Linux 典藏大系

ChinaUnix

资深程序员十年经验总结，深入探讨Linux应用层和内核层的网络编程
详细讲解HTTP服务器、协议栈和防火墙三个典型案例的实际开发过程

# Linux 网络编程

宋敬彬
孙海滨 等编著

- 内容全面：全书涵盖Linux网络编程从基础到高级开发的方方面面
- 内容深入：重点讲解了技术性较强的Linux用户空间网络编程及内核网络编程
- 注重原理：对每个知识点都从原始概念和基本原理进行详细、透彻地分析
- 插图丰富：对比较复杂和难度较高的内容绘制了220余幅原理图进行讲解
- 代码经典：书中的示例代码大多是从实际项目总结而来，有很强的实用性
- 实践性强：贯穿450余个示例、70余个实例及3个案例进行讲解
- 案例典型：详细介绍了HTTP网络服务器、协议栈和防火墙的实现

清华大学出版社

# 第 16 章　Linux 内核中网络部分结构以及分布

第 5～15 章介绍了 Linux 环境下的用户层网络编程知识，基本上可以满足应用程序开发的需要。从本章开始，第 16 章和第 17 章将介绍 Linux 内核层网络架构，主要介绍如何基于 netfileter 框架在 Linux 的内核层挂接自己的网络数据处理函数，对内核层网络数据进行过滤。本章介绍内核层网络架构的基本知识，主要包括以下内容。

- 内核中网络相关代码的基本情况：内核层的网络代码分布情况，内核层的网络处理流程，内核层提供的用户处理网络数据的可插入点，内核层的数据结构及编程框架等处理内核层网络数据的基本技术；
- 简单介绍 netfilter 框架；
- 介绍 Iptables，如何使用 Iptables 控制 netfilter；
- 内核层的软中断报文队列处理方式；
- 中断处理下半部的要点和方式；
- 与一个 socket 有关的数据如何在内核层处理。

## 16.1　概　　述

Linux 网络协议栈的实现在内核代码中，了解 Linux 内核的网络部分代码有助于深刻理解网络编程的概念。Linux 内核层还提供了网络防火墙的框架 netfilter，基于 netfilter 框架编写网络过滤程序是 Linux 环境下内核层网络处理的常用方法。

### 16.1.1　代码目录分布

Linux 的内核源代码可以从 http://www.kernel.org/pub/Linux/kernel/网站上下载，本书使用 Linux-2.6.26.3 版本（可能不是最新版本，读者可以去下载最新的版本），其代码目录结构参见图 16.1。

- Documentation：这个目录下面没有内核的代码，有一套有用的内核文档。其中的文档质量良莠不齐，有很多内核文档的质量很优秀并且相当完整，例如文件系统；但是有的则完全没有文档，例如进程调度。在这个目录里不时可以发现有用的东西。

