第十二章　网络

Linux 的网络功能是 Linux 最显著的特点之一，它作为一种网络操作系统，具有比 Windows NT 安全稳定、简易方便的优点，在操作系统领域成为一支不可忽视的生力军。本章将以面向对象的思想为核心，分别对网络部分的 4 个主要对象：协议、套接字、套接字缓冲区及网络设备接口进行了具体分析。

在 Linux 的应用方面，基于 Linux 的网络服务器是非常成功的范例，并且广泛用于商业领域，在网络服务器平台中所占比例逐年上升，所以对 Linux 的网络部分的研究具有广阔的市场价值和现实意义。

12.1　概述

Linux 优秀的网络功能和它严密科学的设计思想是分不开的。在分析 Linux 网络内容之前，我们先大体上了解一下网络部分的设计思想及其特点，这对于我们后面的分析很有帮助。

（1）Linux 的网络部分沿用了传统的层次结构。网络数据从用户进程传输到网络设备要经过 4 个层次，如图 12.1 所示。

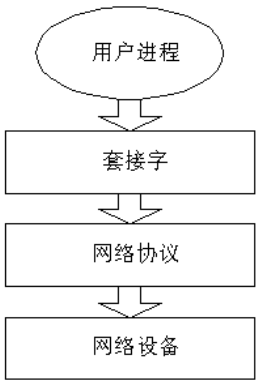

图 12.1　Linux 网络层次模型

每个层次的内部，还可以再细分为很多层次。数据的传输过程只能依照层次的划分，自顶向下进行，不能跨越其中的某个或某些层次，这就使得网络传输只能有一条而且是唯一的一条路径，这样做的目的就是为了提高整个网络的可靠性和准确性。

2．Linux 对以上网络层次的实现采用了面向对象的设计方法，层次模型中的各个层次被抽象为对象，这些对象的详细情况如下所述。

- 网络协议（protoal）。网络协议是一种网络语言，它规定了通信双方之间交换信息的一种规范，它是网络传输的基础。
- 套接字（socket）。一个套接字就是网络中的一个连接，它向用户提供了文件的 I/O，并和网络协议紧密地联系在一起，体现了网络和文件系统、进程管理之间的关系，它是网络传输的入口。
- 设备接口（device and interface）。网络设备接口控制着网络数据由软件—硬件—软件的过程，体现了网络和设备的关系，它是网络传输的桥梁。
- 套接字缓冲区（sh_buff）。网络中的缓冲器叫做套接字缓冲区。它是一块保存网络数据的内存区域，体现了网络和内存管理之间的关系，它是网络传输的灵魂。

这 4 个对象之间的关系如图 12.2 所示。

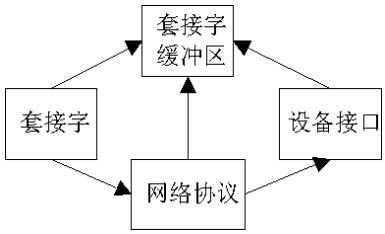

图 12.2　Linux 的网络对象及其之间的关系

从图 12.2 中我们可以看出：这 4 个对象之间的关系是非常紧密的，其中套接字缓冲区的作用非常重要，它和其他 3 个对象均有关系。本章下面的部分将对这 4 个对象及其之间的关系做详细的介绍。

Linux 网络部分为了提高它整体上的兼容性，每一个核心对象都包含了很多种类，为了便于对网络内核的分析，每一个对象我们只选择最常用的一种详细说明，其他种类从略。

12.2　网络协议

一说起网络首先就应该想到网络协议，协议是网络特有的产物，它也是整个网络传输的基础。因为网络协议非常标准规范，它在不同的系统上的用法和工作原理都是一样的，而且介绍协议的书很多，所以，协议这一节不作为本章的重点，我们只是以最常用的一种协议——TCP/IP 为例，简要介绍网络协议的工作原理和过程。

12.2.1　网络参考模型

为了实现各种网络的互连，国际标准化组织（ISO）制定了开放式系统互连（OSI）参考模型。所谓开放，就是指只要遵循标准，一个系统就可以和位于世界上任何地方遵循这一标准的其他任何系统进行通信。OSI 模型提供了一个讨论不同网络协议的参考。

尽管 OSI 的体系结构从理论上讲是比较完整的，但实际上，完全符合 OSI 各层协议的商用产品却很少进入市场。而使用 TCP/IP 的产品却大量涌入市场，几乎所有的工作站都配有

TCP/IP，使得 TCP/IP 成为计算机网络的实际的国际标准。OSI 参考模型和 TCP/IP 参考模型如表 12.1 所示。

表 12.1　　OSI 参考模型和 TCP/IP 参考模型

OSI 参考模型			TCP/IP 参考模型
应用层			应用层
表示层			
对话层			
传输层			传输层
网络层			Internet
数据链路层			网络接口 物理层
物理层			

12.2.2　TCP/IP 工作原理及数据流

TCP/IP 不是一个单独的协议，它是由一组协议组成的协议集，在 TCP/IP 参考模型中各层对应的协议如表 12.2 所示。

表 12.2　　TCP/IP 协议集

应用层	TELNET　FTP　SMTP　DNS
传输层	TCP　　UDP
Internet	IP
网络接口 物理层	ARPANET　　SATNET PPP/SLIP　　LAN

其中最主要的就是 IP（Internet 协议）和 TCP（传输控制协议）。

12.2.3　Internet 协议

IP 不仅是 TCP/IP 的一个重要组成部分，而且也是 OSI 模型的一个基本协议。

IP 定义了一个协议，而不是一个连接，因此与网络连接无关。IP 主要负责数据报在计算机之间的寻址问题，并管理这些数据报的分段过程。该协议在信息数据报格式和由数据报信息组成的报头方面有规范的定义。IP 负责数据报的路由，决定数据报发送到哪里以及在出现问题时更换路由。

IP 数据报的传输具有“不可靠性”，数据报的传输不能受到保障，因为数据报可能会遇到延迟或路由错误，或在数据报分解和重组时遭到破坏。IP 没有能力证实发送的报文是否能被正确的接收，IP 把验证和流量控制的任务交给了分层模型中的其他部件完成。

IP 是无连接的，它不管数据报沿途经过那些节点。它的这些特点都在 IP 报体现。如图

12.3 所示，数据经过 IP 层时，都会被加上 IP 的协议头，其输入／输出是从用户的角度来看的。

TCP/IP 内的数据流

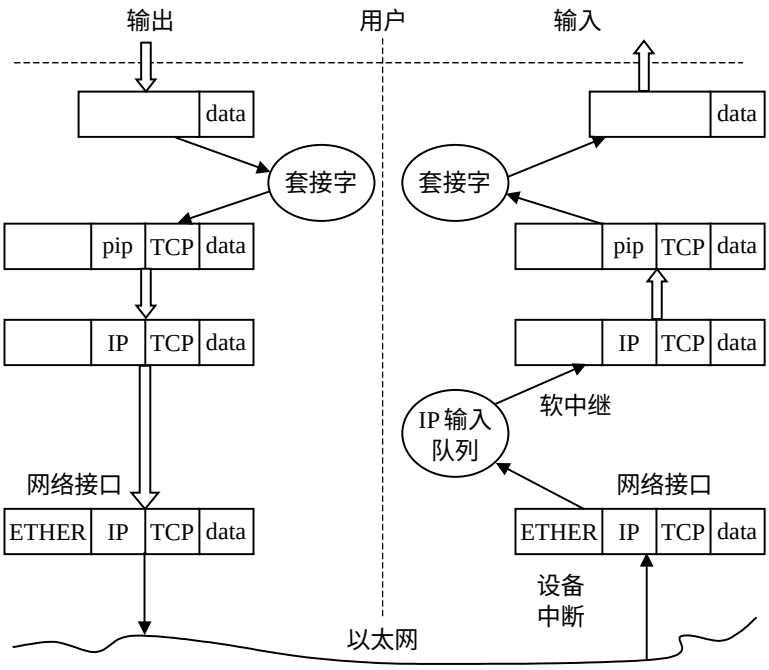

图 12.3　TCP/IP 内数据流

IP 的协议头，也可叫做 IP 数据报或 IP 报头，是 IP 的基本传输单元。IP 协议头的结构如图 12.4 所示。

版本号	长度	服务类型	数据包长度
标识		DF MF	标志偏移量
生存时间	传输		头部校验和
发送地址			
目标地址			
选项			填充

图 12.4　IP 数据报头

12.2.4　TCP

TCP 是传输层中使用最为广泛的一协议，它可以向上层提供面向连接的协议，使上层启动应用程序，以确保网络上所发送的数据报被完整接收。就这种作用而言，TCP 的作用是提

供可靠通信的有效报文协议。一旦数据报被破坏或丢失，通常是 TCP 将其重新传输，而不是应用程序或 IP。

TCP 必须与低层的 IP（使用 IP 定义好的方法）和高层的应用程序（使用 TCP-ULP 元语）进行通信。TCP 还必须通过网络与其他 TCP 软件进行通信。为此，它使用了协议数据单元(PDU)，在 TCP 用语中称为分段。

