

第12章 IP 多播

12.1 引言

第8章讲到，D类IP地址(224.0.0.0到239.255.255.255)不识别互联网内的单个接口，但识别接口组。因为这个原因，D类地址被称为多播组(multicast group)。具有D类目的地址的数据报被提交给互联网内所有加入相应多播组的各个接口。

Internet上利用多播的实验性应用程序包括：音频和视频会议应用程序、资源发现工具和共享白板等。

多播组的成员由于接口加入或离开组而动态地变化，这是根据各系统上运行的进程的请求决定的。因为多播组成员与接口有关，所以多接口主机可能针对每个接口，都有不同的多播组成员关系表。我们称一个特定接口上的组成员关系为一对{接口，多播组}。

单个网络上的组成员利用IGMP协议(第13章)在系统之间通信。多播路由器用多播选路协议(第14章)，如DVMRP(Distance Vector Multicast Routing Protocol，距离向量多播路由选择协议)传播成员信息。标准IP路由器可能支持多播选路，或者用一专用路由器处理多播选路。

如以太网、令牌环和FDDI一类的网络直接支持硬件多播。在Net/3中，如果某个接口支持多播，那么在接口的ifnet结构(图3-7)中的if_flags标志的IFF_MULTICAST比特就被打开。因为以太网被广泛使用，并且Net/3有以太网驱动程序，所以我们将以以太网为例说明硬件支持的IP多播。多播业务通常在如SLIP和环回接口等的点到点网络上实现。

如果本地网络不支持硬件级多播，那么在某个特定接口上就得不到IP多播业务。RFC 1122并不反对接口层提供软件级的多播业务，只要它对IP是透明的。

RFC 1112 [Deering 1989] 描述了多播对主机的要求。分三个级别：

0级：主机不能发送或接收IP多播。

这种主机应该自动丢弃它收到的具有D类目的地址的分组。

1级：主机能发送但不能接收IP多播。

在向某个IP多播组发送数据报之前，并不要求主机加入该组。多播数据报的发送方式与单播一样，除了多播数据报的目的地址是IP多播组之外。网络驱动器必须能够识别出这个地址，把在本地网络上多播数据报。

2级：主机能发送和接收IP多播。

为了接收IP多播，主机必须能够加入或离开多播组，而且必须支持IGMP，能够在至少一个接口上交换组成员信息。多接口主机必须支持在它的接口的一个子网上的多播。

Net/3符合2级主机要求，可以完成多播路由器的的工作。与单播IP选路一样，我们假定所描述的系统是一个多播路由器，并加上了Net/3多播选路的程序。

知名的IP多播组

和UDP、TCP的端口号一样，互联网号授权机构IANA(Internet Assigned Numbers

Authority)维护着一个注册的IP多播组表。当前的表可以在RFC 1700中查到。有关IANA的其他信息可以在RFC 1700中找到。图12-1只给出了一些知名的多播组。

组	描 述	Net/3常量
224.0.0.0	预留	<i>INADDR_UNSPEC_GROUP</i>
224.0.0.1	这个子网上的所有系统	<i>INADDR_ALLHOSTS_GROUP</i>
224.0.0.2	这个子网上的所有路由器	
224.0.0.3	没有分配	<i>INADDR_MAX_LOCAL_GROUP</i>
224.0.0.4	DVMRP路由器	
224.0.0.255	没有分配	
224.0.1.1	NTP网络时间协议	
224.0.1.2	SGI-Dogfight	

图12-1 一些注册的IP多播组

前256个组(224.0.0.0到224.0.0.255)是为实现IP单播和多播选路机制的协议预留的。不管发给其中任意一个组的数据报内IP首部的TTL值如何变化,多播路由器都不会把它转发出本地网络。

RFC 1075只对224.0.0.0组和224.0.0.1组有这个要求,但最常见的多播选路实现mrouted限制这里讨论的其他组。组224.0.0.0(*INADDR_UNSPEC_GROUP*)被预留,组224.0.0.255(*INADDR_MAX_LOCAL_GROUP*)标志着本地最后一个多播组。

对于符合2级的系统,要求其在系统初始化时(图6-17),在所有的多播接口上加入224.0.0.1组(*INADDR_ALLHOSTS_GROUP*),并且保持为该组成员,直到系统关闭。在一个互联网上,没有多播组与每个接口都对应。

想像一下,如果你的语音邮件系统有一个选项,可以向公司里的所有语音邮箱发一个消息。可能你就有这个选项。你发现它有用吗?对更大的公司适用吗?是否有人能向“所有邮箱”组发邮件,或者是否限制这么做?

单播和多播路由可能会加入224.0.0.2组进行互相通信。ICMP路由器请求报文和路由器通告报文可能被分别发往224.0.0.2(“所有路由器”组)和224.0.0.1(“所有主机”组),而不是受限的广播地址(255.255.255.255)。

224.0.0.4组支持在实现DVMRP的多播路由器之间的通信。本地多播组范围内的其他组被类似地指派给其他路由选择协议。

除了前256个组外,其他组(224.0.1.0~239.255.255.255)或者被分配给多个多播应用程序协议,或者仍然没有被分配。图12-1中有两个例子,网络时间协议(224.0.1.1)和SGI-Dogfight(224.0.1.2)。

在本章中,我们注意到,是主机上的运输层发送和接收多播分组。尽管多播程序并不知道具体是哪个传输协议发送和接收多播数据报,但唯一支持多播的Internet传输协议是UDP。

