

第8章 IP：网际协议

8.1 引言

本章我们介绍 IP 分组的结构和基本的 IP 处理过程，包括输入、转发和输出。假定读者熟悉 IP 协议的基本操作，其他 IP 的背景知识见卷 1 的第 3、9 和 12 章。RFC 791 [Postel 1981a] 是 IP 的官方规范，RFC 1122 [Braden 1989a] 中有 RFC 791 的说明。

第 9 章将讨论选项的处理，第 10 章讨论分片和重装。图 8-1 显示了 IP 层常见的组织形式。

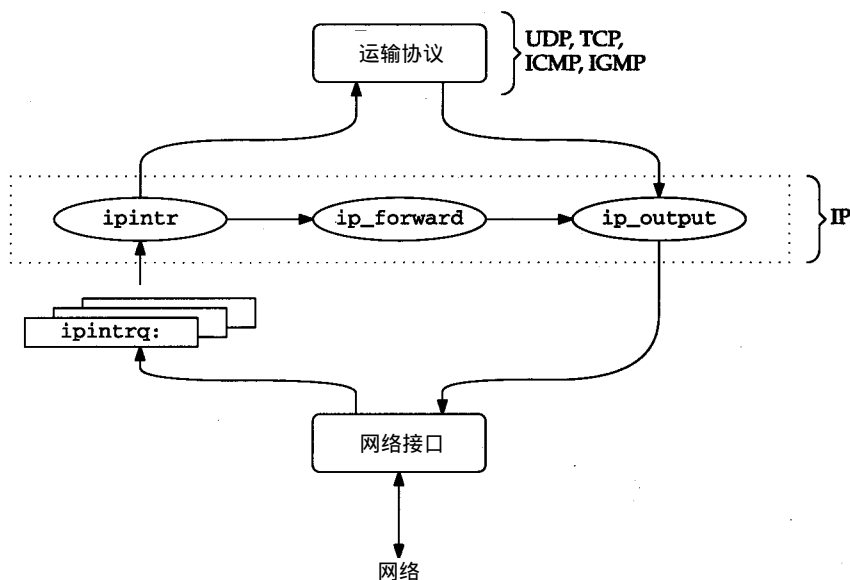


图8-1 IP层的处理

在第4章中，我们看到网络接口如何把到达的 IP 分组放到 IP 输入队列 `ipintrq` 中去，并如何调用一个软件中断。因为硬件中断的优先级比软件中断的要高，所以在发生一次软件中断之前，有的分组可能会被放到队列中。在软件中断处理中，`ipintr` 函数不断从 `ipintrq` 中移走和处理分组，直到队列为空。在最终的目的地，IP 把分组重装为数据报，并通过函数调用把该数据报直接传给适当的运输层协议。如果分组没有到达最后的目的地，并且如果主机被配置成一个路由器，则 IP 把分组传给 `ip_forward`。运输协议和 `ip_forward` 把要输出的分组传给 `ip_output`，由 `ip_output` 完成 IP 首部、选择输出接口以及在必要时对分组分片。最终的分组被传给合适的网络接口输出函数。

当产生差错时，IP 丢弃该分组，并在某些条件下向分组的源站发出一个差错报文。这些报文是 ICMP (第 11 章) 的一部分。Net/3 通过调用 `icmp_error` 发出 ICMP 差错报文，`icmp_error` 接收一个 `mbuf`，其中包含差错分组、发现的差错类型以及一个选项码，提供依赖于差错类型的附加信息。

本章我们讨论 IP 为什么以及何时发送 ICMP 报文，至于有关 ICMP 本身的详细讨论将在第 11 章进行。

8.2 代码介绍

本章讨论两个头文件和三个 C 文件。如图 8-2 所示。

文 件	描 述
net/route.h	路由入口
netinet/ip.h	IP 首部结构
netinet/ip_ input.c	IP 输入处理
netinet/ip_ output.c	IP 输出处理
netinet/ip_ cksum.c	Internet 检验和算法

图8-2 本章描述的文件

8.2.1 全局变量

在 IP 处理代码中出现了几个全局变量，见图 8-3。

变 量	数据类型	描 述
in_ifaddr	struct in_ ifaddr *	IP 地址清单
ip_defttl	int	IP 分组的默认 TTL
ip_id	int	赋给输出的 IP 分组的上一个 ID
ip_protox	int[]	IP 分组的分路矩阵
ipforwarding	int	系统是否转发 IP 分组？
ipforward_rt	struct route	大多数最近转发的路由的缓存
ipintrq	struct ifqueue	IP 输入队列
ipqmaxlen	int	IP 输入队列的最大长度
ipsendredirects	int	系统是否发送 ICMP 重定向？
ipstat	struct ipstat	IP 统计

图8-3 本章中引入的全局变量

8.2.2 统计量

IP 收集的所有统计量都放在图 8-4 描述的 ipstat 结构中。图 8-5 显示了由 netstat -s 命令得到的一些统计输出样本。统计是在主机启动 30 天后收集的。

ipstat 成员	描 述	SNMP 使用的
ips_badhlen	IP 首部长度无效的分组数	•
ips_badlen	IP 首部和 IP 数据长度不一致的分组数	•
ips_badoptions	在选项处理中发现差错的分组数	•
ips_badsum	检验和差错的分组数	•
ips_badvers	IP 版本不是 4 的分组数	•
ips_cantforward	目的站不可到达的分组数	•
ips_delivered	向高层交付的数据报数	•

图8-4 本章收集的统计

ipstat成员	描 述	SNMP使用的
ips_forward	转发的分组数	•
ips_fragdropped	分片丢失数(副本或空间不足)	•
ips_fragments	收到分片数	•
ips_fragtimeout	超时的分片数	•
ips_noproto	具有未知或不支持的协议的分组数	•
ips_reassembled	重装的数据报数	•
ips_tooshort	具有无效数据长度的分组数	•
ips_toosmall	无法包含IP分组的太小的分组数	•
ips_total	全部接收到的分组数	•
ips_cantfrag	由于不分片比特而丢弃的分组数	•
ips_fragmented	成功分片的数据报数	•
ips_localout	系统生成的数据报数(即没有转发的)	•
ips_noroute	丢弃的分组数——到目的地没有路由	•
ips_odropped	由于资源不足丢掉的分组数	•
ips_ofragments	为输出产生的分片数	•
ips_rawout	全部生成的原始ip分组数	
ips_redirectsent	已发送的重定向报文数	

图8-4 (续)

netstat -s 输出	ipstat 成员
27,881,978 total packets received	ips_total
6 bad header checksums	ips_badsum
9 with size smaller than minimum	ips_tooshort
14 with data size < data length	ips_toosmall
0 with header length < data size	ips_badhlen
0 with data length < header length	ips_badlen
0 with bad options	ips_badoptions
0 with incorrect version number	ips_badvers
72,786 fragments received	ips_fragments
0 fragments dropped (dup or out of space)	ips_fragdropped
349 fragments dropped after timeout	ips_fragtimeout
16,557 packets reassembled ok	ips_reassembled
27,390,665 packets for this host	ips_delivered
330,882 packets for unknown/unsupported protocol	ips_noproto
97,939 packets forwarded	ips_forward
6,228 packets not forwardable	ips_cantforward
0 redirects sent	ips_redirectsent
29,447,726 packets sent from this host	ips_localout
769 packets sent with fabricated ip header	ips_rawout
0 output packets dropped due to no bufs, etc.	ips_odropped
0 output packets discarded due to no route	ips_noroute
260,484 output datagrams fragmented	ips_fragmented
796,084 fragments created	ips_ofragments
0 datagrams that can't be fragmented	ips_cantfrag

图8-5 IP统计样本

ips_noproto的值很高,因为当没有进程准备接收报文时,它可对ICMP主机不可达报文进行计数。见第32.5节的详细讨论。

8.2.3 SNMP变量

图8-6显示了IP组和Net/3收集的统计中的SNMP变量之间的关系。

SNMP变量	Ipstat成员	描 述
ipDefaultTTL ipForwarding ipReasmTimeout	ip_defttl ipforwarding IPFRAGTTL	数据报的默认TTL(64跳) 系统是路由器吗? 分片的重装超时(30秒)
ipInReceives	ips_total	收到的全部IP分组数
ipInHdrErrors	ips_badsum+ ips_tooshort+ ips_toosmall+ ips_badhlen+ ips_badlen+ ips_badoptions+ ips_badvers	IP首部出错的分组数
ipInAddrErrors ipForwDatagrams ipReasmReqds ipReasmFails	ips_cantforward ips_forward ips_fragments ips_fragdropped+ ips_fragtimeout	由于错误交付而丢弃的IP分组数(ip_output也失败) 转发的IP分组数 收到的分片数 丢失的分片数
ipReasmOKs ipInDiscards ipInUnknownProtos ipInDelivers	ips_reassembled (未实现) ips_noproto ips_delivered	成功地重装的数据报数 由于资源限制而丢弃的数据报数 具有未知或不支持的协议的数据报数 交付到运输层的数据报数
ipOutRequests ipFragOKs ipFragFails ipFragCreates ipOutDiscards ipOutRoutes	ips_localout ips_fragmented ips_cantfrag ips_ofragments ips_odropped ips_noroute	由运输层产生的数据报数 分片成功的数据报数 由于不分片比特丢弃的IP分组数 为输出产生的分片数 由于资源短缺丢失的IP分组数 由于没有路由丢弃的IP分组数

图8-6 IP组中SNMP变量的例子

8.3 IP分组

为了更准确地讨论 Internet 协议处理，我们必须定义一些名词。图 8-7 显示了在不同的 Internet 层之间传递数据时用来描述数据的名词。

我们把传输协议交给 IP 的数据称为报文。典型的报文包含一个运输层首部 and 应用程序数据。图 8-7 所示的传输协议是 UDP。IP 在报文的首部前加上它自己的首部形成一个数据报。如果在选定的网络中，数据报的长度太大，IP 就把数据报分裂成几个分片，每个分片中含有它自己的 IP 首部和一段原来数据报的数据。图 8-7 显示了一个数据报被分成三个分片。

当提交给数据链路层进行传送时，一个 IP 分片或一个很小的无需分片的 IP 数据报称为分组。数据链路层在分组前面加上它自己的首部，并发送得到的帧。

IP 只考虑它自己加上的 IP 首部，对报文本身既不检查也不修改（除非进行分片）。图 8-8 显示了 IP 首部的结构。

图 8-8 包括 ip 结构（如图 8-9）中各成员的名字，Net/3 通过该结构访问 IP 首部。

47-67 因为在存储器中，比特字段的物理顺序依机器和编译器的不同而不同，所以由 #ifs 保证编译器按照 IP 标准排列结构成员。从而，当 Net/3 把一个 ip 结构覆盖到存储器中的一个 IP