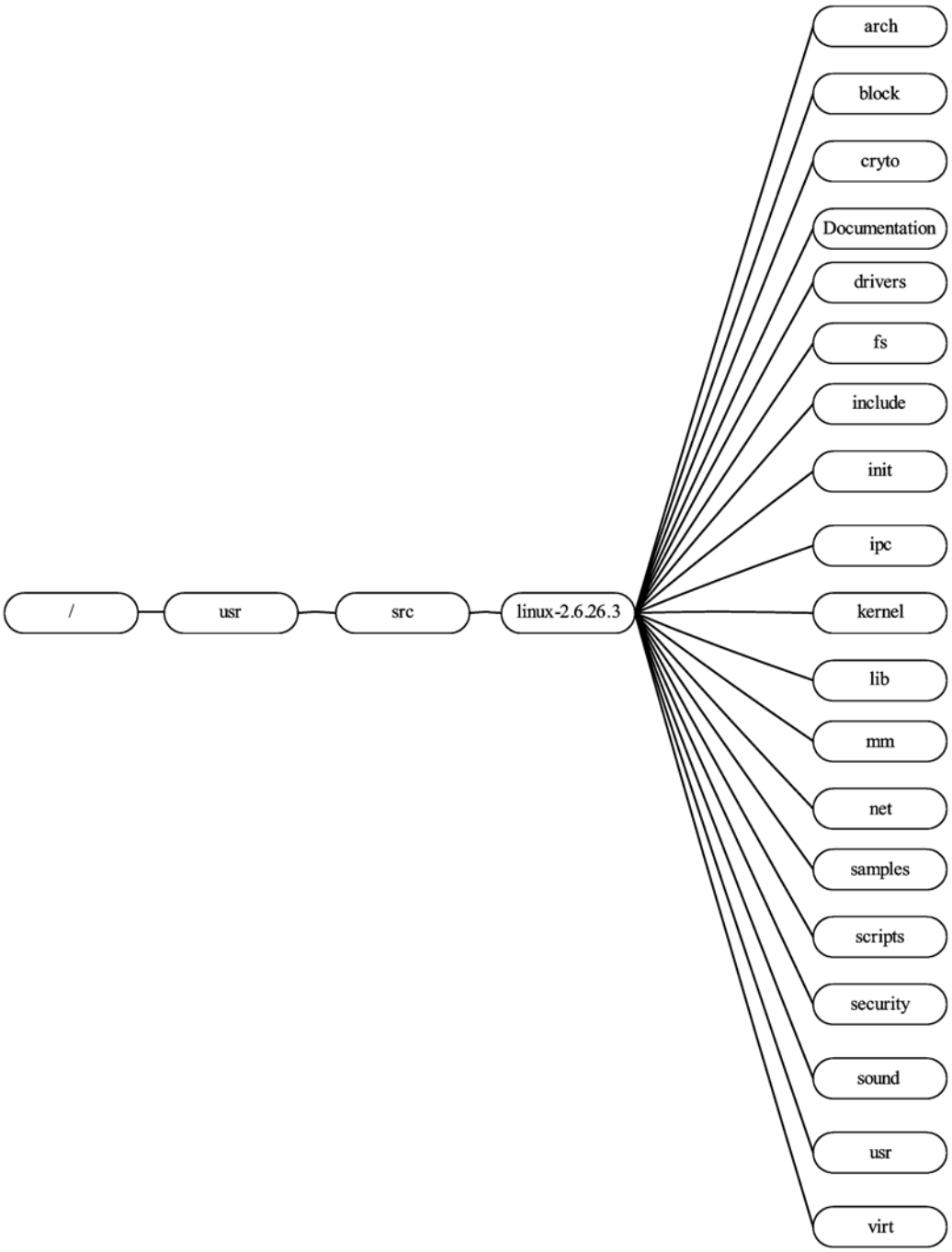



图 16.1　Linux 内核的源代码结构

- arch：此目录下的所有子目录的东西都是体系结构特有的代码。每个体系结构特有的目录下面至少包含 3 个子目录：kernel，不同体系结构内核特有的实现方式，如信号量、计时器、SMP 等；lib，不同体系结构下的高性能通用代码实现，如 memcpy

等；mm，不同体系结构特有的内存管理程序的实现。

- drivers：内核的驱动程序代码。此部分的代码占内核代码的大部分，包括显卡、网卡、PCI 等外围设备的驱动代码。
- fs：文件系统代码。包含 ext2、ext3 等本地文件系统，CD-ROM、isofs 等镜像系统，还有 NFS 等网络文件系统，以及 proc 等伪文件系统。
- include：此目录中包含了 Linux 内核中的大部分头（*.h）文件。
- init：内核初始化过程的代码。
- ipc：进程间通信代码。
- kernel：这部分是 Linux 内核中最重要的，包含了内核中平台无关的基本功能，主要包含进程创建、销毁和调度的代码。
- lib：此目录中主要包含内核中其他模块使用的通用函数和内核自解压的函数。
- mm：此目录中的代码实现了平台无关的内存管理代码。
- scripts：此目录下是内核配置时使用的脚本，当使用 make menuconfig 或者 make xconfig 命令时，会调用此部分代码。
- net：此目录中包含 Linux 内核的网络协议栈的代码。在子目录 netfilter 下为 netfilter 的实现代码，netfilter 构建了一个框架，允许在不重新编译内核的情况下，编写可加载内核，在指定的地方插入回调函数，以用户自己的方式处理网络数据。子目录 ipv4 和 ipv6 为 TCP/IP 协议栈的 IPv4 和 IPv6 的实现，主要包含了 TCP、UDP、IP 协议的代码，还有 ARP 协议、ICMP 协议、IGMP 协议、netfilter 的 TCP/IP 实现等代码实现，以及如 proc、ioctl 等控制相关的代码。本书的重点集中在这个目录中的相关技术。

图 16.2 所示是源代码组织的另一种表现形式，它映射到 Linux 代码的 3 个内核层。

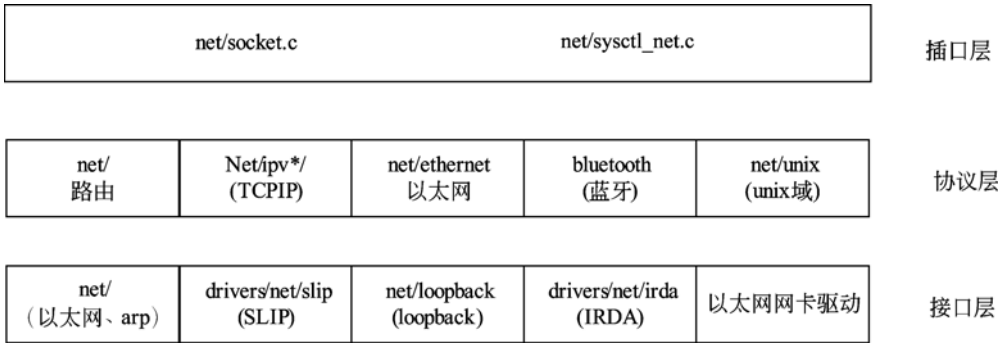



图 16.2　映射到 Linux 代码的 3 个内核层的源代码组织

如图 16.2 所示，以太网相关的为本书涉及的部分，其他（例如 SLIP、IRDA 等）部分是为了比较而用。

## 16.1.2　内核中网络部分流程简介

网络协议栈是由若干个层组成的，网络数据的流程主要是指在协议栈的各个层之间的传递。在第 7.2 节里介绍了 TCP 网络编程的流程，一个 TCP 服务器的流程按照建立 socket()，绑定（bind()）地址端口，侦听端口 listen()，接收连接 accept()，发送数据 send()，接收数据 recv()，关闭 socket()的顺序来进行。与此对应内核的处理过程也是按照此顺序进行的，网络数据在内核中的处理过程主要是在网卡和协议栈之间进行：从网卡接收数据，交给协议栈处理；协议栈将需要发送的数据通过网络发出去。

图 16.3 总结了各层间在网络输入输出时的层间调用关系。由图中可以看出，数据的流向主要有两种。应用层输出数据时，数据按照自上而下的顺序，依次通过插口层、协议层和接口层；当有数据到达的时候，自下而上依次通过接口层、协议层和插口层的方式，在内核层传递。