12.2 代码介绍

本章中讨论的基本多播程序与标准IP程序在相同的文件里。图12-2列出了我们研究的文件。

文 件	描 述
net/if_ether.h	以太网多播数据结构和宏定义
netinet/in.h	其他Internet多播数据结构
netinet/in_var.h	Internet多播数据结构和宏定义
netinet/ip_var.h	IP多播数据结构
net/if_etherubr.c	以太网多播函数
netinet/in.c	组成员函数
netinet/ip_input.c	输入多播处理
netinet/ip_output.c	输出多播处理

图12-2 本章讨论的文件

12.2.1 全局变量

本章介绍了三个新的全局变量(图12-3)。

变 量	数 据 类 型	描 述
ether_ipmulticast_min	u_char []	为IP预留的最小以太网多播地址
ether_ipmulticast_max	u_char []	为IP预留的最大以太网多播地址
ip_mrouter	struct socket	多播选路守护程序创建的指向插口的指针

图12-3 本章引入的全局变量

12.2.2 统计量

本章讨论的程序更新全局 ipstat 结构中的几个计数器。

ipstat成员	描 述
ips_forward	被这个系统转发的分组数
ips_cantforward	不能被系统转发的分组数——系统不是一个路由器
ips_noroute	由于无法访问到路由器而无法转发的分组数

图12-4 多播处理统计量

链路级多播统计放在 ifnet 结构中(图4-5)，还可能统计除IP以外的其他协议的多播。

12.3 以太网多播地址

IP多播的高效实现要求IP充分利用硬件级多播，因为如果没有硬件级多播，就不得不在网络上广播每个多播IP数据报，而每台主机也不得不检查每个数据报，把那些不是给它的丢掉。硬件在数据报到达IP层之前，就把没有用的过滤掉了。

为了保证硬件过滤器能正常工作，网络接口必须把IP多播组目的地址转换成网络硬件识别的链路级多播地址。在点到点网络上，如SLIP和环回接口，必须明确给出地址映射，因为只能有一个目的地址。在其他网络上，如以太网，也需要有一个明确地完成映射地址的函数。以太网的标准映射适用于任何使用802.3寻址方式的网络。

图4-12显示了以太网单播和多播地址的区别：如果以太网地址的高位字节的最低位是1，则它是一个多播地址；否则，它是一个单播地址。单播以太网地址由接口制造商分配，多播

地址由网络协议动态分配。

IP到以太网地址映射

因为以太网支持多种协议，所以要采取措施分配多播地址，避免冲突。IEEE管理以太网多播地址分配。IEEE把一块以太网多播地址分给IANA以支持IP多播。块的地址都以01:00:5e开头。

以00:00:5e开头的以太网单播也被分配给 IANA，但为将来使用预留。

图12-5显示了从一个D类IP地址构造出一个以太网多播地址。

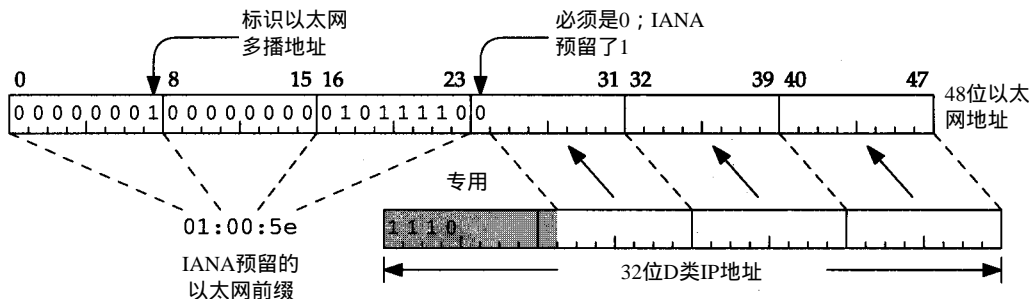


图12-5 IP和以太网地址之间的映射

图12-5显示的映射是一个多到一的映射。在构造以太网地址时，没有使用 D类IP地址的高位9比特。32个 IP多播组映射到一个以太网多播地址（习题12.3）。我们将在 12.14节看到这将如何影响输入的处理。图 12-6显示了Net/3中实现这个映射的宏。

```

61 #define ETHER_MAP_IP_MULTICAST(ipaddr, enaddr) \
62     /* struct in_addr *ipaddr; */ \
63     /* u_char enaddr[6]; */ \
64 { \
65     (enaddr)[0] = 0x01; \
66     (enaddr)[1] = 0x00; \
67     (enaddr)[2] = 0x5e; \
68     (enaddr)[3] = ((u_char *)ipaddr)[1] & 0x7f; \
69     (enaddr)[4] = ((u_char *)ipaddr)[2]; \
70     (enaddr)[5] = ((u_char *)ipaddr)[3]; \
71 }

```

if_ether.h

图12-6 ETHER_MAP_IP_MULTICAST 宏

IP到以太网多播映射

61-71 ETHER_MAP_IP_MULTICAST实现图12-5所示的映射。ipaddr指向D类多播地址，enaddr构造匹配的以太网地址，用 6字节的数组表示。该以太网多播地址的前 3个字节是 0x01，0x00和0x5e，后面跟着0比特，然后是D类IP地址的低23位。

12.4 ether_multi结构

Net/3为每个以太网接口维护一个该硬件接收的以太网多播地址范围表。这个表定义了该设备要实现的多播过滤。因为大多数以太网设备能选择地接收的地址是有限的，所以 IP层必须要准备丢弃那些通过了硬件过滤的数据报。地址范围被保存在 ether_multi结构中(图12-7)：

```

147 struct ether_multi {
148     u_char  enm_addrlo[6];      /* low or only address of range */
149     u_char  enm_addrhi[6];      /* high or only address of range */
150     struct arpcom *enm_ac;      /* back pointer to arpcom */
151     u_int    enm_refcount;      /* no. claims to this addr/range */
152     struct ether_multi *enm_next; /* ptr to next ether_multi */
153 };

