LINUX-1.2.13

内核网络栈实现

源代码分析

中国科学技术大学

（安徽，合肥）

近代物理系

快电子实验室

曹桂平（著）

This page is intentionally left blank.

序言

使用老的 LINUX 版本进行内核代码分析在很多人看来是一种“避实就虚”的“卑鄙”手段。因为老的 LINUX 版本代码较为简单，分析起来基本不费“吹灰之力”，所以为很多“高手”所不齿。而对于很多新手而言，学习老代码似乎觉得有点让人看不起，或者自己有点看不起自己，最“酷”的莫过于捧着最新版本的内核源代码去“啃”；再者，有些新手对于学习老代码的作用有些怀疑，毕竟这些代码已成为历史，现在运行的系统代码较之那些之前的老代码简直是“面目全非”，甚至基本找不到老代码的影子，此时会对学习这些老代码的必要性产生怀疑，本人之前也是抱此怀疑。于是本人就去找最新的版本去看，去研究，比如国内比较有名的关于 LINUX 内核源代码分析的书《LINUX 内核源代码情景分析》上下册。这本书分析的是 LINUX2.4 内核早期版本代码，相比较现在的 2.6 版本而言，这也算是老版本了。读过此书的人恐怕都深有体会，这个体会不是说读完了（真是精神可嘉）这本书感觉受益匪浅，而是感觉相当的痛苦，坦白的说，如果没有对操作系统有深刻的理解并且之前接触过操作系统编程，恐怕只能获得一个非常泛泛的理解。因为其传授的知识太多，一方面让我们不知所措，另一方面也感觉自己“内功”过于薄弱，无法承受这种“上乘武功“。那么之后（最近几年）出版的由同济大学赵炯博士所写的《LINUX-0.11 完全注释》可以说对于很多 LINUX 内核爱好者而言帮助很大。LINUX-0.11 内核版本在如今真可算得上是”古董“了。不过我想研读过这本书的读者肯定收获颇丰。而且之后当我们试图去分析较新版本的内核时，或者再去研读其它较新版本的有关内核分析的文章时，有种似曾相识之感，而且理解上也变得容易。原因很简单，中国有句古语，”麻雀虽小，五脏俱全“，LINUX-0.11 内核版本虽然很小，但是是一个可以运行的操作系统版本，我们研读这本书了解了组成一个可运行操作系统需要完成的工作，通过代码实例可以更深刻的理解操作系统的各种概念。对于很多读者而言包括本人在内，之前对于操作系统”心存敬畏“，可是现在在很多人面前我都敢说，”哦，操作系统实现并不复杂，只是比较麻烦…“。早期代码的分析可以让我们把握问题的实质，从本文分析的主题网络部分代码来看，这点尤为突出。从 LINUX-0.96C 版本开始就包含网络代码，本人从 0.96C 版本开始，对网络部分代码进行过系统分析，一直到 1.3.0 版本，可以将这之前的版本都归为早期版本，这些版本有一个共同特点，即网络部分所有代码都在一个文件夹中，每种协议的实现都只是一个文件与之对应（接近 1.3.0 版本的代码，某些协议开始有多个文件对应），而此之后的版本（从 1.3.0 开始，包括 1.3.0）开始进行细分，这一方面代码更加完善所以也就更加庞大，另一方面随着代码的逐渐成熟，层次关系更加明确。如现在对于内核数据包的封装结构 sk_buff，对于早期代码而言并没有专门的处理文件，而现在对于此结构就有一个专门的操作函数集，集中放置在 sk_buff.c 文件中。

研究早期代码一个最大的优势就是可以直接接触问题的本质。因为早期代码更注重实际功能的实现，而非辅助功能的实现，辅助功能的代码很少或者基本没有，例如早期代码并没有专门的函数集去处理 sk_buff 结构的操作问题，一般在需要分配或操作该结构时直接编码操作，而现在这些”直接的编码“被分离出来，成为专门的操作函数集。从内核的演变来看，这是必要的，但从代码的分析来看，变得越来越复杂，如果对内核代码组织不熟悉（很多新手即如此），甚至不知道一个被调用函数在哪儿，而使用查找工具来回翻看，不一会就会产生疲劳，而且深感内核的庞大复杂，从而削减学习的积极性。而分析早期代码一方面可以更直接的抓住问题实质，另一方面可以将我们的注意力集中于对问题本质的理解上。

本文选择 LINUX-1.2.13 内核所包含的网络部分代码分析（注意网络部分代码与内核代码的演变是分离的，如 LINUX1.2.8 网络代码与 1.2.13 是一样的，而内核显然是有差异的）。LINUX-1.2.13 网络部分所有实现代码仍然是集中在一个文件夹中（net 文件夹），在此之后（1.3.0 内核版本）的网络部分代码在 net 文件夹下进行了细分，除此之外，对于某些协议（如

TCP，IP）协议的实现也从之前的一个单一文件分立为几个文件以加强层次和功能关系。为了消除结构上不必要的复杂性并保持功能实现上的完整性，选择 LINUX1.2.13 内核版本网络代码进行分析是比较适宜的。这个版本代码坦白的说在某些细节的处理上还存在很多问题，但基本功能及网络栈的层次性已经十分清晰，而且与其之后版本结构承接性较好，即后续版本基本完全保存了此版本代码的结构组织，主要是对功能实现上的补充和完善，当然在这过程中某些结构会增加或删除某些字段，但基本实现的功能没有发生改变，所以可以作为其后版本分析的一个较好的出发点。会有极少数结构名称发生改变如到 2.4 版本将 device 结构名改为 net_device。不过对于此类更改不会对后续版本分析造成实质性影响。

选择早期版本分析一方面减小分析的难度，从而同时减小了读者理解上的难度。既然是一个早期的版本，也就是说此处的工作只是为读者建立好一个起跑的起点，如果读者只是为方便理解某些用户接口函数的功能，那么此处的工作已是足够帮助你更好的进行理解了，如果读者有志成为网络黑客，那么有必要在此基础上，向更新版本更复杂的代码发出挑战。在这过程中你会发觉对早期版本代码深刻理解的重要性，你会发现这个挑战的过程会比你之前想象的更为顺利，因为从早期代码的学习中你已经掌握了问题的实质。

各种版本Linux源码（包括本书分析所用的Linux-1.2.13 内核版本）可到如下网站：http://www.kernel.org/ 进行下载。

鉴于作者能力有限，不可能对源代码中每一个问题都理解到位，如果发现理解有误之处，还请广大读者指正，可通过邮件通知我，你我一起进行探讨：ingdxdy@gmail.com。

曹桂平

2009．02．20

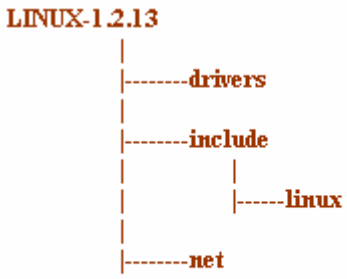
目　　录

引言

如果你主观上感觉去做某件事情会很复杂，很费力，大多数时候是因为你对这件事了解的不够深刻。当你去不断的搜集材料，不断对它进行分析，有一天你会豁然开朗，原来事情确实如此简单。大多数应用程序程序员对操作系统内核编程心存敬畏，认为内核编程是一件非常复杂的事，但是对于大多数分析过 LINUX“古董“级内核版本的 LINUX 爱好者而言，即便其不懂什么 VC++，JAVA，但其对操作系统内核编程却信心十足。LINUX 的开源性使得越来越多的程序员可以觊觎编写内核的那种满足感和成就感。那么我们将范围缩小，编一个网卡驱动程序怎么样，很难吗，我想很多程序员会回答是，因为这是属于内核编程的一部分，无论其作为内核的固定组成部分，还是以一个内核模块的方式加载。而且网卡驱动程序相对而言应该是 LINUX 驱动程序编写中比较复杂的一类。但是如果此时有人跳出来说，编写网卡驱动程序就那么回事，很简单，你是不是觉得他有点造作？但是如果这话由你自己说出，你是不是也是这样认为。对 LINUX 内核网络栈代码进行完全的分析后，回过头来再看网卡驱动程序的编写，简直是不值一提。在你脑海中，你会“画出”一条网络数据包的”旅行“路线，路线上的各个”关卡“以及每一个”关卡“对数据包进行何种处理你一清二楚。而此时网卡驱动程序代码只不过是这些”关卡“中较不起眼的一个。因为它“基本没做什么工作，就写几个寄存器而已”。

对于 LINUX-1.2.13 网络栈代码实现本文采用两种分析方式，第一种是按部就班的逐文件逐函数进行分析，此种分析方式较少涉及各种协议之间的关联性；第二种是从结构上和层次上进行分析，着重阐述数据包接收和发送通道，分析数据包的传输路线上各处理函数，此时将不针对某个函数作具体分析，只是简单交待功能后，继续关注其上下的传输。在第一种分析的基础上辅以第二种分析使问题的本质可以更好被理解和掌握。分析网络栈实现不可不涉及网络协议，如果泛泛而谈，不但浪费篇幅，也不起作用，但是在分析某种协议的代码之前，又不可不对该协议有所了解，本文尽量介绍这些协议的主要方面，但无法对这些协议进行完全介绍，只是在分析某段代码所实现功能时，为帮助理解，在此之前会给出协议的相关方面介绍，如在分析 TCP 协议的拥塞处理代码时，在这之前会对 TCP 的拥塞处理进行介绍。

LINUX-1.2.13 内核网络栈实现代码分布于四个文件夹中，以 LINUX-1.2.13 作为内核源代码根目录，则此四个目录的位置如下所示：



LINUX-1.2.13 内核网络栈代码文件组织形式

本书共分为四章，第一章着重介绍网络协议涉及的头文件，即对 include/linux 子目录下相关文件进行分析；第二章分析网络协议具体实现代码，这是本书的重点，这部分代码都集中在 net 子目录下；第三章介绍网络设备驱动层相关内容，包括网络设备的初始化过程以及驱动程序结构，这部分代码分布在 drivers 子目录下；第四章介绍系统网络栈的初始化流程以及理清网络数据包的上下传送通道，从而再次从整体实现上进行把握。

第一章 网络协议头文件分析

本章内容集中分析网络协议头文件，这些头文件集中在 include/linux 子目录下。表 1.1 以首字母序列出了该子目录下涉及网络协议的所有头文件名称及其功能。（注意 include/linux 子目录下还包括内核其它头文件，此处抽出了所有涉及网络部分的头文件,另外在分析网络实现代码时可能需要一些其它头文件中内容，这在相关处会给出相关代码。）

表 1.1 网络相关头文件列表（以首字母为序）

头文件名	最后修改日期	说明
etherdevice.h	1994-04-18 16:38	以太网协议相关函数声明
icmp.h	1993-12-01 20:44	ICMP 协议结构定议
if.h	1994-12-01 03:53	接口相关结构定议
if_arp.h	1995-02-05 20:27	ARP 协议结构定议
if_ether.h	1995-02-02 02:03	以太网首部及标志位定义
if_plip.h	1995-02-13 03:11	并行线网络协议
if_slip.h	1994-05-07 19:12	串行线协议
igmp.h	1995-01-24 21:32	IGMP 协议结构定义
in.h	1995-02-23 19:32	协议号定义
inet.h	1995-01-27 18:17	INET 域部分函数声明
interrupt.h	1995-01-26 13:38	下半部分结构定义
ip.h	1995-04-17 18:47	IP 协议结构定义
ip_fw.h	1995-03-10 02:33	防火墙相关结构定义
ipx.h	1995-01-31 15:36	IPX 包交换协议结构定义
net.h	1995-02-23 19:26	INET 层关键结构定义
netdevice.h	1995-02-05 19:48	设备相关结构定义
notifier.h	1995-01-07 18:57	事件响应相关结构定义
ppp.h	1994-08-11 00:26	点对点协议结构定义
route.h	1994-08-11 00:26	路由结构定义
skbuff.h	1995-01-24 21:27	数据包封装结构定义
socket.h	1995-02-02 02:03	常数选项定义
sockios.h	1995-01-24 21:27	选项定义
tcp.h	1995-04-17 18:47	TCP 协议结构定义
timer.h	1995-01-23 03:30	定时器相关结构定义
udp.h	1993-12-01 20:44	UDP 协议结构定义
un.h	1994-06-17 12:53	UNIX 域地址结构定义

下文中原始头文件代码显示在两行包含该文件路径的星号之间。如下所示。

```
/* include/linux/etherdevice.h ************************************************/
//etherdevice.h 头文件内容
/* include/linux/etherdevice.h ************************************************/
```

路由器与网关的区别：路由器和网关都是网络中连接不同子网的主机。二者都可对到达该主机的数据包进行转发。但二者具有本质区别。路由器相对网关而言较为简单。路由器工作在 OSI 模型的物理层，链路层和网络层。路由器在多个互联网之间中继包。它们将来自某个网络的包路由到互联网上任何可能的目的网络中。路由器区别于网关的最大之处在于路由器本身只能在使用相同协议的网络中转发数据包。而网关是一个协议转换器，其可以接收一种协议的数据包如 AppleTalk 格式的包，然后在转发之前将其转换成另一种协议形式的包如 TCP/IP 格式继而发送出去。另外网关可能工作在 OSI 模型的所有七层之中。另外需要澄清的一点多协议路由器仅仅表示该路由器可转发多种协议格式的包，如一个路由器既可转发 IP 格式的包，亦可转发 IPX（Novell 网的网络层协议）格式的包，如此工作模式的路由器对于每种协议都有一张路由表。注意多协议路由器与单协议路由器本质相同，且区别于网关，多协议路由器仍然不可对数据包进行协议上的格式转换，而仅仅在于其内部集成了多个协议的路由器，使得其可以转发多种协议格式的数据包，而网关可更改数据包的格式。

1.1 etherdevice.h 头文件

该头文件声明了以太网用于建立 MAC 首部的函数，其中 eth_header 函数用于首次进行 MAC 首部建立，而 eth_rebuild_header 则用于首次建立 MAC 首部失败后之后的重新建立。当暂时无法确知 IP 地址所对应的硬件地址时，系统会在之后调用 eth_rebuild_header 函数重新建立 MAC 首部。eth_trans_type 被处理接收数据包的函数调用用以得知以太网首部中的 TYPE 字段值，系统将根据这个值调用相应的上一层处理函数（如对应 IP 数据包为 ip_rcv）。

```
/* include/linux/etherdevice.h ****************************************************/
/* INET         An implementation of the TCP/IP protocol suite for the LINUX
 *         operating system.   NET   is implemented using the   BSD Socket
 *         interface as the means of communication with the user level.
 *
 *         Definitions for the Ethernet handlers.
 *
 * Version:    @(#)eth.h       1.0.4       05/13/93
 *
 * Authors:    Ross Biro, <bir7@leland.Stanford.Edu>
 *         Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
 *
 *         Relocated to include/linux where it belongs by Alan Cox
 *                                      <gw4pts@gw4pts.ampr.org>
 *
 *         This program is free software; you can redistribute it and/or
 *         modify it under the terms of the GNU General Public License
 *         as published by the Free Software Foundation; either version
 *         2 of the License, or (at your option) any later version.
 *
 *   WARNING: This move may well be temporary.
    This file will get merged with others RSN.
 *
 */
```

```
#ifndef _LINUX_ETHERDEVICE_H
#define _LINUX_ETHERDEVICE_H

#include <linux/if_ether.h>
#ifdef __KERNEL__
extern int        eth_header(unsigned char *buff, struct device *dev,
                             unsigned short type, void *daddr,
                             void *saddr, unsigned len,
                             struct sk_buff *skb);
extern int        eth_rebuild_header(void *buff, struct device *dev,
                             unsigned long raddr, struct sk_buff *skb);
extern unsigned short      eth_type_trans(struct sk_buff *skb, struct device *dev);
#endif
#endif      /* _LINUX_ETHERDEVICE_H */
/* include/linux/etherdevice.h **************************************************/
```

1.2 icmp.h 头文件

该头文件定义 ICMP 首部以及 ICMP 错误类型。一个 ICMP 错误包括两个部分：类型（type）和代码（code），分别对应 ICMP 首部中的 type 和 code 字段。

ICMP 本身是网络层协议，但其报文段并不是直接传送给链路层。实际上，ICMP 报文首先要封装成 IP 数据报。图 1.1 显示了相关关系。

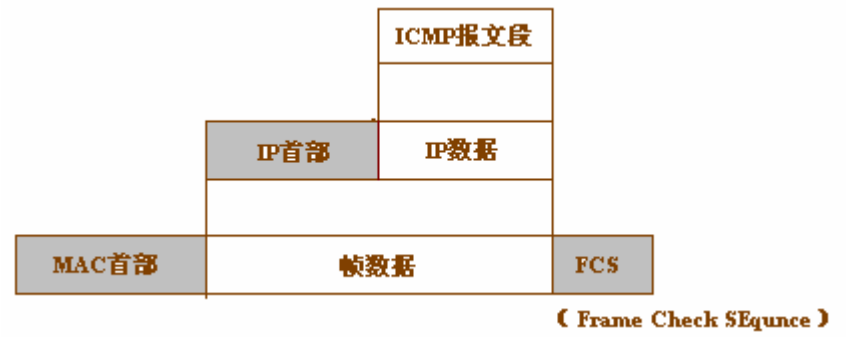


图 1.1 ICMP 的封装

根据 RFC-792，有如下类型 ICMP 错误。

1． 目的端不可达：Type=3

该 ICMP 错误对应的首部格式如图 1.2 所示。

图 1.2 目的端不可达 ICMP 首部格式

其中 Code 取值表示具体的错误代码，可取如下值：
0— 网络不可达
1— 主机不可达
2—（网络层）协议不可达（目的端不认识该协议，无相应处理函数）
3— 端口不可达
4— 需要分片但 IP 首部中 DF 位置 1
5— 源路由失败
其它 Code 值如该文件中定义。
该文件中第一簇 define 语句定义了 ICMP 错误类型，之后的定义为某类型对应的具体代码值如第二簇 define 语句定义的是类型 Type=3 对应的所有的可能代码值，类型值表示错误的分类，而代码值表示的是某个分类中的具体的错误类型。

检验和字段包括 ICMP 报文段的所有数据，包括 ICMP 首部。

首部中“原 IP 首部+8 字节 IP 数据”指的是引起该 ICMP 错误报文的数据包中 IP 首部及其 8 字节 IP 数据。如果从本地的角度看，由于之前发送给远端的数据包存在某种错误，由于这种错误现在本地接收到一个 ICMP 错误通知报文，该报文中包含了之前这个存在错误的数据包的 IP 首部及其 8 字节的 IP 数据。

返回原 IP 首部及其 8 字节 IP 数据可以使得本地得知哪个应用程序（从端口号得知，而端口号包含在 8 字节的 IP 数据中）发送了这个有问题的数据包，从而通知该应用程序或者采取其它相关措施如错误不可恢复时则简单关闭发送该问题数据包的套接字。

```
/* include/linux/icmp.h ************************************************************/
/*
 * INET        An implementation of the TCP/IP protocol suite for the LINUX
 *        operating system.    INET is implemented using the    BSD Socket
 *        interface as the means of communication with the user level.
 *
 *        Definitions for the ICMP protocol.
 *
 * Version:    @(#)icmp.h    1.0.3       04/28/93
 *
 * Author:     Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
 *
 *        This program is free software; you can redistribute it and/or
 *        modify it under the terms of the GNU General Public License
 *        as published by the Free Software Foundation; either version
 *        2 of the License, or (at your option) any later version.
 */
#ifndef _LINUX_ICMP_H
#define    _LINUX_ICMP_H
```

```
#define ICMP_ECHOREPLY      0    /* Echo Reply              */
#define ICMP_DEST_UNREACH   3    /* Destination Unreachable */
#define ICMP_SOURCE_QUENCH  4    /* Source Quench           */
#define ICMP_REDIRECT       5    /* Redirect (change route) */
#define ICMP_ECHO           8    /* Echo Request            */
#define ICMP_TIME_EXCEEDED  11   /* Time Exceeded           */
#define ICMP_PARAMETERPROB  12   /* Parameter Problem       */
#define ICMP_TIMESTAMP      13   /* Timestamp Request       */
#define ICMP_TIMESTAMPREPLY 14   /* Timestamp Reply         */
#define ICMP_INFO_REQUEST   15   /* Information Request     */
#define ICMP_INFO_REPLY     16   /* Information Reply       */
#define ICMP_ADDRESS        17   /* Address Mask Request    */
#define ICMP_ADDRESSREPLY   18   /* Address Mask Reply      */


/* Codes for UNREACH. */
#define ICMP_NET_UNREACH    0    /* Network Unreachable     */
#define ICMP_HOST_UNREACH   1    /* Host Unreachable        */
#define ICMP_PROT_UNREACH   2    /* Protocol Unreachable    */
#define ICMP_PORT_UNREACH   3    /* Port Unreachable        */
#define ICMP_FRAG_NEEDED    4    /* Fragmentation Needed/DF set */
#define ICMP_SR_FAILED      5    /* Source Route failed     */
#define ICMP_NET_UNKNOWN    6
#define ICMP_HOST_UNKNOWN   7
#define ICMP_HOST_ISOLATED  8
#define ICMP_NET_ANO        9
#define ICMP_HOST_ANO       10
#define ICMP_NET_UNR_TOS    11
#define ICMP_HOST_UNR_TOS   12

/* Codes for REDIRECT. */
#define ICMP_REDIR_NET      0    /* Redirect Net            */
#define ICMP_REDIR_HOST     1    /* Redirect Host           */
#define ICMP_REDIR_NETTOS   2    /* Redirect Net for TOS    */
#define ICMP_REDIR_HOSTTOS  3    /* Redirect Host for TOS   */

/* Codes for TIME_EXCEEDED. */
#define ICMP_EXC_TTL        0    /* TTL count exceeded      */
#define ICMP_EXC_FRAGTIME   1    /* Fragment Reass time exceeded */

struct icmphdr {
  unsigned char       type;
  unsigned char       code;
  unsigned short      checksum;
```

```
    union {
        struct {
                unsigned short id;
                unsigned short sequence;
        } echo;
        unsigned long gateway;
    } un;
};


struct icmp_err {
    int             errno;
    unsigned        fatal:1;
};
#endif      /* _LINUX_ICMP_H */
/* include/linux/icmp.h ************************************************************/
```

2．超时错误：Type=11

该错误类型对应的 ICMP 首部格式如图 1.3 所示。

图 1.3 ICMP 超时错误首部格式

其中 Code 字段可取如下值：

0— 传输中 TTL（生存时间）超时

1— 分片组合时间超时（某个数据包的所有分片在规定时间内未完全到达）

注意对于 Code=1，如果第一个分片尚未到达而此时发生超时，则不发送 ICMP 错误报文。

其它字段与前文中所述意义相同。

3．参数错误：Type=12

图 1.4 显示了该错误类型对应的 ICMP 首部格式。

图 1.4 ICMP 参数错误类型首部格式

此处首部中多出一个指针字段。如果 Code=0，则该字段表示错误参数所在的字节偏移量。

为使指针字段有效，一般代码（Code）字段取值为 0。其它字段意义同前文介绍。

4．源端节制（Source Quench）报文：Type=4
图 1.5 显示了该类型 ICMP 报文首部格式。

图 1.5 ICMP 源端节制首部格式

此种类型的 ICMP 报文并不是由本地发送的一个问题报文引起的。而是远端为了缓解接收压力发送给本地的通知其减缓发送报文的速度，本地接收到该报文后所采取的策略应是延缓发送本地报文（即进行源端节制），但实现上（如本版本 LINUX 网络代码）只是更新统计信息，并不采取具体的节制措施。注意此处所指的远端不一定是数据包的最终目的端，大多指的是中间的某个路由器或网关。当这些路由器或网关没有足够内存缓存需要转发的数据包时，就会发送一个这样的 ICMP 报文给这些数据包的发送端（即源端）。

其中 Code 字段取值应为 0。其它字段意义同前文。

5．重定向（Redirect）报文：Type=5
图 1.6 显示了该 ICMP 报文类型对应的首部格式。

图 1.6 ICMP 重定向报文首部格式

其中 Code 可取如下值：
0—基于网络的重定向报文
1—基于主机的重定向报文
2—基于服务类型和网络的重定向报文
3—基于服务类型和主机的重定向报文

新网关 IP 地址表示数据包应该发送到的网关 IP 地址。
一个网关在如下情况下会发送一个重定向 ICMP 报文：
网关 G1 从其连接的局域网某个主机接收到一个数据包，G1 查询路由表项得到下一站网关 G2 的 IP 地址，如果发现 G2 的 IP 地址与所转发数据包中内含的源端 IP 地址同属于一个网络，即表示 G2 与源端主机在同一个局域网中，此时发送一个 ICMP 重定向报文给源端，通知其今后直接发送报文给 G2，不需要经过 G1 转发，此时 ICMP 首部中新网关 IP 地址即是 G2 的 IP 地址。另外一种判断方式是：该数据包进站接口与转发该数据包时的出站接口是同

一个接口。

6．回复（Echo）和回复应答（Echo Reply）报文
Type=8 回复报文，Type=0 回复应答报文
图 1.7 显示了这两种类型报文的 ICMP 首部格式。

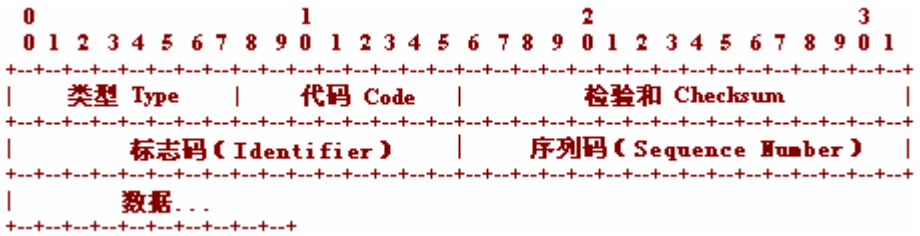


图 1.7 ICMP 回复和回复应答报文首部格式

其中 Code 字段值在两种情况下均取 0。
标志码和序列码由发送回复报文端填写，回复应答报文发送端必须原样返回这两个字段的值进行匹配，使得源端确定此应答报文是否是对本地回复报文的应答。

数据部分由回复报文发送端填写，回复应答发送端也必须原样返回这些数据。
其它字段意义同前文。

7．时间戳（Timestamp）和时间戳应答（Timestamp Reply）报文
Type=13 时间戳报文； Type=14 时间戳应答报文
图 1.8 显示了这两种报文类型的 ICMP 首部格式。

图 1.8 ICMP 时间戳和时间戳应答报文首部格式

其中 Code 取值在两种情况下均为 0。
起始时间戳字段值由本地发送端填写，表示该起始 ICMP 时间戳报文由本地发送的时间。接收时间戳字段表示远端接收到该报文时的时间值，发送时间戳表示远端发送应答报文时的时间值。实现上，远端发送时间戳应答报文时，复制所接收时间戳报文中的标志码，序列码，起始时间戳。之后填写其它首部字段如类型值变为 14。理论上，接收时间戳和发送时间戳是两个不同的值，但实现上，这两个字段都被赋值为相同的值即发送该应答报文时的时间值。

8．信息查询（Information Request）和信息应答（Information Reply）报文
Type=15 信息查询报文； Type=16 信息应答报文
图 1.9 显示了这两种报文的 ICMP 首部格式。

图 1.9 ICMP 信息查询和信息应答首部格式

其中 Code 字段取值均为 0。标志码与序列码用于匹配。

信息查询报文中 IP 首部中源端地址初始化为本地 IP 地址，而目的地址被初始化为 0，表示本地网络。相应的信息应答报文 IP 首部中源端和目的端 IP 地址则按正常方式填写。

信息查询和信息应答报文用于主机查询该主机连接的网络数。

综合以上，有如下 ICMP 类型（以 Type 值为依据）：
0—回复应答（Echo Reply）报文
3—目的端不可达（Destination Unreachable）报文
4—源端节制（Source Quench）报文
5—重定向（Redirect）报文
8—回复（Echo）报文
11—超时（Time Exceeded）报文
12—参数错误（Parameter Problem）报文
13—时间戳（Timestamp）报文
14—时间戳应答（Timestamp Reply）报文
15—信息查询（Information Request）报文
16—信息应答(Information Reply) 报文

此时对应查看 icmp.h 文件中各种常数定义应不难理解。
在该文件尾部定义有 icmp_err 结构，如下所示：

```
struct icmp_err {
  int           errno;
  unsigned      fatal:1;
};
```

结构中 errno 字段表示具体的错误，而 fatal 字段值表示这种错误是否是可恢复的：1-不可恢复，0-可恢复。如果不可恢复，则关闭套接字连接。

该结构使用在 icmp.c 文件中，以下代码片断摘取自 net/icmp.c 文件，

```
/* An array of errno for error messages from dest unreach. */
struct icmp_err icmp_err_convert[] = {
  { ENETUNREACH,   0 },/*   ICMP_NET_UNREACH      */
  { EHOSTUNREACH,  0 },   /*   ICMP_HOST_UNREACH    */
  { ENOPROTOOPT,   1 }, /*   ICMP_PROT_UNREACH    */
  { ECONNREFUSED,  1 },/*   ICMP_PORT_UNREACH    */
  { EOPNOTSUPP,    0 },   /*   ICMP_FRAG_NEEDED      */
```

```
  { EOPNOTSUPP,  0 },   /*  ICMP_SR_FAILED          */
  { ENETUNREACH,  1 }, /*  ICMP_NET_UNKNOWN    */
  { EHOSTDOWN,  1 },   /*  ICMP_HOST_UNKNOWN  */
  { ENONET,  1 },   /*  ICMP_HOST_ISOLATED   */
  { ENETUNREACH,  1 }, /*  ICMP_NET_ANO        */
  { EHOSTUNREACH,  1 },   /*  ICMP_HOST_ANO          */
  { EOPNOTSUPP,  0 },   /*  ICMP_NET_UNR_TOS */
  { EOPNOTSUPP,  0 }    /*  ICMP_HOST_UNR_TOS    */
};
```

ICMP 协议处理函数在接收到一个 ICMP 错误报告时，根据接收到的具体错误查询该表，根据 fatal 字段值确定随后的操作。如果该错误不可恢复（fatal=1），则进行套接字关闭操作，否则忽略之。

1.3 if.h 头文件

该文件定义有接口标志位，ifaddr 结构，ifreq 结构，ifmap 结构，ifconf 结构。

接口标志位标志接口的状态。

ifmap，ifreq，ifconf 结构均使用在选项设置和获取函数中。

ifaddr 目前暂未使用。ifmap 结构用于设置设备的基础信息，这一点可以从 ifmap 结构中字段与 device 结构（netdevice.h）中字段间匹配关系看出。ifmap 结构目前作为 set_config, ether_config 函数的参数而存在。ifreq，ifconf 也是主要用于设备接口信息的获取和设置，使用在 dev_ifconf, dev_ifsioc 函数中（二者均定义在 dev.c 文件中）。 有关各个结构中字段的说明在文件中有简单的注释。

```
/* include/linux/if.h ************************************************************/
/*
 * INET        An implementation of the TCP/IP protocol suite for the LINUX
 *        operating system.  INET is implemented using the  BSD Socket
 *        interface as the means of communication with the user level.
 *
 *        Global definitions for the INET interface module.
 *
 * Version:    @(#)if.h  1.0.2       04/18/93
 *
 * Authors:    Original taken from Berkeley UNIX 4.3, (c) UCB 1982-1988
 *        Ross Biro, <bir7@leland.Stanford.Edu>
 *        Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
 *
 *        This program is free software; you can redistribute it and/or
 *        modify it under the terms of the GNU General Public License
 *        as published by the Free Software Foundation; either version
 *        2 of the License, or (at your option) any later version.
 */
#ifndef _LINUX_IF_H
```

```
#define _LINUX_IF_H

#include <linux/types.h>        /* for "caddr_t" et al          */
#include <linux/socket.h>       /* for "struct sockaddr" et al  */

/* Standard interface flags. */
#define IFF_UP          0x1         /* interface is up          */
#define IFF_BROADCAST   0x2         /* broadcast address valid  */
#define IFF_DEBUG   0x4         /* turn on debugging            */
#define IFF_LOOPBACK    0x8         /* is a loopback net        */
#define IFF_POINTOPOINT     0x10            /* interface is has p-p link    */
#define IFF_NOTRAILERS      0x20            /* avoid use of trailers    */
#define IFF_RUNNING     0x40            /* resources allocated          */
#define IFF_NOARP   0x80            /* no ARP protocol          */
#define IFF_PROMISC     0x100           /* receive all packets          */
/* Not supported */
#define IFF_ALLMULTI    0x200           /* receive all multicast packets*/

#define IFF_MASTER  0x400           /* master of a load balancer    */
#define IFF_SLAVE       0x800           /* slave of a load balancer     */

#define IFF_MULTICAST   0x1000          /* Supports multicast           */

/*
 * The ifaddr structure contains information about one address
 * of an interface.  They are maintained by the different address
 * families, are allocated and attached when an address is set,
 * and are linked together so all addresses for an interface can
 * be located.
 */

struct ifaddr
{
    struct sockaddr     ifa_addr; /* address of interface 接口地址*/
    union {
        struct sockaddr     ifu_broadaddr; //接口广播地址
        struct sockaddr     ifu_dstaddr; //接口目的端地址，只使用在点对点连接中
    } ifa_ifu;
    struct iface            *ifa_ifp;  /* back-pointer to interface    */
    struct ifaddr           *ifa_next;      /* next address for interface   */
};

#define ifa_broadaddr ifa_ifu.ifu_broadaddr     /* broadcast address        */
#define ifa_dstaddr     ifa_ifu.ifu_dstaddr  /* other end of link */
```

```
/*
 *    Device mapping structure. I'd just gone off and designed a
 *    beautiful scheme using only loadable modules with arguments
 *    for driver options and along come the PCMCIA people 8)
 *
 *    Ah well. The get() side of this is good for WDSETUP, and it'll
 *    be handy for debugging things. The set side is fine for now and
 *    being very small might be worth keeping for clean configuration.
 */

struct ifmap
{
      unsigned long mem_start; //硬件读写缓冲区首地址
      unsigned long mem_end; //硬件读写缓冲区尾地址
      unsigned short base_addr; //本设备所使用 I/O 端口地址
      unsigned char irq; //本设备所使用的中断号
      unsigned char dma; //本设备所使用的 dma 通道号
      unsigned char port; 对于 device 结构中的 if_port 字段，该字段使用在某些特殊情况下
      /* 3 bytes spare */
};

/*
 * Interface request structure used for socket
 * ioctl's.    All interface ioctl's must have parameter
 * definitions which begin with ifr_name.    The
 * remainder may be interface specific.
 */
//每个 ifreq 表示一个设备接口的信息。
struct ifreq
{
#define IFHWADDRLEN     6
#define    IFNAMSIZ     16
      union
      {
            char ifrn_name[IFNAMSIZ];        /* if name, e.g. "en0"  设备接口名*/
            char ifrn_hwaddr[IFHWADDRLEN];   /* Obsolete */
      } ifr_ifrn;

      union {
            struct      sockaddr ifru_addr; //设备 IP 地址
            struct      sockaddr ifru_dstaddr; //点对点连接中对端地址
            struct      sockaddr ifru_broadaddr; //广播 IP 地址
            struct      sockaddr ifru_netmask; //地址掩码
```

```
        struct    sockaddr ifru_hwaddr; //设备对应的硬件地址
        short       ifru_flags; //设备对应的标志字段值
        int    ifru_metric; //代价值
        int    ifru_mtu; //设备对应的最大传输单元
        struct    ifmap ifru_map; //该结构用于设置/获取设备的基本信息
        char ifru_slave[IFNAMSIZ]; /* Just fits the size */
        caddr_t    ifru_data; /*caddr_t 定义在 ctype.h 文件中：typedef char * caddr_t
                              *该字段的作用相当于 raw，即只表示存储单元或存储数据。*/
    } ifr_ifru;
};

#define ifr_name    ifr_ifrn.ifrn_name   /* interface name    */
#define old_ifr_hwaddr  ifr_ifrn.ifrn_hwaddr       /* interface hardware      */
#define ifr_hwaddr ifr_ifru.ifru_hwaddr       /* MAC address            */
#define    ifr_addr   ifr_ifru.ifru_addr    /* address              */
#define    ifr_dstaddr      ifr_ifru.ifru_dstaddr       /* other end of p-p lnk    */
#define    ifr_broadaddr  ifr_ifru.ifru_broadaddr   /* broadcast address       */
#define    ifr_netmask     ifr_ifru.ifru_netmask      /* interface net mask      */
#define    ifr_flags  ifr_ifru.ifru_flags   /* flags          */
#define    ifr_metricifr_ifru.ifru_metric /* metric         */
#define    ifr_mtu          ifr_ifru.ifru_mtu     /* mtu                  */
#define ifr_map             ifr_ifru.ifru_map     /* device map         */
#define ifr_slave     ifr_ifru.ifru_slave   /* slave device       */
#define    ifr_data    ifr_ifru.ifru_data     /* for use by interface    */

/*
 * Structure used in SIOCGIFCONF request.
 * Used to retrieve interface configuration
 * for machine (useful for programs which
 * must know all networks accessible).
 */
//该结构用于信息获取，ifc_len 表示缓冲区长度，而 ifc_ifcu 字段则存储具体信息。
//信息是以 ifreq 结构为单元，ifcu_buf 指向的缓冲区中存储的信息是一个 ifreq 结构数组。
//每个 ifreq 结构代表一个设备接口的信息。系统将遍历系统中所有设备接口，向 ifcu_buf 指
//指向的缓冲区中填写设备信息，直到缓冲区满或者设备接口遍历完。
struct ifconf
{
    int    ifc_len;                /* size of buffer */
    union
    {
        caddr_t    ifcu_buf;
        struct      ifreq *ifcu_req;
    } ifc_ifcu;
};
```

```
#define    ifc_buf    ifc_ifcu.ifcu_buf          /* buffer address     */
#define    ifc_req    ifc_ifcu.ifcu_req          /* array of structures      */


/* BSD UNIX expects to find these here, so here we go: */
#include <linux/if_arp.h>
#include <linux/route.h>

#endif /* _NET_IF_H */
/* include/linux/if.h ************************************************************/
```

1.4 if_arp.h 头文件

该文件定义 ARP 首部以及与之相关的常数值。

ARP 协议全称为地址解析协议（Address Resolution Protocol）。ARP 为 IP 地址到对应的硬件地址之间提供动态映射。ARP 数据包的封装形式如图 1.10 所示。

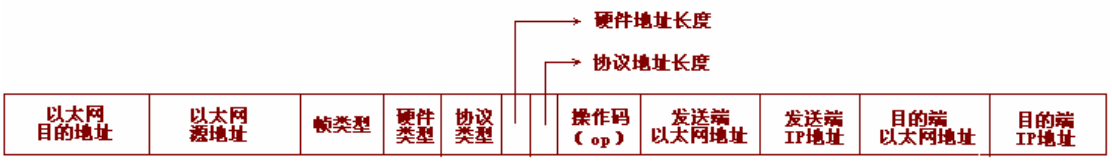


图 1.10 用于以太网的 ARP 请求或应答分组格式

ARP 请求使用广播形式，而 ARP 应答则使用单播形式。一个 ARP 请求数据包是用于询问协议地址（一般为 IP 地址）所对应的硬件地址（即以太网首部中使用的地址，亦称以太网地址）。

以太网头部中的目的硬件地址对于 ARP 请求而言设置为全 1，表示向网络广播 ARP 请求。源端以太网地址表示该 ARP 数据包的本地发送端硬件地址。以太网类型字段对于 ARP 请求和应答而言均为 0x0806.

硬件类型字段表示硬件地址的类型。该字段值为 1 表示以太网地址。

协议类型字段表示要映射的协议地址类型。该字段值为 0x0800 即表示 IP 地址。

接下来两个 1-byte 的字段，即硬件地址长度和协议地址长度字分别指出硬件地址和协议地址的长度，以字节为单位。对于以太网上 IP 地址的请求与应答而言，它们的值分别为 6 和 4。

图 3 中 op 字段为两个字节，该字段具体表示 ARP 协议的类型：

ARP 请求，对应值为 1

ARP 应答，对应值为 2

RARP 请求，值为 3

RARP 应答，值为 4

该字段是必须的。

接下来的四个字段是发送端硬件地址（以太网地址），发送端的协议地址（IP 地址），目的端硬件地址和目的端协议地址。此处有一个重复之地：在以太网的数据帧以太网首部中和 ARP 协议首部中都有发送端的硬件地址。

对于一个 ARP 请求而言，除目的端硬件地址字段外，其他字段都有填充值。当系统接收到一份目的端协议地址为本机的协议地址的 ARP 请求报文后，它就将其硬件地址填充到相应字段，然后在修改 ARP 首部一些字段值之后（如修改操作字段值为 2，修改发送端和目的端硬件地址和协议地址），就将数据包发送出去（注意此时是单播发送）。

```
/* include/linux/if_arp.h ************************************************************/
/*
 * INET        An implementation of the TCP/IP protocol suite for the LINUX
 *        operating system.    INET is implemented using the    BSD Socket
 *        interface as the means of communication with the user level.
 *
 *        Global definitions for the ARP (RFC 826) protocol.
 *
 * Version:    @(#)if_arp.h   1.0.1       04/16/93
 *
 * Authors:    Original taken from Berkeley UNIX 4.3, (c) UCB 1986-1988
 *        Portions taken from the KA9Q/NOS (v2.00m PA0GRI) source.
 *        Ross Biro, <bir7@leland.Stanford.Edu>
 *        Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
 *        Florian La Roche.
 *
 *        This program is free software; you can redistribute it and/or
 *        modify it under the terms of the GNU General Public License
 *        as published by the Free Software Foundation; either version
 *        2 of the License, or (at your option) any later version.
 */
#ifndef _LINUX_IF_ARP_H
#define _LINUX_IF_ARP_H

//arp 首部中硬件地址类型值
/* ARP protocol HARDWARE identifiers. */
#define ARPHRD_NETROM0          /* from KA9Q: NET/ROM pseudo*/
#define ARPHRD_ETHER    1       /* Ethernet 10Mbps      */
#define    ARPHRD_EETHER    2          /* Experimental Ethernet      */
#define    ARPHRD_AX25    3       /* AX.25 Level 2          */
#define    ARPHRD_PRONET    4          /* PROnet token ring          */
#define    ARPHRD_CHAOS5       /* Chaosnet               */
#define    ARPHRD_IEEE802    6          /* IEEE 802.2 Ethernet- huh? */
#define    ARPHRD_ARCNET    7          /* ARCnet                 */
```

```
#define    ARPHRD_APPLETLK  8          /* APPLEtalk              */
/* Dummy types for non ARP hardware */
#define ARPHRD_SLIP 256
#define ARPHRD_CSLIP     257
#define ARPHRD_SLIP6     258
#define ARPHRD_CSLIP6    259
#define ARPHRD_RSRVD     260       /* Notional KISS type          */
#define ARPHRD_ADAPT     264
#define ARPHRD_PPP   512
#define ARPHRD_TUNNEL 768       /* IPIP tunnel             */

//arp 首部中操作码字段值
/* ARP protocol opcodes. */
#define    ARPOP_REQUEST      1          /* ARP request                */
#define    ARPOP_REPLY    2          /* ARP reply              */
#define    ARPOP_RREQUEST    3          /* RARP request               */
#define    ARPOP_RREPLY  4          /* RARP reply             */

//该结构用于设置/获取 arp 表项信息，在 arp_req_set, arp_req_get 函数中（arp.c）使用。
/* ARP ioctl request. */
struct arpreq {
   struct sockaddr   arp_pa;        /* protocol address       */
   struct sockaddr   arp_ha;        /* hardware address       */
   int               arp_flags;/* flags                */
   struct sockaddr           arp_netmask;       /* netmask (only for proxy arps) */
};

//如下定义 ARP 表项的状态值。
/* ARP Flag values. */
#define ATF_COM        0x02          /* completed entry (ha valid)  */
#define    ATF_PERM    0x04          /* permanent entry        */
#define    ATF_PUBL    0x08          /* publish entry          */
#define    ATF_USETRAILERS    0x10            /* has requested trailers  */
#define ATF_NETMASK         0x20                /* want to use a netmask (only
                                for proxy entries) */

/*
   *  This structure defines an ethernet arp header.
 */
//ARP 首部，结合图 1.10 理解该首部定义。
struct arphdr
{
      unsigned short       ar_hrd;          /* format of hardware address*/
      unsigned short       ar_pro;          /* format of protocol address  */
```

```
    unsigned char  ar_hln;          /* length of hardware address */
    unsigned char  ar_pln;          /* length of protocol address  */
    unsigned short ar_op;           /* ARP opcode (command)        */

#if 0
     /*
      *    Ethernet looks like this : This bit is variable sized however...
      */
    unsigned char       ar_sha[ETH_ALEN];   /* sender hardware address    */
    unsigned char       ar_sip[4];      /* sender IP address      */
    unsigned char       ar_tha[ETH_ALEN];   /* target hardware address    */
    unsigned char       ar_tip[4];      /* target IP address      */
#endif

};

#endif   /* _LINUX_IF_ARP_H */
/* include/linux/if_arp.h ************************************************************/
```

1.5 if_ether.h 头文件

以太网首部 ethhdr 定义和一些常量定义。文件尾部 enet_statistics 结构用于统计信息记录。常量主要是以太网首部中类型字段可取值。

```
/* include/linux/if_ether.h ************************************************************/
/*
 * INET         An implementation of the TCP/IP protocol suite for the LINUX
 *          operating system.   INET is implemented using the   BSD Socket
 *          interface as the means of communication with the user level.
 *
 *          Global definitions for the Ethernet IEEE 802.3 interface.
 *
 * Version:     @(#)if_ether.h 1.0.1a      02/08/94
 *
 * Author:      Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
 *          Donald Becker, <becker@super.org>
 *
 *          This program is free software; you can redistribute it and/or
 *          modify it under the terms of the GNU General Public License
 *          as published by the Free Software Foundation; either version
 *          2 of the License, or (at your option) any later version.
 */
#ifndef _LINUX_IF_ETHER_H
#define _LINUX_IF_ETHER_H
```

```
/* IEEE 802.3 Ethernet magic constants.  The frame sizes omit the preamble
   and FCS/CRC (frame check sequence). */
//一些长度常量定义，如以太网硬件地址长度为 6，以太网首部长度为 14，等。
#define ETH_ALEN    6           /* Octets in one ethernet addr   */
#define ETH_HLEN    14          /* Total octets in header.   */
#define ETH_ZLEN    60          /* Min. octets in frame sans FCS */
//IP 首部及其数据部分长度，通常称之为 MTU。
#define ETH_DATA_LEN    1500            /* Max. octets in payload       */
//帧长度，1514=1500+14,有时加上帧尾部的 FCS 字段长度为 1518。
#define ETH_FRAME_LEN 1514              /* Max. octets in frame sans FCS */

//以太网首部中类型字段可取值。
/* These are the defined Ethernet Protocol ID's. */
#define ETH_P_LOOP  0x0060          /* Ethernet Loopback packet  */
#define ETH_P_ECHO  0x0200          /* Ethernet Echo packet      */
#define ETH_P_PUP   0x0400          /* Xerox PUP packet          */
#define ETH_P_IP 0x0800         /* Internet Protocol packet     */
#define ETH_P_ARP   0x0806          /* Address Resolution packet */
#define ETH_P_RARP          0x8035      /* Reverse Addr Res packet   */
#define ETH_P_X25   0x0805          /* CCITT X.25                */
#define ETH_P_ATALK0x809B           /* Appletalk DDP         */
#define ETH_P_IPX   0x8137          /* IPX over DIX              */
#define ETH_P_802_3 0x0001          /* Dummy type for 802.3 frames   */
#define ETH_P_AX25  0x0002          /* Dummy protocol id for AX.25   */
#define ETH_P_ALL   0x0003          /* Every packet (be careful!!!) */
#define ETH_P_802_2 0x0004          /* 802.2 frames              */
#define ETH_P_SNAP  0x0005          /* Internal only             */

//以太网首部定义。
/* This is an Ethernet frame header. */
struct ethhdr {
  unsigned char         h_dest[ETH_ALEN];   /* destination eth addr   */
  unsigned char         h_source[ETH_ALEN]; /* source ether addr*/
  unsigned short    h_proto;        /* packet type ID field   */
};
//进出站数据包统计信息结构
/* Ethernet statistics collection data. */
struct enet_statistics{
  int       rx_packets;             /* total packets received */
  int       tx_packets;             /* total packets transmitted    */
  int       rx_errors;          /* bad packets received         */
  int       tx_errors;          /* packet transmit problems   */
  int       rx_dropped;             /* no space in linux buffers    */
```

```
    int     tx_dropped;             /* no space available in linux */
    int     multicast;          /* multicast packets received */
    int     collisions;

    /* detailed rx_errors: */
    int     rx_length_errors;
    int     rx_over_errors;             /* receiver ring buff overflow */
    int     rx_crc_errors;          /* recved pkt with crc error    */
    int     rx_frame_errors;        /* recv'd frame alignment error */
    int     rx_fifo_errors;         /* recv'r fifo overrun          */
    int     rx_missed_errors;       /* receiver missed packet       */

    /* detailed tx_errors */
    int     tx_aborted_errors;
    int     tx_carrier_errors;
    int     tx_fifo_errors;
    int     tx_heartbeat_errors;
    int     tx_window_errors;
};

#endif  /* _LINUX_IF_ETHER_H */
/* include/linux/if_ether.h ************************************************************/
```

1.6 if_plip.h 头文件

PLIP 是 Parallel Line Internet Procotol 的简称，意为并行线网络协议。PLIP 是一种使用特制电缆通过计算机并口进行数据传送的协议。其类似于使用串口进行数据传输的 SLIP 协议，不过 PLIP 每次可以同时传输 4 个比特。这种特制电缆称为 LabLink 或 null-printer 电缆，允许两台计算机通过各自的并口直接建立通信通道，无需网卡或者调制解调器的帮助。这种 LabLink 电缆将两台计算机并口的五个信号管脚直接相连。由于缺少内部时钟同步信号，同步信号需要软件明确完成。于是在五个信号管脚中特意分出一个管脚用于信号同步。故每次可以传送 4 个比特的数据。这些信号管脚上的逻辑值可以通过 I/O 端口读写命令读取。PLIP 协议现在使用很少，已逐渐被基于网络的以太网协议或其他点对点连接协议所取代。现在只在一些老式的计算机上才有并口，新式的计算机上已不附带并口。

```
/* include/linux/if_plip.h *************************************************/
/*
 *  NET3    PLIP tuning facilities for the new Niibe PLIP.
 *
 *  This program is free software; you can redistribute it and/or
 *  modify it under the terms of the GNU General Public License
 *  as published by the Free Software Foundation; either version
 *  2 of the License, or (at your option) any later version.
 *
```

```
 */

#ifndef _LINUX_IF_PLIP_H
#define _LINUX_IF_PLIP_H

#include <linux/sockios.h>

#define    SIOCDEVPLIP      SIOCDEVPRIVATE

//PLIP 配置命令结构
struct plipconf
{
      unsigned short pcmd; //设置（PLIP_SET_TIMEOUT）或获取（PLIP_GET_TIMEOUT）
      unsigned long    nibble; //数据传输超时时间
      unsigned long    trigger; //同步信号传输超时时间
};

#define PLIP_GET_TIMEOUT     0x1
#define PLIP_SET_TIMEOUT     0x2

#endif
/* include/linux/if_plip.h ************************************************************/
```

1.7 if_slip.h 头文件

S L I P的全称是Serial Line IP。它是一种在串行线路上对IP数据报进行封装的简单形式，在RFC 1055中有详细描述。SLIP适用于家庭中每台计算机几乎都有的RS-232串行端口和高速调制解调器接入网络。

下面的规则描述了SLIP协议定义的帧格式：

1) IP数据报以一个称作END（0xc0）的特殊字符结束。同时，为了防止数据报到来之前的线路噪声被当成数据报内容，大多数实现在数据报的开始处也传一个END字符（如果有线路噪声，那么END字符将结束这份错误的报文。这样当前的报文得以正确地传输，而前一个错误报文交给上层后，会发现其内容毫无意义而被丢弃）。

2) 如果IP报文中某个字符为END，那么就要连续传输两个字节0xdb和0xdc来取代它。0xdb这个特殊字符被称作SLIP的ESC字符，但是它的值与ASCII码的ESC字符（0x1b）不同。

3) 如果IP报文中某个字符为SLIP的ESC字符，那么就要连续传输两个字节0xdb和0xdd来取代它。

由于串行线路的速率通常较低（19200 b/s或更低），而且通信经常是交互式的（如Telnet和Rlogin，二者都使用TCP），因此在S L I P线路上有许多小的TCP分组进行交换。为了传送1个字节的数据需要20个字节的IP首部和20个字节的TCP首部，总数超过40个字节。

SLIP的变体有CSLIP即压缩版的SLIP，CSLIP对SLIP报文中各个首部进行必要的压缩以增加数据传输率。CSLIP一般能把上面的40个字节压缩到3或5个字节。

```
/* include/linux/if_slip.h ************************************************************/
/*
 *    Swansea University Computer Society  NET3
 *
 *    This file declares the constants of special use with the SLIP/CSLIP/
 *    KISS TNC driver.
 */

#ifndef __LINUX_SLIP_H
#define __LINUX_SLIP_H

#define         SL_MODE_SLIP        0
#define         SL_MODE_CSLIP       1
#define         SL_MODE_KISS        4

#define         SL_OPT_SIXBIT               2
#define         SL_OPT_ADAPTIVE         8

#endif
/* include/linux/if_slip.h ************************************************************/
```

该文件主要是对一些标志位的声明。

1.8 igmp.h 头文件

IGMP 是 Internet Group Management Protocol 的简称，意为网络组管理协议。IGMP 已有几个版本，以下分析以版本 2 为依据（不过注意本版本网络实现代码只支持 IGMP 版本 1，第二章中对 net/inet/igmp.c 文件分析时，给出版本 1 格式）。图 1-11 显示了 IGMP 的报文类型。

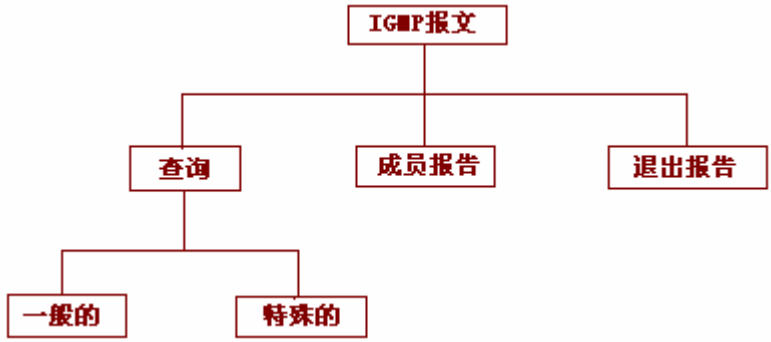


图 1-11 IGMP 报文类型

图 1-12 给出了 IGMPv2 报文首部格式。

图 1-12 IGMP 报文首部格式
类型字段：如表 1-2 所示。
最大响应时间：这个 8 位字段定义了查询必须在多长时间内回答，其单位为 0.1 秒。该字段只在查询报文中有效，在其它类型报文中该字段值为 0。
检验和：计算 IGMP 首部及其数据的检验和。
组地址：在一般查询报文中该字段值为 0。这个字段在特殊查询报文，成员关系报文以及退出报文中为组多播地址（groupip）。

表 1-2 IGMP 类型字段值

类型	值
一般或特殊	0x11
成员关系报告	0x16
退出报告	0x17

类似于 ICMP，虽然 IGMP 协议属于网络层协议，与 IP 协议同属于一个层次，但 IGMP 报文段仍然需要首先被封装在 IP 数据报中，才能传送给链路层。
图 1-13 显示了报文之间的关系。

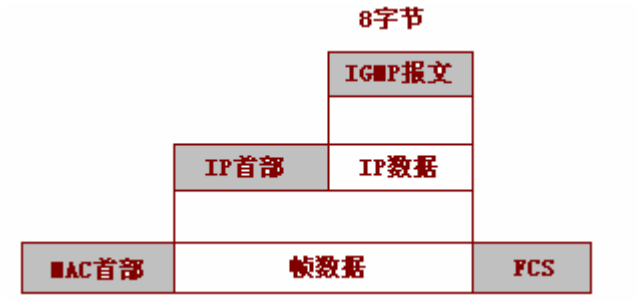


图 1-13 IGMP 分组封装格式
有关 IGMPv2 的详细介绍请参考 RFC-2236。

```
/* include/linux/igmp.h ************************************************************/
/*
 *    Linux NET3:  Internet Gateway Management Protocol    [IGMP]
 *
 *    Authors:
 *          Alan Cox <Alan.Cox@linux.org>
 *
 *
 *    This program is free software; you can redistribute it and/or
 *    modify it under the terms of the GNU General Public License
 *    as published by the Free Software Foundation; either version
 *    2 of the License, or (at your option) any later version.
 */

#ifndef _LINUX_IGMP_H
```

```
#define _LINUX_IGMP_H

/*
 *	IGMP protocol structures
 */
//IGMP 版本 1 首部定义，结合上文介绍不难理解
struct igmphdr
{
	unsigned char type;
	unsigned char unused; //对应版本 2 中的最大响应时间
	unsigned short csum;
	unsigned long group;
};

#define IGMP_HOST_MEMBERSHIP_QUERY 0x11/* From RFC1112 */
//成员关系报告，这是 IGMPv1 版本定义的值，IGMPv2 版本已改为 16。
#define IGMP_HOST_MEMBERSHIP_REPORT	0x12	/* Ditto */
#define IGMP_HOST_LEAVE_MESSAGE	0x17	/* An extra BSD seems to send */

//对于任何一个主机而言，其网络接口在初始化时默认加入 224.0.0.1 多播组。
/* 224.0.0.1 */
#define IGMP_ALL_HOSTS		htonl(0xE0000001L)

/*
 * struct for keeping the multicast list in
 */

#ifdef __KERNEL__
//多播地址及对应的发送接收接口
struct ip_mc_socklist
{
	unsigned long multiaddr[IP_MAX_MEMBERSHIPS];	/* This is a speed trade off */
	struct device *multidev[IP_MAX_MEMBERSHIPS];
};
```

ip_mc_socklist 结构被 sock 结构使用，在本书的分析中，我们可以知道每个套接字都唯一的对应一个 sock 结构，一般而言，一个套接字是与一个进程相绑定的，当进程加入到一个多播组中时，对应的 sock 结构中必须维护加入的多播组的信息，从而底层网络实现可以根据该信息判断是否将接收到的一个多播数据报复制一份送给该 sock 结构对应的进程。所以 ip_mc_socklist 结构从进程的角度上表示了加入的多播组，支持的最大多播组数为 IP_MAX_MEMBERSHIPS.

```
//多播链表
struct ip_mc_list
{
```

```
    struct device *interface; //接收或发送对应多播数据包的接口
    unsigned long multiaddr; //对应的多播地址
    struct ip_mc_list *next;
    struct timer_list timer; //定时器，用于多播查询中，每个主机对于多播查询报文并不立即
                             //应答，而是延迟一段随机时间后，如果在此时间内没有其它具有
                             //相同多播组地址报文被其它主机发送，则方才发送应答报文。
    int tm_running;
    int users;
};
```

ip_mc_list 结构被 device 结构使用，每个 device 结构都唯一对应主机上一个网络设备（此处不考虑虚拟设备）。ip_mc_list 结构即被 device 结构使用，维护多播 IP 地址信息。进程对加入的多播组由一个二元组构成：对应的网络设备以及多播组 IP 地址。换句话说，一个网络接口可以对应本机上多个进程。所以需要有一个结构维护对这些 IP 地址的信息。ip_mc_list 即用于此目的。在第二章中分析 igmp.c 文件时，我们将这种分析该结构的作用。

```
//ip_mc_head 定义在 igmp.c 文件中，作为本地多播地址链表。
extern struct ip_mc_list *ip_mc_head;

//函数声明，这些函数定义在 igmp.c 文件中
extern int igmp_rcv(struct sk_buff *, struct device *, struct options *, unsigned long, unsigned
                    short, unsigned long, int , struct inet_protocol *);
extern void ip_mc_drop_device(struct device *dev);
extern int ip_mc_join_group(struct sock *sk, struct device *dev, unsigned long addr);
extern int ip_mc_leave_group(struct sock *sk, struct device *dev,unsigned long addr);
extern void ip_mc_drop_socket(struct sock *sk);
#endif
#endif
/*include/linux/igmp.h ************************************************************/
```

1.9 in.h 头文件

该头文件主要是一些常量定义。具体情况见文件中注释。

以下简单介绍几种地址分类。

起初使用 IP 地址时，IP 地址使用分类的概念。这种体系结构被称为分类编址。20 世纪 90 年代中期，一种叫做无分类编址的新体系结构出现，这种体系结构将最终替代原来的体系。但目前绝大多数的因特网地址还是使用分类编址方式。本文也只介绍分类编址。

在分类编址中，地址空间被分为 5 类：A，B，C，D，E。每一类占据地址空间的一部分。

1． A 类地址（如下 n 表示网络部分（network），h 代表主机部分（host））

地址组成：0nnnnnnn hhhhhhhh hhhhhhhh hhhhhhhh

1> 第一个比特（MSL 位）为 0，随后 7 比特为网络位，其它 24 比特为主机位。

2> 第一个字节范围：0-127

3> 具有 126 个网络地址：0 和 127 均被保留（如 127 被用于本地回环地址）

4> 每个网络地址对应有 16777214 个主机（2^24）

5> 该类地址网络掩码为：255.0.0.0

2． B 类地址

地址组成：10nnnnnn nnnnnnnn hhhhhhhh hhhhhhhh

1> 前两个比特固定为 10；随后 14 比特为网络部分比特位，其余 16 比特为主机位。

2> 第一个字节范围：128-191

3> 具有 16384 个网络地址

4> 每个网络地址对应有 65532 个主机地址

5> 该类地址网络掩码为：255.255.0.0

3． C 类地址

地址组成：110nnnnn nnnnnnnn nnnnnnnn hhhhhhhh

1> 前三个比特固定为 110；随后 21 个比特为网络地址部分，最后 8 个比特为主机位。

2> 第一字节范围：192-223

3> 具有 2097152 个网络地址

4> 每个网络地址对应有 254 个主机地址

5> 该类地址网络掩码为：255.255.255.0

4． D 类地址(m 代表多播地址位 multicast bit)

地址组成：1110mmmm mmmmmmmm mmmmmmmm mmmmmmmm

1> 前四个比特固定为 1110，其余 28 比特均为多播地址位。

2> 第一字节范围：224-247

3> D 类地址均为多播地址

5． E 类地址（r 代表保留地址位 reserved bit）

地址组成：1111rrrr rrrrrrrr rrrrrrrr rrrrrrrr

1> 前四个比特固定为 1111；其余 28 比特保留。

2> 第一个字节范围：248-255

3> 该类地址为保留地址，或者用于测试目的。

表 1-3 较为清晰的比较了各种地址之间的关系。

表 1-3 各种地址类型比较

地址类别	最左端比特位取值	开始地址	截止地址
A	0xxx	0.0.0.0	127.255.255.255
B	10xx	128.0.0.0	191.255.255.255
C	110x	192.0.0.0	223.255.255.255
D	1110	224.0.0.0	239.255.255.255
E	1111	224.0.0.0	255.255.255.255

结合以上内容不难理解 in.h 文件中地址类别定义。

```
/* include/linux/in.h ************************************************************/
/*
 * INET        An implementation of the TCP/IP protocol suite for the LINUX
 *        operating system.    INET is implemented using the    BSD Socket
 *        interface as the means of communication with the user level.
 *
 *        Definitions of the Internet Protocol.
 *
```

```
 * Version:     @(#)in.h 1.0.1        04/21/93
 *
 * Authors:     Original taken from the GNU Project <netinet/in.h> file.
 *              Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
 *
 *              This program is free software; you can redistribute it and/or
 *              modify it under the terms of the GNU General Public License
 *              as published by the Free Software Foundation; either version
 *              2 of the License, or (at your option) any later version.
 */
#ifndef _LINUX_IN_H
#define _LINUX_IN_H

//IP 首部中上层协议字段取值
/* Standard well-defined IP protocols.  */
enum {
  IPPROTO_IP = 0,           /* Dummy protocol for TCP         */
  IPPROTO_ICMP = 1,         /* Internet Control Message Protocol    */
  IPPROTO_IGMP = 2,         /* Internet Gateway Management Protocol */
  IPPROTO_IPIP = 4,         /* IPIP tunnels (older KA9Q tunnels use 94) */
  IPPROTO_TCP = 6,          /* Transmission Control Protocol  */
  IPPROTO_EGP = 8,          /* Exterior Gateway Protocol      */
  IPPROTO_PUP = 12,         /* PUP protocol                   */
  IPPROTO_UDP = 17,         /* User Datagram Protocol         */
  IPPROTO_IDP = 22,         /* XNS IDP protocol               */

  IPPROTO_RAW = 255,            /* Raw IP packets             */
  IPPROTO_MAX
};

//in_addr 与下面的 sockaddr_in 结构是标准的网络接口地址结构定义。
//熟悉网络编程的读者对此应该不陌生。
/* Internet address. */
struct in_addr {
      unsigned long int    s_addr;
};

//对多播地址或其对应的网络接口进行设置或获取
//该结构作为一个“中间人”而存在
/* Request struct for multicast socket ops */
struct ip_mreq
{
      struct in_addr imr_multiaddr; /* IP multicast address of group */
      struct in_addr imr_interface;  /* local IP address of interface */
```

```
};


/* Structure describing an Internet (IP) socket address. */
#define __SOCK_SIZE__      16          /* sizeof(struct sockaddr)      */
struct sockaddr_in {
   short int            sin_family;     /* Address family           */
   unsigned short int       sin_port;  /* Port number              */
   struct in_addr       sin_addr; /* Internet address          */

   /* Pad to size of `struct sockaddr'. */
   unsigned char            __pad[__SOCK_SIZE__ - sizeof(short int) -
                    sizeof(unsigned short int) - sizeof(struct in_addr)];
};
#define sin_zero    __pad           /* for BSD UNIX comp. -FvK      */

//以下是对几种地址类型的定义
/*
 * Definitions of the bits in an Internet address integer.
 * On subnets, host and network parts are found according
 * to the subnet mask, not these masks.
 */
#define   IN_CLASSA(a)          ((((long int) (a)) & 0x80000000) == 0)
#define   IN_CLASSA_NET       0xff000000
#define   IN_CLASSA_NSHIFT  24
#define   IN_CLASSA_HOST          (0xffffffff & ~IN_CLASSA_NET)
#define   IN_CLASSA_MAX           128

#define   IN_CLASSB(a)          ((((long int) (a)) & 0xc0000000) == 0x80000000)
#define   IN_CLASSB_NET       0xffff0000
#define   IN_CLASSB_NSHIFT  16
#define   IN_CLASSB_HOST          (0xffffffff & ~IN_CLASSB_NET)
#define   IN_CLASSB_MAX           65536

#define   IN_CLASSC(a)          ((((long int) (a)) & 0xe0000000) == 0xc0000000)
#define   IN_CLASSC_NET       0xffffff00
#define   IN_CLASSC_NSHIFT  8
#define   IN_CLASSC_HOST          (0xffffffff & ~IN_CLASSC_NET)

#define   IN_CLASSD(a)          ((((long int) (a)) & 0xf0000000) == 0xe0000000)
#define   IN_MULTICAST(a)         IN_CLASSD(a)
#define IN_MULTICAST_NET    0xF0000000

#define   IN_EXPERIMENTAL(a)      ((((long int) (a)) & 0xe0000000) == 0xe0000000)
```

```
#define    IN_BADCLASS(a)        ((((long int) (a)) & 0xf0000000) == 0xf0000000)

/* Address to accept any incoming messages. */
#define    INADDR_ANY            ((unsigned long int) 0x00000000)

/* Address to send to all hosts. */
#define    INADDR_BROADCAST      ((unsigned long int) 0xffffffff)

/* Address indicating an error return. */
#define    INADDR_NONE           0xffffffff

/* Network number for local host loopback. */
#define    IN_LOOPBACKNET        127

/* Address to loopback in software to local host.    */
#define    INADDR_LOOPBACK       0x7f000001    /* 127.0.0.1      */

/* Defines for Multicast INADDR */
#define INADDR_UNSPEC_GROUP         0xe0000000          /* 224.0.0.0      */
#define INADDR_ALLHOSTS_GROUP       0xe0000001          /* 224.0.0.1      */
#define INADDR_MAX_LOCAL_GROUP      0xe00000ff          /* 224.0.0.255 */

/* <asm/byteorder.h> contains the htonl type stuff.. */

#include <asm/byteorder.h>

#endif     /* _LINUX_IN_H */
/* include/linux/in.h ************************************************************/
```

1.10 inet.h 头文件

该文件只是几个函数的简单声明。

```
/* include/linux/inet.h ************************************************************/
/*
 *        Swansea University Computer Society NET3
 *
 *    This work is derived from NET2Debugged, which is in turn derived
 *    from NET2D which was written by:
 *        Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
 *
 *        This work was derived from Ross Biro's inspirational work
 *        for the LINUX operating system.    His version numbers were:
 *
 *        $Id: Space.c,v         0.8.4.5    1992/12/12 19:25:04 bir7 Exp $
```

```
 *		$Id: arp.c,v		0.8.4.6	1993/01/28 22:30:00 bir7 Exp $
 *		$Id: arp.h,v		0.8.4.6	1993/01/28 22:30:00 bir7 Exp $
 *		$Id: dev.c,v		0.8.4.13 1993/01/23 18:00:11 bir7 Exp $
 *		$Id: dev.h,v		0.8.4.7	1993/01/23 18:00:11 bir7 Exp $
 *		$Id: eth.c,v		0.8.4.4	1993/01/22 23:21:38 bir7 Exp $
 *		$Id: eth.h,v		0.8.4.1	1992/11/10 00:17:18 bir7 Exp $
 *		$Id: icmp.c,v		0.8.4.9	1993/01/23 18:00:11 bir7 Exp $
 *		$Id: icmp.h,v		0.8.4.2	1992/11/15 14:55:30 bir7 Exp $
 *		$Id: ip.c,v		0.8.4.8	1992/12/12 19:25:04 bir7 Exp $
 *		$Id: ip.h,v		0.8.4.2	1993/01/23 18:00:11 bir7 Exp $
 *		$Id: loopback.c,v	0.8.4.8	1993/01/23 18:00:11 bir7 Exp $
 *		$Id: packet.c,v		0.8.4.7	1993/01/26 22:04:00 bir7 Exp $
 *		$Id: protocols.c,v 0.8.4.3	1992/11/15 14:55:30 bir7 Exp $
 *		$Id: raw.c,v		0.8.4.12 1993/01/26 22:04:00 bir7 Exp $
 *		$Id: sock.c,v		0.8.4.6	1993/01/28 22:30:00 bir7 Exp $
 *		$Id: sock.h,v		0.8.4.7	1993/01/26 22:04:00 bir7 Exp $
 *		$Id: tcp.c,v		0.8.4.16 1993/01/26 22:04:00 bir7 Exp $
 *		$Id: tcp.h,v		0.8.4.7	1993/01/22 22:58:08 bir7 Exp $
 *		$Id: timer.c,v		0.8.4.8	1993/01/23 18:00:11 bir7 Exp $
 *		$Id: timer.h,v		0.8.4.2	1993/01/23 18:00:11 bir7 Exp $
 *		$Id: udp.c,v		0.8.4.12 1993/01/26 22:04:00 bir7 Exp $
 *		$Id: udp.h,v		0.8.4.1	1992/11/10 00:17:18 bir7 Exp $
 *		$Id: we.c,v		0.8.4.10 1993/01/23 18:00:11 bir7 Exp $
 *		$Id: wereg.h,v		0.8.4.1	1992/11/10 00:17:18 bir7 Exp $
 *
 *		This program is free software; you can redistribute it and/or
 *		modify it under the terms of the GNU General Public License
 *		as published by the Free Software Foundation; either version
 *		2 of the License, or (at your option) any later version.
 */
#ifndef _LINUX_INET_H
#define _LINUX_INET_H

#ifdef __KERNEL__

extern void		inet_proto_init(struct net_proto *pro);
extern char		*in_ntoa(unsigned long in);
extern unsigned long	in_aton(char *str);

#endif
#endif	/* _LINUX_INET_H */
/* include/linux/inet.h ************************************************************/
```

其中 inet_proto_init 是 INET 域（目前是相对 UNIX 域而言）的初始化函数，定义在

net/inet/af_inet.c 文件中。in_ntoa, in_aton 这两个函数定义在 net/inet/utils.c 文件中，完成的功能是实现点分十进制 ASCII 码表示到 32 位整型数字之间的转换。

1.11 interrupt.h 头文件

将该头文件也列在此处，主要是因为网络代码部分在数据报接收时需要用到此文件中定义的“下半部分（Bottom Half）“结构及相关函数。“下半部分”是一种软件处理手段。为减少中断处理程序的执行时间，一般将中断处理程序人为的分为两个部分：“上半部分”和“下半部分”。“上半部分”主要完成一些比较紧急的工作，在完成这些工作后，其安排“下半部分”完成剩下一般性工作，虽然这些工作也同样重要，但并不要求实时，故可稍后完成。这样就大大减少了实际中断处理程序的执行时间，中断处理程序在处理完“上半部分”后即可返回。从而从整体上提高了系统性能。

“下半部分”在软件实现上是由一个 bh_struct 结构表示。该结构定义如文件中所示，有两个字段，一个是“下半部分”的执行函数，另一个是该函数的参数。系统目前定义有 32 个“下半部分”插槽。即系统中最多可同时存在 32 个下半部分。bh_active 用于表示正处于活跃状态的“下半部分”，当一个“下半部分”处于活跃状态时，即表示该“下半部分”需要执行。函数 mark_bh 用于改变某个“下半部分”的活跃状态。bh_mask 是掩码，即是我们通常所说的使能位，这是 32 位比特的整型数值，每位对应一个“下半部分”。当某位为 0 时，表示屏蔽该“下半部分”，此时即使该“下半部分”处于活跃状态，也不会被系统执行。enable_bh, disable_bh 这两个函数用于改变“下半部分”的使能位。bh_base 是 bh_struct 结构类型的数组，共有 32 个元素，每个元素对应一个系统“下半部分”。系统目前使用了前 8 个元素，如文件中 enum 定义。其中用于网络代码部分的“下半部分”是第五个位置所指向的元素即 bh_base[INET_BH]。而系统中（包括网络部分代码）所使用的大量定时器则是作为第一个“下半部分”元素执行的。

由此可见,虽然“下半部分”并不要求实时性，但一个活跃的“下半部分”必须得到执行，无论其延迟多长时间，因为其完成的功能是很重要的。系统中所有下半部分的执行均是在 do_bottom_half 函数中完成的。该函数定义在 kernel/softirq.c 文件中。由于该函数较短，下面给出该函数的实现。

```
/* kernel/softriq.c - do_bottom_half 函数*****************************************/
asmlinkage void do_bottom_half(void)
{
    unsigned long active;
    unsigned long mask, left;
    struct bh_struct *bh;

    bh = bh_base;
    active = bh_active & bh_mask;
    for (mask = 1, left = ~0 ; left & active ; bh++,mask += mask,left += left) {
        if (mask & active) {
            void (*fn)(void *);
            bh_active &= ~mask;
            fn = bh->routine;
            if (!fn)
```

```
                goto bad_bh;
            fn(bh->data);
        }
    }
    return;
bad_bh:
    printk ("irq.c:bad bottom half entry %08lx\n", mask);
}
/* kernel/softriq.c 代码摘录**********************************************************/
```

该函数本身的实现比较简单，但完成的功能却相当重要。其遍历 bh_base 这个“下半部分”数组，如果某个“下半部分”处于活跃状态并且相应使能位为 1，则执行该“下半部分”，只要其中之一不满足，则跳过该“下半部分”，暂时不执行。

```
/* include/linux/interrupt.h **********************************************************/
/* interrupt.h */
#ifndef _LINUX_INTERRUPT_H
#define _LINUX_INTERRUPT_H

#include <linux/linkage.h>
#include <asm/bitops.h>

struct bh_struct {
    void (*routine)(void *);
    void *data;
};

extern unsigned long bh_active;
extern unsigned long bh_mask;
extern struct bh_struct bh_base[32];

asmlinkage void do_bottom_half(void);

/* Who gets which entry in bh_base.  Things which will occur most often
   should come first - in which case NET should be up the top with SERIAL/TQUEUE! */

enum {
    TIMER_BH = 0,
    CONSOLE_BH,
    TQUEUE_BH,
    SERIAL_BH,
    NET_BH,
    IMMEDIATE_BH,
    KEYBOARD_BH,
```

```
	CYCLADES_BH
};

extern inline void mark_bh(int nr)
{
	set_bit(nr, &bh_active);
}

extern inline void disable_bh(int nr)
{
	clear_bit(nr, &bh_mask);
}

extern inline void enable_bh(int nr)
{
	set_bit(nr, &bh_mask);
}

/*
 * Autoprobing for irqs:
 *
 * probe_irq_on() and probe_irq_off() provide robust primitives
 * for accurate IRQ probing during kernel initialization.  They are
 * reasonably simple to use, are not "fooled" by spurious interrupts,
 * and, unlike other attempts at IRQ probing, they do not get hung on
 * stuck interrupts (such as unused PS2 mouse interfaces on ASUS boards).
 *
 * For reasonably foolproof probing, use them as follows:
 *
 * 1. clear and/or mask the device's internal interrupt.
 * 2. sti();
 * 3. irqs = probe_irq_on();	// "take over" all unassigned idle IRQs
 * 4. enable the device and cause it to trigger an interrupt.
 * 5. wait for the device to interrupt, using non-intrusive polling or a delay.
 * 6. irq = probe_irq_off(irqs);	// get IRQ number, 0=none, negative=multiple
 * 7. service the device to clear its pending interrupt.
 * 8. loop again if paranoia is required.
 *
 * probe_irq_on() returns a mask of snarfed irq's.
 *
 * probe_irq_off() takes the mask as a parameter,
 * and returns the irq number which occurred,
 * or zero if none occurred, or a negative irq number
 * if more than one irq occurred.
```

```
 */
extern unsigned int probe_irq_on(void);      /* returns 0 on failure */
extern int probe_irq_off(unsigned int); /* returns 0 or negative on failure */

#endif
/* include/linux/interrupt.h ************************************************************/
```

该文件除了对“下半部分”结构及相关重要变量和辅助函数进行了声明和定义之外，还另外声明了用于探测中断号的两个函数：probe_irq_on, probe_irq_off. 这两个函数的用法是：

1．清除设备的先前中断标志
2．调用 sti 函数使能系统中断
3．调用 probe_irq_on 函数，保存其返回的所有空闲中断号
4．设置设备使其产生一个中断
5．等待一段合理的时间，等待系统探测到该中断
6．调用 probe_irq_off 函数，并将之前由 probe_irq_on 返回的空闲中断号作为参数传替给 probe_irq_off 函数。
7．检查 probe_irq_off 函数的返回值：
   1>0—没有探测该设备的中断号
   对于此种情况可以在执行一次这个流程，并适当延长第 5 步中等待时间。
   2>负数—检测到多个中断号：根据具体情况分析原因
   3>正数—该中断号可用

1.12 ip.h 头文件

IP 是 Internet Protocol 的简称，意为网际协议。该协议在 TCP/IP 协议簇中占据重要地位。IP 是一种不可靠的无连接的数据报协议，其尽最大努力传输数据报。IP 本身不提供跟踪或差错检查。当可靠性很重要时，IP 必须与可靠的上层协议（如 TCP）配合使用。图 1-14 显示了 IP 首部格式。

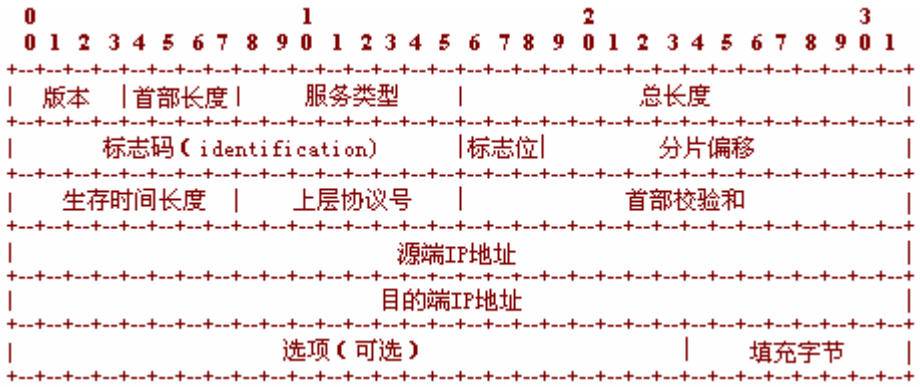


图 1-14 IP 首部格式

版本号：4 比特

这个字段定义了 IP 协议的版本，目前使用较多的版本是 4，但在将来版本 6 会取代版本 4。本文只讨论版本 4。

首部长度：4 比特

该字段以 4 字节为单位。故首部长度最大可达 15*4=60 字节。
其中首部固定（或者说是最小）长度是 20 字节，其它 40 字节用于各种 IP 选项，IP 选项是可选的。

服务类型：8 比特
下图显示了该字段的位分配。

图 1-15 服务类型

优先（precedence）是 3 位字段，它的值从 0（000B）到（111B）。
111—Network Control
110—Internetwork Control
101—CRITIC/ECP
100—Flash Override
011—Flash
010—Immediate
001—Priority
000—Routine
优先字段定义了在出现一些问题时（如拥塞）数据报的优先级。当路由器出现拥塞而必须丢弃一些数据报时，具有低优先级的数据报将首先被丢弃。

总长度：16 比特
该字段长度值包括 IP 首部和 IP 数据，该字段定义了包括首部在内的数据报总长度。
即：IP 数据长度=总长度-首部长度

标志码：16 比特
该字段用于数据报分片，当数据报长度大于 MTU 时，该数据报在源端或中间路由器上会被分割成小的分片（注意仍然是数据报），这些分片陆续到达目的端后，需要被重新组合成原来的一个大的数据报。标志位字段用于将这些分片归类，具有相同标志位字段的分片同属于一个数据报，即只有具有相同标志位字段的分片才会组合在一起。

标志（控制）位：3 比特
下图给出了各控制位的具体意义。

图 1-16 控制位

Bit 0：保留，必须为 0
Bit 1：（DF） 0—如果数据报长度很大超过 MTU，可对其进行分片 1—不可进行分片
Bit 2：（MF） 0—这是最后一个分片 1—还有后续分片（这是第一个分片或者是中间分片）

分片偏移：13 比特
该字段表示该分片在原先整个大数据报中的位置。该字段以 8 字节为单位。第一个分片该字段值为 0。最后一个分片该字段值加上其数据长度为原先整个大数据报的长度值。

生存时间：8 比特
该字段在最初设计时是打算保持一个时间戳，由经过的每一个路由器将这个字段的值减 1。当该字段值变为 0 时，就丢弃这个数据报。但使用时间戳要求所有机器必须是同步的，而且需要知道数据报从前一个机器传送到本机的时间，操作性很差。故后来该字段就简单表示所经过的路由器数目。每一个路由器在处理一个需要转发的数据报时，首先将该字段值减 1，如果该字段值变为 0，则丢弃该数据报，并给该数据报的起始发送端发送一个 ICMP 错误报文。

上层协议字段：8 比特
该字段表示所使用上一层协议号，该字段可取值定义在 in.h 文件中。表 1-4 给出了常用值。

表 1-4 IP 首部上层协议字段取值

值	协议
1	ICMP
2	IGMP
4	IPIP
6	TCP
17	UDP
255	RAW

网络栈将根据该字段值将数据报传送给上层相应的处理函数。如对于 TCP 协议，处理函数为 tcp_rcv, UDP 对应的处理函数为 udp_rcv，等等。

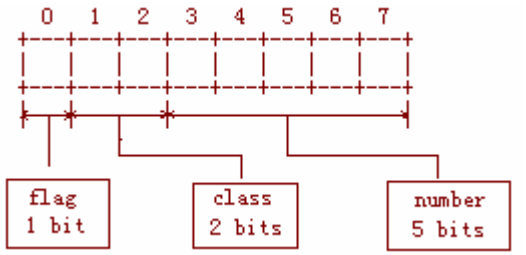
校验和：16 比特
该字段涵盖了范围仅仅为 IP 首部部分（包括 IP 选项，如有），不包括 IP 数据。

源端 IP 地址：32 比特
目的端 IP 地址：32 比特
这两个 32 位地址字段表示了该数据报的发送端 IP 地址和最终目的端 IP 地址。

IP 选贤：长度可变
IP 选项是可选的，也就是说一个 IP 数据报中可以包含也可以不包含 IP 选项。当包含 IP 选项时，可以有一个或多个选项同时存在。正因如此，该字段的长度是不固定的。

IP 选项可以分为两大类：
1>单个字节选项，这种 IP 选项只有一个类型字节。
2>多字节选项，这种选项的结构如下：一个类型字节；一个长度字节；多个数据字节。

类型字节具有内部结构：
图 1-17 给出了其内部结构。



类型字节被分为三个子域：
1>1 比特标志域：是否复制标志位
如该位为 0，则表示对应的选项在数据报进行分片时不用复制到每个分片中，只需复制到第一个分片即可。
如该位为 1，则表示对应的选项必须复制到所有的分片中。

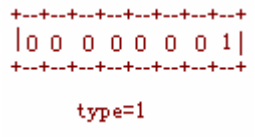
2>2 比特类域：
该子域表示选项所属的类。可取如下值：
0— 控制类
1— 保留将来使用
2— 调试，测试之用
3— 保留将来使用

3>5 比特 option number 子域：
该子域类子域进行细分，与类子域组成具体的选项类型。

具体选项类型：
1. 结束选项（End of Option List）

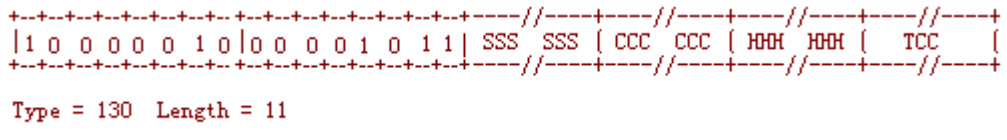
该选项表示所有选项的结束。系统在解析 IP 选项时遇到该选项后则停止解析，因为其后已经不存在其它选项。故该选项使用地点应是在其它所有选项之后。注意不同选项类型之间的填充不可使用该选项，而是使用无操作（No Operation）选项。

2. 无操作选项（No Operation）



该选项主要用于其它选项之间的填充。这种填充是为了使得某些选项位于 32-bit 边界上。

3. 安全级别选项（Security）



该选项使用较少，其中 S 子域，C 子域，H 子域分别为 16 比特，TCC 子域为 24 比特。这些子域所代表的含义为：

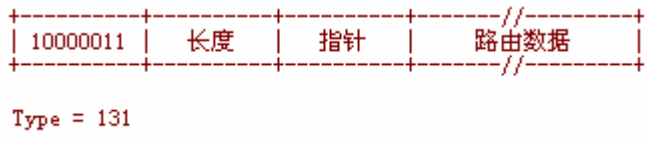
S—安全级别（Security）

C—信息划分（Compartment）

H—限制性处理（Handling Restrictions）

TCC—传输控制码（Transmission Control Code）

4. 松源地址和记录路由（Loose Source and Record Route）



该选项提供了一种机制让数据报发送源端提供路由信息，这些信息可以被网关使用以转发数据报，同时记录下该网关的路由信息。

选项起始于一个类型字节，值为 131。第二个字节为长度，该长度包括类型字节，其本身，指针字节以及其后的路由数据部分。第三个字节为指针字节，该指针指向路由数据第一个字节位置，位置的计算是从类型字节开始的，且起始计算从 1 开始，即如果路由数据在最初没有被使用时，指针字节字段的值应为 4，这也是指针字段最小的合理值。

路由数据字段包括一系列 IP 地址，如果指针字段的值大于长度字段值，则表示无路由数据。例如长度字段值为 3 时，指针字段值取最小值即 4，此时即表示无路由数据。

如果数据报到达 IP 首部中目的端地址所表示的路由器（主机）后，且此时指针字段值小于长度字段值，表示存在有效的路由数据，则将指针指向的 IP 地址填入 IP 首部中目的端地址字段，并且将该路由器的出站设备的 IP 地址覆盖指针指向的值（这个值已被使用）。之后指针值前移 4 字节指向下一个地址。图 1-18 显示了地址之间的交换关系。

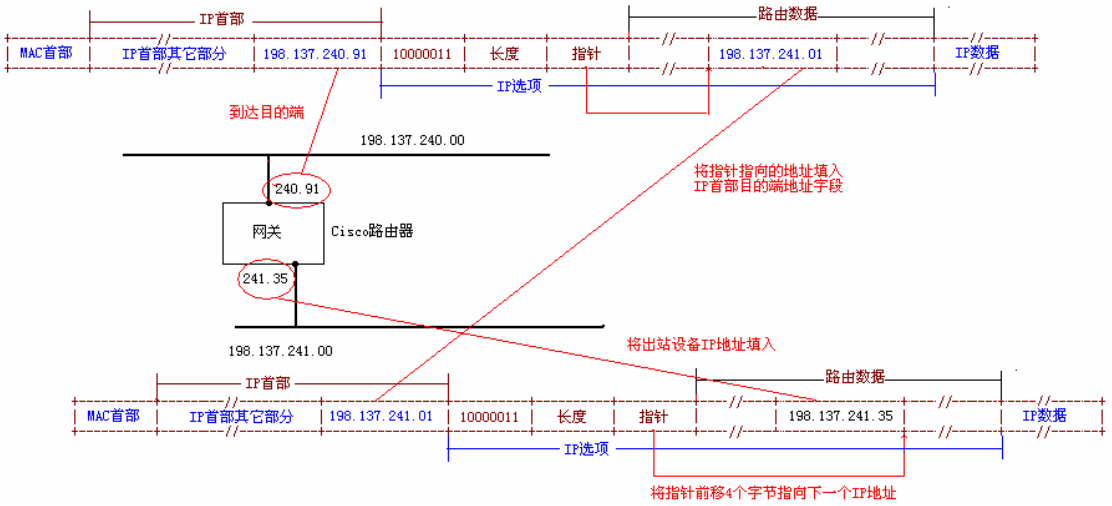
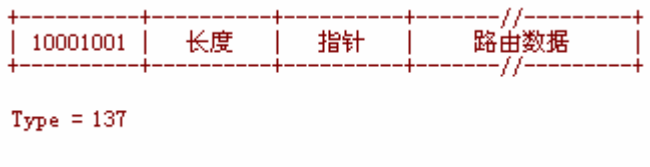


图 1-18 松源地址和记录路由工作方式

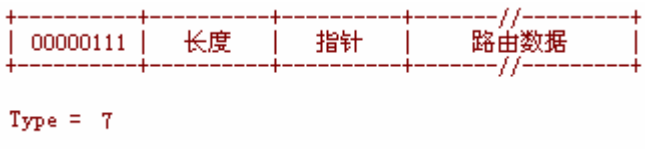
之所以称为松源地址和记录路由是相对于紧源地址和记录路由选项而言。对于松源地址和记录路由选项而言，在选项中路由数据指定的地址所对应的网关之间可以允许经过多个其它网关，而对于紧源地址和记录路由选项而言，则经过的网关则必须是选项中指定的地址所对应的网关，中间不可经过其它网关。

5. 紧源地址和记录路由（Strict Source and Record Route）



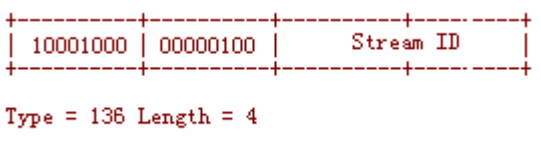
紧源地址和记录路由选项格式与松源地址和记录路由选项相同，二者之间工作方式上存在差别。这种差别前文中业已交待。

6. 记录路由选项（Record Route）



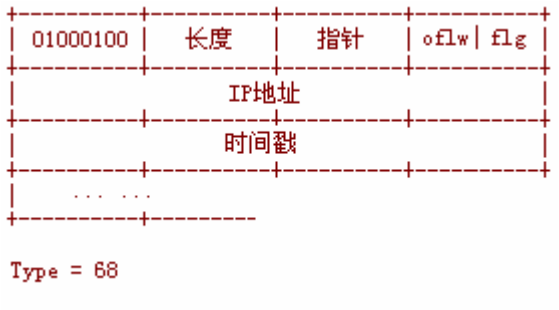
该选项前面部分格式同于松源地址和记录路由选项，路由数据部分格式不同。对于该选项而言，路由数据部分起初并无数据，而是预留的空间供中间路由器填写其转发数据报的出站设备的 IP 地址。这样就可以由此得知该数据报的传输路径。

7. 流标志符（Stream Identifier）



对于不支持流工作方式的网络，该选项提供了一种机制通过传送 16 比特流标志符来模拟流工作方式。

8. 时间戳选项



oflw 字段表示由于缺少空间而无法记录时间戳的主机数目，注意如果经过的主机过多，该字段也会溢出，因为其只有 4 比特，最多可表示 15 个主机。

flg 字段决定了该选项的具体工作方式。该字段占据 4 比特，可取如下值：

0— 只记录时间戳，每个时间戳按 4 字节连续排列，最多可有 37/4=9 个时间戳。

1— 同时记录 IP 地址和时间戳，即当一个转发该数据报时，在当前指针位置所指处填入数据报出站设备的 IP 地址，然后指针前移 4 个字节，紧接着填入转发的时间。这样每经过一个网关，就使用掉 8 个字节。由于 IP 选项最多可有 40 字节，除去类型字段，长度字段，指针字段共 3 个字节，可用字节数为 37 个字节，故最多可记录 37/8=4 个网关信息。

3—与 1 类型类似，但是 IP 地址是数据报原始发送端预先填入的。但转发数据报的该网关 IP 地址与指针所指向的源端预先填入的 IP 地址相同时，则在其后填入转发该数据报的时间戳。由此可见，原始发送端在预先填写 IP 地址时，在每个 IP 地址后都预留有 4 个字节的时间戳空间，以便被指定 IP 地址的网关填入，即 IP 地址是以 4 个字节进行间隔的。此种方式下最多可记录 4 个网关转发时间信息。

9. 填充字节

填充字节是为保证 IP 首部（包括选项部分）在 32 比特边界上结束。填充字节一般取值为 0。

```
/* include/linux/ip.h ************************************************************/
/*
 * INET         An implementation of the TCP/IP protocol suite for the LINUX
 *              operating system.  INET is implemented using the  BSD Socket
 *              interface as the means of communication with the user level.
 *
 *              Definitions for the IP protocol.
 *
 * Version:     @(#)ip.h 1.0.2       04/28/93
 *
 * Authors:     Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
 *
 *              This program is free software; you can redistribute it and/or
 *              modify it under the terms of the GNU General Public License
 *              as published by the Free Software Foundation; either version
```

```
 *              2 of the License, or (at your option) any later version.
 */
#ifndef _LINUX_IP_H
#define _LINUX_IP_H
#include <asm/byteorder.h>

//定义具体选项类型字段值，参考前文中对各选项的介绍。
#define IPOPT_END       0
#define IPOPT_NOOP      1
#define IPOPT_SEC       130 //Security
#define IPOPT_LSRR      131
#define IPOPT_SSRR      137
#define IPOPT_RR        7
#define IPOPT_SID       136     //stream identifier
#define IPOPT_TIMESTAMP 68


#define MAXTTL          255
//时间戳选项
struct timestamp {
 __u8       len; //选项长度
 __u8       ptr; //指针
 union {
#if defined(LITTLE_ENDIAN_BITFIELD)
 __u8       flags:4, //标志字段，该字段指定了的具体的时间戳类型，参考前文所述。
        overflow:4; //溢出字段
#elif defined(BIG_ENDIAN_BITFIELD)
 __u8       overflow:4,
        flags:4;
#else
#error      "Please fix <asm/byteorder.h>"
#endif
 __u8       full_char;
 } x;
 __u32      data[9]; //在只记录时间戳的情况下，最多可有 9 个时间戳缓存空间。
};


#define MAX_ROUTE   16

struct route {
    char            route_size; //选项长度字段
    char            pointer; //指针字段
    unsigned long route[MAX_ROUTE]; //路由数据，虽然 MAX_ROUTE 定义为 16，但实
```

```
                                        //际可用的路由数据空间仅为 37 个字节。
};

//当接收一个具有IP选项的数据报时，内核将根据数据报中具体的IP选项初始化该结构，
//以便内核其它代码的处理。
struct options {
  struct route        record_route;
  struct route        loose_route;
  struct route        strict_route;
  struct timestamp    tstamp;
  unsigned short      security;
  unsigned short      compartment;
  unsigned short      handling; //handling restriction
  unsigned short      stream; //id stream option
  unsigned            tcc; //transmission control code
};

//IP 首部定义，参考前文内容不难理解。
struct iphdr {
#if defined(LITTLE_ENDIAN_BITFIELD)
 __u8       ihl:4,
       version:4;
#elif defined (BIG_ENDIAN_BITFIELD)
 __u8       version:4,
            ihl:4;
#else
#error      "Please fix <asm/byteorder.h>"
#endif
 __u8       tos;
 __u16      tot_len;
 __u16      id;
 __u16      frag_off;
 __u8       ttl;
 __u8       protocol;
 __u16      check;
 __u32      saddr;
 __u32      daddr;
 /*The options start here. */
};


#endif      /* _LINUX_IP_H */
/* include/linux/ip.h ************************************************************/
```

1.13 ip_fw.h 头文件

该文件定义 IP 层防火墙相关数据结构。该版本网络代码防火墙部分实现较为简单，但可作为了解防火墙工作原理的基础。可以将防火墙看作为一个过滤器，其过滤进出主机的网络数据包，根据防火墙预设的规则，对这些数据包进行取舍。防火墙简单的说就是一组规则，内核网络代码对接收和发送的每个数据包都是用这套规则进行检查，如发现某个数据包满足其中的规则，则根据该规则的具体规定的措施决定该数据包的去向：有可能被简单的丢弃，也有可能发送一个 ICMP 数据包给远端，或者继续正常的处理。

软件实现上，每个规则都由一个 ip_fw 数据结构表示，每个 ip_fw 数据结构表示一个规则，将所有同类型（如转发规则集合）组合在一起形成一个链表，便形成了我们通常意义上的防火墙规则群。规则的检查就是遍历该链表，根据数据包的相应的头部信息（如 IP 首部或者 TCP，UDP 首部）进行判断该数据包是否匹配其中的规则。这些信息包括数据包发送的源端 IP 地址，源端端口号，目的端 IP 地址，目的端端口号等。如果数据包的目的端就是本机，可能这些规则中还包括数据包的接收接口设备。

ip_fw.h 文件如下，其第一部分即对 ip_fw 数据结构的定义，诚如上文所述，每个 ip_fw 结构代表一个防火墙规则。ip_fw 结构的字段：

1>fw_next：指向 ip_fw 结构的指针类型，该指针的作用是将所有的规则连接成一个链表的形式，易于管理和规则检查。

2>fw_src, fw_dst：发送端 IP 地址和接收端 IP 地址，用于筛选满足条件的出入站数据包。

3>fw_smsk, fw_dmsk：发送端和接收端 IP 地址掩码，规则检查中用于查看 IP 地址网络部分。

4>fw_via：对于目的端是本机的数据包，该字段指定数据包的接收接口设备 IP 地址。

5>fw_flg：标志位，该字段一方面指定该规则适用的协议范围，另一方面作为控制字段决定对数据包的具体处理方式。

该字段的可取如下值，这些值定义在在 ip_fw 结构之后。

IP_FW_F_ALL：该规则适用于所有协议类型。

IP_FW_F_TCP：该规则仅针对 TCP 协议。

IP_FW_F_UDP：该规则仅针对 UDP 协议。

IP_FW_F_ICMP：该规则仅针对 ICMP 协议。

IP_FW_F_ACCEPT：满足该规则的数据包的处理方式是放行。

IP_FW_F_SRNG：规则指定的发送端端口值定义在一个范围中，而非具体值（但也可包括）。

IP_FW_F_DRNG：规则指定的目的端端口值定义在一个范围中，而非具体值（但也可包括）。

IP_FW_F_PRN：该标志用于调试，即发现一个满足此规则的数据包时，打印出相关信息。

IP_FW_F_BIDIR：该规则同时适用于出站和进站数据包，默认情况下，仅对进站数据包。

IP_FW_F_TCPSYN：该规则仅针对于使用 TCP 协议的 SYN 数据包。SYN 数据包是 TCP 协议建立连接过程中同步数据包。

IP_FW_F_ICMPRPL：该规则表示当数据包被丢弃需发送一个 ICMP 目的端不可达错误类型给该数据报的原始发送端。

6>fw_nsp, fw_ndp：规则定义中指定的发送端和接收端端口数目。

7>fw_pts：short 类型的数组，每个数组元素代表一个端口号，该数组的前 fw_nsp 个元素表示发送端端口，紧接着之后的 fw_ndp 个元素表示接收端端口。注意 fw_nsp 与 fw_ndp 的之和不可大于 IP_FW_MAX_PORTS，即数组的容量大小。

8>fw_pcnt, fw_bcnt：这两个字段用于信息统计，表示满足该规则的数据包的个数，即这些数据包中包含的数据字节数。

防火墙规则目前定义有三个链表：

A. 阻塞（block）规则链表 ip_fw_blk_chain

注意称之为阻塞规则链表并非是指凡是满足该链表中其中之一规则的数据包都要被丢弃，而是指该链表中规则仅对本机原始发送的数据包以及最终目的端是本机的数据包进行检查。至于数据包的取舍会根据其满足的具体规则的规定进行相应处理，如果该规则没有定义如何处理满足该条规则的数据包，则根据全局变量 ip_fw_blk_policy 的值进行取舍。

B. 转发（forward）规则链表 ip_fw_fwd_chain

该链表中规则适用于每个需经过本机转发的数据包。数据包的取舍根据满足的规则规定的方式进行，如果该规则没有规定如何处理数据包，则根据全局变量 ip_fw_fwd_policy 的值进行取舍。

C. 信息统计（accouting）规则链表 ip_acct_chain

该链表中的规则用于检查所有类型数据包（无论是转发的还是本机原始发送的或者最终目的地是本机的），并对该数据包满足的规则进行信息更新。主要是对 ip_fw 结构中 fw_pcnt, fw_bcnt 字段的更新。注意此链表中的规则仅用于信息更新，并不对数据包的去向进行任何决定，也即该链表中规则没有定义数据包的取舍策略，内核网络代码也不会检查这些策略。而是在更新信息之后直接返回了。

用于选项设置和获取的标志位：

IP_FW_ADD_BLK：添加一个阻塞规则。

IP_FW_ADD_FWD：添加一个转发规则。

IP_FW_CHK_BLK：检查阻塞规则链表中规则的有效性。

IP_FW_CHK_FWD：检查转发规则链表中规则的有效性。

IP_FW_DEL_BLK：从阻塞规则链表中删除一个规则。

IP_FW_DEL_FWD：从转发规则链表中删除一个规则。

IP_FW_FLUSH_BLK：删除阻塞规则链表中的所有规则。

IP_FW_FLUSH_FWD：删除转发规则链表中的所有规则。

IP_FW_ZERO_BLK：清除阻塞规则链表中所有规则的统计信息。

IP_FW_ZERO_FWD：清除转发规则链表中所有规则的统计信息。

IP_FW_POLICY_BLK：设置阻塞链表中所有规则的默认数据包处理策略，但某个具体规没有定义对满足该规则的数据包的处理策略时，就使用此处设置的策略。

IP_FW_POLICY_FWD：设置转发链表中所有规则的默认数据包处理策略，具体执行同阻塞链表。

IP_ACCT_ADD：添加一个信息统计规则。

IP_ACCT_DEL：删除一个信息统计规则。

IP_ACCT_FLUSH：删除信息统计规则链表中的所有规则。

IP_ACCT_ZERO：清除信息统计规则链表中所有规则的统计信息。

定义 ip_fwpkt 结构便于处理规则检查。

文件最后是对一些重要全局变量和函数的声明，这些变量和函数均定义在 net/inet/ip_fw.c 文件中。这些全局变量的含义前文中已有论说，具体函数的讨论在下文介绍 ip_fw.c 文件时给出。

```
/* include/linux/ip_fw.h ************************************************************/
/*
 *    IP firewalling code. This is taken from 4.4BSD. Please note the
 *    copyright message below. As per the GPL it must be maintained
 *    and the licenses thus do not conflict. While this port is subject
 *    to the GPL I also place my modifications under the original
 *    license in recognition of the original copyright.
 *
 *    Ported from BSD to Linux,
 *          Alan Cox 22/Nov/1994.
 *    Merged and included the FreeBSD-Current changes at Ugen's request
 *    (but hey it's a lot cleaner now). Ugen would prefer in some ways
 *    we waited for his final product but since Linux 1.2.0 is about to
 *    appear it's not practical - Read: It works, it's not clean but please
 *    don't consider it to be his standard of finished work.
 *          Alan.
 *
 *    All the real work was done by .....
 */

/*
 * Copyright (c) 1993 Daniel Boulet
 * Copyright (c) 1994 Ugen J.S.Antsilevich
 *
 * Redistribution and use in source forms, with and without modification,
 * are permitted provided that this entire comment appears intact.
 *
 * Redistribution in binary form may occur without any restrictions.
 * Obviously, it would be nice if you gave credit where credit is due
 * but requiring it would be too onerous.
 *
 * This software is provided ``AS IS'' without any warranties of any kind.
 */

/*
 *    Format of an IP firewall descriptor
 *
 *    src, dst, src_mask, dst_mask are always stored in network byte order.
 *    flags and num_*_ports are stored in host byte order (of course).
 *    Port numbers are stored in HOST byte order.
```

```
 */

#ifndef _IP_FW_H
#define _IP_FW_H

struct ip_fw
{
      struct ip_fw    *fw_next;             /* Next firewall on chain */
      struct in_addr fw_src, fw_dst;       /* Source and destination IP addr */
      struct in_addr fw_smsk, fw_dmsk;/* Mask for src and dest IP addr */
      struct in_addr fw_via;               /* IP address of interface "via" */
      unsigned short fw_flg;               /* Flags word */
      /* N'of src ports and # of dst ports
        * in ports array (dst ports follow
     * src ports; max of 10 ports in all;
     * count of 0 means match all ports)
     */
   /* fw_nsp, fw_ndp 分别表示本地端口数量和远端端口数量，
     *  端口号在 fw_pts 数组中，其中本地端口占据数组前面元素空间，远端端口占据
     *  数组的后面元素空间。注意通常并非各分一半。
     */
      unsigned short fw_nsp, fw_ndp;
#define IP_FW_MAX_PORTS     10                      /* A reasonable maximum */
      unsigned short fw_pts[IP_FW_MAX_PORTS]; /* Array of port numbers to match */
      unsigned long    fw_pcnt,fw_bcnt;        /* Packet and byte counters */
};

/*
 *    Values for "flags" field .
 */
//ip_fw 结构中 fw_flg 字段的可取值。
#define IP_FW_F_ALL  0x000     /* This is a universal packet firewall*/
#define IP_FW_F_TCP  0x001     /* This is a TCP packet firewall          */
#define IP_FW_F_UDP 0x002      /* This is a UDP packet firewall          */
#define IP_FW_F_ICMP       0x003      /* This is a ICMP packet firewall        */
#define IP_FW_F_KIND       0x003      /* Mask to isolate firewall kind         */
#define IP_FW_F_ACCEPT  0x004      /* This is an accept firewall (as        *
                        *               opposed to a deny firewall)*
                        *                                                    */
#define IP_FW_F_SRNG       0x008      /* The first two src ports are a min    *
                        * and max range (stored in host byte *
                        * order).                                    *
                        *                                                    */
#define IP_FW_F_DRNG       0x010      /* The first two dst ports are a min    *
```

```
                        * and max range (stored in host byte *
                        * order).                              *
                        * (ports[0] <= port <= ports[1])       *
                        *                                      */
#define IP_FW_F_PRN 0x020     /* In verbose mode print this firewall*/
#define IP_FW_F_BIDIR     0x040     /* For bidirectional firewalls          */
#define IP_FW_F_TCPSYN  0x080     /* For tcp packets-check SYN only       */
#define IP_FW_F_ICMPRPL 0x100     /* Send back icmp unreachable packet    */
#define IP_FW_F_MASK      0x1FF     /* All possible flag bits mask          */

/*
 *    New IP firewall options for [gs]etsockopt at the RAW IP level.
 *    Unlike BSD Linux inherits IP options so you don't have to use
 *    a raw socket for this. Instead we check rights in the calls.
 */

#define IP_FW_BASE_CTL     64

#define IP_FW_ADD_BLK       (IP_FW_BASE_CTL)
#define IP_FW_ADD_FWD       (IP_FW_BASE_CTL+1)
#define IP_FW_CHK_BLK       (IP_FW_BASE_CTL+2)
#define IP_FW_CHK_FWD       (IP_FW_BASE_CTL+3)
#define IP_FW_DEL_BLK       (IP_FW_BASE_CTL+4)
#define IP_FW_DEL_FWD       (IP_FW_BASE_CTL+5)
#define IP_FW_FLUSH_BLK    (IP_FW_BASE_CTL+6)
#define IP_FW_FLUSH_FWD    (IP_FW_BASE_CTL+7)
#define IP_FW_ZERO_BLK      (IP_FW_BASE_CTL+8)
#define IP_FW_ZERO_FWD      (IP_FW_BASE_CTL+9)
#define IP_FW_POLICY_BLK (IP_FW_BASE_CTL+10)
#define IP_FW_POLICY_FWD (IP_FW_BASE_CTL+11)

#define IP_ACCT_ADD          (IP_FW_BASE_CTL+16)
#define IP_ACCT_DEL          (IP_FW_BASE_CTL+17)
#define IP_ACCT_FLUSH       (IP_FW_BASE_CTL+18)
#define IP_ACCT_ZERO        (IP_FW_BASE_CTL+19)

struct ip_fwpkt
{
      struct iphdr fwp_iph;               /* IP header */
      union {
            struct tcphdr fwp_tcph;       /* TCP header or */
            struct udphdr fwp_udph;       /* UDP header */
      } fwp_protoh;
      struct in_addr fwp_via;             /* interface address */
```

```
};

/*
 *    Main firewall chains definitions and global var's definitions.
 */

#ifdef __KERNEL__

#include <linux/config.h>

#ifdef CONFIG_IP_FIREWALL
extern struct ip_fw *ip_fw_blk_chain;
extern struct ip_fw *ip_fw_fwd_chain;
extern int ip_fw_blk_policy;
extern int ip_fw_fwd_policy;
extern int ip_fw_ctl(int, void *, int);
#endif

#ifdef CONFIG_IP_ACCT
extern struct ip_fw *ip_acct_chain;
extern void ip_acct_cnt(struct iphdr *, struct device *, struct ip_fw *);
extern int ip_acct_ctl(int, void *, int);
#endif
extern int ip_fw_chk(struct iphdr *, struct device *rif,struct ip_fw *, int, int);
#endif /* KERNEL */

#endif /* _IP_FW_H */
/* include/linux/ip_fw.h ************************************************************/
```

1.14 ipx.h 头文件

该文件定义 IPX 协议使用的数据结构。

IPX 是 Internet Packet eXchange（网络报文交换协议）的缩写，其是一种无连接状态，报文服务协议。对于发送出去的每个数据包，都无需进行确认应答。

每个数据包都被独立的发送到远端。IPX 是 Xerox 网络标准（XNS：Xerox Network Standard）的一种实现。其它 Netware 协议的服务均建立在 IPX 协议之上，如服务声明（SAP），路由（RIP），Netware 核心协议（NCP）等等。

IPX 实现了 OSI 模型网络层次中寻址，路由，数据报交换等任务。IPX 尽最大努力传送数据包，但并不保证一定可以将数据包发送到远端。如果需要提供可靠的数据包传送，则使用 IPX 协议的上层协议必须注意到这一点。

IPX 报文首部需要提供源端和终端地址。IPX 地址模式有三个组成部分。如表 1-7 所示。

表 1-7 IPX 地址组成

地址组成	长度	说明
网络地址（Network）	4bytes	指定一个具体的网络
节点地址（Node）	6bytes	指定网络中的某个具体的节点
插口号（Socket number）	2bytes	指定节点上运行的某个具体的进程

在 IPX 网络中，每个局域网都被赋予一个网络地址。网络地址被路由器用于转发数据包。节点地址指定了局域网中一个工作站或者某个计算机。对于客户端而言，节点地址一般是指网络接口设备的出厂硬件地址。
插口地址则指定了在某个具体节点上运行的进程，这也是数据报的最终目的的。每个进程都被赋予一个插口地址。进程可以使用众所周知的插口地址，也可以使用临时分配的插口地址。插口地址是一种在同一个主机上建立多个通信通道的机制。它的功能类似于其它网络协议中所使用的端口号。

IPX 协议首部格式如图 1-19 所示。

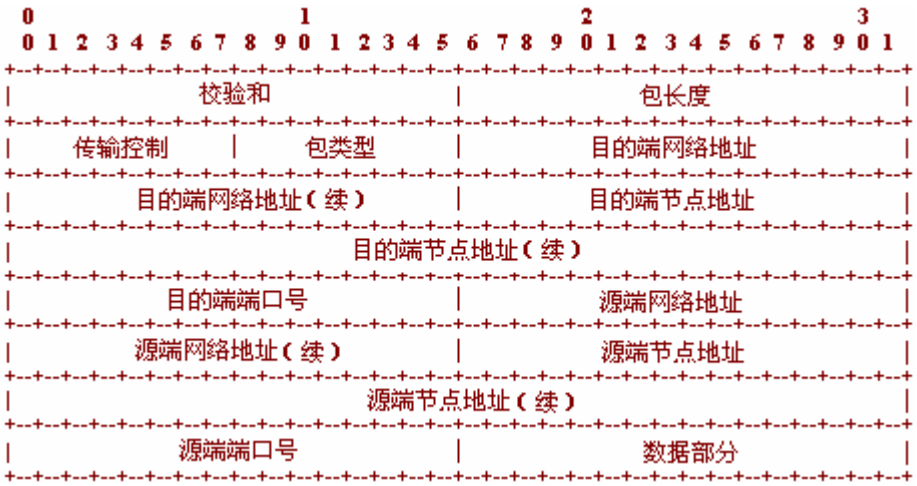


图 1-19 IPX 首部格式

IPX 首部共有 30 个字节，其中：
校验和：2 字节
该字段通常被设置为 0xFFFF，这表示无校验和的计算。

包长度：2 字节
该长度字段包括 IPX 首部及其数据部分总长度。

传输控制：1 字节
该字段表示数据包经过的路由器数目。在发送端该字段被初始化为 0，中间的每个路由器在转发该数据包时都回递增该字段值。如果该字段值增加到 16，即在第 16 个路由器上该数据包将被认为无法到达目的端，将被丢弃。

包类型：1 字节
包类型字段表示包所要求或者提供的服务。该字段可取值如表 1-8 所示。

表 1-8 包类型字段值及其意义

包类型	说明
0x00	未知（SAP 或者 RIP）
0x01	路由信息包类型（RIP：Routing Information Packet）
0x02	回复包类型（Echo Packet）
0x03	错误包类型（Error Packet）
0x04	报文交换协议（PEP：Packet Exchange Protocol）
0x05	分组序列交换协议（SPX：Sequenced Packet eXchange）
0x10 -- 0x1F	在试验协议（Experimental Protocols）
0x11	Netware 核心协议（NCP：Netware Core Protocol）
0x14	NetBIOS 广播

目的端网络地址：4 字节
数据包发送时，该字段必须被设置为目的主机所在网络的网络地址。如果该字段初始化为 0，表示目的网络即本发送端所在的局域网。

目的端节点地址：6 字节
该字段表示某个局域网中某台主机的网络接口设备的硬件地址，这个地址在设备出厂时都已设定。节点地址为 0xFFFFFF 表示对由网络地址部分指定的网络上的所有主机进行广播。

目的端端口号：2 字节
该字段主要是用于分别同一目的主机上多个进程。每个进程在使用 IPX 协议进行通信时，都会被赋予一个端口号，从而实现同一主机上多个进程的同时通信而不互相干扰，这就是通常所说的多路分用。
一些运行在 IPX 协议之上的服务通常具有固定的端口号。这样使得客户端可以更好的跟踪并取得服务。

如下端口号被 IPX 协议保留。(以下列表中为避免中文翻译引起歧义，故直接以英文给出)

端口号	说明
0x02	Echo protocol socket
0x03	Error handler packet

另外 Novell 为特殊目的定义并保留了如下端口号（不全）：

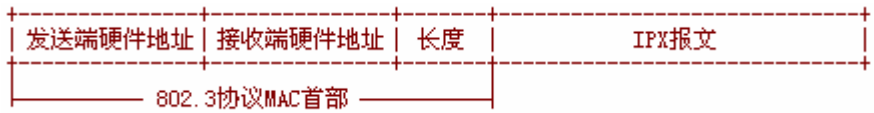
端口号	服务
0x0247	Novell VirtualTerminal (NVT) server
0x0451	Netware Core Protocol (NCP)
0x0452	Netware Service Advertising Protocol (SAP)
0x0453	Netware Routing Protocol (RIP)
0x0456	Netware Diagnostics Protocol (NDP)
0x8063	NVT2 server

0x811E Print server
源端网络地址：4 字节
该字段设置为发送端主机所在网络的网络地址。

源端节点地址：6 字节
该字段应设置为发送端主机自身节点地址（其发送接口设备的硬件地址）。

源端端口号：2 字节
如果发送端与远端只有一个通信通道，则该字段可由 IPX 协议内核代码进行初始化，否则必须由上层应用程序进行初始化操作。

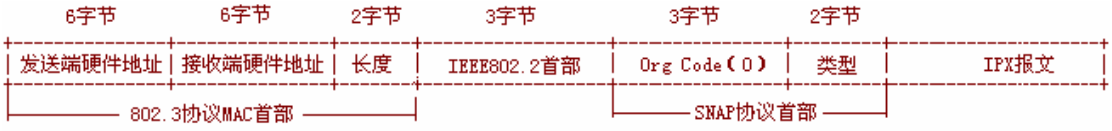
IPX 协议封装格式
1>802.3



IPX 首部紧接在 MAC 首部长度字段之后。
2>IEEE802.2

IPX 首部紧接在 IEEE802.2 首部之后（DSAP，SSAP，Control 字段之后）。
3>Ethernet II

IPX 首部紧接在 MAC 首部类型字段之后，对于 IPX 协议，MAC 首部类型字段值为 0x8137。
4>Ethernet SNAP



IPX 首部紧接在 SNAP 协议 5 个字节的首部之后。

有关 ipx.h 文件说明请参见以下文件内部注释。

```
/*include/linux/ipx.h ************************************************************/
#ifndef _IPX_H_
#define _IPX_H_
#include <linux/sockios.h>
//节点地址长度
```

```
#define IPX_NODE_LEN    6
#define IPX_MTU         576
//IPX 协议接口地址定义，类似于 sockaddr_in, sockaddr_un 等结构。
//注意该结构为 16 字节
struct sockaddr_ipx
{
    short sipx_family; //AF_IPX
    short sipx_port; //端口号
    unsigned long   sipx_network; //网络地址
    unsigned char sipx_node[IPX_NODE_LEN]; //节点地址
    unsigned char  sipx_type;
    unsigned char  sipx_zero;       /* 16 byte fill */
};

/*
 *  So we can fit the extra info for SIOCSIFADDR into the address nicely
 */

#define sipx_special    sipx_port
#define sipx_action sipx_zero
#define IPX_DLTITF    0
#define IPX_CRTITF    1

//用于设置 IPX 协议的路由表项，注意路由表项并非由此结构表示，而是由 ipx_route 结构
//代表一个 IPX 协议路由表项。ipx_route_definition 结构中的信息只是用于创建一个 ipx_route
//结构。这是一个用于信息传替的中间结构。
typedef struct ipx_route_definition
{
    unsigned long ipx_network;
    unsigned long ipx_router_network;
    unsigned char ipx_router_node[IPX_NODE_LEN];
}   ipx_route_definition;

//如同 ipx_route_definition，此结构用于设置或获取主机接口设备信息。在使用 IPX 协议进
//行通信的主机上，每个网络接口均由一个 ipx_interface 结构表示，ipx_route_definition 结构
//的作用也是相当于一个信息传替的中间结构。
typedef struct ipx_interface_definition
{
    unsigned long ipx_network;
    unsigned char ipx_device[16];
    //IPX 协议链路层封装类型，取值如下定义的常量。
    //有关内容参考前文中 IPX 协议封装格式一节。
    unsigned char ipx_dlink_type;
#define IPX_FRAME_NONE          0
```

```
#define IPX_FRAME_SNAP      1
#define IPX_FRAME_8022      2
#define IPX_FRAME_ETHERII   3
#define IPX_FRAME_8023      4
    unsigned char ipx_special; //该字段取值有如下三个常量。
#define IPX_SPECIAL_NONE    0
#define IPX_PRIMARY     1
#define IPX_INTERNAL        2
    unsigned char ipx_node[IPX_NODE_LEN];
}   ipx_interface_definition;

//ipx_config_data 结构完全是一个用于设置或获取内部变量信息的过渡结构。用于 ioctrl 控制
//函数中。
//ipxcfg_auto_select_primary 字段用于设置内核变量 ipxcfg_auto_select_primary 的值，
//同理 ipxcfg_auto_create_interfaces 字段用于设置内核变量 ipxcfg_auto_create_interfaces 的取
//值。这两个内核变量的意义将第二章中对 ipx.c 文件的分析。
typedef struct ipx_config_data
{
    unsigned char  ipxcfg_auto_select_primary;
    unsigned char  ipxcfg_auto_create_interfaces;
}   ipx_config_data;
/*
 * OLD Route Definition for backward compatibility.
 */
//该结构作用同 ipx_route_definition，该结构是为保持向后兼容性而保留的结构。
struct ipx_route_def
{
    unsigned long ipx_network;
    unsigned long ipx_router_network;
#define IPX_ROUTE_NO_ROUTER 0
    unsigned char ipx_router_node[IPX_NODE_LEN];
    unsigned char ipx_device[16];
    unsigned short ipx_flags;
#define IPX_RT_SNAP     8
#define IPX_RT_8022     4
#define IPX_RT_BLUEBOOK         2
#define IPX_RT_ROUTED       1
};
//如下三个标志位用于 ipx_ioctrl 函数中进行相关信息获取和设置。
#define SIOCAIPXITFCRT      (SIOCPROTOPRIVATE)
#define SIOCAIPXPRISLT      (SIOCPROTOPRIVATE+1)
#define SIOCIPXCFGDATA      (SIOCPROTOPRIVATE+2)
#endif
/*include/linux/ipx.h ************************************************************/
```

1.15 net.h 头文件

该结构定义 INET 网络层使用的重要数据结构以及一些常量定义。其中常量定义此处不做解释，读者应该不难理解这些常量的意义。

该文件中定义的 INET 网络层所使用的重要数据结构有：

1>socket 结构

该结构在 INET 层表示一个网络套接字。

主要字段：

type：该套接字所用的流类型，可取值有 SOCK_RAW,SOCK_DGRAM,SOCK_STREAM, SOCK_SEQPACKET,SOCK_PACKET.

state：该套接字所处的连接状态，可取值有：

SS_FREE = 0：这是一个空闲的套接字，“空闲”意为还没有被分配使用。

SS_UNCONNECTED：该 socket 结构已被分配给一个连接，但连接尚未建立。

SS_CONNECTING：正在与远端取得连接（用于 TCP 等有实际连接操作的协议）。

SS_CONNECTED：已与远端取得连接，正在进行数据的正常传送。

SS_DISCONNECTING：正与远端取消连接。

flags：标志字段，该字段目前尚无明确作用。

ops：操作函数集指针，即 INET 层相应域对应的操作函数集，可取值为 unix_proto_ops(UNIX 域：本机进程之间模拟网络操作方式的一种数据交换形式)，inet_proto_ops（INET 域：使用 TCP/IP 协议）。

data：用于保存指向“私有”数据结构的指针。此处“私有”是对内核而言的，即对于不同的域内核使用该指针指向不同的数据结构，如对于 UNIX 域，data 指针指向一个 unix_proto_data 数据结构，而对于 INET 域，其指向 sock 数据结构。

conn，iconn：这两个字段用于 UNIX 域，分别指向建立完全连接的对方 socket 结构以及等待建立连接的 socket 结构。由于 UNIX 域用于本机进程之间，故 conn，iconn 用于指向对方进程所对应的 socket 结构方才有意义。对于 INET 域而言，这两个字段没有意义。

next：构成 socket 结构的链表。

wait：等待队列，当进程要求的操作一时无法满足时（如进程进行读数据操作，而接收队列无数据时，如果进程没有设置为 nonblock 读取形式，内核网络代码会自动安排该进程进入睡眠状态，即通常所说的阻塞读取），就在该队列上睡眠等待，当系统可满足进程的要求时，会唤醒该队列上的进程。

wait_queue 结构定义在 include/linux/wait.h 文件中：

```
struct wait_queue {
        struct task_struct * task;
        struct wait_queue * next;
};
```

inode：指向 inode 结构的指针。socket 结构中安排这个指针的目的在于使得网络套接字可以使用普通文件的读写函数进行数据的读取（read）和写入(write)。内核网络代码每当在新建一个网络套接字连接时（socket 系统调用），都会分配一个 inode 结构，socket 结构中的 inode 字段指向这个 inode 结构，而 inode 结构中的联合类型（union）字段 u 中之 socket_i 字段反向指向这个 socket 结构，另外还会分配一个 file 结构与 inode 结构对应，最后返回一个整型类型的文件描述符。之后用户通过该文件描述符进行套接字操作（如读写）时，系统由该文件描述符得到 file 结构，调用 file 结构 f_op 字段指向的相应的操作函数处理用户命令，而这个操作函数将根据 file 结构中 f_inode 字段指向的 inode 结构得到相应的 socket 结构(inode->u.socket_i)，接着调用 socket 结构中 ops 字段指向具体函数完成用户要求的命令。当然事情远远没有结束，这个逐渐调用的过程会继续进行下去，直到到达传输层才会根据具体的协议进行实质上的处理。

fasync_list：指向 fasync_struct 结构构成的链表。该结构用于同步文件的读写。fasync_struct 结构定义在 include/linux/fs.h 文件中。

```
struct fasync_struct {
    int     magic;
    struct  fasync_struct    *fa_next; /* singly linked list */
    struct file      *fa_file;
};
```

2>操作函数集数据结构 proto_ops

该结构类型作为 socket 结构中的一个字段代表了对应操作域的操作函数集合。该集合定义了网络套接字各种操作的函数指针。例如相应 INET 域对应的函数集合为（net/inet/af_inet.c）

```
static struct proto_ops inet_proto_ops = {
    AF_INET,
    inet_create,
    inet_dup,
    inet_release,
    inet_bind,
    inet_connect,
    inet_socketpair,
    inet_accept,
    inet_getname,
    inet_read,
    inet_write,
    inet_select,
    inet_ioctl,
    inet_listen,
    inet_send,
    inet_recv,
    inet_sendto,
    inet_recvfrom,
```

```
    inet_shutdown,
    inet_setsockopt,
    inet_getsockopt,
    inet_fcntl,
};
```

用户对网络套接字的所有操作都将经过这些函数的中间处理。例如用户使用 read 函数读取网络数据时，所经过的函数调用路径为：
read→sys_read→sock_read→**inet_read**→tcp_read

net_proto 结构用于域协议的定义。
其中 name 为域名称如“UNIX”，“INET”等。init_func 函数指针指向域初始化函数如对于 UNIX 域，其初始化函数为 unix_proto_init，而 INET 域为 inet_proto_init 等。目前定义的域如表 1-5 所示：

表 1-9 域协议

域名称	初始化函数	说明
“UNIX”	unix_proto_init	本机进程间模拟网络的数据一种交换形式
“INET”	inet_proto_ini	使用 TCP/IP 协议族的网络数据包交换协议
“802.2”	p8022_proto_init	使用 802.2 协议
“SNAP”	snap_proto_init	使用 SNAP（Subnetwork Access Protocol）协议
“AX.25”	ipx_proto_init	
“IPX”	ipx_proto_init	使用 IPX 网络层协议进行数据包交换（Novell 网络协议），
“DDP”	atalk_proto_init	AppleTalk

IEEE802 规范对局域网数据包格式定义了一组标准，以 OSI 模型为例，这组标准规定了物理层，链路层的数据包首部格式。其中 802.3，802.4，802.5 为物理层标准，802.2 为链路层标准。802.2 链路层标准定义了 OSI 链路层中 LLC（Logic Link Control：逻辑链路控制）子层。802 标准首部格式如下图（图 1-20）所示：

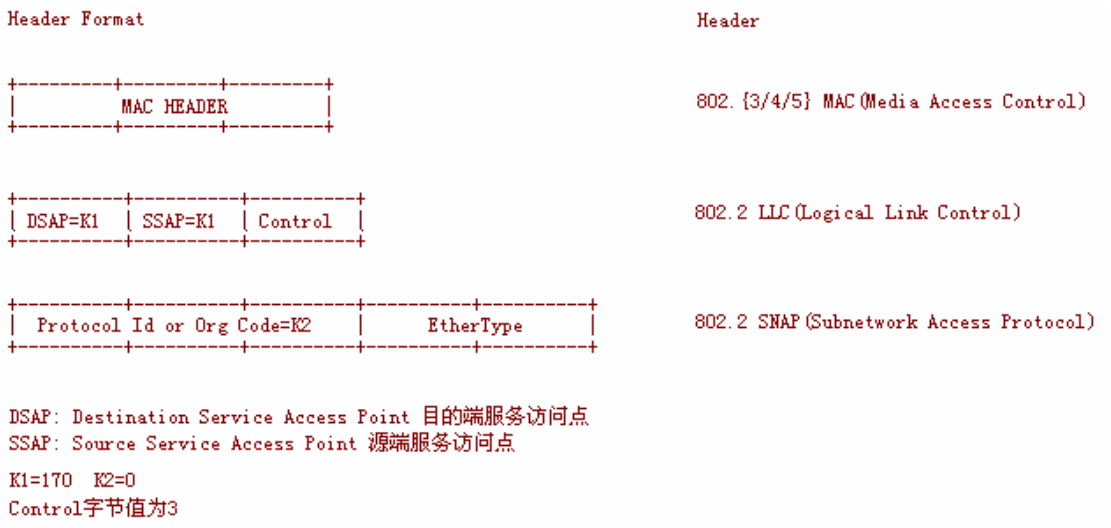


图 1-20 802 标准首部格式

有关使用 802 标准传送数据包的内容参考 RFC-1042。

这些域定义在 net/protocol.c 文件中：

```
struct net_proto protocols[] = {
#ifdef      CONFIG_UNIX
   { "UNIX",   unix_proto_init       },
#endif
#if defined(CONFIG_IPX)||defined(CONFIG_ATALK)
   { "802.2",    p8022_proto_init },
   { "SNAP",   snap_proto_init },
#endif
#ifdef CONFIG_AX25
   { "AX.25",  ax25_proto_init },
#endif
#ifdef      CONFIG_INET
   { "INET",    inet_proto_init},
#endif
#ifdef    CONFIG_IPX
   { "IPX",      ipx_proto_init },
#endif
#ifdef CONFIG_ATALK
   { "DDP",     atalk_proto_init },
#endif
   { NULL,     NULL            }
};
```

文件结尾是对一些函数的声明。在讨论到具体这些函数的具体定义时再给予解释。

```
/* include/linux/net.h ************************************************************/
/*
 * NET          An implementation of the SOCKET network access protocol.
 *          This is the master header file for the Linux NET layer,
 *          or, in plain English: the networking handling part of the
 *          kernel.
 *
 * Version:     @(#)net.h       1.0.3       05/25/93
 *
 * Authors:     Orest Zborowski, <obz@Kodak.COM>
 *          Ross Biro, <bir7@leland.Stanford.Edu>
 *          Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
 *
 *          This program is free software; you can redistribute it and/or
 *          modify it under the terms of the GNU General Public License
 *          as published by the Free Software Foundation; either version
 *          2 of the License, or (at your option) any later version.
 */
#ifndef _LINUX_NET_H
#define _LINUX_NET_H


#include <linux/wait.h>
#include <linux/socket.h>


#define NSOCKETS      2000            /* Dynamic, this is MAX LIMIT   */
#define NSOCKETS_UNIX  128        /* unix domain static limit     */
#define NPROTO          16          /* should be enough for now..*/


#define SYS_SOCKET  1           /* sys_socket(2)            */
#define SYS_BIND        2           /* sys_bind(2)              */
#define SYS_CONNECT     3           /* sys_connect(2)           */
#define SYS_LISTEN  4           /* sys_listen(2)            */
#define SYS_ACCEPT  5           /* sys_accept(2)            */
#define SYS_GETSOCKNAME 6           /* sys_getsockname(2)           */
#define SYS_GETPEERNAME 7           /* sys_getpeername(2)           */
#define SYS_SOCKETPAIR 8            /* sys_socketpair(2)            */
#define SYS_SEND        9           /* sys_send(2)              */
#define SYS_RECV        10          /* sys_recv(2)              */
#define SYS_SENDTO  11          /* sys_sendto(2)            */
#define SYS_RECVFROM    12          /* sys_recvfrom(2)          */
#define SYS_SHUTDOWN    13          /* sys_shutdown(2)          */
```

```
#define SYS_SETSOCKOPT 14          /* sys_setsockopt(2)          */
#define SYS_GETSOCKOPT       15          /* sys_getsockopt(2)          */


typedef enum {
   SS_FREE = 0,                    /* not allocated              */
   SS_UNCONNECTED,                 /* unconnected to any socket  */
   SS_CONNECTING,                  /* in process of connecting   */
   SS_CONNECTED,                   /* connected to socket        */
   SS_DISCONNECTING                /* in process of disconnecting*/
} socket_state;

#define SO_ACCEPTCON    (1<<16)         /* performed a listen         */
#define SO_WAITDATA     (1<<17)         /* wait data to read          */
#define SO_NOSPACE  (1<<18)         /* no space to write          */

#ifdef __KERNEL__
/*
  * Internal representation of a socket. not all the fields are used by
  * all configurations:
  *
  *          server                  client
  * conn          client connected to  server connected to
  * iconn  list of clients          -unused-
  *             awaiting connections
  * wait          sleep for clients,      sleep for connection,
  *          sleep for i/o           sleep for i/o
  */
struct socket {
   short                type;       /* SOCK_STREAM, ...           */
   socket_state         state;
   long                 flags;
   struct proto_ops  *ops;              /* protocols do most everything     */
   void                 *data;          /* protocol data              */
   struct socket              *conn;          /* server socket connected to  */
   struct socket              *iconn;         /* incomplete client conn.s    */
   struct socket              *next;
   struct wait_queue          **wait;         /* ptr to place to wait on */
   struct inode         *inode;
   struct fasync_struct    *fasync_list;    /* Asynchronous wake up list */
};

#define SOCK_INODE(S)     ((S)->inode)
```

```
struct proto_ops {
   int      family;
   int      (*create)  (struct socket *sock, int protocol);
   int      (*dup)          (struct socket *newsock, struct socket *oldsock);
   int      (*release)(struct socket *sock, struct socket *peer);
   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);
   int      (*read)         (struct socket *sock, char *ubuf, int size,
                  int nonblock);
   int      (*write)   (struct socket *sock, char *ubuf, int size,
                  int nonblock);
   int      (*select)  (struct socket *sock, int sel_type,
                  select_table *wait);
   int      (*ioctl)    (struct socket *sock, unsigned int cmd,
                  unsigned long arg);
   int      (*listen)   (struct socket *sock, int len);
   int      (*send)         (struct socket *sock, void *buff, int len, int nonblock,
                  unsigned flags);
   int      (*recv)         (struct socket *sock, void *buff, int len, int nonblock,
                  unsigned flags);
   int      (*sendto) (struct socket *sock, void *buff, int len, int nonblock,
                  unsigned flags, struct sockaddr *, int addr_len);
   int      (*recvfrom)    (struct socket *sock, void *buff, int len, int nonblock,
                  unsigned flags, struct sockaddr *, int *addr_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      (*fcntl)    (struct socket *sock, unsigned int cmd,
                  unsigned long arg);
};

struct net_proto {
      char *name;          /* Protocol name */
      void (*init_func)(struct net_proto *);    /* Bootstrap */
};
```

```
extern int sock_awaitconn(struct socket *mysock, struct socket *servsock, int flags);
extern int sock_wake_async(struct socket *sock, int how);
extern int sock_register(int family, struct proto_ops *ops);
extern int sock_unregister(int family);
extern struct socket *sock_alloc(void);
extern void      sock_release(struct socket *sock);
#endif /* __KERNEL__ */
#endif      /* _LINUX_NET_H */
/* include/linux/net.h ************************************************************/
```

1.16 netdevice.h 头文件

该文件定义有内核网络栈核心数据结构 device 结构（2.4.0 版本后更名为现在的 net_device）。另外还有其它重要数据结构的定义如 dev_mac_list (多播地址)，packet_type (网络层协议结构)，最后是对部分重要函数的声明，这些函数分布定义在多个文件中。

DEV_NUMBUFFS

数据包发送硬件缓冲队列数目，定义为 3，即存在 3 个级别的发送队列，每个发送队列作为 device 结构中 buffs 数组的一个元素而存在。而 buffs 数组的元素数目即为 DEV_NUMBUFFS。

MAX_ADDR_LEN

硬件地址最大长度，定义为 7，实际上一般为 6 个字节，最后一个字节赋值为 0，便于字符串处理。

MAX_HEADER

最大链路层首部长度，定义为 18，其中包括了帧尾部的 FCS 字段 4 个字节。

IS_MYADDR：远端发往本机的数据包
IS_LOOPBACK：本机发往本机的数据包
IS_BROADCAST：远端或本机发送的广播数据包
IS_INVBCAST：无效的广播数据包
IS_MULTICAST：多播数据包
所接收数据包的 IP 目的地址相对于本机而言的地址类别。

dev_mac_list 结构（如下所示）用于表示链路层多播地址。

```
struct dev_mc_list
{
    struct dev_mc_list *next;
    char dmi_addr[MAX_ADDR_LEN];
    unsigned short dmi_addrlen;
    unsigned short dmi_users;
};
```

next：用于构成一个多播地址链表。
dmi_addr：具体多播地址

dmi_addrlen：多播地址长度，一般即为硬件地址长度即 6 个字节。
dmi_users：该多播地址结构的使用数。

device 结构：网络栈代码核心结构,其字段及其含义如下：
name
设备名称

rmem_end, rmem_start
设备读缓冲区空间。

mem_end, mem_start
设备总缓冲区首地址和尾地址。注意这两个字段所表示的缓冲区将被分为两个部分：读缓冲区和写缓冲区。一般写缓冲区占据整缓冲区前面一部分，之后的缓冲区均划为读缓冲区。读缓冲区将有 rmem_start, rmem_end 字段具体表示。

base_addr
设备寄存器读写 I/O 基地址。

irq
设备所使用中断号。

start，tbusy，interrupt
这两个字段表示设备所处的状态：
start=1 表示设备已处于工作状态。
tbusy=1 表示设备正忙于数据包发送，如果再有数据包发送，则需等待。
interrupt=1：软件正在进行设备中断处理。

next
构成一个设备队列，dev_base（drivers/net/Space.c）内核指针即指向这个队列。

init
设备初始化指针，对于系统静态定义的设备，在网络栈初始化时被调用对设备进行相应的初始化工作。而对于“动态插入”的设备（以动态模块方式加载的设备），则是在注册设备时（通过调用 register_netdev 函数），该指针指向的函数被调用初始化设备。对于网卡驱动程序编写而言，如果对应的设备需要初始化，则应将该指针指向对应的函数。

if_port
对于具有多个端口的设备，该字段指定使用的设备端口号。例如设备同时支持铜轴电缆和双绞线以太网连接时，需要使用该字段。但该字段不常使用，因为大多数设备通常只有一个端口。

dma
设备所用的 DMA 通道号，如果设备使用 DMA 方式进行数据的传送。

get_stats
设备信息获取函数指针。所谓信息一般即指经过该设备接收和发送的数据包个数，总字节数，发送或接收失败的数据包个数等等。具体参考 enet_statistics（if_ether.h）结构的定义。

trans_start
该字段的设置功能用于传输超时计算。每当设备新发送一个数据包时，更新该字段为当前系统当前时间（以 jiffies 表示），如果传输超时时间为 T，则当 jiffies>trans_start+T 时，表示传输超时，此时需要对数据包进行重新发送。不过在本版本内核网络代码实现上，并没有使用该字段判断传输超时，而是在传输层进行超时判断和重传（使用系统定时器实现）。

last_rx
该字段表示上次接收一个数据包的时间，以 jiffies 值表示。该字段仅仅用于表示信息，没有使用在其它判断条件上。

flags
设备标志位。该标志位一方面表示设备目前所处的状态，另一方面表示设备所具备的功能。该字段的取值如下（定义在 if.h 文件中）：
IFF_UP
设备正处正常运行状态。

IFF_BROADCAST
设备支持广播，即设备结构中广播字段地址有效。

IFF_DEBUG
打开设备的调试选项，打印调试信息。

IFF_LOOPBACK
这是一个回环设备，该设备并无实际硬件对应，而仅存在于软件上。

IFF_POINTOPOINT
这是一个点到点的网络设备，此时 device 结构中 pa_dstaddr 地址有效，表示对方 IP 地址。

IFF_NOTRAILERS
不使用网络跟踪选项。

IFF_RUNNING
设备正处于正常运行状态，该字段本版本中的含义同 IFF_UP。

IFF_NOARP
该设备不使用 ARP 协议完成 IP 地址到硬件地址的映射，如回环设备。

IFF_PROMISC
该设备处于混杂接收模式，即接收该设备所连接网络上传输的所有数据包，无论该数据包发往何处。

IFF_ALLMULTI
指定该设备接收所有多播数据包。

IFF_MASTER
IFF_SLAVE
这两个字段用于主从设备运行模式。一个主设备管理多个从设备。一般主设备是软件上的设比，而从设备是真正的硬件网卡设备。当发送数据包时，主设备根据某种策略选择一个从设备完成实际的发送。主设备中对应的 device 结构中 flags 标志位设置 IFF_MASTER 标志，而从设备设置 IFF_SLAVE 标志。

IFF_MULTICAST
该设备支持多播，此时 device 结构中 mc_list 字段有效，表示该设备所支持的多播地址。

family
该字段表示设备所属的域协议，可取值如下（均定义在 include/linux/socket.h）：
AF_UNSPEC
AF_UNIX
AF_INET
AF_AX25
AF_IPX
AF_APPLETALK

metric
代价度量值，现在一般由来表示所经过的路由器数目。本版本没有使用该字段。

mtu
该接口设备的最大传输单元。

type
该设备所属的硬件类型。该字段可取值如下（这些值均定义在 if_arp.h 文件中）：
ARPHRD_NETROM
NET/ROM 是一种被无线电业余爱好者广泛使用的协议。LINUX NET/ROM 协议允许使用类似网络套接字的方式接收数据。NET/ROM 协议只支持连接模式，这种模式也被使用在 SOCK_SEQPACKET 类型的套接字通信中。用户必须保证发送的数据被适当的打包，接收的数据也是以一个包的形式接收到缓冲区中。NET/ROM 地址格式由 6 个 ASCII 字符和一个称为 SSID 的数字构成，sockaddr_ax25 数据结构用于表示这种地址结构形式。

ARPHRD_ETHER
10Mbps 以太网硬件类型。

ARPHRD_EETHER
实验型以太网硬件类型，这是早期的叫法，现在应该称之为 EtherII 类型。

ARPHRD_AX25
AX.25 是从 X.25 协议族演变而来的一种链路层协议。该协议被设计为业余无线电爱好者使用。AX.25 协议定义了 OSI 物理层和链路层两个层面，负责在节点之间传输数据，并且探测通信通道中可能出现的错误。AX.25 支持连接的和无连接通信方式。AX.25 大多被使用在无线电工作站之间建立直接的点对点连接。AX.25 定义了一个完备的网络协议族，但较少使用。NET/ROM，ROSE，TexNet 协议通常提供节点之间的路由功能。原理上讲，任何网络层协议都可使用 AX.25 传输数据包，包括最普遍使用的 IP 协议。

ARPHRD_PRONET
Proteon 公司制定的一种令牌环网络协议。该协议本版本内核网络代码没有实现。

ARPHRD_CHAOS
Chaosnet 最早是由 MIT AI 实验室的 Thomas Knight 和 Jack Holloway 发展起来的。主要涉及两方面独立的但又相互联系的技术。第一也是较为广泛使用的是建立在 MIT 校园内部逐渐流行的 Lisp 主机之间的一系列以包为基础的通信协议技术。第二是早期局域网硬件实现技术。Chaosnet 协议最初是基于后一种技术实现的，而这种实现又是基于早期 Xerox PARC 3Mbs 以太网和 ARPANET 网以及 TCP 协议。Chaosnet 网络拓扑结构通常是由一系列线性（而非环形）电缆构成，每条电缆长度可达 1 千米，大约可挂 12 个客户端。Chaosnet 协议后来被作为以太网协议包结构中负载而实现。Chaosnet 特别的被使用在局域网中，对于广域网的支持不够。

ARPHRD_IEEE802
IEEE 制定的对应于 OSI 模型物理层和链路层的标准。

ARPHRD_ARCNET
ARCNET 是 Attached Resource Computer NETwork 首字母缩写形式，是一种局域网网络协议。ARCNET 由 Datapoint 公司的首席工程师 John Murphy 于 1976 年发展起来的，最初被使用在微型机的网络系统中，在 1980 年代办公自动化应用中逐渐流行。自此逐渐在嵌入式系统中占据一席之地。它是第一个以局域网为基础的聚族解决方式，最初的设计理念是作为更大更贵计算机系统的一种替代方案。一个应用程序可以被部署在一个具有哑终端的计算机上，当用户数据超过计算机所允许计算能力时，其它计算机作为一种资源通过 ARCnet 网络连接进来，运行同一个应用程序以提供更强的计算能力。这种逐渐增强计算能力的方式有些类似于今天常说的分布式计算方式。在 1970 年代末，全世界有超过 1 万个 ARCNET 局域网被作为商业应用。为满足更高的宽带要求，一个称为 ARCnet Plus 的标准被开发出来并于 1992 公布。ARCNET 最终被标准化为 ANSI ARCNET 878.1。名字也由 ARCnet 改为 ARCNET。有关ARCNET的更为详细的内容请访问：http://en.wikipedia.org/wiki/ARCnet。

ARPHRD_APPLETLK
AppleTalk 是 Apple 公司为网络计算机开发的一套协议族。在早期 Macintosh 计算机上被使用，现在 AppleTalk 逐渐淡出，Apple 公司也逐渐转向 TCP/IP 协议族。AppleTalk 是一套协议族，而并非单个协议。这个协议族包括：AppleTalk 地址解析协议（AARP），AppleTalk 数据系统协议（ADSP），Apple 文件协议（AFP），AppleTalk 会话协议（ASP），AppleTalk 交换协议（ATP），数据报传输协议（DDP），名字绑定协议（NBP），打印机访问协议（PAP），路由表维护协议（RTMP），区域信息协议（ZIP）。这套协议独成系统完成网络数据交换。

AppleTalk 的设计紧紧依照 OSI 七层模型分层结构。表 1-6 显示了 AppleTalk 物理实现与 OSI 模型的分层对应关系。

表 1-10 AppleTalk 分层结构与 OSI 模型

OSI 模型	对应的 AppleTalk 分层协议
应用层	Apple 文件协议（AFP）
表示层	Apple 文件协议（AFP）
会话层	区域信息协议（ZIP） AppleTalk 会话协议（ASP） AppleTalk 数据流协议（ADSP）
传输层	AppleTalk 交换协议（ATP） AppleTalk 回复协议（AEP） 名字绑定协议（NBP） 路由表维护协议（RTMP）
网络层	数据包传输协议（DDP）
链路层	EtherTalk 链路访问协议（ELAP） LocalTalk 链路访问协议（LLAP） TokenTalk 链路访问协议（TLAP） 光纤分布式数据接口（FDDI）
物理层	LocalTalk 驱动程序，Ethernet 驱动程序，Token Ring 驱动程序…

以下这些类型均无需 ARP 协议进行 IP 地址到硬件地址的解析。
ARPHRD_SLIP
ARPHRD_CSLIP
ARPHRD_SLIP6
ARPHRD_CSLIP6
串行线网际协议。

ARPHRD_RSRVD
保留。

ARPHRD_ADAPT
适配模式下的 SLIP 协议。

ARPHRD_PPP
点对点协议。

ARPHRD_TUNNEL
IP 隧道协议。

hard_header_len
硬件首部长度，如以太网硬件首部长度为 14 字节。

priv
私有数据指针，一般使用在网卡驱动程序中用于指向自定义数据结构。

broadcast[MAX_ADDR_LEN]
链路层硬件广播地址。

dev_addr[MAX_ADDR_LEN]
本设备硬件地址。

addr_len
硬件地址长度，如以太网硬件地址长度为 6 个字节。

pa_addr
本地 IP 地址。

pa_brdaddr
网络层广播 IP 地址。

pa_dstaddr
该字段仅使用在点对点网络中指对点 IP 地址。

pa_mask
IP 地址网络掩码。

pa_alen
IP 地址长度，通常为 4 个字节。

mc_list
mc_count
MAC 多播地址链表，对于一个多播数据包，如果其多播地址是链表中一项，则接收，否则丢弃。mc_count 表示链表中多播地址数目。注意这个链表中存储的是 MAC 多播地址，而 ip_mc_list 链表中存储的是 IP 多播地址。MAC 多播地址与 IP 多播地址之间的映射关系在第二章中分析 igmp.c 文件中给出。到时将对 device 结构，sock 结构中用于多播的字段进行集中分析。

ip_mc_list
网络层 IP 多播地址链表，处理方式同 mc_list.

pkt_queue
该设备缓存的待发送的数据包个数。

slave
用于虚拟主从设备机制中，该字段指向某个从设备。虚拟主从设备机制中，主设备管理多个从设备，且主设备一般仅存在于软件上，具体的发送和接收数据包由从设备完成。当上层发

送某个数据包时，主设备根据某种策略选择其中一个从设备，将数据包交给该从设备完成具体的发送任务。

buffs[DEV_NUMBUFFS]
设备缓存的待发送的数据包，这些数据包由于某种原因之前没有成功发送，故缓存到 device 结构的 buffs 数组指向的某个队列中。DEV_NUMBUFFS 值为 3，即有三个缓冲队列，分为三个优先级。每次发送从第一个元素指向的队列中摘取数据包发送，第一个队列空后，继续发送地址个队列，依次到第三个队列。

open
stop
设备打开和关闭时调用的函数，网卡驱动程序需要实现这个函数，并将这个函数指针指向相应的函数，从而当用户停止或启动网卡工作时进行适当的响应。

hard_start_xmit
数据包发送函数，网卡驱动程序中数据包发送核心函数，网络栈上层代码将调用该函数指针指向的函数发送数据包，该函数硬件相关，主要将内核缓冲区中用户要发送数据复制到硬件缓冲区中并启动设备将数据发送到物理传输介质上（如铜轴电缆或双绞线）。

hard_header
rebuild_header
硬件首部建立函数。这两个函数用于建立 MAC 首部。hard_header 一般仅在第一次建立硬件首部时被调用，如果首次建立硬件首部失败（如暂时无法知道远端的硬件地址），则之后可重复多次调用 rebuild_header 函数重新建立硬件首部。网卡驱动程序必须对这两个字段进行适当初始化，即便是不使用 ARP 协议。

type_trans
该指针指向的函数用于从接收到的数据包提取 MAC 首部中类型字段值，从而将数据包传送给适当的协议处理函数进行处理。

set_multicast_list
该指针指向的函数用于设置该设备的多播地址。

set_mac_address
如果设备支持改变其硬件地址，则该指针指向的函数即可被调用改变该设备对应的硬件地址。

do_ioctl
set_config
这两个函数用于设置或获取设备的有关控制或统计信息。如可改变设备 MTU 值大小。

到此为止，device 结构的所有字段均以解释完毕。对于 device 结构字段的深刻理解是编写好网卡驱动程序的关键。

packet_type 结构用于表示网络层协议。每个网络层协议对应一个 packet_type 结构。内核支持的所有网络层协议构成一个链表，系统变量 ptype_base（net/inet/dev.c）指向这个链表。当系统接收到一个网络数据包时，通过 type_trans 指针指向函数得到数据包所使用的网络层协议，之后遍历 ptype_base 指向的链表，比较 packet_type 结构中 type 字段与数据包所使用的协议是否相符，如相符，则调用 packet_type 结构中 func 指针指向的函数，将数据包交给其处理（即传往上层进行处理）。例如对应 IP 协议，对应的处理函数为 ip_rcv。

```
struct packet_type {
   unsigned short    type;/* This is really htons(ether_type). */
   struct device *   dev;
   int               (*func) (struct sk_buff *, struct device *, struct packet_type *);
   void              *data;
   struct packet_type      *next;
};
```

type

对应的网络层协议。该字段可取值如下（include/linux/if_ether.h）：

ETH_P_LOOP
ETH_P_ECHO
ETH_P_PUP
ETH_P_IP
ETH_P_ARP
ETH_P_RARP
ETH_P_X25
ETH_P_ATALK
ETH_P_IPX
ETH_P_802_3
ETH_P_AX25
ETH_P_ALL
ETH_P_802_2
ETH_P_SNAP

dev

数据包接收网络接口设备。一般初始化为 NULL，即并不计较数据包是从哪个网络接口接收的，只要满足 type 字段，就调用处理函数进行处理。

func

协议处理函数。如对于 IP 协议，该指针指向 ip_rcv 函数。

data

指向特定数据结构以进行某种区分。这在下文分析到具体函数时（dev.c 文件中）在进行交待。

next

该字段用于构成一个由 packet_type 结构组成的链表。

该文件最后部分声明了一系列重要函数，在下文中分析到相关部分时进行讨论，现在读者只需有个认识即可。

```
/* include/linux/netdevice.h **********************************************************/
/*
 * INET        An implementation of the TCP/IP protocol suite for the LINUX
 *        operating system.    INET is implemented using the    BSD Socket
 *        interface as the means of communication with the user level.
 *
 *        Definitions for the Interfaces handler.
 *
 * Version:    @(#)dev.h      1.0.10      08/12/93
 *
 * Authors:    Ross Biro, <bir7@leland.Stanford.Edu>
 *        Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
 *        Corey Minyard <wf-rch!minyard@relay.EU.net>
 *        Donald J. Becker, <becker@super.org>
 *        Alan Cox, <A.Cox@swansea.ac.uk>
 *        Bjorn Ekwall. <bj0rn@blox.se>
 *
 *        This program is free software; you can redistribute it and/or
 *        modify it under the terms of the GNU General Public License
 *        as published by the Free Software Foundation; either version
 *        2 of the License, or (at your option) any later version.
 *
 *        Moved to /usr/include/linux for NET3
 */
#ifndef _LINUX_NETDEVICE_H
#define _LINUX_NETDEVICE_H

#include <linux/if.h>
#include <linux/if_ether.h>
#include <linux/skbuff.h>

/* for future expansion when we will have different priorities. */
#define DEV_NUMBUFFS    3
#define MAX_ADDR_LEN    7
#define MAX_HEADER      18

#define IS_MYADDR    1          /* address is (one of) our own*/
#define IS_LOOPBACK      2          /* address is for LOOPBACK*/
#define IS_BROADCAST     3          /* address is a valid broadcast*/
#define IS_INVBCAST 4          /* Wrong netmask bcast not for us (unused)*/
```

```
#define IS_MULTICAST    5       /* Multicast IP address */

/*
 *    We tag these structures with multicasts.
 */

struct dev_mc_list
{
    struct dev_mc_list *next;
    char dmi_addr[MAX_ADDR_LEN];
    unsigned short dmi_addrlen;
    unsigned short dmi_users;
};

/*
 * The DEVICE structure.
 * Actually, this whole structure is a big mistake.    It mixes I/O
 * data with strictly "high-level" data, and it has to know about
 * almost every data structure used in the INET module.
 */
struct device
{

  /*
   * This is the first field of the "visible" part of this structure
   * (i.e. as seen by users in the "Space.c" file).    It is the name
   * the interface.
   */
  char    *name;

  /* I/O specific fields - FIXME: Merge these and struct ifmap into one */
  unsigned long    rmem_end;        /* shmem "recv" end    */
  unsigned long    rmem_start;      /* shmem "recv" start    */
  unsigned long    mem_end;         /* sahared mem end    */
  unsigned long    mem_start;       /* shared mem start */
  unsigned long    base_addr;       /* device I/O address    */
  unsigned char    irq;             /* device IRQ number    */

  /* Low-level status flags. */
  volatile unsigned char    start,       /* start an operation*/
                            tbusy,       /* transmitter busy  */
                            interrupt;       /* interrupt arrived  */

  struct device    *next;
```

```
/* The device initialization function. Called only once. */
int         (*init)(struct device *dev);

/* Some hardware also needs these fields, but they are not part of the
     usual set specified in Space.c. */
unsigned char     if_port;          /* Selectable AUI, TP,..*/
unsigned char     dma;              /* DMA channel          */

struct enet_statistics* (*get_stats)(struct device *dev);

/*
  * This marks the end of the "visible" part of the structure. All
  * fields hereafter are internal to the system, and may change at
  * will (read: may be cleaned up at will).
  */

/* These may be needed for future network-power-down code. */
unsigned long     trans_start;      /* Time (in jiffies) of last Tx  */
unsigned long     last_rx;    /* Time of last Rx          */

unsigned short    flags;      /* interface flags (a la BSD)   */
unsigned short    family;     /* address family ID (AF_INET)   */
unsigned short    metric;     /* routing metric (not used)    */
unsigned short    mtu;        /* interface MTU value          */
unsigned short    type;       /* interface hardware type      */
unsigned short    hard_header_len;    /* hardware hdr length    */
void     *priv;       /* pointer to private data */

/* Interface address info. */
unsigned char     broadcast[MAX_ADDR_LEN];    /* hw bcast add       */
unsigned char     dev_addr[MAX_ADDR_LEN];     /* hw address  */
unsigned char     addr_len; /* hardware address length     */
unsigned long     pa_addr;  /* protocol address         */
unsigned long     pa_brdaddr;     /* protocol broadcast addr      */
unsigned long     pa_dstaddr;     /* protocol P-P other side addr       */
unsigned long     pa_mask; /* protocol netmask         */
unsigned short    pa_alen;  /* protocol address length      */

struct dev_mc_list      *mc_list; /* Multicast mac addresses     */
int       mc_count;       /* Number of installed mcasts*/

struct ip_mc_list   *ip_mc_list;  /* IP multicast filter chain        */
```

```
    /* For load balancing driver pair support */

    unsigned long     pkt_queue;      /* Packets queued */
    struct device       *slave;     /* Slave device */


    /* Pointer to the interface buffers. */

    struct sk_buff_head       buffs[DEV_NUMBUFFS];

    /* Pointers to interface service routines. */
    int       (*open)(struct device *dev);
    int       (*stop)(struct device *dev);
    int       (*hard_start_xmit) (struct sk_buff *skb, struct device *dev);
    int         (*hard_header) (unsigned char *buff,
                                struct device *dev,
                                unsigned short type,
                                void *daddr,
                                void *saddr,
                                unsigned len,
                                struct sk_buff *skb);
    int         (*rebuild_header)(void *eth, struct device *dev,
                            unsigned long raddr, struct sk_buff *skb);
    unsigned short      (*type_trans) (struct sk_buff *skb, struct device *dev);
#define HAVE_MULTICAST
    void      (*set_multicast_list)(struct device *dev, int num_addrs, void *addrs);
#define HAVE_SET_MAC_ADDR
    int       (*set_mac_address)(struct device *dev, void *addr);
#define HAVE_PRIVATE_IOCTL
    int       (*do_ioctl)(struct device *dev, struct ifreq *ifr, int cmd);
#define HAVE_SET_CONFIG
    int         (*set_config)(struct device *dev, struct ifmap *map);

};

struct packet_type {
    unsigned short    type;/* This is really htons(ether_type). */
    struct device *     dev;
    int         (*func) (struct sk_buff *, struct device *, struct packet_type *);
    void    *data;
    struct packet_type      *next;
};
```

```
#ifdef __KERNEL__

#include <linux/notifier.h>

/* Used by dev_rint */
#define IN_SKBUFF	1

extern volatile char in_bh;

extern struct device loopback_dev;
extern struct device *dev_base;
extern struct packet_type *ptype_base;


extern int		ip_addr_match(unsigned long addr1, unsigned long addr2);
extern int		ip_chk_addr(unsigned long addr);
extern struct device *ip_dev_check(unsigned long daddr);
extern unsigned long	ip_my_addr(void);
extern unsigned long	ip_get_mask(unsigned long addr);

extern void		dev_add_pack(struct packet_type *pt);
extern void		dev_remove_pack(struct packet_type *pt);
extern struct device *dev_get(char *name);
extern int		dev_open(struct device *dev);
extern int		dev_close(struct device *dev);
extern void		dev_queue_xmit(struct sk_buff *skb, struct device *dev,
				       int pri);
#define HAVE_NETIF_RX 1
extern void		netif_rx(struct sk_buff *skb);
/* The old interface to netif_rx(). */
extern int		dev_rint(unsigned char *buff, long len, int flags,
				 struct device * dev);
extern void		dev_transmit(void);
extern int		in_net_bh(void);
extern void		net_bh(void *tmp);
extern void		dev_tint(struct device *dev);
extern int		dev_get_info(char *buffer, char **start, off_t offset, int length);
extern int		dev_ioctl(unsigned int cmd, void *);

extern void		dev_init(void);

/* These functions live elsewhere (drivers/net/net_init.c, but related) */

extern void		ether_setup(struct device *dev);
```

```
extern int      ether_config(struct device *dev, struct ifmap *map);
/* Support for loadable net-drivers */
extern int      register_netdev(struct device *dev);
extern void         unregister_netdev(struct device *dev);
extern int          register_netdevice_notifier(struct notifier_block *nb);
extern int      unregister_netdevice_notifier(struct notifier_block *nb);
/* Functions used for multicast support */
extern void         dev_mc_upload(struct device *dev);
extern void         dev_mc_delete(struct device *dev, void *addr, int alen, int all);
extern void         dev_mc_add(struct device *dev, void *addr, int alen, int newonly);
extern void         dev_mc_discard(struct device *dev);
/* This is the wrong place but it'll do for the moment */
extern void         ip_mc_allhost(struct device *dev);
#endif /* __KERNEL__ */

#endif    /* _LINUX_DEV_H */
/* include/linux/netdevice.h ********************************************************/
```

1.17 notifier.h 头文件

该文件定义了一个用于主动通知的策略：notifer_block 数据结构。对于网卡设备而言，网卡设备的启动和关闭是事件。内核中某些模块需要得到这些事件的通知从而采取相应的措施。这是通过主动通知的方式完成的。当事件发生时，事件通知者遍历系统中某个队列，调用感兴趣者注册的函数，从而通知这些感兴趣者。要得到事件的通知，首先必须表示对这些事件的兴趣，而要表示对某个事件感兴趣，首先必须注册这个“兴趣”，而 notifer_block 结构就表示一个“兴趣”。一般一个队列本身表示某种类型的事件，对此种类型事件感兴趣者注册到这个队列中，就表示了其兴趣所在。当事件发生时，该队列中所有的感兴趣者之前注册的函数将被调用，从而得到对事件发生的通知。

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

notifier_call

感兴趣者注册的被调用函数。当注册的感兴趣事件发生时，事件通知者将调用该函数以通知感兴趣者。

next

该字段用于连接到一个队列中。

prority
优先级，即该感兴趣者对某种事件感兴趣的程度。优先级越大，表示越关注此种事件，所以将最先被通知到。其注册时对应的 notifier_block 结构将被插入到队列前部。

NOTIFY_DONE
NOTIFY_OK
NOTIFY_STOP_MASK
NOTIFY_BAD
这四个常量用于表示何时终止对感兴趣者队列中注册函数的调用。如果前一个注册函数返回 NOTIFY_DONE，表示通知完成，此后即便队列还有尚未被通知到的感兴趣者，依然停止通知操作。返回 NOTIFY_OK 则继续进行通知直到队列尾部。NOTIFY_BAD 表示某个感兴趣者出现问题，此时也是停止通知队列中其它感兴趣者。NOTIFY_STOP_MASK 是位掩码，用于判断返回值。

notifier_chain_register 函数用于感兴趣者进行注册事件发生时被调用的函数从而得到事件发生通知。
其中 list 参数表示插入的队列，诚如上文所述，一般队列本身即表示某种类型的事件，如果对此类事件感兴趣，就注册到该队列中。
参数 n 中含有事件发生时被调用的函数。

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

该函数的功能即将表示感兴趣者的 notifier_block 结构根据其优先级插入到系统某个队列中。

notifier_chain_unregister 函数完成的功能与 notifier_chain_register 函数正好相反。其撤销对某类事件的兴趣，也即将表示兴趣的相应 notifier_block 结构从相应队列中删除。

```
extern __inline__ int notifier_chain_unregister(struct notifier_block **nl, struct notifier_block *n)
{
	while((*nl)!=NULL)
	{
		if((*nl)==n)
		{
			*nl=n->next;
			return 0;
```

```
            }
            nl=&((*nl)->next);
        }
        return -ENOENT;
}
```

至于 notifier_call_chain 则是当事件发生时完成对感兴趣者的通知。通知方式即调用感兴趣者之前注册的函数。

```
extern __inline__ int notifier_call_chain(struct notifier_block **n, unsigned long val, void *v)
{
        int ret=NOTIFY_DONE;
        struct notifier_block *nb = *n;
        while(nb)
        {
            ret=nb->notifier_call(val,v);
            if(ret&NOTIFY_STOP_MASK)
                return ret;
            nb=nb->next;
        }
        return ret;
}
```

该函数结构简单，此处不多作说明。

文件最后定义了几种事件：
NETDEV_UP：某个网卡设备启动
NETDEV_DOWN：某个网卡设备关闭
NETDEV_REBOOT：设备进行重启动

本版本内核网络代码维护一个事件通知队列：由全局变量 netdev_chain 指向的队列。该队列表示了以上三种事件。定义如下（net/inet/dev.c 文件中）：

```
struct notifier_block *netdev_chain=NULL;
```

```
/* include/linux/notifier.h ************************************************/
/*
 *    Routines to manage notifier chains for passing status changes to any
 *    interested routines. We need this instead of hard coded call lists so
 *    that modules can poke their nose into the innards. The network devices
 *    needed them so here they are for the rest of you.
 *
 *    Alan Cox <Alan.Cox@linux.org>
 */

#ifndef _LINUX_NOTIFIER_H
#define _LINUX_NOTIFIER_H
```

```
#include <linux/errno.h>

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


#ifdef __KERNEL__

#define NOTIFY_DONE		0x0000		/* Don't care */
#define NOTIFY_OK		0x0001		/* Suits me */
#define NOTIFY_STOP_MASK	0x8000		/* Don't call further */
#define NOTIFY_BAD		(NOTIFY_STOP_MASK|0x0002)/* Bad/Veto action */

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

/*
 *	Warning to any non GPL module writers out there.. these functions are
 *	GPL'd
 */

extern __inline__ int notifier_chain_unregister(struct notifier_block **nl, struct notifier_block *n)
{
	while((*nl)!=NULL)
	{
		if((*nl)==n)
		{
			*nl=n->next;
			return 0;
		}
```

```
            nl=&((*nl)->next);
        }
        return -ENOENT;
}

/*
 *    This is one of these things that is generally shorter inline
 */

extern __inline__ int notifier_call_chain(struct notifier_block **n, unsigned long val, void *v)
{
        int ret=NOTIFY_DONE;
        struct notifier_block *nb = *n;
        while(nb)
        {
                ret=nb->notifier_call(val,v);
                if(ret&NOTIFY_STOP_MASK)
                        return ret;
                nb=nb->next;
        }
        return ret;
}


/*
 *    Declared notifiers so far. I can imagine quite a few more chains
 *    over time (eg laptop power reset chains, reboot chain (to clean
 *    device units up), device [un]mount chain, module load/unload chain,
 *    low memory chain, screenblank chain (for plug in modular screenblankers)
 *    VC switch chains (for loadable kernel svgalib VC switch helpers) etc...
 */

/* netdevice notifier chain */
#define NETDEV_UP    0x0001    /* For now you can't veto a device up/down */
#define NETDEV_DOWN     0x0002

/* Tell a protocol stack a network interface
         detected a hardware crash and restarted
         - we can use this eg to kick tcp sessions
       once done */
#define NETDEV_REBOOT 0x0003

#endif
#endif
```

/* include/linux/notifier.h **/

1.18 ppp.h 头文件

点对点协议（PPP：Point-to-Point Protocol）为在点对点连接上传输多种网络层协议类型的数据包提供了一种标准方法。该协议修改了 SLIP 协议中所有的缺陷。PPP 协议包括以下三个功能模块：

1>在串行链路上封装数据包的方法。PPP 既支持数据为 8 位和无奇偶校验的异步模式，还支持面向比特的同步连接。

2>建立，配置和测试数据链路的链路控制协议（LCP：Link Control Protocol）。它允许通信双方进行协商决定某些选项的使用，如是否使用压缩，以及进行最大接收单元大小的通报。

3>针对不同网络层协议的网络控制协议（NCP：Network Control Protocol）。如针对 IP 数据包传输的 IPCP（IP Control Protocol）协议。

PPP 协议定义了在链路层数据包的封装格式，如图 1-21 所示。

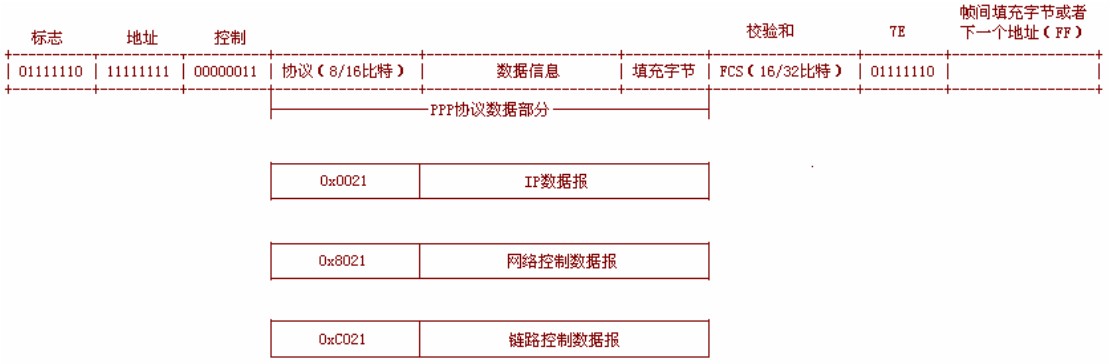


图 1-21 PPP 协议报文封装格式

标志字段：1 字节

PPP 协议每帧均是以一个标志字段开始，该字段值固定为 0x7E。所有 PPP 协议的实现必须检查该字段以进行数据帧的接收同步。在帧与帧之间只需要插入一个标志字段，两个连续的标志字段表示传输了空帧，对于 PPP 协议的实现而言，传输空帧是合法的，该空帧将被丢弃而不造成任何影响。

地址字段：1 字节

该字段值固定为 0xFF。因为对于一个点对点连接，数据传送方向只可能有一个，所以该字段被固定为广播地址。该字段在将来的使用中也可能有其它定义形式。如果一个帧具有一个非法的地址字段值，该数据帧将被丢弃而不进行任何其它操作。

控制字段：1 字节

该字段值固定为 0x03。该字段在将来的使用中可能被定义其它值。如果一个帧具有一个非法的控制字段值，则该数据帧也将被丢弃而不造成任何其它影响。

协议字段：1 字节或者 2 字节

该字段可能为 1 字节，也有可能为 2 字节。该字段值指定了该数据帧所使用的传输协议类型。图 1-20 给出了简单的三种协议值及其对应的数据报类型。该字段值的结构符合 ISO3309 地

址域扩展机制。所有的协议字段值均为奇数，即最低字节的最低位为 1。
该字段值所代表的范围表示不同的含义。
0x0***--0x3***：代表网络层协议（如 IP 协议）报文。
0x4***--0x7***：用于低通量链路连接中的相关协议。
0x8***--0xB***：用于网络控制协议（NCP）报文。
0xC***--0xF***：用于链路控制协议（LCP）报文。

数据信息字段，填充字段：长度由 MRU（最大接收单元）决定
数据信息字段有 0 或多个字节构成，代表协议字段部分指定类型的数据报文。数据字段（包括填充字段，不包括协议字段）的最大长度被称为最大接收单元（MRU：Maximum Receive Unit），该字段默认值一般为 1500 字节。MRU 的值可通过 LCP 协议进行双方协议以使用另一个值。

帧校验序列字段（FCS：Frame Check Sequence）：2 字节（默认）或者 4 字节
该字段用于数据校验。一般为 2 字节长度。4 字节长度的使用可通过双方通过 LCP 协议协商而定，用于某些特殊情况下。
该字段覆盖的计算范围是：地址字段，控制字段，协议字段，数据字段和填充字段。注意由于在数据传输中可能会插入一些辅助位（如异步传输中的起始和结束位，以及同步传输中插入的任何数据位）都不计入校验和的计算。

标志字段：1 字节
同开始处的标志字段，取值固定为 0x7E，该字段与起始的标志字段定义了一帧数据的边界。

帧间填充字节或者下一个地址字段：
如果两个帧的数据是连续的，则该字段应该是下一个地址字段，即取值为 0xFF，之后的封装如同图 1-20 所示。如果两帧数据不是连续的，则帧与帧之间就会插入填充字节。按照传输方式的不同，填充字节取值不同。对于同步传输，在此期间即不断的传输标志字段值 0x7E，直到下一帧数据开始时，则开始传输地址字段值，从而开始下一帧数据的传输。而对于异步传输方式，则在下一帧数据传输之前的空隙当中，应不停的传输 1（称为 mark-hold 状态）直到开始传输下一帧数据同步字段（即标志字段）值 0x7E。

PPP 协议链路操作
为了进行建立在点对点物理连接上的数据传输，连接上的每一段必须首先进行 LCP 数据报文的交换以对链路进行配置和测试。在配置和测试结束之后，根据在此阶段中交换的选项，下一阶段可能需要进行认证。认证是为了保证数据传输的安全性而采取的一种措施，即向对方证明本地确实是授权的一方。认证的方法有两种，这是在配置阶段双方进行选项配置交换时所选取的认证协议决定的。其一是密码认证方式（PAP：Password Authentication Protocol），使用密码认证协议（对应协议字段值为 0xC023）；其二称为查找握手认证方式（CHAP：Challenge Handshake Authentication Protocol），使用 Challenge Handshake Authentication 协议（对应协议字段值为 0xC223）。

在使用 LCP 协议对链路进行测试以及进行相关选项的配置之后，如果成功进入网络协议配置阶段，则需要使用具体的 NCP 协议（如 IPCP）进行相关配置。该阶段配置成功之后，即

可进行具体协议数据报的交互传输。

在配置，维护及最后关闭 PPP 连接过程中，PPP 链路会经历一系列不同的状态，下图（图 1-22）即显示了这些状态之间的转换。

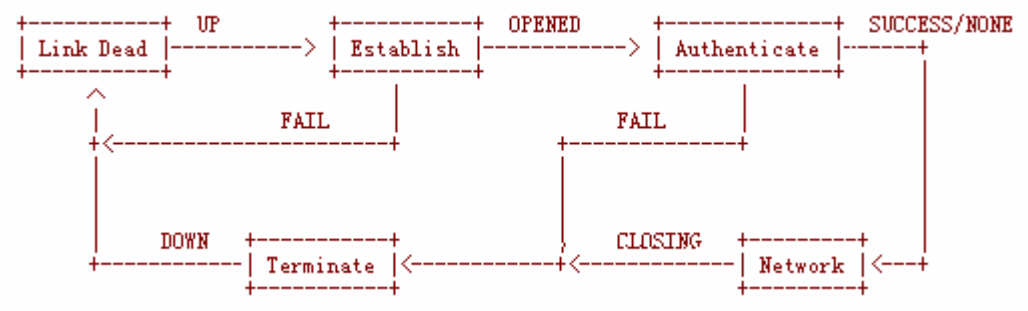


图 1-22 PPP 协议链路状态转换图

1> Link Dead Phase（链路未就绪阶段）

该状态是一个链路的最初以及最终状态。当某个外部事件（如载路探测）表示物理介质层可用时，链路就前进到建立阶段。对于使用调制解调器进行连接的链路，则当调制解调器断开连接时，链路即进入 Link Dead 状态；而对于直接的硬连线连接，则链路处于该状态的时间极其短暂，在此种情况下，链路通常处于建立阶段。

2> Link Establishment Phase（链路建立阶段）

在该阶段，链路控制协议（LCP：Link Control Protocol）被用来建立链路连接。连接的建立是通过交换 LCP 配置数据包完成的。当所有的配置完成后，如果在配置阶段使用了认证选项，则将进入认证阶段，否则直接进入网络协议配置阶段。由此可见，认证阶段是可选的，该阶段是否存在取决于在链路建立阶段中是否选择了认证选项。需要注意的是，所有的选项如果在没有进行明确配置的情况下，都有一个默认值。这就使得用户只需要配置想要改变其默认值的选项，而对于其它选项则可以不进行配置。有关 LCP 配置数据包的格式以及配置的各种选项将在下文详细介绍。

在该阶段中，所有非 LCP 配置数据包均将被丢弃。

3> Authentication Phase（认证阶段，可选）

诚如上文所述，该阶段是可选的。一旦在链路建立阶段中确认使用认证选项，则在认证阶段中，必须使用在链路建立阶段中双方协商的认证协议进行身份认证。目前主要有两种认证协议，其一称为密码认证协议（PAP：Password Authentication Protocol），其二称为查询握手认证协议（CHAP：Challenge Handshake Authentication Protocol）。

密码认证方式通过传输明码方式验证自己的身份，显而易见，此种方式并不保险，该种方式的认证过程是：

1> 被认证的一方发送明码形式的用户名和密码给进行认证的一方。

2> 进行认证的一方通过查询其相关文件确定是否有该用户以及密码是否正确。

3> 进行认证的一方发送认证成功或者失败数据包给被认证的一方。

如果认证成功，则进入网络协议配置阶段，否则进入链路关闭阶段断开链路。

查询握手认证方式使用加密方式进行身份认证。该认证方式也是通过 3 个数据包的交换

进行身份认证（有时称为 3 路握手认证方式）。

1> 进行认证的一方发送查询（challenge）数据包给被认证者。

2> 被认证者使用某种加密算法对查询报文中有效数据部分进行加密后，返回给对方。

3> 进行认证者也使用加密算法对其发送的查询报文中数据部分进行加密，并将计算后结果与被认证者发送过来的计算结果进行比较，如果二者相符，则表示认证成功，否则认证失败。

此种认证方式的关键在于加密算法中使用的密匙。该密匙应仅限于进行认证者和被认证者知道，否则此种认证方式将失去其意义。

认证一般是单向的，当然也可采用双向认证，此时一般需要两个密匙，分别用于两个方向上的加密算法。否则攻击者会伪造认证者发送 Challenge 数据包，获得反馈数据包，从而从中分析出有效信息。

4> Network-layer Protocol Phase（网络协议配置阶段）

当成功进入该阶段，则根据所使用网络层协议的不同将需要使用具体的 NCP 协议进行有关选项的配置（如对于 IP 协议则对应的 NCP 协议为 IPCP：IP Control Protocol）。为讨论方便，以下即以 IPCP 协议为依据进行说明。

IPCP 配置包格式同 LCP 配置包格式，LCP 配置包格式及其选项下文中将有详细说明。以下给出二者不同之处。

A． PPP 协议在链路层使用类 HDLC 格式，对于 IPCP 数据包而言，其首部中协议字段值为 0x8021。

B．代码域有效值为 1~7。其它代码值将被认为为无效的，会引起 Code-Rejects 数据包的发送。

C．IPCP 数据包只在网络协议配置阶段有效。

D． IPCP 使用不同于 LCP 的选项。

对于 IPCP，具有如下选项类型：

1． IP-Addresses（已过时，被 IP-Address 选项替代）

2． IP-Compression-Protocol

3． IP-Address

5> Link Termination Phase（链路关闭阶段）

PPP 可以随时终止链路连接。链路关闭可能有如下原因造成：载路丢失，认证失败，空闲定时器超时或者管理员发出终止命令。

LCP 协议被用于在此阶段进行数据包交互以关闭链路连接。当使用 LCP 协议进行关闭链路连接数据包交换后，具体实现必须发送信号给物理层以完成实际上的连接关闭。

发送者在发送 Terminate-Request 数据包后，等待对方 Terminate-Ack 数据包，一旦接收到该数据包或者定时器超时，则断开链路。接收端在接收到 Terminate-Request 数据包后，应该等待对方断开连接，在发送 Terminate-Ack 数据包给对方后，必须等待一个定时器超时时间后方能断开本地连接。

此状态结束后，进入 Link Dead 状态。

在此阶段中，所有非 LCP 配置包均将被丢弃。

以下将详细介绍各阶段所使用数据包的格式。

1．链路建立/关闭阶段所使用数据包格式（LCP 数据包）

LCP 数据包用于链路建立/关闭阶段，LCP 协议从功能上可分为三类配置包。

1>用于建立和配置链路连接的 LCP 数据包

Configure-Request，Configure-Ack，Configure-Nak，Configure-Reject

2>用于终止链路连接的 LCP 数据包

Terminate-Request，Terminate-Ack

3>用于维护和测试链路连接的 LCP 数据包

Code-Reject，Protocol-Reject，Echo-Request，Echo-Reply，Discard-Request

为简单起见，LCP 数据包中无版本字段。一个正确的 LCP 协议的实现应当对任何无效的数据包进行有效的响应。整个 LCP 数据包作为 PPP 封装形式中的信息数据字段（见前文中说明）。此时对应的帧协议字段值为 0xC021。

下图（图 1-23）显示了 LCP 数据包通用格式。

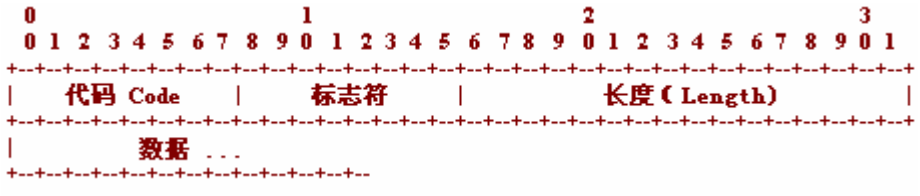


图 1-23 LCP 数据包通用格式

下图（图 1-24）显示了 LCP 数据包在整个数据帧中的封装形式。

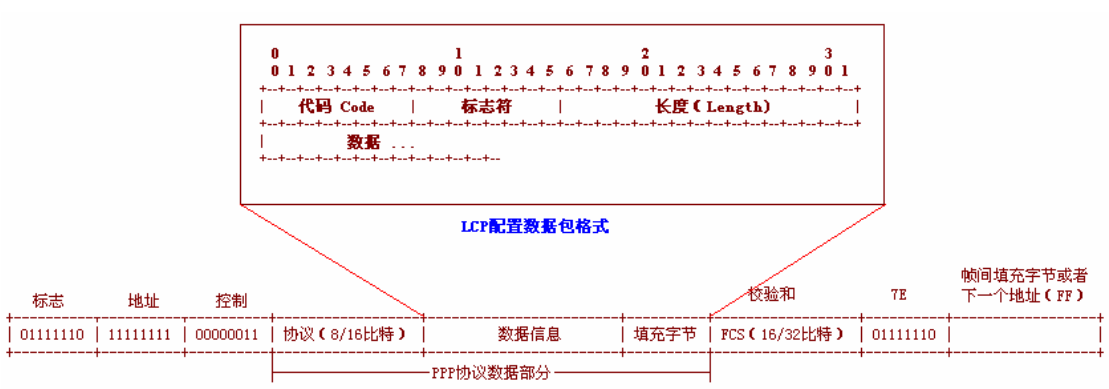


图 1-24 LCP 数据包在整个数据帧中的位置

代码字段：1 字节

该字段表示 LCP 配置包的类型。当一个未知包类型被接收到时，需要响应一个 Code-Reject 数据包。下表显示了该字段值对应的不同包类型。

表 1-11 代码字段值及其意义

代码字段值（Code）	包类型
1	Configure-Request
2	Configure-Ack
3	Configure-Nak
4	Configure-Reject

5	Terminate-Request
6	Terminate-Ack
7	Code-Reject
8	Protocol-Reject
9	Echo-Request
10	Echo-Reply
11	Discard-Request

标志符（Identifier）字段：1 字节
该字段用于匹配查询和应答报文。如对于 Configure-Request 查询报文，其对应的应答报文（如 Configure-Ack）应该返回相同的标志符值以表示这个应答具体是针对哪个查询报文。因为接收端在接收到一个应答之前不会发送另外的查询报文，所以这种匹配关系比较简单，即只对前一个发送的查询报文进行匹配，无须记录多个查询报文的标志符值。如果接收端接收到一个无效的应答报文则应当丢弃。

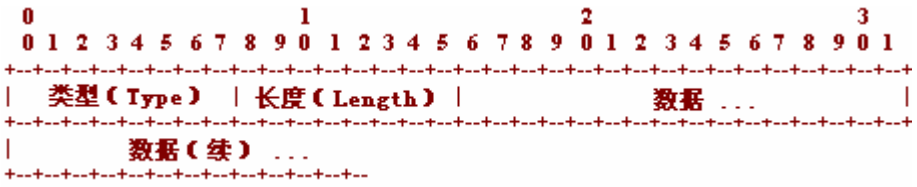
长度字段：2 字节
该字段值表示 LCP 数据包的长度，该长度包括代码字段，标志符字段，其本身及其之后的数据部分。在超出长度字段所表示的范围之外的数据被认为是填充数据。如果接收端接收到一个无效长度字段值的数据包，则将其丢弃。

数据字段：可变长度
对于 LCP 配置数据包而言，该字段大多表示 LCP 选项部分。其具有内部结构，在下文中阐述 LCP 选项时将给与介绍。而对于其它类型数据包，该字段也表示有效信息。

以下将根据不同的代码字段值分别具体阐述 LCP 数据包类型。
1>Configure-Request：配置请求数据包
2>Configure-Ack：配置应答数据包
3>Configure-Nak：配置否定应答数据包
4>Configure-Reject：配置拒绝数据包
这四种数据包用于链路建立阶段进行选项配置，其封装格式相同，如下图所示。

对于这四种数据包类型，通用格式中数据字段此时即表示通信双方需要交换的选项部分。选项部分具有其内部结构。LCP 选项配置数据包允许通信双方改变连接中一些默认特性。如果一个选项在 Configure-Request 没有被指定，则使用其默认值。注意每个选项都有一个默认值对应。

下图显示了选项字段的内部结构。



下图（图 1-25）显示了 LCP 选项在整个数据帧中的位置。

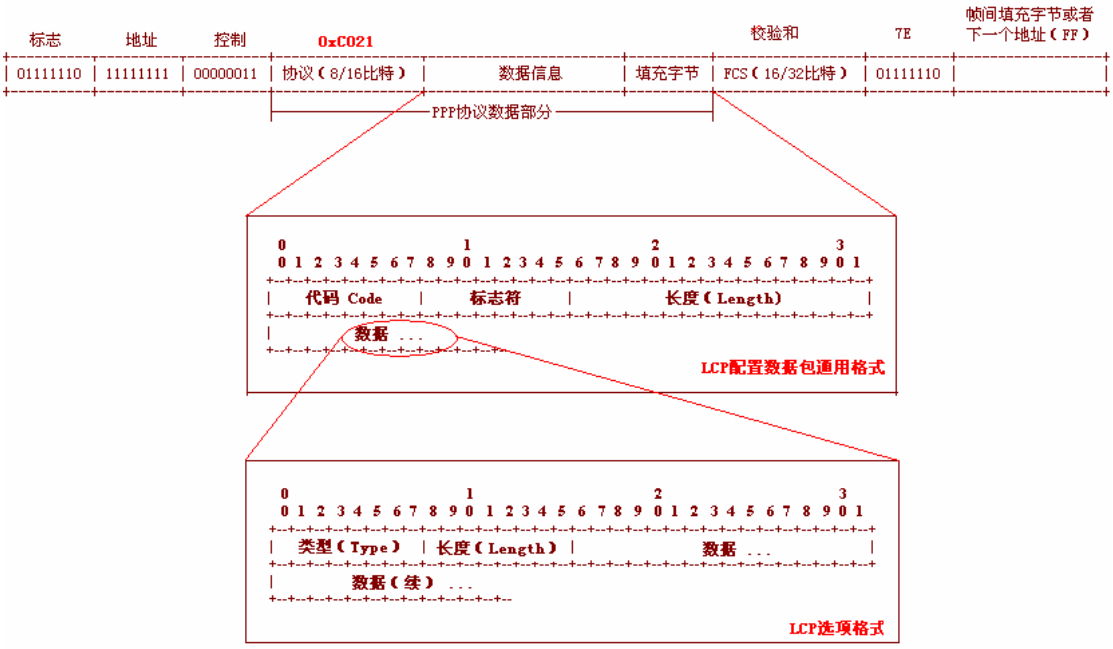


图 1-25 LCP 选项封装在整个数据帧中的位置

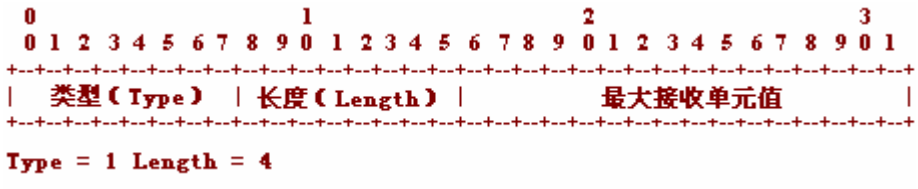
类型（Type）字段：1 字节
该字段表示选项的具体类型。可取如下值。

类型值	意义
---	---
0	保留
1	最大接收单元
2	认证协议
3	链路质量（稳定性）检测协议
4	幻数协商
5	协议字段压缩
6	地址和控制字段压缩

长度（Length）字段：1 字节
该字段表示选项部分总长度：包括类型字段，其本身及之后的数据部分。

数据部分：长度可变
该字段的长度由长度字段值决定。对于不同的类型值（即对于不同的选项配置）该字段的格式有所不同，以下根据不同的类型值分别阐述不同的选项。
A． 最大接收长度选项（MRU：Maximum Receive Unit）

对于该选项，此时数据部分为最大接收单元的协商值。具体格式如下：

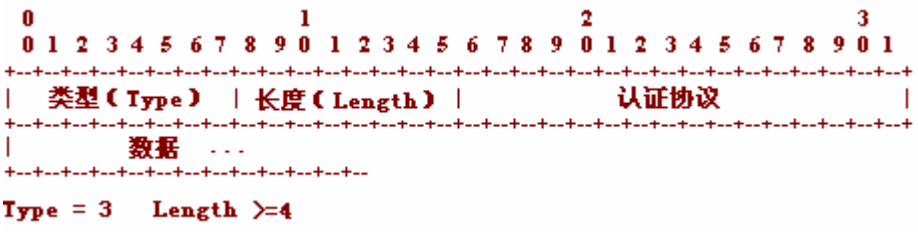


最大接收单元值表示 PPP 封装格式中数据部分和填充部分的数据总长度。

B．认证协议协商

这个选项协商在需要进行认证时使用的认证协议：PAP 或者 CHAP。缺省的情况是不进行认证。一个实现不应当在单个 Configure-Request 数据包中包括多个认证协议，而是分多个数据包进行协商，第一次使用最期望的认证协议，协商失败后，在选择其它认证协议继续进行协商。如果一个 Configure-Request 数据包中包括认证协议的选项，则表示一端希望进行身份认证，则如果认证协议协商成功，之后将进入认证阶段。认证可以是单向或者双向的，对于双向认证，并不要求两个方向上使用相同的认证协议，换句话说，对于连接的两端，A 端对 B 端进行认证时，可以使用 PAP 协议，而 B 端对 A 端进行认证，可以使用 CHAP 协议。

下图显示了认证协议选项的封装格式。



认证协议字段：2 字节

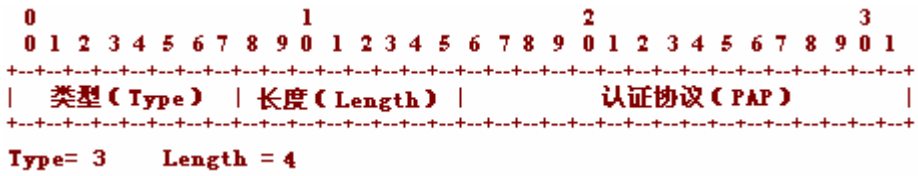
表示双方进行协商的认证协议类型。

协议字段值	所表示协议
0xC023	PAP
0xC223	CHAP

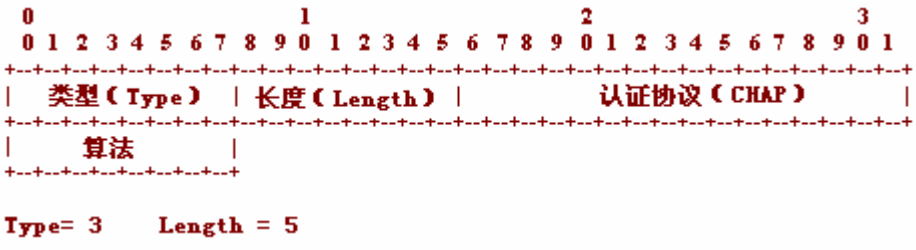
数据字段部分：长度可变

该字段具体长度有长度字段值可推出。

该字段表示的意义有具体进行协商的认证协议决定。对于 PAP 认证协议其无数据部分，即格式如下：



对于 CHAP 协议，则数据部分仅有一个字节，表示所使用的加密算法。此时的格式如下所示。



算法字段：1 字节

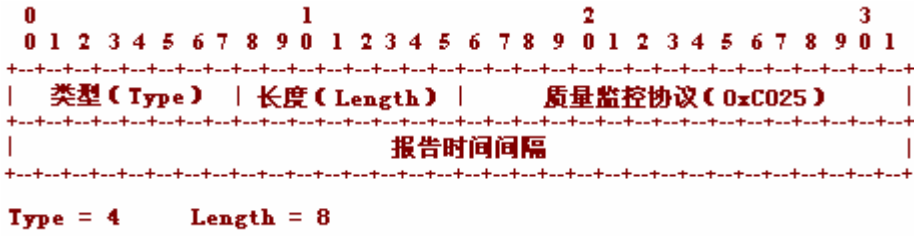
该字段表示所采用的加密算法。一般取值为 5，表示使用 MD5 加密算法。

C．链路质量检测协议协商

链路连接通道很少有完美的，数据包可能因为各种原因而被丢弃或破坏。

在某些链路连接中，可能需要知道何时以及以多大的频率丢弃部分数据包。这个过程被称为链路连接质量监控。实现链路连接质量监控的方式有很多种，PPP 协议使用 Link-Quality-Report 数据包进行监控。PPP 只提供一种检测机制，并没有提供如果在发现数据包被丢弃或被破坏时采取何种措施，这是实现相关的。质量监控使用一系列计数器来计数各种类型数据包，这些计数器的定义是依据 SNMP MIB-II 实现的。

本选项提供了一种机制用于双方协议采用何种监控协议。在通常情况下是不需要对本选项进行协商的，当用于确定对方是使用 Link-Quality-Report 数据包而非其它形式数据包或者协商 Link-Quality-Report 发送间隔时间时，本选项的协商才是必要的。如同认证一样，质量监控可以是双向的。在不同方向上可以使用不同的监控协议。该选项对应的格式如下所示。



质量监控协议字段：2 字节

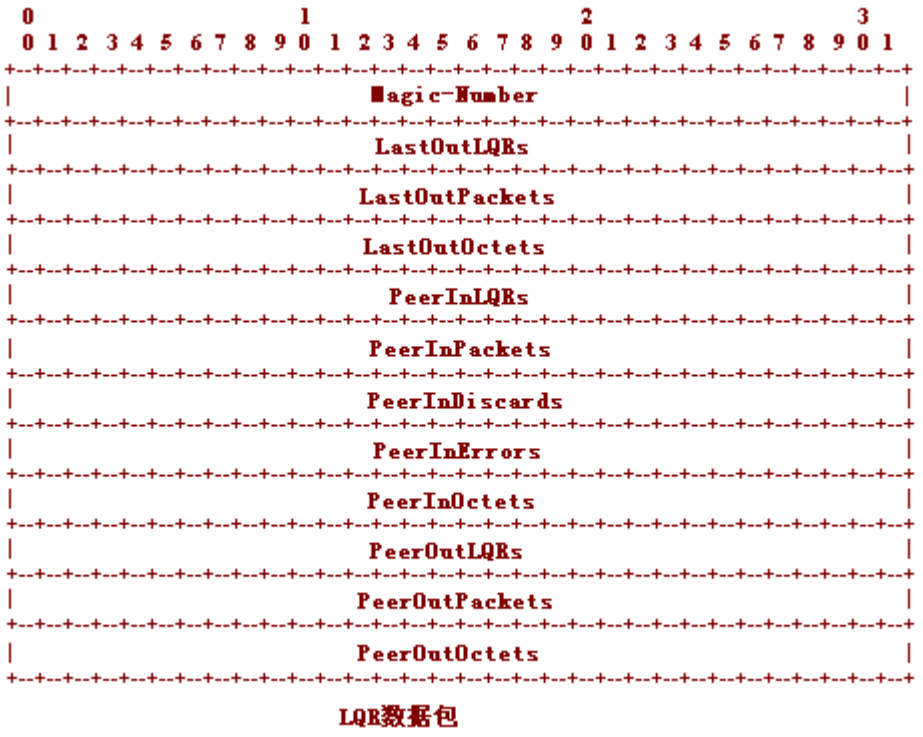
该字段即双方进行协商使用的质量监控协议。该字段取值一般为 0xC025，表示使用“链路连接质量报告”（Link Quality Report）协议。

报告时间间隔：4 字节

对于 LQR 协议，该字段表示两个监控数据包发送之间的最大间隔时间。注意这个时间是对对方发送 LQR 数据包的间隔时间的要求。如果该字段设置为 0，则表示对方无需维护一个定时器，此时对方每当接收到一个本地发送的 LQR 数据包时，即发送一个 LQR 数据包给本地。即对方 LQR 数据包的发送是由本地 LQR 数据包触发的。

Link-Quality-Report 数据包封装格式，如图 1-26 所示。

以下字段的含义是从该数据包接收者的角度解析的。



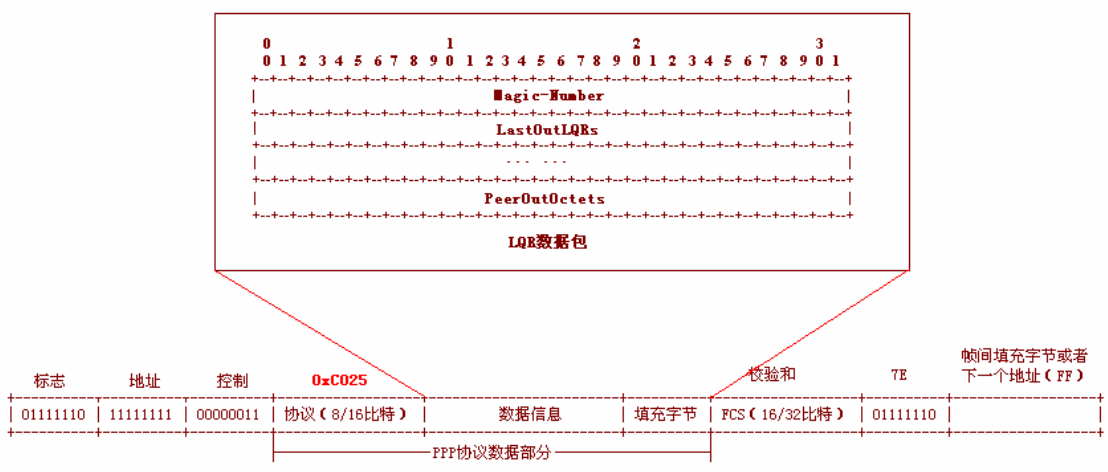


图 1-26 LQR 数据包格式及其在整个数据帧中的位置

以下这些字段（图 1-27）并不进行传输，只是逻辑上附加到以上字段之后。一般在实现中用于本地信息统计。

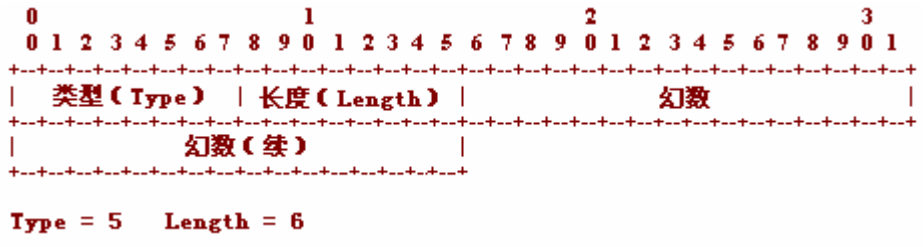
图 1-27 LQR 统计信息（不进行传输）

D. 幻数协商

该配置选项提供了一个检测环路以及其它链路连接异常的机制。默认的，幻数不进行协商，当使用到幻数时，直接用 0 代替。

幻数的选择推荐使用随机数。而且应每次都具有唯一性，即每次应使用不同的幻数。

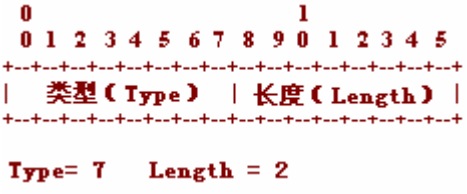
幻数协商选项的格式如下：



E. 协议字段压缩选项

该选项用于双方协商对协议字段进行压缩。默认的，所有实现必须传输使用两字节协议的数据包。对于某些协议而言，可以将其压缩为一个字节。该选项用于本端通知对端本地可以接收单个字节的协议字段，从而减少数据传输量。如果双方协商成功，则实现上应该是既可以接收单个字节协议字段的数据包，也可以接收原先双字节协议字段的数据包。注意对于 LCP 配置包而言，不使用协议字段压缩，以避免不必要的麻烦。当使用协议字段压缩时，FCS 值的计算即使用压缩后的协议字段值，而非原先的双字节协议字段值。

协议字段压缩选项格式如下：

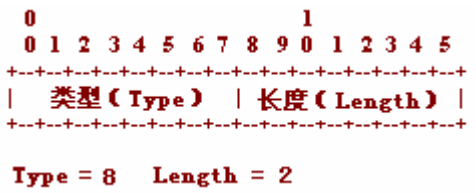


F. 地址和控制字段压缩选项

该选项用于对地址和控制字段进行压缩。如上文中所述，对于 PPP 协议封装格式而言，地址和控制字段值通常为固定值，所以在双方进行协商的条件下，可以对这两个字段进行压缩，从而减少数据传输量。

同样的这种压缩不可以使用在 LCP 配置数据包上。当使用地址和控制字段压缩时，FCS 值的计算是使用压缩后的地址和控制字段值。

该选项格式如下：



Configure-Request 数据包对于所有要进行协商的选项都包含在选项字段部分。在接收到一个 Configure-Request 数据包后，接收端必须进行应答。有三种应答数据包类型：

1．Configure-Ack：完全同意发送端发送的 Configure-Request 数据包中所提出的选项值。

2．Configure-Nak：不完全同意 Configure-Request 数据包中所提供的选项值，Configure-Nak 数据包中将包括那些不同意的选址类型以及对端提供的该选项的可选值。本端必须选择这些可选值中其一继续发送 Configure-Request 数据包进行协商直至最后双端达成妥协。

3．Configure-Reject：对于 Configure-Request 中提供的某些选项无法进行辨别。即某些选项类型不正确，当然此时选项值更无从谈起。Configure-Reject 数据部分将包含这些不认识的选项类型，如此发送 Configure-Request 数据包的一端可以从接收到的 Configure-Reject 数据包中获知对方那些选项无法辨别，从而在下一个 Configure-Request 数据包中剔除这些选项，从而直接使用默认值；或者是本地误传某些不正确选项，从而在下次得到改正。

5>Terminate-Request：请求终止连接数据包

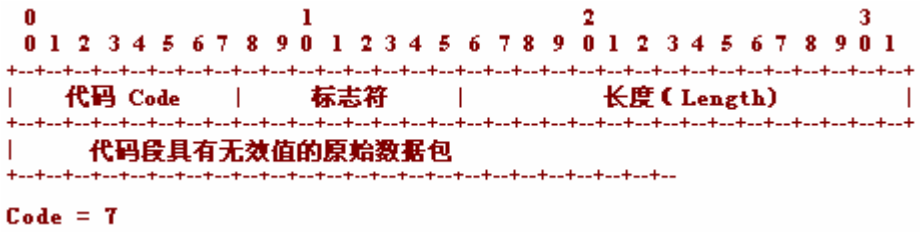
6>Terminate-Ack：终止连接应答数据包

这两个配置包类型使用相同的数据包格式，即上文中给出的通用格式。不过代码字段值对于 Terminate-Request 配置包而言是 5，而 Terminate-Ack 为 6。

这两个配置包类型用于关闭连接。一个具体的实现如果想关闭链路连接，则首先应该发送一个 Terminate-Request 数据包给对端，同时设置一个超时定时器，在没有接收到对端响应的 Terminate-Ack 数据包之前，应该实行超时重传。在接收到 Terminate-Ack 数据包后或者在进行一定次数的重传后，本端将认为 OSI 网络栈下层（此处即物理层）已完成关闭。

7>Code-Reject：代码无效应答数据包

顾名思义，该类型 LCP 数据包表示对方接收到一个本地发送的具有无效代码字段的数据包，此时对方将发送该类型数据包给本地。Code-Reject 数据包的格式如下：



Code = 7

其中原始数据包字段部分只包括原数据包的信息字段部分，不包括链路层帧头和帧尾的 FCS 字段。另外原始数据包的长度如果需要会被截断为 MRU 所允许的长度。

8>Protocol-Reject：协议无效应答数据包

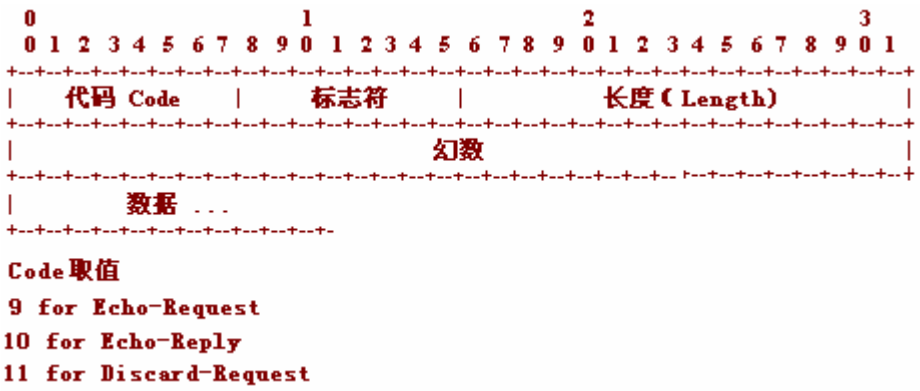
该类型数据包表示对方之前发送的数据包具有无效的协议字段值。

这通常发生在对方在试图使用一个新的传输协议进行数据传输。

该类型数据包的格式如下：

其中原始数据包部分所包含的数据同 Code-Reject 对应字段。

9>Echo-Request：请求回复数据包
10>Echo-Reply：回复应答数据包
11>Discard-Request：请求丢弃数据包
这两种类型的数据包用于测试和检测链路连接的状态。其共同使用如下数据包格式：



在接收到 Echo-Request 数据包时，必须相应一个 Echo-Reply 数据包。标志符字段用于匹配。Echo-Request 和 Echo-Reply 数据包用于检测环路，测试，链路质量检测，传输效率检测等等各种应用。其中幻数字段占据 4 个字节，该字段主要用于环路检测中。如果在链路建立阶段没有使用幻数协商选项，则该字段值必须设置为 0。

接收端在接收到一个 Discard-Requst 数据包后立即丢弃而不对其进行任何处理。该种类型数据包用于环路检测，测试，传说效率检测等等应用中。

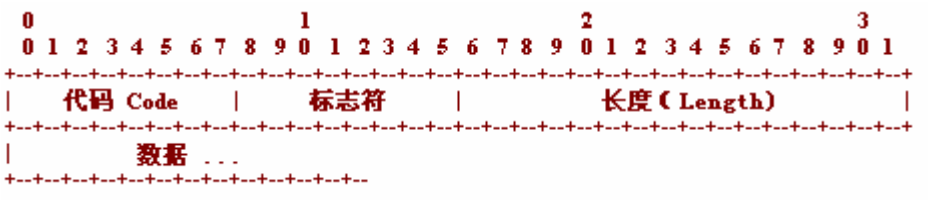
封装中数据部分占据 0 个或多个字节（由长度字段可计算出），其中包括发送端发送的原始数据，这个字段可以包含任何二进制数据。

2．认证阶段所使用数据包封装格式

根据在链路建立阶段使用协商的认证协议的不同，在该阶段将使用不同的协议进行认证。以下分别以 PAP 协议和 CHAP 协议进行讨论。二者所使用数据包的通用格式同 LCP 配置数据包，对于 PAP 协议数据包而言，帧协议字段值为 0xC023，对于 CHAP 协议数据包而言，帧协议字段值为 0xC223，而对于 LCP 配置数据包而言，帧协议字段值为 0xC021。而对于下文中网络协议配置阶段数据包而言，其帧协议字段值为 0x8021.