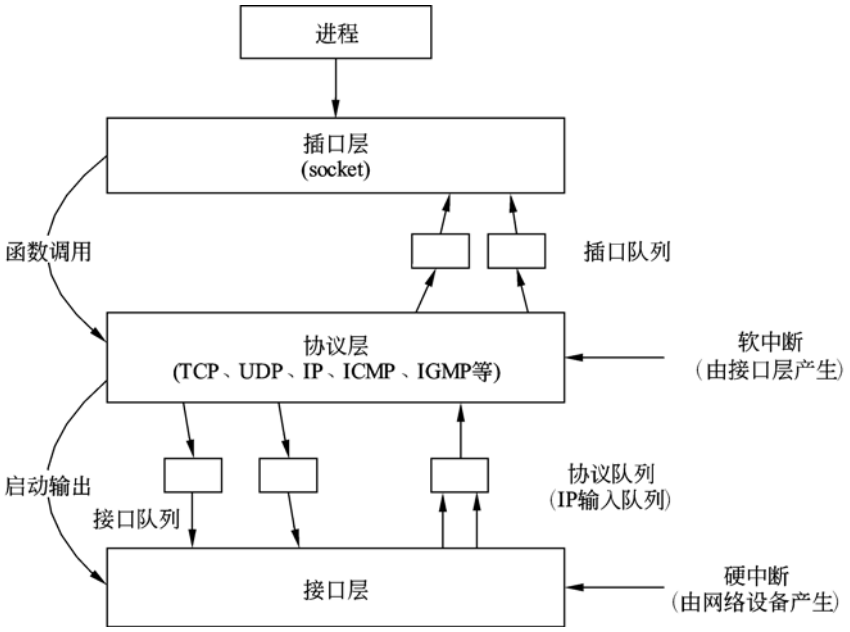



图 16.3　网络输入输出的各层间调用

应用层的 Socket 的初始化、绑定（bind）、销毁通过调用内核层的 socket()函数进行资源的申请、销毁。

发送数据的时候，将数据由插口层传递给协议层，协议层在 UDP 层添加 UDP 的首部、TCP 层添加 TCP 的首部、IP 层添加 IP 的首部，接口层的网卡则添加以太网相关的信息后，通过网卡的发送程序发送到网络上。

接收数据的过程是一个相反的过程，当有数据到来的时候，网卡的中断处理程序将数

据从以太网网卡的 FIFO 对列中接收到内核，传递给协议层，协议层在 IP 层剥离 IP 的首部、UDP 层剥离 UDP 的首部、TCP 层剥离 TCP 的首部后传递给插口层，插口层查询 socket 的标示后，将数据送给用户层匹配的 socket。

图 16.4 为 Linux 内核层的网络协议栈的架构视图。最上面是用户空间层，应用层的程序位于此处。最底部是物理设备，例如以太网网卡等，提供网络数据的连接、收发。中间是内核层，即网络协议栈子系统。流经网络栈内部的是 socket 缓冲区（由结构 sk_buffs 接连），它负责在源和汇点之间传递报文数据。在 16.1.4 节中会对 sk_buff 的结构进行介绍。

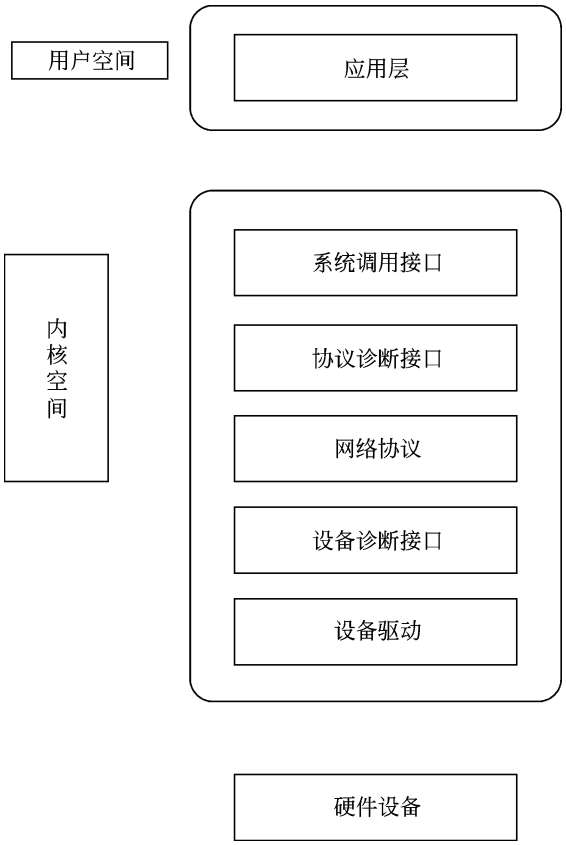



图 16.4　Linux 内核网络协议栈架构

顶部（参见图 16.4）是系统调用接口，它为用户空间的应用程序提供了一种访问内核网络子系统的接口。位于其下面的是一个协议无关层，它提供了一种通用方法来使用底层传输层协议。然后是实际协议，在 Linux 中包括内嵌的协议 TCP、UDP，当然还有 IP。然后是另外一个网络设备协议无关层，提供了与各个设备驱动程序通信的通用接口，最下面是设备驱动程序本身。

