第32章 原 始 IP

32.1 引言

应用进程在Internet域中创建一个SOCK_RAW类型的插口，就可以利用原始IP层。一般有下列3种用法：

1) 应用进程可利用原始插口发送和接收ICMP和IGMP报文。
   Ping程序利用这种类型的插口，发送ICMP回显请求和接收ICMP回显应答。
   有些选路守护程序，利用这一特性跟踪通常由内核处理的ICMP重定向报文段。我们在19.7节中提到，Net/3处理重定向报文段时，会在需重定向的插口上生成RTM_REDIRECT消息，从而无需利用原始插口的这一功能。
   这个特性还用于实现基于ICMP的协议，如路由通告和路由请求(卷1的9.6节)，它们需用到ICMP，不过最好由应用进程，而不是内核完成相应处理。
   多播路由守护程序利用原始IGMP插口，发送和接收IGMP报文。
2) 应用进程可利用原始插口构造自己的IP首部。路由跟踪程序利用这一特性生成自己的UDP数据报，包括IP和UDP首部。
3) 应用进程可利用原始插口读写内核不支持的IP协议的IP数据报。
   gated程序利用这一特性支持基于IP的路由协议：EGP、HELLO和OSPF。
   这种类型的原始插口还可用于设计基于IP的新的运输层协议，而无需增加对内核的支持。调试应用进程代码比调试内核代码容易得多。

本章介绍原始IP插口的实现。

32.2 代码介绍

图32-1给出的C文件中包含了5个原始IP处理函数。

文　件	描　述
netinet/raw_ip.c	原始IP处理函数

图32-1 本章讨论的文件

图32-2给出了5个原始IP函数与其他内核函数间的关系。

带阴影的椭圆表示我们在本章中将要讨论的5个函数。请注意，原始IP函数名中的前缀“rip”表示“原始IP (Raw IP)”，而不是“选路信息协议(Routing Information Protocol)”，后者的缩写也是RIP。

32.2.1 全局变量

本章中用到4个全局变量，如图32-3所示。

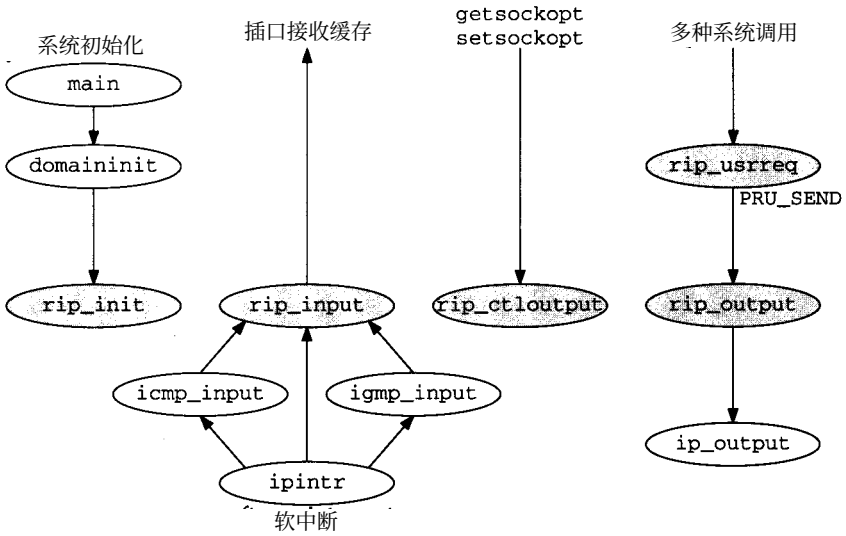

图32-2 原始IP函数与其他内核函数间的关系

变 量	数据类型	描 述
rawinpcb	struct inpcb	原始IP的Internet PCB链表表头
ripsrc	struct sockaddr_in	在输入中包含发送方的IP地址
rip_recvspace	u_long	插口接收缓存大小默认值，8192字节
rip_sendspace	u_long	插口发送缓存大小默认值，8192字节