1． 密码认证协议（PAP）

如下（图 1-28）显示了使用该协议进行认证的数据包格式及其在整个数据帧中的位置。



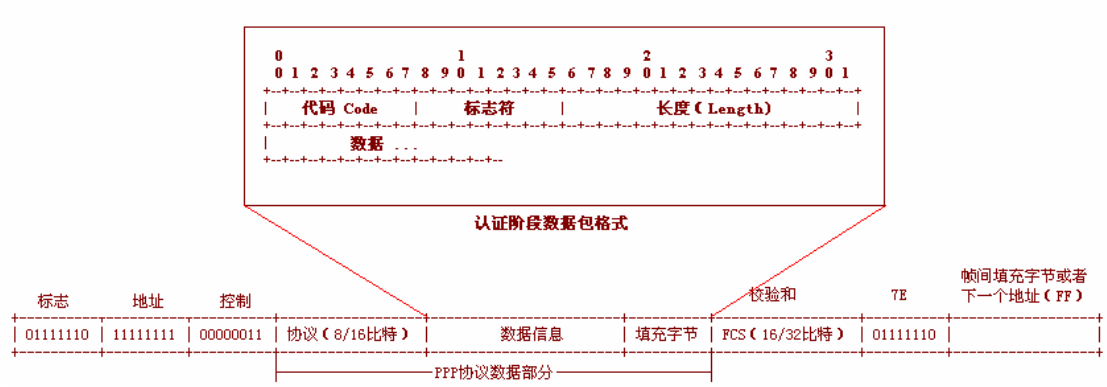


图 1-28 认证数据包格式以及其在整个数据帧中的位置

其中各字段所表示的含义同 LCP 配置数据包。

对于认证数据报而言，代码字段可取如下值：

代码字段值	含义
1	Authentication-Request
2	Authentication-Ack
3	Authentication-Nak

对于不同的代码字段取值，其数据部分代表不同的含义。

1> Authentication-Request

该数据包由被认证的一方主动发送，该数据包同时也启动了 PAP 认证的三路握手认证的第一步，即有被认证的一方发送明码用户名和密码。

进行认证的一方在接收到该数据包后，必须对其进行响应。响应数据包可以是 Authentication-Ack（认证成功）或者 Authentication-Nak(认证失败)。

Authentication-Request 数据包格式如下：

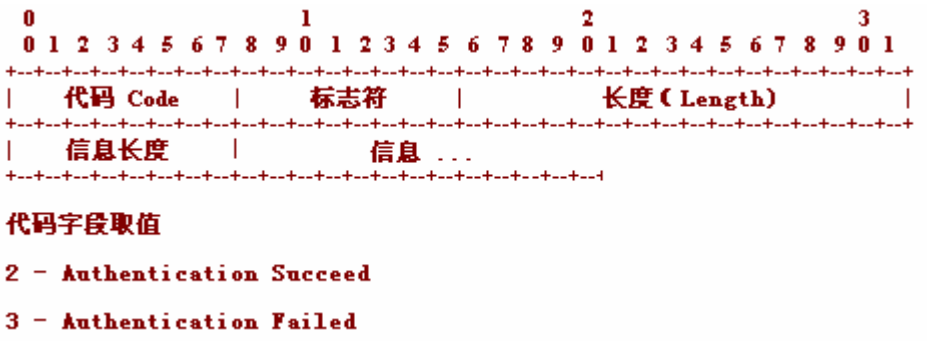
2> Authentication-Ack

3> Authentication-Nak

这两种数据包是对 Authentication-Request 成功和失败时的应答。

当 Authentication-Request 数据包中的 ID 和密码值有效时，认证者即发送 Authentication-Ack 数据包进行应答表示认证成功。否则发送 Authentication-Nak 数

据包表示认证失败。二者数据包格式如下：



信息长度字段：1 字节
该字段表示其后所跟的信息字段的长度值。

信息字段：长度可变
该字段可以有 0 个或者多个字节，实现相关。一般为人可读的信息。

2. 查询握手认证协议（CHAP）

CHAP 协议所用的数据包通用封装格式及其在整个数据帧中的位置同 PAP 协议。不过对于 CHAP 协议而言，代码字段取值所表示的意义不同。

代码字段值	含义
1	Challenge
2	Response
3	Success
4	Failure

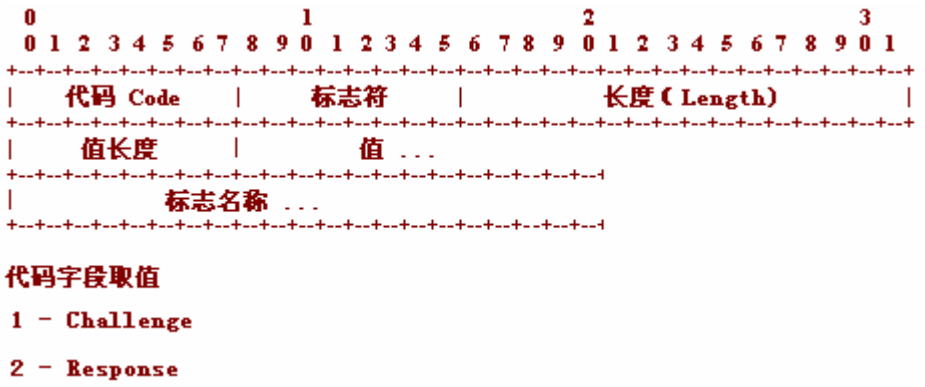
其它标志符字段，长度字段，数据字段意义同 PAP 协议。其中数据字段值的含义同代码字段取值相关。

1> Challenge

2> Response

Challenge 数据包用于启动 CHAP 认证的三路握手阶段。不同于 PAP 的是，第一个数据包（即 Challenge 数据包）由认证者发送（PAP 协议第一个 Authentication-Request 数据包有被认证者发送）。Challenge 数据包除了正常情况下在认证阶段发送外，也可以在网络协议配置阶段发送以确定链路连接没有被改变。当被认证者接收到一个 Challenge 数据包后，其必须发送一个 Response 数据包进行响应并同时返回 Challenge 数据包中包含的经过加密算法计算后的数据给认证者。当认证者接收到 Response 数据包后，其将 Response 中返回的加密数据与本地计算的加密数据进行比较，如果相符，表示认证成功，此时发送 Success 数据包给被认证者，否则表示认证失败，此时发送 Failure 数据包给被认证者。

Challenge 和 Response 数据包采用如下格式。

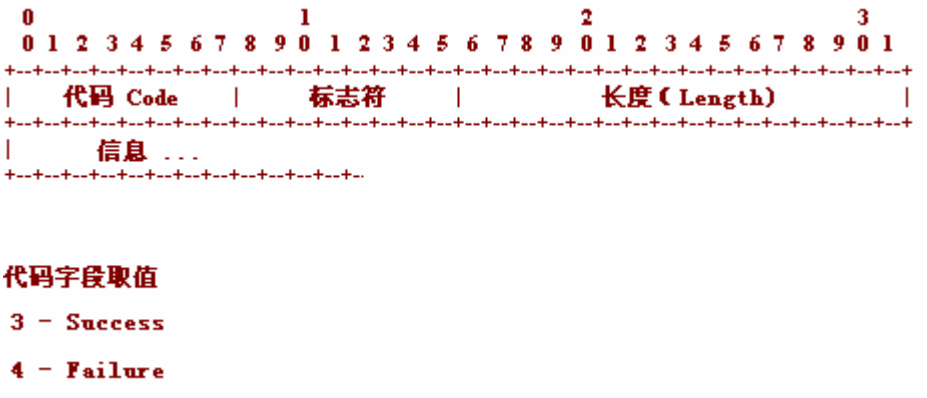


值长度字段：1 字节
该字段指出其后值字段的数据长度。

值字段：长度可变
该字段长度由值长度字段标明。被认证使用预先协商的加密算法进行加密：算法所涉及的域：标志符字段，算法所使用的密匙，值字段所包含的所有数据。计算得到的结果将作为 Response 数据包的值字段返回给认证者。

标志名称字段：长度可变
该字段的长度由长度字段值可计算出。
该字段所表示的意义是发送该数据包的系统标志名称。原则上对此字段的取值没有限制。例如可以是 ASCII 字符串。该字段不可以 NULL 字符或者换行符结束。
由于 CHAP 协议可用于多种不同系统的认证，故该字段值可以作为密匙数据库中的一个关键字用于寻址相应的密匙。

3> Success
4> Failure
如果 Response 数据包中加密数据与本地计算出的值相符，表示认证成功，此时发送 Success 数据包给被认证者，否则表示认证失败，此时发送 Failure 数据包给被认证者。二者使用如下数据包格式。



信息字段：长度可变
该字段长度由长度字段可计算出。
该字段可以有 0 个或者多个字节，其内容是实现相关的。可以是任何人可读形式的数据。

3．网络配置阶段数据包格式

通用格式同链路建立阶段和认证阶段数据包格式，但此时帧协议字段值为 0x8021。

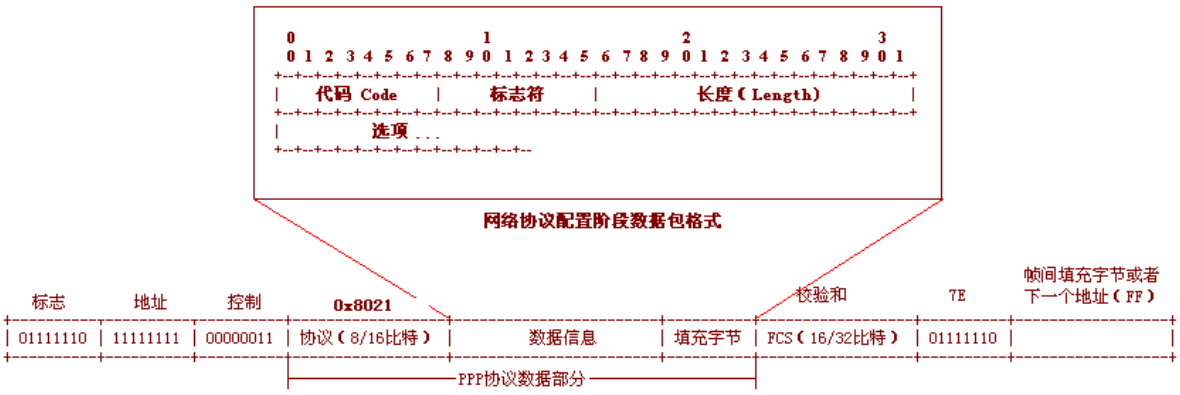


图 1-29 网络协议配置数据包在整个数据帧中的位置

在该阶段对于不同的协议将使用不同网络控制协议，本文只讨论针对 IP 协议进行配置的 IPCP（IP Control Protocol）协议。

对于 IPCP，共有三个配置选项。

1> IP-Addresses

2> IP-Compression-Protocol

3> IP-Address

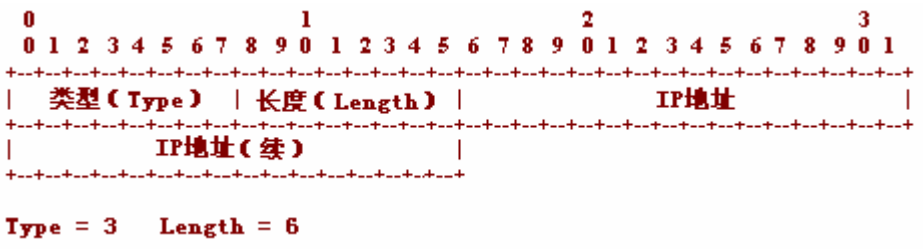
其中 IP-Addresses 选项已过时，被 IP-Address 选项取代。以下介绍 IP-Compression-Protocol 和 IP-Address 选项。

IP-Address 选项

该配置选项提供了一种方式用于协商本地所使用的 IP 地址。该选项允许 Configure-Request 数据包发送者提供其想使用的 IP 地址；或者要求对方提供一个 IP 地址给本地使用，此时相应的地址字段设置为 0 即可。

默认情况下，将不分配 IP 地址。

该选项使用如下数据包格式。



IP 地址字段：4 字节

该字段如果不为 0，则表示该 Configure-Request 数据包（注意选项的配置是在 Configure-Request 数据包中指定的，这对于网络协议配置阶段是一样的）的发送者指定了其想使用的 IP 地址，否则（该字段为 0）则表示需要对方返回一个 IP 地址赋予本地使用。

IP-Compression-Protocol 选项

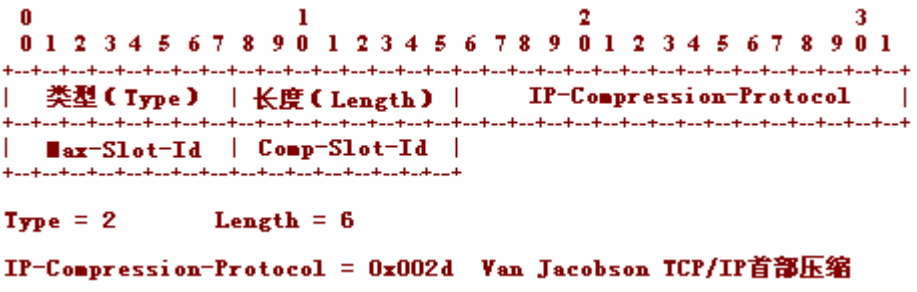
Van Jacobson TCP/IP 首部压缩可以将 TCP/IP 首部长度（最少 40 个字节）减少到 3 个字节。这对于慢速串行线路而言，大大减少了数据传输量。

IP-Compression-Protocol 配置选项用于表示可以解析压缩数据包的能力。如果一端具有这种能力，则在 Configure-Request 数据包中应该包括该选项，从而使得对方可以发送经过压缩后的数据包以优化效率。注意这个选项可以是单向的或者是双向的。

在网络协议配置阶段完成后，当正式传输 IP 数据包时，根据在网络协议配置阶段所使用的压缩选项，帧协议字段可以有如下取值：

帧协议字段值	含义
0x0021	普通 IP 数据包，非压缩
0x002D	压缩数据包。TCP/IP 首部被压缩后首部替代
0x002F	非压缩数据包，IP 首部协议字段（表示传输层协议）是一个槽标志符（slot identifier）

使用 Van Jacobson TCP/IP 首部压缩的 IP-Compression-Protocol 选项的数据包格式如下：



Max-Slot-Id：1 字节

该字段表示最大槽标志符。注意计数是从零开始的，所以该字段值比槽位数小 1，即槽标志符从 0 计数到 Max-Slot-Id。

Comp-Slot-Id：1 字节

该字段表示槽标志符字段是否可以被压缩。取值如下：

0：槽标志符不可被压缩

1：槽标志符可以被压缩

对于 IPCP，推荐如下选项：

1> IP-Compression-Protocol---至少具有 4 个槽位，通常 16 个槽位。

2> IP-Address：仅使用在拨号线路。

```
/* include/linux/ppp.h ************************************************************/
#ifndef _LINUX_PPP_H
#define _LINUX_PPP_H

/* definitions for kernel PPP module
    Michael Callahan <callahan@maths.ox.ac.uk>
    Nov. 4 1993 */

/* how many PPP units? */
#define PPP_NRUNIT         4

#define PPP_VERSION    "0.2.7"

/* line discipline number */
#define N_PPP                  3

/* Magic value for the ppp structure */
#define PPP_MAGIC 0x5002

//获取配置标志位
#define    PPPIOCGFLAGS     0x5490  /* get configuration flags */
//设置配置标志位
#define    PPPIOCSFLAGS     0x5491  /* set configuration flags */
//获取异步控制字符映射图
#define    PPPIOCGASYNCMAP   0x5492  /* get async map */
//设置异步控制字符映射图
#define    PPPIOCSASYNCMAP    0x5493  /* set async map */
#define    PPPIOCGUNIT       0x5494  /* get ppp unit number */
#define PPPIOCSINPSIG        0x5495  /* set input ready signal */
#define PPPIOCSDEBUG       0x5497  /* set debug level */
#define PPPIOCGDEBUG       0x5498  /* get debug level */
#define PPPIOCGSTAT   0x5499  /* read PPP statistic information */
#define PPPIOCGTIME   0x549A /* read time delta information */
#define    PPPIOCGXASYNCMAP 0x549B /* get async table */
#define    PPPIOCSXASYNCMAP 0x549C  /* set async table */
#define PPPIOCSMRU    0x549D /* set receive unit size for PPP */
#define PPPIOCRASYNCMAP      0x549E  /* set receive async map */
#define PPPIOCSMAXCID        0x549F /* set the maximum compression slot id */

/* special characters in the framing protocol */
#define    PPP_ALLSTATIONS     0xff /* All-Stations broadcast address */
#define    PPP_UI          0x03       /* Unnumbered Information */
#define PPP_FLAG       0x7E       /* frame delimiter -- marks frame boundaries */
#define PPP_ADDRESS0xFF       /* first character of frame      <--    (may be      */
```

```
#define PPP_CONTROL       0x03       /* second character of frame    <-- compressed)*/
#define    PPP_TRANS  0x20       /* Asynchronous transparency modifier */
#define PPP_ESC           0x7d       /* escape character -- next character is
                          data, and the PPP_TRANS bit should be
                          toggled. PPP_ESC PPP_FLAG is illegal */

/* protocol numbers */
#define PROTO_IP            0x0021
#define PROTO_VJCOMP      0x002d
#define PROTO_VJUNCOMP 0x002f

/* FCS support */
#define PPP_FCS_INIT      0xffff
#define PPP_FCS_GOOD      0xf0b8

/* initial MTU */
#define PPP_MTU             1500

/* initial MRU */
#define PPP_MRU             PPP_MTU

/* flags */
#define SC_COMP_PROT    0x00000001    /* protocol compression (output) */
#define SC_COMP_AC 0x00000002    /* header compression (output) */
#define    SC_COMP_TCP    0x00000004    /* TCP (VJ) compression (output) */
#define SC_NO_TCP_CCID 0x00000008    /* disable VJ connection-id comp. */
#define SC_REJ_COMP_AC 0x00000010    /* reject adrs/ctrl comp. on input */
#define SC_REJ_COMP_TCP      0x00000020    /* reject TCP (VJ) comp. on input */
#define SC_ENABLE_IP      0x00000100    /* IP packets may be exchanged */
#define SC_IP_DOWN  0x00000200    /* give ip frames to pppd */
#define SC_IP_FLUSH  0x00000400    /* "next time" flag for IP_DOWN */
#define SC_DEBUG      0x00010000    /* enable debug messages */
#define SC_LOG_INPKT      0x00020000    /* log contents of good pkts recvd */
#define SC_LOG_OUTPKT  0x00040000    /* log contents of pkts sent */
#define SC_LOG_RAWIN     0x00080000    /* log all chars received */
#define SC_LOG_FLUSH     0x00100000    /* log all chars flushed */

/* Flag bits to determine state of input characters */
#define SC_RCV_B7_0 0x01000000    /* have rcvd char with bit 7 = 0 */
#define SC_RCV_B7_1 0x02000000    /* have rcvd char with bit 7 = 0 */
#define SC_RCV_EVNP       0x04000000    /* have rcvd char with even parity */
#define SC_RCV_ODDP       0x08000000    /* have rcvd char with odd parity */

#define    SC_MASK            0x0fffffff /* bits that user can change */
```

```
/* flag for doing transmitter lockout */
#define SC_XMIT_BUSY    0x10000000    /* ppp_write_wakeup is active */

/*
 * This is the format of the data buffer of a LQP packet. The packet data
 * is sent/received to the peer.
 */
//Link Quality Protocol 协议数据报文信息统计结构。
//该结构用于在 PPP 链路上进行数据传输。
struct ppp_lqp_packet_hdr {
   unsigned long        LastOutLQRs;/* Copied from PeerOutLQRs      */
   unsigned long        LastOutPackets; /* Copied from PeerOutPackets   */
   unsigned long        LastOutOctets;      /* Copied from PeerOutOctets      */
   unsigned long        PeerInLQRs;   /* Copied from SavedInLQRs      */
   unsigned long        PeerInPackets;/* Copied from SavedInPackets     */
   unsigned long        PeerInDiscards; /* Copied from SavedInDiscards  */
   unsigned long        PeerInErrors;   /* Copied from SavedInErrors      */
   unsigned long        PeerInOctets;  /* Copied from SavedInOctets      */
   unsigned long        PeerOutLQRs;/* Copied from OutLQRs, plus 1    */
   unsigned long        PeerOutPackets; /* Current ifOutUniPackets, + 1  */
   unsigned long        PeerOutOctets;      /* Current ifOutOctets + LQR      */
   };

/*
 * This data is not sent to the remote. It is updated by the driver when
 * a packet is received.
 */
//该结构用于系统信息统计。
struct ppp_lqp_packet_trailer {
   unsigned long        SaveInLQRs;  /* Current InLQRs on reception    */
   unsigned long        SaveInPackets;      /* Current ifInUniPackets      */
   unsigned long        SaveInDiscards; /* Current ifInDiscards           */
   unsigned long        SaveInErrors;  /* Current ifInErrors          */
   unsigned long        SaveInOctets;  /* Current ifInOctects         */
};

/*
 * PPP LQP packet. The packet is changed by the driver immediately prior
 * to transmission and updated upon reception with the current values.
 * So, it must be known to the driver as well as the pppd software.
 */

struct ppp_lpq_packet {
```

```
    unsigned long                   magic;     /* current magic value             */
    struct ppp_lqp_packet_hdr hdr;  /* Header fields for structure    */
    struct ppp_lqp_packet_trailer tail;     /* Trailer fields (not sent)       */
};

/*
 * PPP interface statistics. (used by LQP / pppstats)
 */

struct ppp_stats {
    unsigned long           rbytes;         /* bytes received              */
    unsigned long           rcomp;          /* compressed packets received      */
    unsigned long           runcomp;/* uncompressed packets received */
    unsigned long           rothers;    /* non-ip frames received      */
    unsigned long           rerrors;    /* received errors             */
    unsigned long           roverrun; /* "buffer overrun" counter      */
    unsigned long           tossed;         /* packets discarded               */
    unsigned long           runts;          /* frames too short to process   */
    unsigned long           rgiants;    /* frames too large to process   */
    unsigned long           sbytes;         /* bytes sent                  */
    unsigned long           scomp;          /* compressed packets sent      */
    unsigned long           suncomp;/* uncompressed packets sent  */
    unsigned long           sothers;    /* non-ip frames sent              */
    unsigned long           serrors;    /* transmitter errors          */
    unsigned long           sbusy;          /* "transmitter busy" counter    */
};

/*
 * Demand dial fields
 */

struct ppp_ddinfo {
    unsigned long           ip_sjiffies;    /* time when last IP frame sent */
    unsigned long           ip_rjiffies;    /* time when last IP frame recvd*/
    unsigned long           nip_sjiffies;   /* time when last NON-IP sent      */
    unsigned long           nip_rjiffies;   /* time when last NON-IP recvd    */
};

#ifdef __KERNEL__

struct ppp {
    int                     magic;          /* magic value for structure    */

    /* Bitmapped flag fields. */
```

```
    char            inuse;          /* are we allocated?       */
    char            sending;  /* "channel busy" indicator    */
    char            escape;         /* 0x20 if prev char was PPP_ESC*/
    char            toss;       /* toss this frame    */

    unsigned int    flags;          /* miscellany              */

    unsigned long       xmit_async_map[8]; /* 1 bit means that given control
                            character is quoted on output*/

    unsigned long       recv_async_map; /* 1 bit means that given control
                            character is ignored on input*/
    int             mtu;        /* maximum xmit frame size  */
    int             mru;        /* maximum receive frame size     */
    unsigned short  fcs;        /* FCS field of current frame */

    /* Various fields. */
    int             line;       /* PPP channel number          */
    struct tty_struct   *tty;       /* ptr to TTY structure        */
    struct device       *dev;           /* easy for intr handling  */
    struct slcompress*slcomp; /* for header compression      */
    unsigned long       last_xmit;      /* time of last transmission    */

    /* These are pointers to the malloc()ed frame buffers.
        These buffers are used while processing a packet.      If a packet
        has to hang around for the user process to read it, it lingers in
        the user buffers below. */
    unsigned char       *rbuff;         /* receiver buffer          */
    unsigned char       *xbuff;         /* transmitter buffer           */
    unsigned char       *cbuff;         /* compression buffer           */

    /* These are the various pointers into the buffers. */
    unsigned char       *rhead;         /* RECV buffer pointer (head)      */
    unsigned char       *rend;          /* RECV buffer pointer (end) */
    int             rcount;         /* PPP receive counter          */
    unsigned char       *xhead;         /* XMIT buffer pointer (head)      */
    unsigned char   *xtail;         /* XMIT buffer pointer (end)        */

    /* Structures for interfacing with the user process. */
#define RBUFSIZE 4000
    unsigned char       *us_rbuff;      /* circular incoming packet buf.*/
    unsigned char       *us_rbuff_end;      /* end of allocated space*/
    unsigned char       *us_rbuff_head; /* head of waiting packets   */
    unsigned char       *us_rbuff_tail; /* tail of waiting packets       */
```

```
    unsigned char           us_rbuff_lock;/* lock: bit 0 head bit 1 tail    */
    int                     inp_sig;   /* input ready signal for pgrp */
    int                     inp_sig_pid;    /* process to get notified*/

    /* items to support the select() function */
    struct wait_queue       *write_wait;    /* queue for reading processes      */
    struct wait_queue       *read_wait;     /* queue for writing processes      */

    /* PPP interface statistics. */
    struct ppp_stats   stats;               /* statistic information    */

    /* PPP demand dial information. */
    struct ppp_ddinfo       ddinfo;         /* demand dial information    */
};

#endif     /* __KERNEL__ */
#endif     /* _LINUX_PPP_H */
/* include/linux/ppp.h ************************************************************/
```

1.19 route.h 头文件

该文件定义路由表结构。主要定义有两个结构和一些标志位。两个结构表示的意义基本相同。只是使用在不同的函数调用中。

结构一：old_rentry 结构

```
struct old_rtentry {
      unsigned long rt_genmask;
      struct sockaddr       rt_dst;
      struct sockaddr       rt_gateway;
      short             rt_flags;
      short             rt_refcnt;
      unsigned long rt_use;
      char        *rt_dev;
};
```

该结构使用于 SIOCADDRTOLD 和 SIOCDELRTOLD 选项。

rt_genmask

路由表项 IP 地址网络掩码。

rt_dst

该字段在进行路由时与数据包目的地址进行匹配。

rt_gateway

中间网关（或路由器）地址，当无法直接到达目的地时，需要经过网关进行转发，此为中间网关的地址。

rt_flags

该路由表项所处状态标志位。

rt_refcnt
该路由表项的使用计数。
rt_use
该路由表项是否正在使用。
rt_dev
该路由路径的出站接口设备。

结构二：rentry 结构

```
struct rtentry {
        unsigned long rt_hash;   /* hash key for lookups         */
        struct sockaddr       rt_dst;          /* target address           */
        struct sockaddr       rt_gateway;     /* gateway addr (RTF_GATEWAY)      */
        struct sockaddr       rt_genmask;     /* target network mask (IP)    */
        short             rt_flags;
        short             rt_refcnt;
        unsigned long rt_use;
        struct ifnet      *rt_ifp;
        short             rt_metric;/* +1 for binary compatibility!       */
        char        *rt_dev;   /* forcing the device at add     */
        unsigned long rt_mss;           /* per route MTU/Window */
        unsigned long rt_window;     /* Window clamping */
};
```

该结构被使用于 SIOCADDRT 和 SIOCDELRT 选项。
rt_hash
该路由表项 HASH 值用于快速寻找相应路由表项。
rt_dst
rt_gateway
rt_genmask
rt_flags
rt_refcnt
rt_use
rt_dev
以上七个字段意义同 old_rentry 结构。
rt_ifp
该字段意义不是很明确，且内核网络代码中无法找到 ifnet 结构定义。
rt_metric
路由代价值。
rt_mss
该路由表项所表示路径的最大传输单元。
rt_window
该路由表项所代表路径的窗口大小。窗口是一种限制数据包发送过快的措施。发送数据包的个数或总字节数应该在窗口内。

有一点需要注意的是 rentry，old_rentry 只用于用户空间和内核空间传输参数，内核路由表

项并非使用这两个结构表示，而是由 rtable 结构表示一个路由表项。rentry，old_rentry 这两个结构主要用于选项设置或获取。即通过这两个结构传送参数更改或者获取系统路由表项内容。

该文件结束处定义了路由表项所使用的一些标志值。
RTF_UP
该表项可被使用。
RTF_GATEWAY
该表项通向一个路由器或者网关。
RTF_HOST
该表项提供可直达目的主机的路径。
RTF_REINSTATE
该表项超时后已进行重新确认。每个路由表项(除了永久表项)都有一个超时时间，当超时发生时，该表项必须经过重新确认，方才认为有效。
RTF_DYNAMIC
该表项是根据接收的数据包信息动态建立的，且内容可变。如 ICMP 送来一个 Redirect 信息，可表项表示的路径可被更改。
RTF_MODIFIED
该表项内容根据 ICMP Redirect 信息进行了更改。
RTF_MSS
该表项指定了最大传输单元。数据包大小必须满足该规定。
RTF_WINDOW
该表项指定了窗口大小。

```
/* include/linux/route.h ************************************************************/
/*
 * INET        An implementation of the TCP/IP protocol suite for the LINUX
 *        operating system.    INET is implemented using the    BSD Socket
 *        interface as the means of communication with the user level.
 *
 *        Global definitions for the IP router interface.
 *
 * Version:    @(#)route.h    1.0.3        05/27/93
 *
 * Authors:    Original taken from Berkeley UNIX 4.3, (c) UCB 1986-1988
 *        Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
 *
 *        This program is free software; you can redistribute it and/or
 *        modify it under the terms of the GNU General Public License
 *        as published by the Free Software Foundation; either version
 *        2 of the License, or (at your option) any later version.
 */
#ifndef _LINUX_ROUTE_H
```

```
#define _LINUX_ROUTE_H

#include <linux/if.h>


/* This structure gets passed by the SIOCADDRTOLD and SIOCDELRTOLD calls. */

struct old_rtentry {
        unsigned long rt_genmask;
        struct sockaddr       rt_dst;
        struct sockaddr       rt_gateway;
        short             rt_flags;
        short             rt_refcnt;
        unsigned long rt_use;
        char        *rt_dev;
};

/* This structure gets passed by the SIOCADDRT and SIOCDELRT calls. */
struct rtentry {
        unsigned long rt_hash;   /* hash key for lookups         */
        struct sockaddr       rt_dst;           /* target address           */
        struct sockaddr       rt_gateway;     /* gateway addr (RTF_GATEWAY)      */
        struct sockaddr       rt_genmask;     /* target network mask (IP)    */
        short             rt_flags;
        short             rt_refcnt;
        unsigned long rt_use;
        struct ifnet      *rt_ifp;
        short             rt_metric;/* +1 for binary compatibility!       */
        char        *rt_dev;  /* forcing the device at add    */
        unsigned long rt_mss;           /* per route MTU/Window */
        unsigned long rt_window;     /* Window clamping */
};

/* Route Flags */
#define    RTF_UP         0x0001          /* route usable           */
#define    RTF_GATEWAY    0x0002          /* destination is a gateway        */
#define    RTF_HOST    0x0004          /* host entry (net otherwise)      */
#define RTF_REINSTATE     0x0008          /* reinstate route after tmout      */
#define    RTF_DYNAMIC    0x0010          /* created dyn. (by redirect)      */
#define    RTF_MODIFIED   0x0020          /* modified dyn. (by redirect)    */
#define RTF_MSS         0x0040          /* specific MSS for this route    */
#define RTF_WINDOW 0x0080          /* per route window clamping          */
```

```
/*
 *    REMOVE THESE BY 1.2.0 !!!!!!!!!!!!!!!!
 */

#define    RTF_MTU            RTF_MSS
#define rt_mtu        rt_mss

#endif     /* _LINUX_ROUTE_H */
/* include/linux/route.h ************************************************************/
```

1.20 skbuff.h 头文件

该文件定义了网络栈代码用于封装网络数据的最重要的数据结构 sk_buff 结构以及部分对其进行操作的函数定义。

该文件首部的两个常量定义：

FREE_READ

FREE_WRITE

用于表示释放某个数据包时该数据包时隶属于写缓冲区还是读缓冲区。对于每个套接字内核分配一定的读写缓冲区，当缓冲区不够时，会发生数据包丢弃行为。所以及时维护缓冲区的大小信息是至关重要的。而释放一个数据包时就是在得到更多的空闲空间以供其它数据包使用，而读写缓冲区是不同的，所以释放时需要指示释放的这部分空间是属于哪个缓冲区。FREE_READ 表示释放的空间被加到读缓冲区中，即现在有更多的读空间。FREE_WRITE 以此类比。

sk_buff_head 结构表示数据包队列的头。注意其字段与 sk_buff 结构的开始几个字段时一样的，如此处理可以利用指针转化将 sk_buff_head 当作 sk_buff 结构使用，反过来也如此。内核代码很多地方为方便处理，通常都是这么做的。

sk_buff 是内核网络栈代码的又一核心数据结构。该结构被用来封装网络数据。网络栈代码对数据的处理都是以 sk_buff 结构为单元进行的。

prev，next

这两个字段用于构成 sk_buff 结构的队列。

magic_debug_cookie

调试之目的。

link3

这个指针也是用于构成 sk_buff 结构的队列。这个队列就是数据包重发队列，用于 TCP 协议中。重发队列是由 sock 结构的 send_head, send_tail 字段指向的队列。send_head 指向这个队列的首部，send_tail 指向这个队列的尾部。而队列的连接是使用 sk_buff 结构的 link3 字段完成的。send_head, send_tail 是指向 sk_buff 结构的指针，而非 sk_buff_head 结构。

sk
该数据包所属的套接字。

when
该数据包的发送时间。该字段用于计算往返时间 RTT。

stamp
该字段也是记录时间，但目前暂未使用该字段用于任何目的。

dev
对于一个接收的数据包而言，该字段表示接收该数据包的接口设备。对于一个待发送的数据包而言，该字段如果不为 NULL，则表示将要发送该数据包的接口设备。

mem_addr
该 sk_buff 结构在内存中的基地址，该字段用于释放该 sk_buff 结构。

```
union {
        struct tcphdr   *th;
        struct ethhdr   *eth;
        struct iphdr    *iph;
        struct udphdr   *uh;
        unsigned char   *raw;
        unsigned long   seq;
  } h;
```

该字段一个联合类型，表示了数据包在不同处理层次上所到达的处理位置。如在链路层上，eth 指针有效，指向以太网首部第一个字节位置，在网络层上，iph 指针有效指向 IP 首部第一个字节位置。raw 指针随层次变化而变化，在链路层上时，其等于 eth，在网络层上时，其等于 iph。seq 是针对使用 TCP 协议的待发送数据包而言，此时该字段值表示该数据包的 ACK 值。ACK 值等于数据包中第一个数据的序列号加上数据的长度值。

ip_hdr
指向 IP 首部的指针，此处特别的分出一个字段用于指向 IP 首部主要用于 RAW 套接字。

mem_len
该字段表示 sk_buff 结构大小加上数据帧的总长度。

len
该字段只表示数据帧长度。即 len=mem_len – sizeof(sk_buff).

fraglen
fraglist
这两个字段用于分片数据包。fraglen 表示分片数据包个数，而 fraglist 指向分片数据包队列。

truesize
意义同 mem_len.

saddr
数据包发送的源端 IP 地址。

daddr
数据包最终目的端 IP 地址。

raddr
数据包下一站 IP 地址。

acked, used, free, arp
acked=1 表示该数据包已得到确认，可以从重发队列中删除。
used=1 表示该数据包的数据已被应用程序读完，可以进行释放。
free=1 用于数据包发送，当某个待发送数据包 free 标志位等于 1，则表示无论该数据包是否发送成功，在进行发送操作后立即释放，无需缓存。
arp 用于待发送数据包，该字段等于 1 表示此待发送数据包已完成 MAC 首部的建立。arp=0 表示 MAC 首部中目的端硬件地址尚不知晓，故需使用 ARP 协议询问对方，在 MAC 首部尚未完全建立之前，该数据包一直处于发送缓冲队列中（device 结构中 buffs 数组元素指向的某个队列以及 ARP 协议的某个队列中）。

tries, lock, localroute
tries 字段表示该数据包已进行 tries 试发送，如果试发送超出域值，则会放弃该数据包的发送。如对于 TCP 建立连接之 SYN 数据包，发送次数超过 3 次即放弃发送。
lock 表示该数据包是否正在被系统其它部分使用。
localroute 表示进行路由时是使用局域网路由（localroute=1）还是广域网路由。

pkt_type
该数据包的类型，可取如下值：
PACKET_HOST
这是一个发往本机的数据包。
PACKET_BROADCAST
广播数据包。
PACKET_MULTICAST
多播数据包。
PACKET_OTHERHOST
该数据包是发往其它机器的，如果本机没有被配置为转发功能，该数据包即被丢弃。

users
使用该数据包的模块数。

pkt_class
意义同 pkt_type.

in_dev_queue;
该字段表示该数据包是否正在缓存于设备缓存队列中。

padding[0];
填充字节。目前定义为 0 字节，即无填充。

data[0];
指向数据部分。数据一般紧接着该 sk_buff 结构，也有可能在任何地址处。

SK_WMEM_MAX
SK_RMEM_MAX
这两个字段定义了套接字读写缓冲区的最大空间大小。

SK_FREED_SKB
SK_GOOD_SKB
SK_HEAD_SKB
这三个常量用于调试目的。用于 magic_debug_cookie 字段的取值。

该文件中除了声明了一部分函数外，另外还定义了一部分对 sk_buff 结构进行操作的常用函数，由于这些函数实现上都较为简单，故此处只简单说明这些函数完成的功能，不做一一分析。

struct sk_buff *skb_peek(struct sk_buff_head *list_)
从 list 指向的队列首部取下一个数据包，但该数据包并不从队列中删除。

void skb_queue_head_init(struct sk_buff_head *list)
对一个 sk_buff_head 结构进行初始化操作。

void skb_queue_head(struct sk_buff_head *list_,struct sk_buff *newsk)
将 newsk 指向的数据包插入到 list 队列中的首部。

void skb_queue_tail(struct sk_buff_head *list_, struct sk_buff *newsk)
将 newsk 指向的数据包插入到 list 队列的尾部。

struct sk_buff *skb_dequeue(struct sk_buff_head *list_)
从 list 所指向队列首部取下一个数据包，并将该数据包从此队列中删除。

void skb_insert(struct sk_buff *old, struct sk_buff *newsk)
将 newsk 指向的数据包插入到 old 指向的数据包的前面。

void skb_append(struct sk_buff *old, struct sk_buff *newsk)
将 newsk 指向的数据包插入到 old 指向的数据包的后面。

void skb_unlink(struct sk_buff *skb)
将 skb 结构从其链入的队列中删除。

```
/* include/linux/skbuff.h ************************************************************/
/*
 *    Definitions for the 'struct sk_buff' memory handlers.
 *
 *    Authors:
 *          Alan Cox, <gw4pts@gw4pts.ampr.org>
 *          Florian La Roche, <rzsfl@rz.uni-sb.de>
 *
 *    This program is free software; you can redistribute it and/or
 *    modify it under the terms of the GNU General Public License
 *    as published by the Free Software Foundation; either version
 *    2 of the License, or (at your option) any later version.
 */

#ifndef _LINUX_SKBUFF_H
#define _LINUX_SKBUFF_H
#include <linux/malloc.h>
#include <linux/wait.h>
#include <linux/time.h>
#include <linux/config.h>

#undef CONFIG_SKB_CHECK

#define HAVE_ALLOC_SKB          /* For the drivers to know */


#define FREE_READ    1
#define FREE_WRITE   0


struct sk_buff_head {
   struct sk_buff          * volatile next;
   struct sk_buff          * volatile prev;
#if CONFIG_SKB_CHECK
   int                     magic_debug_cookie;
#endif
};
```

```
struct sk_buff {
   struct sk_buff          * volatile next;
   struct sk_buff          * volatile prev;
#if CONFIG_SKB_CHECK
   int                     magic_debug_cookie;
#endif
   struct sk_buff          * volatile link3;
   struct sock             *sk;
   volatile unsigned long  when;   /* used to compute rtt's   */
   struct timeval          stamp;
   struct device                   *dev;
   struct sk_buff          *mem_addr;
   union {
           struct tcphdr   *th;
           struct ethhdr   *eth;
           struct iphdr    *iph;
           struct udphdr   *uh;
           unsigned char   *raw;
           unsigned long   seq;
   } h;
   struct iphdr       *ip_hdr;       /* For IPPROTO_RAW */
   unsigned long              mem_len;
   unsigned long           len;
   unsigned long              fraglen;
   struct sk_buff          *fraglist;  /* Fragment list */
   unsigned long              truesize;  /* equals to mem_len */
   unsigned long           saddr;
   unsigned long           daddr;     /* This packet's eventual destination */
   unsigned long              raddr;          /* next hop addr */
   volatile char           acked, used, free, arp;
   unsigned char              tries, lock, localroute, pkt_type;
/* pkt_type */
#define PACKET_HOST             0       /* To us */
#define PACKET_BROADCAST        1
#define PACKET_MULTICAST        2
#define PACKET_OTHERHOST        3       /* Unmatched promiscuous */
   unsigned short          users;          /*  User  count  -  see  datagram.c  (and  soon
seqpacket.c/stream.c) */
   unsigned short          pkt_class;/* For drivers that need to cache the packet type with the skbuff
(new PPP) */
#ifdef CONFIG_SLAVE_BALANCING
   unsigned short          in_dev_queue;
#endif
   unsigned long              padding[0];
```

```
  unsigned char                 data[0];
};

#define SK_WMEM_MAX   32767
#define SK_RMEM_MAX   32767

#ifdef CONFIG_SKB_CHECK
#define SK_FREED_SKB    0x0DE2C0DE
#define SK_GOOD_SKB     0xDEC0DED1
#define SK_HEAD_SKB     0x12231298
#endif

#ifdef __KERNEL__
/*
 *    Handling routines are only of interest to the kernel
 */

#include <asm/system.h>

#if 0
extern void                 print_skb(struct sk_buff *);
#endif
extern void                 kfree_skb(struct sk_buff *skb, int rw);
extern void                 skb_queue_head_init(struct sk_buff_head *list);
extern void                 skb_queue_head(struct sk_buff_head *list,struct sk_buff *buf);
extern void                 skb_queue_tail(struct sk_buff_head *list,struct sk_buff *buf);
extern struct sk_buff *         skb_dequeue(struct sk_buff_head *list);
extern void                 skb_insert(struct sk_buff *old,struct sk_buff *newsk);
extern void                 skb_append(struct sk_buff *old,struct sk_buff *newsk);
extern void                 skb_unlink(struct sk_buff *buf);
extern struct sk_buff *         skb_peek_copy(struct sk_buff_head *list);
extern struct sk_buff *         alloc_skb(unsigned int size, int priority);
extern void                 kfree_skbmem(struct sk_buff *skb, unsigned size);
extern struct sk_buff *         skb_clone(struct sk_buff *skb, int priority);
extern void                 skb_device_lock(struct sk_buff *skb);
extern void                 skb_device_unlock(struct sk_buff *skb);
extern void                 dev_kfree_skb(struct sk_buff *skb, int mode);
extern int              skb_device_locked(struct sk_buff *skb);
/*
 *    Peek an sk_buff. Unlike most other operations you _MUST_
 *    be careful with this one. A peek leaves the buffer on the
 *    list and someone else may run off with it. For an interrupt
 *    type system cli() peek the buffer copy the data and sti();
 */
```

```
static __inline__ struct sk_buff *skb_peek(struct sk_buff_head *list_)
{
	struct sk_buff *list = (struct sk_buff *)list_;
	return (list->next != list)? list->next : NULL;
}

#if CONFIG_SKB_CHECK
extern int			skb_check(struct sk_buff *skb,int,int, char *);
#define IS_SKB(skb)		skb_check((skb), 0, __LINE__,__FILE__)
#define IS_SKB_HEAD(skb)	skb_check((skb), 1, __LINE__,__FILE__)
#else
#define IS_SKB(skb)
#define IS_SKB_HEAD(skb)

extern __inline__ void skb_queue_head_init(struct sk_buff_head *list)
{
	list->prev = (struct sk_buff *)list;
	list->next = (struct sk_buff *)list;
}

/*
 *	Insert an sk_buff at the start of a list.
 */

extern __inline__ void skb_queue_head(struct sk_buff_head *list_,struct sk_buff *newsk)
{
	unsigned long flags;
	struct sk_buff *list = (struct sk_buff *)list_;

	save_flags(flags);
	cli();
	newsk->next = list->next;
	newsk->prev = list;
	newsk->next->prev = newsk;
	newsk->prev->next = newsk;
	restore_flags(flags);
}

/*
 *	Insert an sk_buff at the end of a list.
 */

extern __inline__ void skb_queue_tail(struct sk_buff_head *list_, struct sk_buff *newsk)
{
```

```
        unsigned long flags;
        struct sk_buff *list = (struct sk_buff *)list_;

        save_flags(flags);
        cli();

        newsk->next = list;
        newsk->prev = list->prev;

        newsk->next->prev = newsk;
        newsk->prev->next = newsk;

        restore_flags(flags);
}

/*
 *    Remove an sk_buff from a list. This routine is also interrupt safe
 *    so you can grab read and free buffers as another process adds them.
 */

extern __inline__ struct sk_buff *skb_dequeue(struct sk_buff_head *list_)
{
        long flags;
        struct sk_buff *result;
        struct sk_buff *list = (struct sk_buff *)list_;

        save_flags(flags);
        cli();

        result = list->next;
        if (result == list) {
                restore_flags(flags);
                return NULL;
        }

        result->next->prev = list;
        list->next = result->next;

        result->next = NULL;
        result->prev = NULL;

        restore_flags(flags);
        return result;
}
```

```
/*
 *	Insert a packet before another one in a list.
 */

extern __inline__ void skb_insert(struct sk_buff *old, struct sk_buff *newsk)
{
	unsigned long flags;

	save_flags(flags);
	cli();
	newsk->next = old;
	newsk->prev = old->prev;
	old->prev = newsk;
	newsk->prev->next = newsk;

	restore_flags(flags);
}

/*
 *	Place a packet after a given packet in a list.
 */

extern __inline__ void skb_append(struct sk_buff *old, struct sk_buff *newsk)
{
	unsigned long flags;

	save_flags(flags);
	cli();

	newsk->prev = old;
	newsk->next = old->next;
	newsk->next->prev = newsk;
	old->next = newsk;

	restore_flags(flags);
}

/*
 *	Remove an sk_buff from its list. Works even without knowing the list it
 *	is sitting on, which can be handy at times. It also means that THE LIST
 *	MUST EXIST when you unlink. Thus a list must have its contents unlinked
 *	_FIRST_.
 */
```

```
extern __inline__ void skb_unlink(struct sk_buff *skb)
{
	unsigned long flags;

	save_flags(flags);
	cli();

	if(skb->prev && skb->next)
	{
		skb->next->prev = skb->prev;
		skb->prev->next = skb->next;
		skb->next = NULL;
		skb->prev = NULL;
	}
	restore_flags(flags);
}

#endif

extern struct sk_buff *		skb_recv_datagram(struct sock *sk,unsigned flags,int noblock, int *err);
extern int			datagram_select(struct sock *sk, int sel_type, select_table *wait);
extern void			skb_copy_datagram(struct sk_buff *from, int offset, char *to,int size);
extern void			skb_free_datagram(struct sk_buff *skb);

#endif	/* __KERNEL__ */
#endif	/* _LINUX_SKBUFF_H */
/* include/linux/skbuff.h ************************************************************/
```

1.21 socket.h 头文件

该头文件定义了 sockaddr 结构，linger 结构以及大量常量，这些常量主要用于编程接口，选项设置和获取。sockaddr 结构对于网络程序员是再熟悉不过了。其定义了一个通用地址接口。linger 结构用于设置套接字关闭时的行为。关闭套接字时可以在执行关闭操作后立刻返回，即不等于关闭操作的最终完成（因为关闭操作的最终完成一般需要经过 4 路数据包的传送）。如果设置了 linger 选择，则 close 函数必须等待直到套接字完全关闭。其中 linger 结构中 l_onoff 表示是否进行等待（l_onoff=1 表示等待完成）。而 l_linger 字段表示等待的时间，如果等待时间超时，无论实际上是否完成关闭操作，软件处理上都认为已经完成关闭操作。

```
/*include/linux/socket.h ************************************************************/
#ifndef _LINUX_SOCKET_H
#define _LINUX_SOCKET_H
```

```
#include <linux/sockios.h>          /* the SIOCxxx I/O controls   */


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

struct linger {
   int               l_onoff;  /* Linger active              */
   int               l_linger; /* How long to linger for     */
};

/* Socket types. */
/*套接字类型*/
#define SOCK_STREAM     1           /* stream (connection) socket */
#define SOCK_DGRAM      2           /* datagram (conn.less) socket       */
#define SOCK_RAW        3           /* raw socket                */
#define SOCK_RDM        4           /* reliably-delivered message */
#define SOCK_SEQPACKET      5           /* sequential packet socket     */
#define SOCK_PACKET     10          /* linux specific way of  */
                            /* getting packets at the dev   */
                            /* level.    For writing rarp and       */
                            /* other similar things on the  */
                            /* user level.                */

/* Supported address families. */
/*支持的地址（域）类型*/
#define AF_UNSPEC       0
#define AF_UNIX         1
#define AF_INET         2
#define AF_AX25         3
#define AF_IPX          4
#define AF_APPLETALK        5

#define AF_MAX          8       /* For now.. */

/* Protocol families, same as address families. */
/*协议类，同地址类型*/
#define PF_UNSPEC       AF_UNSPEC
#define PF_UNIX         AF_UNIX
#define PF_INET         AF_INET
#define PF_AX25         AF_AX25
```

```
#define PF_IPX          AF_IPX
#define PF_APPLETALK    AF_APPLETALK

#define PF_MAX          AF_MAX

/* Flags we can use with send/ and recv. */
/*数据处理方式标志位*/
#define MSG_OOB             1 //带外数据，作为紧急数据处理
#define MSG_PEEK    2
#define MSG_DONTROUTE       4

/* Setsockoptions(2) level. Thanks to BSD these must match IPPROTO_xxx */
/*选项设置级别（层次）*/
#define SOL_SOCKET  1
#define SOL_IP          0
#define SOL_IPX         256
#define SOL_AX25        257
#define SOL_ATALK   258
#define SOL_TCP         6
#define SOL_UDP         17

/* For setsockoptions(2) */
/*具体选项类型*/
#define SO_DEBUG    1
#define SO_REUSEADDR    2
#define SO_TYPE         3
#define SO_ERROR    4
#define SO_DONTROUTE    5
#define SO_BROADCAST    6
#define SO_SNDBUF   7
#define SO_RCVBUF   8
#define SO_KEEPALIVE    9
#define SO_OOBINLINE    10
#define SO_NO_CHECK     11
#define SO_PRIORITY  12
#define SO_LINGER   13
/* To add :#define SO_REUSEPORT 14 */

/* IP options */
/*IP 选项*/
#define IP_TOS          1
#define    IPTOS_LOWDELAY           0x10
#define    IPTOS_THROUGHPUT         0x08
#define    IPTOS_RELIABILITY  0x04
```

```
#define IP_TTL              2
#ifdef V1_3_WILL_DO_THIS_FUNKY_STUFF
#define IP_HRDINCL    3
#define IP_OPTIONS    4
#endif

#define IP_MULTICAST_IF             32
#define IP_MULTICAST_TTL            33
#define IP_MULTICAST_LOOP           34
#define IP_ADD_MEMBERSHIP           35
#define IP_DROP_MEMBERSHIP          36


/* These need to appear somewhere around here */
#define IP_DEFAULT_MULTICAST_TTL            1
#define IP_DEFAULT_MULTICAST_LOOP           1
#define IP_MAX_MEMBERSHIPS                  20

/* IPX options */
#define IPX_TYPE 1

/* TCP options - this way around because someone left a set in the c library includes */
/*TCP 选项*/
#define TCP_NODELAY     1
#define TCP_MAXSEG 2

/* The various priorities. */
/*数据包缓存到设备队列中的优先级*/
#define SOPRI_INTERACTIVE   0
#define SOPRI_NORMAL        1
#define SOPRI_BACKGROUND 2

#endif /* _LINUX_SOCKET_H */
/* include/linux/socket.h**********************************************************/
```

1.22 sockios.h 头文件

该文件定义了用于各种层次（协议）IO 控制操作选项类型。

```
/*include/linux/sockios.h*********************************************************/
/*
 * INET         An implementation of the TCP/IP protocol suite for the LINUX
 *          operating system.    INET is implemented using the    BSD Socket
 *          interface as the means of communication with the user level.
```

```
 *
 *		Definitions of the socket-level I/O control calls.
 *
 * Version:	@(#)sockios.h	1.0.2	03/09/93
 *
 * Authors:	Ross Biro, <bir7@leland.Stanford.Edu>
 *		Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
 *
 *		This program is free software; you can redistribute it and/or
 *		modify it under the terms of the GNU General Public License
 *		as published by the Free Software Foundation; either version
 *		2 of the License, or (at your option) any later version.
 */
#ifndef _LINUX_SOCKIOS_H
#define _LINUX_SOCKIOS_H

/* This section will go away soon! */

/* Socket-level I/O control calls. */
/*套接字层次 IO 控制*/
#define FIOSETOWN	0x8901
#define SIOCSPGRP	0x8902
#define FIOGETOWN	0x8903
#define SIOCGPGRP	0x8904
#define SIOCATMARK	0x8905
#define SIOCGSTAMP	0x8906		/* Get stamp */

/* Routing table calls. */
/*路由表项控制*/
#define SIOCADDRT	0x890B		/* add routing table entry	*/
#define SIOCDELRT	0x890C		/* delete routing table entry	*/

/* Socket configuration controls. */
/*套接字配置控制*/
#define SIOCGIFNAME	0x8910		/* get iface name		*/
#define SIOCSIFLINK	0x8911		/* set iface channel		*/
#define SIOCGIFCONF	0x8912		/* get iface list		*/
#define SIOCGIFFLAGS	0x8913		/* get flags			*/
#define SIOCSIFFLAGS	0x8914		/* set flags			*/
#define SIOCGIFADDR	0x8915		/* get PA address		*/
#define SIOCSIFADDR	0x8916		/* set PA address		*/
#define SIOCGIFDSTADDR	0x8917		/* get remote PA address	*/
#define SIOCSIFDSTADDR	0x8918		/* set remote PA address	*/
#define SIOCGIFBRDADDR	0x8919		/* get broadcast PA address	*/
```

```
#define SIOCSIFBRDADDR 0x891a          /* set broadcast PA address    */
#define SIOCGIFNETMASK 0x891b          /* get network PA mask         */
#define SIOCSIFNETMASK 0x891c          /* set network PA mask         */
#define SIOCGIFMETRIC    0x891d        /* get metric                  */
#define SIOCSIFMETRIC    0x891e        /* set metric                  */
#define SIOCGIFMEM  0x891f          /* get memory address (BSD)*/
#define SIOCSIFMEM   0x8920         /* set memory address (BSD) */
#define SIOCGIFMTU   0x8921         /* get MTU size                */
#define SIOCSIFMTU   0x8922         /* set MTU size                */
#define    OLD_SIOCGIFHWADDR    0x8923          /* get hardware address         */
#define    SIOCSIFHWADDR        0x8924          /* set hardware address (NI)   */
#define SIOCGIFENCAP     0x8925         /* get/set slip encapsulation     */
#define SIOCSIFENCAP     0x8926
#define SIOCGIFHWADDR  0x8927          /* Get hardware address        */
#define SIOCGIFSLAVE     0x8929        /* Driver slaving support      */
#define SIOCSIFSLAVE     0x8930
/* begin multicast support change */
#define SIOCADDMULTI    0x8931
#define SIOCDELMULTI    0x8932
/* end multicast support change */

/* Routing table calls (oldrtent - don't use) */
/*路由表项控制，使用 old_rentry 结构*/
#define SIOCADDRTOLD    0x8940         /* add routing table entry     */
#define SIOCDELRTOLD     0x8941        /* delete routing table entry  */

/* ARP cache control calls. */
/*ARP 缓存表项控制*/
#define SIOCDARP      0x8950         /* delete ARP table entry      */
#define SIOCGARP      0x8951         /* get ARP table entry         */
#define SIOCSARP      0x8952         /* set ARP table entry         */

/* RARP cache control calls. */
/*RARP 缓存表项控制*/
#define SIOCDRARP    0x8960         /* delete RARP table entry     */
#define SIOCGRARP    0x8961         /* get RARP table entry        */
#define SIOCSRARP    0x8962         /* set RARP table entry        */

/* Driver configuration calls */
/*设备配置选项*/
#define SIOCGIFMAP   0x8970         /* Get device parameters*/
#define SIOCSIFMAP   0x8971         /* Set device parameters */

/* Device private ioctl calls */
```

```
/*
 *    These 16 ioctls are available to devices via the do_ioctl() device
 *    vector. Each device should include this file and redefine these names
 *    as their own. Because these are device dependent it is a good idea
 *    _NOT_ to issue them to random objects and hope.
 */

#define SIOCDEVPRIVATE  0x89F0    /* to 89FF */

/*
 *    These 16 ioctl calls are protocol private
 */

#define SIOCPROTOPRIVATE 0x89E0 /* to 89EF */
#endif     /* _LINUX_SOCKIOS_H */
/*include/linux/sockios.h************************************************************/
```

1.23 tcp.h 头文件

该文件定义了 TCP 首部格式，使用 TCP 套接字所可能处的各种状态。

TCP 协议是一种面向连接的可靠的端到端协议，为互连于计算机通信网络中的两台主机之间提供可靠的进程间通信。TCP 协议是传输层协议，工作在网络层协议之上（如 IP 协议）。网络层协议为 TCP 提供发送和接收变长报文段信息的一种方式，这些报文通常被封装成数据报的形式传送。数据报提供一种方式用于寻址源端和目的端主机地址。另外网络层还为 TCP 协议提供报文分片和重组服务，从而使报文满足某些中间路由器或者网关的传送要求。诚如上述，TCP 协议之首要目的在于为运行于网络中不同主机上的进程提供可靠的通信，要达到这个要求，TCP 协议需要提供如下的服务：

1>数据基本传输要求

2>可靠性保证

3>流量控制

4>多路分用

5>连接

6>优先级和安全性

以下简单介绍这些服务。

1>数据基本传输要求

TCP 提供了一种流式服务。通常 TCP 协议会自行判定何时节制发送数据。有时，上层应用程序需要自己确定其使用 TCP 发送的数据确实已被发送给远端主机。为了达到此目的，系统提供了 push 函数。push 函数会导致 TCP 协议立刻将应用程序数据转发给远端，以及远端在接收到该数据时，立刻将数据传送给上层应用程序。

2>可靠性保证

TCP通过下列方式来提供可靠性：

•应用数据被分割成TCP认为最适合发送的数据块。这和UDP完全不同，应用程序产生的数据报长度将保持不变。由TCP传递给IP的信息单位称为报文段或段（segment）当TCP发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。
•当TCP收到发自TCP连接另一端的数据，它将发送一个确认。这个确认不是立即发送，通常将推迟几分之一秒。
•TCP将保持它首部和数据的检验和。这是一个端到端的检验和，目的是检测数据在传输过程中的任何变化。如果收到段的检验和有差错，TCP将丢弃这个报文段和不确认收到此报文段（希望发端超时并重发）。
•既然TCP报文段作为IP数据报来传输，而IP数据报的到达可能会失序，因此TCP报文段的到达也可能会失序。如果必要，TCP将对收到的数据进行重新排序，将收到的数据以正确的顺序交给应用层。
•既然IP数据报会发生重复，TCP的接收端必须丢弃重复的数据。

TCP 协议必须能够应对数据丢失，数据重复和数据乱序等问题。这些问题的解决方案是赋予每个数据一个序列号，且使用数据接收确认机制。发送端发送的数据，接收端在接收到这些数据后必须给发送端发送一个确认报文以通知发送端这些数据已被接收。如果发送端在发送数据并等待系统定义的一段时间后，没有受到确认，则会重新发送这些数据。由此如果之前发送的同样数据并没有丢失，只是延迟到达，则接收端有可能接收到相同的两份数据，这就产生了数据重复。而发送的数据包如果经过不同的路径到达接收端，则也有可能出现先发送的数据后于后发送的数据到达，从而造成数据的乱序。序列号的引入可以有效地解决这些问题。如果接收端接收到两份相同的数据，首先期会根据其相同的（或者部分重叠的）序列号判别出有重复数据的问题产生，此时采取的措施是丢弃其中的一份，只保留下一份供上层使用；而数据乱序的问题则可以通过数据的序列号进行排序从而得到正确的数据序列。TCP 协议区别于其它协议的本质之处即在于其传输的每个数据都被绑定一个序列号。这是保证 TCP 所声明的功能必不可少的措施。有关序列号的问题在下文中还将有讨论。

3>流量控制
TCP 使用一种称为窗口的机制来节制发送端发送数据的速率。在 TCP 数据交换中，其中传送每个数据报文中 TCP 首部（TCP 首部格式见下文）中 ACK 标志位设置为 1 的报文都会进行接收端窗口大小的通报。由于 TCP 使用全双工方式进行数据传送，故通信的双方都会向对方通报本端的窗口大小，以使得对方合理安排其发送速率。刚刚提到，窗口大小在每个应答报文中都进行通报，而事实上，对于 TCP 协议，一旦建立起连接之后，到连接关闭之前，其间双方传送的所有正常数据包中 ACK 标志位都设置为 1（减少网络中传输数据量），即窗口通报基本贯穿于整个 TCP 连接阶段。窗口大小规定了发送端（即对端）当前所能够发送的最大数据量。本质上讲，窗口大小表示了当前本地接收端接收缓存的空闲空间大小，即可容纳的数据量。所以可以理解一旦不进行窗口通报，即便发送端发送大量数据，在本地也会被丢弃，因为没有足够的内存容纳这些数据。但由此引起的一个问题是，发送端并不知道接收端接收缓存不够，其认为发送的数据在网络中丢失，从而不断进行重发，这导致这个网络的负载加大，在特别情况下，会造成网络的崩溃，而且也会加大接收端的资源消耗（注意在丢弃数据包之前的所有处理都占用 CPU 时间）。通过窗口通报的方式将从根本上杜绝这些问题，因为数据包不会在网络上出现，占用有限的网络带宽。

4>多路分用

为了在单个主机上实现多个进程的同时通信，TCP 协议使用地址和端口号的组合形式来表示一个通信单元。这种地址加端口的构成形式通常称为套接字（socket）或插口。一个通信通道即有两台主机各自的地址和端口号构成。这种构成形式中，地址部分表示进行通信的主机，而端口号则表示主机上某个进行通信的进程。通常端口号与进程之间的绑定是随机的，但将某个服务绑定到固定的端口上并约定俗成可以有效地为其它主机提供服务（如 FTP 服务使用端口 21）。

5>连接

TCP 提供的可靠性以及流量控制功能要求其必须保存通信双方的状态。我们将将通信双方的状态，加上地址和端口组合（即套接字），加上序列号，加上窗口大小等等所有这些构成通常所说的 TCP 连接。当不同主机间需要进行数据传输时（即进行通信时），首先必须建立起一个连接，而当通信结束后，则必须关闭这个连接以释放资源。TCP 协议是构建在不可靠的协议之上，故需要一种“握手”机制来建立连接。通常我们将 TCP 这种建立连接的方式称为“三路握手”机制。TCP 连接中的各种状态及相互之间的转换在下文中将有专门讨论。

6>优先级和安全性

使用 TCP 协议的上层协议可以表示出对通信优先级和安全性的“关注”。如果没有，则系统会提供默认值。

TCP 协议中序列号的使用

TCP 协议设计中一个最基本的概念即是其传输的每个字节都被赋予一个序列号。序列号的使用是保证 TCP 协议实现可靠传输的重要措施。因为每个字节均被编号，故接收端可对接收到的每个字节进行确认应答。不过为了提供网络利用率以及数据传输效率，确认应答采用累加形式，即一次可以对一批数据进行确认，而非每次单个字节。此处“一批”数据通常是指一个包或者多个包的数据。我们通常说的 TCP 是面向流的协议，这是从使用 TCP 协议传送数据的上层协议而言的，而对于 TCP 协议本身及其下层协议（如 IP 协议）而言，数据传送是以包的形式进行的。TCP 协议从其下层接收的数据都是封装在一个包中，所以进行数据确认应答通常是以一个包的数据为单位，当然很多算法为节省网络带宽，会延迟对所接收数据的确认应答，此时可能对多个包（一般两个包）的数据进行一次性确认应答。因为确认应答是以包为单位（无论是单个包还是多个包，不影响此处讨论的问题本质），所以应答序列号应是在包的边界上，换句话说，如果一切正常，确认应答序列号应该是发送端将要发送的下一个数据包中所包含数据的第一个字节数据的序列号。这一点很重要，因为对于 TCP 协议而言，为保证可靠传输，其对每个发送出去的数据包并不进行立即释放，而是缓存在重发队列中，以便在一段时间后，如果没有接收到对端的确认应答，则需要重新发送这些数据包。而当一个确认应答到达时，发送端需要根据这个应答的序列号释放重发队列中缓存的部分数据包，因为一个应答的到来表示之前发送的某些数据包已经成功发送到对方，从而无需继续在重发队列中缓存这些数据包，如果应答序列号不是在包中数据序列号的边界上，那么将会出现包中数据一部分被确认，另一部分没有被确认。这就让发送端“左右为难”了。而实际上这种情况是不会发生的。即数据的接收是以包为单位的，应答的序列号一定是在包的边界上。不会出现一个包中数据只有一部分被应答的情况（即换句话说，一个包中的数据只有一部分被接收，而这是不可能的）。当然此处可能有人会提出异议：即在数据包传送过程中有可能原来的一个数据包备份片成为多个数据包，这样不是会造成原先的包中一部分数据

可能被接收到，而另一部分数丢失的情况？事实上，这种考虑本身没有问题，但对于接收端而言，如果数据包在中间路由器或者网关上被分片，则接收端会在网络层进行组装，在组装完成之后才会传送给上一层及传输层协议使用。即数据包在传输过程中的分片对于 TCP 协议而言是不可见的。而发送端的分片并不影响问题的本质，因为对于 TCP 协议而言，无所谓发送端分片，每个分片都是被作为一个正常的数据包处理的，因为 TCP 提供的是流式数据传输服务。

序列号简单的说就是一个 32 位的整型数。由于只有 32 位表示，故在经过一段时间后，这个数字会进行回环。所以对于一个高速率以太网而言，序列号很快会绕回来。此时对序列号大小的判断需要谨慎处理，特别是在靠近 2**32-1 附近的序列号。

TCP 首部格式如图 1-30 所示。

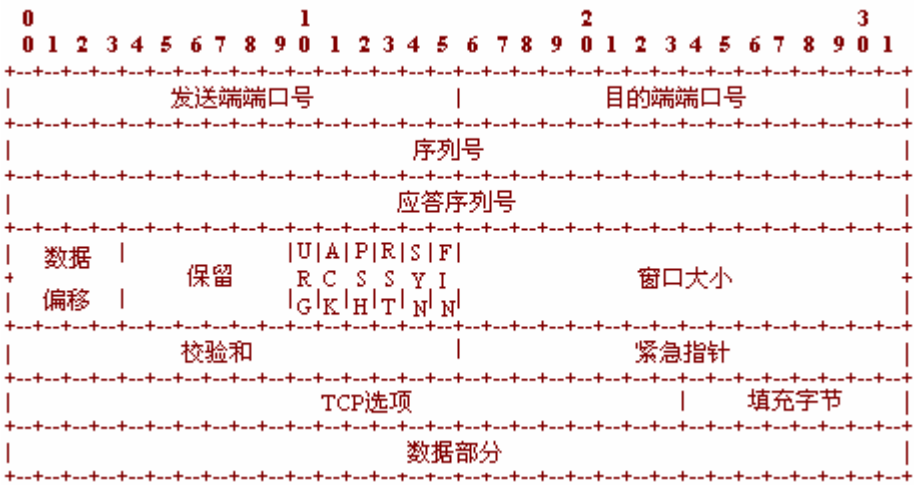


图 1-30 TCP 协议首部格式

发送端端口号：16 比特
标识该数据包源端数据发送进程。

目的端端口号：16 比特
标识该数据包目的端数据接收进程。

序列号：32 比特
本端发送的数据包中所包含数据的第一个字节的序列号。

应答序列号：32 比特
该序列号表示本端想要接收的的下一个数据包中所包含数据中第一个字节数据的序列号。注意不要混淆序列号和应答序列号这两个字段的意义。序列号字段时对本地发送数据的编号，而应答序列号是对对方发送数据的计号：即计算本地已接收到的对方发送的数据的编号。

数据偏移：4 比特
该字段表示 TCP 首部以 4 字节为单位的长度（包括 TCP 选项，如果存在）。换句话说即用户数据的开始起点（从 TCP 首部第一个字节算起）。由此可见如同 IP 首部，TCP 首部必是 4 字节的整数倍，有时这需要在首部结尾处加填充字节。

保留字段：6 比特
必须初始化为 0。

控制位：6 比特
URG：URG=1 表示紧急指针字段值有效，否则无效。紧急数据是将接收的数据迅速传替给应用程序的一种手段。
ACK：ACK=1 表示这是一个应答报文（注意应答报文中可以包含本端发送的数据，事实上，很少只发送一个应答报文而其中不附带数据的，换句话说，应答报文一般都是随数据顺便发送的。
PSH：PSH=1 时的意义如同 URG=1 表示需要将数据立刻交给应用程序。软件处理上是给应用程序发送信号（SIGURG）。
RST：复位标志。当一个报文中 RST=1 时，表示对方要求本地重新建立与对方的连接。该标志位一般用于非严重错误恢复中。
SYN：同步标志。当 SYN=1 时表示这是一个同步报文，同步报文只用在建立连接的过程中。
FIN：结束连接报文。当 FIN=1 时表示本地已无数据发送给对端，但可以继续接收对方发送的数据。FIN 报文通常使用在关闭连接中。

窗口大小：16 比特
该字段用于流量节制：表示当前对方所能容忍的最大数据接收量。本地必须注意到该要求，发送的数据量必须小于该字段所声明的值。

校验和：16 比特
该字段计算 TCP 首部及其之后的所有用户数据部分的校验和。
注意在计算 TCP 校验和（UDP 也如此）还包含一个伪首部的计算。图 1-31 显示了计算 TCP 校验和时所需计入的伪首部格式。

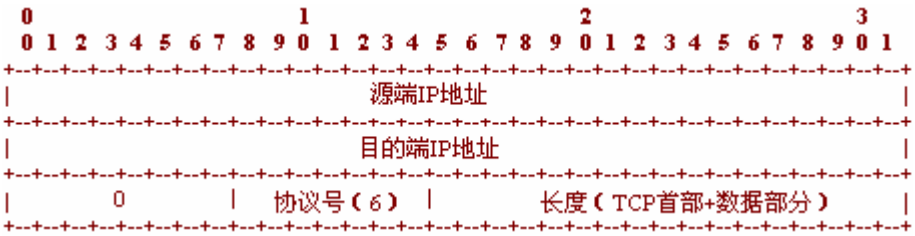


图 1-31TCP 校验和计算中需计入的伪首部格式

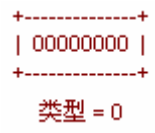
紧急指针：16 比特
该字段表示紧急数据部分，其值是一个偏移量，该偏移量是从该数据包中第一个数据字节算起。只当 URG 标志位为 1 时，紧急指针字段值方才有效。

TCP 选项：长度可变
TCP 选项较少，从结构来分，可分为单字节选项和多字节选项。单字节选项只有一个表示选项类型的字节。多字节选项第一个字节为类型字节，第二个字节为长度字节，长度包括类型字节及其本身以及后续的数据字节，第三个字节及其之后字节为数据字节部分。
此处介绍常用的三种选项，如下所示。

类型	长度	意义

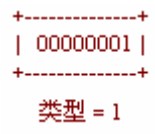
0	--	选项结束
1	--	无操作
2	4	最大报文长度

1>选项结束



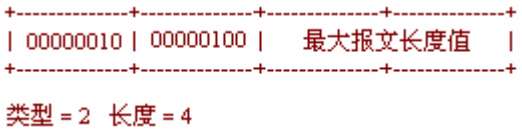
该选项标志着所有选项的结束。在此选项之后不可再有其它选项类型。

2>无操作选项



该选项不表示任何意义，只是作为其它选项之间的一种填充。有时为了将下一个选项安排在 4 字节边界上，有必要进行填充，但又不可简单的填充 0，因为这样可能被解析为上一个选项的数据部分或其它后续部分，而使用无操作选项则无此疑义。

3>最大报文长度（MSS：Maximum Segment Size）选项



该选项只是用 TCP 建立连接过程的报文交换中。交换的得到的对方最大报文长度值在整个后续连接中均不作改变。最大报文长度是指用户可一次性发送的数据最大长度。其与 MTU（Maximum Transmission Unit）即最大传输单元的差别在于（假设网络层使用 IP 协议）：MSS=MTU-（TCP 首部长度+IP 首部长度+MAC 首部长度）

填充值
该字段是为保证 TCP 首部的长度为 4 字节的倍数。填充值一般取 0。

数据部分
此之后即用户实际需要传送的数据。

TCP 连接状态
按 TCP 协议的标准表示法，TCP 可具有如下几种状态：
为讨论方便，如下讨论中区分服务端和客户端，实际软件处理上对二者一视同仁。

CLOSED
关闭状态。在两个通信端使用“三路握手”机制建立连接之前即处于该状态。

LISTEN
监听状态。此状态是对服务器端而言的。处于此状态的套接字正在等待客户端的连接请求。

SYN-SENT
客户端发送一个 SYN 报文请求建立与服务器端的连接后，设置为该状态。注意处于此状态的套接字仅仅表示发送了 SYN 请求报文，但尚未得到对方应答。如果已得到对方应答，则就会进入 ESTABLISHED 状态。

SYN-RECV
一般服务端在接收到客户端发送的 SYN 报文后，会发送一个附带 ACK 的 SYN 报文进行与对方的同步，之后将其状态设置为 SYN-RECV。

ESTABLISHED
客户端接收到其之前发送的 SYN 报文的 ACK 后，设置为此状态。并且紧接着发送一个 ACK 报文给服务端从而建立正式连接。而此时处于 SYN-RECV 状态的服务端在接收到此 ACK 后也会将自己的状态设置为 ESTABLISHED。到此为止，连接完全建立，之后二者既可进行数据的传输。

以下状态将无法区分服务端和客户端，主要视哪端首先发起关闭操作。我们将首先发起关闭操作的一端视为客户端，对应的另一端视为服务端。

FIN-WAIT-1
客户端发起关闭操作，此时客户端将发送一个 FIN 报文给对方，并将其状态设置为 FIN-WAIT-1。

FIN-WAIT-2
处于 FIN-WAIT-1 状态的客户端在接收到之前发送的 FIN 的 ACK 报文后，将其状态设置为 FIN-WAIT-2，然后一直处于该状态，直到接收到对方发送 FIN 报文。注意在接收到 FIN 报文之前，本端仍然可以继续接收对方发送的数据。因为发送 FIN 报文给对方仅仅表示本地不再有数据发送给对方，并不表示本地不再接收对方的数据。这是两回事。处于此状态的套接字接收到对方发送 FIN 时，将进入 TIME-WAIT 状态。

CLOSE-WAIT
服务端在接收到对方的 FIN 报文时，将本地状态设置为 CLOSE-WAIT，并即刻发送 ACK 报文给对方。此后本地依然可以发送尚未发送的数据直到数据发送完。

CLOSING
当连接双方同时发送 FIN 时，会进入此状态。在前面的讨论中，如果本地发送关闭操作，则会发送一个 FIN 报文给对方，并将自己状态设置为 FIN-WAIT-1，正常情况下，紧接着应该接收到对方响应的 ACK 报文这样本地就可以进入 FIN-WAIT-2 状态，然后一直等待对方的 FIN 报文。而如果在等待 ACK 报文时，接收到对方的 FIN 报文，则表示对方同时发起了关闭操作，此时本地并不再进入 FIN-WAIT-2，而是发送一个对此 FIN 的 ACK 报文，并且进入 CLOSING 状态，如果之后接收到对本地之前发送的 FIN 的 ACK 报文，则直接进入 TIME-WAIT 状态。

LAST-ACK
此状态从 CLOSE-WAIT 状态变化而来。承接上文中对 CLOSE-WAIT 状态的讨论，当服务端发送完用户数据后，进行关闭操作，此时便可发送一个 FIN 报文完成通信通道的完全关闭。在服务端发送这个 FIN 报文后，有两个方面状态的改变，对于服务端，则将状态设置为 LAST-ACK，即接下来就等待最后一个客户端响应的 ACK 报文了，一旦接收到该 ACK 报文，则连接便完全关闭，服务端将状态设置为 CLOSED。而对于客户端而言，当接收到服务端发送的 FIN 时，其首先发送一个 ACK 报文给服务端，之后将状态从 FIN-WAIT-2 设置为 TIME-WAIT，表示等待 2MSL 时间（此称为静等待时间）。

TIME-WAIT
静等待状态。这是对于首先发送关闭操作的一端（一般即客户端）所最后经历的状态。应该说在接收到服务端发送的 FIN 后，本地既可设置状态为 CLOSED。之所以需要等待一段时间（通常为 2MSL：Maximum Segment Timelife）一方面是因为避免对服务端 ACK 信号的丢失，这样当服务端重传 FIN 时，客户端可以再响应一个 ACK。这个原因是次要的，因为一般对于 FIN 数据包的超时，会直接将状态设置为 CLOSED，这样处理不会造成任何有效的影响。另一个方面的主要原因是防止之后重新建立的“化生”套接字接收到老的套接字在网络中延迟的数据包。所谓“化生”套接字是指通信双方是之前的同一个信道：即具有相同的源端，目的端 IP 地址和端口号。如果之前的套接字有数据包延迟在网络中，有可能这个新建立的套接字会接收到该数据包从而造成数据误传。等待时间 2MSL 称为报文最大生存时间，顾名思义即数据包可在网络中的最大停留时间。等待这么一段时间后，可以保证“肃清”老的套接字的所有信息。2MSL 等待时间对于应用程序编程的影响即一般在关闭一个客户端套接字后，系统不会允许立刻建立一个同样通道的套接字连接，除非设置诸如 REUSEADDR 的选项。

在对以上内容理解后，读者应该不难理解如下 tcp.h 文件中内容，此处不再作论述。

```
/*include/linux/tcp.h ************************************************************/
/*
 * INET        An implementation of the TCP/IP protocol suite for the LINUX
 *        operating system.    INET is implemented using the    BSD Socket
 *        interface as the means of communication with the user level.
 *
 *        Definitions for the TCP protocol.
 *
 * Version:    @(#)tcp.h       1.0.2       04/28/93
 *
 * Author:     Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
 *
 *        This program is free software; you can redistribute it and/or
 *        modify it under the terms of the GNU General Public License
 *        as published by the Free Software Foundation; either version
 *        2 of the License, or (at your option) any later version.
 */
#ifndef _LINUX_TCP_H
```

```
#define _LINUX_TCP_H


#define HEADER_SIZE 64          /* maximum header size      */

//res 字段表示保留（reserved）。
struct tcphdr {
        __u16   source;
        __u16   dest;
        __u32   seq;
        __u32   ack_seq;
#if defined(LITTLE_ENDIAN_BITFIELD)
        __u16   res1:4,
                doff:4,
                fin:1,
                syn:1,
                rst:1,
                psh:1,
                ack:1,
                urg:1,
                res2:2;
#elif defined(BIG_ENDIAN_BITFIELD)
        __u16   res2:2,
                urg:1,
                ack:1,
                psh:1,
                rst:1,
                syn:1,
                fin:1,
                doff:4,
                res1:4;
#else
#error  "Adjust your <asm/byteorder.h> defines"
#endif
        __u16   window;
        __u16   check;
        __u16   urg_ptr;
};

//TCP 连接可能的状态
enum {
  TCP_ESTABLISHED = 1,
  TCP_SYN_SENT,
  TCP_SYN_RECV,
```

```
  TCP_FIN_WAIT1,
  TCP_FIN_WAIT2,
  TCP_TIME_WAIT,
  TCP_CLOSE,
  TCP_CLOSE_WAIT,
  TCP_LAST_ACK,
  TCP_LISTEN,
  TCP_CLOSING /* now a valid state */
};

#endif     /* _LINUX_TCP_H */
/*include/linux/tcp.h ************************************************************/
```

1.24 timer.h 头文件

该文件定义了两个系统定时器结构 timer_struct 和 timer_list 结构。timer_struct 主要用于实现内核本身所必需的定时器，而 timer_list 用于内核随需要动态添加的定时器。

```
/*include/linux/timer.h 头文件*****************************************************/
#ifndef _LINUX_TIMER_H
#define _LINUX_TIMER_H

/*
 * DON'T CHANGE THESE!! Most of them are hardcoded into some assembly language
 * as well as being defined here.
 */

/*
 * The timers are:
 *
 * BLANK_TIMER          console screen-saver timer
 *
 * BEEP_TIMER       console beep timer
 *
 * RS_TIMER         timer for the RS-232 ports
 *
 * HD_TIMER         harddisk timer
 *
 * HD_TIMER2        (atdisk2 patches)
 *
 * FLOPPY_TIMER         floppy disk timer (not used right now)
 *
 * SCSI_TIMER       scsi.c timeout timer
 *
```

```
 * NET_TIMER            tcp/ip timeout timer
 *
 * COPRO_TIMER              387 timeout for buggy hardware..
 *
 * QIC02_TAPE_TIMER     timer for QIC-02 tape driver (it's not hardcoded)
 *
 * MCD_TIMER            Mitsumi CD-ROM Timer
 *
 */

//这些系统硬编码的定时器是为完成内核必须功能而设置的定时器。
#define BLANK_TIMER      0 //文字输入竖标闪烁定时器
#define BEEP_TIMER   1 //系统铃声
#define RS_TIMER       2 //RS 串口使用

#define HD_TIMER       16 //硬盘使用
#define FLOPPY_TIMER     17 //软件使用
#define SCSI_TIMER    18 //SCSI 接口使用
#define NET_TIMER    19 //网络接口使用
#define SOUND_TIMER      20 //声卡使用
#define COPRO_TIMER      21

#define QIC02_TAPE_TIMER     22    /* hhb */
#define MCD_TIMER   23

#define HD_TIMER2    24 //第二块硬盘使用

//系统本身使用的定时器结构
struct timer_struct {
      unsigned long expires;
      void (*fn)(void);
};

/* kernel/sched.c */
extern unsigned long timer_active;
extern struct timer_struct timer_table[32];

/*
 * This is completely separate from the above, and is the
 * "new and improved" way of handling timers more dynamically.
 * Hopefully efficient and general enough for most things.
 *
 * The "hardcoded" timers above are still useful for well-
 * defined problems, but the timer-list is probably better
```

```
 * when you need multiple outstanding timers or similar.
 *
 * The "data" field is in case you want to use the same
 * timeout function for several timeouts. You can use this
 * to distinguish between the different invocations.
 */
//动态定时器结构
struct timer_list {
	struct timer_list *next;
	struct timer_list *prev;
	unsigned long expires;
	unsigned long data;
	void (*function)(unsigned long);
};

extern void add_timer(struct timer_list * timer);
extern int	del_timer(struct timer_list * timer);

extern inline void init_timer(struct timer_list * timer)
{
	timer->next = NULL;
	timer->prev = NULL;
}

#endif
/* include/linux/timer.h**********************************************************/
```

系统在 sched.c 文件中定义了 timer_head 变量：

```
static struct timer_list timer_head = { &timer_head, &timer_head, ~0, 0, NULL };
```

以及 timer_table 数组：

```
unsigned long timer_active = 0;
struct timer_struct timer_table[32];
```

这两个部分的处理均是在 timer_bh 函数中进行的，而 timer_bh 函数作为 TIMER_BH 标志的下半部分处理函数是在 do_bottom_half 函数（softirq.c)中被调用的。

换句话说，系统定时器虽然有几种类型：如 timer_struct 结构代表的定时器和 timer_list 结构代表的定时器，不过系统在处理他们时并不分类，都是作为下半部分进行处理的，即统一在 timer_bh 函数中被调用执行。有关下半部分的数据结构定义在 linux/interrupt.h, kernel/softirq.c 文件中。每个下半部分都是由一个 bh_struct 结构表示：

```
struct bh_struct {
	void (*routine)(void *);
	void *data;
};
```

而系统目前定义的下半部分共有如下类型：

```
enum {
    TIMER_BH = 0, //处理函数为 timer_bh(kernel/sched.c)
    CONSOLE_BH,
    TQUEUE_BH, //处理函数为 tqueue_bh(kernel/sched.c)
    SERIAL_BH,
    NET_BH, //处理函数为 net_bh(net/dev.c)
    IMMEDIATE_BH, //处理函数为 immediate_bh(kernel/sched.c)
    KEYBOARD_BH,
    YCLADES_BH
};
```

所有下半部分处理函数的调用执行都是在 do_bottom_half 函数中进行的，其它函数（如 do_timer, netif_rx 等等）只是对相应的标志位进行设置(每个标志位作为 bh_active 变量中的一位，使用 mark_bh 函数(linux/interrupt.h)进行设置)，从而使得相关的处理函数具有执行资格。从中可以看出，网络部分数据包接收处理函数 net_bh(dev.c)是作为 NET_BH 类型的处理函数执行的。综合而言，系统定义的所有定时器，无论是 timer_table 数组中的，还是 timer_head 链表中的，都是在 timer_bh 中被调用，do_timer 函数在每次时钟中断时被调用，其查看 timer_table 和 timer_head 中是否有定时器到期，如有，则调用 mark_bh(TIMER_BH)函数设置 TIMER_BH 标志位，从而在 do_bottom_half 函数中处理下半部分时调用 timer_bh 函数进行具体的函数处理。而 netif_rx 在接收到一个数据包时，通过调用 mark_bh(NET_BH)函数设置 NET_BH 标志位，达到如上所述同样的目的，即在 do_bottom_half 函数中调用 net_bh 处理函数进行数据包的向上传送。* 由此可见，下半部分在系统中占据着极其重要的地位。do_bottom_half 函数的被调用之处是在 entry.S 文件中，在每次系统调用返回时，都会对下半部分进行处理。从而不会影响定时器的有效性以及相关操作上的及时性。

1.25 udp.h 头文件

该文件定义 UDP 协议首部格式。

UDP 协议是无连接协议，其不提供可靠的传输，只是尽可能将数据送往接收端。在接收端没有接收到数据包时，也不提供任何手段通知发送端。如果需要提供可靠保证的数据传输，必须使用 TCP 协议。如同 TCP 协议，UDP 协议是一种传输层协议，工作在网络层协议之上（如 IP 协议）。UDP 协议是面向包的，即接收端每次都以包为单位进行接收，应用程序也是以包为单位进行数据接收，如果所分配的用户缓冲区不足，则包中多余的数据就会被直接丢弃，而非等待下一次的读取。

UDP 首部格式，如图 1-32 所示。

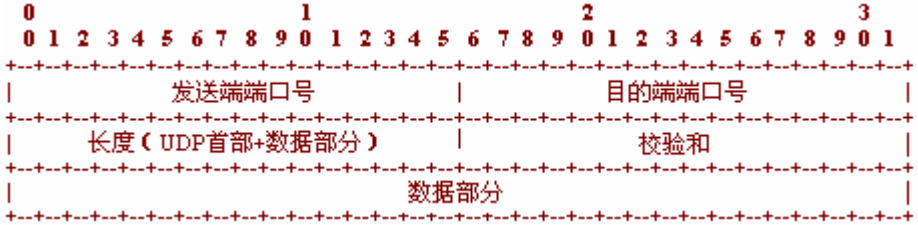


图 1-32 UDP 协议首部格式

发送端端口号：16 比特
目的端端口号：16 比特
标识该数据包源端进程和目的端进程。

长度：16 比特
该字段表示 UDP 首部与其数据部分的总长度。

校验和：16 比特
该字段计算的是 UDP 首部，数据部分和伪首部的校验和。
伪首部格式与计算 TCP 校验和时使用的格式相同，只是协议字段有所不同，如图 1-33 所示。

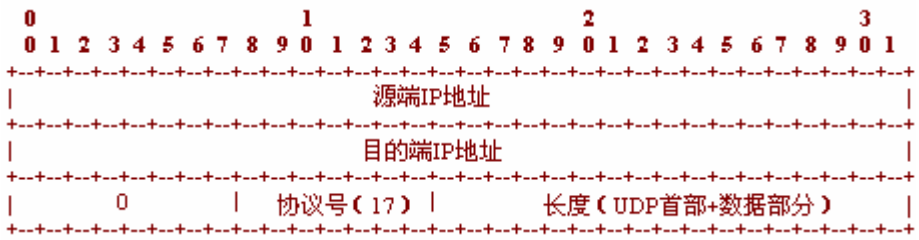


图 1-33 UDP 校验和计算中使用的伪首部格式

udp.h 文件定义了 UDP 首部结构，参考图 1-22 应不难理解。

```
/*include/linux/udp.h ************************************************************/
/*
 * INET		An implementation of the TCP/IP protocol suite for the LINUX
 *		operating system.  INET is implemented using the  BSD Socket
 *		interface as the means of communication with the user level.
 *
 *		Definitions for the UDP protocol.
 *
 * Version:	@(#)udp.h	1.0.2	04/28/93
 *
 * Author:	Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
 *
 *		This program is free software; you can redistribute it and/or
 *		modify it under the terms of the GNU General Public License
 *		as published by the Free Software Foundation; either version
 *		2 of the License, or (at your option) any later version.
 */
#ifndef _LINUX_UDP_H
#define _LINUX_UDP_H

//UDP 首部格式定义
struct udphdr {
  unsigned short	source;
  unsigned short	dest;
  unsigned short	len;
```

```
  unsigned short    check;
};


#endif    /* _LINUX_UDP_H */
/*include/linux/udp.h ************************************************************/
```

1.26 un.h 头文件

该文件定义了 UNIX 域使用的地址结构：sockaddr_un 结构。

```
/* include/linux/un.h*********************************************************/
#ifndef _LINUX_UN_H
#define _LINUX_UN_H

#define UNIX_PATH_MAX  108

struct sockaddr_un {
      unsigned short sun_family;    /* AF_UNIX */
      char sun_path[UNIX_PATH_MAX];    /* pathname */
};

#endif /* _LINUX_UN_H */
/* include/linux/un.h*********************************************************/
```

1.27 本章小结

本章着重对网络协议相关头文件进行了分析，对其中不明白的部分，读者可暂时跳过，在第二章阅读相关协议具体实现代码时可回头进行查看和理解。对于头文件中绝大部分变量和结构定义本章都作了较为详细的解释，某些无法详细说明的字段将在第二章具体代码实现中给与阐述。

第二章 网络协议实现文件分析

网络协议具体实现文件均位于 net 文件夹下，该文件夹下文件组织形式如下图（图 2-1）所示。

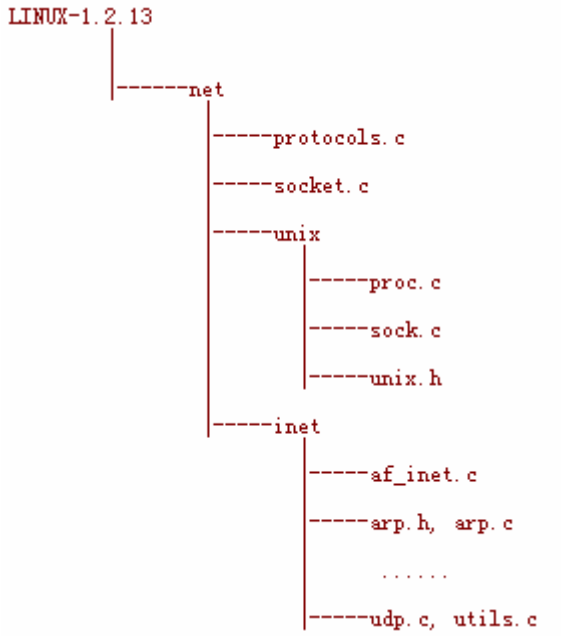


图 2-1 网络协议实现文件组织形式

其中 unix 子文件夹中三个文件是有关 UNIX 域代码，UNIX 域是模拟网络传输方式在本机范围内用于进程间数据传输的一种机制，由于不涉及网络部分且相对较为简单，本文将不对该部分进行分析。下文将集中于对 inet 子文件夹中文件进行详细介绍。不过首先需要对 net 文件夹下两个文件进行分析。

下图（图 2-2）显示了 Linux 网络栈实现与 ISO/OSI 网络栈七层分层之间的对应关系。图中另外简单给出了本版本网络栈实现中各层次的主要对应文件。如 BSD socket 层对应函数集定义在 socket.c 文件中，其中函数将作为对 socket，bind，accept 等系统调用的直接下层响应函数（sock_socket, sock_bind, sock_accept）。不过在下文对 socket.c 文件的分析中可以看出所有的网络调用函数都具有共同的入口函数 sys_socket，由该入口函数调用具体的处理函数（如 sock_socket, sock_bind 等）。BSD socket 层之下是 INET socket 层，文件 af_inet.c 对应该层次。这种上下层次的表示从函数的嵌套调用关系上体现出来。socket.c 文件中函数的实现绝大多数都是简单调用下层函数，而这些下层函数就是 af_inet.c 文件中定义的函数。例如 sock_bind 调用 inet_bind, sock_accept 调用 inet_accept 等等。之所以取名为 INET 层，恐怕与该层函数大多是以 inet 开头有关。

INET 层之下即传输层，在该层不同的协议具有不同的函数集，如 TCP 协议处理函数集定义在 tcp.c 文件中。从下文的分析中，读者可以看到，内核对一个系统调用的响应是层层下放的，而传输层才会真正进行实质上的处理，在 BSD 层和 INET 层并不进行实际处理，而是仅仅进行某些检查后，调用下一层函数：BSD 层调用 INET 层，INET 层调用传输层。这种调用在实现上是通过函数指针的方式进行的，一个套接字在不同层次有不同的数据结构表示，如在 BSD 层，用 socket 结构表示，而在 INET 层有 sock 结构表示（虽然 sock 结构不止使用在 INET 层）。BSD

层调用 INET 层函数通过 socket 结构中 ops 字段完成的。ops 字段是一个 proto_ops 结构类型，该 proto_ops 结构（下文中介绍）主要由函数指针组成。所以在 socket.c 文件中定义的函数大部分都有如下形式的调用语句：socket->ops->bind 或者 socket->ops->accept。而实际上这些函数指针指向 af_inet.c 文件中定义的函数，如 socket->ops->bind 实际上是 inet_bind, socket->ops->accept 实际上对应 inet_accept，其它类同。同理 INET 层对传输层函数的调用也是通过函数指针的形式完成，只不过在此起桥梁作用的是 sock 结构而已。sock 结构中 prot 字段是一个 proto 结构类型。该 proto 结构也是主要由一些函数指针组成。另外由于传输层所使用协议的不同，故将对应不同的函数集。如果使用 TCP 协议则 prot 字段指向 TCP 协议操作函数集，即 INET 层将调用 tcp.c 文件中定义的函数，或者如果使用 UDP 协议，则将调用 udp.c 文件中定义的函数。使用函数指针这种形式为此根据使用协议的不同调用不同的函数集合提供了很大的灵活性。在下文中对 inet_create 函数的分析中，读者可以看到，根据所使用协议的不同以及套接字类型的不同（流式？报文形式？），sock 结构的 prot 字段将初始化为不同的函数操作集合，从而实现了操作函数集合的动态分配。对于 af_inet.c 文件中定义的大多数函数都有如下的对其下层（传输层）函数的调用形式：sock->prot->bind, sock->prot->accept, 对于 TCP 协议，则这些函数指针实际上指向 tcp.c 文件中定义的 tcp_bind, tcp_accept， 其它函数调用类似。

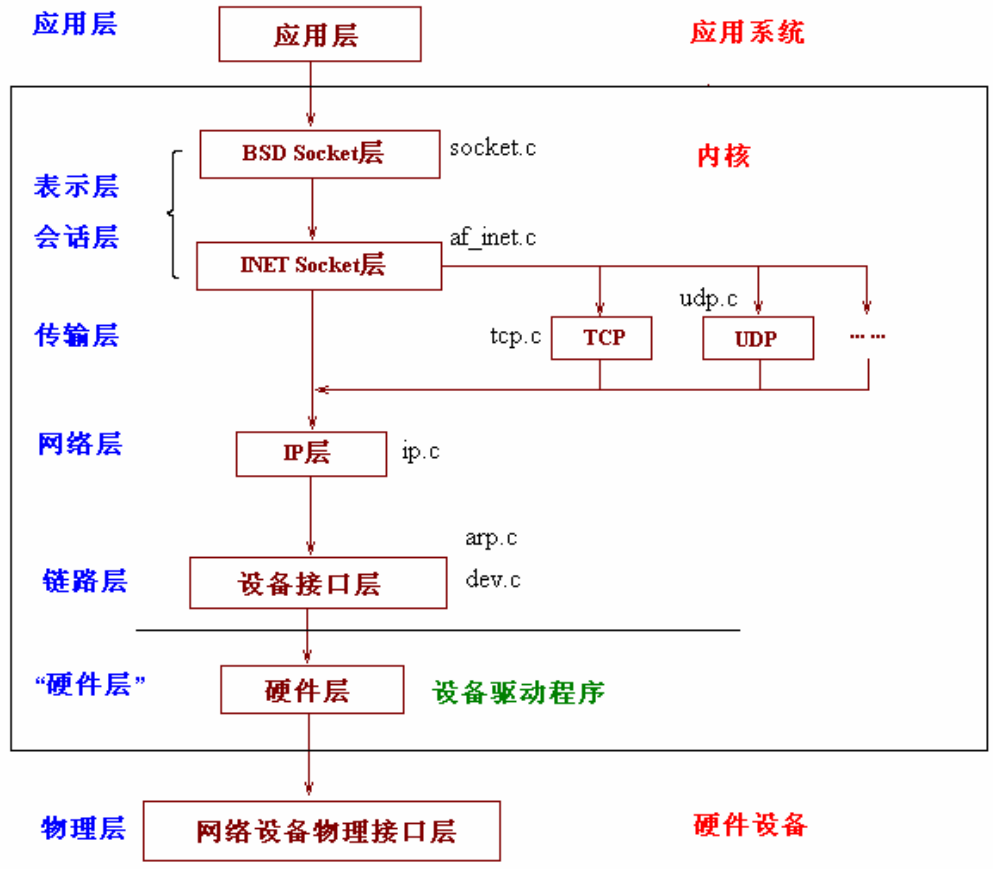


图 2-2 Linux 网络栈实现及其对应文件

2.1 net/protocols.c 文件

该文件定义了链路层所使用协议的初始化函数。主要是对一个 net_proto 数组的定义。net_proto 结构定义在 linux/net.h 文件中。

```
/*include/linux/net.h*/
115  struct net_proto {
116          char *name;             /* Protocol name */
117          void (*init_func)(struct net_proto *);     /* Bootstrap */
118  };
```

该文件内容如下：

```
/*net/protocols.c ****************************************************************/
1     /*
2       *    Protocol initializer table. Here separately for convenience
3       *
4       */



5     #include <linux/config.h>
6     #include <linux/types.h>
7     #include <linux/kernel.h>
8     #include <linux/net.h>


9     #define CONFIG_UNIX        /* always present... */

10    #ifdef      CONFIG_UNIX
11    #include "unix/unix.h"
12    #endif
13    #ifdef      CONFIG_INET
14    #include <linux/inet.h>
15    #endif
16    #ifdef CONFIG_IPX
17    #include "inet/ipxcall.h"
18    #include "inet/p8022call.h"
19    #endif
20    #ifdef CONFIG_AX25
21    #include "inet/ax25call.h"
22    #endif
23    #ifdef CONFIG_ATALK
24    #ifndef CONFIG_IPX
25    #include "inet/p8022call.h"
26    #endif
27    #include "inet/atalkcall.h"
28    #endif
29    #include "inet/psnapcall.h"

30    /*
```

```
31    *    Protocol Table
32    */

33   struct net_proto protocols[] = {
34   #ifdef      CONFIG_UNIX
35   { "UNIX",      unix_proto_init      },//Unix 域，本机内模拟网络通信方式进行进程间数据传送
36   #endif
37   #if defined(CONFIG_IPX)||defined(CONFIG_ATALK)
38   { "802.2",      p8022_proto_init }, //802.2，SNAP 是链路层子协议，与 802.3 头部配合使用
39   { "SNAP",      snap_proto_init },
40   #endif
41   #ifdef CONFIG_AX25
42   { "AX.25",      ax25_proto_init },
43   #endif
44   #ifdef      CONFIG_INET
45   { "INET",       inet_proto_init}, //INET 域，网络栈常用协议初始化
46   #endif
47   #ifdef    CONFIG_IPX
48   { "IPX",  ipx_proto_init },//IPX 协议，网络层协议，ipx_proto_init 用于初始化
49   #endif
50   #ifdef CONFIG_ATALK
51   { "DDP", atalk_proto_init },//Appletalk 协议初始化
52   #endif
53   { NULL, NULL            }
54   };
/*net/protocols.c ************************************************************/
```

该文件定义了一系列条件语句，根据某些变量的声明与否定义相关的初始化函数。protocols 数组包含了一系列初始化函数声明。在网络栈初始化阶段，protocols 数组定义的这些初始化函数将被调用。

2.2 net/socket.c 文件

该文件的定义的函数集，我们通常意义上可以将其 BSD socket 层对应的函数来看待。它是系统调用与下层网络栈实现函数之间的桥梁。系统调用通过 INT $0x80 进入内核执行函数，该函数根据 AX 寄存器中的系统调用号，进一步调用内核网络栈相应的实现函数。对于 socket，bind，connect，sendto，recvfrom 等这些网络栈直接操作函数而言，则 socket.c 文件中定义的函数是作为网络栈的最上层实现函数，即第一层被调用的函数。对于不同的协议具有的不同操作方式，只有到达传输层才可以处理具体的请求，所以在传输层之上的所有层面操作函数集，其实现的功能较为直接：只是检查某些字段或者标志位，然后将请求进一步传替给下层处理函数。对于标准的 TCP/IP 四层结构，我们通常将链路层称为 L2 层，依次类推，传输层即为 L4 层，应用层为 L5 层，则 socket.c 文件中定义的函数集合可以称之为 L6 层，即该文件中定义函数是一个通用的层面，它只是系统调用层与网络栈函数集合的接口层，所以可以预见该文件中定义的函数实现均较为简单，因为它们大多数实现的功能只是调用下一层函数进行处理。

虽然 Linux 中几乎所有的接口都是以文件形式组织的，但对于网络栈在/dev 目录下却无这样的

对应关系。不过内核网络栈实现仍然提供了对于网络数据的普通文件操作方式，如 write，read 函数可直接用于读写网络数据，在 socket.c 文件中可以看到内核提供的针对网络数据的文件操作函数集合的实现。

由于 socket.c 文件相当长，故我们分段进行分析。

```
1     /*
2      * NET        An implementation of the SOCKET network access protocol.
3      *
4      * Version:    @(#)socket.c  1.1.93  18/02/95
5      *
6      * Authors:    Orest Zborowski, <obz@Kodak.COM>
7      *         Ross Biro, <bir7@leland.Stanford.Edu>
8      *         Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
9      *
10     * Fixes:
11     *         Anonymous   :   NOTSOCK/BADF cleanup. Error fix in
12     *                         shutdown()
13     *         Alan Cox:   verify_area() fixes
14     *         Alan Cox:   Removed DDI
15     *         Jonathan Kamens   :   SOCK_DGRAM reconnect bug
16     *         Alan Cox:   Moved a load of checks to the very
17     *                         top level.
18     *         Alan Cox:   Move address structures to/from user
19     *                         mode above the protocol layers.
20     *         Rob Janssen   :   Allow 0 length sends.
21     *         Alan Cox:   Asynchronous I/O support (cribbed from the
22     *                         tty drivers).
23     *         Niibe Yutaka  :   Asynchronous I/O for writes (4.4BSD style)
24     *         Jeff Uphoff   :   Made max number of sockets command-line
25     *                         configurable.
26     *         Matti Aarnio  :   Made the number of sockets dynamic,
27     *                         to be allocated when needed, and mr.
28     *                         Uphoff's max is used as max to be
29     *                         allowed to allocate.
30     *         Linus       :   Argh. removed all the socket allocation
31     *                         altogether: it's in the inode now.
32     *         Alan Cox:   Made sock_alloc()/sock_release() public
33     *                         for NetROM and future kernel nfsd type
34     *                         stuff.
35     *
36     *
37     *         This program is free software; you can redistribute it and/or
38     *         modify it under the terms of the GNU General Public License
39     *         as published by the Free Software Foundation; either version
40     *         2 of the License, or (at your option) any later version.
```

```
41    *
42    *
43    *    This module is effectively the top level interface to the BSD socket
44    *    paradigm. Because it is very simple it works well for Unix domain sockets,
45    *    but requires a whole layer of substructure for the other protocols.
46    *
47    *    In addition it lacks an effective kernel -> kernel interface to go with
48    *    the user one.
49    */

50  #include <linux/config.h>
51  #include <linux/signal.h>
52  #include <linux/errno.h>
53  #include <linux/sched.h>
54  #include <linux/mm.h>
55  #include <linux/kernel.h>
56  #include <linux/major.h>
57  #include <linux/stat.h>
58  #include <linux/socket.h>
59  #include <linux/fcntl.h>
60  #include <linux/net.h>
61  #include <linux/interrupt.h>
62  #include <linux/netdevice.h>

63  #include <asm/system.h>
64  #include <asm/segment.h>
```

以上为 socket.c 对其引入头文件的声明。紧接着即是对网络数据提供普通文件操作接口函数的声明以及相关数据结构（file_operations）的初始化。file_operations 结构定义了普通文件操作函数集。系统中每个文件对应一个 file 结构，file 结构中有一个 file_operations 变量，当使用 write，read 函数对某个文件描述符进行读写操作时，系统首先根据文件描述符索引到其对应的 file 结构，然后调用其成员变量 file_operations 中对应函数完成请求。

```
65  static int sock_lseek(struct inode *inode, struct file *file, off_t offset,
66                        int whence);
67  static int sock_read(struct inode *inode, struct file *file, char *buf,
68                       int size);
69  static int sock_write(struct inode *inode, struct file *file, char *buf,
70                        int size);
71  static int sock_readdir(struct inode *inode, struct file *file,
72                          struct dirent *dirent, int count);
73  static void sock_close(struct inode *inode, struct file *file);
74  static int sock_select(struct inode *inode, struct file *file, int which, select_table *seltable);
75  static int sock_ioctl(struct inode *inode, struct file *file,
```

```
76                      unsigned int cmd, unsigned long arg);
77  static int sock_fasync(struct inode *inode, struct file *filp, int on);
```

```
78  /*
79   *  Socket files have a set of 'special' operations as well as the generic file ones. These don't appear
80   *  in the operation structures but are done directly via the socketcall() multiplexor.
81   */

82  static struct file_operations socket_file_ops = {
83      sock_lseek,
84      sock_read,
85      sock_write,
86      sock_readdir,
87      sock_select,
88      sock_ioctl,
89      NULL,           /* mmap */
90      NULL,           /* no special open code... */
91      sock_close,
92      NULL,           /* no fsync */
93      sock_fasync
94  };
```

以上 socket_file_ops 变量中声明的函数即是网络协议对应的普通文件操作函数集合。从而使得 read，write，ioctl 等这些常见普通文件操作函数也可以被使用在网络接口的处理上。
值得注意的是，这些提供的普通文件接口中并没有定义 open 函数，因为对于网络而言，socket 函数已经完成了类似的功能，并且 open 函数基本调用语义是通过文件路径进行操作，然而诚如上文中所述，对于网络栈内核并无这样的对应文件。

```
95  /*
96   *  The protocol list. Each protocol is registered in here.
97   */
98  static struct proto_ops *pops[NPROTO];
99  /*
100  *  Statistics counters of the socket lists
101  */
102 static int sockets_in_use  = 0;
```

proto_ops 结构定义在 include/linux/net.h 文件中，该结构中声明了一系列操作函数作为 L6 层处理函数。pops 数组将在 sock_register 函数中被初始化，对于不同的操作域，具有不同的操作函数集，如对应 INET 域的 inet_proto_ops 操作函数集，UNIX 域对应 unix_proto_ops 操作函数集。socket_in_use 变量定义了系统当前正在使用的套接字总数目。

```
103 /*
104  *  Support routines. Move socket addresses back and forth across the kernel/user
105  *  divide and look after the messy bits.
106  */

107 #define MAX_SOCK_ADDR   128     /* 108 for Unix domain - 16 for IP, 16 for IPX, about 80 for AX.25 */

108 static int move_addr_to_kernel(void *uaddr, int ulen, void *kaddr)
109 {
110     int err;
111     if(ulen<0||ulen>MAX_SOCK_ADDR)
112         return -EINVAL;
113     if(ulen==0)
114         return 0;
115     if((err=verify_area(VERIFY_READ,uaddr,ulen))<0)
116         return err;
117     memcpy_fromfs(kaddr,uaddr,ulen);
118     return 0;
119 }

120 static int move_addr_to_user(void *kaddr, int klen, void *uaddr, int *ulen)
121 {
122     int err;
123     int len;


124     if((err=verify_area(VERIFY_WRITE,ulen,sizeof(*ulen)))<0)
125         return err;
126     len=get_fs_long(ulen);
127     if(len>klen)
128         len=klen;
129     if(len<0 || len> MAX_SOCK_ADDR)
130         return -EINVAL;
131     if(len)
132     {
133         if((err=verify_area(VERIFY_WRITE,uaddr,len))<0)
134             return err;
135         memcpy_tofs(uaddr,kaddr,len);
136     }
137     put_fs_long(len,ulen);
138     return 0;
139 }
```

move_addr_to_kernel, move_addr_to_user 这两个函数实现的功能较为直接，即实现地址用户空间和内核空间之间的相互移动，主要是通过调用底层函数 memcpy_tofs 和 memcpy_fromfs 实现的。

```
140 /*
141  *    Obtains the first available file descriptor and sets it up for use.
142  */

143 static int get_fd(struct inode *inode)
144 {
145         int fd;
146         struct file *file;

147         /*
148          *    Find a file descriptor suitable for return to the user.
149          */

150         file = get_empty_filp();
151         if (!file)
152                 return(-1);

153         for (fd = 0; fd < NR_OPEN; ++fd)
154                 if (!current->files->fd[fd])
155                         break;
156         if (fd == NR_OPEN)
157         {
158                 file->f_count = 0;
159                 return(-1);
160         }

161         FD_CLR(fd, &current->files->close_on_exec);
162                 current->files->fd[fd] = file;
163         file->f_op = &socket_file_ops;
164         file->f_mode = 3;
165         file->f_flags = O_RDWR;
166         file->f_count = 1;
167         file->f_inode = inode;
168         if (inode)
169                 inode->i_count++;
170         file->f_pos = 0;
171         return(fd);
172 }
```

函数 get_fd 是为网络套接字分配一个文件描述符，在 socket 系统调用处理函数 sys_socket 实现中，内核在分配 inode，socket，sock 结构后，调用 get_fd 获得一个文件描述符作为 socket 系统

调用的返回值。注意函数的参数是一个 inode 结构，因为分配文件描述符的同时需要一个 file 结构，file 结构中 f_inode 字段即指向这个 inode 结构，每个 file 结构都需要有一个 inode 结构对应。注意函数中 f_op 字段的赋值即前文中定义的 socket_file_ops，通过此处的赋值即实现了网络操作的普通文件接口。如果对文件描述符调用 write，read 函数进行操作时，即调用 socket_file_ops 变量中对应的 sock_read, sock_write 函数。

file 结构的获取是通过 get_empty_flip 函数实现的。内核维护一个 file 结构的数组，get_empty_filp 函数即通过检查该数组，获取一个闲置的成员。

```
173 /*
174  *    Go from an inode to its socket slot.
175  *
176  * The original socket implementation wasn't very clever, which is
177  * why this exists at all..
178  */
179 inline struct socket *socki_lookup(struct inode *inode)
180 {
181         return &inode->u.socket_i;
182 }
```

socki_lookup 函数即通过 inode 结构查找对应的 socket 结构，socket 结构可以看作是网络栈操作在 L6，L5 层的表示。从实现来看，socket 结构是作为 inode 结构中一个变量。这点可以从 inode 结构的具体定义看出，此处给出 inode 结构的设计此部分的定义。

```
/*include/linux/fs.h*/
172 struct inode {
            ……
206         union {
207                 struct pipe_inode_info pipe_i;
208                 struct minix_inode_info minix_i;
209                 struct ext_inode_info ext_i;
210                 struct ext2_inode_info ext2_i;
211                 struct hpfs_inode_info hpfs_i;
212                 struct msdos_inode_info msdos_i;
213                 struct umsdos_inode_info umsdos_i;
214                 struct iso_inode_info isofs_i;
215                 struct nfs_inode_info nfs_i;
216                 struct xiafs_inode_info xiafs_i;
217                 struct sysv_inode_info sysv_i;
218                 struct socket socket_i;
219                 void * generic_ip;
220         } u;
221 };
```

从中可以看出 inode 结构中最后一个变量是一个 union 结构，该结构中又定义了一系列针对不同

文件系统的信息结构如对应 ext2 文件系统的 ext2_i 结构，而对于网络而言则对应 socket_i，此时这是一个 socket 结构。

```
183 /*
184  *  Go from a file number to its socket slot.
185  */

186 static inline struct socket *sockfd_lookup(int fd, struct file **pfile)
187 {
188     struct file *file;
189     struct inode *inode;

190     if (fd < 0 || fd >= NR_OPEN || !(file = current->files->fd[fd]))
191         return NULL;

192     inode = file->f_inode;
193     if (!inode || !inode->i_sock)
194         return NULL;

195     if (pfile)
196         *pfile = file;

197     return socki_lookup(inode);
198 }
```

sockfd_lookup 函数实现较为直接，从文件描述符得到其对应的 file 结构，进而得到 inode 结构，然后调用 socki_lookup 函数返回 socket 结构。

```
199 /*
200  *  Allocate a socket.
201  */

202 struct socket *sock_alloc(void)
203 {
204     struct inode * inode;
205     struct socket * sock;

206     inode = get_empty_inode();
207     if (!inode)
208         return NULL;

209     inode->i_mode = S_IFSOCK;
210     inode->i_sock = 1;
211     inode->i_uid = current->uid;
```

```
212         inode->i_gid = current->gid;

213         sock = &inode->u.socket_i;
214         sock->state = SS_UNCONNECTED;
215         sock->flags = 0;
216         sock->ops = NULL;
217         sock->data = NULL;
218         sock->conn = NULL;
219         sock->iconn = NULL;
220         sock->next = NULL;
221         sock->wait = &inode->i_wait;
222         sock->inode = inode;            /* "backlink": we could use pointer arithmetic instead */
223         sock->fasync_list = NULL;
224         sockets_in_use++;
225         return sock;
226 }
```

sock_alloc 函数用于分配一个 socket 结构，从实现来看这个函数实质上是获取一个 inode 结构，get_empty_inode 结构实现的功能基本类同于 get_empty_filp，该函数通过检查系统维护的一个 inode 结构数组，获取一个空闲的 inode 结构返回。之后对此 inode 结构的部分字段进行初始化，之后返回需要的 socket 结构。

```
227 /*
228  *    Release a socket.
229  */

230 static inline void sock_release_peer(struct socket *peer)
231 {
232         peer->state = SS_DISCONNECTING;
233         wake_up_interruptible(peer->wait);
234         sock_wake_async(peer, 1);
235 }

236 void sock_release(struct socket *sock)
237 {
238         int oldstate;
239         struct socket *peersock, *nextsock;

240         if ((oldstate = sock->state) != SS_UNCONNECTED)
241                 sock->state = SS_DISCONNECTING;

242         /*
243          *    Wake up anyone waiting for connections.
244          */
```

```
245     for (peersock = sock->iconn; peersock; peersock = nextsock)
246     {
247         nextsock = peersock->next;
248         sock_release_peer(peersock);
249     }

250     /*
251       * Wake up anyone we're connected to. First, we release the
252       * protocol, to give it a chance to flush data, etc.
253       */

254     peersock = (oldstate == SS_CONNECTED) ? sock->conn : NULL;
255     if (sock->ops)
256         sock->ops->release(sock, peersock);
257     if (peersock)
258         sock_release_peer(peersock);
259     --sockets_in_use;   /* Bookkeeping.. */
260     iput(SOCK_INODE(sock));
261 }
```

sock_release_peer 仅用于 UNIX 域，对于 INET 域 socket 结构中 conn，iconn 字段为 NULL。因为对于 INET 域而言，与远端主机的连接是一一对应的。

sock_release 函数用于释放（关闭）一个套接字。其中需要注意的是对 sock->ops->release 函数的调用，sock->ops 字段是一个 proto_ops 结构，proto_ops 结构是一个操作函数集合定义，这个集合实质上是一个接口函数集，换句话说，函数本身并不完成具体功能，而是将请求送往下层函数。对于 INET 域而言，此时 sock->ops 字段将被赋值为 inet_proto_ops 变量，该变量在 net/inet/af_inet.c 文件中定义：

```
/*net/inet/af_inet.c*/
1313    static struct proto_ops inet_proto_ops = {
1314        AF_INET,

1315        inet_create,
1316        inet_dup,
1317        inet_release,
1318        inet_bind,
1319        inet_connect,
1320        inet_socketpair,
1321        inet_accept,
1322        inet_getname,
1323        inet_read,
1324        inet_write,
1325        inet_select,
1326        inet_ioctl,
```

```
1327            inet_listen,
1328            inet_send,
1329            inet_recv,
1330            inet_sendto,
1331            inet_recvfrom,
1332            inet_shutdown,
1333            inet_setsockopt,
1334            inet_getsockopt,
1335            inet_fcntl,
1336    };
```

从中可以看出实际上对于 release 函数的调用具体是 inet_release。而 inet_release 也仅仅是一个接口函数，其将通过进一步调用下层函数完成具体的请求。在分析 af_inet.c 文件时再作具体说明。

```
353 void sock_close(struct inode *inode, struct file *filp)
354 {
355     struct socket *sock;

356     /*
357      *  It's possible the inode is NULL if we're closing an unfinished socket.
358      */

359     if (!inode)
360         return;

361     if (!(sock = socki_lookup(inode)))
362     {
363         printk("NET: sock_close: can't find socket for inode!\n");
364         return;
365     }
366     sock_fasync(inode, filp, 0);
367     sock_release(sock);
368 }
```

此处将 sock_close 函数提前，因为 sock_close，sock_release, sock_release_peer 这三个函数都是用于套接字的关闭操作且相互嵌套。sock_close 调用 sock_release，sock_release 调用 sock_release_peer. sock_fasync 函数如下所示。

```
369 /*
370  *  Update the socket async list
371  */

372 static int sock_fasync(struct inode *inode, struct file *filp, int on)
373 {
374     struct fasync_struct *fa, *fna=NULL, **prev;
```

```
375     struct socket *sock;
376     unsigned long flags;

377     if (on)
378     {
379         fna=(struct fasync_struct *)kmalloc(sizeof(struct fasync_struct), GFP_KERNEL);
380         if(fna==NULL)
381             return -ENOMEM;
382     }

383     sock = socki_lookup(inode);

384     prev=&(sock->fasync_list);

385     save_flags(flags);
386     cli();

387     for(fa=*prev; fa!=NULL; prev=&fa->fa_next,fa=*prev)
388         if(fa->fa_file==filp)
389             break;

390     if(on)
391     {
392         if(fa!=NULL)
393         {
394             kfree_s(fna,sizeof(struct fasync_struct));
395             restore_flags(flags);
396             return 0;
397         }
398         fna->fa_file=filp;
399         fna->magic=FASYNC_MAGIC;
400         fna->fa_next=sock->fasync_list;
401         sock->fasync_list=fna;
402     }
403     else
404     {
405         if(fa!=NULL)
406         {
407             *prev=fa->fa_next;
408             kfree_s(fa,sizeof(struct fasync_struct));
409         }
410     }
411     restore_flags(flags);
412     return 0;
```

```
413 }
```

该函数将根据输入参数 on 的取值决定是分配还是释放一个 fasync_struct 结构。

```
/*include/linux/fs.h*/
246 struct fasync_struct {
247         int       magic;
248         struct fasync_struct *fa_next; /* singly linked list */
249         struct file              *fa_file;
250 };

251 #define FASYNC_MAGIC 0x4601
```

fasync_struct 结构用于异步唤醒。结构中 fa_file 是一个 file 结构，file 结构中 f_owner 指向该 file 结构对应文件所属的属主，所谓属主即打开该文件进行操作的某个进程，f_owner 即指向该进程进程号。sock_fasync 函数虽然较长，但实现功能简单，此处不多作表述。

socket 结构中最后一个字段 fasync_list 是一个 fasync_struct 结构。如上文所述，该结构用于异步唤醒，对应的唤醒函数为 sock_wake_async。

```
/*include/linux/net.h*/
62    struct socket {
          …
73        struct fasync_struct    *fasync_list;    /* Asynchronous wake up list */
74    };
```

sock_wake_async 函数实现如下文所示。

```
/*如无特别指出，函数定义在 socket.c 文件中*/
414 int sock_wake_async(struct socket *sock, int how)
415 {
416         if (!sock || !sock->fasync_list)
417                 return -1;
418         switch (how)
419         {
420                 case 0:
421                         kill_fasync(sock->fasync_list, SIGIO);
422                         break;
423                 case 1:
424                         if (!(sock->flags & SO_WAITDATA))
425                                 kill_fasync(sock->fasync_list, SIGIO);
426                         break;
427                 case 2:
428                         if (sock->flags & SO_NOSPACE)
429                         {
430                                 kill_fasync(sock->fasync_list, SIGIO);
431                                 sock->flags &= ~SO_NOSPACE;
```

```
432                     }
433                     break;
434         }
435         return 0;
436 }
```

sock_wake_async 函数在 sock_release_peer, sock_awaitconn 函数中调用。sock_release_peer 在释放一个套接字被调用，sock_awaitconn 在试图建立一个连接时被调用，这些调用或者涉及到套接字状态的改变，所以如果有进程等待套接字的状态改变，则需要在改变时通知相应进程或者需要唤醒相关进程进行某种处理，如唤醒服务器端监听进程进行客户端套接字连接的处理。该函数通过遍历 socket 结构中 fasync_list 变量指向的队列，对队列中每个元素调用 kill_fasync 函数，kill_fasync 函数如下所示。

```
/*fs/fcntl.c*/
void kill_fasync(struct fasync_struct *fa, int sig)
{
    while (fa) {
        if (fa->magic != FASYNC_MAGIC) {
            printk("kill_fasync: bad magic number in "
                    "fasync_struct!\n");
            return;
        }
        if (fa->fa_file->f_owner > 0)
            kill_proc(fa->fa_file->f_owner, sig, 1);
        else
            kill_pg(-fa->fa_file->f_owner, sig, 1);
        fa = fa->fa_next;
    }
}
```

从 kill_fasync 函数的实现来看，所谓的异步唤醒功能就一目了然了，即通过向相应的进程发送信号。

以下 6 个函数用于建立网络数据的普通文件操作接口，分别为 sock_lseek, sock_read, sock_write, sock_readdir, sock_select, sock_ioctl. 其中 sock_lseek, sock_readdir 未实现。sock_read, sock_write 这两个函数用于读写网络数据，在具体实现上，均是在完成部分检查后，调用下层函数（inet_read, inet_write）完成具体功能。而 sock_select, sock_ioctl 也是类似的实现，通过调用下层函数 inet_select, inet_ioctl 完成请求。

```
262 /*
263  *  Sockets are not seekable.
264  */

265 static int sock_lseek(struct inode *inode, struct file *file, off_t offset, int whence)
266 {
267         return(-ESPIPE);
```

```
268 }


269 /*
270  *    Read data from a socket. ubuf is a user mode pointer. We make sure the user
271  *    area ubuf...ubuf+size-1 is writable before asking the protocol.
272  */

273 static int sock_read(struct inode *inode, struct file *file, char *ubuf, int size)
274 {
275         struct socket *sock;
276         int err;

277         if (!(sock = socki_lookup(inode)))
278         {
279                 printk("NET: sock_read: can't find socket for inode!\n");
280                 return(-EBADF);
281         }
282         if (sock->flags & SO_ACCEPTCON)
283                 return(-EINVAL);

284         if(size<0)
285                 return -EINVAL;
286         if(size==0)
287                 return 0;
288         if ((err=verify_area(VERIFY_WRITE,ubuf,size))<0)
289                 return err;
290         return(sock->ops->read(sock, ubuf, size, (file->f_flags & O_NONBLOCK)));
291 }

292 /*
293  *    Write data to a socket. We verify that the user area ubuf..ubuf+size-1 is
294  *    readable by the user process.
295  */

296 static int sock_write(struct inode *inode, struct file *file, char *ubuf, int size)
297 {
298         struct socket *sock;
299         int err;

300         if (!(sock = socki_lookup(inode)))
301         {
302                 printk("NET: sock_write: can't find socket for inode!\n");
303                 return(-EBADF);
```

```
304         }

305         if (sock->flags & SO_ACCEPTCON)
306                 return(-EINVAL);

307         if(size<0)
308                 return -EINVAL;
309         if(size==0)
310                 return 0;

311         if ((err=verify_area(VERIFY_READ,ubuf,size))<0)
312                 return err;
313         return(sock->ops->write(sock, ubuf, size,(file->f_flags & O_NONBLOCK)));
314 }

315 /*
316  *    You can't read directories from a socket!
317  */

318 static int sock_readdir(struct inode *inode, struct file *file, struct dirent *dirent,
319                 int count)
320 {
321         return(-EBADF);
322 }

323 /*
324  *    With an ioctl arg may well be a user mode pointer, but we don't know what to do
325  *    with it - thats up to the protocol still.
326  */

327 int sock_ioctl(struct inode *inode, struct file *file, unsigned int cmd,
328             unsigned long arg)
329 {
330         struct socket *sock;

331         if (!(sock = socki_lookup(inode)))
332         {
333                 printk("NET: sock_ioctl: can't find socket for inode!\n");
334                 return(-EBADF);
335         }
336         return(sock->ops->ioctl(sock, cmd, arg));
337 }
```

```
338 static int sock_select(struct inode *inode, struct file *file, int sel_type, select_table * wait)
339 {
340         struct socket *sock;

341         if (!(sock = socki_lookup(inode)))
342         {
343                 printk("NET: sock_select: can't find socket for inode!\n");
344                 return(0);
345         }

346         /*
347          *      We can't return errors to select, so it's either yes or no.
348          */

349         if (sock->ops && sock->ops->select)
350                 return(sock->ops->select(sock, sel_type, wait));
351         return(0);
352 }
```

sock_awaitconn 函数只用于 UNIX 域，用于处理一个客户端连接请求。socket 结构中 iconn，conn 结构用于 UNIX 域中连接操作，其中 iconn 只用于服务器端，表示等待连接但尚未完成连接的客户端 socket 结构链表。

```
437 /*
438  *      Wait for a connection.
439  */

440 int sock_awaitconn(struct socket *mysock, struct socket *servsock, int flags)
441 {
442         struct socket *last;

443         /*
444          *      We must be listening
445          */
    //检查服务器端是否是处于侦听状态，即可以进行连接。
446         if (!(servsock->flags & SO_ACCEPTCON))
447         {
448                 return(-EINVAL);
449         }

450         /*
451          *      Put ourselves on the server's incomplete connection queue.
452          */
```

```
453         mysock->next = NULL;
454         cli();
    //将本次客户端连接的套接字（socket 结构）插入服务器端 socket 结构 iconn 字段指向
    //的链表，即表示客户端正等待连接。
455         if (!(last = servsock->iconn))
456             servsock->iconn = mysock;
457         else
458         {
459             while (last->next)
460                 last = last->next;
461             last->next = mysock;
462         }
463         mysock->state = SS_CONNECTING;
464         mysock->conn = servsock;
465         sti();

466         /*
467          * Wake up server, then await connection. server will set state to
468          * SS_CONNECTED if we're connected.
469          */
    //唤醒服务器端进程，以处理本地客户端连接。
470         wake_up_interruptible(servsock->wait);

    //注意对 sock_wake_async 函数的调用，sock_wake_async 函数将唤醒 servsock 结构中
    //fasync_list 指向的队列中每个元素对应的进程。
471         sock_wake_async(servsock, 0);

    //由于唤醒服务器端需要一定延迟时间，所以这个 if 判断条件一般为 true。
472         if (mysock->state != SS_CONNECTED)
473         {
474             if (flags & O_NONBLOCK)
475                 return -EINPROGRESS;

            //等待服务器端处理本次连接。
476             interruptible_sleep_on(mysock->wait);

        //当本地客户端被唤醒后，继续检查连接状态，以察看是否已完成与服务器的连接。
477             if (mysock->state != SS_CONNECTED &&
478                 mysock->state != SS_DISCONNECTING)
479             {
480             /*
481              * if we're not connected we could have been
482              * 1) interrupted, so we need to remove ourselves
483              *      from the server list
```

```
484             * 2) rejected (mysock->conn == NULL), and have
485             *       already been removed from the list
486             */
                //如果仍然没有建立连接，则原因有二，一是服务器端拒绝连接，此时本地
                //socket 结构已被服务器端从其等待队列 iconn 中删除；二是在睡眠中被其它
                //情况中断但并未建立连接，此时需要主动将本地 socket 结构从对方服务器端
                //等待连接队列 iconn 中删除。
487             if (mysock->conn == servsock)
488             {
489                 cli();
490                 if ((last = servsock->iconn) == mysock)
491                     servsock->iconn = mysock->next;
492                 else
493                 {
494                     while (last->next != mysock)
495                         last = last->next;
496                     last->next = mysock->next;
497                 }
498                 sti();
499             }
                //如果 mysock->conn 为 NULL，则表示服务器拒绝连接，返回 EACCESS。
                //否则返回被中断标志：EINTR。
500             return(mysock->conn ? -EINTR : -EACCES);
501         }
502     }
503     return(0);
504 }
```

以下主要是对网络套接字专用函数的底层对应函数的实现。

```
/*
505 /*
506  *  Perform the socket system call. we locate the appropriate
507  *  family, then create a fresh socket.
508  */

509 static int sock_socket(int family, int type, int protocol)
510 {
511     int i, fd;
512     struct socket *sock;
513     struct proto_ops *ops;

514     /* Locate the correct protocol family. */
515     for (i = 0; i < NPROTO; ++i)
516     {
```

```
517             if (pops[i] == NULL) continue;
518             if (pops[i]->family == family)
519                     break;
520     }

521     if (i == NPROTO)
522     {
523             return -EINVAL;
524     }

525     ops = pops[i];

526 /*
527  *  Check that this is a type that we know how to manipulate and
528  *  the protocol makes sense here. The family can still reject the
529  *  protocol later.
530  */
```

//此下判断参数的有效性。这些标志均定义在 linux/socket.h 文件中。
//SOCK_STREAM:使用流式数据交换，如 TCP
//SOCK_DGRAM:使用报文数据交换，如 UDP
//SOCK_SEQPACKET:序列报文套接字格式，内核处理方式如同流式报文。
//SOCK_RAW:原始套接字，直接在传输层收发数据，应用程序须自行建立传输层首部
//SOCK_PACKET:包类型套接字，直接在网络层收发数据，应用程序须自行建立网络层，
//传输层首部

```
531     if ((type != SOCK_STREAM && type != SOCK_DGRAM &&
532             type != SOCK_SEQPACKET && type != SOCK_RAW &&
533             type != SOCK_PACKET) || protocol < 0)
534                     return(-EINVAL);

535 /*
536  *  Allocate the socket and allow the family to set things up. if
537  *  the protocol is 0, the family is instructed to select an appropriate
538  *  default.
539  */

540     if (!(sock = sock_alloc()))
541     {
542             printk("NET: sock_socket: no more sockets\n");
543             return(-ENOSR);    /* Was: EAGAIN, but we are out of
544                                     system resources! */
545     }
```

```
546         sock->type = type;
547         sock->ops = ops;
548         if ((i = sock->ops->create(sock, protocol)) < 0)
549         {
550                 sock_release(sock);
551                 return(i);
552         }

553         if ((fd = get_fd(SOCK_INODE(sock))) < 0)
554         {
555                 sock_release(sock);
556                 return(-EINVAL);
557         }

558         return(fd);
559 }
```

sock_socket 函数对应 socket 系统调用。该函数首先根据 family 输入参数决定域操作函数集用于 socket 结构中 ops 字段的赋值，如 inet_proto_ops 或者 unix_proto_ops。之后分配 inode 结构和 socket 结构（通过调用 sock_alloc），并对 socket 结构中 ops 字段赋值。之后调用 sock->ops->create 指向的函数（如 inet_create，该函数在本书下文中对应部分进行分析）分配下层 sock 结构，sock 结构是比 socket 结构更底层的表示一个套接字连接的结构。最后调用 get_fd 分配一个文件描述符（及其对应的 file 结构）并返回。
故 sock_socket 函数完成如下工作：
1>分配 socket，sock 结构，这两个结构在网络栈的不同层次表示一个套接字连接。
2>分配 inode，file 结构用于普通文件操作。
3>分配一个文件描述符并返回给应用程序作为以后的操作句柄。

```
560 /*
561  *  Create a pair of connected sockets.
562  */

563 static int sock_socketpair(int family, int type, int protocol, unsigned long usockvec[2])
564 {
565         int fd1, fd2, i;
566         struct socket *sock1, *sock2;
567         int er;

568         /*
569          * Obtain the first socket and check if the underlying protocol
570          * supports the socketpair call.
571          */

572         if ((fd1 = sock_socket(family, type, protocol)) < 0)
```

```
573             return(fd1);
574         sock1 = sockfd_lookup(fd1, NULL);
575         if (!sock1->ops->socketpair)
576         {
577             sys_close(fd1);
578             return(-EINVAL);
579         }

580         /*
581          *    Now grab another socket and try to connect the two together.
582          */

583         if ((fd2 = sock_socket(family, type, protocol)) < 0)
584         {
585             sys_close(fd1);
586             return(-EINVAL);
587         }

588         sock2 = sockfd_lookup(fd2, NULL);
589         if ((i = sock1->ops->socketpair(sock1, sock2)) < 0)
590         {
591             sys_close(fd1);
592             sys_close(fd2);
593             return(i);
594         }

595         sock1->conn = sock2;
596         sock2->conn = sock1;
597         sock1->state = SS_CONNECTED;
598         sock2->state = SS_CONNECTED;

599         er=verify_area(VERIFY_WRITE, usockvec, 2 * sizeof(int));
600         if(er)
601         {
602             sys_close(fd1);
603             sys_close(fd2);
604             return er;
605         }
606         put_fs_long(fd1, &usockvec[0]);
607         put_fs_long(fd2, &usockvec[1]);

608         return(0);
609 }
```

sock_socketpair 只用于 UNIX 域，用于在两个进程间通过套接字建立一个连接进行数据传送。该函数通过指针变量返回描述套接字两通信端的文件描述符。一般在父子进程间通信时用该种方式。而且这种通信方式有点类似管道通信方式。sock_socketpair 函数首先调用 sock_socket 分别在模拟通信的两端建立套接字，然后调用 sock1->ops->socketpair 指针指向的函数初始化相关变量，此后初始化套接字各自的 conn 字段指向对方并修改套接字状态为 SS_CONNECTED 表示完成连接的建立。函数最后返回分别表示通信两端的文件描述符。

```
610 /*
611  *  Bind a name to a socket. Nothing much to do here since it's
612  *  the protocol's responsibility to handle the local address.
613  *
614  *  We move the socket address to kernel space before we call
615  *  the protocol layer (having also checked the address is ok).
616  */

617 static int sock_bind(int fd, struct sockaddr *umyaddr, int addrlen)
618 {
619     struct socket *sock;
620     int i;
621     char address[MAX_SOCK_ADDR];
622     int err;

623     if (fd < 0 || fd >= NR_OPEN || current->files->fd[fd] == NULL)
624         return(-EBADF);

625     if (!(sock = sockfd_lookup(fd, NULL)))
626         return(-ENOTSOCK);

627     if((err=move_addr_to_kernel(umyaddr,addrlen,address))<0)
628         return err;

629     if ((i = sock->ops->bind(sock, (struct sockaddr *)address, addrlen)) < 0)
630     {
631         return(i);
632     }
633     return(0);
634 }
```

sock_bind 函数是系统调用 bind 函数的 BSD 层对应函数，用于绑定一个本地地址。参数 umyaddr 表示需要绑定的地址。sock_bind 函数实现首先通过文件描述符获取其对应的 socket 结构，然后调用 move_addr_to_kernel 将地址从用户缓冲区复制到内核缓冲区，最后调用 sock->ops->bind 指针指向的函数（如 inet_bind）完成具体的操作。

```
635 /*
```

```
636   *    Perform a listen. Basically, we allow the protocol to do anything
637   *    necessary for a listen, and if that works, we mark the socket as
638   *    ready for listening.
639   */

640  static int sock_listen(int fd, int backlog)
641  {
642          struct socket *sock;

643          if (fd < 0 || fd >= NR_OPEN || current->files->fd[fd] == NULL)
644                  return(-EBADF);
645          if (!(sock = sockfd_lookup(fd, NULL)))
646                  return(-ENOTSOCK);

647          if (sock->state != SS_UNCONNECTED)
648          {
649                  return(-EINVAL);
650          }

651          if (sock->ops && sock->ops->listen)
652                  sock->ops->listen(sock, backlog);
653          sock->flags |= SO_ACCEPTCON;
654          return(0);
655  }
```

sock_listen 函数是系统调用 listen 函数 BSD 层对应的函数。该函数用于服务器端为接收客户端连接作准备。该函数通过调用 sock->ops->listen 函数完成具体的操作，listen 函数的底层实现函数主要是几个标志字段的设置，如 sock->flags。这样在接下来的操作中会检查这些标志位的设置。

```
656  /*
657   *    For accept, we attempt to create a new socket, set up the link
658   *    with the client, wake up the client, then return the new
659   *    connected fd. We collect the address of the connector in kernel
660   *    space and move it to user at the very end. This is buggy because
661   *    we open the socket then return an error.
662   */

663  static int sock_accept(int fd, struct sockaddr *upeer_sockaddr, int *upeer_addrlen)
664  {
665          struct file *file;
666          struct socket *sock, *newsock;
667          int i;
668          char address[MAX_SOCK_ADDR];
```

```
669     int len;

670     if (fd < 0 || fd >= NR_OPEN || ((file = current->files->fd[fd]) == NULL))
671         return(-EBADF);
672     if (!(sock = sockfd_lookup(fd, &file)))
673         return(-ENOTSOCK);
674     if (sock->state != SS_UNCONNECTED)
675     {
676         return(-EINVAL);
677     }
678     if (!(sock->flags & SO_ACCEPTCON))
679     {
680         return(-EINVAL);
681     }

682     if (!(newsock = sock_alloc()))
683     {
684         printk("NET: sock_accept: no more sockets\n");
685         return(-ENOSR);    /* Was: EAGAIN, but we are out of system
686                               resources! */
687     }
688     newsock->type = sock->type;
689     newsock->ops = sock->ops;
690     if ((i = sock->ops->dup(newsock, sock)) < 0)
691     {
692         sock_release(newsock);
693         return(i);
694     }

695     i = newsock->ops->accept(sock, newsock, file->f_flags);
696     if ( i < 0)
697     {
698         sock_release(newsock);
699         return(i);
700     }

701     if ((fd = get_fd(SOCK_INODE(newsock))) < 0)
702     {
703         sock_release(newsock);
704         return(-EINVAL);
705     }

706     if (upeer_sockaddr)
707     {
```

```
708             newsock->ops->getname(newsock, (struct sockaddr *)address, &len, 1);
709             move_addr_to_user(address,len, upeer_sockaddr, upeer_addrlen);
710         }
711         return(fd);
712 }
```

sock_accept 函数是系统调用 accept 函数的 BSD 层对应函数，用于服务器端接受一个客户端的连接请求。从该函数实现可以看出我们经常讨论的监听套接字与实际处理数据传送的套接字是不同的。监听套接字在处理一个新的连接时会另外建立一个新的套接字结构专门用于此次数据传输，而监听套接字仍然进行监听。注意函数中对 SO_ACCEPTCON 标志位的检测，如果没有设置该标志位则表示对一个非监听套接字调用了该函数，此时错误返回。在确定是对一个监听套接字处理后，继续调用 sock_alloc 函数创建一个新的套接字用于数据传送，而后需要对此信的套接字进行初始化，初始化信息主要来自监听套接字中原有信息，sock->ops->dup 函数（如 inet_dup）即用于此目的。在对新创建的通信套接字进行初始化后，调用 sock->ops->accept 函数（如 inet_accept）完成与远端的连接建立过程。最后分配一个新文件描述符并返回用于以后的数据传送。函数最后，如果传入的 upeer_sockaddr 指针参数不为 NULL，表示应用程序需要返回通信远端的地址，此时调用 sock->ops->getname 函数从连接请求数据包中取得远端地址并使用 move_addr_to_user 函数将地址复制到用户缓冲区中。

```
713 /*
714  *  Attempt to connect to a socket with the server address.  The address
715  *  is in user space so we verify it is OK and move it to kernel space.
716  */

717 static int sock_connect(int fd, struct sockaddr *uservaddr, int addrlen)
718 {
719         struct socket *sock;
720         struct file *file;
721         int i;
722         char address[MAX_SOCK_ADDR];
723         int err;

724         if (fd < 0 || fd >= NR_OPEN || (file=current->files->fd[fd]) == NULL)
725                 return(-EBADF);
726         if (!(sock = sockfd_lookup(fd, &file)))
727                 return(-ENOTSOCK);

728         if((err=move_addr_to_kernel(uservaddr,addrlen,address))<0)
729                 return err;

730         switch(sock->state)
731         {
732                 case SS_UNCONNECTED:
733                         /* This is ok... continue with connect */
```

```
734                 break;
735             case SS_CONNECTED:
736                 /* Socket is already connected */
737                 if(sock->type == SOCK_DGRAM) /* Hack for now - move this all into the protocol */
738                     break;
739                 return -EISCONN;
740             case SS_CONNECTING:
741                 /* Not yet connected... we will check this. */

742                 /*
743                  *  FIXME:  for all protocols what happens if you start
744                  *  an async connect fork and both children connect. Clean
745                  *  this up in the protocols!
746                  */
747                 break;
748             default:
749                 return(-EINVAL);
750         }
751         i = sock->ops->connect(sock, (struct sockaddr *)address, addrlen, file->f_flags);
752         if (i < 0)
753         {
754             return(i);
755         }
756         return(0);
757 }
```

sock_connect 函数是系统调用 connect 函数 BSD 层的对应函数。该函数首先将要连接的远端地址从用户缓冲区复制到内核缓冲区，之后根据套接字目前所处的状态采取对应的措施，如对已经完成连接的套接字调用该函数，则简单返回 EISCONN。如果状态有效，则调用 sock->ops->connect 函数完成具体的连接，底层实现函数将涉及到网络数据的传输。

```
758 /*
759  *  Get the local address ('name') of a socket object. Move the obtained
760  *  name to user space.
761  */

762 static int sock_getsockname(int fd, struct sockaddr *usockaddr, int *usockaddr_len)
763 {
764         struct socket *sock;
765         char address[MAX_SOCK_ADDR];
766         int len;
767         int err;
```

```
768         if (fd < 0 || fd >= NR_OPEN || current->files->fd[fd] == NULL)
769                 return(-EBADF);
770         if (!(sock = sockfd_lookup(fd, NULL)))
771                 return(-ENOTSOCK);

772         err=sock->ops->getname(sock, (struct sockaddr *)address, &len, 0);
773         if(err)
774                 return err;
775         if((err=move_addr_to_user(address,len, usockaddr, usockaddr_len))<0)
776                 return err;
777         return 0;
778 }

779 /*
780  *    Get the remote address ('name') of a socket object. Move the obtained
781  *    name to user space.
782  */

783 static int sock_getpeername(int fd, struct sockaddr *usockaddr, int *usockaddr_len)
784 {
785         struct socket *sock;
786         char address[MAX_SOCK_ADDR];
787         int len;
788         int err;

789         if (fd < 0 || fd >= NR_OPEN || current->files->fd[fd] == NULL)
790                 return(-EBADF);
791         if (!(sock = sockfd_lookup(fd, NULL)))
792                 return(-ENOTSOCK);

793         err=sock->ops->getname(sock, (struct sockaddr *)address, &len, 1);
794         if(err)
795                 return err;
796         if((err=move_addr_to_user(address,len, usockaddr, usockaddr_len))<0)
797                 return err;
798         return 0;
799 }
```

sock_getsockname 和 sock_getpeername 用于获取通信双端本地和远端的地址。注意二者调用同一个下层函数，下层函数将根据其传入的第三个函数做出判断返回本地地址或者远端地址。地址的获取是相当简单的，因为对于每个套接字连接，下层都有一个 sock 结构对应，该结构中将保存通信双方的地址信息，所以返回地址信息只是相应字段的简单复制或移动。

```
800 /*
```

```
801   *    Send a datagram down a socket. The datagram as with write() is
802   *    in user space. We check it can be read.
803   */

804  static int sock_send(int fd, void * buff, int len, unsigned flags)
805  {
806          struct socket *sock;
807          struct file *file;
808          int err;

809          if (fd < 0 || fd >= NR_OPEN || ((file = current->files->fd[fd]) == NULL))
810                  return(-EBADF);
811          if (!(sock = sockfd_lookup(fd, NULL)))
812                  return(-ENOTSOCK);

813          if(len<0)
814                  return -EINVAL;
815          err=verify_area(VERIFY_READ, buff, len);
816          if(err)
817                  return err;
818          return(sock->ops->send(sock, buff, len, (file->f_flags & O_NONBLOCK), flags));
819  }

820  /*
821   *    Send a datagram to a given address. We move the address into kernel
822   *    space and check the user space data area is readable before invoking
823   *    the protocol.
824   */

825  static int sock_sendto(int fd, void * buff, int len, unsigned flags,
826                  struct sockaddr *addr, int addr_len)
827  {
828          struct socket *sock;
829          struct file *file;
830          char address[MAX_SOCK_ADDR];
831          int err;

832          if (fd < 0 || fd >= NR_OPEN || ((file = current->files->fd[fd]) == NULL))
833                  return(-EBADF);
834          if (!(sock = sockfd_lookup(fd, NULL)))
835                  return(-ENOTSOCK);

836          if(len<0)
837                  return -EINVAL;
```

```
838         err=verify_area(VERIFY_READ,buff,len);
839         if(err)
840                 return err;

841         if((err=move_addr_to_kernel(addr,addr_len,address))<0)
842                 return err;

843         return(sock->ops->sendto(sock, buff, len, (file->f_flags & O_NONBLOCK),
844                 flags, (struct sockaddr *)address, addr_len));
845 }
```

send 和 sendto 函数用于发送数据，不同的是 sendto 可以指定远端地址，对于 TCP 协议而言，指定的远端地址必须是之前与之建立连接的远端的地址。否则错误返回。而对于 UDP 协议则可以在每次发送数据时指定不同的远端地址。这两个函数的实现都是通过调用更底层函数完成具体的发送功能。对于 INET 域而言，下层函数分别为 inet_send, inet_sendto。这些函数将在 net/inet/af_inet.c 文件中分析。

```
846 /*
847  *  Receive a datagram from a socket. This isn't really right. The BSD manual
848  *  pages explicitly state that recv is recvfrom with a NULL to argument. The
849  *  Linux stack gets the right results for the wrong reason and this need to
850  *  be tidied in the inet layer and removed from here.
851  *  We check the buffer is writable and valid.
852  */

853 static int sock_recv(int fd, void * buff, int len, unsigned flags)
854 {
855         struct socket *sock;
856         struct file *file;
857         int err;

858         if (fd < 0 || fd >= NR_OPEN || ((file = current->files->fd[fd]) == NULL))
859                 return(-EBADF);

860         if (!(sock = sockfd_lookup(fd, NULL)))
861                 return(-ENOTSOCK);

862         if(len<0)
863                 return -EINVAL;
864         if(len==0)
865                 return 0;
866         err=verify_area(VERIFY_WRITE, buff, len);
867         if(err)
868                 return err;
```

```
869         return(sock->ops->recv(sock, buff, len,(file->f_flags & O_NONBLOCK), flags));
870 }

871 /*
872  *  Receive a frame from the socket and optionally record the address of the
873  *  sender. We verify the buffers are writable and if needed move the
874  *  sender address from kernel to user space.
875  */

876 static int sock_recvfrom(int fd, void * buff, int len, unsigned flags,
877                          struct sockaddr *addr, int *addr_len)
878 {
879         struct socket *sock;
880         struct file *file;
881         char address[MAX_SOCK_ADDR];
882         int err;
883         int alen;
884         if (fd < 0 || fd >= NR_OPEN || ((file = current->files->fd[fd]) == NULL))
885                 return(-EBADF);
886         if (!(sock = sockfd_lookup(fd, NULL)))
887                 return(-ENOTSOCK);
888         if(len<0)
889                 return -EINVAL;
890         if(len==0)
891                 return 0;

892         err=verify_area(VERIFY_WRITE,buff,len);
893         if(err)
894                 return err;

895         len=sock->ops->recvfrom(sock, buff, len, (file->f_flags & O_NONBLOCK),
896                                 flags, (struct sockaddr *)address, &alen);

897         if(len<0)
898                 return len;
899         if(addr!=NULL && (err=move_addr_to_user(address,alen, addr, addr_len))<0)
900                 return err;

901         return len;
902 }
```

sock_recv 和 sock_recvfrom 用于接收远端发送的数据，不同的是 sock_recvfrom 可以同时返回远端地址，所以 sock_recvfrom 函数主要用于 UDP 协议。从二者的实现来看，与之前其它函数的

实现无异：即通过调用下层函数实现。对于 INET 域而言，二者分别对应 inet_recv, inet_recvfrom。

```
903 /*
904  *  Set a socket option. Because we don't know the option lengths we have
905  *  to pass the user mode parameter for the protocols to sort out.
906  */

907 static int sock_setsockopt(int fd, int level, int optname, char *optval, int optlen)
908 {
909     struct socket *sock;
910     struct file *file;

911     if (fd < 0 || fd >= NR_OPEN || ((file = current->files->fd[fd]) == NULL))
912         return(-EBADF);
913     if (!(sock = sockfd_lookup(fd, NULL)))
914         return(-ENOTSOCK);

915     return(sock->ops->setsockopt(sock, level, optname, optval, optlen));
916 }

917 /*
918  *  Get a socket option. Because we don't know the option lengths we have
919  *  to pass a user mode parameter for the protocols to sort out.
920  */

921 static int sock_getsockopt(int fd, int level, int optname, char *optval, int *optlen)
922 {
923     struct socket *sock;
924     struct file *file;

925     if (fd < 0 || fd >= NR_OPEN || ((file = current->files->fd[fd]) == NULL))
926         return(-EBADF);
927     if (!(sock = sockfd_lookup(fd, NULL)))
928         return(-ENOTSOCK);

929     if (!sock->ops || !sock->ops->getsockopt)
930         return(0);
931     return(sock->ops->getsockopt(sock, level, optname, optval, optlen));
932 }
```

sock_setsockopt 和 sock_getsockopt 函数分别用于设置和获取套接字选项。根据选项类型，这个选项可以处在任何一个层次上。这两个函数的实现通过调用下层函数完成，对于 INET 域而言，分别对应 inet_setsockopt 和 inet_getsockopt。

```
933 /*
934  *    Shutdown a socket.
935  */

936 static int sock_shutdown(int fd, int how)
937 {
938         struct socket *sock;
939         struct file *file;

940         if (fd < 0 || fd >= NR_OPEN || ((file = current->files->fd[fd]) == NULL))
941                 return(-EBADF);
942         if (!(sock = sockfd_lookup(fd, NULL)))
943                 return(-ENOTSOCK);

944         return(sock->ops->shutdown(sock, how));
945 }
```

该函数是系统调用 shutdown 函数的 BSD 对应实现函数，用于套接字半关闭操作。半关闭操作在一些应用中非常常见，因为有时需要以本地半关闭的形式通知对方本地数据已完全传送完毕。半关闭即只关闭发送通道（接收通道不可单独关闭）。一般在完全发送完本地数据后，可以使用 shutdown 关闭本地发送通道。该函数实现通过调用下层函数 inet_shutdown 完成，inet_shutdown 调用 tcp_shutdown,这些实现均不涉及网络数据的传输，主要是相关标志位的设置，所以实现上均较为简单。

```
946 /*
947  *    Perform a file control on a socket file descriptor.
948  */

949 int sock_fcntl(struct file *filp, unsigned int cmd, unsigned long arg)
950 {
951         struct socket *sock;

952         sock = socki_lookup (filp->f_inode);
953         if (sock != NULL && sock->ops != NULL && sock->ops->fcntl != NULL)
954                 return(sock->ops->fcntl(sock, cmd, arg));
955         return(-EINVAL);
956 }
```

该函数用于控制套接字行为。如果对普通文件的控制操作一样，使用 fcntl 可以修改套接字对应结构（如 socket，sock 结构）的某些字段，这是最直接的控制套接字的方式。另外 sock_ioctl 函数也用于类似的目的。目前此函数只用于设置和获取套接字属主（sock 结构的 proc 字段值）。

```
957 /*
958  *    System call vectors. Since I (RIB) want to rewrite sockets as streams,
```

```
959  *    we have this level of indirection. Not a lot of overhead, since more of
960  *    the work is done via read/write/select directly.
961  *
962  *    I'm now expanding this up to a higher level to separate the assorted
963  *    kernel/user space manipulations and global assumptions from the protocol
964  *    layers proper - AC.
965  */

966  asmlinkage int sys_socketcall(int call, unsigned long *args)
967  {
968      int er;
969      switch(call)
970      {
971          case SYS_SOCKET:
972              er=verify_area(VERIFY_READ, args, 3 * sizeof(long));
973              if(er)
974                  return er;
975              return(sock_socket(get_fs_long(args+0),
976                  get_fs_long(args+1),
977                  get_fs_long(args+2)));
978          case SYS_BIND:
979              er=verify_area(VERIFY_READ, args, 3 * sizeof(long));
980              if(er)
981                  return er;
982              return(sock_bind(get_fs_long(args+0),
983                  (struct sockaddr *)get_fs_long(args+1),
984                  get_fs_long(args+2)));
985          case SYS_CONNECT:
986              er=verify_area(VERIFY_READ, args, 3 * sizeof(long));
987              if(er)
988                  return er;
989              return(sock_connect(get_fs_long(args+0),
990                  (struct sockaddr *)get_fs_long(args+1),
991                  get_fs_long(args+2)));
992          case SYS_LISTEN:
993              er=verify_area(VERIFY_READ, args, 2 * sizeof(long));
994              if(er)
995                  return er;
996              return(sock_listen(get_fs_long(args+0),
997                  get_fs_long(args+1)));
998          case SYS_ACCEPT:
999              er=verify_area(VERIFY_READ, args, 3 * sizeof(long));
1000                 if(er)
1001                     return er;
```

```
1002                return(sock_accept(get_fs_long(args+0),
1003                        (struct sockaddr *)get_fs_long(args+1),
1004                        (int *)get_fs_long(args+2)));
1005        case SYS_GETSOCKNAME:
1006                er=verify_area(VERIFY_READ, args, 3 * sizeof(long));
1007                if(er)
1008                        return er;
1009                return(sock_getsockname(get_fs_long(args+0),
1010                        (struct sockaddr *)get_fs_long(args+1),
1011                (int *)get_fs_long(args+2)));
1012        case SYS_GETPEERNAME:
1013                er=verify_area(VERIFY_READ, args, 3 * sizeof(long));
1014                if(er)
1015                        return er;
1016                return(sock_getpeername(get_fs_long(args+0),
1017                        (struct sockaddr *)get_fs_long(args+1),
1018                        (int *)get_fs_long(args+2)));
1019        case SYS_SOCKETPAIR:
1020                er=verify_area(VERIFY_READ, args, 4 * sizeof(long));
1021                if(er)
1022                        return er;
1023                return(sock_socketpair(get_fs_long(args+0),
1024                        get_fs_long(args+1),
1025                        get_fs_long(args+2),
1026                        (unsigned long *)get_fs_long(args+3)));
1027        case SYS_SEND:
1028                er=verify_area(VERIFY_READ, args, 4 * sizeof(unsigned long));
1029                if(er)
1030                        return er;
1031                return(sock_send(get_fs_long(args+0),
1032                        (void *)get_fs_long(args+1),
1033                        get_fs_long(args+2),
1034                        get_fs_long(args+3)));
1035        case SYS_SENDTO:
1036                er=verify_area(VERIFY_READ, args, 6 * sizeof(unsigned long));
1037                if(er)
1038                        return er;
1039                return(sock_sendto(get_fs_long(args+0),
1040                        (void *)get_fs_long(args+1),
1041                        get_fs_long(args+2),
1042                        get_fs_long(args+3),
1043                        (struct sockaddr *)get_fs_long(args+4),
1044                        get_fs_long(args+5)));
1045        case SYS_RECV:
```

```
1046                    er=verify_area(VERIFY_READ, args, 4 * sizeof(unsigned long));
1047                    if(er)
1048                            return er;
1049                    return(sock_recv(get_fs_long(args+0),
1050                            (void *)get_fs_long(args+1),
1051                            get_fs_long(args+2),
1052                            get_fs_long(args+3)));
1053            case SYS_RECVFROM:
1054                    er=verify_area(VERIFY_READ, args, 6 * sizeof(unsigned long));
1055                    if(er)
1056                            return er;
1057                    return(sock_recvfrom(get_fs_long(args+0),
1058                            (void *)get_fs_long(args+1),
1059                            get_fs_long(args+2),
1060                            get_fs_long(args+3),
1061                            (struct sockaddr *)get_fs_long(args+4),
1062                            (int *)get_fs_long(args+5)));
1063            case SYS_SHUTDOWN:
1064                    er=verify_area(VERIFY_READ, args, 2* sizeof(unsigned long));
1065                    if(er)
1066                            return er;
1067                    return(sock_shutdown(get_fs_long(args+0),
1068                            get_fs_long(args+1)));
1069            case SYS_SETSOCKOPT:
1070                    er=verify_area(VERIFY_READ, args, 5*sizeof(unsigned long));
1071                    if(er)
1072                            return er;
1073                    return(sock_setsockopt(get_fs_long(args+0),
1074                            get_fs_long(args+1),
1075                            get_fs_long(args+2),
1076                            (char *)get_fs_long(args+3),
1077                            get_fs_long(args+4)));
1078            case SYS_GETSOCKOPT:
1079                    er=verify_area(VERIFY_READ, args, 5*sizeof(unsigned long));
1080                    if(er)
1081                            return er;
1082                    return(sock_getsockopt(get_fs_long(args+0),
1083                            get_fs_long(args+1),
1084                            get_fs_long(args+2),
1085                            (char *)get_fs_long(args+3),
1086                            (int *)get_fs_long(args+4)));
1087            default:
1088                    return(-EINVAL);
1089    }
```

```
1090    }
```

本函数是网络栈专用操作函数集的总入口函数。该函数功能单一，主要是将请求分配，调用具体的底层函数进行处理。其中参数 call 表示具体的请求，args 为相应请求所需要提供的信息。这些信息是与应用程序接口函数中定义的参数一一对应的。

```
1091    /*
1092     *  This function is called by a protocol handler that wants to
1093     *  advertise its address family, and have it linked into the
1094     *  SOCKET module.
1095     */

1096    int sock_register(int family, struct proto_ops *ops)
1097    {
1098        int i;

1099        cli();
1100    for(i = 0; i < NPROTO; i++)
1101    {
        //如果 pops[i]==NULL，则表示对应表项闲置，即可用，否则继续检查下一个表项。
1102        if (pops[i] != NULL)
1103            continue;
        //找到一个闲置表项后，用输入的参数对其进行初始化即完成操作函数集的注册。
        //从下文 sock_unregister 函数的实现来看，操作函数集合的注销是通过 family 字段
        //匹配的。
1104        pops[i] = ops;
1105        pops[i]->family = family;
1106        sti();
1107        return(i);
1108    }
1109    sti();
1110    return(-ENOMEM);
1111}

1112/*
1113 *  This function is called by a protocol handler that wants to
1114 *  remove its address family, and have it unlinked from the
1115 *  SOCKET module.
1116 */

1117int sock_unregister(int family)
1118{
1119    int i;
```

```
1120    cli();
1121    for(i = 0; i < NPROTO; i++)
1122    {
        //如果是一个闲置表项，则显然不需要注销。
1123        if (pops[i] == NULL)
1124            continue;
        //发现一个非空闲表项，检查其 family 字段值，如果和传入的参数相等，则满足
        //条件，将该表项重新设置为空闲（即完成注销操作）。
1125        if(pops[i]->family == family)
1126        {
1127            pops[i]=NULL;
1128            sti();
1129            return(i);
1130        }
1131    }
1132    sti();
1133    return(-ENOENT);
1134}
```

sock_register 和 sock_unregister 用于注册和注销一个域操作集。sock_register 函数首先在 pops 数组中寻找到一个空闲元素，然后根据传入的参数初始化该元素即完成域操作集的注册。sock_unregister 函数完成相应操作函数集的注销，即设置对应表项重新为空闲状态即可。

```
1135void proto_init(void)
1136{
1137    extern struct net_proto protocols[];/* Network protocols */
1138    struct net_proto *pro;

1139    /* Kick all configured protocols. */
1140    pro = protocols;
1141    while (pro->name != NULL)
1142    {
1143        (*pro->init_func)(pro);
1144        pro++;
1145    }
1146    /* We're all done... */
1147}
```

proto_init 函数完成相关协议初始化操作，protocols 数组定义在 protocols.c 文件中，其中有 IPX，802.2，SNAP，AX.25 等协议的初始化函数，本函数即依次遍历该数据，调用每个初始化函数完成对应协议的初始化工作。

```
1148void sock_init(void)
1149{
```

```
1150      int i;

1151      printk("Swansea University Computer Society NET3.019\n");

1152      /*
1153       *    Initialize all address (protocol) families.
1154       */

          //对 pops 数组进行清空处理。为下面对该数组的初始化作准备。
1155      for (i = 0; i < NPROTO; ++i) pops[i] = NULL;

1156      /*
1157       *    Initialize the protocols module.
1158       */
1159      proto_init();

1160     #ifdef CONFIG_NET
1161      /*
1162       *    Initialize the DEV module.
1163       */

1164      dev_init();

1165      /*
1166       *    And the bottom half handler
1167       */

1168      bh_base[NET_BH].routine= net_bh;
1169      enable_bh(NET_BH);
1170      #endif
1171}
```

本函数在 start_kernel 函数中被调用，用于整个网络栈的初始化操作。该函数主要完成相关协议初始化（proto_init）,部分网卡驱动程序初始化（dev_init, net/inet/dev.c）,用于网络栈操作的下半部分（Bottom Half）初始化。下半部分的初始化主要是对 bh_base 数组 NET_BH 对应表项函数指针进行了赋值（net_bh 函数，net/inet/dev.c）以及使能该下半部分。有关下半部分的功能及相关结构的介绍请参考第一章中相关内容。

socket.c 文件最后定义了一个信息获取函数 socket_get_info。此处指的信息即系统当前使用的套接字总数目（sockets_in_use，这是定义在本文件头部一个变量，用于表示系统正在使用的套接字总数量）。

```
1172int socket_get_info(char *buffer, char **start, off_t offset, int length)
```

```
1173{
1174        int len = sprintf(buffer, "sockets: used %d\n", sockets_in_use);
1175        if (offset >= len)
1176        {
1177                *start = buffer;
1178                return 0;
1179        }
1180        *start = buffer + offset;
1181        len -= offset;
1182        if (len > length)
1183                len = length;
1184        return len;
1185}
```

socket.c 文件小结

socket.c 文件中定义的函数作为 BSD 层的函数集，对于 INET 正常网络操作而言，其下层即对应 inet/af_inet.c 文件中定义的函数。而且这种对应关系几乎是一一对应的。从对 socket.c 文件中函数实现来看，绝大部分函数并没有进行具体所请求功能的实现，而是将请求传给了下一层函数，这个下一层函数即指 af_inet.c 文件中定义的函数。下文将对该文件进行分析，不过从对 af_inet.c 文件的分析来看，其中定义的函数实际上也没有进行具体功能的实现，而是将请求继续传递给下一层（此时即传输层）。这是可以理解的，因为对于大部分操作，根据传输层所使用协议的不同，具体对应的操作也不同，所以请求不传递到这一层，上层函数无法进行具体的处理。而上层函数（socket.c, af_inet.c 文件中定义的函数均是作为上层函数而言）主要是对一些标志位进行检查后即调用下层函数进行处理。

2.3 net/inet/af_inet.c 文件

该文件定义的函数集作为网络栈中 INET 层而存在。之所以称为 INET 层，一是因为 af_inet.c 文件中几乎所有的函数均是以 inet 这四个字符打头；二是该文件中定义的函数作为 INET 域表示层的操作接口而存在。我们可以将该文件定义之函数集合作为 L5 层（也即表示层）的实现。以下将对该文件中定义函数一一进行分析。读者可以发现，作为正常的网络栈实现，该文件中定义函数是作为 socket.c 文件中定义函数的下层函数而存在的。如 socket.c 文件中 sock_bind 对应 af_inet.c 中 inet_bind, sock_accept 对应 inet_accept，sock_connect 对应 inet_connect 等等。

由于该文件函数在进一步调用下层函数时需要都需要使用到 sock 结构，故首先有必须对此结构进行以下说明。由于在 L5 层被大量使用，所以将 sock 结构作为一个套接字连接在 L5 层表示（socket 结构作为 L6 层对于一个套接字连接的表示），当然从实际网络栈的实现来看，sock 结构的使用范围相比 socket 结构要广泛的多，socket 结构主要使用在 L6 层，L5 层作为 L6 层下层，也会有部分地方使用到 socket 结构，但很少。但 sock 结构的使用基本贯穿 L2，L3，L4，L5 层，而且是作为各层的之间的一个联系。这主要是因为无论是发送的还是接收的数据包都要被缓存到 sock 结构中的缓冲队列中。在下面对 sock 结构的介绍中也可以看出这一点。由于 sock 结构相当庞大，故对其进行分段分析。

```
/*net/inet/sock.h*/
/*
```

```
 * This structure really needs to be cleaned up.
 * Most of it is for TCP, and not used by any of
 * the other protocols.
 */
struct sock {
  struct options     *opt; //IP 选项缓存于此处。
  volatile unsigned long wmem_alloc; //当前写缓冲区大小，该值不可大于系统规定的最大值。
  volatile unsigned long rmem_alloc; //当前读缓冲区大小，该值不可大于系统规定最大值。
  //以下三个字段用于 TCP 协议中为保证可靠数据传输而使用的序列号。
  //其中 write_seq 表示应用程序下一次写数据时所对应的第一个字节的序列号。
  //sent_seq 表示本地将要发送的下一个数据包中第一个字节对应的序列号。该字段对应//TCP
  首部中序列号字段，表示本数据包中所包含数据第一个字节的序列号。
  //acked_seq 表示本地希望从远端接收的下一个数据的序列号。该字段对应 TCP 首部中应//答
  序列号字段。
  //有关 TCP 协议为保证可靠性数据传输底层实现方法的论述请参考附录一，其中对序列号
  //的使用的阐述较为详细，读者可以借此理解 sock 结构中这几个字段的意义。
  unsigned long              write_seq;
  unsigned long              sent_seq;
  unsigned long              acked_seq;
  /* 应用程序有待读取（但尚未读取）数据的第一个序列号。*/
  unsigned long              copied_seq;
  //rcv_ack_seq 表示目前本地接收到的对本地发送数据的应答序列号。如 acked_seq=12345,
  //则表示本地之前发送的序列号小于 12345 的所有数据已被远端成功接收。
  unsigned long              rcv_ack_seq;
  //窗口大小，是一个绝对值，表示本地将要发送数据包中所包含最后一个数据的序列号不//可
  大于 window_seq. window_seq 的初始化为 sent_seq 加上远端当前同胞的窗口大小，这//个窗口
  大小是在 TCP 首部中窗口字段指定的。
  unsigned long              window_seq;
  //该字段在对方发送 FIN 数据包时使用，在接收到远端发送的 FIN 数据包后，fin_seq 被
  //初始化为对方的 FIN 数据包最后一个字节的序列号加 1，表示本地对对方此 FIN 数据包//进
  行应答的序列号。
  unsigned long              fin_seq;
  //以下两个字段用于紧急数据处理，urg_seq 表示紧急数据最大序列号。urg_data 是一个标//志
  位，当设置为 1 时，表示接收到紧急数据。
  unsigned long              urg_seq;
  unsigned long              urg_data;
  /*
   * Not all are volatile, but some are, so we
   * might as well say they all are.
   */
  volatile char inuse, //inuse=1 表示其它进程正在使用该 sock 结构，本进程需等待。
                       dead, //dead=1 表示该 sock 结构已处于释放状态。
                       urginline, //urginline=1 表示紧急数据将被当作普通数据处理。
                       intr,
```

```
                blog, //blog=1 表示对应套接字处于节制状态，此时接收的数据包均被丢弃
                done,
                reuse,
                keepopen,//keepopen=1 表示使用保活定时器
                linger,//linger=1 表示在关闭套接字时需要等待一段时间以确认其已关闭。
                delay_acks, //delay_acks=1 表示延迟应答，可一次对多个数据包进行应答
                destroy, //destroy=1 表示该 sock 结构等待销毁
                ack_timed,
                no_check,
                zapped,   /* In ax25 & ipx means not linked */
                broadcast,
                nonagle;  //noagle=1 表示不使用 NAGLE 算法，Nagle 算法指每次最多只有//
                          一个数据包未被应答，换句话说，
                          //在前一个发送的数据包被应答之前，
                          //不可再继续发送其它数据包。

//lingertime 字段表示等待关闭操作的时间，只有当 linger 标志位为 1 时，该字段才有意义。
unsigned long                   lingertime;
int                 proc; //该 sock 结构（即该套接字）所属的进程的进程号。
//以下三个字段用于 sock 的连接。
struct sock         *next;
struct sock         *prev; /* Doubly linked chain.. */
struct sock         *pair;

//send_head, send_tail 用于 TCP 协议重发队列。
struct sk_buff      * volatile send_head;
struct sk_buff      * volatile send_tail;
//back_log 为接收的数据包缓存队列。
struct sk_buff_head         back_log;
//partial 字段用于创建最大长度的待发送数据包。
struct sk_buff      *partial;
//partial_timer 定时器用于按时发送 partial 指针指向的数据包，以免缓存（等待）时间过长。
struct timer_list   partial_timer;
//重发次数
long                retransmits;
//write_queue 指向待发送数据包，其与 send_head,send_tail 队列的不同之处在于
//send_head,send_tail 队列中数据包均已经发送出去，但尚未接收到应答。而 write_queue
//中数据包尚未发送。
//receive-queue 为读队列，其不同于 back_log 队列之处在于 back_log 队列缓存从网络层传
//上来的数据包，在用户进行读取操作时，不可操作 back_log 队列，而是从 receive_queue
//队列中去数据包读取其中的数据，即数据包首先缓存在 back_log 队列中，然后从 back_log
//队列中移动到 receive_queue 队列中方可被应用程序读取。而并非所有 back_log 队列中缓
//存的数据包都可以成功的被移动到 receive_queue 队列中，如果此刻读缓存区太小，则当
//前从 back_log 队列中被取下的被处理的数据包将被直接丢弃，而不会被缓存到
```

```
//receive_queue 队列中。如果从应答的角度看，在 back_log 队列中的数据包由于有可能被
//丢弃，故尚未应答，而将一个数据包从 back_log 移动到 receive_queue 时，表示该数据包
//已被正式接收，即会发送对该数据包的应答给远端表示本地已经成功接收该数据包。
struct sk_buff_head     write_queue, receive_queue;

struct proto            *prot; //该字段十分重要，指向传输层处理函数集。
struct wait_queue       **sleep;
unsigned long           daddr; //sock 结构所代表套接字的远端地址。
unsigned long           saddr; //本地地址
unsigned short          max_unacked; //最大未处理请求连接数（应答数）
unsigned short          window; //远端窗口大小
unsigned short          bytes_rcv; //已接收字节总数
/* mss is min(mtu, max_window) */
//最大传输单元
unsigned short          mtu;        /* mss negotiated in the syn's */
//最大报文长度：MSS=MTU-IP 首部长度-TCP 首部长度
volatile unsigned short mss;        /* current eff. mss - can change */
//用户指定的 MSS 值
volatile unsigned short user_mss;   /* mss requested by user in ioctl */
//最大窗口大小和窗口大小钳制值
volatile unsigned short max_window;
unsigned long           window_clamp;
//本地端口号
unsigned short          num;
//以下三个字段用于拥塞算法
volatile unsigned short cong_window;
volatile unsigned short cong_count;
volatile unsigned short ssthresh;
//本地已发送出去但尚未得到应答的数据包数目
volatile unsigned short packets_out;
//本地关闭标志位，用于半关闭操作
volatile unsigned short shutdown;
//往返时间估计值
volatile unsigned long rtt; //RTTS
volatile unsigned long mdev; //mean deviation, 即 RTTD, 绝对偏差
//RTO 是用 RTT 和 mdev 用算法计算出的延迟时间值。
volatile unsigned long rto; //RTO
/* currently backoff isn't used, but I'm maintaining it in case
* we want to go back to a backoff formula that needs it
*/
volatile unsigned short backoff; //退避算法度量值
volatile short          err; //错误标志值
unsigned char           protocol; //传输层协议值
volatile unsigned char state; //套接字状态值，如 TCP_ESTABLISHED
```

```
  volatile unsigned char ack_backlog; //缓存的未应答数据包个数
  unsigned char          max_ack_backlog; //最大缓存的未应答数据包个数
  unsigned char          priority; //该套接字优先级，在硬件缓存发送数据包时使用
  unsigned char          debug;
  //这两个字段的初始化将下文中对 inet_create 函数的分析。
  unsigned short         rcvbuf; //最大接收缓冲区大小
  unsigned short         sndbuf; //最大发送缓冲区大小
  unsigned short         type; //类型值如 SOCK_STREAM
  //localroute=1 表示只使用本地路由，一般目的端在相同子网时使用。
  unsigned char          localroute;     /* Route locally only */
//此处暂时省略有关 IPX，AX.25,Appletalk 协议相关字段的声明。
    #ifdef CONFIG_IPX
    ……
    #endif
    #ifdef CONFIG_AX25
    …….
    #endif
    #ifdef CONFIG_ATALK
    ……
    #endif
/* IP 'private area' or will be eventually */
  //IP 首部 TTL 字段值，实际上表示路由器跳数。
  int                    ip_ttl;         /* TTL setting */
  //IP 首部 TOS 字段值，服务类型值。
  int                    ip_tos;         /* TOS */
  //缓存的 TCP 首部，在 TCP 协议中创建一个发送数据包时可以利用此字段快速创建 TCP
  //首部。
  struct tcphdr          dummy_th;
  //保活定时器，用于探测对方窗口大小，防止对方通报窗口大小的数据包丢弃，从而造成
  //本地发送通道被阻塞。
  struct timer_list      keepalive_timer;    /* TCP keepalive hack */
  //重发定时器，用于数据包超时重发。
  struct timer_list      retransmit_timer;   /* TCP retransmit timer */
  //延迟应答定时器，延迟应答可以减少应答数据包的个数，但不可无限延迟以免造成远端
  //重发，所以设置定时器定期发送应答数据包。
  struct timer_list      ack_timer;          /* TCP delayed ack timer */
  //该字段为标志位组合字段，用于表示下文中 timer 定时器超时的原因。
  int                    ip_xmit_timeout;    /* Why the timeout is running */
  //如下 4 个字段用于 IP 多播。
#ifdef CONFIG_IP_MULTICAST
  int                    ip_mc_ttl;              /* Multicasting TTL */
  int                    ip_mc_loop;             /* Loopback (not implemented yet) */
  char                   ip_mc_name[MAX_ADDR_LEN];      /* Multicast device name */
  struct ip_mc_socklist  *ip_mc_list;            /* Group array */
```

```
#endif
  //以下两个字段用于通用定时，timeout 表示定时时间值，ip_xmit_timeout 表示此次定时的
  //原因，timer 为定时器。
  /* This part is used for the timeout functions (timer.c). */
  int                timeout;   /* What are we waiting for? */
  struct timer_list  timer; /* This is the TIME_WAIT/receive timer when we are doing IP */
  //时间戳
  struct timeval     stamp;

  /* identd */
  //一个套接字在不同的层次上分别由 socket 结构和 sock 结构表示，该 socket 字段用于指向//
  其对应的 socket 结构以便对某些信息的快速获取。同理在 socket 结构中也有指向 sock
  //结构的指针。
  struct socket                *socket;
  //以下四个函数指针字段指向回调函数。这些字段的设置为自定义回调函数提供的很大的
  //灵活性，内核在发生某些时间时，会调用这些函数，如此可以实现自定义响应。目前这
  //种自定义响应还是完全有内核控制。
  /* Callbacks */
  void                 (*state_change)(struct sock *sk);
  void                 (*data_ready)(struct sock *sk,int bytes);
  void                 (*write_space)(struct sock *sk);
  void                 (*error_report)(struct sock *sk);
};
```

在介绍完 sock 结构之后，下面我们开始对 af_inet.c 文件分析，前文中已说明该文件中定义函数也是作为接口函数而存在的。其绝大部分函数实现均是调用传输层函数（sock 结构中 prot 字段指向的函数集）完成具体的功能，因为诚如前文所述，对于不同的传输层协议完成一个请求的操作方式是很不同的，所以必须将请求传递到传输层方可进行具体的处理，所以 af_inet.c 中定义的函数集作为表示层操作所做的工作也仅限于对某些字段进行检查后，向下层继续传递请求。故可以预见 af_inet.c 文件中大部分函数实现都较为简单，而诸如 tcp.c,udp.c 文件中定义的函数（传输层函数集合）都较为复杂庞大。

/*net/inet/af_inet.c*/

```
1     /*
2      * INET        An implementation of the TCP/IP protocol suite for the LINUX
3      *             operating system.    INET is implemented using the    BSD Socket
4      *             interface as the means of communication with the user level.
5      *
6      *             AF_INET protocol family socket handler.
7      *
8      * Version:    @(#)af_inet.c  (from sock.c) 1.0.17       06/02/93
9      *
10     * Authors:    Ross Biro, <bir7@leland.Stanford.Edu>
11     *             Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
12     *             Florian La Roche, <flla@stud.uni-sb.de>
```

```
13    *        Alan Cox, <A.Cox@swansea.ac.uk>
14    *
15    * Changes (see also sock.c)
16    *
17    *        A.N.Kuznetsov      :      Socket death error in accept().
18    *        John Richardson :   Fix non blocking error in connect()
19    *                            so sockets that fail to connect
20    *                            don't return -EINPROGRESS.
21    *        Alan Cox:           Asynchronous I/O support
22    *        Alan Cox:           Keep correct socket pointer on sock structures
23    *                            when accept() ed
24    *        Alan Cox:           Semantics of SO_LINGER aren't state moved
25    *                            to close when you look carefully. With
26    *                            this fixed and the accept bug fixed
27    *                            some RPC stuff seems happier.
28    *        Niibe Yutaka   :      4.4BSD style write async I/O
29    *        Alan Cox,
30    *        Tony Gale      :      Fixed reuse semantics.
31    *        Alan Cox:           bind() shouldn't abort existing but dead
32    *                            sockets. Stops FTP netin:.. I hope.
33    *        Alan Cox:           bind() works correctly for RAW sockets. Note
34    *                            that FreeBSD at least is broken in this respect
35    *                            so be careful with compatibility tests...
36    *
37    *        This program is free software; you can redistribute it and/or
38    *        modify it under the terms of the GNU General Public License
39    *        as published by the Free Software Foundation; either version
40    *        2 of the License, or (at your option) any later version.
41    */

42    #include <linux/config.h>
43    #include <linux/errno.h>
44    #include <linux/types.h>
45    #include <linux/socket.h>
46    #include <linux/in.h>
47    #include <linux/kernel.h>
48    #include <linux/major.h>
49    #include <linux/sched.h>
50    #include <linux/timer.h>
51    #include <linux/string.h>
52    #include <linux/sockios.h>
53    #include <linux/net.h>
54    #include <linux/fcntl.h>
55    #include <linux/mm.h>
```

```
56  #include <linux/interrupt.h>

57  #include <asm/segment.h>
58  #include <asm/system.h>

59  #include <linux/inet.h>
60  #include <linux/netdevice.h>
61  #include "ip.h"
62  #include "protocol.h"
63  #include "arp.h"
64  #include "rarp.h"
65  #include "route.h"
66  #include "tcp.h"
67  #include "udp.h"
68  #include <linux/skbuff.h>
69  #include "sock.h"
70  #include "raw.h"
71  #include "icmp.h"

72  #define min(a,b)    ((a)<(b)?(a):(b))

73  extern struct proto packet_prot;
```

以上是对文件中头文件的声明，另外是一个宏定义用于取两数中较小者。此后是对 packet_prot 变量的一个声明，注意 extern 关键字，该关键字表示 packet_prot 变量声明在其它文件中，此处只是引用该变量。

```
74  /*
75   *  See if a socket number is in use.
76   */

77  static int sk_inuse(struct proto *prot, int num)
78  {
79      struct sock *sk;

80      for(sk = prot->sock_array[num & (SOCK_ARRAY_SIZE -1 )];
81          sk != NULL;   sk=sk->next)
82      {
83          if (sk->num == num)
84              return(1);
85      }
86      return(0);
87  }
```

sk_inuse 函数用于检测一个端口号是否已被使用。注意到 proto 结构是表示传输层操作函数集的一个结构，该结构也是定义在 net/inet/sock.h 文件中。其完整定义如下所示。

```
/*net/inet/sock.h*/
187  struct proto {
188      struct sk_buff *    (*wmalloc)(struct sock *sk,
189                              unsigned long size, int force,
190                              int priority);
191      struct sk_buff *    (*rmalloc)(struct sock *sk,
192                              unsigned long size, int force,
193                              int priority);
194      void                (*wfree)(struct sock *sk, struct sk_buff *skb,
195                               unsigned long size);
196      void                (*rfree)(struct sock *sk, struct sk_buff *skb,
197                               unsigned long size);
198      unsigned long           (*rspace)(struct sock *sk);
199      unsigned long           (*wspace)(struct sock *sk);
200      void                (*close)(struct sock *sk, int timeout);
201      int                 (*read)(struct sock *sk, unsigned char *to,
202                          int len, int nonblock, unsigned flags);
203      int                 (*write)(struct sock *sk, unsigned char *to,
204                               int len, int nonblock, unsigned flags);
205      int                 (*sendto)(struct sock *sk,
206                              unsigned char *from, int len, int noblock,
207                              unsigned flags, struct sockaddr_in *usin,
208                              int addr_len);
209      int                 (*recvfrom)(struct sock *sk,
210                              unsigned char *from, int len, int noblock,
211                              unsigned flags, struct sockaddr_in *usin,
212                              int *addr_len);
213      int                 (*build_header)(struct sk_buff *skb,
214                              unsigned long saddr,
215                              unsigned long daddr,
216                              struct device **dev, int type,
217                              struct options *opt, int len, int tos, int ttl);
218      int                 (*connect)(struct sock *sk,
219                             struct sockaddr_in *usin, int addr_len);
220      struct sock *           (*accept) (struct sock *sk, int flags);
221      void                (*queue_xmit)(struct sock *sk,
222                                  struct device *dev, struct sk_buff *skb,
223                                  int free);
224      void                (*retransmit)(struct sock *sk, int all);
225      void                (*write_wakeup)(struct sock *sk);
226      void                (*read_wakeup)(struct sock *sk);
227      int                 (*rcv)(struct sk_buff *buff, struct device *dev,
```

```
228                         struct options *opt, unsigned long daddr,
229                         unsigned short len, unsigned long saddr,
230                         int redo, struct inet_protocol *protocol);
231     int             (*select)(struct sock *sk, int which,
232                       select_table *wait);
233     int             (*ioctl)(struct sock *sk, int cmd,
234                       unsigned long arg);
235     int             (*init)(struct sock *sk);
236     void            (*shutdown)(struct sock *sk, int how);
237     int             (*setsockopt)(struct sock *sk, int level, int optname,
238                       char *optval, int optlen);
239     int             (*getsockopt)(struct sock *sk, int level, int optname,
240                       char *optval, int *option);
241     unsigned short  max_header;
242     unsigned long       retransmits;
243     struct sock *       sock_array[SOCK_ARRAY_SIZE];
244     char            name[80];
245     int             inuse, highestinuse;
246 };
```

proto 结构声明用于传输层操作的一系列函数指针，以及几个辅助字段。对于每个传输层协议都有一个 proto 结构对应，如 TCP 协议的 tcp_proto， UDP 协议的 udp_proto。
这些 proto 结构变量都初始化为各自协议对应的具体操作。从 proto 结构的字段名称可以看出，很多函数作为上层函数调用的对象将完成具体的功能，如 connect 指针指向的函数将实际完成连接操作，而 socket.c 文件中 sock_connect, 到 af_inet.c 文件中的 inet_connect 最后都归结于对此处 connect 指针所指向函数的调用。如对于 TCP 协议该指针字段对应值为 tcp_connect, 则调用队列如下：
应用程序调用 connect 函数，connect 函数的内核处理函数为 sys_socket 入口函数，sys_socket 入口函数调用对应的 sock_connect 函数，sock_connect 调用 inet_connect, inet_connect 调用 tcp_connect, tcp_connect 根据 TCP 协议的规范完成具体的连接操作（即进行三路握手连接）。

从这个调用过程可以看出，直到 tcp_connect 函数才进行实际的处理。而之前的函数均进行相应层次的检查工作。当然有一点需要指出的是，上面的调用过程并非直接调用，如 sock_connect 并非直接调用 inet_connect 函数，而是通过函数指针，即 socket->ops->connect 指针指向的函数（此处 socket 代表一个 socket 结构变量），对于 INET 域而言，这个指针即指向 inet_connect 函数，而对于 UNIX 域，则指向 unix_connect。这种使用函数指针的方式可以有效地实现灵活性和进行层次之间的隔离。另外 inet_connect 函数对 tcp_connect 此类函数的调用也是通过函数指针的方式即：sock->proto->connect 指针指向的函数（同理，sock 表示一个 sock 结构变量），所以对于 UDP 协议，这个函数指针就指向了 udp_connect, 从而避免了硬编码方式调用函数，也实现了内核程序的灵活性。在下文的分析中，读者可以大量看到 af_inet.c 文件中函数使用像 sock->proto->connect 这样的调用方式将请求传递给传输层处理。

刚才说到，对于每个传输层协议都有一个 proto 结构对应，并且只有一个 proto 结构对应，如 TCP 协议对应 tcp_proto 变量，UDP 对应 udp_proto。从这种意义上说，tcp_proto, udp_proto 分别是

TCP 协议，UDP 协议各自的代表。所以在很多函数中直接传入一个 proto 结构表示对应使用的协议。

如果同时有多个套接字使用 TCP 协议，就会有多个 sock 结构使用 tcp_proto 变量中字段指向的函数。而内核对于这种情况的管理是通过在 proto 结构定义一个 sock_array 数组来实现的，所有使用该种协议（如 TCP）协议的套接字(对应的 sock 结构)均被插入到 sock_array 数组中元素指向的链表中。sock_array 含有 256 个元素，即同时可以存在 256 个链表。至于一个 sock 结构插入到那个元素指向的链表是由该套接字所使用的本地端口号进行索引的。由于只有 256 个链表，而端口号范围可以达到 65535，所以需要使用求模（%）操作。这样所有使用 TCP 协议的套接字所对应的 sock 结构均被插入到 tcp_proto 变量之 sock_array 数组元素所指向的链表中。

如此 sk_inuse 函数中 for 循环的作用也就很好理解了。用端口号寻址对应 proto 结构中 sock_array 所对应元素所指向的队列，然后遍历该队列中所有 sock 结构，检查 sock 结构 num 字段与参数 num 值是否相同，如果都不同，则表示参数 num 表示的端口号并未使用，可以分批给新的套接字使用。由此亦可看出，对于不同的协议使用相同的端口号不会引起任何问题。

```
88  /*
89   *  Pick a new socket number
90   */

91  unsigned short get_new_socknum(struct proto *prot, unsigned short base)
92  {
93      static int start=0;

94      /*
95       * Used to cycle through the port numbers so the
96       * chances of a confused connection drop.
97       */

98      int i, j;
99      int best = 0;
100     int size = 32767; /* a big num. */
101     struct sock *sk;

102     if (base == 0)
103         base = PROT_SOCK+1+(start % 1024);
104     if (base <= PROT_SOCK)
105     {
106         base += PROT_SOCK+(start % 1024);
107     }

108     /* Now look through the entire array and try to find an empty ptr. */
109     for(i=0; i < SOCK_ARRAY_SIZE; i++)
110     {
```

```
111             j = 0;
112             sk = prot->sock_array[(i+base+1) &(SOCK_ARRAY_SIZE -1)];
113             while(sk != NULL)
114             {
115                     sk = sk->next;
116                     j++;
117             }
118             if (j == 0)
119             {
120                     start =(i+1+start )%1024;
121                     return(i+base+1);
122             }
123             if (j < size)
124             {
125                     best = i;
126                     size = j;
127             }
128     }

129     /* Now make sure the one we want is not in use. */

130     while(sk_inuse(prot, base +best+1))
131     {
132             best += SOCK_ARRAY_SIZE;
133     }
134     return(best+base+1);
135 }
```

get_new_socknum 用于获取一个新的未使用端口号。参数 prot 表示所使用的协议，base 表示最小起始端口号。该函数实现的基本思想是：
1>首先检查 sock_array 数组中具有最少 sock 结构的链表所对应表项。
2>从<1>中结果计算出一个端口号。计算方法如下。
假设由步骤<1>获取的表项索引为 N（从 0 计数）。则表示分配的端口号 n 满足：
n% SOCK_ARRAY_SIZE=N
即 n=N+m* SOCK_ARRAY_SIZE，（m=0，1，2，……，1024<n<65535）
之所以对 n 取值限制在 1024 之上，是因为 1024 之下的端口号被保留，或者必须具有特权方能使用。

get_new_socknum 实现中第一个 for 循环完成步骤 1，此后的 while 循环完成步骤 2。
注意 for 循环次数为 SOCK_ARRAY_SIZE，即无论 for 内部第一个是对那个表项的检查，sock_array 所有表项都将被检查一遍。其中变量 j 表示当前被检查表项中 sock 结构的数目，注意 j=0，则表示该链表尚未使用，所以即可立即返回，因为毫无疑问可以使用该链表，此时端口的计算即取当前端口号。否则获取 j 值最小的表项，进行之后的第 2 步计算。

```
136 /*
137  *    Add a socket into the socket tables by number.
138  */
//将具有确定端口号一个新 sock 结构加入到 sock_array 数组表示的 sock 结构链表群中。
139 void put_sock(unsigned short num, struct sock *sk)
140 {
141     struct sock *sk1;
142     struct sock *sk2;
143     int mask;
144     unsigned long flags;

145     sk->num = num;
146     sk->next = NULL;
147     num = num &(SOCK_ARRAY_SIZE -1);

148     /* We can't have an interrupt re-enter here. */
149     save_flags(flags);
150     cli();

151     sk->prot->inuse += 1;
152     if (sk->prot->highestinuse < sk->prot->inuse)
153         sk->prot->highestinuse = sk->prot->inuse;

154     if (sk->prot->sock_array[num] == NULL)
155     {
156         sk->prot->sock_array[num] = sk;
157         restore_flags(flags);
158         return;
159     }
160     restore_flags(flags);
        //这个 for 语句用于估计本地地址子网反掩码。
161     for(mask = 0xff000000; mask != 0xffffffff; mask = (mask >> 8) | mask)
162     {
163         if ((mask & sk->saddr) &&
164             (mask & sk->saddr) != (mask & 0xffffffff))
165         {
166             mask = mask << 8;
167             break;
168         }
169     }
170     cli();
171     sk1 = sk->prot->sock_array[num];
172     for(sk2 = sk1; sk2 != NULL; sk2=sk2->next)
173     {
```

```
174             if (!(sk2->saddr & mask))
175             {
176                     if (sk2 == sk1)
177                     {
178                             sk->next = sk->prot->sock_array[num];
179                             sk->prot->sock_array[num] = sk;
180                             sti();
181                             return;
182                     }
183                     sk->next = sk2;
184                     sk1->next= sk;
185                     sti();
186                     return;
187             }
188             sk1 = sk2;
189     }

190     /* Goes at the end. */
191     sk->next = NULL;
192     sk1->next = sk;
193     sti();
194 }
```

put_sock 函数用于将一个新 sock 结构插入到对应的链表中。该链表有端口号在 sock->prot->sock_array 数组中进行寻址。实现中使用了本地地址掩码进行了地址排列。但实际上这种排列毫无必要！因为这不是路由表，这种排列方式并没有在其它地方得到利用。所以实际上该函数可以实现的相当简单，而现在弄得比较复杂。不过既然代码这样编写，此处还是有必要进行一下说明。函数中实现的插入顺序有效位（即非零位）从多到少排列的，如 10.16.1.23 排在 10.16.1.0 之前，10.16.1.0 排在 10.16.0.0 之前，依次类推。

```
195 /*
196  *  Remove a socket from the socket tables.
197  */

198 static void remove_sock(struct sock *sk1)
199 {
200     struct sock *sk2;
201     unsigned long flags;

202     if (!sk1->prot)
203     {
204             printk("sock.c: remove_sock: sk1->prot == NULL\n");
205             return;
206     }
```

```
207         /* We can't have this changing out from under us. */
208         save_flags(flags);
209         cli();
210         sk2 = sk1->prot->sock_array[sk1->num &(SOCK_ARRAY_SIZE -1)];
211         if (sk2 == sk1)
212         {
213                 sk1->prot->inuse -= 1;
214                 sk1->prot->sock_array[sk1->num &(SOCK_ARRAY_SIZE -1)] = sk1->next;
215                 restore_flags(flags);
216                 return;
217         }

218         while(sk2 && sk2->next != sk1)
219         {
220                 sk2 = sk2->next;
221         }

222         if (sk2)
223         {
224                 sk1->prot->inuse -= 1;
225                 sk2->next = sk1->next;
226                 restore_flags(flags);
227                 return;
228         }
229         restore_flags(flags);
230 }
```

remove_sock 函数用于从链表中删除一个指定端口号的 sock 结构。该函数实现较为简单，此处不再作说明。

如下 destroy_sock 函数较长，但实现的功能并不复杂，该函数用于销毁一个套接字连接，包括该套接字对应的 sock 结构。而在销毁之前需要对其中缓存的数据包进行释放操作以避免造成内存泄漏。

```
231 /*
232  *    Destroy an AF_INET socket
233  */

234 void destroy_sock(struct sock *sk)
235 {
236         struct sk_buff *skb;

237         sk->inuse = 1;                /* just to be safe. */
```

```
    //检查 dead 标志位，对于销毁的 sock 结构，其 dead 字段必须首先设置为 1。
238         /* In case it's sleeping somewhere. */
239         if (!sk->dead)
240                 sk->write_space(sk);
    //remove_sock 函数用于将 sock 结构从其对应链表中删除。
241         remove_sock(sk);

    //将 sock 结构中各定时器从相应链表中删除。
242         /* Now we can no longer get new packets. */
243         delete_timer(sk);
244         /* Nor send them */
245         del_timer(&sk->retransmit_timer);
    //释放 partial 指针指向的数据包。该 partial 指针用于创建大容量数据包（即使用最大
    //MTU 值创建的数据包，以便减少数据包的数量。此处的操作是释放该数据包。
246         while ((skb = tcp_dequeue_partial(sk)) != NULL) {
247                 IS_SKB(skb);
248                 kfree_skb(skb, FREE_WRITE);
249         }

    //清空写队列。
250         /* Cleanup up the write buffer. */
251         while((skb = skb_dequeue(&sk->write_queue)) != NULL) {
252                 IS_SKB(skb);
253                 kfree_skb(skb, FREE_WRITE);
254         }

255         /*
256           *    Don't discard received data until the user side kills its
257           *    half of the socket.
258           */

259         if (sk->dead)
260         {
        //清空已缓存的待读取数据包。
261                 while((skb=skb_dequeue(&sk->receive_queue))!=NULL)
262                 {
263                 /*
264                   * This will take care of closing sockets that were
265                   * listening and didn't accept everything.
266                   */
267                         if (skb->sk != NULL && skb->sk != sk)
268                         {
269                                 IS_SKB(skb);
```

```
270                     skb->sk->dead = 1;
271                     skb->sk->prot->close(skb->sk, 0);
272                 }
273                 IS_SKB(skb);
274                 kfree_skb(skb, FREE_READ);
275             }
276         }

277         /* Now we need to clean up the send head. */
278         cli();
    //清空重发队列中缓存的数据包。
279         for(skb = sk->send_head; skb != NULL; )
280         {
281             struct sk_buff *skb2;

282             /*
283               * We need to remove skb from the transmit queue,
284               * or maybe the arp queue.
285               */
286             if (skb->next    && skb->prev) {
287 /*                  printk("destroy_sock: unlinked skb\n");*/
288                 IS_SKB(skb);
289                 skb_unlink(skb);
290             }
291             skb->dev = NULL;
292             skb2 = skb->link3;
293             kfree_skb(skb, FREE_WRITE);
294             skb = skb2;
295         }
296         sk->send_head = NULL;
297         sti();

    //清空数据包接收缓存队列。
298         /* And now the backlog. */
299         while((skb=skb_dequeue(&sk->back_log))!=NULL)
300         {
301             /* this should never happen. */
302 /*          printk("cleaning back_log\n");*/
303             kfree_skb(skb, FREE_READ);
304         }

305         /* Now if it has a half accepted/ closed socket. */
306         if (sk->pair)
307         {
```

```
308             sk->pair->dead = 1;
309             sk->pair->prot->close(sk->pair, 0);
310             sk->pair = NULL;
311         }

312         /*
313           * Now if everything is gone we can free the socket
314           * structure, otherwise we need to keep it around until
315           * everything is gone.
316           */
        //如果 sock 结构 dead 标志位为 1，且其读写缓冲区均已被释放，则可以对该 sock 结
        //构本身进行释放操作。
317           if (sk->dead && sk->rmem_alloc == 0 && sk->wmem_alloc == 0)
318           {
319             kfree_s((void *)sk,sizeof(*sk));
320           }
        //否则设置定时器，等待其它进程释放其读写缓冲区后进行 sock 结构的释放。
321           else
322           {
323             /* this should never happen. */
324             /* actually it can if an ack has just been sent. */
325             sk->destroy = 1;
326             sk->ack_backlog = 0;
327             sk->inuse = 0;
328             reset_timer(sk, TIME_DESTROY, SOCK_DESTROY_TIME);
329         }
330 }
```

destroy_sock 函数虽然较长，但功能单一，从上文中对函数的注释可以看出，该函数主要完成对各个队列中缓存数据包的释放操作。并在读写缓冲区均已释放的条件下释放 sock 结构本身，释放 sock 结构后，这个套接字完全消失。

```
331 /*
332  *    The routines beyond this point handle the behaviour of an AF_INET
333  *    socket object. Mostly it punts to the subprotocols of IP to do
334  *    the work.
335  */

336 static int inet_fcntl(struct socket *sock, unsigned int cmd, unsigned long arg)
337 {
338         struct sock *sk;

339         sk = (struct sock *) sock->data;

340         switch(cmd)
```

```
341     {
342         case F_SETOWN:
343             /*
344              * This is a little restrictive, but it's the only
345              * way to make sure that you can't send a sigurg to
346              * another process.
347              */
348             if (!suser() && current->pgrp != -arg &&
349                 current->pid != arg) return(-EPERM);
350             sk->proc = arg;
351             return(0);
352         case F_GETOWN:
353             return(sk->proc);
354         default:
355             return(-EINVAL);
356     }
357 }
```

inet_fcntl 函数用于设置和获取套接字的有关信息。从该函数实现来看，目前仅提供设置和读取套接字属主的功能，即通过改变 sock 结构中 proc 字段值来完成。从底层实现来看，对于此类信息设置或获取函数而言，在实现上都是非常直接的，即通过修改相关结构的字段值来完成这些功能。下文中也有对于一些选项的设置和读取，无论函数本身有多长，但都比较简单。注意 inet_fcntl 函数实现中对于字段值的改变需要超级用户权限，这一点对于所有关键信息的设置都是一样的。

```
358 /*
359  *  Set socket options on an inet socket.
360  */

361 static int inet_setsockopt(struct socket *sock, int level, int optname,
362                 char *optval, int optlen)
363 {
364     struct sock *sk = (struct sock *) sock->data;
365     if (level == SOL_SOCKET)
366         return sock_setsockopt(sk,level,optname,optval,optlen);
367     if (sk->prot->setsockopt==NULL)
368         return(-EOPNOTSUPP);
369     else
370         return sk->prot->setsockopt(sk,level,optname,optval,optlen);
371 }

372 /*
373  *  Get a socket option on an AF_INET socket.
374  */
```

```
375  static int inet_getsockopt(struct socket *sock, int level, int optname,
376                     char *optval, int *optlen)
377  {
378        struct sock *sk = (struct sock *) sock->data;
379        if (level == SOL_SOCKET)
380              return sock_getsockopt(sk,level,optname,optval,optlen);
381        if(sk->prot->getsockopt==NULL)
382              return(-EOPNOTSUPP);
383        else
384              return sk->prot->getsockopt(sk,level,optname,optval,optlen);
385  }
```

inet_setsockopt 和 inet_getsockopt 用于选项的设置和读取。所谓选项简单的说就是套接字对应结构中（sock 结构）中某些字段。这两个函数实现根据选项的层次将分别调用不同的下层处理函数。目前只进行两个层次的区分：SOL_SOCKET 和其它层次。分别调用 sock_xetsockopt 和 sk->prot->xetsockopt 函数（其中 x 表示 s 或者 g，下同）。对于 TCP 协议而言，sock 结构 prot 字段将初始化为 tcp_proto，UDP 协议初始化为 udp_proto。所以对应的函数为 tcp_xetsockopt 和 udp_xegsockopt。sock_xetsockopt 定义在 net/inet/sock.c 文件中。

```
386  /*
387   *    Automatically bind an unbound socket.
388   */

389  static int inet_autobind(struct sock *sk)
390  {
391        /* We may need to bind the socket. */
392        if (sk->num == 0)
393        {
394              sk->num = get_new_socknum(sk->prot, 0);
395              if (sk->num == 0)
396                    return(-EAGAIN);
397              put_sock(sk->num, sk);
398              sk->dummy_th.source = ntohs(sk->num);
399        }
400        return 0;
401  }
```

inet_autobind 函数用于自动绑定一个本地端口号。例如应用程序客户端在使用 connect 函数之前没有使用 bind 函数进行绑定，则 connect 函数的底层实现将需要邦定一个本地端口后方可与对方建立连接。注意此处的邦定主要是分配一个用于连接的本地端口号。此即我们通常所说的自动邦定。get_new_socknum 函数用于分配一个未使用端口号，put_sock 函数将根据端口将 sock 结构插入的相应的队列中，而 sock 结构 dummy_th 字段是一个 tcphdr 结构（即 TCP 首部结构），其 source 字段表示本地端口号（注意该字段使用网络字节序）。

```
402 /*
403  *    Move a socket into listening state.
404  */

405 static int inet_listen(struct socket *sock, int backlog)
406 {
407         struct sock *sk = (struct sock *) sock->data;

408         if(inet_autobind(sk)!=0)
409                 return -EAGAIN;

410         /* We might as well re use these. */
411         /*
412           * note that the backlog is "unsigned char", so truncate it
413           * somewhere. We might as well truncate it to what everybody
414           * else does..
415           */
416         if (backlog > 5)
417                 backlog = 5;
418         sk->max_ack_backlog = backlog;
419         if (sk->state != TCP_LISTEN)
420         {
421                 sk->ack_backlog = 0;
422                 sk->state = TCP_LISTEN;
423         }
424         return(0);
425 }
```

inet_listen 函数不言而喻是对应用程序调用 listen 函数的 L5 层处理。注意 inet_listen 并没有继续调用更底层的函数，即 listen 函数实际上在 L5 层已完成处理。这些处理主要是对 sock 结构中 state 字段的设置。另外需要注意的是从 socket 结构获得其对应 sock 结构的方法：socket 结构的 data 字段用于指向其对应的 sock 结构。如此自然的在 L5，L6 层之间建立了联系。此处内核限制最大连接数为 5。

```
426 /*
427  *    Default callbacks for user INET sockets. These just wake up
428  *    the user owning the socket.
429  */

430 static void def_callback1(struct sock *sk)
431 {
432         if(!sk->dead)
433                 wake_up_interruptible(sk->sleep);
```

```
434 }

435 static void def_callback2(struct sock *sk,int len)
436 {
437         if(!sk->dead)
438         {
439                 wake_up_interruptible(sk->sleep);
440                 sock_wake_async(sk->socket, 1);
441         }
442 }

443 static void def_callback3(struct sock *sk)
444 {
445         if(!sk->dead)
446         {
447                 wake_up_interruptible(sk->sleep);
448                 sock_wake_async(sk->socket, 2);
449         }
450 }
```

以上是对三个回调函数的实现，目前实现的功能只是唤醒相关的进程，并未进行更复杂的响应。这些函数将作为 sock 结构中 state_change，data_ready，write_space，error_report 字段的初始化值。

```
451 /*
452  *    Create an inet socket.
453  *
454  *    FIXME: Gcc would generate much better code if we set the parameters
455  *    up in in-memory structure order. Gcc68K even more so
456  */

457 static int inet_create(struct socket *sock, int protocol)
458 {
459         struct sock *sk;
460         struct proto *prot;
461         int err;

462         sk = (struct sock *) kmalloc(sizeof(*sk), GFP_KERNEL);
463         if (sk == NULL)
464                 return(-ENOBUFS);
465         sk->num = 0;
466         sk->reuse = 0;
      //根据套接字类型进行相关字段的赋值。
467         switch(sock->type)
```

```
468     {
        //流式套接字，使用 TCP 协议操作函数集。
469         case SOCK_STREAM:
470         case SOCK_SEQPACKET:
            //在 socket 系统调用时，我们一般将 protocol 参数设置为 0。如果设置为非 0，
            //则对于不同的类型，必须赋予正确值，否则可能在此处处理时出现问题。
471             if (protocol && protocol != IPPROTO_TCP)
472             {
473                 kfree_s((void *)sk, sizeof(*sk));
474                 return(-EPROTONOSUPPORT);
475             }
476             protocol = IPPROTO_TCP;
            //TCP_NO_CHECK 定义为 1，表示对于 TCP 协议默认使用校验。
477         sk->no_check = TCP_NO_CHECK;
            //注意此处 prot 变量被初始化为 tcp_prot，稍后 sock 结构的 prot 字段将被初始
            //化为 prot 变量值。
478             prot = &tcp_prot;
479             break;
        //报文套接字，即使用 UDP 协议操作函数集。
480         case SOCK_DGRAM:
481             if (protocol && protocol != IPPROTO_UDP)
482             {
483                 kfree_s((void *)sk, sizeof(*sk));
484                 return(-EPROTONOSUPPORT);
485             }
486             protocol = IPPROTO_UDP;
487             sk->no_check = UDP_NO_CHECK;
488             prot=&udp_prot;
489             break;
        //原始套接字类型，注意此时 protocol 参数表示端口号。
        //原始套接字使用 raw_prot 变量指定的函数集。
490         case SOCK_RAW:
491             if (!suser())
492             {
493                 kfree_s((void *)sk, sizeof(*sk));
494                 return(-EPERM);
495             }
496             if (!protocol)
497             {
498                 kfree_s((void *)sk, sizeof(*sk));
499                 return(-EPROTONOSUPPORT);
500             }
501             prot = &raw_prot;
502             sk->reuse = 1;
```

```
503                 sk->no_check = 0;  /*
504                                     * Doesn't matter no checksum is
505                                     * performed anyway.
506                                     */
507                 sk->num = protocol;
508                 break;
        //包套接字，该套接字在网络层直接进行数据包的收发。此时使用由 packet_prot 变
        //量指定的函数集。注意此时 protocol 参数表示套接字端口号。
509             case SOCK_PACKET:
510                 if (!suser())
511                 {
512                     kfree_s((void *)sk, sizeof(*sk));
513                     return(-EPERM);
514                 }
515                 if (!protocol)
516                 {
517                     kfree_s((void *)sk, sizeof(*sk));
518                     return(-EPROTONOSUPPORT);
519                 }
520                 prot = &packet_prot;
521                 sk->reuse = 1;
522                 sk->no_check = 0;  /* Doesn't matter no checksum is
523                                     * performed anyway.
524                                     */
525                 sk->num = protocol;
526                 break;
        //不符合以上任何类型，则出错返回。
527             default:
528                 kfree_s((void *)sk, sizeof(*sk));
529                 return(-ESOCKTNOSUPPORT);
530         }
```

inet_create 函数被上层 sock_socket 函数调用（通过 socket->ops->create 函数指针），用于创建一个套接字对应的 sock 结构并对其进行初始化。该函数首先分配一个 sock 结构，然后根据套接字类型对 sock 相关字段进行初始化。具体分析参见函数中注释。注意其中对 prot 变量的赋值非常重要，该变量所指向的操作函数集将处理该套接字之后所有的实际请求。

```
531         sk->socket = sock;
```

建立与其对应的 socket 结构之间的关系，socket 结构先于 sock 结构建立。

如果定义了 Nagle 算法，则将 sock 结构的 nonagle 字段初始化为 0，否则为 1。

```
532 #ifdef CONFIG_TCP_NAGLE_OFF
533         sk->nonagle = 1;
534 #else
535         sk->nonagle = 0;
```

```
536 #endif
537         sk->type = sock->type; //初始化 sock 结构 type 字段：套接字类型
538         sk->stamp.tv_sec=0;
539         sk->protocol = protocol;    //传输层协议
540         sk->wmem_alloc = 0;
541         sk->rmem_alloc = 0;
542         sk->sndbuf = SK_WMEM_MAX; //最大发送缓冲区大小
543         sk->rcvbuf = SK_RMEM_MAX;    //最大接收缓冲区大小
    //以下是对 sock 结构字段的初始化，这些字段的意义参见前文中对 sock 结构的介绍。
544         sk->pair = NULL;
545         sk->opt = NULL;
546         sk->write_seq = 0;
547         sk->acked_seq = 0;
548         sk->copied_seq = 0;
549         sk->fin_seq = 0;
550         sk->urg_seq = 0;
551         sk->urg_data = 0;
552         sk->proc = 0;
553         sk->rtt = 0;                        /*TCP_WRITE_TIME << 3;*/
554         sk->rto = TCP_TIMEOUT_INIT;         /*TCP_WRITE_TIME*/
555         sk->mdev = 0;
556         sk->backoff = 0;
557         sk->packets_out = 0;
    //cong_window 设置为 1，即 TCP 首先进入慢启动阶段。这是 TCP 协议处理拥塞的
    //一种策略。
558         sk->cong_window = 1; /* start with only sending one packet at a time. */
559         sk->cong_count = 0;
560         sk->ssthresh = 0;
561         sk->max_window = 0;
562         sk->urginline = 0;
563         sk->intr = 0;
564         sk->linger = 0;
565         sk->destroy = 0;
566         sk->priority = 1;
567         sk->shutdown = 0;
568         sk->keepopen = 0;
569         sk->zapped = 0;
570         sk->done = 0;
571         sk->ack_backlog = 0;
572         sk->window = 0;
573         sk->bytes_rcv = 0;
574         sk->state = TCP_CLOSE;    //由于尚未进行连接，状态设置为 CLOSE。
575         sk->dead = 0;
576         sk->ack_timed = 0;
```

```
577         sk->partial = NULL;
578         sk->user_mss = 0;
579         sk->debug = 0;
            //设置最大可暂缓应答的字节数。
580         /* this is how many unacked bytes we will accept for this socket.    */
581         sk->max_unacked = 2048; /* needs to be at most 2 full packets. */
582         /* how many packets we should send before forcing an ack.
583              if this is set to zero it is the same as sk->delay_acks = 0 */
584         sk->max_ack_backlog = 0;
585         sk->inuse = 0;
586         sk->delay_acks = 0;
587         skb_queue_head_init(&sk->write_queue);
588         skb_queue_head_init(&sk->receive_queue);
589         sk->mtu = 576; //MTU 设置为保守的576字节，该大小在绝大多数连接中不会造成分片。

      //注意对于 prot 字段的初始化，prot 变量根据套接字类型已在函数前部分初始化为正确//的
      值，此处用于对 sock 结构的 prot 字段进行初始化。此后该套接字所有实质性的操作
      //均有该变量所指向的操作函数集完成。
590         sk->prot = prot;
591         sk->sleep = sock->wait;
592         sk->daddr = 0;
593         sk->saddr = 0 /* ip_my_addr() */;
594         sk->err = 0;
595         sk->next = NULL;
596         sk->pair = NULL;
597         sk->send_tail = NULL;
598         sk->send_head = NULL;
599         sk->timeout = 0;
600         sk->broadcast = 0;
601         sk->localroute = 0;
602         init_timer(&sk->timer);
603         init_timer(&sk->retransmit_timer);
604         sk->timer.data = (unsigned long)sk;
605         sk->timer.function = &net_timer;
606         skb_queue_head_init(&sk->back_log);
607         sk->blog = 0;
608         sock->data =(void *) sk;
      //sock 结构之 dummy_th 字段是 tcphdr 结构，该结构与 TCP 首部各字段对应，
      //故对以下的代码理解可结合 TCP 首部格式。TCP 首部格式参见第一章的有关介绍。
609         sk->dummy_th.doff = sizeof(sk->dummy_th)/4;
610         sk->dummy_th.res1=0;
611         sk->dummy_th.res2=0;
612         sk->dummy_th.urg_ptr = 0;
613         sk->dummy_th.fin = 0;
```

```
614         sk->dummy_th.syn = 0;
615         sk->dummy_th.rst = 0;
616         sk->dummy_th.psh = 0;
617         sk->dummy_th.ack = 0;
618         sk->dummy_th.urg = 0;
619         sk->dummy_th.dest = 0;
620         sk->ip_tos=0;
621         sk->ip_ttl=64;
622 #ifdef CONFIG_IP_MULTICAST
623         sk->ip_mc_loop=1;
624         sk->ip_mc_ttl=1;
625         *sk->ip_mc_name=0;
626         sk->ip_mc_list=NULL;
627 #endif
    //对 sock 结构中几个回调函数字段的初始化。
628         sk->state_change = def_callback1;
629         sk->data_ready = def_callback2;
630         sk->write_space = def_callback3;
631         sk->error_report = def_callback1;
    //如果该套接字已经分配本地端口号，则对 sock 结构中 dummy_th 结构字段进行赋值。
632         if (sk->num)
633         {
634         /*
635           * It assumes that any protocol which allows
636           * the user to assign a number at socket
637           * creation time automatically
638           * shares.
639           */
640                 put_sock(sk->num, sk);
641                 sk->dummy_th.source = ntohs(sk->num);
642         }

    //对于不同的操作函数集，可能需要进行某些初始化操作，此处调用其对应的初始化函
    //数完成相关操作，如对于 packet_prot 操作函数集，需要建立一个 packet_type 结构插
    //入到内核变量 ptype_base 指向的队列中，以完成在网络层收发数据包的功能。这个
    //packet_type 结构的建立即是在其初始化函数中完成。注意如果需要进行初始化，而初
    //始化失败，则直接销毁该套接字，因为其它部分尚未准备好，即便建立了该套接字，
    //也将无法使用。
643         if (sk->prot->init)
644         {
645                 err = sk->prot->init(sk);
646                 if (err != 0)
647                 {
648                         destroy_sock(sk);
```

```
649                     return(err);
650             }
651     }
652     return(0);
653 }
```

inet_create 函数较长，但实现的功能只有一个：建立套接字对应的 sock 结构并对其进行初始化，由于 sock 结构较为庞大，造成初始化代码较长。结合上文中对相关代码的注视，应不难理解。

```
654 /*
655  *  Duplicate a socket.
656  */

657 static int inet_dup(struct socket *newsock, struct socket *oldsock)
658 {
659     return(inet_create(newsock,((struct sock *)(oldsock->data))->protocol));
660 }
```

inet_dup 函数被服务器端调用。服务器端用于实际通信的套接字与其监听套接字是不同的。当应用程序在服务器端使用 accept 函数接收一个客户端请求时，会创建一个新的套接字用于与客户端进行具体的数据通信，而监听套接字仍然只负责监听其它客户端的请求。在 sock_accept(accept 系统调用的 BSD 层处理函数)在分配一个新的 socket 结构后，调用 inet_dup 函数对此新创建的 socket 结构进行初始化，而初始化信息主要来自监听套接字，inet_dup 函数的输入参数中 newsock 即表示新创建的 socket 结构，而 oldsock 则表示监听套接字 socket 结构。从 inet_dup 实现来看，该函数调用 inet_create 函数完成具体的功能：分配新 socket 结构对应的下层 sock 结构，此处只使用了监听 socket 结构的 protocol 字段。不过在 sock_accept 函数中这个新创建的 socket 结构之 ops，type 字段已经被初始化为监听 socket 结构中相应字段值。而这对于 inet_create 函数已经足够，因为注意到 inet_create 函数需要根据 type，protocol 字段值进行 sock 结构关键字段的赋值。读者可将 sock_accept, inet_create 函数结合来看。

```
661 /*
662  * Return 1 if we still have things to send in our buffers.
663  */
664 static inline int closing(struct sock * sk)
665 {
666     switch (sk->state) {
667         case TCP_FIN_WAIT1:
668         case TCP_CLOSING:
669         case TCP_LAST_ACK:
670             return 1;
671     }
672     return 0;
673 }
```

closing 函数实际上是对套接字状态的检测。当 TCP 连接状态为 FIN_WAIT1,CLOSING, LAST_ACK 时，即表示本地套接字已处于关闭状态。实际上还可以加上 FIN_WAIT2 状态。有关 TCP 协议状态的介绍，请参考第一章中内容。

```
674 /*
675  *    The peer socket should always be NULL (or else). When we call this
676  *    function we are destroying the object and from then on nobody
677  *    should refer to it.
678  */

679 static int inet_release(struct socket *sock, struct socket *peer)
680 {
681         struct sock *sk = (struct sock *) sock->data;
682         if (sk == NULL)
683                 return(0);

684         sk->state_change(sk);

685         /* Start closing the connection.    This may take a while. */

686 #ifdef CONFIG_IP_MULTICAST
687         /* Applications forget to leave groups before exiting */
688         ip_mc_drop_socket(sk);
689 #endif
690         /*
691           * If linger is set, we don't return until the close
692           * is complete.    Other wise we return immediately. The
693           * actually closing is done the same either way.
694           *
695           * If the close is due to the process exiting, we never
696           * linger..
697           */

698         if (sk->linger == 0 || (current->flags & PF_EXITING))
699         {
700                 sk->prot->close(sk,0);
701                 sk->dead = 1;
702         }
703         else
704         {
705                 sk->prot->close(sk, 0);
706                 cli();
707                 if (sk->lingertime)
708                         current->timeout = jiffies + HZ*sk->lingertime;
709                 while(closing(sk) && current->timeout>0)
```

```
710             {
711                     interruptible_sleep_on(sk->sleep);
712                     if (current->signal & ~current->blocked)
713                     {
714                             break;
715 #if 0
716                             /* not working now - closes can't be restarted */
717                             sti();
718                             current->timeout=0;
719                             return(-ERESTARTSYS);
720 #endif
721                     }
722             }
723             current->timeout=0;
724             sti();
725             sk->dead = 1;
726         }
727         sk->inuse = 1;

728         /* This will destroy it. */
729         release_sock(sk);
730         sock->data = NULL;
731         sk->socket = NULL;
732         return(0);
733 }
```

inet_release 函数是 sock_release 调用的 INET 层函数。

该函数将进一步完成套接字的关闭操作。首先通过 sock 结构中 state_change 函数指针通知相关进程套接字状态的变化。之后如果该套接字之前加入了一个多播组，则在关闭之前需要首先退出该多播组。然后根据 sock 结构中 linger 标志位进行相应的具体关闭操作。linger 标志位表示是否等待关闭操作的完成，如果 linger 标志位设置为 1，则执行关闭操作后，需要等待一段时间，等待套接字状态变为 CLOSED；如果 linger 标志位为 0，则执行关闭操作后立刻返回，并不等待一段时间。此处一再提到的执行关闭是指调用下层关闭函数（通过调用 sock->prot->close 函数指针所指函数完成的，如 tcp_close）。注意在 linger 标志位设置为 1 时的等待是通过定时器完成的。在设置定时器后，函数进入一个 while 循环，不断地检测套接字状态。其中 while 语句为真的条件是套接字只是处于“正在关闭”状态（见上文中 closing 函数的实现），而非“关闭（TCP_CLOSED）”状态，另外等待时间未超时。如果等待过程发生中断，则停止等待，返回 ERESTARTSYS，该返回值表示上层应用程序应该重新执行关闭函数。release_sock 函数完成的功能是将 sock 结构中缓存的数据包送给应用程序读取，并在检测套接字已处于关闭且等待销毁时，设置一定时器，在定时器到期时，进行 sock 结构的释放，具体参见下文中对 sock.c 文件分析。inet_release:函数最后解除 sock 结构与 socket 结构之间的互相引用关系，因为二者即将被释放，之前为实现函数调用而维持的上下层次关系已无必要。

```
734 /* this needs to be changed to disallow
735      the rebinding of sockets.      What error
736      should it return? */

737 static int inet_bind(struct socket *sock, struct sockaddr *uaddr,
738                      int addr_len)
739 {
740         struct sockaddr_in *addr=(struct sockaddr_in *)uaddr;
741         struct sock *sk=(struct sock *)sock->data, *sk2;
742         unsigned short snum = 0 /* Stoopid compiler.. this IS ok */;
743         int chk_addr_ret;

744         /* check this error. */
    //在进行地址邦定时，该套接字应该处于关闭状态，否则错误返回。
    //如果套接字处于其它任何状态，则表示该套接字已经被邦定，否则无法进入其它状态。
745         if (sk->state != TCP_CLOSE)
746                 return(-EIO);
    //地址长度字段必须不小于 sockaddr_in 结构长度，否则表示邦定的地址有问题。
747         if(addr_len<sizeof(struct sockaddr_in))
748                 return -EINVAL;

749         if(sock->type != SOCK_RAW)
750         {
        //对于非原始套接字类型，在邦定地址之前，其应该尚无断口号。
        //而对于原始套接字，在 socket 系统调用时，protocol 参数作为端口号处理。
        //具体参见前文中对 inet_create 函数的分析。
751                 if (sk->num != 0)
752                         return(-EINVAL);
        //获取邦定地址中端口号，如果端口号为 0，则系统自动分配一个，这在客户端
        //编程时是经常发生的事。
        snum = ntohs(addr->sin_port);

753                 snum = ntohs(addr->sin_port);

754                 /*
755                  * We can't just leave the socket bound wherever it is, it might
756                  * be bound to a privileged port. However, since there seems to
757                  * be a bug here, we will leave it if the port is not privileged.
758                  */
759                 if (snum == 0)
760                 {
761                         snum = get_new_socknum(sk->prot, 0);
762                 }
        //非超级用户，不可使用 1024 以下的端口。对于自动分配端口号，对于非超级用
```

```
        //户，系统不会分配到 1024 以下的端口的。所以以下 if 语句主要是对用户传入的
        //端口号进行检查。
763             if (snum < PROT_SOCK && !suser())
764                     return(-EACCES);
765     }
    //ip_chk_addr 函数用于检查地址是否是一个本地接口地址。
766     chk_addr_ret = ip_chk_addr(addr->sin_addr.s_addr);
    //如果指定的地址不是本地地址，并且也不是一个多播地址，则错误返回：地址不可用！
767     if (addr->sin_addr.s_addr != 0 && chk_addr_ret != IS_MYADDR && chk_addr_ret !=
IS_MULTICAST)
768             return(-EADDRNOTAVAIL);/* Source address MUST be ours! */
    //如果没有指定地址，则系统自动分配一个本地地址。
769     if (chk_addr_ret || addr->sin_addr.s_addr == 0)
770             sk->saddr = addr->sin_addr.s_addr;
    //此处又对原始套接字进行了区分，对于原始套接字，有专门的 sock 结构队列。
    //对于非原始套接字，则需要将此分配了端口号的 sock 结构插入到对应协议的 sock 结
    //构队列中。有关 sock 结构队列的解释在前文中介绍 sock 结构时有较为详细的说明。
771     if(sock->type != SOCK_RAW)
772     {
773             /* Make sure we are allowed to bind here. */
774             cli();
        //根据端口号寻找到对应的 sock 结构队列后，对队列中已有的 sock 结构进行检查。
775             for(sk2 = sk->prot->sock_array[snum & (SOCK_ARRAY_SIZE -1)];
776                             sk2 != NULL; sk2 = sk2->next)
777             {
778             /* should be below! */
            //如果此已有 sock 结构端口号不同此带插入 sock 结构，则表示无冲突，继续
            //进行下一个 sock 结构的检查。
779                     if (sk2->num != snum)
780                             continue;
            //二者端口号相同，检查该新套接字是否设置了“地址复用”标志，如果没有
            //设置，则返回地址复用错误。这就是通过应用程序编程时使用的
            //REUSEADDR 选项。
781                     if (!sk->reuse)
782                     {
783                             sti();
784                             return(-EADDRINUSE);
785                     }

786                     if (sk2->num != snum)
787                             continue;       /* more than one */
            //以下的检查是针对地址的，与端口号的处理类似。
788                     if (sk2->saddr != sk->saddr)
789                             continue; /* socket per slot ! -FB */
```

```
                //如果 sock 结构的状态是 TCP_LISTEN,则表示该套接字是一个服务端，服务
                //端不可使用地址复用选项。
790                 if (!sk2->reuse || sk2->state==TCP_LISTEN)
791                 {
792                     sti();
793                     return(-EADDRINUSE);
794                 }
795             }
796             sti();
            //remove_sock 函数将 sock 结构从其之前队列中删除，由 put_sock 函数根据新分配
            //的端口号插入到新的队列中。
797             remove_sock(sk);
798             put_sock(snum, sk);
            //初始化 dummy_th 字段，此时本地端口，本地地址子字段都可进行初始化。
799             sk->dummy_th.source = ntohs(sk->num);
800             sk->daddr = 0;
801             sk->dummy_th.dest = 0;
802         }
803         return(0);
804 }
```

inet_bind 函数是 sock_bind 调用的下层函数，完成本地地址邦定。本地地址邦定包括 IP 地址和端口号两个部分。注意如果没有具体指定地址和端口号，则系统在此会自动进行分配。函数实现的其它方面涉及 sock 结构的重新定位。

```
805 /*
806  *  Handle sk->err properly. The cli/sti matter.
807  */

808 static int inet_error(struct sock *sk)
809 {
810     unsigned long flags;
811     int err;
812     save_flags(flags);
813     cli();
814     err=sk->err;
815     sk->err=0;
816     restore_flags(flags);
817     return -err;
818 }
```

inet_error 函数用于返回套接字通信过程中发生的问题。注意 sock 结构 err 字段是一个正数，返回时是作为负数返回的。在 Linux 系统中，错误的返回都是以负数形式进行的。注意在返回错误时会清除当前的错误值。

```
819 /*
820  *    Connect to a remote host. There is regrettably still a little
821  *    TCP 'magic' in here.
822  */

823 static int inet_connect(struct socket *sock, struct sockaddr * uaddr,
824                 int addr_len, int flags)
825 {
826         struct sock *sk=(struct sock *)sock->data;
827         int err;
828         sock->conn = NULL;

829         if (sock->state == SS_CONNECTING && tcp_connected(sk->state))
830         {
831                 sock->state = SS_CONNECTED;
832                 /* Connection completing after a connect/EINPROGRESS/select/connect */
833                 return 0;  /* Rock and roll */
834         }

835         if (sock->state == SS_CONNECTING && sk->protocol == IPPROTO_TCP && (flags &
O_NONBLOCK))
836                 return -EALREADY;     /* Connecting is currently in progress */
```

inet_connect 函数时 sock_connect 下层调用函数，完成套接字连接操作。
函数首先检查已有的连接状态，如果现在已经处于连接状态，则简单返回。由此可见，应用程序在连接建立后多次调用 connect 函数是可以容忍的。如果套接字正在处于建立连接阶段，并且套接字使用 TCP 协议，如果使用了非阻塞选项，则立刻返回连接正在进行错误。注意只有对 TCP 协议才有可能出现这种情况，因为 UDP 协议连接的建立不涉及网络数据的传输，所以虽然 UDP 也存在连接操作，但这种连接是软件上的，是立刻可以完成的。而 TCP 协议 需要网络数据的传输。从此处的代码可以看出，对于 TCP 协议在连接正在进行时，不可连续调用 connect 函数。在对该函数继续分析之前，需要对 TCP 协议的三路握手建立连接的过程进行简单的介绍。

TCP 协议是一种面向连接的流式传输协议，其声称提供可靠性数据传输。使用 TCP 协议必须首先与远端主机建立连接，这是 TCP 协议为保证可靠性数据传输要求的（附录一对此有介绍）。另外为保证 TCP 协议所声称的可靠性数据传输，TCP 协议实现软件对使用该协议的套接字维护其通信状态如 TCP_CLOSED, TCP_SYN_SENT, TCP_FIN_WAIT1 等等，这些状态的维护也是 TCP 协议 RFC 文档所要求的，所以使用 TCP 协议的套接字在通信过程的不同阶段是在这个状态机上不断的转变的。一般的，一个使用 TCP 协议的套接字需要经过几个状态的转变，根据是服务器端还是客户端，其中的转变过程有些不同（此处对于客户端和服务器的定义是：将主动发起连接操作和主动发起关闭操作的一端称为客户端）。

```
/*include/linux/tcp.h*/
52   enum {
53       TCP_ESTABLISHED = 1,
54       TCP_SYN_SENT,
```

```
55      TCP_SYN_RECV,
56      TCP_FIN_WAIT1,
57      TCP_FIN_WAIT2,
58      TCP_TIME_WAIT,
59      TCP_CLOSE,
60      TCP_CLOSE_WAIT,
61      TCP_LAST_ACK,
62      TCP_LISTEN,
63      TCP_CLOSING /* now a valid state */
64  };
```

对于客户端而言，我们一般假设其主动发送连接请求，连接请求是通过发送一个 TCP 首部中 SYN 标志位设置为 1 的数据包完成的。客户端在发送该数据包后，将 sock 结构中 state 字段从 TCP_CLOSE 更新为 TCP_SYN_SENT,表示本地已发送 SYN 请求连接数据包。此时本地将等待服务器应答数据包。服务器监听套接字在接收到远端客户端一个连接请求后，调用相关函数进行处理，这些处理包括分配一个新的套接字用于与远端客户端的实际通信，将这个请求客户端挂入服务器端请求队列等待 accept 函数的读取，之后发送应答数据包给远端，并将之前创建的新的套接字对于 sock 结构中 state 字段值更新为 TCP_SYN_RECV.客户端在接收到服务器发送的应答数据包后，将其状态进一步更新为 TCP_ESTABLISHED，同时发送一个应答数据包给服务器端，服务器端在接收到此应答数据包后将对应通信 sock 结构状态更新为 TCP_ESTABLISHED，二者正式建立连接。这个连接建立过程如下图（图 2-3）所示。

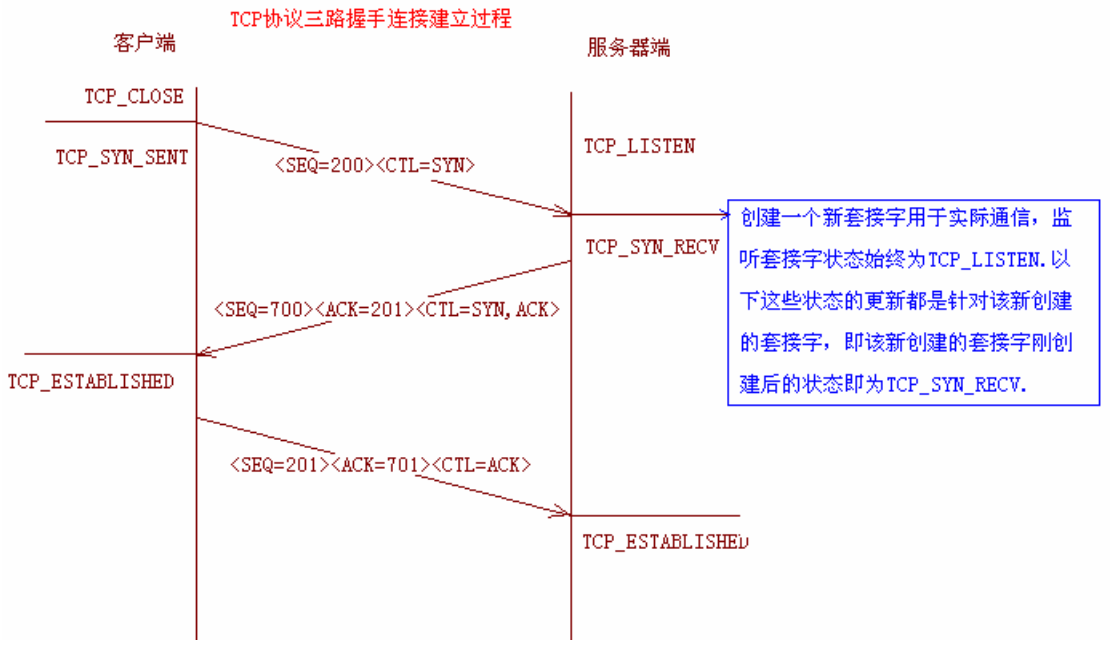


图 2-3 TCP 协议三路握手连接建立过程

```
837         if (sock->state != SS_CONNECTING)
838         {
839             /* We may need to bind the socket. */
840             if(inet_autobind(sk)!=0)
```

```
841                 return(-EAGAIN);
842             if (sk->prot->connect == NULL)
843                 return(-EOPNOTSUPP);
844             err = sk->prot->connect(sk, (struct sockaddr_in *)uaddr, addr_len);
845             if (err < 0)
846                 return(err);
847             sock->state = SS_CONNECTING;
848         }
```

这个 if 语句块判断如果套接字是初次调用 connect 函数则调用下层函数进行实质上的连接建立。并将 socket 结构状态更新为 SS_CONNECTING。注意如果本地尚未邦定地址，则内核会在进行实际连接之间进行地址的自动邦定。对于 TCP 协议而言，sk->prot->connect 实际上调用的是 tcp_connect 函数。该函数将发送 SYN 数据包进行三路握手连接建立过程（如图 2-3 所示）。注意在 tcp_connect 函数中，会将 sock 结构状态更新为 TCP_SYN_SENT.这一更新方使得 inet_connect 函数之后的判断有意义。再看接下来的代码。

```
849         if (sk->state > TCP_FIN_WAIT2 && sock->state==SS_CONNECTING)
850         {
851             sock->state=SS_UNCONNECTED;
852             cli();
853             err=sk->err;
854             sk->err=0;
855             sti();
856             return -err;
857         }
```

如果 sock 结构状态大于 TCP_FIN_WAIT2,则表示 sock 状态可能为如下状态：TCP_TIME_WAIT, TCP_CLOSE, TCP_CLOSE_WAIT.因为在 tcp_connect 函数中 sock 结构更新为 TCP_SYN_SENT, 而现在变成了如上一些状态，表示在连接过程中出现了异常，而 socket 结构状态却表示连接正在进行。对于初次连接这个 if 条件是不可能为真的，只有在多次调用 connect 函数的情况下，此处条件才有可能为真。无论如何，在条件为真的情况下，都表示出现连接错误，此时需要更新 socket 结构状态为 SS_UNCONNECTED, 并返回具体的错误。

在调用底层函数完成具体连接时，此时需要根据阻塞标志位进行等待或立刻返回的操作。所谓等待则需要等待连接完全建立方才返回。而立刻返回的含义是指将连接任务交给下层后即可安全返回。等待操作是在一个 while 循环中进行的，注意 while 循环的条件是 sock 状态为 TCP_SYN_SENT 或者 TCP_SYN_RECV。这两种状态是在连接过程中唯一合法的状态。对于客户端而言，建立连接时常见状态为 TCP_SYN_SENT, 只有当远端同时发送连接建立操作时才可能出现 TCP_SYN_RECV 状态。双方同时建立连接的过程如下图（图 2-4）所示。