```

if_ether.h

图12-7 ether_multi 结构

1. 以太网多播地址

147-153 enm_addrlo和enm_addrhi指定需要被接收的以太网多播地址的范围。当enm_addrlo和enm_addrhi相同时，就指定一个以太网地址。ether_multi的完整列表附在每个以太网接口的arpcom结构中(图3-26)。以太网多播独立于ARP——使用arpcom结构只是为了方便，因为该结构已经存在于所有以太网接口结构中。

我们将看到，这个范围的开头和结尾总是相同的，因为在Net/3中，进程无法指定地址范围。

enm_ac指回相关接口的arpcom结构，enm_refcount跟踪对ether_multi结构的使用。当引用计数变成0时，就释放arpcom结构。enm_next把单个接口的ether_multi结构做成链表。图12-8显示出，有三个ether_multi结构的链表附在le_softc[0]上，这是我们以太网接口示例的ifnet结构。

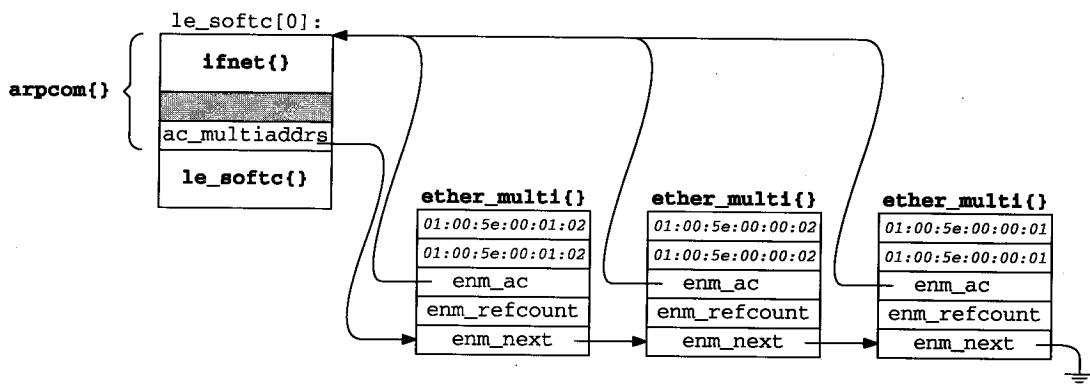


图12-8 有三个ether_multi 结构的LANCSE接口

在图12-8中，我们看到：

- 接口已经加入了三个组。很有可能是 224.0.0.1(所有主机)、224.0.0.2(所有路由器)和 224.0.1.2(SGI-dogfight)。因为以太网到IP地址的映射是一到多的，所以只看到以太网多播地址的结果，无法确定确切的IP多播地址。比如，接口可能已经加入了 225.0.0.1、225.0.0.2和226.0.1.2组。
- 有了enm_ac后向指针，就很容易找到链表的开始，释放某个 ether_multi结构，无需再实现双向链表。
- ether_multi只适用于以太网设备。其他多播设备可能有其他实现。

图12-9中的ETHER_LOOKUP_MULTI宏，搜索某个ether_multi结构，找到地址范围。

2. 以太网多播查找

166-177 addrlo和addrhi指定搜索的范围，ac指向包含了要搜索链表的arpcom结构。

for循环完成线性搜索，在表的最后结束，或者当 `enm_addrlo`和`enm_addrhi`都分别与和所提供的 `addrlo`和`addrhi`匹配时结束。当循环终止时，`enm`为空或者指向某个匹配的 `ether_multi`结构。

```

166 #define ETHER_LOOKUP_MULTI(addrlo, addrhi, ac, enm) \
167     /* u_char addrlo[6]; */ \
168     /* u_char addrhi[6]; */ \
169     /* struct arpcom *ac; */ \
170     /* struct ether_multi *enm; */ \
171 { \
172     for ((enm) = (ac)->ac_multiaddrs; \
173          (enm) != NULL && \
174          (bcmp((enm)->enm_addrlo, (addrlo), 6) != 0 || \
175           bcmp((enm)->enm_addrhi, (addrhi), 6) != 0); \
176          (enm) = (enm)->enm_next); \
177 }

```

if_ether.h

if_ether.h

图12-9 ETHER_LOOKUP_MULTI 宏

12.5 以太网多播接收

从本节以后，本章只讨论 IP多播。但是，在 Net/3中，也有可能把系统配置成接收所有以太网多播分组。虽然对 IP 协议族没有用，但内核的其他协议族可能准备接收这些多播分组。发出图12-10中的 `ioctl`命令，就可以明确地进行多播配置。

命 令	参 数	函 数	描 述
<code>SIOCADMULTI</code>	<code>struct ifreq *</code>	<code>ifioctl</code>	在接收表里加上多播地址
<code>SIOCDELMULTI</code>	<code>struct ifreq *</code>	<code>ifioctl</code>	从接收表里删去多播地址

图12-10 多播 `ioctl` 命令

这两个命令被 `ifioctl`(图12-11)直接传给 `ifreq`结构(图6-12)中所指定的接口的设备驱动程序。

```

440 case SIOCADMULTI:
441 case SIOCDELMULTI:
442     if (error = suser(p->p_ucred, &p->p_acflag))
443         return (error);
444     if (ifp->if_ioctl == NULL)
445         return (EOPNOTSUPP);
446     return ((*ifp->if_ioctl) (ifp, cmd, data));