### 16.1.3　系统提供修改网络流程点

16.1.2 节介绍了网络数据在内核中的流程，网络中的数据在通常情况下按照 16.1.2 节所述的流程传递。Linux 内核中还提供了一种灵活修改网络数据的机制，用户可以利用这种机制获得和修改内核层的网络数据和属性设置。

如图 16.5 所示，白色的框为网络数据的流向，协议栈按照正常的方式进行处理和传递。Linux 内核在网络数据经过的多个地点设置了检查点，当到达检查点的时候，会检查这些点上是否有用户设置的处理方法，按照用户的处理规则对网络数据进行处理后，数据会再次按照正常的网络流程传递。

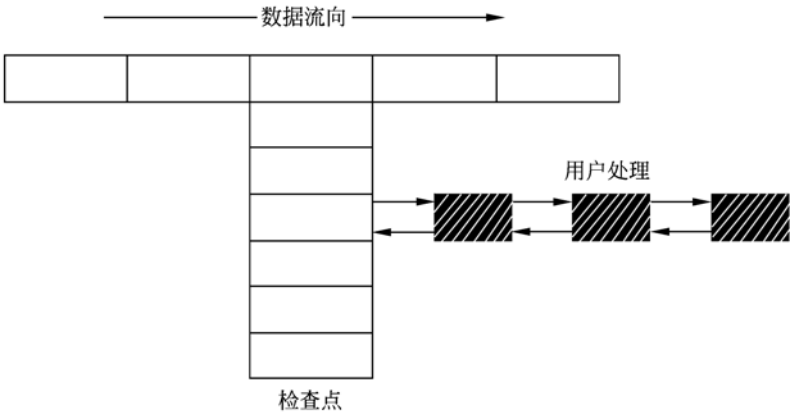



图 16.5　网络数据检查点示意图

### 16.1.4　sk_buff 结构

内核层和用户层在网络方面的差别很大，在内核的网络层中 sk_buff 结构占有重要的地位，几乎所有的处理均与此结构有关系。网络协议栈是一个层次架构的软件结构，层与层之间通过预订的接口传递报文。网络报文中包含了在协议各层使用到的各种信息。由于网络报文之间的大小不是固定的，因此采用合适的数据结构来存储这些网络报文就显得非常重要。

#### 1．结构sk_buff的原型

在 Linux 的 2.6 版本的内核中，采用结构 sk_buff 来存储这些数据。在这个结构中，既有指向网络报文的指针，同时也有描述网络报文的变量。sk_buff 数据结构的代码如下所示。

```
struct sk_buff
{
struct sk_buff        *next;
```

```
struct sk_buff        *prev;
struct sock          *sk;
ktime_t                 tstamp;
struct net_device       *dev;
union
{
struct  dst_entry    *dst;
struct  rtable      *rtable;
};
struct  sec_path     *sp;
char               cb[48];
unsigned int         len,
data_len;
__u16             mac_len,
hdr_len;
union
{
__wsum          csum;
struct
{
__u16           csum_start;
__u16           csum_offset;
};
};
__u32               priority;
__u8                 local_df:1,
              cloned:1,
ip_summed:2,
nohdr:1,
nfctinfo:3;
__u8                pkt_type:3,
fclone:2,
ipvs_property:1,
peeked:1,
nf_trace:1;
__be16                  protocol;
void                    (*destructor)(struct sk_buff *skb);
struct nf_conntrack     *nfct;
struct sk_buff          *nfct_reasm;
struct nf_bridge_info   *nf_bridge;
int                     iif;
__u16                   queue_mapping;
__u16                   tc_index;
__u16                   tc_verd;
__u8                    ndisc_nodetype:2;
dma_cookie_t            dma_cookie;
__u32                   secmark;
__u32                   mark;
sk_buff_data_t          transport_header;
sk_buff_data_t          network_header;
sk_buff_data_t          mac_header;
sk_buff_data_t          tail;
sk_buff_data_t          end;
unsigned char           *head,
                        *data;
unsigned int            truesize;
atomic_t                users;
};
```

sk_buff 主要成员的含义如下：

- next：sk_buff 链表中的下一个缓冲区。
- prev：sk_buff 链表中的前一个缓冲区。以上两个变量将 sk_buff 链接到一个双向链表中。
- sk：本网络报文所属的 sock 结构，此值仅在本机发出的报文中有效，从网络收到的报文此值为空。
- tstamp：报文收到的时间戳。
- dev：收到此报文的网络设备。
- transport_header：传输层头部。
- network_header：网络层头部。
- mac_header：链接层头部。
- cb：用于控制缓冲区。每个层都可以使用此指针，将私有的数据放置于此。
- len：有效数据长度。
- data_len：数据长度。
- mac_len：连接层头部长度，对于以太网，指 MAC 地址所用的长度，为 6。
- hdr_len：skb 的可写头部长度。
- csum：校验和（包含开始和偏移）。
- csum_start：当开始计算校验和时从 skb->head 的偏移。
- csum_offset：从 csum_start 开始的偏移。
- local_df：允许本地分片。
- pkt_type：包的类别。
- priority：包队列的优先级。
- truesize：报文缓冲区的大小。
- head：报文缓冲区的头。
- data：数据的头指针。
- tail：数据的尾指针。
- end：报文缓冲区的尾部。