TCP PDU（通常称为 TCP 报头）的分布如图 12.5 所示。

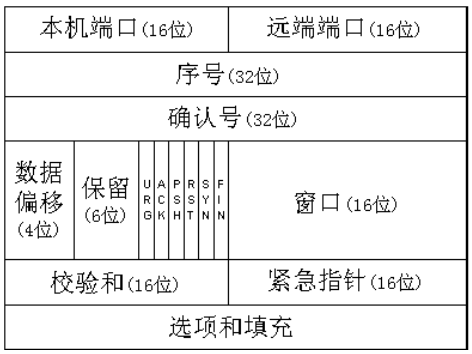

图 12.5　TCP 数据单元

部分域含义如下。

- 本机端口：标识本机 TCP 用户（通常为上层应用程序）的 16 位域。
- 远端端口：标识远程计算机 TCP 用户的 16 位域。
- 序号：指明当前时钟在全文中位置的序号。也可用在两个 TCP 之间以提供初始发送序号（ISS）。
- 确认号：指明下一个预计序列的序号。反过来，它还可以表示最后接收数据的序号，表示最后接收的序号加 1。
- 数据偏移：用于标识数据段的开始。
- URG：如果打开（值为 1），则指明紧急指针域有效。
- ACK：如果打开，则指明确认域有效。
- RST：如果打开，则指明要重复连接。
- SYN：如果打开，则指明要同步的序号。
- FI N：如果打开，则指明发送双方不再发送数据。这与传输结束标志是相同的。

这些域在 TCP 连接和传输数据时会用到。

TCP 对如何通信有许多规则。这些规则以及 TCP 连接、传输要遵循的过程，通常都体现在状态数据报中（因为 TCP 是一个状态驱动协议，其行为取决于状态标志或类似结构）。要完全避免复杂的状态数据报是很困难的，所以流程图对理解 TCP 是一种很有效的方法。下面我们就以 TCP 连接的流程图为例，介绍 TCP 的工作原理。如图 12.6 所示。

此过程以计算机 A 的 TCP 开始，TCP 可从它的 ULP 接收连接请求，通过它向计算机 B 发送一个主动打开原语，所构成的分段应设置 SYN 标志（值为 1），并分配一个序列号 M。图 12.6 用 SYN 50 表示，SYN 标志打开，序号 M 用 50 表示，可任意选择。

计算机 B 上的应用程序将向它的 TCP 发送一个被动打开指令，当接收到 SYN M 分段时，

计算机 B 上的 TCP 将序号 M+1 发回一个确认给计算机 A，图 12.6 用 ACK 51 表示。计算机 B 也为自己设置一个初始发送序号 N，图 12.6 用 SYN 200 表示。

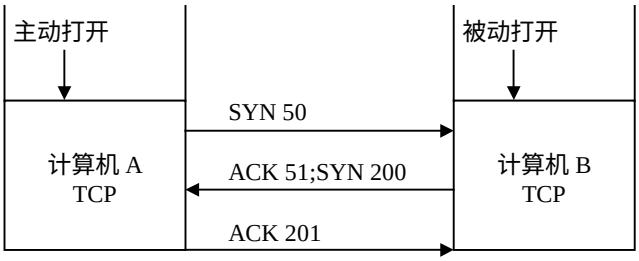

图 12.6　TCP 连接示意图

计算机 A 根据接收到的内容，通过将序号设置为 N+1 ,发回他自己的确认报文，图 12.6 用 ACK 201 表示。然后，打开并确认此次连接，计算机 A 和计算机 B 通过 ULP 将连接打开报文发送到请求的应用程序。

至此两台计算机建立了连接，可以在 TCP 层传输数据。

12.3　套接字（socket）

12.3.1　套接字在网络中的地位和作用

socket 在所有的网络操作系统中都是必不可少的，而且在所有的网络应用程序中也是必不可少的。它是网络通信中应用程序对应的进程和网络协议之间的接口，如图 12.7 所示。

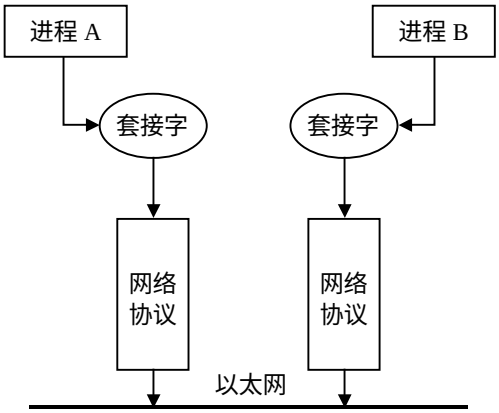

图 12.7　套接字在网络系统中的地位

socket 在网络系统中的作用如下。

（1）socket 位于协议之上，屏蔽了不同网络协议之间的差异。

（2）socket 是网络编程的入口，它提供了大量的系统调用，构成了网络程序的主体。

（3）在 Linux 系统中，socket 属于文件系统的一部分，网络通信可以被看作是对文件的读取，使得我们对网络的控制和对文件的控制一样方便。

12.3.2　套接字接口的种类

Linux 支持多种套接字种类，不同的套接字种类称为“地址族”，这是因为每种套接字种类拥有自己的通信寻址方法。Linux 所支持的套接字地址族见表 12.3。

Linux 将上述套接字地址族抽象为统一的 BSD 套接字接口，应用程序关心的只是 BSD 套接字接口，而 BSD 套接字由各地址族专有的软件支持。一般而言，BSD 套接字可支持多种套接字类型，不同的套接字类型提供的服务不同，Linux 所支持的部分 BSD 套接字类型见表 12.4，但表 12.3 中的套接字地址族并不一定全部支持表 12.4 中的这些套接字类型。

表 12.3　　Linux 支持的套接字地址族

套接字地址族	描述
UNIX	UNIX 域套接字
INET	通过 TCP/IP 协议支持的 Internet 地址族
AX25	Amater radio X25
APPLETALK	Appletalk DDP
IPX	Novell IPX
X25	X25

表 12.4　　Linux 所支持的 BSD 套接字类型

BSD 套接字类型	描述
流（stream）	这种套接字提供了可靠的双向顺序数据流，可保证数据不会在传输过程中丢失、破坏或重复出现。流套接字通过 INET 地址族的 TCP 协议实现
数据报（datagram）	这种套接字也提供双向的数据传输，但是并不对数据的传输提供担保，也就是说，数据可能会以错误的顺序传递，甚至丢失或破坏。这种类型的套接字通过 INET 地址族的 UDP 协议实现
原始（raw）	利用这种类型的套接字，进程可以直接访问底层协议（因此称为原始）。例如，可在某个以太网设备上打开原始套接字，然后获取原始的 IP 数据传输信息
可靠发送的消息	和数据报套接字类似，但保证数据被正确传输到目的端
顺序数据包	和流套接字类似，但数据包大小是固定的
数据包（packet）	这并不是标准的 BSD 套接字类型，它是 Linux 专有的 BSD 套接字扩展，可允许进程直接在设备级访问数据包

下面我们以 INET 套接字地址族、流套接字类型为例，详细介绍套接字的工作原理和通信过程。

12.3.3 套接字的工作原理

INET 套接字就是支持 Internet 地址族的套接字，它位于 TCP 之上，BSD 套接字之下，如图 12.8 所示，这里也体现了 Linux 网络模块分层的设计思想。

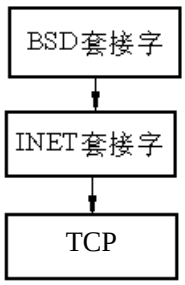

图 12.8 INET 套接字

INET 和 BSD 套接字之间的接口通过 Internet 地址族套接字操作集实现，这些操作集实际是一组协议的操作例程，在 include/linux/net.h 中定义为 proto_ops：

```
    struct proto_ops {
       int   family;

       int   (*release)      (struct socket *sock);
       int   (*bind)         (struct socket *sock, struct sockaddr *umyaddr,
                              int sockaddr_len);
       int   (*connect)      (struct socket *sock, struct sockaddr *uservaddr,
                              int sockaddr_len, int flags);
       int   (*socketpair)   (struct socket *sock1, struct socket *sock2);
       int   (*accept)       (struct socket *sock, struct socket *newsock,
                              int flags);
       int   (*getname)      (struct socket *sock, struct sockaddr *uaddr,
                              int *usockaddr_len, int peer);
    unsigned int (*poll)  (struct file *file, struct socket *sock, struct poll_table_struct
*wait);
       int   (*ioctl)        (struct socket *sock, unsigned int cmd,
                              unsigned long arg);
      int   (*listen)       (struct socket *sock, int len);
      int   (*shutdown)     (struct socket *sock, int flags);
      int   (*setsockopt)   (struct socket *sock, int level, int optname,
                             char *optval, int optlen);
      int   (*getsockopt)   (struct socket *sock, int level, int optname,
                             char *optval, int *optlen);
    int    (*sendmsg)       (struct socket *sock, struct msghdr *m, int  total_len, struct
scm_cookie *scm);
    int   (*recvmsg)      (struct socket *sock, struct msghdr *m, int total_len, int flags,
struct scm_cookie *scm);
    int   (*mmap)         (struct file *file, struct socket *sock, struct vm_area_struct * vma);
    ssize_t (*sendpage)   (struct socket *sock, struct page *page, int offset, size_t size, int
```

```
flags);
    };
```

这个操作集类似于文件系统中的 file_operations 结构。BSD 套接字层通过调用 proto_ops 结构中的相应函数执行任务。BSD 套接字层向 INET 套接字层传递 socket 数据结构来代表一个 BSD 套接字，socket 结构在 include/linux/net.h 中定义如下：

```
    struct socket
    {
           socket_state            state;

           unsigned long           flags;
          struct proto_ops        *ops;
          struct inode            *inode;
          struct fasync_struct    *fasync_list; /* Asynchronous wake up list    */
          struct file             *file;       /* File back pointer for gc     */
          struct sock             *sk;
          wait_queue_head_t       wait;

          short                   type;
          unsigned char           passcred;
      };
```