```

if.c

if.c

图12-11 `ifioctl` 函数：多播命令

440-446 如果该进程没有超级用户权限，或者如果接口没有 `if_ioctl`结构，则 `ifioctl`返回一个错误；否则，把请求直接传给该设备驱动程序。

12.6 `in_multi`结构

12.4节描述的以太网多播数据结构并不专用于 IP；它们必须支持所有内核支持的任意协议族的多播活动。在网络级，IP维护着一个与接口相关的IP多播组表。

为了实现方便，把这个IP多播表附在与该接口有关的 `in_ifaddr`结构中。6.5节讲到，这

个结构中包含了该接口的单播地址。除了它们都与同一个接口相关以外，这个单播地址与所附的多播组表之间没有任何关系。

这是Net/3实现的产品。也可以在一个不接收IP单播分组的接口上，支持IP多播组。

图12-12中的in_multi结构描述了每个IP多播{接口，组}对。

```

111 struct in_multi {
112     struct in_addr inm_addr;    /* IP multicast address */
113     struct ifnet *inm_ifp;     /* back pointer to ifnet */
114     struct in_ifaddr *inm_ia;  /* back pointer to in_ifaddr */
115     u_int    inm_refcount;      /* no. membership claims by sockets */
116     u_int    inm_timer;        /* IGMP membership report timer */
117     struct in_multi *inm_next; /* ptr to next multicast address */
118 };

```

in_var.h

图12-12 in_multi 结构

1. IP多播地址

111-118 inm_addr是一个D类多播地址(如224.0.0.1，所有主机组)。inm_ifp指回相关接口的ifnet结构，而inm_ia指回接口的in_ifaddr结构。

只有当系统中的某个进程通知内核，它要在某个特定的 {接口，组}对上接收多播数据报时，才存在一个in_multi结构。由于可能会有多个进程要求接收发往同一个对上的数据报，所以inm_refcount跟踪对该对的引用次数。当没有进程对某个特定的对感兴趣时，inm_refcount就变成0，in_multi结构就被释放掉。这个动作可能会引起相关的ether_multi结构也被释放，如果此时它的引用计数也变成了0。

inm_timer是第13章描述的IGMP协议实现的一部分，最后，inm_next指向表中的下一个in_multi结构。

图12-13用接口示例le_softc[0]显示了接口，即它的单播地址和它的IP多播组表之间的关系。

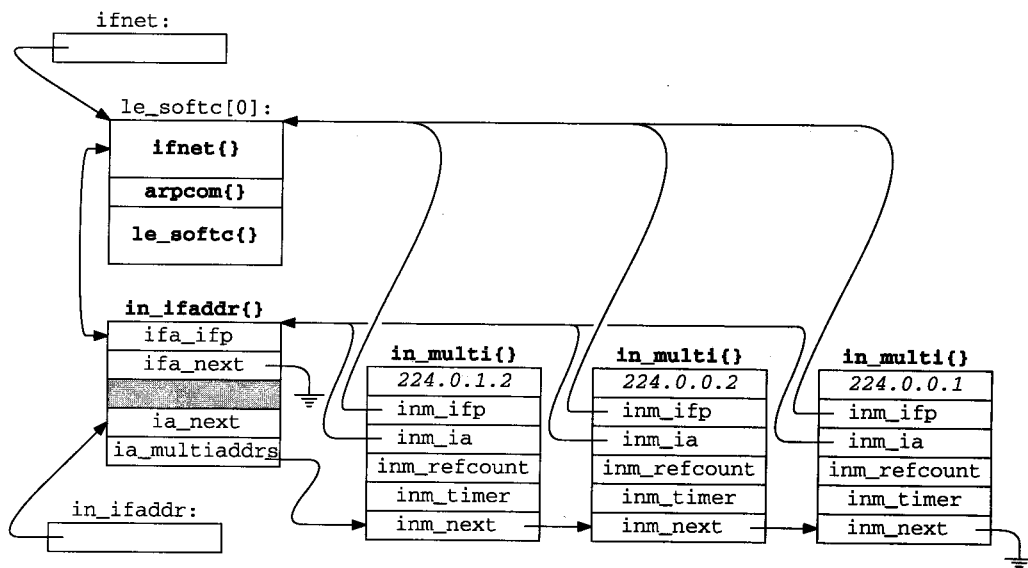


图12-13 le接口的一个IP多播组表

为了清楚起见，我们已经省略了对应的 `ether_multi` 结构(图12-34)。如果系统有两个以太网网卡，第二个可能由 `le_softc[1]` 管理，还可能有它自己的附在 `arpcom` 结构的多播组表。`IN_LOOKUP_MULTI` 宏(图12-14)搜索IP多播表寻找某个特定多播组。

2. IP多播查找

131-146 `IN_LOOKUP_MULTI` 在与接口 `ifp` 相关的多播组表中查找多播组 `addr`。`IFP_TO_IA` 搜索Internet地址表 `in_ifaddr`，寻找与接口 `ifp` 相关的 `in_ifaddr` 结构。如果 `IFP_TO_IA` 找到一个接口，则 `for` 循环搜索它的IP多播表。循环结束后，`inm` 为空或指向匹配的 `in_multi` 结构。

```

131 #define IN_LOOKUP_MULTI(addr, ifp, inm) \
132     /* struct in_addr addr; */ \
133     /* struct ifnet *ifp; */ \
134     /* struct in_multi *inm; */ \
135 { \
136     struct in_ifaddr *ia; \
137 \
138     IFP_TO_IA((ifp), ia); \
139     if (ia == NULL) \
140         (inm) = NULL; \
141     else \
142         for ((inm) = ia->ia_multiaddrs; \
143             (inm) != NULL && (inm)->inm_addr.s_addr != (addr).s_addr; \
144             (inm) = inm->inm_next) \
145             continue; \
146 }

```

in_var.h

in_var.h

图12-14 `IN_LOOKUP_MULTI` 宏

12.7 ip_options结构

运输层通过 `ip_options` 结构包含的多播选项控制多播输出处理。例如，UDP调用 `ip_output` 是：

```

error = ip_output(m, inp->inp_options, &inp->inp_route,
                  inp->inp_socket->so_options & (SO_DONTROUTE|SO_BROADCAST),
                  inp->inp_options);