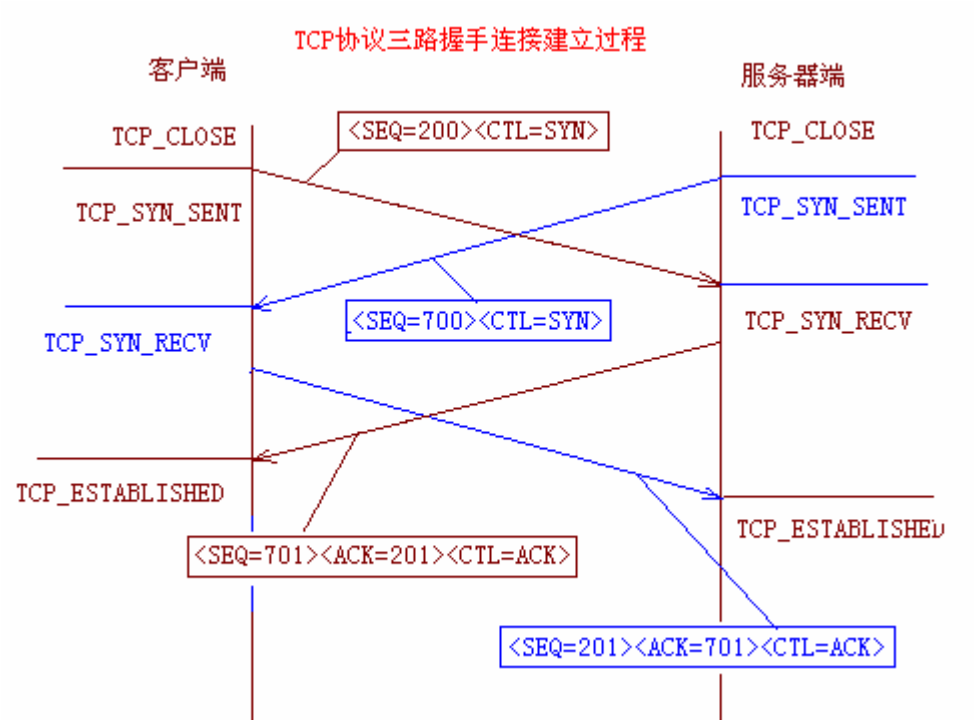


图 2-4 TCP 协议通信双方同时发起连接建立时的情况

```
858     if (sk->state != TCP_ESTABLISHED &&(flags & O_NONBLOCK))
859         return(-EINPROGRESS);

860     cli(); /* avoid the race condition */
861     while(sk->state == TCP_SYN_SENT || sk->state == TCP_SYN_RECV)
862     {
863         interruptible_sleep_on(sk->sleep);
864         if (current->signal & ~current->blocked)
865         {
866             sti();
867             return(-ERESTARTSYS);
868         }
869         /* This fixes a nasty in the tcp/ip code. There is a hideous hassle with
870             icmp error packets wanting to close a tcp or udp socket. */
871         if(sk->err && sk->protocol == IPPROTO_TCP)
872         {
873             sti();
874             sock->state = SS_UNCONNECTED;
875             err = -sk->err;
876             sk->err=0;
877             return err; /* set by tcp_err() */
878         }
879     }
880     sti();
```

```
881         sock->state = SS_CONNECTED;

882         if (sk->state != TCP_ESTABLISHED && sk->err)
883         {
884                 sock->state = SS_UNCONNECTED;
885                 err=sk->err;
886                 sk->err=0;
887                 return(-err);
888         }
889         return(0);
890 }
```

注意如果在等待过程中出现错误，则简单返回错误并置状态为未连接。
如果正常退出 while 循环，则表示连接成功，不过在函数最后还是对 sock 结构状态及错误字段进行了检查。一切正常时，sock 状态将被设置为 TCP_ESTABLISHED, 而 socket 结构状态将被设置为 SS_CONNECTED. 注意 sock 结构和 socket 结构各自使用不同的状态集合。对于 inet_connect 函数的单一分析很难深刻理解前文中的意义，读者可暂时结合 tcp.c 文件中 tcp_connect 函数进行理解。使用 TCP 协议的套接字其状态的转化相对而言是比较麻烦的。关于 TCP 协议的详细说明将在下文中分析 tcp.c 文件是给出。读者也可直接阅读 RFC-793 文档。

```
891 static int inet_socketpair(struct socket *sock1, struct socket *sock2)
892 {
893          return(-EOPNOTSUPP);
894 }
```

对于 INET 域，不存在配对套接字。配对套接字仅对 UNIX 域有效，所以该函数简单返回 EOPNOTSUPP 错误标志（表示不支持该功能）。

```
895 /*
896  *    Accept a pending connection. The TCP layer now gives BSD semantics.
897  */

898 static int inet_accept(struct socket *sock, struct socket *newsock, int flags)
899 {
900         struct sock *sk1, *sk2;
901         int err;

902         sk1 = (struct sock *) sock->data;

903         /*
904          * We've been passed an extra socket.
905          * We need to free it up because the tcp module creates
906          * its own when it accepts one.
907          */
```

```
908     if (newsock->data)
909     {
910         struct sock *sk=(struct sock *)newsock->data;
911         newsock->data=NULL;
912         sk->dead = 1;
913         destroy_sock(sk);
914     }

915     if (sk1->prot->accept == NULL)
916         return(-EOPNOTSUPP);

917     /* Restore the state if we have been interrupted, and then returned. */
918     if (sk1->pair != NULL )
919     {
920         sk2 = sk1->pair;
921         sk1->pair = NULL;
922     }
923     else
924     {
925         sk2 = sk1->prot->accept(sk1,flags);
926         if (sk2 == NULL)
927         {
928             if (sk1->err <= 0)
929                 printk("Warning sock.c:sk1->err <= 0.  Returning non-error.\n");
930             err=sk1->err;
931             sk1->err=0;
932             return(-err);
933         }
934     }
935     newsock->data = (void *)sk2;
936     sk2->sleep = newsock->wait;
937     sk2->socket = newsock;
938     newsock->conn = NULL;
939     if (flags & O_NONBLOCK)
940         return(0);

941     cli(); /* avoid the race. */
942     while(sk2->state == TCP_SYN_RECV)
943     {
944         interruptible_sleep_on(sk2->sleep);
945         if (current->signal & ~current->blocked)
946         {
947             sti();
948             sk1->pair = sk2;
```

```
949                     sk2->sleep = NULL;
950                     sk2->socket=NULL;
951                     newsock->data = NULL;
952                     return(-ERESTARTSYS);
953                 }
954         }
955         sti();

956         if (sk2->state != TCP_ESTABLISHED && sk2->err > 0)
957         {
958                 err = -sk2->err;
959                 sk2->err=0;
960                 sk2->dead=1;  /* ANK */
961                 destroy_sock(sk2);
962                 newsock->data = NULL;
963                 return(err);
964         }
965         newsock->state = SS_CONNECTED;
966         return(0);
967 }
```

inet_accept 用于接收一个套接字。该函数是 accept 函数的 INET 层相应函数，由服务器端调用。服务器端在接收到一个远端客户端连接请求后（在传输层），调用相应函数进行处理（如 TCP 协议的 tcp_conn_request 函数）。诚如上文中所述，所谓处理指：
1>创建一个本地新套接字用于通信，原监听套接字仍然用于监听请求。
2>发送应答数据包
3>将新创建套接字对于的 sock 结构挂接到监听 sock 结构的接收队列中（receive_queue 指向的队列）以供 accept 函数读取。

inet_accept 是由 sock_accept 函数调用的，调用形式如下：
i=newsock->ops->accept(sock, newsock, file->f_flags);
其中 newsock 为 sock_accept 为新的通信套接字创建的对应 socket 结构。
sock_accept 函数在调用 inet_accept 函数之前已经使用 inet_dup 函数创建了其对应的 sock 结构，即新创建 socket 结构的 data 字段指向其对应的新创建的 sock 结构。

所以在 inet_accept 函数中对于 socket->data 字段的检查是必要的，而且所得结果一般都不为 NULL，而是指向了对应的 sock 结构。所以以下代码一般不会执行到。

```
if (newsock->data)
{
        struct sock *sk=(struct sock *)newsock->data;
        newsock->data=NULL;
        sk->dead = 1;
        destroy_sock(sk);
}
```

而如下的 if 语句用于检测是否存在下层处理函数，如果不存在，则错误返回。同前文所说，INET 层函数并不进行实质上的处理，而是将请求传递给下层，如果下层没有对应的处理函数，则必须出错返回。

```
    if (sk1->prot->accept == NULL)
        return(-EOPNOTSUPP);
```

```
/* Restore the state if we have been interrupted, and then returned. */
    if (sk1->pair != NULL )
    {
        sk2 = sk1->pair;
        sk1->pair = NULL;
    }
```

对于这个 if 语句的必要性可以从函数的结尾处代码理解。如果套接字在等待连接的过程中被中断，则监听套接字 sock 结构之 pair 字段将被初始化为指向该被中断的套接字对应的 sock 结构，这样在下一次调用 accept 函数时，可以优先处理这个之前被中断的套接字。

否则将按正常的程序从监听套接字的接收队列中取下一个 sock 结构（该结构是由传输层创建的用于与远端通信的 sock 结构，所以对于监听套接字而言，其对应 sock 结构中接收队列中均是经过处理的请求连接数据包，“经过处理”是指对于这些请求本地都创建了新的 sock 结构）。

```
    newsock->data = (void *)sk2;
    sk2->sleep = newsock->wait;
    sk2->socket = newsock;
    newsock->conn = NULL;
    if (flags & O_NONBLOCK)
        return(0);
```

此后代码初始化 socket，sock 结构各自对应的字段使二者相互联系起来。

之后代码检测是否已经完全建立起连接，这是在 while 循环中进行的，注意如果等待中被中断，则将监听套接字 pair 字段赋值为被中断的套接字，以便下次可以优先处理。

如果 while 循环正常退出，则表示成功建立连接，此后即可进行数据的传送。如果异常退出，则表示出现错误，此时函数的处理是销毁该出错套接字对应的 socket 和 sock 结构（sock 结构是在本函数进行销毁的，而 socket 结构将在 sock_accept 函数被销毁）。否则一切正常，则更改 socket 状态，并正常返回。此时一个通道即成功建立，之后可以进行数据的正常传输。

```
968 /*
969  *  This does both peername and sockname.
970  */

971 static int inet_getname(struct socket *sock, struct sockaddr *uaddr,
972                 int *uaddr_len, int peer)
973 {
974         struct sockaddr_in *sin=(struct sockaddr_in *)uaddr;
```

```
975     struct sock *sk;

976     sin->sin_family = AF_INET;
977     sk = (struct sock *) sock->data;
978     if (peer)
979     {
980         if (!tcp_connected(sk->state))
981             return(-ENOTCONN);
982         sin->sin_port = sk->dummy_th.dest;
983         sin->sin_addr.s_addr = sk->daddr;
984     }
985     else
986     {
987         sin->sin_port = sk->dummy_th.source;
988         if (sk->saddr == 0)
989             sin->sin_addr.s_addr = ip_my_addr();
990         else
991             sin->sin_addr.s_addr = sk->saddr;
992     }
993     *uaddr_len = sizeof(*sin);
994     return(0);
995 }
```

inet_getname 函数用于获取本地和远端地址。改函数是上层 BSD 层 sock_getsockname，sock_getpeername 的 INET 层相应函数。改函数中 peer 参数表示获取本地还是远端地址。对于已经成功建立连接的 TCP 协议而言，远端地址封装在 sock 结构字段中。sock 结构中 daddr 字段表示远端 IP 地址，而 dummy_th 字段用于向远端传送数据包时进行数据包中 TCP 首部的创建，即该字段中含有远端通信端口号。如果是获取远端地址，则这些值即来自于 sock 结构的这些字段，而如果获取本地地址值，原理基本相同。

ip_my_addr 函数定义在 devinet.c 文件中，该函数较短，故先列出如下：

```
/*net/inet/devinet.c*/
144 unsigned long ip_my_addr(void)
145 {
146     struct device *dev;

147     for (dev = dev_base; dev != NULL; dev = dev->next)
148     {
149         if (dev->flags & IFF_LOOPBACK)
150             return(dev->pa_addr);
151     }
152     return(0);
153 }
```

ip_my_addr 的基本实现是通过遍历系统中所有设备，检查该设备标志位中是否包含

IFF_LOOPBACK，如果包含，则返回该设备所配置的 IP 地址。device 结构中 pa_addr 字段是一个 unsigned long 类型，正好表示一个 IP 地址。
注意设置有 IFF_LOOPBACK 标志位的设备通常并不表示其是一个回环设备。通常我们将具有地址值为 127.0.0.1 称为回环设备，不过这个设备通常只存在于软件概念上，即实际并无真正的硬件与之对应。对于一个实际硬件网卡设备，有时我们也对其设置 IFF_LOOPBACK 标志位，从而当从该设备发送数据包时，该设备会回送一份数据包拷贝给本机。

从 inet_getsockname 函数的实现来看，获取地址值对于底层实现而言十分简单的：只是从底层数据结构中读取字段值。

```
996  /*
997   *    The assorted BSD I/O operations
998   */

999  static int inet_recvfrom(struct socket *sock, void *ubuf, int size, int noblock,
1000                        unsigned flags, struct sockaddr *sin, int *addr_len )
1001      {
1002          struct sock *sk = (struct sock *) sock->data;

1003          if (sk->prot->recvfrom == NULL)
1004              return(-EOPNOTSUPP);
1005          if(sk->err)
1006              return inet_error(sk);
1007          /* We may need to bind the socket. */
1008          if(inet_autobind(sk)!=0)
1009              return(-EAGAIN);
1010          return(sk->prot->recvfrom(sk, (unsigned char *) ubuf, size, noblock, flags,
1011                      (struct sockaddr_in*)sin, addr_len));
1012      }
```

inet_recvfrom 是 BSD 层 sock_recvfrom 函数的 INET 层实现函数，当用户调用 recvfrom 函数时，该请求被传递给 sock_recvfrom 函数，而 sock_recvfrom 函数继续调用 inet_recvfrom 函数处理，即该函数。从其实现来看，其继续调用下层（传输层）函数处理请求（对于 TCP 协议而言，即 tcp_recvfrom）。inet_recvfrom 函数在调用下层函数继续处理之前，进行了处理前检查，包括是否定义了下层可调用函数，在这之前是否出现错误，以及该套接字是否绑定了本地地址（主要是断口号）。
注意该函数参数：ubuf 表示读取数据的用户缓冲区，size 表示用户请求的数据长度，noblock 表示在底层尚未读取到数据时是否进行等待，flags 是一些标志位的设置用于表示某种特定信息，如 MSG_OOB 表示读取“带外”数据（一般底层将其解释为紧急数据），MSG_PEEK 标志位表示只是预读取数据，数据被读取后底层仍然保存这些数据以便以后进行正式读取。

```
1013      static int inet_recv(struct socket *sock, void *ubuf, int size, int noblock,
1014                  unsigned flags)
1015      {
```

```
1016            /* BSD explicitly states these are the same - so we do it this way to be sure */
1017            return inet_recvfrom(sock,ubuf,size,noblock,flags,NULL,NULL);
1018    }
```

该函数通过调用 inet_recvfrom 函数实现，本质与 inet_recvfrom，唯一不同之处在于该函数无需返回数据包发送端的地址值，所以对应参数值设置为 NULL。

```
1019    static int inet_read(struct socket *sock, char *ubuf, int size, int noblock)
1020    {
1021            struct sock *sk = (struct sock *) sock->data;

1022            if(sk->err)
1023                    return inet_error(sk);
1024            /* We may need to bind the socket. */
1025            if(inet_autobind(sk))
1026                    return(-EAGAIN);
1027            return(sk->prot->read(sk, (unsigned char *) ubuf, size, noblock, 0));
1028    }
```

能够调用该函数则一定是使用面向连接的协议，对于目前协议簇而言，一般情况下即指 TCP 协议，所以在对 inet_autobind 函数的调用处理上稍微有些不同。inet_autobind 函数在检测到参数对应的套接字尚未分配本地段口号时会自动分配一个端口号，对于 TCP 协议而言，由于需要一个建立连接的过程，而在建立过程中一定会对本地地址进行绑定，所以对于使用 TCP 协议的套接字此处对于 inet_autobind 的调用完全没有必要，而对于其它面向连接的协议类型，同样此处对于 inet_autobind 函数的调用也无必要。对于 inet_read 函数的实现为代码更严密性考虑，倒是应该添加如下一条语句：

```
if (sk->prot->read==NULL)
    return (-EOPNOTSUPP);
```

以下三个函数是针对 INET 层数据发送而言的。基本实现方式是通过调用下层（传输层）函数传递上层请求。在调用之前进行必要的检查。读者可参考以上对于数据读取函数的说明来对这些数据发送函数进行分析。

```
1029    static int inet_send(struct socket *sock, void *ubuf, int size, int noblock,
1030                         unsigned flags)
1031    {
1032            struct sock *sk = (struct sock *) sock->data;
1033            if (sk->shutdown & SEND_SHUTDOWN)
1034            {
1035                    send_sig(SIGPIPE, current, 1);
1036                    return(-EPIPE);
1037            }
1038            if(sk->err)
1039                    return inet_error(sk);
```

```
1040        /* We may need to bind the socket. */
1041        if(inet_autobind(sk)!=0)
1042            return(-EAGAIN);
1043        return(sk->prot->write(sk, (unsigned char *) ubuf, size, noblock, flags));
1044    }

1045    static int inet_write(struct socket *sock, char *ubuf, int size, int noblock)
1046    {
1047        return inet_send(sock,ubuf,size,noblock,0);
1048    }

1049    static int inet_sendto(struct socket *sock, void *ubuf, int size, int noblock,
1050                unsigned flags, struct sockaddr *sin, int addr_len)
1051    {
1052        struct sock *sk = (struct sock *) sock->data;
1053        if (sk->shutdown & SEND_SHUTDOWN)
1054        {
1055            send_sig(SIGPIPE, current, 1);
1056            return(-EPIPE);
1057        }
1058        if (sk->prot->sendto == NULL)
1059            return(-EOPNOTSUPP);
1060        if(sk->err)
1061            return inet_error(sk);
1062        /* We may need to bind the socket. */
1063        if(inet_autobind(sk)!=0)
1064            return -EAGAIN;
1065        return(sk->prot->sendto(sk, (unsigned char *) ubuf, size, noblock, flags,
1066                    (struct sockaddr_in *)sin, addr_len));
1067    }


1068    static int inet_shutdown(struct socket *sock, int how)
1069    {
1070        struct sock *sk=(struct sock*)sock->data;

1071        /*
1072          * This should really check to make sure
1073          * the socket is a TCP socket. (WHY AC...)
1074          */
1075        how++; /* maps 0->1 has the advantage of making bit 1 rcvs and
1076                    1->2 bit 2 snds.
1077                    2->3 */
1078        if ((how & ~SHUTDOWN_MASK) || how==0)    /* MAXINT->0 */
```

```
1079                return(-EINVAL);
1080            if (sock->state == SS_CONNECTING && sk->state == TCP_ESTABLISHED)
1081                sock->state = SS_CONNECTED;
1082            if (!tcp_connected(sk->state))
1083                return(-ENOTCONN);
1084            sk->shutdown |= how;
1085            if (sk->prot->shutdown)
1086                sk->prot->shutdown(sk, how);
1087            return(0);
1088        }
```

inet_shutdown 被上层 sock_shutdown 以及 sock_release 函数调用用于套接字的关闭。
参数 how 用于表示对套接字如何进行关闭操作：即进行半关闭还是全关闭。注意在进一步调用下层函数进行处理之前，how 参数被加 1 处理；而且对套接字目前所处的状态进行验证，如果套接字目前实际上并非处于“连接建立”状态（tcp_connected 函数返回 false），则错误返回，因为不能对一个未连接套接字进行关闭操作。函数最后调用传输层函数（如 tcp_shutdown）进行实际关闭操作。

```
1089    static int inet_select(struct socket *sock, int sel_type, select_table *wait )
1090    {
1091        struct sock *sk=(struct sock *) sock->data;
1092        if (sk->prot->select == NULL)
1093        {
1094            return(0);
1095        }
1096        return(sk->prot->select(sk, sel_type, wait));
1097    }
```

inet_select 函数被上层 sock_select 函数调用，而 inet_select 进一步调用下层函数进行处理（如 tcp_select）。参数 sock 表示被进行检查的套接字，而 sel_type 表示检查的类型：SEL_IN（检查是否有输入数据），SEL_OUT（检查是否有待发送数据），SEL_EX（检查是否出错以及是否有紧急数据需要读取）。至于 select_table 结构类型的 wait 参数用于等待操作，该结构定义在 include/linux/wait.h 文件中，由于该结构不涉及到网络代码的本质部分，此处不做说明。

```
1098    /*
1099     *    ioctl() calls you can issue on an INET socket. Most of these are
1100 *    device configuration and stuff and very rarely used. Some ioctls
1101 *    pass on to the socket itself.
1102 *
1103 *    NOTE: I like the idea of a module for the config stuff. ie ifconfig
1104 *    loads the devconfigure module does its configuring and unloads it.
1105 *    There's a good 20K of config code hanging around the kernel.
1106 */
```

```
1107static int inet_ioctl(struct socket *sock, unsigned int cmd, unsigned long arg)
1108{
1109     struct sock *sk=(struct sock *)sock->data;
1110     int err;

1111     switch(cmd)
1112     {
1113          case FIOSETOWN:
1114          case SIOCSPGRP:
1115               err=verify_area(VERIFY_READ,(int *)arg,sizeof(long));
1116               if(err)
1117                    return err;
1118               sk->proc = get_fs_long((int *) arg);
1119               return(0);
1120          case FIOGETOWN:
1121          case SIOCGPGRP:
1122               err=verify_area(VERIFY_WRITE,(void *) arg, sizeof(long));
1123               if(err)
1124                    return err;
1125               put_fs_long(sk->proc,(int *)arg);
1126               return(0);
1127          case SIOCGSTAMP:
1128               if(sk->stamp.tv_sec==0)
1129                    return -ENOENT;
1130               err=verify_area(VERIFY_WRITE,(void *)arg,sizeof(struct timeval));
1131               if(err)
1132                    return err;
1133               memcpy_tofs((void *)arg,&sk->stamp,sizeof(struct timeval));
1134               return 0;
1135          case SIOCADDRT: case SIOCADDRTOLD:
1136          case SIOCDELRT: case SIOCDELRTOLD:
1137               return(ip_rt_ioctl(cmd,(void *) arg));
1138          case SIOCDARP:
1139          case SIOCGARP:
1140          case SIOCSARP:
1141               return(arp_ioctl(cmd,(void *) arg));
1142#ifdef CONFIG_INET_RARP
1143          case SIOCDRARP:
1144          case SIOCGRARP:
1145          case SIOCSRARP:
1146               return(rarp_ioctl(cmd,(void *) arg));
1147#endif
1148          case SIOCGIFCONF:
1149          case SIOCGIFFLAGS:
```

```
1150            case SIOCSIFFLAGS:
1151            case SIOCGIFADDR:
1152            case SIOCSIFADDR:

1153/* begin multicast support change */
1154            case SIOCADDMULTI:
1155            case SIOCDELMULTI:
1156/* end multicast support change */

1157            case SIOCGIFDSTADDR:
1158            case SIOCSIFDSTADDR:
1159            case SIOCGIFBRDADDR:
1160            case SIOCSIFBRDADDR:
1161            case SIOCGIFNETMASK:
1162            case SIOCSIFNETMASK:
1163            case SIOCGIFMETRIC:
1164            case SIOCSIFMETRIC:
1165            case SIOCGIFMEM:
1166            case SIOCSIFMEM:
1167            case SIOCGIFMTU:
1168            case SIOCSIFMTU:
1169            case SIOCSIFLINK:
1170            case SIOCGIFHWADDR:
1171            case SIOCSIFHWADDR:
1172            case OLD_SIOCGIFHWADDR:
1173            case SIOCSIFMAP:
1174            case SIOCGIFMAP:
1175            case SIOCSIFSLAVE:
1176            case SIOCGIFSLAVE:
1177                return(dev_ioctl(cmd,(void *) arg));

1178            default:
1179                if ((cmd >= SIOCDEVPRIVATE) &&
1180                    (cmd <= (SIOCDEVPRIVATE + 15)))
1181                    return(dev_ioctl(cmd,(void *) arg));

1182                if (sk->prot->ioctl==NULL)
1183                    return(-EINVAL);
1184                return(sk->prot->ioctl(sk, cmd, arg));
1185        }
1186        /*NOTREACHED*/
1187        return(0);
1188}
```

inet_ioctl 函数用于对套接字选项进行控制。该函数根据需要改变的不同选项下发任务到具体的下层模块进行处理，如对于 ARP 协议中使用的地址映射项目进行添加和删除操作时，则调用 arp_ioctl 下层函数进行处理，而对设备进行控制时则调用 dev_ioctl 函数进行处理等等。

```
1189/*
1190  * This routine must find a socket given a TCP or UDP header.
1191  * Everything is assumed to be in net order.
1192  *
1193  * We give priority to more closely bound ports: if some socket
1194  * is bound to a particular foreign address, it will get the packet
1195  * rather than somebody listening to any address..
1196  */

1197struct sock *get_sock(struct proto *prot, unsigned short num,
1198                      unsigned long raddr,
1199                      unsigned short rnum, unsigned long laddr)
1200      {
1201          struct sock *s;
1202          struct sock *result = NULL;
1203          int badness = -1;
1204          unsigned short hnum;

1205          hnum = ntohs(num);

1206          /*
1207           * SOCK_ARRAY_SIZE must be a power of two.    This will work better
1208           * than a prime unless 3 or more sockets end up using the same
1209           * array entry.    This should not be a problem because most
1210           * well known sockets don't overlap that much, and for
1211       * the other ones, we can just be careful about picking our
1212           * socket number when we choose an arbitrary one.
1213           */

1214          for(s = prot->sock_array[hnum & (SOCK_ARRAY_SIZE - 1)];
1215                  s != NULL; s = s->next)
1216          {
1217              int score = 0;

1218              if (s->num != hnum)
1219                  continue;

1220              if(s->dead && (s->state == TCP_CLOSE))
1221                  continue;
1222              /* local address matches? */
```

```
1223                if (s->saddr) {
1224                    if (s->saddr != laddr)
1225                        continue;
1226                    score++;
1227                }
1228                /* remote address matches? */
1229                if (s->daddr) {
1230                    if (s->daddr != raddr)
1231                        continue;
1232                    score++;
1233                }
1234                /* remote port matches? */
1235                if (s->dummy_th.dest) {
1236                    if (s->dummy_th.dest != rnum)
1237                        continue;
1238                    score++;
1239                }
1240                /* perfect match? */
1241                if (score == 3)
1242                    return s;
1243                /* no, check if this is the best so far.. */
1244                if (score <= badness)
1245                    continue;
1246                result = s;
1247                badness = score;
1248            }
1249            return result;
1250        }
```

get_sock 函数虽然看起来较长，原理上非常简单，即根据本地地址和远端地址查找本地套接字对应的 sock 结构。从前文中可知，对于适用同一种协议的所有套接字都被插入到该协议对应 proto（如 tcp_prot）结构中 sock_array 数组中。所以该函数即直接以本地地址和远端地址为关键字对对应协议中 sock_array 数组进行查找。注意 sock_array 是一个数组，其数组中每个元素均对应一个 sock 结构队列，数组元素用本地地址端口号进行索引。即 get_sock 函数中本地端口号用于索引 sock_array 元素（即寻找 sock 结构队列），其它参数：本地 IP 地址，远端端口号和 IP 地址用于队列中对每个 sock 结构进行匹配。

```
1251        /*
1252         *    Deliver a datagram to raw sockets.
1253         */

1254        struct sock *get_sock_raw(struct sock *sk,
1255                                  unsigned short num,
1256                                  unsigned long raddr,
```

```
1257                            unsigned long laddr)
1258    {
1259            struct sock *s;

1260            s=sk;

1261            for(; s != NULL; s = s->next)
1262            {
1263                    if (s->num != num)
1264                            continue;
1265                    if(s->dead && (s->state == TCP_CLOSE))
1266                            continue;
1267                    if(s->daddr && s->daddr!=raddr)
1268                            continue;
1269                    if(s->saddr    && s->saddr!=laddr)
1270                            continue;
1271                    return(s);
1272            }
1273            return(NULL);
1274    }
```

对于 raw 套接字专门有一个 proto 结构变量 raw_prot 进行收集：即所有使用 raw 类型的套接字均插入到 raw_prot 对应 proto 结构中 sock_array 数组中。从 ip.c 文件中 ip_rcv 函数中摘取代码片断可得知 get_sock_raw 函数被调用的环境。

```
/*以下代码片断摘自 ip.c 文件 ip_rcv 函数*/
//iph 表示所接收数据包中 IP 首部。
//对于 raw 类型套接字而言，IP 首部中 protocol 字段用于索引对应的 sock_array 数组中元素。
hash = iph->protocol & (SOCK_ARRAY_SIZE-1);
if((raw_sk=raw_prot.sock_array[hash])!=NULL)
    {
        struct sock *sknext=NULL;
        struct sk_buff *skb1;
        raw_sk=get_sock_raw(raw_sk, hash,    iph->saddr, iph->daddr);
        if(raw_sk)      /* Any raw sockets */
        {
            ……
```

在寻找到对应的 sock 结构队列后，之后的操作即进行具体匹配。

```
1275    #ifdef CONFIG_IP_MULTICAST
1276    /*
1277     *    Deliver a datagram to broadcast/multicast sockets.
1278     */

1279    struct sock *get_sock_mcast(struct sock *sk,
```

```
1280                            unsigned short num,
1281                            unsigned long raddr,
1282                            unsigned short rnum, unsigned long laddr)
1283    {
1284            struct sock *s;
1285            unsigned short hnum;

1286            hnum = ntohs(num);

1287            /*
1288             * SOCK_ARRAY_SIZE must be a power of two.    This will work better
1289             * than a prime unless 3 or more sockets end up using the same
1290             * array entry.    This should not be a problem because most
1291             * well known sockets don't overlap that much, and for
1292             * the other ones, we can just be careful about picking our
1293             * socket number when we choose an arbitrary one.
1294             */

1295            s=sk;

1296            for(; s != NULL; s = s->next)
1297            {
1298                    if (s->num != hnum)
1299                            continue;
1300                    if(s->dead && (s->state == TCP_CLOSE))
1301                            continue;
1302                    if(s->daddr && s->daddr!=raddr)
1303                            continue;
1304                    if (s->dummy_th.dest != rnum && s->dummy_th.dest != 0)
1305                            continue;
1306                    if(s->saddr    && s->saddr!=laddr)
1307                            continue;
1308                    return(s);
1309            }
1310            return(NULL);
1311}

1312    #endif
```

get_sock_mcast 函数实现本上并不复杂，读者对此应该没有问题。该函数在 udp_rcv 函数中被调用用于处理多播数据包，以下代码片断摘自 udp_rcv 函数：

```
/*net/inet/udp.c-udp_rcv*/
#ifdef CONFIG_IP_MULTICAST
    if (addr_type!=IS_MYADDR)
```

```
    {
        /*
         *  Multicasts and broadcasts go to each listener.
         */
        struct sock *sknext=NULL;
        sk=get_sock_mcast(udp_prot.sock_array[ntohs(uh->dest)&(SOCK_ARRAY_SIZE-1)],
uh->dest,
                saddr, uh->source, daddr);
        if(sk)
        {
            do
            {
                struct sk_buff *skb1;

                sknext=get_sock_mcast(sk->next, uh->dest, saddr, uh->source, daddr);
                if(sknext)
                    skb1=skb_clone(skb,GFP_ATOMIC);
                else
                    skb1=skb;
                if(skb1)
                    udp_deliver(sk, uh, skb1, dev,saddr,daddr,len);
                sk=sknext;
            }
            while(sknext!=NULL);
        }
        else
            kfree_skb(skb, FREE_READ);
        return 0;
    }
#endif
```

其中 uh 表示所接收数据包 UDP 首部指针，所以 uh->dest 表示本地通信端口号，该数字被作为索引寻址对应的 sock 队列。注意如果队列中有不止一个 sock 结构满足条件（即有多个套接字加入了同一个多播组），则将复制数据包，以便给每个套接字传送一份该数据包数据。

以下定义 INET 域操作函数集。这是由一个 proto_ops 结构表示的。对于 INET 域而言，该操作函数集由 inet_proto_ops 变量表示，对于 UNIX 域，则对应变量为 unix_proto_ops。读者可以比较前文中对 socket.c 文件的分析，从中可以看出 socket.c 文件中定义的大量函数在此都有对应的函数，如 sock_bind 对应 inet_bind,等等，即 af_inet.c 文件中定义的函数将作为 INET 层处理自

```
1313    static struct proto_ops inet_proto_ops = {
1314        AF_INET,

1315        inet_create,
1316        inet_dup,
1317        inet_release,
```

```
1318            inet_bind,
1319            inet_connect,
1320            inet_socketpair,
1321            inet_accept,
1322            inet_getname,
1323            inet_read,
1324            inet_write,
1325            inet_select,
1326            inet_ioctl,
1327            inet_listen,
1328            inet_send,
1329            inet_recv,
1330            inet_sendto,
1331            inet_recvfrom,
1332            inet_shutdown,
1333            inet_setsockopt,
1334            inet_getsockopt,
1335            inet_fcntl,
1336    };
```

af_inet.c 文件定义的最后一个函数作为 INET 层的初始化函数。其初始化对上，下层的接口和变量。

seq_offset 全局变量为本地套接字创建初始序列号。为了区分不同套接字的数据包，在新建一个套接字时，分配给其的初始序列号是由该全局变量进行统一同步的。如此可以尽量避免使用重叠的序列号。

```
1337    extern unsigned long seq_offset;

1338    /*
1339     *    Called by socket.c on kernel startup.
1340     */

1341    void inet_proto_init(struct net_proto *pro)
1342    {
1343            struct inet_protocol *p;
1344            int i;


1345            printk("Swansea University Computer Society TCP/IP for NET3.019\n");

1346            /*
1347             *    Tell SOCKET that we are alive...
1348             */
```

```
  //注册 INET 域操作函数集，域值为 AF_INET.
  //inet_proto_ops 将被插入到 pops 全局数组中。具体情况请参见前文中对 sock_register 函数
  //的分析。
1349        (void) sock_register(inet_proto_ops.family, &inet_proto_ops);

  //初始化 seq_offset，CURRENT_TIME 表示当前时间值。
1350        seq_offset = CURRENT_TIME*250;

1351        /*
1352         *  Add all the protocols.
1353         */
1354        for(i = 0; i < SOCK_ARRAY_SIZE; i++)
1355        {
1356            tcp_prot.sock_array[i] = NULL;
1357            udp_prot.sock_array[i] = NULL;
1358            raw_prot.sock_array[i] = NULL;
1359        }
1360        tcp_prot.inuse = 0;
1361        tcp_prot.highestinuse = 0;
1362        udp_prot.inuse = 0;
1363        udp_prot.highestinuse = 0;
1364        raw_prot.inuse = 0;
1365        raw_prot.highestinuse = 0;

1366        printk("IP Protocols: ");

    //将静态定义的传输层协议加入 inet_protos 数组中。
    //静态定义的传输层协议是由 inet_protocol_base 变量指向的一个 inet_protocol 结构类型
    //的队列。具体情况参考下文中说明。
1367        for(p = inet_protocol_base; p != NULL;)
1368        {
1369            struct inet_protocol *tmp = (struct inet_protocol *) p->next;
1370            inet_add_protocol(p);
1371            printk("%s%s",p->name,tmp?", ":"\n");
1372            p = tmp;
1373        }
1374        /*
1375         *  Set the ARP module up
1376         */
1377        arp_init();
1378        /*
1379         *  Set the IP module up
1380         */
1381        ip_init();
```

```
1382    }
```

函数最后调用 arp_init,ip_init 函数对地址解析层和 IP 层进行初始化。在下文中分析到对应文件中将对这两个函数进行说明。

以下代码片断摘自 protocol.c 文件。tcp_protocl 变量定义了 TCP 协议的正常以及错误数据包接收函数。其它类似。网络层函数在处理完本层需要的工作后，将根据数据包使用的具体传输层协议查询 inet_protos 数组，从而将数据包传送给对应的接收函数进行上层的处理。

```
/*摘自 net/inet/protocol.c 文件*/
44  static struct inet_protocol tcp_protocol = {
45      tcp_rcv,        /* TCP handler          */
46      NULL,           /* No fragment handler (and won't be for a long time) */
47      tcp_err,        /* TCP error control    */
48      NULL,           /* next                 */
49      IPPROTO_TCP,    /* protocol ID          */
50      0,          /* copy                 */
51      NULL,           /* data                 */
52      "TCP"           /* name                 */
53  };


54  static struct inet_protocol udp_protocol = {
55      udp_rcv,        /* UDP handler          */
56      NULL,           /* Will be UDP fraglist handler */
57      udp_err,        /* UDP error control    */
58      &tcp_protocol,  /* next                 */
59      IPPROTO_UDP,    /* protocol ID          */
60      0,          /* copy                 */
61      NULL,           /* data                 */
62      "UDP"           /* name                 */
63  };


64  static struct inet_protocol icmp_protocol = {
65      icmp_rcv,       /* ICMP handler         */
66      NULL,           /* ICMP never fragments anyway */
67      NULL,           /* ICMP error control   */
68      &udp_protocol,  /* next                 */
69      IPPROTO_ICMP,       /* protocol ID      */
70      0,          /* copy                 */
71      NULL,           /* data                 */
72      "ICMP"          /* name                 */
73  };
```

```
74  #ifndef CONFIG_IP_MULTICAST
75  struct inet_protocol *inet_protocol_base = &icmp_protocol;
76  #else
77  static struct inet_protocol igmp_protocol = {
78      igmp_rcv,          /* IGMP handler            */
79      NULL,              /* IGMP never fragments anyway */
80      NULL,              /* IGMP error control     */
81      &icmp_protocol,/* next                    */
82      IPPROTO_IGMP,          /* protocol ID         */
83      0,             /* copy                    */
84      NULL,              /* data                    */
85      "IGMP"             /* name                    */
86  };

87  struct inet_protocol *inet_protocol_base = &igmp_protocol;
88  #endif

89  struct inet_protocol *inet_protos[MAX_INET_PROTOS] = {
90      NULL
91  };
```

af_inet.c 文件小结

af_inet.c 文件作为 INET 层处理函数定义文件，处理来自 BSD 层的请求，并在完成本层相应的检查工作后继续将请求发送给下层传输层函数进行具体的处理。这种分层实现方式以及实现中函数指针的使用使得程序具有极大的伸缩性和可扩展性。程序功能的模块化和层次化是网络代码的基本特点。由于 INET 层还只是作为请求的过渡层，并不进行请求的实际处理，所以该层实现函数相对较为简单，在下文中我们将逐渐深入网络代码的实质部分。为了保持层次的连贯性，接下来即以 TCP 协议实现文件 tcp.c 为分析对象分析传输层函数的实现。

2.4 net/inet/tcp.c 文件

tcp.c 文件应该是网络栈实现中代码最长的文件，达 5 千多行，这也从一方面验证了 TCP 协议的相对复杂性。TCP 协议提供一种面向连接的可靠性数据传输。所谓面向连接只是可靠性的一个方面。为了保证数据传输的可靠性，使用 TCP 协议的套接字在正式传送数据前必须首先建立与远端的连接通道。当然由于数据传输是经由传输线（如双绞线，同轴电缆等），所以这种连接事实上是一种虚拟通道，所谓的建立连接更准确的说应该是通信双方需要在传输数据之前交换某种重要信息。而这种信息的预先交换则是保证可靠性的基本条件。如 UDP 协议无需可靠新保证，其也就无须在传送数据报文之前建立连接。从 TCP 协议保证可靠性数据传输的实现来看，超时重传，数据应答，序列号这三者是实现可靠性的基本保证。而连接建立过程中一个最为重要信息的交换即是双方序列号的交换。序列号被用于 TCP 通信中的整个过程，双方传送的每个数据都有一个序列号对应。而为解决传输线路的不稳定性所造成的数据包丢失问题，TCP 协议使用发送方超时重传和接收方数据应答的策略。具体情况请参考附录 A-TCP 协议可靠性数据传输底层实现方法分析。建议读者在继续下文之前，先预读附录 A，这样将有助于下文中对于代码的

理解。此处需要提及的几点是：

1> 数据应答是累积的，不可进行跨越式应答。
所谓“不可进行跨越式应答”是指，如果接收到的数据存在“空洞”，则发送应答序列号应当是对这些“空洞”中数据的请求，而不能跨越这个“空洞”，对空洞之后的数据进行应答，这样会造成数据的真正丢失。由数据“空洞”衍生的其它议题如快速重传/恢复机制。

2> 流量控制是 TCP 协议一个重要方面。
流量控制即控制传输线路中传输的数据包总数。有时由于数据包的暂时延迟造成发送方超时重传和接收方启动快速重传机制（启动是自动进行的，通过不断的发送具有相同序列号的应答数据包完成），从而造成线路中充斥着大量数据包，而这些大量进入的数据包又进一步减低线路性能，而接收方和发送方又不断地发送数据包到线路上，从而陷入恶性循环。流量控制是避免此类问题而设计的。在实现上，一方面通过使用“窗口”，另一方面通过使用一些算法来完成。在下文中分析到相应函数时将进行具体的阐述。

使用 TCP 协议进行通信的套接字在不同阶段根据接收数据包内容将处于不同的状态，形成了 TCP 协议中特有的状态机。故在具体分析 tcp.c 文件中定义函数之前，首先简单介绍一下使用 TCP 协议的套接字所经过的各种不同状态(虽然前文中已有相关阐述)。
以下各种状态的中文解释为直译，一般我们在讨论 TCP 状态时，以英文单词为准以避免误解。
TCP_CLOSED
关闭状态：一个新建的 TCP 套接字起始将处于该状态。

TCP_LISTEN
监听状态：一般服务器端套接字在调用 listen 系统调用后即处于该状态。

TCP_SYN_SENT
同步信号已发送状态：该状态指客户端发送 SYN（建立连接的同步）数据包后所处的状态。在接收到远端服务器端应答后，即从该状态进入 TCP_ESTABLISHED 状态。

TCP_SYN_RECEIVED
同步信号已接收状态：服务器端在接收到远端客户端 SYN 数据包后，进行相应的处理（创建通信套接字等)后，发送应答数据包，并将新创建的通信套接字状态设置为 TCP_SYN_RECEIVED，在接收到客户端的应答后，即进入 TCP_ESTABLISHED 状态。

TCP_ESTABLISHED
连接建立状态，这是双方进行正常的数据传送所处的状态。

TCP_FIN_WAIT_1
本地发送 FIN（用于结束连接的）数据包后即进入该状态，等待对方的应答。一般一端发送完其所要发送的数据后，即可发送 FIN 数据包，此时发送通道被关闭，但仍可以继续接收远端发送的数据包。在接收到远端发送的对于 FIN 数据包的应答后，将进入 TCP_FIN_WAIT_2 状态。

TCP_FIN_WAIT_2
进入该状态表示本地已接收到远端发送的对于本地之前发送的 FIN 数据包的应答。进入该状态后，本地仍然可以继续接收远端发送给本地的数据包。在接收到远端发送的 FIN 数据包后（表示远端也已经发送完数据），本地将发送一个应答数据包，并进入 TCP_TIME_WAIT 状态。

TCP_TIME_WAIT 状态存在的时间被称为 2MSL 时间，这一方面是为避免本地发送的应答数据包丢失，另一方面避免一个新创建的套接字接收到旧套接字中遗留的数据包。

TCP_TIME_WAIT
该状态被称为 2MSL 等待状态。如果在此期间接收到远端发送的 FIN 数据包，则表示之前在 TCP_FIN_WAIT_2 状态发送的 ACK 应答数据包在传输中丢失或者长时间被延迟，从而造成远端重新发送了 FIN 数据包，此时重发 ACK 应答数据包。一旦 2MSL 时间到期，则将进入 TCP_CLOSED 状态，即完成关闭操作。

TCP_CLOSE_WAIT
该状态存在于后关闭的一端。当接收到远端发送的 FIN 数据包后，本地发送一个 ACK 应答数据包，并将套接字状态从 TCP_ESTABLISHED 设置为 TCP_CLOSE_WAIT。本地可继续向远端发送数据包，在发送完所有数据后，本地将发送一个 FIN 数据包关闭本地发送通道，并将状态设置为 TCP_LAST_ACK 状态，等待远端对 FIN 数据包的应答数据包。

TCP_CLOSING
如果通信双方同时发送 FIN 数据包，即同时进行关闭操作，则双方将同时进入 TCP_CLOSING 状态。具体的，本地发送一个 FIN 数据包以结束本地数据包发送，如果在等待应答期间，接收到远端发送的 FIN 数据包，则本地将状态设置为 TCP_CLOSING 状态。在接收到应答后，再继续转入到 TCP_CLOSE_WAIT 状态。

TCP_LAST_ACK
作为后关闭的一方，在发送 FIN 数据包后，即进入 TCP_LAST_ACK 状态。此时等待远端发送应答数据包，在接收到应答数据包后，即完成关闭操作，进入 TCP_CLOSED 状态。

以下进入 tcp.c 文件分析。

```
1     /*
2      * INET        An implementation of the TCP/IP protocol suite for the LINUX
3      *        operating system.    INET is implemented using the    BSD Socket
4      *        interface as the means of communication with the user level.
5      *
6      *        Implementation of the Transmission Control Protocol(TCP).
7      *
8      * Version:     @(#)tcp.c       1.0.16      05/25/93
9      *
10     * Authors:     Ross Biro, <bir7@leland.Stanford.Edu>
11     *        Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
12     *        Mark Evans, <evansmp@uhura.aston.ac.uk>
13     *        Corey Minyard <wf-rch!minyard@relay.EU.net>
14     *        Florian La Roche, <flla@stud.uni-sb.de>
15     *        Charles Hedrick, <hedrick@klinzhai.rutgers.edu>
16     *        Linus Torvalds, <torvalds@cs.helsinki.fi>
17     *        Alan Cox, <gw4pts@gw4pts.ampr.org>
```

```
18    *        Matthew Dillon, <dillon@apollo.west.oic.com>
19    *        Arnt Gulbrandsen, <agulbra@no.unit.nvg>
20    *
21    * Fixes:
22    *        Alan Cox:    Numerous verify_area() calls
23    *        Alan Cox:    Set the ACK bit on a reset
24    *        Alan Cox:    Stopped it crashing if it closed while sk->inuse=1
25    *                     and was trying to connect (tcp_err()).
26    *        Alan Cox:    All icmp error handling was broken
27    *                     pointers passed where wrong and the
28    *                     socket was looked up backwards. Nobody
29    *                     tested any icmp error code obviously.
30    *        Alan Cox:    tcp_err() now handled properly. It wakes people
31    *                     on errors. select behaves and the icmp error race
32    *                     has gone by moving it into sock.c
33    *        Alan Cox:    tcp_reset() fixed to work for everything not just
34    *                     packets for unknown sockets.
35    *        Alan Cox:    tcp option processing.
36    *        Alan Cox:    Reset tweaked (still not 100%) [Had syn rule wrong]
37    *        Herp Rosmanith    : More reset fixes
38    *        Alan Cox:    No longer acks invalid rst frames. Acking
39    *                     any kind of RST is right out.
40    *        Alan Cox:    Sets an ignore me flag on an rst receive
41    *                     otherwise odd bits of prattle escape still
42    *        Alan Cox:    Fixed another acking RST frame bug. Should stop
43    *                     LAN workplace lockups.
44    *        Alan Cox:    Some tidyups using the new skb list facilities
45    *        Alan Cox:    sk->keepopen now seems to work
46    *        Alan Cox:    Pulls options out correctly on accepts
47    *        Alan Cox:    Fixed assorted sk->rqueue->next errors
48    *        Alan Cox:    PSH doesn't end a TCP read. Switched a bit to skb ops.
49    *        Alan Cox:    Tidied tcp_data to avoid a potential nasty.
50    *        Alan Cox:    Added some better commenting, as the tcp is hard to follow
51    *        Alan Cox:    Removed incorrect check for 20 * psh
52    *    Michael O'Reilly    :    ack < copied bug fix.
53    *    Johannes Stille        :    Misc tcp fixes (not all in yet).
54    *        Alan Cox:    FIN with no memory -> CRASH
55    *        Alan Cox:    Added socket option proto entries. Also added awareness of them to accept.
56    *        Alan Cox:    Added TCP options (SOL_TCP)
57    *        Alan Cox:    Switched wakeup calls to callbacks, so the kernel can layer network
sockets.
58    *        Alan Cox:    Use ip_tos/ip_ttl settings.
59    *        Alan Cox:    Handle FIN (more) properly (we hope).
60    *        Alan Cox:    RST frames sent on unsynchronised state ack error/
```

```
61    *        Alan Cox:    Put in missing check for SYN bit.
62    *        Alan Cox:    Added tcp_select_window() aka NET2E
63    *                     window non shrink trick.
64    *        Alan Cox:    Added a couple of small NET2E timer fixes
65    *        Charles Hedrick :  TCP fixes
66    *        Toomas Tamm:    TCP window fixes
67    *        Alan Cox:    Small URG fix to rlogin ^C ack fight
68    *        Charles Hedrick    :    Rewrote most of it to actually work
69    *        Linus        :    Rewrote tcp_read() and URG handling
70    *                     completely
71    *        Gerhard Koerting:  Fixed some missing timer handling
72    *        Matthew Dillon   : Reworked TCP machine states as per RFC
73    *        Gerhard Koerting:  PC/TCP workarounds
74    *        Adam Caldwell    :    Assorted timer/timing errors
75    *        Matthew Dillon   :    Fixed another RST bug
76    *        Alan Cox:    Move to kernel side addressing changes.
77    *        Alan Cox:    Beginning work on TCP fastpathing (not yet usable)
78    *        Arnt Gulbrandsen:  Turbocharged tcp_check() routine.
79    *        Alan Cox:    TCP fast path debugging
80    *        Alan Cox:    Window clamping
81    *        Michael Riepe:    Bug in tcp_check()
82    *        Matt Dillon    :    More TCP improvements and RST bug fixes
83    *        Matt Dillon    :    Yet more small nasties remove from the TCP code
84    *                     (Be very nice to this man if tcp finally works 100%) 8)
85    *        Alan Cox:    BSD accept semantics.
86    *        Alan Cox:    Reset on closedown bug.
87    *    Peter De Schrijver    :    ENOTCONN check missing in tcp_sendto().
88    *        Michael Pall    :    Handle select() after URG properly in all cases.
89    *        Michael Pall    :    Undo the last fix in tcp_read_urg() (multi URG PUSH broke
rlogin).
90    *        Michael Pall    :    Fix the multi URG PUSH problem in tcp_readable(), select() after
URG works now.
91    *        Michael Pall    :    recv(...,MSG_OOB) never blocks in the BSD api.
92    *        Alan Cox:    Changed the semantics of sk->socket to
93    *                     fix a race and a signal problem with
94    *                     accept() and async I/O.
95    *        Alan Cox:    Relaxed the rules on tcp_sendto().
96    *        Yury Shevchuk    :    Really fixed accept() blocking problem.
97    *        Craig I. Hagan    :    Allow for BSD compatible TIME_WAIT for
98    *                     clients/servers which listen in on
99    *                     fixed ports.
100   *        Alan Cox:    Cleaned the above up and shrank it to
101   *                     a sensible code size.
102   *        Alan Cox:    Self connect lockup fix.
```

```
103  *      Alan Cox:       No connect to multicast.
104  *      Ross Biro:      Close unaccepted children on master
105  *                      socket close.
106  *      Alan Cox:       Reset tracing code.
107  *      Alan Cox:       Spurious resets on shutdown.
108  *      Alan Cox:       Giant 15 minute/60 second timer error
109  *      Alan Cox:       Small whoops in selecting before an accept.
110  *      Alan Cox:       Kept the state trace facility since it's
111  *                      handy for debugging.
112  *      Alan Cox:       More reset handler fixes.
113  *      Alan Cox:       Started rewriting the code based on the RFC's
114  *                      for other useful protocol references see:
115  *                      Comer, KA9Q NOS, and for a reference on the
116  *                      difference between specifications and how BSD
117  *                      works see the 4.4lite source.
118  *      A.N.Kuznetsov   :       Don't time wait on completion of tidy
119  *                      close.
120  *      Linus Torvalds: Fin/Shutdown & copied_seq changes.
121  *      Linus Torvalds: Fixed BSD port reuse to work first syn
122  *      Alan Cox:       Reimplemented timers as per the RFC and using multiple
123  *                      timers for sanity.
124  *      Alan Cox:       Small bug fixes, and a lot of new
125  *                      comments.
126  *      Alan Cox:       Fixed dual reader crash by locking
127  *                      the buffers (much like datagram.c)
128  *      Alan Cox:       Fixed stuck sockets in probe. A probe
129  *                      now gets fed up of retrying without
130  *                      (even a no space) answer.
131  *      Alan Cox:       Extracted closing code better
132  *      Alan Cox:       Fixed the closing state machine to
133  *                      resemble the RFC.
134  *      Alan Cox:       More 'per spec' fixes.
135  *      Alan Cox:       tcp_data() doesn't ack illegal PSH
136  *                      only frames. At least one pc tcp stack
137  *                      generates them.
138  *
139  *
140  * To Fix:
141  *      Fast path the code. Two things here - fix the window calculation
142  *      so it doesn't iterate over the queue, also spot packets with no funny
143  *      options arriving in order and process directly.
144  *
145  *      Implement RFC 1191 [Path MTU discovery]
146  *      Look at the effect of implementing RFC 1337 suggestions and their impact.
```

```
147 *       Rewrite output state machine to use a single queue and do low window
148 *       situations as per the spec (RFC 1122)
149 *       Speed up input assembly algorithm.
150 *       RFC1323 - PAWS and window scaling. PAWS is required for IPv6 so we
151 *       could do with it working on IPv4
152 *       User settable/learned rtt/max window/mtu
153 *       Cope with MTU/device switches when retransmitting in tcp.
154 *       Fix the window handling to use PR's new code.
155 *
156 *       Change the fundamental structure to a single send queue maintained
157 *       by TCP (removing the bogus ip stuff [thus fixing mtu drops on
158 *       active routes too]). Cut the queue off in tcp_retransmit/
159 *       tcp_transmit.
160 *       Change the receive queue to assemble as it goes. This lets us
161 *       dispose of most of tcp_sequence, half of tcp_ack and chunks of
162 *       tcp_data/tcp_read as well as the window shrink crud.
163 *       Separate out duplicated code - tcp_alloc_skb, tcp_build_ack
164 *       tcp_queue_skb seem obvious routines to extract.
165 *
166 *       This program is free software; you can redistribute it and/or
167 *       modify it under the terms of the GNU General Public License
168 *       as published by the Free Software Foundation; either version
169 *       2 of the License, or(at your option) any later version.
170 *
171 * Description of States:
172 *
173 *   TCP_SYN_SENT        sent a connection request, waiting for ack
174 *
175 *   TCP_SYN_RECV        received a connection request, sent ack,
176 *                       waiting for final ack in three-way handshake.
177 *
178 *   TCP_ESTABLISHED         connection established
179 *
180 *   TCP_FIN_WAIT1       our side has shutdown, waiting to complete
181 *                       transmission of remaining buffered data
182 *
183 *   TCP_FIN_WAIT2       all buffered data sent, waiting for remote
184 *                       to shutdown
185 *
186 *   TCP_CLOSING         both sides have shutdown but we still have
187 *                       data we have to finish sending
188 *
189 *   TCP_TIME_WAIT       timeout to catch resent junk before entering
190 *                       closed, can only be entered from FIN_WAIT2
```

```
191   *                       or CLOSING.  Required because the other end
192   *                       may not have gotten our last ACK causing it
193   *                       to retransmit the data packet (which we ignore)
194   *
195   *     TCP_CLOSE_WAIT            remote side has shutdown and is waiting for
196   *                       us to finish writing our data and to shutdown
197   *                       (we have to close() to move on to LAST_ACK)
198   *
199   *     TCP_LAST_ACK        out side has shutdown after remote has
200   *                       shutdown.  There may still be data in our
201   *                       buffer that we have to finish sending
202   *
203   *     TCP_CLOSE         socket is finished
204   */

205  #include <linux/types.h>
206  #include <linux/sched.h>
207  #include <linux/mm.h>
208  #include <linux/time.h>
209  #include <linux/string.h>
210  #include <linux/config.h>
211  #include <linux/socket.h>
212  #include <linux/sockios.h>
213  #include <linux/termios.h>
214  #include <linux/in.h>
215  #include <linux/fcntl.h>
216  #include <linux/inet.h>
217  #include <linux/netdevice.h>
218  #include "snmp.h"
219  #include "ip.h"
220  #include "protocol.h"
221  #include "icmp.h"
222  #include "tcp.h"
223  #include "arp.h"
224  #include <linux/skbuff.h>
225  #include "sock.h"
226  #include "route.h"
227  #include <linux/errno.h>
228  #include <linux/timer.h>
229  #include <asm/system.h>
230  #include <asm/segment.h>
231  #include <linux/mm.h>

232  /*
```

```
233   *    The MSL timer is the 'normal' timer.
234   */

235  #define reset_msl_timer(x,y,z)       reset_timer(x,y,z)

236  #define SEQ_TICK 3
237  unsigned long seq_offset;
238  struct tcp_mib tcp_statistics;

239  static void tcp_close(struct sock *sk, int timeout);
```

注意到此处的 seq_offset 变量，在 af_inet.c 文件有对此变量的引用，并在 inet_proto_init 函数中对该变量进行了初始化。tcp_statistics 是一个 tcp_mib 结构类型，是对 RFC2233 的信息统计实现。另外是对 tcp_close 和 reset_msl_timer 的声明或定义。reset_timer 函数用一个新的值重新设置定时器。

```
240  /*
241   *    The less said about this the better, but it works and will do for 1.2
242   */

243  static struct wait_queue *master_select_wakeup;
```

等待队列定义。等待队列用于进程在等待某个条件时暂时休眠之地。wait_queue 结构定义如下：

```
// linux/wait.h
7     struct wait_queue {
8             struct task_struct * task;
9             struct wait_queue * next;
10    };

244  static __inline__ int min(unsigned int a, unsigned int b)
245  {
246          if (a < b)
247                  return(a);
248          return(b);
249  }

250  #undef STATE_TRACE

251  #ifdef STATE_TRACE
252  static char *statename[]={
253          "Unused","Established","Syn Sent","Syn Recv",
254          "Fin Wait 1","Fin Wait 2","Time Wait", "Close",
255          "Close Wait","Last ACK","Listen","Closing"
256  };
257  #endif
```

```
258 static __inline__ void tcp_set_state(struct sock *sk, int state)
259 {
260         if(sk->state==TCP_ESTABLISHED)
261                 tcp_statistics.TcpCurrEstab--;
262 #ifdef STATE_TRACE
263         if(sk->debug)
264                 printk("TCP sk=%p, State %s -> %s\n",sk, statename[sk->state],statename[state]);
265 #endif
266         /* This is a hack but it doesn't occur often and it's going to
267                 be a real               to fix nicely */

268         if(state==TCP_ESTABLISHED && sk->state==TCP_SYN_RECV)
269         {
270                 wake_up_interruptible(&master_select_wakeup);
271         }
272         sk->state=state;
273         if(state==TCP_ESTABLISHED)
274                 tcp_statistics.TcpCurrEstab++;
275 }
```

tcp_set_state 函数用于设置套接字状态，参数 state 为新的套接字状态。注意函数实现的开头和结尾形成的对称，从而保证统计信息的准确性。如果新的状态为 TCP_ESTABLISHED 并且原来状态为 TCP_SYN_RECV，表示套接字完成连接。从代码的实现来看，此时将睡眠队列 master_select_wakeup 等待的任务唤醒。由于之前的状态为 TCP_SYN_RECV，所以这种行为一般发生在服务器端，如果从服务器的角度来分析则不难理解。对于服务器端套接字，只有在调用 accept 函数后，才进行真正的通信。如果在调用 accept 函数之前，没有客户端请求连接，则服务器端进行 accept 函数调用的任务将等待，即被挂入 master_select_wakeup 队列中。此处在一个套接字完成建立后，accept 函数即可返回，此时就需要唤醒 master_select_wakeup 队列中的任务。

```
276 /*
277  *  This routine picks a TCP windows for a socket based on
278  *  the following constraints
279  *
280  *  1. The window can never be shrunk once it is offered (RFC 793)
281  *  2. We limit memory per socket
282  *
283  *  For now we use NET2E3's heuristic of offering half the memory
284  *  we have handy. All is not as bad as this seems however because
285  *  of two things. Firstly we will bin packets even within the window
286  *  in order to get the data we are waiting for into the memory limit.
287  *  Secondly we bin common duplicate forms at receive time
288  *      Better heuristics welcome
289  */
```

```
290  int tcp_select_window(struct sock *sk)
291  {
292          int new_window = sk->prot->rspace(sk);

293          if(sk->window_clamp)
294                  new_window=min(sk->window_clamp,new_window);
295          /*
296           *      Two things are going on here.  First, we don't ever offer a
297           *      window less than min(sk->mss, MAX_WINDOW/2).  This is the
298           *      receiver side of SWS as specified in RFC1122.
299           *      Second, we always give them at least the window they
300           *      had before, in order to avoid retracting window.  This
301           *      is technically allowed, but RFC1122 advises against it and
302           *      in practice it causes trouble.
303           *
304           *      Fixme: This doesn't correctly handle the case where
305           *      new_window > sk->window but not by enough to allow for the
306           *      shift in sequence space.
307           */
308          if (new_window < min(sk->mss, MAX_WINDOW/2) || new_window < sk->window)
309                  return(sk->window);
310          return(new_window);
311  }
```

tcp_select_window 窗口选择函数。所谓窗口即对所接收数据包数量的一种限制。在本地发送的每个数据包中 TCP 首部都会包含一个本地声明的窗口大小，远端应节制其数据包发送不可超过本地通报的窗口大小。窗口大小以字节数为单位。窗口大小的设置需要考虑到以下两个因素：
1> RFC793 文档（TCP 协议文档）强烈推荐不可降低窗口大小值。
2> 窗口大小的设置应考虑到本地接收缓冲区的大小。

函数实现首先调用 sk->prot->rspace 指向的函数（sock_rspace）查看接收缓冲区空闲区域的大小。之后检查 sock 结构 window_clamp 字段是否为 0，如不为 0，则表示本地进行窗口节制，此时取节制值与空闲缓冲区大小中之较小值。然后进一步检查这个较小值是否小于最大报文长度值（一般 MAX_WINDOW/2>MSS）,如是，则返回原窗口值。这是优化数据包传送的一种方式。MSS 表示最大报文长度值，如果本地通报给远端的窗口值小于 MSS 值，则会造成小数据包的传送，大量小数据包充斥传输线路，对于充分利用传送线路不利，所以如果新的窗口值小于 MSS 值，则继续通报旧的窗口值。另外 RFC793 要求避免使用降低窗口大小策略，所以如果新的窗口值小于原有的窗口值，则仍然使用原窗口值进行通报。当这些条件都不成立时（即新窗口值大于 MSS 值，且大于原窗口值），方才通报新的窗口值给远端。

```
312  /*
313   *      Find someone to 'accept'. Must be called with
314   *      sk->inuse=1 or cli()
```

```
315   */

/* 对于监听 socket 而言，其接收队列中的数据包是建立连接数据包，即 SYN 数据包，
 * 不含数据数据包，数据的处理是另外的 socket 负责。
 */
316  static struct sk_buff *tcp_find_established(struct sock *s)
317  {
318          struct sk_buff *p=skb_peek(&s->receive_queue);
319          if(p==NULL)
320                  return NULL;
321          do
322          {
323                  if(p->sk->state == TCP_ESTABLISHED || p->sk->state >= TCP_FIN_WAIT1)
324                          return p;
325                  p=p->next;
326          }
327          while(p!=(struct sk_buff *)&s->receive_queue);
328          return NULL;
329  }
```

tcp_find_established 函数的作用是从监听套接字缓冲队列中检查是否存在已经完成连接的远端发送的数据包，该数据包的作用即完成连接。本地监听套接字在处理完该连接，设置相关状态后将该数据包缓存在其队列中。诚如函数前注释所述，对于监听套接字而言，其缓冲队列中缓存的均是连接请求数据包（或者是已经完成连接请求的数据包，由 tcp_conn_request 函数经过处理后缓存到该队列中），对于 accept 系统调用，底层将最终调用此处的 tcp_find_established 函数查看该队列中是否有已经完成连接的数据包，如有，则返回该数据包，由调用者（tcp_dequeue_established，该函数又被 tcp_accept 函数调用）继续处理。读者需要结合下文中对 tcp_conn_request, tcp_dequeue_established, tcp_accept 函数介绍理解该函数作用。

```
330  /*
331   *    Remove a completed connection and return it. This is used by
332   *    tcp_accept() to get connections from the queue.
333   */

334  static struct sk_buff *tcp_dequeue_established(struct sock *s)
335  {
336          struct sk_buff *skb;
337          unsigned long flags;
338          save_flags(flags);
339          cli();
340          skb=tcp_find_established(s);
341          if(skb!=NULL)
342                  skb_unlink(skb);    /* Take it off the queue */
343          restore_flags(flags);
344          return skb;
```

```
345 }
```

tcp_dequeue_established 函数被 tcp_accept 函数调用，而该函数则调用 tcp_find_established 函数查看监听套接字接收队列中是否有已经建立连接的数据包，如果则返回该数据包，有 tcp_accept 函数进行进一步的处理后，返回给客户端本地用于数据传输的套接字。下面给出 tcp_accept 函数实现，展示 accept 系统调用底层处理的流程。

用户调用 accept 函数，由 BSD 层 sock_accept 函数处理，该函数调用 INET 层 inet_accept 函数进行处理，对于 TCP 协议而言，inet_accept 函数则调用 tcp_accept 函数完成具体的任务。众所周知，accept 函数将返回用于通信的另一个套接字（不是监听套接字）描述符。tcp_accept 函数将返回表示该通信套接字的 sock 结构，inet_accept 函数返回值表示下层（及其自身）处理是否成功，返回 0 表示成功，此时 sock_accept 将分配一个未使用的文件描述符返回给用户应用程序。

```
3638    /*
3639     *  This will accept the next outstanding connection.
3640     */

3641    static struct sock *tcp_accept(struct sock *sk, int flags)
3642    {
3643        struct sock *newsk;
3644        struct sk_buff *skb;

3645      /*
3646       * We need to make sure that this socket is listening,
3647       * and that it has something pending.
3648       */

3649        if (sk->state != TCP_LISTEN)
3650        {
3651            sk->err = EINVAL;
3652            return(NULL);
3653        }

3654        /* Avoid the race. */
3655        cli();
3656        sk->inuse = 1;

3657        while((skb = tcp_dequeue_established(sk)) == NULL)
3658        {
3659            if (flags & O_NONBLOCK)
3660            {
3661                sti();
3662                release_sock(sk);
3663                sk->err = EAGAIN;
3664                return(NULL);
3665            }
```

```
3666                release_sock(sk);
3667                interruptible_sleep_on(sk->sleep);
3668                if (current->signal & ~current->blocked)
3669                {
3670                    sti();
3671                    sk->err = ERESTARTSYS;
3672                    return(NULL);
3673                }
3674                sk->inuse = 1;
3675            }
3676            sti();

3677            /*
3678             *  Now all we need to do is return skb->sk.
3679             */

3680            newsk = skb->sk;

3681            kfree_skb(skb, FREE_READ);
3682            sk->ack_backlog--;
3683            release_sock(sk);
3684            return(newsk);
3685        }
```

tcp_accept 函数完成的工作很直接，对于调用 accept 的套接字而言，其首先必须是一个监听套接字，否则错误返回。其后调用 tcp_dequeue_established 函数检查监听套接字接收队列，看是否存在已经完成连接的数据包，如有，则返回用于此连接通信的本地套接字，该通信套接字在 tcp_conn_request 函数中处理连接请求时被建立，并与该请求连接数据包相联系起来，从而此处只需返回该套接字即可：

```
newsk=skb->sk;
……
return (newsk);
```

如果对于查看的监听套接字尚无完成建立的数据包存在（tcp_dequeue_established 函数返回 NULL），则如果此监听套接字之前设置了 NON_BLOCK 选项，则直接返回 NULL；否则调用 interruptible_sleep_on 函数等待睡眠（即用户应用程序即表现为阻塞于 accept 系统调用）。

```
346 /*
347  *  This routine closes sockets which have been at least partially
348  *  opened, but not yet accepted. Currently it is only called by
349  *  tcp_close, and timeout mirrors the value there.
350  */
```

```
/* 关闭一个监听 socket，对于半连接状态的 socket（即对方发送了 SYN 数据包，己方也进
* 行了处理，但己方尚未调用 accept 建立完全连接）逐一进行关闭操作。
 */
351  static void tcp_close_pending (struct sock *sk)
352  {
353         struct sk_buff *skb;

354         while ((skb = skb_dequeue(&sk->receive_queue)) != NULL)
355         {
356                 skb->sk->dead=1;
357                 tcp_close(skb->sk, 0);
358                 kfree_skb(skb, FREE_READ);
359         }
360         return;
361  }
```

tcp_close_pending 函数关闭监听套接字接收队列中所有请求连接的通信端。关闭操作处理发送 FIN 数据包给远端对应通信端外，还需要释放之前创建的用于本地通信的套接字。这通过将该套接字状态设置为 dead，并调用 tcp_close 函数完成，最后释放缓存的请求数据包：
kfree_skb(skb, FREE_READ);

```
362  /*
363   *    Enter the time wait state.
364   */
/* 设置本地 socket 进入 TIME_WAIT 状态，并设置定时期 2MSL 等待时间。*/
365  static void tcp_time_wait(struct sock *sk)
366  {
367         tcp_set_state(sk,TCP_TIME_WAIT);
368         sk->shutdown = SHUTDOWN_MASK;
369         if (!sk->dead)
370                 sk->state_change(sk);
371         reset_msl_timer(sk, TIME_CLOSE, TCP_TIMEWAIT_LEN);
372  }
```

tcp_time_wait 函数设置套接字为 2MSL 等待状态，并启动相应定时器，用于超时处理。有关 TCP 协议中 TIME_WAIT 状态的作用请参阅前文对 TCP 协议的介绍。

```
373  /*
374   *    A socket has timed out on its send queue and wants to do a
375   *    little retransmitting. Currently this means TCP.
376   */

377  void tcp_do_retransmit(struct sock *sk, int all)
```

```
378 {
379         struct sk_buff * skb;
380         struct proto *prot;
381         struct device *dev;
382         int ct=0;

383         prot = sk->prot;
384         skb = sk->send_head;

385         while (skb != NULL)
386         {
387                 struct tcphdr *th;
388                 struct iphdr *iph;
389                 int size;

390                 dev = skb->dev;
391                 IS_SKB(skb);
392                 skb->when = jiffies;

393                 /*
394                  * In general it's OK just to use the old packet.    However we
395                  * need to use the current ack and window fields.    Urg and
396                  * urg_ptr could possibly stand to be updated as well, but we
397                  * don't keep the necessary data.    That shouldn't be a problem,
398                  * if the other end is doing the right thing.    Since we're
399                  * changing the packet, we have to issue a new IP identifier.
400                  */

401                 iph = (struct iphdr *)(skb->data + dev->hard_header_len);
402                 th = (struct tcphdr *)(((char *)iph) + (iph->ihl << 2));
403                 size = skb->len - (((unsigned char *) th) - skb->data);

404                 /*
405                  *    Note: We ought to check for window limits here but
406                  *    currently this is done (less efficiently) elsewhere.
407                  *    We do need to check for a route change but can't handle
408                  *    that until we have the new 1.3.x buffers in.
409                  *
410                  */
```

```
        /* 重发时，并非是重发原始的数据包，而是对数据包进行一些字段的调整后，
         * 将该数据包发送出去，注意，如果数据包在中间路由器被分片，而远端只是由
          * 于某个分片未接收到，
         * 本地重发时还是重发整个数据包，而非某个分片，因为无法确知哪个分片丢失。
         */
```

```
411             iph->id = htons(ip_id_count++);
412             ip_send_check(iph);

413             /*
414              *  This is not the right way to handle this. We have to
415              *  issue an up to date window and ack report with this
416              *  retransmit to keep the odd buggy tcp that relies on
417              *  the fact BSD does this happy.
418              *  We don't however need to recalculate the entire
419              *  checksum, so someone wanting a small problem to play
420              *  with might like to implement RFC1141/RFC1624 and speed
421              *  this up by avoiding a full checksum.
422              */

423             th->ack_seq = ntohl(sk->acked_seq);
424             th->window = ntohs(tcp_select_window(sk));
425             tcp_send_check(th, sk->saddr, sk->daddr, size, sk);

426             /*
427              *  If the interface is (still) up and running, kick it.
428              */

429             if (dev->flags & IFF_UP)
430             {
431                     /*
432                      *  If the packet is still being sent by the device/protocol
433                      *  below then don't retransmit. This is both needed, and good -
434                      *  especially with connected mode AX.25 where it stops resends
435                      *  occurring of an as yet unsent anyway frame!
436                      *  We still add up the counts as the round trip time wants
437                      *  adjusting.
438                      */
439                     if (sk && !skb_device_locked(skb))
440                     {
441                             /* Remove it from any existing driver queue first! */
442                             skb_unlink(skb);
443                             /* Now queue it */
444                             ip_statistics.IpOutRequests++;
445                             dev_queue_xmit(skb, dev, sk->priority);
446                     }
447             }

448             /*
449              *  Count retransmissions
```

```
450             */

451             ct++;
452             sk->prot->retransmits ++;

453             /*
454              *  Only one retransmit requested.
455              */

456             if (!all)
457                     break;

458             /*
459              *  This should cut it off before we send too many packets.
460              */

461             if (ct >= sk->cong_window)
462                     break;
463             skb = skb->link3;
464     }
465 }
```

tcp_do_retransmit 函数用于 TCP 协议数据包重传，该函数遍历相应套接字重传队列，依照要求重传该队列中数据包。数据包重传策略是 TCP 协议保证可靠性数据传输的必要手段，有关内容请参考附录 A。每个套接字对于其发送的每个数据包都会缓存其在重发队列中，该重发队列由 sock 结构中 send_head, send_tail 两个字段以及 sk_buff 结构中 link3 字段完成建立：send_head 指向队列首，send_tail 指向队列尾，而 link3 则用于队列中数据包之间相互连接。
需要注意的一点是，对于重发的数据包，其 IP 首部中用于标识数据包的 id 字段将被赋予新值，以区别于之前发送的数据包。只要数据包中 TCP 首部中本地序列号不发生改变，其表示的数据依然是原先发送的数据，不会在本地发生人为的数据包乱序问题。另外一个值得注意的地方是：该函数对于重发的数据包个数维护一个计数器，以便重发的数据包个数不超过对应套接字设定的节制窗口值（sk->cong_window），而如果参数 all 为 0，则表示只发送一个数据包后即可退出。

```
466 /*
467  *  Reset the retransmission timer
468  */

469 static void reset_xmit_timer(struct sock *sk, int why, unsigned long when)
470 {
471     del_timer(&sk->retransmit_timer);
472     sk->ip_xmit_timeout = why;
473     if((int)when < 0)
474     {
475         when=3;
```

```
476             printk("Error: Negative timer in xmit_timer\n");
477         }
478         sk->retransmit_timer.expires=when;
479         add_timer(&sk->retransmit_timer);
480 }
```

reset_xmit_timer 函数用于复位重传定时器，该定时器用于数据包定时重传操作，具体定时含义由 sock 结构 ip_xmit_timeout 字段指示，而到期执行函数为 retransmit_timer，这在 tcp_conn_request 函数中被初始化：

```
/*newsk 为新创建的用于通信的本地 sock 结构*/
init_timer(&newsk->retransmit_timer);
newsk->retransmit_timer.data = (unsigned long)newsk;
newsk->retransmit_timer.function=&retransmit_timer;
```

所谓复位定时器，是指首先将原有定时器从系统定时器队列中删除，更新定时时间间隔后，重新插入队列。sock 结构中 retransmit_timer 定时器用于普通 TCP 数据包超时重传，窗口探测（PROBE），保活（Keep Alive）数据包发送。

```
481 /*
482  *  This is the normal code called for timeouts.  It does the retransmission
483  *  and then does backoff.  tcp_do_retransmit is separated out because
484  *  tcp_ack needs to send stuff from the retransmit queue without
485  *  initiating a backoff.
486  */


487 void tcp_retransmit_time(struct sock *sk, int all)
488 {
489         tcp_do_retransmit(sk, all);

490         /*
491          * Increase the timeout each time we retransmit.  Note that
492          * we do not increase the rtt estimate.  rto is initialized
493          * from rtt, but increases here.  Jacobson (SIGCOMM 88) suggests
494          * that doubling rto each time is the least we can get away with.
495          * In KA9Q, Karn uses this for the first few times, and then
496          * goes to quadratic.  netBSD doubles, but only goes up to *64,
497          * and clamps at 1 to 64 sec afterwards.  Note that 120 sec is
498          * defined in the protocol as the maximum possible RTT.  I guess
499          * we'll have to use something other than TCP to talk to the
500          * University of Mars.
501          *
502          * PAWS allows us longer timeouts and large windows, so once
503          * implemented ftp to mars will work nicely. We will have to fix
```

```
504          * the 120 second clamps though!
505          */

506         sk->retransmits++;
507         sk->backoff++;
508         sk->rto = min(sk->rto << 1, 120*HZ);
509         reset_xmit_timer(sk, TIME_WRITE, sk->rto);
510 }
```

tcp_retransmit_time 函数一方面调用 tcp_do_retransmit 函数进行 TCP 数据包超时重传，另一方面调用 reset_xmit_timer 复位重传定时器，所依据的原理即 TCP 协议规范中表达的指数退避重传时间计算方式，即下一次数据包重传时间间隔为前一次的两倍，不过不可超过 2 分钟上限，当然这个上限值大多是实现相关的。

```
511 /*
512  *  A timer event has trigger a tcp retransmit timeout. The
513  *  socket xmit queue is ready and set up to send. Because
514  *  the ack receive code keeps the queue straight we do
515  *  nothing clever here.
516  */

517 static void tcp_retransmit(struct sock *sk, int all)
518 {
519         if (all)
520         {
521                 tcp_retransmit_time(sk, all);
522                 return;
523         }
    /*  发送端猜测拥塞发生的唯一方法是它必须重发一个报文段，重传发生在
     * 1>RTO 定时期超时
     * 2>收到三个 ACK 数据报

     * TCP 对拥塞发生时的处理：
     * 1>如超时重传
     * A. 慢启动门限值设置为当前窗口的一半
     * B. 将 CWND 设置为一个报文段，启动慢启动阶段
     */
524         sk->ssthresh = sk->cong_window >> 1; /* remember window where we lost */
525         /* sk->ssthresh in theory can be zero.  I guess that's OK */
526         sk->cong_count = 0;

527         sk->cong_window = 1;

528         /* Do the actual retransmit. */
```

```
529         tcp_retransmit_time(sk, all);
530 }
```

tcp_retransmit 函数是最上层的重传函数，重传定时器到期时，如果当前超时原因是由于 TCP 数据包在规定时间内未得到应答，则需要重传相关数据包，此时到期执行函数 retransmit_timer 将调用 tcp_retransmit 函数重传相关 TCP 数据包（即 sock 结构中由 send_head 指向的队列），而 tcp_retrasmit 函数则进一步调用 tcp_retransmit_time 函数处理重传工作，tcp_retransmit_time 则调用 tcp_do_retransmit 函数具体完成重传数据包的工作。如此逐层调用主要是分层进行各种信息的处理更新。如 tcp_do_retransmit 函数只负责从 send_head 队列中取数据包进行重传，而 tcp_retransmit_time 在调用 tcp_do_retransmit 的基础上，对重传时间间隔进行更新；最上层的 tcp_retransmit 函数在调用 tcp_retransmit_time 函数的基础上，判断造成此次重传的深层原因，如重传是由于连续收到三个应答数据包造成的，则 tcp_retransmit 需要进行拥塞处理。总之，以上这种层层调用的关系是为了分工更加细致，从而使层次更加分明。读者如继续研究内核代码实现，自然而然会发现，后期代码除了在功能上更全面之外，一个显著的特点是原来一个函数完成的工作被分为几个函数完成，如此可以独立出一种层次关系，使得代码更易于阅读和编写。

```
531 /*
532  *  A write timeout has occurred. Process the after effects.
533  */

534 static int tcp_write_timeout(struct sock *sk)
535 {
536         /*
537          *  Look for a 'soft' timeout.
538          */
539         if ((sk->state == TCP_ESTABLISHED && sk->retransmits && !(sk->retransmits & 7))
540                 || (sk->state != TCP_ESTABLISHED && sk->retransmits > TCP_RETR1))
541         {
542                 /*
543                  *  Attempt to recover if arp has changed (unlikely!) or
544                  *  a route has shifted (not supported prior to 1.3).
545                  */
546                 arp_destroy (sk->daddr, 0);
547                 ip_route_check (sk->daddr);
548         }
549         /*
550          *  Has it gone just too far ?
551          */
552         if (sk->retransmits > TCP_RETR2)
553         {
554                 sk->err = ETIMEDOUT;
555                 sk->error_report(sk);
556                 del_timer(&sk->retransmit_timer);
557                 /*
558                  *  Time wait the socket
```

```
559              */
560             if (sk->state == TCP_FIN_WAIT1 || sk->state == TCP_FIN_WAIT2 ||
                                        sk->state == TCP_CLOSING )
561             {
562                 tcp_set_state(sk,TCP_TIME_WAIT);
563                 reset_msl_timer (sk, TIME_CLOSE, TCP_TIMEWAIT_LEN);
564             }
565             else
566             {
567                 /*
568                  *  Clean up time.
569                  */
570                 tcp_set_state(sk, TCP_CLOSE);
571                 return 0;
572             }
573         }
574         return 1;
575 }
```

tcp_write_timeout 函数在重传定时器发生超时时被调用，诚如上文所述，有三种情况发生重传定时器超时：普通 TCP 数据包超时重传，远端窗口探测，保活探测。

tcp_write_timeout 函数将根据套接字当前状态以及超时次数对该套接字进行一些处理：

1） 如果套接字处于正常连接状态，并且超时次数是 8 的倍数，则重新核对通信远端 MAC 地址。核对工作是通过 ip_route_check 函数完成，当然在这之前需要首先将原先的 MAC 地址删除。

2） 如果套接字处于非连接状态，则如果超时次数在 7（TCP_RETR1）次以上,同上将重新校对远端 MAC 地址。

3） 如果检查到超时次数达到 15（TCP_RETR2）以上，则表示很可能发生某种错误：中间路由器坏后者远端主机突然掉电灯等。此时：

A． 如果套接字状态为 TCP_FIN_WAIT1，TCP_FIN_WAIT2，TCP_CLOSING，则设置套接字状态为 TCP_CLOSE,并设置 2MSL 等待时间。

B． 否则直接将套接字状态设置为 TCP_CLOSE,无须等待 2MSL 时间。

C． 有关 TCP 协议各种状态说明在本书前文介绍 TCP 协议时有详细解释，此处不再论述。

```
576 /*
577  *  The TCP retransmit timer. This lacks a few small details.
578  *
579  *  1.  An initial rtt timeout on the probe0 should cause what we can
580  *      of the first write queue buffer to be split and sent.
581  *  2.  On a 'major timeout' as defined by RFC1122 we shouldn't report
582  *      ETIMEDOUT if we know an additional 'soft' error caused this.
583  *      tcp_err should save a 'soft error' for us.
584  */

585 static void retransmit_timer(unsigned long data)
```

```
586 {
587     struct sock *sk = (struct sock*)data;
588     int why = sk->ip_xmit_timeout;

589     /*
590      * only process if socket is not in use
591      */

592     cli();
593     if (sk->inuse || in_bh)
594     {
595         /* Try again in 1 second */
596         sk->retransmit_timer.expires = HZ;
597         add_timer(&sk->retransmit_timer);
598         sti();
599         return;
600     }

601     sk->inuse = 1;
602     sti();

603     /* Always see if we need to send an ack. */

604     if (sk->ack_backlog && !sk->zapped)
605     {
606         sk->prot->read_wakeup (sk);
607         if (! sk->dead)
608             sk->data_ready(sk,0);
609     }

610     /* Now we need to figure out why the socket was on the timer. */

611     switch (why)
612     {
613         /* Window probing */
614         case TIME_PROBE0:
615             tcp_send_probe0(sk);
616             tcp_write_timeout(sk);
617             break;
618         /* Retransmitting */
619         case TIME_WRITE:
620             /* It could be we got here because we needed to send an ack.
621              * So we need to check for that.
622              */
```

```
623             {
624                     struct sk_buff *skb;
625                     unsigned long flags;

626                     save_flags(flags);
627                     cli();
628                     skb = sk->send_head;
629                     if (!skb)
630                     {
631                             restore_flags(flags);
632                     }
633                     else
634                     {
635                             /*
636                               *    Kicked by a delayed ack. Reset timer
637                               *    correctly now
638                               */
639                             if (jiffies < skb->when + sk->rto)
640                             {
641                                     reset_xmit_timer (sk, TIME_WRITE, skb->when + sk->rto - jiffies);
642                                     restore_flags(flags);
643                                     break;
644                             }
645                             restore_flags(flags);
646                             /*
647                               *    Retransmission
648                               */
649                             sk->prot->retransmit (sk, 0);
650                             tcp_write_timeout(sk);
651                     }
652                     break;
653             }
654             /* Sending Keepalives */
655             case TIME_KEEPOPEN:
656                     /*
657                       * this reset_timer() call is a hack, this is not
658                       * how KEEPOPEN is supposed to work.
659                       */
660                     reset_xmit_timer (sk, TIME_KEEPOPEN, TCP_TIMEOUT_LEN);

661                     /* Send something to keep the connection open. */
662                     if (sk->prot->write_wakeup)
663                             sk->prot->write_wakeup (sk);
664                     sk->retransmits++;
```

```
665                 tcp_write_timeout(sk);
666                 break;
667             default:
668                 printk ("rexmit_timer: timer expired - reason unknown\n");
669                 break;
670         }
671         release_sock(sk);
672 }
```

retransmit_timer 函数是重传定时器到期执行函数，其根据重传定时器具体超时原因（由 sock 结构 ip_xmit_timeout 表示）进行相应处理。

1） 如果表示相应超时套接字的 sock 结构正被使用，则继续延迟 1s 后进行处理。

2） 如果本地存在尚未应答的数据包且连接并未被复位，则顺带发送应答数据包给远端。
注意 sock 结构 zapped 字段只在 tcp_std_reset 函数中被赋值为 1，即只在本地接收到远端发送的复位数据包后设置。

3） 此后根据此次超时原因进行具体处理：

A． 窗口探测超时：窗口探测数据包是通信一端发送给另一端的探测窗口大小的数据包。对于 TCP 协议，其首部中包含一个窗口字段，用于表示该端目前接收缓冲区大小，另一端将根据该值节制其数据包的发送，以防止发送过快，使得对方接收缓冲区溢出，这是 TCP 协议提供的一种数据流控制方式。当一方检测到接收缓冲区大小不足以接收一个完整数据包时（一般为 MTU 大小），其不会通知给对方一个小的窗口以造成传输线路上小数据包泛滥，而是直接通报一个 0 窗口，对方在接收到 0 窗口通知后，会暂时停止向另一端发送数据包，直到得到一个非 0 大小的窗口通知，由于该通知可能丢失，此时将会造成一端在等待一个非 0 窗口通知，而另一端认为其发送了非 0 窗口通知（实际上丢失了）则等待对方发送普通数据包，从而双方各自等待，造成死锁。窗口探测数据包的作用即为了阻止这种由于非 0 窗口通知数据包丢失而造成的死锁问题。一般非 0 窗口的通知是由原先声称窗口为 0 的一方主动发送的，为了防止该非 0 窗口通知数据包丢失，另一端在接收到 0 窗口数据包后，会启动一个窗口探测数据包，（注意即便此时远端窗口为 0，仍然可以接收窗口探测数据包，这是 TCP 协议规范规定的），这样即便由于对方非 0 窗口通知数据包丢失，在接收到窗口探测数据包后，对方可以重新发送一个非 0 窗口通知数据包，从而解除上文中所叙述的可能的死锁问题。

在了解到窗口探测数据包的由来后，我们也清楚了何时应该启动窗口探测定时器：即当对方通知 0 窗口时。如此问题即集中在如何判定对方通知 0 窗口大小。具体启动是在 tcp_send_skb 函数和 tcp_ack 函数中进行的。

tcp_send_skb 函数在发送一个数据包时被调用，该函数启动窗口探测定时器的依据是：所有之前发送出去的数据包都已得到对方应答，而对方发送的所有数据包本地也已进行了应答，且当前发送的数据包长度超出远端通知的窗口范围，则启动窗口探测定时器。此时由于双方数据包都已得到对方应答，而本地当前将要发送的数据包由于窗口大小所限发送不出去，则必须得到对方更新窗口大小的数据包，而此时必须发送一个窗口探测数据包方能让对方发送一个更新窗口大小的数据包。原则上我们可以等待对方发送一个普通数据包从而更新窗口大小，但这种消极等待并不能解决问题，如果对方从不主动发送数据包，则本地接下来的所有数据包将永远发送不出去。当然，在实际网络中很少会出现双方都对对方发送的数据包进行了应答的情况下，一端窗口大小出现问题的现象，但对于代码实现必须考虑到各种危险情况，以免造成内核死锁。

tcp_ack 函数用于对对方发送一个应答数据包进行处理。该函数启动窗口探测定时器的机理与 tcp_send_skb 函数相同，实现代码基本一致，此处不再叙述。

如果 retransmit_timer 函数判断出此次超时是窗口探测定时器，则调用 tcp_send_probe0 函数发送窗口探测数据包，并调用 tcp_write_timeout 函数进行后续超时处理，该函数在前文中已有介绍。对于 tcp_send_probe0 函数，其一方面发送窗口探测数据包，另一方面采用指数退避算法更新超时间隔时间后，重新设置窗口探测定时器，以便在对方未响应的情况下，继续进行窗口探测。

B．TCP 数据包传输超时：这是最为普遍的重传定时器超时情况，当使用 TCP 协议的套接字发送一个数据包后，会立刻设置传输超时定时器，如果定时器到期时，尚未得到应答（实际上如果得到应答的话，则该定时器会被删除，不存在定时器到期的问题），则将重新发送该数据包。注意对于使用 TCP 协议的套接字而言，其发送的每一个数据包都暂时被缓存到 sock 结构 send_head 字段指向的队列中，直到接收到对端对该数据包的应答后，方才被真正释放。

C．保活定时器超时：保活定时器是为了防止由于远端通信端在不通知的情况下关闭的现象出现，此时由于本地得不到通知，仍然保存大量资源，如果这样的情况出现较多（如远端采用此种方式进行攻击），则本地将由于资源耗尽，将无法对其它需要的远端客户提供服务。保活数据包就是为了解决这一问题而设置的。保活数据包的作用就是探测远端通信端是否还“在线上”。一般在通常双方交互中，由于各自需要传输数据，无须进行保活数据包的发送（普通数据包即起到保活数据包的作用），但如果双方长时间未进行数据的交换，则本地需要专门发送保活数据包对远端主机的状态进行探测。保活数据包与普通数据包无异，只是其作用并非是发送数据，而是希望发送一个数据包后，得到对方应答，已确知对方还在，所以保活数据包中使用老序列号（即之前正常数据使用过的序列号）。保活数据包的发送即当双方长时间未进行通信的情况下，所以保活定时器就要在双方将要进入长时间“休眠”时进行设置，“休眠”的确定在代码实现上是这样判断的：如果本地发送队列和重发队列中都无数据包，即本地既无需要重发的数据包，也无新的数据包需要发送。可以预见在此后一段时间内，本地将不会发送数据包，即可能将进入一个“休眠”时期，此时处理上将启动保活定时器。当然，此处的判断是完全从本地的角度出发的，在本地无数据包可发送的情况下，远端或许会发送数据包，不过由于对于远端的情况无从把握，所以只好武断的设置保活定时器，这不会造成冲突，因为如果在设置保活定时器后接收到远端发送的数据包，则保活定时器将被删除。保活定时器具体的是在 tcp_send_ack, tcp_ack 函数中被设置的，设置条件如上所述。

如果 retransmit_timer 函数判断是保活定时器超时，其一方面重新设置保活定时器，另一方面调用 tcp_write_wakeup 向远端发送保活数据包。注意对于保活定时器，并不如前定时器一样，对于定时间隔使用指数退避算法，而是维持一个固定的间隔时间，原因很简单，保活的目的即在于探测远端通信端的活性，越快探测出越好，所以就没有必要每次探测后进行进一步的延迟。由于其功能上的不同，有别于上文中所述的窗口探测定时器和数据包重传定时器。

673 /*

```
674   * This routine is called by the ICMP module when it gets some
675   * sort of error condition.    If err < 0 then the socket should
676   * be closed and the error returned to the user.    If err > 0
677   * it's just the icmp type << 8 | icmp code.    After adjustment
678   * header points to the first 8 bytes of the tcp header.    We need
679   * to find the appropriate port.
680   */

681  void tcp_err(int err, unsigned char *header, unsigned long daddr,
682          unsigned long saddr, struct inet_protocol *protocol)
683  {
684          struct tcphdr *th;
685          struct sock *sk;
686          struct iphdr *iph=(struct iphdr *)header;

687          header+=4*iph->ihl;


688          th =(struct tcphdr *)header;
689          sk = get_sock(&tcp_prot, th->source, daddr, th->dest, saddr);

690          if (sk == NULL)
691                  return;

692          if(err<0)
693          {
694                  sk->err = -err;
695                  sk->error_report(sk);
696                  return;
697          }

698          if ((err & 0xff00) == (ICMP_SOURCE_QUENCH << 8))
699          {
700                  /*
701                    * FIXME:
702                    * For now we will just trigger a linear backoff.
703                    * The slow start code should cause a real backoff here.
704                    */
705                  if (sk->cong_window > 4)
706                          sk->cong_window--;
707                  return;
708          }

709  /*      sk->err = icmp_err_convert[err & 0xff].errno;
```

```
                -- moved as TCP should hide non fatals internally (and does) */

710     /*
711       * If we've already connected we will keep trying
712       * until we time out, or the user gives up.
713       */

714     if (icmp_err_convert[err & 0xff].fatal || sk->state == TCP_SYN_SENT)
715     {
716         if (sk->state == TCP_SYN_SENT)
717         {
718             tcp_statistics.TcpAttemptFails++;
719             tcp_set_state(sk,TCP_CLOSE);
720             sk->error_report(sk);       /* Wake people up to see the error (see connect in
sock.c) */
721         }
722         sk->err = icmp_err_convert[err & 0xff].errno;
723     }
724     return;
725 }
```

tcp_err 函数在系统接收到一个表示某种错误的数据包后，被 ICMP 模块（icmp_rcv）调用，用于对使用 TCP 协议的某个套接字进行处理。

1） 如果错误号小于 0，则表示这是一个较为严重的错误，此时一方面将错误号存储到 sock 结构 err 字段以便用户使用 ioctl 函数进行读取，另一方调用相关函数将错误通知给用户。
2） 如果是源端节制 ICMP 通知，则减小拥塞窗口即可。拥塞窗口是本地自身进行数据包发送节制的一种方式，这与远端窗口相对应而言，远端窗口是远端对本地数据包发送进行节制的一种方式。
3） 如果错误是致命的，则将错误存储起来，但并不立刻通知用户，用户在此后的操作中，会被通知到。
4） 如果错误发生在本地试图与远端建立连接的过程中，则取消连接，并及时通知用户，并将错误存储起来（存储在 sock 结构 err 字段中）。

```
726 /*
727  *  Walk down the receive queue counting readable data until we hit the end or we find a gap
728  *  in the received data queue (ie a frame missing that needs sending to us). Not
729  *  sorting using two queues as data arrives makes life so much harder.
730  */

731 static int tcp_readable(struct sock *sk)
732 {
733     unsigned long counted;
734     unsigned long amount;
735     struct sk_buff *skb;
```

```
736     int sum;
737     unsigned long flags;

738     if(sk && sk->debug)
739         printk("tcp_readable: %p - ",sk);

740     save_flags(flags);
741     cli();
742     if (sk == NULL || (skb = skb_peek(&sk->receive_queue)) == NULL)
743     {
744         restore_flags(flags);
745         if(sk && sk->debug)
746             printk("empty\n");
747         return(0);
748     }

749     counted = sk->copied_seq;     /* Where we are at the moment */
750     amount = 0;

751     /*
752      *    Do until a push or until we are out of data.
753      */

754     do
755     {
756         if (before(counted, skb->h.th->seq))      /* Found a hole so stops here */
757             break;
758         sum = skb->len -(counted - skb->h.th->seq);
        /* Length - header but start from where we are up to (avoid overlaps) */
759         if (skb->h.th->syn)
760             sum++;
761         if (sum > 0)
762         {                                 /* Add it up, move on */
763             amount += sum;
764             if (skb->h.th->syn)
765                 amount--;
766             counted += sum;
767         }
768         /*
769          * Don't count urg data ... but do it in the right place!
770          * Consider: "old_data (ptr is here) URG PUSH data"
771          * The old code would stop at the first push because
772          * it counted the urg (amount==1) and then does amount--
773          * *after* the loop.    This means tcp_readable() always
```

```
774                 * returned zero if any URG PUSH was in the queue, even
775                 * though there was normal data available. If we subtract
776                 * the urg data right here, we even get it to work for more
777                 * than one URG PUSH skb without normal data.
778                 * This means that select() finally works now with urg data
779                 * in the queue.    Note that rlogin was never affected
780                 * because it doesn't use select(); it uses two processes
781                 * and a blocking read().    And the queue scan in tcp_read()
782                 * was correct.    Mike <pall@rz.uni-karlsruhe.de>
783                 */
784                if (skb->h.th->urg)
785                        amount--;/* don't count urg data */
786                if (amount && skb->h.th->psh) break;
787                skb = skb->next;
788        }
789        while(skb != (struct sk_buff *)&sk->receive_queue);

790        restore_flags(flags);
791        if(sk->debug)
792                printk("got %lu bytes.\n",amount);
793        return(amount);
794 }
```

tcp_readable 函数用于查看目前可读数据的长度。套接字软件表示 sock 结构 copied_seq 字段表示到目前为止读取的数据序列号，例如 copied_seq=100,则表示序列号在 100 之前的数据均已被用户读取，序列号为 100 以后（包括 100）的数据尚未被用户读取。
又从上层而言，其读取的数据包均被缓冲在 sock 结构 receive_queue 字段指向的队列中，tcp_readable 函数即从该队列中取数据包，查看目前可读取的数据量。
判断的标准是：直到遇到紧急数据或者数据序列号出现断裂。此处紧急数据的含义即发送该数据的远端设置在 TCP 首部中设置了 PUSH 标志位，意为该数据需要立刻上传给用户应用程序；而序列号出现断裂即表示数据包乱序到达，由于数据需要按序送交给用户应用程序，故遇到序列号断裂后，即计算出目前可读取的数据量。在数据量的计算中，需要注意，因为 SYN 标志位占据一个序列号，故在使用序列号计算数据量时，如果数据包中设置了 SYN 标志位，则必须从数据量计算值减去 1。另外该函数实现也不将标志为 urgent 的数据计算在内，因为标志为 urgent 的数据是与普通数据区分处理的（从应用程序角度看也是如此）。

```
795 /*
796  * LISTEN is a special case for select..
797  */
798 static int tcp_listen_select(struct sock *sk, int sel_type, select_table *wait)
799 {
800        if (sel_type == SEL_IN) {
801                int retval;
```

```
802             sk->inuse = 1;
803             retval = (tcp_find_established(sk) != NULL);
804             release_sock(sk);
805             if (!retval)
806                     select_wait(&master_select_wakeup,wait);
807             return retval;
808     }
809     return 0;
810 }
```

tcp_listen_select 函数用于判断监听套接字接收队列中是否有已经与远端通信端完成建立的套接字存在。注意接收队列中缓冲的是数据包，套接字状态的检测可以通过查看 tcp_find_established 函数实现来看。

```
312 /*
313  *  Find someone to 'accept'. Must be called with
314  *  sk->inuse=1 or cli()
315  */

316 static struct sk_buff *tcp_find_established(struct sock *s)
317 {
318     struct sk_buff *p=skb_peek(&s->receive_queue);
319     if(p==NULL)
320         return NULL;
321     do
322     {
323         if(p->sk->state == TCP_ESTABLISHED || p->sk->state >= TCP_FIN_WAIT1)
324             return p;
325         p=p->next;
326     }
327     while(p!=(struct sk_buff *)&s->receive_queue);
328     return NULL;
329 }
```

注意 do{}while 语句块中的判断语句。

如果 tcp_find_establised 函数返回 NULL，则表示尚无已经完成连接建立的套接字存在，此时调用 select_wait 函数将允许进程封装在 wait_queue 结构中加入到由 master_select_wakeup 变量指向的 wait_queue 结构链表中，并返回 NULL，上层处理函数可根据 NULL 返回值将当前进程置于休眠状态，当然也可以仅仅将结构通知给用户应用程序而不做休眠操作。

master_select_wakeup 变量所指向 wait_queue 队列中休眠进程的唤醒是在 tcp_set_state 函数中进行的， tcp_set_state 函数前文已经介绍，此处再次列出其实现代码，进行分析。

```
258 static __inline__ void tcp_set_state(struct sock *sk, int state)
259 {
260     if(sk->state==TCP_ESTABLISHED)
```

```
261             tcp_statistics.TcpCurrEstab--;
262 #ifdef STATE_TRACE
263     if(sk->debug)
264             printk("TCP sk=%p, State %s -> %s\n",sk, statename[sk->state],statename[state]);
265 #endif
266     /* This is a hack but it doesn't occur often and it's going to
267        be a real             to fix nicely */

268     if(state==TCP_ESTABLISHED && sk->state==TCP_SYN_RECV)
269     {
270             wake_up_interruptible(&master_select_wakeup);
271     }
272     sk->state=state;
273     if(state==TCP_ESTABLISHED)
274             tcp_statistics.TcpCurrEstab++;
275 }
```

tcp_set_state 函数在一个套接字状态发生变化时（或者说改变一个套接字状态时）被调用。从实现来看，如果套接字先前的状态为 TCP_SYN_RECV, 而新状态为 TCP_ESTABLISHED，则表示一个套接字完成连接的建立，此时唤醒 master_select_wakeup 所指向队列中各休眠进程，从而使得各进程继续运行。因为之前进程之所以被缓存到该队列，正是由于其处理的监听套接字接收队列中无完成连接建立的通信套接字。当前既然有一个通信套接字完成连接，则可以使得某个进程继续其处理了。

```
811 /*
812  *  Wait for a TCP event.
813  *
814  *  Note that we don't need to set "sk->inuse", as the upper select layers
815  *  take care of normal races (between the test and the event) and we don't
816  *  go look at any of the socket buffers directly.
817  */
818 static int tcp_select(struct sock *sk, int sel_type, select_table *wait)
819 {
820     if (sk->state == TCP_LISTEN)
821             return tcp_listen_select(sk, sel_type, wait);

822     switch(sel_type) {
823     case SEL_IN:
824             if (sk->err)
825                     return 1;
826             if (sk->state == TCP_SYN_SENT || sk->state == TCP_SYN_RECV)
827                     break;

828             if (sk->shutdown & RCV_SHUTDOWN)
```

```
829                 return 1;

830             if (sk->acked_seq == sk->copied_seq)
831                 break;

832             if (sk->urg_seq != sk->copied_seq ||
833                 sk->acked_seq != sk->copied_seq+1 ||
834                 sk->urginline || !sk->urg_data)
835                 return 1;
836             break;

837         case SEL_OUT:
838             if (sk->shutdown & SEND_SHUTDOWN)
839                 return 0;
840             if (sk->state == TCP_SYN_SENT || sk->state == TCP_SYN_RECV)
841                 break;
842             /*
843               * This is now right thanks to a small fix
844               * by Matt Dillon.
845               */

846             if (sk->prot->wspace(sk) < sk->mtu+128+sk->prot->max_header)
847                 break;
848             return 1;

849         case SEL_EX:
850             if (sk->err || sk->urg_data)
851                 return 1;
852             break;
853         }
854         select_wait(sk->sleep, wait);
855         return 0;
856 }
```

tcp_select 函数用于各种情况的测试，使系统调用 select 函数的针对网络栈 TCP 协议的底层实现。

1） 如果所探测的套接字当前状态为 TCP_LISTEN（即这是一个监听套接字），则调用 tcp_listen_select 函数查看是否有已经完成连接建立的套接字存在，具体情况参考上文。

2） 如果所探测的类型为 SEL_IN,即探测是否有数据可读取，则通过检查 sock 结构几个字段值进行判断。此处需要注意的一点是 acked_seq 值在逻辑上（当不发生值回绕时，在数值上）一定大于 copied_seq 字段值。这一点从各字段表示的意义上也可看出：acked_seq 字段表示本地希望接收到的下一个数据序列号，copied_seq 字段表示本地已被读取的最后一个数据的序列号加 1。而被读取的数据一定是已经接收到的，copied_seq 字段值是已经接收的最后一个数据的序列号加 1。

3） 如果探测类型为 SEL_OUT,即表示探测对应套接字发送缓冲区的空闲空间大小。注意此处的

实现是：如果该空闲空间大小不能缓存一个 MTU 长度的数据包，仍然返回无效值。

4） 如果探测类型为 SEL_EX,即表示探测之前操作是否有错误产生。

5） 如果在以上的判断中，该函数没有返回，则最后调用 select_wait 函数将当前执行进程插入到 sock 结构 sleep 队列中。上层函数将根据该函数的返回值判定是否将当前进程置于休眠状态。（应用程序对 select 系统调用一般为阻塞的，所以底层实现上在将当前进程插入 sleep 队列后，会置当前进程为休眠状态。）

```
857  int tcp_ioctl(struct sock *sk, int cmd, unsigned long arg)
858  {
859          int err;
860          switch(cmd)
861          {

862                  case TIOCINQ:
863  #ifdef FIXME /* FIXME: */
864                  case FIONREAD:
865  #endif
866                  {
867                          unsigned long amount;

868                          if (sk->state == TCP_LISTEN)
869                                  return(-EINVAL);

870                          sk->inuse = 1;
871                          amount = tcp_readable(sk);
872                          release_sock(sk);
873                          err=verify_area(VERIFY_WRITE,(void *)arg,
874                                                  sizeof(unsigned long));
875                          if(err)
876                                  return err;
877                          put_fs_long(amount,(unsigned long *)arg);
878                          return(0);
879                  }
880                  case SIOCATMARK:
881                  {
882                          int answ = sk->urg_data && sk->urg_seq == sk->copied_seq;

883                          err = verify_area(VERIFY_WRITE,(void *) arg,
884                                                  sizeof(unsigned long));
885                          if (err)
886                                  return err;
887                          put_fs_long(answ,(int *) arg);
888                          return(0);
889                  }
```

```
890             case TIOCOUTQ:
891             {
892                     unsigned long amount;

893                     if (sk->state == TCP_LISTEN) return(-EINVAL);
894                     amount = sk->prot->wspace(sk);
895                     err=verify_area(VERIFY_WRITE,(void *)arg,
896                                             sizeof(unsigned long));
897                     if(err)
898                             return err;
899                     put_fs_long(amount,(unsigned long *)arg);
900                     return(0);
901             }
902             default:
903                     return(-EINVAL);
904     }
905 }
```

tcp_ioctl 函数用于查询相关信息，此处对于可读取数据量的大小是通过调用 tcp_readable 函数完成；而发送缓冲区空闲空间大小则通过调用 sock_wspace 函数完成（sk->prot->wspace 指向的函数）。函数实现中的 SIOCATMARK 标志字段用于判断下一个将要读取得数据是否为 urgent 数据。tcp_ioctl 函数实现较为直观简单，此处不进一步说明。

```
906 /*
907  *    This routine computes a TCP checksum.
908  */

909 unsigned short tcp_check(struct tcphdr *th, int len,
910           unsigned long saddr, unsigned long daddr)
911 {
912       unsigned long sum;

913       if (saddr == 0) saddr = ip_my_addr();

914 /*
915  * stupid, gcc complains when I use just one __asm__ block,
916  * something about too many reloads, but this is just two
917  * instructions longer than what I want
918  */
919       __asm__("
920             addl %%ecx, %%ebx
921             adcl %%edx, %%ebx
922             adcl $0, %%ebx
923             "
```

```
924         : "=b"(sum)
925         : "0"(daddr), "c"(saddr), "d"((ntohs(len) << 16) + IPPROTO_TCP*256)
926         : "bx", "cx", "dx" );
927         __asm__("
928                 movl %%ecx, %%edx
929                 cld
930                 cmpl $32, %%ecx
931                 jb 2f
932                 shrl $5, %%ecx
933                 clc
934 1:              lodsl
935                 adcl %%eax, %%ebx
936                 lodsl
937                 adcl %%eax, %%ebx
938                 lodsl
939                 adcl %%eax, %%ebx
940                 lodsl
941                 adcl %%eax, %%ebx
942                 lodsl
943                 adcl %%eax, %%ebx
944                 lodsl
945                 adcl %%eax, %%ebx
946                 lodsl
947                 adcl %%eax, %%ebx
948                 lodsl
949                 adcl %%eax, %%ebx
950                 loop 1b
951                 adcl $0, %%ebx
952                 movl %%edx, %%ecx
953 2:              andl $28, %%ecx
954                 je 4f
955                 shrl $2, %%ecx
956                 clc
957 3:              lodsl
958                 adcl %%eax, %%ebx
959                 loop 3b
960                 adcl $0, %%ebx
961 4:              movl $0, %%eax
962                 testw $2, %%dx
963                 je 5f
964                 lodsw
965                 addl %%eax, %%ebx
966                 adcl $0, %%ebx
967                 movw $0, %%ax
```

```
968  5:          test $1, %%edx
969              je 6f
970              lodsb
971              addl %%eax, %%ebx
972              adcl $0, %%ebx
973  6:          movl %%ebx, %%eax
974              shrl $16, %%eax
975              addw %%ax, %%bx
976              adcw $0, %%bx
977              "
978          : "=b"(sum)
979          : "0"(sum), "c"(len), "S"(th)
980          : "ax", "bx", "cx", "dx", "si" );

981          /* We only want the bottom 16 bits, but we never cleared the top 16. */

982          return((~sum) & 0xffff);
983  }
```

tcp_check 函数用于计算 TCP 校验和。TCP 校验和包含 TCP 数据及其负载。读者可借此了解一下内联汇编的格式。GNU 下内联汇编编程在很多资料上都有介绍，为便于读者查阅，附录 B 中介绍了 GNU 内联汇编的一些知识。在实现上需要注意的是，虽然在计算中使用 32 位，不过在最后处理上是将高 16 位与低 16 相加，将此相加得到的结果取反（NOT）后返回低 16 位值。

```
984  void tcp_send_check(struct tcphdr *th, unsigned long saddr,
985              unsigned long daddr, int len, struct sock *sk)
986  {
987          th->check = 0;
988          th->check = tcp_check(th, len, saddr, daddr);
989          return;
990  }
```

tcp_send_check 函数用于计算 TCP 首部中校验和字段，注意在计算之前需要首先将该字段值清零。该函数只用在数据包发送中，对于接收的数据包，检查其 TCP 校验和是否正确的方式是：只需将校验和字段计算在内，检查计算出的结果是否为 0 即可，如为 0，则表示校验和正确，否则表示校验和错误，传输过程中出现了问题。

TCP 协议数据包发送到网络之前可能经过如下队列：

1） partial 指向的队列，该队列只缓存一个数据包，当该数据包长度大于 MSS 值时，将被发送出去或者缓存到其它队列上。

2） write_queue 指向的队列，该队列缓存所有由于窗口限制或者由于本地节制而缓存下来的数据包，该队列中可以无限制的缓存被节制下来的数据包。

以上队列中数据包的缓存是在传输层进行的。

3） send_head 指向的队列（send_tail 指向该队列的尾部），该队列缓存已发送出去但尚未得到对

方应答的数据包。

send_head 指向的队列中的数据包是在网络层中被缓存的。

以上 partial，write_queue, send_head, send_tail 均是 sock 结构中的字段。

partial 队列中缓存的单个数据包源于 TCP 协议的流式传输，对于 TCP 协议，为了避免在网络中传输小数据包，充分利用网络效率，底层网络栈实现对于用户应用程序发送的少量数据进行收集缓存，当积累到一定数量后（MSS），方才作为整个包发送出去。partial 队列中数据包的意义即在于此，对于少量数据，如果数据并非是 OOB 数据（即无需立刻发送给远端），则暂时分配一个大容量的数据包，将数据拷贝到该大数据包中，之后将该数据包缓存到 partial 队列中，当下次用户继续发送数据时，内核首先检查 partial 队列中是否有之前未填充满的数据包，则这些数据可以继续填充到该数据包，直到填满才将其发送出去。当然为了尽量减少对应用程序效率的影响，这个等待填满的时间是一定的，在实现上，内核设置一个定时器，当定时器超时时，如果 partial 队列中缓存有未填满的数据包，仍然将其发送出去，超时发送函数为 tcp_send_partial. 此外在其它条件下，当需要发送 partial 中数据包时，内核也直接调用 tcp_send_partial 函数进行发送。

write_queue 队列中缓存的是由于窗口限制或者本地数据包节制而缓存下来的数据包。该队列中数据包尚未真正发送出去，而是暂时被缓存下来，等待窗口限制解除或者未应答数据包减少从而跳出节制范围时被发送出去。该队列中数据包将由 tcp_write_xmit 函数负责发送出去，tcp_write_xmit 函数被 tcp_ack 调用，而 tcp_ack 函数用于处理接收到的应答数据包。

数据封装后可以不经过 partial 和 write_queue 指向的队列，直接被发送出去，对于不经过 partial 队列的原因如下：
1） 数据为 OOB 数据（亦称为带外数据），需要立刻发送给远端。
2） 数据长度够长，达到 MSS，内核无需进行等待合并。
3） 之前发送的所有数据包都得到应答，且亦无缓存的数据包待发送。
如下情况下，数据包将被暂时缓存到 write_queue 队列中，除此之外，数据包将不经过 write_queue 队列而直接被发送出去。
1） 数据长度受窗口大小限制，即数据最后一个字节的序列号已在远端窗口所声称的最大序列号之外。
2） 内核正在试图进行其它数据的重传。
3） 待应答的数据包个数超出了系统设定值。

对于 TCP 协议而言，其发送的每个数据包都将必然被缓存到 send_head 队列中以保证其所声称的可靠性数据传输。该队列中的数据包可能在传送超时时被重新发送或者在得到应答后从该队列中删除。

当然如果继续向下层分析，其实在表示网卡设备的结构中（device 结构）还存在一个缓存队列，该队列用于缓存由于网卡处于正忙状态而暂时未能发送的数据包。一旦数据包下次被发送出去，数据包即可从队列中删除，网卡设备不负责数据包在传输过程中可能丢失的问题。

由于 tcp_send_skb 函数较长，我们分段进行分析。

```
991  /*
992   *    This is the main buffer sending routine. We queue the buffer
993   *    having checked it is sane seeming.
994   */

995  static void tcp_send_skb(struct sock *sk, struct sk_buff *skb)
996  {
997       int size;
998       struct tcphdr * th = skb->h.th;

999       /*
1000           *    length of packet (not counting length of pre-tcp headers)
1001           */

1002           size = skb->len - ((unsigned char *) th - skb->data);

1003           /*
1004           *    Sanity check it..
1005           */

1006           if (size < sizeof(struct tcphdr) || size > skb->len)
1007           {
1008                printk("tcp_send_skb: bad skb (skb = %p, data = %p, th = %p, len = %lu)\n",
1009                     skb, skb->data, th, skb->len);
1010                kfree_skb(skb, FREE_WRITE);
1011           return;
1012           }

1013           /*
1014           *    If we have queued a header size packet.. (these crash a few
1015           *    tcp stacks if ack is not set)
1016           */

1017           if (size == sizeof(struct tcphdr))
1018           {
1019                /* If it's got a syn or fin it's notionally included in the size..*/
1020                if(!th->syn && !th->fin)
1021                {
1022                     printk("tcp_send_skb: attempt to queue a bogon.\n");
1023                     kfree_skb(skb,FREE_WRITE);
1024                     return;
1025                }
1026           }
```

```
1027        /*
1028         *  Actual processing.
1029         */

1030        tcp_statistics.TcpOutSegs++;
1031        skb->h.seq = ntohl(th->seq) + size - 4*th->doff;
```

以上代码着重检查数据包的有效性。size 参数值表示的是 TCP 首部及其可能的负载的长度。如果该值小于 TCP 首部长度或者大于整个数据包的长度，则表示长度存在问题，此时将直接释放数据包并返回。如果长度值等于 TCP 首部长度，则表示 TCP 负载部分长度为 0，对于 SYN 和 FIN 数据包是有效的，所以在此条件下，代码即检查 TCP 首部中 SYN，FIN 字段的设置情况。如果检查无误，则更新统计信息，并且为了此后对应答的处理，更新 skb->h.seq 字段。该字段的意义表示对该数据包的应答数据包所应具有的最小序列号，tcp_ack 函数将由此判断是否将缓存在 send_head 队列中的该数据包删除。注意该字段的计算方式为初始序列号（该数据包 TCP 首部中 seq 字段值）与纯数据长度之和。

```
1032        /*
1033         *  We must queue if
1034         *
1035         *  a) The right edge of this frame exceeds the window
1036         *  b) We are retransmitting (Nagle's rule)
1037         *  c) We have too many packets 'in flight'
1038         */

1039        if (after(skb->h.seq, sk->window_seq) ||
1040            (sk->retransmits && sk->ip_xmit_timeout == TIME_WRITE) ||
1041             sk->packets_out >= sk->cong_window)
1042        {
1043            /* checksum will be supplied by tcp_write_xmit.   So
1044             * we shouldn't need to set it at all.   I'm being paranoid */
1045            th->check = 0;
1046            if (skb->next != NULL)
1047            {
1048                printk("tcp_send_partial: next != NULL\n");
1049                skb_unlink(skb);
1050            }
1051            skb_queue_tail(&sk->write_queue, skb);

1052            /*
1053             *  If we don't fit we have to start the zero window
1054             *  probes. This is broken - we really need to do a partial
1055             *  send _first_ (This is what causes the Cisco and PC/TCP
1056             *  grief).
1057             */
```

```
1058                if (before(sk->window_seq, sk->write_queue.next->h.seq) &&
1059                    sk->send_head == NULL && sk->ack_backlog == 0)
1060                    reset_xmit_timer(sk, TIME_PROBE0, sk->rto);
1061            }
1062            else
1063            {
1064                /*
1065                 *    This is going straight out
1066                 */

1067                th->ack_seq = ntohl(sk->acked_seq);
1068                th->window = ntohs(tcp_select_window(sk));

1069                tcp_send_check(th, sk->saddr, sk->daddr, size, sk);

1070                sk->sent_seq = sk->write_seq;

1071                /*
1072                 *    This is mad. The tcp retransmit queue is put together
1073                 *    by the ip layer. This causes half the problems with
1074                 *    unroutable FIN's and other things.
1075                 */

1076                sk->prot->queue_xmit(sk, skb->dev, skb, 0);

1077                /*
1078                 *    Set for next retransmit based on expected ACK time.
1079                 *    FIXME: We set this every time which means our
1080                 *    retransmits are really about a window behind.
1081                 */

1082                reset_xmit_timer(sk, TIME_WRITE, sk->rto);
1083            }
1084        }
```

之后根据当前条件决定是将这个数据包缓存到 write_queue 队列中，还是直接将其发送给下层。如果以下三个条件满足，数据包将被暂时缓存到 write_queue 队列中，否则直接发送往下层（通过调用 ip_queue_xmit 函数）。

1） 数据包长度超出远端窗口界限。

2） 正在进行其它数据包的重传。

3） 带应答数据包的个数超出系统规定值。

如果仅仅是由于窗口问题而缓存数据包，则此时还必须启动窗口探测定时器，进行窗口探测，

以免造成双方等待死锁，有关窗口探测定时器的内容请参考前文中介绍。

如果数据包直接被发送往下层，则最后还须启动传输超时定时器。对于使用 TCP 协议的套接字发送的每个数据包，系统在发送出一个数据包后，都会重新启动一个传输超时定时器，从而保证数据可能丢失时进行重传，从而实现 TCP 协议所声称的可靠性数据传输。

```
1085    /*
1086     *  Locking problems lead us to a messy situation where we can have
1087     *  multiple partially complete buffers queued up. This is really bad
1088     *  as we don't want to be sending partial buffers. Fix this with
1089     *  a semaphore or similar to lock tcp_write per socket.
1090     *
1091     *  These routines are pretty self descriptive.
1092     */

1093    struct sk_buff * tcp_dequeue_partial(struct sock * sk)
1094    {
1095            struct sk_buff * skb;
1096            unsigned long flags;

1097            save_flags(flags);
1098            cli();
1099            skb = sk->partial;
1100    if (skb) {
1101            sk->partial = NULL;
1102            del_timer(&sk->partial_timer);
1103    }
1104    restore_flags(flags);
1105    return skb;
1106}
```

tcp_dequeue_partial 函数完成的功能如其名，从 partial 字段指向的队列中取数据包，如果之前设置了等待超时定时器（partial_timer），则同时删除该定时器。

```
1107/*
1108 *  Empty the partial queue
1109 */

1110static void tcp_send_partial(struct sock *sk)
1111{
1112    struct sk_buff *skb;

1113    if (sk == NULL)
1114            return;
```

```
1115        while ((skb = tcp_dequeue_partial(sk)) != NULL)
1116                tcp_send_skb(sk, skb);
1117}
```

tcp_send_partial 函数发送 partial 队列中缓存的数据包，该函数连续调用 tcp_dequeue_partial 和 tcp_send_skb 函数完成发送任务。

```
1118/*
1119 *    Queue a partial frame
1120 */

1121void tcp_enqueue_partial(struct sk_buff * skb, struct sock * sk)
1122{
1123        struct sk_buff * tmp;
1124        unsigned long flags;

1125        save_flags(flags);
1126        cli();
1127        tmp = sk->partial;
1128        if (tmp)
1129                del_timer(&sk->partial_timer);
1130        sk->partial = skb;
1131        init_timer(&sk->partial_timer);
1132        /*
1133         *    Wait up to 1 second for the buffer to fill.
1134         */
1135        sk->partial_timer.expires = HZ;
1136        sk->partial_timer.function = (void (*)(unsigned long)) tcp_send_partial;
1137        sk->partial_timer.data = (unsigned long) sk;
1138        add_timer(&sk->partial_timer);
1139        restore_flags(flags);
1140        if (tmp)
1141                tcp_send_skb(sk, tmp);
1142}
```

tcp_enqueue_partial 函数将一个新的数据包缓存到 partial 队列中，如前文所述，partial 队列中只可缓存一个数据包，所以如果 partial 队列中已存在数据包，则调用 tcp_send_skb 函数将该数据包发送出去。由于缓存一个新的数据包，所以之前为旧数据包设置的定时器（partial）_timer)将重新被设置。注意此处定时器到期执行函数为 tcp_send_partial，定时时间为 HZ，即 1 秒，也即此版本 TCP 协议实现等待合并小数据包的时间延迟为 1 秒。

```
1143/*
1144 *    This routine sends an ack and also updates the window.
1145 */
```

```
1146static void tcp_send_ack(unsigned long sequence, unsigned long ack,
1147              struct sock *sk,
1148              struct tcphdr *th, unsigned long daddr)
1149{
1150      struct sk_buff *buff;
1151      struct tcphdr *t1;
1152      struct device *dev = NULL;
1153      int tmp;

1154      if(sk->zapped)
1155            return;           /* We have been reset, we may not send again */

1156      /*
1157       * We need to grab some memory, and put together an ack,
1158       * and then put it into the queue to be sent.
1159       */

1160      buff = sk->prot->wmalloc(sk, MAX_ACK_SIZE, 1, GFP_ATOMIC);
1161      if (buff == NULL)
1162      {
1163            /*
1164             *    Force it to send an ack. We don't have to do this
1165             *    (ACK is unreliable) but it's much better use of
1166             *    bandwidth on slow links to send a spare ack than
1167             *    resend packets.
1168             */

1169            sk->ack_backlog++;
1170            if (sk->ip_xmit_timeout != TIME_WRITE && tcp_connected(sk->state))
1171            {
1172                  reset_xmit_timer(sk, TIME_WRITE, HZ);
1173            }
1174            return;
1175      }
```

tcp_send_ack 函数用于向远端发送应答数据包，由于该函数较长，将分成几段对其进行介绍。

1） 参数介绍

A． sequence：该参数指定 TCP 首部中 seq 字段，即本地数据目前序列号。

B．ack：该参数指定 TCP 首部中 ack_seq 字段，即应答序列号，或者说希望远端发送的下一个字节数据的序列号。

C．daddr：远端 IP 地址。

2） sock 结构 zapped 字段用于标识本地是否接收到远端发送的 RESET 复位数据包，zapped=1 表示对方发送了复位数据包进行了连接复位，所以在发送任何数据包（包括应答数据包）之前必须重新进行连接，换句话说，如果 zapped=1，此处不必发送应答数据包。

3） 如果数据包内核缓冲区分配失败，则增加应答计数器计数，从而在稍后进行应答，这个“稍后发送“的完成是通过设置重传定时器完成的。通过设置 sock 结构 ip_xmit_timeout 字段为 TIME_WAIT，并调用 reset_xmit_timer 即完成重传定时器的设置，注意此处设置的时间间隔为 HZ（即 1 秒）。

```
1176    /*
1177     *  Assemble a suitable TCP frame
1178     */

1179    buff->len = sizeof(struct tcphdr);
1180    buff->sk = sk;
1181    buff->localroute = sk->localroute;
1182    t1 =(struct tcphdr *) buff->data;

1183    /*
1184     *  Put in the IP header and routing stuff.
1185     */

1186    tmp = sk->prot->build_header(buff, sk->saddr, daddr, &dev,
1187                    IPPROTO_TCP, sk->opt, MAX_ACK_SIZE,sk->ip_tos,sk->ip_ttl);
1188    if (tmp < 0)
1189    {
1190        buff->free = 1;
1191        sk->prot->wfree(sk, buff->mem_addr, buff->mem_len);
1192        return;
1193    }
1194    buff->len += tmp;
1195    t1 =(struct tcphdr *)((char *)t1 +tmp);

1196    memcpy(t1, th, sizeof(*t1));

1197    /*
1198     *  Swap the send and the receive.
1199     */

1200        t1->dest = th->source;
1201        t1->source = th->dest;
1202        t1->seq = ntohl(sequence);
1203        t1->ack = 1;
1204        sk->window = tcp_select_window(sk);
1205        t1->window = ntohs(sk->window);
1206        t1->res1 = 0;
1207        t1->res2 = 0;
1208        t1->rst = 0;
```

```
1209        t1->urg = 0;
1210        t1->syn = 0;
1211        t1->psh = 0;
1212        t1->fin = 0;
```

以上代码即完成 MAC，IP，TCP 首部的建立，通过调用 ip_build_header(由 sk->prot->build_header 指向的函数）完成 MAC，IP 首部的建立，该函数返回 MAC，IP 首部的总长度。之后 tcp_send_ack 函数直接进行 TCP 首部的建立。

```
1213        /*
1214         *  If we have nothing queued for transmit and the transmit timer
1215         *  is on we are just doing an ACK timeout and need to switch
1216         *  to a keepalive.
1217         */

1218        if (ack == sk->acked_seq)
1219        {
1220            sk->ack_backlog = 0;
1221            sk->bytes_rcv = 0;
1222            sk->ack_timed = 0;
1223            if (sk->send_head == NULL && skb_peek(&sk->write_queue) == NULL
1224                        && sk->ip_xmit_timeout == TIME_WRITE)
1225            {
1226                if(sk->keepopen) {
1227                    reset_xmit_timer(sk,TIME_KEEPOPEN,TCP_TIMEOUT_LEN);
1228                } else {
1229                    delete_timer(sk);
1230                }
1231            }
1232        }
```

接着判断应答序列号参数与 sock 结构 acked_seq 字段值是否相等，sock 结构 acked_seq 字段表示该套接字希望从远端接收的下一个字节的序列号，如果二者相等，则表示这个应答数据包是最新的，所以此时可对待应答数据包计数器清零。如果套接字 send_head 重传队列和 write_queue 写队列都已清空，则无需设置传输超时定时器，此时由于双方可能进入长时间休眠（二者都不发送数据包），所以如果该套接字要求进行保活操作，则启动保活定时器，否则直接删除定时器（retransmit_timer）。

```
1233        /*
1234         *  Fill in the packet and send it
1235         */

1236        t1->ack_seq = ntohl(ack);
1237        t1->doff = sizeof(*t1)/4;
1238        tcp_send_check(t1, sk->saddr, daddr, sizeof(*t1), sk);
```

```
1239            if (sk->debug)
1240                    printk("\rtcp_ack: seq %lx ack %lx\n", sequence, ack);
1241            tcp_statistics.TcpOutSegs++;
1242            sk->prot->queue_xmit(sk, dev, buff, 1);
1243    }
```

函数最后设置 TCP 首部 ack_seq 应答序列号字段，计算校验和后，调用 ip_queue_xmit 函数将数据包发往网络层处理。

```
1244    /*
1245     *  This routine builds a generic TCP header.
1246     */

1247    extern __inline int tcp_build_header(struct tcphdr *th, struct sock *sk, int push)
1248    {
1249            memcpy(th,(void *) &(sk->dummy_th), sizeof(*th));

        /* 注意此处的序列号使用的是 write_seq, 而非 sent_seq。
         * 该函数目前仅被 tcp_write 调用构建 TCP 首部。
         */
1250            th->seq = htonl(sk->write_seq);
1251            th->psh =(push == 0) ? 1 : 0;
1252            th->doff = sizeof(*th)/4;
1253            th->ack = 1;
1254            th->fin = 0;
1255            sk->ack_backlog = 0;
1256            sk->bytes_rcv = 0;
1257            sk->ack_timed = 0;
1258            th->ack_seq = htonl(sk->acked_seq);
1259            sk->window = tcp_select_window(sk);
1260            th->window = htons(sk->window);

1261            return(sizeof(*th));
1262    }
```

tcp_build_header 函数专门用于创建 TCP 首部，该函数首先复制 sock 结构 dummy_th 字段，然后对一些通用字段进行赋值，其它调用该函数的例程完成其它 TCP 首部中其它字段的赋值并且可能根据具体情况修改本函数初始化的值。

```
1263    /*
1264     *  This routine copies from a user buffer into a socket,
1265     *  and starts the transmit system.
1266     */
```

```
1267    static int tcp_write(struct sock *sk, unsigned char *from,
1268            int len, int nonblock, unsigned flags)
1269    {
1270        int copied = 0;
1271        int copy;
1272        int tmp;
1273        struct sk_buff *skb;
1274        struct sk_buff *send_tmp;
1275        unsigned char *buff;
1276        struct proto *prot;
1277        struct device *dev = NULL;

1278        sk->inuse=1;
1279        prot = sk->prot;
```

tcp_write 函数被上层调用用于发送数据，这是对系统调用 write 函数的传输层处理函数。该函数也是其它很多此类功能函数的最后”归属“，对于 TCP 协议而言，如 send，sendto 系统调用最后都将调用 tcp_write 函数进行处理。此处函数参数中 from 字段表示待发送数据所在的用户缓冲区，len 表示待发送数据长度，nonblock 表示在暂时发送失败时（如内核缓冲区分配失败）是否进行等待，flags 表示数据的属性，如数据是否为 OOB 数据。

tcp_write 函数本身由一个 while 循环构成，循环条件为 len>0,即直到所有数据都进行了处理后方才退出。
首先判断之前套接字操作是否出现错误，如果出现错误，则返回错误。
其次在发送数据之前我们必须确定套接字状态为可发送数据状态，如果状态条件不满足，则根据 nonblock 参数决定是否等待：
A．对于发送通道被关闭的情况，则需返回管道错误并立刻返回，因为该错误是不可恢复的。
B．对于套接字连接状态不满足的情况，可以根据 nonblock 参数决定是否等待状态改变到可发送状态。
当套接字状态允许发送数据时，则进行内核数据封装结构的创建，复制用户缓冲区数据到内核结构缓冲区中，在成功完成帧首部创建后，将封装后的数据发往下层进行进一步的处理。
所以该函数虽然较长，但结构较为简单，下面我们进入该函数主体部分进行分析。

```
1280        while(len > 0)
1281        {
1282            if (sk->err)
1283            {               /* Stop on an error */
1284                release_sock(sk);
1285                if (copied)
1286                    return(copied);
1287                tmp = -sk->err;
1288                sk->err = 0;
1289                return(tmp);
1290            }
```

```
1291            /*
1292             *  First thing we do is make sure that we are established.
1293             */

1294            if (sk->shutdown & SEND_SHUTDOWN)
1295            {
1296                release_sock(sk);
1297                sk->err = EPIPE;
1298                if (copied)
1299                    return(copied);
1300                sk->err = 0;
1301                return(-EPIPE);
1302            }
```

此段代码完成对套接字之前操作错误检测，和通道状态检测。如果之前套接字操作出现错误，则直接返回错误号退出；如果发送管道被关闭，则返回管道错误退出。注意如果在发送过程中发送管道被关闭，则返回已经发送的数据字节数，而非管道错误号（负值）。

函数接下来将对套接字状态进行检测，这一个内嵌 while 循环。当套接字状态为 TCP_ESTABLISHED 或者为 TCP_CLOSE_WAIT 时退出循环。

```
1303            /*
1304             *  Wait for a connection to finish.
1305             */

1306            while(sk->state != TCP_ESTABLISHED && sk->state != TCP_CLOSE_WAIT)
1307            {
1308                if (sk->err)
1309                {
1310                    release_sock(sk);
1311                if (copied)
1312                        return(copied);
1313                    tmp = -sk->err;
1314                    sk->err = 0;
1315                    return(tmp);
1316                }
```

如果在检测状态的过程中，套接字操作出现错误，则：

1） 如果此时已经发送部分数据，则返回已发送的数据字节数。

2） 否则返回错误号。

```
1317                if (sk->state != TCP_SYN_SENT && sk->state != TCP_SYN_RECV)
1318                {
1319                    release_sock(sk);
1320                    if (copied)
```

```
1321                    return(copied);

1322                if (sk->err)
1323                {
1324                    tmp = -sk->err;
1325                    sk->err = 0;
1326                    return(tmp);
1327                }

1328                if (sk->keepopen)
1329                {
1330                    send_sig(SIGPIPE, current, 0);
1331                }
1332                return(-EPIPE);
1333            }
```

一般用户在调用 connect 函数后立刻调用 write 函数进行数据的发送，但由于建立连接需要时间，所以可能在调用 write 函数时，尚未完成建立连接的过程，此时套接字状态可能为 TCP_SYN_SENT 或者 TCP_SYN_RECV,这表示套接字正在建立连接，如果此时需要发送数据的话，则将根据用户设置的 nonblock 参数决定是否等待。
如果套接字并非是在进行连接，则表示出现某种错误，如果此时已经发送部分数据，则返回发送的数据字节数，如果 sock 结构中缓存有出现的错误问题，则直接将此错误返回给用户进行处理，否则返回管道错误给用户表示发送管道出现某种异常，无法进行发送。

release_sock 函数主要完成接收数据包的处理，以及在套接字状态已经设置为 TCP_CLOSE 后，设置 2MSL 等待时间。

```
1334            if (nonblock || copied)
1335            {
1336                release_sock(sk);
1337                if (copied)
1338                    return(copied);
1339                return(-EAGAIN);
1340            }
```

如果套接字正在进行连接建立（或者是一开始的连接建立或者是复位后的连接建立），此时间根据 nonblock 参数值决定是否进行等待。如果已经发送了一部分数据，则返回的已经发送的数据字节数。

```
1341            release_sock(sk);
1342            cli();

1343            if (sk->state != TCP_ESTABLISHED &&
1344                    sk->state != TCP_CLOSE_WAIT && sk->err == 0)
```

```
1345                {
1346                    interruptible_sleep_on(sk->sleep);

                /*如果被唤醒的原因是中断造成的，则：
                *1>如果已复制部分数据，则返回复制的数据字节数。
                *2>否则返回 ERESTARTSYS 错误标志，该标志用于指示用户程序重新
                *   进行写操作。
                */
1347                    if (current->signal & ~current->blocked)
1348                    {
1349                        sti();
1350                        if (copied)
1351                            return(copied);
1352                        return(-ERESTARTSYS);
1353                    }
1354                }
1355                sk->inuse = 1;
1356                sti();
1357            }
```

当套接字非如下状态时，则无法进行数据的发送：

1） TCP_ESTABLSIHED

连接状态，通信双方在通常情况下只在该状态进行数据的交换。

2） TCP_CLOSE_WAIT

当对方关闭其发送通道后，本地处于该状态，此时本地仍然可以继续发送数据。

以上代码即根据此事实进行判断，当然要求在套接字没有出现错误的情况下进行。当套接字状态不满足发送数据时，则调用 interruptible_sleep_on 函数进入睡眠等待。当被唤醒时，程序首先检查是否由于中断被唤醒，如是，则根据是否已经复制部分数据返回对应值。如果是因为状态转移到可发送数据状态，则设置 sock 结构 inuse 字段为 1，表示本进程在接下来的一段时间内将独占套接字资源进行相关处理，最后开启中断进入下一步操作。程序运行到此，表示套接字可以发送用户数据，我们继续分析函数余下代码。

```
1358            /*
1359             * The following code can result in copy <= if sk->mss is ever
1360             * decreased.  It shouldn't be.  sk->mss is min(sk->mtu, sk->max_window).
1361             * sk->mtu is constant once SYN processing is finished.  I.e. we
1362             * had better not get here until we've seen his SYN and at least one
1363             * valid ack.  (The SYN sets sk->mtu and the ack sets sk->max_window.)
1364             * But ESTABLISHED should guarantee that.  sk->max_window is by definition
1365             * non-decreasing.  Note that any ioctl to set user_mss must be done
1366             * before the exchange of SYN's.  If the initial ack from the other
1367             * end has a window of 0, max_window and thus mss will both be 0.
1368             */
```

```
1369        /*
1370         *    Now we need to check if we have a half built packet.
1371         */

1372            if ((skb = tcp_dequeue_partial(sk)) != NULL)
1373            {
1374                    int hdrlen;

1375                     /* IP header + TCP header */
1376                hdrlen = ((unsigned long)skb->h.th - (unsigned long)skb->data)
1377                         + sizeof(struct tcphdr);

1378                /* Add more stuff to the end of skb->len */
1379                if (!(flags & MSG_OOB))
1380                {
            /* skb->len-hdrlen  表示现有用户数据的长度，
            * sk->mss 表示最大用户数据长度，二者之差表示
            *  可增加用户数据长度。
            */
1381                    copy = min(sk->mss - (skb->len - hdrlen), len);
1382                    /* FIXME: this is really a bug. */
1383                    if (copy <= 0)
1384                    {
1385                        printk("TCP: **bug**: \"copy\" <= 0!!\n");
1386                        copy = 0;
1387                    }

1388                    memcpy_fromfs(skb->data + skb->len, from, copy);
1389                    skb->len += copy;
1390                    from += copy;
1391                    copied += copy;
1392                    len -= copy;
1393                    sk->write_seq += copy;
1394                }
1395                if ((skb->len - hdrlen) >= sk->mss ||
1396                    (flags & MSG_OOB) || !sk->packets_out)
1397                    tcp_send_skb(sk, skb);

            //这种情况下，表示当前需发送的数据长度较小，不足以达到发送长度，
            //故继续缓存该数据包以接收更多数据。
1398                else
1399                    tcp_enqueue_partial(skb, sk);
1400                continue;
```

```
1401                }
```

诚如前文所述，TCP 协议对于小数据会进行合并发送，sock 结构中 partial 字段指向的数据包即用于此目的。如果 partial 队列不为空，则从 partial 队列中取下数据包，用现在的数据填充这个数据包空闲缓冲区。如果当前用户发送数据仍然较少，不足以达到发送数据包的目的，则调用 tcp_enqueue_partial 函数继续将该数据包缓存在 partial 队列中，否则如果数据包长度已足够发送或者当前数据需要立刻发送给远端（设置了 OOB 标志），则调用 tcp_send_skb 立刻将数据包发送出去。对 partial 队列中数据包进行了处理后，程序回到最外的 while 循环继续对数据进行处理。如果用户数据长度较长，则在填充 partial 队列中可能有的数据包后，仍然需要重新分配一个新数据包进行数据的封装，这就是接下来代码所作的工作。

```
1402            /*
1403             * We also need to worry about the window.
1404             * If window < 1/2 the maximum window we've seen from this
1405             *    host, don't use it.    This is sender side
1406             *    silly window prevention, as specified in RFC1122.
1407             *    (Note that this is different than earlier versions of
1408             *    SWS prevention, e.g. RFC813.).    What we actually do is
1409             *    use the whole MSS.    Since the results in the right
1410             *    edge of the packet being outside the window, it will
1411      *     be queued for later rather than sent.
1412             */
1413            copy = sk->window_seq - sk->write_seq;/*当前窗口可容纳数据量*/
        /*如果窗口容限小于最大窗口值的一半，或者大于 MSS 值，都设置最终值为 MSS*/
1414                if (copy <= 0 || copy < (sk->max_window >> 1) || copy > sk->mss)
1415                    copy = sk->mss;
        /*如果容量大于用户实际要发送的数据量，暂时将容量设置为用户数据长度，
        *注意这个设置只是暂时的，下面代码将根据数据的紧迫性重新对容量值进行更新
        */
1416                if (copy > len)
1417                    copy = len;
```

首先计算本次可分配的数据包的容量，注意此处的计算值并非最终值，下面还将对其进行处理。

```
1418            /*
1419             *    We should really check the window here also.
1420             */

1421                send_tmp = NULL;
        /*如果数据非 OOB 数据，则可以进行等待以进行数据合并发送，所以更新容量
        *为 MSS 值。注意以下分配的容量为 MTU 加上首部可占用的最大长度，实际上分
        *配的空间绰绰有余。
        */
1422                if (copy < sk->mss && !(flags & MSG_OOB))
1423                {
1424                    /*
1425                     *    We will release the socket in case we sleep here.
```

```
1426                    */
1427                    release_sock(sk);
1428                    /*
1429                     *  NB: following must be mtu, because mss can be increased.
1430                     *  mss is always <= mtu
1431                     */
1432                    skb = prot->wmalloc(sk, sk->mtu + 128 + prot->max_header, 0,
                                                        GFP_KERNEL);
1433                    sk->inuse = 1;
```

/*注意此处对于临时变量 send_tmp 的设置，下面代码将根据 send_tmp 是否为 *进行不同的处理。此处对 send_tmp 进行赋值表示 send_tmp 不为 NULL！ */

```
1434                    send_tmp = skb;
1435                }
```

/*如果数据为 OOB 数据或容量值已经设置为 MSS，则直接分配以上计算的容量*/

```
1436                else
1437                {
1438                    /*
1439                     *  We will release the socket in case we sleep here.
1440                     */
1441                    release_sock(sk);
1442                    skb = prot->wmalloc(sk, copy + prot->max_header , 0, GFP_KERNEL);
1443                    sk->inuse = 1;
1444                }
```

接下来代码检查以上封装数据包的分配是否成功，如果 skb==NULL,则表示内核数据包结构分配失败，此时设置套接字标志位 SO_NOSPACE,如果 nonblock 参数为 1，则立刻返回。否则将进入睡眠等待。

```
1445                /*
1446                 *  If we didn't get any memory, we need to sleep.
1447                 */

1448                if (skb == NULL)
1449                {
1450                    sk->socket->flags |= SO_NOSPACE;
1451                    if (nonblock)
1452                    {
1453                        release_sock(sk);
1454                        if (copied)
1455                            return(copied);
1456                        return(-EAGAIN);
1457                    }
```

```
1458                    /*
1459                     *  FIXME: here is another race condition.
1460                     */
```

/*对原先的写缓冲区空闲空间大小进行保存，下面代码将不断检查 sock 结构
*wmem_alloc 字段与此处保存的旧值进行比较，从而判断写空闲缓冲区是否
*增大，从而可以进行封装数据包的分配.
*/

```
1461                    tmp = sk->wmem_alloc;
```

/*此处调用 release_sock，间接的发送数据包从而腾出写缓冲区空间*/

```
1462                    release_sock(sk);
1463                    cli();
1464                    /*
1465                     *  Again we will try to avoid it.
1466                     */
```

/*此处的判断条件一方面要检测写缓冲区增大，另一方面要保证套接字可进行
*数据发送，另外在此期间部可出现错误。
*/

```
1467                    if (tmp <= sk->wmem_alloc &&
1468                            (sk->state == TCP_ESTABLISHED||
                                sk->state == TCP_CLOSE_WAIT)
1469                            && sk->err == 0)
1470                    {
1471                            sk->socket->flags &= ~SO_NOSPACE;
                        /*睡眠等待写缓冲区空闲空间增大*/
1472                            interruptible_sleep_on(sk->sleep);
                        /*如果唤醒是由中断造成的，则返回相应值*/
1473                            if (current->signal & ~current->blocked)
1474                            {
1475                                    sti();
1476                                    if (copied)
1477                                            return(copied);
1478                                    return(-ERESTARTSYS);
1479                            }
1480                    }
```

/*程序运行到此处，表示有足够写空闲缓冲区供分配，此时设置 inuse 标志，
*表示本进程在使用该套接字。调用 continue 语句使程序回到前面进行封装
*数据包的分配。
*/

```
1481                    sk->inuse = 1;
1482                    sti();
```

```
1483                    continue;
1484            }
```

承接上文，如果 skb！=NULL，则表示封装数据包分配成功，此时将拷贝用户数据到给数据包中，并进行帧首部的建立，为发送数据包做准备。

```
1485            skb->len = 0;
1486            skb->sk = sk;
1487            skb->free = 0;
1488            skb->localroute = sk->localroute|(flags&MSG_DONTROUTE);
```

以上代码暂时初始化数据封装结构 sk_buff 结构变量 skb 中的一些字段，len 字段值将在稍后被更新。

```
1489            buff = skb->data;
```

buff 字段指向缓冲区首部，该缓冲区中将包含一个完整数据帧。以下代码将完成这个数据帧的创建：包括 MAC，IP，TCP 首部及最后用户数据。

```
1490            /*
1491             * FIXME: we need to optimize this.
1492             * Perhaps some hints here would be good.
1493             */

1494            tmp = prot->build_header(skb, sk->saddr, sk->daddr, &dev,
1495                                IPPROTO_TCP, sk->opt, skb->mem_len,sk->ip_tos,sk->ip_ttl);
1496            if (tmp < 0 )
1497            {
1498                    prot->wfree(sk, skb->mem_addr, skb->mem_len);
1499                    release_sock(sk);
1500                    if (copied)
1501                            return(copied);
1502                    return(tmp);
1503            }
```

调用 ip_build_header 函数创建 MAC，IP 首部，如果创建失败，释放封装数据包，返回相应值。

```
1504            skb->len += tmp;
1505            skb->dev = dev;
1506            buff += tmp;
1507            skb->h.th =(struct tcphdr *) buff;
```

更新字段值，注意此时 buff 变量指向将要创建的 TCP 首部的第一个字节位置处。

```
1508            tmp = tcp_build_header((struct tcphdr *)buff, sk, len-copy);
```

```
1509            if (tmp < 0)
1510            {
1511            prot->wfree(sk, skb->mem_addr, skb->mem_len);
1512                  release_sock(sk);
1513                  if (copied)
1514                        return(copied);
1515                  return(tmp);
1516            }
```

创建 TCP 首部，如果创建失败，返回相应值。如果 TCP 首部创建成功，则表示数据帧所有首部均创建成功，接下来要进行用户数据的复制填充。

```
1517            if (flags & MSG_OOB)
1518            {
1519                  ((struct tcphdr *)buff)->urg = 1;

                  /* 注意 TCP 首部中 urgent 字段的赋值，该字段被赋值为用户数据的长度。
                   * TCP 首部中 urgent 指针的大小为 16-bit，故最大值为 65535，
                    * 从而用其表示数据长度是合理的。
                   * 而不是位置的绝对地址（因为大多数时候序列号会超出该字段最大值）。
                   */
1520                  ((struct tcphdr *)buff)->urg_ptr = ntohs(copy);
1521            }
```

以上针对 OOB 数据设置 TCP 首部中 urg 标志位，该标志位设置为 1，接收端在接收到该数据包后将立刻将数据上交给用户应用程序，所以该标志位作为紧急数据的一种暗示。

```
1522            skb->len += tmp;
          //完成用户数据的复制
1523            memcpy_fromfs(buff+tmp, from, copy);

1524            from += copy;
1525            copied += copy;
1526            len -= copy;
1527            skb->len += copy;
1528            skb->free = 0;
1529            sk->write_seq += copy;
```

更新各变量值，注意 sock 结构之 write_seq 字段值的更新，该字段表示本地所发送数据的最后序列号，用于对数据进行编号，创建 TCP 首部时，其中 seq 字段值即来自于 write_seq。

```
1530            if (send_tmp != NULL && sk->packets_out)
1531            {
1532                  tcp_enqueue_partial(send_tmp, sk);
1533                  continue;
```

```
1534                }
```

注意在前文分析中特别交待，如果 send_tmp!=NULL，则表示可以等待合并后需数据，而且分配的数据包是按最大容量计算的以便合并后续用户数据，所以此处调用 tcp_euqueue_partial 函数将数据包缓存到 partial 队列中。

```
1535                tcp_send_skb(sk, skb);
```

如果无需缓存，则直接调用 tcp_send_skb 函数将该数据包发往下层进行处理。如果剩余数据长度仍然大于 0，则回到 while 循环继续其它剩余数据的处理。

```
1536            }//when(len>0)
```

跳出 while 循环，表示此次用户所以数据均已成功进行了处理，清 err 字段表示操作正常。
如果此时 partial 队列中缓存有数据包（等待后续数据），并且之前发送所有数据包得到应答（即无数据包在网络中，packets_out=0）,或者未采用 Nagle 算法（每次最多只能有一个未应答数据包在外发送）且数据包长度在窗口容限之内，则不再进行后续数据等待，而是调用 tcp_send_partial 将此数据包发送出去，这是为了利用这一段空闲时刻进行数据包的发送。

```
1537            sk->err = 0;

1538        /*
1539         *  Nagle's rule. Turn Nagle off with TCP_NODELAY for highly
1540         *  interactive fast network servers. It's meant to be on and
1541         *  it really improves the throughput though not the echo time
1542         *  on my slow slip link - Alan
1543         */

1544        /*
1545         *  Avoid possible race on send_tmp - c/o Johannes Stille
1546         */

1547            if(sk->partial && ((!sk->packets_out)
1548              /* If not nagling we can send on the before case too.. */
1549                    || (sk->nonagle && before(sk->write_seq , sk->window_seq))
1550                ))
1551                tcp_send_partial(sk);

1552            release_sock(sk);
1553            return(copied);
1554        }
```

到此，tcp_write 函数分析完毕。虽然较长，但实现功能简单。可以简单总结如下：
1） 检测发送通道是否正常，否则等待。
2） 是否需要进行前后数据合并，否则分配新封装数据包，是则填充 partial 队列中老数据包。

3） 是否成功分配封装数据包，否则等待。
4） 是否成功创建数据帧，否则退出，是则发送。
5） 所有操作中是否出现错误，是则返回错误退出。

```
1555    /*
1556     *  This is just a wrapper.
1557     */

1558    static int tcp_sendto(struct sock *sk, unsigned char *from,
1559            int len, int nonblock, unsigned flags,
1560            struct sockaddr_in *addr, int addr_len)
1561    {
1562        if (flags & ~(MSG_OOB|MSG_DONTROUTE))
1563            return -EINVAL;
1564        if (sk->state == TCP_CLOSE)
1565            return -ENOTCONN;
1566        if (addr_len < sizeof(*addr))
1567            return -EINVAL;
1568        if (addr->sin_family && addr->sin_family != AF_INET)
1569            return -EINVAL;
1570        if (addr->sin_port != sk->dummy_th.dest)
1571            return -EISCONN;
1572        if (addr->sin_addr.s_addr != sk->daddr)
1573            return -EISCONN;
1574        return tcp_write(sk, from, len, nonblock, flags);
1575    }
```

由于使用TCP协议的套接字在整个数据传输期间是一对一的，在通信过程中通信双方是固定的。所以一旦建立连接后，发送的所有数据包都是到达同一个目的端应用程序。虽然对于 TCP 协议定义了 sendto 函数，在发送时可以指定远端地址，但这个地址必须与起初建立连接时所指定的远端地址一致，不可指定其它地址以期望将该数据包发送出去。此处 sendto 函数传输层实现函数 tcp_sendto 在调用 tcp_write 函数发送数据进行的检查即为此：查看参数指定的远端地址是否为建立连接时指定的远端地址，如果不是，则直接错误返回。

```
1576    /*
1577     *  Send an ack if one is backlogged at this point. Ought to merge
1578     *  this with tcp_send_ack().
1579     */

1580    static void tcp_read_wakeup(struct sock *sk)
1581    {
1582        int tmp;
1583        struct device *dev = NULL;
```

```
1584        struct tcphdr *t1;
1585        struct sk_buff *buff;

1586        if (!sk->ack_backlog)
1587            return;

1588        /*
1589         * FIXME: we need to put code here to prevent this routine from
1590         * being called.  Being called once in a while is ok, so only check
1591         * if this is the second time in a row.
1592         */

1593        /*
1594         * We need to grab some memory, and put together an ack,
1595         * and then put it into the queue to be sent.
1596         */

1597        buff = sk->prot->wmalloc(sk,MAX_ACK_SIZE,1, GFP_ATOMIC);
1598        if (buff == NULL)
1599        {
1600            /* Try again real soon. */
1601            reset_xmit_timer(sk, TIME_WRITE, HZ);
1602            return;
1603        }

1604        buff->len = sizeof(struct tcphdr);
1605        buff->sk = sk;
1606        buff->localroute = sk->localroute;

1607        /*
1608         *  Put in the IP header and routing stuff.
1609         */

1610        tmp = sk->prot->build_header(buff, sk->saddr, sk->daddr, &dev,
1611                    IPPROTO_TCP, sk->opt, MAX_ACK_SIZE,sk->ip_tos,sk->ip_ttl);
1612        if (tmp < 0)
1613        {
1614            buff->free = 1;
1615            sk->prot->wfree(sk, buff->mem_addr, buff->mem_len);
1616            return;
1617        }

1618        buff->len += tmp;
1619        t1 =(struct tcphdr *)(buff->data +tmp);
```

```
1620            memcpy(t1,(void *) &sk->dummy_th, sizeof(*t1));
```

/* 不同于 tcp_buid_header 函数中该字段的赋值，此处使用 sent-seq，而非 write-seq。
 * 原因是此处构建的数据包会立刻发送出去。而调用 tcp_build_header 函数
* 构建 TCP 首部的对应数据包
 * 可能需要缓存到写队列中，即可能不会立刻发送出去。
 */

```
1621            t1->seq = htonl(sk->sent_seq);
1622            t1->ack = 1;
1623            t1->res1 = 0;
1624            t1->res2 = 0;
1625            t1->rst = 0;
1626            t1->urg = 0;
1627            t1->syn = 0;
1628            t1->psh = 0;
1629            sk->ack_backlog = 0;
1630            sk->bytes_rcv = 0;
1631            sk->window = tcp_select_window(sk);
1632            t1->window = ntohs(sk->window);
1633            t1->ack_seq = ntohl(sk->acked_seq);
1634            t1->doff = sizeof(*t1)/4;
1635            tcp_send_check(t1, sk->saddr, sk->daddr, sizeof(*t1), sk);
1636            sk->prot->queue_xmit(sk, dev, buff, 1);
1637            tcp_statistics.TcpOutSegs++;
1638        }
```

tcp_read_wakeup 该函数的名称令人费解，该函数实际上完成的功能即发送应答数据包。sock 结构中 ack_backlog 字段用于计算目前累计的应发送而未发送的应答数据包的个数。该函数被 cleanup_rbuf 函数调用，从该函数被调用的目的来看，主要是通知远端发送数据包。当然远端接收到这个应答数据包后，确实会对其发送队列进行处理（通过调用 tcp_write_xmit 函数），从这个意义上看，函数应该改为 tcp_write_wakeup 更为合适！实际上，TCP 协议确实定义了一个名为 tcp_write_wakeup 的函数，其与 tcp_read_wakeup 函数实现的区别在于 TCP 首部中对本地序列号的设置，在 tcp_read_wakeup 函数中设置如下：

t1->seq = htonl(sk->sent_seq);

而在 tcp_write_wakeup 函数中设置如下：

t1->seq = htonl(sk->sent_seq-1);

其中 t1 均表示 TCP 首部结构类型变量。seq 表示该数据包用户数据中第一个字节的序列号。二者虽然相差 1，但实际上从远端的角度来看，所起的作用基本相同。从本地的角度看，tcp_read_wakeup 函数可以看作是负责发送未应答数据包的专用函数，而 tcp_write_wakeup 函数的作用如其名，是暗示远端向本地发送数据包。此处给出 tcp_write_wakeup 函数实现如下：

```
4236    /*
4237     *  This routine sends a packet with an out of date sequence
```

```
4238        *   number. It assumes the other end will try to ack it.
4239        */

4240        static void tcp_write_wakeup(struct sock *sk)
4241        {
4242            struct sk_buff *buff;
4243            struct tcphdr *t1;
4244            struct device *dev=NULL;
4245            int tmp;
```

/*如果 sk->zapped=1,则表示该套接字已被远端复位，要发送数据包，必须重新建立
*连接。所以此处没有必要再发送数据包，直接退出。
*/

```
4246            if (sk->zapped)
4247                return;     /* After a valid reset we can send no more */

4248            /*
4249             *  Write data can still be transmitted/retransmitted in the
4250             *  following states.  If any other state is encountered, return.
4251             *  [listen/close will never occur here anyway]
4252             */

4253            if (sk->state != TCP_ESTABLISHED &&
4254                sk->state != TCP_CLOSE_WAIT &&
4255                sk->state != TCP_FIN_WAIT1 &&
4256                sk->state != TCP_LAST_ACK &&
4257                sk->state != TCP_CLOSING
4258            )
4259            {
4260                return;
4261            }

4262            buff = sk->prot->wmalloc(sk,MAX_ACK_SIZE,1, GFP_ATOMIC);
4263            if (buff == NULL)
4264                return;

4265            buff->len = sizeof(struct tcphdr);
4266            buff->free = 1;
4267            buff->sk = sk;
4268            buff->localroute = sk->localroute;

4269            t1 = (struct tcphdr *) buff->data;
```

```
4270            /* Put in the IP header and routing stuff. */
4271            tmp = sk->prot->build_header(buff, sk->saddr, sk->daddr, &dev,
4272                                         IPPROTO_TCP, sk->opt, MAX_ACK_SIZE,sk->ip_tos,sk->ip_ttl);
4273            if (tmp < 0)
4274            {
4275                  sk->prot->wfree(sk, buff->mem_addr, buff->mem_len);
4276                  return;
4277            }

4278            buff->len += tmp;
4279            t1 = (struct tcphdr *)((char *)t1 +tmp);

4280            memcpy(t1,(void *) &sk->dummy_th, sizeof(*t1));

4281            /*
4282             *    Use a previous sequence.
4283             *    This should cause the other end to send an ack.
4284             */

4285            t1->seq = htonl(sk->sent_seq-1);
4286            t1->ack = 1;
4287            t1->res1= 0;
4288            t1->res2= 0;
4289            t1->rst = 0;
4290            t1->urg = 0;
4291            t1->psh = 0;
4292            t1->fin = 0;     /* We are sending a 'previous' sequence, and 0 bytes of data - thus no
FIN bit */
4293            t1->syn = 0;
4294            t1->ack_seq = ntohl(sk->acked_seq);
4295            t1->window = ntohs(tcp_select_window(sk));
4296            t1->doff = sizeof(*t1)/4;
4297            tcp_send_check(t1, sk->saddr, sk->daddr, sizeof(*t1), sk);
4298             /*
4299              *  Send it and free it.
4300              *  This will prevent the timer from automatically being restarted.
4301              */
4302            sk->prot->queue_xmit(sk, dev, buff, 1);
4303            tcp_statistics.TcpOutSegs++;
4304      }
```

需要说明的一点是函数对当前套接字状态的判断，判断的原则是本地状态允许主动发送数据包。tcp_write_wakeup 函数只在 tcp_send_probe0 函数中被调用，tcp_send_probe0 函数用于发送窗口探测数据包，每当窗口探测定时器超时，该函数即被调用。窗口探测定时器是 TCP 协议四大定

时器之一（超时重传定时器，保活定时器，窗口探测定时器，2MSL 定时器），用于探测远端主机窗口，防止远端主机窗口通报数据包丢失从而造成死锁，具体情况参考 TCP 协议规范（RFC793）。从远端主机角度而言，在接收到一个 ACK 包后，其调用 tcp_ack 函数进行处理，该函数实现中检查是否有可以发送的数据包，如有即刻发送，由此而言，tcp_write_wakeup 确实完成了其所声称的作用。

```
1639    /*
1640     *  FIXME:
1641     *  This routine frees used buffers.
1642     *  It should consider sending an ACK to let the
1643     *  other end know we now have a bigger window.
1644     */
```

/* 本函数清除已被用户程序读取完的数据包，并通知远端本地新的窗口大小。*/

```
1645    static void cleanup_rbuf(struct sock *sk)
1646    {
1647        unsigned long flags;
1648        unsigned long left;
1649        struct sk_buff *skb;
1650        unsigned long rspace;

1651        if(sk->debug)
1652            printk("cleaning rbuf for sk=%p\n", sk);

1653        save_flags(flags);
1654        cli();

1655        left = sk->prot->rspace(sk);

1656        /*
1657         *  We have to loop through all the buffer headers,
1658         *  and try to free up all the space we can.
1659         */

1660        while((skb=skb_peek(&sk->receive_queue)) != NULL)
1661        {
1662            if (!skb->used || skb->users)
1663                break;
1664            skb_unlink(skb);
1665            skb->sk = sk;
1666            kfree_skb(skb, FREE_READ);
1667        }

1668        restore_flags(flags);
```

```
1669        /*
1670         *  FIXME:
1671         *  At this point we should send an ack if the difference
1672         *  in the window, and the amount of space is bigger than
1673         *  TCP_WINDOW_DIFF.
1674         */

1675        if(sk->debug)
1676            printk("sk->rspace = %lu, was %lu\n", sk->prot->rspace(sk),
1677                                left);
1678        if ((rspace=sk->prot->rspace(sk)) != left)
1679        {
1680            /*
1681             * This area has caused the most trouble.  The current strategy
1682             * is to simply do nothing if the other end has room to send at
1683             * least 3 full packets, because the ack from those will auto-
1684             * matically update the window.  If the other end doesn't think
1685             * we have much space left, but we have room for at least 1 more
1686             * complete packet than it thinks we do, we will send an ack
1687             * immediately.  Otherwise we will wait up to .5 seconds in case
1688             * the user reads some more.
1689             */
1690            sk->ack_backlog++;
1691        /*
1692         * It's unclear whether to use sk->mtu or sk->mss here.  They differ only
1693         * if the other end is offering a window smaller than the agreed on MSS
1694         * (called sk->mtu here).  In theory there's no connection between send
1695         * and receive, and so no reason to think that they're going to send
1696         * small packets.  For the moment I'm using the hack of reducing the mss
1697         * only on the send side, so I'm putting mtu here.
1698         */

1699            if (rspace > (sk->window - sk->bytes_rcv + sk->mtu))
1700            {
1701                /* Send an ack right now. */
1702                tcp_read_wakeup(sk);
1703            }
1704            else
1705            {
1706                /* Force it to send an ack soon. */
1707                int was_active = del_timer(&sk->retransmit_timer);
1708                if (!was_active || TCP_ACK_TIME < sk->timer.expires)
1709                {
```

```
1710                        reset_xmit_timer(sk, TIME_WRITE, TCP_ACK_TIME);
1711                }
1712                    else
1713                        add_timer(&sk->retransmit_timer);
1714                }
1715            }
1716        }
```

cleanup_rbuf 函数主要负责清除接收队列（receive_queue）中数据包，清除的依据是数据包中数据已被上层应用程序读取完，即 skb->used 设置为 1。上层应用程序在使用完一个数据包后，会相应的将该数据包的 used 字段设置为 1。cleanup_rbuf 函数即以此为据进行清除，从而释放接收缓冲区，如果释放后接收缓冲区增加到可以容纳 MTU 大小的数据包，则发送窗口通报数据包（通过 tcp_read_wakeup 函数发送）。否则简单重置之前设定的定时器。

```
1717    /*
1718     *    Handle reading urgent data. BSD has very simple semantics for
1719     *    this, no blocking and very strange errors 8)
1720     */

1721    static int tcp_read_urg(struct sock * sk, int nonblock,
1722                unsigned char *to, int len, unsigned flags)
1723    {
1724        /*
1725         *    No URG data to read
1726         */
1727        if (sk->urginline || !sk->urg_data || sk->urg_data == URG_READ)
1728            return -EINVAL;    /* Yes this is right ! */

1729        if (sk->err)
1730        {
1731            int tmp = -sk->err;
1732            sk->err = 0;
1733            return tmp;
1734        }

1735        if (sk->state == TCP_CLOSE || sk->done)
1736        {
1737            if (!sk->done) {
1738                sk->done = 1;
1739                return 0;
1740            }
1741            return -ENOTCONN;
1742        }
```

```
1743            if (sk->shutdown & RCV_SHUTDOWN)
1744            {
1745                    sk->done = 1;
1746                    return 0;
1747            }
1748            sk->inuse = 1;
1749            if (sk->urg_data & URG_VALID)
1750            {
1751                    char c = sk->urg_data;
1752                    if (!(flags & MSG_PEEK))
1753                            sk->urg_data = URG_READ;
1754                    put_fs_byte(c, to);
1755                    release_sock(sk);
1756                    return 1;
1757            }
1758            release_sock(sk);

1759            /*
1760              * Fixed the recv(..., MSG_OOB) behaviour.    BSD docs and
1761              * the available implementations agree in this case:
1762              * this call should never block, independent of the
1763              * blocking state of the socket.
1764              * Mike <pall@rz.uni-karlsruhe.de>
1765              */
1766            return -EAGAIN;
1767    }
```

tcp_read_urg 函数用于读取紧急（urgent）数据，TCP 首部中 URG，urg_pointer 字段用于处理紧急数据。当接收到一个 URG 字段设置为 1 的数据包后，表示该数据包中包含紧急数据，此时调用 tcp_urg 函数进行处理，该函数将 skb->urg_data 设置为紧急指针指向的字节：

```
sk->urg_data = URG_VALID | *(ptr + (unsigned char *) th);
```

同时设置 URG_VALID 标志。注意此处的 ptr 已经过换算，表示紧急数据相对于 TCP 首部第 1 个字节的偏移，而非是紧急指针，TCP 首部中紧急指针偏移量是相对于 TCP 首部中的序列号而言的，即相对于第一个纯数据对应的序列号而言的。

如此，tcp_read_urg 函数实现就较为容易理解，其在检查紧急数据有效后（检查 URG_VALID 标志位），即将此紧急数据拷贝到用户缓冲区中。注意此处的紧急数据被处理成一个字节，这与 TCP 协议规范中对于紧急指针的定位有些不一致。当一个数据包 TCP 首部中 URG 字段被设置时，我们称数据传送进入紧急模式，从其含义来看，该数据包中所包含的第一个字节直到紧急指针所指向的字节都应该属于紧急数据范围。对于实现中将紧急数据处理成一个字节，这种处理是实现相关的，因为本身对于紧急数据的处理方式 TCP 规范上并无明确说明，各具体实现可以自行对如何处理紧急数据进行各自的解释，而且 TCP 协议规范虽然定义了紧急模式并且指出紧急指针指向紧急数据的最后一个字节（或者有的地方将紧急指针解释为指向紧急数据过后的

第一个非紧急数据字节），但并没有说明紧急数据从何处开始，只是说当 TCP 首部中 URG 字段被设置时，就进入紧急数据传输模式，所以在实现上，各自实现均可有不同的解释。此版本内核中，紧急数据只被解释为紧急指针指向的那一个单个字节！

有一点需要指出的是，紧急数据与带外（OOB：Out Of Band）数据是不同的，不过现在这种不同被简单的混淆了，很多地方将此二者混为一谈，将他们等同起来。实际上紧急数据与带外数据是不同的概念，紧急数据是指插入到普通数据中一些字节，而待外数据一般需要新创建一个数据通道进行独自传输，如果将 OOB 翻译为待外通道应该更确切一些。另外这种混淆一大部分原因来源于 socket 编程接口直接将紧急数据映射为 OOB 数据。

对于接收数据包，内核维护如下几个队列：
1>全局 backlog 队列，驱动程序调用 netif_rx 将接收到的数据包缓存于该队列中。
2>sock 结构中 back_log 队列，网络层在 tcp_rcv 函数中将接收到的数据包缓存于该队列中，如果该数据包对应的套接字正在忙于处理其它任务，无暇处理该数据包时。
3>sock 结构中 receive_queue 队列，进入该队列的数据包方可由应用层程序读取。

将全局 backlog 队列中数据包向 sock->back_log 队列移动的函数为 net_bh 函数，该函数作为下半部分执行。而将 sock->back_log 中数据包向 sock->receive_queue 队列移动的函数为 release_sock。注意以上所指的移动并非直接的从一个队列中删除，并直接的加入的另一个队列中。此处着重强调的是数据包的转移途径和来源，实际上此处所指的移动要复杂的多，而且数据包从一个队列中删除，并非一定会加入的另一个队列中。

以下所讨论的 tcp_read 函数是系统调用 read 的网络层实现函数，其操作的就是 sock 结构 receive_queue 队列，只有进入该队列中数据包方可将其中的数据递交给应用程序处理。而 tcp_read 函数所实现的主要功能即是从 receive_queue 队列中取数据包，检查其中数据的合法性，并根据用户所要求读取的数据量，尽量拷贝该数量的数据到用户缓冲区中。所谓尽量拷贝是指有时将无法满足用户所要求的数据量，具体情况见下面对 tcp_read 函数的分析。该函数较长，我们分段进行分析。

```
1768    /*
1769     *    This routine copies from a sock struct into the user buffer.
1770     */

1771    static int tcp_read(struct sock *sk, unsigned char *to,
1772            int len, int nonblock, unsigned flags)
1773    {
1774            struct wait_queue wait = { current, NULL };
1775            int copied = 0;
1776            unsigned long peek_seq;
1777            volatile unsigned long *seq;   /* So gcc doesn't overoptimise */
1778            unsigned long used;

1779            /*
1780             *    This error should be checked.
```

```
1781            */
```

//对于侦听套接字而言，其不负责数据的传送，其 receive_queue 队列中缓存的均是]
//连接请求数据包（SYN 数据包），如果要读取的套接字状态为侦听状态，则表示应用
//层请求出现问题。

```
1782            if (sk->state == TCP_LISTEN)
1783                    return -ENOTCONN;

1784            /*
1785             *      Urgent data needs to be handled specially.
1786             */
```

//诚如上文对紧急指针的简单分析，虽然 OOB 数据本质上并非紧急数据，在此处直接
//将二者等同起来，如果上层请求 OOB 数据，则返回了紧急数据！
//tcp_read_urg 函数上文已有介绍，注意该函数返回一个字节的紧急数据，具体情况请
//参考上文分析。

```
1787            if (flags & MSG_OOB)
1788                    return tcp_read_urg(sk, nonblock, to, len, flags);

1789            /*
1790             *      Copying sequence to update. This is volatile to handle
1791             *      the multi-reader case neatly (memcpy_to/fromfs might be
1792             *      inline and thus not flush cached variables otherwise).
1793             */
```

//分配要更新的变量，如果仅仅是 PEEK，则不对内核变量进行更新。

```
1794            peek_seq = sk->copied_seq;
1795            seq = &sk->copied_seq;
1796            if (flags & MSG_PEEK)
1797                    seq = &peek_seq;
```

//将函数开始处初始化的 wait 变量加入到 sock 结构 sleep 睡眠队列中，下文代码
//随时可以进入睡眠等待状态。

```
1798            add_wait_queue(sk->sleep, &wait);
```

//将 sock 结构 inuse 字段设置为 1，表示套接字正忙。这种设置可以减少竞争。如
//tcp_rcv 函数处理数据包，如果检查到 inuse 字段为 1，则将数据包缓存到 sock 结构
//back_log 队列中退出，并不立刻对数据包进行处理。

```
1799            sk->inuse = 1;
```

以上片断代码只是做外围简单工作，对套接字状态的检查，以及对 OOB 数据的特殊处理，注意

此处代码将对 OOB 数据的请求直接映射到了紧急数据。另外上层读取数据时，需要更新 sock 结构 copied_seq 字段，从而跟踪上层应用程序读取的进度。如果仅仅是数据预先读取和检查(MSG_PEEK)，则不更新 copied_seq 字段，而是使用一个临时变量进行本函数的跟踪。最后将 sock 结构 inuse 字段设置为 1，表示套接字正忙，该字段实现了某种程度上的互斥机制。

下面我们进入 tcp_read 函数的主体部分，该部分实现为一个 while 循环，退出条件是上层所要求的数据量已得到满足。注意 while 循环中有几处将直接返回，即并非一定要读取上层要求的数据量方可返回，如上层使用了 NON_BLOCK 选项，即便没有读取到任何数据，也将直接返回。

```
1800        while (len > 0)
1801        {
1802            struct sk_buff * skb;
1803            unsigned long offset;

1804            /*
1805             * Are we at urgent data? Stop if we have read anything.
1806             */
        //此处跳出循环的依据是紧急数据不可与普通数据混为一谈，如果读取过程中
        //碰到紧急数据，则停止读取，返回已读取的数据量。
1807            if (copied && sk->urg_data && sk->urg_seq == *seq)
1808                break;
```

紧急数据是嵌入到普通数据流中的一段数据，在处理上或者其所要求的关注度上不同于普通数据，所以在读取时紧急数据和普通数据是分开的，如果在读取普通数据的过程中碰到紧急数据，则停止读取普通数据，不可将紧急数据作为普通数据的一部分返回给上层，上面代码的所做的处理即是如此。

```
1809            /*
1810             *  Next get a buffer.
1811             */

1812            current->state = TASK_INTERRUPTIBLE;

1813            skb = skb_peek(&sk->receive_queue);
1814            do
1815            {
```

此处将当前进程状态设置为可中断状态，在相关条件不满足时，随时进入睡眠等待。正常情况下，之前进程状态为 TASK_RUNNING。之后从 sock 结构 receive_queue 队列中取数据包，进入 do-while 内部循环。

```
1814            do
1815            {
1816                if (!skb)
1817                    break;
1818                if (before(*seq, skb->h.th->seq))
1819                    break;
1820                offset = *seq - skb->h.th->seq;
1821                if (skb->h.th->syn)
```

```
1822                    offset--;
1823                if (offset < skb->len)
1824                    goto found_ok_skb;
1825                if (skb->h.th->fin)
1826                    goto found_fin_ok;
1827                if (!(flags & MSG_PEEK))
1828                    skb->used = 1;
1829                skb = skb->next;
1830            }
1831            while (skb != (struct sk_buff *)&sk->receive_queue);
```

上面的 do-while 循环所做的工作如同一个检查和任务派遣机关：检查相关条件，在条件满足后，进入相关模块进行处理（使用 goto 跳转语句实现，而非函数调用）。首先检查 receive_queue 是否为空，如接收队列中暂时没有可读取的数据，则跳出此处的 while 循环，进入下面的处理，具体情况参见下文的分析。此后检查数据流是否出现的断裂，seq 变量指向的是当前应用程序的读取位置（读取的最后一个数据的序列号），如果小于接收队列中第一个数据包中数据的序列号，则表示出现了断裂，此时无法进行读取，也跳出此处的 while 循环。如果没有出现断裂，则将 offset 字段初始化为之前已读取最后一个字节在当前要读取数据包中的偏移位置，例如如果 seq 字段为 10，skb->h.th->seq 字段为 6，则表示 skb 所表示的数据包中 6，7，8，9 四个字节为重复字节，此时需要跳过，offset=10-6=4，即只需从偏移 4 处开始读取（偏移从 0 计数，0-3 偏移量对应 6-9 四个字节）。另外我们知道对于 SYN，FIN 标志位，其本身都使用一个序列号，如果该数据包中 SYN 标志位置 1，则将 offset 字段减去 1，由此来看，对于一个包含数据的 SYN 数据包而言，SYN 标志位使用了最开始的一个序列号，数据部分使用其后的序列号。以上例为例，则序列号 6 现在表示 SYN 标志位，数据部分从序列号 7 开始，而 seq 字段（即 copied_seq）为 10，则真正应该跳过 7，8，9 三个字节，所以此时的偏移量应该为 3，这正是将 offset 字段减 1 的原因所在。

如果 offset 小于 skb->len 则表示 skb 所表示的数据包中有可用数据，直接跳转到 found_ok_skb 处执行；否则表示该数据包中所有数据均属于重复数据，如果 MSG_PEEK 字段没有设置的话，则将 skb->used 字段设置为 1，表示该数据包已作处理（即数据已被读取完，可以释放），如果设置了 MSG_PEEK 标志，由于只是“偷看”，当然不能留下痕迹，所以将不作任何处理。另一个需要特殊处理的情况是数据包中 FIN 字段设置为 1，这表示远端发送的文件结束标志（FIN 字段设置为 1，表示远端已发送完其数据而且已请求关闭发送通道），此时无论上层应用程序要求读取多少数量，都需要返回（不可按通产情况进行等待，所以需要另行处理），这正是如下代码的作用，对于 FIN 标志位被设置的数据包，专门跳转到 found_fin_ok 标志符所表示的模块处进行处理。

```
1825                if (skb->h.th->fin)
1826                    goto found_fin_ok;
```

对该段 do-while 作用总结如下：

1>检查接收队列中是否存在可读的数据包，如无，跳出该 do-while 模块。

2>检查数据流是否出现断裂，如果出现断裂，跳出该 do-while 模块。

3>检查数据包中是否有可用数据，如有，跳转到 found_ok_skb 标志符表示的模块处进行处理。

4>检查数据包 FIN 字段设置情况，如果 FIN 字段被设置，则表示该数据包除了可能携带的数据以外，还是一个请求关闭发送通道数据包，此时需要特别处理，跳转到 found_fin_ok 标志符所

表示的模块处进行处理。
5>如果数据包中包含的都是重复数据，则在正常情况下（MSG_PEEK 标志位没有设置），设置 skb->used 字段设置为 1，表示该数据包可以被释放（cleanup_rbuf 函数负责释放），并处理下一个可能的数据包。

注意以上五种情况并非相关排斥，如数据包可以既有可读数据，且其 FIN 标志位也设置为 1，对应以上 3>,4>两种情况都满足，此时处理上对应的相关两个模块都会执行，但分析时我们每次只关注其中一个方面。
根据上面的总结，我们可以将如下分析分为三个方面：
A>暂时没有可读数据，对应以上总结中之 1>,2>两种情况。
B>有可读数据，对应以上总结中之 3>.
C>FIN 标志位被设置，对应以上总结中之 4>.

对于暂时无可读数据的情况，将直接跳出 do-while 循环，执行跟随该 do-while 循环之后的代码段，如下。

```
1832                if (copied)
1833                        break;

1834                if (sk->err)
1835                {
1836                        copied = -sk->err;
1837                        sk->err = 0;
1838                        break;
1839                }

1840                if (sk->state == TCP_CLOSE)
1841                {
1842                        if (!sk->done)
1843                        {
1844                                sk->done = 1;
1845                                break;
1846                        }
1847                        copied = -ENOTCONN;
1848                        break;
1849                }

1850                if (sk->shutdown & RCV_SHUTDOWN)
1851                {
1852                        sk->done = 1;
1853                        break;
1854                }

1855                if (nonblock)
1856                {
```

```
1857                copied = -EAGAIN;
1858                break;
1859            }

1860            cleanup_rbuf(sk);
1861            release_sock(sk);
1862            sk->socket->flags |= SO_WAITDATA;
1863            schedule();
1864            sk->socket->flags &= ~SO_WAITDATA;
1865            sk->inuse = 1;

1866            if (current->signal & ~current->blocked)
1867            {
1868                copied = -ERESTARTSYS;
1869                break;
1870            }
1871            continue;
```

如果了解应用程序编程，我们知道对 TCP 协议使用 read 之类的读取函数时，应该检查其返回值，我们不能期望一定会读取到我们所要求的数据量，由于 TCP 协议是基于流的，所以未读取的数据，我们下次读取也不会出现问题，如果应用程序一定要读取一定数量的数据后，方可进行下一步的处理，则可以在应用程序中实现为一个简单的循环，直到读取到所要求的数据量后才退出，只要在循环条件中作适当的检查即可。

从以上代码的第一个语句即可看出其中的原因，对于 TCP 协议，如果已经读取了部分数据，暂且又无数据可读时，其直接返回已读取的数据量，而无视上层所要求的数据量到底是多少。当然如果一个字节都未读取，其将根据 NON_BLOCK 标志位是否设置来决定到底是立刻返回还是睡眠等待。还有一个例外就是，如果在读取的过程中发生了错误，则即便连一个字节都未读取，也将返回（此时使用的是 break 语句，因为跳出最外部的 while 循环后，还需要作一些处理）。如果既没有读取到数据，期间也没有发生错误，则再次检查以下套接字的状态，如果状态变为 TCP_CLOSE，则表示如果当前没有数据可读取，以后已不会有了，此时必须立刻返回，已经没有其他准备工作需要进行了（即不必使用 break 语句返回，而是直接使用 return 语句）。从 TCP 协议整个上下文来看，sock 结构 done 字段设置为 1，则表示对应套接字接收通道已关闭（或完全关闭：TCP_CLOSE;或者半关闭：RCV_SHUTDOWN，且接收队列（注意是接收队列，其他队列可能有尚未处理的数据包）中所有数据已被上层读取。以下代码实现的含义即如此。

```
1840            if (sk->state == TCP_CLOSE)
1841            {
1842                if (!sk->done)
1843                {
1844                    sk->done = 1;
1845                    break;
1846                }
1847                copied = -ENOTCONN;
1848                break;
```

```
1849            }

1850            if (sk->shutdown & RCV_SHUTDOWN)
1851            {
1852                sk->done = 1;
1853                break;
1854            }
```

其下即是对 NON_BLOCK 标志位的检查，如果该标志位被设置，则应上层需要，可以简单返回，即便连一个字节都未读取到。

如果执行到此处，则表示尚未读取一个字节，套接字状态正常（可以进行数据读取），没有设置 NON_BLOCK 标志，则下面将要进行的工作无非是清除 receive_queue 队列中数据已被读取完的数据包（注意，没有读取到数据，并非表示 receive_queue 队列为空，receive_queue 队列中有数据包，但数据包中包含的都是重复数据也是可能情况之一），然后从其他队列中移动数据包到 receive_queue 队列中（注意，诚如前文对系统三个队列的讨论，应用程序数据只可以从 receive_queue 队列数据包中读取，如果 receive_queue 队列中无可用数据包，则需要调用相关函数从其他队列中将数据包移动到 receive_queue 队列中。release_sock 函数的作用即是将 sock 结构 back_log 队列中缓存的数据包尽可能的转移到 receive_queue 队列中。这儿需要注意的是对 schedule 进程调度函数的调用。由于之前当前执行进程状态已经被设置为 TASK_INTERRUPTIBLE,所以一旦调用 schedule 将本进程调度出去，则只有中断操作将本进程状态设置为 TASK_RUNNING 方可被再次调度执行。对于中断，也分信号中断和内部中断。信号中断是硬中断，此时的处理是退出 tcp_read 函数，即便没有读取一个字节，注意此时的返回值为 ERESTARTSYS，该返回值提示上层程序可以重新发送 read 系统调用进行读取。对于内部中断，则是调用能够诸如 wake_up_interruptible 函数进行的，该函数将相关进程状态重新设置为 TASK_RUNNING，使得进程具备调度的资格。

对于内部中断的情况，关键是中断调用点，我们可以想见，一旦 receive_queue 队列中加入了新的数据包，则可以使用内部中断将这个进程唤醒，继续进行处理，tcp_data 函数（被 tcp_rcv 函数调用，而 tcp_rcv 函数又被 release_sock 函数调用）完成数据包向 receive_queue 队列的加入，该函数在向 receive_queue 队列加入一个新的数据包后，调用 sock 结构 data_ready 函数指针指向的回调函数，data_ready 在 inet_create(af_inet.c)函数中创建一个新的 sock 结构时（socket 系统调用底层响应函数）被初始化为 def_callback2 函数，def_callback2 函数实现为：

```
435  static void def_callback2(struct sock *sk,int len)
436  {
437      if(!sk->dead)
438      {
439          wake_up_interruptible(sk->sleep);
440          sock_wake_async(sk->socket, 1);
441      }
442  }
```

当前进程被重新调度后，检查被唤醒的原因，如果是硬中断，则返回 ERESTARTSYS，跳出外部 while 循环，从而退出 tcp_read 函数，否则再一次调转到（continue 语句）外部 while 循环起始处，从 receive_queue 队列中取数据包，进入 do-while 模块进行处理判断。

对于代码中对 schedule 函数的调用，主要的目的是与内核其他进程共享 CPU，以便 CPU 在网络栈代码中处理时间过长，另外移动数据包的进程也需要得到执行。

分析完第一种情况，下面分析第二种情景：找到一个可读取数据的数据包，此时将跳转到 found_ok_skb 标志符所表示的模块处执行，该模块代码如下。该模块完成的主要工作将是拷贝可读数据到用户缓冲区中，并更新内核相关跟踪变量。

```
1872        found_ok_skb:
1873            /*
1874             *  Lock the buffer. We can be fairly relaxed as
1875             *  an interrupt will never steal a buffer we are
1876             *  using unless I've missed something serious in
1877             *  tcp_data.
1878             */

1879            skb->users++;

1880            /*
1881             *  Ok so how much can we use ?
1882             */

1883            used = skb->len - offset;
1884            if (len < used)
1885                used = len;
1886            /*
1887             *  Do we have urgent data here?
1888             */
```

这段代码首先将 skb 结构 users 字段加 1，防止在使用该数据包时，内核其它地方被释放。实际上，在内核代码中很多结构都包含这样一个字段，用于保护该结构。之后代码计算该数据包中可读取的数据量。offset 字段在前面部分被初始化为可读取数据的起始偏移量，skb->len 表示的是数据包的长度，也可以将其认为是最后一个字节的偏移量，二者相减得到的即可读取的字节数。如果上层要求读取的字节数小于此处提供的字节数，则只读取要求的字节数。used 变量最后初始化为本次需要读取的字节数。

```
1889            if (sk->urg_data)
1890            {
```

当数据包中包含紧急数据时，对应 sock 结构的 urg_data 字段被赋值为紧急数据字节（注意本版本将紧急数据处理为一个字节），即 sk->urg_data 不为 0。

```
                /* The applellation 'urg_offset' is not appropriate, may be 'urg_data_len'
                   would be more suitable.*/

1891                unsigned long urg_offset = sk->urg_seq - *seq;
```

注意 urg_offset 变量被初始化为以 copied_seq 为起点的偏移量，此时和 used 变量具有相同的起点，可以直接进行比较，如 urg_offset<used，则表示紧急数据嵌入在要读取的数据中间，此时需要根据 sock 结构中 urginline 标志位进行不同的处理，见下文分析。

```
                /* 如果 urg_offset=0， 表示紧急数据正好在上次读取中被传送完，
                * 还剩最后一个字节的紧急数据。
                 * 此处的处理方式是忽略该字节。
                 */
1892                    if (urg_offset < used)
1893                    {
```

此处检查紧急数据位置是否在可读取数据中间，如果是， 则需要进行隔离。上文已经交待，如果在读取普通数据的过程中，碰到紧急数据，则读取普通数据到该紧急字节后中止读取。

```
                /*
                * In-line urgent data
                *To force urgent data to be treated the same as normal data.
                *This allowsyou to poll for both urgent and normal data with a single SocketRecv()
                * command. Of course, setting this option means that you will not receive
                * urgent data as soon as possible.
                */
1894                        if (!urg_offset)
1895                        {
1896                            if (!sk->urginline)
1897                            {
1898                                ++*seq;
1899                                offset++;
1900                                used--;
1901                            }
1902                        }
```

urg_offset 等于 0 表示紧急数据是第一个可读的字节，此时根据 sock 结构 urginline 标志位进行不同的处理，如果 sk->urginline 为 0，则简单跳过该字节，读取其之后其它普通数据。注意这种简单跳过不同造成该紧急数据字节的丢失，因为网络栈代码在监测到 TCP 首部中 URG 标志位被设置时，在 sk->urg_data 字段中已经保存了该紧急数据字节（tcp_urg 函数完成），此处跳过的影响紧紧是紧急数据会被延迟提交给上层，紧急数据变得不再“紧急”而已。如果 urginline 设置为 1，则当作普通数据处理。

```
1903                        else
1904                            used = urg_offset;
1905                    }
1906                }
```

此处表达的意思是如果 urg_offset 处于要读取的普通数据中间，则只读取到该紧急数据为止，不可跨越读取。结合 while 循环开始处的如下代码，可以理解为紧急数据需要另行进行处理。

```
1807                if (copied && sk->urg_data && sk->urg_seq == *seq)
```

```
1808                    break;
```

```
1907            /*
1908             *  Copy it - We _MUST_ update *seq first so that we
1909             *  don't ever double read when we have dual readers
1910             */

1911        *seq += used;
```

更新读取序列号，如果非 PEEK 方式，则更新的就是 copied_seq 变量，该变量表示当前读取的最后一个字节的序列号，用于跟踪上层读取进度。

```
1912            /*
1913             *  This memcpy_tofs can sleep. If it sleeps and we
1914             *  do a second read it relies on the skb->users to avoid
1915             *  a crash when cleanup_rbuf() gets called.
1916             */

1917            memcpy_tofs(to,((unsigned char *)skb->h.th) +
1918                skb->h.th->doff*4 + offset, used);
1919            copied += used;
1920            len -= used;
1921            to += used;

1922            /*
1923             *  We now will not sleep again until we are finished
1924             *  with skb. Sorry if you are doing the SMP port
1925             *  but you'll just have to fix it neatly ;)
1926             */

1927            skb->users --;

1928            if (after(sk->copied_seq,sk->urg_seq))
1929                sk->urg_data = 0;
```

这个语句的含义为紧急数据已经被处理，此时对 sk->urg_data 字段进行清除。在有紧急数据时，该字段表示的是紧急数据本身。

```
1930            if (used + offset < skb->len)
1931                continue;
```

如果该数据包中数据都被处理完，则继续下面的处理，否则跳过下面代码进行再一次循环。

```
1932            /*
1933             *  Process the FIN.
1934             */

1935            if (skb->h.th->fin)
1936                goto found_fin_ok;
```

如果数据包中数据此次被读取完，则检查该数据包是否同时是一个 FIN 数据包，如是，则跳转到 found_fin_ok 标志符处的模块进行处理。注意在前文分析内部 do-while 循环模块时也有跳转到 found_fin_ok 处模块进行执行的情况，不过与此处不同，此处是刚处理完一个有数据的数据包后，而前文是对重复数据包进行的处理。

```
1937            if (flags & MSG_PEEK)
1938                continue;
1939            skb->used = 1;
1940            continue;
```

将 skb->used 字段设置为 1，表示该数据包中所有数据均已经得到处理（已被读取），可以进行释放。

下面进入第三部分分析，即如果数据包是一个 FIN 数据包，此时将进入 found_fin_ok 处模块进行执行。

```
1941        found_fin_ok:
1942            ++*seq;
```

加 1 的含义是对 FIN 标志位所占用的一个序列号进行计数。

```
1943            if (flags & MSG_PEEK)
1944                break;
```

如果紧急是 PEEK，则目的是预读数据，无需对 FIN 数据包进行处理，此时直接跳出 while 循环即可，等到真正进行数据读取时，会进行相应处理的。

```
1945            /*
1946             *  All is done
1947             */

1948            skb->used = 1;
1949            sk->shutdown |= RCV_SHUTDOWN;
1950            break;
```

从上文分析来看，有两种情况会跳转到 found_fin_ok 模块处进行执行，其一即这是一个重复数据包；其二即数据包中所有数据已被读取。无论哪种情况，数据包都可以进行释放。此处将

skb->used 字段再次赋值为 1，有些多余，实际上以上两种情况下，该字段已经被初始化为 1。另外既然是接收到了 FIN 数据包，则表示远端已经关闭了其发送通道，相对本地而言，即本地接收通道已被关闭，此处相应的设置 sock 结构 shutdown 字段，最后跳出 while 循环。即主要处理到 FIN 数据包，无论读取了多少字节，都跳出 while 循环，因为已经没有数据可供读取了，FIN 数据包标志了接收数据的结束。说到此处，有一点需要提请注意，receive_queue 中数据包是按序列号进行排列的，所以 FIN 数据包一定是排在 receive_queue 队列的最后面，不存在 FIN 数据包后还有其它未读取数据包的情况。

```
1951            }//end of while
1952            remove_wait_queue(sk->sleep, &wait);
1953            current->state = TASK_RUNNING;

1954            /* Clean up data we have read: This will do ACK frames */
1955            cleanup_rbuf(sk);
1956            release_sock(sk);
1957            return copied;
1958        }//end of tcp-read routine
```

函数结尾处的处理，即将本进程从可能的睡眠队列中删除，重新设置进程状态为 TASK_RUNNING.调用 cleanup_rbuf 处理 receive_queue 中已被处理的数据包(skb->used=1),最后调用 release_sock 函数向 receive_queue 队列中加入新的数据包。

```
1959        /*
1960         *  State processing on a close. This implements the state shift for
1961         *  sending our FIN frame. Note that we only send a FIN for some
1962         *  states. A shutdown() may have already sent the FIN, or we may be
1963         *  closed.
1964         */
/* 执行关闭操作时（用户程序调用 close 函数），系统调用该函数根据目前连接所处的状态进行
 * 相应处理，如是否需要发送 FIN。
 */
1965        static int tcp_close_state(struct sock *sk, int dead)
1966        {
1967            int ns=TCP_CLOSE;
1968            int send_fin=0;
1969            switch(sk->state)
1970            {
1971                case TCP_SYN_SENT:  /* No SYN back, no FIN needed */
1972                    break;
1973                case TCP_SYN_RECV:
1974                case TCP_ESTABLISHED:   /* Closedown begin */
1975                    ns=TCP_FIN_WAIT1;
1976                    send_fin=1;
1977                    break;
```

```
1978                case TCP_FIN_WAIT1: /* Already closing, or FIN sent: no change */
1979                case TCP_FIN_WAIT2:
1980                case TCP_CLOSING:
1981                        ns=sk->state;
1982                        break;
1983                case TCP_CLOSE:
1984                case TCP_LISTEN:
1985                        break;
1986                case TCP_CLOSE_WAIT:    /* They have FIN'd us. We send our FIN and
1987                                           wait only for the ACK */
1988                        ns=TCP_LAST_ACK;
1989                        send_fin=1;
1990        }

1991        tcp_set_state(sk,ns);

1992        /*
1993          *     This is a (useful) BSD violating of the RFC. There is a
1994          *     problem with TCP as specified in that the other end could
1995          *     keep a socket open forever with no application left this end.
1996          *     We use a 3 minute timeout (about the same as BSD) then kill
1997          *     our end. If they send after that then tough - BUT: long enough
1998          *     that we won't make the old 4*rto = almost no time - whoops
1999          *     reset mistake.
2000          */
2001        if(dead && ns==TCP_FIN_WAIT2)
2002        {
2003                int timer_active=del_timer(&sk->timer);
2004                if(timer_active)
2005                        add_timer(&sk->timer);
2006                else
2007                        reset_msl_timer(sk, TIME_CLOSE, TCP_FIN_TIMEOUT);
2008        }

2009        return send_fin;
2010    }
```

tcp_close_state 函数在本地套接字(半）关闭时被调用（tcp_shutdown, tcp_close),该函数实现逻辑也较为简单，即根据套接字的当前状态决定是否发送 FIN 数据包，然后对状态进行更新；所依据的基本思想是，FIN 数据包的发送是为了断开已有的连接（关闭相应的发送通道），如果之前通道尚未建立，则无需发送 FIN 数据包；如果之前成功建立的通道，则在关闭时需要发送一个 FIN 数据包进行关闭操作。

TCP 协议有以下两个状态需要明确发送 FIN 数据包：TCP_ESTABLISHED, TCP_CLOSE_WAIT. 对于实际上 TCP_SYN_RECV 状态也需要发送。TCP_SYN_RECV 状态的进入表示本地接收到

远端一个 SYN 数据包，而且本地对此已经进行了确认，但尚未接收到远端的确认数据包。从这个角度而言，此时远端很可能已经接收到本地发送的 ACK 数据包，并将其状态更新为 TCP_ESTABLISHED,所以需要本地发送一个 FIN 数据包进行关闭操作。对于 TCP 协议其它状态均无需发送 FIN 数据包，只需要根据要求更新套接字状态即可。

函数结尾处理的是远端主机长时间不进行关闭操作，从而使得本地套接字永远滞留在 TCP_FIN_WAIT2 状态的情况。TCP_FIN_WAIT2 状态表示本地已经成功进行了发送通道关闭操作，正在等待远端发送 FIN 数据包进行关闭操作。一般而言，除某些特殊应用外，在一方进行关闭操作后，另一方紧接着也应进行关闭操作，个别情况下会有半关闭，但处于半关闭的时间不应（也不会）太长，否则极有可能远端主机发生问题，如果不对这种情况进行处理，则本地将有可能永远处于半关闭状态，但又无法与远端取得联系（因为本地已经关闭发送通道：注意半关闭只能是关闭发送通道），将造成系统资源浪费。所以一旦本地进入 TCP_FIN_WAIT2 状态后，即启动一个定时器，如果定时器超时后，仍然处于半关闭状态，则认为与对方完全断开，即强行设置进入 TCP_CLOSE 状态，并释放相关系统资源。函数结尾处表达的含义即是如此：如果已经启动了定时器，则直接退出，否则就启动这样一个定时器。

```
2011/*
2012        *    Send a fin.
2013        */

2014        static void tcp_send_fin(struct sock *sk)
2015        {
2016                struct proto *prot =(struct proto *)sk->prot;
2017                struct tcphdr *th =(struct tcphdr *)&sk->dummy_th;
2018                struct tcphdr *t1;
2019                struct sk_buff *buff;
2020                struct device *dev=NULL;
2021                int tmp;

2022                release_sock(sk); /* in case the malloc sleeps. */

2023                buff = prot->wmalloc(sk, MAX_RESET_SIZE,1 , GFP_KERNEL);
2024                sk->inuse = 1;

2025                if (buff == NULL)
2026                {
2027                        /* This is a disaster if it occurs */
2028                        printk("tcp_send_fin: Impossible malloc failure");
2029                        return;
2030                }

2031                /*
2032                  *    Administrivia
```

```
2033            */

2034            buff->sk = sk;
2035            buff->len = sizeof(*t1);
2036            buff->localroute = sk->localroute;
2037            t1 =(struct tcphdr *) buff->data;

2038            /*
2039             *  Put in the IP header and routing stuff.
2040             */

2041            tmp = prot->build_header(buff,sk->saddr, sk->daddr, &dev,
2042                                     IPPROTO_TCP, sk->opt,
2043                                     sizeof(struct tcphdr),sk->ip_tos,sk->ip_ttl);
2044            if (tmp < 0)
2045            {
2046                  int t;
2047                  /*
2048                   *  Finish anyway, treat this as a send that got lost.
2049                   *  (Not good).
2050                   */

2051                  buff->free = 1;
2052                  prot->wfree(sk,buff->mem_addr, buff->mem_len);
2053                  sk->write_seq++;
2054                  t=del_timer(&sk->timer);
2055                  if(t)
2056                        add_timer(&sk->timer);
2057                  else
2058                        reset_msl_timer(sk, TIME_CLOSE, TCP_TIMEWAIT_LEN);
2059                  return;
2060            }

2061            /*
2062             *  We ought to check if the end of the queue is a buffer and
2063             *  if so simply add the fin to that buffer, not send it ahead.
2064             */

2065            t1 =(struct tcphdr *)((char *)t1 +tmp);
2066            buff->len += tmp;
2067            buff->dev = dev;
2068            memcpy(t1, th, sizeof(*t1));
```

/* 注意使用的是 write-seq，因为即便是 FIN，也有可能被缓存，
* 如果有未发送出去的数据包。

```
        * 则 FIN 的发送需要缓存到这些数据包之后。
        */
2069            t1->seq = ntohl(sk->write_seq);
2070            sk->write_seq++;
2071            buff->h.seq = sk->write_seq;
2072            t1->ack = 1;
2073            t1->ack_seq = ntohl(sk->acked_seq);
2074            t1->window = ntohs(sk->window=tcp_select_window(sk));
2075            t1->fin = 1;
2076            t1->rst = 0;
2077            t1->doff = sizeof(*t1)/4;
2078            tcp_send_check(t1, sk->saddr, sk->daddr, sizeof(*t1), sk);

2079            /*
2080             * If there is data in the write queue, the fin must be appended to
2081             * the write queue.
2082             */

2083            if (skb_peek(&sk->write_queue) != NULL)
2084            {
2085                buff->free = 0;
2086                if (buff->next != NULL)
2087                {
2088                    printk("tcp_send_fin: next != NULL\n");
2089                    skb_unlink(buff);
2090                }
2091                skb_queue_tail(&sk->write_queue, buff);
2092            }
2093            else
2094            {
                    /* 每次真正发送数据包时方才更新 sent-seq。*/
2095                sk->sent_seq = sk->write_seq;
2096                sk->prot->queue_xmit(sk, dev, buff, 0);
2097                reset_xmit_timer(sk, TIME_WRITE, sk->rto);
2098            }
2099        }
```

tcp_send_fin 函数完成的功能单一，即创建一个 FIN 数据包，发送到远端进行关闭操作。函数中大部分代码都不难理解，此处不再叙述。需要注意的一点是如果该套接字尚存在之前数据未发送出去，则将此 FIN 数据包插入到队列最后，只当这些之前写入的数据都发送到远端后，方才发送该 FIN 数据包进行关闭操作。

另外一点是对 sock 结构中各表示序列号变量的更新，此处需要着重说明一下，在涉及到数据包写入和发送时都会发现这样的序列号变量的更新而且易于混淆。我们可以从内核维护的几个发

送队列为基础理解这几个涉及数据包发送的序列号变量（均定义在 sock 结构中）。
write_seq 变量对应 write_queue 写入队列；sent_seq 变量对应 send_head, send_tail 表示的发送队列。内核将上层（网络层之上-即应用层）写入的数据和网络层向下层发送的数据区分对待。通常情况下，上层写入的数据会被网络层直接发往下层（网络层之下-即传输层））进行处理，换句话说，应用层写入的数据经过封装后将直接进入 send_head,send_tail 表示的发送队列（不经过 write_queue 写入队列），此时 write_seq 和 sent_seq 两个变量将始终相同，但是如果应用层写入的速度大于网络层（以及下层和网络传输介质）可以处理的速度，则数据需要先在 write_queue 进行缓存，此时 write_seq 就大于 sent_seq。write_seq 队列中数据包表示应用层写入的，但尚未发送出去的数据包；send_head,send_tail 表示的队列表示已经发送出去（此处发送出去并非一定是指已经发送到传输介质上，有可能数据包还缓存在硬件缓冲队列中，但一旦交给硬件，我们即认为已经发送出去）并等待 ACK 的数据包，所以 send_head,send_tail 表示的队列又称为重发队列。TCP 协议发生超时重发时，即从该队列中取数据包重新发送。
从以上分析，对本地要发送的一个普通数据包创建 TCP 首部时，我们可以看到使用的始终将是 write_seq, 只当对一个数据包真正进行发送时，才更新 sent_seq。个别特殊情况下（如窗口探测数据包）使用 sent_seq 进行 TCP 首部的创建。

```
2100    /*
2101     *  Shutdown the sending side of a connection. Much like close except
2102     *  that we don't receive shut down or set sk->dead=1.
2103     */

2104    void tcp_shutdown(struct sock *sk, int how)
2105    {
2106        /*
2107         *  We need to grab some memory, and put together a FIN,
2108         *  and then put it into the queue to be sent.
2109         *      Tim MacKenzie(tym@dibbler.cs.monash.edu.au) 4 Dec '92.
2110         */

2111        if (!(how & SEND_SHUTDOWN))
2112            return;

2113        /*
2114         *  If we've already sent a FIN, or it's a closed state
2115         */

            //对应如下这些状态，均无需作进一步处理，直接返回即可。
2116        if (sk->state == TCP_FIN_WAIT1 ||
2117            sk->state == TCP_FIN_WAIT2 ||
2118            sk->state == TCP_CLOSING ||
2119            sk->state == TCP_LAST_ACK ||
2120                sk->state == TCP_TIME_WAIT ||
2121                sk->state == TCP_CLOSE ||
2122                sk->state == TCP_LISTEN
```

```
2123              )
2124          {
2125                  return;
2126          }
2127          sk->inuse = 1;

2128          /*
2129           * flag that the sender has shutdown
2130           */

2131          sk->shutdown |= SEND_SHUTDOWN;

2132          /*
2133           *    Clear out any half completed packets.
2134           */

2135          if (sk->partial)
2136                  tcp_send_partial(sk);

2137          /*
2138           *    FIN if needed
2139           */

2140          if(tcp_close_state(sk,0))
2141                  tcp_send_fin(sk);

2142          release_sock(sk);
2143      }
```

tcp_shutdown 函数实现为进行半关闭操作。借助于第一章中对 TCP 协议各种状态的介绍，读者不难理解本函数实现代码。注意函数尾部调用 tcp_close_state 函数，根据其返回值决定是否发送 FIN 数据包。另外如果 sock 结构 partial 字段还缓存有合并数据包，则将其立刻发送出去。sock 结构 partial 字段是一个 sk_buff 结构指针，其并非是一个数据包队列，仅仅指向一个数据包，这个指针字段的作用是对小量数据流进行合并。因为 TCP 协议是面向流的，即数据之间不存在界限，所以可以对应用层前后发送的分段小量数据进行合并，从而减少发送到介质上的数据包数量，增加传输介质使用率和传输效率。实现上，如果一次应用层写入的数据量较小时，而且应用层并非急于将数据发送给对方，则网络层处理上将分配一个最大容量（MTU）的数据包，从而收集本次及其后写入的小量数据，最后一次性发送出去。如果在进行关闭操作时，partial 指针指向这样的一个数据包，由于应用层已经进行关闭操作，表示已经没有数据可以积累，则可以现在将其发送出去了，函数中完成的工作即是如此。

```
2144      static int
2145      tcp_recvfrom(struct sock *sk, unsigned char *to,
2146                   int to_len, int nonblock, unsigned flags,
```

```
2147                    struct sockaddr_in *addr, int *addr_len)
2148    {
2149            int result;

2150            /*
2151             *    Have to check these first unlike the old code. If
2152             *    we check them after we lose data on an error
2153             *    which is wrong
2154             */

2155            if(addr_len)
2156                    *addr_len = sizeof(*addr);
2157            result=tcp_read(sk, to, to_len, nonblock, flags);

2158            if (result < 0)
2159                    return(result);

2160            if(addr)
2161            {
2162                    addr->sin_family = AF_INET;
2163                    addr->sin_port = sk->dummy_th.dest;
2164                    addr->sin_addr.s_addr = sk->daddr;
2165            }
2166            return(result);
2167    }
```

tcp_recvfrom 函数调用 tcp_read 函数完成数据的读取，由于其本身需要检查是否需要返回远端地址，在读取相应数据后，还需要进行地址的复制工作，地址来源非常直接，由于 TCP 协议是面向连接的，底层结构在连接建立之时就维护远端地址，所以此处的工作直接从 sock 结构相关字段复制即可。

```
2168    /*
2169     *    This routine will send an RST to the other tcp.
2170     */

2171    static void tcp_reset(unsigned long saddr, unsigned long daddr, struct tcphdr *th,
2172              struct proto *prot, struct options *opt, struct device *dev, int tos, int ttl)
2173    {
2174            struct sk_buff *buff;
2175            struct tcphdr *t1;
2176            int tmp;
2177            struct device *ndev=NULL;

2178            /*
```

```
2179            *    Cannot reset a reset (Think about it).
2180            */

2181            if(th->rst)
2182                    return;

2183            /*
2184             * We need to grab some memory, and put together an RST,
2185             * and then put it into the queue to be sent.
2186             */

2187            buff = prot->wmalloc(NULL, MAX_RESET_SIZE, 1, GFP_ATOMIC);
2188            if (buff == NULL)
2189                    return;

2190            buff->len = sizeof(*t1);
2191            buff->sk = NULL;
2192            buff->dev = dev;
2193            buff->localroute = 0;

2194            t1 =(struct tcphdr *) buff->data;

2195            /*
2196             *    Put in the IP header and routing stuff.
2197             */

2198            tmp = prot->build_header(buff, saddr, daddr, &ndev, IPPROTO_TCP, opt,
2199                                     sizeof(struct tcphdr),tos,ttl);
2200            if (tmp < 0)
2201            {
2202                    buff->free = 1;
2203                    prot->wfree(NULL, buff->mem_addr, buff->mem_len);
2204                    return;
2205            }

2206            t1 =(struct tcphdr *)((char *)t1 +tmp);
2207            buff->len += tmp;
2208            memcpy(t1, th, sizeof(*t1));

2209            /*
2210             *    Swap the send and the receive.
2211        */

2212            t1->dest = th->source;
```

```
2213            t1->source = th->dest;
2214            t1->rst = 1;
2215            t1->window = 0;

  /* 当需要发送一个 RST 数据包给对方时，
  * 此时根据引起该 RST 的数据包的 ACK 标志位是否设置情况进行对应的处理：
   * 如果对方 ACK=1， 则将 seq 设置为对方 TCP 首部中 ack-seq 字段值，
  * 否则设置 seq=0，ack-seq=th->seq+data_len.
   */

2216            if(th->ack)
2217            {
2218                    t1->ack = 0;
2219                    t1->seq = th->ack_seq;
2220                    t1->ack_seq = 0;
2221            }
2222            else
2223            {
2224                    t1->ack = 1;
2225                    if(!th->syn)
2226                            t1->ack_seq=htonl(th->seq);
2227                    else
2228                            t1->ack_seq=htonl(th->seq+1);
2229                    t1->seq=0;
2230            }

2231            t1->syn = 0;
2232            t1->urg = 0;
2233            t1->fin = 0;
2234            t1->psh = 0;
2235            t1->doff = sizeof(*t1)/4;
2236            tcp_send_check(t1, saddr, daddr, sizeof(*t1), NULL);
2237            prot->queue_xmit(NULL, ndev, buff, 1);
2238            tcp_statistics.TcpOutSegs++;
2239    }
```

tcp_reset 函数功能单一，即向对方发送一个 RST 复位数据包。复位数据包的效果是促使对方断开与本地的连接，如果需要的话，对方可以重新发起连接请求；或者是如果对方就是在请求连接，则表示本地无对方请求的服务，对方在接收到该 RST 数据包，应该做出相应处理，不可“坚持不懈”的对本地进行请求。

本函数唯一需要说明的地方即对 RST 数据包 TCP 首部中 ack 标志位，seq 和 ack_seq 字段的设置。如果对方在请求本地未提供的服务（即此 RST 数据包是对对方 SYN 数据包的应答），则将 ack_seq 字段设置为对方想要的序列号，至于本地序列号则简单设置为 0，反正以后也犯不着与

对方”啰嗦“了，所以也就无关紧要。当然实际上对于 ack_seq 应答序列号的设置也可以随意，但由于对方缓存了这个 SYN 数据包，需要本地 ACK 一下，所以正确设置这个 ack_seq 字段还是必要的。当然如果不是请求连接的情况，那么通常情况下引起发送 RST 的数据包本身是一个 ACK 数据包，此时就需要正确的设置本地 seq 字段，而对于 ack_seq 字段则无关紧要，因为既然本地发送一个 RST 数据包给对方，摆明了不想再从对方接收到任何数据。注意对于 TCP 协议，一旦连接建立后，直到最后发送的一个数据包，期间相互交往的所有的数据包中 TCP 首部 ACK 标志位都被设置为 1。所以上文说，如果不是一个 SYN 请求连接数据包，则可以认定为一个 ACK 数据包。

下一个介绍的函数是 tcp_options，该函数专门用于处理 TCP 选项，所以在介绍该函数之前，先讨论一下 TCP 协议定义的如下几个选项。这样将大大有助于对于 tcp_options 函数的理解。

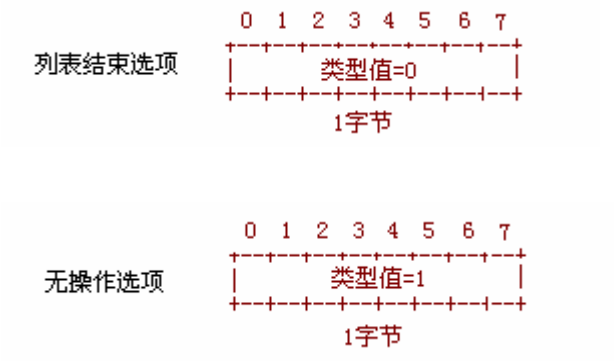
如同IP协议一样，在正常的首部之后，TCP协议也可以附带选项。最初的TCP协议规范（RFC793）只定义三种选项：列表结束选项，无操作选项，MSS 选项。新的协议规范（RFC1323）定义了额外的选项，如窗口变化因子选项，时间戳选项，这些选项只在某些新的网络栈代码中才有实现，当然对于本书分析的网络栈代码只实现了最初的三个选项，所以下面这对这三个选项进行说明，读者可察看 RFC1323 了解更多的 TCP 选项内容。

1〉 列表结束选项（End of Option List），该选项标志着所有 TCP 选项的结束，选项类型值为 0。

2〉 无操作选项（No Operation），该选项的主要作用用于填充，从而使得选项长度始终保持为 4 字节的倍数，或者对其下一选项在 4 字节边界上。该选项对应的类型值为 1。

3〉 最大 TCP 报文长度选项，该选项用于在建立连接时向对方通报本地可接受的最大 TCP 报文长度，用于节制对方发送的最大报文大小。该选项只使用在连接建立的过程中。

如下显示了以上三种选项具体格式。



最大报文长度选项


```
2240    /*
2241     *  Look for tcp options. Parses everything but only knows about MSS.
2242     *      This routine is always called with the packet containing the SYN.
2243     *      However it may also be called with the ack to the SYN.  So you
2244     *      can't assume this is always the SYN.  It's always called after
2245     *      we have set up sk->mtu to our own MTU.
2246     *
2247     *  We need at minimum to add PAWS support here. Possibly large windows
2248     *  as Linux gets deployed on 100Mb/sec networks.
2249     */

2250    static void tcp_options(struct sock *sk, struct tcphdr *th)
2251    {
2252        unsigned char *ptr;
2253        int length=(th->doff*4)-sizeof(struct tcphdr);
2254        int mss_seen = 0;

2255        ptr = (unsigned char *)(th + 1);

2256        while(length>0)
2257        {
2258            int opcode=*ptr++;
2259            int opsize=*ptr++;
2260            switch(opcode)
2261            {
2262                case TCPOPT_EOL:
2263                    return;
2264                case TCPOPT_NOP:    /* Ref: RFC 793 section 3.1 */
2265                    length--;
2266                    ptr--;          /* the opsize=*ptr++ above was a mistake */
2267                    continue;

2268                default:
2269                    if(opsize<=2)  /* Avoid silly options looping forever */
```

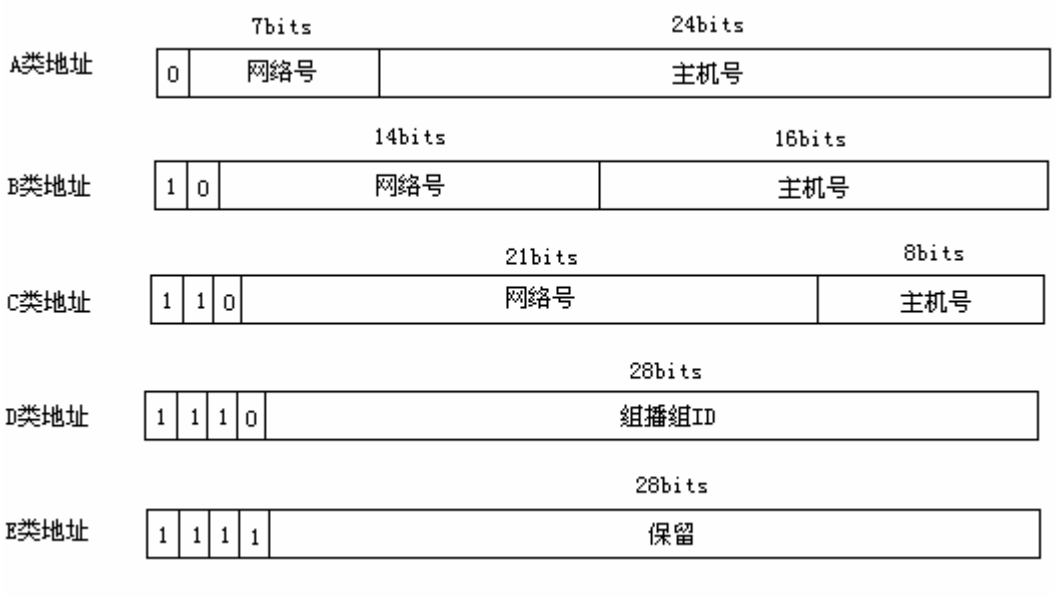
```
2270                            return;
2271                        switch(opcode)
2272                        {
2273                            case TCPOPT_MSS:
2274                                if(opsize==4 && th->syn)
2275                                {
2276                                    sk->mtu=min(sk->mtu,ntohs(*(unsigned short *)ptr));
2277                                    mss_seen = 1;
2278                                }
2279                                break;
2280                                /* Add other options here as people feel the urge to implement
stuff like large windows */
2281                        }
2282                        ptr+=opsize-2;
2283                        length-=opsize;
2284                    }
2285                }
2286                if (th->syn)
2287                {
2288                    if (! mss_seen)
2289                        sk->mtu=min(sk->mtu, 536);    /* default MSS if none sent */
2290                }
2291        #ifdef CONFIG_INET_PCTCP
2292                sk->mss = min(sk->max_window >> 1, sk->mtu);
2293        #else
2294                sk->mss = min(sk->max_window, sk->mtu);
2295        #endif
2296        }
```

读者对照函数之前对于 TCP 选项（及其格式）的介绍，将不难理解 tcp_options 函数中代码，此处不再叙述。需要注意的是，在建立连接时，MSS 选项是必须的，而且该选项也只能使用在连接建立的过程中。MSS 选项用于远端对本地发送数据包大小进行限制，所以在发送一个数据包时，除了检查本地 MTU 外，还需要对远端声明的 MSS 值进行考虑。 另外函数最后将 sock 结构 mss 字段初始化为 mtu 字段值（一般 mtu 小于 max_window),有些令人费解，而且函数中在处理 MSS 选项时，使用 MSS 选项值对 MTU 字段进行初始化也是不妥。这要从这两个字段所表示的含义出发进行理解。MSS 表示最大段长度，是远端加于本地的最大段大小。MTU 表示最大传输单元，是本地网络加于本地的数据包大小限制。最后发送的数据包对二者都要进行考虑。实际上 MSS 值表示的 TCP 数据负载的长度，而 MTU 表示的是 IP 首部及其负载长度，即 MSS=MTU-sizeof(ip header)-sizeof(tcp header)。二者虽然都是对数据包大小的限制，但表示的范围并不相同，函数中将二者混为一谈，实在不该。实际上，对于 1.2.13 内核版本对应的网络栈代码在所有地方都将此二者等同起来了，这是实现上的一个错误。

```
2297    static inline unsigned long default_mask(unsigned long dst)
2298    {
```

```
2299            dst = ntohl(dst);
2300            if (IN_CLASSA(dst))
2301                    return htonl(IN_CLASSA_NET);
2302            if (IN_CLASSB(dst))
2303                    return htonl(IN_CLASSB_NET);
2304            return htonl(IN_CLASSC_NET);
2305    }
```

default_mask 函数用于返回对应地址的默认网络掩码。网络地址从总体上被分为 5 类，分别称为 A，B，C，D，E。具体如下所示。



地址类别	地址范围
A	0.0.0.0 - 127.255.255.255
B	128.0.0.0 - 191.255.255.255
C	192.0.0.0 - 223.255.255.255
D	224.0.0.0 - 239.255.255.255
E	240.0.0.0 - 255.255.255.255

函数中 IN_CLASSA 等宏均定义在 include/linux/in.h 头文件中，读者可参考第一章相关内容。

```
2306    /*
2307     *  Default sequence number picking algorithm.
2308     *  As close as possible to RFC 793, which
2309     *  suggests using a 250kHz clock.
2310     *  Further reading shows this assumes 2MB/s networks.
2311  *  For 10MB/s ethernet, a 1MHz clock is appropriate.
```

```
2312        *    That's funny, Linux has one built in!    Use it!
2313        */

2314        extern inline unsigned long tcp_init_seq(void)
2315        {
2316            struct timeval tv;
2317            do_gettimeofday(&tv);
2318            return tv.tv_usec+tv.tv_sec*1000000;
2319        }
```

tcp_init_seq 函数用于在 TCP 协议连接建立时创建本地初始序列号。从实现来看，初始序列号根据当前系统时间计算而得。

```
2320        /*
2321         *    This routine handles a connection request.
2322         *    It should make sure we haven't already responded.
2323         *    Because of the way BSD works, we have to send a syn/ack now.
2324         *    This also means it will be harder to close a socket which is
2325         *    listening.
2326         */

//参数中 daddr，saddr 的理解应从远端角度出发，即实际上 daddr 表示的是本地地址，
//saddr 表示的是远端地址。opt 表示接收到的 IP 选项（如有）。
//seq 是函数调用 tcp_init_seq()的返回值，表示本地初始序列号。
//dev 表示接收该数据包（skb 参数表示）的接口设备。
2327        static void tcp_conn_request(struct sock *sk, struct sk_buff *skb,
2328                    unsigned long daddr, unsigned long saddr,
2329                    struct options *opt, struct device *dev, unsigned long seq)
2330        {
```

当 tcp_rcv 函数接收一个 SYN 连接请求数据包后，其调用 tcp_conn_request 函数进行具体处理。tcp_conn_request 函数实现的功能如同函数名，专门用于处理连接请求。该函数虽然较长，但逻辑上非常简单：其首先创建一个新的 sock 结构（这就是我们通常所说的，侦听套接字在接收到一个连接请求时，会创建一个新的套接字用于通信，其本身继续侦听其他客户端的请求）并对该 sock 结构进行初始化（该函数较长即源于初始化 sock 结构字段的代码较长）；此后发送一个应答数据包，并将新创建的 sock 结构状态设置为 TCP_SYN_RECV（实际上在初始化 sock 结构时已经进行了设置，注意侦听套接字状态仍然为 TCP_LISTEN),函数最后将该新创建的 sock 结构与请求连接数据包绑定并挂接到侦听套接字的 receive_queue 队列中。本书在前文中一再强调，侦听套接字接收队列 receive_queue 中缓存的均是请求连接数据包，不包含普通数据包。accept 系统调用即从侦听套接字 receive_queue 中取数据包，获得该数据包对应的 sock 结构，检查其状态，如果状态为 TCP_ESTABLISHED,则 accept 系统调用成功返回，否则等待该 sock 结构状态进入 TCP_ESTABLISHED.注意 tcp_conn_request 函数发送应答时，相应 sock 结构状态设置为 TCP_SYN_RECV,该 sock 结构状态转为 TCP_ESTABLISHED 是由 tcp_ack 函数完成的，tcp_ack 函数专门负责对方发送的 ACK 数据包，当监测到某个 ACK 数据包是三路握手连接过程中的完