但在 INET 套接字层中，它利用自己的 sock 数据结构来代表该套接字，因此，这两个结构之间存在着链接关系，sock 结构定义于 include/net/sock.h（此结构有 80 多行，在此不予列出）。在 BSD 的 socket 数据结构中存在一个指向 sock 的指针 sk，而在 sock 中又有一个指向 socket 的指针，这两个指针将 BSD socket 数据结构和 sock 数据结构链接了起来。通过这种链接关系，套接字调用就可以方便地检索到 sock 数据结构。实际上，sock 数据结构可适用于不同的地址族，它也定义有自己的协议操作集 proto。在建立套接字时，sock 数据结构的协议操作集指针指向所请求的协议操作集。如果请求 TCP，则 sock 数据结构的协议操作集指针将指向 TCP 的协议操作集。

进程在利用套接字进行通信时，采用客户/服务器模型。服务器首先创建一个套接字，并将某个名称绑定到该套接字上，套接字的名称依赖于套接字的底层地址族，但通常是服务器的本地地址。套接字的名称或地址通过 sockaddr 数据结构指定，该结构定义于 include/linux/socket.h 中：

```
    struct sockaddr {
           sa_family_t     sa_family;      /* address family, AF_xxx       */
           char            sa_data[14];    /* 14 bytes of protocol address */
    };
```

对于 INET 套接字来说，服务器的地址由两部分组成，一个是服务器的 IP 地址，另一个是服务器的端口地址。已注册的标准端口可查看 /etc/services 文件。将地址绑定到套接字之后，服务器就可以监听请求链接该绑定地址的传入连接。连接请求由客户生成，它首先建立一个套接字，并指定服务器的目标地址以请求建立连接。传入的连接请求通过不同的协议层最终到达服务器的监听套接字。服务器接收到传入的请求后，如果能够接受该请求，服务器必须创建一个新的套接字来接受该请求并建立通信连接（用于监听的套接字不能用来建立通信连接），这时，服务器和客户就可以利用建立好的通信连接传输数据。

BSD 套接字上的详细操作与具体的底层地址族有关，底层地址族的不同实际意味着寻址

方式、采用的协议等的不同。Linux 利用 BSD 套接字层抽象了不同的套接字接口。在内核的初始化阶段，内建于内核的不同地址族分别以 BSD 套接字接口在内核中注册。然后，随着应用程序创建并使用 BSD 套接字。

内核负责在 BSD 套接字和底层的地址族之间建立联系。这种联系通过交叉链接数据结构以及地址族专有的支持例程表建立。

在内核中，地址族和协议信息保存在 inet_protos 向量中，其定义于 include/net/protocol.h：

```
        struct inet_protocol *inet_protos[MAX_INET_PROTOS];

        /* This is used to register protocols. */
    struct inet_protocol
    {
             int                     (*handler)(struct sk_buff *skb);
             void                    (*err_handler)(struct sk_buff *skb, u32 info);
            struct inet_protocol    *next;
             unsigned char           protocol;
             unsigned char           copy:1;
            void                    *data;
            const char              *name;
    };
```

每个地址族由其名称以及相应的初始化例程地址代表。在引导阶段初始化套接字接口时，内核调用每个地址族的初始化例程，这时，每个地址族注册自己的协议操作集。协议操作集实际是一个例程集合，其中每个例程执行一个特定的操作。

12.3.4 socket 的通信过程

请先看如图 12.9 所示的 socket 通信过程。

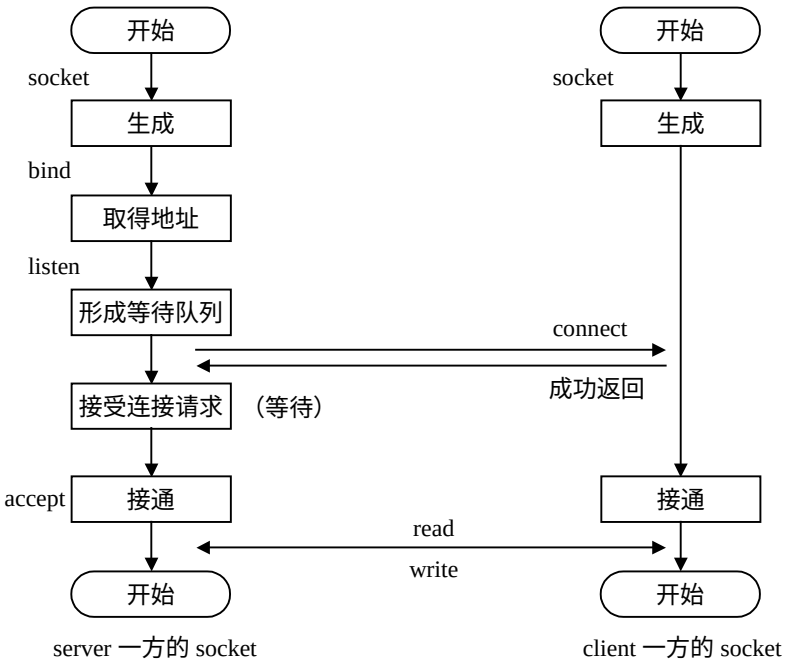

图 12.9　socket 的通信过程

1．建立套接字

Linux 在利用 socket()系统调用建立新的套接字时，需要传递套接字的地址族标识符、套接字类型以及协议，其函数定义于 net/socket.c 中：

```
asmlinkage long sys_socket(int family, int type, int protocol)
{
     int retval;
     struct socket *sock;

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

      retval = sock_map_fd(sock);
      if (retval < 0)
             goto out_release;

out:
      /* It may be already another descriptor 8) Not kernel problem. */
     return retval;

out_release:
      sock_release(sock);
     return retval;
}
```

实际上，套接字对于用户程序而言就是特殊的已打开的文件。内核中为套接字定义了一

种特殊的文件类型，形成一种特殊的文件系统 sockfs，其定义于 net/socket.c：

```
static struct vfsmount *sock_mnt;
static DECLARE_FSTYPE(sock_fs_type, "sockfs",sockfs_read_super, FS_NOMOUNT);
```

在系统初始化时，要通过 kern_mount()安装这个文件系统。安装时有个作为连接件的 vfsmount 数据结构，这个结构的地址就保存在一个全局的指针 sock_mnt 中。所谓创建一个套接字，就是在 sockfs 文件系统中创建一个特殊文件，或者说一个节点，并建立起为实现套接字功能所需的一整套数据结构。所以，函数 sock_create()首先是建立一个 socket 数据结构，然后将其“映射”到一个已打开的文件中，进行 socket 结构和 sock 结构的分配和初始化。

新创建的 BSD socket 数据结构包含有指向地址族专有的套接字例程的指针，这一指针实际就是 proto_ops 数据结构的地址。

BSD 套接字的套接字类型设置为所请求的 SOCK_STREAM 或 SOCK_DGRAM 等。然后，内核利用 proto_ops 数据结构中的信息调用地址族专有的创建例程。

之后，内核从当前进程的 fd 向量中分配空闲的文件描述符，该描述符指向的 file 数据结构被初始化。初始化过程包括将文件操作集指针指向由 BSD 套接字接口支持的 BSD 文件操作集。所有随后的套接字（文件）操作都将定向到该套接字接口，而套接字接口则会进一步调用地址族的操作例程，从而将操作传递到底层地址族，如图 12.10 所示。

实际上，socket 结构与 sock 结构是同一事物的两个方面。如果说 socket 结构是面向进程和系统调用界面的，那么 sock 结构就是面向底层驱动程序的。可是，为什么不把这两个数据结构合并成一个呢？

我们说套接字是一种特殊的文件系统，因此，inode 结构内部的 union 的一个成分就用作 socket 结构，其定义如下：

```
struct inode  {
    …
  union {
     …
          struct socket             socket_i;
       }
  }
```