### 2．sk_buff的含义

图 16.6 是结构 sk_buff 的框图，其中的 tail、end、head 和 data 是对网络报文部分的描述。

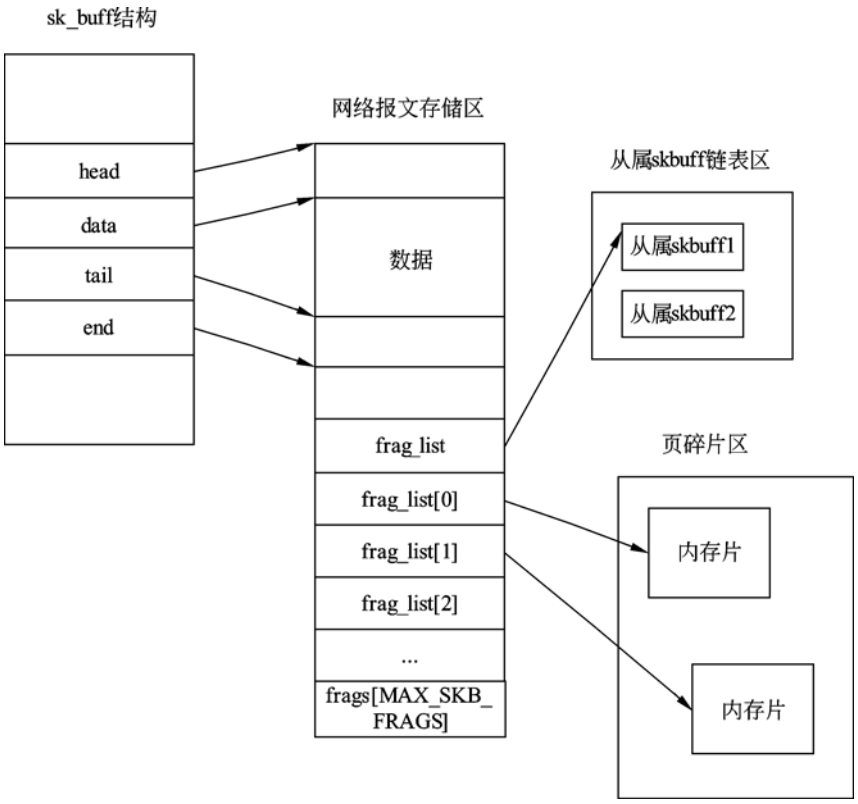



图 16.6　sk_buff 的数据结构

网络报文存储空间是在应用层发送网络数据或者网络设备收到网络数据时动态分配的，分配成功之后，将接收或者发送的网络数据填充到这个存储空间中去。将网络数据填充到存储空间时，在存储空间的头部预留了一定数量的空隙，然后从此偏移量开始将网络报文复制到存储空间中。

结构 sk_buff 以 sk_buff_head 构成一个环状的链，如图 16.7 所示，next 变量指向下一个 sk_buff 结构，prev 变量指向前一个 sk_buff 结构。内核程序通过访问其中的各个单元来遍历整个协议栈中的网络数据。

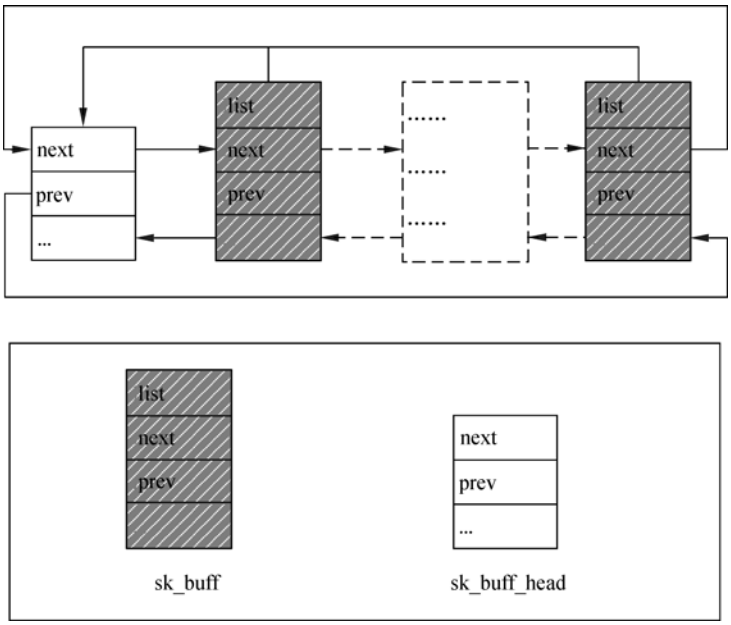



图 16.7　sk_buff_head 与 sk_buff 生成的链表

## 16.1.5　网络协议数据结构 inet_protosw

第 5 章中对 TCP/IP 的网络协议族进行了介绍（IP、TCP、UDP 等）。其中协议 TCP、UDP、RAW 在文件 Linux-2.6.26.3/net/ipv4/af_inet.c 中一个名为 inet_init()的函数中进行了初始化（因为 TCP 和 UDP 都是 inet 簇协议的一部分）。inet_init()函数使用 proto_register()函数来注册每个内嵌协议。

通过 Linux/net/ipv4/目录中 tcp.c 和 raw.c 文件中的 proto 接口，可以了解各个协议是如何标识自己的。这些协议接口每个都按照类型和协议映射到 inetsw_array，该数组将内嵌协议与操作映射到一起。inetsw_array 结构及其关系如图 16.8 所示。最初，会调用 inet_init ()中的 inet_register_protosw() 将这个数组中的每个协议都初始化为 inetsw。函数 inet_init() 也会对各个 inet 模块进行初始化，例如 ARP、ICMP 和 IP 模块，以及 TCP 和 UDP 模块。