图32-3 本章介绍的全局变量

32.2.2 统计量

原始IP在ipstat结构(图8-4)中维护两个计数器，如图32-4所示。

ipstat成员变量	描 述	SNMP变量使用
ips_noproto	协议类型未知或不支持的数据报数目	•
ips_rawout	生成的原始IP数据报总数	

图32-4 ipstat 结构中维护的原始IP统计量

图8-6给出了如何在SNMP中使用ips_noproto计数器。图8-5给出了这两个计数器输出值的例子。

32.3 原始IP的protosw结构

与所有其他协议不同，inetsw数组有多条记录都可以读写原始IP。inetsw结构中有4个记录的插口类型都等于SOCK_RAW，但协议类型则各不相同：

- IPPROTO_ICMP(协议值1)；
- IPPROTO_IGMP(协议值2)；
- IPPROTO_RAW(协议值255)；和

- 原始IP通配记录(协议值0)。

其中ICMP和IGMP，前面已介绍过(图11-12和图13-9)。四项记录间的区别总结如下：

- 如果应用进程创建了一个原始插口 (SOCK_RAW)，协议值非零(socket的第三个参数)，并且如果协议值等于IPPROTO_ICMP、IPPROTO_IGMP或IPPROTO_RAW，则会使用对应的protosw记录。
- 如果应用进程创建了一个原始插口 (SOCK_RAW)，协议值非零，但内核不支持该协议，pffindproto会返回协议值为0的通配记录，从而允许应用进程处理内核不支持的 IP协议，而无需修改内核代码。

我们在7.8节中提到，ip_protox数组中的所有未知记录都指向IPPROTO_RAW，它的协议转换类型如图32-5所示。

成 员	inetsw[3]	描 述
pr_type	*SOCK_RAW*	原始插口
pr_domain	*& inetdomain*	属于Internet域的原始IP
pr_protocol	*IPPROTO_RAW*(255)	出现在IP首部的ip_p字段
pr_flags	*PR_ATOMIC|PR_ADDR*	插口层标志，不用于协议处理
pr_input	*rip_input*	从IP层接收报文段
pr_output	*0*	原始IP不使用
pr_ctlinput	*0*	原始IP不使用
pr_ctloutput	*rip_ctlinput*	响应应用进程的管理请求
pr_usrreq	*rip_usrreq*	响应应用进程的通信请求
pr_init	*0*	原始IP不使用
pr_fasttimo	*0*	原始IP不使用
pr_slowtimo	*0*	原始IP不使用
pr_drain	*0*	原始IP不使用
pr_sysctl	*0*	原始IP不使用

图32-5 原始IP的protosw 结构

本章中我们将介绍3个以rip_开头的函数，此外还大致提一下 rip_output函数，它没有出现在协议转换记录中，但输出原始 IP报文段时，rip_usrreq将会调用它。

第五个原始IP函数，rip_init，只出现在通配处理记录中。初始化函数只能调用一次，所以它既可以出现在IPPROTP_RAW记录中，也可以放在通配记录中。

不过，图32-5中并没有说明其他协议 (ICMP和IGMP)，在它们自己的 protosw结构中也用到了一些原始IP函数。图32-6对4个SOCK_RAW协议各自protosw结构的相关成员变量做了一个比较。为了强调指出彼此间的区别，不同之处都用黑体字标出。

protosw 记录	SOCK_RAW 协议类型			
	IPPROTO_ICMP (1)	IPPROTO_IGMP (2)	IPPROTO_RAW (255)	通配(0)
pr_input	***icmp_input***	***igmp_input***	***rip_input***	***rip_input***
pr_output	*rip_output*	*rip_output*	*rip_output*	*rip_output*
pr_ctloutput	*rip_ctloutput*	*rip_ctloutput*	*rip_ctloutput*	*rip_ctloutput*
pr_usrreq	*rip_usrreq*	*rip_usrreq*	*rip_usrreq*	*rip_usrreq*
pr_init	*0*	***igmp_init***	*0*	***rip_init***
pr_sysctl	***icmp_sysctl***	*0*	*0*	*0*
pr_fasttimo	*0*	***igmp_fasttimo***	*0*	*0*

图32-6 原始插口的协议散转值的比较

不同BSD版本中，原始IP的实现各有不同。ip_protox表中，协议号等于IPPROTO_RAW记录通常都用做通配记录以支持未知的IP协议，而协议号等于0的记录通常做为默认记录，从而允许应用进程读写内核不支持的IP协议数据报。