成连接建立的应答数据包时，tcp_ack 函数会将对应该连接的本地 sock 结构状态设置为 TCP_ESTABLISHED.具体请参考下文中对于 tcp_ack 函数的相关分析。
对于 tcp_conn_request 函数中的大部分代码，读者可对照 sock 结构（net/inet/sock.h）定义理解。

```
2331        struct sk_buff *buff;
2332        struct tcphdr *t1;
2333        unsigned char *ptr;
2334        struct sock *newsk;
2335        struct tcphdr *th;
2336        struct device *ndev=NULL;
2337        int tmp;
2338        struct rtable *rt;

2339        th = skb->h.th;
```

此处将 th 变量设置为所接收数据包的 TCP 首部第一个字节地址，便于下面的处理。

```
2340        /* If the socket is dead, don't accept the connection. */
2341        if (!sk->dead)
2342        {
2343            sk->data_ready(sk,0);
2344        }
```

注意 sk 变量表示的是侦听套接字 sock 结构，如果该侦听套接字状态正常，则回调 data_ready 函数指针指向的函数（def_callback2，af_inet.c），这个回调函数会唤醒睡眠在该侦听套接字睡眠队列中的进程。一般而言，对于侦听套接字，进程在对其使用 accept 系统调用时，如果暂时无法获得连接请求套接字，则会等待于侦听套接字的睡眠队列中。本函数是处理连接请求的，或者更确切的说是处理对方发送的第一个 SYN 数据包，所以在这个函数中无法完成连接建立的三路握手全过程，连接的最终完成是在 tcp_ack 中完成的，在 tcp_ack 函数中，其在检查到一个连接完全建立后，会调用侦听套接字 sock 结构中 state_change 函数指针指向的回调函数(def_callback1, af_inet.c)，该回调函数只完成一个功能即唤醒侦听套接字睡眠队列中的进程，此时的唤醒将使得进程可以重新检查是否有可用的已经完成连接建立的套接字请求，从而从 accept 函数中返回。对于本函数此处对 data_ready 的调用有些显得多余。

```
2345        else
2346        {
2347            if(sk->debug)
2348                printk("Reset on %p: Connect on dead socket.\n",sk);
2349            tcp_reset(daddr, saddr, th, sk->prot, opt, dev, sk->ip_tos,sk->ip_ttl);
2350            tcp_statistics.TcpAttemptFails++;
2351            kfree_skb(skb, FREE_READ);
2352            return;
2353        }
```

如果侦听套接字状态已经被设置为 dead，则表示该侦听套接字不能在继续接受远端请求，即不再提供之前的服务，所以如果此时接收到一个服务请求，则回复 RST 数据包，断开连接或者称为复位连接，中断与远端的进一步交互。

```
2354        /*
2355         * Make sure we can accept more.   This will prevent a
2356         * flurry of syns from eating up all our memory.
2357         */

2358        if (sk->ack_backlog >= sk->max_ack_backlog)
2359        {
2360            tcp_statistics.TcpAttemptFails++;
2361            kfree_skb(skb, FREE_READ);
2362            return;
2363        }
```

sock 结构中 max_ack_backlog 变量表示的是最大接收的连接请求数据包个数，从实现的结果来看，表达的是 sock 结构 receive_queue 队列中可缓存的最大数据包个数，这个数据包个数包括已经完成连接请求和尚未完成连接请求的数据包。max_ack_backlog 变量是在 listen 系统调用中进行初始化的，这一点在介绍 inet_listen(af_inet.c)函数将会清楚，另外 inet_listen 函数将 max_ack_backlog 最大值限制为 5。如果连接请求个数超过已经达到最大可处理的请求数，则简单丢弃该数据包。注意这种丢弃将造成远端延迟重发请求，这是合理的，不同于发送一个 RST 数据包。简单丢弃表示请求暂时无法得到满足，在对方的“一再坚持”下，稍候可能获得服务，而发送一个 RST 数据包，则表示本地根本不提供此种服务，“再坚持”也没用，所以发送一个 RST 数据包明确的告知对方本地无此服务项，不要再发送此类服务请求，而远端接收到一个 RST 数据包后，也会停止发送服务请求数据包，并返回一个错误给上层应用程序，至于上层应用程序如何处理，则由应用程序员负责了。

接下来的这段代码较长，但完成的功能较为简单，即完成新套接字 sock 结构的创建和初始化，初始化信息主要来自于侦听套接字中已有信息，另一方面由该新创建套接字的作用决定。
以下将对其中部分值得注意的地方进行注释，其他部分代码容易理解，此处不再阐述。

```
2364        /*
2365         * We need to build a new sock struct.
2366         * It is sort of bad to have a socket without an inode attached
2367         * to it, but the wake_up's will just wake up the listening socket,
2368         * and if the listening socket is destroyed before this is taken
2369         * off of the queue, this will take care of it.
2370         */

2371        newsk = (struct sock *) kmalloc(sizeof(struct sock), GFP_ATOMIC);
2372        if (newsk == NULL)
2373        {
2374            /* just ignore the syn.   It will get retransmitted. */
2375            tcp_statistics.TcpAttemptFails++;
2376            kfree_skb(skb, FREE_READ);
2377            return;
2378        }
```

分配新套接字，如果分配失败，则简单丢弃连接请求数据包。

```
2379            memcpy(newsk, sk, sizeof(*newsk));
```

注意此处的复制，下面只需要对新套接字 sock 结构中需要进行修改的地方进行操作即可。而且这也表示了新创建套接字中主要信息来自于侦听套接字中的已有信息。

```
2380            skb_queue_head_init(&newsk->write_queue);
2381            skb_queue_head_init(&newsk->receive_queue);
2382            newsk->send_head = NULL;
2383            newsk->send_tail = NULL;
2384            skb_queue_head_init(&newsk->back_log);
2385            newsk->rtt = 0;             /*TCP_CONNECT_TIME<<3*/
2386            newsk->rto = TCP_TIMEOUT_INIT;
```

由于是连接建立期间，所以将 RTO 值设置为 TCP_TIMEOUT_INIT. 在连接建立后，RTO 值将根据相关算法有 RTT 值计算而得，这在下文中介绍 tcp_ack 函数时会有分析。

```
2387            newsk->mdev = 0;
```

sock 结构中 mdev 字段用于计算 RTO 值。

```
2388            newsk->max_window = 0;
2389            newsk->cong_window = 1;
2390            newsk->cong_count = 0;
2391            newsk->ssthresh = 0;
2392            newsk->backoff = 0;
2393            newsk->blog = 0;
2394            newsk->intr = 0;
2395            newsk->proc = 0;
2396            newsk->done = 0;
2397            newsk->partial = NULL;
2398            newsk->pair = NULL;
2399            newsk->wmem_alloc = 0;
2400            newsk->rmem_alloc = 0;
2401            newsk->localroute = sk->localroute;

2402            newsk->max_unacked = MAX_WINDOW - TCP_WINDOW_DIFF;

2403            newsk->err = 0;
2404            newsk->shutdown = 0;
2405            newsk->ack_backlog = 0;
2406            newsk->acked_seq = skb->h.th->seq+1;
2407            newsk->copied_seq = skb->h.th->seq+1;
2408            newsk->fin_seq = skb->h.th->seq;
```

此处对新创建 sock 结构中这三个序列号字段的设置非常重要。所接收数据包中 TCP 首部中 seq 字段表示其初始序列号，由于 SYN 标志位本身占据一个序列号，所以本地将从 seq+1 开始期望，acked_seq 表示本地希望从远端接收的下一个字节的序列号，copied_seq 表示本地已经送达给上

层应用的最后一个字节序列号，至于 fin_seq 表示的则是本地最后发送的 FIN 标志位所对应的序列号。

```
2409            newsk->state = TCP_SYN_RECV;
//it is very important to set this new-created sock to TCP_SYN_RECV state.
//注意此处新创建的套接字状态被设置为 TCP_SYN_RECV，熟悉 TCP 协议的读者知道该状//态是三路握手连接建立过程中服务器端在接收到一个 SYN 数据包，服务器回送一个
//SYN+ACK 数据包后进入的状态。只不过单从协议介绍中，只是简单的从表面上进行了说
//明，实际上 TCP_SYN_RECV 状态的设置是针对新创建的套接字的，侦听套接字从其起初
//创建直到其最后关闭，状态将一直是 TCP_LISTEN。

2410            newsk->timeout = 0;
2411            newsk->ip_xmit_timeout = 0;
2412            newsk->write_seq = seq;
2413            newsk->window_seq = newsk->write_seq;
2414            newsk->rcv_ack_seq = newsk->write_seq;
2415            newsk->urg_data = 0;
2416            newsk->retransmits = 0;
2417            newsk->linger=0;
2418            newsk->destroy = 0;
2419            init_timer(&newsk->timer);
2420            newsk->timer.data = (unsigned long)newsk;
2421            newsk->timer.function = &net_timer;
2422            init_timer(&newsk->retransmit_timer);
2423            newsk->retransmit_timer.data = (unsigned long)newsk;
2424            newsk->retransmit_timer.function=&retransmit_timer;
2425            newsk->dummy_th.source = skb->h.th->dest;
2426            newsk->dummy_th.dest = skb->h.th->source;

2427            /*
2428             *    Swap these two, they are from our point of view.
2429             */

2430            newsk->daddr = saddr;
2431            newsk->saddr = daddr;
```

从此处赋值，读者对照前文中对 tcp_conn_request 函数参数的分析。

```
        /* Note that the new-created data socket uses the same local port at listener socket.*/
2432            put_sock(newsk->num,newsk);
2433            newsk->dummy_th.res1 = 0;
2434            newsk->dummy_th.doff = 6;
2435            newsk->dummy_th.fin = 0;
2436            newsk->dummy_th.syn = 0;
2437            newsk->dummy_th.rst = 0;
```

```
2438            newsk->dummy_th.psh = 0;
2439            newsk->dummy_th.ack = 0;
2440            newsk->dummy_th.urg = 0;
2441            newsk->dummy_th.res2 = 0;
2442            newsk->acked_seq = skb->h.th->seq + 1;
2443            newsk->copied_seq = skb->h.th->seq + 1;
2444            newsk->socket = NULL;

2445            /*
2446             *    Grab the ttl and tos values and use them
2447             */

2448            newsk->ip_ttl=sk->ip_ttl;
2449            newsk->ip_tos=skb->ip_hdr->tos;
```

至此，这个新创建的套接字完成了主要初始化工作。

```
2450            /*
2451             *    Use 512 or whatever user asked for
2452             */

2453            /*
2454             *    Note use of sk->user_mss, since user has no direct access to newsk
2455             */

2456            rt=ip_rt_route(saddr, NULL,NULL);
```

注意 saddr 表示的远端地址，此处以此地址为目的地址在 IP 路由表查找表项，返回一个可能的路有表项。

```
2457            if(rt!=NULL && (rt->rt_flags&RTF_WINDOW))
2458                    newsk->window_clamp = rt->rt_window;
2459            else
2460                    newsk->window_clamp = 0;
```

如果返回一个可用的路有表项，则 newsk 对应 sock 结构中 window_clamp 字段的赋值即来自于该表项（rtable 结构）中 rt_window 字段。否则直接初始化为 0。sock 结构中 window_clamp 字段表示的是本地窗口的最大限制，该字段的具体使用在介绍相关函数时会进行说明。

对于本版本网络内核代码实现中，对于 MTU，MSS 值的处理有些随意，所以与现在这两个值的含义冲突，所以读者在看有关 MTU，MSS 值的初始化时，不可太计较即可。

```
2461            if (sk->user_mss)
2462                    newsk->mtu = sk->user_mss;
```

sock 结构中 user_mss 字段是由用户明确指定的 MSS 值。

```
                /* 不可思议！ MTU<MSS!!! */
2463            else if(rt!=NULL && (rt->rt_flags&RTF_MSS))
2464                    newsk->mtu = rt->rt_mss - HEADER_SIZE;
2465            else
2466            {
2467    #ifdef CONFIG_INET_SNARL    /* Sub Nets Are Local */
2468                    if ((saddr ^ daddr) & default_mask(saddr))
2469    #else
2470                    if ((saddr ^ daddr) & dev->pa_mask)
2471    #endif
2472                            newsk->mtu = 576 - HEADER_SIZE;
2473                    else
2474                            newsk->mtu = MAX_WINDOW;
2475            }

2476            /*
2477             *    But not bigger than device MTU
2478             */

2479            newsk->mtu = min(newsk->mtu, dev->mtu - HEADER_SIZE);

2480            /*
2481             *    This will min with what arrived in the packet
2482             */

2483            tcp_options(newsk,skb->h.th);
```

tcp_options 函数用于处理远端发送的 TCP 选项，对于远端发送的 SYN 数据包，其中必须包含 MSS 选项，用于向本地通报远端最大报文长度。所以 tcp_options 函数主要是从 MSS 选项中抽出 MSS 值，并对 newsk 对应的 mss 字段进行初始化。在发送数据包给远端时，本地不可发送报文长度大于此处远端声明的 MSS 值。

服务器端在进行新的套接字创建后，接下来的任务即回送一个 SYN+ACK 数据包，下面的代码即完成该 SYN+ACK 数据包的创建并调用下层模块函数发送出去。对于这段代码，将不作详细介绍，读者应该可以理解。

```
2484            buff = newsk->prot->wmalloc(newsk, MAX_SYN_SIZE, 1, GFP_ATOMIC);
2485            if (buff == NULL)
2486            {
2487                    sk->err = ENOMEM;
2488                    newsk->dead = 1;
2489                    newsk->state = TCP_CLOSE;
2490                    /* And this will destroy it */
2491                    release_sock(newsk);
2492                    kfree_skb(skb, FREE_READ);
```

```
2493            tcp_statistics.TcpAttemptFails++;
2494            return;
2495        }
```

如果空间分配失败，则将其处理为简单丢弃的情况，将新创建的套接字状态设置为 TCP_CLOSE, 且将 dead 字段设置为 1，表示该套接字已不可使用。

```
2496        buff->len = sizeof(struct tcphdr)+4;
2497        buff->sk = newsk;
2498        buff->localroute = newsk->localroute;

2499        t1 =(struct tcphdr *) buff->data;

2500        /*
2501         *  Put in the IP header and routing stuff.
2502         */

2503        tmp = sk->prot->build_header(buff, newsk->saddr, newsk->daddr, &ndev,
2504                                IPPROTO_TCP, NULL, MAX_SYN_SIZE,sk->ip_tos,sk->ip_ttl);
```

sock 结构 prot 字段初始化为 ip_build_header 函数，ip_build_header 函数完成 MAC，IP 首部的创建，返回值为 MAC，IP 首部的总长度，如果创建过程中出现错误，则返回的长度值取反（即返回一个负数）。

```
2505        /*
2506         *  Something went wrong.
2507         */

2508        if (tmp < 0)
2509        {
2510            sk->err = tmp;
2511        buff->free = 1;
2512            kfree_skb(buff,FREE_WRITE);
2513            newsk->dead = 1;
2514            newsk->state = TCP_CLOSE;
2515            release_sock(newsk);
2516            skb->sk = sk;
2517            kfree_skb(skb, FREE_READ);
2518            tcp_statistics.TcpAttemptFails++;
2519            return;
2520        }
```

如果 MAC，IP 首部创建失败，则也检查处理为丢弃数据包。

```
2521        buff->len += tmp;
2522        t1 =(struct tcphdr *)((char *)t1 +tmp);
```

```
2523            memcpy(t1, skb->h.th, sizeof(*t1));
    /* 这个语句应该放在 t1->seq = ntohl(newsk->write_seq++)语句之后！ */
2524            buff->h.seq = newsk->write_seq;
2525            /*
2526             *  Swap the send and the receive.
2527             */
2528            t1->dest = skb->h.th->source;
2529            t1->source = newsk->dummy_th.source;
2530            t1->seq = ntohl(newsk->write_seq++);
2531            t1->ack = 1;
2532            newsk->window = tcp_select_window(newsk);
2533            newsk->sent_seq = newsk->write_seq;
2534            t1->window = ntohs(newsk->window);
2535            t1->res1 = 0;
2536            t1->res2 = 0;
2537            t1->rst = 0;
2538            t1->urg = 0;
2539            t1->psh = 0;
2540            t1->syn = 1;
2541            t1->ack_seq = ntohl(skb->h.th->seq+1);
2542            t1->doff = sizeof(*t1)/4+1;
2543            ptr =(unsigned char *)(t1+1);
2544            ptr[0] = 2;
2545            ptr[1] = 4;
2546            ptr[2] = ((newsk->mtu) >> 8) & 0xff;
2547            ptr[3] =(newsk->mtu) & 0xff;
```

如上代码创建 TCP 首部，注意此处必须包含一个 MSS 选项。

```
2548            tcp_send_check(t1, daddr, saddr, sizeof(*t1)+4, newsk);
2549            newsk->prot->queue_xmit(newsk, ndev, buff, 0);
2550            reset_xmit_timer(newsk, TIME_WRITE , TCP_TIMEOUT_INIT);
```

在计算校验值后，调用 ip_queue_xmit(prot->queue_xmit 指针指向的函数)发送出去，并启动一个超时重发定时器。

```
2551            skb->sk = newsk;
```

这个语句意义关键：将这个连接请求数据包属主更改为新创建的套接字，这与上文中的相关讨论以及下文中将要进行的有关讨论思想一致。

```
2552            /*
2553             *  Charge the sock_buff to newsk.
2554             */
```

```
2555            sk->rmem_alloc -= skb->mem_len;
2556            newsk->rmem_alloc += skb->mem_len;

2557            skb_queue_tail(&sk->receive_queue,skb);
2558            sk->ack_backlog++;
2559            release_sock(newsk);
2560            tcp_statistics.TcpOutSegs++;
2561        }
```

函数结尾处的处理很‘明智’：侦听套接字只负责侦听服务请求，一旦远端发送了一个请求，侦听套接字会创建一个实际负责提供服务的新套接字，其本身继续等待其他远端的请求。既然把实际提供服务的任务交给了这个新的套接字，那么花费的代价就应该由该新套接字承担，所以将正在处理的这个连接请求数据包转交给这个新创建的套接字，自然该数据包所占据的空间也应该由新套接字承担。最后将表示连接请求的这个数据包挂接到侦听套接字 receive_queue 队列中，注意表示这个数据包的 sk_buff 结构中的 sk 指针指向的是新创建的 sock 结构，表示真正负责该数据包的套接字，侦听套接字只不过是其暂居之所，一旦连接完全建立，就与侦听套接字无任何关系！另外注意，侦听套接字 sock 结构的 receive_queue 队列中缓存的数据包均是连接请求数据包，系统调用 accept 函数传输层对应函数 tcp_accept 即从该队列中取数据包，检查取下的数据包 sk_buff 结构中 sk 指针指向的套接字状态，如果状态已经变为连接建立完成(TCP_ESTABLISHED),则返回 sk 指针指向的 sock 结构，否则等待于侦听套接字的睡眠队列中（注意是侦听套接字），tcp_ack 函数在完成三路握手连接后，会通过侦听套接字设置的回调函数唤醒侦听套接字睡眠队列中正在睡眠的进程，从而继续 accept（tcp_accept）函数的处理。这一点在前文中已有所阐述。tcp_conn_request 函数最后更新 ack_backlog 变量值，该变量表示侦听套接字 receive_queue 队列中最大可缓存的连接请求数据包个数。注意此处的数据包个数包括已完成连接建立和尚未完成连接建立的数据包。

在下文中介绍 tcp_accept 函数中读者将会体会到这一点。

函数对 release_sock(newsk)的调用显得多余。对于一个新创建的 sock 结构，其不可能进行数据包的接收，所以对其调用 release_sock 函数毫无意义。

```
2562    static void tcp_close(struct sock *sk, int timeout)
2563    {
2564        /*
2565         * We need to grab some memory, and put together a FIN,
2566         * and then put it into the queue to be sent.
2567         */

2568        sk->inuse = 1;
    //套接字是侦听套接字的情况
2569        if(sk->state == TCP_LISTEN)
2570        {
2571            /* Special case */
2572            tcp_set_state(sk, TCP_CLOSE);
2573            tcp_close_pending(sk);
2574            release_sock(sk);
```

```
2575                return;
2576        }
```

//普通数据交换套接字（如客户端套接字）

```
2577        sk->keepopen = 1;
2578        sk->shutdown = SHUTDOWN_MASK;

2579        if (!sk->dead)
2580                sk->state_change(sk);
```

因为要进行关闭操作，所以将这种状态的改变通过回调函数通知到对此套接字进行操作的进程。

```
2581        if (timeout == 0)
2582        {
```

timeout 参数表示等待关闭的时间，如果时间设置为 0，则表示立刻进行关闭，此时将对对应 sock 结构中 receive_queue 接收队列中数据包进行释放操作，因为进行应用程序要求立刻关闭，对于 receive_queue 队列中尚未读取的数据可以丢弃。不必等待这些数据都被读取完后才进行关闭操作。

```
2583                struct sk_buff *skb;

2584                /*
2585                 *  We need to flush the recv. buffs.  We do this only on the
2586                 *  descriptor close, not protocol-sourced closes, because the
2587                 *  reader process may not have drained the data yet!
2588                 */

2589                while((skb=skb_dequeue(&sk->receive_queue))!=NULL)
2590                        kfree_skb(skb, FREE_READ);
2591                /*
2592                 *  Get rid off any half-completed packets.
2593                 */

2594                if (sk->partial)
2595                        tcp_send_partial(sk);
2596        }
```

注意，在进行关闭操作时，如果 sock 结构中 partial 指针非空，则表示有一个收集数据的数据包正待发送，由于现在要关闭该套接字（关闭其发送通道），所以现在就把该数据包直接发送出去。

```
2597        /*
2598         *  Timeout is not the same thing - however the code likes
2599         *  to send both the same way (sigh).
2600         */
```

```
2601            if(timeout)
2602            {
2603                    tcp_set_state(sk, TCP_CLOSE);    /* Dead */
2604            }
```

如果 timeout 参数非 0，则简单设置状态为 TCP_CLOSE,但并不立刻发送一个 FIN 数据包。注意状态设置为 TCP_CLSOE 后，此后本地不可再发送其他数据包。

```
2605            else
2606            {
2607                    if(tcp_close_state(sk,1)==1)
2608                    {
2609                            tcp_send_fin(sk);
2610                    }
2611    }
```

否则根据当前套接字的状态决定是否需要发送一个 FIN 数据包给远端，具体情况参考 tcp_close_state 函数实现。

```
2612            release_sock(sk);
```

release_sock 函数处理该套接字接收的数据包，注意本地进行关闭操作后，实现上只是关闭了本地发送通道，但仍可以进行数据的接收。

```
2613    }//end of tcp_close
```

tcp_close 函数是系统调用 close 函数的传输层实现。该函数首先检查进行关闭操作的套接字是否为侦听套接字，如果是，则首先将其状态设置为 TCP_CLOSE,之后调用 tcp_close_pending 函数对侦听套接字 receive_queue 队列中连接请求数据包进行处理。注意侦听套接字并不进行数据交换，所以对于其进行关闭操作，主要是对其 receive_queue 队列中连接请求数据包进行处理即可，状态的转换较为简单，此处简单设置为 TCP_CLOSE 即可。对于非侦听套接字的处理如上文分析。

```
2614    /*
2615     *  This routine takes stuff off of the write queue,
2616     *  and puts it in the xmit queue. This happens as incoming acks
2617     *  open up the remote window for us.
2618     */

2619    static void tcp_write_xmit(struct sock *sk)
2620    {
2621            struct sk_buff *skb;
```

tcp_write_xmit 函数处理 write_queue 队列，将队列中缓存的之前由于应用层发送数据较快而越

出发送窗口的数据包发送出去。该函数被 tcp_ack 函数调用，当 tcp_ack 函数接收一个应答数据包后，其根据新的发送窗口处理 write_queue 队列，从而将其中满足发送要求的数据包及时发送出去。

```
2622        /*
2623         *  The bytes will have to remain here. In time closedown will
2624         *  empty the write queue and all will be happy
2625         */

2626        if(sk->zapped)
2627            return;
```

当接收到远端发送的 RST 复位信号后，本地 sock 结构 zapped 字段将被设置为 1，这表示通信通道已被关闭。在传送数据之前，需要重新建立连接方可。

```
2628        /*
2629         *  Anything on the transmit queue that fits the window can
2630         *  be added providing we are not
2631         *
2632         *  a) retransmitting (Nagle's rule)
2633         *  b) exceeding our congestion window.
2634         */

2635        while((skb = skb_peek(&sk->write_queue)) != NULL &&
2636            before(skb->h.seq, sk->window_seq + 1) &&
2637            (sk->retransmits == 0 ||
2638              sk->ip_xmit_timeout != TIME_WRITE ||
2639              before(skb->h.seq, sk->rcv_ack_seq + 1))
2640            && sk->packets_out < sk->cong_window)
2641        {
```

2635 行这个 while 语句即表示了 write_queue 中缓存的数据包得以发送需要满足的条件，共有四个，我们分别列出如下：

1> write_queue 队列中首先必须有需要发送的数据包，如果该队列为空，则无需进行任何操作。
2> 数据包中包含的所有数据必须在发送窗口之内。TCP 协议发送窗口的本质是远端接收缓冲区中可用空间大小。
3> 之前发送的数据包可以得到应答，即现在没有处于超时重发状态；另外当前从 write_queue 队列中取下的数据包必须是一个新的数据包，即数据包中数据序列号应该在对方已应答序列号之外。
4> 现在已发送出去而尚未得到应答的数据包的个数（sk->packets_out）必须小于拥塞窗口所允许的值（sk->cong_window）。拥塞窗口值表示当前网络的拥塞程度，是通过拥塞算法计算而得的一个估计值（计算主要在 tcp_ack 函数中进行）。

当满足以上四个条件时，就表示可以继续处理该数据包，将其发往下层进行处理。

```
2642                IS_SKB(skb);
2643                skb_unlink(skb);

2644                /*
2645                 *  See if we really need to send the packet.
2646                 */

2647                if (before(skb->h.seq, sk->rcv_ack_seq +1))
2648                {
2649                    /*
2650                     *  This is acked data. We can discard it. This
2651                     *  cannot currently occur.
2652                     */

2653                    sk->retransmits = 0;
2654                    kfree_skb(skb, FREE_WRITE);
2655                    if (!sk->dead)
2656                        sk->write_space(sk);
2657                }
```

2643 将数据包从 write_queue 队列中正式取下（注意 while 语句中使用了 skb_peek 函数）进行处理。2647 行是对数据包发送必要性的检查，虽然在 2635 行 while 语句中对此作过检查，但从 C 语言“||”操作符的执行方式而言，如果第一，二两个条件其中之一满足，则并不会执行到 before(skb->h.seq, sk->rcv_ack_seq +1)语句，所以 2647 行对此进行重新检查，如果数据包中数据的序列号在已应答序列号之中，则表示之前已经发送了相同的数据，那么该数据包就无发送的必要，作直接丢弃处理。2656 行是通过应用层又有可用的空闲写缓冲区（因为 2654 行刚刚释放了一个数据包占用的空间）。如果 2647 行 if 语句不满足，那么就表示可以将该数据包真正发往下层进行处理，并最终将其发送到网络介质上。这种情况对应如下的 else 语句块。

```
2658                else
2659                {
2660                    struct tcphdr *th;
2661                    struct iphdr *iph;
2662                    int size;
2663    /*
2664     * put in the ack seq and window at this point rather than earlier,
2665     * in order to keep them monotonic.    We really want to avoid taking
2666     * back window allocations.    That's legal, but RFC1122 says it's frowned on.
2667     * Ack and window will in general have changed since this packet was put
2668     * on the write queue.
2669     */
2670                    iph = (struct iphdr *)(skb->data +
2671                                           skb->dev->hard_header_len);
2672                    th = (struct tcphdr *)(((char *)iph) +(iph->ihl << 2));
2673                    size = skb->len - (((unsigned char *) th) - skb->data);
```

```
2674                    th->ack_seq = ntohl(sk->acked_seq);
2675                    th->window = ntohs(tcp_select_window(sk));

2676                    tcp_send_check(th, sk->saddr, sk->daddr, size, sk); //TCP 校验和计算

2677                    sk->sent_seq = skb->h.seq;

2678                    /*
2679                     *  IP manages our queue for some crazy reason
2680                     */

2681                    sk->prot->queue_xmit(sk, skb->dev, skb, skb->free);

2682                    /*
2683                     *  Again we slide the timer wrongly
2684                     */

2685                    reset_xmit_timer(sk, TIME_WRITE, sk->rto);
2686                }//end of else
2687            }//end of while
2688        }//end of fucntion
```

2658-2686 行代码初始化数据包中必要字段，并最终调用能够 ip_queue_xmit(2681 行)函数将数据包送往下层进行处理。2674 行对 TCP 首部中应答序列号（ack_seq）字段进行赋值，该字段表示本地从远端已接收到的最后数据字节的序列号加 1（加 1 并非特意而为之，只是序列号从 0 算起，所以当加上数据的长度后得到的序列号自然比最后一个字节的序列号多出 1 个单位）。2675 行初始化窗口值，这个窗口值作为远端的发送窗口，表示的是本地接收缓冲区空闲区域大小，用以 TCP 协议的流量控制功能实现。2677 行赋值很重要，每个连接都有通信双方各自的 sock 结构表示，而 sock 结构中 sent_seq 字段表示当前本端已发送数据的最后一个字节的序列号（加 1，因为序列号从 0 算起，以下类此，不再特别指出），这个字段在关闭连接时变得很重要，因为可以据此检查对方是否已接收到己方发送的所有数据，只要比较对方发送的 FIN 序列号和本地维护的 sent_seq 字段值即可。另外需要注意的是 sock 结构中还维护一个 write_seq 字段，这两个字段表示不同的含义：write_seq 表示当前应用层发给网络栈的所有数据的最后一个字节的序列号，而 sent_seq 表示网络栈已经发送出去（其含义表示至少数据已经交给硬件负责，已经脱离网络栈负责范围）的所有数据的最后一个字节的序列号。

下面要介绍的 tcp_ack 函数较长，在分析该函数之前，我们先从理论上分析该函数应该完成的任务，然后再看具体代码，可以帮助我们理解 tcp_ack 函数的具体实现。

首先 tcp_ack 函数用于处理本地接收到的 ACK 数据包，在本书其它地方已经提及，使用 TCP 协议的套接字，在连接建立完成后，此后发送的每个数据包中 ACK 标志位都被设置为 1，所以一个 ACK 数据包本身也将包含数据，不过数据的处理专门有其他函数（tcp_data, tcp_urg）负责，tcp_ack 函数将只对 ACK 标志位及其相关联字段进行处理，这一点需要注意。

1. 首先既然是一个 ACK 数据包，则表示本地发送的数据已经成功被远端接收，此时可以对重发队列中已得到 ACK 的数据包进行释放。
2. 只要是从远端接收的数据包（包括 ACK 数据包），该数据包中包含远端的当前窗口大小，本地将对此窗口进行检查，从而决定是否将写队列中的相关数据包发送出去（对应窗口增加）。或者是将重发队列中部分数据包重新缓存到写队列中（对应窗口减小的情况）。
3. 如果数据包的交换着重于状态的更新（如连接建立，连接关闭），则根据套接字的当前状态进行相应的更新（如将状态从 TCP_SYN_RECV 更新为 TCP_ESTABLISHED）。
4. 如果该 ACK 数据包对本地当前连接而言是一个非法 ACK 数据包，则也需要进行相关的处理。

以上涉及到发送过程中的两个队列：

1〉 写队列，对应 sock 结构中 write_queue 字段指向的队列，这是一个双向队列。该队列接收应用层发送的数据包（传输层将数据封装为数据包，将其挂接到 write_queue 队列中）。该队列中数据包尚未发送出去。

2〉 重发队列，对应 sock 结构中 send_head, send_tail 字段指向的队列，这是一个单向队列，send_head 指向队列头部，send_tail 指向队列尾部。传输层（实际上是网络层）将数据包发送出去以后，将数据包缓存到该队列中，以防止发送的数据包可能丢失后的重发工作。

下面我们进入 tcp_ack 函数实现代码的具体分析，在代码相关部分我们会对应到上面提到的需要实现功能的 4 个方面。

```
2689    /*
2690     *  This routine deals with incoming acks, but not outgoing ones.
2691     */

2692    extern __inline__ int tcp_ack(struct sock *sk, struct tcphdr *th, unsigned long saddr, int len)
2693    {
2694        unsigned long ack;
2695        int flag = 0;

2696        /*
2697         * 1 - there was data in packet as well as ack or new data is sent or
2698         *      in shutdown state
2699         * 2 - data from retransmit queue was acked and removed
2700         * 4 - window shrunk or data from retransmit queue was acked and removed
2701         */

2702        if(sk->zapped)
2703            return(1); /* Dead, cant ack any more so why bother */
```

sock 结构 zapped 字段设置为 1，表示该套接字之前接收到远端发送的一个 RST 数据包，所以任何从远端接收到的数据包都简单丢弃，不用处理，ACK 数据包也不例外。

```
2704        /*
2705         *  Have we discovered a larger window
```

```
2706            */

2707            ack = ntohl(th->ack_seq);
```

将 ack 字段设置为远端期望从本地接收的下一个字节的序列号。此处对 ack_seq 的含义需要进行说明。ack_seq 一方面表示应答序列号，另一方面也表示请求序列号。如果本地发送一个数据包，被远端接收后，从远端的角度看，该数据包 TCP 首部中 ack_seq 字段的含义为本地期望从远端接收的下一个字节的序列号，换句话说，在 ack_seq 字段值表示的序列号之前的数据都已经成功被本地接收（注意不包含 ack_seq 本身所代表的序列号）。

```
2708            if (ntohs(th->window) > sk->max_window)
2709            {
2710                    sk->max_window = ntohs(th->window);
2711#ifdef CONFIG_INET_PCTCP
2712                    /* Hack because we don't send partial packets to non SWS
2713                       handling hosts */
2714                    sk->mss = min(sk->max_window>>1, sk->mtu);
2715    #else
2716                    sk->mss = min(sk->max_window, sk->mtu);
2717    #endif
2718            }
```

以上这段代码对该数据包中所声称的窗口进行处理，并相应的更新 MSS 值。

```
2719            /*
2720             *  We have dropped back to keepalive timeouts. Thus we have
2721             *  no retransmits pending.
2722             */

2723            if (sk->retransmits && sk->ip_xmit_timeout == TIME_KEEPOPEN)
2724                    sk->retransmits = 0;
```

保活（Keepalive）定时器用于在双方长时间内暂无数据交换时，进行连接保持，以防止一方崩溃后，另一方始终占用资源的情况发生。如果对应 sock 结构定时器当前被设置为保活定时器，则表示当前应无重发情况，如果 sock 结构 retransmits 字段非 0，则对其进行清零操作。注意对该字段的清零操作较为重要，该字段被作为其他相关处理的判断条件，在介绍到相关内容时会做出说明。

```
2725            /*
2726             *  If the ack is newer than sent or older than previous acks
2727             *  then we can probably ignore it.
2728             */

2729            if (after(ack, sk->sent_seq) || before(ack, sk->rcv_ack_seq))
2730            {
2731                    if(sk->debug)
2732                            printk("Ack ignored %lu %lu\n",ack,sk->sent_seq);
```

```
2733            /*
2734             *  Keepalive processing.
2735             */

2736            if (after(ack, sk->sent_seq))
2737            {
2738                return(0);
2739            }

2740            /*
2741             *  Restart the keepalive timer.
2742             */

2743            if (sk->keepopen)
2744            {
2745                if(sk->ip_xmit_timeout==TIME_KEEPOPEN)
2746                    reset_xmit_timer(sk, TIME_KEEPOPEN, TCP_TIMEOUT_LEN);
2747            }
2748            return(1);
2749        }
```

这段代码对应答序列号进行检查，ack 变量在前文中被初始化为所接收 ACK 数据包中 TCP 首部中 ack_seq 字段。sock 结构中 sent_seq 字段表示本地已发送的最后一个字节的序列号，注意此处已发送并非是指这些数据已得到对方应答，而只是本地通过调用网卡驱动发送到网络介质上，这些数据包可能仍然缓存于本地重发队列中（send_head, send_tail 指向的队列），尚未得到应答。rcv_ack_seq 字段表示本地当前为止从远端接收到的最后一个 ACK 数据包中所包含的应答序列号。如果当前接收的 ACK 数据包中应答序列号在 sent_seq 之后，则表示这是一个无效的应答序列号，处理上可以直接丢弃该应答数据包。事实上，上面这段代码也是如此处理的，但是返回值设置为 0，对于 tcp_ack 函数而言，返回 0 表示出现错误，从这一点来看，实际实现上，对于应答序列号超前的情况是作为错误处理的。如果当前接收的应答序列号在 rcv_ack_seq 之前，表示这是一个过期的应答序列号，对于过期的应答可能是该数据包在网络某个节点（路由器）上被延迟的原因，所以对于这种情况，作简单丢弃处理，返回值设置为 1，表示一切正常，但无需进行下面代码的进一步执行。注意对于以上这两种情况，均表示远端与本地的通信通道正常（当然双方主机也正常），如果 TCP 当前处于保活阶段，则可以将保活定时器重置。

代码往下执行，表示应答序列号正常，此时需要进行进一步的处理。

```
2750        /*
2751         *  If there is data set flag 1
2752         */

2753        if (len != th->doff*4)
2754            flag |= 1;
```

len 参数表示 IP 数据负载的长度：包括 TCP 首部和 TCP 数据负载。如果这个长度大于 TCP 首部长度，则表示该 ACK 数据包同时包含有正常数据。此时将 flag 标志位进行相应的设置（最低有效位设置为 1）。

```
2755            /*
2756             *    See if our window has been shrunk.
2757             */

2758            if (after(sk->window_seq, ack+ntohs(th->window)))
2759            {
2760                /*
2761                 * We may need to move packets from the send queue
2762                 * to the write queue, if the window has been shrunk on us.
2763                 * The RFC says you are not allowed to shrink your window
2764                 * like this, but if the other end does, you must be able
2765                 * to deal with it.
2766                 */
2767                struct sk_buff *skb;
2768                struct sk_buff *skb2;
2769                struct sk_buff *wskb = NULL;

2770                skb2 = sk->send_head;
2771                sk->send_head = NULL;
2772                sk->send_tail = NULL;

2773                /*
2774                 *    This is an artifact of a flawed concept. We want one
2775                 *    queue and a smarter send routine when we send all.
2776                 */
//窗口大小发生变化（被缩减），设置相应的标志位。
2777                flag |= 4;  /* Window changed */

//注意本地 sock 结构 window_seq 字段的赋值方式，它的值设置为应答序列号加上 TCP 首部
//中的窗口大小，所以 window_seq 字段表示的是绝对序列号数值，而非仅仅是一个大小。
//这一点值得注意。从此处赋值语句的方式，对于前面的 if 条件判断语句应不难理解：判断
//本地窗口序列号是否大于这个 ACK 数据包所声明的窗口序列号，如果大于，则表示窗口
//大小被缩减了，此时需要对重发队列中缓存的部分数据包进行回送（到写队列中）处理。
2778                sk->window_seq = ack + ntohs(th->window);
2779                cli();
2780                while (skb2 != NULL)
2781                {
2782                    skb = skb2;
2783                    skb2 = skb->link3;
2784                    skb->link3 = NULL;
2785                    if (after(skb->h.seq, sk->window_seq))
```

```
2786                    {
2787                        if (sk->packets_out > 0)
2788                            sk->packets_out--;
2789                        /* We may need to remove this from the dev send list. */
2790                        if (skb->next != NULL)
2791                        {
2792                            skb_unlink(skb);
2793                        }
2794                        /* Now add it to the write_queue. */
2795                        if (wskb == NULL)
2796                            skb_queue_head(&sk->write_queue,skb);
2797                        else
2798                            skb_append(wskb,skb);
2799                        wskb = skb;
2800                    }
2801                    else
2802                    {
2803                        if (sk->send_head == NULL)
2804                        {
2805                            sk->send_head = skb;
2806                            sk->send_tail = skb;
2807                        }
2808                        else
2809                        {
2810                            sk->send_tail->link3 = skb;
2811                        sk->send_tail = skb;
2812                        }
2813                        skb->link3 = NULL;
2814                    }
2815                }
2816                sti();
2817            }
```

TCP 协议极不推荐窗口大小缩减的行为，但要求 TCP 协议实现必须能够处理这种缩减行为。如上这段代码即是处理窗口大小缩减的情况。本地 sock 结构中 window_seq 字段表示的远端窗口大小，这是一个以远端发送的应答的序列号（这个也随时在改变）为基点的绝对数值（虽然被称为窗口大小）。从上文中对该字段的赋值方式可以看出这点（参见上文中相关注释）。本质上，远端窗口大小表示的是当前远端接收缓冲区中可用空间大小，表示了本地当前最多可以发送的数据量。TCP 首部中 window 字段表示的即是实际的窗口值，但本地在表示上加上了 TCP 首部中的应答序列号字段值。这样表成为了一个绝对数值，这种表示方式方便本地在发送数据包时进行序列号检查。无论何种表示方式本地上都是一致的。如果窗口大小被缩减了，则在处理上必须将重发队列中序列号超出窗口之外的数据包回送到写队列中。这段代码完成的工作即是如此。其中重发队列由 send_head,send_tail 所指向的单向队列表示，写队列由 write_queue 指向的双向队列表示。这段代码遍历重发队列，对队列中每个数据包进行序列号检查，如果该数据包

中数据序列号超出当前窗口之外，则将该数据包从重发队列中删除，插入到写队列中。处理完后，本地 sock 结构中 sent_seq 字段将处于不一致状态。不过这种不一致状态不会对网络栈造成实际工作上的影响。因为该字段将很快得到更新。

```
2818            /*
2819             *    Pipe has emptied
2820             */

2821            if (sk->send_tail == NULL || sk->send_head == NULL)
2822            {
2823                sk->send_head = NULL;
2824                sk->send_tail = NULL;
2825                sk->packets_out= 0;
2826            }

2827            /*
2828             *    Update the right hand window edge of the host
2829             */

2830            sk->window_seq = ack + ntohs(th->window);
```

在处理完可能的窗口大小被缩减的情况后，重新检查重发队列是否为空，如为空，则将 sock 结构中 send_head, send_tail, packets_out 字段设置为正确值。另外更新 window_seq 字段为远端当前声明的窗口值（前面只是用于 if 条件判断，并非进行更新，所以此处进行窗口的正式更新）。

```
2831            /*
2832             *    We don't want too many packets out there.
2833             */

2834            if (sk->ip_xmit_timeout == TIME_WRITE &&
2835                sk->cong_window < 2048 && after(ack, sk->rcv_ack_seq))
2836            {
2837                /*
2838                 * This is Jacobson's slow start and congestion avoidance.
2839                 * SIGCOMM '88, p. 328.    Because we keep cong_window in integral
2840                 * mss's, we can't do cwnd += 1 / cwnd.    Instead, maintain a
2841                 * counter and increment it once every cwnd times.    It's possible
2842                 * that this should be done only if sk->retransmits == 0.    I'm
2843                 * interpreting "new data is acked" as including data that has
2844                 * been retransmitted but is just now being acked.
2845                 */
2846                if (sk->cong_window < sk->ssthresh)
2847                    /*
2848                     *    In "safe" area, increase
2849                     */
```

```
2850                    sk->cong_window++;
2851                else
2852                {
2853                    /*
2854                     *  In dangerous area, increase slowly.  In theory this is
2855                     *      sk->cong_window += 1 / sk->cong_window
2856                     */
2857                    if (sk->cong_count >= sk->cong_window)
2858                    {
2859                        sk->cong_window++;
2860                        sk->cong_count = 0;
2861                    }
2862                    else
2863                        sk->cong_count++;
2864                }
2865            }
```

网络栈代码每当新发送一个数据包后，就将 sock 结构 ip_xmit_timeout 字段设置为 TCP_WRITE，并同时启动超时重传定时器，此时接收到的这个 ACK 数据包，其无论是对之前发送的数据包的应答，还是刚刚发送的这个数据包的应答，都表示通信通道正常。cong_window 字段含义为拥塞窗口大小，表示的是本地最大可同时发送但未得到应答的数据包个数，注意这时本地加入的限制，远端通过 TCP 首部中窗口大小对本地数据包发送加以限制。cong_window 字段主要用于处理拥塞（与慢启动配合使用），如果发送的数据包得到应答，则相应的增加该窗口大小，直到达到某个最大值（sock 结构中 ssthresh 字段表示这个最大值）。cong_count 字段在此版本网络代码中用途不明，此处对 cong_count 进行了赋值，但并非在其他地方进行使用（只在 tcp_retransmit 函数中进行满启动过程时将该字段重新设置为 0），在发送数据包时，使用的是 cong_window 字段和远端窗口大小进行检查。

上面的 if 条件语句判断接收到的应答序列号是否是一个好的序列号，此处硬编码的 2048 常数有些诡异，正确的比较应该是与 sock 结构中 ssthresh 字段进行。在判断出一个好的序列号后，对 cong_window 字段进行更新（在达到最大值之前进行加 1 操作）。

```
2866            /*
2867             *  Remember the highest ack received.
2868             */

2869            sk->rcv_ack_seq = ack;
```

更新 sock 结构中 rcv_ack_seq 字段值，这个字段的含义从此处赋值语句可以看出：表示最近一次接收的应答序列号。（大部分时候，我们通过对一个变量的赋值方式来理解该变量的意义，这一点在分析内核代码时是一个重要和有效的方法。）

```
2870            /*
2871             *  If this ack opens up a zero window, clear backoff.  It was
2872             *  being used to time the probes, and is probably far higher than
```

```
2873            *    it needs to be for normal retransmission.
2874            */

2875            if (sk->ip_xmit_timeout == TIME_PROBE0)
2876            {
2877                sk->retransmits = 0;        /* Our probe was answered */

2878                /*
2879                 *    Was it a usable window open ?
2880                 */

2881                if (skb_peek(&sk->write_queue) != NULL &&      /* should always be non-null */
2882                    ! before (sk->window_seq, sk->write_queue.next->h.seq))
2883                {
2884                    sk->backoff = 0;

2885                    /*
2886                     *    Recompute rto from rtt.    this eliminates any backoff.
2887                     */

2888                    sk->rto = ((sk->rtt >> 2) + sk->mdev) >> 1;
2889                    if (sk->rto > 120*HZ)
2890                        sk->rto = 120*HZ;
2891                    if (sk->rto < 20)     /* Was 1*HZ, then 1 - turns out we must allow about
2892                                             .2 of a second because of BSD delayed acks - on a
100Mb/sec link
2893                                             .2 of a second is going to need huge windows (SIGH) */
2894                    sk->rto = 20;
2895                }
2896            }
```

这段代码判断该 ACK 数据包是否是一个窗口通报数据包，如果发送端发送数据包的速度大于接收端处理的速度，则一段时间后，接收端接收缓冲区将耗尽，此时在接收端回复的 ACK 数据包中窗口大小将被设置为 0（注意窗口大小本质上是接收缓冲区可用空间大小），此时发送端将停止发送数据包，直到接收端通报一个非 0 窗口大小。接收端在经过一段时间的处理后，闲置出部分接收缓冲区，其容量可以容纳一个数据包长度后（该长度一般为一个最大报文长度），就发送一个窗口通报数据包给发送端，从而解除发送端的发送禁令。但这种方式存在的一个可能局面是双方限于死锁状态：接收端发送的非 0 窗口大小通报数据包不幸丢失，此时发送端“傻乎乎”的等待这个非 0 窗口大小通报数据包，在没有等来该数据包之前，就一直禁止发送数据包；而接收端认为其已经发送了非 0 窗口大小通报数据包，其“乐乎乎”的等待发送端发送其他数据包，这样双方都在盲目的等待，“直到永远”。而造成这种局面的原因仅仅是丢失了该非 0 窗口大小通报数据包。为了解决这个问题，TCP 协议规范定义了一个窗口探测定时器，发送端一旦接收到一个 0 窗口大小的数据包，就启动该窗口探测定时器，定时间隔也采用指数退避算法，每隔一个间隔就发送这样一个窗口探测数据包（注意虽然接收端已经声明了 0 窗口，但其必须

对窗口探测数据包进行接收和处理以及回复)，这样即便接收端主动发送非 0 窗口大小数据包发生丢失的情况，在接收端窗口探测数据包后，其可以再回复一个非 0 窗口数据包，可以解决之前的死锁问题。

以上这段代码就是判断这个数据包是否是一个非 0 窗口通报数据包，如果当前定时器设置为窗口探测定时器，则接收到的这个 ACK 数据包就是一个非 0 窗口通报数据包（无论是远端主动发送的，还是响应本地发送的窗口探测数据包而发送的，都不影响问题本质)，此时一方面更新窗口大小值(这在前面对窗口大小进行处理时已经得到更新)，另一方面对相关其他变量进行更新，这包括清除重发标志（将 sock 结构中 retransmits 字段设置为 0)，清除指数退避算法（将 sock 结构中 backoff 字段设置为 0，并重新计算 RTO 值，注意重新计算后自动清除指数退避算法引起的加倍操作)，并对计算后结果进行调整（RTO 值不可太大，也不可太小，这个大小的掌握由具体实现决定，此处最大不可大于 2 分钟，最小不可小于 20 个系统 tick 数，一个 tick 表示一次系统时钟中断)。

注意此处处理完窗口探测情况后并不是直接退出该函数的执行，而是进行执行下面的代码，因为窗口探测发生于一个 0 窗口通告，而这时发生在数据正在传输过程中，而非 0 窗口通告数据包其本身并非具有特别的格式，或者说非 0 窗口通告只是作为 TCP 首部中的一个窗口字段而存在，这个数据包本身还可能包含有发送到另一端的数据，并且同时也是一个 ACK 数据包，所以从无所谓的角度而言，对于窗口通告数据包的处理只是涉及到其关联的几个字段而已，而且也不是关键字段，所以处理完这些字段后，还需要对其他重要方面进行处理，这些重要方面中其中之一就是对 ACK 标志位进行处理，这正是下文代码所处理的任务。
对 ACK 标志位的处理体现在对重发队列中数据包的处理,所以以下这段代码主要是遍历重发队列，对每个数据包进行序列号检查，查看接收的应答序列号是否对该数据包中数据进行了应答，由于重发队列中数据包是按序列号排序的，所以一旦碰到一个数据序列号在应答序列号之外的数据包即可停止遍历过程。此处一个隐含的思想是一个应答数据包可以对多个数据包进行应答，所以发送的数据包和接收到的应答数据包不是一一对应的，应答数据包个数通常小于发送的数据包（如果使用了 Nagle 算法，则这种关系将是一一对应的，因为 Nagle 算法的要求即是在为接收到上一个发送的数据包应答之前，不可继续发送下一个数据包，不过 Nagle 算法主要用于处理小量数据传输的情况，用于节约网络上数据包的个数和增加网络使用效率，绝大多数应用包括有些小数据量传输的应用（为了保证良好的交互性）都会禁用 Nagle 算法)。

```
2897        /*
2898         *  See if we can take anything off of the retransmit queue.
2899         */

2900        while(sk->send_head != NULL)
2901        {
2902            /* Check for a bug. */
2903            if (sk->send_head->link3 &&
2904                after(sk->send_head->h.seq, sk->send_head->link3->h.seq))
2905                printk("INET: tcp.c: *** bug send_list out of order.\n");
```

重发队列中数据包是按序列号进行排序的，如果发现乱序的情况，打印错误信息。注意此处没有对乱序进行处理，实际上是期望通过本 ACK 序列号和将来的 ACK 序列号进行自动修复，修复的原理读者可自行仔细体会。

```
2906            /*
2907             *  If our packet is before the ack sequence we can
2908             *  discard it as it's confirmed to have arrived the other end.
2909             */
```

对于发送的数据包 sk_buff 结构中 h 字段（union 类型）中 seq 子字段，其表示的含义该数据包中最后一个字节的序列号（读者可参看前文中 tcp_send_skb 函数的相关代码）。在比较一个 ACK 数据包是否是对一个本地发送的数据包的应答时，当然应该使用本地发送数据包中最后一个数据字节的序列号是否在应答序列号包含范围之内。下面的这个 if 语句即作此判断。如果在应答范围之内，则可以对该数据包进行释放，以释放本地发送缓冲区空间。

```
2910            if (before(sk->send_head->h.seq, ack+1))
2911            {
2912                    struct sk_buff *oskb;
2913                    if (sk->retransmits)
2914                    {
2915                            /*
2916                             *  We were retransmitting.  don't count this in RTT est
2917                             */
2918                            flag |= 2;
```

如果 sock 结构 retransmits 字段非 0，则表示已经进行过数据包重新发送操作，此时将相应设置 flag 标志位。这个标志位将决定下面是否使用该 ACK 数据包进行 RTO 计算。如果发生重发情况，则应为无法知道该 ACK 数据包是对哪一次重发进行的应答，根据 Karn 算法，将不使用该 ACK 数据包对 RTO 值进行重新计算。

```
2919                            /*
2920                             * even though we've gotten an ack, we're still
2921                             * retransmitting as long as we're sending from
2922                             * the retransmit queue.  Keeping retransmits non-zero
2923                             * prevents us from getting new data interspersed with
2924                             * retransmissions.
2925                             */

2926                            if (sk->send_head->link3)     /* Any more queued retransmits? */
2927                                    sk->retransmits = 1;
2928                            else
2929                                    sk->retransmits = 0;
```

如果重发队列中只有一个数据包，则由于收到一个 ACK 数据包，此时即结束了重发过程，则相应的将 retransmits 字段设置为 0；否则由于暂且无法知道这个 ACK 数据包是否对重发队列中所有数据包进行了应答，则不能将 retransmits 字段设置为 0，此处的处理是将其设置为 1（取 1 的原因有些随意，并非非要设置为 1）。

```
2930                    }
2931                    /*
2932                     * Note that we only reset backoff and rto in the
```

```
2933                     * rtt recomputation code.    And that doesn't happen
2934                     * if there were retransmissions in effect.    So the
2935                     * first new packet after the retransmissions is
2936                     * sent with the backoff still in effect.    Not until
2937                     * we get an ack from a non-retransmitted packet do
2938                     * we reset the backoff and rto.    This allows us to deal
2939                     * with a situation where the network delay has increased
2940                     * suddenly.    I.e. Karn's algorithm. (SIGCOMM '87, p5.)
2941                     */

2942                    /*
2943                     *    We have one less packet out there.
2944                     */

2945                    if (sk->packets_out > 0)
2946                         sk->packets_out --;
```

sock 结构 packets_out 字段表示当前已被发送出去但尚未接收到应答的数据包个数。由于发现一个被应答的数据包，所以此处对应的将此字段减 1。注意相关这些变量的更新是必要的，因为在网络代码的其他地方将根据这些变量的数值决定行为。

```
2947                    /*
2948                     *    Wake up the process, it can probably write more.
2949                     */
2950                    if (!sk->dead)
2951                         sk->write_space(sk);
```

更新 packets_out 字段后，就可以对等待发送数据包的进程进行唤醒操作。注意在数据包发送函数中 packets_out 字段将作为判断条件之一来决定是否将数据包发送出去，即系统对发送出去但未得到应答的数据包个数有限制。

```
2952                    oskb = sk->send_head;
```

oskb 指向一个待释放的数据包。

```
2953                    if (!(flag&2))  /* Not retransmitting */
2954                    {
```

如果 flag 变量对应标志位没有被设置，则表示可以使用该 ACK 数据包进行 RTO 的更新。由于 flag 变量对应标志位的设置参见上文分析。另外如下代码对 RTO 的更新算法请参考 TCP 协议规范，此处也不做解释。

```
2955                         long m;

2956                         /*
2957                          *    The following amusing code comes from Jacobson's
```

```
2958                    *    article in SIGCOMM '88.    Note that rtt and mdev
2959                    *    are scaled versions of rtt and mean deviation.
2960                    *    This is designed to be as fast as possible
2961                    *    m stands for "measurement".
2962                    */

2963                   m = jiffies - oskb->when;    /* RTT */
2964                   if(m<=0)
2965                         m=1;              /* IS THIS RIGHT FOR <0 ??? */
2966                   m -= (sk->rtt >> 3);       /* m is now error in rtt est */
2967                   sk->rtt += m;                 /* rtt = 7/8 rtt + 1/8 new */
2968                   if (m < 0)
2969                         m = -m;          /* m is now abs(error) */
2970                   m -= (sk->mdev >> 2);      /* similar update on mdev */
2971                   sk->mdev += m;                /* mdev = 3/4 mdev + 1/4 new */

2972                   /*
2973                    *    Now update timeout.    Note that this removes any backoff.
2974                    */

2975                   sk->rto = ((sk->rtt >> 2) + sk->mdev) >> 1;
2976                   if (sk->rto > 120*HZ)
2977                         sk->rto = 120*HZ;
2978                   if (sk->rto < 20)     /* Was 1*HZ - keep .2 as minimum cos of the BSD
delayed acks */
2979                         sk->rto = 20;
```

注意在重新计算 RTO 值后，就需要清除指数退避算法，如下将 backoff 字段设置为 0 即是此目的。

```
2980                   sk->backoff = 0;
2981             }
```

此处重新更新 flag 变量值，对于 flag|=2 的情况可以理解，因为无论是不允许重新计算 RTO 值还是允许计算 RTO 值，代码执行到此处，可以等同于一个结论：不允许计算 RTO 值（对于允许的情况，刚才已经计算过了，所以此处设置该标志位，已经无关紧要）。对于 flag|=4 的含义需要在下文代码使用到该标志位的地方才可推测出该标志位的意义，此处暂且不论。

```
2982             flag |= (2|4);    /* 2 is really more like 'don't adjust the rtt
2983                                    In this case as we just set it up */
2984             cli();
2985             oskb = sk->send_head;
2986             IS_SKB(oskb);
2987             sk->send_head = oskb->link3;
```

```
2988                    if (sk->send_head == NULL)
2989                    {
2990                            sk->send_tail = NULL;
2991                    }
```

这段代码对数据包进行释放，并更新相关字段。

```
2992            /*
2993             *      We may need to remove this from the dev send list.
2994             */

2995                    if (oskb->next)
2996                            skb_unlink(oskb);
```

sk_buff 结构 next 指针用于硬件队列中，既然该数据包已经得到应答，如果该数据包仍然缓存于硬件队列中，则此时可以将其从中删除了。注意重发队列中数据包每重新发送一次，都要首先缓存到硬件队列中，如果在这个 ACK 数据包到达之前，刚刚又进行了重发操作，则有可能这个数据包还被挂接在硬件队列中。

```
2997                    sti();
2998                    kfree_skb(oskb, FREE_WRITE); /* write. */
```

解除了该数据包与其它部分的关联后，现在可以安全的调用 kfree_skb 进行释放了。

```
2999                    if (!sk->dead)
3000                            sk->write_space(sk);
```

每释放一个数据包，就表示发送缓冲区又有可用空间了，如果有进程等待于发送缓冲区，则需要对这些进程进行唤醒，如上代码的目的即是如此。

```
3001            }
3002            else
3003            {
```

这个 else 语句对应重发队列中数据包最后一个字节的序列号不在应答序列号范围之内，此时可以跳出 while 循环了，因为应答序列号所应答的数据包已经处理完毕（注意基于的前提是重发队列中数据包是按序列号排序的，实际上也是这样操作的）。对于出现乱序的问题，只不过一个应该被释放的数据包仍然挂接在重发队列中，后续的 ACK 数据包会对此进行处理的。所以这种乱序不会造成不能工作的结局，而且乱序的情况也不会轻易出现。

```
3004                    break;
3005            }
3006    }
```

对于 ACK 标志位的处理，诚如上文所述，即遍历重发队列，对每个数据包进行检查，具体分析请参考代码中注释。一旦序列号超出应答序列号之外，则进入最后的 else 语句，通过 break 跳出 while 循环，结束对 ACK 标志位的处理。

tcp_ack 函数执行到此处，通过以上对窗口的处理，以及对重发队列中有关数据包的释放，主要有如下变量得到更新：

1〉 远端窗口大小（远端接收缓冲区可用空间大小）

2〉 本地发送缓冲区可用空间大小

3〉 已发送出去但尚未得到应答的数据包个数可能减小

条件 1〉，3〉 允许我们调整 write_queue 写队列中的数据包，将其尽可能发送出去（转移到 send_head 重发队列中）；条件 2〉 允许上层应用程序向 write_queue 写队列中写入更多数据包。对于向 write_queue 的写入是通过唤醒上层应用进程完成的，这个写入内核网络栈无法自行解决，但对于 write_queue 中已有的数据包，在条件允许时，可以将其发送出去，下面的一段代码完成的工作即是如此。

```
3007            /*
3008             * XXX someone ought to look at this too.. at the moment, if skb_peek()
3009             * returns non-NULL, we complete ignore the timer stuff in the else
3010             * clause.    We ought to organize the code so that else clause can
3011          * (should) be executed regardless, possibly moving the PROBE timer
3012             * reset over.    The skb_peek() thing should only move stuff to the
3013             * write queue, NOT also manage the timer functions.
3014             */

3015            /*
3016             * Maybe we can take some stuff off of the write queue,
3017             * and put it onto the xmit queue.
3018             */
3019            if (skb_peek(&sk->write_queue) != NULL)
3020            {
3021                if (after (sk->window_seq+1, sk->write_queue.next->h.seq) &&
3022                        (sk->retransmits == 0 ||
3023                     sk->ip_xmit_timeout != TIME_WRITE ||
3024                     before(sk->write_queue.next->h.seq, sk->rcv_ack_seq + 1))
3025                    && sk->packets_out < sk->cong_window)
3026                {
3027                    /*
3028                     *    Add more data to the send queue.
3029                     */
3030                    flag |= 1;
3031                    tcp_write_xmit(sk);
3032                }
3033                else if (before(sk->window_seq, sk->write_queue.next->h.seq) &&
3034                    sk->send_head == NULL &&
```

```
3035                    sk->ack_backlog == 0 &&
3036                    sk->state != TCP_TIME_WAIT)
3037            {
3038                    /*
3039                     *  Data to queue but no room.
3040                     */
3041                        reset_xmit_timer(sk, TIME_PROBE0, sk->rto);
3042            }
3043    }
```

注意首先使用的是 skb_peek 函数查看 write_queue 中是否缓存有数据包，对于没有缓存的情况，我们下文分析，如果有数据包缓存，那么，现在可以对其进行处理了。当然首先要检查数据包序列号是否在远端窗口序列号之内，并且检查这些数据包是否是需要发送的新的数据，以及发送出去但尚未得到应答的数据包个数是否在系统允许范围之内，另外判断条件中加入的一项是检查本地是否已经处于超时重发过程，如果还是处于超时重发，则不会处理 write_queue 中数据包，原因很简单，对于超时的情况，有可能网络当前很忙，或者传输线路暂时出现某种问题（或中间某个路由器挂了，或远端主机死了等等），此时不可继续进行发送从而使得问题更加严重。对于以上检查数据包中数据是否有必要发送这个条件需要说明一下，在前面代码分析中，我们刚刚分析有远端窗口大小缩减时，重发队列中数据包会被重新送回到写队列中，由于重发队列中数据包已经发送出去，只是等待应答，如果对这些数据包的应答在传输中被延迟，而远端后发送的窗口大小缩减数据包由于选择了另一条线路先行到达，则此时有可能这些被回送到写队列中的数据已经得到应答（即远端已经成功接收这些数据），通过对 sock 结构 rcv_ack_seq 字段检查，我们可以决定是否有必要发送这些数据（注意 rcv_ack_seq 字段的含义，参见前文对其的赋值方式和相关分析）。

如果以上条件均满足发送，则调用 tcp_write_xmit 函数处理写队列，此函数在前文中已有分析，此处不再论述。

如果条件不满足发送，则检查是否需要启动窗口探测定时器，注意启动的条件，重发队列为空，本地也无需要发送的 ACK 数据包，通信通道还没有被关闭，但窗口大小不能允许本地发送数据包（注意本地维护的窗口大小是一个绝对值，所以不可直接与 0 进行比较，而是应该与需要发送的第一个数据包的序列号进行比较，加上其它辅助条件检查，可以推断出窗口大小是否为 0），如果判断出窗口大小为 0，则启动窗口探测定时器。

以上 tcp_ack 函数的所有代码都没有涉及到状态转换的处理，对于三路握手连接建立过程和连接关闭过程（此时需要 4 个数据包），都需要根据 ACK 数据包进行状态的更新。函数中对状态的更新放在函数结尾处，此时将对状态的处理分成两个模块：

1〉 在对其他方面（主要是数据包的发送）进行了处理后进行。该模块是在一个 else 语句中，对应的 if 语句用于判断 write_queue 中是否有待发送的数据包，如果有，则直接进入 if 语句块，不会进行这些状态的更新，所表达的意思是如果双方都有数据包等待发送，则表示双方正在进行数据交互，对于这种情况，对应的某些状态无需进行更新或者处理。这些状态有：TCP_TIME_WAIT，TCP_CLOSE。

2〉 无论是否有其他数据包需要处理，对于每个接收到的 ACK 数据包都必须进行处理的状态，这些状态的检查不与任何其他条件形成互斥，即只有代码可以执行到此处，则一定对这些

状态进行更新。这些状态有：TCP_LAST_ACK，TCP_FIN_WAIT1，TCP_CLOSING，TCP_SYN_RECV。

先看第一个模块中状态的处理，诚如刚才的分析，该模块放在一个 else 语句块中。

```
3044            else
3045            {
3046                /*
3047                 * from TIME_WAIT we stay in TIME_WAIT as long as we rx packets
3048                 * from TCP_CLOSE we don't do anything
3049                 *
3050                 * from anything else, if there is write data (or fin) pending,
3051                 * we use a TIME_WRITE timeout, else if keepalive we reset to
3052                 * a KEEPALIVE timeout, else we delete the timer.
3053                 *
3054                 * We do not set flag for nominal write data, otherwise we may
3055                 * force a state where we start to write itsy bitsy tidbits
3056                 * of data.
3057                 */

3058                switch(sk->state) {
3059                case TCP_TIME_WAIT:
3060                    /*
3061                     * keep us in TIME_WAIT until we stop getting packets,
3062                     * reset the timeout.
3063                     */
3064                    reset_msl_timer(sk, TIME_CLOSE, TCP_TIMEWAIT_LEN);
3065                    break;
3066                case TCP_CLOSE:
3067                    /*
3068                     * don't touch the timer.
3069                     */
3070                    break;
3071                default:
3072                    /*
3073                     *    Must check send_head, write_queue, and ack_backlog
3074                     *    to determine which timeout to use.
3075                     */
3076                    if    (sk->send_head    ||    skb_peek(&sk->write_queue)    !=    NULL    ||
sk->ack_backlog) {
3077                        reset_xmit_timer(sk, TIME_WRITE, sk->rto);
3078                    } else if (sk->keepopen) {
3079                        reset_xmit_timer(sk, TIME_KEEPOPEN, TCP_TIMEOUT_LEN);
3080                    } else {
```

```
3081                        del_timer(&sk->retransmit_timer);
3082                        sk->ip_xmit_timeout = 0;
3083                    }
3084                    break;
3085                }
3086            }
```

对于 TCP_TIME_WAIT 状态的处理很简单，只要从远端接收到（FIN）数据包，则重置 2MSL 定时器。TCP_TIME_WAIT 状态表示本地已经接收到远端发送的 FIN 数据包，并且已经发送了 ACK 数据包，如果此时还继续从远端能够接收到数据包，则很有可能是本地发送的 ACK 数据包丢失，造成远端正在重发 FIN 数据包，对于这种情况，则通过重置 2MSL 定时器对这些重发的 FIN 数据包进行接收和处理。对于 TCP_CLOSE 状态，则无需进行任何处理，有关的处理在其他函数关联部分会进行，对于 tcp_ack 函数就不用做什么事了。缺省状态的处理主要是对几个定时器进行检查更新。结合该模块执行的条件加上此处本身的判断条件，这些语句应不难理解。

下面将进入到几个关键状态的更新处理。不过首先判断是否需要将用于小量数据收集的数据包发送出去。允许发送的条件是，本地所有其他数据包都已经得到处理（即已发送被得到应答）。为何在 tcp_ack 函数中插入该语句，原因是只有在 tcp_ack 函数中才会同时对写队列和重发队列进行清理。我们知道网络代码最核心的理念是完成数据的传送和接收，所以一有机会，就将数据包发送出去，在这一点网络代码是相当的“贪婪”和“得寸进尺”的。这也是一个网络栈实现应有的理念。

```
3087            /*
3088             *  We have nothing queued but space to send. Send any partial
3089             *  packets immediately (end of Nagle rule application).
3090             */

3091            if (sk->packets_out == 0 && sk->partial != NULL &&
3092                skb_peek(&sk->write_queue) == NULL && sk->send_head == NULL)
3093            {
3094                flag |= 1;
3095                tcp_send_partial(sk);
3096            }
```

好了，既然该发送的都发送了，现在终于可以处理状态的更新了。首先是 TCP_LAST_ACK 状态，代码如下：

```
3097            /*
3098             * In the LAST_ACK case, the other end FIN'd us.  We then FIN'd them, and
3099             * we are now waiting for an acknowledge to our FIN.  The other end is
3100             * already in TIME_WAIT.
3101             *
3102             * Move to TCP_CLOSE on success.
3103             */
```

```
3104            if (sk->state == TCP_LAST_ACK)
3105            {
3106                    if (!sk->dead)
3107                            sk->state_change(sk);
3108                    if(sk->debug)
3109                            printk("rcv_ack_seq: %lX==%lX, acked_seq: %lX==%lX\n",
3110                            sk->rcv_ack_seq,sk->write_seq,sk->acked_seq,sk->fin_seq);
3111            if (sk->rcv_ack_seq == sk->write_seq /*&& sk->acked_seq == sk->fin_seq*/)
3112            {
3113                    flag |= 1;
3114                    tcp_set_state(sk,TCP_CLOSE);
3115                    sk->shutdown = SHUTDOWN_MASK;
3116            }
3117    }
```

处于 TCP_LAST_ACK 状态表示远端已经完成关闭操作，本地已经发送 FIN 数据包，双方连接的完全关闭现在寄希望于本地接收一个 ACK 数据包，处于 TCP_LAST_ACK 状态就是在等待这么一个 ACK 数据包。现在来了一个 ACK 数据包，如果序列号满足，那么终于可以“寿终正寝”了。此处 rcv_ack_seq 字段在前面代码中已经被更新为应答序列号值，write_seq 表示本地写入的最后一个字节的序列号（FIN 标志位当然是最一个序列号），如果二者相等，那么表示这个 ACK 正是对之前发送的 FIN 的应答，可以完全关闭连接了。

```
3118    /*
3119     *  Incoming ACK to a FIN we sent in the case of our initiating the close.
3120     *
3121     *  Move to FIN_WAIT2 to await a FIN from the other end. Set
3122     *  SEND_SHUTDOWN but not RCV_SHUTDOWN as data can still be coming in.
3123     */

3124            if (sk->state == TCP_FIN_WAIT1)
3125            {

3126                    if (!sk->dead)
3127                            sk->state_change(sk);
3128                    if (sk->rcv_ack_seq == sk->write_seq)
3129                    {
3130                            flag |= 1;
3131                            sk->shutdown |= SEND_SHUTDOWN;
3132                            tcp_set_state(sk, TCP_FIN_WAIT2);
3133                    }
3134            }
```

处于 TCP_FIN_WAIT1 表示本地属于首先关闭的一方，并且已经发送了 FIN 数据包，正在等待 ACK 数据包，从而进入到 TCP_FIN_WAIT2 状态，此处的比较同上，状态设置为

TCP_FIN_WAIT2。进入 TCP_FIN_WAIT2 状态后，仍然可以进行数据的接收，但不可再进行数据的发送，这也是为何连接关闭需要四个数据包的原因，一端发送 FIN 数据包只表示其关闭了发送通道，接收通道需要远端进行关闭，对于远端而言，此时只表示接收通道被关闭，发送通道需要其发送一个 FIN 数据包完成。读者不要混淆了，说道这儿，有一点需要提及，一般我们在分析代码时尤其是网络栈代码这种处理双方交互的情况，不能老是从本地的角度考虑问题，对于一个问题的理解，有时还需要从远端进行理解，换句话说，相同的网络栈实现是运行在所有主机上的，这个代码是同时作为本地和远端而存在的。

```
3135        /*
3136         *  Incoming ACK to a FIN we sent in the case of a simultaneous close.
3137         *
3138         *  Move to TIME_WAIT
3139         */

3140        if (sk->state == TCP_CLOSING)
3141        {

3142            if (!sk->dead)
3143                sk->state_change(sk);
3144            if (sk->rcv_ack_seq == sk->write_seq)
3145            {
3146                flag |= 1;
3147                tcp_time_wait(sk);
3148            }
3149        }
```

进入到 TCP_CLOSING 状态表示发生了双方同时关闭的操作，即本地发送 FIN 数据包后，在接收到对应的 ACK 数据包前，接收到远端发送的 FIN 数据包，在发送对此 FIN 数据包的应答后，此时将进入 TCP_CLOSING 状态，一旦接收到 ACK 数据包，则直接进入 TCP_TIME_WAIT 状态，进行 2MSL 等待。tcp_time_wait 函数设置进入 TCP_TIME_WAIT 状态并启动 2MSL 定时器。

```
3150        /*
3151         *  Final ack of a three way shake
3152         */

3153        if(sk->state==TCP_SYN_RECV)
3154        {
3155            tcp_set_state(sk, TCP_ESTABLISHED);
3156            tcp_options(sk,th);
3157            sk->dummy_th.dest=th->source;
3158            sk->copied_seq = sk->acked_seq;
3159            if(!sk->dead)
3160                sk->state_change(sk);
3161            if(sk->max_window==0)
```

```
3162                {
3163                    sk->max_window=32;   /* Sanity check */
3164                    sk->mss=min(sk->max_window,sk->mtu);
3165                }
3166            }
```

对于 TCP_SYN_RECV 状态的处理比较重要，在此处主要完成三路握手连接过程。进入 TCP_SYN_RECV 状态表示处于被动打开的一方接收到了远端发送的 SYN 数据包，并且发送了对应的 ACK 数据包，现在等待远端回复一个 ACK 数据包，即宣告连接建立的完成。此时接收到一个有效的 ACK 数据包（注意在检查出当前状态是 TCP_SYN_RECV 后，并无进行进一步的检查，这一点很容易理解，因为在连接建立之前没有其他数据包，没有乱序到达，重复到达等等问题，前面代码或者其他函数代码已经判断出这是一个有效序列号的应答，那么对于连接建立过程而言，这些判断已经足以，只要是 TCP_SYN_RECV 状态就可以直接进行处理，无需进一步的检查，此处也没什么好检查的），对于 TCP_SYN_RECV 的处理首先是设置状态为 TCP_ESTABLISHED,对 TCP 选项进行处理（获取 MSS 值），更新 sock 结构中相关字段，通知上层进程（阻塞于 accept 调用的进程）有可用的套接字连接了。

```
3167            /*
3168             * I make no guarantees about the first clause in the following
3169             * test, i.e. "(!flag) || (flag&4)".    I'm not entirely sure under
3170             * what conditions "!flag" would be true.    However I think the rest
3171             * of the conditions would prevent that from causing any
3172             * unnecessary retransmission.
3173             *      Clearly if the first packet has expired it should be
3174             * retransmitted.    The other alternative, "flag&2 && retransmits", is
3175             * harder to explain:    You have to look carefully at how and when the
3176             * timer is set and with what timeout.    The most recent transmission always
3177             * sets the timer.    So in general if the most recent thing has timed
3178             * out, everything before it has as well.    So we want to go ahead and
3179             * retransmit some more.    If we didn't explicitly test for this
3180             * condition with "flag&2 && retransmits", chances are "when + rto < jiffies"
3181             * would not be true.    If you look at the pattern of timing, you can
3182             * show that rto is increased fast enough that the next packet would
3183             * almost never be retransmitted immediately.    Then you'd end up
3184             * waiting for a timeout to send each packet on the retransmission
3185             * queue.    With my implementation of the Karn sampling algorithm,
3186             * the timeout would double each time.    The net result is that it would
3187             * take a hideous amount of time to recover from a single dropped packet.
3188             * It's possible that there should also be a test for TIME_WRITE, but
3189             * I think as long as "send_head != NULL" and "retransmit" is on, we've
3190             * got to be in real retransmission mode.
3191             *      Note that tcp_do_retransmit is called with all==1.    Setting cong_window
3192             * back to 1 at the timeout will cause us to send 1, then 2, etc. packets.
3193             * As long as no further losses occur, this seems reasonable.
```

```
3194             */

3195            if (((!flag) || (flag&4)) && sk->send_head != NULL &&
3196                        (((flag&2) && sk->retransmits) ||
3197                        (sk->send_head->when + sk->rto < jiffies)))
3198            {
3199                    if(sk->send_head->when + sk->rto < jiffies)
3200                            tcp_retransmit(sk,0);
3201                    else
3202                    {
3203                            tcp_do_retransmit(sk, 1);
3204                            reset_xmit_timer(sk, TIME_WRITE, sk->rto);
3205                    }
3206            }

3207            return(1);
3208    }
```

最后一段处理超时重发，其中的 if 语句判断条件有些难于理解，从主体上分为三个 AND 语句，以上代码中分为三行进行显示。第一个判断语句：(!flag) || (flag&4)，要为真的话，flag=0 或者 flag|4 等于 1，对于 flag=0 的情况表示这个一个纯粹的 ACK 数据包，其中只包括 IP，TCP 首部，无任何数据负载。对于 flag 变量中 bit2 的设置在窗口大小被缩减以及重新计算 RTO 值后才进了设置。此处的判断诚如其源代码中注释所述，意义不明。第二个判断条件是重发队列不为空，这个条件需要和其他条件结合理解方有意义。第三个条件是系统已经进入重发过程或者当前进行重发的定时器已经超时，此时需要进行数据包的重发（如果重发队列非空）。对于系统进入重发过程和当前定时器超时的情况，在进入 if 语句块后进行了划分，重发定时器超时的情况除了重新发送 send_head 队列中的数据包外，还需要加倍 RTO 值以及进行拥塞处理（tcp_retransmit 函数）；而对于系统进入重发过程，此时只是重新发送 send_head 队列中的数据包，并不进行拥塞处理和 RTO 值加倍，其中的思想是既然现在接收到一个有效的 ACK 应答，则表示线路或者远端可以响应了，之前发送的数据包可能由于之前的堵塞丢失了，与其等待远端明确的索要，不如本地主动重新发送一遍。当然在某些情况下，如果线路还是比较拥挤（之前的那个 ACK 数据包是跌跌撞撞跑到这边来的），这种重发会加剧线路的拥挤度，但我们总是喜欢想象好的方面。

函数最后返回 1，表示处理正常。

tcp_ack 函数代码较长，处理的方面也较多，在分析该函数之前，我们试图从理论上解析该函数应该实现的功能，从具体代码分析上，我们也是首先从功能上进行入手，然后才进入的代码解析，由于关联的部分较多，在分析上不免有些漏洞（加上本人理解有限），所以读者对于某些尚不明白的地方结合 TCP 协议相关介绍仔细捉摸，相信能够进行理解。tcp_ack 函数是 TCP 协议实现上几个关键函数之一，对于该函数的理解将大大加深对 TCP 协议本身的理解，希望读者进行多次分析，应尽量理解的更多更深。

```
3209    /*
```

```
3210        *    Process the FIN bit. This now behaves as it is supposed to work
3211  *    and the FIN takes effect when it is validly part of sequence
3212        *    space. Not before when we get holes.
3213        *
3214        *    If we are ESTABLISHED, a received fin moves us to CLOSE-WAIT
3215        *    (and thence onto LAST-ACK and finally, CLOSE, we never enter
3216        *    TIME-WAIT)
3217        *
3218        *    If we are in FINWAIT-1, a received FIN indicates simultaneous
3219        *    close and we go into CLOSING (and later onto TIME-WAIT)
3220        *
3221        *    If we are in FINWAIT-2, a received FIN moves us to TIME-WAIT.
3222        *
3223        */

3224      static int tcp_fin(struct sk_buff *skb, struct sock *sk, struct tcphdr *th)
3225      {
3226            sk->fin_seq = th->seq + skb->len + th->syn + th->fin;
```

将本地 sock 结构中 fin_seq 字段初始化为远端发送的 FIN 标志位序列号，这种多个字段相加的赋值方式原因是 FIN 数据包很有可能同时包含数据，所以此处根据接收到的该 FIN 数据包中初始序列号加上数据长度加上 FIN 标志位本身占用的一个序列号。注意此处也加上了对 SYN 标志位的处理，一般而言这种情况很难出现，即在建立连接的过程中同时进行关闭操作，在此处同时对 SYN 标志位进行检测应是为了保险起见。

```
3227            if (!sk->dead)
3228            {
3229                  sk->state_change(sk);
3230                  sock_wake_async(sk->socket, 1);
3231            }
```

在接收到 FIN 数据包后，相应的需要通知上层应用程序。state_change 函数指针指向函数完成的工作仅仅是唤醒睡眠在 sock 结构 sleep 队列中的进程。具体的，state_change 变量在 af_inet.c 文件中被初始化为 def_callback1,该函数实现如下：

```
//net/inet/af_inet.c
430  static void def_callback1(struct sock *sk)
431  {
432        if(!sk->dead)
433              wake_up_interruptible(sk->sleep);
434  }
```

sock_wake_async 函数定义在 socket.c 文件中，其作用下面将逐渐分析，不过首先需要指出 socket 结构中的两个字段（socket 结构的具体定义参见第一章中内容），如下所示。

```
struct wait_queue    **wait;          /* ptr to place to wait on */
```

```
struct fasync_struct    *fasync_list;/* Asynchronous wake up list */
```

相关结构定义如下：

```
// linux/wait.h
7     struct wait_queue {
8             struct task_struct * task;
9             struct wait_queue * next;
10    };

//linux/fs.h
246  struct fasync_struct {
247         int        magic;
248         struct fasync_struct *fa_next; /* singly linked list */
249         struct file              *fa_file;
250  };
```

wait 字段是一个真正的进程睡眠队列，当由于用户作出的要求暂时无法得到满足时（如读数据时底层无数据可读时），用户进程就将被置于该队列中睡眠。fasync_list 字段称为文件异步操作队列中，表示依赖于该套接字的当前文件，一般而言，一个套接字只对应一个文件描述符（即一个文件，这是由于不允许建立两个同样套接字），介绍了这两个结构定义，问题依然不明朗，我们需要看 sock_wake_faync 函数实现，该函数定义在 socket.c 文件中，如下：

```
//net/socket.c
414  int sock_wake_async(struct socket *sock, int how)
415  {
416         if (!sock || !sock->fasync_list)
417                return -1;
418         switch (how)
419         {
420                case 0:
421                       kill_fasync(sock->fasync_list, SIGIO);
422                       break;
423                case 1:
424                       if (!(sock->flags & SO_WAITDATA))
425                              kill_fasync(sock->fasync_list, SIGIO);
426                       break;
427                case 2:
428                       if (sock->flags & SO_NOSPACE)
429                       {
430                              kill_fasync(sock->fasync_list, SIGIO);
431                              sock->flags &= ~SO_NOSPACE;
432                       }
433                       break;
434         }
435         return 0;
436  }
```

sock_wake_faync 函数继续调用 kill_fasync 函数，不过从其传入给 kill_fasync 函数的参数我们也可大致推断出 kill_fasync 函数功能，第一个参数是 socket 结构中 fasync_list 指向的 fasync_struct 结构队列，第二个参数是一个中断信号标志，而 fasync_struct 结构中唯一有意义的参数时 file 结构（对应 fa_file 字段），file 结构中有包含有使用该 file 的用户进程号，好了，由于用户进程号，又给了需要的信号标志，则很明显 kill_fasync 函数将遍历 fasync_list 指向的队列，给其中每个 fasync_struct 结构对应的用户进程发送一个指定的信号量（此处为 SIGIO）。下面给出 kill_fasync 实际实现代码。

```
//fs/fcntl.c
174  void kill_fasync(struct fasync_struct *fa, int sig)
175  {
176         while (fa) {
177                 if (fa->magic != FASYNC_MAGIC) {
178                         printk("kill_fasync: bad magic number in "
179                                 "fasync_struct!\n");
180                         return;
181                 }
182                 if (fa->fa_file->f_owner > 0)
183                         kill_proc(fa->fa_file->f_owner, sig, 1);
184                 else
185                         kill_pg(-fa->fa_file->f_owner, sig, 1);
186                 fa = fa->fa_next;
187         }
188  }
```

对一个进程发送信号的结果是该进程被设置为 TASK_RUNNING 状态（即唤醒进程），并对表示进程的结构（task_struct）中设置相关标志位，一旦进程被调度运行，由于大部分进程是在处理内核部分代码时被设置为睡眠状态，此时被唤醒后，将继续执行内核代码，通过对被唤醒原因的检查，采取不同的方式，我们可以通过借助于 tcp_read 函数中如下代码片断进行说明。

```
/*tcp_read 函数代码片断*/
1862                    sk->socket->flags |= SO_WAITDATA;
1863                    schedule();
1864                    sk->socket->flags &= ~SO_WAITDATA;
1865                    sk->inuse = 1;

1866                    if (current->signal & ~current->blocked)
1867                    {
1868                            copied = -ERESTARTSYS;
1869                            break;
1870                    }
1871                    continue;
```

注意 tcp_read 函数中此段代码之前已经将当前进程状态设置为 TASK_INTERRUPTIBLE，表示可被信号唤醒的睡眠状态。

```
1812                    current->state = TASK_INTERRUPTIBLE;
```

1863 行代码通过 schedule 函数调用将该进程调度出去，由于其状态已经被设置为 TASK_INTERRUPTIBLE，而且 tcp_read 函数在函数开始处已经将该进程加入到 sock 结构 sleep

睡眠队列中，所以此处的 schedule 函数调用相当于将当前进程设置为睡眠挂起了。在被唤醒后，通过此处被唤醒的具体原因（是 wake_up_interruptible 函数调用方式还是信号中断方式），对于 wake_up_interruptible 函数调用方式则继续进行执行，否退出处理信号中断。设置这两种不同的方式来唤醒进程是因为内核必须对两种不同的情况进行区分：一是由于其他工作的完成，进程所请求的条件可能会得到满足，此时唤醒进程进行检查（对应 wake_up_interruptible 函数调用方式）；二是由于某些行为的方式，进程所要求的条件将永远无法得到满足，此时必须唤醒进程进行处理（对应信号中断方式）。对应于前文中接收到一个 FIN 数据包的情况，如果该 FIN 数据包不包含任何数据，则 tcp_read 函数将永远无法读取到数据，此时仅仅通过 wake_up_interruptible 调用将可能无法退出 while 循环（注意 1871 行的 continue 调用），那么这个进程就成为了一个不可被停止的进程。tcp_fin 函数中 3227-3231 行代码的作用就是防止这样的问题。另外 kill_proc,kill_pg 函数均定义在 kernel/exit.c 文件中，较为容易理解，读者可自行查看。

tcp_fin 函数接下来要完成的工作是检查当前套接字状态从而决定该如何响应远端发送的这个 FIN 数据包。

```
3232        switch(sk->state)
3233        {
3234            case TCP_SYN_RECV:
3235            case TCP_SYN_SENT:
3236            case TCP_ESTABLISHED:
3237                /*
3238                 * move to CLOSE_WAIT, tcp_data() already handled
3239                 * sending the ack.
3240                 */
3241                tcp_set_state(sk,TCP_CLOSE_WAIT);
3242                if (th->rst)
3243                    sk->shutdown = SHUTDOWN_MASK;
3244                break;
```

对应于以上这三种状态，本地套接字需要发送一个 ACK 数据包，并同时更新本地套接字状态，代码将状态设置为 TCP_CLOSE_WAIT, 对于后关闭一方的套接字而言，根据 TCP 协议规范，接收到远端发送的 FIN 数据包后，在回复一个 ACK 数据包后（由 tcp_data 函数完成），将进入 TCP_CLOSE_WAIT 状态。另外代码还对 RST 标志位进行了检查，如果 TCP 首部中 RST 标志位被设置，则表示本地还需要同时关闭发送通道，对于一个 FIN 数据包而言，一般只是表示本地接收通道被关闭。RST 标志位的含义本书多个地方涉及到，此处不再说明。

```
3245            case TCP_CLOSE_WAIT:
3246            case TCP_CLOSING:
3247                /*
3248                 * received a retransmission of the FIN, do
3249                 * nothing.
3250                 */
3251                break;
```

如果是这两种状态，则表示之前已经接收到 FIN 数据包，那么此时接收到的 FIN 数据包就是一个副本，无需重复处理，直接退出即可。

```
3252            case TCP_TIME_WAIT:
3253                /*
3254                 * received a retransmission of the FIN,
3255                 * restart the TIME_WAIT timer.
3256                 */
3257                reset_msl_timer(sk, TIME_CLOSE, TCP_TIMEWAIT_LEN);
3258                return(0);
```

进入 TCP_TIME_WAIT 状态表示本地已经关闭发送通道（即本地已经发送了 FIN 数据包，也得到了对方的应答），而且也接收到远端发送的 FIN 数据包，并回复了 ACK 数据包，正处于 2MSL 等待时间，此时又接收到一个 FIN 数据包，则很有可能本地发送的 ACK 数据包丢失或者长时间延迟在网络中，造成远端的超时重传，又发送了一个 FIN 数据包，对于这种情况，应当再次回复一个 ACK 数据包，并且重新设置 2MSL 定时器。以上代码只是重新设置了 2MSL 定时器，并没有回复一个 ACK 数据包，从整体实现来看，这不会造成什么影响，因为双方都设置了定时器，在定时器超期后，都会设置各自的套接字为 TCP_CLOSE 状态。注意 2MSL 等待时间的提出一方面是为了防止 ACK 数据包的丢失，另一方面是为了接收远端之前发送的在网络中受到延迟的其他一切数据包，所以每当从远端接收到一个这样的数据包，都会对 2MSL 定时器进行重置。

```
3259            case TCP_FIN_WAIT1:
3260                /*
3261                 * This case occurs when a simultaneous close
3262                 * happens, we must ack the received FIN and
3263                 * enter the CLOSING state.
3264                 *
3265                 * This causes a WRITE timeout, which will either
3266                 * move on to TIME_WAIT when we timeout, or resend
3267                 * the FIN properly (maybe we get rid of that annoying
3268                 * FIN lost hang). The TIME_WRITE code is already correct
3269                 * for handling this timeout.
3270                 */

3271                if(sk->ip_xmit_timeout != TIME_WRITE)
3272                    reset_xmit_timer(sk, TIME_WRITE, sk->rto);
3273                tcp_set_state(sk,TCP_CLOSING);
3274                break;
```

处于 TCP_FIN_WAIT1 状态表示本地已经发送了 FIN 数据包但尚未得到对方的应答，或者说处于该状态的本地套接字正在等待对方应答数据包，而此时却收到一个 FIN 数据包，这就发生了双方同时关闭的情况，对于这种情况，需要对接收到的 FIN 数据包回复一个应答数据包，然后将进入 TCP_CLOSING 状态，这样在接收到对方发送的应答数据包后，直接进入

TCP_TIME_WAIT 状态，而不是 TCP_FIN_WAIT2 状态。对于应答数据包的发送，在 tcp_data 函数可能已经得到处理，此处所做的工作只是重置定时器为超时重传定时器。注意虽然 sock 结构中定一个四个字段分别对应 TCP 协议中声明的四种定时器：超时重传定时器，保活（keepalive）定时器，窗口探测（probe）定时器，2MSL 定时器，但实现上前三个定时器使用的是同一个定时器（超时重传定时器），由 ip_xmit_timeout 字段表示定时器当前定时的目的。这一点可以从 reset_xmit_timer 函数的实现看出，reset_xmit_timer 函数在前文中已经介绍，此处只是给出其代码方便读者查看。

```
466 /*
467  *  Reset the retransmission timer
468  */

469 static void reset_xmit_timer(struct sock *sk, int why, unsigned long when)
470 {
471     del_timer(&sk->retransmit_timer);
472     sk->ip_xmit_timeout = why;
473     if((int)when < 0)
474     {
475         when=3;
476         printk("Error: Negative timer in xmit_timer\n");
477     }
478     sk->retransmit_timer.expires=when;
479     add_timer(&sk->retransmit_timer);
480 }
```

TCP_FIN_WAIT2 状态表示本地已经关闭发送通道，正在等待对方关闭发送通道从而完全关闭连接。在接收到对方发送的 FIN 数据包后，即可会送 ACK 数据包（由 tcp_data 函数完成），并进入 TCP_TIME_WAIT 状态，一旦进入该状态就需要相应的设置 2MSL 定时器进入 2MSL 等待时间。

```
3275            case TCP_FIN_WAIT2:
3276                /*
3277                 * received a FIN -- send ACK and enter TIME_WAIT
3278                 */
3279                reset_msl_timer(sk, TIME_CLOSE, TCP_TIMEWAIT_LEN);
3280                sk->shutdown|=SHUTDOWN_MASK;
3281                tcp_set_state(sk,TCP_TIME_WAIT);
3282                break;
3283            case TCP_CLOSE:
3284                /*
3285                 * already in CLOSE
3286                 */
3287                break;
```

对已处于 TCP_CLOSE 状态的套接字无需进行任何操作。

```
3288                default:
3289                        tcp_set_state(sk,TCP_LAST_ACK);

3290                        /* Start the timers. */
3291                        reset_msl_timer(sk, TIME_CLOSE, TCP_TIMEWAIT_LEN);
3292                        return(0);
```

其他状态（TCP_LISTEN, TCP_LAST_ACK），重置 2MSL 定时器（因为我们接收到对方发送的一个数据包）。对于 TCP_LISTEN 状态的套接字是不会接收到对方发送的 FIN 数据包，因为其只负责连接建立，之后所有的工作由新创建的套接字负责，所以要关闭一个 TCP_LISTEN 状态的套接字，只能是本地进行关闭。所以此处的 default 不含 TCP_LISTEN 状态，那么就只剩下 TCP_LAST_ACK 状态，处于该状态的套接字表示该套接字已经发送了 FIN 数据包，现在整个连接只差对方发送一个 ACK 数据包了，现在却收到一个 FIN 数据包，可能是远端出现了问题，这个本地管不了，本地所做的工作只是仍然等待，但等待不能无限期，所以设置一个定时器，一旦超时，无论这个最后的 ACK 数据包是否到来，都设置进入 TCP_CLOSE 状态，完全关闭连接。

```
3293            }

3294            return(0);
3295        }
```

tcp_fin 函数处理远端发送的 FIN 数据包，所采取的基本策略是本地在满足条件的情况下，发送一个 ACK 数据包，并相应的更新状态。对该函数有关具体说明请参考代码中分析。

```
3296        /*
3297         *  This routine handles the data.  If there is room in the buffer,
3298         *  it will be have already been moved into it.  If there is no
3299         *  room, then we will just have to discard the packet.
3300         */

3301        extern __inline__ int tcp_data(struct sk_buff *skb, struct sock *sk,
3302                unsigned long saddr, unsigned short len)
3303        {
```

tcp_data 函数处理被接收数据包中可能包含的数据，将数据挂接到 receive_queue 所指向的接收队列中。参数中 saddr 表示的是远端地址。参数 len 表示 TCP 首部及其负载的总长度。skb 即为接收到的数据包，sk 表示对应的本地套接字 sock 结构。

```
3304            struct sk_buff *skb1, *skb2;
3305            struct tcphdr *th;
```

```
3306            int dup_dumped=0;
3307            unsigned long new_seq;
3308            unsigned long shut_seq;

3309            th = skb->h.th;
3310            skb->len = len -(th->doff*4);

3311        /*
3312             *  The bytes in the receive read/assembly queue has increased. Needed for the
3313             *  low memory discard algorithm
3314             */

3315            sk->bytes_rcv += skb->len;

3316            if (skb->len == 0 && !th->fin)
3317            {
3318                /*
3319                 *  Don't want to keep passing ack's back and forth.
3320                 *  (someone sent us dataless, boring frame)
3321                 */
3322                if (!th->ack)
3323                    tcp_send_ack(sk->sent_seq, sk->acked_seq,sk, th, saddr);
3324                kfree_skb(skb, FREE_READ);
3325                return(0);
3326            }
```

代码实现计算负载数据的长度，将其保存在 skb->len 字段中，并更新 sock 结构 bytes_rcv 字段，该字段的含义顾名思义，表示套接字从创建之时起到目前为止接收到的总字节数。代码之后对负载数据长度和相关标志位进行了检查。如果负载数据长度为 0，且 FIN 标志位没有设置，此时无需对该数据包进一步进行处理，简单丢弃即可。3322 行还对数据包是否是一个 ACK 数据包进行了检查，实际上，对于 TCP 协议而言，除了三路握手连接中第一个数据包没有设置 ACK 标志位外，此后来往的所有数据包中 ACK 标志位均被设置为 1。所以如果将 3316 行与 3322 行的代码综合起来看，那就是这是一个没有负载用户数据，FIN，ACK 标志位都没有设置的数据包，那么该数据包就一定设置了 SYN 标志位（即是一个 SYN 数据包，对于 SYN 数据包，是不会执行到 tcp_data 函数的），否则对于本版本网络实现而言，这就是一个无效的数据包。所以 3322 行代码更多的是一种心理安慰（有了总比没有好）。

```
3327            /*
3328             *  We no longer have anyone receiving data on this connection.
3329             */

3330        #ifndef TCP_DONT_RST_SHUTDOWN

3331            if(sk->shutdown & RCV_SHUTDOWN)
```

```
3332        {
```

这个条件语句判断接收通道是否已经被关闭，对于接收通道被关闭的情况，如果此时接收到一个负载数据长度非 0 的数据包，则表示很可能出现问题，此时需要对可能的问题进行检查。

```
3333            /*
3334             *  FIXME: BSD has some magic to avoid sending resets to
3335             *  broken 4.2 BSD keepalives. Much to my surprise a few non
3336             *  BSD stacks still have broken keepalives so we want to
3337             *  cope with it.
3338             */

3339            if(skb->len)    /* We don't care if it's just an ack or
3340                            a keepalive/window probe */
3341            {
3342                new_seq= th->seq + skb->len + th->syn; /* edge of data part of frame */
```

对于一个接收通道被关闭的套接字，如果又接收到一个负载数据长度非 0 的数据包，则检查该数据包是否是一个被延迟的数据包，即查看其序列号是否是过期的。如果不是过期的（即序列号比 FIN 序列号还要大），则根据当前套接字是否正处于释放状态进行相应的处理。对于处于释放状态的套接字表示用户应用程序已经关闭该套接字，所以对于接收到的数据直接丢弃即可，不需要进行缓存了。我们姑且不管远端在发送了 FIN 数据包后，是如何又冒出了一个序号号在 FIN 序列号之后的数据包，从本地处理的角度看，只要是从远端接收到的数据，如果应用程序仍然在使用该套接字的话，就将数据缓存在接收队列中，等候用户进行处理。这种处理方式让我们的思维方式跳出了框框，一般我们将 FIN 数据包看作是权威的，只要接收了 FIN 数据包，就表示本地“不可”再从接收通道中接收数据（或者更本质的说，远端一定不会再发送数据了），实际上 FIN 数据包说到底只是一种约定，如果有一方不遵从约定，那就变成了什么也不是（就像远端明明已经关闭了其发送通道，现在又在发送数据包，简直是“儿戏”），既然如此，只要你发，我就收，至于怎么办，让用户操心去吧，我网络代码只能这么办了。3353-3370 行代码就是这个意思。

```
3343                /* Do this the way 4.4BSD treats it. Not what I'd
3344                    regard as the meaning of the spec but it's what BSD
3345                    does and clearly they know everything 8) */

3346                /*
3347                 *  This is valid because of two things
3348                 *
3349                 *  a) The way tcp_data behaves at the bottom.
3350                 *  b) A fin takes effect when read not when received.
3351                 */

3352                shut_seq=sk->acked_seq+1;   /* Last byte */
```

```
3353                    if(after(new_seq,shut_seq))
3354                    {
3355                        if(sk->debug)
3356                            printk("Data arrived on %p after close
                                    [Data right edge %lX, Socket shut on %lX] %d\n",
3357                                sk, new_seq, shut_seq, sk->blog);
3358                        if(sk->dead)
3359                        {
3360                            sk->acked_seq = new_seq + th->fin;
3361                            tcp_reset(sk->saddr, sk->daddr, skb->h.th,
3362                                sk->prot, NULL, skb->dev, sk->ip_tos, sk->ip_ttl);
3363                            tcp_statistics.TcpEstabResets++;
3364                            tcp_set_state(sk,TCP_CLOSE);
3365                            sk->err = EPIPE;
3366                            sk->shutdown = SHUTDOWN_MASK;
3367                            kfree_skb(skb, FREE_READ);
3368                            return 0;
3369                        }
3370                    }
3371                }
3372            }

3373      #endif
```

无论怎么折腾，既然可以执行到此处，就表示需要将这个数据包缓存到用户接收队列中去。下面代码就是完成这个工作，代码较长的原因是接收队列中缓存的数据包需要按序列号进行排序，由于数据包的乱序到达，所以将数据包插入到接收队列中的哪个位置还需要遍历队列中的现有数据包进行序列号比较。注意在进行序列号比较时，使用的是数据包中第一个数据字节的序列号，以及数据的长度，这样做的目的是可以检测到数据包重复（一个数据包完全覆盖另一个数据包）的情况，从而进行替换；对于重叠（一个数据包部分覆盖另一个数据包）的情况不进行替换，都插入到队列中，重叠的情况在 tcp_read 函数中会得到处理（至于怎么处理请参考前文中 tcp_read 函数的讨论）。

```
3374            /*
3375             *  Now we have to walk the chain, and figure out where this one
3376             *  goes into it.   This is set up so that the last packet we received
3377             *  will be the first one we look at, that way if everything comes
3378             *  in order, there will be no performance loss, and if they come
3379             *  out of order we will be able to fit things in nicely.
3380             *
3381             *  [AC: This is wrong. We should assume in order first and then walk
3382             *   forwards from the first hole based upon real traffic patterns.]
3383             *
```

```
3384            */

3385        if (skb_peek(&sk->receive_queue) == NULL)     /* Empty queue is easy case */
3386        {
3387            skb_queue_head(&sk->receive_queue,skb);
3388            skb1= NULL;
3389        }
```

接收队列中没有数据包，这没什么好说的，直接挂到队列首部即可。

```
3390        else
3391        {
3392            for(skb1=sk->receive_queue.prev; ; skb1 = skb1->prev)
3393            {
3394                if(sk->debug)
3395                {
3396                    printk("skb1=%p :", skb1);
3397                    printk("skb1->h.th->seq = %ld: ", skb1->h.th->seq);
3398                    printk("skb->h.th->seq = %ld\n",skb->h.th->seq);
3399                    printk("copied_seq = %ld acked_seq = %ld\n", sk->copied_seq,
3400                            sk->acked_seq);
3401                }
```

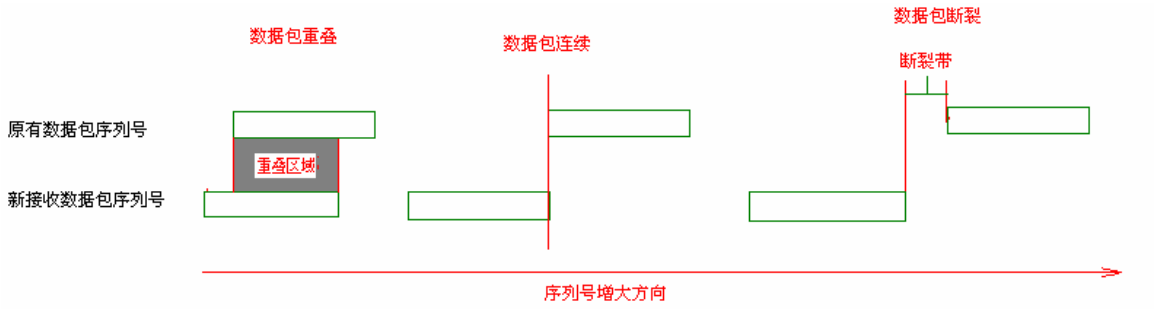
否则进行队列遍历，对队列中每个数据包进行检查，如果需要，打印调试信息，我们对此不关心。

```
3402                /*
3403                 *  Optimisation: Duplicate frame or extension of previous frame from
3404                 *  same sequence point (lost ack case).
3405                 *  The frame contains duplicate data or replaces a previous frame
3406                 *  discard the previous frame (safe as sk->inuse is set) and put
3407                 *  the new one in its place.
3408                 */

3409                if (th->seq==skb1->h.th->seq && skb->len>= skb1->len)
3410                {
3411                    skb_append(skb1,skb);
3412                    skb_unlink(skb1);
3413                    kfree_skb(skb1,FREE_READ);
3414                    dup_dumped=1;
3415                    skb1=NULL;
3416                    break;
3417                }
```

如果已有数据包与新接收的数据包开始序列号相同，但新接收的数据包长度较长，这就发生了覆盖的情况，此时删除已有旧数据包，将新接收的数据包挂接到原来的位置上。

```
3418                    /*
3419                     *    Found where it fits
3420                     */

3421                    if (after(th->seq+1, skb1->h.th->seq))
3422                    {
3423                            skb_append(skb1,skb);
3424                            break;
3425                    }
```

新接收的数据包序列号比已有的数据包序列号大，插入到该数据包之后，注意在 for 循环中，是从队列最后一个数据包向前遍历的，即从具有最大序列号的数据包开始比较的，所以一旦发送一个数据包序列号比当前接收的数据包的序列号小，则可以停止比较，将新接收的数据包插入该数据包之后即可。此时包括了数据重叠，连续，断裂的情况。如下图所示。



注意：以上所指序列号均指数据包中用户数据第一个字节所对应的序列号，即对应TCP首部中sequence字段。

```
3426                    /*
3427                     *    See if we've hit the start. If so insert.
3428                     */
3429                    if (skb1 == skb_peek(&sk->receive_queue))
3430                    {
3431                            skb_queue_head(&sk->receive_queue, skb);
3432                            break;
3433                    }
3434                }
3435        }//else
```

如果我们比较到最后，还没有找到一个序列号比当前所接收数据包小的原有数据包，则表示当前数据包序列号最小，此时应该插入到 receive_queue 接收队列的首部。receive_queue 接收队列中数据包是按其中所包含第一个数据字节序列号从小到大排列的。

将数据包插入的接收队列中正确位置后，下面就需要判断是否需要立刻发送一个 ACK 数据包以

及 ACK 数据包所使用的应答序列号。

```
3436        /*
3437         *  Figure out what the ack value for this frame is
3438         */

3439        th->ack_seq = th->seq + skb->len;
3440        if (th->syn)
3441            th->ack_seq++;
3442        if (th->fin)
3443            th->ack_seq++;
```

首先计算当前被接收数据包的应答序列号。并将该应答序列号暂存于所接收数据包 TCP 首部中应答序列号（ack_seq)字段。注意 TCP 首部中应答序列号字段表示的是远端希望从本地接收的下一个字节的序列号，同时也表示了在此序列号之前的数据都已被远端成功接收，本地可以从其重发队列中删除序列号（指数据包最后一个字节的序列号）在此应答序列号之前的数据包。

```
3444        if (before(sk->acked_seq, sk->copied_seq))
3445        {
3446            printk("*** tcp.c:tcp_data bug acked < copied\n");
3447            sk->acked_seq = sk->copied_seq;
3448        }
```

此段代码对 sock 结构中当前相关字段进行合法性检查，其中 acked_seq 字段表示本地当前应答对方的序列号或者说是期望从对方接收的下一个字节的序列号，而 copied_seq 表示已经提交给用户应用程序的最后一个字节的序列号加 1，所以 copied_seq 一定小于或等于 acked_seq，如果出现 copied_seq 大于 acked_seq 的情况，则表示内核出现不一致，可能存在系统 BUG，此处的修复是简单的将 acked_seq 赋值为 copied_seq,其中所基于的思想是，既然上层声称已接收到序列号到 copied_seq 的数据，而我所跟踪的情况是本地压根就没有请求到那一步，既然你上层这样说，那就依你，我就跳过中间的一段数据，直接请求你上层所要求的下一个数据，到时出了问题你上层自己解决。

```
3449    /*
3450    *Now figure out if we can ack anything. This is very messy because we really want two
3451     *receive queues, a completed and an assembly queue. We also want only one transmit
3452    *queue.
3453     */

3454        if ((!dup_dumped && (skb1 == NULL || skb1->acked)) ||
                                    before(th->seq, sk->acked_seq+1))
3455        {
```

如上 if 语句为真时需要满足如下条件之一：
1>dup_dump=0 并且 skb1=null 或者已经对 skb1 进行了应答。

2>th->seq<=sk->acked_seq，即当前所接收数据包中第一个字节序列号小于或等于本地当前从远端请求的下一个字节的序列号，对于这种情况，我们还需要进一步检查数据包中最后一个字节的序列号是否也在 acked_seq 之内，如果在这之内，这就表示这是一个重复数据包；否则该数据包就是一个合法的数据包。

对于条件一中，如果 dup_dump=0 的话，则 skb1 则一定不为 null，此时只可能 skb1->acked=1 即序列号比当前数据包稍小的数据包已经对其发送了一个 ACK 数据包给远端。对于这种情况的处理方式是并不立即对当前接收的这个数据包回复一个 ACK 数据包，而是等待接收下一个数据包，即一个 ACK 数据包对多个接收的数据包进行应答。所以对于条件一，我们还需要根据其他条件决定是否发送一个 ACK 数据包，暂时无需特别处理。而对于条件二，此时也表示条件以不满足，即此时可能已经积累了多个数据包没有回复 ACK 数据包了。对于这种情况我们进一步分析其代码。

```
                //对应以上分析中的第二种情况。以下我们分段进行说明。
3456            if (before(th->seq, sk->acked_seq+1))
3457            {
3458                int newwindow;

3459                if (after(th->ack_seq, sk->acked_seq))
3460                {
3461                    newwindow = sk->window-(th->ack_seq - sk->acked_seq);
3462                    if (newwindow < 0)
3463                        newwindow = 0;
3464                    sk->window = newwindow;
3465                    sk->acked_seq = th->ack_seq;
3466                }
3467                skb->acked = 1;
```

这是对应当前可能已经累积了未应答数据包的情况，并且当前所接收数据包中包含了本地尚未接收的新数据。此时根新窗口大小（从当前窗口中减去这些新的数据），并且检查更新后窗口大小，如果小于 0，则设置为 0 即可。另外更新当前应答序列号位当前所接收数据包的应答序列号，因为接下来就对刚刚接收到的这个数据包以及之前可能尚未进行应答的数据包一并应答一个数据包。针对前文中分析的第二种情况，无论当前所接收数据包中是否包含有新的数据，都需要进行应答，所以设置当前接收数据包 sk_buff 结构中 acked 应答标志字段为 1，表示该数据包已经进行了应答（接下来要完成的工作）。

```
3468                /*
3469                 *  When we ack the fin, we do the FIN
3470                 *  processing.
3471                 */

3472                if (skb->h.th->fin)
3473                {
3474                    tcp_fin(skb,sk,skb->h.th);
3475                }
```

检查当前所接收数据包是不是一个 FIN 数据包，如果是，则调用 tcp_fin 函数进行相应处理（tcp_fin 函数刚刚讨论，主要是对本地套接字状态的更新）。

```
3476                    for(skb2 = skb->next;
3477                        skb2 != (struct sk_buff *)&sk->receive_queue;
3478                        skb2 = skb2->next)
3479                    {
```

注意 skb 表示当前刚接收的数据包，在前面代码中已经被插入 receive_queue 接收队列中了，所以此处 for 循环遍历的是比当前刚接收的这个数据包序列号大的 receive_queue 队列中已有的数据包。

```
3480                        if (before(skb2->h.th->seq, sk->acked_seq+1))
3481                        {
3482                            if (after(skb2->h.th->ack_seq, sk->acked_seq))
3483                            {
3484                                newwindow = sk->window -
3485                                  (skb2->h.th->ack_seq - sk->acked_seq);
3486                                if (newwindow < 0)
3487                                    newwindow = 0;
3488                                sk->window = newwindow;
3489                                sk->acked_seq = skb2->h.th->ack_seq;
3490                            }
3491                            skb2->acked = 1;
3492                            /*
3493                              *    When we ack the fin, we do
3494                              *    the fin handling.
3495                              */
3496                            if (skb2->h.th->fin)
3497                            {
3498                                tcp_fin(skb,sk,skb->h.th);
3499                            }

3500                            /*
3501                              *    Force an immediate ack.
3502                              */

3503                            sk->ack_backlog = sk->max_ack_backlog;
3504                        }
```

3480-3504 这个 if 语句块是寻求对多个数据包同时进行应答的情况，加上 3476 的 for 语句，就是在寻找一个合适的应答序列号，从刚接收的数据包开始（见下面的分析），以序列号增大的顺序，分析每个数据包，寻找序列号断裂处。3480，3482，3489 行代码进行了数据包的连续性判

断，请读者仔细体会，应该可以理解。这三行的功能与 3456，3459，3465 这三行代码作用于当前所接收数据包的作用是一样的。如果读者稍微观察一下，就会发现，3480-3499 行代码 3456-3499 行代码完成的功能是一样的。所以刚刚说道在寻找一个合适的应答序列号时是从当前所接收的数据包开始检查的。另外一个值得注意的一点是，一旦 3480 行 if 语句为真，则 3503 行代码一定会得到执行。前面将当前所接收数据包插入 receive_queue 队列的过程已经保证了该队列中不存在重复数据包，所以一旦可以进入到 3480 行 if 语句块执行，则 3482 行所表示的语句块也一定会得到执行，换句话说，3480 行为真，则表示此时至少有两个数据包需要进行应答，将 sock 结构中 ack_backlog 字段设置为 max_ack_backlog 字段值，就是强制在下文中发送应答，所基于的思想是一旦有两个或两个以上数据包累积，则立刻回复一个应答数据包。即可以对多个数据包进行积累一并对他们进行应答，但数据包的个数达到两个，则必须立刻回复一个应答数据包。防止累积过多产生成累积的时间过长，从而造成远端超时重传。

```
3505                    else
3506                    {
3507                        break;
3508                    }
```

这个 else 语句表示出现序列号断裂，也表示此时已经寻找到了一个合适的应答序列号。

```
3509                }//for(skb2 = skb->next；...
```

注意在寻找合适的应答序列号的过程中，对于序列号连续的所有数据包其对应 sk_buff 结构中 acked 字段均被设置为 1，表示对应数据包已经发送了应答数据包。原因很简单，因为我们寻找应答序列号中的过程中，已经将应答序列号设置为包含这些数据包在内（这正是寻找的过程所进行的工作）。

```
3510                /*
3511             *  This also takes care of updating the window.
3512                 *  This if statement needs to be simplified.
3513                 */
3514                if (!sk->delay_acks ||
3515                    sk->ack_backlog >= sk->max_ack_backlog ||
3516                    sk->bytes_rcv > sk->max_unacked || th->fin) {
3517                    /*tcp_send_ack(sk->sent_seq, sk->acked_seq,sk,th, saddr); */
3518                }
3519                else
3520                {
3521                    sk->ack_backlog++;
3522                    if(sk->debug)
3523                        printk("Ack queued.\n");
3524                    reset_xmit_timer(sk, TIME_WRITE, TCP_ACK_TIME);
3525                }
3526            }
3527        }
```

既然找到了一个合适的应答序列号，那么就要根据情况进行应答数据包的发送了。3514-3525 行代码完成的工作即是如此。sock 结构中 delay_acks 标志位字段决定是否采用对数据包累积应答（而非一个数据包对应一个应答），如果值设置为 0，表示不使用累积应答，此时必须立刻发送一个应答数据包；否则检查是否有两个或两个以上数据包累积（即通过比较 ack_backlog 字段与 max_ack_backlog 字段值）；如果没有检查到累积的情况，那么进一步检查接收的数据字节数是否超过了 sock 结构设置未应答字节的最大值，如果超过，立刻回复应答数据包；再否则，则检查刚接收的数据包是不是一个 FIN 数据包，此时就不能累积其后的数据包一起进行应答了，因为这是对方发送的最后一个数据包了。只要以上条件之一满足，立刻发送一个应答数据包（调用 tcp_send_ack 函数完成，但是，注意调用 tcp_send_ack 函数进行 ACK 数据包发送的 3517 行被注释掉了，这点需要结合 3562-3565 行代码分析，见下文）。如果以上条件都不满足，则等待接收更多数据包进行累积应答。但这个等待的时间不能无限期，否则造成远端超时重传，无谓的增加网络拥挤度。所以设置一个定时器，定时器到期后，无论有没有累积到数据包，都要立刻回复一个应答数据包。可以想见，这个定时时间间隔一定小于远端设置的超时重传间隔（实际上也是本地设置的超时重传间隔，从网络栈实现代码本身既是本地端，也是远端这点考虑的话）。

```
3528        /*
3529         *  If we've missed a packet, send an ack.
3530         *  Also start a timer to send another.
3531         */

3532        if (!skb->acked)
3533        {
```

对于 skb（表示当前接收到的数据包）中 acked 字段为 0 的情况，从前文代码（3456 行代码不满足）可以分析出，此时出现了序列号断裂的情况，即当前所接收的数据包并非本地想要的，其中第一个字节的序列号大于本地的应答序列号（也即本地期望从远端接收的下一个字节的序列号），此时很有可能出现了数据包丢失的情况，所以立刻发送一个 ACK 应答数据包告知对方这种情况，三次发送同样请求序列号的 ACK 数据包会引起远端快速重传机制，从而重传可能丢失的数据包，而非一定等待重传定时器超时（这个时间一般较长），这个过程从本地的角度而言，也被称为快速恢复机制。3532 行 if 语句块所进行的工作即是进行可能的快速恢复。

```
3534        /*
3535         *  This is important.  If we don't have much room left,
3536         *  we need to throw out a few packets so we have a good
3537         *  window.  Note that mtu is used, not mss, because mss is really
3538         *  for the send side.  He could be sending us stuff as large as mtu.
3539         */

3540            while (sk->prot->rspace(sk) < sk->mtu)
3541            {
3542                skb1 = skb_peek(&sk->receive_queue);
3543                if (skb1 == NULL)
```

```
3544                    {
3545                        printk("INET: tcp.c:tcp_data memory leak detected.\n");
3546                        break;
3547                    }

3548                    /*
3549                     *  Don't throw out something that has been acked.
3550                     */

3551                    if (skb1->acked)
3552                    {
3553                        break;
3554                    }

3555                    skb_unlink(skb1);
3556                    kfree_skb(skb1, FREE_READ);
3557                }// while (sk->prot->rspace(sk) < sk->mtu)
```

因为现在可能丢失序列号排在前面的数据包，接收到一些序列号大的排在后面的数据包，而这些排在后面的数据包有可能将本地接收缓冲区使用完，从而造成即使远端实行快速重发那些丢失的排在前面的数据包，也由于接收缓冲区满，造成本地窗口很小，限制远端的发送（注意远端发送数据包还需要受到本地所声明的窗口大小的节制），从而造成两难的境地：排在前面的数据包由于丢失需要重传，而排在后面的数据包占用了空间造成了前面的数据包不能被重传。这就是死锁。此处解决的方式是释放接收队列中一些数据包，空出接收空间，从而释放接收缓冲区空间，以接收那些丢失被重传的数据包。在释放接收队列中数据包时，很直接的想法即是释放那些序列号大的（即排在后面的）数据包。这样在后面还可以再对他们进行请求。此处的实际做法是：遍历接收队列，凡是没有回复应答的数据包都被作为释放对象，直到释放出足够的空间为止，这种方式的结果是从断裂处前后开始进行数据包释放，在都满足释放条件时，从序列号较小的开始释放。从上面寻找应答序列号的过程来看，这种做法显然不经济。所以此处是一个可以优化的地方。3540-3557 行代码完成这个遍历和释放的过程。

```
3558                tcp_send_ack(sk->sent_seq, sk->acked_seq, sk, th, saddr);
3559                sk->ack_backlog++;
3560                reset_xmit_timer(sk, TIME_WRITE, TCP_ACK_TIME);
3561            }
```

在释放出足够空间后，现在发送一个 ACK 数据包请求远端重传可能丢失的数据包。此处也设置一个定时器，这个定时器的作用需要和 sock 结构中 ack_backlog 字段结合起来分析。以上是定时器到期处理函数 retransmit_timer 中的代码片断：

```
// retransmit_timer
604     if (sk->ack_backlog && !sk->zapped)
605     {
606         sk->prot->read_wakeup (sk);
607         if (! sk->dead)
```

```
608                 sk->data_ready(sk,0);
609         }
```

其中 sk->prot->read_wakeup 指向 tcp_read_wakeup 函数，该函数作用的就是向远端发送一个 ACK 数据包。tcp_read_wakeup 函数以 sock 结构中 ack_backlog 字段值决定是否发送一个 ACK 数据包，如果该字段为 0，则直接返回。如下代码片断所示：

```
//tcp_read_wakeup
1586            if (!sk->ack_backlog)
1587                return;
```

如下 else 语句对应 skb->acked=1 的情况，此处就要提到 3517 行代码被注释掉了的情况了，那行未实现的功能在此处 else 模块中完成了。此处 else 模块实现有别于 3517 行的地方在于：3517 行明确要求了数据包累积，而此处并未明确要求，换句话说，既可以进行累积（对应 3476 行条件为真，查找应答序列号的过程），也可以不进行累积（对应 3476 行条件为假的情况）。

```
3562            else
3563            {
3564                tcp_send_ack(sk->sent_seq, sk->acked_seq, sk, th, saddr);
3565            }
```

到此为止，我们已经完成了数据包的接纳（挂入接收队列中正确位置上）以及寻找合适应答序列号进行了应答处理，现在我们需要通知上层应用有数据可用了，以防当前有进程阻塞读数据请求上。如下 3569-3574 行代码完成这个通知的工作。

```
3566            /*
3567             *  Now tell the user we may have some data.
3568             */

3569            if (!sk->dead)
3570            {
3571                    if(sk->debug)
3572                            printk("Data wakeup.\n");
3573                sk->data_ready(sk,0);
3574            }
3575            return(0);
3576        }
```

至此我们完成对 tcp_data 函数的分析，大体上，该函数完成如下工作：

1>完成数据包中数据合法性检查。

2>完成数据包插入到接收队列中的工作。

3>完成寻找合适应答序列号对数据包进行应答的工作。

4>完成对上层应用有可用数据的通知。

```
3577        /*
3578         *  This routine is only called when we have urgent data
```

```
3579        *    signalled. Its the 'slow' part of tcp_urg. It could be
3580        *    moved inline now as tcp_urg is only called from one
3581        *    place. We handle URGent data wrong. We have to - as
3582        *    BSD still doesn't use the correction from RFC961.
3583        */

3584       static void tcp_check_urg(struct sock * sk, struct tcphdr * th)
3585       {
3586            unsigned long ptr = ntohs(th->urg_ptr);

3587            if (ptr)
3588                 ptr--;
3589            ptr += th->seq;

3590            /* ignore urgent data that we've already seen and read */
3591            if (after(sk->copied_seq, ptr))
3592                 return;

3593            /* do we already have a newer (or duplicate) urgent pointer? */
3594            if (sk->urg_data && !after(ptr, sk->urg_seq))
3595                 return;

3596            /* tell the world about our new urgent pointer */
3597            if (sk->proc != 0) {
3598                 if (sk->proc > 0) {
3599                      kill_proc(sk->proc, SIGURG, 1);
3600                 } else {
3601                      kill_pg(-sk->proc, SIGURG, 1);
3602                 }
3603            }
3604            sk->urg_data = URG_NOTYET;
3605            sk->urg_seq = ptr;
3606       }
```

tcp_check_urg 函数被 tcp_urg 函数调用计算当前接收数据包中紧急数据指针值。本版本网络代码实现将紧急数据处理为一个字节，并且将 TCP 首部中紧急指针处理为指向紧急数据后的第一个字节。所以 3588 行将紧急指针减 1，从而使得紧急指针指向紧急数据本身；另外紧急指针是一个偏移量，是相对于 TCP 首部中 seq 字段而言的，所以 3589 加上这个 seq 字段获得紧急数据的绝对序列号并保存在 sock 结构 urg_seq 字段中。3591 行检查紧急指针是否已经在被用户读取的数据序列号之内，如果是，则没有必须标志出来，直接返回即可。3594 行检查是否之前已经接收了紧急数据，由于 sock 结构只能标识一个紧急数据位置，所以以绝对序列号最大的为主，如果此次紧急数据绝对序列号比之前声明的紧急数据序列号大，则保存本次声明的紧急数据序列号，否则直接忽略之。3597-3603 行代码用于通知该套接字对应的用户进程，有紧急数据到达，提示他们立刻进行读取。最后设置 sock 结构中 urg_seq 字段表示这个紧急数据字节序列号，并

将 urg_data 设置为 URG_NOTYET,表示紧急数据到达，但尚未从数据包中提取，urg_data 字段值将在 tcp_urg 函数中被检查，从而提取紧急数据字节。参考下面对 tcp_urg 函数的分析。

```
3607    /*
3608     *  This is the 'fast' part of urgent handling.
3609     */

3610    extern __inline__ int tcp_urg(struct sock *sk, struct tcphdr *th,
3611    unsigned long saddr, unsigned long len)
3612    {
3613        unsigned long ptr;

3614        /*
3615         *  Check if we get a new urgent pointer - normally not
3616         */

3617        if (th->urg)
3618            tcp_check_urg(sk,th);

3619        /*
3620         *  Do we wait for any urgent data? - normally not
3621         */

3622        if (sk->urg_data != URG_NOTYET)
3623            return 0;

3624        /*
3625         *  Is the urgent pointer pointing into this packet?
3626         */

3627        ptr = sk->urg_seq - th->seq + th->doff*4;
3628        if (ptr >= len)
3629            return 0;

3630        /*
3631         *  Ok, got the correct packet, update info
3632         */

3633        sk->urg_data = URG_VALID | *(ptr + (unsigned char *) th);
3634        if (!sk->dead)
3635            sk->data_ready(sk,0);
3636        return 0;
3637    }
```

tcp_urg 函数被 tcp_rcv 函数调用用于处理数据包可能的紧急数据，该函数通过检查 TCP 首部中 URG 标志位值判断是否有紧急数据到达，如果该标志位被设置为 1，则表示存在紧急数据，此时调用 tcp_check_urg 函数计算紧急数据位置，并将紧急数据绝对序列号保存在 sock 结构 urg_seq 字段中，同时设置 sock 结构 urg_data 字段值为 URG_NOTYET,表示尚未从数据包读取该紧急数据保存，tcp_urg 函数接下来主要完成就是读取该紧急数据并将其保存在 urg_data 字段中。3627 行代码计算紧急数据相对于接收数据包第一个数据字节的偏移量。如果紧急数据偏移量超出全部数据的长度，则直接返回。否则读取这个紧急数据（一个字节），保存到 urg_data 字段中，并同时设置 URG_VALID 标志位，表示 urg_data 字段表示一个有效的紧急数据。注意 3633 行的赋值，我们说紧急数据是一个字节，只占用低 8 比特，而 URG_VALID 常量使用高位比特，所以不会对紧急数据造成影响。最后 3634-3635 通知用户进程读取紧急数据。

```
3638    /*
3639     *    This will accept the next outstanding connection.
3640     */

3641    static struct sock *tcp_accept(struct sock *sk, int flags)
3642    {
3643        struct sock *newsk;
3644        struct sk_buff *skb;

3645      /*
3646       * We need to make sure that this socket is listening,
3647       * and that it has something pending.
3648       */

3649        if (sk->state != TCP_LISTEN)
3650        {
3651            sk->err = EINVAL;
3652            return(NULL);
3653        }

3654        /* Avoid the race. */
3655        cli();
3656        sk->inuse = 1;

3657        while((skb = tcp_dequeue_established(sk)) == NULL)
3658        {
3659            if (flags & O_NONBLOCK)
3660            {
3661                sti();
3662                release_sock(sk);
3663                sk->err = EAGAIN;
3664                return(NULL);
```

```
3665                }

3666                release_sock(sk);
3667                interruptible_sleep_on(sk->sleep);
3668                if (current->signal & ~current->blocked)
3669                {
3670                    sti();
3671                    sk->err = ERESTARTSYS;
3672                    return(NULL);
3673                }
3674                sk->inuse = 1;
3675            }
3676            sti();

3677            /*
3678             *  Now all we need to do is return skb->sk.
3679             */

3680            newsk = skb->sk;

3681            kfree_skb(skb, FREE_READ);
3682            sk->ack_backlog--;
3683            release_sock(sk);
3684            return(newsk);
3685        }
```

tcp_accept 函数是 accept 系统调用的传输层 TCP 协议对应实现函数，该函数实现比较简单：从侦听套接字接收队列中取数据包，查看其是否完成 TCP 三路握手建立过程（tcp_dequeue_established 函数完成的功能），如果没有完成，则无限期等待。否则返回数据包对应的 sock 结构。实际上本函数只是一个获取结果函数，并为完成任何实际工作，这些实际工作（TCP 连接建立）均有其他函数完成了（tcp_conn_request, tcp_ack 等）。就本函数独立的来看，只有一点需要指明，这在前文中已有交待：即侦听套接字接收队列中缓存的均是请求连接（完成或尚未完成连接过程）的数据包，即都是由客户端发送的 SYN 数据包。具体连接完成后的双方通信由新创建的数据通信套接字（即此处返回的 newsk，该新通信套接字是在 tcp_conn_request 函数中创建的，具体参照下文对 tcp_conn_request 函数的分析）负责。

```
3686        /*
3687         *  This will initiate an outgoing connection.
3688         */

3689        static int tcp_connect(struct sock *sk, struct sockaddr_in *usin, int addr_len)
3690        {
3691            struct sk_buff *buff;
3692            struct device *dev=NULL;
```

```
3693            unsigned char *ptr;
3694            int tmp;
3695            int atype;
3696            struct tcphdr *t1;
3697            struct rtable *rt;

3698            if (sk->state != TCP_CLOSE)
3699            {
3700                    return(-EISCONN);
3701            }

3702            if (addr_len < 8)
3703                    return(-EINVAL);

3704            if (usin->sin_family && usin->sin_family != AF_INET)
3705                    return(-EAFNOSUPPORT);

3706            /*
3707             *      connect() to INADDR_ANY means loopback (BSD'ism).
3708             */

3709            if(usin->sin_addr.s_addr==INADDR_ANY)
3710                    usin->sin_addr.s_addr=ip_my_addr();

3711      /*
3712             *      Don't want a TCP connection going to a broadcast address
3713             */

3714            if      ((atype=ip_chk_addr(usin->sin_addr.s_addr))     ==      IS_BROADCAST    ||
atype==IS_MULTICAST)
3715                    return -ENETUNREACH;

3716            sk->inuse = 1;
3717            sk->daddr = usin->sin_addr.s_addr;
3718            sk->write_seq = tcp_init_seq();
3719            sk->window_seq = sk->write_seq;
3720            sk->rcv_ack_seq = sk->write_seq -1;
3721            sk->err = 0;
3722            sk->dummy_th.dest = usin->sin_port;
3723            release_sock(sk);

3724            buff = sk->prot->wmalloc(sk,MAX_SYN_SIZE,0, GFP_KERNEL);
3725            if (buff == NULL)
3726            {
```

```
3727            return(-ENOMEM);
3728        }
3729        sk->inuse = 1;
3730        buff->len = 24;
3731        buff->sk = sk;
3732        buff->free = 0;
3733        buff->localroute = sk->localroute;

3734        t1 = (struct tcphdr *) buff->data;

3735        /*
3736         *  Put in the IP header and routing stuff.
3737         */

3738        rt=ip_rt_route(sk->daddr, NULL, NULL);


3739        /*
3740         *  We need to build the routing stuff from the things saved in skb.
3741         */

3742        tmp = sk->prot->build_header(buff, sk->saddr, sk->daddr, &dev,
3743                                IPPROTO_TCP, NULL, MAX_SYN_SIZE,sk->ip_tos,sk->ip_ttl);
3744        if (tmp < 0)
3745        {
3746            sk->prot->wfree(sk, buff->mem_addr, buff->mem_len);
3747            release_sock(sk);
3748            return(-ENETUNREACH);
3749        }

3750        buff->len += tmp;
3751        t1 = (struct tcphdr *)((char *)t1 +tmp);

3752        memcpy(t1,(void *)&(sk->dummy_th), sizeof(*t1));
3753        t1->seq = ntohl(sk->write_seq++);
3754        sk->sent_seq = sk->write_seq;
3755        buff->h.seq = sk->write_seq;
3756        t1->ack = 0;
3757        t1->window = 2;
3758        t1->res1=0;
3759        t1->res2=0;
3760        t1->rst = 0;
3761        t1->urg = 0;
3762        t1->psh = 0;
```

```
3763            t1->syn = 1;
3764            t1->urg_ptr = 0;
3765            t1->doff = 6;
3766            /* use 512 or whatever user asked for */

3767            if(rt!=NULL && (rt->rt_flags&RTF_WINDOW))
3768                    sk->window_clamp=rt->rt_window;
3769            else
3770                    sk->window_clamp=0;
```

```
/* 以下代码说明作者对于 MTU，MSS 概念不是很清楚，所以这段对于 MTU 赋值，
 * 以下对于 TCP
* 首部中 MSS 选项值的选定都不正确!
* MTU 表示最大传输单元长度，包括 IP 首部及其数据（TCP 首部以及 TCP 数据负载）。
* MSS 表示用户数据长度，即 TCP 数据负载部分（不包括 TCP 首部）
*/
3771            if (sk->user_mss)
3772                    sk->mtu = sk->user_mss;
3773            else if(rt!=NULL && (rt->rt_flags&RTF_MTU))
3774                    sk->mtu = rt->rt_mss;
3775            else
3776            {
3777    #ifdef CONFIG_INET_SNARL
3778                    if ((sk->saddr ^ sk->daddr) & default_mask(sk->saddr))
3779    #else
3780                    if ((sk->saddr ^ sk->daddr) & dev->pa_mask)
3781    #endif
3782                            sk->mtu = 576 - HEADER_SIZE;
3783                    else
3784                            sk->mtu = MAX_WINDOW;
3785            }
3786            /*
3787             *      but not bigger than device MTU
3788             */

3789            if(sk->mtu <32)
3790                    sk->mtu = 32; /* Sanity limit */

3791            sk->mtu = min(sk->mtu, dev->mtu - HEADER_SIZE);

3792            /*
3793             *      Put in the TCP options to say MTU.
3794             */
```

```
3795            ptr = (unsigned char *)(t1+1);
3796            ptr[0] = 2;
3797            ptr[1] = 4;
3798            ptr[2] = (sk->mtu) >> 8;
3799            ptr[3] = (sk->mtu) & 0xff;
3800            tcp_send_check(t1, sk->saddr, sk->daddr,
3801                    sizeof(struct tcphdr) + 4, sk);

3802            /*
3803             *    This must go first otherwise a really quick response will get reset.
3804             */

3805            tcp_set_state(sk,TCP_SYN_SENT);
3806            sk->rto = TCP_TIMEOUT_INIT;
3807        #if 0 /* we already did this */
3808            init_timer(&sk->retransmit_timer);
3809        #endif
3810            sk->retransmit_timer.function=&retransmit_timer;
3811        sk->retransmit_timer.data = (unsigned long)sk;
3812            reset_xmit_timer(sk, TIME_WRITE, sk->rto);    /* Timer for repeating the SYN until
an answer */
3813            sk->retransmits = TCP_SYN_RETRIES;

3814            sk->prot->queue_xmit(sk, dev, buff, 0);
3815            reset_xmit_timer(sk, TIME_WRITE, sk->rto);
3816            tcp_statistics.TcpActiveOpens++;
3817            tcp_statistics.TcpOutSegs++;

3818            release_sock(sk);
3819            return(0);
3820        }
```

tcp_connect 函数是 connect 系统调用传输层对于 TCP 协议的实现。该函数实现的功能顾名思义是发送 SYN 请求连接数据包，用在主动打开（active open）的一端（一般我们称为客户端）。对于该函数并没有什么值得特别说明的地方，需要注意的是该函数中对于 TCP 首部各字段的赋值以及对于 socket 状态的更新（设置为 TCP_SYN_SENT）和最后超时重发定时器的设置（注意最初定时时间间隔为 TCP_TIMEOUT_INIT）。TCP 协议连接建立期间通报各自的 MSS 值，这是作为 TCP 选项而存在的，在 TCP 连接建立时，这个选项是必须的。函数中对于 MSS 值的设置有些混淆，其直接使用 MTU 值，而其于 MSS 是完全不同的。MTU 表示最大传输单元大小，解释为 IP 首部及其数据负载，IP 数据负载即包括 TCP 首部和 TCP 数据负载。而 MSS 值表示最大段大小（Maximum Segment Size），仅指 TCP 数据负载大小。
即：MSS=MTU — sizeof(ip header) — sizeof(tcp header). 实际上，本版本网络栈实现代码对于很多数据（如分配单个数据包的缓存大小）处理的都比较随意，不可仔细考量！

```
3821    /* This functions checks to see if the tcp header is actually acceptable. */
3822    extern __inline__ int tcp_sequence(struct sock *sk, struct tcphdr *th, short len,
3823                    struct options *opt, unsigned long saddr, struct device *dev)
3824    {
3825            unsigned long next_seq;

3826            next_seq = len - 4*th->doff;
3827            if (th->fin)
3828                    next_seq++;
3829            /* if we have a zero window, we can't have any data in the packet.. */
3830            if (next_seq && !sk->window)
3831                    goto ignore_it;
3832            next_seq += th->seq;

3833            /*
3834             * This isn't quite right.    sk->acked_seq could be more recent
3835             * than sk->window.    This is however close enough.    We will accept
3836             * slightly more packets than we should, but it should not cause
3837             * problems unless someone is trying to forge packets.
3838             */

3839            /* have we already seen all of this packet? */
3840            if (!after(next_seq+1, sk->acked_seq))
3841                    goto ignore_it;
3842            /* or does it start beyond the window? */
3843            if (!before(th->seq, sk->acked_seq + sk->window + 1))
3844                    goto ignore_it;

3845            /* ok, at least part of this packet would seem interesting.. */
3846            return 1;

3847    ignore_it:
3848            if (th->rst)
3849                    return 0;

3850            /*
3851             *    Send a reset if we get something not ours and we are
3852             *    unsynchronized. Note: We don't do anything to our end. We
3853             *    are just killing the bogus remote connection then we will
3854             *    connect again and it will work (with luck).
3855             */

3856            if (sk->state==TCP_SYN_SENT || sk->state==TCP_SYN_RECV)
3857            {
```

```
3858                tcp_reset(sk->saddr,sk->daddr,th,sk->prot,NULL,dev, sk->ip_tos,sk->ip_ttl);
3859                return 1;
3860        }

3861        /* Try to resync things. */
3862        tcp_send_ack(sk->sent_seq, sk->acked_seq, sk, th, saddr);
3863        return 0;
3864    }
```

tcp_sequence 函数用于检查接收的数据包序列号是否正确，或者更准确的说，是否需要对该数据包进行进一步的处理。

如果本地窗口为 0，数据包包含数据（注 FIN，SYN 均使用了一个序列号），则简单丢弃该数据包。窗口为 0 表示本地接收缓冲区满，无法继续接收远端数据包。

如果数据包是一个副本（之前已经接收了同样的一份数据包），则简单丢弃。
如果数据包序列号在本地窗口之外，则违背了远端窗口数据流节制机制，也简单丢弃。

如果序列号正确，则返回 1，内核继续处理该数据包，否则根据当前套接字状态，发送一个 RST 数据包复位对方套接字，或者是简单的发送一个 ACK 数据包，敷衍了事。

```
3865    /*
3866     *    When we get a reset we do this.
3867     */

3868    static int tcp_std_reset(struct sock *sk, struct sk_buff *skb)
3869    {
3870        sk->zapped = 1;
3871        sk->err = ECONNRESET;
3872        if (sk->state == TCP_SYN_SENT)
3873            sk->err = ECONNREFUSED;
3874        if (sk->state == TCP_CLOSE_WAIT)
3875            sk->err = EPIPE;
3876    #ifdef TCP_DO_RFC1337
3877        /*
3878         *    Time wait assassination protection [RFC1337]
3879         */
3880        if(sk->state!=TCP_TIME_WAIT)
3881        {
3882            tcp_set_state(sk,TCP_CLOSE);
3883            sk->shutdown = SHUTDOWN_MASK;
3884        }
3885    #else
3886        tcp_set_state(sk,TCP_CLOSE);
```

```
3887            sk->shutdown = SHUTDOWN_MASK;
3888        #endif
3889            if (!sk->dead)
3890                sk->state_change(sk);
3891            kfree_skb(skb, FREE_READ);
3892            release_sock(sk);
3893            return(0);
3894        }
```

tcp_std_reset 函数负责处理接收到的 RST 数据包，RST 数据包用于对连接进行复位（简单的说，即断开连接）。如果我们发送一个请求连接数据包，远端并无对应的服务存在，在远端回送一个 RST 数据包，将本地请求复位，上层显示可能为：connection refused.
sock 结构 zapped 字段专门用于标志被复位套接字。当 zapped 字段被设置为 1 时，表示本地套接字已被复位（即远端断开了本地的连接或者请求），对于已被复位的连接，很多工作就无需进行，其它相关函数在处理之前会检查该字段。tcp_std_reset 函数主要实现功能为根据当前套接字的状态相应的设置新的状态。如果该套接字仍在使用（dead 字段为 0），则通过回调函数通知上层应用该套接字状态的改变（sk->state_change 调用）。

```
4305        /*
4306         *    A window probe timeout has occurred.
4307         */

4308        void tcp_send_probe0(struct sock *sk)
4309        {
4310            if (sk->zapped)
4311            return;         /* After a valid reset we can send no more */

4312            tcp_write_wakeup(sk);

4313            sk->backoff++;
4314            sk->rto = min(sk->rto << 1, 120*HZ);
4315            reset_xmit_timer (sk, TIME_PROBE0, sk->rto);
4316            sk->retransmits++;
4317            sk->prot->retransmits ++;
4318        }
```

tcp_send_probe0 函数用于进行远端窗口探测。远端窗口是一个远端对本地进行数据流节制的机制。本地发送的数据包不可超过远端通报的窗口范围。本质上，远端窗口大小表示远端当前可用接收缓冲区的大小。如果远端窗口大小通报为 0，表示远端接收缓冲区已满，本地不可继续发送数据包。当远端应用程序读取部分数据包后，远端将发送一个窗口通报数据包，本地在接收到该数据包后，方可继续发送待发送数据包。这种机制的隐患是，一旦远端通报非 0 窗口大小的数据包丢失了，则双方将陷入死锁：本地等待非 0 窗口通报数据包，远端认为其已经发送非 0 通报数据包，等待本地发送其它数据。为了防止非 0 窗口数据包可能丢失引起的死锁，本地在接收到远端 0 窗口数据包后，启动窗口探测定时器，每隔一段时间发送一个窗口探测数据

包，如此即便对方非 0 窗口通报数据包发送丢失的行为，在接收该窗口探测数据包，其发送的 ACK 数据包中仍将通报一个非 0 窗口，从而解除之前可能的死锁行为。

具体的，tcp_send_probe0 函数是通过调用 tcp_write_wakeup 函数完成窗口探测数据包的发送。tcp_write_wakeup 函数前文中已有论述，其发送一个旧序列号的数据包，这样做主要是因为目的是探测，而非是发送数据。使用旧序列号的原因还在于与其它函数（主要指 tcp_sequeue）配合使用，实际上探测数据包并非一定要使用一个旧序列号，使用当前序列号也无不可。从 tcp_sequeuce 函数如下实现片断即可看出使用旧序列号的原因：

```
……
if (!after(next_seq+1, sk->acked_seq))
    goto ignore_it;
……

ignore_it:
……
tcp_send_ack(sk->sent_seq, sk->acked_seq, sk, th, saddr);
```

探测数据包的目的主要是接收一个远端数据包（无论是 ACK 数据包还是普通数据包，其都会包含本地需要知道的远端窗口大小），最好是 ACK 数据包（极端情况是可能本地也是 0 窗口，此时只能接收 ACK 数据包），使用旧序列号以及 tcp_sequence 以上这段代码的配合，本地可以达到接收一个远端 ACK 数据包的目的。
注意 tcp_send_probe0 函数对于探测数据包也使用指数退避算法计算发送时间间隔。

```
4319    /*
4320     *    Socket option code for TCP.
4321     */

4322    int tcp_setsockopt(struct sock *sk, int level, int optname, char *optval, int optlen)
4323    {
4324        int val,err;

4325        if(level!=SOL_TCP)
4326            return ip_setsockopt(sk,level,optname,optval,optlen);

4327        if (optval == NULL)
4328            return(-EINVAL);

4329        err=verify_area(VERIFY_READ, optval, sizeof(int));
4330        if(err)
4331            return err;

4332        val = get_fs_long((unsigned long *)optval);
```

```
4333            switch(optname)
4334            {
4335                    case TCP_MAXSEG:
4336        /*
4337         * values greater than interface MTU won't take effect.    however at
4338         * the point when this call is done we typically don't yet know
4339         * which interface is going to be used
4340         */
4341                            if(val<1||val>MAX_WINDOW)
4342                                    return -EINVAL;
4343                            sk->user_mss=val;
4344                            return 0;
4345                    case TCP_NODELAY:
4346                            sk->nonagle=(val==0)?0:1;
4347                            return 0;
4348                    default:
4349                            return(-ENOPROTOOPT);
4350            }
4351    }
```

tcp_setsockopt 函数用于处理 TCP 选项设置，本版本支持的可设置选项为 MSS 值和 Nagle 算法使能。如果选项不是针对 TCP，则调用 ip_setsocket 函数进行处理。

Nagle 算法是为解决网络中充斥大量小数据包问题而设计的。其算法基本思想是：如果之前发送的一个数据包尚未得到应答，则本地不可继续发送其它数据包。欢聚换说，Nagle 算法实现的是等待-发送策略，每次只能发送一个数据包，直到该数据包得到应答，方可发送下一个数据包。这种算法有利于在等待上一个数据包应答期间，尽量积累上层数据，以便在下一个数据包中一并发送出去，从而减少发送的数据包数量。Nagle 算法引起的问题是增加了程序之间交互的时间，有些情况下必须禁止使用 Nagle 算法，这些情况大多存在于客户-服务体系架构中，如 X 窗口服务必须禁止 Nagle 算法，如此方可对相关事件（如鼠标移动事件）作出快速反应，此时使用 Nagle 算法造成的延迟将是不可容忍的。

```
4352    int tcp_getsockopt(struct sock *sk, int level, int optname, char *optval, int *optlen)
4353    {
4354            int val,err;

4355            if(level!=SOL_TCP)
4356                    return ip_getsockopt(sk,level,optname,optval,optlen);

4357            switch(optname)
4358            {
4359                    case TCP_MAXSEG:
4360                            val=sk->user_mss;
4361                            break;
```

```
4362                case TCP_NODELAY:
4363                    val=sk->nonagle;
4364                    break;
4365                default:
4366                    return(-ENOPROTOOPT);
4367            }
4368            err=verify_area(VERIFY_WRITE, optlen, sizeof(int));
4369            if(err)
4370                return err;
4371            put_fs_long(sizeof(int),(unsigned long *) optlen);

4372            err=verify_area(VERIFY_WRITE, optval, sizeof(int));
4373            if(err)
4374                return err;
4375            put_fs_long(val,(unsigned long *)optval);

4376            return(0);
4377        }
```

tcp_getsockopt 函数用于获取选项值，其与 tcp_setsockopt 函数对应，此处不再叙述。

至此，关于 TCP 协议还剩下最后一个函数 tcp_rcv 尚未涉及，该函数是 TCP 协议数据包接收的总入口函数，网络层协议（如 IP 协议）在判断数据包使用的是 TCP 协议后，将调用 tcp_rcv 函数对该数据包进行传输层的相关处理。从实现上看，tcp_rcv 函数更像是一个任务分发器，其根据数据包中各标志位的设置，将数据包进一步派送给其它相关函数进行具体处理，其本身并不进行细节上的处理工作。

从大的方面来看，可以将数据包分为以下几种类型：SYN 请求连接数据包，ACK 应答数据包，RST 数据包，普通数据包，FIN 断开连接数据包。
其中在 TCP 连接期间，ACK 应答数据包和普通数据包是作为一个数据包来传输的。即数据包中包含普通数据且 TCP 首部中 ACK 字段被设置为 1：也即发送本地数据的同时也对已成功接收的远端数据进行应答。

由此，tcp_rcv 函数从功能上看，主要有以下几个模块组成：
1> 数据包合法性检测模块，对应函数：tcp_sequence
2> 请求连接处理模块，对应函数：tcp_conn_request
3> RST 数据包处理模块，对应函数：tcp_reset
4> 应答处理模块(完成连接建立模块)，对应函数：tcp_ack
5> 数据处理模块，对应函数：tcp_urg, tcp_data
6> 断开连接处理模块（此时指处理对方发送的 FIN 数据包），对应函数：tcp_fin

下面我们将分段对该函数进行分析，首先对函数参数进行说明。

```
3895    /*
3896     *  A TCP packet has arrived.
```

```
3897        */

3898        int tcp_rcv(struct sk_buff *skb, struct device *dev, struct options *opt,
3899                unsigned long daddr, unsigned short len,
3900                unsigned long saddr, int redo, struct inet_protocol * protocol)
3901        {
```

参数说明：
skb：表示被接收的数据包。
dev：表示接收该数据包的网络设备，对于每个网络接收设备，内核均以 device 结构表示。
opt：表示被接收数据包可能的 IP 选项，IP 选项的处理是在 do_options（ip.c）函数中完成的。
daddr：IP 首部中的远端地址字段值，所以从本地接收的角度看，指的是本地 IP 地址。
len：IP 数据负载的长度：包括 TCP 首部以及 TCP 数据负载。
saddr：IP 首部中源端 IP 地址，从本地接收的角度，指的是发送端 IP 地址。
redo：这是一个标志位，准确地说，tcp_rcv 函数在两个地方被调用，其一就是上文中刚刚提到的，被下层网络层模块调用，用于接收新的数据包，这是 redo 标志位设置为 0，表示这是一个新的数据包；其二就是在 release_sock 函数中被调用，release_sock 函数对先前缓存在 sock 结构 back_log 队列中的数据包调用 tcp_rcv 函数进行重新接收。而 back_log 中数据包是由 tcp_rcv 函数进行缓存的，读者在下文中即可看到，当 tcp_rcv 函数发送套接字当前正忙时（sock 结构 inuse 字段为 1），则将数据包缓存于 back_log 队列中后直接返回，此后由 release_sock 函数负责将数据包再次递给 tcp_rcv 函数进行重新处理，此时 redo 字段即被设置为 1，表示这是先前被缓存数据包的再次处理。

protocol：这是一个 inet_protocol 结构类型的变量，表示该套接字所使用的协议以及协议对应的接收函数。inet_protocol 结构定义在 protocol.h 文件中，如下：

```
/*net/inet/protocol.h*/
24  /* This is used to register protocols. */
25  struct inet_protocol {
26      int             (*handler)(struct sk_buff *skb, struct device *dev,
27                              struct options *opt, unsigned long daddr,
28                              unsigned short len, unsigned long saddr,
29                              int redo, struct inet_protocol *protocol);
30      int             (*frag_handler)(struct sk_buff *skb, struct device *dev,
31                              struct options *opt, unsigned long daddr,
32                              unsigned short len, unsigned long saddr,
33                              int redo, struct inet_protocol *protocol);
34      void            (*err_handler)(int err, unsigned char *buff,
35                                  unsigned long daddr,
36                                  unsigned long saddr,
37                                  struct inet_protocol *protocol);
38      struct inet_protocol *next;
39      unsigned char       protocol;
40      unsigned char       copy:1;
```

```
41      void            *data;
42      char            *name;
43  };
```

而相关实例定义在 protocol.c 文件中，对于 TCP 协议对应的 inet_protocol 结构如下：

```
/*net/inet/protocol.c*/
44  static struct inet_protocol tcp_protocol = {
45      tcp_rcv,        /* TCP handler          */
46      NULL,           /* No fragment handler (and won't be for a long time) */
47      tcp_err,        /* TCP error control    */
48      NULL,           /* next                 */
49      IPPROTO_TCP,    /* protocol ID          */
50      0,          /* copy                 */
51      NULL,           /* data                 */
52      "TCP"           /* name                 */
53  };
```

这个结构定义了对应协议数据包接收函数，错误处理函数，协议编号（TCP 为 6）等等。UDP，ICMP 协议都有这样的一个结构对应，在分析 protocol.c 文件时读者将会看到。这些结构被网络层模块使用，其根据 IP 首部中上层协议字段值判断该调用哪个函数进行数据包的进一步处理。对于 TCP 协议，这个处理函数为 tcp_rcv。

在明白参数含义后，我们进入函数主体的分析，首先是对数据包合法性的检查。

```
3902        struct tcphdr *th;
3903        struct sock *sk;
3904        int syn_ok=0;

3905        if (!skb)
3906        {
3907            printk("IMPOSSIBLE 1\n");
3908            return(0);
3909        }
            //数据包没有经过网口设备，怎们可能会被接收？
3910        if (!dev)
3911        {
3912            printk("IMPOSSIBLE 2\n");
3913            return(0);
3914        }

3915        tcp_statistics.TcpInSegs++;
            //如果不是发送给本地的数据包，在网络层就已经被处理，根本不会跑到
            //传输层来。
3916        if(skb->pkt_type!=PACKET_HOST)
```

```
3917        {
3918            kfree_skb(skb,FREE_READ);
3919            return(0);
3920        }
```

这些检查有些显得比较冗余，如对 skb 参数是否为 NULL 的检查；这些检查所依据的思想是保险。虽然我们按常理分析，程序应该不会出现某种情况，但由于计算机在运行程序过程后，有可能出现意想不到的极端情况，此时如果不对这些原本不应该出现的情况的监测，则进一步处理的话，会使系统崩溃。所以在系统代码的很多地方，都会出现这些对“不应该”出现的条件的监测。以上这段代码都是这个意思。

```
3921        th = skb->h.th;

3922        /*
3923         *  Find the socket.
3924         */

3925        sk = get_sock(&tcp_prot, th->dest, saddr, th->source, daddr);
```

get_sock 函数定义在 af_inet.c 中，用于根据 TCP 套接字四元素查询本地对应的 sock 结构。

```
3926        /*
3927         *  If this socket has got a reset it's to all intents and purposes
3928         *  really dead. Count closed sockets as dead.
3929         *
3930         *  Note: BSD appears to have a bug here. A 'closed' TCP in BSD
3931         *  simply drops data. This seems incorrect as a 'closed' TCP doesn't
3932         *  exist so should cause resets as if the port was unreachable.
3933         */

3934        if (sk!=NULL && (sk->zapped || sk->state==TCP_CLOSE))
3935            sk=NULL;
```

如果本地有对应的 sock 结构，但是本地状态显示该套接字已经被复位或者已经处于关闭状态，则不可接收该数据包，此时通过将 sk 变量设置为 NULL 来完成，下面代码中将有对该字段的检测，此处设置为 NULL，将阻止下文中的进一步处理。

```
3936        if (!redo)
3937        {
3938            if (tcp_check(th, len, saddr, daddr ))
3939            {
3940                skb->sk = NULL;
3941                kfree_skb(skb,FREE_READ);
3942                /*
```

```
3943                     *    We don't release the socket because it was
3944                     *    never marked in use.
3945                     */
3946                    return(0);
3947                }
3948                th->seq = ntohl(th->seq);

3949                /* See if we know about the socket. */
3950                if (sk == NULL)
3951                {
3952                    /*
3953                     *    No such TCB. If th->rst is 0 send a reset (checked in tcp_reset)
3954                     */
3955                    tcp_reset(daddr, saddr, th, &tcp_prot, opt,dev,skb->ip_hdr->tos,255);
3956                    skb->sk = NULL;
3957                    /*
3958                     *    Discard frame
3959                     */
3960                    kfree_skb(skb, FREE_READ);
3961                    return(0);
3962                }

3963                skb->len = len;
3964                skb->acked = 0;
3965                skb->used = 0;
3966                skb->free = 0;
3967                skb->saddr = daddr;
3968                skb->daddr = saddr;

3969                /* We may need to add it to the backlog here. */
3970                cli();
3971                if (sk->inuse)
3972                {
3973                    skb_queue_tail(&sk->back_log, skb);
3974                    sti();
3975                    return(0);
3976                }
3977                sk->inuse = 1;
3978                sti();
3979            }
```

3936-3979 行对应的 if 语句块表示这是一个新的数据包，所以对此数据包的合法性需要进行检查，如果数据包没有问题，则对表示该数据包的 skb 变量进行一些字段的赋值，合法性包括：
1>TCP 校验是否正确：tcp_check

2>本地是否有对应的套接字
如果以上两个条件都表示允许进一步的处理，则开始对表示该数据包的 sk_buff 结构中某些字段进行赋值，这对应 3963-3968 行语句。最后检查当前套接字是否正在忙于处理其他事务（sk->inuse 是否为 1），如果是，则先将数据缓存到 sock 结构 back_log 队列中，稍后由 release_sock 函数再次提交给本函数进行处理，此时对应的 redo 参数为 1，那么就进入下面的 else 模块。

```
3980        else
3981        {
3982            if (sk==NULL)
3983            {
3984                tcp_reset(daddr, saddr, th, &tcp_prot, opt,dev,skb->ip_hdr->tos,255);
3985                skb->sk = NULL;
3986                kfree_skb(skb, FREE_READ);
3987                return(0);
3988            }
3989        }
```

这个 else 语句块对应 redo=1 的情况，完成的工作只是对套接字存在性进行检查，如果在缓存过程中套接字被复位或者被关闭，那么就不需要对该数据包进行接收，对于关闭的情况，即表示本地不提供相关服务，此时回送一个 RST 数据包，复位对方请求，防止其一再进行数据包的发送。

```
3990        if (!sk->prot)
3991        {
3992            printk("IMPOSSIBLE 3\n");
3993            return(0);
3994        }
```

sock 结构 prot 字段是一个 proto 类型结构变量，表示所使用的传输层协议处理函数集合，在创建一个套接字时，会根据所使用流类型进行该字段的相应初始化，如对于 SOCK_STREAM 字段，prot 字段将被初始化为 tcp_prot, 这个 tcp_prot 变量定义在 tcp.c 文件尾部，我们在分析完 tcp_rcv 函数后，最后查看该变量的定义。实际上，sock 结构中 prot 字段的初始化应根据 socket 系统调用中的第三个参数值，但一般我们在指定前两个参数后，第三个参数我们不指定，即由于系统接口的局限性，前两个参数已经决定第三个参数的意义，如第一，二参数为 AF_INET, SOCK_STREAM，系统即认为使用 TCP 协议。因为对使用流式传输的协议，实际实现上只有 TCP 协议可供选择。

```
3995        /*
3996         *  Charge the memory to the socket.
3997         */

3998        if (sk->rmem_alloc + skb->mem_len >= sk->rcvbuf)
3999        {
4000            kfree_skb(skb, FREE_READ);
```

```
4001                release_sock(sk);
4002                return(0);
4003        }
```

3998 行代码对接收缓冲区空余空间进行检查，以查看剩余空间是否足够接收当前数据包，如果剩余空间过小，则简单丢弃该数据包返回。权当没“看见”，这将造成远端超时重发。当然这正是本地想要的，因为或许之后会有足够空闲的缓冲区接收重发的数据包。

```
4004        skb->sk=sk;
4005        sk->rmem_alloc += skb->mem_len;
```

如果接收缓冲区空闲空间足够接收该数据包，则更新空闲缓冲区值：从当前空闲值中减去数据包数据长度值。
一旦更新接收缓冲区值，我们即接收了该数据包，下面需要对该数据包的意图进行判断，根据本地套接字当前的状态进行具体的处理。

```
4006    /*
4007     *This basically follows the flow suggested by RFC793, with the corrections in RFC1122.We
4008    *don't implement precedence and we process URG incorrectly (deliberately so) for BSD bug
4009     *compatibility. We also set up variables more thoroughly [Karn notes in the
4010     *KA9Q code the RFC793 incoming segment rules don't initialise the variables for all paths].
4011      */
            //这个 if 语句块管到 4173 行。
4012        if(sk->state!=TCP_ESTABLISHED)      /* Skip this lot for normal flow */
4013        {

4014            /*
4015             *    Now deal with unusual cases.
4016             */
```

套接字状态不是 TCP_ESTABLISHED，那么按双方协调的方式，此时数据包就是一个请求数据包（而非数据传送数据包），下面将以此假设为前提，对本地对应的可用于表示或者提供请求的套接字状态进行检查。

```
                //这是一个侦听套接字，该 if 语句块管到 4047 行。
4017            if(sk->state==TCP_LISTEN)
4018            {

4019                if(th->ack)/* These use the socket TOS.. might want to be the received TOS */
4020                    tcp_reset(daddr,saddr,th,sk->prot,opt,dev,sk->ip_tos, sk->ip_ttl);
```

对于侦听套接字，其只响应连接请求（SYN 数据包），其他所有的数据包类型其都不负责，如果这个数据包是一个 ACK 应答数据包，则表示这个数据包发错了地方，此时回复一个 RST 数据包，复位对方的连接，阻止其一再发送无用的数据包给本地，浪费双方资源。

```
4021            /*
4022             *  We don't care for RST, and non SYN are absorbed (old segments)
4023             *  Broadcast/multicast SYN isn't allowed. Note - bug if you change the
4024             *  netmask on a running connection it can go broadcast. Even Sun's have
4025             *  this problem so I'm ignoring it
4026             */

4027            if(th->rst || !th->syn || th->ack || ip_chk_addr(daddr)!=IS_MYADDR)
4028            {
4029                kfree_skb(skb, FREE_READ);
4030                release_sock(sk);
4031                return 0;
4032            }
```

4027 行语句块进行检查所依据的思想同 4019 行代码：侦听套接字只负责连接请求（请求对象当然是本地），如果其一不满足，数据包即被简单丢弃。注意对于 RST，以及请求对象非本地的情况，无需回复 RST 数据包，因为此时远端不会无休止的向本地发送无用数据包，浪费资源。

```
4033            /*
4034             *  Guess we need to make a new socket up
4035             */

4036            tcp_conn_request(sk, skb, daddr, saddr, opt, dev, tcp_init_seq());
```

经过的以上的检查，我们可以认定这是一个 SYN 数据包（注意 4027 行代码中对 SYN 标志位的检查），调用 tcp_conn_request 函数对连接请求做出响应。tcp_conn_request 函数在前文中已有分析，这个函数主要完成新通信套接字的创建和初始化工作。一旦 tcp_conn_request 函数成功返回，那么对应这个请求数据包发送端此后进行的所有工作将由这个新创建的套接字负责，侦听套接字只是负责第一个 SYN 数据包，这一点非常重要！即对于同一个发送端发送的数据包，如还是三路握手中的第三个 ACK 数据包，进入 tcp_rcv 函数处理时，查找到的对应本地 sock 结构不再是侦听套接字对应的 sock 结构，而是这个新创建的套接字对应 sock 结构，换句话说，4049 行对套接字状态的检查中，sk 表示的套接字与 4017 行是不同的，当然他们分属于两次不同的 tcp_rcv 调用，对于这段话的含义读者请想明白，在理解了这个后，对于 TCP 协议将有更进一步的理解。

```
4037            /*
4038             *  Now we have several options: In theory there is nothing else
4039             *  in the frame. KA9Q has an option to send data with the syn,
4040             *  BSD accepts data with the syn up to the [to be] advertised window
4041             *  and Solaris 2.1 gives you a protocol error. For now we just ignore
4042             *  it, that fits the spec precisely and avoids incompatibilities. It
4043             *  would be nice in future to drop through and process the data.
4044             */
```

```
4045                    release_sock(sk);
```

对 release_sock 函数的调用是处理 back_log 队列中先前被缓存的其他连接请求，并做出响应。

```
4046                    return 0;
4047                }//TCP_LISTEN

4048                /* retransmitted SYN? */
4049                if (sk->state == TCP_SYN_RECV && th->syn && th->seq+1 == sk->acked_seq)
4050                {
4051                    kfree_skb(skb, FREE_READ);
4052                    release_sock(sk);
4053                    return 0;
4054                }// TCP_SYN_RECV
```

在 tcp_conn_request 函数中，在创建一个新的专门用于此后数据交换的套接字后，该套接字状态被设置为 TCP_SYN_RECV,并且该套接字以 TCP 套接字四元素（双方 IP 地址，双方端口号）为计算条件通过 put_sock 插入到系统相关队列中，下一次在此处理从同一远端接收的数据包时，查找到的 sock 结构（get_sock 函数）将是这个新创建的套接字对应 sock 结构，而不再是侦听套接字了，或者说，侦听套接字与 tcp_conn_request 是绑定的，一旦 tcp_conn_request 函数被调用，侦听套接字即脱离干系。
4049 行代码对重复发送的 SYN 数据包进行了检查，对于重发的 SYN 数据包，处理方式为简单丢弃。一般远端重发 SYN 数据包，应该是因为本地发送的三路握手中的第二个数据包丢失了，但这不会引起问题，因为本地也会超时重发。

```
4055                /*
4056                 *  SYN sent means we have to look for a suitable ack and either reset
4057                 *  for bad matches or go to connected
4058                 */
                    //这个 if 语句块管到 4138 行，是对 TCP_SYN_SENT 状态进行的相应处理。
4059                if(sk->state==TCP_SYN_SENT)
4060                {
```

对 TCP_SYN_SENT 状态的处理，对于处于这一状态的套接字是作为主动连接端（一般为客户端），在发送一个 SYN 请求连接数据包后，即设置进入 TCP_SYN_SENT 状态，此时等待远端发送一个 SYN+ACK 数据包（意为 TCP 首部中 SYN，ACK 都被设置为 1，这个三路握手中一般有服务器端发送的第二个数据包）。最主要是对本地 SYN 的应答，所以首先对所接收数据包中 ACK 标志位进行检查，由此下面代码被分为两个模块，分别对应 if，else 语句。else 语句块对可能的同时连接的情况进行处理。

```
4061                    /* Crossed SYN or previous junk segment */
4062                    if(th->ack)
4063                    {
```

4062 行 if 语句对应 ACK 标志位被设置的情况，这种情况还需要对 SYN 标志位进行检查。如果 SYN 标志位没有被设置，还是无法完成三路握手连接过程。

```
4064                    /* We got an ack, but it's not a good ack */
4065                    if(!tcp_ack(sk,th,saddr,len))
4066                    {
```

调用 tcp_ack 函数对 ACK 标志位被设置的情况进行处理，注意如果一切正常，本地套接字状态在 tcp_ack 函数中将被设置为 TCP_ESTABLISHED。 tcp_ack 函数返回 1 表示正常，返回 0 表示出现错误，对于错误的情况，本地简单丢弃该数据包并回复一个 RST 数据包（RST 数据包的主要作用是复位对方连接，阻止其发送其他数据包给本地，本书下面讲到回复 RST 数据包时都是这个意思，故不再做出解释）。

```
4067                        /* Reset the ack - its an ack from a
4068                           different connection    [ th->rst is checked in tcp_reset()] */
4069                        tcp_statistics.TcpAttemptFails++;
4070                        tcp_reset(daddr, saddr, th,
4071                              sk->prot, opt,dev,sk->ip_tos,sk->ip_ttl);
4072                        kfree_skb(skb, FREE_READ);
4073                        release_sock(sk);
4074                        return(0);
4075                    }

4076                    if(th->rst)
4077                        return tcp_std_reset(sk,skb);
```

对方给了一个数据包，表示之前发送的 SYN 请求数据包所表示的服务对方不提供，所以回复一个 RST 数据包，告知本地不要在发送 SYN 数据包了，即复位连接，本地调用 tcp_std_reset 函数进行处理。tcp_std_reset 函数主要是关闭本地套接字。

```
4078                    if(!th->syn)
4079                    {
4080                        /* A valid ack from a different connection
4081                           start. Shouldn't happen but cover it */
4082                        kfree_skb(skb, FREE_READ);
4083                        release_sock(sk);
4084                        return 0;
4085                    }
```

应答数据包中 SYN 标志位没有设置，无法完成连接过程，处理方式为简单丢弃，等待合适的数据包。注意此时不能回复一个 RST 数据包，因为对于这种情况，本地还希望从远端接收其他数据包完成连接建立过程。对于 SYN 标志位被设置的情况，表示对于本地而言，连接已经完成，下面更新相关字段值。

```
4086                    /*
4087                     *  Ok.. it's good. Set up sequence numbers and
4088                     *  move to established.
4089                     */
4090                    syn_ok=1;       /* Don't reset this connection for the syn */
4091                    sk->acked_seq=th->seq+1;
4092                    sk->fin_seq=th->seq;
4093                    tcp_send_ack(sk->sent_seq,sk->acked_seq,sk,th,sk->daddr);
4094                    tcp_set_state(sk, TCP_ESTABLISHED);
4095                    tcp_options(sk,th);
4096                    sk->dummy_th.dest=th->source;
4097                    sk->copied_seq = sk->acked_seq;
4098                    if(!sk->dead)
4099                    {
4100                        sk->state_change(sk);
4101                        sock_wake_async(sk->socket, 0);
4102                    }
4103                    if(sk->max_window==0)
4104                    {
4105                        sk->max_window = 32;
4106                        sk->mss = min(sk->max_window, sk->mtu);
4107                    }
4108                }
```

首先 4091 行更新本地应答序列号，这个序列号也表示本地希望从远端接收到的下一个字节的序列号。对于 fin_seq 字段的更新较少，此处为其一，其二为接收到一个 FIN 数据包时。4093 行回复一个应答数据包，从而帮助远端完成连接过程。之后设置本地状态为 TCP_ESTABLISHED,并对数据包中 MSS 选项进行处理。对于 TCP 连接，其必须包含 MSS 选项，从而告知对方自己可接收的数据包最大长度。4096 行对远端 IP 地址进行确认更新，4097 行对 copied_seq 字段进行更细，该字段表示已经拷贝给上层应用的数据序列号，这个序列号对应拷贝的最后一个字节的序列号加 1。由于状态进入 TCP_ESTABLISHED,通知使用该套接字的进程可以进行进一步的处理的（一般是从 connect 系统调用中返回）。最后更新本地 MSS 值，用于节制此后发送的所有数据包的大小。

```
4109                else
4110                {
```

这个 else 语句块对应 ACK 标志位没有被设置的情况，此时需要对可能的同时打开连接的情况进行检查。对于同时打开连接的情况是指本地在发送 SYN 请求连接数据包后，在接收对应的应答数据包之前，接收到了一个对方发送 SYN 请求数据包，此时即进入同时打开连接的状态。所以首先代码对数据包 SYN 标志位进行检查。

```
4111                    /* See if SYN's cross. Drop if boring */
```

```
4112                    if(th->syn && !th->rst)
4113                    {
4114                        /* Crossed SYN's are fine - but talking to
4115                            yourself is right out... */
4116                        if(sk->saddr==saddr && sk->daddr==daddr &&
4117                            sk->dummy_th.source==th->source &&
4118                            sk->dummy_th.dest==th->dest)
4119                        {
4120                                tcp_statistics.TcpAttemptFails++;
4121                                return tcp_std_reset(sk,skb);
4122                        }
4123                        tcp_set_state(sk,TCP_SYN_RECV);

4124                            /*
4125                              *    FIXME:
4126                              *    Must send SYN|ACK here
4127                              */
4128                    }// if(th->syn && !th->rst)
```

4112-4128 行是对同时发起连接情况的处理，4116 行对这个 SYN 数据包的合法性进行检查，该行 if 条件语句通过检查 TCP 套接字四要素判断这个接收的数据包是否是自己发送的：即试图自己与自己通信的情况，这种情况是不允许的。通过 4116 行的检查就表示这是一个合法的 SYN 数据包，此时对应于同时打开连接的情况，4123 行将套接字状态设置为 TCP_SYN_RECV，这是对应于同时打开情况的状态转换图，正常情况下是从 TCP_SYN_SENT 状态进入 TCP_ESTABLISHED。4124-4127 行注释表示应该在这儿发送一个 ACK 数据包。

```
4129                    /* Discard junk segment */
4130                    kfree_skb(skb, FREE_READ);
4131                    release_sock(sk);
4132                    return 0;
4133                }//else
4134                /*
4135                  *    SYN_RECV with data maybe.. drop through
4136                */
4137                goto rfc_step6;
```

对于本地原状态时 TCP_SYN_SENT 的情况，在处理完 ACK 数据包或者可能的同时打开连接的情况后，直接跳转到 rfc_step6 标志符处进行执行，从下文的代码看，这跳过了其中对于数据包序列号的检查，SYN 标志位的处理，ACK 标志位的处理，以及 RST 标志位的处理，直接进行 URG 标志位，以及普通数据的处理。跳过如上这些处理，是因为在 TCP_SYN_SENT 对应状态的模块中已经对这些情况进行了处理。

```
4138            }// TCP_SYN_SENT
```

下面这段代码是对处于 2MSL 状态的服务器端套接字从同一客户端重新发起一个新的连接的情况的处理。我们先从表层上进行阐述，然后深入本质查看具体的处理方式。一般而言，对于处理 2MSL 状态的套接字（一般为服务器端），是不允许接受新的连接请求的，但套接字编程系统接口允许应用程序通过设置一个 REUSEADDR 选项，达到使处于 2MSL 状态的套接字重新接受从相同客户端发起的新的连接请求。很多教科书上都绘这么说，但其中有一个最为关键的矛盾大家都避而不谈，那就是在客户端在请求连接时，服务器端是新创建一个套接字用于通信的，侦听套接字依然处于侦听状态，其不进行任何实际数据的交换。那么现在双方关闭连接，那么实际上从服务器的角度而言，处于 2MSL 等待状态的也是先前那个新创建的用于通信的套接字，这时候侦听套接字依然处于侦听状态，而且所谓接不接受新的连接请求，只能对侦听套接字而言，对于那个新创建的用于数据交换的套接字是不存在所谓的接不接受连接请求的情况的。那么此处就涉及到一个低层 sock 数据结构的存储和查找的问题。前文中我们分析到 get_sock 函数，该函数根据套接字四元素（双方 IP 地址，双方端口号）从系统队列中寻找满足条件的 sock 结构。实际上对于侦听套接字 sock 结构只有本地 IP 地址，本地端口号进行标识，其不与任何远端 IP 地址，远端端口号进行绑定，只有在处理一个连接请求时新创建的那个套接字方才进行四元素的绑定，而且这个新创建的套接字本地 IP 地址，本地端口号大多数情况下（对于只有一个网卡的普通主机而言，IP 地址，端口号必然相同，对于有多块网卡的路由器可能 IP 地址不同）是和侦听套接字相同的。由于 sock 结构查找过程中首先是通过本地端口号进行数组元素队列的寻址（请查询本书前文中对 get_sock 函数的分析，这个函数定义在 af_inet.c 文件中），所以对于一个侦听套接字而言，其在调用 tcp_conn_request 函数处理一个连接请求时创建的新的套接字对应的 sock 结构都与该侦听套接字 sock 结构处于同一个队列中。只不过在查找具体 sock 结构时，是查找最佳匹配的 sock 结构，由于侦听套接字只绑定了本地 IP 地址，本地端口号两个元素，而负责通信的新创建的套接字绑定了本地 IP 地址，本地端口号，远端 IP 地址，远端端口号四个元素，在查找时匹配度较之侦听套接字高，所以通常在通信套接字处于非 TCP_CLOSE 状态，其 sock 结构依然处于系统队列中时，返回的是通信套接字 sock 结构，而非侦听套接字 sock 结构；那么对于新的连接请求，为何返回侦听套接字 sock 结构？这时候，侦听套接字之前创建的所有的新的套接字本地 IP 地址以及本地端口号都符合条件，原因是：其一在查找对应 sock 结构时，远端 IP 地址，远端端口号都没有匹配项；其二侦听套接字处于队列的最前端，虽然所有的由侦听套接字之前创建的新的 sock 结构都满足本地 IP 地址，本地端口号匹配，但在查找规则设置是除非找到更合适的（匹配元素更多的，这点参考 get_sock 函数实现即可一目了然），否则返回第一个满足条件的 sock 结构。另外一个查找策略是如果套接字状态为 TCP_CLOSE，则将该套接字排除在查找对象之外。综上所述，虽然在检查服务器端套接字状态为 2MSL 时使用的是通信套接字，但在接受来自同一个远端（包括使用了相同远端 IP 地址，远端端口号）的连接请求时，使用的却是侦听套接字。

```
4139            /*
4140             *  BSD has a funny hack with TIME_WAIT and fast reuse of a port. There is
4141             *  a more complex suggestion for fixing these reuse issues in RFC1644
4142             *  but not yet ready for general use. Also see RFC1379.
4143             */

4144        #define BSD_TIME_WAIT
4145        #ifdef BSD_TIME_WAIT
4146                if (sk->state == TCP_TIME_WAIT && th->syn && sk->dead &&
4147                    after(th->seq, sk->acked_seq) && !th->rst)
```

```
4148            {
```

4146 行 if 语句是对处于 2MSL 状态的套接字是否接受到一个连接请求进行判断，如果条件都满足，则表示接收到一个具有相同远端地址，远端端口号的连接请求（这一点是由 4540 行代码保证的）。在处理上是将听任原来的这个通信套接字释放，而将请求转移给侦听套接字，通过调用 tcp_conn_request 函数重新创建一个套接字用于通信。下面我们逐行分析这时如何完成的。

```
4149                long seq=sk->write_seq;
```

保存原来这个通信套接字本地发送序列号最后值，下面（4165 行）将这个序列号加上 128000 作为新创建套接字的初始序列号。

```
4150                if(sk->debug)
4151                    printk("Doing a BSD time wait\n");
4152                tcp_statistics.TcpEstabResets++;
4153                sk->rmem_alloc -= skb->mem_len;
4154                skb->sk = NULL;
4155                sk->err=ECONNRESET;
4156                tcp_set_state(sk, TCP_CLOSE);
4157                sk->shutdown = SHUTDOWN_MASK;
4158                release_sock(sk);
```

4152-4158 行代码将原来的这个通信套接字完全置于关闭状态，首先将这个新的连接请求数据包与这个通信套接字脱离干系（4153，4154 行），并设置套接字状态为完全关闭（4156，4157 行）。最后调用 release_sock 函数进行释放（注意 release_sock 函数除了对缓存与 sock 结构中 back_log 队列中数据包调用 tcp_rcv 进行重新处理外，对于处于 TCP_CLOSE 状态以及 sk->dead 设置为 1 的套接字进行释放操作（具体的通过设置一个定时器完成，定时器到期后方才进行释放）。换句话说，以上这段代码是将原先的通信套接字完全置于“毁灭”。下面对于相同远端的连接请求通过转移给侦听套接字，而侦听套接字通过调用 tcp_conn_request 函数创建一个新的通信套接字儿完成。

```
4159                sk=get_sock(&tcp_prot, th->dest, saddr, th->source, daddr);
```

当 4156 行代码将原来的处理那个相同远端的套接字被设置为 TCP_CLOSE 状态后，其就不具备被查找的资格，所以 4159 行代码调用 get_sock 函数返回的就是侦听套接字 sock 结构!即从 4159 行开始，如下的 sk 变量指向的都是侦听套接字的 sock 结构，那么当然如果服务应用程序仍然运行的话，4160 行为真，从进行 4165 行 tcp_conn_request 函数的调用处理连接请求（即又创建一个新的通信套接字进行数据交互）。当然如果服务应用程序被关闭，简单丢弃此次连接请求。如果远端再次发送连接请求，在对其下一个连接请求数据包进行处理时，本地会“毫不客气”的回复一个 RST 数据包的。

```
4160                if (sk && sk->state==TCP_LISTEN)
4161                {
4162                    sk->inuse=1;
```

```
4163                        skb->sk = sk;
4164                        sk->rmem_alloc += skb->mem_len;
4165                        tcp_conn_request(sk, skb, daddr, saddr,opt, dev,seq+128000);
4166                        release_sock(sk);
4167                        return 0;
4168                    }
4169                    kfree_skb(skb, FREE_READ);
4170                    return 0;
4171                }
4172    #endif
4173        }// if(sk->state!=TCP_ESTABLISHED)
```

通过 4146-4171 行代码的分析，读者现在应该明白通常我们所说的使用 REUSEADDR 选项的侦听套接字究竟是如何处理的（注意 REUSEADDR 选项只可被用于侦听套接字）。在处理上，原来的通信套接字仍然进行释放，只不过侦听套接字又创建了一个新的套接字用于同一远端的通信。本质上，这与平常的处理连接请求的方式并无区别，仅仅在于现在这个请求发生在原来的套接字还没有被释放（那么在处理上就加速了释放过程，从而为创建新的套接字扫清道路），而且原来通信中可能被延迟的数据包会被发送到这个新创建的连接通道中。当然对于使用 REUSEADDR 选项的应用而言，如果真的发生这种情况，这也是自找的。对于以上代码的分析，读者需要结合 get_sock 函数进行理解。

代码执行到此处，即表示可以对数据包中可能的数据进行处理了，但从上文代码来看，只在对 TCP_SYN_SENT 状态的处理模块中处理了各标志位，对于其他状态的处理都没有涉及，所以在处理完 TCP_SYN_SENT 状态后，模块直接跳转到 rfc_step6 标志符处，而对于其他模块则进行下面代码的执行，完成对各标志位的处理。由于被调用的相关函数前文中都有分析，而对各标志位的处理方式前文中多有论述，故这段代码就由读者结合代码中注释自行分析理解，此处不再阐述。

```
4174        /*
4175         *  We are now in normal data flow (see the step list in the RFC)
4176         *  Note most of these are inline now. I'll inline the lot when
4177         *  I have time to test it hard and look at what gcc outputs
4178         */
            //对数据包中数据序列号进行合法性检查。
4179        if(!tcp_sequence(sk,th,len,opt,saddr,dev))
4180        {
4181            kfree_skb(skb, FREE_READ);
4182            release_sock(sk);
4183            return 0;
4184        }

            //这是一个 RST 数据包，调用 tcp_std_reset 函数进行 RST 标志位处理。
4185        if(th->rst)
```

```
4186                return tcp_std_reset(sk,skb);

4187        /*
4188         *    !syn_ok is effectively the state test in RFC793.
4189         */

            //对于不在相应状态（TCP_LISTEN）的套接字发送 SYN 请求连接数据包
4190        if(th->syn && !syn_ok)
4191        {
4192            tcp_reset(daddr,saddr,th, &tcp_prot, opt, dev, skb->ip_hdr->tos, 255);
4193            return tcp_std_reset(sk,skb);
4194        }

4195        /*
4196         *    Process the ACK
4197         */

            //这是一个应答数据包，但应答序列号不合法。
4198        if(th->ack && !tcp_ack(sk,th,saddr,len))
4199        {
4200            /*
4201             *    Our three way handshake failed.
4202             */
                //如果正处于三路握手连接建立过程，则连接建立失败。
                //并发送一个 RST 复位数据包，阻止其继续发送不合法的数据包，浪费资源。
4203            if(sk->state==TCP_SYN_RECV)
4204            {
4205                tcp_reset(daddr, saddr, th,sk->prot, opt, dev,sk->ip_tos,sk->ip_ttl);
4206            }
4207            kfree_skb(skb, FREE_READ);
4208            release_sock(sk);
4209            return 0;
4210        }
```

如下 tcp_rcv 函数的最后这段代码才进行数据包中可能的数据处理，首先调用 tcp_urg 函数进行紧急数据处理。此后调用 tcp_data 函数进行普通数据处理，注意如果处理过程中发生错误，则表示数据包格式或者相关条件不满足接收，此时对数据包进行简单丢弃。至于对 release_sock 函数的一再调用是为了处理之前由于套接字忙而被 tcp_rcv 函数缓存于对应 sock 结构中 back_log 队列中的数据包（重新递交给 tcp_rcv 函数处理）。

```
4211  rfc_step6:        /* I'll clean this up later */

4212        /*
4213         *    Process urgent data
```

```
4214            */

4215            if(tcp_urg(sk, th, saddr, len))
4216            {
4217                    kfree_skb(skb, FREE_READ);
4218                    release_sock(sk);
4219                    return 0;
4220            }


4221            /*
4222             *      Process the encapsulated data
4223             */

4224            if(tcp_data(skb,sk, saddr, len))
4225            {
4226                    kfree_skb(skb, FREE_READ);
4227                    release_sock(sk);
4228                    return 0;
4229            }

4230            /*
4231             *      And done
4232             */

4233            release_sock(sk);
4234            return 0;
4235    }
```

tcp_rcv 函数是 TCP 协议数据包处理的总中心，这个函数的作用有些类似于有关驱动程序中中断处理函数的作用：检查中断产生的原因，调用对应函数进行具体的处理。此处是根据数据包设置的标志为以及套接字当前的状态调用对应的其他函数完成针对性的具体处理。
虽然在分析时，采用的是从头到尾按函数定义顺序分析方式，不过在案这种方式分析完所有函数之后，我们需要进行总体考虑，以 tcp_rcv 函数入手，理解代码对 TCP 协议的具体实现，这对加深理解很有好处。所谓变换一个角度去理解问题，会很有收获。

文件最后是对 TCP 协议操作函数集合 proto 结构的定义，变量为 tcp_prot. 每个套接字在创建时，系统都创建该套接字的对应的一个 socket 结构，一个 sock 结构。当应用程序进行套接字相关调用，如 read 系统调用，调用将通过 socket 结构和 sock 结构以函数指针的方式逐层向下传递。如 read 函数调用首先通过系统接口层进入 sys_socketcall 总入口函数，该函数进一步将请求传递给 sock_read 函数处理，sock_read 函数根据 socket 结构中 ops 字段指向的处理函数集中 read 函数指针指向的函数调用 inet_read 函数，而 inet_read 函数根据 sock 结构中 prot 指向的传输层协议操作函数集中同样 read 函数指针指向的 tcp_read 完成数据读取工作，在这过程中用户缓冲区将一直作为参数被传递，从而 tcp_read 函数可以将从接收队列中读取的数据直接复制到用户缓冲

区中，最终完成应用程序数据读取工作。

对于如下 tcp_prot 变量的定义，读者可结合 proto 结构的定义来看各字段的赋值情况。proto 结构类型定义在 net/inet/sock.h 文件中。

```
4378    struct proto tcp_prot = {
4379            sock_wmalloc,
4380            sock_rmalloc,
4381            sock_wfree,
4382            sock_rfree,
4383            sock_rspace,
4384            sock_wspace,
4385            tcp_close,
4386            tcp_read,
4387            tcp_write,
4388            tcp_sendto,
4389            tcp_recvfrom,
4390            ip_build_header,
4391            tcp_connect,
4392            tcp_accept,
4393            ip_queue_xmit,
4394            tcp_retransmit,
4395            tcp_write_wakeup,
4396            tcp_read_wakeup,
4397            tcp_rcv,
4398            tcp_select,
4399            tcp_ioctl,
4400            NULL,
4401            tcp_shutdown,
4402            tcp_setsockopt,
4403            tcp_getsockopt,
4404            128,
4405            0,
4406            {NULL,},
4407            "TCP",
4408            0, 0
4409    };
```

tcp.c 文件小结

TCP 协议实现代码是网络栈实现中最为复杂的部分，这主要是由于 TCP 协议本身的复杂性造成的，其所要求的可靠性数据传输保证以及流式传输方式使得实现上必须进行数据重传以及重新排序的处理措施。基于对每个数据进行编号的方式有效的解决了这个问题，但由于网络传输通道的不稳定性，使得在代码实现上必须考虑某些极端情况，从而又有对拥塞处理，快速恢复机制的支持。另外由于提供面向连接的传输方式，必须防止一方无故中断连接，而另一端不知的

情况发生，以及在采用窗口方式对数据传输速度进行节制时由于非 0 窗口通报数据包丢失的情况发生，从而造成双方死锁，在实现上还必须采用主动探测的方式，而且又不可频繁探测，所以必须设置定时器进行辅助。总之，TCP 协议实现要考虑的方面很多，在分析实现代码时，有的地方也很难说清楚，但只要抓住 TCP 协议的核心：对数据进行编号和应答机制对很多问题都容易理解。最后一点需要提及的是，无论何种协议都是为了主机进程之间的数据交换，网络特别于同一台主机的区别在于其传输通道不可靠，所以需要定义一系列措施保证可靠性，仅此而言，而 TCP 协议专门就是为了保证这种可靠性而设计的。当然如果不要求可靠性，那么就直接传着吧，权当是数据都成功到达对方了。

2.5 net/inet/tcp.h 文件

在 inclue/linux/tcp.h 文件中定义的 TCP 协议基本信息，如 TCP 首部格式，而 net/inet/tcp.h 文件是与 net/inet/tcp.c 文件中定义的函数进行声明，以及一些常量的定义。

```
1   /*
2    * INET        An implementation of the TCP/IP protocol suite for the LINUX
3    *        operating system.    INET is implemented using the    BSD Socket
4    *        interface as the means of communication with the user level.
5    *
6    *        Definitions for the TCP module.
7    *
8    * Version:    @(#)tcp.h       1.0.5       05/23/93
9    *
10   * Authors:    Ross Biro, <bir7@leland.Stanford.Edu>
11   *        Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
12   *
13   *        This program is free software; you can redistribute it and/or
14   *        modify it under the terms of the GNU General Public License
15   *        as published by the Free Software Foundation; either version
16   *        2 of the License, or (at your option) any later version.
17   */
18  #ifndef _TCP_H
19  #define _TCP_H

20  #include <linux/tcp.h>

21  #define MAX_SYN_SIZE    44 + MAX_HEADER
22  #define MAX_FIN_SIZE    40 + MAX_HEADER
23  #define MAX_ACK_SIZE    40 + MAX_HEADER
24  #define MAX_RESET_SIZE 40 + MAX_HEADER
```

MAX_HEADER 常量定义在 include/linux/netdevice.h 文件中，值为 18。包括 MAC 首部 14 个字节以及尾部冗余校验序列 4 个字节。21-24 分别定义了 SYN，FIN，ACK，RST 数据包的长度。

其中 40 表示 20 字节 IP 首部加上 20 字节 TCP 首部。至于 SYN 数据包多出的 4 个字节用于 MSS 选项。对于 TCP 协议而言，在建立 TCP 连接时，必须使用 MSS 选项，声明本地可接受的最大报文长度。

```
25  #define MAX_WINDOW      16384
26  #define MIN_WINDOW      2048
27  #define MAX_ACK_BACKLOG  2
```

MAX_WINDOW，MIN_WINDOW 表示本地最大，最小窗口。这两个字段主要是用在对接收缓冲区当前大小的判断上（sock_rspace-sock.c）。我们在上文中一再提到，本地窗口值本质上表示本地接收缓冲区大小。在使用 TCP 协议发送数据包时，TCP 首部中窗口字段表示的是本地接收缓冲区中空闲区大小。MAX_ACK_BACKLOG 表示侦听套接字最大可接受的连接数（实际上该常量没有被使用）。

```
28  #define MIN_WRITE_SPACE     2048
29  #define TCP_WINDOW_DIFF     2048
```

MIN_WRITE_SPACE 常量没有被使用，含义不明。TCP_WINDOW_DIFF 常量表示一个差值，在 tcp_conn_request 函数中对新创建的套接字 sock 结构 max_unacked 字段进行初始化时被使用到，如下：

```
//inet/tcp.c
2402            newsk->max_unacked = MAX_WINDOW - TCP_WINDOW_DIFF;
```

max_unacked 字段表示已成功接收但尚未对其进行应答的数据总量；这儿对于 MAX_WINDOW 常量的使用有些混乱，从此处赋值的含义来看，此时 MAX_WINDOW 表示的应是远端的最大窗口值，TCP_WINDOW_DIFF 表示窗口中剩余空间大小。读者只要明白意思即可，对于这些早期代码不可过于苛求。

```
30  /* urg_data states */
31  #define URG_VALID   0x0100
32  #define URG_NOTYET  0x0200
33  #define URG_READ    0x0400
```

31-33 行这三个常量用于表示紧急数据处理的状态。URG_VALID 表示 sock 结构中 urg_data 字段表示一个有效的紧急数据字节；URG_NOTYET 表示检测到紧急数据包，sock 结构中 urg_seq 表示该紧急数据序列号，但该紧急数据尚未被复制到 urg_data 字段中；而 URG_READ 则表示 urg_data 中紧急数据已被上层读取。读者可查看 tcp_read_urg, tcp_urg 函数对这三个字段进行理解。

```
34  #define TCP_RETR1   7   /*
35                      * This is how many retries it does before it
36                      * tries to figure out if the gateway is
37                      * down.
38                      */
```

```
39  #define TCP_RETR2     15    /*
40                              * This should take at least
41                              * 90 minutes to time out.
42                              */
```

TCP_RETR1, TCP_RETR2 常量表示在丢弃本地一个数据包之前，尝试发送的次数。如果重传次数超过 TCP_RETR1，则重新进行远端主机 ARP 请求，监测是否远端主机 MAC 地址发生变换；如果重传次数超出 TCP_RETR2，则直接丢弃发送的数据包，放弃发送。

```
43  #define TCP_TIMEOUT_LEN      (15*60*HZ) /* should be about 15 mins              */
44  #define TCP_TIMEWAIT_LEN (60*HZ) /* how long to wait to successfully
45                                * close the socket, about 60 seconds  */
46  #define TCP_FIN_TIMEOUT (3*60*HZ) /* BSD style FIN_WAIT2 deadlock breaker */

47  #define TCP_ACK_TIME     (3*HZ)    /* time to delay before sending an ACK*/
48  #define TCP_DONE_TIME   250  /* maximum time to wait before actually
49                              * destroying a socket                 */
50  #define TCP_WRITE_TIME 3000       /* initial time to wait for an ACK,
51                                   * after last transmit                  */
52  #define TCP_TIMEOUT_INIT (3*HZ) /* RFC 1122 initial timeout value  */
53  #define TCP_SYN_RETRIES      5     /* number of times to retry opening a
54                              * connection                          */
55  #define TCP_PROBEWAIT_LEN 100  /* time to wait between probes when
56                              * I've got something to write and
57                              * there is no window                  */
```

43-57 行是对 TCP 协议使用的定时器相关定时间隔时间的定义。

```
58  #define TCP_NO_CHECK    0     /* turn to one if you want the default
59                              * to be no checksum                   */
```

对于 TCP 协议而言，TCP 校验是必须的。UDP，ICMP 协议校验是可选的。

```
60  /*
61   *    TCP option
62   */

63  #define TCPOPT_NOP        1     /* Padding */
64  #define TCPOPT_EOL        0     /* End of options */
65  #define TCPOPT_MSS        2     /* Segment size negotiating */
```

63-65 行是 TCP 协议最初规范 RFC-793 中定义的 TCP 使用的三种选项，其中 TCPOPT_MSS 选项必须使用在连接建立过程中且也只可使用在该过程中。

```
66  /*
67   *    We don't use these yet, but they are for PAWS and big windows
68   */
69  #define TCPOPT_WINDOW       3     /* Window scaling */
70  #define TCPOPT_TIMESTAMP    8     /* Better RTT estimations/PAWS */
```

这是后来新添加的 TCP 选项。本版本网络代码没有对其进行实现。

```
71  /*
72   * The next routines deal with comparing 32 bit unsigned ints
73   * and worry about wraparound (automatic with unsigned arithmetic).
74   */

75  extern __inline int before(unsigned long seq1, unsigned long seq2)
76  {
77              return (long)(seq1-seq2) < 0;
78  }

79  extern __inline int after(unsigned long seq1, unsigned long seq2)
80  {
81          return (long)(seq1-seq2) > 0;
82  }



83  /* is s2<=s1<=s3 ? */
84  extern __inline int between(unsigned long seq1, unsigned long seq2, unsigned long seq3)
85  {
86          return (after(seq1+1, seq2) && before(seq1, seq3+1));
87  }

88  /*
89   * List all states of a TCP socket that can be viewed as a "connected"
90   * state.    This now includes TCP_SYN_RECV, although I am not yet fully
91   * convinced that this is the solution for the 'getpeername(2)'
92   * problem. Thanks to Stephen A. Wood <saw@cebaf.gov>    -FvK
93   */
94  extern __inline const int
95  tcp_connected(const int state)
96  {
97     return(state == TCP_ESTABLISHED || state == TCP_CLOSE_WAIT ||
98            state == TCP_FIN_WAIT1      || state == TCP_FIN_WAIT2 ||
99            state == TCP_SYN_RECV);
100 }
```

tcp_connected 函数用于检测连接是否被断开，监测的依据是只要有一方可以发送数据，连接就不算是断开的，即双方还处于连接。当然此处是以通常情况进行推理，不计其他特殊状况，如一方突然崩溃。

```
//对 tcp_prot 全局变量的声明。
101  extern struct proto tcp_prot;


102  extern void     tcp_err(int err, unsigned char *header, unsigned long daddr,
103                      unsigned long saddr, struct inet_protocol *protocol);
104  extern void     tcp_shutdown (struct sock *sk, int how);
105  extern int tcp_rcv(struct sk_buff *skb, struct device *dev,
106                      struct options *opt, unsigned long daddr,
107                      unsigned short len, unsigned long saddr, int redo,
108                      struct inet_protocol *protocol);

109  extern int tcp_ioctl(struct sock *sk, int cmd, unsigned long arg);

110  extern int tcp_select_window(struct sock *sk);
111  extern void tcp_send_check(struct tcphdr *th, unsigned long saddr,
112                 unsigned long daddr, int len, struct sock *sk);
113  extern void tcp_send_probe0(struct sock *sk);
114  extern void tcp_enqueue_partial(struct sk_buff *, struct sock *);
115  extern struct sk_buff * tcp_dequeue_partial(struct sock *);

116  #endif     /* _TCP_H */
```

102-115 行是对一些函数的声明。

2.6 net/inet/udp.c 文件

udp.c 文件是 UDP 协议的实现文件，相比较 TCP 协议而言，这个实现要简单多了。RFC768 是 UDP 协议的正式描述。UDP 协议提供不可靠的数据传送方式，如同 IP 协议一样，其尽量传输，但不保证远端一定会接收到传输的数据。需要数据可靠到达的应用程序可以使用 TCP 协议，或者自行提供某种可靠性实现。UDP 协议实现所完成的工作较少，简单的说就是将用户数据从用户缓冲区复制到内核缓冲区并进行封装，之后发往下层 IP 协议进行处理。udp.c 文件定义了 UDP 协议操作函数集，并类似 TCP 协议，定义了一个 udp_prot 全局变量，使用 UDP 的套接字在创建时其对应 sock 结构 prot 字段将被初始化为 udp_prot 变量（参考 inet_create 函数实现）。下面我们对 UDP 协议实现文件 udp.c 进行分析。

```
1     /*
2      * INET        An implementation of the TCP/IP protocol suite for the LINUX
3      *        operating system.    INET is implemented using the    BSD Socket
4      *        interface as the means of communication with the user level.
5      *
6      *        The User Datagram Protocol (UDP).
7      *
8      * Version:    @(#)udp.c      1.0.13      06/02/93
9      *
10     * Authors:    Ross Biro, <bir7@leland.Stanford.Edu>
11     *        Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
12     *
13     * Fixes:
14     *        Alan Cox:       verify_area() calls
15     *        Alan Cox:       stopped close while in use off icmp
16     *                        messages. Not a fix but a botch that
17     *                        for udp at least is 'valid'.
18     *        Alan Cox:       Fixed icmp handling properly
19     *        Alan Cox:       Correct error for oversized datagrams
20     *        Alan Cox:       Tidied select() semantics.
21     *        Alan Cox:       udp_err() fixed properly, also now
22     *                        select and read wake correctly on errors
23     *        Alan Cox:       udp_send verify_area moved to avoid mem leak
24     *        Alan Cox:       UDP can count its memory
25     *        Alan Cox:       send to an unknown connection causes
26     *                        an ECONNREFUSED off the icmp, but
27     *                        does NOT close.
28     *        Alan Cox:       Switched to new sk_buff handlers. No more backlog!
29     *        Alan Cox:       Using generic datagram code. Even smaller and the PEEK
30     *                        bug no longer crashes it.
31     *        Fred Van Kempen    :       Net2e support for sk->broadcast.
32     *        Alan Cox:       Uses skb_free_datagram
33     *        Alan Cox:       Added get/set sockopt support.
34     *        Alan Cox:       Broadcasting without option set returns EACCES.
```

```
35    *        Alan Cox:    No wakeup calls. Instead we now use the callbacks.
36    *        Alan Cox:    Use ip_tos and ip_ttl
37    *        Alan Cox:    SNMP Mibs
38    *        Alan Cox:    MSG_DONTROUTE, and 0.0.0.0 support.
39    *        Matt Dillon  :    UDP length checks.
40    *        Alan Cox:    Smarter af_inet used properly.
41    *        Alan Cox:    Use new kernel side addressing.
42    *        Alan Cox:    Incorrect return on truncated datagram receive.
43    *
44    *
45    *        This program is free software; you can redistribute it and/or
46    *        modify it under the terms of the GNU General Public License
47    *        as published by the Free Software Foundation; either version
48    *        2 of the License, or (at your option) any later version.
49    */

50    #include <asm/system.h>
51    #include <asm/segment.h>
52    #include <linux/types.h>
53    #include <linux/sched.h>
54    #include <linux/fcntl.h>
55    #include <linux/socket.h>
56    #include <linux/sockios.h>
57    #include <linux/in.h>
58    #include <linux/errno.h>
59    #include <linux/timer.h>
60    #include <linux/termios.h>
61    #include <linux/mm.h>
62    #include <linux/config.h>
63    #include <linux/inet.h>
64    #include <linux/netdevice.h>
65    #include "snmp.h"
66    #include "ip.h"
67    #include "protocol.h"
68    #include "tcp.h"
69    #include <linux/skbuff.h>
70    #include "sock.h"
71    #include "udp.h"
72    #include "icmp.h"
73    #include "route.h"

74    /*
75     *    SNMP MIB for the UDP layer
76     */
```

```
77  struct udp_mib          udp_statistics;
```

udp_mib 结构类型定义在 net/inet/snmp.h 文件中，主要用于统计 UDP 各种数据，如接收或发送的数据包个数等。对于 IP，TCP，ICMP 协议都定义有类似这样一个数据结构。目前对于这个我们并不关心。读者如有兴趣，可阅读 SNMP 协议对应规范 RFC1157。

```
78  static int udp_deliver(struct sock *sk, struct udphdr *uh, struct sk_buff *skb, struct device *dev, long saddr, long daddr, int len);

79  #define min(a,b)    ((a)<(b)?(a):(b))
```

78 行是对 udp_deliver 函数的声明，79 行定义了 min 宏，取两个数中最小者。

```
80  /*
81   * This routine is called by the ICMP module when it gets some
82   * sort of error condition.  If err < 0 then the socket should
83   * be closed and the error returned to the user.  If err > 0
84   * it's just the icmp type << 8 | icmp code.
85   * Header points to the ip header of the error packet. We move
86   * on past this. Then (as it used to claim before adjustment)
87   * header points to the first 8 bytes of the udp header.  We need
88   * to find the appropriate port.
89   */

90  void udp_err(int err, unsigned char *header, unsigned long daddr,
91      unsigned long saddr, struct inet_protocol *protocol)
92  {
```

udp_err 函数的作用类似于 tcp_err, 被 ICMP 协议实现模块调用，当接收到一个错误信息时，首先 IP 协议实现模块调用 ICMP 模块进行处理，而 ICMP 模块根据使用的协议调用传输层模块处理。对于 UDP 协议，被 ICMP 模块调用的函数即为 udp_err。从 ICMP 模块调用原型中可以看出各参数的意义：
err：bit0-bit7，ICMP 首部中 code 值；bit8-bit15，ICMP 首部中 type 值。
header：引起该 ICMP 数据包产生的原数据包 IP 首部以及 8 字节的传输层数据（传输层协议首部）。
daddr：发送该错误通知数据包的远端 IP 地址。
saddr：本地 IP 地址。
protocol：该参数的意义同 tcp_rcv，tcp_err，udp_rcv。

```
93      struct udphdr *th;
94      struct sock *sk;
95      struct iphdr *ip=(struct iphdr *)header;
```

```
96          header += 4*ip->ihl;
```

header 变量指向回送的 8 字节传输层协议首部数据。有些读者之前没有接触过协议，此处需要做出一些说明，如果本地发送了一个数据包，远端或者中间路由器发生错误（如远端无对应进程活动在本地请求端口上，或者路由器没有目的端路由路径），此时远端或者中间路由器都会回送一个 ICMP 错误通知数据包给本地，该数据包首先传递给网络层 IP 协议模块处理，IP 协议模块再将该数据包传递给 ICMP 协议模块处理，ICMP 协议模块进而将该数据包传递给传输层协议模块处理，对于 UDP 协议而言，此处就是由 udp_err 函数进行处理。所谓处理，无外乎根据具体错误进行套接字状态的更新以及更新状态所需的必要措施。

```
97          /*
98           *  Find the 8 bytes of post IP header ICMP included for us
99           */

100         th = (struct udphdr *)header;

101         sk = get_sock(&udp_prot, th->source, daddr, th->dest, saddr);

102         if (sk == NULL)
103             return;     /* No socket for error */
```

将 th 变量设置为指向原数据包中 UDP 首部，根据 UDP 首部中端口号以及双方 IP 地址查找本地对应 sock 结构。如果本地没有对应的 sock 结构，则无需进一步处理，直接返回。

```
104         if (err & 0xff00 ==(ICMP_SOURCE_QUENCH << 8))
105         {   /* Slow down! */
106             if (sk->cong_window > 1)
107                 sk->cong_window = sk->cong_window/2;
108             return;
109         }
```

这是对类型码进行检查，如果类型码为源端节制，就表示远端处理速度跟不上本地发送速度，故其要求本地节制数据包的发送，本地在接收到这种 ICMP 通知数据包后，需要相应的减缓本地发送数据包的速度。以上代码通过将拥塞窗口值减半达到的，拥塞窗口表示本地发送出去但尚未得到远端应答的最大数据包个数，对于 UDP 协议而言，其不存在应答机制，所以无所谓拥塞窗口。这儿只是形式上的实现，实际上 UDP 协议实现模块在发送数据包并不检查拥塞窗口大小。正因如此，UDP 协议提供不可靠的数据传输方式。

```
110         /*
111          *  Various people wanted BSD UDP semantics. Well they've come
112          *  back out because they slow down response to stuff like dead
113          *  or unreachable name servers and they screw term users something
114          *  chronic. Oh and it violates RFC1122. So basically fix your
115          *  client code people.
```

```
116         */

117 #ifdef CONFIG_I_AM_A_BROKEN_BSD_WEENIE
118         /*
119          *  It's only fatal if we have connected to them. I'm not happy
120          *  with this code. Some BSD comparisons need doing.
121          */

122         if (icmp_err_convert[err & 0xff].fatal && sk->state == TCP_ESTABLISHED)
123         {
124                 sk->err = icmp_err_convert[err & 0xff].errno;
125                 sk->error_report(sk);
126         }
127 #else
128         if (icmp_err_convert[err & 0xff].fatal)
129         {
130                 sk->err = icmp_err_convert[err & 0xff].errno;
131                 sk->error_report(sk);
132         }
133 #endif
134 }
```

117-134 行代码根据错误的严重性进行处理，设置 sock 结构 err 字段，并通知等待该 sock 结构的进程，进程此后在操作该 sock 结构时，会被通知到这个错误的（返回 err 字段值）。sock 结构 error_report 函数指针在 inet_create(af_inet.c)函数中被初始化为 def_callback1，该函数实现如下：

```
//net/inet/af_inet.c
430 static void def_callback1(struct sock *sk)
431 {
432         if(!sk->dead)
433                 wake_up_interruptible(sk->sleep);
434 }
```

udp_check 函数用于计算 UDP 校验值，UDP 校验是可选的。函数返回一个 16bits 值，计算过程虽然使用内联编程方式，但实现思想较为简单，就是连续相加。有关内联编程，请参考本书附录 B，其中介绍了有关内联编程的内容。

```
135 static unsigned short udp_check(struct udphdr *uh, int len, unsigned long saddr, unsigned long daddr)
136 {
137         unsigned long sum;

138         __asm__(    "\t addl %%ecx,%%ebx\n"
139                     "\t adcl %%edx,%%ebx\n"
```

```
140             "\t adcl $0, %%ebx\n"
141             : "=b"(sum)
142             : "0"(daddr), "c"(saddr), "d"((ntohs(len) << 16) + IPPROTO_UDP*256)
143             : "cx","bx","dx" );

144     if (len > 3)
145     {
146         __asm__("\tclc\n"
147                 "1:\n"
148                 "\t lodsl\n"
149                 "\t adcl %%eax, %%ebx\n"
150                 "\t loop 1b\n"
151                 "\t adcl $0, %%ebx\n"
152                 : "=b"(sum) , "=S"(uh)
153                 : "0"(sum), "c"(len/4) ,"1"(uh)
154                 : "ax", "cx", "bx", "si" );
155     }

156     /*
157      *  Convert from 32 bits to 16 bits.
158      */

159     __asm__("\t movl %%ebx, %%ecx\n"
160         "\t shrl $16,%%ecx\n"
161         "\t addw %%cx, %%bx\n"
162         "\t adcw $0, %%bx\n"
163         : "=b"(sum)
164         : "0"(sum)
165         : "bx", "cx");

166     /*
167      *  Check for an extra word.
168      */

169     if ((len & 2) != 0)
170     {
171         __asm__("\t lodsw\n"
172                 "\t addw %%ax,%%bx\n"
173                 "\t adcw $0, %%bx\n"
174                 : "=b"(sum), "=S"(uh)
175                 : "0"(sum) ,"1"(uh)
176                 : "si", "ax", "bx");
177     }
```

```
178         /*
179          *    Now check for the extra byte.
180          */

181         if ((len & 1) != 0)
182         {
183                 __asm__("\t lodsb\n"
184                         "\t movb $0,%%ah\n"
185                         "\t addw %%ax,%%bx\n"
186                         "\t adcw $0, %%bx\n"
187                         : "=b"(sum)
188                         : "0"(sum) ,"S"(uh)
189                         : "si", "ax", "bx");
190         }

191         /*
192          *    We only want the bottom 16 bits, but we never cleared the top 16.
193          */

194         return((~sum) & 0xffff);
195 }
```

注意 udp_check 函数最后（194 行）对相加计算结果的取反，实际上，采用何种计算方式并不重要，重要的是大家都保持一致。另外该函数只负责对一系列连续数据进行计算，下面介绍的 udp_send_check 首先将 UDP 首部中校验值字段清零，然后调用 udp_check 函数计算校验值，并将该校验值赋值给 UDP 首部中校验值字段。

```
196 /*
197  *    Generate UDP checksums. These may be disabled, eg for fast NFS over ethernet
198  *    We default them enabled.. if you turn them off you either know what you are
199  *    doing or get burned...
200  */

201 static void udp_send_check(struct udphdr *uh, unsigned long saddr,
202                         unsigned long daddr, int len, struct sock *sk)
203 {
204         uh->check = 0;
205         if (sk && sk->no_check)
206                 return;
207         uh->check = udp_check(uh, len, saddr, daddr);

208         /*
209          *    FFFF and 0 are the same, pick the right one as 0 in the
210          *    actual field means no checksum.
```

```
211         */

212         if (uh->check == 0)
213                 uh->check = 0xffff;
214 }
```

函数 212-213 行检查计算后校验值字段是否为 0，如果为 0，则改为全 1。其实这两种情况从远端的角度来看，没有什么区别。计算校验值得目的就是远端接收到该数据包后通过重新计算校验值与本地计算的值进行比较，如果相同，就认为数据包传输过程中没有受到破坏；如果不相同，则简单丢弃。但远端在重新计算校验值时，并不是同本地一样，先将校验值字段清零，然后才计算，而是连同首部中校验值字段（注意该字段被设置为本地计算的校验值）一起计算，如果计算结果为 0，则表示数据包正常。这一点可以从数学上计算出来，那么也可以从数学上进行推导，得出将校验值字段设置为犬 0 和全 1 是同样的效果（注意在计算校验值时，双方使用的是同一种算法，更确切的说应该是同一个函数：udp_check）。

对于 TCP 协议，在发送数据时，我们使用较多的是 write 函数，因为 TCP 是面向连接的，在发送数据之前，已经建立了与远端的连接，所以此后每次发送的数据都发送到同一个远端，当然无需再指定地址。而对于 UDP 协议，在发送数据时，我们却通常使用 sendto，send 函数，其原因当然需要和 TCP 协议对照起来说明，我们读者应该明白。因为 UDP 没有连接一说，故每次发送数据时都需要指定数据发送的目的端。当然 UDP 协议也支持 connect 函数，虽然对应 UDP 协议的 connect 函数并无任何网络数据的传送，但此后的数据发送都无需继续指定远端地址，换句话说，此后，如果不明确指明远端地址的话，都表示数据是发送到之前调用 connect 时指定的地址。如下的 udp_send 函数是上层系统调用 send 函数的传输层实现。由于 UDP 协议不提供数据包可靠性传输保证，其主要完成的工作即从用户缓冲区复制数据到内核缓冲区，并对数据进行封装后，发往下层网络层模块进行处理。

```
215 static int udp_send(struct sock *sk, struct sockaddr_in *sin,
216         unsigned char *from, int len, int rt)
217 {
218         struct sk_buff *skb;
219         struct device *dev;
220         struct udphdr *uh;
221         unsigned char *buff;
222         unsigned long saddr;
223         int size, tmp;
224         int ttl;

225         /*
226          *    Allocate an sk_buff copy of the packet.
227          */

228         size = sk->prot->max_header + len;
```

计算所需要分配的封装数据的缓冲区大小。此时 sock 结构 prot 字段为 udp_prot, 对应

max_header 值为 128，这个长度足以容纳各协议首部：MAC 首部 14 字节，IP 首部 60 字节，UDP 首部 8 字节。

```
229         skb = sock_alloc_send_skb(sk, size, 0, &tmp);
```

sock_alloc_send_skb 函数定义在 net/inet/sock.c 中，用于分配指定大小的 sk_buff 结构用于封装数据。该函数还对当前套接字状态和错误标志进行检查等。在发送缓冲区暂不满足分配空间时，该函数会进入睡眠等待。

```
230         if (skb == NULL)
231                 return tmp;

232         skb->sk             = NULL;      /* to avoid changing sk->saddr */
233         skb->free           = 1;
```

当前内核版本数据重发队列的创建是在网络层进行的，换句话，虽然 TCP 协议属于传输层，但其提供的可靠性数据传输中超时重发队列是在网络层进行更新的。使用 TCP 协议的数据包将其对应 sk_buff 结构中 free 字段设置为 0，表示网络层模块在将该数据包发往下层处理之前，需要将数据包缓存到重发队列中，以便超时重发从而提供可靠性数据传输。对于 UDP 协议，其不提供可靠性数据传输，所以使用 UDP 协议的所有数据包对应的 sk_buff 结构中 free 字段设置为 1，表示网络层模块进行相应处理后，可直接发往下层，无需缓存到重发队列中，或者在下层无法接收该数据包时，直接简单丢弃。

```
234         skb->localroute = sk->localroute|(rt&MSG_DONTROUTE);
```

234 行代码设置路有查询方式，有关路有相关内容在介绍 route.c 文件中着重说明。此处按下不表。

```
235         /*
236          *    Now build the IP and MAC header.
237          */
```

既然已经成功分配内核缓冲区，现在进行数据复制，不过在这之前，首先创建 MAC 首部和 IP 首部，其实顺序先后重要性不大。

```
238         buff = skb->data;
239         saddr = sk->saddr;
240         dev = NULL;
241         ttl = sk->ip_ttl;
242  #ifdef CONFIG_IP_MULTICAST
243         if (MULTICAST(sin->sin_addr.s_addr))
244                 ttl = sk->ip_mc_ttl;
```

如果目的地址是多播，则设置 TTL 值为 1，表示局限于本地网络，不可跨越路有器。

```
245  #endif
```

```
246     tmp = sk->prot->build_header(skb, saddr, sin->sin_addr.s_addr,
247                 &dev, IPPROTO_UDP, sk->opt, skb->mem_len,sk->ip_tos,ttl);

248     skb->sk=sk;    /* So memory is freed correctly */

249     /*
250       *    Unable to put a header on the packet.
251       */

252     if (tmp < 0 )
253     {
254         sk->prot->wfree(sk, skb->mem_addr, skb->mem_len);
255         return(tmp);
256     }
```

如果 ip_build_header(246 行实际调用的函数)返回负值，表示首部创建失败，此时取消本地发送，返回错误给上层应用（注意对于 send 函数返回负值表示发送失败，否则返回值为实际发送的字节数，其他发送函数如 write 等类似）。

```
257     buff += tmp;
258     saddr = skb->saddr; /*dev->pa_addr;*/
259     skb->len = tmp + sizeof(struct udphdr) + len;      /* len + UDP + IP + MAC */
260     skb->dev = dev;
```

继续更新数据封装结构中部分字段值。

```
261     /*
262       *    Fill in the UDP header.
263       */

264     uh = (struct udphdr *) buff;
265     uh->len = htons(len + sizeof(struct udphdr));
266     uh->source = sk->dummy_th.source;
267     uh->dest = sin->sin_port;
268     buff = (unsigned char *) (uh + 1);
```

264-267 创建 UDP 首部，主要是双方端口号字段的初始化。

```
269     /*
270       *    Copy the user data.
271       */

272     memcpy_fromfs(buff, from, len);
```

memcpy_fromfs 函数复制用户缓冲区数据到内核缓冲区中。

```
273         /*
274          *  Set up the UDP checksum.
275          */
            //计算 UDP 校验值。
276         udp_send_check(uh, saddr, sin->sin_addr.s_addr, skb->len - tmp, sk);

277         /*
278          *  Send the datagram to the interface.
279          */

280         udp_statistics.UdpOutDatagrams++;
            //调用 ip_queue_xmit 函数将数据包发完网络层模块处理。
281         sk->prot->queue_xmit(sk, dev, skb, 1);
282         return(len);
283 }
```

注意 281 行实际调用的是 ip_queue_xmit 函数，对于 UDP 协议对应操作函数集 udp_prot 各函数指针指向的实际函数在 udp.c 文件结尾处给出。其中 queue_xmit 指向是 ip_queue_xmit 函数。

```
284 static int udp_sendto(struct sock *sk, unsigned char *from, int len, int noblock,
285             unsigned flags, struct sockaddr_in *usin, int addr_len)
286 {
287         struct sockaddr_in sin;
288         int tmp;

289         /*
290          *  Check the flags. We support no flags for UDP sending
291          */
292         if (flags&~MSG_DONTROUTE)
293                 return(-EINVAL);
294         /*
295          *  Get and verify the address.
296          */

297         if (usin)
298         {
299                 if (addr_len < sizeof(sin))
300                         return(-EINVAL);
301                 memcpy(&sin,usin,sizeof(sin));
302                 if (sin.sin_family && sin.sin_family != AF_INET)
303                         return(-EINVAL);
```

```
304             if (sin.sin_port == 0)
305                     return(-EINVAL);
306         }
307         else
308         {
309             if (sk->state != TCP_ESTABLISHED)
310                     return(-EINVAL);
311             sin.sin_family = AF_INET;
312             sin.sin_port = sk->dummy_th.dest;
313             sin.sin_addr.s_addr = sk->daddr;
314         }

315         /*
316          *  BSD socket semantics. You must set SO_BROADCAST to permit
317          *  broadcasting of data.
318          */

319         if(sin.sin_addr.s_addr==INADDR_ANY)
320             sin.sin_addr.s_addr=ip_my_addr();

321         if(!sk->broadcast && ip_chk_addr(sin.sin_addr.s_addr)==IS_BROADCAST)
322                 return -EACCES;                 /* Must turn broadcast on first */

323         sk->inuse = 1;

324         /* Send the packet. */
325         tmp = udp_send(sk, &sin, from, len, flags);

326         /* The datagram has been sent off.    Release the socket. */
327         release_sock(sk);
328         return(tmp);
329 }
```

udp_sendto 函数通过调用 udp_send 函数发送数据包，不过该函数在发送数据包之前对远端地址的合法性进行了检查，这种检查也只是表面，由于不涉及网络数据传送，所以无法验证这个地址存在性。首先 292 行对标志位进行检查，除了 MSG_DONTROUTE 外，UDP 协议不支持任何其他标志位，如果设置了其他标志位，则返回错误。此后 297，307 分别对应本地调用明确指定远端地址和没有指定的情况，对于明确指定的情况，则直接对给出的地址进行检查，如果没有明确执行，那么检查之前是否调用了 connect 函数进行了地址绑定，如果进行了绑定，则将远端地址设置为这个绑定的地址，否则出错返回。319 行处理尚未指定本地地址的情况；321 处理广播的情况，代码都容易理解。最后 325 行调用 udp_send 函数将用户数据发送出去。

```
330 /*
```

```
331   *    In BSD SOCK_DGRAM a write is just like a send.
332   */

333  static int udp_write(struct sock *sk, unsigned char *buff, int len, int noblock,
334            unsigned flags)
335  {
336        return(udp_sendto(sk, buff, len, noblock, flags, NULL, 0));
337  }
```

udp_write 函数直接对 udp_sendto 函数进行封装，不直接调用 udp_send 函数的目的在于 write 系统调用必须有一个默认远端地址，所以需要 udp_sendto 函数中的地址处理代码。

```
338  /*
339   *    IOCTL requests applicable to the UDP protocol
340   */

341  int udp_ioctl(struct sock *sk, int cmd, unsigned long arg)
342  {
343        int err;
344        switch(cmd)
345        {
346              case TIOCOUTQ:
347              {
348                    unsigned long amount;

349                    if (sk->state == TCP_LISTEN) return(-EINVAL);
350                    amount = sk->prot->wspace(sk)/*/2*/;
351                    err=verify_area(VERIFY_WRITE,(void *)arg,
352                                sizeof(unsigned long));
353                    if(err)
354                          return(err);
355                    put_fs_long(amount,(unsigned long *)arg);
356                    return(0);
357              }

358              case TIOCINQ:
359              {
360                    struct sk_buff *skb;
361                    unsigned long amount;

362                    if (sk->state == TCP_LISTEN) return(-EINVAL);
363                    amount = 0;
364                    skb = skb_peek(&sk->receive_queue);
365                    if (skb != NULL) {
366                          /*
```

```
367                         * We will only return the amount
368                         * of this packet since that is all
369                         * that will be read.
370                         */
371                        amount = skb->len;
372                }
373                err=verify_area(VERIFY_WRITE,(void *)arg,
374                                sizeof(unsigned long));
375                if(err)
376                        return(err);
377                put_fs_long(amount,(unsigned long *)arg);
378                return(0);
379            }

380            default:
381                return(-EINVAL);
382        }
383        return(0);
384 }
```

udp_ioctl 函数支持的选项不多，TIOCOUTQ 选项用于检测当前发送缓冲区中空闲空间大小；TIOCINQ 选项用于查询下一个可读取数据包中数据长度。注意 UDP 协议是面向报文的，不是面向流的，即对于 UDP 协议而言，其接收的每个数据包被认为是独立的，可以是从不同的远端发送的。所以在查询可读数据量时，不可如同 TCP 协议一样，对多个序列号连续的数据包进行汇总。以上代码较为浅显，读者自行分析理解。

```
385 /*
386  *    This should be easy, if there is something there we\
387  *    return it, otherwise we block.
388  */

389 int udp_recvfrom(struct sock *sk, unsigned char *to, int len,
390                int noblock, unsigned flags, struct sockaddr_in *sin,
391                int *addr_len)
392 {
393        int copied = 0;
394        int truesize;
395        struct sk_buff *skb;
396        int er;

397        /*
398         *    Check any passed addresses
399         */

400        if (addr_len)
```

```
401             *addr_len=sizeof(*sin);

402         /*
403          *  From here the generic datagram does a lot of the work. Come
404          *  the finished NET3, it will do _ALL_ the work!
405          */

406         skb=skb_recv_datagram(sk,flags,noblock,&er);
407         if(skb==NULL)
408             return er;

409         truesize = skb->len;
410         copied = min(len, truesize);

411         /*
412          *  FIXME : should use udp header size info value
413          */

414         skb_copy_datagram(skb,sizeof(struct udphdr),to,copied);
415         sk->stamp=skb->stamp;

416         /* Copy the address. */
417         if (sin)
418         {
419             sin->sin_family = AF_INET;
420             sin->sin_port = skb->h.uh->source;
421             sin->sin_addr.s_addr = skb->daddr;
422         }

423         skb_free_datagram(skb);
424         release_sock(sk);
425         return(truesize);
426 }
```

udp_recvfrom 函数是 UDP 协议数据读取函数，只有一点需要提请注意，每次读取只可从接收队列（sock 结构 receive_queue 字段指向的队列）读取单个数据包。414 行 skb_recv_datagram 函数完成的工作就是从 receive_queue 队列中取下一个数据包，如果暂时队列中无数据包的话，则睡眠等待。skb_recv_datagram 函数定义在 net/inet/datagram.c 中，除了从接收队列中取数据包外，其还完成一系列检查，如对 sock 结构 err 字段的检查，检查套接字是否发生错误，对 NON_BLOCK 标志位的检查，从而在当前无数据包可读取时，是否睡眠等待等等。如果该函数返回 NULL，则其一使用了 NON_BLOCK 选项，且当前无可读数据包；其二在睡眠等待数据包过程中，被其它条件中断。无论如何，只要返回 NULL，就表示无数据包可读，此时返回可能的错误状态值，当然如果没有发生错误的话，返回值将是 0，表示没有读取到数据。如果成功返回可以可读数据包，则对读取的数据长度进行检查，410 行取数据包可读长度和用户要求长度中的最小值，这会造成两种结果，一是用户要求长度大于实际可读