分组上时，结构成员能够访问到分组中正确的比特。

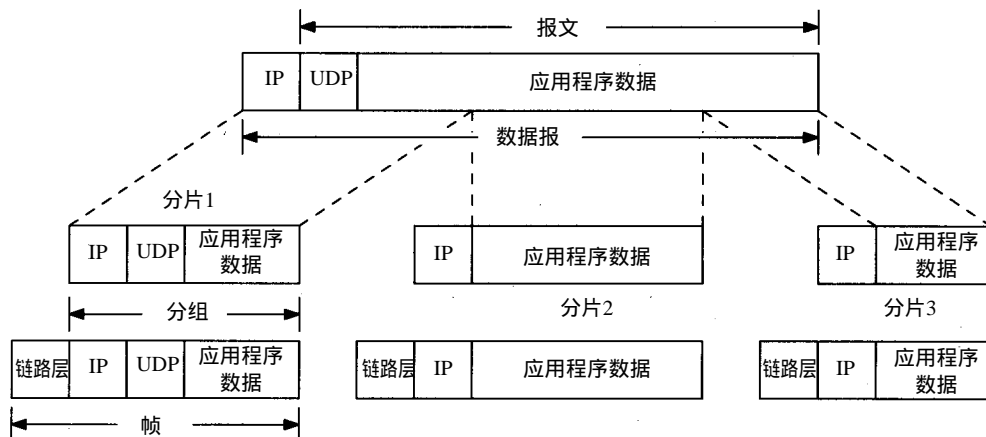


图8-7 帧、分组、分片、数据报和报文

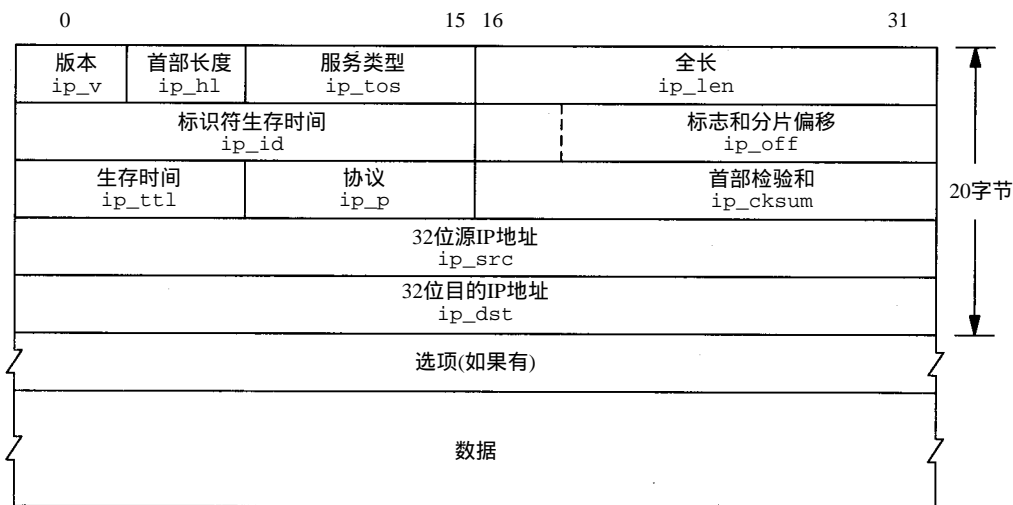


图8-8 IP数据报，包括ip结构名

```

40 /*
41  * Structure of an internet header, naked of options.
42  *
43  * We declare ip_len and ip_off to be short, rather than u_short
44  * pragmatically since otherwise unsigned comparisons can result
45  * against negative integers quite easily, and fail in subtle ways.
46  */
47 struct ip {
48 #if BYTE_ORDER == LITTLE_ENDIAN
49     u_char  ip_hl:4,          /* header length */
50     ip_v:4;                  /* version */
51 #endif
52 #if BYTE_ORDER == BIG_ENDIAN
53     u_char  ip_v:4,          /* version */
54     ip_hl:4;                /* header length */
55 #endif

```

ip.h

图8-9 ip结构

```

56     u_char  ip_tos;           /* type of service */
57     short   ip_len;          /* total length */
58     u_short  ip_id;          /* identification */
59     short    ip_off;         /* fragment offset field */
60 #define IP_DF 0x4000         /* dont fragment flag */
61 #define IP_MF 0x2000         /* more fragments flag */
62 #define IP_OFFMASK 0x1fff    /* mask for fragmenting bits */
63     u_char  ip_ttl;          /* time to live */
64     u_char  ip_p;            /* protocol */
65     u_short ip_sum;           /* checksum */
66     struct in_addr ip_src, ip_dst; /* source and dest address */
67 };

```

ip.h

图8-9 (续)

IP首部中包含IP分组格式、内容、寻址、路由选择以及分片的信息。

IP分组的格式由版本 `ip_v` 指定，通常为4；首部长度 `ip_hl`，通常以4字节单元度量；分组长度 `ip_len` 以字节为单位度量；传输协议 `ip_p` 生成分组内数据；`ip_sum` 是检验和，检测在发送中首部的变化。

标准的IP首部长度是20个字节，所以 `ip_hl` 必须大于或等于5。大于5表示IP选项紧跟在标准首部后。如 `ip_hl` 的最大值为15 ($2^4 - 1$)，允许最多40个字节的选项 ($20 + 40 = 60$)。IP数据报的最大长度为65535 ($2^{16} - 1$) 字节，因为 `ip_len` 是一个16 bit 的字段。图8-10是整个构成。

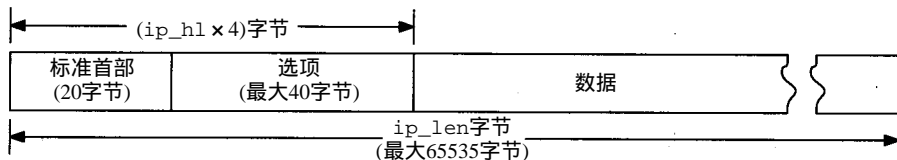


图8-10 有选项的IP分组构成

因为 `ip_hl` 是以4字节为单元计算的，所以IP选项必须常常被填充成4字节的倍数。

8.4 输入处理：ipintr函数

在第3、第4和第5章中，我们描述了示例的网络接口如何对到达的数据报排队以等待协议处理：

- 1) 以太网接口用以太网首部中的类型字段分路到达帧 (见4.3节)；
- 2) SLIP接口只处理IP分组，所以无需分用 (见5.3节)；
- 3) 环回接口在 `looutput` 函数中结合输入和输出处理，用目的地址中的 `sa_family` 成员对数据报分用 (见5.4节)。

在以上情况中，当接口把分组放到 `ipintrq` 上排队后，通过 `schednetisr` 调用一个软中断。当该软中断发生时，如果IP处理过程已经由 `schednetisr` 调度，则内核调用 `ipintr`。在调用 `ipintr` 之前，CPU的优先级被改变成 `splnet`。

8.4.1 ipintr 概观

`ipintr` 是一个大函数，我们将在4个部分中讨论：(1)对到达分组验证；(2)选项处理及转发；(3)分组重装；(4)分用。在 `ipintr` 中发生分组的重装，但比较复杂，我们将单独放在第

10章中讨论。图8-11显示了ipintr的整体结构。

```

100 void
101 ipintr()
102 {
103     struct ip *ip;
104     struct mbuf *m;
105     struct ipq *fp;
106     struct in_ifaddr *ia;
107     int hlen, s;
108
109     next:
110     /*
111      * Get next datagram off input queue and get IP header
112      * in first mbuf.
113      */
114     s = splimp();
115     IF_DEQUEUE(&ipintrq, m);
116     splx(s);
117     if (m == 0)
118         return;
119
120     /* input packet processing */
121     /* Figures 8.12, 8.13, 8.14, 8.15, and 8.16 */
122
123     goto next;
124 bad:
125     m_freem(m);
126     goto next;
127 }

```

ip_input.c

图8-11 ipintr 函数

100-117 标号next标识主要的分组处理循环的开始。ipintr从ipintrq中移走分组，并对之加以处理直到整个队列空为止。如果到函数最后控制失败，goto把控制权传回给next中最上面的函数。ipintr把分组阻塞在splimp内，避免当它访问队列时，运行网络的中断程序（例如slinput和ether_input）。

332-336 标号bad标识由于释放相关mbuf并返回到next中处理循环的开始而自动丢弃的分组。在整个ipintr中，都是跳到bad来处理差错。

8.4.2 验证

我们从图8-12开始：把分组从ipintrq中取出，验证它们的内容。损坏和有差错的分组被自动丢弃。

```

118 /*
119  * If no IP addresses have been set yet but the interfaces
120  * are receiving, can't do anything with incoming packets yet.
121  */
122 if (in_ifaddr == NULL)
123     goto bad;
124 ipstat.ips_total++;
125 if (m->m_len < sizeof(struct ip) &&

```

ip_input.c

图8-12 ipintr 函数

```

126         (m = m_pullup(m, sizeof(struct ip))) == 0) {
127     ipstat.ips_toosmall++;
128     goto next;
129 }
130 ip = mtod(m, struct ip *);
131 if (ip->ip_v != IPVERSION) {
132     ipstat.ips_badvers++;
133     goto bad;
134 }
135 hlen = ip->ip_hl << 2;
136 if (hlen < sizeof(struct ip)) {      /* minimum header length */
137     ipstat.ips_badhlen++;
138     goto bad;
139 }
140 if (hlen > m->m_len) {
141     if ((m = m_pullup(m, hlen)) == 0) {
142         ipstat.ips_badhlen++;
143         goto next;
144     }
145     ip = mtod(m, struct ip *);
146 }
147 if (ip->ip_sum == in_cksum(m, hlen)) {
148     ipstat.ips_badsum++;
149     goto bad;
150 }
151 /*
152  * Convert fields to host representation.
153  */
154 NTOHS(ip->ip_len);
155 if (ip->ip_len < hlen) {
156     ipstat.ips_badlen++;
157     goto bad;
158 }
159 NTOHS(ip->ip_id);
160 NTOHS(ip->ip_off);
161 /*
162  * Check that the amount of data in the buffers
163  * is as at least much as the IP header would have us expect.
164  * Trim mbufs if longer than we expect.
165  * Drop packet if shorter than we expect.
166  */
167 if (m->m_pkthdr.len < ip->ip_len) {
168     ipstat.ips_tooshort++;
169     goto bad;
170 }
171 if (m->m_pkthdr.len > ip->ip_len) {
172     if (m->m_len == m->m_pkthdr.len) {
173         m->m_len = ip->ip_len;
174         m->m_pkthdr.len = ip->ip_len;
175     } else
176         m_adj(m, ip->ip_len - m->m_pkthdr.len);
177 }

```

ip_input.c

图8-12 (续)

1. IP 版本

118-134 如果in_ifaddr表(见第6.5节)为空,则该网络接口没有指派IP地址,ipintr必须丢掉所有的IP分组;没有地址,ipintr就无法决定该分组是否要到该系统。通常这是一种

暂时情况，是当系统启动时，接口正在运行但还没有配置好时发生的。我们在 6.3 节中介绍了地址是如何分配的问题。

在 `ipintr` 访问任何 IP 首部字段之前，它必须证实 `ip_v` 是 4 (IPVERSION)。RFC 1122 要求某种实现丢弃那些具有无法识别版本号的分组而不回显信息。

