，设备驱动程序首先要完成中断处理程序的代码，来实现外部设备与内核之间的数据交换；其次在系统启动或驱动程序模块加载时，设备驱动程序要为其设备硬件向内核注册中断处理程序，将硬件中断信号与中断处理程序的关联注册到内核中，这是驱动程序初始化的任务之一。Linux 内核实现了一组 API 来管理中断资源。

（1）建立中断处理程序与硬件中断信号之间的关联

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

这个函数首先确认设备驱动程序申请的中断号有效，然后注册设备驱动程序的中断处理程序*handler，将*handler 与设备中断号关联。

（2）释放中断资源

```
   void free_irq(unsigned int irq, void *dev_id)
```

函数 free_irq 用于释放中断资源。如果没有别的设备向中断号 irq 注册中断处理程序，内核就禁止该中断。注意，在释放中断资源时，需要发送中断号和设备标识符，因为中断信号可以由多个设备共享，内核需要知道是哪个设备以及有多少个设备已释放了中断资源。

3．网络设备中断事件类型

在网络设备中，当设备发出中断时，可以向驱动程序报告的事件类型有：

- 收到数据包，这是最常见的中断事件。
- 发送失败，比如数据包发送超时，响应出错等。
- DMA 发送结束。

在向网络发送数据包时，一旦数据包成功地放入网络设备硬件的缓冲区，从网络设备驱动程序角度而言就已完成数据包的发送，驱动程序就释放存放数据包的 Socket Buffer，随后，将数据包发到网络传输介质的过程由网络设备硬件完成。所以在同步发送过程中（非 DMA 方式），数据包送到设备硬件缓冲区后，驱动程序就认为发送结束；但 DMA 方式为异步发送方式，需由网络设备硬件产生中断通知驱动程序本次发送操作结束。

当硬件设备缓冲区没有足够的空间容纳新的网络数据包时，驱动程序停止设备的发送队列，这样内核就不能再发送新的数据包到设备了。随着数据不断向外发送，设备缓冲区空间有效，它将产生一个中断重新启动发送队列。

这个过程的实现逻辑通常这样安排：在数据包发送代码执行前，停止发送队列，查看设备是否有足够的空间接收新的数据包，如果有，则重新启动发送队列，否则在以后以中断的方式重新启动发送队列。以下代码是实现上述逻辑的示例。

```
//网络数据包发送功能函数
static int el3_start_xmit(struct sk_buff *skb, struct net_device *dev)
{
  …
//发送数据包前调用 netif_stop_queue 函数停止设备发送队列
  netif_stop_queue(dev);
…
//如果设备缓冲区长度大于 1536 字节（以太网数据帧长度为 1536 字节），重新启动设备发送队列
if ( inw (ioaddr + TX_FREE) > 1536 )
       netif_start_queue(dev);
  else
      outw(SetTxThreshold + 1536, ioaddr + EL3_CMD);
  …
}
```

设备驱动程序用函数 netif_stop_queue 停止发送队列，禁止内核再提交新的数据；然后，驱动程序检查设备是否有足够的空间容纳 1536 个字节的数据包，如果有，驱动程序就重启发送队列，再次允许内核提交数据包，否则，驱动程序就发控制指令给设备（outw，写设备的控制寄存器），当条件满足时产生硬件中断通知内核空间重新有效。

以上是中断工作的基本原理。从学习第 2、3 章内容的经验我们可以知道，任何一个资源或硬件实体要得到内核识别，在内核中都需要有相应的代码来表示，这就是数据结构。中断资源的代码描述是什么？

5.2.3　中断在内核的实现

系统中每个中断信号线由一个唯一的标识符来标识，称为中断号。每一个硬件中断资源在内核中由一个 struct irq_desc 类型的数据结构表示。struct irq_desc 数据结构定义在 Linux 源码树型目录的 include/linux/irq 文件中。

1．硬件中断信号在内核中的表示 struct irq_desc

```
struct irq_desc {                                                    //include/linux/irq.h
  unsigned int          irq;
   …
  irq_flow_handler_t    handle_irq;
  struct irq_chip       *chip;
  struct msi_desc       *msi_desc;
```

```
    void                    *handler_data;
    void                    *chip_data;
    struct irqaction *action;
    unsigned int            status;

    unsigned int            depth;
    unsigned int            wake_depth;
    unsigned int            irq_count;
    unsigned long           last_unhandled;
    unsigned int            irqs_unhandled;
    spinlock_t              lock;
}
```

struct irq_desc 数据结构描述了硬件中断信号线的属性，一些关键的数据域如下。

- Irq：该中断信号描述符描述的中断号。
- handle_irq：函数指针，指向高层的中断事件处理程序，即由哪个函数来调度硬件的中断处理程序的执行。如果该函数指针为 NULL，默认的由内核函数 __do_IRQ 调度。
- chip：指向描述中断信号所属中断控制器芯片的数据结构指针。
- msi_desc：指向描述中断屏蔽字数据结构的指针。该结构中描述了哪些中断信号线被 CPU 屏蔽，哪些中断信号可以获取 CPU 的响应。
- action：指向描述中断信号与中断处理程序对应关系数据结构的指针。
- status：表示当前该中断信号的状态。比如，被屏蔽、响应或者在等待被服务。
- depth：中断处理程序的执行可以嵌套。也即当 CPU 正在执行一个中断的处理程序时，一个优先级别更高的中断信号线产生了中断，CPU 在中断程序中转去执行更高级别中断的服务程序，称为中断嵌套。depth 数据域描述了中断可以嵌套的层数（或称为深度）。
- wake_depth 唤醒被打断的嵌套中断服务程序的深度。即恢复嵌套被打断的中断程序的执行，可恢复到嵌套的哪个层次。
- irq_count：本中断被别的中断打断的次数。
- last_unhandled：本中断最后一次未被 CPU 响应的时间。
- irqs_unhandled：本中断未被 CPU 响应的次数。

从上述描述我们可以看到，在 struct irq_desc 数据结构中，除了描述了硬件中断信号的属性外，还包含了该硬件中断在系统运行过程中的管理、统计，以及在每个 CPU 上管理的信息。系统中有多少个硬件中断信号线，就有多少个 struct irq_desc 结构的实例（硬件中断信号的数量是与 CPU 硬件平台相关的，如 X86 有 15 个中断信号，PPC860 有 32 个中断信号等），也即 struct irq_desc 数据结构实例是硬件中断信号在 Linux 内核中的表示。所有 struct irq_desc 结构实例存放在 irq_desc 数组中（数组名与其结构名一样）组成中断向量表。Linux 内核的中断向量表也定义在 include/linux/irq.h 文件中。

```
extern  struct irq_desc        irq_desc[NR_IRQS];
```

NR_IRQS 代表了 Linux 内核中断向量表可存放的中断向量的个数，该数据与具体的 CPU 平台相关。

在 Linux 内核中除了要描述中断信号外，硬件中断信号与该中断信号产生的中断事件

的处理程序需要关联起来，这样 CPU 在接到中断信号请求后，才知道应执行什么函数，这个关系在 Linux 内核中由 struct irqaction 数据结构描述。struct irqaction 定义在 include/linux/interrupt.h 文件中。

2．中断信号与中断处理程序的关联

```
struct irqaction {                                          //include/linux/interrupt.h
  irq_handler_t handler;
  unsigned long flags;
  cpumask_t mask;
  const char *name;
  void *dev_id;
  struct irqaction *next;
  int irq;
  struct proc_dir_entry *dir;
};
```

每个中断信号可以有多个 struct irqaction 数据结构的实例。在一个中断信号由多个设备共享的情况下，每个设备的中断处理程序都不同，因此对同一中断信号线需要产生多个 struct irqaction 数据结构的实例来描述中断处理程序与中断信号的关联。struct irqaction 数据结构的关键数据域为：

- *handler：函数指针，指向设备驱动程序实现的中断处理函数。当内核接收到中断信号为 irq 的硬件设备产生的中断时，它就执行与 irq 对应的 handler 所指向的函数。
- Flags：中断类型的标志。Linux 中定义的主要中断类型如表 5-1 所示。

表 5-1　Linux 中定义的主要中断类型

中断类型描述符	含　义
SA_SHIRQ	当 flags 设置为该标志时，设备驱动程序可以处理共享中断
SA_SAMPLE_RANDON	设备将自己标记为一个随机事件源。这可以帮助内核产生内部使用的随机数
SA_INTERRUPT	CPU 运行中断处理程序时，会禁止本地 CPU 上的其他中断。这种类型的中断应分配给中断处理程序完成得非常快的中断

还有一些其他的中断类型，但都很少使用或没有被使用。

- Mask：中断屏蔽位标识。描述了在执行本中断的服务程序时，要屏蔽哪些中断信号。
- Name：产生中断信号的设备名，以字符串形式描述。一个设备驱动程序可以同时为多个设备使用，所以它需要一个设备标识符来区分不同的设备，指明当前服务的是哪个设备。
- dev_id：设备在内核中的数据结构实例，对于网络设备，就是指向 struct net_device 数据结构实例的指针。之所以将 dev_id 定义为 void *类型，是因为系统中不止一种类型设备会使用中断与 CPU 交换数据。各种设备使用的数据结构不同。
- irq：产生中断事件的中断信号，即由哪一条中断信号线向 CPU 发出了中断请求。
- *next：所有共享同一中断信号的设备，中断服务处理程序与中断信号关联的 struct irqaction 数据结构实例，用该指针链接成链表。当中断信号为 irq 的中断信号产生中断请求时，如果该链表不为空，CPU 需要首先在链表中查询是哪个设备产生的中断，再执行正确的中断服务程序。

3．中断向量表的组织

现在我们知道了内核如何描述硬件中断信号，如何描述中断信号与中断处理程序的关联，将以上两个数据结构组织在一起形成的结构，就称为中断向量表，其内部组件如图 5-4 所示。

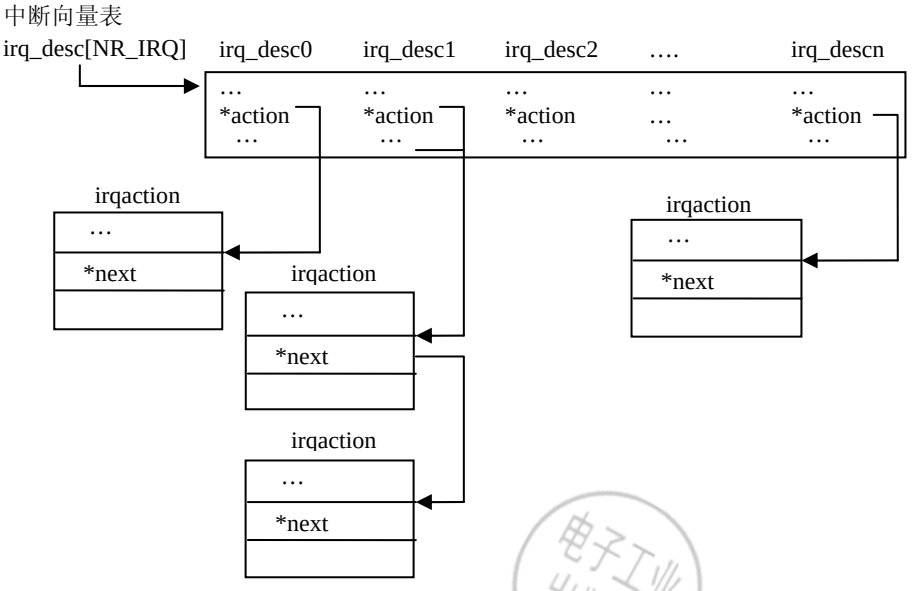


图 5-4　中断向量表的组织形式

图 5-4 中 irq_desc1 成员描述的就是一个中断信号由多个设备共享时，在中断向量表中的组织方式。图 5-4 表示了这样几层含义：

- 系统中有多少个实际的硬件中断信号线，就有多少个 strict irq_desc 数据结构实例，这些实例放在中断向量表 irq_desc[NR_IRQ]数组中。
- 当设备驱动程序调用 irq_request 函数向内核注册自己的中断处理程序时，内核产生一个 struct irqaction 数据结构的实体，描述中断处理程序与中断号的关联。
- 内核将 struct irqaction 数据结构实例挂接到中断向量表对应中断号的描述符中。
- 设备产生中断后将自己的中断号通过数据总线传给 CPU，CPU 以中断号为索引查询中断向量表（irq_desc 数组），找到正确的中断描述符，执行挂接在中断描述符上的中断处理程序。

5.2.4　软件中断

CPU 接到中断请求转去执行中断处理程序时，内核代码在一个特殊的现场（context）中执行，称为中断执行现场（interrupt context）。在中断执行现场中，CPU 会禁止其他所有中断，这意味着 CPU 在执行中断服务程序时不能响应其他中断请求，也不能执行其他程序。所以中断处理程序的执行应尽可能快，如果中断被禁止的时间太长，可能会丢失其他硬件

的数据。

网络数据的收发比较复杂，因为数据包需要经过 TCP/IP 协议栈各层协议实例的处理，过程较多。但不是所有这些处理都需要在中断服务程序中进行，一部分操作可以放到以后再执行，即放到中断后半段（Bottom half）来执行。在内核中有几种技术来实现中断后半段,而在网络子系统中，主要使用的是软件中断（softirq）来完成网络数据包接收后半段的工作。网络数据包的收发过程离不开中断后半段机制，在本节我们就描述 Linux 内核中，网络子系统中断后半段，也即软件中断的实现机制。实现 Bottom half 的机制分为以下几个步骤：

① 将 Bottom half 适当分类。

② 将 Bottom half 索引号与处理程序关联，向内核注册 Bottom half 和它们的处理程序。

③ 在适当的时候调度 Bottom half 执行。

④ 通知内核执行 Bottom half 的处理程序。

1．软件中断的类型

当前 Linux 版本的一个重大改进是，一旦涉及中断，中断处理程序只执行最紧急的任务，比如读入硬件缓冲区中的数据，清除设备硬件状态等；设备硬件状态清除后，设备就可以开始接收新数据，接着再次允许设备硬件中断；大部分对数据的处理都放在 Bottom half 中来完成，这样极大提高了内核响应硬件中断请求的速度，提高了硬件的吞吐量。以软件中断实现的 bottom half 是多线程处理方式，多个线程可以同时执行多个类型的软件中断实例，除此之外同一软件中断的实例还可以同时运行在不同的 CPU 上，唯一的限制是每种软件中断在同一个 CPU 上，一次只能运行一个实例。

Linux 内核定义了 6 种类型的软件中断，如下。

```
enum{
  HI_SOFTIRQ = 0,
  TIMER_SOFTIRQ,
  NET_TX_SOFTIRQ,
  NET_RX_SOFTIRQ,
  SCSI_SOFTIRQ,
  TASKLET_SOFTIRQ,
};
```

其中两种软件中断 NET_TX_SOFTIRQ/NET_RX_SOFTIRQ 用在网络子系统中。

软件中断处理的优点是，内核不允许在同一 CPU 上同时运行相同类型软件中断的实例，这样极大减少了管理并发访问控制的工作量。同一软件中断实例可以在不同的 CPU 上同时运行，类似网络数据包处理这样较费时的任务，多个数据包可以在多个 CPU 上同时处理，会极大地提高处理速度。软件中断的处理函数只需控制多个 CPU 对共享数据的并发访问。

从前面学习的硬件中断实现过程可以得到类似的软件中断实现经验：其一，内核中应有描述软件中断的数据结构；其二，各子系统应实现自己的软件中断处理程序；其三，将软件中断处理程序与 6 个软件中断索引号相关联，注册到内核；其四，内核合理调度软件中断执行。以下我们就沿着这样的线索分析软件中断在内核的实现过程，在第 6 章我们会分析网络子系统软件中断的处理功能。

2．内核中描述软件中断的数据结构

软件中断的数据结构在内核中的实现代码几乎都定义在 kernel/softirq.c 文件中。描述软件中断的数据结构为：

```
struct softirq_action
{
  void  (*action)(struct softirq_action *);
};
```

可以看到 struct softirq_action 数据结构中的唯一数据域是指向软件中断处理函数的指针。管理软件中断的向量表是 struct softirq_action 数据结构类型的数组。

```
static struct softirq_action softirq_vec[NR_SOFTIRQS] __cacheline_aligned_in_smp
```

3．注册软件中断处理程序

将软件中断处理程序注册到内核，要使用 open_softirq 函数，其函数声明如下：

```
void open_softirq(int nr, void(*action)(struct softirq_action *), void *data)
```

传给软件中断注册函数 open_softirq 的参数，描述了将软件中断处理程序与软件中断索引号关联需要的信息。

- int nr：软件中断类型的索引号（如 NET_TX_SOFTIRQ）。
- void(*action)(struct softirq_action*)：指向软件中断处理函数的指针。
- void *data：传给软件中断处理程序的数据。

内核子系统调用 open_softirq 函数后，open_softirq 函数以软件中断号为索引，将软件中断处理程序插入软件中断向量表 softirq_vec，这样软件中断处理程序就与软件中断号关联起来。

这个注册过程应在内核启动，初始化各子系统时就完成，这样内核的中断系统才能处于正常工作状态。例如网络子系统接收数据包的软件中断处理程序为 net_rx_action，发送数据包的软件中断处理程序为 net_tx_action。它们都在网络子系统的初始化函数 net_dev_init 中，分别注册给了索引号为 NET_RX_SOFTIRQ 和 NET_TX_SOFTIRQ 的软件中断。

4．调度软件中断的时刻

软件中断是实现硬件中断后半段的一种机制，调度软件中断来执行的时刻最直接的地方就应是硬件中断执行完后立即调度。除此之外内核在其他几种例程的执行过程中，也会查看是否有中断的后半段在等待被调度执行，如果有，内核就用 do_softirq（定义在 kernel/softirq.c 文件中）方法来调度执行软件中断，内核会在以下几处检查是否有软件中断等待执行。

（1）do_IRQ

当内核接到硬件中断请求时，它用函数 do_IRQ 来调度执行硬件中断的处理程序，因为大量的软件中断是由硬件中断处理程序传递的，所以在 do_IRQ 执行后就可能有软件中断等待被调度，这时立即调度软件中断执行所产生的延迟最小，例如在各种硬件中断、时钟中断产生后常常立即执行 do_softirq。

do_IRQ 函数实现与具体的 CPU 体系结构有关，定义在与 CPU 体系结构相关的文件 arch/arch-name/kernel/irq.c 中，例如 x86 CPU 实现的 do_IRQ 函数定义在 arch/x86/kernel/

irq_32.c 文件中。它的实现流程如下：

```
unsigned int do_IRQ(struct pt_regs *regs)
{
   struct pt_regs *old_regs;
   int overflow;
   unsigned vector = ~regs->orig_ax;
   struct irq_desc *desc;
   unsigned irq;
//函数首先保存原执行程序的寄存器内容，获取中断号，当前 CPU 中断向量及中断号的描述符 desc
old_regs = set_irq_regs(regs);
   irq_enter();
   irq = __get_cpu_var(vector_irq)[vector];
   …
   desc = irq_to_desc(irq);
   …
   //执行相应中断号的中断处理程序
         desc->handle_irq(irq, desc);
  …
  //退出中断，恢复原寄存器值
   irq_exit();
   set_irq_regs(old_regs);
   return 1;
}
```

从以上的流程并看不出在 do_IRQ 函数中调用了 do_softirq 函数，从 Linux 2.6 版本以后，对 do_softirq 函数的调用嵌入在 irq_exit 函数中。

```
#ifdef __ARCH_IRQ_EXIT_IRQS_DISABLED
# define invoke_softirq()      __do_softirq()              //定义 invoke_softirq 为调用软件中断
#else                                                     //函数 do_softirq
# define invoke_softirq()      do_softirq()
#endif
//退出中断执行现场，如果有软件中断等待执行，且满足执行条件，则执行软件中断
void irq_exit(void)
{
//执行软件中断的前提条件是当前不在中断执行现场，当前 CPU 上有等待执行的软件中断挂起
   if (!in_interrupt() && local_softirq_pending())
         invoke_softirq();
    …
}
```

（2）从硬件中断和异常（包括系统调用）返回时

从硬件中断返回时，硬件中断的处理程序已执行完，在硬件中断中我们只处理最紧急的任务，所以当它返回时，大量可以推迟的处理过程就留给软件中断来完成。

（3）在 CPU 上打开软件中断时（local_bh_enable）

当在 CPU 上重新打开软件中断时，以前如有挂起等待被调度的软件中断，就用 do_softirq 函数调度它们执行。

（4）内核线程 ksoftirqd_cpun

为了防止某种软件中断独占 CPU（这种情况在网络流量高峰期时极有可能发生），NET_TX_SOFTIRQ/NET_RX_SOFTIRQ 比其他用户进程的优先级高，Linux 的开发人员使用了一组新的与每个 CPU 相关的线程，它们分别为：ksoftirqd_cpu0， ksoftirqd_cpu1 等，来避免这种情况的发生。内核线程在本节的后面将会介绍。

（5）其他调用软件中断的地方

netif_rx_ni 函数我们会在第 6 章介绍，它是 netif_rx 的第一部分，这两个函数合起来将数据包从网络设备硬件缓冲区复制到内核的 Socket Buffer 中。这时，对内核而言就有数据包需要处理，可调度软件中断来完成。

嵌在内核中的流量管理器（定义在 net/core/pktgen.c 文件中）也调用 do_softirq。

5．标记传递的软件中断

硬件中断结束后，如果该硬件中断要发送中断服务程序的后半段来处理数据，它们调用以下几个函数来设置一个位图标志。位图标志的每一位代表一种软件中断，如果某一位被设置，则代表有对应类型的软件中断挂起等待执行。

（1）__raise_softirq_irqoff

此函数设置要运行的软件中断在位图中的标志位，稍后内核查看到对应软件中断的标志位被设置后，与它相关的软件中断就会被调度到 CPU 上执行。

```
#define __raise_softirq_irqoff(nr) do { or_softirq_pending(1UL << (nr)); } while (0)

#define or_softirq_pending(x)  (local_softirq_pending() |= (x))
```

函数__raise_softirq_irqoff 读出原软件中断的位图标志，将硬件新传递的软件中断与原位图标志做“或”操作，标志有新的软件中断在等待执行。

（2）raise_softirq_irqoff

这是__raise_softirq_irqoff 的包装函数，调用__raise_softirq_irqoff 设置新传递的软件中断，如果当前在硬件中断或软件中断的执行现场，等到中断服务程序或软件中断返回后，新传递的软件中断会在中断退出函数中被调度执行。如果当前不是在以上执行现场，我们显示的调度内核线程，保证软件中断在不久会被调度执行。

```
inline void raise_softirq_irqoff(unsigned int nr)
{
  __raise_softirq_irqoff(nr);

  if (!in_interrupt())                          //当前不在中断执行现场
       wakeup_softirqd();                       //唤醒内核线程
}
```

（3）raise_softirq

它是 raise_softirq_irq_off 的包装函数，传递软件中断时，需在关闭了硬件中断的情况下执行 raise_softirq_irq_off。

6．执行软件中断

通过前面的叙述我们知道，软件中断在系统运行的多个时刻由 do_softirq 函数调度执行，do_softirq 函数读取软件中断的位图标志，将位图中所有设置了等待执行的软件中断依次调用执行。

（1）do_softirq 的功能

调度执行软件中断的前提条件是：当前没有任何硬件中断或正在执行别的软件中断，否则 do_softirq 停止运行，什么也不做。

如以上条件满足，do_softirq 将当前设置的软件中断位图标志保存在局部变量 pending

中，并逐个执行这些软件中断。

```
#ifndef __ARCH_HAS_DO_SOFTIRQ
asmlinkage void do_softirq(void)                    //kernel/softirq.c
{
  __u32 pending;
  unsigned long flags;

  if (in_interrupt())                               //当前是否在中断执行现场
       return;                                      //如果是则返回，do_softirq 什么都不做

  local_irq_save(flags);                            //保存当前软件中断位图设置

  pending = local_softirq_pending();                //读软件中断的位图标志

  if (pending)                                      //如位图标志中有软件中断等待调度执行,执行软件中断

       __do_softirq();
  local_irq_restore(flags);                         //恢复原软件中断位图标志
}
#endif
```

从上述的代码中可以看到，do_softirq 在关中断的条件下运行，但实际上只在操作软件中断位图标志时才关中断，你会看到一旦进入__do_softirq 后，中断就立即打开了。

有的软件中断在 do_softirq 运行期间可以多次调度执行，因为在运行软件中断处理程序时，硬件中断是打开的，所以软件中断可能被硬件中断打断。硬件中断服务程序在执行时可以修改软件中断位图标志，传递新的软件中断等待调度。因此在__do_softirq 打开中断之前，do_softirq 保存当前软件中断位图标志（在局部变量 pending 中），清除 CPU 软件中断位图设置（保存在 softnet_data 数据结构中），然后按照局部变量 pending 中保存的位图标志一个一个地执行软件中断。

do_softirq 定义在 kernel/softirq.c 中，不同 CPU 体系结构的实现代码可以重载这段代码（但执行软件中断最终还是调用__do_softirq），这就是为什么 kernel/softirq.c 中定义的 do_softirq 由__ARCH_HAS_DO_SOFTIRQ 函数包装着。有些体系结构的 CPU，包括 i386（参见 arch/i386/kernel/irq.c）都定义了它们自己的 do_softirq，这些体系结构的 CPU 一般使用 4KB 的栈（而不是 8KB），并使用 4KB 的栈实现硬件中断和软件中断。

（2）__do_softirq 的功能

```
#define MAX_SOFTIRQ_RESTART 10                      //do_softirq 能执行的最长时间

asmlinkage void __do_softirq(void)
{
  struct softirq_action *h;
  __u32 pending;
  int max_restart = MAX_SOFTIRQ_RESTART;
  int cpu;
  pending = local_softirq_pending();                //读取软件中断设置的位图标志
  …
restart:
  //在开中断之前，清 CPU 的软件中断位图标志，以便硬件中断可以传递新的软件中断，随后开中断
set_softirq_pending(0);
  local_irq_enable();
  //获取软件中断向量表，以便找到软件中断的处理函数，如果有挂起的软件中断，依次调用软件中断的处理函数（由 action
  //指针指向）执行
```

```
    h = softirq_vec;
    do {
          if (pending & 1) {
          …
                h->action(h);
          …
                }

    …
                }
          h++;
          pending >>= 1;
    while (pending);
    //原挂起的软件中断执行完后，如仍有软件中断等待被调度执行，并且 do_softirq 还没用完执行时间，则重复前面的过程
    pending = local_softirq_pending();
    if (pending && --max_restart)
          goto restart;
    //如果 do_softirq 已用完它的执行时间，则唤醒内核线程
    if (pending)
          wakeup_softirqd();
    …
}
```

一旦所有挂起的软件中断的处理程序都被执行了，__do_softirq 就会查看是否还有新的软件中断等待被调度执行，即使只有一个软件中断挂起，整个过程会再次重复。但__do_softirq 只能连续执行 MAX_SOFTIRQ_RESTART 给定的时间。

使用 MAX_SOFTIRQ_RESTART 是为了避免单一的网络类型中断一直持续传递软件中断调度执行，别的硬件中断传递的软件中断不能被执行。

在__do_softirq 中，每一次一个软件中断被调度执行后，其对应的标志位就从本地变量 pending 中清除，当位图清空时，调度软件中断的循环结束。最后，如果 do_softirq 的执行时间已达到 MAX_SOFTIRQ_RESTART，do_softirq 必须返回时，还有未执行的软件中断在等待被调度，它就唤醒内核线程 ksoftirqd 在以后来执行它们。因为 do_softirq 在内核中有多处被调用，可能在一段时间后，内核线程 ksoftirqd 被调度执行前系统又调用了 do_softirq 来处理那些挂起的软件中断。

7．内核线程 ksoftirqd

内核线程的任务是作为后备来查看是否还有前面过程中未执行的软件中断在等待调度；如果有，在将 CPU 交给别的活动之前尽量调度软件中断执行。每个 CPU 都有一个内核线程，分别为 ksoftirqd_cpu0、ksoftirqd_cpu1 等。

与各 CPU 相关的内核线程 ksoftirqd 的任务非常简单，它们都定义在 kernel/softirq.c 文件中。

```
static int ksoftirqd(void * __bind_cpu)
{
  …
        while (local_softirq_pending()) {
              …
              do_softirq();
              …
        }
…
}
```

内核中进程的优先级一共划分成-20（最高）到+19（最低），内核线程的优先级为 19。这样做的目的就是避免在网络流量高峰期频繁传递的 NET_RX_SOFTIRQ 软件中断独占 CPU。内核线程一旦启动，它就循环调用 do_softirq 直到以下一个条件被满足：

- 已没有软件中断等待调度了（local_softirq_pending 返回 FALSE）。在这种情况下，函数 do-softirq 将线程的状态设为 TASK_INTERRUPTIBLE，调用 schedule 释放 CPU；线程可以由 wakeup_softirqd 唤醒，唤醒过程可以在__do_softirq 中，也可以在 raise_softirq_irqoff 中。
- 线程执行的时间已到期，必须释放 CPU。时钟中断处理程序会设置 need_resched 标志来通知当前进程/线程它们的执行时间片已到期，这时 ksoftirqd 释放 CPU，保持它的状态为 TASK_RUNNING，以后线程的执行可以被恢复。

8．网络子系统的软件中断

网络子系统分配了两个不同的软件中断：NET_RX_SOFTIRQ，处理输入数据包；NET_TX_SOFTIRQ，处理输出数据包。它们都是在 net_dev_init 函数中初始化的（参见 4.2.3 节）。

因为同一类型软件中断的不同实例可以在不同的 CPU 上同时运行，所以网络代码的延迟都很小，其执行时间可以度量。两个网络软件中断的优先级比一般类型的 tasklet(TASKLET_SOFTIRQ)的优先级高，但低于其他类型 tasklet 的软件中断优先级（HI_SOFTIRQ）。这种优先级的安排在网络流量的高峰期也能保证其他高优先级的任务及时得到响应。

5.3　网络设备驱动程序的实现

现在我们已讲解了编写网络设备驱动程序需要的基本知识。在本节，我们将以 Linux 内核提供的网络设备驱动程序样本 isa-skeletion 为例，来讲解网络设备驱动程序的实现。

图 5-1 给出了网络设备驱动程序的构架，本节我们首先描述网络设备驱动程序中各函数应完成的功能，再用示例来说明如何实现这些函数功能。

5.3.1　网络适配器的初始化

第 4 章中我们已经详细讲解了网络子系统和网络设备的初始化过程，在实现网络设备驱动程序时会运用到这些基础知识。Linux 系统在具备网络功能之前，需要识别系统中的网络设备，将网络设备对应的数据结构实例加入到系统的全局网络设备管理链表中。因此，驱动程序要提供一个 xxx_probe 函数来负责探测驱动程序自己的设备，并初始化它的 struct net_device 数据结构实例。

在大多数网络设备驱动程序中，探测函数分成两个部分实现：一是网络设备驱动程序中执行实际设备探测的函数，一般命名为 xxx_probe1；二是实际探测函数的包装函数，一般命名为 xxx_probe，向实际探测函数传递正确的参数，指定探测函数在某一确定的 I/O 端口地址上探测设备，还是在所有可能的 I/O 端口上探测设备。xxx_probe 是提供给内核调用

的自动探测函数（在 space.c 文件中），内核启动时调用该函数探测网络设备。

上述探测函数的几个实现部分，在 driver/net/isa_skeletion.c 中分别为：

- netcard_probe1：执行实际探测的函数，探测驱动程序的设备是否出现在 I/O 端口上。
- do_net_probe：netcard_probe1 的包装函数，向 netcard_probe1 传递正确的参数。
- netcard_probe：内核启动时调用的自动探测函数。

内核启动的过程如图 5-5 所示。

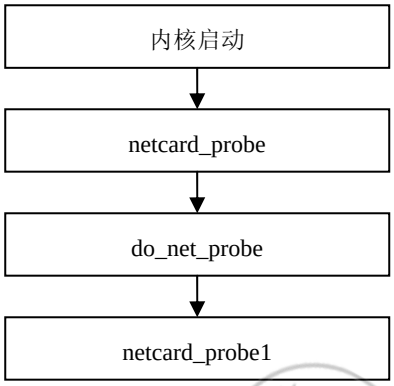


图 5-5　内核启动的过程

1. 常量及局部变量定义

```
//为设备定义设备名，在申请 I/O 端口、中断、DMA 时需要用
static const char* cardname = "netcard";
/* 以 0 结尾的 I/O 地址列表，用于探测可能的 I/O 端口地址 */
static unsigned int netcard_portlist[] __initdata =
  { 0x200, 0x240, 0x280, 0x2C0, 0x300, 0x320, 0x340, 0};
…
/* 设备的私有数据结构，用于保存设备自身的配置信息和统计信息，这里，私有数据结构没有什么实际意义，只是作为一个示例 */
struct net_local {
  struct net_device_stats stats;
  long open_time;
  spinlock_t lock;
};
/* MAC 地址的前三个字节是生产商 ID */
#define SA_ADDR0 0x00
#define SA_ADDR1 0x42
#define SA_ADDR2 0x65
```

2．探测函数的包装函数

在包装函数 do_netcard_probe 中区分两种探测网卡的情况。

- 指定基地址探测，这时将前面创建的网络设备的 struct net_device 数据结构实例作为参数传给探测函数，调用者可以使用 struct net_device 数据结构的 base_addr 数据域事先为 I/O 端口指定基地址，如在指定地址没有探测到设备，函数返回错误代码-ENODEV。基地址可用以下两种方式指定。
  - ◆ 驱动程序以模块的方式装载：在装载模块时发送参数，给出 I/O 端口基地址

（如 io=0x280）。这时该参数应在驱动程序的 init_module 方法中传给网络设备的 struct net_device 数据结构实例，供搜索网络设备之用。

- ◆ 驱动程序静态编译到内核：这时我们可以在系统中通过命令行传参数。这些参数存放在 dev_boot_setup 列表中，通过 netdev_boot_setup_check 方法放入 struct net_device 数据结构。

- 在一个已知的地址范围内搜索。一个网络适配器通常可以工作在几个 I/O 端口基地址上（插入不同的插槽，某个端口地址已被别的设备占用），如果调用自动探测函数时没有指定具体要在哪个 I/O 端口地址上探测，就按照事先定义在地址列表中的 I/O 端口地址逐一地探测，如果所有的 I/O 端口基地址处都没有发现设备，就返回错误代码-ENODEV。

```
/*探测函数 netcard_probe1 的包装函数，探测所有地址或某个指定地址，或不探测*/
static int __init do_netcard_probe(struct net_device *dev)
{
  int i;
  int base_addr = dev->base_addr;
  int irq = dev->irq;
/* 探测指定 I/O 端口地址 */
  if (base_addr > 0x1ff)
        return netcard_probe1(dev, base_addr);
//* 不探测 */
  else if (base_addr != 0)
        return -ENXIO;
/探测所有 I/O 端口地址处有无网络设备*/
  for (i = 0; netcard_portlist[i]; i++) {
        int ioaddr = netcard_portlist[i];
        if (netcard_probe1(dev, ioaddr) == 0)
              return 0;
        dev->irq = irq;
  }
  return -ENODEV;
}
```

3．由内核调用的自动探测函数

函数 netcard_probe 的功能是：首先为网络设备的 struct net_device 数据结构实例分配空间，然后查看内核启动时有无命令行参数给出设备的配置信息，如有，则将其保存在 struct net_device 数据结构的相关数据域中，调用探测函数查看是否有此硬件。

```
/*自动探测函数*/
struct net_device * __init netcard_probe(int unit)
{
/*分配网络设备的 struct net_device 数据结构实例*/
  struct net_device *dev = alloc_etherdev(sizeof(struct net_local));
  int err;
  if (!dev)
        return ERR_PTR(-ENOMEM);
/*为网络设备分配设备名*/
  sprintf(dev->name, "eth%d", unit);
/*查看系统启动时有无命令行参数*/
  netdev_boot_setup_check(dev);
/*调用探测函数的包装函数，来探测设备*/
  err = do_netcard_probe(dev);
/*如探测到设备，返回指向设备 net_struct 数据结构实例的指针，否则释放 dev*/
```

```
    if (err)
        goto out;
    return dev;
out:
    free_netdev(dev);
    return ERR_PTR(err);
}
```

4．实际实现探测的函数

在网络设备的自动探测过程中，一旦我们选择了要探测网络设备的 I/O 端口基地址，实际探测函数 netcard_probe1 就测试指定设备是否出现在该地址上。为了实现这个测试，函数必须查看指定的网卡，因此对网络设备的访问应限制在特定的 I/O 端口，以保证读的不是别的适配器信息。这时尚不知道我们要探测的网络设备是否出现在指定的 I/O 端口基地址处。

一种识别网络适配器的简单方式是将网络设备的 MAC 地址与生产商的标识符做比较。每个网络适配器都有一个唯一的 MAC 地址，其中前三个字节是生产商的标识符，该标识符需要与我们要搜索的网络设备生产商标识符相吻合。除此之外需做的比较要视具体的网络设备硬件的要求而定。

如果探测到网络设备出现在指定的 I/O 端口处，当前探测成功的网络设备的 I/O 端口基地址就存入 struct net_device 数据结构实例的 dev->base_addr 数据域，然后初始化网络设备，以 ioaddr 为起始地址的 I/O 端口通过 request_region 作为资源分配给探测到的网络设备，以保证没有别的设备访问这个区域。网络适配器初始化过程可以分成以下 3 个阶段。

① 如果网络适配器不支持动态中断分配，是通过跳线来决定中断信号线的，这时应为其分配中断资源。内核提供了搜索中断的方法，首先驱动程序调用 probe_irq_on 函数，它返回一个位图变量，其值是系统尚未分配的中断号的标识。这时驱动程序应控制硬件产生一个中断，然后关中断。调用函数 probe_irq_off，将 probe_irq_on 的返回值作为参数传给 probe_irq_off。如这个过程期间没有中断产生，probe_irq_off 返回 0 值；如果有中断产生，probe_irq_off 的返回值就是刚才产生中断的中断号。接着用函数 request_irq 为网络设备分配中断资源。另外，DMA 通道也在此时分配。

现代网络适配器都无须用跳线指定中断信号线和 DMA 通道，所以这两种资源不在这里分配，而是在打开（open）设备的方法中分配。

② 分配系统资源后，要为网络设备的私有数据结构申请内存空间，并初始化私有数据结构。

③ 最后，初始化网络设备的 struct net_device 数据结构实例中的函数指针，并注册设备。

```
/*实际的网络设备探测和初始化函数*/
static int __init netcard_probe1(struct net_device *dev, int ioaddr)
{
    struct net_local *np;
    static unsigned version_printed;
    int i;
    int err = -ENODEV;
/*确定探测的 I/O 端口地址是否已被其他设备占用，如果没有，申请这个端口的地址*/
if (!request_region(ioaddr, NETCARD_IO_EXTENT, cardname))
        return -EBUSY;
/*从以太网适配器读入 MAC 地址，比较前三个字节是否与生产商标识符相同，以确定在该 I/O 端口地址处的是要探测的网络设备
```

```
   if (inb(ioaddr + 0) != SA_ADDR0
         ||    inb(ioaddr + 1) != SA_ADDR1
         ||    inb(ioaddr + 2) != SA_ADDR2)
         goto out;
/*如果探测到网络设备，将 I/O 端口基地址填入 dev 的 base_addr 数据域
   dev->base_addr = ioaddr;
/*读入网络设备的硬件地址（MAC 地址），填入 dev 的 dev_addr[]数据域*/
   for (i = 0; i < 6; i++)
         dev->dev_addr[i] = inb (ioaddr + i);
   err = -EAGAIN;
#ifdef jumpered_interrupts
/*如果该网络适配器支持跳线分配中断，在这里分配中断向量，跳线中断不由板卡报告，所以需要自动产生 IRQ 来发现中断号*/
   if (dev->irq == -1)
         ;                                                  /*什么都不做，用户程序会设置中断号 */
   else if (dev->irq < 2) {                                 /* 自动探测中断号 */
         unsigned long irq_mask = probe_irq_on();
/*在这里触发一个中断，让硬件产生一个中断，关中断，读回产生中断的中断号*/
         dev->irq = probe_irq_off(irq_mask);

/*为设备分配中断资源*/
   {
         int irqval = request_irq(dev->irq, &net_interrupt, 0, cardname, dev);
         …
   }
#endif
#ifdef jumpered_dma
/*如果该网络设备使用跳线分配 DMA 通道，在这里应探测使用的哪个 DMA 通道，并分配 DMA 资源*/
   if (dev->dma == 0) {
         if (request_dma(dev->dma, cardname)) {
               printk("DMA %d allocation failed.\n", dev->dma);
               goto out1;
         } else
               printk(", assigned DMA %d.\n", dev->dma);
   } else {
         s
…
#endif  /* jumpered DMA */
/*初始化设备的私有数据结构*/
   np = netdev_priv(dev);
   spin_lock_init(&np->lock);
/*初始在网络设备驱动程序方法的函数指针*/
   dev->open= net_open;
   dev->stop= net_close;
   dev->hard_start_xmit= net_send_packet;
   dev->get_stats= net_get_stats;
   dev->set_multicast_list = &set_multicast_list;

    dev->tx_timeout = &net_tx_timeout;
    dev->watchdog_timeo= MY_TX_TIMEOUT;
/*注册网络设备*/
   err = register_netdev(dev);
   …
}
```

5.3.2 网络设备活动功能函数

当内核探测到网络设备，为网络设备创建了 struct net_device 数据结构实例，并注册到系统中后，这时网络设备还不能开始收发数据包，网络设备还需激活才能开始工作。这一节将介绍实现打开/关闭网络设备、收发数据包等关于网络设备活动功能的实现。

1．open 和 stop 方法

在网络设备可以发送数据包之前，内核必须打开网络接口，并为网络接口分配 IP 地址。引起内核打开或关闭接口活动的是来自用户地址空间的命令 ifconfig。当使用 ifconfig 给网络接口分配地址时，它完成以下两个任务。

- 用 ioctl(SIOCSIFADDR)（Socket I/O Contreal Set Interface Address）来为网络接口分配地址。
- 用 ioctl(SIOCSIFFLAGS)（Socket I/O Contreal Set Interface Flags）来设置 dev->flags 标志的 IFF_UP,以激活接口。

对网络设备而言，执行 ioctl(SIOCISFADDR)命令时，它无须做任何操作，这时也没有调用任何驱动程序的函数，这个过程由内核完成（dev->do_ioctl），其后的命令：ioctl(SIOCSIFFLAGS)就会调用驱动程序的 open 方法。

类似的，在停止网络设备时，ifconfig 使用 ioctl(SIOCISFFLAGS)来清除 IFF_UP 标志位，并调用驱动程序的 stop 方法。

（1）open 方法

与各种设备驱动程序一样，在实际的数据发送任务开始之前，网络设备驱动程序必须完成一系列的常规任务，来初始化和激活网络适配器。open 方法在打开设备时申请一切网络设备活动所需要的系统资源，当所有的系统资源分配成功时，将相关的值写入适配器的寄存器来初始化适配器硬件，并激活网络设备，这时设备就可以开始正常工作了。当一切执行成功时，open 函数返回 0。

网络设备驱动程序在打开网络设备时需要完成的一系列操作为：

- 在网络接口可以与外界通信前，将设备硬件地址（MAC 地址）从硬件设备中复制到 dev->dev_addr 数据域中。
- 一旦网络接口准备好，可以开始传发/接收数据了，要激活网络接口硬件发送队列（这样网络接口才能接收由内核传来的数据包），这个过程通过调用函数 netif_start_queue 完成。

实现 open 方法的通用模板如下：

```
int xxx_open(struct net_device *dev)
/*申请各种需要的系统资源，如中断，I/O 端口等*/
request_region();
  request_irq();
…
/*复制硬件地址，一般从网络接口中的 EEPROM 中读取网络适配器的硬件地址*/
memcpy(dev->dev_addr, src_addr, length);
…
/*激活硬件传发队列*/
netif_start_queue(dev);
return 0;
```

接下来我们看看在 isa_skeletion.c 中实现的 open 方法。

```
/*打开/初始化网络适配器.在每次打开设备时，该函数应将所有的条目更新*/
static int net_open(struct net_device *dev)
{
  struct net_local *np = netdev_priv(dev);
  int ioaddr = dev->base_addr;
/*为设备申请中断资源，针对可以动态分配中断号的新网络设备*/
  if (request_irq(dev->irq, &net_interrupt, 0, cardname, dev)) {
       return -EAGAIN;
  }
/*为设备分配 DMA 通道，DMA 通道总是在分配中断后分配，如失败清除*/
  if (request_dma(dev->dma, cardname)) {
       free_irq(dev->irq, dev);
       return -EAGAIN;
  }
/*复位硬件，设置网络设备的 I/O 端口地址*/
  chipset_init(dev, 1);
  outb(0x00, ioaddr);
  np->open_time = jiffies;
//设备准备好，可以开始接收发送数据，启动网络设备发送队列
  netif_start_queue(dev);
  return 0;
}
```

在这里因为没有涉及硬件操作，open 方法需要完成的功能并不多。

（2）stop 方法

在停止网络设备的 stop 方法中，释放所有在 open 方法中分配的系统资源，它只需做与 open 操作相反的步骤。stop 方法的通用模板如下：

```
int xxx_release (struct net_device *dev)
{
/*释放网络设备 I/O 端口、中断资源、DMA 通道等*/
  …
/*停止硬件发送队列，设备不能再发送网络数据包*/
  netif_stop_queue(dev);
  return 0;
}
```

在 isa_skeletion.c 中实现的 stop 方法中，我们可以看到它以与 open 方法步骤逆序的过程完成资源释放，停止网络设备的操作。

```
static int net_close(struct net_device *dev)
{
  struct net_local *lp = netdev_priv(dev);
  int ioaddr = dev->base_addr;
  lp->open_time = 0;
  //停止网络设备的硬件发送队列
  netif_stop_queue(dev);
  //在这里将发送队列中的数据全部传出并禁止接收数据，释放 DMA 通道，释放中断
  disable_dma(dev->dma);
  outw(0x00, ioaddr+0);                    //如果设备支持跳线中断，则释放物理中断线
  free_irq(dev->irq, dev);
  free_dma(dev->dma);
  return 0;
}
```

2. 数据传输

网络接口最主要的任务是数据收发，数据发送相对于数据接收要简单一些，我们先介绍数据包是如何由内核向网络发送的。

（1）数据发送函数的通用模板

发送数据是将数据包通过网络连接线路送出。无论什么时候，内核准备好要发送的数据包后（数据包存放在 Socket Buffer 中），都会调用驱动程序的 ndo_start_xmit 函数把数据包放到网络设备的硬件数据缓冲区，并启动硬件发送。ndo_start_xmit 是一个函数指针，指向驱动程序实现的数据发送函数。传给 ndo_start_xmit 的 Socket Buffer 包含了实际要传输的物理数据、协议栈的协议头；接口无须关心数据包的具体内容，也无须修改数据包的值。由第 2 章我们介绍的 struct sk_buff 数据结构组成可以知道，skb->data 给出了要发送数据的起始地址，skb->len 是数据包的长度。实现数据包发送函数的模板为：

```
int xxx_tx(struct sk_buff *skb, struct net_device *dev)
{
// 停止发送队列
 netif_stop_queue
//向网络控制器命令/状态寄存器写发送命令，等待硬件准备好
//将 Socket Buffer 传给硬件
//如果发送成功，更新统计信息，如发送的字节数
//启动发送超时计时器
//释放 Socket Buffer
return 0;
}
```

如果 ndo_start_xmit 执行成功则返回 0，这时负责发送该数据的驱动程序应尽最大努力保证数据包的发送成功，最后释放存放发送完成数据包的 Socket Buffer。如果 ndo_start_xmit 返回值非 0，则说明这次发送不成功，内核过一段时间后会重发。这时驱动程序应停止发送队列，直到发送失败错误恢复。

对 ndo_start_xmit 函数的并发访问控制由 net_device 数据结构中的 tx_global_lock 锁来保护。一旦 ndo_start_xmit 函数返回，就可以进行下一次数据发送。该函数在向硬件发了发送指令后就返回，但实际的硬件操作可能还没结束。硬件内部存放发送数据包的缓冲区空间有限，一旦网络设备硬件发送缓冲区满（有的网络设备硬件一次只能容纳一个发送数据包），驱动程序就需要通知网络子系统不要再启动别的发送，直到硬件准备好接收新的数据包为止。这个通知由 netif_stop_queue 函数完成，前面我们介绍了，这个函数的功能是停止发送队列，如果你的驱动程序停止了发送队列，一定要在适当的时候重启队列，这样网络设备才能再次接收新的数据包。重启发送队列可以用函数 netif_wake_queue 完成。这个函数的功能与 netif_start_queue 类似，除此之外它还会通知网络子系统重新启动数据包的发送过程。

现在大多数网络设备硬件中都有多个数据包的发送队列，这样可以实现更好的网络发送性能，因此这类网络设备的驱动程序也必须支持多数据包发送，但无论设备硬件的缓冲区是否能容纳多个数据包，硬件缓冲区都会有被填满的时候，一旦网络设备的硬件发送缓冲区满，不能再容纳更多的数据包时，驱动程序都应停止发送队列直到硬件缓冲区再次有效为止。

（2）数据发送函数在 isa_skeletion.c 中的实现

```
static int net_send_packet(struct sk_buff *skb, struct net_device *dev)
{
  //局部变量，np 指向设备的私有数据结构，ioaddr 为网络设备 I/O 端口基地址，length 为 Socket Buffer 长度，buf
  //指向 Socket Buffer 中存放数据包的缓冲区
  struct net_local *np = netdev_priv(dev);
  int ioaddr = dev->base_addr;
  short length = ETH_ZLEN < skb->len ? skb->len : ETH_ZLEN;
  unsigned char *buf = skb->data;
  //如果在发送数据包时发生了错误，函数应返回 1。这种情况下你不能对 Socket Buffer 做任何修改，Socket Buffer 仍
  //属于网络设备的发送队列，所以也不能修改 Socket Buffer 的数据域，也不能释放 Socket Buffer
#if TX_RING
  //以下处理针对大多数现代网络设备，发送队列可容纳多个网络数据包
  spin_lock_irq(&np->lock);
  //将数据包放入网络设备的循环发送队列，记录发送起始时间
  add_to_tx_ring(np, skb, length);
  dev->trans_start = jiffies;
  //如果网络设备的发送队列已满，告诉内核停止向设备发送新的数据包，即停止设备发送队列
  if (tx_full(dev))
        netif_stop_queue(dev);
  //当数据包发送结束的中断到达时，在这里应更新网络发送的数据包的统计信息
        spin_unlock_irq(&np->lock);
#else
  //以下代码用于老的网络设备，一次只能发送一个网络数据包，数据包直接写入网络设备硬件 I/O 端口，更新统计信息，记录
  //发送起始时间
  hardware_send_packet(ioaddr, buf, length);
  np->stats.tx_bytes += skb->len;
  dev->trans_start = jiffies;
  //如果发送出错，这里需要复位硬件，更新错误统计信息，释放网络数据包的 Socket Buffer
  if (inw(ioaddr) == 81)
        np->stats.tx_aborted_errors++;
  dev_kfree_skb (skb);
#endif
  return 0;
}
```

一个网络适配口器就是一个接口，按照给定的 MAC 协议（以太网，令牌环等）自动向网络介质发送或接收数据包，也即网络适配器有与 CPU 并行的独立控制逻辑，它们通过 I/O 端口（硬件寄存器）、中断与 CPU 交互，当 CPU 需要发送数据给网络设备时，CPU 将数据写到 I/O 端口，并向网络设备的命令寄存器写发送控制命令；相反，当网络设备需要发送数据给 CPU 时，它产生一个中断，CPU 执行中断处理程序来为网络设备服务。

dev->hard_start_xmit/ndo_start_xmit（在上例中该函数指针指向 net_send_packet 函数）负责把数据包放到网络适配器中，后者会自动将其放到网络发送介质上。当 Socket Buffer 中的数据被复制到网络适配器硬件缓冲区中，并且设置发送开始时间 dev->trans_start = jiffies，标志着开始发送。如果数据复制成功，CPU 就认为发送成功，这时 hard_start_xmit/ndo_start_xmit 应返回 0，否则返回 1，返回 1 时内核就知道数据包不能被发送。

在内核和网络设备之间发送数据时，我们需区分两种不同的技术。

- 老式的网络设备在网络适配器的硬件缓冲区中，一次只能容纳一个数据包，如果缓冲区空，数据包可以立即复制到网络设备上，内核就可以释放相应的 Socket Buffer。
- 现代大多数网络设备的工作方式不同，设备驱动程序管理着一个循环缓冲区，其中存放了 16～64 个指针，指向要发送数据包的 Socket Buffer。当一个数据包

准备好可以向外发送时，相应的 Socket Buffer 的起始地址就放入这个循环缓冲区中，一个数据包发送完，网络设备产生中断，刚发送完成的 Socket Buffer 就可以释放。

内核调用函数 hard_start_xmit/ndo_start_xmit 时，可以认为设备的循环缓冲区中至少可以容纳一个网络数据包；在此之前，可以调用函数 netif_queue_stopped 来查看网络设备发送队列是否已满。在将新的 Socket Buffer 放到网络设备发送队列之前，应查看网络设备的发送队列是否已满，如果是，网络设备驱动程序就已停止队列以防止内核继续发送数据包给设备。直到一次发送成功后，设备产生中断，在中断中如果发现网络设备发送队列有新的空间可以接收数据包，应重新启动设备发送队列。

3．数据发送超时处理

任何驱动实际网络硬件设备的驱动程序，都需考虑硬件失效的情况。驱动程序处理这种问题的方式是设置一个时钟，根据设备的特点初始化一个时间值，在向硬件发出发送指令后，启动时钟计时，如果超过事先设定的时间值后，发送仍没结束，则判断为发送过程出错，系统调用驱动程序的发送超时来处理这类错误。发送超时处理函数的主要任务是清除错误，让已在进行的发送过程正确执行，最重要的是驱动程序没有丢失由网络子系统传给它的 Socket Buffer。

（1）发送超时处理函数的模板

```
void xxx_tx_timeout(struct net_device *dev)
//将发送超时的出错信息发送给日志，记录发送出错的统计信息
dev->stats.tx_errors++;
//对硬件做相应的复位，其操作由具体的硬件决定
//启动发送队列，恢复网络发送
netif_wake_queue(dev);
return ;
```


（2）isa_skeletion 的发送超时处理

当把一个数据包传给网络设备后，还不能确定它是否能正确发送出去，可能发生数据冲突、中断丢失等错误。当这些错误发生时，我们需要对其做出处理，恢复网络发送过程。在 isa_skeletion.c 中，进行网络设备初始化（netcard_probe1 函数）时，我们初始化了 dev->watchdog_timeo 的值；数据包开始发送时，程序标志了最后一次发送开始的时间 dev->trans_start = jiffies。如果发送过程所花的时间超过了 dev->watchdog_timeo 的值，发送还没完成（即发送完成通知中断没有到），时钟狗触发执行发送超时处理函数 tx_timeout。这个方法负责分析错误，一般解决问题的方式是重启和重新初始化网络设备硬件。在重启网络设备硬件时，应尽量重传队列的数据包，将其发送出去。

```
static void net_tx_timeout(struct net_device *dev)
{
  struct net_local *np = netdev_priv(dev);
/*重初始化硬件，更新错误信息统计*/
  chipset_init(dev, 1);
  np->stats.tx_errors++;
/*如果网络设备发送队列有空间接收新的发送请求，唤醒发送队列。在重新初始化设备时已将队列中的数据全传出并清空了队列。
如果发送队列仍保持满，应在发送完成中断中唤醒队列*/
  if (!tx_full(dev))
       netif_wake_queue(dev);
```

```
}
```

4．接收数据包

网络数据包的接收比发送要复杂一些，是因为数据进入网络设备缓冲区后，驱动程序需要为新接收的数据包分配 Socket Buffer，向上层协议栈发送数据包，这些过程都是在中断处理程序中完成的。在 5.2 节中我们曾介绍，大部分网络设备驱动程序都以中断的方式来实现网络数据包的接收。接收数据包的处理函数 xxx_rx 是在硬件接收到网络数据产生中断后，从中断处理程序中调用的，这时数据已在硬件设备缓冲区中，xxx_rx 的主要任务是给接收到的数据包加入一些管理信息供上层协议代码使用，随后将数据包向上发送。

（1）接收网络数据包处理函数的模板

接收数据包处理函数需要获取的信息有：接收网络数据的设备、收到的数据长度。通常接收到的数据包长度可以通过读取网络设备状态寄存器获取，在这里我们假设用一个参数发送给接收数据包处理函数。

```
void xxx_rx(struct net_device *dev, struct xxx_packet *pkt)
{
   struct sk_buff *skb;
   struct xxx_priv *priv = netdev_priv(dev);
//为新接收的数据包分配 Socket Buffer，如果 Socket Buffer 分配失败，扔掉数据包
   skb = dev_alloc_skb(pkt->datalen + 2);
   if (!skb) {
//Socket Buffer 分配失败，更新扔掉的数据包统计信息
            goto out;
         }
//将接收到的网络数据复制到新分配的 Socket Buffer
         memcpy(skb_put(skb, pkt->datalen), pkt->data, pkt->datalen);
//初始化 skb 的部分数据域，供上层协议使用：接收数据的网络设备，收到的数据包协议，是否对数据包做校验和等，然后调用
//netif_rx 将数据包传给上层协议
         skb->dev = dev;
         skb->protocol = eth_type_trans(skb, dev);
         skb->ip_summed = CHECKSUM_UNNECESSARY;
         priv->stats.rx_packets++;                    //接收到的数据包总数
         priv->stats.rx_bytes += pkt->datalen;        //接收到的总字节数
         netif_rx(skb);                               //发送给上层协议
out:
   return;
}
```

接收数据包的第一步，是分配一个 Socket Buffer 来存放网络数据包。注意，这时申请 Socket Buffer 的内存是在中断代码的执行现场进行，所以要以 FP_ATOMIC 标志来分配；内存分配的返回值必须检查，一旦获取有效的 Socket Buffer，就将数据从网络设备硬件缓冲区复制到 Socket Buffer 中。

第二步，对 skb 必要的数据域赋值，网络层协议在处理数据包之前需要获取一些数据包的管理信息。驱动程序在将数据包上传给网络层之前，必须填写 skb->dev 和 skb->protocol 数据域，以太网类网络设备代码给出了一个公共函数（eth_type_trans）来给 skb->protocol 填入适当的值。随后，还需要告诉上层协议如何对数据包做校验和，或已对数据包做了什么样的校验和。

在这里需要注意的是，在初始化 net_device 数据结构实例时，它的 features 数据域已设置了如何处理校验和的标志位，为什么这里还要指定校验和的状态？features 是告诉内核设

备如何处理输出数据包的校验和，而不是用于输入数据包的校验和，对输入数据包校验和的处理方式必须另外标记。

第三步，驱动程序需要更新接收的数据包的统计信息，统计数据结构由多个数据构成，其中最重要的是 rx_packets、 rx_bytes、 tx_packets 和 tx_bytes，包含了接收和发送的数据包数和总的接收/发送的字节数。

最后数据包的接收过程由 netif_rx 完成，它将数据包传给上层网络协议栈，netif_rx 的实现我们将在第 6 章中讲解，它的返回值是一个整数，有以下几个值。

- NET_RX_SUCCES(0)：数据包接收成功。
- NET_RX_CN_LOW/ NET_RX_CN_MOD/ NET_RX_CN_HIGH：说明网络子系统堵塞程度。
- NET_RX_DROP：意味着丢包。

网络驱动程序需要利用这些值来判断内核的堵塞程度，一旦堵塞程度太高，应停止向内核发送数据包。

（2）在 isa_skeletion.c 中数据包接收的实现

```
static void net_rx ( struct net_device *dev)
{
  struct net_local *lp = netdev_priv (dev);
  int ioaddr = dev->base_addr;
  int boguscount = 10;

  do {
//读网络设备状态寄存器，获取当前数据包接收状态与接收数据长度
        int status = inw(ioaddr);
        int pkt_len = inw(ioaddr);
//如接收数据长度已为 0，所有接收数据包已处理完，退出循环
        if (pkt_len == 0)
              break;
//根据从状态寄存器读入的值，按数据包接收状态处理
        if (status & 0x40) {                          /* 接收有错误，更新接收错误的统计值*/
              lp->stats.rx_errors++;
              if (status & 0x20) lp->stats.rx_frame_errors++;
              if (status & 0x10) lp->stats.rx_over_errors++;
              if (status & 0x08) lp->stats.rx_crc_errors++;
              if (status & 0x04) lp->stats.rx_fifo_errors++;
        } else {
//网络数据包接收正确，分配一个新的 Socket Buffer，更新接收字节统计数。如 Socket Buffer 分配不成功，向日志发送
//错误信息，更新扔包统计信息
              struct sk_buff *skb;

              lp->stats.rx_bytes+=pkt_len;

              skb = dev_alloc_skb(pkt_len);
              if (skb == NULL) {
                    printk(KERN_NOTICE "%s: Memory squeeze, dropping packet.\n",
                           dev->name);
                    lp->stats.rx_dropped++;
                    break;
              }
              skb->dev = dev;
//将数据从设备缓冲区复制到 Socket Buffer，用 memcpy（设备缓冲区为 memory 映射方式），或从 I/O 端口读入/（设备
```

```
//以 I/O 端口方式工作）写入 skb->data 指向的 Socket Buffer 放数据的缓冲区
              memcpy(skb_put(skb,pkt_len), (void*)dev->rmem_start,
                     pkt_len);
              /* or */
              insw(ioaddr, skb->data, (pkt_len + 1) >> 1);
//数据包上传给协议栈，更新接收统计信息
              netif_rx(skb);
              lp->stats.rx_packets++;
              lp->stats.rx_bytes += pkt_len;
        }
   } while (--boguscount);
   return;
}
```

与发送数据包相反，接收数据包是内核事先无法预见的事件。网络设备接收数据包的过程与内核的操作是并行的，如我们在 5.2 节中讨论的一样，网络设备驱动程序把数据包推送给内核，有两种方式通知内核数据到达。

轮询，即内核周期性地查询网络设备。这种方式的问题是内核需要多长时间询问网络设备一次。如果间隔太短，会无谓地浪费 CPU 时间；如果间隔太长，数据发送的延迟会增加，可能会丢失数据。

中断，即内核执行中断处理程序接收数据包，将其存放在 CPU 的接收队列中，其后的处理可以等 CPU 空闲时再完成。net_rx 是在接收中断中调用的处理接收数据的函数。

net_rx 的任务是申请一个 Socket Buffer，将硬件数据复制到 Socket Buffer 中，再将接收设备注册到 Socket Buffer 的 dev 数据域，识别数据包的协议类型。接着由 netif_rx()函数将接收到的数据包放入 CPU 的接收队列，更新接收统计信息，这时接收处理程序继续接收下一个数据包，或者结束返回。

5. 中断处理程序

网络设备驱动程序中，硬件产生中断的原因有很多，比如接收到了一个网络数据，一次发送结束，网络适配器产生某种错误，网络适配器状态变化等。在网络数据的收发过程中我们关心的中断主要有两个：

- 接收到一个新的网络数据包。
- 上次数据包发送结束。

网络设备硬件中断的处理程序要区分不同的中断，然后调用相应的函数来完成中断处理过程。通常当 CPU 接到网络适配器的硬件中断请求，响应设备中断请求转去执行硬件中断服务程序后，中断处理程序首先应从网络设备硬件的控制/状态寄存器中读出状态值来判断中断产生的原因。在这里我们假设中断产生原因存放在 dev->priv 数据域中。

（1）网络设备驱动程序中的中断处理程序模板

```
static irqreturn_t  xxx_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
   int statusword;
   //做必须的正确性检查，
   //读取中断状态字寄存器，获取中断原因
         statusword = 中断原因;

    if (statusword & 接收中断原因的编码 ) {    // 如果是接收中断
…
          xxx_rx(dev, pkt);                  //调用接收数据处理函数
```

```
        }

    //如果是上次发送数据包完成中断
    if (statusword & 发送完成中断原因的编码 ) {
       /* 释放已发送数据包的 Socket Buffer ，修改发送统计信息*/
       priv->stats.tx_packets++;
       priv->stats.tx_bytes += priv->tx_packetlen;
       dev_kfree_skb(priv->skb);
    }
    if (pkt) release_buffer(pkt);
return;
}
```

最后需要说明的是，如果网络设备驱动程序在发送开始时停止了发送队列，在发送中断处理完时，通常要重启发送队列（netif_wake_queue）。

（2）在 isa_skeletion.c 中的中断处理的实现

```
static irqreturn_t net_interrupt(int irq, void *dev_id)
{
  struct net_device *dev = dev_id;
  struct net_local *np;
  int ioaddr, status;
  int handled = 0;
  ioaddr = dev->base_addr;
  np = netdev_priv(dev);
//读网络设备状态寄存器，如果当前无中断产生，则中止中断
  status = inw(ioaddr + 0);
  if (status == 0)
        goto out;
  handled = 1;
//如果接收到新数据包产生的中断，则调用接收数据包处理函数
  if (status & RX_INTR) {
        net_rx(dev);
  }
//如果是发送结束中断，且网络设备硬件有发送队列，调发送结束处理函数 net_tx，重启发送队列
#if TX_RING
  if (status & TX_INTR) {
        net_tx(dev);
        np->stats.tx_packets++;
        netif_wake_queue(dev);
  }
#endif
  …
out:
  return IRQ_RETVAL(handled);
}
```


（3）在 isa_skeletion.c 中的发送结束中断处理函数

net_tx 函数的主要任务是更新统计信息，重新启动网络设备的发送队列。net_tx 在硬件产生发送结束中断时执行，它实际是中断处理程序的一部分。在执行 net_tx 时首先应锁定队列，以防止并发访问；更新发送字节数的统计信息，释放已发送的 Socket Buffer。最后查看队列在上次发送数据包前是否被暂停过（因队列满）。现在至少有一个 Socket Buffer 被释放，因此可以重新启动发送队列。

```
#if TX_RING
//该例程处理由设备通过中断提交的数据包发送完成事件
```

```
void net_tx(struct net_device *dev)
{
   struct net_local *np = netdev_priv(dev);
   int entry;
//获取防止并发执行 dev->hard_start_xmit 函数的锁
   spin_lock(&np->lock);
   entry = np->tx_old;
   while (tx_entry_is_sent(np, entry)) {
         struct sk_buff *skb = np->skbs[entry];
//更新统计信息，释放 Socket Buffer
         np->stats.tx_bytes += skb->len;
         dev_kfree_skb_irq (skb);
         entry = next_tx_entry(np, entry);
   }
   np->tx_old = entry;
//如果曾经因为发送队列满而停止过队列，而现在发送队列有新的空间接纳数据包，则唤醒队列
if (netif_queue_stopped(dev) && ! tx_full(dev))
         netif_wake_queue(dev);
   spin_unlock(&np->lock);
}
#endif
```

（4）缓解接收中断

当以上述的方式编写网络设备驱动程序时，每接收一个网络数据包就会产生一次中断，在大多数情况下这种工作方式可以很正常地处理网络数据的接收。但对于高带宽的网络接口，每秒要接收上千个数据包，中断的次数太多，反而会降低系统的性能。

为了解决这个问题，Linux 内核为网络设备建立了一种新的驱动程序模式（NAPI），它是基于轮询，同时结合了中断的工作模式。以前轮询模式被认为是一种效率不高也不智能化的工作模式，主要因为在这种模式下即使没有任何数据需要交换，CPU 也会不停地查询外部设备的状态。当系统配备了高速网络接口来处理高流量的网络数据交换时，网络设备中总有更多的数据包需要处理。这时如果采用轮询的工作模式就不需每次都中断 CPU 的正常工作流程，也能从网络接口读入数据包。

减少接收中断可以大大减轻 CPU 的负载。支持 NAPI 的驱动程序也可以利用拥塞管理来决定是否停止向内核发送数据包，但目前不是所有的设备都支持 NAPI，可以按 NAPI 方式工作的网络设备必须能存放多个数据包（在硬件上或接收队列中），在连续读接口的数据包时，接口应能关闭自己设备的接收中断，而打开发送或别的事件中断。

目前只有少数网络接口实现了 NAPI，如果你需要为高速网络设备编写驱动程序（如每秒钟会产生大量的接收中断），可以考虑用 NAPI 来实现。为了实现 NAPI 的驱动程序，首先需初始化代表网络设备的 net_device 数据结构实例的 poll 数据域。

```
dev->poll = xxx_poll;
dev->weight = 2;
```

poll 数据域是指向驱动程序实现的轮询设备数据的函数指针。 weight 描述的是在资源有限的情况下，CPU 一次可从网络设备中读入多少个数据包；这并没有严格的限制，通常 10Mbps 的以太网卡设置为 16 数据包，快速网络接口设置为 64 数据包，你不能把 weight 的值设置为大于接口能缓存的最大数据包数。

创建 NAPI 兼容的网络驱动程序的第二个步骤是修改中断处理程序。当网络接口产生接收中断信号时（这时接收中断是打开的），接收中断处理程序处理该数据包，这时驱动程

序需要关闭接收中断，并让内核开始轮询网络接口。基于前面的描述，接收中断应做如下修改：

```
    if ( statuword & xxx_RX_INTR ) {
          //关接收中断
          netif_rx_schedule(dev);
    }
```

当网络接口指明网络数据已在接口缓冲区时，中断处理程序让网络数据包留在接口中，中断处理程序需要做的就是调用 netif_rx_schedule。在 netif_rx_schedule 函数中，它会调用驱动程序的 poll 函数来处理数据包，poll 函数的实现框架为：

```
static int xxx_poll(struct net_dev *dev, int *budget)
//比较网络设备能传给内核的数据包数最大值与内核一次可读入数据包数的预算，取其最小值

   quota = min(dev->quota, *budget);
   循环（已读入的数据包数小于预算且接收队列不为空）{
          从网络接口的接收队列读入一个数据包
          分配 sk_buff，复制数据包
          填写 skb 的相关数据域：skb->dev
                                 skb->protocol
                                 skb->ipsumed
          将 skb 传给上层协议：调用 netif_receive_skb(skb)
                修改统计信息
          否   硬件接收队列空？
                是
          将设备放入数据接收完成设备队列
          开接收中断，返回
}
```


代码的中心部分是创建 skb 来接收数据包，这部分代码与前面的 net_rx 中的是一样的，其中有部分流程不一样：

- budget 是内核一次最多能从接口读入的数据包数（即每个接口占用 CPU 的预算），在代表设备的数据结构的变量 der 中，其数据域 quota 给出另一个值，是设备根据 weight 设定的，驱动程序应取 budget 与 quota 的最小值，而且这两个都应减去实际已接收到的数据包数（已用去多少预算）。
- 复制到 skb 中的数据包用 netif_receive_skb 上传给协议栈，而不是用 netif_rx。
- 如果 poll 函数在预算内已读入接口中的所有数据包，应重新打开接收中断，调

用 netif_rx_complete 关掉 polling，返回 0。poll 函数返回 1 说明接口中还有数据（在以上的过程中虽然关了中断，但数据包还可以从网络上进入接口的数据缓冲区），所以在一次中断产生后，只要在设备占用 CPU 的预算内，就可以一直读入数据包。

Linux 内核的网络子系统会保证任何网络设备驱动程序的 poll 函数不会在多个 CPU 上同时被调用，这样避免了对并发访问控制的复杂性。

5.3.3　网络设备管理函数

在网络设备驱动程序中除了实现常规的数据收/发功能外，还要管理网络设备与网络的连接状态、对各种信息进行统计；在某些情况下，还需要在运行时修改网络设备的资源配置，比如中断不能被自动识别，或中断与别的设备有冲突等。网络设备驱动程序需要为用户应用程序在系统运行状态下提供对网络设备配置修改的途径。这一节我们将看到这些功能如何在网络设备驱动程序中实现。

1. 定制 ioctl 命令

ioctl 命令是 Linux 内核与用户空间通信的有效工具。应用程序在用户空间任意创建一个 socket，使用 ioctl 命令可以设置或获取 TCP/IP 协议栈、网络设备的选项。在一个 socket 上执行一个 ioctl 命令时，该命令首先会传给网络协议层的 ioctl 命令，如果 ioctl 命令不是协议层的，内核将其传给网络设备。网络设备可以在驱动程序的 do_ioctl 函数中实现自己的 ioctl 命令。

```
ioctl( sd, SIOCSIFADDR, &ifreq );
```

使用 ioctl 命令需要传递 3 个参数，分别是：

- 创建的 socket 的描述符 sd。
- 调用哪个 ioctl 命令。比如 SIOCSIFADDR 表示 Socket I/O 控制命令设置网络接口的地址命令（Socket I/O Control Set Interface Address），Linux 网络子系统支持的所有 ioctl 命令的符号定义在 include/linux/sockios.h 文件中。
- 传给 ioctl 命令的设置参数。它存放在 struct ifreq 数据结构中。

下面主要来看看，网络子系统如何使用 ioctl 命令第三个参数。当一个 socket 发出 ioctl 命令时，ioctl 命令会转而调用内核的 sock_ioctl 函数，sock_ioctl 函数会直接调用 TCP/IP 协议栈协议层的相关函数执行 ioctl 命令。

任何 ioctl 命令，如果在协议层没有被识别就传给网络设备层的 dev_ioctl，这些与设备相关的 ioctl 命令，从用户空间接收第三个参数，它是 struct ifreq 数据结构类型的指针，该数据结构定义在 include/linux/if.h 文件中。如果用户程序要重新配置网络设备，就会将要设置的网络设备选项值通过 ifreq 数据结构传过来，由网络设备驱动程序的 ioctl 命令执行。如果用户程序要读取网络设备的配置，网络设备驱动程序执行 ioctl 命令后将查询到的设备配置值，通过 ifreq 传回给应用程序。

内核实现了一系列标准的 ioctl 命令，除了使用这些标准的调用外，每个网络接口可以定义自己的 ioctl 命令。在网络设备层 socket 还可以识别另外 16 个私有 ioctl 命令，命令索引号在 SIOCDEVPRIVATE 到 SIOCDEVPRIVATE+15 之间。一旦这些私有的 ioctl 命令被系

统识别，就执行网络设备驱动程序提供的 ioctl 命令：dev->do_ioctl， 它也接收 struct ifreq 类型的指针作为参数。

例如，在 isa_skeletion.c 中实现的 ioctl 命令，我们假设 SIOCDEVPRIVATE 命令为可以向用户程序返回网络设备的物理地址，SIOCDEVPRIVATE+1 与 SIOCDEVPRIVATE+2 为网络设备驱动程序实现的某些特定的 ioctl 命令。实现 ioctl 命令的流程及格式如下：

```
static int net_ioctl(struct net_device *dev, struct ifreq *ifr, int cmd)
{
   struct net_local *lp=(struct net_local *)dev ->priv;
   long ioaddr=dev->base_addr;
   u16 *data =(u16 *)&ifr ->ifr_data;
   int phy =1p->phy[0] & 0x1f;

   switch(cmd) {
      case SIOCDEVPRIVATE:   /*获取当前网络设备使用的物理地址 */
         data[0]=phy;
      case SIOCDECPRIVATE+1: /* isa_skeletion 设备驱动程序特有的 ioctl 命令 1 */
         special_ioctl_1();
      case SIOCDEVPRIVATE+2: /* isa_skeletion 设备驱动程序特有的 ioctl 命令 2 */
         special_ioctl_2();
     …
        /*其他 ioctl 命令 */
        default:
           return -EOPNOTSUPP;
   }
}
```

ioctl 命令的参数 ifr 指向内核地址空间，其中有从用户地址空间发送过来的数据结构的融本，在 ioctl 命令返回后，ifr 的融本将复制回用户地址空间，这样网络设备驱动程序可以使用私有 ioctl 命令与用户空间通信。

网络设备驱动程序特有的 ioctl 命令可以选择使用 struct ifreq 数据结构中的数据域，但它们已经成为传递命令的标准结构了，所以驱动程序不可能按自己的需要使用这个数据结构。其中 ifr_data 数据域是一个 caddr_t 的数据条目，它的意思是驱动程序可以按设备特定的需要使用。

2．统计信息

大多数时候我们都希望获取网络设备操作和设备本身的统计信息，详细的记录事件日志可以帮助我们发现和解决错误。要获取当前设备工作情况的统计信息，需要实现驱动程序的 get_stats 方法，该方法返回一个指向统计数据的指针。get_stats 方法是驱动程序中最后一个必须要完成的方法。这个方法的实现非常简单，即使这个网络设备驱动程序管理多个网络设备,该方法也一样工作正常，因为统计信息存放在网络设备私有数据结构中。

（1）get_stats 方法的实现模板

```
struct net_device_stats *xxx._get_stats(struct net_device *dev)
{
  struct xxx_priv *priv = netxev_prive(dev) ;
  return *priv->stats ;
}
```

（2）在 isa_skeletion.c 中获取网络统计信息方法的实现

```
static struct net_device_stats *net_get_stats(struct net_device *dev)
{
  struct net_local *lp = netdev_priv(dev);
  short ioaddr = dev->base_addr;

  /* 从网络设备寄存器中更新统计信息 */
  lp->stats.rx_missed_errors = inw(ioaddr+1);
  return &lp->stats;
}
```

从上面的 net_get_stats 方法的实现中我们可以看到，网络设备工作状态的统计信息都是存放在设备的私有数据结构中，这样即使同一个驱动程序驱动多个设备，仍可以获取正确的统计信息。

（3）统计信息的更新

对统计信息的更新操作分布在网络设备驱动程序的各函数中，比如接收数据包的函数累计该网络设备接收的数据包总数和接收的总字节数；发送数据包函数累计本网络设备发送数据包总数和发送的总字节数。这些统计信息定义在 struct net_device_stats 数据结构中，struct net_device_stats 数据结构中最重要的数据域如下。

- unsigned long rx_packets/tx_packeds：由接口成功接收/发送的数据包数。
- unsigned long rx_bytes/tx_bytes：由接口成功接收/发送的字节数。
- unsigned long rx_errors/tx_errors：接收/发送过程中产生的错误总数。

在网络数据传输过程中仅仅统计产生了多少次接收错误，多少次发送错误还不够，在 include/linux/netdevice.h 文件中定义了 6 种接收错误，5 种类型的发送错误。在编写网络设备驱动程序时应尽量详细地对各种类型的错误分别统计，这样才能最大限度地帮助系统管理员跟踪和分析网络错误。

- unsigned long rx_dropped/tx_dropped：在接收/发送过程中扔掉的数据包数。当系统中没有内存空间存放接收数据包时，就会丢包，而 tx_dropped 很少使用。
- unsigned long collisions：传输介质拥塞时发生冲突的次数。
- unsigned long multicast：接收到的组发送的数据包数。

需要说明的是 get_stats 可以在任何时候调用，即使网络接口驱动程序被卸载了以后仍可以用 get_stats 获取网络设备的统计信息，所以驱动程序在运行 stop 方法时不应释放统计信息。

5.3.4　在适配器中支持组发送

网络设备通过分析数据包中目的 MAC 地址来判断是否应该接收数据包，这个过程发生在网络适配器硬件上，不需要驱动程序及 CPU 的干预。单地址数据包的接收很简单，网络适配器只需检查目的地址是否与自己的 MAC 地址相符即可，广播数据包也简单，网络适配器接收所有的数据包。情况比较复杂的是组发送，网络设备要判断主机是否应接收某个组发送地址的数据包。有时 CPU 需要花大量的时间查看每个组发送数据包（网卡硬件不支持保存组发送地址）。现在的网络适配器都提供了很好的在 MAC 层对组发送的支持，这类网络适配器管理了一个组发送地址列表，该列表由主机发送给网络设备，网络设备就根

据列表中的地址接收主机所需的组发送数据包。

网络设备驱动程序支持组发送功能，需要完成的任务就是接收主机发送过来的组发送地址，并将主机设置的组发送地址送往设备硬件。

组发送数据包是指可由多个主机接收的数据包，但并不是所有的主机都可以接收。组发送功能通过为一组主机分配特殊的硬件地址来实现。送往某个特殊硬件地址的数据包可以被这个组中的所有主机接收到。在以太网中，组发送地址设在数据包目的地址第一个字节的最低位。

组发送中最复杂的部分已由应用程序和内核实现，网络接口驱动程序不需要关心这些问题。组发送的发送数据包非常简单，它们与普通的发送数据包没什么不同。接口在发送这类数据包时，把它们放到网络通信介质上，并不需要查看数据包的目的地址，由内核为这类数据包分配正确的目的地址。

内核需要完成的工作是跟踪有效的组发送地址列表，因为这些地址列表可能随时都在发生变化。修改组发送地址列表的工作由应用程序完成，用户可以随时更改组发送地址。驱动程序的任务是接收组发送地址列表，任何一个数据包的目的地址如果包含在设置的组发送地址列表中，驱动程序就将该数据包发送给内核。驱动程序如何实现组发送地址列表的接收和管理，依赖于网络设备硬件功能。目前大多数硬件属于以下 3 种：

- 接口自己不能处理组发送功能。这类接口接收目的地址与设备硬件地址一致的数据包，或者接收所有的数据包，它们只有通过接收所有的数据包来实现组发送。这种接收方式会造成接收数据包过多而使内核负载过大，对这种设备不能使用组发送功能，dev->falgs 的 IFF_MULTICAST 位域不应设置。

但点对点接口是一个特殊的情况，因为点对点设备不做任何地址过滤，它们总是接收所有数据包。

- 接口可以区分组发送数据包与其他数据包（主机到主机，或广播），这些设备可以接收所有的组发送数据包，由软件来决定数据包是否为本机需要接收的组发送数据包。这种方式内核的工作负载在可接收范围内，通常在某段网络上组发送的数据包并不多。
- 接口本身可检测组发送的硬件地址，这些接口保存了一个组发送地址列表，说明需要接收哪些组发送数据包，它们忽略没有缓存在本机组发送地址列表中的组发送数据包。对内核而言这是最优化的工作方式，处理器无须耗时去扔掉不属于本机接收的组发送数据包。

组发送地址列表发生变化时，内核通知网络设备驱动程序，将新的地址列表传给驱动程序，这样硬件就可以按照新的地址列表过滤数据包。

1．内核对组发送的支持

组发送功能的实现在内核中分为以下几个部分：网络设备驱动程序的函数、数据结构和设备标志。

（1）数据结构

与组发送相关的数据结构有两个：其一，本网络设备应接收的所有组发送数据包的地址，存放在一个列表中（struct dev_addr_list *dev->mc_list），它嵌套在 struct net_device 数据结构实例中，指向本网络设备相关的组发送地址列表的指针；其二，组发送地址列表中

的成员个数 int dev->mc_count。当查看是否需做组发送时，查看 mc_count 的值是否为 0，比查看组发送地址列表更快。

（2）设备标志

设备标志标明了当前网络设备是否工作在组发送模式，是否应接收或发送组发送数据包。与组发送功能相关的设备标志有如下几个。

- IFF_MULTICAST：除非驱动程序设置了 dev->flags 的 IFF_MULTICAST 位标志，否则接口不会处理组发送数据包。无论什么时候当 dev->falgs 的设置发生变化时，内核都会调用驱动程序的 set_multicast_list 组发送方法。在接口未被激活时，组发送地址列表也可能发生变化。
- IFF_ALLMULTI：dev-flags 中的标志位，让驱动程序查看所有从网络上到达的组发送数据包。当打开了组发送路由时就需要这个功能。如果这个标志位已设置，就不应再用 dev->mc_list 来过滤组发送数据包。
- IFF_PROMISC：将网络接口设置为混杂模式，这时网络上的所有数据包都要接收。这种模式独立于组发送地址列表 dev->mc_list 的设置。

（3）驱动程序方法 dev->set_multicast_list

任何时候，当与设备相关的组发送地址列表改变时调用该方法。当 dev->flags 的值发生变化时也需要调用这个方法，因为某些位域（如 IFF_PROMISC）也需要重新编程网络设备硬件地址过滤器。

实现对组发送的支持，驱动程序开发人员最后要做的事就是定义一个 struct dev_addr_list 的实例，dev_addr_list 数据结构定义在 include/linux/netdevice.h 文件中。

```
struct dev_addr_list
{
  struct dev_addr_list   *next;                    //列表中的下一个地址
  u8              da_addr[MAX_ADDR_LEN];           //组发送地址字符串
  u8              da_addrlen;                      //地址长度
  u8              da_synced;
  int             da_users;                        //用户数
  int             da_gusers;                       //组数
};
```

数据包组发送的硬件地址独立于数据包，所以这个数据结构在不同的网络上都能用，每个地址由地址字符串和其长度标识。

2．组发送函数的模板

描述 set_multicast_list 实现的最好的方法是给出一些伪代码。下面的函数是一个支持组发送网络驱动程序的全功能实现，这种网络接口硬件具有地址过滤功能，硬件可以保存来自主机的组发送地址列表，表的最大长度为 FF_TABLE_SIZE。以下函数中所有带有 ff_前缀的是指硬件操作。

```
void ff_set_multicast_list(struct net_device *dev)
{
  struct dev_mc_list *mcptr;
//如果网络设备的接收工作模式设置为混杂模式，接收所有的数据包
  if ( dev->flags & IFF_PROMISC) {
    ff_get_all_packets( );
    return;
```

```
}
//工作模式为接收所有组发送数据包，或组发送地址列表中的地址数大于设备硬件能保存的最大地址数，则接收所有组发送数据
//包，由软件对数据包进行分类
if ( dev->flags & IFF_ALLMULTI ) || dev->mc_count >FF_TABLE_SIZE)
{
  ff_get_all_multicast_packets( );
  return;
}
//将所有组发送地址保存在网络适配器的硬件地址过滤器中
ff_clear_mc_list( );
for ( mcptr = dev->mc_list; mcptr; mcptr = mcptr->next)
       ff_store_mc_address(mcptr->da_addr);
ff_get_packets_in_multicast_list();
}
```

如果网络接口硬件不能存放组发送地址列表，这时 FF_TABLE_SIZE 就为 0，以上函数最后 4 行代码就不需要了。

即使网络接口硬件不能处理组发送包，也需要实现 set_multicast_list 方法，用以通知 dev->flags 的值发生了变化，这时它的实现就非常简单了。

```
void nf_set_multicast_list(struct net_device *dev)
{
  if ( dev->flags & IFF_PROMISC )         //如果网络设备工作在混杂模式
       nf_get_all_packets ( );            //接收所有网络数据包
  else
      nf_get_only_own_packets ( );        //否则只接收与本主机地址相符的数据包
}
```

在驱动程序中实现 IFF_PROMISC 是非常重要的，否则用户就不能运行 tcpdump 或别的网络分析工具。另外，如果接口是点对点的设备，就不需要实现 set_multicast_list 方法了，因为用户总是接收所有数据包。

3．在 isa_skeletionc 中组发送的实现

```
static void set_multicast_list(struct net_device *dev)
{
  short ioaddr = dev->base_addr;
  if (dev->flags&IFF_PROMISC)
  {
//设备工作于混杂模式，将混杂模式写入设备端口控制寄存器
       outw(MULTICAST | PROMISC, ioaddr);
//将组发送与混杂模式的编码做与运算，并写入设备端口
  }
//设备工作模式设置为接收所有组发送数据包，或组发送地址大于硬件能存放的最大组发送地址数，禁止混杂模式，接收所有组
//发送数据包
else if((dev->flags&IFF_ALLMULTI) || dev->mc_count > HW_MAX_ADDRS)
  {
       hardware_set_filter(NULL);
       outw (MULTICAST, ioaddr);
  }
//只接收组发送地址列表中指定的组发送数据包
  else if (dev->mc_count)
  {
       hardware_set_filter(dev->mc_list);
       outw ( MULTICAST, ioaddr);
  }
  else
```

```
            outw(0, ioaddr);
    }
```

当网络设备工作于混杂模式时，该模式在网卡上被激活，这时要接收所有的组发送数据包；组发送地址列表中的地址数大于适配器上地址过滤器能存放的有效地址数时，也接收所有的组发送包，否则只接收期望的组发送数据包，这里用 hardware_set_filter(dev->mc_list)表示。

5.4　CS8900A 网络适配器驱动程序的实现

前面的章节介绍了实现网络设备驱动程序的基本要领、网络设备驱动程序方法模板，以及在无实际硬件情况下设备驱动程序实现的实例，本节我们将以一个实际的网络适配器为例，分析网络设备驱动程序在内核中的设计与实现。

了解网络设备驱动程序实现的基本原理后，当我们需要为一个新的网络控制器开发驱动程序时，首先要理解网络控制器的硬件特点，比如 I/O 端口地址的设置、中断设置、是否支持 DMA 发送等基本组成特性和操作特点。通常做网络设备驱动程序开发的人员需要阅读网络控制芯片手册（Data sheet）来获取这些信息。

5.4.1　CS8900A 网络控制芯片的功能概述

CS8900A 是 Cirus Logic（凌云逻辑公司）的以太网网络控制器，其主要构成如下。

- ISA 总线接口：与主机通信，可选择配置 4 个中断信号，3 个 DMA 通道。
- 802.3 MAC 引擎：完成以太网数据包的创建，处理所有以太网数据包的接收和发送，还可完成冲突检测，数据包协议头的创建和检测，校验和 CRC 的创建与检测，冲突自动重传等功能。
- 片上 4KB 的 RAM：称为 PacketPage Memory，简称为 PP_，可操作在内存映射模式、I/O 端口模式和扩展 DMA 模式，是 CS8900A 网络控制器的寄存器和数据收发缓冲器。
- 一个串行 EEPROM 接口：可外接 EEPROM，存放网络适配器配置信息和用户定制信息，启动时自动调入配置网络适配器。
- 10BASE-T/AUI 网络连接器。

CS8900A 构成框图如图 5-6 所示。

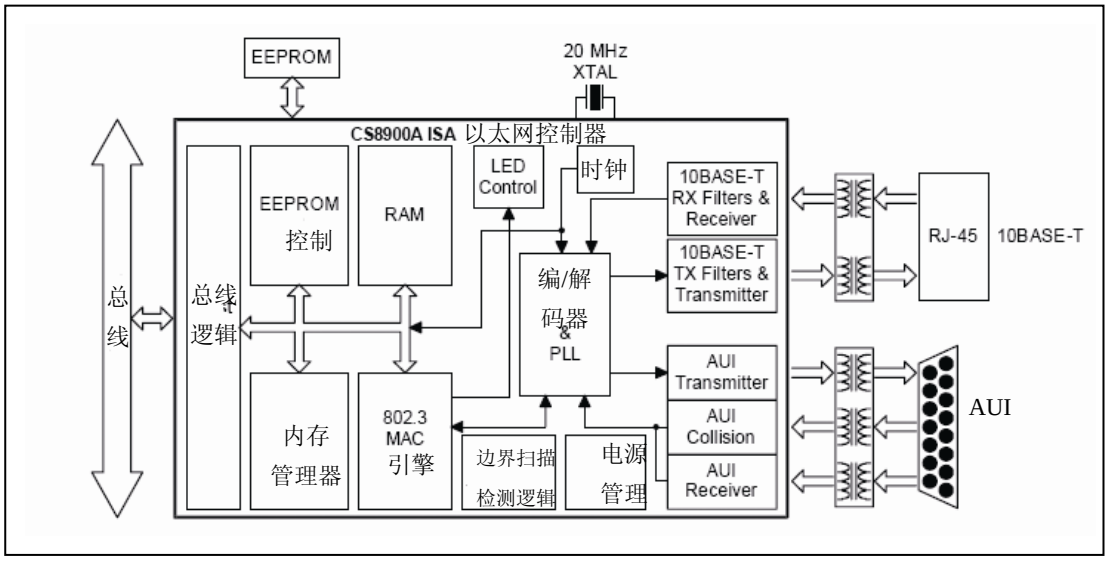


图 5-6　CS8900A 的构成框图

1．芯片配置

在常规操作期间，CS8900A 执行两个基本功能：以太网数据包的接收与发送。在网络适配器可以接收和发送数据之前，需要对 CS8900A 进行配置，各种参数必须写入芯片的配置和控制寄存器，例如接收/发送数据包缓冲区的基地址、以太网的物理地址、接收包的类型、使用的介质接口。对适配器的配置数据可以由主机写入 CS8900A 芯片的控制寄存器，也可以从外接在 CS8900A 上的 EEPROM 中自动装入。配置完成后，数据收/发操作就可以开始了。

2．发送数据包

数据包的发送分为两个阶段进行。第一阶段主机将以太网数据包放入 CS8900A 的数据缓冲区。主机发出一条发送命令时，第一阶段就开始了这条命令通知 CS8900A 有一个数据包需要发送，告诉芯片什么时候开始发送（比如在 5，380，1021 或所有字节被缓冲后），如何发送数据包（比如是否要做 CRC 校验，在数据包后是否有补零的填充位等）。主机给出发送命令后，紧接着给出数据包的长度，指出需要多少数据缓冲区的空间。当 CS8900A 上的发送数据缓冲区空间有效时，主机就将数据包写入 CS8900A 的内部数据缓冲区中。

发送的第二阶段，CS8900A 将数据包转换成以太网的数据包格式，发送到网络介质上。一旦相应的字节数写入 CS8900A 的发送缓冲区中（5，381，1021 或全部数据），第二阶段就起始于 CS8900A 发送 preamble 和数据包起始标志（Start-of-Frame delimiter）。preamble 和数据包起始标志后跟的是数据包的目标地址、源地址、长度域和 LLC 数据（这些都由主机提供。如果数据包的长度小于 64 个字节，包括 CRC 在内，CS8900A 在数据包末尾补零凑足 64 个字节，如果芯片配置了此功能），最后 CS8900A 加入 32 位的 CRC 校验值。第二阶段的发送过程完全由适配器硬件完成。

3．接收数据包

与数据包的发送一样，数据包的接收也分为两个阶段。第一阶段 CS8900A 接收到一个

以太网数据包，将数据包存于芯片上的缓冲区。第一阶段起始于接收帧经过芯片的模拟数据分析前端。接着 CS8900A 将 preamble 和数据帧起始标志从接收到的数据包中去掉，然后将数据包传给地址过滤器。如果数据包的目标 MAC 地址与地址过滤器中配置的 MAC 地址一致，数据包就存放在 CS8900A 的数据缓冲区中。CS8900A 查看 CRC，根据配置信息通知 CPU 收到了数据包。这一阶段也由 CS8900A 硬件完成，无须 CPU 干预。

第二阶段，主机将收到的数据包复制到主机内存。接收数据包的发送可能按内存映射方式、I/O 端口操作或 DMA 操作发送。CS8900A 也提供在内存映射方式与 I/O 端口方式之间的切换。DMA 操作使用自动切换到 DMA 的方式，按数据流发送。

4．ISA 总线接口

CS8900A 芯片有一个直接连到 ISA 总线上的接口，将 CS8900A 优化为 16 位的数据发送器。通过 ISA 总线 CS8900A 与主机的通信可以工作在两种模式下。

（1）内存映射模式

CS8900A 配置为内存映射模式时，CS8900A 的内部寄存器和数据包缓冲区映射到内存中一个连续的 4KB 区域，使主机可以直接访问 CS8900A 的内部寄存器和数据缓冲存储器。

（2）I/O 模式

CS8900A 配置为 I/O 模式时，主机通过 CS8900A 的 8 个 16 位的 I/O 端口访问芯片的内部寄存器与数据缓冲区。这 8 个端口映射到主机系统的 I/O 地址空间 16 个连续单元上。I/O 模式是 CS8900A 的默认工作模式。

CS8900A 有 4 个中断请求输出引脚，可以直接连接到 ISA 总线的 4 个中断请求信号线上（见表 5-2）。一次只能使用一个中断输出。中断信号线是在初始化时选择的，将中断号写入数据页面存储器基地址 （PacketPage Memory base） +0022H 单元。不使用的中断引脚处于高阻状态。中断产生时，初始化了的中断信号线为高电平，触发中断。当中断状态队列（Interrupt Status Queue，ISQ）标识的中断都读出后，信号线为低电平。表 5-2 给出了 CS8900A 的中断信息与 ISA 总线的连接方式。

表 5-2　CS8900A 的中断信号线与 ISA 总线的连接方式

CS8900A 的中断信号线	ISA 总线的中断线	0022h 寄存器中的值
INTRQ3	IRQ5	0003h
INTRQ0	IRQ10	0000h
INTRQ1	IRQ11	0001h
INTRQ2	IRQ12	0002h

CS8900A 接口可以直接连到主机的 DMA 控制器上，实现数据接收帧的 DMA 发送。CS8900A 有三对引脚，可以连到主机的三个 DMA 通道上，一次只能使用一个 DMA 通道，在初始化期间将选择的通道号写入 0024h 单元。

5.4.2　CS8900A 的 PacketPage 结构

对 CS8900A 的控制和访问，是基于芯片内部的高性能页面存储器（PacketPage Memory），页面存储器中包含了 CS8900A 芯片的内部寄存器和数据缓冲器。根据 CS8900A 工作模式的配置，主机可以在内存映射模式或 I/O 端口模式下访问 CS8900A 的页面存储器。

要理解 CS8900A 驱动程序的实现，最重要的一点就是了解 CS8900A 内部页面存储器的组织，这样才知道在数据包的发送/接收过程中应从何处获取网络设备状态，向何处写发送命令，接收/发送的数据存放在芯片的什么位置。

CS8900A 体系结构的中心是内部集成的一个 4KB 页面的 RAM，称为页面存储器。页面存储器用于临时存储发送和接收的数据包、芯片内部的控制/状态寄存器的值。主机对页面存储器的操作可以如访问主机内存空间一样，直接读写（CS8900A 配置为内存映射模式），或通过 I/O 指令访问（CS8900A 配置为 I/O 端口模式）。内存映射模式可以提供更优的访问性能，I/O 端口模式是 CS8900A 的默认工作模式。

在 CS8900A 的页面存储器中，用户可访问的部分空间组成 6 个部分，如表 5-3 所示。

表 5-3　CS8900A 的页面存储器中，用户可访问的空间

基于 PacketPage 基地址的偏移量地址范围	内　　容
0000h~0045h	总线接口寄存器
0100h~013Fh	状态/控制寄存器
0140h~014Fh	初始发送寄存器
0150h~015Dh	地址过滤寄存器
0400h	接收数据包相关信息的起始区域
0A00h	发送数据包存储区域

（1）总线接口寄存器

总线接口寄存器中保存的是对 CS8900A 的 ISA 总线接口工作状态的配置，比如当前 CS8900A 使用的是哪一个中断信号线、DMA 通道；是工作在内存映射模式还是 I/O 端口模式；工作在内存映射模式下时，页面存储器映射到主存哪一块区域；工作于 I/O 端口模式时，I/O 端口的起始地址是什么等。

这一部分寄存器大多数在芯片初始化期间为只写操作，在 CS8900A 正常工作期间寄存器的值通常不可修改。除了管理 DMA 发送的寄存器外，在 DMA 方式发送期间，寄存器的值在不断变化。

（2）状态/控制寄存器

从 CS8900A 的状态寄存器我们可以获取网络适配器当前的工作状态。主机发往 CS8900A 的命令将写入控制寄存器。

（3）初始发送寄存器

这部分寄存器包含了两个基本寄存器，发送控制命令寄存器 TxCMD 和发送数据包长度寄存器 TxLength。用于主机初始化一个以太网数据包的发送。

（4）地址过滤寄存器

地址过滤寄存器用于过滤数据包的目标地址，以确定从网络上接收的数据包是否为传给本主机的数据包。

（5）接收与发送数据包存储区域

这部分存储区域用于存放 CS8900A 从网络上接收到的数据包与主机写入 CS8900A 向外发送的数据包。主机只需直接读/写这部分区域，芯片内部的缓冲区在发送和接收期间按需要动态分配，这样可以极大地利用页面存储器的空间。这样在接收和发送期间，只有一个接收数据包（位于 PacketPage 基地址+0400h 处）和一个发送数据包（位于 PacketPage 基地址+0A00h 处）可直接访问。

基于以上描述，CS8900A 的页面存储的地址映射如表 5-4 所示。

表 5-4　CS8900A 页面存储器的地址映射表

	PacketPage 地址	占用字节数	操　　作	寄存器描述
总线接口寄存器（Bus Interface Regisgers）	0000h	4	只读	存放产品标识号
	0004h	28	–	保留
	0020h	2	读/写	I/O 端口基地址（0、1、2 或 3）
	0022h	2	读/写	中断号
	0024h	2	读/写	DMA 通道号（0、1 或 2）
	0026h	2	只读	DMA 数据包起始地址
	0028h	2	只读	DMA 数据包数
	002Ah	2	只读	DMA 接收字节数
	002Ch	4	读/写	内存映射基地址寄存器（20 位）
	0030h	4	读/写	Boot EPROM 基地址
	0034h	4	读/写	Boot EPROM 地址屏蔽码
	0038h	8	–	保留
	0040h	2	读/写	EEPROM 命令寄存器
	0042h	2	读/写	EEPROM 数据
	0044h	12	–	保留
	0050h	2	只读	接收数据包字节计数器
	0052h	174	–	保留
状态和控制寄存（Status & Control Register）	0100h	32	读/写	配置与控制寄存器（Configuration & Control Registers，每个寄存器占 2 个字节）
	0120h	32	只读	状态与事件寄存器（Status & Event Registers，每个寄存器占 2 个字节）
	0140h	4	–	保留
初始发送寄存器（Initiate Transmit Registers）	0144h	2	只写	发送命令寄存器 TxCMD
	0146h	2	只写	发送数据包长度 TxLength
	0148h	8	–	保留
地址过滤寄存器（Address Filter Registers）	0150h	8	读/写	逻辑地址过滤器
	0158h	6	读/写	硬件地址
	015Eh	674	–	保留
数据包驻留区（Frame Location）	0400h	2	只读	接收数据包状态 RXStatus
	0402h	2	只读	接收数据包长度 RxLength
	0404h		只读	接收数据包缓冲区起始
	0A00h		只写	发送数据包缓冲区起始

要真正理解 CS8900A 网络适配器驱动程序的实现，开发人员还应阅读 CS8900A 芯片手册中对以上寄存器各位含义的描述，这样才能理解芯片配置在什么工作模式，当前数据包接收/发送配置，以及是什么事件引起中断，驱动程序如何处理中断事件等。这里限于篇幅我们就不对寄存器各位的含义进行描述了，在讲解到驱动程序实现的相应部分时再对寄存器的各位进行解释，读者可以下载 CS8900A 的芯片手册对照理解驱动程序的实现。

5.4.3　CS8900A 的操作

了解了 CS8900A 芯片的体系结构，我们还需要了解 CS8900A 网络适配器各种操作的

流程，然后才可以具体分析 CS8900A 网络适配器驱动程序的实现。

1．CS8900A 芯片的配置

每次芯片复位后，CS8900A 都会查看是否有外接的 EEPROM，如果有外接的 EEPROM，CS8900A 自动调用存放在 EEPROM 中的配置数据到芯片内部寄存器；如果没有发现外接的 EEPROM，CS8900A 就使用默认的配置。这些配置数据主要写入 CS8900A 内部页面存储器的总线接口寄存器与状态/控制寄存器的配置控制寄存器部分。它们描述了 CS8900A 的工作方式配置。在没有外接 EE-PROM 时，CS8900A 的部分默认配置如表 5-5 所示。

表 5-5　CS8900A 的默认配置

	PacketPage 地址	寄存器描述	寄存器默认值
总线接口寄存器（Bus Interface Register）	0020h	存放 I/O 端口基地址	0300h
	0022h	中断号	XXXX XXXX XXXX XX10
	0024h	存放 DMA 通道号	XXXX XXXX XXXX XX11
	0026h	存放 DMA 数据包起始偏移量	0000h
	0028h	存放 DMA 数据包数据	X000h
	002Ah	接 DMA 字节数	0000h
	002Ch	存放内存映射基地址	XXX0 0000h
	0030h	Boot PROM 基地址寄存器	XXX0 0000h
	0034h	Boot PROM 地址屏蔽寄存器	XXX0 0000hh
状态和控制寄存（Status & Control Registers ）	0102h	寄存器 3：接收配置寄存器 RxCFG	0003h
	0104h	寄存器 5：接收控制寄存器 RxCTL	0005h
	0106h	寄存器 7：发送配置寄存器 TxCFG	0007h
	0108h	寄存器 9：发送命令寄存器 TxCMD	0009h
	010Ah	寄存器 B：缓冲区配置寄存器 BufCFG	000Bh
	0112h	寄存器 13：线路控制寄存器 LineCTL	00013h
	0114h	寄存器 15：自控制寄存器 SelfCTL	0015h
	0116h	寄存器 17：总线控制寄存器 BusCTL	0017h
	0118h	寄存器 19：测试控制寄存器 TestCTL	0019h

2．CS8900A 的工作模式

CS8900A 的工作模式可以配置为内存映射模式（MemoryMap，这时网络适配器上的寄存器和数据缓冲器与内存统一编址）与 I/O 模式（网络适配器上的寄存器与数据缓冲器单独编址）。

（1）内存映射模式

在内存映射模式下，PacketPage 的基地址必须映射到以 X000h 边界为起始地址的主机内存区域。CS8900A 芯片复位后，默认配置是工作在 I/O 端口模式。一旦选择了内存映射模式（通过设置总线控制寄存器 BusCTL 的 E 位），CS8900A 的寄存器都可以由主机直接访问。

在内存映射模式下，对 CS8900A 访问支持标准的或总线准备好时钟周期，不需插入其他的等待状态，这样使主机对芯片的访问速度更快。

CS8900A 工作在内存映射模式时，CS8900A 适配器的接收/发送数据缓冲器（即数据包驻留区）映射的内存区域如表 5-6 所示。

表 5-6　CS8900A 内存映射模式工作中数据收发缓冲区在内存中的地址

描　　述	寄 存 器 名	操　　作	基于内存映射基地址的偏移量
接收状态	RxStatus	只读	0400h~0401h
接收长度	RxLength	只读	0402h~0403h
接收数据包存放位置	RxFrame	只读	起始于 0404h
发送数据包存放位置	TxFrame	只写	起始于 0A00h

如果将 CS8900A 配置为工作在内存映射模式下，内存映射基地址须写入 PacketPage 基地址+ 002Ch 单元中（见表 5-3）。

当 CS8900A 工作于内存映射模式时，基本的发送过程为：

- 主机将发送命令和发送数据包长度分别写入 CS8900A 的初始发送寄存器（Initiate Transmit Registers）的发送命令寄存器 TxCMD（地址为内存映射基地址+0144h）和接收数据包长度寄存器 RxLength（地址为内存映射基地址+ 0146h）中。如果发送长度值有错，本次发送命令丢弃，CS8900A 通过 ISA 总线向主机返回错误信息。
- 主机读入总线状态寄存器 BusST（寄存器 18，内存映射基地址+0138h）的值。如果该寄存器的发送准备好位 RdyTxNOW（第 8 位）置 1，主机就可以向 CS8900A 芯片写发送数据包。如果 RdyTxNOW 位被清除，主机必须等到 CS8900A 上的缓冲区有效后才能写入发送数据帧。
- 一旦 CS8900A 准备好接收主机传来的发送数据包，主机循环执行存储区到存储区（memory-to-memory）的数据复制到内存映射基地址+0A00h 区域。发送数据包就从主机内存复制到 CS8900A 芯片上的发送数据包缓冲区中。

CS8900A 网络适配器在内存映射模式下的接收操作按以下流程进行：

- CS8900A 接收到一个有效数据包，触发一个中断。
- 主机读入 CS8900A 的中断状态队列（内存映射基地址+0120h），获知网络适配器收到一个网络数据包。
- 主机读入接收状态寄存器 RxStatus（内存映射基地址+0400h），获取接收到的数据包状态。
- 主机读入接收数据包长度寄存器 RxLength（内存映射基地址+0402h），获取接收到的数据包的长度。
- 主机执行循环内存复制，从 CS8900A 芯片接收数据缓冲区（内存映射基地址+0404h 起始处）将整个数据包从芯片缓冲区复制到主机内存。

（2）I/O 模式

当配置为 I/O 模式时，CS8900A 通过 8 个 16 位的 I/O 端口访问，这 8 个端口映射到主机系统 I/O 地址空间中的 16 个连续单元。I/O 模式是 CS8900A 的默认工作模式。在 CS8900A 的默认配置中，芯片上电时，I/O 端口基地址存放在 0020h 寄存器中，I/O 端口基地址默认值为 300h。I/O 端口基地址可以配置为任意有效的 XXX0h 区域。表 5-7 为 CS8900A 工作在 I/O 模式下各端口的映射值。

表 5-7　I/O 模式下端口的映射值

相对 I/O 端口的偏移量	操 作 类 型	描　述
000h	读/写	接收/发送数据端口（端口 0）
0002h	读/写	接收/发送数据端口（端口 1）
0004h	只写	发送命令端口 TxCMD
0006h	只写	发送数据包长度端口 TxLength
0008h	只读	中断状态队列端口
000Ah	读/写	PacketPage 指针端口
000Ch	读/写	PacketPage 数据（端口 0）
000Eh	读/写	PacketPage 数据（端口 1）

- 接收/发送数据端口 0 和端口 1。这两个端口用于发送数据包到 CS8900A 和从 CS8900A 读入接收到的数据包。端口 0 用于 16 位数据操作，端口 0 和端口 1 一起用于 32 位数据操作。
- 发送命令端口。主机在发送操作开始时将发送数据包命令写入该端口。发送命令告诉 CS8900A 主机有数据包要发送，以及数据包应如何发送。
- 发送数据帧长度端口。在主机写入发送命令到端口后，将要发送数据包的长度写到该端口。
- 中断状态队列端口。该端口中包含了中断状态队列（ISQ）中的当前值。ISQ 驻留在 PacketPage 基地址+0120h 处。
- PacketPage 指针端口。主机要访问 CS8900A 的内部寄存器时，写 PacketPage 指针端口。该端口的前 12 位（第 0 位到第 B 位）提供当前访问操作的内部目标寄存器地址，另外 3 位为只读，其值总是设置为 011b。最高位指明 PacketPage 指针在操作结束后是否自动递增 1。
- PacketPage 数据端口 0 和端口 1。PacketPage 数据端口用于向 CS8900A 内部寄存器发送数据或从 CS8900A 内部寄存器读出数据。端口 0 用于 16 位操作，端口 0 与端口 1 一起用于 32 位操作（低位字节在端口 0 中）。

要进行 I/O 读或写操作时，在 ISA 地址总线上的 16 位 I/O 地址应与 CS8900A 所在的 I/O 端口地址相符合。

I/O 模式下的发送操作按以下步骤进行：

① 主机向发送命令端口 TxCMD （I/O 端口基地址+0004h）写入发送数据包命令，向发送长度端口 TxLength（I/O 端口基地址+0006h）写入要发送的数据包长度。

② 主机读入总线状态寄存器 BusST（寄存器 18），查看 Rdy4TxNOW 位是否设置，不过为了读 BusST 寄存器，主机首先必须设置 PacketPage 指针为要操作的寄存器地址，即将 0138h 写入 PacketPage 指针端口（I/O 端口基地址+000Ah），主机就可以从 PacketPage 数据端口（I/O 基地址+000Ch）读入总线状态寄存器的值了。如果 Rdy4TxNOW 位已设置，主机可向 CS8900A 写入要发送的数据包。如果 Rdy4TxNOW 被清除，则主机必须等到 CS8900A 的缓冲区有效后才能发送数据帧。如果 CS8900A 的缓冲区配置寄存器 BufCFG 的发送中断允许位 Rdy4TxiE 被置 1 了，当缓冲区有效时（缓冲区事件寄存器 BufEvent 的发送准备好位 Rdy4Tx 已设置），CS8900A 会向主机产生一个中断，指明主机可以开始写发送数据包。

③ 一旦 CS8900A 准备好接收发送数据包，主机循环执行 I/O 写指令，将数据写入接

收/发送数据端口（I/O 端口基地址+0000h），将整个数据包从主机内存写到 CS8900A 存储器。

CS8900A 网络适配器基于 I/O 模式的接收操作按以下步骤进行：

① CS8900A 接收到一个数据包，触发一个中断。

② 主机读入中断状态队列端口（I/O 端口基地址+0008h），获知 CS8900A 接收到一个数据包。

③ 主机循环执行 I/O 读操作，从 CS8900A 的接收/发送数据端口（I/O 端口基地址+0000h）将数据包从 CS8900A 的存储器读入主机内存。处理与数据包相关的内容有接收状态寄存器 RxStatus（PacketPage 基地址+0400h）和接收数据包长度寄存器 RxLength（PacketPage 基地址+0402h）。

以上我们对 CS8900A 的基本构成和工作原理作了概要介绍，网络设备驱动程序开发人员在开发设备驱动程序时还应参照网络控制芯片数据手册了解更多的细节，比如 CS8900A 中断事件有哪些，各配置寄存器中各位的含义等。

5.4.4　CS8900A 设备驱动程序分析

在理解了网络控制器芯片的内部结构、工作模式、内部寄存器与缓冲区构成、数据包接收/发送操作流程后，就可开始实现网络设备驱动程序代码了。下面我们就以 CS8900A 为例来分析有实际硬件操作情况下的驱动程序实现。CS8900A 的网络设备驱动程序定义在 driver/net/cs89x0.c 文件中。相应地对 CS8900 系列芯片各寄存器、I/O 端口地址、PacketPage 基地址、寄存器各位的定义在 drivers/net/cs89x0.h 文件中。

与我们在 5.3 节中介绍的 isa_skeletion.c 中的网络设备驱动程序示例类似， CS8900A 驱动程序的起始部分定义了一系列的局部变量，用于探测、申请 I/O 端口、中断、DMA 等时使用。

不同网络适配器网络设备的 I/O 端口地址配置列表与中断号映射表对是不同的，在 cs89x0.c 文件中可以看到，系统当前配置的网络设备的不同，定义的 I/O 端口地址列表与中断号映射表也不同。最后在内核配置时没有指定某特定的网络设备板时，列出 cs89x0 网络设备所有可能使用的 I/O 端口地址与中断号映射列表。

cs89x0.c 中的驱动程序设置了两个命令行参数，用户可以在系统启动时给出使用的 DMA 通道号 g_cs89x0_dma 与网络设备连接网络的介质接口配置信息 g_cs89x0_media_force。

```
//定义驱动程序的版本，驱动程序名称
static char version[] __initdata =
"cs89x0.c: v2.4.3-pre1 Russell Nelson <nelson@crynwr.com>, Andrew Morton\n";
#define DRV_NAME "cs89x0"
…
//以 0 结尾的 I/O 地址列表,用于探测 CS8900A 网络设备可能的 I/O 端口地址。定义以 0 结尾的中断号映射列表,指明 CS8900A
//的中断号与系统中断号的映射关系
static unsigned int netcard_portlist[] __used __initdata =
  { 0x300, 0x320, 0x340, 0x360, 0x200, 0x220, 0x240, 0x260, 0x280, 0x2a0, 0x2c0, 0x2e0, 0};
static unsigned int cs8900_irq_map[] = {10,11,12,5};
//网络设备实例的私有数据结构，保存设备自身的配置与统计信息，这样同一驱动程序同时支持多个网络设备工作时，这些信息
//对每个设备都有效
```

```
struct net_local {
  struct net_device_stats stats;          //网络设备的统计信息，如收发数据包数等
  int chip_type;                          //芯片类型，比如为 CS8900、 CS8920 或 CS8920M
  char chip_revision;                     //芯片类型后跟的扩展版本字符，如 CS8900A 中的“A”
  int send_cmd;                           //发送控制命令，可以为 TX_NOW、TX_AFTER_381 或 TX_AFTER_ALL
  int auto_neg_cnf;                       // 从 EEPROM 自动协商字
  int adapter_cnf;                        //从 EEPROM 配置适配器
  int isa_config;                         //从 EEPROM 配置 ISA
  int irq_map;                            //从 EEPROM 获取中断映射
  int rx_mode;                            //当前网络设备的接收模式 0，RX_MULTCAST_ACCEPT
                                          //RX_ALL_ACCEPT
  int curr_rx_cfg;                        //当前接收配置，接收配置寄存器 RxCFG 的一个拷贝
   …
  spinlock_t lock;
//使用 DMA 通道发送数据的相关定义
…
};
```

1．网络设备初始化

cs89x0 网络设备探测函数由两部分组成：一个是放在 net/Space.c 文件中由内核启动时调用的 cs89x0 网络设备自动探测函数 cs89x0_probe，它也是实际执行设备探测函数 cs89x0_probe1 的包装函数，它向 cs89x0_probe1 传递正确的参数，是在指定的某 I/O 端口地址探测设备还是探测所有可能的 I/O 端口地址，或是不探测。

（1）内核调用的网络设备探测函数

```
struct net_device * __init cs89x0_probe(int unit)
{
//为设备分配一个 net_device 数据结构实例，包括私有数据结构的大小，如分配不成功，则返回错误，退出探测过程
  struct net_device *dev = alloc_etherdev(sizeof(struct net_local));
  unsigned *port;
  int err = 0;
  int irq;
  int io;

  if (!dev)
       return ERR_PTR(-ENODEV);
//为网络设备分配设备名 ethi，查看在系统启动时用户有没有通过命令行参数给出网络设备端口地址、中断号等的配置信息。如
//果有，则将配置参数传给 dev 的数据域
  sprintf(dev->name, "eth%d", unit);
  netdev_boot_setup_check(dev);
  io = dev->base_addr;
  irq = dev->irq;
//调用实际的网络设备探测函数，如果 I/O 端口基地址不为空，在指定端口地址探测，
//如果为空，则在所有可能的端口地址处探测
  if (io > 0x1ff)  {                          //探测指定的 I/O 端口地址//
       err = cs89x0_probe1(dev, io, 0);
  } else if (io != 0) {                       //不探测//
       err = -ENXIO;
  } else {                                    //按列表探测所有可能的 I/O 端口地址//
       for (port = netcard_portlist; *port; port++) {
            if (cs89x0_probe1(dev, *port, 0) == 0)
                 break;
            dev->irq = irq;
       }
       if (!*port)                            //没有探测到设备，设置 ENODEV 错误代码
            err = -ENODEV;
  }
```

```
    if (err)
         goto out;
    return dev;                                    //探测成功，返回指向 dev 的指针
out:
    free_netdev(dev);                              //探测不成功，释放 let-device 实例
    …
```

（2）实际网络设备探测与初始化函数

如果 cs89x0 不是编译成模块装载，cs89x0 的实际探测函数 cs89x0_probe1 首先清空网络设备的私有数据，然后根据命令行参数初始化部分设备私有数据结构的数据域如：DMA 通道号、网络介质连接接口类型等。

随后 cs89x0_probe1 函数按照包装函数 cs89x0_probe 传来的 I/O 端口基地址探测设备是否存在，即调用 request_region 函数向系统申请指定的 I/O 端口资源。如该区域没有被别的设备占用，资源申请成功，cs89x0_probe1 函数从 I/O 基地址+0000h 单元读入设备标识符，对其进行比较，判断该设备是否为我们要找寻的网络设备。

参照 CS8900A 芯片手册（CS8900A_DATA SHEET）4.3.1 节的 CS8900A 总线接口寄存器的说明可知，在 CS8900A 的 PacketPage 基地址+0000h 处是产品标识代码寄存器。这是一个 32 位寄存器，该寄存器的布局为：

0000h 单元	0001h 单元	0002h 单元	0003h 单元
Crystal 半导体公司的 EISA 注册号的第一个字节	Crystal 半导体公司的 EISA 注册号的第 2 个字节	产品标识符的前 8 位	产品标识符的后 3 位

该寄存器中包含了一个 32 位的唯一产品标识符，指明是 CS8900A 芯片，以及何种芯片。驱动程序从该寄存器读出标识值，判断是否为自己要找的设备，是哪种类型的网络设备，由此确定网络设备的接收配置。寄存器中包含的值应为：

0000 1110 0110 0011 0000 0000 000X XXXX。在驱动程序中定义的符号与值：

0000 1110 0110 0011 ⟶ 63E0，CHIP_EISA_ID_SIG

对 CS8900A X XXXX 编码芯片的扩展版本字符，可以为：

Rev B：0 0111

Rev C：0 1000

Rev D：0 1001

Rev F：0 1010

在 cs89x0.c 中探测函数的实现代码为：

```
//向系统申请指定的 I/O 端口资源，若不成功，则返回错误。
if (!request_region(ioaddr & ~3, NETCARD_IO_EXTENT, DRV_NAME)) {
        …
        retval = -EBUSY;
        goto out1;
//从申请到的 I/O 端口地址读入产品标识符寄存器的值，查看是否与 CS8900A 产品标识号相符；如相符，用该 I/O 端口地址初
//始化 dev->base_addr 数据域
tmp = readword (ioaddr, DATA_PORT);
   if (tmp != CHIP_EISA_ID_SIG) {
   …
```

```
            retval = -ENODEV;
            goto out2;
      }
      dev->base_addr = ioaddr;
    //从产品标识寄存器中读入 CS8900A 的扩展版本，设置设备版本信息与发送配置获取芯片类型
      rev_type = readreg (dev, PRODUCT_ID_ADD);
      lp->chip_type = rev_type &~ REVISON_BITS;
      lp->chip_revision = ((rev_type & REVISON_BITS) >> 8) + 'A';

      //查看 CS8900 芯片类型和扩展版本字符，设置正确的发送控制命令
      //CS8900C 和 CS8900F 支持快速发送
      lp->send_cmd = TX_AFTER_381;
      if (lp->chip_type == CS8900 && lp->chip_revision >= 'F')
            lp->send_cmd = TX_NOW;
      if (lp->chip_type != CS8900 && lp->chip_revision >= 'C')
            lp->send_cmd = TX_NOW;
    //复位芯片
    reset_chip(dev);
```

随后驱动程序判别网络适配器上是否有外接 EEPROM，从 EEPROM 调用网络设备的配置参数来初始化 CS8900A 芯片的控制与配置寄存器。初始化设备的硬件地址（MAC 地址）、连接介质接口类型（RJ-45/同轴电缆等）、中断号映射表等。

最后 cs89x0_probe1 函数初始化设备驱动程序的函数指针，向系统注册设备。

上述源代码是通用 CS8900A 网络适配器的驱动程序，支持各种网卡，但嵌入式系统针对的是定制硬件，不需考虑是否配置别的硬件平台，代码会简化很多。

2．设备活动功能函数

从前面章节我们知道管理网络设备活动功能的函数包括打开/停止网络设备、数据接收/发送功能函数、中断等。针对某个具体网络控制芯片的驱动程序，涉及具体的硬件操作，函数实现会更复杂一些。