数量，则此时可以读取多少就返回多少，这是我们一般使用 UDP 协议时常见的情况（此时我们要求的长度实际上是用户缓冲区大小）；二是用户要求长度小于实际可读取量（无论是用户缓冲区分配的太小，还是用户真正只要求了这么多数据），此时多出的那部分数据将被简单丢弃，无法如同 TCP 协议，在下层读取时返回这些这次没有读完的数据，根本原因还是 UDP 协议是面向报文的，其接收到的每个数据包都是独立的，可能来自不同的远端。当然内核实现上，也可以将数据没有被读完的数据包从小回插入接收队列中，从而在应用下次读取时进行返回。这是实现相关的，就看实现支持不支持这种情况。本版本网络代码不支持这种分多次读取一个数据包的情况。414 行完成数据从内核缓冲区到用户缓冲区的复制。417 行处理返回发送数据包的远端地址的情况，如果对应参数要求返回远端地址的话。423 行完成数据包的释放。

```
427  /*
428   *    Read has the same semantics as recv in SOCK_DGRAM
429   */

430  int udp_read(struct sock *sk, unsigned char *buff, int len, int noblock,
431           unsigned flags)
432  {
433          return(udp_recvfrom(sk, buff, len, noblock, flags, NULL, NULL));
434  }
```

udp_read 函数是对 udp_recvfrom 函数的直接封装，只不过其不要求返回远端地址，因为是用 read，write 函数调用 UDP 套接字时，在这之前一定通过 connect 系统调用对远端地址进行了设置。

下面我们即看 UDP 协议 connect 函数究竟完成了哪些工作。

```
435  int udp_connect(struct sock *sk, struct sockaddr_in *usin, int addr_len)
436  {
437          struct rtable *rt;
438          unsigned long sa;
439          if (addr_len < sizeof(*usin))
440                  return(-EINVAL);

441          if (usin->sin_family && usin->sin_family != AF_INET)
442                  return(-EAFNOSUPPORT);
443          if (usin->sin_addr.s_addr==INADDR_ANY)
444                  usin->sin_addr.s_addr=ip_my_addr();

445          if(!sk->broadcast && ip_chk_addr(usin->sin_addr.s_addr)==IS_BROADCAST)
446                  return -EACCES;                 /* Must turn broadcast on first */

447          rt=ip_rt_route(usin->sin_addr.s_addr, NULL, &sa);
448          if(rt==NULL)
449                  return -ENETUNREACH;
```

```
450         sk->saddr = sa;             /* Update source address */
451         sk->daddr = usin->sin_addr.s_addr;
452         sk->dummy_th.dest = usin->sin_port;
453         sk->state = TCP_ESTABLISHED;
454         return(0);
455 }
```

udp_connect 函数完成任务如下：

1> 进行远端合法性检查，包括地址长度及地址类型（AF_INET）,以及地址可达性，即本地路由中是否包含到达该远端地址的路由项。另外对于远端地址设置为广播地址的情况也进行了检查（445 行），如果本地套接字不支持广播，则返回错误：EACCESS，无法访问。

2> 在通过地址合法性检查后，设置 sock 结构中对应远端地址各字段。

3> 最后设置本地套接字状态为 TCP_ESTABLISHED,这是 UDP 协议模拟 TCP 协议进行的状态设置，实际上 UDP 协议在完成连接建立的过程中实际没有发送一个数据包到网络介质上。

在调用 connect 函数后，此后应用程序就可以使用 read，write 函数如同 TCP 协议一样进行数据读取和发送。从而避免了每次函数调用时都要传递远端地址。

```
456 static void udp_close(struct sock *sk, int timeout)
457 {
458         sk->inuse = 1;
459         sk->state = TCP_CLOSE;
460         if (sk->dead)
461                 destroy_sock(sk);
462         else
463                 release_sock(sk);
464 }
```

如同 udp_connect 函数一样，upd_close 也不涉及实际数据包的传送。主要还是状态的设置以及本地已接收数据包的处理。如果 sock 结构 dead 字段已经设置为 1，则可以立刻进行套接字相关结构释放，否则继续处理已接收的数据包。

```
465 /*
466  *  All we need to do is get the socket, and then do a checksum.
467  */

468 int udp_rcv(struct sk_buff *skb, struct device *dev, struct options *opt,
469         unsigned long daddr, unsigned short len,
470         unsigned long saddr, int redo, struct inet_protocol *protocol)
471 {
```

udp_rcv 函数作用如同 tcp_rcv，它是 UDP 协议数据包处理总入口函数，不过他的处理方式没有 tcp_rcv 那么麻烦，要分那么多种情况。根本原因在于 UDP 协议是一个无状态协议。没有状态的转换之类的控制，所以对于下层传递上来的数据包的处理方式较为直接：直接挂接到对应 sock 结构的 receive_queue 指向的接收队列中。不过从函数实现来看，代码还是不短，原因在于要判断该不该接收这个数据包还是一件麻烦事！有关 udp_rcv 函数调用中各函数含义同 tcp_rcv 函数。

```
472     struct sock *sk;
473     struct udphdr *uh;
474     unsigned short ulen;
475     int addr_type = IS_MYADDR;

476     if(!dev || dev->pa_addr!=daddr)
477         addr_type=ip_chk_addr(daddr);
```

首先我们要检查一下这个数据包是不是发给本地的。dev 变量表示接收该数据包的网口设备。虽然 dev 变量为 NULL 不大可能发生，但有可能 daddr！=dev->pa_addr, 因为或者这是一个多播或者广播数据包。ip_chk_addr 函数的目的就是对此进行检查。dev->pa_addr 表示网口被赋予的 IP 地址，另外 dev->ha_addr 表示网口的硬件地址。

```
478     /*
479      *  Get the header.
480      */
481     uh = (struct udphdr *) skb->h.uh;

482     ip_statistics.IpInDelivers++;

483     /*
484      *  Validate the packet and the UDP length.
485      */

486     ulen = ntohs(uh->len);

487     if (ulen > len || len < sizeof(*uh) || ulen < sizeof(*uh))
488     {
489         printk("UDP: short packet: %d/%d\n", ulen, len);
490         udp_statistics.UdpInErrors++;
491         kfree_skb(skb, FREE_WRITE);
492         return(0);
493     }
```

486 行 ulen 变量被初始化为 UDP 首部长度字段值，该字段表示的长度包括 UDP 首部及其用户数据负载的长度。这个值应该小于传入的 len 参数值，参数 len 在网络层 IP 模块调用该 udp_rcv 函数被初始化为 IP 负载的长度：即 UDP 首部及其用户数据负载的长度再加上填充数据长度。因为当前使用最多的是以太网，网络上的每台主机竞争使用网络介质，为了让网

卡设备有时间探测到数据发送冲突（具体请查阅对 CSMA/CD 协议的介绍），对于发送的数据包长度有个最小限制，如果不计最后 4 字节的冗余校验的话，这个值就是 60 字节。或者我们通常所说的 46 字节（60 字节减去 14 字节 MAC 首部长度）所以应该有：ulen<=len。487 行其它判断容易理解，如果数据包被检测出不合法，则在更新相关统计信息后丢弃。

```
494         if (uh->check && udp_check(uh, len, saddr, daddr))
495         {
496                 /* <mea@utu.fi> wants to know, who sent it, to
497                    go and stomp on the garbage sender... */
498                 printk("UDP: bad checksum. From %08lX:%d to %08lX:%d ulen %d\n",
499                         ntohl(saddr),ntohs(uh->source),
500                         ntohl(daddr),ntohs(uh->dest),
501                         ulen);
502                 udp_statistics.UdpInErrors++;
503                 kfree_skb(skb, FREE_WRITE);
504                 return(0);
505         }
506         len=ulen;
```

494-505 行进行数据包校验值检查。506 行将 len 参数值赋值为实际数据长度。

```
507  #ifdef CONFIG_IP_MULTICAST
508         if (addr_type!=IS_MYADDR)
509         {
510                 /*
511                  *    Multicasts and broadcasts go to each listener.
512                  */
513                 struct sock *sknext=NULL;
514
    sk=get_sock_mcast(udp_prot.sock_array[ntohs(uh->dest)&(SOCK_ARRAY_SIZE-1)],
uh->dest,
515                         saddr, uh->source, daddr);
516                 if(sk)
517                 {
518                         do
519                         {
520                                 struct sk_buff *skb1;

521                                 sknext=get_sock_mcast(sk->next, uh->dest, saddr, uh->source, daddr);
522                                 if(sknext)
523                                         skb1=skb_clone(skb,GFP_ATOMIC);
524                                 else
525                                         skb1=skb;
526                                 if(skb1)
```

```
527                             udp_deliver(sk, uh, skb1, dev,saddr,daddr,len);
528                         sk=sknext;
529                     }
530                     while(sknext!=NULL);
531             }
532             else
533                     kfree_skb(skb, FREE_READ);
534             return 0;
535     }
536 #endif
```

507-536 行代码是对多播情况的处理。首先 508 行判断出该数据包目的地址不是本地地址，排除了单播的情况。此后 514 行调用 get_sock_mcast 函数返回可能的侦听在对应端口上的本地多播套接字队列，此后对队列进行遍历，对满足条件的所有套接字都复制一份该数据包。get_sock_mcast 函数定义在 af_inet.c 中，本书前文中已经对该函数进行了分析，此处再次列出，便于查看：

```
//net/inet/af_inet.c
1275    #ifdef CONFIG_IP_MULTICAST
1276    /*
1277     *    Deliver a datagram to broadcast/multicast sockets.
1278     */

1279    struct sock *get_sock_mcast(struct sock *sk,
1280                                unsigned short num,
1281                                unsigned long raddr,
1282                                unsigned short rnum, unsigned long laddr)
1283    {
1284            struct sock *s;
1285            unsigned short hnum;

1286            hnum = ntohs(num);

1287            /*
1288             * SOCK_ARRAY_SIZE must be a power of two.    This will work better
1289             * than a prime unless 3 or more sockets end up using the same
1290             * array entry.    This should not be a problem because most
1291             * well known sockets don't overlap that much, and for
1292             * the other ones, we can just be careful about picking our
1293             * socket number when we choose an arbitrary one.
1294             */

1295            s=sk;

1296            for(; s != NULL; s = s->next)
```

```
1297            {
1298                    if (s->num != hnum)
1299                            continue;
1300                    if(s->dead && (s->state == TCP_CLOSE))
1301                            continue;
1302                    if(s->daddr && s->daddr!=raddr)
1303                            continue;
1304                    if (s->dummy_th.dest != rnum && s->dummy_th.dest != 0)
1305                            continue;
1306                    if(s->saddr    && s->saddr!=laddr)
1307                            continue;
1308                    return(s);
1309            }
1310            return(NULL);
1311}

1312    #endif
```

注意 get_sock_mcast 函数第一个参数是一个 sock 结构，即调用该函数时必须已经找到了多播套接字 sock 结构队列，对于 udp_rcv 函数，这是在 514 行完成的。在 508 行判断出这不是一个发往本地单播地址的数据包后，直接以本地远端口（uh->dest）作为索引提出系统数组中 sock 结构队列，至于队列中存不存在多播套接字，则通过数据报的目的地址查询即可。一旦 get_sock_mcast 返回一个 sock 结构，则表示本地存在有相关的多播套接字，那么就对该数据包进行接收。注意对于广播的处理包含在对多播的处理中。数据包的具体接收是通过 udp_deliver 函数完成的。在分析完 udp_rcv 函数后，我们再看 udp_deliver 函数。

```
537     sk = get_sock(&udp_prot, uh->dest, saddr, uh->source, daddr);
538     if (sk == NULL)
539     {
540             udp_statistics.UdpNoPorts++;
541             if (addr_type == IS_MYADDR)
542             {
543                     icmp_send(skb,  ICMP_DEST_UNREACH,  ICMP_PORT_UNREACH,  0,
dev);
544             }
545             /*
546               * Hmm.    We got an UDP broadcast to a port to which we
547               * don't wanna listen.    Ignore it.
548               */
549             skb->sk = NULL;
550             kfree_skb(skb, FREE_WRITE);
551             return(0);
552     }
```

处理完可能的多播的情况，现在就要进行单播处理，如果 537 没有找到一个本地对应的套接字，则表示这个数据包的目的进程不存在或者已经死亡，此时本地将回复一个端口不可达的 ICMP 错误报文（这就是 543 行完成的工作），更新相关信息，丢弃该数据包。

```
553         return udp_deliver(sk,uh,skb,dev, saddr, daddr, len);
554  }
```

如果一切正常，那么就调用 udp_deliver 函数进行接收。该函数定义如下：

```
555  static int udp_deliver(struct sock *sk, struct udphdr *uh, struct sk_buff *skb, struct device *dev, long saddr, long daddr, int len)
556  {
557         skb->sk = sk;
558         skb->dev = dev;
559         skb->len = len;

560         /*
561          *    These are supposed to be switched.
562          */

563         skb->daddr = saddr;
564         skb->saddr = daddr;



565         /*
566          *    Charge it to the socket, dropping if the queue is full.
567          */

568         skb->len = len - sizeof(*uh);

569         if (sock_queue_rcv_skb(sk,skb)<0)
570         {
571                 udp_statistics.UdpInErrors++;
572                 ip_statistics.IpInDiscards++;
573                 ip_statistics.IpInDelivers--;
574                 skb->sk = NULL;
575                 kfree_skb(skb, FREE_WRITE);
576                 release_sock(sk);
577                 return(0);
578         }
579         udp_statistics.UdpInDatagrams++;
580         release_sock(sk);
581         return(0);
582  }
```

Udp_deliver 函数首先对数据包对应的 sk_buff 封装结构中一些字段进行初始化，之后调用 sock_queue_rcv_skb 函数将数据包挂接到 sk 变量表示的套接字的接收队列中，从而完成套接字的接收。sock_queue_rcv_skb 函数定义在 net/inet/sock.c 中，如下：

```
//net/inet/sock.c
446 /*
447  *  Queue a received datagram if it will fit. Stream and sequenced protocols
448  *  can't normally use this as they need to fit buffers in and play with them.
449  */

450 int sock_queue_rcv_skb(struct sock *sk, struct sk_buff *skb)
451 {
452     unsigned long flags;
453     if(sk->rmem_alloc + skb->mem_len >= sk->rcvbuf)
454         return -ENOMEM;
455     save_flags(flags);
456     cli();
457     sk->rmem_alloc+=skb->mem_len;
458     skb->sk=sk;
459     restore_flags(flags);
460     skb_queue_tail(&sk->receive_queue,skb);
461     if(!sk->dead)
462         sk->data_ready(sk,skb->len);
463     return 0;
464 }
```

sock_queue_rcv_skb 函数对接收缓冲区进行更新，将数据包插入接收队列尾部，最后通知可能等待读取数据被置于睡眠的进程进行数据包的读取。

如同 tcp.c 文件最后对 tcp_prot 变量的定义，udp.c 最后也是对 UDP 协议操作函数集和的这样一个定义：udp_prot。

```
583 struct proto udp_prot = {
584     sock_wmalloc,
585     sock_rmalloc,
586     sock_wfree,
587     sock_rfree,
588     sock_rspace,
589     sock_wspace,
590     udp_close,
591     udp_read,
592     udp_write,
593     udp_sendto,
594     udp_recvfrom,
595     ip_build_header,
596     udp_connect,
```

```
597         NULL,
598         ip_queue_xmit,
599         NULL,
600         NULL,
601         NULL,
602         udp_rcv,
603         datagram_select,
604         udp_ioctl,
605         NULL,
606         NULL,
607         ip_setsockopt,
608         ip_getsockopt,
609         128,
610         0,
611         {NULL,},
612         "UDP",
613         0, 0
614 };
```

2.7 net/inet/udp.h 头文件

该头文件主要是对一些函数原型的声明，故简单列出如下。

```
1   /*
2    * INET         An implementation of the TCP/IP protocol suite for the LINUX
3    *              operating system.  INET is implemented using the  BSD Socket
4    *              interface as the means of communication with the user level.
5    *
6    *              Definitions for the UDP module.
7    *
8    * Version:     @(#)udp.h       1.0.2       05/07/93
9    *
10   * Authors:     Ross Biro, <bir7@leland.Stanford.Edu>
11   *              Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
12   *
13   * Fixes:
14   *              Alan Cox : Turned on udp checksums. I don't want to
15   *                                 chase 'memory corruption' bugs that aren't!
16   *
17   *              This program is free software; you can redistribute it and/or
18   *              modify it under the terms of the GNU General Public License
19   *              as published by the Free Software Foundation; either version
20   *              2 of the License, or (at your option) any later version.
21   */
22  #ifndef _UDP_H
```

```
23  #define _UDP_H

24  #include <linux/udp.h>


25  #define UDP_NO_CHECK    0


26  extern struct proto udp_prot;


27  extern void     udp_err(int err, unsigned char *header, unsigned long daddr,
28                      unsigned long saddr, struct inet_protocol *protocol);
29  extern int udp_recvfrom(struct sock *sk, unsigned char *to,
30                              int len, int noblock, unsigned flags,
31                              struct sockaddr_in *sin, int *addr_len);
32  extern int udp_read(struct sock *sk, unsigned char *buff,
33                        int len, int noblock, unsigned flags);
34  extern int udp_connect(struct sock *sk,
35                             struct sockaddr_in *usin, int addr_len);
36  extern int udp_rcv(struct sk_buff *skb, struct device *dev,
37                      struct options *opt, unsigned long daddr,
38                      unsigned short len, unsigned long saddr, int redo,
39                      struct inet_protocol *protocol);
40  extern int udp_ioctl(struct sock *sk, int cmd, unsigned long arg);


41  #endif     /* _UDP_H */
```

在介绍完传输层两个最主要的协议之后，我们对这一层两个至关重要的数据结构尚未介绍：sock 结构（该结构实际上在本书的前面在介绍 af_inet.c 文件时已经作出了分析），proto 结构。另外还有 TCP，UDP 用来操作发送和接收缓冲区的功能函数如 sock_rspace, sock_rmalloc, sock_wmalloc 等都为介绍，这些函数或者变量定义在两个文件中：sock.c, sock.h。下面我们就对这两个文件进行分析。首先从 sock.h 头文件入手先对 sock 结构和 proto 结构进行分析。

2.8 net/inet/sock.h 头文件

对该文件的分析将主要集中在对 sock 结构和 proto 结构的分析上。在本书前文分析 af_inet.c 文件时，已经对 sock 结构中各字段的含义进行了较为详细的介绍，另外 proto 结构定义为操作函数集，其主要由函数指针组成，前文中 tcp_prot, udp_prot 全局变量都是 proto 结构类型，分别定义了 TCP 协议和 UDP 协议的操作函数集。此处为保证本书的完整性，系统的给出这两个结构的定义，但不再进行重复说明。

```
1   /*
2    * INET        An implementation of the TCP/IP protocol suite for the LINUX
3    *        operating system.  INET is implemented using the  BSD Socket
4    *        interface as the means of communication with the user level.
5    *
6    *        Definitions for the AF_INET socket handler.
7    *
8    * Version:    @(#)sock.h    1.0.4    05/13/93
9    *
10   * Authors:    Ross Biro, <bir7@leland.Stanford.Edu>
11   *        Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
12   *        Corey Minyard <wf-rch!minyard@relay.EU.net>
13   *        Florian La Roche <flla@stud.uni-sb.de>
14   *
15   * Fixes:
16   *        Alan Cox:    Volatiles in skbuff pointers. See
17   *                     skbuff comments. May be overdone,
18   *                     better to prove they can be removed
19   *                     than the reverse.
20   *        Alan Cox:    Added a zapped field for tcp to note
21   *                     a socket is reset and must stay shut up
22   *        Alan Cox:    New fields for options
23   *   Pauline Middelink  :    identd support
24   *
25   *        This program is free software; you can redistribute it and/or
26   *        modify it under the terms of the GNU General Public License
27   *        as published by the Free Software Foundation; either version
28   *        2 of the License, or (at your option) any later version.
29   */
30  #ifndef _SOCK_H
31  #define _SOCK_H

32  #include <linux/timer.h>
33  #include <linux/ip.h>          /* struct options */
34  #include <linux/tcp.h>         /* struct tcphdr */
35  #include <linux/config.h>

36  #include <linux/skbuff.h>      /* struct sk_buff */
37  #include "protocol.h"          /* struct inet_protocol */
38  #ifdef CONFIG_AX25
39  #include "ax25.h"
40  #endif
41  #ifdef CONFIG_IPX
42  #include "ipx.h"
```

```
43  #endif
44  #ifdef CONFIG_ATALK
45  #include <linux/atalk.h>
46  #endif

47  #include <linux/igmp.h>

48  #define SOCK_ARRAY_SIZE     256         /* Think big (also on some systems a byte is
faster */


49  /*
50    * This structure really needs to be cleaned up.
51    * Most of it is for TCP, and not used by any of
52    * the other protocols.
53    */
54  struct sock {
55      struct options          *opt;
56      volatile unsigned long  wmem_alloc;
57      volatile unsigned long  rmem_alloc;
58      unsigned long           write_seq;
59      unsigned long           sent_seq;
60      unsigned long           acked_seq;
61      unsigned long           copied_seq;
62      unsigned long           rcv_ack_seq;
63      unsigned long           window_seq;
64      unsigned long           fin_seq;
65      unsigned long           urg_seq;
66      unsigned long           urg_data;

67      /*
68        * Not all are volatile, but some are, so we
69        * might as well say they all are.
70        */
71      volatile char                   inuse,
72                    dead,
73                    urginline,
74                    intr,
75                    blog,
76                    done,
77                    reuse,
78                    keepopen,
79                    linger,
80                    delay_acks,
```

```
81                      destroy,
82                      ack_timed,
83                      no_check,
84                      zapped,   /* In ax25 & ipx means not linked */
85                      broadcast,
86                      nonagle;
87      unsigned long                   lingertime;
88      int                     proc;
89      struct sock             *next;
90      struct sock             *prev; /* Doubly linked chain.. */
91      struct sock             *pair;
92      struct sk_buff          * volatile send_head;
93      struct sk_buff          * volatile send_tail;
94      struct sk_buff_head             back_log;
95      struct sk_buff          *partial;
96      struct timer_list       partial_timer;
97      long                    retransmits;
98      struct sk_buff_head             write_queue,
99                      receive_queue;
100     struct proto            *prot;
101     struct wait_queue               **sleep;
102     unsigned long                   daddr;
103     unsigned long                   saddr;
104     unsigned short          max_unacked;
105     unsigned short          window;
106     unsigned short          bytes_rcv;
107 /* mss is min(mtu, max_window) */
108     unsigned short          mtu;            /* mss negotiated in the syn's */
109     volatile unsigned short     mss;            /* current eff. mss - can change */
110     volatile unsigned short     user_mss;   /* mss requested by user in ioctl */
111     volatile unsigned short     max_window;
112     unsigned long           window_clamp;
113     unsigned short          num;
114     volatile unsigned short     cong_window;
115     volatile unsigned short     cong_count;
116     volatile unsigned short     ssthresh;
117     volatile unsigned short     packets_out;
118     volatile unsigned short     shutdown;
119     volatile unsigned long  rtt;
120     volatile unsigned long  mdev;
121     volatile unsigned long  rto;
122 /* currently backoff isn't used, but I'm maintaining it in case
123  * we want to go back to a backoff formula that needs it
124  */
```

```
125     volatile unsigned short     backoff;
126     volatile short              err;
127     unsigned char               protocol;
128     volatile unsigned char      state;
129     volatile unsigned char      ack_backlog;
130     unsigned char               max_ack_backlog;
131     unsigned char               priority;
132     unsigned char               debug;
133     unsigned short              rcvbuf;
134     unsigned short              sndbuf;
135     unsigned short              type;
136     unsigned char               localroute;     /* Route locally only */
137 #ifdef CONFIG_IPX
138     ipx_address                 ipx_dest_addr;
139     ipx_interface               *ipx_intrfc;
140     unsigned short              ipx_port;
141     unsigned short              ipx_type;
142 #endif
143 #ifdef CONFIG_AX25
144 /* Really we want to add a per protocol private area */
145     ax25_address                ax25_source_addr,ax25_dest_addr;
146     struct sk_buff *volatile    ax25_retxq[8];
147     char                        ax25_state,ax25_vs,ax25_vr,ax25_lastrxnr,ax25_lasttxnr;
148     char                        ax25_condition;
149     char                        ax25_retxcnt;
150     char                        ax25_xx;
151     char                        ax25_retxqi;
152     char                        ax25_rrtimer;
153     char                        ax25_timer;
154     unsigned char               ax25_n2;
155     unsigned short              ax25_t1,ax25_t2,ax25_t3;
156     ax25_digi                   *ax25_digipeat;
157 #endif
158 #ifdef CONFIG_ATALK
159     struct atalk_sock           at;
160 #endif

161 /* IP 'private area' or will be eventually */
162     int                         ip_ttl;         /* TTL setting */
163     int                         ip_tos;         /* TOS */
164     struct tcphdr               dummy_th;
165     struct timer_list           keepalive_timer;    /* TCP keepalive hack */
166     struct timer_list           retransmit_timer;   /* TCP retransmit timer */
167     struct timer_list           ack_timer;          /* TCP delayed ack timer */
```

```
168     int                     ip_xmit_timeout;    /* Why the timeout is running */
169 #ifdef CONFIG_IP_MULTICAST
170     int                     ip_mc_ttl;              /* Multicasting TTL */
171     int                     ip_mc_loop;             /* Loopback (not implemented yet) */
172     char                    ip_mc_name[MAX_ADDR_LEN];       /* Multicast device name */
173     struct ip_mc_socklist       *ip_mc_list;            /* Group array */
174 #endif

175     /* This part is used for the timeout functions (timer.c). */
176     int                     timeout;   /* What are we waiting for? */
177     struct timer_list       timer;          /* This is the TIME_WAIT/receive timer when we
are doing IP */
178     struct timeval          stamp;

179     /* identd */
180     struct socket               *socket;

181     /* Callbacks */
182     void                    (*state_change)(struct sock *sk);
183     void                    (*data_ready)(struct sock *sk,int bytes);
184     void                    (*write_space)(struct sock *sk);
185     void                    (*error_report)(struct sock *sk);

186 };

187 struct proto {
188     struct sk_buff *    (*wmalloc)(struct sock *sk,
189                             unsigned long size, int force,
190                             int priority);
191     struct sk_buff *    (*rmalloc)(struct sock *sk,
192                             unsigned long size, int force,
193                             int priority);
194     void                (*wfree)(struct sock *sk, struct sk_buff *skb,
195                           unsigned long size);
196     void                (*rfree)(struct sock *sk, struct sk_buff *skb,
197                           unsigned long size);
198     unsigned long           (*rspace)(struct sock *sk);
199     unsigned long           (*wspace)(struct sock *sk);
200     void                (*close)(struct sock *sk, int timeout);
201     int                 (*read)(struct sock *sk, unsigned char *to,
202                         int len, int nonblock, unsigned flags);
203     int                 (*write)(struct sock *sk, unsigned char *to,
204                           int len, int nonblock, unsigned flags);
205     int                 (*sendto)(struct sock *sk,
```

```
206                         unsigned char *from, int len, int noblock,
207                         unsigned flags, struct sockaddr_in *usin,
208                         int addr_len);
209     int             (*recvfrom)(struct sock *sk,
210                         unsigned char *from, int len, int noblock,
211                         unsigned flags, struct sockaddr_in *usin,
212                         int *addr_len);
213     int             (*build_header)(struct sk_buff *skb,
214                         unsigned long saddr,
215                         unsigned long daddr,
216                         struct device **dev, int type,
217                         struct options *opt, int len, int tos, int ttl);
218     int             (*connect)(struct sock *sk,
219                         struct sockaddr_in *usin, int addr_len);
220     struct sock *       (*accept) (struct sock *sk, int flags);
221     void            (*queue_xmit)(struct sock *sk,
222                             struct device *dev, struct sk_buff *skb,
223                             int free);
224     void            (*retransmit)(struct sock *sk, int all);
225     void            (*write_wakeup)(struct sock *sk);
226     void            (*read_wakeup)(struct sock *sk);
227     int             (*rcv)(struct sk_buff *buff, struct device *dev,
228                         struct options *opt, unsigned long daddr,
229                         unsigned short len, unsigned long saddr,
230                         int redo, struct inet_protocol *protocol);
231     int             (*select)(struct sock *sk, int which,
232                         select_table *wait);
233     int             (*ioctl)(struct sock *sk, int cmd,
234                         unsigned long arg);
235     int             (*init)(struct sock *sk);
236     void            (*shutdown)(struct sock *sk, int how);
237     int             (*setsockopt)(struct sock *sk, int level, int optname,
238                         char *optval, int optlen);
239     int             (*getsockopt)(struct sock *sk, int level, int optname,
240                         char *optval, int *option);
241     unsigned short  max_header;
242     unsigned long       retransmits;
243     struct sock *       sock_array[SOCK_ARRAY_SIZE];
244     char            name[80];
245     int             inuse, highestinuse;
246 };
```

注意 243 行 proto 结构中 sock_array 字段，SOCK_ARRAY_SIZE 常量在本文件前面被定义为 256。sock_array 字段表示一个 sock 结构数组，每个数组元素表示一个 sock 结构队列。我们

在前面代码分析中使用 get_sock, put_sock 函数对 sock 函数进行查找或者将其挂入到系统队列中时，操作的对象就此处 sock_array 表示的数组中各元素指向的 sock 结构队列。要解释清楚这个问题，我们以 TCP 协议为例（UDP 协议类同）。在分析 tcp.c 文件时，在文件尾部定义了一个 tcp_prot 全局变量，这是一个系统全局变量，工作该版本网络代码之上的所有使用 TCP 协议进行网络数据传输的进程所使用的套接字（抱歉，有点长），都使用这个 tcp_prot 变量，换句话说，本地所有使用 TCP 协议的套接字所对应的 sock 结构都挂接在 tcp_prot 这个 proto 结构类型的 sock_array 数组中。UDP 协议也一样，本机上所有使用 UDP 协议的套接字所对应的 sock 结构都挂接在 udp_prot 这个 proto 结构类型的 sock_array 数组中。系统在创建一个新的套接字时，都会对应创建一个 sock 结构，这个 sock 结构 prot 字段的初始化：如果使用 TCP 协议，则被初始化为 tcp_prot;如果使用 UDP 协议，则被初始化为 udp_prot。同时在 sock 结构初始化完成后，其本身被插入 tcp_prot 中 sock_array 数组元素的相应队列中，数组元素的索引即为本地端口号对数组长度取余（本地端口号%数组长度-1）。在接收到一个新的数据包时，我们调用 get_sock 函数或者本地对应的 sock 结构，如下再次给出 get_sock 函数代码进行分析：

```
//net/inet/af_inet.c
1197struct sock *get_sock(struct proto *prot, unsigned short num,
1198                       unsigned long raddr,
1199                       unsigned short rnum, unsigned long laddr)
1200     {
```

我们再给出对应 TCP 协议和 UDP 协议调用该函数时的参数设置情况。

```
//net/inet/udp.c—udp_rcv
537             sk = get_sock(&udp_prot, uh->dest, saddr, uh->source, daddr);
```

uh 表示所接收数据包的 UDP 首部。

```
//net/inet/tcp.c –-tcp_rcv
3925            sk = get_sock(&tcp_prot, th->dest, saddr, th->source, daddr);
```

th 表示所接收数据包的 TCP 首部。

那么意义就很明显了，get_sock 函数中 num 表示的是本地端口号，再看 get_sock 函数中 1205，1214 行代码，首先将网络字节序（Big-endian）转换为主机字节序，之后以本地端口号作为索引对 udp_prot 或者 tcp_prot 中 sock_array 数组进行寻址，得到对应元素指向的那个 sock 结构队列。进而以其他参数进行合适的 sock 结构查找。

```
1201            struct sock *s;
1202            struct sock *result = NULL;
1203            int badness = -1;
1204            unsigned short hnum;

1205            hnum = ntohs(num);

1206            /*
1207             * SOCK_ARRAY_SIZE must be a power of two.    This will work better
1208             * than a prime unless 3 or more sockets end up using the same
1209             * array entry.    This should not be a problem because most
```

```
1210            * well known sockets don't overlap that much, and for
1211        * the other ones, we can just be careful about picking our
1212            * socket number when we choose an arbitrary one.
1213            */

1214           for(s = prot->sock_array[hnum & (SOCK_ARRAY_SIZE - 1)];
1215                   s != NULL; s = s->next)
1216           {
1217                   int score = 0;

1218                   if (s->num != hnum)
1219                           continue;

1220                   if(s->dead && (s->state == TCP_CLOSE))
1221                           continue;
1222                   /* local address matches? */
1223                   if (s->saddr) {
1224                           if (s->saddr != laddr)
1225                                   continue;
1226                           score++;
1227                   }
1228                   /* remote address matches? */
1229                   if (s->daddr) {
1230                           if (s->daddr != raddr)
1231                                   continue;
1232                           score++;
1233                   }
1234                   /* remote port matches? */
1235                   if (s->dummy_th.dest) {
1236                           if (s->dummy_th.dest != rnum)
1237                                   continue;
1238                           score++;
1239                   }
1240                   /* perfect match? */
1241                   if (score == 3)
1242                           return s;
1243                   /* no, check if this is the best so far.. */
1244                   if (score <= badness)
1245                           continue;
1246                   result = s;
1247                   badness = score;
1248           }
1249           return result;
1250   }
```

此处所要表达的意义是：tcp_prot 对于 TCP 协议是唯一的，所有使用 TCP 协议进行跨主机间通信的进程（或者更近一步：套接字）都要使用这个变量，而且套接字对应的 sock 结构都被插入到 tcp_prot 所表示 proto 结构中 sock_array 数组对应元素指向的 sock 结构队列中（这可真拗口）。
至于是 sock_array 数组中哪个元素，由本地端口进行索引。udp_prot 变量之于 UDP 协议的作用类同。

```
247  #define TIME_WRITE   1    //超时重传
248  #define TIME_CLOSE   2     //2MSL 定时
249  #define TIME_KEEPOPEN   3 //保活
250  #define TIME_DESTROY     4 //套接字释放
251  #define TIME_DONE    5     /* used to absorb those last few packets */
252  #define TIME_PROBE0  6    //非 0 窗口探测
```

以上这些变量定义了 retransmit_timer 定时器当前定时的目的。TCP 协议规范上定义了 4 个系统定时器：超时重发定时器，窗口探测定时器，保活（Keepopen or Keepalive）定时器，2MSL 定时器。本版本 sock 结构中虽然对应有这四个定时器字段，但实际上只使用了超时重传定时器，2MSL 定时器。而窗口探测，保活都是使用超时重传定时器，由于多个功能集中于使用同一个定时器，所以 sock 结构中 ip_xmit_timeout 字段专门用于指定超时重传定时器当前的定时目的。

```
253  #define SOCK_DESTROY_TIME 1000      /* about 10 seconds             */
```

```
254  #define PROT_SOCK    1024/* Sockets 0-1023 can't be bound too unless you are superuser
*/
```

PROT_SOCK 常量定义了自动分配本地通信端口号的起始值或者说最小开始值。

```
255  #define SHUTDOWN_MASK      3    //完全关闭
256  #define RCV_SHUTDOWN  1      //接收通道被关闭（远端发送了 FIN 数据包）
257  #define SEND_SHUTDOWN      2 //发送通道关闭（本地主动发送 FIN 数据包）
```

以下是对一些函数的声明。

```
258  extern void              destroy_sock(struct sock *sk);
259  extern unsigned short          get_new_socknum(struct proto *, unsigned short);
260  extern void              put_sock(unsigned short, struct sock *);
261  extern void              release_sock(struct sock *sk);
262  extern struct sock       *get_sock(struct proto *, unsigned short,
263                                 unsigned long, unsigned short,
264                                 unsigned long);
265  extern struct sock       *get_sock_mcast(struct sock *, unsigned short,
266                                 unsigned long, unsigned short,
267                                 unsigned long);
268  extern struct sock       *get_sock_raw(struct sock *, unsigned short,
```

```
269                                 unsigned long, unsigned long);

270  extern struct sk_buff          *sock_wmalloc(struct sock *sk,
271                                     unsigned long size, int force,
272                                     int priority);
273  extern struct sk_buff          *sock_rmalloc(struct sock *sk,
274                                     unsigned long size, int force,
275                                     int priority);
276  extern void                sock_wfree(struct sock *sk, struct sk_buff *skb,
277                                     unsigned long size);
278  extern void                sock_rfree(struct sock *sk, struct sk_buff *skb,
279                                     unsigned long size);
280  extern unsigned long           sock_rspace(struct sock *sk);
281  extern unsigned long           sock_wspace(struct sock *sk);

282  extern int             sock_setsockopt(struct sock *sk,int level,int op,char *optval,int optlen);

283  extern int             sock_getsockopt(struct sock *sk,int level,int op,char *optval,int *optlen);
284  extern struct sk_buff          *sock_alloc_send_skb(struct sock *skb, unsigned long size, int
noblock, int *errcode);
285  extern int             sock_queue_rcv_skb(struct sock *sk, struct sk_buff *skb);

286  /* declarations from timer.c */
287  extern struct sock *timer_base;

288  void delete_timer (struct sock *);
289  void reset_timer (struct sock *, int, unsigned long);
290  void net_timer (unsigned long);

291  #endif     /* _SOCK_H */
```

对本文件的理解着重于对 sock 结构和 proto 结构中 sock_array 数组，以及两个全局变量 tcp_prot, udp_prot 和其他套接字 sock 结构之间关系的理解。分清他们之间的关系，将对网络实现代码进入了更深层次的理解。

2.9 net/inet/sock.c 文件

sock.c 文件中定义函数完成的功能独立于传输层协议，但为传输层协议工作。如内存分配函数，接收或发送缓冲区大小设置等等。所谓独立于传输层协议是指其将传输层协议同样的地方抽出来，单独实现，避免各自有一套相同的代码，减少了代码的冗余。从这个大的方向把握，有助于我们对 sock.c 文件的理解。

```
1     /*
2      * INET        An implementation of the TCP/IP protocol suite for the LINUX
3      *             operating system.    INET is implemented using the    BSD Socket
```

```
4       *       interface as the means of communication with the user level.
5       *
6       *       Generic socket support routines. Memory allocators, sk->inuse/release
7       *       handler for protocols to use and generic option handler.
8       *
9       *
10      * Version:      @(#)sock.c      1.0.17      06/02/93
11      *
12      * Authors:      Ross Biro, <bir7@leland.Stanford.Edu>
13      *       Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
14      *       Florian La Roche, <flla@stud.uni-sb.de>
15      *       Alan Cox, <A.Cox@swansea.ac.uk>
16      *
17      * Fixes:
18      *       Alan Cox:       Numerous verify_area() problems
19      *       Alan Cox:       Connecting on a connecting socket
20      *                       now returns an error for tcp.
21      *       Alan Cox:       sock->protocol is set correctly.
22      *                       and is not sometimes left as 0.
23      *       Alan Cox:       connect handles icmp errors on a
24      *                       connect properly. Unfortunately there
25      *                       is a restart syscall nasty there. I
26      *                       can't match BSD without hacking the C
27      *                       library. Ideas urgently sought!
28      *       Alan Cox:       Disallow bind() to addresses that are
29      *                       not ours - especially broadcast ones!!
30      *       Alan Cox:       Socket 1024 _IS_ ok for users. (fencepost)
31      *       Alan Cox:       sock_wfree/sock_rfree don't destroy sockets,
32      *                       instead they leave that for the DESTROY timer.
33      *       Alan Cox:       Clean up error flag in accept
34      *       Alan Cox:       TCP ack handling is buggy, the DESTROY timer
35      *                       was buggy. Put a remove_sock() in the handler
36      *                       for memory when we hit 0. Also altered the timer
37      *                       code. The ACK stuff can wait and needs major
38      *                       TCP layer surgery.
39      *       Alan Cox:       Fixed TCP ack bug, removed remove sock
40      *                       and fixed timer/inet_bh race.
41      *       Alan Cox:       Added zapped flag for TCP
42      *       Alan Cox:       Move kfree_skb into skbuff.c and tidied up surplus code
43      *       Alan Cox:       for new sk_buff allocations wmalloc/rmalloc now call alloc_skb
44      *       Alan Cox:       kfree_s calls now are kfree_skbmem so we can track skb resources
45      *       Alan Cox:       Supports socket option broadcast now as does udp. Packet and raw
need fixing.
46      *       Alan Cox:       Added RCVBUF,SNDBUF size setting. It suddenly occurred to me
```

```
how easy it was so...
47    *        Rick Sladkey  :      Relaxed UDP rules for matching packets.
48    *        C.E.Hawkins  :      IFF_PROMISC/SIOCGHWADDR support
49    *    Pauline Middelink  :      identd support
50    *        Alan Cox:      Fixed connect() taking signals I think.
51    *        Alan Cox:      SO_LINGER supported
52    *        Alan Cox:      Error reporting fixes
53    *        Anonymous    :      inet_create tidied up (sk->reuse setting)
54    *        Alan Cox:      inet sockets don't set sk->type!
55    *        Alan Cox:      Split socket option code
56    *        Alan Cox:      Callbacks
57    *        Alan Cox:      Nagle flag for Charles & Johannes stuff
58    *        Alex         :      Removed restriction on inet fioctl
59    *        Alan Cox:      Splitting INET from NET core
60    *        Alan Cox:      Fixed bogus SO_TYPE handling in getsockopt()
61    *        Adam Caldwell      :      Missing return in SO_DONTROUTE/SO_DEBUG code
62    *        Alan Cox:      Split IP from generic code
63    *        Alan Cox:      New kfree_skbmem()
64    *        Alan Cox:      Make SO_DEBUG superuser only.
65    *        Alan Cox:      Allow anyone to clear SO_DEBUG
66    *                       (compatibility fix)
67    *
68    * To Fix:
69    *
70    *
71    *        This program is free software; you can redistribute it and/or
72    *        modify it under the terms of the GNU General Public License
73    *        as published by the Free Software Foundation; either version
74    *        2 of the License, or (at your option) any later version.
75    */

76   #include <linux/config.h>
77   #include <linux/errno.h>
78   #include <linux/types.h>
79   #include <linux/socket.h>
80   #include <linux/in.h>
81   #include <linux/kernel.h>
82   #include <linux/major.h>
83   #include <linux/sched.h>
84   #include <linux/timer.h>
85   #include <linux/string.h>
86   #include <linux/sockios.h>
87   #include <linux/net.h>
88   #include <linux/fcntl.h>
```

```
89  #include <linux/mm.h>
90  #include <linux/interrupt.h>

91  #include <asm/segment.h>
92  #include <asm/system.h>

93  #include <linux/inet.h>
94  #include <linux/netdevice.h>
95  #include "ip.h"
96  #include "protocol.h"
97  #include "arp.h"
98  #include "rarp.h"
99  #include "route.h"
100 #include "tcp.h"
101 #include "udp.h"
102 #include <linux/skbuff.h>
103 #include "sock.h"
104 #include "raw.h"
105 #include "icmp.h"

106 #define min(a,b)    ((a)<(b)?(a):(b))
```

宏 min 的作用显然：取两数中小值。

接下来分析的两个函数为参数设置函数，我们前文在分析 TCP，UDP 协议时，他们各自也有相应的参数设置函数如 tcp_setsockopt 诸如此类。但这些嵌入在具体协议实现代码中的参数设置所对应的参数是协议相关的，如对 TCP 协议 Nagle 标志位的设置。而本文件中设置的参数是传输层协议无关的，换句话说，对于 TCP，UDP 协议，都有这样的参数需要设置如发送缓冲区大小，如此就没有必要分别在 TCP 和 UDP 协议中设置，而是将这个共性抽出来，作为一个单独的模块实现，诚如上文刚刚所述，这就是 sock.c 文件的本质。
我们首先看参数设置函数，对于参数获取函数，是与之对应的，之后我们不再进行分析，只列出其代码。

```
107 /*
108  *  This is meant for all protocols to use and covers goings on
109  *  at the socket level. Everything here is generic.
110  */

111 int sock_setsockopt(struct sock *sk, int level, int optname,
112                 char *optval, int optlen)
113 {
114         int val;
115         int err;
116         struct linger ling;
```

```
117     if (optval == NULL)
118         return(-EINVAL);
```

这一点毫无疑义，对于参数设置而言，如果没有指定参数值，那么就没有必要进行下面的处理工作。

```
119     err=verify_area(VERIFY_READ, optval, sizeof(int));
120     if(err)
121         return err;
```

verify_area 函数验证参数地址空间是否存在，如果不存在，还需要进行分配和映射。

```
122     val = get_fs_long((unsigned long *)optval);
123     switch(optname)
124     {
```

获取参数值，然后根据具体的参数名称进行参数设置。

```
125         case SO_TYPE:
126         case SO_ERROR:
127             return(-ENOPROTOOPT);

128         case SO_DEBUG:
129             if(val && !suser())
130                 return(-EPERM);
131             sk->debug=val?1:0;
132             return 0;
133         case SO_DONTROUTE:
134             sk->localroute=val?1:0;
135             return 0;
136         case SO_BROADCAST:
137             sk->broadcast=val?1:0;
138             return 0;
139         case SO_SNDBUF:
140             if(val>32767)
141                 val=32767;
142             if(val<256)
143                 val=256;
144             sk->sndbuf=val;
145             return 0;
146         case SO_LINGER:
147             err=verify_area(VERIFY_READ,optval,sizeof(ling));
148             if(err)
149                 return err;
150             memcpy_fromfs(&ling,optval,sizeof(ling));
```

```
151                 if(ling.l_onoff==0)
152                     sk->linger=0;
153                 else
154                 {
155                     sk->lingertime=ling.l_linger;
156                     sk->linger=1;
157                 }
158                 return 0;
```

SO_LINGER 选项可以让我们设置服务器端套接字在设置为 TCP_CLOSE 状态依然“驻留”的时间。所谓驻留（Linger）表示套接字在关闭后保持为“被使用（inuse）“状态的时间。保持为”被使用“状态，换句话说，在这期间，不可被重新使用，类似于我们前文中介绍的 TCP_TIME_WAIT 状态时 2MSL 等待时间的概念。不过一般我们是对服务器端套接字使用 SO_LINGER 选项，该选项在套接字被创建时被设置为 0，表示套接字不进行关闭后”驻留“等待。linger 结构定义在 include/linux/socket.h 文件中，如下所示：

```
//include/linux/socket.h
8   struct linger {
9       int             l_onoff;    /* Linger active            */
10      int             l_linger;   /* How long to linger for       */
11  };
```

其中 l_onff 是表示是否进行关闭后等待的标志位；l_linger 则表示等待时间。

```
159             case SO_RCVBUF:
160                 if(val>32767)
161                     val=32767;
162                 if(val<256)
163                     val=256;
164                 sk->rcvbuf=val;
165                 return(0);
```

SO_RCVBUF 选项用于设置接收缓冲区的大小。139 行代码是对发送缓冲区大小的设置。

```
166             case SO_REUSEADDR:
167                 if (val)
168                     sk->reuse = 1;
169                 else
170                     sk->reuse = 0;
171                 return(0);

172             case SO_KEEPALIVE:
173                 if (val)
174                     sk->keepopen = 1;
175                 else
176                     sk->keepopen = 0;
```

```
177                 return(0);
```

SO_KEEPALIVE 选项用于设置是否进行保活探测。这个是由 sock 结构中 keepopen 字段决定的。

```
178             case SO_OOBINLINE:
179                 if (val)
180                     sk->urginline = 1;
181                 else
182                     sk->urginline = 0;
183                 return(0);
```

OOB 数据称为带外数据（Out Of Band），这种数据与紧急数据其实是不同的，带外数据的传输需要重建数据通道，而紧急数据是嵌入在普通数据流中的，通过 TCP 首部中紧急数据指针指出紧急数据范围。sock 结构中 urginline 标志位表示怎样对待紧急数据，如果该标志位被设置为 1，则将紧急数据处理为普通数据。即在读取时，不再进行区别对待。

```
184             case SO_NO_CHECK:
185                 if (val)
186                     sk->no_check = 1;
187                 else
188                     sk->no_check = 0;
189                 return(0);
```

SO_NO_CHECK 选项用于设置是否进行校验和计算，TCP 协议校验和计算是必须的。无论此处设置为何值，都会进行 TCP 校验和计算。UDP，ICMP 协议根据此处的设置决定是否进行校验和计算。

```
190              case SO_PRIORITY:
191                 if (val >= 0 && val < DEV_NUMBUFFS)
192                 {
193                     sk->priority = val;
194                 }
195                 else
196                 {
197                     return(-EINVAL);
198                 }
199                 return(0);
```

SO_PRIORITY 选项用于设置对应该套接字的所有发送数据包缓存到硬件缓冲区队列中的优先级。网络层模块在发送数据包给链路层时（此处层次的概念只是为了便于理解，实际代码实现上相互耦合），链路层模块将数据包缓存到硬件队列中，硬件队列区分优先级，或者更准确的说，共有 DEV_NUMBUFFS 个硬件队列，硬件驱动在发送数据时，从第一个队列开始遍历处理，这样由于发送先后上的顺序就自然了形成了数据包不同的发送优先级。此处 SO_PRIORITY 选项的作用就是设置该套接字发送的数据包所缓存的硬件队列。

```
200             default:
201                     return(-ENOPROTOOPT);
202         }
203 }
```

以上对几个重要的比较难以理解的选项进行了解释，其他选项的意义都比较明显，读者可自行分析理解。

```
204 int sock_getsockopt(struct sock *sk, int level, int optname,
205                     char *optval, int *optlen)
206 {
207         int val;
208         int err;
209         struct linger ling;

210         switch(optname)
211         {
212             case SO_DEBUG:
213                     val = sk->debug;
214                     break;

215             case SO_DONTROUTE:
216                     val = sk->localroute;
217                     break;

218             case SO_BROADCAST:
219                     val= sk->broadcast;
220                     break;

221             case SO_LINGER:
222                     err=verify_area(VERIFY_WRITE,optval,sizeof(ling));
223                     if(err)
224                         return err;
225                     err=verify_area(VERIFY_WRITE,optlen,sizeof(int));
226                     if(err)
227                         return err;
228                     put_fs_long(sizeof(ling),(unsigned long *)optlen);
229                     ling.l_onoff=sk->linger;
230                     ling.l_linger=sk->lingertime;
231                     memcpy_tofs(optval,&ling,sizeof(ling));
232                     return 0;

233             case SO_SNDBUF:
```

```
234                 val=sk->sndbuf;
235                 break;

236             case SO_RCVBUF:
237                 val =sk->rcvbuf;
238                 break;

239             case SO_REUSEADDR:
240                 val = sk->reuse;
241                 break;

242             case SO_KEEPALIVE:
243                 val = sk->keepopen;
244                 break;

245             case SO_TYPE:
246 #if 0
247                 if (sk->prot == &tcp_prot)
248                     val = SOCK_STREAM;
249                 else
250                     val = SOCK_DGRAM;
251 #endif
252                 val = sk->type;
253                 break;

254             case SO_ERROR:
255                 val = sk->err;
256                 sk->err = 0;
257                 break;

258             case SO_OOBINLINE:
259                 val = sk->urginline;
260                 break;

261             case SO_NO_CHECK:
262                 val = sk->no_check;
263                 break;

264             case SO_PRIORITY:
265                 val = sk->priority;
266                 break;

267             default:
268                 return(-ENOPROTOOPT);
```

```
269         }
270         err=verify_area(VERIFY_WRITE, optlen, sizeof(int));
271         if(err)
272                 return err;
273         put_fs_long(sizeof(int),(unsigned long *) optlen);

274         err=verify_area(VERIFY_WRITE, optval, sizeof(int));
275         if(err)
276                 return err;
277         put_fs_long(val,(unsigned long *)optval);

278         return(0);
279 }
```

sock_getsockopt 函数与 sock_setsockopt 函数是相对应的。此处不再讨论。

```
280 struct sk_buff *sock_wmalloc(struct sock *sk, unsigned long size, int force, int priority)
281 {
282         if (sk)
283         {
284                 if (sk->wmem_alloc + size < sk->sndbuf || force)
285                 {
286                         struct sk_buff * c = alloc_skb(size, priority);
287                         if (c)
288                         {
289                                 unsigned long flags;
290                                 save_flags(flags);
291                                 cli();
292                                 sk->wmem_alloc+= c->mem_len;
293                                 restore_flags(flags); /* was sti(); */
294                         }
295                         return c;
296                 }
297                 return(NULL);
298         }
299         return(alloc_skb(size, priority));
300 }
```

sock_wmalloc 是发送数据包时内核缓冲区分配函数，其在分配之前，主要是检查当前发送缓冲区是否有足够剩余空闲空间分配指定大小的缓冲区。此处我们可以澄清一个概念，所谓发送缓冲区并非是指系统分配一段连续的指定大小的缓冲区，然后发送的数据都依次填充到这个区域被内核网络代码处理。实际上所谓发送缓冲区（接收缓冲区也如此）紧急指定的时一个大小，至于具体的缓冲区空间则随时随地进行分配，换句话说，这些分配的缓冲区不存在连续的概念，内核所关心的只是这些分配的缓冲区大小的总和是否超过了指定的大小。一旦

超出这个指定值，就表示缓冲区溢出，此时无法在分配缓冲区。所以一言以蔽之，即发送或者接收缓冲区只是一个大小上的限定，没有固定分配一段内存空间之说。sock_wmalloc 函数以及下面将要介绍的 sock_rmalloc, sock_wfree, sock_rfree, sock_rspace, sock_wspace 等函数都被传输层协议使用完成对应的工作。读者可查看 tcp_prot, udp_prot 变量的定义方式进行验证。以上这些函数的意义如同其函数名称一样，都是用于操作发送或者接收缓冲区，具体功能在下文分析中一一道来，首先看 sock_wmalloc 函数。

282 检查缓冲区分配是否与一个套接字绑定，如果进行了绑定，则分配的缓冲区需要进行统计，如果对应套接字当前分配的缓冲区大小已经超出限制值，则返回 NULL，表示无法分配缓冲区，发送缓冲区已经溢出。具体的分配工作交给 alloc_skb 函数，alloc_skb 函数定义在 net/inet/sk_buff.c 中，其调用 kmalloc 函数完成内存分配，并且将这段分配内存封装在 sk_buff 结构中，并对 sk_buff 结构相关字段进行初始化。具体情况在分析 sk_buff.c 文件时进行说明。如果缓冲区分配不与一个套接字绑定，则直接分配要求数量的内存，此时无需进行缓冲区分配大小统计字段的更新。

```
301  struct sk_buff *sock_rmalloc(struct sock *sk, unsigned long size, int force, int priority)
302  {
303          if (sk)
304          {
305                  if (sk->rmem_alloc + size < sk->rcvbuf || force)
306                  {
307                          struct sk_buff *c = alloc_skb(size, priority);
308                          if (c)
309                          {
310                                  unsigned long flags;
311                                  save_flags(flags);
312                                  cli();
313                                  sk->rmem_alloc += c->mem_len;
314                                  restore_flags(flags); /* was sti(); */
315                          }
316                          return(c);
317                  }
318                  return(NULL);
319          }
320          return(alloc_skb(size, priority));
321  }
```

sock_rmalloc 函数用于分配接收缓冲区，内存分配基本相同，只不过此时更新的是接收缓冲区当前分配大小值，这一点是在 313 行完成的。305 行进行预先检查。

```
322  unsigned long sock_rspace(struct sock *sk)
323  {
324          int amt;

325          if (sk != NULL)
```

```
326         {
327             if (sk->rmem_alloc >= sk->rcvbuf-2*MIN_WINDOW)
328                 return(0);
329             amt = min((sk->rcvbuf-sk->rmem_alloc)/2-MIN_WINDOW, MAX_WINDOW);
330             if (amt < 0)
331                 return(0);
332             return(amt);
333         }
334         return(0);
335 }
```

sock_rspace 函数用于检查接收缓冲区空闲空间大小，该函数被调用的较为频繁，而且在处理上有些“欺诈“性质，其根本原因在于这个函数返回值将直接被用于 TCP 首部中窗口字段的创建。该窗口字段用于向远端表明本地当前可接收的数据量，为了防止产生小数据包，所以本地缓冲区空闲空间计算上并不是将限制值减去当前使用值，而是预留了一个最小窗口，327 行将这个最小窗口设置为 2*MIN_WINDOW, MIN_WINDOW 定义在 tcp.h 中为 2048。如果当前接收缓冲区空闲空间小于这个最小窗口，则作为 0 对待。329 行代码计算空闲空间的方式很特别，其将真正空闲空间减半后再减去 MIN_WINDOW 后的结果作为当前接收缓冲区空闲大小。通过这种方式计算出的空闲空间大小大大缩水了。此处所要说明的是，这种计算方式是实现相关的，其基本底线是本地向远端声明的本地窗口大小最好不要发生缩减的情况（即本地声明的窗口大小小于上一次声明的值），而且这个窗口大小不要小于一个报文长度（以防促使远端产生小数据包，当然如果远端自己需要产生小数据包，则另当别论，如 Telnet，Rlogin 应用等）。

```
336 unsigned long sock_wspace(struct sock *sk)
337 {
338         if (sk != NULL)
339         {
340             if (sk->shutdown & SEND_SHUTDOWN)
341                 return(0);
342             if (sk->wmem_alloc >= sk->sndbuf)
343                 return(0);
344             return(sk->sndbuf-sk->wmem_alloc );
345         }
346         return(0);
347 }
```

对于发送缓冲区当前空闲空间大小，sock_wspace 比较直接，因为这个返回值被本地使用，既然“大家都是一家人“，就”不说两家话“了，我有多少空闲空间，就通知你多少（344 行代码）。

```
348 void sock_wfree(struct sock *sk, struct sk_buff *skb, unsigned long size)
349 {
350 #ifdef CONFIG_SKB_CHECK
351         IS_SKB(skb);
352 #endif
```

```
353         kfree_skbmem(skb, size);
354         if (sk)
355         {
356                 unsigned long flags;
357                 save_flags(flags);
358                 cli();
359                 sk->wmem_alloc -= size;
360                 restore_flags(flags);
361                 /* In case it might be waiting for more memory. */
362                 if (!sk->dead)
363                         sk->write_space(sk);
364                 return;
365         }
366 }


367 void sock_rfree(struct sock *sk, struct sk_buff *skb, unsigned long size)
368 {
369 #ifdef CONFIG_SKB_CHECK
370         IS_SKB(skb);
371 #endif
372         kfree_skbmem(skb, size);
373         if (sk)
374         {
375                 unsigned long flags;
376                 save_flags(flags);
377                 cli();
378                 sk->rmem_alloc -= size;
379                 restore_flags(flags);
380         }
381 }
```

sock_wfree, sock_rfree 函数用于释放发送和接收缓冲区，并更新相关统计字段值。这两个函数实现较为浅显，不必多说。

```
382 /*
383  *    Generic send/receive buffer handlers
384  */

385 struct sk_buff *sock_alloc_send_skb(struct sock *sk, unsigned long size, int noblock, int *errcode)
386 {
```

sk 表示哪个套接字需要分配内存；size 为数据帧长度（包括各协议首部长度和用户数据长度）；noblock 表示如果暂时无法分配内存时是否进行睡眠等待；errcode 是一个地址，用于返回可能出现的错误。

```
387     struct sk_buff *skb;
388     int err;

389     sk->inuse=1;

390     do
391     {
```

390 行这个 do-while 循环用于“坚持不懈“的分配一个指定大小的 sk_buff 封装数据结构。

```
392         if(sk->err!=0)
393         {
394             cli();
395             err= -sk->err;
396             sk->err=0;
397             sti();
398             *errcode=err;
399             return NULL;
400         }
```

392 行这个 if 语句检查在分配内存的过程中，是否发生了错误，如果返回错误，则终止内存分配过程，返回这个错误值。

```
401         if(sk->shutdown&SEND_SHUTDOWN)
402         {
403             *errcode=-EPIPE;
404             return NULL;
405         }
```

sock_alloc_send_skb 函数用于分配发送数据包，如果对应套接字已经关闭发送通道，则返回 EPIPE 错误。因为这种可能“费尽了千辛万苦“分配得到的内存最后却被简单释放，实在是浪费资源，所以直接返回错误。

```
406         skb = sock_wmalloc(sk, size, 0, GFP_KERNEL);
```

sock_alloc_send_skb 函数封装对 sock_wmalloc 函数的调用更新发送缓冲区，并分配内存空间。

```
407         if(skb==NULL)
408         {
409             unsigned long tmp;
```

```
410                 sk->socket->flags |= SO_NOSPACE;
411                 if(noblock)
412                 {
413                     *errcode=-EAGAIN;
414                     return NULL;
415                 }
416                 if(sk->shutdown&SEND_SHUTDOWN)
417                 {
418                     *errcode=-EPIPE;
419                     return NULL;
420                 }
421                 tmp = sk->wmem_alloc;
422                 cli();
423                 if(sk->shutdown&SEND_SHUTDOWN)
424                 {
425                     sti();
426                     *errcode=-EPIPE;
427                     return NULL;
428                 }

429                 if( tmp <= sk->wmem_alloc)
430                 {
431                     sk->socket->flags &= ~SO_NOSPACE;
432                     interruptible_sleep_on(sk->sleep);
433                     if (current->signal & ~current->blocked)
434                     {
435                         sti();
436                         *errcode = -ERESTARTSYS;
437                         return NULL;
438                     }
439                 }
440                 sti();
441             }// if(skb==NULL)
```

407 行表示内存分配失败，此时根据 noblock 参数的设置决定是否睡眠等待。416 行对发送通道的再次检查源于对 sock_wmalloc 函数的调用可能发生睡眠等待，在这过程中，可能发送通道由于某种原因被关闭，所以此处进行再次检查；而 423 行的检查源于 cli()函数调用。一种解释就是在开启中断后，可能当前进程被中断，中断返回后可能调度其他进程运行，那么在本进程被重新换回来之后，需要再次对发送通道进行检查。代码中一再对发送通道进行检查的原因前文中已经交待：防止做无用功。429-439 行代码等待睡眠，等待发送缓冲区中有足够的空闲空间。

一旦被唤醒，而且唤醒不是因为被信号中断（此种情况直接返回处理信号中断），那么回到 406 行重新进行缓冲区分配。

```
442         }
443         while(skb==NULL);

444         return skb;
445 }
```

sock_alloc_send_skb 函数被 udp_send 函数调用用于分配一个 sk_buff 封装数据结构。TCP 协议在分配接收或者发送缓冲区时，直接调用 sock_wmalloc, sock_rmalloc 函数进行操作。而 UDP 协议则通过 sock_alloc_send_skb 以及下文将要介绍的 sock_queue_rcv_skb 函数进行分配。

```
446 /*
447  *  Queue a received datagram if it will fit. Stream and sequenced protocols
448  *  can't normally use this as they need to fit buffers in and play with them.
449  */

450 int sock_queue_rcv_skb(struct sock *sk, struct sk_buff *skb)
451 {
452         unsigned long flags;
453         if(sk->rmem_alloc + skb->mem_len >= sk->rcvbuf)
454                 return -ENOMEM;
455         save_flags(flags);
456         cli();
457         sk->rmem_alloc+=skb->mem_len;
458         skb->sk=sk;
459         restore_flags(flags);
460         skb_queue_tail(&sk->receive_queue,skb);
461         if(!sk->dead)
462                 sk->data_ready(sk,skb->len);
463         return 0;
464 }
```

sock_queue_rcv_skb 函数被 udp_deliver 函数调用将接收到的数据包插入到接收队列中。453 行判断当前接收缓冲区是否有足够空闲空间接收该数据包。如果有，则更新（457 行）统计字段值。此后调用（460 行）skb_queue_tail 将接收到的数据包插入到接收队列尾部，最后通知应用进程有数据包到达，可以进行处理了。

```
465 void release_sock(struct sock *sk)
466 {
467         unsigned long flags;
468 #ifdef CONFIG_INET
469         struct sk_buff *skb;
470 #endif
```

```
471         if (!sk->prot)
472                 return;
473         /*
474           *    Make the backlog atomic. If we don't do this there is a tiny
475           *    window where a packet may arrive between the sk->blog being
476           *    tested and then set with sk->inuse still 0 causing an extra
477           *    unwanted re-entry into release_sock().
478           */

479         save_flags(flags);
480         cli();
481         if (sk->blog)
482         {
483                 restore_flags(flags);
484                 return;
485         }
486         sk->blog=1;
487         sk->inuse = 1;
488         restore_flags(flags);
489 #ifdef CONFIG_INET
490         /* See if we have any packets built up. */
491         while((skb = skb_dequeue(&sk->back_log)) != NULL)
492         {
493                 sk->blog = 1;
494                 if (sk->prot->rcv)
495                         sk->prot->rcv(skb, skb->dev, sk->opt,
496                                 skb->saddr, skb->len, skb->daddr, 1,
497                                 /* Only used for/by raw sockets. */
498                                 (struct inet_protocol *)sk->pair);
499         }
500 #endif
501         sk->blog = 0;
502         sk->inuse = 0;
503 #ifdef CONFIG_INET
504         if (sk->dead && sk->state == TCP_CLOSE)
505         {
506                 /* Should be about 2 rtt's */
507                 reset_timer(sk, TIME_DONE, min(sk->rtt * 2, TCP_DONE_TIME));
508         }
509 #endif
510 }
```

release_sock 函数被调用的最为频繁，如果读者自己留意一下，在 TCP 协议（以及 UDP 协

议）实现中，大多数函数“有事没事“都调用一下该函数，那么这个函数究竟有什么”吸引之处“？秘密就在 491-499 行。网络层模块在将一个数据包传递给传输层模块处理时（tcp_rcv），如果当前对应的套接字正忙，则将数据包插入到 sock 结构 back_log 队列中。但插入该队列中的数据包并不能算是被接收，该队列中的数据包需要进行一系列处理后插入 receive_queue 接收队列中时，方才算是完成接收。而 release_sock 函数就是从 back_log 中取数据包重新调用 tcp_rcv 函数对数据包进行接收。所谓 back_log 队列只是数据包暂居之所，不可久留，所以也就必须对这个队列中数据包尽快进行处理，那么也就表示必须对 release_sock 函数进行频繁调用。这就是 release_sock 函数如此受”照顾“的原因。504-508 行代码在套接字关闭后等待一段时间后将内核相关数据结构完全释放。reset_timer 函数定义在 net/inet/timer.c 中，用新的定时间隔重置定时器。reset_timer 函数使用的是 sock 结构中 timer 字段指向的定时器，而定时器到期执行函数为 net_timer, net_timer 函数也是定义在 timer.c 中。该函数中对应 TIME_DONE 的处理代码如下，注意 111 行对 destroy_sock 函数的调用，用于释放对应套接字（destroy_sock 函数定义在 af_inet.c 中，在本书上文中已经进行了分析）。

```
//net/inet/timer.c
104             case TIME_DONE:
105                     if (! sk->dead || sk->state != TCP_CLOSE)
106                     {
107                             printk ("non dead socket in time_done\n");
108                             release_sock (sk);
109                             break;
110                     }
111                     destroy_sock (sk);
112                     break;
```

至此，完成 sock.c 文件的分析，该文件作为一个传输层通用功能实现文件而存在，主要是对发送和接收缓冲区分配进行处理。

2.10 net/inet/datagram.c 文件

实际上在介绍 udp.c 文件实现之前，就应该先介绍 datagram.c 文件，该文件定义了几个功能函数被 UDP 协议使用，主要涉及数据包接收和释放以及数据复制，此外还有一个查询函数。我们下面进入对该文件的分析。

```
1     /*
2      *    SUCS NET3:
3      *
4      *    Generic datagram handling routines. These are generic for all protocols. Possibly a generic IP version on top
5      *    of these would make sense. Not tonight however 8-).
6      *    This is used because UDP, RAW, PACKET and the to be released IPX layer all have identical select code and mostly
7      *    identical recvfrom() code. So we share it here. The select was shared before but buried in udp.c so I moved it.
8      *
9      *    Authors:  Alan Cox <iiitac@pyr.swan.ac.uk>. (datagram_select() from old udp.c code)
```

```
10      *
11      *    Fixes:
12      *          Alan Cox:       NULL return from skb_peek_copy() understood
13      *          Alan Cox:       Rewrote skb_read_datagram to avoid the skb_peek_copy stuff.
14      *          Alan Cox:       Added support for SOCK_SEQPACKET. IPX can no longer use the
SO_TYPE hack but
15      *                          AX.25 now works right, and SPX is feasible.
16      *          Alan Cox:       Fixed write select of non IP protocol crash.
17      *          Florian  La Roche:      Changed for my new skbuff handling.
18      *          Darryl Miles  :     Fixed non-blocking SOCK_SEQPACKET.
19      *
20      *    Note:
21      *          A lot of this will change when the protocol/socket separation
22      *    occurs. Using this will make things reasonably clean.
23      */

24    #include <linux/types.h>
25    #include <linux/kernel.h>
26    #include <asm/segment.h>
27    #include <asm/system.h>
28    #include <linux/mm.h>
29    #include <linux/interrupt.h>
30    #include <linux/in.h>
31    #include <linux/errno.h>
32    #include <linux/sched.h>
33    #include <linux/inet.h>
34    #include <linux/netdevice.h>
35    #include "ip.h"
36    #include "protocol.h"
37    #include "route.h"
38    #include "tcp.h"
39    #include "udp.h"
40    #include <linux/skbuff.h>
41    #include "sock.h"

42    /*
43      *    Get a datagram skbuff, understands the peeking, nonblocking wakeups and possible
44      *    races. This replaces identical code in packet,raw and udp, as well as the yet to
45      *    be released IPX support. It also finally fixes the long standing peek and read
46      *    race for datagram sockets. If you alter this routine remember it must be
47      *    re-entrant.
48      */

49    struct sk_buff *skb_recv_datagram(struct sock *sk, unsigned flags, int noblock, int *err)
```

```
50  {
51      struct sk_buff *skb;
52      unsigned long intflags;

53      /* Socket is inuse - so the timer doesn't attack it */
54      save_flags(intflags);
55  restart:
56      sk->inuse = 1;
57      while(skb_peek(&sk->receive_queue) == NULL)/* No data */
58      {
59          /* If we are shutdown then no more data is going to appear. We are done */
60          if (sk->shutdown & RCV_SHUTDOWN)
61          {
62              release_sock(sk);
63              *err=0;
64              return NULL;
65          }

66          if(sk->err)
67          {
68              release_sock(sk);
69              *err=-sk->err;
70              sk->err=0;
71              return NULL;
72          }

73          /* Sequenced packets can come disconnected. If so we report the problem */
74          if(sk->type==SOCK_SEQPACKET && sk->state!=TCP_ESTABLISHED)
75          {
76              release_sock(sk);
77              *err=-ENOTCONN;
78              return NULL;
79          }

80          /* User doesn't want to wait */
81          if (noblock)
82          {
83              release_sock(sk);
84              *err=-EAGAIN;
85              return NULL;
86          }
87          release_sock(sk);

88          /* Interrupts off so that no packet arrives before we begin sleeping.
```

```
89                  Otherwise we might miss our wake up */
90              cli();
91              if (skb_peek(&sk->receive_queue) == NULL)
92              {
93                      interruptible_sleep_on(sk->sleep);
94                      /* Signals may need a restart of the syscall */
95                      if (current->signal & ~current->blocked)
96                      {
97                              restore_flags(intflags);;
98                              *err=-ERESTARTSYS;
99                              return(NULL);
100                     }
101                     if(sk->err != 0)       /* Error while waiting for packet
102                                             eg an icmp sent earlier by the
103                                             peer has finally turned up now */
104                     {
105                             *err = -sk->err;
106                             sk->err=0;
107                             restore_flags(intflags);
108                             return NULL;
109                     }
110             }
111             sk->inuse = 1;
112             restore_flags(intflags);
113         }
114         /* Again only user level code calls this function, so nothing interrupt level
115              will suddenly eat the receive_queue */
116         if (!(flags & MSG_PEEK))
117         {
118             skb=skb_dequeue(&sk->receive_queue);
119             if(skb!=NULL)
120                     skb->users++;
121             else
122                     goto restart;     /* Avoid race if someone beats us to the data */
123         }
124         else
125         {
126             cli();
127             skb=skb_peek(&sk->receive_queue);
128             if(skb!=NULL)
129                     skb->users++;
130             restore_flags(intflags);
131             if(skb==NULL)      /* shouldn't happen but .. */
132                     *err=-EAGAIN;
```

```
133         }
134         return skb;
135 }
```

skb_recv_datagram 函数被 UDP 协议中 udp_recvfrom 函数调用从接收队列中取数据包。这个函数虽然较长，但实现思想非常简单：查看套接字接收队列中是否有数据包，如有，则直接返回该数据包，否则睡眠等待。当然在进行睡眠等待之前必须检查等待的必要性，这些检查包括：

1〉 套接字是否已经被关闭接收通道，对于这种情况，盲目等待是不可取的。此时调用 release_sock 函数从可能的其他缓存队列中转移数据包（实际上调用 release_sock 函数对使用 UDP 协议的套接字接收队列不会造成任何影响，因为 UDP 协议根本没有使用 back_log 暂存队列），并直接返回 NULL。

2〉 套接字在处理过程中是否发生错误，如果发生错误，则首先需要处理这些错误，此时也返回。

3〉 如果 noblock 标志位被设置，则调用者要求不进行睡眠等待，当然也就不能进行睡眠等待。

4〉 对 SOCK_SEQPACKET 类型套接字，如果套接字当前状态不是 TCP_ESTABLISHED,则也返回，并设置连接错误标志。

对于 SOCK_SEQPACKET 类型套接字，此处需要简单介绍一下，这个套接字类如同 SOCK_STREAM 套接字类型一样，提供面向连接的可靠性数据传输。二者的区别在于 SOCK_SEQPACKET 是面向报文的，而 SOCK_STREAM 是面向流的。所谓面向报文指数据包之间是独立的记录，接收端在读取时不可跨越数据包进行数据读取，所以在对数据包的操作上类似于 UDP 协议（如果一个数据包中数据没有读取完，则数据包中剩余的数据很可能被丢弃）。而面向流是指数据包只是作为数据传输的一种方式，所有包中的数据是连续的，可以跨越包进行数据读取，一个数据包中数据可以分几次进行读取。总之，对于 SOCK_SEQPACKET 类型套接字的处理，当考虑两台主机之间数据包交换时，我们将其类比于 TCP 协议，其提供可靠性数据传输，TCP 协议使用的那一套对数据进行编号和应答机制，它也使用；但是当数据包在发送或者被远端接收后，应用程序进行读取时，其操作的方式我们将其类比于 UDP 协议，其处理上是以包为边界的。这是 SOCK_SEQPACKET 类型套接字应该工作的方式，但是实际在实现上究竟对于 SOCK_SEQPACKET 这个套接字类型如何处理是实现相关的。如本版本网络代码实现 SOCK_SEQPACKET 类型在数据包处理上完全等同于 SOCK_STREAM，我们可以从 inet_create 函数实现看出：

```
//net/inet/af_inet.c ---inet_create
469             case SOCK_STREAM:
470             case SOCK_SEQPACKET:
471                 if (protocol && protocol != IPPROTO_TCP)
472                 {
473                     kfree_s((void *)sk, sizeof(*sk));
474                     return(-EPROTONOSUPPORT);
475                 }
476                 protocol = IPPROTO_TCP;
477                 sk->no_check = TCP_NO_CHECK;
478                 prot = &tcp_prot;
```

```
479                     break;
```

inet_create 函数将 SOCK_STREAM, SOCK_SEQPACKET 套接字类型都用 TCP 协议实现进行处理。而且在内核网络栈代码的其他地方也未发现对二者处理上的区别。在 skb_recv_datagram 函数中突然冒出对 SOCK_SEQPACKET 类型的特殊“照顾”倒是令人诧异。

90-110 行代码进行睡眠等待以及被唤醒后对唤醒原因的检查和对是否发生错误的检查，这写代码都约定俗成了。当我们跳出 57 行的 while 循环时，就表示接收队列中已经存在数据包供读取了。116-123 行代码处理正常读取的情况，124-133 行代码处理预先读取的情况。最后返回表示数据包的 sk_buff 结构。

```
136  void skb_free_datagram(struct sk_buff *skb)
137  {
138          unsigned long flags;

139          save_flags(flags);
140          cli();
141          skb->users--;
142          if(skb->users>0)
143          {
144                  restore_flags(flags);
145                  return;
146          }
147          /* See if it needs destroying */
148          if(!skb->next && !skb->prev)       /* Been dequeued by someone - ie it's read */
149                  kfree_skb(skb,FREE_READ);
150          restore_flags(flags);
151  }
```

skb_free_datagram 函数释放一个数据包，141 行递减用户计输，每个使用该数据包的进程都回增加该 sk_buff 结构的 users 字段，一旦该字段为 0，表示这是一个游离的数据包，可以进行释放，否则表示还有进程在使用该数据包，此时不可进行释放，直接返回。148 行检查数据包是否仍然处于系统某个队列中，如果数据包还被挂接在系统队列中，也不可对其进行释放。否则调用 kfree_skb 函数释放数据包所占用的内存空间。

```
152  void skb_copy_datagram(struct sk_buff *skb, int offset, char *to, int size)
153  {
154          /* We will know all about the fraglist options to allow >4K receives
155              but not this release */
156          memcpy_tofs(to,skb->h.raw+offset,size);
157  }
```

skb_copy_datagram 函数被 udp_recvfrom 函数调用，将内核缓冲区中数据复制到用户缓冲区中。

```
158 /*
159  *    Datagram select: Again totally generic. Moved from udp.c
160  *    Now does seqpacket.
161  */

162 int datagram_select(struct sock *sk, int sel_type, select_table *wait)
163 {
164         select_wait(sk->sleep, wait);
165         switch(sel_type)
166         {
167                 case SEL_IN:
168                         if (sk->type==SOCK_SEQPACKET && sk->state==TCP_CLOSE)
169                         {
170                                 /* Connection closed: Wake up */
171                                 return(1);
172                         }
173                         if (skb_peek(&sk->receive_queue) != NULL || sk->err != 0)
174                         {       /* This appears to be consistent
175                                    with other stacks */
176                                 return(1);
177                         }
178                         return(0);

179                 case SEL_OUT:
180                         if (sk->type==SOCK_SEQPACKET && sk->state==TCP_SYN_SENT)
181                         {
182                                 /* Connection still in progress */
183                                 return(0);
184                         }
185                         if (sk->prot && sk->prot->wspace(sk) >= MIN_WRITE_SPACE)
186                         {
187                                 return(1);
188                         }
189                         if (sk->prot==NULL && sk->sndbuf-sk->wmem_alloc >=
                                                        MIN_WRITE_SPACE)
190                         {
191                                 return(1);
192                         }
193                         return(0);

194                 case SEL_EX:
195                         if (sk->err)
196                                 return(1); /* Socket has gone into error state (eg icmp error) */
```

```
197                     return(0);
198             }
199             return(0);
200     }
```

datagram_select 函数的作用如同 tcp_select 函数，我们将这类函数称为信息查询函数，如查询内核发送缓冲区当前可用空间大小，查询当前是否存在可接收的连接请求，如果读者对应 socket 应用程序编程较熟的话，那么这些传输层 select 函数就是系统调用 select 函数的底层实现。

从函数实现上来看，分为三种查询情况：

1〉 对当前可接收状态进行查询：查询是否存在可接收的数据。对于侦听套接字而言，就表示是否有连接请求到达。注意 SOCK_SEQPACKET 不同于 SOCK_STREAM 的地方在于内核和应用层接口对数据处理上，在内核处理上同于 SOCK_STREAM，使用 TCP 协议进行连接和交互数据。

2〉 对当前可发送状态进行查询，主要通过查询当前发送缓冲可用空间大小是否满足发送一个数据包。

3〉 对当前错误情况进行查询，一旦套接字在其处理过程中出现错误，则将错误保存在 sock 结构 err 字段，一般在用户应用程序请求套接字某种操作时，会返回这个错误，当然也可如同此处进行明确查询，返回可能发生的错误状态。

datagram.c 文件在早期版本中是不存在的，其实现的功能直接内嵌于 UDP 协议实现文件中，随着网络栈代码的演化，这些可以独立出来的功能逐渐被分离成独立的文件。sk_buff.c 文件就是一个典型，这个文件专门用于处理 sk_buff 相关功能如 sk_buff 结构分配和释放等。早期代码这些分配和释放是随时随地完成的。随着代码层次性逐渐清晰，这些代码也逐渐分离出来作为专门文件形式而存在。当然这在一定程度上增加了代码分析的复杂度，有时为了理解一个函数，我们不得不查阅好几个文件，不像早期版本，所有要实现的功能均“当地”进行实现，不需要调用其他包装函数。这也是本书选择 1.2.13 这个版本作为分析对象的主要原因，虽然这个版本代码也开始进行封装，但程度不深，而且从整体文件结构上还未进行细分（IP 协议，TCP 协议等实现文件都还在一个文件夹中），便于我们集中进行分析。

2.11 net/inet/icmp.c 文件

将 ICMP 协议放在此处进行分析，即认为 ICMP 协议从属于传输层协议，此话一出，某些“专家”就要跳出来，大发一通“危言耸论”了。单从协议角度看，我们一般将 ICMP 协议放在网络层讨论，即与 IP 协议同层，但实际上 ICMP 数据是作为 IP 协议数据负载形式存在的。所以从实现角度上，ICMP 协议工作在 IP 协议之上，但其又不与 TCP 协议或者 UDP 协议同级，而是工作在他们的下一级，即一般 ICMP 模块在处理完后，还需要进一步调用 TCP 协议或者 UDP 协议模块进行处理。所以在 ICMP 协议层次归属上确实很难决定将其到底是归入到网络层还是传输层。不过既然其工作在 IP 模块之上，当然在介绍 IP 协议实现之前，先介绍 ICMP 协议实现。ICMP 协议的目的并非是为了传送数据，而是用于通报错误或者探测远端主机信息。有关 ICMP 协议首部格式以及各种 ICMP 类型在本书第一章中已有介绍，此处不再讨论，我们直接进入对 ICMP 协议实现文件的分析。

```
1     /*
2      * INET         An implementation of the TCP/IP protocol suite for the LINUX
```

```
3     *        operating system.  INET is implemented using the  BSD Socket
4     *        interface as the means of communication with the user level.
5     *
6     *        Internet Control Message Protocol (ICMP)
7     *
8     * Version:  @(#)icmp.c  1.0.11  06/02/93
9     *
10    * Authors:  Ross Biro, <bir7@leland.Stanford.Edu>
11    *        Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
12    *        Mark Evans, <evansmp@uhura.aston.ac.uk>
13    *        Alan Cox, <gw4pts@gw4pts.ampr.org>
14    *        Stefan Becker, <stefanb@yello.ping.de>
15    *
16    * Fixes:
17    *        Alan Cox:  Generic queue usage.
18    *        Gerhard Koerting:  ICMP addressing corrected
19    *        Alan Cox:  Use tos/ttl settings
20    *        Alan Cox:  Protocol violations
21    *        Alan Cox:  SNMP Statistics
22    *        Alan Cox:  Routing errors
23    *        Alan Cox:  Changes for newer routing code
24    *        Alan Cox:  Removed old debugging junk
25    *        Alan Cox:  Fixed the ICMP error status of net/host unreachable
26    *  Gerhard Koerting  :  Fixed broadcast ping properly
27    *        Ulrich Kunitz  :  Fixed ICMP timestamp reply
28    *        A.N.Kuznetsov  :  Multihoming fixes.
29    *        Laco Rusnak  :  Multihoming fixes.
30    *        Alan Cox:  Tightened up icmp_send().
31    *        Alan Cox:  Multicasts.
32    *        Stefan Becker  :  ICMP redirects in icmp_send().
33    *
34    *
35    *
36    *        This program is free software; you can redistribute it and/or
37    *        modify it under the terms of the GNU General Public License
38    *        as published by the Free Software Foundation; either version
39    *        2 of the License, or (at your option) any later version.
40    */
41   #include <linux/types.h>
42   #include <linux/sched.h>
43   #include <linux/kernel.h>
44   #include <linux/fcntl.h>
45   #include <linux/socket.h>
46   #include <linux/in.h>
```

```
47  #include <linux/inet.h>
48  #include <linux/netdevice.h>
49  #include <linux/string.h>
50  #include "snmp.h"
51  #include "ip.h"
52  #include "route.h"
53  #include "protocol.h"
54  #include "icmp.h"
55  #include "tcp.h"
56  #include "snmp.h"
57  #include <linux/skbuff.h>
58  #include "sock.h"
59  #include <linux/errno.h>
60  #include <linux/timer.h>
61  #include <asm/system.h>
62  #include <asm/segment.h>


63  #define min(a,b)    ((a)<(b)?(a):(b))


64  /*
65   *  Statistics
66   */

67  struct icmp_mib     icmp_statistics={0,};
```

如同 UDP，TCP 协议实现文件一样，ICMP 实现文件在开始处也定义了一个信息统计结构。类似这样的结构都定义在 snmp.h 文件中。我们对这样的统计信息不是很关心。

```
68  /* An array of errno for error messages from dest unreach. */
69  struct icmp_err icmp_err_convert[] = {
70      { ENETUNREACH,  0 },  /*    ICMP_NET_UNREACH      */
71      { EHOSTUNREACH,     0 },  /*    ICMP_HOST_UNREACH     */
72      { ENOPROTOOPT,  1 },  /*    ICMP_PROT_UNREACH     */
73      { ECONNREFUSED, 1 },  /*    ICMP_PORT_UNREACH     */
74      { EOPNOTSUPP,       0 },  /*    ICMP_FRAG_NEEDED      */
75      { EOPNOTSUPP,       0 },  /*    ICMP_SR_FAILED        */
76      { ENETUNREACH,  1 },  /*    ICMP_NET_UNKNOWN      */
77      { EHOSTDOWN,        1 },  /*    ICMP_HOST_UNKNOWN     */
78      { ENONET,       1 },  /*    ICMP_HOST_ISOLATED    */
79      { ENETUNREACH,  1 },  /*    ICMP_NET_ANO          */
80      { EHOSTUNREACH,     1 },  /*    ICMP_HOST_ANO         */
81      { EOPNOTSUPP,       0 },  /*    ICMP_NET_UNR_TOS      */
82      { EOPNOTSUPP,       0 }   /*    ICMP_HOST_UNR_TOS     */
```

```
83  };
```

69-83 行代码定义了 ICMP 各种错误类型或者说是错误代码，icmp_err 结构定义在 include/linux/icmp.h 中，如下：

```
//include/linux/icmp.h
66  struct icmp_err {
67      int             errno;
68      unsigned        fatal:1;
69  };
```

第二个字段是一个 1bit 的标志位，表示对应的错误是否可恢复，或者说是内核网络实现代码在接收到对应错误时是否继续处理套接字，还是将错误返回并释放套接字。以上 69-83 行代码定义了一系列常见错误及其是否为 fatal 的标志位。

```
84  /*
85   *  Send an ICMP message in response to a situation
86   *
87   *  Fixme: Fragment handling is wrong really.
88   */

89  void icmp_send(struct sk_buff *skb_in, int type, int code, unsigned long info, struct device *dev)
90  {
```

icmp_send 函数被 IP 模块实现代码调用最多，例如当 IP 模块发现无法找到一个有效的路由选项发送该数据包时，就会调用该函数回复一个 ICMP 错误通报数据包。例如下调用原型：

```
//net/inet/ip.c
1122        rt = ip_rt_route(iph->daddr, NULL, NULL);
1123        if (rt == NULL)
1124        {
1125            /*
1126             *  Tell the sender its packet cannot be delivered. Again
1127             *  ICMP is screened later.
1128             */
1129            icmp_send(skb, ICMP_DEST_UNREACH, ICMP_NET_UNREACH, 0, dev);
1130            return;
1131        }
```

1122 行调用 ip_rt_route 函数查询路由表获取可发送该数据包的路由表项，如果没有找到，则表示数据包无法发送，此时调用 icmp_send 回复一个网络不可达错误通报数据包。注意其中参数设置情况，第一个参数是引起错误的原数据包，其他参数含义易见（info 参数表示自定义信息，一般不使用，简单设置为 0 即可）。

```
91          struct sk_buff *skb;
92          struct iphdr *iph;
```

```
93      int offset;
94      struct icmphdr *icmph;
95      int len;
96      struct device *ndev=NULL;   /* Make this =dev to force replies on the same interface */
97      unsigned long our_addr;
98      int atype;

99      /*
100       *    Find the original IP header.
101       */

102     iph = (struct iphdr *) (skb_in->data + dev->hard_header_len);

103     /*
104       *    No replies to MAC multicast
105       */

106     if(skb_in->pkt_type!=PACKET_HOST)
107           return;
```

如果数据包目的地址不是指向本地的单播地址，则不产生 ICMP 错误通报数据包。对于数据包目的地址类型主要定义又如下常量：

```
//include/linux/sk_buff.h
64  #define PACKET_HOST             0       /* To us */
65  #define PACKET_BROADCAST        1
66  #define PACKET_MULTICAST        2
67  #define PACKET_OTHERHOST        3       /* Unmatched promiscuous */
```

网口驱动程序在调用 alloc_skb 函数分配一个 sk_buff 结构封装数据时默认将 pkt_type 字段设置为 PACKET_HOST。该字段在 alloc_skb 中被初始化为 PACKET_HOST 后一直未曾改变，读者可以将该字段的含义理解为链路层目的地址类型，而非网络层目的地址类型。真正网络层地址的检查是用以下 111 行调用 ip_chk_addr 函数完成的。

```
108     /*
109       *    No replies to IP multicasting
110       */

111     atype=ip_chk_addr(iph->daddr);
112     if(atype==IS_BROADCAST || IN_MULTICAST(iph->daddr))
113           return;
```

111 行调用 ip_chk_addr 检查原数据包最终目的地址类型，如果是一个多播或者广播地址，则不满足产生 ICMP 数据包的条件，直接返回到调用者。由于 ICMP 错误报文产生的条件请参见函数后的说明。

```
114     /*
115      *  Only reply to first fragment.
116      */

117     if(ntohs(iph->frag_off)&IP_OFFSET)
118         return;
```

117 行代码检查是否是第一个数据包分片，注意这种情况也包含了对单分片数据包（即未分片）情况的检查。ICMP 错误报文只对第一个分片产生。

```
119     /*
120      *  We must NEVER NEVER send an ICMP error to an ICMP error message
121      */

122     if(type==ICMP_DEST_UNREACH||type==ICMP_REDIRECT||
                type==ICMP_SOURCE_QUENCH||type==ICMP_TIME_EXCEEDED)
123     {
```

122 行检查原数据包本身是否是一个 ICMP 错误报文，对于这种情况，不可产生另一个 ICMP 错误报文，否则就陷入死循环了。135 行是对原 ICMP 错误报文错误值的进一步检查。

```
124         /*
125          *  Is the original packet an ICMP packet?
126          */

127         if(iph->protocol==IPPROTO_ICMP)
128         {
129             icmph = (struct icmphdr *) ((char *) iph +
130                                                 4 * iph->ihl);
131             /*
132              *  Check for ICMP error packets (Must never reply to
133              *  an ICMP error).
134              */

135             if (icmph->type == ICMP_DEST_UNREACH ||
136                 icmph->type == ICMP_SOURCE_QUENCH ||
137                 icmph->type == ICMP_REDIRECT ||
138                 icmph->type == ICMP_TIME_EXCEEDED ||
139                 icmph->type == ICMP_PARAMETERPROB)
140                 return;
141         }
142     }
143     icmp_statistics.IcmpOutMsgs++;
```

```
144     /*
145      *  This needs a tidy.
146      */
```

经过了以上检查，现在可以安全的发送一个 ICMP 错误通报数据包了，不过首先我们需要更新一下 ICMP 错误报文统计信息。icmp_statistics 变量定义了 icmp.c 文件开始处。

```
147     switch(type)
148     {
149         case ICMP_DEST_UNREACH:
150             icmp_statistics.IcmpOutDestUnreachs++;
151             break;
152         case ICMP_SOURCE_QUENCH:
153             icmp_statistics.IcmpOutSrcQuenchs++;
154             break;
155         case ICMP_REDIRECT:
156             icmp_statistics.IcmpOutRedirects++;
157             break;
158         case ICMP_ECHO:
159             icmp_statistics.IcmpOutEchos++;
160             break;
161         case ICMP_ECHOREPLY:
162             icmp_statistics.IcmpOutEchoReps++;
163             break;
164         case ICMP_TIME_EXCEEDED:
165             icmp_statistics.IcmpOutTimeExcds++;
166             break;
167         case ICMP_PARAMETERPROB:
168             icmp_statistics.IcmpOutParmProbs++;
169             break;
170         case ICMP_TIMESTAMP:
171             icmp_statistics.IcmpOutTimestamps++;
172             break;
173         case ICMP_TIMESTAMPREPLY:
174             icmp_statistics.IcmpOutTimestampReps++;
175             break;
176         case ICMP_ADDRESS:
177             icmp_statistics.IcmpOutAddrMasks++;
178             break;
179         case ICMP_ADDRESSREPLY:
180             icmp_statistics.IcmpOutAddrMaskReps++;
181             break;
182     }
```

在更新了统计信息后，下面就创建一个 ICMP 报文进行回复。如下这段代码实现思想非常简单，就是创建一个 ICMP 协议数据包，这包括 MAC，IP 首部创建，以及 ICMP 首部创建，因为 ICMP 报文是使用 IP 协议发送的。

```
183         /*
184          *  Get some memory for the reply.
185          */

186         len = dev->hard_header_len + sizeof(struct iphdr) + sizeof(struct icmphdr) +
187               sizeof(struct iphdr) + 32;/* amount of header to return */

188         skb = (struct sk_buff *) alloc_skb(len, GFP_ATOMIC);
189         if (skb == NULL)
190         {
191               icmp_statistics.IcmpOutErrors++;
192               return;
193         }
194         skb->free = 1;

195         /*
196          *  Build Layer 2-3 headers for message back to source.
197          */

198         our_addr = dev->pa_addr;
199         if (iph->daddr != our_addr && ip_chk_addr(iph->daddr) == IS_MYADDR)
200               our_addr = iph->daddr;
```

dev 参数表示接收原数据包的网络设备，每个网络设备都会分配一个网络地址，如果原数据包目的地址不是接收该数据包的网络设备对应的地址，但是原数据包目的地址确实属于本机，那么表示这台主机有多个 IP 地址（可能有多块网卡，而原数据包目的地址是其他网卡设备的 IP 地址），既然如此，我们在回复 ICMP 报文时，本地地址就设置为原数据包中使用的目的地址，以保持一致。

```
201         offset = ip_build_header(skb, our_addr, iph->saddr,
202                               &ndev, IPPROTO_ICMP, NULL, len,
203                               skb_in->ip_hdr->tos,255);
```

ip_build_header 函数定义在 ip.c 文件中，用于创建 MAC，IP 首部，返回两个首部的总长度。如果返回值为负值，就表示创建失败，此时 204-210 行代码更新统计信息，释放原数据包。

```
204         if (offset < 0)
205         {
206               icmp_statistics.IcmpOutErrors++;
207               skb->sk = NULL;
208               kfree_skb(skb, FREE_READ);
```

```
209             return;
210         }

211         /*
212          *  Re-adjust length according to actual IP header size.
213          */

214         skb->len = offset + sizeof(struct icmphdr) + sizeof(struct iphdr) + 8;

215         /*
216          *  Fill in the frame
217          */

218         icmph = (struct icmphdr *) (skb->data + offset);
219         icmph->type = type;
220         icmph->code = code;
221         icmph->checksum = 0;
222         icmph->un.gateway = info;   /* This might not be meant for
223                                        this form of the union but it will
224                                        be right anyway */
```

218-222 行代码创建 ICMP 首部，这个首部的创建较为简单，因为各字段值（最重要是 type，code 字段）都已经确定。

```
225         memcpy(icmph + 1, iph, sizeof(struct iphdr) + 8);
```

225 行代码表示 ICMP 错误通报数据包中 ICMP 数据负载为原数据包 IP 首部加上传输层 8 个字节。这样从传输层 8 个字节可以获取对应的端口号，从而通知相关上层应用程序进程。

```
226         icmph->checksum = ip_compute_csum((unsigned char *)icmph,
227                                           sizeof(struct icmphdr) + sizeof(struct iphdr) + 8);

228         /*
229          *  Send it and free it once sent.
230          */
231         ip_queue_xmit(NULL, ndev, skb, 1);
```

226 行代码在计算 ICMP 校验值后，直接调用（231 行）ip_queue_xmit 函数将数据包送往下层进行发送。ip_queue_xmit 函数定义在 IP 模块，使提供给传输层的接口函数，传输层在处理完毕本层工作后，调用该函数进入网络层的处理。TCP，UDP 协议都使用该函数（通过 prot->queue_xmit 函数指针调用）将数据包发往网络层进行处理。

```
232 }//end of icmp_send
```

icmp_send 函数被调用发送一个 ICMP 错误通报数据包。ICMP 错误通报数据包一般在接收到远端一个不合法的数据包后产生的，用于通知远端发生错误的原因，例如远端发送的一个数据包中远端端口不合法，则远端将产生一个端口不可达 ICMP 错误通报数据包给本地，本地在接收到这个数据包响应的作出响应，如停止发送到不可达端口的数据包，但远端并非在任何错误的情况下都会回复这样一个 ICMP 数据包，如接收到的数据包满足如下条件之一，不可产生 ICMP 错误通报数据包：

1〉 一个 ICMP 错误通报数据包。
2〉 数据包远端地址是一个广播地址或者是一个多播地址。
3〉 链路广播数据包（MAC 远端地址设置为全 1）。
4〉 分片数据包中非第一个数据包（换句话说，只对所有分片中第一个分片产生 ICMP 错误通报数据包，原因很简单，只有第一个分片才包含有传输层协议头部）。
5〉 数据包源端地址非单播地址，即全 0 地址，回环地址，多播或广播地址。

满足以上 5 个条件中之一，即不可产生一个 ICMP 错误通报数据包。
icmp_send 函数实现代码虽然较长，但实现思想非常简单，首先判断引起错误的数据包是否满足以上 5 种条件之一，如果满足，则直接返回调用这，不产生 ICMP 错误通报数据包，否则更新本地 ICMP 统计信息后，构建一个 ICMP 错误通报数据包并直接调用 ip_queue_xmit 函数发送个 IP 模块进行处理从而将这个 ICMP 错误通报数据包发送给引起该错误的原数据包的发送端。具体情况请参见前文分析。

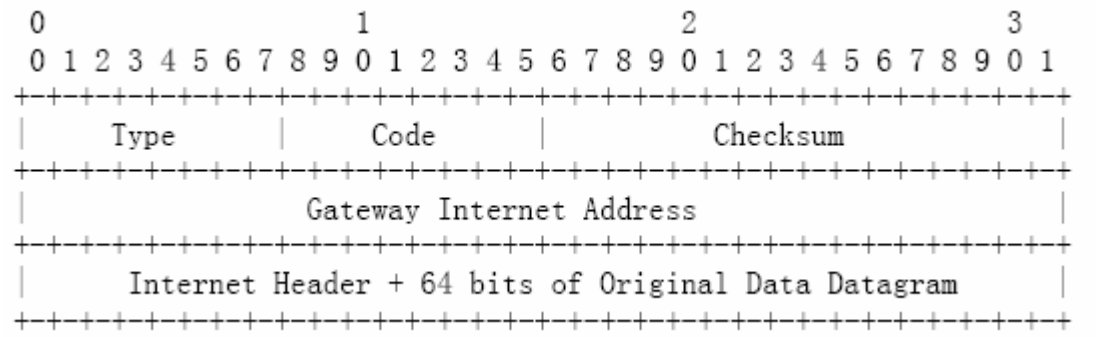
```
233 /*
234  *  Handle ICMP_UNREACH and ICMP_QUENCH.
235  */

236 static void icmp_unreach(struct icmphdr *icmph, struct sk_buff *skb)
237 {
238     struct inet_protocol *ipprot;
239     struct iphdr *iph;
240     unsigned char hash;
241     int err;

242     err = (icmph->type << 8) | icmph->code;
243     iph = (struct iphdr *) (icmph + 1);

244     switch(icmph->code & 7)
245     {
246         case ICMP_NET_UNREACH:
247             break;
248         case ICMP_HOST_UNREACH:
249             break;
250         case ICMP_PROT_UNREACH:
251             printk("ICMP: %s:%d: protocol unreachable.\n",
252                 in_ntoa(iph->daddr), ntohs(iph->protocol));
```

```
253                 break;
254             case ICMP_PORT_UNREACH:
255                 break;
256             case ICMP_FRAG_NEEDED:
257                 printk("ICMP: %s: fragmentation needed and DF set.\n",
258                                         in_ntoa(iph->daddr));
259                 break;
260             case ICMP_SR_FAILED:
261                 printk("ICMP: %s: Source Route Failed.\n", in_ntoa(iph->daddr));
262                 break;
263             default:
264                 break;
265         }

266         /*
267          *  Get the protocol(s).
268          */

269         hash = iph->protocol & (MAX_INET_PROTOS -1);

270         /*
271          *  This can't change while we are doing it.
272          */

273         ipprot = (struct inet_protocol *) inet_protos[hash];
274         while(ipprot != NULL)
275         {
276             struct inet_protocol *nextip;

277             nextip = (struct inet_protocol *) ipprot->next;

278             /*
279              *  Pass it off to everyone who wants it.
280              */
281             if (iph->protocol == ipprot->protocol && ipprot->err_handler)
282             {
283                 ipprot->err_handler(err, (unsigned char *)(icmph + 1),
284                                     iph->daddr, iph->saddr, ipprot);
285             }

286             ipprot = nextip;
287         }
288         kfree_skb(skb, FREE_READ);
289 }
```

icmp_unreach 函数处理的 ICMP 错误类型比较广，这点可以从下文中介绍的 icmp_rcv 总入口函数看出。此处我们就事论事，单方面进行该函数的分析。函数实现思想较为简单，对相关错误打印错误信息后，调用传输层协议错误处理函数进行处理（tcp_err, udp_err）。注意 242 行代码 err 变量被初始化为 ICMP 类型和代码值。最低 8bit 为代码（code）值，次低 8bit 为类型（type）值。243 行代码初始化 iph 变量指向原数据包中 IP 首部以及传输层 8 字节数据，从而传输层错误处理函数可以由此获知上层对应应用进程。269 行代码计算传输层协议在 inet_protos 数组中的位置。inet_protos 一个元素类型为 inet_protocol 结构的数组，数组中每个元素对应一个传输层协议，协议在数组中的位置由协议编号索引，如 TCP 协议对应索引号为 6，UDP 为 17。274-287 行代码完成对传输层协议的遍历，调用协议对应错误处理函数进行返回错误的处理。注意 inet_protos 数组中每个元素对应一个协议，所以 274 行在一次循环后会退出，但这种编码方式提高了程序的可扩展性。

```
290 /*
291  *  Handle ICMP_REDIRECT.
292  */

293 static void icmp_redirect(struct icmphdr *icmph, struct sk_buff *skb,
294         struct device *dev, unsigned long source)
295 {
```

icmp_redirect 函数处理重定向报文，有关重定向的说明，请看下文函数后的说明。
ICMP 重定向报文格式如下图所示：



其中类型（Type），代码（Code），校验和（Checksum）以及原数据包 IP 首部加上 8 字节传输层数据所有的 ICMP 数据包类型都具有这些字段，对于重定向报文来说，多出一个网关 IP 地址字段，这个字段表示更优路由器的 IP 地址。

```
296         struct rtable *rt;
297         struct iphdr *iph;
298         unsigned long ip;

299         /*
300          *  Get the copied header of the packet that caused the redirect
301          */
```

302 iph = (struct iphdr *) (icmph + 1);
iph 变量被初始化为原数据包 IP 首部。所谓原数据包即本地原先发送的数据包，该数据包引起远端（或中间路由器）回复了一个 ICMP 重定向报文，这个 ICMP 重定向报文就是我们当前正在处理的数据包，这个 ICMP 重定向数据包在 ICMP 首部之后包含了原先本地发送数据包的 IP 首部以及 8 字节传输层数据。此处 iph 变量就是被初始化为指向原数据包中 IP 首部。

303 ip = iph->daddr;
ip 字段被初始化为要发送数据包的远端 IP 地址。注意 iph 表示本地发送的数据包的 IP 首部，所以 iph->daddr 表示的是远端 IP 地址（当然是从本地的角度看）。

以下对重定向的类型进行判断，代码支持目的主机和网络重定向，暂不支持服务类型重定向报文。所有重定向报文的类型值为 5，但代码字段可取如下 4 个值，分别表示 4 种重定向报文。

Code 代码取值

0 = Redirect datagrams for the Network.

网络重定向

1 = Redirect datagrams for the Host.

主机重定向

2 = Redirect datagrams for the Type of Service and Network.

服务类型和网络重定向

3 = Redirect datagrams for the Type of Service and Host.

服务类型和主机重定向

在 include/linux/icmp.h 中定义有如下常量：

```
//include/linux/icmp.h
46  /* Codes for REDIRECT. */
47  #define ICMP_REDIR_NET      0   /* Redirect Net             */
48  #define ICMP_REDIR_HOST     1   /* Redirect Host            */
49  #define ICMP_REDIR_NETTOS   2   /* Redirect Net for TOS     */
50  #define ICMP_REDIR_HOSTTOS  3   /* Redirect Host for TOS    */
```

如下代码即根据具体的重定向类型分别进行处理。

```
304     switch(icmph->code & 7)
305     {
306         case ICMP_REDIR_NET:
307             /*
308              *  This causes a problem with subnetted networks. What we should do
309              *  is use ICMP_ADDRESS to get the subnet mask of the problem route
```

```
310                  *    and set both. But we don't..
311                  */
312 #ifdef not_a_good_idea
313                  ip_rt_add((RTF_DYNAMIC | RTF_MODIFIED | RTF_GATEWAY),
314                       ip, 0, icmph->un.gateway, dev,0, 0);
315                  break;
316 #endif
```

对于网络重定向类型，直接添加路由表项，对于划分子网的情况，由于无法获得确切的子网掩码，所以这个添加的表项可能无法正常工作。另外对于使用重定向报文创建或者修改的路有表项都需要设置 M（RTF_MODIFIED）标志位。RTF_DYNAMIC 标志位表示这是一个动态路由表项，动态路由表项是相对于永久表项而言的。动态路由表项会设置定时器进行刷新处理。RTF_GATEWAY 标志位表示添加的是一个网关地址。

```
317             case ICMP_REDIR_HOST:
318                  /*
319                   *    Add better route to host.
320                   *    But first check that the redirect
321                   *    comes from the old gateway..
322                   *    And make sure it's an ok host address
323                   *    (not some confused thing sending our
324                   *    address)
325                   */
326                  rt = ip_rt_route(ip, NULL, NULL);
327                  if (!rt)
328                       break;
329                  if (rt->rt_gateway != source || ip_chk_addr(icmph->un.gateway))
330                       break;
331                  printk("redirect from %s\n", in_ntoa(source));
332                  ip_rt_add((RTF_DYNAMIC | RTF_MODIFIED | RTF_HOST |
                                                          RTF_GATEWAY),
333                       ip, 0, icmph->un.gateway, dev,0, 0);
334                  break;
```

对于主机重定向的处理较为谨慎。首先不可直接创建主机重定向路由表项，而只能更改原有的表项，327 行代码保证了这一点，即如果不存在原表项，则返回。另外发送这个重定向报文的路由器必须是原表项中指定的路由器，换句话说，路由器重定向报文只可将其自身替换掉（替换成一个更好的路由器发送通道），另外所提供的新的路由器不可是本地地址，多播地址或者是广播地址（即 ip_chk_addr 返回值不可为非 0）。通过 329 行检查后，332 行调用 ip_rt_add 对原表项进行修改，将原路由器替换为重定向报文中声明的路由器。注意 RTF_MODIFIED 标志位被设置，表示这是经过 ICMP 重定位数据包修改过的。

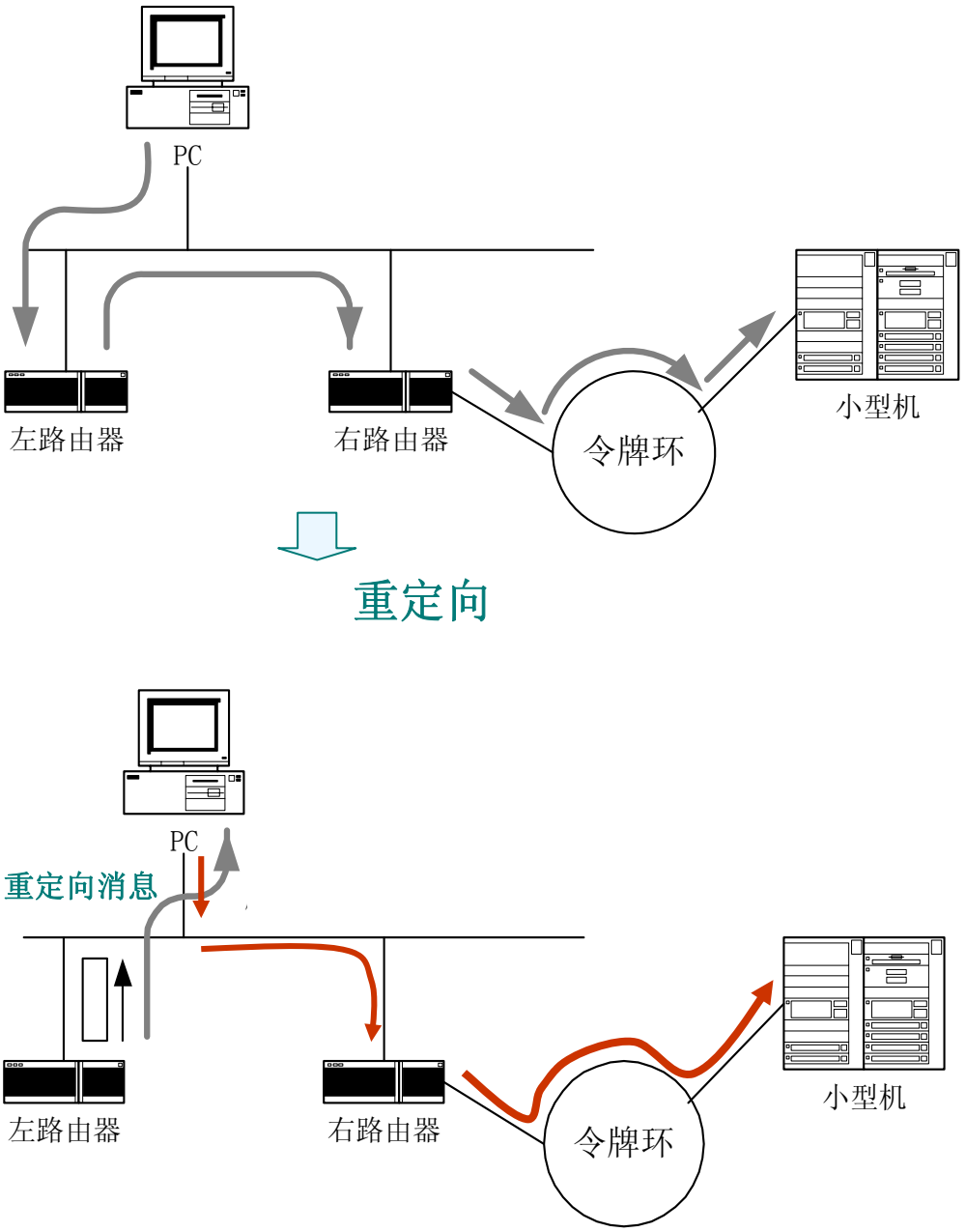
```
335             case ICMP_REDIR_NETTOS:
336             case ICMP_REDIR_HOSTTOS:
```

```
337                 printk("ICMP: cannot handle TOS redirects yet!\n");
338                 break;
```

对于服务类型和网络，以及服务类型和主机重定向报文本版本不支持。即便支持，我们也可以预见其处理方式，对于修改表项操作方式而言是一致的，区别在于发送这些报文时所依据的判断条件。对于这两种重定向报文，考虑的不单是更短的路由途径，还需要考虑服务质量。

```
339             default:
340                 break;
341         }
342         /*
343          *  Discard the original packet
344          */
345         kfree_skb(skb, FREE_READ);
346 }
```

icmp_redirect 函数处理路有重定向 ICMP 数据包。这种数据包是由路由器发送，用于通知本地到达某个网络或主机的一个更好的路由途径。路由器为特定的目的主机发送重定向消息将主机重定向到一个更优的路由器或者告诉主机目的主机实际上是在同一链路上的邻居节点。



如上图所示，PC 机要发送一个数据包到小型机，其原路有表项表示到达小型机需要通过左路有器转发，但左路由器发现被转发的数据包发送接口与接收该数据包的接口是同一个，这就表示这个目的主机要么就在本地链路上，要么下一站路由器在本地链路上，这种情况下，PC 机直接将数据包发送到这个目的主机或者这个“下一站路由器”（下图中即为右路由器）即可。那么此时左路由器就会产生一个 ICMP 重定向报文，通知 PC 机一个更合理的路有表项，PC 机在更新这个表项后，以后发送给小型机的所有数据包都直接发送给右路由器进行转发即可，这样极大的提高了网络利用率以及节省左路由器的资源。

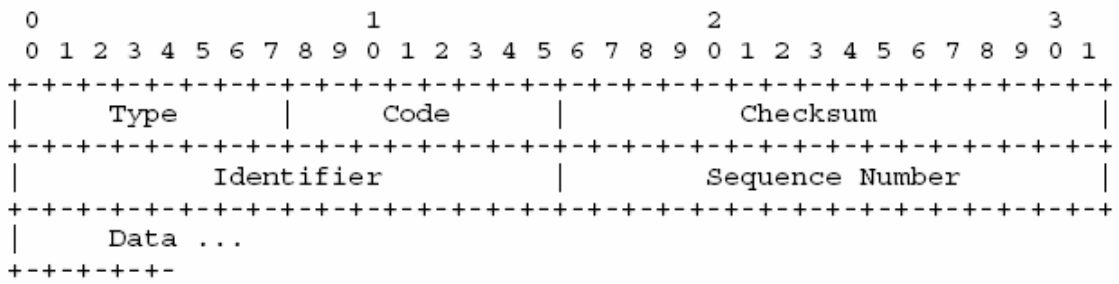
```
347  /*
348   *    Handle ICMP_ECHO ("ping") requests.
349   */
```

```
350 static void icmp_echo(struct icmphdr *icmph, struct sk_buff *skb, struct device *dev,
351         unsigned long saddr, unsigned long daddr, int len,
352         struct options *opt)
353 {
```

本函数实现功能单一，即回复一个 Echo 应答数据包，所以完成的主要工作即进行数据包创建，主要是协议首部创建。函数所传入参数中 saddr 从本地角度看，实际表示的是远端地址，而 daddr 表示的是本地地址。

Echo 请求和应答 ICMP 首部格式相同，不同的只是 Type 类型字段取值，具体格式如下：Code 代码值均设置为 0。

Echo or Echo Reply Message ICMP Echo请求和Echo应答报文格式



Type 类型值

8 for echo message;
Echo请求

0 for echo reply message.
Echo应答

其中 Identifier（标识）和 Sequence Number（序列号）字段用于匹配请求和应答报文。根据以上 ECHO 请求和应答报文 ICMP 首部格式，387-392 行代码应不难理解，并注意到 include/linux/icmp.h 中如下常量定义：

```
//include/linux/icmp.h
19  #define ICMP_ECHOREPLY      0     /* Echo Reply            */
23  #define ICMP_ECHO           8     /* Echo Request          */
```

include/linux/icmp.h 中定义有 ICMP 所有报文类型常量。这个文件在本书第一章中已有说明。

```
354         struct icmphdr *icmphr;
355         struct sk_buff *skb2;
356         struct device *ndev=NULL;
357         int size, offset;

358         icmp_statistics.IcmpOutEchoReps++;
359         icmp_statistics.IcmpOutMsgs++;

360         size = dev->hard_header_len + 64 + len;
```

此处对这个长度字段的赋值需要进行一下说明，dev->hard_header_len 表示 MAC 首部长度，len 表示 Echo 请求数据包中 IP 数据负载长度：包括 ICMP 首部长度和可能的用户数据，Echo 应答数据包必须原样返回这些数据（另外 Echo 请求数据包中 ICMP 首部中标识字段和序列号字段也要原样返回）。我们从包含有 IP 选项的情况考虑（如对 PING 使用-R 选项），此时 IP 首部需要 60 字节，以上 64 字节多出的 4 字节可能计入了尾部 4 字节帧校验序列。

```
361     skb2 = alloc_skb(size, GFP_ATOMIC);

362     if (skb2 == NULL)
363     {
364         icmp_statistics.IcmpOutErrors++;
365         kfree_skb(skb, FREE_READ);
366         return;
367     }
368     skb2->free = 1;

369     /* Build Layer 2-3 headers for message back to source */
370     offset = ip_build_header(skb2, daddr, saddr, &ndev,
371         IPPROTO_ICMP, opt, len, skb->ip_hdr->tos,255);
372     if (offset < 0)
373     {
374         icmp_statistics.IcmpOutErrors++;
375         printk("ICMP: Could not build IP Header for ICMP ECHO Response\n");
376         kfree_skb(skb2,FREE_WRITE);
377         kfree_skb(skb, FREE_READ);
378         return;
379     }

380     /*
381      *  Re-adjust length according to actual IP header size.
382      */

383     skb2->len = offset + len;

384     /*
385      *  Build ICMP_ECHO Response message.
386      */
387     icmphr = (struct icmphdr *) (skb2->data + offset);
388     memcpy((char *) icmphr, (char *) icmph, len);
389     icmphr->type = ICMP_ECHOREPLY;
390     icmphr->code = 0;
391     icmphr->checksum = 0;
392     icmphr->checksum = ip_compute_csum((unsigned char *)icmphr, len);
```

```
393         /*
394          *  Ship it out - free it when done
395          */
396         ip_queue_xmit((struct sock *)NULL, ndev, skb2, 1);

397         /*
398          *  Free the received frame
399          */

400         kfree_skb(skb, FREE_READ);
401 }
```

icmp_echo 函数处理 Echo 请求，PING 应用程序即使用 Echo ICMP 请求数据包探测远端主机的可达性。对于 Echo 请求的处理是在接收到一个远端地址指向本地的数据包后，回复一个 Echo 应答数据包。icmp_echo 完成的功能就是回复一个 Echo 应答数据包。必要的检查实现代码已经在调用该函数的函数（icmp_rcv）中完成了。

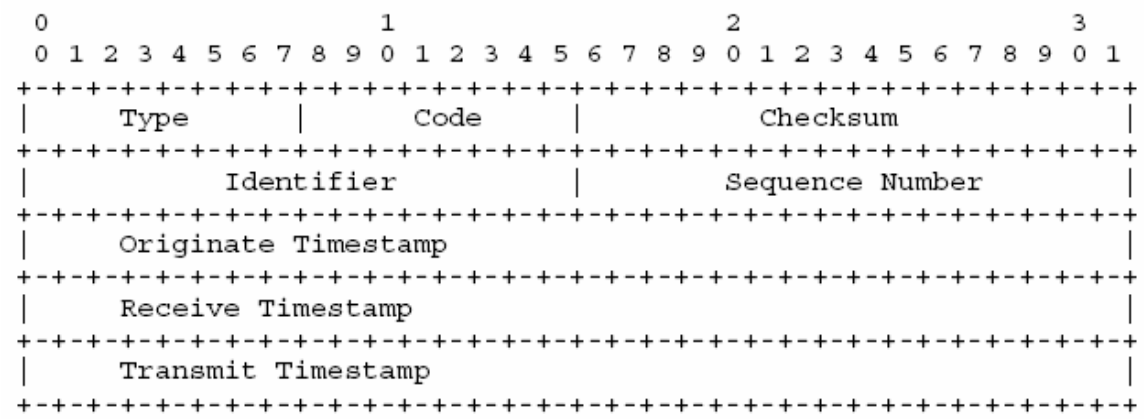
```
402 /*
403  *  Handle ICMP Timestamp requests.
404  */

405 static void icmp_timestamp(struct icmphdr *icmph, struct sk_buff *skb, struct device *dev,
406             unsigned long saddr, unsigned long daddr, int len,
407             struct options *opt)
408 {
```

icmp_timestamp 函数处理 ICMP Timestamp 数据包。首先看一下传入参数的意义。
imcph：时间戳请求数据包 ICMP 首部。
skb：时间戳请求数据包。
dev：时间戳请求数据包接收设备。
saddr：发送请求数据包主机 IP 地址。
daddr：接收请求数据包主机（即本地）的 IP 地址。
len：IP 数据负载长度，包括 ICMP 首部和可能的 ICMP 数据负载长度。一般 ICMP 时间戳请求数据包 IP 负载部分就是一个 ICMP 时间戳格式的首部（格式见下文介绍），长度为 20 字节。
opt：可能的 IP 选项，如无，则为 NULL。

icmp_timestamp 处理 ICMP 时间戳（请求）数据包。ICMP 时间戳请求和应答数据包具有相同的 ICMP 首部格式，如下图所示，所不同的是类型（Type）字段值不同，代码字段均设置为 0。

ICMP时间戳（请求）和应答报文格式

Timestamp or Timestamp Reply Message



Type 类型字段取值

13 for timestamp message;

时间戳（请求）

14 for timestamp reply message.

时间戳应答

标识（Identifier）和序列号（Sequence Number）字段匹配请求和应答数据包。发送或者说起源时间戳字段（Originate Timestamp）由发送时间戳请求报文的一方设置为发送数据包时发送端主机的从当天零时计起的时间值（以毫秒计，以下同）。接收时间戳字段（Receive Timestamp）由接收方设置为接收该请求报文时接收方主机从当天零时计起的时间值。而回复时间戳字段（Transmit Timestamp）设置为接收方发送（回复）时间戳应答报文时接收方主机从当天零时计起的时间值。一般实现上，将接收时间戳字段和回复时间戳字段设置为相同值，这点很容易理解，如果对接收时间戳进行跟踪花费较大，如果要实现的话，不单是对 ICMP 数据包，对使用 TCP，UDP 协议的数据包都要进行这个时间的纪录，因为对于这个接收时间的处理只有到传输层才被（此时将 ICMP 协议认为是传输层协议）处理，传输层之下的模块中，网卡驱动模块需要进行纪录（暂且不论何时进行纪录才合理），链路层，网络层模块都要对接收时间进行存储。这样需要单独分配一个字段用于保存接收时间戳。而本身 ICMP 时间戳作为一种主机间时间校正的手段使用并不很多，所以以上这种方式有些浪费，况且提供接收时间戳本身的意义并不大，网络波动性本身足可以抵销接收时间戳和回复时间戳之间的差异！所以几乎所有实现中将接收时间戳和回复时间戳设置为相同值并无大碍。

```
409     struct icmphdr *icmphr;
410     struct sk_buff *skb2;
411     int size, offset;
412     unsigned long *timeptr, midtime;
413     struct device *ndev=NULL;

414     if (len != 20)
```

```
415         {
416                 printk(
417                   "ICMP: Size (%d) of ICMP_TIMESTAMP request should be 20!\n",
418                   len);
419                 icmp_statistics.IcmpInErrors++;
420 #if 1
421                         /* correct answers are possible for everything >= 12 */
422                 if (len < 12)
423 #endif
424                     return;
425         }
```

首先检查 ICMP 时间戳选项首部长度是否满足条件，ICMP 时间戳选项首部长度应该为 20 字节，或者至少是 12 字节：即应该包括发送端时间戳字段。如果小于 12 字节长度，则表示数据包格式有误，此时直接返回，如果只是长度不满足 20 字节，但包含了发送端时间戳字段，还可以继续进行处理。

```
426         size = dev->hard_header_len + 84;
```

此处 84 表示 60 字节 IP 首部加上 ICMP20 字节首部加上 4 字节帧校验字段。

```
427         if (! (skb2 = alloc_skb(size, GFP_ATOMIC)))
428         {
429                 skb->sk = NULL;
430                 kfree_skb(skb, FREE_READ);
431                 icmp_statistics.IcmpOutErrors++;
432                 return;
433         }
434         skb2->free = 1;

435 /*
436  *  Build Layer 2-3 headers for message back to source
437  */

438         offset = ip_build_header(skb2, daddr, saddr, &ndev, IPPROTO_ICMP, opt, len,
439                                 skb->ip_hdr->tos, 255);
440         if (offset < 0)
441         {
442                 printk("ICMP: Could not build IP Header for ICMP TIMESTAMP Response\n");
443                 kfree_skb(skb2, FREE_WRITE);
444                 kfree_skb(skb, FREE_READ);
445                 icmp_statistics.IcmpOutErrors++;
446                 return;
447         }
```

```
448         /*
449          *    Re-adjust length according to actual IP header size.
450          */
451         skb2->len = offset + 20;

452         /*
453          *    Build ICMP_TIMESTAMP Response message.
454          */

455         icmphr = (struct icmphdr *) ((char *) (skb2 + 1) + offset);
456         memcpy((char *) icmphr, (char *) icmph, 12);
457         icmphr->type = ICMP_TIMESTAMPREPLY;
458         icmphr->code = icmphr->checksum = 0;

459         /* fill in the current time as ms since midnight UT: */
460         midtime = (xtime.tv_sec % 86400) * 1000 + xtime.tv_usec / 1000;
```

Xtime 变量是在 kernel/sched.c 中定义了一个 timeval 结构类型的全局变量，表示系统当前时间值。

timeval 结构定义在 include/linux/time.h 中，如下所示：

```
//include/linux/time.h
3     struct timeval {
4             long tv_sec;          /* seconds */
5             long tv_usec;   /* microseconds */
6     };
```

460 行代码中 86400 表示一天的秒数（24x60x60=86400），所以 xtime.tv_sec%86400 表示从当天零时起的秒数，乘 1000，即转换为毫秒数，xtime.tv_usec 表示当前秒下走过的微妙书，除以 1000，换算成毫秒数，二者相加即可当天零时起的毫秒数。这个计算出的数值将被下文中 465 行代码使用对 ICMP 首部中接收时间戳和回复时间戳字段进行赋值操作。从而完成时间戳应答报文中 ICMP 首部中最重要两个字段的初始化。456 行代码通过复制接收到的时间戳请求报文 ICMP 首部可以按需原样返回标识和序列号字段值。457 行将类型字段值设置为时间戳应答类型。由此完全完成 ICMP 首部的创建工作。之后就可以调用 ip_queue_xmit 接口函数将该时间戳应答报文发送出去了。

```
461         timeptr = (unsigned long *) (icmphr + 1);
```

461 行代码将 timeptr 指向第一个时间戳即发送端时间戳，这个时间戳的赋值由发送端负责，换句话，本地回复的时间戳应答数据包中可以不管该字段，这里只需对第二，三个时间戳字段进行初始化即可。

```
462         /*
463          *    the originate timestamp (timeptr [0]) is still in the copy:
```

```
464          */
465         timeptr [1] = timeptr [2] = htonl(midtime);

466         icmphr->checksum = ip_compute_csum((unsigned char *) icmphr, 20);

467         /*
468          *    Ship it out - free it when done
469          */

470         ip_queue_xmit((struct sock *) NULL, ndev, skb2, 1);
471         icmp_statistics.IcmpOutTimestampReps++;
472         kfree_skb(skb, FREE_READ);
473 }
```

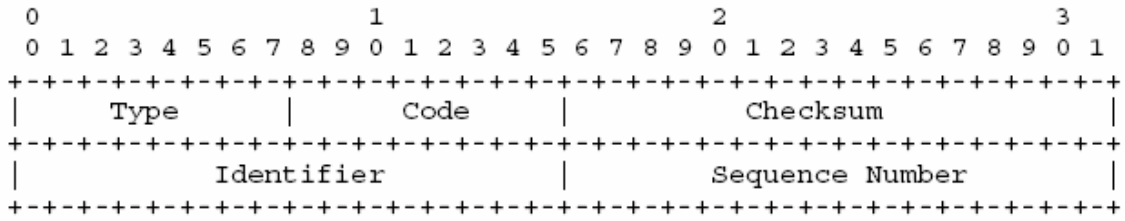
函数最后还对 ICMP 协议统计信息进行了更新，并释放原时间戳请求数据包。在介绍了 TimestampICMP 报文格式的情况下，对于 icmp_timestamp 函数理解应该不成问题。

```
474 /*
475  *    Handle the ICMP INFORMATION REQUEST.
476  */

477 static void icmp_info(struct icmphdr *icmph, struct sk_buff *skb, struct device *dev,
478           unsigned long saddr, unsigned long daddr, int len,
479           struct options *opt)
480 {
481         /* Obsolete */
482         kfree_skb(skb, FREE_READ);
483 }
```

icmp_info 函数处理 ICMP 信息请求（Information Request），报文中 ICMP 首部格式如下图所示。读者可以将此格式与 Echo 请求和应答报文相比较，可以发现 Information 请求和应答报文其实可以算成是 Echo 请求和应答报文，只不过此时没有附带任何用户数据而已！换句话说，Information 请求和应答完全可以用 Echo 请求和应答代替，当然这纯粹是从格式上考虑，ICMP 协议规范（RFC792）定义了这种格式但并未对这种格式的具体用途作出说明，可以有些实现上将此用于某种特殊的目的。然而就本书分析 LINUX1.2.13 对应网络实现代码而言，并没有对此报文格式进行实现，只是将接收到的此格式报文简单丢弃。icmp_info 函数一目了然。

信息请求和应答报文ICMP首部格式

Information Request or Information Reply Message



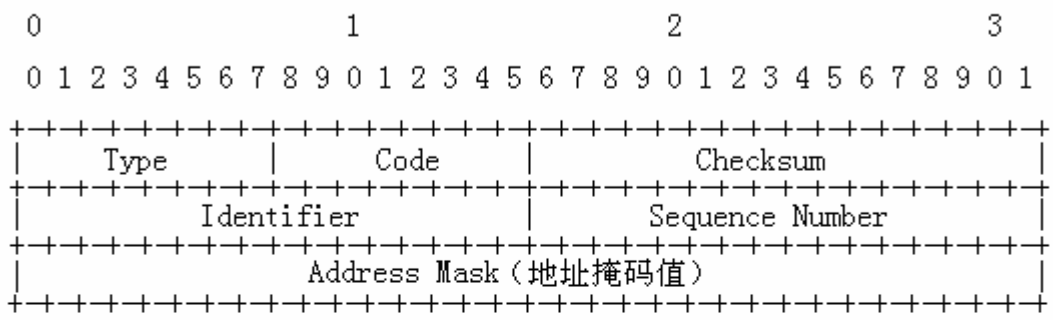
Type	类型字段取值	Code	代码字段取值
15	for information request message; 信息请求消息	0	
16	for information reply message. 信息应答消息		

```
484 /*
485  *    Handle ICMP_ADDRESS_MASK requests.
486  */

487 static void icmp_address(struct icmphdr *icmph, struct sk_buff *skb, struct device *dev,
488           unsigned long saddr, unsigned long daddr, int len,
489           struct options *opt)
490 {
```

icmp_address 函数本身实现比较简单，即返回本地设备（对应接收地址掩码请求报文的网卡设备）IP 地址对应的地址掩码（532 行）。整个函数就是在创建一个 ICMP 地址掩码应答数据包，构建 MAC，IP，ICMP 首部。ICMP 地址掩码请求用于无盘系统在引导过程中获取自己的子网掩码。系统广播它的 ICMP 请求报文（这一过程与无盘系统在引导过程中用 RARP 获取 IP 地址是类似的）。同一链路上主机在接收到该地址掩码请求后，都回复一个地址掩码应答报文。当然无盘系统还可以通过 BOOTP 协议获取子网掩码。ICMP 地址掩码请求报文格式如下图所示。

ICMP地址掩码请求和应答报文格式

Address Mask Request or Address Mask Reply



Type 类型字段取值

17 for Address Mask Request Message

地址掩码请求消息

18 for Address Mask Reply Message

地址掩码应答消息

Code 代码字段取值

0

了解了报文格式，如下代码不再做细致分析。497 行值 64 的由来读者（参照前文其他函数相关介绍）应该可以自行理解了。

```
491     struct icmphdr *icmphr;
492     struct sk_buff *skb2;
493     int size, offset;
494     struct device *ndev=NULL;

495     icmp_statistics.IcmpOutMsgs++;
496     icmp_statistics.IcmpOutAddrMaskReps++;

497     size = dev->hard_header_len + 64 + len;
498     skb2 = alloc_skb(size, GFP_ATOMIC);
499     if (skb2 == NULL)
500     {
501         icmp_statistics.IcmpOutErrors++;
502         kfree_skb(skb, FREE_READ);
503         return;
504     }
505     skb2->free = 1;
```

```
506         /*
507          *    Build Layer 2-3 headers for message back to source
508          */

509         offset = ip_build_header(skb2, daddr, saddr, &ndev,
510                      IPPROTO_ICMP, opt, len, skb->ip_hdr->tos,255);
511         if (offset < 0)
512         {
513             icmp_statistics.IcmpOutErrors++;
514             printk("ICMP: Could not build IP Header for ICMP ADDRESS Response\n");
515             kfree_skb(skb2,FREE_WRITE);
516             kfree_skb(skb, FREE_READ);
517             return;
518         }

519         /*
520          *    Re-adjust length according to actual IP header size.
521          */

522         skb2->len = offset + len;

523         /*
524          *    Build ICMP ADDRESS MASK Response message.
525          */

526         icmphr = (struct icmphdr *) (skb2->data + offset);
527         icmphr->type = ICMP_ADDRESSREPLY;
528         icmphr->code = 0;
529         icmphr->checksum = 0;
530         icmphr->un.echo.id = icmph->un.echo.id;
531         icmphr->un.echo.sequence = icmph->un.echo.sequence;
532         memcpy((char *) (icmphr + 1), (char *) &dev->pa_mask, sizeof(dev->pa_mask));

533         icmphr->checksum = ip_compute_csum((unsigned char *)icmphr, len);

534         /* Ship it out - free it when done */
535         ip_queue_xmit((struct sock *)NULL, ndev, skb2, 1);

536         skb->sk = NULL;
537         kfree_skb(skb, FREE_READ);
538 }
```

注意在回复报文中，标识（Identifier）和序列号（Sequence Number）字段必须原样返回，以便于发送请求报文的一端进行应答匹配。530-531 行代码即对这两个字段执行原样返回的

工作（icmph 参数表示传入的请求报文 ICMP 首部）。

下面我们进入到ICMP协议总入口函数icmp_rcv的分析，icmp_rcv函数的作用类似于tcp_rcv, udp_rcv 函数，当 IP 模块发现上层使用 ICMP 协议时（即 IP 首部中协议字段取值为 1），即调用 icmp_rcv 函数将数据包上传（这种调用通过函数指针的方式，而非硬编码）。icmp_rcv 函数将根据这个接收的 ICMP 报文的具体类型调用其对应的处理函数，如对应 Echo 请求报文，则调用 icmp_echo 函数进行处理，而对于重定向报文，则调用 icmp_redirect 函数进行处理，诸如此类。
而对于 ICMP 报文类型的判断，是通过检查报文 ICMP 首部中类型（Type）字段完成的。根据该函数实现的功能，我们可以将该函数分为两个部分进行分析：其一为数据包合法性检查；其二判断报文类型，调用对应性处理函数。

```
539 /*
540  *  Deal with incoming ICMP packets.
541  */

542 int icmp_rcv(struct sk_buff *skb1, struct device *dev, struct options *opt,
543         unsigned long daddr, unsigned short len,
544         unsigned long saddr, int redo, struct inet_protocol *protocol)
545 {
```

参数说明：
skb1：接收的 ICMP 报文（对应 sk_buff 结构）。
dev：接收该 ICMP 报文的网络设备。
opt：可能的 IP 选项，如无，则为 NULL，本函数不使用该参数。
daddr：该 ICMP 报文的最终目的 IP 地址。
len：IP 负载长度：包括 ICMP 首部以及可能的 ICMP 负载。
saddr：发送该 ICMP 报文的源端 IP 地址。
redo：该参数表示这是一个新接收的数据包（redo=0），还是一个之前被缓存的数据包。
protocol：对于 ICMP 协议而言，该参数指向 icmp_protocol 全局变量。icmp_protocol 变量定义在 net/inet/protocol.c 中，如下：

```
//net/inet/protocol.c
64  static struct inet_protocol icmp_protocol = {
65      icmp_rcv,       /* ICMP handler          */
66      NULL,           /* ICMP never fragments anyway */
67      NULL,           /* ICMP error control    */
68      &udp_protocol,  /* next                  */
69      IPPROTO_ICMP,       /* protocol ID       */
70      0,          /* copy                  */
71      NULL,           /* data                  */
72      "ICMP"          /* name                  */
73  };
```

```
546     struct icmphdr *icmph;
547     unsigned char *buff;

548     /*
549      *  Drop broadcast packets. IP has done a broadcast check and ought one day
550      *  to pass on that information.
551      */

552     icmp_statistics.IcmpInMsgs++;


553     /*
554      *  Grab the packet as an icmp object
555      */

556     buff = skb1->h.raw;
```

IP 模块在提交数据包给上层协议模块处理之前，已经将 skb1->h.raw 设置为指向上层协议首部。如同链路层处理模块将数据包上交给 IP 模块处理之前，将该字段设置为指向 IP 首部一样。

```
557     icmph = (struct icmphdr *) buff;

558     /*
559      *  Validate the packet first
560      */

561     if (ip_compute_csum((unsigned char *) icmph, len))
562     {
563         /* Failed checksum! */
564         icmp_statistics.IcmpInErrors++;
565         printk("ICMP: failed checksum from %s!\n", in_ntoa(saddr));
566         kfree_skb(skb1, FREE_READ);
567         return(0);
568     }

569     /*
570      *  Parse the ICMP message
571      */

572     if (ip_chk_addr(daddr) != IS_MYADDR)
573     {
574         if (icmph->type != ICMP_ECHO)
575         {
576             icmp_statistics.IcmpInErrors++;
```

```
577                 kfree_skb(skb1, FREE_READ);
578                 return(0);
579             }
580             daddr=dev->pa_addr;
581         }
```

572 行代码检查接收到的 ICMP 报文最终目的 IP 地址是否为本地地址，如果不是，表示这个 ICMP 报文不是发送给本机，此时正常的处理应该是简单丢弃该数据包。但 574 行另外对 ICMP Echo 请求报文开了“绿灯”，如果是 ICMP Echo 请求报文，则继续下面的处理，即调用 icmp_echo 回复一个 ICMP Echo 应答报文。此种情况下，进行应答的主机一定是一个网关或者一个路由器。首先网络设备可以对该报文进行接收，一定是报文中 MAC 目的地址为本地网络设备地址，而报文中最终目的 IP 地址却非本机，则表示接收这个报文的本机是一个“代理”，此处“代理”的含义我们既可以理解为一个网关，或者是一个路由器，此种情况下，“代理”可以进行应答。不过这种代理应答只能作为一个可选项，毕竟 Echo 请求数据包探测的是远端主机的可达性。注意 580 行将最终目的地址更改为应答主机的地址，如此发送源端可以得知究竟是不是最终目的主机的应答，或者是中间路由器或者网关的应答（因为 daddr 变量下文中将作为回复数据包中 IP 源地址）。

下面进行 ICMP 报文类型的分类处理，这是在 switch 语句中完成的。

```
582         switch(icmph->type)
583         {
584             case ICMP_TIME_EXCEEDED:
585                 icmp_statistics.IcmpInTimeExcds++;
586                 icmp_unreach(icmph, skb1);
587                 return 0;
588             case ICMP_DEST_UNREACH:
589                 icmp_statistics.IcmpInDestUnreachs++;
590                 icmp_unreach(icmph, skb1);
591                 return 0;
592             case ICMP_SOURCE_QUENCH:
593                 icmp_statistics.IcmpInSrcQuenchs++;
594                 icmp_unreach(icmph, skb1);
595                 return(0);
```

ICMP_TIME_EXCEEDED 类型报文表示在到达最终目的端之前 TTL 值变为 0，此时中间路由器回复一个此种类型的 ICMP 通知报文。ICMP_DEST_UNREACH 表示目的主机不可达，这个目的主机可能指最终目的主机，也可能是一个中间路由器，不可达也可能是端口不可达，具体情况还需要根据 ICMP 首部中代码（Code）字段值进行判断。ICMP_SOURCE_QUENCH 表示远（注意这个“远”是从本地角度来看，从接收端而言，就应该是“源”）端节制，这主要是本地发送速度比之接收端相对较快，接收端无法及时处理数据包，所以发送这样一个数据包通知本地暂时不要再发送数据包，那边已经“饱了”。对于以上三种情况，结果都是数据包无法到达最终目的端，此时都调用 icmp_unreach 函数进行统一处理，处理方式是调用传输层协议错误处理函数（如 tcp_err, udp_err 函数）进行具体分析并采取对应措施。

```
596             case ICMP_REDIRECT:
```

```
597                 icmp_statistics.IcmpInRedirects++;
598                 icmp_redirect(icmph, skb1, dev, saddr);
599                 return(0);
600             case ICMP_ECHO:
601                 icmp_statistics.IcmpInEchos++;
602                 icmp_echo(icmph, skb1, dev, saddr, daddr, len, opt);
603                 return 0;
604             case ICMP_ECHOREPLY:
605                 icmp_statistics.IcmpInEchoReps++;
606                 kfree_skb(skb1, FREE_READ);
607                 return(0);
608             case ICMP_TIMESTAMP:
609                 icmp_statistics.IcmpInTimestamps++;
610                 icmp_timestamp(icmph, skb1, dev, saddr, daddr, len, opt);
611                 return 0;
612             case ICMP_TIMESTAMPREPLY:
613                 icmp_statistics.IcmpInTimestampReps++;
614                 kfree_skb(skb1,FREE_READ);
615                 return 0;
616             /* INFO is obsolete and doesn't even feature in the SNMP stats */
617             case ICMP_INFO_REQUEST:
618                 icmp_info(icmph, skb1, dev, saddr, daddr, len, opt);
619                 return 0;
620             case ICMP_INFO_REPLY:
621                 skb1->sk = NULL;
622                 kfree_skb(skb1, FREE_READ);
623                 return(0);
624             case ICMP_ADDRESS:
625                 icmp_statistics.IcmpInAddrMasks++;
626                 icmp_address(icmph, skb1, dev, saddr, daddr, len, opt);
627                 return 0;
628             case ICMP_ADDRESSREPLY:
629                 /*
630                  *  We ought to set our netmask on receiving this, but
631                  *  experience shows it's a waste of effort.
632                  */
633                 icmp_statistics.IcmpInAddrMaskReps++;
634                 kfree_skb(skb1, FREE_READ);
635                 return(0);
```

596-635 行代码是显而易见的，只有一点值得注意，对于应答报文，实现上直接丢弃应答报文，这一点不会造成影响，例如 PING 程序，虽然其使用 ICMP Echo 请求和应答报文，但实际上其通过直接操作 ICMP 报文完成，而不是借助于内核 icmp_echo 函数。对于其他使用 ICMP 报文的应用而言，一般都是如此。所以网络实现只要处理好请求的情况即可，对于应

答的报文如果应用需要，会进行拦截。内核代码不对这些应答作任何处理，而且其也无法进行任何有效的处理（因为具体如何处理要看应用需要实现何种目的）。

```
636             default:
637                     icmp_statistics.IcmpInErrors++;
638                     kfree_skb(skb1, FREE_READ);
639                     return(0);
640         }
641     /*NOTREACHED*/
642         kfree_skb(skb1, FREE_READ);
643         return(-1);
644 }
```

如果没有找到匹配类型，函数最后返回-1，实际上，当前并未对这个返回值进行检查，所以函数返回什么值无关紧要。

```
645 /*
646  *  Perform any ICMP-related I/O control requests.
647  *  [to vanish soon]
648  */

649 int icmp_ioctl(struct sock *sk, int cmd, unsigned long arg)
650 {
651         switch(cmd)
652         {
653             default:
654                     return(-EINVAL);
655         }
656         return(0);
657 }
```

icmp_ioctl 函数目前不支持任何参数设置。

icmp.c 文件小结

icmp.c 文件是 ICMP 协议实现文件，文件本身代码较少，实现思想也较为简单，在熟悉各种 ICMP 报文格式的情况下，很容易理解相关函数实现，ICMP 报文从总体上共分为两种类型：错误通知报文以及查询报文。Echo 请求和应答，地址掩码请求和应答等都属于查询报文，而重定向，目的端不可达都属于错误通知报文。对于错误通知报文，其返回的 ICMP 错误通知报文中 ICMP 负载还必须包括引起该错误的原数据包的 IP 首部以及传输层 8 字节数据，这也是区分查询报文和错误报文的一种方式。

2.12 net/inet/icmp.h 头文件

该文件主要是对 icmp.c 中定义函数的声明，简单列出代码如下：

```
1   /*
```

```
2     * INET        An implementation of the TCP/IP protocol suite for the LINUX
3     *             operating system.    INET is implemented using the    BSD Socket
4     *             interface as the means of communication with the user level.
5     *
6     *             Definitions for the ICMP module.
7     *
8     * Version:    @(#)icmp.h    1.0.4       05/13/93
9     *
10    * Authors:    Ross Biro, <bir7@leland.Stanford.Edu>
11    *             Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
12    *
13    *             This program is free software; you can redistribute it and/or
14    *             modify it under the terms of the GNU General Public License
15    *             as published by the Free Software Foundation; either version
16    *             2 of the License, or (at your option) any later version.
17    */
18    #ifndef _ICMP_H
19    #define    _ICMP_H

20    #include <linux/icmp.h>


21    extern struct icmp_err icmp_err_convert[];
22    extern struct icmp_mib icmp_statistics;


23    extern void      icmp_send(struct sk_buff *skb_in,    int type, int code,
24                          unsigned long info, struct device *dev);
25    extern int icmp_rcv(struct sk_buff *skb1, struct device *dev,
26                        struct options *opt, unsigned long daddr,
27                        unsigned short len, unsigned long saddr,
28                        int redo, struct inet_protocol *protocol);

29    extern int icmp_ioctl(struct sock *sk, int cmd,
30                            unsigned long arg);

31    #endif     /* _ICMP_H */
```

2.13 net/inet/igmp.c 文件*

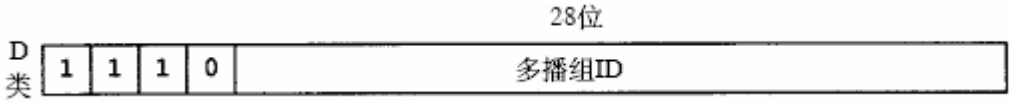
igmp.c 文件是处理 IGMP 协议的专用函数，要理解该文件中函数实现，我们首先必须对多播进行说明，其后给出相关结构中涉及到多播的字段和相关数据结构。在了解这些结构后，对于多播组的加入以及多播查询和应答将很容易理解。

多播提供两类服务：

1〉 向多个目的地址传送数据。如交互式会议系统。

2〉 客户对服务器的请求。如无盘工作站的启动。

D 类地址专门用于多播地址，D 类地址格式如下：



低 28 比特全部用于多播组号，用点分十进制表示的话，地址范围为：224.0.0.0 – 239.255.255.255。 我们将能够接收发送一个特定多播组地址数据的主机集合称为主机组。一个主机组可以跨越多个网络，主机组中成员可随时加入或离开主机组。主机组中对加入的主机数量不进行限制。如下一些多播组 IP 地址是分配好的：

224.0.0.1 – 该子网内所有系统组

224.0.0.2 – 该子网内的所有路由器组

224.0.1.1 – 用于网络时间协议 NTP

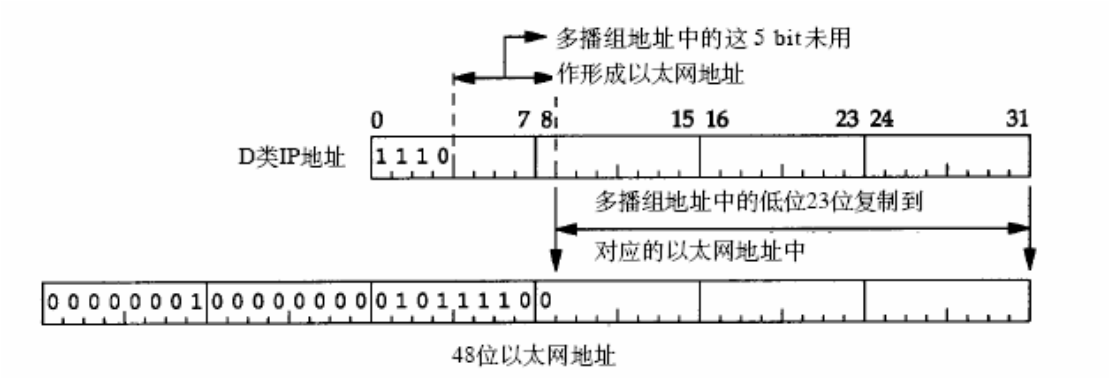
224.0.0.9 – 用于路由协议 RIP-2

等等

多播地址到以太网 MAC 地址的转换

我们在广播一个数据包时，MAC 目的地址被设置为全 1，但是对于一个多播数据报，其 MAC 目的地址如何产生？由于多播数据报是发送到一组主机的，而非全部主机，我们不能使用全 1，当然不能简单使用一个远端主机的 MAC 地址作为 MAC 目的地址（这样就成了单播了）。对于多播 MAC 目的地址的构建，是通过多播 IP 地址映射得到的。不过由于 IPv4 地址长度为 32 比特，而 MAC 地址长度为 48 比特，所有还必须引入一个固定标识符之类的前缀（或者后缀）。对于多播 MAC 地址构建，这个前缀就是 01:00:5e. 由于 IP 地址只贡献低 23 比特，所以与 IP 多播相对应的以太网 MAC 地址范围为 01:00:5e:00:00:00 – 01:00:5e:7f:ff:ff。

下图表示了多播 IP 地址与对应 MAC 地址之间的映射关系：



* 注：本节内容中对 IGMP 协议的说明主要来自文献[1]。

由于只使用了 D 类 IP 多播地址的低 23 比特，除去 D 类地址最高 4 比特固定位，还有 5 比特的未使用位，换句话说，32 个不同的 IP 多播地址将被映射为同一个多播 MAC 地址。为了进行多播数据包过滤，所以在网卡驱动层必须维护一个多播 IP 地址列表。另外对于一个网络接收设备而言，对应多播的支持一般是通过进行 MAC 目的地址中比特位之间的异或计算或者诸如此类的计算方式得到一个比特，由此设置相关寄存器完成的。所以对于网络设备本身而言，其需要维护一个多播 MAC 地址列表。在本版本网络栈实现中，对于多播 MAC 地址和多播 IP 地址的维护都是在 device 结构中进行的。另外对于与进程绑定的多播地址（二元组构成：IP 多播地址和网口设备）是在 sock 结构中进行维护的。由于这些绑定用于不同的目的，所以相关数据结构也有差别。对于网卡设备本身维护的多播 MAC 地址，使用 dev_mc_list 结构（netdevice.h）；对于网卡驱动程序维护的多播 IP 地址，使用 ip_mc_list 结构(igmp.h)；对于进程使用的多播地址（二元组），由 ip_mc_socklist 结构（igmp.h）表示。为便于下文代码分析上的理解，此处再次给出这些结构的定义。

/*include/linux/netdevice.h*/

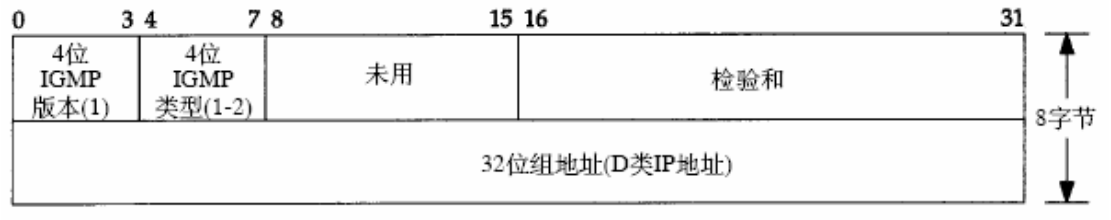
```
41  struct dev_mc_list
42  {
43      struct dev_mc_list *next;
44      char dmi_addr[MAX_ADDR_LEN];
45      unsigned short dmi_addrlen;
46      unsigned short dmi_users;
47  };
```

/*include/linux/igmp.h*/

```
34  struct ip_mc_socklist
35  {
36      unsigned long multiaddr[IP_MAX_MEMBERSHIPS];/* This is a speed trade off */
37      struct device *multidev[IP_MAX_MEMBERSHIPS];
38  };

39  struct ip_mc_list
40  {
41      struct device *interface;
42      unsigned long multiaddr;
43      struct ip_mc_list *next;
44      struct timer_list timer;
45      int tm_running;
46      int users;
47  };
```

对于单个网络的多播是比较简单的，多播进程将目的 IP 地址指明为多播地址，设备驱动层将它转换为响应的 MAC 地址，然后将数据发送出去即可。这些接收进程必须通知他们的 IP 层，他们想接收发送给定多播地址的数据报，并且设备驱动层必须公狗接收这些多播帧。这个过程就是“加入一个多播组”（使用“接收进程”复数形式的原因在于对一确定的多播信息，在同一主机或多个主机上存在多个接收者，这也是为什么要首先使用多播的原因）。当一个主机收到多播数据报时，它必须向属于那个多播组的每个进程均传送一个复制。这和

单个进程收到单播UDP数据报不同。使用多播，一个主机上可能存在多个属于同一多播组的进程。当把多播扩展到单个物理网络以外需要通过路由器转发多播数据时，复杂性就增加了。需要有一个协议让多播路由器了解确定网络中属于确定多播组的任何一个主机。这个协议就是Internet组管理协议（IGMP）。所以IGMP协议主要是管理进程加入和退出多播组的协议，用于支持主机和路由器进行多播。这其中当然同时涉及到相关数据结构的维护。不同于其他多数协议，IGMP协议首部具有固定长度（8字节），没有可选数据，本版本网络栈实现只支持IGMP协议版本1。在第一章中分析igmp.h文件时，我们给出的是IGMPv2的首部格式，下面我们给出版本1的首部格式，比照来看，版本2只是使用了版本1中空闲字段，将版本1中原先一个两个分别为4比特的字段合并为一个8比特字段。其他字段含义相同。下图表示了IGMPv1首部格式。



虽然在IGMPv1首部定义中，第一个字节被分为两个4比特字段，但实现上仍然作为一个字段进行处理，原因是这两个字段取值有限，对于一个固定的版本，4比特版本号字段值是固定的，而4比特类型可取值只有两个（一个为查询，一个为报告），所以没有必要进行nibble（一个nibble表示4个比特）操作，增加代码复杂性。

加入一个多播组

多播的基础就是一个进程的概念（使用的术语进程是指操作系统执行的一个程序），该进程在一个主机的给定接口上加入了一个多播组。在一个给定接口上的多播组中的成员是动态的，它随时因进程加入和离开多播组而变化。这里所指的进程必须以某种方式在给定的接口上加入某个多播组。进程也能离开先前加入的多播组。这些是一个支持多播主机中任何A P I所必需的部分。使用限定词“接口”是因为多播组中的成员是与接口相关联的。一个进程可以在多个接口上加入同一多播组。

IGMP 报告和查询

多播路由器使用IGMP报文来记录与该路由器相连网络中组成员的变化情况。使用规则如下：

1) 当第一个进程加入一个组时，主机就发送一个IGMP报告。如果一个主机的多个进程加入同一组，只发送一个IGMP报告。这个报告被发送到进程加入组所在的同一接口上。

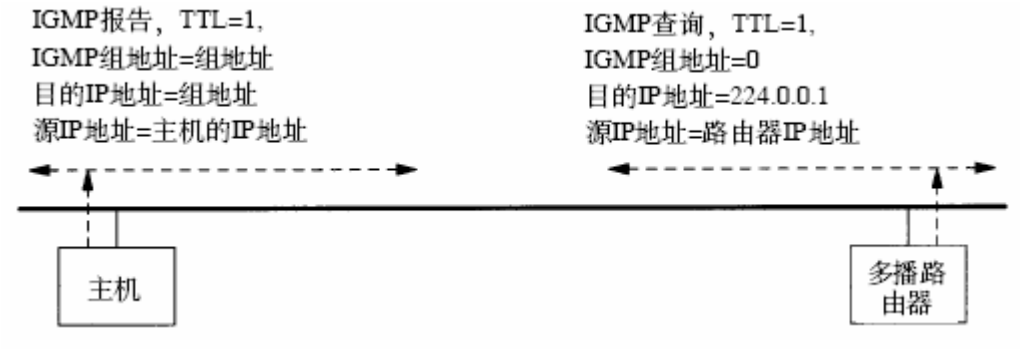
2) 进程离开一个组时，主机不发送IGMP报告，即便是组中的最后一个进程离开。主机知道在确定的组中已不再有组成员后，在随后收到的IGMP查询中就不再发送报告报文。

3) 多播路由器定时发送IGMP查询来了解是否还有任何主机包含有属于多播组的进程。多播路由器必须向每个接口发送一个IGMP查询。因为路由器希望主机对它加入的每个多播组均发回一个报告，因此IGMP查询报文中的组地址被设置为0。

4) 主机通过发送IGMP报告来响应一个IGMP查询，对每个至少还包含一个进程的组均要发回IGMP报告。

使用这些查询和报告报文，多播路由器对每个接口保持一个表，表中记录接口上至少还

包含一个主机的多播组。当路由器收到要转发的多播数据报时，它只将该数据报转发到（使用相应的多播链路层地址）还拥有属于那个组主机的接口上。下图显示了两个IGMP报文，一个是主机发送的报告，另一个是路由器发送的查询。该路由器正在要求那个接口上的每个主机说明它加入的每个多播组。



实现细节

为改善该协议的效率，有许多实现的细节要考虑。首先，当一个主机首次发送IGMP报告（当第一个进程加入一个多播组）时，并不保证该报告被可靠接收（因为使用的是IP交付服务）。下一个报告将在间隔一段时间后发送。这个时间间隔由主机在0-10秒的范围内随机选择。其次，当一个主机收到一个从路由器发出的查询后，并不立即响应，而是经过一定的时间间隔后才发出一些响应（采用“响应”的复数形式是因为该主机必须对它参加的每个组均发送一个响应）。既然参加同一多播组的多个主机均能发送一个报告，可将它们的发送间隔设置为随机时延。在一个物理网络中的所有主机将收到同组其他主机发送的所有报告，因为如上图所示的报告中的目的地址是那个组地址。这意味着如果一个主机在等待发送报告的过程中，却收到了发自其他主机的相同报告，则该主机的响应就可以不必发送了。因为多播路由器并不关心有多少主机属于该组，而只关心该组是否还至少拥有一个主机。的确，一个多播路由器甚至不关心哪个主机属于一个多播组。它仅仅想知道在给定的接口上的多播组中是否还至少有一个主机。在没有任何多播路由器的单个物理网络中，仅有的IGMP通信量就是在主机加入一个新的多播组时，支持IP多播的主机所发出的报告。

另外需要注意的地方是上图中TTL值的设置，此处的TTL即表示IP首部中TTL字段。TTL表示生存时间，而现实中实现上表示的是经过中间路由器的数目（我们称之为跳数）。TTL设置为0，则数据包被局限于本机中。在默认情况下，待传多播数据报的TTL被设置为1，这将使多播数据报仅局限在同一子网内传送。更大的TTL值能被多播路由器转发。从224.0.0.0到224.0.0.255的特殊地址空间是打算用于多播范围不超过1跳的应用。不管TTL值是多少，多播路由器均不转发目的地址为这些地址中的任何一个地址的数据报。其中224.0.0.1被称为全主机组，这个多播地址涉及在一个物理网络中的所有具备多播能力的主机和路由器。当接口初始化后，所有具备多播能力接口上的主机均自动加入这个多播组。这个组的成员无需发送IGMP报告。

在经过以上对IGMP协议的由来以及有关多播组的介绍后，下面对于igmp.c文件函数的实现将很容易理解。我们即将进入对igmp.c文件对于IGMP协议实现的分析。不过首先介绍一下本版本网络实现对多播和IGMP协议的支持。前文中已经引出了dev_mc_list, ip_mc_list, ip_mc_socklist三个数据结构，用来表示多播地址（MAC地址或者IP地址）。但是只有表示多播地址的结构还不够，还需要内核使用这些结构对多播地址进行维护，从而实现对多播以

及IGMP协议的支持。上文中我们还提到涉及到多播以及IGMP协议的三个方面：网络设备本身，网络设备驱动程序，以及套接字本身或者更确切的说是用户进程。网络设备本身只能维护多播MAC地址，但由于多播MAC地址与多播IP地址映射关系不是一一对应的，一个多播MAC地址可以对应有32个不同的IP地址，所有需要网络设备驱动程序进一步进行多播数据报的过滤，即驱动程序还需要维护一个多播IP地址列表。由于驱动程序维护的列表是针对本机上所有使用该网络设备发送数据包的套接字，而每个套接字可能关心来自不同多播组的数据报，所以套接字本身必须维护一个多播IP地址列表，注意这个列表不同于由驱动程序维护的列表，这个列表中包含有多播IP地址以及对应的网络接收设备，而由驱动程序维护的列表仅仅包含IP多播地址，具体的驱动程序使用ip_mc_list结构，而套接字使用ip_mc_socklist，二者的不同均体现在两个结构的不同上；网络设备本身使用dev_mc_list结构。在实现上，为了减小驱动程序的复杂度，将本由驱动程序维护IP多播地址列表，也作为表示网络设备的数据结构的一个字段，这样三个列表都在内核相关结构中有所体现，而无须依赖于驱动程序中自定义数据结构。具体地，有device结构（表示一个网络设备）和sock结构中如下字段。

```
/*include/linux/netdevice.h*/
54  struct device
55  {

    ......
105   struct dev_mc_list     *mc_list;  /* Multicast mac addresses  */
106   int            mc_count;  /* Number of installed mcasts   */
107   struct ip_mc_list  *ip_mc_list;   /* IP multicast filter chain    */

    ......
138 };

/*net/inet/sock.h*/
54  struct sock {

    ......
169 #ifdef CONFIG_IP_MULTICAST
170   int               ip_mc_ttl;          /* Multicasting TTL */
171   int               ip_mc_loop;         /* Loopback (not implemented yet) */
172   char              ip_mc_name[MAX_ADDR_LEN];   /* Multicast device name */
173   struct ip_mc_socklist     *ip_mc_list;            /* Group array */
174 #endif

    ......
186 };


1   /*
2    *  Linux NET3: Internet Gateway Management Protocol  [IGMP]
3    *
4    *  Authors:
```

```
5    *      Alan Cox <Alan.Cox@linux.org>
6    *
7    *  WARNING:
8    *      This is a 'preliminary' implementation... on your own head
9    *  be it.
10   *
11   *  This program is free software; you can redistribute it and/or
12   *  modify it under the terms of the GNU General Public License
13   *  as published by the Free Software Foundation; either version
14   *  2 of the License, or (at your option) any later version.
15   */


16  #include <asm/segment.h>
17  #include <asm/system.h>
18  #include <linux/types.h>
19  #include <linux/kernel.h>
20  #include <linux/sched.h>
21  #include <linux/string.h>
22  #include <linux/config.h>
23  #include <linux/socket.h>
24  #include <linux/sockios.h>
25  #include <linux/in.h>
26  #include <linux/inet.h>
27  #include <linux/netdevice.h>
28  #include "ip.h"
29  #include "protocol.h"
30  #include "route.h"
31  #include <linux/skbuff.h>
32  #include "sock.h"
33  #include <linux/igmp.h>

34  #ifdef CONFIG_IP_MULTICAST
```

34行ifdef定义管到igmp.c文件结尾，换句话说，整个IGMP协议实现只有在定义内核常量CONFIG_IP_MULTICAST时才被包含在内核之中。而这些常量的定义在进行内核编译时通常作为一个选项而存在。

```
35  /*
36   *  Timer management
37   */


38  static void igmp_stop_timer(struct ip_mc_list *im)
```

```
39  {
40      del_timer(&im->timer);
41      im->tm_running=0;
42  }
```

如上文中对IGMP报告报文的说明，当一个主机首次发送IGMP报告（当第一个进程加入一个多播组）时，并不保证该报告被可靠接收（因为使用的是IP交付服务）。下一个报告将在间隔一段时间后发送。这个时间间隔由主机在0-10秒的范围内随机选择。其次，当一个主机收到一个从路由器发出的查询后，并不立即响应，而是经过一定的时间间隔后才发出一些响应。因为多播路由器并不关心有多少主机属于该组，而只关心该组是否还至少拥有一个主机。这意味着如果一个主机在等待发送报告的过程中，却收到了发自其他主机的相同报告，则该主机的响应就可以不必发送了。igmp_stop_timer函数被两个函数调用:igmp_timer_expire，igmp_heard_report。igmp_heard_report函数在接收到同组其他主机发送的IGMP报告报文时被调用，依据以上的设计思想，此时可以不发送报文。所以停止定时器。igmp_timer_expire则表示定时器正常到期，此时发送一个IGMP报告报文，在发送报告报文的同时，也停止定时器。本版本并未实现IGMP报告报文的主动通知，而是响应一个IGMP查询报文时，才发送IGMP报告报文，在发送时，如上所述，将延迟一段0-10秒的随机时间，如果在这段时间内接收到相同子网内其他主机的IGMP报告报文，则中断定时器延迟，取消发送。否则定时器正常到期，发送一个IGMP报告报文，这通常是延迟时间最短的主机发送的第一个IGMP报告报文。

```
43  static int random(void)
44  {
45      static unsigned long seed=152L;
46      seed=seed*69069L+1;
47      return seed^jiffies;
48  }

49  static void igmp_start_timer(struct ip_mc_list *im)
50  {
51      int tv;
52      if(im->tm_running)
53          return;
54      tv=random()%(10*HZ);        /* Pick a number any number 8) */
55      im->timer.expires=tv;
56      im->tm_running=1;
57      add_timer(&im->timer);
58  }
```

igmp_start_timer函数被igmp_heard_query函数调用，当接收到路由器发送的IGMP查询报文时，设置一个0-10秒内随机延迟时间的定时器，在定时器到期后，发送一个IMGP报告报文。注意这个定时器是作为ip_mc_list结构中一个字段存在的。这个定时器的初始化是在igmp_init_timer函数中完成的，我们提前介绍igmp_init_timer函数，该函数实现如下：

```
92  static void igmp_init_timer(struct ip_mc_list *im)
```

```
93  {
94      im->tm_running=0;
95      init_timer(&im->timer);
96      im->timer.data=(unsigned long)im;
97      im->timer.function=&igmp_timer_expire;
98  }
```

定时器到期执行函数为igmp_timer_expire。我们不妨将igmp_timer_expire函数也提前介绍，该函数实现如下；

```
86  static void igmp_timer_expire(unsigned long data)
87  {
88      struct ip_mc_list *im=(struct ip_mc_list *)data;
89      igmp_stop_timer(im);
90      igmp_send_report(im->interface, im->multiaddr,
                            IGMP_HOST_MEMBERSHIP_REPORT);
91  }
```

正如所料想的一样，igmp_timer_expire函数调用igmp_send_report函数发送一个IMGP报告报文。同时停止定时器。这些函数实现本身都比较简单。关键是对IGMP协议本身的了解，这一点在前文中已经进行了着重介绍。如果读者还不能对其中的关系进行理解，建议仔细阅读前文中对IGMP协议以及多播的说明。

igmp_send_report函数完成发送一个IMGP报告报文的功能。经过本书前文中对TCP,UDP,ICMP等协议的介绍，相信读者对于发送一个数据包时所需要进行的工作进行很了解了：主要是完成对各协议首部的创建。用户数据的封装简单的说就是将数据从用户缓冲区复制到指定的内核缓冲区中。虽然如同ICMP协议一样，一般我们也将IGMP协议认为是网络层协议，但实现上IGMP报文中传输是封装在IP报文之中的，所以协议首部的创建包括MAC，IP，IGMP首部。对于MAC，IP首部通过ip_build_header函数完成（这个函数在介绍ICMP，TCP，UDP协议时已经多次遇到），所以创建首部的工作主要针对IGMP首部。由于IGMP首部格式简单，长度固定，所以这个工作一目了然。值得注意的是在调用ip_build_header函数传入的多播IP地址（多播MAC地址根据多播IP地址构建，构建关系如上文所述）。

```
59  /*
60   *  Send an IGMP report.
61   */

62  #define MAX_IGMP_SIZE (sizeof(struct igmphdr)+sizeof(struct iphdr)+64)

63  static void igmp_send_report(struct device *dev, unsigned long address, int
type)
64  {
65      struct sk_buff *skb=alloc_skb(MAX_IGMP_SIZE, GFP_ATOMIC);
66      int tmp;
67      struct igmphdr *igh;
```

```
68      if(skb==NULL)
69          return;
70      tmp=ip_build_header(skb, INADDR_ANY, address, &dev, IPPROTO_IGMP, NULL,
71                  skb->mem_len, 0, 1);
72      if(tmp<0)
73      {
74          kfree_skb(skb, FREE_WRITE);
75          return;
76      }
77      igh=(struct igmphdr *)(skb->data+tmp);
78      skb->len=tmp+sizeof(*igh);
79      igh->csum=0;
80      igh->unused=0;
81      igh->type=type;
82      igh->group=address; //多播IP组地址直接来自调用本函数的函数
83      igh->csum=ip_compute_csum((void *)igh,sizeof(*igh));
84      ip_queue_xmit(NULL,dev,skb,1);
85  }

99  static void igmp_heard_report(struct device *dev, unsigned long address)
100 {
101     struct ip_mc_list *im;
102     for(im=dev->ip_mc_list;im!=NULL;im=im->next)
103         if(im->multiaddr==address)
104             igmp_stop_timer(im);
105 }

106 static void igmp_heard_query(struct device *dev)
107 {
108     struct ip_mc_list *im;
109     for(im=dev->ip_mc_list;im!=NULL;im=im->next)
110         if(!im->tm_running && im->multiaddr!=IGMP_ALL_HOSTS)
111             igmp_start_timer(im);
112 }
```

igmp_heard_report函数以及igmp_heard_query函数顾名思义是在分别接收到IGMP报告报文和IGMP查询报文时被调用。对于IGMP报告报文的情况，由于路由器并不关心哪台主机加入到那个多播组，而只关心是否有主机在某个多播组中，所以对于一个加入某个多播组的主机而言，如果其他主机已经发送这个多播组的报告报文，那么本机就不需要再发送这样的报告报文，igmp_heard_report函数完成的工作即是如此，但接收到一个有其他主机发送的IGMP报告报文时,其检查本机中是否也设置了发送针对同一多播组的IGMP报告报文定时器,如果有，则停止该定时器。注意函数实现中是对多播IP地址的检查，是对device结构中ip_mc_list指向的多播IP地址列表的遍历。对于IGMP查询报文，如果本机有任何加入除了全主机多播组之外的其他多播组，则必须将此多播组报告给路由器，此时设置一个定时器，在延迟0-10

秒的随机时间后，发送这样一个报告报文，igmp_heared_query函数完成的工作即如此。对于全主机多播组是无须进行报告的，因为默认的凡是支持多播的主机，默认的都要进入该多播组。

```
113 /*
114  *  Map a multicast IP onto multicast MAC for type ethernet.
115  */

116 static void ip_mc_map(unsigned long addr, char *buf)
117 {
118     addr=ntohl(addr);
119     buf[0]=0x01;
120     buf[1]=0x00;
121     buf[2]=0x5e;
122     buf[5]=addr&0xFF;
123     addr>>=8;
124     buf[4]=addr&0xFF;
125     addr>>=8;
126     buf[3]=addr&0x7F;
127 }
```

ip_mc_map函数完成多播IP地址到多播MAC地址之间的映射。参数addr表示多播IP地址，由此映射而成的MAC地址被填充到buf参数中而返回。至于多播IP地址以及MAC地址之间的映射关系请读者参考前文说明，该函数实现自然一目了然。

```
128 /*
129  *  Add a filter to a device
130  */

131 void ip_mc_filter_add(struct device *dev, unsigned long addr)
132 {
133     char buf[6];
134     if(dev->type!=ARPHRD_ETHER)
135         return; /* Only do ethernet now */
136     ip_mc_map(addr,buf);
137     dev_mc_add(dev,buf,ETH_ALEN,0);
138 }

139 /*
140  *  Remove a filter from a device
141  */

142 void ip_mc_filter_del(struct device *dev, unsigned long addr)
143 {
```

```
144     char buf[6];
145     if(dev->type!=ARPHRD_ETHER)
146         return; /* Only do ethernet now */
147     ip_mc_map(addr,buf);
148     dev_mc_delete(dev,buf,ETH_ALEN,0);
149 }
```

在上文中对多播和IGMP协议的介绍中，我们提到对于多播地址的维护是分为三个不同方面进行的。设备本身维护MAC地址列表，因为对于一个具体的网络设备而言，其对于数据报接受与否的判断根据相关寄存器设置的情况而定，对于多播的支持，一般是对MAC地址位做异或之类的计算后通过设置寄存器相关位完成。总之，对于网络接收设备而言，无法直接使用多播IP地址，必须将多播IP地址转换为MAC地址后方可为网络设备所使用，从而完成第一道多播数据报的过滤防线。ip_mc_filter_add和ip_mc_filter_del函数即完成从相应IP多播地址到MAC地址的添加和删除工作。当新添加一个IP多播地址时，需要调用ip_mc_filter_add函数将该多播地址转换为MAC地址，并重新设置网络设备硬件寄存器，从而加入对此类多播数据报的接收。当删除一个多播地址时，ip_mc_filter_del函数被调用，重新设置网络设备，过滤掉对应多播数据报的接收。这两个函数实现上首先调用ip_mc_map函数完成从多播IP地址到MAC的映射，然后以此MAC地址调用相关函数对设备本身的多播MAC地址列表进行操作，并同时重新设置硬件寄存器（要使新的操作有效，一般还需要从软件上重启网络设备）。此处相关函数dev_mc_add, dev_mc_delete函数定义在dev_mcast.c中。在分析完igmp.c后，我们就对该文件进行分析。

```
150 static void igmp_group_dropped(struct ip_mc_list *im)
151 {
152     del_timer(&im->timer);
153     igmp_send_report(im->interface, im->multiaddr, IGMP_HOST_LEAVE_MESSAGE);
154     ip_mc_filter_del(im->interface, im->multiaddr);
155 /*  printk("Left group %lX\n",im->multiaddr);*/
156 }

157 static void igmp_group_added(struct ip_mc_list *im)
158 {
159     igmp_init_timer(im);
160     igmp_send_report(im->interface, im->multiaddr,
                         IGMP_HOST_MEMBERSHIP_REPORT);
161     ip_mc_filter_add(im->interface, im->multiaddr);
162 /*  printk("Joined group %lX\n",im->multiaddr);*/
163 }
```

igmp_group_added和igmp_group_dropped函数即负责多播组地址添加和删除。首先一个新添加的多播组有ip_mc_list结构表示，对于组添加的情况，我们需要对表示这个组的ip_mc_list结构中相关字段进行初始化，最主要的就是对定时器的初始化工作，igmp_init_timer函数上文中已经介绍，该函数将定时器到期执行函数设置为igmp_timer_expire，igmp_timer_expire函数负责发送一个IGMP报告报文。无论是新添加一

个组，还是退出一个组，此处都立刻发送一个IGMP报告报文，通知路由器相关变化。在前文对IGMP协议的介绍中，我们提到对于新加入的一个组的情况，我们一般需要发送一个IGMP报告报文，而对于退出一个组，则无须发送IGMP报告报文，路由器在发送IGMP查询报文后如果没有收到对应组的报告报文，自然会删除对该IP多播组的维护。但是此处在退出一个组后，调用了igmp_send_report立刻发送一个IGMP报告报文，这也并非错误。二者都可。
函数最后各自调用ip_mc_filter_del和ip_mc_filter_add函数更改设备MAC多播地址列表反映新的变化。最后需要提及的是，igmp_group_added和igmp_group_dropped函数实现上虽然负责加入和退出一个组的工作，但这两个函数并非是在上层加入和退出一个组时，直接被调用的函数，因为从实现中可以看出这两个函数只涉及到设备维护的MAC地址列表的操作（通过对ip_mc_filter_del和ip_mc_filter_add函数的调用），但并没有涉及驱动程序和套接字维护的多播IP地址的操作，所以他们只是作为一个多播组被添加和删除时的一部分实现，换句话说，还有更上层的函数调用它们。这个更上层的函数将在下文中介绍。

下面我们介绍接收到一个IGMP报文时该如何处理，即igmp_rcv函数的实现。该函数工作类似icmp_rcv, tcp_rcv, udp_rcv， 是对使用IGMP协议的数据报进行处理的总入口函数。负责处理所有接收到的使用IGMP协议的数据报文。根据前文所述，IGMP报文只有两种类型：IGMP查询报文，IGMP报告报文。那么igmp_rcv函数主要也是对这两个类型进行检查，从而调用具体函数进行处理。

```
164 int igmp_rcv(struct sk_buff *skb, struct device *dev, struct options *opt,
165     unsigned long daddr, unsigned short len, unsigned long saddr, int redo,
166     struct inet_protocol *protocol)
167 {
168     /* This basically follows the spec line by line -- see RFC1112 */
169     struct igmphdr *igh=(struct igmphdr *)skb->h.raw;

170     if(skb->ip_hdr->ttl!=1 || ip_compute_csum((void *)igh,sizeof(*igh)))
171     {
172         kfree_skb(skb, FREE_READ);
173         return 0;
174     }

175     if(igh->type==IGMP_HOST_MEMBERSHIP_QUERY && daddr==IGMP_ALL_HOSTS)
176         igmp_heard_query(dev);
177     if(igh->type==IGMP_HOST_MEMBERSHIP_REPORT && daddr==igh->group)
178         igmp_heard_report(dev,igh->group);
179     kfree_skb(skb, FREE_READ);
180     return 0;
181 }
```

函数实现上非常简单，170行是对TTL字段的检查，对于多播数据报，TTL值必须设置为1。175，177行代码分别处理IGMP查询报文和IGMP报告报文。

我们刚刚介绍igmp_group_dropped, igmp_group_added函数并指出这两个函数被更上层的函数调用完成多播组的加入和退出工作。“更上层”这个词有些不准确，从前文中涉及内核

对多播支持的三个方面来看，应该说，igmp_group_dropped, igmp_group_added函数负责了网络设备维护的MAC多播地址列表，而驱动程序IP多播地址列表以及套接字对应的多播地址列表并未涉及，这就表明还有其他函数负责这些列表的维护。对于驱动程序IP多播地址列表的维护即由如下ip_mc_inc_group和ip_mc_dec_group函数负责。

```
182 /*
183  *  Multicast list managers
184  */


185 /*
186  *  A socket has joined a multicast group on device dev.
187  */

188 static void ip_mc_inc_group(struct device *dev, unsigned long addr)
189 {
190     struct ip_mc_list *i;
191     for(i=dev->ip_mc_list;i!=NULL;i=i->next)
192     {
193         if(i->multiaddr==addr)
194         {
195             i->users++;
196             return;
197         }
198     }
199     i=(struct ip_mc_list *)kmalloc(sizeof(*i), GFP_KERNEL);
200     if(!i)
201         return;
202     i->users=1;
203     i->interface=dev;
204     i->multiaddr=addr;
205     i->next=dev->ip_mc_list;
206     igmp_group_added(i);
207     dev->ip_mc_list=i;
208 }

209 /*
210  *  A socket has left a multicast group on device dev
211  */

212 static void ip_mc_dec_group(struct device *dev, unsigned long addr)
213 {
214     struct ip_mc_list **i;
215     for(i=&(dev->ip_mc_list);(*i)!=NULL;i=&(*i)->next)
```

```
216     {
217         if((*i)->multiaddr==addr)
218         {
219             if(--((*i)->users))
220                 return;
221             else
222             {
223                 struct ip_mc_list *tmp= *i;
224                 igmp_group_dropped(tmp);
225                 *i=(*i)->next;
226                 kfree_s(tmp,sizeof(*tmp));
227             }
228         }
229     }
230 }
```

如前文所述，驱动程序使用ip_mc_list结构表示IP多播地址，ip_mc_inc_group和ip_mc_dec_group函数即完成对一个表示新多播地址对应的ip_mc_list结构的创建工作，当然首先我们必须检查当前地址列表中是否已经有这样一个相同的组地址存在，如果存在，则简单增加用户使用计数即可。因为驱动程序在这方面如同路由器一样，并不关心究竟有多少上层套接字加入了这个多播组,维护用户计数的首要目的是防止该ip_mc_list结构被提前释放，从而造成非法内存访问之类的系统错误。驱动程序维护的多播地址列表有device结构ip_mc_list字段指向，如果当前没有对应的多播地址，则创建一个新的ip_mc_list结构，并加入到由device结构ip_mc_list字段(注意此处不要混淆，device结构中对驱动程序维护的IP地址列表的指针名称正好与表示一个多播地址的结构名称相同)指向的地址列表中,为了增加软件处理效率，这个新的ip_mc_list结构被加入到列表的首部。在完成对应驱动程序的IP多播地址列表的操作后,各自调用igmp_group_added和igmp_group_dropped函数完成对应设备的MAC多播地址列表的操作。那么对于一个多播（组）地址的加入和退出现在只剩下对应套接字多播地址列表的操作了，这个操作定义在ip_mc_join_group和ip_mc_leave_group函数中。

```
264 /*
265  *  Join a socket to a group
266  */

267 int ip_mc_join_group(struct sock *sk , struct device *dev, unsigned long addr)
268 {
269     int unused= -1;
270     int i;
271     if(!MULTICAST(addr))
272         return -EINVAL;
273     if(!(dev->flags&IFF_MULTICAST))
274         return -EADDRNOTAVAIL;
275     if(sk->ip_mc_list==NULL)
276     {
```

```
277         if((sk->ip_mc_list=(struct ip_mc_socklist *)
                        kmalloc(sizeof(*sk->ip_mc_list), GFP_KERNEL))==NULL)
278             return -ENOMEM;
279         memset(sk->ip_mc_list,'\0',sizeof(*sk->ip_mc_list));
280     }
281     for(i=0;i<IP_MAX_MEMBERSHIPS;i++)
282     {
283         if(sk->ip_mc_list->multiaddr[i]==addr &&
                                sk->ip_mc_list->multidev[i]==dev)
284             return -EADDRINUSE;
285         if(sk->ip_mc_list->multidev[i]==NULL)
286             unused=i;
287     }

288     if(unused==-1)
289         return -ENOBUFS;
290     sk->ip_mc_list->multiaddr[unused]=addr;
291     sk->ip_mc_list->multidev[unused]=dev;
292     ip_mc_inc_group(dev,addr);
293     return 0;
294 }

295 /*
296  *  Ask a socket to leave a group.
297  */

298 int ip_mc_leave_group(struct sock *sk, struct device *dev, unsigned long addr)
299 {
300     int i;
301     if(!MULTICAST(addr))
302         return -EINVAL;
303     if(!(dev->flags&IFF_MULTICAST))
304         return -EADDRNOTAVAIL;
305     if(sk->ip_mc_list==NULL)
306         return -EADDRNOTAVAIL;

307     for(i=0;i<IP_MAX_MEMBERSHIPS;i++)
308     {
309         if(sk->ip_mc_list->multiaddr[i]==addr &&
                            sk->ip_mc_list->multidev[i]==dev)
310         {
311             sk->ip_mc_list->multidev[i]=NULL;
312             ip_mc_dec_group(dev,addr);
313             return 0;
```

```
314         }
315     }
316     return -EADDRNOTAVAIL;
317 }
```

套接字维护的地址列表由sock结构中ip_mc_list字段指向（同样，这个字段名称又与表示一个IP多播地址的结构名相同，不过sock结构中ip_mc_list字段是一个ip_mc_socklist类型的指针），ip_mc_list字段是一个ip_mc_socklist结构类型，对于由套接字维护的多播地址列表有一个二元组构成：IP多播地址和绑定的网络设备接口。为便于读者理解，重新给出ip_mc_socklist结构定义如下：

```
/*include/linux/igmp.h*/
34  struct ip_mc_socklist
35  {
36      unsigned long multiaddr[IP_MAX_MEMBERSHIPS];/*This is a speed trade off */
37      struct device *multidev[IP_MAX_MEMBERSHIPS];
38  };
```

ip_mc_socklist结构中multiaddr字段和mutltidev字段各元素一一对应。

ip_mc_join_group和ip_mc_leave_group函数参数中sk表示对应操作的套接字，dev表示多播地址绑定的网络设备，addr即表示这个多播组IP地址。对于加入的情况，我们首先检查参数addr表示的地址是不是一个多播地址(271行)，以及指定绑定的设备是否支持多播(273行)，如果这两个条件通过，那么就可以对这个新的多播组IP地址进行添加了，不过在添加之前还需要检查一下当前是否已经存在这样一个多播组IP地址，如果存在，则无须进行任何添加操作。否则就从ip_mc_socklist结构中寻找一个空闲元素项，将多播地址和对应绑定设备加入到多播组IP地址数组中。对照ip_mc_socklist结构的定义，很容易理解ip_mc_join_group和ip_mc_leave_group函数实现。当然ip_mc_join_group和ip_mc_leave_group函数只是完成对应套接字多播IP地址的操作，所以其进一步调用ip_mc_inc_group和ip_mc_dec_group函数完成其他两个方面多播地址列表的操作，这在前文中已经进行了讨论。

综上所述，一个多播组的加入和退出涉及到如下6个函数的调用，6个函数分为3组，每组函数完成一个方面多播地址列表的维护工作。并由最下层的一组函数（igmp_group_added和igmp_group_dropped函数）完成IGMP报告报文的发送。

1. ip_mc_join_group和ip_mc_leave_group函数—完成对应套接字多播地址列表的维护
2. ip_mc_inc_group和ip_mc_dec_group函数—完成对应驱动程序多播地址列表的维护
3. igmp_group_added和igmp_group_dropped函数—完成对应网络设备多地址列表的维护

```
231 /*
232  *  Device going down: Clean up.
233  */

234 void ip_mc_drop_device(struct device *dev)
235 {
236     struct ip_mc_list *i;
```

```
237     struct ip_mc_list *j;
238     for(i=dev->ip_mc_list;i!=NULL;i=j)
239     {
240         j=i->next;
241         kfree_s(i,sizeof(*i));
242     }
243     dev->ip_mc_list=NULL;
244 }
```

ip_mc_drop_device函数处理一个网络设备停止工作的情况，此时需要释放用于维护多播地址的内存空间，不过这个函数仅仅释放了对应驱动程序的IP多播地址列表，没有释放对应网络设备MAC多播地址列表，对此我们可以这样理解：对应套接字的多播地址列表在套接字关闭时会自行得到处理,对应网络设备的多播地址列表网络设备本身(即device结构被释放时)也会得到处理；对应驱动程序多播地址列表原则上讲这个列表应该完全有驱动程序本身负责，对于网络设备应该不可见，为了降低驱动程序复杂性或者是内核对多播处理的一致性，这个对应驱动程序的多播地址列表现在放在了device结构中，用个简单的例子就是，一个我朋友的不属于我的东西寄存在我这儿,现在我要走了,我自己的东西当然我自己会处理好(我自己带走)，但这个寄存的朋友的东西我不能带走，在离开之前，我就必须处理掉。此处的思想类似如此。

```
245 /*
246  *  Device going up. Make sure it is in all hosts
247  */

248 void ip_mc_allhost(struct device *dev)
249 {
250     struct ip_mc_list *i;
251     for(i=dev->ip_mc_list;i!=NULL;i=i->next)
252         if(i->multiaddr==IGMP_ALL_HOSTS)
253             return;
254     i=(struct ip_mc_list *)kmalloc(sizeof(*i), GFP_KERNEL);
255     if(!i)
256         return;
257     i->users=1;
258     i->interface=dev;
259     i->multiaddr=IGMP_ALL_HOSTS;
260     i->next=dev->ip_mc_list;
261     dev->ip_mc_list=i;
262     ip_mc_filter_add(i->interface, i->multiaddr);

263 }
```

ip_mc_allhost函数在接口启动工作时被调用，用于自动添加全多播组地址（224.0.0.1）。259行中IGMP_ALL_HOSTS常量定义为224.0.0.1，注意这个多播组地址不与任何套接字绑定，

所以此处只对涉及到驱动程序多播地址列表和网络设备多播地址列表，没有套接字多播地址列表的操作。

```
318 /*
319  *  A socket is closing.
320  */

321 void ip_mc_drop_socket(struct sock *sk)
322 {
323     int i;

324     if(sk->ip_mc_list==NULL)
325         return;

326     for(i=0;i<IP_MAX_MEMBERSHIPS;i++)
327     {
328         if(sk->ip_mc_list->multidev[i])
329         {
330             ip_mc_dec_group(sk->ip_mc_list->multidev[i],
                            sk->ip_mc_list->multiaddr[i]);
331             sk->ip_mc_list->multidev[i]=NULL;
332         }
333     }
334     kfree_s(sk->ip_mc_list,sizeof(*sk->ip_mc_list));
335     sk->ip_mc_list=NULL;
336 }

337 #endif
```

ip_mc_drop_socket函数处理一个使用多播的套接字被关闭时对多播地址列表的处理。324行检查该套接字是否使用了多播，如果没有，则直接返回。否则遍历套接字对应多播地址列表，对每个多播地址对应的各层结构进行释放（注意330行对ip_mc_dec_group函数的调用）。

337行对应34行ifdef语句。

至此，完成对IGMP协议实现文件igmp.c的分析。IGMP协议主要完成对主机和路由器中多播地址的管理，主要是多播组地址的加入和退出，以及完成加入和退出需要的各项工作。IGMP协议比较简单，所以对应实现函数也比较简单，结合前文对多播和IGMP协议的较为详细的介绍，在经过对TCP协议的理解后，这些代码应不造成任何障碍。

上面分析中，涉及到dev_mcast.c中两个函数没有介绍：dev_mc_add, dev_mc_delete. 从dev_mcast.c整个文件实现功能上来看，这个文件与igmp.c文件是绑定的，都是对IGMP协议的实现。下面我们就对dev_mcast.c文件实现进行分析。

2.14 net/inet/dev_mcast.c 文件

该文件只定义了四个函数实现，在分析上为了体现逻辑性，我们对此文件不再简单的顺序分析，而是以函数之间的调用关系为线进行分析，既然该文件是对IGMP协议的辅助实现，下面我们即以igmp.c中调用dev_mcast.c中的函数为入口点进行分析，不过首先还是给出dev_mcast.c文件开始处一些头定义语句。

```
1   /*
2    *  Linux NET3: Multicast List maintenance.
3    *
4    *  Authors:
5    *      Tim Kordas <tjk@nostromo.eeap.cwru.edu>
6    *      Richard Underwood <richard@wuzz.demon.co.uk>
7    *
8    *  Stir fried together from the IP multicast and CAP patches above
9    *      Alan Cox <Alan.Cox@linux.org>
10   *
11   *  Fixes:
12   *      Alan Cox    :   Update the device on a real delete
13   *                  rather than any time but...
14   *
15   *  This program is free software; you can redistribute it and/or
16   *  modify it under the terms of the GNU General Public License
17   *  as published by the Free Software Foundation; either version
18   *  2 of the License, or (at your option) any later version.
19   */

20  #include <asm/segment.h>
21  #include <asm/system.h>
22  #include <asm/bitops.h>
23  #include <linux/types.h>
24  #include <linux/kernel.h>
25  #include <linux/sched.h>
26  #include <linux/string.h>
27  #include <linux/mm.h>
28  #include <linux/socket.h>
29  #include <linux/sockios.h>
30  #include <linux/in.h>
31  #include <linux/errno.h>
32  #include <linux/interrupt.h>
33  #include <linux/if_ether.h>
34  #include <linux/inet.h>
35  #include <linux/netdevice.h>
36  #include <linux/etherdevice.h>
37  #include "ip.h"
38  #include "route.h"
```

```
39  #include <linux/skbuff.h>
40  #include "sock.h"
41  #include "arp.h"


42  /*
43   *  Device multicast list maintenance. This knows about such little matters as promiscuous mode and
44   *  converting from the list to the array the drivers use. At least until I fix the drivers up.
45   *
46   *  This is used both by IP and by the user level maintenance functions. Unlike BSD we maintain a usage count
47   *  on a given multicast address so that a casual user application can add/delete multicasts used by protocols
48   *  without doing damage to the protocols when it deletes the entries. It also helps IP as it tracks overlapping
49   *  maps.
50   */
```

igmp.c中ip_mc_filter_add函数在对设备维护的多播地址进行操作时,在完成从IP多播地址到MAC地址的映射后，调用了dev_mc_add函数完成对设备维护多播地址的更新，下面即是dev_mcast.c中dev_mc_add函数的实现。

```
112 /*
113  *  Add a device level multicast
114  */

115 void dev_mc_add(struct device *dev, void *addr, int alen, int newonly)
116 {
```

参数dev表示对应的网络设备，addr表示MAC多播地址，alen表示MAC地址长度，newonly参数在调用时被简单设置为0，该参数表示的意义根据下文代码实现的意义来看，表示如果存在相同地址，是否增加已有地址的使用计数，还是不进行任何操作，换句话说，newonly表示只有加入的多播地址是一个全新的地址时，才进行响应的操作。由于dev_mc_add函数被调用的目的就是对新加入的多播地址进行设备层的添加，所以下面的代码主要就是操作device结构中mc_list字段指向多播MAC地址链表,诚如前文中对IGMP协议的说明,设备维护多播MAC地址列表中每个元素都是一个dev_mc_list结构，为便于读者理解，此处再次给出该结构定义：

```
/*include/linux/netdevice.h*/
41  struct dev_mc_list
42  {
43      struct dev_mc_list *next;
44      char dmi_addr[MAX_ADDR_LEN];
```

```
45      unsigned short dmi_addrlen;
46      unsigned short dmi_users;
47  };
```

结合如上dev_mc_list结构的定义，下面的代码就很容易理解，device结构涉及到对应设备多播MAC地址列表的字段有二：其一即dev_mc_list结构类型的链表，由m_list字段指向；其二为多播地址的数目，由mc_count字段表示。

```
117     struct dev_mc_list *dmi;
118     for(dmi=dev->mc_list;dmi!=NULL;dmi=dmi->next)
119     {
120         if(memcmp(dmi->dmi_addr,addr,dmi->dmi_addrlen)==0 &&
                                        dmi->dmi_addrlen==alen)
121         {
122             if(!newonly)
123                 dmi->dmi_users++;
124             return;
125         }
126     }
127     dmi=(struct dev_mc_list *)kmalloc(sizeof(*dmi),GFP_KERNEL);
128     if(dmi==NULL)
129         return; /* GFP_KERNEL so can't happen anyway */
130     memcpy(dmi->dmi_addr, addr, alen);
131     dmi->dmi_addrlen=alen;
132     dmi->next=dev->mc_list;
133     dmi->dmi_users=1;
134     dev->mc_list=dmi;
135     dev->mc_count++;
136     dev_mc_upload(dev);
137 }
```

118-126行代码对device结构中mc_list字段指向的多播地址列表进行查询,检查是否有相同的多播地址已经加入到列表中，如果存在，则根据newonly参数的设置，决定是仅仅增加已有地址的使用计数，还是不进行任何操作的返回。

代码执行到127行，表示这是一个全新的多播地址，此时分配一个新的dev_mc_list结构，插入到有mc_list指向的列表首部，最后调用dev_mc_upload函数重新启动设备，从而使新加入的多播地址生效。下面我们就对dev_mc_upload函数进行分析，注意函数参数就是对应的设备结构表示。

```
51  /*
52   *  Update the multicast list into the physical NIC controller.
53   */

54  void dev_mc_upload(struct device *dev)
55  {
```

注意dev_mc_upload函数被调用的目的就是让新的多播地址的更新得到生效或被设备使用。这个“新的多播地址的更新”既包括进行的多播地址的加入，也包括原有多播地址的删除，所以在dev_mc_delete中，在删除一个原有的多播地址后，为了让新的设置生效，也调用该函数进行处理。从下文本函数的实现来看，主要是通过调用device结构set_multicast_list函数指针指向的函数完成对这些新的设置的响应。而set_multicast_list指向的函数是由网络设备驱动程序提供的，因为新的设置的生效需要重新启动设备，即如需要涉及到设备相关寄存器操作，而这只有对应设备驱动程序可以完成。

```
56      struct dev_mc_list *dmi;
57      char *data, *tmp;

58      /* Don't do anything till we up the interface
59         [dev_open will call this function so the list will
60          stay sane] */

61      if(!(dev->flags&IFF_UP))
62          return;
```

如果设备还没有启动工作，那么就无需进行任何操作，因为在设备启动工作时，我们刚才新加入的多播地址会自动生效，而对于设备已经处于工作状态而言，如果不重新启动设备，就无法让我们新加入的多播地址生效，这才需要进行如下的处理。

```
63      /* Devices with no set multicast don't get set */
64      if(dev->set_multicast_list==NULL)
65          return;
```

如果驱动程序没有提供相应的多播地址设置函数，则简单返回，因为这个新的多播地址设置生效必须由驱动程序配合才能实现，如果驱动程序没有提供这个功能，那么从底层上就不支持多播地址的变动性。

```
66      /* Promiscuous is promiscuous - so no filter needed */
67      if(dev->flags&IFF_PROMISC)
68      {
69          dev->set_multicast_list(dev, -1, NULL);
70          return;
71      }
```

对于混杂模式，网络设备接受所有的数据包，无需进行数据包过滤设置。

```
72      if(dev->mc_count==0)
73      {
74          dev->set_multicast_list(dev,0,NULL);
75          return;
```

```
76      }
```

device结构中set_multicast_list指针指向的函数第二个参数表示多播地址个数，第三个参数表示具体的多播地址，这些地址紧密排列，set_multicast_list指向的函数将根据第二个参数指定的多播地址的个数，依次对第三个参数指向的地址列表进行处理。如果第二个参数为0，则表示当前不使用多播，换句话说，网络设备将被设置成为丢弃所有多播数据包（根本不对多播数据包进行接收）。

```
77      data=kmalloc(dev->mc_count*dev->addr_len, GFP_KERNEL);
78      if(data==NULL)
79      {
80          printk("Unable to get memory to set multicast list on %s\n",dev->name);
81          return;
82      }
83      for(tmp = data, dmi=dev->mc_list;dmi!=NULL;dmi=dmi->next)
84      {
85          memcpy(tmp,dmi->dmi_addr, dmi->dmi_addrlen);
86          tmp+=dev->addr_len;
87      }
88      dev->set_multicast_list(dev,dev->mc_count,data);
89      kfree(data);
90  }
```

77-88行代码完成对新的多播列表的处理，代码实现很简单，因为复杂的工作都被屏蔽在由set_multicast_list指向的函数中了，如上文所述，这个函数将有网络设备驱动程序提供。实现的工作是根据具体硬件对多播地址的设置方式，对每个MAC多播地址进行硬件指定的计算（一般计算得到一个比特位用于设置硬件寄存器中对应比特位），并配置多播相关寄存器，完成对新的设置的响应，这个过程中需要暂时停止网络设备的工作，在配置完成后，重新启动，从而使新的设置生效。具体的情况网络接收设备相关。

```
91  /*
92   *  Delete a device level multicast
93   */

94  void dev_mc_delete(struct device *dev, void *addr, int alen, int all)
95  {
```

此处对于参数all需要进行说明，该函数从下面代码（102行）含义来看，表示是否对device结构中mc_list字段指向的所有多播地址进行检查，如果设置为1，则表示对所有地址进行检查；否则当检测到一个匹配后，继续进行剩余地址的检测。这种对所有地址的检测针对地址列表中存在重复项的情况，如果在插入地址时不允许这种重复情况，则无需这种对所有地址的检查。这个参数可以对可能的列表被“污染”（具有重复项）的情况进行修复。当前实现该函数被调用时all参数被设置为0，因为mc_list指向的地址列表在插入时，不允许相同地址项存在。对于相同地址的插入，通过增加使用计数的方式完成。

```
96      struct dev_mc_list **dmi;
97      for(dmi=&dev->mc_list;*dmi!=NULL;dmi=&(*dmi)->next)
98      {
99          if(memcmp((*dmi)->dmi_addr,addr,(*dmi)->dmi_addrlen)==0 &&
                        alen==(*dmi)->dmi_addrlen)
100         {
101             struct dev_mc_list *tmp= *dmi;
102             if(--(*dmi)->dmi_users && !all)
103                 return;
104             *dmi=(*dmi)->next;
105             dev->mc_count--;
106             kfree_s(tmp,sizeof(*tmp));
107             dev_mc_upload(dev);
108             return;
109         }
110     }
111 }
```

dev_mc_delete函数删除一个多播MAC地址，最后也是调用dev_mc_upload函数使新的设置生效。函数实现主要还是对device结构中mc_list, mc_count字段进行更新。代码比较简单，故不再阐述。

dev_mcast.c中定义的最后一个函数是dev_mc_discard，该函数完成的功能是对device结构中mc_list字段指向的列表中所有地址进行释放，这个函数在关闭一个设备时被调用，具体的是在dev_close函数（dev.c）中被调用。

```
138 /*
139  *  Discard multicast list when a device is downed
140  */

141 void dev_mc_discard(struct device *dev)
142 {
143     while(dev->mc_list!=NULL)
144     {
145         struct dev_mc_list *tmp=dev->mc_list;
146         dev->mc_list=dev->mc_list->next;
147         kfree_s(tmp,sizeof(*tmp));
148     }
149     dev->mc_count=0;
150 }
```

dev_mc_discard函数实现遍历device结构mc_list字段指向的地址列表，对其中每个地址对应的dev_mc_list结构调用kfree_s函数进行释放，最后更新mc_count字段值为0。

2.15 net/inet/snmp.h头文件

在完成对IGMP协议实现文件的介绍后，我们下面继续对SNMP协议进行介绍，SNMP称为组管理协议（Simple Network Management Protocol）。正式定义文档为RFC1157。文献[1]中第25章对SNMP协议有比较详细的介绍。此处不再对该协议本身进行说明，只给出本版本网络实现中对SNMP协议的具体实现。SNMP协议实现文件是net/inet/snmp.h，这个文件定义TCP，UDP，ICMP，IP协议的管理信息库（MIB：Management Information Base）。所谓管理信息库，或者M I B，就是所有代理进程包含的、并且能够被管理进程进行查询和设置的信息的集合。所以snmp.h文件主要是针对各协议的统计信息字段集合的定义。

```
1   /*
2    *
3    *      SNMP MIB entries for the IP subsystem.
4    *
5    *      Alan Cox <gw4pts@gw4pts.ampr.org>
6    *
7    *      We don't chose to implement SNMP in the kernel (this would
8    *      be silly as SNMP is a pain in the backside in places). We do
9    *      however need to collect the MIB statistics and export them
10   *      out of /proc (eventually)
11   *
12   *      This program is free software; you can redistribute it and/or
13   *      modify it under the terms of the GNU General Public License
14   *      as published by the Free Software Foundation; either version
15   *      2 of the License, or (at your option) any later version.
16   *
17   */

18  #ifndef _SNMP_H
19  #define _SNMP_H

20  /*
21   *  We use all unsigned longs. Linux will soon be so reliable that even these
22   *  will rapidly get too small 8-). Seriously consider the IpInReceives count
23   *  on the 20Gb/s + networks people expect in a few years time!
24   */

25  struct ip_mib
26  {
27      unsigned long   IpForwarding;
28      unsigned long   IpDefaultTTL;
29      unsigned long   IpInReceives;
30      unsigned long   IpInHdrErrors;
31      unsigned long   IpInAddrErrors;
32      unsigned long   IpForwDatagrams;
```

```
33      unsigned long   IpInUnknownProtos;
34      unsigned long   IpInDiscards;
35      unsigned long   IpInDelivers;
36      unsigned long   IpOutRequests;
37      unsigned long   IpOutDiscards;
38      unsigned long   IpOutNoRoutes;
39      unsigned long   IpReasmTimeout;
40      unsigned long   IpReasmReqds;
41      unsigned long   IpReasmOKs;
42      unsigned long   IpReasmFails;
43      unsigned long   IpFragOKs;
44      unsigned long   IpFragFails;
45      unsigned long   IpFragCreates;
46  };


47  struct icmp_mib
48  {
49      unsigned long   IcmpInMsgs;
50      unsigned long   IcmpInErrors;
51      unsigned long   IcmpInDestUnreachs;
52      unsigned long   IcmpInTimeExcds;
53      unsigned long   IcmpInParmProbs;
54      unsigned long   IcmpInSrcQuenchs;
55      unsigned long   IcmpInRedirects;
56      unsigned long   IcmpInEchos;
57      unsigned long   IcmpInEchoReps;
58      unsigned long   IcmpInTimestamps;
59      unsigned long   IcmpInTimestampReps;
60      unsigned long   IcmpInAddrMasks;
61      unsigned long   IcmpInAddrMaskReps;
62      unsigned long   IcmpOutMsgs;
63      unsigned long   IcmpOutErrors;
64      unsigned long   IcmpOutDestUnreachs;
65      unsigned long   IcmpOutTimeExcds;
66      unsigned long   IcmpOutParmProbs;
67      unsigned long   IcmpOutSrcQuenchs;
68      unsigned long   IcmpOutRedirects;
69      unsigned long   IcmpOutEchos;
70      unsigned long   IcmpOutEchoReps;
71      unsigned long   IcmpOutTimestamps;
72      unsigned long   IcmpOutTimestampReps;
73      unsigned long   IcmpOutAddrMasks;
74      unsigned long   IcmpOutAddrMaskReps;
```

```
75  };

76  struct tcp_mib
77  {
78      unsigned long   TcpRtoAlgorithm;
79      unsigned long   TcpRtoMin;
80      unsigned long   TcpRtoMax;
81      unsigned long   TcpMaxConn;
82      unsigned long   TcpActiveOpens;
83      unsigned long   TcpPassiveOpens;
84      unsigned long   TcpAttemptFails;
85      unsigned long   TcpEstabResets;
86      unsigned long   TcpCurrEstab;
87      unsigned long   TcpInSegs;
88      unsigned long   TcpOutSegs;
89      unsigned long   TcpRetransSegs;
90  };

91  struct udp_mib
92  {
93      unsigned long   UdpInDatagrams;
94      unsigned long   UdpNoPorts;
95      unsigned long   UdpInErrors;
96      unsigned long   UdpOutDatagrams;
97  };


98  #endif
```

此处不再对该文件进行讨论，读者可参考文献[1]或者RFC1157理解以上字段的意义。本版本网络实现只是对以上字段进行了更新，但并未真正实现SNMP协议。所以此处不再对此进行阐述。

在完成对TCP，UDP，ICMP，IGMP，SNMP协议的说明后，我们可以进入网络层进行分析，不过在此之前要对protocol.c和protocol.h文件进行说明。这一对文件定义了网络层协议（一般就是IP协议）模块可调用的上层协议的结构和全局变量。通常对于网络层模块而言，在完成本层次的处理后，需要根据数据包所使用上层协议调用上层相关协议函数将数据包传递给上层进行处理。那么这个上层函数就必须首先被网络层模块知道，这样才谈得上被调用。本版本网络栈实现上将这些上层协议组织成链表的形式。比如说IP协议模块在完成其自身对于数据包的处理后，根据IP首部中上层协议字段值，遍历如上链表，查找到一个合适的元素，从而从该元素所表示的结构中调用指定函数进行处理，完成数据包从网路层到传输层得传递。此处“该元素所表示的结构”即inet_protocol结构，该结构定义在net/inet/protocol.h中。而net/inet/protocol.c中就将上层协议组织成一个链表的形式。注意此处“上层协议”

是指使用IP协议进行封装的协议，包括TCP，UDP，ICMP，IGMP。虽然一般我们将ICMP，IGMP协议当作是网络层协议，但由于他们发送数据时仍然需要使用IP协议进行封装，在完成IP模块处理后，才会调用它们各自的处理函数，所以此处ICMP，IGMP也被包括在“上层协议”中。

2.16 net/inet/protocol.h头文件

```
1   /*
2    * INET     An implementation of the TCP/IP protocol suite for the LINUX
3    *      operating system.  INET is implemented using the  BSD Socket
4    *      interface as the means of communication with the user level.
5    *
6    *      Definitions for the protocol dispatcher.
7    *
8    * Version: @(#)protocol.h  1.0.2   05/07/93
9    *
10   * Author:  Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
11   *
12   *      This program is free software; you can redistribute it and/or
13   *      modify it under the terms of the GNU General Public License
14   *      as published by the Free Software Foundation; either version
15   *      2 of the License, or (at your option) any later version.
16   *
17   *  Changes:
18   *      Alan Cox    :   Added a name field and a frag handler
19   *                  field for later.
20   */

21  #ifndef _PROTOCOL_H
22  #define _PROTOCOL_H


23  #define MAX_INET_PROTOS 32      /* Must be a power of 2     */


24  /* This is used to register protocols. */
25  struct inet_protocol {
26    int           (*handler)(struct sk_buff *skb, struct device *dev,
27                     struct options *opt, unsigned long daddr,
28                     unsigned short len, unsigned long saddr,
29                     int redo, struct inet_protocol *protocol);
30    int           (*frag_handler)(struct sk_buff *skb, struct device *dev,
31                     struct options *opt, unsigned long daddr,
32                     unsigned short len, unsigned long saddr,
33                     int redo, struct inet_protocol *protocol);
```

```
34    void          (*err_handler)(int err, unsigned char *buff,
35                         unsigned long daddr,
36                         unsigned long saddr,
37                         struct inet_protocol *protocol);
38    struct inet_protocol *next;
39    unsigned char     protocol;
40    unsigned char     copy:1;
41    void          *data;
42    char          *name;
43  };
```

诚如前文中说明，inet_protocol结构被用来表示一个被网络层模块调用的上层协议，这个结构中定义了数据包的正常接收函数指针(handler),错误时调用的函数指针(err_handler)以及其他辅助字段（如protocol表示这个上层协议的协议号，TCP对应6，UDP-17等）。

```
44  extern struct inet_protocol *inet_protocol_base;
45  extern struct inet_protocol *inet_protos[MAX_INET_PROTOS];
```

inet_protocol_base变量以链表的形式管理系统中所有网络层之上的协议(这是从使用的角度来看，包括TCP，UDP，ICMP，IGMP）。而inet_protos则以数组的形式管理这些协议。数组管理方式在查找一个协议时较快。系统首先预先定义由inet_protocol_base变量指向的链表（具体如下protocol.c文件），在内核对网络栈初始化过程中由inet_protocol_base指向的链表生成inet_protos变量表示的数组形式。

```
46  extern void     inet_add_protocol(struct inet_protocol *prot);
47  extern int      inet_del_protocol(struct inet_protocol *prot);
```

在网络栈初始化过程中，由inet_protocol_base指向的链表生成inet_protos表示的数组形式，就是遍历链表中每个元素，对每个元素调用inet_add_protocol函数进行协议复制转移。我们可以给出转化的这段代码，这段代码定义在af_inet.c中的inet_proto_init函数中，我们只摘取部分相关片段列出如下：

```
/*net/inet/af_inet.c*/
1341    void inet_proto_init(struct net_proto *pro)
1342    {
1343        struct inet_protocol *p;

            ……
1366        printk("IP Protocols: ");
1367        for(p = inet_protocol_base; p != NULL;)
1368        {
1369            struct inet_protocol *tmp = (struct inet_protocol *) p->next;
1370            inet_add_protocol(p);
1371            printk("%s%s",p->name,tmp?", ":"\n");
1372            p = tmp;
```

```
1373        }

            ......
1382    }
```

对应inet_del_protocol函数用于从inet_protos数组中删除对应的inet_protocol结构，但是这个inet_protocol结构仍然在inet_protocol_base变量指向系统队列中。
inet_add_protocol和inet_del_protocol函数均定义在如下protocol.c中，下面我们就对protocol.c文件进行分析，读者也可以看到inet_protocol_base这个变量的定义。

```
48  #endif  /* _PROTOCOL_H */
```

2.17 net/inet/protocol.c头文件

```
1   /*
2    * INET     An implementation of the TCP/IP protocol suite for the LINUX
3    *      operating system.  INET is implemented using the  BSD Socket
4    *      interface as the means of communication with the user level.
5    *
6    *      INET protocol dispatch tables.
7    *
8    * Version: @(#)protocol.c  1.0.5   05/25/93
9    *
10   * Authors: Ross Biro, <bir7@leland.Stanford.Edu>
11   *      Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
12   *
13   * Fixes:
14   *      Alan Cox    : Ahah! udp icmp errors don't work because
15   *                udp_err is never called!
16   *      Alan Cox    : Added new fields for init and ready for
17   *                proper fragmentation (_NO_ 4K limits!)
18   *
19   *      This program is free software; you can redistribute it and/or
20   *      modify it under the terms of the GNU General Public License
21   *      as published by the Free Software Foundation; either version
22   *      2 of the License, or (at your option) any later version.
23   */
24  #include <asm/segment.h>
25  #include <asm/system.h>
26  #include <linux/types.h>
27  #include <linux/kernel.h>
28  #include <linux/sched.h>
29  #include <linux/string.h>
30  #include <linux/config.h>
```

```
31  #include <linux/socket.h>
32  #include <linux/in.h>
33  #include <linux/inet.h>
34  #include <linux/netdevice.h>
35  #include <linux/timer.h>
36  #include "ip.h"
37  #include "protocol.h"
38  #include "tcp.h"
39  #include <linux/skbuff.h>
40  #include "sock.h"
41  #include "icmp.h"
42  #include "udp.h"
43  #include <linux/igmp.h>
```

下面就是各上层协议对应inet_protocol结构的定义：

```
//TCP协议
44  static struct inet_protocol tcp_protocol = {
45    tcp_rcv,       /* TCP handler      */
46    NULL,          /* No fragment handler (and won't be for a long time) */
47    tcp_err,       /* TCP error control    */
48    NULL,          /* next         */
49    IPPROTO_TCP,       /* protocol ID      */
50    0,             /* copy         */
51    NULL,          /* data         */
52    "TCP"          /* name         */
53  };

//UDP协议
54  static struct inet_protocol udp_protocol = {
55    udp_rcv,       /* UDP handler      */
56    NULL,          /* Will be UDP fraglist handler */
57    udp_err,       /* UDP error control    */
58    &tcp_protocol,     /* next         */
59    IPPROTO_UDP,       /* protocol ID      */
60    0,             /* copy         */
61    NULL,          /* data         */
62    "UDP"          /* name         */
63  };

//ICMP协议
64  static struct inet_protocol icmp_protocol = {
65    icmp_rcv,      /* ICMP handler     */
66    NULL,          /* ICMP never fragments anyway */
67    NULL,          /* ICMP error control   */
```

```
68    &udp_protocol,     /* next         */
69    IPPROTO_ICMP,      /* protocol ID      */
70    0,             /* copy         */
71    NULL,          /* data         */
72    "ICMP"         /* name         */
73  };

74  #ifndef CONFIG_IP_MULTICAST
75  struct inet_protocol *inet_protocol_base = &icmp_protocol;
76  #else
//IGMP协议
77  static struct inet_protocol igmp_protocol = {
78    igmp_rcv,      /* IGMP handler     */
79    NULL,          /* IGMP never fragments anyway */
80    NULL,          /* IGMP error control   */
81    &icmp_protocol,    /* next         */
82    IPPROTO_IGMP,      /* protocol ID      */
83    0,             /* copy         */
84    NULL,          /* data         */
85    "IGMP"         /* name         */
86  };

87  struct inet_protocol *inet_protocol_base = &igmp_protocol;
88  #endif
```

对于IGMP协议只有CONFIG_IP_MULTICAST宏定义的情况下才包括到系统队列中，即网络栈首先才将IGMP协议实现包括进来。以上这段代码很容易理解。所有inet_protocol结构都被组织中链表的形式，由inet_protocol_base变量指向这个链表的头部。读者可结合inet_protocol结构的定义对以上各协议对应字段的赋值进行查看。在本书前文中，我们说tcp_rcv，udp_rcv,icmp_rcv,igmp_rcv分别是TCP，UDP，ICMP，IGMP协议数据包接收的总入口函数，在此处就可以得到验证。网络层模块完成本层处理后，通过网络层协议首部中相关字段查找上层使用协议，然后根据该上层协议的协议号查找inet_protos数组，找到对应的inet_protocol结构，调用该结构handler函数指针字段指向的函数将数据包传递给上层协议继续进行处理，对于TCP协议而言，这个handler字段指向的函数就是tcp_rcv函数（45行）。其他协议类似。

```
89  struct inet_protocol *inet_protos[MAX_INET_PROTOS] = {
90    NULL
91  };
```

虽然定义了inet_protocol_base变量指向系统中所有定义的这些“上层协议”（此处再次说明，“上层协议”包括的范围是从实现的角度来考虑的，所以ICMP，IGMP都被归为上层协议，下文中如无特殊说明，凡是谈到“上层协议”，都包括ICMP，IGMP协议），但网络层模块对所使用“上层协议”对应inet_protocol结构的查找并不使用这个系统队列，而是通过

inet_protos数组查找。每个“上层协议”根据协议号插入到数组中的对应位置，如TCP协议即插入到索引号为6的数组对应位置上。如此网络层模块在进行“上层协议”查找时，可以有较高的查找效率。在上文中对protocol.h文件的分析中，我们提到需要调用inet_add_protocol函数将由inet_protocol_base指向的链表中inet_protocol结构插入到由inet_protos表示的数组中。从87行（或者75行-当不包括IGMP协议时）和89行我们也可看出，inet_protocol_base变量是静态初始化的，但inet_protos初始时为空，所以需要在网络栈初始化时对inet_protos进行初始化，这个数组元素的来看就是inet_protocol_base指向的队列，而inet_add_protocol函数就是“桥梁”，相关调用代码（调用inet_add_protocol进行转化）在上文对protocol.h文件说明中已经给出。下面我们就来看inet_add_protocol函数的实现。

```
103 void
104 inet_add_protocol(struct inet_protocol *prot)
105 {
106   unsigned char hash;
107   struct inet_protocol *p2;

108   hash = prot->protocol & (MAX_INET_PROTOS - 1);
109   prot ->next = inet_protos[hash];
110   inet_protos[hash] = prot;
111   prot->copy = 0;

112   /* Set the copy bit if we need to. */
113   p2 = (struct inet_protocol *) prot->next;
114   while(p2 != NULL) {
115     if (p2->protocol == prot->protocol) {
116         prot->copy = 1;
117         break;
118     }
119     p2 = (struct inet_protocol *) prot->next;
120   }
121 }
```

108行以协议号为索引计算在inet_protos数组中的对应位置。109-111行将由参数表示的inet_protocol结构插入到对应位置上，并将inet_protocol结构中copy字段赋值为0，该字段表示在inet_protos数组相同位置上是否存在相同协议号的inet_protocol结构。这一点可以从114-120行代码看出，这段代码就是检测重复的情况，如果存在一个相同协议号的inet_protocol结构，则将结构中copy字段设置为1。注意这段代码的操作方式，如果inet_protos数组相同位置上存在多个相同协议号的inet_protocol结构（此时这些inet_protocl结构构成队列形式，队列首部由协议号索引的数据元素指向），则最后一个inet_protocol结构的copy字段依然为0，其他所有inet_protocol结构中copy字段设置为1。这种方式可以得到具有相同协议号的inet_protocol结构到何处截止，如此在进行遍历时可以及时跳出循环。119行语句有问题，正确的语句应该如下：

```
119     p2 = (struct inet_protocol *) p2->next;
```

inet_add_protocol使用于添加一个inet_protocol结构到inet_protos数组中的，而inet_del_protocol则是进行相反的操作，从inet_protos数组删除一个inet_protocol结构。原则上删除一个inet_protocol较为简单，但涉及到copy字段的设置时，还是需要进行一番检查。inet_del_protocol函数就是完成这两个工作：其一从inet_protos数组中删除参数指定的inet_protocol结构；其二按情况更新余下具有相同协议号inet_protocol结构中copy字段值。

```
122 int
123 inet_del_protocol(struct inet_protocol *prot)
124 {
125   struct inet_protocol *p;
126   struct inet_protocol *lp = NULL;
127   unsigned char hash;

128   hash = prot->protocol & (MAX_INET_PROTOS - 1);
129   if (prot == inet_protos[hash]) {
130     inet_protos[hash] = (struct inet_protocol *) inet_protos[hash]->next;
131     return(0);
132   }
```

128-132行代码处理被删除元素是队列中第一个元素的情况，对于这种情况最为简单，直接删除即可，无需对其他元素copy字段的更新。只有被删除元素是具有相同协议号的最后一个元素时，才需要对之前的倒数第二个元素进行copy字段更新（从1设置为0）。

```
133   p = (struct inet_protocol *) inet_protos[hash];
134   while(p != NULL) {
135     /*
136      * We have to worry if the protocol being deleted is
137      * the last one on the list, then we may need to reset
138      * someone's copied bit.
139      */
140     if (p->next != NULL && p->next == prot) {
141         /*
142          * if we are the last one with this protocol and
143          * there is a previous one, reset its copy bit.
144          */
145          if (p->copy == 0 && lp != NULL) lp->copy = 0;
146          p->next = prot->next;
147          return(0);
148     }
```

133-148行代码处理被删除元素非第一个元素的情况，145行判断是否为最后一个元素（其对应copy字段为0），如果是，则将其之前的具有相同协议号的元素的copy字段值设置为0。因为删除当前元素后，其之前的元素就变成了最后一个元素。146行对参数表示元素进行删除。

145行代码有误，正确代码如下：

```
145          if (prot->copy == 0 && lp != NULL) lp->copy = 0;
```

```
149     if (p->next != NULL && p->next->protocol == prot->protocol) {
150         lp = p;
151     }
```

149-151行代码对lp变量进行更新，让这个变量始终指向具有相同协议号在被删除元素之前的那个元素，以便在检测到被删除元素是最后一个具有相同协议号的元素后，对copy字段方便的进行更新。

```
152     p = (struct inet_protocol *) p->next;
153   }
154   return(-1);
155 }
```