应用进程使用IPPROTO_RAW记录，最早见于Van Jacobson开发的Traceout，这是第一个需要自己写IP首部(改变TTL字段)的应用进程。为了支持Traceout，修订了4.3BSD和Net/1，包括修改rip_output，在收到协议号等于IPPROTO_RAW的数据报时，假定应用进程提交了一个完整的IP数据报，包括IP首部。在Net/2中，引入了IP_HDRINCL插口选项，简化了IPPROTO_RAW的用法，允许应用进程利用通配记录发送自己的IP首部。

32.4 rip_init函数

系统初始化时，domaininit函数调用原始IP初始化函数rip_init (图32-7)。

raw_ip.c

```
47 void
48 rip_init()
49 {
50     rawinpcb.inp_next = rawinpcb.inp_prev = &rawinpcb;
51 }
```

raw_ip.c

图32-7 rip_init 函数

这个函数执行的唯一操作是令PCB首部(rawinpcb)中的前向和后向指针都指向自己，实现一个空的双向链表。

只要某个socket系统调用创建了SOCK_RAW类型的插口，下面将介绍的原始IP PRU_ATTACH函数就创建一个Internet PCB，并插入到rawinpcb链表中。

32.5 rip_input函数

因为ip_protox数组中保存的所有关于未知协议记录都指向IPPROTO_RAW(图7-8)，且后者的pr_input函数指向rip_input (图32-6)，所以只要某个接收IP数据报的协议号内核无法识别，就会调用此函数。但从图32-2可看出，ICMP和IGMP都可能调用rip_input，只要满足下列条件：

- icmp_input调用rip_input处理所有未知的ICMP报文类型和所有非响应的ICMP报文。
- igmp_input调用rip_input处理所有IGMP分组。

上述两种情况下，调用rip_input的一个原因是允许创建了原始插口的应用进程处理新增的ICMP和IGMP报文，内核可能不支持它们。

图32-8给出了rip_input函数。

raw_ip.c

```
59 void
60 rip_input(m)
61 struct mbuf *m;
62 {
63     struct ip *ip = mtod(m, struct ip *);
64     struct inpcb *inp;
```

图32-8 rip_input 函数

```
65      struct socket *last = 0;

66      ripsrc.sin_addr = ip->ip_src;
67      for (inp = rawinpcb.inp_next; inp != &rawinpcb; inp = inp->inp_next) {
68          if (inp->inp_ip.ip_p && inp->inp_ip.ip_p != ip->ip_p)
69              continue;
70          if (inp->inp_laddr.s_addr &&
71              inp->inp_laddr.s_addr == ip->ip_dst.s_addr)
72              continue;
73          if (inp->inp_faddr.s_addr &&
74              inp->inp_faddr.s_addr == ip->ip_src.s_addr)
75              continue;
76          if (last) {
77              struct mbuf *n;
78              if (n = m_copy(m, 0, (int) M_COPYALL)) {
79                  if (sbappendaddr(&last->so_rcv, &ripsrc,
80                                   n, (struct mbuf *) 0) == 0)
81                      /* should notify about lost packet */
82                      m_freem(n);
83                  else
84                      sorwakeup(last);
85              }
86          }
87          last = inp->inp_socket;
88      }
89      if (last) {
90          if (sbappendaddr(&last->so_rcv, &ripsrc,
91                           m, (struct mbuf *) 0) == 0)
92              m_freem(m);
93          else
94              sorwakeup(last);
95      } else {
96          m_freem(m);
97          ipstat.ips_noproto++;
98          ipstat.ips_delivered--;
99      }
100 }
```

raw_ip.c

图32-8 (续)

59-66 IP数据报中的源地址被保存在全局变量 ripsrc中，只要找到了匹配的 PCB，ripsrc将做为参数传给 sbappendaddr。与UDP不同，原始IP没有端口号的概念，因此sockaddr_in结构中的sin_port总等于0。

2. 在所有原始IP PCB中寻找一个或多个匹配的记录

67-88 原始IP处理PCB表的方式与UDP和TCP不同。前面介绍过，这两个协议维护一个指针，总是指向最近收到的报文段 (单报文段缓存)，并调用通用函数in_pcblookup寻找一个最佳匹配(如果收到的数据报不同于缓存中的记录)。由于原始IP数据报可能发送到多个插口上，所以无法使用 in_pcblookup，因此，必须遍历原始 PCB链表中的所有 PCB。这一点类似于UDP处理广播报文段和多播报文段的方式 (图23-26)。