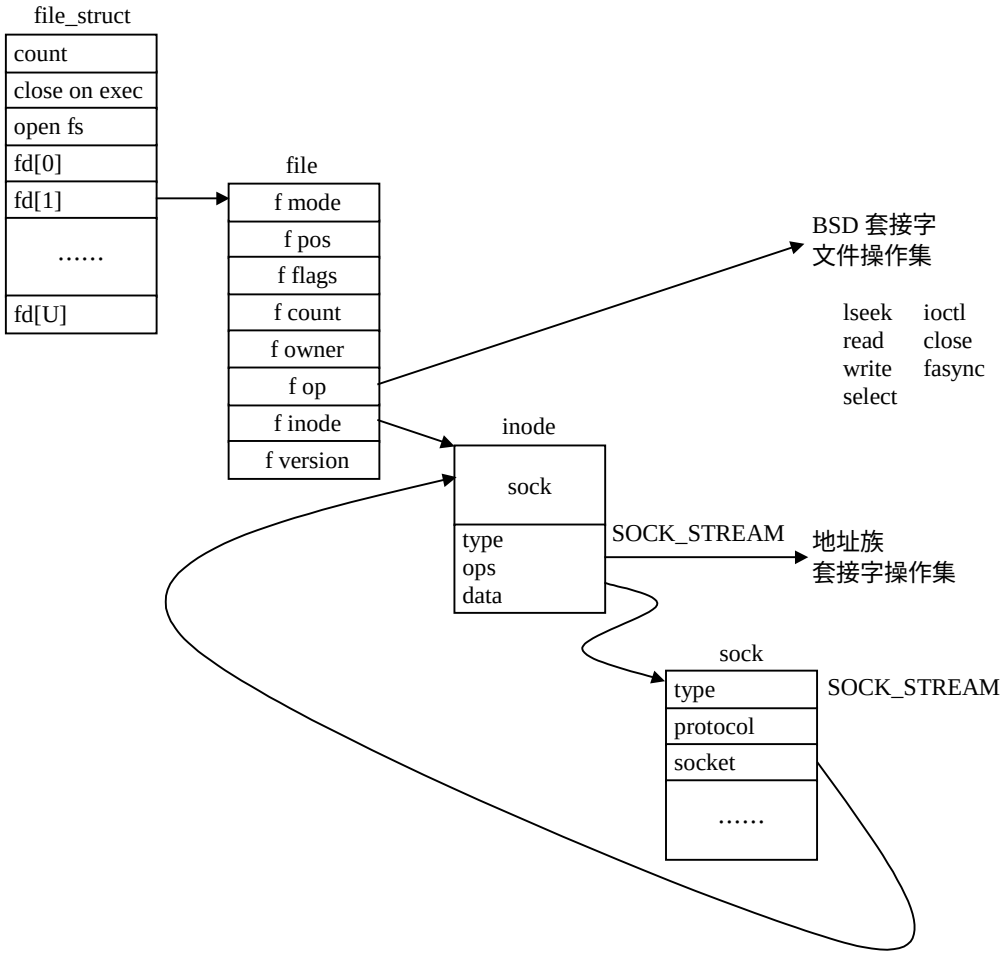

图 12.10　socket 在文件系统 inode 中的位置

由于套接字操作的特殊性，这个结构中需要大量的结构成分。可是，如果把这些结构成分全都放在 socket 结构中，则 inode 结构中的这个 union 就会变得很大，从而 inode 结构也会变得很大，而对于其他文件系统，这个 union 成分并不需要那么庞大。因此，就把套接字所需的这些结构成分拆成两部分，把与文件系统关系比较密切的那一部分放在 socket 结构中，把与通信关系比较密切的那一部分则单独组成一个数据结构，即 sock 结构。由于这两部分数据在逻辑上本来就是一体的，所以要通过指针互相指向对方，形成一对一的关系。

2．在 INET BSD 套接字上绑定（bind）地址

为了监听传入的 Internet 连接请求，每个服务器都需要建立一个 INET BSD 套接字，并且将自己的地址绑定到该套接字。绑定操作主要在 INET 套接字层中进行，还需要底层 TCP 层和 IP 层的某些支持。将地址绑定到某个套接字上之后，该套接字就不能用来进行任何其他的通信，因此，该 socket 数据结构的状态必须为 TCP_CLOSE。传递到绑定操作的 sockaddr 数据结构中包含要绑定的 IP 地址，以及一个可选的端口地址。通常而言，要绑定的地址应该是赋予某个网络设备的 IP 地址，而该网络设备应该支持 INET 地址族，并且该设备是可

用的。利用 ifconfig 命令可查看当前活动的网络接口。被绑定的 IP 地址保存在 sock 数据结构的 rcv_saddr 和 saddr 域中，这两个域分别用于哈希查找和发送用的 IP 地址。端口地址是可选的，如果没有指定，底层的支持网络会选择一个空闲的端口。

当底层网络设备接收到数据包时，它必须将数据包传递到正确的 INET 和 BSD 套接字以便进行处理，因此，TCP 维护多个哈希表，用来查找传入 IP 消息的地址，并将它们定向到正确的 socket/sock 对。TCP 并不在绑定过程中将绑定的 sock 数据结构添加到哈希表中，在这一过程中，它仅仅判断所请求的端口号当前是否正在使用。在监听操作中，该 sock 结构才被添加到 TCP 的哈希表中。

3．在 INET BSD 套接字上建立连接（connect）

创建一个套接字之后，该套接字不仅可以用于监听入站的连接请求，也可以用于建立出站的连接请求。不论怎样都涉及到一个重要的过程：建立两个应用程序之间的虚拟电路。出站连接只能建立在处于正确状态的 INET BSD 套接字上，因此，不能建立于已建立连接的套接字，也不能建立于用于监听入站连接的套接字。也就是说，该 BSD socket 数据结构的状态必须为 SS_UNCONNECTED。

在建立连接过程中，双方 TCP 要进行 3 次“握手”，具体过程在本章 12.2 节——网络协议一中有详细介绍。如果 TCP sock 正在等待传入消息，则该 sock 结构添加到 tcp_listening_hash 表中，这样，传入的 TCP 消息就可以定向到该 sock 数据结构。

4．监听（listen）INET BSD 套接字

当某个套接字被绑定了地址之后，该套接字就可以用来监听专属于该绑定地址的传入连接。网络应用程序也可以在未绑定地址之前监听套接字，这时，INET 套接字层将利用空闲的端口编号并自动绑定到该套接字。套接字的监听函数将 socket 的状态改变为 TCP_LISTEN。

当接收到某个传入的 TCP 连接请求时，TCP 建立一个新的 sock 数据结构来描述该连接。当该连接最终被接受时，新的 sock 数据结构将变成该 TCP 连接的内核 bottom_half 部分，这时，它要克隆包含连接请求的传入 sk_buff 中的信息，并在监听 sock 数据结构的 receive_queue 队列中将克隆的信息排队。克隆的 sk_buff 中包含有指向新 sock 数据结构的指针。

5．接受连接请求（accept）

接受操作在监听套接字上进行，从监听 socket 中克隆一个新的 socket 数据结构。其过程如下：接受操作首先传递到支持协议层，即 INET 中，以便接受任何传入的连接请求。相反，接受操作进一步传递到实际的协议，例如 TCP 上。接受操作可以是阻塞的，也可以是非阻塞的。接受操作为非阻塞的情况下，如果没有可接受的传入连接，则接受操作将失败，而新建立的 socket 数据结构被抛弃。接受操作为阻塞的情况下，执行阻塞操作的网络应用程序将添加到等待队列中，并保持挂起直到接收到一个 TCP 连接请求为至。当连接请求到达之后，包含连接请求的 sk_buff 被丢弃，而由 TCP 建立的新 sock 数据结构返回到 INET 套接字层，在这里，sock 数据结构和先前建立的新 socket 数据结构建立链接。而新 socket 的文件描述符（fd）被返回到网络应用程序，此后，应用程序就可以利用该文件描述符在新建

立的 INET BSD 套接字上进行套接字操作。

12.3.5　socket 为用户提供的系统调用

socket 系统调用是 socket 最有价值的一部分，也是用户唯一能够接触到的一部分，它是我们进行网络编程的接口。如表 12.5 所示。

表 12.5　　socket 系统调用

系 统 调 用	说　　明
Accept	接收套接字上连接请求
Bind	在套接字绑定地址信息
Connet	连接两个套接字
Getpeername	获取已连接端套接字的地址
Getsockname	获取套接字的地址
Getsockopt	获取套接字上的设置选项
Listen	监听套接字连接
Recv	从已连接套接字上接收消息
Recvfrom	从套接字上接收消息
Send	向已连接的套接字发送消息
Sendto	向套接字发送消息
Setdomainname	设置系统的域名
Sethostid	设置唯一的主机标识符
Sethostname	设置系统的主机名称
Setsockopt	修改套接字选项
Shutdown	关闭套接字
Socket	建立套接字通信的端点
Socketcall	套接字调用多路复用转换器
Socketpair	建立两个连接套接字

12.4　套接字缓冲区（sk_buff）

套接字缓冲区是网络部分一个重要的数据结构，它描述了内存中的一块数据区域，该数据区域存放着网络传输的数据包。在整个网络传输中，套接字缓冲区作为数据的载体，保证了数据传输的可靠和稳定，而且，网络部分各层次都和该数据结构密切相关，由此可见，套

接字缓冲区在网络传输过程中的作用举足轻重，对它的理解，将是我们对网络内核进行分析的主要内容。

12.4.1 套接字缓冲区的特点

套接字缓冲区和其他部分的缓冲区相比，它有自己的特点。在网络传输的源主机上，它创建于套接字层（其名字的来历），沿网络层自上而下传递，它先在协议层流动，最后在物理层消失，同时把它所带的数据传递给目标主机的物理层的套接字缓冲区，该缓冲区自下而上传递到目标主机的套接字层，并把数据传递给用户进程，目标主机的套接字缓冲区也同时消失。请参看图 12.11 所示的示例。

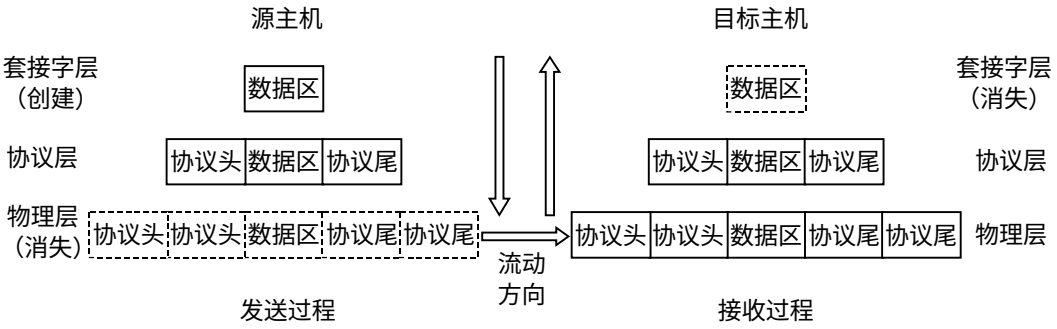

图 12.11 套接字缓冲区流程图

当套接字缓冲区在协议层流动过程中，每个协议都需要对数据区的内容进行修改，也就是每个协议都需要在发送数据时向缓冲区添加自己的协议头和协议尾，而在接收数据时去掉这些协议头和协议尾，这样就存在一个问题，当缓冲区在不同的协议之间传递时，每层协议都要寻找自己特定的协议头和协议尾，从而导致数据缓冲区的传递非常困难。我们设置 sk_buff 数据结构的主要目的就是为网络部分提供一种统一有效的缓冲区操作方法，从而可让协议层以标准的函数或方法对缓冲区数据进行处理，这是 Linux 系统网络高效运行的关键。