除了inet_add_protocol和inet_del_protocol函数外，protocol.c文件还定义了另外一个函数，下面我们就对该函数进行分析已完成对protocol.c文件的说明。

```
92  struct inet_protocol *
93  inet_get_protocol(unsigned char prot)
94  {
95    unsigned char hash;
96    struct inet_protocol *p;

97    hash = prot & (MAX_INET_PROTOS - 1);
98    for (p = inet_protos[hash] ; p != NULL; p=p->next) {
99      if (p->protocol == prot) return((struct inet_protocol *) p);
100   }
101   return(NULL);
102 }
```

inet_get_protocol函数根据参数表示的协议号返回一个对应的inet_protocol结构。函数实现非常简单。注意由于inet_protos数据容量有限，所以有可能具有不同协议号的inet_protocol结构经过97行的计算方式后，被组织到同一个队列中，所以99行需要对这种情况进行检查，只有协议号相吻合后，才返回对应的inet_protocol结构。

至此我们基本完成网络层之上所有方面的介绍，还有一个proc.c文件是支持proc文件系统的，这个文件系统以文件方式向用户提供内核信息，或者用户通过改变文件内容改变内核的相关设置。由于proc.c文件主要是从内核结构复制字段值到用户缓冲区，所以不再对此文件进行具体分析，如下只是简单列出该文件源代码。

在 Linux 文件系统中，存在一个特殊的目录/proc，该目录下文件表示一些内核配置的信息，我们可以通过直接修改文件的方式对内核参数进行配置。下面要介绍的这个文件就实现为网络配置或者输出网络栈信息的/proc 目录下文件。

2.18 net/inet/proc.c 文件

```
1    /*
2     * INET        An implementation of the TCP/IP protocol suite for the LINUX
3     *        operating system.    INET is implemented using the    BSD Socket
4     *        interface as the means of communication with the user level.
5     *
6     *        This file implements the various access functions for the
7     *        PROC file system.    It is mainly used for debugging and
8     *        statistics.
9     *
10    * Version:    @(#)proc.c    1.0.5    05/27/93
11    *
12    * Authors:    Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
13    *        Gerald J. Heim, <heim@peanuts.informatik.uni-tuebingen.de>
14    *        Fred Baumgarten, <dc6iq@insu1.etec.uni-karlsruhe.de>
15    *        Erik Schoenfelder, <schoenfr@ibr.cs.tu-bs.de>
16    *
17    * Fixes:
18    *        Alan Cox:    UDP sockets show the rxqueue/txqueue
19    *                     using hint flag for the netinfo.
20    *   Pauline Middelink :    identd support
21    *        Alan Cox:    Make /proc safer.
22    *   Erik Schoenfelder :    /proc/net/snmp
23    *        Alan Cox:    Handle dead sockets properly.
24    *
25    *        This program is free software; you can redistribute it and/or
26    *        modify it under the terms of the GNU General Public License
27    *        as published by the Free Software Foundation; either version
28    *        2 of the License, or (at your option) any later version.
29    */
30   #include <asm/system.h>
31   #include <linux/autoconf.h>
32   #include <linux/sched.h>
33   #include <linux/socket.h>
34   #include <linux/net.h>
35   #include <linux/un.h>
36   #include <linux/in.h>
37   #include <linux/param.h>
38   #include <linux/inet.h>
39   #include <linux/netdevice.h>
```

```
40  #include "ip.h"
41  #include "icmp.h"
42  #include "protocol.h"
43  #include "tcp.h"
44  #include "udp.h"
45  #include <linux/skbuff.h>
46  #include "sock.h"
47  #include "raw.h"

48  /*
49   * Get__netinfo returns the length of that string.
50   *
51   * KNOWN BUGS
52   *    As in get_unix_netinfo, the buffer might be too small. If this
53   *    happens, get__netinfo returns only part of the available infos.
54   */
55  static int
56  get__netinfo(struct proto *pro, char *buffer, int format, char **start, off_t offset, int length)
57  {
```

我们刚刚介绍完 ip_fw.c 文件，其中 ip_chain_procinfo 函数用于获取防火墙规则链信息。我们查看 get__netinfo 函数参数形式，与 ip_chain_procinfo 函数有些类似，其中 start，offset，length 参数含义相同，此处不再说明，读者可查看前文中对 ip_chain_procinfo 函数的说明。第一个参数 pro 是一个 proto 类型的参数，该参数表示获取何种协议的信息。至于 format 参数的含义我们在下文中使用该参数的代码时再进行说明。

```
58          struct sock **s_array;
59          struct sock *sp;
60          int i;
61          int timer_active;
62          unsigned long    dest, src;
63          unsigned short destp, srcp;
64          int len=0;
65          off_t pos=0;
66          off_t begin=0;

67          s_array = pro->sock_array;
```

每个 proto 类型变量表示一个传输层协议操作函数集合，如 tcp_prot 表示 TCP 协议操作函数集合；udp_prot, icmp_prot, raw_prot 表示的意义类似。proto 结构中 sock_array 字段是一个 sock 类型指针的数组。每个数组元素又指向一个 sock 类型结构队列。所有使用 TCP 协议进行通信的套接字对应的 sock 结构均挂接在 tcp_prot 所在 proto 结构的 sock_array 数组队列中。同理 UDP 协议，ICMP 协议。下面的代码就是对指定协议的 sock_array 数组中每个元素执行的队列进行遍历，抽出需要返回的信息。

```
68          len+=sprintf(buffer, "sl  local_address rem_address   st tx_queue rx_queue tr
tm->when uid\n");
69      /*
70       *  This was very pretty but didn't work when a socket is destroyed at the wrong moment
71       *  (eg a syn recv socket getting a reset), or a memory timer destroy. Instead of playing
72       *  with timers we just concede defeat and cli().
73       */
74          for(i = 0; i < SOCK_ARRAY_SIZE; i++)
75          {
76              cli();
77              sp = s_array[i];
78              while(sp != NULL)
79              {
80                  dest    = sp->daddr;
81                  src     = sp->saddr;
82                  destp = sp->dummy_th.dest;
83                  srcp    = sp->dummy_th.source;

84                  /* Since we are Little Endian we need to swap the bytes :-( */
85                  destp = ntohs(destp);
86                  srcp    = ntohs(srcp);
87                  timer_active = del_timer(&sp->timer);
88                  if (!timer_active)
89                      sp->timer.expires = 0;
90                  len+=sprintf(buffer+len, "%2d: %08lX:%04X %08lX:%04X %02X
%08lX:%08lX %02X:%08lX %08X %d %d\n",
91                      i, src, srcp, dest, destp, sp->state,
92                      format==0?sp->write_seq-sp->rcv_ack_seq:sp->rmem_alloc,
93                      format==0?sp->acked_seq-sp->copied_seq:sp->wmem_alloc,
```

注意 92-93 行对 format 参数不同设置所返回信息的不同，此处没有什么好说的，代码就是这样实现的，我们可以按字面解释为当 format 等于 0 时，如何如何，对于 format 不同值的处理是该实现决定的。其他调用该函数的代码只需要知道这一点即可，如何实现随意。从下文的调用环境来看，获取 TCP 协议信息时，format 参数被设置为 0，其他协议 UDP，ICMP，设置为 1，由此我们也可推出为何此处要使用 format 这样一个参数，因为对于 UDP，ICMP 协议，他们不提供可靠性数据传输，自然就没有序列号一说，所以进行信息获取时只能返回本机已分配的读写缓冲区大小；而对于 TCP 协议，则可以返回序列号差值。

```
94                      timer_active, sp->timer.expires, (unsigned) sp->retransmits,
95                      sp->socket?SOCK_INODE(sp->socket)->i_uid:0,
96                      timer_active?sp->timeout:0);
97                  if (timer_active)
98                      add_timer(&sp->timer);
```

```
99                      /*
100                       * All sockets with (port mod SOCK_ARRAY_SIZE) = i
101                       * are kept in sock_array[i], so we must follow the
102                       * 'next' link to get them all.
103                       */
104                     sp = sp->next;
105                     pos=begin+len;
106                     if(pos<offset)
107                     {
108                             len=0;
109                             begin=pos;
110                     }
111                     if(pos>offset+length)
112                             break;
113             }
114             sti();/* We only turn interrupts back on for a moment, but because the interrupt queues anything built up
115                              before this will clear before we jump back and cli, so it's not as bad as it looks */
116             if(pos>offset+length)
117                     break;
118     }
119     *start=buffer+(offset-begin);
120     len-=(offset-begin);
121     if(len>length)
122             len=length;
123     return len;
124 }
```

get__netinfo 函数区别于 ip_chain_procinfo 函数之处仅在于所获取信息的不同，其他实现代码基本一致，此处不再做说明，读者可参考前文中对 ip_chain_procinfo 函数的相关分析说明。

```
125 int tcp_get_info(char *buffer, char **start, off_t offset, int length)
126 {
127     return get__netinfo(&tcp_prot, buffer,0, start, offset, length);
128 }
```

```
129 int udp_get_info(char *buffer, char **start, off_t offset, int length)
130 {
131     return get__netinfo(&udp_prot, buffer,1, start, offset, length);
132 }
```

```
133  int raw_get_info(char *buffer, char **start, off_t offset, int length)
134  {
135          return get__netinfo(&raw_prot, buffer,1, start, offset, length);
136  }
```

125-136 行定义的三个函数实现为对不同协议信息进行获取，具体实现上都是对 get__netinfo 函数的封装，注意第一个参数和第三个参数的传入值的不同。第一个参数表示针对哪种协议进行信息获取。第三个参数 format 则是在获取信息时进行微调。具体见上文中 get__netinfo 函数的相关代码。

```
137  /*
138   *    Report socket allocation statistics [mea@utu.fi]
139   */
140  int afinet_get_info(char *buffer, char **start, off_t offset, int length)
141  {
142          /* From    net/socket.c    */
143          extern int socket_get_info(char *, char **, off_t, int);
144          extern struct proto packet_prot;

145          int len    = socket_get_info(buffer,start,offset,length);

146          len += sprintf(buffer+len,"SOCK_ARRAY_SIZE=%d\n",SOCK_ARRAY_SIZE);
147          len += sprintf(buffer+len,"TCP: inuse %d highest %d\n",
148                          tcp_prot.inuse, tcp_prot.highestinuse);
149          len += sprintf(buffer+len,"UDP: inuse %d highest %d\n",
150                          udp_prot.inuse, udp_prot.highestinuse);
151          len += sprintf(buffer+len,"RAW: inuse %d highest %d\n",
152                          raw_prot.inuse, raw_prot.highestinuse);
153          len += sprintf(buffer+len,"PAC: inuse %d highest %d\n",
154                          packet_prot.inuse, packet_prot.highestinuse);
155          *start = buffer + offset;
156          len -= offset;
157          if (len > length)
158                  len = length;
159          return len;
160  }
```

afinet_get_info 函数首先调用 socket_get_info 函数获取信息，此后返回一些全局变量表示的统计信息。socket_get_info 函数定义在 net/socket.c 中，其返回 sockets_in_use 变量值。

```
161  /*
162   *    Called from the PROCfs module. This outputs /proc/net/snmp.
163   */
```

```
164  int snmp_get_info(char *buffer, char **start, off_t offset, int length)
165  {
166          extern struct tcp_mib tcp_statistics;
167          extern struct udp_mib udp_statistics;
168          int len;
169  /*
170      extern unsigned long tcp_rx_miss, tcp_rx_hit1,tcp_rx_hit2;
171  */

172          len = sprintf (buffer,
173                  "Ip: Forwarding DefaultTTL InReceives InHdrErrors InAddrErrors ForwDatagrams InUnknownProtos InDiscards InDelivers OutRequests OutDiscards OutNoRoutes ReasmTimeout ReasmReqds ReasmOKs ReasmFails FragOKs FragFails FragCreates\n"
174                  "Ip: %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu\n",
175                          ip_statistics.IpForwarding, ip_statistics.IpDefaultTTL,
176                          ip_statistics.IpInReceives, ip_statistics.IpInHdrErrors,
177                          ip_statistics.IpInAddrErrors, ip_statistics.IpForwDatagrams,
178                          ip_statistics.IpInUnknownProtos, ip_statistics.IpInDiscards,
179                          ip_statistics.IpInDelivers, ip_statistics.IpOutRequests,
180                          ip_statistics.IpOutDiscards, ip_statistics.IpOutNoRoutes,
181                          ip_statistics.IpReasmTimeout, ip_statistics.IpReasmReqds,
182                          ip_statistics.IpReasmOKs, ip_statistics.IpReasmFails,
183                          ip_statistics.IpFragOKs, ip_statistics.IpFragFails,
184                          ip_statistics.IpFragCreates);

185          len += sprintf (buffer + len,
186                  "Icmp: InMsgs InErrors InDestUnreachs InTimeExcds InParmProbs InSrcQuenchs InRedirects InEchos InEchoReps InTimestamps InTimestampReps InAddrMasks InAddrMaskReps OutMsgs OutErrors OutDestUnreachs OutTimeExcds OutParmProbs OutSrcQuenchs OutRedirects OutEchos OutEchoReps OutTimestamps OutTimestampReps OutAddrMasks OutAddrMaskReps\n"
187                  "Icmp: %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu\n",
188                          icmp_statistics.IcmpInMsgs, icmp_statistics.IcmpInErrors,
189                          icmp_statistics.IcmpInDestUnreachs, icmp_statistics.IcmpInTimeExcds,
190                          icmp_statistics.IcmpInParmProbs, icmp_statistics.IcmpInSrcQuenchs,
191                          icmp_statistics.IcmpInRedirects, icmp_statistics.IcmpInEchos,
192                          icmp_statistics.IcmpInEchoReps, icmp_statistics.IcmpInTimestamps,
193                          icmp_statistics.IcmpInTimestampReps, icmp_statistics.IcmpInAddrMasks,
194                          icmp_statistics.IcmpInAddrMaskReps, icmp_statistics.IcmpOutMsgs,
195                          icmp_statistics.IcmpOutErrors, icmp_statistics.IcmpOutDestUnreachs,
196                          icmp_statistics.IcmpOutTimeExcds, icmp_statistics.IcmpOutParmProbs,
```

```
197                 icmp_statistics.IcmpOutSrcQuenchs, icmp_statistics.IcmpOutRedirects,
198                 icmp_statistics.IcmpOutEchos, icmp_statistics.IcmpOutEchoReps,
199                 icmp_statistics.IcmpOutTimestamps,
icmp_statistics.IcmpOutTimestampReps,
200                 icmp_statistics.IcmpOutAddrMasks, icmp_statistics.IcmpOutAddrMaskReps);

201         len += sprintf (buffer + len,
202             "Tcp: RtoAlgorithm RtoMin RtoMax MaxConn ActiveOpens PassiveOpens
AttemptFails EstabResets CurrEstab InSegs OutSegs RetransSegs\n"
203             "Tcp: %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu %lu\n",
204                 tcp_statistics.TcpRtoAlgorithm, tcp_statistics.TcpRtoMin,
205                 tcp_statistics.TcpRtoMax, tcp_statistics.TcpMaxConn,
206                 tcp_statistics.TcpActiveOpens, tcp_statistics.TcpPassiveOpens,
207                 tcp_statistics.TcpAttemptFails, tcp_statistics.TcpEstabResets,
208                 tcp_statistics.TcpCurrEstab, tcp_statistics.TcpInSegs,
209                 tcp_statistics.TcpOutSegs, tcp_statistics.TcpRetransSegs);

210         len += sprintf (buffer + len,
211             "Udp: InDatagrams NoPorts InErrors OutDatagrams\nUdp: %lu %lu %lu %lu\n",
212                 udp_statistics.UdpInDatagrams, udp_statistics.UdpNoPorts,
213                 udp_statistics.UdpInErrors, udp_statistics.UdpOutDatagrams);
214 /*
215           len += sprintf( buffer + len,
216             "TCP fast path RX:    H2: %ul H1: %ul L: %ul\n",
217                 tcp_rx_hit2,tcp_rx_hit1,tcp_rx_miss);
218 */

219         if (offset >= len)
220         {
221             *start = buffer;
222             return 0;
223         }
224         *start = buffer + offset;
225         len -= offset;
226         if (len > length)
227             len = length;
228         return len;
229 }
```

snmp_get_info 函数返回 SNMP 统计信息，tcp_statictis, udp_statistics, icmp_statistics, ip_statistics 变量分别表示 TCP 协议，UDP 协议，ICMP 协议，IP 协议的 SNMP 统计信息，这些变量对应的类型均定义在 net/inet/snmp.h 中，读者参阅本书前文对 snmp.h 的介绍，即可得知各自的类型，结合这些结构定义，以上代码只是流水账形式的记录。

proc.c 文件小结

该文件提供了/proc 目录下对网络栈进行配置（本版本不支持）和信息获取的接口，用户可以通过查看文件的方式查看内核信息。这种实现方式为用户和操作系统之间提供了一个信息查看的桥梁。对于该文件代码没有进行详细说明，大多是略带而过，因为函数实现思想上都比较简单，只要读者理解本书前文中对相关文件的分析，结合涉及到的相关结构定义，这些函数实现代码不应存在任何理解上的问题。

网络层模块实现，主要是指IP协议实现文件，在该层一个重要的概念就是IP路由：即根据最终目的IP地址选择合适的传输路径。在IP协议实现中，凡是发送数据包，都要查询路由表，选择合适的路由项，确定下一站的地址，由此构造MAC首部，进而将数据包发往下一层（链路层）进行处理。另外防火墙功能也基于网络层实现，在IP协议实现中，也夹杂着大量使用防火墙函数对数据包进行过滤的函数调用。为了便于对IP协议实现的理解，我们首先对这两个方面进行介绍，此后再进入对IP协议实现的分析。

2.19 net/inet/route.h头文件

在include/linux/route.h文件中，我们介绍了rtentry和old_rtentry结构，这两个结构并非用来表示路由表项，而是上层使用这两个结构作为参数对路由表项进行更新。真正的路由表中表项是在net/inet/route.h中定义的rtable结构。下面我们就对rtable结构等进行介绍，不过我们首先查看一下系统路由表的内容及格式，在LINUX终端提示符下敲入‘netstate －rn’或者‘route’，就可以看到系统路由表的内容及显示格式，如下所示：

```
[linux@localhost linux]$ netstat -rn
Kernel IP routing table
Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
192.168.0.0     0.0.0.0         255.255.255.0   U         0 0          0 eth0
169.254.0.0     0.0.0.0         255.255.0.0     U         0 0          0 eth0
127.0.0.0       0.0.0.0         255.0.0.0       U         0 0          0 lo
0.0.0.0         192.168.0.1     0.0.0.0         UG        0 0          0 eth0
[linux@localhost linux]$ _
```

这是一个IP地址为192.168.0.4的内部局域网主机上打印的路由表信息。网关地址为192.168.0.1。

虽然表项的打印内容与底层实际表示结构稍有差别，不过我们还是可以看到主要的信息输出，包括目的IP地址，网关地址（注意如果目的主机在同一链路上，则网关地址打印为0，这与Windows下此时显示了本机接口地址不同），标志位，MSS值，接口名称等等。

```
1   /*
2    * INET     An implementation of the TCP/IP protocol suite for the LINUX
3    *      operating system.  INET  is implemented using the  BSD Socket
4    *      interface as the means of communication with the user level.
5    *
6    *      Definitions for the IP router.
7    *
8    * Version: @(#)route.h 1.0.4    05/27/93
9    *
```

```
10   * Authors: Ross Biro, <bir7@leland.Stanford.Edu>
11   *      Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
12   * Fixes:
13   *      Alan Cox    :   Reformatted. Added ip_rt_local()
14   *      Alan Cox    :   Support for TCP parameters.
15   *
16   *      This program is free software; you can redistribute it and/or
17   *      modify it under the terms of the GNU General Public License
18   *      as published by the Free Software Foundation; either version
19   *      2 of the License, or (at your option) any later version.
20   */
21  #ifndef _ROUTE_H
22  #define _ROUTE_H


23  #include <linux/route.h>


24  /* This is an entry in the IP routing table. */
25  struct rtable
26  {
27      struct rtable       *rt_next; //指向下一个rtable表项
28      unsigned long       rt_dst; //目的地址（对应以上图示中的目的地址）
29      unsigned long       rt_mask; //子网掩码
30      unsigned long       rt_gateway;  //网关地址
31      unsigned char       rt_flags;  //标志位
32      unsigned char       rt_metric;  //度量值（代价值）
33      short           rt_refcnt;  //使用计数
34      unsigned long       rt_use;  //被使用标志
35      unsigned short      rt_mss; //MSS值
36      unsigned long       rt_window; //窗口大小
37      struct device       *rt_dev; //与该路由项绑定的接口
38  };


39  extern void     ip_rt_flush(struct device *dev);
40  extern void     ip_rt_add(short flags, unsigned long addr, unsigned long mask,
41                      unsigned long gw, struct device *dev, unsigned short mss,
unsigned long window);
42  extern struct rtable   *ip_rt_route(unsigned long daddr, struct options *opt,
unsigned long *src_addr);
43  extern struct rtable   *ip_rt_local(unsigned long daddr, struct options *opt,
unsigned long *src_addr);
44  extern int      rt_get_info(char * buffer, char **start, off_t offset, int
```

```
length);
45  extern int      ip_rt_ioctl(unsigned int cmd, void *arg);

46  #endif  /* _ROUTE_H */
```

rtable结构被用以表示一个路由表项，系统路由表即是由许多rtable结构构成的一个链表。以上对于rtable结构字段的注释中很多都是泛泛而谈，如rt_use字段，rt_metric字段究竟表示何意？rt_gateway在目的主机属于相同链路上时，如何赋值？下面我们进入对路由表项进行操作的函数的分析，这些字段的意义自会清楚，首先还是含糊一点比较好，因为字段的含义只有在具体代码使用中才能体现出来。所以要真正理解一个字段的意义，看它如何被使用方是正道。下面我们即进入操作路由表函数集合的分析：net/inet/route.c文件。

2.20 net/inet/route.c文件

```
1   /*
2    * INET     An implementation of the TCP/IP protocol suite for the LINUX
3    *      operating system.  INET is implemented using the  BSD Socket
4    *      interface as the means of communication with the user level.
5    *
6    *      ROUTE - implementation of the IP router.
7    *
8    * Version: @(#)route.c 1.0.14  05/31/93
9    *
10   * Authors: Ross Biro, <bir7@leland.Stanford.Edu>
11   *      Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
12   *      Alan Cox, <gw4pts@gw4pts.ampr.org>
13   *      Linus Torvalds, <Linus.Torvalds@helsinki.fi>
14   *
15   * Fixes:
16   *      Alan Cox    :   Verify area fixes.
17   *      Alan Cox    :   cli() protects routing changes
18   *      Rui Oliveira    :   ICMP routing table updates
19   *      (rco@di.uminho.pt)  Routing table insertion and update
20   *      Linus Torvalds  :   Rewrote bits to be sensible
21   *      Alan Cox    :   Added BSD route gw semantics
22   *      Alan Cox    :   Super /proc >4K
23   *      Alan Cox    :   MTU in route table
24   *      Alan Cox    :   MSS actually. Also added the window
25   *                  clamper.
26   *      Sam Lantinga    :   Fixed route matching in rt_del()
27   *
28   *      This program is free software; you can redistribute it and/or
29   *      modify it under the terms of the GNU General Public License
30   *      as published by the Free Software Foundation; either version
31   *      2 of the License, or (at your option) any later version.
```

```
32   */

33  #include <asm/segment.h>
34  #include <asm/system.h>
35  #include <linux/types.h>
36  #include <linux/kernel.h>
37  #include <linux/sched.h>
38  #include <linux/mm.h>
39  #include <linux/string.h>
40  #include <linux/socket.h>
41  #include <linux/sockios.h>
42  #include <linux/errno.h>
43  #include <linux/in.h>
44  #include <linux/inet.h>
45  #include <linux/netdevice.h>
46  #include "ip.h"
47  #include "protocol.h"
48  #include "route.h"
49  #include "tcp.h"
50  #include <linux/skbuff.h>
51  #include "sock.h"
52  #include "icmp.h"

53  /*
54   *  The routing table list
55   */

56  static struct rtable *rt_base = NULL;

57  /*
58   *  Pointer to the loopback route
59   */

60  static struct rtable *rt_loopback = NULL;
```

56行rt_base变量指向路由表项构成的链表。IP模块在进行路由选择时，相关路由查询函数将以此变量为据，对各表项进行查询，返回一个合适的路由表项。本文件实现中，几乎所有的函数凡是涉及到路由表的操作，都需要使用该变量。另外虽然通常我们说到“表”，大脑中出现的图像形似一个二维数组之类的表格形式。但实现上并非如此，“表”只是概念上的，关键的是对表项的维护，而表项本身是一个结构，就形成了这种二维关系。使用链表而非数组架构，在于链表不受表项个数容量的限制，而且利于内存使用，只有在必要时分配一个新的表项。 rt_loopback变量专门用于本地路由，该变量指向的路由表中通常只有一个表项，这在下文代码分析中自见。

```
61  /*
62   *  Remove a routing table entry.
63   */

64  static void rt_del(unsigned long dst, char *devname)
65  {
```

rt_del函数删除一个路由表项，参数dst表示这个表项对应的目的地址，devname表示表项绑定的接口名称（注意每个路由表项都必须与一个接口绑定）。函数实现无非是从rt_base变量指向的路由表首部出发，检查其中每个表项的对应字段，寻找满足条件的表项，将其从链表中删除。

```
66      struct rtable *r, **rp;
67      unsigned long flags;

68      rp = &rt_base;

69      /*
70       *  This must be done with interrupts off because we could take
71       *  an ICMP_REDIRECT.
72       */

73      save_flags(flags);
74      cli();
75      while((r = *rp) != NULL)
76      {
77          /* Make sure both the destination and the device match */
78          if ( r->rt_dst != dst ||
79          (devname != NULL && strcmp((r->rt_dev)->name,devname) != 0) )
80          {
81              rp = &r->rt_next;
82              continue;
83          }
84          *rp = r->rt_next;

85          /*
86           *  If we delete the loopback route update its pointer.
87           */

88          if (rt_loopback == r)
89              rt_loopback = NULL;
90          kfree_s(r, sizeof(struct rtable));
91      }
92      restore_flags(flags);
```

```
93  }
```

78-79行代码完成对表项内容具体的检查，如果目的地址相符且使用相同的接口，则表示寻找到一个符合的路由表项，此时84行完成该表项的删除工作。88行对该表项是不是一个本地路由表项进行检查，如果是，则相应的更新rt_loopback变量。最后90行对被删除表项对应的rtable结构进行释放。

```
94  /*
95   *  Remove all routing table entries for a device. This is called when
96   *  a device is downed.
97   */

98  void ip_rt_flush(struct device *dev)
99  {
100     struct rtable *r;
101     struct rtable **rp;
102     unsigned long flags;

103     rp = &rt_base;
104     save_flags(flags);
105     cli();
106     while ((r = *rp) != NULL) {
107         if (r->rt_dev != dev) {
108             rp = &r->rt_next;
109             continue;
110         }
111         *rp = r->rt_next;
112         if (rt_loopback == r)
113             rt_loopback = NULL;
114         kfree_s(r, sizeof(struct rtable));
115     }
116     restore_flags(flags);
117 }
```

ip_rt_flush函数删除与一个设备相绑定的所有路由表项，这个函数在设备停止工作时被调用，因为设备被关闭，无法通过该设备传输数据包，此时与该设备绑定的所有路由表项将变得不可用，所以调用能够该函数对这些路由表项进行删除。对这些符合删除的条件的检查是在107行完成的，即检查对应表项绑定的接口是否是这个停止工作的设备，如果是，则删除该表项。112行完成对rt_loopback变量同样的更新。

```
118 /*
119  *  Used by 'rt_add()' when we can't get the netmask any other way..
120  *
```

```
121  *  If the lower byte or two are zero, we guess the mask based on the
122  *  number of zero 8-bit net numbers, otherwise we use the "default"
123  *  masks judging by the destination address and our device netmask.
124  */

125 static inline unsigned long default_mask(unsigned long dst)
126 {
127     dst = ntohl(dst);
128     if (IN_CLASSA(dst))
129         return htonl(IN_CLASSA_NET);
130     if (IN_CLASSB(dst))
131         return htonl(IN_CLASSB_NET);
132     return htonl(IN_CLASSC_NET);
133 }
```

当创建一个路由表项时，如果没有提供子网掩码，则调用default_mask函数计算对应目的地址的网络掩码，该函数通过对地址类别的检查计算网络掩码，不过这种计算方式，在涉及到子网时会出现问题。当然在没有明确提供子网掩码的情况下，内核所能做的只有如此了。IN_CLASSA，IN_CLASSA_NET等这些宏或者常量均定义在include/linux/in.h中。该函数被下面的guess_mask函数调用。

```
134 /*
135  *  If no mask is specified then generate a default entry.
136  */

137 static unsigned long guess_mask(unsigned long dst, struct device * dev)
138 {
139     unsigned long mask;

140     if (!dst)
141         return 0;
142     mask = default_mask(dst);
143     if ((dst ^ dev->pa_addr) & mask)
144         return mask;
145     return dev->pa_mask;
146 }
```

guess_mask函数首先调用default_mask根据地址类别计算对应目的地址的网络掩码，143行对这个默认的掩码进行检查，首先将目的地址与本地接口地址进行异或，异或的结果是二者主机部分非0，如果进行了子网划分且不属同一子网，则子网部分也非0，如果大家根本就不是一家，那么网络部分也非0，由于调用default_mask函数返回的是网络掩码，换句话说，如果143行if语句为真，则表示目的地址与本地接口地址网络部分都不相同，此时就不能使用我们自己接口的掩码，就必须返回这个通过default_mask函数计算的网络掩码值。否则至少说明目的地址与本地接口地址在同一个网络中，但是有可能不属于相同子网，不过属于相

同子网的可能性大些，或者，就假设二者属于相同子网，所以最后就直接返回本地接口地址的掩码了（145行）。这个函数本身就是对目的地址对应掩码的一种猜测，既然是猜测，那么以上这个假设就变得合法了，既然都是猜，那么我们总是取我们希望取的值。

```
147 /*
148  *  Find the route entry through which our gateway will be reached
149  */

150 static inline struct device * get_gw_dev(unsigned long gw)
151 {
152     struct rtable * rt;

153     for (rt = rt_base ; ; rt = rt->rt_next)
154     {
155         if (!rt)
156             return NULL;
157         if ((gw ^ rt->rt_dst) & rt->rt_mask)
158             continue;
159         /*
160          *  Gateways behind gateways are a no-no
161          */

162         if (rt->rt_flags & RTF_GATEWAY)
163             return NULL;
164         return rt->rt_dev;
165     }
166 }
```

以下讨论中网关或者路由器用词不是很规范,在此使用的目的主要是强调二者所具有的数据包转发功能，其它区别此处不考虑，所以读者此时可以将二者等同对待。

通常给出一个目的地址，如果该目的地址对应的主机不在同一链路上，则需要先将数据包发送给本地链路上的一个网关或者路由器，由它们进行转发。此时通过查找路由表返回这个表项的网关地址就是本地链路上这个路由器或者网关的IP地址，但是光有这个网关地址，我们并不能直接发送数据，我们还需要查询可以到达该网关的接口以及对应该网关IP地址的MAC地址（这通过ARP请求完成）。
get_gw_dev函数即完成寻找到达该网关或者路由器的本地接口的工作。查找的过程类似查找一个新的路由表项，不过此时的目的地址变为本地链路上网关或者路由器的IP地址，157是对路由表项检查，该行语句表达的含义是：网关或者路由器地址与本地接口地址同属一个子网，即在同一个链路上，此时该接口可直达该网关或者路由器。157行将两个地址异或操作后在于子网掩码作与操作，检查的就是两个地址的网络号和子网号是否相同，当然这是从本地接口的角度考虑的，因为使用的是本地接口地址的子网掩码。如果162行是对表项类型的检查，如果该表项本身又是一个通过网关转发的地址，则表示出现了循环的状况。因为一般在设置网关转发地址时，该网关应该是一个本地链路上的主机，而不是本地链路之外的网关

或者路由器。在成功查找到一个合适的路由表项后，返回该表项绑定的网络接口，通过这个接口我们可以直达由参数表示的这个网关或者路由器。

下面的这个函数用于添加一个新的路由表项，函数写的比较长，我们分段对其进行讨论。

```
167 /*
168  *  Rewrote rt_add(), as the old one was weird - Linus
169  *
170  *  This routine is used to update the IP routing table, either
171  *  from the kernel (ICMP_REDIRECT) or via an ioctl call issued
172  *  by the superuser.
173  */

174 void ip_rt_add(short flags, unsigned long dst, unsigned long mask,
175     unsigned long gw, struct device *dev, unsigned short mtu,
        unsigned long window)
176 {
```

参数说明：
flags：路由表项标志位。
dst：目的IP地址。
mask：子网掩码。
gw：到达目的主机的网关IP地址。
dev：发送到目的地址主机的接口。
mtu：发送对应目的地址主机的最大报文长度。
window：窗口值，意义在相关代码中给出。
对于这些传入的参数，ip_rt_add函数并非一成不变的使用，而是根据需要对参数值进行修改，例如如下对mask子网掩码的修改。

```
177     struct rtable *r, *rt;
178     struct rtable **rp;
179     unsigned long cpuflags;

180     /*
181      *  A host is a unique machine and has no network bits.
182      */

183     if (flags & RTF_HOST)
184     {
185         mask = 0xffffffff;
186     }
```

如果目的地址是主机地址，那么子网掩码就是255.255.255.255.

```
187     /*
```

```
188      *  Calculate the network mask
189      */

190     else if (!mask)
191     {
192         if (!((dst ^ dev->pa_addr) & dev->pa_mask))
193         {
194             mask = dev->pa_mask;
195             flags &= ~RTF_GATEWAY;
196             if (flags & RTF_DYNAMIC)
197             {
198                 /*printk("Dynamic route to my own net rejected\n");*/
199                 return;
200             }
201         }
202         else
203             mask = guess_mask(dst, dev);
204         dst &= mask;
205     }
```

否则表示目的地址是一个网络，此时如果参数mask没有指定子网掩码，那么190-205行代码就对子网掩码进行猜测。192行检测目的网络与本地接口地址是否属于同一个子网，如是，则将子网掩码设置为本地接口地址的掩码，由于目的网络与本地接口地址属于同一个子网，就表示这个目的网络是直接相连的，此时就将flags参数中可能的RTF_GATEWAY标志位清除掉。RTF_GATEWAY标志位表示目的主机或者网络不是在同一个链路上，需要通过网关或者路由器进行数据包转发。如果192行代码检测出目的网络与本地接口地址不属于同一个子网，那么就调用guess_mask函数对网络子网掩码进行猜测，实际上这种猜测使得子网不可见。最后注意204行代码对目的主机进行的调整，这样做是为了保证路由表项目的地址和其子网掩码的一致性。如果调用者提供对应的子网掩码，那么204行的计算将不会改变目的地址值，相反，如果没有提供，那么既然调用者可以“放心”让我们猜，那么他就要承受204行所作的改变，有可能目的网络是一个子网，经过204行的计算后，子网被屏蔽了。

```
206     /*
207      *  A gateway must be reachable and not a local address
208      */

209     if (gw == dev->pa_addr)
210         flags &= ~RTF_GATEWAY;
```

如果输入参数gw是本地接口地址，那么就表示目的主机或者网络是直达的，所以清除RTF_GATEWAY.

```
211     if (flags & RTF_GATEWAY)
212     {
213         /*
```

```
214          *  Don't try to add a gateway we can't reach..
215          */

216         if (dev != get_gw_dev(gw))
217             return;

218         flags |= RTF_GATEWAY;
219     }
220     else
221         gw = 0;
```

如上代码对非直达地址进行检测，首先判断传入的dev参数与调用get_gw_dev函数返回的设备是否是同一个设备，get_gw_dev函数返回可直达网关的本地接口设备。如果不相等，就表示指定的设备与实际可直达的设备不一致，出现了问题，此时直接返回，不对这个表项进行添加。注意220-221行代码和前文中对路由表项的打印中对于目的地址是直达网络或者主机的网关地址为0的情况，此处得到的验证。即如果目的地址是一个直达网络或者主机，那么网关地址被设置为0。这与WINDOWS下不一样，WINDOWS下对于目的地址是直达网络或者主机的情况，网关地址被设置为本地接口地址。

```
222     /*
223      *  Allocate an entry and fill it in.
224      */
```

经过以上对子网掩码和网关地址以及路由标志位的重新校正， 现在我们可以创建一个新的路由表项了。每个路由表项在内核中均有一个rtable结构表示。函数中余下代码完成的工作就是分配一个新的rtable结构，并对其进行初始化，最后将其插入到系统路由表中，即有rt_base变量指向的链表，如果这个表项是一个回环路由，则还需对rt_loopback变量进行更新。

```
225     rt = (struct rtable *) kmalloc(sizeof(struct rtable), GFP_ATOMIC);
226     if (rt == NULL)
227     {
228         return;
229     }
230     memset(rt, 0, sizeof(struct rtable));
231     rt->rt_flags = flags | RTF_UP;
232     rt->rt_dst = dst;
233     rt->rt_dev = dev;
234     rt->rt_gateway = gw;
235     rt->rt_mask = mask;
236     rt->rt_mss = dev->mtu - HEADER_SIZE;
237     rt->rt_window = 0;  /* Default is no clamping */
```

结合rtable结构定义，231-237行代码理解上比较容易，我们在前文对window参数的含义收的不是很明白，因为没有上下文，无法对一个参数的具体函数进行猜测，此处从237行代码来看，window参数应该是对rtable结构的rt_window字段进行赋值，从该行代码之后的注释我们可以猜测出rt_window字段的意义：通过窗口对发送数据速率进行节制。当本地发送数据到对应远端时，则进行路由表查找时，返回该表项rtable结构，rt_window字段就被用作为数据的发送窗口。此处初始化为0，下文中对该字段进行更正。

```
238     /* Are the MSS/Window valid ? */

239     if(rt->rt_flags & RTF_MSS)
240         rt->rt_mss = mtu;

241     if(rt->rt_flags & RTF_WINDOW)
242         rt->rt_window = window;
```

239-242行代码对于MSS，WINDOW值进行更正，如果标志位中明确要求指定这些值的话。

```
243     /*
244      *  What we have to do is loop though this until we have
245      *  found the first address which has a higher generality than
246      *  the one in rt.  Then we can put rt in right before it.
247      *  The interrupts must be off for this process.
248      */

249     save_flags(cpuflags);
250     cli();

251     /*
252      *  Remove old route if we are getting a duplicate.
253      */

254     rp = &rt_base;
255     while ((r = *rp) != NULL)
256     {
257         if (r->rt_dst != dst ||
258             r->rt_mask != mask)
259         {
260             rp = &r->rt_next;
261             continue;
262         }
263         *rp = r->rt_next;
264         if (rt_loopback == r)
265             rt_loopback = NULL;
266         kfree_s(r, sizeof(struct rtable));
```

```
267     }
```

254-267行代码删除路由表中重叠的表项，所谓重叠表项是指目的地址和子网掩码都相同的表项。

```
268     /*
269      *  Add the new route
270      */

271     rp = &rt_base;
272     while ((r = *rp) != NULL) {
273         if ((r->rt_mask & mask) != mask)
274             break;
275         rp = &r->rt_next;
276     }
277     rt->rt_next = r;
278     *rp = rt;
```

271-278行代码将新的路由表项插入到系统路由表中，即有rt_base指向的rtable类型的链表。从插入算法来看，rt_base指向的路由链表中表项是按照子网掩码长度（1的个数）由长到短的顺序排列的。换句话说，目的地址为单个主机的排在链表最前面，其次为网络地址，最后为默认路由项（即目的地址为全0，子网掩码为全0）。

```
279     /*
280      *  Update the loopback route
281      */

282     if ((rt->rt_dev->flags & IFF_LOOPBACK) && !rt_loopback)
283         rt_loopback = rt;
```

282-283行代码检测新加入的路由表项是不是一个本地回环路由表项（目的地址为127.0.0.0),如果是，且rt_loopback为NULL，则将rt_loopback变量赋值为该新加入的表项。

```
284     /*
285      *  Restore the interrupts and return
286      */

287     restore_flags(cpuflags);
288     return;
289 }
```

ip_rt_add函数被调用用来添加一个新的路由表项，函数实现上首先对子网掩码和网关地址以及路由标志位进行校正，此后创建一个新的rtable结构（每个路由表项由一个rtable结构

表示）并对其进行初始化，最后将其加入到系统路由表中。在加入新的表项之前，还必须对路由表中已有表项进行检查，剔除重复表项。

```
290 /*
291  *  Check if a mask is acceptable.
292  */

293 static inline int bad_mask(unsigned long mask, unsigned long addr)
294 {
295     if (addr & (mask = ~mask))
296         return 1;
297     mask = ntohl(mask);
298     if (mask & (mask+1))
299         return 1;
300     return 0;
301 }
```

bad_mask函数用于检测对应一个地址的子网掩码是否正确。一般而言，一个子网掩码是高位连续为1，低位连续为0。不可能出现0，1交错的情况。对于这种交错的情况，就表示子网掩码不正确。295行将子网掩码取反后与地址进行与操作，正常情况下，这个计算结果一定为0，如果非0，则表示地址对应的子网掩码不正确。例如主机地址为192.168.0.2,正确的对应子网掩码为255.255.255.255，对该子网掩码取反后，子网掩码为0，此时与原地址作与操作，则结果一定为0。又如192.168.1.0,正确的对应子网掩码为255.255.255.0,子网掩码取反后为0.0.0.255,那么作与运算后,结果是0。所以295行在对地址配置了正确子网掩码的情况下，计算结果一定为0，如果计算结果非0，则表示子网掩码配置错误。298行是对子网掩码中0，1交错情况的检查，对于0，1交错的子网掩码一定是不正确的。

```
302 /*
303  *  Process a route add request from the user
304  */

305 static int rt_new(struct rtentry *r)
306 {
```

rtentry和old_rtentry结构被上层使用，用以配置路由表项。rt_new函数即根据上层传入的rtentry参数添加一个新的路由表项。所以该函数完成的主要工作就是对rtentry结构中个字段值合法性进行检查，并根据rtentry结构字段值产生路由表项所需要的字段值，最后调用ip_rt_add函数完成对新的路由表项的添加。为便于读者理解，如下再次给出rtentry结构的定义：

```
/*include/linux/route.h*/
31  /* This structure gets passed by the SIOCADDRT and SIOCDELRT calls. */
32  struct rtentry {
33      unsigned long   rt_hash;    /* hash key for lookups     */
34      struct sockaddr rt_dst;     /* target address       */
```

```
35      struct sockaddr rt_gateway; /* gateway addr (RTF_GATEWAY)    */
36      struct sockaddr rt_genmask; /* target network mask (IP) */
37      short       rt_flags;
38      short       rt_refcnt;
39      unsigned long   rt_use;
40      struct ifnet    *rt_ifp;
41      short       rt_metric;  /* +1 for binary compatibility! */
42      char        *rt_dev;    /* forcing the device at add    */
43      unsigned long   rt_mss;     /* per route MTU/Window */
44      unsigned long   rt_window;  /* Window clamping */
45  };
```

rtentry结构与路由表项表示结构rtable结构中字段几乎形成一一对应关系，不过我们不可直接根据rtentry结构中字段创建一个新的rtable结构，由于这些字段值是由用户设置的，所以必然的，我们需要对这些字段值的合法性进行检查和修改。

下面我们即开始对rt_new函数的分析。

```
307     int err;
308     char * devname;
309     struct device * dev = NULL;
310     unsigned long flags, daddr, mask, gw;

311     /*
312      *  If a device is specified find it.
313      */

314     if ((devname = r->rt_dev) != NULL)
315     {
316         err = getname(devname, &devname);
317         if (err)
318             return err;
319         dev = dev_get(devname);
320         putname(devname);
321         if (!dev)
322             return -EINVAL;
323     }
```

314-323行代码根据设备名称查找具体的设备结构，getname函数将名称字符串从用户缓冲区复制到内核缓冲区中，dev_get函数遍历dev_base变量指向的系统中所有设备结构列表，查找对应名称的设备，返回该设备对应的device结构。Putname函数释放在getname中分配的内核缓冲区。如果以上代码通过，则首先我们根据设备名称查找到了具体的设备结构。因为路由表项中需要指向绑定的设备，所以此处的查找为其后可能创建一个新的路由表项作准备。

```
324     /*
325      *  If the device isn't INET, don't allow it
326      */

327     if (r->rt_dst.sa_family != AF_INET)
328         return -EAFNOSUPPORT;
```

注意,rtentry结构中地址的表示都是sockaddr结构,327行代码对地址所属域类型进行检查，如果地址不属于AF_INET域，则直接返回。

```
329     /*
330      *  Make local copies of the important bits
331      */

332     flags = r->rt_flags;
333     daddr = ((struct sockaddr_in *) &r->rt_dst)->sin_addr.s_addr;
334     mask = ((struct sockaddr_in *) &r->rt_genmask)->sin_addr.s_addr;
335     gw = ((struct sockaddr_in *) &r->rt_gateway)->sin_addr.s_addr;
```

332-335行代码取出参数rtentry结构中各字段值。下面我们要对这些值进行检查和修改。

```
336     /*
337      *  BSD emulation: Permits route add someroute gw one-of-my-addresses
338      *  to indicate which iface. Not as clean as the nice Linux dev technique
339      *  but people keep using it...
340      */

341     if (!dev && (flags & RTF_GATEWAY))
342     {
```

如果dev变量为NULL，则表示：其一rtentry结构参数中没有指定设备名称；其二就是指定的设备不存在。如果dev为NULL，并且RTF_GATEWAY标志位被设置，那么就需要对提供的网关地址进行检查。这其中似乎也不存在什么因果联系，所以这个“那么”使用的有些不合理。应该说对于网关地址的检查，在dev无论是NULL非NULL的情况都要进行。如果网关地址是本地接口地址，则表示这不是一个合法的网关地址，或者说目的主机或者网络是可直达的。在如下实现上，取的是后一种解释：即目的主机或者网络可直达，所以清除RTF_GATEWAY标志位。

```
343         struct device *dev2;
344         for (dev2 = dev_base ; dev2 != NULL ; dev2 = dev2->next)
345         {
346             if ((dev2->flags & IFF_UP) && dev2->pa_addr == gw)
347             {
348                 flags &= ~RTF_GATEWAY;
349                 dev = dev2;
350                 break;
```

```
351             }
352         }
353     }
```

```
354     /*
355      *  Ignore faulty masks
356      */

357     if (bad_mask(mask, daddr))
358         mask = 0;
```

这是对子网掩码的正确性进行检查，如果bad_mask函数返回非0，表示子网掩码字段不正确，此时将子网掩码设置为0，下文中会对子网掩码进行重新初始化，另外在调用的ip_rt_add函数中还会进一步对子网掩码合法性进行检查。

```
359     /*
360      *  Set the mask to nothing for host routes.
361      */

362     if (flags & RTF_HOST)
363         mask = 0xffffffff;
364     else if (mask && r->rt_genmask.sa_family != AF_INET)
365         return -EAFNOSUPPORT;
```

362行：如果目的地址是主机地址，则子网掩码设置为全1即可。364行对子网掩码地址的域类型进行检查，如果不是AF_INET域，则出错返回。

```
366     /*
367      *  You can only gateway IP via IP..
368      */

369     if (flags & RTF_GATEWAY)
370     {
371         if (r->rt_gateway.sa_family != AF_INET)
372             return -EAFNOSUPPORT;
373         if (!dev)
374             dev = get_gw_dev(gw);
375     }
376     else if (!dev)
377         dev = ip_dev_check(daddr);
```

这段代码重新对接口设备进行检查和初始化。对于一个非直达网络或者主机而言，查找的接

口设备是可直达网关的设备（get_gw_dev函数），而对于一个可直达网络或者主机，那么接口设备就是一个与目的地址在同一个子网上的设备（ip_dev_check）.总之，完成这段代码后，必须有一个可使用的接口设备，否则无法进行新的路由表项的创建。

```
378     /*
379      *  Unknown device.
380      */

381     if (dev == NULL)
382         return -ENETUNREACH;

383     /*
384      *  Add the route
385      */

386     ip_rt_add(flags, daddr, mask, gw, dev, r->rt_mss, r->rt_window);
387     return 0;
388 }
```

函数在完成对所有参数必要的检查和更正之后，调用ip_rt_add函数（386行）完成新的路由表项的创建并将其加入到系统路由表中。

```
389 /*
390  *  Remove a route, as requested by the user.
391  */

392 static int rt_kill(struct rtentry *r)
393 {
394     struct sockaddr_in *trg;
395     char *devname;
396     int err;

397     trg = (struct sockaddr_in *) &r->rt_dst;
398     if ((devname = r->rt_dev) != NULL)
399     {
400         err = getname(devname, &devname);
401         if (err)
402             return err;
403     }
404     rt_del(trg->sin_addr.s_addr, devname);
405     if ( devname != NULL )
406         putname(devname);
407     return 0;
```

```
408 }
```

rt_kill函数完成的工作正好与rt_new函数相反：其根据传入的参数删除一个路由表项。删除的情况相对添加的情况较为简单，因为无需对各种参数进行检查，只要路由表中存在对应目的地址的表项，则简单删除之即可达到目的。函数实现比较简单，关键是在404行对rt_del函数的调用，传入的参数是要删除表项对应的目的地址和绑定设备名称。rt_del函数在前文中已经作出分析，其主要有rt_base指向的路由表项链表出发，对其中每个表项进行目的地址和设备名称的比较，如果发现二者均匹配的表项，则删除之。

```
409 /*
410  *  Called from the PROCfs module. This outputs /proc/net/route.
411  */

412 int rt_get_info(char *buffer, char **start, off_t offset, int length)
413 {
414     struct rtable *r;
415     int len=0;
416     off_t pos=0;
417     off_t begin=0;
418     int size;

419     len += sprintf(buffer,
420          "Iface\tDestination\tGateway
\tFlags\tRefCnt\tUse\tMetric\tMask\t\tMTU\tWindow\n");
421     pos=len;

422     /*
423      *  This isn't quite right -- r->rt_dst is a struct!
424      */

425     for (r = rt_base; r != NULL; r = r->rt_next)
426     {
427             size = sprintf(buffer+len,
"%s\t%08lX\t%08lX\t%02X\t%d\t%lu\t%d\t%08lX\t%d\t%lu\n",
428             r->rt_dev->name, r->rt_dst, r->rt_gateway,
429             r->rt_flags, r->rt_refcnt, r->rt_use, r->rt_metric,
430             r->rt_mask, (int)r->rt_mss, r->rt_window);
431         len+=size;
432         pos+=size;
433         if(pos<offset)
434         {
435             len=0;
436             begin=pos;
437         }
```

```
438         if(pos>offset+length)
439             break;
440     }

441     *start=buffer+(offset-begin);
442     len-=(offset-begin);
443     if(len>length)
444         len=length;
445     return len;
446 }
```

rt_get_info返回系统路由表项信息，我们在shell下敲入‘route’命令或者‘netstat -rn’命令时显示的路由表信息即是该函数的返回值。
函数实现也很简单，读者结合rtable结构定义很容易理解。

至此，虽然路由表的主要作用是被IP协议模块查询寻找到达某个目的地址的合适的路由途径，但到现在还都是对路由表的维护，下面我们进入使用路由表函数的分析，这些函数将被调用查询到达某个目的地址的路由表项。

```
447 /*
448  *  This is hackish, but results in better code. Use "-S" to see why.
449  */

450 #define early_out ({ goto no_route; 1; })
```

early_out宏的使用在下文使用中再进行介绍，此处无法对此进行分析。只需注意的是其中no_route标志符与使用该宏的函数是绑定的，即要使用该宏定义，则函数中必须定义no_route这样一个标志符，否则会出现编译错误。

```
451 /*
452  *  Route a packet. This needs to be fairly quick. Florian & Co.
453  *  suggested a unified ARP and IP routing cache. Done right its
454  *  probably a brilliant idea. I'd actually suggest a unified
455  *  ARP/IP routing/Socket pointer cache. Volunteers welcome
456  */

457 struct rtable * ip_rt_route(unsigned long daddr, struct options *opt,
                unsigned long *src_addr)
458 {
```

ip_rt_route函数被IP模块调用查找一个合适的路由表项，用于对将要发送的数据包进行路由。
参数说明：
daddr：目的地址，这是寻找合适路由表项的关键字。

opt：可能的IP选项，此处传入这个参数有些不伦不类，实际上函数也没有使用到该参数。
src_addr：这是一个地址，ip_rt_route函数填入查询到的路由表项中接口设备对应的IP地址。

```
459     struct rtable *rt;
460     for (rt = rt_base; rt != NULL || early_out ; rt = rt->rt_next)
461     {
462         if (!((rt->rt_dst ^ daddr) & rt->rt_mask))
463             break;
464         /*
465          *  broadcast addresses can be special cases..
466          */
467         if (rt->rt_flags & RTF_GATEWAY)
468             continue;
469         if ((rt->rt_dev->flags & IFF_BROADCAST) &&
470             (rt->rt_dev->pa_brdaddr == daddr))
471             break;
472     }
```

如果读者将460-472行代码与下一个介绍的ip_rt_local函数对照，会发现代码实现上微妙的差别，紧紧是调整了代码的上下位置，所获得的效果截然不同。ip_rt_route函数查询路由表项不对目的地址进行限制，而ip_rt_local将目的地址限制在本地链路上（这正是函数名中local的含义），读者可以仔细观察为实现这个功能代码实现上的微妙差别。
不过现在我们还是对以上460-472行代码进行介绍。460行对路由表中每个表项进行检查，注意其中对early_out宏的使用，此处对early_out宏的使用考虑到或运算符(||)的微妙特性，如果rt!=NULL条件为真，则不会对early_out所表示的语句进行执行，只有当rt为NULL时，才进行early_out宏的执行，前文中early_out宏定义如下：

```
450  #define early_out ({ goto no_route; 1; })
```

即跳转到goto no_route标志符处执行，no_route标志符定义在ip_rt_route函数结尾处，直接返回NULL，退出函数。462行代码对路由表项目的地址与要路由的目的地址是否同属于一个子网进行检查，如果同属于一个子网（注意其中包括了目的地址是一个主机地址的情况），那么就表示找到了一个合适的路由选项，此时463行break语句跳出循环检测。如果462行不满足条件，则467行检查该表项是否表示一个非直达网络或者主机地址，对于非直达的情况，无需继续进行检查，直接进行下一个表项的检查；否则检查路由表项绑定的设备是否支持广播，如果支持，并且查询关键字所表示的目的地址也是一个广播地址，那么该表项也可使用，此时也跳出循环检测。

由于路由表中表项是按子网掩码中高位1的数目长度从长到短排列的，换句话说，目的地址为主机地址的排列在最前面，其次为网络地址，最后为默认路由表项（即目的地址为0.0.0.0）。所以对于存在默认路由表项的路由表，462行代码一定会得到满足。对于没有配置默认路由表项的主机，则如果没有找到合适的表项，则返回主机不可达或者网络不可达的错误。如果在LINUX下，路由表没有正确配置时，在PING一个远端主机时，界面上通常显示

一个网络不可达的错误。在找到一个合适的路由表项后，下面即进行字段的更新。

```
473     if(src_addr!=NULL)
474         *src_addr= rt->rt_dev->pa_addr;

475     if (daddr == rt->rt_dev->pa_addr) {
476         if ((rt = rt_loopback) == NULL)
477             goto no_route;
478     }
479     rt->rt_use++;
480     return rt;
481 no_route:
482     return NULL;
483 }
```

473行对src_addr进行赋值，这个参数指向一个地址，ip_rt_route函数根据寻找到的路由表项中绑定设备接口的IP地址对该参数进行赋值：即本地源端地址。475行对目的地址是本地接口地址的情况进行检查，如果路由目的地址是本地接口地址，则表示这是一个回环路由，对于回环路由，使用rt_loopback变量指向路由表（该路由表一般只有一个表项），如果对应路由表为空，则表示没有可用的路由表项，此时就表示不可达（这很滑稽是不是，发送给本机的数据包居然得到目的不可达的错误！）。对于一个网络配置不正确的主机，回环通道当然也可能出现问题。所以发送到本机的数据包出现目的不可达的情况并不奇怪！函数最后增加查找到的路由表项的使用计数（479行），返回该表项。

```
484 struct rtable * ip_rt_local(unsigned long daddr, struct options *opt,
                               unsigned long *src_addr)
485 {
```

ip_rt_local函数完成对本地链路上主机或者网络地址的路由查询工作。这个函数实现上与ip_rt_route函数基本相同，只是在查询路由表项时几个语句的顺序调整了一下，具体如下492-495行代码，ip_rt_local将对路由表项RTF_GATEWAY标志位的检查放在了地址比较的前面，这个语句（492行代码）的提前对所有目的地址非直达的表项都跳过了检查，只对目的地址可直达的表项进行比较。这样就实现了只对目的地址在本地链路上的表项进行检查，也就是函数名中local的含义。

```
486     struct rtable *rt;

487     for (rt = rt_base; rt != NULL || early_out ; rt = rt->rt_next)
488     {
489         /*
490          *  No routed addressing.
491          */
492         if (rt->rt_flags&RTF_GATEWAY)
493             continue;
```

```
494         if (!((rt->rt_dst ^ daddr) & rt->rt_mask))
495             break;
496         /*
497          *  broadcast addresses can be special cases..
498          */

499         if ((rt->rt_dev->flags & IFF_BROADCAST) &&
500              rt->rt_dev->pa_brdaddr == daddr)
501             break;
502     }

503     if(src_addr!=NULL)
504         *src_addr= rt->rt_dev->pa_addr;

505     if (daddr == rt->rt_dev->pa_addr) {
506         if ((rt = rt_loopback) == NULL)
507             goto no_route;
508     }
509     rt->rt_use++;
510     return rt;
511 no_route:
512     return NULL;
513 }

514 /*
515  *  Backwards compatibility
516  */

517 static int ip_get_old_rtent(struct old_rtentry * src, struct rtentry * rt)
518 {
519     int err;
520     struct old_rtentry tmp;

521     err=verify_area(VERIFY_READ, src, sizeof(*src));
522     if (err)
523         return err;
524     memcpy_fromfs(&tmp, src, sizeof(*src));
525     memset(rt, 0, sizeof(*rt));
526     rt->rt_dst = tmp.rt_dst;
527     rt->rt_gateway = tmp.rt_gateway;
528     rt->rt_genmask.sa_family = AF_INET;
529     ((struct sockaddr_in *) &rt->rt_genmask)->sin_addr.s_addr =
tmp.rt_genmask;
```

```
530     rt->rt_flags = tmp.rt_flags;
531     rt->rt_dev = tmp.rt_dev;
532     printk("Warning: obsolete routing request made.\n");
533     return 0;
534 }
```

ip_get_old_rtent函数完成从old_rtentry结构到rtentry结构的转换。结合这两个结构的定义，本函数实现比较简单。此处不再阐述。

```
535 /*
536  *  Handle IP routing ioctl calls. These are used to manipulate the routing
tables
537  */

538 int ip_rt_ioctl(unsigned int cmd, void *arg)
539 {
540     int err;
541     struct rtentry rt;

542     switch(cmd)
543     {
544         case SIOCADDRTOLD:  /* Old style add route */
545         case SIOCDELRTOLD:  /* Old style delete route */
546             if (!suser())
547                 return -EPERM;
548             err = ip_get_old_rtent((struct old_rtentry *) arg, &rt);
549             if (err)
550                 return err;
551             return (cmd == SIOCDELRTOLD) ? rt_kill(&rt) : rt_new(&rt);

552         case SIOCADDRT:     /* Add a route */
553         case SIOCDELRT:     /* Delete a route */
554             if (!suser())
555                 return -EPERM;
556             err=verify_area(VERIFY_READ, arg, sizeof(struct rtentry));
557             if (err)
558                 return err;
559             memcpy_fromfs(&rt, arg, sizeof(struct rtentry));
560             return (cmd == SIOCDELRT) ? rt_kill(&rt) : rt_new(&rt);
561     }

562     return -EINVAL;
563 }
```

ip_rt_ioctl函数提供一个操作路由表的接口给用户。对于使用老的old_rtentry结构进行设置的情况，内核首先通过ip_get_old_rtent函数将old_rtentry转化为rtentry结构，然后调用rt_kill或者rt_new进行路由表项的删除或者添加。对于使用新的rtentry结构进行设置的情况，无需以上转换，直接根据命令对rt_kill, rt_new函数进行调用。函数中四个设置标志的含义一目了然，此处不再做多余的说明。

route.c文件小结

一般谈到路由表时，我们通常将其想象的很复杂，通过对route.c文件的分析，首先使得我们减少了对路由表复杂性的畏惧，仔细分析之下，发现其实很简单。通常谈到“表”的概念，我们很直接的想象成表示数据列表的那种方方正正的形式,这种想象方式首先人为的增加了对路由表的理解。从route.c文件的分析中，我们看到系统路由表实际上是由一个变量指向的链表，每个表项由一个rtable结构表示，而查询路由表，简单的说就是对链表中每个元素进行检查,检查的根据就是对表项中目的地址和实际要发送数据包中的目的地址进行网络号（和子网号）的比较。一切都是如此简单，由此我们也可以克服对其他方面的为难情绪，因为大多数时候，困难都只存在于我们的想象之中。读者可以对route.c文件进行多次分析，从而完全弄清楚其中的基本原理，现在版本中的IP路由虽然变得十分庞大，但基本原理不会改变。这也是我们分析早期代码的意义所在！

在对路由进行了说明，我们可以正式进入对IP协议实现文件的分析，虽然“前路依旧漫漫”，我们“仍需继续前进”。首先我们明确一下上层接口，这一点在前文中已经一再强调，每个“上层协议”都由一个inet_protocol结构表示，IP协议模块完成其自身处理后，根据IP首部中上层协议字段值从inet_protos数组中查找对应inet_protocol结构，调用该结构中handler函数指针字段指向的函数，如TCP协议handler指向tcp_rcv, UDP为udp_rcv, ICMP对应icmp_rcv, IGMP对应igmp_rcv。这些对上层函数的调用是在IP协议实现模块中ip_rcv函数中完成的。

2.21 net/inet/ip.c文件

```
1   /*
2    * INET     An implementation of the TCP/IP protocol suite for the LINUX
3    *      operating system.  INET is implemented using the  BSD Socket
4    *      interface as the means of communication with the user level.
5    *
6    *      The Internet Protocol (IP) module.
7    *
8    * Version: @(#)ip.c     1.0.16b 9/1/93
9    *
10   * Authors: Ross Biro, <bir7@leland.Stanford.Edu>
11   *      Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
12   *      Donald Becker, <becker@super.org>
13   *      Alan Cox, <gw4pts@gw4pts.ampr.org>
14   *      Richard Underwood
15   *      Stefan Becker, <stefanb@yello.ping.de>
16   *
```

```
17   *
18   * Fixes:
19   *      Alan Cox    :   Commented a couple of minor bits of surplus code
20   *      Alan Cox    :   Undefining IP_FORWARD doesn't include the code
21   *                  (just stops a compiler warning).
22   *      Alan Cox    :   Frames with >=MAX_ROUTE record routes, strict routes
or loose routes
23   *                  are junked rather than corrupting things.
24   *      Alan Cox    :   Frames to bad broadcast subnets are dumped
25   *                  We used to process them non broadcast and
26   *                  boy could that cause havoc.
27   *      Alan Cox    :   ip_forward sets the free flag on the
28   *                  new frame it queues. Still crap because
29   *                  it copies the frame but at least it
30   *                  doesn't eat memory too.
31   *      Alan Cox    :   Generic queue code and memory fixes.
32   *      Fred Van Kempen :   IP fragment support (borrowed from NET2E)
33   *      Gerhard Koerting:   Forward fragmented frames correctly.
34   *      Gerhard Koerting:   Fixes to my fix of the above 8-).
35   *      Gerhard Koerting:   IP interface addressing fix.
36   *      Linus Torvalds  :   More robustness checks
37   *      Alan Cox    :   Even more checks: Still not as robust as it ought to
be
38   *      Alan Cox    :   Save IP header pointer for later
39   *      Alan Cox    :   ip option setting
40   *      Alan Cox    :   Use ip_tos/ip_ttl settings
41   *      Alan Cox    :   Fragmentation bogosity removed
42   *                  (Thanks to Mark.Bush@prg.ox.ac.uk)
43   *      Dmitry Gorodchanin :    Send of a raw packet crash fix.
44   *      Alan Cox    :   Silly ip bug when an overlength
45   *                  fragment turns up. Now frees the
46   *                  queue.
47   *      Linus Torvalds/ :   Memory leakage on fragmentation
48   *      Alan Cox    :   handling.
49   *      Gerhard Koerting:   Forwarding uses IP priority hints
50   *      Teemu Rantanen  :   Fragment problems.
51   *      Alan Cox    :   General cleanup, comments and reformat
52   *      Alan Cox    :   SNMP statistics
53   *      Alan Cox    :   BSD address rule semantics. Also see
54   *                  UDP as there is a nasty checksum issue
55   *                  if you do things the wrong way.
56   *      Alan Cox    :   Always defrag, moved IP_FORWARD to the config.in file
57   *      Alan Cox    :   IP options adjust sk->priority.
58   *      Pedro Roque :   Fix mtu/length error in ip_forward.
```

```
59   *      Alan Cox    :   Avoid ip_chk_addr when possible.
60   *  Richard Underwood   :   IP multicasting.
61   *      Alan Cox    :   Cleaned up multicast handlers.
62   *      Alan Cox    :   RAW sockets demultiplex in the BSD style.
63   *      Gunther Mayer   :   Fix the SNMP reporting typo
64   *      Alan Cox    :   Always in group 224.0.0.1
65   *      Alan Cox    :   Multicast loopback error for 224.0.0.1
66   *      Alan Cox    :   IP_MULTICAST_LOOP option.
67   *      Alan Cox    :   Use notifiers.
68   *      Bjorn Ekwall    :   Removed ip_csum (from slhc.c too)
69   *      Bjorn Ekwall    :   Moved ip_fast_csum to ip.h (inline!)
70   *      Stefan Becker   :       Send out ICMP HOST REDIRECT
71   *      Alan Cox    :   Only send ICMP_REDIRECT if src/dest are the same net.
72   *
73   *
74   * To Fix:
75   *      IP option processing is mostly not needed. ip_forward needs to know
about routing rules
76   *      and time stamp but that's about all. Use the route mtu field here too
77   *      IP fragmentation wants rewriting cleanly. The RFC815 algorithm is much
more efficient
78   *      and could be made very efficient with the addition of some virtual memory
hacks to permit
79   *      the allocation of a buffer that can then be 'grown' by twiddling page
tables.
80   *      Output fragmentation wants updating along with the buffer management
to use a single
81   *      interleaved copy algorithm so that fragmenting has a one copy overhead.
Actual packet
82   *      output should probably do its own fragmentation at the UDP/RAW layer.
TCP shouldn't cause
83   *      fragmentation anyway.
84   *
85   *      This program is free software; you can redistribute it and/or
86   *      modify it under the terms of the GNU General Public License
87   *      as published by the Free Software Foundation; either version
88   *      2 of the License, or (at your option) any later version.
89   */

90  #include <asm/segment.h>
91  #include <asm/system.h>
92  #include <linux/types.h>
93  #include <linux/kernel.h>
94  #include <linux/sched.h>
```

```
95  #include <linux/mm.h>
96  #include <linux/string.h>
97  #include <linux/errno.h>
98  #include <linux/config.h>

99  #include <linux/socket.h>
100 #include <linux/sockios.h>
101 #include <linux/in.h>
102 #include <linux/inet.h>
103 #include <linux/netdevice.h>
104 #include <linux/etherdevice.h>

105 #include "snmp.h"
106 #include "ip.h"
107 #include "protocol.h"
108 #include "route.h"
109 #include "tcp.h"
110 #include "udp.h"
111 #include <linux/skbuff.h>
112 #include "sock.h"
113 #include "arp.h"
114 #include "icmp.h"
115 #include "raw.h"
116 #include <linux/igmp.h>
117 #include <linux/ip_fw.h>

118 #define CONFIG_IP_DEFRAG
```

118行对CONFIG_IP_DEFRAG宏的定义表示实现中支持数据包分片处理功能。

```
119 extern int last_retran;
120 extern void sort_send(struct sock *sk);
```

last_retran, sort_send在本文件没有任何地方使用，我们当然也无法了解他们的含义，只有先放着。在drivers/slhc.c文件中定义了last_retran变量，slhc.c文件是对使用Serial Line（串行线）进行通信时压缩和解压缩数据的实现。

```
121 #define min(a,b)    ((a)<(b)?(a):(b))
122 #define LOOPBACK(x) (((x) & htonl(0xff000000)) == htonl(0x7f000000))
```

LOOPBACK宏是对回环地址进行检查。

```
123 /*
124  *  SNMP management statistics
125  */
```

```
126 #ifdef CONFIG_IP_FORWARD
127 struct ip_mib ip_statistics={1,64,};    /* Forwarding=Yes, Default TTL=64 */
128 #else
129 struct ip_mib ip_statistics={0,64,};    /* Forwarding=No, Default TTL=64 */
130 #endif
```

ip_statistics变量是对IP协议相关信息的统计，这是一个ip_mib结构类型的变量，ip_mib结构定义在net/inet/snmp.h中，其前两个字段如下：第一个字段取值表示是否支持数据包转发，第二个字段取值表示本地在创建IP首部时，TTL字段默认值。以上126-130行代码的意义就很明显了。如果定义了CONFIG_IP_FORWARD宏，则支持数据包转发，IpForwarding字段设置为1，否则设置为0；而默认TTL值设置为64。

```
/*net/inet/snmp.h*/
25  struct ip_mib
26  {
27      unsigned long   IpForwarding;
28      unsigned long   IpDefaultTTL;

        ......
46  };
```

RFC791是IP协议的正式说明文档。有关IP首部格式以及IP选项在第一章中介绍ip.h头文件时已经交待的比较详细，此处不多作说明。如果读者在下面介绍中对某些地方不熟悉，建议重新阅读第一章中有关IP协议的介绍，或者直接阅读文档RFC791。

```
131 /*
132  *  Handle the issuing of an ioctl() request
133  *  for the ip device. This is scheduled to
134  *  disappear
135  */

136 int ip_ioctl(struct sock *sk, int cmd, unsigned long arg)
137 {
138     switch(cmd)
139     {
140         default:
141             return(-EINVAL);
142     }
143 }
```

ip_ioctl函数用于对IP协议相关参数进行设置，本版本不支持任何参数设置功能。

```
144 /* these two routines will do routing. */
```

```
145 static void
146 strict_route(struct iphdr *iph, struct options *opt)
147 {
148 }

149 static void
150 loose_route(struct iphdr *iph, struct options *opt)
151 {
152 }

153 /* This routine will check to see if we have lost a gateway. */
154 void
155 ip_route_check(unsigned long daddr)
156 {
157 }
```

strict_route函数处理IP精确源站路由选项，loose_route函数处理松散源站路由选项，这两个选项本版本也未提供支持。函数实现上作为空体实现。ip_route_check按其函数前注释内容看，用于检查在路由过程中是否漏记一个中间路由器或者网关，但这个解释本身就很含糊，由于没有给出函数实现，我们也就确切知道该函数的作用。不管他！

```
158 #if 0
159 /* this routine puts the options at the end of an ip header. */
160 static int
161 build_options(struct iphdr *iph, struct options *opt)
162 {
163   unsigned char *ptr;
164   /* currently we don't support any options. */
      //目前我们不支持任何IP选项。
165   ptr = (unsigned char *)(iph+1);
166   *ptr = 0;
167   return (4);
168 }
169 #endif
```

build_options函数被调用创建IP选项，IP选项是紧跟在IP首部之后最大长度可达40字节的内容，从函数实现来看，本版本尚不支持IP选项，所以这个函数形如空体。注意返回为4，由于表示IP首部及其选项总长度的字段是以4字节为单位的，而IP首部为20字节，所以选项长度必须为4的倍数。

```
170 /*
171  *  Take an skb, and fill in the MAC header.
172  */
```

```
173 static int ip_send(struct sk_buff *skb, unsigned long daddr, int len,
                    struct device *dev, unsigned long saddr)
174 {
```

参数说明：
skb：待创建MAC首部的数据包。
daddr：数据包下一站IP地址，注意不同于最终接收端地址。
len：IP首部及其负载长度。
dev：数据包本地发送设备接口。
saddr：本地IP地址。这个参数没有被使用。

```
175     int mac = 0;
176     skb->dev = dev;
177     skb->arp = 1;
178     if (dev->hard_header)
179     {
180         /*
181          * Build a hardware header.Source address is our mac, destination
unknown
182          *      (rebuild header will sort this out)
183          */
184         mac = dev->hard_header(skb->data, dev, ETH_P_IP, NULL, NULL, len, skb);
185         if (mac < 0)
186         {
187             mac = -mac;
188             skb->arp = 0;
189             skb->raddr = daddr; /* next routing address */
190         }
191     }
192     return mac;
193 }
```

ip_send并不是如函数名本身表示的意义要发送一个数据包,而是仅仅完成创建MAC首部。MAC首部的创建主要还是通过调用dev->hard_header指向的函数完成的，在device结构被创建时，该函数指针被初始化为eth_header（eth.c),但是网络设备驱动程序可以更改该函数指针值，提供字节的MAC首部创建函数。我们在分析到eth_header时再进行如何创建MAC首部的说明，此处不进行跟踪。如果MAC首部创建成功，则eth_header或者驱动程序提供的函数必须返回MAC首部长度值(14字节)，如果创建过程中出现错误，则返回对应的相反值(即-14)，这样调用者(如ip_send)可以根据返回值的正负判断出MAC首部是否创建成功。此处对于MAC地址首部的创建，sk_buff结构arp字段起着标志作用，如果该字段为1，则表示数据包MAC首部已经成功完成创建，否则表示MAC首部仍待完成，在驱动程序发送数据包时，其首先检查该字段，如果该字段为0，则表示MAC首部没有创建成功，此时就要暂缓该数据包的发送，将数据包缓存到ARP缓存队列中，发出ARP请求，请求目的地址对应的MAC地址，完成MAC首部

的创建工作，ARP协议实现模块在接收到一个ARP应答后，完成对应数据包的MAC首部创建后，将数据包重新交给驱动程序，从而将数据包最后发送出去。那么这儿又有一个值得注意的问题，一般而言，通常我们请求的MAC地址与数据包的最终地址对应的MAC地址是不同的，在进行数据包转发时，这是经常发生的事情。例如数据包进行一个中间路由器转发时，数据包最终地址（即IP首部中的目的地址字段值）是最后接收该数据包的远端地址，而在这次的发送过程中，实际上接收者是这个中间路由器，实际上进行ARP请求时，也是对这个中间路由器的MAC地址进行请求，只有最终远端主机与本机在同一个链路上时，MAC首部中目的地址才是最终接收者对应的MAC地址。那么软件实现上是如何进行区分的呢？如果我们仔细观察sk_buf结构字段的话，就会发现解决方式，我们将sk_buff结构定义相关字段列出如下：

```
/*include/linux/skbuff.h*/
30  struct sk_buff {

    ……
56    unsigned long         saddr;
57    unsigned long         daddr;
58    unsigned long         raddr;      /* next hop addr */

    ……
75  };
```

问题就很明显了，saddr字段始终表示数据包最初发送端IP地址，daddr字段表示数据包最终接收端IP地址，而raddr就表示下一站IP地址，这个下一站IP地址就是中间路由器的IP地址。只有当下一站就是最终数据包接收端时，daddr才等于raddr。MAC首部创建时始终使用raddr字段，而查询路由表时始终使用daddr。

ip_send函数中189行对raddr赋值中使用的参数addr表示的就是下一站IP地址，而非最终接收端IP地址，这一点要注意。

```
194 int ip_id_count = 0;
```

ip_id_count变量用于IP首部中标识字段赋值。这个字段用于标志数据包的唯一性，在发生数据包分片时，各分片的重新组装就需要根据该字段进行判断大家原先是否从同一个数据包中分出来的。这是一个循环递增的变量，由于表示范围有限，当然一段时间后，又会绕回来，但不会造成问题，因为标识字段只是起小范围或者小时间段内的区分作用。

```
195 /*
196  * This routine builds the appropriate hardware/IP headers for
197  * the routine.  It assumes that if *dev != NULL then the
198  * protocol knows what it's doing, otherwise it uses the
199  * routing/ARP tables to select a device struct.
200  */
201 int ip_build_header(struct sk_buff *skb, unsigned long saddr,
             unsigned long daddr,
202          struct device **dev, int type, struct options *opt, int len,
```

```
                int tos, int ttl)
203 {
```

参数说明：
skb：待创建首部的数据包。
saddr：数据包发送的最初源地址，由于被转发的数据包无需进行IP首部创建，所以只有本地发送的数据包才需要调用该函数进行处理,所以此处的saddr参数表示的就是本地IP地址。
daddr：最终远端接收端IP地址。
dev：这是一个指针类型变量的地址，由本函数对其进行赋值。本函数将根据daddr查询路由表，将返回的表项中绑定的接口赋值给dev参数。
type：上层协议类型，如TCP为6，UDP为17。
opt：IP选项，本版本不支持，这个参数就是一个空架子。
len：IP首部和IP负载总长度。
tos，ttl：这两个参数对应IP首部中tos，ttl字段值。TOS表示服务类型，TTL表示跳数。

这个函数的长度有些“不友好”，本来很简单的一件事，费这么大篇幅，真是有些小题大做了。这个函数就是创建MAC首部和IP首部。由于MAC首部的创建是通过调用ip_send函数完成（一行代码），所以本函数中绝大部分代码就是在进行IP首部字段初始化。那我们看看究竟IP首部中什么字段需要这么大的篇幅来进行初始化。

```
204     static struct options optmem;
205     struct iphdr *iph;
206     struct rtable *rt;
207     unsigned char *buff;
208     unsigned long raddr;
209     int tmp;
210     unsigned long src;

211     buff = skb->data;

212     /*
213      *  See if we need to look up the device.
214      */

215 #ifdef CONFIG_INET_MULTICAST
216     if(MULTICAST(daddr) && *dev==NULL && skb->sk && *skb->sk->ip_mc_name)
217         *dev=dev_get(skb->sk->ip_mc_name);
218 #endif
```

215-218行是对多播地址的处理，sock结构ip_mc_name在初始化时（inet_create）被设置为空，该字段表示当数据包发往多播组时所使用接口的名称，dev_get函数就是根据该接口名获得接口设备对应的device结构，当然这要在参数dev指向NULL时。

```
219     if (*dev == NULL)
```

```
220     {
221         if(skb->localroute)
222             rt = ip_rt_local(daddr, &optmem, &src);
223         else
224             rt = ip_rt_route(daddr, &optmem, &src);
225         if (rt == NULL)
226         {
227             ip_statistics.IpOutNoRoutes++;
228             return(-ENETUNREACH);
229         }

230         *dev = rt->rt_dev;
231         /*
232          *  If the frame is from us and going off machine it MUST MUST MUST
233          *  have the output device ip address and never the loopback
234          */
235         if (LOOPBACK(saddr) && !LOOPBACK(daddr))
236             saddr = src;/*rt->rt_dev->pa_addr;*/
237         raddr = rt->rt_gateway;

238         opt = &optmem;
239     }
```

219-239行代码对dev参数进行初始化，并且获得下一站IP地址（对raddr变量初始化）。221行根据路由类型（本地路由？）调用相关函数进行路由表查询，如果sk_buff结构中localroute设置为1，就表示这是一个本地路由，即数据包最终接收端和本地发送端在同一个链路上,此时调用ip_rt_local函数进行路由表查询,否则调用ip_rt_route函数进行查询，这两个函数实现基本相同，差别只在于个别语句被调整了顺序，在前文中对route.c文件分析中，我们已经对两个函数进行了比较，读者可以参考前文中的相关分析进行理解。如果查询到一个表项，222或者224行src参数将被设置为本地接口IP地址。如果没有寻找到一个合适的路由表项（225行检查），则错误返回。230行对dev参数进行初始化，值为到达下一站的接口设备。LOOPBACK宏检查IP地址是不是一个回环地址（127打头的地址，见122行代码），235行检查远端地址是不是一个回环地址，如果不是，而本地源端地址被设置为回环地址，则需要对本地地址进行修改，236行将本地接口地址作为了源端地址使用，这是必要的。因为数据包要发送到网络上，不能使用127.0.0.1之类的地址。最后237将raddr地址设置为下一站路由器IP地址，注意如果最终目的主机是在同一链路上，则rt->rt_gateway为0，此时raddr即被赋值为0，下文代码中将对此进行检查。目前我们不关心选项，本版本也不对此进行支持，所以238行代码主要还是为了代码一致性或者此后代码扩展考虑。

```
240     else
241     {
```

这个else语句对应条件为*dev!=NULL,即函数调用时已经指定了发送接口设备，此时仍然需要进行路由表查询，寻找下一站IP地址，只不过无需进行接口设备的初始化（即无需对dev进行赋值了），如下253行代码完成的功能如同235行，255行代码对raddr变量进行初始化。

注意不同于if块的地方是，如果没有寻找到一个合适的表项，我们并不出错返回，而是进行执行，因为我们已经知道发送数据包的接口设备，既然如此，如果没有找到一个合适的路由表项，就认为最终目的主机在同一个链路上。255行代码在rt为NULL时将raddr设置为0，就是表达这个意思。

```
242         /*
243          *  We still need the address of the first hop.
244          */
245         if(skb->localroute)
246             rt = ip_rt_local(daddr, &optmem, &src);
247         else
248             rt = ip_rt_route(daddr, &optmem, &src);
249         /*
250          *  If the frame is from us and going off machine it MUST MUST MUST
251          *  have the output device ip address and never the loopback
252          */
253         if (LOOPBACK(saddr) && !LOOPBACK(daddr))
254             saddr = src;/*rt->rt_dev->pa_addr;*/

255         raddr = (rt == NULL) ? 0 : rt->rt_gateway;
256     }

257     /*
258      *  No source addr so make it our addr
259      */
260     if (saddr == 0)
261         saddr = src;
```

如果还没有指定本地地址，则设置源端地址为本地接口地址。注意235和253行只是对saddr为回环地址进行了检查，没有检查saddr为0的情况，所以此处对此进行检测。在发送数据包时，除非主机在使用BOOTP协议进行网络启动，否则源端IP地址为0的数据包是不合法的。

```
262     /*
263      *  No gateway so aim at the real destination
264      */
265     if (raddr == 0)
266         raddr = daddr;
```

注意raddr为0,就表示最终目的主机和发送主机(即本机)在同一个链路上,所以我们将raddr变量设置为最终目的主机的IP地址。这样在进行目的MAC地址查询时，就可以直接使用目的主机MAC地址从而将数据包直接发送给目的主机。

```
267     /*
268      *  Now build the MAC header.
```

```
269      */

270     tmp = ip_send(skb, raddr, len, *dev, saddr);
```

调用ip_send函数进行MAC首部创建，注意目的地址使用的是下一站地址（raddr），即MAC首部中目的主机地址对应下一站主机MAC地址，而非最终接收端MAC地址，我想对于TCP/IP协议族有所了解的读者都应该知道这一点。ip_send函数返回值为MAC首部长度，错误被封装在ip_send函数中了，所以ip_send函数始终返回一个正数。如果MAC首部创建失败，那么只有一个原因，下一站主机的MAC地址不知道，此时就需要使用ARP请求报文进行MAC地址请求。不过这个ARP请求报文的发送是在数据包将要被发送时进行的，当前如果MAC首部没有被成功创建的话，数据包对应sk_buff结构中arp字段被设置为0，以后驱动程序发送该数据包会根据该字段值决定是先缓存到ARP缓存队列中（arp字段值为0），发送一个ARP请求报文，还是可以立刻将该数据包发送出去（arp字段值为1）。

```
271     buff += tmp;
272     len -= tmp;

273     /*
274      *  Book keeping
275      */

276     skb->dev = *dev;
277     skb->saddr = saddr;
278     if (skb->sk)
279         skb->sk->saddr = saddr;
```

更新sk_buff结构中相关字段值。272行代码无关紧要，而且表达的意义不明，其实len参数是一个值类型的参数，此处这样对该参数进行修改不会对调用者产生任何影响，而ip_build_header函数下面代码中也没有再使用到该参数，所以272行代码显得有些多余。

```
280     /*
281      *  Now build the IP header.
282      */

283     /*
284      *  If we are using IPPROTO_RAW, then we don't need an IP header, since
285      *  one is being supplied to us by the user
286      */

287     if(type == IPPROTO_RAW)
288         return (tmp);
```

对于RAW类型套接字，不使用IP协议，所以在创建MAC首部后，可以直接返回了。注意RAW类型套接字直接使用链路层协议进行数据的传输。

```
289     iph = (struct iphdr *)buff;
290     iph->version  = 4;
291     iph->tos      = tos;
292     iph->frag_off = 0;
293     iph->ttl      = ttl;
294     iph->daddr    = daddr;
295     iph->saddr    = saddr;
296     iph->protocol = type;
297     iph->ihl      = 5;
298     skb->ip_hdr   = iph;
```

289-298行代码进行IP首部的创建。结合IP首部格式定义，这段代码很容易理解。注意ihl字段是以4字节为单位的，所以赋值为5表示IP首部长度为20字节。

```
299     /* Setup the IP options. */
300 #ifdef Not_Yet_Avail
301     build_options(iph, opt);
302 #endif

303     return(20 + tmp);   /* IP header plus MAC header size */
304 }
```

ip_build_header函数返回值为MAC首部和IP首部总长度（303行）。本版本不支持IP选项，所以虽然在301行有对build_options函数的调用，但该函数只是返回4，并未进行任何有效的IP选项创建工作。

build_options函数用于本地对IP选项进行创建，而do_options函数用于处理接收到的IP选项。下面我们对do_options函数进行分析。该函数代码也较长，不过只要了解各种IP选项的格式,处理上如同流水线,不存在理解上的问题。有关IP各种选项的格式在第一章中介绍ip.h头文件时有较为详细的介绍，读者也可以直接阅读RFC791文档进行熟悉，这对do_options函数的理解将大有好处。

```
305 static int
306 do_options(struct iphdr *iph, struct options *opt)
307 {
```

参数说明：
iph：IP首部
opt：将解析出的选项填入该参数指向的缓冲区中，这个参数是一个options类型的参数，options结构定义在include/linux/ip.h中，再次给出如下：

```
/*include/linux/ip.h*/
52  struct options {
53    struct route      record_route;
```

```
54    struct route      loose_route;
55    struct route      strict_route;
56    struct timestamp  tstamp;
57    unsigned short    security;
58    unsigned short    compartment;
59    unsigned short    handling;
60    unsigned short    stream;
61    unsigned      tcc;
62  };
```

options结构定义中route，timestamp结构定义在同一个文件中。此处不再列出。下面我们即进入do_options函数的分析。

```
308   unsigned char *buff;
309   int done = 0;
310   int i, len = sizeof(struct iphdr);

311   /* Zero out the options. */
312   opt->record_route.route_size = 0;
313   opt->loose_route.route_size  = 0;
314   opt->strict_route.route_size = 0;
315   opt->tstamp.ptr              = 0;
316   opt->security                = 0;
317   opt->compartment             = 0;
318   opt->handling                = 0;
319   opt->stream                  = 0;
320   opt->tcc                     = 0;
321   return(0);
```

312-320对参数opt各字段进行初始化清零操作。321行代码很“搞”，直接返回了，原因是本版本并不支持IP选项，所以对于IP选项并不进行处理。自322行以下的代码都是为以后代码扩展做准备。我们权当321行不存在，继续下面的分析。

```
322   /* Advance the pointer to start at the options. */
323   buff = (unsigned char *)(iph + 1);
```

buff变量初始化为指向IP选项，IP选项紧跟在IP首部之后。

```
324   /* Now start the processing. */
325   while (!done && len < iph->ihl*4) switch(*buff) {
```

当处理过程中，遭遇到IPOPT_END选项，就表示选项处理完毕。这个IPOPT_END选项标志着IP选项的结束。变量len表示在310行初始化为IP首部长度，而IP首部中ihl字段表示IP首部以及IP选项的总长度，如果这个总长度大于IP首部长度，则表示存在IP选项。

```
326     case IPOPT_END:
327         done = 1;
328         break;
```

IPOPT_END（Type=0）选项标志IP选项的结束，此时设置done标志变量，从而退出325行while循环。

```
329     case IPOPT_NOOP:
330         buff++;
331         len++;
332         break;
```

IPOPT_NOOP（Type=1）选项表示无操作选项，这个选项的作用在于填充，使得其他有效选项进行内存位置对齐，如对齐到4字节的边界上。

```
333     case IPOPT_SEC:
334         buff++;
335         if (*buff != 11) return(1);
336         buff++;
337         opt->security = ntohs(*(unsigned short *)buff);
338         buff += 2;
339         opt->compartment = ntohs(*(unsigned short *)buff);
340         buff += 2;
341         opt->handling = ntohs(*(unsigned short *)buff);
342         buff += 2;
343         opt->tcc = ((*buff) << 16) + ntohs(*(unsigned short *)(buff+1));
344         buff += 3;
345         len += 11;
346         break;
```

IPOPT_SEC(Type=130)安全选项为主机进行数据包安全传输提供了一种策略。该选项又分一系列乱七八糟的子字段。对于334-345行代码的含义读者参考IP安全选项格式很容易理解，此处不做解释。

```
347     case IPOPT_LSRR:
348         buff++;
349         if ((*buff - 3)% 4 != 0) return(1);
350         len += *buff;
351         opt->loose_route.route_size = (*buff -3)/4;
352         buff++;
353         if (*buff % 4 != 0) return(1);
354         opt->loose_route.pointer = *buff/4 - 1;
355         buff++;
356         buff++;
```

```
357         for (i = 0; i < opt->loose_route.route_size; i++) {
358             if(i>=MAX_ROUTE)
359                 return(1);
360             opt->loose_route.route[i] = *(unsigned long *)buff;
361             buff += 4;
362         }
363         break;
```

IPOPT_LSRR(Type=131)松源路由选项，此处所进行主要工作在357-361行代码上，将该IP选项中存储的路由器地址复制到opt参数对应字段中。

```
364     case IPOPT_SSRR:
365         buff++;
366         if ((*buff - 3)% 4 != 0) return(1);
367         len += *buff;
368         opt->strict_route.route_size = (*buff -3)/4;
369         buff++;
370         if (*buff % 4 != 0) return(1);
371         opt->strict_route.pointer = *buff/4 - 1;
372         buff++;
373         buff++;
374         for (i = 0; i < opt->strict_route.route_size; i++) {
375             if(i>=MAX_ROUTE)
376                 return(1);
377             opt->strict_route.route[i] = *(unsigned long *)buff;
378             buff += 4;
379         }
380         break;
```

IPOPT_SSRR(Type=137)紧源路由选项，这个选项格式以及此处的处理方式同IPOPT_LSRR选项。

```
381     case IPOPT_RR:
382         buff++;
383         if ((*buff - 3)% 4 != 0) return(1);
384         len += *buff;
385         opt->record_route.route_size = (*buff -3)/4;
386         buff++;
387         if (*buff % 4 != 0) return(1);
388         opt->record_route.pointer = *buff/4 - 1;
389         buff++;
390         buff++;
391         for (i = 0; i < opt->record_route.route_size; i++) {
392             if(i>=MAX_ROUTE)
```

```
393                 return 1;
394             opt->record_route.route[i] = *(unsigned long *)buff;
395             buff += 4;
396         }
397         break;
```

IPOPT_RR(Type=7)路由器地址记录选项，该选项要求发送主机以及中间每个路由器在发送数据包之前都要将其出口地址保存到该选项缓冲区中。如果本地支持该选项，并且进行数据包转发，则同样需要将本地发送数据包的接口IP地址写入到该选项缓冲区中对应位置。此处对该选项的处理方式仅仅是记录已有的地址，保存到opt参数对应字段中。

```
398     case IPOPT_SID:
399         len += 4;
400         buff +=2;
401         opt->stream = *(unsigned short *)buff;
402         buff += 2;
403         break;
```

IPOPT_SID(Type=136)流标志选项，该选项用一个ID号标志一个流。此处的处理方式仅将该ID保存到opt参数stream字段中。

```
404     case IPOPT_TIMESTAMP:
405         buff++;
406         len += *buff;
407         if (*buff % 4 != 0) return(1);
408         opt->tstamp.len = *buff / 4 - 1;
409         buff++;
410         if ((*buff - 1) % 4 != 0) return(1);
411         opt->tstamp.ptr = (*buff-1)/4;
412         buff++;
413         opt->tstamp.x.full_char = *buff;
414         buff++;
415         for (i = 0; i < opt->tstamp.len; i++) {
416             opt->tstamp.data[i] = *(unsigned long *)buff;
417             buff += 4;
418         }
419         break;
```

IPOPT_TIMESTAMP(Type=68)时间戳选项，这个选项本身又分为三个字选项，表示记录时间戳的方式：其一为只记载时间戳；其二同时记载IP地址和时间戳；其三记载指定的IP地址的时间戳。此处处理方式（415-418）则是不管以上何种子类型，将选项缓冲区中内容都保存下来，具体的分析调用do_options的函数处理。

```
420     default:
```

```
421         return(1);
422   }

423   if (opt->record_route.route_size == 0) {
424     if (opt->strict_route.route_size != 0) {
425         memcpy(&(opt->record_route), &(opt->strict_route),
426                             sizeof(opt->record_route));
427     } else if (opt->loose_route.route_size != 0) {
428         memcpy(&(opt->record_route), &(opt->loose_route),
429                             sizeof(opt->record_route));
430     }
431   }
```

由于无论松源还是紧源路由选项，其中都包含有之间路由器IP地址，而路由记录字段从以上代码来看，只有使用了路由记录选项时方不为空，由于选项缓冲区大小有限（最大40字节），所以对于这些需使用大量空间的选项而言，不可同时使用，既然紧源和松源路由选项都含有中间路由器的地址，那么也可以“勉强”的将这些地址作为路由记录选项的值。不过需要提出的是，423-431行代码的操作是实现相关的。此处删除这段代码也没有问题。

```
432   if (opt->strict_route.route_size != 0 &&
433       opt->strict_route.route_size != opt->strict_route.pointer) {
434     strict_route(iph, opt);
435     return(0);
436   }
```

432-436行对紧源路由选项进行检查，如果使用了该选项，而且选项缓冲区还有空间，则进行该选项的处理（首先从缓冲区中取下一个可用IP地址，同时将本地发送接口IP地址覆盖刚才使用的IP地址对应的缓冲区空间）。具体处理函数为strict_rouet，这个函数前文中已作处理，实现为空函数，即本版本尚未对这些选项提供支持。

```
437   if (opt->loose_route.route_size != 0 &&
438       opt->loose_route.route_size != opt->loose_route.pointer) {
439     loose_route(iph, opt);
440     return(0);
441   }
```

437-441行处理松源路由选项，处理方式同上，此处具体调用loose_route函数，该函数同样实现为空函数。

```
442   return(0);
443 }
```

do_options函数处理接收到的数据包中可能包含的IP选项，对于该函数中选项的处理进行了比较简单的讨论，原因是，其中不存在理解上的问题，读者只要对照各选项对应的格式，代

码一目了然。RFC791文档是一个很好的参考源。

```
444 /*
445  * This routine does all the checksum computations that don't
446  * require anything special (like copying or special headers).
447  */

448 unsigned short ip_compute_csum(unsigned char * buff, int len)
449 {
450     unsigned long sum = 0;

451     /* Do the first multiple of 4 bytes and convert to 16 bits. */
452     if (len > 3)
453     {
454         __asm__("clc\n"
455         "1:\t"
456         "lodsl\n\t"
457         "adcl %%eax, %%ebx\n\t"
458         "loop 1b\n\t"
459         "adcl $0, %%ebx\n\t"
460         "movl %%ebx, %%eax\n\t"
461         "shrl $16, %%eax\n\t"
462         "addw %%ax, %%bx\n\t"
463         "adcw $0, %%bx"
464         : "=b" (sum) , "=S" (buff)
465         : "0" (sum), "c" (len >> 2) ,"1" (buff)
466         : "ax", "cx", "si", "bx" );
467     }
468     if (len & 2)
469     {
470         __asm__("lodsw\n\t"
471         "addw %%ax, %%bx\n\t"
472         "adcw $0, %%bx"
473         : "=b" (sum), "=S" (buff)
474         : "0" (sum), "1" (buff)
475         : "bx", "ax", "si");
476     }
477     if (len & 1)
478     {
479         __asm__("lodsb\n\t"
480         "movb $0, %%ah\n\t"
481         "addw %%ax, %%bx\n\t"
482         "adcw $0, %%bx"
483         : "=b" (sum), "=S" (buff)
```

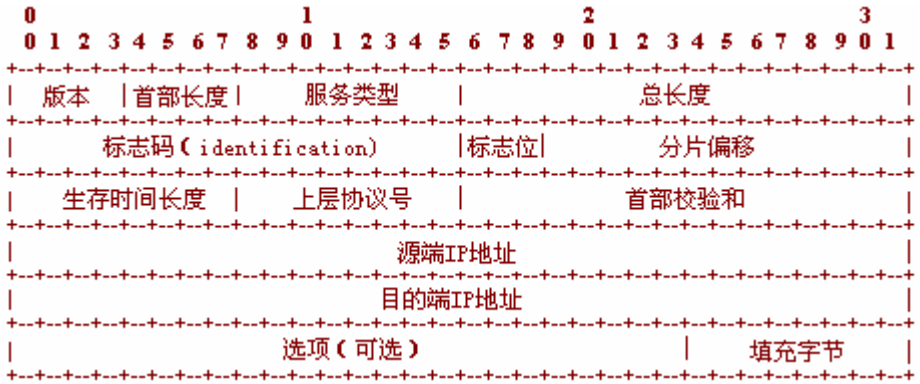
```
484          : "0" (sum), "1" (buff)
485          : "bx", "ax", "si");
486      }
487      sum =~sum;
488      return(sum & 0xffff);
489 }
```

ip_compute_csum函数用于计算IP校验和，注意对于IP校验，其计算范围只包括IP首部部分（参见下面的ip_send_check函数对ip_compute_csum调用时第二个参数的计算方式）。计算方式就是对数据进行简单相加，最后折叠为16比特，取反后返回。其他协议中校验和计算都采用这种算法。

```
490 /*
491  *  Generate a checksum for an outgoing IP datagram.
492  */

493 void ip_send_check(struct iphdr *iph)
494 {
495      iph->check = 0;
496      iph->check = ip_fast_csum((unsigned char *)iph, iph->ihl);
497 }
```

ip_compute_csum函数完成实际的校验和计算，而ip_send_check函数则完成对IP首部中校验和字段的赋值，其首先将IP首部中校验和字段清零，然后调用ip_compute_csum函数对IP首部计算校验和，将返回值赋值给IP首部中校验和字段。

下面我们进入IP协议实现中很重要的一个环节，对数据包进行分片和重组的实现。为了保证各主机公平共享传输介质，对于主机传输的最长数据长度都有一个最大限制，如对于以太网这个值就是我们通常所说的MTU（最大传输单元），目前设置值为1500字节。这个最大限制表示的IP首部及其负载的总长度，所以必须在IP协议实现模块中进行报文长度的检查，如果传输层传递下来的数据包长度大于这个最大值，则IP协议在将数据单元发往下一层（链路层）之前，必须对大数据包进行分片，从而将数据包长度限制在MTU之内。IP首部中专门设置有字段对IP分片进行表示。我们将IP首部格式定义重列如下：



IP首部格式中，如下字段用于分片处理：
标志码字段：用于表示分片原属的数据包，由一个大的数据包分片而得的所有小数据包其IP

首部中该字段均具有相同的值。
标志位MF（More Fragment）：如果设置为1，表示还有后续分片；只有最后一个分片中该标志位方才设置为0。对于没有进行分片的数据包，该字段值为0。
分片偏移字段：表示该分片中包含的数据在原大数据包中的偏移量，由于该字段只有13个比特，比总长度字段少3的比特，所以为了表示所有的数据偏移，该字段以8字节为单位，换句话说，一个分片中分片偏移字段表示的实际偏移量是该字段值乘以8，更进一步说，就是每个分片的数据长度均是8的倍数。这一点在创建分片时我们也可以看到，当然我们不能保证原来的大的数据包中数据长度总是8的倍数，所以最后一个分片中数据长度无此限制，但除了最后一个分片外，其他所有分片数据长度都必须是8的倍数，从而相对后续分片中IP首部中分片偏移方才有效！注意对于未进行分片的普通数据包该字段为0。

综合以上字段说明，我们可以对分片的特征进行表述，即如何判断一个接收的数据包是否是一个分片：
1〉 MF=1，偏移字段为0：是一个分片，且是第一个分片。
2〉 MF=1，偏移字段非0：是一个分片，处于中间位置。
3〉 MF=0，偏移字段非0：是一个分片，且是最后一个分片。
4〉 MF=0，偏移字段为0：不是分片，是一个普通数据包。

为了对下文中数据分片和重组进行分析，我们首先必须了解相关数据结构，所以下面我们将暂时脱离对ip.c文件的分析，转而进入对net/inet/ip.h文件的分析，这个对应的头文件中定义有表示分片的数据结构以及对分片进行重组时需要使用的结构。

/*net/inet/ip.h***/
```
1    /*
2     * INET        An implementation of the TCP/IP protocol suite for the LINUX
3     *        operating system.    INET is implemented using the    BSD Socket
4     *        interface as the means of communication with the user level.
5     *
6     *        Definitions for the IP module.
7     *
8     * Version:    @(#)ip.h  1.0.2       05/07/93
9     *
10    * Authors:    Ross Biro, <bir7@leland.Stanford.Edu>
11    *        Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
12    *        Alan Cox, <gw4pts@gw4pts.ampr.org>
13    *
14    *        This program is free software; you can redistribute it and/or
15    *        modify it under the terms of the GNU General Public License
16    *        as published by the Free Software Foundation; either version
17    *        2 of the License, or (at your option) any later version.
18    */
19   #ifndef _IP_H
20   #define _IP_H
```

```
21  #include <linux/ip.h>
22  #include <linux/config.h>

23  #ifndef _SNMP_H
24  #include "snmp.h"
25  #endif

26  #include "sock.h"   /* struct sock */

27  /* IP flags. */
28  #define IP_CE       0x8000          /* Flag: "Congestion"          */
29  #define IP_DF       0x4000          /* Flag: "Don't Fragment"      */
30  #define IP_MF           0x2000          /* Flag: "More Fragments"      */
31  #define IP_OFFSET       0x1FFF          /* "Fragment Offset" part      */

32  #define IP_FRAG_TIME        (30 * HZ)       /* fragment lifetime*/

33  #ifdef CONFIG_IP_MULTICAST
34  extern void             ip_mc_dropsocket(struct sock *);
35  extern void             ip_mc_dropdevice(struct device *dev);
36  extern int      ip_mc_procinfo(char *, char **, off_t, int);
37  #define MULTICAST(x)        (IN_MULTICAST(htonl(x)))
38  #endif
```

ipfrag结构用来表示一个分片，每当接收到一个分片时，就创建一个ipfrag结构对该分片进行封装，如果这个分片是接收到的第一个分片，则同时创建队列（这个队列由下面的ipq结构表示），对后续属于相同源的分片进行缓存和重组。

```
39  /* Describe an IP fragment. */
40  struct ipfrag {
41      int             offset;         /* offset of fragment in IP datagram    */
```

该分片的偏移地址，注意该地址并非一定等于对应分片数据包中IP首部中的offset值，因为可能数据存在重叠。这个offset是经过处理后的有效数据的偏移值。

```
42      int             end;        /* last byte of data in datagram    */
43      int             len;        /* length of this fragment          */
44      struct sk_buff *skb;                /* complete received fragment           */
45      unsigned char           *ptr;       /* pointer into real fragment data  */
46      struct ipfrag       *next;          /* linked list pointers             */
47      struct ipfrag       *prev;
48  };
```

ipfrag结构只是为分片重组提供支持，所以ipfrag结构本身并不包含数据，其通过skb字段指向该结构对应的分片数据包，ptr指针指向实际有效的数据，因为我们必须对数据重叠的情况进行处理，所以ipfrag结构中offset字段并非一定该结构对应分片数据包IP首部中偏移字段值，而是在进行数据重叠检查后赋予的值，一般情况下，这两个值是一样的。如果存在数据重叠现象，如本分片中前面100字节与前一个分片重叠了，则ipfrag结构中offset字段就要在对应数据包IP首部中所表示的偏移量的基础上加上100，相应的ptr数据指针也要加上100的偏移量，从而跳过前面的100字节数据。同理，ipfrag结构中len表示的也是可用的数据长度，end是经过计算后的实际数据结尾偏移。对于这些字段的理解，在分析实际操作代码时自会明白。

```
49  /* Describe an entry in the "incomplete datagrams" queue. */
50  struct ipq  {
51      unsigned char       *mac;       /* pointer to MAC header        */
52      struct iphdr  *iph;         /* pointer to IP header         */
53      int         len;        /* total length of original datagram*/
54      short           ihlen;      /* length of the IP header      */
55      short   maclen;         /* length of the MAC header     */
56      struct timer_list timer;/* when will this queue expire?     */
57      struct ipfrag       *fragments;     /* linked list of received fragments*/
58      struct ipq      *next;      /* linked list pointers         */
59      struct ipq      *prev;
60      struct device *dev;         /* Device - for icmp replies */
61  };
```

ipq结构表示一个队列，该队列中缓存的都是属于同一个源的分片数据包，所谓属于同一个源即这些分片都是来自同一个大数据包，他们IP首部中标识符字段都相同。在判断出接收到的数据包是一个分片数据包时，系统创建一个ipfrag结构表示这个分片数据包，如果这是接收到的属于同一个源的第一个分片时（注意此处第一个含义是指这个分片数据包是第一个接收到的分片数据包，而非从分片数据包所包含数据偏移量上考虑），那么就需要同时创建一个分片队列，从而对后续分片数据包进行接收和缓存，以便于最后对这些分片进行重组合并为原始的大数据包。ipq结构就表示这样一个分片队列，结构中mac指向一个MAC首部缓冲区，iph字段表示数据包IP首部，这个字段指向的IP首部包含了如下关键信息用于对后续接收的其他分片进行匹配：标识符，源，目的IP地址，上层使用协议。这四个信息将被用来判断一个新接收到分片属于哪个分片队列，从而将这个分片插入到合适的队列中。timer字段用于超时处理，在接收到第一个分片创建了分片队列后，该定时器即被启动，以后每当接收到一个新的分片，就对该定时器进行重置，如果在定时器到期时，没有接收到下一个分片，且队列中还缺分片，则对该队列中先有分片进行清空处理，即假设此次重组失败，发送端需要重新发送原始数据包（这是由传输层协议负责）。fragments字段即指向分片队列，分片队列中每个分片都是由一个ipfrag结构表示。next，prev字段用于ipq结构的连接，dev表示接收这些分片的网络设备（由于所有分片都来自同一个主机，当然我们可以预知这些分片都来自于同一个网络设备）。

```
62  extern int      backoff(int n);
```

```
63  extern void             ip_print(const struct iphdr *ip);
64  extern int          ip_ioctl(struct sock *sk, int cmd,
65                               unsigned long arg);
66  extern void             ip_route_check(unsigned long daddr);
67  extern int          ip_build_header(struct sk_buff *skb,
68                                      unsigned long saddr,
69                                      unsigned long daddr,
70                                      struct device **dev, int type,
71                                      struct options *opt, int len,
72                                      int tos,int ttl);
73  extern unsigned short       ip_compute_csum(unsigned char * buff, int len);
74  extern int          ip_rcv(struct sk_buff *skb, struct device *dev,
75                                    struct packet_type *pt);
76  extern void             ip_send_check(struct iphdr *ip);
77  extern int          ip_id_count;
78  extern void             ip_queue_xmit(struct sock *sk,
79                                        struct device *dev, struct sk_buff *skb,
80                                        int free);
81  extern int              ip_setsockopt(struct sock *sk, int level, int optname, char *optval, int
optlen);
82  extern int              ip_getsockopt(struct sock *sk, int level, int optname, char *optval, int
*optlen);
83  extern void             ip_init(void);

84  extern struct ip_mib        ip_statistics;
```

62-84行代码是对相关函数以及变量的声明。

```
85  /*
86   *  This is a version of ip_compute_csum() optimized for IP headers, which
87   *  always checksum on 4 octet boundaries.
88   *  Used by ip.c and slhc.c (the net driver module)
89   *  (Moved to here by bj0rn@blox.se)
90   */

91  static inline unsigned short ip_fast_csum(unsigned char * buff, int wlen)
92  {
93      unsigned long sum = 0;

94      if (wlen)
95      {
96      unsigned long bogus;
97        __asm__("clc\n"
```

```
98              "1:\t"
99              "lodsl\n\t"
100             "adcl %3, %0\n\t"
101             "decl %2\n\t"
102             "jne 1b\n\t"
103             "adcl $0, %0\n\t"
104             "movl %0, %3\n\t"
105             "shrl $16, %3\n\t"
106             "addw %w3, %w0\n\t"
107             "adcw $0, %w0"
108             : "=r" (sum), "=S" (buff), "=r" (wlen), "=a" (bogus)
109             : "0"    (sum),    "1" (buff),    "2" (wlen));
110         }
111         return (~sum) & 0xffff;
112 }
113 #endif     /* _IP_H */
```

/*net/inet/ip.h***/

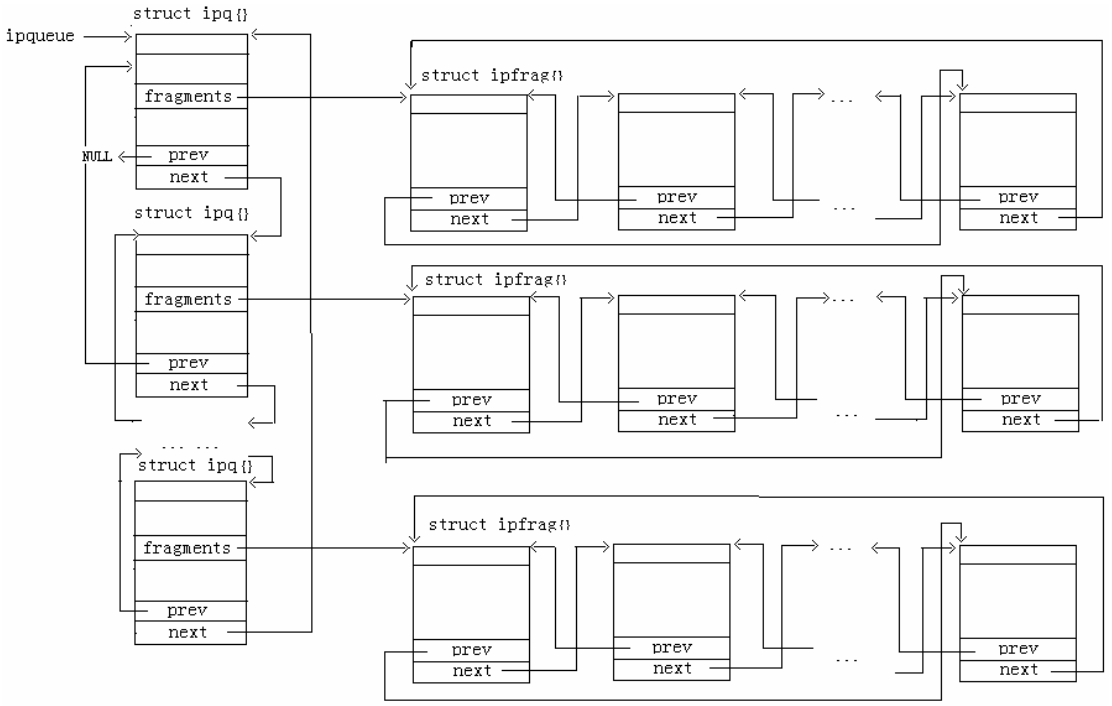
ip_fast_csum函数用于IP校验和的快速计算。因为IP校验和只计算IP首部，而IP首部长度总是4的倍数，所以可以避免多余的检查，实现IP校验和的快速计算。算法和TCP，UDP等相同。

在完成对分片相关结构的介绍后，下面我们进入IP协议实现模块中对于数据包分片和重组的处理。

```
498 /****************** Fragment Handlers From NET2E******************/
499 /*
500  *    This fragment handler is a bit of a heap. On the other hand it works quite
501  *    happily and handles things quite well.
502  */

503 static struct ipq *ipqueue = NULL;              /* IP fragment queue      */
```

ipqueue变量用于创建ipq结构类型的变量，在前文中我们已经提到ipq结构指向的队列是由ipfrag结构构成的，每个ipq结构指向的队列及其本身表示一个被分片，等待重组的数据包；而ipqueue表示正在重组多个数据包的情况，ipqueue变量指向的每个ipq结构都表示一个正在等待重组的数据包。相关结构之间的关系如下图所示。

注意：ipqueue指向的ipq结构队列中，第一个元素的prev字段指向NULL。




```
504 /*
505  *  Create a new fragment entry.
506  */

507 static struct ipfrag *ip_frag_create(int offset, int end, struct sk_buff *skb,
unsigned char *ptr)
508 {
509     struct ipfrag *fp;

510     fp = (struct ipfrag *) kmalloc(sizeof(struct ipfrag), GFP_ATOMIC);
511     if (fp == NULL)
512     {
513         printk("IP: frag_create: no memory left !\n");
514         return(NULL);
515     }
516     memset(fp, 0, sizeof(struct ipfrag));

517     /* Fill in the structure. */
518     fp->offset = offset;
519     fp->end = end;
520     fp->len = end - offset;
521     fp->skb = skb;
522     fp->ptr = ptr;

523     return(fp);
524 }
```

ip_frag_create函数用于创建一个新的ipfrag结构用于表示新接收到的分片数据包。结合ipfrag结构定义，该函数实现非常明显，无需多做说明。需要注意的是调用该函数的其他函数对相关传入参数的设置。

```
525 /*
526  *  Find the correct entry in the "incomplete datagrams" queue for
527  *  this IP datagram, and return the queue entry address if found.
528  */

529 static struct ipq *ip_find(struct iphdr *iph)
530 {
531     struct ipq *qp;
532     struct ipq *qplast;

533     cli();
534     qplast = NULL;
535     for(qp = ipqueue; qp != NULL; qplast = qp, qp = qp->next)
536     {
537         if (iph->id== qp->iph->id && iph->saddr == qp->iph->saddr &&
538             iph->daddr == qp->iph->daddr && iph->protocol ==
qp->iph->protocol)
539         {
540             del_timer(&qp->timer);  /* So it doesn't vanish on us. The timer
will be reset anyway */
541             sti();
542             return(qp);
543         }
544     }
545     sti();
546     return(NULL);
547 }
```

在接收到一个新的分片数据包后，内核通过调用ip_find函数查询该分片数据包所对应的ipfrag队列，这个队列的头部有一个ipq结构类型指向，具体请参考上图。ip_find函数输入的参数是被接收分片数据包的IP首部，通过对IP首部中标识符字段，源，目的IP地址字段以及上层协议字段进行比较，返回相应的ipq结构。注意540行将定时器清零，调用ip_find函数的其他函数会相应的对定时器进行进一步的处理。所以540行代码并非十分重要。没有也可以，下面在分析到ip_defrag函数时，读者即会看到这一点。

```
548 /*
549  *  Remove an entry from the "incomplete datagrams" queue, either
550  *  because we completed, reassembled and processed it, or because
551  *  it timed out.
```

```
552  */

553 static void ip_free(struct ipq *qp)
554 {
```

ip_free对由一个ipq指向的分片队列中各分片进行释放。参数qp即表示这个ipq结构。释放的原因可能因为所有分片都已到达，而且已经完成分片重组，或者是在定时器到期之前没有接收到需要的其他分片数据包（即极有可能发生分片数据包丢失）。

```
555     struct ipfrag *fp;
556     struct ipfrag *xp;

557     /*
558      * Stop the timer for this entry.
559      */

560     del_timer(&qp->timer);
```

既然是将要释放所有分片，定时器已经没有存在的必要，故删除之。

```
561     /* Remove this entry from the "incomplete datagrams" queue. */
562     cli();
563     if (qp->prev == NULL)
564     {
565         ipqueue = qp->next;
566         if (ipqueue != NULL)
567             ipqueue->prev = NULL;
568     }
```

ipqueue指向的ipq结构类型队列中第一个ipq结构元素的prev字段设置为NULL，所以此处对要删除的这个ipq结构元素的prev字段进行检查，查看其是否是队列第一个元素，具体处理上比较简单。如果不是第一个元素，那么就需要按照通常方式对双向指针进行调整。

```
569     else
570     {
571         qp->prev->next = qp->next;
572         if (qp->next != NULL)
573             qp->next->prev = qp->prev;
574     }

575     /* Release all fragment data. */

576     fp = qp->fragments;
577     while (fp != NULL)
578     {
```

```
579         xp = fp->next;
580         IS_SKB(fp->skb);
581         kfree_skb(fp->skb,FREE_READ);
582         kfree_s(fp, sizeof(struct ipfrag));
583         fp = xp;
584     }
```

576-584行代码释放对应ipq元素指向的ipfrag结构队列,这个队列是由ipq结构中fragments字段指向的。所以代码实现上也比较简单，就是遍历该ipfrag结构队列，对其中每个结构进行内存释放操作。

```
585     /* Release the MAC header. */
586     kfree_s(qp->mac, qp->maclen);

587     /* Release the IP header. */
588     kfree_s(qp->iph, qp->ihlen + 8);

589     /* Finally, release the queue descriptor itself. */
590     kfree_s(qp, sizeof(struct ipq));
591     sti();
592 }
```

586,588,590行代码对其他内存使用空间进行释放。ipq结构中mac,iph字段都是通过kmalloc函数分配的，所以在释放ipq结构之前，必须对这些字段指向的内存空间进行释放。

```
593 /*
594  *  Oops- a fragment queue timed out.  Kill it and send an ICMP reply.
595  */

596 static void ip_expire(unsigned long arg)
597 {
598     struct ipq *qp;

599     qp = (struct ipq *)arg;

600     /*
601      *  Send an ICMP "Fragment Reassembly Timeout" message.
602      */

603     ip_statistics.IpReasmTimeout++;
604     ip_statistics.IpReasmFails++;
605     /* This if is always true... shrug */
606     if(qp->fragments!=NULL)
607         icmp_send(qp->fragments->skb,ICMP_TIME_EXCEEDED,
```

```
608                     ICMP_EXC_FRAGTIME, 0, qp->dev);

609     /*
610      *  Nuke the fragment queue.
611      */
612     ip_free(qp);
613 }
```

前文中我们说到,对于数据包重组内核设置有一个定时器,如果定时器到期这段时间间隔内,没有接收到其他数据包,就表示可能分片数据包传输出现问题,我们不能永久等待一个可能永远无法到达的分片数据包,所以如果定时器超时,就对分片数据包队列进行释放。以防系统资源(被分片使用的内存空间等)被永久保留,从而造成资源不可用。而每当接收到一个新的分片数据包后,都会对定时器进行重置。换句话说,如果分片在规定的时间内到达,是不会发生定时器超时事件的。一旦发生定时器超时事件,就调用ip_expire函数进行处理,对目前接收到的分片数据包进行释放,从而释放表示这些分片所使用的内存空间。具体的释放工作是通过调用ip_free函数完成的。606-608行代码发送一个ICMP错误报文,表示分片数据包重组超时。

每个ipq结构对应一个带重组的分片数据包队列以及相关辅助工具,如定时器,所以定时器的设置是在ipq结构中完成的,这一点也可以从ipq结构的定义看出,而且我们知道每当接收到一个新的分片数据包,该定时器都会被重置,直到接收到所有的分片。但是定时器的设置是在何处完成的呢?下面介绍的ip_create函数将给出答案。ip_create函数用于最初接收到一个分片数据包时,创建一个ipq结构来对将要到达的其他分片进行缓存,定时器的设置即在该函数中进行。

```
614 /*
615  *  Add an entry to the 'ipq' queue for a newly received IP datagram.
616  *  We will (hopefully :-) receive all other fragments of this datagram
617  *  in time, so we just create a queue for this datagram, in which we
618  *  will insert the received fragments at their respective positions.
619  */

620 static struct ipq *ip_create(struct sk_buff *skb, struct iphdr *iph, struct
device *dev)
621 {
```

当IP协议模块检查到接收到了一个分片数据包,而且尚无对应的ipq结构,则调用ip_create函数创建一个ipq结构用于此后的分片数据包缓存。参数skb表示接收到的分片数据包,iph表示分片数据包的IP首部,dev表示接收该分片数据包的网络设备。

```
622     struct ipq *qp;
623     int maclen;
624     int ihlen;
```

```
625     qp = (struct ipq *) kmalloc(sizeof(struct ipq), GFP_ATOMIC);
626     if (qp == NULL)
627     {
628         printk("IP: create: no memory left !\n");
629         return(NULL);
630         skb->dev = qp->dev;
631     }
632     memset(qp, 0, sizeof(struct ipq));
```

625-632行代码分配一个新的ipq结构，并对该结构进行清零操作。下面将是对该ipq结构各字段的初始化。

```
633     /*
634      *  Allocate memory for the MAC header.
635      *
636      *  FIXME: We have a maximum MAC address size limit and define
637      *  elsewhere. We should use it here and avoid the 3 kmalloc() calls
638      */

639     maclen = ((unsigned long) iph) - ((unsigned long) skb->data);
640     qp->mac = (unsigned char *) kmalloc(maclen, GFP_ATOMIC);
641     if (qp->mac == NULL)
642     {
643         printk("IP: create: no memory left !\n");
644         kfree_s(qp, sizeof(struct ipq));
645         return(NULL);
646     }

647     /*
648      *  Allocate memory for the IP header (plus 8 octets for ICMP).
649      */

650     ihlen = (iph->ihl * sizeof(unsigned long));
651     qp->iph = (struct iphdr *) kmalloc(ihlen + 8, GFP_ATOMIC);
652     if (qp->iph == NULL)
653     {
654         printk("IP: create: no memory left !\n");
655         kfree_s(qp->mac, maclen);
656         kfree_s(qp, sizeof(struct ipq));
657         return(NULL);
658     }

659     /* Fill in the structure. */
660     memcpy(qp->mac, skb->data, maclen);
```

```
661     memcpy(qp->iph, iph, ihlen + 8);
```

ipq结构中mac，iph字段分别指向MAC，IP首部，639-661行代码即是对这两个字段的初始化。注意这两个字段指向的内存空间是通过kmalloc函数调用分配的一块新的缓冲区，而非直接指向分片数据包对应sk_buff结构的相应字段，所以在进行ipq结构释放时，需要另外对这两个字段指向的内存空间进行释放，这一点可以从ip_free函数实现代码看出。

```
662     qp->len = 0;
663     qp->ihlen = ihlen;
664     qp->maclen = maclen;
665     qp->fragments = NULL;
666     qp->dev = dev;

667     /* Start a timer for this entry. */
668     qp->timer.expires = IP_FRAG_TIME;       /* about 30 seconds */
669     qp->timer.data = (unsigned long) qp;        /* pointer to queue */
670     qp->timer.function = ip_expire;         /* expire function  */
671     add_timer(&qp->timer);
```

667-671行代码即是对定时器的设置，超时执行函数设置为ip_expire，ip_expire函数执行时的参数为对应的ipq结构。注意定时间隔为30秒（IP_FRAG_TIME=30*HZ）.

```
672     /* Add this entry to the queue. */
673     qp->prev = NULL;
674     cli();
675     qp->next = ipqueue;
676     if (qp->next != NULL)
677         qp->next->prev = qp;
678     ipqueue = qp;
679     sti();
680     return(qp);
681 }
```

ip_create函数最后完成的工作就是将这个新创建的ipq结构插入到ipqueue变量指向的ipq结构类型队列中，这个插入的顺序是无关紧要的，实现上将每个新创建的ipq结构都插入到队列首部。函数最后返回这个新创建的ipq结构。

```
682 /*
683  *  See if a fragment queue is complete.
684  */

685 static int ip_done(struct ipq *qp)
686 {
687     struct ipfrag *fp;
```

```
688     int offset;

689     /* Only possible if we received the final fragment. */
690     if (qp->len == 0)
691         return(0);

692     /* Check all fragment offsets to see if they connect. */
693     fp = qp->fragments;
694     offset = 0;
695     while (fp != NULL)
696     {
697         if (fp->offset > offset)
698             return(0);  /* fragment(s) missing */
699         offset = fp->end;
700         fp = fp->next;
701     }

702     /* All fragments are present. */
703     return(1);
704 }
```

ip_done函数用于检查是否所以分片都已到达，当接收到数据序列排列上最后一个分片时，根据该分片偏移量字段以及本身的数据长度，我们可以计算出原始数据包数据长度，此时将对ipq结构中len字段进行初始化，初始化为原始大数据包中数据长度，否则，该字段将为0。690行检查该字段值，如果为0，表示至少最后分片尚未到达，所以返回0，表示还有分片尚未到达。如果len字段非0，则表示序列号排序上地处最后的一个分片已经到达，但是可能由于乱序到达情况的发生，中间一个，甚至第一个分片有可能尚未到达，695-701行代码对这些情况进行检查,如果发现数据不连续的情况,则表示尚有分片没有到达。注意fp表示ipfrag结构，每个ipfrag结构对应一个分片数据包，ipfrag结构中offset字段表示对应分片数据包中所包含数据在原始数据包中的偏移量，而end字段表示分片数据包中最后一个数据的偏移量加1,697行即检查下一个分片中第一个数据偏移量是否大于前一个分片中最后一个数据偏移量加1，如果是，则表示出现断裂，尚有其他分片没有到达，返回0，继续等待，否则返回1，表示其他模块可以进行数据包重组操作了。有关ipfrag结构中各字段的设置情况可阅读下文中对ip_defrag函数的分析，然后结合此处及前文中分析，可将所以这些串联起来，从而真正理解其中的原理。

```
705 /*
706  *  Build a new IP datagram from all its fragments.
707  *
708  *  FIXME: We copy here because we lack an effective way of handling lists
709  *  of bits on input. Until the new skb data handling is in I'm not going
710  *  to touch this with a bargepole. This also causes a 4Kish limit on
711  *  packet sizes.
712  */
```

```
713 static struct sk_buff *ip_glue(struct ipq *qp)
714 {
715     struct sk_buff *skb;
716     struct iphdr *iph;
717     struct ipfrag *fp;
718     unsigned char *ptr;
719     int count, len;

720     /*
721      *  Allocate a new buffer for the datagram.
722      */

723     len = qp->maclen + qp->ihlen + qp->len;
```

计算原始大数据包的长度，这个长度包括MAC，IP以及IP负载长度。ipq结构中len字段值即表示IP数据负载长度。

```
724     if ((skb = alloc_skb(len,GFP_ATOMIC)) == NULL)
725     {
726         ip_statistics.IpReasmFails++;
727         printk("IP: queue_glue: no memory for gluing queue 0x%X\n", (int) qp);
728         ip_free(qp);
729         return(NULL);
730     }
```

724-730行代码分配一个新的sk_buff结构用于对合成后的原始数据包进行封装。

```
731     /* Fill in the basic details. */
732     skb->len = (len - qp->maclen);
733     skb->h.raw = skb->data;
734     skb->free = 1;
```

因为重组原始数据包后，需要传递给IP协议其他模块进行处理，所以此处将sk_buff结构中len字段设置为IP首部及其负载长度。另外将union结构类型的h字段之raw子字段设置为指向sk_buff结构中数据缓冲区起始处，下面（736行）即开始对数据缓冲区进行数据复制。

```
735     /* Copy the original MAC and IP headers into the new buffer. */
736     ptr = (unsigned char *) skb->h.raw;
737     memcpy(ptr, ((unsigned char *) qp->mac), qp->maclen);
738     ptr += qp->maclen;
739     memcpy(ptr, ((unsigned char *) qp->iph), qp->ihlen);
740     ptr += qp->ihlen;
741     skb->h.raw += qp->maclen;
```

736-741对MAC，IP首部进行复制，并更新sk_buff结构中相关字段值（741行将h字段指向为IP首部，这个h字段是随着数据包所在层次变化的，当处于网络层，这个字段指向IP首部，当处于传输层时，这个字段指向传输层协议首部，当然，开始时，是直接指向MAC首部的）。注意740行将ptr指针指向了第一个IP数据负载所处的地址，下面的IP数据负载的复制将紧接着IP首部开始。

```
742     count = 0;
```

count变量表示当前复制的数据长度。

```
743     /* Copy the data portions of all fragments into the new buffer. */
744     fp = qp->fragments;
745     while(fp != NULL)
746     {
747         if(count+fp->len > skb->len)
748         {
749             printk("Invalid fragment list: Fragment over size.\n");
750             ip_free(qp);
751             kfree_skb(skb,FREE_WRITE);
752             ip_statistics.IpReasmFails++;
753             return NULL;
754         }
755         memcpy((ptr + fp->offset), fp->ptr, fp->len);
756         count += fp->len;
757         fp = fp->next;
758     }
```

744-758行代码完成数据的复制，这是通过对每个分片数据包中所包含数据进行单独复制完成的。注意此处还处理了可能的分片数据包之间数据重叠的情况。755行中对memcpy函数调用中参数设置方式解决了寻找目的缓冲区地址和可能出现的分片数据包乱序存放的异常情况。

```
759     /* We glued together all fragments, so remove the queue entry. */
760     ip_free(qp);
```

在完成对所有分片数据包中的数据提取，可以对这些分片以及相关表示结构进行释放了，760行对ip_free函数的调用即完成此项工作。

```
761     /* Done with all fragments. Fixup the new IP header. */
762     iph = skb->h.iph;
763     iph->frag_off = 0;
764     iph->tot_len = htons((iph->ihl * sizeof(unsigned long)) + count);
765     skb->ip_hdr = iph;
```

762-764行代码对重组后数据包中IP首部中相关字段进行设置，如偏移量字段需要设置为0，总长度字段设置为IP首部加上IP数据负载的长度。765行对sk_buff结构中ip_hdr字段进行初始化，使其指向IP首部。

```
766     ip_statistics.IpReasmOKs++;
767     return(skb);
768 }
```

如果调用ip_done函数返回1，则表示所有分片都已到达，此时ip_glue函数将被调用对这些分片进行重组，从而还原出对方发送的原始大数据包。对于流式传输协议（如TCP协议），由于数据之间是连续的，没有包的界限限制，所以一般在传输层（即TCP协议实现模块）就会对发往网络层的数据包大小进行限制,以免在网络层进行数据包分片以及给通信对端添加重组工作；但对于面向报文的协议（如UDP协议），是有包的界限限制的，用户单次写入传输的数据，不可以在UDP协议实现模块中预先分为多个包传送，因为网络层对于上层每次传递下来的数据包都会分配不同的标识符，如果UDP协议采用TCP协议相同的方式，也进行预先数据分散工作，那么实际上接收端看到的就直接是多个数据包，而不是多个分片了，这就完全不同了。所以对于TCP协议而言，可能不会使用到网络层协议（IP协议）提供的数据包分片和重组工作，但对于UDP协议而言，如果数据长度大于MTU，都要使用数据包分片以及接收端分片重组功能。ip_glue函数将所有分片数据包合成为一个大的数据包。具体分析参见代码中分析。

IP协议总入口函数为ip_rcv,当从链路层模块接收到一个数据包时，首先由ip_rcv函数进行处理,该函数根据数据包中IP首部相关字段只判断该数据包是发往本机的还是需要通过本机进行转发。对于发往本机的数据包，还要检查其是否为一个分片数据包，从而在网络层进行数据包重组后，才将重组后原始数据包送往传输层进行处理。所以ip_rcv函数之于IP协议的作用如同tcp_rcv之于TCP协议，udp_rcv之于UDP协议等。而下面要介绍的ip_defrag函数是对分片数据包进行处理的总入口函数,当ip_rcv检查到一个分片数据包后,就调用ip_defrag函数进行处理，ip_defrag函数将缓存该分片数据包，并检查是否所有数据包都已到达，如果都已到达，ip_defrag函数调用ip_glue函数进行分片数据包重组工作。ip_rcv函数将根据ip_defrag函数返回是否为NULL来决定是否继续进行处理。为更好的理解ip_defrag函数的作用，下面给出ip_rcv函数中对ip_defrag的调用环境。

```
1244    int ip_rcv(struct sk_buff *skb, struct device *dev, struct packet_type *pt)
1245    {

            ......


1306        /*
1307         *  Remember if the frame is fragmented.
1308         */
1309        if(iph->frag_off)
1310        {
1311            if (iph->frag_off & 0x0020)
```

```
1312                is_frag|=1;
1313            /*
1314             *  Last fragment ?
1315             */
1316            if (ntohs(iph->frag_off) & 0x1fff)
1317                is_frag|=2;
1318        }

            ......
1384        /*
1385         * Reassemble IP fragments.
1386         */
1387        if(is_frag)
1388        {
1389            /* Defragment. Obtain the complete packet if there is one */
1390            skb=ip_defrag(iph,skb,dev);
1391            if(skb==NULL)
1392                return 0;
1393            skb->dev = dev;
1394            iph=skb->h.iph;
1395        }

            ......
1479        return(0);
1480    }
```

ip_rcv函数中1309-1318对数据包是否为一个分片数据包进行检查，非分片数据包IP首部中MF字段为0，偏移字段为0。分片数据包满足如下条件：

1〉 第一个分片数据包：MF=1，offset=0

2〉 中间分片数据包：MF=1，offset！=0

3〉 最后一个分片数据包：MF=0，offset！=0

所以1309行将偏移量非0作为判断条件并没有对第一个分片进行检查，从1310-1318行代码来看，此处实际上只是对中间分片和最后分片进行了检查。所以可以说，ip_rcv函数实现中对于分片数据包的判断存在问题，或者我们可以将之看作是一个系统BUG，因为缺少了对第一个分片数据包的判断，如此第一个分片数据包将被作为普通数据包传送给上层传输层进行处理，这是不对的。

1387-1395是对一个分片数据包的处理，如果判断出是一个分片数据包，则调用ip_defrag函数进行处理，如果该函数返回NULL，就表示分片数据包尚未到达完整，还要等待其他分片的进一步到达，此时已经完成对该分片数据包的处理，无需进行下面将数据包传送给传输层的处理，所以1392行直接返回。否则还需要将重组后的数据包进一步传递给上层传输层进行处理。我们看ip_rcv对ip_defrag函数的调用参数设置，第一个参数为分片数据包的IP首部，第二个参数为被封装为sk_buff结构的分片数据包本身，第三个参数为接收分片数据包的网

络设备。ip_defrag函数如下，我们将一步步阐述该函数的代码实现，首先再次说明一下该函数的完成的功能，从而可以从大的方面进行把握，便于下文的理解。ip_defrag函数负责处理分片数据包，如果接收当前分片后，所有分片均已到达，则调用ip_glue函数对所有分片进行重组，ip_defrag函数返回这个重组后的数据包（即返回给ip_rcv函数）；否则将分片数据包缓存到ipq结构fragments字段指向的队列中，当然这首先需要完成相关数据结构的分配。具体情况见如下对ip_defrag函数的分析。

```
769 /*
770  *  Process an incoming IP datagram fragment.
771  */

772 static struct sk_buff *ip_defrag(struct iphdr *iph, struct sk_buff *skb,
                  struct device *dev)
773 {
```

ip_defrag函数的参数含义已经在前文中进行了说明，iph表示分片数据包IP首部，skb表示分片数据包本身，dev表示接收这个分片数据包的网络设备。

```
774     struct ipfrag *prev, *next;
775     struct ipfrag *tfp;
776     struct ipq *qp;
777     struct sk_buff *skb2;
778     unsigned char *ptr;
779     int flags, offset;
780     int i, ihl, end;

781     ip_statistics.IpReasmReqds++;

782     /* Find the entry of this IP datagram in the "incomplete datagrams" queue.
*/
783     qp = ip_find(iph);
```

ip_find函数通过比较IP首部中相关字段（标识符；源，目的IP地址；传输层使用协议）查找分片缓存队列（由ipq结构表示），ip_find函数返回值将在下文用于判断是否需要生成一个新的队列（如果当前分片数据包是第一个被接收的数据包，则应无对应的分片缓存队列，此时就需要创建一个新的分片缓存队列）。

```
784     /* Is this a non-fragmented datagram? */
785     offset = ntohs(iph->frag_off);
786     flags = offset & ~IP_OFFSET;
787     offset &= IP_OFFSET;
```

784-787行代码分离标志位字段和分片偏移字段，IP_OFFSET定义在net/inet/ip.h中，为

0x1FFF，我们知道偏移字段占据低13比特，标志位字段占据高3位比特，所以这段代码正是将标志位字段和偏移字段进行分离，便于下文的处理和判断。

```
788     if (((flags & IP_MF) == 0) && (offset == 0))
789     {
790         if (qp != NULL)
791             ip_free(qp);    /* Huh? How could this exist?? */
792         return(skb);
793     }
```

788-793行代码是对当前被处理数据包是否为一个分片数据包的判断，如果788行为真，则表示这是一个普通数据包，如果此时对应有分片缓存队列，就表示出现了异常，对应的处理方式是释放分片队列中所有分片并释放相关辅助结构，并直接返回这个普通数据包（792行）。实际上我们从ip_defrag的调用环境来看，ip_rcv函数只对分片数据包调用ip_defrag函数进行处理，所以此处的检查没有多大的必要。

```
794     offset <<= 3;       /* offset is in 8-byte chunks */
```

在前文中讨论IP首部中有关分片处理的字段时，我们已经说到因为偏移量字段只有13比特，比总长度字段（16比特）少三个比特，所以偏移量字段实际上以8字节单位，这也就要求除去最后一个分片外，其他所有分片中包含的数据长度必须是8的倍数。这一点我们在ip_fragment对数据包进行分片处理函数中可以看到这一点。794行代码通过乘以8来获得真正的偏移量。

```
795     /*
796      * If the queue already existed, keep restarting its timer as long
797      * as we still are receiving fragments.  Otherwise, create a fresh
798      * queue entry.
799      */

800     if (qp != NULL)
801     {
802         del_timer(&qp->timer);
803         qp->timer.expires = IP_FRAG_TIME;   /* about 30 seconds */
804         qp->timer.data = (unsigned long) qp;    /* pointer to queue */
805         qp->timer.function = ip_expire;     /* expire function */
806         add_timer(&qp->timer);
807     }
```

如果ip_find函数调用返回非NULL，则表示已经存在一个分片缓存队列（即表示当前处理分片不是第一个到达的分片），此时需要重置超时定时器，超时定时器的作用在前文分析中已经进行了较为详细的分析，此处不再讨论。

```
808     else
809     {
810         /*
811          *  If we failed to create it, then discard the frame
812          */
813         if ((qp = ip_create(skb, iph, dev)) == NULL)
814         {
815             skb->sk = NULL;
816             kfree_skb(skb, FREE_READ);
817             ip_statistics.IpReasmFails++;
818             return NULL;
819         }
820     }
```

808-820行代码对应当前处理分片数据包是第一个到达的分片（注意第一个到达的分片并不一定是第一个分片，读者需要明白二者的不同），我们需要创建一个分片缓存队列，即创建一个ipq结构，这是通过调用ip_create函数完成的。ip_create函数在前文中已经进行了分析。代码执行到此处，我们已经有了缓存这个分片的队列了，下面就是创建一个表示这个分片的ipfrag结构了，从而将这个分片缓存到分片队列中。

```
821     /*
822      *  Determine the position of this fragment.
823      */

824     ihl = (iph->ihl * sizeof(unsigned long));
825     end = offset + ntohs(iph->tot_len) - ihl;
```

824行计算IP首部长度，825行计算此分片数据包中包含数据中最后一个数据的偏移量加1。注意end字段是分片数据包中最后一个字节数据偏移量加1，这在ip_done函数中进行分片数据包连续性检查中会使用到这一点。我们可以用一个简单的例子说明加1的含义，offset变量表示分片数据包中第一个数据的偏移量，我们不妨假设为100，分片数据包中数据长度假设为200，则最后一个数据字节的偏移量将为299（注意不是300），即该分片数据包中包含的数据偏移量范围为100-299,共200个字节。而根据这些假设,825行计算出end变量将为300，这就是加1的所在。

```
826     /*
827      *  Point into the IP datagram 'data' part.
828      */

829     ptr = skb->data + dev->hard_header_len + ihl;
```

829行对ptr变量进行初始化，该变量指向实际的数据区，即传输层首部（如果是第一个分片的话）及其数据负载。

```
830     /*
831      *  Is this the final fragment?
832      */

833     if ((flags & IP_MF) == 0)
834         qp->len = end;
```

如果MF标志位为0，则表示这是最后一个分片，那么此时可以得到原始被分片的数据包长度，因为偏移量是从0计算起的，所以最后一个分片中最后一个数据偏移量加1就表示原始数据包中数据总长度。注意如果ipq结构中len字段被初始化为非0值，就表示最后一个分片到达，当然由于可能不同的路由途径，还有其他分片尚未到达。

```
835     /*
836      *  Find out which fragments are in front and at the back of us
837      *  in the chain of fragments so far.  We must know where to put
838      *  this fragment, right?
839      */

840     prev = NULL;
841     for(next = qp->fragments; next != NULL; next = next->next)
842     {
843         if (next->offset > offset)
844             break;  /* bingo! */
845         prev = next;
846     }
```

840-846行代码在分片缓存队列中寻找合适的插入位置，只需要注意到分片缓存队列中各分片是按照各分片偏移量大小从小到大进行排列的。当跳出841行循环时，prev指向插入位置的前一个分片，而next指向插入位置的后一个分片。

```
847     /*
848      *  We found where to put this one.
849      *  Check for overlap with preceding fragment, and, if needed,
850      *  align things so that any overlaps are eliminated.
851      */
852     if (prev != NULL && offset < prev->end)
853     {
```

如果前一个分片有效，而且当前分片偏移量在前一个分片所表示数据偏移量之内，就表示各分片之间发生了数据重叠，下面需要对这种重叠情况进行处理，处理方式上是适当调整当前分片的偏移量，从而取消重叠。具体代码如854-856行所示。

```
854         i = prev->end - offset;
855         offset += i;    /* ptr into datagram */
```

```
856         ptr += i;   /* ptr into fragment data */
857     }
```

854行代码计算重叠数据量的大小，855，856行更新偏移量值，以及同时更新ptr指针变量值，这个变量指向实际有效数据，在上文中829行ptr变量已经被初始化为指向分片数据包中包含的数据，提出（856行）即跳过重叠数据区域。这个ptr变量值将在创建ipfrag结构时赋值给ipfrag结构中对应ptr字段，前文中介绍ipfrag结构时，我们说到该结构ptr字段并非一定指向其对应分片数据包中包含的数据起始处，可能存在一个偏移，此处代码正好说明了这一点。前面处理的是当前分片数据包与其插入位置之前分片数据重叠的情况，下面将要检查其与插入位置之后的分片可能的数据重叠。

```
858     /*
859      * Look for overlap with succeeding segments.
860      * If we can merge fragments, do it.
861      */

862     for(; next != NULL; next = tfp)
863     {
864         tfp = next->next;
865         if (next->offset >= end)
866             break;      /* no overlaps at all */
```

注意end变量在825行初始化为当前处理分片数据包中最后一个数据字节偏移量加1。如果下一个分片数据包中第一个数据字节的偏移量大于或者等于该end变量，则表示不存在数据重叠问题，由于分片缓存队列中分片数据包是按偏移量大小从小到大进行排列的，所以无需对后续分片进行检查，直接跳出检查即可。如果865行判断条件不满足，则表示当前处理分片与已缓存分片发生了数据重叠，那么就需要对这些已缓存分片进行调整。注意调整时所依据的方式是只动分片中起始偏移量，即指从前面进行压缩，不从后面进行压缩。采用这种统一的方式一方面便于处理，另一方面也不容易产生混淆。

```
867         i = end - next->offset;         /* overlap is 'i' bytes */
868         next->len -= i;             /* so reduce size of    */
869         next->offset += i;          /* next fragment    */
870         next->ptr += i;
```

867-870行代码完成的工作如同854-856行代码，868行对已缓存分片数据包的有效数据长度进行更新。这个更新在如下875行if语句中用到。

```
871         /*
872          *  If we get a frag size of <= 0, remove it and the packet
873          *  that it goes with.
874          */
875         if (next->len <= 0)
876         {
```

如果更新后有效数据长度变为0或者负数，则表示这个已缓存分片中所有数据都与当前处理分片发生了重叠，此时就没有必要缓存该分片，需要将其从缓存队列中删除。下面即完成此项工作。

```
877             if (next->prev != NULL)
878                 next->prev->next = next->next;
879             else
880                 qp->fragments = next->next;
```

877-880行代码对数据被完全覆盖的已缓存分片进行删除操作。如果ipfrag结构prev字段为NULL，则表示这个分片是缓存队列中第一个分片，此时直接调整ipq结构的fragments字段即可，否则按照878行的一般方式进行删除。

```
881             if (tfp->next != NULL)
882                 next->next->prev = next->prev;
```

881-882是对被删除分片之后的下一个分片中prev字段进行更新。864行每次循环开始处ftp变量被初始化为指向当前被检查分片的下一个分片，所以881行的检查存在问题，正确代码应该如下所示：

```
                if (tfp != NULL)
```

882行next->next就是tfp，所以tfp->prev就是对当前被删除分片的下一个分片的prev字段进行更新，更新为指向当前被删除分片的前一个分片（next->prev）.

```
883             kfree_skb(next->skb,FREE_READ);
884             kfree_s(next, sizeof(struct ipfrag));
```

883行释放被删除分片对应的sk_buff封装结构，注意由于sk_buff结构封装了数据，所以同时也实际删除了对应的数据包，884行对ipfrag表示结构进行删除。

```
885         }
886     }//for
```

经过以上对重叠数据的检查和处理,现在我们终于可以将当前接收到的这个分片数据包插入到分片缓存队列中去了。

```
887     /*
888      *  Insert this fragment in the chain of fragments.
889      */

890     tfp = NULL;
891     tfp = ip_frag_create(offset, end, skb, ptr);
```

首先调用ip_frag_create函数创建一个ipfrag结构表示这个分片，注意ptr参数的传入。

```
892     /*
893      *  No memory to save the fragment - so throw the lot
894      */

895     if (!tfp)
896     {
897         skb->sk = NULL;
898         kfree_skb(skb, FREE_READ);
899         return NULL;
900     }
```

895行判断内存分配是否成功，如果失败，则简单丢弃当前接收的分片数据包。否则将表示这个分片数据包的ipfrag结构插入到分片缓存队列中。由于prev，next变量分别表示插入位置的前一个和后一个分片，所以插入操作变得非常方便，具体如下901-908行代码。

```
901     tfp->prev = prev;
902     tfp->next = next;
903     if (prev != NULL)
904         prev->next = tfp;
905     else
906         qp->fragments = tfp;

907     if (next != NULL)
908         next->prev = tfp;

909     /*
910      *  OK, so we inserted this new fragment into the chain.
911      *  Check if we now have a full IP datagram which we can
912      *  bump up to the IP layer...
913      */

914     if (ip_done(qp))
915     {
916         skb2 = ip_glue(qp);     /* glue together the fragments */
917         return(skb2);
918     }
919     return(NULL);
920 }
```

ip_defrag函数对当前接收分片数据包完成缓存操作后，检查到目前为止，是否已经接收到所有的分片数据包，即是否可以进行分片重组工作。具体的，是通过调用ip_done(914行)函数检查分片数据包的到达情况，如果全部到达，则调用ip_glue(916行)函数完成分片重组工作，并返回这个重组得到的原始数据包，否则返回NULL，表示当前接收分片数据包已经完成处理，但尚有其他数据包没有到达，无法完成分片重组工作，还需等待。

完成了对分片重组的分析，下面我们进行对大数据包进行分片函数的介绍。如果传输层传递给IP模块的数据包长度大于最大传输单元大小，则IP模块需要对这个大的数据包进行分片后传送。分片所依据的准则是：其一除了最后一个分片外，其他所有分片包含数据长度必须是8的倍数；其二只需要在第一个分片中包含原始数据包中传输层首部，其他分片数据包中IP首部之后直接跟随用户数据（注意对应每个分片数据包都要创建一个新的IP首部）。

IP协议实现模块中，ip_fragment函数负责对大的数据包进行分片，在下面具体介绍ip_fragment函数之前，我们首先查看该函数被调用环境，便于我们从总体上理解ip_fragment函数。当前内核代码中涉及到对ip_fragment函数的调用共有两处：其一是在ip_forward函数对一个接收的目的地址非本地的数据包进行转发时；其二是在调用ip_queue_xmit函数完成一个本地创建数据包的发送时。这两处对ip_fragment函数的调用环境基本相同，如下代码所示：

```
    if(skb->len > dev->mtu + dev->hard_header_len)
    {
        ip_fragment(sk,skb,dev,0); //ip_queue_xmit
        /*ip_fragment(sk,skb,dev,is_frag); //ip_forward */
        IS_SKB(skb);
        kfree_skb(skb,FREE_WRITE);
        return;
    }
```

这个if条件语句判断是否需要进行数据包分片，skb->len表示数据包的长度，包括MAC首部长度和IP首部及其负载长度。MTU最大报文长度表示的仅仅是IP首部及其数据负载的长度，不包含MAC首部长度，所以在进行长度比较时，需要加上MAC首部长度的值。ip_fragment函数最后一个参数表示被分片数据包是否本身就是一个分片，这种情况发生在数据包需要经过不同传输介质或者协议传输时，例如本地发送一个大的数据包，则第一次在本地就会发生一次分片，对于以太网而言，每个分片大小是1500+14字节，如果到达最终目的端之前，需要经过一段串行线传输（MTU=296），则又需要对分片本身进行分片。当然对分片进行分片只可能发生了转发过程中，对于本地产生的数据包，不会产生这种情况，故而对于ip_fragment函数调用时对于最后一个参数的设置，ip_forward和ip_queue_xmit函数稍有不同。那么这两种情况为何需要区分对待？原因是如果是对一个分片本身进行再分片，则新产生的第一个分片中偏移字段值应该来自原分片偏移字段，而不是另一种情况下的0。下面我们即对ip_fragment函数进行具体分析。

```
921 /*
922  *  This IP datagram is too large to be sent in one piece.  Break it up into
923  *  smaller pieces (each of size equal to the MAC header plus IP header plus
924  *  a block of the data of the original IP data part) that will yet fit in a
925  *  single device frame, and queue such a frame for sending by calling the
926  *  ip_queue_xmit().  Note that this is recursion, and bad things will happen
927  *  if this function causes a loop...
928  *
```

```
929  *  Yes this is inefficient, feel free to submit a quicker one.
930  *
931  *  **Protocol Violation**
932  *  We copy all the options to each fragment. !FIXME!
933  */
934 void ip_fragment(struct sock *sk, struct sk_buff *skb,
                              struct device *dev, int is_frag)
935 {
```

虽然前文中对于最后一个参数已经进行了简单说明，此处再次强调一下：如果是本地发送的数据包，则is_frag参数被设置为0；如果是转发的数据包，则is_frag参数表示被转发数据包是否本身就是一个分片，根据ip_fragment在此种情况下的调用环境，此时被ip_forward函数调用,其调用ip_fragment函数时使用的is_frag参数直接来自于ip_rcv函数中的判断和设置，具体如下：
1〉 if(is_frag&1)为真，则表示这个被分片数据包本身是一个分片，而且是一个中间分片。
2〉 if(is_frag&2)为真，则表示这个被分片数据包本身是一个分片，而且是最后一个分片。

注意由于ip_rcv函数中对于分片数据包判断的不充分，此处缺少对第一个分片情况的判断，这个系统BUG影响到了好几个函数（ip_defrag,ip_forward, ip_fragment等）。在下文的分析中，我们可以看到ip_fragment函数本身也存在一个BUG!

```
936     struct iphdr *iph;
937     unsigned char *raw;
938     unsigned char *ptr;
939     struct sk_buff *skb2;
940     int left, mtu, hlen, len;
941     int offset;
942     unsigned long flags;

943     /*
944      *  Point into the IP datagram header.
945      */

946     raw = skb->data;
947     iph = (struct iphdr *) (raw + dev->hard_header_len);

948     skb->ip_hdr = iph;

949     /*
950      *  Setup starting values.
951      */

952     hlen = (iph->ihl * sizeof(unsigned long));
953     left = ntohs(iph->tot_len) - hlen;  /* Space per frame */
```

```
954     hlen += dev->hard_header_len;       /* Total header size */
955     mtu = (dev->mtu - hlen);        /* Size of data space */
956     ptr = (raw + hlen);         /* Where to start from */
```

946-956行代码对一些变量进行初始化，为下文的处理做好准备。具体的，raw变量指向缓冲区首部（同时指向MAC首部）；iph变量指向IP首部；hlen变量被初始化为MAC，IP首部的长度；mtu变量初始化为用户数据长度（不过此处的初始化有问题，事实上，本版本网络实现中对于MTU，MSS值的处理上都存在问题！）；ptr变量指向IP负载数据开始处。

```
957     /*
958      *  Check for any "DF" flag. [DF means do not fragment]
959      */

960     if (ntohs(iph->frag_off) & IP_DF)
961     {
962         /*
963          *  Reply giving the MTU of the failed hop.
964          */
965         ip_statistics.IpFragFails++;
966         icmp_send(skb,ICMP_DEST_UNREACH, ICMP_FRAG_NEEDED, dev->mtu, dev);
967         return;
968     }
```

因为需要马上进行分片处理，所以需要在此之前检查发送端是否允许进行分片，如果IP首部中DF（Don’t Fragment）标志位被设置，则表示发送端不允许进行分片操作。此时返回一个ICMP错误，错误类型为ICMP_FRAG_NEEDED.

```
969     /*
970      *  The protocol doesn't seem to say what to do in the case that the
971      *  frame + options doesn't fit the mtu. As it used to fall down dead
972      *  in this case we were fortunate it didn't happen
973      */

974     if(mtu<8)
975     {
```

变量mtu在955行被初始化为IP数据负载长度，由于分片中数据长度必须是8的倍数，如果mtu小于8，将无法为其创建分片。当然mtu小于8在当前网络中是不会发生的，所以此处的检查有些多余，除非网络设备（对MTU值）的配置存在问题！

```
976         /* It's wrong but it's better than nothing */
977         icmp_send(skb,ICMP_DEST_UNREACH,ICMP_FRAG_NEEDED,dev->mtu, dev);
978         ip_statistics.IpFragFails++;
979         return;
```

```
980     }

981     /*
982      *  Fragment the datagram.
983      */

984     /*
985      *  The initial offset is 0 for a complete frame. When
986      *  fragmenting fragments it's wherever this one starts.
987      */

988     if (is_frag & 2)
989         offset = (ntohs(iph->frag_off) & 0x1fff) << 3;
990     else
991         offset = 0;
```

上文中我们已经说明，ip_fragment调用环境有两个，一是被ip_forward调用在转发数据包过程中进行分片；二是本地发送数据包的分片。对于第一种转发的情况，is_frag参数将表示被转发数据包的本身的分片情况，如果is_frag非0，则表示被转发的数据包本身就是一个分片，现在是对这个分片进行再分片。再分片中需要注意的地方是第一个分片中的IP首部中偏移字段值就不再是0，而是来自原分片的偏移字段值。988-991完成的工作即是如此。通过对is_frag标志位的检查，判断此次分片是否针对一个本身就是一个分片的数据包。从ip_fragment在ip_forward的调用环境来看，is_frag表示的含义如下：
1〉如果is_frag&1非0，表示现在在对一个中间分片进行再分片。
2〉如果is_frag&2非0，表示现在是在对原来的最后一个分片进行再分片。
所以988行判断语句设置的不正确，正确的检查方式应该如下：

```
988     if (is_frag & 3)
```

正如前文中所述，对于分片的判断上，在ip_rcv函数中也存在错误，从而导致ip_defrag等其他相关函数也都有类似错误！

```
992     /*
993      *  Keep copying data until we run out.
994      */
```

下面的这个while循环管到函数结尾，对原始数据包进行循环分片。此处需要提前交待的是，进行分片时：
1〉除了最后一个分片外，其他所有分片大小都是MTU大小，即每个分片尽量携带最多数据量。
2〉除了最后一个分片外，其他所有分片大小必须是8的倍数，如果MTU大小本身是8的倍数，那么就是如1〉中所述为MTU大小，否则必须截断为小于MTU值最大的一个数，这个数为8的倍数。
以上两个条件如果读者认为矛盾，那是在下没有说清楚。二者所依据的基本思想是：每个分片尽量携带最多数据，当然这个最多数据不能大于MTU值，因为这正是分片的目的；其次，每个分片除了最后一个分片外，其所包含的数据量必须是8的倍数，这是由IP首部中偏移量

字段为13比特，而总长度字段是16比特造成的。
依据这两个基本思想，下面的代码就很容易理解。注意对于每个分片必须创建独立的MAC，IP首部。第一个分片IP首部后跟随原数据包中传输层协议首部，而其他分片IP首部后直接跟随数据，对于这一点原本无需交待，因为在进行分片时，是对IP数据负载进行分块，以上只是分块的自然结果。

```
995     while(left > 0)
996     {
997         len = left;
998         /* IF: it doesn't fit, use 'mtu' - the data space left */
999         if (len > mtu)
1000                len = mtu;
```

如果剩余长度仍然大于MTU，则表示还未到达最后一个分片，分片的长度设置为MTU大小，如果999行判断出len<=mtu,则表示这是最后一个分片了，那么就将剩余的数据长度（len）作为这个分片的最终长度，即没有第1000行代码的执行。

```
1001            /* IF: we are not sending upto and including the packet end
1002               then align the next start on an eight byte boundary */
1003            if (len < left)
1004            {
1005                len/=8;
1006                len*=8;
1007            }
```

在997行，len被初始化为left，如果999行不满足，那么就有len=left，如此1003行没有意义。要使1003行发生作用，那么必须1000行代码得到执行，即有len>mtu，也即left>mtu，换句话说，我们这是在创建第一个分片或者是一个中间分片，上文中我们一再阐述，除了最后一个分片外，其他所有分片长度必须是8的倍数，此处分片长度被设置为mtu（1000行）值，那么我们需要检查这个mtu值是否为8的倍数，1003-1007行代码实际上一方面完成了检查，另一方面将这个分片长度设置为小于mtu值的最大的一个为8的倍数的数值。注意len为整型变量，所以1005行计算结果（小数部分）会被截断，经过1006行乘法运算，现在len就被设置为小于mtu值的最大的一个为8的倍数的数值。这正是我们想要的结果。得到分片正确的长度后，下面所要完成的工作就比较简单了：创建一个新的分片数据包，将指定长度（len）的数据包从原数据包拷贝到该分片数据包中，对分片数据包创建IP，MAC首部，设置分片相关字段，调用ip_queue_xmit函数将分片数据包发往下层从而传送出去。此处IP首部的创建需要对各字段进行正确设置，而MAC首部直接复制原始数据包的MAC首部即可（当然如果使用802.3协议，还必须对MAC首部中长度字段进行正确设置，如果是以太网封装形式，则简单复制即可）。

```
1008            /*
1009             *  Allocate buffer.
1010             */
```

```
1011            if ((skb2 = alloc_skb(len + hlen,GFP_ATOMIC)) == NULL)
1012            {
1013                printk("IP: frag: no memory for new fragment!\n");
1014                ip_statistics.IpFragFails++;
1015                return;
1016            }

1017            /*
1018             *  Set up data on packet
1019             */

1020            skb2->arp = skb->arp;
```

sk_buff结构中arp字段表示MAC首部是否创建成功，这个字段在链路层发送数据包用于检查数据包是否满足发送条件，即目的MAC地址是否已经创建成功，如果arp字段为0，则表示目的MAC地址尚未完成创建，此时就需要首先发送一个ARP请求，获得目的主机MAC地址后方可进行数据包发送。由于分片数据包MAC首部中源，目的MAC地址均来自于原数据包，所以arp字段的设置当然也要和原数据包一致。

```
1021            if(skb->free==0)
1022                printk("IP fragmenter: BUG free!=1 in fragmenter\n");
```

此处对free字段的检查，其实在前文中已经进行了说明。由于本版本中对于数据包的重发缓存是在网络层进行的，传输层在将数据包传递给网络层之前必须正确free字段值，如果free字段为0，则表示网络层在发送数据包之前必须对其进行缓存，以便于日后的重发，对于TCP协议这样提供可靠性传输保证的协议而言，必须将free设置为0。而对于UDP协议而言，free字段就被设置为1，表示网络层只需将数据包发送出去即可，不需要再保存该数据包，因为上层根本不提供可靠的数据包传送服务。所以free字段的设置是是否提供可靠性传输保证的一个标志，或者说是区分面向流式服务和面向报文服务的一个标志。而在前文中我们说到，对于面向流的服务，在网络层已经对每个数据包的大小进行了合理划分，或者更直接的说，在传输层已经进行了分片，只不过对于流式传输方式而言，由于数据之间是连续的，接收端在读取数据时也是一个一个字节顺序读取的，采用流式传输方式到达的报文在接收端被“衔接”，即接收端看不到一个一个报文结构，看到的就是一连串数据。而面向报文的传输方式，接收端看到的就是一个一个独立的报文，各个报文之间不可以进行“衔接”，换句话说，每次读取不可以跨越报文边界，一次只可以读取一个报文中的内容，这一点与流式传输方式正好相反。我们说流式传输方式是不需要使用到网络层提供的分片功能的，事实上，网络层分片功能只用于面向报文的，且报文之间不存在互相联系的协议如UDP协议。对于这些协议free字段一定设置为1，即不提供可靠性传输保证。1021行检查的意义即在于此。下面1023行将新创建的分片数据包free字段设置为1所依据的思想也在于此！

```
1023            skb2->free = 1;
1024            skb2->len = len + hlen;
1025            skb2->h.raw=(char *) skb2->data;
1026            /*
```

```
1027            *  Charge the memory for the fragment to any owner
1028            *  it might possess
1029            */

1030           save_flags(flags);
1031           if (sk)
1032           {
1033               cli();
1034               sk->wmem_alloc += skb2->mem_len;
1035               skb2->sk=sk;
1036           }
```

sk变量表示原数据包所属的套接字，1031-1036对相关变量进行更新。这些代码读者应该可以自行理解了。

```
1037           restore_flags(flags);
1038           skb2->raddr = skb->raddr;    /* For rebuild_header - must be here
*/

1039           /*
1040            *  Copy the packet header into the new buffer.
1041            */

1042           memcpy(skb2->h.raw, raw, hlen);
```

变量hlen在954行被初始化为MAC，IP首部长度。此处将原数据包中这两个首部直接复制到新的分片数据包中，因为分片数据包MAC，IP首部大多数字段来自于原来的数据包，只有部分字段需要进行修改，这种复制方式可以大大节省首部创建的时间。

```
1043           /*
1044            *  Copy a block of the IP datagram.
1045            */
1046           memcpy(skb2->h.raw + hlen, ptr, len);
1047           left -= len;
```

1046-1047复制数据，并更新剩余数据量。注意ptr变量在956行被初始化为指向原数据包中IP数据负载起始处。每次创建一个分片进行数据复制后，都会对该变量进行更新（1060行）。

```
1048           skb2->h.raw+=dev->hard_header_len;
```

1048行直接前进到IP首部，换句话说，分片数据包中MAC首部原封不动的来自于原数据包。对于以太网MAC首部而言，这是正确的。但对使用802.3首部格式的数据包，这可能会造成问题，当然对于早期代码，我们不能要求太多！

```
1049            /*
1050             *  Fill in the new header fields.
1051             */
1052            iph = (struct iphdr *)(skb2->h.raw/*+dev->hard_header_len*/);
1053            iph->frag_off = htons((offset >> 3));
1054            /*
1055             *  Added AC : If we are fragmenting a fragment thats not the
1056             *         last fragment then keep MF on each bit
1057             */
1058            if (left > 0 || (is_frag & 1))
1059                iph->frag_off |= htons(IP_MF);
```

1052行设置iph变量为分片数据包首部起始处，1053行设置偏移字段值，注意偏移字段被初始化为偏移量除以8后的数值。1058-1059行设置MF标志位：其一除了最后分片外，其他所有分片都必须设置该标志位；其二如果被分片的数据包本身就是一个分片，那么由此产生的所有子分片中MF标志位都要被设置。1058行中left>0是对第一种情况的判断，is_frag&1是对第二种情况的判断。

```
1060            ptr += len;
1061            offset += len;

1062            /*
1063             *  Put this fragment into the sending queue.
1064             */

1065            ip_statistics.IpFragCreates++;

1066            ip_queue_xmit(sk, dev, skb2, 2);
1067        }
1068        ip_statistics.IpFragOKs++;
1069    }
```

1060-1061行对相关变量进行更新，最后调用ip_queue_xmit函数将新创建的分片数据包传递给链路层处理，从而最终将这个分片数据包发送出去。

自此我们完成对ip_fragment函数的分析，该函数相对其他函数代码较长，但实现思想还是比较简单，在分析ip_fragment函数时，只要紧紧抓住如下几点，将很容易理解ip_fragment函数：

1> 网络层数据包分片功能只用于面向报文的传输协议，不对流式传输协议使用。
2> 除了最后一个分片外，其他所有分片长度必须是8的倍数。
3> 每个分片应尽量传输最多数据，即除了最后一个分片外，其他分片长度应在满足8的倍数情况下尽量以MTU大小传输。
4> 如果被分片数据包本身是一个分片，那么需要正确处理子分片中IP首部偏移字段和MF标志位字段值。

5> 每个分片都需要创建新的MAC，IP首部。分片的自然结果是除了第一个分片IP首部跟随上层协议首部外，其他分片IP首部之后跟随的是纯数据。当然如果对一个分片本身进行再分片，那么同样的有类似结果，这种情况是在对原始数据包分片中将IP数据负载（传输层首部以及纯数据）当作普通数据简单处理的结果（即部队传输层首部和其他用户数据进行区分）。

数据包转发是一项非常重要功能，正是有了数据包转发，相隔千里之外的两台主机才可能相互通信，进行数据传送，一般，路由器都被配置为具有转发功能，而不同主机则不具备转发功能，由于同样的网络协议实现代码被使用在路由器和普通主机上，所以是否具有转发功能就以一个选项的形式存在。下面的介绍的ip_forward负责数据包的转发，该函数被嵌入在CONFIG_IP_FORWARD宏中，只有在CONFIG_IP_FORWARD宏被定义的情况下，网络实现方才包含这个函数，换句话说，即主机配置为具有数据转发功能。在具体介绍ip_forward函数之前，我们首先了解一下该函数被调用的环境，从而让我们可以更好理解ip_forward函数实现。ip_forward函数只在一处被调用，即在IP协议模块总入口函数ip_rcv函数中。具体被调用环境如下代码所示：

```
/*ip_rcv*/
1330    if(iph->daddr!= skb->dev->pa_addr && (brd=ip_chk_addr(iph->daddr))==0)
1331    {
1332        /*
1333         *  Don't forward multicast or broadcast frames.
1334        */

1335        if(skb->pkt_type!=PACKET_HOST || brd==IS_BROADCAST)
1336        {
1337            kfree_skb(skb,FREE_WRITE);
1338            return 0;
1339        }

1340        /*
1341         *  The packet is for another target. Forward the frame
1342        */

1343    #ifdef CONFIG_IP_FORWARD
1344        ip_forward(skb, dev, is_frag);
1345    #else
1346    /*  printk("Machine %lx tried to use us as a forwarder to %lx but
                                          we have forwarding disabled!\n",
1347                iph->saddr,iph->daddr);*/
1348        ip_statistics.IpInAddrErrors++;
1349    #endif
1350        /*
1351        *   The forwarder is inefficient and copies the packet. We
1352         *  free the original now.
```

```
1353        */

1354        kfree_skb(skb, FREE_WRITE);
1355        return(0);
1356    }
```

ip_rcv函数中1330行代码检查被接收数据包最终目的主机是否是本机，如果不是就检查目的IP地址是否为广播地址，如果不是，而且最终目的主机也不是本机，则可能需要进行数据包转发。注意对于目的IP地址是广播的地址，将不进行转发。当然能否进行转发还需要根据主机是否被配置为具有转发功能。具体的代码实现在1343-1349行，如果主机可以进行数据包转发，则调用ip_forward函数完成具体的转发功能，否则只是更新一些统计变量，并简单丢弃所接收的数据包。下面我们即进行ip_forward函数的具体分析。该函数实现代码相对较长，但实现思想较为简单：

1〉 检查可能有的防火墙转发规则，对数据包进行过滤。

2〉 进行路由选择。

3〉 调用下层发送函数，将数据包转发（发送）出去。

```
1070    #ifdef CONFIG_IP_FORWARD

1071    /*
1072     *  Forward an IP datagram to its next destination.
1073     */

1074    static void ip_forward(struct sk_buff *skb, struct device *dev,
                               int is_frag)
1075    {
```

参数说明：

skb：被转发数据包。

dev：接收该被转发数据包的网络设备。函数将用以判断是否需要进行ICMP重定向报文的发送。

is_frag：表示被转发数据包是否为一个分片数据包，以及分片所处的位置。当is_frag&1为真时，表示这是一个位置处于中间的分片数据包；当is_frag&2为真时，表示这是一个位置处于最后的分片数据包；由于ip_rcv函数实现的缺陷，没有对第一个分片进行判断，所以此处也就无法表示位置处于最前的第一个分片数据包。

```
1076        struct device *dev2;    /* Output device */
1077        struct iphdr *iph;  /* Our header */
1078        struct sk_buff *skb2;   /* Output packet */
1079        struct rtable *rt;  /* Route we use */
1080        unsigned char *ptr; /* Data pointer */
1081        unsigned long raddr;    /* Router IP address */

1082        /*
```

```
1083         *  See if we are allowed to forward this.
1084         */

1085    #ifdef CONFIG_IP_FIREWALL
1086        int err;

1087       if((err=ip_fw_chk(skb->h.iph, dev, ip_fw_fwd_chain, ip_fw_fwd_policy,
0))!=1)
1088        {
1089            if(err==-1)
1090                icmp_send(skb, ICMP_DEST_UNREACH, ICMP_HOST_UNREACH, 0, dev);
1091            return;
1092        }
1093    #endif
```

1085-1093是检查防火墙转发规则对被转发数据包进行过滤。如果转发规则不允许对该数据包进行转发，则发送一个错误为ICMP_DEST_UNREACH的ICMP错误报文给数据包发送起始端。有关防火墙的代码均定义在ip_fw.c中，在对该文件进行分析会着重进行讲解，此处读者简单理解即可。所谓防火墙规则就是对数据包中包含的双方IP地址，端口号等进行限制的一些规定，如果一个数据包满足这些规则，我们说命中，此时根据这条规则所规定的策略，来决定数据包的丢弃（或者转发）与否。所有的转发规则都定义在ip_fw_fwd_chain指向的队列中，ip_fw_fwd_policy为该队列的默认策略，所谓默认策略，即表示如果其中一条规则没有明确定义策略，就是用该默认策略来决定一个命中数据包的“命运”。更加详细的分析在介绍ip_fw.c时进行说明。

```
1094        /*
1095         *  According to the RFC, we must first decrease the TTL field. If
1096         *  that reaches zero, we must reply an ICMP control message telling
1097         *  that the packet's lifetime expired.
1098         *
1099         *  Exception:
1100         *  We may not generate an ICMP for an ICMP. icmp_send does the
1101         *  enforcement of this so we can forget it here. It is however
1102         *  sometimes VERY important.
1103         */

1104        iph = skb->h.iph;
1105        iph->ttl--;
1106        if (iph->ttl <= 0)
1107        {
1108            /* Tell the sender its packet died... */
1109            icmp_send(skb, ICMP_TIME_EXCEEDED, ICMP_EXC_TTL, 0, dev);
1110            return;
1111        }
```

1104-1111行代码对数据包中包含的TTL值进行减一处理，TTL实际表示经过路由器的跳数，每当经过一个中间路由器时，路由器都对TTL值进行减一，如果TTL减为0，则需要丢弃对应的数据包，并发送一个ICMP错误报文，有些应用程序（如traceroute）就是利用这点进行工作的。

```
1112        /*
1113         *  Re-compute the IP header checksum.
1114         *  This is inefficient. We know what has happened to the header
1115         *  and could thus adjust the checksum as Phil Karn does in KA9Q
1116         */

1117        ip_send_check(iph);
```

由于IP首部中TTL字段值被改变，所以需要重新计算IP校验和，1117行调用ip_send_check函数完成该计算。在进行以上的处理后，下面即进入到数据包转发的具体操作：路由选择。

```
1118        /*
1119         * OK, the packet is still valid.  Fetch its destination address,
1120         * and give it to the IP sender for further processing.
1121         */

1122        rt = ip_rt_route(iph->daddr, NULL, NULL);
1123        if (rt == NULL)
1124        {
1125            /*
1126             *  Tell the sender its packet cannot be delivered. Again
1127             *  ICMP is screened later.
1128             */
1129            icmp_send(skb, ICMP_DEST_UNREACH, ICMP_NET_UNREACH, 0, dev);
1130            return;
1131        }
```

在介绍route.c函数中，我们已经对ip_rt_route等函数进行了分析，该函数以最终目的IP地址（iph->daddr）为关键字进行路由表查询，如果返回值为NULL，则表示无对应路由表项可到达目的端，此时需要对源端回复一个ICMP错误报文，报文类型为ICMP_DEST_UNREACH。否则我们将使用该表项进行数据包转发。

```
1132        /*
1133         * Gosh.  Not only is the packet valid; we even know how to
1134         * forward it onto its final destination.  Can we say this
1135         * is being plain lucky?
1136         * If the router told us that there is no GW, use the dest.
1137         * IP address itself- we seem to be connected directly...
```

```
1138         */

1139        raddr = rt->rt_gateway;
```

在分析路由表实现文件route.c时，我们说到本版本路由表实现中如果目的主机与本机在同一链路上时，则表项中网关地址被设置为0。1139，1140行代码即对此进行检查，如果网关地址不为0，则表示最终主机不在同一链路上，下一站还是一个中间路由器，此时需要以此网关地址本身为关键字进行路由表查询，寻找可直达该网关的网络设备接口。

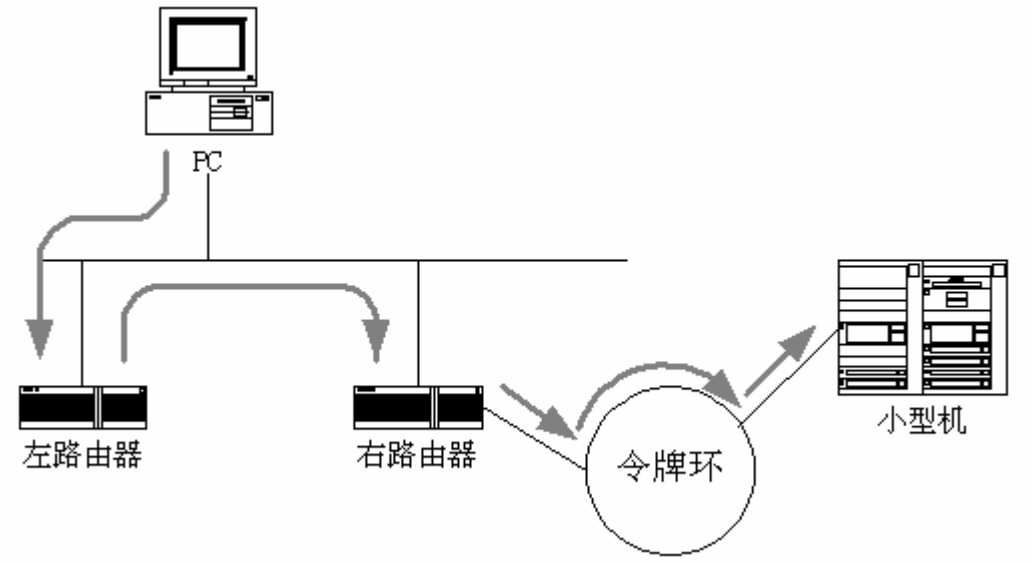
```
1140        if (raddr != 0)
1141        {
1142            /*
1143             *  There is a gateway so find the correct route for it.
1144             *  Gateways cannot in turn be gatewayed.
1145             */
1146            rt = ip_rt_route(raddr, NULL, NULL);
1147            if (rt == NULL)
1148            {
1149                /*
1150                 *  Tell the sender its packet cannot be delivered...
1151                 */
1152                icmp_send(skb, ICMP_DEST_UNREACH, ICMP_HOST_UNREACH, 0, dev);
1153                return;
1154            }
1155            if (rt->rt_gateway != 0)
1156                raddr = rt->rt_gateway;
1157        }
1158        else
1159            raddr = iph->daddr;
```

1139行将raddr设置为下一站网关地址，诚如上文所述，1146行以此地址为关键字，再一次调用ip_rt_route函数进行路由表查询，找到直达该网关的设备接口，如果1147行判断为真，则表示无法到达网关，同样对发送源端回复一个ICMP错误报文。1155行再次对返回表项中网关地址进行检查，正常情况下，此时rt->rt_gateway应该为0，如果非0，则表示可能进入一个路由循环，此处的处理方式只是简单将一站IP地址设置为rt->rt_gateway表示的值，即如果路由表项存在问题，则由此造成的数据包可能无法到达目的端的后果将是应得的，此处则假装“没看见”。1158行else语句与1140行对应，表示如果网关地址为0，则表示最终目的主机和本机处于同一链路上，所以1159行将下一站IP地址设置为最终目的主机IP地址，即我们终于“到家”了。

```
1160        /*
1161         *  Having picked a route we can now send the frame out.
1162         */
```

```
1163        dev2 = rt->rt_dev;

1164        /*
1165         *  In IP you never have to forward a frame on the interface that it
1166         *  arrived upon. We now generate an ICMP HOST REDIRECT giving the route
1167         *  we calculated.
1168         */
1169    #ifdef CONFIG_IP_NO_ICMP_REDIRECT
1170        if (dev == dev2)
1171            return;
1172    #else
1173        if (dev == dev2 && (iph->saddr&dev->pa_mask) == (iph->daddr &
dev->pa_mask))
1174            icmp_send(skb, ICMP_REDIRECT, ICMP_REDIR_HOST, raddr, dev);
1175    #endif
```

1163行将dev2初始化为到达下一站主机的接口设备，如果该接口与接收数据包的接口是同一个接口，则表示上一站路由器主机中存在一个不合理路由表项，请参考下图。



假设PC机接收到一个上面网络（没有画出）发送的数据包，需要转发给最终目的小型机，PC机查询得到的下一站路由器为左路由器，左路由器查询其路由表，发现其要转发数据包的下一站路由器（右路由器）所在的接口与接收被转发数据包所在的接口是同一个接口，就表示其上一站路由器主机（即PC机）存在一个不合理的路由表项，因为其（PC机）本可以直接将数据包转发给右路由器的，此时左路由器根据配置可以需要发送一个ICMP路由重定向报文给PC机，从而修正PC机的相应路由表项。以上1169-1175行代码即完成此项功能。这段代码嵌入在一个宏定义中，即重定向报文的发送需要在主机配置为可发送重定向报文的情况方可进行发送，否则对此不合理情况将“视而不见“。1173行if语句中，对源，目的IP地址的掩码判断是出于安全考虑，因为ICMP路由重定向报文在某些情况下可能被利用作为系统安全漏洞被攻击，详细情况读者可自行查阅相关资料。

在经过以上所有的处理后，现在我们就可以将此数据包真正的转发出去，从表面上理解，所谓转发数据包，就是将原来的数据包直接变换一下，就发送出去，实际上这种操作方式也可行，但是下面的这段代码却是重新创建一个新的数据包，将数据从被转发数据包中复制到该新创建的数据包中，最后将这个新创建的数据包发送出去完成数据包的转发功能。如下大部分代码都比较容易理解，对于需要进行说明的地方就是可能需要进行数据包分片发送，具体请看下文中对应分析。

```
1176        /*
1177         * We now allocate a new buffer, and copy the datagram into it.
1178         * If the indicated interface is up and running, kick it.
1179         */

1180        if (dev2->flags & IFF_UP)
1181        {

1182            /*
1183             *  Current design decrees we copy the packet. For identical header
1184             *  lengths we could avoid it. The new skb code will let us push
1185             *  data so the problem goes away then.
1186             */

1187            skb2 = alloc_skb(dev2->hard_header_len + skb->len, GFP_ATOMIC);
1188            /*
1189             *  This is rare and since IP is tolerant of network failures
1190             *  quite harmless.
1191             */
1192            if (skb2 == NULL)
1193            {
1194                printk("\nIP: No memory available for IP forward\n");
1195                return;
1196            }
1197            ptr = skb2->data;
1198            skb2->free = 1;
1199            skb2->len = skb->len + dev2->hard_header_len;
1200            skb2->h.raw = ptr;

1201            /*
1202             *  Copy the packet data into the new buffer.
1203             */
1204            memcpy(ptr + dev2->hard_header_len, skb->h.raw, skb->len);

1205            /* Now build the MAC header. */
1206            (void) ip_send(skb2, raddr, skb->len, dev2, dev2->pa_addr);
```

```
1207            ip_statistics.IpForwDatagrams++;

1208            /*
1209             *  See if it needs fragmenting. Note in ip_rcv we tagged
1210             *  the fragment type. This must be right so that
1211             *  the fragmenter does the right thing.
1212             */

1213            if(skb2->len > dev2->mtu + dev2->hard_header_len)
1214            {
1215                ip_fragment(NULL,skb2,dev2, is_frag);
1216                kfree_skb(skb2,FREE_WRITE);
1217            }
```

1213-1217行对可能需要进行的数据包分片发送的情况进行检查，1213行中skb2->len表示被转发数据包的长度，dev2->mtu加上dev2->hard_header_len表示通过dev2接口可发送数据报的最大长度，如果if语句判断为真，则表示在发送之前需要进行数据包分片，1215行即调用ip_fragment函数完成具体的分片发送。这个函数我们刚刚介绍，此处不多作论述。

```
1218            else
1219            {
1220    #ifdef CONFIG_IP_ACCT
1221                /*
1222                 *  Count mapping we shortcut
1223                 */

1224                ip_acct_cnt(iph,dev,ip_acct_chain);
1225    #endif
```

1224行调用调用ip_acct_cnt进行信息统计，acct表示accounting,意为统计，与通常MIB信息管理统计不同的是，此处统计是以规则匹配为前提的，所有的规则均有ip_acct_chain指向的规则队列表示，相关具体情况在介绍ip_fw.c文件时进行说明。

```
1226                /*
1227                 *  Map service types to priority. We lie about
1228                 *  throughput being low priority, but it's a good
1229                 *  choice to help improve general usage.
1230                 */
1231                if(iph->tos & IPTOS_LOWDELAY)
1232                    dev_queue_xmit(skb2, dev2, SOPRI_INTERACTIVE);
1233                else if(iph->tos & IPTOS_THROUGHPUT)
1234                    dev_queue_xmit(skb2, dev2, SOPRI_BACKGROUND);
1235                else
1236                    dev_queue_xmit(skb2, dev2, SOPRI_NORMAL);
```

1231-1236行代码提供了不同服务质量的实现，这段代码根据数据包中TOS(Type Of Service：服务质量）字段值将该被转发数据包插入的硬件缓冲区的不同级别队列中，当前硬件缓冲区有三个队列表示，每个队列具有不同的优先级，高优先级队列中的数据包将得到优先发送。相关的三个优先级定义在include/linux/socket.h中：

```
/*include/linux/socket.h*/
90  /* The various priorities. */
91  #define SOPRI_INTERACTIVE   0
92  #define SOPRI_NORMAL        1
93  #define SOPRI_BACKGROUND    2
```

而三个硬件缓冲队列定义在device结构中，具体如下：

```
/*include/linux/netdevice.h*/
29  /* for future expansion when we will have different priorities. */
30  #define DEV_NUMBUFFS    3
……
54  struct device
55  {
    ……
111   /* Pointer to the interface buffers. */
112   struct sk_buff_head     buffs[DEV_NUMBUFFS];

    ……
138 };//struct device
```

每个队列对应device结构buffs数组中的一个元素，即buffs数组中每个元素表示一个数据包缓冲队列，DEV_NUMBUFFS在30行被定义为3，则目前有三个不同优先级的硬件缓冲队列，socket.h中91-93行定义了表示这三个不同优先级的变量。硬件驱动程序在发送数据包时，将首先处理高优先级队列中的数据包，只有高优先级队列被清空后，才对低优先级队列中的数据包进行处理，这种方式提供了一种不同服务质量的服务，当然这只是对涉及服务质量的一种简单实现方式，真正要提供各种不同质量的服务，实现上要复杂的多。

```
1237            }
1238        }
1239    }

1240    #endif
```

至此我们完成对ip_forward函数的分析，诚如前文所述，该函数被ip_rcv函数调用完成数据包转发功能。虽然代码较长，但基于的实现思想却较简单，虽然在介绍该函数之前已经给出相关总结，不过我们不厌其烦的再一次给出总结如下：

1〉 进行转发规则检查，对数据包进行过滤。

2〉 进行路由选择，获取下一站地址（对前一站进行可能的路由重定向操作）。

3〉 进行可能的分片操作检查。

4〉 调用下层发送函数将数据包转发出去。

通常我们都说网络栈是分层次的，无论是经典的四层模型还是OSI七层模型，我们在理解上总是情愿于这些层次之间是完全独立的。然而在实现上，各层次之间只能说从功能上划分是具有分层结构，如TCP协议以tcp_rcv函数作为对下层的接口，此后均由该函数负责调用TCP协议实现模块的其他函数完成数据的处理，然而对于TCP协议可靠性数据传输的实现，除了在TCP协议本身实现模块中存在对应的实现代码外，IP协议模块中也存在有对应的实现代码，而且这些IP协议模块提供的对TCP协议可靠性数据传输的支持非常重要，这些支持中之一就是对使用TCP协议的数据包发送后进行缓存，从而用于之后可能的超时重发。所以我们说网络栈是分层的，大多是从纯理论上讨论，一旦涉及到具体实现，各层次之间代码是相互辅助的，通常无法做到完全的接口独立性。当然诚如刚才所述，每个层次都有一个总入口函数，这个总入口函数可以作为对应模块的中心枢纽，其他函数都将围绕这个中心函数，或者换句话说，实现具体功能的其他函数将由这个中心函数进行调用。如TCP协议对应总入口函数为tcp_rcv,UDP协议对应总入口函数为udp_rcv, ICMP对应icmp_rcv等等。在分析网络栈实现时，我们可以以这些总入口函数为基地，“攻克”其他所有实现。对应IP协议的这样一个总入口中心函数即为ip_rcv. 另外我们在分析TCP，UDP，ICMP协议时说道，网络层协议实现模块如何将数据包传递给对应的传输层协议，这是根据网络层协议首部中的对应字段进行判断的。同理链路层协议实现模块也是根据相关协议字段进行判断的。一般对于链路层，常用的以太网，802.3封装结构，在其协议首部中都对应有上层使用协议字段，根据该字段链路层将数据包传递给网络层协议模块，如对于IP协议，对应的协议字段值为0x0800；对于ARP协议，对应协议字段值为0x0806。

下面我们开始对IP协议总入口函数ip_rcv进行分析，在分析该函数之前，我们还是先看其被调用环境。在本书前文中，我们分析到网络层协议如何对上层传输层协议进行调用的：每个传输层协议创建一个inet_protocol结构，所有的传输层协议对应的inet_protocol结构构成一个系统链表，由专门变量指向。inet_protocol结构中包含有传输层协议对应的协议号，总入口函数，错误处理函数以及其他辅助信息。当网络层协议完成对数据包的处理后，其根据其首部中对应的传输层协议号，寻找到对应的inet_protocol结构，从而得到该传输层协议的总入口函数进行调用，从而将数据包传递给传输层协议进行处理，这个过程在前文相关部分已经进行了较为详细的分析，此处不再叙述。现在我们关心的是链路层模块是如何调用合适的网络层协议总入口函数的，实现上类似于网络层协议模块对传输层协议模块的调用。每个网络层协议协议都维护有一个数据结构，具体为packet_type结构，这个结构定义在include/linux/netdevice.h中，如下：

```
/*include/linux/netdevice.h*/
139 struct packet_type {
140   unsigned short    type;   /* This is really htons(ether_type). */
141   struct device *   dev;
142   int           (*func) (struct sk_buff *, struct device *,
143                  struct packet_type *);
144   void          *data;
145   struct packet_type    *next;
146 };
```

同理type字段表示网络层协议号，如IP协议对应为0x0800，ARP对应为0x0806，RARP对应为

0x8035等；func函数指针指向对应网络层协议的总入口函数；data为网络层协议自定义私有数据；next指针构成一个packet_type结构的队列，这个队列有ptype_base系统变量指向，这个系统变量定义在net/inet/dev.c中，dev.c文件可以认为是链路层接口模块，其负责驱动程序与网络层协议实现模块之间的交互。驱动程序接收到一个数据包后，调用netif_rx函数（dev.c）将数据包传递给接口模块，接口模块根据链路层协议首部中对应网络层协议号查找ptype_base指向的队列，从中寻找到对应的packet_type结构，从而调用该协议对应的总入口函数，将数据包传递给网络层协议进行处理。

IP协议对应的packet_type结构定义如下（ip.c）：

```
2027    /*
2028     *  IP protocol layer initialiser
2029     */

2030    static struct packet_type ip_packet_type =
2031    {
2032        0,  /* MUTTER ntohs(ETH_P_IP),*/
2033        NULL,   /* All devices */
2034        ip_rcv,
2035        NULL,
2036        NULL,
2037    };
```

对照packet_type结构本身的定义，我们可以看到IP协议对应总入口函数即为ip_rcv,注意此处IP协议对应的这个packet_type结构并不完整，因为其将协议号设置为0，对于IP协议而言，应该设置为0x0800，这个协议号的正确设置是在ip_init函数中完成的，ip_init函数也定义在ip.c中，我们不妨提前对该函数进行介绍。

```
2052    /*
2053     *  IP registers the packet type and then calls the subprotocol initialisers
2054     */

2055    void ip_init(void)
2056    {
2057        ip_packet_type.type=htons(ETH_P_IP);
2058        dev_add_pack(&ip_packet_type);

2059        /* So we flush routes when a device is downed */
2060        register_netdevice_notifier(&ip_rt_notifier);
2061    /*  ip_raw_init();
2062        ip_packet_init();
2063        ip_tcp_init();
2064        ip_udp_init();*/
2065    }
```

2057行将IP协议对应的packet_type结构中协议号正确设置为0x0800（ETH_P_IP），并调用

dev_add_pack(dev.c)函数将IP协议对应的packet_type（由变量ip_packet_type表示）插入到ptype_base指向的队列中，完成对下层的注册。2060行register_netdevice_notifier函数的调用是注册网络接收设备事件处理函数，所涉及的其他相关代码如下：

```
2038    /*
2039     *  Device notifier
2040     */

2041    static int ip_rt_event(unsigned long event, void *ptr)
2042    {
2043        if(event==NETDEV_DOWN)
2044            ip_rt_flush(ptr);
2045        return NOTIFY_DONE;
2046    }

2047    struct notifier_block ip_rt_notifier={
2048        ip_rt_event,
2049        NULL,
2050        0
2051    };
```

即具体的事件处理函数为ip_rt_event，当前只对网络设备停止工作事件（NETDEV_DOWN）做出响应，其调用ip_rt_flush函数对涉及到该网络设备的路有表项进行清除。

网络层协议被调用处理总结如下：每个网络层协议都对应有一个packet_type结构，在结构中声明其总入口函数，链路层接口模块在接收到一个数据包后，根据下层协议首部中对应的网络层协议号查找ptype_base指向的packet_type结构队列，寻找到网络层协议对应的packet_type结构，调用该结构中func指针指向的网络层协议总入口函数，完成数据包向网络层的传递。对应链路层接口模块中的具体调用环境如下：

```
/*net/inet/dev.c-net_bh()函数*/
pt_prev = NULL;
for (ptype = ptype_base; ptype != NULL; ptype = ptype->next)
{
    if ((ptype->type == type || ptype->type == htons(ETH_P_ALL)) &&
                  (!ptype->dev || ptype->dev==skb->dev))
            {
                /*
                 *  We already have a match queued. Deliver
                 *  to it and then remember the new match
                 */
                if(pt_prev)
                {
                    struct sk_buff *skb2;

                    skb2=skb_clone(skb, GFP_ATOMIC);
```

```
                    /*
                     *  Kick the protocol handler. This should be fast
                     *  and efficient code.
                     */

                    if(skb2)
                        pt_prev->func(skb2, skb->dev, pt_prev);
                }
                /* Remember the current last to do */
                pt_prev=ptype;
        }//if
} /* End of protocol list loop */

if(pt_prev)
    pt_prev->func(skb, skb->dev, pt_prev);
```

以上这段代码摘自dev.c文件中的net_bh函数，具体情景在分析dev.c时进行说明，读者现在只需要了解涉及到当前所叙主题的部分，即链路层接口模块是如何对网络层协议实现模块进行调用的。这段代码符合上文中我们的一再说明。在经过这一系列说明后，下面我们进入到对IP协议总入口函数ip_rcv的介绍。

```
1241    /*
1242     *  This function receives all incoming IP datagrams.
1243     */

1244    int ip_rcv(struct sk_buff *skb, struct device *dev, struct packet_type *pt)
1245    {
```

参数说明：
skb：待接收处理的数据包。
dev：当前被处理数据包的接收设备。
pt：IP协议对应的packet_type结构类型。

```
1246        struct iphdr *iph = skb->h.iph;
1247        struct sock *raw_sk=NULL;
1248        unsigned char hash;
1249        unsigned char flag = 0;
1250        unsigned char opts_p = 0;   /* Set iff the packet has options. */
1251        struct inet_protocol *ipprot; //每个传输层协议对应一个inet_protocol
1252        static struct options opt; /* since we don't use these yet, and they
1253                    take up stack space. */
1254        int brd=IS_MYADDR;
1255        int is_frag=0; //分片标志
```

```
1256    #ifdef CONFIG_IP_FIREWALL
1257        int err;
1258    #endif

1259        ip_statistics.IpInReceives++;

1260        /*
1261         *  Tag the ip header of this packet so we can find it
1262         */

1263        skb->ip_hdr = iph; //sk_buff结构中的ip_hdr字段专门指向IP首部

1264        /*
1265         *  Is the datagram acceptable?
1266         *
1267         *  1.  Length at least the size of an ip header
1268         *  2.  Version of 4
1269         *  3.  Checksums correctly. [Speed optimisation for later, skip
loopback checksums]
1270         *  (4. We ought to check for IP multicast addresses and undefined
types.. does this matter ?)
1271         */

1272        if (skb->len<sizeof(struct iphdr) || iph->ihl<5 || iph->version != 4
                        || ip_fast_csum((unsigned char *)iph, iph->ihl) !=0)
1273        {
1274            ip_statistics.IpInHdrErrors++;
1275            kfree_skb(skb, FREE_WRITE);
1276            return(0);
1277        }
```

1272-1277行代码检查数据包的合法性：
1〉数据包长度至少为IP首部长度，这样至少满足IP协议的处理要求，skb->len在网络设备驱动程序中被设置为接收长度，注意这个长度并非一定是数据包真正长度，因为可能存在填充字节。下面1294行代码才设置为数据包的真正长度。
2〉IP首部长度至少为20字节，且IP协议号为4（本版本不支持IPv6）。
3〉IP校验和无误。
如果以上条件之一不满足，则简单丢弃该数据包。

```
1278        /*
1279         *  See if the firewall wants to dispose of the packet.
1280         */

1281    #ifdef  CONFIG_IP_FIREWALL
```

```
1282        if ((err=ip_fw_chk(iph,dev,ip_fw_blk_chain,ip_fw_blk_policy, 0))!=1)
1283        {
1284            if(err==-1)
1285                icmp_send(skb, ICMP_DEST_UNREACH, ICMP_PORT_UNREACH, 0, dev);
1286            kfree_skb(skb, FREE_WRITE);
1287            return 0;
1288        }
1289    #endif
```

1281-1289行代码进行数据包过滤处理，通过查看ip_fw_blk_chain中现有规则，对数据包进行匹配，并进行对应的操作（丢弃或者接收），如果数据包匹配后的策略是丢弃，则回复一个ICMP错误报文后直接丢弃该数据包。

```
1290        /*
1291         *  Our transport medium may have padded the buffer out. Now we know
it
1292         *  is IP we can trim to the true length of the frame.
1293         */

1294        skb->len=ntohs(iph->tot_len);
```

1294行设置skb->len为数据包真正长度，剔出可能的填充字节。IP首部中总长度字段包含IP首部以及负载（传输层协议首部及用户数据）的总长度。

```
1295        /*
1296         *  Next analyse the packet for options. Studies show under one packet
in
1297         *  a thousand have options....
1298         */

1299        if (iph->ihl != 5)
1300        {   /* Fast path for the typical optionless IP packet. */
1301            memset((char *) &opt, 0, sizeof(opt));
1302            if (do_options(iph, &opt) != 0)
1303                return 0;
1304            opts_p = 1;
1305        }
```

IP首部固定长度为20字节，如果IP首部长度大于20字节，那么就表示存在IP选项，此时调用do_options函数对这些IP选项进行处理。注意IP首部中ihl字段（IP首部长度字段）值是以32比特为单位的。

```
1306        /*
1307         *  Remember if the frame is fragmented.
```

```
1308         */

1309        if(iph->frag_off)
1310        {
1311            if (iph->frag_off & 0x0020)
1312                is_frag|=1;
1313            /*
1314             *  Last fragment ?
1315             */

1316            if (ntohs(iph->frag_off) & 0x1fff)
1317                is_frag|=2;
1318        }
```

1309-1318判断所接收数据包是否为一个分片数据包。这段代码存在BUG，即只对中间分片和最后一个分片进行了判断，没有判断第一个分片。有关如何判断分片数据包的方法在前文中已经进行了详细的论述，此处不表。

```
1319        /*
1320         *  Do any IP forwarding required.  chk_addr() is expensive -- avoid
it someday.
1321         *
1322         *  This is inefficient. While finding out if it is for us we could also
compute
1323         *  the routing table entry. This is where the great unified cache
theory comes
1324         *  in as and when someone implements it
1325         *
1326         *  For most hosts over 99% of packets match the first conditional
1327         *  and don't go via ip_chk_addr. Note: brd is set to IS_MYADDR at
1328         *  function entry.
1329         */

1330        if ( iph->daddr != skb->dev->pa_addr && (brd = ip_chk_addr(iph->daddr))
== 0)
1331        {
1332            /*
1333             *  Don't forward multicast or broadcast frames.
1334             */

1335            if(skb->pkt_type!=PACKET_HOST || brd==IS_BROADCAST)
1336            {
1337                kfree_skb(skb,FREE_WRITE);
1338                return 0;
```

```
1339            }

1340            /*
1341             *  The packet is for another target. Forward the frame
1342             */

1343    #ifdef CONFIG_IP_FORWARD
1344            ip_forward(skb, dev, is_frag);
1345    #else
1346    /*      printk("Machine %lx tried to use us as a forwarder to %lx but we
have forwarding disabled!\n",
1347                iph->saddr,iph->daddr);*/
1348            ip_statistics.IpInAddrErrors++;
1349    #endif
1350            /*
1351             *  The forwarder is inefficient and copies the packet. We
1352             *  free the original now.
1353             */

1354            kfree_skb(skb, FREE_WRITE);
1355            return(0);
1356        }
```

1330-1356行代码处理数据包转发。首先1330行if语句判断当前接收数据包最终目的IP地址是否指向本机，如果不是，则调用ip_chk_addr函数进一步获取最终目的IP地址的类型（单播，组播，广播）。因为在需要进行数据包转发时，我们不能转发一个广播数据包（受限数据包）。如果满足转发条件，且本机被配置为具有转发功能，则1344行调用ip_forward进行数据包转发工作。

```
1357    #ifdef CONFIG_IP_MULTICAST

1358        if(brd==IS_MULTICAST && iph->daddr!=IGMP_ALL_HOSTS &&
                        !(dev->flags&IFF_LOOPBACK))
1359        {
1360            /*
1361             *  Check it is for one of our groups
1362             */
1363            struct ip_mc_list *ip_mc=dev->ip_mc_list;
1364            do
1365            {
1366                if(ip_mc==NULL)
1367                {
1368                    kfree_skb(skb, FREE_WRITE);
1369                    return 0;
```

```
1370                }
1371                if(ip_mc->multiaddr==iph->daddr)
1372                    break;
1373                ip_mc=ip_mc->next;
1374            }
1375            while(1);
1376        }
1377    #endif
```

1357-1377行代码处理多播情况，变量brd在1330行初始化为ip_chk_addr函数的返回值，这个返回值表示最终目的地址的类型，在include/linux/netdevice.h中定义有如下地址类型：

```
/*include/linux/netdevice.h*/
33  #define IS_MYADDR   1       /* address is (one of) our own  */
34  #define IS_LOOPBACK 2       /* address is for LOOPBACK  */
35  #define IS_BROADCAST    3       /* address is a valid broadcast */
36  #define IS_INVBCAST 4       /* Wrong netmask bcast not for us (unused)*/
37  #define IS_MULTICAST    5       /* Multicast IP address */
```

而在include/linux/igmp.h中则有：

```
/*include/linux/igmp.h*/
28  /* 224.0.0.1 */
29  #define IGMP_ALL_HOSTS      htonl(0xE0000001L)
```

1358行if语句首先检查最终目的地址是否为一个多播地址，并且将所有主机默认加入的多播地址（224.0.0.1）排除在外，且接收该多播数据包的设备不是回环设备。回环设备没有对应硬件，只存在于软件层面上，如果接收设备是一个回环设备，则表示该多播数据包是由本机发送的，对于多播和广播的情况，本机也将接收一个副本，所以无需进行多播列表匹配检查，直接进入下面的处理。对其其他多播地址，则需要检查多播列表，因为根据本书前文中相关介绍，由于多播MAC地址的特殊创建方式，不同多播地址可能对应同一个MAC多播地址，所以在接收到一个多播数据包后，还需要通过软件进行一次过滤，为此表示设备的device结构中维护一个多播列表，每个接收到的多播数据包将根据该列表进行匹配，只有成功匹配后，才将该多播数据包送往上层进一步处理，否则直接丢弃。1363-1375行代码完成的工作即如此。1366行表示检查完多播列表，还没有找到匹配项，则简单丢弃该多播数据包；否则1371-1372行表示找到一个匹配项，则跳出do-while循环，继续下面的处理。

```
1378        /*
1379         *  Account for the packet
1380         */

1381    #ifdef CONFIG_IP_ACCT
1382        ip_acct_cnt(iph,dev, ip_acct_chain);
1383    #endif
```

ip_acct_cnt用于信息统计，这个统计是进行分类的，分类的依据是首先进行规则匹配，这

些规则有ip_acct_chain变量指向。具体情景在分析ip_fw.c时进行说明。

```
1384        /*
1385         * Reassemble IP fragments.
1386         */

1387        if(is_frag)
1388        {
1389            /* Defragment. Obtain the complete packet if there is one */
1390            skb=ip_defrag(iph,skb,dev);
1391            if(skb==NULL)
1392                return 0;
1393            skb->dev = dev;
1394            iph=skb->h.iph;
1395        }
```

1387-1395对分片数据包进行可能的重组，因为所有分片可能尚未到达，所以如果存在未到达的分片，暂时还只能等待，现在只需将当前接收到的分片数据包缓存到分片队列中即可。此时ip_defrag返回为NULL，所以1392行直接返回。否则表示所有分片都已到达，而且重组完成，此时ip_defrag返回重组后的大数据包，我们可以继续下面的处理。

```
1396        /*
1397         *  Point into the IP datagram, just past the header.
1398         */

1399        skb->ip_hdr = iph;
1400        skb->h.raw += iph->ihl*4;
```

sk_buff结构用于封装一个数据包，ip_hdr字段专门指向IP首部，h字段是一个union类型，其子字段raw是一个char类型，这个h字段可根据数据包所处的网络栈不同层次进行更新，更新的依据是h字段始终指向当前正在处理层次所用协议的首部。1400行将h字段初始化为指向传输层首部，因为我们下面要将该数据包上传给传输层进行处理。

```
1401        /*
1402         *  Deliver to raw sockets. This is fun as to avoid copies we want to
make no surplus copies.
1403         */
```

本书前文中曾经花费大量篇幅讨论网络实现代码如何管理使用的当前所有套接字，以proto结构为基础，使用相同传输层协议的所有套接字将被统一保存到相关队列中。tcp_prot变量是一个proto结构，表示TCP协议的操作函数集，所以使用TCP协议进行数据传送的套接字sock结构都被缓存到tcp_prot对应结构所在的sock_array数组中，该数组每个元素均指向一个sock结构类型的链表。对于RAW类型套接字也不例外，raw_prot变量表示RAW协议的操作函数集，凡是使用RAW类型的所有套接字对应sock结构也都被缓存到raw_prot的sock_array数组

中。下面这段代码就是对使用RAW类型套接字的处理。首先以传输层使用协议号作为索引在raw_prot的sock_array数组找到对应sock结构队列，然后遍历该队列，对其中每个RAW类型套接字都复制一份数据包进行处理。

```
1404        hash = iph->protocol & (SOCK_ARRAY_SIZE-1);

1405        /* If there maybe a raw socket we must check - if not we don't care less
*/
1406        if((raw_sk=raw_prot.sock_array[hash])!=NULL)
1407        {
1408            struct sock *sknext=NULL;
1409            struct sk_buff *skb1;
1410            raw_sk=get_sock_raw(raw_sk, hash,  iph->saddr, iph->daddr);
1411            if(raw_sk)  /* Any raw sockets */
1412            {
1413                do
1414                {
1415                    /* Find the next */
1416                    sknext=get_sock_raw(raw_sk->next, hash, iph->saddr,
iph->daddr);
1417                    if(sknext)
1418                        skb1=skb_clone(skb, GFP_ATOMIC);
1419                    else
1420                        break;  /* One pending raw socket left */
1421                    if(skb1)
1422                        raw_rcv(raw_sk, skb1, dev, iph->saddr,iph->daddr);
1423                    raw_sk=sknext;
1424                }
1425                while(raw_sk!=NULL);
1426                /* Here either raw_sk is the last raw socket, or NULL if none
*/
1427                /* We deliver to the last raw socket AFTER the protocol checks
as it avoids a surplus copy */
1428            }
1429        }
```

1404-1429行代码处理RAW类型的套接字，其中get_sock_raw函数返回下一个源，目的IP地址都匹配的RAW类型套接字sock结构，raw_rcv函数用于接收数据包。以上代码只有一个不完整指出，即1419行退出时，最后一个RAW套接字没有对数据包进行接收，这个接收被推迟到1471行完成。在完成对RAW类型套接字的处理后，下面进入到数据包的正常处理，即将数据包传递给传输层协议进行处理。有关RAW类型套接字的具体使用和相关介绍，可参考《UNIX网络编程》一书。

```
1430        /*
```

```
1431         *  skb->h.raw now points at the protocol beyond the IP header.
1432         */

1433        hash = iph->protocol & (MAX_INET_PROTOS -1);
1434        for (ipprot = (struct inet_protocol *)inet_protos[hash];ipprot !=
NULL;ipprot=(struct inet_protocol *)ipprot->next)
1435        {
1436            struct sk_buff *skb2;

1437            if (ipprot->protocol != iph->protocol)
1438                continue;
1439           /*
1440        *   See if we need to make a copy of it.  This will
1441        *   only be set if more than one protocol wants it.
1442        *   and then not for the last one. If there is a pending
1443        *   raw delivery wait for that
1444        */
1445            if (ipprot->copy || raw_sk)
1446            {
1447                skb2 = skb_clone(skb, GFP_ATOMIC);
1448                if(skb2==NULL)
1449                    continue;
1450            }
1451            else
1452            {
1453                skb2 = skb;
1454            }
1455            flag = 1;

1456               /*
1457            * Pass on the datagram to each protocol that wants it,
1458            * based on the datagram protocol.  We should really
1459            * check the protocol handler's return values here...
1460            */
1461            ipprot->handler(skb2, dev, opts_p ? &opt : 0, iph->daddr,
1462                    (ntohs(iph->tot_len) - (iph->ihl * 4)),
1463                    iph->saddr, 0, ipprot);

1464        }
```

诚如本书前文所述，每个传输层协议有一个inet_protocol结构表示，所有传输层协议（或者更准确地说，应该是使用IP协议的其它协议）对应的inet_protocol结构构成由inet_protocol_base变量指向的队列，在网络栈初始化过程中，该队列中inet_protocol结构以协议号为关键字被散列到inet_protos数组中，便于查找。1433-1464行代码即以协议号

为关键字寻找到inet_protos数组中对应元素指向的队列，对该队列中每个inet_protocol结构中定义的接收函数进行调用，从而完成数据包从网络层到传输层的传递。由于inet_protos数组元素个数的有限性，所以不同协议号可能散列到一个元素上，1437-1438行对此进行进一步检查，以确定协议号相符。1445行检查这是否为最后一次数据包传递，如果是，则无需进行数据包复制，如果此后还需要进行数据包传递，则必须进行数据包复制。变量raw_sk是处理RAW类型套接字的遗留物，正如前文所述，在处理RAW类型套接字时，最后个raw套接字没有进行数据包的接收，这在1471-1472行才进行了处理。所以如果raw_sk不为NULL，则表示下面还需要这个数据包，所以当前对数据包进行处理时，必须首先创建一个副本。如果是对这个数据包的最后一个处理，则如1453行代码直接使用这个原始数据包即可。我们在介绍inet_add_protocol(protocol.c)函数时，注意到如果加入了相同协议号的两个inet_protocol结构，则前一个inet_protocol结构中的copy字段被设置为1，更进一步说，在相同队列中（由于具有相同协议号，一定处在相同队列中）具有相同协议号的所有inet_protocol结构，除了最后一个inet_protocol结构外，其他inet_protocol结构中copy字段都被设置为1，最后一个inet_protocol结构中copy字段设置为0，用来标志具有相同协议号的inet_protocol结构的结束。1445行对于inet_protocol结构copy字段的检查就是在检查是否还有后续的满足接收条件的inet_protocol结构，如果有（copy=1），则必须进行数据包复制。1461行调用在inet_protocol结构中注册的数据包处理函数句柄完成数据包从网络层到传输层的传递。注意1455行flag变量设置为1，表示数据包得到本地接收，如果flag为0，则表示本地没有合适的传输层处理函数，即在传输层没有对应的接收进程，这个flag标志在下面的1473行进行检查，从而决定是否需要发送的目的主机不可达（无对应传输层协议）ICMP错误通报报文。当然如果被接收的数据包是一个多播或者广播数据包，则无需回复ICMP报文。

```
1465        /*
1466         * All protocols checked.
1467         * If this packet was a broadcast, we may *not* reply to it, since that
1468         * causes (proven, grin) ARP storms and a leakage of memory (i.e. all
1469         * ICMP reply messages get queued up for transmission...)
1470         */

1471        if(raw_sk!=NULL)    /* Shift to last raw user */
1472            raw_rcv(raw_sk, skb, dev, iph->saddr, iph->daddr);
1473        else if (!flag)     /* Free and report errors */
1474        {
1475            if (brd != IS_BROADCAST && brd!=IS_MULTICAST)
1476                icmp_send(skb, ICMP_DEST_UNREACH, ICMP_PROT_UNREACH, 0, dev);
1477            kfree_skb(skb, FREE_WRITE);
1478        }

1479        return(0);
1480    }
```

至此，我们完成对ip_rcv函数的分析，该函数作为IP协议实现模块的总入口函数，起着一个任务派发者的作用。可以将ip_rcv函数所作的工作总结如下:

1> 数据包合法性检查。
2> 防火墙过滤。
3> 分片判断。
4> 转发处理。
5> 多播处理。
6> 分片重组处理。
7> RAW类型套接字处理。
8> 调用传输层接口函数完成数据包的向上层传递。

```
1481    /*
1482     *  Loop a packet back to the sender.
1483     */

1484    static void ip_loopback(struct device *old_dev, struct sk_buff *skb)
1485    {
1486        extern struct device loopback_dev;
1487        struct device *dev=&loopback_dev;
1488        int len=skb->len-old_dev->hard_header_len;
1489        struct sk_buff *newskb=alloc_skb(len+dev->hard_header_len,
GFP_ATOMIC);

1490        if(newskb==NULL)
1491            return;

1492        newskb->link3=NULL;
1493        newskb->sk=NULL;
1494        newskb->dev=dev;
1495        newskb->saddr=skb->saddr;
1496        newskb->daddr=skb->daddr;
1497        newskb->raddr=skb->raddr;
1498        newskb->free=1;
1499        newskb->lock=0;
1500        newskb->users=0;
1501        newskb->pkt_type=skb->pkt_type;
1502        newskb->len=len+dev->hard_header_len;


1503        newskb->ip_hdr=(struct iphdr *)(newskb->data+ip_send(newskb,
                    skb->ip_hdr->daddr, len, dev, skb->ip_hdr->saddr));
1504        memcpy(newskb->ip_hdr,skb->ip_hdr,len);

1505        /* Recurse. The device check against IFF_LOOPBACK will stop infinite
recursion */
```

```
1506        /*printk("Loopback output queued [%lX to %lX].\n",
                        newskb->ip_hdr->saddr,newskb->ip_hdr->daddr);*/
1507        ip_queue_xmit(NULL, dev, newskb, 1);
1508    }
```

ip_loopback函数被调用以向本机回送一个数据包。注意1487行代码将dev变量设置为loopback_dev,这个变量在1507行用于ip_queue_xmit函数调用。这个变换很重要，否则将会造成函数调用的死循环，因为ip_queue_xmit函数中也存在对ip_loopback函数的调用。ip_loopback函数实现比较简单，此处不多做讨论。

```
1509    /*
1510     * Queues a packet to be sent, and starts the transmitter
1511     * if necessary.  if free = 1 then we free the block after
1512     * transmit, otherwise we don't. If free==2 we not only
1513     * free the block but also don't assign a new ip seq number.
1514     * This routine also needs to put in the total length,
1515     * and compute the checksum
1516     */

1517    void ip_queue_xmit(struct sock *sk, struct device *dev,
1518               struct sk_buff *skb, int free)
1519    {
```

ip_queue_xmit函数作为数据包发送函数，被网络层和传输层协议共同调用。如果说ip_rcv是数据包上行通道函数，那么ip_queue_xmit就是数据包下行通道函数。在完成一个数据桢的创建后，ip_queue_xmit函数即被调用将数据包发往下层（链路层，通过调用dev_queue_xmit函数）进行处理。所以ip_queue_xmit是通往链路层的功能接口函数，所处的位置和工作十分重要。在分析ip_fragment函数时，每当完成一个分片的创建，ip_queue_xmit就被调用，将这个被创建分片发送出去。而在TCP协议，UDP协议中，ip_queue_xmit函数则被调用的相当频繁，在这些协议实现文件中，调用时通过函数指针完成的，tcp_prot->queue_xmit, udp_prot->queue_xmit都指向ip_queue_xmit函数，而ICMP协议实现中，数据包的发送则是直接调用ip_queue_xmit函数。下面我们就对ip_queue_xmit函数进行分析。

参数说明：
sk：被发送数据包对应的套接字。
dev：发送数据包的网络设备。
skb：被发送的数据包。
free：是否对数据包进行缓存以便于此后的超时重发，该字段主要配合TCP协议工作。UDP协议，ICMP协议等在调用ip_queue_xmit时将该参数设置为1。

```
1520        struct iphdr *iph;
1521        unsigned char *ptr;
```

```
1522        /* Sanity check */
1523        if (dev == NULL)
1524        {
1525            printk("IP: ip_queue_xmit dev = NULL\n");
1526            return;
1527        }

1528        IS_SKB(skb);
```

1523-1527行对发送设备进行检查；1528对数据包的合法性进行检查。IS_SKB主要检查封装数据包的sk_buff结构的相关字段。

```
1529        /*
1530         *  Do some book-keeping in the packet for later
1531         */



1532        skb->dev = dev;
1533        skb->when = jiffies;
```

sk_buff结构when字段表示数据包发送时间，用于超时重传检查。因为超时定时器只有一个，而每当发送数据包时，该定时器即被重新设置，一旦定时器超时，最近发送的数据包有可能超时，原先发送的数据包也可能需要重传，设置一个发送时间字段便于检查。

```
1534        /*
1535         *  Find the IP header and set the length. This is bad
1536         *  but once we get the skb data handling code in the
1537         *  hardware will push its header sensibly and we will
1538         *  set skb->ip_hdr to avoid this mess and the fixed
1539         *  header length problem
1540         */

1541        ptr = skb->data;
1542        ptr += dev->hard_header_len;
1543        iph = (struct iphdr *)ptr;
1544        skb->ip_hdr = iph;
1545        iph->tot_len = ntohs(skb->len-dev->hard_header_len);



1546    #ifdef CONFIG_IP_FIREWALL
1547        if(ip_fw_chk(iph, dev, ip_fw_blk_chain, ip_fw_blk_policy, 0) != 1)
1548            /* just don't send this packet */
1549            return;
```

```
1550    #endif
```

1546-1550行代码进行数据包过滤，这与ip_rcv函数中对被接收数据包的过滤基本理念一致，用于防火墙安全性检查。

```
1551        /*
1552         *  No reassigning numbers to fragments...
1553         */

1554        if(free!=2)
1555            iph->id      = htons(ip_id_count++);
1556        else
1557            free=1;
```

在ip_fragment中调用ip_queue_xmit函数发送分片数据包时，free参数被设置为2，表示这是在调用ip_queue_xmit函数发送一个分片数据包。1554行对分片数据包进行此检查，如果free=2，则表示这是一个分片数据包，此时不可对IP首部中的ID字段进行更新，因为所有的分片必须具有相同的ID字段值，这样接收端才能对他们进行重组，一旦判断出是一个分片数据包，1557行将free参数重新设置为1，便于下文代码的统一检查；对于一个非分片数据包，则需要设置IP首部中的ID字段值，ip_id_count++是一个整型全局变量，每次发送一个数据包时，该变量值加1，用于下一个数据包中IP首部中ID字段的赋值。

```
1558        /* All buffers without an owner socket get freed */
1559        if (sk == NULL)
1560            free = 1;
```

如果数据包无对应套接字，则将free参数设置为1，因为没有对应sock结构，则无法对数据包进行缓存，所以在将数据包发往下层后，释放数据包（注意发往下层处理返回后，才释放，如果下层无法成功发送数据包，则本层进行释放后，即表示数据包发送失败）。

```
1561        skb->free = free;

1562        /*
1563         *  Do we need to fragment. Again this is inefficient.
1564         *  We need to somehow lock the original buffer and use
1565         *  bits of it.
1566         */

1567        if(skb->len > dev->mtu + dev->hard_header_len)
1568        {
1569            ip_fragment(sk,skb,dev,0);
1570            IS_SKB(skb);
1571            kfree_skb(skb,FREE_WRITE);
1572            return;
```

```
1573        }
```

1567行对数据包的长度进行检查，如果数据包长度大于MTU，则进行分片发送，这是通过调用ip_fragment函数完成，ip_fragment函数本身每当完成一个分片的创建后，又重新调用ip_queue_xmit函数发送分片，此时1567行代码就不再为真，直接进入下面的数据包发送工作。

```
1574        /*
1575         *  Add an IP checksum
1576         */

1577        ip_send_check(iph); //IP首部校验和计算。

1578        /*
1579         *  Print the frame when debugging
1580         */

1581        /*
1582         *  More debugging. You cannot queue a packet already on a list
1583         *  Spot this and moan loudly.
1584         */
1585        if (skb->next != NULL)
1586        {
1587            printk("ip_queue_xmit: next != NULL\n");
1588            skb_unlink(skb);
1589        }
```

在发送一个数据包时，这个数据包应该不属于任何队列，如果当前被发送数据包仍然被缓存在某个队列中，同时又要发送该数据包，则表示可能出现的内核不一致的问题，在处理上，打印一个警告信息，并在发送数据包之前，将数据包从相关队列中删除。

```
1590        /*
1591         *  If a sender wishes the packet to remain unfreed
1592         *  we add it to his send queue. This arguably belongs
1593         *  in the TCP level since nobody else uses it. BUT
1594         *  remember IPng might change all the rules.
1595         */

1596        if (!free)
1597        {
1598            unsigned long flags;
1599            /* The socket now has more outstanding blocks */

1600            sk->packets_out++;
```

```
1601            /* Protect the list for a moment */
1602            save_flags(flags);
1603            cli();

1604            if (skb->link3 != NULL)
1605            {
1606                printk("ip.c: link3 != NULL\n");
1607                skb->link3 = NULL;
1608            }
1609            if (sk->send_head == NULL)
1610            {
1611                sk->send_tail = skb;
1612                sk->send_head = skb;
1613            }
1614            else
1615            {
1616                sk->send_tail->link3 = skb;
1617                sk->send_tail = skb;
1618            }
1619            /* skb->link3 is NULL */

1620            /* Interrupt restore */
1621            restore_flags(flags);
1622        }
1623        else
1624            /* Remember who owns the buffer */
1625            skb->sk = sk;
```

1596-1625行代码是对数据包是否需要缓存的处理。对于TCP协议等提供可靠性数据传输的协议，在发送数据包之前（或者之后）需要对数据包进行缓存，防止数据包在传输过程中被丢弃，缓存的目的在于一旦发生丢弃的情况，可以进行重新发送，free参数用于标志数据包是否需要进行缓存。数据包重发缓存队列是由sock结构中send_head, send_tail两个字段表示的，这是一个单向队列，sk_buff结构中的link3字段用于队列中数据包之间的连接，而send_head指向队列头部，send_tail指向队列尾部。sk_buff结构中packets_out字段表示已发送出去但尚未得到应答的数据包个数。这个由send_head, send_tail指向的缓存队列是可靠性数据传输保证的实现之一。加上超时重传定时器等其他实现共同完成数据的可靠性传输。从此处我们亦可看出，TCP协议虽然属于传输层协议，但在实现可靠性数据传输上并非所有的代码都在传输层上实现，其他层（如此处的网络层）也存在实现代码，所以我们一再强调，网络层分层结构主要是从概念上或者说理论上讨论，真正涉及到实现，则往往各层之间相关交织，相互辅助。以上这段根据free参数对数据包进行缓存的代码比较简单，此处不再讨论。再次重申，这段代码主要是为配合TCP协议实现可靠性数据传输，读者可结合TCP协议实现相关函数（如retransmit_timer）理解这段代码。

```
1626        /*
1627         *  If the indicated interface is up and running, send the packet.
1628         */

1629        ip_statistics.IpOutRequests++;
1630    #ifdef CONFIG_IP_ACCT
1631        ip_acct_cnt(iph,dev, ip_acct_chain);
1632    #endif

1633    #ifdef CONFIG_IP_MULTICAST

1634        /*
1635         *  Multicasts are looped back for other local users
1636         */

1637        if (MULTICAST(iph->daddr) && !(dev->flags&IFF_LOOPBACK))
1638        {
1639            if(sk==NULL || sk->ip_mc_loop)
1640            {
1641                if(iph->daddr==IGMP_ALL_HOSTS)
1642                    ip_loopback(dev,skb);
1643                else
1644                {
1645                    struct ip_mc_list *imc=dev->ip_mc_list;
1646                    while(imc!=NULL)
1647                    {
1648                        if(imc->multiaddr==iph->daddr)
1649                        {
1650                            ip_loopback(dev,skb);
1651                            break;
1652                        }
1653                        imc=imc->next;
1654                    }
1655                }
1656            }
1657            /* Multicasts with ttl 0 must not go beyond the host */

1658            if(skb->ip_hdr->ttl==0)
1659            {
1660                kfree_skb(skb, FREE_READ);
1661                return;
1662            }
1663        }
1664    #endif
```

```
1665        if((dev->flags&IFF_BROADCAST) && iph->daddr==dev->pa_brdaddr
&& !(dev->flags&IFF_LOOPBACK))
1666            ip_loopback(dev,skb);
```