3. 协议比较

68-69 如果PCB中的协议字段非零，并且与IP首部的协议字段不匹配，则PCB被忽略。也说明协议值等于0(socket的第三个参数)的原始插口能够匹配所有收到的原始 IP报文段。

4. 比较本地和远端IP地址

70-75 如果PCB中的本地地址非零，并且与IP首部的目的IP地址不匹配，则PCB被忽略。如果PCB中的远端地址非零，并且与IP首部的源IP地址不匹配，PCB被忽略。

上述3种测试说明应用进程能够创建一个协议号等于0的原始插口，即不绑定到本地地址，也不与远端地址建立连接，可以接收经rip_input处理的所有数据报。

代码71行和74行都有同样的错误：相等测试，实际应为不相等测试。

5. 递交接收数据报的复制报文段以备处理

76-94 sbappendaddr向应用进程提交一个接收数据报的复制报文段。变量last的使用与图23-26中的用法类似：因为sbappendaddr把报文段放入到适当队列中后将释放所有mbuf，如果有多个进程接收数据报的复制报文段，rip_input必须调用m_copy保存一份复制报文段。但如果只有一个应用进程接收数据报，则无需复制。

6. 无法上交的数据报

95-99 如果无法为数据报找到相匹配的插口，则释放mbuf，递增ips_noproto，递减ips_delivered。IP在调用rip_input之前已经递增过后一个计数器(图8-15)。由于数据报实际上没有上交给运输层，因此，必须递减ips_delivered，确保两个SNMP计数器，ipInDiscards和ipInDelivers(图8-16)，的正确性。

本节开始时，我们提到，icmp_input会为未知报文类型或非响应报文调用rip_input，意味着如果收到ICMP主机不可达报文，且rip_input找不到可匹配的原始插口PCB，ips_noproto会递增。这也说明为什么图8-5中的计数器值较大。在前面对该计数器的描述中提到“未知或不支持的协议”，这种说法是不正确的。

如果收到的IP数据报带有的协议字段，既无法为内核辩识，也无法由某个应用进程通过原始插口处理，Net/3不会生成差错代码等于2(协议不可达)的ICMP目的不可达报文。RFC 1122建议出现此种情况时应该生成ICMP差错报文(参见习题32.4)。

32.6 rip_output函数

图32-6中，ICMP、IGMP和原始IP都调用rip_output实现原始IP输出。应用进程调用5个写函数之一：send、sendto、sendmsg、write和writev，系统将输出报文段。如果插口已建立连接，就可以任意调用上述5个函数，尽管sendto和sendmsg中不能规定目的地址。如果插口没有建立连接，则只能调用sendto和sendmsg，且必须规定目的地址。

图32-9给出了rip_output函数。

1. 内核填充IP首部

119-128 如果IP_HDRINCR插口选项未定义，M_PREPEND为IP首部分配空间，并填充IP首部各字段。此处未填充的字段留待ip_output初始化(图8-22)。协议字段等于PCB中保存的值，并且是图32-10中socket系统调用的第三个参数。

TOS等于0，TTL等于255。内核为原始IP插口填充各首部字段时，通常都使用这些固定值。这与UDP和TCP不同，应用进程能够通过插口选项设定IP_TTL和IP_TOS值。

129 应用程序通过IP_OPTIONS插口选项设定的所有IP选项，都通过opts变量传给ip_output函数。

2. 调用者填充IP首部：IP_HDRINCR插口选项

130-133 如果选用了IP_HDRINCR插口选项，调用者在数据报前提供完整的IP首部。如果应用进程提供的ID字段等于0，对此类IP首部需做的唯一修改是ID字段。IP数据服的ID字段可以等于0。此处，rip_output对ID字段的赋值可以简化应用进程的处理，直接设 ID字段等于0，rip_output向内核请求内核变量ip_id的当前值，做为IP报文段的ID值。