12.4.2 套接字缓冲区操作基本原理

在传输过程中，存在着多个套接字缓冲区，这些缓冲区组成一个链表，每个链表都有一个链表头 sk_buff_head ，链表中每个节点分别对应内存中一块数据区。因此对它的操作有两种基本方式：第 1 种是对缓冲区链表进行操作；第 2 种是对缓冲区对应的数据区进行控制。

当我们向物理接口发送数据时或当我们从物理接口接收数据时，我们就利用链表操作；当我们要对数据区的内容进行处理时，我们就利用内存操作例程。这种操作机制对网络传输是非常有效的。

前面我们讲过，每个协议都要在发送数据时向缓冲区添加自己的协议头和协议尾，而在

接收数据时去掉协议头和协议尾，那么具体的操作是怎样进行的呢？我们先看看对缓冲区操作的两个基本的函数：

```
    void append_frame(char *buf, int len){
        struct sk_buff *skb=alloc_skb(len, GFP_ATOMIC);     /*创建一个缓冲区*/
       if(skb==NULL)
                  my_dropped++;
       else    {
                       kb_put(skb,len);
                       memcpy(skb->data,data,len);      /*向缓冲区添加数据*/
                      skb_append(&my_list, skb);  /*将该缓冲区加入缓冲区队列*/
                  }
      }
    void process_frame(void){
        struct sk_buff *skb;
        while((skb=skb_dequeue(&my_list))!=NULL)
         {
    process_data(skb);                                          /*将缓冲区的数据传递给协
议层*/
    kfree_skb(skb, FREE_READ);                                  /*释放缓冲区，缓冲区从此消失*/
         }
     }
```

这两个非常简单的程序片段，虽然它们不是源程序，但是它们恰当地描述了处理数据包的工作原理，append_frame( )描述了分配缓冲区。创建数据包过程 process_frame( )描述了传递数据包，释放缓冲区的的过程。关于它们的源程序，可以去参见 net/core/dev.c 中 netif_rx( )函数和 net_bh( )函数。你可以看出它们和上面我们提到的两个函数非常相似。这两个函数非常复杂,因为他们必须保证数据能够被正确的协议接收并且要负责流程的控制，但是他们最基本的操作是相同的。

让我们再看看上面提到的函数——append_frame( )。当 alloc_skb( ) 函数获得一个长度为 len 字节的缓冲区（如图 12.12 (a)所示）后，该缓冲区包含以下内容：

- 缓冲区的头部有零字节的头部空间；
- 零字节的数据空间；
- 缓冲区的尾部有零字节的尾部空间。

再看 skb_put( )函数（如图 12.12 (d)所示），它的作用是从数据区的尾部向缓冲区尾部不断扩大数据区大小，为后面的 memcpy( )函数分配空间。

当一个缓冲区创建以后，所有的可用空间都在缓冲区的尾部。在没有向其中添加数据之前，首先被执行的函数调用是 skb_reserve( )（如图 12.12 (b)所示）,它使你在缓冲区头部指定一定的空闲空间，因此许多发送数据的例程都是这样开头的：

```
skb=alloc_skb(len+headspace, GFP_KERNEL);

skb_reserve(skb, headspace);
skb_put(skb,len);
memcpy_fromfs(skb->data,data,len);
pass_to_m_protocol(skb);
```

图 12.12 向我们展示了以上过程进行时，sk_buff 的变化情况。

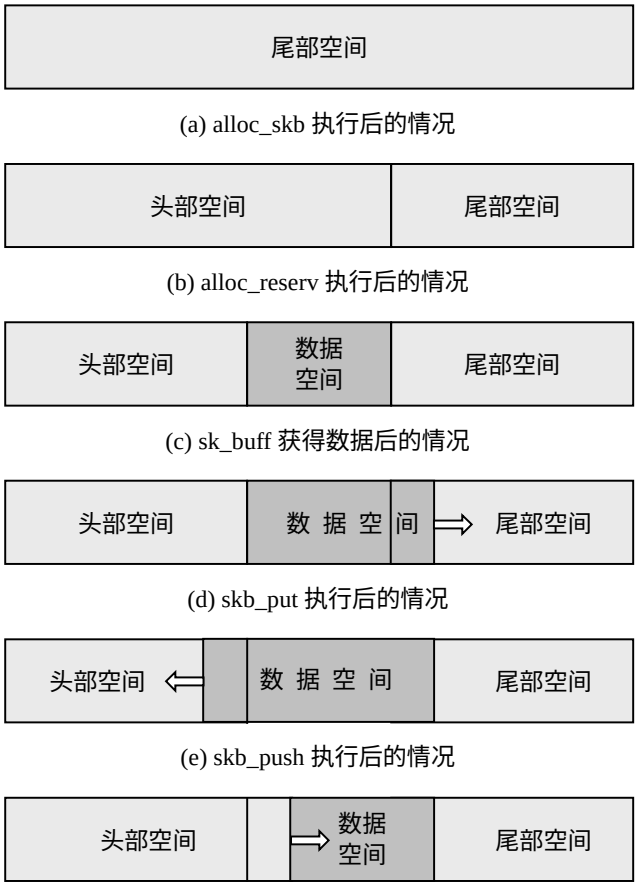

图 12.12　sk_buff 的变化过程

12.4.3　sk_buff 数据结构的核心内容

sk_buff 数据结构中包含了一些指针和长度信息，从而可让协议层以标准的函数或方法对应用程序的数据进行处理，其定义于 include/linux/skbuff.h 中：

```
struct sk_buff {
         /* These two members must be first. */
        struct sk_buff  * next;       /* Next buffer in list*/
        struct sk_buff  * prev;        /* Previous buffer in list*/

         struct sk_buff_head * list;      /* List we are on */
          struct sock     *sk;           /* Socket we are owned by */
              struct timeval  stamp;                  /* Time we arrived */
struct net_device       *dev;         /* Device we arrived on/are leaving by  */

         /* Transport layer header */
       union
```

```
        {
                struct tcphdr   *th;
                struct udphdr   *uh;
                struct icmphdr  *icmph;
                struct igmphdr  *igmph;
                struct iphdr    *ipiph;
                 struct spxhdr   *spxh;
                 unsigned char   *raw;
         } h;

         /* Network layer header */
         union
         {
                struct iphdr    *iph;
                struct ipv6hdr  *ipv6h;
                struct arphdr   *arph;
                struct ipxhdr   *ipxh;
                 unsigned char   *raw;
        } nh;

        /* Link layer header */
         union
         {
                 struct ethhdr   *ethernet;
                 unsigned char   *raw;
         } mac;

        struct  dst_entry *dst;

        /*
          * This is the control buffer. It is free to use for every
          * layer. Please put your private variables there. If you
         * want to keep them across layers you have to do a skb_clone()
         * first. This is owned by whoever has the skb queued ATM.
          */
    char            cb[48];

    unsigned int    len;          /* Length of actual data*/
    unsigned int    data_len;
    unsigned int    csum;             /* Checksum   */
    unsigned char   __unused,       /* Dead field, may be reused  */
cloned, /* head may be cloned (check refcnt to be sure). */
                pkt_type,         /* Packet class   */
               ip_summed;        /* Driver fed us an IP checksum    */
  __u32           priority;      /* Packet queueing priority */
       atomic_t        users;     /* User count - see datagram.c,tcp.c */
       unsigned short  protocol;    /* Packet protocol from driver.  */
       unsigned short  security;   /* Security level of packet */
      unsigned int    truesize;         /* Buffer size */
      unsigned char   *head;         /* Head of buffer  */
      unsigned char   *data;                  /* Data head pointer
      unsigned char   *tail;                  /* Tail pointer
```

```
    unsigned char   *end;                /* End pointer  */
        void      (*destructor)(struct sk_buff *);  /* Destruct function  */
    ...
  }
```

该结构的示意图如图 12.13 所示。

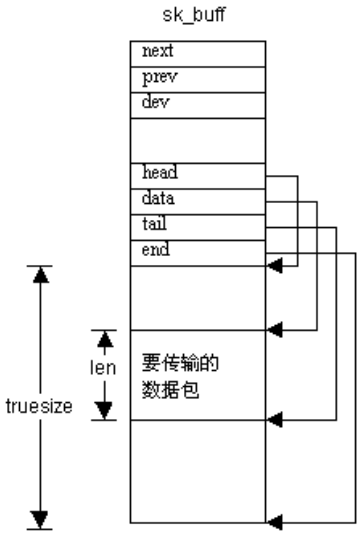

图 12.13　sk_buff 结构

每个 sk_buff 均包含一个数据块、4 个数据指针以及两个长度字段。利用 4 个数据指针，各协议层可操纵和管理套接字缓冲区的数据，这 4 个指针的用途如下所述。

head：指向内存中数据区的起始地址。sk_buff 和相关数据块在分配之后，该指针的值是固定的。

data：指向协议数据的当前起始地址。该指针的值随当前拥有 sk_buff 的协议层的变化而变化。

tail：指向协议数据的当前结尾地址。和 data 指针一样，该指针的值也随当前拥有 sk_buff 的协议层的变化而变化。

end：指向内存中数据区的结尾。和 head 指针一样，sk_buff 被分配之后，该指针的值也固定不变。

sk_buff 有两个非常重要长度字段，len 和 truesize，分别描述当前协议数据包的长度和数据缓冲区的实际长度。

12.4.4　套接字缓冲区提供的函数

1．操纵 sk_buff 链表的函数

sk_buff 链表是一个双向链表，它包括一个链表头而且每一个缓冲区都有一个 prev 和 next 指针，指向链表中前一个和后一个缓冲区节点。

```
struct sk_buff *skb_dequeue(struct skb_buff_head *list)
```