Net/2 不检查 `ip_v`。目前大多数正在使用的 IP 实现，包括 Net/2，都是在 IP 的版本 4 之后产生的，因此无需区分不同 IP 版本的分组。因为目前正在对 IP 进行修正，所以不久将来的实现都将检查 `ip_v`。

IEN 119 [Forgie 1979] 和 RFC 1190 [Topolcic 1990] 描述了使用 IP 版本 5 和 6 的实验协议。版本 6 还被选为下一个正式的 IP 标准 (IPv6)。保留版本 0 和 15，其他的没有赋值。

在 C 中，处理位于一个无类型存储区域中数据的最简单的方法是：在该存储区域上覆盖一个结构，转而处理该结构中的各个成员，而不再对原始的字节进行操作。如第 2 章所言，`mbuf` 链把一个字节的逻辑序列，例如一个 IP 分组，储存在多个物理 `mbuf` 中，各 `mbuf` 相互连接在一个链表上。因为覆盖技术也可用于 IP 分组的首部，所以首部必须驻留在一段连续的存储区内 (也就是说，不能把首部分开放在不同的存储器缓存区)。

135-146 下面的步骤保证 IP 首部 (包括选项) 位于一段连续的存储器缓存区上：

- 如果在第一个 `mbuf` 中的数据小于一个标准的 IP 首部 (20 字节)，`m_pullup` 会重新把该标准首部放到一个连续的存储器缓存区上去。

链路层不太可能会把最大的 (60 字节) IP 首部分在两个 `mbuf` 中从而使用上面的 `m_pullup`。

- `ip_hl` 通过乘以 4 得到首部字节长度，并将其保存在 `hlen` 中。
- 如果 IP 分组首部的字节数长度 `hlen` 小于标准首部 (20 字节)，将是无效的并被丢弃。
- 如果整个首部仍然不在第一个 `mbuf` 中 (也就是说，分组包含了 IP 选项)，则由 `m_pullup` 完成其任务。

同样，这不一定是必需的。

检验和计算是所有 Internet 协议的重要组成。所有的协议均使用相同的算法 (由函数 `in_cksum` 完成)，但应用于分组的不同部分。对 IP 来说，检验和只保证 IP 的首部 (以及选项，如果有的话)。对传输协议，如 UDP 或 TCP，检验和覆盖了分组的数据部分和运输层首部。

2. IP 检验和

147-150 `ipintr` 把由 `in_cksum` 计算出来的检验和保存首部的 `ip_sum` 字段中。一个未被破坏的首部应该具有 0 检验和。

正如我们将在 8.7 节中看到的，在计算到达分组的检验和之前，必须对 `ip_sum` 清零。通过把 `in_cksum` 中的结果储存在 `ip_sum` 中，就为分组转发作好了准备 (尽管还没有减小 TTL)。`ip_output` 函数不依赖这项操作；它为转发的分组重新计算检验和。

如果结果非零，则该分组被自动丢弃。我们将在 8.7 节中详细讨论 `in_cksum`。

3. 字节顺序

151-160 Internet标准在指定协议首部中多字节整数值的字节顺序时非常小心。NTOHS把IP首部中所有16 bit的值从网络字节序转换成主机字节序：分组长度(ip_len)、数据报标识符(ip_id)和分片偏移(ip_off)。如果两种格式相同，则NTOHS是一个空的宏。在这里就转换成主机字节序，以避免Net/3每次检查该字段时必须进行一次转换。

4. 分组长度

161-177 如果分组的逻辑长度(ip_len)比储存在mbuf中的数据量(m_pkthdr.len)大，并且有些字节被丢失了，就必须丢弃该分组。如果mbuf比分组大，则去掉多余的字节。

丢失字节的一个常见原因是因为数据到达某个没有或只有很少缓存的串口设备，例如许多个人计算机。设备丢弃到达的字节，而IP丢弃最后的分组。

多余的字节可能产生，如在某个以太网设备上，当一个IP分组的大小比以太网要求的最小长度还小时。发送该帧时加上的多余字节就在这里被丢掉。这就是为什么IP分组的长度被保存在首部的原因之一；IP允许链路层填充分组。

现在，有了完整的IP首部，分组的逻辑长度和物理长度相同，检验和表明分组的首部无损地到达。

8.4.3 转发或不转发

图8-13显示了ipintr的下一部分，调用ip_dooptions(见第9章)来处理IP选项，然后决定分组是否到达它最后的目的地。如果分组没有到达最后目的地，Net/3会尝试转发该分组(如果系统被配置成路由器)。如果分组到达最后目的地，就被交付给合适的运输层协议。

```

178      /*
179      * Process options and, if not destined for us,
180      * ship it on. ip_dooptions returns 1 when an
181      * error was detected (causing an icmp message
182      * to be sent and the original packet to be freed).
183      */
184      ip_nhops = 0;                /* for source routed packets */
185      if (hlen > sizeof(struct ip) && ip_dooptions(m))
186          goto next;

187      /*
188      * Check our list of addresses, to see if the packet is for us.
189      */
190      for (ia = in_ifaddr; ia; ia = ia->ia_next) {
191 #define satosin(sa) ((struct sockaddr_in *) (sa))

192          if (IA_SIN(ia)->sin_addr.s_addr == ip->ip_dst.s_addr)
193              goto ours;

194          /* Only examine broadcast addresses for the receiving interface */
195          if (ia->ia_ifp == m->m_pkthdr.rcvif &&
196              (ia->ia_ifp->if_flags & IFF_BROADCAST)) {
197              u_long t;

198              if (satosin(&ia->ia_broadaddr)->sin_addr.s_addr ==
199                  ip->ip_dst.s_addr)
200                  goto ours;
201              if (ip->ip_dst.s_addr == ia->ia_netbroadcast.s_addr)

```

图8-13 续ipintr

```

202         goto ours;
203     /*
204     * Look for all-0's host part (old broadcast addr),
205     * either for subnet or net.
206     */
207     t = ntohl(ip->ip_dst.s_addr);
208     if (t == ia->ia_subnet)
209         goto ours;
210     if (t == ia->ia_net)
211         goto ours;
212     }
213 }

```

```

/* multicast code (Figure 12-39) */

```

```

258     if (ip->ip_dst.s_addr == (u_long) INADDR_BROADCAST)
259         goto ours;
260     if (ip->ip_dst.s_addr == INADDR_ANY)
261         goto ours;
262     /*
263     * Not for us; forward if possible and desirable.
264     */
265     if (ipforwarding == 0) {
266         ipstat.ips_cantforward++;
267         m_freem(m);
268     } else
269         ip_forward(m, 0);
270     goto next;
271     ours:

```

—ip_input.c

图8-13 (续)

1. 选项处理

178-186 通过对ip_nhops(见9.6节)清零, 丢掉前一个分组的原路由。如果分组首部大于默认首部, 它必然包含由 ip_dooptions处理的选项。如果 ip_dooptions返回0, ipintr将继续处理该分组; 否则, ip_dooptions通过转发或丢弃分组完成对该分组的处理, ipintr可以处理输入队列中的下一个分组。我们把对选项的进一步讨论放到第9章进行。

处理完选项后, ipintr通过把IP首部内的ip_dst与配置的所有本地接口的IP地址比较, 以决定分组是否已到达最终目的地。ipintr必须考虑与接口相关的几个广播地址、一个或多个单播地址以及任意个多播地址。

2. 最终目的地

187-261 ipintr通过遍历in_ifaddr(图6-5), 配置好的Internet地址表, 来决定是否有与分组的目的地址的匹配。对清单中的每个 in_ifaddr结构进行一系列的比较。要考虑4种常见的情况:

- 与某个接口地址的完全匹配(图8-14中的第一行),
- 与某个与接收接口相关的广播地址的匹配(图8-14的中间4行),
- 与某个与接收接口相关的多播组之一的匹配(图12-39), 或
- 与两个受限的广播地址之一的匹配(图8-14的最后一行)。

图8-14显示的是当分组到达我们的示例网络里的主机 sun上的以太网接口时要测试的地址，将在第12章中讨论的多播地址除外。









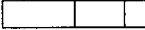
变量	以太网	SLIP	环回	线路 (图8.13)
ia_addr	 140.252.13.33	 140.252.1.29	 127.0.0.1	192-193
ia_broadaddr	 140.252.13.224			198-200
ia_netbroadcast	 140.252.255.255			201-202
ia_subnet	 140.252.13.32			207-209
ia_net	 140.252.0.0			210-211
INADDR_BROADCAST		 255.255.255.255		258-259
INADDR_ANY		 0.0.0.0		260-261

图8-14 为判定分组是否到达最终目的地进行的比较

对ia_subnet、ia_net和INADDR_ANY的测试不是必需的，因为它们表示的是4.2BSD使用的已经过时的广播地址。但不幸的是，许多 TCP/IP实现是从4.2BSD派生而来的，所以，在某些网络中能够识别出这些旧广播地址可能十分重要。

3. 转发

262-271 如果ip_dst与所有地址都不匹配，分组还没有到达最终目的地。如果还没有设置ipforwarding，就丢弃分组。否则，ip_forward尝试把分组路由到它的最终目的地。

当分组到达的某个地址不是目的地址指定的接口时，主机会丢掉该分组。在这种情况下，Net/3将搜索整个in_ifaddr表；只考虑那些分配给接收接口的地址。RFC 1122 称此为强端系统(strong end system)模型。

对多主机而言，很少出现分组到达接口地址与其目的地址不对应的情况，除非配置了特定的主机路由。主机路由强迫相邻的路由器把多主机作为分组的下一跳路由器。弱端系统(weak end system)模型要求该主机接收这些分组。实现人员可以随意选择两种模型。Net/3实现弱端系统模型。

8.4.4 重装和分用

最后，我们来看ipintr的最后一部分(图8-15)，在这里进行重装和分用。我们略去了重装的代码，推迟到第10章讨论。当无法重装完全的数据报时，略去的代码将把指针ip设成空。否则，ip指向一个已经到达目的地的完整数据报。

运输分用

325-332 数据报中指定的协议被ip_p用ip_protox数组(图7-22)映射到inetsw数组的下

标。ipintr调用选定的protosw结构中的pr_input函数来处理数据报包含的运输报文。当pr_input返回时，ipintr继续处理ipintrq中的下一个分组。

注意，运输层对分组的处理发生在ipintr处理循环的内部。在IP和传输协议之间没有到达分组的排队，这与TCP/IP中SVR4流实现的排队不同。

—————ip_input.c

```

325  /*
326   * If control reaches here, ip points to a complete datagram.
327   * Otherwise, the reassembly code jumps back to next (Figure 8.11)
328   * Switch out to protocol's input routine.
329   */
330  ipstat.ips_delivered++;
331  (*inetsw[ip_protox[ip->ip_p]].pr_input) (m, hlen);
332  goto next;