（1）open 方法：net_open

在 open 方法中为了让设备准备好接收和发送网络数据，net_open 函数要为设备申请支持设备活动的各种资源，包括：中断资源、DMA 通道、网络设备物理地址等。在 CS8900A 网络适配器中，为了给设备申请中断资源，要先允许设备产生中断，这由 CS8900A 的总线控制寄存器 BusCTL（位于 PacketPage 基地址+0116h）管理，BusCTL 的最高位 F 位 EnableRQ 置 1，即为允许设备产生中断。

```
static int net_open(struct net_device *dev)
{
  struct net_local *lp = netdev_priv(dev);
…
//写 CS8900A 总线控制寄存器，允许中断
        writereg(dev, PP_BusCTL, readreg(dev, PP_BusCTL)|ENABLE_IRQ );
//按中断映射列表申请中断资源，如果申请成功，将中断号写入 CS8900A 的中断寄存器，地址为 PacketPage 基地址 + 0022h
        for (i = 2; i < CS8920_NO_INTS; i++) {
              if ((1 << i) & lp->irq_map) {
                    if (request_irq(i, net_interrupt, 0, dev->name, dev) == 0) {
                          dev->irq = i;
                          write_irq(dev, lp->chip_type, i);
                          break;
                    }
              }
```

```
        }

//若中断申请不成功，写总线控制寄存器，禁止中断
        if (i >= CS8920_NO_INTS) {
              writereg(dev, PP_BusCTL, 0); /* 禁止中断 */
              printk(KERN_ERR "cs89x0: can't get an interrupt\n");
              ret = -EAGAIN;
              goto bad_out;
        }
```

如果允许以 DMA 方式发送数据，则需要为网络适配器申请 DMA 通道资源。

```
//如果允许 DMA 工作方式，首先为 DMA 发送分配数据缓冲区页面，如果分配不成功，则释放已申请的中断，退出
#if ALLOW_DMA
   if (lp->use_dma) {
        if (lp->isa_config & ANY_ISA_DMA) {
              unsigned long flags;
              lp->dma_buff = (unsigned char *)__get_dma_pages(GFP_KERNEL,
                                    get_order(lp->dmasize * 1024));

              if (!lp->dma_buff) {
                    printk(KERN_ERR "%s: cannot get %dK memory for DMA\n", dev->name, lp->dmasize);
                    goto release_irq;
              }
…
//为设备申请 DMA 通道，如果申请不成功则释放已分配的中断，退出；如果申请成功，将 DMA 通道号写入 DMA 通道寄存器
//write_dma
              memset(lp->dma_buff, 0, lp->dmasize * 1024);
              if (request_dma(dev->dma, dev->name)) {
                    printk(KERN_ERR "%s: cannot get dma channel %d\n", dev->name, dev->dma);
                    goto release_irq;
              }
              write_dma(dev, lp->chip_type, dev->dma);
//设置 DMA 的配置信息，比如 DMA 发送的缓冲区起始地址与结束地址，发送方式为一次 DMA 发送字节数等。
              lp->rx_dma_ptr = lp->dma_buff;
              lp->end_dma_buff = lp->dma_buff + lp->dmasize*1024;
              spin_lock_irqsave(&lp->lock, flags);
              disable_dma(dev->dma);
              clear_dma_ff(dev->dma);
              set_dma_mode(dev->dma, 0x14);
              set_dma_addr(dev->dma, isa_virt_to_bus(lp->dma_buff));
              set_dma_count(dev->dma, lp->dmasize*1024);
              enable_dma(dev->dma);
              spin_unlock_irqrestore(&lp->lock, flags);
        }
   }
#endif
//将网络设备的硬件地址 MAC 写入地址过滤器寄存器的地址寄存器（PacketPage 基地址+0158h）中允许网络设备工作于内存
//映射模式
   for (i=0; i < ETH_ALEN/2; i++)
        writereg(dev, PP_IA+i*2, dev->dev_addr[i*2] | (dev->dev_addr[i*2+1] << 8));
   //写总线控制寄存 BusCTL，允许内存映射模式
        writereg(dev, PP_BusCTL, MEMORY_ON);
```

接着网络设备查看设备连接网络介质接口类型，将接口类型写入 CS8900A 的线路控制寄存器（Line Control），允许网络设备的接收器与发送器工作。以上控制都由 CS8900A 的线路控制器 LineCTL（PacketPage 基地址+0112h）管理。LineCTL 布局如下：

7	6	5	4	3	2	1	0
SerTxOn	SerRxON	0	1	0	0	1	1
F	E	D	C	B	A	9	8
	LoRx Squelch	2-part DefDis	PolarityDis	Mod BackoffE		Auto	AUI/10BT

从 LineCTL 寄存器的布局可以看到，LineCTL 寄存器的第 7 位 SerTxOn、第 6 位 SerRxON 分别为网络适配器的发送器与接收器的开关。第 9、8 位的组合设置给出了当前网络连接介质的接口类型，如表 5-8 所示。

表 5-8　LineCTL 中第 8 位与第 9 位的组合设置及含义

Auto AUI/10BT	AUIonly	物 理 接 口
1	N/A	AUI
0	0	10BASE-T
0	1	自动选择

最后，open 方法设置设备的接收模式与发送工作模式，由 CS8900A 的接收配置寄存器、接收控制寄存器与发送配置寄存器实现。

接收控制寄存器 RxCTL（PacketPage 基地址 +0104h）的布局如下：

7	6	5	4	3	2	1	0
PromiscuousA	IAHashA	0	0	0	1	0	1
F	E	D	C	B	A	9	8
	ExtradataA	RuntA	CRCerrorA	BroadcastA	IndividualA	MulticastA	RxOKA

该寄存器标明了网络适配器的接收模式，比如混杂模式（由 PromiscuousA 位设置）、广播模式（BroadcastA），唯一主机地址（IndividualA），组发送（MulticastA）。如果为 RxOKA 位则说明网络设备接收到一个 CRC 校验正确的数据包。

接收配置寄存器 RxCFG（PacketPage 基地址 +0102h）的布局如下：

7	6	5	4	3	2	1	0
StreamE	Skip_1	0	0	0	0	1	1
F	E	D	C	B	A	9	8
	ExtradataiE	RuntiE	CRCerroriE	BufferCRC	AutoRx DMA	RxDMA only	RxOKiE

CS8900A 的接收配置寄存器标明了网络设备允许哪些中断，比如，如果 RxOKiE 位被设置了，当设备接收到一个正确数据包时会产生一个中断。

相应地在发送配置寄存器 TxCFG 中设置了 TxOKiE 位，数据包发送完成后网络设备会产生一个中断。

```
//打开接收器与发送器
   writereg(dev, PP_LineCTL, readreg(dev, PP_LineCTL) | SERIAL_RX_ON | SERIAL_TX_ON);
//设置网络适配器的接收模式，并写入 CS8900A 的接收控制寄存器 RxCTL 与接收配置寄存器 RxCFG，将发送工作模式写入发送
//配置寄存器 TxCFG
   lp->rx_mode = 0;
   writereg(dev, PP_RxCTL, DEF_RX_ACCEPT);
   lp->curr_rx_cfg = RX_OK_ENBL | RX_CRC_ERROR_ENBL;
   if (lp->isa_config & STREAM_TRANSFER)
```

```
        lp->curr_rx_cfg |= RX_STREAM_ENBL;
…
   writereg(dev, PP_RxCFG, lp->curr_rx_cfg);          //写接收配置寄存器
   writereg(dev, PP_TxCFG, TX_LOST_CRS_ENBL | TX_SQE_ERROR_ENBL | TX_OK_ENBL |TX_LATE_COL_ENBL |
   TX_JBR_ENBL | TX_ANY_COL_ENBL | TX_16_COL_ENBL);         //写发送配置寄存器

   writereg(dev, PP_BufCFG, READY_FOR_TX_ENBL | RX_MISS_COUNT_OVRFLOW_ENBL |TX_COL_COUNT_
   OVRFLOW_ENBL | TX_UNDERRUN_ENBL);                        //写缓冲区配置寄存器

   writereg(dev, PP_BusCTL, ENABLE_IRQ | (dev->mem_start?MEMORY_ON : 0)
//启动设备发送队列

netif_start_queue(dev);
```

（2）停止设备的方法：net_close

在 net-close 中释放已为设备分配的所有资源。

```
static int net_close(struct net_device *dev)
{
#if ALLOW_DMA
   struct net_local *lp = netdev_priv(dev);
#endif
//停止网络设备的发送队列
   netif_stop_queue(dev);
//分别清除网络适配器的接收配置寄存器 RxCFG、发送配置寄存器 TxCFG、缓冲区配置寄存 BufCFG、总线控制寄存器 BusCTL，
//释放中断，释放 DMA 通道，释放 DMA 的数据缓冲区
   writereg(dev, PP_RxCFG, 0);
   writereg(dev, PP_TxCFG, 0);
   writereg(dev, PP_BufCFG, 0);
   writereg(dev, PP_BusCTL, 0);
   free_irq(dev->irq, dev);
#if ALLOW_DMA
   if (lp->use_dma && lp->dma) {
         free_dma(dev->dma);
         release_dma_buff(lp);
   }
#endif
}
```

（3）发送数据包

CS8900A 网络适配器数据包发送函数方法 hard_start_xmit 的实现函数是 net_send_packet。net_send_packet 函数的流程与我们在 5.4.2 节中介绍的 CS8900A 芯片数据包发送流程一致，驱动程序先将发送命令与发送数据包长度写入芯片的相应寄存器，芯片接收到命令与数据包长度后，在芯片的缓冲区中为发送数据包分配缓冲区，随后主机将要发送的数据包复制到芯片，从驱动程序的角度，本次发送就结束了，余下的工作由 CS8900A 芯片硬件完成。

```
static int net_send_packet(struct sk_buff *skb, struct net_device *dev)
{
   struct net_local *lp = netdev_priv(dev);
   unsigned long flags;
…
//禁止本地中断，停止网络设备的发送队列
   local_irq_save(flags);
   netif_stop_queue(dev);
//将发送命令写入发送命令寄存器 TxCMD，发送数据帧长度写入发送长度寄存器 TxLength
```

```
    writereg(dev, PP_TxCMD, lp->send_cmd);
    writereg(dev, PP_TxLength, skb->len);
  //读总线状态寄存器 BusST，查看 CS8900 芯片为数据帧分配缓冲区是否成功，如果不成功，打开本地 CPU 中断，将数据帧重新
  //放入设备发送队列
    if ((readreg(dev, PP_BusST) & READY_FOR_TX_NOW) == 0) {
          local_irq_restore(flags);
          return 1;
    }
  //如果芯片分配发送缓冲区成功，将发送数据帧从主机内存拷贝到芯片缓冲区，设置发送起始时间，用于控制发送超时，释放主
  //机存放发送数据帧的 Socket Buffer
    skb_copy_from_linear_data(skb, (void *)(dev->mem_start + PP_TxFrame),
                      skb->len+1);

    local_irq_restore(flags);
    dev->trans_start = jiffies;
    dev_kfree_skb (skb);
    return 0;
  }
```

（4）接收数据包

数据包接收函数从网络设备驱动程序的中断程序执行现场调用，当网络设备硬件从网络介质上接收到一个本机应接收的数据包时，在中断允许的情况下，产生一个接收中断，中断服务程序读取 CS8900A 的中断队列，判断当前中断事件为接收中断，调用接收数据包处理函数 net_rx。

```
  static void net_rx(struct net_device *dev)
  {
    struct net_local *lp = netdev_priv(dev);
    struct sk_buff *skb;
    int status, length;
  //从数据帧接收端口读入接收数据帧状态与数据帧长度信息
    int ioaddr = dev->base_addr;
    status = readword(ioaddr, RX_FRAME_PORT);
    length = readword(ioaddr, RX_FRAME_PORT);
  //如果数据帧接收不正确，则更新接收错误统计信息，返回
    if ((status & RX_OK) == 0) {
          count_rx_errors(status, lp);
          return;
    }
  //如果数据帧接收正确，分配 Socket Buffer，若缓冲区分配不成功，则扔掉数据包，更新统计信息
    skb = dev_alloc_skb(length + 2);
    if (skb == NULL) {
          lp->stats.rx_dropped++;
          return;
    }
  //为了存放 MAC 协议头时能按 16 字节对齐预留两个字节的空间，因为以太网 MAC 的协议头为 14 个字节，随后从芯片读入数据
  //帧，写入 Socket Buffer
    skb_reserve(skb, 2);   /* longword align L3 header */

    readwords(ioaddr, RX_FRAME_PORT, skb_put(skb, length), length >> 1);
    if (length & 1)
          skb->data[length-1] = readword(ioaddr, RX_FRAME_PORT);
  //填写接收数据帧协议信息，调用 netif_rx 将数据帧上传给协议栈，更新接收统计信息
        skb->protocol=eth_type_trans(skb,dev);
    netif_rx(skb);
    lp->stats.rx_packets++;
    lp->stats.rx_bytes += length;
  }
```

（5）中断处理程序

在 CS8900A 芯片中有一个中断状态队列（Interrupt Status Queue，ISQ），中断状态队列可用于内存映射模式，也可用于 I/O 模式，ISQ 为主机提供 CS8900A 产生中断的原因信息。当一个事件发生触发了一个中断时，CS8900A 在 5 个寄存器中设置相应的位以记录中断事件，把这些寄存器的内容映射到 ISQ 寄存器中。映射到 ISQ 寄存器的有三个事件寄存器：接收事件寄存器 RxEvent、发送事件寄存器 TxEvent、缓冲区事件寄存器 BufEvent。另外两个寄存器是接收错误寄存器 RxMISS 与发送冲突 TxCOL 寄存器。

CS8900A 的中断服务程序一次要读出 ISQ 中的所有中断信息，根据中断类型调用相应的处理函数处理相应事件。

```
static irqreturn_t net_interrupt(int irq, void *dev_id)
{
  struct net_device *dev = dev_id;
  struct net_local *lp;
  int ioaddr, status;
  int handled = 0;

  ioaddr = dev->base_addr;
  lp = netdev_priv(dev);
//从 ISQ 中读出所有中断信息，根据中断产生原因做相应处理
  while ((status = readword(dev->base_addr, ISQ_PORT))) {
        handled = 1;
        switch(status & ISQ_EVENT_MASK) {
        case ISQ_RECEIVER_EVENT:                    //接收数据帧事件
              net_rx(dev);                          //调用数据帧接收处理函数
              break;
        case ISQ_TRANSMITTER_EVENT:                 //发送数据帧完成事件
              lp->stats.tx_packets++;               //更新发送统计信息
              netif_wake_queue(dev);                //唤醒设备发送队列
//根据发送事件产生原因，更新相应的发送事件的统计信息，如数据发送正确完成，数据发送发生冲突等。
              if ((status & (  TX_OK |TX_LOST_CRS |TX_SQE_ERROR |
                         TX_LATE_COL |TX_16_COL)) != TX_OK) {
                   if ((status & TX_OK) == 0) lp->stats.tx_errors++;
                   if (status & TX_LOST_CRS) lp ->stats.tx_carrier_errors++;
                   if (status & TX_SQE_ERROR) lp ->stats.tx_heartbeat_errors++;
                   if (status & TX_LATE_COL) lp ->stats.tx_window_errors++;
                   if (status & TX_16_COL) lp ->stats.tx_aborted_errors++;
              }
              break;
        case ISQ_BUFFER_EVENT:                      //缓冲区事件中断
              if (status & READY_FOR_TX) {
                   netif_wake_queue(dev);           //重启设备发送队列
              }
…
        case ISQ_RX_MISS_EVENT:
              lp->stats.rx_missed_errors += (status >>6);
              break;
        case ISQ_TX_COL_EVENT:
              lp->stats.collisions += (status >>6);
              break;
        }
  }
  return IRQ_RETVAL(handled);
}
```

（6）发送超时处理函数

在 CS8900A 的发送超时处理函数 net_timeout 中仅仅重启网络设备的发送队列：netif_wake_queue(dev)。

3．设备管理/统计方法

CS8900A 驱动程序中实现的设备管理/统计方法 net_get_stats 向调用者返回设备的各种统计信息，这些统计信息在驱动程序的各处更新，存放于代表网络设备的 net-device 数据结构实例的私有数据结构中。CS8900A 驱动程序没有实现自己私有的 ioctl 命令，支持内核实现的标准 ioctl 命令。

```
static struct net_device_stats * net_get_stats(struct net_device *dev)
{
  struct net_local *lp = netdev_priv(dev);
  unsigned long flags;

  spin_lock_irqsave(&lp->lock, flags);
//从设备寄存器中读入最新统计信息，并存于设备私有数据结构
  lp->stats.rx_missed_errors += (readreg(dev, PP_RxMiss) >> 6);
  lp->stats.collisions += (readreg(dev, PP_TxCol) >> 6);
  spin_unlock_irqrestore(&lp->lock, flags);
//返回存放设备统计信息的设备私有数据结构
return &lp->stats;
}
```

4．组发送功能方法

CS8900A 实现的组发送功能方法 set_multicast_list 根据设备设置的数据接收方式（比如：混杂模式、组发送模式等），将控制信息写入 CS8900A 芯片的接收控制寄存器 RxCTL 与接收配置寄存器 RxCFG。CS8900A 会根据设置来过滤从网络上发送来的数据包地址。

```
static void set_multicast_list(struct net_device *dev)
{
  struct net_local *lp = netdev_priv(dev);
  unsigned long flags;

  spin_lock_irqsave(&lp->lock, flags);
//如果设备工作于混杂模式，设置设备私有数据结构的接收模式为接收所有数据包
  if(dev->flags&IFF_PROMISC)
  {
        lp->rx_mode = RX_ALL_ACCEPT;
  }
//如设备工作于组发送模式，或组发送地址列表不为空，设置设备私有数据结构的接收模式为接收组发送数据包
  else if((dev->flags&IFF_ALLMULTI)||dev->mc_list)
  {
        lp->rx_mode = RX_MULTCAST_ACCEPT;
  }
  else
        lp->rx_mode = 0;
//将接收模式写入 CS8900A 芯片的接收控制寄存器 RxCTL 与接收配置寄存器 RxCFG
  writereg(dev, PP_RxCTL, DEF_RX_ACCEPT | lp->rx_mode);
  writereg(dev, PP_RxCFG, lp->curr_rx_cfg |
      (lp->rx_mode == RX_ALL_ACCEPT? (RX_CRC_ERROR_ENBL|RX_RUNT_ENBL|RX_EXTRA_DATA_ENBL) : 0));
  spin_unlock_irqrestore(&lp->lock, flags);
}
```

在 CS8900A 的驱动程序中还实现了支持 DMA 发送的一系列方法及其辅助函数，比如从 CS8900A 寄存器中读入一个字，向 CS8900A 寄存器写一个字等。由此可见我们在实现网络设

备驱动程序时，首先可以按照图 5-1 中给出的网络设备驱动程序的构架完成驱动程序主体函数的开发，然后按照实际的需要实现主体函数所需调用的辅助功能函数和其他可选功能函数。例如在 CS8900A 的驱动程序中提供了修改网络设备硬件地址的方法 set_mac_address。从我们对 CS8900A 的驱动程序的分析可以看到，各方法函数的主要流程与我们提供的模板流程一致，只不过在此基础上加入了实际的硬件操作，这必须参考具体的网络控制芯片手册来实现。

5.5　本章总结

随着网络设备硬件的不断发展，新的网络设备硬件层出不穷，因此网络设备驱动程序也是 Linux 内核网络子系统中需要最频繁更新的一个部分。但无论设备硬件怎么发展，设备与内核通信的机制却没有变，网络设备驱动程序的架构也是一致的，变化的主要是一些实现细节。因此，编写网络设备驱动程序最好、最快捷的方式就是在理解驱动程序的架构的基础上，以一个类似的、已实现的网络设备驱动程序为模板，实现新驱动程序的基本功能，再做细节上的修改。为了掌握网络设备驱动程序的开发技巧，驱动程序开发人员在理解 net_device 数据结构与网络设备创建、初始化过程的基础上，需要掌握以下内容。

- 网络设备驱动程序的架构：它是网络设备驱动程序应完成的最基本功能模块。
- 网络设备硬件与内核通信的机制：Linux 实现内核与硬件通信的快速、高效，数据分段处理，如硬件中断与软件中断。
- 网络设备驱动程序各功能模块的任务：本章以代码模板方式给出。
- 网络设备驱动程序开发的技巧与过程：要掌握网络设备控制器的硬件特点，以实例为参考实现。

第 6 章　数据链路层数据帧的收发

本章主要介绍网络数据在 Linux 内核 TCP/IP 协议栈中数据链路层的接收与发送的处理过程，在分析了内核中管理网络数据包、网络设备队列的数据结构的基础上，讲解了数据收/发基本功能函数、负责接收/发送的软件中断代码的实现。在本章最后描述了数据链路层与网络层之间数据接收管理接口，为下一章的内容做了铺垫。

在前面的章节中，我们了解了：网络设备驱动程序如何通过中断通知内核，使网络设备从网络介质上接收到数据帧；设备驱动程序如何在中断服务程序中，将数据从设备硬件缓冲区复制到内核地址空间的 Socket Buffer；内核如何调用设备驱动程序的方法将数据帧发送到网络介质上。

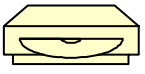
网络设备驱动程序是 Linux 内核 TCP/IP 协议栈底层的数据收/发软件，是整个协议栈中直接与网络设备硬件交互的代码部分。不同的网络设备驱动程序因网络设备硬件不同，其代码实现也不同。Linux 内核的网络子系统结合 net_device 数据结构与数据链路层处理网络数据帧的方法，实现了 TCP/IP 协议栈上层协议与网络设备硬件之间的统一接口，该接口对上层协议栈屏蔽了网络设备的硬件细节以及设备驱动程序之间的差异。其中 net_device 数据结构的详细定义我们已在第 3 章讲解了，而 net_device 数据结构的应用与函数指针的方法实现在第 5 章做了详细描述。本章我们就将分析网络数据帧在数据链路层传递时的处理。

1．数据包在数据链路层的接收处理

在第 5 章网络设备驱动程序实现的数据帧接收方法 net_rx 中，我们看到网络设备驱动程序在将数据从设备硬件缓冲区复制到内核地址空间后，调用的是数据链路层的 netif_rx 函数将数据帧向上层推送。netif_rx 函数是现在大多数网络设备驱动程序与数据链路层接收数据帧处理之间的 API。Linux 内核从 2.5 版本后在数据链路层实现了一种新的处理输入数据帧的 API，称为 NAPI。目前只有少数网络设备的驱动程序升级到了 NAPI 上，因为这需要网络设备硬件与设备驱动程序支持。在本章讲解数据帧接收处理实现时，我们会介绍这两种 API 的实现原理。

2．数据包在数据链路层的发送处理

向外发送数据帧时，需要先处理两个问题：其一，如果数据帧由本地主机产生，存放在 Socket Buffer 中的网络数据帧已由 TCP/IP 协议栈的上层协议实例处理完成，这时如何通过数据链路层的方法来启动网络设备硬件将数据帧发送到网络介质上；其二，如果向外发送的数据帧不是本机产生，是本机从网络上接收到的数据帧，需要通过本机继续向前发送，这样的数据帧发送路径应如何确定。


文件路径

数据链路层 API 在 Linux 源文件的目录树： net/core/dev.c 文件中。

数据帧的接收、发送和前送这三种情况在 TCP/IP 协议栈中的发送途径不同，如图 6-1 所示。

从图 6-1 可以看到，确定前送数据帧的路径，主要是在网络层完成的。从数据链路层的角度来看，前送数据帧与对本机产生的输出数据帧的处理没有差别。本章要讨论的函数全部在以下路径中。

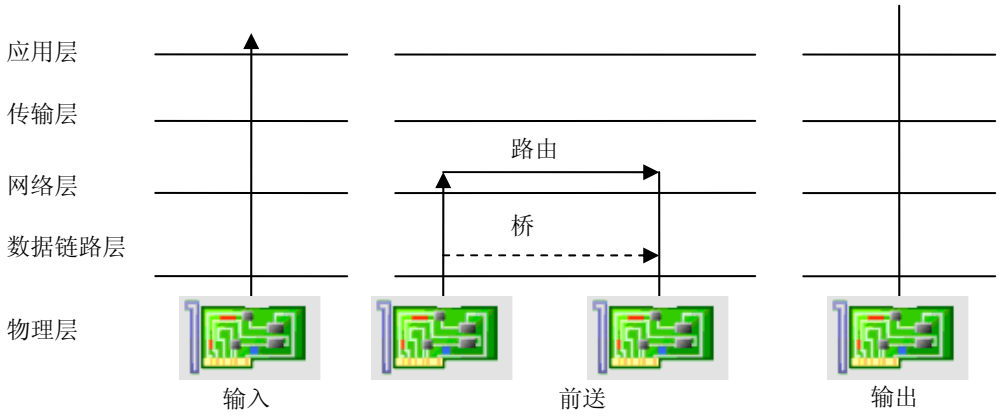


图 6-1　网络数据包的传送路径

在数据链路层有几种活动，来完成网络设备硬件与 TCP/IP 协议栈上层协议之间的接口任务，如图 6-2 所示。

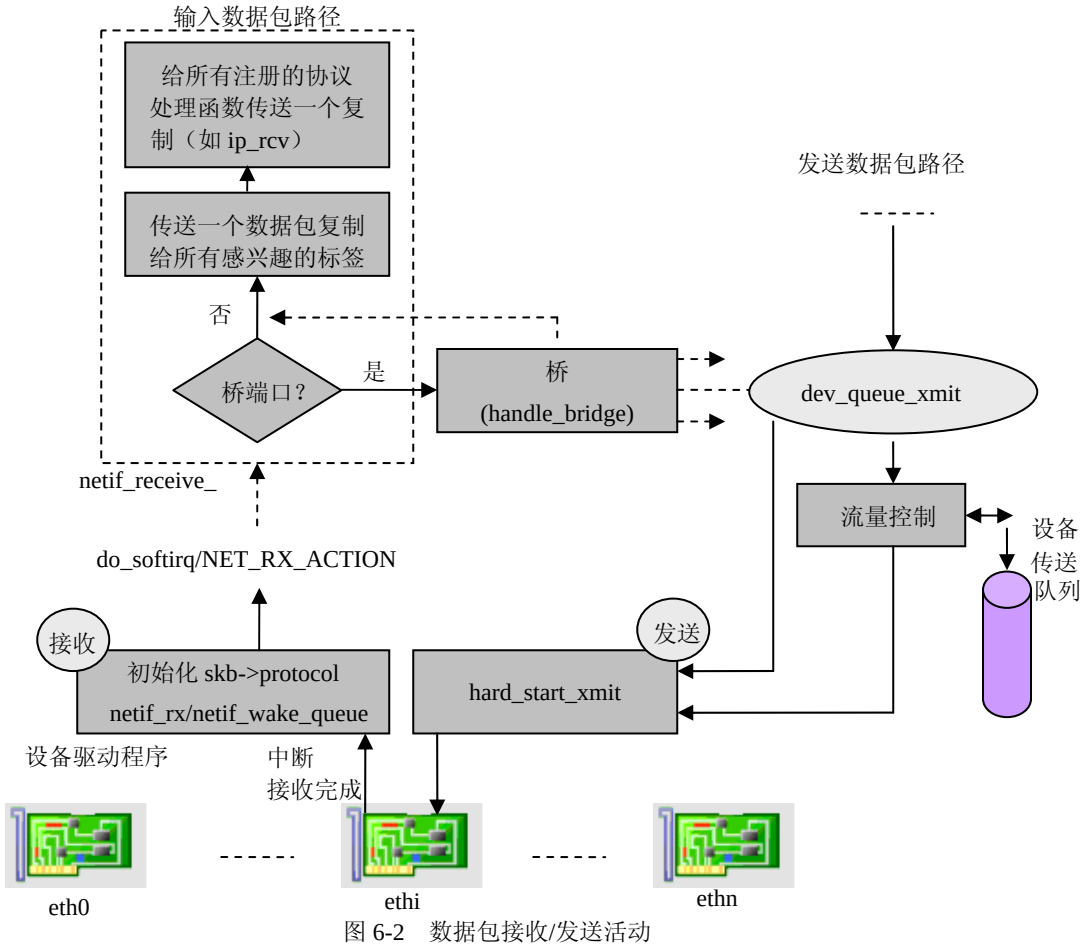


图 6-2　数据包接收/发送活动

在图 6-2 中，前送数据帧的路径也可以在数据链路层由桥来选择决定，限于篇幅，本书就不再讲解桥在 Linux 内核 TCP/IP 协议栈中的实现了。

6.1 关键数据结构

在 TCP/IP 协议栈中，数据链路层的关键任务是：将由网络设备驱动程序从设备硬件缓冲区复制到内核地址空间的网络数据帧挂到 CPU 的输入队列，并通知上层协议有网络数据帧到达，随后上层协议就可以从 CPU 的输入队列中获取网络数据帧并处理。

另外，上层协议实例要向外发送的数据帧会由数据链路层放到设备输出队列，再由设备驱动程序的硬件发送函数 hard_start_xmit 将设备输出队列中的数据帧复制到设备硬件缓冲区，实现对外发送。

在这一章描述数据链路层的工作流程时，会常提到接收队列（ingress queue）和发送队列（egress queue）。每个队列都有两个指针：一个指向实现发送操作的网络设备实例，一个指向存放数据帧的 Socket Buffer。发送队列直接与网络设备相关，流量控制（Traffic Control 子系统中的 QoS）子系统为每个网络设备定义了一个或多个队列，Linux 内核只需跟踪等待发送数据帧的网络设备信息，而不是数据帧信息；而对于输入数据帧，输入队列与每个 CPU 相关，每个 CPU 有自己的输入队列。在介绍数据链路层的工作原理前，我们需要对管理数据帧队列的数据结构有所了解。

6.1.1 struct napi_struct 数据结构

Linux 内核 2.5 以后的版本在数据链路层实现了一组新的处理网络输入数据帧的 API，即 NAPI，其具体工作流程我们会在 6.2 节中分析。在现有网络设备中只有少数网络设备驱动程序升级到了 NAPI，除 struct net_device 数据结构描述的网络设备基本属性外，内核还定义了一个新的数据结构 struct napi_struct 来管理这类设备的新特性与操作。系统运行过程中，支持 NAPI 模式的网络设备收到网络数据帧后，该网络设备的 struct napi_struct 数据结构的实例会放到 CPU 的 struct softnet_data 数据结构的 poll_list 链表中；网络子系统的接收软件中断 NET_RX_SOFTIRQ 被调度执行时，在 poll_list 链表中的设备的 poll 方法会被依次调用执行，一次读入设备硬件缓冲区中的多个输入数据帧。

支持 NAPI 发送的网络设备具有这样的特性：由 poll 函数将设备硬件缓冲区中数据复制到内核，poll 函数在接收软件中断中执行，一次可读入多个数据帧。由此可以设想 struct napi_struct 数据结构的设计应考虑以下因素：

- 在 SMP 系统中，设备的 poll 函数当前在哪个 CPU 上执行是由网络设备的 struct napi_struct 数据结构放在哪个 CPU 的 poll_list 链表上决定的。
- 虽然 NAPI 设备一次可读入多个数据帧，但每个 CPU 的输入队列长度有限，这也决定了我们应该对一次从设备读入的数据帧数量进行限制。
- NAPI 设备的 poll 函数，是在网络子系统的接收软件中断中执行的，由于软件中断不能执行太长的时间，以免占用 CPU 资源太久，影响其他任务的处理，这就决定了我们需要限制每个设备 poll 函数的执行时间，称为执行预算。
- 当前以 NAPI 方式接收网络数据的是哪个网络设备。
- 当前 NAPI 设备的状态：poll 函数是正在等待被调用执行，还是已执行完成等。

这些因素决定了 struct napi_struct 数据域的设置，struct napi_sturct 数据结构定义在 include/linux/netdevice.h 文件中。

```
struct napi_struct {                                          //include/linux/netdevice.h
  struct list_head poll_list;
  unsigned long          state;
  int              weight;
  int              (*poll)(struct napi_struct *, int);
#ifdef CONFIG_NETPOLL
  spinlock_t       poll_lock;
  int              poll_owner;
#endif
  struct net_device      *dev;
  struct list_head dev_list;
  struct sk_buff         *gro_list;
  struct sk_buff         *skb;
};
```

struct napi_struct 数据结构中各数据域的含义如下。

- poll_list：将网络设备的 struct napi_struct 数据结构实例链接到 CPU poll_list 列表中的数据域。poll_list 列表中的网络设备都是已产生中断的网络设备，它们通知 CPU 收到了网络数据帧。而它们的 poll 函数等待在接收软件中断中被调用执行，来读入网络数据帧。
- state：poll_list 列表中的 NAPI 网络设备，只由 state 变量中的 NAPI_STATE_SCHED 位管理；即 state 的 NAPI_STATE_SCHED 标志置位的网络设备，其 napi_struct 数据结构实例会加入到 CPU 的 poll_list 列表中，NAPI_STATE_SCHED 位清除的设备会从 poll_list 列表中移走。
- weight：定义了每个设备的 poll 函数在 NET_RX_SOFTIRQ 期间执行时，可以从设备缓冲区中读入的最大数据帧数，以太网卡的默认值为 64，该值由 net/core/dev.c 文件中的 weight_p 变量值给出。内核将 weight_p 的值在/proc 文件系统的/proc/sys/net/core/dev_weight 文件中输出，用户可以读取与修改该值。
- poll：指向 NAPI 网络设备驱动程序的 poll 函数的指针。
- *dev：指向接收数据帧的包网络设备的 struct net_device 数据结构实例。该数据结构描述了该网络设备的通用属性。
- dev_list：内核的全局网络设备列表。该列表管理系统中所有网络设备的 struct net_device 数据结构实例。
- *gro_list：数据帧被分割成片段后，所有分片数据 Socket Buffer 的列表。
- *skb：存放接收数据帧的 Socket Buffer。

6.1.2　struct softnet_data 数据结构

每个 CPU 都有自己的队列来管理输入数据帧，正因为如此，每个 CPU 都有自己的数据结构来管理网络数据的输入和输出流量，这样就没必要对数据操作加锁定机制了。管理 CPU 队列的数据结构为 struct softnet_data，定义在 include/linux/netdevice.h 文件中。

```
struct softnet_data                                   //include/linux/netdevice.h
{
```

```
    struct Qdisc           *output_queue;
    struct sk_buff_head    input_pkt_queue;
    struct list_head poll_list;
    struct sk_buff         *completion_queue;
    struct napi_struct     backlog;
};
```

struct softnet_data 数据结构中包含了用于接收和发送两方面操作的数据域，网络的输入数据帧会放在 input_pkt_queue 链表中；网络输出数据帧放在专门的流量管理数据结构中，由流量控制子系统处理，不由软件中断和 struct softnet_data 数据结构管理，但事后软件中断会负责清除已发送完的 Socket Buffer。

以下就对 struct softnet_data 数据结构的数据域进行说明，有的细节,比如用于 NAPI 的 struct napi_struct 数据结构在 6.1.1 节中已描述，不再赘述。有的驱动程序使用 NAPI 接口，有的网络设备还没有升级到 NAPI 上，不过在数据链路层对以上两种驱动程序都支持，都会使用 struct softnet_data 数据结构。

struct softnet_data 数据结构中各数据域的含义如下。

- *output_queue：这是流量控制子系统用于专门管理网络设备输出队列的数据结构，该数据结构的详细描述将在 6.4 节中介绍数据链路层的数据帧输出时给出。
- input_pkt_queue：每个 CPU 的输入队列，该队列是存放网络输入数据帧 Socket Buffer 的链表。input_pkt_queue 是输入队列链表的起始地址。
- poll_list：网络设备链表，在该链表中的网络设备是以 NAPI 方式接收网络数据的设备，这些设备的驱动程序实现了 poll 函数，用来从设备接口中读入数据帧。链表中的网络设备接口中有从网络上收入的数据帧，它们的 poll 函数，会在内核调度网络子系统的接收软件中断（NET_RX_SOFTIRQ）处理程序 net_rx_action 时被调用执行，poll 函数的任务是将设备硬件缓冲区中的数据帧复制到内核地址空间的 Socket Buffer，放到 CPU 输入队列。
- *completion_queue：这是 Socket Buffer 的列表，这些套接字缓冲区中的数据已被成功发送或接收，缓冲区可以释放。对缓冲区的释放过程会在网络子系统的发送软件中断（NET_TX_SOFTIRQ）的处理程序 net_tx_action 中进行。
- Backlog：struct napi_struct 数据结构类型的变量（见 6.1.1 节），其中的 poll 函数指针在网络子系统初始化时，统一初始化为指向 process_backlog 函数。process_backlog 函数的任务是将 CPU 输入队列 input_pkt_queue 中的数据帧向上层协议实例推送。

每个 CPU 都有各自的 struct softnet_data 数据结构实例，访问 struct softnet_data 数据结构实例时不需锁定机制。Struct softnet_data 数据结构的初始化由网络子系统的初始化函数 struct net_dev_init 在内核启动时赋值（见 4.2.3 节）。

```
static int __init net_dev_init(void)
{
…
   //初始化各 CPU 的网络数据包接收队列，首先获取各 CPU，由局部变量 queue 指向各 CPU 的 struct softnet_data 数据
   //结构实例，初始化 struct softnet_data 数据结构的数据域
   for_each_possible_cpu(i) {
        struct softnet_data *queue;
```

```
        queue = &per_cpu(softnet_data, i);
        skb_queue_head_init(&queue->input_pkt_queue);       //初始化输入队列
        queue->completion_queue = NULL;                     //初始化完成传送队列
        INIT_LIST_HEAD(&queue->poll_list);                  //初始化 NIAP 设备队列
        //初始化队列的 napi_struct 数据域，比如 poll 函数指针与 poll 函数的执行预算
        queue->backlog.poll = process_backlog;
        queue->backlog.weight = weight_p;
        queue->backlog.gro_list = NULL;
    }
…
```

在上述网络子系统初始化代码片段中，处理 CPU 输入队列上数据帧的函数 queue->backlog.poll 初始化为 process_backlog，这是内核应用 NAPI 数据结构来优化输入队列数据帧处理的方式。

在 6.2.2 节中我们介绍数据链路层数据帧接收函数 netif_rx 实现分析后可以看到，当 netif_rx 函数把网络数据帧放入 CPU 的输入队列 input_pkt_queue 后，netif_rx 函数调用 napi_schedule 把 CPU 队列的 struct napi_struct 数据结构实例放入当前 CPU 的 poll_list 列表中，并标记网络子系统的接收软件中断 NET_RX_SOFTIRQ。随后内核调度接收软件中断 NET_RX_SOFTIRQ 时，会执行接收软件中断的处理程序 net_rx_action，而 net_rx_action 会展开 CPU softnet_data->poll_list 列表中的所有实体，执行它们的 poll 函数，这样将输入队列中数据帧推送给上层协议的 process_backlog 函数就会在软件中断中被调用执行。

每个 CPU 输入队列 input_pkt_queue 的最大长度由变量 netdev_max_backlog（同样定义在 net/core/dev.c 文件中）的值定义，默认值为 1000，即输入队列中最多可以容纳 1000 个数据帧。内核将该值通过/proc 文件系统，在/proc/sys/net/core 目录下的 netdev_max_backlog 文件中输出，用户也可以修改该值。

在数据链路层数据帧接收/发送过程中，操作最多的就是以上两个数据结构。另外与操作相关的几个常量还包括：输入队列的长度、poll 函数一次可从设备读入的数据帧数、poll 函数一次执行时间的预算。以上几个值的默认值都定义在 net/core/dev.c 文件中，分别如下：

```
int netdev_max_backlog __read_mostly = 1000;    //CPU 输入队列长度
int netdev_budget __read_mostly = 300;          //软件中断一次可处理的最大数据帧数
int weight_p __read_mostly = 64;                //poll 函数可从设备读入的数据帧数
```

6.2　数据帧的接收处理

网络设备收到的数据帧由网络设备驱动程序推送到内核地址空间后，在数据链路层中如何将数据帧放入 CPU 的输入队列呢？Linux 内核网络子系统实现了两种机制。

1. netif_rx

这是目前大多数网络设备驱动程序将数据帧复制到 Socket Buffer 后，调用的数据链路层方法。它通知内核接收到了网络数据帧；标记网络接收软件中断，执行接收数据帧的后续处理。这种机制每接收一个数据帧会产生一个接收中断。

2. NAPI

这是内核实现的新接口，在一次中断中可以接收多个网络数据帧，减少了 CPU 响应中

断请求进行中断服务程序与现行程序之间切换所花费的时间。目前只有少数网络设备升级到了这种工作方式，driver/net/tg3.c 是首先转换到 NAPI 上的驱动程序。

本节接下来就讨论在这两种工作模式下，数据帧在数据链路层的接收处理流程，并分析它们的实现源码。

6.2.1 NAPI 的实现

NAPI 是 Linux 网络子系统中实现的一种新的提高网络处理效率的技术，它结合中断和轮询两种工作模式，在一次中断后从网络设备中读取多个网络数据包，在高负载的情况下可以获取更好的网络数据处理性能，减轻 CPU 的负担。

在原有的网络数据接收模式下，网络设备驱动程序每收到一个数据帧就产生一次中断；在高负载的网络中，频繁地产生中断需要 CPU 花大量的时间来处理中断请求和程序切换，造成系统资源浪费。NAPI 的核心概念是使用中断与轮询相结合的方式来代替纯中断模式：当网络设备从网络上收到数据帧后，向 CPU 发出中断请求，内核执行设备驱动程序的中断服务程序接收数据帧；在内核处理完前面收到的数据帧前，如果设备又收到新的数据帧，这时设备不需产生新的中断（设备中断为关闭状态），内核继续读入设备输入缓冲区中的数据帧（通过调用驱动程序的 poll 函数来完成），直到设备输入缓冲区为空，再重新打开设备中断。这样设备驱动程序同时具有了中断与轮询两种工作模式的优点。

- 异步事件通知：当有数据帧到达时用中断通知内核，内核无须不断查询设备的状态。
- 如果设备队列中仍有数据，无须浪费时间处理中断通知、程序切换等。

NAPI 工作模式在内核中已实现，网络设备要利用 NAPI 工作模式的特性，需将网络设备驱动程序做以下的升级：

① 增加新的数据结构。

支持 NAPI 工作模式的网络设备，除了有一个 struct net_device 数据结构实例描述网络设备的通用属性外，还需一个 struct napi_struct 数据结构实例描述与 NAPI 相关的属性与操作。

例如，driver/net/tg3.c 是支持 NAPI 工作模式的网络设备驱动程序，在定义自己的设备数据结构 struct tg3 时，既包含了 struct net_device 数据结构实例，又包含了 struct napi_struct 数据结构实例。struct tg3 数据结构定义在 driver/net/tg3.h 文件中。

```
struct tg3 {
  //driver/net/tg3.h
…
struct net_device        *dev;
struct pci_dev           *pdev;
…
struct napi_struct       napi;
  void                   (*write32_rx_mbox) (struct tg3 *, u32, u32);
…
```

② 实现 poll 函数。

设备驱动程序要实现自己的 poll 函数，来轮询自己的设备，将网络输入数据帧复制到内核地址空间的 Socket Buffer，再放入 CPU 输入队列。

③ 对接收中断处理程序进行修改。

执行中断处理程序后，不是调用 netif_rx 函数将 Socket Buffer 放入 CPU 输入队列，而

是调用 netif_rx_schedule 函数。

struct softnet_data 数据结构、struct napi_struct 数据结构与 struct net_device 数据结构之间的关系如图 6-3 所示。

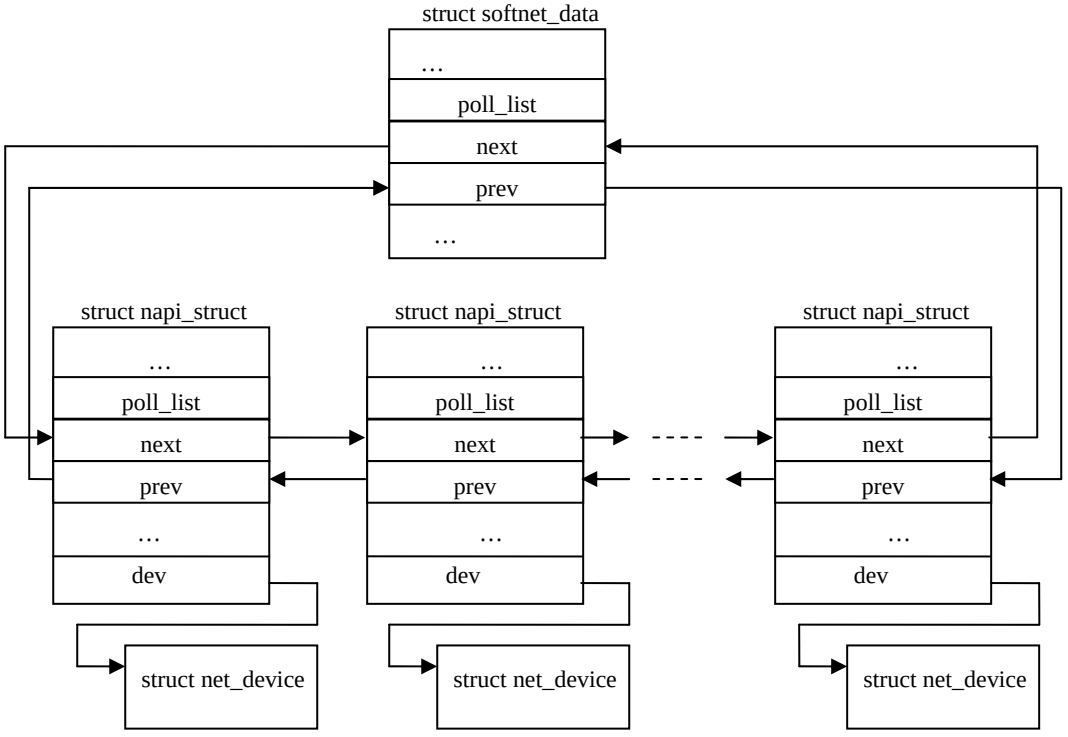


图 6-3　struct softnet_data、struct napi_struct、struct net_device 数据结构之间的关系

从网络设备驱动程序的角度看，按 NAPI 方式工作和不按 NAPI 方式工作的网络设备有两点不同：前者实现 poll 函数，并且通过__netif_rx_schedule 函数而不是 netif_rx 函数将接收到的数据帧上传给 TCP/IP 协议栈。

1．NAPI 的工作流程

图 6-4 给出了按 NAPI 机制接收网络数据帧的工作流程，网络设备驱动程序接收中断处理程序、__netif_rx_schedule 函数、接收软件中断处理函数 net_rx_action 与设备驱动程序 poll 函数之间的调用关系。

首先网络设备从网络介质上接收到网络数据帧后，存入网络设备的硬件缓冲区，然后向 CPU 发出中断请求，CPU 响应设备中断请求后执行接收中断处理程序，网络设备接收中断处理程序将数据复制到内核地址空间的 Socket Buffer 中最后调用 netif_rx_schedule 函数。

随后 netif_rx_schedule 函数将 Socket Buffer 挂到 CPU 的输入数据帧队列，查看设备中是否有新的数据帧进入，将设备的 struct napi_struct 数据结构实例放到 CPU 的 poll_list 队列，挂起接收软件中断 NET_RX_SOFTIRQ。

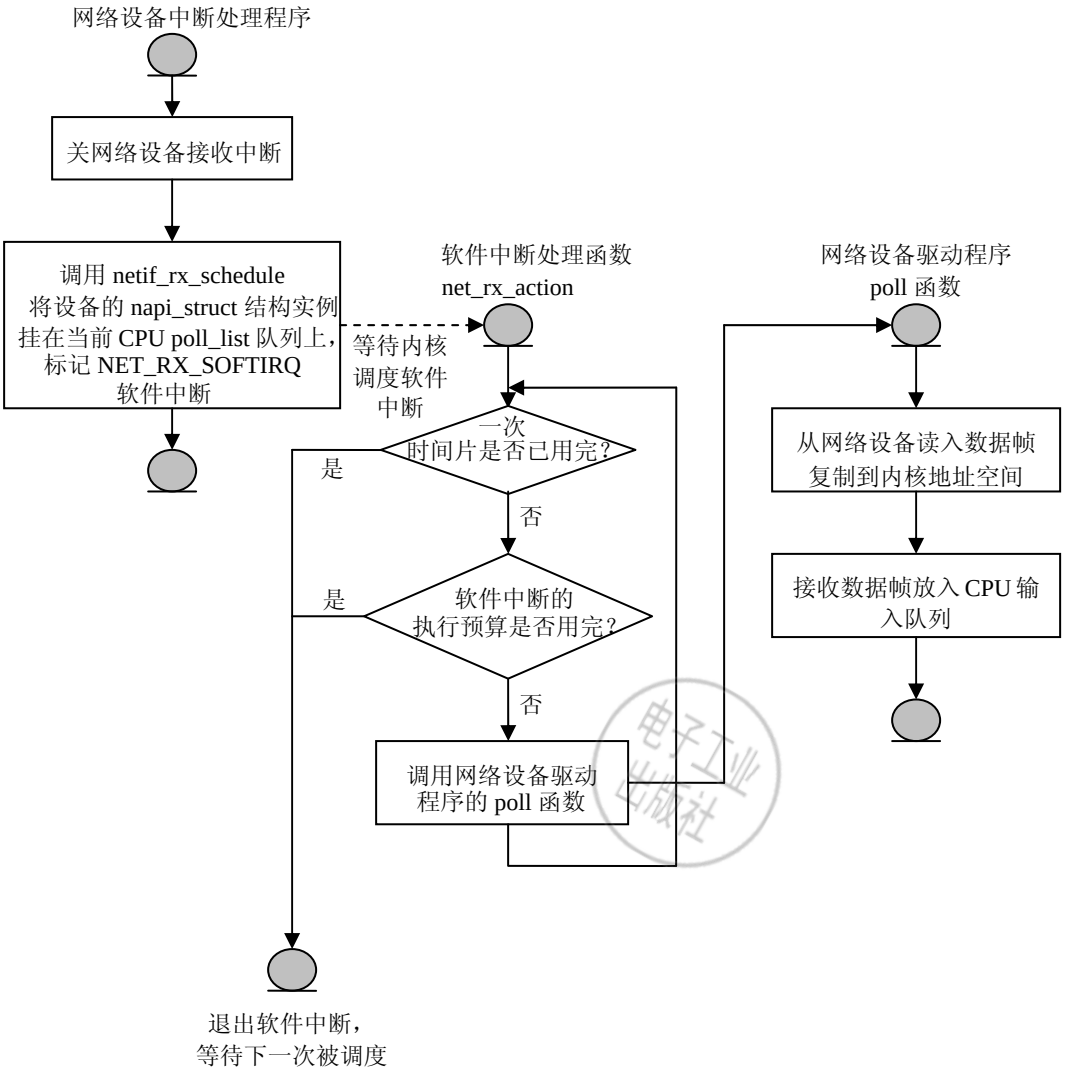


图 6-4　NAPI 接收机制的工作流程

当接收软件中断被调度执行时，其处理函数 net_rx_action 调用 CPU 的 poll_list 队列中设备的 poll 函数，来读入网络设备接收到的新数据帧，直到以下的条件满足，退出软件中断：

- 一次时间片用完。
- 软件中断的执行预算用完。
- CPU 的输入队列满。

设备驱动程序的 poll 函数的功能就是轮询网络设备，查看设备中是否有新的数据到达，如果有，就一直从网络设备中读入数据帧，复制到内核地址空间的 Socket Buffer 中，把 Socket Buffer 放入 CPU 的输入队列。当 poll 函数读入的数据帧数已达到一次能读入的最大值时，或设备中已无新数据时，停止调用设备的 poll 函数。

2．实现 NAPI 的关键函数

在介绍 NAPI 的工作原理和工作流程时我们提到，网络设备的接收中断处理程序调用__netif_rx_schedule 或 netif_rx_schedule 函数，将数据帧发送给上层协议，并将网络设备的 poll 函数放入 CPU 的 poll_list 队列中。netif_rx_schedule 是__netif_rx_schedule 的包装函数，netif_rx_schedule 函数查看设备是否正在接收数据帧，但软件中断还没有调度。内核中实现 NAPI 功能的有以下关键函数（各函数间的调用关系如图 6-5 所示）：

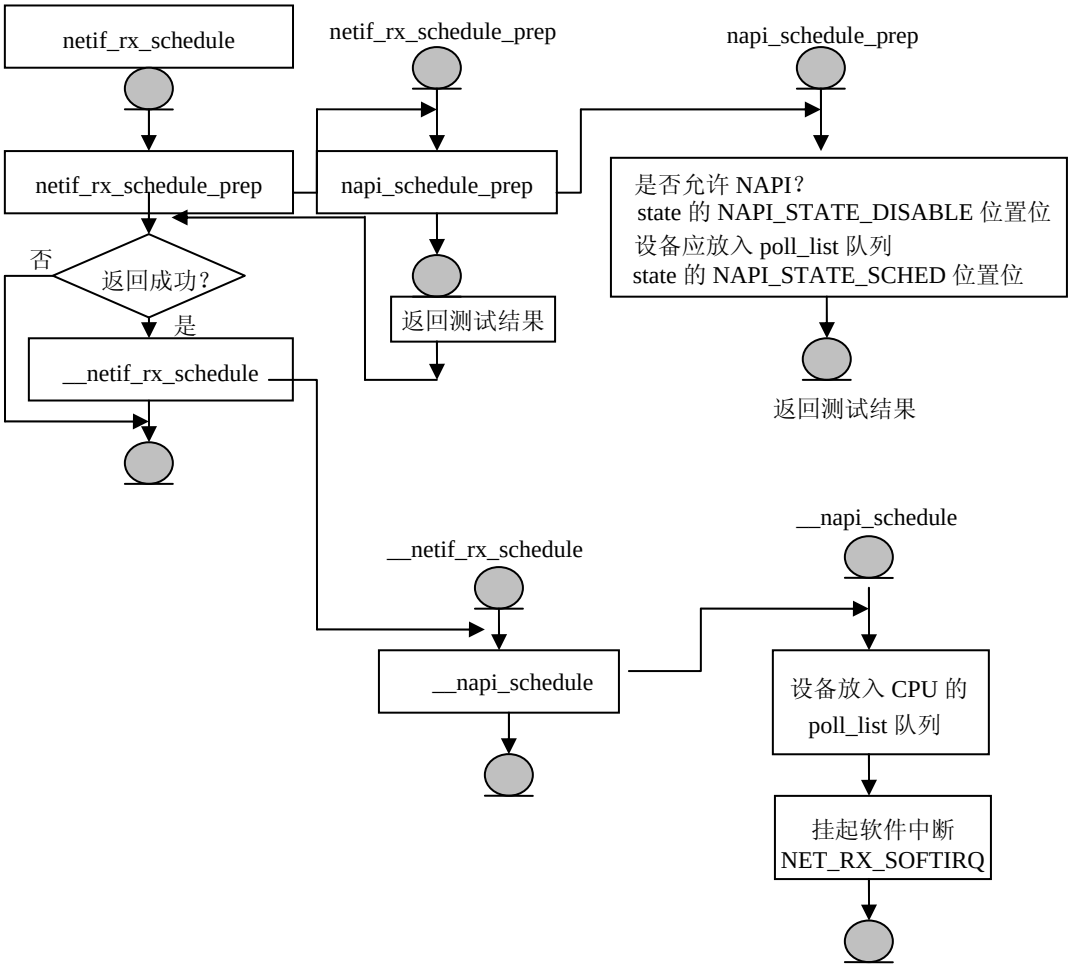


图 6-5　内核实现 NAPI 机制的关键函数

（1）netif_rx_schedule 函数

它是__netif_rx_schedule 函数的包装函数，在将设备放入 CPU 的 poll_list 队列之前，查看把设备放入 CPU 的 poll_list 队列的条件是否满足。这是通过调用 netif_rx_schedule_prep 函数，返回 napi_schedule_prep 对 naip_struct 的数据域 state 的 NAPI_STATE_DISABLE 位与 NAPI_STATE_SCHED 位来确定。NAPI_STATE_SCHED 位置 1，说明设备接收了新数据

帧，应放入 poll-list 队列，如 NAPI_STATE_SCHED 位清 0，设备应从 poll_list 队列中移出。

如果条件满足，就调用__netif_rx_schedule 函数。

（2）__netif_rx_schedule 函数

此函数调用__napi_schedule 函数将网络设备的 struct napi_struct 数据结构实例挂到 CPU 的 poll_list 队列上，标记网络数据接收软件中断 NET_RX_SOFTIRQ。

（3）__napi_schedule 函数

在本地 CPU 中断已关闭的情况下，将网络设备的 struct napi_struct 数据结构实例挂到 CPU 的 poll_list 队列尾，调用__raise_softirq_irqoff 升挂起网络接收软件中断 NET_RX_SOFTIRQ，然后开本地 CPU 中断。

6.2.2 netif_rx 函数分析

netif_rx 函数由常规网络设备驱动程序在接收中断中调用，它的任务就是把输入数据帧放入 CPU 的输入队列中，随后标记软件中断来处理后续上传数据帧给 TCP/IP 协议栈功能。netif_rx 函数会在以下几种场合调用：

- 网络设备驱动程序接收中断的执行现场。
- 处理 CPU 掉线事件的回调函数 dev_cpu_callback 中。
- loopback 设备的接收数据帧函数。

dev_cpu_callback 函数是网络子系统注册到 CPU 事件通知链中的回调函数，在对称多处理器系统 SMP 中，当一个 CPU 掉线时，会向其事件通知链发送事件消息，调用所有注册到 CPU 事件通知链中的回调函数来对事件做出反应。dev_cpu_callback 函数就将掉线 CPU 的 struct softnet_data 数据结构实例中的发送数据帧完成队列、输出设备队列、input_pkt_queue 加入到其他 CPU 的 struct softnet_data 数据结构实例相关队列中。将输入数据帧从掉线 CPU 的输入队列移到别的 CPU 输入队列，也是由 netif_rx 函数实现的。

常规情况下，netif_rx 函数在网络设备驱动程序的中断执行现场被调用（但也有例外，该函数被 loopback 设备调用时，不在中断执行现场，因为 loopback 设备不是真实的网络设备），所以开始执行 netif_rx 函数时要关本地 CPU 中断，等 netif_rx 执行完成后再开中断。

有一点需要记住的是：netif_rx 可以在不同 CPU 上同时运行，每个 CPU 有自己的 struct softnet_data 数据结构实例来维护状态，所以对 struct softnet_data 数据结构实例的访问不会出现问题。以下我们就对 netif_rx 函数的实现流程及源代码进行分析，首先我们给出 netif_rx 函数的实现流程图，再对照流程图分析函数源代码。

1．netif_rx 函数实现流程图

netif_rx 函数实现流程如图 6-6 所示。从图中描述可以看出 netif_rx 函数的主要任务为：

- 初始化 sk_buff 数据结构的某些数据域，如接收到数据帧的时间。
- 将接收到的数据帧放入 CPU 的私有输入队列，通过标记通知内核有数据帧到达。
- 更新 CPU 的接收统计信息。

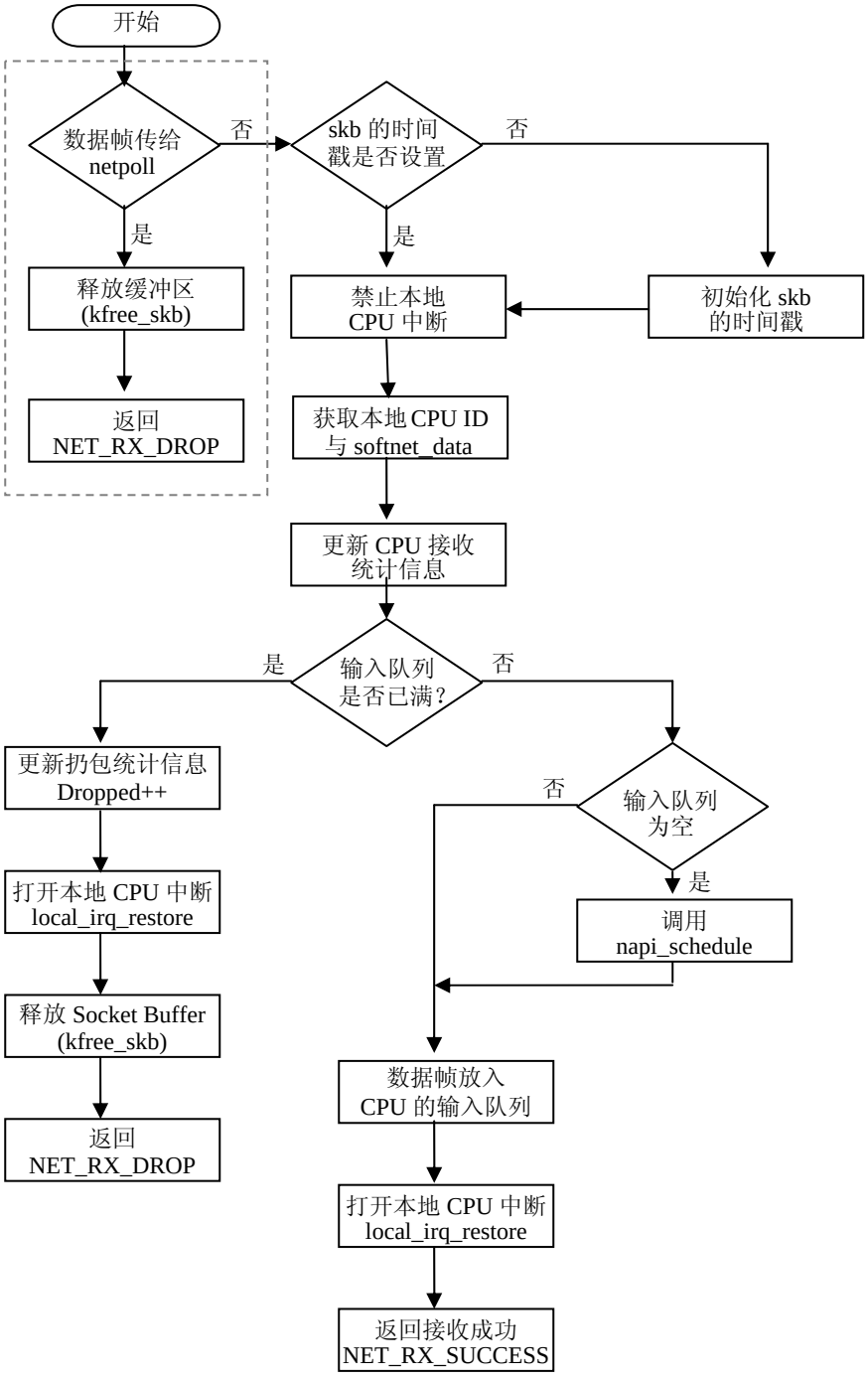


图 6-6　netif_rx 流程图

2．netif_rx 源代码分析

（1）输入参数

netif_rx 函数的唯一输入参数是：存放网络设备接收到的数据帧的 Socket Buffer。

（2）返回值

netif_rx 函数的返回值有两个：如果数据帧放入 CPU 的输入队列，返回 NET_RX_SUCCESS；如果数据帧扔掉则返回 NET_RX_DROP。

（3）源代码分析

netif_rx 函数首先查看 Socket Buffer 中的数据帧是否为 netpoll 需要的数据帧，如果是，则该数据帧不由上层协议处理，因此不放入 CPU 的输入队列，将其扔掉。netpoll 的功能是为了让内核在整个网络和 I/O 子系统都失效的情况下发送和接收数据帧，它用于远程网络控制终端和通过网络远程调试内核。

```
int netif_rx(struct sk_buff *skb)
{
  struct softnet_data *queue;
  unsigned long flags;
  //如果是 netpoll 需要的数据包，不做处理，扔掉数据包
  if (netpoll_rx(skb))
       return NET_RX_DROP;
```

① netif_rx 的初始化任务

将接收到数据帧的时间标记到 skb 的 tsstamp 数据域就意味着 netif_rx 开始执行了。为了将数据包放入 CPU 的输入队列，必须获取当前 CPU 标识 ID 与 softnet_data 数据结构实例。

```
  if (!skb->tstamp.tv64)
       net_timestamp(skb);
//关当前 CPU 本地中断，获取当前 CPU 的 softnet_data 数据结构实例，更新统计信息。该统计信息是总的统计信息，既包含
//接收到的数据帧数，也包含扔包数
  local_irq_save(flags);
  queue = &__get_cpu_var(softnet_data);

  __get_cpu_var(netdev_rx_stat).total++;
```

② 管理队列

CPU 的输入队列由 softnet_data->input_pkt_queue 管理，每个输入队列可接纳的最大数据帧数由全局变量 netdev_max_backlog 给出，默认值为 1000。即每个 CPU 的输入队列中可以存放 1000 个等待处理的网络数据帧（与系统中网络设备数量无关）。

以下代码的功能是：判断将数据帧放入队列，如果满足条件则将数据帧插入队列，最后判断在什么条件下调度处理数据帧的软件中断。

```
//当前队列中的数据帧数小于队列可缓存的最大数据帧数，队列未满
  if (queue->input_pkt_queue.qlen <= netdev_max_backlog) {
//队列中数据帧数不为 0，则队列不是空队列
       if (queue->input_pkt_queue.qlen) {
enqueue:
//队列未满且不为空，数据帧放入 CPU 输入队列后，开中断，返回接收成功
            __skb_queue_tail(&queue->input_pkt_queue, skb);
```

```
                local_irq_restore(flags);
                return NET_RX_SUCCESS;
            }
    //队列为空，调用 napi_schedule 标记网络接收软件中断 NET_RX_SOFTIRQ
            napi_schedule(&queue->backlog);
            goto enqueue;
      }
```

将网络数据帧放入 CPU 输入队列的速度非常快，这个过程没有任何复制，只是指针操作。

③ 将数据帧推送给上层协议

数据帧推送给上层协议，是在网络子系统的接收软件中断处理程序中完成的。软件中断是硬件中断处理程序的后半段（bottom half），在网络数据帧的接收处理过程中，硬件中断服务程序只完成最紧要的任务，即将数据从硬件缓冲区复制到内核地址空间，以防止数据丢失。而对数据帧的处理，如发送至上层协议实例，就由网络子系统的软件中断处理程序在适当的时候（CPU 不忙时）来完成。

网络子系统中，将 CPU 输入队列中的数据包上传给上层协议的处理函数是 process_backlog。process_backlog 函数的地址在网络子系统初始化时传给了管理 CPU 队列的数据结构 struct netsoft_data 的 backlog 数据域 poll 函数指针：

```
    queue->backlog.poll = process_backlog;
```

netif_rx 函数调用 napi_schedule 函数将 CPU 队列的 struct napi_struct 数据结构实例 backlog 挂到 CPU 的 poll_list 队列中，并标记网络子系统的接收软件中断 NET_RX_SOFTIRQ。

netif_rx 函数标记软件中断的条件是，CPU 的输入队列 input_pkt_queue 为空时调用 napi_schedule。如果程序运行至 netif_rx 函数且 CPU 的输入队列为空，说明当前是收到的第一个数据帧，应标记接收软件中断，并将数据帧放入 CPU 的输入队列（goto enqueue）。如果 CPU 的输入队列不为空，表明前面已标记过接收软件中断，只需将数据帧放入 CPU 的输入队列，返回成功。

napi_schedule 函数是__napi_schedule 函数的包装函数，从 6.2.1 节对 NAPI 工作机制的关键函数的分析我们知道，__napi_schedule 完成的功能就是将 struct napi_struct 数据结构实例放入 CPU 的 poll_list 队列，挂起网络接收软件中断 NET_RX_SOFTIRQ。这样推送数据帧给上层协议实例的处理函数，就会在内核调度的网络接收软件中断处理程序的 net_rx_action 函数中被执行。

④ netif_rx 结束处理

如果当前 CPU 的接收队列已满，netif_rx 将扔掉数据帧，释放缓冲区占用的内存空间，更新 CPU 扔包统计信息。

```
    __get_cpu_var(netdev_rx_stat).dropped++;
    local_irq_restore(flags);
    kfree_skb(skb);
    return NET_RX_DROP;
}
```

6.3 网络接收软件中断

网络接收软件中断（NET_RX_SOFTIRQ）的处理程序 net_rx_action 是接收网络数据中断的后半段。引起 net_rx_action 函数执行的是网络设备产生的接收数据硬件中断，它通知内核收到了网络数据帧，触发内核调度接收中断的后半段。net_rx_action 函数的任务就是将设备收到的数据帧上传给 TCP/IP 协议栈的上层协议处理。要推送给上层协议实例的数据帧可以缓存在以下两个地方等待 net_rx_action 函数来处理。

- 多个设备共享的 CPU 输入队列。不支持 NAPI 的网络设备的中断处理程序调用 netif_rx 函数，把数据帧放在当前执行中断处理程序的 CPU 的输入队列 softnet_data->input_pkt_queue 中。net_rx_action 从输入队列中获取数据帧上传给上层协议。CPU 的输入队列由所有不支持 NAPI 的网络设备共享。
- 设备硬件缓冲区。支持 NAPI 的网络设备驱动程序用 poll 函数直接从设备硬件缓冲区中读出数据帧推送给上层协议。

6.3.1 net_rx_action 的工作流程

图 6-7 给出了 net_rx_action 的实现流程图。从图中我们可以看到 net_rx_action 完成的任务非常简单：它展开 poll_list 队列链表，从队列中获取每一个 struct napi_struct 数据结构的实例，调用它们的 poll 函数指针指向的实例，直到遇到以下条件。

- poll_list 队列中已无 struct napi_struct 数据结构实例。
- 处理的数据帧数已达到 net_rx_action 可处理的上限值（budget）。budget 在 net_rx_action 函数的起始处初始化为 netdev_budget，而 netdev_budget 在 net/core/dev.c 文件中初始化为 300。
- net_rx_action 的执行时间已太长（2 个 tick，即 jiffers + 2），此时应释放 CPU。

CPU 的输入队列 input_pkt_queue 中最多可以缓存 1000 个数据帧，由 netdev_max_backlog 给出。net_rx_action 工作在中断允许的状态下，在 net_rx_action 运行时，新的数据帧可以不断加入到设备的缓冲区与 CPU 的输入队列中，因此等待处理的数据帧数可能会远远大于 net_rx_action 允许处理的上限值，net_rx_action 必须保证不能占用 CPU 太长时间，以致影响其他任务的执行。

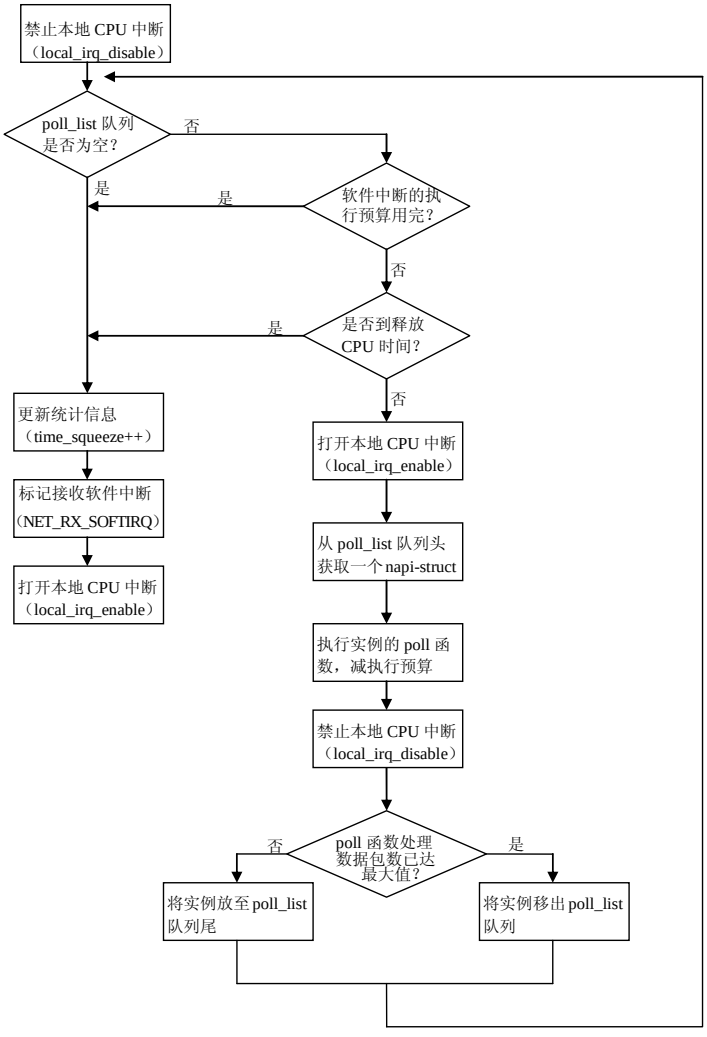


图 6-7　net_rx_action 的执行流程

6.3.2　net_rx_action 函数的实现细节

在理解 net_rx_action 的基本功能与实现流程后，我们现在可以深入到 net_rx_action 函数源代码的实现细节中来看看。首先初始化部分局部变量控制 net_rx_action 函数的执行，随后由主循环控制遍历 poll_list 中的每个 struct napi_struct 数据结构实例，并在允许的条件下执行每个 struct napi_struct 数据结构实例的 poll 函数。

1．函数初始化

函数的初始化阶段要完成的任务是：获取当前 CPU 的队列；设置 net_rx_action 函数能执行的最长时间为 2 个 tick（jiffies+2），net_rx_action 一次能处理的最大数据帧数为 netdev_budget。

```
static void net_rx_action(struct softirq_action *h)
{
  struct list_head *list = &__get_cpu_var(softnet_data).poll_list;
  unsigned long time_limit = jiffies + 2;
  int budget = netdev_budget;
  void *have;
```

2．函数主体

接下来进入函数主循环，遍历 poll_list 队列的每个 napi_struct 实例，执行循环体的条件为：

- poll_list 队列不为空。
- net_rx_action 处理的数据帧未达到上限值 300。
- net_rx_action 函数运行时间未达到最大值。

从 poll_list 获取一个 struct napi_struct 数据结构实例，执行其 poll 函数处理输入数据包。每个 struct napi_struct 数据结构实例的 poll 函数可以处理的最大数据帧数为 weight，该值初始化为 napi_struct->weight，而 napi_struct->weight 值初始，为全局变量 weight_p,其值为 64。即每个 poll 函数一次可以处理最大的数据帧数是 64。

```
  local_irq_disable ();
  while (!list_empty(list)) {
       struct napi_struct *n;
       int work, weight;
       //软件中断处理的数据帧数是否已达最大值（netdev_budget=300），或已运行了 2 个 tick（jiffies+2）
       if (unlikely(budget <= 0 || time_after(jiffies, time_limit)))
            goto softnet_break;
       local_irq_enable();
       //这时即使中断重新打开，这个访问仍然安全，因为中断只能向链表尾加入新的数据帧，只有 poll 函数可以从队列
       //头移出数据帧，获取一个 struct napi_struct 数据结构实例
       n = list_entry(list->next, struct napi_struct, poll_list);
       have = netpoll_poll_lock(n);
       //初始化每个 napi_struct 实例的 poll 函数一次可以处理的最大数据帧数，执行 poll 函数
       weight = n->weight;
       work = 0;
       if (test_bit(NAPI_STATE_SCHED, &n->state))
            work = n->poll(n, weight);
       WARN_ON_ONCE(work > weight);
      //将软件中断处理数据帧数的预算减去 poll 函数处理的数据帧数
       budget -= work;
       local_irq_disable();
      //如果 poll 函数处理的数据帧数已达上限值，清除 NAPI_STATE_SCHED，将该实例移出 poll_list 队列，否则将其
      //移到 poll_list 队尾
       if (unlikely(work == weight)) {
            if (unlikely(napi_disable_pending(n)))
                 __napi_complete(n);
            else
                 list_move_tail(&n->poll_list, list);
       }
       netpoll_poll_unlock(have);
  }
out:
  local_irq_enable();
```

3．函数结束处理

函数的最后片段是在 CPU 输入队列中仍有数据帧，但 net_rx_action 函数被迫返回，这

时需要重新标记网络子系统的接收软件中断 NET_RX_SOFTIRQ，以便下次内核调度软件中断时执行 net_rx_action 函数，处理余下的数据帧。

```
softnet_break:
  __get_cpu_var(netdev_rx_stat).time_squeeze++;
  __raise_softirq_irqoff(NET_RX_SOFTIRQ);
  goto out;
}
```

6.3.3　从输入队列中读取数据帧

CPU 队列 softnet_data 的 poll 虚函数，初始化为默认的 process_backlog，为不支持 NAPI 的设备来处理队列中的数据帧。分析 netif_rx 的执行流程，我们知道 netif_rx 函数在接收到第一个网络数据帧时，会将处理 CPU 输入队列数据帧上传的函数 process_backlog 放到当前 CPU 的 poll_list 队列，在 net_rx_action 函数中调用执行。

因为 process_backlog 要处理网络设备共享的输入队列中的数据帧，所以需要考虑一个重要因素。当 process_backlog 函数运行时，网络设备的硬件中断是打开的，所以 process_backlog 函数的执行可能被硬件中断打断，基于这个原因，访问 struct softnet_data 数据结构时总是禁止本地 CPU 中断（由 local_irq_disable 完成，特别是调用__skb_dequeue 时），这个锁定机制对 NAPI 设备而言是不需要的，因为它们的 poll 函数在执行时，设备硬件中断是关闭的，况且每个设备有自己的输入队列。下面深入分析函数 process_backlog 的执行流程。

1．函数初始化

函数已开始初始化部分局部变量，以控制 process_backlog 函数的执行。

- 获取当前 CPU 的输入队列
- process_backlog 函数一次能处理的最大数据帧数 weitht_p。
- process_backlog 函数的执行起始时间，以免 process_backlog 执行太长时间。

```
static int process_backlog(struct napi_struct *napi, int quota)
{
  int work = 0;
  struct softnet_data *queue = &__get_cpu_var(softnet_data);
  unsigned long start_time = jiffies;
  napi->weight = weight_p;
```

2．函数主体

接着进入主循环，在主循环中 process_backlog 函数要完成的任务就是从 CPU 输入队列中读取所有的存放网络数据帧的 Socket Buffer。当遇到以下条件时，循环结束：

- CPU 输入队列已空。
- 所读取的数据帧数已达到 process_backlog 函数能处理的上限值 weight_p。
- process_backlog 函数的运行时间太长（超过 1 个 tick）。

```
    do {
         struct sk_buff *skb;
        //从 CPU 的输入队列读取 skb，如队列已空，将队列的 struct napi_struct 数据结构实例从 CPU 的 poll_list 队列中移出
         local_irq_disable();
```

```
            skb = __skb_dequeue(&queue->input_pkt_queue);
            if (!skb) {
                  __napi_complete(napi);
                  local_irq_enable();
                  break;
            }
            local_irq_enable();
            //将 skb 推送给上层协议实例的数据接收处理函数
            netif_receive_skb(skb);
      } while (++work < quota && jiffies == start_time);

      return work;
}
```

上面的两个循环结束条件与 net_rx_action 的执行限制类似，这是因为 process_backlog 函数是在 net_rx_action 循环中调用的，它们执行的限制条件也应相互配合。process_backlog 函数的执行流程如图 6-8 所示。

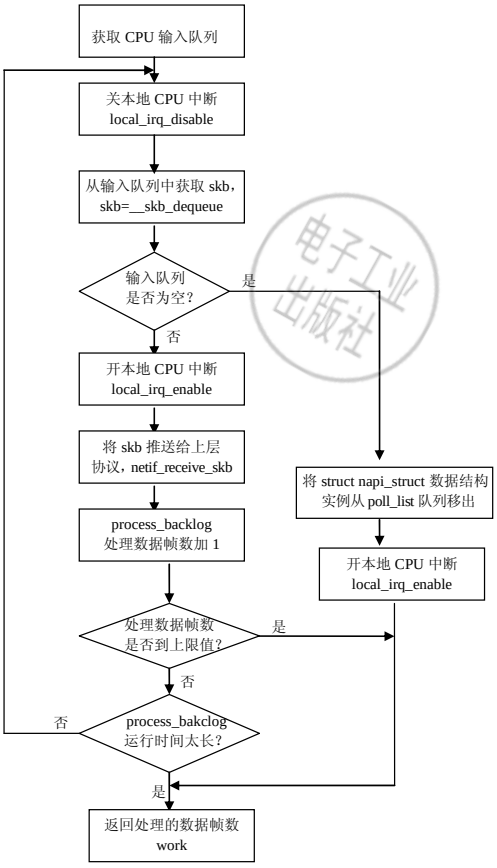


图 6-8　process_backlog 函数的实现流程

6.3.4　处理输入数据帧

netif_receive_skb 函数是 napi_struct 实例的 poll 函数使用的，帮助函数处理输入数据帧。在数据链路层（L2）和网络层（L3）可以实现多个协议实例，每个网络设备驱动程序

都与某特定硬件协议（如以太网、令牌环网等）相关，网络设备驱动程序很容易解析数据链路层协议头信息，并提取网络层使用的协议。在 net_rx_action 执行时，网络层协议标识已经从数据链路层协议头中提取出来，由驱动程序存放在 skb->protocol 中。

1．netif_receive_skb 的实现流程

netif_receive_skb 函数的任务主要有 3 个：

- 给每个协议标签发送一个数据帧的复制。
- 给 skb->protocol 中指定的网络层协议处理程序发送一个数据帧拷贝。
- 执行需要在该层处理的功能特点，比如桥。

如果上层没有 skb->protocol 中指定的协议处理程序，也没有别的功能特点要在 netif_receive_skb 函数中处理该数据帧，内核就不知如何处理数据帧，因此会扔掉数据帧。netif_receive_skb 函数的实现流程如图 6-9 所示。

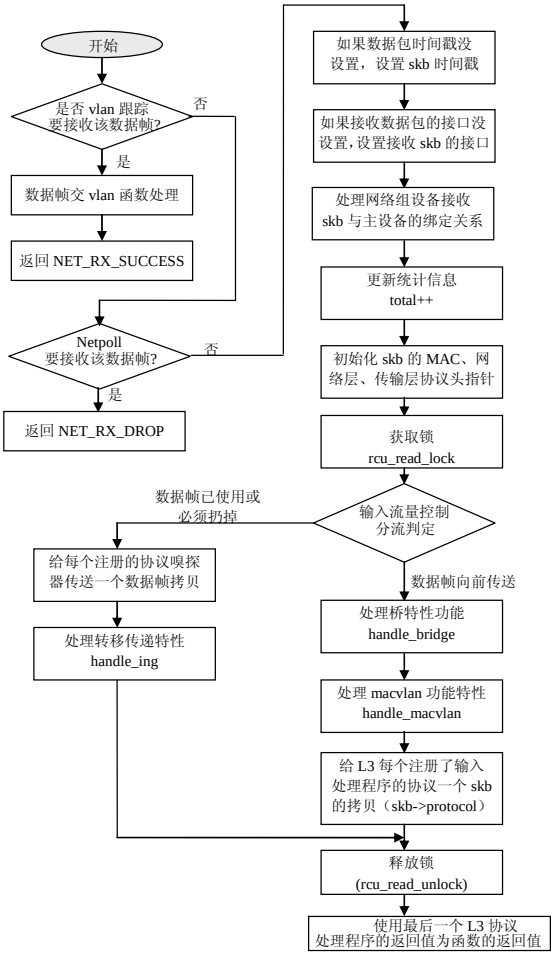


图 6-9　netif_receive_skb 函数的实现流程

2．netif_receive_skb 源码分析

在将输入数据帧传给上层协议处理程序之前，netif_receive_skb 函数还必须处理一些功

能特性，这些功能特性会对数据帧做一些小的改变。

```
int netif_receive_skb(struct sk_buff *skb)
{
  struct packet_type *ptype, *pt_prev;
  struct net_device *orig_dev;
  struct net_device *null_or_orig;
  int ret = NET_RX_DROP;
  __be16 type;
  //vlan 跟踪要接收该数据帧，由 vlan 功能代码接收处理数据帧，返回 NET_RX_SUCCESS
  if (skb->vlan_tci && vlan_hwaccel_do_receive(skb))
       return NET_RX_SUCCESS;
  //如果 netpoll 要接收该数据帧，将 skb 传给 netpoll 处理函数，返回 NET_RX_DROP
  if (netpoll_receive_skb(skb))
       return NET_RX_DROP;
  //设置数据帧的时间戳，接收数据帧网络接口，等信息到 skb 相应数据域
  if (!skb->tstamp.tv64)
       net_timestamp(skb);
  if (!skb->iif)
       skb->iif = skb->dev->ifindex;
  //处理数据帧与组网络设备绑定功能特性
  null_or_orig = NULL;
  orig_dev = skb->dev;
  if (orig_dev->master) {
       if (skb_bond_should_drop(skb))
            null_or_orig = orig_dev;
       else
            skb->dev = orig_dev->master;
  }
```

所谓绑定就是将多个网络接口组成一个设备组，当成一个网络接口来处理。如果数据帧是从网络设备组中的一个接口收到的，在将数据帧上传给网络层处理程序之前，sk_buff 数据结构中对接收数据帧设备的引用计数必须改为对设备组主设备的引用，这是处理绑定的目的。

一旦所有的协议嗅探器（sniffer）都接收到它们的数据帧拷贝，在实际的协议处理程序得到它们的拷贝之前，netif_receive_skb 函数必须处理输入流量控制、桥等功能特点。

如果桥代码与输入流量控制代码都不处理输入数据帧，数据帧就传递给 L3 层的协议处理程序（通常只有一个协议处理程序接收数据帧，但也可以注册多个协议处理程序接收同一个数据帧）。在 Linux 内核的早期版本中，将数据帧发送给上层协议处理程序是 netif_receive_skb 要完成的唯一功能。现在在内核网络协议栈中加入的网络协议越来越多，加入的协议越多，netif_receive_skb 函数要处理的功能特性就越多，数据帧经过网络协议栈的路径就越复杂。

```
  //更新统计信息，初始化 sk_buff 数据结构的 MAC 层、网络层和传输层的协议头指针
  __get_cpu_var(netdev_rx_stat).total++;
  skb_reset_network_header(skb);
  skb_reset_transport_header(skb);
  skb->mac_len = skb->network_header - skb->mac_header;
  pt_prev = NULL;
  //从以下开始处理数据帧继续传递的路径，在此之前先获取锁，根据输入流量控制配置的对数据帧的过滤特性，决定如何传递数据帧
  rcu_read_lock();
```

3．特殊功能特性的处理

以往流量控制总是用于实现数据帧输出路径的质量服务（QoS）功能，在现在的内核

版本中，用户也可能为输入流量网络数据过滤器配置。基于这样的配置，handle_ing 中的 ing_filter 可以确定输入数据帧是应该扔掉还继续做进一步处理。

如配置了桥功能特性，netif_receive_skb 函数在网络接收软件中断处理程序 net_rx_action 中执行，net_rx_action 函数是网络驱动程序与 L3 协议处理程序之间的执行边界，所以在进入下一阶段之前，在 netif_receive_skb 函数中必须处理一些特殊功能。例如内核配置支持桥功能，handle_bridge 被初始化为一个功能函数来查看桥代码是否应处理输入数据帧。在桥代码要处理输入数据帧的情况下，数据帧会传给桥功能，handle_bridge 函数返回 1。相反 handle_bridge 返回 0，netif_receive_skb 将继续处理 skb。

```
//给所有的协议嗅探器发送一个数据帧的拷贝
  list_for_each_entry_rcu(ptype, &ptype_all, list) {
        if (ptype->dev == null_or_orig || ptype->dev == skb->dev ||
           ptype->dev == orig_dev) {
             if (pt_prev)
                   ret = deliver_skb(skb, pt_prev, orig_dev);
             pt_prev = ptype;
        }
  }
//如果配置了输入数据帧过滤特性，由 ing_filter 决定如何对数据帧的进一步处理
#ifdef CONFIG_NET_CLS_ACT
  skb = handle_ing(skb, &pt_prev, &ret, orig_dev);
  if (!skb)
        goto out;
ncls:
#endif
//处理桥功能特性与 macvlan 特性
  skb = handle_bridge(skb, &pt_prev, &ret, orig_dev);
  if (!skb)
        goto out;
  skb = handle_macvlan(skb, &pt_prev, &ret, orig_dev);
  if (!skb)
        goto out;
```

4．将数据帧上传给上层协议处理程序

最后 netif_receive_skb 将输入数据帧发送给网络层所有注册了接收处理程序的协议实例。

```
type = skb->protocol;
  list_for_each_entry_rcu(ptype,
             &ptype_base[ntohs(type) & PTYPE_HASH_MASK], list) {
        if (ptype->type == type &&
           (ptype->dev == null_or_orig || ptype->dev == skb->dev ||
            ptype->dev == orig_dev)) {
             if (pt_prev)
                   ret = deliver_skb(skb, pt_prev, orig_dev);
             pt_prev = ptype;
        }
  }
```

6.4 数据链路层与网络层的接口

在本章前面的讨论中，结合网络设备驱动程序与 net/core/dev.c 文件中的 netif_rx/netif_rx_schedule 、网络接收软件中断处理程序 net_rx_action、TCP/IP 协议栈的数据链路层，完成了对网络输入数据帧的处理。输入数据帧通过数据链路层应进一步向网络层推送。在 TCP/IP 协议栈各层都有多个协议实例，Linux 内核的网络体系结构允许同一个输入数据帧在 TCP/IP 协议栈各层由某个协议实例接收，也允许其同时由多个协议实例接收。例如在网络层就有 IPv4、IPv6、ARP 等协议实例。另外，Linux 内核实现的网络体系结构还可以同时支持多个协议栈。

本节我们就将描述数据链路层与其直接相关的上层网络层之间发送/接收数据帧的接口。在 Linux 内核中实现的数据链路层与网络层之间的接口，极具扩展性与兼容性，它可以非常方便地在协议栈中加入新的协议实例，同时极好地隐藏了协议栈层与层之间的实现细节。

我们首先会给出数据链路层与网络层之间接口的数据结构，随后会解释如何在接口中加入新的协议实例来接收数据帧。注意，上层加入新的协议实例完全不会影响其他协议层实例接收功能的实现。

6.4.1 输入数据帧协议解析

网络数据由网络设备驱动程序接收后，从网络设备硬件缓冲区穿过。

TCP/IP 协议栈最后到达应用程序，每一层的处理都需要确定应如何将数据帧发送给正确的协议实例。下面我们主要解释数据链路层如何识别网络层的协议，在 Linux 内核 TCP/IP 协议栈中，相同的方法可应用于协议栈各层。

当网络设备驱动程序接收到一个数据帧后，驱动程序将数据帧存放在 sk_buff 数据结构中，并初始化 sk_buff 的 protocol。这个数据域的值可以是定义在内核中某个协议的标识符，也可以是输入数据帧 MAC 协议头中的一个数据域，netif_receive_skb 函数参考 sk_buff->protocol 数据域的值来确定应将输入数据帧上传给网络层的哪个协议的接收处理程序，完成数据帧在 L3 的处理。各层协议标识解析过程如图 6-10 所示。

由图 6-10 可以看到，网络设备的硬件类型确定了数据链路层的协议，而且各类型的网络设备调用自己的驱动程序来接收数据帧。数据链路层的处理程序解析数据链路层的协议头（MAC 头）信息，据此判断该数据帧（去掉 MAC 协议头后）应传给网络层的哪个协议处理程序。

网络层接收到数据帧后，从网络层的协议头中解析该数据帧的传输层使用的协议，确定应将数据帧传给传输层的哪个协议处理程序接收。

传输层从自身的协议头信息中提取目标端口号，以确定接收数据帧的应用程序。

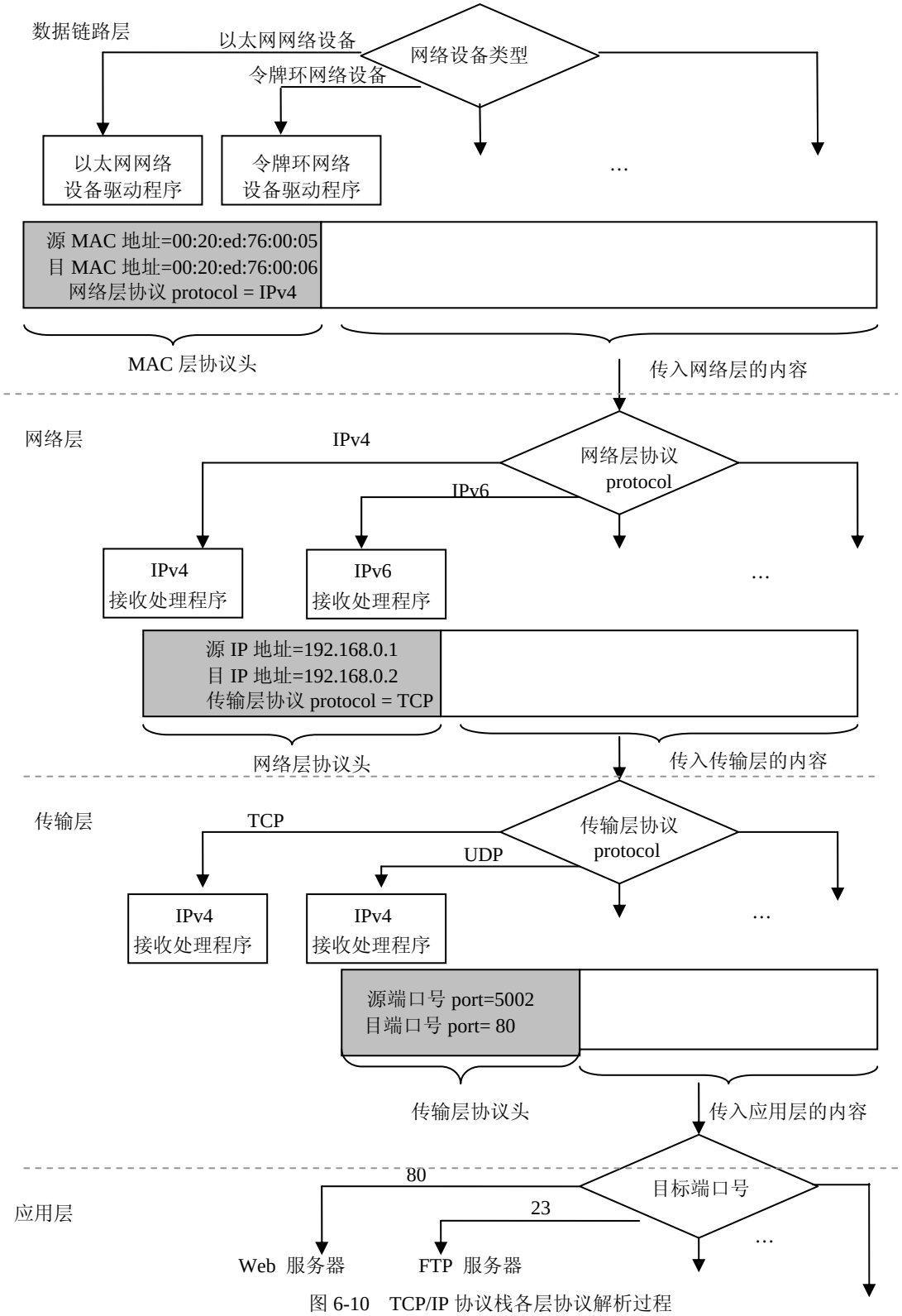


图 6-10　TCP/IP 协议栈各层协议解析过程

Linux 内核支持的协议标识符定义在 include/linux/if_ether.h 文件中，标识符的命名格式为 ETH_P_XXX，这些值就是 skb->protocol 引用的值。表 6-1 列出了 Linux 内核内部使用的值，每个协议都实现了相应的数据帧接收处理函数。数据链路层处理函数 netif_receive_skb 以网络驱动程序分配给 skb->protocol 的协议标识符为索引，将数据帧传给正确的网络层协议接收处理程序。

图 6-11 给出了 netif_receiv_skb 以 skb->protocol 为索引将数据帧上传给网络层处理程序的过程。

表 6-1　Linux 内核支持的协议标识符定义

协议标识符	值	处 理 函 数
ETH_P_IP	0x0800	ip_rcv
ETH_P_ARP	0x0806	arp_rcv
…...		
ETH_P_IPV6	0x86DD	ipv6_rcv
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，用于处理如网络嗅探器接收所有的网络数据帧
ETH_P_802_2	0x0004	llc_rcv
ETH_P_TR_802_2	0x0011	llc_rcv
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

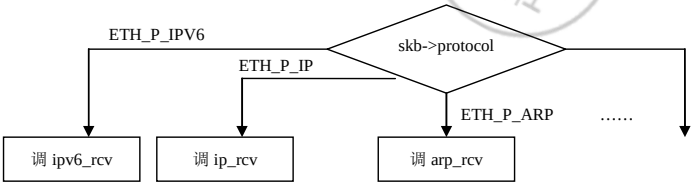


图 6-11　netif_receiv_skb 按协议标识上传数据帧

6.4.2　实现数据链路层与网络层接口的关键数据结构

每个网络协议，无论它在 TCP/IP 协议栈的哪一层，都会初始化一个接收数据帧的函数。实现协议栈层与层之间数据发送的关键是,无论各层有多少个协议实例（如在网络层的协议实例有 Ipv4、Ipv6 和 ARP 等），数据帧传给一个协议处理程序还是多个协议处理程序，上层或下层的实现都不会影响其他层的协议实例实现。为了实现这样的体系结构，Linux 内核的网络子系统定义了两个数据结构来管理协议标识符与协议接收处理程序之间的关系，并将数据结构实例按一定的方式组织，以便 L2 层接收函数 netif_receive_skb 能快速有效地将数据帧上传给 L3 层。

本节讲述实现接口的关键数据结构与数据结构的组织，下一节将讲述如何在接口中加入/移出协议实例处理程序。

1．实现接口的数据结构 struct packet_type

struct packet_type 数据结构描述了网络层协议标识符、接收处理程序与接收网络设备等的相互关联，每个在网络层的协议实例都有一个 struct packet_type 类型的变量，来定义协议处理接收数据帧的实体。struct packet_type 数据结构定义在 include/linux/netdevice.h 文件中。

```
struct packet_type {                          //include/linux/netdevice.h
  __be16                type; /* This is really htons(ether_type). */
  struct net_device     *dev; /* NULL is wildcarded here   */
  int              (*func) (struct sk_buff *,
                            struct net_device *,
                            struct packet_type *,
                            struct net_device *);
  …
  void             *af_packet_priv;
  struct list_head list;
};
```

struct packet-type 数据结构中各数据域的含义如下。

- type：协议标识符，其值为表 6-1 所列的 ETH_P_XXX 之一。协议标识符须由网络设备驱动程序接收例程从数据帧中解析出来。
- dev：指向接收数据帧的网络设备的 struct net_device 数据结构实例的指针。如该指针不为空，表明只有从指定网络设备接收到的数据帧才会传给协议处理程序。如 dev 指针为空，则为常规使用方式，在选择协议处理程序时，网络设备不起作用。
- func：指向协议实例实现的接收处理程序的函数指针，当 skb->protocol=type 时由 netif_receive_skb 调用该处理程序。在协议初始化时，协议实例将自己的处理函数注册到 struct packet_type 数据结构中。例如 IP 协议注册到 struct packet_type 数据结构的处理程序为 ip_rcv。
- af_packet_priv：指向协议使用的私有数据的指针，通常没有使用。
- list：用于链接几个 struct packet_type 数据结构的链表。

struct packet_type 数据结构的其他数据域是处理数据帧分片的变量与处理函数。

2．struct packet_type 数据结构实例化

要实现 L2 与 L3 之间的接收接口，TCP/IP 协议栈网络层（L3）中的每个协议首先需要定义一个 struct packet_type 数据结构的变量实例，并初始化 packet_type 数据结构的各数据域。例如 IPv4 协议实例在 net/ipv4/af_inet.c 文件中定义了 IPv4 协议的 struct packet_type 数据结构的变量实例。

```
static struct packet_type ip_packet_type = {          //net/ipv4/af_inet.c
  .type = __constant_htons(ETH_P_IP),
  .func = ip_rcv,
  .gso_send_check = inet_gso_send_check,
  .gso_segment = inet_gso_segment,
  .gro_receive = inet_gro_receive,
  .gro_complete = inet_gro_complete,
};
```

类似的 IPv6 协议与 ARP 协议也分别定义了自己的 struct packet_type 数据结构的变量

实例，IPv6 协议的 struct packet_type 数据结构的变量实例定义在 net/ipv6/af_inet6.c 文件中，ARP 协议定义在 net/ipv4/arp.c 文件中。

6.4.3 接口的组织

在 Linux 内核中定义了两个哈希链表，哈希链表中的每个元素都是一个 struct list_head 变量，指向网络层协议的 struct packet_type 数据结构类型变量实例，这两个向量表形成了数据链路层与网络层之间接收数据帧的接口。在 Linux 内核中，我们要区分两种网络层协议类型：

1．接口向量表

以上两种接口分别为 struct list_head ptype_all，其中的接收处理程序接收所有数据帧，该接口主要用于网络工具和网络探测器接收数据帧；struct list_head ptype_base[PTYPE_HASH_SIZE]，其实例只接收与协议标识符相匹配的网络数据帧。PTYPE_HASH_SIZE 的值定义为 16；接口对 struct packet_type 变量实例的组织如图 6-12 所示。

在 struct ptype_base 数据结构哈希链表中，链接在同一个 struct list_head 链表中的多个 struct packet_type 数据结构变量实例表明某个数据帧可以由多个协议实例接收处理。

netif_receive_skb 函数向网络层发送数据帧时，查询哈希链表的函数遍历 ptype_base 链表，找到其 type 数据域与 skb->protocol 相匹配的 packet_type 实例成员，将数据帧发送给其处理函数 func。

- 将所有网络设备接口收到的数据包发送给网络层协议。
- 只将与协议标识符相匹配的数据包发送给网络层协议。

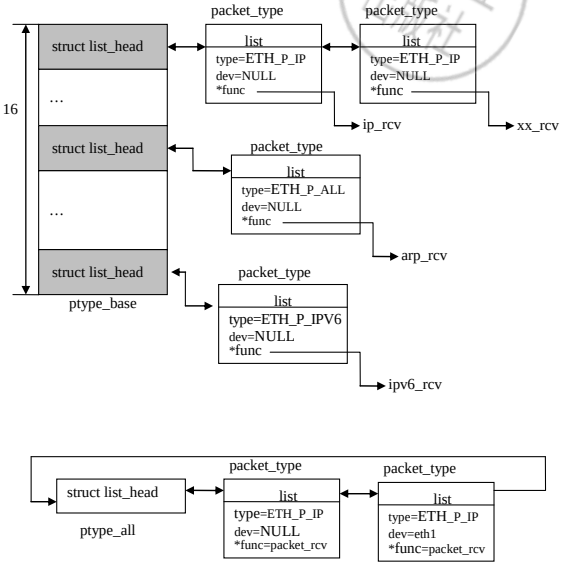


图 6-12　packet_type 实例的组织

2．向接口中增加/删除协议

Linux 内核实现了两个 API：dev_add_pack 和 dev_remove_pack，分别将 packet_type 实例变量加入/移出 ptype_base/ptype_all 哈希链表。

dev_add_pack 函数非常简单。它查看处理程序是否为协议探测器（packet_type->type == htons(ETH_P_ALL)），如果是，dev_add_pack 函数就将 packet_type 实例加入到 ptype_all 链表中，并增加 ptype_all 链表的成员计数（netdev_nit++）。如果处理程序不是协议探测器，dev_add_pack 函数在 ptype_base 表中找到一个空插槽，产生一个哈希值，根据获取的哈希值将 packet_type 实例插入到 ptype_base 哈希链表中。

```
void dev_add_pack(struct packet_type *pt)                    //net/core/dev.c
{
  int hash;

  spin_lock_bh(&ptype_lock);
  if (pt->type == htons(ETH_P_ALL))
       list_add_rcu(&pt->list, &ptype_all);
  else {
       hash = ntohs(pt->type) & PTYPE_HASH_MASK;
       list_add_rcu(&pt->list, &ptype_base[hash]);
  }
  spin_unlock_bh(&ptype_lock);
}
```

__dev_remove_pack 函数将 packet_type 实例从 ptype_all 或 ptype_base 向量表中移出。

```
void __dev_remove_pack(struct packet_type *pt)
{
  struct list_head *head;
  struct packet_type *pt1;

  spin_lock_bh(&ptype_lock);

  if (pt->type == htons(ETH_P_ALL))
       head = &ptype_all;
  else
       head = &ptype_base[ntohs(pt->type) & PTYPE_HASH_MASK];

  list_for_each_entry(pt1, head, list) {
       if (pt == pt1) {
            list_del_rcu(&pt->list);
            goto out;
       }
  }

  printk(KERN_WARNING "dev_remove_pack: %p not found.\n", pt);
out:
  spin_unlock_bh(&ptype_lock);
}
```

3．何时注册 packet_type 实例

为了向 ptype_base 中注册协议的 packet_type 实例，协议需要首先定义自己的 packet_type 变量，并且为数据域赋值（见 6.4.2 节），随后在协议的初始化函数中调用 dev_add_pack 函数将 packet_type 实例插入 ptype_base 哈希链表中。

例如在内核启动时间初始化 IPv4 协议时，会执行 IPv4 的初始化函数 inet_init，而 inet_init 的任务之一就是调用 dev_add_pack 函数将 IPv4 的 packet_type 实例 ip_packet_type 插入 ptype_base 哈希链表中。

```
static struct packet_type ip_packet_type = {
  type = __constant_htons(ETH_P_IP),
  .func = ip_rcv,
  …
};

static int __init inet_init(void)
{
  …
  dev_add_pack(&ip_packet_type);
  …
}
```

6.5 数据链路层对数据帧发送的处理

在本章前面我们已讨论了大量关于接收数据帧，将数据帧上传至 TCP/IP 协议栈上层的主题，接收数据帧是指处理从网络介质进入系统的数据；而本节讨论的数据帧发送与数据帧接收的过程正好相反，是指将上层协议投递给数据链路层的数据传送至网络介质上，我们将讨论在数据帧向外发送的路径上数据链路层的主要处理任务，涵盖的主题有：

- 允许/禁止设备发送数据帧。
- 调度一个网络设备来发送数据帧。
- 如何从网络设备输出队列等待发送的数据帧中选择下一个发送的数据帧。
- 处理发送过程的主要函数实现。

在数据帧的处理过程中，发送和接收的许多步骤都有对称性。在本节要介绍的大部分数据结构、处理函数与前面讨论的接收过程的数据结构和处理函数成对出现，只是其处理过程相反。例如接收软件中断是 NET_RX_SOFTIRQ，相对应的发送数据有软件中断 NET_TX_SOFTIRQ；它们的软件中断处理程序分别是 net_rx_action 和 net_tx_action。所以如果读者很好地理解了在前面章节中介绍的数据帧的接收实现，再掌握发送过程就容易多了。它们的相似之处还表现在：

- 在网络数据帧的接收过程中，每个 CPU 队列数据结构 softnet_data 中的 poll_list 数据域是网络设备的队列，在 poll_list 队列中的网络设备缓冲区中有接收到的数据帧；相应的，在 softnet_data 数据结构中的 output_queue 数据域也是网络设备实例的队列，设备中有数据需要发送。
- 只有激活了网络设备（设置 dev->state 的__LINK_STATE_START 标志位），网络设备才能被调度来接收网络数据。相应的，只有网络设备的发送队列启动（清除 dev->_tx->state 的__QUEUE_STATE_XOFF 位），设备才可以调度发送数据。
- 当设备被调度来接收数据，放入 CPU 的 poll_list 列表时，NAPI_STATE_SCHED 标志位应被设置，相应的，当一个设备被调度来发送数据时，其发送队列的调度策略状态位_QDISC_STATE_SCHED 标志应被设置。

在数据接收过程中，netif_rx 函数是发送设备数据缓冲区的数据到内核接收队列的接口，在数据发送过程中，dev_queue_xmit 函数在数据帧的输出路径上起类似作用，dev_queue_xmit 函数将内核地址空间的队列数据传给驱动程序的缓冲区。本节我们将介绍数据链路层发送数据帧过程在内核的实现。

6.5.1　启动/停止设备发送数据

网络设备驱动程序完成了网络设备实例的创建、初始化、注册并打开激活的设备后，网络设备就可以正常运行，开始接收与发送数据帧了，在设备运行过程中为什么还需要允许/禁止设备发送数据帧呢？其中一个原因是网络设备的发送缓冲区满，不能再缓存内核向外发送的数据帧，如果不通知内核，会引起发送过程失败。允许/禁止内核继续向网络设备发送数据帧，通过启动/停止网络设备的发送队列来实现。

网络设备的发送队列状态由 net_device->state 状态标志的__QUEUE_STATE_XOFF 位来描述，内核实现了一系列 API 来操作、检查网络设备的发送队列状态，这些 API 以及描述网络设备输出队列状态的数据结构都定义在 include/linux/netdevice.h 文件中。

1．netif_start_queue

启动网络设备的发送队列，允许网络设备发送。该函数通常在激活网络设备时（见第 5 章网络设备驱动程序 open 方法的实现）被调用，也会在重启一个已被停止的网络设备时被调用。

2．netif_stop_queue

停止网络设备的发送队列，也就禁止了网络设备的发送过程。网络设备发送队列停止后，任何在该网络设备上的发送尝试都会失败。

3．netif_queue_stopped

返回网络设备输出队列当前的状态：允许或禁止。该函数返回 net_device->state 标志中_QUEUE_STATE_XOFF 位当前的状态。

4．netif_wake_queue

唤醒网络设备发送队列，重新启动网络设备的发送过程。

网络设备驱动程序的 open 方法除了为设备分配需要的资源、初始化 net_device 数据结构的数据域与函数指针外，其中一个重要的任务就是激活网络设备，允许网络设备的接收/发送过程。在 open 方法的最后就调用了 netif_start_queue 函数来启动网络设备发送队列。当设备驱动程序意识到设备硬件没有足够的空间缓存新的数据帧时，它就用 netif_stop_queue 函数来停止网络设备的发送队列，这样可以避免在将来的发送过程中浪费资源，因为内核已知发送会失败，就不会尝试发送新数据帧给网络设备。

例如我们在第 5 章中分析的 CS8900A 网络设备驱动程序的发送函数中。（hard_start_xmit 方法的实现函数 net_send_packet）。在开始发送之前，函数首先停止网络设备的发送队列，因为在本次发送结束之前，网络设备硬件没有足够的空间缓存新的发送数据帧，因此停止设备的发送队列，通知内核不要再向设备发送新的数据帧。

```
static int net_send_packet(struct sk_buff *skb, struct net_device *dev)
{
  spin_lock_irq(&lp->lock);
  netif_stop_queue(dev);
  …
```

那么网络设备的发送队列在什么时候重新启动呢？在 CS8900A 的驱动程序中这个过程由中断处理程序来处理，如果网络设备产生的中断是上次数据发送完成中断，则驱动程

序重启网络设备的发送队列。

```
static irqreturn_t net_interrupt(int irq, void *dev_id)
{

  while ((status = readword(dev->base_addr, ISQ_PORT))) {
        switch(status & ISQ_EVENT_MASK) {
        case ISQ_RECEIVER_EVENT:
              net_rx(dev);
              break;
        case ISQ_TRANSMITTER_EVENT:
              lp->stats.tx_packets++;
              netif_wake_queue(dev);
```

中断处理程序调用了 netif_wake_queue 函数来重启网络设备的发送队列，这个函数不仅重新允许网络发送，同时也要求内核查看在网络设备的发送队列中是否有数据帧在等待发送。

6.5.2 调度设备发送数据帧

在描述数据帧接收处理过程时，我们看到当网络设备收到一个数据帧，网络设备驱动程序就调用内核的 API，netif_rx/netif_rx_schedule，来将数据帧放入 CPU 的输入队列，或者将设备放入 CPU 的 poll_list 队列，随后调度接收软件中断 NET_RX_SOFTIRQ。

数据帧的发送也有类似的过程。内核实现了 dev_queue_xmit 函数，该函数将上层协议发送来的数据帧放到网络设备的发送队列（针对有发送队列的网络设备），随后流量控制系统按照内核配置的队列管理策略，将网络设备发送队列中的数据帧依次发送出去。发送时，从网络设备的输出队列上获取一个数据帧，将数据帧发送给设备驱动程序的 dev->netdev_ops->ndo_start_xmit 方法。

dev_queue_xmit 函数可能因为各种原因执行失败：①网络设备的发送队列被停止（如前描述）；②获取保护输出队列并发访问的锁失败。

在第②种 dev_queue_xmit 函数执行失败的情况下，内核实现了函数_netif_schedule 来重新调度网络设备发送数据帧，它将网络设备放到 CPU 的发送队列 softnet_data->output_queue 上，随后标识网络发送软件中断 NET_TX_SOFTIRQ，当发送软件中断被内核调度执行时，CPU 输出队列 output_queue 中的设备会重新被调度来发送数据帧。（其作用就类似 netif_rx_schedule 函数处理输入路径的功能）。__netif_schedule 函数定义在 net/core/dev.c 文件中。

```
void __netif_schedule(struct Qdisc *q)
{
  if (!test_and_set_bit(__QDISC_STATE_SCHED, &q->state))
        __netif_reschedule(q);
}

static inline void __netif_reschedule(struct Qdisc *q)
{
  struct softnet_data *sd;
  unsigned long flags;

  local_irq_save(flags);
```

```
    sd = &__get_cpu_var(softnet_data);
    q->next_sched = sd->output_queue;
    sd->output_queue = q;
    raise_softirq_irqoff(NET_TX_SOFTIRQ);
    local_irq_restore(flags);
}
```

__netif_schedule 是__netif_reschedule 的包装函数，它通过调用__netif_reschedule 完成以下主要任务：

1．将设备放入 CPU 的设备输出队列

将网络设备实例加入到 CPU 的输出设备队列 output_queue 里，像接收数据帧设备的 poll_list 队列一样，每个 CPU 都有一个自己的 output_queue 队列，output_queue 队列中的网络设备由 Qdisc->next_sched 指针链接在一起。

output_queue 是网络设备队列，这些网络设备或者是有数据需要向外发送，在前面的发送尝试中失败的设备；或者是输出队列被停止了一段时间后重新启动的设备。因为__netif_schedule 函数可以在中断执行现场内或中断执行现场外被调用，所以在__netif_reschedule 函数将设备加入 output_queue 队列前要关中断。

2．调度网络发送软件中断

__netif_schedule 在__netif_reschedule 函数中完成的第二个任务是调度网络子系统的发送软件中断 NET_TX_SOFTIRQ 的处理程序。__QDISC_STATE_SCHED 用于标志网络设备在 output_queue 队列中，这些网络设备有数据需要发送。注意在__netif_schedule 函数首先查看__QDISC_STATE_SCHED 标志位，即如果设备已经被调度过了，__netif_schedule 函数不做任何处理。

如果网络设备的发送过程在前面的处理中被停止过，我们用 netif_wake_queue 函数重新允许设备发送，并调度设备发送过程，即将设备放入 CPU 的发送设备队列中。

调用 netif_wake_queue 函数相当于调用了 netif_start_queue 与__netif_schedule，唤醒网络设备的队列是网络设备驱动程序的任务，同样，禁止/允许设备发送也是网络设备驱动程序的任务，不由内核函数来控制。通常上层内核函数只是调度设备发送过程，设备驱动程序禁止重新允许设备发送队列，所以 netif_wake_queue 函数通常由网络设备驱动程序调用，__netif_schedule 函数在其他地方调用，如在 net_tx_action 中。

网络设备驱动程序会在以下的情况下调用 netif_wake_queue 函数：

网络设备驱动程序使用 watchdog 时钟来恢复挂起的发送过程。在这种情况下，net_device->tx_timeout 虚函数通常先重启网络设备；在网络设备处于不可用的状态期间，可能有数据帧发送给网络设备，所以网络设备驱动程序应启动设备的发送队列，并调度设备，将设备放入 CPU 输出设备队列。

设备给驱动程序返回的信息表示，它又有足够的缓冲区来处理新的发送数据帧时，设备可以被唤醒。这时调用 netif_wake_queue 函数的原因是，在驱动程序停止设备队列期间可能有新的数据帧要发送。

```
static inline void netif_wake_queue(struct net_device *dev)
{
  netif_tx_wake_queue(netdev_get_tx_queue(dev, 0));
}
```

```
static inline void netif_tx_wake_queue(struct netdev_queue *dev_queue)
{
#ifdef CONFIG_NETPOLL_TRAP
  if (netpoll_trap()) {
        clear_bit(__QUEUE_STATE_XOFF, &dev_queue->state);
        return;
  }
#endif
  if (test_and_clear_bit(__QUEUE_STATE_XOFF, &dev_queue->state))
        __netif_schedule(dev_queue->qdisc);
}
```

6.5.3 队列策略接口

几乎所有的网络设备都使用队列来调度管理数据帧的输出流量，内核可以使用队列策略算法来安排先发送哪个数据帧后发送哪个数据帧，使发送过程更高效。详细的队列策略处理属于流量控制子系统的内容，在这一节中我们主要概略性地介绍有发送队列的网络设备驱动程序与数据链路层之间的主要接口及调度队列的方法。

1．队列策略

流量控制系统中的每种队列策略都提供了自己的函数，供数据链路层调用来完成队列的操作。这些方法都是以虚函数的方式实现的，其中最重要的虚函数有如下几种。

- Enqueue：向队列加入一个成员。
- Dequeue：从队列中提取一个成员。
- Requeue：将前面从队列中提取的数据成员重新放回队列（可能因为发送失败）。

当一个设备被调度来发送数据帧时，下一个要发送的数据帧由 qdisc_run 函数来选择，qdisc_run 函数间接调用与队列策略相关的 Dequeue 虚函数。实际的工作由 qdisc_restart 函数完成，qdisc_run 函数只是 qdisc_restart 的包装函数，它在调用 qdisc_restart 函数之前，过滤掉那些输出队列被停止的网络设备。qdisc_run 函数定义在 include/net/pkt_sched.h 文件中。

```
static inline void qdisc_run(struct Qdisc *q)
{
  if (!test_and_set_bit(__QDISC_STATE_RUNNING, &q->state))
        __qdisc_run(q);
}

void __qdisc_run(struct Qdisc *q)                          //net/sched/sch_generic.c
{
  unsigned long start_time = jiffies;

  while (qdisc_restart(q)) {

        if (need_resched() || jiffies != start_time) {
              __netif_schedule(q);
              break;
        }
  }

  clear_bit(__QDISC_STATE_RUNNING, &q->state);
```

```
}
```

2. qdisc_restart 函数

前面我们看到了网络设备在以下几种情况下会放入 CPU 的输出设备队列，被调度来发送数据帧：设备的输出队列中有数据等待发送；设备的输出队列被停止而现在重新启动，因此在前面的发送失败后，设备的输出队列中可能有数据等待发送。驱动程序本身不知道什么时候有发送数据帧到达，如果有数据等待发送，内核必须调度设备来完成。如果没有数据等待发送，dequeue 函数的调用会失败。qdisc_restart 函数根据 dequeue 函数的返回值来决定进一步的操作。qdisc_restart 函数定义在 net/sched/sch_generic.c 文件中。

```
static inline int qdisc_restart(struct Qdisc *q)
{
  struct netdev_queue *txq;
  int ret = NETDEV_TX_BUSY;
  struct net_device *dev;
  spinlock_t *root_lock;
  struct sk_buff *skb;
  //获取设备的输出队列，从队列中读一个数据帧的 skb 缓冲区，skb=q->dequeue()
  if (unlikely((skb = dequeue_skb(q)) == NULL))
       return 0;
```

（1）函数初始部分

qdisc_restart 函数的初始部分是为数据帧的发送做准备，如从数据帧缓冲区队列读取 Socket Buffer，获取发送过程需要的保护锁等。

qdisc_restart 函数执行时，首先就在 dequeue_skb 函数中调用 dequeue 函数获取一个 Socket Buffer。如果 dequeue 执行成功，接下来发送数据帧需要获取两个锁：

- 保护数据缓冲区队列的锁 qdisc->q.lock。这是 qdisc_restart 的调用者（dev_queue_xmit）需要的。
- 保护网络设备发送队列的锁 txq->_xmit_lock（在 HARD_TX_LOCK 中获取）。该锁由驱动程序的 hard_start_xmit 函数管理，如果设备驱动程序实现了自己的锁定机制，它设置 dev->features 的 NETIF_F_LLTX 标志位（无锁发送特点）来告诉内核不需要获取 txq->_xmit_lock 锁。

HARD_TX_LOCK 首先测试 dev->features 的 NETIF_F_LLTX 标志位，如果该位已设置，HARD_TX_LOCK 什么也不做，否则它将尝试获取防止对设备输出队列并发访问的锁 txq->_xmit_lock。

如果设备驱动程序不支持 NETIF_F_LLTX 特性，网络设备发送队列的锁已被其他所有者获取，发送失败；这意味着当前有别的 CPU 通过同一个网络设备发送数据。这时 qdisc_restart 要做的处理是将数据帧缓冲区 skb 重新放回队列，重新调度网络设备来发送数据帧。

```
//获取保护 Socket buffer 队列与网络设备输出队列的锁，防止并发访问
root_lock = qdisc_lock(q);

  spin_unlock(root_lock);
  //从 CPU 输出设备队列获取发送数据帧的网络设备，获取网络设备发送队列，将 Socket Buffer 挂到设备输出队列中
  dev = qdisc_dev(q);
```

```
    txq = netdev_get_tx_queue(dev, skb_get_queue_mapping(skb));
    //如果设备驱动程序没有实现对发送队列的锁定机制，则获取对网络设备传送队列保护的锁。设置通过网络设备发送队列传送数据的 CPU
    HARD_TX_LOCK(dev, txq, smp_processor_id());
    //如果设备发送队列未停止，调用 dev_hard_start_xmit 将数据帧放入设备发送队列，释放队列保护锁，释放 skb 队列保护锁
    if (!netif_tx_queue_stopped(txq) &&
       !netif_tx_queue_frozen(txq))
          ret = dev_hard_start_xmit(skb, dev, txq);
    HARD_TX_UNLOCK(dev, txq);
```