这个函数作用是把第 1 个缓冲区从链表中移走。返回取出的 sk_buff，如果队列为空，就返回空指针。添加缓冲区用到 skb_queue_head 和 skb_queue_tail 两个例程。

```
int skb_peek(struct sk_buff_head *list)
```

返回指向缓冲区链表第 1 个节点的指针。

```
int skb_queue_empty(struct sk_buff_head *list)
```

如果链表为空，返回 true 。

```
void skb_queue_head(struct sk_buff *skb)
```

这个函数在链表头部添加一个缓冲区。

```
void skb_queue_head_init(struct sk_buff_head *list)
```

初始化 sk_buff_head 结构 。该函数必须在所有的链表操作之前调用，而且它不能被重复执行。

```
__u32 skb_queue_len(struct sk_buff_head *list)
```

返回队列中排队的缓冲区的数目。

```
void skb_queue_tail(struct sk_buff *skb)
```

这个函数在链表的尾部添加一个缓冲区，这是在缓冲区操作函数中最常用的一个函数。

```
void skb_unlink(struct sk_buff *skb)
```

这个函数从链表中移去一个缓冲区。它只是将缓冲区从链表中移去，但并不释放它。

许多更复杂的协议，如 TCP 协议，当它接收到数据时，需要保持链表中数据帧的顺序或对数据帧进行重新排序。有两个函数完成这些工作：

```
void skb_append(struct sk_buff *entry, struct sk_buff *new_entry)
void skb_insert(struct sk_buff *entry, struct sk_buff *new_entry)
```

它们可以使用户把一个缓冲区放在链表中任何一个位置。

2．创建或取消一个缓冲区结构的函数

这些操作用到内存处理方法，它们的正确使用对管理内存非常重要。sk_buff 结构的数量和它们占用内存大小会对机器产生很大的影响，因为网络缓冲区的内存组合是最主要一种的系统内存组合。

```
struct sk_buff *alloc_skb(int size, int priority)
```

创建一个新的 sk_buff 结构并将它初始化。

```
void kfree_skb(struct sk_buff *skb, int rw)
```

释放一个 skb_buff。

```
struct sk_buff *skb_clone(struct sk_buff *old, int priority)
```

复制一个 sk_buff，但不复制数据部分。

```
struct sk_buff *skb_copy(struct sk_buff *skb)
```

完全复制一个 sk_buff。

3．对 sk_buff 结构数据区进行操作的操作

这些函数用到了套接字结构体中两个域：缓冲区长度(skb->len) 和缓冲区中数据包的实际起始地址 (skb->data)。这些两个域对用户来说是可见的，而且它们具有只读属性。

```
unsigned char *skb_headroom(struct sk_buff *skb)
```

返回 sk_buff 结构头部空闲空间的字节数大小。

```
unsigned char *skb_pull(struct sk_buff *skb, int len)
```

该函数将 data 指针向数据区的末尾移动，减少了 len 字段的长度。该函数可用于从接收到的数据头上移去数据或协议头。

unsigned char *skb_push(struct sk_buff *skb, int len)

该函数将 data 指针向数据区的前端移动，增加 了 len 字段的长度。在发送数据的过程中，利用该函数可在数据的前端添加数据或协议头。

unsigned char *skb_put(struct sk_buff *skb, int len)

该函数将 tail 指针向数据区的末尾移动，增加了 len 字段的长度。在发送数据的过程中，利用该函数可在数据的末端添加数据或协议尾。

unsigned char *skb_reserve(struct sk_buff *skb, int len)

该函数在缓冲区头部创建一块额外的空间，这块空间在 skb_push 添加数据时使用。因为套接字建立时并没有为 skb_push 预留空间。它也可以用于在缓冲区的头部增加一块空白区域，从而调整缓冲区的大小，使缓冲区的长度统一。这个函数只对一个空的缓冲区才能使用。

unsigned char *skb_tailroom(struct sk_buff *skb)

返回 sk_buff 尾部空闲空间的字节数大小。

unsigned char *skb_trim(struct sk_buff *skb, int len)

该函数和 put 函数的功能相反，它将 tail 指针向数据区的前端移动，减小了 len 字段的长度。该函数可用于从接收到的数据尾上移去数据或协议尾。如果缓冲区的长度比“ len ”还长，那么它就通过移去缓冲区尾部若干字节，把缓冲区的大小缩减到“ len ”长度。

12.4.5 套接字缓冲区的上层支持例程

我们上面讲了套接字缓冲区基本的操作方法，利用它们就可以完成数据包的发送和接收工作。为了保证网络传输的高效和稳定，我们需要对整个过程进行流程控制，因此，我们又引进了两个支持例程。它们是利用信号的交互来完成任务的。

sock_queue_rcv_skb（）函数用来对数据的接收进行控制，通常调用它的的形式为：

```
sk=my_find_socket(whatever);
if(sock_queue_rcv_skb(sk,skb)==-1)
{
  myproto_stats.dropped++;
  kfree_skb(skb,FREE_READ);
  return;
 }
```

它利用套接字的读队列的计数器，从而避免了大量的数据包堆积在套接字层。一旦到达这个极限，其余的数据包就会被丢弃。这样做是为了保障高层的应用协议有足够快的读取速度，比如 TCP，包含对该流程的控制，当接收端不能再接收数据时，TCP 就告诉发送端的机器停止传输。

在数据传输方面，sock_alloc_send_skb（） 可以对发送队列进行控制，我们不能把所有的缓冲区都填充数据，使得发送队列总有空余，避免了数据堵塞。这个函数在具体应用时有很多微妙之处，所以推荐编写网络协议的作者尽可能使用它。

许多发送例程利用这个函数几乎可以做所有的工作：

```
skb=sock_alloc_send_skb(sk,....)
if(skb==NULL)
```

```
        return -err;
    skb->sk=sk;
    skb_reserve(skb, headroom);
    skb_put(skb,len);
    memcpy(skb->data, data, len);
    protocol_do_something(skb);
```

上面大部分代码我们前面已经见过。其中最重要的一句是 skb->sk=sk。sock_alloc_send_skb（）负责把缓冲区送到套接字层。通过设置 skb->sk，告诉内核无论哪个例程对缓冲区进行 kfree_skb（）处理，都必须保证缓冲区已经成功地送到套接字层。因此一旦网络设备驱动程序发送一个缓冲区，并将之释放，我们就认为数据已经发送成功，这样我们就可以继续发送数据了。在源代码中我们看到 kfree_skb 操作一执行就会触发 sock_alloc_send_skb（）。

12.5　网络设备接口

Linux 网络设备接口的适用面很广，所有的 Linux 网络设备都遵循同样的接口，它提供了丰富的接口功能，但是每一个设备并不是完全都要用到这些功能。在前面我们讲过，网络部分使用的是面向对象的设计方法，每一种设备都被看成一种对象，因此在源代码中，我们就用一种具有一系列操作方法的数据结构来描述网络设备，这样做的目的就是为了在 C 语言中引入 C++的面向对象的思想。

文件 drivers/net/skeleton.c 包含了网络设备驱动程序的基本骨架。最好能找到一个最新版本的源代码放在手边，我们下边的内容将自始至终都将围绕着它展开。

12.5.1　基本结构

如图 12.14 是网络设备驱动程序的结构，从中我们可以看出，网络设备驱动程序的功能分为两部分：发送数据和接受数据。在发送数据时，设备驱动程序全权负责把来自协议层的

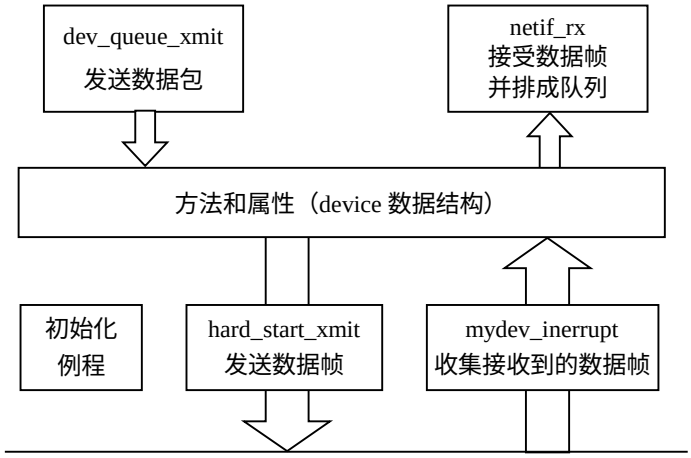

图 12.14　Linux 网络设备驱动程序的结构

网络缓冲区发送到物理介质，并且接收硬件产生的应答信号；在接收数据时，设备驱动程序接收来自网络介质上的数据帧，并把它转换成能被网络协议识别的网络缓冲区，然后把它传递给 netif_rx ( )函数。这个函数的功能是把数据帧传递到网络协议层进行进一步的处理。

每一种网络设备驱动程序都提供了一套相应的函数，它们负责对数据的传输过程进行各种控制（包括停止、开始等），它们也负责对数据进行封装。所有这些控制信息也都保存在设备驱动程序的数据结构中。

12.5.2　命名规则

所有的 Linux 网络设备都有唯一的名字，这个名字和文件系统所规定的设备的名字没有任何联系。事实上，网络设备并没有使用文件系统的表示方法。 传统上名字只表示设备类型而不代表生产厂商，如果同一类型的网络设备有多个，它们的名字就用从 0 开始的数字加以区别，例如，如果我们装了多块以太网卡，它们的名字就是：“ eth0 ”，“ eth1 ”，“ eth2 ”等。 这种命名机制非常重要，它使得用户在编写程序和配置系统时，可以使用“ 一个以太网卡 ”来统一地表示一块网卡，而不必考虑设备来自哪个厂商 ；而且，如果我们更换网络设备，也不需重新编译内核。