在图 16.8 中，proto 结构定义了传输特有的方法，而 proto_ops 结构则定义了通用的 socket()方法。可以通过调用 inet_register_protosw()将其他协议加入到 inetsw 协议中。例如，SCTP 就是通过调用 Linux/net/sctp/protocol.c 中的 sctp_init 加入其中的。

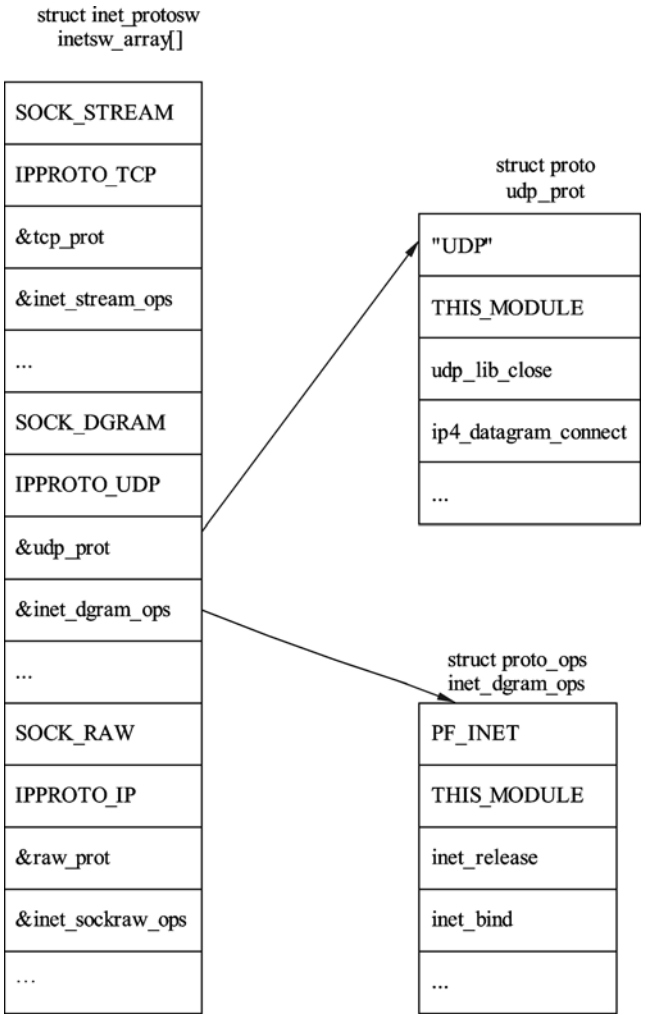



图 16.8　inet_protosw 数组结构

# 16.2　软中断 CPU 报文队列及其处理

软中断是 Linux 内核中的一种概念，它利用硬件中的中断概念，用软件方式对此进行模拟，实现相似的执行效果。

## 16.2.1　Linux 内核网络协议层的层间传递手段——软中断

网络协议栈是分层实现的，如何实现高效的网络数据是协议栈设计的核心问题之一。

### 1．Linux内核中软中断的机制

在 Linux 内核中是采用软中断的方式实现的，软中断机制的实现原理如图 16.9 所示。

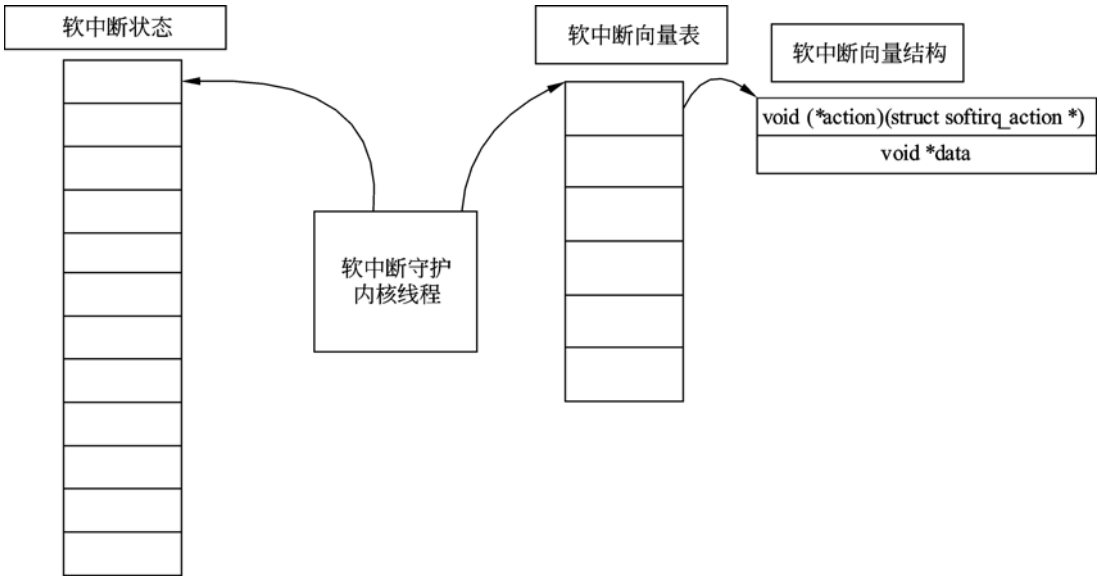



图 16.9　软中断架构示意图

软中断机制的构成核心元素包括软中断状态、软中断向量表和软中断守护内核线程。

- 软中断状态：即是否有触发的软中断未处理。
- 软中断向量表：包含两个成员变量，一个是处理此软中断的回调函数，另一个是处理时所需的参数。
- 软中断守护内核线程：内核建立一个内核线程 ksoftirqd 来轮询软中断状态，调用软中断向量表中的软中断回调函数处理中断。