1633-1666行代码对多播和广播数据包进行处理，对于多播和广播数据包，按照要求，其必须复制一份会送给本机。这段代码完成的工作就是检查是否需要进行数据包的回送。注意1637行代码首先对目的地址是否为一个多播地址进行检查，在确定是一个多播地址后，进一步检查发送设备是否为一个回环设备，这一检查非常重要，防止进入函数调用循环。因为数据包回送是通过调用ip_loopback函数完成的，ip_loopback函数又重新调用ip_queue_xmit函数发送数据包，在下层（dev_queue_xmit函数中）数据包被回送到本机。正是由于数据包的回送是在下层（链路层）完成的，则由网络层进入到下层链路层的正常通道就是通过ip_queue_xmit函数调用，所以ip_loopback函数不得已而为之，又重新调用ip_queue_xmit函数，期望该函数将数据包发往下层，由下层处理函数将数据包又回送给本机。当然实现上我们完全可以直接在ip_loopback函数完成数据包的回送（如通过直接调用ip_rcv函数完成），但本版本网络实现代码中ip_loopback函数就是要调用ip_queue_xmit函数完成，所以但是在调用ip_queue_xmit函数之前，ip_loopback函数将发送设备换成了回环设备（由loopback_dev表示，这是一个纯软件设备），用于ip_queue_xmit函数检查，防止进行函数调用循环。1637行中对于回环设备的检测的原因即在如此。如果设备是一个回环设备，则表示ip_queue_xmit是被ip_loopback函数调用的，此时就不能在ip_queue_xmit中又调用ip_loopback函数。

1938行检查多播地址类型，如果是IGMP_ALL_HOSTS(224.0.0.1),则回送该数据包。所有主机默认加入多播地址224.0.0.1。否则检查多播地址列表，对数据包进行匹配，如果存在匹配项，则回送数据包没，否则继续以下的处理。

1956行检查IP首部中TTL字段，任何数据包，只要TTL值为0，都不可进行发送（转发）。从此处代码也可判断出多播数据包在可转发性上不同于广播数据包,多播数据包在TTL值不为0时可以进行转发，这也是有别于广播数据包之处。

1665行是对广播数据包的判断，device结构中pa_brdaddr表示设备的广播地址（初始化为全1），如果设备支持广播，则也回送该数据包。同样对回环设备进行检查防止进入函数调用循环。

```
1667        if (dev->flags & IFF_UP)
1668        {
1669            /*
1670             *  If we have an owner use its priority setting,
1671             *  otherwise use NORMAL
1672             */

1673            if (sk != NULL)
1674            {
1675                dev_queue_xmit(skb, dev, sk->priority);
1676            }
```

```
1677            else
1678            {
1679                dev_queue_xmit(skb, dev, SOPRI_NORMAL);
1680            }
1681        }
```

1667-1681行代码则是调用下层接口函数dev_queue_xmit将数据包发往链路层进行处理。1667行是对发送设备当前状态的检查，如果当前发送设备处于非工作状态，则无法发送数据包，此时进入1682行对应的else语句块执行，否则根据数据包是否属于一个套接字，进行dev_queue_xmit函数调用，是否属于一个套接字产生的影响是对数据包发送优先级的设置不同。

```
1682        else
1683        {
1684            ip_statistics.IpOutDiscards++;
1685            if (free)
1686                kfree_skb(skb, FREE_WRITE);
1687        }
1688    }
```

1682行else语句块对应发送设备当前处于非工作状态的情况进行处理，处理也较简单：直接丢弃数据包。对于提供可靠性数据传输的协议如TCP协议，这将造成此后的超时重传，则对于不提供可靠性数据传输的协议而言如UDP协议，则表示数据包发送失败。

至此，我们完成对ip_queue_xmit函数分析，该函数作为通往下层的接口函数，被网络层和传输层相关函数调用，将数据包发送给链路层处理。我们可以将该函数完成的主要功能总结如下：

1> 相关合法性检查。
2> 防火墙过滤。
3> 对数据包是否需要分片发送进行检查。
4> 进行可能的数据包缓存处理。
5> 对多播和广播数据包是否需要回送本机进行检查。
6> 调用下层接口函数dev_queue_xmit将数据包送往下层（链路层）进行处理。

```
1689    #ifdef CONFIG_IP_MULTICAST

1690    /*
1691     *  Write an multicast group list table for the IGMP daemon to
1692     *  read.
1693     */

1694    int ip_mc_procinfo(char *buffer, char **start, off_t offset, int length)
1695    {
```

```
1696        off_t pos=0, begin=0;
1697        struct ip_mc_list *im;
1698        unsigned long flags;
1699        int len=0;
1700        struct device *dev;

1701        len=sprintf(buffer,"Device    : Count\tGroup    Users Timer\n");
1702        save_flags(flags);
1703        cli();

1704        for(dev = dev_base; dev; dev = dev->next)
1705        {
1706                    if((dev->flags&IFF_UP)&&(dev->flags&IFF_MULTICAST))
1707                    {
1708                            len+=sprintf(buffer+len,"%-10s: %5d\n",
1709                        dev->name, dev->mc_count);
1710                            for(im = dev->ip_mc_list; im; im = im->next)
1711                            {
1712                                    len+=sprintf(buffer+len,
1713                        "\t\t\t%08lX %5d %d:%08lX\n",
1714                                            im->multiaddr, im->users,
1715                        im->tm_running, im->timer.expires);
1716                                pos=begin+len;
1717                                if(pos<offset)
1718                                {
1719                                        len=0;
1720                                        begin=pos;
1721                                }
1722                                if(pos>offset+length)
1723                                        break;
1724                            }
1725                    }
1726        }
1727        restore_flags(flags);
1728        *start=buffer+(offset-begin);
1729        len-=(offset-begin);
1730        if(len>length)
1731            len=length;
1732        return len;
1733    }

1734    #endif
```

ip_mc_procinfo函数用于获取多播地址信息。dev_base变量指向系统当前所有网络设备队列

头部，函数实现思想比较简单，结合device结构定义，读者自行分析。

```
1735    /*
1736     *  Socket option code for IP. This is the end of the line after any TCP,UDP
etc options on
1737     *  an IP socket.
1738     *
1739     *  We implement IP_TOS (type of service), IP_TTL (time to live).
1740     *
1741     *  Next release we will sort out IP_OPTIONS since for some people are kind
of important.
1742     */

1743    int ip_setsockopt(struct sock *sk, int level, int optname, char *optval,
                    int optlen)
1744    {
```

ip_setsockopt函数用于IP选项设置。参数sk表示对应被设置的套接字。

```
1745        int val,err;
1746    #if defined(CONFIG_IP_FIREWALL) || defined(CONFIG_IP_ACCT)
1747        struct ip_fw tmp_fw;
1748    #endif
1749        if (optval == NULL)
1750            return(-EINVAL);

1751        err=verify_area(VERIFY_READ, optval, sizeof(int));
1752        if(err)
1753            return err;

1754        val = get_fs_long((unsigned long *)optval);
```

1749-1754行代码对设置参数合法性进行检查，并获取参数值。

```
1755        if(level!=SOL_IP)
1756            return -EOPNOTSUPP;
```

1755检查选项层次，ip_setsockopt函数只支持IP层次的选项。
下面开始对各选项进行处理。

```
1757        switch(optname)
1758        {
1759            case IP_TOS:
```

```
1760                if(val<0||val>255)
1761                    return -EINVAL;
1762                sk->ip_tos=val;
1763                if(val==IPTOS_LOWDELAY)
1764                    sk->priority=SOPRI_INTERACTIVE;
1765                if(val==IPTOS_THROUGHPUT)
1766                    sk->priority=SOPRI_BACKGROUND;
1767                return 0;
1768            case IP_TTL:
1769                if(val<1||val>255)
1770                    return -EINVAL;
1771                sk->ip_ttl=val;
1772                return 0;
```

1759-1772对sock结构中一些字段进行设置，这些字段在创建IP首部时用到。另外1763-1766是对数据包发送优先级进行设置。

```
1773    #ifdef CONFIG_IP_MULTICAST
1774            case IP_MULTICAST_TTL:
1775            {
1776                unsigned char ucval;

1777                ucval=get_fs_byte((unsigned char *)optval);
1778                if(ucval<1||ucval>255)
1779                                    return -EINVAL;
1780                sk->ip_mc_ttl=(int)ucval;
1781                        return 0;
1782            }
1783            case IP_MULTICAST_LOOP:
1784            {
1785                unsigned char ucval;

1786                ucval=get_fs_byte((unsigned char *)optval);
1787                if(ucval!=0 && ucval!=1)
1788                     return -EINVAL;
1789                sk->ip_mc_loop=(int)ucval;
1790                return 0;
1791            }
1792            case IP_MULTICAST_IF:
1793            {
1794                /* Not fully tested */
1795                struct in_addr addr;
1796                struct device *dev=NULL;
```

```
1797                /*
1798                 *  Check the arguments are allowable
1799                 */

1800                err=verify_area(VERIFY_READ, optval, sizeof(addr));
1801                if(err)
1802                    return err;

1803                memcpy_fromfs(&addr,optval,sizeof(addr));

1804                printk("MC bind %s\n", in_ntoa(addr.s_addr));

1805                /*
1806                 *  What address has been requested
1807                 */

1808                if(addr.s_addr==INADDR_ANY) /* Default */
1809                {
1810                    sk->ip_mc_name[0]=0;
1811                    return 0;
1812                }

1813                /*
1814                 *  Find the device
1815                 */

1816                for(dev = dev_base; dev; dev = dev->next)
1817                {
1818                    if((dev->flags&IFF_UP)&&(dev->flags&IFF_MULTICAST)&&
1819                        (dev->pa_addr==addr.s_addr))
1820                        break;
1821                }

1822                /*
1823                 *  Did we find one
1824                 */

1825                if(dev)
1826                {
1827                    strcpy(sk->ip_mc_name,dev->name);
1828                    return 0;
1829                }
1830                return -EADDRNOTAVAIL;
1831            }
```

```
1832            case IP_ADD_MEMBERSHIP:
1833            {

1834    /*
1835     *  FIXME: Add/Del membership should have a semaphore protecting them from
re-entry
1836     */
1837                struct ip_mreq mreq;
1838                static struct options optmem;
1839                unsigned long route_src;
1840                struct rtable *rt;
1841                struct device *dev=NULL;

1842                /*
1843                 *  Check the arguments.
1844                 */

1845                err=verify_area(VERIFY_READ, optval, sizeof(mreq));
1846                if(err)
1847                    return err;

1848                memcpy_fromfs(&mreq,optval,sizeof(mreq));

1849                /*
1850                 *  Get device for use later
1851                 */

1852                if(mreq.imr_interface.s_addr==INADDR_ANY)
1853                {
1854                    /*
1855                     *  Not set so scan.
1856                     */
1857                    if((rt=ip_rt_route(mreq.imr_multiaddr.s_addr,&optmem,
&route_src))!=NULL)
1858                    {
1859                        dev=rt->rt_dev;
1860                        rt->rt_use--;
1861                    }
1862                }
1863                else
1864                {
1865                    /*
1866                     *  Find a suitable device.
```

```
1867                     */
1868                    for(dev = dev_base; dev; dev = dev->next)
1869                    {
1870
    if((dev->flags&IFF_UP)&&(dev->flags&IFF_MULTICAST)&&
1871                                (dev->pa_addr==mreq.imr_interface.s_addr))
1872                                break;
1873                    }
1874                }

1875                /*
1876                 *  No device, no cookies.
1877                 */

1878                if(!dev)
1879                    return -ENODEV;

1880                /*
1881                 *  Join group.
1882                 */

1883                return ip_mc_join_group(sk,dev,mreq.imr_multiaddr.s_addr);
1884            }

1885            case IP_DROP_MEMBERSHIP:
1886            {
1887                struct ip_mreq mreq;
1888                struct rtable *rt;
1889                static struct options optmem;
1890                            unsigned long route_src;
1891                struct device *dev=NULL;

1892                /*
1893                 *  Check the arguments
1894                 */

1895                err=verify_area(VERIFY_READ, optval, sizeof(mreq));
1896                if(err)
1897                    return err;

1898                memcpy_fromfs(&mreq,optval,sizeof(mreq));

1899                /*
1900                 *  Get device for use later
```

```
1901                 */

1902                if(mreq.imr_interface.s_addr==INADDR_ANY)
1903                {
1904                    if((rt=ip_rt_route(mreq.imr_multiaddr.s_addr,&optmem,
&route_src))!=NULL)
1905                        {
1906                        dev=rt->rt_dev;
1907                        rt->rt_use--;
1908                    }
1909                }
1910                else
1911                {
1912                    for(dev = dev_base; dev; dev = dev->next)
1913                    {
1914                        if((dev->flags&IFF_UP)&&
(dev->flags&IFF_MULTICAST)&&
1915                                (dev->pa_addr==mreq.imr_interface.s_addr))
1916                            break;
1917                    }
1918                }

1919                /*
1920                 *  Did we find a suitable device.
1921                 */

1922                if(!dev)
1923                    return -ENODEV;

1924                /*
1925                 *  Leave group
1926                 */

1927                return ip_mc_leave_group(sk,dev,mreq.imr_multiaddr.s_addr);
1928            }
1929    #endif
```

1773-1929是对多播地址进行添加和删除操作，以及对多播相关参数进行设置。这段代码中涉及到的所有函数在本书前文中都已作分析，ip_mreq结构定义在include/linux/in.h中，为便于理解，重新给出如下：

```
/*include/linux/in.h*/
38  /* Request struct for multicast socket ops */
39  struct ip_mreq
40  {
```

```
41      struct in_addr imr_multiaddr;   /* IP multicast address of group */
42      struct in_addr imr_interface;   /* local IP address of interface */
43  };
```

具体其他代码留作读者自行分析。

```
1930    #ifdef CONFIG_IP_FIREWALL
1931            case IP_FW_ADD_BLK:
1932            case IP_FW_DEL_BLK:
1933            case IP_FW_ADD_FWD:
1934            case IP_FW_DEL_FWD:
1935            case IP_FW_CHK_BLK:
1936            case IP_FW_CHK_FWD:
1937            case IP_FW_FLUSH_BLK:
1938            case IP_FW_FLUSH_FWD:
1939            case IP_FW_ZERO_BLK:
1940            case IP_FW_ZERO_FWD:
1941            case IP_FW_POLICY_BLK:
1942            case IP_FW_POLICY_FWD:
1943                if(!suser())
1944                    return -EPERM;
1945                if(optlen>sizeof(tmp_fw) || optlen<1)
1946                    return -EINVAL;
1947                err=verify_area(VERIFY_READ,optval,optlen);
1948                if(err)
1949                    return err;
1950                memcpy_fromfs(&tmp_fw,optval,optlen);
1951                err=ip_fw_ctl(optname, &tmp_fw,optlen);
1952                return -err;    /* -0 is 0 after all */

1953    #endif
1954    #ifdef CONFIG_IP_ACCT
1955            case IP_ACCT_DEL:
1956            case IP_ACCT_ADD:
1957            case IP_ACCT_FLUSH:
1958            case IP_ACCT_ZERO:
1959                if(!suser())
1960                    return -EPERM;
1961                if(optlen>sizeof(tmp_fw) || optlen<1)
1962                    return -EINVAL;
1963                err=verify_area(VERIFY_READ,optval,optlen);
1964                if(err)
1965                    return err;
1966                memcpy_fromfs(&tmp_fw, optval,optlen);
```

```
1967                err=ip_acct_ctl(optname, &tmp_fw,optlen);
1968                return -err;    /* -0 is 0 after all */
1969    #endif
1970            /* IP_OPTIONS and friends go here eventually */
1971            default:
1972                return(-ENOPROTOOPT);
1973        }
1974    }
```

ip_setsockopt函数最后两段代码分别对防火墙规则和信息统计时匹配规则进行设置，涉及到两个新的函数：ip_fw_ctl，ip_acct_ctl。注意两个函数第一个参数为具体需要设置的选项，第二个参数为选项值，第三个参数为选项长度。在下文中分析ip_fw.c文件再进行详细说明。

```
1975    /*
1976     *  Get the options. Note for future reference. The GET of IP options gets
the
1977     *  _received_ ones. The set sets the _sent_ ones.
1978     */

1979    int ip_getsockopt(struct sock *sk, int level, int optname,
                          char *optval, int *optlen)
1980    {
```

相比较ip_setsockopt函数，ip_getsockopt函数实现代码较短，该函数用于IP选项值的获取。

```
1981        int val,err;
1982    #ifdef CONFIG_IP_MULTICAST
1983        int len;
1984    #endif

1985        if(level!=SOL_IP)
1986            return -EOPNOTSUPP;

1987        switch(optname)
1988        {
1989            case IP_TOS:
1990                val=sk->ip_tos;
1991                break;
1992            case IP_TTL:
1993                val=sk->ip_ttl;
1994                break;
```

```
1995    #ifdef CONFIG_IP_MULTICAST
1996            case IP_MULTICAST_TTL:
1997                val=sk->ip_mc_ttl;
1998                break;
1999            case IP_MULTICAST_LOOP:
2000                val=sk->ip_mc_loop;
2001                break;
2002            case IP_MULTICAST_IF:
2003                err=verify_area(VERIFY_WRITE, optlen, sizeof(int));
2004                if(err)
2005                    return err;
2006                len=strlen(sk->ip_mc_name);
2007                err=verify_area(VERIFY_WRITE, optval, len);
2008                if(err)
2009                    return err;
2010                put_fs_long(len,(unsigned long *) optlen);
2011                memcpy_tofs((void *)optval,sk->ip_mc_name, len);
2012                return 0;
2013    #endif
2014            default:
2015                return(-ENOPROTOOPT);
2016        }
2017        err=verify_area(VERIFY_WRITE, optlen, sizeof(int));
2018        if(err)
2019            return err;
2020        put_fs_long(sizeof(int),(unsigned long *) optlen);

2021        err=verify_area(VERIFY_WRITE, optval, sizeof(int));
2022        if(err)
2023            return err;
2024        put_fs_long(val,(unsigned long *)optval);

2025        return(0);
2026    }
```

函数实现比较简单，无需要分析之处，读者自行理解。1999行对sock结构中ip_mc_loop字段值进行获取以及在ip_setsockopt函数中对该字段的设置，仅仅在值的传递，换句话说，本版本实现并未使用到该字段。具体功能不详。

```
2027    /*
2028     *  IP protocol layer initialiser
2029     */

2030    static struct packet_type ip_packet_type =
```

```
2031    {
2032        0,  /* MUTTER ntohs(ETH_P_IP),*/
2033        NULL,   /* All devices */
2034        ip_rcv,
2035        NULL,
2036        NULL,
2037    };
```

2030行定义IP协议对应的packet_type结构，基于链路层模块的网络层每个协议都对应有一个packet_type结构，packet_type结构定义在include/linux/netdevice.h中，如下：

```
/*include/linux/netdevice.h*/
139 struct packet_type {
140   unsigned short    type;   /* This is really htons(ether_type). */
141   struct device *   dev;
142   int           (*func) (struct sk_buff *, struct device *,
143                  struct packet_type *);
144   void          *data;
145   struct packet_type    *next;
146 };
```

对照packet_type结构定义，IP协议将接收函数设置为ip_rcv，这是IP协议数据包接收的总入口函数；type字段暂时设置为0，在下面的ip_init函数注册ip_packet_type时，该字段将被正确设置为ETH_P_IP（0x0800）。所有网络层协议对应的packet_type结构构成一个队列，ptype_base变量指向这个队列的首部，链路层模块在处理完一个数据包准备上传至网络层处理时，将遍历该队列，根据链路层协议首部中类型字段查找合适的packet_type结构，调用该结构中func指向的函数，将数据包从链路层传递给网络层进行处理。

```
2038    /*
2039     *  Device notifier
2040     */

2041    static int ip_rt_event(unsigned long event, void *ptr)
2042    {
2043        if(event==NETDEV_DOWN)
2044            ip_rt_flush(ptr);
2045        return NOTIFY_DONE;
2046    }

2047    struct notifier_block ip_rt_notifier={
2048        ip_rt_event,
2049        NULL,
2050        0
2051    };
```

```
2052    /*
2053     *  IP registers the packet type and then calls the subprotocol initialisers
2054     */

2055    void ip_init(void)
2056    {
2057        ip_packet_type.type=htons(ETH_P_IP);
2058        dev_add_pack(&ip_packet_type);

2059        /* So we flush routes when a device is downed */
2060        register_netdevice_notifier(&ip_rt_notifier);
2061    /*  ip_raw_init();
2062        ip_packet_init();
2063        ip_tcp_init();
2064        ip_udp_init();*/
2065    }
```

2041-2065行代码在前文中都已进行分析，此处为保证ip.c文件的完整性，列出这些代码如上。

ip.c文件小结

ip.c文件是IP协议实现文件，IP协议作为网络层协议，是最为广泛使用的协议。从整个实现结构来看，可以分别以ip_rcv和ip_queue_xmit函数为起点进行分析。ip_rcv函数作为IP协议数据包接收的总入口函数，起着数据包中转和派发的作用，其根据数据包的具体类型，调用IP协议模块中的其他相关函数进行处理；在完成网络层的处理后，调用传输层协议入口函数将数据包传递给传输层处理。从实现的功能来看，IP协议实现模块主要完成如下工作：

1> 数据包过滤，即防火墙功能实现在IP协议模块中。

2> 数据包转发实现在IP协议模块中。

3> 数据包分片也由IP协议模块负责。

4> 最后最为链路层和传输层之间的层次协议，起着承上启下的数据包传递作用。

2.22 net/inet/ip_fw.c文件

在介绍ip.c文件时，多次提到数据包过滤，即防火墙功能的实现。另外在调用ip_setsockopt函数时也有两个未进行分析的函数：ip_fw_ctl，ip_acct_ctl。防火墙功能实现文件为ip_fw.c，下面我们即对该文件进行分析。不过首先我们需要熟悉一下相关数据结构，虽然这些结构在本书第一章介绍ip_fw.h文件时已有论述，为了便于下文的分析，再次列出如下：

```
/*include/linux/ip_fw.h*/
41  struct ip_fw
42  {
43      struct ip_fw  *fw_next;          /* Next firewall on chain */
```

fw_next字段用于ip_fw结构组成队列。

```
44      struct in_addr fw_src, fw_dst;       /* Source and destination IP addr */
45      struct in_addr fw_smsk, fw_dmsk;     /* Mask for src and dest IP addr */
```

fw_src,fw_dst字段为防火墙规则中设置的源端，目的端IP地址。而fw_smsk,fw_dmsk字段为源端和目的端IP地址掩码。

```
46      struct in_addr fw_via;           /* IP address of interface "via" */
```

fw_via字段表示数据包接收或者发送设备对应的IP地址。

```
47      unsigned short fw_flg;           /* Flags word */
```

fw_flg表示规则对应的数据包处理策略以及如下端口号匹配规则(范围匹配或者精确匹配)。

```
48      unsigned short fw_nsp, fw_ndp;   /* N'of src ports and # of dst ports */
49                          /* in ports array (dst ports follow */
50                              /* src ports; max of 10 ports in all; */
51                              /* count of 0 means match all ports) */
52  #define IP_FW_MAX_PORTS 10              /* A reasonable maximum */
53      unsigned short fw_pts[IP_FW_MAX_PORTS]; /* Array of port numbers to match
*/
```

fw_nsp, fw_ndp, fw_pts构成该规则使用的匹配端口号；fw_nsp表示源端口号数量，fw_ndp表示目的端口号数量；所有端口号均存储在fw_pts数组中，该数组前fw_nsp个元素表示源端口号，紧接着的fw_ndp个元素表示目的端口号。端口号表示可以使用范围表示法（fw_pts数组中一对元素表示端口范围），也可以每个元素对应单个端口号。

```
54      unsigned long  fw_pcnt,fw_bcnt;      /* Packet and byte counters */
```

fw_pcnt, fw_bcnt对匹配该规则的数据包进行统计，其中fw_pcnt以数据包个数为单位进行痛惜，而fw_bcnt以字节数为单位进行匹配数据包的统计。

```
55  };
```

ip_fw.h 文件中还定义有很多常量，此处不再列出，读者可查阅第一章中 ip_fw.h 文件内容。下面我们即进入对防火墙实现文件 ip_fw.c 的分析。

```
1   /*
2    *  IP firewalling code. This is taken from 4.4BSD. Please note the
3    *  copyright message below. As per the GPL it must be maintained
4    *  and the licenses thus do not conflict. While this port is subject
5    *  to the GPL I also place my modifications under the original
6    *  license in recognition of the original copyright.
7    *                  -- Alan Cox.
8    *
9    *  Ported from BSD to Linux,
10   *      Alan Cox 22/Nov/1994.
11   *  Zeroing /proc and other additions
12   *      Jos Vos 4/Feb/1995.
13   *  Merged and included the FreeBSD-Current changes at Ugen's request
14   *  (but hey it's a lot cleaner now). Ugen would prefer in some ways
15   *  we waited for his final product but since Linux 1.2.0 is about to
16   *  appear it's not practical - Read: It works, it's not clean but please
17   *  don't consider it to be his standard of finished work.
18   *      Alan Cox 12/Feb/1995
19   *  Porting bidirectional entries from BSD, fixing accounting issues,
20   *  adding struct ip_fwpkt for checking packets with interface address
21   *      Jos Vos 5/Mar/1995.
22   *
23   *  All the real work was done by .....
24   */

25  /*
26   * Copyright (c) 1993 Daniel Boulet
27   * Copyright (c) 1994 Ugen J.S.Antsilevich
28   *
29   * Redistribution and use in source forms, with and without modification,
30   * are permitted provided that this entire comment appears intact.
31   *
32   * Redistribution in binary form may occur without any restrictions.
33   * Obviously, it would be nice if you gave credit where credit is due
34   * but requiring it would be too onerous.
35   *
36   * This software is provided ``AS IS'' without any warranties of any kind.
37   */

38  #include <linux/config.h>
39  #include <asm/segment.h>
40  #include <asm/system.h>
41  #include <linux/types.h>
```

```
42  #include <linux/kernel.h>
43  #include <linux/sched.h>
44  #include <linux/string.h>
45  #include <linux/errno.h>
46  #include <linux/config.h>

47  #include <linux/socket.h>
48  #include <linux/sockios.h>
49  #include <linux/in.h>
50  #include <linux/inet.h>
51  #include <linux/netdevice.h>
52  #include <linux/icmp.h>
53  #include <linux/udp.h>
54  #include "ip.h"
55  #include "protocol.h"
56  #include "route.h"
57  #include "tcp.h"
58  #include <linux/skbuff.h>
59  #include "sock.h"
60  #include "icmp.h"
61  #include <linux/ip_fw.h>

62  /*
63   *    Implement IP packet firewall
64   */

65  #ifdef CONFIG_IPFIREWALL_DEBUG
66  #define dprintf1(a)         printk(a)
67  #define dprintf2(a1,a2)         printk(a1,a2)
68  #define dprintf3(a1,a2,a3)      printk(a1,a2,a3)
69  #define dprintf4(a1,a2,a3,a4) printk(a1,a2,a3,a4)
70  #else
71  #define dprintf1(a)
72  #define dprintf2(a1,a2)
73  #define dprintf3(a1,a2,a3)
74  #define dprintf4(a1,a2,a3,a4)
75  #endif
```

我们跳过前面文件说明部分和头文件声明部分，65-75 行代码用于代码调试，如果 CONFIG_IPFIREWALL_DEBUG 被定义，则打印调试信息，这一般用在代码编写过程中。

```
76  #define print_ip(a)    printf("%d.%d.%d.%d",(ntohl(a.s_addr)>>24)&0xFF,\
77                                  (ntohl(a.s_addr)>>16)&0xFF,\
```

```
78                                      (ntohl(a.s_addr)>>8)&0xFF,\
79                                      (ntohl(a.s_addr))&0xFF);

80  #ifdef IPFIREWALL_DEBUG
81  #define dprint_ip(a)        print_ip(a)
82  #else
83  #define dprint_ip(a)
84  #endif
```

print_ip 使用点分十进制方式打印 IP 地址。80-84 行代码同样用于方便调试。

```
85  #ifdef CONFIG_IP_FIREWALL
86  struct ip_fw *ip_fw_fwd_chain;
87  struct ip_fw *ip_fw_blk_chain;
88  int ip_fw_blk_policy=IP_FW_F_ACCEPT;
89  int ip_fw_fwd_policy=IP_FW_F_ACCEPT;
90  #endif
91  #ifdef CONFIG_IP_ACCT
92  struct ip_fw *ip_acct_chain;
93  #endif
```

85-93 行定义有三个规则链：数据包转发规则链-ip_fw_fwd_chain; 本地规则链-ip_fw_blk_chian;信息统计规则链-ip_acct_chain。注意这些规则链的包含均需要在定义相关宏的情况下，即防火墙实现在当前内核版本中还是一个可选组件。如果没有进行相关宏的定义，则不会包含对应防火墙功能实现代码。

```
94  #define IP_INFO_BLK  0
95  #define IP_INFO_FWD 1
96  #define IP_INFO_ACCT       2
```

94-96 行定义的常量用于匹配规则链。这在下文中相关代码处将有说明。

```
97  /*
98   *    Returns 1 if the port is matched by the vector, 0 otherwise
99   */

100 extern inline int port_match(unsigned short *portptr, int nports,
                                        unsigned short port, int range_flag)
101 {
102         if (!nports)
103                 return 1;
104         if ( range_flag )
105         {
106                 if ( portptr[0] <= port && port <= portptr[1] )
```

```
107             {
108                     return( 1 );
109             }
110             nports -= 2;
111             portptr += 2;
112         }
113         while ( nports-- > 0 )
114         {
115             if ( *portptr++ == port )
116             {
117                     return( 1 );
118             }
119         }
120         return(0);
121 }
```

port_match 函数用于端口匹配，匹配方式有两种：其一为范围匹配，即端口号在规则规定的某一范围内时则认为匹配；其二为精确匹配，即端口号是否为规则明确指定的端口。参数 range_flag 决定匹配方式，如果该参数为 1，则使用范围匹配方式，此时 portptr 指向的端口数组中，每两个构成一对，表示端口范围；如果 range_flag 为 0，则表示精确匹配，即遍历 portptr 指向的数组中的每个端口，一一进行检查，只有与 port 参数表示的端口完全相等时，才认为匹配。port_match 函数实现即根据 range_flag 参数的设置采用不同的匹配方式。函数返回 1 表示发现匹配项，返回 0 表示没有发现匹配项。

```
122 #if defined(CONFIG_IP_ACCT) || defined(CONFIG_IP_FIREWALL)

123 /*
124  *    Returns 0 if packet should be dropped, 1 if it should be accepted,
125  *    and -1 if an ICMP host unreachable packet should be sent.
126  *    Also does accounting so you can feed it the accounting chain.
127  *    If opt is set to 1, it means that we do this for accounting
128  *    purposes (searches all entries and handles fragments different).
129  *    If opt is set to 2, it doesn't count a matching packet, which
130  *    is used when calling this for checking purposes (IP_FW_CHK_*).
131  */



132 int ip_fw_chk(struct iphdr *ip, struct device *rif, struct ip_fw *chain, int policy, int opt)
133 {
```

ip_fw_chk 函数在 IP 协议实现模块中被多次调用对数据包进行过滤操作。如 ip_forward 数据包转发函数，ip_rcv 数据包接收函数，以及 ip_queue_xmit 数据包发送函数。
参数说明：
ip：被检查数据包 IP 首部，这个首部中字段用于规则匹配。

rif：数据包接收或者发送的网络设备。
chain：匹配规则链，如 ip_forward 函数调用时使用 ip_fw_fwd_chain 转发链，而 ip_rcv, ip_queue_xmit 函数调用时使用的 ip_fw_blk_chain 本地规则链。
policy：该参数为规则的默认使用策略，在发现一个匹配规则，但该规则没有定义数据包处理策略时，就使用该默认策略。
opt：该参数表示调用 ip_fw_chain 函数的目的，正如函数前的说明文字所述，如果 opt 设置为 1，则表示仅仅进行信息统计；设置为 2，则表示仅仅进行检查，既不进行信息统计，也不对数据包进行任何丢弃处理。

在进行代码的具体分析之前，还是要对防火墙实现代码分析上的为难情绪进行一番“安慰”。防火墙实现有些类似于路有表实现，虽然二者功能完全不同，但实现方式却是如出一辙。在本书前文中对路由表实现文件 route.c 进行分析，我们说到路由表其实就是一个队列，队列中每个元素表示一个路由表项。而防火墙实现也是由一个队列构成，队列中每个元素表示一个规则。每个路由表表项由一个 rtable 结构表示，每个防火墙规则由一个 ip_fw 结构表示；在进行路由表查询时，系统遍历由 rt_base 全局变量指向 rtable 结构构成的路由表项队列，以最终目的 IP 地址为关键字进行路由表项的匹配；对于防火墙规则，不同于路由表只有一个队列（rt_loopback 变量指向的队列只有一个元素，专门用于本地路由，此处我们不对此进行考虑），防火墙有多个规则链：
1〉数据包转发规则链：专门用于数据包转发时的过滤，相应调用环境为 ip_forward。
2〉本地数据包规则链：即对发往本地或者从本地发出的数据包进行过滤，相应调用环境为 ip_queue_xmit，ip_rcv。
3〉信息统计规则链：对分门别类的对数据包进行统计。

注意一个数据包可能经过多个规则链的过滤，如一个数据包先后经过 ip_rcv, ip_forward 函数处理，将经过以上所有三个规则链进行过滤。本文中涉及到防火墙时，多次使用“过滤”，“过滤”的含义是指对数据包进行规则匹配，在发现匹配项后，根据规则对应的策略对数据包进行处理（丢弃或者通过），如果没有任何匹配项，结果可能为丢弃或者通过，这取决于默认策略设置，所以“过滤”的含义更多为检查，而非指丢弃。

承接刚才的比较，在进行数据包过滤时，也是遍历某个规则链，对其中每个元素表示的规则进行匹配，所谓匹配就是进行相应字段的比较（这也是 IP 首部作为参数传入的目的所在！），如果比较结果满足匹配条件，就表示发现一个匹配规则，此时就根据该规则对应的数据包处理策略对数据包进行相应的处理（丢弃或者通过）。上文已经提到，每个防火墙规则由一个 ip_fw 结构表示，ip_fw 结构中定义有源，目的 IP 地址，端口号，这些都作为匹配规则的一部分进行比较。另外需要提请注意的是，数据包过滤分为两步：其一为规则匹配；其二为匹配后的数据包处理策略。所以规则匹配本身不是目的，当一个数据包没有发现一个匹配规则项时，其“命运”与发现一个匹配规则项时是同样的。即匹配本身并无代表任何意义，关键是匹配后对数据包的处理方式，即规则链定义时使用的数据包默认处理策略。

在理解了防火墙规则匹配无非就是一些字段的比较后，下面的代码就变的很容易，当然这种说法有些狂妄，或者说，为了使得“防火墙”这个单词名副其实，在比较代码的编写上人们都极尽复杂之能事，能不让人看懂最好就那样编。当然对于早期网络实现代码，无论其如何“做作”，终究还是比较“朴实”，分析起来相对而言还比较不费事（但是有些代码实现上，没有任何道理可讲，它就是这样实现的），现在的防火墙实现代码则是真正达到了“能不让

人看懂，就一定不让人看懂”的目的。

```
134     struct ip_fw *f;
135     struct tcphdr       *tcp=(struct tcphdr *)((unsigned long *)ip+ip->ihl);
136     struct udphdr       *udp=(struct udphdr *)((unsigned long *)ip+ip->ihl);
```

参数 ip 指向 IP 首部，其后跟随着传输层协议首部，由于目前尚不知晓传输层协议为何，135，136 行代码作了两手准备。因为防火墙规则中处理 IP 地址比较外，还需要进行端口号的比较，所以必须使用传输层协议首部中的字段。

```
137     __u32               src, dst;
138     __u16               src_port=0, dst_port=0;
139     unsigned short      f_prt=0, prt;
140     char            notcpsyn=1, frag1, match;
141     unsigned short      f_flag;

142     /*
143      *  If the chain is empty follow policy. The BSD one
144      *  accepts anything giving you a time window while
145      *  flushing and rebuilding the tables.
146      */

147     src = ip->saddr;
148     dst = ip->daddr;

149     /*
150      *  This way we handle fragmented packets.
151      *  we ignore all fragments but the first one
152      *  so the whole packet can't be reassembled.
153      *  This way we relay on the full info which
154      *  stored only in first packet.
155      *
156      *  Note that this theoretically allows partial packet
157      *  spoofing. Not very dangerous but paranoid people may
158      *  wish to play with this. It also allows the so called
159      *  "fragment bomb" denial of service attack on some types
160      *  of system.
161      */

162     frag1 = ((ntohs(ip->frag_off) & IP_OFFSET) == 0);
163     if (!frag1 && (opt != 1) && (ip->protocol == IPPROTO_TCP ||
164             ip->protocol == IPPROTO_UDP))
165         return(1);
```

162 行代码检查 IP 首部中的偏移字段是否为 0，如果为 0，则有两种可能：
1〉这是一个分片数据包，而且是第一个分片。
2〉这是普通数据包，不存在分片。
如果偏移字段不为 0，即 frag1 为 FALSE，则表示当前被过滤数据包一定是一个分片数据包，而且非第一个分片。在前文对函数参数说明中有：如果 opt 为 1，则表示对 ip_fw_chk 函数的调用并非进行数据包过滤，而是仅仅用于信息统计，如果 opt 不为 1，则表示是对数据包进行过滤（将 opt 设置为 2 时的数据包检查包括在内）。163 行表达的意思是，数据包过滤指对第一个分片和普通数据包进行。对于分片数据包属于中间分片和最后分片的情况不进行数据包过滤操作，当然 if 语句中的最后一项将协议限制为 TCP，UDP。为何 ICMP 协议无需检查，因为使用 ICMP 协议的数据包不会产生分片。

```
166     src = ip->saddr;
167     dst = ip->daddr;
```

如果我们没有记错的话，147-148 行已经进行了类似的操作，也许此代码编写者在编写该函数已经是“深夜了”，当然“很累了”，这一点我们需要谅解。

```
168     /*
169      *  If we got interface from which packet came
170      *  we can use the address directly. This is unlike
171      *  4.4BSD derived systems that have an address chain
172      *  per device. We have a device per address with dummy
173      *  devices instead.
174      */
```

下面对传输层使用协议进行判断，选择正确传输层首部格式进行相关变量的初始化操作，为此后的规则匹配做好准备。相关变量的使用在后续代码分析中进行说明，此处不再对 175-207 行代码一一进行说明。唯一需要注意的地方是对 frag1 变量的检查，frag1 为 TRUE，表示是一个分片，只有第一个分片中才包含传输层首部，即才有我们需要的端口号，所以在进行端口号索取前必须对此进行判断。

```
175     dprintf1("Packet ");
176     switch(ip->protocol)
177     {
178         case IPPROTO_TCP:
179             dprintf1("TCP ");
180             /* ports stay 0 if it is not the first fragment */
181             if (frag1) {
182                 src_port=ntohs(tcp->source);
183                 dst_port=ntohs(tcp->dest);
184                 if(tcp->syn && !tcp->ack)
185                     /* We *DO* have SYN, value FALSE */
186                     notcpsyn=0;
```

```
187                 }
188                 prt=IP_FW_F_TCP;
189                 break;
190             case IPPROTO_UDP:
191                 dprintf1("UDP ");
192                 /* ports stay 0 if it is not the first fragment */
193                 if (frag1) {
194                     src_port=ntohs(udp->source);
195                     dst_port=ntohs(udp->dest);
196                 }
197                 prt=IP_FW_F_UDP;
198                 break;
199             case IPPROTO_ICMP:
200                 dprintf2("ICMP:%d ",((char *)portptr)[0]&0xff);
201                 prt=IP_FW_F_ICMP;
202                 break;
203             default:
204                 dprintf2("p=%d ",ip->protocol);
205                 prt=IP_FW_F_ALL;
206                 break;
207         }
208         dprint_ip(ip->saddr);
```

变量 prt 表示对何种协议规则进行匹配，防火墙中设置的每个规则都有其对应的协议号，只有协议号相符时才进行规则匹配操作。

```
209         if (ip->protocol==IPPROTO_TCP || ip->protocol==IPPROTO_UDP)
210             /* This will print 0 when it is not the first fragment! */
211             dprintf2(":%d ", src_port);
212         dprint_ip(ip->daddr);
213         if (ip->protocol==IPPROTO_TCP || ip->protocol==IPPROTO_UDP)
214             /* This will print 0 when it is not the first fragment! */
215             dprintf2(":%d ",dst_port);
216         dprintf1("\n");
```

209-216 用于代码调试，打印 IP 地址和端口号信息，我们对此无需关心。

```
217         for (f=chain;f;f=f->fw_next)
218         {
```

下面即进入到具体的规则匹配过程，操作上相当简单，遍历以参数形式传入的规则链，对其中每一个规则进行检查，查找匹配项。每个规则都由一个 ip_fw 结构表示，寻找匹配项的过程也就是相关字段进行比较的过程。

```
219             /*
220              *  This is a bit simpler as we don't have to walk
221              *  an interface chain as you do in BSD - same logic
222              *  however.
223              */

224             /*
225              *  Match can become 0x01 (a "normal" match was found),
226              *  0x02 (a reverse match was found), and 0x03 (the
227              *  IP addresses match in both directions).
228              *  Now we know in which direction(s) we should look
229              *  for a match for the TCP/UDP ports.    Both directions
230              *  might match (e.g., when both addresses are on the
231              *  same network for which an address/mask is given), but
232              *  the ports might only match in one direction.
233              *  This was obviously wrong in the original BSD code.
234              */
235             match = 0x00;
```

变量 match 表示匹配的类型，诚如 224-227 行注释所述，match 为 0x01 表示一个正常的匹配项；0x02 表示一个逆向匹配项（逆向的含义由下文具体代码实现给出）；0x03 表示 IP 地址在两个方向上都匹配（具体含义见下面代码实现）。

```
236             if ((src&f->fw_smsk.s_addr)==f->fw_src.s_addr
237             &&    (dst&f->fw_dmsk.s_addr)==f->fw_dst.s_addr)
238                     /* normal direction */
239                     match |= 0x01;
```

236 行进行源，目的 IP 地址的比较，比较方式采用 IP 地址掩码方式，为何要这样做，没有原因，代码就是这样编的，所造成的影响只是在设置防火墙规则时考虑到这一点就可以了，当然此处也可以直接进行地址的比较，不要使用地址掩码方式。这种方式称为正常匹配。下面我们看逆向匹配的含义，我们无需解释，读者应该明白逆向的意思了。

```
240             if ((f->fw_flg & IP_FW_F_BIDIR) &&
241                     (dst&f->fw_smsk.s_addr)==f->fw_src.s_addr
242             &&    (src&f->fw_dmsk.s_addr)==f->fw_dst.s_addr)
243                     /* reverse direction */
244                     match |= 0x02;
```

如果防火墙规则设置为双向的，表示除了源端和源端 IP 地址进行比较，目的地址和目的地址进行比较外，还可以进行源端地址和目的端地址进行比较，这种方式称为逆向比较，如果比较得出二者相等，则表示逆向匹配。另外所谓双向，就表示在地址比较上可以进行逆向比较。

```
245             if (match)
246             {
```

如果 IP 地址匹配成功，继续对其他条件进行匹配。250 行是对接收设备 IP 地址进行比较。由此我们可以得出 ip_fw 结构中 fw_via 字段表示接收设备或者发送设备的 IP 地址，参数 rif 表示数据包接收或者发送设备，此时进行二者之间的 IP 地址直接比较，如果相同，表示匹配成功，可以继续对其他条件（端口号）进行匹配，否则对下一个规则进行检查（253 行）。当然如果连 IP 地址都不匹配，则在 260 行就进行下一规则的匹配。

```
247                 /*
248                  *    Look for a VIA match
249                  */
250                 if(f->fw_via.s_addr && rif)
251                 {
252                         if(rif->pa_addr!=f->fw_via.s_addr)
253                                 continue; /* Mismatch */
254                 }
255                 /*
256                  *    Drop through - this is a match
257                  */
258             }
259             else
260                     continue;

261             /*
262              *    Ok the chain addresses match.
263              */
```

执行到此处，表示 IP 地址匹配，网络设备 IP 地址也匹配。变量 f 表示以上条件匹配的规则项，264 行检查该规则使用的协议范围。如果该规则是对特定协议（或者数据包类型）适用（265 行），则需要对特定的协议（或者数据包类型）进行检查。

```
264             f_prt=f->fw_flg&IP_FW_F_KIND;
265             if (f_prt!=IP_FW_F_ALL)
266             {
```

变量 f_prt 不等于 IP_FW_F_ALL,就表示匹配的规则适用于特定的协议或者数据包类型，而非针对所有协议和数据包。

```
267                 /*
268                  * This is actually buggy as if you set SYN flag
269                  * on UDP or ICMP firewall it will never work,but
270                  * actually it is a concern of software which sets
271                  * firewall entries.
```

```
272                 */

273                 if((f->fw_flg&IP_FW_F_TCPSYN) && notcpsyn)
274                     continue;
```

273 行对 SYN 数据包类型规则进行检查，如果该规则只适用于 SYN 数据包，则如果当前数据包不是一个 SYN 数据包（notcpsyn 为 1），则表示该规则对当前数据包不适用（虽然多个 IP 地址已经匹配）。

```
275                 /*
276                  *  Specific firewall - packet's protocol
277                  *  must match firewall's.
278                  */

279                 if(prt!=f_prt)
280                     continue;
```

变量 prt 在函数前面代码中被初始化为当前数据包所使用传输层协议，如果当前规则适用协议和当前数据包适用协议不同，则也表示当前规则不适用于该数据包，此时也跳过该规则。否则就表示当前规则适用于当前被检查数据包，终于可以进行端口号的检查了。

```
281                 if(!(prt==IP_FW_F_ICMP || ((match & 0x01) &&
282                     port_match(&f->fw_pts[0], f->fw_nsp, src_port,
283                         f->fw_flg&IP_FW_F_SRNG) &&
284                     port_match(&f->fw_pts[f->fw_nsp], f->fw_ndp, dst_port,
285                         f->fw_flg&IP_FW_F_DRNG)) || ((match & 0x02) &&
286                     port_match(&f->fw_pts[0], f->fw_nsp, dst_port,
287                         f->fw_flg&IP_FW_F_SRNG) &&
288                     port_match(&f->fw_pts[f->fw_nsp], f->fw_ndp, src_port,
289                         f->fw_flg&IP_FW_F_DRNG))))
290                 {
291                     continue;
292                 }
```

281 行 if 语句真是够复杂的，我们将其分解，首先有：

！（a||b||c）=（!a）&& (!b) && (!c) 或者 $\overline{A + B + C} = \overline{A} \cdot \overline{B} \cdot \overline{C}$

其中：

a：prt==IP_FW_F_ICMP

b：(match & 0x01) &&
port_match(&f->fw_pts[0], f->fw_nsp, src_port,f->fw_flg&IP_FW_F_SRNG) &&
port_match(&f->fw_pts[f->fw_nsp], f->fw_ndp, dst_port,f->fw_flg&IP_FW_F_DRNG)

c：(match & 0x02) &&
port_match(&f->fw_pts[0], f->fw_nsp, dst_port,f->fw_flg&IP_FW_F_SRNG) &&
port_match(&f->fw_pts[f->fw_nsp], f->fw_ndp, src_port,f->fw_flg&IP_FW_F_DRNG)
要使得 291 行代码被执行，则必须同时有!a=TRUE, !b=TRUE, !c=TRUE。
下面我们逐一分析这个条件：
1>!a=TRUE，即 a=FALSE。这表示可以对端口进行匹配检查，因为 ICMP 协议没有端口号比较这一说。换句话说，如果使用的是 ICMP 协议，则无需进行端口号匹配检查，只要通过以上 IP 地址检查，即表示发现一个匹配项。
2>!b=TRUE，即 b=FALSE。首先对正常 IP 地址匹配方式进行端口号匹配检查， 第一个 port_match 函数对源端端口号进行检查，ip_fw 结构中 fw_pts 数组中存储具体端口号，而 fw_nsp 字段表示源端口号的数量，而 fw_ndp 表示目的端口号数量。源端口号和目的端口号使用同一个 fw_pts 数组，该数组前面 fw_nsp 个元素存储的是源端口号，其后 fw_ndp 个元素为目的端口号; fw_flg 标志字段表示端口匹配方式，如果设置了 IP_FW_F_SRNG 标志位，则端口匹配使用范围匹配方式，此时 fw_pts 数组中一对元素表示端口范围，否则采用精确端口匹配方式。函数 port_match 返回 1 表示匹配，返回 0 表示不匹配。所以 b=FALSE，表示端口号（源端口号或者目的端口号）不匹配。
3>!c=TRUE, 即 c=FALSE。该条件与 2>类似，只不过是对逆向 IP 地址匹配方式的检查。对于逆向 IP 地址匹配，端口号也采用逆向匹配方式；b=FALSE，表示逆向端口号不匹配。

综上所述，281 行 if 语句如果为真，则表示端口号不匹配，跳过当前规则（虽然很遗憾，因为 IP 地址好不容易通过了匹配检查）。

```
293             }// if (f_prt!=IP_FW_F_ALL)
```

可执行到此处，表示发现一个匹配规则项，现在我们就要根据该匹配规则对应的数据包处理策略对数据包的命运进行宣判了。不过在这之前，为了庆祝我们发现一个匹配项，让我们展示一下相关信息。

```
294  #ifdef CONFIG_IP_FIREWALL_VERBOSE
295             /*
296              * VERY ugly piece of code which actually
297              * makes kernel printf for denied packets...
298              */

299             if (f->fw_flg & IP_FW_F_PRN)
300             {
301                     if(opt != 1) {
302                             if(f->fw_flg&IP_FW_F_ACCEPT)
303                                     printk("Accept ");
304                             else if(f->fw_flg&IP_FW_F_ICMPRPL)
305                                     printk("Reject ");
306                             else
307                                     printk("Deny ");
308                     }
```

```
309                 switch(ip->protocol)
310                 {
311                     case IPPROTO_TCP:
312                         printk("TCP ");
313                         break;
314                     case IPPROTO_UDP:
315                         printk("UDP ");
316                     case IPPROTO_ICMP:
317                         printk("ICMP ");
318                         break;
319                     default:
320                         printk("p=%d ",ip->protocol);
321                         break;
322                 }
323                 print_ip(ip->saddr);
324                 if(ip->protocol == IPPROTO_TCP || ip->protocol == IPPROTO_UDP)
325                     printk(":%d", src_port);
326                 printk(" ");
327                 print_ip(ip->daddr);
328                 if(ip->protocol == IPPROTO_TCP || ip->protocol == IPPROTO_UDP)
329                     printk(":%d",dst_port);
330                 printk("\n");
331             }
332 #endif
```

294-332 行代码打印相关信息：301 行检查当前函数调用是否为信息统计，对于信息统计，我们不泄露规则对应的数据包处理策略，否则打印出匹配规则的数据包处理策略，以示“公正”（到时候数据包被丢弃了，不要说我们作弊）。309-322 行打印当前被处理数据包使用协议；323-330 行打印数据包源，目的 IP 地址和端口号。

```
333             if (opt != 2) {
334                 f->fw_bcnt+=ntohs(ip->tot_len);
335                 f->fw_pcnt++;
336             }
```

参数 opt 设置为 2，表示对规则设置进行检查，并不使用规则定义的策略处理数据包，也不更新相关规则统计信息；设置为 1，表示仅进行信息统计；否则使用相关策略处理数据包。

```
337             if (opt != 1)
338                 break;
339         } /* Loop */

340         if(opt == 1)
341             return 0;
```

对于信息统计的情况，不进行数据包处理，所以 341 行直接返回。否则进行下面的数据包命运裁决。

```
342         /*
343           * We rely on policy defined in the rejecting entry or, if no match
344           * was found, we rely on the general policy variable for this type
345           * of firewall.
346           */

347         if(f!=NULL)   /* A match was found */
348                 f_flag=f->fw_flg;
```

变量 f 不为 NULL，表示发现一个匹配规则，此时使用该规则配置的数据包处理策略对数据包进行处理，变量 f_flag 初始化为规则对应的数据包处理策略。

```
349         else
350                 f_flag=policy;
```

如果 f 为 NULL，表示没有发现匹配规则，此时使用默认数据包处理策略，policy 是作为参数参数传入的，该参数在 ip_fw_chk 函数被调用时根据不同的规则链设置为对应规则链默认的数据包处理策略。这些默认策略定义在 ip_fw.c 文件开始处，重新列出如下；

```
88    int ip_fw_blk_policy=IP_FW_F_ACCEPT;
89    int ip_fw_fwd_policy=IP_FW_F_ACCEPT;
```

对于当前设置，所有的规则链默认策略为通过防火墙，即数据包没有被防火墙丢弃，将继续数据包的其他方面的处理。

```
351         if(f_flag&IP_FW_F_ACCEPT)
352                 return 1;
353         if(f_flag&IP_FW_F_ICMPRPL)
354                 return -1;
355         return 0;
356   }
```

在 348 行或者 350 行完成 f_flag 变量的设置后，351-355 行根据该变量表示的处理策略返回对应值：

1〉 返回 1：通过防火墙检查，数据包继续其他方面的处理。

如 ip_rcv 函数相应调用环境如下：

```
/*net/inet/ip.c—ip_rcv 函数代码片断*/
1281        #ifdef      CONFIG_IP_FIREWALL
1282                if ((err=ip_fw_chk(iph,dev,ip_fw_blk_chain,ip_fw_blk_policy, 0))!=1)
1283                {
1284                        if(err==-1)
1285                                icmp_send(skb, ICMP_DEST_UNREACH,
```

```
                                        ICMP_PORT_UNREACH, 0, dev);
1286                kfree_skb(skb, FREE_WRITE);
1287                return 0;
1288            }
1289        #endif
```

2〉 返回-1，表示数据包被丢弃，但需要调用 ip_fw_chk 函数的其他函数（如 ip_rcv）回复一个 ICMP 错误通知报文。IP_FW_F_ICMPRPL 标志位的含义为 ICMP RePLy。
如上 ip_rcv 函数片断中 1284-1285 行代码即对应此种情况。

3〉 返回 0，数据包被丢弃，不返回任何信息给数据包发送端。

自此我们完成对数据包进行过滤的函数 ip_fw_chk 的分析，该函数完成的工作就是通常意义上我们所说的防火墙。其对数据包相关字段：IP 地址，端口号，接收设备进行规则匹配检查，其后根据规则定义的数据包处理策略对被检查的数据包进行处理。代码实现上理解的难点在于系统如何实现防火墙功能。实现方式在前文中我们与系统路由表的实现方式进行了比较。从本质上说，系统维护一个特定结构类型的队列，队列中每个元素表示一个规则或者表项，所谓匹配或者路由查询就是遍历该队列，对其中每个元素进行检查，寻找匹配项。理解了这一点，克服自己由于之前阅读有关资料对他们产生的畏难情绪的基础上，对于防火墙（以及路由表）的实现原理以及具体代码应该有充分的信心掌握，事实上，在经过以上的分析后，这些以往貌似复杂的东西其实很简单，之前的问题在于我们通常被一些他们自己弄不明白时就把问题说的很复杂的所谓专家们吓住了。

```
357  static void zero_fw_chain(struct ip_fw *chainptr)
358  {
359      struct ip_fw *ctmp=chainptr;
360      while(ctmp)
361      {
362          ctmp->fw_pcnt=0L;
363          ctmp->fw_bcnt=0L;
364          ctmp=ctmp->fw_next;
365      }
366  }
```

zero_fw_chain 函数对指定规则链中每个规则的统计信息字段进行清零操作。每个规则有一个 ip_fw 结构表示，该结构中定义有两个信息字段：fw_pcnt 表示当前匹配该规则的数据包总个数；fw_bcnt 表示当前匹配该规则的数据包中包含的总字节数。二者都表示相同的意义，只不过使用了不同的单位进行统计。

```
367  static void free_fw_chain(struct ip_fw *volatile* chainptr)
368  {
369      unsigned long flags;
370      save_flags(flags);
371      cli();
```

```
372         while ( *chainptr != NULL )
373         {
374                 struct ip_fw *ftmp;
375                 ftmp = *chainptr;
376                 *chainptr = ftmp->fw_next;
377                 kfree_s(ftmp,sizeof(*ftmp));
378         }
379         restore_flags(flags);
380 }
```

free_fw_chain 函数则是释放指定规则链中当前定义的所有规则。参数 chainptr 表示进行释放的规则链。为了不造成可能的内核不一致情况的发生，在进行释放之前调用 cli()函数禁止中断。

```
381 /* Volatiles to keep some of the compiler versions amused */

382 static int add_to_chain(struct ip_fw *volatile* chainptr, struct ip_fw *frwl)
383 {
```

函数 add_to_chain 用于向指定规则链中添加规则，参数 chianptr 表示规则链，frwl 表示具体要添加的新规则。这个函数的实现代码较长，主要代码集中在新规则添加位置的选择上花费了大量代码用于地址和端口号的比较。

```
384         struct ip_fw *ftmp;
385         struct ip_fw *chtmp=NULL;
386         struct ip_fw *volatile chtmp_prev=NULL;
387         unsigned long flags;
388         unsigned long m_src_mask,m_dst_mask;
389         unsigned long n_sa,n_da,o_sa,o_da,o_sm,o_dm,n_sm,n_dm;
390         unsigned short n_sr,n_dr,o_sr,o_dr;
391         unsigned short oldkind,newkind;
392         int addb4=0;
393         int n_o,n_n;

394         save_flags(flags);

395         ftmp = kmalloc( sizeof(struct ip_fw), GFP_ATOMIC );
396         if ( ftmp == NULL )
397         {
398 #ifdef DEBUG_CONFIG_IP_FIREWALL
399                 printf("ip_fw_ctl:    malloc said no\n");
400 #endif
401                 return( ENOMEM );
```

```
402     }
```

395 行重新分配一个 ip_fw 结构，因为作为参数传入的 ip_fw 结构在调用该函数的其他函数后可能被释放，不可直接实现参数表示的规则直接进行插入，另外对于新规则中某些字段需要重新进行初始化，故而需要重新创建一个 ip_fw 结构，将其插入到规则链中。

```
403     memcpy(ftmp, frwl, sizeof( struct ip_fw ) );

404     ftmp->fw_pcnt=0L;
405     ftmp->fw_bcnt=0L;

406     ftmp->fw_next = NULL;
```

403-406 行代码对新创建的 ip_fw 结构进行初始化。注意 404-405 对统计信息的清零操作，防止参数 frwl 中进行了不正确的设置。对于一个新添加的规则，这些字段应该进行清零。下面就进行新规则的插入，在插入之前必须寻找到合适的插入位置。408-533 行代码即进行插入位置的查找，最后将其加入到规则链中。由于需要对规则链进行操作，为了保持内核一致性，407 行调用 cli()函数禁止中断。

```
407     cli();

408     if (*chainptr==NULL)
409     {
410         *chainptr=ftmp;
411     }
```

408-411 对应当前规则链为空的情况，在此种情况下，无需进行寻找，直接将新的规则挂接到该规则链中即可。如果当前规则链非空，则进入如下 else 语句块，进行合适插入位置的查找，并将新的规则插入到该位置上。

```
412     else
413     {
414         chtmp_prev=NULL;
415         for (chtmp=*chainptr;chtmp!=NULL;chtmp=chtmp->fw_next)
416         {
```

415 行这个 for 语句遍历规则链对现有规则和新规则进行比较，查找新规则合适的插入位置。变量 addb4（add before：加入到之前）意为插入到当前现有的某个规则之前，如果进行一系列比较后，addb4 变量大于 0，则表示新的规则应该插入到该现有规则之前。

```
417             addb4=0; //在每次进行下一个当前规则比较之前，将 addb4 初始化为 0。
418             newkind=ftmp->fw_flg & IP_FW_F_KIND;
419             oldkind=chtmp->fw_flg & IP_FW_F_KIND;
```

ip_fw 结构中 fw_flg 字段表示规则适用的协议类型以及端口号的表示方式（范围表示法，精确表示法），418-419 分别将 newkind，oldkind 变量初始化为当前被比较规则和新规则适用的协议类型。下面将对二者适用的协议进行比较。

```
420             if (newkind!=IP_FW_F_ALL
421                 &&  oldkind!=IP_FW_F_ALL
422                 &&  oldkind!=newkind)
423             {
424                 chtmp_prev=chtmp;
425                 continue;
426             }
```

420-426 行代码对新规则和被比较的现有规则进行适用协议的比较，规则链中规则排列顺序是：
1〉在适用协议上，适用协议范围窄的规则排在规则链的前端，如仅适用于 TCP 协议的规则排列在适用于所有协议的规则之前。
2〉适用相同协议的规则，对应子网的规则排在对应网络的规则之前（即子网掩码长度较长对应的规则排在规则链的较前端）。
3〉在 2〉基础上，如果子网掩码长度相同而且属于同一网络，则端口范围小的规则处于较前的位置上。

420-426 行首先对适用协议范围进行比较，newkind，oldkind 表示新规则和现有被比较规则的适用范围。如果二者都适用于特定的协议（而非所有协议），且二者适用的协议不同，则跳过对该现有规则的其他检查，因为没有比较意义，425 行代码进行到下一个现有规则的检查。

```
427             /*
428              *  Very very *UGLY* code...
429              *  Sorry,but i had to do this....
430              */

431             n_sa=ntohl(ftmp->fw_src.s_addr);
432             n_da=ntohl(ftmp->fw_dst.s_addr);
433             n_sm=ntohl(ftmp->fw_smsk.s_addr);
434             n_dm=ntohl(ftmp->fw_dmsk.s_addr);

435             o_sa=ntohl(chtmp->fw_src.s_addr);
436             o_da=ntohl(chtmp->fw_dst.s_addr);
437             o_sm=ntohl(chtmp->fw_smsk.s_addr);
438             o_dm=ntohl(chtmp->fw_dmsk.s_addr);

439             m_src_mask = o_sm & n_sm;
440             m_dst_mask = o_dm & n_dm;
```

```
441             if ((o_sa & m_src_mask) == (n_sa & m_src_mask))
442             {
443                 if (n_sm > o_sm)
444                     addb4++;
445                 if (n_sm < o_sm)
446                     addb4--;
447             }

448             if ((o_da & m_dst_mask) == (n_da & m_dst_mask))
449             {
450                 if (n_dm > o_dm)
451                     addb4++;
452                 if (n_dm < o_dm)
453                     addb4--;
454             }
```

431-454 行代码对应对子网掩码进行比较，431-440 行代码首先对被比较的参数进行初始化。注意 439-440 行进行新规则和当前被比较规则子网掩码的合并，即将子网掩码合并为较短子网掩码的值。这样做的目的在 441 行体现出来，即用于两个规则之间网络号的比较。在较宽范围内的网络号相同的情况下，443-446，450-453 对子网进一步比较。如果新的添加规则子网掩码长度较长（443，450 行代码），则新的规则应该插入到当前被比较规则之前。因为此时新的规则适用的范围较窄。

```
455             if (((o_da & o_dm) == (n_da & n_dm))
456                     &&((o_sa & o_sm) == (n_sa & n_sm)))
457             {
```

455 行进一步对二者子网号进行检查，如果二者具有相同的子网号，即两个规则对应于同一个网络，此时进一步对端口范围进行比较。前文中 420 行代码只是对适用于特定协议的规则具有不同的适用协议情况的检查，下面 458 行代码进一步检查二者适用协议的范围。在二者共同针对相同网络的情况下，如果新的规则的适用于特定的协议，而现有被比较规则适用于所有规则，则新的规则应该添加到现有规则之前。

```
458                 if (newkind!=IP_FW_F_ALL &&
459                     oldkind==IP_FW_F_ALL)
460                     addb4++;
461                 if (newkind==oldkind && (oldkind==IP_FW_F_TCP
462                     ||  oldkind==IP_FW_F_UDP))
463                 {
```

如果新规则和现有被比较规则适用于相同协议，且协议为 TCP，UDP（此时才有端口号比较的必要性），则进一步进行端口号的比较。

```
464                     /*
```

```
465                     *   Here the main idea is to check the size
466                     *   of port range which the frwl covers
467                     *   We actually don't check their values but
468                     *   just the wideness of range they have
469                     *   so that less wide ranges or single ports
470                     *   go first and wide ranges go later. No ports
471                     *   at all treated as a range of maximum number
472                     *   of ports.
473                     */

474                    if (ftmp->fw_flg & IP_FW_F_SRNG)
475                        n_sr=ftmp->fw_pts[1]-ftmp->fw_pts[0];
476                    else
477                        n_sr=(ftmp->fw_nsp)?
478                            ftmp->fw_nsp : 0xFFFF;

479                    if (chtmp->fw_flg & IP_FW_F_SRNG)
480                        o_sr=chtmp->fw_pts[1]-chtmp->fw_pts[0];
481                    else
482                        o_sr=(chtmp->fw_nsp)?chtmp->fw_nsp : 0xFFFF;
```

474-482 行代码获取新规则和当前被比较规则的适用端口范围，用相关变量进行表示，便于下文的比较（此处只针对源端端口）。注意 ip_fw 结构中 fw_flg 字段表示了端口表示方法，如果 IP_FW_F_SRNG 标志位被设置，则表示端口采用的是范围表示法；否则端口采用穷尽法表示，此时，fw_nsp,fw_ndp 分别表示源端和目的端端口号的个数。此处在对端口号的比较上，紧紧是以端口号表示的个数进行的，没有对具体端口号进行比较。

```
483                    if (n_sr<o_sr)
484                        addb4++;
485                    if (n_sr>o_sr)
486                        addb4--;
```

483-486 行代码对源端端口范围进行检查，如果新规则表示的端口范围较窄，则表示新规则应该添加到当前规则之前。

```
487                    n_n=ftmp->fw_nsp;
488                    n_o=chtmp->fw_nsp;

489                    /*
490                     * Actually this cannot happen as the frwl control
491                     * procedure checks for number of ports in source and
492                     * destination range but we will try to be more safe.
493                     */
```

```
494                     if ((n_n>(IP_FW_MAX_PORTS-2)) ||
495                         (n_o>(IP_FW_MAX_PORTS-2)))
496                         goto skip_check;

497                     if (ftmp->fw_flg & IP_FW_F_DRNG)
498                             n_dr=ftmp->fw_pts[n_n+1]-ftmp->fw_pts[n_n];
499                     else
500                             n_dr=(ftmp->fw_ndp)? ftmp->fw_ndp : 0xFFFF;

501                     if (chtmp->fw_flg & IP_FW_F_DRNG)
502                         o_dr=chtmp->fw_pts[n_o+1]-chtmp->fw_pts[n_o];
503                     else
504                         o_dr=(chtmp->fw_ndp)? chtmp->fw_ndp : 0xFFFF;
505                     if (n_dr<o_dr)
506                         addb4++;
507                     if (n_dr>o_dr)
508                         addb4--;
```

487-508 行代码对目的端口范围进行比较。首先对目的端口是否可进行比较进行检查。487-488 行首先将 n_n, n_o 变量初始化为新规则和当前被比较规则的源端端口个数，ip_fw 结构中表示端口的数组为 fw_pts，该数组最大长度为 IP_FW_MAX_PORTS（10）。目的端口的可比较性在于：最少存在两个端口对目的端口进行表示，因为此时还可使用范围表示法表示规则适用的端口范围。如果规则中 fw_pts 数组最多只有一个元素表示目的端口，就失去了比较的意义，因为在对端口进行比较时，采用的是端口个数的比较方式。494-496 行代码对目的端口个数进行了检查，如果不满足比较条件，则直接跳过对目的端口的比较。否则同源端端口一样，对目的端口也进行类型的端口范围大小的比较。

```
509  skip_check:
510                 }//端口号比较结束
511                 /* finally look at the interface address */
512                 if ((addb4 == 0) && ftmp->fw_via.s_addr &&
513                         !(chtmp->fw_via.s_addr))
514                     addb4++;
```

如果比较到此，新规则和当前被比较规则在 IP 地址，端口号比较上不分上下，则对规则中网络接收设备接口的 IP 地址进行比较。如果新规则中对接收设备 IP 地址进行了指定，而当前被比较规则中没有指定，则表示新的规则比当前被比较规则更具体，新规则应该添加到当前规则之前。

```
515             }
516             if (addb4>0)
517             {
518                 if (chtmp_prev)
519                 {
```

```
520                         chtmp_prev->fw_next=ftmp;
521                         ftmp->fw_next=chtmp;
522                 }
523                 else
524                 {
525                         *chainptr=ftmp;
526                         ftmp->fw_next=chtmp;
527                 }
528                 restore_flags(flags);
529                 return 0;
530         }
```

516-530 行代码根据以上所有比较的结果决定新规则的添加位置。516 行检查 addb4 变量值，如果该变量大于 0，则表示寻找到合适的添加位置，即添加到当前被比较规则之前。具体的添加代码如 518-527 行代码所示，在添加时，对当前被比较规则的位置进行了区分。在完成了新的规则添加后，529 行直接返回。如果 addb4 不大于 0，则表示新规则无法添加到当前被比较规则之前，我们还需要进行下一个规则的比较，531 行保存当前被比较规则，为下一轮循环中可能的添加操作（518-527 行）提供方便。

```
531                 chtmp_prev=chtmp;
532             }
533         }

534         if (chtmp_prev)
535             chtmp_prev->fw_next=ftmp;
536         else
537                 *chainptr=ftmp;
```

如果在规则链中查找到一个合适的添加位置，则在 529 行就已经返回，不会执行到 534 行代码。代码执行到 534 行，则表示已经行进到规则链的尾部，此时毫无疑义的新规则将作为规则链中最后一个规则被添加。534-537 行代码 if-else 语句块对规则链为空的情况再一次进行了检查。

```
538         restore_flags(flags);
539         return(0);
540 }
```

至此我们完成对新规则进行添加操作的 add_to_chain 函数的介绍，该函数显得比较长，长度的增加主要在于对新规则添加位置的寻找上，函数实现思想本身比较简单。在添加位置的寻找过程中，规则适用协议范围，网络范围，端口范围以及接口 IP 地址都作为了被比较对象，所依据的比较标准是范围越窄，规则处于规则链的越前端。以上所列举的四个比较条件中，优先级从大到小，只有在前一个条件无法确定结果时，才进行下一个条件的比较，这样将大大增加 add_to_chain 函数的执行效率。

```
541  static int del_from_chain(struct ip_fw *volatile*chainptr, struct ip_fw *frwl)
542  {
```

对应于 add_to_chain 规则添加函数，del_from_chain 函数用于删除规则。参数 chainptr 表示被删除规则所在的规则链；frwl 表示被删除的规则本身。在进行如下的具体分析之前，首先必须谨记的一点是，在进行规则的删除操作时，对满足删除条件的匹配规则的查找是精确匹配的。精确匹配的函数在下文中对该函数实现代码的分析中将体现出来。

```
543         struct ip_fw     *ftmp,*ltmp;
544         unsigned short tport1,tport2,tmpnum;
545         char         matches,was_found;
546         unsigned long         flags;

547         save_flags(flags);
548         cli();

549         ftmp=*chainptr;

550         if ( ftmp == NULL )
551         {
552  #ifdef DEBUG_CONFIG_IP_FIREWALL
553                 printk("ip_fw_ctl:    chain is empty\n");
554  #endif
555                 restore_flags(flags);
556                 return( EINVAL );
557         }
```

550 行对规则链进行检查，如果规则链为空，则表示删除操作不合法，我们无法对其中不包含任何规则的规则链进行规则删除操作。如果规则链中存在规则，下面就遍历该规则链，对其中每个规则进行匹配。

```
558         ltmp=NULL;
559         was_found=0;
```

变量 was_found 表示查找到满足参数 frwl 删除条件的规则，这个参数用于函数结尾处决定函数返回值。注意在查找到一个匹配规则后，并不是在删除该规则后就返回，而是继续对规则链中后续规则进一步进行查找。因为 frwl 表示的删除条件可能对应规则链中多个规则，即不止删除一个规则。

```
560         while( ftmp != NULL )
561         {
562                 matches=1;
```

首先将 matches 变量设置为 1，即首先假定当前被检查规则符合删除条件，下面进行具体比较时，如果出现不匹配的条件，才将该变量设置为 0。

```
563             if (ftmp->fw_src.s_addr!=frwl->fw_src.s_addr
564                 || ftmp->fw_dst.s_addr!=frwl->fw_dst.s_addr
565                 || ftmp->fw_smsk.s_addr!=frwl->fw_smsk.s_addr
566                 || ftmp->fw_dmsk.s_addr!=frwl->fw_dmsk.s_addr
567                 || ftmp->fw_via.s_addr!=frwl->fw_via.s_addr
568                 || ftmp->fw_flg!=frwl->fw_flg)
569                     matches=0;
```

563-569 行代码对网络号，子网掩码，接口地址，以及标志位（规则适用协议）进行比较，只有其中之一不匹配，则表示当前规则不满足删除条件，将 matches 变量重设为 0。

```
570         tport1=ftmp->fw_nsp+ftmp->fw_ndp;
571         tport2=frwl->fw_nsp+frwl->fw_ndp;
572         if (tport1!=tport2)
573                 matches=0;
574         else if (tport1!=0)
575         {
576             for (tmpnum=0;tmpnum < tport1 &&
                                    tmpnum < IP_FW_MAX_PORTS;tmpnum++)
577                 if (ftmp->fw_pts[tmpnum]!=frwl->fw_pts[tmpnum])
578                     matches=0;
579         }
```

570-579 行代码进一步检查端口号，首先对端口个数进行检查（572 行），如果端口个数不相同，就没有必要进行端口号的具体检查，直接将 matches 变量设置为 0。否则进一步进行端口号的具体检查。576-578 行代码即为具体检查的实现代码，这是对 fw_pts 数组中每个元素进行一一比较完成的。

```
580         if(matches)
581         {
582             was_found=1;
583             if (ltmp)
584             {
585                 ltmp->fw_next=ftmp->fw_next;
586                 kfree_s(ftmp,sizeof(*ftmp));
587                 ftmp=ltmp->fw_next;
588                 }
589                 else
590                 {
591                     *chainptr=ftmp->fw_next;
592                 kfree_s(ftmp,sizeof(*ftmp));
593                 ftmp=*chainptr;
594             }
```

```
595             }
```

580-595 行代码对匹配的情况进行处理，matches 变量不为 0，表示寻找到一个满足删除条件的规则项。下面就对该规则进行删除操作。删除时对被删除规则的位置进行了区分：如果是一个中间规则，直接更改前一个规则相关指针即可；如果是规则链中第一个规则，则更新指向规则链的全局变量值；无论何种情况，在将规则从规则链中删除后，最后都是调用 kfree_s 函数对规则进行内存的释放。注意在查找到一个满足删除条件的规则，并对其进行删除后，并不是从函数中退出，而是继续对规则链中后续规则的检查（587，593 行）。

```
596             else
597             {
598                     ltmp = ftmp;
599                     ftmp = ftmp->fw_next;
600              }
```

596 行 else 语句块对应 matches 变量为 0 的情况，表示当前被检查规则不满足删除条件，此时前进到规则链中下一个规则进行检查。

```
601         }
602         restore_flags(flags);
603         if (was_found)
604                 return 0;
605         else
606                 return(EINVAL);
607 }
```

```
608 #endif    /* CONFIG_IP_ACCT || CONFIG_IP_FIREWALL */
```

函数最后根据是否查找到满足删除条件的规则返回不同的值，如果寻找到满足删除条件的规则，则返回 0；否则返回 EINVAL，表示参数不合法，没有在指定规则链中寻找到对应删除条件的规则。del_from_chain 函数完成规则的删除操作，在进行删除条件的匹配时，进行的匹配检查是非常严格的，各比较条件必须一一匹配，因为规则的删除涉及到防火墙功能的完备以及主机的安全性，如果删除一个不该删除的规则，则很可能构成严重的系统安全漏洞。所以对于该函数的实现代码中严格的条件检查应该能够“谅解”。

```
609 struct ip_fw *check_ipfw_struct(struct ip_fw *frwl, int len)
610 {
```

check_ipfw_struct 函数对一个表示规则的 ip_fw 结构中各字段进行合法性检查。参数 frwl 表示被检查的规则，len 应该表示的是 ip_fw 结构的长度。

```
611         if ( len != sizeof(struct ip_fw) )
612         {
```

```
613 #ifdef DEBUG_CONFIG_IP_FIREWALL
614             printk("ip_fw_ctl: len=%d, want %d\n",m->m_len,
615                                 sizeof(struct ip_fw));
616 #endif
617             return(NULL);
618         }

619         if ( (frwl->fw_flg & ~IP_FW_F_MASK) != 0 )
620         {
621 #ifdef DEBUG_CONFIG_IP_FIREWALL
622             printk("ip_fw_ctl: undefined flag bits set (flags=%x)\n",
623                     frwl->fw_flg);
624 #endif
625             return(NULL);
626         }
```

结构 ip_fw 中 fw_flg 字段设置一些标志位对其他字段的意义进行设置，如端口表示法，规则适用协议等。如果 fw_flg 字段设置了其他保留标志位字段，则认为是错误设置（这比较严格，从使用的角度而言，这不会产生影响，但此处的处理认为是一种错误，由此也可见，对于防火墙规则的严格检查）。

```
627         if ( (frwl->fw_flg & IP_FW_F_SRNG) && frwl->fw_nsp < 2 )
628         {
629 #ifdef DEBUG_CONFIG_IP_FIREWALL
630             printk("ip_fw_ctl: src range set but n_src_p=%d\n",
631                     frwl->fw_nsp);
632 #endif
633             return(NULL);
634         }

635         if ( (frwl->fw_flg & IP_FW_F_DRNG) && frwl->fw_ndp < 2 )
636         {
637 #ifdef DEBUG_CONFIG_IP_FIREWALL
638             printk("ip_fw_ctl: dst range set but n_dst_p=%d\n",
639                     frwl->fw_ndp);
640 #endif
641             return(NULL);
642         }
```

627-642 行代码对端口设置进行检查，如果端口使用范围表示法，则至少必须由两个端口进行表示。否则表示设置有误。

```
643         if ( frwl->fw_nsp + frwl->fw_ndp > IP_FW_MAX_PORTS )
644         {
```

```
645  #ifdef DEBUG_CONFIG_IP_FIREWALL
646              printk("ip_fw_ctl: too many ports (%d+%d)\n",
647                      frwl->fw_nsp,frwl->fw_ndp);
648  #endif
649              return(NULL);
650      }
```

643-650 行代码对端口数组是否溢出数组进行检查，如果溢出，则表示设置有误。

```
651      return frwl;
652  }
```

在通过所有合法性检查后，返回这个被检查的规则对应 ip_fw 结构，否则返回 NULL，表示被检查规则设置有误。

综上所述，以上介绍的 zero_fw_chain, free_fw_chain, add_to_chain, del_from_chain 以及 check_ipfw_struct 函数都是用于对规则链进行设置，这些函数上层用户对规则进行相关操作的底层实现函数，下面介绍对规则链进行操作的 ioctl 函数时，将会看到对这些函数的具体调用。

```
653  #ifdef CONFIG_IP_ACCT

654  void ip_acct_cnt(struct iphdr *iph, struct device *dev, struct ip_fw *f)
655  {
656      (void) ip_fw_chk(iph, dev, f, 0, 1);
657      return;
658  }
```

ip_acct_cnt 函数用于信息统计，该函数简单调用 ip_fw_chk 函数完成具体的统计功能，注意最后一个参数设置为 1，用于调制 ip_fw_chk 函数的相关功能；参数 f 表示被检查的规则链。

在 ip_setsockopt 函数中，我们曾经留下两个函数没有解释，这两个函数分别用于操作防火墙的规则链：其一为 ip_acct_ctl，对信息统计规则链进行操作；其二为 ip_fw_ctl，对数据包过滤规则链进行操作。为了便于对这两个函数的理解，下面我们再次给出 ip_setsockopt 函数中对这两个函数的调用环境。

```
/*net/inet/ip.c – ip_setsockopt 函数*/
1930      #ifdef CONFIG_IP_FIREWALL
1931              case IP_FW_ADD_BLK: //本地规则链规则添加
1932              case IP_FW_DEL_BLK: //本地规则链规则删除
1933              case IP_FW_ADD_FWD: //转发规则链规则添加
1934              case IP_FW_DEL_FWD: //转发规则链规则删除
1935              case IP_FW_CHK_BLK:   //本地规则链规则检查 （Check）
1936              case IP_FW_CHK_FWD: //转发规则链规则检查（Check）
```

```
1937                case IP_FW_FLUSH_BLK: //对本地规则链进行清空（删除所有规则）
1938                case IP_FW_FLUSH_FWD: //对转发规则链进行清空
1939                case IP_FW_ZERO_BLK: //对本地规则链中规则统计信息清零
1940                case IP_FW_ZERO_FWD: //对转发规则链中规则统计信息清零
1941                case IP_FW_POLICY_BLK: //设置本地规则链的默认数据包处理策略
1942                case IP_FW_POLICY_FWD: //设置转发规则链的默认数据包处理策略
1943                    if(!suser()) //对防火墙规则的任何操作都需要超级用户权限
1944                        return -EPERM;
1945                    if(optlen>sizeof(tmp_fw) || optlen<1)
1946                        return -EINVAL;
1947                    err=verify_area(VERIFY_READ,optval,optlen);
1948                    if(err)
1949                        return err;
1950                    memcpy_fromfs(&tmp_fw,optval,optlen);
1951                    err=ip_fw_ctl(optname, &tmp_fw,optlen);
1952                    return -err;    /* -0 is 0 after all */

1953    #endif
1954    #ifdef CONFIG_IP_ACCT
1955                case IP_ACCT_DEL: //删除信息统计规则链规则
1956                case IP_ACCT_ADD: //信息统计规则链规则添加
1957                case IP_ACCT_FLUSH: //对信息统计规则链清空
1958                case IP_ACCT_ZERO: //对信息统计规则链中所有规则统计信息进行清零
1959                    if(!suser())
1960                        return -EPERM;
1961                    if(optlen>sizeof(tmp_fw) || optlen<1)
1962                        return -EINVAL;
1963                    err=verify_area(VERIFY_READ,optval,optlen);
1964                    if(err)
1965                        return err;
1966                    memcpy_fromfs(&tmp_fw, optval,optlen);
1967                    err=ip_acct_ctl(optname, &tmp_fw,optlen);
1968                    return -err;    /* -0 is 0 after all */
1969    #endif
```

注意我们将由全局变量 ip_fw_blk_chain 指向的 ip_fw 结构类型队列表示的防火墙规则链表称为本地规则链，仅仅是为了区别于由 ip_fw_fwd_chain 变量指向的转发规则链。此处的名称不指定任何意义。同理我们将由变量 ip_acct_chain 指向的规则链称为信息统计规则链。从以上代码片断可见，所以对于转发规则链和本地规则链进行的操作均有 ip_fw_ctl 函数负责；而对信息统计规则链进行的所有操作均有 ip_acct_ctl 函数负责。下面我们就对这两个函数进行介绍。

```
659  int ip_acct_ctl(int stage, void *m, int len)
660  {
```

```
661         if ( stage == IP_ACCT_FLUSH )
662         {
663                 free_fw_chain(&ip_acct_chain);
664                 return(0);
665         }
666         if ( stage == IP_ACCT_ZERO )
667         {
668                 zero_fw_chain(ip_acct_chain);
669                 return(0);
670         }
671         if ( stage == IP_ACCT_ADD
672            || stage == IP_ACCT_DEL
673              )
674         {
675                 struct ip_fw *frwl;

676                 if (!(frwl=check_ipfw_struct(m,len)))
677                         return (EINVAL);

678                 switch (stage)
679                 {
680                         case IP_ACCT_ADD:
681                                 return( add_to_chain(&ip_acct_chain,frwl));
682                                 case IP_ACCT_DEL:
683                                 return( del_from_chain(&ip_acct_chain,frwl));
684                         default:
685                                 /*
686                                   *    Should be panic but... (Why ??? - AC)
687                                   */
688 #ifdef DEBUG_CONFIG_IP_FIREWALL
689                                 printf("ip_acct_ctl:    unknown request %d\n",stage);
690 #endif
691                                 return(EINVAL);
692                 }
693         }
694 #ifdef DEBUG_CONFIG_IP_FIREWALL
695         printf("ip_acct_ctl:    unknown request %d\n",stage);
696 #endif
697         return(EINVAL);
698 }
699 #endif
```

ip_acct_ctl 函数实现思想很简单，经过前文中对于相关具体操作函数的分析介绍，该函数的实现代码对于读者应该不成问题，此处不再对其讨论。同理下面的 ip_fw_ctl 函数我们也留

给读者自行分析理解。

```
700  #ifdef CONFIG_IP_FIREWALL
701  int ip_fw_ctl(int stage, void *m, int len)
702  {
703          int ret;

704          if ( stage == IP_FW_FLUSH_BLK )
705          {
706                  free_fw_chain(&ip_fw_blk_chain);
707                  return(0);
708          }

709          if ( stage == IP_FW_FLUSH_FWD )
710          {
711                  free_fw_chain(&ip_fw_fwd_chain);
712                  return(0);
713          }

714          if ( stage == IP_FW_ZERO_BLK )
715          {
716                  zero_fw_chain(ip_fw_blk_chain);
717                  return(0);
718          }

719          if ( stage == IP_FW_ZERO_FWD )
720          {
721                  zero_fw_chain(ip_fw_fwd_chain);
722                  return(0);
723          }

724          if ( stage == IP_FW_POLICY_BLK || stage == IP_FW_POLICY_FWD )
725          {
726                  int *tmp_policy_ptr;
727                  tmp_policy_ptr=(int *)m;
728                  if ( stage == IP_FW_POLICY_BLK )
729                          ip_fw_blk_policy=*tmp_policy_ptr;
730                  else
731                          ip_fw_fwd_policy=*tmp_policy_ptr;
732                  return 0;
733          }

734          if ( stage == IP_FW_CHK_BLK || stage == IP_FW_CHK_FWD )
735          {
```

```
736             struct device viadev;
737             struct ip_fwpkt *ipfwp;
738             struct iphdr *ip;

739             if ( len < sizeof(struct ip_fwpkt) )
740             {
741 #ifdef DEBUG_CONFIG_IP_FIREWALL
742                     printf("ip_fw_ctl: length=%d, expected %d\n",
743                             len, sizeof(struct ip_fwpkt));
744 #endif
745                     return( EINVAL );
746             }

747             ipfwp = (struct ip_fwpkt *)m;
748             ip = &(ipfwp->fwp_iph);

749             if ( ip->ihl != sizeof(struct iphdr) / sizeof(int))
750             {
751 #ifdef DEBUG_CONFIG_IP_FIREWALL
752                     printf("ip_fw_ctl: ip->ihl=%d, want %d\n",ip->ihl,
753                                     sizeof(struct ip)/sizeof(int));
754 #endif
755                     return(EINVAL);
756             }

757             viadev.pa_addr = ipfwp->fwp_via.s_addr;

758             if ((ret = ip_fw_chk(ip, &viadev,
759                     stage == IP_FW_CHK_BLK ?
760                                     ip_fw_blk_chain : ip_fw_fwd_chain,
761                     stage == IP_FW_CHK_BLK ?
762                                     ip_fw_blk_policy : ip_fw_fwd_policy, 2 )) > 0
763                    )
764                     return(0);
765                     else if (ret == -1)
766                     return(ECONNREFUSED);
767             else
768                     return(ETIMEDOUT);
769         }

770 /*
771  *    Here we really working hard-adding new elements
772  *    to blocking/forwarding chains or deleting 'em
773  */
```

```
774         if ( stage == IP_FW_ADD_BLK || stage == IP_FW_ADD_FWD
775                 || stage == IP_FW_DEL_BLK || stage == IP_FW_DEL_FWD
776                 )
777         {
778                 struct ip_fw *frwl;
779                 frwl=check_ipfw_struct(m,len);
780                 if (frwl==NULL)
781                         return (EINVAL);

782                 switch (stage)
783                 {
784                         case IP_FW_ADD_BLK:
785                                 return(add_to_chain(&ip_fw_blk_chain,frwl));
786                         case IP_FW_ADD_FWD:
787                                 return(add_to_chain(&ip_fw_fwd_chain,frwl));
788                         case IP_FW_DEL_BLK:
789                                 return(del_from_chain(&ip_fw_blk_chain,frwl));
790                         case IP_FW_DEL_FWD:
791                                 return(del_from_chain(&ip_fw_fwd_chain,frwl));
792                         default:
793                         /*
794                           *    Should be panic but... (Why are BSD people panic obsessed ??)
795                           */
796 #ifdef DEBUG_CONFIG_IP_FIREWALL
797                                 printk("ip_fw_ctl:    unknown request %d\n",stage);
798 #endif
799                                 return(EINVAL);
800                 }
801         }

802 #ifdef DEBUG_CONFIG_IP_FIREWALL
803         printf("ip_fw_ctl:    unknown request %d\n",stage);
804 #endif
805         return(EINVAL);
806 }
807 #endif /* CONFIG_IP_FIREWALL */



808 #if defined(CONFIG_IP_FIREWALL) || defined(CONFIG_IP_ACCT)

809 static int ip_chain_procinfo(int stage, char *buffer, char **start,
810                 off_t offset, int length, int reset)
```

```
811 {
```

ip_chain_procinfo 函数用于获取防火墙规则链中所有规则定义，结合 ip_fw 结构的定义，本函数很容易理解。当用户需要打印当前所有规则时，该函数将被调用返回所有规则定义信息。参数 stage 指定需要打印的规则链。此处需要对 start，offset，length 参数进行着重说明，否则以下 857-874 行代码很难理解，或者说我们可以从 857-874 行代码倒推出这些参数的实际意义如下：
start：表示用户实际要求返回的数据在 buffer 数组中的起始偏移量。
offset：表示用户实际要求返回的数据在整个规则信息集合中的偏移量。如果我们将指定规则链中所有规则构成的信息集合看作一个流，那么 offset 参数就表示用户要求的数据在这个流中的起始偏移量。
length：表示用户要求返回的数据量长度。
如果读者还不明白，请查看如下 857-874 行代码实现所体现的含义。

```
812         off_t pos=0, begin=0;
813         struct ip_fw *i;
814         unsigned long flags;
815         int len, p;


816         switch(stage)
817         {
818 #ifdef CONFIG_IP_FIREWALL
819             case IP_INFO_BLK:
820                 i = ip_fw_blk_chain;
821                 len=sprintf(buffer, "IP firewall block rules, default %d\n",
822                     ip_fw_blk_policy);
823                 break;
824             case IP_INFO_FWD:
825                 i = ip_fw_fwd_chain;
826                 len=sprintf(buffer, "IP firewall forward rules, default %d\n",
827                     ip_fw_fwd_policy);
828                 break;
829 #endif
830 #ifdef CONFIG_IP_ACCT
831             case IP_INFO_ACCT:
832                 i = ip_acct_chain;
833                 len=sprintf(buffer,"IP accounting rules\n");
834                 break;
835 #endif
836             default:
837                 /* this should never be reached, but safety first... */
838                 i = NULL;
839                 len=0;
```

```
840                 break;
841         }
```

816-841 行代码根据 stage 参数决定对哪个规则链进行规则信息打印。在激昂变量 i 初始化为指定的规则链后，844 行代码对该规则链进行遍历，返回每个规则的信息。

```
842         save_flags(flags);
843         cli();

844         while(i!=NULL)
845         {
846             len+=sprintf(buffer+len,"%08lX/%08lX->%08lX/%08lX %08lX %X ",
847                 ntohl(i->fw_src.s_addr),ntohl(i->fw_smsk.s_addr),
848                 ntohl(i->fw_dst.s_addr),ntohl(i->fw_dmsk.s_addr),
849                 ntohl(i->fw_via.s_addr),i->fw_flg);
850             len+=sprintf(buffer+len,"%u %u %lu %lu",
851                 i->fw_nsp,i->fw_ndp, i->fw_pcnt,i->fw_bcnt);
852             for (p = 0; p < IP_FW_MAX_PORTS; p++)
853                 len+=sprintf(buffer+len, " %u", i->fw_pts[p]);
854             buffer[len++]='\n';
855             buffer[len]='\0';
856             pos=begin+len;
857             if(pos<offset)
858             {
859                 len=0;
860                 begin=pos;
861             }
```

变量 pos 表示当前获取的信息在整个信息流中的偏移量，而 offset 参数如前文中所述为用户所要求数据在整个信息流中的起始偏移量，如果 pos 小于 offset，则表示当前填充在 buffer 缓冲区中的数据不是用户要求的数据，此时将 len 字段清零，从而下一次进行信息填充时，将覆盖本次填充的信息，即依然从 buffer 缓冲区起始处填充；如果 pos 大于 offset，则表示当前填充的信息至少有一部分是用户请求的，我们将 offset 减去上一次保存的 pos 值，就得到 buffer 数组中用户实际请求数据的起始偏移量，在数组开始处，可能存在用户不需要的数据。begin 变量被赋值为 pos，这个赋值的作用即用于计算用户实际请求数据在 buffer 数组中的起始偏移量，这在 873 行代码中体现出来。

```
862             else if(reset)
863             {
864                 /* This needs to be done at this specific place! */
865                 i->fw_pcnt=0L;
866                 i->fw_bcnt=0L;
867             }
```

862 行代码对 reset 参数设置为 1 的情况进行处理，注意 862 行代码的执行与 857 行互斥，即如果当前规则信息不在用户请求范围内，则不可对该规则中相关信息统计字段进行清零。只有在请求范围内时，如果 reset 参数设置为 1，则进行清零处理。

```
868             if(pos>offset+length)
869                     break;
```

868 行检查是否已经获得到用户请求的数据量，如果已经获取了足够的信息，则可以停止对剩下规则的统计了。

```
870             i=i->fw_next;
871     }
872     restore_flags(flags);
873     *start=buffer+(offset-begin);
```

offset 参数表示用户请求数据在整个数据流中的偏移，而 begin 字段表示 buffer 数组中填充的第一个字节数据在整个数据流中的偏移，offset 减去 begin 即得到用户请求数据在 buffer 数组中的偏移。注意 start 参数是一个地址，此处 873 行将其指向的内存初始化为用户所请求数据第一个字节所在的内存地址。

```
874     len-=(offset-begin);
875     if(len>length)
876             len=length;
```

874 行将 len 变量赋值为所统计数据的长度，如果这个长度大于用户请求的长度，则 876 行简单将该变量设置为用户要求的长度，并返回。

```
877     return len;
878 }
879 #endif
```

在本书前文中，也有相关信息获取函数，有类似的实现，对于其中 offset，start，length 等参数都没有进行详细说明，ip_chain_procinfo 函数信息获取函数中的其中之一，实现思想类同，此处对这些参数的说明读者可直接对应到其他相关函数中进行理解。此后，如果遇到类似代码，本书将不再做详细分析。

```
880 #ifdef CONFIG_IP_ACCT

881 int ip_acct_procinfo(char *buffer, char **start, off_t offset, int length, int reset)
882 {
883     return ip_chain_procinfo(IP_INFO_ACCT, buffer,start,offset,length,reset);
884 }

885 #endif
```

```
886  #ifdef CONFIG_IP_FIREWALL

887  int ip_fw_blk_procinfo(char *buffer, char **start, off_t offset, int length, int reset)
888  {
889          return ip_chain_procinfo(IP_INFO_BLK, buffer,start,offset,length,reset);
890  }

891  int ip_fw_fwd_procinfo(char *buffer, char **start, off_t offset, int length, int reset)
892  {
893          return ip_chain_procinfo(IP_INFO_FWD, buffer,start,offset,length,reset);
894  }

895  #endif
```

880-895 行代码是对上文中介绍的 ip_chain_procinfo 函数的封装调用，分别获取不同规则链的规则信息。

ip_fw.c 文件小结

ip_fw.c 文件是防火墙功能实现文件，防火墙简单的说就是对进出数据包进行检查的一套规则，在实现上，每个规则有一个 ip_fw 结构表示。本版本网络栈实现代码中，共定义有三个规则链：由 ip_fw_fwd_chain 变量指向的数据包转发规则链；由 ip_fw_blk_chain 变量指向的本地规则链；以及由 ip_acct_chain 变量指向的数据包进出流量统计规则链。对数据包的过滤（或检查）操作实现为对具体的规则链进行遍历，根据每个规则的具体规定检查数据包中各字段信息进行规则匹配，一旦匹配成功，则根据该规则配置的数据包处理策略对进出的数据包进行对应的操作。ip_fw.c 文件中数据包过滤功能具体由 ip_fw_chk 函数实现，在 IP 协议实现模块中，在数据包进出端口函数中，都会调用该函数的相关代码，对数据包进行检查和过滤操作；至于 ip_fw.c 文件中其他函数都是对规则链本身的操作，如添加，删除规则等等。

防火墙实现以及路由表实现，至此我们都已进行了介绍，将二者对照来看，具有很多类似的地方。而这两个功能在目前版本网络栈实现中，代码已经变得相当复杂庞大。但所基于的根本原理类似。在理解了基本原理后，在对后续版本代码进行分析时，就可以首先从全局上把握。这也是我们分析早期代码实现的根本意义所在！对于防火墙和路由表的实现代码，读者应该深入的进行反复分析和理解，从而克服我们长期以来由于各种原因形成的对他们的为难情绪。

在本书前文中分析 IP 协议之前，我们着重对 TCP，UDP，ICMP，IGMP 协议进行了分析，因为这些协议都需要使用 IP 协议进行封装（虽然我们从概念上一般将 ICMP，IGMP 作为网络层协议，与 IP 协议同层）。以上每个协议都由一个 inet_protocol 结构表示，IP 协议实现模块数据包接收总入口函数 ip_rcv 在进行完本层处理后，将数据包传递给上层进行处理时就是根据 IP 首部中对应的上层协议号，从 inet_protos 数组中索引上层协议对应的 inet_protocol

结构，调用该结构中 handler 指针指向的函数，如 TCP 协议对应的 tcp_rcv, UDP 协议对应的 upd_rcv, 以及 ICMP 协议的 icmp_rcv 函数等等。但在 ip_rcv 函数实现中，在向上层进行数据包传递时，有如下另外的其他代码：

/*net/inet/ip.c—ip_rcv 函数代码片段*/

```
1404        hash = iph->protocol & (SOCK_ARRAY_SIZE-1);

1405        /* If there maybe a raw socket we must check - if not we don't care less */
1406        if((raw_sk=raw_prot.sock_array[hash])!=NULL)
1407        {
1408            struct sock *sknext=NULL;
1409            struct sk_buff *skb1;
1410            raw_sk=get_sock_raw(raw_sk, hash,    iph->saddr, iph->daddr);
1411            if(raw_sk)      /* Any raw sockets */
1412            {
1413                do
1414                {
1415                    /* Find the next */
1416                    sknext=get_sock_raw(raw_sk->next, hash, iph->saddr, iph->daddr);
1417                    if(sknext)
1418                        skb1=skb_clone(skb, GFP_ATOMIC);
1419                    else
1420                        break;     /* One pending raw socket left */
1421                    if(skb1)
1422                        raw_rcv(raw_sk, skb1, dev, iph->saddr,iph->daddr);
1423                    raw_sk=sknext;
1424                }
1425                while(raw_sk!=NULL);
1426                /* Here either raw_sk is the last raw socket, or NULL if none */
1427                /* We deliver to the last raw socket AFTER the protocol checks as it
avoids a surplus copy */
1428            }
1429        }
        ……
1471        if(raw_sk!=NULL) /* Shift to last raw user */
1472            raw_rcv(raw_sk, skb, dev, iph->saddr, iph->daddr);
```

这段代码处理 RAW 套接字类型，RAW 类型套接字操作函数集合由 raw_prot 变量表示，其对应实现文件为 net/inet/raw.c，RAW 类型套接字也是建立在 IP 协议之上，即使用 IP 协议进行封装。在处理上又区别于其他使用 IP 协议进行封装的协议，在设计 IP 协议与其接口的方式上，RAW 类型套接字并非使用一个 inet_protocol 结构供调用，从如上代码来看，ip_rcv 函数对 RAW 类型套接字实行特殊对待，直接从 raw_prot 变量的 sock_array 数组队列中进行上层协议套接字的查找，并且统一使用 raw_rcv 函数将数据包挂接到对应套接字接收队列中。如 TCP，UDP 协议类似，所有 RAW 类型套接字都被挂接在 raw_prot 变量的 sock_array 数组队列中。一个具体的 RAW 类型套接字所在队列在 sock_array 数组中的索引号由进行 RAW 类型套接字创建时用户传入的参数决定（socket 系统调用的第三个参数，对于 RAW 类型套

接字，这个参数值为 IPPROTO_RAW：255），这个参数在创建 IP 首部时被作为 IP 首部中的上层协议字段值。这一点可以从 inet_create(af_inet.c)函数和下文中将要介绍的 raw_sendto 函数得出。在 inet_create 函数中，新创建的 RAW 类型套接字对应 sock 结构中 protocol 字段被初始化为 socket 系统调用时用户输入的第三个参数值，而在 raw_sendto 函数中进行 IP 首部创建时，sock 结构中 protocol 字段值又被作为 IP 首部的上层协议字段值，同时在 inet_create 函数中 sock 结构 num 字段也被初始化为相同的值，在将 sock 结构添加到 raw_prot 中 sock_array 数组队列中时，sock 结构 num 字段值被作为 sock_array 数组索引寻址对应的 sock 结构队列。在上文代码片段中，IP 首部中上层协议字段值被作为索引寻址 raw_prot 中 sock_array 数组队列，这就统一到一点了。但有一点很可笑的是，由于所有 RAW 类型套接字创建时使用的 protocol 值都是 IPPROTO_RAW，即 255，这表示所有的 RAW 类型套接字对应 sock 结构都被挂接到 raw_prot 中 sock_array 数组中的同一个队列中，其他其他均没有被使用。

下面我们就对 RAW 类型套接字实现文件 raw.c 进行分析，RAW 类型数据包使用 IP 协议进行封装，但在 IP 首部之后，其没有规定的其他首部格式，即在 IP 首部之后的内容完全由用户定制，这一点不同于 TCP 协议（以及 UDP 协议，ICMP 协议等）套接字，在 IP 首部之后跟随一个固定格式的 TCP 首部，而且这个 TCP 首部是必须的，RAW 类型套接字在这一点上要灵活的多。

2.23 net/inet/raw.c 文件

```
1     /*
2      * INET        An implementation of the TCP/IP protocol suite for the LINUX
3      *        operating system.    INET is implemented using the    BSD Socket
4      *        interface as the means of communication with the user level.
5      *
6      *        RAW - implementation of IP "raw" sockets.
7      *
8      * Version:    @(#)raw.c       1.0.4       05/25/93
9      *
10     * Authors:    Ross Biro, <bir7@leland.Stanford.Edu>
11     *        Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
12     *
13     * Fixes:
14     *        Alan Cox:       verify_area() fixed up
15     *        Alan Cox:       ICMP error handling
16     *        Alan Cox:       EMSGSIZE if you send too big a packet
17     *        Alan Cox:       Now uses generic datagrams and shared skbuff
18     *                        library. No more peek crashes, no more backlogs
19     *        Alan Cox:       Checks sk->broadcast.
20     *        Alan Cox:       Uses skb_free_datagram/skb_copy_datagram
21     *        Alan Cox:       Raw passes ip options too
22     *        Alan Cox:       Setsocketopt added
23     *        Alan Cox:       Fixed error return for broadcasts
```

```
24   *          Alan Cox :      Removed wake_up calls
25   *          Alan Cox :      Use ttl/tos
26   *          Alan Cox :      Cleaned up old debugging
27   *          Alan Cox :      Use new kernel side addresses
28   *    Arnt Gulbrandsen  :      Fixed MSG_DONTROUTE in raw sockets.
29   *          Alan Cox :      BSD style RAW socket demultiplexing.
30   *
31   *          This program is free software; you can redistribute it and/or
32   *          modify it under the terms of the GNU General Public License
33   *          as published by the Free Software Foundation; either version
34   *          2 of the License, or (at your option) any later version.
35   */
36  #include <asm/system.h>
37  #include <asm/segment.h>
38  #include <linux/types.h>
39  #include <linux/sched.h>
40  #include <linux/errno.h>
41  #include <linux/timer.h>
42  #include <linux/mm.h>
43  #include <linux/kernel.h>
44  #include <linux/fcntl.h>
45  #include <linux/socket.h>
46  #include <linux/in.h>
47  #include <linux/inet.h>
48  #include <linux/netdevice.h>
49  #include "ip.h"
50  #include "protocol.h"
51  #include <linux/skbuff.h>
52  #include "sock.h"
53  #include "icmp.h"
54  #include "udp.h"


55  static inline unsigned long min(unsigned long a, unsigned long b)
56  {
57          if (a < b)
58                  return(a);
59          return(b);
60  }

61  /* raw_err gets called by the icmp module. */
62  void raw_err (int err, unsigned char *header, unsigned long daddr,
63          unsigned long saddr, struct inet_protocol *protocol)
64  {
```

raw_err 函数定义的初衷类似于 tcp_err, udp_err 等函数，但由于此类函数的调用一般在 ICMP 实现模块中，ICMP 实现模块对这些函数的调用又是通过 inet_protocol 结构队列进行，而对于 RAW 类型套接字，并没有定义这样一个 inet_protocol 结构，所以 raw_err 函数在本版本实现中没有被任何其他函数调用。这可以作为本版本网络栈实现的一个未完善之处。这个结果的造成是由于对于 RAW 类型套接字，内核实现上没有使用统一的接口，或者说，ip_rcv 函数实现上区分对待，而非使用统一的调用接口（通过 inet_protocol 结构完成）。另外一点一直没有提及，使用 RAW 类型套接字时，用户必须自行进行 IP 首部的创建，这一点在后文介绍中自会清楚。另外本版本网络栈实现代码还定义有 PACKET 类型套接字，其直接使用链路层发送接口函数进行数据包的发送，换句话说，对于 PACKET 类型套接字用户必须自己完成 MAC，IP 首部的创建，以及 IP 首部之后内容的实现（即用户必须完成整个数据帧的生成），这一点在下文中分析 PACKET 类型套接字实现文件 packet.c 时自会明白。

```
65          struct sock *sk;

66          if (protocol == NULL)
67                  return;
68          sk = (struct sock *) protocol->data;
69          if (sk == NULL)
70                  return;

71          /* This is meaningless in raw sockets. */
72          if (err & 0xff00 == (ICMP_SOURCE_QUENCH << 8))
73          {
74                  if (sk->cong_window > 1) sk->cong_window = sk->cong_window/2;
75                  return;
76          }
```

如果错误类型是源端节制，表示接收端处理速率跟不上本地发送速度，需要本地进行发送速率的节制，处理上通过检查拥塞窗口即可。此处是将拥塞窗口大小减半处理（74 行）。本地发送速率由远端接收缓冲区大小和本地拥塞窗口大小进行节制。

```
77          sk->err = icmp_err_convert[err & 0xff].errno;
78          sk->error_report(sk);
```

对于接收错误的最终处理是将错误类型值保存到 sock 结构的 err 字段中，并回调错误通知函数通知用户，用户进程将读取 err 字段值，返回给用户，进行错误通知。

```
79          return;
80  }

81  /*
82   *  This should be the easiest of all, all we do is
83   *  copy it into a buffer. All demultiplexing is done
```

```
84     *    in ip.c
85     */

86    int raw_rcv(struct sock *sk, struct sk_buff *skb, struct device *dev, long saddr, long daddr)
87    {
```

raw_rcv 函数就是在 ip_rcv 函数被调用对 RAW 类型数据包进行接收的函数。读者可参考前文中列出的 ip_rcv 函数的对应代码片段。第一个参数 sk 表示数据包所属套接字对应的 sock 结构；第二个参数 skb 表示当前接收的数据包；其他参数含义自明，需要注意的是 saddr 表示远端地址，daddr 表示的才是本地 IP 地址，这一点从 raw_rcv 函数的调用环境即可看出：

```
/*net/inet/ip.c – ip_rcv 函数代码片段*/
1422        raw_rcv(raw_sk, skb1, dev, iph->saddr,iph->daddr);
```

```
88          /* Now we need to copy this into memory. */
89          skb->sk = sk;
90          skb->len = ntohs(skb->ip_hdr->tot_len);
91          skb->h.raw = (unsigned char *) skb->ip_hdr;
92          skb->dev = dev;
93          skb->saddr = daddr;
94          skb->daddr = saddr;

95          /* Charge it to the socket. */

96          if(sock_queue_rcv_skb(sk,skb)<0)
97          {
98                ip_statistics.IpInDiscards++;
99                skb->sk=NULL;
100               kfree_skb(skb, FREE_READ);
101               return(0);
102         }

103         ip_statistics.IpInDelivers++;
104         release_sock(sk);
105         return(0);
106   }
```

数据包的接收对于 RAW 类型套接字而言非常简单，89-94 行对数据包封装结构 sk_buff 中部分字段进行更新，之后调用 sock_queue_rcv_skb 函数将数据包直接挂接到数据包所属套接字对应 sock 结构的接收队列中（sock 结构 receive_queue 字段指向的队列）。sock_queue_rcv_skb 函数定义在 net/inet/sock.c 文件中，这个函数在本书前文中已经进行了介绍，不过此处为了方便理解，重新给出其实现代码如下。

```
/*net/inet/sock.c -- sock_queue_rcv_skb 函数*/
450   int sock_queue_rcv_skb(struct sock *sk, struct sk_buff *skb)
451   {
```

```
452         unsigned long flags;
453         if(sk->rmem_alloc + skb->mem_len >= sk->rcvbuf)
454                 return -ENOMEM;
455         save_flags(flags);
456         cli();
457         sk->rmem_alloc+=skb->mem_len;
458         skb->sk=sk;
459         restore_flags(flags);
460         skb_queue_tail(&sk->receive_queue,skb);
461         if(!sk->dead)
462                 sk->data_ready(sk,skb->len);
463         return 0;
464 }
```

结合 sock_queue_rcv_skb 函数实现，raw_rcv 函数实现就比较简单。由于 RAW 类型套接字 IP 首部后内容是由用户定制，所以将数据包从 IP 模块传递上来后，直接挂接到接收队列中即可，其他具体的处理由用户应用程序负责。

```
107 /*
108  *    Send a RAW IP packet.
109  */

110 static int raw_sendto(struct sock *sk, unsigned char *from,
111         int len, int noblock, unsigned flags, struct sockaddr_in *usin, int addr_len)
112 {
```

raw_sendto 函数用于 RAW 类型数据包的发送。参数 from 表示将发送数据包 IP 首部后的内容，len 为内容长度，其他字段意义自明。

```
113         struct sk_buff *skb;
114         struct device *dev=NULL;
115         struct sockaddr_in sin;
116         int tmp;
117         int err;

118         /*
119          *    Check the flags. Only MSG_DONTROUTE is permitted.
120          */

121         if (flags & MSG_OOB)        /* Mirror BSD error message compatibility */
122                 return -EOPNOTSUPP;

123         if (flags & ~MSG_DONTROUTE)
124                 return(-EINVAL);
```

121-124 行代码对相关标志位进行检查，首先 RAW 类型套接字不支持 OOB（带外数据）；其次也不支持除了 MSG_DONTROUTE 之外的其他任何标志位。

```
125     /*
126      *  Get and verify the address.
127      */

128     if (usin)
129     {
130         if (addr_len < sizeof(sin))
131             return(-EINVAL);
132         memcpy(&sin, usin, sizeof(sin));
133         if (sin.sin_family && sin.sin_family != AF_INET)
134             return(-EINVAL);
135     }
136     else
137     {
138         if (sk->state != TCP_ESTABLISHED)
139             return(-EINVAL);
140         sin.sin_family = AF_INET;
141         sin.sin_port = sk->protocol;
142         sin.sin_addr.s_addr = sk->daddr;
143     }
```

128 行对 usin 参数进行检查，如果该参数不为 NULL，则表示目的地址临时指定，此时对域类型进行检查，RAW 类型套接字只支持 AF_INET 域，否则无效返回。如果 usin 参数为 NULL，则检查套接字连接状态，注意 RAW 类型套接字连接操作同 UDP 协议，只进行状态设置，不涉及任何数据包的传输，换句话说，RAW 类型套接字同 UDP 协议一样，是尽力传输协议，不提供可靠性数据传输。注意 RAW 类型套接字 sock 结构中 protocol 字段用于标志该 RAW 类型套接字，这个字段的值将被作为 IP 首部中上层协议字段的值，对于 RAW 类型套接字而言，对应的上层协议号为 255，也就是常量 IPPROTO_RAW 表示的值。

```
144     if (sin.sin_port == 0)
145         sin.sin_port = sk->protocol;

146     if (sin.sin_addr.s_addr == INADDR_ANY)
147         sin.sin_addr.s_addr = ip_my_addr();
```

144-147 行连带 141，142 行代码都是对 sin 变量中远端“端口号”和 IP 地址进行设置。

```
148     if (sk->broadcast == 0 && ip_chk_addr(sin.sin_addr.s_addr)==IS_BROADCAST)
149         return -EACCES;
```

```
150     skb=sock_alloc_send_skb(sk, len+sk->prot->max_header, noblock, &err);
151     if(skb==NULL)
152         return err;

153     skb->sk = sk;
154     skb->free = 1;
155     skb->localroute = sk->localroute | (flags&MSG_DONTROUTE);

156     tmp = sk->prot->build_header(skb, sk->saddr,
157                 sin.sin_addr.s_addr, &dev,
158                 sk->protocol, sk->opt, skb->mem_len, sk->ip_tos,sk->ip_ttl);
```

注意对于 RAW 类型套接字，对应 sock 结构中 protocol 字段赋值为 IPPROTO_RAW，这一点非常重要。对于 RAW 类型套接字，raw_prot 中 build_header 指向 ip_build_header 函数，该函数实现代码中有如下检查语句：

```
/*net/inet/ip.c -- ip_build_header 函数片段*/
283     /*
284      *  If we are using IPPROTO_RAW, then we don't need an IP header, since
285      *  one is being supplied to us by the user
286      */

287     if(type == IPPROTO_RAW)
288         return (tmp);
```

即对于 RAW 类型套接字，ip_build_header 函数在完成 MAC 首部创建后将直接返回，不进行 IP 首部的创建，因为对于 RAW 类型套接字而言，IP 首部的创建是由用户完成的，即 raw_sendto 函数参数 from 指向的缓冲区中数据应该包含一个 IP 首部，之后跟随其他数据。所以 156 行 ip_build_header 函数返回值仅为 MAC 首部的长度。

```
159     if (tmp < 0)
160     {
161         kfree_skb(skb,FREE_WRITE);
162         release_sock(sk);
163         return(tmp);
164     }

165     memcpy_fromfs(skb->data + tmp, from, len);
```

如果 ip_build_header 函数返回值小于 0，表示 MAC 首部创建失败，则放弃该数据包的发送。否则直接从用户缓冲区拷贝数据到 sk_buff 封装结构中，注意用户数据被复制到 MAC 首部之后，所以用户缓冲区中数据必须包含 IP 首部，即 RAW 类型套接字，由用户直接进行 IP 首部的创建，这也是设计 RAW 类型套接字的目的—可以直接操作 IP 首部字段值以满足特殊的应用。

```
166     /*
167      *  If we are using IPPROTO_RAW, we need to fill in the source address in
168      *      the IP header
169      */

170     if(sk->protocol==IPPROTO_RAW)
171     {
172         unsigned char *buff;
173         struct iphdr *iph;

174         buff = skb->data;
175         buff += tmp;

176         iph = (struct iphdr *)buff;
177         iph->saddr = sk->saddr;
178     }
```

170-178 行代码更改本地发送端的 IP 地址，这将从内核角度取消本地 IP 地址的伪装。因为恶意用户可能将 IP 首部中源端 IP 地址设置为非本机地址，用于攻击其他主机，以免被发现。177 行直接取消了用户的这一恶意行为。所以 177 行代码并不是为了防止用户错误设置本地 IP 地址，而是为了防止用户恶意设置本地 IP 地址。

```
179     skb->len = tmp + len;

180     sk->prot->queue_xmit(sk, dev, skb, 1);
181     release_sock(sk);
182     return(len);
183 }
```

函数 179 更新 skb->len 字段为数据包数据帧长度，之后调用 ip_queue_xmit 函数（180 行）将数据包将数据包发往下层处理，从而将数据包最终发送到目的远端。

在完成对 raw_sendto 函数的分析后，我们再次提请注意：对于 RAW 类型套接字，IP 首部的创建将由用户完成，并和普通数据一起交给内核，内核只负责进行 MAC 首部的创建，之后内核将用户缓冲区中数据直接复制到 MAC 首部之后，所以如果使用 RAW 类型套接字，而没有正确创建 IP 首部或者根本没有进行 IP 首部的创建，则数据包将无法发送到目的远端。

```
184 static int raw_write(struct sock *sk, unsigned char *buff, int len, int noblock,
185         unsigned flags)
186 {
187     return(raw_sendto(sk, buff, len, noblock, flags, NULL, 0));
188 }
```

RAW 类型套接字是面向报文的，其使用的传输策略如同 UDP 协议，不提供可靠性数据传输，而且没有连接建立过程，每次进行数据发送时都临时指定目的远端地址，当然为了减少这种冗余的地址指定，对于此类套接字也可以进行 connect 系统调用，先指定远端地址，此后直接使用 write 函数进行数据发送。raw_write 就是在之前进行了 connect 系统调用，在已指定远端地址的情况下，被调用进行数据发送。从下文对 raw_prot 变量的初始化来看，RAW 类型套接字 connect 系统调用的底层实现函数使用 UDP 协议的 udp_connect 函数实现，udp_connect 函数用于提前指定目的远端地址，更改相关状态变量，不涉及任何实际数据包的发送（不同于 TCP 协议建立连接的过程）。因为在调用 raw_write 之前，已经进行了远端目的地址指定，所以在进行 raw_sendto 函数的封装调用时，就没有再指定远端地址，最后两个参数设置为 NULL 和 0。不过这就引申出另一个问题，既然对于 RAW 类型套接字，内核并不负责进行 IP 首部创建，那么预先指定地址和临时指定地址有何不同，又有何意义？首先要回答的是，预先指定地址和临时指定地址对内核而言没有任何本质上的区别，预先指定地址只是从用户角度上为用户提供方便性而已，仅此而已；诚如上文所述，对于 RAW 类型套接字内核不需要进行 IP 首部创建，又不需要端口号，那么指定远端目的地址（无论采用何种方式）的意义何在，意义在于 MAC 首部创建时需要使用 IP 地址。因为硬件 MAC 地址是从根据 IP 地址产生的（一般要使用 ARP 协议进行映射），读者可查看 raw_sendto 函数中对 ip_builder_header 函数调用时传入的第三个参数，该参数表示远端 IP 地址，在创建 MAC 首部时将被用于创建目的端 MAC 硬件地址。

```
189  static void raw_close(struct sock *sk, int timeout)
190  {
191          sk->state = TCP_CLOSE;
192  }
```

由于本质上没有连接建立的概念，当然也就不存在断开连接，所以对于关闭操作，只是更改相关状态位即可。注意只有面向连接的协议（如 TCP 协议）才真正有连接建立一说，其他面向报文的协议（如 UDP，RAW），也有类似于连接建立，断开连接的函数，但这些函数只是从概念上模拟面向连接的协议，从本质上而言，他们并无这一操作要求，所以这些函数实现也仅局限于在本地进行相关变量的修改更新，不涉及任何数据包的实际发送。

```
193  static int raw_init(struct sock *sk)
194  {
195          return(0);
196  }
```

raw_init 函数将在 inet_create(af_inet.c)中被调用，inet_create 函数创建一个对应类型的新的套接字，由于每次创建一个新的套接字时，inet_create 函数都要调用一次，所以 raw_init 之类的函数实现代码应该是单个套接字相关的，而单个套接字相关的初始化代码在 inet_create 函数中已经完成，所以如 raw_init 之类的函数几无用武之地，所以 raw_init 函数直接返回，而 TCP，UDP 协议根本就不提供这样的一个函数。

```
197  /*
198   *    This should be easy, if there is something there
```

```
199   *    we return it, otherwise we block.
200   */

201  int raw_recvfrom(struct sock *sk, unsigned char *to, int len,
202           int noblock, unsigned flags, struct sockaddr_in *sin,
203                   int *addr_len)
204  {
```

raw_recvfrom 函数是对上层 recvfrom 系统调用的底层实现，用于 RAW 类型套接字的数据读取。从其如下的实现我们可以看出，RAW 类型套接字读取方式如同 UDP 类型套接字，是面向报文的，单次只可读取一个报文，最多可读取的数据量为当前被读取数据包中所包含数据的数量，单次不可跨越报文边界进行读取；如果读取的数据量小于当前被读取数据包中包含的数据量，则多余的数据将被丢弃，不会留到下次读取时返回给用户。

```
205       int copied=0;
206       struct sk_buff *skb;
207       int err;
208       int truesize;

209       if (flags & MSG_OOB)
210             return -EOPNOTSUPP;

211       if (sk->shutdown & RCV_SHUTDOWN)
212             return(0);

213       if (addr_len)
214             *addr_len=sizeof(*sin);

215       skb=skb_recv_datagram(sk,flags,noblock,&err);
216       if(skb==NULL)
217             return err;

218       truesize=skb->len;
219       copied = min(len, truesize);

220       skb_copy_datagram(skb, 0, to, copied);
221       sk->stamp=skb->stamp;

222       /* Copy the address. */
223       if (sin)
224       {
225             sin->sin_family = AF_INET;
226             sin->sin_addr.s_addr = skb->daddr;
227       }
```

```
228         skb_free_datagram(skb);
229         release_sock(sk);
230         return (truesize);     /* len not copied. BSD returns the true size of the message so you know a bit fell off! */
231  }
```

raw_recvfrom 函数实现比较简单，我们不再一一进行分析，函数实现中涉及到几个对其他函数的调用，虽然根据上下文，我们可以推测出这些函数完成的功能，但如果没有具体的代码，还是有些不放心，这些函数定义在 datagram.c 文件中，我们在本书前文中已经进行了介绍，此处我们对这些其他函数简单进行一下说明。首先是 skb_recv_datagram 函数，该函数从对应套接字 sock 结构接收队列中取数据包，注意第三个参数用于决定在暂时无数据包可取时，是否进行睡眠等待。其次是 skb_copy_datagram，该函数用于数据复制，从内核缓冲区到用户缓冲区，注意对复制数据长度的设置，为用户请求数据和数据包中包含数据的较小值。第三是 skb_free_datagram 函数，该函数对数据包封装结构进行释放（注意内核数据缓冲区作为该结构的一部分也被释放）。

```
232  int raw_read (struct sock *sk, unsigned char *buff, int len, int noblock,unsigned flags)
233  {
234         return(raw_recvfrom(sk, buff, len, noblock, flags, NULL, NULL));
235  }
```

raw_read 函数类似于 raw_write 函数，其封装对 raw_recvfrom 函数的调用，所基于的思想同 raw_write 函数的实现初衷，这一点在前文中已经进行了较为详细的说明，此处不再讨论。

raw.c 文件最后的代码定义了 raw_prot 这样一个全局变量，该变量表示了 RAW 类型套接字操作函数集合，作用如同 tcp_prot 之于 TCP 协议，udp_prot 之于 UDP 协议，他们都是 proto 类型的变量，所有使用对应协议传输的套接字都使用同一个以上变量，如所有使用 RAW 类型的套接字，都是用 raw_prot 变量，此时套接字对应 sock 结构 prot 字段被赋值为 raw_prot，在进行相关调用时，总会有如下形式的代码（sk 表示 sock 结构类型的变量）：

```
sk->prot->write(…)
```

这个代码对于 RAW 类型套接字即调用 raw_write，对于 TCP 则对应于 tcp_write，而对于 UDP 则对应于 udp_write，诸如此类。

```
236  struct proto raw_prot = {
237         sock_wmalloc,
238         sock_rmalloc,
239         sock_wfree,
240         sock_rfree,
241         sock_rspace,
242         sock_wspace,
243         raw_close,
244         raw_read,
245         raw_write,
246         raw_sendto,
```

```
247         raw_recvfrom,
248         ip_build_header,
249         udp_connect,
250         NULL,
251         ip_queue_xmit,
252         NULL,
253         NULL,
254         NULL,
255         NULL,
256         datagram_select,
257         NULL,
258         raw_init,
259         NULL,
260         ip_setsockopt,
261         ip_getsockopt,
262         128,
263         0,
264         {NULL,},
265         "RAW",
266         0, 0
267  };
```

对于 raw_prot 变量的定义，有一个字段的赋值需要提请注意：如 udp_connect, ip_builder_header, raw_init。

raw.c 文件小结

该文件是 RAW 类型套接字实现文件，从其实现代码中我们可以得出如下几点：

1〉 使用 RAW 类型套接字，应用层必须自行进行 IP 首部的创建。

2〉 由于 IP 首部的自定义，用户可能会进行本地 IP 地址伪装（使用非本机的 IP 地址），用于攻击目的，这一恶意操作被内核自动取消（见 raw_sendto 函数实现）。所以如果想使用 RAW 类型套接字进行此类攻击，即便在应用层随便使用了一个非本机 IP 地址，数据包发出本机时，IP 首部中相应字段还是本地主机地址，所以不可以为成功进行了伪装，你暴露给远端接收主机的依然是你自己主机的 IP 地址。

3〉 RAW 类型套接字如同 UDP 协议只提供面向报文的，不可靠数据传输方式。

4〉 RAW 类型套接字数据读取方式如同 UDP，每次只能读取一个数据包，不可跨越多个数据包进行读取，单次为读完的数据将被丢弃。

RAW 类型套接字总体实现上比较简单，简单的原因在以上 1〉，3〉，4〉所述中，RAW 类型套接字的主要应用有 traceroute 程序。

2.24 net/inet/raw.h 文件

该头文件对应于 raw.c 文件，所以只是对相关函数原型的声明。此处给出代码，不做讨论。

```
1     /*
2      * INET         An implementation of the TCP/IP protocol suite for the LINUX
3      *              operating system.    INET is implemented using the    BSD Socket
```

```
4       *               interface as the means of communication with the user level.
5       *
6       *               Definitions for the RAW-IP module.
7       *
8       * Version:      @(#)raw.h       1.0.2       05/07/93
9       *
10      * Author:       Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
11      *
12      *               This program is free software; you can redistribute it and/or
13      *               modify it under the terms of the GNU General Public License
14      *               as published by the Free Software Foundation; either version
15      *               2 of the License, or (at your option) any later version.
16      */
17      #ifndef _RAW_H
18      #define _RAW_H


19      extern struct proto raw_prot;


20      extern void     raw_err(int err, unsigned char *header, unsigned long daddr,
21                              unsigned long saddr, struct inet_protocol *protocol);
22      extern int      raw_recvfrom(struct sock *sk, unsigned char *to,
23                              int len, int noblock, unsigned flags,
24                              struct sockaddr_in *sin, int *addr_len);
25      extern int      raw_read(struct sock *sk, unsigned char *buff,
26                              int len, int noblock, unsigned flags);
27      extern int      raw_rcv(struct sock *, struct sk_buff *, struct device *,
28                              long, long);

29      #endif  /* _RAW_H */
```

在完成对 RAW 类型套接字实现分析后，我们将进入到另一个类似的套接字类型，这种套接字类型比 RAW 类型更为底层，因为此种套接字类型直接使用链路层接口函数（dev_queue_xmit）进行数据包的发送（RAW 类型套接字使用网络层接口函数 ip_queue_xmit），换句话说，使用此种类型套接字，应用层必须完成整个数据帧的创建，即完成 MAC 首部，IP 首部和 IP 负载的创建。下面我们就对该类型套接字实现文件进行分析。从其实现文件来看，我们将此种套接字类型称为 PACKET 类型套接字，注意区别于 SEQPACKET 类型套接字，SEQPACKET 使用 TCP 协议，是面向连接的，并提供可靠性数据传输，使用 tcp_prot 表示的操作函数集。在当前版本网络栈实现中，SEQPACKET 类型套接字用 SOCK_SEQPACKET 表示，其等同于 SOCK_STEAM，即实现文件也为 tcp.c；而 PACKET 类型套接字实现文件为 packet.c，面向报文，不提供可靠性数据传输，读取方式如同 UDP 协议，操作函数集合由 packet_prot 表示。下面我们即对 PACKET 类型套接字实现文

件 packet.c 进行分析。

2.25 net/inet/packet.c 文件

在分析 PACKET 类型套接字具体实现之前，我们需要对该类型套接字的数据包接收和发送方式进行一下说明。诚如上文所述，使用 PACKET 类型套接字时，应用层必须完成整个数据帧的创建，并直接调用链路层接口函数（dev_queue_xmit）将数据包发送出去。在接收端，按照 PACKET 类型套接字工作原理，应用层也应该进行整个数据帧的接收，但是在将数据据包挂接到套接字接收队列之前，需要通过链路层接口函数（net_bh），只要经过链路层处理，其都会剥离 MAC 首部，为了按要求将整个数据帧传递给应用层，PACKET 类型套接字实现又重新加上了 MAC 首部（注意 MAC 首部的剥离和加上主要是通过变更相关变量值完成的，并无数据的复制和重新创建，这一点在 net_bh, packet_rcv 函数中体现）。另外由于接收时需要经过链路层模块，所以 PACKET 套接字类型必须提供一个接口供链路层模块调用，在这一点上，其采用同 IP 协议相同的方式，使用 packet_type 结构进行回调函数的封装，并将这个 packet_type 结构加入到系统变量 ptype_base 指向的队列中，链路层实现模块在将数据包传递给上层模块时，将遍历由 ptype_base 变量指向的队列，寻找合适的 packet_type 结构，调用其中的回调函数（如 packet_rcv, ip_rcv），将数据包从链路层模块传递给上层（网络层）模块。对于以上的说明，下面我们将从如下的实际代码分析中得到证明。首先我们给出 net_bh(dev.c)函数实现代码片段，看其如何剥离 MAC 首部以及如何将数据包传递给上层模块。

```
/*net/inet/dev.c—net_bh 函数代码片段*/
565             skb->h.raw = skb->data + skb->dev->hard_header_len;
566             skb->len -= skb->dev->hard_header_len;
```

565-566 行代码剥离 MAC 首部，注意 skb->data 字段值始终不变，该字段始终指向整个数据帧的起始处。566 行将 skb->len 减去了 MAC 首部长度，这就是在进行 MAC 首部剥离。

```
/* 567 – 576  为注释部分，省略*/
577             type = skb->dev->type_trans(skb, skb->dev);

/* 578 – 588  为注释部分，省略*/
589             pt_prev = NULL;
590             for (ptype = ptype_base; ptype != NULL; ptype = ptype->next)
591             {
592                 if ((ptype->type == type || ptype->type == htons(ETH_P_ALL)) &&
                                                (!ptype->dev || ptype->dev==skb->dev))
593                 {
594                     /*
595                       *   We already have a match queued. Deliver
596                       *   to it and then remember the new match
597                       */
598                     if(pt_prev)
599                     {
600                         struct sk_buff *skb2;
601                         skb2=skb_clone(skb, GFP_ATOMIC);
```

```
602                             /*
603                              *  Kick the protocol handler. This should be fast
604                              *  and efficient code.
605                              */
606                             if(skb2)
607                                 pt_prev->func(skb2, skb->dev, pt_prev);
608                         }
609                         /* Remember the current last to do */
610                         pt_prev=ptype;
611                     }
612                 } /* End of protocol list loop */
613                 /*
614                  *  Is there a last item to send to ?
615                  */
616                 if(pt_prev)
617                     pt_prev->func(skb, skb->dev, pt_prev);
```

577-617 行代码将数据包传递给上层模块，注意对 ptype_base 变量指向的队列的遍历。592 行对队列中每个 packet_type 结构类型的元素进行检查，寻找到合适的 packet_type 结构，从而对其中注册的接收函数进行调用，将数据包从链路层模块传递给上层（网络层）模块。针对以上链路层中实现代码，下面我们查看 PACKET 类型套接字实现的应对方式，首先必须重新加上 MAC 首部，其次必须创建一个表示 PACKET 类型套接字的 packet_type 结构并将其加入到 ptype_base 指向的队列中。这两个工作分别在 packet_rcv, packet_init 函数中完成，下文中介绍到这两个函数时，请读者注意留意。

/*net/inet/packet.c*/

```
1   /*
2    * INET        An implementation of the TCP/IP protocol suite for the LINUX
3    *        operating system.    INET is implemented using the    BSD Socket
4    *        interface as the means of communication with the user level.
5    *
6    *        PACKET - implements raw packet sockets.
7    *
8    * Version:    @(#)packet.c  1.0.6       05/25/93
9    *
10   * Authors:    Ross Biro, <bir7@leland.Stanford.Edu>
11   *        Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
12   *        Alan Cox, <gw4pts@gw4pts.ampr.org>
13   *
14   * Fixes:
15   *        Alan Cox:    verify_area() now used correctly
16   *        Alan Cox:    new skbuff lists, look ma no backlogs!
17   *        Alan Cox:    tidied skbuff lists.
18   *        Alan Cox:    Now uses generic datagram routines I
```

```
19    *                       added. Also fixed the peek/read crash
20    *                       from all old Linux datagram code.
21    *          Alan Cox:    Uses the improved datagram code.
22    *          Alan Cox:    Added NULL's for socket options.
23    *          Alan Cox:    Re-commented the code.
24    *          Alan Cox:    Use new kernel side addressing
25    *          Rob Janssen  :    Correct MTU usage.
26    *          Dave Platt   :    Counter leaks caused by incorrect
27    *                       interrupt locking and some slightly
28    *                       dubious gcc output. Can you read
29    *                       compiler: it said _VOLATILE_
30    *
31    *          This program is free software; you can redistribute it and/or
32    *          modify it under the terms of the GNU General Public License
33    *          as published by the Free Software Foundation; either version
34    *          2 of the License, or (at your option) any later version.
35    *
36    */

37    #include <linux/types.h>
38    #include <linux/sched.h>
39    #include <linux/mm.h>
40    #include <linux/fcntl.h>
41    #include <linux/socket.h>
42    #include <linux/in.h>
43    #include <linux/inet.h>
44    #include <linux/netdevice.h>
45    #include "ip.h"
46    #include "protocol.h"
47    #include <linux/skbuff.h>
48    #include "sock.h"
49    #include <linux/errno.h>
50    #include <linux/timer.h>
51    #include <asm/system.h>
52    #include <asm/segment.h>

53    /*
54     *    We really ought to have a single public _inline_ min function!
55     */

56    static unsigned long min(unsigned long a, unsigned long b)
57    {
58          if (a < b)
59                return(a);
```

```
60          return(b);
61  }


62  /*
63   *  This should be the easiest of all, all we do is copy it into a buffer.
64   */

65  int packet_rcv(struct sk_buff *skb, struct device *dev,   struct packet_type *pt)
66  {
```

packet_rcv 被注册为 PACKET 类型套接字的接收函数，该函数将被链路层模块函数（net_bh）调用，进行 PACKET 套接字类型数据包的接收。调用方式是 PACKET 类型套接字实现代码通过向 ptype_base 指向的队列中注册 packet_type 结构完成的。这个注册过程具体实现在 packet_init 函数中。

```
67          struct sock *sk;
68          unsigned long flags;

69          /*
70           *  When we registered the protocol we saved the socket in the data
71           *  field for just this event.
72           */

73          sk = (struct sock *) pt->data;

74          /*
75           *  The SOCK_PACKET socket receives _all_ frames, and as such
76           *  therefore needs to put the header back onto the buffer.
77           *  (it was removed by inet_bh()).
78           */

79          skb->dev = dev;
80          skb->len += dev->hard_header_len;
```

80 行代码取消链路层模块对 MAC 首部的剥离，又重新加上了 MAC 首部，这行代码应和 packet_recvfrom 函数中如下代码进行结合后一起看待：

```
/*packet_recvfrom 代码片段*/
267         truesize = skb->len;
268         copied = min(len, truesize);
269         memcpy_tofs(to, skb->data, copied);     /* We can't use skb_copy_datagram here */
```

由于 packet_rcv 函数实现中（以上第 80 行代码）将 skb->len 重新包含进 MAC 首部，而 skb->data 始终指向整个数据帧首部，所以 269 行代码保证了将整个数据帧传递给应用层。

```
81          /*
82           *  Charge the memory to the socket. This is done specifically
83           *  to prevent sockets using all the memory up.
84           */

85          if (sk->rmem_alloc & 0xFF000000) {
86                  printk("packet_rcv: sk->rmem_alloc = %ld\n", sk->rmem_alloc);
87                  sk->rmem_alloc = 0;
88          }

89          if (sk->rmem_alloc + skb->mem_len >= sk->rcvbuf)
90          {
91  /*                  printk("packet_rcv:  drop,  %d+%d>%d\n",  sk->rmem_alloc,  skb->mem_len,
sk->rcvbuf); */
92                  skb->sk = NULL;
93                  kfree_skb(skb, FREE_READ);
94                  return(0);
95          }

96          save_flags(flags);
97          cli();

98          skb->sk = sk;
99          sk->rmem_alloc += skb->mem_len;

100         /*
101          *  Queue the packet up, and wake anyone waiting for it.
102          */

103         skb_queue_tail(&sk->receive_queue,skb);
104         if(!sk->dead)
105                 sk->data_ready(sk,skb->len);

106         restore_flags(flags);

107         /*
108          *  Processing complete.
109          */

110         release_sock(sk);    /* This is now effectively surplus in this layer */
111         return(0);
112 }
```

packet_rcv 函数其他部分实现都比较简单，此处不再讨论，103 行将数据包挂接到对应套接字的接收队列中。

```
113 /*
114  *  Output a raw packet to a device layer. This bypasses all the other
115  *  protocol layers and you must therefore supply it with a complete frame
116  */

117 static int packet_sendto(struct sock *sk, unsigned char *from, int len,
118                 int noblock, unsigned flags, struct sockaddr_in *usin,
119                 int addr_len)
120 {
```

packet_sendto 函数用于 PACKET 类型套接字的数据发送，参数设置同 UDP，RAW 类型套接字，此处不再作多余说明。PACKET 类型套接字提供面向报文的尽力传输服务。

```
121     struct sk_buff *skb;
122     struct device *dev;
123     struct sockaddr *saddr=(struct sockaddr *)usin;

124     /*
125      *  Check the flags.
126      */

127     if (flags)
128         return(-EINVAL);

129     /*
130      *  Get and verify the address.
131      */

132     if (usin)
133     {
134         if (addr_len < sizeof(*saddr))
135             return(-EINVAL);
136     }
137     else
138         return(-EINVAL);   /* SOCK_PACKET must be sent giving an address */

139     /*
140      *  Find the device first to size check it
141      */
```

```
142        saddr->sa_data[13] = 0;
```

变量 saddr 是 sockaddr 结构类型，该结构定义在 include/linux/socket.h 中，如下所示：

```
/*include/linux/socket.h*/
4     struct sockaddr {
5         unsigned short    sa_family;      /* address family, AF_xxx     */
6         char              sa_data[14];    /* 14 bytes of protocol address      */
7     };
```

其中 sa_data 字段在通常情况下（如使用 TCP，UDP 协议时）表示端口号以及 IP 地址值，对于 PACKET 类型套接字由于由应用层创建整个数据帧，所以内核不需要这些参数值，由于需要直接实现链路层模块接口函数进行数据发送，所以该参数被用于指定发送设备接口，即 sa_data 字段用于指定网络设备接口。142 行代码将最后一个字节清零，从而创建一个以 NULL 结尾的标准 ASCII 字符串，用于 143 行函数调用：从设备名称寻找对应设备的 device 结构，dev_get 函数遍历 dev_base 队列，比较设备名，返回给定设备名称的 device 结构。注意对于 PACKET 类型套接字必须进行发送设备的指定，即 packet_sendto 函数调用中 usin 参数不可为 NULL，否则错误返回（137 行），这一点不同于 UDP，RAW 类型套接字。

```
143        dev = dev_get(saddr->sa_data);
144        if (dev == NULL)
145        {
146             return(-ENXIO);
147        }

148        /*
149          *    You may not queue a frame bigger than the mtu. This is the lowest level
150          *    raw protocol and you must do your own fragmentation at this level.
151          */

152        if(len>dev->mtu+dev->hard_header_len)
153             return -EMSGSIZE;
```

PACKET 类型套接字在本版本实现中不支持数据包分片功能，如果数据包过大，则返回 EMSGSIZE 错误。

```
154        skb = sk->prot->wmalloc(sk, len, 0, GFP_KERNEL);

155        /*
156          *    If the write buffer is full, then tough. At this level the user gets to
157          *    deal with the problem - do your own algorithmic backoffs.
158          */

159        if (skb == NULL)
160        {
161             return(-ENOBUFS);
```

```
162         }

163         /*
164          *    Fill it in
165          */

166         skb->sk = sk;
167         skb->free = 1;
168         memcpy_fromfs(skb->data, from, len);
169         skb->len = len;
170         skb->arp = 1;         /* No ARP needs doing on this (complete) frame */
```

154-170 行代码完成数据包封装结构的创建以及数据帧的复制。注意代码中没有任何首部的创建代码，直接在 168 行将用户数据复制到内核封装结构中，并在下面的 175 行直接调用 dev_queue_xmit 这个链路层模块接口函数将数据包发送出去。这与我们在前文中一再声称的对于 PACKET 类型套接字有应用层完成整个数据帧的创建是一致的。170 行将 skb->arp 设置为 1，表示 MAC 首部成功创建，无需使用 ARP 协议进行地址解析。167 行将 skb->free 字段设置为 1，表示 PACKET 类型套接字不提供可靠性数据传输。

```
171         /*
172          *    Now send it
173          */

174         if (dev->flags & IFF_UP)
175               dev_queue_xmit(skb, dev, sk->priority);
176         else
177               kfree_skb(skb, FREE_WRITE);
178         return(len);
179   }
```

packet_sendto 函数最后调用 dev_queue_xmit 函数将数据包直接传递给链路层模块进行处理，并最终将数据包发送出去。注意如果指定的网络设备当前处于非工作状态，则简单丢弃该数据包。这也是 PACKET 类型套接字不提供可靠性数据传输的体现之一。

```
180   /*
181    *    A write to a SOCK_PACKET can't actually do anything useful and will
182    *    always fail but we include it for completeness and future expansion.
183    */

184   static int packet_write(struct sock *sk, unsigned char *buff,
185                 int len, int noblock,    unsigned flags)
186   {
187         return(packet_sendto(sk, buff, len, noblock, flags, NULL, 0));
188   }
```

packet_write 函数直接封装对 packet_sendto 的调用，注意在分析 packet_sendto 函数时，我们得出表示远端地址的参数（最后两个参数）不可为 NULL，因为这个参数被用来指定发送设备，如果为 NULL，则无效返回。所以此处 packet_write 函数在调用 packet_sendto 函数时将地址参数设置为 NULL，将返回一个 EINVAL 错误，换句话说，PACKET 类型套接字不支持 write 系统调用，只可使用 sendto 系统调用，且必须在地址参数中指定网络发送设备。

```
189  /*
190   *    Close a SOCK_PACKET socket. This is fairly simple. We immediately go
191   *    to 'closed' state and remove our protocol entry in the device list.
192   *    The release_sock() will destroy the socket if a user has closed the
193   *    file side of the object.
194   */

195  static void packet_close(struct sock *sk, int timeout)
196  {
197          sk->inuse = 1;
198          sk->state = TCP_CLOSE;
199          dev_remove_pack((struct packet_type *)sk->pair);
200          kfree_s((void *)sk->pair, sizeof(struct packet_type));
201          sk->pair = NULL;
202          release_sock(sk);
203  }
```

packet_close 函数的作用并不是简单的关闭一个 PACKET 类型套接字，因为 PACKET 类型套接字是面向报文，其本身并不存在关闭操作。此处的关闭函数实现为将 PACKET 类型套接字对应的 packet_type 结构从 ptype_base 指向的队列中删除。具体由 dev_remove_pack 函数完成从 ptype_base 指向的队列中删除函数参数指定的 packet_type 结构元素，sk->pair 被初始化为指向对应的 packet_type 结构，这个初始化是在 packet_init 函数中完成的。读者可能对此有所疑虑，如果将 PACKET 类型套接字对应的 packet_type 结构从 ptype_base 指向的队列中删除，那么其他使用 PACKET 协议的套接字岂非接收不到任何数据包。这一点不用担心，对于这个的解释需要综合来看，我们首先完成对下面 packet_init 函数的介绍，在解释其中的原因。

```
204  /*
205   *    Create a packet of type SOCK_PACKET. We do one slightly irregular
206   *    thing here that wants tidying up. We borrow the 'pair' pointer in
207   *    the socket object so we can find the packet_type entry in the
208   *    device list. The reverse is easy as we use the data field of the
209   *    packet type to point to our socket.
210   */

211  static int packet_init(struct sock *sk)
212  {
```

```
213     struct packet_type *p;

214     p = (struct packet_type *) kmalloc(sizeof(*p), GFP_KERNEL);
215     if (p == NULL)
216         return(-ENOMEM);

217     p->func = packet_rcv;
218     p->type = sk->num;
219     p->data = (void *)sk;
220     p->dev = NULL;
221     dev_add_pack(p);

222     /*
223      *  We need to remember this somewhere.
224      */

225     sk->pair = (struct sock *)p;

226     return(0);
227 }
```

packet_init 函数在 inet_create 函数中当创建一个新的使用 PACKET 协议的套接字时被调用，每次创建时都被调用一次，我们看 packet_init 函数实现代码，其创建一个新的 packet_type 结构，将其加入到 ptype_base 变量指向的队列中（通过 dev_add_pack 函数），从而向链路层模块注册数据包接收函数。换句话说，每个使用 PACKET 协议的套接字在 ptype_base 队列中都对应有一个 packet_type 结构，所以 packet_close 函数中删除套接字对应的 packet_type 结构，并不会对其他使用 PACKET 协议的套接字产生影响，注意 225 行初始化代码，这个代码保证了在 packet_close 函数中删除的是对应的 packet_type 结构。注意 218 行代码将 packet_type 结构中 type 字段设置为 sk->num, sk->num 在 inet_create 函数中被初始化为 socket 系统调用时传入的第三个参数，可以用以区分使用 PACKET 协议的不同套接字。packet_type 结构中 type 字段在 net_bh 函数中被用于在 ptype_base 队列中寻找合适的 packet_type 结构，所以可以用于区分使用 PACKET 协议的不同套接字，保证接收的数据包传递给正确的用户。到此，我们完成在 packet_close 函数种遗留问题的解释，即 packet_close 函数实现方式不会对其他使用 PACKET 协议的套接字造成影响。

```
228 /*
229  *  Pull a packet from our receive queue and hand it to the user.
230  *  If necessary we block.
231  */

232 int packet_recvfrom(struct sock *sk, unsigned char *to, int len,
233                     int noblock, unsigned flags, struct sockaddr_in *sin,
234                     int *addr_len)
```

```
235 {
```

packet_recvfrom 函数是 PACKET 协议数据读取函数，是对上层应用层 sendto 系统调用的底层实现，该读取函数返回整个数据帧供应用层处理。

```
236     int copied=0;
237     struct sk_buff *skb;
238     struct sockaddr *saddr;
239     int err;
240     int truesize;

241     saddr = (struct sockaddr *)sin;

242     if (sk->shutdown & RCV_SHUTDOWN)
243         return(0);

244     /*
245      *  If the address length field is there to be filled in, we fill
246      *  it in now.
247      */

248     if (addr_len)
249         *addr_len=sizeof(*saddr);

250     /*
251      *  Call the generic datagram receiver. This handles all sorts
252      *  of horrible races and re-entrancy so we can forget about it
253      *  in the protocol layers.
254      */

255     skb=skb_recv_datagram(sk,flags,noblock,&err);

256     /*
257      *  An error occurred so return it. Because skb_recv_datagram()
258      *  handles the blocking we don't see and worry about blocking
259      *  retries.
260      */

261     if(skb==NULL)
262         return err;

263     /*
264      *  You lose any data beyond the buffer you gave. If it worries a
265      *  user program they can ask the device for its MTU anyway.
```

```
266         */

267         truesize = skb->len;
268         copied = min(len, truesize);

269         memcpy_tofs(to, skb->data, copied);     /* We can't use skb_copy_datagram here */
```

255 行从套接字接收队列中取数据包，注意 skb->len 在 packet_rcv 函数中被重新设置为包含 MAC 首部长度，从而 267-269 行代码保证了传递给应用层的是整个数据帧，当然如果应用层请求数据长度较短，则可能丢弃一部分数据，但 MAC，IP 首部都预先提供（因为他们位于数据帧的最前端）。

```
270         /*
271          *  Copy the address.
272          */

273         if (saddr)
274         {
275                 saddr->sa_family = skb->dev->type;
276                 memcpy(saddr->sa_data,skb->dev->name, 14);
277         }
```

从 273-277 行代码可见，对于 PACKET 类型套接字，返回的地址信息为接收该数据包的网络接口设备名称，这一点与 TCP，UDP 协议返回的是端口号和 IP 地址不同，但这与 packet_sendto 函数相对应起来。

```
278         /*
279          *  Free or return the buffer as appropriate. Again this hides all the
280          *  races and re-entrancy issues from us.
281          */

282         skb_free_datagram(skb);

283         /*
284          *  We are done.
285          */

286         release_sock(sk);
287         return(truesize);
288 }
```

注意函数最后返回的是数据帧的整个长度，而非用户请求的数据量，从而根据返回值用户可以判断是否少读了数据。

```
289 /*
290  *  A packet read can succeed and is just the same as a recvfrom but without the
291  *  addresses being recorded.
292  */

293 int packet_read(struct sock *sk, unsigned char *buff,
294                 int len, int noblock, unsigned flags)
295 {
296         return(packet_recvfrom(sk, buff, len, noblock, flags, NULL, NULL));
297 }
```

packet_read 函数对应 packet_write 函数，不过不同于 packet_write 函数，packet_read 函数将地址参数设置为 NULL 是合法的，而 packet_write 函数则是无效的。函数实现就是对 packet_recvfrom 函数进行封装。

packet.c 文件最后定义了表示 PACKET 协议操作函数集合的变量 packet_prot, 该变量作用应该不用多说，前面我们累计已经介绍了 tcp_prot, udp_prot, raw_prot，这些变量所表示的意义都相同，都是 proto 结构类型，读者对照 proto 结构定义，可以查看各字段的初始化情况（根据以下赋值的函数名我们也可以看出对应函数的作用）。

```
298 /*
299  *  This structure declares to the lower layer socket subsystem currently
300  *  incorrectly embedded in the IP code how to behave. This interface needs
301  *  a lot of work and will change.
302  */

303 struct proto packet_prot =
304 {
305         sock_wmalloc,
306         sock_rmalloc,
307         sock_wfree,
308         sock_rfree,
309         sock_rspace,
310         sock_wspace,
311         packet_close,
312         packet_read,
313         packet_write,
314         packet_sendto,
315         packet_recvfrom,
316         ip_build_header,     /* Not actually used */
317         NULL,
318         NULL,
319         ip_queue_xmit,              /* These two are not actually used */
```

```
320         NULL,
321         NULL,
322         NULL,
323         NULL,
324         datagram_select,
325         NULL,
326         packet_init,
327         NULL,
328         NULL,               /* No set/get socket options */
329         NULL,
330         128,
331         0,
332         {NULL,},
333         "PACKET",
334         0, 0
335  };
```

packet.c 文件小结

该文件是 PACKET 协议实现文件，PACKET 协议用于特殊应用，可以直接操作数据帧的任何部分，包括 MAC 首部，IP 首部的创建，当然原则上我们可以将封装 TCP 或者 UDP 首部，即数据包的格式完全由用户定制（当然为了保证数据包的传输，必须具有 MAC，IP 首部）。综上所述，我们可以总结 PACKET 协议如下：

1〉 使用 PACKET 类型套接字，应用层必须自行进行整个数据帧的创建，包括 MAC 首部，IP 首部以及数据负载部分。

2〉 由于 IP 首部的自定义，用户可能会进行本地 IP 地址伪装（使用非本机的 IP 地址），用于攻击目的；同样由于 MAC 首部的自定义，用户可以进行源 MAC 地址的任意指定，原则上这不会影响数据包的发送，如果用于攻击目的（不想接收对方的数据包，只要对方可以介绍本地发送的数据包即可），则正好满足条件，对于 IP 源地址伪装以及源 MAC 地址伪装，内核实现上没有采取任何动作，所以 PACKET 类型套接字原则上提供了一种较佳的攻击手段。

3〉 PACKET 类型套接字如同 UDP 协议只提供面向报文的，不可靠数据传输方式。

4〉 PACKET 类型套接字数据读取方式如同 UDP，每次只能读取一个数据包，不可跨越多个数据包进行读取，单次为读完的数据将被丢弃。

5〉 PACKET 类型套接字直接使用链路层模块接口函数进行数据包的发送和接收。对于接收，需要提供 packet_type 结构注册数据包接收回调函数；对于发送，直接调用 dev_queue_xmit 这个链路层接口函数发送数据。

诚如 2〉中所述，PACKET 类型套接字可以进行源端地址（IP 地址，MAC 地址）伪装，不失为一种有效的攻击方式。但是由于一般路由器都配置为只转发源端为本地网络地址的数据包，所以如果对广域网中某个主机进行攻击，攻击数据包在中间某个路由器上就可能被丢弃，加上防火墙规则过滤，攻击目的很难达到。不过目的主机进行处理的本身（如进行防火墙规则检查）就是一种 DoS 攻击方式。

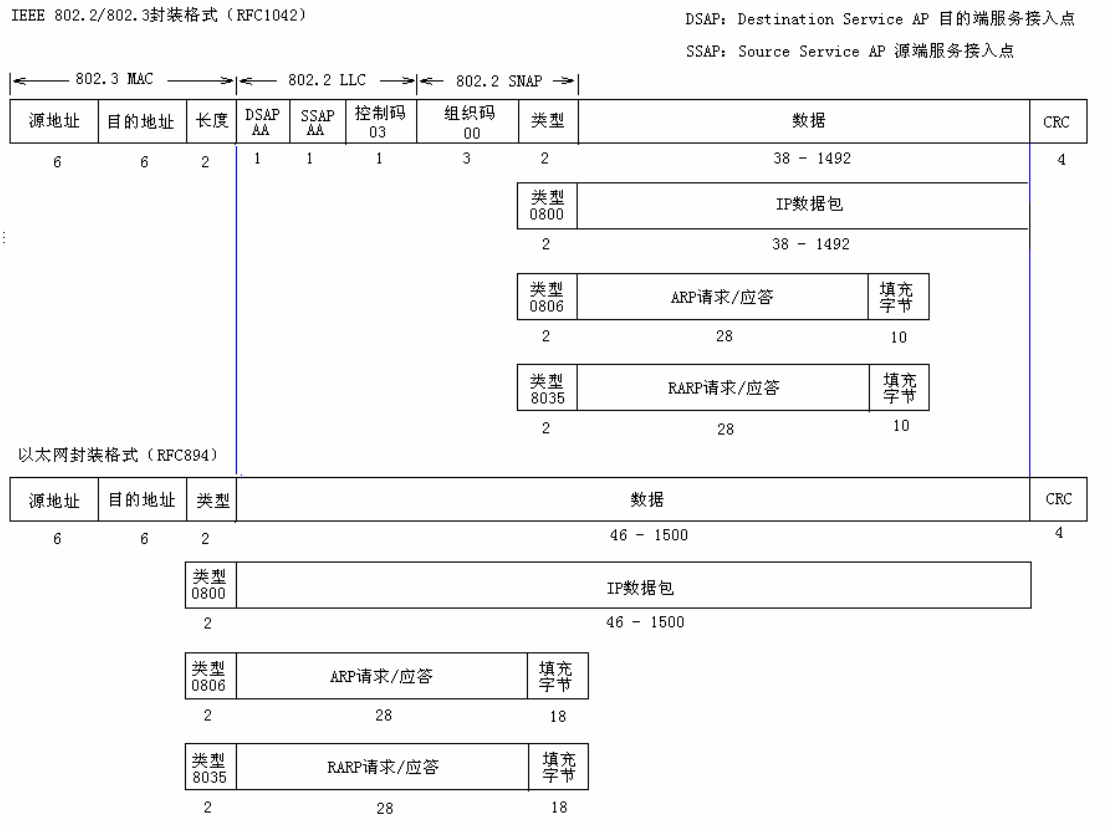
通常我们谈到 Ethernet（以太网）时，指的是 DEC，Intel，Xerox Corp 三家公司于 1982 年发布的以太网标准。这个以太网标准被广泛使用在当前的 TCP/IP 网络中，其使用 CSMA/CD 机制处理以太网上多个主机同时发送数据包时的冲突问题，使用 48 比特硬件地址。此后 IEEE802 委员会发布了一组不同的标准，其中 802.3 标准针对使用 CSMA/CD 机制的网络，802.4 标准针对令牌总线网络，802.5 标准针对令牌环网络。而 802.2 标准则对以上三个标准定义了一个通用的链路控制层（LLC）标准。可惜的是 802 系列标准与原有的以太网标准不兼容，他们采用不同的封装格式，RFC894 描述了以太网封装格式，RFC1042 描述了 802 系列标准封装格式。由于两种标准使用封装格式不同，为了在使用两种不同格式的主机之间进行数据包传送，相关 RFC 定义了网络栈实现的一套规则：

1〉 必须能够接收和发送 RFC894 封装格式（即以太网封装格式）。

2〉 能够同时接收 RFC1042 封装格式（即 802 封装格式）。

3〉 可以发送 RFC1042 封装格式数据包，但是对此不要求一定实现。

RFC894 即以太网封装格式是最常用的封装格式，下图显示了两种不同的封装格式。图中数字为对应字段所占据的字节数。在下文中对 802.2 以及 SNAP 协议实现文件的分析中，请读者记住如下这幅图，这将大大有助于下文中的理解。



链路层（或者说设备层）模块实现文件为 dev.c 文件，该文件实现的模块作为驱动程序和网络层（以及 LLC 控制层）之间的接口模块而存在。以太网首部创建模块实现文件为 eth.c；802 格式首部创建模块由 p8022.c, p8022.h, p8022call.h, p8023.c 四个文件实现；在 LLC 控制层之后添加的 SNAP 子网接入层模块实现由 psnap.c, psnap.h, psnapcall.h 三个文件共同完成。这些首部格式创建模块都被链路层接口模块（dev.c）调用。所以在介绍 dev.c 之前，我们首

先对这些不同首部格式创建模块以及子层（LLC，SNAP）实现模块进行介绍。从上图中，我们可以看出以太网封装格式在链路层首部之后紧随着 IP 数据包，从本书前文介绍中，相关接收函数（net_bh）也是经过链路层一些处理后，直接将数据包传递给上层函数（如 ip_rcv）进行处理，但对于 802 封装格式，其在 IP 数据包（或者 ARP，RARP 数据包）之前又划分了两个子层：LLC 链路控制子层，SNAP 子网接入点子层。从数据包传递层次而言，相比以太网封装格式，802 标准实现模块在将数据包传递给网络层函数之前（如 ip_rcv），还需要首先经过 LLC，SNAP 子层的处理，即 net_bh 接收函数向上层传递数据包时，对应于 802 标准，将首先传递给 LLC 子层接收函数，其后再传递给 SNAP 子层接收函数，最后才传递给网络层接收函数（如 ip_rcv）。

下面我们将从具体代码实现上，查看这种数据包传递关系。在前文中对 ip_rcv 函数的介绍中，我们说到网络层通过 packet_type 结构将接收函数注册给链路层模块调用，此处同理 802.2LLC 链路控制模块也是采用这种方式，下面即分析 802.2 LLC 控制模块的实现文件 p8022.c p8022.h, p8022call.h。

2.26 net/inet/p8022.h 头文件

```
1    struct datalink_proto *register_8022_client(unsigned char type,
         int (*rcvfunc)(struct sk_buff *, struct device *, struct packet_type *));
```

该文件实现只有一行，对 register_8022_client 函数进行声明，该函数将被处于 802.2 LLC 控制模块之上的模块（SNAP 模块）进行函数注册之用，即 LLC 上层模块向 LLC 注册数据包接收函数。这个函数的使用在介绍 SNAP 模块实现文件时即可看到，此处我们先且跳过。

2.27 net/inet/p8022call.h 头文件

```
1    /* Separate to keep compilation of Space.c simpler */
2    extern void p8022_proto_init(struct net_proto *);
```

该文件对 p8022_proto_init 函数进行声明，p8022_proto_init 函数在网络栈初始化过程中被调用用于初始化 802.2 协议实现模块。此处我们将花费笔墨介绍简单介绍一些网络栈各模块的初始化过程(对此在第四章还将有详细介绍)，我们从 LINUX 内核初始化入口函数 start_kernel 开始，如下代码片段：

```
/*init/main.c*/
310  asmlinkage void start_kernel(void)
311  {

     ……
351          sock_init();

     ……
369          for(;;)
370               idle();
371  }
```

网络栈初始化总入口函数为 sock_init，该函数定义在 net/socket.c 文件中，实现如下：

```
/*net/socket.c*/
1148void sock_init(void)
1149{
```

```
1150    int i;

1151    printk("Swansea University Computer Society NET3.019\n");

1152    /*
1153     *    Initialize all address (protocol) families.
1154     */

1155    for (i = 0; i < NPROTO; ++i) pops[i] = NULL;

1156    /*
1157     *    Initialize the protocols module.
1158     */

1159    proto_init();

1160#ifdef CONFIG_NET
1161    /*
1162     *    Initialize the DEV module.
1163     */

1164    dev_init();

1165    /*
1166     *    And the bottom half handler
1167     */

1168    bh_base[NET_BH].routine= net_bh;
1169    enable_bh(NET_BH);
1170#endif
1171}
```

1159 行函数调用 proto_init 函数对协议模块进行初始化，proto_init 也实现在 socket.c 中，如下所示：

```
/*net/socket.c*/
1135void proto_init(void)
1136{
1137    extern struct net_proto protocols[];/* Network protocols */
1138    struct net_proto *pro;

1139    /* Kick all configured protocols. */
1140    pro = protocols;
1141    while (pro->name != NULL)
1142    {
```

```
1143            (*pro->init_func)(pro);
1144            pro++;
1145        }
1146        /* We're all done... */
1147}
```

该函数遍历 protocols 变量指向的 net_proto 结构队列，调用队列中每个元素中注册的模块初始化函数进行相应协议模块的初始化工作。Protocols 变量定义在 net/protocols.c 中，如下代码片段所示：

```
/*net/protocols.c 代码片段*/
16  #ifdef CONFIG_IPX
17  #include "inet/ipxcall.h"
18  #include "inet/p8022call.h"
19  #endif
    ……
33  struct net_proto protocols[] = {
34  #ifdef      CONFIG_UNIX
35      { "UNIX",   unix_proto_init      },
36  #endif
```

UNIX 域实现模块初始化函数注册为 unix_proto_init.

```
37  #if defined(CONFIG_IPX)||defined(CONFIG_ATALK)
38      { "802.2",    p8022_proto_init },
39      { "SNAP",   snap_proto_init },
40  #endif
```

37-40 行代码即对 802.2，SNAP 协议实现模块进行初始化函数注册，注意 802.2 协议初始化函数为 p8022_proto_init，18 行代码中对 p8022call.h 头文件的包含就是为了此处的对 p8022_proto_init 函数进行引用。

```
41  #ifdef CONFIG_AX25
42      { "AX.25",  ax25_proto_init },
43  #endif
44  #ifdef      CONFIG_INET
45      { "INET",    inet_proto_init},
46  #endif
47  #ifdef    CONFIG_IPX
48      { "IPX",      ipx_proto_init },
49  #endif
50  #ifdef CONFIG_ATALK
51      { "DDP",     atalk_proto_init },
52  #endif
53      { NULL,     NULL            }
```

```
54  };
```

有关网络栈的详细初始化过程将在本书的第四章钟进行介绍，此处到此为止。下面我们真正进入 802.2 协议实现文件的分析。不过首先我们需要对相关结构进行说明，这主要涉及 datalink.h 头文件。

2.28 net/inet/datalink.h 头文件

```
1   #ifndef _NET_INET_DATALINK_H_
2   #define _NET_INET_DATALINK_H_

3   struct datalink_proto {
4       unsigned short    type_len;
5       unsigned char  type[8];
6       char        *string_name;
7       unsigned short header_length;
8       int    (*rcvfunc)(struct sk_buff *, struct device *,
9                          struct packet_type *);
10      void (*datalink_header)(struct datalink_proto *, struct sk_buff *,
11                              unsigned char *);
12      struct datalink_proto      *next;
13  };

14  #endif
```

该函数定义了 datalink_proto 结构，这个结构将被 802.2 协议实现模块使用用于保存上层协议（SNAP）相关函数句柄，具体作用在如下相关函数讨论中将会有所说明。

2.29 net/inet/p8022.c 文件

该文件是 802.2 协议实现文件，802.2 协议又称为链路控制协议（LLC），处于 802.3 协议和网络层协议之间，在数据帧中占用 3 个字节，具体格式如前文所示。由于 802.2 协议用于控制目的，所以链路层在将数据包传递给网络层之前，首先需要将数据包传递给链路控制模块进行处理，一般如果使用了链路控制协议，则同时也会使用 SNAP 协议，SNAP 协议在介绍 psnap.c 文件时再进行分析。链路控制模块既然处于链路层和网络之间，则其必须向下（链路层）注册其数据包接收函数，同时提供注册函数供上层协议（SNAP）调用进行数据包接收回调函数注册。从以下代码实现，我们可以得知，对于向下层（链路层）的注册是通过 packet_type 结构完成的；而 register_8022_client 则是提供给上层的数据包接收注册函数。

```
1   #include <linux/netdevice.h>
2   #include <linux/skbuff.h>
3   #include "datalink.h"
4   #include <linux/mm.h>
5   #include <linux/in.h>

6   static struct datalink_proto *p8022_list = NULL;
```

p8022_list 变量维护上层协议队列，对于使用 LLC 协议模块的每个上层协议（主要是指 SNAP 协议）以一个 datalink_proto 结构表示，该结构中定义有数据包接收函数，协议首部创建函数等等信息，从而完成与 LLC 模块的交互。p8022_list 变量所指向队列中新结构的链入是在 register_8022_client 函数中进行的，这是 LLC 协议模块提供给使用 LLC 的上层协议进行注册的函数，即将表示上层协议的 datalink_proto 结构接入到 p8022_list 变量维护的队列中。每当 LLC 模块接收到链路层传递上来的数据包后，其将遍历 p8022_list 变量指向的队列，对其中每个元素进行检查，寻找合适的上层协议对应的结构，调用其中的函数句柄将数据包传递给上层进行处理。寻找的依据是查看 datalink_proto 结构中的 type 字段是否匹配，当前使用 LLC 协议模块的上层协议只有 SNAP 协议，type 字段被设置为 0xAA。

```
7   static struct datalink_proto *
8   find_8022_client(unsigned char type)
9   {
10      struct datalink_proto   *proto;

11      for (proto = p8022_list;
12          ((proto != NULL) && (*(proto->type) != type));
13          proto = proto->next)
14          ;

15      return proto;
16  }
```

find_8022_client 函数被 p8022_rcv 函数调用从 p8022_list 变量指向的 datalink_proto 结构队列中查找合适的元素，并返回该元素。查找的依据就是进行 type 字段的比较（12 行）。

```
17  int
18  p8022_rcv(struct sk_buff *skb, struct device *dev, struct packet_type *pt)
19  {
20      struct datalink_proto   *proto;

21      proto = find_8022_client(*(skb->h.raw));
22      if (proto != NULL) {
23          skb->h.raw += 3;
24          skb->len -= 3;
25          return proto->rcvfunc(skb, dev, pt);
26      }

27      skb->sk = NULL;
28      kfree_skb(skb, FREE_READ);
29      return 0;
30  }
```

p8022_rcv 函数之于 802.2 协议如同 ip_rcv 函数之于 IP 协议，我们在下面 50 行代码中将看到在创建 802.2 协议对应的 packet_type 结构时，packet_type 结构中接收函数指针即被设置为 p8022_rcv 函数，所以当链路层函数（net_bh）在决定将数据包传递给哪个时，遍历 ptype_base 变量指向的 packet_type 结构队列，根据数据帧链路首部中对应字段查找合适的 packet_type 结构，并调用其中的接收函数将数据包传递给上层进行处理，如果判断出链路首部是 802.3 格式，则这个上层指的就是 802.2 协议模块，即此处定义的 p8022_rcv 函数将被（net_bh 函数）调用。注意在将数据包传递给上层协议之前，连路层模块已经将 skb->h.raw 设置为指向上层协议的首部，所以 21 行代码中 skb->h.raw 即指向 802.2 首部，当前 802.2 首部固定为 3 个字节，且每个字节的数值也固定，分别为 0xAA，0xAA，0x03，所以在调用 find_8022_client 函数时输入的参数值即为 0xAA，而 SNAP 协议在进行 datalink_proto 结构注册时使用的类型值也是 0xAA（snap_proto_init 函数，psnap.c 文件），所以 802.2 模块会进一步将数据包传递给 SNAP 模块中数据包接收函数进行处理。虽然当前使用 802.2 协议的上层协议只有 SNAP 协议，但通过维护 datalink_proto 结构队列方式为以后的可能扩展提供了极大的方便性。注意在进行 SNAP 接收函数调用之前，802.2 模块在 23 行将 skb->h.raw 字段设置为指向 SNAP 首部（802.2 协议首部固定为 3 个字节），24 行相应的更新长度字段。

```
31  static void
32  p8022_datalink_header(struct datalink_proto *dl,
33                  struct sk_buff *skb, unsigned char *dest_node)
34  {
35          struct device    *dev = skb->dev;
36          unsigned long len = skb->len;
37          unsigned long hard_len = dev->hard_header_len;
38          unsigned char  *rawp;

39          dev->hard_header(skb->data, dev, len - hard_len,
40                          dest_node, NULL, len - hard_len, skb);
41          rawp = skb->data + hard_len;
42          *rawp = dl->type[0];
43          rawp++;
44          *rawp = dl->type[0];
45          rawp++;
46          *rawp = 0x03; /* UI */
47          rawp++;
48          skb->h.raw = rawp;
49  }
```

p8022_datalink_header 用于创建 802.3 首部以及 802.2 协议首部。对于 802.3 协议首部将通过调用 device 结构中 hard_header 指针指向的函数（p8023_datalink_header）完成。 42-46 完成 802.2 首部创建，由于 802.2 首部长度，格式，数值都固定，所以代码比较简单，注意 p8022_datalink_header 将被 SNAP 协议调用，用于创建 802.3，802.2 首部，所以 p8022_datalink_header 函数第一个参数将是表示 SNAP 协议的 datalink_proto 结构，该结构中 type 字段被设置为 0xAA，用于在 find_p8022_client 函数进行 datalink_proto 结构的匹配。所以 42，44 行代码正好将 802.2 首部前两个字节设置为 0xAA，第三个字节设置为 0x03。

```
50  static struct packet_type p8022_packet_type =
51  {
52          0,      /* MUTTER ntohs(ETH_P_IPX),*/
53          NULL,           /* All devices */
54          p8022_rcv,
55          NULL,
56          NULL,
57  };
```

50-57 行完成对应 LLC 协议 packet_type 结构的创建，该结构将在如下的 p8022_proto_init 函数被插入到 ptype_base 指向的队列中，完成 802.2 协议模块向链路层模块的注册，从而为接收使用 802.2 协议的数据包建立起通道。

```
58  void p8022_proto_init(struct net_proto *pro)
59  {
60          p8022_packet_type.type=htons(ETH_P_802_2);
61          dev_add_pack(&p8022_packet_type);
62  }
```

p8022_proto_init 函数即完成 802.2 协议对应 packet_type 结构的注册，这如同 IP 协议对应 packet_type 结构的注册，注册的目的和意义都相同。注意 60 行中 packet_type 结构中 type 字段被赋值为 ETH_P_802_2，IP 协议中该字段则为 ETH_P_IP。结合前文中 802 系列协议和以太网协议数据帧的封装格式，此处注册的意义一目了然。

```
63  struct datalink_proto *
64  register_8022_client(unsigned char type,
                         int (*rcvfunc)(struct sk_buff *, struct device *, struct packet_type *))
65  {
66          struct datalink_proto       *proto;

67          if (find_8022_client(type) != NULL)
68                  return NULL;

69          proto = (struct datalink_proto *) kmalloc(sizeof(*proto), GFP_ATOMIC);
70          if (proto != NULL) {
71                  proto->type[0] = type;
72                  proto->type_len = 1;
73                  proto->rcvfunc = rcvfunc;
74                  proto->header_length = 3;
75                  proto->datalink_header = p8022_datalink_header;
76                  proto->string_name = "802.2";
77                  proto->next = p8022_list;
78                  p8022_list = proto;
```

```
79          }

80          return proto;
81  }
```

register_8022_client 是 802.2 协议模块提供给 SNAP 协议进行接收函数注册的函数。SNAP 协议调用该函数将其接收函数注册给 802.2 协议模块，从而在 802.2 协议模块和 SNAP 协议模块之间建立数据包传递通道。我们在说到 IP 协议和 TCP，UDP 等协议建立数据包传递通道时使用 inet_protocol 结构，TCP，UDP 以及 ICMP 每个协议都初始化一个 inet_protocol 结构并注册到 IP 协议模块维护的队列中，在 IP 协议模块接收到一个数据包，需要向上传递时，其遍历维护的 inet_protocol 结构队列，查找满足相关匹配条件的（协议号匹配）inet_protocol 结构，从而调用其中的接收函数句柄，完成数据包的传递。此处 802.2 协议与其上层协议（SNAP）类型，不同的是使用了不同的注册结构：datalink_proto 结构。SNAP 协议初始化一个 datalink_proto 结构，并调用 register_8022_client 函数将该结构注册到 802.2 协议实现模块维护的队列中，从而当 802.2 协议接收到一个数据包，需要将数据包传递给上层协议时，在其维护的 datalink_proto 结构队列中根据相关字段值进行匹配 datalink_proto 结构的查找（由于当前该队列中只维护有 SNAP 协议对应的 datalink_proto 结构，所以比较简单），查找到这样一个匹配项后，调用其中的接收函数句柄，从而将数据包传递给上层协议（SNAP 协议）进行处理。
67 行代码检查重复注册的情况。69-79 即完成一个新的 datalink_proto 结构的创建，并根据输入参数值进行初始化，最后插入到由 802.2 协议模块维护的 datalink_proto 结构队列中（由 p8022_list 变量指向）。

至此我们完成对 802.2 协议实现文件的分析，802.2 协议实现模块所处的位置和作用类似于 IP 协议实现模块，其直接位于链路层模块之上，之所以出现这两种不同情况，主要是因为两种不同的数据帧封装格式，其一为以太网封装格式，链路层有以太网风转格式，之上就是 IP 协议模块；其二为 802 系列封装格式，链路层有 802.3 封装格式，其上为 802.2 封装格式，其上为 SNAP 封装，再其上方为 IP 协议模块。主机网络栈实现规范上明确要求必须同时可以对这两种封装格式的数据包进行接收和处理，所以如果接收到 802 系列封装格式的数据包，在链路层模块之后，必须经过 802.2 协议模块和 SNAP 协议模块方可到达 IP 协议模块。由于链路层提供了注册接口，所以 802.2 协议模块在进行其接收函数注册时，使用了以太网下 IP 协议同样的注册方式以及相同的注册结构，从而迎合链路层模块完成注册过程。这一点在 p8022.c 文件中 p8022_packet_type 变量的创建中即可看出。由于在到达 IP 协议模块之前，还必须经过 SNAP 模块，所以如何建立 802.2 模块和 SNAP 模块之间的数据通道就是自己进行定制了，从上文实现来看，还是由较下的一层协议（802.2 协议）实现模块提供注册函数（register_8022_client）和注册所用结构（datalink_proto），较上的一层协议（SNAP 协议）进行对应的注册结构初始化并调用对应注册函数进行注册即可，下层协议在接收到数据包后，自然调用之前较上层协议注册的接收函数将数据包传递给较上层协议实现模块进行处理。
经过以上的分析讨论，我们下面自然而然将对 SNAP 协议实现模块进行分析，之后我们分析出 SNAP 协议模块如何与其上层协议 IP 协议建立数据通道。涉及到 SNAP 协议的文件如下：psnap.h, psnap.c psnapcall.h。

2.30 net/inet/psnap.h 头文件

```
1    struct datalink_proto *register_snap_client(unsigned char *desc,
         int (*rcvfunc)(struct sk_buff *, struct device *, struct packet_type *));
```

该文件对 register_snap_client 进行声明，这是由 SNAP 协议模块提供给其上层协议的注册函数，其作用类似于 802.2 协议实现模块中 register_8022_client 函数。从这个函数原型可见，SNAP 协议提供给其上层协议（IP 协议）的注册结构仍然为 datalink_proto，这就与以太网下链路层提供给 IP 协议的注册结构（packet_type 结构）不同，所以为了能够同时进行两种封装格式数据包的接收，IP 协议实现模块必须使用不同的数据结构分别向 802 封装格式下 SNAP 协议和以太网封装格式下链路层协议进行注册。这一注册过程在 ip_init(ip.c)函数中完成。不过本版本中虽然实现有 802.2 协议，SNAP 协议的文件，但 IP 协议并未向 SNAP 协议进行注册，换句话说，本版本将不支持对 802 封装格式数据包的接收，只能对以太网封装格式的数据包进行接收。

2.31 net/inet/psnapcall.h 头文件

```
1    /* Separate to keep compilation of Space.c simpler */
2    extern void snap_proto_init(struct net_proto *);
```

该文件对 SNAP 协议模块初始化函数进行声明，这个函数的被调用过程与上文中介绍 p8022_proto_init 函数的调用流程相同。

2.32 net/inet/psnap.c 文件

该文件是 SNAP 协议实现文件，使用数据结构以及基本实现代码都类同于 802.2 协议实现，故对相关结构和代码将不再进行说明。

```
1    /*
2     *    SNAP data link layer. Derived from 802.2
3     *
4     *        Alan Cox <Alan.Cox@linux.org>, from the 802.2 layer by Greg Page.
5     *        Merged in additions from Greg Page's psnap.c.
6     *
7     *        This program is free software; you can redistribute it and/or
8     *        modify it under the terms of the GNU General Public License
9     *        as published by the Free Software Foundation; either version
10    *        2 of the License, or (at your option) any later version.
11    */

12   #include <linux/netdevice.h>
13   #include <linux/skbuff.h>
14   #include "datalink.h"
15   #include "p8022.h"
16   #include "psnap.h"
17   #include <linux/mm.h>
18   #include <linux/in.h>

19   static struct datalink_proto *snap_list = NULL;
```

```
20  static struct datalink_proto *snap_dl = NULL;          /* 802.2 DL for SNAP */
```

19 行 snap_list 维护上层协议注册的 datalink_proto 结构队列。20 行 snap_dl 也同样指向一个 datalink_proto 结构队列，这个队列中维护是 SNAP 协议向 802.2 协议注册的所有 datalink_link 结构。由于当前只有 SNAP 协议使用 802.2 协议，所以这个由 snap_dl 指向的队列与 p8022.c 文件中有 p8022_list 变量指向的队列完全相同！在 SNAP 协议实现模块中自己另外维护一个 datalink_proto 结构队列的目的在于对 802.2 协议取相关信息时比较方便，因为 snap_dl 维护的队列中 datalink_proto 结构都是调用 register_8022_client 函数返回的，这个返回的 datalink_proto 结构中所以字段都是由 802.2 协议模块初始化的，其中包含了 802.2 协议首部长度值，802.2 协议首部创建函数句柄等等。这些信息在 SNAP 协议模块相关函数实现中将很有用处！具体可以从下面 SNAP 协议实现中看出。

```
21  /*
22   *  Find a snap client by matching the 5 bytes.
23   */

24  static struct datalink_proto *find_snap_client(unsigned char *desc)
25  {
26      struct datalink_proto   *proto;

27      for (proto = snap_list; proto != NULL && memcmp(proto->type, desc, 5) ;
                proto = proto->next);
28      return proto;
29  }
```

find_snap_client 函数的作用如同 find_8022_client，用于在 snap_list 指向的队列中查找对应的 datalink_proto 结构并返回。注意匹配字段是 datalink_proto 结构的 type 字段。

```
30  /*
31   *  A SNAP packet has arrived
32   */

33  int snap_rcv(struct sk_buff *skb, struct device *dev, struct packet_type *pt)
34  {
35      static struct packet_type psnap_packet_type =
36      {
37          0,
38          NULL,       /* All Devices */
39          snap_rcv,
40          NULL,
41          NULL,
42      };

43      struct datalink_proto   *proto;
```

```
44          proto = find_snap_client(skb->h.raw);
45          if (proto != NULL)
46          {
47                  /*
48                    *    Pass the frame on.
49                    */

50                  skb->h.raw += 5;
51                  skb->len -= 5;
52                  if (psnap_packet_type.type == 0)
53                          psnap_packet_type.type=htons(ETH_P_SNAP);
54                  return proto->rcvfunc(skb, dev, &psnap_packet_type);
55          }
56          skb->sk = NULL;
57          kfree_skb(skb, FREE_READ);
58          return 0;
59  }
```

snap_rcv 函数是 SNAP 数据包接收函数，这个函数被 p8022_rcv 函数调用。函数实现较简单，其查找其维护的上层协议注册的 datalink_proto 结构队列（snap_list 指向），调用对应结构中接收函数将数据包传递给上层协议模块进行处理。50 行加 5 用于去除 SNAP 首部，从而在将数据包传递给上层协议处理时，skb->h.raw 字段指向对应上层协议首部。一般而言，此处所述的上层协议为 IP 协议。注意为了维护对上层接收函数统一的函数接口，在调用接收函数时，其首先创建了一个 SNAP 协议对应的 packet_type 结构，将该结构作为第三个参数传入上层数据包接收函数。这个参数表示上一个具备数据包接收条件的协议对应的 packet_type 结构（具体参见本书下文中 net_bh 函数实现-dev.c），对于 SNAP 协议，其上为 IP 协议，所以 54 行是对 ip_rcv 函数的调用，此时 35-42 行临时创建的 SNAP 协议对应的 packet_type 结构传入。实际上，对于这第三个参数，所有的接收函数中（ip_rcv, p8022_rcv）都没有使用，所以此处主要是为了维护调用接口的一致性，无他目的，当然也可以说是为保持代码的可扩展性。另外此处将 ip_rcv 函数的第三个参数设置为 psnap_packet_type，即等同于将 SNAP 协议和 IP 协议看作是同一层次，这是不对的，如果使用 802 封装格式，则 IP 协议是建立在 SNAP 协议之上的，不可等同对待，所以 54 行调用中对第三个参数的处理有些随意（虽然对结果没有造成任何影响）。

```
60  /*
61    *    Put a SNAP header on a frame and pass to 802.2
62    */

63  static void snap_datalink_header(struct datalink_proto *dl, struct sk_buff *skb,
                                unsigned char *dest_node)
64  {
65          struct device    *dev = skb->dev;
66          unsigned char  *rawp;
```

```
67          rawp = skb->data + snap_dl->header_length+dev->hard_header_len;
68          memcpy(rawp,dl->type,5);
69          skb->h.raw = rawp+5;
70          snap_dl->datalink_header(snap_dl, skb, dest_node);
71  }
```

snap_datalink_header 用于创建 802.3，802.2，以及其自身 SNAP 协议的首部。首部的创建是层层负责的，即一般对应协议模块只负责本协议首部的实际创建，而下层协议首部的创建是通过调用下层提供的接口函数完成的，如此处是调用 snap_dl->datalink_header 指向的函数，诚如上文所述，snap_dl 指向的 datalink_proto 结构是调用下层注册函数（register_8022_client）返回的，该返回的 datalink_proto 结构中 datalink_header 字段被初始化为对应下层协议首部的创建函数（此处对应为 p8022_datalink_header），所以 70 行的调用即调用下层协议的对应函数创建下层协议的首部。67 行将 rawp 变量设置为指向 SNAP 首部起始处，68 行创建 SNAP 首部（5 个字节），此处 dl 表示上层协议注册的 datalink_proto 结构，其 type 字段将被初始化为 SNAP 协议首部相关字段需要的值，如同 SNAP 协议将其对应的 datalink_proto 结构中 type 字段设置为 0xAA 供 802.2 协议使用一样。从 p8022_datalink_header 函数所起的作用，我们可以想见 snap_datalink_header 函数将在初始化上层协议对应 datalink_proto 结构时被赋值为该结构中 datalink_header 字段，从而被上层协议模块调用创建 SNAP 协议及其以下层协议的首部。这一点我们将在下面介绍的 register_snap_client 函数看到。

```
72  /*
73   *  Set up the SNAP layer
74   */

75  void snap_proto_init(struct net_proto *pro)
76  {
77          snap_dl=register_8022_client(0xAA, snap_rcv);
78          if(snap_dl==NULL)
79                  printk("SNAP - unable to register with 802.2\n");
80  }
```

snap_proto_init 函数为 SNAP 协议模块的初始化函数，其调用 register_8022_client 向下层协议模块注册接收函数，register_8022_client 函数由 802.2 协议实现模块提供，其将创建一个新的 datalink_proto 结构，将传入的第一个参数值作为新结构中 type 字段值，第二个参数作为新结构中接收句柄，另外其还将新结构中 datalink_header 字段设置为 802.2 协议首部的创建函数。77 行保存这个新创建 datalink_proto 结构的目的就在于可以对这些由 802.2 协议模块赋值的字段进行利用。如以上相关函数实现中对 802.2 协议首部长度以及创建函数的使用。

```
81  /*
82   *  Register SNAP clients. We don't yet use this for IP or IPX.
83   */

84  struct datalink_proto *register_snap_client(unsigned char *desc,
```

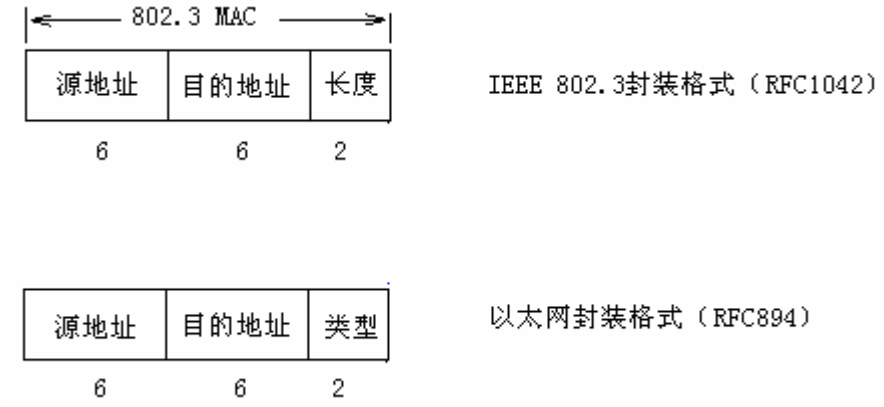
```
                int (*rcvfunc)(struct sk_buff *, struct device *, struct packet_type *))
85  {
86      struct datalink_proto      *proto;

87      if (find_snap_client(desc) != NULL)
88          return NULL;

89      proto = (struct datalink_proto *) kmalloc(sizeof(*proto), GFP_ATOMIC);
90      if (proto != NULL)
91      {
92          memcpy(proto->type, desc,5);
93          proto->type_len = 5;
94          proto->rcvfunc = rcvfunc;
95          proto->header_length = 5+snap_dl->header_length;
96          proto->datalink_header = snap_datalink_header;
97          proto->string_name = "SNAP";
98          proto->next = snap_list;
99          snap_list = proto;
100     }

101     return proto;
102 }
```

register_8022_client 函数是由 802.2 协议提供给 SNAP 协议模块的注册函数，而 register_snap_client 则是 SNAP 协议模块提供给其上协议（如 IP 协议）的注册函数。其实现基本类同于 register_8022_client 函数：创建一个新的 datalink_proto 结构，92，94 行利用传入的参数值，其他字段的赋值则是 SNAP 协议模块本身的信息，这些信息将通过返回给上层模块的 datalink_proto 结构被上层协议模块使用，如 SNAP 协议首部长度及其首部创建函数（snap_datalink_header）。99 行 snap_list 维护对上层协议注册的 datalink_proto 结构队列。注意这种赋值方式表示 snap_list 指向的队列中将只有一个可用元素。同样 802.2 协议实现模块中由 p8022_list 变量指向的队列也只有一个可用元素，因为当前建立在 802.2 协议之上的只有 SNAP 协议；但在 SNAP 协议之上，可以有 IP 协议和 ARP，RARP 协议，所以 99 行的赋值方式可能会出问题，当然本版本实现中并不支持 802 封装格式的数据包接收，所以这些实际上并没有被使用。但我们还是必须了解他们之间的关系。

到此为止，我们完成对 802.2 以及 SNAP 协议实现文件的分析，这两种协议只有在采用 802 封装格式的数据包中方可见到，由于其处在链路层和网络层之间，为了保持对链路层和网络层接口的一致性，所以在实现上对于 802.2 协议与其链路层协议（802.3 协议）之间以及 SNAP 协议与网络层协议（IP 协议）之间都保持了和以太网封装格式下链路层和网络层同样的接口形式。即变更 802.2 协议和 SNAP 协议实现以迎合链路层和网络层协议实现模块，从而在两种不同的封装格式下保持链路层和网络层接口的一致性（虽然对于两种不同的封装格式，链路层使用了不同的封装格式）。

既然讨论到链路层不同的封装格式，此处我们就继续深入下去。对于 802 封装格式，链路层

使用 802.3 协议，而以太网封装格式，即使用通常的以太网协议（其实也不能称为协议，就是一种封装格式，此处是为了和 802.3 协议形成对应）。因为涉及到不同的封装格式，所以就对应有两种不同的首部创建函数，也就对应有不同的实现文件。802.3 首部格式创建由 p8023.c 文件完成，而以太网首部格式创建则由 eth.c 文件完成，下面我们分别对这两个文件进行介绍。链路层上二者封装格式只存在些许差别。鉴于此，我们先行介绍以太网首部封装格式创建文件实现。为便于读者理解，我们重新给出两种不同的链路层封装格式。如下图所示，注意到两种封装格式中只有最后两个字节的含义不同，对于 802.3 封装格式，最后两个字段表示链路层首部之后数据长度，而以太网封装格式中，最后两个字段则表示其后封装数据报文的类型，如 0x0800 表示 IP 报文。



2.33 net/inet/eth.c 文件

该文件负责创建以太网封装格式链路层首部，从其实现来看，其对 802.3 首部格式也进行了检查，因为两种封装格式只有首部中最后一个字段有所差异，而这个字段只需根据传入参数表示的封装格式类型即可判断出如何赋值，所以相关函数中也一并对 802.3 首部格式的创建进行了负责。我们知道链路层首部中承载是本地和下一站主机的硬件地址（6 字节），对于下一站硬件地址，我们一般不知道，但通过已经建立好的路由表，我们知道下一站主机的 IP 地址，所以就需要一种机制完成 IP 地址到硬件地址之间的映射，这就是 ARP 协议完成的功能。对于 ARP 协议实现文件本书下文中将有介绍。为了避免每次进行数据帧发送时使用 ARP 进行地址解析所造成的延迟，本地将维护一个 ARP 缓存，其中包含了最近使用过的 IP 地址到其硬件地址之间的映射关系。所以如果一个下一站主机存在有对应的 ARP 缓存项，则无需使用 ARP 请求，应答机制，直接从缓存中取对应 IP 地址的硬件地址即可。但是由于网络上拓扑结构的变化性，以及可能主机硬件地址变更，所以 ARP 缓存中每个映射关系都有一个生存期，过了这个生存期的项目将变为无效，此时就需要重新使用 ARP 机制进行地址映射。至于 ARP 缓存的实现，类似于路由表和防火墙的实现。这一点在本书下文中介绍 ARP 协议实现文件时将着重介绍。

```
1   /*
2    * INET        An implementation of the TCP/IP protocol suite for the LINUX
3    *             operating system.  INET is implemented using the  BSD Socket
4    *             interface as the means of communication with the user level.
5    *
```

```
6     *        Ethernet-type device handling.
7     *
8     * Version:    @(#)eth.c       1.0.7       05/25/93
9     *
10    * Authors:    Ross Biro, <bir7@leland.Stanford.Edu>
11    *        Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
12    *        Mark Evans, <evansmp@uhura.aston.ac.uk>
13    *        Florian    La Roche, <rzsfl@rz.uni-sb.de>
14    *        Alan Cox, <gw4pts@gw4pts.ampr.org>
15    *
16    * Fixes:
17    *        Mr Linux : Arp problems
18    *        Alan Cox : Generic queue tidyup (very tiny here)
19    *        Alan Cox : eth_header ntohs should be htons
20    *        Alan Cox : eth_rebuild_header missing an htons and
21    *                   minor other things.
22    *        Tegge          : Arp bug fixes.
23    *        Florian        : Removed many unnecessary functions, code cleanup
24    *                   and changes for new arp and skbuff.
25    *        Alan Cox : Redid header building to reflect new format.
26    *        Alan Cox : ARP only when compiled with CONFIG_INET
27    *        Greg Page      : 802.2 and SNAP stuff
28    *
29    *        This program is free software; you can redistribute it and/or
30    *        modify it under the terms of the GNU General Public License
31    *        as published by the Free Software Foundation; either version
32    *        2 of the License, or (at your option) any later version.
33    */
34   #include <asm/segment.h>
35   #include <asm/system.h>
36   #include <linux/types.h>
37   #include <linux/kernel.h>
38   #include <linux/sched.h>
39   #include <linux/string.h>
40   #include <linux/mm.h>
41   #include <linux/socket.h>
42   #include <linux/in.h>
43   #include <linux/inet.h>
44   #include <linux/netdevice.h>
45   #include <linux/etherdevice.h>
46   #include <linux/skbuff.h>
47   #include <linux/errno.h>
48   #include <linux/config.h>
```

```
49  #include "arp.h"
```

因为在创建链路层首部时要使用 ARP 机制，所以我们包含 arp.h 头文件对其中一些函数进行引用。

```
50  void eth_setup(char *str, int *ints)
51  {
52      struct device *d = dev_base;

53      if (!str || !*str)
54          return;
55      while (d)
56      {
57          if (!strcmp(str,d->name))
58          {
59              if (ints[0] > 0)
60                  d->irq=ints[1];
61              if (ints[0] > 1)
62                  d->base_addr=ints[2];
63              if (ints[0] > 2)
64                  d->mem_start=ints[3];
65              if (ints[0] > 3)
66                  d->mem_end=ints[4];
67              break;
68          }
69          d=d->next;
70      }
71  }
```

eth_setup 函数用于对网络设备相关参数进行设置，此处涉及到的参数有中断号，硬件控制和配置寄存器组基地址，硬件缓冲区起始和结束地址。参数 str 用于查找对应的设备，所以的网络设备都被缓存在 dev_base 指向的 device 结构队列中。

```
72  /*
73   *    Create the Ethernet MAC header for an arbitrary protocol layer
74   *
75   *  saddr=NULL  means use device source address
76   *  daddr=NULL  means leave destination address (eg unresolved arp)
77   */

78  int eth_header(unsigned char *buff, struct device *dev, unsigned short type,
79          void *daddr, void *saddr, unsigned len,
80          struct sk_buff *skb)
81  {
```

```
82      struct ethhdr *eth = (struct ethhdr *)buff;

83      /*
84       *  Set the protocol type. For a packet of type ETH_P_802_3 we put the length
85       *  in here instead. It is up to the 802.2 layer to carry protocol information.
86       */

87      if(type!=ETH_P_802_3)
88          eth->h_proto = htons(type);
89      else
90          eth->h_proto = htons(len);

91      /*
92       *  Set the source hardware address.
93       */

94      if(saddr)
95          memcpy(eth->h_source,saddr,dev->addr_len);
96      else
97          memcpy(eth->h_source,dev->dev_addr,dev->addr_len);

98      /*
99       *  Anyway, the loopback-device should never use this function...
100      */

101     if (dev->flags & IFF_LOOPBACK)
102     {
103         memset(eth->h_dest, 0, dev->addr_len);
104         return(dev->hard_header_len);
105     }

106     if(daddr)
107     {
108         memcpy(eth->h_dest,daddr,dev->addr_len);
109         return dev->hard_header_len;
110     }

111     return -dev->hard_header_len;
112 }
```

eth_header 函数即用于进行链路层首部的创建。参数 type 表示首部格式的类型，这一参数在 87-90 行被用于对首部中最后两个字节的含义进行解释。换句话说，该函数同时负责了 802.3 以及以太网格式首部的创建工作。参数 saddr 表示本地硬件地址，如果为 NULL，则使用 dev 参数对应设备的硬件地址；参数 daddr 表示下一站主机硬件地址，如果为 NULL，表示下一

站主机硬件地址未知，此时就需要使用 ARP 机制进行地址解析。所以链路层首部的创建成功与否主要在于下一站主机硬件地址是否存在于本地 ARP 缓存中，如果不存在，则 eth_header 函数返回一个负值，表示首部创建未成功，下一站主机硬件地址未知，此时调用 eth_header 函数的其他函数将数据包对应 sk_buff 结构中 arp 字段设置为 0，表示链路层首部中缺下一站主机硬件地址。在 dev_queue_xmit 函数中，发送数据包之前，其将检查数据包对应 sk_buff 结构中 arp 字段，如果该字段为 0，则表示需要重新构建链路层首部，此时调用 eth_rebuild_header 函数再次进行数据包的链路层首部的构建工作，不过 eth_rebuild_header 与 eth_header 函数实现有所不同。这一点在下文中介绍 eth_rebuild_header 函数时将给出说明。

87-90 判断首部格式的类型，如果是 802.3 格式，则首部中最后一个字段表示的是数据包的长度，否则就表示上层协议的类型。94-97 设置首部中本地硬件地址，如果参数中没有指定硬件地址，则使用发送设备的硬件地址。101-105 对发送设备是回环设备的情况进行检查，此种情况直接将下一站主机硬件地址设置为 0，因为经过回环设备发送的数据包根本就不会离开本机，所以这个字段设置为任何值都无关紧要。最后 106-110 行代码对下一站主机硬件地址字段进行初始化，如果参数 daddr 没有指定硬件地址，则直接返回负值，表示链路层创建失败，否则就使用 daddr 参数值直接初始化链路层首部中下一站主机硬件地址。注意这个 daddr 表示的地址值并不一定就是下一站主机的硬件地址，链路层首部创建是否成功是由数据帧对应的 sk_buff 结构中 arp 字段值决定的，如果 arp 字段为 1，则表示数据帧链路层首部完全创建成功，否则就表示没有创建成功，下一站主机硬件地址需要解析。所以即便 eth_header 函数返回非负值，daddr 参数不为 NULL，也不表示链路层首部创建完毕。因为 eth_header 函数的调用者除了可以将 daddr 参数设置为 NULL 外（如 ip_send, ip.c），还可以将其设置为 IP 地址（注意本版本实现中只有 ip_send 函数中直接调用 eth_header 函数，其他函数如 ip_build_header 都是通过调用 ip_send 函数进行链路层首部创建，而 ip_send 函数调用 eth_header 时传入的 daddr 参数始终为 NULL）。

从 eth_header 函数实现来看，该函数完全依赖于输入参数进行链路层首部的创建工作，如果参数中没有包含足够的信息（如下一站主机硬件地址值），则简单返回一个负值表示链路层首部创建失败，并无进行任何其他“补救措施”，这一点如下面介绍的 eth_rebuild_header 函数是不同的。

```
113  /*
114   *    Rebuild the Ethernet MAC header. This is called after an ARP
115   *    (or in future other address resolution) has completed on this
116   *    sk_buff. We now let ARP fill in the other fields.
117   */

118  int eth_rebuild_header(void *buff, struct device *dev, unsigned long dst,
119                         struct sk_buff *skb)
120  {
121          struct ethhdr *eth = (struct ethhdr *)buff;

122          /*
123           *    Only ARP/IP is currently supported
124           */
```

```
125         if(eth->h_proto != htons(ETH_P_IP))
126         {
127                 printk("eth_rebuild_header: Don't know how to resolve type %d addresses?\n",(int)eth->h_proto);
128                 memcpy(eth->h_source, dev->dev_addr, dev->addr_len);
129                 return 0;
130         }

131         /*
132           *    Try and get ARP to resolve the header.
133           */
134 #ifdef CONFIG_INET
135         return arp_find(eth->h_dest, dst, dev, dev->pa_addr, skb)? 1 : 0;
136 #else
137         return 0;
138 #endif
139 }
```

我们可以给出 eth_rebuild_header 函数被调用的环境，如下代码片段摘自 dev_queue_xmit 数据包发送函数（dev.c）：

```
/*net/inet/dev.c—dev_queue_xmit 函数代码片段*/
309         if (!skb->arp && dev->rebuild_header(skb->data, dev, skb->raddr, skb)) {
310                 return;
311         }
```

在实际发送数据帧之前，dev_queue_xmit 函数检查数据帧对应 sk_buff 封装结构中 arp 字段值，如果 arp 字段为 0，则表示链路层首部尚未完成创建，此时调用发送设备 device 结构中 rebuild_header 指向的函数，对于以太网，即为 eth_rebuild_header 函数。我们从前文对 eth_header 函数的分析中得知，eth_header 函数完全依赖输入参数进行链路层首部的创建，而在发送一个数据帧到一个主机时，很有可能这个主机的硬件地址未知，所以通常情况下，eth_header 函数只是创建了一个链路层首部的框架，并为真正完成创建工作，并且鉴于本版本中对 eth_header 函数的调用方式（ip_send 调用该函数时，下一站主机硬件地址参数始终为 NULL），所以诚如上述，eth_header 函数只是完成链路层首部框架的创建，对于最为关键的下一站主机硬件地址字段没有进行正确的赋值。该字段的真正赋值是由 eth_rebuild_header 函数完成的，该函数检查 ARP 缓存，查找是否有现成的映射关系，如果没有，则将数据包缓存到 ARP 协议数据包暂存队列中，并进行 ARP 地址解析过程，返回 1，表示数据包的发送将由 ARP 协议模块负责，此时 dev_queue_xmit 函数中 310 行直接返回。

下面我们对 eth_rebuild_header 函数中代码进行说明。125-130 行代码检查数据包是否使用的是 IP 协议，因为 ARP 协议只配合以太网网络拓扑结构工作，对于点对点网络，无须这种机制。注意此处检查是否使用 IP 协议（ETH_P_IP），只是进行区分，因为其他无须进行 ARP 请求机制的网络结构也使用 IP 协议（如点对点网络），并非是 ARP 协议与 IP 协议进行绑定。因为只要调用 eth_rebuild_header 函数，就表示之前链路层首部创建失败，所以 135 行调用 arp_find 函数首先检查当前 ARP 缓存中是否存在对应下一站主机地址的映射关系，注意传

入 eth_rebuild_header 函数的第三个参数 dst 表示下一站主机的 IP 地址。此处作为 arp_find 函数的第二个参数传入，作为寻找 ARP 缓存中合适项的依据。如果 arp_find 函数在 ARP 缓存中成功查找到下一站主机硬件地址，则将该数据帧对应 sk_buff 结构中 arp 字段设置为 1，同时返回 0 表示链路层首部创建成功，这种 dev_queue_xmit 函数将跳过 310 行代码，直接进行其下代码的执行；如果 arp_find 函数没有在 ARP 缓存中寻找到对应的下一站主机的硬件地址，则将数据包插入到 ARP 协议模块维护的数据包暂存队列中，并在 ARP 缓存中创建一个新表项，启动 ARP 请求应答机制，进行硬件地址请求，返回 1。在 ARP 缓存中创建一个新表项的目的在于方便以后到相同下一站主机数据包的发送工作。返回 1 表示这个数据包的发送工作将由 ARP 协议模块负责完成，dev_queue_xmit 函数在 310 行可以直接进行返回了。一旦 ARP 协议模块中数据包接收函数接收到对应硬件地址的应答报文，其完成先前被缓存数据包链路层首部的创建工作，重新调用 dev_queue_xmit 函数将数据包发送出去，此时 309 行 skb->arp 字段由于被设置为 1，则直接跳过对 eth_rebuild_header 函数的调用进行 dev_queue_xmit 函数中以下数据包发送代码的执行。有关 arp_find 函数实现以及相关 ARP 缓存以及数据包暂存队列以及 ARP 请求，应答报文等等将在分析 ARP 协议实现文件 arp.c 时进行详细说明，此处不表。

```
140 /*
141  *  Determine the packet's protocol ID. The rule here is that we
142  *  assume 802.3 if the type field is short enough to be a length.
143  *  This is normal practice and works for any 'now in use' protocol.
144  */

145 unsigned short eth_type_trans(struct sk_buff *skb, struct device *dev)
146 {
147     struct ethhdr *eth = (struct ethhdr *) skb->data;
148     unsigned char *rawp;

149     if(*eth->h_dest&1)
150     {
151         if(memcmp(eth->h_dest,dev->broadcast, ETH_ALEN)==0)
152             skb->pkt_type=PACKET_BROADCAST;
153         else
154             skb->pkt_type=PACKET_MULTICAST;
155     }

156     if(dev->flags&IFF_PROMISC)
157     {
158         if(memcmp(eth->h_dest,dev->dev_addr, ETH_ALEN))
159             skb->pkt_type=PACKET_OTHERHOST;
160     }

161     if (ntohs(eth->h_proto) >= 1536)
162         return eth->h_proto;
```

```
163         rawp = (unsigned char *)(eth + 1);

164         if (*(unsigned short *)rawp == 0xFFFF)
165                 return htons(ETH_P_802_3);

166         return htons(ETH_P_802_2);
167 }
```

eth_type_trans 函数用于接收的数据包封装格式检查。149 行对多播和组播数据包的情况进行检查，注意数据帧中使用的网络字节序，所以 149 行检查的是目的硬件地址最高字节的最低位，该位表示目的硬件地址的类型，如果硬件地址最高字节最低位设置为 1，则表示此硬件地址是一个多播地址（注意广播地址属于多播地址），此时将进一步判断是否为广播地址，广播地址为全 1，158 行中 dev->dev_addr 字段在初始化一个 device 结构时被设置为全 1，所以 158 行是对广播地址的情况进行检查，被接收数据包封装结构 sk_buff 中 packet_type 字段表示被数据包的类型，此处类型指的是数据包之于本机的关系，152，154，159 中使用的三个常量都是一种关系的表示。802.3 链路层首部封装格式中，最后两个字节表示数据帧的长度（除去数据帧最后 4 字节的 CRC 字段），这个长度值不会超过 1536 字节，如果大于 1536，则表示该字段不是表示长度值，那么就是类型值（如 0x0800-IP，0x0806-ARP，0x8035-RARP），此时直接返回这个类型值即可，否则继续寻找类型值（注意 eth_type_trans 函数返回值表示所使用上层协议类型，调用者将据此决定数据包的向上传递通道）。164-166 行代码有些费解，首先 163 行代码跳过链路层首部指向链路层首部之后第一个字节，164 行代码检查链路层首部之后两个字节的数值，与 0xFFFF 进行比较，如果相等，则返回 ETH_P_802_3 常数，否则返回 ETH_P_802_2 常数。对于返回 ETH_P_802_3 常数的情况，此处表达的含义是 ETH_P_802_3 协议首部被封装在以太网首部之后，使用了隧道技术，否则就是通常的 802 封装格式，在 802.3 链路层首部之后，跟随的是 802.2 协议首部，所以返回 ETH_P_802_2 常数。这将使得 p8022_rcv 函数在链路层模块处理函数后被调用，而非以太网封装格式下的 ip_rcv 函数。

eth.c 文件实现小结

该文件主要实现了链路层首部创建函数以及用于判断被接收数据包封装格式的判断函数，链路层首部创建分为两步，其一是 eth_header 函数实现方式，根据调用者提供的参数进行首部创建，并且仅此而已，不做出任何额外的工作；其二是 eth_rebuild_header 函数实现方式，其“挑起”首部创建的重任，查找 ARP 缓存，在查找未果的情况下，使用 ARP 机制进行地址解析过程，想方设法完成链路层首部的创建工作。另外 eth_type_trans 函数用于判定一个被接收数据包的类型（发送本机？多播？广播？）以及数据包所使用的封装类型（以太网封装？802 封装？），这将直接被链路层实现模块用于如何向上传递这个接收的数据包（先通过 802.2 协议模块，还是直接传递给 IP 协议模块）。

2.34 net/inet/eth.h 头文件

该头文件中对 eth.c 中定义的函数进行声明。此处简单列出该头文件代码。

```
1   /*
2    * INET        An implementation of the TCP/IP protocol suite for the LINUX
3    *             operating system.  NET  is implemented using the  BSD Socket
4    *             interface as the means of communication with the user level.
```

```
5       *
6       *       Definitions for the Ethernet handlers.
7       *
8       * Version:      @(#)eth.h       1.0.4       05/13/93
9       *
10      * Authors:      Ross Biro, <bir7@leland.Stanford.Edu>
11      *       Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
12      *
13      *       This program is free software; you can redistribute it and/or
14      *       modify it under the terms of the GNU General Public License
15      *       as published by the Free Software Foundation; either version
16      *       2 of the License, or (at your option) any later version.
17      */
18  #ifndef _ETH_H
19  #define _ETH_H


20  #include <linux/if_ether.h>


21  extern char             *eth_print(unsigned char *ptr);
22  extern void             eth_dump(struct ethhdr *eth);
23  extern int      eth_header(unsigned char *buff, struct device *dev,
24                                  unsigned short type, unsigned long daddr,
25                                  unsigned long saddr, unsigned len);
26  extern int      eth_rebuild_header(void *buff, struct device *dev);
27  extern void             eth_add_arp(unsigned long addr, struct sk_buff *skb,
28                                  struct device *dev);
29  extern unsigned short   eth_type_trans(struct sk_buff *skb, struct device *dev);

30  #endif      /* _ETH_H */
```

在完成对以太网封装格式的实现文件后，下面我们介绍对应的 802.3 链路层首部风转格式实现文件 p8023.c。

2.35 net/inet/p8023.c 文件

```
1   #include <linux/netdevice.h>
2   #include <linux/skbuff.h>
3   #include "datalink.h"
4   #include <linux/mm.h>
5   #include <linux/in.h>

6   static void
```

```
7   p8023_datalink_header(struct datalink_proto *dl,
8               struct sk_buff *skb, unsigned char *dest_node)
9   {
10      struct device   *dev = skb->dev;
11      unsigned long len = skb->len;
12      unsigned long hard_len = dev->hard_header_len;

13      dev->hard_header(skb->data, dev, len - hard_len,
14                  dest_node, NULL, len - hard_len, skb);
15      skb->h.raw = skb->data + hard_len;
16  }
```

p8023_datalink_header 函数用于创建 802.3 链路层首部，在 eth_header 函数中我们注意到该函数同时对使用 802.3 协议的情况进行了检查，并相应的进行了对应的处理，即 eth_header 函数同时处理了两种不同的封装格式首部创建工作，所以此处 p8023_datalink_header 函数简单调用 eth_header 函数进行处理。注意两种不同封装格式只在于首部中最后两个字节含义不同，而这在 eth_header 函数已经得到了处理，如果下一站主机硬件地址创建不成功，则 eth_rebuild_header 函数是两种不同封装格式共同的处理函数。

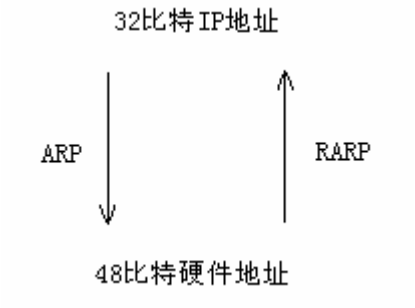
```
17  struct datalink_proto *
18  make_8023_client(void)
19  {
20      struct datalink_proto      *proto;

21      proto = (struct datalink_proto *) kmalloc(sizeof(*proto), GFP_ATOMIC);
22      if (proto != NULL) {
23          proto->type_len = 0;
24          proto->header_length = 0;
25          proto->datalink_header = p8023_datalink_header;
26          proto->string_name = "802.3";
27      }

28      return proto;
29  }
```

make_8023_client 函数的作用类似于 802.2 协议实现模块中 register_8022_client 函数，用于被上层协议调用注册相关函数，从实现来看，其只是将 p8023_datalink_header 函数返回给上层协议使用进行 802.3 协议首部的创建，上层协议并无注册对应的数据包接收函数。首先因为链路层数据包的处理由 dev.c 文件代表的实现模块负责，eth.c, p8023.c 文件都只是对相关首部创建工作进行负责，即主要用于发送数据包时数据帧格式的创建，数据包的发送和接收以及与驱动程序的交互完全是由 dev.c 文件实现负责。驱动程序接收一个数据帧后，对其进行 sk_buff 结构封装，之后交给 dev.c 文件实现的模块，该模块调用 eth_type_trans 函数获取数据帧封装类型，如果是以太网封装，则直接交给网络层协议（如 IP 协议）进行处理（也有可能是 ARP 协议，当接收的数据包是一个 ARP 请求或应答时）；如果是 802 封装，则首

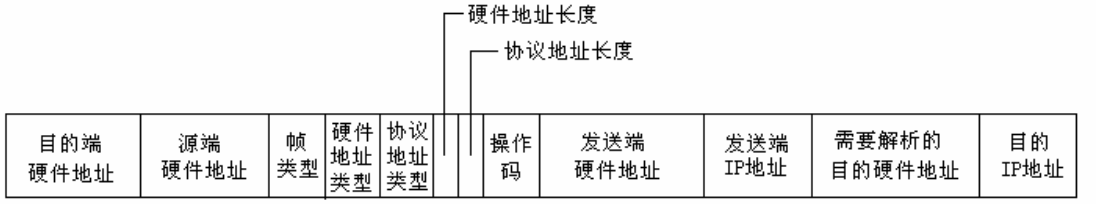
先需要将数据包传递给 LLC（802.2 协议模块）链路控制模块进行处理，该模块又将数据包传递给 SNAP 协议模块，之后才交给网络层协议（IP 协议）进行处理。至于链路层首部采用的是 802.3 封装还是以太网封装，只是在处理链路层首部时对相关字段的含义进行不同的解析，而并非对两种不同的封装对应有两种不同的链路层模块处理函数集合。即链路层模块实现是唯一的（由 dev.c 文件表示），不同的链路层首部格式决定了链路层之上数据包不同的传递通道。而链路层为了实现这一目的，向上提供了统一的接口：上层协议要建立与链路层的传递通道，必须注册一个 packet_type 结构，在其中定义其数据包接收函数。这一点在介绍 IP 协议实现模块以及 802.2 协议实现模块时我们已经看到了对应的相关结构。所以此处 make_8023_client 函数定义的意义不大，在本版本实现中也没有在任何其他地方得到调用，加之 eth_header 函数中对 802.3 首部格式也进行了实现，所以 p8023_datalink_header 函数实现也是不必要的，或者说 p8023.c 文件本身就是不必要的。我们此处给出其代码主要是为了保持本书的完整性，各模块之间，实现文件之间以及函数之间的关系需要读者自行形成自己的思想后进行分析得到。

前文中分析链路层首部创建时，我们说到硬件地址的概念，实际上，更确切的说，应该说成是链路层地址，对于以太网而言，我们通常习惯称之为硬件地址，更不规范的说法就是 MAC 地址。在创建 IP 协议首部时，我们需要 IP 地址，根据数据包的最终目的端 IP 地址，我们查询路由表，寻找下一站主机 IP 地址，根据下一站主机 IP 地址，查找 ARP 缓存或者发起 ARP 请求得到下一站主机硬件地址，从而创建链路层首部，完成数据帧的创建。所以对于一个发送到远端的数据帧而言，其 IP 协议首部中源，目的地址是始终不变的，在数据帧路由（转发）过程中，链路层首部中源，目的硬件地址在不停的变化。因为硬件地址只有在局域网范围中方才有效（或者说硬件地址作用域限于局域网内），而要出局域网发送数据包，则 IP 地址必须具有全局（全球）唯一性。一般而言，对于一个网络接收设备，其出厂时就具有其特定的硬件地址，而且一般不可改变，所以硬件地址对于不同的网络设备也不相同，换句话说，其一般也具有全局唯一性，但这种唯一性并非发送数据包要求的，原则上，我们并不要求并在同一局域网内的两台主机网络设备一定具有不同的硬件地址，而且在不同局域网中的两台主机网络设备具有相同的硬件地址是允许的，这不会对数据包发送和接收造成任何影响，因为硬件地址作用域只限于局域网中。数据包不断转发的过程就是跨越每个局域网的过程，我们在这个过程中，不断地重新创建数据帧中链路层首部，使用不同的源，目的硬件地址。在一次转发过程中，源硬件地址就是发送主机对应网络设备的硬件地址，目的硬件地址就是接收主机对应网络设备的硬件地址，因为在最底层，数据帧首先是由网络设备进行接收，硬件电路会对数据帧中目的硬件地址进行检查，从而决定是否接收该数据帧，一旦发现目的硬件地址并非本设备地址，则不再继续接收后续数据，一般而言只需读取数据帧的前 6 个字节即可，所以网络设备一般都有一个大于 6 字节的 FIFO 缓冲区，从而可以对目的硬件地址进行检查。如果检查出，数据帧中包含的目的硬件地址是本设备配置的地址，则继续对数据帧后续部分进行接收，当完成整个数据帧的接收后，触发中断，由驱动程序中断处理函数负责将硬件缓冲区中数据帧拷贝到内核缓冲区中进行封装交给链路层实现模块进行处理。

下面我们即对 IP 地址和硬件地址之间进行解析的协议：ARP 协议（地址解析协议：Address Resolution Protocol），RARP 协议进行介绍。
对于以太网或者令牌环网络结构而言，硬件地址是一个 48 比特的地址单元。ARP 协议和 RARP 协议完成的工作如下所示：



ARP 协议完成 IP 地址到硬件地址的解析，(R)ARP 请求和应答报文的数据帧格式如下：



上图中有两个目的硬件地址，链路层首部中目的端硬件地址在 ARP 请求报文被设置为全 1，表示链路广播，在 ARP 应答报文中被设置为发送 ARP 请求报文的主机硬件地址。需要解析的目的硬件地址即对应目的 IP 地址的硬件地址，这个字段在 ARP 请求报文中被设置为 0 或者任意值，接收该报文的主机如果检查到目的 IP 地址是其对应地址，则必须回复一个 ARP 应答报文，从而报告其硬件地址。ARP 报文采用链路广播发送方式，而 ARP 应答则是单播发送方式。以上数据帧格式中，源端硬件地址字段和发送端硬件地址字段值是一样的。操作码取值如下：

1：ARP 请求报文；

2：ARP 应答报文；

3：RARP 请求报文；

4：RARP 应答报文。

硬件地址类型字段用于表示使用何种硬件地址，取值为 1 表示 48 比特以太网硬件地址。

协议地址类型字段则表示使用协议的地址类型，取值为 0x0800 表示 32 比特的 IP 地址类型。

硬件地址长度和协议地址长度字段取值分别为 6 和 4。

为了避免每次进行数据帧发送时进行 IP 地址到硬件地址解析所造成的延迟，实现上使用 ARP 缓存机制，即对之前完成的 IP 地址到硬件地址的解析关系进行缓存，这样下一次再进行相同远端的的数据帧发送时，在创建链路层首部时，就可以直接使用缓存中的映射关系，但为了避免远端主机硬件地址可能的变化以及网络结构特殊变化，每个缓存的映射关系都有一个生存期限，一旦过了这个期限，必须重新进行 ARP 机制地址解析过程，从而对现有的映射关系进行更新和覆盖。这个对 IP 地址到硬件地址映射关系的缓存就是我们通常所说的 ARP 缓存。如下图所示是在 Linux 2.4.20-8 内核版本上 ARP 缓存局域网内测试的情况。首先我们对 ARP 缓存进行清空，然后 PING 远端主机，由于 ARP 缓存为空，那么首先将有一个 ARP 地址解析的过程，之后我们检查 ARP 缓存，缓存中有一项远端主机 IP 地址到硬件地址的映射关系项。下一次发送数据帧到这个远端主机时，就需要进行 ARP 地址解析过程，直接使用 ARP 缓存中的项目即可，从而减少了发送延迟时间。

```
[root@localhost linux]# arp
[root@localhost linux]# ping 192.168.0.1 -c 1
PING 192.168.0.1 (192.168.0.1) 56(84) bytes of data.
64 bytes from 192.168.0.1: icmp_seq=1 ttl=255 time=13.4 ms

--- 192.168.0.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 13.418/13.418/13.418/0.000 ms
[root@localhost linux]# arp -a
? (192.168.0.1) at 00:1B:11:44:E4:E2 [ether] on eth0
[root@localhost linux]# _
```

```
[root@localhost linux]# ip address show eth0
2: eth0: <BROADCAST,MULTICAST,PROMISC,UP> mtu 1500 qdisc pfifo_fast qlen 100
    link/ether 00:0c:29:74:38:2c brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.6/24 brd 192.168.0.255 scope global eth0
```

下面是对应上图中 ARP 缓存清空情况下，对远端主机 PING 过程中 tcpdump 程序抓取的 ARP 请求和应答报文以及 ICMPecho 请求和应答报文：注意 ARP 应答报文中回复的硬件地址与 ARP 缓存中硬件地址相同。

```
[linux@localhost linux]$ tcpdump arp or icmp
tcpdump: listening on eth0
23:42:40.898865 arp who-has 192.168.0.1 tell 192.168.0.6
23:42:40.908048 arp reply 192.168.0.1 is-at 0:1b:11:44:e4:e2
23:42:40.911578 192.168.0.6 > 192.168.0.1: icmp: echo request (DF)
23:42:40.912178 192.168.0.1 > 192.168.0.6: icmp: echo reply (DF)
```

下面我们即对 ARP 协议实现文件 arp.c 进行分析。

2.36 net/inet/arp.c 文件

```
1   /* linux/net/inet/arp.c
2    *
3    * Copyright (C) 1994 by Florian    La Roche
4    *
5    * This module implements the Address Resolution Protocol ARP (RFC 826),
6    * which is used to convert IP addresses (or in the future maybe other
7    * high-level addresses into a low-level hardware address (like an Ethernet
8    * address).
9    *
10   * FIXME:
11   *    Experiment with better retransmit timers
12   *    Clean up the timer deletions
13   *    If you create a proxy entry set your interface address to the address
14   *    and then delete it, proxies may get out of sync with reality - check this
15   *
16   * This program is free software; you can redistribute it and/or
17   * modify it under the terms of the GNU General Public License
18   * as published by the Free Software Foundation; either version
19   * 2 of the License, or (at your option) any later version.
20   *
```

```
21  *
22  * Fixes:
23  *      Alan Cox:       Removed the ethernet assumptions in Florian's code
24  *      Alan Cox:       Fixed some small errors in the ARP logic
25  *      Alan Cox:       Allow >4K in /proc
26  *      Alan Cox:       Make ARP add its own protocol entry
27  *
28  *              Ross Martin     :       Rewrote arp_rcv() and arp_get_info()
29  *      Stephen Henson  :       Add AX25 support to arp_get_info()
30  *      Alan Cox:       Drop data when a device is downed.
31  *      Alan Cox:       Use init_timer().
32  *      Alan Cox:       Double lock fixes.
33  *      Martin Seine    :       Move the arphdr structure
34  *                              to if_arp.h for compatibility.
35  *                              with BSD based programs.
36  *              Andrew Tridgell :       Added ARP netmask code and
37  *                                      re-arranged proxy handling.
38  *      Alan Cox:       Changed to use notifiers.
39  *      Niibe Yutaka    :       Reply for this device or proxies only.
40  *      Alan Cox:       Don't proxy across hardware types!
41  */

42  #include <linux/types.h>
43  #include <linux/string.h>
44  #include <linux/kernel.h>
45  #include <linux/sched.h>
46  #include <linux/config.h>
47  #include <linux/socket.h>
48  #include <linux/sockios.h>
49  #include <linux/errno.h>
50  #include <linux/if_arp.h>
51  #include <linux/in.h>
52  #include <linux/mm.h>
53  #include <asm/system.h>
54  #include <asm/segment.h>
55  #include <stdarg.h>
56  #include <linux/inet.h>
57  #include <linux/netdevice.h>
58  #include <linux/etherdevice.h>
59  #include "ip.h"
60  #include "route.h"
61  #include "protocol.h"
62  #include "tcp.h"
63  #include <linux/skbuff.h>
```

```
64  #include "sock.h"
65  #include "arp.h"
66  #ifdef CONFIG_AX25
67  #include "ax25.h"
68  #endif


69  /*
70   *  This structure defines the ARP mapping cache. As long as we make changes
71   *  in this structure, we keep interrupts of. But normally we can copy the
72   *  hardware address and the device pointer in a local variable and then make
73   *  any "long calls" to send a packet out.
74   */

75  struct arp_table
76  {
77      struct arp_table    *next;          /* Linked entry list        */
```

next 字段用于 arp_table 结构之间的相互串接，构成一个链表。

```
78      unsigned long       last_used;  /* For expiry               */
```

last_used 字段用于表示该表项上次被使用的时间，如果上次使用时间在 10 分钟之间，则清除该表项。这种动态创建和删除 ARP 表项的方式使得 ARP 缓存中维护的都是有效映射关系，可以反映出硬件地址可能的改变（如某台主机可能更换了网卡设备）。

```
79      unsigned int        flags;          /* Control status           */
80      unsigned long       ip;         /* ip address of entry      */
81      unsigned long       mask;   /*netmask-used for generalised proxy arps (tridge) */
```

flags 字段用于维护 ARP 表项的一些标志位，如表项当前所处状态，可以为以后的 ARP 缓存表项的功能扩展做好准备。
ip 字段表示表项所表示映射关系中的 IP 地址；mask 字段为对应 IP 地址的网络掩码。

```
82      unsigned char       ha[MAX_ADDR_LEN];/* Hardware address         */
83      unsigned char       hlen;       /* Length of hardware address   */
84      unsigned short      htype;          /* Type of hardware in use  */
```

ha 字段表示映射关系中的硬件地址，hlen 为硬件地址长度，htype 为硬件地址类型。

```
85      struct device       *dev;           /* Device the entry is tied to  */
```

dev 字段表示该 ARP 表项绑定的网络设备，对于只有一个网络接口的主机，所有数据包的发送当然只有一个发送通道，但对于具有两个或者更多网络接口的主机，数据包就有多种可