134-136 令opts为空，忽略应用进程通过IP_OPTIONS可能设定的任何IP选项。如果调用者构建了自己的IP首部，其中肯定已包括了调用者希望加入的 IP选项。flags变量中必须有IP_RAWOUTPUT标志，告诉ip_output不要修改IP首部。

raw_ip.c

```
105 int
106 rip_output(m, so, dst)
107 struct mbuf *m;
108 struct socket *so;
109 u_long  dst;
110 {
111     struct ip *ip;
112     struct inpcb *inp = sotoinpcb(so);
113     struct mbuf *opts;
114     int     flags = (so->so_options & SO_DONTROUTE) | IP_ALLOWBROADCAST;
115     /*
116      * If the user handed us a complete IP packet, use it.
117      * Otherwise, allocate an mbuf for a header and fill it in.
118      */
119     if ((inp->inp_flags & INP_HDRINCL) == 0) {
120         M_PREPEND(m, sizeof(struct ip), M_WAIT);
121         ip = mtod(m, struct ip *);
122         ip->ip_tos = 0;
123         ip->ip_off = 0;
124         ip->ip_p = inp->inp_ip.ip_p;
125         ip->ip_len = m->m_pkthdr.len;
126         ip->ip_src = inp->inp_laddr;
127         ip->ip_dst.s_addr = dst;
128         ip->ip_ttl = MAXTTL;
129         opts = inp->inp_options;
130     } else {
131         ip = mtod(m, struct ip *);
132         if (ip->ip_id == 0)
133             ip->ip_id = htons(ip_id++);
134         opts = NULL;
135         /* XXX prevent ip_output from overwriting header fields */
136         flags |= IP_RAWOUTPUT;
137         ipstat.ips_rawout++;
138     }
139     return (ip_output(m, opts, &inp->inp_route, flags, inp->inp_moptions));
140 }
```

raw_ip.c

图32-9 rip_output 函数

137 计数器ips_rawout递增。执行Traceroute时，Traceroute每发送一个变量就会导致此变量加1。

rip_output的操作在不同版本中也有所变化。在Net/3中使用IP_HDRINCL插口选项时，rip_output对IP首部所做的唯一修改就是填充ID字段，如果应用进程将其定为0。因为IP_RAWOUTPUT标志置位，Net/3中的ip_output函数不改动IP首

部。但在Net/2中，即使IP_HDRINCL插口选项设定时，它也会修改IP首部中特定字段：IP版本号等于4，分片偏移量等于0，分片标志被清除。

32.7 rip_usrreq函数

协议的用户请求处理函数能够完成多种操作。与UDP和TCP的用户请求处理函数类似，rip_usrreq是一个很大的switch语句，每个PRU_*xxx*请求，都有一个对应的case子句。

图32-10给出的PRU_ATTACH请求，来自socket系统调用。

raw_ip.c

```
194 int
195 rip_usrreq(so, req, m, nam, control)
196 struct socket *so;
197 int     req;
198 struct mbuf *m, *nam, *control;
199 {
200     int     error = 0;
201     struct inpcb *inp = sotoinpcb(so);
202     extern struct socket *ip_mrouter;
203     switch (req) {

204     case PRU_ATTACH:
205         if (inp)
206             panic("rip_attach");
207         if ((so->so_state & SS_PRIV) == 0) {
208             error = EACCES;
209             break;
210         }
211         if ((error = soreserve(so, rip_sendspace, rip_recvspace)) ||
212             (error = in_pcballoc(so, &rawinpcb)))
213             break;
214         inp = (struct inpcb *) so->so_pcb;
215         inp->inp_ip.ip_p = (int) nam;
216         break;
```

raw_ip.c

图32-10 rip_usrreq 函数：PRU_ATTACH 请求

194-206 每次socket函数被调用时，都会创建新的socket结构，此时还没有指向某个Internet PCB。

1. 确认超级用户

207-210 只有超级用户才能创建原始插口，这是为了防止普通用户向网络发送自己的IP数据报。

2. 创建Internet PCB，保留缓存空间

211-215 为输入和输出队列保留所需空间，调用in_pcballoc分配新的Internet PCB，添加到原始IP PCB链表中(rawinpcb)，并与socket结构建立对应关系。rip_usrreq的nam参数就是socket系统调用的第三个参数：协议。它被保存在PCB中，因为rip_input需用它上交收到的数据报，rip_output也要把它填入到外出数据报的协议字段中(如果IP_HDRINCL未设定)。

原始IP插口与远端IP地址建立的连接，与UDP插口和远端IP地址建立的连接相类似。它固定了原始插口只能接收来自于特定地址的数据报，如我们在rip_input中所看到的。原始IP