```

—————ip_input.c

图8-15 续ipintr

8.5 转发：ip_forward函数

到达非最终目的地系统的分组需要被转发。只有当ipforwarding非零(6.1节)或当分组中包含源路由(9.6节)时，ipintr才调用实现转发算法的ip_forward函数。当分组中包含源路由时，ip_dooptions调用ip_forward，并且第2个参数srcrt设为1。

ip_forward通过图8-16中显示的route结构与路由表接口。

—————route.h

```

46 struct route {
47     struct rtentry *ro_rt;      /* pointer to struct with information */
48     struct sockaddr ro_dst;     /* destination of this route */
49 };

```

—————route.h

图8-16 route 结构

46-49 route结构有两个成员：ro_rt，指向rtentry结构的指针；ro_dst，一个sockaddr结构，指定与ro_rt所指的路由项相关的目的地。目的地是在内核的路由表中用来查找路由信息的关键字。第18章对rtentry结构和路由表有详细的描述。

我们分两部分讨论ip_forward。第一部分确定允许系统转发分组，修改IP首部，并为分组选择路由。第二部分处理ICMP重定向报文，并把分组交给ip_output进行发送。见图8-17。

1. 分组适合转发吗

867-871 ip_froward的第1个参数是指向一个mbuf链的指针，该mbuf中包含了要被转发的分组。如果第2个参数srcrt为非零，则分组由于源路由选项(见9.6节)正在被转发。

879-884 if语句识别并丢弃以下分组。

• 链路层广播

任何支持广播的网络接口驱动器必须为收到的广播分组把M_BCAST标志置位。如果分组寻址是到以太网广播地址，则ether_input(图4-13)就把M_BCAST置位。不转发链路层的广播分组。

RFC 1122不允许以链路层广播的方式发送一个寻址到单播 IP地址的分组，并在这里将该分组丢掉。

- 环回分组

对寻址到环回网络的分组，`in_canforward`返回0。这些分组将被 `ipintr`提交给 `ip_forward`，因为没有正确配置反馈接口。

- 网络0和E类地址

对这些分组，`in_canforward`返回0。这些目的地址是无效的，而且因为没有主机接收这些分组，所以它们不应该继续在网络中流动。

- D类地址

寻址到D类地址的分组应该由多播函数 `ip_mforward`而不是由 `ip_forward`处理。`in_canforward`拒绝D类(多播)地址。

RFC 791 规定处理分组的所有系统都必须把生存时间 (TTL)字段至少减去1，即使TTL是以秒计算的。由于这个要求，TTL通常被认为是IP分组在被丢掉之前能经过的跳的个数的界限。从技术角度说，如果路由器持有分组超过1秒，就必须把 `ip_ttl`减去多于1。

```

867 void
868 ip_forward(m, srcrt)
869 struct mbuf *m;
870 int      srcrt;
871 {
872     struct ip *ip = mtod(m, struct ip *);
873     struct sockaddr_in *sin;
874     struct rtentry *rt;
875     int      error, type = 0, code;
876     struct mbuf *mcopy;
877     n_long  dest;
878     struct ifnet *destifp;
879
880     dest = 0;
881     if (m->m_flags & M_BCAST || in_canforward(ip->ip_dst) == 0) {
882         ipstat.ips_cantforward++;
883         m_freem(m);
884         return;
885     }
886     HTONS(ip->ip_id);
887     if (ip->ip_ttl <= IPTTLDEC) {
888         icmp_error(m, ICMP_TIMXCEED, ICMP_TIMXCEED_INTRANS, dest, 0);
889         return;
890     }
891     ip->ip_ttl -= IPTTLDEC;
892
893     sin = (struct sockaddr_in *) &ipforward_rt.ro_dst;
894     if ((rt = ipforward_rt.ro_rt) == 0 ||
895         ip->ip_dst.s_addr != sin->sin_addr.s_addr) {
896         if (ipforward_rt.ro_rt) {
897             RTFREE(ipforward_rt.ro_rt);
898             ipforward_rt.ro_rt = 0;
899         }
900         sin->sin_family = AF_INET;
901         sin->sin_len = sizeof(*sin);
902         sin->sin_addr = ip->ip_dst;
903         rtalloc(&ipforward_rt);
904     }

```

—ip_input.c

图8-17 ip_forward 函数：路由选择

```

902         if (ipforward_rt.ro_rt == 0) {
903             icmp_error(m, ICMP_UNREACH, ICMP_UNREACH_HOST, dest, 0);
904             return;
905         }
906         rt = ipforward_rt.ro_rt;
907     }
908     /*
909     * Save at most 64 bytes of the packet in case
910     * we need to generate an ICMP message to the src.
911     */
912     mcopy = m_copy(m, 0, imin((int) ip->ip_len, 64));
913     ip_ifmatrix[rt->rt_ifp->if_index +
914                if_index * m->m_pkthdr.rcvif->if_index]++;

```

ip_input.c

图8-17 (续)

这就产生了一个问题：在 Internet 上，最长的路径有多长？这个度量称为网络的直径(diameter)。除了通过实验外无法知道直径的大小。[Olivier 1994]中有37跳的路径。

2. 减小TTL

885-890 由于转发时不再需要分组的标识符，所以标识符又被转换回网络字节序。但是当 ip_forward 发送包含无效 IP 首部的 ICMP 差错报文时，分组的标识符又应该是正确的顺序。

Net/3 漏做了对已被 ipintr 转换成主机字节序的 ip_len 的转换。作者注意到在大头机器上，这不会产生问题，因为从未对字节进行过转换。但在小头机器如 386 上，这个小的漏洞允许交换了字节的值在 ICMP 差错中的 IP 首部中。返回从运行在 386 上的 SVR4(可能是 Net/1 码)和 AIX3.2(4.3BSD Reno 码)返回的 ICMP 分组中可以观察到这个小的漏洞。

如果 ip_ttl 达到 1(IPTTLDEC)，则向发送方返回一个 ICMP 超时报文，并丢掉该分组。否则，ip_forward 把 ip_ttl 减去 IPTTLDEC。

系统不接受 TTL 为 0 的 IP 数据报，但 Net/3 在即使出现这种情况时也能生成正确的 ICMP 差错，因为 p_ttl 是在分组被认为是在本地交付之后和被转发之前检测的。

3. 定位下一跳

891-907 IP 转发算法把最近的路由缓存在全局 route 结构的 ipforward_rt 中，在可能时应用于当前分组。研究表明连续分组趋向于同一目的地址 ([Jain 和 Routhier 1986] 和 [Mogul 1991])，所以这种向后一个 (one-behind) 的缓存使路由查询的次数最少。如果缓存为空 (ipforward_rt) 或当前分组的目的地不是 ipforward_rt 中的路由，就取消前面的路由，ro_dst 被初始化成新的目的地，rtalloc 为当前分组的目的地找一个新路由。如果找不到路由，则返回一个 ICMP 主机不可达差错，并丢掉该分组。

908-914 由于在产生差错时，ip_output 要丢掉分组，所以 m_copy 复制分组的前 64 个字节，以便 ip_forward 发送 ICMP 差错报文。如果调用 m_copy 失败，ip_forward 并不终止。在这种情况下，不发送差错报文。ip_ifmatrix 记录在接口之间进行路由选择的分组的个数。具有接收和发送接口索引的计数器是递增的。

重定向报文

当主机错误地选择某个路由器作为分组的第一跳路由器时，该路由器向源主机返回一个 ICMP 重定向报文。IP 网络互连模型假定主机相对地并不知道整个互联网的拓扑结构，把维护正确路由选择的责任交给路由器。路由器发出重定向报文是向主机表明它为分组选择了一个不正确的路由。我们用图 8-18 说明重定向报文。

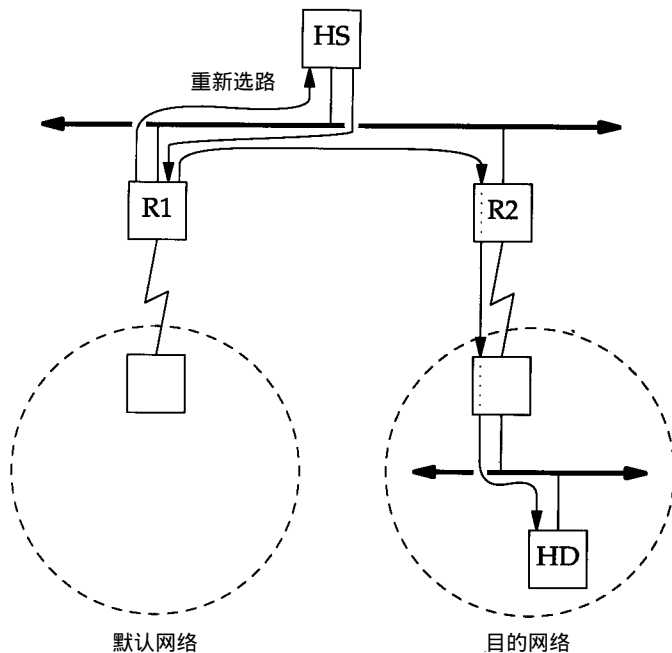


图8-18 路由器R1重定向主机HS使用路由器R2到达HD

通常，管理员对主机的配置是：把到远程网络的分组发送到某个默认路由器上。在图 8-18 中，主机 HS 上 R1 被配置成它的默认路由器。当 HS 首次向 HD 发送分组时，它不知道 R2 是合适的选择，而把分组发给 R1。R1 识别出差错，就把分组转发给 R2，并向 HS 发回一个重定向报文。接收到重定向报文后，HS 更新它的路由表，下一次发往 HD 的分组就直接发给 R2。

RFC 1122 推荐只有路由器才发重定向报文，而主机在接收到 ICMP 重定向报文后必须更新它们的路由表 (11.8 节)。因为 Net/3 只在系统被配置成路由器时才调用 `ip_forward`，所以 Net/3 采用 RFC 1122 的推荐。

在图 8-19 中，`ip_forward` 决定是否发重定向报文。

1. 在接收接口上离开吗

915-929 路由器识别重定向情况的规则很复杂。首先，只有在同一接口 (`rt_ifp` 和 `rcvif`) 上接收或重发分组时，才能应用重定向。其次，被选择的路由本身必须没有被 ICMP 重定向报文创建或修改过 (`RTF_DYNAMIC | RTF_MODIFIED`)，而且该路由也不能是到默认目的地的 (0.0.0.0)。这就保证系统在未授权时不会生成路由选择信息，并且不与其他系统共享自己的默认路由。

通常，路由选择协议使用特殊目的地址 0.0.0.0 定位默认路由。当到某目的地的某个路由不能使用时，与目的地 0.0.0.0 相关的路由就把分组定向到一个默认路由器上。