Linux 内核中的软中断的工作框架模拟了实际的硬中断处理过程。当某一软中断事件发生后，首先调用 raise_softirq()函数设置对应的中断标记位，触发中断事务。然后会检测中断状态寄存器的状态，如果 ksoftirqd 通过查询发现某一软中断事务发生之后，那么通过软中断向量表调用软中断服务程序 action。

软中断的过程与硬中断是十分类似的，二者的唯一不同之处是从中断向量到中断服务程序的映射过程。在 CPU 的硬件中断发生之后，CPU 的具体的服务程序通过中断向量值进行映射，这个过程是硬件自动完成的。但是软中断的中断映射不是自动完成的，需要中断服务守护线程去实现这一过程，这也就是软件模拟的中断。

### 2．Linux内核中软中断的使用方法

在 Linux 系统中最多可以同时注册 32 个软中断，目前系统使用了 6 个软中断，它们是定时器处理、SCSI 处理、网络收发处理以及 tasklet 机制，这里的 tasklet 机制就是用来实现下半部的，描述软中断的核心数据结构为中断向量表，其定义如下：

```
struct softirq_action
{
    void (*action)(struct softirq_action *);
    void *data;
};
```

- action 为软中断服务程序。
- data 为服务程序输入参数。

软中断守护程序是软中断机制实现的核心，它的实现过程比较简单。通过查询软中断的状态来判断是否发生事件，当发生事件就会映射软中断向量表，调用执行注册的 action()函数就可以了。从这一点分析可以看出，软中断的服务程序是 daemon。在 Linux 中软中断 daemon 线程函数为 do_softirq()。

触发软中断事务通过 raise_softirq()来实现，该函数就是在中断关闭的情况下设置软中断状态位，然后判断如果不再中断上下文，那么直接唤醒守护 daemon。

常用的软中断函数列表如下：

- open_softirq()：它注册一个软中断，将软中断的服务程序注册到系统的软中断向量表。
- raise_softirq()：设置软中断状态映射表，触发软中断事务响应。

Linux 软中断的处理框架也采用了上半部和下半部的处理方式。软中断的上半部处理紧急的、需要立即处理的、关键性的处理动作，例如网卡驱动的接收动作，当有中断到达的时候，先查询网卡的中断寄存器，判断为何种方式的中断，清空中断寄存器后，复制数据，然后设置软中断的状态，触发软中断。软中断的下半部进行数据处理，而下半部则相对来说并不是非常紧急的，通常还是比较耗时的，因此由系统自行安排运行时机，不在中断服务上下文中执行。

### 16.2.2　网络收发处理软中断的实现机制

网络收发的处理通过软中断进行处理，考虑到优先级问题，分别占用了向量表中的 2 号和 3 号软中断来分别处理接收和发送。网络协议栈的软中断机制的实现原理如图 16.10 所示。

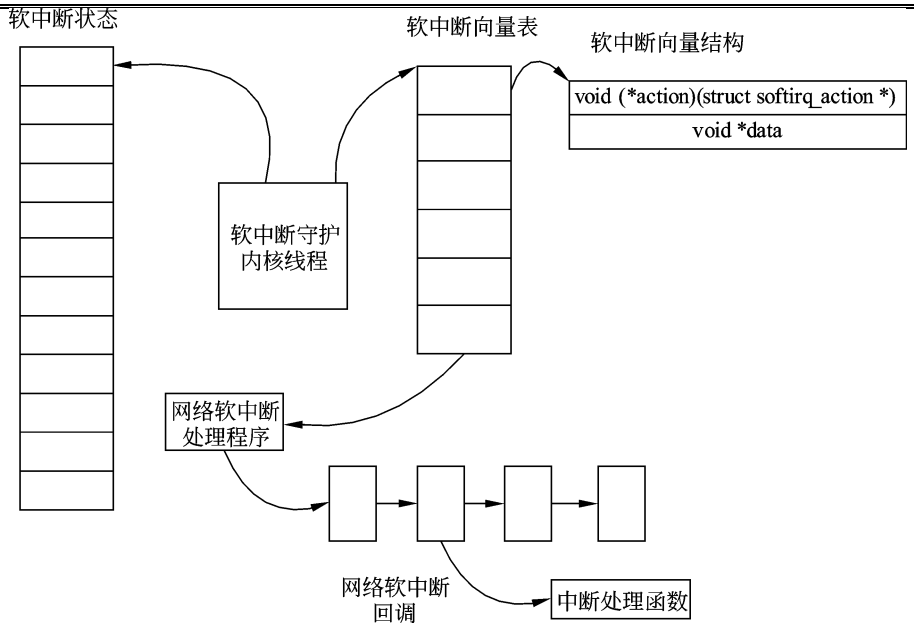



图 16.10　协议栈中的软中断架构示意图

当网络的软中断事件发生之后，执行 net_rx_action()或者 net_tx_action()的软中断服务程序，该服务程序会扫描一个网络中断状态的值，查找中断源，执行具体服务程序。在这里举一个例子加以说明：

当网络上有数据时，发生了硬件中断，硬件中断服务程序会接收网络数据，设置中断状态，并将网络数据挂接到链表上，进行中断调度，这一步可以通过 net_schedule()函数完成。硬件中断服务程序最后退出并且 CPU 开始调度软中断，软中断 daemon 会发现网络软中断发生了事件，其会执行网络中断对应的服务程序，即进入网络协议栈处理程序。