（2）执行发送

一旦驱动程序获取锁成功，就可以释放对数据帧缓冲区队列的保护锁，这样别的 CPU 就可以访问缓冲区队列。这时 qdisc_restart 函数就准备好了做实际的发送工作。

qdisc_restart 函数查看设备队列的状态 netif_queue_stopped，最后 qdisc_restart 通过 dev_hard_start_xmit 函数调用网络设备的虚函数 dev->netdev_ops->ndo_start_xmit 来发送数据帧。

函数 dev_hard_start_xmit 执行结果的返回值 NETDEV_TX_XXX 定义在 include/linux/netdevice.h 文件中，各返回值的含义如下:

- NETDEV_TX_OK：发送成功。这时已发送的数据帧的缓冲区还没有被释放，驱动程序本身不释放缓冲区，内核在执行网络发送软件中断 NET_TX_SOFTIRQ 中来集中释放发送完成的缓冲区，这样比驱动程序每发送一个就释放一个的效率更高。
- NETDEV_TX_BUSY：驱动程序发现网络设备硬件中没有足够的空间缓存要发送的数据帧，当检测到这个条件时，驱动程序通常也会同时调用 netif_stop_queue。
- NETDEV_TX_LOCKED：驱动程序已被锁定。

（3）函数结束处理

最后 qdisc_restart 函数根据驱动程序的返回值做结束处理。如果发送成功，qdisc_restart 不做任何别的处理；发送失败，将数据帧缓冲区重新放回队列，再调度设备发送。总结起来在以下条件成立的情况下，发送失败，数据帧需重新放回队列：

- 网络设备发送队列停止。
- 别的 CPU 获取了网络设备发送队列的保护锁。
- 驱动程序执行失败。

```
    spin_lock(root_lock);
    //根据 dev_hard_start_xmit 函数执行结果做处理
    switch (ret) {
    case NETDEV_TX_OK:

    //数据帧发送成功
          ret = qdisc_qlen(q);
          break;
    //驱动程序获取锁失败，说明当前有别的 CPU 通过该网络设备发送数据。做冲突处理，将 Socket Buffer 重新放回原队列
    case NETDEV_TX_LOCKED:
          ret = handle_dev_cpu_collision(skb, txq, q);
          break;
    //设备中没有足够的空间缓存发送数据帧，将数据帧重新放回队列
```

```
    default:
         if (unlikely (ret != NETDEV_TX_BUSY && net_ratelimit()))
              printk(KERN_WARNING "BUG %s code %d qlen %d\n",
                   dev->name, ret, q->q.qlen);
         ret = dev_requeue_skb(skb, q);
         break;
    }

    if (ret && (netif_tx_queue_stopped(txq) ||
            netif_tx_queue_frozen(txq)))
         ret = 0;

    return ret;
  }
```

6.5.4　dev_queue_xmit 函数

dev_queue_xmit 函数是 TCP/IP 协议栈网络层执行发送操作与设备驱动程序之间的接口。dev_queue_xmit 可以通过两个途径来调用网络设备驱动程序的发送函数 ndo_start_xmit：

- 流量控制接口：通过调用 qdisc_run 函数执行，在 6.5.3 中已讨论。
- 直接调用 ndo_start_xmit：用于网络设备不使用流量控制体系结构的情况。

当网络层（IP 层或 L3 层）调用 dev_queue_xmit 函数来发送数据帧时，所有数据帧所需要的信息如：输出网络设备、下一跳 IP 地址、数据链路层地址（MAC 地址）都已准备好，由网络层来初始化这些信息。

1．dev_queue_xmit 的主要功能

dev_queue_xmit 函数唯一的输入参数就是 skb，其中包含了发送数据帧需要的所有信息；skb->dev 是发送数据帧的网络设备；skb->data 指向发送负载数据的起始地址； skb->len 是负载数据的长度。dev_queue_xmit 完成下列主要功能：

- 查看数据帧是否被分割过，是否有其他的分片数据。如果是，查看网络设备是否具有 scatter/gather DMA 的功能特点，能直接处理数据片。如果设备不能处理，将数据片组装成一个完整的数据帧。
- 除非网络设备硬件能完成传输层（L4）的数据校验和计算，否则确定传输层完成了数据帧的校验和计算。
- 选择要发送的数据帧（因为有 skb 的队列，所以由 sk_buff 指针指向的缓冲区可能不是当前就能发送的数据帧缓冲区）。

2．dev_queue_xmit 的源码分析

下面我们就进入到 dev_queue_xmit 函数的源代码内部，分析 dev_queue_xmit 处理数据帧发送的过程。

（1）处理分片数据

在以下代码中，skb_shinfo(skb)->frag_list 不为空时，说明发送数据帧的负载数据被分割成了更小的片段，由分片数据缓冲区的链表组成；否则，负载数据是一个完整的数据块。

如果负载数据被分片，设备硬件不支持 scatter/gather DMA 功能特性，就将分片数据组

装成一个完整的数据块。另外，即使设备支持 scatter/gather DMA 功能特性，但分片数据存放在存储器的高端地址，设备不能访问这些高端地址，函数也需将分片数据重新组装（illegal_highdma）。

将分片数据重新组装的任务由__skb_linerize 函数完成，不过__skb_linerize 函数可能会由于以下原因执行失败：

- 分配缓冲区失败。
- 当前 sk_buff 是与其他子系统共享的缓冲区（对缓冲区的引用计数大于 1）。

```
int dev_queue_xmit(struct sk_buff *skb)
{
  struct net_device *dev = skb->dev;
  struct netdev_queue *txq;
  struct Qdisc *q;
  int rc = -ENOMEM;
  …
  //如果负载数据被分片，网络设备硬件不支持 scatter/gather 功能特性，就重新组装分片数据，使其成为一个完整数据包
  if (skb_shinfo(skb)->frag_list &&
     !(dev->features & NETIF_F_FRAGLIST) &&
     __skb_linearize(skb))
        goto out_kfree_skb;
  //如果被分片的数据存放在高端存储器区域，函数需要处理这种情况，将其组装
  if (skb_shinfo(skb)->nr_frags &&
     (!(dev->features & NETIF_F_SG) || illegal_highdma(dev, skb)) &&
     __skb_linearize(skb))
        goto out_kfree_skb;
```

（2）处理传输层校验和

传输层校验和的计算可以由软件完成也可以由硬件完成，但不是所有的网络设备硬件都可以完成校验和计算。如果网络设备硬件支持传输层校验和计算，会在设备初始化时设置 net_device->features 的相应标志位，通知协议栈上层不需考虑校验计算的问题。在以下情况下需要软件完成校验和计算：

- 设备硬件不支持对校验和计算。
- 设备硬件只支持在 IP 协议上发送的 TCP/UDP 数据帧的校验和计算，但当前发送的数据帧不使用 IP 协议，或使用传输层的其他协议。

当采用软件计算校验和时，这个任务事实上是由 skb_checksum_help 函数完成的。一旦数据帧的校验和处理完成，待发送数据帧的所有头信息就已准备好，下一步就是决定将哪个数据帧交给网络设备发送。

```
//如果数据帧还没有完成校验和计算，而设备硬件不支持计算校验和功能，就由软件完成校验计算
if (skb->ip_summed == CHECKSUM_PARTIAL) {
      skb_set_transport_header(skb, skb->csum_start -
                              skb_headroom(skb));
      if (!dev_can_checksum(dev, skb) && skb_checksum_help(skb))
            goto out_kfree_skb;
}
```

这时函数的执行取决于网络设备是否使用流量控制体系结构来完成发送，也即取决于相应的队列策略。dev_queue_xmit 只处理了一个缓冲区（在需要时重新组装分片数据，完成校验和），下一步的发送主要取决于是否使用队列策略，使用的什么队列策略和网络设备

输出队列的状态，立即发送的数据帧可能不是 dev_queue_xmit 才收到的数据帧。dev_queue_xmit 函数的处理流程如图 6-13 所示。

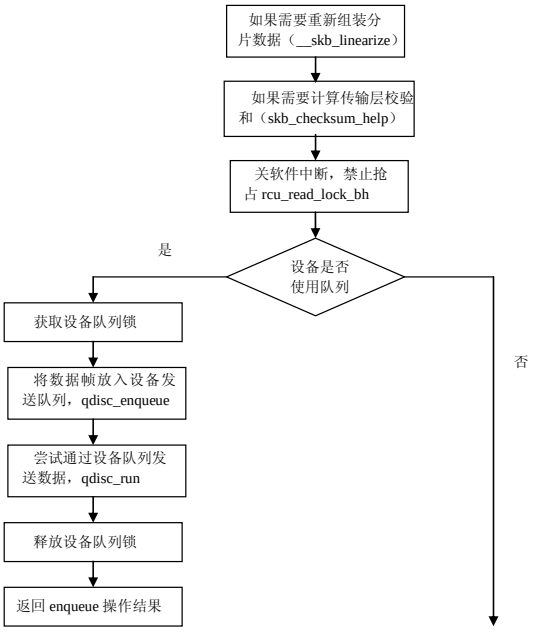


图 6-13　dev_queue_xmit 函数的实现流程

（3）有队列的网络设备

如果设备有发送队列，设备的队列策略可通过设备队列的 dev->txq->qdisc 访问，输入的数据帧由 enqueue 虚函数放入设备发送队列，随后一个数据帧会从网络设备发送队列中读出，由 qdisc_run 函数发送（见 6.5.3 节）。

```
//禁止软件中断与抢占优先，获取设备的发送队列
rcu_read_lock_bh();
txq = dev_pick_tx(dev, skb);
q = rcu_dereference(txq->qdisc);
…
//将接到的数据帧放入设备发送队列 q->enqueue，获取队列保护锁
if (q->enqueue) {
      spinlock_t *root_lock = qdisc_lock(q);
      spin_lock(root_lock);
//测试设备队列是否处于激活状态，如果是，调用 qdisc_run 尝试发送数据帧，否则扔掉数据包，释放锁
      if (unlikely(test_bit(__QDISC_STATE_DEACTIVATED, &q->state))) {
            kfree_skb(skb);
            rc = NET_XMIT_DROP;
      } else {
            rc = qdisc_enqueue_root(skb, q);
            qdisc_run(q);
      }
      spin_unlock(root_lock);
      goto out;
}
```

（4）无队列设备

有的设备没有发送队列，如 loopback 设备，对于这样的设备无论什么时候要发送数据帧，都会立即发送（因为设备中没有队列缓存数据帧，如果遇到错误数据帧就直接扔掉，没有第 2 次发送机会）。如果查看 drivers/net/loopback.c 文件中 loopback 设备的数据帧发送函数 loopback_xmit，可以看到在函数的末尾直接调用了 netif_rx 接收刚发送的数据帧。因为没有队列，发送函数完成两个任务：在一端发送，在另一端立即接收。无队列网络设备与有队列网络设备在发送数据帧时的区别对比如图 6-14 所示。

dev_queue_xmit 函数的最后一部分是处理没有使用队列策略的设备，这些设备没有输出队列，处理过程与 qdisc_run 的类似，有以下区别：

发送失败时，驱动程序不能把缓冲区放回队列，所以 dev_queue_xmit 直接扔掉数据帧，如果上层使用了可靠发送协议如 TCP，上层协议最终会控制重传数据帧。

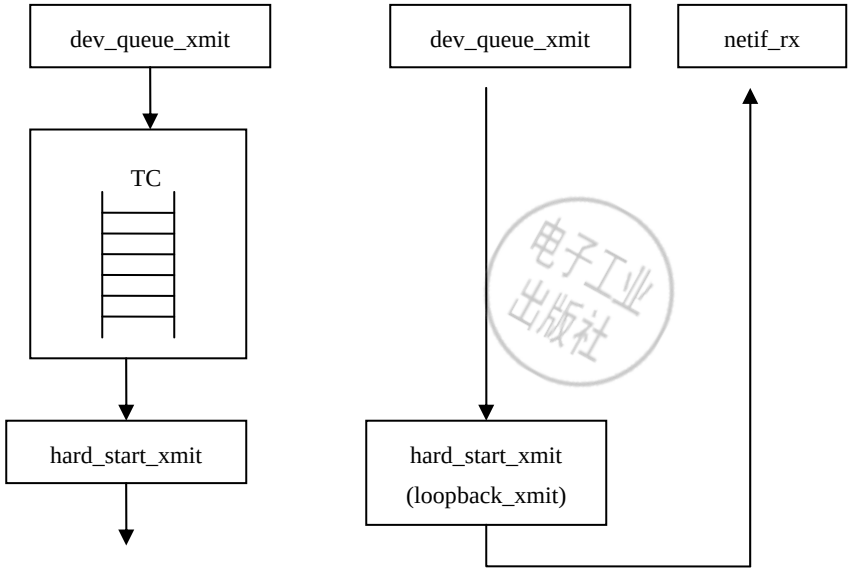


图 6-14　无队列设备与有队列设备的对比

6.5.5　发送软件中断

net_rx_action 函数是网络接收软件中断 NET_RX_SOFTIRQ 的处理程序，NET_RX_SOFTIRQ 的执行由设备驱动程序触发（在某些特殊条件下由其自己触发），对输入数据帧作部分处理，是中断处理程序的后半段。

网络发送软件中断 NET_TX_SOFTIRQ 的处理程序 net_tx_action 与 net_rx_action 的工作方式类似，它由设备在以下两处不同的执行现场调用 raise_softirq_irqoff(NET_TX_SOFTIRQ) 触发执行：

- 当网络设备发送允许时，由 netif_wake_queue 触发。这时它保证需要的条件满

足时（如设备有足够的空间缓存数据帧），等待发送的数据帧确实会被发送。

- 当发送结束，网络设备驱动程序通知内核相关的缓冲区可以释放时，由 dev_kfree_skb_irq 触发。这时 net_tx_action 回收发送成功的数据帧的 Socket Buffer 内存空间。

释放 Socket buffer 的任务之所以放在发送软件中断中来完成，是因为我们都知道发送数据帧的设备驱动程序运行在中断执行现场，中断服务程序的执行应尽可能的快，只完成最重要的任务的处理；释放内存空间花费的时间较长，所以把其放在 net_tx_action 软件中断中来完成。设备驱动程序调用 dev_kfree_skb_irq 函数而不是 dev_kfree_skb 函数来完成缓冲区释放。后者把 sk_buff 放回每个 CPU 的高速缓冲存储器中，前者只是把缓冲区的指针放入 CPU 的 softnet_data 数据结构的 completion_queue 队列中，让 net_tx_action 来完成实际的释放任务。

接下来我们进入 net_tx_action 函数源代码内部，来分析它如何完成其两个主要功能。

1．回收发送完成的缓冲区

net_tx_queue 函数一开始将回收所有在 completion_queue 队列中的缓冲区，这些缓冲区由设备驱动程序调用 dev_kfree_skb_irq 函数放入 completion_queue 队列。net_tx_action 运行在非中断执行现场，设备驱动程序可以在任何时候把缓冲区加入到 completion_queue 队列。这样 net_tx_action 访问 completion_queue 队列时要关中断，为了确保关中断的时间尽量短，net_tx_action 首先将 completion_queue 队列清空，将队列指针保存在局部变量 clist 中，随后 net_tx_action 遍历局部队列，释放所有的缓冲区__kfree_skb，驱动程序就可以继续向 completion_queue 队列中加入新成员了。

```
static void net_tx_action(struct softirq_action *h)
{
//获取当前 CPU 的标识，如果 completion_queue 队列不为空，处理 completion_queue 队列中发送完成的缓冲区
  struct softnet_data *sd = &__get_cpu_var(softnet_data);
  if (sd->completion_queue) {
        struct sk_buff *clist;
//关中断，将 completion_queue 队列存入局部变量，清空队列，开中断
        local_irq_disable();
        clist = sd->completion_queue;
        sd->completion_queue = NULL;
        local_irq_enable();
//遍历局部队列 clist 释放所有队列中的缓冲区
        while (clist) {
              struct sk_buff *skb = clist;
              clist = clist->next;
              WARN_ON(atomic_read(&skb->users));
              __kfree_skb(skb);
        }
  }
```

2．发送数据帧

net_tx_action 函数的第二个任务是发送数据帧，工作过程与前类似：首先使用局部变量保存 CPU 发送设备队列的地址，以便保证硬件中断能安全访问该队列。要注意的是，任何设备在试着发送任何数据之前，要获取输出设备队列的保护锁；如果获取锁失败（因为别的 CPU 持有锁），net_tx_action 调用__netif_schedule 函数重新调度设备发送数据。

我们已经在 6.5.2 节中给出了 qdisc_work 函数的处理过程，net_tx_action 函数从设备队列头开始处理数据帧的发送，遍历所有的设备。

```
//如果 CPU 设备队列不为空，则设备有数据需要发送
 if (sd->output_queue) {
       struct Qdisc *head;
 //关中断，将 CPU 输出设备队列地址保存在局部变量 head 中，清空 CPU 设备队列，开中断
       local_irq_disable();
       head = sd->output_queue;
       sd->output_queue = NULL;
       local_irq_enable();
       //遍历输出设备队列，获取设备输出队列锁，调用 qdisc_run 尝试发送数据帧
       while (head) {
             struct Qdisc *q = head;
             spinlock_t *root_lock;
             head = head->next_sched;
             root_lock = qdisc_lock(q);
             if (spin_trylock(root_lock)) {
                   smp_mb__before_clear_bit();
                   clear_bit(__QDISC_STATE_SCHED,
                         &q->state);
                   qdisc_run(q);
                   spin_unlock(root_lock);
             } else {
                   //如果获取设备输出队列锁不成功，重新调度设备发送数据，即重放入 CPU 的输出设备队列
                   if (!test_bit(__QDISC_STATE_DEACTIVATED,
                             &q->state)) {
                         __netif_reschedule(q);
                   } else {
                         smp_mb__before_clear_bit();
                         clear_bit(__QDISC_STATE_SCHED,
                               &q->state);
                   }
```


总结起来，数据帧的发送路径如图 6-15 所示。

3．释放缓冲区的执行现场

在结束对 net_tx_action 函数的分析之前，我们看看函数调用释放缓冲区的执行现场，有的函数需要既可以在中断执行现场释放缓冲区也可以在非中断执行现场释放缓冲区，这时内核提供了一个包装函数来优化处理过程。

```
void dev_kfree_skb_any(struct sk_buff *skb)
{
  if (in_irq() || irqs_disabled())
        dev_kfree_skb_irq(skb);                 //在中断执行现场
  else
        dev_kfree_skb(skb);                     //非中断执行现场
}
```

dev_kfree_skb_irq 函数是在中断执行现场调用的，缓冲区只能在没有别的进程引用时才能释放。

```
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);
    }
}
```

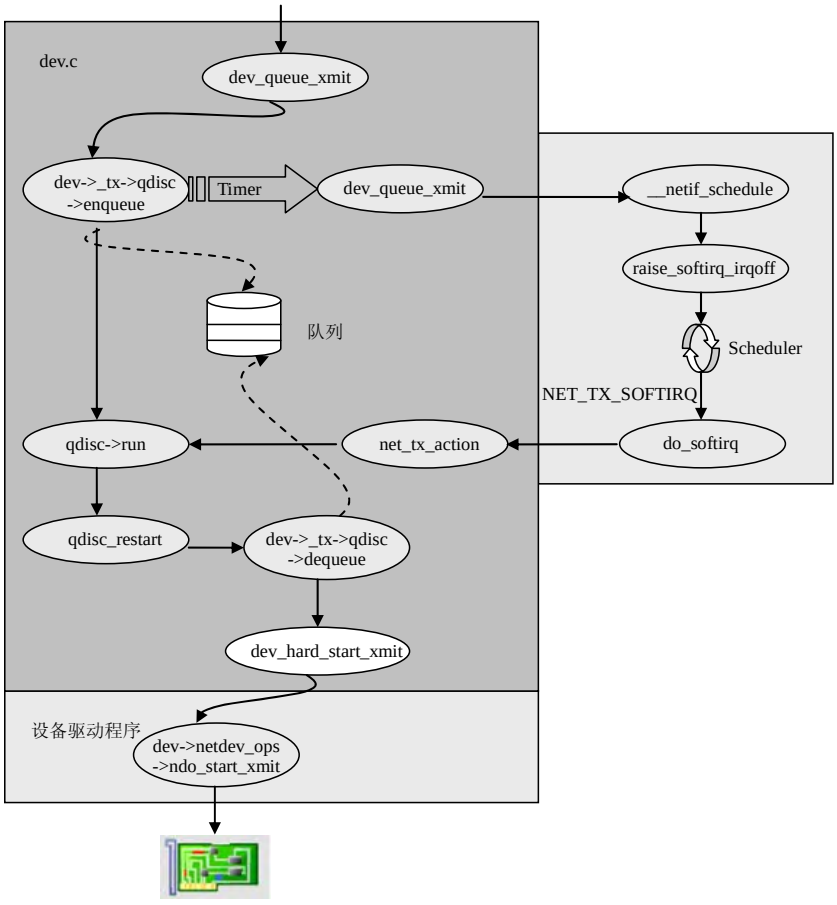


图 6-15　数据帧的发送路径

6.5.6　Watchdog 时钟

在介绍允许/禁止发送数据帧的部分我们看到，发送过程在某些条件下可以被设备驱动程序关闭，内核认为发送过程被关闭应该是临时的，所以如果发送过程在一段时间内没有重新被打开，内核就认为设备出现了故障，要重启设备。

- trans_start：该数据域每次由设备驱动程序启动发送最后一个数据帧时初始化的时间戳。

- watchdog_timer：由流量控制启动的时钟，该时钟超时后执行的是 dev_watchdog 函数，此函数定义在 net/sched/sch_generic.c 文件中。
- watchdog_timeo：是时钟应等待的时间值，由驱动程序初始化，当该值设置为 0 时，watchdog_timer 不启动。
- tx_timeout：由设备驱动程序实现的处理程序，在 watchdog_timeo 时钟超时时，dev_watchdog 重启设备。

当时钟超时时，内核的处理程序 dev_watchdog 调用 tx_timeout 函数指针指定的函数，后者通常重新启动网络设备，调用 netif_wake_queue 重启网络设备的调度器。

watchdog_timeo 的值需根据网络设备的类型不同，设置相应的值，如果驱动程序没有初始化该值，其默认值为 5。最后我们将数据帧的接收/发送途径总结如图 6-16 所示。

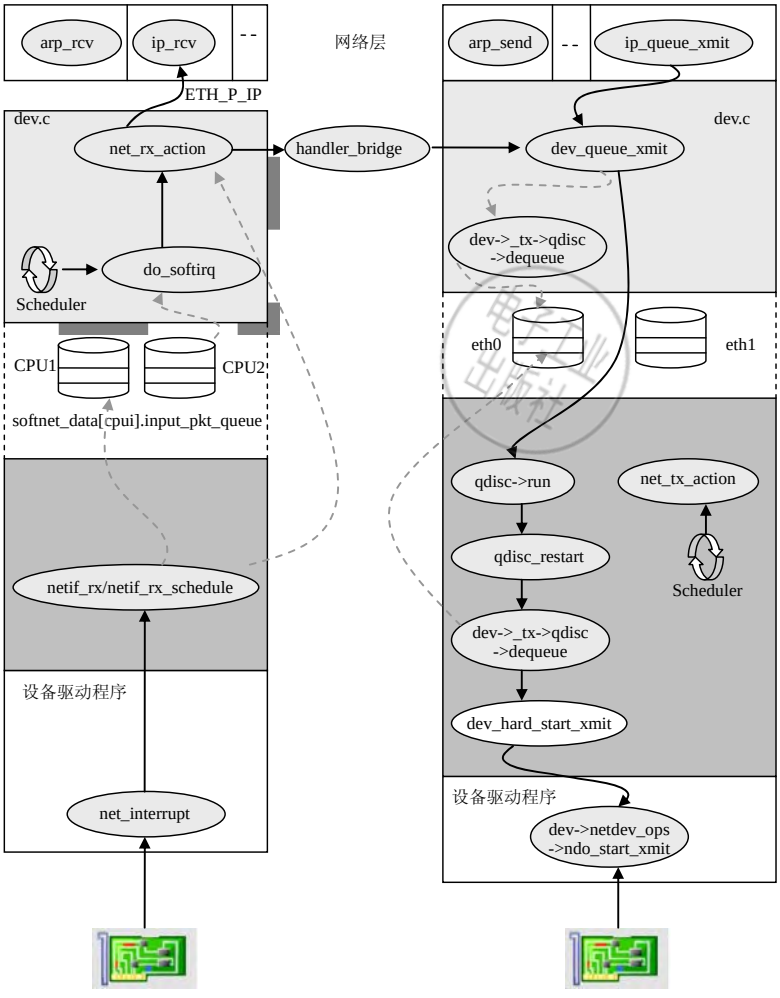


图 6-16　数据帧接收/发送路径

这个过程由每个设备的时钟来控制，在函数 dev_active 激活设备时，设备的时钟由

dev_watchdog_up 函数设置；时钟会周期性地超时，确保设备运行一切正常后，时钟重启。当时钟查看到设备的问题是因为设备的输出队列停止（netif_queue_stopped 返回 TRUE）时间太长引起时，该时钟由网络设备驱动程序发送最后一个数据帧时启动，如果时钟处理程序发现设备发送队列停止的时间过长，就会重新启动网络设备。以下是 net_device 用于实现该机制的数据域。

6.6　本章总结

数据链路层最基本的任务是使用物理层提供的服务（通过网络设备驱动程序），将网络层传来的数据传输出去，将从网络设备接收到的数据上传到网络层，因此，无论是接收数据还是发送数据，在数据链路层都需要做一定的处理。本章的任务就是分析在 Linux 内核的 TCP/IP 协议栈中，数据链路层的功能在以下几个方面的实现技术：

（1）管理 CPU 上的网络数据帧队列、网络设备队列的数据结构。

（2）Linux 内核中实现的两个与网络设备驱动程序之间接收网络数据帧的接口。

（3）数据链路层处理接收网络数据帧的主要函数，网络接收软件中断处理程序实现的流程与源代码分析。

（4）数据链路层处理发送网络数据帧的主要函数，网络发送软件中断处理程序的实现流程与源代码分析。

（5）数据链路层与网络层之间接收数据帧的接口设计与管理。

从以上几个方面可以看到数据链路层在 Linux 内核中实现的几个特点：其一，结合 struct net_device 数据结构与 net/core/dev.c 文件中的代码形成了对上层协议的统一接口；其二，在每个 CPU 上实现的队列管理数据结构，完成对输入数据帧与输出网络设备的高效管理；其三，数据链路层与网络层之间的接收接口可以有效地将数据帧传给多个上层协议处理，实现接口透明化。

第 7 章　网络层传送

本章主要介绍 TCP/IP 协议栈中网络层 Internet 协议的基本任务，IP 协议数据包格式与 IP 协议选项的使用，并在此基本上讲解 Linux 内核实现的 IP 协议实例的数据结构、接收/传送数据特点、实现流程与主要功能函数的源代码，以及网络层与传输层的接口设计。

网络层协议是 TCP/IP 协议栈与网络设备驱动程序所管理的硬件发送的连接处，网络数据包经过网络层的处理后，已包含了所有向外发送需要的信息，如源地址、目标地址、协议头等，可交由网络设备发送。另一方面，接收到的网络数据包在网络层确定其进一步发送的路径，是在本机向上传递，还是继续向前发送。数据链路层的驱动程序只关心将上层交给的数据包向外发送，将接到的数据包向上投递，而网络层协议的关键任务就是决定数据包的去向。

网络层在 TCP/IP 协议栈中处于第 3 层，我们在叙述中常以 L3 表示网络层，在网络层中应用最广泛的协议就是 Internet 协议，本章我们就分析 Internet 协议在 Linux 内核协议栈的实现，所以在描述时又常以 IP 层表示本章所述的网络层。

Linux 内核支持许多网络层的协议，如 AppleTalk、DECnet 和 IPX。本章我们介绍的是目前应用最广泛的 IP 协议在 Linux 中的实现。在前面的章节中我们给出的是 TCP/IP 协议栈的一个总体分层结构框图，这里首先对 TCP/IP 协议栈各层常用协议及协议功能做一个概述，TCP/IP 协议栈各层常用协议实例如图 7-1 所示。

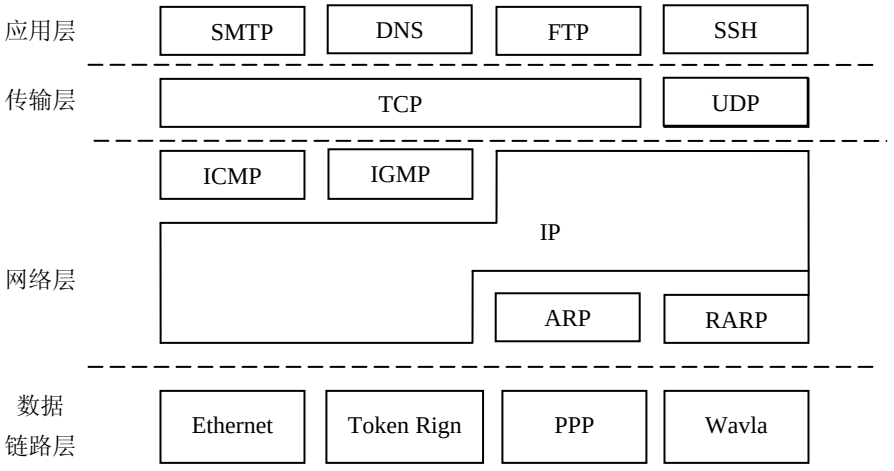


图 7–1　TCP/IP 协议栈

- 数据链路层：在这一层我们看到的是网络适配器和它们的驱动程序。
- ARP（Address Resolution Protocol）：也属于数据链路层，它将 IP 地址转换成本地网络适配器的 MAC 地址。

- IP （ Internet Protocol ）：在整个 TCP/IP 协议栈中占核心地位，IP 协议使在网络中互连的主机可以相互通信。
- ICMP （Internet Control Message Protocol）：处理 IP 协议传输过程中的错误信息。
- IGMP （Internet Group Management Protocol）：用于管理局域网中的组发送。
- TCP （Transmission Control Protocol）：可靠的、面向连接的传输层协议。它负责在两个应用之间提供可靠的数据传输，在 Internet 中 TCP 是使用最广泛的传输层协议。
- UDP（User Datagram Protocol）：传输层的一个简单协议，在 Internet 的两个应用之间提供无连接和不可靠的数据包传输。
- 应用层：在应用层中驻留了各种标准的应用协议。如 HTTP、TELNET、FTP、SMTP、DNS 等。在本书中不讨论应用层协议的实现，因为它们不是 Linux 内核的一部分，但我们会讨论网络应用程序开发的技术。

本书以后的章节就将逐一分析 TCP/IP 协议栈中的各协议在 Linux 内核中的设计与实现。我们首先从分析网络层 IP 协议的实现开始，本章的主要内容包括：

- IP 层的主要任务和应用策略。
- Linux 内核中 IP 层的接收函数如何处理输入数据包，如何处理 IP 选项。
- 数据包在 IP 层的前送和本地发送过程，即输入数据包如何在本地发送给传输层的处理程序，或当 IP 目标地址不是本机时，数据包如何前送。
- 传输层协议如何与 IP 层接口。
- 数据的分片和重装处理。

7.1　Internet 协议的基本概念

在理解 IP 协议在 Linux 内核中的实现之前，首先需要了解网络层，特别是 IP 协议的功能，即它在整个协议栈中的作用。在网络数据包中，描述网络层 IP 协议功能、数据包在网络层的处理驻处，包含在负载数据的 IP 协议头中。这一节就描述 IP 协议要完成的功能，以及 IP 协议头信息如何支持这些功能。

7.1.1　Internet 协议的任务

网络数据包（无论输入数据包还是输出数据包）进入网络层后，IP 协议的函数要对数据包做以下处理：

1．数据包校验和检验

数据包如有错误可以立即扔掉，这些错误可能是校验和不正确、协议头超过了边界，或者别的原因。

2．防火墙对数据包过滤（Firewalling）

网络过滤防火墙子系统（用户可以用 iptable 命令配置）要对数据包做安全检查，确定是否允许数据包继续向前发送。在数据包发送过程中，有好几处都会用到网络过滤子系统的功能来检查数据包的安全性，根据事先配置的网络过滤要求，网络过滤子系统可能对数

据做一些改动。

3．IP 选项处理（Handling Options）

网络数据包负载数据中的 IP 协议头包含选项，说明了 IP 协议应对数据包做何种处理，因为有的信息应用层也要使用。这些选项的设置可以让数据包发送方和网络系统管理人员跟踪数据包发送的过程。

4．数据分片和重组

IP 数据包的最大长度可以达到 64KB，这是由 IP 协议头中描述 IP 数据包长度的数据域占用的位数决定的（16 位，见图 7-2）。大多数数据包都没有达到这个最大值。

当数据包从一个网络传到另一个网络时，各网络中网络适配器硬件的最大传送单元（MTU）值有很大不同。所以，如果一个数据包的长度在传输过程中，相对于某一跳网络设备硬件的 MTU 而言太大，这时就必须将数据包分割成更小的数据片；并且在以后的传输过程中，每个分割了的数据片还可以继续分割。在这些数据片到达最终目的地址后，接收数据包的目标 IP 协议处理程序又需要将它们组装还原成最初的数据包。

5．接收、发送和前送

在 IP 层要处理的数据包分为以下 3 类，每一类由不同的 IP 协议函数处理。

- 接收数据包：从网络上接收到的数据包，通过数据链路层上传到网络层后，由 IP 层的接收函数处理。
- 外传数据包：本地主机产生的输出数据包，经 TCP/IP 协议栈上层下传至网络层后，由 IP 层的发送函数处理。
- 前送数据包：是与发送相关的功能，但处理的是从网络上接收到的数据包，该数据包的目标 IP 地址不是本地主机，只是经本地主机继续前送。

网络层 IP 协议处理函数处理网络数据包的依据是 IP 协议头。在 TCP/IP 协议栈中，各层协议都有自己的协议头信息，来描述各层协议需要对数据包做何处理，所以要理解 IP 协议层处理函数的设计和实现原理，我们需要知道 IP 协议头中都有什么数据域，这些数据域的含义与作用，在 Linux 内核中是如何描述 IP 协议头信息的。

7.1.2 Internet 协议头

Internet 协议头的基本组成如图 7-2 所示，下面分别介绍各数据域的含义。

1．Internet 协议版本（Version）

当前发送的数据包使用的 Internet 协议版本，目前只实现了 IPv4 和 IPv6。

2．IP 协议头长度（ Header Length，IHL)

IP 协议头长度描述的是协议头信息一共占有多少个单元，由 4 位描述，说明 IP 协议头最大可以有 16 个单元，每个单元 32 位。

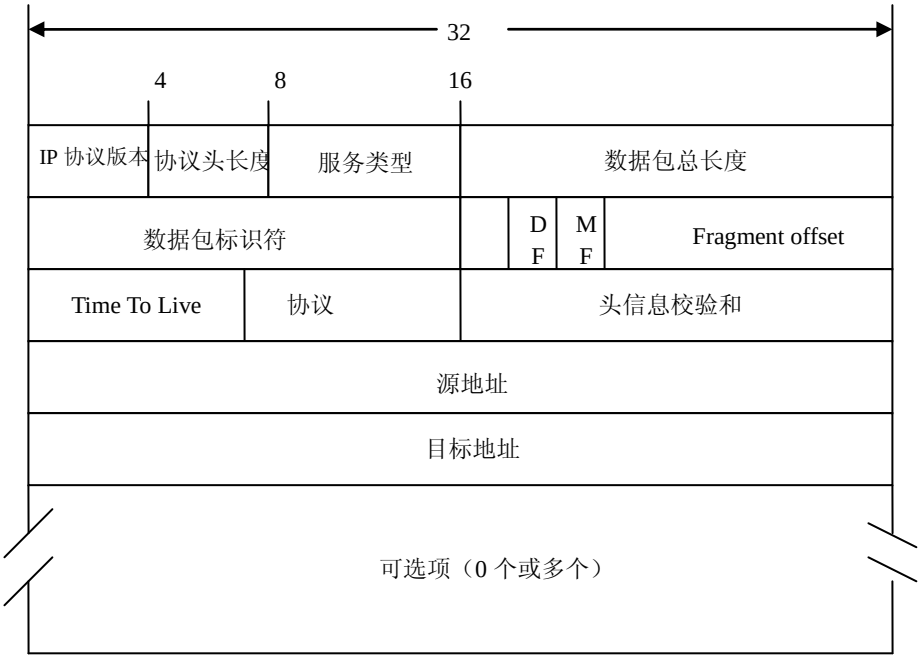


图 7-2　IP 协议头示意图

3．**服务类型**（Type of Service，ToS)

数据包的服务类型是一个 8 位的数据域。由 QoS（Quality of Service，质量服务）子系统使用。QoS 用于描述网络的服务质量和数据优先权，要实现 QoS，首先要对数据进行分类，在 RFC1349 规范中，ToS 字段用于告诉路由器数据包的发送者最关心的是哪个发送特性（最小延迟、最大吞吐量等），如图 7-3（a）所示。其最高位必须为 0，其余 3 位描述的是数据优先级别，一共可以有 0~7，共 8 个优先级。

图 7-3　ToS 字段的构成

在 RFC2474 规范中重新定义了 ToS 字段，引入了差分服务（Differentiated Services，简写为 diffserv）模式。其构成如图 7-3（b）所示。DSCP 表示 Diff Serv Code Point，占 6 位，是一个大小不超过 63 的值。每一个合法值都是唯一的，具有特殊含义，描述了如何处理数据包。CU 为当前未使用。

4．**数据包总长度**（Total Length）

这个数据域指明整个 IP 数据包的总长度，包括 IP 协议头信息在内，以字节为单位计算，如图 7-4 所示。

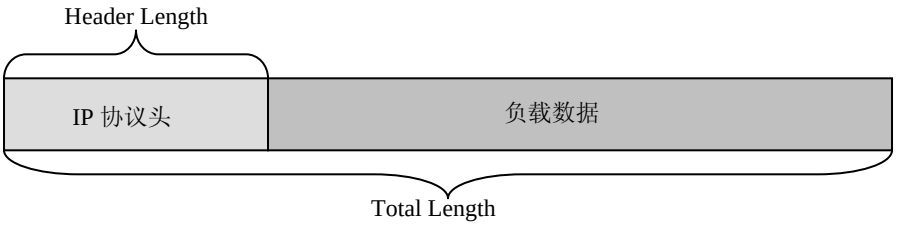


图 7-4　数据包总长度示意图

5．**数据包标识符**（Identification）

每个 IP 数据包都有一个唯一的标识符以相互区分。这个数据域在数据的分片和重组处理中起着重要作用。一个数据包被分割成小的数据片后，属于同一数据包的数据片有相同的数据包标识符（ID）；数据片传到目标主机后，IP 函数需要依据协议头中的数据包标识符（ID）信息确定哪些数据片属于同一数据包，应该组装在一起。

6．DF（Don’t Fragment）/MF（More Fragment/Fragement Offset）

DF 表示不对数据包进行分割（即使数据包的长度超过了网络设备最大发送单元 MTU 的值）；MF 表示有更多数据片，以及当前数据片在原数据包中相对于数据起始地址的偏移量；这个数据域同时表示随后还会接收到更多的数据片。DF/MF 及数据包标识符 Identification 都是用于 IP 协议的数据分片和重组处理的。

7．**数据包存活期**（Time to Live, TTL）

TTL 表示数据包的存活期，规定从 IP 数据包开始发送起经过多少秒后，数据包还没到达目标地址，就扔掉该数据包。每个路由器在将收到的数据包前送之前都对该数据域的值减 1；当 TTL 的值为 0 时数据包还没到达目标地址就会被扔掉。数据包的 TTL 初值由数据包负载类型确定，但在大多数情况下，都使用默认值 64。如数据包被扔掉，数据包发送方会收到一条 ICMP 消息（Internet Control Message Protocol）。

8．**上层协议**（Protocol）

代表网络数据包中网络层的上一层（传输层）使用的协议。IP 层的协议处理程序根据这个字段的值把数据包传给正确的上层协议处理程序。

9．**校验和**（Header checksum）

对数据包 IP 协议头信息做的校验和，保证 IP 协议头在发送后仍是正确的。该校验和不覆盖数据包的负载数据部分。

10．**源地址/目标地址**

发送数据包的主机 IP 地址与接收数据包的目标主机 IP 地址。

11．IP **选项**（Options）

IP 选项域可以为空，或者为占多个字节的选项值。选项值最大可以包含 40 个字节，

它的大小为：协议头的总长度-20（20 是 IP 协议头没有选项时的长度）。IP 选项的最大长度为 40，是因为：描述协议头总长度（Header Length）信息的数据域占了 4 位（bit），它描述了协议头信息一共可以有多少个单元。4 位能表示最大值为 15，协议头信息中每个单元是 32 位（4 个字节）。所以 IP 协议头的最大长度为 15×4 字节 = 60 字节。IP 协议头占了 20 个字节，还余 40 个字节可供 IP 选项使用。

7.1.3　Linux 内核中描述 IP 协议头的数据结构

在 Linux 内核中，IP 协议头由数据结构 iphdr 表示，当数据包到达 IP 层后，sk_buff->data 指针就已事先调整到 Socket Buffer 中存放 IP 协议头信息的起始地址处。IP 层的协议处理函数就从 Socket Buffer 中解析出 IP 协议头信息，并放入 iphdr 类型的变量中，以便下一步协议处理函数按照 IP 协议头信息处理数据包。

```
struct iphdr {                                  include/linux/ip.h
#if defined(__LITTLE_ENDIAN_BITFIELD)           //如果使用的是 little endian 字段顺序
  __u8  ihl : 4,
        version: 4;
#elif defined (__BIG_ENDIAN_BITFIELD)           //如果使用的是 big endian 字段顺序
  __u8  version: 4,
        ihl: 4;                                 //IP 协议头长度
#else
#endif
  __u8        tos;                              //服务类型
  __be16      tot_len;                          //IP 数据包总长度
  __be16      id;                               //IP 数据包标识符
  __be16      frag_off;                         //分片数据在数据包中的偏移量
  __u8        ttl;                              //数据包生存周期
  __u8        protocol;                         //上层协议
  __sum16     check;                            //校验和
  __be32      saddr;                            //数据包源地址
  __be32           daddr;                       //数据包目标地址
}
```

各数据域的含义与前面描述的 IP 协议头一一对应。在理解了 IP 协议规范的基础上，在本章以后的各节中我们将逐一分析 IP 协议层的任务：接收、发送和前送数据包；IP 选项处理；数据包分片/重组；校验和处理等在 Linux 内核中的实现。

7.2　IP 协议实现前的准备工作

Linux 内核中任何组件要在系统运行期间被正确地调用执行，首先需要在适当的条件和时间初始化，建立其运行环境并向内核注册，IP 协议子系统的初始化是在内核启动期间完成的。

7.2.1　协议初始化

IP 协议正确运行的前提是 PF_INET 协议族正确初始化。PF_INET 协议族的初始化程序建立起 TCP/IP 协议栈运行的环境，调用 TCP/IP 协议栈各层协议实例的初始化程序，将协议实例注册到内核（如果协议为直接编译到内核，比如编译为模块，则协议实例在装载

模块时初始化）。

1．PF_INET 协议族初始化

PF_INET 协议族的初始函数定义在 net/ipv4/af_inet.c 文件中，由 inet_init 函数实现。在内核启动期间，内核调用 inet_init 函数完成 PF_INET 协议族的初始化任务，inet_init 函数完成了 PF_INET 协议族的基本初始化后，会调用协议族中各层协议的初始化函数建立起 TCP/IP 协议栈。协议栈中有的协议是 Linux 内核网络功能的基础，不能从内核中移去，必须直接编译到内核中，如 TCP/UDP、IP 协议；有的协议实例可以以模块方式在运行期间动态加载到内核。Tnet-init 函数的主要功能为：

- 注册协议栈各协议实例，如 TCP、UDP、ICMP 等。
- 初始化 IP 协议。
- 为套接字操作建立内存槽（TCP 和 UDP），以便在打开套接字时为其分配所需的内存。
- 调用 dev_add_pack 注册 IP 数据包的接收处理函数 ip_rcv。将 IP 协议数据包的处理函数 ip_rcv 插入到网络层与数据链路层的接口链表 ptype_base 中（见 6.4 节）。
- 初始化 AF_INET 协议族在/proc 文件系统中的入口。

这里我们主要看看 Tnet-init 函数中与各协议实例初始化相关的代码片段。

```
static int __init inet_init(void)                    //net/ipv4/af_inet.c
{
  …
  //注册传输层协议数据包处理数据结构实体，加入 TCP/IP 协议栈各基础协议，建立协议栈运行环境，初始化各协议实例
  arp_init();                                        //建立 ARP 协议模块
  ip_init();                                         //建立 IP 协议模块
  tcp_v4_init();
  //建立 TCP/UDP 内存槽等
  tcp_init();
  udp_init();
  …
  dev_add_pack(&ip_packet_type);
  …
}
fs_initcall(inet_init);
```

```
static structpacket_type ip_packet_type = {
  .type=__constant_htons(ETH_P_IP),
  .func=ip_rcv,
  .gso_send_check=inet_gso_send_check,
  .gso_segment=inet_gso_segment,
  .gro_receive=inet_gro_receive,
  .gro_complete=inet_gro_complete,
};
```

从 inet_init 函数的前缀_init 可以知道，inet_init 函数会在内核启动系统初始化时执行，宏 fs_initcall(inet_init)说明了 inet_init 函数在系统初始化中的调用顺序，PF_INET 协议族的初始化会在网络子系统初始化（subsys_initcall）之后执行。

2．IP 协议初始化

IPv4 协议初始化由 ip_init 函数完成，当在 IP 层获取数据包后，要完成的最重要的任

务是决定如何传递数据，所以 IPv4 协议初始化的任务就是：

- 初始化路由子系统，建立独立于协议的路由缓冲表。路由表中保存了数据包发送的目标或下一跳的地址信息。
- 初始化管理裸 IP 的基本功能。
- 如果配置了 IP 组发送功能，由于组发送功能使用/proc 文件系统，所以还要初始化组发送在/proc 文件系统中的入口。

```
void __init ip_init(void)                  //net/ipv4/ip_output.c
{
  ip_rt_init();                            //初始化路由子系统
  inet_initpeers();                        //初始化裸 IP
#if defined(CONFIG_IP_MULTICAST) && defined(CONFIG_PROC_FS)
  igmp_mc_proc_init();
#endif
}
```

IP 协议实例不能从内核中卸载（即它不能编译成模块，只能直接编译到内核中），所以 IP 协议没有卸载模块的函数。

网络层 IP 协议处理数据包的接收和发送过程中，有多处都会调用网络过滤子系统和路由子系统的回调函数。使用网络过滤子系统是为了确保系统安全，在数据包进一步处理前，对数据包进行安全性检查；路由子系统则给出数据包向何处发送的路由信息。网络过滤子系统的内容在本书中不会介绍，这里只说明其在 TCP/IP 协议栈中的调用位置及其回调函数的使用方法和输出值，以便理解代码的走向。路由子系统我们会在第 12 章中详细讲解，这里给出在 TCP/IP 协议栈中使用的以上两个子系统的 API。

7.2.2　与网络过滤子系统的交互

数据包到达后，为了保证网络安全，首先需对数据包做过滤检查才能决定是否对数据包做进一步处理。这项检查由网络过滤子系统实现。

在 TCP/IP 协议栈中，数据包所经之处会多次调用网络过滤子系统的回调函数来过滤网络数据包，网络过滤子系统函数在 TCP/IP 协议栈中出现的场合有：

- 接收数据包时。
- 前送数据包（路由决策前）。
- 前送数据包（路由决策后）。
- 发送数据包时。

在以上列举的每一处：接收、前送、发送数据包，相应的协议处理函数都会分两个阶段实现。

第一阶段函数名通常为 do_something（如 ip_rcv），do_something 函数只对数据包做必要的合法性检查，或为数据包预留需要的内存空间。

第二阶段函数名通常为 do_something_finish（如 ip_rcv_finish），是实际完成接收/发送操作的函数。do_something 函数在完成对数据包的合法性检查、内存空间的预留后，在函数结束之前调用网络过滤子系统的回调函数 NF_HOOK 来对网络数据包进行安全检查。该回调函数需要的参数之一就是：当前是在哪个功能点处调用的网络过滤回调函数（接收数据包时还是发送数据包时）。

如果用户使用 iptables 命令配置了网络过滤规则，则会执行网络过滤子系统的回调函数，如没有配置，就跳过数据包过滤接着执行 do_something_finish。

网络过滤子系统回调函数的调用格式为：

```
   NF_HOOK(PROTOCOL,HOOK_POSITION_IN_THE_STACK,SKB_BUFFER,IN_DEVICE,OUT_DEVICE,
o_something_finish)
```

NF_HOOK 的输出值可以是以下几种。

- do_something_finish 函数的输出：如果数据包通过安全检查，最终将执行到该函数。
- -EPERM ：表示经过网络过滤扔掉了数据包 SKB_BUFF。
- -ENOMEM：表示执行网络过滤时内存出错。

7.2.3 与路由子系统的交互

数据包在 IP 层的处理过程中有多处都需要与路由子系统交互，例如接收到的数据包需继续前送时，以及发送数据包时，IP 层处理函数首先需要获取将数据包送达目标地址的发送路径，这些信息就需要从路由子系统获取。在第 12 章中我们会讲解路由子系统的具体实现，这里主要介绍三个在 IP 层需要用到的查询路由表的 API。

- ip_route_input：确定网络输入数据包的目标地址，数据包可能由本机接收、前送或扔掉。
- ip_route_output_flow：在发送一个数据包之前使用，该函数返回数据包发送路径中下一跳的网关 IP 地址和发送数据包的网络设备。
- dst_pmtu：该函数的输入参数为路由表的入口，返回数据包发送路径上的最大传输单元 PMTU。

在 IP 层查询路由表时，路由表做出路由决策的主要依据是以下各数据域：

- IP 数据包的目标地址。
- IP 数据包的源地址。
- 服务类型（ToS）。
- 接收数据包的网络设备。
- 可用于发送数据包的网络设备列表。

路由子系统将以上各数据域的值作为条件，按照事先配置的路由策略选择路由，并返回给调用程序。

上述路由子系统的函数将从路由表查询得出的结果存放在 skb->dst 数据域中。如果对路由表的查询不成功，函数将返回错误代码。

路由子系统创建了一个路由高速缓冲区（路由 Cache），来缓存最近使用过的路由信息，以便以后做路由决策时可快速获取路由信息。

7.3 输入数据包在 IP 层的处理

在 6.4 节中我们介绍了数据链路层与网络层之间接收数据包的接口，Linux 内核定义了 ptype_base 链表来实现接口，网络层各协议将自己的接收数据包处理函数注册到 ptype_base

链表中，数据链路层按接收数据包的 skb->protocol 值在 ptype_base 链表中找到匹配的协议实例，将数据包传给注册的处理函数。IP 协议在 PF_INET 协议族初始化函数 inet_init 中调用 dev_add_pack 注册的处理函数是 ip_rcv。ip_rcv 函数是网络层 IP 协议处理输入数据包的入口函数，IP 协议通过 ip_rcv、ip_rcv_finish 等函数，完成 IP 层对输入数据包的处理。接下来我们将从 ip_rcv 函数开始分析 IP 数据包在 Linux 内核网络层的处理流程。

ip_rcv 是典型的两阶段实现的函数。该函数的功能主要是对数据包作正确性检查，然后调用网络过滤子系统的回调函数对数据包进行安全过滤，而对数据包的实际处理功能大部分在 ip_rcv_finish 函数中实现。

7.3.1　ip_rcv 函数分析

阅读 ip_rcv 的源代码，你可以看到都是在对 skb 中的数据包做各种合法性检查：协议头长度、协议版本、数据包长度、校验和等。传给 ip_rcv 函数处理的数据包存放在 skb 中，ip_rcv 函数需要从 skb 管理结构中获取各种信息，作为处理的依据，并将数据链路层与网络层的处理关联在一起。

1．当前 skb 各数据域的状态

ip_rcv 函数由数据链路层的接收函数 netif_receive_skb 调用执行，在 netif_receive_skb 函数中，已经将 Socket Buffer 中 skb 的网络层头信息地址指针 skb->network_header 设置到网络层（IP 层）协议头信息的起始地址，这时 IP 层的处理函数可以直接将 IP 的协议头信息从 Socket Buffer 数据缓冲区中复制到 iphdr 数据结构中。

在调用 ip_rcv 函数之前，skb 数据结构中的大部分数据域已经初始化了（在从网络接口以中断的方式通知内核接收到了数据包到调用网络层的处理函数期间完成）。图 7-5 给出了开始执行 ip_rcv 时 skb 一些数据域的值。注意，skb->data 是指向整个负载数据起始地址的指针，这时数据链路层的处理已结束，数据链路层的头信息已不再重要，skb->data 指向了 IP 层的协议头信息的起始地址。

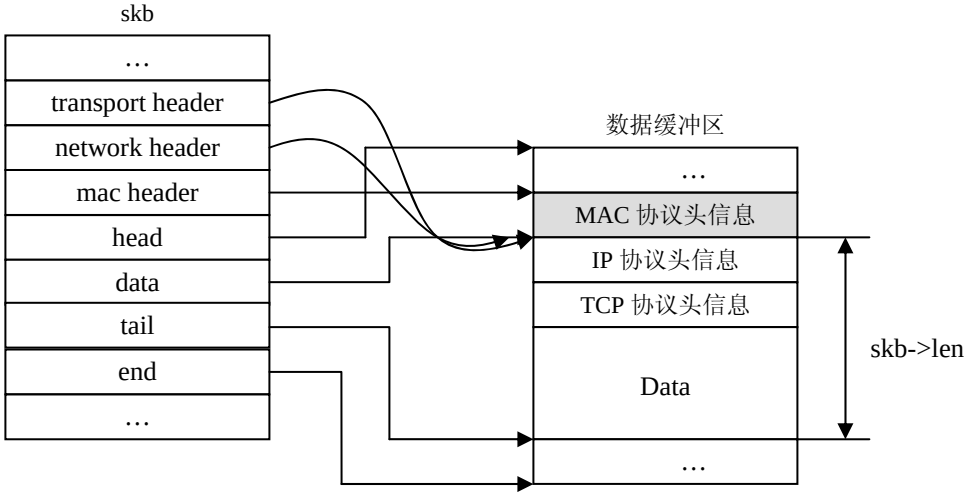


图 7-5　sk_buff 各数据域的值

2．skb->pkt_type 数据域的值

如果接收到的数据包的 MAC 地址不是本机网络设备的 MAC 地址，数据包的类型 skb->pkt_type 的值会设为 PACKET_OTHERHOST，在正常工作情况下网络设备会扔掉这些数据包。除非网络设备的工作模式是混杂模式，它会不管数据包的 MAC 地址是否与本机 MAC 地址相同，接收所有的网络数据包，并将数据包上传给 TCP/IP 协议栈。但 ip_rcv 函数并不关心其他网络地址的数据包，所以这时直接扔掉 skb->pkt_type 为 PACKET_OTHERHOST 的数据包。

注意，收到别的主机数据包（L2 地址不是本机地址）与将数据包按路由前送给其他主机的情形不一样。后者数据包的 MAC 地址是本机，但 IP 地址不是本机，如果设备配置成路由器，它首先会接收这样的数据包，然后再前送。

```
   int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt, struct net_device
*orig_dev)                                        //net/ipv4/ip_input.c
   {
     struct iphdr *iph;
     u32 len;

   /*数据包的 MAC 地址不是本机网络设备的 MAC 地址，扔掉数据包*/

     if (skb->pkt_type == PACKET_OTHERHOST)
          goto drop
```

3．数据包共享的处理

skb_share_check 查看数据包的引用计数是否大于 1，如果是，说明内核中有其他进程也在使用这个数据包。ip_rcv 是由 netif_receive_skb 函数调用的，在调用网络层 IP 协议的处理函数前，netif_receive_skb 会对 Socket Buffer 的引用计数加 1，IP 协议处理函数查看到数据包的引用计数大于 1 时，会创建一个数据包的拷贝，这样 IP 协议处理函数就可以任意修改数据包的值，而 netif_receive_skb 接下来调用网络层其他协议处理函数收到的就是原始数据包。

```
   /*查看数据包引用计数是否大于 1，如果是就创建一个 skb 的拷贝*/
   if ((skb = skb_share_check(skb, GFP_ATOMIC)) == NULL) {
          IP_INC_STATS_BH(dev_net(dev), IPSTATS_MIB_INDISCARDS);
          goto out;
     }
```

4．IP 协议头信息的正确性检查

（1）判断 IP 协议头长度是否正确

pskb_may_pull 的功能是保证 skb->data 指向的区域至少包含一个数据块，其长度与 IP 协议头信息的长度一致。因为每个 IP 数据包（包括分割成数据片的数据包）必须包含完整的 IP 协议头信息。如果 IP 协议头信息正确，就不做任何操作。如果 pskb_may_pull 操作失败，函数返回错误信息并结束操作，扔掉数据包。如果操作成功，函数必须再次初始化 iphdr，因为 pskb_may_pull 可以改变头信息存放缓冲区的结构。

（2）判断 IP 协议头信息是否正确

接下来对 IP 协议头信息的正确性做检查。首先确定 IP 协议基本的头信息正确，IP 协议基本的头信息长度应该是 20 个字节，协议头信息的存放单位是 32 位（4 字节），如果其

长度值小于 5，则意味着有错。在源代码中 if 语句中的第二个条件非常重要，当前 Linux 内核中 TCP/IP 协议栈实现了两个 IP 协议实例：IPv4 和 IPv6。这里 if 语句的作用是确定数据包是一个 IPv4 的数据包。IPv4 和 IPv6 的数据包是由不同的函数来处理接收操作的，当数据包协议是 IPv6 时，ip_rcv 函数就不会被调用。

接下来重复一些前面做过的检查，但这次用整个 IP 协议头信息的长度来做合法性检查（包括 IP 选项，即 IP 协议头信息中协议头的总长度*4，iph->ihl * 4）。这个检查直到这时才做是因为函数首先需要确保在从协议头中读取需要的数据之前，基本的协议头信息没有被破坏。

在完成以上两项协议头信息的合法性检查后，函数需计算校验和，查看计算结果是否与协议头中保存的原校验和相匹配，如果不匹配，扔掉数据包。校验和的检查由 ip_fast_csum 函数完成。

```
 /*对数据包的 IP 协议头做各种合法性检查：协议头的长度，协议版本，校验和是否正确，如有任何一项出错，扔掉数据包*/
if (!pskb_may_pull(skb, sizeof(struct iphdr)))
  goto inhdr_error;
  iph=ip_hdr(skb);
if (iph->ihl < 5 || iph->version != 4)
  goto inhdr_error;

if (!pskb_may_pull(skb, iph->ihl*4))
  goto inhdr_error;

iph = ip_hdr(skb);

if (unlikely(ip_fast_csum((u8 *)iph, iph->ihl)))
  goto inhdr_error;
```

5．数据包正确性检查

当校验和通过正确性检查后，ip_rcv 函数还要对负载数据本身做正确性检查，其中包括两方面的操作：

- 确定缓冲区的长度（接收数据包）大于或等于 IP 协议头信息中报告的数据包的长度。
- 确定整个数据包的长度至少不小于 IP 协议头的长度，即数据包中至少包含了 IP 协议头数据。

为什么要做这两项检查？数据链路层的协议（假设是以太网协议）可能会在负载数据后加补丁，以保证发送的数据包长度不小于数据链路层协议允许的最小数据包长度，这样在 Socket Buffer 中的数据包长度（由 skb->len 数据域给出）可能大于实际的数据包长度（由 IP 协议头信息 len = ntohs(iph->tot_len)给出）。

第二个检查是因为 IP 协议头不能被分割，如果数据包被分割成小的数据片，每个 IP 数据片至少包含一个 IP 协议头（iph->ihl*4）。这个检查极少失败，如果失败，意味着校验和的计算是基于一个已被损坏了的数据包得出的，而正好产生的校验和的结果与原数据包产生的校验和值一样。

```
//对数据包的长度做正确性检查，如果发现数据包长度有任何不合理，扔掉数据包
len = ntohs(iph->tot_len);
if (skb->len < len) {
IP_INC_STATS_BH(dev_net(dev), IPSTATS_MIB_INTRUNCATEDPKTS);
```

```
goto drop;
} else if (len < (iph->ihl*4))
  goto inhdr_error;
```

6．清理工作，获取一个干净的数据包

现在我们知道数据包已到达 IP 层，不再需要数据链路层给数据包尾所加的补丁，可以把 Socket Buffer 中的数据缓冲区裁减成实际大小（由 pskb_trim_rcsum(skb,len)完成）。接着清空 Socket Buffer 中的控制缓冲区，以备 IP 层使用。

```
/*去掉网络传输介质在数据包中加的填充字节*/
 if (pskb_trim_rcsum(skb, len)) {
IP_INC_STATS_BH(dev_net(dev), IPSTATS_MIB_INDISCARDS);
goto drop;
}

/*清空 skb 中的控制缓冲区 skb->cb，供 IP 层处理函数使用*/

memset(IPCB(skb), 0, sizeof(struct inet_skb_parm));

/*最后调用网络过滤子系统的回调函数，对数据包进行安全过滤*/

return NF_HOOK(PF_INET, NF_INET_PRE_ROUTING, skb, dev, NULL,
              ip_rcv_finish);
```

7．函数结束操作

这时到达了函数的结束之处，注意在这里还没有决定或选择任何处理例程，ip_rcv 在调用网络过滤子系统的回调函数后结束：

```
return NF_HOOK(PF_INET, NF_INET_PRE_ROUTING, skb, dev, NULL,
              ip_rcv_finish);
```

对于以上语句可以这样来理解： skb 是由 dev 收到的数据包，网络过滤子系统查看该数据包是否允许继续处理，是否需要对数据包做修改；这由网络协议栈的 NF_INET_PRE_ROUTING 来决定（这时数据包还没被扔掉）。通过检查后如果数据包没被扔掉，则继续执行 ip_rcv_finish。

从以上分析中我们可以看到，在函数 ip_rcv 中对数据包做了各方面的合法性检查，这些检查看起来似乎有点多余，实际上所有这些正确性检查对保证系统的稳定性至关重要。TCP/IP 协议栈是 Linux 内核的组件，这个时候如果 sk_buff 结构的数据域初始化不正确，或 IP 协议头信息有损坏，Linux 内核就会按错误的方式处理数据包，或访问到无效的内存区域，造成系统崩溃。

7.3.2 ip_rcv_finish 函数分析

除了正确性检查，ip_rcv 函数没有对数据包做更多的操作，主要的处理过程都是由 ip_rcv_finish 函数完成的，这些过程包括：

- 确定数据包是前送还是在本机协议栈中上传，如果是前送，需要确定输出网络设备和下一个接收站点的地址。
- 解析和处理部分 IP 选项。

1．获取路由信息

要发送数据包，首先需要知道向什么地方传，这个信息可以从以下两处获取。

- skb->dst：该数据域中可能已包含了数据包如何到达目的地址的路由信息。
- 路由子系统的 ip_route_input 函数：如果路由子系统返回的结果说明数据包发送的目的地址不可到达，就扔掉数据包。

ip_route_input 函数完成路由决策的依据是数据包的源 IP 地址、目标 IP 地址、服务类型以及接收数据包的网络设备。

```
static int ip_rcv_finish(struct sk_buff *skb)
{
const struct iphdr *iph = ip_hdr(skb);  //获取 IP 协议头信息
struct rtable *rt;                      //路由表结构变量

/*获取数据包传递的路由信息，如果 skb->dst 数据域为空，就通过路由子系统获取，如果 ip_route_input 返回错误信息，
表明数据包目标地址不正确，扔掉数据包*/

if ( skb_dst( skb ) == NULL) {
int err = ip_route_input(skb, iph->daddr, iph->saddr, iph->tos, skb->dev);
if (unlikely(err)) {
  if (err == -EHOSTUNREACH)
IP_INC_STATS_BH(dev_net(skb->dev),  IPSTATS_MIB_INADDRERRORS);
  else if (err == -ENETUNREACH)
IP_INC_STATS_BH(dev_net(skb->dev),IPSTATS_MIB_INNOROUTES);
              goto drop;
        }
  }
```

2．更新由流量控制系统（QoS）使用的统计信息

```
/*如果在配置内核时配置了流量控制功能，则更新 QoS 的统计信息*/
#ifdef CONFIG_NET_CLS_ROUTE 工
  if (unlikely( skb_dst(skb)->tclassid )) {
        struct ip_rt_acct *st = per_cpu_ptr(ip_rt_acct, smp_processor_id());
        u32 idx = skb_dst(skb)->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
```

3．处理 IP 选项

如果 IP 协议头的长度大于 20 个字节（5×4，32 位），说明 IP 协议头中包含 IP 选项信息需要处理。IP 选项由函数 ip_rcv_options(skb)来处理。如果这个 Socket Buffer 有别的进程在共享，首先调用 skb_cow（Copy on Write）生成一个 Socket Buffer 的 skb 头的拷贝，因为处理 IP 协议选项可能会修改 IP 协议头信息。

```
/*如果 IP 协议头的长度大于 20 个字节，说明有 IP 选项，处理选项*/
if (iph->ihl>5&& ip_rcv_options(skb))
        goto drop;

/*统计收到的各类数据包数：组传送、广播数据包*/

  rt = skb_rtable( skb );
```

```
    if (rt->rt_type==RTN_MULTICAST)
          IP_INC_STATS_BH(dev_net(rt->u.dst.dev), IPSTATS_MIB_INMCASTPKTS);
    else if (rt->rt_type==RTN_BROADCAST)
          IP_INC_STATS_BH(dev_net(rt->u.dst.dev), IPSTATS_MIB_INBCASTPKTS);

  /*将负责数据包传送的 skb->dst->input 函数指针指向的处理函数*/

    return dst_input(skb);
  drop:
    kfree_skb(skb);
    return NET_RX_DROP;
  }
```

4．函数结束时的处理

ip_rcv_finish 结束于对 dst_input 函数的调用，此函数用于确定下一步对数据包的处理函数是哪一个，实际的调用存放在 skb->dst->input 数据域，根据数据包的目标地址，skb->dst->input 设置成 ip_local_deliver 或 ip_forward。

到此，对 dst->input 的调用就完成了 IP 层对输入数据包的处理。

7.3.3 接收操作中 IP 选项的处理

如果接收到的数据包的 IP 协议头设置了 IP 选项，就要对部分 IP 选项做处理。Linux 内核中 IP 选项的格式、定义和它们在内核中的实现将在下一节介绍。在 IP 协议接收函数中要处理的主要是：如果应用程序设置了 IP 选项中的源路由选项，即给出了数据包发送路径上各节点的 IP 地址列表，需要把下一跳的 IP 地址设置给 skb->dst。

1．处理 IP 选项

处理 IP 选项这个任务由函数 ip-rcv-options 来完成。要处理 IP 选项信息，首先需要将 IP 选项从数据包中读出，放入 skb 的控制缓冲区，再按照 IP 选项的定义格式分析 IP 选项各数据域的含义。如果该数据包有别的进程共享，则产生一个 skb 头的拷贝，因为在处理 IP 选项的时候可能会对 skb 头的数据做修改。

```
  static inline int ip_rcv_options(struct sk_buff *skb)
  {
    struct ip_options *opt;
    struct iphdr *iph;
    struct net_device *dev = skb->dev;

    // 如果 Socket Buffer 由多个进程共享，则先产生一个 skb 头的拷贝，若产生拷贝不成功，则将数据包扔掉

    if (skb_cow(skb, skb_headroom(skb))) {
          IP_INC_STATS_BH(dev_net(dev), IPSTATS_MIB_INDISCARDS);
          goto drop;
    }

    //从头信息中获取 IP 选项，存放在 skb 的控制缓冲区中，控制缓冲区用 IPCB 宏访问

    iph = ip_hdr(skb);
    opt = &(IPCB(skb)->opt);
    opt->optlen = iph->ihl*4 - sizeof(struct iphdr);

    //解析 IP 选项，如果 IP 选项中设置了源路由选项，将路由信息设置给 skb->dst
```

```
if(ip_options_compile(dev_net(dev), opt, skb)){ IP_INC_STATS_BH(dev_net(dev), IPSTATS_MIB_INHDRERRORS);
  goto drop;
  }
if (unlikely(opt->srr)) {           //设置了源路由选项
  struct in_device *in_dev = in_dev_get(dev);
  if (in_dev) {
    if (!IN_DEV_SOURCE_ROUTE(in_dev)){
```

2．解析 IP 选项

解析 IP 选项的任务由 ip_options_compile 函数完成。后面会介绍 ip_options_compile 的实现细节，现在我们只关心该函数的输出。我们知道在 skb 结构中有一个控制缓冲区 skb->cb，用于存放当前处理 skb 例程的私有数据结构，这时 IP 层用 skb->cb 来存放解析好的 IP 选项和一些别的信息（如数据包分片相关信息），在控制缓冲区 skb->cb 中存放的解析完成的 IP 头选项以 inet_skb_parm 数据结构来管理，当要访问 skb->cb 中的数据时需使用宏 IPCB。

```
struct inet_skb_parm{               //include/net/ip.h
  struct                  ip_options opt;
  unsigned char    flag;
  #define IPSKB_FORWARDED | _XFRAM_TUNNEL
}

//解析 IP 选项，如果 IP 选项中设置了源路由选项，将路由信息设置给 skb->dst

if(ip_options_compile(dev_net(dev), opt, skb)){ IP_INC_STATS_BH(dev_net(dev), IPSTATS_MIB_INHDRERRORS);
  goto drop;
  }
if (unlikely(opt->srr)) {           //设置了源路由选项
  struct in_device *in_dev = in_dev_get(dev);
  if (in_dev) {
    if (!IN_DEV_SOURCE_ROUTE(in_dev)){
```

3．选项的错误处理

如果选项中有任何错误，则扔掉数据包。这时主机会向数据包的发送者传回一个特定的 ICMP 消息，通知数据包发送者：发送出了问题。在 ICMP 消息中会指出错误在 IP 协议头信息的何处出现，ICMP 消息可以帮助数据包发送者找出问题原因。

注意： ip_options_compile 只检查 IP 选项是否正确，并将其存放在 ip_options 数据结构中，由 skb->cb 指针指向其地址，ip_options_compile 函数并不对选项本身做处理。

```
//如果系统禁止使用源路由选项，扔掉数据包
if (IN_DEV_LOG_MARTIANS(in_dev) && net_ratelimit())
printk(KERN_INFO "source route option %pI4 -> %pI4\n",
       &iph->saddr, &iph->daddr);in_dev_put(in_dev);
goto drop;
}
in_dev_put(in_dev);
}
if (ip_options_rcv_srr(skb))
  goto drop;
}
```

4．源路由选项处理

数据包的 IP 选项中如果设置了源路由选项，Linux 内核需要检查网络设备的配置是否允许使用这些选项。如果没有特别配置，默认情况下系统是允许使用源路由选项的。相反，如果数据包的 IP 选项中设置了使用源路由，而系统设置中该选项没打开，就扔掉数据包（这时不会产生 ICMP 消息）。

如果系统设置中允许使用源路由，调用 ip_options_rcv_srr 函数将路由设置在 skb->dst 中，并确定用哪个设备将数据包发送到源路由列表中指定的下一跳，通常要求下一站点是另一主机，则函数只设置 opt->srr.is_hit，指明发现了地址。

ip_options_rcv_srr 在设置路由时需要考虑这种情况：下一个站点是本地主机的接口，如果是这样，该函数就将 IP 地址写入 IP 头的目标 IP 地址域，再到源路由列表中去查看；如果列表中有这个地址，ip_options_rcv_srr 继续展开，在 IP 协议头选项中的源路由列表中找下一个站点的地址，直到找到一个 IP 地址不是本地主机的 IP 地址为止。

常规情况下在源路由列表中不应有多于一个的 IP 地址为本地主机 IP 地址，但是有多于一个本地主机 IP 地址也是合法的。当最后找到了不是本地主机 IP 地址的目标时，设置 IP 选项数据结构中的 srr_is_hit 标志域，表示数据包还没到达它的最后地址，需继续前送。如果数据包将继续前送，srr_is_hit 的初始值将告诉 ip_forward_options 通过在 IP 协议头中插入需要的数据来处理源路由选项。

总结起来，这里完成的功能为：

- 数据包的正确性和安全性检查，保证接收到的数据包是一个干净的数据包。
- 确定数据包的发送路由。
- 解析 IP 选项。
- 根据路由将数据包传给处理函数。

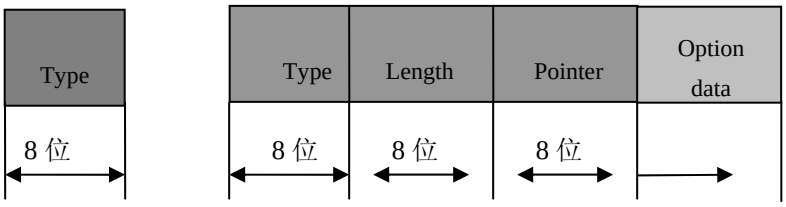
7.4 IP 选项

数据包的处理过程：上传、前送、发送，都涉及 IP 选项的分析和处理，这里我们先对 IP 选项的定义和解析做一番介绍。

7.4.1 IP 选项的格式

在 IP 协议头中除了常规的协议头信息外，还可以包含 IP 选项（0~40 个字节）。Linux 的 TCP/IP 协议栈中实现了一系列的 IP 选项，供应用程序使用。应用程序通过 IP 选项可以跟踪 IP 数据包的发送过程。如果在 IP 层驻留了选项，IP 协议头可由原来基本的 20 个字节扩展至最多 60 个字节。

大多数 IP 选项都使用得很少，不同的选项可以组合在同一个数据包中。图 7-6 给出了选项的格式。



（a）单选项格式　　（b）多选项格式

图 7-6

IP 选项可以为单字节，也可以为多字节，它包含了以下几个基本的数据域。

1．Type 数据域

Type 数据域描述的是当前 IP 数据包的协议头信息中设置了什么 IP 选项，以及在对数据包进行分片时如何处理该选项。

Type 数据域是一个 8 位的数据域，它划分成 3 个子域，如图 7-7 所示。

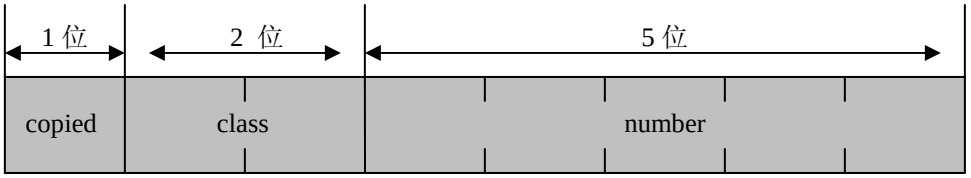


图 7-7　IP 选项中 Type 域的格式

- copied：如果设置了 copied 域，IP 层在对数据包分片时，必须将选项复制到每个分片的数据段中。
- class：class 域将 IP 选项按 4 个特性分类，可用于按 IP 选项过滤数据，或对这些数据包使用不同 QoS 参数。
- number：选项类型的编码。为了便于理解和记忆，Linux 内核按选项编码含义定义了编码描述符。表 7-1 给出了 Linux 内核中使用的与 number 的对应选项描述符，以及 3 个子域的含义。

表 7-1　IP 选项 Type 的子数据域 number 的编码和符号定义

选项描述	在内核源码中使用的符号	number 域的值	copied 域	class control(00)/reserved(01)/ measurement(10)/reserved(11)
选项结束（End of Options List）	IPOPT_END	0	0	control
无选项操作（No Operation）	IPOPT_NOOP	1	0	control
安全选项（Security）	IPOPT_SEC	2	1	control
松散原路由并记录路由（Loose Source and Record Route）	IPOPT_LSRR	3	1	control
记录时间戳（Timestamp）	IPOPT_TIMESTAMP	4	0	measurement
记录路由（Record Route）	IPOPT_RR	7	0	control
Stream ID	IPOPT_SID	8	1	control
严格源路由并记录路由（Strict Source and Record Route）	IPOPT_SSRR			
路由报警（Router Alert）	IPOPT_RA			

在 Linux 内核源码的 include/linux/ip.h 文件中，可以看到选项 Type 的定义和操作其子域的宏定义。

```
/* 以下为 IP 选项相关定义 */
#define IPOPT_COPY              0x80
#define IPOPT_CLASS_MASK        0x60
#define IPOPT_NUMBER_MASK       0x1f
/*操作 copy, class, number 域的宏*/
#define IPOPT_COPIED(o)         ((o)&IPOPT_COPY)
#define IPOPT_CLASS(o)          ((o)&IPOPT_CLASS_MASK)
#define IPOPT_NUMBER(o)         ((o)&IPOPT_NUMBER_MASK)
#define IPOPT_CONTROL           0x00
#define IPOPT_RESERVED1         0x20
#define IPOPT_MEASUREMENT       0x40
#define IPOPT_RESERVED2         0x60
/* Linux 内核中使用的 IP 选项的符号*/
#define IPOPT_END               (0 | IPOPT_CONTROL)
#define IPOPT_NOOP       (1 | IPOPT_CONTROL)
…
```

图 7-6（b）描述的是 IP 多字节选项，其中 Type 数据域与单字节选项格式中的含义相同，其他数据域如下。

- Length：选项的长度（以字节为单位），包括 Type 和 Length 在内。
- Pointer：从选项的起始地址开始的偏移量，描述选项在何处结束，在主机解析选项时使用。
- Option data： 用于存放选项需要的或需添加的任何数据。

2．End of Options 和 No Operation

没有 IP 选项的 IP 头信息的长度是 20 个字节，当 IP 选项的长度不是 4 个字节的整数倍时，发送方用 End of Options 选项在 IP 协议头的结尾处补充相应的字节，使其在 4 字节边界对齐。

No Operation 用于在选项之间加补充字节，主要是让各 IP 选项序列在某个字节边界对齐。

3．Source Route

源路由选项（Source Route）允许发送方指定数据包在发送途径上经过的站点 IP 地址。源路由可以在数据链路层（L2）和网络层（L3）指定，我们将讨论的是源路由在 L3 的实现。

源路由是一个多字节选项，它按顺序给出数据包在发送过程中各站点的 IP 地址列表。如果列表中一个路由器下线了，通常上层协议会计算一个新的路由，而设置了源路由选项的数据包不能利用动态路由来重发数据包。这往往是出于安全控制的考虑，所以不允许使用新路由。

源路由可以有两种类型，严格（strict）源路由和松散（loose）源路由。在严格源路由中，发送方必须列出沿途每个路由器的 IP 地址，在发送过程中不能改变。而在松散源路由中， 当从一个路由器到另一个路由器之间有多个选择时，可使用另一个路由，不是必须使用列表中的路由器，但必须按发送方指定的顺序来发送。

图 7-8 给出了使用源路由的一个例子。如果主机 A 要传一个数据包给主机 B，使用严格源路由选项时，主机 A 必须指定所有中间路由器的 IP 地址。在图 7-8 中严格源路由应该为：数据包从 A 经 Router_1、Router_2、Router_3 到主机 B。当使用松散源路由时只需给

出 Router_1、Router_3 的 IP 地址就可以了。非相邻两个路由器（Router_1、Router_3）之间的路由器可用 Router_2 也可以用 Router_2b（如果 Router_2 下线了），反之，也一样。

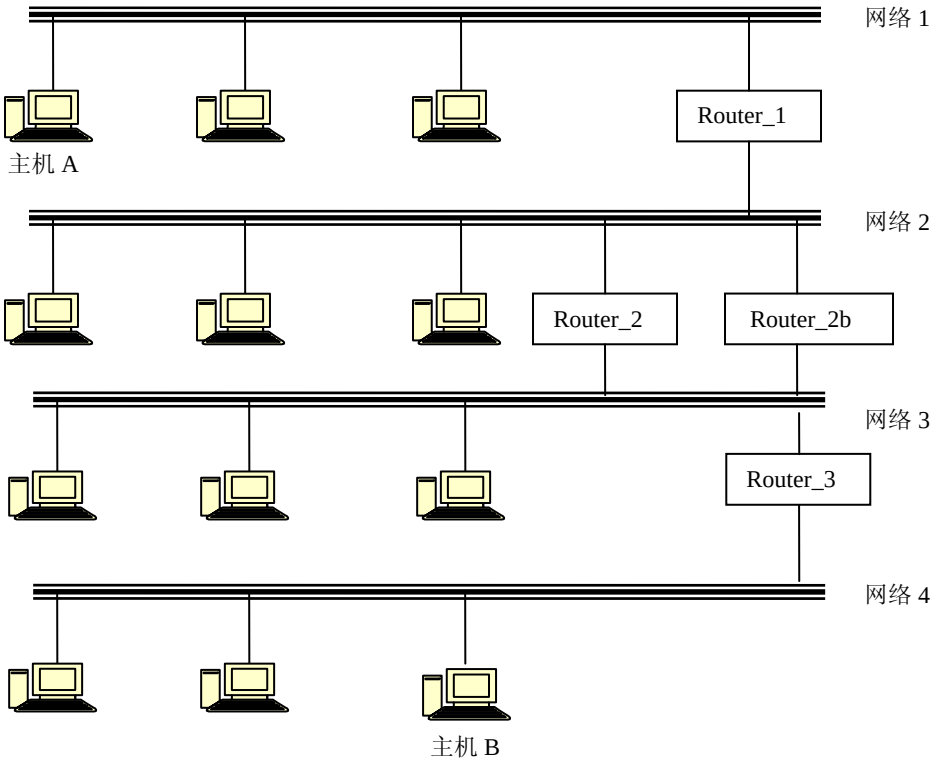


图 7-8　IP 源路由选项示例

4．Record Route

使用这个选项的目的，是让从源地址到目的地址之间的各路由器在数据包的 IP 头信息中记录自己前送这个数据包的网络接口的 IP 地址。由于 IP 协议头的空间有限，最多可以记录 9 个路由器的 IP 地址（有可能更少，视 IP 头中有多少选项而定）。这样数据包从源地址到目的地址的前 9 个路由器网络接口的 IP 地址可以存放在 IP 协议头的选项中。以后经过的路由器的 IP 地址，数据包接收方就无法获取了。这个选项会使数据包的头信息不断增长，同时也意味着数据包的长度也在增长。考虑到 IP 协议头中可能还会有别的选项，发送方用 Record Route 选项时，通常要在 IP 协议头中为存放沿途的路由器 IP 地址预留空间，如果预留的空间被填满了，即使 IP 协议头中还有空间，其后的地址也不会再写入 IP 协议头，也不会有错误信息返回给发送方。

图 7-9 给出了 IP 协议头中选项的部分变化。每个路由器都填写自己的地址，同时也会修改 Pointer 域，指明选项中数据的结束位置。

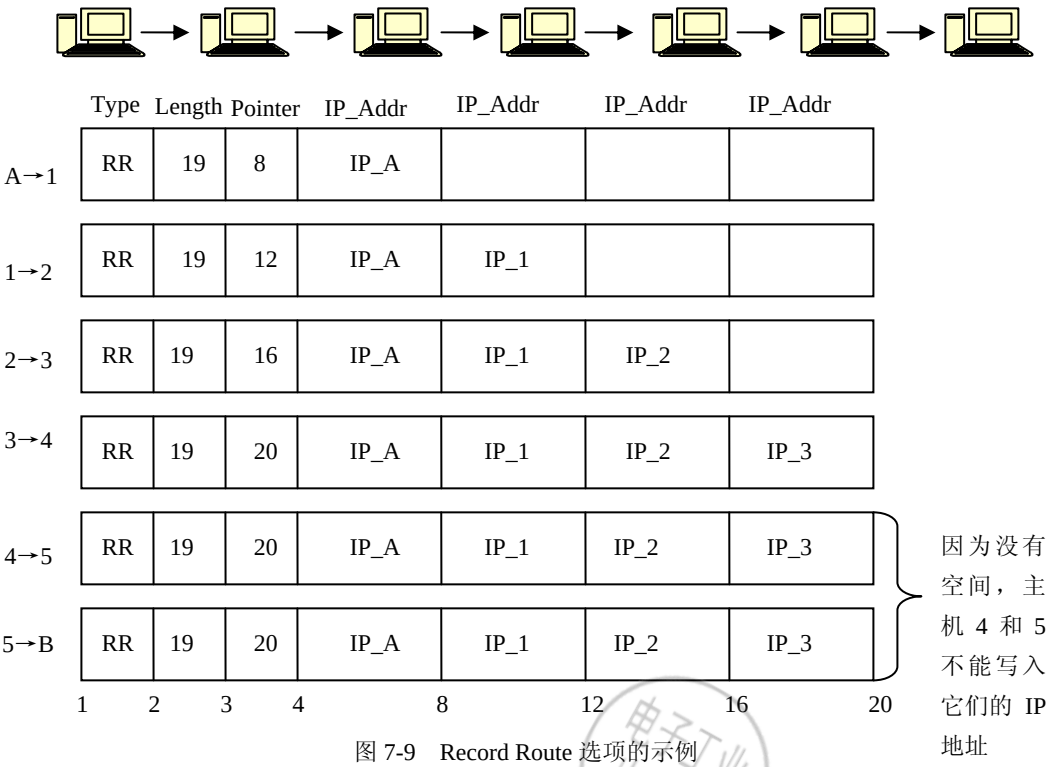


图 7-9　Record Route 选项的示例

5．时间戳 Timestamp 选项

时间戳（Timestamp）选项是所有 IP 选项中最复杂的一个，它中间还包含了子选项。另外，管理时间戳选项的方式与记录路由（Record Route）选项的方式不同，处理记录路由选项时，如果 IP 协议头中已无空间记录路由器的 IP 地址，则不再向 IP 协议头中写入任何信息，也不对这种情况做任何处理；在没有空间记录数据包达到路由器的时间时，时间戳选项要做溢出处理，它需要在它的数据域中加入额外字节管理两个概念：其一，如何记录时间，其二，管理溢出。时间戳选项的格式如图 7-10 所示。

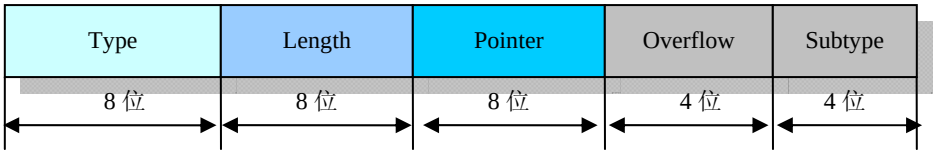


图 7-10　IP 时间戳选项的格式

前 3 个字节与其他选项的含义相同：Type、Length、Pointer，第 4 个字节划分成了两个部分 Overflow 和 Subtype。

Subtype 是时间戳选项的子编码，描述了如何在 IP 协议头的选项空间中记录数据包达到路由器的时间。Subtype 的值包括以下几种。

- RECORD TIMESTAMPS：路由器在 IP 头中记录其收到数据包的时间。

- RECORD ADDRESSES AND TIMESTAMPS：路由器记录数据包到达时间和接收数据包的网络接口 IP 地址。
- RECORD TIMESTAMPS ONLY AT THE PRESPECIFIED SYSTEMS：只有发送方指定的 IP 地址的路由器才记录它们收到数据包的时间。

在以上 3 种情况中，时间都是以 ms 表示的。

Overflow 是溢出域，如果 IP 协议头中已无空间让路由器写入它们收到数据包的时间，时间戳选项每次对 Overflow 加 1。Overflow 本身只有 4 位，这样它最多只能记录 15 次溢出，在现代网络中 Overflow 域自身就很容易溢出。当 Overflow 数据域溢出时，路由器需给数据包原始发送者返回一个 ICMP 参数错的信息。

Subtype 的前两个子选项比较类似(所不同的只是每个节点存什么信息到 IP 协议头中)，第三个子选项有所不同，数据包的发送方对其关心的 IP 地址进行列表，并记录数据包到达该 IP 地址列表中的路由器的时间值。图 7-11 给出了处理时间戳选项的一个示例，图的上端有下画线的主机是要写入收到时间到数据包的 IP 协议头选项中的主机。

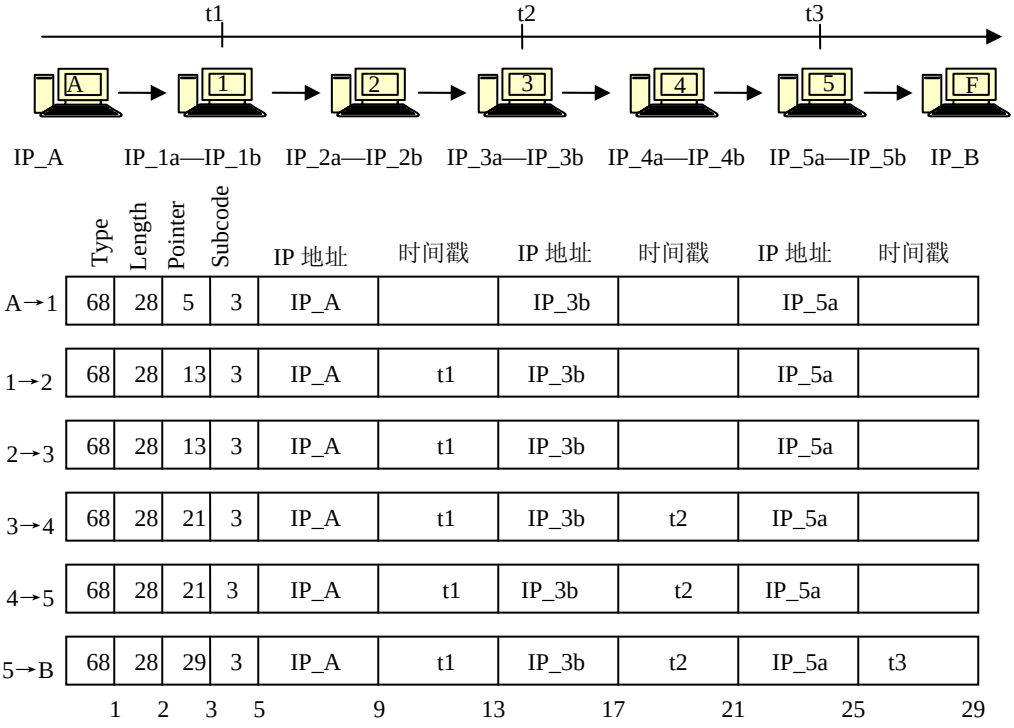


图 7-11　IP 时间戳选项处理示例

6．Router Alert

这个选项标志着数据需要特殊的处理而不是只查看数据包的目的地址就前送。使用该选项是为了告诉路由器需要以特殊的方式处理数据包，到目前为止最后两个字节只有一个分配了值：0，表示让路由器查看数据包。当该选项为其他值时，将被忽略并扔掉数据包，然后向数据包的发送方传一个 ICMP 错误信息。

7.4.2 描述 IP 选项的数据结构

发送/前送数据包的 IP 选项经解析信息存放在 ip_options 类型的变量中，IP 选项存放在这个数据结构比全部存放在 IP 协议头信息中更容易读取和分析。ip_options 数据结构的定义在 include/net/inet_sock.h 文件中。

```
struct  ip_options {                        include/net/inet_sock.h
  __be32                faddr;                  // 第一个结点的 IP 地址
  _unsigned char        optlen;
  unsigned char         srr;                    //源路由选项
  unsigned char         rr;                     //记录路由选项
  unsigned char         ts;                     //时间戳选项
  unsigned char         is_strictroute : 1,     //严格源路由
                srr_is_hit : 1,
                is_changed: 1,                  //协议头信息是否被修改
                rr_needaddr:1,
                ts_needtime: 1,
                ts_needaddr : 1;
  unsigned char    router_alert;
  unsigned char    cipso;
  unsigned char    __pad2;
  unsigned char    __data[0];
};
```

ip_options 数据结构中存放了需要向外发送和前送数据包的 IP 选项，下面介绍各数据域的基本含义。

- Faddr：只针对向外发送，并设置了源路由选项的数据包有意义，faddr 存放源路由列表中的第一个 IP 地址。
- Optlen：这个数据域表示设置了的 IP 选项的总长度，它的最大长度不能超过 40 个字节。
- Srr：指明源路由选项存放在协议头中的偏移量。
- Rr：如果 rr 的值非 0，IP 选项中设置了记录数据经过路由的 IP 地址选项，rr 的值代表路由器应在 IP 协议头何处记录路由选项的偏移量。
- Ts：如果 ts 为非 0 值，表示 IP 选项中设置了时间戳选项，而 ts 代表了时间戳选项记录在 IP 协议头中位置的偏移量。

接下来的数据域为一个标志位数据域，各标志位在该数据域中占 1 位标志 IP 选项的设备状态及应对选项或数据包做的处理。

- is_strictroute ：该位为 TRUE 时，说明 IP 选项中源路由选项设置的是严格源路由。
- srr_is_hit ：该位为 TRUE 时，说明数据包中设置了源路由选项。在为发送数据包做路由决策时，通过 skb->dst 或路由表获取的下一跳 IP 地址与路由列表中下一跳 IP 地址一致称为 IP 地址命中（hit）。
- is_changed：若 IP 协议头发生变化，则设置该位。IP 协议头是否发生变化决定了是否需要重新计算 IP 校验和。
- rr_needaddr：当 IP 选项中设置了记录路由选项时，如果 rr_needaddr 的值为 1，表明协议头中还有空间记录其他路由信息，这时当前站点应将发送数据包的网络接口 IP 地址复制到协议头中 rr 指定的偏移量处。

- ts_needtime 与 ts_needaddr：这两个标志与时间戳选项相关，分别表明了时间戳选项是否记录数据包到达站点的时间和 IP 地址。
- router_alert：如果 IP 选项中设置了路由报警选项，router_alert 指明路由报警选项在协议头中存放位置的偏移量。
- __data[0]：这个数据域用于将本地主机产生的数据包向外发送情况下，以及本地站点要求回答的 ICMP 请求传回应答数据包时。__data[0]指向存放要加入到数据包协议头中的 IP 选项的地址。
- __pad：此数据域是为使 IP 选项处于 32 位地址边界对齐而加在最后的填充数据。

对 IP 选项的处理散布在接收/前送/发送流程的各处。在 IP 选项从数据包中解析出来后，会设定一系列的标志：如源路由选项是否为严格路由（is_strictroute），时间戳选项是需要记录时间值（ts_needtime）还是需要记录 IP 地址（ts_needaddr）或者都需要记录。等到具体处理数据包发送的函数处，再根据这些标志在 IP 协议头的选项部分来记录信息。

7.4.3　Linux 内核对 IP 选项的处理

处理 IP 选项需花费大量时间，而在实际应用中 IP 选项使用得并不多。如前所述 IP 选项的处理散布在数据包发送途径的各处，Linux 内核实现了一组函数来实现各种状况下对 IP 选项的处理。本节中我们会逐一阐述 IP 选项如何在 Linux 内核中实现。

表 7-2 给出了管理 IP 选项的主要 API，所有 API 都定义在 net/ipv4/ip_options.c 文件中。

表 7-2　管理 IP 选项的 API

函数	功能描述
ip_options_compile	解析 IP 协议头中的选项，初始化 struct ip_options 数据结构的各数据域。该数据结构中包含的标志和指针用于告诉路由子系统，在处理前送的数据包时应在 IP 协议选项的什么位置写入什么信息
ip_options_build	发送本地数据包构建 IP 选项，根据应用设置的选项参数来初始化 struct ip_options 数据结构
ip_options_fragment	IP 数据包分片时，处理分片数据包的选项
ip_forward_options	处理前送数据包的 IP 选项
ip_options_get	输入参数为 IP 选项块，调用 ip_options_compile 来解析参数，用解析结果来初始化 struct ip_options 数据结构
ip_options_echo	创建给数据包发送方，用于返回信息的数据包的 IP 选项

1．IP 选项在数据包分片中的处理

在分割 IP 数据包时，IP 选项由 ip_options_fragment 函数处理。在对 IP 数据包分片时，只有第一个数据片继承了原始数据包的所有选项，所以它的协议头信息会等于或大于其他的分片数据。Linux 简化了这个规则，让所有分片数据的头信息大小一样，Linux 的这种实现方式使对数据包的分割过程更简单有效。首先将原始数据包中的所有选项复制到各数据片中，随后某些数据片中不需要的选项（它们的 IPOPT_COPY 没有设置）用空选项（IPOPT_NOOP）来覆盖，并清除它们在 struct ip_options 数据结构中的标志。发送时第一个分片数据传出之后，Linux 内核会调用 ip_options_fragment 来修改 IP 协议头，为后面的分片数据包写新的头信息。

2．IP 选项在数据包前送中的处理

前送数据包时，有的 IP 选项需要在 IP 协议头的选项部分写入新的数据（如时间戳选项）。ip_options_compile 函数解析原始 IP 选项，并初始化 struct ip_options 数据结构中一系列标志来表明解析结果，随后的修改在 ip_forward_options 函数中完成。

3．IP 选项的来源

ip_options_get 函数接收选项数据块作为输入参数，调用 ip_options_compile 函数来解析它们，将解析的结果存放在它分配的 struct ip_options 数据结构中。它可以从内核地址空间接收选项，也可以从用户地址空间接收，它有一个参数指明选项的来源。例如可以使用 ip_setsockopt 函数（由传输层的 TCP/UDP 协议使用）在某个套接字上设置 IP 选项。

接下来将介绍以上函数在实际中是如何应用的。图 7-12 给出了以上处理 IP 选项的 API 在数据包处理的各阶段的调用位置。

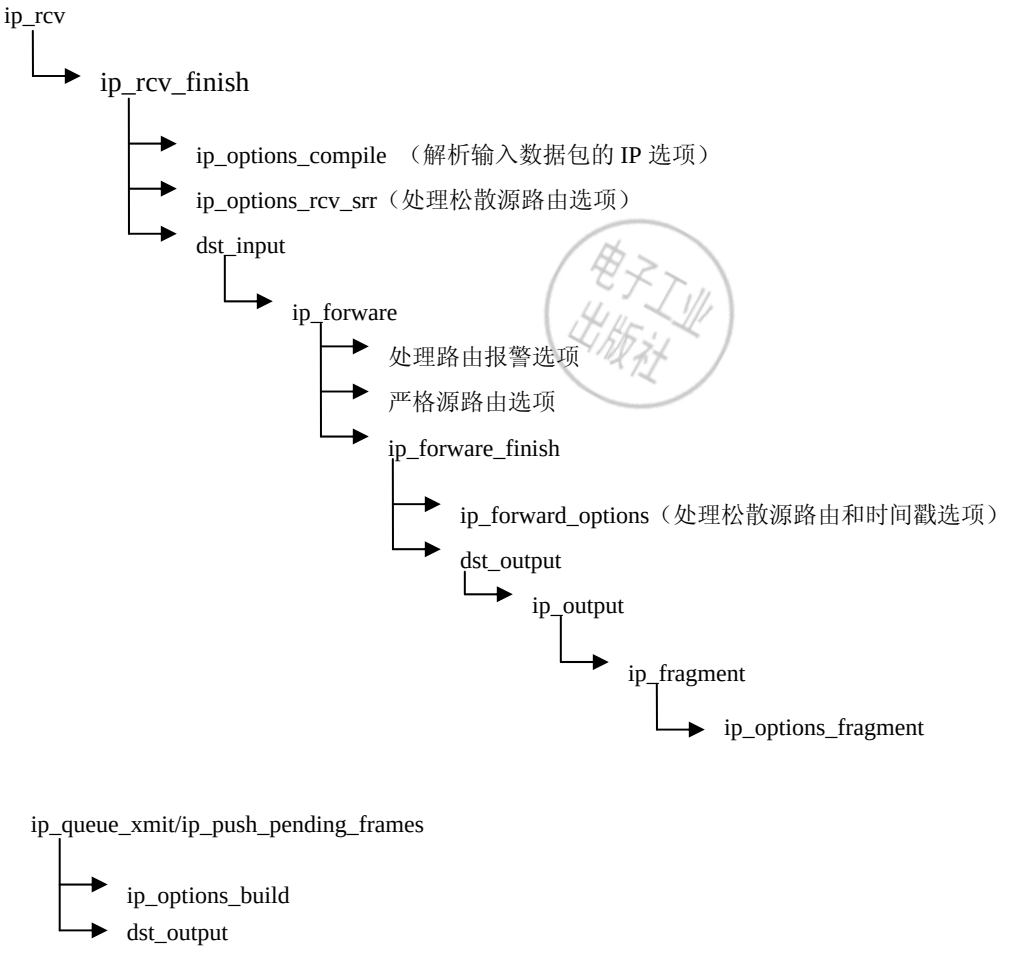


图 7-12　处理 IP 选项的 API 的调用位置

7.4.4　Linux 内核对 IP 选项处理的具体实现

处理 IP 选项的过程包括从数据包中读取 IP 选项信息并解析，在数据包的处理流程中按解析出 IP 选项信息查询路由，添加新信息到 IP 协议头，重建 IP 协议头等。

这里所说的对 IP 选项进行解析的过程，是指将 IP 选项从数据包的头信息中提取出来进行分析，将分析结果存放在 struct ip_options 数据结构中，以方便在程序中处理。不同的 IP 选项需要在 IP 协议层的不同代码部分处理，将选项存放在事先定义好的数据结构中可以使处理过程更快捷。在理解 IP 选项的解析原理时，需要了解以下的问题。

- 解析函数的调用场合。解析函数的调用场合不同，原始 IP 选项的存放位置也不同。
- 从何处获取原 IP 选项，解析完后结果应如何处理。
- IP 选项的正确性保证及错误的处理。
- 各种 IP 选项的处理流程。

以下我们就从这几个方面来分析 IP 选项在内核中的解析过程，解析 IP 选项的任务由 ip_options_compile 函数完成。ip_options_compile 会在以下两种情况下被调用。

- 由 ip_rcv_finish 函数调用来解析输入 IP 数据包的 IP 选项。ip_rcv_finish 函数会被调用来处理所有的输入数据包：需本机接收的或前送的数据包。在本节中当提到输入数据包时，也包括将要前送的数据包的情况。
- 由 ip_options_get 调用，解析由 setsockopt 系统调用为 AF_INET 套接字设置的 IP 的选项。

1．解析函数的调用场合，获取原始 IP 选项

```
int ip_options_compile(struct net *net, struct ip_options * opt, struct sk_buff * skb)
```

根据 ip_options_compile 函数后两个参数（opt 和 skb）的值可获知该函数被调用的场合及原始 IP 选项存放的位置。

- skb 不为空，opt 为空：解析的是输入数据包的 IP 选项，选项存放在 Socket Buffer 的数据包中，应从 Socket Buffer 数据包的 IP 协议头中取出 IP 选项解析，结果存到 opt 指向的数据结构中。
- skb 为空，opt 不为空：解析外传数据包的 IP 选项，选项存放在 opt 指向的数据结构的__data 数据域，应从 opt 指向的数据结构中提取 IP 选项进行解析。解析结果用于创建 IP 选项数据，存入到 Socket Buffer 数据缓冲区的 IP 协议中。

2．IP 选项解析的错误处理

如果 IP 选项解析失败，ip_options_compile 立即返回。该函数的调用程序将按以下方式之一处理返回值：

- 在接收数据包中有非法选项：给数据包发送方返回一个 ICMP 错误消息。
- 在发送数据包中有非法选项：通过返回的错误值获取通知。

3．IP 选项解析失败的原因

- 在 IP 协议头中单字节的同一选项不能出现多次。唯一的例外是 IPOPT_END 或 IPOPT_NOOP，这两个选项都可以出现任意次，经常用于数据对齐。
- IP 协议头的数据域分配了不合法值，或当前用户禁止使用的值，或有内 Linux 核不

能识别的选项子编码。这种情况主要出现在本地产生的数据包中。只有超级用户可以产生有选项的 IP 数据包。

4．ip_options_compile 函数的处理流程

在 ip_options_compile 函数的处理中使用了大量的局部变量，理解这些局部变量在函数处理过程中的作用和变化，对理解函数的流程至关重要。

- l：尚未处理的 IP 选项的总长度。
- iph：指向 IP 协议头起始地址的指针。
- optptr：当前正处理的 IP 选项的起始地址。
- optlen：当前正在处理的 IP 选项的长度。
- pp_ptr：如果 IP 选项出错，指向出错位置的地址指针（给 ICMP 使用）。

```
int ip_options_compile(struct net *net, struct ip_options *opt, struct sk_buff *skb)
{
  int l;
  unsigned char * iph;
  unsigned char * optptr;
  int optlen;
  unsigned char * pp_ptr = NULL;
  struct rtable *rt = NULL;
/*根据 skb 的值确定 ip_options_compile 是在何处被调用。
输入数据包：IP 选项在 skb 的 IP 协议头中
输出数据包：IP 选项在 opt 结构的__data 数据域中*/
if ( skb !=  NULL ) {                                          //处理接收数据包
            rt  = skb -> rtable
      optptr = (unsigned char *) &(ip_hdr( skb[1] );           //获取 IP 选项地址
}else                                                          //处理本地产生的数据包
  optptr = opt->__data;                                        //从 opt 参数中获取选项
iph = optptr – opt->__data;
}
```

（1）ip_options_compile 函数的主循环

ip_options_compile 函数中的 for 主循环就是一个接一个地查看选项，将解析的结果存放在 struct ip_options 数据结构中。在理解代码时，需掌握以下几点：

- optptr 指向选项块中当前正在分析的位置。optptr[1]是该选项的长度，optptr[2]是存放选项的指针（指明选项从什么位置开始存放），如图 7-13 所示。

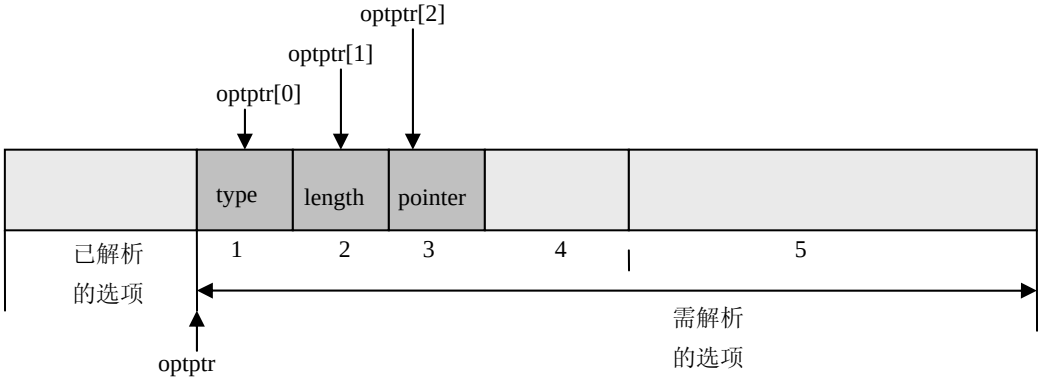


图 7-13　ip_options_compile 函数的参数说明

- 对每个选项的处理总是先做两项正确性检查。

每个选项的起始两个字节是 type 和 length，如果 length 的值小于 2 或大于尚未分析的选项块的长度，则选项信息有错；选项至少为 4 个字节长。因为每个选项的头是 3 个字节长，pointer 数据域的值不能小于 4。

- optlen 获取当前选项的初始长度。不要将 optlen 与 opt->optlen 混淆，opt->optlen 是 IP 选项的总长度，局部变量 optlen 是当前要解析的 IP 选项的长度。
- is_changed 标志用于跟踪协议头被处理的情况，表明协议头是否被修改过。
- 在 IPOPT_END 选项后不应再有别的选项，因此，一旦发现 IPOPT_END 选项，无论其后跟的是什么内容，都被改写为 IPOPT_END。

```
/*开始 ip_options_compile 主循环体*/
for ( l =  opt->optlen ;  l >  0;   ) {
  switch( *optptr ) {

/*任何 IPOPT_END 后的信息都被 重写为 IPOPT_END，并设置头信息被修改的标志*/

    case IPOPT_END:
          for ( optptr++,  l--;  l  > 0; optptr++, l--   ) {
       if ( *optptr != IPOPT_END ) {
            *optptr = IPOPT_END;
            opt->is_changed = 1;
 }
     }
     goto eol;
   case IPOPT_NOOP:
  l--;
  optptr++;
  countinue;
}

/*逐一获取选项，对其做正确性检查，当前选项长度应大于 2，小于余下选项块的总长度。如未通过以上检查，则选项有错，记录
  出错位置，供 ICMP 消息使用*/

optlen=optptr[1]                          //取当前待解析的选项的长度
        if (  optlen< 2  ||  optlen > l  ) {
             pp_ptr=optptr;               //保存选项出错的位置，用于 ICMP 消息
      goto error;
        }
```

（2）IP 选项错误处理

当 IP 选项中发生错误时，一个特定的 ICMP 消息会被发送给数据包发送方。这个 ICMP 消息包中包含原始的 IP 协议头，后跟 8 个字节的负载，以及一个指针指明错误发生的位置（相对于起始地址的偏移量）。8 个字节的负载是传输层协议头起始地址和端口号；这使接收 ICMP 消息方能找到与本次 IP 数据包发送失败套接字。在返回错误消息之前，ip_options_compile 函数会初始化 pp_ptr 指针指向出错的位置。

在 ip_options_compile 函数的 for 主循环的 switch 语句中，根据 IP 选项的 type 数据域的值分析 IP 选项。

```
    switch( *optptr )
```

下面我们逐一地分析多字节选项的解析过程。

5．严格和松散源路由选项

（1）解析过程

在提取当前选项后，做第二项正确性检查。

- 当前选项长度至少大于 3（至少需包含 type、lengh、 pointer 数据域）；如果选项的长度小于 3 ，则认为选项有错，记录出错位置，转到错误处理。
- pointer optptr[2]中的值不应小于 4。

```
  switch ( *optptr ) {
        case  IPOPT_SSRR:
        case  IPOPT_LSRR:
          if ( optlen < 3 ) {                    // 当前选项长度应用大于或等于 3
        pp_ptr  =  optptr  + 1;
        goto  error;
}
if ( optptr [ 2 ] < 4 ) {                        //pointer 的值应大于或等于 4
  pp_ptr = optptr + 2;
  goto error;
}
if (opt->srr) {                                  //如源路由选项不止设置了一次，
        pp_ptr = optptr;                         //记录错误位置
        goto error;
        }
}
```

当 skb 参数为 NULL 时，意味着 ip_options_compile 由输出数据包（由本地主机创建的数据包，不是前送的数据包）函数调用，当前解析的是输出数据包的选项。这时，源路由选项的数据，IP 地址列表来自于用户地址空间，存放在 opt->__data 数据组中。我们要将第一个 IP 地址复制到 opt->faddr 数据域，供数据包发送函数创建 IP 选项时使用，随后用 memmove 将第一个 IP 地址从 opt->__data 中移走，后面的 IP 选项移至 pointer 后。

```
if (!skb) {
  if (optptr[2]!=4||optlen<7||((optlen-3)&3)){
    pp_ptr = optptr + 1;
    goto error;
  }

//将 opt->__data 中第一个 IP 地址复制到 opt->faddr 中，将 opt->__data 中后面的 IP 地址前移

  memcpy(&opt->faddr,&optptr[3],4);
  if (optlen>7)
    memmove(&optptr[3],&optptr[7],optlen-7);
}

//如果为严格源路由选项，设置选项数据结构中的 is_strictroute 标志，设置路由记录位置 srr

opt->is_strictroute=(optptr[0] == IPOPT_SSRR);
opt->srr = optptr-iph;
break;
```

（2）解析源路由选项时要设置的标志

- 在 IP 协议头中只能出现一个源路由选项。如果 IP 选项中设置了源路由选项，并且在其后的解析过程中没有出现错误，则设置 opt->srr 标志。
- opt->is_strictroute 标志用于告诉调用程序，源路由是严格路由还是松散路由。

图 7-14 与图 7-15 给出了函数 ip_options_compile 的流程图。

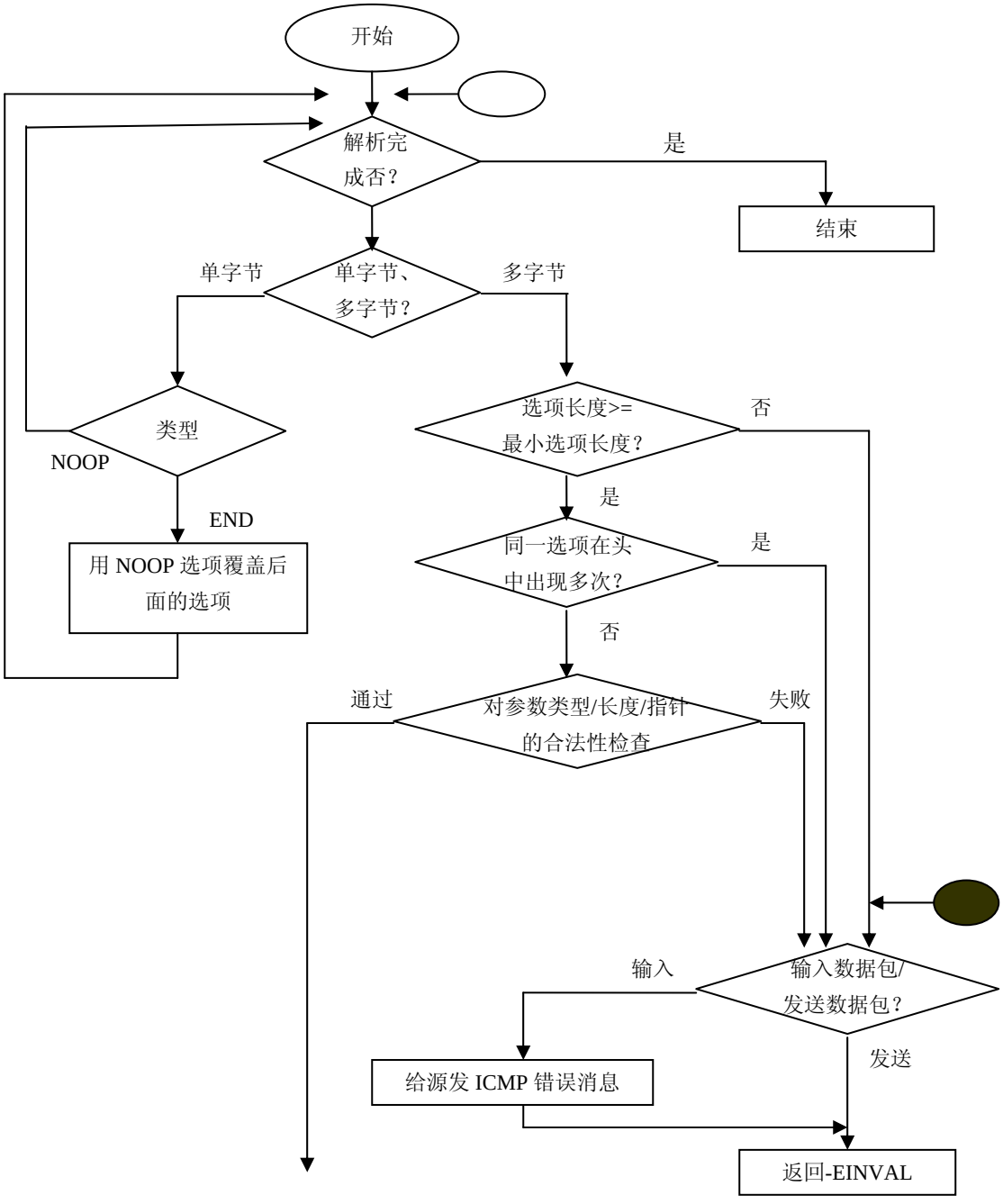


图 7-14　ip_options_compile 函数的处理流程

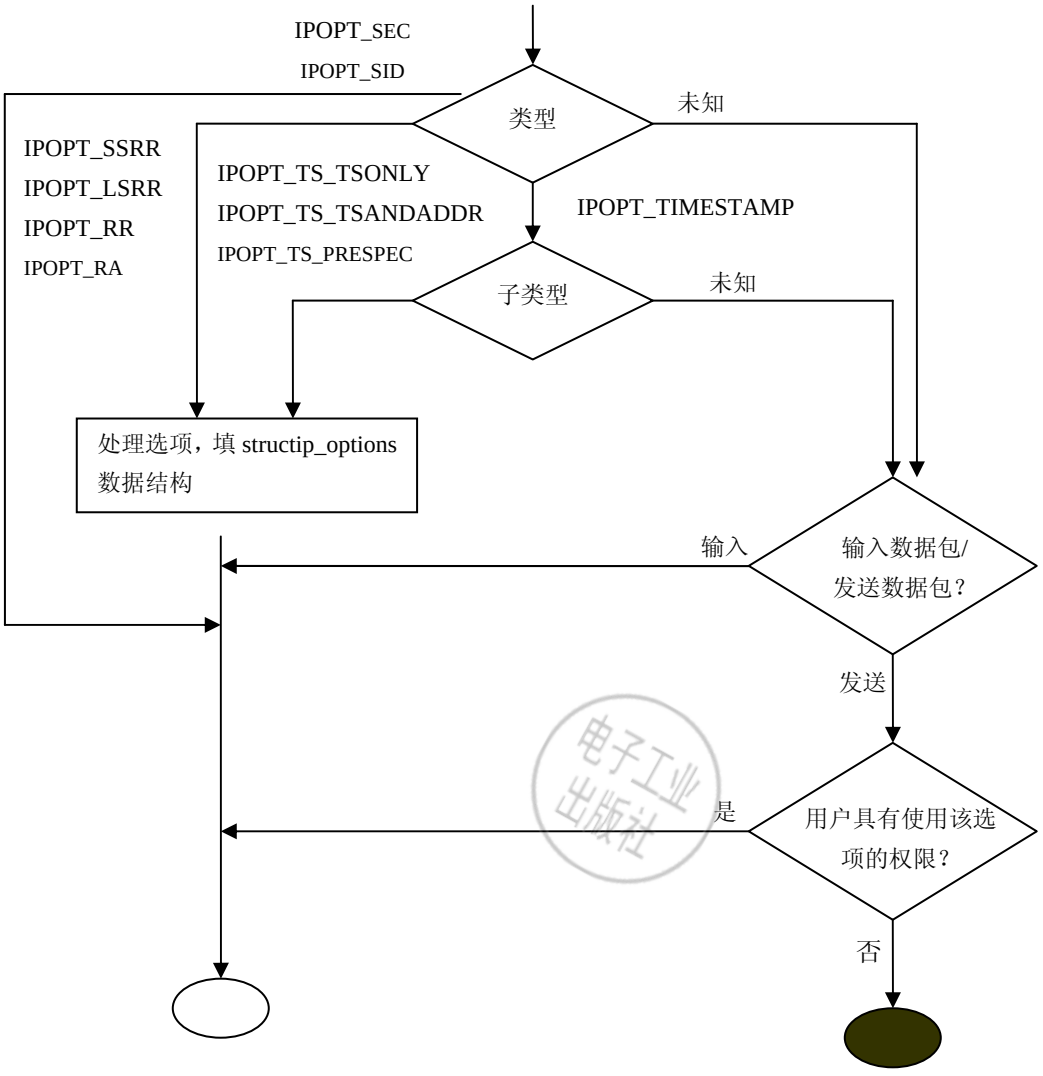


图 7-15　ip_options_compile 函数的处理流程（续）

6．记录路由选项

如果设置了记录路由选项和时间戳选项，则数据包的发送者都要在协议头中预留空间，为以后填写信息使用。这样当各个站点处理选项时，在协议头中还有空余位置的情形下，站点会向协议头中写入新的内容。如果协议头中事先预留的空间还能容纳新的成员，ip_options_compile 函数会设置 struct ip_options 数据结构的 rr_needaddr 标志，告诉路由子系统在完成路由决策后将输出网络设备接口的 IP 地址并写入 IP 协议头中。

（1）解析过程

① 对选项的正确性检查与前面完全类似。记录路由选项在 IP 选项中只能设置一次，当前选项的长度不能小于 3，pointer 的值不能小于 4。

② 输入数据包。如果 skb 非空，则复制要记录的源 IP 地址到 IP 协议头中记录地址的列表中。

③ 前送和输出数据包。在 IP 选项中记录源 IP 地址的位置预留 4 个字节的空间，optptr 指针后移 4 个字节。是否复制 IP 地址到 IP 协议头中，由前送数据包和输出数据包处理函数根据 opt->rr_needaddr 标志决定。

```
if (optptr[2]<=optlen){
  if (optptr[2]+3>optlen) {
        pp_ptr=optptr + 2;
        goto error;
}

//选项中设置记录路由选项，为输入数据包时复制 IP 地址到 IP 协议头，设置协议头已修改标志，如果为输出数据，则设置需记
//录 IP 地址标志 rr_needaddr 和定入位置 rr

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;
```

（2）标志设置

- 输入数据包：将源 IP 地址复制到 IP 协议头中后，更新 opt->is_changed 标志，设置该标志后会强制 IP 层更新校验和。
- 输出数据包：设置 opt->rr_needaddr 标志，告诉数据包前送/输出函数在构造 IP 选项时要记录当前面向站点源 IP 地址和写入位置 rr。

7．时间戳选项

（1）选项解析过程

对选项正确性检查与前面相同。选项通过正确性检查后，设置标志 opt->ts。随后查看 IP 协议头中是否还有空间来存放新的信息。

optlen 代表的是当前正在分析的 IP 选项的长度，以下的 if 语句用来查看 IP 协议头中是否还有空间来存放新的信息，这时选项的长度 optlen 代表的是数据包发送方为该 IP 选项预留的空间（迄今为止还没使用的空间）。例如，放选项数据的起始位置 optptr[2]小于选项长度 optlen，表明协议头中还有空间写入新数据。

```
if (optptr[2] <= optlen) {
     __be32 *timeptr = NULL;
     if ( optptr [ 2 ] + 3  > optptr[1]) {
        pp_ptr = optptr + 2;
        goto error;
     }
…
```