```

在第22章中我们将看到，`inp` 指向某个Internet协议控制块(PCB)，并且UDP为每个由进程创建的socket关联一个PCB。在PCB内，`inp_options` 是指向某个 `ip_options` 结构的指针。这里我们看到，对每个输出的数据报，都可以给 `ip_output` 传一个不同的 `ip_options` 结构。图12-15是 `ip_options` 结构的定义。

```

100 struct ip_options {
101     struct ifnet *imo_multicast_ifp; /* ifp for outgoing multicasts */
102     u_char imo_multicast_ttl; /* TTL for outgoing multicasts */
103     u_char imo_multicast_loop; /* 1 => hear sends if a member */
104     u_short imo_num_memberships; /* no. memberships this socket */
105     struct in_multi *imo_membership[IP_MAX_MEMBERSHIPS];
106 };

```

ip_var.h

ip_var.h

图12-15 `ip_options` 结构

多播选项

100-106 ip_output通过imo_multicast_ifp指向的接口对输出的多播数据报进行选择。如果imo_multicast_ifp为空,就通过目的站多播组的默认接口(第14章)。

imo_multicast_ttl为外出的多播数据报指定初始的IP TTL。默认值是1,把多播数据报保留在本地网络内。

如果imo_multicast_loop是0,就不回送数据报,也不把数据报提交给正在发送的接口,即使该接口是多播组的成员。如果imo_multicast_loop是1,并且如果正在发送的接口是多播组的成员,就把多播数据报回送给该接口。

最后,整数imo_num_memberships和数组imo_membership维护与该结构相关的{接口,组}对。所有对该表的改变都转告给IP,由IP在所连到的本地网络上宣布成员的变化。imo_membership数组的每个入口都是指向一个in_multi结构的指针,该in_multi结构附在适当接口的in_ifaddr结构上。

12.8 多播的插口选项

图12-16显示了几个IP级的插口选项,提供对ip_moptions结构的进程级访问。

命 令	参 数	函 数	描 述
IP_MULTICAST_IF	struct in_addr	ip_ctloutput	为外出的多播选择默认接口
IP_MULTICAST_TTL	u_char	ip_ctloutput	为外出的多播选择默认的TTL
IP_MULTICAST_LOOP	u_char	ip_ctloutput	允许或使能回送外出的多播
IP_ADD_MEMBERSHIP	struct ip_mreq	ip_ctloutput	加入一个多播组
IP_DROP_MEMBERSHIP	struct ip_mreq	ip_ctloutput	离开一个多播组

图12-16 多播插口选项

我们在图8-31中看到ip_ctloutput函数的整体结构。图12-17显示了与改变和检索多播选项有关的情况语句。

```

448         case PRCO_SETOPT:
449             switch (optname) {
450
451                 /* other set cases */
452
453                 case IP_MULTICAST_IF:
454                 case IP_MULTICAST_TTL:
455                 case IP_MULTICAST_LOOP:
456                 case IP_ADD_MEMBERSHIP:
457                 case IP_DROP_MEMBERSHIP:
458                     error = ip_setmoptions(optname, &inp->inp_moptions, m);
459                     break;
460                 freeit:
461                 default:
462                     error = EINVAL;
463                     break;
464             }
465             if (m)

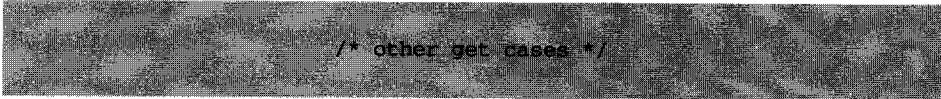
```

图12-17 ip_ctloutput 函数：多播选项

```

499             (void) m_free(m);
500             break;

501         case PRCO_GETOPT:
502             switch (optname) {



539             case IP_MULTICAST_IF:
540             case IP_MULTICAST_TTL:
541             case IP_MULTICAST_LOOP:
542             case IP_ADD_MEMBERSHIP:
543             case IP_DROP_MEMBERSHIP:
544                 error = ip_getmoptions(optname, inp->inp_moptions, mp);
545                 break;

546             default:
547                 error = ENOPROTOOPT;
548                 break;
549             }

```

ip_output.c

图12-17 (续)

486-491 所有多播选项都由 `ip_setmoptions` 和 `ip_getmoptions` 函数处理。
`ip_moptions` 结构由引用传给

539-549 `ip_getmoptions` 和 `ip_setmoptions`，该结构与发布 `ioctl` 命令的那个插口
 关联。

对于 `PRCO_SETOPT` 和 `PRCO_GETOPT` 两种情况，选项不识别时返回的差错码是
 不一样的。`ENOPROTOOPT` 是更合理的选择。

12.9 多播的 TTL 值

多播的 TTL 值难以理解，因为它们有两个作用。TTL 值的基本作用，如 IP 分组一样，是限制分组在互联网内的生存期，避免它在网络内部无限地循环。第二个作用是，把分组限制在管理边界所指定的互联网的某个区域内。管理区域是由一些主观的词语指定的，如“这个结点”，“这个公司”，“这个州”等，并与分组开始的地方有关。与多播分组有关的区域叫做它的辖域(scope)。

RFC 1122 的标准实现把生存期和辖域这两个概念合并到 IP 首部的一个 TTL 值里。当 IP TTL 变成 0 时，除了丢弃该分组外，多播路由器还给每个接口关联了一个 TTL 阈值，限制在该接口上的多播传输。一个要在该接口上传输的分组必须具有大于或等于该接口阈值的 TTL。由于这个原因，多播分组可能会在它的 TTL 到 0 之前就被丢弃了。