## 16.3　socket 数据如何在内核中接收和发送

socket 数据在内核中的流程主要包含初始化、销毁、接收和发送网络数据。其过程涉及网卡驱动、网络协议栈和应用层的接口函数。

### 16.3.1　socket()的初始化

创建 socket()需要传递 family、type、protocol 这 3 个参数。创建 socket()其实就是创建一个 socket 实例，然后创建一个文件描述符结构。创建套接字文件描述符会互相建立一些关联，即建立互相连接的指针，并且初始化这些对文件的写读操作映射到 socket 的 read()、write()函数上来。

在初始化套接字的时候，同时初始化 socket 的操作函数（proto_ops 结构)。如果传入的 type 参数是 STREAM 类型，那么就初始化为 SOCKET->ops 为 inet_stream_ops。

参数 inet_stream_ops 是一个结构体，包含了 stream 类型的 socket 操作的一些入口函数。在这些函数里主要做的是对 socket 进行相关的操作。

创建 socket 的同时还创建一个 sock 结构的数据空间。初始化 sock，初始化过程主要做的事情是初始化 3 个队列：receive_queue（接收到的数据包 sk_buff 链表队列)，send_queue（需要发送数据包的 sk_buff 链表队列），backlog_queue（主要用于 tcp 中三次握手成功的那些数据包)。根据 family、type 参数，初始化 sock 的操作，例如对于 family 为 inet 类型的，type 为 stream 类型的，sock->proto 初始化为 tcp_prot，其中包括 stream 类型的协议 sock 操作对应的入口函数。

应用层关闭 socket 时，内核需要释放所关闭 socket 申请的资源。

### 16.3.2　接收网络数据 recv()

网络数据接收依次经过网卡驱动和协议栈程序，以 DM9000A 网卡为例进行介绍接收数据的过程。

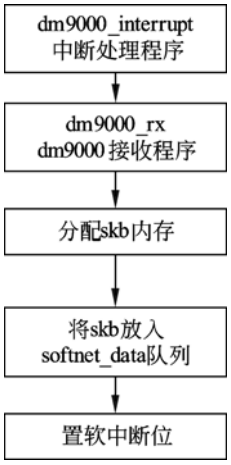
如图 16.11 所示，网卡在一个数据包到来时，会产生一个硬中断，网络驱动程序会执行中断处理过程：首先申请一个 skb 结构及 pkt_len+5 大小的内存用于保存数据，然后便将接收到的数据从网卡复制到这个 skb 的数据部分中。当数据从网卡中成功接收后，调用 netif_rx(skb)进一步处理数据，将 skb 加入到相应的 input_pkt_queue 队列中，并调用 netif_rx_schedule()，会产生一个软中断来执行网络协议栈的例程。这样，中断的上半部已完成，以下的工作则交由中断的下半部来实现。

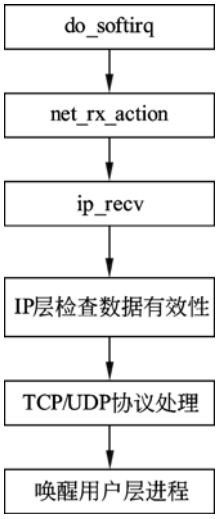
如图 16.12 所示，下半部的内核守护线程 do_softirq()，将执行 net_rx_action()，对数据进行处理。IP 层输入处理程序轮询处理输入队列中的每个 IP 数据，在整个队列处理完毕后返回。IP 层验证 IP 首部的校验和，处理 IP 选项，验证 IP 主机地址和正确性等，并调用相应协议（TCP 或者 UDP 等）处理程序。接收的进程在网络协议栈处理完毕后会收到唤醒的信号，并收到发送来的网络数据。

### 16.3.3　发送网络数据 send()

Linux 对网络数据的发送过程的处理与接收过程相反。在一端对 socket 进行 write()的过程中，首先会把要 write 的字符串缓冲区整理成 msghdr 的数据结构形式，然后调用 sock_sendmsg()把 msghdr 的数据传送至 inet 层。

对于 msghdr 结构中数据区中的每个数据包，创建 sk_buff 结构，填充数据，挂至发送队列。一层层往下层协议传递，如图 16.13 所示。以下每层协议不再对数据进行复制，而是对 sk_buff 结构进行操作。



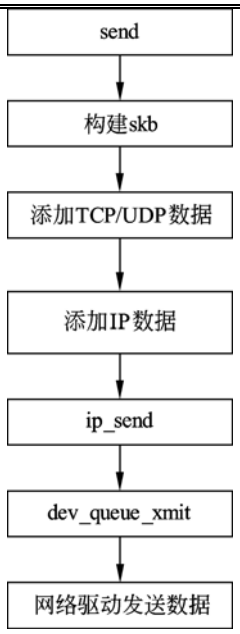




图 16.11　网卡接收数据流程　　图 16.12　协议栈处理数据流程　　图 16.13　协议栈处理数据流程

最后调用网络驱动，发送数据，在网络发送成功后要产生中断，将发送结果反馈回应用层，此过程与接收网络数据的过程类似。

## 16.4　小　　结

本章介绍了 Linux 内核代码的架构，特别是网络相关的部分，并对结构 sk_buff 进行了详细的分析，简单分析了网络数据的流程。介绍了 Linux 的软中断方式，对网络协议栈中使用的软中断处理报文队列的方式进行了简单介绍。对插口层的网络数据发送接收的流程进行了分析。