与UDP一样，是一个无连接协议，下面两种情况下会发送PRU_DISCONNECT请求：

1) 关闭建立连接的原始插口时，在PRU_DETACH之前会先发送PRU_DISCONNECT请求。

2) 如果对一个已建立连接的原始插口调用connect，soconnect在发送PRU_CONNECT请求前会先发送PRU_DISCONNECT请求。

图32-11给出了PRU_DISCONNECT、PRU_ABORT和PRU_DETACH请求。

raw_ip.c

```
217     case PRU_DISCONNECT:
218         if ((so->so_state & SS_ISCONNECTED) == 0) {
219             error = ENOTCONN;
220             break;
221         }
222         /* FALLTHROUGH */

223     case PRU_ABORT:
224         soisdisconnected(so);
225         /* FALLTHROUGH */

226     case PRU_DETACH:
227         if (inp == 0)
228             panic("rip_detach");
229         if (so == ip_mrouter)
230             ip_mrouter_done();
231         in_pcbdetach(inp);
232         break;
```

raw_ip.c

图32-11 rip_usrreq 函数：PRU_DISCONNECT 、PRU_ABORT 和PRU_DETACH 请求

217-222 如果处理PRU_DISCONNECT请求的插口没有进入连接状态，则返回错误。

223-225 尽管禁止在一个原始插口上发送PRU_ABORT请求，这个case语句实际上是PRU_DISCONNECT请求处理的延续。插口转入断开状态。

226-230 close系统调用发送PRU_DETACH请求，这个case语句还将结束PRU_DISCONNECT请求的处理。如果socket结构用于多播选路(ip_mrouter)，则调用ip_mrouter_done取消多播选路。一般情况下，mrouted (8)守护程序会通过DVMPR_DONE插口选项取消多播选路，因此，这个条件用于防止mrouted (8) 在没有正确设定插口选项之前就异常终止了。

231 调用in_pcbdetach释放Internet PCB，并从原始IP PCB表(rawinpcb)中删除。

通过PRU_BIND请求，可以把原始IP插口绑定到某个本地IP地址上，如图32-12所示。我们在rip_input中指出，插口将只能接收发向该地址的数据报。

233-250 应用进程向sockaddr_in结构填充本地IP地址。下列3个条件必须全真，否则将返回差错代码EADDRNOTAVAIL：

1) 至少配置了一个IP接口；

2) 地址族应等于AF_INET (或者AF_IMPLINK，历史上人为造成的不一致)；和

3) 如果绑定的IP地址不等于0.0.0.0，它必须对应于某个本地接口。调用者的sockaddr_in中的端口号必须等于0，否则，ifa_ifwithaddr将返回错误。

本地IP地址保存在PCB中。

raw_ip.c

```
233     case PRU_BIND:
234         {
235             struct sockaddr_in *addr = mtod(nam, struct sockaddr_in *);
236             if (nam->m_len != sizeof(*addr)) {
237                 error = EINVAL;
238                 break;
239             }
240             if ((ifnet == 0) ||
241                 ((addr->sin_family != AF_INET) &&
242                  (addr->sin_family != AF_IMPLINK)) ||
243                 (addr->sin_addr.s_addr &&
244                  ifa_ifwithaddr((struct sockaddr *) addr) == 0)) {
245                 error = EADDRNOTAVAIL;
246                 break;
247             }
248             inp->inp_laddr = addr->sin_addr;
249             break;
250         }
```

raw_ip.c

图32-12 rip_usrreq 函数：PRU_BIND 请求

应用进程还可以在原始IP插口与某个特定远端IP地址间建立连接。我们在rip_input中指出，这样可以限制应用进程只能接收源 IP地址等于连接对端IP地址的数据报。应用进程可以同时调用bind和connect，或者两者都不调用，取决于它希望rip_input对接收数据报采用的过滤方式。图32-13给出了PRU_CONNECT请求的处理逻辑。