第18章对默认路由有详细的讨论。

全局整数 `ip_sendredirects` 指定系统是否被授权发送重定向（第8.9节），`ip_sendredirects` 的默认值为1。当传给 `ip_forward` 的参数 `srcrt` 指明系统是对分组路由选择的源时，禁止系统重定向，因为假定源主机要覆盖中间路由器的选择。

```

915  /*
916  * If forwarding packet is using same interface that it came in on,
917  * perhaps should send a redirect to sender to shortcut a hop.
918  * Only send redirect if source is sending directly to us,
919  * and if packet was not source routed (or has any options).
920  * Also, don't send redirect if forwarding using a default route
921  * or a route modified by a redirect.
922  */
923  #define satoisin(sa) ((struct sockaddr_in *) (sa))
924  if (rt->rt_ifp == m->m_pkthdr.rcvif &&
925      (rt->rt_flags & (RTF_DYNAMIC | RTF_MODIFIED)) == 0 &&
926      satoisin(rt_key(rt))->sin_addr.s_addr != 0 &&
927      ip_sendredirects && !srcrt) {
928  #define RTA(rt) ((struct in_ifaddr *) (rt->rt_ifa))
929      u_long src = ntohl(ip->ip_src.s_addr);

930      if (RTA(rt) &&
931          (src & RTA(rt)->ia_subnetmask) == RTA(rt)->ia_subnet) {
932          if (rt->rt_flags & RTF_GATEWAY)
933              dest = satoisin(rt->rt_gateway)->sin_addr.s_addr;
934          else
935              dest = ip->ip_dst.s_addr;
936          /* Router requirements says to only send host redirects */
937          type = ICMP_REDIRECT;
938          code = ICMP_REDIRECT_HOST;
939      }
940  }

```

ip_input.c

图8-19 ip_forward (续)

2. 发送重定向吗

930-931 这个测试决定分组是否产生于本地子网。如果源地址的子网掩码位和输出接口的地址相同，则两个地址位于同一 IP 网络中。如果源接口和输出的接口位于同一网络中，则该系统就不应该接收这个分组，因为源站可能已经把分组发给正确的第一跳路由器了。ICMP 重定向报文告诉主机正确的第一跳目的地。如果分组产生于其他子网，则前一系统是个路由器，这个系统就不应该发重定向报文；差错由路由选择协议纠正。

在任何情况下，都要求路由器忽略重定向报文。尽管如此，当 `ipforwarding` 被置位时（也就是说，当它被配置成路由器时），Net/3 并不丢掉重定向报文。

3. 选择合适的路由器

932-940 ICMP 重定向报文中包含正确的下一个系统的地址，如果目的主机不在直接相连的网络上时，该地址是一个路由器的地址；当目的主机在直接相连的网络中时，该地址是主机地址。

RFC 792 描述了重定向报文的 4 种类型：(1) 网络；(2) 主机；(3) TOS 和网络；(4) TOS 和主机。RFC 1009 推荐在任何时候都不发送网络重定向报文，因为无法保证接收到重定向报文的主机能为目的网络找到合适的子网掩码。RFC 1122 推荐主机把网络重定向看作是主机重定向，

以避免二义性。Net/3只发送主机重定向报文，并省略所有对 TOS的考虑。在图 8-20中，ipintr把分组和所有的ICMP报文提交给链路层。

```

941     error = ip_output(m, (struct mbuf *) 0, &ipforward_rt,
942                       IP_FORWARDING | IP_ALLOWBROADCAST, 0);
943     if (error)
944         ipstat.ips_cantforward++;
945     else {
946         ipstat.ips_forward++;
947         if (type)
948             ipstat.ips_redirectsent++;
949         else {
950             if (mcopy)
951                 m_freem(mcopy);
952             return;
953         }
954     }
955     if (mcopy == NULL)
956         return;
957     destifp = NULL;
958
959     switch (error) {
960     case 0:                /* forwarded, but need redirect */
961         /* type, code set above */
962         break;
963
964     case ENETUNREACH:      /* shouldn't happen, checked above */
965     case EHOSTUNREACH:
966     case ENETDOWN:
967     case EHOSTDOWN:
968     default:
969         type = ICMP_UNREACH;
970         code = ICMP_UNREACH_HOST;
971         break;
972
973     case EMSGSIZE:
974         type = ICMP_UNREACH;
975         code = ICMP_UNREACH_NEEDFRAG;
976         if (ipforward_rt.ro_rt)
977             destifp = ipforward_rt.ro_rt->rt_ifp;
978         ipstat.ips_cantfrag++;
979         break;
980
981     case ENOBUFS:
982         type = ICMP_SOURCEQUENCH;
983         code = 0;
984         break;
985     }
986     icmp_error(mcopy, type, code, dest, destifp);
987 }

```

图8-20 ip_forward(续)

重定向报文的标准化是在子网化之前，在一个非子网化的互联网中，网络重定向很有用，但在一个子网化的互联网中，由于重定向报文中没有有关子网掩码的信息，所以容易产生二义性。

4. 转发分组

941-954 现在，`ip_forward`有一个路由，并决定是否需要 ICMP重定向报文。`ip_output`把分组发送到路由`ipforward_rt`所指定的下一跳。`IP_ALLOWBROADCAST`标志位允许被转发分组是个到某局域网的广播。如果 `ip_output`成功，并且不需要发送任何重定向报文，则丢掉分组的前64字节，`ip_forward`返回。

5. 发送ICMP差错报文？

955-983 `ip_forward`可能会由于 `ip_output`失败或重定向而发送ICMP报文。如果没有原始分组的复制（可能当要复制时，曾经缓存不足），则无法发送重定向报文，`ip_forward`返回。如果有重定向，`type`和`code`以前又被置位，但如果 `ip_output`失败，`switch`语句基于从`ip_output`返回的值重新设置新的ICMP类型和码值。`icmp_error`发送该报文。来自失败的`ip_output`ICMP报文将覆盖任何重定向报文。

处理来自`ip_output`的差错的`switch`语句非常重要。它把本地差错翻译成适当的ICMP差错报文，并返回给分组的源站。图 8-21对差错作了总结。第 11章更详细地描述了ICMP报文。

当`ip_output`返回`ENOBUFS`时，Net/3通常生成ICMP源站抑制报文。Router Requirements(路由器需求)RFC [Almquist和Kastenholz 1994]不赞成源站抑制并要求路由器不产生这种报文。

来自ip_output的差错码	生成的ICMP报文	描 述
EMSGSIZE	ICMP_UNREACH_NEEDFRAG	对所选接口来说，发出的分组太大，并且禁止分片（第10章）
ENOBUFS	ICMP_SOURCEQUENCH	接口队列满或内核运行内存不足。本报文向源主机指示降低数据率
EHOSTUNREACH	ICMP_UNREACH_HOST	找不到到主机的路由
ENETDOWN		路由指明的输出接口没在运行
EHOSTDOWN		接口无法把分组发给选定的主机
default		所有不识别的差错均作为 ICMP_UNREACH_HOST差错报告

图8-21 来自ip_output 的差错

8.6 输出处理：ip_output函数

IP输出代码从两处接收分组：`ip_forward`和运输协议（图8-1）。让`inetsw[0].pr_output`能访问到IP输出操作似乎很有道理，但事实并非如此。标准的Internet传输协议（ICMP、IGMP、UDP和TCP）直接调用`ip_output`，而不查询`inetsw`表。对标准Internet传输协议而言，`protosw`结构不必具有一般性，因为调用函数并不是在与协议无关的情况下接入IP的。在第20章中，我们将看到与协议无关的路由选择插口调用`pr_output`接入IP。

我们分三个部分描述`ip_output`：

- 首部初始化；
- 路由选择；和
- 源地址选择和分片。

8.6.1 首部初始化

图8-22显示了ip_output的第一部分，把选项与外出的分组合并，完成传输协议提交（不是ip_forward提交的）的分组首部。

44-59 传给ip_output的参数包括：m0，要发送的分组；opt，包含的IP选项；ro，缓存的到目的地的路由；flags，见图8-23；imo，指向多播选项的指针，见第12章。

IP_FORWARDING被ip_forward和ip_mforward（多播分组转发）设置，并禁止ip_output重新设置任何IP首部字段。

```
44 int
45 ip_output(m0, opt, ro, flags, imo)
46 struct mbuf *m0;
47 struct mbuf *opt;
48 struct route *ro;
49 int flags;
50 struct ip_moptions *imo;
51 {
52     struct ip *ip, *mhip;
53     struct ifnet *ifp;
54     struct mbuf *m = m0;
55     int hlen = sizeof(struct ip);
56     int len, off, error = 0;
57     struct route iproute;
58     struct sockaddr_in *dst;
59     struct in_ifaddr *ia;

60     if (opt) {
61         m = ip_insertoptions(m, opt, &hlen);
62         hlen = len;
63     }
64     ip = mtod(m, struct ip *);
65     /*
66      * Fill in IP header.
67      */
68     if ((flags & (IP_FORWARDING | IP_RAWOUTPUT)) == 0) {
69         ip->ip_v = IPVERSION;
70         ip->ip_off &= IP_DF;
71         ip->ip_id = htons(ip_id++);
72         ip->ip_hl = hlen >> 2;
73         ipstat.ips_localout++;
74     } else {
75         hlen = ip->ip_hl << 2;
76     }
```

图8-22 函数ip_output

标 志	描 述
IP_FORWARDING	这是一个转发过的分组
IP_ROUTE_TO_IF	忽略路由表，直接路由到接口
IP_ALLOW_BROADCAST	允许发送广播分组
IP_RAWOUTPUT	包含一个预构IP首部的分组

图8-23 ip_output : flag 值

send、sendto和sendmsg的MSG_DONTROUTE标志使IP_ROUTETOIF有效，并进行一次写操作(见16.4)，而SO_DONTROUTE插口选项使IP_ROUTETOIF有效，并在某个特定插口上进行任意的写操作(见8.8节)。该标志被传输协议传给ip_output。

IP_ALLOWBROADCAST标志可以被SO_BROADCAST插口选项(见8.8节)设置，但只被UDP提交。原来的IP默认地设置IP_ALLOWBROADCAST。TCP不支持广播，所以IP_ALLOWBROADCAST不能被TCP提交给ip_output。不存在广播的预请求标志。

1. 构造IP首部

60-73 如果调用程序提供任何IP选项，它们将被ip_insetoptions(见9.8节)与分组合并，并返回新的首部长度。

我们将在8.8节中看到，进程可以设置IP_OPTIONS插口选项来为一个插口指定IP选项。插口的运输层(TCP或UDP)总是把这些选项提交给ip_output。

被转发分组(IP_FORWARDING)或有预构首部(IP_RAWOUTPUT)分组的IP首部不能被ip_output修改。任何其他分组(例如，产生于这个主机的UDP或TCP分组)需要有几个IP首部字段被初始化。ip_output把ip_v设置成4(IPVERSION)，把DF位需要的ip_off清零，并设置成调用程序提供的值(见第10章)，给来自全局整数的ip->ip_id赋一个唯一的标识符，把ip_id加1。ip_id是在协议初始化时由系统时钟设置的(见7.8节)。ip_hl被设置成用32 bit字度量的首部长度的。

IP首部的其他字段——长度、偏移、TTL、协议、TOS和目的地址——已经被传输协议初始化了。源地址可能没被设置，因为是在确定了到目的地的路由后选择的(图8-25)。