能的选择，如果每次发送时都进行发送接口的查询会不必要的增加系统开销，由于数据帧创建中链路层首部的创建都需要进行硬件地址解析，即查询 ARP 缓存，所以在 ARP 表项中维护一个对应发送接口的指针，就可以在进行数据帧创建时一并解决通过哪个网络接口发送数据包的问题。另外对于多个网络接口的主机，每个网口都接在不同的网络上，而远端主机只可能属于一个网络，所以我们在进行 ARP 表项创建时可以知道这个主机属于哪个网络，从而初始化接入该网络的网口设备指针。由此可见，ARP 缓存表项中 dev 字段值将来自于路由表项。而路由表项要么是手工配置，要么根据运行路由协议创建，而根据接收路由数据包的网络接口我们可以进行路由表项中网口设备字段的初始化。

```
86          /*
87           *  The following entries are only used for unresolved hw addresses.
88           */

89          struct timer_list       timer;          /* expire timer         */
90          int                     retries;        /* remaining retries    */
```

timer 字段用于定时重发 ARP 请求数据包，以防止 ARP 请求未得到响应。不过重发的次数是有限制的，当前设置的最大次数为 3，由 ARP_MAX_TRIES 变量（下文介绍）表示。ARP 请求数据包发送间隔为 25 秒，由 ARP_RES_TIME 变量表示。

```
91          struct sk_buff_head     skb;            /* list of queued packets       */
```

最后一个字段 skb 表示暂时由于 IP 地址到硬件地址的解析未完成而被缓存的数据包队列该字段的具体使用在下文中介绍到相关函数时进行说明。

```
92  };
```

75-92 行代码定义了 arp_table，该结构被用于表示 ARP 缓存中的表项。其作用类似于 rtable 结构被用于路由表以及 ip_fw 结构被用于防火墙。ARP 缓存用于维护 MAC 地址到 IP 地址的映射关系，每个表项都有一个生存期，在本版本中被设置为 10 分钟，如果一个表项在 10 分钟中没有被使用，则清除该表项。每个表项由一个 arp_table 结构表示，将这些个 arp_table 结构有组织的串联起来，就构成了 ARP 缓存。从下文分析来看，ARP 缓存的维护是通过数组加链表的形式，每个数组元素指向一个链表，链表是由 Hash 值相同的 arp_table 结构构成。在进行 ARP 表项的查找时，以被解析的 IP 地址为关键字，首先根据 IP 地址进行 Hash 运算，在数组中寻找到对应元素指向的链表，之后遍历该链表，逐一比较 IP 地址是否相同，如果相同，则返回对应表项，即对应的 arp_table 结构。

```
93  /*
94   *  Configurable Parameters (don't touch unless you know what you are doing
95   */

96  /*
97   *  If an arp request is send, ARP_RES_TIME is the timeout value until the
```

```
98      *    next request is send.
99      */

100  #define ARP_RES_TIME            (250*(HZ/10))

101  /*
102   *    The number of times an arp request is send, until the host is
103   *    considered unreachable.
104   */

105  #define ARP_MAX_TRIES           3

106  /*
107   *    After that time, an unused entry is deleted from the arp table.
108   */

109  #define ARP_TIMEOUT             (600*HZ)

110  /*
111   *    How often is the function 'arp_check_retries' called.
112   *    An entry is invalidated in the time between ARP_TIMEOUT and
113   *    (ARP_TIMEOUT+ARP_CHECK_INTERVAL).
114   */

115  #define ARP_CHECK_INTERVAL      (60 * HZ)
```

100，105，109，115 行定义了 ARP 协议实现中使用的几个常量值。ARP_RES_TIME 表示重发 ARP 请求数据包的时间间隔；ARP_MAX_TRIES 表示在放弃之前重发 ARP 请求的次数；ARP_TIMEOUT 表示一个未使用 ARP 表项最长驻留时间，超过此时间后，该表项将从 ARP 缓存中清除；ARP_CHECK_INTERVAL 表示对 ARP 缓存进行检查的时间间隔，系统每个一段时间就对 ARP 缓存中各表项的合法性进行检查，清除过期表项。

```
116  enum proxy {
117      PROXY_EXACT=0,
118      PROXY_ANY,
119      PROXY_NONE,
120  };
```

116-120 行定义了 proxy 枚举类型表示代理 ARP 类型，在下文相关函数分析中我们再阐述这些类型的意义。

```
121  /* Forward declarations. */
122  static void arp_check_expire (unsigned long);
123  static struct arp_table *arp_lookup(unsigned long paddr, enum proxy proxy);
```

```
124  static struct timer_list arp_timer =
125          { NULL, NULL, ARP_CHECK_INTERVAL, 0L, &arp_check_expire };
```

124-125 行定义了一个专用于 ARP 协议的系统定时器，用于定期扫描 ARP 缓存，检查缓存中表项，清除过期表项，从而对系统 ARP 缓存进行维护。

```
126  /*
127   * The default arp netmask is just 255.255.255.255 which means it's
128   * a single machine entry. Only proxy entries can have other netmasks
129   *
130  */

131  #define DEF_ARP_NETMASK      (~0)
```

DEF_ARP_NETMASK 常量表示 ARP 缓存中 IP 地址的网络掩码，对于普通 ARP 缓存表项而言，IP 地址就是主机地址，所以其网络掩码为全 1；只有对代理 ARP 表项 IP 地址可以表示为一个网络地址。所谓代理 ARP，通常是指一个服务器或者网关代理局域网内所有主机的数据包转发和接收，即此时局域网对外只有一个全局 IP 地址，这个 IP 地址就是服务器或者网关地址，而局域网内所有主机使用专门的内部地址如 192.168.x.x B 类 IP 地址。其他代理 ARP 使用的地方如移动 IPv4 中的家乡代理使用代理 ARP 进行数据包的拦截，从而转发数据包到移动节点。

```
132  /*
133   *    The size of the hash table. Must be a power of two.
134   *    Maybe we should remove hashing in the future for arp and concentrate
135   *    on Patrick Schaaf's Host-Cache-Lookup...
136   */

137  #define ARP_TABLE_SIZE    16

138  /* The ugly +1 here is to cater for proxy entries. They are put in their
139       own list for efficiency of lookup. If you don't want to find a proxy
140       entry then don't look in the last entry, otherwise do
141  */

142  #define FULL_ARP_TABLE_SIZE (ARP_TABLE_SIZE+1)

143  struct arp_table *arp_tables[FULL_ARP_TABLE_SIZE] =
144  {
145          NULL,
146  };
```

```
147 /*
148  *  The last bits in the IP address are used for the cache lookup.
149  *          A special entry is used for proxy arp entries
150  */

151 #define HASH(paddr)          (htonl(paddr) & (ARP_TABLE_SIZE - 1))
152 #define PROXY_HASH       ARP_TABLE_SIZE
```

所谓 ARP 缓存，在实现上采用数组加链表（或者称为队列）的方式，每个 ARP 缓存表项由一个 arp_table 结构表示，所以数组中每个元素就指向一个 arp_table 结构类型的链表。对于具体数组元素的寻址由被解析的 IP 地址通过 Hash 算法（151 行）而得。所以查询 ARP 缓存的过程就可表述为：首先根据被解析 IP 地址索引数组中对应元素指向的 arp_table 结构链表，然后遍历该链表，对 IP 地址进行匹配，如果查找一个 IP 地址精确匹配的表项，则返回该表项，如果没有寻找到，则表示当前 ARP 缓存中没有对应表项，系统此时将创建一个新的 ARP 表项，并发出 ARP 请求报文，而当前发送的数据包就被缓存到该表项设置的数据包缓存队列中，当接收到 ARP 应答报文时，一方面完成原先表项的创建（硬件地址字段的初始化），另一方面将该表项中之前缓存的所有待发送数据包发送出去。

142 行在设置表示 ARP 缓存的数组大小时加 1 的含义是数组中最后一个元素用于代理 ARP 表项。ARP 代理队列的查询不进行 Hash 算法寻址，直接查找数据最后一个元素指向的队列即可。所以 151 行 Hash 值的计算中并没有包含代理 ARP 队列。代理 ARP 队列的 Hash 值（用于索引 ARP 缓存数组元素）由 152 行 PROXY_HASH 常量专门表示。

143-146 行定义的 arp_tables 数组即表示 ARP 缓存。每个数组元素都指向一个 arp_table 结构类型的链表。

```
153 /*
154  *  Check if there are too old entries and remove them. If the ATF_PERM
155  *  flag is set, they are always left in the arp cache (permanent entry).
156  *  Note: Only fully resolved entries, which don't have any packets in
157  *  the queue, can be deleted, since ARP_TIMEOUT is much greater than
158  *  ARP_MAX_TRIES*ARP_RES_TIME.
159  */

160 static void arp_check_expire(unsigned long dummy)
161 {
```

arp_check_expire 函数清除过期的 ARP 缓存中表项。

```
162     int i;
163     unsigned long now = jiffies;
164     unsigned long flags;
165     save_flags(flags);
166     cli();
```

```
167     for (i = 0; i < FULL_ARP_TABLE_SIZE; i++)
168     {
169         struct arp_table *entry;
170         struct arp_table **pentry = &arp_tables[i];

171         while ((entry = *pentry) != NULL)
172         {
173             if ((now - entry->last_used) > ARP_TIMEOUT
174                 && !(entry->flags & ATF_PERM))
175             {
176                 *pentry = entry->next;    /* remove from list */
177                 del_timer(&entry->timer);     /* Paranoia */
178                 kfree_s(entry, sizeof(struct arp_table));
179             }
180             else
181                 pentry = &entry->next;  /* go to next entry */
182         }
183     }
```

167-183 行代码遍历系统 ARP 缓存表（即 arp_tables 数组），对其他每个表项的上次使用时间标志进行检查，如果检查到上次使用时间已经在 ARP_TIMEOUT 时间之前，就表示这是一个过期的表项，对于过期的表项我们进一步检查其是否设置了 ATF_PERM 标志位，对于设置 ATF_PERM 标志的表项，表示该表项是用户设置的永久表项，不可被定时清除；如果没有设置 ATF_PERM 标志位，则对于过期表项的处理是：从 ARP 缓存表中删除，停止该表项设置的定时器，最后对该表项占用的内存使用空间进行释放。
以上 170 行代码和 171 行结合对数组中每个元素指向的 arp_table 结构类型组成的 ARP 缓存表项进行检查。

```
184     restore_flags(flags);

185     /*
186      *  Set the timer again.
187      */

188     del_timer(&arp_timer);
189     arp_timer.expires = ARP_CHECK_INTERVAL;
190     add_timer(&arp_timer);
```

188-190 重新设置定时期，为下一次的检查进行定时。注意定时间隔设置为 ARP_CHECK_INTERVAL，即 60 秒。

```
191 }
```

arp_check_expire 函数检查 ARP 缓存中过期表项并对之进行清除。以下两种情况即便表项已经过期也不可进行清除：

1〉 表项 ATF_PERM 标志位被设置，即表示这是一个用户配置的永久表项，不可被清除。

2〉 如果表项中缓存有未发送出去的数据包，则也不可被清除。

对于以上第二点，仅仅是一个条件，实际上这种情况不可能发生：即过期表项中存在有被缓存的尚未发送出去的数据包。数据包被缓存没有发送出去，表示对应表项尚未完成映射关系，即尚处于 ARP 请求报文的发送过程中，而发送最大次数为 ARP_MAX_TRIES，发送时间间隔为 ARP_RES_TIME，表项过期时间为 ARP_TIMEOUT，而 ARP_TIMEOUT>ARP_MAX_TRIES*ARP_RES_TIME，一旦所有的 ARP 请求报文得不到回复，则该 ARP 表项连同之前缓存的所有数据包即都被释放，根本不会轮到 arp_check_expire 函数来处理。

```
192 /*
193  *  Release all linked skb's and the memory for this entry.
194  */

195 static void arp_release_entry(struct arp_table *entry)
196 {
197     struct sk_buff *skb;
198     unsigned long flags;

199     save_flags(flags);
200     cli();
201     /* Release the list of `skb' pointers. */
202     while ((skb = skb_dequeue(&entry->skb)) != NULL)
203     {
204         skb_device_lock(skb);
205         restore_flags(flags);
206         dev_kfree_skb(skb, FREE_WRITE);
207     }
208     restore_flags(flags);
209     del_timer(&entry->timer);
210     kfree_s(entry, sizeof(struct arp_table));
211     return;
212 }
```

arp_release_entry 函数专门被用于释放一个 ARP 缓存表项，被释放的表项以参数形式传入。该函数将被 arp_expire_request 函数调用在所有 ARP 请求报文未得到应答的情况下将一个未完成映射关系创建的 ARP 表项释放掉。因为此时该表项中可能缓存有等待发送的数据包，所以释放工作并不简单是释放表项本身，还包括之前被缓存在该表项中的待发送数据包。注意每个表项都有一个定时器，所以在释放一个表项时必须同时将定时器从系统中删除，这一点很重要，否则会造成内核状态的不一致（所以内核状态的不一致通常指出现内核资源泄

露，即由于系统漏洞，有些资源不再在内核的管理范围之内，在极端情况下，可能会造成系统崩溃，如当释放一个表项，但没有删除该表项对应的定时器，当定时器到期后，相关函数被调用，而该函数会访问对应表项，但是这个对应表项已经被释放，就会造成非法访问内存的系统错误）。

```
213 /*
214  *  Purge a device from the ARP queue
215  */

216 int arp_device_event(unsigned long event, void *ptr)
217 {
218     struct device *dev=ptr;
219     int i;
220     unsigned long flags;

221     if(event!=NETDEV_DOWN)
222         return NOTIFY_DONE;
223     /*
224      *  This is a bit OTT - maybe we need some arp semaphores instead.
225      */

226     save_flags(flags);
227     cli();
228     for (i = 0; i < FULL_ARP_TABLE_SIZE; i++)
229     {
230         struct arp_table *entry;
231         struct arp_table **pentry = &arp_tables[i];

232         while ((entry = *pentry) != NULL)
233         {
234             if(entry->dev==dev)
235             {
236                 *pentry = entry->next;   /* remove from list */
237                 del_timer(&entry->timer);     /* Paranoia */
238                 kfree_s(entry, sizeof(struct arp_table));
239             }
240             else
241                 pentry = &entry->next;  /* go to next entry */
242         }
243     }
244     restore_flags(flags);
245     return NOTIFY_DONE;
246 }
```

arp_device_event 函数是对系统事件做出响应的函数，系统事件主要指网络设备停止工作，即 NETDEV_DOWN 事件。因为每个 ARP 表项都绑定在一个网络设备上，如果对应的网络设备不再工作，则这些被绑定的表项就不可再被使用，相当于我们关闭了一个发送端口，那么需要经过该端口发送的所有数据包将都无法发送，为了尽量降低系统开销，这种无法发送的判断应该越早越好，所以系统通过清除相关 ARP 表项完成，因为在创建一个待发送数据帧时，都需要进行 ARP 缓存的查询。另外维护无法有效使用的 ARP 表项本身就是一个开销。所以一旦某个网络设备停止工作，则系统将从 ARP 缓存中清除所有与之绑定的 ARP 表项。arp_device_event 函数完成的任务即是如此。函数实现较为简单，此处不再一一论述。注意 221-222 行表示涉及到 ARP 缓存时，只需对 NETDEV_DOWN 事件做出响应。

```
247 /*
248  *  Create and send an arp packet. If (dest_hw == NULL), we create a broadcast
249  *  message.
250  */

251 void arp_send(int type, int ptype, unsigned long dest_ip,
252                   struct device *dev, unsigned long src_ip,
253                   unsigned char *dest_hw, unsigned char *src_hw)
254 {
255       struct sk_buff *skb;
256       struct arphdr *arp;
257       unsigned char *arp_ptr;

258       /*
259        *  No arp on this interface.
260        */

261       if(dev->flags&IFF_NOARP)
262             return;

263       /*
264        *  Allocate a buffer
265        */

266       skb = alloc_skb(sizeof(struct arphdr)+ 2*(dev->addr_len+4)
267                         + dev->hard_header_len, GFP_ATOMIC);
268       if (skb == NULL)
269       {
270             printk("ARP: no memory to send an arp packet\n");
271             return;
272       }
273       skb->len = sizeof(struct arphdr) + dev->hard_header_len + 2*(dev->addr_len+4);
274       skb->arp = 1;
```

```
275         skb->dev = dev;
276         skb->free = 1;

277         /*
278           *    Fill the device header for the ARP frame
279           */

280
      dev->hard_header(skb->data,dev,ptype,dest_hw?dest_hw:dev->broadcast,src_hw?src_hw:N
ULL,skb->len,skb);

281         /* Fill out the arp protocol part. */
282         arp = (struct arphdr *) (skb->data + dev->hard_header_len);
283         arp->ar_hrd = htons(dev->type);
284  #ifdef CONFIG_AX25
285         arp->ar_pro     =     (dev->type     !=     ARPHRD_AX25)?     htons(ETH_P_IP)     :
htons(AX25_P_IP);
286  #else
287         arp->ar_pro = htons(ETH_P_IP);
288  #endif
289         arp->ar_hln = dev->addr_len;
290         arp->ar_pln = 4;
291         arp->ar_op = htons(type);

292         arp_ptr=(unsigned char *)(arp+1);

293         memcpy(arp_ptr, src_hw, dev->addr_len);
294         arp_ptr+=dev->addr_len;
295         memcpy(arp_ptr, &src_ip,4);
296         arp_ptr+=4;
297         if (dest_hw != NULL)
298               memcpy(arp_ptr, dest_hw, dev->addr_len);
299         else
300               memset(arp_ptr, 0, dev->addr_len);
301         arp_ptr+=dev->addr_len;
302         memcpy(arp_ptr, &dest_ip, 4);

303         dev_queue_xmit(skb, dev, 0);
304  }
```

arp_send 函数用于构建一个 ARP 请求报文并发送出去。结合前文中对 ARP 协议报文的格式说明，以及其他协议报文创建方式，该函数代码比较容易理解。注意由于 ARP 请求报文不包含任何用户数据，所以报文长度是固定的，273 行即对对应报文的 sk_buff 结构中 len 字段进行了初始化。另外由于 ARP 请求报文是为了解析 IP 地址到硬件地址的映射关系，所以其

本身并不需要进行这种解析，所以 274 行将 arp 字段设置为 1，这样避免系统调用循环。另外一点需要注意的是 ARP 请求报文中目的硬件地址设置为链路广播地址，即全 1，同一链路上所有主机都会对该 ARP 请求报文进行接收，如果某台主机检测到被解析 IP 地址即是其自身 IP 地址，则其以单播方式回复一个 ARP 应答报文，从而完成解析过程。280 行调用 eth_header 函数创建 MAC 首部时，目的硬件地址即使用就是链路广播地址（全 1 地址）。所以无需进行任何地址解析过程。而 ARP 协议首部的创建读者结合 ARP 协议首部格式很容易理解，此处不再讨论。

```
305 /*
306  *  This function is called, if an entry is not resolved in ARP_RES_TIME.
307  *  Either resend a request, or give it up and free the entry.
308  */

309 static void arp_expire_request (unsigned long arg)
310 {
```

arp_expire_request 函数处理 ARP 请求报文的重发，ARP 请求报文的重发原因：其一可能是之前的报文由于某种原因被丢失（由于 ARP 请求报文不会被转发，被丢失的可能性很小）；其二目的主机没有及时进行处理；其三在链路上根本没有对应被解析的 IP 地址的主机。无论何种原因，总之就是之前发送的 ARP 请求没有得到应答，需要重新发送，系统可以支持最大的发送次数由 ARP_MAX_TRIES 常量表示，目前该常量值被设置为 3，即在最终放弃解析之前，最多可以发送 3 个 ARP 请求报文。如果 3 个 ARP 请求报文都未得到应答，则表示 IP 地址无法解析，此时将调用 arp_release_entry 函数对之前创建的对应 ARP 表项进行释放。arp_release_entry 函数具体完成的功能在前文中已有论述。下面我们一步步分析 arp_expire_request 函数的代码实现的功能。

```
311     struct arp_table *entry = (struct arp_table *) arg;
312     struct arp_table **pentry;
313     unsigned long hash;
314     unsigned long flags;

315     save_flags(flags);
316     cli();

317     /*
318      *  Since all timeouts are handled with interrupts enabled, there is a
319      *  small chance, that this entry has just been resolved by an incoming
320      *  packet. This is the only race condition, but it is handled...
321      */

322     if (entry->flags & ATF_COM)
323     {
324         restore_flags(flags);
325         return;
```

```
326         }
```

如果当前被解析表项中 ATF_COM 标志位被设置，则表示该表项已经完成解析过程，即这是一个有效的可用的表项，IP 地址到硬件地址的映射关系已经完成。如果 ATF_COM 标志位没有设置，则表示我们需要重新发送一个 ARP 请求报文。

```
327         if (--entry->retries > 0)
328         {
329             unsigned long ip = entry->ip;
330             struct device *dev = entry->dev;

331             /* Set new timer. */
332             del_timer(&entry->timer);
333             entry->timer.expires = ARP_RES_TIME;
334             add_timer(&entry->timer);
335             restore_flags(flags);
336             arp_send(ARPOP_REQUEST, ETH_P_ARP, ip, dev, dev->pa_addr,
337                     NULL, dev->dev_addr);
338             return;
339         }
```

327-339 行代码检查重新发送 ARP 请求报文的次数是否已经超过最大此处，在创建一个未完成映射关系的 ARP 表项时，该表项 retries 字段被设置为可重新发送 ARP 请求报文的最大次数，即 ARP_MAX_TRIES 常量表示的量（3），之后每次重新发送一个 ARP 请求报文，retries 字段值就被减一，直至减到 0 为止。如果减到 0 时该表项仍未完成映射关系的解析，则释放该表项。327-339 行代码对应 retries 字段值尚未减到 0 的情况，此时还可以继续发送一个 ARP 请求报文，这是通过 arp_send 函数完成的，另外再次设置重发定时器，为下一次可能的重新发送 ARP 请求报文做好预备。注意重发 ARP 请求报文的间隔时间被设置为 ARP_RES_TIME，即 25 秒。

```
340         /*
341          *  Arp request timed out. Delete entry and all waiting packets.
342          *  If we give each entry a pointer to itself, we don't have to
343          *  loop through everything again. Maybe hash is good enough, but
344          *  I will look at it later.
345          */
```

以下代码对应发送资格被用完的情况，即已经发送了 ARP_MAX_TRIES 个 ARP 请求报文，但没有得到任何 ARP 应答，此时表示解析过程失败，下面所要做的就是对之前创建的表项进行释放。

```
346         hash = HASH(entry->ip);

347         /* proxy entries shouldn't really time out so this is really
```

```
348             only here for completeness
349         */
350         if (entry->flags & ATF_PUBL)
351             pentry = &arp_tables[PROXY_HASH];
352         else
353             pentry = &arp_tables[hash];
```

此处要释放一个表项，首先我们必须对其进行定位，因为 ARP 缓存的维护是通过单向队列完成的，我们无法只根据表项的指针将其从队列中删除。根据表项中 ATF_PUBL 标志位是否设置，我们根据不同的算法对表项所在队列进行定位，如果 ATF_PUBL 标志位被设置，则表示这是一个代理表项，此时直接遍历 arp_tables 数组的最后一个元素指向的队列即可；如果 ATF_PUBL 标志位没有设置，则表示这是一个普通 ARP 表项，此时根据 IP 地址 Hash 运算值在 arp_tables 数组中索引对应元素指向的队列。由此可见，ATF_PUBL 标志位的作用是区分普通 ARP 表项和代理 ARP 表项。

```
354         while (*pentry != NULL)
355         {
356             if (*pentry == entry)
357             {
358                 *pentry = entry->next;    /* delete from linked list */
359                 del_timer(&entry->timer);
360                 restore_flags(flags);
361                 arp_release_entry(entry);
362                 return;
363             }
364             pentry = &(*pentry)->next;
365         }
```

在对当前被处理表项进行定位后（即寻找到其所在的队列），我们就可以对该队列进行遍历，在查找到该表项的具体位置后，将该表项从队列中删除，删除该表项设置的系统定时器，此后就可以安全的调用 arp_release_entry 函数对该表项进行释放了。

```
366         restore_flags(flags);
367         printk("Possible ARP queue corruption.\n");
368         /*
369          *    We should never arrive here.
370          */
371 }
```

综合以上对 arp_expire_request 函数实现代码的分析，我们可以简单的将 arp_expire_request 函数实现功能总结如下：

1〉 如果发送资格尚未用完，则重发一个 ARP 请求报文，同时重新设置重发定时器为下一次的超时重发做好准备。

2〉 否则，释放当前被解析的表项。

```
372 /*
373  *  This will try to retransmit everything on the queue.
374  */

375 static void arp_send_q(struct arp_table *entry, unsigned char *hw_dest)
376 {
377     struct sk_buff *skb;

378     unsigned long flags;

379     /*
380      *  Empty the entire queue, building its data up ready to send
381      */

382     if(!(entry->flags&ATF_COM))
383     {
384         printk("arp_send_q: incomplete entry for %s\n",
385                 in_ntoa(entry->ip));
386         return;
387     }

388     save_flags(flags);

389     cli();
390     while((skb = skb_dequeue(&entry->skb)) != NULL)
391     {
392         IS_SKB(skb);
393         skb_device_lock(skb);
394         restore_flags(flags);
395         if(!skb->dev->rebuild_header(skb->data,skb->dev,skb->raddr,skb))
396         {
397             skb->arp    = 1;
398             if(skb->sk==NULL)
399                 dev_queue_xmit(skb, skb->dev, 0);
400             else
401                 dev_queue_xmit(skb,skb->dev,skb->sk->priority);
402         }
403         else
404         {
405             /* This routine is only ever called when 'entry' is
406                complete. Thus this can't fail. */
407             printk("arp_send_q: The impossible occurred. Please notify Alan.\n");
408             printk("arp_send_q: active entity %s\n",in_ntoa(entry->ip));
```

```
409                     printk("arp_send_q: failed to find %s\n",in_ntoa(skb->raddr));
410             }
411         }
412         restore_flags(flags);
413 }
```

我们在前文中一再说到对应每个 ARP 表项都有一个待发送数据包缓存队列。这个队列是由表示 ARP 表项的 arp_table 结构中 skb 字段指向的。当系统发送一个数据包，在进行 MAC 首部创建时，如果无法立刻完成 IP 地址到 MAC 层硬件地址的解析，即当前 ARP 缓存中没有对应的表项，则会创建一个新的 ARP 表项表示当前的映射关系，并将带发送的数据包缓存到这个新创建的 ARP 表项缓存队列中。当然此时新表项中映射关系中的 MAC 地址部分尚未进行赋值，为完成 ARP 表项中所有字段（主要是硬件地址字段）的初始化，系统发出一个 ARP 请求报文，对 IP 地址进行解析，希望对应 IP 地址的远端主机回复一个 ARP 应答报文，从而完成解析过程，并最终完成 ARP 表项的创建。此时如果该 ARP 表项缓存队列中有之前由于没有完成地址解析而被缓存的数据包，则 ARP 协议实现必须在完成地址解析过程后将这些数据包发送出去。arp_send_q 函数就是在完成地址解析后被调用发送 ARP 表项缓存队列中数据包的函数，可以想见，这个函数将被处理 ARP 应答报文的函数调用，这个函数就是 ARP 协议数据包接收总入口函数 arp_rcv，这个函数的作用类同于 ip_rcv，在链路层模块检测到 MAC 首部中协议字段表示一个 ARP 协议数据包时，就调用 arp_rcv 函数进行处理。我们从前文分析中知道，链路层维护对上层调用的统一接口，所以需要直接接收从链路层模块上传的数据包的模块必须实现一个 packet_type 结构，并向链路层模块进行注册，为从链路层模块接收数据包，ARP 协议当然也不例外，这一点将在下文代码的分析中得到验证。

382-387 行对表项的状态进行验证以防万一，如果表项中 ATF_COM 标志位没有被设置，则表示该表项尚未完成地址解析过程，此时不可使用该表项进行 MAC 首部的创建，当然也就无法发送数据包，所以直接返回。

390-411 行完成表项缓存队列中数据包的发送工作，395 行完成 MAC 首部的创建，因为数据包被缓存到该队列中根本原因就在于之前无法完成 MAC 首部的创建，因为缺少目的 MAC 地址，在经过 ARP 协议模块的一番努力后（地址解析过程），现在终于得到远端主机的 MAC 地址，现在就可以完成数据帧的 MAC 首部创建了，而完成 MAC 首部的创建，就表示可以将该数据帧发送出去了，函数中直接调用链路层数据包发送接口函数 dev_queue_xmit 将数据包发往下层-链路层进行处理并调用网卡驱动程序发送函数最终将数据包通过网络设备发送出去。

```
414 /*
415  *  Delete an ARP mapping entry in the cache.
416  */

417 void arp_destroy(unsigned long ip_addr, int force)
418 {
```

arp_destroy 函数用于删除一个 ARP 表项，参数说明：

ip_addr：表示被删除表项对应的 IP 地址。
force：表示如果被删除表项中 ATF_PERM 标志位被设置，是否进行强制删除。ATF_PERM 标志位被设置表示这是一个用户设置的永久表项，一般操作（如定时清除操作）动不了该表项，所以此处设置 force 参数表示即便是设置了 ATF_PERM 标志位的表项，只要 IP 地址符合，也要将其删除。

```
419         int checked_proxies = 0;
420         struct arp_table *entry;
421         struct arp_table **pentry;
422         unsigned long hash = HASH(ip_addr);

423  ugly:
424         cli();
425         pentry = &arp_tables[hash];
426         if (! *pentry) /* also check proxy entries */
427             pentry = &arp_tables[PROXY_HASH];
```

422 行首先根据 IP 地址计算 Hash 值索引 arp_tables 数组元素，425 行将 pentry 变量设置为指向对应队列，这个队列是由 422 行计算的 Hash 值队 arp_tables 数组进行索引得到。426-427 行代码表达的含义是如果经过通常 Hash 运算得到的队列为空，则对代理 ARP 队列进行检查。注意 423 行 ugly 标志符。首先需要记住的是系统 ARP 缓存中不可具有相同 IP 地址的两个重复表项，如此就会发生冲突。425-427 行代码结合下面的有关代码本来要表达的含义是如果在普通 ARP 表项队列中没有查找到满足条件的表项，则继续在代理 ARP 表项队列中进行查找。而一旦在普通表项队列中找到一个匹配项，则无需对代理 ARP 表项队列进行查找了，因为系统 ARP 缓存表中不会有具有相同 IP 地址的两个不同表项存在。但是此处代码由一个漏洞，也就是如果在普通 ARP 表项队列中找到一个匹配项，而且该队列中也只有这一个 ARP 表项，则在删除该匹配表项后，原本可以退出的操作现在变成了继续对代理 ARP 表项队列进行查找，因为在 434 行删除了这个匹配表项后，回到 425 行 pentry 将为 NULL，所以 427 行完成赋值，继续对代理 ARP 队列进行检查。而实际上这是不必要的。二 445，446 行代码则对应没有在普通 ARP 表项队列中寻找到匹配项的情况，此时 450 行将对代理 ARP 表项队列进行匹配查找。

```
428         while ((entry = *pentry) != NULL)
429         {
430             if (entry->ip == ip_addr)
431             {
432                 if ((entry->flags & ATF_PERM) && !force)
433                     return;
434                 *pentry = entry->next;
435                 del_timer(&entry->timer);
436                 sti();
437                 arp_release_entry(entry);
438                 /* this would have to be cleaned up */
439                 goto ugly;
```

```
440                 /* perhaps like this ?
441                 cli();
442                 entry = *pentry;
443                 */
444             }
445             pentry = &entry->next;
446             if (!checked_proxies && ! *pentry)
447               { /* ugly. we have to make sure we check proxy
448                    entries as well */
449                 checked_proxies = 1;
450                 pentry = &arp_tables[PROXY_HASH];
451               }
452         }
```

430-444 行代码具体对应查询到一个满足删除条件的表项的情况，此时将该表项释放，并将 pentry 变量指向队列中下一表项，在 439 行调至 423 行重新对该队列其他表项进行检查。445-451 行对应普通 ARP 表项队列中没有匹配项的情况，此时在遍历完成普通表项队列后，将进一步对代理 ARP 表项进行遍历查找。

```
453         sti();
454 }
```

arp_destroy 函数用于删除一个 ARP 表项，这个函数作为用户控制接口底层实现函数的其中之一。用户可对当前 ARP 缓存表进行操作，这些操作包括添加一个 ARP 表项并且可设置为永久标志位以防定时清除，删除一个 ARP 表项，以及更改一个 ARP 表项。arp_destroy 函数就是响应用户删除一个 ARP 表项的具体底层实现函数。

```
455 /*
456  *  Receive an arp request by the device layer. Maybe I rewrite it, to
457  *  use the incoming packet for the reply. The time for the current
458  *  "overhead" isn't that high...
459  */

460 int arp_rcv(struct sk_buff *skb, struct device *dev, struct packet_type *pt)
461 {
```

arp_rcv 函数之于 ARP 协议的作用如同 ip_rcv 函数之于 IP 协议，arp_rcv 函数是所有使用 ARP 协议进行数据包传送的总入口函数，链路层模块接口函数判定链路层首部中协议字段为 ARP 协议时，调用 arp_rcv 函数进行数据包的处理。本书前文中介绍 IP 协议实现文件时曾多次提到网络层和链路层之间的接口组织方式，即网络层协议通过初始化预定义数据结构 packet_type(netdevice.h)并向链路层进行注册完成网络层协议和链路层协议之间的衔接。注册是通过调用链路层提供的接口函数 dev_pack_add 完成的，该函数将网络层协议注册的 packet_type 结构挂入链路层维护的 packet_type 结构队列中，这个队列由 ptype_base 全局变量指向。IP 协议的注册是在 ip_init 函数中完成的，读者可查阅前文中对 ip.c 文件的分析。

同理 ARP 协议作为网络层协议，其实现模块为了从链路层接收使用 ARP 协议的数据包进行处理，也同样必须初始化一个 packet_type 结构并通过调用 dev_pack_add 函数向链路层模块进行注册，这个注册是在 arp_init 函数中完成的，这在下文中分析到该函数时将进行说明。此处我们继续 arp_rcv 函数的分析。
从 arp_rcv 函数被调用环境来看，其传入参数含义如下：
skb：接收的 ARP 协议数据包；
dev：接收数据包的网络设备；
pt：指向 ARP 协议本身的 packet_type 结构，该参数在函数实现中目前未被使用。

```
462  /*
463   *    We shouldn't use this type conversion. Check later.
464   */

465       struct arphdr *arp = (struct arphdr *)skb->h.raw;
466       unsigned char *arp_ptr= (unsigned char *)(arp+1);
467       struct arp_table *entry;
468       struct arp_table *proxy_entry;
469       int addr_hint,hlen,htype;
470       unsigned long hash;
471       unsigned char ha[MAX_ADDR_LEN]; /* So we can enable ints again. */
472       long sip,tip;
473       unsigned char *sha,*tha;
```

465-473 行是对将要使用变量的声明和初始化。首先 arp 变量被初始化为指向数据帧中 ARP 协议首部。注意内核网络栈实现在将数据包传送给上一层协议模块处理之前，会更新数据包封装结构 sk_buff 结构中 h.raw 字段，使其指向该上一层协议的首部（即跳过当前处理层协议首部字节长度即可），从而便于上一层模块的处理，所以在调用 arp_rcv 函数之前，skb->h.raw 字段在链路层模块中（具体是在 net_bh 函数中）就被更新为指向 ARP 协议首部。但由于 skb->h.raw 字段类型为 char 指针类型，不便于处理，所以 465 行通过类型转换，将这段内存区域定义为 arphdr 类型便于下文中对 ARP 协议首部中各字段的检查和处理。466 行则将 arp_ptr 变量初始化为指向 ARP 首部之后的第一个字段，注意 arp 变量类型为 arphdr，所以 arp+1 表示跳过了一个 ARP 首部的长度。entry，proxy_entry 用于下文中对 ARP 缓存表进行操作。其他变量含义下文分析中自明。

```
474  /*
475   *    The hardware length of the packet should match the hardware length
476   *    of the device.  Similarly, the hardware types should match.  The
477   *    device should be ARP-able.  Also, if pln is not 4, then the lookup
478   *    is not from an IP number.  We can't currently handle this, so toss
479   *    it.
480   */
481       if (arp->ar_hln != dev->addr_len        ||
482                   dev->type != ntohs(arp->ar_hrd) ||
483             dev->flags & IFF_NOARP                ||
```

```
484             arp->ar_pln != 4)
485         {
486             kfree_skb(skb, FREE_READ);
487             return 0;
488         }
```

481-488 行代码对 ARP 首部中各字段进行检查，以判断数据包的有效性。根据上文 ARP 协议首部格式定义，这段代码不难理解。只需注意的是，被比较的对象是网络设备中的相关字段，dev->addr_len 表示网络设备的硬件地址长度，dev->type 表示网段的类型（如以太网类型）。因为网络设备是接收数据包的设备，如果数据包中定义格式与网络设备中规定格式不符，则表示数据包格式不正确，此时直接将数据包作丢弃处理。

```
489 /*
490  *  Another test.
491  *  The logic here is that the protocol being looked up by arp should
492  *  match the protocol the device speaks.  If it doesn't, there is a
493  *  problem, so toss the packet.
494  */
495         switch(dev->type)
496         {
497 #ifdef CONFIG_AX25
498             case ARPHRD_AX25:
499                 if(arp->ar_pro != htons(AX25_P_IP))
500                 {
501                     kfree_skb(skb, FREE_READ);
502                     return 0;
503                 }
504                 break;
505 #endif
506             case ARPHRD_ETHER:
507             case ARPHRD_ARCNET:
508                 if(arp->ar_pro != htons(ETH_P_IP))
509                 {
510                     kfree_skb(skb, FREE_READ);
511                     return 0;
512                 }
513                 break;

514             default:
515                 printk("ARP: dev->type mangled!\n");
516                 kfree_skb(skb, FREE_READ);
517                 return 0;
518         }
```

495-518 行代码进一步对网络层协议类型进行检查，以以太网为例，508 行检查 ARP 首部中网络层协议类型是否为 IP，如否，则表示数据包格式有误，直接丢弃。本质的讲，网络协议就是为了实现各自分离的主机之间的通信而规定的一套数据包封装格式，既然是规定的，所以没有那么多为什么，即对于以太网类型而言，网络协议类型就应该是 ETH_P_IP，如果不符，则数据包格式有误，没有其他的类似 ETH_P_IP 的常量存在以供继续检查。

```
519 /*
520  *  Extract fields
521  */

522       hlen   = dev->addr_len;
523       htype = dev->type;

524       sha=arp_ptr;
525       arp_ptr+=hlen;
526       memcpy(&sip,arp_ptr,4);
527       arp_ptr+=4;
528       tha=arp_ptr;
529       arp_ptr+=hlen;
530       memcpy(&tip,arp_ptr,4);
```

522-530 根据数据包中内容对一些变量进行初始化。首先需要明确的是 arphdr 结构定义中并未包含 ARP 首部中硬件地址和 IP 地址字段，为便于读者理解，此处重新给出 arphdr 结构的定义：

```
/*include/linux/if_arp.h*/
64  struct arphdr
65  {
66        unsigned short ar_hrd;        /* format of hardware address*/
67        unsigned short ar_pro;        /* format of protocol address  */
68        unsigned char  ar_hln;        /* length of hardware address */
69        unsigned char  ar_pln;        /* length of protocol address  */
70        unsigned short ar_op;         /* ARP opcode (command)        */

71  #if 0
72          /*
73           *  Ethernet looks like this : This bit is variable sized however...
74           */
75        unsigned char       ar_sha[ETH_ALEN];    /* sender hardware address    */
76        unsigned char       ar_sip[4];       /* sender IP address       */
77        unsigned char       ar_tha[ETH_ALEN];    /* target hardware address     */
78        unsigned char       ar_tip[4];       /* target IP address       */
79  #endif
80  };
```

即 arp_rcv 函数开始处对 arp_ptr 变量的初始化是将该变量指向了 ARP 首部的 ar_sha 字段。故 524 行将 sha 变量初始化为指向发送端硬件地址，526 行将 sip 初始化为发送端主机 IP 地址，528 行将 tha 变量初始化为指向接收端硬件地址，530 行将 tip 初始化为接收端主机 IP 地址。自此我们可以根据这些变量值进行 ARP 数据包的其他检查和具体处理，如下。

```
531  /*
532   *    Check for bad requests for 127.0.0.1.    If this is one such, delete it.
533   */
534        if(tip == INADDR_LOOPBACK)
535        {
536              kfree_skb(skb, FREE_READ);
537              return 0;
538        }
```

534-538 行检查接收端主机 IP 地址是否为回环地址（127.0.0.1），如是，则表示这个 ARP 数据包是由本机发送的，此时直接丢弃该数据包，因为没有任何处理的必要（ARP 数据包的作用在于进行 IP 地址到硬件地址的解析，没有必要对自己进行解析）。

```
539  /*
540   *    Process entry.    The idea here is we want to send a reply if it is a
541   *    request for us or if it is a request for someone else that we hold
542   *    a proxy for.    We want to add an entry to our cache if it is a reply
543   *    to us or if it is a request for our address.
544   *    (The assumption for this last is that if someone is requesting our
545   *    address, they are probably intending to talk to us, so it saves time
546   *    if we cache their address.    Their address is also probably not in
547   *    our cache, since ours is not in their cache.)
548   *
549   *    Putting this another way, we only care about replies if they are to
550   *    us, in which case we add them to the cache.    For requests, we care
551   *    about those for us and those for our proxies.    We reply to both,
552   *    and in the case of requests for us we add the requester to the arp
553   *    cache.
554   */
```

以上这段注释意为：如果这是一个 ARP 请求报文，并且是发往本机的或者发往由本机进行代理的主机的，那么就回复一个 ARP 应答报文。无论这个 ARP 报文是何类型，只要是发送给本机，本地都将在 ARP 缓存中加入一个发送该 ARP 报文的主机的映射表项，以防将来本机有可能发送数据包到该主机。（这是假设在以下基础之上的：如果有主机在询问本机硬件地址，那么极有可能远处主机将要发送数据包给本机，而本机同样需要发送相关数据包给远处主机，所以预先在 ARP 缓存中加入远处主机的 ARP 表项可以免除本机发送数据包到远处主机时所要进行的 ARP 解析过程。）（作者加：当然，由于 ARP 缓存中每个表项都有一个生存期，如果过了生存期，那么该表项将被清除，到时如果需要发送数据包到相应主机，则重新进行 ARP 解析过程，否则可直接使用 ARP 缓存中表项的内容。）换一种说法，我们只关

心发往本地的 ARP 应答报文，此时我们在 ARP 缓存中加入相关表项；对于 ARP 请求，我们同时注意那些发送给本地的和发送给由本地进行代理的主机的 ARP 请求报文，此时我们都在 ARP 缓存中加入远方请求主机的 ARP 表项，以防将来应答和回复数据包给远方主机之时需要。

如果以上注释并不十分清楚，那么如下再次进行总结说明：
首先 ARP 数据包类型分为两种：ARP 请求数据包，ARP 应答数据包。
对于 ARP 请求数据包，又分为两种情况：其一为发送给本机的 ARP 请求，此时表示远方主机极有可能要和本机进行通信，同时也就表示本机极有可能需要发送数据包给远方主机，那么此时除了对该 ARP 请求进行应答之外，还要在本地 ARP 缓存中加入远方主机的映射表项，避免此后对远方主机同样进行 ARP 解析过程（注意 ARP 请求报文中包含了发送该请求报文的远方主机的硬件地址和 IP 地址）；其二为发送给由本机进行代理的主机的 ARP 请求，这种情况的处理同上，唯一的区别在于此时回复的是代理主机的硬件地址。对于代理 ARP，专门有代理 ARP 缓存，这一点将在下文代码中有所体现。而在前文中 arp.c 文件之初，我们注意到 ARP 缓存实际上是由一个 arp_table 结构指针数组构成，数组的最后一个表项专门用于代理 ARP 表项，即所有的代理 ARP 表项均在由数组最后一个表项指向的 arp_table 结构队列中。普通 ARP 表项则通过散列算法加入到其他数组元素指向的 arp_table 结构队列中。代理 ARP 表项和普通 ARP 表项除了存储位置上的差异之外，最注意的不同在于代理 ARP 表项一般是由用户配置而成，而非根据所接收数据包进行更新；而普通 ARP 表项主要是根据 ARP 解析过程和所接收的 ARP 数据包进行动态更新完成。对于由用户配置而成的代理 ARP 表项，这些表项是完整的，其中有被代理的 IP 地址（或者网络地址）以及对应的硬件地址，当主机接收到一个 ARP 请求报文时，其检查 ARP 请求报文中被请求 IP 地址是否出于被代理范围，如果是，则由本机发送一个 ARP 应答报文进行 ARP 代理应答，从而进行数据包的拦截处理（这在配置有网关的局域网工作环境中经常见到，此时网关完成地址转换作用，使得局域网中主机对外不可见，且只使用一个全局 IP 地址即可。）

对于 ARP 应答数据包，此时的处理比较简单，即完善 ARP 缓存中的相关表项，并将之前由于地址解析未完成而滞留的数据包发送出去。具体情况见下文中代码分析。

```
555         addr_hint = ip_chk_addr(tip);

556         if(arp->ar_op == htons(ARPOP_REPLY))
557         {
558             if(addr_hint!=IS_MYADDR)
559             {
560 /*
561  *  Replies to other machines get tossed.
562  */
563                 kfree_skb(skb, FREE_READ);
564                 return 0;
565             }
566 /*
567  *  Fall through to code below that adds sender to cache.
568  */
```

```
569         }
```

555 行调用 ip_chk_addr 函数检查 ARP 首部中目的端 IP 地址，判断该 ARP 报文是否发送给本机。ip_chk_addr 函数可能返回值有：IS_BROADCAST，IS_MYADDR，IS_MULTICAST，这些常量均定义在 include/linux/netdevice.h，各自的含义从其名称即可看出，addr_hint 变量将在下文中用于对目的端 IP 地址进行检查。
556-569 行代码检查数据包是否为一个 ARP 应答报文，如果是，但是并非发送给本机（558），那么简单丢弃数据包后返回。否则根据该 ARP 应答报文进行 ARP 表项的完善或者创建，这些工作将在下文代码中完成。

```
570         else
571         {
```

570 行这个 else 语句表示数据包是一个 ARP 请求数据包，根据上文中分析，此时存在两种情况：发往本机的或者发送给由本机代理的主机的。

```
572 /*
573  *  It is now an arp request
574  */
575 /*
576  * Only reply for the real device address or when it's in our proxy tables
577  */
578             if(tip!=dev->pa_addr)
579             {
```

578-616 行对应代理的情况，dev->pa_addr 表示协议地址（protocol address），578 行即检查 ARP 请求报文是否发往本机，如果不是，表示有可能发送给由本机代理的主机的。所以下面进行的工作：其一检查目的 IP 地址是否在被代理范围之内；其二根据检查的结果决定是否回复 ARP 应答数据包或者说仅仅是简单丢弃。检查的方式即遍历 ARP 缓存中代理表项，对 IP 地址进行匹配检查，这个工作在如下 586-601 行代码中完成。

```
580 /*
581  *  To get in here, it is a request for someone else.  We need to
582  *  check if that someone else is one of our proxies.  If it isn't,
583  *  we can toss it.
584  */
585                 cli();
586                 for(proxy_entry=arp_tables[PROXY_HASH];
587                     proxy_entry;
588                     proxy_entry = proxy_entry->next)
589                 {
590                   /* we will respond to a proxy arp request
591                      if the masked arp table ip matches the masked
592                      tip. This allows a single proxy arp table
593                      entry to be used on a gateway machine to handle
```

```
594                     all requests for a whole network, rather than
595                     having to use a huge number of proxy arp entries
596                     and having to keep them uptodate.
597                     */
598                 if (proxy_entry->dev != dev && proxy_entry->htype == htype &&
599                     !((proxy_entry->ip^tip)&proxy_entry->mask))
600                     break;
```

在前文中，我们提到 ARP 代理表项专门有一个队列表示，这个队列由 arp_tables 数组的最后一个元素指向，586 行将 proxy_entry 变量首先初始化为指向这个队列，此后对该队列中所有 ARP 代理表现进行检查，具体检查是在 598 行完成的，除了 IP 地址外，另外对表项所绑定的网络设备以及硬件地址类型同时进行了检查以进行精确匹配。注意对 IP 地址的匹配采用了掩码方式，即以网络地址方式进行匹配，采用网络地址方式可以有效减少 ARP 代理表项的数目，对于 IP 地址方式，则将掩码设置为全 1 即可。如果查找到匹配表项，则 proxy_entry 变量将指向这个匹配的表项，那么下文将根据该表项中硬件地址进行 ARP 应答。当然如果没有查找到匹配表项，则 proxy_entry 变量将为 NULL。以下 602-615 行代码将据此决定如何处理。

```
601             }
602             if (proxy_entry)
603             {
604                 memcpy(ha, proxy_entry->ha, hlen);
605                 sti();
606                 arp_send(ARPOP_REPLY,ETH_P_ARP,sip,dev,tip,sha,ha);
607                 kfree_skb(skb, FREE_READ);
608                 return 0;
609             }
```

602-609 行对应查找到匹配代理表项的情况，此时回复一个 ARP 应答（通过 arp_send 函数完成），注意硬件地址来自于该匹配表项，一般就是代理主机对应网段网络设备的硬件地址。

```
610             else
611             {
612                 sti();
613                 kfree_skb(skb, FREE_READ);
614                 return 0;
615             }
```

610-615 行对应没有查找到匹配表项的情况，此时直接丢弃数据包即可。

```
616         }
```

```
617             else
618             {
619 /*
620  *  To get here, it must be an arp request for us.  We need to reply.
621  */
622                     arp_send(ARPOP_REPLY,ETH_P_ARP,sip,dev,tip,sha,dev->dev_addr);
623             }
624         }
```

617-623 行对应 tip==dev->pa_addr 的情况，即 ARP 请求报文是发送给本机，此时毫无意义，首先回复一个 ARP 应答报文，但函数并未就此返回，从下文执行代码来看，其还完成在 ARP 缓存中添加一个远方主机的 ARP 表项的工作，因为诚如上述，既然远方主机在询问本机硬件地址，则表示远方主机极有可能将要与本机进行通信，那么同时本机主机也需要发送相关数据包给远方主机，为此，本地主机借此在其 ARP 缓存中添加一个远方主机的 ARP 表项，那么之后发送数据包给远方主机时，就可以不用进行对远方主机的 ARP 地址解析过程，从而加快数据包发送速率。我们注意到前文中对于代理 ARP 的情况在回复一个 ARP 应答后直接返回了，而并没有执行下文中代码在 ARP 缓存中添加远方主机的表项，这一点我们可以作如下解释：这要从代理 ARP 的工作环境出发，最常见的工作环境是一个网关代理局域网中所有主机，此种工作环境下，必须有局域网内主机主动向远方主机发起通信过程，因为对于远方主机而言，其根本不知道局域网内主机的存在！既然如此，在远方主机反过来要发送数据包给局域网中主机时，其局域网网关 ARP 缓存中一定存在一个远方主机的 ARP 表项，因为之前在网关转发局域网中主机发送给远方主机数据包时，网关就已经对远方主机进行了 ARP 地址解析过程。当然这种解释只是便于读者理解，实际上，即便以上 608 行不做返回，那么以下的处理也不会造成明显影响，这仅仅是一种实现方式！

以下的工作就是在 ARP 缓存中添加远方主机的对应表项，首先我们检查 ARP 缓存中是否存在一个未完成表项（对应接收到一个 ARP 应答的情况，即本地主动发起 ARP 请求），如果存在，则完成相关字段（即硬件地址字段）的赋值操作。如果不存在，则创建一个新的表项。

```
625 /*
626  * Now all replies are handled.  Next, anything that falls through to here
627  * needs to be added to the arp cache, or have its entry updated if it is
628  * there.
629  */

630         hash = HASH(sip);
631         cli();
632         for(entry=arp_tables[hash];entry;entry=entry->next)
633             if(entry->ip==sip && entry->htype==htype)
634                 break;
```

630-634 行对应查找是否已存在表项的情况。630 行获取表项所在队列索引，此后 632 行遍历队列中表项进行匹配查找，查找关键字是远方主机的 IP 地址。

```
635         if(entry)
636         {
637 /*
638  *      Entry found; update it.
639  */
640                 memcpy(entry->ha, sha, hlen);
641                 entry->hlen = hlen;
642                 entry->last_used = jiffies;
643                 if (!(entry->flags & ATF_COM))
644                 {
645 /*
646  *      This entry was incomplete.    Delete the retransmit timer
647  *      and switch to complete status.
648  */
649                         del_timer(&entry->timer);
650                         entry->flags |= ATF_COM;
651                         sti();
652 /*
653  *      Send out waiting packets. We might have problems, if someone is
654  *      manually removing entries right now -- entry might become invalid
655  *      underneath us.
656  */
657                         arp_send_q(entry, sha);
658                 }
659                 else
660                 {
661                         sti();
662                 }
663         }
```

635-663 行对应 ARP 缓存中存在未完成表项的情况，此时表示是由本地主动发起 ARP 请求，此时接收到一个 ARP 应答，一方面需要进行表项的完成，另一方面需要将由于表项尚未完成从而滞留的数据包发送出去。因为 ARP 请求的主动发起是在发送普通数据包时无从建立链路层首部的情况下进行的，内核在进行 ARP 地址解析的过程中，会将这个数据包暂时缓存在新创建的 ARP 表项（未完成，缺远方主机硬件地址）的相关队列中，等到这个新创建的表项完成所有字段的初始化后（主要是硬件地址字段），方可完成之前滞留数据包的链路层首部的创建继而将其发送出去，具体工作实在 arp_send_q 函数中完成的。640-642 行更新硬件地址字段并更新访问时间字段，643-658 行对应以上所述情况，此时设置表项为完成状态，并将滞留的数据包发送出去。而对于之前已经处于完成状态的表项而言，此时完成的工作仅仅是更新硬件地址和访问时间而已。这种更新是必要的，可以体现远方主机可能的硬件地址变化的情况。另外注意的是，此段代码的执行并非仅是接收到一个 ARP 应答报文，当接收到一个 ARP 请求报文时也可能执行，所以此处的更新硬件地址操作正是为了保证表项内容的及时性和有效性。

```
664         else
665         {
666 /*
667  *  No entry found.    Need to add a new entry to the arp table.
668  */
669             entry = (struct arp_table *)kmalloc(sizeof(struct arp_table),GFP_ATOMIC);
670             if(entry == NULL)
671             {
672                 sti();
673                 printk("ARP: no memory for new arp entry\n");

674                 kfree_skb(skb, FREE_READ);
675                 return 0;
676             }

677             entry->mask = DEF_ARP_NETMASK;
678             entry->ip = sip;
679             entry->hlen = hlen;
680             entry->htype = htype;
681             entry->flags = ATF_COM;
682             init_timer(&entry->timer);
683             memcpy(entry->ha, sha, hlen);
684             entry->last_used = jiffies;
685             entry->dev = skb->dev;
686             skb_queue_head_init(&entry->skb);
687             entry->next = arp_tables[hash];
688             arp_tables[hash] = entry;
689             sti();
690         }
```

664-690 行则对应 ARP 缓存中不存在未完成表项的情况，此时只可能对应 ARP 请求报文的情况，所以此时完成的工作仅仅是新创建一个 ARP 表项并根据 ARP 请求报文内容进行初始化。代码较为简单，留作读者自行理解。

```
691 /*
692  *  Replies have been sent, and entries have been added.    All done.
693  */
694         kfree_skb(skb, FREE_READ);
695         return 0;
696 }
```

最后 694 行对数据包所用内存进行释放操作。至此我们完成对 arp_rcv 函数的分析，该函数是 ARP 协议数据包处理的总入口函数，相对于 ip_rcv 该函数完成的工作较为简单，主要是根据 ARP 报文内容在 ARP 缓存中添加相关表项。之前进行的一系列检查工作是为了保证此

后添加的表项的合法性。

```
697 /*
698  *  Find an arp mapping in the cache. If not found, post a request.
699  */

700 int arp_find(unsigned char *haddr, unsigned long paddr, struct device *dev,
701         unsigned long saddr, struct sk_buff *skb)
702 {
703     struct arp_table *entry;
704     unsigned long hash;
705 #ifdef CONFIG_IP_MULTICAST
706     unsigned long taddr;
707 #endif
```

arp_find 函数根据目的 IP 地址在系统 ARP 缓存中查找匹配的表项从而完成数据帧中链路层首部的创建工作（主要是针对目的主机硬件地址字段）。该函数实现思想比较简单：根据目的 IP 地址（参数 paddr）查找匹配项，如果找到，则从匹配项中复制硬件地址到 haddr 参数指向的缓冲区中，否则创建一个新 ARP 表项，启动 ARP 地址解析过程，将由 skb 参数表示的待发送数据包缓存于新表项的暂存队列中。从上文对 arp_rcv 函数的分析中，我们可以看到，这个滞留的数据包的重新发送是在得到一个 ARP 应答后，调用 arp_send_q 函数完成的。注意 705-707 行表示对于多播的支持在 linux-1.2.13 版本中被作为一个内核选项，只有在编译内核过程中选择了支持多播，内核网络栈实现中才会包括这段代码，下文中同。

```
708     switch (ip_chk_addr(paddr))
709     {
710         case IS_MYADDR:
711             printk("ARP: arp called for own IP address\n");
712             memcpy(haddr, dev->dev_addr, dev->addr_len);
713             skb->arp = 1;
714             return 0;
715 #ifdef CONFIG_IP_MULTICAST
716         case IS_MULTICAST:
717             if(dev->type==ARPHRD_ETHER || dev->type==ARPHRD_IEEE802)
718             {
719                 haddr[0]=0x01;
720                 haddr[1]=0x00;
721                 haddr[2]=0x5e;
722                 taddr=ntohl(paddr);
723                 haddr[5]=taddr&0xff;
724                 taddr=taddr>>8;
725                 haddr[4]=taddr&0xff;
726                 taddr=taddr>>8;
727                 haddr[3]=taddr&0x7f;
```

```
728                     return 0;
729                 }
730             /*
731               *   If a device does not support multicast broadcast the stuff (eg AX.25 for now)
732               */
733 #endif

734             case IS_BROADCAST:
735                 memcpy(haddr, dev->broadcast, dev->addr_len);
736                 skb->arp = 1;
737                 return 0;
738         }
```

708-738 行是对目的 IP 地址的类型进行检查并做相应的处理。710-714 行对应目的 IP 地址是本地地址的情况；715-733 行对应多播地址；734-737 对应广播地址。对于本地地址和广播地址的情况的处理较为简单，直接复制网络设备结构中相应字段值即可。对于多播的情况，则根据 IPv4 多播硬件地址创建规则进行目的硬件地址的创建，创建规则如下：
IP 组播地址的低 23 位映射到 MAC 地址的低 23 位，前面加上 01:00:5E

01 00 5E	0+IP组播地址低23位

所以以上 727 行之所以是 0x7f 的原因即在于最高一位必须是 0，因为只取组播地址的低 23 位而已。

凡是符合以上三种情况，即完成目的硬件地址的查询工作，此时都直接返回，无须进行以下的 ARP 缓存的查询，只有目的 IP 地址（paddr 参数）表示的是远方主机的单播 IP 地址时，才进行 ARP 缓存的查询和查询失败时需进行的 ARP 地址解析工作。

```
739         hash = HASH(paddr);
740         cli();

741         /*
742           *   Find an entry
743           */
744         entry = arp_lookup(paddr, PROXY_NONE);
```

从 739-744 行代码来看，具体的查询工作被封装在 arp_lookup 函数中，该函数使用参数为：查询关键字以及匹配类型，具体情况在下文中分析该函数实现时再进行详细说明，arp_lookup 函数返回匹配 ARP 表项，如果没有匹配表项，则返回 NULL。可以想见，下文中将根据该参数决定是否需要发起 ARP 地址解析过程。

```
745         if (entry != NULL)      /* It exists */
746         {
747                 if (!(entry->flags & ATF_COM))
748                 {
```

```
749                 /*
750                  *  A request was already send, but no reply yet. Thus
751                  *  queue the packet with the previous attempt
752                  */

753                 if (skb != NULL)
754                 {
755                         skb_queue_tail(&entry->skb, skb);
756                         skb_device_unlock(skb);
757                 }
758                 sti();
759                 return 1;
760         }

761         /*
762          *  Update the record
763          */

764         entry->last_used = jiffies;
765         memcpy(haddr, entry->ha, dev->addr_len);
766         if (skb)
767                 skb->arp = 1;
768         sti();
769         return 0;
770 }
```

745-770 行对应查询到匹配表项情况，此时需要进一步对表项的完成状态进行检查，如果表项处于尚未完成状态（对应 747-760 行代码），则表示 ARP 地址解析过程正在进行，此时直接将待发送数据包直接缓存到该表项暂存队列中即返回；否则就表示这是一个可用 ARP 表项，此时将根据表项中内容返回硬件地址，并设置数据包 skb 中 arp 字段值为 1，表示数据帧完全完成创建工作。链路层模块发送函数（dev_queue_xmit）会检查该字段值，如果该字段值为 0，则表示数据帧链路层首部没有完成创建，此时会调用相关函数进行处理，最终会进入 ARP 模块，所以此处对于该字段的设置至关重要，否则有可能会造成循环调用，造成系统崩溃。注意对应查询到匹配项的情况，无论匹配项是否完成，该函数在进行相关处理后，都直接返回，不会发送 ARP 请求报文，因为没有必要。

```
771 /*
772  *  Create a new unresolved entry.
773  */

774 entry = (struct arp_table *) kmalloc(sizeof(struct arp_table),
775                                 GFP_ATOMIC);
776 if (entry != NULL)
777 {
```

```
778             entry->mask = DEF_ARP_NETMASK;
779             entry->ip = paddr;
780             entry->hlen = dev->addr_len;
781             entry->htype = dev->type;
782             entry->flags = 0;
783             memset(entry->ha, 0, dev->addr_len);
784             entry->dev = dev;
785             entry->last_used = jiffies;
786             init_timer(&entry->timer);
787             entry->timer.function = arp_expire_request;
788             entry->timer.data = (unsigned long)entry;
789             entry->timer.expires = ARP_RES_TIME;
790             entry->next = arp_tables[hash];
791             arp_tables[hash] = entry;
792             add_timer(&entry->timer);
793             entry->retries = ARP_MAX_TRIES;
794             skb_queue_head_init(&entry->skb);
795             if (skb != NULL)
796             {
797                 skb_queue_tail(&entry->skb, skb);
798                 skb_device_unlock(skb);
799             }
800         }
801         else
802         {
803             if (skb != NULL && skb->free)
804                 kfree_skb(skb, FREE_WRITE);
805         }
```

774-800 行对应不存在匹配 ARP 表项的情况，此时创建一个新的表项，将带发送数据包缓存到新表项暂存队列中，并发起 ARP 地址解析过程。774-794 行即进行创建和初始化工作。读者可结合前文中 arp_table 结构的定义，查看此处的初始化代码，从而对各字段的含义进一步的理解。795-799 行完成待发送数据包的缓存工作。注意以上代码执行完后，并为返回，因为虽然创建了一个新表项，但该表项中最重要的一个字段即硬件地址字段尚未初始化，这个初始化工作需要依赖 ARP 地址解析，所以下面进行的工作就是发送一个 ARP 请求报文。

```
806         sti();
807         /*
808          *  If we didn't find an entry, we will try to send an ARP packet.
809          */
810         arp_send(ARPOP_REQUEST, ETH_P_ARP, paddr, dev, saddr, NULL,
811                  dev->dev_addr);

812         return 1;
```

```
813 }
```

810 行即调用 arp_send 函数完成 ARP 请求报文的发送从而启动 ARP 地址解析过程。该新表项的最终完成在 arp_rcv 函数中对 ARP 应答数据包进行处理时完成，并同时将表项中暂存的滞留数据包发送出去。

```
814 /*
815  *    Write the contents of the ARP cache to a PROCfs file.
816  */

817 #define HBUFFERLEN 30

818 int arp_get_info(char *buffer, char **start, off_t offset, int length)
819 {
820         int len=0;
821         off_t begin=0;
822         off_t pos=0;
823         int size;
824         struct arp_table *entry;
825         char hbuffer[HBUFFERLEN];
826         int i,j,k;
827         const char hexbuf[] =    "0123456789ABCDEF";

828         size = sprintf(buffer,"IP address       HW type     Flags       HW address
Mask\n");

829         pos+=size;
830         len+=size;

831         cli();
832         for(i=0; i<FULL_ARP_TABLE_SIZE; i++)
833         {
834                 for(entry=arp_tables[i]; entry!=NULL; entry=entry->next)
835                 {
836 /*
837  *    Convert hardware address to XX:XX:XX:XX ... form.
838  */
839 #ifdef CONFIG_AX25

840                         if(entry->htype==ARPHRD_AX25)
841                                 strcpy(hbuffer,ax2asc((ax25_address *)entry->ha));
842                         else {
843 #endif
```

```
844                 for(k=0,j=0;k<HBUFFERLEN-3 && j<entry->hlen;j++)
845                 {
846                     hbuffer[k++]=hexbuf[ (entry->ha[j]>>4)&15 ];
847                     hbuffer[k++]=hexbuf[    entry->ha[j]&15        ];
848                     hbuffer[k++]=':';
849                 }
850                 hbuffer[--k]=0;

851 #ifdef CONFIG_AX25
852                 }
853 #endif
854                 size = sprintf(buffer+len,
855                     "%-17s0x%-10x0x%-10x%s",
856                     in_ntoa(entry->ip),
857                     (unsigned int)entry->htype,
858                     entry->flags,
859                     hbuffer);
860                 size += sprintf(buffer+len+size,
861                       "         %-17s\n",
862                         entry->mask==DEF_ARP_NETMASK?
863                           "*":in_ntoa(entry->mask));

864                 len+=size;
865                 pos=begin+len;

866                 if(pos<offset)
867                 {
868                     len=0;
869                     begin=pos;
870                 }
871                 if(pos>offset+length)
872                     break;
873             }
874         }
875         sti();

876         *start=buffer+(offset-begin);  /* Start of wanted data */
877         len-=(offset-begin);        /* Start slop */
878         if(len>length)
879             len=length;                     /* Ending slop */
880         return len;
881 }
```

arp_get_info 函数被上层调用查看 ARP 缓存内容，该函数实现方式本书前文中讨论其他协议

实现文件时已有分析，此处不再对该函数进行分析，读者结合 arp_table 结构可自行理解。

刚刚在分析 arp_find 函数时，其调用 arp_lookup 函数完成具体的 ARP 匹配表项查询工作，下面我们即对 arp_lookup 函数进行分析。

```
882  /*
883   *   This will find an entry in the ARP table by looking at the IP address.
884   *         If proxy is PROXY_EXACT then only exact IP matches will be allowed
885   *         for proxy entries, otherwise the netmask will be used
886   */

887  static struct arp_table *arp_lookup(unsigned long paddr, enum proxy proxy)
888  {
```

参数 paddr 表示目的 IP 地址，此为查询关键字。enum proxy 枚举类型定义在 116 行，为便于分析，重新给出其定义如下：

```
116  enum proxy {
117       PROXY_EXACT=0,
118       PROXY_ANY,
119       PROXY_NONE,
120  };
```

这个枚举类型用于定制 arp_lookup 函数的查询方式，主要与代理 ARP 表项相联系，其中 PROXY_NONE 表示不进行代理 ARP 表项的查询，即在查询过程中排出代理 ARP 表项。其他两个 enum 类型：PROXY_EXACT，PROXY_ANY 用于定制在进行代理 ARP 表项查询时，采用何种匹配方式，其中 PROXY_EXACT 表示采用精确匹配方式，即被代理主机 IP 地址必须与作为参数传入的 IP 地址精确匹配，所谓精确匹配，即全 32 位匹配，这是相对于网络匹配方式而言，当 proxy 参数为 PROXY_ANY 时，即表示采用网络匹配方式，此时只要被查询 IP 地址与代理 ARP 表项中网络部分地址相同即可（在比较过程中，主机地址部分被掩码屏蔽掉了）。arp_find 函数中对 arp_lookup 函数进行调用时，使用的是 PROXY_NONE，即不对代理 ARP 表项进行查询。

```
889       struct arp_table *entry;
890       unsigned long hash = HASH(paddr);

891       for (entry = arp_tables[hash]; entry != NULL; entry = entry->next)
892            if (entry->ip == paddr) break;
```

890-892 行是对普通 ARP 表项的匹配查询，从 892 行来看，匹配采用 IP 地址精确匹配方式，如果查询到匹配项，则 entry 变量将指向匹配表项，否则 entry 变量值为 NULL。该变量值将在下面代码中被检查，从而决定是否有必要（以及在必要的前提下，是否有资格）查询代理 ARP 表项。

```
893       /* it's possibly a proxy entry (with a netmask) */
894       if (!entry && proxy != PROXY_NONE)
895       for (entry=arp_tables[PROXY_HASH]; entry != NULL; entry = entry->next)
```

```
896             if ((proxy==PROXY_EXACT) ? (entry->ip==paddr)
897                                              : !((entry->ip^paddr)&entry->mask))
898                     break;
```

894-898 行代码对代理 ARP 表项进行匹配查询。首先 894 行判断是否在普通 ARP 表项中查询到匹配项，如果没有，则查看 proxy 参数，据其决定是否进行代理 ARP 表项的查询。如果 entry 为 NULL，且 proxy 参数不为 PROXY_NONE，则表示需要进一步查询代理 ARP 表项。在对代理 ARP 表项查询时，根据 proxy 参数是否为 PROXY_EXACT，从而决定是否采用 IP 地址精确匹配方式。这一点在 896-897 行代码中体现。

```
899         return entry;
900 }
```

在完成对代理 ARP 表项的查询后，直接返回 entry 变量所指向的表项，如果在代理 ARP 表项仍未查询得到匹配项，则 entry 值为 NULL，此时返回 NULL，表示查询失败。

```
901 /*
902  *  Set (create) an ARP cache entry.
903  */

904 static int arp_req_set(struct arpreq *req)
905 {
906         struct arpreq r;
907         struct arp_table *entry;
908         struct sockaddr_in *si;
909         int htype, hlen;
910         unsigned long ip;
911         struct rtable *rt;
```

arp_req_set 函数用于更新或者创建一个新的 ARP 表项。该函数是用户对 ARP 缓存进行配置的底层实现函数。arpreq 结构定义在 include/linux/if_arp.h 中，为便于读者理解，重新给出该结构定义如下：

```
/*include/linux/if_arp.h*/
47  /* ARP ioctl request. */
48  struct arpreq {
49      struct sockaddr   arp_pa;       /* protocol address  IP 地址*/
50      struct sockaddr   arp_ha;       /* hardware address  硬件地址*/
51      int               arp_flags;    /* flags    */
52      struct sockaddr           arp_netmask;    /* netmask (only for proxy arps)    掩码*/
53  };
```

下面我们继续对 arp_req_set 函数的分析：

```
912         memcpy_fromfs(&r, req, sizeof(r));
913         /* We only understand about IP addresses... */
```

```
914         if (r.arp_pa.sa_family != AF_INET)
915                 return -EPFNOSUPPORT;
```

912 行将参数值从用户缓冲区复制到内核缓冲区中，914 行检查 IP 地址类别，当前只支持 IP 地址类型（AF_INET 域）。

```
916         /*
917          * Find out about the hardware type.
918          * We have to be compatible with BSD UNIX, so we have to
919          * assume that a "not set" value (i.e. 0) means Ethernet.
920          */

921         switch (r.arp_ha.sa_family) {
922                 case ARPHRD_ETHER:
923                         htype = ARPHRD_ETHER;
924                         hlen = ETH_ALEN;
925                         break;

926                 case ARPHRD_ARCNET:
927                         htype = ARPHRD_ARCNET;
928                         hlen = 1;  /* length of arcnet addresses */
929                         break;

930 #ifdef CONFIG_AX25
931                 case ARPHRD_AX25:
932                         htype = ARPHRD_AX25;
933                         hlen = 7;
934                         break;
935 #endif
936                 default:
937                         return -EPFNOSUPPORT;
938         }
```

921-938 继续对参数中硬件地址类型进行检查，并对相关变量进行初始化，这些变量在下文中创建 ARP 表项使用。变量 htype 表示硬件地址类型，hlen 表示硬件地址的长度。如对于以太网而言，hlen 等于 6 字节。其他网络类型如 AX.25 此处不做分析，感兴趣读者可自行搜索相关内容。

```
939         si = (struct sockaddr_in *) &r.arp_pa;
940         ip = si->sin_addr.s_addr;
941         if (ip == 0)
942         {
943                 printk("ARP: SETARP: requested PA is 0.0.0.0 !\n");
944                 return -EINVAL;
```

```
945     }
```

939-945 行代码获取 IP 地址，并对 IP 地址合法性进行检查，即 IP 地址不可谓全 0。

```
946     /*
947      *  Is it reachable directly ?
948      */

949     rt = ip_rt_route(ip, NULL, NULL);
950     if (rt == NULL)
951         return -ENETUNREACH;
```

949-951 行体现的思想是 ARP 表项中所有表项代表的主机都在同一个冲突域中，即这些主机都连接在相同的链路上。通常我们将每个网络设备在出厂之时，都被赋予了唯一的硬件地址，这很容易被误解为硬件地址必须具有全局唯一性，其实不然，硬件地址只需具有链路唯一性，即在同一个链路上不可具有相同的硬件地址，如果处于不同的链路，则无需关注硬件地址的唯一性，这在某些嵌入式系统开发中尤其明显，即硬件地址可由用户任意配置，只要满足链路唯一性即可，无需全球唯一性。所以进行 IP 地址到硬件地址映射功能的 ARP 协议具有链路局限性，即 ARP 数据包不可跨越路由器，也即 ARP 缓存中所有表项表示的主机必须是可直达的。对于动态配置的 ARP 表项，这一点由内核代码保证，但是对于用户配置的表项，则必须进行直达性检查，否则将会导致一个无用表项，949-951 行即完成可直达性检测。

```
952     /*
953      *  Is there an existing entry for this address?
954      */

955     cli();

956     /*
957      *  Find the entry
958      */
959     entry = arp_lookup(ip, PROXY_EXACT);
960     if (entry && (entry->flags & ATF_PUBL) != (r.arp_flags & ATF_PUBL))
961     {
962         sti();
963         arp_destroy(ip,1);
964         cli();
965         entry = NULL;
966     }
```

959-966 行对前文中需要加入的新表项 IP 地址为关键字对已有表项进行查询，从而确定有无重复表项。如果 entry 返回值为 NULL，则表示没有重复表项，下面即可进行新表项的创建工作，否则检查原有表项与待加入表项之间的关系。ATF_PUBL 标志位表示对应表项是一个代理 ARP 表项。如果原有表项和待加入表项的 ATF_PUBL 标志位设置不相同，则表示不

可进行硬件地址的简单更新，此处的处理方式是直接删除原有表项，已待加入表项为准。如果 ATF_PUBL 标志位相同，从下文中可看出，则进行硬件地址的简单更新以及访问时间的更新操作。

如果 959 行返回 NULL，或者原有表项被删除，则都需要重新创建一个新的 arp_table 结构表示这个新表项。下面代码即完成这个工作。

```
967     /*
968      *  Do we need to create a new entry
969      */

970     if (entry == NULL)
971     {
972         unsigned long hash = HASH(ip);
973         if (r.arp_flags & ATF_PUBL)
974             hash = PROXY_HASH;

975         entry = (struct arp_table *) kmalloc(sizeof(struct arp_table),
976                                 GFP_ATOMIC);
977         if (entry == NULL)
978         {
979             sti();
980             return -ENOMEM;
981         }
982         entry->ip = ip;
983         entry->hlen = hlen;
984         entry->htype = htype;
985         init_timer(&entry->timer);
986         entry->next = arp_tables[hash];
987         arp_tables[hash] = entry;
988         skb_queue_head_init(&entry->skb);
989     }
```

970-989 行代码对应新表项 arp_table 结构的分配和初始化工作。注意 973 行代码对 ATF_PUBL 标志位的检查，如果该标志位被设置，则 hash 变量被初始化为 PROXY_HASH，即指向代理 ARP 表项队列。986-987 行即完成新表项的插入工作。这段代码对 arp_table 结构的初始化中缺了对硬件地址字段和掩码字段的初始化操作，这两个字段的赋值或者说更新在下文代码中进行，从而顾及到 959 行返回非 NULL 的情况。

```
990     /*
991      *  We now have a pointer to an ARP entry.  Update it!
992      */
993     memcpy(&entry->ha, &r.arp_ha.sa_data, hlen);
994     entry->last_used = jiffies;
995     entry->flags = r.arp_flags | ATF_COM;
996     if ((entry->flags & ATF_PUBL) && (entry->flags & ATF_NETMASK))
```

```
997         {
998             si = (struct sockaddr_in *) &r.arp_netmask;
999             entry->mask = si->sin_addr.s_addr;
1000        }
1001            else
1002                entry->mask = DEF_ARP_NETMASK;
1003            entry->dev = rt->rt_dev;
1004            sti();

1005            return 0;
1006    }
```

993-1003 行代码完成 arp_table 结构中其他字段的初始化或者更新操作。需要注意的是 996 行代码对 ATF_PUBL 标志位的检查。另外该行同时对 ATF_NETMASK 标志位也进行了检查，ATF_NETMASK 标志位表示代理 ARP 表项中 IP 地址字段是一个网络地址，对于网络地址，我们需要网络掩码，999，1002 行是根据不同情况对网络掩码字段进行初始化工作。DEF_ARP_NETMASK 常量在 131 行初始化为全 1，即此代理 ARP 表项仅仅是代理某台特定主机。

至此我们完成对 arp_req_set 函数的分析，该函数被 arp_ioctl 函数调用对用户配置 ARP 缓存命令进行具体处理。下面介绍的 arp_req_get 函数，其作用与 arp_req_set 函数对称，即返回用户指定的 ARP 表项内容，arp_req_get 函数也是在 arp_ioctl 函数被调用。

```
1007    /*
1008     *  Get an ARP cache entry.
1009     */

1010    static int arp_req_get(struct arpreq *req)
1011{
1012            struct arpreq r;
1013            struct arp_table *entry;
1014            struct sockaddr_in *si;

1015            /*
1016             *  We only understand about IP addresses...
1017             */

1018            memcpy_fromfs(&r, req, sizeof(r));

1019            if (r.arp_pa.sa_family != AF_INET)
1020                    return -EPFNOSUPPORT;

1021            /*
1022             *  Is there an existing entry for this address?
1023             */
```

```
1024            si = (struct sockaddr_in *) &r.arp_pa;
1025            cli();
1026            entry = arp_lookup(si->sin_addr.s_addr,PROXY_ANY);

1027            if (entry == NULL)
1028            {
1029                    sti();
1030                    return -ENXIO;
1031            }

1032            /*
1033             *      We found it; copy into structure.
1034             */

1035            memcpy(r.arp_ha.sa_data, &entry->ha, entry->hlen);
1036            r.arp_ha.sa_family = entry->htype;
1037            r.arp_flags = entry->flags;
1038            sti();

1039            /*
1040             *      Copy the information back
1041             */

1042            memcpy_tofs(req, &r, sizeof(r));
1043            return 0;
1044    }
```

arp_req_get 函数实现较为简单，此处不再对其进行分析，留作读者自行理解。

```
1045    /*
1046     *      Handle an ARP layer I/O control request.
1047     */

1048    int arp_ioctl(unsigned int cmd, void *arg)
1049    {
1050            struct arpreq r;
1051            struct sockaddr_in *si;
1052            int err;

1053            switch(cmd)
1054            {
1055                    case SIOCDARP:
1056                            if (!suser())
1057                                    return -EPERM;
```

```
1058                    err = verify_area(VERIFY_READ, arg, sizeof(struct arpreq));
1059                    if(err)
1060                        return err;
1061                    memcpy_fromfs(&r, arg, sizeof(r));
1062                    if (r.arp_pa.sa_family != AF_INET)
1063                        return -EPFNOSUPPORT;
1064                    si = (struct sockaddr_in *) &r.arp_pa;
1065                    arp_destroy(si->sin_addr.s_addr, 1);
1066                    return 0;
1067                case SIOCGARP:
1068                    err = verify_area(VERIFY_WRITE, arg, sizeof(struct arpreq));
1069                    if(err)
1070                        return err;
1071                    return arp_req_get((struct arpreq *)arg);
1072                case SIOCSARP:
1073                    if (!suser())
1074                        return -EPERM;
1075                    err = verify_area(VERIFY_READ, arg, sizeof(struct arpreq));
1076                    if(err)
1077                        return err;
1078                    return arp_req_set((struct arpreq *)arg);
1079                default:
1080                    return -EINVAL;
1081            }
1082            /*NOTREACHED*/
1083            return 0;
1084    }
```

arp_ioctl 函数响应上层用户对 ARP 缓存的配置，根据不同的配置选项调用具体的实现函数进行处理。该函数目前支持三个选项：
SIOCDARP：删除一个 ARP 表项，处理函数为 arp_destroy；
SIOCGARP：获取一个 ARP 表项内容，处理函数为 arp_req_get；
SIOCSARP：更新或者新建一个 ARP 表项，处理函数为 arp_req_set。
以上删除或者更新一个表项时，需要超级用户权限。具体 arp_ioctl 函数实现代理较为简单浅显，此处不再赘述。

arp.c 文件最后向链路层模块注册 ARP 协议，这一点我们在前文中多有论述，下面看具体代码。

```
1085    /*
1086     *  Called once on startup.
1087     */

1088    static struct packet_type arp_packet_type =
1089    {
```

```
1090            0, /* Should be: __constant_htons(ETH_P_ARP) - but this _doesn't_ come out
constant! */
1091            NULL,           /* All devices */
1092            arp_rcv,
1093            NULL,
1094            NULL
1095    };
```

首先定义了 packet_type 结构，此处只对结构中 func 字段进行了初始化，赋值为 arp_rcv，即 ARP 协议数据包总入口函数，与 ip_rcv 函数的作用类似。其他字段的初始化以及向链路层模块的注册在下面 arp_init 函数中完成。

```
1096    static struct notifier_block arp_dev_notifier={
1097            arp_device_event,
1098            NULL,
1099            0
1100};
```

因为 ARP 表项如同路由表项也和相关网络设备绑定，所以此处定义了网络设备事件接收接口，对网络设备启动或者停止工作事件进行接收。事件接收函数的注册通过 notifier_block 结构以及 register_netdevice_notifier 注册函数完成。

```
1101void arp_init (void)
1102{
1103    /* Register the packet type */
1104    arp_packet_type.type=htons(ETH_P_ARP);
1105    dev_add_pack(&arp_packet_type);
1106    /* Start with the regular checks for expired arp entries. */
1107    add_timer(&arp_timer);
1108    /* Register for device down reports */
1109    register_netdevice_notifier(&arp_dev_notifier);
1110}
```

arp_init 函数主要完成两个功能：向链路层注册 ARP 协议总入口函数；以及向内核注册网络设备事件处理函数。各自通过 dev_add_pack, register_netdevice_notifier 函数完成。注意 1104 行正确设置了协议类型号。另外 1107 行启动了一个 ARP 缓存表项定期刷新定时器，用于清除超过生存期的 ARP 表项，从而保证 ARP 表项内容的有效性。arp_timer 变量在前文中 124 行定义：

```
124  static struct timer_list arp_timer =
125     { NULL, NULL, ARP_CHECK_INTERVAL, 0L, &arp_check_expire };
```

定期检查函数为 arp_check_expire，该函数前文中已有论述，此处不再赘述。

arp.c 文件小结

到此，我们完成 ARP 协议实现文件 arp.c 的分析。我们得到如下启示：

1>ARP 缓存简单的说就是一个数组，每个数组元素指向一个队列，队列中每个元素表示一个 ARP 表项。
2>每个 ARP 表项由一个 arp_table 结构表示，该结构中含有 IP 地址，硬件地址映射关系，表项状态标志以及其他相关辅助字段（如数据包暂存队列）。
3>对 ARP 缓存的操作，具体的是对每个 arp_table 结构的操作。
4>ARP 缓存即如此简单，不存在任何神秘之处。

2.37 net/inet/arp.h 头文件
为保证 ARP 协议实现文件的完整性，我们列出 arp.h 头文件如下，该文件只是对 arp.c 文件中定义函数的声明，所以只列出如下，不再说明。

```
1     /* linux/net/inet/arp.h */
2     #ifndef _ARP_H
3     #define _ARP_H

4     extern void      arp_init(void);
5     extern void      arp_destroy(unsigned long paddr, int force);
6     extern void      arp_device_down(struct device *dev);
7     extern int arp_rcv(struct sk_buff *skb, struct device *dev,
8                        struct packet_type *pt);
9     extern int arp_find(unsigned char *haddr, unsigned long paddr,
10                struct device *dev, unsigned long saddr, struct sk_buff *skb);
11    extern int arp_get_info(char *buffer, char **start, off_t origin, int length);
12    extern int arp_ioctl(unsigned int cmd, void *arg);
13    extern void          arp_send(int type, int ptype, unsigned long dest_ip,
14                         struct device *dev, unsigned long src_ip,
15                         unsigned char *dest_hw, unsigned char *src_hw);

16    #endif     /* _ARP_H */
```

由于 RARP 协议适用较少（一般只在主机启动时使用），且其实现代码非常类似于 ARP 协议，故本书不对 RARP 协议实现文件 rarp.c, rarp.h 进行分析。感兴趣读者可结合以上对 ARP 协议的实现代码自行对其进行分析。

2.38 net/inet/devinit.c 文件
devinit.c 文件定义有四个功能函数，如下：
ip_get_mask：根据 IP 地址获取其对应掩码，获取方式根据 IP 地址 A,B,C,D 等类别分类方式。
ip_chk_addr：判定作为参数传入的 IP 地址的类型：单播？组播？广播？
ip_my_addr：获取本地 IP 地址。
ip_dev_check：根据 IP 地址获取该地址对应的网络设备。

下面我们对以上四个函数进行具体分析。

```
1     /*
2      *    NET3      IP device support routines.
```

```
3     *
4     *         This program is free software; you can redistribute it and/or
5     *         modify it under the terms of the GNU General Public License
6     *         as published by the Free Software Foundation; either version
7     *         2 of the License, or (at your option) any later version.
8     *
9     *    Derived from the IP parts of dev.c 1.0.19
10    *         Authors:  Ross Biro, <bir7@leland.Stanford.Edu>
11    *                   Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
12    *                   Mark Evans, <evansmp@uhura.aston.ac.uk>
13    *
14    *    Additional Authors:
15    *         Alan Cox, <gw4pts@gw4pts.ampr.org>
16    */

17  #include <asm/segment.h>
18  #include <asm/system.h>
19  #include <asm/bitops.h>
20  #include <linux/types.h>
21  #include <linux/kernel.h>
22  #include <linux/sched.h>
23  #include <linux/string.h>
24  #include <linux/mm.h>
25  #include <linux/socket.h>
26  #include <linux/sockios.h>
27  #include <linux/in.h>
28  #include <linux/errno.h>
29  #include <linux/interrupt.h>
30  #include <linux/if_ether.h>
31  #include <linux/inet.h>
32  #include <linux/netdevice.h>
33  #include <linux/etherdevice.h>
34  #include "ip.h"
35  #include "route.h"
36  #include "protocol.h"
37  #include "tcp.h"
38  #include <linux/skbuff.h>
39  #include "sock.h"
40  #include "arp.h"

41  /*
42   *    Determine a default network mask, based on the IP address.
43   */
```

```
44  unsigned long ip_get_mask(unsigned long addr)
45  {
46          unsigned long dst;

47          if (addr == 0L)
48                  return(0L);     /* special case */

49          dst = ntohl(addr);
50          if (IN_CLASSA(dst))
51                  return(htonl(IN_CLASSA_NET));
52          if (IN_CLASSB(dst))
53                  return(htonl(IN_CLASSB_NET));
54          if (IN_CLASSC(dst))
55                  return(htonl(IN_CLASSC_NET));

56          /*
57           *      Something else, probably a multicast.
58           */

59          return(0);
60  }
```

ip_get_mask 函数实现非常简单，由于没有其他条件，所以根据 IP 地址判断其掩码只能根据 IP 地址大的分类方式，50-55 行代码中使用的相关宏均定义在 include/linux/in.h 头文件中，结合该文件，ip_get_mask 函数较为容易理解。需要注意的是 47-48 两行对 IP 地址为 0 的特殊处理，此时返回的掩码也为全 0，这种处理方式用于默认网关地址的掩码使用上。

```
61  /*
62   *      Check the address for our address, broadcasts, etc.
63   *
64   *      I intend to fix this to at the very least cache the last
65   *      resolved entry.
66   */

67  int ip_chk_addr(unsigned long addr)
68  {
69          struct device *dev;
70          unsigned long mask;
```

ip_chk_addr 函数检查作为参数传入的 IP 地址的类型，该函数返回值如下：
IS_BROADCAST：参数表示的 IP 地址是一个广播地址。
IS_MYADDR：是一个本地主机 IP 地址。
IS_MULTICAST：是一个多播 IP 地址。
以上常量都定义在 include/linux/netdevice.h 头文件中。为便于下文分析，我们重新给出其他

相关常量定义如下：

/*include/linux/in.h*/

```
79  /* Address to accept any incoming messages. */
80  #define   INADDR_ANY          ((unsigned long int) 0x00000000)

81  /* Address to send to all hosts. */
82  #define   INADDR_BROADCAST    ((unsigned long int) 0xffffffff)
```

我们继续对 devinit.c 文件的分析：

```
71      /*
72        *   Accept both `all ones' and `all zeros' as BROADCAST.
73        *   (Support old BSD in other words). This old BSD
74        *   support will go very soon as it messes other things
75        *   up.
76        *   Also accept `loopback broadcast' as BROADCAST.
77        */

78      if (addr == INADDR_ANY || addr == INADDR_BROADCAST ||
79          addr == htonl(0x7FFFFFFFL))
80          return IS_BROADCAST;
```

78-80 行表示对于 0.0.0.0, 255.255.255.255, 127.255.255.255 均当作是广播地址对待。

```
81      mask = ip_get_mask(addr);
82      /*
83        *   Accept all of the `loopback' class A net.
84        */

85      if ((addr & mask) == htonl(0x7F000000L))
86          return IS_MYADDR;
```

81-86 行代码检查 IP 地址是否为一个回环地址，即 IP 地址形式为 127.X.X.X 的所有地址均当作为回环地址对待，此时返回 IS_MYADDR 表示这是一个本地 IP 地址。

经过对以上两种特殊情况的处理，下面我们需要具体的检查每个网络设备所配置的 IP 地址，检查传入参数表示的 IP 地址类型。

```
87      /*
88        *   OK, now check the interface addresses.
89        */

90      for (dev = dev_base; dev != NULL; dev = dev->next)
91      {
92          if (!(dev->flags & IFF_UP))
93              continue;
```

dev_base 是一个全局变量，所以网络设备对应的 device 结构都在由 dev_base 变量指向的队列中，维护这样一个队列便于对系统中当前注册的所有网络设备进行查询以及相关操作。dev_base 变量定义在 drivers/net/Space.c 文件中。92 行表示跳过对未工作网络设备的检查，因为只有对于一个处于工作状态的网络设备，IP 地址才有意义。

```
94          /*
95           *  If the protocol address of the device is 0 this is special
96           *  and means we are address hunting (eg bootp).
97           */

98          if ((dev->pa_addr == 0)/* || (dev->flags&IFF_PROMISC)*/)
99              return IS_MYADDR;
```

如果一个网络设备协议地址字段 pa_addr 被设置为 0，则表示该设备正处于启动阶段，尚未配置 IP 地址，此时尚且无法对传入的 IP 地址进行判定，此处进行了一种极端假设，即此时假设传入的 IP 地址就是一个本地 IP 地址。

```
100         /*
101          *  Is it the exact IP address?
102          */

103         if (addr == dev->pa_addr)
104             return IS_MYADDR;
105         /*
106          *  Is it our broadcast address?
107          */

108         if ((dev->flags & IFF_BROADCAST) && addr == dev->pa_brdaddr)
109             return IS_BROADCAST;
```

103-109 行代码的判断较为简单，比较网络设备的相关字段值，返回对应地址类型。

```
110         /*
111          *  Nope. Check for a subnetwork broadcast.
112          */

113         if (((addr ^ dev->pa_addr) & dev->pa_mask) == 0)
114         {
115             if ((addr & ~dev->pa_mask) == 0)
116                 return IS_BROADCAST;
117             if ((addr & ~dev->pa_mask) == ~dev->pa_mask)
118                 return IS_BROADCAST;
119         }
```

113-119 行对网络地址进行了检查，如果参数表示的 IP 地址与网络设备 IP 地址网络部分相同，同时主机部分为全 0 或者为全 1，则作为该子网段的广播地址对待。

```
120             /*
121               *    Nope. Check for Network broadcast.
122               */

123             if (((addr ^ dev->pa_addr) & mask) == 0)
124             {
125                     if ((addr & ~mask) == 0)
126                             return IS_BROADCAST;
127                     if ((addr & ~mask) == ~mask)
128                             return IS_BROADCAST;
129             }
130     }
```

123-129 行代码基本与 113-119 行代码相同，不同之处在于 123 此时使用的是 mask 变量表示的掩码，该变量初始化为 ip_get_mask 函数的返回值，而 ip_get_mask 函数是根据大的 IP 地址分类方式判断得到的网络掩码，而 113 中使用的网络设备配置的子网掩码，一般而言，该配置的子网掩码是根据具体的子网段所使用网络地址类型得到的，所以称为子网掩码，而 ip_get_mask 函数返回的是网络掩码，所以 123-129 行代码是在大的 IP 地址分类之下对网络部分地址的再次检查，而 113-119 行可以说是对子网网络地址部分的检查。无论哪种方式，最后判断的依据都是同样的，即网络部分必须相同，主机部分为全 0 或全 1 时，都表示一个广播地址。

```
131     if(IN_MULTICAST(ntohl(addr)))
132             return IS_MULTICAST;
133     return 0;       /* no match at all */
134 }
```

函数最后对组播地址类型进行了检查，IN_MULTICAST 定义如下：

```
/*include/linux/in.h*/
74  #define   IN_CLASSD(a)          ((((long int) (a)) & 0xf0000000) == 0xe0000000)
75  #define   IN_MULTICAST(a)           IN_CLASSD(a)
```

即如果传入的 IP 地址是一个 D 类地址，则表示这是一个多播 IP 地址。如果找不到匹配项，那么返回 0。表示无法确知传入的 IP 地址类型，一般这是不可能的，133 行更多的作用是避免出现警告信息。

```
135 /*
136  *  Retrieve our own address.
137  *
138  *  Because the loopback address (127.0.0.1) is already recognized
139  *  automatically, we can use the loopback interface's address as
```

```
140   *    our "primary" interface.    This is the address used by IP et
141   *    al when it doesn't know which address to use (i.e. it does not
142   *    yet know from or to which interface to go...).
143   */

144  unsigned long ip_my_addr(void)
145  {
146          struct device *dev;

147          for (dev = dev_base; dev != NULL; dev = dev->next)
148          {
149                  if (dev->flags & IFF_LOOPBACK)
150                          return(dev->pa_addr);
151          }
152          return(0);
153  }
```

ip_my_addr 函数返回本地地址，从 149 行可以看出，该函数返回设置 IFF_LOOPBACK 标志位的网络设备的 IP 地址，这也称为一个回环设备，回环设备只存在于软件中，没有对应的物理设备与之对应，回环设备并不仅仅在于回送本地发送的数据包，许多内核组件的工作方式都依赖于该回环设备提供的功能。发送到回环设备的数据包在网络栈内部就被送回去了，数据根本不会出现在物理传输介质上。一般回环设备 IP 地址配置为 127.0.0.1，这是最为典型的。根据 138-142 行的注释，此处使用该地址的原因是：因为 127.0.0.1 地址已经可以被自动的识别，我们通常将回环接口作为我们的“优先考虑”接口。通常在不知道数据包从何处接收或者从何设备发送出去时，我们都优先考虑回环设备（地址）。

```
154  /*
155   *    Find an interface that can handle addresses for a certain address.
156   *
157   *    This needs optimising, since it's relatively trivial to collapse
158   *    the two loops into one.
159   */

160  struct device * ip_dev_check(unsigned long addr)
161  {
162          struct device *dev;

163          for (dev = dev_base; dev; dev = dev->next)
164          {
165                  if (!(dev->flags & IFF_UP))
166                          continue;
167                  if (!(dev->flags & IFF_POINTOPOINT))
168                          continue;
169                  if (addr != dev->pa_dstaddr)
```

```
170                 continue;
171             return dev;
172         }
173         for (dev = dev_base; dev; dev = dev->next)
174         {
175             if (!(dev->flags & IFF_UP))
176                 continue;
177             if (dev->flags & IFF_POINTOPOINT)
178                 continue;
179             if (dev->pa_mask & (addr ^ dev->pa_addr))
180                 continue;
181             return dev;
182         }
183         return NULL;
184 }
```

ip_dev_check 函数返回参数 IP 地址所在的网络设备。即返回配置有参数表示的 IP 地址的网络设备。该函数实现中两段代码极为相似，不过对参数采用了截然不同的解释。163-172 行是对点对点连接方式下网络设备的检查，此时参数表示的对端 IP 地址，返回本地连接端的网络设备。173-182 行是对一般网络设备（工作于以太网上）的检查，而且只检测子网网络地址部分，如果相符，返回该网络设备。在两种不同的连接下，返回该网络设备的前提条件都是该网络设备必须处于工作状态，否则免谈！因为只有对于一个工作的网络设备，其配置的 IP 地址才有意义（才能用于通信）。

net/inet/devinit.c 文件小结

该文件相对于其他文件较为简单，只定义了四个功能函数，都是关于 IP 地址相关方面的，被其它协议模块调用对 IP 地址进行判断或者进行相关联信息的获取，实现较为简单，此处不再赘述。

前文中一直在提链路层模块，其实既非指以太网首部处理文件 eth.c，或者 802 系列协议首部处理文件 p8022.c, p8023.c 等等，而是指 dev.c 文件所表示函数集合。之所以将其称为链路层模块，主要是因为从功能上看，其位于网络设备驱动程序和网络层协议实现模块之间，作为二者之间的传输通道，并调用以太网或者 802 系列协议首部处理函数进行相关处理，我们可以说 dev.c 文件所代表的操作函数集是作为网络设备驱动模块和网络层协议模块之间的接口模块而存在，其同时还调用链路层首部相关处理函数进行辅助处理。对驱动层的接口函数 netif_rx, 以及对网络层的接口函数 net_bh 函数均定义在 dev.c 文件中，所以在整个网络栈实现中，dev.c 文件作用重大，从其实现的功能的角度，我们就将其称为链路层模块实现文件。下面我们即对该文件进行分析，其中我们需要特别关注其与其下的驱动层和其上的网络层之间的衔接，或者说其如何完成驱动层和网络层之间数据包的传输。

2.39 net/inet/dev.c 文件

```
1   /*
2    *  NET3    Protocol independent device support routines.
3    *
```

```
4     *         This program is free software; you can redistribute it and/or
5     *         modify it under the terms of the GNU General Public License
6     *         as published by the Free Software Foundation; either version
7     *         2 of the License, or (at your option) any later version.
8     *
9     *    Derived from the non IP parts of dev.c 1.0.19
10    *         Authors: Ross Biro, <bir7@leland.Stanford.Edu>
11    *                  Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
12    *                  Mark Evans, <evansmp@uhura.aston.ac.uk>
13    *
14    *    Additional Authors:
15    *         Florian la Roche <rzsfl@rz.uni-sb.de>
16    *         Alan Cox <gw4pts@gw4pts.ampr.org>
17    *         David Hinds <dhinds@allegro.stanford.edu>
18    *
19    *    Changes:
20    *         Alan Cox      :     device private ioctl copies fields back.
21    *         Alan Cox      :     Transmit queue code does relevant stunts to
22    *                       keep the queue safe.
23    *         Alan Cox      :     Fixed double lock.
24    *         Alan Cox      :     Fixed promisc NULL pointer trap
25    *         ????????      :     Support the full private ioctl range
26    *         Alan Cox      :     Moved ioctl permission check into drivers
27    *         Tim Kordas    :     SIOCADDMULTI/SIOCDELMULTI
28    *         Alan Cox      :     100 backlog just doesn't cut it when
29    *                       you start doing multicast video 8)
30    *         Alan Cox      :     Rewrote net_bh and list manager.
31    *         Alan Cox      :     Fix ETH_P_ALL echoback lengths.
32    *
33    *    Cleaned up and recommented by Alan Cox 2nd April 1994. I hope to have
34    *    the rest as well commented in the end.
35    */

36   /*
37    *    A lot of these includes will be going walkies very soon
38    */

39   #include <asm/segment.h>
40   #include <asm/system.h>
41   #include <asm/bitops.h>
42   #include <linux/config.h>
43   #include <linux/types.h>
44   #include <linux/kernel.h>
45   #include <linux/sched.h>
```

```
46  #include <linux/string.h>
47  #include <linux/mm.h>
48  #include <linux/socket.h>
49  #include <linux/sockios.h>
50  #include <linux/in.h>
51  #include <linux/errno.h>
52  #include <linux/interrupt.h>
53  #include <linux/if_ether.h>
54  #include <linux/inet.h>
55  #include <linux/netdevice.h>
56  #include <linux/etherdevice.h>
57  #include <linux/notifier.h>
58  #include "ip.h"
59  #include "route.h"
60  #include <linux/skbuff.h>
61  #include "sock.h"
62  #include "arp.h"

63  /*
64   *  The list of packet types we will receive (as opposed to discard)
65   *  and the routines to invoke.
66   */

67  struct packet_type *ptype_base = NULL;
```

在分析 ARP 协议以及 IP 协议时，我们一再谈到为了能够从链路层模块接收数据包，他们都必须定义一个 packet_type 结构并向链路层模块进行注册。那么链路层模块如何维护这些注册的 packet_type 结构？此处我们便得到了答案：即链路层将网络层协议注册的 packet_type 结构都插入到由 ptype_base 指向的队列中，当链路层模块完成数据包的本层处理后，需要向上层传送该数据包时，其遍历由 ptype_base 变量指向的 packet_type 结构队列，根据数据包链路层首部中的协议类型字段，对队列中每个 packet_type 结构进行匹配，当查询到一个匹配项，其调用该 packet_type 结构中 func 指针指向的接收函数，从而完成数据包的向上传送，具体代码我们在下文中 net_bh 函数实现中可以见到。

```
68  /*
69   *  Our notifier list
70   */

71  struct notifier_block *netdev_chain=NULL;
```

netdev_chain 变量的作用类似于 ptype_base，不过其维护的是 notifier_block 结构类型的队列，这是一种事件注册和通知机制。我们在上文中刚刚讨论的 ARP 协议实现中，在 arp_init 函数中，就调用 register_netdevice_notifier 函数进行了事件通知注册，即注册一个事件发生时被调用的函数，从而在所关注事件发生时，调用该函数完成对事件的响应。事件发生时的调

用函数以及相关参数（如关注的事件类型）都由一个 notifier_block 结构表示。当模块表示对某事件进行关注时，其必须定义一个 notifier_block 结构，并向事件发生通知者进行注册，如此，当事件发生时，事件通知者通过调用 notifier_block 结构中所注册的函数完成事件的通知。那么此处我们看到链路层模块作为与低层驱动层直接接触的模块，其负责网络设备相关事件（如停止或启动工作）的通知，那么其他模块（如 ARP 协议模块）如果需要得到相关事件的通知，其就必须定义一个 notifier_block 结构，并向链路层模块进行注册。链路层模块提供注册函数：register_netdevice_notifier，并将其它模块注册的所有的 notifier_block 结构都插入到由 netdev_chain 全局变量指向的队列中。如此，一旦发生相关事件（主要是网络设备工作状态的变化），其通过遍历 netdev_chain 队列，调用队列中每个 notifier_block 结构元素中包含的函数完成事件的通知，至于说，此次事件是否为该函数所关注的事件，那么由该函数的具体实现决定。读者可回头查看前文中 ARP 协议实现模块中相关的注册代码（arp_init）及注册的函数（实现代码），从而获得一个基本的认识。

```
72  /*
73   *  Device drivers call our routines to queue packets here. We empty the
74   *  queue in the bottom half handler.
75   */

76  static struct sk_buff_head backlog =
77  {
78      (struct sk_buff *)&backlog, (struct sk_buff *)&backlog
79  #ifdef CONFIG_SKB_CHECK
80      ,SK_HEAD_SKB
81  #endif
82  };

83  /*
84   *  We don't overdo the queue or we will thrash memory badly.
85   */

86  static int backlog_size = 0;
```

76,86 这两行定义的两个变量 backlog，backlog_size 的作用十分重要，这是链路层模块暂存由驱动层上传的被接收数据包的存储之所。网络设备驱动程序（一般）在其中断程序中完成数据包的封装后（即将数据封装在 sk_buff 结构中），通过调用 netif_rx 函数（该函数在 Linux 内核当前版本中也一直在使用，虽然代码有所改变，但完成的作用相同）将数据包上传给网络栈进行接收和处理。我们可以从 netif_rx 函数的实现代码中看到，该函数由于一般可能在驱动程序的中断程序中被调用，为了减少执行时间，其通过下半部分执行方式进行数据包的实际接收和处理，不过其必须完成数据包在本层的初步接收（即需要完成数据包从驱动层到链路层的转移），这个初步接收就是指将数据包暂时缓存到 backlog 指向的队列中，backlog_size 则是控制队列中数据包的个数不可过大，从而造成数据包的过多累计，占用大量内存空间，而且不会对接收速度形成正面影响。所以我们说，主机接收的数据包都会经过 backlog 暂存队列，具体的这个暂存队列中数据包的处理和向上层传送则由下半部分执行函数 net_bh 完成。有关 netif_rx, net_bh 函数，下文中都有分析介绍。

```
87  /*
88   *  Return the lesser of the two values.
89   */

90  static __inline__ unsigned long min(unsigned long a, unsigned long b)
91  {
92      return (a < b)? a : b;
93  }


94
    /*****************************************************************************
*****************

95              Protocol management and registration routines

96
    *****************************************************************************
*****************/

97  /*
98   *  For efficiency
99   */

100 static int dev_nit=0;
```

dev_nit 变量表示 ptype_base 指向的队列中数据包接收类型为 ALL 的元素的个数，所谓数据包接收类型为 ALL：即表示注册的接收函数将接收所有数据包。我们称之为软件上的一种混杂模式（与网络设备可设置的混杂模式相对应）。这种软件上对混杂模式的支持对某些上层应用软件的实现提供了必要的条件，如抓包工具，tcpdump 应用程序都需要使用这种混杂模式工作方式，当然要完成他们的功能，还需对硬件进行相应的配置，不过只是后话。

```
101 /*
102  *  Add a protocol ID to the list. Now that the input handler is
103  *  smarter we can dispense with all the messy stuff that used to be
104  *  here.
105  */

106 void dev_add_pack(struct packet_type *pt)
107 {
108     if(pt->type==htons(ETH_P_ALL))
109         dev_nit++;
110     pt->next = ptype_base;
```

```
111         ptype_base = pt;
112 }

113 /*
114  *    Remove a protocol ID from the list.
115  */

116 void dev_remove_pack(struct packet_type *pt)
117 {
118         struct packet_type **pt1;
119         if(pt->type==htons(ETH_P_ALL))
120                 dev_nit--;
121         for(pt1=&ptype_base; (*pt1)!=NULL; pt1=&((*pt1)->next))
122         {
123                 if(pt==(*pt1))
124                 {
125                         *pt1=pt->next;
126                         return;
127                 }
128         }
129 }
```

dev_add_pack，dev_remove_pack 函数具体完成网络层协议的注册（加载）和卸载功能。dev_add_pack 函数实现非常简单，即将表示新注册协议的 packet_type 结构插入到 ptype_base 队列的头部。上文中我们刚刚提到软件上的混杂模式，此处（108 行代码）就表示为 ETH_P_ALL 协议类型。dev_remove_pack 函数实现与 dev_add_pack 函数相对应，此处不再赘述。

以下三个函数称为设备接口函数，都是和网络设备操作关联的，具体地：
dev_get：根据其传入参数表示的设备名称查找其 device 结构；
dev_open：打开一个设备；设备打开后将处于正常工作状态；
dev_close：关闭一个设备；即设置设备为非工作状态。
具体实现代码如下。

```
130 /*****************************************************************************
131                             Device Interface Subroutines
132 *****************************************************************************/

133 /*
134  *    Find an interface by name.
135  */

136 struct device *dev_get(char *name)
137 {
```

```
138     struct device *dev;

139     for (dev = dev_base; dev != NULL; dev = dev->next)
140     {
141         if (strcmp(dev->name, name) == 0)
142             return(dev);
143     }
144     return(NULL);
145 }

146 /*
147  *  Prepare an interface for use.
148  */

149 int dev_open(struct device *dev)
150 {
```

每个网络设备在内核中都由一个 device 结构（后改名为 netdevice）表示。这些 device 结构既可在驱动程序（以模块方式加入内核）中动态创建，亦可静态的编入内核映像中。由于各具体网络设备硬件的不同操作方式，故在 device 结构中定义了一系列函数指针，这些函数指针指向的函数中包括设备打开函数，数据包发送函数等，因为这些函数具体实现涉及到具体的硬件寄存器操作，所以对于不同的设备，实现则不同，为了维护他们对内核表现形式上的一致性，故通过在 device 结构中定义一系列函数指针的方式来进行。当内核需要操作某个具体网络设备硬件时，其首先得到该硬件对应的 device 结构，通过调用该结构中相关函数指针指向的函数对该硬件进行具体的实际操作，从而保持了内核调用方式上的一致性，同时支持多种设备。dev_open 函数所体现的思想也是如此，该函数是一个系统函数，完成具体设备的启动工作，在执行方式上，其就是调用该设备本身特定的打开函数完成设备的启动工作。这也就是 device 结构作为软件上对硬件进行抽象所实现的功能（对不同的硬件进行不同的控制）。

```
151     int ret = 0;

152     /*
153      *  Call device private open method
154      */
155     if (dev->open)
156         ret = dev->open(dev);

157     /*
158      *  If it went open OK then set the flags
159      */

160     if (ret == 0)
161     {
```

```
162             dev->flags |= (IFF_UP | IFF_RUNNING);
163             /*
164              *    Initialise multicasting status
165              */
166 #ifdef CONFIG_IP_MULTICAST
167             /*
168              *    Join the all host group
169              */
170             ip_mc_allhost(dev);
171 #endif
172             dev_mc_upload(dev);
173             notifier_call_chain(&netdev_chain, NETDEV_UP, dev);
174         }
175         return(ret);
176 }
```

155-156 行通过调用具体设备对应的打开函数完成设备的启动，函数返回 0 表示启动成功，此时还需要完成其它协议辅助工作，如正确设置设备对应的标志位从而标志设备为工作状态，配置组播地址（dev_mc_upload），通知其它模块该设备的启动（notifier_call_chain）。

对于一个具体的网络接口设备，其需要配置的有关寄存器包括：硬件地址（这在某些嵌入式系统中经常如此，不过对于普通 PC 一般硬件地址是固化的，无须配置）；组播地址；广播，组播数据包接收使能等等。换句话说，组播地址仅仅维护在软件形式上远远不够，还需要进行硬件寄存器的具体配置，这个工作由 dev_mc_upload 函数，该函数定义在 dev_mcast.c 中，在本书前文中已有介绍，该函数继续调用硬件特定的设置函数对该硬件相关寄存器进行操作，完成组播地址的配置。此处所要表达的思想是，网络栈虽然提供了一系列对硬件操作的函数，但具体的工作的完成需要具体操作硬件，而不同硬件具有不同的操作方式，为了维护统一的调用接口，内核采用数据结构封装方式，以函数指针的工作方式对不同的设备赋予不同的操作函数，从而完成最终的任务，而这些具体硬件操作函数的编程则是由驱动程序员完成，如果在 Linux 系统下进行过设备驱动开发，那么以上的说明应该较为容易理解，没有这方面经验的读者，可参考[2]。notifier_call_chain 由于实现较为简单，我们如下只给出其定义，不再赘述。

```
/*include/linux/notifier.h*/
12  struct notifier_block
13  {
14          int (*notifier_call)(unsigned long, void *);
15          struct notifier_block *next;
16          int priority;
17  };

19  #define NOTIFY_DONE             0x0000          /* Don't care */
20  #define NOTIFY_OK           0x0001          /* Suits me */
21  #define NOTIFY_STOP_MASK    0x8000          /* Don't call further */
22  #define NOTIFY_BAD          (NOTIFY_STOP_MASK|0x0002) /* Bad/Veto action  */
```

```
......
55  extern __inline__ int notifier_call_chain(struct notifier_block **n,unsigned long val,void *v)
56  {
57      int ret=NOTIFY_DONE;
58      struct notifier_block *nb = *n;
59      while(nb)
60      {
61          ret=nb->notifier_call(val,v);
62          if(ret&NOTIFY_STOP_MASK)
63              return ret;
64          nb=nb->next;
65      }
66      return ret;
67  }
......
76  #define NETDEV_UP   0x0001   /* For now you can't veto a device up/down */
77  #define NETDEV_DOWN   0x0002
```

以上代码均出自 include/linux/notifier.h 头文件中，包括了 notifier_block 结构的定义以及 notifier_call_chain 函数实现和相关事件定义，此处给出了网络栈协议关注的两个事件定义。

```
177 /*
178  *  Completely shutdown an interface.
179  */

180 int dev_close(struct device *dev)
181 {
182     /*
183      *  Only close a device if it is up.
184      */

185     if (dev->flags != 0)
186     {
187         int ct=0;
188         dev->flags = 0;
189         /*
190          *  Call the device specific close. This cannot fail.
191          */
192         if (dev->stop)
193             dev->stop(dev);

194         notifier_call_chain(&netdev_chain, NETDEV_DOWN, dev);
195 #if 0
```

```
196             /*
197              *    Delete the route to the device.
198              */
199 #ifdef CONFIG_INET
200             ip_rt_flush(dev);
201             arp_device_down(dev);
202 #endif
203 #ifdef CONFIG_IPX
204             ipxrtr_device_down(dev);
205 #endif
206 #endif
207             /*
208              *    Flush the multicast chain
209              */
210             dev_mc_discard(dev);
211             /*
212              *    Blank the IP addresses
213              */
214             dev->pa_addr = 0;
215             dev->pa_dstaddr = 0;
216             dev->pa_brdaddr = 0;
217             dev->pa_mask = 0;
218             /*
219              *    Purge any queued packets when we down the link
220              */
221             while(ct<DEV_NUMBUFFS)
222             {
223                 struct sk_buff *skb;
224                 while((skb=skb_dequeue(&dev->buffs[ct]))!=NULL)
225                     if(skb->free)
226                         kfree_skb(skb,FREE_WRITE);
227                 ct++;
228             }
229         }
230         return(0);
231 }
```

dev_close 函数相对较长，其实现思想比较简单，首先调用硬件特定关闭函数（dev->close 指向的函数）进行设备的关闭，并将此事件通知到其它感兴趣协议模块（notifier_call_chain），此后对设备对应 device 结构中一些字段进行了调整：首先将维护的组播地址列表释放；其次将设备配置的 IP 地址清零；再者释放设备中仍在缓存的尚未发送出去的所有数据包。以上这些所有的操作的理解，我们应从一个设备启动工作的角度考虑，当一个网络设备启动工作时，其应该是完全“清白”的，不应存在有之前配置的任何信息。因为我们可能配置设备为另一种工作方式，如果存在原有信息，可能造成设备的非正常工作。199-202 行代码涉及

到将与设备绑定的路有表项和 ARP 表项进行清除，这是必要的，因为对应的设备停止工作后，这些表项将不再有意义，使用这些表项将无法将数据包从对应设备发送出去，因为对应设备已经停止工作了。

```
232 /*
233  *  Device change register/unregister. These are not inline or static
234  *  as we export them to the world.
235  */

236 int register_netdevice_notifier(struct notifier_block *nb)
237 {
238         return notifier_chain_register(&netdev_chain, nb);
239 }

240 int unregister_netdevice_notifier(struct notifier_block *nb)
241 {
242         return notifier_chain_unregister(&netdev_chain,nb);
243 }
```

register_netdevice_notifier, unregister_netdevice_notifier 这两个函数是链路层模块提供给其他协议模块进行事件通知注册和注销的两个接口函数。涉及到 netdev_chain 变量，notifier_block 结构，这些在前文中都已进行了说明。此处调用的 notifier_chain_register，notifier_chain_unregister 都定义在 include/linux/notifier.h 头文件中，我们此处同样给出他们的实现代码，如下：

```
/*include/linux/notifier.h*/
23  extern __inline__ int notifier_chain_register(struct notifier_block **list, struct notifier_block *n)
24  {
25          while(*list)
26          {
27                  if(n->priority > (*list)->priority)
28                          break;
29                  list= &((*list)->next);
30          }
31          n->next = *list;
32          *list=n;
33          return 0;
34  }

35  /*
36   *  Warning to any non GPL module writers out there.. these functions are
37   *  GPL'd
38   */
```

```
39 extern __inline__ int notifier_chain_unregister(struct notifier_block **nl, struct notifier_block
   *n)
40  {
41      while((*nl)!=NULL)
42      {
43          if((*nl)==n)
44          {
45              *nl=n->next;
46              return 0;
47          }
48          nl=&((*nl)->next);
49      }
50      return -ENOENT;
51  }
```

notifier_chain_register，notifier_chain_unregister 实现都较简单，我们不再赘述，继续我们对 dev.c 文件其它函数的分析。

```
244 /*
245  *  Send (or queue for sending) a packet.
246  *
247  *  IMPORTANT: When this is called to resend frames. The caller MUST
248  *  already have locked the sk_buff. Apart from that we do the
249  *  rest of the magic.
250  */

251 void dev_queue_xmit(struct sk_buff *skb, struct device *dev, int pri)
252 {
253     unsigned long flags;
254     int nitcount;
255     struct packet_type *ptype;
256     int where = 0;      /* used to say if the packet should go    */
257                         /* at the front or the back of the    */
258                         /* queue - front is a retransmit try  */
```

dev_queue_xmit 是提供给网络层进行数据包发送的一个接口函数。网络层协议实现模块完成其本层的处理后，调用 dev_queue_xmit 函数将数据包下传给链路层模块，而 dev_queue_xmit 函数本身负责将数据包传递给驱动程序，由驱动程序最终将数据发送到物理介质上。本书前文中我们已经对网络层 IP，ARP 协议实现模块等进行了分析，这些模块中数据包发送函数都调用 dev_queue_xmit 函数进行数据包的发送，如 ip_queue_xmit, arp_send 函数等，由此完成从网络层到链路层数据发送通道的衔接。当然，dev_queue_xmit 函数并非只被上层模块调用，其链路层本身也调用该函数进行（在链路层被缓存的）数据包的发送（具体是在 dev_tint 函数中）。

dev_queue_xmit 函数实现代码较长，我们分段进行分析，首先对函数参数进行说明：
skb：被发送的数据包；
dev：数据包发送网络接口设备；
pri：网络接口设备忙时，缓存该数据包时使用的优先级。

第三个参数表示如果由于设备忙，无法立刻发送该数据包，那么缓存该数据包时使用何种优先级。优先级高下最终体现在数据包插入到硬件缓存三个队列（DEV_NUMBUFFS=3）中的哪一个，网络设备从硬件缓存队列中取数据包进行发送时，先取第一个队列中数据包，第一个队列取完后，才取第二个队列中数据包，其次第三个队列。所以插入队列的不同将造成数据包发送延迟的不同。高优先级的数据包自然插入第一个队列，其次第二，其次第三。从 device 结构定义中我们可以看到，硬件队列的维护是通过数组方式进行的，我们从 netdevice.h 头文件中可以了解到这一点，具体如下：

```
/*include/linux/netdevice.h*/
30   #define DEV_NUMBUFFS   3
.....
54   struct device
55   {
     ......
111     /* Pointer to the interface buffers. */
112     struct sk_buff_head       buffs[DEV_NUMBUFFS];
     ......
138  };
```

关于 device 结构的完整定义，请参考本书第一章中 netdevice.h 头文件的有关内容。

```
259        if (dev == NULL)
260        {
261             printk("dev.c: dev_queue_xmit: dev = NULL\n");
262             return;
263        }

264        if(pri>=0 && !skb_device_locked(skb))
265             skb_device_lock(skb);   /* Shove a lock on the frame */
```

259-265 行对发送设备接口以及数据包的锁定状态进行检查。在调用硬件发送函数之前，数据包必须被锁定，以避免其它地方代码同时操作该数据包，造成内核不一致。

```
266  #ifdef CONFIG_SLAVE_BALANCING
267        save_flags(flags);
268        cli();
269        if(dev->slave!=NULL && dev->slave->pkt_queue < dev->pkt_queue &&
270                        (dev->slave->flags & IFF_UP))
271             dev=dev->slave;
272        restore_flags(flags);
```

```
273 #endif
274 #ifdef CONFIG_SKB_CHECK
275         IS_SKB(skb);
276 #endif
277         skb->dev = dev;
```

266-273 行检查是否使用了主从设备的连接方式，如果采用了这种方式，则发送数据包时，可在两个设备之间平均负载。275 行是检查数据包的合法性，主要是对 sk_buff 结构中一些字段值的检查。IS_SKB 宏定义在 include/linux/sk_buff.h 中，具体执行函数为 skb_check，定义在 net/inet/sk_buff.c 中，sk_buff.c 文件定义了操作 sk_buff 结构的一系列功能函数，我们将在本章最后进行介绍。277 行初始化 skb->dev 字段指向数据包发送设备对应结构。

```
278         /*
279          *    This just eliminates some race conditions, but not all...
280          */
281         if (skb->next != NULL)
282         {
283                 /*
284                  *    Make sure we haven't missed an interrupt.
285                  */
286                 printk("dev_queue_xmit: worked around a missed interrupt\n");
287                 dev->hard_start_xmit(NULL, dev);
288                 return;
289     }
```

281-289 行代码较为难懂，因为无论是驱动程序层的间接调用（通过 net_bh）还是网络栈系统代码的主动调用，在调用 dev_queue_xmit 函数时，传入的数据包都首先从其所在队列中删除，即 skb->next 一定为 NULL，281 行的 if 语句似乎不可能发生，如果发生，则表示系统代码出现漏洞，仅靠 287 行的调用是不会解决问题的。但是此处的检查又可以说是必要的，一个数据包不能同时存在于两个队列之中，以免造成竞争条件，287 行以参数 NULL 调用硬件发送函数的目的在于：一方面绕过对当前的这个数据包的处理；另一方面从正常队列中取数据包进行发送。我们可以看到 8390.c 文件中如下这段代码，这段代码是硬件发送函数 ei_start_xmit 中的一段，即当硬件是 DP8390 时，dev-> hard_start_xmit 指针指向的函数：

/*drivers/net/8390.c – ei_start_xmit 函数代码片段*/

```
142         /* Sending a NULL skb means some higher layer thinks we've missed an
143                 tx-done interrupt. Caution: dev_tint() handles the cli()/sti()
144                 itself. */
145         if (skb == NULL) {
146                 dev_tint(dev);
147                 return 0;
148         }
```

此处驱动程序发送函数当检测到 skb 为 NULL 时，调用 dev_tint 函数进行数据包的发送。dev_tint 函数即定义在 dev.c 中，下文中我们可以看到其实现代码，该函数从硬件队列中取

数据包，重新调用 dev_queue_xmit 函数进行数据包的发送，在介绍该函数时，我们将特别关注 dev_tint 函数调用 dev_queue_xmit 时所使用的第三个参数将为负值，表示当前发送的数据包是从硬件缓存队列中取的，而非上层传递的新数据包。

综合而言，281-289 行代码仍是对数据包的合法性进行检查，调用 dev_queue_xmit 函数进行发送的数据包必须游离于任何队列之外，即 skb->next 必须为 NULL，否则该函数将不对该数据包进行任何操作，287 行调用的目的在于维护数据包发送通道的连续性，对此我们可以假设一种极端的情况，假设主机在一段时间内没有接收到任何数据包，而且此次调用 dev_queue_xmit 函数，系统上层也将沉静一段时间，如果 281 行 if 语句为真，不经 287 行调用直接返回，那么硬件队列中可能滞留的其它数据包在此后一段时间内将得不到处理。因为硬件队列中的数据包的处理是由数据包接收函数（net_bh）和发送函数（dev_queue_xmit）驱动的（在下文中介绍 net_bh 函数实现时，读者可以理解这一点）。

```
290     /*
291      *  Negative priority is used to flag a frame that is being pulled from the
292      *  queue front as a retransmit attempt. It therefore goes back on the queue
293      *  start on a failure.
294      */

295     if (pri < 0)
296     {
297         pri = -pri-1;
298         where = 1;
299     }
```

优先级参数为负数，表示当前处理的数据包是从硬件队列中取下的，而非上层传递的新数据包，这种情况在 dev_tint 函数中会看到。297-298 两行代码一方面得到真正的优先级（注意这种计算方式与 dev_tint 函数中的计算方式对应，以得到该数据包原有的优先级，如此，当此次由于设备忙，无法发送出去时，则将该数据包插入到其原先所在队列中），另一方面将 where 变量设置为 1，表示当前数据包是一个老的数据包，是从硬件队列中获取的。这个参数的作用下面将会看到。

```
300     if (pri >= DEV_NUMBUFFS)
301     {
302         printk("bad priority in dev_queue_xmit.\n");
303         pri = 1;
304     }
```

我们说硬件队列的维护是以数组的方式进行的，device 结构中 buffs 字段就表示这个硬件队列，这个字段是一个数组，每个数组元素指向一个 sk_buff 结构队列，现数组共有 DEV_NUMBUFFS 元素，即最大索引为 DEV_NUMBUFFS-1，以索引 0 为最高优先级。300 行即检测优先级范围是否超过限定值，如果超过，则设置优先级为中等优先级（因为 DEV_NUMBUFFS=3，所以当前只有三个优先级：0，1，2）。

```
305     /*
306      *  If the address has not been resolved. Call the device header rebuilder.
307      *  This can cover all protocols and technically not just ARP either.
308      */

309     if (!skb->arp && dev->rebuild_header(skb->data, dev, skb->raddr, skb)) {
310         return;
311     }
```

skb->arp 字段的含义前文中介绍 ARP 协议实现模块时以及本书其它部分已经有所交待，该字段值表示是否完成链路层硬件地址解析，该字段值设置为 1，表示成功完成链路层首部的创建，否则需要调用 dev->rebuild_header 函数（对于以太网而言，其指向 eth_rebuild_header 函数）完成链路层首部的创建工作，这有可能启动 ARP 地址解析过程，而一旦启动 ARP 解析过程，则数据包的发送则由 ARP 协议模块负责，所以 310 行直接返回，没有将数据包重新插入到硬件队列中。

```
312     save_flags(flags);
313     cli();
314     if (!where) {
315 #ifdef CONFIG_SLAVE_BALANCING
316         skb->in_dev_queue=1;
317 #endif
318         skb_queue_tail(dev->buffs + pri,skb);
319         skb_device_unlock(skb); /* Buffer is on the device queue and can be freed safely */
320         skb = skb_dequeue(dev->buffs + pri);
321         skb_device_lock(skb);       /* New buffer needs locking down */
322 #ifdef CONFIG_SLAVE_BALANCING
323         skb->in_dev_queue=0;
324 #endif
325     }
326     restore_flags(flags);
```

312-326 行代码表示调用 dev_queue_xmit 进行数据包发送时，实际发送的数据包并非一定是作为参数传入的数据包。当 where=0 时，表示这是从上层接收的新数据包，那么首先将其挂入硬件队列中，再从硬件队列中取数据包。注意此时从硬件队列中取的数据包可能不是我们挂入的数据包，因为当挂入队列时，是插入到队列的尾部，而取数据包时，是从队列头部取，所以如果队列中已经有数据包，则此次发送的数据包就不是作为参数传入的数据包。这种先挂入再取的方式是为了公平性，因为早些发送的相同优先级的数据包可能由于设备正忙被挂入到硬件队列中，那么此时后来发送的数据包就不能越过这些早些的数据包提前发送出去，其应该排在这些早些发送的数据包之后发送，这种方式一方面保证公平性，另一方面也是为了保证上层模块的正常工作。因为如果此处进行乱序发送，将极有可能造成数据包的真正丢失（其中原因请读者自己结合 TCP 协议工作方式思考）。

```
327     /* copy outgoing packets to any sniffer packet handlers */
```

```
328     if(!where)
329     {
330         for (nitcount= dev_nit, ptype = ptype_base; nitcount > 0 && ptype != NULL;
ptype = ptype->next)
331         {
332             /* Never send packets back to the socket
333              * they originated from - MvS (miquels@drinkel.ow.org)
334              */
335             if (ptype->type == htons(ETH_P_ALL) &&
336                 (ptype->dev == dev || !ptype->dev) &&
337                 ((struct sock *)ptype->data != skb->sk))
338             {
339                 struct sk_buff *skb2;
340                 if ((skb2 = skb_clone(skb, GFP_ATOMIC)) == NULL)
341                     break;
342                 /*
343                  *  The protocol knows this has (for other paths) been taken off
344                  *  and adds it back.
345                  */
346                 skb2->len-=skb->dev->hard_header_len;
347                 ptype->func(skb2, skb->dev, ptype);
348                 nitcount--;
349             }
350         }
351     }
```

328-351 行代码是内核对软件混杂模式的一种支持，对于一个新的数据包，其遍历 ptype_base 指向的网络层协议队列，在其中查找混杂模式接收协议，将发送的数据包回送一份给这些协议从而完成对软件混杂模式的支持。其中 dev_nit 变量表示 ETH_P_ALL 协议类型的个数，从而完成对这些协议的处理后可以及时退出循环，避免循环到底。正如定义 dev_nit 变量时对其目的的阐述：为了提高效率。

```
352     if (dev->hard_start_xmit(skb, dev) == 0) {
353         /*
354          *  Packet is now solely the responsibility of the driver
355          */
356         return;
357     }
```

经过了以上处理，352-357 行调用 device 结构中 hard_start_xmit 指针指向的硬件发送函数最终将数据包发送出去。诚如前文所述，每个网络接口设备都对应一个 device 结构，都有其自身特定的操作（如发送）函数，从而对特定的硬件寄存器进行操作，此处通过调用 device 结构中的发送函数完成对具体硬件的操作，从而完成数据包的最终发送（即数据以电平形式出现在物理介质上）。发送函数返回 0 表示成功操作硬件进行了发送，否则表示发送失败（如

设备正忙于发送其它数据包），此时需要将数据包重新插入到硬件队列中，等待下次重新发送。下面的代码即完成数据包的重新缓存。

```
358         /*
359          *  Transmission failed, put skb back into a list. Once on the list it's safe and
360          *  no longer device locked (it can be freed safely from the device queue)
361          */
362         cli();
363 #ifdef CONFIG_SLAVE_BALANCING
364         skb->in_dev_queue=1;
365         dev->pkt_queue++;
366 #endif
367         skb_device_unlock(skb);
368         skb_queue_head(dev->buffs + pri,skb);
369         restore_flags(flags);
370 }
```

注意 368 行重新将数据包插入硬件队列时，是将数据包插入到队列的头部，因为其在发送时间上仍然在其它相同优先级之前，下一次发送时，该数据包仍然优先发送。这与一个从上层传递下来的数据包的插入方式不同。这一点请读者（结合以上 318，320，368 行代码）注意一下。

```
371 /*
372  *  Receive a packet from a device driver and queue it for the upper
373  *  (protocol) levels.  It always succeeds. This is the recommended
374  *  interface to use.
375  */

376 void netif_rx(struct sk_buff *skb)
377 {
```

netif_rx 函数作为驱动程序向上传送数据包的接口函数，完成数据包从驱动层到链路层的传递。该函数一般在驱动程序接收中断例程中被调用，驱动程序完成接收数据的封装后调用 netif_rx 函数将数据包传递给链路层进行处理。所以尽量减少执行时间是 netif_rx 函数实现的基本特点。我们从其如下的实现方式（即使用下半部分：bottom half）也可看出这一点。

```
378         static int dropping = 0;

379         /*
380          *  Any received buffers are un-owned and should be discarded
381          *  when freed. These will be updated later as the frames get
382          *  owners.
383          */
384         skb->sk = NULL;
```

```
385         skb->free = 1;
386         if(skb->stamp.tv_sec==0)
387                 skb->stamp = xtime;

388         /*
389           *    Check that we aren't overdoing things.
390           */

391         if (!backlog_size)
392                 dropping = 0;
393         else if (backlog_size > 300)
394                 dropping = 1;

395         if (dropping)
396         {
397                 kfree_skb(skb, FREE_READ);
398                 return;
399         }

400         /*
401           *    Add it to the "backlog" queue.
402           */
403 #ifdef CONFIG_SKB_CHECK
404         IS_SKB(skb);
405 #endif
406         skb_queue_tail(&backlog,skb);
407         backlog_size++;

408         /*
409           *    If any packet arrived, mark it for processing after the
410           *    hardware interrupt returns.
411           */

412         mark_bh(NET_BH);
413         return;
414 }
```

backlog_size 表示 backlog 队列中缓存的数据包的数目，这个数目有一个限定值，不可过大，否则将有可能耗费所有的系统内存。当数据包的数目大于 300（393 行）时，则对数据包进行简单丢弃操作。否则将数据包暂时挂入 backlog 队列中，具体的处理交给下半部分处理，412 行即启动下半部分。对应 NET_BH 的下半部分执行函数为 net_bh，我们将在下文中分析该函数代码。

注意：netif_rx 函数作为驱动程序向网络栈上层（即链路层）的接口函数，在当前最新 Linux

版本中仍然在使用，虽然实现代码有所差异，但其完成的作用相同，而且该函数现在作为唯一的接口函数供驱动程序调用。在本版本中（linux-1.2.13），驱动层和链路层耦合的依然较为紧密，两层之间还存在过多的交织，如 dev_tint, dev_rint, mark_bh 等等函数仍然对驱动层可见，而且内核还依赖于驱动层对这些函数的调用！

```
415 /*
416  *  The old interface to fetch a packet from a device driver.
417  *  This function is the base level entry point for all drivers that
418  *  want to send a packet to the upper (protocol) levels.   It takes
419  *  care of de-multiplexing the packet to the various modules based
420  *  on their protocol ID.
421  *
422  *  Return values: 1 <- exit I can't do any more
423  *                 0 <- feed me more (i.e. "done", "OK").
424  *
425  *  This function is OBSOLETE and should not be used by any new
426  *  device.
427  */

428 int dev_rint(unsigned char *buff, long len, int flags, struct device *dev)
429 {
430     static int dropping = 0;
431     struct sk_buff *skb = NULL;
432     unsigned char *to;
433     int amount, left;
434     int len2;
```

正如函数前的注释所述，该函数是一个遗留物，为了保持与之前版本的兼容性。因为在这之前的版本中，驱动程序需要调用该函数完成数据包的向上传递，虽然现在的新的接口为 netif_rx 函数，但为了这些早期驱动程序仍然能够工作，还必须包含这个函数。我们下面从该函数的实现方式上可以看出，其也是在向 netif_rx 函数靠拢，从而对上形成统一的接口（即都使用 netif_rx 函数）。

```
435     if (dev == NULL || buff == NULL || len <= 0)
436         return(1);
```

以上两行是对相关参数的必要检查，这点非常浅显，我们略过不提。
从该函数实现目的来看，下面要做的是对数据进行封装，并调用 netif_rx 函数向上传递数据包，下面的代码即完成数据封装工作。

```
437     if (flags & IN_SKBUFF)
438     {
439         skb = (struct sk_buff *) buff;
440     }
```

flags 标志位表示参数 buff 指向的数据的存在形式，如果设置了 IN_SKBUFF，则表示驱动层在调用 dev_rint 函数之前已经进行了数据封装工作，那么我们所需做的就是一个简单赋值，否则将由 dev_rint 函数完成封装工作，如下 else 语句块对应的代码。

```
441     else
442     {
443         if (dropping)
444         {
445             if (skb_peek(&backlog) != NULL)
446                 return(1);
447             printk("INET: dev_rint: no longer dropping packets.\n");
448             dropping = 0;
449         }

450         skb = alloc_skb(len, GFP_ATOMIC);
451         if (skb == NULL)
452         {
453             printk("dev_rint: packet dropped on %s (no memory) !\n",
454                     dev->name);
455             dropping = 1;
456             return(1);
457         }

458         /*
459          *  First we copy the packet into a buffer, and save it for later. We
460          *  in effect handle the incoming data as if it were from a circular buffer
461          */

462         to = skb->data;
463         left = len;

464         len2 = len;
465         while (len2 > 0)
466         {
467             amount = min(len2, (unsigned long) dev->rmem_end -
468                     (unsigned long) buff);
469             memcpy(to, buff, amount);
470             len2 -= amount;
471             left -= amount;
472             buff += amount;
473             to += amount;
474             if ((unsigned long) buff == dev->rmem_end)
475                 buff = (unsigned char *) dev->rmem_start;
476         }
```

```
477         }
```

441-447 行完成数据的封装工作。需要注意的是在进行数据的复制时，采用 467 行比较谨慎的方式进行，以免造成缓冲区溢出，dev->rmem_end 表示硬件接收缓存区的最末地址，对应的 dev->rmem_start 表示最初地址，从 465-476 行代码可以看出，硬件缓冲区采用了环形工作方式，此种情况下，数据有可能被分割在缓冲区的尾部和开头一段空间，这才有此段执行方式。

```
478         /*
479          *    Tag the frame and kick it to the proper receive routine
480          */

481         skb->len = len;
482         skb->dev = dev;
483         skb->free = 1;

484         netif_rx(skb);
485         /*
486          *    OK, all done.
487          */
488         return(0);
489 }
```

一旦完成数据的封装，我们就可以调用数据包接收统一接口函数 netif_rx，将数据包正式交给链路层进行处理。此处通过调用 netif_rx 函数的方式，而非直接将数据包挂入 backlog 队列中，是为了提供统一的操作接口。或者说是为了最终取消 dev_rint 函数的存在。从 dev_rint 函数返回结果来看，返回 1 表示数据接收失败，返回 0 表示成功。至于失败时，驱动层如何处理由具体实现定制。

```
490 /*
491  *    This routine causes all interfaces to try to send some data.
492  */

493 void dev_transmit(void)
494 {
495         struct device *dev;

496         for (dev = dev_base; dev != NULL; dev = dev->next)
497         {
498                 if (dev->flags != 0 && !dev->tbusy) {
499                         /*
500                          *    Kick the device
501                          */
502                         dev_tint(dev);
```

```
503             }
504         }
505 }
```

dev_transmit 函数被下半部分执行函数 net_bh 调用用于发送数据包。从下文中 net_bh 函数实现来看，该函数不仅仅将 backlog 中缓存的数据包向上传递，还极力发送硬件缓存队列中滞留的数据包，dev_transmit 函数就是调用接口，其被 net_bh 函数调用用于处理硬件缓存队列中滞留的数据包。dev_transmit 函数本身的实现也较为简单，其遍历系统中所有网络设备，对每个设备调用 dev_tint 函数发送该设备的硬件缓存队列（由对应设备的 device 结构中 buffs 指向）中缓存的数据包。为理解方便，我们提前介绍 dev_tint 函数，如下。

```
641 /*
642  *  This routine is called when an device driver (i.e. an
643  *  interface) is ready to transmit a packet.
644  */

645 void dev_tint(struct device *dev)
646 {
647         int i;
648         struct sk_buff *skb;
649         unsigned long flags;

650         save_flags(flags);
651         /*
652          *  Work the queues in priority order
653          */

654         for(i = 0;i < DEV_NUMBUFFS; i++)
655         {
656                 /*
657                  *  Pull packets from the queue
658                  */


659                 cli();
660                 while((skb=skb_dequeue(&dev->buffs[i]))!=NULL)
661                 {
662                         /*
663                          *  Stop anyone freeing the buffer while we retransmit it
664                          */
665                         skb_device_lock(skb);
666                         restore_flags(flags);
667                         /*
668                          *  Feed them to the output stage and if it fails
669                          *  indicate they re-queue at the front.
```

```
670                     */
671                     dev_queue_xmit(skb,dev,-i - 1);
672                     /*
673                      *    If we can take no more then stop here.
674                      */
675                     if (dev->tbusy)
676                         return;
677                     cli();
678                 }
679         }
680         restore_flags(flags);
681 }
```

dev_tint 函数实现不用多说，其遍历对应设备硬件缓存队列，对队列中每个数据包重新调用 dev_queue_xmit 函数进行发送。此处需要注意的是调用 dev_queue_xmit 函数时优先级的计算方式（671 行）：变量表示实际优先级，在作为参数传入时，变为-i-1，我们回头查看 dev_queue_xmit 函数中当检测到优先级为负值时的计算方式：pri = -pri-1。此处的 pri 即（-i-1），那么 pri=-pri-1=-（-i-1）-1=i。换句话说，在 dev_queue_xmit 函数中我们重新计算得到其原先实际的优先级，采用负值优先级和这种折腾式计算的目的在于正确获得 where 变量的初始化。至于为何采用这种计算方式恐怕只有这段代码的编写者方可做出解释，不过本质上都是一样的。

下面我们介绍链路层模块至为重要的下半部分执行函数 net_bh。由于实现代码较长，我们分段进行分析。

```
506 /*****************************************************************
507                     Receive Queue Processor
508 ******************************************************************/

509 /*
510  *    This is a single non-reentrant routine which takes the received packet
511  *    queue and throws it at the networking layers in the hope that something
512  *    useful will emerge.
513  */

514 volatile char in_bh = 0;   /* Non-reentrant remember */
```

变量 in_bh 是为了防止重复执行下半部分，即下半部分不允许重入。从 net_bh 实现的功能来看，重入很可能造成内核的不一致。

```
515 int in_net_bh()/* Used by timer.c */
516 {
517         return(in_bh==0?0:1);
518 }
```

```
519 /*
520  *  When we are called the queue is ready to grab, the interrupts are
521  *  on and hardware can interrupt and queue to the receive queue a we
522  *  run with no problems.
523  *  This is run as a bottom half after an interrupt handler that does
524  *  mark_bh(NET_BH);
525  */

526 void net_bh(void *tmp)
527 {
528     struct sk_buff *skb;
529     struct packet_type *ptype;
530     struct packet_type *pt_prev;
531     unsigned short type;

532     /*
533      *  Atomically check and mark our BUSY state.
534      */

535     if (set_bit(1, (void*)&in_bh))
536         return;
```

设置 in_bh 变量，如果该变量已经设置为 1，则表示下半部分已经在执行，此时立刻返回。即该函数不允许重入。

```
537     /*
538      *  Can we send anything now? We want to clear the
539      *  decks for any more sends that get done as we
540      *  process the input.
541      */

542     dev_transmit();
```

调用 dev_transmit 对硬件缓存队列进行处理，我们可以看到 net_bh 中前后 N 次（N 与接收数据包的个数 n 关联：N=n+2）调用该函数！可见 net_bh 是极力“希望”将所有的滞留数据包尽快地发送出去。

```
543     /*
544      *  Any data left to process. This may occur because a
545      *  mark_bh() is done after we empty the queue including
546      *  that from the device which does a mark_bh() just after
547      */
```

```
548     cli();

549     /*
550      *  While the queue is not empty
551      */

552     while((skb=skb_dequeue(&backlog))!=NULL)
553     {
```

这个 while 循环管到 630 行，对 backlog 队列中缓存的所有数据包进行处理。所谓处理：即对每个数据包，遍历 ptype_base 指向的网络层协议队列，检查匹配的协议，进而调用协议接收函数进行数据包的接收。

```
554             /*
555              *  We have a packet. Therefore the queue has shrunk
556              */
557             backlog_size--;

558             sti();

559                 /*
560             *   Bump the pointer to the next structure.
561             *   This assumes that the basic 'skb' pointer points to
562             *   the MAC header, if any (as indicated by its "length"
563             *   field).  Take care now!
564             */

565             skb->h.raw = skb->data + skb->dev->hard_header_len;
566             skb->len -= skb->dev->hard_header_len;

567                 /*
568             *   Fetch the packet protocol ID.  This is also quite ugly, as
569             *   it depends on the protocol driver (the interface itself) to
570             *   know what the type is, or where to get it from.  The Ethernet
571             *   interfaces fetch the ID from the two bytes in the Ethernet MAC
572             *   header (the h_proto field in struct ethhdr), but other drivers
573             *   may either use the ethernet ID's or extra ones that do not
574             *   clash (eg ETH_P_AX25). We could set this before we queue the
575             *   frame. In fact I may change this when I have time.
576             */

577             type = skb->dev->type_trans(skb, skb->dev);

578             /*
```

```
579             *   We got a packet ID.    Now loop over the "known protocols"
580             *   table (which is actually a linked list, but this will
581             *   change soon if I get my way- FvK), and forward the packet
582             *   to anyone who wants it.
583             *
584             *   [FvK didn't get his way but he is right this ought to be
585             *   hashed so we typically get a single hit. The speed cost
586             *   here is minimal but no doubt adds up at the 4,000+ pkts/second
587             *   rate we can hit flat out]
588             */
589            pt_prev = NULL;
590            for (ptype = ptype_base; ptype != NULL; ptype = ptype->next)
591            {
592                if ((ptype->type == type || ptype->type == htons(ETH_P_ALL)) &&
(!ptype->dev || ptype->dev==skb->dev))
593                {
594                    /*
595                     *   We already have a match queued. Deliver
596                     *   to it and then remember the new match
597                     */
598                    if(pt_prev)
599                    {
600                        struct sk_buff *skb2;

601                        skb2=skb_clone(skb, GFP_ATOMIC);

602                        /*
603                         *   Kick the protocol handler. This should be fast
604                         *   and efficient code.
605                         */

606                        if(skb2)
607                            pt_prev->func(skb2, skb->dev, pt_prev);
608                    }
609                    /* Remember the current last to do */
610                    pt_prev=ptype;
611                }
612            } /* End of protocol list loop */

613            /*
614             *   Is there a last item to send to ?
615             */

616            if(pt_prev)
```

```
617                 pt_prev->func(skb, skb->dev, pt_prev);
618             /*
619              *  Has an unknown packet has been received ?
620              */

621             else
622                 kfree_skb(skb, FREE_WRITE);

623             /*
624              *  Again, see if we can transmit anything now.
625              *  [Ought to take this out judging by tests it slows
626              *    us down not speeds us up]
627              */

628             dev_transmit();
629             cli();
630         }   /* End of queue loop */
```

577 行获取数据包使用网络层协议类型（对于以太网类型，skb->dev->type_trans 指向 eth_type_trans 函数，该函数返回链路层首部中的协议字段），在 592 行用于比较。注意在将数据把传递给上层之前，565，566 行对 sk_buff 结构中相关字段进行了更新，使得 h.raw 指向上一层（网络层）协议首部，len 字段值已除去本层（链路层）首部长度。在进行上层协议数据包接收函数的调用方式上，在 589 行引入了 pt_prev 变量，这样虽然 607 行是对前一个匹配协议项函数的调用，616 行避免了漏掉最后一个匹配项的可能性。另外 628 行对 dev_transmit 函数的调用让我们再次领略到系统发送数据包的决心。

```
631         /*
632          *  We have emptied the queue
633          */

634         in_bh = 0;
635         sti();

636         /*
637          *  One last output flush.
638          */

639         dev_transmit();
640 }
```

在处理完 backlog 队列中现有数据包后，我们退出循环，并结束此次下半部分的执行，634 行将 in_bh 设置为 0。639 行再次调用 dev_transmit 函数，我想该发送的应该都已经发送完了！

文件 dev.c 接下来定义的函数作为控制函数，作为用户配置命令的底层执行函数，本书前文中已经对各种数据结构进行了说明，而且至此已经完成所有常见协议实现模块的分析，所以对下面这些控制函数不再做分析讨论，简单列出代码如下，供读者自行理解。

```
682 /*
683  *    Perform a SIOCGIFCONF call. This structure will change
684  *    size shortly, and there is nothing I can do about it.
685  *    Thus we will need a 'compatibility mode'.
686  */

687 static int dev_ifconf(char *arg)
688 {
689         struct ifconf ifc;
690         struct ifreq ifr;
691         struct device *dev;
692         char *pos;
693         int len;
694         int err;

695         /*
696          *    Fetch the caller's info block.
697          */

698         err=verify_area(VERIFY_WRITE, arg, sizeof(struct ifconf));
699         if(err)
700                 return err;
701         memcpy_fromfs(&ifc, arg, sizeof(struct ifconf));
702         len = ifc.ifc_len;
703         pos = ifc.ifc_buf;

704         /*
705          *    We now walk the device list filling each active device
706          *    into the array.
707          */

708         err=verify_area(VERIFY_WRITE,pos,len);
709         if(err)
710                 return err;

711         /*
712          *    Loop over the interfaces, and write an info block for each.
713          */

714         for (dev = dev_base; dev != NULL; dev = dev->next)
715         {
```

```
716                     if(!(dev->flags & IFF_UP))    /* Downed devices don't count */
717                             continue;
718             memset(&ifr, 0, sizeof(struct ifreq));
719             strcpy(ifr.ifr_name, dev->name);
720             (*(struct sockaddr_in *) &ifr.ifr_addr).sin_family = dev->family;
721             (*(struct sockaddr_in *) &ifr.ifr_addr).sin_addr.s_addr = dev->pa_addr;

722             /*
723              *    Write this block to the caller's space.
724              */

725             memcpy_tofs(pos, &ifr, sizeof(struct ifreq));
726             pos += sizeof(struct ifreq);
727             len -= sizeof(struct ifreq);

728             /*
729              *    Have we run out of space here ?
730              */

731             if (len < sizeof(struct ifreq))
732                     break;
733     }

734     /*
735      *    All done.    Write the updated control block back to the caller.
736      */

737     ifc.ifc_len = (pos - ifc.ifc_buf);
738     ifc.ifc_req = (struct ifreq *) ifc.ifc_buf;
739     memcpy_tofs(arg, &ifc, sizeof(struct ifconf));

740     /*
741      *    Report how much was filled in
742      */

743     return(pos - arg);
744 }


745 /*
746  *    This is invoked by the /proc filesystem handler to display a device
747  *    in detail.
748  */
```

```
749  static int sprintf_stats(char *buffer, struct device *dev)
750  {
751          struct enet_statistics *stats = (dev->get_stats ? dev->get_stats(dev): NULL);
752          int size;

753          if (stats)
754                  size = sprintf(buffer, "%6s:%7d %4d %4d %4d %4d %8d %4d %4d %4d %5d
%4d\n",
755                          dev->name,
756                          stats->rx_packets, stats->rx_errors,
757                          stats->rx_dropped + stats->rx_missed_errors,
758                          stats->rx_fifo_errors,
759                          stats->rx_length_errors + stats->rx_over_errors
760                          + stats->rx_crc_errors + stats->rx_frame_errors,
761                          stats->tx_packets, stats->tx_errors, stats->tx_dropped,
762                          stats->tx_fifo_errors, stats->collisions,
763                          stats->tx_carrier_errors + stats->tx_aborted_errors
764                          + stats->tx_window_errors + stats->tx_heartbeat_errors);
765          else
766                  size = sprintf(buffer, "%6s: No statistics available.\n", dev->name);

767          return size;
768  }

769  /*
770   *   Called from the PROCfs module. This now uses the new arbitrary sized /proc/net
interface
771   *   to create /proc/net/dev
772   */

773  int dev_get_info(char *buffer, char **start, off_t offset, int length)
774  {
775          int len=0;
776          off_t begin=0;
777          off_t pos=0;
778          int size;

779          struct device *dev;


780          size = sprintf(buffer, "Inter-|   Receive                                |  Transmit\n"
781                                  " face |packets errs drop fifo frame|packets errs drop fifo colls carrier\n");

782          pos+=size;
```

```
783         len+=size;


784         for (dev = dev_base; dev != NULL; dev = dev->next)
785         {
786             size = sprintf_stats(buffer+len, dev);
787             len+=size;
788             pos=begin+len;

789             if(pos<offset)
790             {
791                 len=0;
792                 begin=pos;
793             }
794             if(pos>offset+length)
795                 break;
796         }

797         *start=buffer+(offset-begin);  /* Start of wanted data */
798         len-=(offset-begin);        /* Start slop */
799         if(len>length)
800             len=length;             /* Ending slop */
801         return len;
802 }


803 /*
804  *    This checks bitmasks for the ioctl calls for devices.
805  */

806 static inline int bad_mask(unsigned long mask, unsigned long addr)
807 {
808         if (addr & (mask = ~mask))
809             return 1;
810         mask = ntohl(mask);
811         if (mask & (mask+1))
812             return 1;
813         return 0;
814 }
```

bad_mask 函数检查掩码的合法性。

```
815 /*
816  *    Perform the SIOCxIFxxx calls.
```

```
817  *
818  *    The socket layer has seen an ioctl the address family thinks is
819  *    for the device. At this point we get invoked to make a decision
820  */

821  static int dev_ifsioc(void *arg, unsigned int getset)
822  {
823          struct ifreq ifr;
824          struct device *dev;
825          int ret;

826          /*
827           *    Fetch the caller's info block into kernel space
828           */

829          int err=verify_area(VERIFY_WRITE, arg, sizeof(struct ifreq));
830          if(err)
831                  return err;

832          memcpy_fromfs(&ifr, arg, sizeof(struct ifreq));

833          /*
834           *    See which interface the caller is talking about.
835           */

836          if ((dev = dev_get(ifr.ifr_name)) == NULL)
837                  return(-ENODEV);

838          switch(getset)
839          {
840                  case SIOCGIFFLAGS:   /* Get interface flags */
841                          ifr.ifr_flags = dev->flags;
842                          memcpy_tofs(arg, &ifr, sizeof(struct ifreq));
843                          ret = 0;
844                          break;
845                  case SIOCSIFFLAGS:   /* Set interface flags */
846                          {
847                                  int old_flags = dev->flags;
848  #ifdef CONFIG_SLAVE_BALANCING
849                                  if(dev->flags&IFF_SLAVE)
850                                          return -EBUSY;
851  #endif
852                                  dev->flags = ifr.ifr_flags & (
853                                          IFF_UP | IFF_BROADCAST | IFF_DEBUG | IFF_LOOPBACK |
```

```
854                             IFF_POINTOPOINT | IFF_NOTRAILERS | IFF_RUNNING |
855                             IFF_NOARP | IFF_PROMISC | IFF_ALLMULTI | IFF_SLAVE |
IFF_MASTER
856                             | IFF_MULTICAST);
857  #ifdef CONFIG_SLAVE_BALANCING
858                         if(!(dev->flags&IFF_MASTER) && dev->slave)
859                         {
860                             dev->slave->flags&=~IFF_SLAVE;
861                             dev->slave=NULL;
862                         }
863  #endif
864                         /*
865                          *    Load in the correct multicast list now the flags have changed.
866                          */

867                         dev_mc_upload(dev);
868  #if 0
869                         if( dev->set_multicast_list!=NULL)
870                         {

871                             /*
872                              *    Has promiscuous mode been turned off
873                              */

874                             if ( (old_flags & IFF_PROMISC) && ((dev->flags &
IFF_PROMISC) == 0))
875                                 dev->set_multicast_list(dev,0,NULL);

876                             /*
877                              *    Has it been turned on
878                              */

879                     if ( (dev->flags & IFF_PROMISC) && ((old_flags & IFF_PROMISC) == 0))
880                                 dev->set_multicast_list(dev,-1,NULL);
881                         }
882  #endif
883                         /*
884                          *    Have we downed the interface
885                          */

886                         if ((old_flags & IFF_UP) && ((dev->flags & IFF_UP) == 0))
887                         {
888                             ret = dev_close(dev);
889                         }
```

```
890                     else
891                     {
892                         /*
893                          *    Have we upped the interface
894                          */

895                         ret = (! (old_flags & IFF_UP) && (dev->flags & IFF_UP))
896                         ? dev_open(dev) : 0;
897                         /*
898                          *    Check the flags.
899                          */
900                         if(ret<0)
901                             dev->flags&=~IFF_UP; /* Didn't open so down the if */
902                     }
903                     }
904                 break;

905             case SIOCGIFADDR:    /* Get interface address (and family) */
906                 (*(struct sockaddr_in *)
907                         &ifr.ifr_addr).sin_addr.s_addr = dev->pa_addr;
908                 (*(struct sockaddr_in *)
909                         &ifr.ifr_addr).sin_family = dev->family;
910                 (*(struct sockaddr_in *)
911                         &ifr.ifr_addr).sin_port = 0;
912                 memcpy_tofs(arg, &ifr, sizeof(struct ifreq));
913                 ret = 0;
914                 break;

915             case SIOCSIFADDR:    /* Set interface address (and family) */
916                 dev->pa_addr = (*(struct sockaddr_in *)
917                         &ifr.ifr_addr).sin_addr.s_addr;
918                 dev->family = ifr.ifr_addr.sa_family;

919 #ifdef CONFIG_INET
920                 /* This is naughty. When net-032e comes out It wants moving into the net032
921                    code not the kernel. Till then it can sit here (SIGH) */
922                 dev->pa_mask = ip_get_mask(dev->pa_addr);
923 #endif
924                 dev->pa_brdaddr = dev->pa_addr | ~dev->pa_mask;
925                 ret = 0;
926                 break;

927             case SIOCGIFBRDADDR:    /* Get the broadcast address */
928                 (*(struct sockaddr_in *)
```

```
929                     &ifr.ifr_broadaddr).sin_addr.s_addr = dev->pa_brdaddr;
930             (*(struct sockaddr_in *)
931                     &ifr.ifr_broadaddr).sin_family = dev->family;
932             (*(struct sockaddr_in *)
933                     &ifr.ifr_broadaddr).sin_port = 0;
934             memcpy_tofs(arg, &ifr, sizeof(struct ifreq));
935             ret = 0;
936             break;

937         case SIOCSIFBRDADDR:    /* Set the broadcast address */
938             dev->pa_brdaddr = (*(struct sockaddr_in *)
939                     &ifr.ifr_broadaddr).sin_addr.s_addr;
940             ret = 0;
941             break;

942         case SIOCGIFDSTADDR: /*Get the destination address (for point-to-point links) */
943             (*(struct sockaddr_in *)
944                     &ifr.ifr_dstaddr).sin_addr.s_addr = dev->pa_dstaddr;
945             (*(struct sockaddr_in *)
946                     &ifr.ifr_broadaddr).sin_family = dev->family;
947             (*(struct sockaddr_in *)
948                     &ifr.ifr_broadaddr).sin_port = 0;
949                     memcpy_tofs(arg, &ifr, sizeof(struct ifreq));
950             ret = 0;
951             break;

952         case SIOCSIFDSTADDR:  /* Set the destination address (for point-to-point links) */
953             dev->pa_dstaddr = (*(struct sockaddr_in *)
954                     &ifr.ifr_dstaddr).sin_addr.s_addr;
955             ret = 0;
956             break;

957         case SIOCGIFNETMASK:    /* Get the netmask for the interface */
958             (*(struct sockaddr_in *)
959                     &ifr.ifr_netmask).sin_addr.s_addr = dev->pa_mask;
960             (*(struct sockaddr_in *)
961                     &ifr.ifr_netmask).sin_family = dev->family;
962             (*(struct sockaddr_in *)
963                     &ifr.ifr_netmask).sin_port = 0;
964             memcpy_tofs(arg, &ifr, sizeof(struct ifreq));
965             ret = 0;
966             break;

967         case SIOCSIFNETMASK:    /* Set the netmask for the interface */
```

```
968                 {
969                     unsigned long mask = (*(struct sockaddr_in *)
970                         &ifr.ifr_netmask).sin_addr.s_addr;
971                     ret = -EINVAL;
972                     /*
973                      *    The mask we set must be legal.
974                      */
975                     if (bad_mask(mask,0))
976                         break;
977                     dev->pa_mask = mask;
978                     ret = 0;
979                 }
980                 break;

981             case SIOCGIFMETRIC:/* Get the metric on the interface (currently unused) */

982                 ifr.ifr_metric = dev->metric;
983                 memcpy_tofs(arg, &ifr, sizeof(struct ifreq));
984                 ret = 0;
985                 break;

986             case SIOCSIFMETRIC: /* Set the metric on the interface (currently unused) */
987                 dev->metric = ifr.ifr_metric;
988                 ret = 0;
989                 break;

990             case SIOCGIFMTU:    /* Get the MTU of a device */
991                 ifr.ifr_mtu = dev->mtu;
992                 memcpy_tofs(arg, &ifr, sizeof(struct ifreq));
993                 ret = 0;
994                 break;

995             case SIOCSIFMTU:    /* Set the MTU of a device */

996                 /*
997                  *    MTU must be positive and under the page size problem
998                  */

999                 if(ifr.ifr_mtu<1 || ifr.ifr_mtu>3800)
1000                     return -EINVAL;
1001                 dev->mtu = ifr.ifr_mtu;
1002                 ret = 0;
1003                 break;
```

```
1004        case SIOCGIFMEM:    /* Get the per device memory space. We can add this but currently
1005                                do not support it */
1006            printk("NET: ioctl(SIOCGIFMEM, %p)\n", arg);
1007            ret = -EINVAL;
1008            break;

1009        case SIOCSIFMEM:    /* Set the per device memory buffer space. Not applicable in our case */
1010            printk("NET: ioctl(SIOCSIFMEM, %p)\n", arg);
1011        ret = -EINVAL;
1012            break;

1013        case OLD_SIOCGIFHWADDR:  /* Get the hardware address. This will change and SIFHWADDR will be added */
1014            memcpy(ifr.old_ifr_hwaddr,dev->dev_addr, MAX_ADDR_LEN);
1015            memcpy_tofs(arg,&ifr,sizeof(struct ifreq));
1016            ret=0;
1017            break;

1018        case SIOCGIFHWADDR:
1019            memcpy(ifr.ifr_hwaddr.sa_data,dev->dev_addr, MAX_ADDR_LEN);
1020            ifr.ifr_hwaddr.sa_family=dev->type;
1021            memcpy_tofs(arg,&ifr,sizeof(struct ifreq));
1022            ret=0;
1023            break;

1024        case SIOCSIFHWADDR:
1025            if(dev->set_mac_address==NULL)
1026                return -EOPNOTSUPP;
1027            if(ifr.ifr_hwaddr.sa_family!=dev->type)
1028                return -EINVAL;
1029            ret=dev->set_mac_address(dev,ifr.ifr_hwaddr.sa_data);
1030            break;

1031        case SIOCGIFMAP:
1032            ifr.ifr_map.mem_start=dev->mem_start;
1033            ifr.ifr_map.mem_end=dev->mem_end;
1034            ifr.ifr_map.base_addr=dev->base_addr;
1035            ifr.ifr_map.irq=dev->irq;
1036            ifr.ifr_map.dma=dev->dma;
1037            ifr.ifr_map.port=dev->if_port;
1038            memcpy_tofs(arg,&ifr,sizeof(struct ifreq));
1039            ret=0;
```

```
1040                    break;

1041            case SIOCSIFMAP:
1042                    if(dev->set_config==NULL)
1043                            return -EOPNOTSUPP;
1044                    return dev->set_config(dev,&ifr.ifr_map);

1045            case SIOCGIFSLAVE:
1046    #ifdef CONFIG_SLAVE_BALANCING
1047                    if(dev->slave==NULL)
1048                            return -ENOENT;
1049                    strncpy(ifr.ifr_name,dev->name,sizeof(ifr.ifr_name));
1050                    memcpy_tofs(arg,&ifr,sizeof(struct ifreq));
1051                    ret=0;
1052    #else
1053                    return -ENOENT;
1054    #endif
1055                    break;
1056    #ifdef CONFIG_SLAVE_BALANCING
1057            case SIOCSIFSLAVE:
1058            {

1059            /*
1060             *      Fun game. Get the device up and the flags right without
1061             *      letting some scummy user confuse us.
1062             */
1063                    unsigned long flags;
1064                    struct device *slave=dev_get(ifr.ifr_slave);
1065                    save_flags(flags);
1066                    if(slave==NULL)
1067                    {
1068                            return -ENODEV;
1069                    }
1070                    cli();
1071
    if((slave->flags&(IFF_UP|IFF_RUNNING))!=(IFF_UP|IFF_RUNNING))
1072                    {
1073                            restore_flags(flags);
1074                            return -EINVAL;
1075                    }
1076                    if(dev->flags&IFF_SLAVE)
1077                    {
1078                            restore_flags(flags);
1079                            return -EBUSY;
```

```
1080                    }
1081                    if(dev->slave!=NULL)
1082                    {
1083                        restore_flags(flags);
1084                        return -EBUSY;
1085                    }
1086                    if(slave->flags&IFF_SLAVE)
1087                    {
1088                        restore_flags(flags);
1089                        return -EBUSY;
1090                    }
1091                    dev->slave=slave;
1092                    slave->flags|=IFF_SLAVE;
1093                    dev->flags|=IFF_MASTER;
1094                    restore_flags(flags);
1095                    ret=0;
1096                }
1097                break;
1098    #endif

1099                case SIOCADDMULTI:
1100                if(dev->set_multicast_list==NULL)
1101                    return -EINVAL;
1102                if(ifr.ifr_hwaddr.sa_family!=AF_UNSPEC)
1103                    return -EINVAL;
1104                dev_mc_add(dev,ifr.ifr_hwaddr.sa_data, dev->addr_len, 1);
1105                return 0;

1106            case SIOCDELMULTI:
1107                if(dev->set_multicast_list==NULL)
1108                    return -EINVAL;
1109                if(ifr.ifr_hwaddr.sa_family!=AF_UNSPEC)
1110                    return -EINVAL;
1111                dev_mc_delete(dev,ifr.ifr_hwaddr.sa_data,dev->addr_len, 1);
1112                return 0;
1113            /*
1114             *    Unknown or private ioctl
1115             */

1116            default:
1117                if((getset >= SIOCDEVPRIVATE) &&
1118                   (getset <= (SIOCDEVPRIVATE + 15))) {
1119                    if(dev->do_ioctl==NULL)
1120                        return -EOPNOTSUPP;
```

```
1121                    ret=dev->do_ioctl(dev, &ifr, getset);
1122                    memcpy_tofs(arg,&ifr,sizeof(struct ifreq));
1123                    break;
1124            }

1125            ret = -EINVAL;
1126    }
1127    return(ret);
1128}



1129/*
1130 *  This function handles all "interface"-type I/O control requests. The actual
1131 *  'doing' part of this is dev_ifsioc above.
1132 */

1133int dev_ioctl(unsigned int cmd, void *arg)
1134{
1135    switch(cmd)
1136    {
1137        case SIOCGIFCONF:
1138            (void) dev_ifconf((char *) arg);
1139            return 0;

1140        /*
1141         *  Ioctl calls that can be done by all.
1142         */

1143        case SIOCGIFFLAGS:
1144        case SIOCGIFADDR:
1145        case SIOCGIFDSTADDR:
1146        case SIOCGIFBRDADDR:
1147        case SIOCGIFNETMASK:
1148        case SIOCGIFMETRIC:
1149        case SIOCGIFMTU:
1150        case SIOCGIFMEM:
1151        case SIOCGIFHWADDR:
1152        case SIOCSIFHWADDR:
1153        case OLD_SIOCGIFHWADDR:
1154        case SIOCGIFSLAVE:
1155        case SIOCGIFMAP:
1156            return dev_ifsioc(arg, cmd);

1157        /*
```

```
1158            *    Ioctl calls requiring the power of a superuser
1159            */

1160            case SIOCSIFFLAGS:
1161            case SIOCSIFADDR:
1162            case SIOCSIFDSTADDR:
1163            case SIOCSIFBRDADDR:
1164            case SIOCSIFNETMASK:
1165            case SIOCSIFMETRIC:
1166            case SIOCSIFMTU:
1167            case SIOCSIFMEM:
1168            case SIOCSIFMAP:
1169            case SIOCSIFSLAVE:
1170            case SIOCADDMULTI:
1171            case SIOCDELMULTI:
1172                 if (!suser())
1173                      return -EPERM;
1174                 return dev_ifsioc(arg, cmd);

1175            case SIOCSIFLINK:
1176                 return -EINVAL;

1177            /*
1178             *    Unknown or private ioctl.
1179             */

1180            default:
1181                 if((cmd >= SIOCDEVPRIVATE) &&
1182                    (cmd <= (SIOCDEVPRIVATE + 15))) {
1183                      return dev_ifsioc(arg, cmd);
1184                 }
1185                 return -EINVAL;
1186       }
1187}
```

以上这些函数的理解关键在于弄清各函数使用数据结构的定义，此后就是对现有结构字段值的获取和设置。此处不再赘述。

```
1188/*
1189 *    Initialize the DEV module. At boot time this walks the device list and
1190 *    unhooks any devices that fail to initialise (normally hardware not
1191 *    present) and leaves us with a valid list of present and active devices.
1192 *
1193 *    The PCMCIA code may need to change this a little, and add a pair
```

```
1194 *    of register_inet_device() unregister_inet_device() calls. This will be
1195 *    needed for ethernet as modules support.
1196 */

1197void dev_init(void)
1198{
1199      struct device *dev, *dev2;

1200          /*
1201           *    Add the devices.
1202           *    If the call to dev->init fails, the dev is removed
1203           *    from the chain disconnecting the device until the
1204           *    next reboot.
1205           */

1206          dev2 = NULL;
1207          for (dev = dev_base; dev != NULL; dev=dev->next)
1208          {
1209              if (dev->init && dev->init(dev))
1210              {
1211              /*
1212                   *    It failed to come up. Unhook it.
1213                   */

1214                  if (dev2 == NULL)
1215                      dev_base = dev->next;
1216                  else
1217                      dev2->next = dev->next;
1218              }
1219              else
1220              {
1221                  dev2 = dev;
1222              }
1223          }
1224      }
```

dev_init 函数是 dev.c 文件中定义的最后一个函数，在内核网络栈初始化过程中被调用对系统中存在的所有网络设备进行初始化操作，诚如前文所述，由于各设备的不同，初始化代码也将不同，实现上由具体硬件驱动函数进行初始化，即调用 device 结构中 init 字段指向的函数完成各自特定硬件设备的初始化工作。注意如果设备初始化失败，该设备将从系统设备列表中（由 dev_base 指向）删除。

dev.c 文件小结

该文件作为链路层模块实现文件存在，其跨接在驱动层和网络层之间完成数据包两个方向的

传递。该层实现功能较为简单：
1>向驱动层提供接收函数接口，供驱动层调用将接收到的数据包传递给系统。
2>向网络层提供发送函数接口，供网络层调用发送数据包。
3>提供数据包缓存功能，维护驱动层和网络层（主要是驱动层）的正常工作。
4>提供硬件控制接口，用于控制具体硬件的状态。

至此我们完成基本所有协议实现文件的分析，还遗留有：
1> ipx.c，ipx.h，ipxcall.h：这三个文件是关于 IPX 协议的实现文件，因为并非广芳使用，本书不对此进行分析，感兴趣的读者可自行进行分析和理解。
2> rarp.c，rarp.h：RARP 协议实现文件，因为与 ARP 协议实现代码非常类似，而且较为容易理解，本书对此也不做分析。
3> sk_buff.c：sk_buff 结构操作功能实现文件，这个文件专门定义了一系列函数对 sk_buff 结构进行操作，包括 sk_buff 结构的分配，常用字段的初始化，挂入或从队列中删除等等。由于代码实现非常简单，不存在理解上的困难，本书也放弃对该文件的分析。
4> pe2.c：该文件实现类同 psnap.c, p8022.c，而且该文件中定义的函数并没有在其他部分被调用，故没有分析的必要，且实现简单，故本书也不做讨论。

2.40 本章小结

在本书第二部分，我们完成了 Linux-1.2.13 内核网络栈协议实现代码的全部分析，采用了自上而下的分析方式，从应用层（net/socket.c），到 INET 层（net/inet/af_inet.c），再到传输层（tcp.c, udp.c, raw.c, packet.c），再到网络层（ip.c），最后到链路层（eth.c, p8022.c, p8023.c, psnap.c, dev.c）以及其他一系列功能实现函数文件（如 sock.c, datagram.c, devinit.c 等）。理解网络栈实现代码最关键的一点是首先要从总体上进行把握，首先弄清各层对应的实现文件，此后针对各文件查看函数调用关系。在分析和理解中，我们最好能以发送一个数据包为线，从最上层循其传递路径到达最下层，可以了解网络栈的基本调用关系，从而方便自己的理解。虽然本书在此部分中对各文件进行了一一分析，但没有特别交待层次关系，对于网络栈实现代码各文件之间的关系以及各函数之间的调用关系需要读者能够在理解的基础上自己串联起来。为了便于读者的理解，本书第三部分，将着重讨论网络栈实现代码中的层次关系。

第三章 网络设备驱动程序分析

本章分析文件均分布在 drivers/net 目录下，我们不再继续详细分析其中每个文件，而是着重于网络设备初始化和网卡驱动架构的分析，以及内核其他一些初始化关键代码的分析。Linux-1.2.13 版本内核网络栈实现中并未做到很好的驱动层和网络栈实现层之间的隔离，内核一些关键代码仍然散布于驱动层中，在本章中，我们将以代码片段为单位进行分析，不再遵循第二章中对整个文件的分析方式，并且更为关注各函数之间的调用关系，以及一个网卡设备如何被初始化并进入正常工作状态。具体地，我们将关注于如下几个文件：
Space.c：系统网络设备列表 dev_base 变量定义。
net_init.c：支持设备动态注册，注销的函数定义，以及以太网设备初始化函数定义。
ne.c, 8390.c：据此分析网络设备初始化过程。

drivers/net 目录下还定义有其他很多函数，如 PPP 协议实现文件 ppp.c, 回环接口驱动实现文件 loopback.c 等等，本书不再作分析，相信经过本书之前的积累，对这些代码的理解应不存在问题。故留作感兴趣读者自行分析。

3.1 关键变量，函数定义及网络设备驱动初始化

首先我们查看系统网络设备列表的定义，即 dev_base 全局变量的定义，这个变量定义在 Space.c 中，如下代码片段所示：

```
/*drivers/net/Space.c*/
64   static int
65   ethif_probe(struct device *dev)
66   {
67        short base_addr = dev->base_addr;

68        if (base_addr < 0    ||    base_addr == 1)
69             return 1;         /* ENXIO */
......
80   #if defined(CONFIG_NE2000) || defined(NE2000)
81        && ne_probe(dev)
82   #endif
......

152       return 0;
153  }

178  static struct device eth3_dev = {
179       "eth3", 0,0,0,0,0xffe0 /* I/O base*/, 0,0,0,0, NEXT_DEV, ethif_probe };
180  static struct device eth2_dev = {
181       "eth2", 0,0,0,0,0xffe0 /* I/O base*/, 0,0,0,0, &eth3_dev, ethif_probe };
182  static struct device eth1_dev = {
183       "eth1", 0,0,0,0,0xffe0 /* I/O base*/, 0,0,0,0, &eth2_dev, ethif_probe };
```

```
184  static struct device eth0_dev = {
185       "eth0", 0, 0, 0, 0, ETH0_ADDR, ETH0_IRQ, 0, 0, 0, &eth1_dev, ethif_probe };

186  #    undef NEXT_DEV
187  #    define NEXT_DEV(&eth0_dev)
.....

288  extern int loopback_init(struct device *dev);
289  struct device loopback_dev = {
290       "lo",          /* Software Loopback interface          */
291       0x0,           /* recv memory end                      */
292       0x0,           /* recv memory start                    */
293       0x0,           /* memory end                           */
294       0x0,           /* memory start                         */
295       0,             /* base I/O address             */
296       0,             /* IRQ                          */
297       0, 0, 0,       /* flags                  */
298       NEXT_DEV,      /* next device                          */
299       loopback_init  /* loopback_init should set up the rest   */
300  };

301  struct device *dev_base = &loopback_dev;
```

此处只给出了 loopback 回环设备的 device 结构定义，其中 NEXT_DEV 宏定义表示其下一个指向的网络设备 device 结构，我们此处省略，读者可查看 Space.c 文件完整定义。系统维护一个所有设备的列表目的在于可以方便的进行设备查询，以获知当前系统中可用设备，或者根据设备名称查询特定的设备，从而对其进行相关操作。

另外值得注意的是对于以上定义的几个以太网设备其 device 结构中 init 指针均指向了 ethif_probe 函数，该函数同样定义在 Space.c 中，为了和 ne.c 文件中代码联系起来，我们同样给出了 ethif_probe 函数的部分代码。

net_init.c 文件中定义有两个重要函数用于支持网络设备驱动模块的动态加载：即 register_netdev 和 unregister_netdev。

register_netdev 函数用于注册一个网络设备，注册的含义即将设备对应的 device 结构加入到 dev_base 变量指向的系统设备列表中，其次调用设备特定的初始化函数（dev->init）对该设备进行初始化使其进入预知状态。函数实现较为浅显，我们给出其代码，但不做分析。

```
/*drivers/net/net_init.c*/
189  int register_netdev(struct device *dev)
190  {
191       struct device *d = dev_base;
192       unsigned long flags;
```

```
193     int i=MAX_ETH_CARDS; // MAX_ETH_CARDS 定义在相同文件中，值为 16。

194     save_flags(flags);
195     cli();

196     if (dev && dev->init) {
197         if (dev->name &&
198             ((dev->name[0] == '\0') || (dev->name[0] == ' '))) {
199             for (i = 0; i < MAX_ETH_CARDS; ++i)
200                 if (ethdev_index[i] == NULL) {
201                     sprintf(dev->name, "eth%d", i);
202                     printk("loading device '%s'...\n", dev->name);
203                     ethdev_index[i] = dev;
204                     break;
205                 }
206         }

207         if (dev->init(dev) != 0) {
208             if (i < MAX_ETH_CARDS) ethdev_index[i] = NULL;
209             restore_flags(flags);
210             return -EIO;
211         }

212         /* Add device to end of chain */
213         if (dev_base) {
214             while (d->next)
215                 d = d->next;
216             d->next = dev;
217         }
218         else
219             dev_base = dev;
220         dev->next = NULL;
221     }
222     restore_flags(flags);
223     return 0;
224 }
```

unregister_netdev 函数负责注销一个网络设备，注销的函数即将设备从系统设备列表中删除，注意在调用 unregister_netdev 函数注销一个设备之前，必须先明确关闭该设备，使其处于非工作状态，否则将无法对设备进行注销操作。unregister_netdev 函数实现代码如下。

```
225 void unregister_netdev(struct device *dev)
226 {
227     struct device *d = dev_base;
```

```
228     unsigned long flags;
229     int i;

230     save_flags(flags);
231     cli();

232     printk("unregister_netdev: device ");

233     if (dev == NULL) {
234         printk("was NULL\n");
235         restore_flags(flags);
236         return;
237     }
238     /* else */
239     if (dev->start)
240         printk("'%s' busy\n", dev->name);
241     else {
242         if (dev_base == dev)
243             dev_base = dev->next;
244         else {
245             while (d && (d->next != dev))
246                 d = d->next;

247             if (d && (d->next == dev)) {
248                 d->next = dev->next;
249                 printk("'%s' unlinked\n", dev->name);
250             }
251             else {
252                 printk("'%s' not found\n", dev->name);
253                 restore_flags(flags);
254                 return;
255             }
256         }
257         for (i = 0; i < MAX_ETH_CARDS; ++i) {
258             if (ethdev_index[i] == dev) {
259                 ethdev_index[i] = NULL;
260                 break;
261             }
262         }
263     }
264     restore_flags(flags);
265 }
```

下面我们以 ne.c，8390.c 两个文件为例介绍 NE 系列网卡驱动程序模块的初始化过程，并着

重阐述网卡驱动程序结构组织。为了避免和本书第三部分内容造成重复，此处我们只介绍网络设备驱动程序的初始化过程，整个系统网络栈的初始化将在第三部分进行阐述，所以此处我们就从 dev_init 函数（定义于 dev.c 文件中）出发，该函数作为所有网络设备驱动程序模块初始化工作的总入口函数，将在系统网络栈初始化过程中被调用。

为便于此处的分析，我们重新给出 dev_init 函数实现代码，如下；

```
/*net/inet/dev.c—dev_init 函数*/
1197void dev_init(void)
1198{
1199     struct device *dev, *dev2;

1200          /*
1201           *    Add the devices.
1202           *    If the call to dev->init fails, the dev is removed
1203           *    from the chain disconnecting the device until the
1204           *    next reboot.
1205           */

1206          dev2 = NULL;
1207          for (dev = dev_base; dev != NULL; dev=dev->next)
1208          {
1209               if (dev->init && dev->init(dev))
1210               {
1211               /*
1212                    *    It failed to come up. Unhook it.
1213                    */

1214                    if (dev2 == NULL)
1215                         dev_base = dev->next;
1216                    else
1217                         dev2->next = dev->next;
1218               }
1219               else
1220               {
1221                    dev2 = dev;
1222               }
1223          }
1224     }
```

dev_init 函数遍历由 dev_base 变量指向的系统网络设备列表，对其中每个设备调用其 init 函数进行初始化，我们在本章前文中给出了 dev_base 变量的定义，并给出了部分网络设备的 device 结构定义形式，此处我们以 eth 接口为例进行说明，其注册的 init 函数为 ethif_probe 函数，ethif_probe 函数的部分实现代码如本章前文所示，我们进入该函数执行，进入到 ne_probe 函数。

ne_probe 函数定义在 ne.c 中，其对各个地址区域进行扫描，探测网络设备的存在，最终的探测工作由 ne_probe1 函数完成：

```
/*drivers/net/ne.c –ne_probe1 函数代码片段*/
113  static int ne_probe1(struct device *dev, int ioaddr)
114  {
......
/*在 ioaddr 地址处进行物理设备的探测，即读写相关地址处的值，对返回值进行检查*/
/*以下 121-123 行代码对应无对应物理设备的情况*/
121        int reg0 = inb_p(ioaddr);
122        if (reg0 == 0xFF) //返回值为 0xFF，表示该地址处不存在可读的寄存器
123             return ENODEV;
......

/*经过以上的一系列检查，我们在 ioaddr 处发现有一网络设备存在*/
235        dev->irq = 9; /*此处采用硬编码方式进行中断号的赋值*/

236        /* Snarf the interrupt now.    There's no point in waiting since we cannot
237             share and the board will usually be enabled. */
/*调用 request_irq 注册中断处理函数为 ei_interrupt ， 该函数将负责接收和发送中断*/
238        {
239        int irqval = request_irq (dev->irq, ei_interrupt, 0, wordlength==2 ? "ne2000":"ne1000");
240        if (irqval) {
241             printk (" unable to get IRQ %d (irqval=%d).\n", dev->irq, irqval);
242             return EAGAIN;
243        }
244        }

245        dev->base_addr = ioaddr;
/*将 ioaddr 开始的一段空间申请为己有*/
246        request_region(ioaddr, NE_IO_EXTENT, wordlength==2 ? "ne2000":"ne1000");
/*将从设备相关硬件区域中读取的 MAC 地址存储到 device 结构相关字段中*/
247        for(i = 0; i < ETHER_ADDR_LEN; i++)
248             dev->dev_addr[i] = SA_prom[i];
/*调用 ethdev_init 函数进行 device 结构字段的初始化*/
249        ethdev_init(dev);
250        printk("\n%s: %s found at %#x, using IRQ %d.\n",
251             dev->name, name, ioaddr, dev->irq);
......
266        NS8390_init(dev, 0); /*配置 DP8390 到预知状态*/
267        return 0; //返回 0 表示成功探测到设备
268  }
```

ne_probe1 函数中最为重要的一个函数为 ethdev_init 函数，该函数定义在 8390.c 文件中，具体实现如下：

```
/*drivers/net/8390.c--ethdev_init 函数*/
512  /* Initialize the rest of the 8390 device structure. */
513  int ethdev_init(struct device *dev)
514  {
/*分配私有数据结构，保存一些关键信息，便于驱动程序本身的操作，省去此段代码*/
......

526         /* The open call may be overridden by the card-specific code. */
527         if (dev->open == NULL)
528              dev->open = &ei_open;

529         /* We should have a dev->stop entry also. */
530         dev->hard_start_xmit = &ei_start_xmit;
531         dev->get_stats = get_stats;

532  #ifdef HAVE_MULTICAST
533         dev->set_multicast_list = &set_multicast_list;
534  #endif

535         ether_setup(dev);

536         return 0;
537  }
```

527-530 行代码至关重要，其初始化了 device 结构中非常重要的两个函数指针：open 和 hard_start_xmit 函数指针。open 指向的函数用于打开设备，即配置设备到正常工作状态；而 hard_start_xmit 指向的函数则被 dev_queue_xmit 函数调用操作具体的硬件将数据发送到物理介质上去。此外 535 行继续调用能够 ether_setup 函数对 device 结构中其他一些字段进行初始化操作，我们继续跟踪 ether_setup 函数，其定义在 net_init.c 文件中，如下：

```
/*drivers/net/net_init.c --ether_setup 函数*/
136  void ether_setup(struct device *dev)
137  {
138         int i;
139         /* Fill in the fields of the device structure with ethernet-generic values.
140              This should be in a common file instead of per-driver.    */
141         for (i = 0; i < DEV_NUMBUFFS; i++)
142              skb_queue_head_init(&dev->buffs[i]);

143         /* register boot-defined "eth" devices */
144         if (dev->name && (strncmp(dev->name, "eth", 3) == 0)) {
145              i = simple_strtoul(dev->name + 3, NULL, 0);
146              if (ethdev_index[i] == NULL) {
147                   ethdev_index[i] = dev;
148              }
```

```
149             else if (dev != ethdev_index[i]) {
150                     /* Really shouldn't happen! */
151                     printk("ether_setup: Ouch! Someone else took %s\n",
152                             dev->name);
153             }
154     }

155     dev->hard_header   = eth_header;
156     dev->rebuild_header = eth_rebuild_header;
157     dev->type_trans = eth_type_trans;

158     dev->type               = ARPHRD_ETHER;
159     dev->hard_header_len = ETH_HLEN;
160     dev->mtu        = 1500; /* eth_mtu */
161     dev->addr_len = ETH_ALEN;
162     for (i = 0; i < ETH_ALEN; i++) {
163             dev->broadcast[i]=0xff;
164     }

165     /* New-style flags. */
166     dev->flags              = IFF_BROADCAST|IFF_MULTICAST;
167     dev->family             = AF_INET;
168     dev->pa_addr  = 0;
169     dev->pa_brdaddr = 0;
170     dev->pa_mask = 0;
171     dev->pa_alen  = sizeof(unsigned long);
172 }
```

ether_setup 函数实现代码我想已不用多做解释，注意 155-157 行对相关函数指针的初始化。hard_header 指向的函数被调用创建链路层首部；rebuild_header 指向的函数在数据包发送之前被调用对链路层首部中目的端硬件地址字段进行赋值，这有可能启动 ARP 地址解析过程。这两个指针所指向的函数在本书第二章中都有介绍，此处不再赘述。

至此我们完全完成对应网络设备的初始化工作，现在即等系统启动设备进入正常工作状态。启动工作由 dev_open(dev.c)函数完成，该函数调用具体硬件对应的 open 函数配置设备进入正常工作状态，如在 ethdev_init 函数中 NE 系列设备 open 指针指向 ei_open 函数，ei_open 函数将配置 NE 网卡进入工作状态。同样在 ethdev_init 函数中其发送函数注册为 ei_start_xmit，从而我们就有了发送通道，当网络栈上层调用 dev_queue_xmit 函数发送一个数据包时，其调用 ei_start_xmit 函数操作具体硬件将数据发送出去。那么设备又是如何接收数据的？这就要根据硬件设备本身的工作方式来进行，但一般的工作方式都是由中断驱动，当设备硬件接收到一个数据包时（这个接收过程完全由硬件负责，这一点读者无需担心）就会触发一个接收中断，内核将调用我们之前的注册的中断处理程序进行处理，如前文中介绍 ne_probe1 函数时调用 request_irq 注册的 ei_interrupt 中断处理函数，那么当硬件触发一个中断时，系统将调用 ei_interrupt 函数执行。ei_interrupt 函数的作用可想而知，其读取硬件有

关中断状态寄存器，判断此次中断的来源，当发现这是一个接收中断，其将申请一段系统缓存区，将数据从设备硬件缓存区中拷贝到系统缓冲区中，以 sk_buff 结构进行封装后，调用 netif_rx 函数将数据传送给链路层模块进行处理，从而完成数据包在驱动层的接收工作。到了 netif_rx 函数，那么就和我们第二章中介绍的内容接上了，netif_rx 将数据包缓存到 backlog 队列中，此后 net_bh 函数将数据包传递给网络层协议接收函数如 arp_rcv, ip_rcv 等进行处理，以此向上传递，最终到达用户程序。

3.2 网络设备驱动程序结构小结

综上所述，我们可以总结网络设备驱动程序结构如下：

无论是动态模块形式，还是静态编入内核，首先将由内核调用我们注册的初始化函数，在初始化函数中我们必须完成 device 结构的创建（对于静态编入内核的情况，这一步可省略）和其中字段的初始化。其中内核提供了 ether_setup 等辅助函数供调用，从而简化以太网设备的初始化工作。其次为了与网络栈衔接，驱动程序必须定义发送函数，设备打开函数（可无），中断处理函数，以及其他一些辅助函数如信息获取函数，组播地址配置函数等。总之，驱动程序是围绕一个 device 结构而展开的，剩下的就是对具体硬件寄存器的操作！

结合以上说明，读者可着重分析 ne.c, 8390.c 文件进行理解。另外参考文献[2]中对于网络驱动程序的编写有较为详细的说明，虽然这是针对 Linux 最新版本的，但网络驱动程序的基本结构都是一致的！

3.3 本章小结

虽然在介绍 Linux-1.2.13 下具体驱动程序的意义不大，但对驱动程序工作原理的分析却是必要的。虽然 Linux 发展至今，网络栈代码已经变得非常庞大，但其对驱动层的接口变化极其微小，而且驱动程序结构基本保持了一致，如果说变化，那么最大的变化就是从 device 结构变为了 netdevice 结构。本章所讨论的内容（即网络设备初始化过程和工作原理及其结构）都可以在当前版本 Linux 系统下做同样的理解。

第四章 系统网络栈初始化

本章我们将着重介绍系统网络栈的初始化过程，由于其中大多数函数在本书第一部分中都已有介绍，所以本章将不再拘泥于某个函数的分析之上，关键在于理清网络栈的初始化流程。我们从 init/main.c 文件出发，在执行了前面一系列具体处理器架构初始化代码之后，最终将进入到 init/main.c 中的 start_kernel 函数执行，系统网络栈初始化总入口函数将在 start_kernel 中被调用：即 sock_init 函数，非常干净，就从该函数出发完成整个网络栈的初始化过程。下面我们将跟随 sock_init 函数进行一段令人欣奋的旅程--深呼吸.....，啊！Linux 世界是如此美妙！

4.1 网络栈初始化流程

sock_init 函数定义在 net/socket.c 中，其调用：

1>proto_init：进行协议实现模块初始化。

2>dev_init：进行网络设备驱动层模块初始化，这个过程在第三章已有论述，本章不述。

3>初始化专门用于网络处理的下半部分执行函数：net_bh，并使能之。

4>附带的，其将由变量 pops 指向的域操作函数集合（如 INET 域，UNIX 域）数组清零，在此后的网络栈初始化中会对其进行正确的初始化。

proto_init 函数定义在 net/socket.c 中，其完成：

遍历由 protocols 全局变量（定义在 net/protocols.c 中）指向的域初始化函数集，进行各域的初始化，如 INET 域的初始化函数为 inet_proto_init，其将在 proto_init 函数被调用以进行 INET 域的初始化。当然除了 INET 域外，还有其他域类型，如 UNIX 域，本章我们只讨论标准的网络栈实现 INET 域。

inet_proto_init 函数定义在 net/inet/af_inet.c 中，其调用：

inet_protocol_add：

进行传输层和网络层之间的衔接。具体是通过将由 inet_protocol_base 全局变量指向的 inet_protocol 结构队列中元素散列到 inet_protos 数组中，从而被网络层使用。inet_protocol_base 和 inet_protos 变量以及 inet_protocol_add 函数均定义在 net/inet/protocol.c 中，inet_protocol 结构的作用类似于 packet_type 结构，都是作为两层之间的衔接之用，只不过 inet_protocol 结构用于传输层和网络层之间，而 packet_type 结构用于网络层和链路层之间。当网络层处理函数结束本层的处理后，其将查询 inet_protos 数组，匹配合适的传输层协议，调用其对应 inet_protocol 结构中注册的接收函数，从而将数据包传递给传输层协议处理。当然原则上，我们在网络层也可以采用直接查询由 inet_protocol_base 指向的队列，但由于传输层协议较多，采用数组散列方式，可以提高效率也便于进行代码的扩展。有关网络层向传输层的数据包传递过程请参考 ip_rcv 函数（net/inet/ip.c）。

arp_init：

ARP 协议初始化函数，ARP 协议是网络层协议，为了从链路层接收数据包，其必须定义一个 packet_type 结构，并调用 dev_add_pack 函数（dev.c）向链路层模块注册（这一点已经在第二章介绍 ARP 协议时着重谈到），arp_init 函数主要完成这个注册工作，同时注册事件通知句柄函数，监听网络设备状态变化事件，从而对 ARP 缓存进行及时刷新，以维护 ARP 缓

存内容的有效性。

ip_init：
IP 协议初始化函数，同样完成该协议接收函数对链路层的注册，同时由于路由表项与网络设备绑定的原因，其也必须注册网络设备状态变化事件侦听函数对此类事件进行侦听。

IP 协议和 ARP 协议都作为网络层协议，他们的初始化函数完成的功能基本相同：
1> 向链路层注册数据包接收函数，完成网络层和链路层之间的衔接。
2> 由于 ARP 表项以及路由表项都与某个网络设备接口绑定，故为了保证这些表项的有效性，二者都必须对网络设备状态变化事件进行检测，以便在某个网络设备状态变化时作出反应，所以在 ARP 协议，IP 协议初始化代码中，都注册了网络设备事件侦听函数。

至此，网络栈初始化工作全部完成，经过 inet_proto_init, arp_init, ip_init 函数的执行，系统完成了由下而上的各层接口之间的衔接：
1> 链路层和网络层通过 ptype_base 指向的 packet_type 结构队列进行衔接，每个 packet_type 结构表示一个网络层协议，结构中定义有网络层协议号及其接收函数，链路层模块将根据链路层首部中标识的网络层协议号在队列中进行查找，从而调用对应的接收函数将数据包上传给网络层协议进行处理。
2> 网络层和传输层通过 inet_protos 散列表（实际就是一个数组）进行衔接，表中每个表项指向一个 inet_protocol 结构队列，每个 inet_protocol 结构表示一个传输层协议，结构中定义有传输层协议号及其接收函数，网络层模块将根据网络层协议首部中标示的传输层协议号在 inet_protos 表中进行匹配查询，从而得到传输层协议对应的 inet_protocol 结构，调用结构中注册的接收函数，将数据包上传给传输层进行处理。

4.2 数据包传送通道解析
综上所述，我们得到网络栈自下而上的数据包传输通道，如下：
1> 硬件监听物理传输介质，进行数据的接收，当完全接收一个数据包后，产生中断。（注意这个过程完全由网络设备硬件负责。）
2> 中断产生后，系统调用驱动程序注册的中断处理程序进行数据接收，一般在接收例程中，我们完成数据从硬件缓冲区到内核缓冲区的复制，将其封装为内核特定结构（sk_buff 结构），最后调用链路层提供的接口函数 netif_rx，将数据包由驱动层传递给链路层。
3> 在 netif_rx 中，为了减少中断执行时间（注意 netif_rx 函数一般在驱动层中断例程中被调用），该函数将数据包暂存于链路层维护的一个数据包队列中（具体的由 backlog 变量指向），并启动专门进行网络处理的下半部分对 backlog 队列中数据包进行处理。
4> 网络下半部分执行函数 net_bh，从 backlog 队列中取数据包，在进行完本层相关处理后，遍历 ptype_base 指向的网络层协议（由 packet_type 结构表示）队列，进行协议号的匹配，找到协议号匹配的 packet_type 结构，调用结构中接收函数，完成数据包从链路层到网络层的传递。对于 ARP 协议，那么调用的就是 arp_rcv 函数，IP 协议则是 ip_rcv 函数，以下我们以 IP 协议为例。
5> 假设数据包使用的是 IP 协议，那么从链路层传递到网络层时，将进入 ip_rcv 总入口函数，ip_rcv 完成本层的处理后，以本层首部（即 IP 首部）中标识的传输层协议号为散列值，对 inet_protos 散列表进行匹配查询，以寻找到合适的 inet_protocol 结构，进而调用结构中接收函数，完成数据包从网络层到传输层的传递。对于 UDP 协议，那么调用 udp_rcv 函数，TCP 协议为 tcp_rcv，ICMP 协议为 icmp_rcv，IGMP 协议为 igmp_rcv。注意一般我们将 ICMP，

IGMP 协议都称为网络层协议，但他们都封装在 IP 协议中，所以实现上，是作为传输层协议看待的。

6> 令数据包使用的是 TCP 传输层协议，那么此时将进入 tcp_rcv 函数。在第二章介绍 TCP 协议实现文件时，本书一再强调，所有使用 TCP 协议的套接字对应 sock 结构都被挂入 tcp_prot 全局变量表示的 proto 结构之 sock_array 数组中，采用以本地端口号为索引的插入方式，所以当 tcp_rcv 函数接收到一个数据包，在完成必要的检查和处理后，其将以 TCP 协议首部中目的端口号(对于一个接收的数据包而言,其目的端口号就是本地所使用的端口号)为索引，在 tcp_prot 对应 sock 结构之 sock_array 数组中得到正确的 sock 结构队列，在辅之以其他条件遍历该队列进行对应 sock 结构的查询，在得到匹配的 sock 结构后，将数据包挂入该 sock 结构中的缓存队列中（由 sock 结构中 receive_queue 字段指向），从而完成数据包的最终接收。

7> 当用户需要读取数据时，其首先根据文件描述符得到对应的节点（inode 结构表示），由节点得到对应的 socket 结构(作为 inode 结构中 union 类型字段存在)，进而得到对应的 sock 结构（socket 和 sock 结构之间维护有相互指向的指针字段）。之后即从 sock 结构之 receive_queue 指向的队列中取数据包，将数据包中数据拷贝到用户缓冲区，从而完成数据的读取。

以上 1>~7>即表述了数据包接收的完整通道，在如今 Linux 最新版本中，虽然网络栈实现代码作了很大的改变，但这个通道基本未变！

在前文的分析中,我们讲到网络栈初始化过程中各层协议的初始化都注重于上层协议向下层协议的衔接工作，即注重于数据包接收通道的创建工作，那么数据包发送通道是如何创建的？方式很简单：即下层向上层提供发送接口函数供上层直接进行调用，如驱动层通过 device 结构之 hard_start_xmit 函数指针向链路层提供发送函数，链路层提供 dev_queue_xmit 发送函数供网络层调用，而网络层提供 ip_queue_xmit 函数供传输层调用。除了 hard_start_xmit 函数指针可根据不同的设备动态赋值外，dev_queue_xmit 和 ip_queue_xmit 函数名称都是硬编码的,虽然传输层在调用 ip_queue_xmit 函数上采用了一种间接的方式(先初始化一个函数指针指向 ip_queue_xmit 函数，然后调用这个函数指针指向的函数)，但本质上一样。注意 dev_queue_xmit, ip_queue_xmit 并未采用任何向上层模块进行注册的方式工作，换句话说，它们都是作为上层模块的已知函数。当传输层向下层传递数据包，其就直接调用一个名为 ip_queue_xmit 的函数，而网络层向下传递数据包，其直接调用一个名为 dev_queue_xmit 的函数，这就是硬编码的含义！下面我们着重分析自上而下的数据包发送通道。在这之前我们假设使用 TCP 协议，而且已经完成三路握手建立过程，现在我们发送一个包含有普通数据的正常的 TCP 数据包。

从应用层调用 write 函数开始，我们会先后经历 sock_write(net/socket.c), inet_write(net/inet/af_inet.c), tcp_write(net/inet/tcp.c)， 只有到达 tcp_write 函数时，才进行数据的真正处理，之前的 sock_write 经过简单检查，调用 inet_write，而 inet_write 经过必要检查，调用 tcp_write。在这个过程中，这种函数调用关系的建立是通过 socket，sock 数据结构完成的，具体情况读者可参考以上各函数的代码实现，此处不再赘述。为何在传输层才进行用户数据的真正处理，因为数据的处理方式需要根据所使用的传输层协议决定，更深层的原因在于只有在传输层才能进行数据的封装,因为我们只有在此层才知道需要预留多大的内核空间创建一个数据帧。对于七层分层方式中，在表示层所进行的数据加密等操作此处没有讨论的意义，因为本版本网络栈实现是针对四层模型的。

1> tcp_write 函数完成数据的封装：即将数据从用户缓冲区复制到内核缓冲区中，并封装在 sk_buff 结构中，根据网络的拥塞情况，此时可能出现两种情况：其一不经过 sock 结构之 write_queue 队列直接被发送出去，其二数据包被暂时缓存在 write_queue 队列中，稍后发送。作为传输层协议而言，其将调用 ip_queue_xmit 函数将数据包发往下层-网络层进行处理。
2> ip_queue_xmit 函数继续对数据帧进行完善后，调用 dev_queue_xmit 函数将数据包送往链路层进行处理，同时将数据包缓存于 sock 结构之 send_head 队列。这个缓存的目的在于 TCP 协议需要保证可靠性数据传输，即其必须防止数据包的丢失，一旦发生丢失，就需要重新发送 send_head 队列中数据包。另外注意的一点是，数据包在传递给 ip_queue_xmit 函数时，已经从 write_queue 队列中删除，所以不会出现数据包同时存在于 write_queue 和 send_head 队列的情况。这两个队列中数据包的区别在于：write_queue 队列中数据包是从用户层接收的新数据包，尚未（调用 ip_queue_xmit）进行发送；而 send_head 队列中数据包是 TCP 协议为了保证可靠性数据传输而缓存的已经（调用 ip_queue_xmit）发送出去的数据包。对于网络层（ip_queue_xmit 函数）而言，其将直接调用 dev_queue_xmit 函数进行发送。
3> dev_queue_xmit 函数完成其本层的处理后，调用发送设备 device 结构之 hard_start_xmit 指针指向的具体硬件数据发送函数，例如第三章对于 NE 系列网络设备的 ei_start_xmit 函数，该函数首先将数据从内核缓冲区复制到 NE 网卡设备的硬件缓冲区，操作具体的硬件相关寄存器，由硬件完成最终的发送工作。

至此，我们完成发送通道和接收通道的梳理。了解通道中的各个发送和接收节点（函数）对于整个网络栈实现代码的理解非常重要。在清楚了解了总体架构的前提下，更便于我们对代码的理解和消化。

4.3 本章小结

本章阐述了 Linux-1.2.13 网络栈的初始化流程，着重介绍了流程中调用的各节点函数及其完成的主要功能。其后，较为详细的分析了数据包发送和接收的传输通道，了解了网络栈实现的整体架构，这对于理解整个网络栈的实现很有意义。

附录 A　TCP 协议可靠性数据传输实现原理分析

TCP 协议是一种面向连接的，为不同主机进程间提供可靠数据传输的协议。TCP 协议假定其所使用的网络栈下层协议（如 IP 协议）是非可靠的，其自身提供机制保证数据的可靠性传输。在目前的网络栈协议族中，在需要提供可靠性数据传输的应用中，TCP 协议是首选的，有时也是唯一的选择。TCP 协议是在最早由 Cerf 和 Kahn[1]所提出的有关网络数据包传输协议的概念之上建立的。TCP 协议被设计成符合分层协议结构，工作在 ISO/OSI 七层网络模型中的传输层中，使用网络层协议（如最常见的 IP 协议）提供的服务。网络层协议尽最大努力传输上层提供的数据但并不保证数据传输的可靠性。可靠性保证必须由上层协议（如 TCP 协议）提供。网络层协议主要完成的工作有：

1> 实现不同网络（主机）间的数据包路由传递。
2> 在发送端（或中转站）提供数据包分片功能以使数据包大小满足 PMTU（Path-MTU）。
3> 在接收端提供数据包分片重组功能。
4> 负责数据包优先级，安全性等问题。

传输层协议（主要针对 TCP 协议而言）主要完成的工作有（并非所有的传输层协议都需要提供这些功能如 UDP 协议就不提供可靠性数据传输）：

1> 提供多路复用。
2> 实现数据基本传输功能。
3> 建立通信通道。
4> 提供流量控制。
5> 提供数据可靠性传输保证。

数据可靠性传输保证是其中最为重要的方面，也是 TCP 协议区别于其它协议的最重要特性。所谓提供数据可靠性传输不仅仅指将数据成功的由本地主机传送到远端主机，数据可靠性传输包括如下内容：

1> 能够处理数据传输过程中被破坏问题。
2> 能够处理重复数据接收问题。
3> 能够发现数据丢失以及对此进行有效解决。
4> 能够处理接收端数据乱序到达问题。

1．TCP 协议可靠性数据传输实现基本原理

TCP 协议必须提供对所有这些问题的解决方案方可保证其所声称的数据可靠性传输。TCP 协议规范和当前绝大多数 TCP 协议实现代码均采用数据重传和数据确认应答机制来完成 TCP 协议的可靠性数据传输。数据超时重传和数据应答机制的基本前提是对每个传输的字节进行编号，即我们通常所说的序列号。数据超时重传是发送端在某个数据包发送出去，在一段固定时间后如果没有收到对该数据包的确认应答，则（假定该数据包在传输过程中丢失）重新发送该数据包。而数据确认应答是指接收端在成功接收到一个有效数据包后，发送一个确认应答数据包给发送端主机，该确认应答数据包中所包含的应答序列号即指已接收到的数据中最后一个字节的序列号加 1，加 1 的目的在于指出此时接收端期望接收的下一个数据包中第一个字节的序列号。**数据超时重传和数据确认应答以及对每个传输的字节分配序列号是 TCP 协议提供可靠性数据传输的核心本质。**

1）数据确认应答数据包中应答序列号的含义

应答序列号并非其表面上所显示的意义，其实际上是指接收端希望接收的下一个字节的序列号。所以接收端在成功接收到部分数据后，其发送的应答数据包中应答序列号被设置为这些数据中最后一个字节的序列号加一。所以从其含义上来说，应答序列号称为请求序列号有时更为合适。应答序列号在 TCP 首部中应答序列号字段中被设置。而 TCP 首部中序列号字段表示包含该 TCP 首部的数据包中所包含数据的第一个字节的序列号（令为 N）。如果接收端成功接收该数据包，之前又无丢失数据包，则接收端发送的应答数据包中的应答序列号应该为：N+LEN。其中 LEN 为接收的数据包的数据长度。该应答序列号也是发送端将要发送的下一个数据包中第一个字节的序列号（由此亦可看出上文中将应答序列号称为请求序列号的原因所在）。

2）数据确认应答中的累积效应

TCP 协议中接收端对所接收数据的应答是累积的。累积的含义有二：

1>应答序列号是逐渐递增的，这与发送端数据编号是递增的相吻合。

2>不可进行跨越式数据应答。

所谓不可进行跨越式数据应答，可以以数据包乱序到达为例进行说明。如果由于发送端所选择传输路径的不同，较后发送的序列号较大的数据包先到达接收端，而先发送的序列号较小的数据包由于线路问题（或路由器故障）被暂时延迟在网络中，此时接收端不可对这些序列号较大的数据进行应答。如果接收端需要发送一个应答数据包，则应答序列号仍然应该设置成对序列号较小的数据包的请求（注意应答序列号指的是接收端希望接收的下一个字节的序列号，故在数据传输过程中将应答数据包称为数据请求数据包更为合适）。举例来说，如果接收端目前的应答序列号为 201，表示接收端正在等待发送端发送从 201 开始编号的数据，之后发送端连续发送了两个数据包，第一个数据包中数据序列号范围为 201-300，第二个数据包中数据序列号范围为 301-400。如果由于选择了不同的传输路径造成第二个数据包最先到达接收端，而第一个数据包在网络中延迟了一段时间，则接收端不可对第二个数据包进行应答，即不可发送应答序列号为 401 的确认应答数据包，而是不断发送应答序列号为 201 的应答数据包直到该序列号的数据到达。我们通常所说的快速重传机制即发送端在连续接收到 3 个相同序列号的应答数据包后需要立刻重传应答序列号所表示的数据。因为此时表示极有可能出现了数据包丢失的情况，如上例中第一个数据包如果丢失在网络中并且发送端重传的相同数据包由于选择相同的线路也未能到达接收端，则接收端将不断发送应答序列号为 201 的应答数据包而不会将应答序列号设置为 401。注意此时接收端已接收到序列号从 301-400 的数据。

3）重传应答机制与序列号结合：

1> 能够处理数据在传输过程中被破坏的问题。

首先通过对所接收数据包的校验，确认该数据包中数据是否存在错误。如果有，则简单丢弃或者发送一个应答数据包重新对这些数据进行请求。发送端在等待一段时间后，则会重新发送这些数据。本质上，数据传输错误的解决是通过数据重传机制完成的。

2> 能够处理接收重复数据问题。

首先利用序列号可以发现数据重复问题。因为每个传输的数据均被赋予一个唯一的序列号，如果到达的两份数据具有重叠的序列号（如由发送端数据包重传造成），则表示出现数据重复问题，此时只须丢弃其中一份保留另一份即可。多个数据包中数据重叠的情况解决方式类似。本质上，数据重复问题的解决是通过检查序列号完成的。

3> 能够发现数据丢失以及进行有效解决。
首先必须说明，此处数据包丢失的概念是指在一段合理时间内，应该到达的数据包没有到达，而非我们平常所理解的永远不到达。所以数据包丢失与数据包乱序到达有时在判断上和软件处理上很难区分。
数据丢失的判断是猜测性的，我们无法确定一个数据包一定丢失在传输过程中，大多是被延迟在网络中，即实质的问题只是数据包乱序到达。将二者区分开来的一个主要依据是在合理的时间内，由这个可能丢失的数据包所造成的序列号“空洞“是否能够被填补上。可能的数据丢失一个显然的结果是在接收端接收的数据出现序列号不连续现象。如接收端只接收到序列号从 1 到 100 的数据包，之后又接收到序列号从 200 到 300 的数据包，而且在一段合理的时间内（由此基本排除乱序问题），序列号从 101 到 199 的数据一直未到达，则表示包含序列号从 101 到 199 的数据包在传输过程中很可能丢失（或者有极不正常的延迟）。 对数据包是否丢失判断的另外一个干扰因素是发送端的重传机制，如果一个序列号较前的数据包在网络中丢失，造成序列号较后的数据包提前到达接收端，也会暂时造成序列号不连续，但由于发送端在没有接收到确认应答时，会重新发送序列号较前的那个数据包，如果此后接收端接收到一个重传的数据包，则仅仅只会在接收端造成数据包乱序到达的表象。

无论实质如何，如果软件实现判断出数据包丢失，则接收端将通过不断发送对这些丢失的数据的请求数据包（也即应答数据包，见前文中对数据应答数据包和数据应答累积效应的说明）来迫使发送端重新发送这些数据。通常发送端自身会自发的重传这些未得到对方确认的数据，但由于重传机制采用指数退避算法，每次重传的间隔时间均会加倍，所以通过发送方主动重传机制恢复的时间较长，而接收端通过不断发送对这些丢失数据的请求，发送端在接收到三个这样的请求数据包后（三个请求数据包中具有同一个请求序列号--也即前文中所说的应答序列号），会立刻触发对这些数据的重新发送，这称为快速恢复或者快速重传机制。

本质上，对于数据丢失问题的解决是通过数据重传机制完成的。在此过程中序列号和数据确认应答起着关键的作用。

4> 能够处理接收端数据乱序到达问题。
如果通信双方存在多条传输路径， 则有可能出现数据乱序问题，即序列号较大的数据先于序列号较小的数据到达，而发送端确实是按序列号由小到大的顺序发送的。数据乱序的本质是数据都成功到达了，但到达的顺序不尽如人意。对这个问题的解决相对比较简单，只需对这些数据进行重新排序即可。

本质上，对数据乱序问题的解决是通过排序数据序列号完成的。

2．TCP 协议可靠性数据传输软件实现基本原理
由上文可见，序列号，数据超时重传和数据确认应答机制保证了 TCP 协议可靠性传输的要求。由于需要对所发送的数据进行编号，又需要对接收的数据进行应答，所以使用 TCP 协议的通信双方必须通过某种机制了解对方的初始序列号。只有在确切知道对方的初始序列号的情况下，才能从一开始对所接收数据的合法性进行判断。另外还需要在本地维护一个对方应答的序列号，以随时跟随对方的数据请求。在最后通信通道关闭时，可以确知本地发送的数是否已被对方完全接收；此外这个对方应答序列号在控制本地数据通量方面也发挥着重要的作用：用本地发送序列号减去对方应答序列号则可以立刻得知目前发送出去的数据有多少

没有得到对方的应答。综上所述，可靠性传输要求通信双方维护如下序列号：
SND.NXT
本地将要发送的下一个序列号。该变量对应 TCP 首部中序列号字段。表示该数据包中所包含数据的第一个字节的序列号。每次发送一个数据包，该变量都需要进行更新：
SND.NXT = SND.NXT + 本次发送的数据包中包含的数据长度

SND.ACKED
对方对本地所发送数据到目前为止进行了应答的序列号，换句话说，SND.ACKED+1 表示本地已发送出去但尚未得到对方应答的数据集中对应的第一个（最小的）序列号。

RCV.NXT
本地希望接收的下一个序列号。该序列号被称为应答序列号，也可称为请求序列号，在本地发送的应答报文中，TCP 首部中应答序列号字段即设置为该变量的值，表示本地希望从对方接收的下一个字节的序列号。
图 1　TCP 首部格式

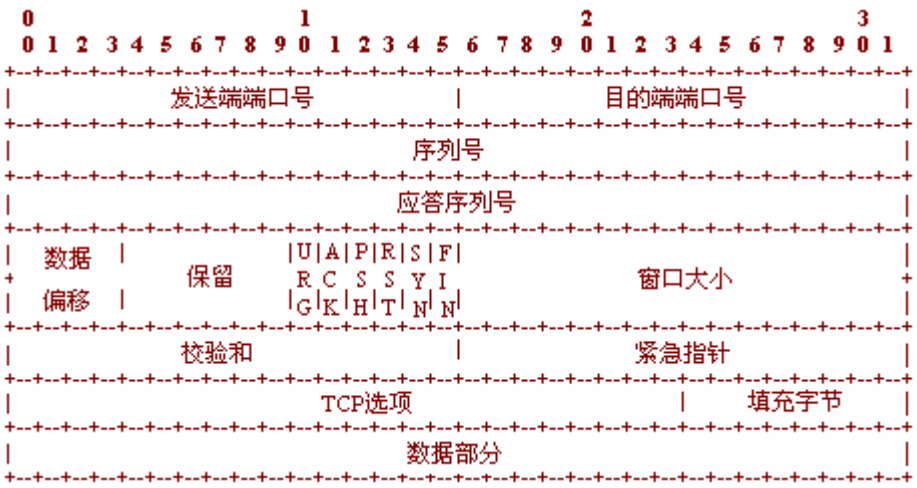


图 1（上图）显示了 TCP 首部格式。序列号字段对应前文中 SND.NXT 变量，应答序列号字段对应前文中 RCV.NXT 变量。ACK 标志位设置为 1 表示这是一个应答数据包。
实际上对于 TCP 协议而言，在成功建立连接后，此后发送的所有数据包的 ACK 标志位均被设置为 1，即在传送正常数据的同时传送应答，如此处理可以减少网络中传输的数据包数量。

3．TCP 协议建立连接的必要性
图 1TCP 首部格式中 SYN 标志位仅使用在建立 TCP 连接的过程中，TCP 建立连接的过程被称为“三路握手“连接，即一般通信双方共需要传输三个数据包方能成功建立一个 TCP 连接。我们通常将建立连接作为使用 TCP 协议理所当然的前导过程，但很少去质疑这样一个建立连接过程的必要性。实际上，在上文中已经做出部分解释，使用 TCP 协议必须首先建立一个连接是保证 TCP 协议可靠性数据传输的基本前提（当然由于 TCP 协议是一个有状态协议，必须通过某种机制进行通信双方状态上的同步，而建立连接就是这样一种机制）。至于为何需要三个数据包，原因是建立连接过程中信息的交换必须至少使用三个数据包，从下文的分析来看，建立连接最多需要使用四个数据包。需要再次提到的是：SYN 标志位只是用在建立连接的三个（或者四个）数据包中，一旦连接建立完成后，之后发送的所有数据包不可设置 SYN 标志位。

单从保证数据可靠性传输角度而言，TCP 协议需要在正式数据传输之前首先进行某些信息的交换，这个信息即是双方的初始序列号（另外的一些信息包括最大报文长度通报等）。
诚如前文所述，序列号的使用对于 TCP 协议而言至关重要，在正式数据传输之前，双方必须得到对方的初始字节数据的编号，这样才有可能对其所接收数据的合法性进行判断，才有其它的对数据重复，数据重叠等一系列问题的进一步判别和解决。故交换各自的初始序列号必须在正式数据传输之前完成，我们美其名曰这个过程为连接建立过程。至于双方 TCP 协议各自状态的更新主要是软件设计上可靠性保证的一个辅助，并非这个所谓的建立过程所主要关注的问题。

初始序列号的交换从最直接的角度来说需要四个数据包：
1> 主机 A 向主机 B 发送其初始序列号。
2> 主机 B 向主机 A 确认其发送的初始序列号。
3> 主机 B 向主机 A 发送其初始序列号。
4> 主机 A 向主机 B 确认其发送的初始序列号。

我们将<2><3>两步合为一步，即 B 向 A 确认其（A 之前发送的）初始序列号的同时发送其（即 B 自己的）初始序列号。所谓确认数据包即将数据包的 ACK 标志位设置为 1 即可。注意这三个（或四个）数据包中 SYN 标志位设置为 1，而且 SYN 标志位也仅在这三个（或四个）数据包中被设置为 1。

此处有一个问题：即 A，B 主机在通报各自初始序列号的同时能否传输一些正常数据，原理上可以（TCP 协议规范上并没有说不可以），但是大多数实现在通报初始序列号时都不附带正常数据，而是将其作为一个单独的过程，由此正式确立建立连接一说。

小结

TCP 协议声称可靠性数据传输，其底层实现机制主要包括三个方面：使用序列号对传输的数据进行编号，数据超时重传，数据确认应答。本文主要阐述了此三个方面为何能够实现可靠性传输并简单解释了其中的内部机理，此后对 TCP 协议可靠性数据传输实现提出了其基本原理，并在文章最后从保证可靠性数据传输的角度简单阐述了使用 TCP 协议时为何需要一个建立连接的过程。TCP 协议是 TCP/IP 协议族中较为复杂的一个，其复杂性的最主要来源之一即其需要提供可靠性数据传输，本文旨在对 TCP 协议保证可靠性数据传输的基本实现原理介绍中降低读者对 TCP 协议理解的复杂度。

主要参考文献

[1] W. Richard Stevens, TCP/IP 详解 卷 1：协议（英文版），北京：机械工业出版社，2002.01。
[2] Jonathan Corbet 等著，魏永明等译，Linux 设备驱动程序，第三版，北京：中国电力出版社，2005.11。
[3] 赵炯著，Linux 内核完全剖析，北京：机械工业出版社，2006.01
[4] Christian Benvenuti 著，深入理解 Linux 网络内幕（影印版），南京：东南大学出版社，2006.05
[5] 毛德操，胡希明著，Linux 内核源代码情景分析（上，下册），杭州：浙江大学出版社，2001.09
[6] W. Richard Stevens，Unix 网络编程，卷 1（英文版），北京：清华大学出版社，1998.07
[7] Richard Blum 著，马朝晖等译，汇编语言程序设计，北京：机械工业出版社，2006.01
[8] Behrouz A. Forouzan 著，吴时霖等译，数据通信与网络，北京：机械工业出版社，2002.01
[9] Tony Bautts 等著，O'Reilly Taiwan 公司译，Linux 网络管理员指南，南京：东南大学出版社，2006.07
[10] 李善平等著，Linux 内核 2.4 版源代码分析大全，北京：机械工业出版社，2002.01
[11] Samuel P.Harbison III 等著，C 语言参考手册，第五版（英文版），北京：人民邮电出版社，2007.07
[12] David C. Plummer, "An Ethernet Address Resolution Protocol", RFC826, Nov. 1982.
[13] Jon Postel, "Internet Protocol – DARPA Internet Protocol Specification", RFC791, Sep. 1981.
[14] Jon Postel, "Internet Control Message Protocol", RFC792, Sep, 1981.
[15] Jon Postel, "Transmission Control Protocol", RFC793, Sep. 1981.
[16] Jon Postel, "User Datagram Protocol", RFC768, Aug. 1980.
[17] W. Fenner, "Internet Group Management Protocol, Version 2", RFC2236, Nov. 1997.
[18] W. Simpson, "The Point-to-Point Protocol (PPP)", RFC1661, July. 1994.
[19] N. Freed, "Behavior of and Requirements for Internet Firewalls", RFC2979, Oct. 2000.
[20] B. Lloyd, W. Simpson, "PPP Authentication Protocols", RFC1334, Oct. 1992
[21] G. McGregor, "The PPP Internet Protocol Control Protocol (IPCP)", RFC1332, May. 1992.