251-270 如果调用者的sockaddr_in初始化正确，且至少配置了一个IP接口，则指定的远端地址将存储在PCB中。注意，这一处理和UDP插口建立与远端IP地址的连接有所不同。对于UDP，in_pcbconnect申请到达远端地址的一条路由，并把外出接口视为本地地址(图22-9)。对于原始IP，只有远端IP地址存储到PCB中，除非应用进程还调用了bind，rip_input将只比较远端地址。

raw_ip.c

```
251     case PRU_CONNECT:
252         {
253             struct sockaddr_in *addr = mtod(nam, struct sockaddr_in *);
254             if (nam->m_len != sizeof(*addr)) {
255                 error = EINVAL;
256                 break;
257             }
258             if (ifnet == 0) {
259                 error = EADDRNOTAVAIL;
260                 break;
261             }
262             if ((addr->sin_family != AF_INET) &&
263                 (addr->sin_family != AF_IMPLINK)) {
264                 error = EAFNOSUPPORT;
265                 break;
266             }
267             inp->inp_faddr = addr->sin_addr;
268             soisconnected(so);
269             break;
270         }
```

raw_ip.c

图32-13 rip_usrreq 函数：PRU_CONNECT 请求

应用进程结束发送数据后，调用shutdown，生成PRU_SHUTDOWN请求，尽管应用进程很少为原始IP插口调用shutdown。图32-14给出了PRU_CONNECT2和PRU_SHUTDOWN请求的处理逻辑。

raw_ip.c

```
271     case PRU_CONNECT2:
272         error = EOPNOTSUPP;
273         break;

274         /*
275          * Mark the connection as being incapable of further input.
276          */
277     case PRU_SHUTDOWN:
278         socantsendmore(so);
279         break;
```

raw_ip.c

图32-14 PRU_CONNECT2 和PRU_SHUTDOWN 请求

271-273 原始IP插口不支持PRU_CONNECT2请求。

274-279 socantsendmore置位插口标志，禁止所有输出。

图23-14中，我们给出了5个写函数如何调用协议的pr_usrreq函数，发送PRU_SEND请求。图32-15给出了这个请求的处理逻辑。

raw_ip.c

```
280         /*
281          * Ship a packet out.  The appropriate raw output
282          * routine handles any massaging necessary.
283          */
284     case PRU_SEND:
285         {
286             u_long  dst;

287             if (so->so_state & SS_ISCONNECTED) {
288                 if (nam) {
289                     error = EISCONN;
290                     break;
291                 }
292                 dst = inp->inp_faddr.s_addr;
293             } else {
294                 if (nam == NULL) {
295                     error = ENOTCONN;
296                     break;
297                 }
298                 dst = mtod(nam, struct sockaddr_in *)->sin_addr.s_addr;
299             }
300             error = rip_output(m, so, dst);
301             m = NULL;
302             break;
303         }
```

raw_ip.c

图32-15 rip_usrreq 函数：PRU_SEND 请求

280-303 如果插口处于连接状态，则调用者不能指定目的地址(nam参数)。如果插口未建立连接，则需要指明目的地址。不管哪种情况，只要条件满足，dst将等于目的IP地址。rip_output发送数据报。令mbuf指针m为空，防止函数结束时释放mbuf链。因为接口输出

例程发送数据报之后会释放 mbuf链(记住，rip_output向ip_output提交mbuf链，ip_output把它加入到接口的输出队列中)。

图32-16给出了rip_usrreq的最后一部分代码。由fstat系统调用生成的PRU_SENSE请求，没有返回值。PRU_SOCKADDR和PRU_PEERADDR请求分别由getsockname和getpeername系统调用生成。原始IP插口不支持其余请求。

319-324 函数in_setsockaddr和in_setpeeraddr能够从PCB中读取信息，在nam参数中返回结果。

raw_ip.c

```
304     case PRU_SENSE:
305         /*
306          * fstat: don't bother with a blocksize.
307          */
308         return (0);

309         /*
310          * Not supported.
311          */
312     case PRU_RCVOOB:
313     case PRU_RCVD:
314     case PRU_LISTEN:
315     case PRU_ACCEPT:
316     case PRU_SENDOOB:
317         error = EOPNOTSUPP;
318         break;

319     case PRU_SOCKADDR:
320         in_setsockaddr(inp, nam);
321         break;

322     case PRU_PEERADDR:
323         in_setpeeraddr(inp, nam);
324         break;

325     default:
326         panic("rip_usrreq");
327     }
328     if (m != NULL)
329         m_freem(m);
330     return (error);
331 }
```

raw_ip.c

图32-16 rip_usrreq 函数：剩余的请求

32.8 rip_ctloutput函数

setsockopt和getsockopt函数会调用rip_ctloutput，它处理一个IP插口选项和8个用于多播选路的插口选项。

图32-17给出了rip_ctloutput函数的第一部分。

raw_ip.c

```
144 int
145 rip_ctloutput(op, so, level, optname, m)
146 int     op;
147 struct socket *so;
```