阈值是管理员在配置多播路由器时分配的，这些值确定了多播分组的辖域。管理员使用的阈值策略以及数据报的源站与多播接口之间的距离定义多播数据报的初始 TTL 值的意义。

图12-18显示了多种应用程序的推荐TTL值和推荐的阈值。

第一栏是IP首部中的 `ip_ttl` 初始值。第二栏是应用程序专用阈值 ([Casner 1993])。第三栏是与该TTL值相关的推荐的辖域。

例如，一个要与本地结点外的网络通信的接口，多播阈值要被配置成 32。所有开始时

TTL为32(或小于32)的数据报到达该接口时, TTL都小于32(假定源站和路由器之间至少有一跳), 所以它们在被转发到外部网络之前, 都被丢弃了——即使TTL远大于0。

TTL初始值是128的多播数据报可以通过阈值为32的结点接口(只要它以少于 $128 - 32 = 96$ 跳到达接口), 但将被阈值为128的洲际接口丢弃。

ip_ttl	应用程序	辖 域
0		同一接口
1		同一子网
31	本地事件视频	
32		同一地点
63	本地事件音频	
64		同一区域
95	IETF频道2视频	
127	IETF频道1视频	
128		同一州
159	IETF频道2音频	
191	IETF频道1音频	
223	IETF频道2低速率音频	
255	IETF频道1低速率音频, 辖域不受限	

图12-18 IP多播数据报的TTL值

12.9.1 MBONE

Internet上有一个路由器子网支持 IP多播选路。这个多播骨干网称为 MBONE,[Casner 1993] 对其作了描述。它是为了支持用 IP多播的实验——尤其是用音频和视频数据流的实验。在MBONE里, 阈值限制了多种数据流传播的距离。在图 12-18中, 我们看到本地事件视频分组总是以TTL 31开始。阈值为32的接口总是阻止本地事件视频。另外, IETF频道1低速率音频, 只受到IP TTL固有的最大255跳的限制。它能传播通过整个MBONE。MBONE 内的路由器的管理员可以选择阈值, 有选择地接受或丢弃 MBONE数据流。

12.9.2 扩展环搜索

多播TTL的另一种用处是, 只要改变探测数据报的初始 TTL值, 就能在互联网上探测资源。这个技术叫做扩展环搜索(expanding-ring search, [Boggs 1982])。初始TTL 为0的数据报只能到达与外出接口相关的本地网络上的一个资源; TTL为1, 则到达本地子网(如果存在)上的资源; TTL为2, 则到达相距2跳的资源。应用程序指数地增加 TTL的值, 迅速地在大的互联网上探测资源。

RFC 1546 [Partridge、Mendez和Milliken 1993] 描述了一种相关业务的任播(anycasting)。任播依赖一组显著的 IP地址来表示更像多播的多个主机的组。与多播地址不同, 网络必须传播所有任播的分组, 直到它被至少一个主机接收。这样简化了应用程序的实现, 不再进行扩展环搜索。

12.10 ip_setsockopt函数

ip_setsockopt函数块包括一个用来处理各选项的 switch语句。图 12-19是

ip_setmoptions的开始和结束。下面几节讨论 switch 的语句体。

ip_output.c

```

650 int
651 ip_setmoptions(optname, imop, m)
652 int      optname;
653 struct ip_moptions **imop;
654 struct mbuf *m;
655 {
656     int      error = 0;
657     u_char  loop;
658     int      i;
659     struct in_addr addr;
660     struct ip_mreq *mreq;
661     struct ifnet *ifp;
662     struct ip_moptions *imo = *imop;
663     struct route ro;
664     struct sockaddr_in *dst;
665     if (imo == NULL) {
666         /*
667          * No multicast option buffer attached to the pcb;
668          * allocate one and initialize to default values.
669          */
670         imo = (struct ip_moptions *) malloc(sizeof(*imo), M_IPMOPTS,
671                                             M_WAITOK);
672         if (imo == NULL)
673             return (ENOBUFS);
674         *imop = imo;
675         imo->imo_multicast_ifp = NULL;
676         imo->imo_multicast_ttl = IP_DEFAULT_MULTICAST_TTL;
677         imo->imo_multicast_loop = IP_DEFAULT_MULTICAST_LOOP;
678         imo->imo_num_memberships = 0;
679     }
680     switch (optname) {
681         /* switch cases */
682
683     default:
684         error = EOPNOTSUPP;
685         break;
686     }
687     /*
688      * If all options have default values, no need to keep the mbuf.
689      */
690     if (imo->imo_multicast_ifp == NULL &&
691         imo->imo_multicast_ttl == IP_DEFAULT_MULTICAST_TTL &&
692         imo->imo_multicast_loop == IP_DEFAULT_MULTICAST_LOOP &&
693         imo->imo_num_memberships == 0) {
694         free(*imop, M_IPMOPTS);
695         *imop = NULL;
696     }
697     return (error);
698 }

```

ip_output.c

图12-19 ip_setmoptions 函数

650-664 第一个参数, optname, 指明正在改变哪个多播参数。第二个参数, imop, 是指向某个 ip_motions 结构的指针。如果 *imop 不空, ip_setmoptions 修改它所指向的

结构。否则，`ip_setmoptions`分配一个新的`ip_moptions`结构，并把它地址保存在`*imop`里。如果没有内存了，`ip_setmoptions`立即返回`ENOBUFS`。后面的所有错误都通告`error`，`error`在函数的最后被返回给调用方。第三个参数，`m`，指向存放要改变选项数据的`mbuf`(图12-16的第二栏)。