2. 分组已经包括首部

74-76 对一个已转发的分组(或一个有首部的原始IP分组)，首部长度的(以字节数度量)被保存在hlen中，留给将来分片算法使用。

8.6.2 路由选择

在完成IP首部后，ip_output的下一个任务就是确定一条到目的地的路由。见图8-24所示。

```

77      /*
78      * Route packet.
79      */
80      if (ro == 0) {
81          ro = &iproute;
82          bzero((caddr_t) ro, sizeof(*ro));
83      }
84      dst = (struct sockaddr_in *) &ro->ro_dst;
85      /*
86      * If there is a cached route,
87      * check that it is to the same destination
88      * and is still up. If not, free it and try again.
89      */
90      if (ro->ro_rt && ((ro->ro_rt->rt_flags & RTF_UP) == 0 ||
91          dst->sin_addr.s_addr != ip->ip_dst.s_addr)) {
92          RTFREE(ro->ro_rt);
93          ro->ro_rt = (struct rtable *) 0;
94      }

```

图8-24 ip_output (续)

```

95     if (ro->ro_rt == 0) {
96         dst->sin_family = AF_INET;
97         dst->sin_len = sizeof(*dst);
98         dst->sin_addr = ip->ip_dst;
99     }
100    /*
101     * If routing to interface only,
102     * short circuit routing lookup.
103     */
104    #define ifatoia(ifa)    ((struct in_ifaddr *) (ifa))
105    #define sintosa(sin)    ((struct sockaddr *) (sin))
106    if (flags & IP_ROUTETOIF) {
107        if ((ia = ifatoia(ifa_ifwithdstaddr(sintosa(dst)))) == 0 &&
108            (ia = ifatoia(ifa_ifwithnet(sintosa(dst)))) == 0) {
109            ipstat.ips_noroute++;
110            error = ENETUNREACH;
111            goto bad;
112        }
113        ifp = ia->ia_ifp;
114        ip->ip_ttl = 1;
115    } else {
116        if (ro->ro_rt == 0)
117            rtalloc(ro);
118        if (ro->ro_rt == 0) {
119            ipstat.ips_noroute++;
120            error = EHOSTUNREACH;
121            goto bad;
122        }
123        ia = ifatoia(ro->ro_rt->rt_ifa);
124        ifp = ro->ro_rt->rt_ifp;
125        ro->ro_rt->rt_use++;
126        if (ro->ro_rt->rt_flags & RTF_GATEWAY)
127            dst = (struct sockaddr_in *) ro->ro_rt->rt_gateway;
128    }

```

ip_output.c

图8-24 (续)

1. 验证高速缓存中的路由

77-99 ip_output可能把一条在高速缓存中的路由作为ro参数来提供。在第24章中，我们将看到UDP和TCP维护一个与各插口相关的路由缓存。如果没有路由，则ip_output把ro设置成指向临时route结构iproute。

如果高速缓存中的目的地不是去当前分组的目的地，就把该路由丢掉，新的目的地址放在dst中。

2. 旁路路由选择

100-114 调用方可通过设置IP_ROUTETOIF标志(见8.8节)禁止对分组进行路由选择。ip_output必须找到一个与分组中指定目的地网络直接相连的接口。ifa_ifwithdstaddr搜索点到点接口，而in_ifwithnet搜索其他接口。如果任一函数找到与目的网络相连的接口，就返回ENETUNREACH；否则，ifp指向选定的接口。

这个选项允许路由选择协议绕过本地路由表，并使分组通过某特定接口退出系

统。通过这个方法，即使本地路由表不正确，也可以与其他路由器交换路由选择信息。

3. 本地路由

115-122 如果分组正被路由选择 (IP_ROUTETOIF为关状态)，并且没有其他缓存的路由，则rtalloc找到一条到dst指定地址的路由。如果rtalloc没找到路由，则ip_output返回EHOSTUNREACH。如果ip_forward调用ip_output，就把EHOSTUNREACH转换成ICMP差错。如果某个传输协议调用ip_output，就把差错传回给进程(图8-21)。

123-128 ia被设成指向选定接口的地址(ifaddr结构)，而ifp指向接口的ifnet结构。如果下一跳不是分组的最终目的地，则把dst改成下一跳路由器地址，而不再是分组最终目的地址。IP首部内的目的地址不变，但接口层必须把分组提交给dst，即下一跳路由器。

8.6.3 源地址选择和分片

ip_output的最后一部分如图8-25所示，保证IP首部有一个有效源地址，然后把分组提交给与路由相关的接口。如果分组比接口的MTU大，就必须对分组分片，然后一片一片地发送。像前面的重装代码一样，我们省略了分片代码，并推迟到第10章再讨论。

```

212      /*
213      * If source address not specified yet, use address
214      * of outgoing interface.
215      */
216      if (ip->ip_src.s_addr == INADDR_ANY)
217          ip->ip_src = IA_SIN(ia)->sin_addr;
218      /*
219      * Look for broadcast address and
220      * verify user is allowed to send
221      * such a packet.
222      */
223      if (in_broadcast(dst->sin_addr, ifp)) {
224          if ((ifp->if_flags & IFF_BROADCAST) == 0) { /* interface check */
225              error = EADDRNOTAVAIL;
226              goto bad;
227          }
228          if ((flags & IP_ALLOWBROADCAST) == 0) { /* application check */
229              error = EACCES;
230              goto bad;
231          }
232          /* don't allow broadcast messages to be fragmented */
233          if ((u_short) ip->ip_len > ifp->if_mtu) {
234              error = EMSGSIZE;
235              goto bad;
236          }
237          m->m_flags |= M_BCAST;
238      } else
239          m->m_flags &= ~M_BCAST;
240      sendit:
241      /*
242      * If small enough for interface, can just send directly.
243      */
244      if ((u_short) ip->ip_len <= ifp->if_mtu) {


```

图8-25 ip_output(续)

```

245     ip->ip_len = htons((u_short) ip->ip_len);
246     ip->ip_off = htons((u_short) ip->ip_off);
247     ip->ip_sum = 0;
248     ip->ip_sum = in_cksum(m, hlen);
249     error = (*ifp->if_output) (ifp, m,
250                               (struct sockaddr *) dst, ro->ro_rt);
251     goto done;
252 }

```



```

339 done:
340     if (ro == &iproute && (flags & IP_ROUTETOIF) == 0 && ro->ro_rt)
341         RTFREE(ro->ro_rt);
342     return (error);
343 bad:
344     m_freem(m0);
345     goto done;
346 }

```

ip_output.c

图8-25 (续)

1. 选择源地址

212-239 如果没有指定 `ip_src`，则 `ip_output` 选择输出接口的 IP 地址 `ia` 作为源地址。这不能在早期填充其他 IP 首部字段时做，因为那时还没有选定路由。转发的分组通常都有一个源地址，但是，如果发送进程没有明确指定源地址，产生于本地主机的分组可能没有源地址。

如果目的 IP 地址是一个广播地址，则接口必须支持广播 (`IFF_BROADCAST`，图3-7)，调用方必须明确使能广播 (`IP_ALLOWBROADCAST`，图8-23)，而分组必须足够小，无需分片。

最后的测试是一个策略决定。IP 协议规范中没有明确禁止对广播分组的分片。但是，要求分组适合接口的 MTU，就增加了广播分组被每个接口接收的机会，因为接收一个未损坏的分组的机会要远大于接收两个或多个未损坏分组的机会。

如果这些条件都不满足，就扔掉该分组，把 `EADDRNOTAVAIL`、`EACCES` 和 `EMSGSIZE` 返回给调用方。否则，设置输出分组的 `M_BCAST`，告诉接口输出函数把该分组作为链路级广播发送。21.20 节中，我们将看到 `arpresolve` 把 IP 广播地址翻译成以太网广播地址。

如果目的地址不是广播地址，则 `ip_output` 把 `M_BCAST` 清零。

如果 `M_BCAST` 没有清零，则对一个作为广播到达的请求分组的应答将可能作为一个广播被返回。我们将在第 11 章中看到，ICMP 应答将以这种方式作为 TCP RST 分组(见26.9节)在请求分组内构造。

2. 发送分组

240-252 如果分组对所选择的接口足够小，`ip_len` 和 `ip_off` 被转换成网络字节序，IP 校验和与 `in_cksum`(见8.7节)一起计算，把分组提交给所选接口的 `if_output` 函数。

3. 分片分组

253-338 大分组在被发送之前必须分片。这里我们省略这段代码，推迟到第 10 章讨论。

4. 清零

339-346 对每一路由入口都有一个引用计数。我们提到过，如果参数 `ro` 为空，`ip_`

output可能会使用一个临时的route结构(iproute)。如果需要，RTFREE发布iproute内的路由入口，并把引用计数减1。Bad处的代码在返回前扔掉当前分组。

引用计数是一个存储器管理技术。程序员必须对一个数据结构的外部引用计数；当计数返回为0时，就可以安全地把存储器返回给空存储器池。引用计数要求程序员遵守一些规定，在恰当的时机增加或减小引用计数。

8.7 Internet检验和：in_cksum函数

有两个操作占据了处理分组的主要时间：复制数据和计算检验和 ([Kay和Pasquale 1993])。mbuf数据结构的灵活性是Net/3中减少复制操作的主要方法。由于对硬件的依赖，所以检验和的有效计算相对较难。Net/3中有几种in_cksum的实现(图8-26)。

版 本	源 文 件
portable C	sys/netinet/in_cksum.c
SPARC	net3/sparc/sparc/in_cksum.c
68k	net3/luna68k/luna68k/in_cksum.c
VAX	sys/vax/vax/in_cksum.c
Tahoe	sys/tahoe/tahoe/in_cksum.c
HP 3000	sys/hp300/hp300/in_cksum.c
Intel 80386	sys/i386/i386/in_cksum.c

图8-26 在Net/3中的几个in_cksum 版本

即使是可移植C实现也已经被相当好地优化了。RFC 1071 [Braden、Borman和Partridge 1988] 和RFC 1141 [Mallory和Kullberg 1990]讨论了Internet检验和函数的设计和实现。RFC 1141被RFC 1624 [Rijsinghani 1994] 修正。从RFC 1071：

- 1) 把被检验的相邻字节成对配成16 bit整数，就形成了这些整数的二进制反码的和。
- 2) 为生成检验和，把检验和字段本身清零，把16 bit的二进制反码的和以及这个和的二进制反码放到检验和字段。
- 3) 为检验检验和，对同一组字节计算它们的二进制反码的和。如果结果为全1(在二进制反码运算中-0，见下面的解释)，则检验成功。

简而言之，当对用二进制反码表示的整数进行加法运算时，把两个整数相加后再加上进位就得到加法的结果。在二进制反码运算中，只要把每一位求补就得到一个数的反。所以在二进制反码运算中，0有两种表示方法：全0，和全1。有关二进制反码的运算和表示的详细讨论见 [Mano 1982]。