（2）根据子选项的设置做处理

无论子选项是什么类型的，选项处理程序都需要以下两方面的信息：

- 它是否需要记录 IP 地址、时间戳或两个都要记录。

- 新数据应写入 IP 协议头的什么位置。

如果子选项为 IPOPT _TS_ONLY，只记录数据包到达本站点的时间，则对时间的记录处理如下：

- skb 非空：当前处理的数据包为输入数据包，timeptr 指针被初始化为时间戳写入 IP 协议头的位置指针。
- 处理输出数据包：timeptr 指针不会被初始化（保持为 NULL）。这时在 IP 协议头中为记录时间戳数据预留空间。时间戳数据会在 ip_options_build 函数中，当 opt->ts_needtime 标志位设置为 1 时写入 IP 协议头中。

无论是处理输入数据包还是输出数据包，如果经解析后确定 IP 选项中设置了记录时间戳选项，则设置 opt->ts_needtime 标志。

如果子选项为 IPOPT _TS_TSANDADDR，记录时间和 IP 地址，则对选项的处理如下：

- 处理输入数据包：从路由表中复制 IP 地址到数据包的 IP 协议头中，timeptr 指针被初始化为时间戳写入 IP 协议头的位置指针。
- 处理输出数据包：在 IP 协议头中为时间和 IP 地址预留 8 个字节的空间。
- 最后设置 opt->ts_needtime 和 opt->ts_needaddr skb 标志。

如果子选项为 IPOPT_TS_PRESPEC，只记录数据包到达列表中 IP 地址主机的时间，则对选项的处理如下：

- 从 IP 协议头中将 IP 地址列表的当前 IP 地址复制出来，通过 inet_addr_type 函数来查看该 IP 地址是否与本机 IP 地址匹配。如果与本主机 IP 地址匹配，则做以下处理：

```
{
  __be32 addr;

//从 IP 协议头选项中复制当前 IP 地址，调 inet_addr_type 查看该 IP 地址是否与本机 IP 地址匹配，如匹配说明需要记录数据包到达本机的时间，初始化记录时间位置指针 timeptr 与标志 ts_needtime

  memcpy( &addr, &optptr[optptr[2]-1], 4 );
  if ( intet_addr_type(addr)==RTN_UNICAST )
       break;
  if ( skb )
       timeptr = ( __u32 * )&optptr[ optptr [ 2 ]+3 ] ;
}
opt->ts_needtime = 1;
optptr[2] += 8;
```

- 输入数据包：初始化 timeptr 指针为存放时间的地址。
- 输出数据包：为记录数据预留空间。设置 opt->ts_needtime 标志。

（3）写入时间值

在处理时间戳选项时，需按照子选项的类型，将时间戳写入 IP 协议头的不同偏移处，以上代码按子选项的类型先正确初始化 timeptr（时间戳写入位置指针），在下述代码中再将时间戳复制到正确位置。

```
//如果 timeptr 指针为非空，说明根据前面的解析，需要记录数据包到达本机的时间戳，将时间戳复制到 timeptr 指针指定的
//协议头位置，并设置 IP 协议头已改变标志
if ( timeptr ) {
  struct  timeval  tv;
```

```
    __u32           midtime;
    do_gettimeofday( &tv  );                          //读取时间值
    midtime=htonl ( ( tv.tv_sec %86400 ) * 1000 + tv.tv_usec /1000);
    memcpy(  timeptr,  &midtime,  sizeof(__u32  )  );  //复制时间戳
    opt->is_changed = 1;
  }
```

（4）查看记录时间戳选项空间是否溢出

8．路由告警选项

这个选项的最后两个字节必须为 0，如果这个选项通过了正确性检查，ip_options _compile 初始化 router_alert 标志，随后 ip_forward 会处理它（opt->router_alert 是布尔变量，其值为 0 或非 0）。

```
  if ( optptr[2]==0 && optptr[3] == 0  )
    opt->router_alert = optptr – iph;
```

9．处理解析错误

如果在解析本地生成数据包的 IP 选项时发生错误，函数给调用程序返回错误代码，如果是在解析接收到数据包 IP 选项时出错，要给发送数据包的源主机返回一条 ICMP 消息。

```
  error:
    if ( skb) {
          icmp_send(  skb,  ICMP_PARAMETERPROB, 0,  htonl((pp_ptr – iph ) <<  24 ) );
  }
  return   –EINVAL;
```

7.5　IPv4 数据包的前送和本地发送

在 ip_rcv_finish 函数结束处，如果数据包的目标地址不是本机，内核需要将数据包前送给适当的主机；反之，如果数据包的目标地址是本地主机，内核需要将数据包上传给 TCP/IP 协议栈网络层以上的协议实例。处理函数的选择由 dst_input 完成，它根据路由将处理函数设置为 ip_forward 或 ip_local_deliver。本节中我们将分析数据包前送和本地传递在 Linux 内核中的实现。

7.5.1　数据包的前送

与本章前面描述的许多处理过程一样，数据包前送也分为两个阶段完成：ip_forward 和 ip_forward_finish。第二个函数在第一个函数执行结束时调用。完成数据包前送功能的函数都定义在 net/ipv4/ip_forward.c 文件中。

当程序运行到 ip_forward 函数时，数据包前送需要的信息都已准备好，这些信息包括以下几种。

- 数据包前送路径的路由信息：存放在 skb->dst 数据域中或由 ip_rcv_finish 函数调用 ip_route_input 获取路由存放在 skb->dst 中。
- IP 选项设置已解析完成存放在 ip_options 数据结构类型变量中。

处理数据包前送的主要步骤如下：

① 处理 IP 选项。主要是在 IP 协议头中记录本机 IP 地址和数据包到达本机的时间信息（如果 IP 选项设置要求记录这些信息）。

② 基于 IP 协议头的数据域确定数据包可以前送。

③ 对 IP 协议头中的 TTL（Time To Live）数据域减 1，如果 TTL 的值变为 0，扔掉数据包。

④ 基于路由的 MTU，如果数据包的长度大于 MTU，处理对数据包的分片。

⑤ 将数据包通过选定的网络接口发送出去。

⑥ 处理错误。如果由于某种原因或错误，数据包不能前送，源主机会收到一条 ICMP 的消息描述遇到的问题。当数据包通过另一路由重定向前送时，源主机也会收到一条 ICMP 通知消息。以下我们就对数据包前送的函数 ip_forward 与 ip_forward_finish 中的活动进行分析。

1．ip_forward 函数分析

ip_forward 是接收到的网络数据包的目标地址（MAC 地址）不是本机地址时，是由 ip_rcv_finish 函数调用执行的。传给该函数的输入参数，是与收到的数据包相连的 skb 数据结构。处理数据包需要的所有信息都包含在 skb 结构中。图 7-16 总结了 ip_forward 函数的内部流程。整个函数处理步骤如下：

① 处理 Router Alert IP 选项。

如果数据包的 IP 协议头设置了 Router Alert 选项，这时会对该选项做处理。处理 Router Alert 选项的函数是 ip_call_ra_chain，ip_call_ra_chain 函数依据一个全局套接字列表：ip_ra_chain，其中的套接字对所有设置了 IP_ROUTER_ALERT 选项的数据包感兴趣。

```
struct ip_ra_chain
{
  struct ip_ra_chain    *next;
  struct sock           *sk;
  void                  (destructor)(struct sock * );
};
extern  struct          ip_ra_chain*ip_ra_chain;
```

当一个输入的 IP 数据包已分片时，ip_call_ra_chain 首先重组整个 IP 数据包，然后将数据包传递给 ip_ra_chain 列表中的裸套接字（Socket），返回成功。

如果 IP 协议头中没有 Router Alert 选项，或该选项设置了，但运行的进程都不处理它（这时 ip_call_ra_chain 返回 FALSE），则 ip_forward 继续处理。

```
if (IPCB(skb)->opt.router_alert && ip_call_ra_chain(skb))
        return NET_RX_SUCCESS;
```

② 查看数据类型。

检查确定数据包在数据链路层寻址的是本主机。skb->pkt_type 在 L2 层初始化，它定义了数据包的类型。当数据包在数据链路层的地址（MAC 地址）是本机的网络接口时，它把数据包的类型设置为 PACKET_HOST，如底层的函数工作正确，这里无须做这个检查，但在这里做此检查的目的是，第一时间避免接收不该接收的数据包。

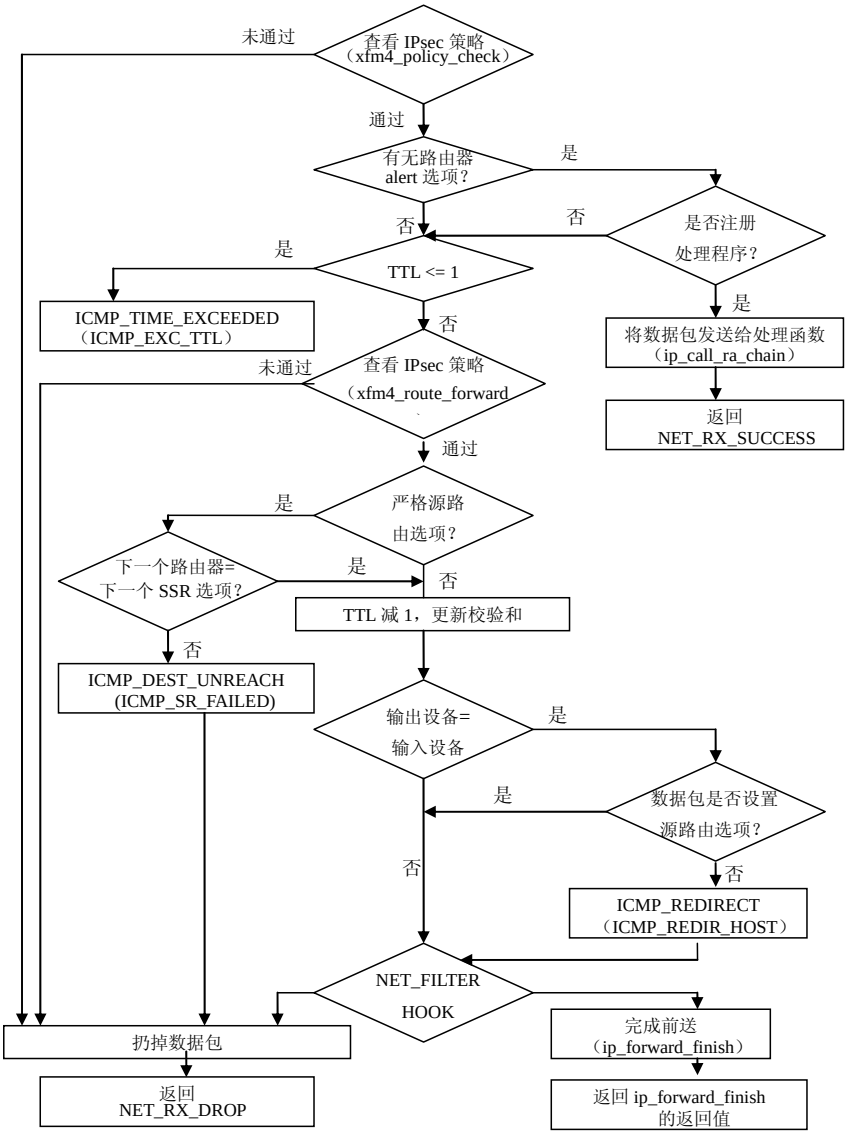


图 7-16　ip_forward 函数的处理流程

```
if (skb->pkt_type != PACKET_HOST)
        goto drop;
```

③ 开始真正的前送处理过程。

对 TTL 数据域减 1，IP 协议的定义要求当 TTL 的值为 0 时（即你收到数据包时，TTL 的值是 1，对 TTL 减 1 后它变成“0”了），就扔掉数据包，并给数据包的原始发送方回传一个 ICMP 消息，通知其数据包已被扔掉。

```
if (ip_hdr(skb)->ttl <= 1)
        goto too_many_hops;
```

注意，此时 TTL 的值还没有递减，递减的操作会在几行代码以后做。此时没对 TTL

递减，是因为此时数据包可能还与别的子系统共享，还不能对头信息进行修改。

④ 处理 IP 选项。

如果 IP 协议头信息中设置了严格源路由选项，但从选项中提取的下一个站点 IP 地址与从路由子系统中得出的不匹配，则源路由失败，扔掉数据包。

```
rt = skb->rtable;
  if (opt->is_strictroute && rt->rt_dst != rt->rt_gateway)
       goto sr_failed;
```

⑤ 处理分段。

如果数据包的长度大于传输路由上的最大传输单元（MTU），但协议头中设置了不能分段标志（IP_DF），扔掉数据包，给数据包发送方返回一个 ICMP 消息。

```
if (unlikely(skb->len > dst_mtu(&rt->u.dst) && !skb_is_gso(skb) &&
           (ip_hdr(skb)->frag_off & htons(IP_DF))) && !skb->local_df) {
  IP_INC_STATS(dev_net(rt->u.dst.dev), IPSTATS_MIB_FRAGFAILS);
       icmp_send(skb, ICMP_DEST_UNREACH, ICMP_FRAG_NEEDED,
              htonl(dst_mtu(&rt->u.dst)));
       goto drop;
  }
```

⑥ 结束处理。

在大多数的正确性检查完成后，函数修改数据包的协议头信息，然后把处理交给 ip_forward_finish，这时我们要改变缓冲区中的内容，如果缓冲区是与别的子系统共享的，首先需为自己制作一个局部复制，由 skb_cow 完成。

```
// 如果 skb 与别的进程共享，制作一个局部复制。将 TTL 值减 1，同时更新校验和
if ( skb_cow ( skb,  LL_RESERVED_SPACE(rt->u.dst.dev) + rt ->u.dst.header_len))
       goto drop;
  iph = ip_hdr(skb);
  ip_decrease_ttl(iph);
//如果 IP 选项没有设置源路由选项，路由子系统获取的路由更优化，则对数据包重定向，并向数据包的发送方传送一 ICMP 消息，通知数据包被重定向
  if (rt->rt_flags&RTCF_DOREDIRECT && !opt->srr && !skb_sec_path(skb))
       ip_rt_send_redirect(skb);
  skb->priority = rt_tos2priority(iph->tos);
```

其次完成对 TTL 减 1 操作， 同时更新校验和。

随后，在发送数据包的主机没有要求源路由时，如果从路由子系统中得出的传输路径更好，则源主机会得到一个 ICMP REDIRECT 路由重定向消息的通知。

最后，设置 skb->priority 数据域，该数据域给 IP 协议头的 Type of Service 使用，流量控制子系统（QoS）使用优先级决定数据包发送顺序。

函数结束于请求网络过滤子系统执行 ip_forward_finish，如果网络过滤子系统中没有规则禁止数据包前送，则执行 ip_forward_finish。

```
return NF_HOOK ( PF_INET, NF_IP_FORWARD, skb, skb->dev, rt->u.dst, dev, ip_forward_finish)
```

2．ip_forward_finish 函数

如果数据包的处理流程执行到了 ip_forward_finish，说明数据包通过了所有正确性与数据包安全性检查，准备通过网络设备送出给网络上的另一个主机。到目前为止，IP 协议头中的两个选项已经处理（如果设置了）：Router Alert 和 Strict Source Routing，余下的 IP 选

项会在 ip_forward_finish 函数中处理。

ip_forward_options 函数处理前送数据包的 IP 选项，它通过查询由 ip_options_compile 解析选项时初始化的标志，如 opt -> rr_needaddr、opt->ts_needaddr 来决定要在数据包的 IP 协议头中添加什么信息，ip_forward_options 在 IP 协议头信息改变的情况下会重新计算校验和。最后数据包由 dst_output 处理。

```
static int ip_forward_finish(struct sk_buff *skb)    //net/ipv4/ip_forward.c
{
  struct ip_options * opt= &(IPCB(skb)->opt);
  IP_INC_STATS_BH(dev_net(skb->dst->dev), IPSTATS_MIB_OUTFORWDATAGRAMS);
  //如果 IP 协议头中设置了 IP 选项，在前送数据包前处理选项
  if (unlikely(opt->optlen))
        ip_forward_options(skb);
  //数据包交给 dst_output 处理
  return dst_output(skb);
}
```

7.5.2　dst_output 函数的实现

所有数据包发送，无论是本地创建的数据包或是接收的来自网络向别的主机前送的数据包，最终都是通过 dst_output 送达它们的目标。这时 IP 协议头处理已结束，协议头中嵌入了需要发送的信息和所有本地系统需要加入的信息。

dst_output 调用虚函数 output，虚函数 output 根据数据包的目标地址类型来初始化。目标地址为某一主机地址时，output 初始化成 ip_output，如果目标地址是组发送地址，则它初始化成 ip_mc_output，数据包的发送分 3 个阶段处理。

1．ip_output 函数实现

ip_output 函数初始化数据包的输出网络设备和传输协议，最后由网络过滤子系统对数据包进行过滤，并调用 ip_output_finish 函数完成实际的发送操作。

```
int  ip_output (struct sk_buff *skb)                 //net/ipv4/ip_output.c
{
  struct net_device *dev = skb->dst->dev;
  IP_INC_STATS (dev_net(dev), IPSTATS_MIB_OUTREQUESTS);
//设置发送数据包输出网络设备，设置数据包协议，调用网络过滤子系统回调函数过滤网络数据包，如果通过安全检查，调用
//ip_finisg_output 发送数据包
  skb->dev = dev;
  skb->protocol = htons (ETH_P_IP);
  return NF_HOOK_COND(PF_INET, NF_INET_POST_ROUTING, skb, NULL, dev,ip_finish_output,!(IPCB
(skb)->flags & IPSKB_REROUTED));
}
```

2．　ip_finish_output 函数实现

ip_finish_output 的主要任务是根据网络配置确定是否需要对数据包进行重路由，是否需要对数据包进行分割，最后调用 ip_finish_output2 与相邻子系统接口，将数据包目标地址中的 IP 地址转换成目标主机的 MAC 地址。

```
static int ip_finish_output(struct sk_buff *skb)
{
#if defined (CONFIG_NETFILTER) && defined(CONFIG_XFRM)
  if (skb->dst->xfrm != NULL) {
        IPCB(skb)->flags |= IPSKB_REROUTED;
```

```
            return dst_output(skb);
      }
#endif
//如果数据包长度 skb->len 大于路由上设备的最大传送单元，则对数据包进行分片，分片后交给 ip_finish_output2 传送，
//否则直接交给 ip_finish_output2 传送
      if (skb->len > ip_skb_dst_mtu(skb) && !skb_is_gso(skb))
            return ip_fragment(skb, ip_finish_output2);
      else
            return ip_finish_output2(skb);
}
```

注意，如数据包需要重路由，dst_output 函数可能被调用多次，这种情况可能发生在如：一个简单的机制调用一系列的 dst_output。

3．ip_finish_output2 函数的实现

ip_finish_output2 函数是与相邻子系统接口的函数，该函数的主要任务是，在数据包中为数据链路层协议插入其协议头，或为数据链路层协议头预留分配空间，然后将数据包交给相邻子系统发送函数 dst->neighbour->output 处理。

```
static inline int ip_finish_output2(struct sk_buff *skb)
{
      struct dst_entry *dst = skb->dst;
      struct rtable *rt = (struct rtable *)dst;
      struct net_device *dev = dst->dev;
      unsigned int hh_len = LL_RESERVED_SPACE(dev);
      //根据数据包的类型，更新组传送，广播传送的统计信息
      if (rt->rt_type == RTN_MULTICAST)
            IP_INC_STATS(dev_net(dev), IPSTATS_MIB_OUTMCASTPKTS);
      else if (rt->rt_type == RTN_BROADCAST)
            IP_INC_STATS(dev_net(dev), IPSTATS_MIB_OUTBCASTPKTS);
      //如果当前数据包缓冲区的 headroom 小于数据链路层协议头长度，且设备初始化了操作协议头函数，则为 L2 层协议头分配
空间
      if (unlikely(skb_headroom(skb) < hh_len && dev->header_ops)) {
            struct sk_buff *skb2;
            skb2 = skb_realloc_headroom(skb, LL_RESERVED_SPACE(dev));
            if (skb2 == NULL) {
                  kfree_skb(skb);        //分配空间不成功，扔掉数据包
                  return -ENOMEM;
            }
            if (skb->sk)                 //如原 skb 属于某个套节字
                  skb_set_owner_w(skb2, skb->sk);//为新 skb2 设置所属套节字
            kfree_skb(skb);
            skb = skb2;
      }
      //将数据包交给相邻子系统输出处理函数 neigh_hh_output 或 dst->neighbour->output 处理
      if (dst->hh)
            return neigh_hh_output(dst->hh, skb);
      else if (dst->neighbour)
            return dst->neighbour->output(skb);
…
```

7.5.3 本地发送的处理

当数据包的目标地址为地本机时，在 ip_rcv_finish 函数中会将 skb->dst->input 初始化为 ip_local_deliver。本地发送功能也分两个阶段完成：ip_local_deliver 和 ip_local_deliver_finish。

本地发送的一个重要任务就是重组分了片的数据包，除了某些特殊情况外，如网络过滤子系统重组了数据包来查看其内容，这时分过片的数据包已完成重组。与此相反，在大多数情况下，前送操作不必关心数据包的重组，前送操作可以独立前送每个分片数据包。

在本地重组数据包后，将原始数据包上传给上层协议处理函数。

1．在本地重组数据包

ip_local_deliver 函数的主要任务就是重组数据包，重组数据包的功能通过调用 ip_defrag 函数完成。ip_defrag 完成数据包重组后返回指向完整数据包的指针，如果这时还没有收到数据包的所有分片，数据包还不完整，则它返回 NULL。

数据包重组成功，调用网络过滤子系统的回调函数来查看数据包的配置，并执行 ip_local_deliver_finish 完成数据包上传功能。

```
int ip_local_deliver(struct sk_buff *skb)            //net/ipv4/ip_input.c
{
/*如数据包有分段，重组数据包，重组不成功，返回 NULL，否则通过网络过滤子系统调用 ip_local_deliver_finish 上传
数据包*/
  if (ip_hdr(skb)->frag_off & htons(IP_MF | IP_OFFSET)) {
        if (ip_defrag(skb, IP_DEFRAG_LOCAL_DELIVER))
              return 0;
  }
//对数据包进行过滤，如果通过安全检查，数据交给 ip_local_deliver_finish 向上层协议传递
return NF_HOOK(PF_INET, NF_INET_LOCAL_IN, skb, skb->dev, NULL, ip_local_deliver_finish);
}
```

2．数据包从 IP 层上传至传输层

完成数据包从网络层向传输层发送功能的函数是 ip_local_deliver_finish，它的主要任务是：

① 将数据包传给正确的上层协议处理函数。

② 将数据包传给裸 IP。

③ 强制执行数据安全策略（如在内核配置时选择了安全策略）。

要完成以上功能，ip_local_deliver_finish 函数需要获取以下信息：

- 传输层处理数据包的协议：来自 IP 协议头的 protocol 数据域。
- 上层协议的处理函数是什么。

首先我们看看传输层协议处理函数在内核中的组织，再来分析 ip_local_deliver_finish 函数向上层协议发送数据包的实现流程。

（1）传输层的协议

在 IP 协议头中 protocol 数据域占了 8 位，所以传输层最多可以有 256 个不同的协议，Linux 内核中支持传输层的协议标识符和编码定义在 include/linux/in.h 文件中，协议标识符的格式都为：IPPROTO_XXX。

```
enum {                          //include/linux/in.h
  IPPROTO_IP = 0,
  IPPROTO_ICMP = 1,
  IPPROTO_IGMP = 2,
  IPPROTO_IPIP = 4,
  IPPROTO_TCP = 6,
  IPPROTO_EGP = 8,
  IPPROTO_PUP = 12,
```

```
  IPPROTO_UDP = 17,
…
  IPPROTO_RAW  = 255,          /* Raw IP packets                */
  IPPROTO_MAX
};
```

（2）传输层协议处理函数的组织

传输层的每个协议都实现了自己接收输入网络数据包的处理函数，完成对输入数据包各方面的处理，如分段、出错处理等，这些处理函数组织在 struct inet_protocol 数据结构中，传输层的各协议实例定义自己的 struct inet_protocol 数据结构实例，实现 struct inet_protocol 数据结构中的函数指针，传输层各协议实例的 struct inet_protocol 数据结构实例按一定的方式组织，形成传输层与网络层之间接收网络数据包的接口。以这样的方式组织的接口，使网络层的数据包可以传给传输层的一个或多个协议处理函数。struct inet_protocol 数据结构定义在 include/net/protocol.h 文件中。

```
struct inet_protocol {                              //include/net/protocol.h
  int              (*handler)(struct sk_buff *skb);
  void             (*err_handler)(struct sk_buff *skb, u32 info);
  int              (*gso_send_check)(struct sk_buff *skb);
  struct sk_buff         *(*gso_segment)( struct sk_buff *skb, int features);
  struct sk_buff        **(*gro_receive)(struct sk_buff **head, struct sk_buff *skb);
  int              (*gro_complete)(struct sk_buff *skb);
  unsigned int           no_policy:1,
                   netns_ok:1;
};
```

struct inet_protocol 数据结构中关键的函数如下。

- Handler：协议注册到内核处理输入数据包的函数。
- err_handler：协议注册到内核处理接收到 ICMP UNREACHABLE 错误消息的处理函数。
- no_policy：在 TCP/IP 协议栈中某些关键部分要根据这个数据域的设置来确定是否需要为协议查看 IPsec 策略。如果其值为 1 则不需要查看 IPsec 策略。

（3）网络层与传输层之间的接口组

Linux 内核支持的传输层协议都可以实现自己的协议处理函数，这些协议实例首先定义自己的 struct inet_protocol 数据结构变量实例，初始化 struct inet_protocol 数据结构中的函数指针指向其协议处理函数，然后向内核注册整个 struct inet_protocol 结构实体。

内核中所有注册了的传输层协议处理函数实例，存放在一个 struct inet_protocol 类型的 inet_protos 全局数组中，形成向量表，网络层协议头中 protocol 数据域描述的协议编码，就是该协议的 struct inet_protocol 实例在 inet_protos 向量表中的索引号。

```
extern struct inet_protocol *inet_protos[MAX_INET_PROTOS];
```

传输层协议分别调用 inet_add_protocol 函数和 inet_del_protocol 函数向内核中注册自己的协议处理函数结构块，调用 inet_add_protocol 函数后，协议处理函数结构块就插入到 inet_protos 全局数组中，全局数组中的指针指向协议处理函数结构块的起始地址。

对 inet_protos 向量表的访问按以下方式管理：

- 读-写操作用 inet_proto_lock 来控制并发访问。
- 只读操作（在 ip_local_deliver_finish 中访问 inet_protos 时就是只读操作），用

rcu_read_lock/rcu_read_unlock 来控制并发访问。

网络层与传输层之间的接收数据包处理接口组织如图 7-17 所示。

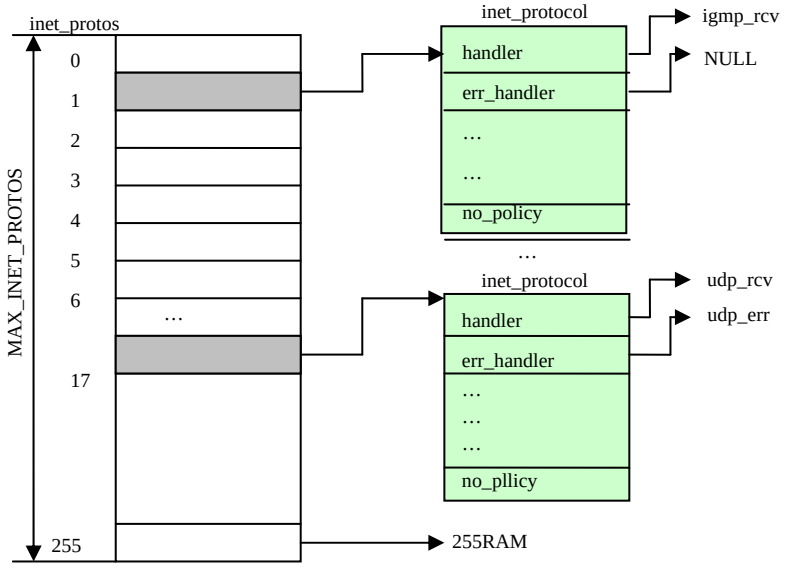


图 7-17　网络层与传输层协议之间的接口

3．ip_local_deliver_finish 函数的处理流程

在理解了内核对传输层协议处理函数结构块的组织后，再来看 ip_local_deliver_finish 函数的实现流程就比较容易了，函数结构非常清晰，划分为以下几个部分：

（1）函数的初始化部分

初始化 skb->data 指针，设置传输层协议头指针 skb->transplant_header，使其指向传输层协议头的起始地址处，这时 IP 层的处理已完成，不再需要 IP 协议头。但 IP 协议头仍可通过 skb->network_header 指针访问到。

（2）正式处理部分

ip_local_deliver_finish 函数的正式处理部分从获取访问 inet_protos 向量表的读保护锁 rcu_read_lock 开始，依次完成以下过程：

① 从 IP 协议头中获取传输层协议的编码：iphdr->protocol。

② 如果该协议是裸套接字处理函数，则将数据包传给裸套接字处理函数，调用 IPsec 检查策略查看数据包。

③ 以 iphdr->protocol 数据域为索引号在 inet_protos 向量表中查询，找到对应的传输层协议处理函数结构块，做如下处理：

- 如果配置了 IPsec 策略检查，则调用 IPsec 策略检查函数。如果未通过 IPsec 策略检查，则扔掉数据包。
- 把数据包发送给协议处理函数。如果发送不成功，则跳回到②处再发送。

（3）错误处理

在大多数情况下，数据包都是由传输层某个具体的协议处理函数来接收的，如果 inet

_protos 向量表中没有 iphdr->protocol 协议编号对应的协议处理函数，也没有裸套接字要处理数据包，则扔掉数据包。同时给数据包的发送者返回一个数据包不可到达的 ICMP 消息。

ip_local_deliver_finish 函数完成后，数据包就离开网络层上传至 TCP/IP 协议栈的传输层。

除了内核要处理网络数据包外，应用程序也可以处理数据包。因此，无论协议是否在内核中注册了处理函数，ip_local_deliver_finish 都要查看是否有应用程序创建了裸套接字来处理协议相关的数据包，如果有，就复制一个数据包传递给应用程序。

裸套接字是在用户地址空间实现传输层的协议，我们将在介绍应用层时讲解裸套接字发送/接收 IP 数据包的实现方式。

7.6 在 IP 层的发送

这一节我们讨论数据包在网络层对外发送的处理过程，对外发送指的是数据包离开本地主机前往另一主机。数据包可以是传输层初始化的本地数据，或是前送数据包的最后阶段，数据包对外发送阶段在内核的处理任务包括：

1．查找下一个站点

IP 层需要知道完成数据包输出功能的是哪一个网络设备，以及到达下一个站点的路由器，寻找路由的任务由 ip_route_output_flow 函数来完成的，它由网络层或传输层调用。

2．初始化 IP 头

在这个阶段需要填写 IP 协议头的几个数据域，如数据包标识符 ID；如果是前送数据包，一部分工作已在前面做了（包括更新 TTL、校验和、处理部分 IP 选项），但为完成最后的发送，在现阶段仍有很多工作需要完成。

- 处理其他 IP 选项：输出函数必须对需要在 IP 协议头中记录 IP 地址和时间信息的 IP 选项要求做处理。
- 数据包分片：如果发送数据包对输出设备而言太大了，则必须对数据包分片。
- 处理校验和 ：在完成所有对协议头的处理后，必须重新计算 IP 校验和。

3．由网络过滤子系统检查数据包

Linux 的防火墙子系统在数据包处理的各个阶段都要检查数据包，并可能扔掉数据包，包括发送阶段。

4．更新统计信息

根据发送处理的结果（成功或失败）或对数据包分片等活动的结果，更新与 SNMP 相关的统计信息。

要理解 IP 层数据包的向外发送过程，我们需要了解以下内容：

- 数据包从传输层送到网络层的机理。
- 除负载数据包外传输层还需向网络层发送什么额外信息来创建 IP 数据包。
- 这些数据存放在什么数据结构中。
- 数据包在 IP 层如何处理。
- 最后通过什么途径向外发送。

在本节中我们将会按以上线索来分析 IP 层的发送过程。

7.6.1　执行发送的关键函数

Linux 内核在传输层实现了多个传输层协议实例，在应用与 TCP/IP 协议栈之间发送数据包时，如果协议实例组织数据包不同，则向网络层发送数据包的方式也不同。图 7-18 给出了介于传输层和网络层发送数据包最后阶段的关键函数。

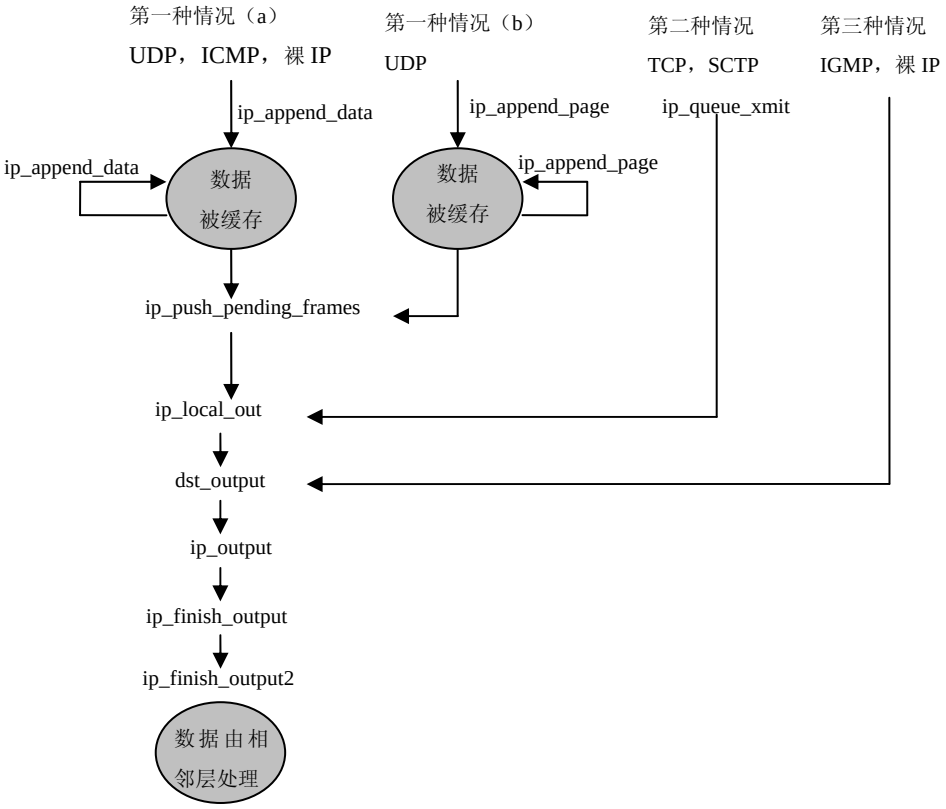


图 7-18 传输层与网络层输出函数的对应关系

图的上端显示了大部分传输层使用的协议，从图中可以看出，发送数据包的关键函数分成了两组：

第一组：列在图中右边的传输协议（TCP 和 SCTP：Stream Control Transmission Protocol），在将数据包传给网络层时已为数据包的分片做了大量预处理，留给 IP 层的工作已很少。

第二组：列在图中左端的 UDP 协议和裸 IP 协议，将所有数据包分片需要做的工作都留给了 IP 层。

TCP 和 SCTP 协议在向网络层传送数据包时调用第一组函数，UDP 和 ICMP 协议调用第二组网络层的函数来发送它们的数据包。当网络层的函数完成了工作后，就将数据包传给 dst_output。dst_output 根据数据包的路由信息将输出函数初始化为 ip_output。

对于裸 IP，当它使用 IP_HDRINCL 选项时，它自己负责构造全部 IP 协议头，所以它直接调用 dst_output。IGMP：Internet Group Management Protocol 也直接调用 dst_output（在自己初始化 IP 协议头后）。表 7-3 列出了数据包发送的关键函数。在本节中我们将按照数据从上向下发送的顺序，依次分析数据包的发送实现。

表 7-3　IP 层数据发送的关键函数

函数名	功能描述
ip_queue_xmit	由传输层的 TCP 协议调用，将数据包从传输层发送网络层，创建协议头和 IP 选项到数据包中，调用 dst_output 发送
ip_append_data	由传输层的 UDP 等协议调用，缓存从传输层传送到网络层的请求发送数据包缓冲区
ip_append_page	由传输层的 UDP 等协议调用，缓存从传输层传送到网络层的请求发送数据页面
ip_push_pending_frames	将 ip_append_data 和 ip_append_page 创建的输出队列发送出去
dst_output	数据包发送函数，当数据包的目标地址是其他主机时，初始化为 ip_output
ip_build_and_send_pkt	用于 TCP 发送同步回答消息时创建 IP 协议头和选项并发送
ip_send_reply	用于 TCP 发送回答消息和复位时创建 IP 协议头和选项并发送数据包

图 7-16 只给出了最常用的情况，在 TCP 协议管理连接并向通信伙伴发送回答消息和复位连接时，ip_send_reply 调用 ip_append_data 和 ip_push_pending_frame 来缓存回答消息数据包，所以 TCP 不只是调用 ip_quene_xmit 来发送数据，也使用 ip_append_data 来发送 TCP 建立连接需要发送的数据。

7.6.2　发送数据包相关信息的数据结构

当传输层向网络层发送数据包时，其他用于构建 IP 协议头、IP 选项、数据包分片等的控制信息也应一同传给 IP 协议层。这些控制信息由用户程序设置，通过套接字发送到 TCP/IP 协议栈。

套接字在 Linux 中以 struct sock 数据结构表示，struct sock 数据结构非常大，它定义在 include/net/sock.h 文件中。在 struct sock 的定义中对它的各数据域做了很详细的说明。关于套接字层的实现我们会在第 11 章中进行介绍。

在 Linux 内核中实现的套接字支持各种协议族（Protocol Family ），struct sock 数据结构的内存分配嵌套在特定协议族套接字数据结构中，例如，PF_INET 对应的套接字中嵌套了通用套接字的数据结构 struct sock。实现 TCP/IP 协议栈的特定协议族套接字的数据结构为 struct inet_sock ，定义在 include/linux/ip.h 文件中。struct inet_sock 数据结构的第一个数据域就是 struct sock 类型变量实例，其他的数据域存储 PF_INET 的私有信息，如源和目标 IP 地址、IP 选项、数据包 ID 和 cork 数据结构。表 7-4 列出了存放数据包控制信息的主要数据结构，理解这些数据结构的设置是理解数据包发送的基础。

表 7-4　存放数据包控制信息的主要数据结构

数据结构名	描述	定义的文件
struct sock	套接字数据结构	include/net/sock.h
struct　inet_sock	PF_INET 特定协议族套接字结构	include/linux/ip.h
struct ipcm_cookie	包含发送数据包需要的各种信息的数据结构	include/linux/ip.h
struct　cork	嵌套在 inet_sock 数据结构中，处理套接字阻塞时的选项信息	include/linux/ip.h
struct dst_entry	路由表入口	include/net/route.h
struct rtable	路由表结构	include/net/dst.h
struct in_device	存放网络设备的所有与 IPv4 协议相关的配置	
struct in_ifaddr	如果为一个网络接口分配了 IPv4 网络地址，该数据结构用于存放 IP 地址和与地址配置相关的信息	

1．重要数据结构描述

（1）struct sock 数据结构

是描述套接字层信息的数据结构。在该数据结构的定义文件中，对数据结构各数据域的含义给出了详细说明，这里就不再一一列举了。

（2）struct ipcm_cookie 数据结构

```
struct ipcm_cookie{
  __be32                addr;
  int                   oif;
  struct ip_options     *opt;
};
```

- addr：输出数据包的目标 IP 地址。
- oif：输出数据包的网络设备索引号。
- *opt：指向存放 IP 选项数据结构 ip_options 的指针。

（3）struct cork 数据结构

```
struct {
       unsigned int     flags;
       unsigned int     fragsize;
       struct ip_options*opt;
       struct dst_entry *dst;
       int         length;
       __be32           addr;
       struct flowi     fl;
  } cork;
```

- fags：标志位。当前在 IPv4 中该标志位只能设置成 IPCORK_OPT。表示在 opt 中有选项。
- fragsize：产生的分片数据段的大小。其大小包括网络层的协议头和负载数据，通常与 PMTU 的值相同。
- opt：指向存放 IP 选项的数据结构地址。
- dst：用于传输 IP 数据包的路由表缓冲区入口。
- length：IP 数据包的大小（包括所有分段数据的总和，但不包括 IP 协议头）。
- addr：发送数据包的目标 IP 地址。
- fl： TCP 协议是可靠协议，在发送数据包之前，要在发送两端建立连接，TCP 协议也要维护通信两端的连接。该数据域中存放通信两端的连接信息。

（4）struct inet_sock 数据结构

struct inet_sock 数据结构继承了通用套接字的属性（嵌入了 struct sock 数据结构），是 Linux 内核实现 TCP/IP 协议栈 PF_INET 协议族的套接字数据结构。struct inet_sock 数据结构中包含了发送数据包的大部分控制信息：数据包的目标 IP 地址、发送数据包的网络设备和 IP 选项等。而且 struck inet_sock 数据结构中包含了指向套接字 sock 的指针，嵌套了 struct cork 数据结构。

```
struct inet_sock {
  struct sock      sk;
#if defined(CONFIG_IPV6) || defined(CONFIG_IPV6_MODULE)
  struct ipv6_pinfo     *pinet6;
#endif
  __be32                daddr;
  __be32                rcv_saddr;
  __be16                dport;
  __u16                 num;
  __be32                saddr;
  __s16                 uc_ttl;
  __u16                 cmsg_flags;
  struct ip_options     *opt;
  __be16                sport;
  __u16           id;
  __u8            tos;
  __u8            mc_ttl;
  __u8            pmtudisc;
  __u8            recverr:1,
                  is_icsk:1,
                  freebind:1,
                  hdrincl:1,
                  mc_loop:1,
                  transparent:1;
  int             mc_index;
  __be32                mc_addr;
  struct ip_mc_socklist  *mc_list;
  struct {
        …
  } cork;
};
```


- daddr：外部 IP 地址，即数据包的目标 IP 地址。
- rcv_saddr：与创建数据包套接字绑定的本地 IP 地址。
- dport：数据包目标地址的套接字端口号，端口号标志了目标地址接收数据包的应用程序。
- num：本地应用程序创建该套接字的端口号。
- saddr：发送数据包的源 IP 地址。
- uc_ttl：当发送数据包的目标地址为单一主机地址时，数据包存活时间的 ttl 值。
- cmsg_flags：控制消息的标志。
- opt：输出数据包的 IP 选项。
- sport：发送数据包源端口号。
- id：未分片数据包的数据包标识符。
- tos：数据包的服务类型 ToS，代表数据包发送的优先级。

- mc_ttl：当数据包的目标地址为组发送地址时，数据包存活时间的 ttl 值。
- pmtudisc：发送路由上网络设备的最大发送单元 MTU。
- recverr：接收错误标志。
- is_icsk：标志该套接字是一个连接套接字 inet_connection_socket。
- freebind：标志该套接字可以与任意 IP 地址绑定。

……

- mc_index：组发送网络设备的索引号。
- mc_addr：组发送 IP 地址。
- mc_list：组发送的套接字列表数据结构。

在 struct inet_sock 数据结构中，struct sock sk 和 struct ipv6_pinfo　*pinet6 必须为 struct inet_sock 的前两个数据成员。当我们需要获取 struct inet_sock 数据结构地址时，定义一个指向 struct sock 数据结构的指针，IP 层用 inet_sk 宏做强制类型转换，将指针指向 struct inet_sock 数据结构，即 struct inet_sock 数据结构的基地址和 struct sock 数据结构的基地址是一样的。

struct inet_sock 的 struct cork 数据域在 ip_append_data 和 ip_append_page 中起着重要作用：它存储了这两个函数实现对数据包进行正确分割所需的信息，struct cork 数据结构中，还有 IP 协议头中的选项和分段长度，当发送数据包是由本地主机产生时，每个 sk_buff 都是由某个套接字创建的，应与一个 struct sock 数据结构实例相关，这种关联存放在 skb->sk 数据域中。

2．操作 struct sock、struct inet_sock 数据结构的关键函数

在数据包的发送过程中要用到各种函数来读取和设置 struct sock 和 struct inet_sock 的数据域，这里我们需要理解以下的 API：

（1）sk_dst_set　和　__sk_dst_set 函数

支持 TCP 协议的套接字，需要管理 TCP 连接；一旦套接字建立了连接，以上两个函数用以保存到达目标地址使用的路由；这些路由信息保存在 struct sock 数据结构中。sk_dst_set 是__sk_dst_set 的包装函数，sk_dst_set 在调用__sk_dst_set 之前获取防止并发访问的锁；如果无并发访问的情况，则无须获取锁，可直接调用__sk_dst_set 函数。

（2）sk_dst_check 和__sk_dst_check 函数

正如其函数名所示，可用这两个 API 测试到达目标地址的路由是否有效；如果路由有效，则它们返回有效路由信息。这两个函数可用于重新提取路由，不只是测试路由是否有效（一个无效的路由返回 NULL）；这两个函数非常相似，它们在技术上的不同之处在于，如何清除缓冲区中原保存的不再有效的路由。

（3）skb_set_owner_w 函数

指定一个数据包所属的套接字，即设定 sk_buff 数据结构的 skb->sk 数据域。

（4）sock_alloc_send_skb 和 sock_wmalloc 函数

分配 sk_buff 所需的内存空间。sock_alloc_send_skb 分配单个缓冲区或一系列分片数据包的第一个缓冲区；sock_wmalloc 管理其余的子分片。两个函数最终都是调用 alloc_skb 来完成对 sk_buff 的内存分配。但第一个函数更复杂，多方面因素可能导致内存分配失败；如果第一个 sk_buff 需要的内存分配成功，则接下来为分片数据分配内存就很少失败。

在理解了管理数据包发送的关键数据结构和操作数据结构的 AIP 后，接下来我们就逐一分析数据包发送的实现过程。在本节我们分析的函数都定义在 net/ipv4/ip_output.c 文件中。

7.6.3 ip_queue_xmit 函数

ip_queue_xmit 函数由 TCP 和 SCTP 协议调用，处理本地产生的外送数据包，是传输层向网络层传送数据包时调用的函数。函数原型为：

```
int ip_queue_xmit(struct  sk_buff  *skb,  int  ipfragok  )
```

ip_queue_xmit 函数的两个输入参数含义如下。

- skb：要发送的数据包。
- ipfragok：SCTP 用的标志，是否允许分段。

在具体处理输出数据包之前，ip_queue_xmit 函数要完成一系列对数据包管理初始化工作，获取发送数据包的控制信息。

这些控制信息包括数据包所属的套接字、特定套接字协议族数据结构的地址（其中包含了所有数据包发送的控制信息）、应用程序设置的 IP 选项。

当数据包是由本地应用程序创建时，该数据包一定属于某个套接字。在存放数据包 Socket Buffer 管理信息的 skb 结构中，包含了指向套接字数据结构的指针 skb->sk。

套接字 sk 的起始地址与特定协议族套接字结构 struct inet_sock 的起始地址相同，这样我们可以通过强制类型转换获取该数据包的 struct inet_sock 数据结构地址。

与 skb 相关的 struct inet_sock 套接字中包含了一个 opt 指针，指向存放 IP 协议头选项的数据结构。描述 IP 协议头选项的数据结构使 IP 层访问选项时更快捷、有效。这个结构之所以包含在 struct inet_sock 数据结构中，是因为通过同一套接字发送的数据包，它们的 IP 选项都一样，如果为每个数据包重建这些信息会非常浪费时间。

```
struct  sock  *sk = skb->sk;
struct  inet_sock  * inet= inet_skb(sk );
struct  ip_options  *opt=  inet->opt  ;
```

在 opt 数据结构中包含了各种标志位数据域与偏移量数据域，指明函数应在协议头的什么位置存放 IP 选项要求的时间戳、IP 地址等信息。注意 opt 数据结构本身不缓存 IP 协议选项，只包含一些信息告诉我们应在 IP 协议头中写入什么内容。接下来 ip_queue_xmit 函数就按以下步骤来处理外送数据包：

1．设置路由

如果 Socket Buffer 已分配了适当的路由信息（skb->rtable），就不需要参考路由表了。

```
rt = skb->rtable;               //如果 skb->rtable 数据域非空，则数据包已路由
  if (rt != NULL)
     goto packet_routed;
```

在另一种情况下，即 skb 还没有分配路由， ip_queue_xmit 函数就查看路由是否已缓存在套接字结构中，如果套接字结构中已缓存了一个有效的路由，确定它现在仍有效；这项检查由__sk_dst_check 函数完成。

```
    //如果数据包还未被路由，则查看套节字中是否已缓存有效路由
    rt = (struct rtable *)__sk_dst_check(sk, 0);
    //如果套节字中没有缓存有效的路由信息，IP 选项中配置了源路由选项，从选项的 IP 地址列表中读取下一跳的 IP 地址，在路由
表中查询，查询路由的目标 IP 地址取自 inet->daddr 数据域
      if (rt == NULL) {
         __be32 daddr;
        daddr = inet->daddr;
    //如果 IP 选项设置了源路由选项，查询新路由和 daddr 取自源路由 IP 地址列表中

        if(opt && opt->srr)
           daddr = opt->faddr;
         {
      //为用于查询路由的数据结构 struct flowi 的数据域赋值，调用 ip_route_output_flow 获
      //取新路由
           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,
                                 .flags = inet_sk_flowi_flags(sk),
                                .uli_u = { .ports =
                                      { .sport = inet->sport,
                                       .dport = inet->dport } } };
           security_sk_classify_flow(sk, &fl);
           if (ip_route_output_flow(sock_net(sk), &rt, &fl, sk, 0))
                goto no_route;
           }
           sk_setup_caps(sk, &rt->u.dst);
      }
      skb->dst = dst_clone(&rt->u.dst);
```

如果套接字没有为数据包缓冲路由，或目前 IP 层用的一个路由已无效了，ip_queue_xmit 需要用 ip_route_output_flow 来寻找一个新的路由，并把新路由存放在 sk 数据结构中。用于查找新路由的关键信息是否是数据包的目标地址，如果是则保存在局部变量 daddr 中，daddr 变量的设置为：

- IP 协议头中没有设置源路由选项，daddr 的值设置成数据包的最终目标地址 inet->daddr。
- IP 协议头中设置了源路由选项 opt->srr，daddr 的值设置成源路由选项 IP 地址列表中下一跳的 IP 地址 inet->faddr。

在严格源路由选项的情况下，由 ip_route_output_flow 查找到的下一跳的 IP 地址必须与源路由列表中下一跳的 IP 地址一致。

设置用于查找新路由的 struct flowi 数据结构的数据域值为：数据包目标地址，发送数据包源地址、数据包服务类型（ToS）、传输层协议、源端口号和目的端口号等，以上数据域设置好后调用 ip_route_output_flow 函数来查找新路由。

sk_setup_cap 将用于发送数据包的输出网络设备信息，存放在套接字的 sk 数据结构中。

如果 ip_route_output_flow 执行失败，则扔掉数据包；如果路由查找成功，则将新路由信息存放在 sk 数据结构中，当下次发送数据包时就可以直接使用该路由，不需要再查询路由表了。如果由于某种原因，路由再次失效，ip_queue_xmit 就又调用 ip_route_output_flow 在路由表中查找一条新的路由。dst->clone 对于 skb->dst 数据结构的引用计数加 1。

如果数据包被扔掉，IP 层会向上层协议返回一个错误代码，更新与 SNMP 相关的统计信息，注意这时函数不需要向源地址回送任何 ICMP 消息。

如果整个处理过程成功，我们就获取了所有发送数据包需要的信息，这时就可以构建 IP 协议头了。

2．构建 IP 协议头

到目前为止，skb 中只包含了 IP 数据包的常规负载数据、协议头和从传输层传来的负载数据。TCP 或 SCTP 协议在为 Socket Buffer 分配内存时，总是按最大可能的需要来向系统申请内存，其中考虑了下层协议头需要的空间。这样做的缺点是：会造成较大内存空间浪费；但它的优点是：避免了 IP 层或更低层协议在空间不够的情况下做缓冲区的复制或重分配，提高了执行效率。

```
   //查看 skb 的 header_room 中是否有足够的空间存放 IP 协议头，在初始化 skb 中管理
   //IP 层协议头地址指针 network_header 和 iph 指针
   skb_push (  skb,  sizeof ( struct  iphdr )  +  ( opt ? opt->optlen : 0 ) );
     skb_reset_network_header ( skb );
     iph = ip_hdr ( skb );
   //初始化用于构建 IP 协议头的 iph 数据结构的各数据域：version、ihl、ToS、//DF/DM、TTL、协议、源 IP 地址、目的
IP 地址
     *( (__be16 *) iph ) =  htons ( ( 4 << 12) | (5 << 8) | (inet->tos & 0xff ) );
     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;
   //如果没有 IP 选项，则在 IP 协议头中构建 IP 选项，产生数据包标识符 ID
     if (opt && opt->optlen ) {
          iph->ihl += opt->optlen >> 2;
          ip_options_build( skb, opt, inet->daddr, rt, 0 );
     }

     ip_select_ident_more ( iph, &rt->u.dst, sk,
                    ( skb_shinfo ( skb )->gso_segs ?: 1) - 1);

     skb->priority = sk->sk_priority;
     skb->mark = sk->sk_mark;
     return ip_local_out(skb);
```


当 ip_queue_xmit 收到 skb 时，skb->data 指针指向网络层负载数据的起始地址处，即传输层写完了数据后的位置；网络层的协议头应放在该指针所指位置的前面，所以，skb_push 在这里用于将 skb->data 指针向前移动，指向网络层或 IP 协议头的起始地址，skb->network_header 指针和 iph 也初始化为指向该区域。初始化 skb->data 指针是为了存放 IP 协议头的预留空间，如图 7-19 所示。

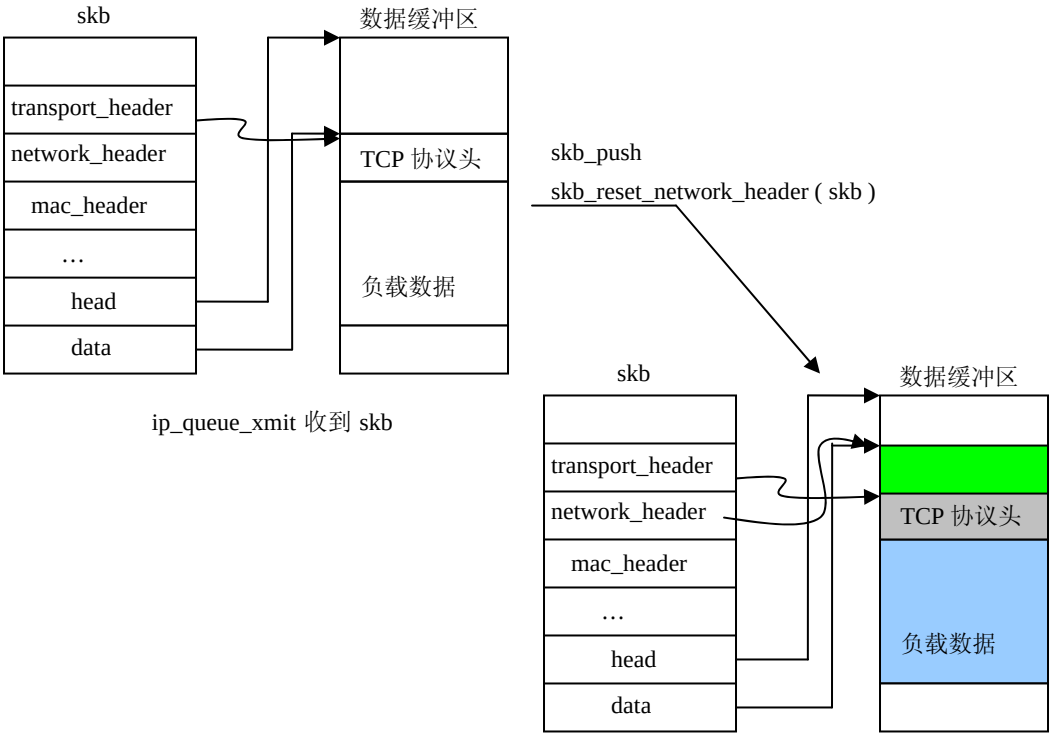


图 7-19　为 IP 协议头预留空间

接下来的代码初始化 IP 协议头一系列的数据域，初始化 IP 协议头的部分值来自 skb，有的来源于 rt。

- 为 IP 协议版本 version、 协议头长度 ihl 和服务类型 ToS 3 个数据域分配值，它们共享一个 16 位的字，语句将 IP 协议版本设置在字的高 4 位，将协议头长度设置在其后的 5 位，服务类型的值来自 inet -> tos。
- 是否允许对数据包进行分段标志 DF/MF、数据包生存时间的 ttl 值、源地址、目标地址等。
- 如果 IP 协议头包含选项，调用 ip_options_build 来构造协议头的选项数据块，ip_options_build 用 opt 局部变量（先前由 inet->opt 初始化）中的值与标志，将选项中要求的值加到 IP 协议头中。需注意的是，ip_options_build 的最后一个参数设置为 0，指明协议头不属于分片数据。
- ip_select_ident_more 函数根据是否对 IP 数据包进行分片，在 IP 头中设置 IP 数据包的标识符 ID。
- skb->priority 是由流量控制子系统，用于决定数据包应放到网络设备的哪个输出队列中等待发送，这也可以间接地确定数据包要等多久才可以被发送出去。该函数中的值是从 struct sock 数据结构中获取的，而在 ip_forward 中（其数据不是本地数据，因此没有本地套接字），它的值是从一个基于 IP ToS 值的转换表得来的。

3．函数结束处理

由 ip_local_out 调用__ip_local_out， ip_send_check(iph)计算校验和。

最后在__ip_local_out 中调用网络过滤子系统的回调函数，来查看数据包是否有权限跳到下一个步骤（dst_output）继续发送。

```
return nf_hook(PF_INET, NF_INET_LOCAL_OUT, skb, NULL, skb->dst->dev,dst_output);
```

7.6.4 ip_append_data 函数预备

这个函数由传输层的协议调用，缓存要发送的数据。ip_append_data 函数不传送数据，而是把数据放到一个大小适中的缓冲区中，随后的函数来对数据包进行分段处理（如果需要），并将数据包发送出去，它不创建或管理任何 IP 协议头。数据缓冲到一定程度，为了将由 ip_append_data 缓冲的数据包发送出去，传输层协议处理函数需调用 ip_push_pending_frames，由 ip_push_pending_frames 函数来维护 IP 协议头。

如传输层协议要求快速地响应时间，可以在每次调用 ip_append_data 函数后，立即调用 ip_push_pending_frames 函数将数据包发送出去。这两个函数的主要作用是，使传输层协议可以尽可能多地缓存数据（直至达到 PMTU 的大小），然后一次性发送，这样发送的效率更高。

1．ip_append_data 的主要功能

如果前所述，ip_append_data 函数是由网络层实现的，提供给传输层协议调用，来缓冲发送数据的函数，它需要完成的主要功能如下：

（1）缓存传输层协议发送来的数据段

将从传输层传送来的数据放在缓冲区中，这些缓冲区的组织方式、分配的容量大小，在需要进行 IP 数据包分割时，使对 IP 数据包的分割实现更容易。包括将那些数据段以某种方式放入缓冲区，其后网络层和数据链路层很容易在其中加入它们的协议头。

（2）优化内存分配

函数在分配缓存数据包的内存时，将 TCP/IP 协议栈的高层协议从应用程序套接字获得的发送数据包控制信息和输出网络设备的能力都考虑进去，特别是：

- 短时间内有多个数据段要缓存。

如果上层协议给出的标志显示在短时间内有更多的数据包需要发送（通过 MSG_MORE 标志），ip_append_data 函数可以为此分配一个较大的缓冲区，当数据从上层协议传送来时可直接放入缓冲区中，不必再申请内存分配。

- 硬件是否支持 Scatler/Gather I/O 功能。

如果输出设备支持 Scatler/Gather I/O 功能，分段操作可以安排在内存页面中，以优化内存的处理。

- 处理传输层校验和。

如何处理传输层的校验和由 skb -> ip_summed 数据域说明，该数据域的初始化基于输出网络设备的能力（硬件是否可以计算校验和）和一些其他因素。

2．ip_append_data 函数原型与输入参数

与 ip_queue_xmit 相比，ip_append_data 的工作更复杂，它的原型也更复杂，ip_append_data 函数的原型如下：

```
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 **rtp,
          unsigned int flags)
```

由于 ip_append_data 函数的功能较复杂，在分析 ip_append_data 函数的内部实现以前，首先要明白 ip_append_data 函数各输入参数的含义，运行产生什么输出结果，再来看函数的内部实现会更容易理解。

- sk：与该数据包发送绑定的套接字，即发送数据包是由哪个套接字创建。这个数据结构中包含了一些参数（如 IP 选项），以后需由 ip_push_pending_frame 函数填入到 IP 协议头中。
- getfrag：是传输层协议实例各自实现的函数，用于将传输层协议从套接字收到的负载数据复制到缓冲区中。getfrag 函数中的参数指明负载数据的来源地址、复制到缓冲区的目标地址、数据长度等信息。
- from：指向传输层要发送递给网络层缓存的数据（payload）起始地址，这个地址可以是一个内核地址空间的指针，也可以是一个用户地址空间的指针。
- length：发送数据的总长度（包括传输层的协议头和负载数据）。
- transhdrlen：传输层协议头的大小。
- ipc：正确前送数据包需要的信息。包括目标地址、输出网络设备和输出数据包的 IP 选项。
- rtp：与数据包发送相关的路由在路由表高速缓存中的记录入口，ip_queue_xmit 自己提取该信息，ip_append_data 函数依赖其调用函数即 ip_route_output_flow 收集该信息。
- flags：该变量中包含 MSG_XXX 形式的标志，定义在 include/linux/socket.h 中，其中有 3 个值该函数会使用，如下：
  - MSG_MORE：由应用程序使用，告诉传输层层协议短时间内有更多的数据包要发送，这个标志再传给网络层，随后我们会看到，在分配内存时会应用这个值。
  - MSG_DONTWAIT：当设置了这个标志时，对 ip_append_data 的调用不能被阻塞，在需要为套接字数据结构 sk 分配内存时，应调用 sock_alloc_send_skb 函数，如果 sock_alloc_send_skb 的资源已耗尽，ip_append_data 的运行可以在堆中按时间阻塞；在阻塞超时之前堆中通常有部分内存会变为有效，套接字数据结构获取需要的内存，ip_append_data 函数继续执行；或者超时后堆中仍无有效内存供套接字使用，ip_append_data 执行失败。这个标志就是在阻塞等待执行和直接失败退出之间做选择。
  - MSG_PROBE：用户设置这个标志，是为了探测发送路径的信息，如该标志可用在探测给定的 IP 地址路径上的 PMTU。如果设置了这个标志，ip_append_data 立即返回成功标志。

ip_append_data 函数是一个又长又复杂的函数，定义了大量的与变量名非常类似的局部变量，代码跟踪较其他函数更困难。我们将其按主要步骤划分成几段，下面先描述 ip_append_data 应给出的输出，然后是其初始化任务，最后是它的主循环，这样来分析函数实现数据缓存的机理。

在 ip_append_data 函数中使用了大量的局部变量，图 7-20 给出了局部变量在函数中的应用。

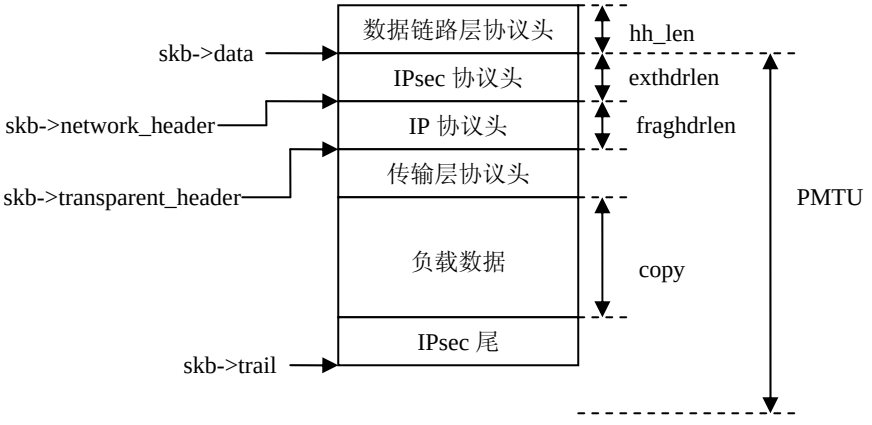


图 7-20　ip_append_data 函数中使用的局部变量

3．ip_append_data 对缓冲区的组织

ip_append_data 最终的输出指向缓冲区的指针，是数据段存放在内存的缓冲区。了解从 ip_append_data 函数输出的数据段如何在内存中转换成 IP 数据包非常重要，即内存如何分配和组织。接下来我们将着重介绍管理输出数据的数据结构，以及如何使用这些数据结构。

传输层协议传送给 ip_queue_xmit 的数据组织与此相同，ip_queue_xmit 是 TCP 用于代替 ip_append_data 函数的，无论传输层协议使用哪个函数传送数据段给 IP 层，缓冲区最终都由 dst_output 处理，下面来看看实际中可能出现的几个情况。

（1）数据长度小于路由最大传输单元

ip_append_data 可以创建一到多个 sk_buff 实例，每一个代表不同的 IP 数据包（或 IP 分片数据）。假设要发送的数据包总的大小在 PMTU 范围之内（即不需分段），同时假设由于主机的配置，我们需要使用 IPsec 协议栈中的一个协议，最后为了简化过程假设在内存分配时不考虑优化，这时 ip_append_data 的结果如下：

- 因不需分段，我们只需分配一个缓冲区。
- 使用了 IPsec 协议栈的协议，它需要在数据包外加头和尾，包裹在常规缓冲区外，在分配内存和从传输层复制数据到缓冲区时，需要考虑有 IPsec 协议头的情况。
- 我们也需要为数据链路层的协议头预分配内存。

这里我们给传输层以后要使用的所有协议都预留空间，同时指向协议头的某些指针（如 transparent_header 和 network_header）也被初始化，随后相应的协议可以填入其协议头在数据包中预留的空间中，由 ip_append_data 填入缓冲区的部分只是传输层的负载数据，其他部分按以下方式填入：

- 传输层协议头将由 ip_push_pending_frames 函数来填入，该函数可以被直接调用或通过包装函数调用（如 UDP 使用的就是包装函数：udp_push_pending_frames）。
- 网络协议头（包括 IP 选项）由 ip_push_pending_frame 函数来填入。

（2）数据长度大于路由最大传输单元

接下来我们以数据需要分段的情况来看内存的分配和组织。在第一种情况的基础上，数据包中不使用 IPsec 协议，扩大负载数据的长度，让其超过 PMTU，图 7-21 给出了这种状况的示例。

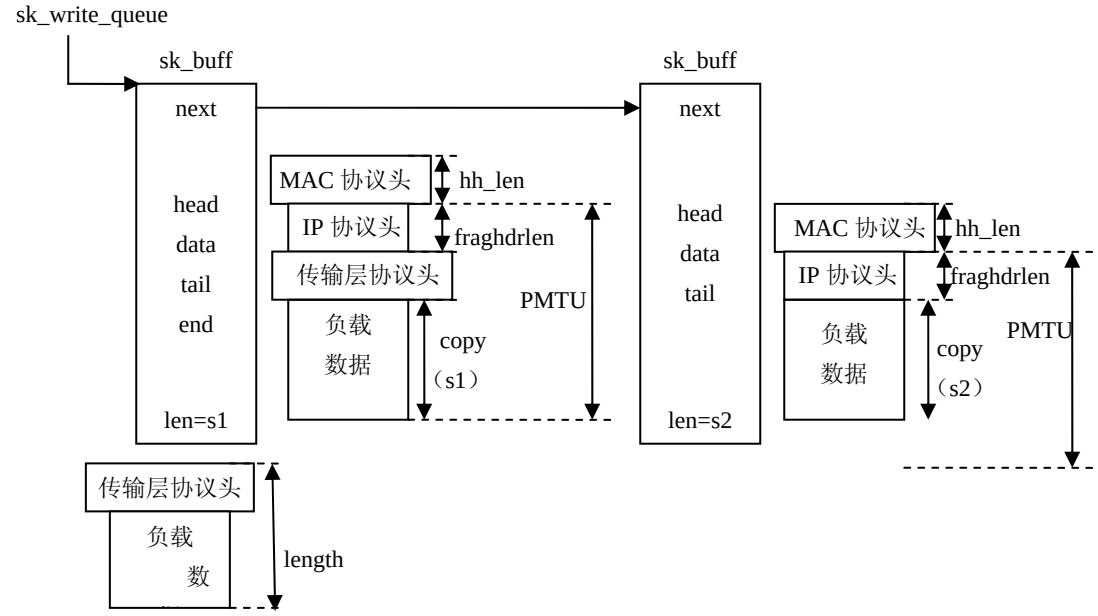


图 7-21 传送数据段大于 PMTU 时的内存组织

在图左下方的对象是 ip_append_data 收到的输入缓冲区，长度为 length。由于 ip_append_data 收到的缓冲区长度大于 PMTU，因此 ip_append_data 需要分配两个缓冲区，在图的上方。

ip_append_data 函数向系统申请缓冲区时，基于 PMTU 的大小来创建缓冲区的数量。如在上图中第一个缓冲区中包含的数据段与 PMTU 的大小一样，在第二个缓冲区中存放余下的数据。因为余下的数据长度小于 PMTU，第二个缓冲区的长度也小于 PMTU。

（3）传输层多次调用 ip_append_data 函数追加数据时的内存组织

前面说过，ip_append_data 不会发送任何数据，它只是创建缓冲区，为以后数据分段做好准备。在数据没有最终发送出去之前，传输层协议都可以再次调用 ip_append_data 函数向缓冲区追加或发送数据。现在我们在第二例的基础上来看在传输层多次调用 ip_append_data 函数向缓冲区追加数据时，如何组织内存才能达到最佳的发送性能。

在（2）中，ip_append_data 分配内存的方式是为余下的数据分配大小适中的缓冲区，在传输层协议多次调用 ip_append_data 函数来缓存数据段的情况下，且第二个缓冲区已满，当传输层协议追加数据时，ip_append_data 函数被迫分配一个新缓冲区，最优化的情况是除了最后一个缓冲区外，所有数据段的长度都达到 PMTU。

为了达到缓存区优化利用，最简单的解决方法是，分配另一个长度等于 PMTU 的缓冲区，从第二个缓冲区中将数据复制过来，删除第二个缓冲区，把新数据合并到新缓冲区中。当新缓冲区中没有足够的空间时，再分配第三个缓冲区。但这种解决方法并不能提供好的性能，内存复制非常耗时，且这样做违背了一个原则：调用 ip_fragment 做数据分割前，避免额外的数据复制。

在这种情况下我们就需要使用 flag = MSG_MORE 标志了。以以上第二种情况为例，当传输层向网络层传送数据时，使用了 MSG_MORE 标志。IP 层 的 ip_append_data 函数知道很快会有再次对 ip_append_data 的调用传来新数据，就直接为第二个缓冲区分配最大的空间（长度=PMTU），这个过程产生的输出如图 7-22 所示。

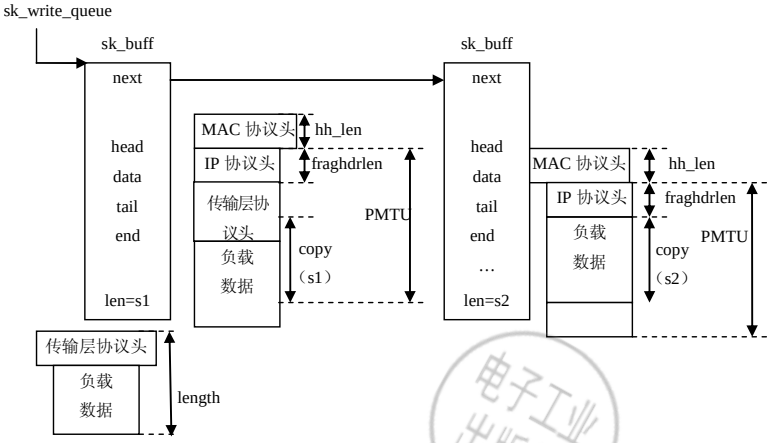


图 7-22　多次调用 ip_append_data 函数时的内存组

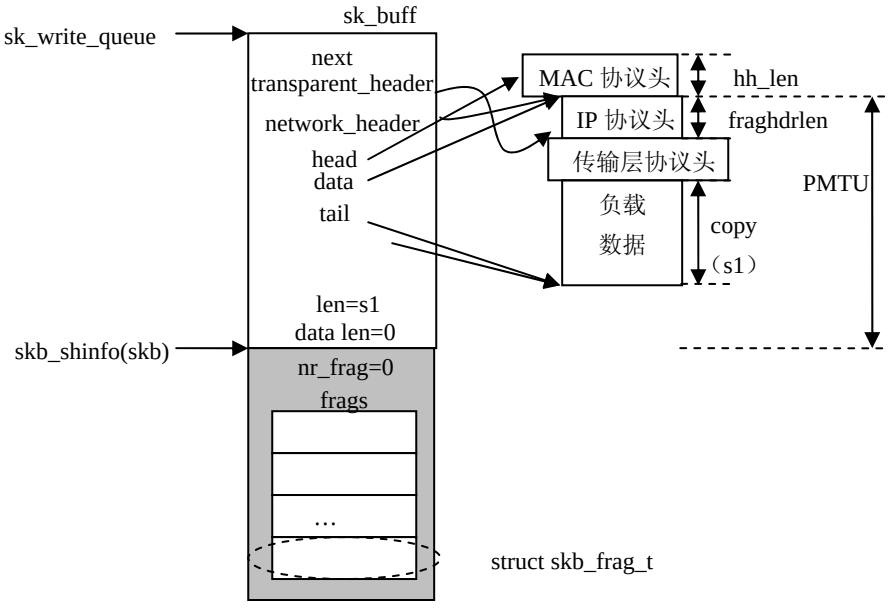
在调用 ip_push_pending_frames 把数据发送出去之前，传输层协议函数如果调用了 ip_append_data 向网络层追加数，ip_append_data 会首先填满第二个缓冲区剩余的空间，再分配第三个。

（4）网络设备支持 Scatter/Gather I/O 功能

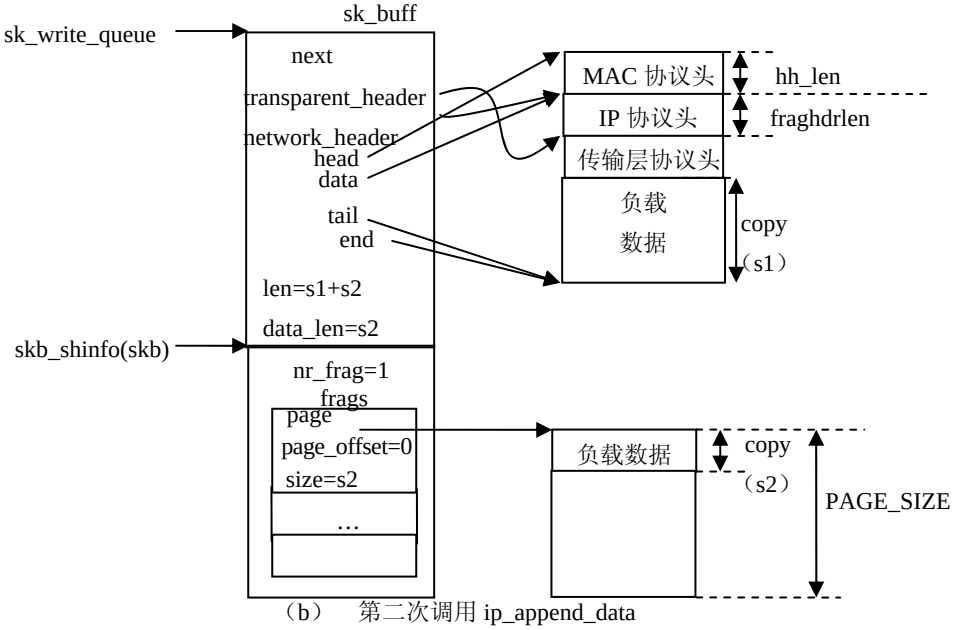
当网络设备支持分割/收集输入/输出（Scattter/Gather I/O）的发送功能时，传输层向网络层传送数据就无须将数据复制到网络层的缓冲区。数据可以留在传输层的地址空间，让设备结合这些缓冲区来发送数据。Scatter/Gather I/O 功能的优点是，它减少了内存分配和数据复制带来的额外开销。

考虑到这种情况，上层协议在连续操作过程中产生了很多小的数据条目，传输层可能将其存放在内核地址空间的不连续缓冲区中，然后要求网络层用一个 IP 数据包发送给它们。在网络设备不支持 Scatter/Gather I/O 功能的情况下，网络层必须把所有的数据条目复制到一个新的缓冲区中，形成一个完整的数据包。但如果网络设备支持 Scatter/Gather I/O 功能，则数据在离开主机之前就可以存放在产生它们的地方。

当设备支持 Scatter/Gather I/O 功能时，由 skb->data 指针指向的存放数据的内存区域只在第一次使用，接下来数据块就被复制到内存页面，该内存页面是专门为传输层向网络层追加数据分配的，图 7-23 和图 7-24 给出了 ip_append_data 函数在网络层接收传输层协议传送来的数据时，网络设备支持 Scatter/Gather I/O 功能和不支持 Scatter/Gather I/O 功能存在的不同。



（a）第一次调用 ip_append_data



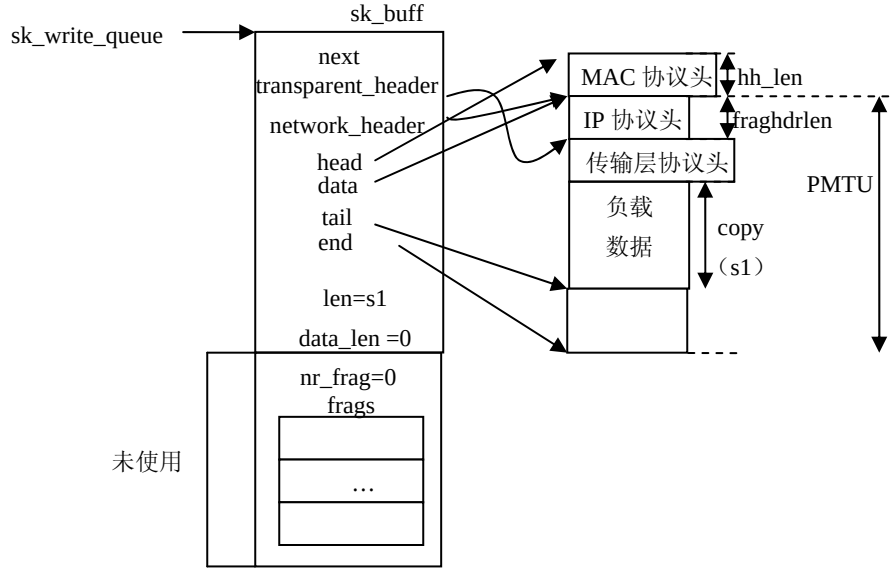
（b）　第二次调用 ip_append_data

图 7-23　网络设备支持 Scatter/Gather I/O 功能

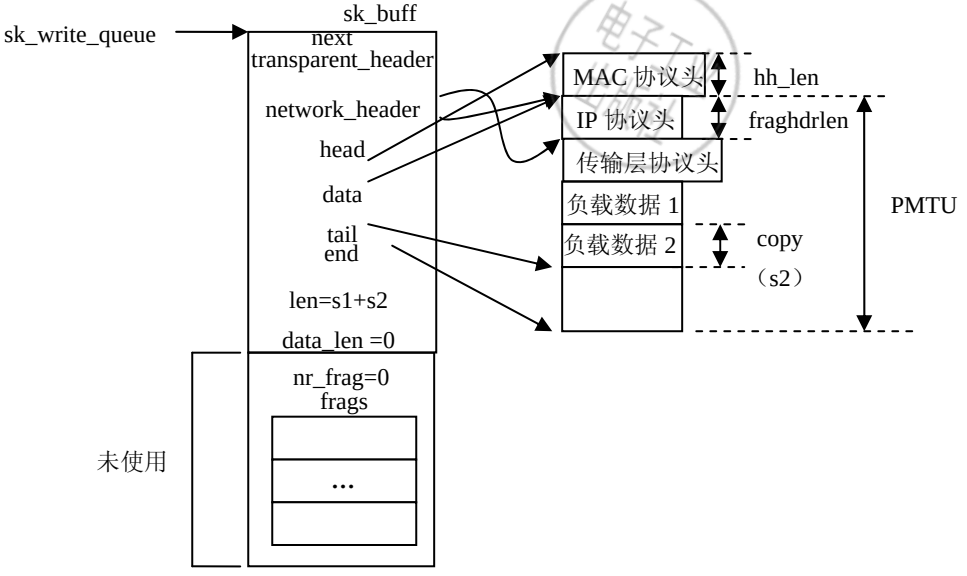
图 7-23（a）给出了第一次调用 ip_append_data 后内存的使用情况，7-23（b）给出了第二次调用 ip_append_data 后内存的使用情况。这时由传输层协议向网络层追加的数据存放在页面缓冲区中。当设备支持 Scatter/Gather I/O 功能时，由这些小数据片（frags）使用的缓冲区就是页面缓冲区。注意图 7-23（b）中的数据段不需要任何协议头。sk_buff 数据

结构管理的所有数据段都属于同一个 IP 数据包。

（5）网络调备不支持 Scatter Gather I/O 功能



（a） 第一次调用 ip_append_data



（b） 第二次调用 ip_append_data

图 7-24 网络设备不支持 Scatter/Gather I/O 功能

图 7-24(a)给出了禁止 Scatter/Gather I/O 功能时，传输层协议第一次调用 ip_append_data 函数后内存的使用情况，7-24（b）是第二次调用 ip_append_data 后内存的使用情况。

（6）使用页面缓冲区

要支持 Scatter/Gather I/O 功能需要一些附属数据结构，除了第一个缓冲区以外（无论有无 Scatter/Gather I/O 支持，第一个缓冲区都以相同的方式分配）。随后的数据段都存放在

skb_shinfo(skb)->frags 数据结构中。我们需要回顾第 2 章介绍的 sk_buff 数据结构的内容，每个 sk_buff 数据结构后紧跟一个 skb_shared_info 类型的结构。skb_shared_info 通过宏 skb_shinfo 来访问。这个结构可以通过向链表中增加新的存储区域来扩展缓冲区的大小，这些存储区域可以驻留在内存的任何位置，skb_shared_info 的 nr_frags 数据域帮助 IP 层记住，在一个 IP 数据包中有多少个 Scatter/Gather I/O 缓冲区。注意，nr_frags 数据域记录的是 Scatter/Gather I/O 的缓冲区数，而不是 IP 的数据包数。

接下来我们需要了解的是为什么在内核使用这类页面缓冲区时，设备上需要特殊的支持。为了能引用在内容上连续但在内存区域的存储位置不连续的空间，设备必须能处理这类缓冲区。图 7-25 给出了一个最简单的例子，图中只有一个页面包含了两个数据片的内存区域，无论是在同一页面还是在不同的页面，数据片在内存中的存放位置可以不相邻。

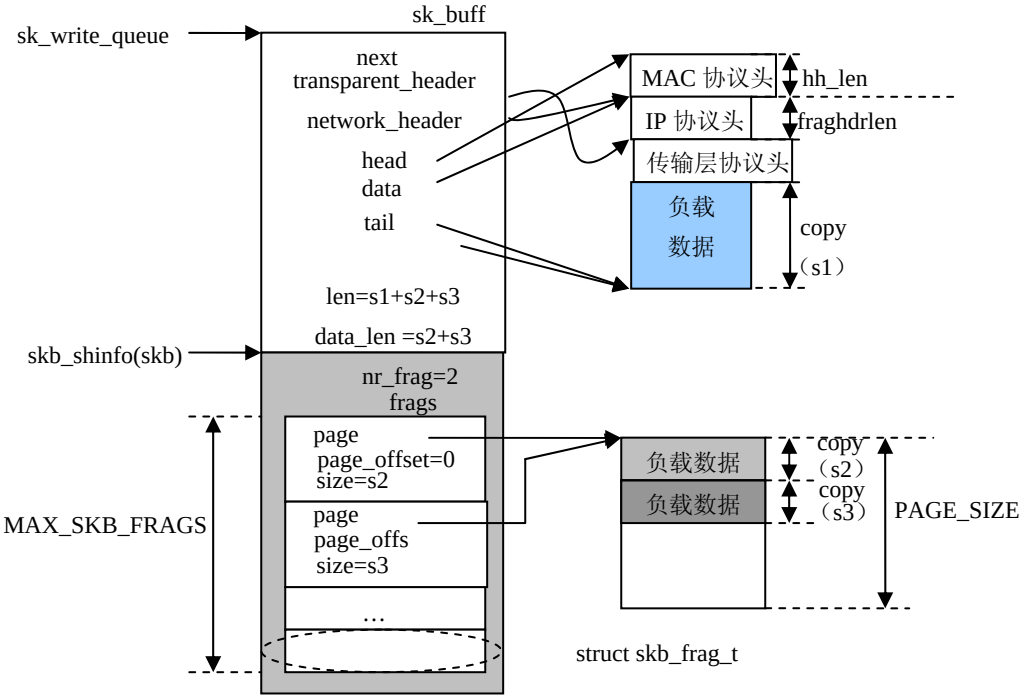


图 7-25　使用页面缓冲区时的内存组织

frags 数组中的每个成员都是一个 skb_frag_t 结构类型的实例，skb_frag_t 结构的数据成员有如下几种。

- page 指针：指向数据片所在的页面缓冲区的起始地址。
- page_offset：数据片在页面缓冲区中相对于页面起始地址的偏移量。
- size：数据片的大小等。

图 7-25 中的两个数据片驻留在同一个页面缓冲区中，它们的页面（page）指针指向同一个页面。数据片的最大数量是 MAX_SKB_FRAGS，该值是基于 IP 数据包的最大长度（64KB）和内存中页面的大小（i386 默认为 4KB）来定义的。

图 7-25 给出的是只有一个页面的情况，实际发送过程中可以有多个页面，frags 数组成员的 page 指针指向各数据片自己所在的页面，一个数据片不能跨两个页面，当一个新的数

据片的大小比当前页面的剩余空间大时，数据片被分成两个部分，一个填充到现有页面的空余空间中，另二个放入新的页面。

Scatter/Gather I/O 功能是独立于 IP 数据段的，其功能，只允许代码和硬件工作在不连续的内存区域中，从逻辑上认为它们是连续的；除此之外，每个数据片的大小受 PMTU 值的限制，即使 PAGE_SIZE 的默认值或设定值比 PMTU 大。当 sk_buff->data 指向的缓冲区中的数据加上 frags 数组引用的页面缓冲区中的数据大于 PMTU 时，会创建一个新的 sk_buff，重复以上过程。

（7）不同数据段的数据片处于同一页面缓冲区

同一个页面中可以存放不同 IP 数据段的分片数据，如图 7-26 所示，当不同 IP 数据段的分片数据加到页面中时，需对页面引用计数加 1。IP 数据片发送出去后，释放在页面中的数据片，同时对页面的引用计数递减。最后当页面的引用计数为 0 时，释放页面。

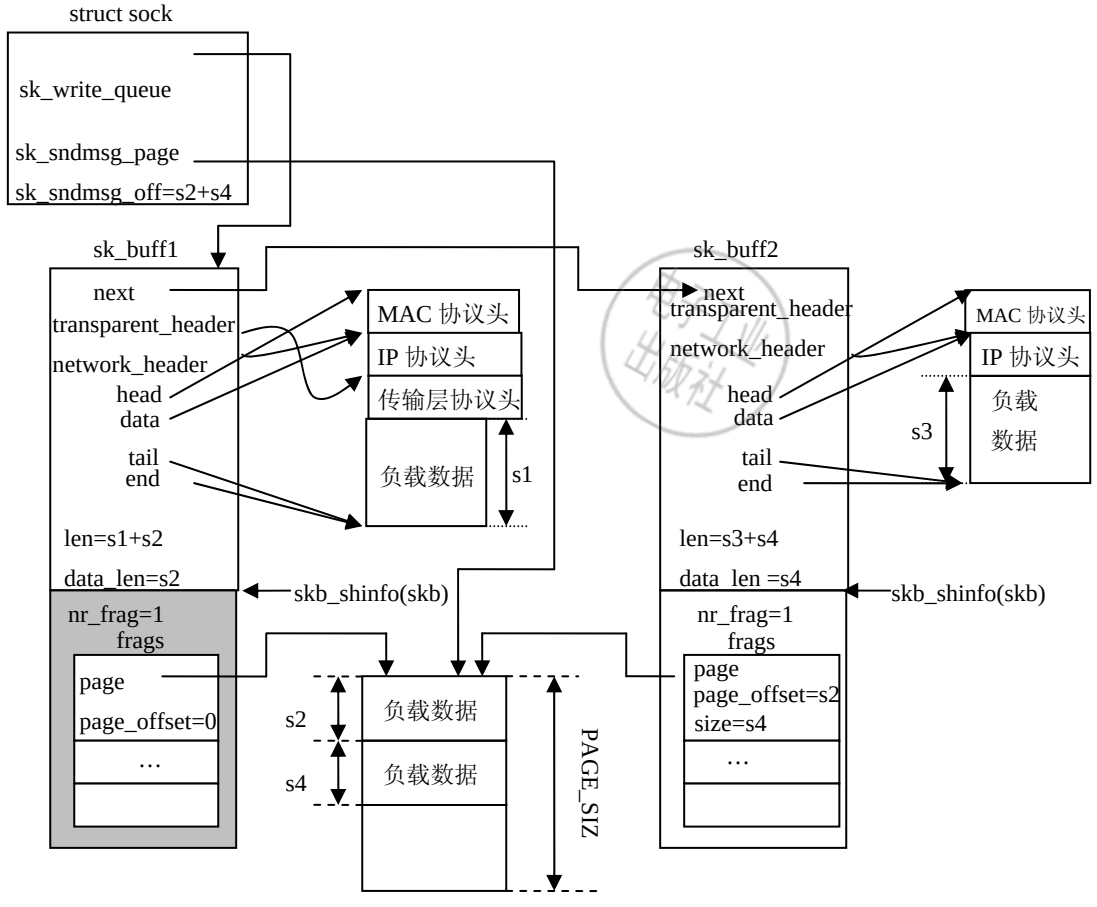


图 7-26　不同 IP 数据段的分片数据处于同一页面缓冲区

图 7-26 中顶部的 struct sock 数据结构包含了两个指针：指向最后一个放数据片的页面（page）的指针（sk_sndmsg_page）和数据片在页面中的偏移位置（sk_sndmsg_off），这样在传输层协议下一次调用 ip_append_data 时，指明了下一个数据片应放在哪里。

4．处理数据片缓冲区的关键例程

为了理解我们将要分析的 ip_append_data 函数功能，现在需要回顾在第 2 章中介绍的操作缓冲区（sk_buff）的关键函数和其他一些函数。

（1）skb_is_nolinear

当 Socket Buffer 有分片数据在页面缓冲区时，返回 true（skb->data_len 非空）。

（2）skb_headlen

给定一个包含了分片数据的缓冲区 sk_buff，返回在主缓冲区中数据的大小。skb->data 指定数据缓冲区的大小（它既不计算 frag 数据片，也不计算 frag_list 中的数据）。不要把 skb_headlen 和 skb_headroom 混淆，skb_headroom 返回 skb->head 和 skb->data 之间的空余空间。

（3）skb_pagelen

分片数据缓冲区的总长度，计算主缓冲区中的数据（skb_headlen）大小和 frags 页面缓冲区中保存的数据片的总和。但它不计算任何链接到 frags_list 链表中的数据，注意 skb->len 包括 frags 中的分段数据和 frags_list 中的数据。

这里再对 frags 数组做进一步的说明：frags 数组中的数据是主缓冲区（skb->data）数据的延伸。frags_list 中代表的是另一些独立的，由 sk_buff 数据结构管理的数据包的分片数据（各自会作为独立的 IP 分片发送）。

5．ip_append_data 函数的输出：IP 数据包队列

无论什么时候，当 ip_append_data 函数分配了一个新的 sk_buff 数据结构来处理新的 IP 数据包时，ip_append_data 将这个 sk_buff 放在套接字 sk 管理 IP 数据包的输出队列中（sk_write_queue）。如图 7-24 所示，可以看到 sk_write_queue 队列就是所有要发送的 sk_buff 缓冲区链表。sk_write_queue 由 ip_append_data 的输入参数 struct sk 寻址。

在 ip_append_data 运行结束时，sk_write_queue 队列是 ip_append_data 函数的输出，随后的函数只需在其前面加入 IP 协议头，把它们放到数据链路层，由 dst_output 发送出去。

sk_write_queue 列表是先进先出（FIFO）队列，其管理特性为：

（1）新成员放在队列尾

它遵从的原则是，队列中的第一个成员包含额外的协议头，如 IPsec（如果有）和传输层协议头或部分传输层协议头，在 PMTU 相对较小的情况下。

（2）何时创建新成员

新成员只有在 sk_write_queue 中的最后一个缓冲区的长度达到了最大值时才会被创建并加到队列中（这里 frags->size 的意思是指作为数据包中一部分的数据，是图 7-24 中灰色的部分，它不是页面缓冲区的大小，分配的页面缓冲区的大小可能比放入的有效数据大，这是因为 ip_append_data 绝不会创建一个比路由的 PMTU 大的数据片，当使用 Scatter/Gather I/O 功能时，新的数据片存放在内存页面中，而不是存放在 sk->data 所指的区域中）。

现在我们知道了 ip_append_data 函数产生的输出是 sk_write_queue 队列，就可以来分析函数代码的实现了。记住：传输层在将缓冲区用 ip_push_pending_frames 发送出去之前，可以多次调用 ip_append_data 函数来向页面缓冲区追加数据。

假设 UDP 发出了 3 次 ip_append_data 函数调用，其传递的负载数据大小分别是 300、

250 和 200 个字节。同时假设 PMTU 是 500 个字节；首先需要清楚的是，如果 UDP 是要将 750 个字节作为完整负载数据包发送，IP 层会创建两个数据段，第一个数据段大小为 500 个字节，第二个数据段为 250 个字节。但应用程序也可能想使用 UDP 套接字发送 3 个独立的 IP 数据包，大小分别为 300、250 和 200 个字节，它需要告诉 ip_append_data 如何处理这些数据包。如果 UDP 的上层应用希望获取高的发送性能，则它可以指定 MSG_MORE 标志告诉 ip_append_data 创建最大的分段。ip_append_data 的执行结果是：

第一个数据段是 500 个字节，第二个数据段是 250 个字节。如果应用不指定 MSG_MORE 标志，UDP 就独立地传每个负载数据。

7.6.5 ip_append_data 函数分析

1．设置 ip_append_data 函数的执行现场

ip_append_data 的第一部分是初始化一些局部变量，如果需要还会改变一些输入参数的值，实际要完成的任务判断函数是否正在创建 sk_write_queue 队列的第一个 IP 数据段，还是向 sk_write_queue 队列中加入数据包的后续缓冲区？当创建 sk_write_queue 队列中的第一个 IP 数据段时，需要清空 sk_write_queue 队列。

创建 sk_write_queue 队列中的第一个成员，ip_append_data 初始化过程需要为 inet->cork 和 inet 数据结构的部分数据域赋值。这些数据域在 ip_append_data 函数及 ip_push_pending_frames 函数完成数据分割成数据片、创建 IP 协议头时会用到。

1）创建 sk_write_queue 队列第一个 IP 数据段的函数初始化过程

```
if ( skb_queue_empty( &sk->sk_write_queue ) ) {
/*输入参数 ipc 中包含了发送输出数据包时需要的所有信息。如果有 IP 选项，则将 IP 选项复制到 struct cork 数据结构的
IP 选项数据域，并设置有 IP 选项标志、数据包最终目标地址*/
opt = ipc->opt;
if (opt) {                                    //如有 IP 选项
  if (inet->cork.opt == NULL) {               //为 inet->cork.opt 分配内存
     inet->cork.opt = kmalloc(sizeof ( struct ip_options) + 40, sk->sk_allocation);

     if ( unlikely (inet->cork.opt == NULL))
        return -ENOBUFS;
  }
  //复制 IP 选项到 inet->cork.opt 数据域
  memcpy(inet->cork.opt,  opt,  sizeof ( struct ip_options)+ opt ->optlen );
  inet->cork.flags |= IPCORK_OPT;       //设置有 IP 选项的标志
  inet->cork.addr = ipc->addr;
}
rt = *rtp;
/*缓存路由信息到 struct cork 数据结构中，以便提高以后数据包的传送速度。初始化 struct cork 的其他数据域。数据段
所在页面和在页面的偏移位置。如果使用 IPsec 协议，则会加额外协议头，只有第一个 IP 数据段要加传输层协议头*/
*rtp = NULL;
inet -> cork.fragsize = mtu = inet->pmtudisc == IP_PMTUDISC_PROBE ?
        rt->u.dst.dev->mtu :
        dst_mtu( rt->u.dst.path );        //根据 PMTU 的值确定是否需要对数据分片
inet->cork.dst = &rt->u.dst;              //从路由表获数据包目标地址缓存在 cork.dst
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;
  }
  }
```

2）向 sk_write_queue 队列中加入后续页面缓冲区后初始化变量

在前面创建第一个 IP 数据段时，已初始化了 struct cork 和 struct inet_socks 数据结构的相关数据域，现在用这些数据域的值来初始化局部变量。

- rt：在路由表高速缓冲区中记录入口。
- mtu：发送路由上的最大发送单元。
- opt：存放 IP 选项的局部变量。

因为只有第一个 IP 数据段才需要加入传输层协议头和额外协议使用的协议头，后续的页面缓冲区不需要加入这些协议头，这时需将 transhdrlen 和 exthdrle 清零。

```
//向 sk_write_queue 队列中追加后续缓冲区，如果前面初始化 inet->cork.flags 表明有 IP 选项，则将 IP 选项复制到局
//部变量 opt 中
else {
  rt = ( struct  rtable *) inet->cork.dst;
  if ( inet->cork.flags & IPCORK_OPT )
       opt = inet->cork.opt;
  transhdrlen = 0;            //非第一个缓冲区，将传送层协议头长度变量清零
  exthdrlen = 0;              //同上，将额外协议头长度变量清零
  mtu = inet->cork.fragsize;
  }
```

随后 transhdrlen 的值可以用于帮助 ip_append_data 判断当前是否正在创建 sk_write_queue 队列的第一个 IP 数据段。

- transhdrle != 0 ， 意味着 ip_append_data 在处理 sk_write_queue 队列的第一个 IP 数据段。因为第一个 IP 数据段需要加入传输层协议头。
- transhdrle = 0，即 ip_append_data 不是在处理 sk_write_queue 队列的第一个 IP 数据段。

2．准备创建 IP 数据段

从传输层传来的每个 IP 数据段的大小不一样，在从传输层复制数据段到页面缓冲区前，ip_append_data 函数先初始化以下几个局部变量：

```
hh_len = LL_RESERVED_SPACE ( rt ->u.dst.dev );                    // 数据链路层协议头长度
/* IP 协议头的长度，当有 IP 选项时，包括 IP 选项的长度*/
fragheaderlen = sizeof(struct iphdr )+( opt ?  opt ->optlen : 0 );
maxfraglen=( ( mtu-fragheaderlen )& ~ 7)+fragheaderlen;           //一个数据段的最大长度
```

在把 IP 数据段放入缓冲区之前，ip_append_data 需要知道为数据链路层协议头预留多少空间。这样在网络设备驱动程序初始化数据链路层的协议头时，就不需重分配内存了。

接下来 ip_append_data 需要跟踪接收到的每个 IP 数据段的长度是否超过 64KB，因为 IP 数据包的最大长度是 64KB。最后初始化产生校验和的方式。

```
if  ( inet ->cork.length+length>0xFFFF-fragheaderlen ) {
ip_local_error (sk , EMSGSIZE ,  rt ->rt_dst,  inet ->dport,   mtu-exthdrlen );
  return  -EMSGSIZE;
}
```

3．实现 IP 数据段复制的函数 getfrag

ip_append_data 函数可由传输层的任何协议实例调用，来向网络层发送数据。它的一个重要任务就是，将数据从传输层复制到 ip_append_data 创建的缓冲区中。不同协议对数据的复制操作不一样。其中一个重要的不同之处就是，如何计算传输层的校验和（ICMP 协议不需要计算校验和），另一个不同之处就是数据的来源，数据是由本地用户地址空间产生的数据包，还是内核地址空间前送的数据包或直接由内核产生的数据包。

为了使操作更清晰，传输层中各协议都有自己的数据段复制函数；各传输层协议在调用 ip_append_data 函数时，将输入参数 getfrag 的函数指针初始化为自己的数据复制函数，也即 ip_append_data 使用的 getfrag 将输入数据复制到缓冲区中。

表 7-5 列出了传输层最常用协议调用 ip_append_data 函数时，使用的数据复制函数。

表 7-4　传输层协议使用的 getfrag 例程

协议	数据复制函数
ICMP	icmp_glue_bits
UDP	ip_generic_getfrag
裸 IP	ip_generic_getfrag
TCP	ip_reply_glue_bits

getfrag 函数接收 4 个输入参数：from、to、offset 和 len。getfrag 的功能就是从 from 指针所指地址处复制 len 个字节到 to+offset 目标处。from 可以是用户地址空间的指针，如果 from 是用户地址空间的指针，必须做相应处理（如从用户地址空间转换到内核地址空间），最后 getfrag 计算传输层的校验和，按 skb->ip_summed 的配置计算校验和并放入 skb->csum 数据域。

4．缓冲区分配

在 ip_append_data 函数将传输层发送来的数据复制到网络层之前，ip_append_data 函数需为接收这些数据分配缓冲区所需内存，缓冲区的组织方式在前面已经介绍了，这里就不再赘述。ip_append_data 函数基于以下因素分配缓冲区：

（1）单次调用发送或多次调用发送

如果 ip_append_data 函数获知在短时间内传输层有多次数据发送到达（设置了 MSG_MORE 标志），ip_append_data 函数会分配一个更大的缓冲区，这样以后传来的数据可以合并到已分配的缓冲区中，不需要再次申请内存分配。

（2）网络设备支持 Scatter/Gather I/O 功能

如果网络设备支持 Scatter/Gather I/O 功能（设备驱动程序初始化时设置 dev->features 数据域），各数据片就可以存放在内存页面。以下代码段就是按以上两条原则来确定分配的缓冲区大小。

```
//传输层会多次调用传送数据，网络设备不支持 Scatter/Gather I/O 功能

if ( (flags & MSG_MORE) &&
  !( rt ->u.dst.dev->features& NETIF_F_SG ) )
  alloclen = mtu;
else
//单次调用或网络设备不支持 Scatter/Gather I/O 功能
```

```
    alloclen = datalen + fragheaderlen;
  if ( datalen  ==  length + fraggap )
    alloclen += rt->u.dst.trailer_len;
```

这里需要注意的是，在 ip_append_data 函数生成最后一个数据段时，它需要考虑后缀（如果使用了 Ipsec，则它会在数据包尾加入 trailer）。

5．ip_append_data 函数的主循环流程

ip_append_date 函数的主循环流程图，如图 7-27 所示。

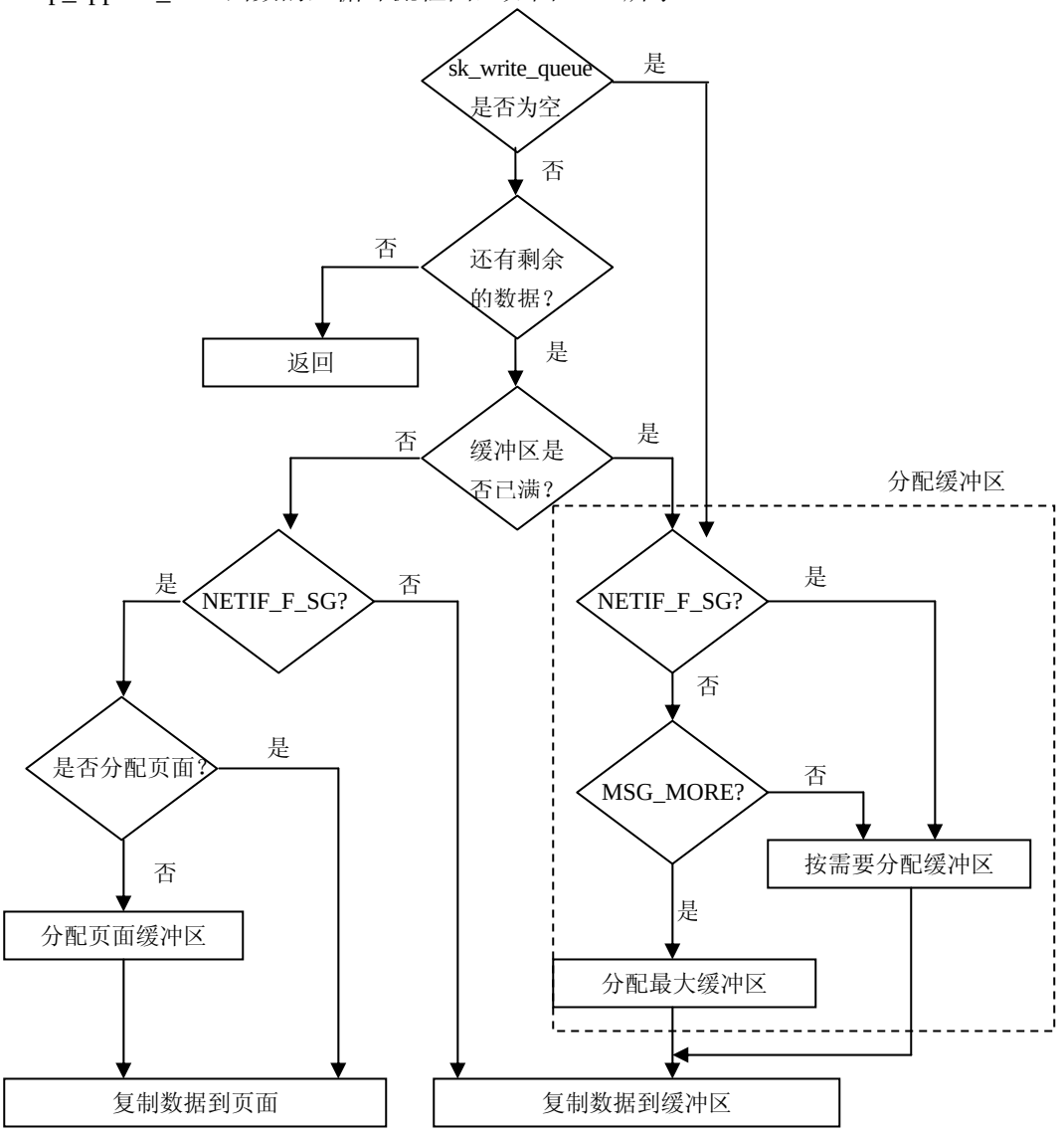


图 7-27　ip_append_data 函数的主循环流程图

最后我们分析 ip_append_data 函数主循环的执行流程，这是 ip_append_data 函数分配缓冲区，复制数据到缓冲区的实现所在。了解 ip_append_data 函数的主循环，则主要了解

以下几点，就可以抓住 ip_append_data 函数实现的关键。

（1）输入参数 length 的变化

最初 length 代表的是在传输层协议调用 ip_append_data 函数时，发送给它的数据的总长度。进入循环后，length 的值就代表未复制完的数据的长度。在每次循环结束处需要更新 length 的值，直到 length 值为 0，数据复制完成。

（2）MSG_MORE 标志

MSG_MORE 标志指明传输层会多次调用 ip_append_data 函数以向缓冲区中追加数据。

（3）NETIF_F_SG 标志

指明网络设备是否支持 Scatter/Gather I/O 功能。

这几个标志的设定，在最初 ip_append_data 函数创建 sk_write_queue 队列的第一个缓冲区时，不影响函数的任何处理功能。这时 ip_append_data 在分配和初始化 sk_buff 数据结构，因为第一个数据段总是复制到 sk_buff 区域。

（4）分配新的 sk_buff 数据结构

- ip_append_data 在以下条件满足时分配一个新的 sk_buff 数据结构，并把 sk_buff 放入 sk_write_queue 队列。
- sk_write_queue 队列为空时（即 ip_append_data 在处理 sk_write_queue 队列中的第一个数据段）。
- sk_write_queue 的最后一个成员的空间已填满时。

循环前的以下代码就是处理上述的第一种情况：

```
if ( ( skb = skb_peek_tail( & sk ->sk_write_queue ) ) == NULL)
  goto alloc_new_skb;
```


while 循环体内的第一部分处理上述第二种分配 sk_buff 的情况。它首先初始化拷贝为当前存放 IP 数据段的缓冲区中余下的空间长度：mtu-skb->len。如果未复制的数据 length 大于缓冲区中余下的空间拷贝，就需要分配一个新的 IP 数据段的缓冲区。

- copy> 0 时，即 skb（sk_write_queue 队列中的最后一个成员）的数据缓冲区中还有空间可存放数据。ip_append_data 首先利用这部分空间。如果余下的空间不够（即 length 比有效空间 copy 大），则循环回到起始处，这时循环将落入下一类型的处理。
- 当 copy = 0 时，这时 sk_write_queue 队列中最后 sk_buff 成员的数据缓冲区空间已填满，需要分配新的 sk_buff。这时 if 语句块中的代码实现分配一个新的缓冲区 sk_buff，将输入数据复制到缓冲区中，把新缓冲区 sk_buff 放入 sk_write_queue 队列。
- 当 copy< 0 时，这是前面一种情况的特例。当拷贝为负值时，意味着部分数据必须从 IP 数据段中移到一个新的 IP 数据段中。

每次在代码运行到循环体底部时，ip_append_data 函数需要将指针移到新数据要复制的位置（offset），更新剩余需复制的数据长度（length）。一旦将数据段 sk_buff 放入队列 __skb_queue_tail 中，如果还有数据没复制，则需要重新开始循环。

以上我们分几个层次介绍了 ip_append_data 函数的实现，在理解以上过程后，读者最好从头至尾将 ip_append_data 函数源代码阅读一遍，这样就能完整地理解传输层协议向网络层发送数据段的过程。

7.6.6　ip_append_page 函数

当数据发送请求来自用户地址空间，应用程序调用 sndmsg 来请求将数据从用户地址空间移动到内核地址空间时，这个复制是由 ip_append_data 函数的输入参数 getfrag 函数来完成的。

内核还为用户地址空间的应用提供了另一个接口 sendfile，它允许应用程序优化发送并复制数据。这个接口称为“零复制”TCP/UDP。

sendfile 接口只有在网络设备支持 Scatter/Gather I/O 功能时才能使用。这时 ip_append_data 的逻辑实现不需要复制任何数据（即用户要求发送的数据仍保存在其创建处）。内核只需将 frags 数组初始化指向接收数据缓冲区的位置，在必要的时候计算传输层的校验和。这种复制逻辑由 ip_append_page 函数实现。

一旦 ip_append_page 收到存放接收数据位置的 void* 指针，即指向用户地址空间内存页面的指针和数据在页面中的偏移量，这样它可以直接用收到的页面指针和偏移量来初始化 frag 数组的一个成员。

ip_append_page 和 ip_append_data 函数的唯一不同之处在处理 Scatter/Gather I/O 功能时，其代码如下：

```
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) {
  __wsum csum;
  csum = csum_page ( page, offset, len );
  skb->csum = csum_block_add(skb->csum, csum, skb->len);
}
```

当向页面加入一个新的数据段时，ip_append_page 函数首先看是否可以将新的数据段与页面中已有的数据段合并。这项检查由 skb_can_coalesce 函数完成。skb_can_coalesce 函数查看新的指针是否是页面中最后一个成员指针的结束位置。如果新数据段可以与页面已有数据段合并，则 ip_append_page 函数需要完成的工作就是更新页面中数据段的长度，即原长度加上新数据的长度。

如果不能合并，则 ip_append_page 函数就用 skb_fill_page_desc 初始化一个新的数据段。这时它需要用 get_page 对页面的引用计数加 1。这里必须对页面的引用计数加 1，是因为 ip_append_page 将它接收的页面作为输入，而这个页面可能有别的进程在使用。

目前只有 UDP 使用 ip_append_page 函数。TCP 不使用 ip_append_data 和 ip_push_pending_frames 函数，TCP 在 tcp_sendmsg 中实现了同样的逻辑。对于“零复制”接口，在 TCP 不使用 ip_append_page 时，在 do_tcp_sendpage 函数中实现了同样的逻辑，

TCP 调用 ip_queue_xmit 函数向网络层发送的 IP 数据段。TCP 协议与 UDP 协议不同的是，TCP 只有在网络输出设备硬件支持传输层的校验和计算时，应用程序才能使用“零复制”接口。

7.6.7 ip_push_pending_frames 函数

如本节开始所述，ip_push_pending_frames 完成 ip_append_data 和 ip_append_page 的后续工作。当传输层的协议决定要将由 ip_append_data 和 ip_append_page 缓存的数据缓冲区发送出去的时候就调用 ip_push_pending_frames 函数。

1．ip_push_pending_frames 的输入参数

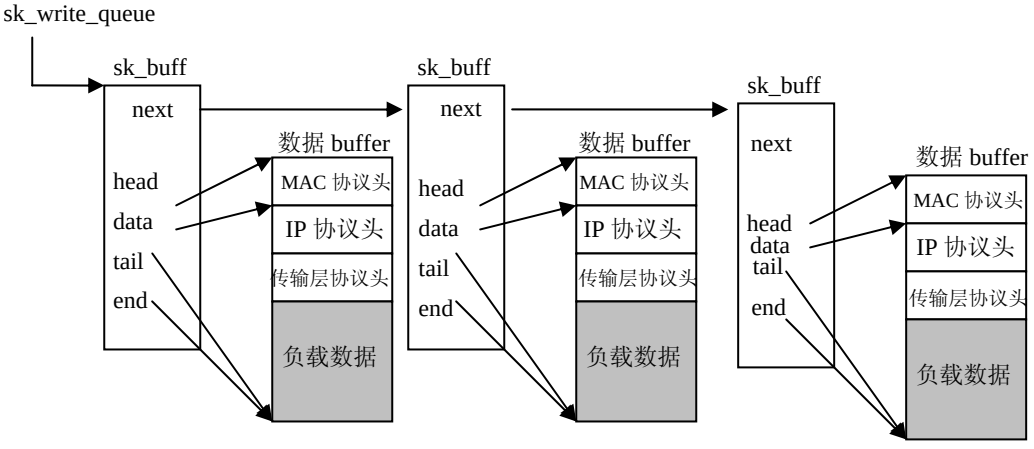
struct sock *sk 是 ip_push_pending_frames 函数唯一的输入参数，sk 是指向数据包所属的套接字的指针，sk 中包含了指向缓存数据包队列 sk_write_queue 数据结构的指针。

2．取发送缓冲区

ip_push_pending_frames 函数首先从 sk_write_queue 队列中取出缓冲的数据缓冲区。队列中第一个缓冲区一定是一个 sk_buff，地址由局部变量 skb 保存。队列中随后的缓冲区链接到 skb 的 frag_list 链表上，更新 sk_buff 的 len 和 data_len 数据域指明所有发送缓冲区的总长度 len，以及在 frag_list 中缓冲区的总长度。一旦缓冲区链接到 frag_list 后，就从 sk_write_queue 队列中清除。

将 sk_write_queue 队列中的缓冲区取出链接成一个新的链表的操作非常快。这里没有数据复制，只需移动指针，最后释放 sk_write_queue 队列，这样传输层又可以向网络层发送新数据。

ip_push_pending_frames 接收到的输入如图 7-28（a）所示，把缓冲区从队列中取下后如图 7-28（b）所示。



（a）在 sk_write_queue 队列中的缓冲区

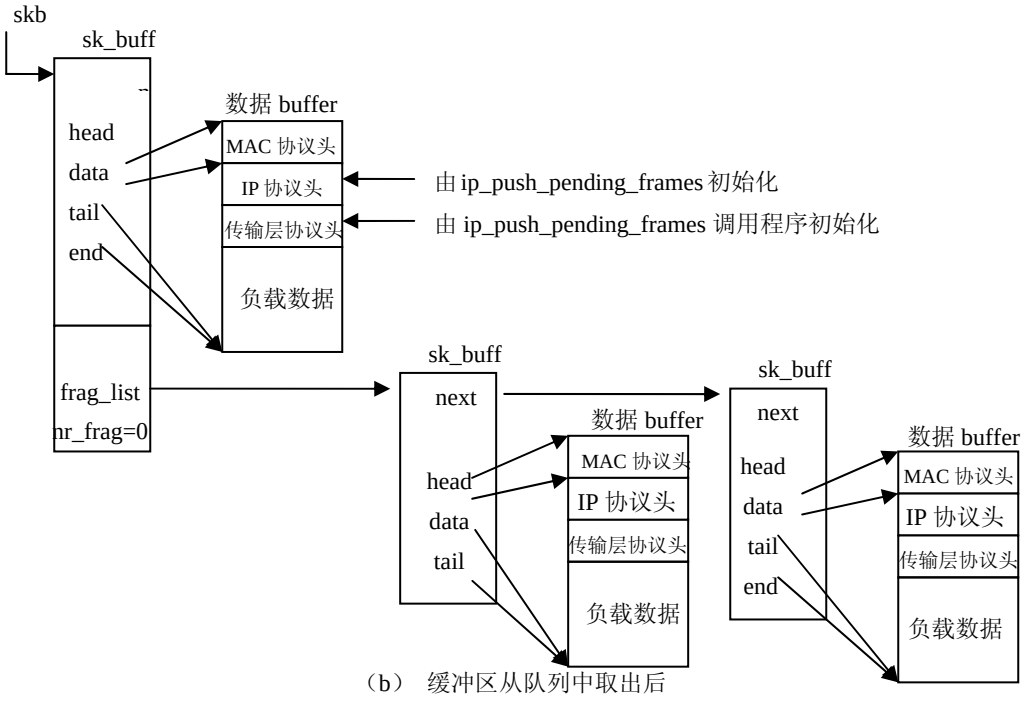


图 7-28　ip_push_pending_frames 函数处理队列缓冲区的示例

3．处理 IP 协议头

接下来应该向数据包中填写 IP 协议头。如果在配置套接字控制信息时，在 IP 协议头设置了不分割数据包的标志 IP_DF，且该标志作用于所有数据包：IP_PMTUDISC_DO；即使数据包的长度大于路由的最大传输单元 PMTU，也不分割数据包，就将 IP_DF 标志设置在 IP 协议头的相应数据域。

```
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);
```

如果数据包 IP 协议头中有 IP 选项，则获取 IP 选项。根据数据包的目标地址是组发送地址（RTN_MULTICAST）还是唯一主机地址，选定 iphdr->ttl 的值。接下来就设定一系列 IP 协议头的值：版本（version）、协议头长度（ihl），以及调用 ip_options_build 函数构建 IP 选项等。

```
/*如果有 IP 选项，则从输入参数中获取 IP 选项保存在局部变量 opt 中，根据数据包目标地址类型设置 ttl 值*/
if ( inet ->cork.flags & IPCORK_OPT )
  opt = inet ->cork.opt;
if ( rt ->rt_type  ==  RTN_MULTICAST )
  ttl = inet->mc_ttl;
else
ttl =  ip_select_ttl ( inet, &rt->u.dst );
/*设置数据包 IP 协议头的其他数据域，创建 IP 选项*/
iph = (struct  iphdr *  )skb ->data;
```

```
iph ->version = 4;
iph ->ihl  = 5;
if  ( opt ) {
   iph ->ihl  +=  opt->optlen >> 2;
   ip_options_build ( skb, opt,  inet->cork.addr,  rt , 0 );  //创建 IP 选项
}
iph ->tos=inet -> tos;
iph ->frag_off = df;
ip_select_ident ( iph , &rt->u.dst , sk );
// 产生数据包 ID
iph ->ttl=ttl ;
iph ->protocol=sk ->sk_protocol;
iph->saddr=rt->rt_src;
iph->daddr=rt->rt_dst;
```

最后在 dst_output 完成数据发送之前，函数需要请求网络过滤子系统对数据包进行过滤，允许数据包向外发送。这里网络子系统对数据包的过滤，是一次性对数据包的所有数据片进行过滤。在早期 Linux 的 2.4 版本内核中，网络过滤子系统要分析每一个数据片，这样可以更好地保证 IP 数据包的安全性，但同时网络过滤子系统必须重组数据包并再次分割数据包，这样运行效率太低。

4．函数结束处理

在函数返回之前，ip_push_pending_frames 函数要清除 IPCORK_OPT 域。清除了 IPCORK_OPT 域后，会使 struct cork 数据结构中的内容无效，这是为了在以后发送数据包的目标地址与当前发送数据包的目标地址相同时，可以重新使用 struct cork 数据结构，而且 IP 层也需要知道什么时候释放旧数据。该功能由 ip_cork_release(inet)函数完成。

7.6.8 发送数据包的整体过程

在这节中我们将总结把数据包从传输层发送到网络层，再发送出去的整体过程。我们以传输层 UDP 协议为例，来说明 UDP 协议是如何通过 udp_sendmsg 函数调用 ip_append_data 和 ip_push_pending_frames 来实现数据包的发送过程的，分析它们协同完成发送任务的流程。

```
int udp_sendmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg, size_t len)
   {
…
       struct udp_opt *up = udp_sk(sk);    //获取 UPD 套接字设置的选项
       //设置 UDP 数据传送标志，加入 MSG_MORE 标志，多次调用传送数据
       int corkreq = up->corkflag || msg->msg_flags & MSG_MORE;
       //调用 ip_append_data 函数向网络层缓冲数据，UDP 实现数据复制的函数是 ip_generic_getfrag，数据来自
msg->msg_iov 用户地址空间
       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);
      //如果函数 ip_append_data 调用失败，释放 sk_write_queue 队列中的数据缓冲区，如果 corkreg 中没有设置
MSG_MORE 标志，则立即传送缓冲的数据
       if (err)
          udp_flush_pending_frames(sk);
       else if (!corkreq)
          err = udp_push_pending_frames(sk, up);
```

udp_sendmsg 首先调用 ip_append_data 函数缓冲要发送的数据，随后如果在 corkreq 标

志中没有设置 MSG_MORE，则立即调用 udp_push_pending_frames 函数将刚缓冲的数据包发送出去，即 UDP 协议不等待。如果 ip_append_data 执行失败，则 udp_sendmsg 将数据包从 sk_write_queue 队列中取出释放，该过程由 udp_push_pending_frames 函数完成，该函数是 IP 层 ip_push_pending_frames 函数的包装函数。