下面是一些常用网络设备的命名形式：

ethn	以太网控制器，包括 10mbit/s 和 100Mbit/s
trn	令环网设备
sln	SLIP （串行接口通信协议）设备和 AX.25 KISS 方式
pppn	PPP （点对点协议）设备，包括同步和异步方式
plipn	PLIP 单元; 数目和打印机端口对应。
tunln	IPIP encapsulated tunnels
nrn	NetROM 虚拟设备
isdnn	ISDN 接口 （isdn4linux）
dummyn	空设备 NULL devices
lo	回环设备

12.5.3　设备注册

每一个设备的建立都需要在设备数据结构类型中添加一个设备对象，并将它传递给 register_netdev(struct device *) 函数。这样就把你的设备数据结构和内核中的网络设备表联系起来。如果你要传递的数据结构正被内核使用，就不能释放它们，直到你卸载该设备，卸载设备用到 unregister_netdev(struct device *) 函数。这些函数调用通常在系统启动时或网络模块安装或卸载时执行。

内核不允许用同一个名字安装多个设备。因此，如果你的设备是可安装的模块，就应该利用 struct device *dev_get(const char *name)　函数来确保名字没有被使用。如果名字已经被使用，那么就必须另选一个，否则新的设备将安装失败。如果发现有设备冲突，就可以使用 unregister_netdev（）注销一个使用该名字的设备。

下面是一个典型的设备注册的源代码：

```
int register_my_device(void)
{
  int i=0;
  for(i=0;i<100;i++)
  {
    sprintf(mydevice.name,"mydev%d",i);
    if(dev_get(mydevice.name)==NULL)
    {
      if(register_netdev(&mydevice)!=0)
        return -EIO;
      return 0;
    }
  }
  printk("100 mydevs loaded. Unable to load more.<\\>n");
  return -ENFILE;
}
```

12.5.4　网络设备数据结构

网络设备数据结构 device，是网络驱动程序的最重要的部分，也是理解 Linux 网络接口的关键，它的源代码 保存在 include / linux / netdevice.h 中，这个结构比较庞大，在此不予列出，仅仅对主要的域给予解释。

所有的网络设备的信息和操作都保存在设备数据结构中。每注册一个网络设备，都需要提供数据结构中各个域的数据，这些域的含义下面将具体解释，其中，也包括对网络设备的设置。

1．名称

name 域指网络设备的名称，我们应该按上面讨论的命名方式为设备起名。该域也可以为空，这种情况下系统自动地分配一个 ethn 名字。在 Linux 2.0 版本以后，我们可以用 dev_make_name("eth") 函数来为设备命名。

2．总线接口参数

总线接口参数用来设置设备在设备地址空间的位置。

irq：指设备使用的中断请求号（IRQ），它通常在启动时或被初始化函数时设置。如果设备没有分配中断请求号，该域可以置 0。中断请求号也可以设置为变量，由系统自动搜索一个空闲的中断请求号分配给该设备。网络设备驱动程序通常使用一个全局整型变量 irq 表示中断号，因此用户可以使用“ insmod mydevice irq=5 ”这样的命令装载一个网络设备。最后，IRQ 域也可以利用 ifconfig 命令很方便地进行设置。

base_addr（基地址）：指设备占用的基本输入输出（I/O）地址空间。如果设备没有被分配 I/O 地址或该设备运在一个没有 I/O 空间概念的系统上，该域就置 0。当该地址由用户设置时，它通常用一个全局变量 io 来表示。I/O 接口地址也可以由 ifconfig 设置。

网络设备存在着两个硬件共享内存空间的情况，例如 ISA 总线和以太网卡共享内存空间。在网络设备的 device 数据结构中有 4 个相关的域。在共享内存时，rmem_start 和 rmem_end 域就被舍弃，并且置 0；mem_start 和 mem_end 两个域标识设备共享内存块的起始地址和结束地址。如果没有共享内存的情况，上面两个域就置 0。有一些设备允许用户设置内存地址，我们通常用一个全局变量 mem 表示。

dma：标志设备正在使用的 DMA 通道。Linux 允许 DMA（像中断一样）被系统自动探测。如果没有使用 DMA 通道或 DMA 通道没有设置，该域就置 0（最新的 PC 主版上， ISA 总线 DMA 通道 0 被硬件占用，它没有和内存的刷新联系起来）。如果由用户设置 DMA 通道，通常使用一个全局变量 dma 来表示。

我们应该认识到，上面提到的这些设备硬件信息都是从用户角度来对网络设备进行控制，它们和设备的内部函数功能是一样的。如果不注册它们，它们就有可能被重用，因此设备驱动程序必须分配并注册 I/O、DMA 和中断向量这些参数。这些操作和其他设备驱动程序都用到相同的内核函数，关于具体的操作过程请参阅设备驱动程序一章相关内容。

if_port：标识一些多功能网络设备的类型，例如 combo Ethernet boards。

3．协议层参数

为了使网络协议层能智能化地执行任务，网络设备驱动程序也需要协议层提供一些性能标志和变量，这些参数都保存在设备数据结构中。

mtu：指网络接口的最大负荷，也就是网络可以传输的最大的数据包尺寸，它不包括设备自身提供的低层数据头的大小，该值常被协议层（如 IP）使用，用来选择大小合适数据包进行发送。

family：指该设备支持的地址族。常用的地址族是 AF_INET，关于地址族的概念和具体解释，请参阅本章第三节套接字。 Linux 允许一个设备同时使用多个地址族，具体情况可以参考关于 BSD 网络 API 方面的书籍。

interface hardware type ：指设备所连接的物理介质的硬件接口类型，它的值来自物理介质类型表。支持 ARP 的物理介质，它们的接口类型被 ARP 使用（参看 RCF1700）；其他的接口类型是为其他物理层定义的。新的接口类型，只有当它对内核和 net-tools（注:net-tools 也是一段源代码，它随内核一起发布，用来对 Linux 网络进行调试）都是必

需时才会添加。包含像 ifconfig 这样的工具包可以对该域进行解码。该域的定义形式为：

/*该定义来自 RFC1700（RFC 即 Request For Comments 用户数据报协议）*/

ARPHRD_NETROMARPHRD_ETHER	10mbit/s 和 100mbit/s 以太网卡
ARPHRD_EETHER	实验用网卡（没有使用）
ARPHRD_AX25	AX.25 2 级接口
ARPHRD_PRONET	PROnet token ring（没有使用）
ARPHRD_CHAOS	ChaosNET（没有使用）
ARPHRD_IEE802	802.2 networks notably token ring
ARPHRD_ARCNET	ARCnet 接口
ARPHRD_DLCI	Frame Relay DLCI

由 Linux 定义：

ARPHRD_SLIP	Serial Line IP protocol
ARPHRD_CSLIP	SLIP with VJ header compression
ARPHRD_SLIP6	6bit encoded SLIP
ARPHRD_CSLIP6	6bit encoded header compressed SLIP
ARPHRD_ADAPT	SLIP interface in adaptive mode
ARPHRD_PPP	PPP interfaces (async and sync)
ARPHRD_TUNNEL	IPIP tunnels
ARPHRD_TUNNEL6	IPv6 over IP tunnels
ARPHRD_FRAD	Frame Relay Access Device
ARPHRD_SKIP	SKIP encryption tunnel
ARPHRD_LOOPBACK	Loopback device
ARPHRD_LOCALTLK	Localtalk apple networking device
ARPHRD_METRICOM	Metricom Radio Network

上面标注“没有使用”的接口，是因为它们虽然被定义了类型，但是目前还没有支持它们的 net-tools。Linux 内核为以太网和令环网提供了额外的支持例程。

- pa_addr：用来保持 IP 地址。
- pa_brdaddr：网络广播地址。
- pa_dstaddr：点对点连接中的目标地址。
- pa_mask：网络掩码。

上面所有域都被初始化为 0。

pa_alen：保存一个地址的长度，就 IP 地址而言，应该初始化为 4。

4．链接层变量

hard_header_len ：标识在网络缓冲区的头部，为硬件帧头准备的空间大小。这个值和将来添加的硬件帧头的字节数不一定相等。这样当 sk_buff 到达设备之前，就事先为硬件帧头准备了一块空间（scratch pad）。

在 1.2.x 系列的内核版本中，skb->data 指针指向整个缓冲区的开始，没有真正指向数据区，因此必须小心不能将“ scratch pad ”也发送出去。 这也暗示着 hard_header_len

的长度必须大于硬件帧头的长度（硬件帧头可以和数据相临）。在 1.3.x 以后的版本里，问题变得非常容易，因为 sk_buff 可以设置得足够大，不会出现空间不够的情况。至于这块空间 （scratch pad）的建立，要用到本章 12.4 节中的 skb_push( ) 函数。

物理介质的地址由分别保存在 dev_addr 和 broadcast 两个域中，我们用字符数组来保存物理地址。如果物理地址的长度比数组的长度小，那么物理地址在数组中就从左边开始保存，右边可以空余。addr_len 域用来存储物理地址的长度。因为许多介质没有物理地址，该域就置 0。还有一些其他类型的接口，物理地址必须由用户程序设置，对接口物理地址的设置可以用 ifconfig 工具。这种情况下，物理地址就不必进行初始化设置，但是我们从源代码中可以看出，如果在一个设备物理地址没有被设置，就不允许它进行传输数据。