检验和算法在发送分组之前计算出要放在IP首部检验和字段的值。为了计算这个值，先把首部的检验和字段设为0，然后计算整个首部(包括选项)的二进制反码的和。把首部作为一个16 bit整数数组来处理。让我们把这个计算结果称为 a 。因为检验和字段被明确设为0，所以 a 是除了检验和字段外所有IP首部字段的和。 a 的二进制反码，用 $-a$ 表示，被放在检验和字段中，发送该分组。

如果在传输过程中没有比特位被改变，则在目的地计算的检验和应该等于 $(a + -a)$ 的二进制反码。在二进制反码运算中 $(a + -a)$ 的和是-0(全1)，而它的二进制反码应该等于0(全0)。所以在目的地，一个没有损坏分组计算出来的检验和应该总是为0。这就是我们在图8-12中看到

的。下面的C代码(不是Net/3的内容)是这个算法的一种原始的实现：

```

1 unsigned short
2 cksum(struct ip *ip, int len)
3 {
4     long    sum = 0;          /* assume 32 bit long, 16 bit short */
5
6     while (len > 1) {
7         sum += *((unsigned short *) ip)++;
8         if (sum & 0x80000000) /* if high-order bit set, fold */
9             sum = (sum & 0xFFFF) + (sum >> 16);
10        len -= 2;
11    }
12
13    if (len) /* take care of left over byte */
14        sum += (unsigned short) *(unsigned char *) ip;
15
16    while (sum >> 16)
17        sum = (sum & 0xFFFF) + (sum >> 16);
18
19    return ~sum;
20 }
```

图8-27 IP检验和计算的一种原始的实现

1-16 这里唯一提高性能之处在于累计 sum 高 16 bit 的进位。当循环结束时，累计的进位被加在低 16 bit 上，直到没有其他进位发生。RFC 1071 称此为延迟进位 (deferred carries)。在没有有进位加法指令或检测进位代价很大的机器上，这个技术非常有效。

现在我们显示 Net/3 的可移植 C 版本。它使用了延迟进位技术，作用于存储在一个 mbuf 链中的分组。

42-140 我们的新检验和实现假定所有被检验字节存储在一个连续缓存而不是 mbuf 中。这个版本的检验和计算采用相同的底层算法来正确地处理 mbuf：用 32bit 整数的延迟进位对 16 bit 字作加法。对奇数个字节的 mbuf，多出来的一个字节被保存起来，并与下一个 mbuf 的第一个字节配对。因为在大多数体系结构中，对 16 bit 字的不对齐访问是无效的，甚至会产生严重差错，所以不对齐字节将被保存，in_cksum 继续加上下一个对齐的字。当这种情况发生时，in_cksum 总是很小心地交换字节，保证位于奇数和偶数位置的字节被放在单独的和字节中，以满足检验和算法的要求。

循环展开

93-115 函数中的三个 while 循环在每次迭代中分别在和中加上 16 个字、4 个字和 1 个字。展开的循环减小了循环的耗费，在某些体系结构中可能比一个直接循环要快得多。但代价是代码长度和复杂性增大。

```

42 #define ADDCARRY(x)  (x > 65535 ? x -= 65535 : x)
43 #define REDUCE(l_util.l = sum; sum = l_util.s[0] + l_util.s[1]; ADDCARRY(sum);}
44 int
45 in_cksum(m, len)
46 struct mbuf *m;
47 int    len;
48 {
49     u_short *w;
50     int    sum = 0;
```

in_cksum.c

图8-28 IP检验和计算的一个优化的可移植 C 程序

```

51     int      mlen = 0;
52     int      byte_swapped = 0;
53     union {
54         char   c[2];
55         u_short s;
56     } s_util;
57     union {
58         u_short s[2];
59         long    l;
60     } l_util;
61     for (; m && len; m = m->m_next) {
62         if (m->m_len == 0)
63             continue;
64         w = mtod(m, u_short *);
65         if (mlen == -1) {
66             /*
67              * The first byte of this mbuf is the continuation of a
68              * word spanning between this mbuf and the last mbuf.
69              *
70              * s_util.c[0] is already saved when scanning previous mbuf.
71              */
72             s_util.c[1] = *(char *) w;
73             sum += s_util.s;
74             w = (u_short *) ((char *) w + 1);
75             mlen = m->m_len - 1;
76             len--;
77         } else
78             mlen = m->m_len;
79         if (len < mlen)
80             mlen = len;
81         len -= mlen;
82         /*
83          * Force to even boundary.
84          */
85         if ((1 & (int) w) && (mlen > 0)) {
86             REDUCE;
87             sum <= 8;
88             s_util.c[0] = *(u_char *) w;
89             w = (u_short *) ((char *) w + 1);
90             mlen--;
91             byte_swapped = 1;
92         }
93         /*
94          * Unroll the loop to make overhead from
95          * branches &c small.
96          */
97         while ((mlen -= 32) >= 0) {
98             sum += w[0]; sum += w[1]; sum += w[2]; sum += w[3];
99             sum += w[4]; sum += w[5]; sum += w[6]; sum += w[7];
100            sum += w[8]; sum += w[9]; sum += w[10]; sum += w[11];
101            sum += w[12]; sum += w[13]; sum += w[14]; sum += w[15];
102
103            w += 16;
104        }
105        mlen += 32;
106        while ((mlen -= 8) >= 0) {
107            sum += w[0]; sum += w[1]; sum += w[2]; sum += w[3];
108
109            w += 4;

```

图8-28 (续)

```
108     }
109     mlen += 8;
110     if (mlen == 0 && byte_swapped == 0)
111         continue;
112     REDUCE;
113     while ((mlen -= 2) >= 0) {
114         sum += *w++;
115     }
116     if (byte_swapped) {
117         REDUCE;
118         sum <= 8;
119         byte_swapped = 0;
120         if (mlen == -1) {
121             s_util.c[1] = *(char *) w;
122             sum += s_util.s;
123             mlen = 0;
124         } else
125             mlen = -1;
126     } else if (mlen == -1)
127         s_util.c[0] = *(char *) w;
128 }
129 if (len)
130     printf("cksum: out of data\n");
131 if (mlen == -1) {
132     /* The last mbuf has odd # of bytes. Follow the standard (the odd
133        byte may be shifted left by 8 bits or not as determined by
134        endian-ness of the machine) */
135     s_util.c[1] = 0;
136     sum += s_util.s;
137 }
138 REDUCE;
139 return (~sum & 0xffff);
140 }
```

in_cksum.c

图8-28 (续)

其他优化

RFC 1071提到两个在Net/3中没有出现的优化：联合的有检验和的复制操作和递增的检验和更新。对IP首部检验和来说，把复制和检验和操作结合起来并不像对TCP和UDP那么重要，因为后者覆盖了更多的字节。在23.12节中对这个合并的操作进行了讨论。[Partridge和Pink 1993]报告了IP首部检验和的一个内联版本比调用更一般的in_cksum函数要快得多，只需6~8个汇编指令就可以完成(标准的20字节IP首部)。

检验和算法设计允许改变分组，并在不重新检查所有字节的情况下更新检验和。RFC 1071对该问题进行简明的讨论。RFC 1141和1624中有更详细的讨论。该技术的一个典型应用是在分组转发的过程中。通常情况下，当分组没有选项时，转发过程中只有TTL字段发生变化。在这种情况下，可以只用一次循环进位，重新计算检验和。

为了进一步提高效率，递增的检验和也有助于检测到被有差错的软件破坏的首部。如果递增地计算检验和，则下一个系统可以检测到被破坏的首部。但是如果不是递增计算检验和，那么检验和中就包含了差错的字节，检测不到有问题的首部。UDP和TCP使用的检验和算法在最终目的主机检测到该差错。我们将在第23和25章看到UDP和TCP检验和包含了IP首部的几个部分。

使用硬件有进位加法指令一次性计算 32 bit 检验和的检验和函数，可参见 `./sys/vax/vax/in_cksum.c` 文件中 VAX 实现的 `in_cksum`。

8.8 setsockopt和getsockopt系统调用

Net/3 提供 `setsockopt` 和 `getsockopt` 两个系统调用来访问一些网络互连的性质。这两个系统调用支持一个动态接口，进程可用该动态接口来访问某种网络互连协议的一些性质，而标准系统调用通常不支持该协议。这两个调用的原型是：

```
int setsockopt(int s, int level, int optname, void *optval, int optlen);
int getsockopt(int s, int level, int optname, const void *optval, int optlen);
```

大多数插口选项只影响它们在其上发布的插口。与 `sysctl` 参数相比，后者影响整个系统。与多播相关的插口选项是一个明显的例外，将在第 12 章中讨论。

`setsockopt` 和 `getsockopt` 设置和获取通信栈所有层上的选项。Net/3 按照与 `s` 相关的协议和由 `level` 指定的标识符处理选项。图 8-29 列出了在我们讨论的协议中 `level` 可能取得的值。

在第 17 章中，我们描述了 `setsockopt` 和 `getsockopt` 的实现，但在其他适当章节中讨论有关选项的实现。本章讨论访问 IP 性质的选项。

字段	协议	level	函 数	参 考
任意	任意	<i>SOL_SOCKET</i>	<i>so</i> setopt和 <i>so</i> getopt	图17-5和图17-11
IP	UDP	<i>IPPROTO_IP</i>	<i>ip</i> _ctloutput	图8-31
	TCP	<i>IPPROTO_TCP</i> <i>IPPROTO_IP</i>	<i>tcp</i> _ctloutput <i>ip</i> _ctloutput	30.6节 图8-31
	原始IP ICMP IGMP	<i>IPPROTO_IP</i>	<i>rip</i> _ctloutput和 <i>ip</i> _ctloutput	32.8节

图8-29 *so*setopt 和 *so*getopt 参数

我们把本书中出现的所有插口选项总结在图 8-30 中。该图显示了 `IPPROTO_IP` 级的选项。选项出现在第 1 列，`optval` 指向变量的数据类型出现在第 2 列，第 3 列显示的是处理该选项的函数。

选 项 名	Optval类型	函 数	描 述
<i>IP_OPTIONS</i>	void*	<i>in</i> _pcbopts	设置或获取发出的数据报中的 IP 选项
<i>IP_TOS</i>	int	<i>ip</i> _ctloutput	设置或获取发出的数据报中的 IP TOS
<i>IP_TTL</i>	int	<i>ip</i> _ctloutput	设置或获取发出的数据报中的 IP TTL
<i>TP_RECV DSTADDR</i>	int	<i>ip</i> _ctloutput	使能或禁止 IP 目的地址 (只有 UDP) 的排队
<i>IP_RECVOPTS</i>	int	<i>ip</i> _ctloutput	使能或禁止对到达 IP 选项作为控制信息的排队 (只对 UDP；还没有实现)
<i>IP_RECVRETOPTS</i>	int	<i>ip</i> _ctloutput	使能或禁止与到达数据报相关的逆源路由 (只对 UDP；还没有实现)