1. 构造默认值

665-679 当分配一个新的`ip_moptions`结构时，`ip_setmoptions`把默认的多播接口指针初始化为空，把默认TTL初始化为1(`IP_DEFAULT_MULTICAST_TTL`)，使能多播数据报的回送，并清除组成员表。有了这些默认值后，`ip_output`查询路由表选择一个输出的接口，多播被限制在本地网络中，并且，如果输出的接口是目的多播组的成员，则系统将接收它自己的多播发送。

2. 进程选项

680-860 `ip_setmoptions`体由一个`switch`语句组成，其中对每种选项都有一个`case`语句。`default`情况(对未知选项)把`error`设成`EOPNOTSUPP`。

3. 如果默认值是OK，丢弃结构

861-872 `switch`语句之后，`ip_setmoptions`检查`ip_moptions`结构。如果所有多播选项与它们对应的默认值匹配，就不再需要该结构，将其释放。`ip_setmoptions`返回0或公布的差错码。

12.10.1 选择一个明确的多播接口：IP_MULTICAST_IF

当`optname`是`IP_MULTICAST_IF`时，传给`ip_setmoptions`的`mbuf`中就包含了多播接口的单播地址，该地址指定了在这个插口上发送的多播所使用的特定接口。图 12-20是这个选项的程序。

```

681     case IP_MULTICAST_IF:
682         /*
683          * Select the interface for outgoing multicast packets.
684          */
685         if (m == NULL || m->m_len != sizeof(struct in_addr)) {
686             error = EINVAL;
687             break;
688         }
689         addr = *(mtod(m, struct in_addr *));
690         /*
691          * INADDR_ANY is used to remove a previous selection.
692          * When no interface is selected, a default one is
693          * chosen every time a multicast packet is sent.
694          */
695         if (addr.s_addr == INADDR_ANY) {
696             imo->imo_multicast_ifp = NULL;
697             break;
698         }
699         /*
700          * The selected interface is identified by its local
701          * IP address. Find the interface and confirm that
702          * it supports multicasting.
703          */

```

ip_output.c

图12-20 `ip_setmoptions` 函数：选择多播输出接口

```

704     INADDR_TO_IFP(addr, ifp);
705     if (ifp == NULL || (ifp->if_flags & IFF_MULTICAST) == 0) {
706         error = EADDRNOTAVAIL;
707         break;
708     }
709     imo->imo_multicast_ifp = ifp;
710     break;

```

—ip_output.c

图12-20 (续)

1. 验证

681-698 如果没有提供 mbuf，或者 mbuf 中的数不是一个 in_addr 结构的大小，则 ip_setmoptions 通告一个 EINVAL 差错；否则把数据复制到 addr。如果接口地址是 INADDR_ANY，则丢弃所有前面选定的接口。对后面用这个 ip_moptions 结构的多播，将根据它们的目的多播组进行选路，而不再通过一个明确命名的接口 (图12-40)。

2. 选择默认接口

699-710 如果 addr 中有地址，就由 INADDR_TO_IFP 找到匹配接口的位置。如果找不到匹配或接口不支持多播，就发布 EADDRNOTAVAIL。否则，匹配接口 ifp 成为与这个 ip_moptions 结构相关的输出请求的多播接口。

12.10.2 选择明确的多播 TTL : IP_MULTICAST_TTL

当 optname 是 IP_MULTICAST_TTL 时，缓存中有一个字节指定输出多播的 IP TTL。这个 TTL 是 ip_output 在每个发往相关插口的多播数据报中插入的。图 12-21 是该选项的程序。

```

711     case IP_MULTICAST_TTL:
712         /*
713          * Set the IP time-to-live for outgoing multicast packets.
714          */
715         if (m == NULL || m->m_len != 1) {
716             error = EINVAL;
717             break;
718         }
719         imo->imo_multicast_ttl = *(mtod(m, u_char *));
720         break;

```

—ip_output.c

—ip_output.c

图12-21 ip_setmoptions 函数：选择明确的多播 TTL

验证和选项默认的 TTL

711-720 如果缓存中有一个字节的数据，就把它复制到 imo_multicast_ttl。否则，发布 EINVAL。

12.10.3 选择多播环回 : IP_MULTICAST_LOOP

通常，多播应用程序有两种形式：

- 一个系统内一个发送方和多个远程接收方的应用程序。这种配置中，只有一个本地进程向多播组发送数据报，所以无需回送输出的多播。这样的例子有多播选路守护进程和会议系统。
- 一个系统内的多个发送方和接收方。必须回送数据报，确保每个进程接收到系统其他发

送方的传送。

IP_MULTICAST_LOOP选项(图12-22)为ip_moptions结构选择回送策略。

```

721     case IP_MULTICAST_LOOP:
722         /*
723          * Set the loopback flag for outgoing multicast packets.
724          * Must be zero or one.
725          */
726         if (m == NULL || m->m_len != 1 ||
727             (loop = *(mtod(m, u_char *))) > 1) {
728             error = EINVAL;
729             break;
730         }
731         imo->imo_multicast_loop = loop;
732         break;

```

ip_output.c

ip_output.c

图12-22 ip_setmoptions 函数：选择多播环回

验证和选择环回策略

721-732 如果m为空，或者没有1字节数据，或者该字节不是0或1，就发布EINVAL。否则，将该字节复制到imo_multicast_loop。0指明不要把数据报回送，1允许环回机制。

图12-23显示了多播数据报的最大辖域值之间的关系： imo_multicast_ttl和 imo_multicast_loop。

imo_multicast-		Recipients			
_loop	_ttl	Outgoing Interface?	Local Network?	Remote Networks?	Other Interfaces?
1	0	•			
1	1	•	•		
1	>1	•	•	•	see text