5．接口标志

接口标志包含一些接口属性，其中一些是为了提高网络接口的兼容性而设的，因为这些标志并没有直接的用途。内核中用到的接口标志有：

- IFF_UP：接口已经激活。

在 Linux 中，IFF_RUNNIN 和 IFF_UP 标志基本上是成对出现，因为两者是相辅相成的。如果一个接口没有被标识 IFF_UP，它就不能够被删除。这和 BSD 不同，在 BSD 中，一个接口如果没有收到数据，就不会标识 IFF_UP。

- IFF_BROADCAST：设置设备广播地址有效。
- IFF_DEBUG：标识设备调试能力打开。目前并不使用。
- IFF_LOOPBACK：只有回环设备（lo）才使用该标志。
- IFF_POINTOPOINT：该设备是点对点链路设备（如 SLIP 或 PPP）。 通常点对点的连接没有子网掩码和广播地址，但是如果需要可以激活。
- IFF_RUNNING：见 IFF_UP。
- IFF_NOARP：接口不支持 ARP。这样的接口必须有一个静态的地址转换表，或者它不需要执行地址映射。NetROM 接口就是一个很好的例子。

6．数据包队列

该队列包含了等待由该网络设备发送的 sk_buff 数据包。发送到网络设备接口的数据包都由内核中协议层的代码排成队列。在每一个设备中，数据包按优先级顺序存放在 buffs [ ] 数组里。它们完全有内核代码控制，但是在启动时由设备自身进行初始化，初始化代码为：

```
int ct=0;
while(ct < DEV_NUMBUFFS)
{
    skb_queue_head_init(&dev->buffs[ct]);
    ct++;
}
```

其他域被初始化为 0。

网络设备通过设置 dev->tx_queue_len 来决定传输队列的长度。通常以太网的队列长度为 100，serial lines 为 10 。

12.5.5　支持函数

每个网络驱动程序都提供了一系列非常实用的函数，这些函数都是底层的基本的函数；每个设备还包含了一组标准的例程，协议层可以将这些例程当作设备链路层的部分而调用。关于这些函数和例程，下面我们详细介绍。

1．初始化设置（init)

init 函数在设备初始化和注册时被调用，它执行的是底层的确认和检查工作。在初始化程序里可以完成对硬件资源的配置。如果设备没有就绪或设备不能注册或其他任何原因而导致初始化工作不能正常进行，该函数就返回出错信息。一旦初始化函数返回出错信息，register_netdev（）也返回出错信息，这样该设备就不能安装。

2．打开（open）

open 这个函数在网络设备驱动程序里是网络设备被激活的时候被调用（即设备状态由down-->up）。所以实际上很多在 init 中的工作可以放到这里来做。比如资源的申请，硬件的激活。如果 dev->open 返回非零（error），则硬件的状态还是 down。 open 函数另一个作用是如果驱动程序作为一个模块被装入，则要防止模块卸载时设备处于打开状态。在 open 方法里要调用 MOD_INC_USE_COUNT 宏。

3．关闭（stop）

close 函数做和 open 函数相反的工作。可以释放某些资源以减少系统负担。close 是在设备状态由 up 转为 down 时被调用的。另外如果是作为模块装入的驱动程序，close 里应调用 MOD_DEC_USE_COUNT，减少设备被引用的次数，以使驱动程序可以被卸载。另外 close 方法必须返回成功(0==success)。

4．数据帧传输例程

所有的设备驱动程序都必须提供传输例程，如果一个设备不能传输，也就没有存在的必要性。事实上，设备的所谓的传输仅仅是释放传送给它的缓冲区，而真正实现传输功能是虚拟设备。

dev->hard_start_xmit（）：该函数的功能是将网络缓冲区，也就是 sk_buff 发送到硬件设备。如果设备不能接受缓冲区，它就会返回 1，并置 dev->tbusy 为非零值。这样缓冲区就排成队列，等待着 dev->tbusy 置零以后会再次发送。如果协议层决定释放被设备抛弃的缓冲区，那么缓冲区就不会再被送回设备；如果设备知道缓冲区短时间内不被能传送，例如设备严重堵塞，那么它就调用 dev_kfree_skb（）函数丢掉缓冲区，该函数返回零值标明缓冲区已经被处理完毕。

当缓冲区被传送到硬件以后，硬件应答信号标识传输已经完毕，驱动程序必须调用 dev_kfree_skb(skb，FREE_WRITE)函数释放缓冲区，一旦该调用结束，缓冲区就会很自然地消失，这样，驱动程序就不能再涉及缓冲区了。

该函数传送下来的 sk_buff 中的数据已经包含硬件需要的帧头。所以在发送方法里不需

要再填充硬件帧头，数据可以直接提交给硬件发送。sk_buff 是被锁住的（ocked）确保其他程序不会存取它。

5．硬件帧头

前面我们讲过，数据帧在传送之前先要排成对列，在加入队列之前，还要在每个数据帧的开始添加硬件帧头，这项工作对于数据传送非常必要。网络设备驱动程序提供了一个 dev->hard_header( ) 例程，来完成添加硬件帧头的工作。协议层在发送数据之前会在缓冲区的开始留下至少 dev->hard_header_len 长度字节的空闲空间。这样 dev->hard_header( ) 程序只要调用 skb_push（），然后正确填入硬件帧头就可以了。

调用这个例程需要给出和缓冲区相关的信息：设备指针、协议类型、指向源地址和目标地址（指硬件地址）的指针、数据包的长度。因为这个例程是在协议层发送函数触发之前被调用，所以一个非常重要参数值得我们注意：在这个例程中用的是 length 参数，而不是用缓冲区的长度做参数，因为调用 dev->hard_header( )时数据可能还没完全组织好。

源地址可以为“ NULL ”，这意味着“ 使用默认地址 ”；目标地址也可以为“ NULL ”，这意味着“ 目标未知 ”。如果目标地址“ 未知 ”，数据帧头的操作就不能完成，本来为硬件帧头预留的空间全部被其他信息占用,那么函数就返回填充硬件帧头空间的字节数的相反数(一定为负数）。当硬件帧头完全建立以后，函数返回所添加的数据帧头的字节数。

如果一个硬件帧头不能够完全建立，协议层必须试图解决地址问题，因为硬件地址对于数据的发送是必需的。一旦这种情况发生，dev->rebuild_header( )函数就会被调用，通常是利用 ARP（地址解析协议）来完成。如果硬件帧头还不能被解决，该函数就返回零，并且会再次尝试，协议层总是相信硬件帧头的解决是可能的。

6．数据接收

网络设备驱动程序没有关于接收的处理，当数据到来时，总是驱动程序通知系统。对一个典型的网络设备，当它收到数据后都会产生一个中断，中断处理程序调用 dev_alloc_skb（），申请一个大小合适的缓冲区（sk_buff），把从硬件传来的数据放入缓冲区。接着，设备驱动程序分析数据包的类型，把 skb->dev 设置为接收数据的设备类型 ，把 skb->protocol 设置为数据帧描述的协议类型，这样，数据帧就可以被发送到正确的协议层。硬件帧头指针保存在 skb->mac.raw 中，并且硬件帧头通过调用 skb_pull（）被去掉，因此网络协议就不涉及硬件的信息。最后还要设置 skb->pkt_type，标明链路层数据类型，设备驱动程序必须按以下类型设置 skb->pkt_type ：

PACKET_BROADCAST	链接层广播地址
PACKET_MULTICAST	链接层多路地址
PACKET_SELF	发给自己的数据帧
PACKET_OTHERHOST	发向另一个主机的数据帧（监听模式时会收到）

最后，设备驱动程序调用 netif_rx( )，把缓冲区向上传递给协议层。缓冲区首先排成一个队列，然后发出中断请求，中断请求响应后，缓冲区队列才被协议层进行处理。这种处理机制，延长了缓冲区等待处理的时间，但是减少了请求中断的次数，从而整体上提高了数据传输效率。一旦 netif_rx( )被调用，缓冲区就不在属设备驱动程序所有，它不能被修改，

而且设备驱动程序也不能再涉及它了。

在协议层，接收数据包的流程控制分两个层次：首先，netif_rx（）函数限制了从物理层到协议层的数据帧的数量。第二，每一个套接字都有一个队列，限制从协议层到套接字层的数据帧的数量。在传输方面，驱动程序的 dev->tx_queue_len 参数用来限制队列的长度。队列的长度通常是 100 帧，在进行大量数据传输的高速连接中，它足以容纳下所有等待传输的缓冲区，不会出现大量缓冲区阻塞的情况。在低速连接中，例如 slip 连接，队列的长度长设为 10 帧左右，因为传输 10 帧的数据就要花费数秒的时间排列数据。

本章的主要目的是介绍 Linux 操作系统网络部分的基本工作原理。因为 Linux 支持多种协议类型和多种网络设备，加上网络部分本身就比较复杂，所以本章所涉及的内容十分有限。为了便于说明，采用了以点带面的方法，着重介绍了网络部分的 4 个核心对象。

本章首先介绍了 Linux 网络部分源代码的面向对象设计思想，指出了 4 个核心对象：

- 网络协议；
- 套接字；
- 套接字缓冲区；
- 网络设备接口。

随后，用面向对象的分析方法，分别介绍了这 4 个核心对象的相关内容和它们之间的关系，这对于理解 Linux 网络的工作原理有很大帮助。

关于内容，本章尽可能详细具体，贴近实际应用，对于和实际应用有关系的地方都做了深入介绍。如果读者要做网络编程，请参考 12.2 网络协议，12.3 套接字，12.4 套接字缓冲区三节；如果要编写网络驱动程序，请参考 12.4 套接字缓冲区和 12.5 网络设备接口两节。