图 7-29 给出了 udp_push_pending_frames 内部的执行流程。

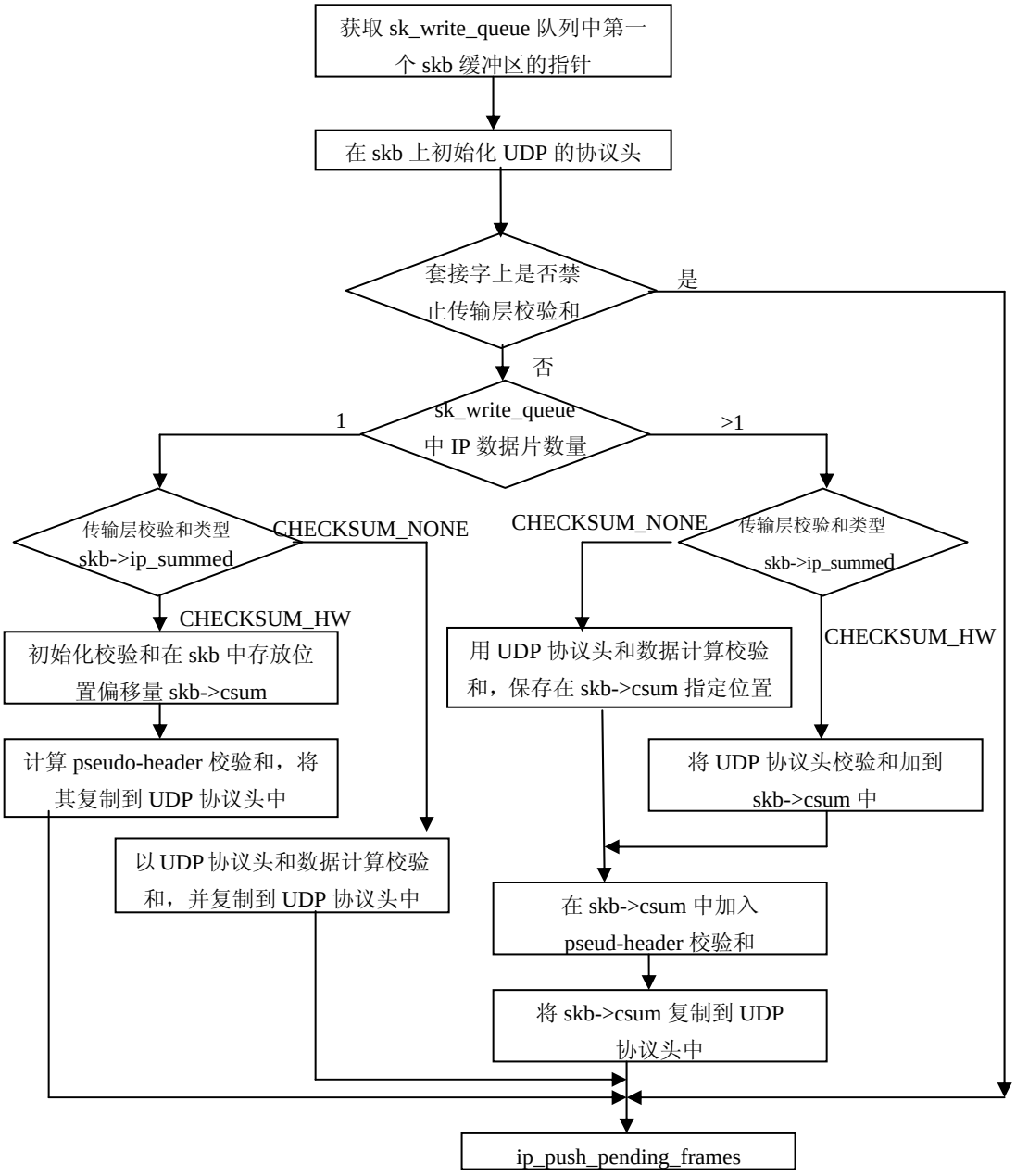


图 7-29　udp_push_pending_frames 函数的实现流程

如果要了解 ip_append_page 函数是如何被调用的，则可以参看 udp_sendpage 函数的实现。

最后在图 7-30 中给出了在 IP 层数据接收和发送时各 API 之间调用关系的总图。

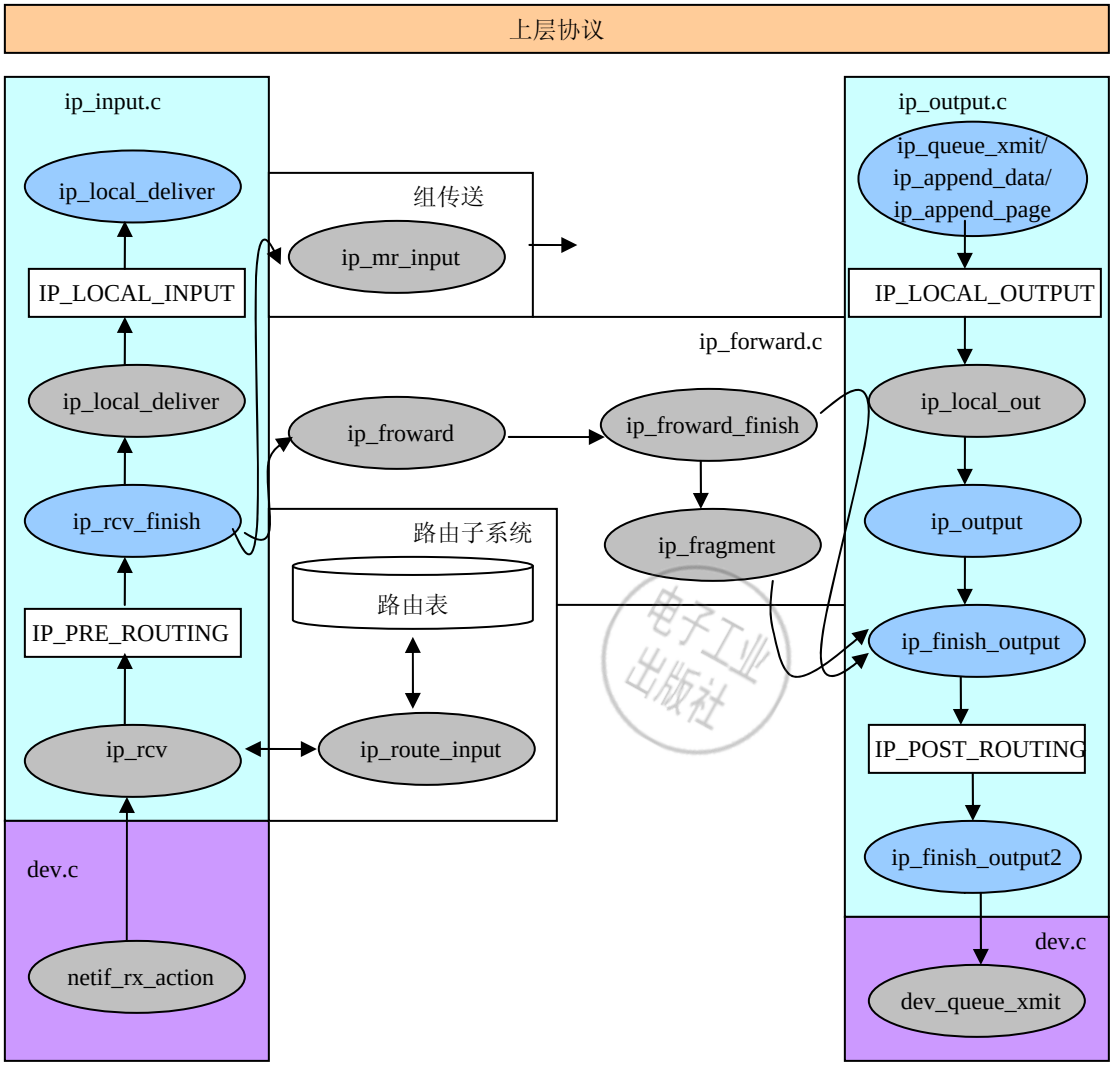


图 7-30　网络层实现数据包接收/发送的 API 调用关系总图

7.7　与相邻子系统的接口

在网络层数据包的发送结束于对函数 ip_output 的调用。ip_output 函数是网络过滤子系统回调函数的包装函数。ip_ouput 函数是两阶段工作函数，在数据包通过网络过滤子系统检查后，最终调用 ip_finish_output 函数，将数据包发送给相邻子系统。

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

  IP_INC_STATS ( dev_net (dev ),  IPSTATS_MIB_OUTREQUESTS );
//设置发送数据包的输出网络设备，设置网络层使用协议
  skb ->dev = dev;
  skb ->protocol  =  htons ( ETH_P_IP );
//调用网络过滤子系统的回调函数检查网络数据包，通过检查后调用 ip_finish_output    //函数完成数据发送流程
return  NF_HOOK_COND ( PF_INET, NF_INET_POST_ROUTING, skb, NULL, dev,
                ip_finish_output,
                ! (IPCB( skb )->flags & IPSKB_REROUTED));
}
static int ip_finish_output (struct sk_buff *skb)
{
  …
//如果数据包需要分割，则对数据包分片后调用 ip_finish_output2 函数继续发送流程；//如果不需分割数据包，则直接调用 ip_finish_output2 函数
  if ( skb ->len >  ip_skb_dst_mtu (skb ) &&  !skb_is_gso (skb ) )
        return  ip_fragment (skb , ip_finish_output2 );
  else
        return ip_finish_output2 ( skb );
}
static inline int ip_finish_output2 ( struct sk_buff * skb )    //net/ipv4/ip_output.c

//根据数据包目标地址类型：组传送/唯一主机地址，更新统计信息
//如果 headroom 空间不够存放数据链路层协议头，为数据链路层协议重新分配预留存放协议头的空间

if (dst ->hh )
        return  neigh_hh_output (dst ->hh,  skb );
  else if  (dst ->neighbour )
        return  dst ->neighbour->output ( skb );
```

当所有的数据都放入数据包后（包括数据链路层的协议头），IP 层协议就调用 dev_queue_xmit 函数（通过 hh->hh_output 和 dst->neighbour->output）来处理硬件发送。

dst->hh->hh_output 指针和 dst->neighbour->output 指针在相邻子系统中都初始化为 dev_queue_xmit 函数。

```
static struct neigh_ops arp_generic_ops  = {              //net/ipv4/arp.c
  .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,
};
```

7.8 数据包的分片与重组

数据包的分片和重组是 IP 协议的主要任务之一，IP 协议的数据包最大可以达到 64KB，这是由 IP 协议头的 iphdr->len 数据域的长度决定的。数据包的大小以字节计算，len 是一个 16 位的数据域。事实上能发送 64KB 数据包的接口并不多，即当 IP 层发送的数据包的大小大于输出网络接口的 MTU 时，它需要将数据包分成小的片段，数据包分片是创建一系列大小相同的数据段，如图 7-31 所示，图中的 MF 和 Offset 在本节中将会解释。

一个已分片的 IP 数据包通常在目的主机上重组，但中间当主机需要查看整个 IP 数据时，也会重组数据包。例如，防火墙和 NAT（Network Address Translation）路由器就需要重组数据包。

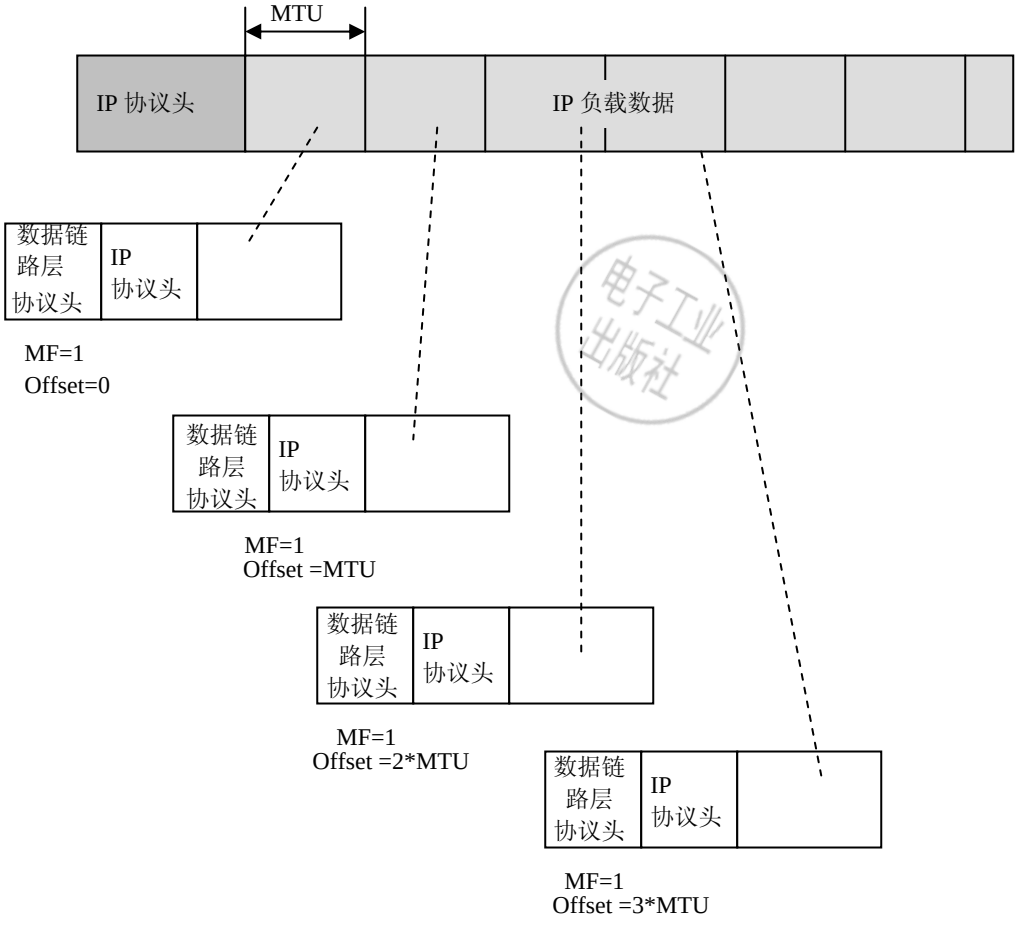


图 7-31 数据包分片示意图

7.8.1　数据分片需要考虑的问题

1．内存分配

以前，接收数据包的主机为收到的 IP 数据片分配一个与原完整数据包一样大的缓冲区来存放 IP 数据段。实际上，接收方是分配了一个可能用到的最大缓冲区。整个 IP 数据的大小要在收到全部数据片后才知道。现在要避免使用这种方法，其一，是因为这样太浪费空间；其二是，这样容易使主机受到攻击。任何攻击都可以通过发送大量的、非常小的数据段使路由器崩溃。

2．接收主机需要哪些信息来重组数据包

（1）每个数据片所属 IP 数据包及位置信息

每个 IP 数据包都可以分段，在发送过程中已分段的数据片还可以再被分割成更小的数据片，所以必须有某种方式通知接收数据的主机，每个数据片属于哪个 IP 数据包，它们在原 IP 数据包中应放在什么位置。

（2）原数据的长度

接收主机也需要源 IP 数据包的大小信息，才知道数据片是否已全部接收到。

（3）分片时是否复制协议头和选项

什么时候需要将源 IP 数据包的协议头信息复制到其分片数据包中，内核并不复制所有的选项，只有那些 copied 域设置的选项，当合并 IP 数据片时，合并好的 IP 数据包和源 IP 数据包一样，包含了所有的选项。

3．分片数据的校验和

IP 的校验和只覆盖了 IP 协议头信息（负载数据一般由高层协议的校验和覆盖），当创建 IP 分片数据时，每个段的头信息不同，所以要为每个段重新计算它们自己的校验和，在接收端做相应的检验。

7.8.2　在上层分片的效率

对数据包进行分片和重组既耗费 CPU 的资源又占用内存。对于一个高负载的服务器，对额外资源的需求非常大，对数据的分段也会占用更多的发送带宽。因为每个数据片中都必须包含数据链路层和网络层的协议头信息，如果数据分的片越小，额外负载就越大。

从理论上讲，协议栈的上层不知道网络层什么时候会对数据分片，即使 TCP 和 UDP 不知道分片/重组的过程，但最上层的应用程序有可能需要关注数据包的分片和重组。数据包的分片/重组会影响系统的性能，增加额外的延迟。对传输延迟非常敏感的应用程序要尽可能地避免对数据包的分片和重组，比如视频会议系统就会关心数据包是否分片。

7.8.3　数据包分片/重组使用的 IP 协议头数据域

1．DF（ Don’t Fragment）

如果数据包分片会严重影响上层的性能，例如，分片会对流媒体等应用造成很大的性能影响。有时发送方知道数据接收方只实现了简单的 IP 协议实现，不能处理数据包的分片

和重组，这时在 IP 数据包协议头信息中有一个数据域，说明是否允许对数据包进行分片。如果在传输路径中，数据包的大小超过了某些设备的最大传输单元，数据包就会被扔掉。

2．MF（More Fragments）

如果一个节点对数据包进行了分片，则它在每个分片数据包中把该标志设置为 TRUE，最后一个除外。在接收方收到没有设置标志的最后一个分片数据包后，即使没有收到全部的数据片，也能获取原数据包大小的信息。

3．Fragment Offset

Fragment Offset 代表了数据片在原 IP 数据包中的位置信息。这是一个 13 位的数据域。因为 len 是 16 位的数据域，分片数据必须创建在 8 字节的边界，这个字节为 8 的整数倍（即左移 3 位）。当 Offset 为 0 时说明数据片是数据包中的第一个数据片；在第一个数据片中通常都包含了最完整的协议头信息，所以这个信息很重要。

4．ID

ID 是 IP 数据包的标识符，这个值在一个 IP 数据包的所有数据片都是一样的。有了这个值，接收方才知道收到的数据片应组装在哪个数据包中。

以上是对数据包分片的基本描述，限于篇幅在本书中我们将不对数据包的分片与重组实现做详细描述，读者可以参照前面分析数据包接收/发送实现的方法，通过阅读 net/ipv4/ip_fragment.c 中的源代码来分析数据包的分片/重组在 Linux 内核中的实现。

7.9 本章总结

Internet 协议是 TCP/IP 协议栈的核心，IP 协议中规定了在 Internet 上进行通信时应遵循的规则，包括 IP 数据包应如何构成、数据包的路由等。本章我们在分析 IP 协议的主要任务：数据包寻址、路由选择、数据包的分片与重组的基础上，讲解了 Linux 内核中 IP 协议功能在以下方面实现技术：

- 管理 IP 协议的数据结构。数据包在 IP 协议中的处理信息都包含在 IP 协议头中，这些信息主要分为两个部分：IP 协议头与 IP 协议选项，这些信息在内核中都有相应的数据结构描述。
- 接收数据包在网络层中 IP 协议的处理流程、IP 选项的处理过程，以及主要功能函数的实现流程、源代码分析。
- IP 选项的解析、处理技术，主要功能函数实现流程与源代码分析。
- 发送网络数据包在网络层中 IP 协议的处理，IP 协议头、IP 协议的创建过程。IP 协议与 TCP/UDP 协议之间发送函数的实现流程及源代码分析。
- IP 层与传输层、邻居协议、数据链路层之间的接口。

由此帮助读者理解 IP 数据包在网络层的传输、处理路径。

第 8 章　传输层 UDP 协议的实现

本章主要介绍了传输层 UDP 协议的功能与传输特点，分析了 Linux 内核实现 UDP 协议的数据、UDP 协议层与套接字层和网络层 IP 协议之间的接口；重点讲解了 UDP 协议在套接字层与网络层之间接收/发送网络数据流程与函数代码的实现。

用户数据报协议（UDP: User Datagram Protocol）是一个简单的传输层协议，UDP 协议规范在 RFC768 中进行描述。UDP 协议提供的是低开销、无连接、不可靠数据报发送服务。在这里“不可靠”的含义是指在 UDP 协议中没有检查数据丢失或数据包是否为复制数据包的检错、纠错机制。与 TCP 一样，UDP 是在应用层与网络层之间传递数据的。应用程序可以根据实际需要来选择采用 TCP 或是 UDP 服务。

8.1　UDP 协议基础

如前所述 UDP 是不可靠、无连接的数据报协议。“不可靠”仅仅意味着在 UDP 协议中没有检测数据是否能够到达网络另一端的机制；在主机内，UDP 可以保证正确地传递数据。

既然 UDP 是不可靠发送协议，为什么应用程序还要选用 UDP 来做数据传输服务？因为如果要发送的数据量很小，创建连接的开销、保证可靠发送需要做的工作可能比发送数据本身的工作量还要大很多时，UDP 就是一个很好的选择。有的应用程序本身就具有保证数据可靠发送的机制，不要求传输层协议提供这方面的服务，这时用 UDP 也是一个很好的选择。另外一些面向业务的应用程序，如域名系统（DNS：Domain Name System），在这类应用程序中，只有一个请求和相关的回答需要发送，这时建立连接和维护连接的开销太大，那么 UDP 也适用于这类应用程序。

UDP 在数据包的协议头中使用一个 32 位字节来描述数据包发送的源地址与目标地址，其中 16 位字节描述源端口号，另外 16 位字节描述目的端口号，然后 UDP 通过这 32 位字的描述将数据传给正确的应用程序，图 8-1 描述了 UDP 数据包的格式。

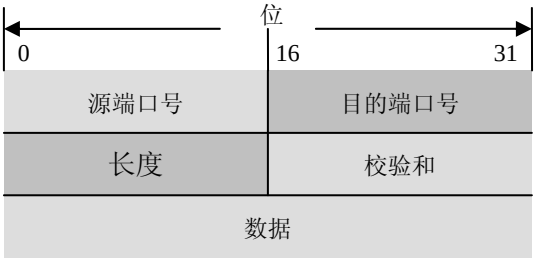


图 8-1　UDP 协议数据包格式

- 源端口号（source port）：是发送数据进程时使用的端口号，其取值范围为 1~65535。
- 目的端口号（destination port）：用于寻址系统中接收 UDP 数据包的目的应用。

- 长度（length）：指整个 UDP 数据包的长度，包括协议头和负载数据。UDP 数据包的最小长度可以是 8 个字节，其发送的最大负载数据可以是 65535 – 8 = 65527 字节。
- 校验和（checksum）：与 TCP 协议一样，UDP 校验和的计算包括协议头和负载数据。

在应用程序发送的信息（message）中写入一个 UDP 套接字，UDP 协议将从套接字接收到的信息封装在 UDP 数据包中，随后进一步封装入 IP 数据包，发送到目标主机进行接收。

8.2 UDP 协议实现的关键数据结构

在 Linux 内核中 UDP 协议的实现并不需要特别复杂的数据结构。在这一节中我们将描述 Linux 内核实现 UDP 协议的数据结构，这些数据结构用于在套接字接口上发送 UDP 负载数据，描述 UDP 数据包格式、常规套接字缓冲区结构，以及将实现 UDP 协议集成到 TCP/IP 协议栈中的数据结构。

8.2.1 UDP 协议头的数据结构

struct udphdr 数据结构描述的是 UDP 协议头信息，其各数据域的含义与图 8-1 中的 UDP 数据包格式定义相对应。

```
struct udphdr {                                          //include/linux/udp.h
  __be16      source;
  __be16      dest;
  __be16      len;
  __sum16     check;};
};
```


8.2.2 UDP 的控制缓冲区

在 Socket Buffer 的 sk_buff 结构中有一个控制缓冲区，供 TCP/IP 协议栈中各层协议实例存放自身的私有数据。UDP 也有自己的控制缓冲区，它使用该缓冲区来指明 UDP 协议实例进行校验和的方式等信息。

```
struct udp_skb_cb {                                      //include/net/udp.h
  union {
        struct inet_skb_parm   h4;
#if defined(CONFIG_IPV6) || defined (CONFIG_IPV6_MODULE)
        struct inet6_skb_parm  h6;
#endif
  } header;
  __u16       cscov;
  __u8        partial_cov;
};
#define UDP_SKB_CB(__skb)     (struct udp_skb_cb *)((__skb)->cb))
```

- h4、h6：分别为 IPv4 和 IPv6（如果内核配置在网络层使用 IPv6 协议）的选项信息。这两个数据域必须是 UDP 控制缓冲区的前两个数据域。
- cscov：UDP 计算校验和时，校验和覆盖的 UDP 数据包长度。

- partial_cov：如果设置这个数据域，它指明 UDP 协议计算部分校验和，以及校验和覆盖的 UDP 数据包长度。

访问 UDP 控制缓冲区只能通过宏 udp_skb_cb (__skb)来访问。

8.2.3　UDP 套接字的数据结构

struct udp_sock 数据结构描述的是 UDP 套接字专有的特性。struct udp_sock 数据结构是在 struct inet_sock 数据结构的基础上扩展而来。struct inet_sock 数据结构是 AF_INET 地址族域（Domain）套接字专用数据结构，它是通用套接字数据结构 struct sock 的扩展，在具有了基本的套接字属性的基础上，struct inet_sock 数据结构又增加了 AF_INET 地址族域套接字特有的属性，如 TTL、IP 地址、端口号等。而 struct udp_sock 数据结构又是在继承了通用套接字与 AF_INET 地址族域套接字属性的基础上，扩展了 UDP 套接字本身的属性，在 struct udp_sock 数据结构中包含了发送 UDP 数据包所需要的全部管理、控制信息。udp-sock 数据结构的完整定义如下。

```
struct udp_sock {                    //include/linux/udp.h
  struct inet_sock inet;
  int         pending;
  unsigned int      corkflag;
  __u16       encap_type;
  __u16       len;
  __u16       pcslen;
  __u16       pcrlen;
  __u8        pcflag;
  __u8        unused[3];
  int (*encap_rcv)(struct sock *sk, struct sk_buff *skb);
};
```

- pending：指明当前是否有等待发送（悬挂）的数据包。
- corkflag：此标识指明当前是否需要暂时阻塞套接字。
- encap_type：指明此套接字是否为封装的套接字。
- len：指等待发送（悬挂）的数据包的总长度。在套接字没有阻塞的情况下，该数据域中包含的信息，可用于创建 UDP 协议头。
- pcslen、pcrlen：这两个字段用于轻 UDP 套接字。pcslen 数据域指明当前在 UDP 套接字中等待发送的数据包长度。pcrlen 为当前套接字上等待接收的数据包长度。
- Pcflag：当 pcflag 的值大于 0 时，表示当前 UDP 套接字是轻套接字。
- *encap_rcv：指是封装 UDP 套接字的接收函数。

8.2.4　应用程序发送给 UDP 负载数据的数据结构

与 TCP 协议实例一样，应用程序在套接字接口上通过 sendmsg 系统调用发送数据，sendmsg 系统调用的执行将转而调用传输层的函数，将应用程序的数据从用户地址空间复制到内核地址空间，用户地址空间的负载数据用 struct msghdr 数据结构描述，在 UDP 协议实例中 struct msghdr 数据结构把包含的信息传给 udp_sendmsg 函数，形成 UDP 数据包向外发送。struct msghdr 数据结构我们将在 9.2.5 节中给出详细描述。

在发送 UDP 数据包时，msghdr->msg_name 实际上不是套接字名，而是一个指针，指向 sockaddr_in 数据结构变量，sockaddr_in 数据结构中包含了 IP 地址和端口号。

8.3 UDP、套接字层、IP 层之间的接口

UDP 协议实例向上有与套接字层之间的接口，向下有与网络层（IP 层）之间的接口，在完成将应用程序数据复制到内核地址空间后继续向网络层传递，或将从网络层收到的数据包传给正确的应用程序。

8.3.1 UDP 协议实例与套接字层间的接口

UDP 与套接字层的接口由 struct proto 数据结构描述，该数据结构我们将在 9.3.1 节中进行介绍。struct proto 数据结构中的虚函数在 UDP 协议实例中的实现如表 8-1 所示。

表 8-1 UDP 协议实例实现与套接字层之间的接口函数

struct proto 数据结构中的数据域	UDP 协议实例中的函数
close	udp_lib_close
connect	ip4_datagram_connect
disconnect	udp_disconnect
ioctl	udp_ioctl
destroy	udp_destroy_sock
setsockopt	udp_setsockopt
getsockopt	udp_getsockopt
sendmsg	udp_sendmsg
recvmsg	udp_recvmsg
sendpage	udp_sendpage
backlog_rcv	__udp_queue_rcv_skb
hash	udp_lib_hash
unhash	udp_lib_unhash
get_port	udp_v4_get_port

由于 UDP 套接字是无连接的套接字，所以它对连接的管理和状态的管理比 TCP 协议实例简单很多。从 udp_lib_close 到 udp_destroy_sock 这 5 个函数用于管理套接字连接，udp_sendmsg、udp_recvmsg 实现 UDP 套接字上的数据发送和接收，关于 udp_sendmsg、udp_recvmsg 函数的实现，我们将分别在 8.4 节和 8.5 节中加以分析和讲解。在 UDP 数据接收期间，必须确定接收到的数据包应分配给哪个套接字，以便将数据包放入套接字的接收队列，随后由用户进程提取。为了实现接收数据包与套接字的匹配，UDP 上打开的所有套接字由 udp_v4_hash 函数注册在 struct sock *udp_hash[UDP_HTABLE_SIZE]哈希链表中，端口号是查询哈希链表的 hash 值。在释放套接字时调用 udp_v4_unhash 函数，将套接字结构从 UDP 哈希链表中移出。

1. 接口初始化

UDP 协议实例化及初始化一个套接字与传输层之间接口的过程在 net/ipv4/udp.c 文件中

实现，在 net/ipv4/udp.c 文件中，UDP 协议定义了 struct proto 数据结构的变量实例 udp_prot，对 udp_prot 变量的各数据域（函数指针）初始化，将虚函数实例化。

```
struct proto udp_prot = {                          //net/ipv4/udp.c
   .name   = "UDP",
   .owner  = THIS_MODULE,
   .close  = udp_lib_close,
   .connect     = ip4_datagram_connect,
   .disconnect  = udp_disconnect,
   .ioctl = udp_ioctl,
   .destroy  = udp_destroy_sock,
   .setsockopt   = udp_setsockopt,
   .getsockopt   = udp_getsockopt,
   .sendmsg  = udp_sendmsg,
   .recvmsg= udp_recvmsg,
   .sendpage= udp_sendpage,
   .backlog_rcv  = __udp_queue_rcv_skb,
   .hash   = udp_lib_hash,
   .unhash  = udp_lib_unhash,
   .get_port= udp_v4_get_port,

   …

   }
```

UDP 协议实例仍由 int proto_register(struct proto *prot, int alloc_slab)函数将 UDP 协议与套接字层之间的接口注册到 TCP/IP 协议栈中。通过 void proto_unregister(struct proto *prot)函数将 UDP 协议实例从系统中注销。

2. 注册协议接口

向 TCP/IP 协议栈注册 UDP 与套接字层之间的接口，要在 AF_INET 协议族初始化过程中完成。在内核启动时，会按优先级顺序调用各组件注册在内核中的初始函数，AF_INET 协议族初始化函数 inet_init（定义在 net/ipv4/af_inet.c 文件中）会在网络子系统初始化之后被调用执行。inet_init 函数的功能之一就是，完成传输层各协议实例的实始化，其中也包括注册 UDP 与套接字的接口。

```
static int __init inet_init(void)                  //net/ipv4/af_inet.c
{
   int rc = -EINVAL;
…
   rc = proto_register( &udp_prot, 1);
   if (rc)
        goto out_unregister_tcp_proto;
…
}
```

8.3.2　UDP 协议与 IP 层之间的接口

IP 层与 UDP 协议之间接收数据包的接口由 struct net_protocol 数据结构描述，struct net_protocol 数据结构的详细描述将在 9.3.3 节中进行介绍。struct net_protocol 数据结构中定义了一系列的函数指针，都是传输层接收来自网络层数据包的处理函数。

在 Linux 内核网络体系结构中，TCP/IP 协议栈各层之间的接口定义得非常清楚，层次分明。如果我们需要在协议栈中加入一个新的协议实例，实现起来非常容易，常规实现步

骤为：

① 按照新协议规范拟定新协议实例需要实现的接收函数。

② 定义 struct net_protocol 结构的变量实例，如 struct net_protocol my_protocol。

③ 将 struct net_protocol 结构变量实例中的函数指针实例化。

④ 用接口注册函数 inet_add_protocol，将新协议的接口 my_protocol 加入到全局数组 struct net_protocol *inet_protos[MAX_INET_PROTOS]中。

⑤ 实现各接口函数。

UDP 的 struct net_protocol 变量实例化在 net/ipv4/af_inet.c 文件中完成。在同一个文件的 inet_init 函数中，将 UDP 接口 udp_protocol 加入到 inet_protos 全局数组中。

```
static struct net_protocol udp_protocol = {                //net/ipv4/af_inet.c
  .handler = udp_rcv,
  .err_handler =   udp_err,
  .no_policy =     1,
  .netns_ok =1,
};
```

udp_rcv 是 UDP 协议接收输入数据包的处理函数，udp_err 函数处理 ICMP 错误信息。

```
static int __init inet_init(void)                          //net/ipv4/af_inet.c
{
…
  if (inet_add_protocol(&udp_protocol, IPPROTO_UDP) < 0)
        printk(KERN_CRIT "inet_init: Cannot add UDP protocol\n");
…
}
```

UDP 协议与 IP 层之间没有定义发送接口，为了通过 IP 层发送数据，UDP 协议实例在 udp_sendmsg 函数中调用 IP 层发送数据包的回调函数 ip_append_data（定义在 net/ipv4/ip_output.c 文件中），或在 udp_sendpage 函数中调用 IP 层的回调函数 ip_append_page（定义在 net/ipv4/ip_output.c 文件中），将 UDP 数据报放入 IP 层。

8.4 发送 UDP 数据报的实现

用户程序在应用层调用 socket 系统调用，打开一个 SOCK_DGRAM 类型的套接字，在通过该套接字接收和发送数据时，数据包会通过 UDP 协议穿过 TCP/IP 协议栈向外发送，或由 UDP 协议上传给套接字到达应用程序。在本节中将分析数据包是如何由 UDP 协议向外发送的。

8.4.1 初始化一个连接

我们已经知道，UDP 协议各数据报的发送相互独立，在两个通信站点之间没有实际的连接需要维护，在 UDP 数据报的发送过程中没有子状态的切换和维护。虽然 UDP 不支持连接，但 UDP 协议仍支持 connect 套接字的系统调用。既然 SOCK_DGRAM 类套接字支持连接 connect 系统调用，接下来在发送数据的 send 系统调用中就可以省略数据包发送的目

标地址。在这一小节中，我们将讨论套接字的连接 connect 系统调用是如何在 UDP 协议中处理的。

1. connect 系统调用与 UDP

connect 系统调用也可以应用于 UDP 套接字上，用户程序调用 connect 系统调用时，需要指定要连接的目标地址。在 UDP 协议上支持 connect 系统调用的主要目的是建立到达目标地址的路由，并把该路由放入路由高速缓冲存储器中。一旦路由建立起来，接下来在通过 UDP 套接字发送数据包时就可以使用路由高速缓冲区中的信息了。这种方式称之为在连接套接字上的快速路径“fast path”。

当在一个打开的 SOCK_DGRAM 类套接字上调用 connect 系统调用时，套接字层会转而调用 ip4_datagram_connect 函数（定义在 net/ipv4/datagram.c 文件中）。以下我们将对 ip4_datagram_connect 函数的实现进行分析。

2. UDP 套接字连接实现

在建立 UDP 套接字连接时，需要的信息都由输入参数传给了 ip4_datagram_connect 函数。

（1）函数的输入参数

- sk：指向打开的 UDP 套接字的 struct sock 数据结构。
- uaddr：当前 UDP 套接字要与之创建连接路由的目标地址。
- addr_len：是目的地址的长度，目标地址是 struct sockaddr 数据结构类型。

（2）函数初始化过程

函数的起始部分使用函数输入参数来初始化部分局部变量，将 IPv4 特定的地址信息和选项信息，从套接字数据结构 sk 中初始化到 AF_INET 协议族选项 struct inet_opt 数据结构类型的指针 inet 变量中；将路由目标地址初始化到 usin 局部变量中。rt 将初始化为指向路由高速缓冲区的入口地址。oif 为输出网络设备接口的索引号，数据包将通过该网络接口，经目标路由到达目标地址。

```
int ip4_datagram_connect(struct sock *sk, struct sockaddr *uaddr, int addr_len)
{                                                                    //net/ipv4/datagram.c
  struct inet_sock *inet = inet_sk(sk);
  struct sockaddr_in *usin = (struct sockaddr_in *) uaddr;
  struct rtable *rt;
  __be32 saddr;
  int oif;
  int err;

}
```

（3）正确性检查

在建立连接、寻址路由，前系统首先需对目标地址和套接字类型做正确性检查。这项检查主要为查看目标地址的长度是否正确，目标地址类型是否为 AF_INET 协议族地址。

其次，在寻址新目标路由前，先复位套接字 sk 数据结构中原目标地址在路由高速缓冲区中的入口指针。

再次，因为要打开的套接字可能是与某个网络接口绑定了的套接字，如果绑定的网络接口已知，则 oif 变量会初始化为输出网络接口的索引号。如果套接字没有与网络接口绑定，

则 oif 的值为 0。

最后，需要查看目标地址是否为组传送地址，如果目标地址不是一个组传送地址，说明用户程序并不是要连接一组目标地址，只是想要传送一系列的数据包到同一地址。另外如果目标地址是组传送地址，为了让接收方获取数据包的来源信息，我们就从套节字数据结构 sk 的 inet_opt 数据域中获取输出接口和源地址信息。

```
//对建立连接的信息做正确性检查：地址长度、协议类型
  if (addr_len < sizeof(*usin))
       return -EINVAL;
  if (usin->sin_family != AF_INET)
       return -EAFNOSUPPORT;
//复位套按字中源目标地址在路由缓存中的记录
  sk_dst_reset(sk);
//套接字为与网络接口绑定的套接字，将绑定的网络接口信息保存在 oif 局部变量中
  oif = sk->sk_bound_dev_if;
  saddr = inet->saddr;
//如果建立连接的地址是组传送地址，则重新初始化 oif 与源地址 saddr

  if (ipv4_is_multicast(usin->sin_addr.s_addr)) {
       if (!oif)
            oif = inet->mc_index;
       if (!saddr)
            saddr = inet->mc_addr;
  }
```

（4）寻址目标路由

接下来的调用是该函数中最重要的部分。ip_route_connect 返回一个到达目标地址（存放在 usin 变量中）的新路由，并将 rt 指针指向新路由在路由高速缓冲区中的入口。如果函数 ip_route_connect 的返回值为非零，即函数没能寻址到新路由或不能将新路由加入到路由高速缓冲区中，则 ipv4_datagram_connect 函数向调用程序返回错误信息；如果寻址到的是广播地址路由，函数也返回错误。

```
//为连接寻址一个新路由，如果寻址新路由成功，则将新路由放入缓存
  err = ip_route_connect(&rt, usin->sin_addr.s_addr, saddr,
                  RT_CONN_FLAGS(sk), oif,
                  sk->sk_protocol,
                  inet->sport, usin->sin_port, sk, 1);
//如果寻址新路由不成功，则更新错误统计信息，返回错误代码
  if (err) {
       if (err == -ENETUNREACH)
            IP_INC_STATS_BH(sock_net(sk), IPSTATS_MIB_OUTNOROUTES);
       return err;
  }

//如果寻址的新路由为广播地址路由，则释放该路由在路由缓存中的入口，返回错误代码
  if ((rt->rt_flags & RTCF_BROADCAST) && !sock_flag(sk, SOCK_BROADCAST)) {
       ip_rt_put(rt);
       return -EACCES;
  }
```

（5）函数结束处理，更新信息

此时函数从路由高速缓冲区获取信息，为 UDP 连接更新发送 UDP 数据报的源地址和目标地址，目标端口号由用户指定。

然后将套接字的状态设置为 TCP_ESTABLISHED，这个状态说明目标路由已缓存在路

由高速缓冲区中（对 UDP 套接字而言，这个状态并不能说明已经建立了一个）。当用户程序发送数据包时，如果没有给出数据包的目标地址，数据包仍可发送，因为套接字的状态指明路由已建立。

最后，函数将存放数据包的 Socket Buffer 目标地址的 sk->dst 数据域更新为在路由高速缓冲存储器中的入口，函数结束处理并返回。

```
    //用从路由表中获取的信息更新 UDP 连接的源地址与目标地址
    if (!inet->saddr)
          inet->saddr = rt->rt_src;
    if (!inet->rcv_saddr)
          inet->rcv_saddr = rt->rt_src;
    inet->daddr = rt->rt_dst;
    //UDP 连接的目标端口号来自用户程序，设置套接字状态为 TCP ESTABLISHED
    inet->dport = usin->sin_port;
    sk->sk_state = TCP_ESTABLISHED;
    inet->id = jiffies;
    //新路由在路由高速缓存中的入口保存于套接字 sk->sk_dst_cache 数据域
    sk_dst_set(sk, &rt->u.dst);
    return(0);
}
```

8.4.2　在 UDP 套接字上发送数据包

在这一节中我们将讲解 UDP 协议是如何处理来自应用层的数据包发送请求的，以及什么时候会调用 UDP 协议发送函数。在用户程序通过 sendmsg 系统调用或别的应用层发送函数向打开的 SOCK_DGRAM 类套接字写数据时，UDP 协议会处理这项请求。

UDP 协议处理函数的特点是：

- 不需要缓存数据。
- 不需要管理连接。
- 当创建 UDP 协议头时，只需要数据包的源端口号和目标端口号信息。

在 UDP 协议实例中如何获取发送数据包的目标 IP 地址和目标端口号：通常目标 IP 地址和端口号是在套节字名中一起给出的。如果目标端口号已知，目标 IP 地址就已知；如果目标 IP 地址在套接字名中已给出，目标 IP 地址会在数据包传送给 IP 层前先保存起来。另外，在数据包传送给 IP 层前，UDP 协议需要查看目标 IP 地址是内部网络地址还是外部 IP 地址，是否需要为数据包寻址路由。如果目标地址可能是组发送地址或广播地址，则 IP 层就不需为数据包寻址路由。在 UDP 协议完成处理前，数据包的源端口号和目标端口号需放入 UDP 数据包的协议头中，而 IP 地址会留到 IP 层中再做处理。

Linux 内核 TCP/IP 协议栈实现的关键是使处理过程高效，所做的所有优化都致力于避免数据的重复复制或不必要的处理。在 UDP 协议实例的实现过程中，其复杂之处主要体现在找出数据包的发送路径，在整个过程中只有一次实际的数据复制。

当应用程序在一个打开的 SOCK_DGRAM 类型套接字上调用写数据函数时，套接字层调用由 struct prot 数据结构的 sendmsg 数据域函数指针指向的函数。对于 SOCK_DGRAM 套接字，它会调用 udp_sendmsg 函数，该函数是在 SOCK_DGRAM 类套接字，即 UDP 协议实例上执行发送功能的函数。以下我们就对 UDP 协议发送数据包的具体过程进行分析说明。

1．udp_sendmsg 函数输入参数及局部变量初始化

udp_sendmsg 函数完成的功能是从用户地址空间接收发送数据，复制到内核地址空间；通过有效的路由向外发送。要完成以上功能，udp_sendmsg 函数需要知道数据来源、与发送数据相关的套接字等，这些信息通过输入参数提供给函数，udp_sendmsg 函数用这些参数初始化局部变量，以便于后续处理。

（1）输入参数

- kiocb：是为提高对用户地址空间操作效率的数据结构。
- sk：指向打开的套接字的数据结构，其中包含了该套接字的所有设置和选项信息。
- msg：存放和管理来自用户地址空间的数据。
- len：从用户地址空间复制数据的总长度。

（2）局部变量初始化

函数从套接字数据结构 sk 中提取 IP 协议选项信息和 UDP 套接字信息，分别存放在局部变量 inet 和 up 中。

用户地址空间传入数据的长度，从输入参数 len 中初始化到局部变量 ulen 中。ipc 中将存放从 IP 层的 ICMP 协议返回的控制消息值。

rt 将指向路由高速缓冲存储器中数据包目标路由的入口；commected 变量将存放数据包目标路由是否已缓存的标志，当其值为 ESTABLISHED 时，说明数据包的目标路由已缓存；daddr 为数据包的目标 IP 地址；tos 是 IP 协议头的 ToS 数据域；is_udplite 标志该 UDP 套接字是否为轻 UDP 套接字。

```
int udp_sendmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg, size_t len)
{                                                    //net/ipv4/udp.c
  //用输入参数初始化函数局部变量
  struct inet_sock *inet = inet_sk(sk);
  struct udp_sock *up = udp_sk(sk);
  int ulen = len;
  struct ipcm_cookie ipc;
  struct rtable *rt = NULL;
  int free = 0;
  int connected = 0;
  __be32 daddr, faddr, saddr;
  __be16 dport;
  u8  tos;
  int err, is_udplite = IS_UDPLITE(sk);
  int corkreq = up->corkflag || msg->msg_flags&MSG_MORE;
  int (*getfrag)(void *, char *, int, int, int, struct sk_buff *);
}
```

2. 正确性检查

在本书中分析 TCP/IP 协议栈各层协议实例的实现时，大家可以看到各类函数在起始部分都会做大量的正确性检查，因为 TCP/IP 协议栈是内核的代码，特别是传输层协议 TCP 和 UDP 是内核与用户地址空间的接口，一旦从用户地址空间传来的数据或指针有错，会引起对错误地址空间的访问，导致内核崩溃，同时也就会导致整个系统的崩溃和对系统的破坏。这里首先需对从用户地址空间传来的信息做以下正确性检查：

- 发送数据长度是否越界。

- 查看调用程序是否对该类型的套接字设置了非法的标志。对 UDP 套接字而言，唯一非法的标志是 MSG_OOB，该标志只对 SOCK_STREAM 类的套接字有效。

```
//数据长度是否正确
if (len > 0xFFFF)
      return -EMSGSIZE;
//用户程序是否对 UDP 套接字设置了非法标志
if (msg->msg_flags&MSG_OOB)
      return -EOPNOTSUPP;
```

3. 处理早期悬挂数据

首先，查看在当前套接字中是否有悬挂 pending 的数据包等待发送。如果有，则处理函数需要持有该套接字，获取套接字的锁，并阻塞该套接字以便处理悬挂的数据包。然后跳转到将数据包传给 IP 层的标签处：do_append_data。

```
if (up->pending) {                                          //套接字中有挂起的数据帧等待发送
        lock_sock(sk);                                      //获取套按字
        if (likely(up->pending)) {
              if (unlikely(up->pending != AF_INET)) {       //挂起的数据帧不是 AF_INET 类
                    release_sock(sk);                       //释放套按字
                    return -EINVAL;
              }
              goto do_append_data;                          //跳转到将数据帧加入 IP 层缓冲区处理标签处
        }
        release_sock(sk);
  }
```

4. 处理新数据

如果本套接字上没有悬挂的数据包，则函数开始处理用户地址空间传来的数据。首先调用 udp_sendmsg 函数查看目标 IP 地址是否有效，当前套接字是否处于已连接状态。

（1）目标 IP 地址有效

前面我们说到，数据包的目标 IP 地址由套按字名给出；而套接字名在 struct msghdr 数据结构中给出，由输入参数 msg 传入 udp_sendmsg 函数：msg->msg_name。

在获取目标 IP 地址后，需对目标 IP 地址的正确性进行检查，这些正确性检查包括地址长度、地址所属的协议族是否为 AF_INET。如果没有通过正确性检查，则函数将向调用程序返回错误代码。在目标 IP 地址有效的情况下，用目标 IP 地址初始化存放数据包目标 IP 地址的局部变量 daddr 和端口号 dport。

（2）套接字已连接

当套接字状态为已连接状态（sk->state = ESTABLISHED）时，说明目标路由已缓存在路由高速缓冲存储器中，即目标地址已知。这时即使目标 IP 地址为空，也可以在连接的 UDP 套接字（即已路由的套接字）上发送数据包。

如果套接字已路由，则在 IP 层可以直接使用目标地址在路由缓冲区中的入口。

```
ulen += sizeof(struct udphdr);
if (msg->msg_name) {
//从输入参数 msg->msg_name 获取目标 IP 地址，保存在局部变量 usin 中，对目标 IP 地址做正确性检查
      struct sockaddr_in * usin = (struct sockaddr_in*)msg->msg_name;
      if (msg->msg_namelen < sizeof(*usin))              //地址长度是否正确
```

```
                return -EINVAL;
        if (usin->sin_family != AF_INET) {                    //目标地址类型是否正确
                if (usin->sin_family != AF_UNSPEC)
                        return -EAFNOSUPPORT;
        }
//目标地址通过正确性检查后，用套接字名中的 IP 地址初始化和端口号局部变量
        daddr = usin->sin_addr.s_addr;
        dport = usin->sin_port;
        if (dport == 0)
                return -EINVAL;
  } else {

//如果套接字名 msg_name 中的目标 IP 地址为空，套接字也没有建立连接，则数据包传送的目标 IP 地址无效。如果套接字
//已连接，则用连接信息初始化目标 IP 地址与端口号
        if (sk->sk_state != TCP_ESTABLISHED)
                return -EDESTADDRREQ;
        daddr = inet->daddr;
        dport = inet->dport;
        connected = 1;
  }
```

（3）处理套接字的控制信息

在处理完发送数据包的目标 IP 地址后，接下来调用 udp_sendmsg 函数查看用户空间是否对该套接字配置了相应的控制信息。如果在套接字上配置了控制信息，函数接下来就对控制信息进行处理。

控制信息随用户地址空间的 struct msghdr 数据结构的 msg_control 数据域传送过来。如果有控制信息，则控制信息长度数据域 msg->msg_controllen 值应为非零。

解析好的套接字控制信息结果存放在 struct ipcm_cookie 数据结构类型的局部变量 ipc 中。ipc 的某些数据域事先已初始化，如 IP 选项和输出网络接口索引号（如果套接字与输出网络接口绑定）。

如果用户设置了套接字的控制信息，这些控制信息由函数 ip_cmsg_send 来处理。套接字控制信息指明了设置和提取 UDP 的地址和端口信息的方式。ip_cmsg_send 函数会把对控制信息的处理结果存放在 ipc 中。

如果套接字上没有设置有效的选项，inet 的选项数据域中包含地址信息，可以直接从中提取来设置控制信息。UDP 协议处理以下两类控制信息：

- I_RETOPTS：从 IP 协议头中提取选项数据域，返回指向选项数据域的指针，存放在 ipc->opt 数据域中。
- IP_PKTINFO：如果控制信息中设置了 IP_PKTINFO 控制域，则 ip_cmsg_send 函数就将网络接口的索引号返回给 ipc->oif 数据域，将接口的 IP 地址返回给 ipc->addr 数据域。struct ipcm_cookie 数据结构定义在 include/net/ip.h 头文件中。

```
struct ipcm_cookie
{
  __be32                addr;           //输出网络设备的 IP 地址
  int                   oif;            //输出网络设备索引号
  struct ip_options     *opt;           //IP 协议选项
};
  ipc.oif = sk->sk_bound_dev_if;
```

```
  if (msg->msg_controllen) {


//控制信息长度不为空，套接字设置的是 IP 层控制信息，将 IP 层选项设置在 ipc 变量中
        err = ip_cmsg_send(sock_net(sk), msg, &ipc);
        if (err)
              return err;
        if (ipc.opt)
              free = 1;
        connected = 0;
  }
//如果套接字上没设置有效控制信息，则从 inet 的选项数据域中提取来设置控制信息

  if (!ipc.opt)
        ipc.opt = inet->opt;
  saddr = ipc.addr;
  ipc.addr = faddr = daddr;
//如果 IP 选项设置了源路由，则下一站点的目标地址从源路由的 IP 地址列表中获取

  if (ipc.opt && ipc.opt->srr) {
        if (!daddr)
              return -EINVAL;
        faddr = ipc.opt->faddr;
        connected = 0;
  }
```

8.4.3　向 IP 层发送数据包

在向 IP 层传送数据包时，函数可以开始创建 UDP 协议头；同时要向 IP 层发送数据包的管理信息，这些管理信息包括目标地址、目标端口号、IP 选项、数据包长度等，接下来函数需要查看数据包应如何路由。

1. 无须设置路由

在以下条件都满足的情况下，不需要寻址数据包路由：

- 数据包在本地局域网中发送：sk->localroute。
- msg_flags 标志为不需路由。
- IP 选项设置了严格源路由。

同时根据以上条件设定数据包的服务类型（ToS）。

```
  tos = RT_TOS(inet->tos);

//如果数据包在本地局域网中传送：SOCK_LOCALROUTE，msg_flags 标志为不需路由或 IP 选项设置了严格源路由，设备 ToS
//为 RTO_ONLINK，则不需要寻址数据包路由

  if (sock_flag(sk, SOCK_LOCALROUTE) ||
     (msg->msg_flags & MSG_DONTROUTE) ||
     (ipc.opt && ipc.opt->is_strictroute)) {
        tos |= RTO_ONLINK;
        connected = 0;
  }
```

- 如果目标地址是组发送地址，不需要寻址数据包路由。

```
//目标 IP 地址是组传送地址，则不需要寻址数据包路由
if (ipv4_is_multicast(daddr)) {
      if (!ipc.oif)
            ipc.oif = inet->mc_index;
      if (!saddr)
            saddr = inet->mc_addr;
      connected = 0;
}
```

2. 路由已知

如果是在已连接的套接字上，路由已知，则将路由高速缓冲区入口设置给局部变量 rt。

```
if (connected)
      rt = (struct rtable*)sk_dst_check(sk, 0);
```

3. 目前无有效路由

如果目前无有效路由，则要在路由表中搜索目标路由。因此，首先创建流信息结构 struct flowi 数据结构变量，初始化 struct flowi 的数据域，为搜索路由做准备。然后调用 ip_route_output_flow 函数搜索路由表，建立目标路由。如果 ip_route_output_flow 函数返回的路由入口指明该路由是广播路由，但套接字没有设置 SO_BROADCAST 选项标志，则返回错误。

```
   if (rt == NULL) {

//如目前无有效路由，则初始化 struct flowi 数据结构的数据域，调用 ip_route_output_flow 函数在路由表中寻址新路由

   struct flowi fl = { .oif = ipc.oif,                    //准备在路由表中搜索路由
                       .nl_u = { .ip4_u =
                              { .daddr = faddr,
                                .saddr = saddr,
                                .tos = tos } },
                       .proto = sk->sk_protocol,
                       .flags = inet_sk_flowi_flags(sk),
                       .uli_u = { .ports =
                               { .sport = inet->sport,
                                 .dport = dport } } };
        struct net *net = sock_net(sk);
        security_sk_classify_flow(sk, &fl);
        err = ip_route_output_flow(net, &rt, &fl, sk, 1);        //在路由表中搜索路由

//寻址新路由失败，失败原因可能是：1.路由不可到达；2.寻址的路由是广播路由，但套接字层没有设置 SOCK_BROADCAST
//标志，返回错误信息
        if (err) {
              if (err == -ENETUNREACH)
                    IP_INC_STATS_BH(net, IPSTATS_MIB_OUTNOROUTES);
              goto out;
        }
        err = -EACCES;
        if ((rt->rt_flags & RTCF_BROADCAST) &&        //是广播地址路由
           !sock_flag(sk, SOCK_BROADCAST))
              goto out;

        }
```

4. 有效路由已建立

程序运行到此处，发送数据包的有效路由已具备；如果我们工作在已连接的套接字上，

设置一个指针指向目标路由缓存的路由记录。这时调用 udp_msgsend 函数查看应用程序在设置数据处理标志时，是否要求 UDP 套接字返回路由确认信息：MSG_CONFIRM；如果应用程序设置了 MSG_CONFIRM 标志，则程序跳转到返回路由信息处理标签处。函数 dst_confirm 向应用程序返回路由确认信息。

```
        if (connected)                                          //路由已有效
    sk_dst_set(sk, dst_clone(&rt->u.dst));
  //如果应用程序要求返回路由有效信息，则向应用程序返回路由确认信息

    if (msg->msg_flags&MSG_CONFIRM)
        goto do_confirm;
```

5. 开始向 IP 层发送数据

到此需要预处理的信息已完成，UDP 协议实例开始向 IP 层发送数据，在发送数据前 udp_sendmsg 函数应锁定套接字。其处理过程需经过以下 3 个阶段：

① 在预处理过程中如果套接字已阻塞，这里应是有一个 bug 释放套接字，返回错误信息。

```
  back_from_confirm:
    saddr = rt->rt_src;
    if (!ipc.addr)
        daddr = ipc.addr = rt->rt_dst;
    lock_sock(sk);                              //获取套接字锁
    if (unlikely(up->pending)) {                //套接字已阻塞，释放套接字
        release_sock(sk);
        LIMIT_NETDEBUG(KERN_DEBUG "udp cork app bug 2\n");
        err = -EINVAL;
        goto out;
    }
```

② 锁定套接字成功，阻塞套接字以便追加额外的数据，为发送数据做准备。

```
    inet->cork.fl.fl4_dst = daddr;
    inet->cork.fl.fl_ip_dport = dport;
    inet->cork.fl.fl4_src = saddr;
    inet->cork.fl.fl_ip_sport = inet->sport;
    up->pending = AF_INET;
```

③ 调用 IP 层的 ip_append_data 函数发送 UDP 的数据报到 IP 层。ip_append_data 函数将各独立的数据片组建成一个大的数据报。ip_append_data 函数的第二个参数 ip_generic_getfrag 是在 UDP 协议中实现的函数，ip_generic_getfrag 函数完成实际数据复制操作，ip_generic_getfrag 函数是在 IPv4 中执行的回调函数。

```
  do_append_data:
    up->len += ulen;
  //初始化复制数据操作函数

    getfrag=is_udplite ?  udplite_getfrag : ip_generic_getfrag;
    //调用 IP 层的回调函数 ip_append_data，向 IP 层传送 UDP 数据报
    err = ip_append_data(sk, getfrag, msg->msg_iov, ulen,
              sizeof(struct udphdr), &ipc, &rt,
              corkreq ? msg->msg_flags|MSG_MORE : msg->msg_flags);
    //如果数据传送错误，则释放套接字中的数据，清空套接字中等待传送的队列 sk_write_queue
    if (err)
        udp_flush_pending_frames(sk);
```

```
    else if (!corkreq)
          err = udp_push_pending_frames(sk);
    else if (unlikely(skb_queue_empty(&sk->sk_write_queue)))
          up->pending = 0;
    //释放套接字
    release_sock(sk);
```

如果向 IP 层发送数据报失败，则 UDP 释放套接字缓冲区中的数据，该功能由 udp_push_pending_frames 函数实现。清空套接字发送等待队列 sk->sk_write_queue 。

8.4.4 从用户地址空间复制数据到数据报

最后我们简单分析 ip_generic_getfrag 函数是如何从用户地址空间获取数据片，并将数据片追加到数据报中的。该函数是将 UDP 协议传送给 ip_append_data 函数的一个参数，是从 ip_append_data 函数中调用的回调函数，是完成实际地从用户地址空间复制数据到 IP 层的函数。

用户数据存放在一个或多个用户地址空间的缓冲区中，缓冲区的首地址由 iov 变量给出，通常 UDP 数据报一次只有一个缓冲区需要复制。ip_generic_getfrag 函数在进行数据复制时，需完成以下操作流程：

```
int
ip_generic_getfrag(void *from, char *to, int offset, int len, int odd, struct sk_buff *skb)
  {
  struct iovec *iov = from;
```


1. 复制时不需要计算校验和

如果网络设备硬件可以计算校验和，则调用 memcpy_fromiovecend 函数实现用户数据的复制。如果该函数不计算校验和，则将校验和计算留给以后让硬件实现。

```
    if (skb->ip_summed == CHECKSUM_PARTIAL) {
          if (memcpy_fromiovecend(to, iov, offset, len) < 0)
                return -EFAULT;
    }
```

2. 复制时需要计算校验和

调用 csum_partial_copy_fromiovecend 函数计算部分校验和，然后复制数据到 IP 层。

```
else {
        __wsum csum = 0;
         //计算校验和并复制数据，如果复制数据不成功或计算校验和不成功，则返回错误
        if (csum_partial_copy_fromiovecend(to, iov, offset, len, &csum) < 0)
              return -EFAULT;
        //将计算出的校验和存放在 skb->csum 指定的位置
        skb->csum = csum_block_add(skb->csum, csum, odd);
  }
  return 0;
}
```

UDP 校验和包括 UDP 头、IP 的源地址和目标地址、IP 协议头和长度数据域。因此校验和的计算分成 3 个阶段：

① 计算数据的校验和。

② 加入对实际 UDP 协议头校验和的计算。

③ 加入对 IP 协议头校验和的计算。

从用户地址空间复制数据到内核地址空间由 iovec 实用例程完成，在复制时要计算数据的校验和，这样执行效率最高。在 Linux 内核中，TCP/IP 协议栈实现的一个重要原则是避免对同一数据包操作两次。将数据从 iov 指针指向的用户地址空间复制到内核地址空间的局部缓冲区 sk_buff 中。

8.5　UDP 协议接收的实现

传输层所有 AF_INET 协议族的协议实例：TCP、UDP，以及其他一些协议成员都是从网络层 IP 处接收数据包的。在 IP 层处理完数据包后，IP 层将数据包传递给上层协议的接收函数，究竟传输层由哪个协议来接收该数据包，由 IP 协议头中的 protocol 数据域译码得出。根据 protocol 数据域的译码结果作为索引值查询哈希链表 inet_protos[MAX_INET_PROTOS]，来确定是由 TCP、UDP 或其他协议接收函数来接收数据包。

在 8.3 节中我们已讲解了 UDP 协议是如何在 AF_INET 初始化函数 inet_init（net/ipv4/af_inet.c）中调用 inet_add_protocol 函数（net/ipv4/protocol.c），注册其数据包接收处理函数到 inet_protos[MAX_INET_PROTOS]全局哈希链表的。在 UDP 的 struct inet_protocol 数据结构变量实例中，协议域 protocol 的值为 IPPPROTO_UDP，定义在处理函数数据域的是 udp_rcv（net/ipv4/udp.c），udp_rcv 函数是__udp4_lib_rcv 的包装函数，__udp4_lib_rcv 是实际处理来自 IP 层的所有输入数据包的函数。

8.5.1　UDP 协议接收的处理函数

__udp4_lib_rcv 函数是处理 UDP 输入数据包的第一个接收函数，它完成的主要功能是，对接收到的数据包进行正确性检查、地址类型分析（唯一主机地址、组发送或广播地址），调用相应的函数处理输入过程。

1. 输入参数与局部变量

__udp4_lib_rcv 函数是 UDP 协议接收网络数据包的函数，这时的发送是从内核地址空间向用户地址空间复制数据，UDP 协议需要确定接收该数据包的用户进程是哪个。用户进程由与 UDP 协议头中的目标端口号相匹配的套接字给出，每个打开的套接字都保存在 UDP 哈希链表中，便于快速查找。

（1）__udp4_lib_rcv 函数的输入参数

- skb：存放输入数据包的 Socket Buffer。
- proto：传输层使用协议编码。
- udptable：UDP 哈希链表。

（2）局部变量初始化

- sk：指向 UDP 套接字数据结构的指针。
- uh：指向 UDP 协议头。
- ulen：UDP 数据包长度，包括 UDP 协议头和负载数据。

- saddr、daddr：IP 数据包中的源 IP 地址和目标 IP 地址。

```
int __udp4_lib_rcv(struct sk_buff *skb, struct udp_table *udptable, int proto)
                                                                    //net/ipv4/udp.c
{
  struct sock *sk;
  struct udphdr *uh;
  unsigned short ulen;
  struct rtable *rt = (struct rtable*)skb->dst;
  __be32 saddr, daddr;
  struct net *net = dev_net(skb->dev);
```

2. 数据包合法性检查

在对数据包进行处理之前，所做的正确性与合法性检查包括以下几项。

- 协议头长度正确。调用 pskb_may_pull 函数确认在 Socket Buffer 中存放的 UDP 的协议头正确。
- 数据包长度正确。ulen > skb->len。
- UDP 数据包校验和正确。如果数据包合法性检查未通过，扔掉数据包；否则从数据包中提取 UDP 协议的信息头，存放在局部变量 uh 中。

```
  //如果数据包的 UDP 协议头不正确，则扔掉数据包
  if (!pskb_may_pull(skb, sizeof(struct udphdr)))
        goto drop;
  uh= udp_hdr(skb);                       //获取 UDP 协议头
  ulen = ntohs(uh->len);                  //从 UDP 协议头中提取数据包长度信息
  if (ulen > skb->len)                    //查看数据包长度是否正确
        goto short_packet;                //数据包长度检查不正确，扔掉数据包

//如果数据包中 IP 协议头的 protocol 数据域指明该数据包为 UDP 数据包，则查看校验和是否正确，如果不正确，则
//扔掉数据包

  if (proto == IPPROTO_UDP) {
        if (ulen < sizeof(*uh) || pskb_trim_rcsum(skb, ulen))
              goto short_packet;
        uh = udp_hdr(skb);
  }
  if (udp4_csum_init(skb, uh, proto))
        goto csum_error;
```

3. 根据数据包的地址类型确定如何发送数据包

当路由表入口指明该输入数据包的发送地址是广播地址或组发送地址时，调用 __udp4_lib_mcast_deliver 函数完成发送过程。

__udp4_lib_lookup_skb 用数据包中的 UDP 协议头信息查询 UDP 哈希链表，以确定在 UDP 协议头中指定的端口上，是否有已打开的套接字在等待接收数据；如果有打开的套接字在等待接收数据，则该数据包就由 udp_queue_rcv_skb 函数发送给套接字的接收缓冲区队列，接收处理过程完成。

```
//从 IP 协议头信息中提取数据的源 IP 地址与目标 IP 地址
  saddr = ip_hdr(skb)->saddr;
  daddr = ip_hdr(skb)->daddr;
//如果路由标志为组传送或广播地址，则完成数据包的广播传送与组传送

  if (rt->rt_flags & (RTCF_BROADCAST|RTCF_MULTICAST))
```

```
        return __udp4_lib_mcast_deliver(net, skb, uh,
                    saddr, daddr, udptable);
//是否有打开的套接字等待接收数据包
  sk = __udp4_lib_lookup_skb(skb, uh->source, uh->dest, udptable);
  if (sk != NULL) {                                    //有打开的套接字在等待接收数据包
        int ret = udp_queue_rcv_skb(sk, skb);          //将数据包传给套接字接收缓冲区
        sock_put(sk);                                  //释放套接字
        if (ret > 0)
              return -ret;
        return 0;
  }
```

当 udp_queue_rcv_skb 函数的返回值大于 0 时，__udp4_lib_rcv 函数需告诉调用程序重新提交输入数据包。

4. 没有打开的 UDP 套接字

如果当前没有打开的套接字在等待接收输入数据包，则对数据包的处理要完成以下过程：

- 完成数据包校验和计算：如果校验和检验不正确，则数据包扔掉，因为 UDP 是不可靠发送协议，这时接收方不向数据发送方返回错误信息。
- 校验和检验正确：其一，更新 UDP_MIB_NOPORTS 错误统计信息；其二，向数据包发送端返回 ICMP 错误信息，告知端口不可到达，释放 Socket Buffer。

```
  if (!xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb))
        goto drop;
  nf_reset(skb);

  //udp_lib_checksum_complete 完成校验和计算，如果不正确退出程序
  if (udp_lib_checksum_complete(skb))
        goto csum_error;
  //校验和计算正确，更新错误统计信息，向发送方返回端口不可到达 ICMP 消息
  UDP_INC_STATS_BH(net, UDP_MIB_NOPORTS, proto == IPPROTO_UDPLITE);
  icmp_send(skb, ICMP_DEST_UNREACH, ICMP_PORT_UNREACH, 0);
  kfree_skb(skb);
  return 0;
```

5. 错误处理

当函数最后为在前面对数据包进行合法性检查时，如果发现坏数据包，则扔掉数据包，更新接收错误的统计信息。

8.5.2　将数据包放入套接字接收队列的处理函数

在 8.5.1 节的__udp4_lib_rcv 函数中，数据包合法检查正确后，__udp4_lib_rcv 函数以 UDP 协议头中的目标端口号为关键字，在 UDP 哈希链表中查询匹配的套接字，确认在数据包目标端口上有打开的套接字在等待接收该数据包。__udp4_lib_rcv 会调用 udp_queue_rcv_skb 函数将数据包放入套接字的接收缓冲区队列。

udp_queue_rcv_skb 函数根据套接字是轻套接字还是常规套接字，来确定数据包的接收函数。当套接字为常规套接字时，接收函数执行的关键步骤为：

① 锁定套接字。

② 获取等待接收数据包的用户进程。

③ 数据包放入套接字接收队列。

如果 udp_queue_rcv_skb 函数获取用户进程成功，则数据包就由__udp_queue_rcv_skb 函数放入套接字的接收缓冲区队列。

如果 udp_queue_rcv_skb 函数没有发现等待接收数据包的用户进程，则将数据包放入套接字的 backlog 队列，等待以后的套接字接收。

```
int udp_queue_rcv_skb(struct sock * sk, struct sk_buff *skb)    //net/ipv4/udp.c
{
…
bh_lock_sock(sk);

//如果套接字上有用户进程在等待接收数据包，则将数据包放入套节字接收缓冲区队列

if (!sock_owned_by_user(sk))
        rc = __udp_queue_rcv_skb(sk, skb);
//如果套接字上无用户进程等待接收数据包，则将数据包放入 backlog 队列
  else
        sk_add_backlog(sk, skb);
  bh_unlock_sock(sk);
…
}
```

8.5.3 UDP 协议接收广播与组发送数据包

组发送和广播数据包会传给多个目标地址，在同一个主机上就可能有多个目标端口在等待接收数据包。当 UDP 协议实例处理组发送或广播数据包时，协议接收函数查看是否有多个打开的套接字要接收该数据包。当输入数据包的路由表入口标志设置了组发送或广播发送标志时，UDP 的接收函数__udp4_lib_rcv 就调用 UDP 协议的组接收函数__udp4_lib_mcast_deliver，将数据包分发给所有有效的侦听套接字，函数的处理流程需要遍历 UDP 哈希链表，找到所有接收数据包的套接字。

1. 找到第一个有效的套接字

首先锁定 UDP 的哈希链表，然后在哈希链表中找到第一个打开的套接字，该套接字的端口号应与输入数据包 UDP 协议头中的目标端口号相匹配。

```
static int __udp4_lib_mcast_deliver(struct net *net, struct sk_buff *skb,
                        struct udphdr  *uh,
                        __be32 saddr, __be32 daddr,
                        struct udp_table *udptable)                    //net/ipv4/udp.c
{
  struct sock *sk;
  struct udp_hslot *hslot = &udptable->hash[udp_hashfn(net, ntohs(uh->dest))];
  int dif;

//锁定 UDP 哈希链表，在哈希链表中找到第一个匹配的目标套接字

  spin_lock(&hslot->lock);
  sk = sk_nulls_head(&hslot->head);
  dif = skb->dev->ifindex;
```

```
    sk = udp_v4_mcast_next(net, sk, uh->dest, daddr, uh->source, saddr, dif);
    if (sk) {                                                    //如果找到第一个匹配的套接字
          struct sock *sknext = NULL;
```

2. 遍历 UDP 哈希链表，发送数据包

接下来，函数遍历 UDP 协议的哈希链表查看每一个匹配的入口。一旦函数找到了一个正在侦听的匹配套接字，函数就复制一个套接字。将克隆好的 Socket Buffer 放到侦听套接字的接收缓冲区队列中（调用 udp_queue_rcv_skb 函数）。

```
          do {

              //遍历 UDP 哈希链表，每找到一个匹配的套节字，复制一个套接字放入套接字缓冲区队列
                struct sk_buff *skb1 = skb;
                sknext = udp_v4_mcast_next(net, sk_nulls_next(sk), uh->dest,
                                   daddr, uh->source, saddr,
                                   dif);
                if (sknext)
                      skb1 = skb_clone(skb, GFP_ATOMIC);
                if (skb1) {
                      int ret = udp_queue_rcv_skb(sk, skb1);
                      if (ret > 0)
```

当 udp_queue_rcv_skb 函数的返回值大于 0 时，在源代码的注释中提示应重新处理数据包而不是在这里扔掉数据包。当前 Linux 内核版本中没有这样的处理。

```
                            kfree_skb(skb1);
                }
                sk = sknext;
          } while (sknext);
    } else
          kfree_skb(skb);
    spin_unlock(&hslot->lock);
    return 0;
}
```

8.5.4　UDP 的哈希链表

一个应用程序在打开的 SOCK_DGRAM 类的套接字上接收 UDP 的数据包，这类套接字可以在各种地址与端口相组合的套接字上侦听。例如，它接收的数据包的目标 IP 地址可以为任何地址，但端口是指定端口；它也可以接收传送给组发送地址或广播地址的数据包。到达__udp4_lib_rcv 函数的数据包可能需要传送给多个侦听套接字；这样 UDP 协议要解决的一个问题是，如何快速找到接收数据包的某个或多个套接字，来提高网络性能。Linux 使用 UDP 哈希链表来快速查找侦听数据包的套接字。

1. UDP 的哈希链表数据结构

```
struct udp_table {                                              //include/net/udp.h
  struct udp_hslot hash[UDP_HTABLE_SIZE];
};
struct udp_hslot {
  struct hlist_nulls_head     head;
  spinlock_t       lock;
```

```
} __attribute__((aligned(2 * sizeof(long))));
extern struct udp_table udp_table;
```

哈希链表一共有 128 个槽，哈希链表中的每一个位置指向一个 struct sock 数据结构的链表。在哈希链表中搜索的索引值是由 UDP 端口号的低 7 位计算出来的。

根据哈希值在哈希链表中搜索目标套接字的函数是__udp4_lib_lookup_skb，该函数根据收到的 Socket Buffer 中的 IP 协议头获取数据包的源 IP 地址、目标 IP 地址，以及来自套接字数据结构的源端口号和目标端口号，在 UDP 哈希链表中搜索目标套接字。实际完成搜索功能的是__udp4_lib_lookup 函数。

2. 在哈希链表中查找套接字的实现函数

（1）__udp4_lib_lookup_skb 函数

```
static inline struct sock *__udp4_lib_lookup_skb(struct sk_buff *skb,
                                __be16 sport, __be16 dport,
                                struct udp_table *udptable)
{
  struct sock *sk;
  const struct iphdr *iph = ip_hdr(skb);

  if (unlikely(sk = skb_steal_sock(skb)))
       return sk;
  else
       return __udp4_lib_lookup(dev_net(skb->dst->dev), iph->saddr, sport,
                        iph->daddr, dport, inet_iif(skb),
                        udptable);
}
```

（2）__udp4_lib_lookup 函数

从__udp4_lib_lookup_skb 中调用另一个函数__udp4_lib_lookup，试着在哈希链表中找到最适合的套接字。至少目标端口号与侦听套接字的端口号相匹配。随后该函数用更多的条件：源 IP 地址、目标 IP 地址和输入网络接口索引号，来提取更多匹配的套接字。

```
static struct sock *__udp4_lib_lookup(struct net *net, __be32 saddr,
        __be16 sport, __be32 daddr, __be16 dport,
        int dif, struct udp_table *udptable)               //net/ipv4/udp.c
{
  struct sock *sk, *result;
  struct hlist_nulls_node *node;
  unsigned short hnum = ntohs(dport);                      //查询关键字为目标端口号
  unsigned int hash = udp_hashfn(net, hnum);               //根据目标端口号计算 hash 值
  struct udp_hslot *hslot = &udptable->hash[hash];
  int score, badness;
//获取 UDP 哈希链表读操作锁，开始在哈希链表中搜索

  rcu_read_lock();
begin:
  result = NULL;
  badness = -1;
  sk_nulls_for_each_rcu(sk, node, &hslot->head) {
```

这里我们以目标端口号为条件，在哈希链表中查找与输入数据包目标地址最匹配的打开套接字，首先找到第一个匹配的套接字，这个套接字会接收输入数据包。

在 compute_score 函数中用更多的条件来查找匹配的套接字，如果获取的结果为 NULL，

get_nulls_value(node) != hash，即插槽中的 hash 值与用于搜索的 hash 值不匹配，程序跳转到 goto begin 处重新开始查找。

```
        //用更多的条件来查找匹配的套接字，计算匹配的可能性，如 score>badness，找到可能的套接字
          score = compute_score(sk, net, saddr, hnum, sport,  daddr, dport, dif);
          if (score > badness) {
                result = sk;
                badness = score;
          }
    }
    //如果哈希链表插槽中的 hash 值与用于搜索的 hash 值不匹配，则重新开始搜索
    if (get_nulls_value(node) != hash)
          goto begin;
    if (result) {
    //找到可能的套接字，对套接字的引用计数加 1
          if (unlikely(!atomic_inc_not_zero(&result->sk_refcnt)))
                result = NULL;
          else if (unlikely(compute_score(result, net, saddr, hnum, sport,
                      daddr, dport, dif) < badness)) {
                sock_put(result);
                goto begin;
          }
    }
    rcu_read_unlock();
    return result;
}
```

（3）compute_score 函数

在 compute_score 函数中，用更多的条件来搜索匹配的套接字，如目标地址、网络设备接口等。

```
static inline int compute_score(struct sock *sk, struct net *net,
                    const unsigned short hnum, const __be32 daddr,
                const int dif)                                    //net/ipv4/inet_hashtables.c
{
    int score = -1;
    struct inet_sock *inet = inet_sk(sk);
    //套接字不是一个 IPv6 的套接字
    if (net_eq(sock_net(sk), net) && inet->num == hnum &&
                !ipv6_only_sock(sk)) {
          __be32 rcv_saddr = inet->rcv_saddr;
          //地址类型是否正确
          score = sk->sk_family == PF_INET ? 1 : 0;
          if (rcv_saddr) {
          //查找匹配的目标地址
                if (rcv_saddr != daddr)
                      return -1;
                score += 2;
          }
          //最后查看数据包的输入网络接口是否匹配
```

```
        if (sk->sk_bound_dev_if) {
            if (sk->sk_bound_dev_if != dif)
                return -1;
            score += 2;
        }
    }
    return score;
}
```

8.5.5 将数据包放到套接字接收队列

udp_queue_rcv_skb 函数是将 UDP 数据包放到套接字接收队列中的函数，该函数是在套接字被用户进程持有时从后半段 bottom half 中调用的，这时套接字不能再接收其他数据。该函数是在编译时初始化给 UDP struct proto 数据结构的 backlog_rcv 数据域。UDP 的接收函数__udp4_lib_rcv 调用 udp_queue_rcv_skb 函数来完成对输入数据包的处理，该函数的主要任务是将输入数据包放入套接字的接收缓冲区队列中，并更新 UDP 输入统计信息。

首先，查看套接字的接收队列是否已满，如果已满则扔掉数据包。

```
int udp_queue_rcv_skb(struct sock * sk, struct sk_buff *skb)          //net/ipv4/udp.c
{
    struct udp_sock *up = udp_sk(sk);
    int rc;
    int is_udplite = IS_UDPLITE(sk);
    if (!xfrm4_policy_check(sk, XFRM_POLICY_IN, skb))
        goto drop;
    nf_reset(skb);
```


其次，查看该套接字是否为 IPSec 协议封装的套接字，如果是一个封装的套接字，输入的数据包一定被封装在 IPSec 协议的数据包中，则要将数据包从封装的数据包中解析出来并接收，并调用 up->encap_rcv 函数来处理该数据包。

```
    if (up->encap_type) {                                    //该数据包是封装的数据包
        if (skb->len > sizeof(struct udphdr) &&
            up->encap_rcv != NULL) {                         //处理封装数据包函数不为空
            int ret;
            ret = (*up->encap_rcv)(sk, skb);                 //由封装处理函数处理数据包
            if (ret <= 0) {                                  //如果处理不正确，则更新统计信息
                UDP_INC_STATS_BH(sock_net(sk),
                                 UDP_MIB_INDATAGRAMS,
                                 is_udplite);
                return -ret;
            }
```

如果套接字不是封装的套接字，则数据包是普通数据包。首先完成对数据包校验和的计算。如果校验和计算不正确，则扔掉数据包。

然后调用套接字层的接收函数__udp_queue_rcv_skb，将数据放入套接字的接收缓冲区队列中，并唤醒套接字，这样应用程序就可以从套接字接收队列中读取接收到的数据。

如果套接字没有被有效的用户进程持有，则数据包由 sk_add_backlog 函数放入套接字的 backlog 队列中。

```
    if (sk->sk_filter) {
```

```
            if (udp_lib_checksum_complete(skb))
                  goto drop;
      }
…
      bh_lock_sock(sk);
      if (!sock_owned_by_user(sk))
            rc = __udp_queue_rcv_skb(sk, skb);
      else
            sk_add_backlog(sk, skb);
      bh_unlock_sock(sk);
…
```

8.6　UDP 协议在套接字层的接收处理

当应用程序在一个打开的套接字上调用读 read 函数时，套接字层就调用 struct proto 数据结构中 rcvmsg 数据域指向的函数处理应用程序的读操作。对于 UDP 协议，在编译时间 rcvmsg 数据域初始化指向 udp_rcvmsg 函数。udp_rcvmsg 函数是在所有 SOCK_DGRAM 类套接字上执行的接收函数。在本节中，我们将分析 udp_rcvmsg 函数实现流程。

8.6.1　函数输入参数

函数的输入参数有如下几种。

- struct kiocb *iocb：应用层 I/O 控制缓冲区。
- struct sock *sk：指向接收数据包的套接字结构。
- struct msghdr *msg：接收数据包应用层缓冲块及处理函数。
- size_t len：数据信息长度。
- int noblock：当没有数据接收时，应用程序是否阻塞的标志。
- int flag：套接字接收队列中的数据包信息标志。
- int *addr_len：应用层存放发送方地址长度。

函数使用输入参数来初始化部分局部变量以控制读操作，在函数起始部分初始化的局部变量包括 INET 协议族套接字结构 struct inet_sock *inet，inet 局部变量初始数据来源于输入参数 sk。从套接字名 msg->msg_name 中提取数据包目标 IP 地址信息存放在局部变量 struct sockaddr_in *sin 中。局部变量 ulen 将用于存放要复制的数据总长度。最后数据通过多次循环复制到缓冲区，局部变量 copied 存放一次循环已复制的数据长度。

8.6.2　函数处理流程

首先设置用户地址空间存放发送端地址长度的参数，查看套接字的错误信息队列中是否有任何错误消息需要处理，如果套接字错误消息队列中有数据，则调用 ip_recv_error 函数来处理错误信息。

接收函数调用__skb_recv_datagram 函数从套接字的接收缓冲区队列中读取下一个数据包缓冲区，该缓冲区的首地址存放在局部变量 skb 中。如果 skb 指针为空，说明套接字接

收缓冲区队列中没有等待读入的数据包，结束函数处理。

```
int udp_recvmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
        size_t len, int noblock, int flags, int *addr_len)                //net/ipv4/udp.c
{
  //获取数据包发送端源地址长度
  if (addr_len)
        *addr_len=sizeof(*sin);
  //如果套接字的错误消息队列有信息需要处理
  if (flags & MSG_ERRQUEUE)
        return ip_recv_error(sk, msg, len);
try_again:
  //从套接字接收缓冲区队列中读取数据包
  skb = __skb_recv_datagram(sk, flags | (noblock ? MSG_DONTWAIT : 0),
                    &peeked, &err);
  if (!skb)
        goto out;
```

查看用户程序是否需要读取比当前数据包负载中更多的数据。在网络数据包的发送过程中，一个重要的原则是避免对数据进行多次处理，提高网络数据吞吐量。在接收网络数据包的过程中，如果在对数据包进行校验和标志时要求对数据包进行全校验和检验，则应在复制数据的同时完成校验和处理。

反之，如果只对数据包进行部分校验和检验，则应在数据复制前完成校验和处理。这时的处理过程为：

- 对数据包进行部分校验和检验，数据跨越了多个 skb，如果部分校验和出错，则扔掉数据包。
- 如果不需要对数据包进行校验和（由 skb->ip_summed 数据域确定），则调用 skb_copy_datagram_iovec 函数将 skb 中的数据从内核地址空间复制到用户地址空间。
- 如果需要对数据包做校验和检验，则调用 skb_copy_and_csum_datagram_iovec 函数将 skb 中的数据从内核地址空间复制到用户地址空间，同时完成校验和计算。

```
  //从数据包中扔掉 UDP 协议头
  ulen = skb->len - sizeof(struct udphdr);
  //调整一次循环复制的数据长度
  copied = len;
  if (copied > ulen)
        copied = ulen;
  else if (copied < ulen)
        msg->msg_flags |= MSG_TRUNC;
  if (copied < ulen || UDP_SKB_CB(skb)->partial_cov) {
        //如果只做部分校验和检验，先完成校验和处理再复制数据
        if (udp_lib_checksum_complete(skb))
              goto csum_copy_err;
  }
  //如果不需做校验和检验，则直接复制数据
  if (skb_csum_unnecessary(skb))
        err = skb_copy_datagram_iovec(skb, sizeof(struct udphdr),
                              msg->msg_iov, copied);
  else {
//如果对数据包进行全校验检验，则在复制数据的同时完成校验和计算
        err=skb_copy_and_csum_datagram_iovec(skb,sizeof(struct udphdr), msg->msg_iov);
```

这时标记数据包的接收时间戳，如果用户应用程序提供了有效的缓冲区 sin，来接收数据包发送端源 IP 地址和端口号，则将数据包的源地址信息从数据包的协议头中复制到 sin 指定的用户缓冲区中。

```
//为数据包设置接收时间戳
sock_recv_timestamp(msg, sk, skb);
//如果应用程序提供了缓冲区 sin，来存放数据发送端的源 IP 地址和源端口号，则将数据包发送端地址复制到 sin 中
  if (sin)
  {
        sin->sin_family = AF_INET;
        sin->sin_port = udp_hdr(skb)->source;
        sin->sin_addr.s_addr = ip_hdr(skb)->saddr;
        memset(sin->sin_zero, 0, sizeof(sin->sin_zero));
  }
```

在结束函数处理之前，调用 udp_recvmsg 函数查看控制信息标志域 cmsg_flags，看是否设置了任何 IP 套接字选项。例如，IP_TOS IP 选项要求将 IP 协议头复制到用户地址空间。如果 IP 协议头中设置了任何控制标志，则调用 ip_cmsg_rev 函数完成对 IP 选项值的提取。

```
  if (inet->cmsg_flags)
        ip_cmsg_recv(msg, skb);
  err = copied;
  if (flags & MSG_TRUNC)
        err = ulen;
```

最后是函数错误处理出口，udp_recvmsg 函数有以下 3 处错误处理出口：

- 如果从内核地址空间复制数据到用户地址空间不成功，则释放 Socket Buffer 和用户进程持有的套接字。
- 当从套接字接收队列读取一个数据包时，队列已空，直接返回。
- 当校验和检验出错时，扔掉数据包，更新错误统计信息，释放用户进程持有的套接字。

8.7　本章总结

本章沿着网络数据在 TCP/IP 协议栈中的传输路径，描述了数据在传输层 UDP 协议实例中的处理过程。UDP 是 TCP/IP 协议栈在传输层上另一个重要的协议实例，相比 TCP 协议，UDP 实现的是更简单、快速的、不可靠、无连接数据传输服务。从本章的内容可以看到，UDP 的接收/发送数据处理过程比 TCP 简单很多，因为它不需要维护连接状态。本章从以下几个方面介绍了 Linux 内核实现 UDP 协议的技术特点：

- 描述 UDP 数据报格式的数据结构。
- UDP 与套接字层、网络层 IP 协议的接口数据结构、接口初始化特点。
- 数据包在 UDP 协议中的发送途径。
- UDP 接收/发送网络数据包主要功能函数的实现流程、代码分析。

第 9 章　传输层 TCP 协议的实现

本章主要介绍了传输层 TCP 协议的功能、TCP 协议管理状态连接机制、TCP 协议时间等待状态的管理。从而介绍了 TCP 协议实现面向连接、可靠数据传送的原理。在此基础上讲解了 Linux 内核中实现 TCP 协议数据的结构、数据传输功能函数、TCP 连接状态数据管理的实现与流程分析。

传输层的主要任务是保证数据能准确传送到目的地。在 TCP/IP 协议栈中这个功能由传输控制协议（TCP，Transmission Control Protocol）完成。同时传输层紧接在应用层下，它也是用户地址空间的数据进入内核地址空间的接口。TCP/IP 协议栈在传输层提供两个最常用的协议：TCP 和 UDP。UDP 协议在 Linux 内核中的实现在第 8 章已经介绍了，这里就不再赘述。本章主要介绍 TCP 协议在 Linux 内核中的实现。

9.1　CP 协议简介

应用程序使用传输层的 TCP 协议来完成数据的可靠传输，是因为 TCP 协议能保证数据完整准确地、按正确的序列在网络上传送到目标地址。TCP 协议的特性是提供可靠的、面向连接、字节流传送服务。TCP 实现传送的过程为首先在通信双方：客户端与服务器之间提供连接，一个 TCP 客户与某个给定服务器建立连接，通过建立的连接与服务器交换数据，最后结束连接。

9.1.1　CP 是可靠协议

TCP 协议使用肯定回答和重传（PAR：Positive Acknowledgment with Re-transmission）机制来提供可靠数据传送功能。一方面，在使用 TCP 协议传送数据后，发送方要等待接收方的回答消息；系统如果没有得到远端系统的回答，TCP 自动重传数据并等待更长的时间。如果几次重传后仍不成功，则 TCP 协议放弃重传。TCP 协议花在重传上的时间通常在 4~10 分钟之间。TCP 协议包含了一个算法来估算数据段在客户端与服务器之间的动态传送时间 RTT（RTT：round-trip time），以确定要等待多长时间能收到回答信息。

另一方面，在 TCP 两端协议协作数据的交换单位为数据段。每个数据段中都有校验和来检验数据是否受损。如果在接收方接收到完好无损的数据段，则接收方需向数据发送方返回正确的回答信息。如果数据段被破坏，接收方扔掉数据段，在一段时间后，发送数据段的 TCP 模块重传没有收到接收正确回答信息的数据段。TCP 数据段的格式如图 9-1 所示。

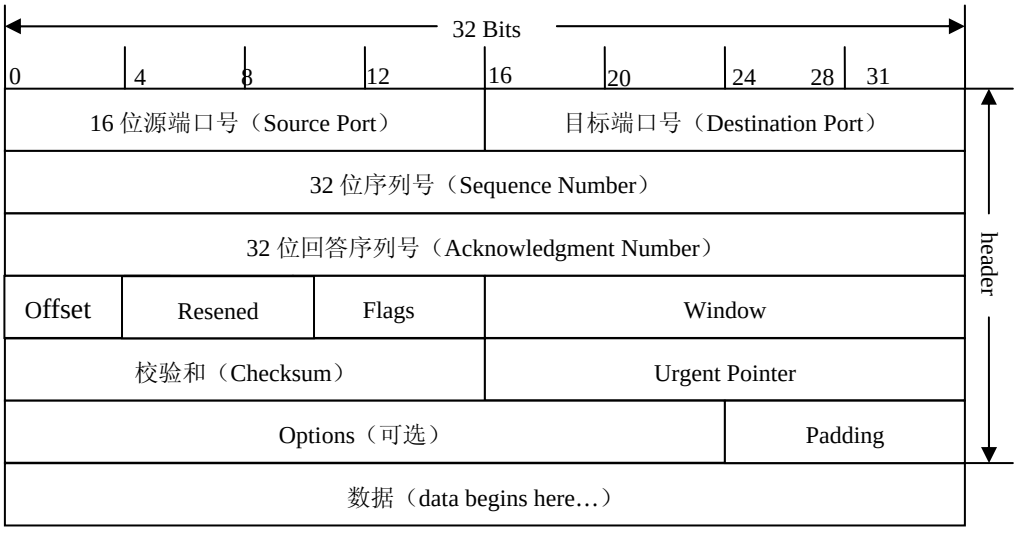


图 9-1　TCP 数据段的格式

- 16 位的源端口号（Source Port）：指明系统中发送数据段的一个进程。
- 16 位的目标端口号（Destination Port）：指明数据接收端的一个进程。
- 32 位序列号（Sequence Number）：指明数据段中第一个数据字节的序列号。
- 32 位回答序列号（Acknowledgment Number）：如果设置了回答控制位，该值指明了发送方的下一个序列号。
- Offset：协议头长度，该域占 4 位，指明协议头占多少个 32 位字的单元。协议头最长为 15x4（32 位，4 个字）= 60 个字节，即协议头最多有 15 个单元的 32 位字组成。
- Reserved：预留为将来使用，必须设置为 NULL。
- Flags：划分以下几种标志。
  - URG（ Urgent Pointer ）：指明紧接着传送的是重要数据。在设置标志位后，协议头中的 Urgent Pointer 指针有效。
  - SYN：指明传送的数据段是用于建立连接的同步数据段，如果 SYN=1，则要求建立连接。
  - ACK：如果设置该标志位，则指明传送的数据段是回答数据段。
  - RST：如果设置该标志位，则传送的数据段是要求复位连接的数据段。如果 RST = 1，则指明要求复位一个连接。
  - PSH：如果设置该标志位，则 TCP 协议实例应立即将接收到的数据传送给上层应用。
  - FIN：如果设置该标志位，则表明连接已断开。
- Window：指明接收数据方还有多少缓冲区空间来接收数据。
- Checksum：包含 TCP 协议头和数据所做的校验和。
- Urgent Pointer：指明重要数据的最后一个字节地址。
- Options：TCP 协议选项，该数据域的长度可变。

9.1.2 TCP 是面向连接的协议

TCP 协议在两个通信的主机之间建立点到点的逻辑连接。在数据开始传送之前，两个端点之间通过交换握手控制信息来建立对话连接。TCP 协议通过在协议头的 flags 字段设置标志位，来指明当前传送的数据段是控制信息还是负载数据。

1. 建立连接与终止

TCP 使用的握手控制信息称为“三次握手”（three-way handshake），因为在建立连接时两个端点之间一共要交换 3 个控制数据段。图 9-2 给出了最简单的三次握手的方式。

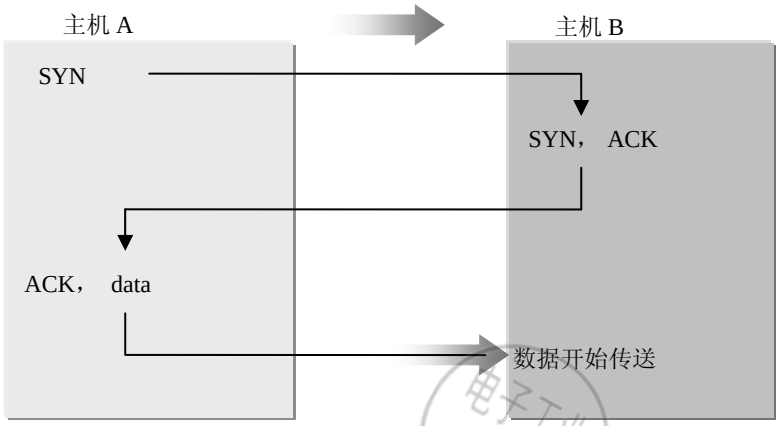


图 9-2 TCP 协议三次握手建立连接

主机 A 要与主机 B 建立连接，首先主机 A 要向主机 B 发送一个数据段，该数据段设置了同步序列号（SYN，Synchronize sequence numbers）位。这个数据段告诉主机 B，主机 A 要与之建立连接，并告诉主机 B 它传送数据的起始序列号（序列号的作用是使数据以正确的顺序传送）。然后主机 B 返回主机 A 一个回答数据段，该数据段设置了回答位（ACK）和同步位（SYN）标志。B 的回答数据段是响应 A 的请求数据段，并通知主机 A、B 的起始序列号。最后主机 A 送出收到主机 B 回答的确认数据段，然后开始传送实际的数据。

在以上的三次握手信息交换后，主机 A 的 TCP 实例可以确定远程 TCP 协议实例启动，并准备好接收数据。一旦建立连接，就可以开始传送数据。当 TCP 两端交互的模块结束数据传送后，它们又将交换三次握手数据段，这些数据段中设置了“发送方已无数据传送”的结束位标志（FIN 位）来关闭连接。TCP 协议提供的是在两个系统之间建立点到点数据交换的逻辑连接。

2. TCP 选项

建立连接与断开连接的同步数据段中可以包含 TCP 选项，最常用的选项有以下几种。

（1）MSS 选项

在 TCP 发送 SYN 数据段时如果设置了该选项，则它声明 TCP 的最大数据段大小，即在该连接上每个接收的 TCP 数据段中的最大字节数。TCP 数据段发送方使用它从数据段接收方收到的 MSS 值作为其传送数据段的最大值。

（2）Window 尺寸选项

在 TCP 协议头中描述窗口尺寸的数据域占 16 位，相应的 TCP 两端能向对方发布的窗口最大尺寸是 65535。目前高速 Internet 连接或长延迟路径（卫星连接）网络上需要更大的窗口尺寸，以获取最大数据吞吐量。新的选项规范指出在 TCP 协议头中窗口尺寸由该数据域的 0～14 位来发布，最大窗口尺寸可以达到 1GB（65535×2^{14}）。连接的两端必须都支持该窗口尺寸。

（3）时间戳选项

该选项用于高带宽网络连接中，避免新发送数据段与旧数据段、延迟数据段或复制数据段发生冲突。

9.1.3　TCP 是按字节流交换的协议

TCP 是以连续字节流的格式交换数据的，而不是独立的数据包。因此 TCP 协议要维护字节传送的顺序。在 TCP 协议头中的序列号和回答序列号就是用于跟踪字节传送顺序。

TCP 协议标准并没有规定系统必须以某个特定的序列号为起始序列号，各系统可自行选择一个起始序列号作为数据传送的开始。为了保证正确跟踪字节流，连接的两端必须知道对方的起始序列号。连接的两端在握手期间，通过交换同步数据段来同步字节数。在握手同步数据段的序列号数据域中包含了初始序列号（ISN），ISN 是传送字节数起始点。出于安全的考虑，ISN 应该是一个随机数。

传送数据的每个字节从 ISN 开始顺序计数，发送的第一个数据字节序列号是 ISN+1。数据段头中的序列号指明了数据段中第一个字节在数据流中的顺序位置。例如，如果在数据流中第一个字节的序列号是 1（ISN =0），有 4000 字节的数据已经传送，这时当前段中数据的第一个字是 4001，序列号也应为 4001。

回答控制数据段有两个功能：确认回答和流控制。确认信息告诉数据发送方已收到多少字节的数据，接收方还能接收多少字节的数据。TCP 协议规范并不要求每个数据包都需要一个独立的回答，回答段的序列号是接收到的所有字节数的和。例如，第一个发送的字节的序列号是 1，有 2000 个字节已正确接收，回答序列号就是 2001。

在协议头中的窗口域包含了一个窗口，或称为远端系统还可以接收的字节数。如果接收端还可以接收另外 6000 个字节，窗口就为 6000。接收方通过改变窗口的大小来控制发送方的字节流。如果窗口值为 0，则告诉发送方停止向其继续发送数据，直到发送方收到窗口不为零后，再继续发送。图 9-3 给出了一个 TCP 传送字节流控制的示例。

在图 9-3 中 TCP 数据流的起始序列号为 0。接收系统已收到 2000 个字节并给出了回答，当前回答序列号就是 2001。这时接收方还有足够的缓冲区空间可以接收 6000 个字节，那么窗口的大小就是 6000。这时发送方传送了一个 1000 字节的数据段，起始序列号为 4001。从 2001 开始发送端就没有收到回答信号，在窗口值范围内它继续发送数据。如果发送方填满了接收方的窗口，仍没有收到对它以前发送的数据段的回答，发送方将等待一定的时间，等待超时后，发送方重发未收到确认回答的所有数据。

在图 9-3 中，如果一直未收到确认回答，则重传将从 2001 字节开始，这个过程将保证在网络的另一端可靠地接收到数据。

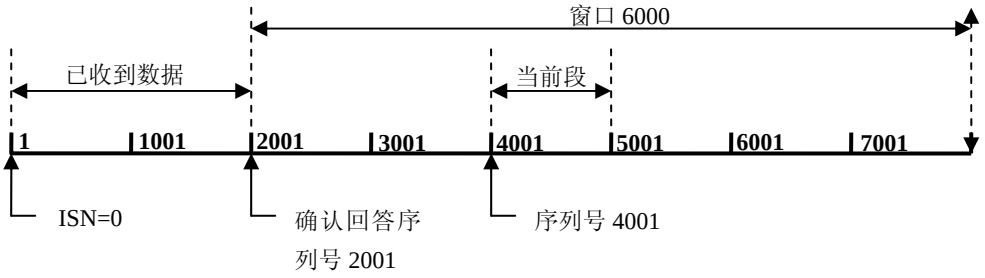


图 9-3　TCP 传送字节流控制

最后 TCP 协议也要负责将从 IP 接收到的数据传送给正确的应用。与数据绑定的应用由 16 位的端口号标识。源端口号与目的端口号都包含在 TCP 的协议头中，从应用层正确接收数据和将数据正确发送给应用层，也是 TCP 协议提供的服务中重要的部分。

9.1.4　TCP 协议实现的功能

从以上对 TCP 协议特性的描述我们可以总结出，要实现 TCP/IP 协议栈中传输层的 TCP 协议实例，应完成以下功能。

- 数据收发：包括 TCP 从应用进程接收数据、通过网络发送和接收数据，将数据传送到正确的应用进程。
- 连接管理：在数据开始传送之前，建立数据交换两端的连接；数据传送完成以后，关闭连接。
- 流控制管理：保证数据按正确的顺序被接收。
- 回答超时管理：如果发送数据端没收到确认回答的数据，则在一定时间后应重发数据。

在本章中我们将从以上几个方面来分析，TCP/IP 协议栈中的 TCP 协议实例在 Linux 内核中的实现。

9.2　描述 TCP 协议实现的关键数据结构

TCP 连接两端如何处理发送/接收 TCP 数据段，这些信息包含在 TCP 协议头中和 TCP 协议选项中。另外传输层是 TCP/IP 协议栈与应用层之间的接口，TCP 协议需要接收来自应用层的数据或向应用层传送 TCP/IP 协议栈通过网络接收到的外部数据段，这些数据段如何在用户地址空间与内核地址空间之间进行交换，在内核中需要相应的数据结构和 API 来描述、操作。

9.2.1　TCP 协议头数据结构

TCP 协议头描述了 TCP 数据段发送的源地址、目标地址、数据段传送管理和连接管理

的信息，是 TCP 协议实现的重要数据结构之一。TCP 协议头由数据结构 struct tcphdr 描述，struct tcphdr 数据结构字段的设置与在 9.1.1 中描述的 TCP 协议数据段格式相对应，struct tcphdr 数据结构定义在 include/linux/tcp.h 文件中。

其中，doff 为协议头长度，紧随该数据域后的就是控制标志位 flags 的定义，flags 的最后两位 ece：1，cwr：1 用于网络拥塞和窗口控制，在原 RFC 定义中没有。在 TCP 数据段传送时，传送函数根据 struct tcphdr 数据结构的设置来创建 TCP 协议头。

9.2.2　TCP 的控制缓冲区

TCP 是完全异步的协议，实际数据段的传送独立于所有来自套接字层的写操作。TCP 层分配 Socket Buffer 来存放应用层写入套接字的数据，但应用程序控制管理数据包的信息存放在 TCP 的控制缓冲区中。TCP 控制缓冲区由 struct tcp_skb_cb 数据结构描述。可以看到 TCP 控制缓冲区中的信息与 TCP 协议头的信息有部分重合，当数据从应用程序复制到 TCP 层的 Socket Buffer 时，函数需要用 TCP 控制缓冲区中 TCP 协议头的信息，来设置 struct tcphdr 数据结构中的相关数据域。

```
struct tcp_skb_cb {                                          //include/net/tcp.h
  union {
        struct inet_skb_parm   h4;
#if defined(CONFIG_IPV6) || defined (CONFIG_IPV6_MODULE)
        struct inet6_skb_parm  h6;
#endif
  } header;
  __u32      seq;
  __u32      end_seq;
  __u32      when;
  __u8       flags;
  __u8       sacked;
  __u32      ack_seq;
};
```

- inet_skb_parm、inet6_skb_parm：存放输入数据段的 IP 选项；一个用于 IPv4 协议，一个用于 IPv6 协议。
- seq：输出数据段的起始序列号。
- end_seq：最后一个输出数据段结束序列号。结束序列号的值为最后发送的数据段序列号 + 一个 SYN 数据段 + 一个 FIN 段 + 数据段长度，即 end_seq = seq + SYN + FIN + 当前段长度。
- when：用于计算 RTT（Round-trip time）的值，即数据段在网络上的传送时间。when 数据域用来管理数据段传输起始计时和数据重传。TCP 数据段发送方根据 when 的值计算传出数据后要等待多长时间，如果等待一定时间后仍未收到数据接收端回答信息（ACK），就重传数据段。
- flags：与 TCP 协议头中的 flags 数据域定义相同。这个数据域中的值应与 TCP 协议头中 flags 标志的设置相匹配。
- sacked：保存了选择回答（SACK ：Selective Acknowledge）和前送回答（FACK：Forward Acknowledge）的状态标志，有效的状态标志位含义如表 9-1 所示。

表 9-1　有效的状态标志位

标志	值	作用
TCPCB_SACKED_ACKED	1	SACK 块已给出了 skb 数据缓冲区中的段回答信息
TCPCB_SACKED_RETRANS	2	数据段需要重传
TCPCB_LOST	4	数据段丢失
TCPCB_TAGBITS		结合前面 3 个标志来标识数据段
TCPCB_EVER_RETRANS	0X80	指明数据段以前是否重传过
TCPCB_RETRANS		指明数据段是一个重传过的数据段

TCPCB_RETRANS = TCPCB_SACKED_RETRANS | TCPCB_EVER_RETRANS

- ack_seq：与 TCP 协议头中的 ack 数据域相同。

9.2.3　TCP 套接字的数据结构

TCP 套接字的数据结构中包含了管理 TCP 协议各方面的信息，如发送和接收序列号、TCP 窗口尺寸、避免网络阻塞等。这些管理信息由 TCP 套接字的数据结构 struct tcp_sock 描述。struct tcp_sock 数据结构是 SOCK_STREAM 类的套接字数据结构 stuct sock 中的一部分，在分配套接字的同时也分配内存空间。

struct tcp_sock 是一个庞大的数据结构，其中包含了 TCP 层管理数据传送需要的所有信息。在 struct tcp_sock 数据结构的定义文件 include/linux/tcp.h 中有非常详尽的注释。在这里就不一一介绍其每个数据域的含义了，主要了解一些关键字段。

- inet_conn：INET 协议族是面向连接的套接字结构，定义在 include/net/inet_connection_sock 文件中，其中包含的 struct inet_connection_sock_af_ops *icsk_af_ops 数据结构，是套接字操作函数指针数据块，各协议实例在初始化时将函数指针初始化为自己的函数实例。struct inet_connection_sock inet_conn 数据结构必须为 tcp_sock 的第一个成员。
- tcp_header_len：传送数据段 TCP 协议头的长度。
- xmit_size_goal：输出数据段的目标。
- pred_flags：TCP 协议头预定向完成标志，在 9.3 节介绍 TCP 数据段接收时会介绍，用该标志确定数据包是否通过“Fast Path”接收。
- rcv_nxt：下一个输入数据段的序列号。
- snd_nxt：下一个发送数据段的序列号。

在 struct tcp_sock 数据结构中关键是理解嵌套其中的 struct ucopy 数据结构，它是实现数据段“Fast Path”接收的关键。struct ucopy 数据结构的数据域含义如下。

- prequeue：输入队列。其中包含等待由“Fast Path”处理的 Socket Buffer 链表。
- task：用户进程。接收 prequeue 队列中数据段的用户进程。
- iov：向量指针。指向用户地址空间中存放接收数据的数组。
- memory：在 prequeue 队列中所有 Socket Buffer 中数据长度的总和。
- Len： prequeue 队列上 Socket Buffer 缓冲区的个数。
- #ifdef CONFIG_NET_DMA

```
struct dma_chan          *dma_chan;
```

```
int                             wakeup;
struct dma_pinned_list   *pinned_list;
dma_cookie_t       dma_cookie;
```

- #endif

用于数据异步复制。当网络设备支持 Scatter/Gather I/O 功能时，可以利用 DMA 直接内存访问，将数据异步从网络设备硬件缓冲区中复制到应用程序地址空间的缓冲区。

```
struct tcp_sock {                                          include/linux/tcp.h
…
  struct {
        struct sk_buff_head    prequeue;
        struct task_struct     *task;
        struct iovec           *iov;
        int   memory;
        int   len;
#ifdef CONFIG_NET_DMA
        struct dma_chan        *dma_chan;
        int        wakeup;
        struct dma_pinned_list *pinned_list;
        dma_cookie_t           dma_cookie;
#endif
  } ucopy;
…
}
```

9.2.4　TCP 协议选项 Options

在 9.2.3 节描述的 struct tcp_sock 数据结构中，包含了 TCP 协议全部选项的变量。TCP 协议是可配置协议，在本章之后讨论 TCP 协议在 Linux 内核的实现过程时，我们可以看到如何按照 TCP 选项来设置、处理 TCP 数据段，这些选项影响着 TCP 协议的活动。TCP 选项是通过 setsockopt 系统来设置的，getsockopt 系统调用返回当前设置的 TCP 选项值。这些选项在 struct tcp_sock 数据结构中已经介绍了。本节我们描述关键 TCP 选项值的含义。

1. TCP_CORK/nonagle

如果设置了这个选项，TCP 不立即发送数据段，直到数据段中的数据达到 TCP 协议数据段的最大长度。它使应用程序可以在路由的 MTU 小于 TCP 数据段最大段大小（MSS：Maximum Segment Size）时停止传送，该选项存放在 struct tcp_sock 数据结构的 nonagle 数据域中，TCP_CORK 选项与 TCP_NODELAY 选项是互斥的。

2. TCP_DEFER_ACCEPT/defer_accept

应用程序调用者在数据还没到达套接字之前，可以处于休眠状态。但当数据到达套接字时则应用程序被唤醒。如果等待超时应用程序也会被唤醒。调用者设定一个时间值（秒）来描述应用程序等待数据到达超时时间。该选项保存在 struct tcp_sock 数据结构的 defer_accept 数据域中。

3. TCP_INFO

调用程序使用此选项可以提取大部分套接字的配置信息。在提取配置信息后，返回到 struct tcp_info 数据结构中。

```
struct tcp_info                          //include/linux/tcp.h
```

```
{
//第一个数据域 tcpi_state 包含的是当前 TCP 的连接状态，其后的数据域直到 tcpi_fackets，包含的是连接统计信息
  __u8  tcpi_state;                    /当前 TCP 的连接状态
  …
  __u32 tcpi_fackets;
//以下 4 个数据域是事件的时间戳信息
  __u32 tcpi_last_data_sent;
…
  __u32 tcpi_last_ack_recv;
//最后一部分是 TCP 协议的度量值，如 MTU、发送门限值、环行传送时间和阻塞窗口
  __u32 tcpi_pmtu;
…
  __u32 tcpi_total_retrans;
};
```

4. TCP_KEEPCNT

使用此选项，应用程序调用者可以设置在断开连接之前通过套接字发送多少个保持连接活动（keepalive）的探测数据段。该选项存放在 struct tcp_sock 数据结构中的 keepalive_probes 数据域中，如果要使这个选项有效，则还必须设置套接字层的 SO_KEEPALIVE 选项。

5. TCP_KEEPIDLE

在 TCP 开始传送连接是否保持活动的探测数据段之前，连接处于空闲状态的时间值（以秒为单位）。该选项存放在 struct tcp_sock 数据结构的 keepalive_time 数据域中，默认值为两个小时。该选项只有在套接字设置了 SO_KEEPALIVE 选项时才有效。

6. TCP_KEEPINTVL

设定在两次传送探测连接保持活动数据段之间要等待多少秒。该值存放在 struct tcp_sock 数据结构的 deepalive_intvl 数据域中，初始值为 75 秒。

7. TCP_LINGER2

这个选项指定处于 FIN_WAIT2 状态的孤立套接字还应保持存活多长时间。如果其值为 0，则关闭选项。Linux 使用常规方式处理 FIN_WAIT_2 和 TIME_WAIT 状态。如果值小于 0，则套接字立即从 FIN_WAIT_2 状态进入 CLOSED 状态，不经过 TIME_WAIT。该选项的值存放在 struct tcp_sock 数据结构的 linger2 数据域中，默认值由 sysctl 决定。

8. TCP_MAXSEG

在套接字连接建立之前，该选项指定最大数据段大小的值。送给 TCP 选项的 MSS 值就是由此选项决定的，但 MSS 的值不能超过接口的 MTU。TCP 连接两端的站点可以协商数据段大小的值。

9. TCP_NODELAY

当设置这个选项后，TCP 会立即向外发送数据段，而不会等待数据段中填入更多数据。该选项值存放在 struct tcp_sock 数据结构的 nonagle 数据域中。如果设置了 TCP_CORK 选项，这个选项就失效。

10. TCP_QUICKACK

当这个选项的值设置为 1 时，会关闭延迟回答，或为 0 时允许延迟回答。延迟回答是

Linux TCP 操作的一个常规模式。在延迟回答时，ACK 数据的发送会延迟到可以与一个等待发送到另一端的数据段合并时，才发送出去。如果这个选项的值设为 1，则将 struct tcp_sock 数据结构中的 ack 部分的 pingpong 数据域设为 0，就可以禁止延迟发送。TCP_QUICKACK 选项只会暂时影响 TCP 协议的操作行为。

11. TCP_SYNCNT

这个选项用于在尝试建立 TCP 连接时，如果连接没能建立起来，在重发多少次 SYN 后，才放弃建立连接请求。这个选项存放在 struct tcp_syn_retries 数据域中。

12. TCP_WINDOW_CLAMP

指定套接字窗口大小。窗口的最小值是 SOCK_MIN_RCVBUF 除以 2，等于 128 个字节。这个选项的值存放在 struct tcp_sock 数据结构的 window_clamp 数据域中。

9.2.5　应用层传送给传输层信息的数据结构

struct msghdr 数据结构中包含了来自应用层数据的信息，它将作为参数从套接字传送给 TCP 协议实例的传送处理函数，在传送处理函数中应用数据从套接字复制到内核地址空间的 Socket Buffer 中。

```
struct msghdr {                          //include/linux/socket.c
  void  *               msg_name;        /* 套接字的名字*/
  int                   msg_namelen;     /* 套接字名字的长度*/
  struct iovec *        msg_iov;         /* 数据块链表*/
  __kernel_size_t  msg_iovlen;           * 数据块数量  */
  void  *          msg_control;          /* 文件控制描述符指针 */
  __kernel_size_t  msg_controllen;       * 控制描述链表长度 */
  unsigned              msg_flags;       /*messages 传送时控制标志*/
};
struct iovec                             /include/linux/uio.h
{
  void __user *iov_base;                 /*存放用户地址空间数据缓冲区的基地址*/
  __kernel_size_t iov_len;               /*缓冲区的大小*/
};
```

- msg_name：套接字名。msg_nam 数据域中不是实际的套接字名，它是一个指针，指向 struct sockaddr_in 数据结构的变量，struct sockaddr_in 数据结构中包含了发送数据段到达的目标 IP 地址和端口号。
- msg_namelen：由 msg_name 指向的地址信息的长度。
- msg_iov：指向缓冲区数组的起始地址。这些缓冲区中包含了要发送或接收到的数据段。我们常把这些缓冲区称做分割/收集数组（Scatter/Gather array），但这些缓冲区不仅用于 DMA 操作，也用于不使用 DMA 方式传送数据的操作。
- msg_iovlen：是 msg_iov 缓冲区数组中缓冲区的个数。
- msg_control：用于支持应用程序的控制消息 API 功能，向套接字层下的协议传送控制信息。
- msg_controllen：为控制信息 msg_control 的长度。
- msg_flags：为套接字从应用层接收到控制的标志。所有标志值的含义如表 9-2 所示。

表 9-2　msg_flags 中标志位有效值

标　志	值	作　用
MSG_OOB	1	请求 out-of-bound 数据
MSG_PEEK	2	从接收队列中返回数据，但不把数据从队列中移走
MSG_DONTROUTE	4	不对该数据路由。常用于在 ping 例程中传送 ICMP 数据包
MSG_TRYHARD	4	不用于 TCP/IP 协议栈，与 MSG_DONTROUTE 同义
MSG_CTRUNC	8	用于 SOL_IP 内部控制信息
MSG_PROBE	0x10	用于发现 MTU 的数据段
MSG_TRUNC	0x20	Truncate 消息
MSG_DONTWAIT	0x40	调用应用程序是否使用不被阻塞的 I/O
MSG_EOR	0x80	信息结束
MSG_WAITALL	0x100	在返回数据前等待所有数据到达
MSG_FIN	0x200	TCP 结束数据段（FIN）
MSG_SYN	0x800	TCP 同步数据段（SYN）
MSG_CONFIRM	0x800	在传送数据包前确认路径连接有效
MSG_RST	0x1000	TCP 复位连接数据段（RST）
MSG_ERRQUEUE	0x2000	从错误队列读取数据段
MSG_NOSIGNAL	0x4000	当确定连接已断开时，不产生 SIGPIPE 信号
MSG_MORE	0x8000	设置该标志后，指明发送者将传送更多的信息

9.3　在 TCP 协议、套接字、IP 层之间的接口

在第 7 章中已经介绍了 IP 层与传输层的接口关系，这一章在讲解传输层的内部机制以前，我们首先介绍传输层 TCP 协议的实现函数是如何与套接字函数接口的。通过接口机制，应用程序可以直接与传输层各类型的套接字交互传送数据。

9.3.1　管理套接字与 TCP 接口的数据结构

管理 TCP 传输层与套接字层之间接口的关系，是由 struct proto 数据结构定义的，struct proto 数据结构中的大部分数据域都是函数指针，每一个函数指针指向传输层的一个功能函数，正如在 IP 层与传输层之间接收数据包的接口——struct inet_protocol 数据结构一样。

```
struct proto {                                                    //include/net/sock.h
   void            (*close)(struct sock *sk , long timeout);
   int             (*connect)(struct sock *sk , struct sockaddr *uaddr, int addr_len);
   int             (*disconnect)(struct sock *sk, int flags);
   struct sock *         (*accept) (struct sock *sk, int flags, int *err);
   int             (*ioctl)(struct sock *sk, int cmd , unsigned long arg);
   int             (*init)(struct sock *sk);
   void            (*destroy)(struct sock *sk);
   void            (*shutdown)(struct sock *sk, int how);
…
}
```

struct proto 数据结构的虚函数，即在 TCP 协议层中实现的实例函数如表 9-3 所示。

表 9-3　TCP 协议实例实现的 struct proto 数据结构中的函数

协议函数结构块数据域字段	TCP 协议实例实现的函数
close	tcp_close
connect	tcp_v4_connect
disconnect	tcp_disconnect
accept	inet_csk_accept
ioctl	tcp_ioctl
init	tcp_v4_init_sock
destroy	tcp_v4_destroy_sock
shutdown	tcp_shutdown
setsockopt	tcp_setsockopt
getsockopt	tcp_getsockopt
sendmsg	tcp_sendmsg
recvmsg	tcp_recvmsg
backlog_rcv	tcp_v4_do_rcv
hash	inet_hash
unhash	inet_unhash
get_port	inet_csk_get_port

从 tcp_close 到 tcp_shutdown 的函数，是管理 TCP 连接的函数实例，连接管理的实现我们将在 9.6 节中进行分解。

TCP 数据接收是由 tcp_recvmsg 和 tcp_v4_do_rcv 函数实现的，我们将在 9.4 节中介绍其实现过程。

TCP 数据发送是由 tcp_sendmsg 函数实现的，这个过程将在 9.5 中进行介绍。

最后 3 个函数是管理套接字端点地址信息的函数：hash 和 unhash 函数用于操作哈希链表，这些链表中存放的是打开的套接字的端点地址（端口号），是传输层协议端口号与套接字端口号的对应链表。hash 函数将套接字的引用 sk 放入链表。unhash 将套接字的引用从哈希链表中移出去。get_port 返回与套接字相关的端口号。通常端口号是从某个协议的端口哈希链表中获取的。

9.3.2 初始化套接字与传输层之间的接口

如果我们要在 Linux 内核 TCP/IP 协议栈的传输层实现一个新的协议，需要完成的步骤包括：

- 确定这个新的传输层协议应实现 struct proto 数据结构中的哪些函数。
- 声明一个 struct proto 数据结构类型的变量实例，并给 struct proto 数据结构类型的变量实例中的各函数指针赋值。
- 初始化接口，注册协议实例函数块变量到 TCP/IP 协议栈中。
- 编写各函数的实现代码。

对于本章描述的 TCP 协议实例，需要实现的函数如表 9-3 所示。第二个步骤是声明 struct proto 数据结构类型的变量实例，初始化变量如函数指针在 net/ipv4/tcp_ipv4.c 文件中完成。

```
struct proto tcp_prot = {                              net/ipv4/tcp_ipv4.c
   .name = "TCP",
   .owner= THIS_MODULE,
   .close= tcp_close,
```

```
    .connect= tcp_v4_connect,
    .disconnect= tcp_disconnect,
    .accept= inet_csk_accept,
    .ioctl= tcp_ioctl,
    .init = tcp_v4_init_sock,
    .destroy= tcp_v4_destroy_sock,
    .shutdown= tcp_shutdown,
    .setsockopt= tcp_setsockopt,
    .getsockopt= tcp_getsockopt,
    .recvmsg= tcp_recvmsg,
    .backlog_rcv= tcp_v4_do_rcv,
…
}
```

协议实例通过 int proto_register(struct proto *prot, int alloc_slab)函数注册到 TCP/IP 协议栈中。函数 void proto_unregister(struct proto *prot)将协议实例从系统中注销。以上两个函数都定义在 net/core/sock.c 文件中。

TCP 协议实例与套接字之间的接口函数在 inet_init 中注册并初始化。