图32-17 rip_usrreq 函数：处理IP_HDRINCL 插口选项

```
148 int     level, optname;
149 struct mbuf **m;
150 {
151     struct inpcb *inp = sotoinpcb(so);
152     int     error;

153     if (level != IPPROTO_IP)
154         return (EINVAL);

155     switch (optname) {

156     case IP_HDRINCL:
157         if (op == PRCO_SETOPT || op == PRCO_GETOPT) {
158             if (m == 0 || *m == 0 || (*m)->m_len < sizeof(int))
159                         return (EINVAL);
160             if (op == PRCO_SETOPT) {
161                 if (*mtod(*m, int *))
162                             inp->inp_flags |= INP_HDRINCL;
163                 else
164                     inp->inp_flags &= ~INP_HDRINCL;
165                 (void) m_free(*m);
166             } else {
167                 (*m)->m_len = sizeof(int);
168                 *mtod(*m, int *) = inp->inp_flags & INP_HDRINCL;
169             }
170             return (0);
171         }
172         break;
```

raw_ip.c

图32-17 (续)

144-172 保存新选项值或者选项当前值的mbuf至少要能容纳一个整数。对于setsockopt系统调用，如果mbuf中的整数值非零，则设定该标志，否则清除它。对于getsockopt系统调用，mbuf中的返回值要么等于0，要么是非零的选项值。函数返回，以避免switch语句结束时处理其他IP选项。

图32-18给出了rip_ctloutput函数的最后一部分，处理8个多播选路插口选项。

raw_ip.c

```
173     case DVMRP_INIT:
174     case DVMRP_DONE:
175     case DVMRP_ADD_VIF:
176     case DVMRP_DEL_VIF:
177     case DVMRP_ADD_LGRP:
178     case DVMRP_DEL_LGRP:
179     case DVMRP_ADD_MRT:
180     case DVMRP_DEL_MRT:
                    /* shown in Figure 14.9 */
188     }
189     return (ip_ctloutput(op, so, level, optname, m));
190 }
```

raw_ip.c

图32-18 rip_usrreq 函数：处理多播选路插口选项

173-188 这8个插口选项只对 setsockopt系统调用有效，它们由图 14-9讨论的

ip_mrouter_cmd函数处理。

189 所有其他IP插口选项，如设定IP选项的IP_OPTIONS，则由ip_ctloutput处理。

32.9 小结

原始插口为IP主机提供3种功能。

1) 用于发送和接收ICMP和IGMP报文。

2) 支持应用进程构建自己的IP首部。

3) 允许应用进程支持基于IP的其他协议。

原始IP较为简单——只填充IP首部的有限几个字段——但它允许应用进程提供自己的IP首部。例如，调试程序就能发送任何类型的IP数据报。

原始IP输入提供了3种处理方式，能够选择性地接收进入的IP数据报。应用进程基于下列因素选择接收数据报：(1) 协议字段；(2) 源IP地址(由connect指明)；(3) 目的IP地址(由bind指明)。应用进程可以任意组合上述3种过滤条件。

习题

32.1 假定IP_HDRINCL插口选项未设定。如果socket的第三个参数等于0，rip_output填入IP首部协议字段(ip_p)的值是多少？如果socket的第三个参数等于IPPROTO_RAW (255)，rip_output填入该段(ip_p)的值又是多少？

32.2 应用进程创建了一个原始插口，协议值等于IPPROTO_RAW (255)。应用进程在这个插口上将收到什么类型的IP数据报？

32.3 应用进程创建了一个原始插口，协议值等于0。应用进程在这个插口上将收到什么类型的IP数据报？

32.4 修改rip_input，在适当情况下发送代码等于2 (协议不可达)的ICMP目的不可达报文。请注意，不要为rip_input正处理的ICMP或IGMP数据报生成一个差错。

32.5 如果应用进程希望生成自己的IP数据报，自己填充IP首部字段，可使用IP_HDRINCL选项置位的原始IP插口，或者采用BPF(第31章)，两种方法的区别是什么？

32.6 什么时候应用进程应该读取原始IP插口？什么时候读取BPF？