图8-30 插口选项：`SOCK_RAW`、`SOCK_DGRAM` 和 `SOCK_STREAM` 插口的 `IPPROTO_IP` 级

图 8-31 显示了用于处理大部分 `IPPROTO_IP` 选项的 `ip_ctloutput` 函数的整个结构。在 32.8 节中我们给出与 `SOCK_RAW` 插口一起使用的 `IPPROTO_IP` 选项。

ip_output.c

```

431 int
432 ip_ctloutput(op, so, level, optname, mp)
433 int    op;
434 struct socket *so;
435 int    level, optname;
436 struct mbuf **mp;
437 {
438     struct inpcb *inp = sotoinpcb(so);
439     struct mbuf *m = *mp;
440     int    optval;
441     int    error = 0;

442     if (level != IPPROTO_IP) {
443         error = EINVAL;
444         if (op == PRCO_SETOPT && *mp)
445             (void) m_free(*mp);
446     } else
447         switch (op) {
448             case PRCO_SETOPT:
449                 switch (optname) {

```

```

/* PRCO_SETOPT processing (Figures 8.32 and 12.17) */

```

```

493         freeit:
494         default:
495             error = EINVAL;
496             break;
497         }
498         if (m)
499             (void) m_free(m);
500         break;

501     case PRCO_GETOPT:
502         switch (optname) {

```

```

/* PRCO_GETOPT processing (Figures 8.33 and 12.17) */

```

```

546         default:
547             error = ENOPROTOOPT;
548             break;
549         }
550         break;
551     }
552     return (error);
553 }

```

*ip_output.c*图8-31 `ip_ctloutput` 函数：概貌

431-447 `ip_ctloutput` 的第一个参数 `op`，可以是 `PRCO_SETOPT` 或者 `PRCO_GETOPT`。第二个参数 `so`，指向向其发布请求的插口。`level` 必须是 `IPPROTO_IP`。`optname` 是要改变或要检索的选项，`mp` 间接地指向一个含有与该选项相关数据的 `mbuf`，`m` 被初始化为指向由 `*mp` 引用的 `mbuf`。

448-500 如果在调用 `setsockopt` 时指定了一个无法识别的选项(因此，在 `switch` 中调用 `PRCO_SETOPT` 语句)，`ip_ctloutput` 释放掉所有调用方传来的缓存，并返回 `EINVAL`。

501-553 getsockopt传来的无法识别的选项导致ip_ctloutput返回ENOPROTOOPT。在这种情况下，调用方释放mbuf。

8.8.1 PRCO_SETOPT的处理

对PRCO_SETOPT的处理如图8-32所示。

```

450             case IP_OPTIONS:
451                 return (ip_pcbopts(&inp->inp_options, m));
452             case IP_TOS:
453             case IP_TTL:
454             case IP_RECVOPTS:
455             case IP_RECVRETOPTS:
456             case IP_RECVDSTADDR:
457                 if (m->m_len != sizeof(int))
458                     error = EINVAL;
459                 else {
460                     optval = *mtod(m, int *);
461                     switch (optname) {
462                         case IP_TOS:
463                             inp->inp_ip.ip_tos = optval;
464                             break;
465                         case IP_TTL:
466                             inp->inp_ip.ip_ttl = optval;
467                             break;
468 #define OPTSET(bit) \
469     if (optval) \
470         inp->inp_flags |= bit; \
471     else \
472         inp->inp_flags &= ~bit;
473                         case IP_RECVOPTS:
474                             OPTSET(INP_RECVOPTS);
475                             break;
476                         case IP_RECVRETOPTS:
477                             OPTSET(INP_RECVRETOPTS);
478                             break;
479                         case IP_RECVDSTADDR:
480                             OPTSET(INP_RECVDSTADDR);
481                             break;
482                     }
483                 }
484             break;

```

ip_output.c

ip_output.c

图8-32 ip_ctloutput 函数：处理PRCO_SETOPT

450-451 IP_OPTIONS是由ip_pcbopts处理的(图9-32)。

452-484 IP_TOS、IP_TTL、IP_RECVOPTS、IP_RECVRETOPTS以及IP_RECVDSTADDR选项都需要在由m指向的mbuf中有一个整数。该整数储存在optval中，用来改变与插口有关的ip_tos和ip_ttl的值，或者用来设置或复位与插口相关的INP_RECVOPTS、INP_RECVRETOPTS和INP_RECVDSTADDR标志位。如果optval是非零(或0)，则宏OPTSET设置(或复位)指定的比特。

图8-30中显示没有实现IP_RECVOPTS和IP_RECVRETOPTS。在第23章中，我

们将看到UDP忽略了这些选项的设置。

8.8.2 PRCO_GETOPT的处理

图8-33显示的一段代码完成了当指定PRCO_GETOPT时对IP选项的检索。

```

503         case IP_OPTIONS:
504             *mp = m = m_get(M_WAIT, MT_SOOPTS);
505             if (inp->inp_options) {
506                 m->m_len = inp->inp_options->m_len;
507                 bcopy(mtod(inp->inp_options, caddr_t),
508                     mtod(m, caddr_t), (unsigned) m->m_len);
509             } else
510                 m->m_len = 0;
511             break;
512
513         case IP_TOS:
514         case IP_TTL:
515         case IP_RECVOPTS:
516         case IP_RECVRETOPTS:
517         case IP_RECVDSTADDR:
518             *mp = m = m_get(M_WAIT, MT_SOOPTS);
519             m->m_len = sizeof(int);
520             switch (optname) {
521                 case IP_TOS:
522                     optval = inp->inp_ip.ip_tos;
523                     break;
524                 case IP_TTL:
525                     optval = inp->inp_ip.ip_ttl;
526                     break;
527                 #define OPTBIT(bit) (inp->inp_flags & bit ? 1 : 0)
528                 case IP_RECVOPTS:
529                     optval = OPTBIT(INP_RECVOPTS);
530                     break;
531                 case IP_RECVRETOPTS:
532                     optval = OPTBIT(INP_RECVRETOPTS);
533                     break;
534                 case IP_RECVDSTADDR:
535                     optval = OPTBIT(INP_RECVDSTADDR);
536                     break;
537             }
538             *mtod(m, int *) = optval;
539             break;

```

ip_output.c

图8-33 ip_ctloutput 函数：PRCO_GETOPT 的处理

503-538 对IP_OPTIONS, ip_ctloutput返回一个缓存, 该缓存中包含了与该插口相关的选项的备份。对其他选项, ip_ctloutput返回ip_tos和ip_ttl的值, 或与该选项相关的标志的状态。返回的值放在由m指向的mbuf中。如果在inp_flags中的bit是打开(或关闭)的, 则宏OPTBIT将返回1(或0)。

8.9 ip_sysctl函数

图7-27显示, 在调用sysctl中, 当协议和协议族的标识符是0时, 就调用ip_sysctl函

数。图8-34显示了ip_sysctl支持的三个函数。

sysctl常量	Net/3变量	描 述
IPCTL_FORWARDING	ipforwarding	系统是否转发IP分组？
IPCTL_SENDREREDIRECTS	ipsendredirects	系统是否发ICMP重定向？
IPCTL_DEFTTL	ip_defttl	IP分组的默认TTL

图8-34 sysctl 参数

图8-35显示了ip_sysctl函数。

```

984 int
985 ip_sysctl(name, namelen, oldp, oldlenp, newp, newlen)
986 int *name;
987 u_int namelen;
988 void *oldp;
989 size_t *oldlenp;
990 void *newp;
991 size_t newlen;
992 {
993     /* All sysctl names at this level are terminal. */
994     if (namelen != 1)
995         return (ENOTDIR);

996     switch (name[0]) {
997     case IPCTL_FORWARDING:
998         return (sysctl_int(oldp, oldlenp, newp, newlen, &ipforwarding));
999     case IPCTL_SENDREREDIRECTS:
1000         return (sysctl_int(oldp, oldlenp, newp, newlen,
1001                             &ipsendredirects));
1002     case IPCTL_DEFTTL:
1003         return (sysctl_int(oldp, oldlenp, newp, newlen, &ip_defttl));
1004     default:
1005         return (EOPNOTSUPP);
1006     }
1007     /* NOTREACHED */
1008 }

```

ip_input.c

ip_input.c

图8-35 ip_sysctl 函数

因为ip_sysctl并不把sysctl请求转发给其他函数，所以在name中只能有一个成员。否则返回ENOTDIR。

Switch语句选择恰当的调用 sysctl_int，它访问或修改 ipforwarding、ipsendredirects或ip_defttl。对无法识别的选项返回EOPNOTSUPP。

8.10 小结

IP是一个最佳的数据报服务，它为所有其他 Internet协议提供交付机制。标准IP首部长度为20字节，但可跟最多40字节的选项。IP可以把大的数据报分片发送，并在目的地重装分片。对选项处理的讨论放在第9章和第10章讨论分片和重装。

ipintr保证IP首部到达时未经破坏，通过把目的地址与系统接口地址及其他几个广播地址比较来确定它们是否到达最终目的地。ipintr把到达最终目的地的数据报传给分组内指定的运输层协议。如果系统被配置成路由器，就把还没有到达最终目的地的分组发给

ip_forward转发到最终目的地。分组有一个受限的生命期。如果 TTL字段变成 0，则 ip_forward就丢掉该分组。

许多Internet协议都使用Internet检验和函数，Net/3用in_cksum实现。IP检验和只覆盖首部(和选项)，不覆盖数据，数据必须由传输协议级的检验和保护。作为 IP中最耗时的操作，检验和函数通常要对不同的平台进行优化。

习题

- 8.1 当没有为任何接口分配IP地址时，IP是否该接收广播分组？
- 8.2 修改ip_forward和ip_output，当转发一个没有选项的分组时，对IP检验和进行递增的更新。
- 8.3 当拒绝转发分组时，为什么需要检测链路级广播（某缓存中的M_BCAST标志）和IP级广播(in_canforward)？在何种情况下，把一个具有IP单播目的地的分组作为一个链路层广播接收？
- 8.4 当一个IP分组到达时有检验和差错，为什么不向发送方返回一个差错信息？
- 8.5 假定一个多接口主机上的某个进程为它发出的分组选择了一个明确的源地址。而且，假定是通过一个接口而不是作为分组源地址所选择的地址到达的。当第一跳路由器发现分组应该到另一个路由器时，会发生什么情况？会向主机发送重定向报文吗？
- 8.6 一个新的主机被连到一个已划分子网的网络中，并被配置成完成路由选择的功能(ipforwarding等于1)，但它的网络接口没有分配子网掩码。当该主机接收一个子网广播分组时会出现什么情况？
- 8.7 图8-17中，在检测ip_ttl后(与之前相比)，为什么需要把它减1？
- 8.8 如果两个路由器都认为对方是分组的最佳下一跳目的地，将发生什么情况？
- 8.9 图8-14中，对一个到达SLIP接口的分组，不检测哪些地址？有没有其他在图 8-14中没有列出的地址被检测？
- 8.10 ip_forward在调用icmp_error之前，把分片的id从主机字节序转换成网络字节序。为什么它不对分片的偏移进行转换？