```
static int __init inet_init(void)                          //net/ipv4/af_inet.c
{
…
   rc = proto_register ( &tcp_prot, 1);
   if (rc)
         goto out
…
```

9.3.3 TCP 与 IP 层之间的接收接口

在第 7 章已经介绍了 IP 层处理完接收数据包后，将数据包向传输层上传的接口。IP 层是通过 ip_local_deliver_finish 函数将数据包上传给传输层的。IP 层如何找到传输层正确地接收函数，涉及传输层与 IP 层之间的数据接收接口的组织和注册。在第 7 章已介绍了传输层与 IP 层之间的网络数据包接收接口数据结构组织和注册过程，这里主要对其做个总结，使读者形成套接字与 TCP 之间，TCP 与 IP 层之间接口关系的完整体系结构。

1. 描述 IP 层与传输层数据包接收接口的数据结构

```
struct net_protocol {
   include/net/protocol.h
   int              (*handler)(struct sk_buff *skb);
   void             (*err_handler)(struct sk_buff *skb, u32 info);
   int              (*gso_send_check)(struct sk_buff *skb);
   struct sk_buff         *(*gso_segment)(struct sk_buff *skb,
                               int features);
   struct sk_buff        **(*gro_receive)(struct sk_buff **head,
                               struct sk_buff *skb);
   int              (*gro_complete)(struct sk_buff *skb);
   unsigned int           no_policy:1,netns_ok:1;
};
```

struct net_protocol 数据结构字段的含义参见 7.5.3 节。在相同文件中系统定义了一个 struct net_protocol 类型的全局向量：

extern struct net_protocol *inet_protos[MAX_INET_PROTOS];用于存已注册的传输层协议实例从 IP 层接收数据包的处理函数块。全局向量的基地址由：extern struct net_protocol

*inet_protocol_base 指定。

2. 定义 TCP 协议实例的 struct net_protocol 变量实例并初始化数据域

Linux 内核的 TCP/IP 协议栈中在传输层实现的各协议实例都在 net/ipv4/af_inet.c 文件中定义了自己的 struct net_protocol 数据结构类型的变量实例，并对各数据域赋值。

定义在 net/ipv4/protocol.c 文件中的函数 int inet_add_protocol(struct net_protocol *prot, unsigned char protocol)，将协议的 struct net_protocol 变量实例加入到系统中，即放入 inet_protos 全局向量中。

函数 int inet_del_protocol(struct net_protocol *prot, unsigned char protocol)将协议从 inet_protos 全局向量中移走。

```
static struct net_protocol tcp_protocol = {                    net/ipv4/af_inet.c
   .handler =tcp_v4_rcv,
   .err_handler =tcp_v4_err,
   .gso_send_check = tcp_v4_gso_send_check,
   .gso_segment =tcp_tso_segment,
   .gro_receive =   tcp4_gro_receive,
   .gro_complete =tcp4_gro_complete,
   .no_policy =1,
   .netns_ok =1,
};
static struct net_protocol udp_protocol = {
   .handler =udp_rcv,
   .err_handler =   udp_err,
   .no_policy =1,
   .netns_ok =1,
};

…
```

3. 向系统注册协议实例

在内核启动期间，在执行 INET 协议族初始化函数 inet_init 时，调用 inet_add_protocol 函数将 TCP 协议实例注册到内核，即放入 inet_protos[MAX_INET_PROTOS]全局向量中。

```
static int __init inet_init(void)                                  net/ipv4/af_inet.c
{
…
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");
//向内核注册 TCP 接收 IP 数据包的接收处理函数数据块
   if (inet_add_protocol(&tcp_protocol, IPPROTO_TCP) < 0)
        printk(KERN_CRIT "inet_init: Cannot add TCP protocol\n");
…
}
```

9.3.4　TCP 与 IP 层之间的发送接口

1. 描述接口的数据结构

在 9.5 节初始化 TCP 套接字时，已经介绍了 TCP 套接字的初始化包含了初始化 inet_connection_sock_af_ops 类型指针, inet_connection_sock_af_ops 指针指向 TCP 套接字结

构的 ipv4_specific 数据域，它是 inet_connection_sock_af_ops 数据结构的一个实例，包含了一组 AF_INET 地址族中 TCP 协议实例的操作函数。其目的是实现一组 IPv4 和 IPv6 都可以共享 TCP 与网络层之间的接口。

接口由数据结构 struct inet_connection_sock_af_ops 描述。各函数的作用如表 9-4 所示。

```
struct inet_connection_sock_af_ops {                    include/net/inet_connection_sock.h
  int      (*queue_xmit)(struct sk_buff *skb, int ipfragok);
…
}
```

表 9-4　inet_connection_sock_af_ops 数据结构的 IPv4 实例

数据域	TCP 实例函数	描述
queue_xmit	ip_queue_xmit	IPv4 网络层传送函数
send_check	tcp_v4_send_check	计算 TCP 发送数据段校验和函数
rebuild_header	inet_sk_rebuild_header	创建 TCP 协议头
conn_request	tcp_v4_conn_request	处理连接请求数据段
cyn_recv_sock	tcp_v4_syn_recv_sock	从另一端点收到SYNACK回答后创建新的子套接字的函数
remember_stamp	tcp_v4_remember_stamp	用于保存从某个站点收到最后一个数据包的时间戳
net_header_len	sizeof(struct iphdr)	网络层协议头的大小，设置为 IPv4 协议头长度
setsockopt	ip_setsockopt	设置 IPv4 在网络层的套接字选项
getsockopt	ip_getsockopt	获取 IPv4 在网络层的套接字选项
addr2sockaddr	inet_csk_addr2sockaddr	为 IPv4 生成常规 sockaddr_in 类型地址
sockaddr_len	sizeof(struct sockaddr_in	IPv4 的 sockaddr_in 类型地址大小

2. 定义 TCP 协议的 struct inet_connection_sock_af_ops 变量实例，并赋初值

在 net/ipv4/tcp_ipv4.c 文件中系统定义了 TCP 的 struct inet_connection_sock_af_ops 数据结构类型的变量实例，并对其数据域赋初值。

```
struct inet_connection_sock_af_ops ipv4_specific = { // net/ipv4/tcp_ipv4.c
   .queue_xmit= ip_queue_xmit,
   .send_check= tcp_v4_send_check,
   .rebuild_header= inet_sk_rebuild_header,
   .conn_request= tcp_v4_conn_request,
   .syn_recv_sock= tcp_v4_syn_recv_sock,
   .remember_stamp= tcp_v4_remember_stamp,
…
}
```

3. 注册 TCP 协议实例

系统在 tcp_v4_init_sock 套接字初始化函数中，注册传输层各协议实例的 struct inet_connection_sock_af_ops 数据结构的变量实例。

```
static int tcp_v4_init_sock(struct sock *sk)                    //net/ipv4/tcp_ipv4.c
{
…
icsk->icsk_af_ops = &ipv4_specific;
…
}
```

套接字、TCP、IP 层之间的接口函数关系如图 9-4 所示。

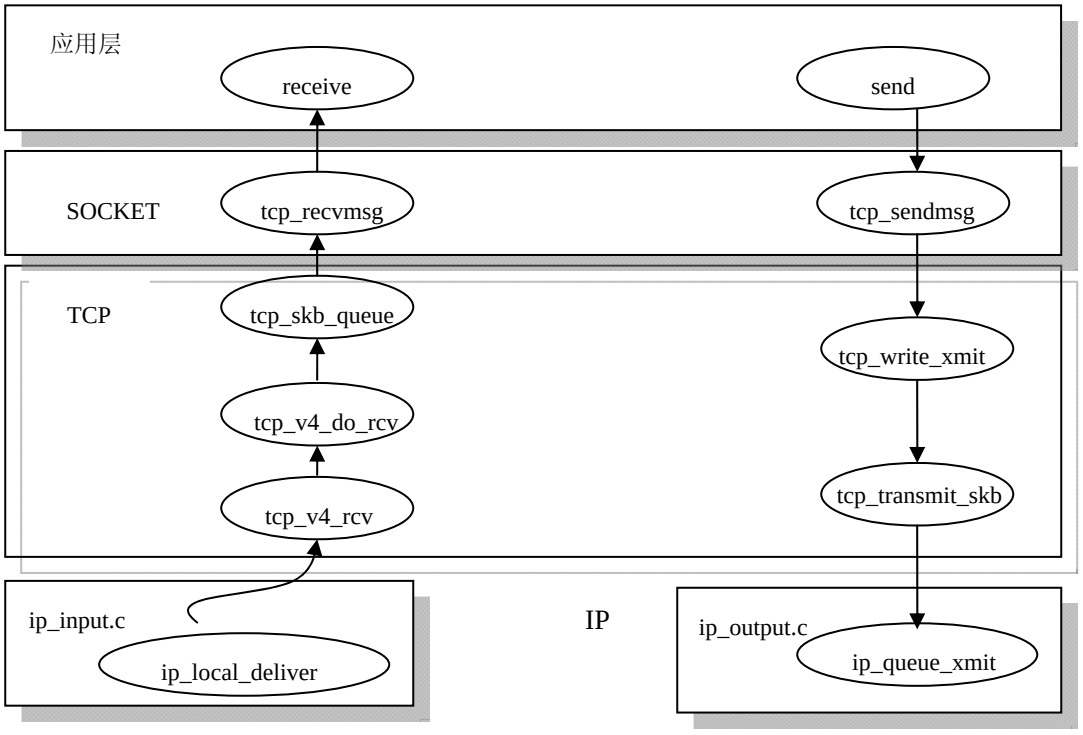


图 9-4　套节字、TCP、IP 层之间的接口函数关系

9.3.5　初始化 TCP 套接字

在介绍了各层之间的接口后，我们来看看如何完成 SOCK_STREAM 类套接字初始化。这个功能由 tcp_v4_init_sock 函数完成。该函数是在套接字数据结构创建之后被调用的，套接字数据结构的许多数据域都已初始化为 0。关于套接字的细节我们将在第 11 章进行介绍。这个函数初始化的主要是 TCP 套接字结构，即在前面讲解的 struct tcp_sock 数据结构。

```
static int tcp_v4_init_sock(struct sock *sk)                    //net/ipv4/tcp_ipv4.c
{

//获取套接字指针
  struct inet_connection_sock *icsk = inet_csk(sk);
  struct tcp_sock *tp = tcp_sk(sk);

//初始化 out_of_order_queue 队列；初始化 TCP 输出队列，该队列只由 TCP 使用；初始化传送超时时钟；初始化 prequeue
//输入队列，该队列用 “Fast Path” 接收
  skb_queue_head_init(&tp->out_of_order_queue);
  tcp_init_xmit_timers(sk);
  tcp_prequeue_init(tp);

//初始化数据包重传时间、rto、介质偏差时间 mdev，mdev 用于衡量 RTT（Round Trip //Time），设置为 3 秒
  icsk->icsk_rto = TCP_TIMEOUT_INIT;
```

```
    tp->mdev = TCP_TIMEOUT_INIT;

  //接下来就是初始化 TCP 选项的某些域：发送拥塞窗口的大小（send congestion window，cwnd）=2；send slow start threshold：snd_ssthresh 设置为最大值 32，有效的禁止 slow start 算法。发送拥塞窗口声明 snd_cwnd_clamp 最大设置为 16 位值 mss_cache 是 TCP 最小段大小为 536

  tp->snd_ssthresh = 0x7fffffff;
    tp->snd_cwnd_clamp = ~0;
    tp->mss_cache = 536;

  //按系统配置控制值初始化 TCP 选项结构的重排序域 reordering。套接字的状态 state 目前保持关闭。初始化 inet 连接套接字阻塞管理操作的函数为 tcp_init_congestion_ops

    tp->reordering = sysctl_tcp_reordering;
    icsk->icsk_ca_ops = &tcp_init_congestion_ops;
    sk->sk_state = TCP_CLOSE;

  //套接字 sk->sk_write_space 指针，指向套接字的回调函数，当套接字的写缓冲区有效时调用该函数，它指向 sk_stream_write_space。icsk_af_ops 设置成 TCP 协议特定的 AF_INET 函数

    sk->sk_write_space = sk_stream_write_space;
    sock_set_flag(sk, SOCK_USE_WRITE_QUEUE);
    icsk->icsk_af_ops = &ipv4_specific;
    icsk->icsk_sync_mss = tcp_sync_mss;

  //将套接字的发送缓冲区 sk_sndbuf 和接收缓冲区 sk_rcvbuf 大小初始化为系统配置控制值，可以用系统调用 setsockopt 来改变它们的大小
    sk->sk_sndbuf = sysctl_tcp_wmem[1];
    sk->sk_rcvbuf = sysctl_tcp_rmem[1];

  //tcp_sockets_allocated 是一个全局变量，保存了打开的 TCP 套接字的数量，这里对该值递增 1
  percpu_counter_inc(&tcp_sockets_allocated);
    return 0;
  }
```


9.4 TCP 协议实例接收过程的实现

TCP 协议是 Linux 网络体系结构中最复杂的一个部分。协议实例使用了大量的算法和外部功能扩展机制来实现协议。这一节将讲解这些机制是如何在 Linux 内核中实现的，以及它们是如何作用于 TCP 实现的。

首先介绍 TCP 协议中的数据接收和发送过程，然后介绍连接管理——TCP 是如何建立和断开连接的，最后介绍连接传送超时管理。

为了将输入数据包传送给传输层正确的协议处理函数，输入数据包使用的传输层协议在 IP 层处理时已设定。在传输层，各协议输入处理函数在协议初始化时注册到内核 TCP/IP 协议栈接口，IP 层通过调用 ip_local_deliver 函数在 IP 数据包协议头 iphdr->protocol 数据域中设定的值，在传输层与 IP 层之间管理协议处理接口的哈希链表 inet_protocol 中查询，找

到正确的传输层协议处理函数块，并上传数据包。对于 TCP 协议，它在 inet_protocol 结构中初始化的输入数据包处理函数是 tcp_v4_rcv，该函数定义在 net/ipv4/tcp_ipv4.c 文件中。

9.4.1　tcp_v4_rcv 函数的实现

tcp_v4_rcv 函数的输入参数就是 sk_buff 数据结构实例，它指向由 IP 层上传过来的数据包。正如在第 7 章中介绍 IP 协议实现时强调的一样，TCP 协议的接收函数是 Linux 内核的代码，一旦内核代码出现差错，则会引起整个系统的崩溃。所以在 tcp_v4_rcv 函数的起始部分对接收到的 skb 数据包做了一系列的正确性检查。tcp_v4_rcv 函数主要功能包括以下两个方面：

- 数据包合法性检查。
- 确定数据包是“快速路径”处理，还是“慢速路径”处理。

1. 数据包正确性检查

以下代码中一系列的 if 语句就是在对输入的数据包做各方面的正确性检查:

① skb->pkt_type 指明数据包的目标地址不是本主机地址时，扔掉数据包，这种情况可能发生于网络设备工作在混杂模式状态时。

② 用 pskb_may_pull 函数查看 TCP 协议头的正确性。

③ 保存在局部指针 th 变量中的 TCP 协议头数据结构地址，获取 TCP 协议头信息。查看协议头长度值 th->doff 的正确性，其长度至少应为不包含 TCP 协议选项时的协议头长度。th->doff 中保存的是 TCP 协议头的单元数。从 9.1.1 节中给出的 TCP 协议数据段格式可知，TCP 协议头的每个单元占 32 位，即 4 个字节。所以 sizeof(struct tcphdr)/4 为 TCP 协议头占用的单元数。

④ 后查看 TCP 数据段的校验和是否正确，如果校验和不正确则说明数据段已损坏。

⑤ 以上正确性检查过程中，任何一个环节不正确将扔掉数据包。

```
int tcp_v4_rcv (struct sk_buff * skb )                          net/ipv4/tcp_ipv4.c
{
  const struct iphdr * iph;
  struct tcphdr * th;
  struct sock * sk;
  int ret;
…
//数据包非本主机接收数据包
  if ( skb->pkt_type != PACKET_HOST)
       goto discard_it;
//是否能正确获取 TCP 协议头信息
  if (!pskb_may_pull (skb, sizeof(struct tcphdr)))
       goto discard_it;
//读取 TCP 协议头信息，查看协议头中 doff 数据域的正确性
  th = tcp_hdr( skb );
  if (th->doff < sizeof(struct tcphdr) / 4)
       goto bad_packet;
  if ( !pskb_may_pull(skb, th->doff * 4))
       goto discard_it;
```

```
//查看数据包校验和的正确性
  if ( !skb_csum_unnecessary (skb) && tcp_v4_checksum_init(skb))
       goto bad_packet;
```

2. 保存协议头信息

将 TCP 协议头的某些数据域保存在 skb 的控制缓冲区中，便于以后访问。skb 控制缓冲区是为各层协议存放私有数据缓冲区的。例如，在第 7 章介绍过的，IP 层协议实例用 skb 控制缓冲区来存放 IP 选项。TCP 协议用它来保存：

- TCP 数据段的起始序列号 seq。
- TCP 回答序列号 ack_seq。

输入数据段最后一个字节的位置 end_seq。end_seq 的值应为 seq+数据段长度+SYN 序列号或 FIN 序列号。

- when 和 sacked 数据域设为 0。when 用于计算数据包重传时间。sacked 用于选择回答。

skb 控制缓冲区 cb 用宏 TCP_SKB_CB(skb)来访问。

```
th = tcp_hdr (skb );
  iph = ip_hdr (skb );
  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 = iph->tos;
  TCP_SKB_CB(skb)->sacked = 0;
```


3. 查看数据段是否属于某个套接字

当以上过程完成后，调用__inet_lookup_skb 函数来查看接收到的数据包是否属于某个打开的套接字，查看的依据是接收数据包的网络接口、源端口号和目的端口号。如果数据段属于某个套接字，则将 sk 变量设置为指向打开的套接字的数据结构，接着继续处理数据段。

4. 数据段的处理

对数据段的处理起始于 process 标签。

① 旦获取数据段所属的套接字，就查看套接字的连接状态。如果套接字的连接状态为 TIME_WAIT，则必须以特殊的方式立即处理数据包。

```
process:
  if (sk->sk_state==TCP_TIME_WAIT)
       goto do_time_wait;
```

② IPsec 策略检查和网络过滤。如果内核配置使用了 IPsec 协议栈，则对数据包进行 IPsec 策略检查，此项检查由网络过滤子系统完成。如果未通过检查，则扔掉数据包。

```
  if (!xfrm4_policy_check (sk, XFRM_POLICY_IN, skb))
       goto discard_and_relse;
  nf_reset(skb);
  if (sk_filter(sk, skb))
       goto discard_and_relse;
```

③ 获取套接字锁并开始处理数据。在开始将数据向套接字传送前，首先要获取防止并发访问套接字的锁。如果获取套接字的保护锁不成功，说明有其他进程锁定了套接字，这时套接字不能接收其他数据段，则调用 sk_add_backlog 函数将输入段放入 backlog queue 队列中。

如果锁定套接字成功，则将数据段放入 prequeue 队列中。prequeue 队列存放在用户进程管理复制数据的 struct ucopy 数据结构中（我们知道 struct ucopy 数据结构是 struct tcp_sock 数据结构的一部分）。一旦数据段放入 prequeue 队列后，就由用户程序来处理数据，而不是由内核进程来处理数据。这样作为 TCP 提供最高的执行效率，能将核进程和用户进程之间执行现场的切换最小化。

如果 tcp_prequeue 返回 0，则说明当前没有用户进程在处理这个套接字，这时调用 tcp_v4_do_rcv 函数继续数据段的“Slow Path”接收处理。

当把数据段放入以上所说的某个队列等待处理后，函数可以释放对套接字的锁定，以便其他进程可以使用该套接字。sock_put 对套接字的引用即数据减 1，表明对套接字的处理已完成。

```
    skb->dev = NULL;
    bh_lock_sock_nested(sk);
    ret = 0;

  //锁定数据包所属套接字
    if (!sock_owned_by_user(sk)) {

  //如果配置了 DMA 传送，则以 DMA 方式在设备缓冲区与用户进程缓冲区之间复制数据
  #ifdef CONFIG_NET_DMA
          struct tcp_sock *tp = tcp_sk(sk);
          if (!tp->ucopy.dma_chan && tp->ucopy.pinned_list)
                tp->ucopy.dma_chan = dma_find_channel(DMA_MEMCPY);
          if (tp->ucopy.dma_chan)
                ret = tcp_v4_do_rcv(sk, skb);
          else
  #endif  {

  //将数据段放入 prequeue 队列，数据段按"Fast Path"路径处理，不成功调用 tcp_v4_do_rcv 将数据段按“Slow Path”
  //路径处理

                if (!tcp_prequeue(sk, skb))
                ret = tcp_v4_do_rcv(sk, skb);
          }
    } else
  //如果锁定套接字不成功，则将数据段放入 backlog queue 队列中
          sk_add_backlog(sk, skb);
    bh_unlock_sock(sk);
    sock_put(sk);                              //释放套接字
    return ret;
```

5. 没有打开的套接字的处理

如果程序执行到 no_tcp_socket 标签，则表明接收过程将在当前 TCP 套接字没有打开的

情况下结束，这时仍需要完成对数据包校验和的检验，看数据包是否为一个损坏的数据包。如果数据包不正确，则更新接收错误信息。如果数据包完好无损，则意味着程序试图发送一个数据段给一个未打开的套接字，函数需要对外发送一个复位请求来关闭连接，然后扔掉数据包，释放内存。

```
no_tcp_socket:
  if (!xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb))
        goto discard_it;

  if (skb->len < (th->doff << 2) || tcp_checksum_complete(skb)) {
bad_packet:
        TCP_INC_STATS_BH(net, TCP_MIB_INERRS);
  } else {
        tcp_v4_send_reset(NULL, skb);
  }
…
```

6. 套接字连接状态为 TIME_WAIT 时的处理

当套接字的连接状态为 TIME_WAIT 时，程序分支到 do_time_wait 标签处。套接字连接状态为 TIME_WAIT 时，需要一些特别的处理，细节部分我们将在 9.5 中进行介绍。在处理之前同样首先需要保证数据包是完好的数据包。然后调用 tcp_timewait_state_process 函数来查看收到的数据包是否为请求连接的同步数据段（SYN）、结束连接数据段（FIN）或是一个包含实际传送数据的段，再确定该做何处理。由函数 tcp_timewait_state_process 返回的状态值如表 9-5 所示。

```
do_time_wait:
  …

  //当套接字状态为 TIME_WAIT 时对数据段的处理，根据 tcp_timewait_state_process 函数返回的数据段类型做相应处理
  switch (tcp_timewait_state_process(inet_twsk(sk), skb, th)) {
  //如果接收到的数据段为同步请求 SYN 数据段，则打开套接字连接
  case TCP_TW_SYN: {
        struct sock *sk2 = inet_lookup_listener(dev_net(skb->dev),
                                    &tcp_hashinfo,
                                    iph->daddr, th->dest,
                                    inet_iif(skb));
        if (sk2) {
              inet_twsk_deschedule(inet_twsk(sk), &tcp_death_row);
              inet_twsk_put(inet_twsk(sk));
              sk = sk2;
              goto process;
        }
  }

  //收到最后一个数据段为 ACK 数据段
  case TCP_TW_ACK:
        tcp_v4_timewait_ack(sk, skb);
        break;
```

```
case TCP_TW_RST:
     goto no_tcp_socket;
case TCP_TW_SUCCESS:;
}
goto discard_it;
```

表 9-5　TCP 套接字连接状态为 TIME_WAIT 时的值

符号	值	描述
TCP_TW_SUCCESS	0	延迟数据段或原 ACK 的复制，扔掉数据包
TCP_TW_RST	1	收到结束连接 FIN，传送复位给另一端
TCP_TW_ACK	2	收到最后一个回答 ACK，传送回答 AKC 给另一端，并关闭连接
TCP_TW_SYN	3	收到建立连接请求 SYN，打开连接

按照 TCP 协议规范，当 TCP 收到 SYN 建立连接请求时，可以在套接字的 TIME_WAIT 连接状态下“唤醒”连接。这时调用 tcp_v4_lookup_listener 函数来创建侦听连接进程并建立新连接。

tcp_v4_rcv 此函数处理流程图如图 9-5 所示。此函数内部各例程之间的调用关系如图 9-6 所示。

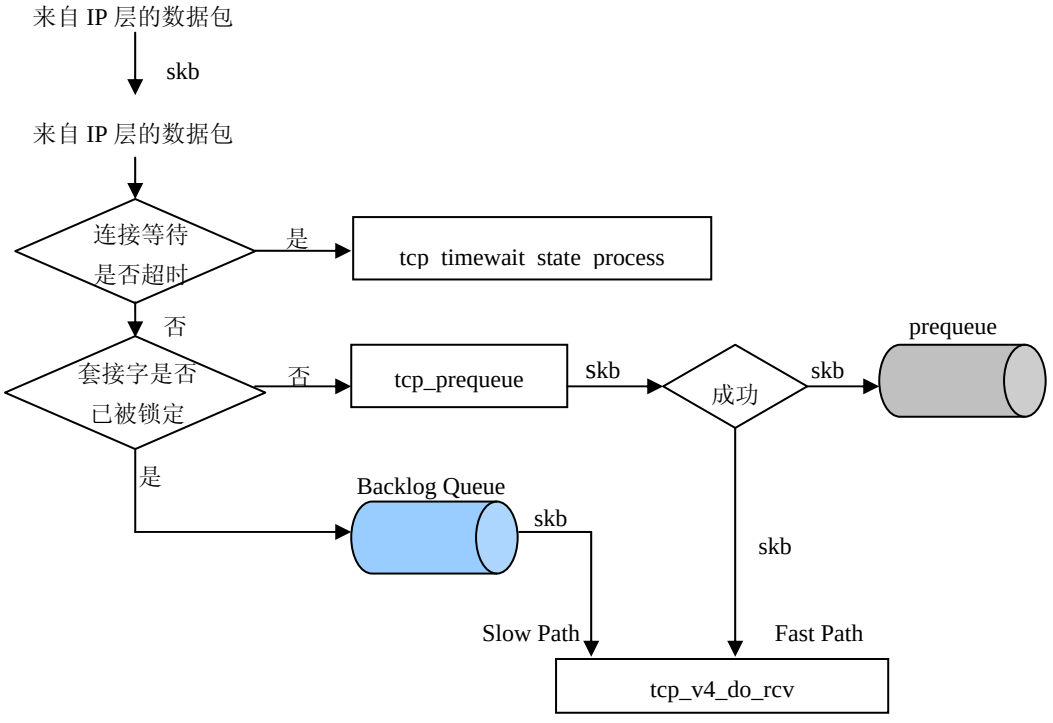


图 9-5　tcp_v4_rcv 函数的处理流程图

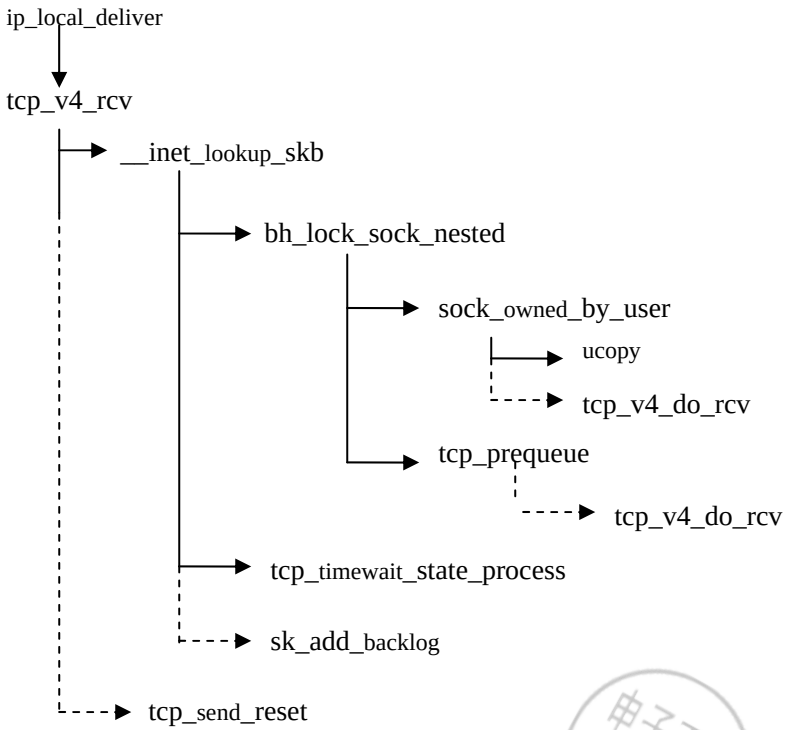


图 9-6　tcp_v4_rcv 函数内部各流程之间的调用关系

9.4.2　Fast Path 和 prequeue 队列的处理

Linux TCP/IP 协议栈中，在 TCP 层有两条路径处理输入数据包：“Fast Path” 和“Slow Path”。“Fast Path”是内核优化 TCP 处理输入数据包的方式。当 TCP 协议实例收到一个数据包后，它首先通过协议头来预定向数据包的去处：“Fast Path”或“Slow Path”。如果将数据包放入“Fast Path”处理，则需要满足以下条件：

- 收到的数据段中包含的是数据，而不是 ACK。
- 数据段是顺序传送数据中的一个完整数据段，接收顺序正确。

当收到数据段时，套接字连接状态为 ESTABLISHED。

满足以上条件的数据段会放入 prequeue 队列中，这时用户进程被唤醒，在 prequeue 队列中的数据段就由用户层的进程来处理，这个过程省略很多“Slow Path”处理中的步骤，从而加大了数据吞吐量。

之所以可以建立“Fast Path”处理过程，是因为 TCP 协议头预定向算法可以认为在套接字连接期间到达的数据包至少有一半是包含数据段的数据包，而不是 ACK 段数据包。通过使用 TCP 协议头预定向，TCP 协议低层接收函数事先确定哪些数据包符合以上条件，这些数据包就立即放入 prequeue 队列中。

接下来分析“Fast Path”在内核中的实现，关于“Slow Path”在内核中的实现，将在 9.43 节进行介绍。

一旦 TCP 接收到的数据包满足在“Fast Path”路径处理的条件，数据包就会放入 prequeue 队列中。prequeue 队列定义在 struct ucopy 数据结构中。struct ucopy 数据结构嵌套在另一个更大的 TCP 套接字 struct tcp_sock 数据结构中。struct ucopy 数据结构的定义和数据字段含义在 9.2.3 节中已说明，这里就不再赘述。

1. “Fast Path”的初始化

由函数 tcp_prequeue_init 完成“Fast Path”的初始化，它初始化 struct ucopy 数据结构中的成员，以及 prequeue 队列中的 Socket Buffer 链表。初始化过程是用户进程在 AF_INET 地址族上打开任意 SOCK_STREAM 类型的套接字时调用的。struct ucopy 的初始化是套接字状态信息初始化（由 tcp_v4_init_sock 函数完成）的一部分。

```
static inline void tcp_prequeue_init(struct tcp_sock *tp)  include/net/tcp.h
{
  tp->ucopy.task = NULL;                                  //处理数据复制的用户进程
  tp->ucopy.len = 0;                                      //prequeue 队列上缓冲区个数
  tp->ucopy.memory = 0;                                   //数据量总长度
  skb_queue_head_init ( & tp- > ucopy.prequeue );         //初始化 prequeue 队列
…
}
```

2. 将数据包放入 prequeue 队列的处理函数

由 tcp_prequeue 函数完成将 Socket Buffer 放入 prequeue 队列的功能。tcp_prequeue 函数将 Socket Buffer 放入 prequeue 队列的条件是，有一个用户进程在打开的套接字上等待接收数据。这个条件通过 struct ucopy 数据结构的 task 数据域非空来验证。

```
static inline int tcp_prequeue(struct sock *sk, struct sk_buff *skb)  include/net/tcp.h
{
struct tcp_sock *tp = tcp_sk(sk);

//如果用户空间有进程在等待接收数据段，ucopy.task 为非空，则将套接字缓冲区 skb 放到 prequeue 队列
  if (!sysctl_tcp_low_latency && tp->ucopy.task) {
        __skb_queue_tail(&tp->ucopy.prequeue, skb);
        tp->ucopy.memory += skb->truesize;
```

① 检验当前是否有用户进程在套接字上等待接收数据。如果有，将 Socket Buffer 放入 prequeue 队列尾。另外用户可以通过 sysctl 控制这个过程，我们也在这里检测。

② 将 prequeue 队列中的缓冲区移入套接字缓冲区。如果 prequeue 队列中缓冲区的总长度已大于套接字接收缓冲区的长度，则调用 sk_backlog_rcv 函数将 prequeue 队列中的 Socket Buffer 放入 backlog queue 队列，由“Slow Path”路径函数处理。直至到 prequeue 队列中的所有缓冲区都处理完成。此后，通过对 struct ucopy 数据结构的 memory 数据域清 0 来清空 prequeue 队列。

```
//如果 prequeue 队列中缓冲区的总长度大于套接字接收缓冲区的长度，将缓冲区放入“ Slow Path”路径队列处理
        if (tp->ucopy.memory > sk->sk_rcvbuf) {
              struct sk_buff *skb1;
              BUG_ON(sock_owned_by_user(sk));
//在 prequeue 队列不为空时，从 prequeue 队列上取下 skb，交给 sk_backlog_rcv 函数处理
              while ((skb1 = __skb_dequeue(&tp->ucopy.prequeue)) != NULL) {
                    sk_backlog_rcv( sk, skb1);
                    NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPPREQUEUEDROPPED);
              }
```

```
//清空 prequeue 队列
                tp->ucopy.memory = 0;
        }
```

③ 只要 prequeue 队列上有一个 Socket Buffer，就唤醒用户进程。

```
  //prequeue 队列上至少有一个缓冲区，唤醒用户进程接收数据
else if (skb_queue_len(&tp->ucopy.prequeue) == 1) {
                wake_up_interruptible(sk->sk_sleep);
                if (!inet_csk_ack_scheduled(sk))
                      inet_csk_reset_xmit_timer(sk, ICSK_TIME_DACK,
                                                (3 * TCP_RTO_MIN) / 4,
                                               TCP_RTO_MAX);
        }
        return 1;
   }
   return 0;
}
```

④ 函数返回值。当有数据包放入 prequeue 队列时，函数返回 1。当无数据包放入队列时，函数返回 0。返回 0 值表明 prequeue 队列已满，数据包只能放入 backlog 接收队列中。

9.4.3 处理 TCP 的 Blocklog 队列

相对于“Fast Path”处理过程，TCP 的 Backlog 队列处理，即“Slow Path”是常规输入数据包的处理方式。它根据套接字当前的状态来确定输入数据包的去向。在“Slow Path”处理方式中，当套接字的接收缓冲区已满时，就不能再接收新的 Socket Buffer，或当套接字忙（被别的进程锁定）时，TCP 需将接收到的数据包放入套接字阻塞时的等待队列 backlog queue 中。将输入数据包放入 backlog queue 队列的前提条件是：

- 输入数据包中包含的是数据段，不是 ACK 段。
- 数据段完好无损。
- 套接字缓冲区已满或套接字被别的用户进程占用。

这个过程需要的处理步骤较多，一旦数据包缓冲区放入队列，套接字就被唤醒，进程调度器（scheduler）调度用户进程，开始从 Backlog queue 队列中读取数据包缓冲区。“Slow Path”的处理过程由 tcp_v4_do_rcv 函数完成。以下我们将分析 tcp_v4_do_rcv 函数的处理流程。

1. 套接字正处于连接状态：ESTABLISHED

根据前面所述，首先需查看当前套接字所处的状态，如果套接字当前正处于连接状态，数据包可能可以通过“Fast Path”路径处理。调用 tcp_rcv_established 函数完成套接字为连接状态时对输入数据包的处理。此函数使用 TCP 协议头预定向来确定数据包是否可用“Fast Path”方式处理。

如果 tcp_rcv_established 函数返回 0，则对数据包的处理成功，如果返回 1 时，则必须向数据包发送方发出复位连接的请求。

```
int tcp_v4_do_rcv (struct sock *sk, struct sk_buff *skb)        //net/ipv4/tcp_ipv4.c
{
  struct sock *rsk;
…
```

```
 //如果套接字当前状态为连接状态，缓冲区可能可以通过“Fast Path”路径处理，由 tcp_rcv_established 函数
//完成此功能

   if (sk->sk_state == TCP_ESTABLISHED) { /* Fast path *
        TCP_CHECK_TIMER(sk);
        if (tcp_rcv_established ( sk, skb, tcp_hdr(skb), skb->len)) {
             rsk = sk;
             goto reset;
        }
        TCP_CHECK_TIMER(sk);
        return 0;
   }
```

2. 检查数据包的正确性

通过查看 TCP 协议头的长度和数据包校验和是否正确，以确认接收的数据包为完好的数据包。

```
   if (skb->len < tcp_hdrlen(skb) || tcp_checksum_complete(skb))
        goto csum_err;
```

（1）套接字状态切换与数据包处理

当接收数据包到达时，它立即处理数据包的传送，但除套接字正处于连接状态外。如果套接字处于其他状态，则需按套接字的状态进行切换。

（2）侦听状态：LISTEN

这时查看输入数据包 skb 是否为一个请求连接的 SYN 数据包，如果收到的是一个有效的 SYN 数据段，则将套接字状态转变为接收状态。这个过程由函数 tcp_v4_hnd_req 完成，它使连接有效并返回建立连接的套接字数据结构地址 struct sock *snk 或 NULL。返回的套接字就处于 ESTABLISHED 状态。

```
        if (sk->sk_state == TCP_LISTEN) {
        struct sock *nsk = tcp_v4_hnd_req(sk, skb);
        if (!nsk)
             goto discard;
```

（3）转换至连接状态：ESTABLISHED

原套接字 sk 继续侦听，调用 tcp_child_process 函数在子套接字 nsk 上处理接收。如果接收处理成功，则函数 tcp_child_process 返回 0，此后新的子套接字就处于连接状态，可以传送和接收数据。

```
        if (nsk != sk) {
             if (tcp_child_process(sk, nsk, skb)) {
                  rsk = nsk;
                  goto reset;
             }
             return 0;
```

3. 当前套接字不处于侦听状态

如果套接字当前不处于侦听状态，则调用 tcp_rcv_state_process 函数处理套接字的常规状态切换。如果返回 0 则说明处理成功，如果返回非 0 值则说明我们必须复位套接字连接。

```
   TCP_CHECK_TIMER(sk);
   if (tcp_rcv_state_process(sk, skb, tcp_hdr(skb), skb->len)) {
```

```
        rsk = sk;
        goto reset;
    }
    TCP_CHECK_TIMER(sk);
    return 0;
```

4. 错误状态处理

函数的 3 个退出标签：reset、discard 和 csumm_err 分别处理复位套接字连接、扔掉数据包和更新接收错误统计信息。

套接字的常规状态及它们之间的切换条件如图 9-7 所示。

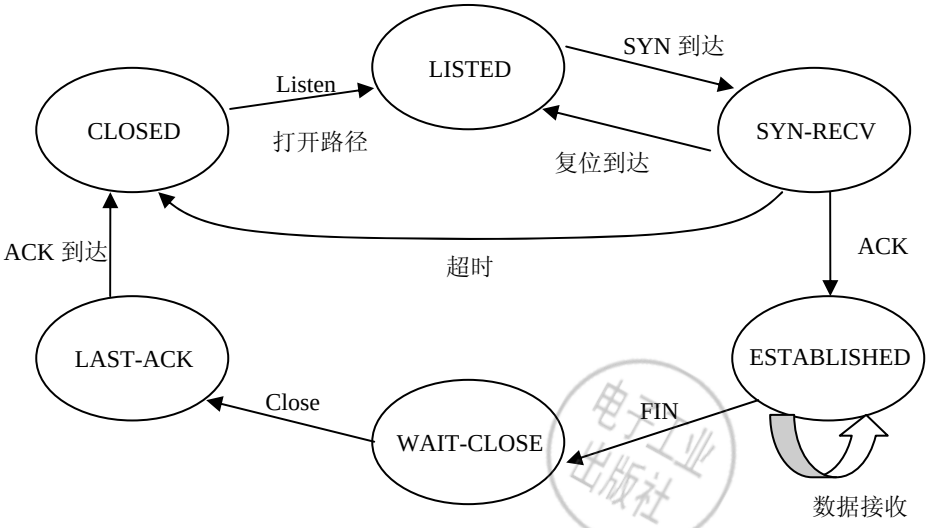


图 9-7　TCP 套接字常规状态切换

9.4.4　套接字层的接收函数

当用户进程通过获得信号得知在打开的套接字上有数据等待用户进程来接收时，用户进程调用 receive 或 read 系统调用来读取套接字缓冲区中的数据。当这些系统调用将读取的数据传送到套接字层时，转而会调用 tcp_recvmsg 函数来执行具体的传送操作。tcp_recvmsg 函数从打开的套接字上将数据复制到用户缓冲区。

```
int tcp_recvmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
        size_t len, int nonblock, int flags, int *addr_len)          //net/ipv4/tcp.c
{
   struct tcp_sock *tp = tcp_sk(sk);            //从套接字数据结构获取 TCP 套接字结构
   …
   lock_sock(sk);
   TCP_CHECK_TIMER(sk);
   err = -ENOTCONN;
//如果当前套接字处于侦听状态，说明还没数据等待接收，跳出
   if (sk->sk_state == TCP_LISTEN)
        goto out;
timeo = sock_rcvtimeo(sk, nonblock);
//如果设置了 MSG_OOB 标志处理紧急数据（即输入段设置了 URG 标志），seq 初始化为下一个准备读的字节，copied_seq 中
//包含的是已处理的最后一个字节
```

```
  if (flags & MSG_OOB)
        goto recv_urg;
  seq = &tp->copied_seq;
  if (flags & MSG_PEEK) {
        peek_seq = tp->copied_seq;
        seq = &peek_seq;
  }

//将本次读的字节数 target，设置为 sk->rcvlowat 和 len 中小的值。MSG_WAITALL 标志指明本次调用是否会阻塞
  target = sock_rcvlowat(sk, flags & MSG_WAITALL, len);
//如果网络设备支持 Scatter/Gather I/O 功能，内核配置了 DMA，则可以通过直接内存访问复制数据到用户地空间
#ifdef CONFIG_NET_DMA
  tp->ucopy.dma_chan = NULL;
  preempt_disable();
  skb = skb_peek_tail(&sk->sk_receive_queue);
  {
        int available = 0;
        if (skb)
              available = TCP_SKB_CB(skb)->seq + skb->len - (*seq);
        if ((available < target) &&
           (len > sysctl_tcp_dma_copybreak) && !(flags & MSG_PEEK) &&
           !sysctl_tcp_low_latency &&
           dma_find_channel(DMA_MEMCPY)) {
              preempt_enable_no_resched();
              tp->ucopy.pinned_list =
                          dma_pin_iovec_pages(msg->msg_iov, len);
        } else {
              preempt_enable_no_resched();
        }
  }
#endif
//Do 循环是 tcp_recvmsg 的主循环，在该循环中，我们将继续复制字节到用户地址空间，直至达到 target，或对输入数据段
//检测到了其他异常条件，退出循环
  do {
        u32 offset;
//如果复制了一些数据后，遇到紧急数据，则停止处理过程
        if (tp->urg_data && tp->urg_seq == *seq) {
              if (copied)
                    break;

//这时检测该套接字上是否有信号等待处理，以确保能正确处理 SIGUSR 信号。接下来查看套接字是否超时，如果超时则返回错
//误代码
              if (signal_pending(current)) {
                    copied = timeo ? sock_intr_errno(timeo) : -EAGAIN;
                    break;
              }
        }
        //获取接收队列上的第一个缓冲区指针，在 do{}while 循环内遍历接收队列直到发现第一个数据段。发现第一个数据
        //段才知道有多少个字节需要复制，跳到 found_ok-skb 的同时计算应从 skb 复制的字节数
        skb = skb_peek(&sk->sk_receive_queue);
        //在循环内一直查看直到发现有效数据段，在此过程中查看 FIN 和 SYN。如果发现 SYN，则调整要复制的字节数，从
        //偏移量中减 1。如果是 FIN，跳出循环
        do {
              if (!skb)
                    break;
```

```
                    //现在查看当前处理字节不在最近处理接收数据段的第一个字节前，这样可判别当前处理是否超出当前
//队列数据包同步范围。这实际上是一个冗余检查，因为我们锁定了套节字，而且可以了解套接字有多个队列防止数据丢失
                    if (before(*seq, TCP_SKB_CB(skb)->seq)) {
                    printk(KERN_INFO "recvmsg bug: copied %X "
                          "seq %X\n", *seq, TCP_SKB_CB(skb)->seq);
                    break;
               }
               offset = *seq - TCP_SKB_CB(skb)->seq;
               if (tcp_hdr(skb)->syn)
                    offset--;
               if (offset < skb->len)
                    goto found_ok_skb;
               if (tcp_hdr(skb)->fin)
                    goto found_fin_ok;
          WARN_ON(!(flags & MSG_PEEK));
               skb = skb->next;
          } while (skb != (struct sk_buff *)&sk->sk_receive_queue);
          //程序运行到此处说明套接字接收队列中已无数据，如果 backlog 队列中有数据包，则处理 backlog 队列中的数据
          if (copied >= target && !sk->sk_backlog.tail)
               break;
          //这里必须做一些相应的检查看是否应停止处理数据包，看套接字是否已关闭或从远端收到断开连接要求（在接收数
          //据包中有 RST 标志）
          if (copied) {
               if (sk->sk_err ||
                  sk->sk_state == TCP_CLOSE ||
                  (sk->sk_shutdown & RCV_SHUTDOWN) ||
                  !timeo ||
                  signal_pending(current))
                    break;
          } else {
               if (sock_flag(sk, SOCK_DONE))
                    break;

               if (sk->sk_err) {
                    copied = sock_error(sk);
                    break;
               }
          if (sk->sk_shutdown & RCV_SHUTDOWN)
                    break;
               if (sk->sk_state == TCP_CLOSE) {
               //当用户关闭了套节字时，会设置 SOCK_DONE 标志，所以当连接状态是 CLOSED 时其值为非零。如果在 CLOSED
               //套接字上，则 SOCK_DONE 为非零，说明应用程序试图从一个从未建立起连接的套接字上读数据，这是一个
               //错误条件
                    if (!sock_flag(sk, SOCK_DONE)) {
                         copied = -ENOTCONN;
                         break;
                    }
                    break;
               }
               if (!timeo) {
                    copied = -EAGAIN;
                    break;
               }
               if (signal_pending(current)) {
                    copied = sock_intr_errno(timeo);
                    break;
```

```
                }
        }
        tcp_cleanup_rbuf(sk, copied);

        //这时在接收队列中已无数据段需要处理。我们将处理在 prequeue 队列上的数据包。在此之前，协议头预定向指明
        //在连接为 ESTABLISHED 状态下可能收到有数据段。Prequeue 队列是由用户进程现场处理而不是 bottom half。
        //将 Ucopy.task 设置为当前进程，current 强制复制 prequeue 队列中的数据段到用户地址空间
        if (!sysctl_tcp_low_latency && tp->ucopy.task == user_recv) {
        //这里安装一个新的读进程
                if (!user_recv && !(flags & (MSG_TRUNC | MSG_PEEK))) {
                        user_recv = current;
                        tp->ucopy.task = user_recv;
                        tp->ucopy.iov = msg->msg_iov;
                }
                tp->ucopy.len = len;
                WARN_ON(tp->copied_seq != tp->rcv_nxt &&
                        !(flags & (MSG_PEEK | MSG_TRUNC)));

               //如果 prequeue 队列为非空，则在释放套接字前必须处理这些数据包。如果这个处理没有完成，则数据段的
              //顺序将被破坏。在数据包接收端的处理顺序可以假设成 4 个队列：在 flight 中的数据包、backlog、prequeue
              //队列和常规接收队列。每个队列只有在其前面的队列数据包处理完后才能处理，接收队列现在为空，但 prequeue
               //队列中在循环结束套接字释放前可能又加入了数据包，prequeue 队列的数据包在 do_prequeue 标签处处理

                if (!skb_queue_empty(&tp->ucopy.prequeue))
                        goto do_prequeue;
        }
        if (copied >= target) {
        //现在，处理 backlog 队列，看是否可能从该队列上直接复制数据。这时我们已经处理了 prequeue 队列。
        //Release_sock 在唤醒等待在套接字上的用户进程前，先遍历 backlog 队列上的所有数据包 sk->backlog

                release_sock(sk);
                lock_sock(sk);
        }
        } else
                //如果已经没有数据要处理，则等待新的数据到来。sk_wait_data 把套接字放入等待状态
                sk_wait_data(sk, &timeo);
#ifdef CONFIG_NET_DMA
        tp->ucopy.wakeup = 0;
#endif
        if (user_recv) {
                int chunk;

                //计算在前面的步骤是否从 backlog 队列中直接复制了任何数据，也将进程调度器返回其正常状态

                if ((chunk = len - tp->ucopy.len) != 0) {
                        NET_ADD_STATS_USER(sock_net(sk),LINUX_MIB_TCPDIRECTCOPYFROMBACKLOG, hunk);
                        len -= chunk;
                        copied += chunk;
                }
                if (tp->rcv_nxt == tp->copied_seq &&
                   !skb_queue_empty(&tp->ucopy.prequeue)) {
                   do_prequeue:
                      //处理 prequeue 队列处。由 tcp_prequeue_process 函数完成这项处理。调用此函数后，调整
                      //从 prequeue 队列中已复制的数据量
                      tcp_prequeue_process(sk);
                      if ((chunk = len - tp->ucopy.len) != 0) {
```

```
                        NET_ADD_STATS_USER(sock_net(sk), LINUX_MIB_TCPDIRECTCOPYFROMPREQUEUE,
                        chunk);
                        len -= chunk;
                        copied += chunk;
                  }
            }
      }
      if ((flags & MSG_PEEK) &&
         (peek_seq - copied - urg_hole != tp->copied_seq)) {
            if (net_ratelimit())
                  printk(KERN_DEBUG "TCP(%s:%d): Application bug, race in MSG_PEEK.\n",
                  current->comm, task_pid_nr(current));
            peek_seq = tp->copied_seq;
      }
      continue;
//这里是在前面的循环中，我们发现在接收队列中有数据段时，就跳到此处。从 skb->len 数据域计算有多少数据要复制，以及
//它们的偏移量
found_ok_skb:
      used = skb->len - offset;
      if (len < used)
            used = len;
      //这时首先必须查看紧急数据，除非设置了套节字选项 SO_OOBINLINE，这时我们不处理紧急数据，因为它已单独处理
      if (tp->urg_data) {
            u32 urg_offset = tp->urg_seq - *seq;
            if (urg_offset < used) {
                  if (!urg_offset) {
                        if (!sock_flag(sk, SOCK_URGINLINE)) {
                              ++*seq;
                              urg_hole++;
                              offset++;
                              used--;
                              if (!used)
                                    goto skip_copy;
                        }
                  } else
                        used = urg_offset;
            }
      }
      //复制数据到用户地址空间，如果在复制过程中有错，则返回 EFAULT
      if (!(flags & MSG_TRUNC)) {
#ifdef CONFIG_NET_DMA
            if (!tp->ucopy.dma_chan && tp->ucopy.pinned_list)
                  tp->ucopy.dma_chan = dma_find_channel(DMA_MEMCPY);
            if (tp->ucopy.dma_chan) {
                  tp->ucopy.dma_cookie = dma_skb_copy_datagram_iovec(
                        tp->ucopy.dma_chan, skb, offset,
                        msg->msg_iov, used,
                        tp->ucopy.pinned_list);
                  if (tp->ucopy.dma_cookie < 0) {
                        printk(KERN_ALERT "dma_cookie < 0\n");
                        if (!copied)
                              copied = -EFAULT;
                        break;
}
            }
                  if ((offset + used) == skb->len)
```

```
                        copied_early = 1;
            } else
#endif
            {
                  err = skb_copy_datagram_iovec(skb, offset,
                              msg->msg_iov, used);
                  if (err) {
                    if (!copied)
                              copied = -EFAULT;
                        break;
                  }
            }
        }
        *seq += used;
        copied += used;
        len -= used;
        tcp_rcv_space_adjust(sk);
skip_copy:
        if (tp->urg_data && after(tp->copied_seq, tp->urg_seq)) {
              tp->urg_data = 0;
              //现在处理完了紧急数据，转到Fast Path上，如果输入处理遇到有紧急数据的段，则Fast Path会关闭
              tcp_fast_path_check(sk);
        }
        if (used + offset < skb->len)
              continue;
        if (tcp_hdr(skb)->fin)
              goto found_fin_ok;
        if (!(flags & MSG_PEEK)) {
              sk_eat_skb(sk, skb, copied_early);
              copied_early = 0;
        }
        continue;

//如果在接收队列中的数据包有FIN标志，跳入此处。按照RFC793规范，必须在序列号中计算FIN的一个字节，并重新计算
//TCP窗口
found_fin_ok:
        ++*seq;
        if (!(flags & MSG_PEEK)) {
              sk_eat_skb(sk, skb, copied_early);
              copied_early = 0;
        }
        break;
  } while (len > 0);

//处于while循环体外处理套接字，直到满足应用程序调用要求的数据总量len。如果现在跳出了循环，但在prequeue队列中
//留下了数据，则现在必须进行处理
  if (user_recv) {
        if (!skb_queue_empty(&tp->ucopy.prequeue)) {
              int chunk;
              tp->ucopy.len = copied > 0 ? len : 0;
              tcp_prequeue_process(sk);
              if (copied > 0 && (chunk = len - tp->ucopy.len) != 0) {
              NET_ADD_STATS_USER(sock_net(sk),LINUX_MIB_TCPDIRECTCOPYFROMPREQUEUE, chunk);
                    len -= chunk;
                    copied += chunk;
              }
```

```
        }
        tp->ucopy.task = NULL;
        tp->ucopy.len = 0;
    }
#ifdef CONFIG_NET_DMA
    if (tp->ucopy.dma_chan) {
        dma_cookie_t done, used;
        dma_async_memcpy_issue_pending(tp->ucopy.dma_chan);
        while (dma_async_memcpy_complete(tp->ucopy.dma_chan,
                                tp->ucopy.dma_cookie, &done,
                                &used) == DMA_IN_PROGRESS) {
            while ((skb = skb_peek(&sk->sk_async_wait_queue)) &&
                (dma_async_is_complete(skb->dma_cookie, done,
                                used) == DMA_SUCCESS)) {
                __skb_dequeue(&sk->sk_async_wait_queue);
                kfree_skb(skb);
            }
        }
    //Cleanup_rbuf 清除 TCP 接收缓冲区，如果需要它也会发送 ACK
    tcp_cleanup_rbuf(sk, copied);
    TCP_CHECK_TIMER(sk);
    release_sock(sk);
    return copied;
    //处理完成，释放套接字，跳出
    out:
    TCP_CHECK_TIMER(sk);
    release_sock(sk);
    return err;
    //在处理数据段时遇到紧急数据，跳到此片。Tcp_recv_urg 复制紧急数据到用户地址空间
    recv_urg:
    err = tcp_recv_urg(sk, timeo, msg, len, flags, addr_len);
    goto out;
}
```


9.5 Linux 内核中 TCP 发送功能的实现

TCP 协议实现的传送功能是指将从应用层通过打开的套接字写入的数据移入内核，通过 TCP/IP 协议栈，最终通过网络设备发送到远端接收主机。一旦应用层打开一个 SOCK_STREAM 类型的套接字，并发出写数据请求，就会调用 TCP 协议实现的传送例程来处理所有从打开的套接字上传来的写数据请求。TCP 传送的特点如下。

- 异步传送：TCP 的实际传送独立于应用层。
- 汇集从套接字传送的数据，形成 TCP 协议的数据段，复制到 Socket Buffer。
- 管理和维护 Socket Buffer 链表，形成 Socket Buffer 传送队列缓存传送数据，准备以后发送。
- 管理 Socket Buffer 缓冲区分配，如果队列中最后一个 Socket Buffer 已满，但套接字缓冲区中还有新的数据要写入，则分配一个新的 Socket Buffer 以接收数据。

如网络设备支持 Scatter/Gather I/O 功能，即所谓的 chained DMA，TCP 就为网络设备建立一个传送缓冲区链表，该链表建立在 struct shared info 数据结构的 fragement 链表中。

9.5.1　将数据从用户地址空间复制到内核 Socket Buffer

如前所述，当应用程序打开一个 SOCK_STREAM 类型的套接字，并向套接字写数据时，会调用 TCP 协议实例的 tcp_sendmsg 函数在打开的套接字上处理写请求。所有的写操作在套接字层会转而调用这个函数，所以理解 TCP 在发送端操作的最好方式就是分析该函数的处理过程。

```
int tcp_sendmsg(struct kiocb *iocb, struct socket *sock, struct msghdr *msg, size_t size)
```

tcp_sendmsg 函数是 TCP 协议初始化时在协议函数块中注册发送函数。完成的功能为：

- 将数据复制到 Socket Buffer 中。
- 把 Socket Buffer 放入发送队列。
- 设置 TCP 控制块结构，用于构造 TCP 协议发送方的头信息。

1. 数据来源和函数局部变量初始化

（1）源数据结构描述

应用层传输的 messages 通过参数 struct msghdr *msg 传入 TCP 层，tcp_sendmsg 函数从该参数中提取传送 messages 的描述信息，来初始化相关的局部变量。关于 struct msghdr 数据结构的含义在 9.2.4 节中已经讲解，这里就不再赘述。

在 tcp_sendmsg 函数中复制数据要用到的数据域主要为 msg_iov、msg_iovlen、msg_flags。

（2）函数局部变量的初始化

- iov：提取参数 struct msghdr *msg 的数据块链表的起始地址 msg_iov。
- iovlen：提取参数 struct msghdr *msg 的数据块个数 msg_iovlen。
- flags：提取参数 struct msghdr *msg 中数据发送控制标志 msg_flags。标志可以是 MSG_MORE、MSG_DONTWAIT 等表明数据是如何发送的信息。
- tp：指向 struct tcp_sock 数据结构，其中包含了 TCP 选项。该结构通过参数 struct sock *sk 传入。
- skb：指向新分配的 Socket Buffer，用于存放要传送的数据。
- mss_now：存放当前打开的套接字最大段的长度（MSS ：maximum segment size），该值通常与 MTU 相关。
- timeo：存放 SO_SENDTIMEO（套接字设定的传送超时的时间）选项，除设置了 MSG_DONTWAIT 标志外。

```
int tcp_sendmsg(struct kiocb *iocb, struct socket *sock, struct msghdr *msg,
        size_t size)                                                    //net/ipv4/tcp.c
{ …
  flags = msg->msg_flags;
  timeo = sock_sndtimeo(sk, flags & MSG_DONTWAIT);
```

2. 数据复制的准备

（1）等待连接建立

在开始传送数据前，首先应与通信端建立连接。如果当前打开的套接字状态不是 TCPF_ESTABLISHED 或 TCPF_CLOSE_WAIT，说明 TCP 协议实例还没有准备好传送数

据，它必须等待连接建立起来。由 SO_SNDTIMEO 套接字选项设定等待连接超时的时间值作为参数，传给等待连接建立函数 sk_stream_wait_connect。

```
if ((1 << sk->sk_state) & ~(TCPF_ESTABLISHED | TCPF_CLOSE_WAIT))
     if ((err = sk_stream_wait_connect(sk, &timeo)) != 0)
          goto out_err;
```

（2）创建发送数据

理解发送数据的创建过程我们需首先知道以下几个问题：

- 发送数据来源是用户地址空间，通过输入参数 struct msghdr 传入。我们从中提取存放传送数据数组指针和发送的数据 messages 成员个数，分别保存在局部变量 iov 和 iovlen 中。
- 将数据复制到内核地址空间处，Socket Buffer 缓冲区中。
- 什么时候分配 Socket Buffer 缓冲区，发送等待队列中尚无缓冲的 Socket Buffer 或队列中最后一个 Socket Buffer 已满。

准备将发送数据从用户地址空间复制到内核地址空间的处理过程如下：

```
 /* mss_now 当前套接字可发送的最大 TCP 数据段的大小，这是根据 MTU 设定的*/
 mss_now = tcp_current_mss(sk, !(flags&MSG_OOB));

/*提取发送数据地址和发送数据总数信息*/
 iovlen = msg->msg_iovlen;
 iov = msg->msg_iov;
 copied = 0;
 …
 /*准备从用户地址空间复制数据，设定与复制相关的参数*/
 while (--iovlen >= 0) {
      int seglen = iov->iov_len;                          //I/O 数组中数组成员个数
      unsigned char __user *from = iov->iov_base;         //I/O 数组基地址
      iov++;
      while (seglen > 0) {
           int copy;
           skb = tcp_write_queue_tail (sk );              //指向队列中最后一个 skb
           if (!tcp_send_head (sk) ||                     //队列中还没有 skb
              (copy = size_goal - skb->len) <= 0) {       //或余下空间不足
new_segment:
           if (!sk_stream_memory_free(sk))                //如已无内存可供分配
                    goto wait_for_sndbuf;                 // 等待新的内存空间释放
skb = sk_stream_alloc_skb(sk, select_size(sk), sk->sk_allocation);
                                                          //分配 Socket Buffer
        if (!skb)
            goto wait_for_memory;                         //Socket Buffer 分配不成功
/*分配 Socket Buffer，如果分配成功则 skb 变量为返回的 Socket Buffer 地址指针，设置 skb 的某些数据域，如校验和
  计算方式，看网络设备硬件是否可计算校验和，将 skb 放入队列，向 skb 复制数据*/
if (sk->sk_route_caps & NETIF_F_ALL_CSUM)
           skb->ip_summed = CHECKSUM_PARTIAL;
      skb_entail(sk, skb);
      copy = size_goal;
 }
```

3. 从用户地址空间复制数据到 Socket Buffer

数据复制开始于第一个 while 语句，局部变量 seglen 初始化为一个数据块中的字节数。在开始数据复制以前，根据当前传送等待队列中 Socket Buffer 现有空间的状态，数据可能复制到缓冲区的以下几处：

（1）Socket Buffer 的数据缓冲区

如果等待队列中的最后一个 skb 的数据缓冲区中还有空间，则将数据放入队列的最后一个 skb 数据缓冲区中，如图 9-8 所示。

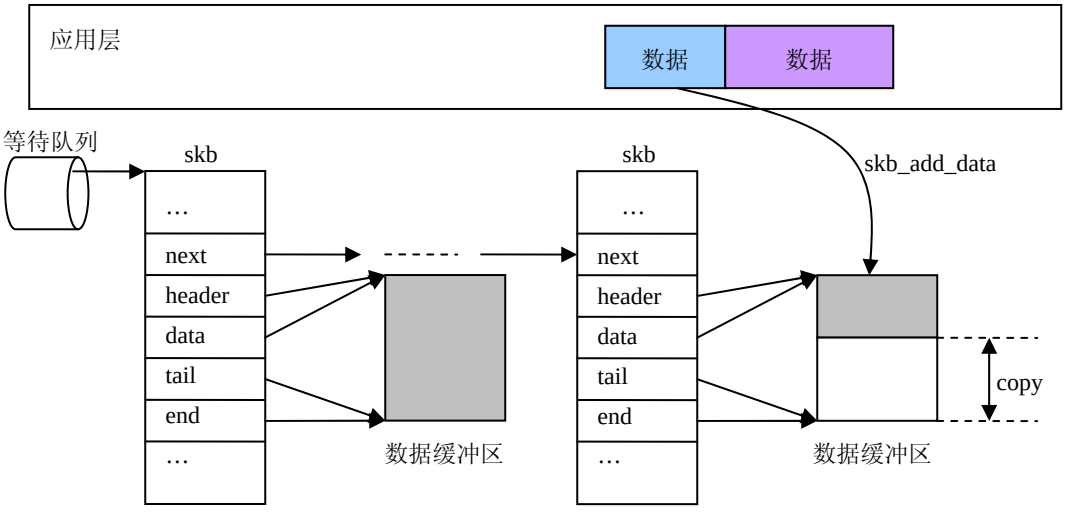


图 9-8　将用户层数据复制到内核地址空间等待传送队列

skb_tailroom(skb)就是用于判断 Socket Buffer 的数据缓冲区中是否还有剩余空间，并返回剩余空间的大小。copy 代表了本次复制的数据量。函数 skb_add_data 完成具体的数据复制操作。

（2）Socket Buffer 的 struct skb_shared_info 数据结构的 frags 页面数组的最后一个页面

如果 Socket Buffer 的数据缓冲区已满，则查看紧接在 Socket Buffer 数据缓冲区后的 struct skb_shared_info 数据结构的 frags 页面数组中的最后一个页面是否还有空间，如果有就将数据合并到最后一个页面中，如图 9-9 所示。

函数 skb_can_coalesce 用于计算页面中是否仍有剩余空间接收新的数据，如果 merge = 1，则表明可以将数据合并到 i = skb_shinfo(skb)->nf_frags 第 i 个页面中。由函数 skb_copy_to_page 完成将数据复制到页面的操作。

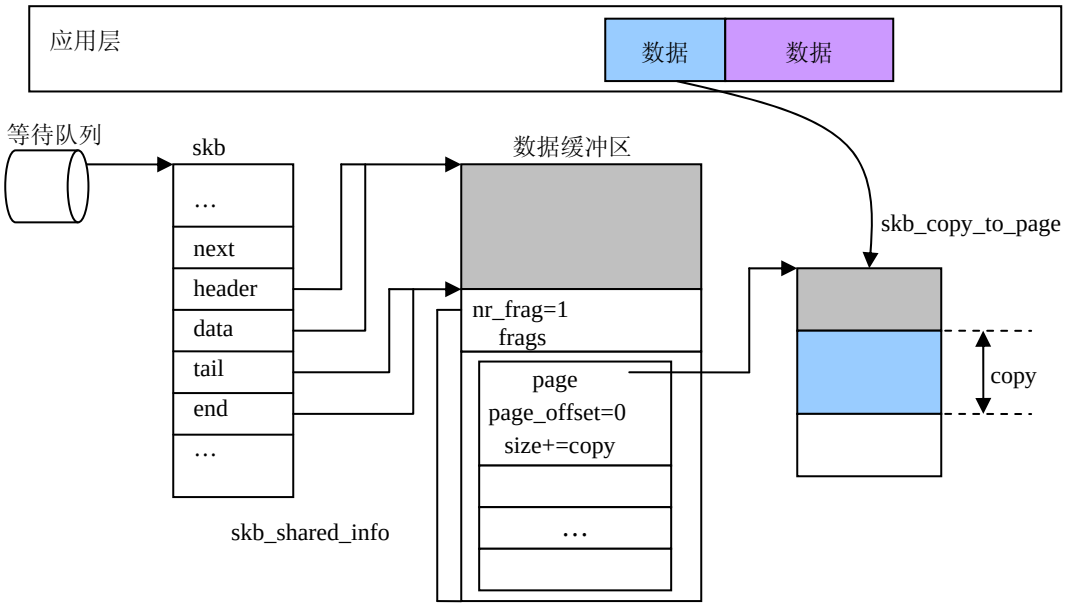


图 9-9　将用户层数据复制到 frags 页面的剩余空间

（3）复制到 frags 页面数组的新页面

如果 struct skb_shared_info 数据结构中 frags 页面数组的最后一个页面也已满，且 nr_frag 的成员数小于数组的最大值（MAX_SKB_FRAGS），则分配一个新的页面，将数据复制到新页面中，如图 9-10 所示。

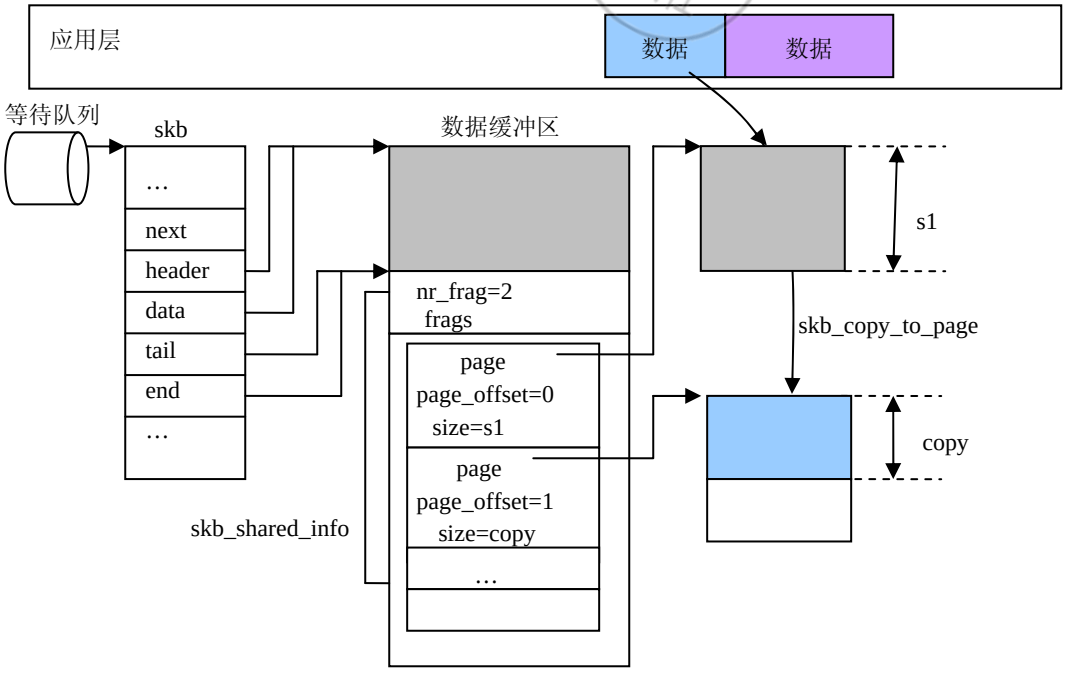


图 9-10　将用户层数据复制到 frags 的新页面中

由 sk_stream_alloc_page 函数完成新页面的分配。将数据复制到新页面仍由 skb_copy_to_page 函数完成，该函数会转而调用 csum_and_copy_from_user 函数从用户地址空间复制数据到内核地址空间，并计算校验和或调用 copy_from_user 函数。

更新 skb 中描述数据大小的信息，如 skb->len、skb->data_len、skb->truesize 等。

如果 struct skb_shared_info 的页面数组中所有页面已填满，并且 nr_frags=MAX_SKB_FRAGS，即页面数据量已达最大值。当前 TCP 段已不能再容纳更多数据，设置当前数据段 TCP 头的 PSH 标志（tcp_mark_push(tp, skb)），表明当前数据段已可以发送。如果网络设备支持 Scatter/Gather I/O 功能，则分配一个新的 Socket Buffer 来接收数据。此过程与 9.3.1 所描述的过程一样。

4. 更新 Socket Buffer 的描述信息

这时数据复制已完成，Socket Buffer 中已增加了新的数据，现在需要对 Socket Buffer 的管理数据结构 struct sk_buff 中的信息进行更新，以反映新增加的数据在 Socket Buffer 中的描述。

如果 merge=1，则说明数据合并到了 struct skb_shared_info 数据结构的 frags 页面数组的最后一个页面 i 中。需对第 i 个页面管理数据结构中描述数据量大小的 size 域更新，因复制的数据量为 copy：skb_shinfo(skb)->frags[i - 1].size +=copy。

将数据复制到 frags 页面数组的新页面中，应更新 struct skb_shared_info 数据结构中 frags 数组中的页面数量，以及新页面在数组中的偏移量。由 skb_fill_page_desc 函数填写新页面在 frags 中的描述信息。

更新 TCP 层 Socket Buffer 控制缓冲区信息。

最后，如果应用层当前 messages 在本次复制中没有全部移至 Socket Buffer 中，或应用层还有别的 messages（seglen 为当前 messages 的数据量，seglen - copy 不为 0；iovlen 总的 messages，iovlen – 1 不为 0），循环重复以上准备数据复制和数据复制过程。

5. tcp_sendmsg 函数的结束处理

这时，tcp_sendmsg 函数大部分功能已完成：数据已从用户地址空间移至 Socket Buffer，Socket Buffer 的 frags 页面数组已更新为连接了一组装满发送的数据段的页面。最终 TCP 数据段什么时候向外发送，分为以下几种情形来处理。

（1）更新控制循环复制的各变量

在复制数据的过程中，单次循环复制的数据为 copy，每次循环代码执行完一遍后用 copy 来更新的控制变量如下。

- copyed：数据复制的总量，每循环一次将 copy 累加到 copyed 上。
- from：数据复制时源数据的起始地址，每循环一次，from 指针向前移动 copy 个字节，指向下次循复制数据的起始地址。
- 序列号：在 struct tcp_sock 数据结构中的数据段序列号 tp->write_seq 代表了 TCP 发送缓冲区保存的字节数，每次累加 copy，以及控制缓冲区中的 end_seq 每次累加 copy。
- TCP 协议头中的 PSH 标志：当局部变量 copyed 值为 0 时，表明这是数据复制循环体的第一次执行数据复制，数据移至最初数据段缓冲区中，所以将 TCP 协议头中的 PSH 标志（是否将数据推送出去）设为 0。

这时 TCP 协议头的信息并不直接设置在数据包中存放协议头的部位，而是设置在 TCP 层 Socket Buffer 控制缓冲区的 TCP 协议头信息中，当数据段从队列中取出发送时，才创建 TCP 协议头。

```
if (!copied)
        TCP_SKB_CB(skb)->flags &= ~TCPCB_FLAG_PSH;
```

（2）什么时候传送 TCP 的 Socket Buffer

- 立即发送：这时即使只是一个小的 TCP 数据段也立即向外发送。函数 forced_push 用于查看是否立即发送数据段，如果立即发送，则调用 tcp_mark_push 函数设置 TCP 协议头中的 PSH 标志。

```
if (forced_push ( tp ) ) {
                    tcp_mark_push ( tp, skb);
                    __tcp_push_pending_frames (sk, mss_now, TCP_NAGLE_PUSH);
               } else if (skb == tcp_send_head( sk ) )
                    tcp_push_one (sk, mss_now);
               continue;
```

- 缓存数据段：如果队列中还没有足够的缓冲区或页面，则等到有一定数据量的有效缓冲区后再发送。

```
wait_for_sndbuf:
               set_bit(SOCK_NOSPACE, &sk->sk_socket->flags);
wait_for_memory:
               if (copied)
                    tcp_push(sk, flags & ~MSG_MORE, mss_now, TCP_NAGLE_PUSH);
               if ((err = sk_stream_wait_memory(sk, &timeo)) != 0)
                    goto do_error;
```


9.5.2 TCP 数据段输出

上节我们描述的是 TCP 协议层与套接字之间的接口，是数据从用户地址空间复制到内核地址空间的过程。从 TCP 协议实例向外传送的数据段，除了来自用户地址空间的数据包外，还有大量由 TCP 协议层本地产生的数据包，如：

- 重传数据包 tcp_retransmit_skb。
- 探测路由最大传送单元数据包。
- 发送复位连接数据包。
- 发送连接请求数据包。
- 发送回答数据包。
- 0 窗口探测数据包等。

所有这些数据包通过不同的函数在 TCP 层创建连同从用户地址空传来的数据包，最终都是通过 tcp_transmit_skb 函数向 IP 层传送的。这节描述 TCP 数据段实例向外发送的过程。TCP 协议实例各项功能与最终发送函数的关系如图 9-11 所示。

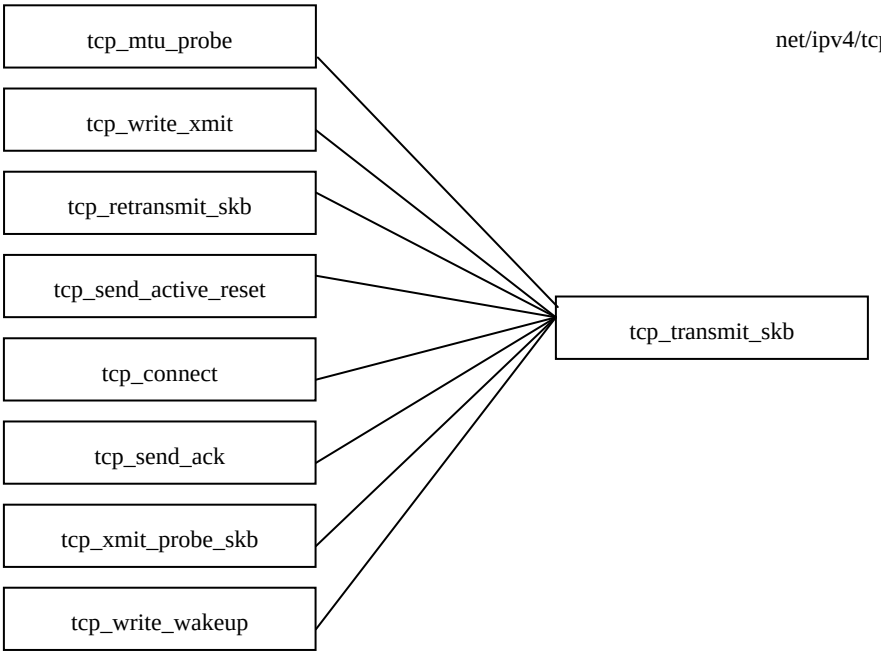


图 9-11　使用 tcp_transmit_skb 函数发送 TCP 数据段的函数

从上图可见，TCP 数据段最终是通过 tcp_transmit_skb 函数传送给 IP 层的。再回过头看 tcp_sendmsg 函数的实现过程。当它复制完来自应用层的数据后，无论是立即发送存放了 TCP 数据段的 Socket Buffer，还是缓冲发送，它们调用的__tcp_push_pending_frames/ tcp_push_one/ tcp_push 函数，最终是通过调用 tcp_write_xmit 函数转而调用 sk_transmit_skb 函数来将数据段发送出去的。在这一节主要通过分析 sk_transmit_skb 函数的实现来理解 TCP 数据段的发送过程。

1. tcp_transmit_skb 函数初始化

tcp_transmit_skb 函数是实际实现 TCP 数据段发送的地方，该函数会在 TCP 协议实例状态机的各状态阶段需要发送数据段时调用。tcp_transmit_skb 函数完成的主要功能包括创建 TCP 协议头、将数据段传给 IP 层向外发送。构造向外发送的 TCP 数据段需要的重要信息有以下几种。

- inet：指向 inet 选项结构，其中包含了 AF_INET 地址族 SOCK_STREAM 类套接字的所有信息。
- tp：指向 TCP 选项结构，其中包含了 TCP 配置和连接的大部分信息。
- tcb：指向 TCP 控制缓冲，包含了用于构造 TCP 协议头的各项标志。
- th：指向 TCP 协议头数据结构，此后它将指向 skb 中存放 TCP 协议头的位置。
- icsk：inet 连接控制套接字。

2. 确定 TCP 数据段协议头包含的内容

要构造的 TCP 协议头，包括协议头本身和协议选项两个部分。如同构造两个部分，需根据当前发送的数据段的类型而定：是包含实际传送数据的包还是传送同步信息的 SYN

段。这阶段的处理包括以下步骤：

（1）克隆 Socket Buffer

如果输入参数 clone_it 指明当前 Socket Buffer 还有其他进程在使用，则需要首先克隆套接字缓冲区。

```
static int tcp_transmit_skb (struct sock *sk, struct sk_buff *skb, int clone_it,
                gfp_t gfp_mask)                                    //net/ipv4/tcp_output.c
{
        if (likely(clone_it)) {
        if (unlikely(skb_cloned(skb)))
              skb = pskb_copy(skb, gfp_mask);
        else
              skb = skb_clone(skb, gfp_mask);
        if (unlikely(!skb))
              return -ENOBUFS;
  }
```

（2）根据当前数据包是否为发送 SYN 数据段的包确定 TCP 协议选项长度。

接下来查看输出数据包是否为一个 SYN 数据包（TCPCB_FLAG_SYN），如果是，则调用 tcp_syn_options 函数构造 SYN 数据段的选项数据，这时 TCP 选项通常包含时间戳信息、窗口大小和选择回答（SACK）等。否则调用 tcp_established_options 函数构造常规数据段的 TCP 协议选项，两个函数都将返回 TCP 选项数据块的长度。协议头的总长度 tcp_header_size 为 TCP 选项的长度+协议头的大小。

```
  if (unlikely(tcb->flags & TCPCB_FLAG_SYN))
        tcp_options_size = tcp_syn_options(sk, skb, &opts, &md5);
  else
        tcp_options_size = tcp_established_options(sk, skb, &opts,
                                    &md5);
  tcp_header_size = tcp_options_size + sizeof(struct tcphdr);
```


（3）网络拥塞控制管理

确定网络上有多少数据包最好。在大多数情况下是按保守的状况处理，网络上有多少数据包最好的详细信息从接收到的 SACK 的信息来确定的，这样我们可以放更多的包在网络上。使用这项功能来做拥塞控制， tp->packets_out 用于确定发送队列中是否为空。拥塞控制计算方式为：

在传送队列上的数据包数 – 留在网上的数据包 + 须快速重传的数据包

如果结果表明不会造成网络阻塞，则设置发送数据事件标志。

```
  if (tcp_packets_in_flight(tp) == 0)
        tcp_ca_event(sk, CA_EVENT_TX_START);
```

（4）构造 TCP 协议头

现在我们已获取了需要构造 TCP 协议头的信息。开始创建 TCP 协议头：

① 填写相关数据域。

② 计算窗口的大小。如果是 SYN 数据段则不需计算窗口大小，一般的数据段调用 tcp_select_window 函数计算窗口大小。

③ 如果设置的是紧急模式（urgent mode），计算紧急模式指针，在 TCP 头中设置紧急传送标志。

④ 将选项写入 TCP 协议头（tcp_options_write），用 TCP_ECN_send 传送网络阻塞通知。

```
    skb_push(skb, tcp_header_size);              //设置 skb->data 指针
    skb_reset_transport_header(skb);             //设置 skb->transport_header 指针
    skb_set_owner_w(skb, sk);                    //设置 skb 所属的套接字
    th = tcp_hdr(skb);
    th->source = inet->sport;                    //TCP 头的源端口号
    th->dest   = inet->dport;                    //TCP 头的目的端口号
    th->seq    = htonl(tcb->seq);                //数据段的初始序列号
    th->ack_seq= htonl(tp->rcv_nxt);             //ACK 序列号
    *(((__be16 *)th) + 6)  = htons(((tcp_header_size >> 2) << 12) |
         tcb->flags);
    if (unlikely(tcb->flags & TCPCB_FLAG_SYN)) {       //窗口大小
         th->window = htons(min(tp->rcv_wnd, 65535U));
    } else {
         th->window = htons(tcp_select_window(sk));
    }
  if (unlikely(tcp_urg_mode(tp) &&
              between(tp->snd_up, tcb->seq + 1, tcb->seq + 0xFFFF))) {
         th->urg_ptr= htons(tp->snd_up - tcb->seq);
         th->urg= 1;
    }
    tcp_options_write((__be32 *)(th + 1), tp, &opts, &md5_hash_location);
    if (likely((tcb->flags & TCPCB_FLAG_SYN) == 0))
         TCP_ECN_send(sk, skb, tcp_header_size);
```

3. 发送数据

最后调用实际传送函数将该数据段传送给 IP 层。首选更新发送的数据包类型的统计数据。然后发送数据段，该数据段将放入队列等待下一阶段处理。err = icsk->icsk_af_ops->queue_xmit(skb, 0)，在 TCP 协议中，queue_xmit 函数指针指向的 ip_queue_xmit 函数，ip_queue_xmit 函数是 TCP 层向 IP 层传送数据段时调用的函数，该函数由 IP 层实现，是 IP 层提供给 TCP 层的接口函数，该函数的实现我们已在第 7 章介绍过。

```
    if (likely(tcb->flags & TCPCB_FLAG_ACK))
         tcp_event_ack_sent(sk, tcp_skb_pcount(skb));
    if (skb->len != tcp_header_size)
         tcp_event_data_sent(tp, skb, sk);
    if (after(tcb->end_seq, tp->snd_nxt) || tcb->seq == tcb->end_seq)
         TCP_INC_STATS(sock_net(sk), TCP_MIB_OUTSEGS);
    err = icsk->icsk_af_ops->queue_xmit(skb, 0);
  …
```

9.5.3 发送过程的状态机

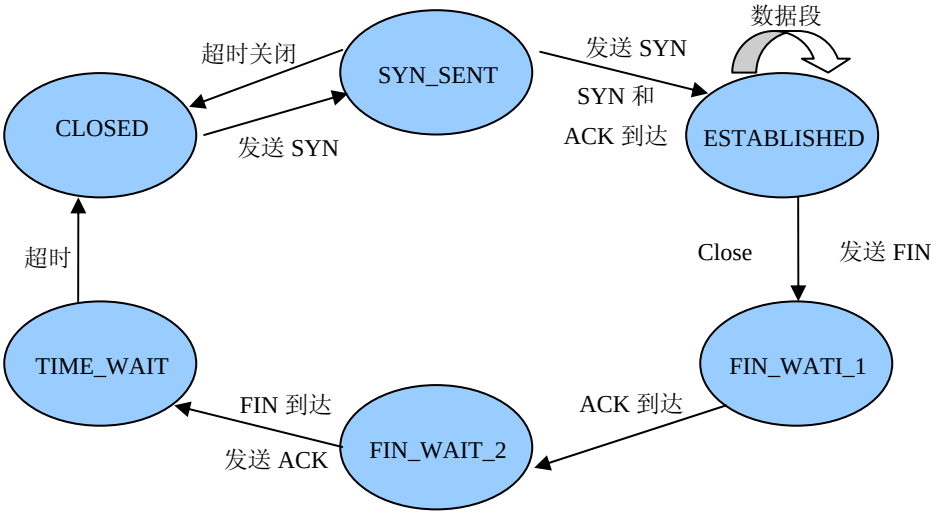


图 9-12 TCP 协议发送数据段连接的状态机

9.6 TCP 套接字的连接管理

TCP 是面向连接的协议，TCP 协议实例的大量处理过程都与建立和断开连接相关。使用 SOCK_STREAM 类型的应用是典型的客户/服务器类型的应用。当客户端向服务器发出连接请求时，服务器响应客户端的连接请求并建立连接，双方交换数据，在此期间还有大量的其他状态需要 TCP 协议实例维护和管理。TCP 协议的状态切换关系如图 9-13 所示。

在 TCP 的状态机中为一个连接定义了 11 个不同的状态，基于 TCP 连接当前的状态和在该状态时 TCP 接收到的数据段类型，TCP 定义了相应的规则指明 TCP 连接状态如何从一个状态切换到另一个。如图 9-10 所示，如果一个应用程序在 TCP 套接字的 CLOSED 状态时执行 OPEN 操作，TCP 发送一个 SYN，TCP 的新状态就变为 SYN_SENT。如果 TCP 接收的下一个数据段是一个带 ACK 的 SYN 数据段，则 TCP 送出一个 ACK 数据段，这时新的连接建立，TCP 套接字的状态就为 ESTABLISHED，ESTABLISHED 状态通常为连接两端交换数据的状态。

从 ESTABLISHED 出来的两个箭头处理连接终止，如果一个应用在收到 FIN 数据段前调用了 CLOSE，TCP 状态切换到 FIN_WAIT_1 状态，但如 TCP 套接字在 ESTABLISHED 状态时收到了 FIN 数据段（正常关闭），TCP 状态切换到 CLOSE_WAIT 状态。

在管理 TCP 连接时通信两端交换的数据包如图 9-14 所示。图中客户端声明的 MSS 是 536，服务器端声明的 MSS 是 1460，双方的 MSS 可以不同。

一旦连接建立，客户端产生一个请求发送给服务器，我们假设这个请求在一个数据段中（即小于服务器声明的 MSS=1460）。服务器处理请求送出回答，我们假设回答也能填充

在一个数据段中（小于客户端的 MSS=536），连接双方交换数据。

从图中可以看到，结束连接双方交换了 4 个段，注意执行活动 CLOSE 的结果是进入 TIME_WAIT 状态，TIME_WAIT 状态也将在这一节中进行介绍。

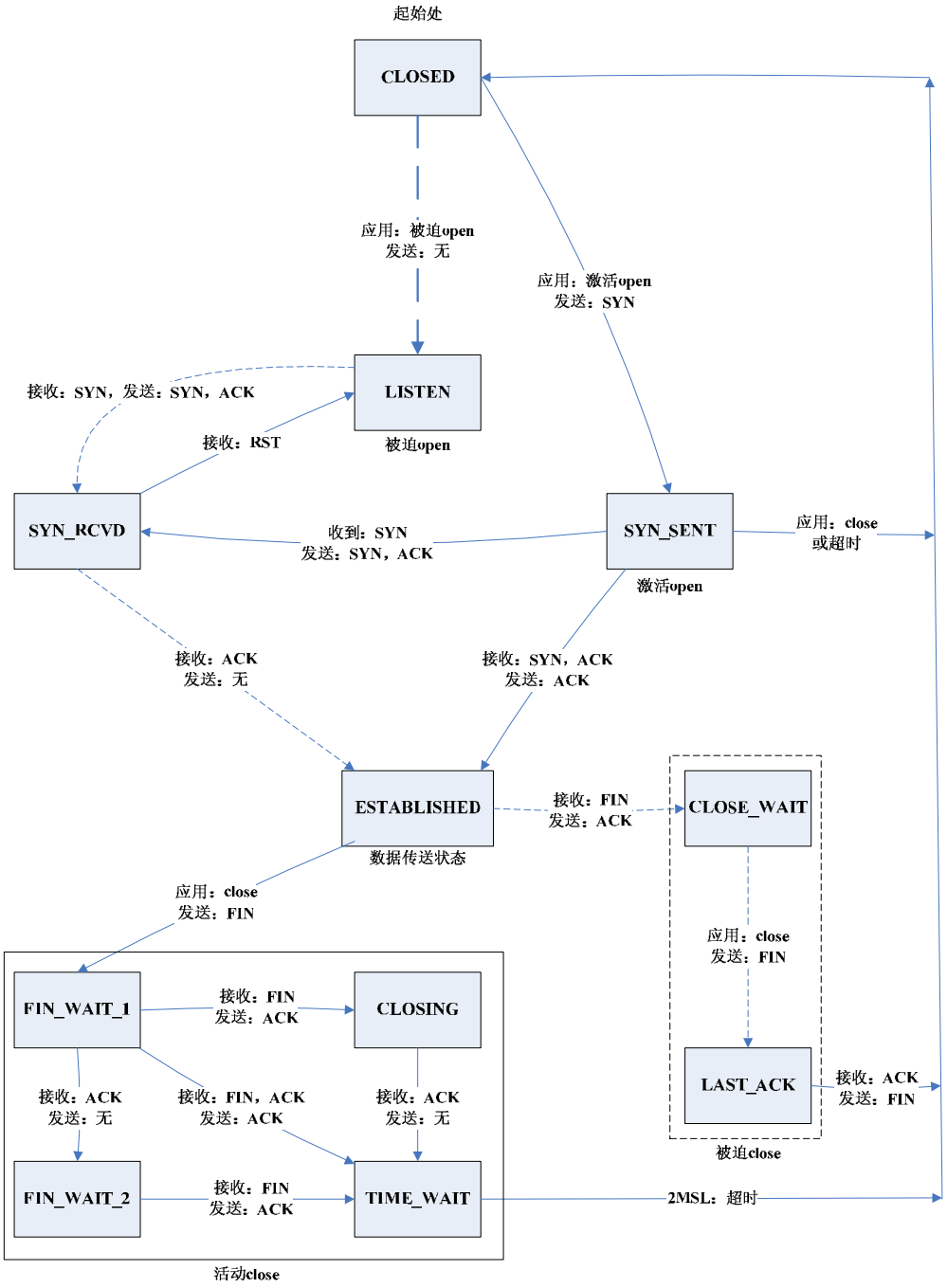


图 9-13　TCP 状态机

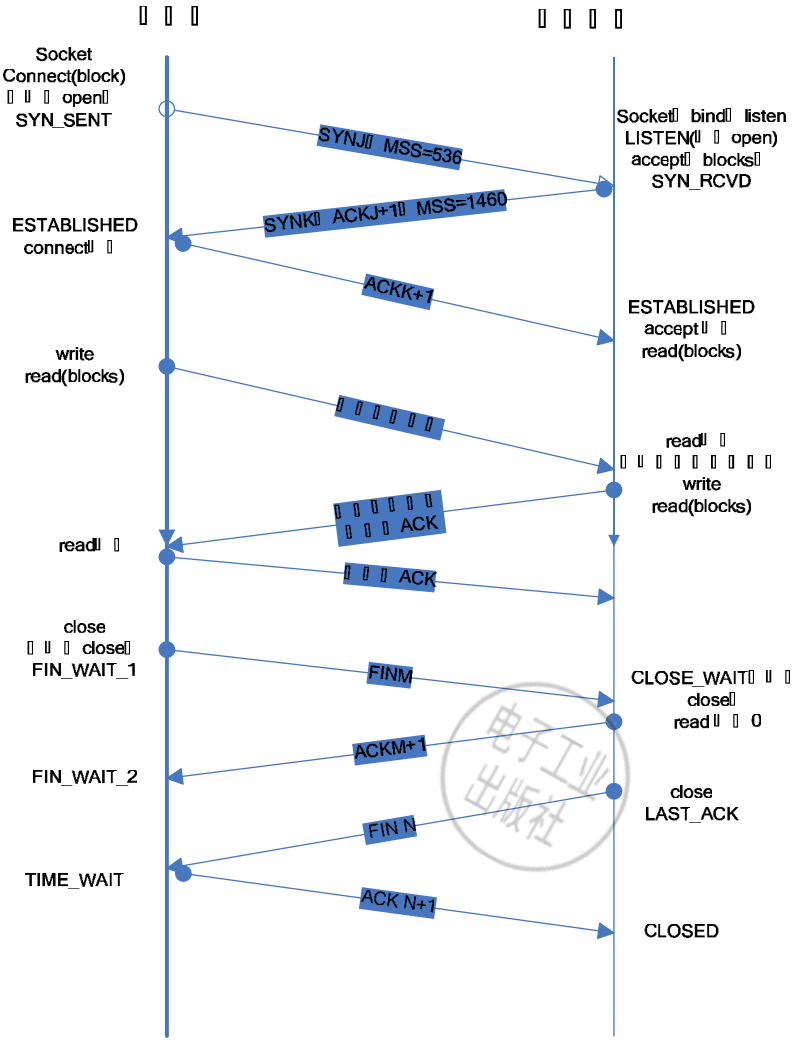


图 9-14　管理 TCP 连接发送的数据包

TCP 协议实例连接状态存放在 struct sock 数据结构的 state 数据域中。当 TCP 协议实例连接处于不同状态时，对数据包的处理不一样，所以每个输入的数据包都要来查询 TCP 状态机，整个状态机制划分成 3 个阶段：

第一阶段：连接建立阶段。

第二阶段：数据传送阶段。

第三阶段：断开连接阶段。

管理 TCP 连接状态机的重要函数是 tcp_rcv_state_process，它的主要任务是连接管理；tcp_rcv_state_process 函数根据接收到的信息切换 TCP 协议实例的连接状态；但在连接还没有建立时，TIME_WAIT 是收到数据包所在的唯一状态。

在这一节中，我们将按照 TCP 协议实例连接状态的管理过程：初始化连接、连接状态管理、连接已建立的处理、连接断开、TIME_WAIT 的顺序来分析 TCP 连接管理在 Linux 内核中的实现。

9.6.1　TCP 连接初始化

TCP 通信双方实例在传送数据之前必须已连接，如在本章第一节中所描述的，一个连接的建立需要通过三次握手处理以避免出现连接错误。如图 9-10 所示的 TCP 状态机中，连接建立阶段要处理以下状态。

- LISTEN：套接字被迫打开后，本地 TCP 等待一个请求连接的 SYN 数据包来建立连接。
- SYN_SENT：发送 SYN 请求连接后，本地 TCP 等待通信伙伴 TCP 实例响应连接请求。
- SYN_RECV：本地 TCP 等待连接已建立的回答（用 ACK 回答 SYN）。
- ESTABLISHED：连接已建立，通信双方可以交换数据。建立连接阶段结束。

现在来看看连接建立阶段是如何从应用程序发起，并通过套接字传送到 TCP 层的。关于套接字编程将在第 12 章中进行详细讲解，这里在分析 TCP 连接建立实现之前，先简单了解其代码关系。

1. 应用程序创建套接字

```
int sd;
sd = sock( AF_INET， SOCK_STREAM，NULL );
```

上述两行代码，创建并打开了一个 SOCK_STREAM 类型的套接字（TCP 协议使用的套接字），地址类型为 AF_INET；套接字创建成功后返回套接字的描述符 sd。随后我们通过套接字描述符访问套接字。

2. 应用程序发连接请求

通过套接字向服务器发连接请求。建立连接需要的信息包括服务器的 IP 地址、端口号、地址类型等。AF_INET 类地址信息由 struct sockaddr_in 数据结构描述。

```
struct sockaddr_in {
    sa_family_t   sin_family;  /* 地址族 */
    in_port_t     sin_port;    /* 端口号 */
    struct in_addr sin_addr;    /* IPv4 的 IP 地址 */
 };
```

定义一个 struct sockaddr_in 数据结构变量，对各数据域赋初值：

```
struct sockaddr_in daddr;
daddr.sin_family = AF_INET;
daddr.sin_sin_addr.s_addr = htonl( 目标 IP 地址);
daddr.sin_port = htons( 目标端口号 );
```

3. 设置 TCP 选项

在请求连接以前，应用程序也可以通过 setsockopt 系统调用设置 TCP 的选项来控制管理连接传送过程；setsockopt 调用 tcp_setsockopt 函数来设置 TCP 的选项，这些选项即在 9.2.4 节中讲解的 TCP 选项，如 TCP_MAXSEG、CP_NODELAY 等。这些选项会通过套接字传给 TCP 的连接初始化函数 tcp_v4_connect。

4. 发出连接请求

```
connect( sd , (struct sockaddr *)&daddr, sizeof(struct sockaddr));
```

套接字层支持客户端连接请求的调用函数是 connect。当在一个打开的套接字上执行 connect 时，套接字层就调用协议函数来处理连接请求。对于 AF_INET 地址族的 SOCK_STREAM 类协议，执行套接字层连接请求的函数是 tcp_v4_connect，相应的以上连接地址通过 struct sock 数据结构传到 tcp_v4_connect 函数。

9.6.2 TCP 状态从 CLOSED 切换到 SYN_SENT

tcp_v4_connect 函数将初始化一个对外的连接请求。tcp_v4_connect 函数完成的主要任务如图 9-15 所示。

创建一个有 SYN 标志的请求连接数据包发送出去。

将 TCP 的状态从初始的 CLOSED 切换到 SYN_SENT 状态。

初始化 TCP 部分选项，如数据包序列号、窗口大小、MSS、套接字传送超时等。

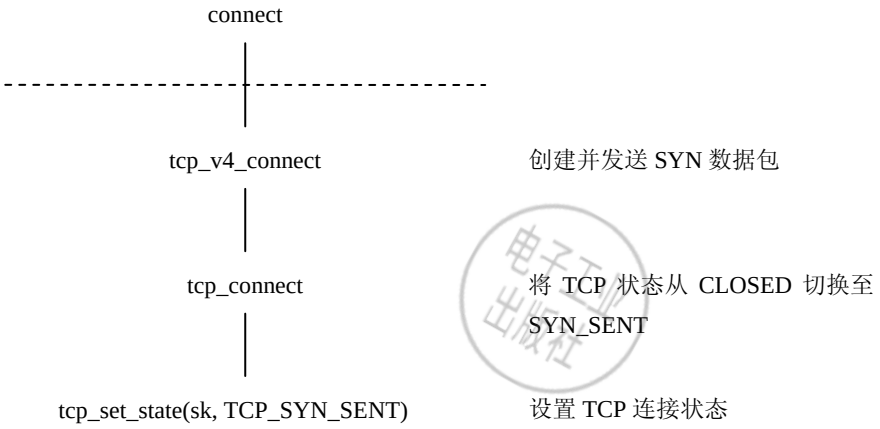


图 9-15　tcp_v4_connect 函数完成的主要任务

以下我们分析 tcp_v4_connect 函数的源码实现流程。

1. 函数初始化部分

（1）输入参数及局部变量初始化

① 输入参数

tcp_v4_connect 函数有以下几个输入参数。

- struct sock *sk：指向打开的套接字指针，在该套接字上建立连接。
- struct sockaddr *uaddr：套接字地址结构，其中包含了要连接另一端的目标 IP 地址、端口号和地址类型。
- int addr_len：套接字地址的长度。

② 局部变量初始化

```
int tcp_v4_connect(struct sock *sk, struct sockaddr *uaddr, int addr_len)
{                                                          //net/ipv4/tcp_ipv4.c
  struct inet_sock *inet = inet_sk(sk);
  struct tcp_sock *tp = tcp_sk(sk);
  struct sockaddr_in *usin = (struct sockaddr_in *)uaddr;
```

```
struct rtable *rt;
__be32 daddr, nexthop;
int tmp;
int err;
```

- tp：是指向 struct sock 数据结构中的 struct tcp_sock TCP 特有套接字结构的指针。
- rt：是路由表中的一个记录。它指向路由高速缓冲区中的一个路由，这个路由是为通过该套接字向外发送数据包选定的路由。
- daddr：将初始化为 IP 目的地址。
- nexthop：是路由上网关的 IP 地址（如果有网关）。

（2）正确性检查

确定输入参数给定的地址正确后，以用户给定的套接字地址（由输入参数 uaddr 给出）初始化网关地址 nexthop 和目标地址 daddr。

```
if (addr_len < sizeof(struct sockaddr_in))              //地址长度正确
      return -EINVAL;
if (usin->sin_family != AF_INET)                        //地址类型为 AF_INET 族地址
      return next=daddr=usin->sin_addrs_addrs -EAFNOSUPPORT;
```

3）根据 IP 选项重新设置目标地址和网关地址

查看该套接字是否设定了源路由 IP 选项，如果设定了源路由选项，而数据包的目标地址不为空，则从用户给定的源路由列表中取出第一个 IP 地址（opt->faddr）赋给网关地址。

```
if (inet->opt && inet->opt->srr) {
      if (!daddr)
            return -EINVAL;
      nexthop = inet->opt->faddr;
}
tmp = ip_route_connect(&rt, nexthop, inet->saddr,
                RT_CONN_FLAGS(sk), sk->sk_bound_dev_if,
                IPPROTO_TCP,
                inet->sport, usin->sin_port, sk, 1);
```

2. 寻址连接目标，确定路由

在确定了目标地址、网关地址等信息后，调用 ip_route_connect 函数来寻址目标路由。寻址目标路由的依据源是：目标 IP 地址、网络设备接口、源端口号（inet->sport）、目标端口号（usin->sin_port）等信息，寻址好的路由在路由表中的记录索引返回到 rt 变量中。

TCP 连接必须建立在唯一主机地址上（unicast address），所以如果在路由缓冲区中的路由是组传送地址（MULTICAST）或广播地址（BROADCAST），则返回错误。

```
if (rt->rt_flags & (RTCF_MULTICAST | RTCF_BROADCAST)) {
      ip_rt_put(rt);
      return -ENETUNREACH;                              //组传送地址或广播地址，返回错误
}
if (!inet->opt || !inet->opt->srr)                      //如果没有设置 IP 选项或 IP 源路由选项
      daddr = rt->rt_dst;                               //使用路由表寻址的路由
if (!inet->saddr)
      inet->saddr = rt->rt_src;
inet->rcv_saddr = inet->saddr;
if (tp->rx_opt.ts_recent_stamp && inet->daddr != daddr) {
      tp->rx_opt.ts_recent   = 0;
      tp->rx_opt.ts_recent_stamp = 0;
      tp->write_seq          = 0;
```

如果没有设置源路由选项，目标地址为寻址完路由后的目标地址，daddr = rt->rt_dst。

如果当前套接字建立连接的目标地址与套接字数据结构中保存的目标地址不一样，则复位原 TCP 继承的状态。

3. 建立连接过程的 TIME_WAIT 状态处理

当套接字当前处于 TIME_WAIT 状态时，套接字最后一次使用时间存放在与路由入口相关的 struct inet_peer 数据结构中。这时如果用户发出了连接请求，最近套接字接收信息的时间，需写入 struct tcp_sock 数据结构的 timestamp 数据域，其值从存放在 struct inet_peer 数据结构中的值获取。一旦连接重建立，这个方法是检测复制段的方法。

```
//提取最近套接字使用时间
if (tcp_death_row.sysctl_tw_recycle &&
      !tp->rx_opt.ts_recent_stamp && rt->rt_dst == daddr) {
         struct inet_peer *peer = rt_get_peer(rt);
         if (peer != NULL &&
            peer->tcp_ts_stamp + TCP_PAWS_MSL >= get_seconds()) {
               tp->rx_opt.ts_recent_stamp = peer->tcp_ts_stamp;
               tp->rx_opt.ts_recent = peer->tcp_ts;
         }     }

//目标端口号、目标地址，如果有选项，则更新选项头长度

   inet->dport = usin->sin_port;
   inet->daddr = daddr;
   inet_csk(sk)->icsk_ext_hdr_len = 0;
   if (inet->opt)
         inet_csk(sk)->icsk_ext_hdr_len = inet->opt->optlen;
   tp->rx_opt.mss_clamp = 536;                              //设置 MSS 初值
```

用输入参数端口号初始化套接字的目标端口，用前面确定的目标 IP 地址初始化套接字的目标地址。TCP 的扩展协议头长度为 0。但如果套接字有选项设置，则协议头长度初始化为选项头的长度。这个长度包括了 TCP 和 IP 协议头长度的总和。

初始化 MSS 为 TCP 允许接收的最大数据段 536。虽然我们是在找开的套接字上进行了相应的处理，这时标识套节字的信息还不完整，例如套节字源端口号这时就还没有分配。

4. 连接状态切换为 SYN_SENT

```
//将套接字状态设为 TCP_SYN_SENT，将套接字 sk 放入 TCP 连接管理哈希链表
tcp_set_state (sk, TCP_SYN_SENT);
   err = inet_hash_connect (&tcp_death_row, sk);
   if (err)
         goto failure;
//为连接分配一个临时端口号，以便于以后在哈希链表中查找套接字 sk

   err = ip_route_newports(&rt, IPPROTO_TCP,
                  inet->sport, inet->dport, sk);
   if (err)
         goto failure;
   sk->sk_gso_type = SKB_GSO_TCPV4;
   sk_setup_caps(sk, &rt->u.dst);
```

这时我们将套接字设置为 TCP_SYN_SENT 状态，调用 inet_hash_connect 函数把套接字指针 sk 放入 TCP 连接的哈希链表中。这时为连接分配一个临时的端口号，并将其放入

struct sock 数据结构的 sport 数据域。临时端口号是以后在哈希链表中查找已建立连接的套接字的关键字。如果套接字处于 ESTABLISHED 状态，当数据包到达这个端口时，则 TCP 可以通过端口号在哈希链表中快速查找到正确的套接字。

为了注解这个连接的目标地址，我们将 struct sock 数据结构的 dst 数据域指向与路由相连的目标地址 rt->u.dst。当数据包的目标路由是向外发送到网络上其他站点时，dst 将指向网关。如果目标地址是一个直接连接的主机地址，则 dst 的信息是目标主机的信息。

5. 发送连接请求

最后函数初始化第一个序列号，调用 tcp_connect 函数完成建立连接的工作，包括发送 SYN。在 tcp_connect 函数中将创建好的 SYN 数据段放入套接字的传送队列，最后调用 tcp_transmit_skb 函数发送出去。

```
        if (!tp->write_seq)
//初始化 TCP 数据段序列号
        tp->write_seq = secure_tcp_sequence_number(inet->saddr,
                                        inet->daddr,
                                        inet->sport,
                                        usin->sin_port);
  inet->id = tp->write_seq ^ jiffies;
//调用 tcp_connect 函数完成实际的连接建立操作

  err = tcp_connect(sk);
  rt = NULL;
  if (err)
        goto failure;
  return 0;
```

6. 连接建立处理失败

如果连接建立处理失败，则将套接字从已连接的哈希链表中移出并释放本地端口。将 TCP 状态切换回 CLOSED 状态，并返回错误信息。

```
failure:
  tcp_set_state(sk, TCP_CLOSE);
  ip_rt_put (rt );
  sk->sk_route_caps = 0;
  inet->dport = 0;
  return err;
}
```

9.6.3　TCP 连接的状态管理

TCP 协议连接初始化后的状态管理和切换由 tcp_rcv_state_process 函数来完成。TCP 协议实例收到数据包后（由 tcp_v4_receive 函数接收），TCP 必须查看协议头，以区分数据段类型是只包含纯传送负载数据，还是包含控制信息 SYN、FIN、RST、ACK 等，不同类型的数据包在 TCP 协议头中标志位的设置不同。根据数据包的类型，调用 tcp_rcv_state_process 函数确定 TCP 连接状态应如何切换，数据包应如何处理。各状态下的数据包处理过程大部分都在 tcp_rcv_state_process 函数中完成，除 ESTABLISHED 和 TIME_WAIT 这两个状态以外。

如果数据包到达，TCP 连接为 CLOSED 状态，则扔掉数据包。

1. LISTEN 到 SYN_RECV：服务器收到连接请求

```
int tcp_rcv_state_process(struct sock *sk, struct sk_buff *skb,
                struct tcphdr *th, unsigned len)                        //net/ipv4/tcp_input.c
{
…

//从套节字数据结构 sk 获取当前 TCP 状态，如果为 CLOSED，则扔掉数据包；如果为 LISTEN，收到 ACK 返回 1，收到 RST 扔
//掉数据包，收到 SYN 切换到 SYN_RECV

   switch (sk->sk_state) {
   case TCP_CLOSE:
         goto discard;
   case TCP_LISTEN:
         if (th->ack)
               return 1;
         if (th->rst)
               goto discard;
         if (th->syn) {
//处理连接请求
               if (icsk->icsk_af_ops->conn_request(sk, skb) < 0)
                     return 1;
               kfree_skb(skb);
               return 0;
         }
         goto discard;
```

当前 TCP 套接字所在状态是 LISTEN，说明这个套接字是一个服务器（server），它在等待一个连接请求。这时 TCP 协议发送的标志为判别输入数据包类型。

- ACK：发送连接复位。
- RST：连接由客户端复位，扔掉数据包。
- SYN：客户端来发送的连接请求，调用 icsk_af_ops->conn_request 函数完成连接请求处理。将 TCP 连接状态切换至 SYN_RECV。icsk_af_ops->conn_request 函数指针在 TCP 协议实例中初始化为 tcp_v4_conn_request。
- 其他数据包：当前连接还未建立，扔掉。

在函数源代码中有一段注释讨论了，关于我们是否应该处理由非 IP 协议上传到 TCP 的输入数据包。当前版本的内核是扔掉数据包。

在 tcp_v4_conn_request（定义在 net/ipv4/tcp_ipv4.c 文件中）函数中将初始化序列号，发送一个有 SYN 和 ACK 标志的回答数据段，并将 TCP 连接状态设置为 TCP_SYN_RECV。

2. 从 SYN_SENT 到 ESTABLISHED：客户端等待连接回答

如果当前套接字状态是 SYN_SENT，说明套接字为客户端，它发送了一个 SYN 数据包请求连接，并将自己设置为 SYN_SENT 状态。这时我们必须查看输入数据段中的 ACK 或 SYN 标志，以确定是否将状态转换到 ESTABLISHED。由 tcp_rcv_synsent_state_process 函数完成处理 SYN_SENT 状态的工作。

```
case TCP_SYN_SENT:
        queued = tcp_rcv_synsent_state_process(sk, skb, th, len);
        if (queued >= 0)
              return queued;
        tcp_urg(sk, skb, th);
        __kfree_skb(skb);
```

```
        tcp_data_snd_check(sk);
        return 0;
    }
```

（1）收到 ACK，数据包合法

tcp_rcv_synsent_state_process 函数会对数据包和 TCP 的协议头标志进行检验，如果接收到的数据包合法且设置了正确的 ACK 标志，则 tcp_rcv_synsent_state_process 函数将 TCP 状态切换到 ESTABLISHED 状态。

```
tcp_set_state(sk, TCP_ESTABLISHED);
security_inet_conn_established(sk, skb)
```

（2）收到连接复位

如果收到连接复位请求，则复位连接并扔掉数据包。

```
if (th->rst) {
            tcp_reset(sk);
            goto discard;
        }
```

（3）收到 SYN

如果收到 SYN，而没有收到服务器序列号，则扔掉数据包。

```
if (!th->syn)
  goto discard_and_undo;
```

（4）tcp_rcv_synsent_state_process 函数返回负值

tcp_rcv_synsent_state_process 函数可以返回一个负值，即输入数据段中有数据等待处理，除了查看 URG 标志外，我们不对数据做任何处理。查看有无 pending 的数据段，如果有，则发送出去。

```
res = tcp_validate_incoming(sk, skb, th, 0);
  if (res <= 0)
        return -res;
```

随后如果函数 tcp_rcv_state_process 运行到这里，即 TCP 连接不是处于 SYN_SENT、LISTEN 和 CLOSE 状态，接下来按照 RFC793 规范，依次处理。

3. 数据包有效性检查

数据包有效性检查由函数 tcp_validate_incoming 完成，在 tcp_validate_incoming 函数中检查的内容按照 RFC793 规范进行。

第一步，序列号有效。

tcp_sequence 查看序列号是否在窗口范围内，如果它超出了当前窗口范围，则 tcp_sequence 函数返回 0。

如果收到的是一个错误序列号，则调用 tcp_send_dupack 函数并扔掉输入数据段。如果输入数据包中包含了一个复位标志 RST，则直接扔掉数据段并返回。

```
if (!tcp_sequence(tp, TCP_SKB_CB(skb)->seq, TCP_SKB_CB(skb)->end_seq))
{
        if (!th->rst)
              tcp_send_dupack(sk, skb);
        goto discard;
  }
```

第二步：如果数据包中有复位连接标志 RST，则复位连接，并扔掉数据包。

在确定序列号在窗口范围内后，struct tcp_sock 的 tp->rx_opt.ts_recent 数据域为最近接收时间戳。

第三步：安全检查和优先级忽略。

第四步：收到的数据包如果是 SYN，查看其序列号是否在当前窗口范围内。如果收到的 SYN 在当前窗口范围内，这是一个错误条件，连接应复位。

```
if ( th->syn && !before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt))
   tcp_reset(sk);
```

当 tcp_validate_incoming 函数检查完成时，如果返回值正确则说明当前收到的是有效数据包。

4. 数据包有效：有 ACK 标志

第五步：收到的数据包 skb 中有 ACK 标志。

这一步处理非常复杂，其基本处理是：如果回答是可接收的数据包，则将 TCP 连接状态转换到 ESTABLISHED 状态。

```
//如果数据包设有 ACK 标志，FLAG_SLOWPATH 参数告诉 tcp_ack 在窗口更新时立即做进一步检查，如果 ACK 数据段正确可接
//收返回 1
if (th->ack) {
        int acceptable = tcp_ack(sk, skb, FLAG_SLOWPATH);

        //如果收到的 ACK 数据正确，则说明连接正处于 SYN_RECV 状态，这时我们最大的可能是处于“被迫打开”的状态，
        //应将状态切换到 ESTABLISH 状态。

        switch (sk->sk_state) {
        case TCP_SYN_RECV:
             if (acceptable) {
                  tp->copied_seq = tp->rcv_nxt;
                  smp_mb();
                  tcp_set_state(sk, TCP_ESTABLISHED);
                  sk->sk_state_change(sk);

//sk_wake_async 唤醒套接字，被迫打开的套接字就不能被唤醒，因为 sk 的套接字数据域为 NULL
if (sk->sk_socket)
  sk_wake_async(sk,
            SOCK_WAKE_IO, POLL_OUT);

//更新 SND_UNA：ACK 序列号，更新 SND_WND：窗口向前移动，tcp_init_wl 更新 snd_wl1 存放输入数据包的序列号
tp->snd_una = TCP_SKB_CB(skb)->ack_seq;
tp->snd_wnd = ntohs(th->window) <<
       tp->rx_opt.snd_wscale;
        tcp_init_wl(tp, TCP_SKB_CB(skb)->ack_seq,
   TCP_SKB_CB(skb)->seq);
```

现在套接字进入到 ESTABLISHED 状态，我们必须让套接字做好接收数据的准备：

首先，tcp_ack 不计算 RTT，所以如果接收到的 ACK 数据包有时间戳选项，RTT 就基于时间戳计算（tcp_ack_saw_tstamp）。RTT 的值保存在 struct tcp_sock 数据结构的 srtt 数据域中。将 MSS 的值调整为可以容纳下时间戳，重建协议头。

tcp_init_metrics 初始化套接字的某些字段并计算。

tp->lsndtime = tcp_time_stamp 防止假的阻塞窗口在第一个数据包上重启。

tcp_initialize_rcv_mss 接收到的 MSS 值依据收到的窗口大小 RCV.WND 初始化为一个猜测值。套接字上的预留缓冲区的空间基于收到的 MSS 的值和其他因素来确定。

tcp_init_buffer_space 为套接字预留缓冲区空间。

最后，tcp_fast_path_on 计算 struct tcp_sock 数据结构上的 pred_flags 数据域，该数据域确定接收到的数据包是否应交给“Fast Path”处理，是否使用协议头预定向。我们转入“ Fast Path”，因为我们将进入 ESTABLISHED 状态。

5. 收到的数据包无效

如果输入的数据包是一个不可接受的回答数据包，则返回 1，告诉调用者（tcp_v4_do_rcv）发送复位。

6. FIN_WAIT_1 状态处理

（1）由 FIN_WAIT_1 切换到 FIN_WAIT_2

当前 TCP 连接状态为 FIN_WAIT_1，如果收到一个 ACK 数据包，则将 TCP 连接状态切换到 FIN_WAIT_2；同时设置套接字的 shutdown 数据域的值为 SEND_SHUTDOWN，指明随后在套接字切换成 CLOSED 状态时应向站点发送包含 RST 的数据包 shutdown。

如果套接字不是一个死套接字（SOCK_DEAD），则唤醒套接字并将套接字切换到 FIN_WAIT_2 状态。

```
sk->sk_state_change(sk);
```

（2）处理 TCP 选项

- TCP_LINGER2 选项：决定 TCP 套接字在进入 CLOSED 状态前，需要在 FIN_WAIT_2 状态上等待多长时间，其值存放在 struct tcp_sock tp->linger2 数据域中。如果 linger2 的值为负，则套接字立即切换到 CLOSED 状态，期间不经过 FIN_WAIT_2 和 TIME_WAIT 状态。
- keepalive 选项：由 cp_fin_time 函数查看 keepalive 选项 SO_LINGER。它依据选项的值计算套接字在 FIN_WAIT_2 状态上等待的时间，将默认时间设置为与重传时间相同的值。keepalive 时钟在超时的情况下被复位(inet_csk_reset_keepalive_timer)。
- 如果收到的 ACK 数据包是最后一个回答 FIN，或套接字被其他进程锁定，则复位 keepalive 时钟，如果不这样做，就可能丢失输入的 FIN。

7. CLOSING 和 LAST_ACK

（1）CLOSING

收到了 ACK 后套接字直接进入到 TIME_WAIT 状态，说明在连接的发送端没有其他等待向外发送的数据了。

（2）LAST_ACK

如果套接字被迫关闭，则响应应用程序的 close 调用。在这个状态上接收到 ACK 意味着可以关闭套接字，所以调用 tcp_done 函数。到此我们已处理完输入的 ACK 数据段。

8. 紧急数据处理

第六步：处理紧急请求。这时我们查看输入数据段的 UGR 位，这项检测由 tcp_urg 函数完成，该函数也会继续处理紧急数据。

```
tcp_urg(sk,skb, th)
```

9. 处理段中的数据内容

第七步：处理数据段中的内容。按照 RFC793 规范在以下五种状态时都应将接到的数据段放入队列中：TCP_CLOSE_WAIT、TCP_CLOSING、TCP_LAST_ACK、TCP_FIN_WAIT1、TCP_FIN_WAIT2。按照 RFC1122 规范应该发送 RST，BSD 操作系统从 4.4 版本以后才这样做，Linux 也发送一个复位。

10. ESTABLISHED 状态处理

以下就是套接字为 ESTABLISHED 状态时收到常规数据段的处理，它调用 tcp_data_queue 函数把数据段放入套接字的输入缓冲队列。

调用 tcp_data_snd_check 和 tcp_ack_snd 函数确定应向另一站点发送的是包含数据的数据段还是 ACK 数据段。

9.6.4 TCP 连接为 ESTABLISHED 状态时的接收处理

TCP 协议的目的是可靠、快速地传送数据。当两个站点的连接为 ESTABLISHED 状态时，表明连接已成功建立，这时开始数据的相互传送。tcp_v4_do_rcv 函数在处理输入数据包时，会查看套接字是否处于 ESTABLISHED 状态，如果是，则调用 tcp_rcv_established 函数来完成具体的数据接收过程。这个函数的主要目的是将数据段中的数据复制到用户地址空间。图 9-16 给出了 TCP 连接为 ESTABLISHED 状态的接收处理过程。

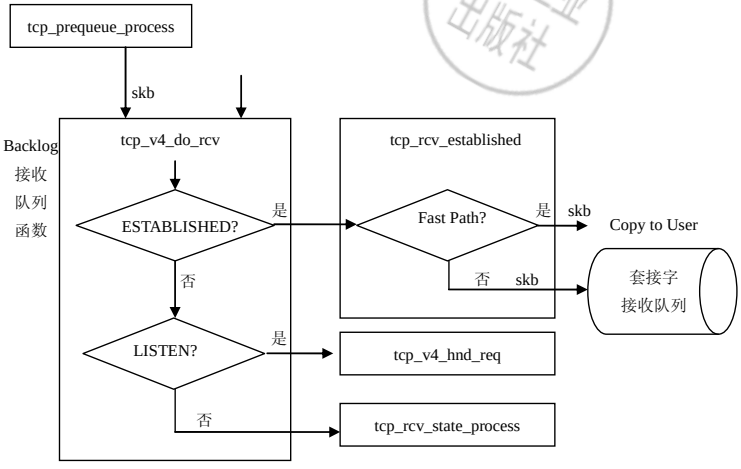


图 9-16 TCP 连接建立后的接收处理过程

1. 数据包进入“Fast Path”路径条件

Linux 内核提供了“Fast Path”处理路径来加速 TCP 数据传送。Linux 使用协议头预定向来选择哪些数据包应放到“Fast Path”路径上处理，其条件为：

- 数据包接收顺序正确。
- 数据包不需要进一步分析，可直接复制到应用程序的接收缓冲区中。

因此，“Fast Path”处理过程是在应用程序进程的现场执行，在应用进程现场执行一个

套接字的读操作函数。

如果所有的 TCP 输入数据包都经“Fast Path”处理，则数据包的处理速度是最快的。正确进入“Fast Path”路径的关键是，选择最有可能包含常规数据的数据包，这样的数据包不需要任何额外的处理。为了完成这个选择，Linux 内核使用协议头预定向处理，在收到数据包之前快速标记这些候选的数据包，协议头预定向处理是计算协议头中的预定向标志，该标志用于定向数据包是由“Fast Path”处理还是由“Slow Path”处理，协议头预定向值由以下公式计算：

prediction flags = hlen << 26 ^ ackw ^ SND.WND。

- hlen：是协议头的长度。
- ackw：是 ACK<< 20 位后的布尔变量。

以上处理公式产生一个预定向标志值，记录在 TCP 协议头的字节 13~20 的位置（按网络字节顺序，TCP 协议头以 32 位字描述）。因此，预定向标志会在输入数据包 TCP 协议头的第三个 32 位字处，该输入数据包应有 ACK 标志，但没有 TCP 选项。

预定向标志存放在 TCP 的 struct tcp_sock 数据结构的 tp->pred_flags 数据域中。预定向标志由 inline 函数__tcp_fast_path_on 计算，定义在 include/net/tcp.h 文件中。

虽然协议头中预定向标志已计算出来，而在 ESTABLISHED 状态中处理输入数据包时，还需做一些别的检查，其中任何一项条件不符合，数据包就送给“Slow Path”处理。

- TCP 连接的接收端宣布接收窗口大小为 0。对 0 窗口的探测处理只能在“Slow Path”上处理。
- 数据包接收顺序不正确，也不会放入“Fast Path”处理。
- 遇到的数据是紧急数据，这时“Fast Path”被禁止；将紧急数据复制到用户地址空间后，“Fast Path”重新打开。
- 如果用户地址空间没有更多的接收缓冲区剩余，禁止“Fast Path”。
- 任何协议头预定向过程失败都会将数据段放入“Slow Path”处理。
- “Fast Path”只支持不会再改向的数据传输，所以如果我们必须把数据传送到别的路径，则将默认的输入数据段放入“Slow Path”处理。
- 如果在输入数据包中除时间戳选项外还有别的选项，则将数据包送到“Slow Path”处理。

由 tcp_rcv_established 函数完成绝大部分对输入数据段的处理，这个函数只会在套接字连接处于 ESTABLISHED 状态时调用。因此它假设数据包在“Fast Path”路径上处理，在处理途中如果该函数发现数据包需要进一步检查，则它会将数据包放入“Slow path”处理。

2. ESTABLISHED 上的处理流程

（1）数据包是否满足“Fast Path”处理条件

```
int tcp_rcv_established(struct sock *sk, struct sk_buff *skb,struct tcphdr *th, unsigned len)
                                                                    //net/ipv4/tcp_input.c
{

数据包是否满足“Fast Path”条件
  a）预定向标志与输入数据段的标志作比较
  b）数据段接收顺序正确
```

```
if ((tcp_flag_word(th) & TCP_HP_BITS) == tp->pred_flags &&
     TCP_SKB_CB(skb)->seq == tp->rcv_nxt) {
        int tcp_header_len = tp->tcp_header_len;
```

c）除时间戳选项外没有别的选项，如果有别的选项，则将该数据段送到“Slow Path”处理，TCP 的 saw_tstamp 选项域指明输入数据包有时间戳选项

```
if (tcp_header_len == sizeof(struct tcphdr) + TCPOLEN_TSTAMP_ALIGNED) {
        if (!tcp_parse_aligned_timestamp(tp, th))
                goto slow_path;
```

d）对数据段做 PAWS 快速检查，如果检查失败，则将数据包放入“Slow Path”处理

```
if ((s32)(tp->rx_opt.rcv_tsval - tp->rx_opt.ts_recent) < 0)
                goto slow_path;
if (len <= tcp_header_len) {
             if (len == tcp_header_len) {
```

e）预期数据包在窗口范围内

```
                if (tcp_header_len ==
                   (sizeof(struct tcphdr) + TCPOLEN_TSTAMP_ALIGNED) &&
                   tp->rcv_nxt == tp->rcv_wup)
                      tcp_store_ts_recent(tp);
```

f）查看协议头的长度是否太小，数据包中无任何数据。如果是说明当前正在处理单向数据传送，即只发送数据，收到的将是给我们发送数据的答复

```
                tcp_ack(sk, skb, 0);
                __kfree_skb(skb);
                tcp_data_snd_check(sk);
                return 0;
             }
        //协议头太小，所以扔掉数据包

} else {
                TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_INERRS);
                goto discard;
             }
```


（2）确定运行现场

“Fast Path”路径运行在应用程序进程现场，现在查看当前应用进程是否为等待数据的进程。

Linux 内核中当前进程的全局指针 current 总是指向当前正在运行的进程，当前运行进程应为：进程状态为 TASK_RUNNING，放在运行进程链表中的第一个进程。

如果查看当前套接字是否运行在当前用户进程现场，则通过比较 ucopy 数据结构中的进程指针是否与当前运行进程相同。

存放在 struct ucopy 数据结构中的当前进程是 tcp_recvmsg，tcp_recvmsg 是套接字层的接收函数，它由应用程序调用（通过 Linux 系统调用接口）。

```
if (tp->ucopy.task == current && sock_owned_by_user(sk) && !copied_early)
 {
   __set_current_state(TASK_RUNNING);
   if (!tcp_copy_to_iovec(sk, skb, tcp_header_len))
         eaten = 1;
}
```

tcp_copy_to_iovec 函数在复制数据时，也完成校验和的计算。如果数据复制成功：

if(eaten){ 代码块中去掉 TCP 的协议头，更新下一个将收到的数据包序列号 rcv_nxt。

（3）复制不成功

if(!eaten)：程序到达这里有两个原因：

- 没有用户进程在运行。
- 向用户空间复制数据失败。

如果向用户空间复制数据失败，可能是因为校验和不正确，则需要重新完成校验和计算。如果校验和仍不正确，则跳出程序。

如果套接字中没有多余空间，则在“Slow Path”路径中完成处理数据复制。

```
if( !eaten) {
…
goto step5;
```

（4）连续大量复制数据

NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPHPHITS

这是大块数据传送的接收端：我们将协议头从 skb 中移走，把数据段中包含的数据部分放入接收队列。

```
__skb_pull(skb,tcp_header_len)
__skb_queue_tail(&sk->sk_receive_queue, skb);
…
```

（5）数据接收后处理

因为现在接收到的是有效的数据包，由以下几个小节的代码处理发送和接收回答数据段。

- tcp_event_data_recv 更新延迟回答时钟超时间隔值。

tcp_ack 处理输入 ACK。

如果不需要发送 ACK，则跳到 no_ack 标签处。

```
if ( !inet_csk_ack_scheduled(sk))
              goto no_ack;
```

- no_ack 标签：if (eaten)表示数据已传送，删除 skb，否则数据未传送。

sk->sk_data_ready(sk, 0)表示调用 data_ready 回调函数指明套接字已准备好为下一次应用读。

（6）“Slow Path”处理

以下代码部分是处理接收数据段的“Slow Path”，这也是函数的结束处理。如果函数是由内核内部的“bottom half”调用的则会进入到这里，或者由于某种原因 prequeue 不能处理数据，也会进到这里。之所以在“Slow Path”路径中处理接收数据包，是因为没有符合“Fast Path”条件的数据段，所以我们沿着 RFC793 规范的要求继续处理。

在 9.6.3 节中已经介绍了，按照 RFC792 规范一共有 7 个步骤。调用 tcp_validate_incomming 函数来查看输入数据包是否为有效数据包完成了前 4 个步骤，接下来按顺序进行余下的 3 步处理。

（7）完成最后处理

第五步：在 ESTABLISHED 状态查看 ACK 标志，发送 ACK。

第六步：处理 URG 标志。

第七步：处理数据段中的数据。

tcp_data_queue 把数据放入套接字的常规接收队列中。它把接收顺序不正确的数据段放入 out_of_order_queue 队列。对于套接字常规接收队列，当应用程序在打开的套接字上执行读系统调用时处理将继续，这时会调用 tcp_recvmsg 函数。

9.6.5 TCP 的 TIME_WAIT 状态处理

从图 9-13 中我们可以看到活动关闭通过 TIME_WAIT 状态完成，TCP 在 TIME_WAIT 状态上等待的时间是数据段最大存活周期（MSL ： maxinum segment lifetime）的两倍，称为 2MSL。

每种系统对 TCP 的实现，都需选择一个 MSL 的值，RFC1122 规范中的建议值是 2 分钟，即 TCP 在 TIME_WAIT 状态上等待的时间为 1～4 分钟。MSL 是任何 IP 数据包可在网络上存活的最长时间。对于 IP 数据包我们知道，在 IP 协议头中有一个 8 位的数据域 TTL，是 IP 数据报可经过的最大跳数 255。TTL 是数据包可经过的最大跳数，不是时间的限制值，结合 TCP 和 IP 可以知道，一个数据包在网络中可路过的最大跳数是 255，但经过的时间不能超过 MSL。

数据包在网络中丢失常常是因为路由异常引起的，如一个路由口崩溃了或两个路由器之间的链路下线。路由协议需要花几秒或几分钟的时间来建立稳定路由或寻找一条新的替换路由，在此期间可能发生路由死循环（路由器 A 发送数据包给路由器 B，路由器 B 又将它们送回给路由器 A）。同时假设丢失的数据包是 TCP 数据段，TCP 发送超时后会重传同一数据包。重传数据包通过替代路由到达目标地址。随后路由死循环结束，路由异常修复后原路由恢复正常，在以上死循环路由上的数据包最后也送达目标地址，这些原数据包称为丢失复制数据段（lost duplicate）或迷失复制段（wandering duplicate），TCP 必须处理这些复制段。

TCP 连接需要维护 TIME_WAIT 状态有两个原因：

- 当结束 TCP 连接时，使连接两端可靠结束。
- 让网络中旧的复制段超时。

第一个原因我们可以以图 9-14 为例来说明，假设最后一个 ACK 数据段丢失，服务器会重传结束连接的 FIN 数据段，所以客户端必须维护状态信息，以便重传最后的 ACK 数据段。如果客户端没有维护自己的状态信息，它会发送 RST 数据段来回答服务器的 FIN 数据段，服务器接收到客户端的 RST 回答数据段会将其作为错误数据包处理。如果 TCP 在终止一个连接时，TCP 要完成所有需要的数据流，才能使断开连接结束得干净。它必须正确处理终止连接的 4 个数据段丢失的情况。这也说明了为什么执行活动关闭时 TCP 仍要在 TIME_WAIT 状态上等待，因为连接的一端可能会重传最后一个 ACK 数据段。

现在假设在 192.168.11.254 端口 5001 与 206.168.111.210 端口 21 之间有一个 TCP 连接。该连接在数据传送后关闭，一段时间后，我们在同一地址两端建立一个新连接，后一个连接称为前一个连接的化身（incarnation），因为它们的地址完全相同。TCP 必须阻止前一个连接上的旧复制段出现在新的连接化身上。为了处理这种状况，TCP 在连接还在

TIME_WAIT 状态上时不会初始化新的连接化身。因为 TCP 在 TIME_WAIT 状态上等待的时间是 MSL 的两倍，其可以在一个 MSL 期间丢失数据包，在另一个 MSL 期间扔掉回答。为了强制执行这条规则，我们保证成功建立了一个 TCP 连接后，网络中所有前一个连接的复制段都超时。

由此可知一旦 TCP 连接的一端接收到关闭连接的要求，它必须进入 TIME_WAIT 状态等待一段时间（2*MSL），最后才终止连接。这一节我们就描述当 TCP 连接为 TIME_WAIT 状态时如何处理输入数据包。

在这里需要区分两个重要的概念，关闭连接与连接被迫关闭。关闭连接来自于连接一端发出一个明确关闭连接要求；但连接被迫关闭是因为 TCP 收到一个带 FIN 标志的 TCP 数据段。所以将 TCP 连接设置为 TIME_WAIT 状态可以达到以下目的：

- 防止已关闭连接上的数据段与新连接数据段混淆。如果一个已关闭连接上的数据还在网络上游走，而与前次连接地址完全相同的一个新连接开始。
- 使连接保持一段时间，这段时间大于数据包重传所需的最长时间，在这段时间里即使最后一个 ACK 数据包丢失，也有足够的时间重传最后一个 ACK（或 ACK 与数据段的组合）。

在一个大型服务器中，TCP/IP 协议栈要解决的一个重大问题就是，需要维护大量的套接字在 TIME_WAIT 状态所耗费的大量内存资源。虽然这对于嵌入式系统而言不是一个主要问题，但 Linux 内核中对 TIME_WAIT 状态的实现，展示了 Linux TCP/IP 协议栈的体系结构是如何满足为服务器维护成百或上千个打开连接的需要的。

在 Linux TCP/IP 协议栈中记录 TCP 连接、连接状态、接收缓冲区的是 struct sock 数据结构；struct sock 数据结构的每个实例都需要分配内存，其中包括 struct sock 数据结构自己和由 sock 数据结构管理的数据缓冲区。在每一个服务器上都有大量的连接在不停地打开和关闭，如果仍用 sturct sock 数据结构来维护这些等待关闭连接的套接字则开销非常大。所以 Linux 内核使用了另一个数据结构来管理 TCP 连接的 TIME_WAIT 状态。

1. struct inet_timewait_sock 数据结构

为了降低对内存的消耗，对 TIME_WAIT 状态处理不使用 struct sock 数据结构。它使用一个更小的数据结构，而且在数据结构后不跟接收缓冲区。这个数据结构为 struct inet_timewait_sock，定义在 include/net/inet_timewait_sock.h 文件中。struct inet_timewait_sock 与 struct sock 数据结构共享前 16 个数据域，所以它们可以使用同一个链表维护指针和函数。

```
struct inet_timewait_socl {
/*与 struct sock 数据结构共享的部分*/

struct sock_common __tw_common;
#define tw_family      __tw_common.skc_family
#define tw_state         __tw_common.skc_state
#define tw_reuse         __tw_common.skc_reuse
…

/*子状态 substate 中存放的是处理被迫关闭过程的状态 FIN_WAIT_1、FIN_WAIT_2、CLOSING 和 TIME_WAIT。*/

volatile unsigned char   tw_substate;
unsigned char            tw_rcv_wscale;
```

```
/*以下数据域用于套接字对输入数据包的多路比较。这5个数据域也在struct inet_sock数据结构中*/

__u16               tw_sport;
…
__u16               tw_num;

/*以下数据域只用于inet_timewait_sock*/

__u8                tw_ipv6only:1,
                    tw_transparent:1;
__u16               tw_ipv6_offset;
unsigned long       tw_ttd;
struct inet_bind_bucket *tw_tb;
struct hlist_node        tw_death_node;
}
```

2. tcp_timewait_state_process 函数

由 tcp_timewait_state_process 函数完成大部分 TCP 连接在 TIME_WAIT 状态时对输入数据包的处理。它也管理连接在 FIN_WAIT_2 状态时对输入数据包的处理，FIN_WAIT_2 是 TCP 状态机中套接字关闭活动 TIME_WAIT 状态前的一个 TCP 状态。tcp_timewait_state_process 函数返回值是一个联合（union）类型的数据结构：enum_timewait_state_process，其值如表 9-6 所示。如果仔细分析 tcp_timewait_state_process 函数，可以看到它重复了 tcp_rcv_state_process 函数的大部分工作，但在 TIME_WAIT 中是简略形式。

表 9-6　TCP TIME_WAIT 状态返回值

符号	值	描述
TCP_TW_SUCCESS	0	延迟段或 ACK 的复制，扔掉数据包
TCP_TW_RST	1	接收到 FIN，发送 RST 给对方站点
TCP_TW_ACK	2	接收到最后一个 ACK，返回 ACK 组给对方站点
TCP_TW_SYN	3	接收到 AYN，重新打开连接

```
enum tcp_tw_status
tcp_timewait_state_process(struct inet_timewait_sock *tw, struct sk_buff *skb,
                const struct tcphdr *th)
{
  struct tcp_timewait_sock *tcptw = tcp_twsk((struct sock *)tw);
  struct tcp_options_received tmp_opt;
  int paws_reject = 0;

  tmp_opt.saw_tstamp = 0;

//首先查看输入数据包是否有时间戳，如果有，则对一个接收顺序不正确的数据段做PAWS检查
  if (th->doff > (sizeof(*th) >> 2) && tcptw->tw_ts_recent_stamp) {
        tcp_parse_options(skb, &tmp_opt, 0);

        if (tmp_opt.saw_tstamp) {
              tmp_opt.ts_recent      = tcptw->tw_ts_recent;
              tmp_opt.ts_recent_stamp     = tcptw->tw_ts_recent_stamp;
              paws_reject = tcp_paws_check(&tmp_opt, th->rst);
        }
  }
```

```
   //与 tcp_rcv_state_process 类似，查看当前套接字是否在 FIN_WAIT2 状态，如果是则需查看输入数据段是否有 FIN、ACK
   //标志或它是一个接收顺序不正确的数据段，如果前面的 PAWS 检查说明输入段是接收顺序不正确的数据段，则必须传送一个 ACK
     if (tw->tw_substate == TCP_FIN_WAIT2) {

         //在 RFC793 规范中要求在 TIME_WAIT 状态下如果收到一个接收顺序不正确的数据段，函数应向调用者返回
TCPTW_ACK，告诉调用程序发送 ACK

           if (paws_reject ||
              !tcp_in_window(TCP_SKB_CB(skb)->seq, TCP_SKB_CB(skb)->end_seq,
                        tcptw->tw_rcv_nxt,
                        tcptw->tw_rcv_nxt + tcptw->tw_rcv_wnd))
                 return TCP_TW_ACK;
           //如果输入段中设置了 RST 标志，就可以彻底结束该连接
           if (th->rst)
                 goto kill;

           //如果收到的是 SYN 数据段，但是一个“旧”数据段或其序列号在窗口范围外，则发送 RST 关闭连接
           if (th->syn && !before(TCP_SKB_CB(skb)->seq, tcptw->tw_rcv_nxt))
                 goto kill_with_rst;
           //查看输入数据段是否为一个 ACK 段的复制口，如果是，则扔掉数据段

           if (!after(TCP_SKB_CB(skb)->end_seq, tcptw->tw_rcv_nxt) ||
              TCP_SKB_CB(skb)->end_seq == TCP_SKB_CB(skb)->seq) {
                 inet_twsk_put(tw);
                 return TCP_TW_SUCCESS;
           }

           //如果输入的数据段中包含有新数据，必须向远端发送一个复位关闭连接，停止 TIME_WAIT 时钟
           if (!th->fin ||
              TCP_SKB_CB(skb)->end_seq != tcptw->tw_rcv_nxt + 1) {
   kill_with_rst:
                 inet_twsk_deschedule(tw, &tcp_death_row);
                 inet_twsk_put(tw);
                 return TCP_TW_RST;
           }

   //如果输入数据段为 FIN 段，但连接状态仍为 FIN_WAIT_2，则将连接切换到实际 TIME_WAIT 状态，同时处理接收时间戳，将
   //接收时间存放在 tw_ts_recent 数据域中，它标志数据到达的时间
   tw->tw_substate=TCP_TIME_WAIT;
           tcptw->tw_rcv_nxt=TCP_SKB_CB(skb)->end_seq;
           if (tmp_opt.saw_tstamp) {
                 tcptw->tw_ts_recent_stamp = get_seconds();
                 tcptw->tw_ts_recent     = tmp_opt.rcv_tsval;
           }
   //inet_twsk_schedule 函数管理时钟，它确定连接还会在 TIME_WAIT 状态上等多长时间。按照规范，该时间值应是数据段
   //生命周期的两倍，但如果可能 Linux 会基于重传超时时间减少在 TIME_WAIT 状态上的等待时间
                 if (tw->tw_family == AF_INET &&
              tcp_death_row.sysctl_tw_recycle && tcptw->tw_ts_recent_stamp &&
              tcp_v4_tw_remember_stamp(tw))
                 inet_twsk_schedule(tw, &tcp_death_row, tw->tw_timeout,
                              TCP_TIMEWAIT_LEN);
           else
                 inet_twsk_schedule(tw, &tcp_death_row, TCP_TIMEWAIT_LEN,
                              TCP_TIMEWAIT_LEN);
           return TCP_TW_ACK;
```

```
    }
          //以下进入实际的 TIME_WAIT 状态，如果连接在 TIME_WAIT 状态时收到 SYN，则重新打开连接，但为新连接分配
          //的初始序列号必须大于前一个连接的序列号的//最大值。如果收到的 SYN 是前一个连接旧 SYN 段的复制，则连
          //接必须返回 TIME_WAIT 状态
          if (!paws_reject &&
       (TCP_SKB_CB(skb)->seq == tcptw->tw_rcv_nxt &&
        (TCP_SKB_CB(skb)->seq == TCP_SKB_CB(skb)->end_seq || th->rst))) {

         //paws_reject 为非零值指明输入数据段在窗口范围内，因些它是 RST 或 ACK 段，但//不包含数据
          if (th->rst) {
                //这时输入 RST 可能导致 TIME_WAIT 状态被暗中结束（TWA）。系统调用 sysctl_tcp_rfc1887 可以
                //阻止 TWA，缺省的 Linux TCP 不阻止 TWA，所以如果系统调用没有设置防止 TWA，则结束所有连接
                if (sysctl_tcp_rfc1337 == 0) {
   kill:
                     inet_twsk_deschedule(tw, &tcp_death_row);
                     inet_twsk_put(tw);
                     return TCP_TW_SUCCESS;
                }
          }
          //如果输入段是一个复制 ACK，则扔掉，调用 inet_twsk_schedule 函数更新时钟
          inet_twsk_schedule(tw, &tcp_death_row, TCP_TIMEWAIT_LEN,
                        TCP_TIMEWAIT_LEN);
          if (tmp_opt.saw_tstamp) {
                tcptw->tw_ts_recent    =tmp_opt.rcv_tsval;
                tcptw->tw_ts_recent_stamp = get_seconds();
          }
          inet_twsk_put(tw);
          return TCP_TW_SUCCESS;
    }
   //如果程序运行到此处，说明 PAWS 测试失败，所以接收到的是一个窗口范围外的段或新 SYN。所有超出窗口范围的段都立即给
//出回答，可接收的 SYN 不能是旧 SYN 段的复制。虽然 PAWS 检查已足够，不需要做序列号检查，但这里仍托管 PAWS 做序列号检查
    if (th->syn && !th->rst && !th->ack && !paws_reject &&
       (after(TCP_SKB_CB(skb)->seq, tcptw->tw_rcv_nxt) ||
        (tmp_opt.saw_tstamp &&
         (s32)(tcptw->tw_ts_recent - tmp_opt.rcv_tsval) < 0))) {
          u32 isn = tcptw->tw_snd_nxt + 65535 + 2;
          if (isn == 0)
                isn++;
          TCP_SKB_CB(skb)->when = isn;
          return TCP_TW_SYN;
    }
    if (paws_reject)
          NET_INC_STATS_BH(twsk_net(tw), LINUX_MIB_PAWSESTABREJECTED);
    if (!th->rst) {
          //只在输入段是 ACK 段或不在窗口范围内时，复位 TIME_WAIT 状态时钟
          if (paws_reject || th->ack)
                inet_twsk_schedule(tw, &tcp_death_row, TCP_TIMEWAIT_LEN,
                                   TCP_TIMEWAIT_LEN);
          //返回 TCP_TW_ACK，告诉函数调用者对接收到不正确的数据段发送回答
          return TCP_TW_ACK;
    }
     //接收到 RST，结束连接
    inet_twsk_put(tw);
    return TCP_TW_SUCCESS;
   }
```

图 9-17 总结了 TCP 层数据段接收/传送的路径。

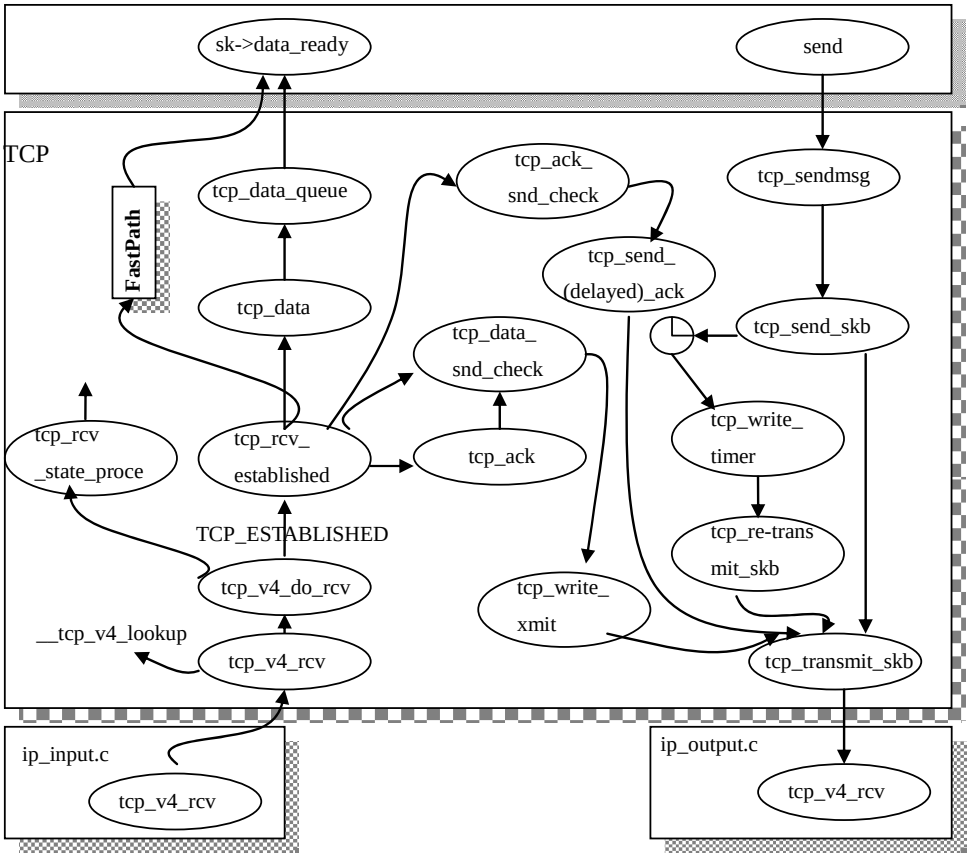


图 9-17　TCP 层数据段接收/发送的路径

9.7　本章总结

TCP 协议是 TCP/IP 协议栈在传输层实现的可靠、面向连接的数据传输服务，本章我们分析了 TCP 协议的主要功能特点，如数据格式、TCP 连接管理机制。沿着数据传送路径分析了内核实现 TCP 协议实例的以下特点：

- TCP 协议实例在内核中的数据结构描述。
- TCP 协议与网络层、套接字层之间的接口数据结构设计、初始化过程。
- TCP 协议数据传送/接收主要功能函数实现的流程、源代码分析。
- TCP 协议连接状态管理主要功能函数实现的流程、源代码分析。

第 10 章　套接字层实现

本章主要介绍网络应用程序与 TCP/IP 协议栈之间的接口——套接字（Socket）在 Linux 内核中的设计与实现。在讲解了套接字的基本概念的基础上，措述了 Linux 管理套接字的数据结构、套接字与应用层、传输层之间的接口。重点分析了套接字在应用程序与内核之间数据交换的流程实现。

Linux 内核实现的套接字接口，将 UNIX 的“一切都是文件操作”的概念应用于网络连接访问上，使应用程序可以用常规文件操作 API 访问网络连接。套接字接口最初是 BSD 操作系统的一部分，在应用层与 TCP/IP 协议栈之间接供了一套标准的独立于协议的接口。从 TCP/IP 协议栈的角度来看，传输层以上的都是应用程序的一部分。Linux 与传统的 UNIX 类似，TCP/IP 协议栈驻留在内核中，与内核的其他组件共享内存。传输层以上执行的网络功能都是在用户地址空间完成的。Linux 使用内核套接字概念与用户空间套接字通信，这样使实现和操作更简单。Linux 提供了一套 API 和套接字数据结构，这些服务向下与内核接口，向上与用户空间接口，应用程序使用这一套 API 访问内核中的网络功能。

10.1　套接字概述

Linux 创建的初始目的是开发一套在功能上与 Unix 系统兼容的操作系统。套接字 API 是 UNIX 应用与网络程序间最著名的接口。Linux 也提供了一个套接字层，在传统 UNIX 中实现的几乎所有基于 TCP/IP 的网络应用都成功地移植到了 Linux，这些功能无论是在大型客户-服务器系统中，还是在嵌入式系统中都得到了很好的实现。在这一章中，会介绍 Linux 基于 TCP/IP 协议栈实现的套接字的设计结构、套接字层与应用层、套接字层与 TCP/IP 协议栈之间 API 的实现。

Linux 套接字 API 适合所有的应用标准，现在的应用层协议也全部移植到了 Linux 系统中。但在套接字层下的基础体系结构实现却是 Linux 系统独有的，在这一章中我们将讨论 netlink 套接字、应用程序人员使用套接字的各种方式，以及通过套接字来与 TCP/IP 协议栈、其他协议交互。我们将讨论套接字的定义，详细讨论套接字的数据结构 struct sock、套接字 API、套接字与文件系统之间的所有调用映射关系。除此之外，还将介绍增加新的网络协议如何使用套接字，应用程序如何使用 netlink 机制来控制 TCP/IP 内部协议的操作。Linux 内核支持的套接字结构如图 10-1 所示。

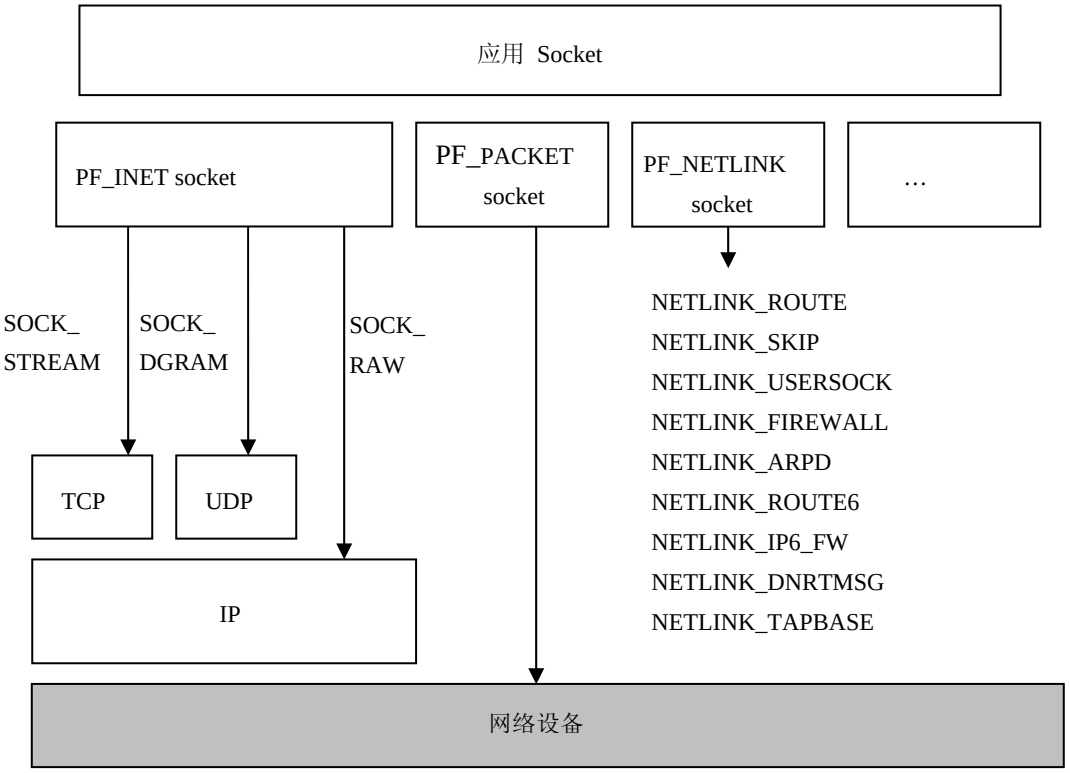


图 10-1　Linux 内核中支持的套接字结构

10.1.1　什么是套接字

套接字接口的一种定义为：它是 TCP/IP 协议栈中传输层协议的接口，也是传输层以上所有协议的实现。同时套接字接口在网络程序功能中是内核与应用层之间的接口。TCP/IP 协议栈的所有数据和控制功能都来自于套接字接口，与 OSI 网络分层模型相比较，TCP/IP 协议栈本身在传输层以上就不包含任何其他协议，Linux 替代传输层以上协议实体的是一个标准接口，称为套接字，它实现传输层以上所有的功能。可以说套接字是 TCP/IP 协议栈对外的窗口。现代的操作系统，包括 Linux 在内，在与 TCP/IP 协议栈交互时，套接字接口是应用程序利用 TCP/IP 协议栈网络协议的唯一途径。

1. 套接字接口的基本功能

套接字有 3 个基本功能：传输数据、为 TCP 管理连接、控制或调节 TCP/IP 协议栈的操作。套接字接口设计得既简单又好用，这也是为什么 TCP/IP 协议栈获得广泛应用的原因。套接字是一个通用的接口，它不仅适用于 TCP/IP 协议栈的协议，还可以用于 Linux 内部进程与进程之间的通信，如使用 AF_UNIX 类套接字。Linux 套接字支持的协议族类型如表 10-1 所示。Linux 套接字支持的地址族的宏定义在 include/linux/socket.h 文件中。

表 10-1　Linux 套接字支持的协议族

名称	值	协议
AF_UNSPEC	0	无特定地址族
AF_UNIX	1	UNIX 域套接字
AF_LOCAL	1	AF_UNIX 地址族的 POSIX 名
AF_INET	2	Internet，TCP/IP 协议族
AF_AX25	3	AX.25 协议
AF_IPX	4	Novell IPX 协议套接字
AF_APPLETALK	5	AppleTalk DDP 套接字
AF_NETROM	6	NET/ROM 套接字
AF_BRIDGE	7	多协议桥套接字
AF_ATMPVC	8	ATM PVCs 套接字
AF_X25	9	为 X.25 套接字预留
AF_INET6	10	IPv6 套接字
AF_ROSE	11	X.25 PLP 套接字
AF_DECnet	12	为 DECnet 预留的套接字
AF_NETBEUI	13	为 802.2LLC 预留的套接字
AF_SECURITY	14	安全回调 pseudo 地址族
AF_KEY	15	PF_KEY 关键管理 API 套接字
AF_NETLINK	16	Netlink 协议栈，在应用层与内核之间传递信息
AF_ROUTE	16	仿 4.4BSD 路由套接字的别名，与 Linux 中的 AF_NETLINK 一样
AF_PACKET	17	Packet 族套接字
AF_ASH	18	Ash 套接字
AF_ECONET	19	Acorn Econet 套接字
AF_ATMSVC	20	ATM SVC 套接字
AF_SNA	22	Linux SNA SVC 套接字
AF_IRDA	23	IRDA 套接字
AF_PPPOX	24	PPPoX 套接字
AF_WANPIPE	25	Wanpipe API 套接字
AF_LLC	26	Linux LLC 套接字
AF_CAN	29	Controller Area Network （CAN）套接字
AF_TIPC	30	TIPC 套接字
AF_BLUETOOTH	31	Bluetooth 套接字
AF_IUCV	32	IUCV 套接字
AF_RXRPC	33	RxRPC 套接字
AF_ISDN	34	mISDN 套接字
AF_PHONET	35	Phonet 套接字
AF_MAX	32	目前未用

2. 套接字层 API 构成

套接字 API 实际上有两个部分：一部分由网络的功能组成（由 struct prot 数据结构描述）；另一部分包含了一种方式，将网络的操作映射到 Linux 常规 I/O 操作上，这样应用程序人员就可以如调用普通文件 I/O 操作一样使用 TCP/IP 来传送和接收数据（struct proto_ops 数据结构描述）。例如，相对于普通打开文件（open）功能，在套接字接口上就是打开一个套接字，一旦套接字打开，普通的 I/O，读/写调用就可以通过打开的套接字移动数据。

10.1.2　套接字与管理套接字的数据结构

在 Linux 中定义了一系列的数据结构来描述套接字本身、套接字传送的数据格式、套接字的属性和管理套接字连接状态的数据结构。

管理套接字传送数据的结构是 Socket Buffer，struct sk_buff 数据结构中存放了套接字接收/发送的数据。在发送数据时，Socket Buffer 缓冲区与管理数据结构在套接字层被创建，存放来自应用程序的数据。在接收数据包时，Socket Buffer 在网络设备的驱动程序中创建，存放来自网络的数据。在 TCP/IP 协议栈的其他层也会创建 struct sk_buff 数据结构来存放本地产生的一些数据，如 TCP 发送的 PMTU 探测数据包、零窗口探测数据包等。关于 Socket Buffer 的定义在第 2 章中已做了详细描述。

每个程序使用的套接字都有一个 struct socket 数据结构与 struct sock 数据结构的实例。Linux 内核在套接字层定义了包含套接字通用属性的数据结构 struct socket 与 struct sock，它们独立于具体协议；具体的协议族与协议实例继承通用套接字的属性，加入协议相关属性，形成管理协议本身套接字的结构。例如，在 TCP/IP 协议栈中，INET 协议族、TCP/UDP 类套接字与基础套接字的继承和包含关系如图 10-2 所示。

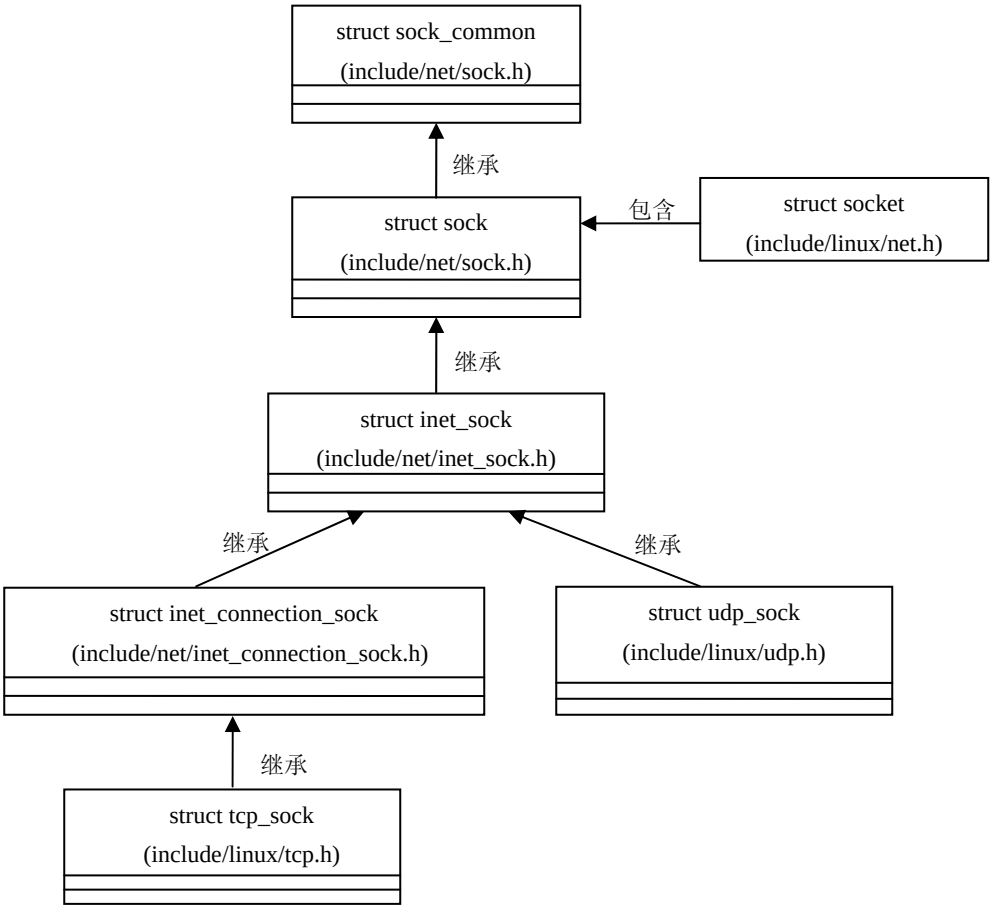


图 10-2　Linux 套接字的继承与包含关系

struct socket *sock 不是特定只为 TCP/IP 服务，该数据结构是一个通用数据结构，其中存放的是套接字层的控制和状态信息。每个打开的套接字都有一个 struct socket 数据结构的实例。struct socket 数据结构中还包含一个文件描述符可由应用程序代码引用，该文件描述符从套接字层返回给应用程序。

1. struct socket 数据结构

```
struct socket {                                                    //include/linux/net.h
  socket_state          state;
  short                 type;
  unsigned long         flags;
  const struct proto_ops *ops;
  struct fasync_struct  *fasync_list;
  struct file      *file;
  struct sock      *sk;
  wait_queue_head_t     wait;
};
```

- State：描述当前套接字状态。其合法取值定义在同一文件 include/linux/net.h 的枚举类型 enum socket_state 中，描述套接字的各种状态如表 10-2 所示。

表 10-2　套接字状态 socket_state 的取值

符号	值	描述
SS_FREE	0	套接字未分配
SS_UNCONNECTED	1	套接字未连接
SS_CONNECTING	2	正在连接
SS_CONNECTED	3	套接字已连接
SS_DISCONNECTING	4	正在断开连接

表 10-2 中所列的值与传输层协议连接的建立与关闭毫无关系，它反映的是内核外部（用户地址空间）的常规套接字状态。

- Type：套接字的类型。其取值为 SOCK_XXXX 形式，Linux 内核支持的套接字类型值定义在 include/linux/net.h 文件的枚举类型 enum socket_type 中，如表 10-3 所示。

表 10-3　Linux 内核支持的套接字类型

符号	值	描述
SOCK_STREAM	1	数据流（面向连接）套接字
SOCK_DGRAM	2	数据包（无连接）套接字
SOCK_RAW	3	裸套接字
SOCK_RDM	4	可靠-传递信息
SOCK_SEQPACKET	5	顺序数据包套接字
SOCK_DCCP	6	数据拥塞控制协议套接字
SOCK_PACKET	10	Linux 中以特定方式从设备层获取数据包

- Flags：套接字的设置标志。存放套接字等待缓冲区的状态信息，其值的形式如 SOCK_ASYNC_NOSPACE 等。套接字的标志值有：

  #define SOCK_ASYNC_NOSPACE　0
  #define SOCK_ASYNC_WAITDATA　1
  #define SOCK_NOSPACE　2
  #define SOCK_PASSCRED　3

#define SOCK_PASSSEC　　4

- *ops：套接字层的操作函数块。在 struct proto_ops 数据结构中定义了一组套接字层的标准操作函数指针供应用程序调用。在 owner 以后的数据域中对应某个对套接字的系统调用函数。

表 10-4　套接字层的操作函数

函数名	描述
family	套接字所属的地址族，如 IPv4，该域设置为 AF_INET
owner	拥有该套接字的模块
release	释放套接字
bind	套接字绑定函数
connect	客户端向服务器发送建立套接字连接请求
socketpair	将两个套接字配对建立连接
accept	服务器端接收套接字连接
getname	获取套接字名称。套接字名称中包含了地址、端口信息
poll	套接字 poll 函数
ioctl	套接字的系统控制函数
compat_ioctl	兼容 ioctl 函数
listen	侦听套接字函数
shutdown	关闭套接字
setsockopt	设置套接字选项
getsockopt	读取套接字选项
compat_setsockopt	设置套接字选项兼容函数
compat_getsockopt	读取套接字选项兼容函数
sendmsg	从套接字发送数据
recvmsg	从套接字接收数据
mmap	实现与文件操作映射的函数
sendpage	传送函数

struct proto_ops 数据结构中的许多函数指针与 C 标准库中对应功能的函数名相同。这并不是巧合，这里的函数直接与 socketcall 系统调用中存放的函数指针一一对应。struct proto_ops 数据结构定义在同一文件 struct socket 数据结构中。

- *fasync_list：等待被唤醒的套接字列表，该链表用于异步文件调用。
- *file：套接字所属的文件描述符，在创建和打开套接字时，套接字层向应用程序返回该文件描述符，应用程序通过文件描述符访问套接字。
- Sk：指向存放套接字属性的结构指针。
- Wait：套接字的等待队列。

2. struct sock 数据结构

在 Linux 内核的早期版本中，struct sock 数据结构非常复杂，从 Linux2.6 版本以后，从两个方面对该数据结构做了优化。其一是将 struct sock 数据结构划分成了两个部分：一部分为描述套接字的共有属性，所有协议族的这些属性都一样；另一部分属性定义在 struct sock_common 数据结构中。其二是为新套接字创建 struct sock 数据结构实例时，从协议特有的缓冲槽中分配内存，不再从通用缓冲槽中分配内存。

struct sock 数据结构包含了大量的内核管理套接字的信息，内核把最重要的成员存放在 struct sock_common 数据结构中，struct sock_common 数据结构嵌入在 struct sock 数据结构

中，是 struct sock 数据结构的第一个成员。

（1）struct sock_common 数据结构

struct sock_common 数据结构是套接字在网络中的最小描述。它包含了内核管理套接字最重要信息的集合。而 struct sock 数据结构中包含了套接字的全部信息与特点，有的特性很少甚至根本就没有用到。

```
struct sock_common {                                          //include/net/sock.h
  unsigned short        skc_family;
  volatile unsigned char skc_state;
  unsigned char         skc_reuse;
  int              skc_bound_dev_if;
  union {
       struct hlist_node      skc_node;
       struct hlist_nulls_node skc_nulls_node;
  };
  struct hlist_node     skc_bind_node;
  atomic_t         skc_refcnt;
  unsigned int          skc_hash;
  struct proto          *skc_prot;
#ifdef CONFIG_NET_NS
  struct net       *skc_net;
#endif
};
```

- skc_family：网络地址家族，如 IPv4 的地址族是 AF_INET。
- skc_state：套接字的连接状态。
- skc_reuse：存放 SO_REUSEADDR 套接字选项，套接字层地址重用。
- skc_bound_dev_if：存放与套接字绑定的网络接口索引号。
- skc_node：主哈希链表，与各协议实例查询表相链接。
- skc_nulls_node： UDP/UDP-Lite 协议的主哈希链表。
- skc_bind_node：与各协议实例查询表绑定的哈希链表。
- skc_refcnt：对套接字的引用计数。
- skc_hash：各协议实例查询哈希链表的关键字。
- skc_prot：网络协议族中协议处理函数块。
- skc_net：该套接字引用的网络名字空间。

（2）struct sock 数据结构

struct sock 数据结构定义了内核管理、控制套接字操作需要的所有信息，包括套接字接收/发送队列、套接字的所有选项设置、套接字操作的回调函数等。其中第一个成员必须是前面描述的 struct sock_common 数据结构。struct sock_common 数据结构必须为 struct sock 第一个成员，是因为在协议实例的数据结构中如 tcp_w_bucket 或其他结构中，struct sock_common 数据结构也是第一个成员，在这些数据结构上，我们可以使用相同的链表处理和队列处理函数。

① struct sock 数据结构的数据域

在 struct sock 数据结构的 struct sock_common 数据成员之后，定义了一些 struct sock_common 数据结构中数据成员的别名，以方便对 struct sock_common 数据结构的访问。

```
#define sk_family       __sk_common.skc_family
```

```
#define sk_state        __sk_common.skc_state
#define sk_reuse        __sk_common.skc_reuse…
```

- sk_shutdown：存放套接字的 RCV_SHUTDOWN 和 SEND_SHUTDOWN 选项。在第 9 章介绍 TCP 协议实例的实现时我们已经看到，当设置了 SEND_SHUTDOWN 选项时，TCP 在关闭套接字时会发送 RST 数据包。
- sk_no_check：包含套接字 SO_NO_CHECK 选项的设置，该选项指明是否禁止校验和。
- sk_userlocks：存放套接字的 SO_SNDBUF 和 SO_RCVBUF 选项的设置。SO_SNDBUF 和 SO_RCVBUF 套接字选项分别用于设置套接字发送缓冲区与接收缓冲区的大小。
- sk_protocol：在某个网络协议族中，该套接字属于哪个协议使用。
- sk_type：套接字类型，其值格式为 SOCK_XXX 形式。例如对于 IPv4，其套接字的类型为 SOCK_STREAM。
- sk_rcvbuf：套接字的接收缓冲区的大小。其度量单位为字节，应用程序可以通过 SO_RCVBUF 套接字选项（保存在 sk_userlocks 数据域中）设置。
- sk_lock：访止对套接字并发访问的锁。
- struct {

```
      struct sk_buff *head;
      struct sk_buff *tail;
} sk_backlog;
```

套接字的 backlog 队列。如在第 9 章 TCP 协议实例数据包接收实现中介绍的接收过程分“Fast Path”和“Slow Path”，快速路径的数据包放在套接字的 prequeue 队列中，慢速路径的数据包放在套接字的 backlog 队列中。

- *sk_sleep：该套接字所在的等待队列。
- *sk_dst_cache：套接字目标地址在路由表高速缓冲区中的入口。
- *sk_policy[2];：用于安全策略库的字段（SPD：Security Policy Database）。
- sk_dst_lock：防止对路由表并发访问的锁。

接下来的 4 个数据域用于描述套接字队列属性的数据域如表 10-5 所示。

表 10-5　描述套接字队列属性的数据域

数据域	说明
sk_rmem_alloc	在接收队列中收到数据包的字节数
sk_wmem_alloc	在发送队列中发送数据包的字节数
sk_omem_alloc	套接字选项或其他设置的长度，以字节计算。“o”代表“option”或“other”
sk_sndbuf	套接字发送缓冲区的大小，其单位以字节数计。由 SO_SNDBUF 套接字选项设置。SO_SNDBUF 选项值保存在 sk_userlocks 数据域中

- sk_receive_queue：套接字的接收队列。将从协议栈上传给应用程序的网络数据包缓冲区放入该队列。
- sk_write_queue：套接字的发送队列。将应用程序写入套接字向外发送的数据包缓冲区放在此队列中。
- sk_async_wait_queue：用直接存储器访问（DMA）方式传送数据包的队列。
- sk_wmem_queued：是永久写队列（persistent）的大小，其度量单位以字节计算。

- sk_forward_alloc：预分配页面的字节数。
- sk_allocation：　内存分配模式标志。
- sk_route_caps：路由权限，即 NETIF_F_TSO。
- sk_gso_type：　GSO 的类型，如 SKB_GOS_TCPV4。
- sk_gso_max_size：可构造的最大 GSO 段的大小。
- sk_rcvlowat：存放套接字 SO_RCVLOWAT 选项的设置，SO_RCVLOWAT 选项用于设置接收数据前的缓冲区内的最小字节数。在 Linux 中，缓冲区内的最小字节数是固定的，为 1。
- sk_flags：包含 SO_BROADCAST、SO_KEEPALIVE 和 SO_OOBINLINE 套接字选项的设置。SO_BROADCAST 选项允许或禁止发送广播数据包。SO_KEEPALIVE 选项用于保持套接字活动，如果协议是 TCP，并且当前的套接字不在侦听或关闭状态，且设置了 SO_KEEPALIVE 选项，就启用保持 TCP 活动定时器。SO_OOBINLINE 选项指明将紧急数据放入普通数据流。
- sk_lingertime：如果设置了 SO_LINGER 套接字选项，sk_lingertime 字段设置为 TRUE。如果设置了 SO_LINGER 选项，则套接字的关闭或下线（shutdown）操作将等到所有套接字里等待发送的消息成功发送或等待延迟时钟超时后才会返回，否则调用立即返回。

通过套接字设置 SO_LINGER 选项的参数是一个 linger 结构，linger 结构组成如下：

```
struct linger{
  int l_onoff;                //套接字延迟打开/关闭状态
  int l_linger;               //延迟时间
};
```


- sk_error_queue：套接字错误信息队列。
- *sk_prot_creator：指向协议处理函数的指针。这些协议处理函数是定义在 AF_INET 协议族或其他协议族内的处理函数块。
- sk_callback_lock：用于防止对 6 个回调函数并发访问的锁。这 6 个套接字的回调函数定义在 struct sock 数据结构的底部。
- sk_err：在该套接字上的最后一个错误。
- sk_err_soft：在套接字上出现的错误。该错误不会引起套接字操作完全失败，只是套接字操作超时。
- sk_drops：裸套接字、UDP 套接字上丢失的数据包计数。
- sk_ack_backlog：当前套接字正在侦听的 backlog 队列。
- sk_max_ack_backlog：在 backlog 队列上设置的可以听的最大进入连接数。
- sk_priority：存放套接字的 SO_PRIORITY 选项设置值。SO_PRIORITY 选项设置套接字上发送数据包的协议指定的优先级，Linux 通过该值来排列网络队列。
- sk_peercred：存放套接字选项 SO_PEERCRED 的设置。该选项只用于 PF_UNIX 套接字，用于设置套接字的进程 ID、用户 ID、组 ID。
- sk_rcvtimeo、sk_sndtimeo：这两个数据域分别存放套接字的 SO_RCVTIMEO、SO_SNDTIMEO 选项设置。
- *sk_filter：套接字过滤指令。

- *sk_protinfo：指向为各种协议使用的私有数据域。
- sk_timer：清除 struct sock 数据结构值的时钟。
- sk_stamp：最近在该套接字上接收到数据包的时间戳。
- *sk_socket：指向存放该套接字的 struct socket 数据结构的指针。
- *sk_user_data：存放远程过程调用（RPC：Remote Procedure Call）层的私有数据。
- *sk_sndmsg_page：缓存发送 message 的页面链表。
- *sk_send_head：传送数据前的信息。
- sk_sndmsg_off：缓存的发送数据在缓存中的偏移量。
- sk_write_pending：一个向 STREAM 类的套接字的写操作挂起了，等待启动。
- sk_mark：常规数据包的标志。

② struct sock 数据结构中的回调函数

以下 6 个数据域指向该套接字的回调函数。

- *sk_state_change：套接字状态改变时的处理函数。
- *sk_data_ready：通知套接字上的数据已准备好，等待用户进程处理。
- *sk_write_space：发送队列中存在有效空间，可以发送新的数据包。
- *sk_error_report：有错误信息到达。
- *sk_backlog_rcv：处理套接字的 backlog 队列的函数。
- *sk_destruct：释放套接字资源的处理函数。

系统中 struct sock 数据结构组织在特定协议的哈希链表中，skc_node 是连接哈希链表中成员的哈希节点，skc_hash 是引用的哈希值。

接收和发送数据放在数据 struct sock 数据结构的两个等待队列中：sk_receive_queue 和 sk_write_queue。这两个队列中包含的都是 Socket Buffer。

内核使用 struct sock 数据结构实例中的回调函数，来获取套接字上某些事件发生的消息或套接字状态发生变化。其中使用最频繁的回调函数是 sk_data_ready，用户进程等待数据到达时会调用该回调函数。

不要将 struct socket 数据结构中 struct proto_ops 类型的数据成员 ops，与 struct sock 数据结构中 struct proto 类型的数据成员混淆。struct proto 数据结构在第 8 章与第 9 章介绍传输层的 TCP/UDP 协议实现中已介绍过，struct proto 数据结构也定义在 include/net/sock.h 文件中。

```
struct proto {                                                    //include/net/sock.h
  void            (*close)(struct sock *sk, long timeout);
  int             (*connect)(struct sock *sk struct sockaddr *uaddr, int addr_len);
  int             (*disconnect)(struct sock *sk, int flags);
  struct sock *         (*accept) (struct sock *sk , int flags , int *err);
  int             (*ioctl)(struct sock *sk, int cmd, unsigned long arg);
  …
```

struct proto_ops 与 struct proto 数据结构的成员的函数名非常类似，但它们代表的是不同的函数。struct proto 数据结构中的函数用于套接字层与传输层之间在内核地址空间的通信。在 struct socket 数据结构中的函数指针块 struct proto_ops 是设计用于与系统调用通信的函数。换言之，struct proto_ops 数据结构中的函数是用户地址空间套接字与内核地址空间套接字的链接。

10.1.3 套接字与文件

一旦套接字的连接建立起来，用户进程可以使用常规文件操作访问套接字，这种方式如何在内核中实现，这要取决于 Linux 虚拟文件系统层（VFS）的实现。

在 VFS 中每个文件都有一个 VFS inode 结构，每个套接字都分配了一个该类型的 inode，套接字中的 inode 指针连接管理常规文件的其他结构。操作文件的函数存放在一个独立的指针表中：

```
struct inode {                          //include/linux/fs.h
…
  struct file_operation *i_fop;         //指向默认文件操作函数块
…
}
```

套接字的文件描述符的文件访问的重定向，对网络协议栈各层是透明的。套接字使用以下文件操作：

```
static const struct file_operations socket_file_ops = {                    //net/socket.c
  .owner =THIS_MODULE,
  .llseek =no_llseek,
  .aio_read =sock_aio_read,
  .aio_write =sock_aio_write,
  .poll =sock_poll,
  .unlocked_ioctl = sock_ioctl,
#ifdef CONFIG_COMPAT
  .compat_ioctl = compat_sock_ioctl,
#endif
  .mmap =sock_mmap,
  .open =sock_no_open,    /*特殊打开代码，不允许通过/proc 文件系统打开*/
  .release = sock_close,
  .fasync =sock_fasync,
  .sendpage =sock_sendpage,
  .splice_write = generic_splice_sendpage,
  .splice_read = sock_splice_read,
};
```


sock_函数是一个包装函数，所有 sock_operations 都会调用另外一个例程。sock_mmap 函数的实现如下：

```
static int sock_mmap(struct file *file, struct vm_area_struct *vma)
{
  struct socket *sock = file->private_data;

  return sock->ops->mmap(file, sock, vma);
}
```

inode 和 socket 的链接是通过直接分配一个辅助数据结构来实现的：

```
struct socket_alloc{
   struct socket socket;
   struct inode vfs_inode;
};
```

内核实现了两个宏来执行将 inode 指针转换为对应的 socket 的实现（SOCKET_I），相反的转换由 SOCK_INODE 宏实现。为了简化操作，无论什么时候，一个套接字与一个文

件相关联时，sock_attach_fd 将 struct file 中的私有数据结构成员 private_data 指向 socket 实例。sock_mmap 给出了以上实现的示例*sock = file->private_data。

10.2　套接字层的初始化

Linux 的网络体系结构可以支持多个协议栈和网络地址类型。内核支持的每一个协议栈都会在套接字层注册一个地址族。这就是为什么在套接字层可以有一个通用的 API 供完全不同的协议栈使用。Linux 内核支持的地址族已在表 10-1 中列出。在本书中我们只讨论 TCP/IP 协议栈，TCP/IP 协议栈在套接字层注册的地址族是 AF_INET。AF_INET 地址族是在内核启动时注册到内核中的，TCP/IP 协议栈与 AF_INET 地址族相连的处理函数，是在套接字初始化时与 AF_INET 地址连接起来；也可以在套接字中动态地注册新的协议栈。在这一节中我们主要分析 AF_INET 地址族的套接字的初始化，同时分析 TCP/IP 协议栈中各协议成员是如何注册到套接字层中的。

就如 Linux 内核中的其他组件一样，套接字层也需提供自身的初始化函数，在内核启动时调用，以初始化套接字的基本属性，注册协议栈的地址族。套接字的初始化函数为 sock_init，该函数必须在 Internet 协议注册前调用，在协议注册前，套接字层的基本初始化必须完成。

1. 套接字层初始化完成的基本任务

套接字层的初始化要为以后各协议初始化 struct sock 数据结构对象、套接字缓冲区 Socket Buffer 对象等做好准备，预留内存空间，套接字层初始化要完成的基本任务如下。

（1）初始化套接字的缓存槽

初始化套接字的缓存槽由 sk_init 函数完成，套接字的缓存槽是套接字层初始化的内存对象，为套接字的数据结构 struct sock 分配内存时，就从该缓存槽中获取内存对象实例。在套接字使用完成，释放 struct sock 时内存对象返回该缓存槽。

（2）为 Socket Buffer 创建内存缓存槽

为 Socket Buffer 创建内存缓存槽由 skb_init 函数完成。Socket Buffer 内存缓存槽为 Socket Buffer 分配内存预先格式化好的内存对象。当内核创建 Socket Buffer 对象时，从该内存缓存槽中获取对象实例，释放的 Socket Buffer 同样返回该内存缓存槽。

（3）创建虚拟文件系统

创建虚拟文件传统的步骤如下：

① 调用 init_inodecache 函数创建文件系统所需的 inode 高速缓冲存储区。在创建套接字文件的 inode 时，从该高速缓存中获取 inode 对象。在 Linux 中与 Unix 操作系统一样，使用 inode 结构作为文件系统实现的基本单元。

② 调用 register_filesystem 函数为套接字创建名为 sock_fs_type 的虚拟文件系统。在 Linux 内实现了统一的 I/O 系统，这样应用程序对 I/O 的调用可以使用统一的 API 传送给它们要访问的设备、文件或套接字。设备、文件或套接字必须注册到文件系统中，以便使用 I/O 系统调用。

③ 在套接字的文件系统创建后，调用 kern_mount 函数来安装套接字的文件系统。

2. 套接字层初始化代码的实现

```
static int __init sock_init(void)                                    //net/socket.c
{
//初始化套接字缓存槽
  sk_init();
//初始化套接字缓冲区 Socket Buffer 内存槽
  skb_init();
//初始化创建套接字文件系统所需的 inode 的缓存
  init_inodecache();
  register_filesystem(&sock_fs_type);
  sock_mnt = kern_mount(&sock_fs_type);
#ifdef CONFIG_NETFILTER
  netfilter_init();
#endif
  return 0;
}
core_initcall(sock_init)
```

sock_init 函数实际会在 main.c 文件中执行 do_initcalls 函数时被调用执行。sock_init 函数由 core_initcall 宏说明，指明了 sock_init 函数在内核启动时执行初始化套接字层，同时 core_initcall 宏也说明了 sock_init 函数在内核启动时被调用的优化级，它会在网络子系统及协议初始化前被调用，执行完成套接字层的初始化。最后 sock_init 函数需要完成的任务是，如果在配置内核时选择了网络过滤功能，则 sock_init 函数调用 netfilter_init 初始化网络过滤子系统。

10.3 地址族的值和协议交换表

套接字是一个通用接口，它可以与多个协议族接口，每个协议族中又可以实现多个协议实例。TCP/IP 协议栈处理完输入数据包后，将数据包交给套接字层，放在套接字的接收缓冲区队列（sk_rcv_queue），数据包从套接字层离开内核，送给应用层等待数据包的用户程序。用户程序向外发送的数据包缓存在套接字的传送缓冲区队列（sk_write_queue），从套接字层进入内核地址空间。

在同一个主机中，可以同时在多个协议上打开多个套接字，来接收和发送网络数据，套接字层必须确定哪个套接字是当前数据包的目标套接字。

10.3.1 协议交换表的数据结构

在 Linux 内核的套接字层，定义了一个数据结构来管理和描述套接字层对应系统调用套接字操作函数块 struct proto_ops，与内核协议相关套接字操作函数块 struct proto 之间的对应关系。该数据结构为 struct inet_protosw。

```
struct inet_protosw {                                    include/net/protocol.h
  struct list_head list;
  unsigned short    type;
  unsigned short    protocol;
  struct proto      *prot;
  const struct proto_ops *ops;
```

```
    int          capability;
    char         no_check;
    unsigned char    flags;
};
```

- type：AF_INET 协议族套接字的类型，如 TCP 协议实例，套接字类型 type=SOCK_STREAM。AF_INET 协议族的所有套接字类型如表 10-1 所示。
- Protocol：协议族中某个协议实例的编号。如 TCP 协议的编码为 IPPROTO_TCP。
- *prot：指向协议实例实现的套接字操作函数块数据结构。TCP 的操作函数如表 9-3 所示。UDP 的操作函数如表 8-1 所示。
- *ops：指向套接字层对应的 socketcall 系统调用，是套接字操作函数块数据结构的指针。所有操作如表 10-4 描述。
- capability：使用该套接字需要的权限，如管理员权限等。
- no_check：对接收/发送数据包是否做校验和。
- flags：该套接字属性的相关标志。其值的定义紧跟在 struct inet_protosw 数据结构的定义之后，flags 标志位有效值的格式为 INET_PROTOSW_XXXX，如表 10-6 所示。

表 10-6　flags 标志位的有效值

符号	值	描述
INET_PROTOSW_REUSE	0x01	端口自动可重用
INET_PROTOSW_PERMANENT	0x02	协议为永久不可卸载的协议
INET_PROTOSW_ICSK	0x04	是否为一个面向连接的套接字

- list：协议族中同一套接字类型 struct inet_protosw 数据结构块的链表，如图 10-3 所示。

内核使用 struct inet_protosw 数据结构实现的协议交换表，将应用程序通过 socketcall 系统调用指定的套接字操作，转换成对某个协议实例实现的套接字操作函数的调用。

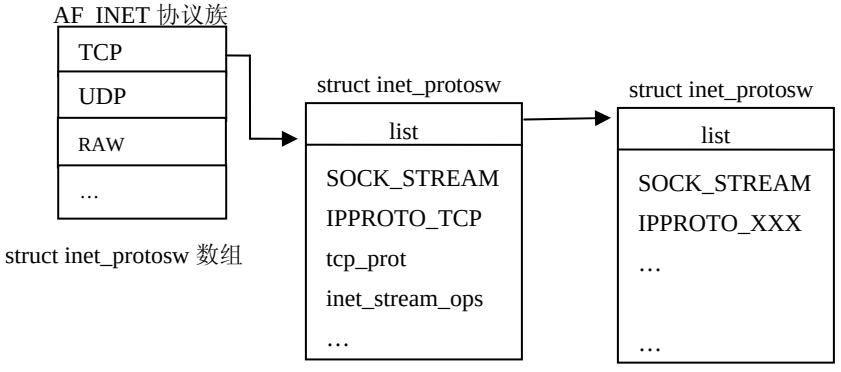


图 10-3　协议交换表队列

10.3.2　套接字支持多协议栈的实现

Linux 套接字的实现独立于具体的网络协议，Linux 套接字可以同时支持多个网络协议栈。定义在 net/ipv4/af_inet.c 文件中的 static struct list_head inetsw[SOCK_MAX] 表中，是协议交换表数组。数组中的每个成员都是一个协议族的交换表，用于存放某个协议族中各

协议实例的套接字系统调用函数与协议套接字函数的对应关系。

在本书中我们只讨论 Linux 套接字对 TCP/IP 协议栈的支持，也就是 Linux 套接字是如何支持 AF_INET 协议族的，AF_INET 协议族的协议交换表是如何注册到内核的。在 Linux 内核中，有的协议是永久协议，即不能从内核中卸载，不能配置为模块的协议实例。例如，TCP、UDP 是 TCP/IP 协议栈的实现基础，就不能从内核中卸载。有的协议可以以模块的方式动态地加载到内核和从内核卸载。针对 AF_INET 协议族，Linux 内核在 net/ipv4/af_inet.c 文件中定义了管理 AF_INET 协议族实例的 struct inet_protosw 数据结构类型的向量数组 inetsw_array[]：

```
static struct inet_protosw inetsw_array[] =                //net/ipv4/af_inet.c
{
  {
        .type =SOCK_STREAM,
        .protocol =IPPROTO_TCP,
        .prot =&tcp_prot,
        .ops =&inet_stream_ops,
        .capability = -1,
        .no_check =0,
        .flags =INET_PROTOSW_PERMANENT |
              INET_PROTOSW_ICSK,
  },

  {
        .type =SOCK_DGRAM,
        .protocol =IPPROTO_UDP,
        .prot =&udp_prot,
        .ops =&inet_dgram_ops,
        .capability = -1,
        .no_check =UDP_CSUM_DEFAULT,
        .flags =INET_PROTOSW_PERMANENT,
     },

     {
        .type =SOCK_RAW,
        .protocol =IPPROTO_IP,             /*通配符*/
        .prot =&raw_prot,
        .ops =&inet_sockraw_ops,
        .capability = CAP_NET_RAW,
        .no_check =UDP_CSUM_DEFAULT,
        .flags =INET_PROTOSW_REUSE,
     }
};
```


从以上定义可以看到，在 AF_INET 协议族中，TCP、UDP 的标志都为“永久”，不能从内核卸载。第三个协议为“裸”（raw）IP 协议，套接字类型为 SOCK_RAW，标志为可重用套接字。注意其协议 protocol 数据域的值为 IPPROTO_IP（该值为 0），并注释为“通配符”，其含义为裸套接字，实际是用于在 AF_INET 协议族中为各协议设置套接字选项的通用套接字。

如果要动态地在 TCP/IP 协议栈中加入新协议或在 Linux 内核中加入新协议栈，可以调用以下两个函数分别加入协议实例或卸载协议实例。

```
void  inet_register_protosw(struct inet_protosw *p)
void  inet_unregister_protosw(struct inet_protosw *p)
```

由 struct inet_protosw 数据结构类型数组 inetsw_array[]构成的向量表，称为协议交换表。整个协议交换表注册机制就由以上所说的注册/注销协议实例函数及数据结构 struct inet_protosw 来维护和管理。当调用 inet_register_protosw 函数向套接层注册某个协议族的协议实例的协议交换表时，函数以输入参数 struct inet_protosw *p 的 p->type 和 p->protocol 为关键字，用 p->type 在数组中搜索，找到注册协议所属的套接字类型，将协议实例的 struct inet_protosw 数据结构块放入数组的某个链表中。

AF_INET 协议族的协议交换表如图 10-4 所示。

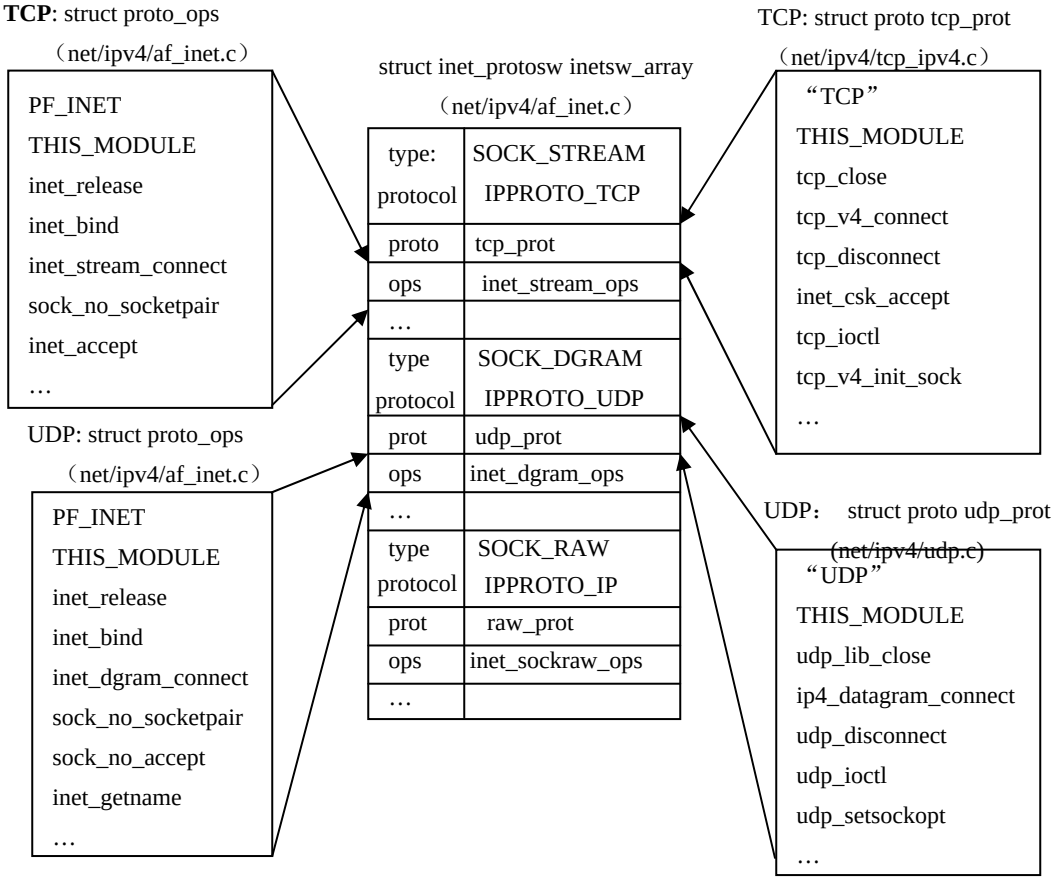


图 10-4　AF_INET 协议族的协议交换表

由上图可知，在 struct inet_protosw 协议交换表左边的是 AF_INET 协议族中 TCP、UDP 协议实例实现的套接字的管理、控制函数，是套接字层与应用层之间的调用接口，通过系统调用 socketcall 访问。在 struct inet_protosw 协议交换表右端的是 AF_INET 协议族，即 TCP/IP 协议栈在传输层协议实例的网络操作功能。来自应用层的请求由套接字层的 API 经协议交换表，由协议实例的网络功能函数完成所要求的操作。

10.4 IPv4 中协议成员注册和初始化

以上是 AF_INET 协议族的协议实例套接字层的操作函数块 struct proto_ops 实例与网络功能函数块 struct proto 数据结构实例的对应关系，在该对应与转换注册到内核后，由协议交换表 inetsw_array[]来管理。

1. 套接字层的操作函数块

前面已经介绍了，协议套接字与系统调用 socketcall 之间的接口 API 数据结构块，是由函数 inet_register_protosw 注册到 inetsw_array[]协议交换表中的，由函数 inet_unregister_protosw 将结构块从协议交换表中移出。

2. 协议实例的网络功能函数块

协议实例的网络功能函数块 struct proto 则是由 proto_register 函数注册到 inetsw_array[]协议交换表中，由 proto_unregister 函数将其从 inetsw_array[]协议交换表中移出。以上两个函数在第 8 章与第 9 章中都有简单介绍。

3. 协议交换表初始化

inetsw_array[]协议交换表在何处初始化？TCP/IP 协议栈属于 AF_INET 协议族，inetsw_array[]协议交换表初始化应在 AF_INET 协议族初始化函数 inet_init 中完成。

```
static int __init inet_init(void)
{
//将 TCP 协议实例网络功能函数放入套接字协议交换表
  rc = proto_register(&tcp_prot, 1);
   …
//将 UDP 协议实例网络功能函数放入套接字协议交换表
  rc = proto_register(&udp_prot, 1);
//初始化协议交换表的套接字层的存放各协议族 API 的链表
  for (r = &inetsw[0]; r < &inetsw[SOCK_MAX]; ++r)
        INIT_LIST_HEAD(r);

//将 AF_INET 协议族套接字层的 API 注册到协议交换表

  for (q = inetsw_array; q < &inetsw_array[INETSW_ARRAY_LEN]; ++q)
        inet_register_protosw(q);
…
```

由以上代码可知，inet_init 对 inetsw_array 数组中的每个协议调用 inet_register_protosw 函数将其放入协议都是交换表中的。目前在数组中的协议有 UDP、TCP 和裸 IP。宏 fs_initcall(inet_init)标记了 TCP/IP 协议栈的初始化函数 inet_init 会在内核启动过程中执行，完成 AF_INET 协议族的整个初始过程。

一旦注册完成，其他协议常以模块的方式实现，可以调用 inet_register_protocols 函数将其动态地加入到协议交换表中。

10.5　套接字 API 系统调用的实现

在早期的嵌入式系统中，无论是操作系统还是应用程序，都是在一个连续地址空间中实现的，没有用户地址空间或内核地址空间之分，因为早期的 CPU 内都没有 MMU（Memory Management Unit）存储器管理单元。MMU 单元的主要任务是将内存的物理地址空间映射到虚拟地址空间。这时用户应用程序与操作系统代码工作在同一地址空间，用户程序可以直接调用内核函数，不需要做地址空间的转换。但这种系统的最大问题是系统稳定性和安全性存在很大的隐患。

现代嵌入式 CPU 芯片都带有 MMU 单元，而且 Linux 操作系统是工作在虚拟存储器上的操作系统，需要处理器 MMU 单元来实现虚—实地址转换。工作在虚拟存储器上的操作系统，其用户虚拟地址空间与内核虚拟地址空间是隔开的，而且每个用户进程都有它们自己的虚拟地址空间，任何一个用户进程出错都不会影响到其他用户进程的执行，也不会影响内核代码，可以使系统运行得更稳定和安全。在这样的工作模式下，用户进程需要经过系统调用来获取内核的功能服务。

10.5.1　系统调用简述

在 Linux 系统中用户进程与内核进程工作在不同的虚拟地址空间，当用户程序执行的某个功能需要访问内核功能函数时，需要解决以下问题：

- 用户程序的函数调用如何切换到正确的内核函数。
- 用户程序的函数参数如何传递给内核代码。
- 用户程序如何获取内核函数的返回值。

在 Linux 操作系统中以上过程通过系统调用来实现。系统调用完成将用户程序的函数调用转换成对内核功能服务调用，并获取内核功能服务的返回值，然后传递给用户程序。实现系统调用的基本过程如下：

① 内核实现了一系列的系统调用，所有系统调用函数以 sys_为前缀名（如 sys_bind），对应用户程序的函数调用。内核为每个系统调用分配一个索引号，索引号与系统调用函数的对应关系保存在系统调用表 system call table 中。

② 用户程序调用参数在用户地址空间，内核功能函数要访问这些参数，需将其从用户地址空间映射到内核地址空间。在这一节中，这些参数是通过 TCP/IP 协议栈接收或发送的数据、控制信息。

在某些 CPU 体系结构中，如 X86 CPU 体系结构，它们是将存放用户数据地址的指针放入 CPU 的寄存器中，传给内核代码，然后内核代码将用户数据复制到内核地址空间，这项功能由 copy_from_user 函数完成。

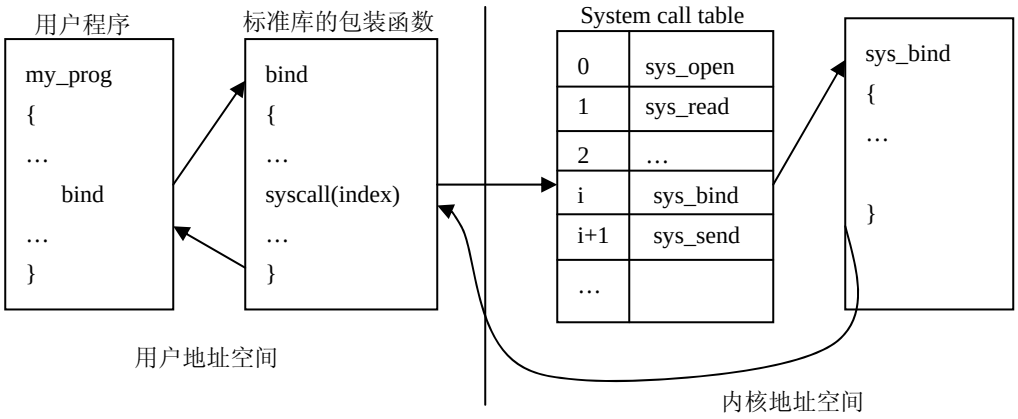


图 10-5　系统调用表与用户程序对应关系

③ 如果系统调用成功，则向用户程序返回正整数或 0。如果不成功，则系统调用向用户程序返回错误代码。

10.5.2　套接字 API 系统调用的实现

套接字 API 除了支持 AF_INET 协议族（即 TCP/IP 协议栈）外，还可以支持其他协议族。由 Linux 支持的所有协议族如表 10-1 所示。除此之外，Linux 的网络体系结构还允许用户定义新的未知协议族，以模块的形式动态地加入内核或从内核中移去。因为 Linux 支持多个协议族，所以在应用程序调用套接字 API 时，应用层的函数调用转换为内核协议实例函数需经过以下步骤：

① 将用户地址空间参数映射到内核地址空间。

② 将常规套接字层的函数调用转换到某协议族的函数，如 AF_INET 协议族。

③ 将协议族的函数调用传递给协议实例的网络功能函数。

1. socketcall 系统调用

常规文件的读/写操作由虚拟文件系统的系统调用传到内核后，会重定向到 socket_file_ops 函数指针结构上。但套接字的操作除了常规的读/写操作外，还需完成一些其他操作功能，如创建套接字 socket、套接字与地址绑定 bind 和套接字侦听系统调用。为此 Linux 提供了 socketcall 系统调用来实现以上功能，在内核中由 sys_socketcall 函数实现。

在 Linux 内核中只有一个系统调用 sys_socketcall 完成用户程序对所有（17 个）套接字操作的调用，这需要以不同的参数来决定如何处理用户程序的调用。传给 sys_socketcall 函数的第一个参数是一个数字常数，sys_socketcall 以该常数为索引选择需要调用的实际函数，这些值可以是 SYS_SOCKET、SYS_BIND 等，所有这些索引值定义在 include/linux/net.h 文件中。

```
//include/linux/net.h
#define SYS_SOCKET            1          /* sys_socket*/
#define SYS_BIND              2          /* sys_bind*/
#define SYS_CONNECT           3          /* sys_connect*/
```

```
#define SYS_LISTEN            4          /* sys_listen*/
#define SYS_ACCEPT            5          /* sys_accept*/
#define SYS_GETSOCKNAME       6          /* sys_getsockname*/
#define SYS_GETPEERNAME       7          /* sys_getpeername*/
#define SYS_SOCKETPAIR        8          /* sys_socketpair*/
#define SYS_SEND              9          /* sys_send*/
#define SYS_RECV              10         /* sys_recv*/
#define SYS_SENDTO            11         /* sys_sendto*/
…
```

以上注释中以 sys_为前缀的函数，都定义在 net/socket.c 文件中。这些函数的主要功能是调用协议族的套接字功能函数；协议族功能函数通过 struct socket 数据结构的 ops 指针访问。前面已经介绍了，struct socket 数据结构包含在 struct sock 数据结构中；struct sock 数据结构中包含了一个打开套接字所需的所有信息。通过如图 10-4 所示的协议交换表，对协议族功能函数调用传递到协议族中协议实例功能函数，来完成实际网络操作功能。

2. **套接字分路器** sys_socketcall

sys_socketcall 函数接收所有来自应用程序对套接字的 17 种操作的系统调用，sys_socketcall 函数的功能并不复杂，它犹如一个分路器，将来自应用程序的系统调用分支到其他函数上，每个函数实现一个小的系统调用功能。

应用程序调用应用层套接字的 API 函数时，会产生一个内核系统调用中断，该中断转而调用 sys_socketcall 函数（定义在 include/linux/syscalls.h 文件中），sys_socketcall 函数定义原型如下：

```
asmlinkage logn sys_socketcall(int call, unsigned long __user *args);
```

sys_socketcall 函数主要完成两个功能：

- 将从用户地址空间传来的每一个地址映射到内核地址空间。该功能通过调用 copy_from_user 函数完成。
- 将应用层套接字的 API 函数映射到内核实现函数上。例如，用户程序调用标准库函数套接字的绑定功能函数 bind 来完成打开套接字与 IP 地址、端口号绑定的功能，即 sys_socketcall 函数把用户层的 bind 映射到内核功能函数 sys_bind 上。

sys_socketcall 函数的两个输入参数是实现以上两个功能的依据。

- call：系统调用函数的索引号。
- *args：指向用户地址空间参数地址的指针。

sys_socketcall 函数首先检查函数调用索引号是否正确，其后调用 copy_from_user 函数将用户地址空间参数复制到内核地址空间。最后 switch 词句根据系统调用函数索引号，实现套接字分路器的功能，将来自应用程序的系统调用转到内核实现函数 sys_xxx 上。

```
SYSCALL_DEFINE2(socketcall, int, call, unsigned long __user *, args)       //net/socket.c
{
…

//复制用户地址空间参数到内核地址空间
  if (copy_from_user(a, args, nargs[call]))

//将应用层套接字 API 调用转换到内核函数
  switch (call) {
  case SYS_SOCKET:
```

```
            err = sys_socket(a0, a1, a[2]);
            break;
    case SYS_BIND:
            err = sys_bind(a0, (struct sockaddr __user *)a1, a[2]);
            break;
    case SYS_CONNECT:
            err = sys_connect(a0, (struct sockaddr __user *)a1, a[2]);
            break;
    case SYS_LISTEN:
        …
```

3. 由内核系统调用函数到协议实例函数

以上由 sys_socketcall 函数调用分路的内核套接字调用函数的实现也定义在 net/socket.c 文件中。

（1）sys_bind 函数

通过 sys_socketcall 函数后，应用层调用套接字 bind 功能，会转而调用 sys_bind 函数。

```
asmlinkage long sys_bind(int, struct sockaddr __user *, int);
SYSCALL_DEFINE3(bind, int, fd, struct sockaddr __user *, umyaddr, int, addrlen)
{
  struct socket *sock;
  struct sockaddr_storage address;
  int err, fput_needed;
  sock = sockfd_lookup_light(fd, &err, &fput_needed);
  …
err = sock->ops->bind(sock, (struct sockaddr *)&address, addrlen);
…
  }
```


（2）sys_listen 函数

通过 sys_socketcall 函数后，应用层调用套接字 listen 功能，会转而调用 sys_listen 函数。

```
asmlinkage long sys_listen(int, int)
SYSCALL_DEFINE2(listen, int, fd, int, backlog)
{
  struct socket *sock;
  int err, fput_needed;
  int somaxconn;

  sock = sockfd_lookup_light(fd, &err, &fput_needed);
…
  err = sock->ops->listen(sock, backlog);
…
}
```

每个内核的系统调用函数返回一个文件描述符（fd），返回的文件描述符也作为套接字的引用。为了支持标准 I/O 操作，从 I/O 调用的角度看套接字和文件描述符是等同的。内核中每个系统套接字函数都将打开套接字的文件描述符作为第一个参数。当内核的系统套接字的 API 被调用时，在内核代码中，我们使用文件描述符来提取套接字结构的指针，该套接字结构是在打开套接字时创建的。一旦获取套接字结构，就可以从套接字结构中提取协议族信息 sock->type，知道协议族类型，就可以通过 struct socket 数据结构的 struct proto_ops *ops 数据域访问特定协议的函数。

为了说明如何映射套接字的调用，这里以应用层套接字 API send 为例来说明整个过程。当应用程序调用 send 函数在打开的套接字上传送数据时，内核通过 sys_socketcall 套接字分路器将 send 函数调用翻译成 sys_sendto，sys_sendto 会调用 sock_sendmsg 函数，sock_sendmsg 函数又会调用由 struct socket 数据结构中 struct proto_ops *ops 数据域引用的协议实现函数 sendmsg：

```
//从应用层套接字 API 的 send 转而调用内核 sys_sendto
SYSCALL_DEFINE4(send, int, fd, void __user *, buff, size_t, len,
        unsigned, flags)                                                    //net/socket.c
{
  return sys_sendto(fd, buff, len, flags, NULL, 0);
}

SYSCALL_DEFINE6(sendto, int, fd, void __user *, buff, size_t, len,
        unsigned, flags, struct sockaddr __user *, addr,
        int, addr_len)
{
  struct socket *sock;
  …

//由文件描述符 fd 获取套接字数据结构 struct socket *sock，套接字数据结构在打开套接字时创建
  sock = sockfd_lookup_light(fd, &err, &fput_needed);
  …

//复制用户地址空间地址参数到内核地址空间
  if (addr) {
        err = move_addr_to_kernel(addr, addr_len, (struct sockaddr *)&address);
…

//sys_sendto 调用特定协议族的套接字实现函数 sock_sendmsg
  err = sock_sendmsg(sock, &msg, len);
…
}

int sock_sendmsg(struct socket *sock, struct msghdr *msg, size_t size)
{
  …
  ret = __sock_sendmsg(&iocb, sock, msg, size);
  …
}

static inline int __sock_sendmsg(struct kiocb *iocb, struct socket *sock,
                    struct msghdr *msg, size_t size)
{
  …
  return sock->ops->sendmsg(iocb, sock, msg, size);
}
```

4. 内核套接字系统调用函数说明

sys_bind、sys_listen、sys_connect、sys_getsockname、sys_getpeername 完成的功能不多，除了将用户地址空间的地址参数复制到内核地址空间以外，还可以直接调用协议实例的实现函数： sock->ops->bind、sock->ops->listen、sock->ops->connect 等。

用户程序用几种途径来进行网络数据传送，它们可以使用两个网络相关的系统调用 sendto 与文件层的 send 或 write 与 writev 函数系统调用来实现，应用层的调用会传给内核系统调用函数 sys_send，该函数直接调用 sys_sendto 函数。sys_sendto 函数调用套接字层的传送函数 sock_sendmsg。图 10-6 为 sys_sendto 函数的调用流程。

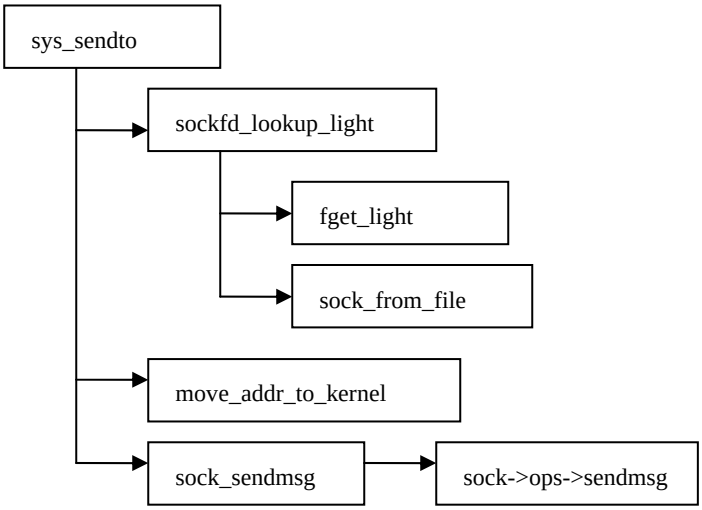


图 10-6 sys_sendto 函数的调用流程

fget_light 和 sock_from_file 函数负责查寻由文件描述符引用的相关套接字，在 sock_sendmsg 函数调用协议实例的传送函数之前，传送数据由 move_addr_to_kernel 函数从用户地址空间复制到内核地址空间。

用户程序接收数据使用 recvfrom 和 recv 系统调用、文件相关的 readv 和 read 函数完成。因为这些功能的代码都非常相似，所以合并为对一个接收 sys_recv 函数的调用，sys_recv 函数则是直接调用 sys_recvfrom 函数，sys_recvfrom 函数转而调用套接字层的 sock_recvmsg 接收函数。sys_recvfrom 函数的调用流程如图 10-7 所示。

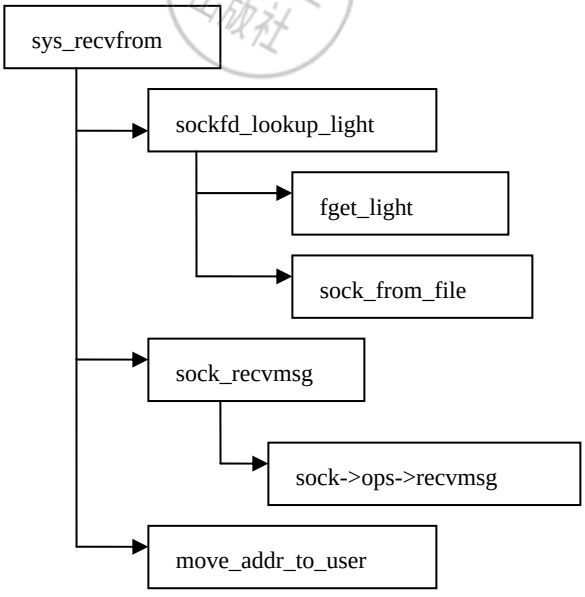


图 10-7 sys_recvfrom 函数的调用流程

通过系统调用传来的文件描述符指明用户程序通过哪个套接字传送数据，因此第一个

任务就是找到相关的套接字。首先由 fget_ligh 函数访问进程数据结构的描述符表，查询到相符的 file 实例，sock_from_file 函数确定与文件实例相关的 inode，最后用 SOCKET_I 宏将 inode 映射到套接字描述符上。

经过一些预处理后，sock_recvmsg 调用协议实例接收例程 sock->ops->recv_msg 函数来接收数据。例如，TCP 的接收函数是 tcp_recvmsg，UDP 协议的接收函数是 udp_recvmsg。

最后 move_addr_to_user 将内核地址空间的数据复制到用户地址空间。

用户程序可以在打开的套接字上分别设置套接字层、传输层 TCP/UDP、IP 层的选项。设置选项的系统调用函数 sys_setsockopt 和读取选项的 sys_getsockopt 函数，首先要查看用户参数指定要设置哪个层次的选项：套接字层、传输层 TCP 或 IP 层。如果指明协议层参数值为 SOL_SOCKET，则说明当前用户程序设置（或读取）套接字层选项。以上两个系统调用函数分别转而调用套接字层的 sock_setsockopt 或 sock_getsockopt 函数。如果指明协议层参数指定的是其他 TC 协议栈层，相应的协议实例函数通过 struct socket *sock 套接字数据结构的 struct proto_ops *ops 数据域（sock->ops->…）调用具体的实现函数。

sys_shutdown 系统调用也通过 sock->ops->shutdown 数据域直接调用相应的协议实例函数。

sys_sendmsg 和 sys_recvmsg 系统调用函数比其他系统调用函数实现要复杂点。首先，它们必须查看 iovec 缓冲区数组中包含的用户空间地址是否有效；然后，将每个地址从用户地址空间映射到内核地址空间；最后，调用套接字实现函数 sock_sendmsg 和 sock_recvmsg 来接收和发送消息。

sys_accept 系统调用的实现就更复杂一点，它必须创建一个新的套接字来管理接收到的连接。首先，它调用 sock_alloc 分配一个新的套接字数据结构对象；然后它需获取套接字名，该功能调用 sock->ops->getsockname 函数实现，因为套接字名中包含了数据包的目标 IP 地址和端口号信息。最后，调用 sock_map_fd 函数将新的套接字映射到套接字文件系统上。

接下来就是分别实现内核系统调用函数和套接字层的功能函数。这些函数都定义在 net/socket.c 文件中。

AF_INET 协议族中传输层 UDP、TCP 协议实例函数的实现在第 8~9 章已经介绍了，这里就不再赘述。

10.6　创建套接字

在应用程序使用 TCP/IP 协议栈的功能之前，必须调用套接字库函数 API 创建一个新的套接字；对库函数创建套接字的调用会转换为内核套接字创建函数的系统调用。这时完成的是通用套接字创建的初始化功能，不与具体的协议族相关。这个过程即在应用程序中执行 socket 函数，socket 产生系统调用中断执行内核的套接字分路函数 sys_socketcall，在 sys_socketcall 套接字函数分路器中将调用传送到 sys_socket 函数，由 sys_socket 函数调用套接字的通用创建函数 sock_creat。sock_create 函数完成通用套接字创建、初始化任务后，再调用特定协议族的套接字创建函数，例如，由 AF_INET 协议族的 inet_create 函数完成套

接字与特定协议族的关联。

10.6.1 sock_create 函数创建套接字

一个新的 struct socket 数据结构起始由 sock_create 函数创建，该函数直接调用 __sock_create 函数，__sock_create 函数的任务是为套接字预留需要的内存空间，由 sock_alloc 函数完成这项功能。sock_alloc 函数不仅会为 struct socket 数据结构实例预留空间，也会为 struct inode 数据结构实例分配需要的内存空间，这样可以使两个数据结构的实例相关联。

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

//查看要创建的套接字所属的协议族是否在 Linux 支持的协议族中，见表 10-1
//套接字类型在 Linux 内核支持的套接字类型范围内
  if (family < 0 || family >= NPROTO)
        return -EAFNOSUPPORT;
  if (type < 0 || type >= SOCK_MAX)
        return -EINVAL;

//调用 sock_alloc 函数为新套接字分配内存空间，如果分配成功，则函数将返回指向新套接字结构的指针
  sock = sock_alloc();
…

//在新分配的套接字内存空间上创建新套接字并初始化
err = pf->create(net, sock, protocol);
```


套接字数据结构 struct socket 实际上也是 inode 文件系统节点数据结构的一部分，它是由 sock_alloc 函数调用 new_inode 创建的。Inode 节点的作用是在 Linux 的通用 I/O 系统调用中与套接字交互。struct inode 数据结构中的大部分数据域只对实际的文件系统有意义，但其中有一部分为套接字所用，如表 10-7 所示。

表 10-7 套接字使用的 struct inode 数据结构中的字段

数据域	原型	作用
u		在套接字类型的 inode 中，该数据域包含的是 struct socket 数据结构
i_dev	kdev_t	设置为设备，对于套接字 inode 该数据域总为 NULL
i_sock	unsigned char	套接字 inode 设置为 1
i_mode	umode_t	指明 I/O 访问权限和接口类型
i_uid	uid_t	打开套接字的当前用户进程 ID
i_gid	i_gid	打开套接字的用户进程的组进程 GID
i_fop	struct file_operations *	指向套接字的文件操作

一旦 struct inode 数据结构创建成功，sock_alloc 函数就从 struct inode 数据结构中提取 struct socket 数据结构的信息。然后它初始化 struct socket 数据结构的部分数据域，将 struct

socket 数据结构的 inode 数据域设置为指向包含该套接字结构的 inode 结构实例，将 fasync_list 设置为空。fasync_list 是支持 fsync 系统调用的，fsync 是为实现在内存的文件与永久存储器之间同步的函数，对于套接字，fsync 会将缓冲区中的数据存入套接字层。

套接字的数据结构需要维护一个状态，这个状态说明打开的套接字是否为与另一个站点建立了连接的套接字。该状态存放在 struct socket 数据结构的 state 数据域中。套接字的各种状态如表 10-2 所示。初始化时 sock->state = SS_UNCONNECTED。

这时需要注意的是，现阶段套接字并不只用于 TCP/IP 协议栈。在 struct socket 数据结构中的状态不完全与 TCP 的状态相同，TCP 套接字的状态更复杂一些。套接字的状态只反映套接字是否为一个活动的、建立了连接的套接字。

```
    static struct socket *sock_alloc(void)                  //net/socket.c
{
    struct inode *inode;
    struct socket *sock;
    inode = new_inode(sock_mnt->mnt_sb);
    if (!inode)
          return NULL;

//将 struct inode 数据结构上的指针转换到对应的 struct socket 数据结构上
    sock = SOCKET_I(inode);
//设置文件操作权限，当前用户 ID、用户组 ID
    inode->i_mode = S_IFSOCK | S_IRWXUGO;
    inode->i_uid = current_fsuid();
    inode->i_gid = current_fsgid();

//对套接字引用计数加 1，返回指向新创建套接字的指针
    get_cpu_var(sockets_in_use)++;
    put_cpu_var(sockets_in_use);
    return sock;
}
```

当具体的协议与新套接字相连时，其内部状态的管理由协议自身维护。现在，函数将 struct socket 数据结构的 struct proto_ops *ops 设置为 NULL。随后当某个协议族中的协议成员的套接字创建函数被调用时，ops 将指向协议实例的操作函数。

这时将 struct socket 数据结构的 flags 数据域设置为 0，创建时还没有任何标志需要设置。在以后的调用中，应用程序调用 send 或 receive 套接字库函数时会设置 flags 数据域。表 10-4 中描述了套接字 API 使用的 flags 值。最后将其他两个数据域 sk 和 file 初始化为 NULL。sk 数据域随后将由协议特有的套接字创建函数设置为指向内部套接字结构。file 将在调用 sock_ma_fd 函数时设置为分配的文件返回的指针。文件指针用于访问打开套接字的虚拟文件系统的文件状态。

在 sock_alloc 函数返回后，sock_create 函数调用协议族的套接字创建函数 err = pf->create(net, sock, protocol)，它通过访问 net_families 数组获取协议族的创建函数，对于 TCP/IP 协议栈，协议族将设置为 AF_INET。

套接字创建的流程如图 10-8 所示。

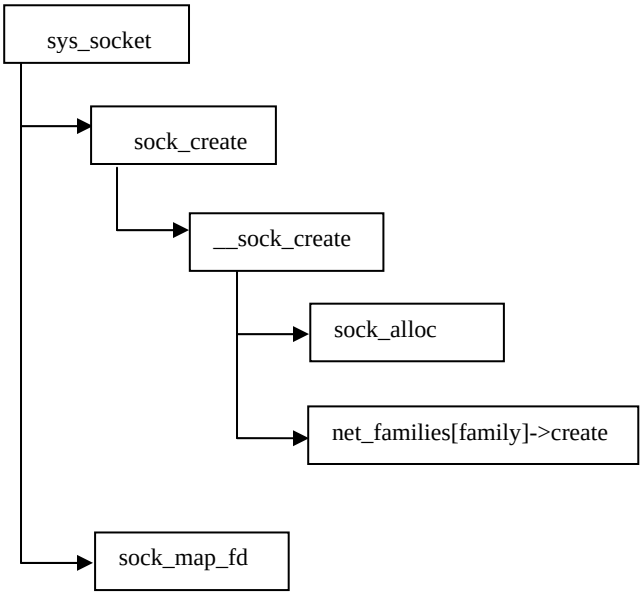


图 10-8　sys_socket 函数的调用流程图

10.6.2　协议族套接字创建函数的管理

内核中所有传输层协议的套接字创建函数结构块组织在 static struct net_proto_family *net_families[NPROTO]数组中，数组中的每个成员是各协议的套接字创建、初始化函数。

1. 协议族套接字创建函数的数据结构

协议族的套接字创建函数由 struct net_proto_family 数据结构管理，该数据结构定义在 include/linux/net.h 文件中。

```
struct net_proto_family {                                    //include/linux/net.h
  int        family;
  int        (*create)(struct net *net, struct socket *sock, int protocol);
  struct module    *owner;
};
```

2. 协议族套接字创建函数的数组

所有传输层协议族的套接字创建函数存放在 net_families 全局数组中。该数组定义在 net/socket.c 文件中。

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

3. 协议族套接字创建函数的实例化

各协议族套接字创建函数在协议族实现文件内初始化，完成代码实现。AF_INET 协议族的初始化及创建函数的实现定义在 net/ipv4/af_inet.c 文件中。

```
static struct net_proto_family inet_family_ops = {
  .family = PF_INET,
```

```
    .create = inet_create,
    .owner= THIS_MODULE,
};
```

4. 将协议族套接字创建函数注册到 net_families 数组中

AF_INET 协议族的套接字创建函数在 inet_init 初始化函数中，调用 sock_register 函数实现注册。将 AF_INET 协议族的套接字创建函数结构块 inet_family_ops 放入 net_families 全局数组中。

```
static int __init inet_init(void)
{
…
//将 AF_INET 协议族套接字创建函数数据结构放入 net_families 数组中
        (void)sock_register(&inet_family_ops);
 …
}
```

在 net_families[NPROTO]全局数组中，存放了所有传输层协议实例的套接字创建数据结构成员。

10.6.3　AF_INET 套接字的创建

在 TCP/IP 协议栈中，AF_INET 协议族套接字的创建由 inet_create 函数实现，该函数定义在 net/ipv4/af_inet.c 文件中。

inet_create 函数定义为静态函数 static，是因为该函数不是直接调用，它是通过 struct net_proto_family 数据结构的 create 函数指针来调用的。在 AF_INET 协议族中，inet_create 函数在 sys_socket 函数内，当协议族类型参数设置为 AF_INET 时，调用来创建协议族特定套接字。inet_create 函数的任务是，创建一个新的 struct sock 数据结构类型的实例，在套接字数据结构创建成功后，它返回指向 struct sock 数据结构的指针 sk，并初始化 sk 的部分数据域。

新的 struct sock 数据结构从套接字的内存槽缓冲区中分配 sock 对象，内存对象分配由 inet_sk_slab 函数完成，它创建一个新的 struct sock 数据结构实例。Linux 内有多个 inet_sk_slab 内存缓冲槽，每个协议都有一个。Linux 内存槽都是为特有的内存对象专门设置的，事先格式化了一系列的对象。这样可以使数据结构对象的创建更有效，因为许多数据域可以事先初始化为常用的值。在 inet_create 函数中调用 sk_alloc 函数从内存槽上为 struct sock 数据结构分配空间，并指定当前在替什么类型协议分配 struct sock 数据结构。

inet_create 函数搜索协议交换表查找匹配的协议。在从协议交换表中获取结果后，查看当前进程的权限标志，如果调用进程无权限创建该类型的套接字，inet_create 函数就向用户层返回 EPERM 错误信息。

随后，inet_create 函数初始化新 struct sock 数据结构的某些数据域，许多数据域在为 struct sock 数据结构分配内存空间时已预先初始化过。将 sk_family 数据域设置为 PF_INET。将新分配成功的 struct sock 数据结构的其他数据域初始化为：

- prot 数据域设置为传输层的协议处理函数块。
- no_check 和 ops 按照从协议交换表中查询的值设置。
- 如果 sk->type 的值为 SOCK_RAW，则将协议编号域设置为通用协议编号等。

```
static int inet_create(struct net *net, struct socket *sock, int protocol)
{
  struct sock *sk;
  struct inet_protosw *answer;
  struct inet_sock *inet;
  struct proto *answer_prot;
  …
//查询协议交换表，根据协议族套接字创建类型 type 获取要创建的协议实例。如果是 TCP 协议，type 为 SOCK_STREAM
lookup_protocol:
  …
  list_for_each_entry_rcu(answer, &inetsw[sock->type], list) {
  …
//查看用户进程是否有权限创建该类型的套接字
  if (answer->capability > 0 && !capable(answer->capability))
       goto out_rcu_unlock;
  …
//分配一个新的 struct sock *sk 数据结构，套接字类型为 PF_INET，协议为从协议交换表中获取的协议类型
  sk = sk_alloc(net, PF_INET, GFP_KERNEL, answer_prot);
   …
  sk->sk_no_check = answer_no_check;
  if (INET_PROTOSW_REUSE & answer_flags)
       sk->sk_reuse = 1;

//AF_INET 支持两个协议 IPv4 和 IPv6，由 inet_sk 宏确定是哪个协议
  inet = inet_sk(sk);
  inet->is_icsk = (INET_PROTOSW_ICSK & answer_flags) != 0;
//初始化 sk 的某些数据域
  sock_init_data(sock, sk);

  sk->sk_destruct   = inet_sock_destruct;
  sk->sk_family   = PF_INET;
  sk->sk_protocol   = protocol;
  sk->sk_backlog_rcv = sk->sk_prot->backlog_rcv;
…
}
```


10.7 I/O 系统调用和套接字

Linux 系统是 UNIX 兼容的操作系统，Linux 中有统一的 I/O 例程。所有的 I/O 系统独立实现；I/O 系统的操作独立于设备、文件或套接字。对大多数应用程序而言，不需要区分 I/O 系统下的具体设备是什么。这种实现的最大优点是，对 Linux 应用程序人员而言，他们不需要记住三组 API 函数功能—— 一组为文件操作的 API，另一组用于设备 I/O 操作的 API，第三组用于网络操作的 API。在这一节中将介绍文件 I/O 是如何工作于网络套接字应用的，以及为了实现对应用层统一的 I/O 操作接口，套接字层自身需完成哪些功能。

系统中所有的 I/O 设备、文件和其他一些实体都有一个文件描述符，文件描述符引用文件系统中的 inode 对象，所有类文件系统操作的对象都有 inode 引用。inode 对象允许套

接字与虚拟文件系统（VFS：Virtual File System）相连。每个 I/O 操作系统调用如 read、write、fcntl、ioctl 和 close/open 等都要访问 struct inode 数据结构。在一个打开的套接字上执行 I/O 操作系统调用时，需从 struct inode 数据结构中提取指向 struct socket 数据结构的指针。但 open 系统调用不能用于创建套接字，创建套接字必须使用 socket 库函数调用。一旦套接字创建成功，I/O 系统调用在套接字上的工作方式就与在文件或设备上的工作方式一致。

struct inode 数据结构由 sock_alloc（定义在 net/socket.c 文件中）函数创建和初始化。如前所述，sock_alloc 函数由 sock_create 函数调用。sock_alloc 函数既创建 struct socket 数据结构，也创建 struct inode 数据结构。online 函数的 SOCKET_I（定义在 include/linux/socket.h 文件中）将 inode 映射到 socket，SOCK_INODE 将 socket 映射到 inode。

```
static inline struct socket *SOCKET_I( struct inode *inode)
static inline struct inode *SOCK_INODE( struct socket *socket)
```

一旦创建了 struct inode 数据结构，套接字层就可以映射到 I/O 系统调用上，在 net/socket.c 文件中套接字实现代码定义了一个 struct file_operation 文件操作函数结构的实例 socket_file_ops，socket_file_ops 的数据域初始化为套接字的 I/O 系统调用函数。

```
static const struct file_operations socket_file_ops = {
   .owner =THIS_MODULE,

//将 llseek 调置为常规的“no_llseek”，因为套接字不支持 llseek 系统调用
   .llseek =no_llseek,
   .aio_read =sock_aio_read,
   .aio_write =sock_aio_write,
   .poll =sock_poll,
   .unlocked_ioctl = sock_ioctl,

//如果内核配置了向后兼容
#ifdef CONFIG_COMPAT
   .compat_ioctl = compat_sock_ioctl,
#endif
   .mmap =sock_mmap,
//套接字不支持 open 系统调用，open 设置为 sock_no_open 不允许通过 /proc 文件系统打开套接字
   .open =sock_no_open,
   .release =  sock_close,
   .fasync =   sock_fasync,
   .sendpage =sock_sendpage,
   .splice_write = generic_splice_sendpage,
   .splice_read =    sock_splice_read,
};
```

10.8　本章总结

本章讨论的是 Linux 内核套接字的概念与套接字在 Linux 内核中的实现。套接字是 UNIX 兼容系统的一大特色，是 UNIX 一切皆是文件操作概念的具体实现，Linux 在此基础上实现了内核套接字与应用程序套接字接口，在用户地址空间与内核地址空间之间提供了一套标准接口，实现应用套接字库函数与内核功能之间的一一对应，简化了用户地址空间与内核地址空间交换数据的过程。本章在介绍了套接字基本概念的基础上，描述了 Linux

内核套接字实现的以下特点：

- 从描述 Linux 套接字接口的数据结构、套接字接口初始化过程可知，Linux 套接字体系结构独立于具体网络协议栈的套接字，可以同时支持多个网络协议栈的工作。
- Linux 套接字体系结构支持 Internet 协议族的实现过程，具体分析了套接字从创建、协议接口注册与初始化过程。从中可以看到任何协议栈都可以在套接字通用体系结构的基础上派生出具有协议族特点的套接字接口。
- 套接字系统调用各功能函数传送到内核套接字功能函数上的过程。

第 11 章　应用层——网络应用套接字编程

本章主要介绍在 Linux 系统中，基于套接字编写网络应用程序的技术。本章描述了实现 Linux 系统网络应用程序编程需要掌握的基本 API、网络地址结构，在此基础上给出了应用各种套接字 API 实现网络通信应用程序的实例。

在编写跨越计算机网络通信的应用程序时，首先要确定采用什么协议，即通信双方协定将如何相互通信。在高层网络应用编程中，必须确定哪个程序来初始化通信，请求经过多长时间后要获取回答。例如，Web 服务器是一个长期运行的后台服务器，总是在接到从网络上发来的请求后返回消息给请求端。发出请求的一端称为 Web 客户端，通常为浏览器，负责发起与服务器通信初始化。这种方式是网络应用中最普遍的客户/服务器模式。客户端初始化通信可以简化协议也可以简化网络应用编程。

从应用编程人员的角度看，客户/服务器应用程序只是通过某个应用网络协议相互通信，实际上整个通信过程涉及了多层网络协议。在介绍完 TCP/IP 协议栈在 Linux 内核的实现后，本章将讲解基于 TCP/IP 协议栈的网络应用程序编程技术。结合前面章节的内容，来理解数据是如何在应用程序中产生、通过套接字传送到内核、穿越 TCP/IP 协议栈的，最后是如何经过网络设备发送出去的整个过程。图 11-1 给出了基于 TCP/IP 协议栈的客户/服务器模式示意图。

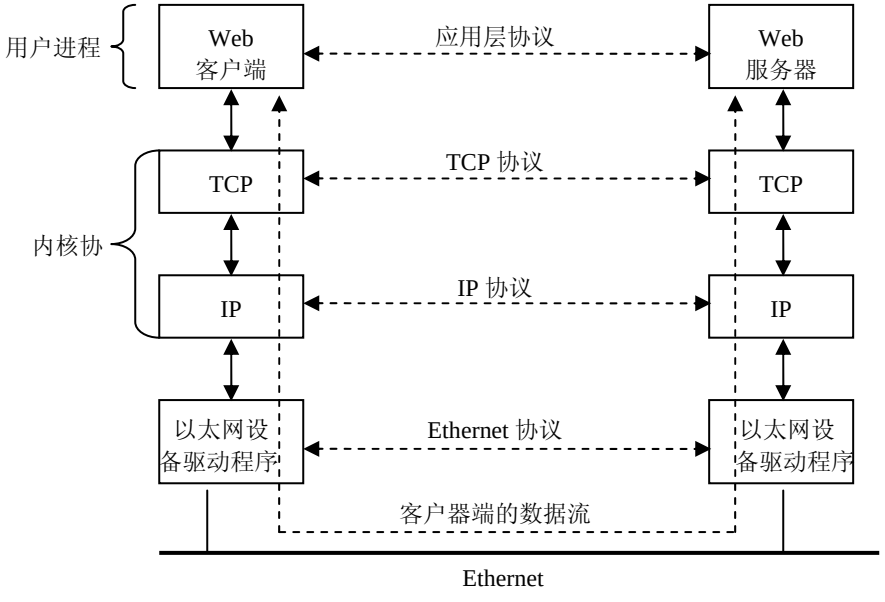


图 11-1　基于 TCP/IP 协议栈的客户/服务器模式

应用程序要访问 Linux 操作系统内核实现的网络功能，需要应用程序编程接口（API：Application Programming Interface）。在 Linux 中，从应用层访问网络传输层最通用的接口就是套接字编程 API。图 11-2 给出了基于 TCP 协议开发客户/服务器模式的网络应用程序需要的 API，以及客户/服务器端应用程序运行的时间安排。

首先启动服务器运行，稍后客户端就可以开始连接服务器。客户端向服务器端发出连接请求，服务器回答客户端的请求，连接建立起来，双方可以开始交换数据，这个过程一直延续到客户端关闭了其连接，向服务器端发送关闭连接的通知，随后服务器也关闭连接等待下一次客户连接请求。

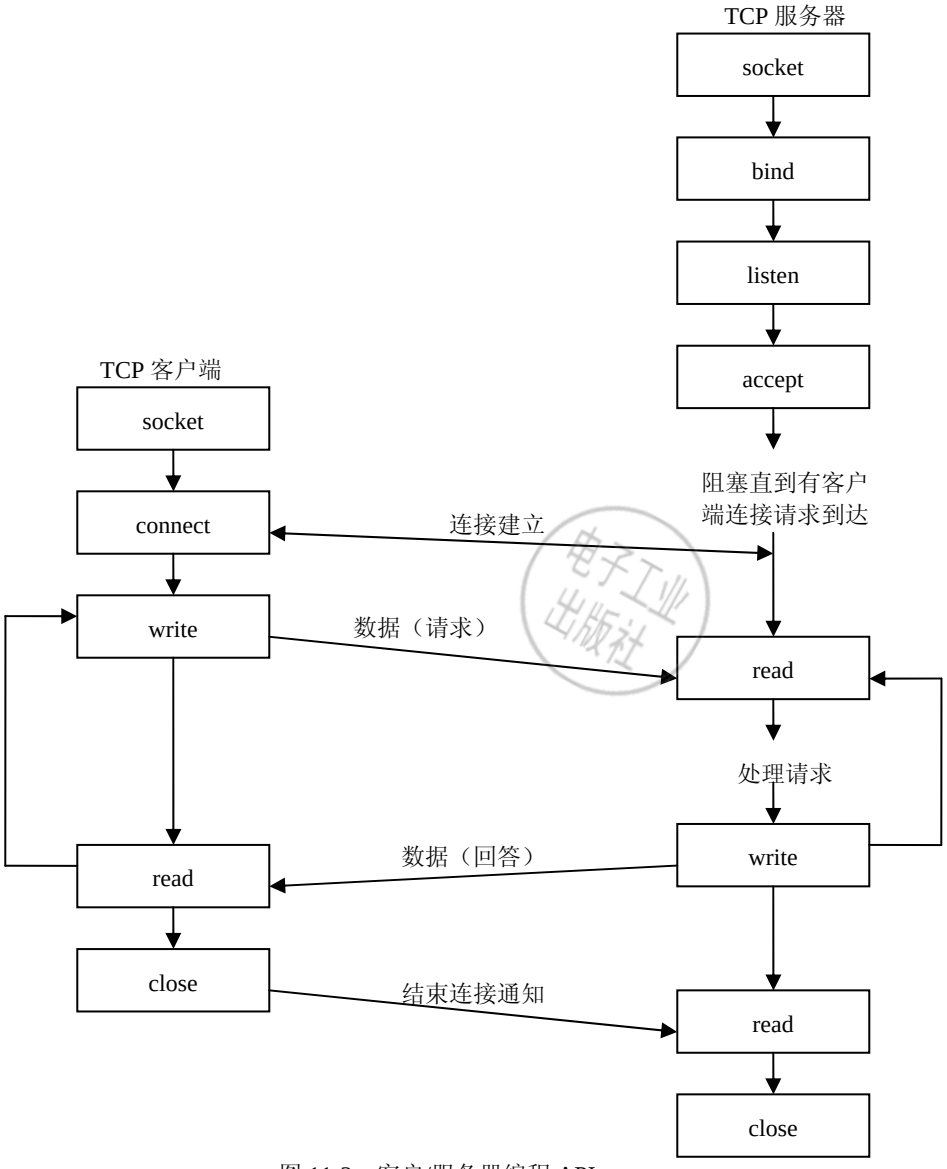


图 11-2　客户/服务器编程 API

在本章中，我们将描述套接字网络进程间通信接口，使用套接字网络接口可以使不同主机或同一主机之间的进程通过网络功能进行通信。这也是套接字网络接口设计的目的。同一 API 既可用于同一主机内进程间的通信，也可用于不同主机间的进程通信。虽然套接字接口可以在许多不同的网络协议上实现进程间通信，在本书中讨论的是 TCP/IP 协议栈在 Linux 系统上的

实现，所以在本章也仅限于讨论基于 TCP/IP 协议栈的套接字网络应用程序编程。

11.1　套接字描述符

一个套接字是一个通信端点的抽象，就如使用文件描述符来访问一个文件一样，应用程序使用套接字描述符来访问套接字。在 Linux 中实现了一套机制，使套接字的实现与文件描述符实现一样，使应用程序可以像访问文件一样访问套接字。许多用文件描述符访问文件的函数如 read 和 write，也同样可以用于套接字访问。

在应用程序中，为了通过套接字 API 使用 Linux 内核的网络功能，首先要创建套接字。创建套接字需调用 socket 函数，指定用于通信的协议类型。socket 函数的原型如下：

```
#include <sys/socket.h>
int socket ( int family, int type, int protocol );
//如果套接字创建成功，函数返回套接字描述符 file(socket) descriptor，否则返回错误代码
```

11.1.1　family 参数

调用 socket 函数需要向它传递 3 个参数：family、type、protocol。参数 family 指定协议族或地址族（该参数又常以 domain 为参数名），描述了通信的类型。表 11-1 列出了我们常用地址族（domain）。各地址族符号以 AF_为前缀，代表地址族 address family。每个地址族都有自己特定的地址格式。

表 11-1　常用地址族

地　址　族	描　　述
AF_INET	IPv4 Internet 域
AF_INET6	IPv6 Internet 域
AF_UNIX	UNIX 域
AF_UNSPEC	没有特别的指定

在大多数系统中还定义了 AF_LOCAL 地址域，AF_LOCAL 域是 AF_UNIX 域的别名，AF_UNSPEC 域可以广泛地代表任何域。

11.1.2　type 参数

type 参数定义了套接字的类型，套接字类型定义了套接字通信的特点。表 11-2 给出 Linux 常用的套接字类型。

表 11-2　套接字类型

套接字类型	描　　述
SOCK_DGRAM	用于固定长度、无连接、不可靠信息网络传送。对于 TCP/IP 协议栈，在应用程序使用 UDP 协议通信时，使用这类套接字
SOCK_RAW	与 IP 接口的裸套接字
SOCK_SEQPACKET	固定长度、顺序传送、可靠、面向连接的消息传送套接字
SOCK_STREAM	顺序传送、可靠、面向连接的字节流套接字。对于 TCP/IP 协议栈，在应用程序利用 TCP 协议通信时使用该套接字

11.1.3 protocol 参数

通常将 protocol 参数设置为 0，即对给定的域和套接字类型选择默认的协议。当同一个域和套接字类型可以支持多个网络协议时，应用程序可以使用 protocol 参数来指定使用什么网络协议进行通信。在 AF_INET 协议族中，protocol 参数可使用的值如表 11-3 所示。

表 11-3 协议类型

Protocol 值	描 述
IPPROTO_TCP	传输层 TCP 协议
IPPROTO_UDP	传输层 UDP 协议
IPPROTO_SCTP	传输层 SCTP 协议

在 AF_INET 协议域中，SOCK_STREAM 类套接字使用的默认协议是 TCP，SOCK_DGRAM 类套接字使用的默认通信协议是 UDP。

对于 SOCK_DGRAM 类型套接字接口，在通信时两个站点之间没有逻辑连接，在传送数据时，需要将数据地址（源地址和目标地址）也传给通信两端进程的套接字。

数据报（datagram）是一个自包含数据包，发送数据报的网络数据包类似于给某个人发送一封信件，你可以寄出很多信件，但不能保证信件传递顺序，而且有的信件在邮寄过程中还可能丢失。每封信件的信封上都必须有收信人的地址和发信人地址，信件之间相互独立。

数据报提供的是无连接服务，相反 SOCK_STREAM 类型的套接字，在两个站点交换数据时，必须在源套接字和目标套接字之间建立逻辑连接。

使用面向连接的协议进行通信就像平时打电话。首先你需要通过拨电话号码与想通信的一方建立连接，连接建立以后，通话双方可以双向地进行通信。连接是点对点的形式，但通信的内容中不需要包括地址信息。因为在通信双方端点之间存在一个虚拟连接，连接自身就包含了数据的源地址和目标地址。

在用 SOCK_STREAM 类型的套接字传送数据时，应用程序本身不知道传送的数据的边界，因为这类套接字提供的是字节流服务。即当应用程序从套接字读数据时，它返回的字节数可能与发送进程每次传送的字节数不同，但经过多次函数调用，最终应用程序可以收到全部的发送数据内容。

SOCK_SEQPACKET 类套接字与 SOCK_STREAM 套接字类似，唯一不同的是，SOCK_SEQPACKET 套接字提供的是基于消息的服务，而不是字节流服务。即从 SOCK_SEQPACKET 套接字收到的数据字节数与发送方写的数据长度完全一样。流控制传输协议（SCTP：Stream Control Transmission Protocol）在 Internet 域中提供顺序数据传送服务。

SOCK_RAW 套接字提供的是直接访问网络层数据报接口的套接字（在 Internet 域的 IP 层），当使用 SOCK_RAW 套接字时，如果数据没有经过传输层协议（如 TCP 和 UDP）处理，则应用程序需要自己创建协议头。使用 SOCK_RAW 套接字需要超级用户权限，以阻止恶意应用程序绕过系统安全机制传送数据包。

11.1.4 AF_XXX 与 PF_XXX 形式的常数

有时我们会看到 socket 函数的第一个参数是 PF_XXX 形式的常数，而不是 AF_XXX 形式的常数。AF_前缀代表的是地址族 address family， PF_前缀代表的是协议族 protocol

family。最初在开发套接字 API 时，是为了一个协议族可以支持多个地址族，创建套接字时 PF_的值代表协议族，AF_的值用于套接字的地址结构。但实际应用中一个协议族支持多个地址结构从来没有在系统中提供，在<sys/socket.h>头文件中定义了某个协议的 PF_值与该协议的 AF_的值相同。

套接字上的通信是双向通信，应用程序人员可以调用 shutdown 函数禁止在套接字上的 I/O 操作。

```
#include <sys/socket.h>

int shutdown ( int socketfd, int how );                    //函数执行，如果正确则返回 0，如果错误则返回 1
```

如何关闭套接字上的 I/O 功能，关闭方式由输入参数 how 给出。

- SHUT_RD：禁止从套接字上读数据。
- SHUT_WR：不能在套接字上发送数据。
- SHUT_RDWR：禁止在套接字上读/写数据。

做了以上操作后，程序就可以关闭套接字。为什么我们在关闭套接字前需要调用 shutdown？主要有以下几个原因：

- close 只会在最后一个引用套接字的进程释放了套接字后，才会释放网络端点的连接。即如果某个应用程序复制了套接字，套接字就不会被释放，直到应用程序关闭了最后一个引用套接字的文件描述符。而 shutdown 函数可以不考虑其他文件描述符对套接字的引用，直接停止套接字的活动。
- 某些时候从某个站点的一端单方面关闭套接字会更方便。例如，一个端点可以单方面关闭套接字的写活动，这样可以让与之通信的进程知道什么时候可以完成数据传送活动，同时自己的进程还可以继续接收其他进程传送的数据。以下为创建 TCP 协议套接字与 UDP 协议套接字的示例。

```
//如何创建一个使用 TCP 协议传送数据的套接字
#include <sys/socket.h>

main ( )
{
  int sd;

  sd = socket( AF_INET , SOCK_STREAM, 0 );
  if ( sd < 0 )
    return -1;
  …
}
```

在上例中，我们用 socket 函数创建了一个使用 Internet 地址族、面向连接的、按字节流传送的协议套接字，协议类型参数为 0，即是用 SOCK_STREAM 类型套接字的默认协议 TCP 协议。

```
//创建一个使用 UDP 协议传送的套接字
#include <sys/socket.h>

main ( )
{
   int sd;
```

```
    sd = socket ( AF_INET, SOCK_DGRAM, 0 );
    if ( sd< 0 )
      return -1;
    …
}
```

11.2 地址格式

在上一节介绍了如何创建和关闭套接字，应用程序要使用创建好的套接字进行数据传送和接收，大多数套接字函数都需要给出数据包发送/接收的目标地址。套接字的实现独立于网络协议，可以支持多个协议栈的 API。不同网络协议族都定义了自己的地址格式，而且数据、地址在网络上传送与在主机中传送的顺序也有不同。所以在进行套接字网络编程的进一步讨论之前，先介绍如何标识我们想与之通信的进程地址。在使用套接字进行网络通信时，识别与自身进程通信的另一进程需要以下两个部分信息。

- 主机网络地址：用于识别我们想与之通信的、连接在网络上的主机。
- 服务：就是运行在主机上的某个进程，该进程就是本进程要与之通信的另一个进程。

11.2.1 字节顺序

如果在同一主机的进程之间进行通信，就不需要考虑字节顺序的问题。字节顺序与处理器的体系结构有关，表示的是大型数据类型在内存中的存放顺序。例如，现有一个整型数（integer），图 11-3 给出了一个 32 位整数的两种不同字节顺序在内存中的存放格式。

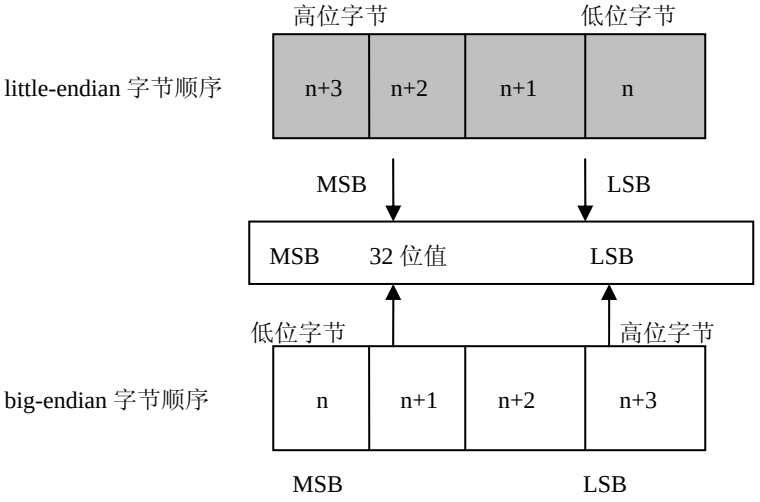


图 11-3　32 位整数的两种不同字节顺序格式

如果处理器的体系结构支持 big-endian 字节顺序，则高位字节地址在 least significant byte（LSB）。支持 Little-endian 字节顺序的处理器体系结构正好相反，least significant byte

中包含的是低位字节数据。注意，无论 CPU 支持什么字节顺序，the most significant byte（MSB）总是在左端，least significant byte 总是在右端。因此，如果我们分配了一个 32 位的整数，其值为 0x05030209，无论字节顺序是什么，则最高位字节中存放的应该是 5，最低位字节中存放的应该是 9。如果用一个强制类型转换的字符指针（cp：character pointer）来寻址整数的地址，就会看到字节因为顺序不同而带来的差异。在支持 little-endian 字节顺序的处理器中，cp[0]指向最低位字节，其中包含的数值为 9；cp[3]指向最高位字节，其中包含的数值是 5；相反在 big-endian 字节顺序处理器中，cp[0]中包含的值为 5，指向最高位字节，而 cp[3]中包含的是数值 9，指向最低位字节。

网络协议有自身特定的字节顺序，这样不同的计算机系统可以任意交换网络数据，而不必关心网络字节顺序。TCP/IP 协议栈使用 big-endian 字节顺序。对于 TCP/IP 协议栈，地址以网络字节顺序表示，所以应用程序有时需要在以处理器字节顺序存放的地址信息与网络字节顺序存放的地址信息之间进行转换。

套接字库函数提供了 4 个常用函数，为 TCP/IP 应用在处理器字节顺序和网络字节顺序之间进行转换。

```
#include <arpa/inet.>
//将 32 位的主机字节顺序转换成网络字节顺序
uint32_t htonl ( uint32_t  hostint32 );
//将 16 位整数的主机字节顺序转换成网络字节顺序
uint16_t htons ( uint16_t  hostint16 )
//将 32 位整数的网络字节顺序转换成主机字节顺序
uint32_t ntohl ( uint32_t  netint32) ;
//将 16 位整数的网络字节顺序转换成主机字节顺序
uint16_t ntohs ( uint16_t  netint16 ) ;
```

在以上函数中“h”表示“host”字节顺序，即主机字节顺序，“n”表示“network”字节顺序，即网络字节顺序。“l”表示“long”(即 4 字节)整数，“s”表示“short”（即 2 字节整数）。所以“htonl”即“host to network long”：长整型数，主机字节顺序转换成网络字节顺序。这 4 个函数定义在<arpa/inet.h>头文件中。

在应用程序开发中使用这些函数时，不需要关心实际的值在主机中的字节顺序与在网络中的字节顺序。需要做的是，调用正确的函数将给定数据在主机字节顺序与网络字节顺序之间进行转换，对于字节顺序与 TCP/IP 协议栈字节顺序相同的处理器，以上 4 个函数通常以 NULL 宏定义。

11.2.2　地址结构

一个地址标识了某个通信协议域中套接字应用程序运行的站点，不同协议域的地址格式不同。所以不同格式的地址传给套接字函数时，地址被强制转换成通用地址格式 sockaddr，通用网络地址格式 sockaddr 结构如下：

```
struct sockaddr {
   sa_family_t      sa_family;       /* 地址族*/
    char            sa_data[] ;      /* 网络地址，长度可变*/
....
};
```

各操作系统对以上地址格式实现时可自由地在 sa_data 成员中加入额外的数据成员。例

如在 Linux 系统，以上地址结构的定义为：

```
struct sockaddr {
  sa_family_t      sa_family ;                /* 地址族 */
  char             sa_data[14] ;              /* 地址，长度可变*/
};
```

但在 FreeBSD 系统上，以上地址结构就定义为：

```
struct sockaddr {
  unsigned char    sa_len;                    /* 地址结构的总长度 */
  sa_family_t      sa_family;                 /*  地址族  */
  char             sa_data[14]                /* 地址，长度可变  */
};
```

1．IPv4 使用的地址结构

在 Linux 中支持 Internet 地址的数据结构定义在<netinet/in.h>库文件中。对于 IPv4 Internet 域（AF_INET），套接字的地址由 struct sockaddr_in 数据结构描述：

```
struct in_addr {                                                    <netinet/in.h>
  in_addr_t             s_addr;              /*IPv4 的 IP 地址  */
};

struct sockaddr_in {
  sa_family_t           sin_family;          /*地址族*/
  in_port_t                   sin_port ;     /* 端口号 */
  struct in_addr              sin_addr ;     / *IPv4 网络地址*/
unsigned char           sin_zero[8];         /* 填充字节 */
};
```

in_port_t 数据类型的定义为 uint16_t，in_addr_t 数据类型的定义为 uint32_t。 这些整型数据类型指明了这些数据类型所占的字节数，以上整数类型定义在<stdint.h>库文件中。sin_zero 成员是填充字段，并不是指其所有值要设置为 0。

2．IPv6 协议使用的地址结构

在 AF_INET 地址族中，Linux 中支持的 IPv6 Internet 协议域（AF_INET6），套接字地址由 struct sockaddr_in6 数据结构描述：

```
struct in6_addr {
  uint8_t               s6_addr[16]          /* IPv6 地址 */
};

struct sockaddr_in6 {
  sa_family_t           sin6_family;         /*地址族*/
  in_port_t                   sin6_port ;    /* 端口号 */
  uint32_t                    sin6_flowinfo ;/* 流量类和流控制信息*/
  struct in6_addr       sin6_addr ;          /*Ipv6 网络地址*/
  uint32_t                    sin6_scope_id; /* 符合范围的接口集 */
};
```

虽然 struct sockaddr_in 和 struct sockaddr_in6 数据结构的差别很大，但当它们传给套接字例程时，都会强制转换成 struct sockaddr 数据结构。

11.2.3　支持地址格式转换的函数

有时，套接字的地址信息需要在显示终端上打印出来显示给操作人员看，这时就需要以操作人员习惯的地址格式表示，而不是以机器的地址格式表示。Linux 标准网络库函数中提供了 inet_addr 和 inet_ntoa 函数，来将二进制的地址格式转换成字符串格式的地址格式：十进制数，以圆点隔开的形式（即 192.168.112.96）。这些函数只对 IPv4 地址格式有效。另外两个新的库函数 inet_ntop 和 inet_pton 的功能与 inet_addr 和 inet_ntoa 函数的功能类似，但 inet_ntop 和 inet_pton 同时支持 IPv4 和 IPv6 地址格式。

1. 支持 IPv4 地址格式转换的函数

```
#include <arpa/inet.h>
//将字符串格式的网络地址转换成二进制格式的网络地址
int inet_aton ( const char * strptr, struct in_addr * addrptr)
//如果字符串有效，则返回 1，如果出错则返回 0
//返回 32 位二进制网络字节顺度的 IPv4 网络地址
in_addr_t  inet_addr ( const char *strptr);
//返回指向十进制，以圆点隔开的字符串网络地址的指针
char *inet_ntoa (struct in_addr inaddr)
```

2．同时支持 IPv6 和 IPv4 地址格式转换的函数

```
#include <arpa/inet.h>
const char *inet_ntop( int domain, const void * addr, char *str, socklen_t size );
//函数执行成功，返回指向存放地址的字符串的指针，如果不成功则返回 NULL
int inet_pton( int domain, const char *str, void addr);
//函数执行成功，返回 1，地址无效返回 0
```

inet_ntop 函数将网络字节顺序的二进制地址转换成字符串；inet_pton 函数将以字符串表示的地址格式转换成网络字节顺序的二进制地址格式。以上函数只支持两个地址域 AF_INET 和 AF_INET6。

在 inet_ntop 函数中，size 参数指定的是存放地址字符串的缓冲区大小。系统中定义了两个常数来描述地址缓冲区长度，使编程工作简单许多。

- INET_ADDRSTRLEN：存放 IPv4 地址字符串的长度，该长度足够存放 IPv4 格式的地址。
- INET6_ADDRSTRLEN：为存放 IPv6 地址字符串的长度，该长度足够存放 IPv6 格式的地址。

在 inet_pton 函数中，存放地址的缓冲区是 addr，当协议域是 AF_INET 时，addr 缓冲区的大小需足够存放一个 32 位的地址。当协议域为 AF_INET6 时，addr 缓冲区的大小应能存放一个 128 位地址。

11.2.4　获取网络配置信息

按常规，应用程序不需要知道套接字内部的地址结构。如果一个应用程序只是简单地将套接字地址设置为 sockaddr 结构而不关心任何网络协议的特性，应用程序就可以与多个协议交互，在各种网络协议基础上提供相同的服务。

Linux 网络库函数提供了一套 API 来访问各种网络配置信息。由这些函数返回的网络配置信息可以保存在许多地方。可以保存在静态文件中（如/etc/hosts，/etc/services 等文件

中），它们也可以由名字服务器来管理，如 DNS 服务器（Domain Name System）或 NIS 服务器（Network Information Service）来管理。无论这些配置信息保存在哪里，都可以用以下的函数来访问这些网络配置信息。

1．访问网络配置信息的 API

```
#include <netdb.h>
//如果函数执行成功，则返回指向主机数据库文件的指针；如果函数执行不成功则返回 NULL
struct hostent  *gethostent ( void );
//打开主机数据库文件，将指针放在上次操作后的位置，如果文件已打开，则将文件指针放在文件起始处
void   sethostent ( int  stayopen);
//关闭主机数据库文件
void   endhostent( void );
```

（1）gethostent 函数说明

gethostent 函数的功能是获取主机中存放的网络配置库文件的内容；如果主机的数据库文件还没有打开，则 gethostent 会打开该文件。gethostent 函数返回文件中的当前记录。

gethostent 返回后，获取一个指向 struct hostent 数据结构的指针，该指针指向一个静态数据缓冲区，该缓冲区的内容在每次调用 gethostent 函数时都会被改写。

（2）struct hostent 数据结构

struct hostent 数据结构中包含以下数据成员：

```
struct hostent {
   char  *h_name;                    /* 主机名  */
   char  **h_aliases;                /* 指针，指向可替换的主机名数组 */
   int         h_addrtype;           /* 地址类型 */
   int         h_length              /* 地址的长度，按字节计算 */
   char  **h_addr_list;              /* 指向存放网络地址的数组*/
   …
};
```

返回到 struct hostent 数据函数中的地址，按网络字节顺序存放。

（3）sethostent 与 endhostent 函数

sethostent 函数会打开主机的数据库文件；如果文件已打开，则 sethostent 函数绕回文件头。endhostent 函数会关闭主机的数据库文件。

2．获取设备名与设备号信息

可以通过以下函数获取网络设备名和网络设备接口索引号。

```
#include  < netdb.h >
struct netent      *getnetbyaddr ( uint32_t net, int type );
struct netent      *getnetbyname (  const char  *name  );
struct netent *getnetent ( void );
//如果函数执行成功则返回指针，如果不成功则返回 NULL
void               setnetent ( int  stayopen );
void               endnetent ( void );
```

（1）struct netent 数据结构

函数返回信息存放在 struct netent 数据结构类型的变量中，struct netent 数据结构包含以下数据域：

```
struct netent {
   char            *n_name;                  /* 网络设备名 */
   char            **n_aliases;              /* 指向可替换的网络设备名数组 */
```

```
    int             n_addrtype;      /* 地址类型 */
    uint32_t        n_net;           /* 网络设备索引号 */
    …
};
```

返回的网络设备索引号（n_net）按网络字节顺序存放，地址类型（n_addrtype）是某个地址族常数（如 AF_INET）。

（2）协议索引号与协议名的对映

可以用以下函数将协议名和协议索引号之间映射：

```
#include < netdb.h >
struct protoent                     *getprotobyname ( const char *name);
struct protoent                     *getprotobynumber ( int proto );
struct protoent                     *getprotoent ( void );
//函数执行成功，则返回指向 struct protoent 数据结构的指针，如果失败则返回 NULL
void setprotoend ( int stay open );
void endprotoend ( void );
```

函数返回信息存放在 struct protoent 数据结构类型变量中。

（3）struct protoent 数据结构

struct protoent 数据结构包含了以下数据域：

```
struct protoent {
   char  *p_name;                   /* 协议名 */
   char  **p_aliases;               /*  指向可替换的协议名的数组 */
   int   p_proto;                   /*  协议编码 */
   …
};
```

3．获取应用服务程序信息

应用程序提供的服务由协议网络地址的端口号部分标识，每个服务都有一个唯一的端口号。我们可以调用 getservbyname 函数将服务名与端口号对应，调用 getservbyport 函数将端口号与服务名对应，或调用 getservent 函数在服务数据库中顺序扫描查询，获取服务的信息。setprotoend 与 endprotoend 函数可以分别用于打开服务数据库文件、关闭服务数据库文件。

```
#include < netdb.h >
struct  servent    *getservbyname   (const char *name, const char *proto);
struct servent           *getservbyport ( int port, const char *proto );
struct  servent    *getservent ( void );
//如果函数执行成功，则返回指向 struct servent 数据结构的指针，如果失败则返回 NULL
void          setservent ( int stayopen );

void          endservent ( void );
```

函数返回的服务信息存放在 struct servent 数据结构类型的变量中。

struct servent 数据结构包含的数据成员如下：

```
struct servent {
   char        *s_name;             /* 服务名 */
   char        **s_aliases;         /* 指向可替换的服务名数组的指针 */
   int         s_port;              /* 端口号  */
   char        *s_proto;            /* 协议名 */
   …
};
```

4．主机名、服务名与网络地址的映射

应用程序可以将主机名和服务名映射到网络地址上，或将网络地址映射为主机名与服务名。getaddrinfo 函数允许应用程序将主机名和服务名映射到网络地址上。

```
#include < sys/socket.h >
#include < netdb.h >

int getaddrinfo ( const char *host,
                        const char *service,
                        const struct addrinfo * hint,
                        struct addrinfo  **res );
//如果函数执行成功则返回0，如果不成功则返回错误代码
void freeaddrinfo( struct addrinfo *ai );
```

在调用函数时，需要提供主机名、服务名或两者都提供作为函数的输入参数。如果只提供了其中一个，另一个指针应为空指针 NULL。主机名既可以为节点名，也可以是字符串格式表示的（圆点分隔的十进制主机地址）网络地址。

（1）struct addrinfo 数据结构

getaddrinfo 函数返回 struct addrinfo 数据结构链接起来的地址链表，struct addrinfo 数据结构的数据成员为：

```
struct addrinfo{
  int              ai_flags;              /* 标志，定制活动标志*/
  int              ai_family;             /* 地址族 */
  int              ai_socktype;           /* 套接字类型 */
  int              ai_protocol;           /* 协议 */
  socklen_t        ai_addrlen;            /* 地址长度，按字节计算 */
  struct sockaddr *ai_addr;               /* 地址 */
  char        *ai_canonname;              /* 主机名 */
  struct addrinfo *ai_next;               /* 链表中的下一个成员*/
  …
};
```

我们可以使用 freeaddrinfo 函数来释放一个或多个 getaddrinfo 函数返回的地址结构 struct addrinfo，最多能释放的结构数量取决于 struct addrinfo 数据结构中 ai_next 数据域中链接了多少个地址信息。

（2）getaddrinfo 函数的 hint 参数

在调用 getaddrinfo 函数时，可以提供一个选项 hint 来过滤地址，以满足某个特定的标准，hint 是用来过滤地址的模板，只用于 struct addrinfo 数据结构中的 ai_family、ai_flags、ai_protocol 和 ai_socktype 数据域。其他的整数域需保持为 0，指针域必须为 NULL。表 11-4 总结了标志数据域及其含义。

表 11-4　struct addrinfo 数据结构的标志数据域

flags	描　　述
AI_ADDRCONFIG	查询哪个地址类型的配置（IPv4 或 IPv6）
AI_ALL	查询 IPv4 和 IPv6 地址配置（只与 AI_V4MAPPED）
AI_CANONNAME	请求一个主机名（相反是别名）
AI_NUMBERICHOST	返回以数字格式的主机地址
AI_NUMERICSERV	返回服务的端口号
AI_PASSIVE	与 listen 绑定的内部套接字地址
AI_V4MAPPED	如果没有找到 IPv6 地址，则返回 IPv4 地址，并将其映射为 IPv6 格式

（3）getaddrinfo 函数错误代码处理

如果 getaddrinfo 函数执行失败，则不能使用 perror 或 strerror 来产生错误信息，而是需要调用 gai_strerror 函数来将错误代码转换成错误消息。

```
#include < netdb.h >
const char *gai_strerror ( int error );
//返回指向存放错误字符串的指针
```

5．将地址转换为主机名与服务名

getnameinfo 函数将地址转换成主机名和服务名。getnameinfo 函数实现的功能是，将套接字地址（addr）转换成主机名和服务名。

如果参数 host 为非空指针，则它指向一个长度为 hostlen 字节的缓冲区，该缓冲区用于存放主机名。同样，如果 service 为非空指针，则它指向一个长度为 servlen 字节的缓冲区，该缓冲区用于存放服务名。

```
#include < sys/ socket.h >
#include < netdb.h>

//如果函数执行成功则返回 0，否则返回非零的错误代码
int getnameinfo ( const struct sockaddr * addr,
                  socklen_t alen, char * host,
                  socklen_t        hostlen, char *  service,
                  socklen_t   servlen, unsigned int flags );
```

flags 参数用于指明如何转换这些信息，表 11-5 总结了 flags 的值及其含义。

表 11-5　getnameinfo 函数的标志 flags 值

flags	描　述
NI_DGRAM	服务是数据报（datagram）类型，而不是字节流（stream）类型
NI_NAMEREQD	如果找不到主机名，则作为错误处理
NI_NOFQDN	对本地主机，只返回全部域名的主机名部分
NI_NUMERICHOST	返回数字格式的主机地址，而不是主机名
NI_NUMERICSERV	返回服务地址的数字部分（端口号），而不是服务名

11.2.5　编程示例

以下是使用 getaddrinfo 函数获取网络配置信息的实例，如果在主机上的某个服务程序可以支持多个协议，则程序会打印出多个服务的名称。在该例中，我们只打印支持 IPv4 协议的地址信息，即 ai_family 值为 AF_INET。如果应用要严格限制只输出 AF_INET 协议族的地址，则可以在 hint 数据域中设置对地址族 ai_family 的过滤。

```
//打印主机和服务信息的编程示例
#include <netdb.h>
#include < arpa/inet.h>
#include <sys/socket.h>
#include <netinet/in.h>
//按 struct addrinfo 数据结构中 ai_family 数据域的值打印地址族信息函数
void print_addr_family(struct addrinfo *afp)
{
   printf(" 打印地址族信息\n ");
   switch (afp->ai_family) {
```

```
    case AF_INET:
        printf("地址族为：INET");
        break;
    case AF_INET6:
        printf("地址族为：INET6");
        break;
    case AF_UNIX:
        printf("地址族为：UNIX");
        break;
    case AF_UNSPEC:
        printf("地址族为：通用网络地址");
        break;
    default:
        printf("地址族未知");
    }
}
//根据 struct addrinfo 数据结构中 ai_socktype 数据域的值打印套接字类型信息
void    print_type_info(struct addrinfo *afp)
{
    printf(" 套接字类型信息显示\n ");
    switch (afp->ai_socktype) {
    case SOCK_STREAM:
        printf("字节流套接字：stream");
        break;
    case SOCK_DGRAM:
        printf("数据报套接字：datagram");
        break;
    case SOCK_SEQPACKET:
        printf("顺序数据包套接字：seqpacket");
        break;
    case SOCK_RAW:
        printf("裸套接字：raw");
        break;
    default:
        printf("套接字类型未知 (%d)", afp->ai_socktype);
    }
}
//根据 struct addrinfo 数据结构中 protocol 数域的值打印套接字使用的网络协议
void print_protocol_info(struct addrinfo *afp)
{
    printf("打印套接字使用的协议信息\n");
    switch (afp->ai_protocol) {
    case 0:
        printf("套接字使用默认网络协议");
        break;
    case IPPROTO_TCP:
        printf("套接字使用 TCP 网络协议");
        break;
    case IPPROTO_UDP:
        printf("套接字使用 UDP 网络协议");
        break;
    case IPPROTO_RAW:
        printf("套接字使用裸网络协议");
        break;
    default:
        printf("unknown (%d)", aip->ai_protocol);
    }
}
//根据 struct addrinfo 数据结构中 ai_flags 数据域的值打印当前查询的网络信息类型
```

```
void print_flags_info(struct addrinfo *afp)
{
   printf("当前查询的网络信息类型\n");
   if (afp->ai_flags == 0) {
      printf(" 0");

   } else {
      if (afp ->ai_flags & AI_PASSIVE)
         printf(" 与侦听套接字绑定的地址");
      if (afp ->ai_flags & AI_CANONNAME)
         printf(" 查询主机名");
      if (afp ->ai_flags & AI_NUMERICHOST)
         printf(" 查询主机地址");
#if defined(AI_NUMERICSERV)
      if (afp ->ai_flags & AI_NUMERICSERV)
         printf(" 查询服务端口号");
#endif
#if defined(AI_V4MAPPED)
      if (afp ->ai_flags & AI_V4MAPPED)
         printf(" 以 IPv4 映射的 IPv6 地址");
#endif
#if defined(AI_ALL)
      if (afp ->ai_flags & AI_ALL)
         printf(" IPv4 与 IPv6 地址配置");
#endif
   }
}
//示例主程序
int main(int argc, char *argv[])
{
   struct addrinfo    *ailist, *afp;
   struct addrinfo    addrfilter;
   struct sockaddr_in  *sinp;
   const char        *addr;
   int             err;
   char             abuf[INET_ADDRSTRLEN];

   if (argc != 3){
     printf("使用: %s 主机服务", argv[0]);
  exit(1);
  }
//查询主机名信息
   addrfilter.ai_flags = AI_CANONNAME;
   addrfilter.ai_family = 0;
   addrfilter.ai_socktype = 0;
   addrfilter.ai_protocol = 0;
   addrfilter.ai_addrlen = 0;
   addrfilter.ai_canonname = NULL;
   addrfilter.ai_addr = NULL;
   addrfilter.ai_next = NULL;
   if ((err = getaddrinfo(argv[1], argv [2], &hint, &ailist)) != 0)
      exit(-1);
   for (afp = ailist; afp!= NULL; afp = afp ->ai_next) {
      print_flags_info(afp);
      print_addr_info(afp);
      print_type_family(afp);
      print_protocol_info(afp);
      printf("\n\thost %s", ailist->ai_canonname? ailist->ai_canonname:"-");
      if (afp ->ai_family == AF_INET) {
```

```
            sinp = (struct sockaddr_in *) afp ->ai_addr;
            addr = inet_ntop(AF_INET, &sinp->sin_addr, abuf,
               INET_ADDRSTRLEN);
            printf(" address %s", addr?addr:"unknown");
            printf(" port %d", ntohs(sinp->sin_port));
         }
         printf("\n");
      }
```

11.2.6 将地址与套接字绑定

在将客户端的套接字与地址相关联时，可以让系统为套接字选择一个默认地址。对于服务器，我们需要为服务器套接字分配相关的地址，使从客户端来的请求能够到达服务器上的服务。客户端需要用某种方式查询到与服务器连接的地址，最简单的机制就是为服务器预留一个地址，并把它注册在/etc/services 中或以服务名存放在文件中。

1．bind 函数的使用方法

应用程序使用 bind 函数把地址与套接字相绑定。

```
#include < sys/socket.h >
//如果地址绑定成功，则函数返回0，否则返回错误代码
int bind ( int sockfd, const struct sockaddr *addr, socklen_len );
```

（1）对网络地址的限制

可以使用的网络地址有以下几个限制：

- 指定的网络地址必须在服务进程运行的主机上有效，不能指定一个其他主机的地址。
- 地址格式必须与我们创建的套接字的地址族相匹配。
- 地址中的端口号不能小于 1024，除非进程有一定的优先权。
- 通常，站点上只有一个套接字可与给定的地址绑定。

（2）INADDR_ANY

对于 Internet 协议域，如果给定的是一个特殊的 IP 地址 INADDR_ANY，站点套接字就可以与系统中所有的网络接口绑定。这表明我们可以从系统中任何网络接口上接收数据包。在下一节中会介绍系统可以选择一个地址，把地址与套接字绑定，如果我们事先将一个地址与套接字绑定，则可以调用 connect 或 listen 在绑定的套接字上建立连接，侦听该连接上的请求。

以下是一个与主机所有地址、端口绑定的套接字的示例。

```
#include <sys/socket.h>

int net_conn ( )
{
  int sd, rc;
  struct sockaddr_in local_server;

  sd = socket( AF_INET, SOCK_STREAM, 0 );
  if ( sd < 0 )
    exit( -1 );
  local_server.sin_family = AF_INET;
  local_server.sin_addr.s_addr = htonl( INADDR_ANY);
```

```
    local_server.sin_port = htons( 0 );

    rc = bind( sd, (struct sockaddr *) & local_server, sizeof( struct sockaddr_in));
    if ( rc < 0 ) {
          printf( "%s","地址绑定失败\n");
       exit(-1 );
  }
  return sd;
  }
```

2．获取绑定到套接字上的地址信息

在应用程序中，可以使用 getsockname 函数来查询绑定到套接字上的地址。

```
#include < sys/socket.h >
//如果函数执行成功则返回0，如果失败则返回错误代码
int getsockname ( int sockfd , struct sockaddr * addr, socklen_t * alenp );
```

在调用 getsockname 函数时，设置 alenp 指针指向一个整数，该整数是存放 sockaddr 的缓冲区大小。当函数返回时，返回的整数为套接字地址的大小。如果地址与缓冲区的大小不匹配，则地址的部分值会被截掉。如果当前没有地址与套接字绑定，则函数执行结果不确定。

3．获取裸地址信息

如果套接字是与裸地址相连的，则可以调用 getpeername 函数来发现地址。

```
#include < sys/socket.h >
int getpeername ( int sockfd, struct sockaddr * addr, socklen_t * alenp );
```

11.3　套接字连接

如果应用程序处理的是面向连接的网络服务（ SOCK_STREAM 或 SOCK_SEQPACKET），在交换数据之前，需要在请求连接服务的进程（ 客户 ）与提供服务的进程（服务器）之间建立连接。

11.3.1　connect 函数分析

建立客户/服务器套接字之间的连接，由 connect 函数来完成。

```
#include < sys/socket.h >
//如果连接成功，则函数返回连接的套接字的描述符。否则返回负值的错误代码
int connect ( int sockfd, const struct sockaddr *addr, socklen_t len )
```

1．connect 函数的调用说明

该函数用于客户端的套接字向服务端套接字发出连接请求。给 connect 函数传递的参数有以下几种。

- sockfd：为客户端事先创建的套接字返回的套接字描述符。如果 sockfd 事先没有与地址绑定，connect 函数会将其与一个默认的地址绑定。
- Addr：地址参数，是服务器端套接字的地址，客户端进程请求与该服务器连接。

- Len：服务器端套接字地址的长度。

当客户进程试着连接一个服务器时，connect 函数的请求可能会失败，原因如下：

- 客户端要连接的主机必须运行并上线，主机上服务进程已启动。
- 主机上的服务器套接字必须与客户端进程要连接的套接字地址绑定。
- 在服务器的连接队列中还有空间接收新的连接。

2．使用 connect 编程示例

客户应用程序要能处理 connect 函数返回的错误代码。下例中给出了如何处理 connect 错误的示例。出现连接错误可能是因为服务器的负载太重，如果调用 connect 失败，进程短暂休眠后再试着连接服务器，每次尝试连接的延迟时间都会增长一点直至达到最大延迟时间。

```
//重复连接服务器示例
#include < sys/socket.h >
#define MAXSLEEP 256
int connect_repeat ( int sd, const struct sockaddr *serv_addr, socklen_t len )
{
  int wait_time;

  for ( wait_time = 1; wait_time <= MAXSLEEP; wait_time <<= 1 ) {
        if ( connect ( sd, serv_addr , len ) == 0 ) {
///如果建立连接成功
              return ( 0 );
        }
//建立连接失败，延迟一段时间后再尝试新的连接
if ( wait_time <= MAXSLEEP/2 )
              sleep ( wait_time);
        }
        return ( -1 );
}
```


3．connect 函数的其他使用方式

如果套接字描述符为非阻塞模式的套接字（非阻塞套接字在后面章节中讨论），在连接不能立即建立时，connect 函数将返回 1，errno 将设置为一个特殊的错误代码 EINPROGRESS。应用程序可以使用 poll 或 select 来确定文件描述符什么时候有效，且连接已建立，可以数据传送操作。

connect 函数也可以用于无连接网络服务（SOCK_DGRAM）。在无连接的网络服务中，使用 connect 函数可以优化网络的传送。如果将 connect 函数用于 SOCK_DGRAM 类无连接套接字，传送的所有数据都会送到 connect 函数中指定的目标地址，这样就不需要在每次传送数据时都提供传送的目标地址。另外，客户端程序也只能从 connect 指定的地址接收数据报。

11.3.2 服务器套接字建立侦听队列

在服务器端，接收请求的服务器进程调用 listen 函数来侦听是否有客户端传来的连接请求，服务器进程一旦调用了 listen 函数来侦听套接字，表明服务器已正确启动，可以开始接收连接请求。

```
#include < sys/socket.h >
//如果函数执行成功则返回0，如果失败则返回错误代码
int listen ( int sd , int  number );
```

listen 函数的各输入参数如下。

- Sd：套接字描述符。服务器在打开的套接字 sd 上侦听来自客户端进程的连接请求。
- Number：告诉服务器进程侦听队列可以接收多少个连接请求，实际的值由系统确定。服务器进程实际可接收的连接数量上限由 SOMAXCONN 确定，该常数定义在< sys/socket.h >文件中。不同网络协议的最大连接数不同，例如，TCP 协议套接字，服务器能接收的最大连接数的默认值是 128。

一旦服务器套接字上侦听队列的连接请求已满，系统将拒绝新的连接请求，所以 number 值的选择要基于服务器的负载情况，以及一次必须接收的连接请求的进程数。

11.3.3　建立套接字连接

服务器调用了 listen 函数后，服务器进程就可以接收客户端的连接请求。当客户端的连接请求到达时，服务器进程可调用 accept 函数来提取套接字连接等待队列的连接请求，并将服务器状态切换到已连接状态。

```
#include < sys/socket.h >
//如果函数执行成功，则返回建立连接的套接字的文件描述符，否则返回错误代码
int accept ( int sd, struct sockaddr * addr, socklen_t *len );
```

accept 函数的输入参数如下。

- sd：接收客户端建立连接的侦听套接字 sd（客户端调用 connect 函数请求连接）。
- addr：存放客户端的套标地址的缓冲区。
- len：存放客户端的套标地址的缓冲区长度。

在 accept 函数执行后，返回一个新的套接字，这个新的套接字描述符与最初服务器创建套接字时，设置的套接字地址族与套接字类型、使用的协议一样。服务器原来创建的套接字不与连接关联，它继续在原套接字上侦听，以便接收其他连接请求。

如果服务器进程不关心客户端的标识符，可以将地址 addr 和长度 len 设置为 NULL，否则在调用 accept 函数以前，需要将 addr 参数设置为一个缓冲区的首地址。该缓冲区用于存放与服务器建立了连接的客户端的地址，整数指针 len 指向的缓冲区，用于存放客户端地址的长度。当 accept 函数返回时，会将客户端的地址和长度信息填入 addr 缓冲区和 len 缓冲区。

如果服务器的侦听队列中没有连接请求，accept 函数将会阻塞直到有一个新的连接请求到达。在创建套接字时，如果将 sd 套接字描述符设置为非阻塞模式，在没有连接请求时，accept 将返回 1，并将 errno 设置为 EAGAIN 或 EWOULDBLOCK。

在服务器调用 accept 函数后，如果没有连接请求出现，服务器将阻塞直到有一个连接请求到达后被唤醒。相反一个服务器可以使用 poll 或 select 来等待连接请求的到达。在这种情况下，在一个套接字上出现的连接请求就为可读。

```
//创建用于面向连接网络服务的套接字，在打开的套接字上侦听
#include <errno.h>
#include <sys/socket.h>
```

```
int listen_server(int type, struct sockaddr *addr, socklen_t len,
  int qlen)
{
   int sd;
   int ret = 0;
//调用 socket 函数创建套接字，如果创建套接字失败，则返回
   if ((sd = socket(addr->sa_family, type, 0)) < 0)
      return(-1);
//将套接字与地址绑定
   if (bind(sd, addr, len) < 0) {
      ret = errno;
      goto errout;
}
//如果创建的套接字是面向连接的套接字，在创建的套接字上侦听
   if (type == SOCK_STREAM || type == SOCK_SEQPACKET) {
      if (listen(sd, qlen) < 0) {
         ret = errno;
         goto errout;
      }
   }
   return(sd);
errout:
   close(sd);
   errno = ret;
   return(-1);
}
```


11.4 数据的传送

一个套接字由文件描述符代表，应用程序人员可以使用 read 和 write 函数来与套接字通信。数据报（datagram）套接字也可以建立连接，如果我们使用 connect 函数在打开的套接字上建立与默认地址的连接，在套接字描述符上用 read 和 write 来操作就非常有意义，这样可以将套接字描述符传递给本地文件。

虽然应用程序可以使用 read 和 write 函数在套接字上交换数据，这两个函数完成的功能就只是读/写数据的常规操作。如果应用程序需要给套接字指定选项，则从多个客户端接收数据包或传送 out-of-band 数据的数据包，就需要使用专门用于套接字传送数据的函数。

在套接字应用中，3 个函数用于数据传送；3 个函数用于数据接收。首先介绍发送数据函数。

11.4.1 send 函数

套接字应用中最简单的传送函数是 send，send 函数的作用类似于 write，但 send 函数允许应用程序指定标志，规定如何对待传送数据。

```
#include < sys/socket.h >
//如果函数执行成功，则返回套接字成功传送的字节数，否则返回错误代码
ssize_t send ( int sd, void * buff, size_t nbytes, int flags );
```

send 函数的输入参数说明如下。

- sd：发送数据的套接字描述符。
- buff： buff 参数指向一个缓冲区的首地址，该缓冲区用于存放要发送的数据。
- nbytes：为要发送数据的字节数。
- flags：标志位，指明如何传送数据。flags 的有效值如表 11-6 所示。

表 11-6　send 的 flags 标志位取值及含义。

flags	描　　述
MSG_DONTROUTE	数据包离开局域网后，不对数据包进行路由
MSG_DONTWAIT	允许不阻塞操作（相当于在文件操作中使用 O_NONBLOCK）
MSG_EOR	表明传送记录结束，如果该标志由协议支持
MSG_OOB	如果协议支持，发送 out-of-band 数据

send 函数返回发送成功，并不意味着在连接的另一端的进程可以收到数据，这里只能保证发送 send 函数执行成功，发送给网络设备驱动程序的数据没有出错。

发送数据有边界划分的网络协议，如果发送的信息总长度超过协议能支持的最大值，send 函数的执行将会失败， errno 错误代码设置为 EMSGSIZE。支持字节流传送的网络协议（byte-stream），send 函数在所有数据发送完成之前将阻塞。

11.4.2　传送数据的函数

实现网络数据发送的另外两个函数分别为 sendto、sendmsg。

1．sendto 函数

sendto 函数完成的功能与 send 类似，sendto 函数与 send 函数的不同之处是，它可以指定数据发送的目标地址，该函数可用于无连接的套接字。

```
   #include < sys/socket.h >
   //如果函数执行成功，则返回发送的字节数，否则返回错误代码
   ssize_t  sendto ( int sd, void *buf, size_t nbytes, int flags, struct sockaddr *destaddr, socklen_t
destlen );
```

一个面向连接的套接字，在传送数据之前已调用 connect、listen、accept 等函数在客户端/服务器套接字上建立了连接，这些套接字事先与地址绑定，所以面向连接的套接字在数据传送时可以忽略目标地址，目标地址已由连接指定。

无连接的套接字，不能调用 send 函数来传送数据，除非事先调用了 connect 函数设置了目标地址，而 sendto 函数给出了另一个实现发送数据的方式。

2．sendmsg 函数

在发送数据时，还有另一个选择，可以在套接字上实现数据传送，即 sendmsg 函数，用 struct msghdr 数据结构来指定 sendmsg 函数传送的多个数据缓冲区。

```
   #include < sys/socket.h >
   //如果函数执行成功将返回发送的字节数，否则返回错误代码
   ssize_t sendmsg ( int sd , struct msghdr *msg, int  flags );
```

struct msghdr 数据结构的定义在第 9 章中已介绍，这里就不在赘述。

11.4.3 接收数据的函数

recv 函数与文件读 read 函数类似，recv 函数中可以指定标志来控制如何接收数据。

```
#include < sys/socket.h>
//函数返回读到的字节数，如果套接字中没有有效数据则返回 0，或者发送方站点结束发送后顺序关闭站点，函数也返回 0。否则
//返回错误代码
ssize_t recv ( int sd , void * buf, size_t nbytes, int flags );
```

recv 函数的输入参数如下。

- sd：接收数据的套接字描述符。
- buf：指向用于接收数据的缓冲区。
- nbytes：为接收到的数据字节数。
- flags：为标志位，指明了函数应如何处理接收数据。表 11-7 为 flags 有效值

表 11-7 recv 的 flags 有效值

flags	描 述
MSG_OOB	如果协议支持，提取 out-of-band 数据
MSG_PEEK	返回数据包的内容，但不消耗队列中的数据包
MSG_TRUNC	返回数据包的实际长度，即使数据包的长度被截短
MSG_WAITALL	等到所有数据到齐（只对 SOCK_STREM 有效）

当应用程序给 recv 函数指定了 MSG_PEEK 标志后，应用程序可以看到下一个接收到的数据，但并不提取队列中的数据，下一次 recv 函数执行就可以读到已看到的数据。

面向连接的 SOCK_STREAM 类套接字，接收到的数据可以比实际要求的数据少。设置 MSG_WAITALL 标志可以阻止这种情况的发生，让 recv 函数将所有要求的数据都接收到后，recv 函数才返回。

对于 SOCK_DGRAM 类套接字和 SOCK_SEQPACKET 类套接字，设置 MSG_WAITALL 标志对接收活动没有影响。因为这些套接字基于消息，在一次读操作过程中会返回整个消息数据包。

如果发送方调用 shutdown 函数来结束传送过程，或者网络协议默认支持顺序 shutdown，随后发送方关闭套接字，recv 函数在收到所有数据后会返回 0。

11.4.4 recvfrom、recvmsg 函数

如果接收方想获取数据包发送端的标识符，应用程序可以调用 recvfrom 函数来获取数据包发送方的源地址。

```
#include < sys/socket.h >
//函数返回读取消息长度，如果没有消息了则函数返回 0，如果数据发送已结束并且发送方顺序关闭站点函数也返回 0 否则返回错
//误代码
ssize_t recvfrom ( int sd , void * buf, size_t len, int flags, struct sockaddr * addr, socklen_t
* addrlen );
```

recvfrom 函数的输入参数说明如下。

- sd：接收数据的套接字描述符。
- buf：存放接收数据的缓冲区。

- len：接收数据缓冲区的长度。
- flags：标志位，指明接收函数如何处理接收数据，其含义如表 11-8 所示。
- addr：发送数据方源套接字地址。
- addrlen：发送数据方源套接字地址长度。

如果 recvfrom 函数的参数 addr 为非空，其中会存放数据发送端的套接字地址。当调用 recvfrom 函数时，需要设置 addrlen 参数指向一个整数，该整数用于存放发送方套接字地址的长度，即 addr 指针指向的缓冲区长度。当函数返回时，addrlen 整数中将包含地址的实际长度。

因为该函数允许数据接收方获取发送数据方的地址，recvfrom 函数通常用于无连接套接字。否则 recvfrom 函数的其他功能与 recv 函数相同。

为了接收多个数据缓冲区，或接收辅助数据，可以调用 recvmsg 函数来实现网络数据接收。

```
#include < sys/socket.h >
//函数返回接收的数据长度，如果已无有效数据，则返回 0，或发送方数据发送完，关闭套接字，函数也返回 0。否则返回错误代码
ssize_t recvmsg ( int sd, struct msghdr *msg, int flags );
```

在 sendmsg 函数中，struct msghdr 数据结构中存放的是要发送的数据缓冲区数组。在 recvmsg 函数中，struct msghdr 数据结构中存放的是接收到的数据缓冲区数组。我们可以通过设置标志 flags 来改变 recvmsg 函数默认的接收方式。在 recvmsg 函数返回时，struct msghdr 数据结构中的 msg_flags 数据域中包含的是接收数据的特点。在 recvmsg 函数中，将忽略 msg_flags 标志。表 11-8 总结了 recvmsg 函数中接收的标志的含义。

表 11-8　recvmsg 函数中接收标志 flags 的有效值

flags	描　　述
MSG_CTRUNC	控制数据被截短
MSG_DONTWAIT	recvmsg 函数为非阻塞模式
MSG_EOR	接收到的记录已结束
MSG_OOB	收到 Out-of-band 数据
MSG_TRUNC	常规数据被截短

11.4.5　编程示例

以下给出了创建套接字、建立套接字连接、从套接字上读取程序的编程示例，来帮助理解套接字 API 的使用。

1．面向连接的客户端

在客户端创建一个面向连接的套接字，它与服务器建立连接，从服务器读取数据，将读到的数据打印到标准输出终端。该例中使用的是 SOCK_STREAM 套接字，该套接字不能保证一次 recv 函数调用就能读到所有的数据，所以需要反复调用 recv 函数来读取套接字上的数据，直到 recv 函数返回 0。

getaddrinfo 函数可以返回多个候选地址，如果服务器支持多个网络接口或多个网络协议，在读数据时需要依次尝试每个候选套接字，直到找到一个可以连接上的服务为止。

```
int main(int argc, char *argv[])
```

```
{
   struct addrinfo *ailist, *afp;
   struct addrinfo addrfilt;
   int          sockfd, err;

   if (argc != 2)
      exit(-1);

   addrfilt.ai_flags=0;
   addrfilt.ai_family=0;
   addrfilt.ai_socktype=SOCK_STREAM;
   addrfilt.ai_protocol=0;
   addrfilt.ai_addrlen=0;
   addrfilt.ai_canonname=NULL;
   addrfilt.ai_addr=NULL;
addrfilt.ai_next=NULL;
  //获取主机地址信息
   if ((err = getaddrinfo(argv[1], "ruptime", & addrfilt, &ailist))!=0)
      err_quit("getaddrinfo error: %s", gai_strerror(err));
  //从获取的服务器地址链表上的每个地址尝试创建套接字，建立连接
   for (afp=ailist; afp!= NULL; afp=afp->ai_next) {
      if ((sockfd=socket(afp->ai_family, SOCK_STREAM, 0)) < 0)
         err=errno;
      if (connect_repeat(sockfd, afp->ai_addr, afp->ai_addrlen) < 0) {
         err=errno;
      } else {

  //如果某个套接字上建立连接成功，则从套接字上读取数据
         print_uptime(sockfd);
         exit(0);
      }
   }
   fprintf(stderr, "can't connect to %s: %s\n", argv[1],
     strerror(err));
   exit(1);
}
```


2．面向连接的服务器

以下是面向连接的服务器示例。为了发现地址，服务器需要获取其运行的主机名，由主机名获取服务器运行地址。应用程序调用 gethostname 函数来获取主机名与发现远端 uptime 服务器地址，gethostname 函数可能返回多个地址，在多个套接字站点中我们选择第一个可以建立连接的套接字。

```
#include <netdb.h>
#include <errno.h>
#include <syslog.h>
#include <sys/socket.h>

#define BUFLEN  128
#define QLEN 10
#define HOST_NAME_MAX 256
extern int listen_server(int, struct sockaddr *, socklen_t, int);
//提供服务的进程在打开的套接字上接收进入的连接
void serve(int sockfd)
{
   int    clfd;
   FILE   *fp;
```

```
    char   buf[BUFLEN];
for (;;) {
//接收来自客户端的连接请求
      clfd = accept(sockfd, NULL, NULL);
      if (clfd < 0) {
         syslog(LOG_ERR, "ruptimed: accept error: %s",
          strerror(errno));
         exit(1);
      }
//如果接收到请求，运行用户进程 uptimer，将读入数据放进 buf，发送给客户端
      if ((fp = popen("/usr/bin/uptime", "r")) == NULL) {
         sprintf(buf, "error: %s\n", strerror(errno));
         send(clfd, buf, strlen(buf), 0);
      } else {
         while (fgets(buf, BUFLEN, fp) != NULL)
            send(clfd, buf, strlen(buf), 0);
         pclose(fp);
      }
      close(clfd);
   }
}
int  main(int argc, char *argv[])
{
   struct addrinfo *ailist, *afp;
   struct addrinfo addrfilt;
   int          sockfd, err, n;
   char         *host;

   if (argc != 1)
exit( -1 );
   n = HOST_NAME_MAX;
   host = malloc(n);
   if (host == NULL)
      err_sys("分配内存出错，程序退出");
   if (gethostname(host, n) < 0)
      err_sys("获取主机名信息出错，程序退出");
   daemonize("ruptimed");
   addrfilt.ai_flags = AI_CANONNAME;
   addrfilt.ai_family = 0;
   addrfilt.ai_socktype = SOCK_STREAM;
   addrfilt.ai_protocol = 0;
   addrfilt.ai_addrlen = 0;
   addrfilt.ai_canonname = NULL;
   addrfilt.ai_addr = NULL;
   addrfilt.ai_next = NULL;
//获取服务器主机地址信息，从 getaddrinfo 函数返回的地址列表中，依次打开套节字，侦听套节字，在第一个成功打开的字
//上建立连接
   if ((err = getaddrinfo(host, "uptime", & addrfilt, &ailist)) != 0) {
      syslog(LOG_ERR, "uptimed: getaddrinfo error: %s",
       gai_strerror(err));
      exit(1);
   }
   for (afp = ailist; afp != NULL; afp = afp ->ai_next) {
      if ((sockfd = listen_server(SOCK_STREAM, afp ->ai_addr,
       afp ->ai_addrlen, QLEN)) >= 0) {             //侦听套接字
         serve(sockfd);                             //接收客户端套接字连接请求
         exit(0);
      }
   }
```

```
    exit(1);
}
```

前面已经介绍了，使用文件描述符来访问套接字应用很有意义，它允许应用程序在对网络环境不了解的情况下也可以使用网络功能。在示例中给出了这种情形的代码实现。在前面的例子中应用程序调用 popen 运行 uptime 命令行，读入数据，打印到缓冲区后发送给客户端，在该示例中服务器将 uptime 命令的标准输出和标准错误连接到客户端的套接字上。

前例中我们使用 popen 来运行 uptime 命令，从连接到命令的标准输出管道上读取命令的输出，在下例中我们调用 fork 函数创建一个子进程，然后调用 dup2 函数将子进程的标准输入 STDIN_FILENO 的数据复制到打开的/dev/null 中，将标准输出 STDOUT_FILENO 和标准错误 STDERR_FILENO 的数据复制到打开的套接字。当执行 uptime 命令时，uptime 命令将其输出结果写到它的标准输出中，该标准输出连在套接字上，uptime 的输出数据被送回到 uptime 客户端。

父进程可以安全地关闭连接到客户端的文件描述符，在子进程还处于打开状态时，父进程要等待子进程完成处理后才继续。只有子进程不占用太长的时间运行 uptime 命令，子进程才不会成为 zombie 的进程，这样父进程可以等待子进程退出后，再接收下一个连接请求。这个工作模式不适合于子进程运行太长时间的情况。

```
void  serve(int sockfd)
{   int    clfd, status;
   pid_t  pid;
for (;;) {
//接收来自客户端的连接请求
      clfd = accept(sockfd, NULL, NULL);
      if (clfd < 0) {
         syslog(LOG_ERR, "uptimed: accept error: %s",
           strerror(errno));
         exit(1);
      }
//如果接收到客户端的连接请求，创建子进程套接字来处理连接，原进程继续侦听原套节字
      if ((pid = fork()) < 0) {
         syslog(LOG_ERR, "uptimed: fork error: %s",
           strerror(errno));
         exit(1);
      } else if (pid == 0) {
//子进程：复制套接字描述符，子进程运行 uptime 命令，将套接字的标准输出与标准错误终端连接
         if (dup2(clfd, STDOUT_FILENO) != STDOUT_FILENO ||
           dup2(clfd, STDERR_FILENO) != STDERR_FILENO) {
            syslog(LOG_ERR, "uptimed: unexpected error");
            exit(1);
         }
         close(clfd);
         execl("/usr/bin/uptime", "uptime", (char *)0);
         syslog(LOG_ERR, "run uptimed: unexpected return from exec: %s",
           strerror(errno));
//父进程，关闭连接 ，待子进程结束
      } else {
         close(clfd);
         waitpid(pid, &status, 0);
      }
   }
}
//示例主程序
```

```
int  main (int argc, char *argv[])
{
   struct addrinfo *ailist, *aip;
   struct addrinfo addrfilt;
   int          sockfd, err, n;
   char         *host;

   if (argc != 1)
      err_quit("usage: ruptimed");
   n = HOST_NAME_MAX;
   host = malloc(n);
   if (host == NULL)
      err_sys("malloc error");
   if (gethostname(host, n) < 0)
      err_sys("gethostname error");
   daemonize("ruptimed");
   addrfilt.ai_flags = AI_CANONNAME;
   addrfilt.ai_family = 0;
   addrfilt.ai_socktype = SOCK_STREAM;
   addrfilt.ai_protocol = 0;
   addrfilt.ai_addrlen = 0;
  addrfilt.ai_canonname = NULL;
   addrfilt.ai_addr = NULL;
   addrfilt.ai_next = NULL;
   if ((err = getaddrinfo(host, "runuptime", & addrfilt, &ailist)) != 0) {
      syslog(LOG_ERR, "run uptimed: getaddrinfo error: %s",
        gai_strerror(err));
      exit(1);
   }
   for (aip = ailist; aip != NULL; aip = aip->ai_next) {
      if ((sockfd = listen_server(SOCK_STREAM, aip->ai_addr,
       aip->ai_addrlen, QLEN)) >= 0) {
         serve(sockfd);
         exit(0);
      }
   }
   exit(1);
}
```

前面的例子中给出的是面向连接的套接字，在实际应用中如何选择适当的套接字类型。什么时候应使用面向连接的套接字，什么时候应使用无连接的套接字，取决于传送的数据量和传送过程中对错误的容忍程度。

使用无连接的套接字，数据包到达的顺序可以与发送顺序不一样，所以如果要传送的数据不能一次放在一个数据包中全部发送出，则接收程序端收到的数据包顺序可能不正确。不同的协议能传送的最大数据包不同，如果是无连接套接字，则数据包丢失的可能性更大。如果应用程序不允许数据包丢失，就应采用面向连接的套接字。

如果应用程序允许数据包丢失，这时有两个选择，如果希望与站点之间的通信为可靠通信，接收应用程序必须记录数据包数，在检查到数据包丢失的情况下要求重传。另外应用程序必须能够识别复制的数据包并扔掉，因为发送的数据包可能延迟，这时接收程序以为数据包已丢失，要求重传，而随后原数据包又达到了，此时需要识别这些与已接收到的数据包相同的复制品。

面向连接套接字的缺点是，系统需要将很多开销花在建立连接与管理上，每个连接套

接字都会比无连接套接字占用更多的系统资源。

3. 无连接套接字使用示例

基于数据报（datagram）客户端的 main 函数与面向连接的客户端类似，只是安装了一个信号处理程序来处理 SIGALM 信号。在示例中使用 alarm 来避免在调用 recvfrom 函数时被阻塞。

在面向连接的协议中，交换数据之前需要与服务器端建立连接。服务器从接到的连接请求获取信息，使服务器知道客户端需要什么样的服务。如果使用数据报（datagram）协议，则需要发送信息通知服务器，客户端需要什么样的服务。例如，只向服务器发送一个字节的信息，服务器接收到该信息后，可以从数据包中获取客户端的地址，使用这个地址返回它的响应。如果服务器提供了多个服务，则可以使用请求消息数据包指明客户端希望获取什么样的服务。

如果服务器没有运行，客户端进程在调用 recvfrom 函数时会被阻塞，在面向连接的例子中，客户端调用 connect 函数时，如果服务器进程没有运行,connect 函数的调用会返回失败。为了避免客户进程被标识为阻塞，我们在调用 recvfrom 函数前设置一个 alarm 时钟。

```
#include <netdb.h>
#include <errno.h>
#include <sys/socket.h>

#define BUFLEN     128
#define TIMEOUT    30

void sigalrm(int signo)
{
}
void print_uptime(int sockfd, struct addrinfo *aip)
{
   int    n;
   char   buf[BUFLEN];

buf[0] = 0;
//客户端向服务器端发送自身的地址与地址长度信息
   if (sendto(sockfd, buf, 1, 0, aip->ai_addr, aip->ai_addrlen) < 0)
      err_sys("传送数据出错，程序退出");
   alarm(TIMEOUT);
//接收服务器端返回的数据
   if ((n = recvfrom(sockfd, buf, BUFLEN, 0, NULL, NULL)) < 0) {
      if (errno != EINTR)
         alarm(0);
      err_sys("接收数据错");
   }
alarm(0);
//将服务器接收到的数据发送到标准输出终端打印
   write(STDOUT_FILENO, buf, n);
}
int main(int argc, char *argv[])
{
   struct addrinfo    *ailist, *aip;
   struct addrinfo     addrfilt;
   int              sockfd, err;
   struct sigaction    sa;
```

```
    if (argc != 2)
       err_quit("usage: ruptime hostname");
//安装信号处理程序
    sa.sa_handler = sigalrm;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);
    if (sigaction(SIGALRM, &sa, NULL) < 0)
       err_sys("sigaction error");
    addrfilt.ai_flags = 0;
    addrfilt.ai_family = 0;
    addrfilt.ai_socktype = SOCK_DGRAM;
    addrfilt.ai_protocol = 0;
    addrfilt.ai_addrlen = 0;
    addrfilt.ai_canonname = NULL;
    addrfilt.ai_addr = NULL;
    addrfilt.ai_next = NULL;
    if ((err = getaddrinfo(argv[1], "ruptime", & addrfilt, &ailist)) != 0)
       exit (-1);
    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
       if ((sockfd = socket(aip->ai_family, SOCK_DGRAM, 0)) < 0) {
          err = errno;
       } else {
          print_uptime(sockfd, aip);
          exit(0);
       }
     }
     fprintf(stderr, "不能建立连接 %s: %s\n", argv[1], strerror(err));
     exit(1);
}
```

4．无连接的服务器

本例是 uptime 无连接的服务器版本。服务器阻塞在一个 recvfrom 服务请求中。当请求到达时，服务器保存请求方地址，使用 popen 运行 uptime 命令。使用 sendto 函数将数据返回客户端，将返回数据包的地址设置为请求端的地址。

```
#include <netdb.h>
#include <errno.h>
#include <syslog.h>
#include <sys/socket.h>

#define BUFLEN     128
#define MAXADDRLEN  256

#define HOST_NAME_MAX 256

extern int listen_server(int, struct sockaddr *, socklen_t, int);

void serve(int sockfd)
{
   int       n;
   socklen_t  alen;
   FILE      *fp;
   char      buf[BUFLEN];
   char      abuf[MAXADDRLEN];
   for (;;) {
      alen = MAXADDRLEN;
      if ((n = recvfrom(sockfd, buf, BUFLEN, 0,
```

```
          (struct sockaddr *)abuf, &alen)) < 0) {
            syslog(LOG_ERR, "run uptimed: 从客户端接收数据错 : %s",
              strerror(errno));
            exit(1);
         }
         if ((fp = popen("/usr/bin/uptime", "r")) == NULL) {
            sprintf(buf, "错误: %s\n", strerror(errno));
            sendto(sockfd, buf, strlen(buf), 0,
              (struct sockaddr *)abuf, alen);
         } else {
            if (fgets(buf, BUFLEN, fp) != NULL)
               sendto(sockfd, buf, strlen(buf), 0, (struct sockaddr *)abuf, alen);
            pclose(fp);
         }
      }
   }

   int main(int argc, char *argv[])
   {
      struct addrinfo *ailist, *aip;
      struct addrinfo addrfilt;
      int          sockfd, err, n;
      char         *host;

      if (argc != 1)
         err_quit("usage: ruptimed");
      n = HOST_NAME_MAX;
      host = malloc(n);
      if (host == NULL)
         err_sys("malloc error");
      if (gethostname(host, n) < 0)
         err_sys("gethostname error");
      daemonize("ruptimed");
      addrfilt.ai_flags = AI_CANONNAME;
      addrfilt.ai_family = 0;
      addrfilt.ai_socktype = SOCK_DGRAM;
      addrfilt.ai_protocol = 0;
      addrfilt.ai_addrlen = 0;
      addrfilt.ai_canonname = NULL;
      addrfilt.ai_addr = NULL;
      addrfilt.ai_next = NULL;
      if ((err = getaddrinfo(host, "ruptime", & addrfilt, &ailist)) != 0) {
         syslog(LOG_ERR, "run uptimed: 获取地址信息错: %s",
           gai_strerror(err));
         exit(1);
      }
      for (aip = ailist; aip!= NULL; aip = aip ->ai_next) {
         if ((sockfd = listen_server(SOCK_DGRAM, aip ->ai_addr,
           aip ->ai_addrlen, 0)) >= 0) {
            serve(sockfd);
            exit(0);
         }
      }
      exit(1);
   }
```

11.5　套接字选项

套接字机制提供了两类套接字选项 API，以便我们可以控制套接字的行为。一个 API 用于设置套接字的选项，另一个 API 用于查询套接字的选项设置。我们可以设置和读取以下 3 类套接字选项。

- 常规套接字选项，用于所有套接字类型。
- 在套接字层管理的选项，但取决于套接字层支持的协议。
- 协议特定的选项，只对独立的协议有意义。

11.5.1　设置套接字选项

在 Linux 系统中只定义了套接字层选项（前面列出的前两种选项类型），我们可以调用 setsockopt 函数来设置套接字的选项。

```
#include < sys/socket.h >
//如果选项设置成功，则函数返回0，否则返回错误代码
int setsockopt ( int sd, int level, int option, void *val, socklen_t len );
```

setsockopt 函数的输入参数说明如下。

- level：该参数用于指定选项作用的协议层。如果要设置的选项只是普通套接字层的选项，level 的值设置为 SOL_SOCKET，否则 level 设置为要控制的协议编码。例如，当设置 TCP 选项时，level 的值为 IPPROTO_TCP；当设置 IP 层选项时，level 的值为 IPPROTO_IP。
- option：要设置的套接字选项值。表 11-9 中总结了常规套接字层选项的定义。

表 11-9　套接字选项列表

选 项 符 号	变 量 类 型	描　　述
SO_ACCEPTCONN	int	套接字是否允许侦听
SO_BROADCAST	int	如果*val 为非零，广播数据报
SO_DEBUG	int	如果*val 为非零，允许在网络驱动程序中调试
SO_DONTROUTE	int	如果*val 为非零，绕开常规路由，不对数据包路由
SO_ERROR	int	返回套接字错误信息后，清掉错误信息
SO_KEEPALIVE	int	如果*val 为非零，周期性地允许发送保持套接字活动消息
SO_LINGER	struct linger	该选项由 TCP 协议使用，当该选项设置了时，如有关套接字的 close 或 shutdown 调用，这些调用会被阻塞，直到队列中缓存的消息全部发送完成，或阻塞到指定的秒数超时为止
SO_OOBINLINE	int	当*val 为非零时，out-of-band 数据放入常规数据包
SO_RCVBUF	int	接收缓冲区的字节数
SO_RCVLOWAT	int	在调用接收函数时返回的最小数据字节总数
SO_RCVTIMEO	struct timeval	套接字接收调用的超时时间
SO_REUSEADDR	int	如果*val 为非零，在绑定 bind 中重用套接字地址
SO_SNDBUF	int	发送缓冲区的大小，按字节数计算
SO_SNDLOWAT	int	一次发送调用可以发送的最小数据字节数
SO_SNDTIMEO	struct timeval	套接字发送调用的超时时间
SO_TYPE	int	标识套接字的类型

- Val：指向一个数据结构或一个整数，查看设置的选项是什么，有的选项只是在打开/关闭之间切换。如果 val 指向的是一个整数，且该整数为非零，则允许设置选项。如果整数的值为 0，则禁止设置选项。
- Len：指明 val 对象的大小。

11.5.2 读取套接字选项

应用程序可以调用 getsockopt 函数来获取当前选项中的设置值。

```
#include < sys/socket.h >
//如果函数执行成功则返回0
int getsockopt ( int sd, int level, int option, void * val, socklen_t * lenp );
```

getsockopt 函数的输入参数说明如下。

- sd：读取套接字选项的套接字描述符。
- level：该参数用于指定选项作用的协议层（同 setsockopt 函数）。
- option：要读取的套接字选项值。
- val：存放读取的套接字选项值的缓冲区。
- lenp：指向的是一个整数，该整数描述了缓冲区的大小。套接字选项的值会复制到该缓冲区中。如果选项的实际值大于缓冲区的长度，选项会被截短。如果选项的实际值比缓冲区小或与缓冲区的长度相等，则整数会更新为选项实际长度。

当 TCP 套接字绑定一个地址失败时，在超时退出之前，TCP 协议会阻止绑定同一个地址，等待超时通常需要几分钟，如果使用了 SO_REUSEADDR 套接字选项，则可以绕过这个限制。

为了允许使用 SO_REUSEADDR 选项，我们设置一个整数的值为非零，将整数的地址传给 setsockopt 函数的 val 参数，设置 len 参数指向存放整数的大小。

```
#include <errno.h>
#include <sys/socket.h>

int listen_server(int type, const struct sockaddr *addr, socklen_t alen,  int qlen)
{
   int fd, err;
   int reuse = 1;

   if ((fd = socket(addr->sa_family, type, 0)) < 0)
      return(-1);
   if (setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &reuse,
     sizeof(int)) < 0) {
      err = errno;
      goto errout;
   }
   if (bind(fd, addr, alen) < 0) {
      err = errno;
      goto errout;
   }
   if (type == SOCK_STREAM || type == SOCK_SEQPACKET) {
      if (listen(fd, qlen) < 0) {
         err = errno;
         goto errout;
```

```
        }
      }
      return(fd);
errout:
      close(fd);
      errno = err;
      return(-1);
}
```

11.6　out-of-band 数据

out-of-band 是由某些通信协议支持的可选特性，允许更高优先级的数据先于常规数据传送。out-of-band 数据会先于任何已在队列中等待的数据传送。TCP 支持 out-of-band 数据传送方式，但 UDP 不支持。套接字接口的 out-of-band 数据传送在很大程度上受 TCP 对 out-of-band 数据传送的实现方式的影响。

TCP 将 out-of-band 数据看做“紧急数据”。TCP 只支持单字节的紧急数据，但允许紧急数据的传送不按常规数据的传送机制传送。为了产生紧急数据，我们可以在这 3 个传送函数 send、sendto、sendmsg 中用 MSG_OOB 标志。如果用 MSG_OOB 标志传送了一个字节以上的数据，最后一个字节会当做紧急数据字节。

如果在套接字安装了 SIGURG 信号，在接收方套接字收到了紧急数据后，套接字会发送 SIGURG 信号。事先应用程序可以使用 fcntl 函数的 F_SETOWN 设置拥有套接字的进程，当 SIGURG 信号产生时，拥有套接字的进程将处理该信号。

```
fcntl( sockfd, F_SETOWN, pid);
```

当 fcntl 的第 3 个参数 pid 为一个正数时，它代表一个进程的标识符，如果 pid 为一个负数但不是−1，它代表进程组标识符 GID。

F_GETOWN 命令可以用于提取拥有套接字的进程。与 F_SETOWN 命令一样，一个负数代表进程组标识符，一个正数代表进程 ID，因此，owner = fcntl (sockfd, F_GETOWN, 0) 。

如果 owner 的值为正数，则返回到 owner 变量中的值就等于配置在套接字上接收信号的进程 ID；如果 owner 的值为负数，则其绝对值为接收套接字信号的组进程的 ID。

TCP 支持象征紧急数据（urgent）的标志：指向常规数据流的指针，表明将紧急数据放到哪里。如果设置了 SO_OOBINLINE 套接字选项，则可以选择在常规数据中接收紧急数据。为了标识什么时候有紧急数据标志到达，我们可以使用 sockatmark 函数。

```
#include < sys/socket.h >
//如果有标志，则返回 1；如果没有标志则返回 0
int sockatmark ( int sockfd );
```

如果读入的下一个字节数据设置了紧急标志，则 sockatmark 会返回 1。

out-of-band 数据在套接字的队列中出现了，select 函数将像对待一个异常条件一样返回文件描述符。我们可以选择在常规数据中接收紧急数据，也可以在 recv 函数中用 MSG_OOB 标志指定，函数在所有常规数据之前接收紧急数据，TCP 队列中只能有一个字节的紧急数据。如果在接收当前紧急数据之前，新的紧急数据到达，就扔掉现有的紧急数据。

11.7 非阻塞和异步 I/O 操作

通常，在没有有效数据可接收时，recv 函数会阻塞。类似的，当套接字的输出队列没有更多的空间接收发送数据时，send 函数也会阻塞。我们可以把套接字设置为非阻塞工作模式，来改变套接字的活动方式。在这种情况下，以上函数会返回失败而不是被阻塞，并将错误代码设置为EWOULDBLOCK或EAGAIN。在这种情况下，我们可以使用poll或select函数来确定套接字什么时候可以接收或传送数据。

在 Linux 中对实时的扩展实现了一套常规异步 I/O 操作机制。但套接字有它自己的一套处理异步 I/O 操作的机制，即典型的基于套接字的异步 I/O 机制，称为“基于信号的 I/O”，它与用异步 I/O 操作来扩展实时的机制不同。

对于基于套接字的 I/O 操作，在从套接字中读取数据后或套接字的写队列中有有效空间时，我们可以向进程传送信号 SIGIO。实现套接字异步 I/O 操作有两个步骤：

①建立套接字的拥有进程，这样当信号产生时，信号可以传送到处理进程处。

②告诉套接字，需要它在 I/O 操作不会被阻塞的情况下发送信号。

可以用以下 3 种方式完成第一步操作：

- 在 fcntl 中使用 F_SETOWN 命令设置套接字的拥有进程。
- 在 ioctl 中使用 FIOSETOWN 命令设置套接字的拥有进程。
- 在 ioctl 中使用 SIOCSPGRP 命令设置套接字的拥有进程。

用两种方式可以完成第二步操作：

- 在 fcntl 中使用 F_SETFL 命令，允许 O_ASYNC 异步文件操作标志。
- 在 ioctl 中使用 FIOASYNC 命令，允许异步操作。

11.8 本章总结

套接字编程是实现进程间通信的技术，相互通信的两个进程既可以在同一个计算机上，也可以是通过网络连接的不同主机上的两个进程。通过应用套接字 API 编写网络应用程序，我们可以利用 Linux 内核 TCP/IP 协议栈提供的网络通信服务，实现应用数据在网络上快速、有效的传送。除此之外，套接字编程还可以使我们获取网络、主机的各种管理、统计信息。创建套接字应用程序一般要经过以下步骤：

- 创建套接字。
- 将套接字与地址绑定，设置套接字选项。
- 建立套接字之间的连接。
- 接收、发送数据。
- 关闭、释放套接字。

第 12 章　嵌入式系统网络应用技术

本章主要介绍嵌入式 Linux 开发技术，重点内容包括嵌入式开发硬件、软件环境的建立、Linux 内核的配置过程、网络设备驱动程序与应用程序集成至目标文件的过程、目标文件下载至嵌入式目标板的实现技术。

嵌入式系统是结合了硬件和软件，针对某特定应用领域和特定功能而定制的专用系统。所谓专用系统即按照应用的需求量身定制的系统，且这种定制包含了硬件和软件两方面的定制。嵌入式系统与通用系统相对应，但其针对性强，不会造成资源的浪费，便于成本的控制。

12.1　嵌入式系统的设计要素

嵌入式系统是针对专用系统定制的系统，在设计嵌入式系统时，首先要考虑的因素是成本控制。按需求设计，无论是硬件还是软件设计，以满足应用的要求为前提，这样最终才能达到控制产品成本，量身定做的目的。嵌入式系统的特色在功能的实现细节上是透明的，使产品更具有竞争优势。

嵌入式设计要考虑的第二个方面是功耗要低。嵌入式的最大应用领域常常为便携式产品，这些产品在开发设计时要尽量考虑如何降低功耗。在电池技术的发展没有重大突破之前，对产品而言，如何降低功耗就显得尤其重要。功耗大的主要原因来自于工作频率高，以及电子线路的消耗，嵌入式系统把原来很多板级的构造集中到了一个芯片上，减少了板与板的连接和板级走线，就更有效地降低了功耗。

第三个方面是 CPU 使用率。由于嵌入式系统的设计前提是满足应用要求即可，不要造成资源的浪费，所以当系统设计开发完成后，CPU 的使用率应达到 100%。如果系统资源的应用还有空闲，说明在设计上还存在一定的问题。所以嵌入式的设计应从低配置、有限的资源做起，在有限资源的前提下，完成最大的功能需求。

以上 3 个因素孰轻孰重，根据应用的要求而定。如果开发的是消费类产品，降低成本是第一位的；如果开发的是电信类产品，则首要任务是降低功耗，几乎所有电讯类产品需要全年 24 小时不间断地运行。

12.2　嵌入式系统开发环境的构成

最终的嵌入式系统产品，被称之为目标机（target），其资源非常有限，但开发嵌入式系统需要一套功能强大的开发工具，这套开发工具不可能直接运行在目标机上，所以我们在开发时，将开发工具安装在主机系统上。开发完成后，再将系统下载到目标机上测试运行。在建立嵌入式系统的硬件开发环境时，涉及运行开发工具、完成开发功能的主机和最终运行开发完成的嵌入式系统的目标机，以及它们之间的连接方式。

12.2.1 硬件构成

嵌入式 Linux 系统开发最常用的主机类型为 Linux Workstation，也是我们推荐的首选，因为在开发嵌入式 Linux 系统时，要求开发人员对 Linux 环境非常熟悉，所以没有什么方法比每天都使用 Linux 系统更好。

1．开发工作站主机

最常用的 Linux 工作站就是通用的 PC 机。但这并不表示 Linux 只能运行在 PC 机上，Linux 还可以运行在各种各样的硬件平台上。

在主机上，可以安装、使用各种 Linux 产品，如 Ubuntu、Debian， Mandrake、Fedora、SuSe 或 Yellow Dog 。在本书中，我们以标准的 Linux kernel 来描述开发环境，并不假设使用的是哪一种 Linux 产品系统。在硬件的配置上，使用最快的硬件配置对开发有很大的帮助，但中档配置的机器配以适当的 RAM 就能满足嵌入式系统的开发，Linux 可以非常好地利用硬件资源。而真正需要的是较大的存储空间，包括硬盘空间和 RAM 的容量。在主机上，除了需要安装主机运行的 Linux 系统之外，还需要规划至少将 2GB 到 3GB 的磁盘空间留给嵌入式系统开发环境和项目的工作空间。一个未压缩的 kernel 源码，在编译前，会占大约 100MB 的存储空间，编译需要的空间还会增长得更多，如果同时使用 3~4 种 kernel（针对不同的目标机硬件体系结构），就需要将更多的磁盘容量留给 kernel 作为工作空间。

开发工作站主机也可以使用 UNIX 主机或运行 Windows 操作系统的主机，但这都不是最佳的选择。

2．目标机系统的硬件

典型的嵌入式目标机构成如图 12-1 所示，在硬件上包括以下几个重要部分。

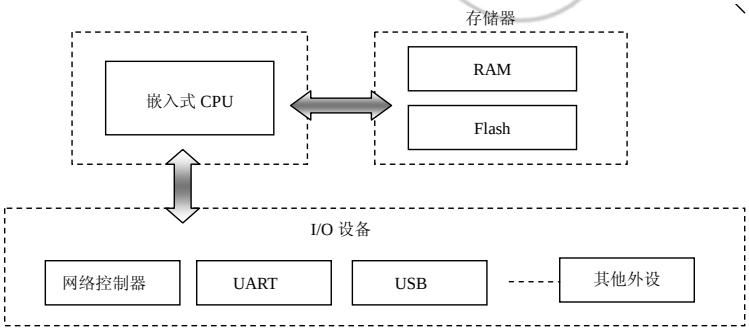


图 12-1　嵌入式控制芯片构成

（1）CPU

目标机中的核心部件就是 CPU，CPU 是运行 OS、应用程序的单元。早期嵌入式系统目标机的 CPU 都是 8 位机或 16 位机。现在的 CPU 都是 32 位机，原因是目前的 32 位机的成本与早期的 8 位机的成本已经一样了。

（2）内存

在嵌入式系统中配置多大的内存是根据不同的应用领域和应用功能而定的，其中一个重要的指标是嵌入式系统工程的 footprint。所谓 footprint 是嵌入式系统在配好 kernel 及应用程序后的所有程序所占存储空间的大小。它是衡量最终应在嵌入式系统中配置多大内存

的重要依据。

通常小规模的嵌入式应用系统，如工业控制应用中内存的配置为几十 KB 至几 MB。中大规模的嵌入式系统，如含有人机交互的嵌入式应用系统，需要 16MB 至上百 MB 的内存。大规模的嵌入式系统，如游艺机、飞行模拟器等需上 GB 的内存配置。

（3）外设

嵌入式系统目标机的特点是集成了大量的外设接口，将应用系统需要的外设接口集成在一个目标板上，使整个系统的集成提高了，这样功率小，出错率低，抗干扰的能力就强了。

12.2.2　典型的硬件开发环境

在嵌入式系统开发过程中，主机和目标机典型的连接方式为：主机和目标机通过串口和 Ethernet 网络连接。

如图 12-2 所示，在开发主机上运行开发嵌入式应用系统所需的交叉平台开发环境，在主机开发完成或需调试的嵌入式软件，通过 RS-232 串口或网络连接下载到目标机上调试、运行。最终集成好的嵌入式系统应包括启动目标机的 bootloader、通过配置裁减了的 Linux 内核、root 文件系统。

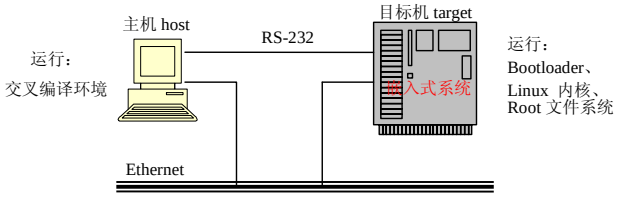


图 12-2　典型的硬件开发连接环境（1）

主机与目标机第二种连接方式如图 12-3 所示，主机与目标机除通过串口、网络连接外，还通过共享硬盘连接。目标机运行的根文件系统可以直接存储在共享硬盘中，调试完成后再下载至目标机集成。

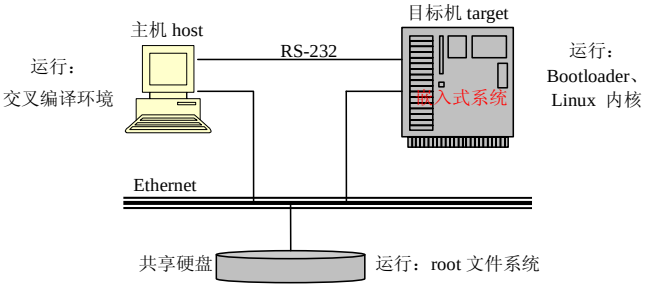


图 12-3　典型的硬件开发连接环境（2）

第三种常用的嵌入式 Linux 硬件连接方式如图 12-4 所示，除以上硬件连接外，还配置一个专门的启动服务器，通过网络连接来启动目标机。

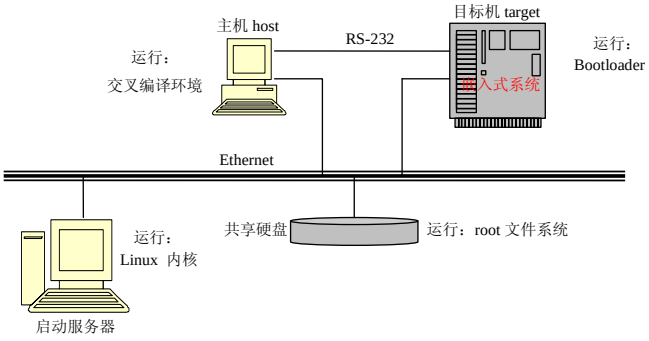


图 12-4　典型的硬件开发连接环境（3）

虽然我们可以通过多种途径和方式建立 Linux 嵌入式的硬件连接环境，但在实际开发过程中应用最多、最直接的方式还是如图 12-2 所示的连接环境。

12.2.3　软件交叉平台开发环境

在嵌入式开发过程中，除了需要建立适当的硬件开发环境外，还需要在开发主机上建立一套软件交叉平台开发环境。为什么称为交叉平台开发环境？是因为我们在主机开发的系统最终是运行在目标机上的，开发主机的 CPU 体系结构与目标机的 CPU 体系结构通常不一样，所以我们不能直接用开发主机上运行的开发工具来构造目标机上可运行的目标代码。我们必须在开发主机上运行一套开发工具软件集，该开发工具软件可以生成目标机 CPU 体系结构的可执行文件（即目标代码），将生成好的执行文件从开发主机下载到目标机执行。这样的开发环境我们称之为交叉平台开发环境。

建立交叉平台开发环境需要的开发工具软件常包括：

- 二进制链接、实用工具，如 ld、gas、ar、OBJCOPY、OBJDUMP 等。
- C 编译器，如 GCC。
- C 库，如 glibc、uclibc。
- 汇编程序语言 as。
- 项目管理构造工具 make。
- 源代码调试器，如 GDB 等。

目前在 Linux 环境下，使用最广泛的交叉编译工具链就是 GNU 的工具链，该工具链可以从 FSF 的 FTP 站点 ftp://ftp.gnu.org/gnu/下载。

除此之外，在开发主机上我们还需要一个适合于嵌入式 CPU 体系结构的 Linux 内核源码包。

12.2.4　嵌入式软件的开发步骤

完成嵌入式应用系统开发通常需要经过以下开发流程。

1．**移植** bootloader

bootloader 是嵌入式系统开机运行的第一个软件，bootloader 的主要任务是完成系统硬件最基本的初始化，将硬件设置为一个基本的可工作状态，为 Linux 内核建立基本的运行

环境，它的功能通常包括：

- 将 CPU 初始化为可工作状态。
- 初始化主存储器。
- 初始化系统中断。
- 初始化系统时钟系统。
- 将 Linux 内核调度到内存的执行地址。

2．配置 Linux 内核

如前所述，嵌入式系统是按需求定制的系统，我们只按照实际应用的需求选择最需要的软件，这不仅包括应用软件的定制开发，也包括对 Linux 内核的配置与裁减。Linux 内核是一个庞大的系统，将一个通用的 Linux 操作系统应用于某一特定的嵌入式系统之前，往往需要对内核的组件进行选择配置，只配置需要的组件。内核的配置常包括如图 12-5 所示的内容。

3．开发设备驱动程序

嵌入式系统的外部设备往往各具特色，新的设备不断涌现，如果应用程序的外部设备需要的驱动程序在现有的 Linux 内核中还不具备，就需要我们自己开发设备驱动程序，Linux 嵌入式系统中外部设备驱动程序主要包括 3 大类：

- 字符设备驱动程序。
- 块设备驱动程序。
- 网络设备驱动程序。

本书中主要介绍了网络设备驱动程序的开发（见第 5 章），以及如何将新的网络设备驱动程序集成到 Linux 内核。

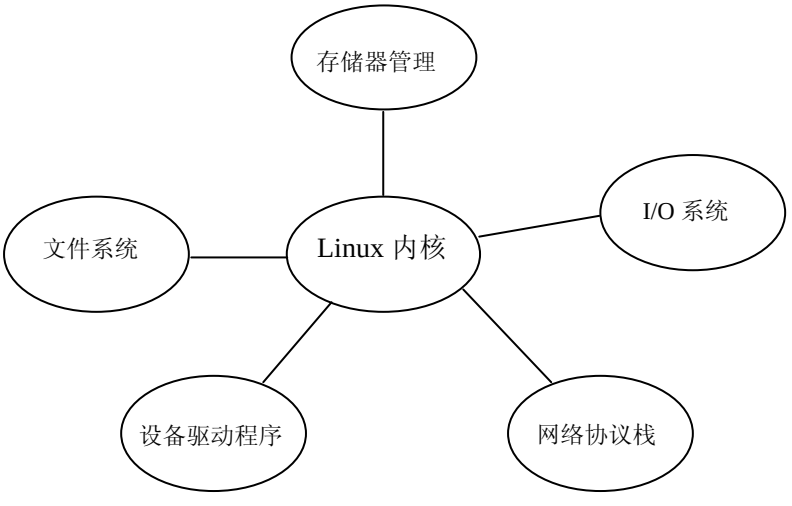


图 12-5　内核配置内容

4．集成嵌入式应用

最后需要完成的就是嵌入式应用程序的开发，并将嵌入式应用构建在根文件系统中，

与 Bootloader、Linux 内核一起下载到目标机上运行。

以下主要针对网络应用过程描述嵌入式系统开发、集成的步骤。

12.3 将网络设备驱动程序加入内核

网络设备驱动程序的开发过程在第 5 章中已详细讲解，假设现在制造好了一个嵌入式硬件目标开发板，目标开发板上所用的网络控制器就是在第 5 章讲解的 CS8900A 以太网控制器，而在使用的 Linux 内核源码中没有 CS8900A 网络控制器的驱动程序，到目前为止也完成了 CS8900A 网络设备驱动程序源代码的开发调试，代码存放在 Linux 内核源代码树的 linux-2.6.29/drivers/net/ 目录下的 cs89x0.c 文件中，现在要解决的问题是，在为嵌入式系统编译 Linux 内核，构造嵌入式系统内核 image 二进制文件时，如何将新的网络设备驱动程序 CS89x0.c 源文件编译、链接到内核执行文件中？其次，对于嵌入式系统使用的 Linux 内核，如何将 CS8900A 配置为系统的网络设备？本节就将描述解决这两个问题步骤。

这里还需要说明的是，对于不同的嵌入式 CPU 体系结构，都有一款相应的经过重新配置、裁减、开发了的 Linux 内核最适合，这里以 uClinux 为例来说明内核的配置、编译过程。uClinux 的含义是“针对微控制领域而设计的 Linux 系统”（Micro-Control-Linux），是嵌入式领域广泛应用的一个 Linux 系统。

12.3.1 配置新网络设备

在开始具体的配置操作步骤之前，需要首先了解 uClinux 源码树型结构主要目录文件分布。uClinux 的源码树分布如图 12-6 所示。

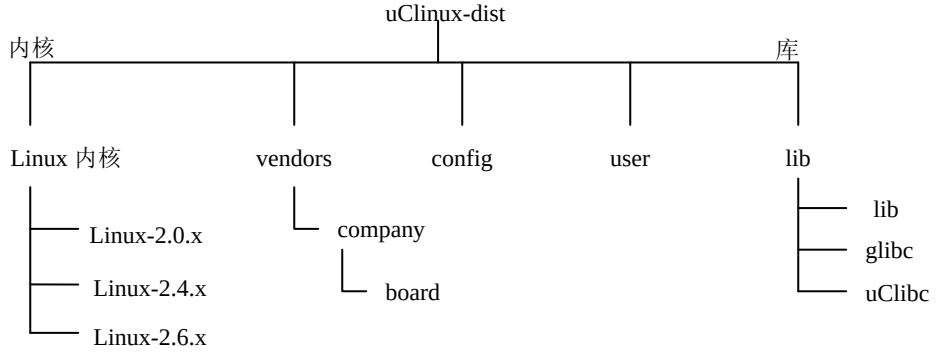


图 12-6 uClinux 目录树

- 内核源码目录：uClinux 软件包中包含了 3 个版本的 Linux 内核源码树，分别是 Linux-2.0.x、Linux-2.4.x、Linux-2.6.x。
- vendors 目录：在 vendors 目录下是各嵌入式目标板硬件制造商公司目录，各公司目录下是 uClinux 支持的各公司嵌入式目标板硬件。

- config 目录：config 目录下是应用程序的配置文件。当我们要在系统中加入新的应用程序时，在该目录中配置。
- user 目录：user 目录下存放的是应用程序的源文件。
- lib 目录：lib 目录为 C 库文件。

当要加入新的设备驱动程序时，由于设备驱动程序运行在内核地址空间中，它是内核的一个部分，所以我们应该工作在 Linux 内核的某个版本目录下。假如我们选择在 Linux-2.6 版本上开发新的嵌入式系统，网络设备驱动程序驻留在 uClinux-dist/linux-2.6.x/drivers/net 目录下。

在 uClinux-dist/linux-2.6.x/drivers/net 目录下包含了 uClinux-2.6 版本内核支持的各种网络设备驱动程序的源文件、头文件，以及一个项目构造文件 Makefile 和一个网络设备驱动程序配置文件 Kconfig。

用任意文本编辑工具打开 uClinux-dist/linux-2.6.x/drivers/net 目录下的网络设备驱动程序配置文件 Kconfig，可以看到其中包含了对各种网络设备驱动程序的配置命令，如[luser@localhost net] $ vi Kconfig，来编辑 Kconfig 配置文件。通常按照设备驱动程序名字母顺序在文件的适当位置加入以下语句：

```
config  CS89x0
     tristate  "CS80x0 Support"
      depends on NET_ETHERNET && (ISA || EISA || MACH_IXDP2351…
     --- help ---
       Support for CS89x0 chipset based ETHERNET. If you have a network
       (Ethernet) card or embedded chipset of this type, say Y .
       To compile this driver as module, choose M here. The module will be called
       cs89x0.
```

在 Kconfig 文件中使用“tristate”语句，该驱动程序可以选择 Y、M、N；“depends on”语句说明如果装载该驱动程序则要依赖其他模块，在装载设备驱动程序之前，依赖的模块应首先装载到内核中。

“--- help ---”语句是对驱动程序的帮助说明，用户在配置内核的时候可以查看帮助说明来了解该配置项的含义。

保存 Kconfig 文件并退出后，在用户配置内核时，可以在配置网络相关的内容时看到新加入到内核中的网络设备驱动程序，当用户选择了 CS8900A 作为目标板的驱动程序时，配置就保存在 CONFIG_CS89x0 变量中。

12.3.2　编译新驱动程序

除了修改网络设备驱动程序目录下的配置文件外，为了在生成内核的二进制文件时将驱动程序的模块链接到内核中，我们还需要修改 uClinux-dist/linux-2.6.x/drivers/net 目录下的项目管理文件 Makefile，将 CS8900A 的目标文件加入其中，用任意的文件编辑工具打开 uClinux-dist/linux-2.6.x/drivers/net 目录下的 Makefile 文件，如[luser@localhost net] $ vi Makefile 来编辑项目管理文件。将 CS8900A 驱动程序的目标文件按以下语法加入：

```
obj -$(CONFIG_CS89x0) += cs89x0.o
```

这样在配置 Linux 内核时，如果用户选择了 CS8900A 网络设备驱动程序，就会设置

CONFIG_CS89x0 变量。在编译 Linux 内核时，CS8900A 驱动程序的二进制模块就会编译到内核中，成为内核的一部分。

12.4 内核配置

现在如果我们完成了一个新的目标板硬件开发，选择 uClinux 作为目标板的操作系统，以下就以 coldfire uc5272 系列的嵌入式目标板为例，来讲解如何让 uClinux 支持新的目标板，如何配置内核、构造根文件系统集成嵌入式应用。

要配置内核，我们首先需要获取 uClinux 源码包，这可以从 http://www.uclinux.org/pub/uClinux/站点下载，放在工作目录下，用以下命令将 uClinux 源码解压在工作目录下。

如果获取的是 uClinux-dist-20090618.tar.bz2 文件，解压命令为：

$ tar –jxvf uClinux-dist-20090618.tar.bz2↙

如果从站点上获取的是 uClinux-dist-20090618.tar.gz 文件，则解压命令应为：

$ tar –zxvf uClinux-dist-20090618.tar.gz↙

文件解压后会在工作目录下生成一个 uClinux-dist 目录，uClinux 内核的源文件全部驻留在该目录下，进入 uClinux-dist 目录，在此目录下运行对内核的配置如下：

```
$ cd uClinux-dist↙
$ make xconfig↙
```

在 uClinux-dist 目录下运行 make xconfig 命令后，就会进入 uClinux 内核配置的图形界面。内核配置共分为 3 大部分：第一部分是硬件制造商/目标硬件产口与内核版本/库函数选择；第二部分是内核组件选择；第三部分是应用配置选择。以下我们就按照上述过程来完成内核配置操作。

12.4.1 目标硬件及内核、库配置

uClinux 内核配置第一部分的图形界面如图 12-7 所示。

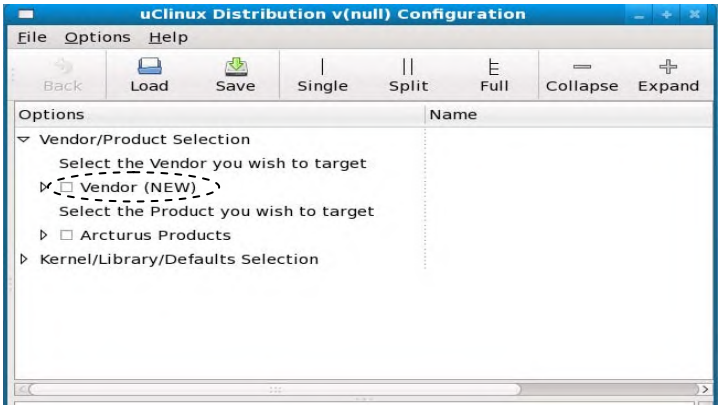


图 12-7 uClinux 硬件及内核版本、库函数配置界面

1．硬件目标板选择

coldfire uc5272 是 Arcturus 公司生产的一款适合于工业控制的嵌入式目标板，在“Vendor/Product Selection”下展开“Vendor”目录树，选择“Arcturus”选项。随后在“Arcturus Products”目录下选择的产品，如图 12-8 所示。

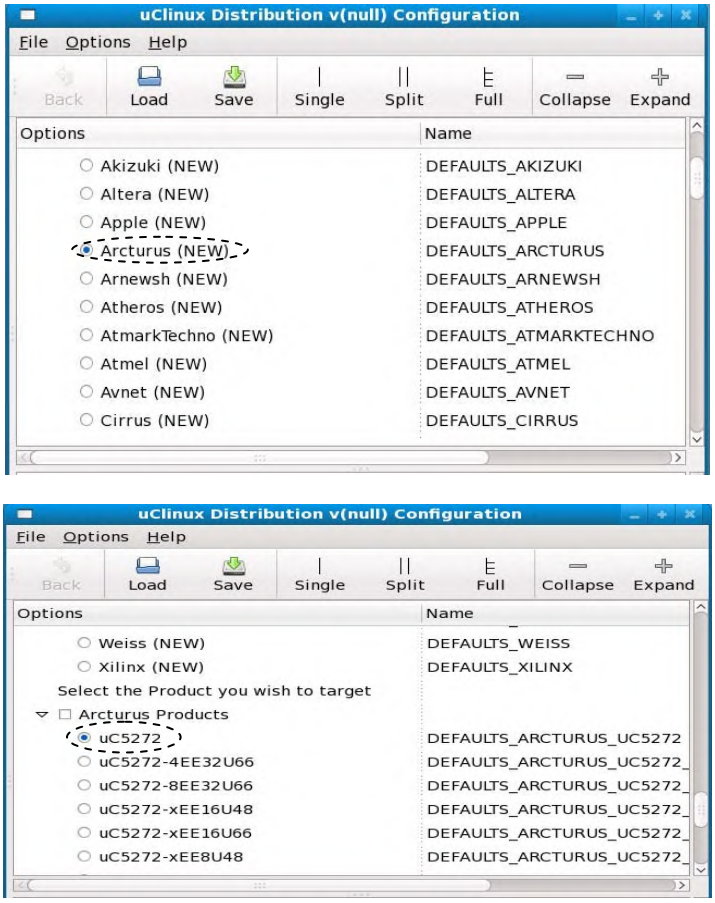


图 12-8　制造商及硬件目标板选择界面

2．内核版本及库函数选择

展开“Kernel/Library/Defaults Selection”目录树，在“Kernel Version”目录下选择 Linux-2.6.x，在“Libc Version”目录下选择“uClibc”库函数，如图 12-9 所示。uClibc 是一个面向嵌入式 Linux 系统的小型 C 标准库。最初 uClibc 就是为了支持 uClinux 而开发的。它比一般用于 Linux 发行版的 C 库 GNU C Library（glibc）要小得多，glibc 目标是要支持最大范围的硬件和内核平台的所有 C 标准，而 uClibc 专注于嵌入式 Linux，很多功能可以根据空间需求进行取舍。

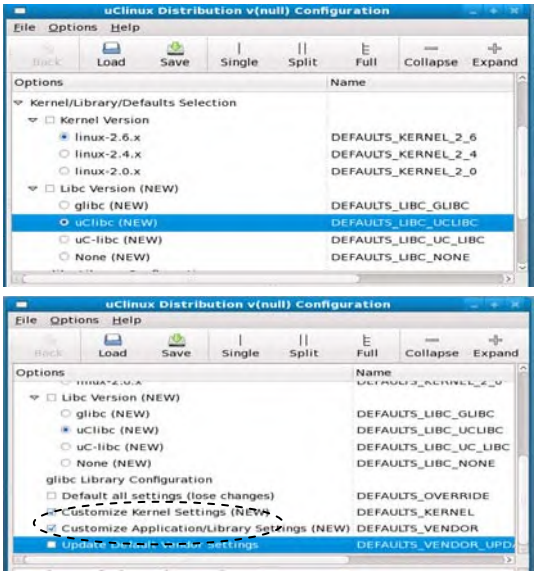


图 12-9　内核版本及库函数配置界面

最后选择“Kernel/Library/Defaults Selection”目录下的“Customize Kernel Settings”和“Customize Application/Library Settings”，复选框让系统记住刚才所做的配置。

在“File”菜单下保存配置后退出。在 uClinux 系统编译一段时间，更新内核配置后，进入内核配置的第二部分。

12.4.2　内核组件配置

在完成第一部分的配置后，系统经过更新自动进入 uClinux 配置的第二部分。如果在第一部分的配置过程中，没有进行任何修改或没有选择“Customize Kernel Settings”和“Customize Application/Library Settings”复选框让系统记住所做的更新，则从“File”退出后，系统不会进入第二部分配置，内核组件配置（Kernel Configuration）图形界面如图 12-10 所示。

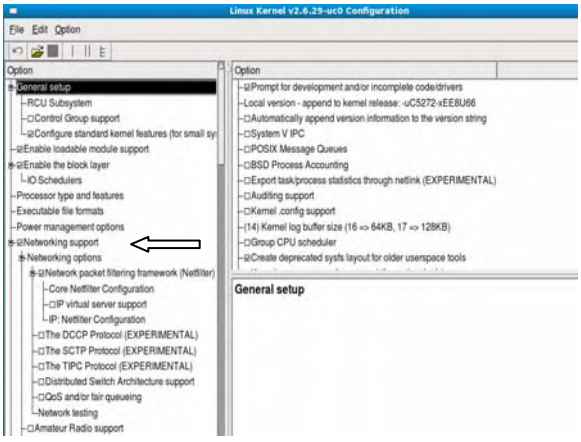


图 12-10　内核组件配置界面

在这里包含了所有内核组件，我们所做的配置过程如图 12-5 所示一样，应有对 CPU 功能特点的配置、存储器管理配置、网络栈功能配置、I/O 设备配置及文件系统配置等。

1．CPU 类型及特性配置

在下图 12-11 的“Processor type and features”目录下，我们应对目标板的 CPU 类型及功能特性进行配置，以选择最佳参数，让嵌入式目标系统运行最充分地利用 CPU。CPU 的类型及特性配置通常包括以下各项。

- 选择 CPU 类型：MCF5272。
- CPU 的时钟频率：66MHz，这是系统的核心频率。
- “Arcturus Networks uC5272 uCdimm board support”：是 Arcturus 的嵌入式网络目标板支持。
- “Use uCbootstrap system calls”：支持 uCbootstrap 系统调用。
- RAM 大小为 8MB，16 位总线宽度。
- 内核 kernel 从 RAM 执行，调度到 RAM 0x00020000 起始地址处。

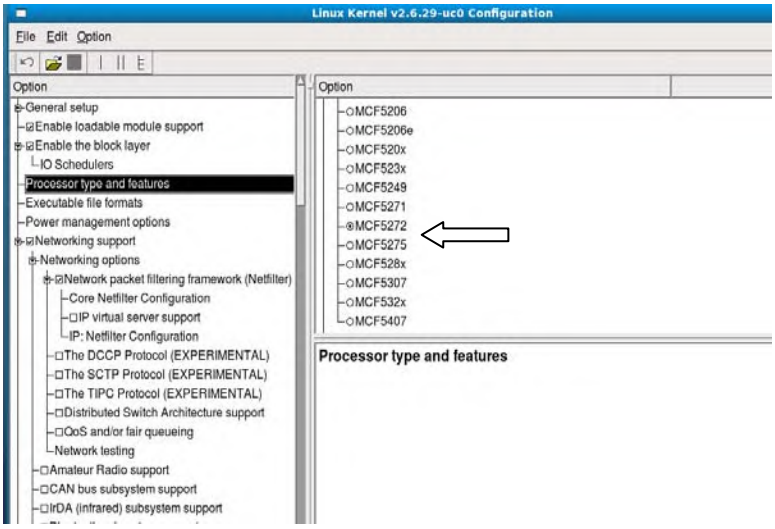


图 12-11　CPU 类型及功能特性配置界面

2．网络功能支持配置

在内核组件配置图形界面 12-10 中的“Networking support”目录树下选择对目标板网络功能的配置。“Networking support”功能支持这一项一定要选择，有的程序即使是单机运行，也需要内核网络功能支持才能正确运行。

选择“Networking support”→“Networking options”→“TCP/IP networking”配置，TCP/IP 协议是在 Internet 与局域网中应用最广泛的网络协议，同样即使系统没有连接到网络上在单机运行，许多程序也要使用 TCP/IP 协议栈提供的功能服务，如图 12-12 所示。

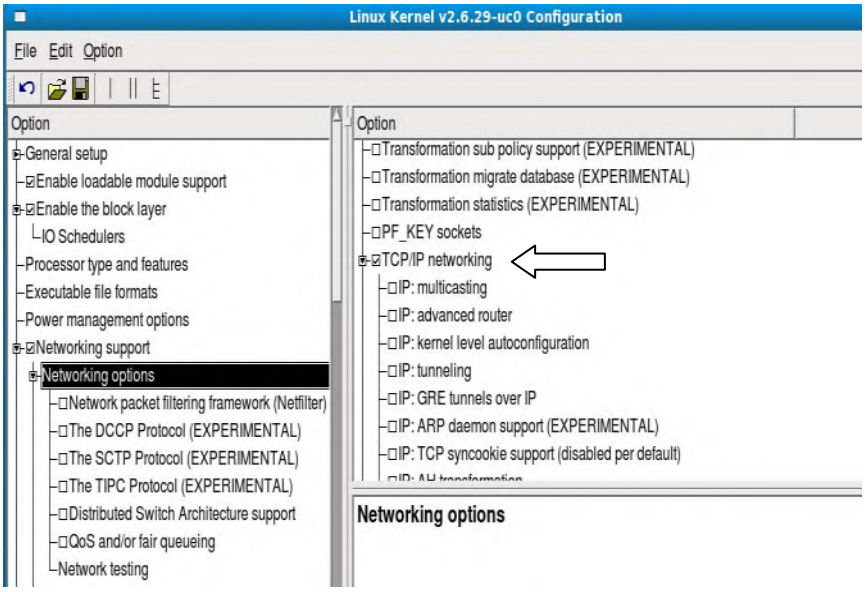


图 12-12　目标板网络功能支持配置界面

3．可执行文件格式

在“Executable file formats”目录下配置内核支持的可执行文件格式“Kernel Support for flat binaries”，这是支持 uClinux 的 FLAT 格式二进制执行文件。“Enable ZFLAG support”，是支持 FLAT 格式的压缩二进制文件。

4．存储器管理配置

在“Device Drivers”目录树下的“Block devices”选择块设备支持，在该目录中选择“RAM block device support”。在选择了 RAM 块设备支持后，可以将 RAM 存储器的一部分作为一个块设备使用，这样我们可以把文件系统安装在 RAM 块设备上，可以将 RAM 块设备当做一般的块设备（如硬盘）进行读写操作，如图 12-13 所示。

5．网络设备驱动程序配置

对网络设备驱动程序的配置也在“Device Drivers”目录树下选择“Network device support”，如果硬件目标板上有网络设备控制器，而且应用程序也需要使用该网络设备控制器作为目标板连接上网的接口，就需要选择该配置。

如前所述，我们在目标板上配置了 CS8900A 网络控制器，则在此选择“Ethernet （10 or 100Mbit）”。如果在前面编辑 uClinux-dist/linux-2.6.x/drivers/net/Xconfig 配置文件中，正确加入了 CS8900A 网络设备驱动程序的配置项，这里在“ Ethernet(10 or 100Mbit)”选项下就可以看到 CS8900A 的支持选项“CS89x0 support”，如图 12-13 所示。

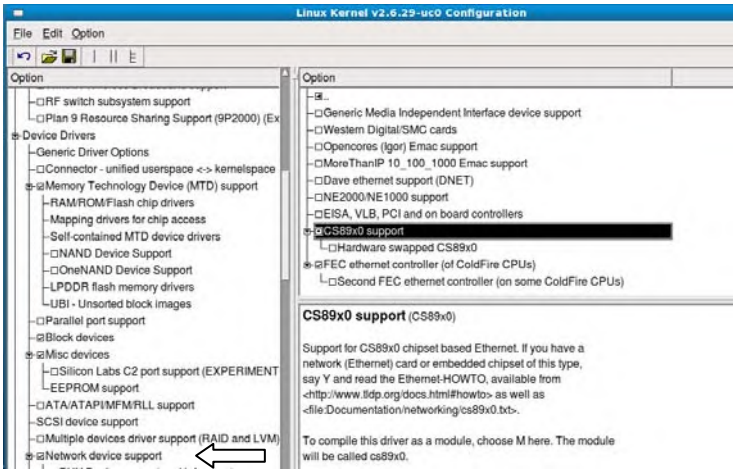


图 12-13　CS8900A 网络设备驱动程序配置界面

6．字符设备配置

前面已经介绍了，典型的开发主机与目标机的连接方式是：主机与目标机通过串口、以太网相连（见图 12-2）。在最初启动目标板时，嵌入式系统的 bootloader、内核文件、根文件系统往往通过串口下载至目标板。所以在这里就需要配置对目标板串口的支持，串口的配置路径为：

选择“Device Drivers”→“Character devices”→“Serial drivers”→“Coldfire serial support”。这样在编译内核时，支持 Freescale Coldfire 串口设备的驱动程序就会编译到内核中,如图 12-14 所示。

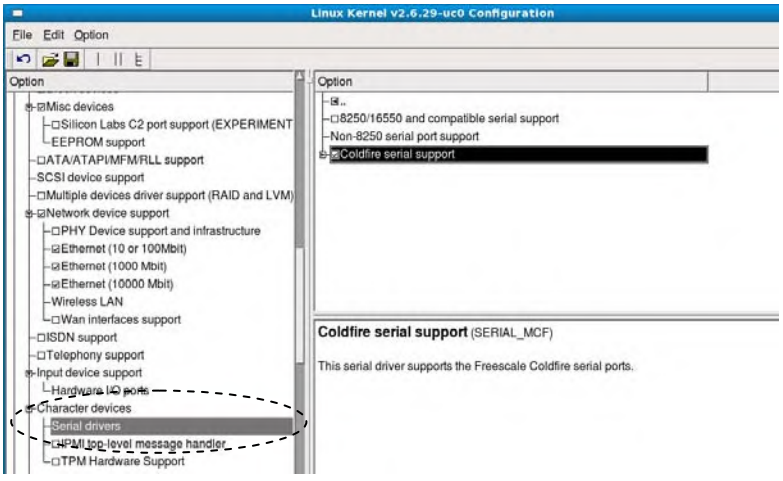


图 12-14　串口驱动程序配置界面

7．文件系统配置

在文件系统支持配置中，通常需要完成以下 3 项配置：

（1）目标板使用的 root 文件系统

在嵌入式系统中我们通常选择压缩的 ROM 文件系统支持（cramfs）或 ROM 文件系统

支持，压缩 ROM 文件系统是设计用于嵌入式系统，简单的、小型 ROM 文件系统，文件系统驻留在 ROM 上，为只读文件系统。

（2）虚拟文件系统/proc 支持

选择“File systems”→“Pseudo filesystems”→“/proc file system support”和“sysfs file system support”。

配置了以上虚拟文件系统支持后，内核中的许多组件都可以在/proc 目录下输出内核的配置参数，如图 12-15 所示。

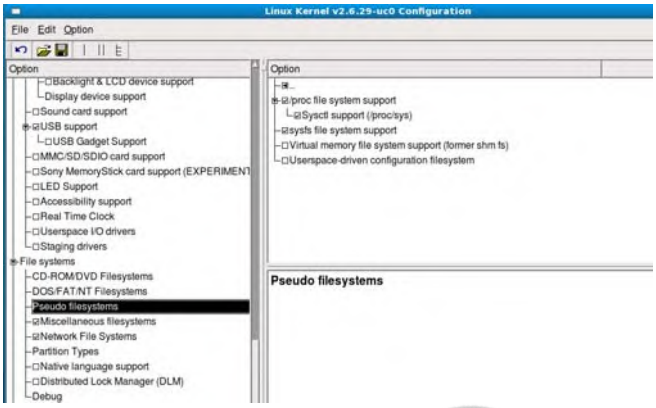


图 12-15　虚拟文件系统配置界面

（3）网络文件系统

在网络文件系统支持中，将目标板配置为 NFS 的客户端，这样在将内核文件、应用文件下载至目标板时，可以使用网络连接下载，速度更快。网络文件系统支持配置路径为：“File systems”→“Network File Systems”→“NFS client support”→“NFS client support for NFS version3”，如图 12-16 所示。

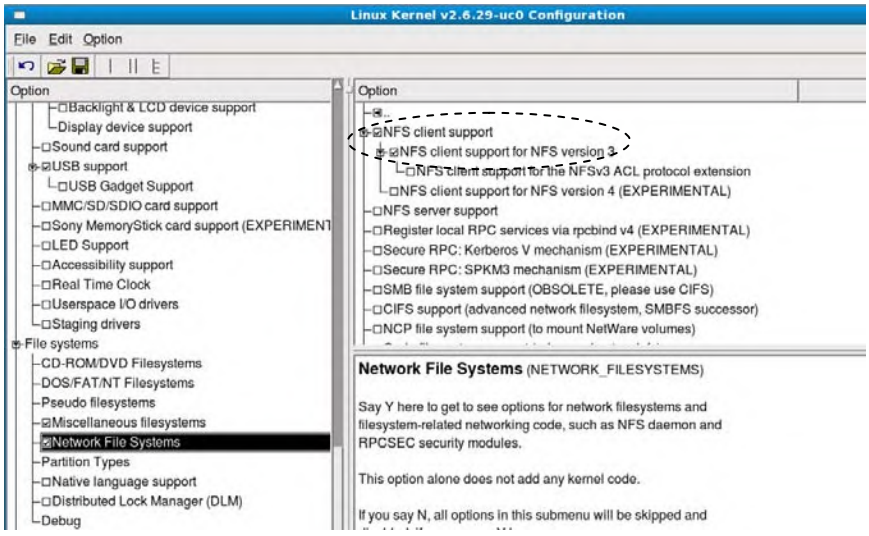


图 12-16　网络文件系统配置界面

12.4.3　应用配置

内核组件配置完成，在“File”菜单中保存配置并退出后，系统自动进入 uClinux 的应用配置图形界面，如图 12-17 所示。

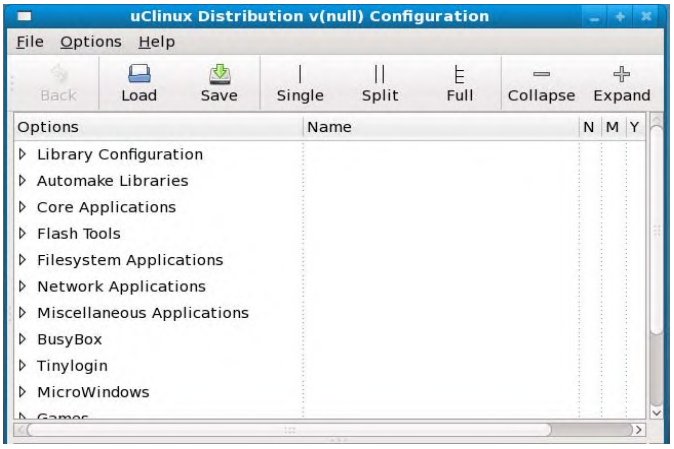


图 12-17　系统应用配置界面（1）

在应用配置中主要选择在嵌入式 root 文件系统上运行的 shell 程序，在/bin 目录及/sbin 目录下运行的 shell 命令，如 cat、ls、mount、ping 等应用。需要选择配置哪些应用根据系统需求而定，如图 12-18 所示。

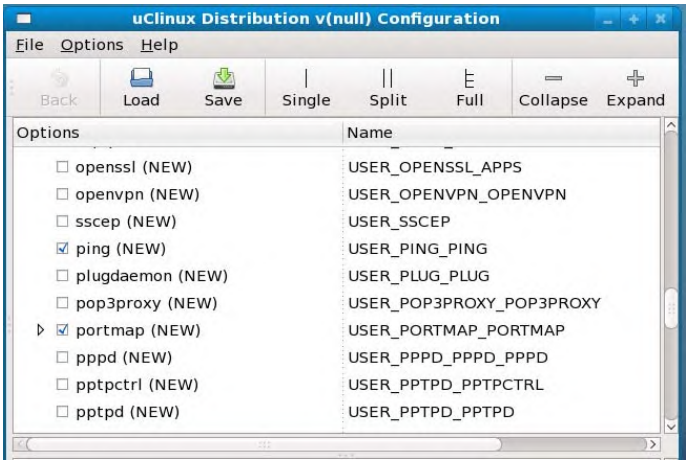


图 12-18　系统应用配置（2）

完成内核配置后，在 uClinux-dist/目录下运行 make 命令：

```
$ make ↙
```

编译配置完成的内核文件及应用。make 命令编译、链接完成后会在 uClinux-dist/目录下生成一个 images 目录，将生成的内核二进制文件 image.bin 复制到该目录下，即最后生成的目标文件为：uClinux-dist/images/image.bin。

12.5 集成应用程序并下载至目标板

在前面的过程中我们向 Linux 内核加入了新的网络设备驱动程序，在配置 Linux 内核时选择的是内核中已有组件、网络设备功能支持、新的网络设备驱动程序和 uClinux 已有的应用。接下来将讲解嵌入式系统新开发的应用程序应如何集成在 uClinux 的可执行文件中。

12.5.1 集成应用程序

在 uClinux 树型结构中（见图 12-6），如果要集成应用程序则需要工作在 uClinux-dist/config 与 uClinux-dist/user 目录下。在 uClinux-dist/config 目录下需要编辑应用程序的配置文件，将新的应用程序加入到配置文件中。应用程序源代码驻留在 uClinux-dist/user 目录下。

1．编写应用程序

在 uClinux-dist/user 目录下加入自己的应用程序，我们以输出“Hello，World”的 C 程序为例来讲解在 uClinux 中加入应用程序的过程。在 uClinux-dist/user 目录下创建一个新目录：MyApp，将新应用程序的源代码、Makefile 文件都放入该目录下。

```
mkdir MyApp↙
cd  MyApp↙
```

用任意文件编辑器工具如 vi，编写 hello.c 与 Makefile 源文件。

```
// hello.c 源文件
#include <stdio.h>

int main ()
{
  printf( "Hello , World !! \n");
  return 0;
}
//Makefile 文件
hello.o: hello.c
  gcc –c hello.c
```

2．加入新配置条目

在 uClinux-dist/user 目录下的 Kconfig 文件中加入应用程序的配置条目。

（1）编辑应用程序配置文件

```
$ cd uClinux-dist/user
$ vi Kconfig
```

在 Kconfig 文件的末尾加入以下条目：

```
menu "My Application"
comment " My App "
config USER_MYAPP_HELLO
    bool"hello"
endmenu
```

在应用程序配置文件中加入以上条目后，就会在 uClinux 内核配置的第三部分“应用配置”菜单中，看到“My Application”配置菜单项供配置选择，如选择了这一应用项，新的应用程序在内核编译时就可以集成到可执行文件中。

（2）编辑应用程序 Makefile 文件

在 uClinux-dist/user 目录下编辑管理项目的 Makefile 文件，将应用程序的目标文件加入到应用程序的编译、链接过程中。

```
$ cd uClinux-dist/user
$ vi Makefile
```

在 Makefile 文件末尾加入以下条目：

```
…
dir_$(CONFIG_USER_ZEBRA_ZEBRA_ZEBRA) += zebra
dir_$(CONFIG_USER_MYAPP_HELLO)+= hello
…
```

在 uClinux-dist/user 目录下的项目管理文件 Makefile 中加入了以上条目后，新的应用程序的目标文件就集成到 uClinux 的整个应用程序包中。

12.5.2　将执行文件下载至目标板

编译好的目标文件集成了 uClinux 内核、应用程序，按如图 12-2 所示的主机与目标机连接的方式，我们可以通过串口 RS-232 将目标文件下载至目标板。我们编辑一个脚本文件来完成目标文件自动下载至目标板的过程。

```
$ vi auto_download.sh
```

执行以上脚本文件，编辑好的目标文件就会下载至嵌入式目标板，我们可以通过 Linux 下的超级终端 minicom 与嵌入式目标板来管理与通信，查看调试目标文件的执行状况。

```
#==================================
#!/bin/sh
# set up stuff
//指定下载的目标文件所在目录及设备下载的串口
XFILE=/uClinux-dist/images/image.bin
PORT=/dev/ttyS0
echo "reset the target board please"
sleep 1
//设置串口波特率
# 9600 and set speed to fast
stty 9600 < $PORT
echo "fast" > $PORT
sleep 1
//开始下载目标文件至嵌入式板
# send the rx command
stty 115200 < $PORT
echo "rx" > $PORT
sleep 1

# now download the file
/usr/bin/sx -vv $XFILE >$PORT <$PORT
sleep 1
```

```
echo "slow" > $PORT
echo "Done"
```

12.6 本章总结

嵌入式应用系统的开发与常规桌面的开发不同，嵌入式开发的难点就在于嵌入式应用系统的开发在主机上进行，目标代码需要在嵌入式目标板上运行，如何建立嵌入式应用系统的开发、调试环境，建立主机与目标板的通信方式，方便目标文件下载是嵌入式系统开发首先要解决的问题。本章在以下几个方面对嵌入式应用系统开发技术进行了描述：

- 嵌入式应用系统的开发特点。
- 嵌入式应用系统开发环境的建立。
- 对 Linux 内核的裁减与移植，按实际应用需求定制嵌入式应用目标文件。
- 将新开发设备驱动程序与应用程序集成到内核中的过程。
- 将目标文件下载至目标板的过程。

嵌入式Linux网络体系结构设计与TCP/IP协议栈

本书涵盖了Linux嵌入式系统开发中网络体系结构实现的主要内容。包括：Linux网络包传输的关键数据结构——Socket Buffer；网络设备在内核中的抽象——struct net_device数据结构等。

ISBN：978-7-121-12976-6

书价：69.00