图12-23 环回和TTL对多播辖域的影响

图12-23显示了根据发送的环回策略，指定的 TTL值接收多播分组的接口的设置。如果硬件接收自己的发送，则不管采用什么环回策略，都接收分组。数据报可能通过选路穿过该网络，并到达与系统相连的其他接口(习题12.6)。如果发送系统本身是一个多播路由器，输出的分组可能被转发到其他接口，但是，只有一个接口接受它们进行输入处理(第14章)。

12.11 加入一个IP多播组

除了内核自动加入(图6-17)的IP所有主机组外，其他组成员是由进程明确发出请求产生的。加入(或离开)多播组选项比其他选项更多使用。必须修改接口的 in_multi表以及其他链路层多播结构，如我们在以太网中讨论的 ether_multi。

当optname是IP_ADDMEMBERSHIP时，mbuf中的数据是一个如图 12-24所示的 ip_mreq结构。

```

148 struct ip_mreq {
149     struct in_addr imr_multiaddr; /* IP multicast address of group */
150     struct in_addr imr_interface; /* local IP address of interface */
151 };

```

in.h

in.h

图12-24 ip_mreq 结构

148-151 imr_multiaddr指定多播组，imr_interface用相关的单播IP地址指定接口。
ip_mreq结构指定{接口，组}对表示成员的变化。

图12-25显示了加入和离开与我们的以太网接口例子相关的多播组时，所调用的函数。

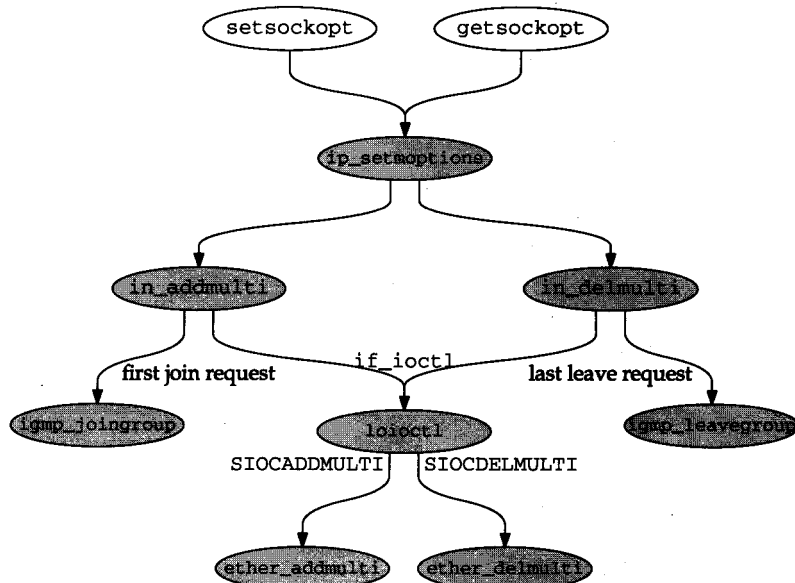


图12-25 加入和离开一个多播组

我们从ip_setsockopt(图12-26)的IP_ADD_MEMBERSHIP情况开始，在这里修改ip_moptions结构。然后我们跟踪请求通过IP层、以太网驱动程序，一直到物理设备——在这里，是LANCE以太网网卡。

```

733 case IP_ADD_MEMBERSHIP:                                     ip_output.c
734     /*
735     * Add a multicast group membership.
736     * Group must be a valid IP multicast address.
737     */
738     if (m == NULL || m->m_len != sizeof(struct ip_mreq)) {
739         error = EINVAL;
740         break;
741     }
742     mreq = mtod(m, struct ip_mreq *);
743     if (!IN_MULTICAST(ntohl(mreq->imr_multiaddr.s_addr))) {
744         error = EINVAL;
745         break;
746     }
747     /*
748     * If no interface address was provided, use the interface of
749     * the route to the given multicast address.
750     */
751     if (mreq->imr_interface.s_addr == INADDR_ANY) {
752         ro.ro_rt = NULL;
753         dst = (struct sockaddr_in *) &ro.ro_dst;
754         dst->sin_len = sizeof(*dst);
755         dst->sin_family = AF_INET;

```

图12-26 ip_setsockopt 函数：加入一个多播组

```

756         dst->sin_addr = mreq->imr_multiaddr;
757         rtalloc(&ro);
758         if (ro.ro_rt == NULL) {
759             error = EADDRNOTAVAIL;
760             break;
761         }
762         ifp = ro.ro_rt->rt_ifp;
763         rtfree(ro.ro_rt);
764     } else {
765         INADDR_TO_IFP(mreq->imr_interface, ifp);
766     }
767     /*
768     * See if we found an interface, and confirm that it
769     * supports multicast.
770     */
771     if (ifp == NULL || (ifp->if_flags & IFF_MULTICAST) == 0) {
772         error = EADDRNOTAVAIL;
773         break;
774     }
775     /*
776     * See if the membership already exists or if all the
777     * membership slots are full.
778     */
779     for (i = 0; i < imo->imo_num_memberships; ++i) {
780         if (imo->imo_membership[i]->inm_ifp == ifp &&
781             imo->imo_membership[i]->inm_addr.s_addr
782             == mreq->imr_multiaddr.s_addr)
783             break;
784     }
785     if (i < imo->imo_num_memberships) {
786         error = EADDRINUSE;
787         break;
788     }
789     if (i == IP_MAX_MEMBERSHIPS) {
790         error = ETOOMANYREFS;
791         break;
792     }
793     /*
794     * Everything looks good; add a new record to the multicast
795     * address list for the given interface.
796     */
797     if ((imo->imo_membership[i] =
798         in_addmulti(&mreq->imr_multiaddr, ifp)) == NULL) {
799         error = ENOBUFS;
800         break;
801     }
802     ++imo->imo_num_memberships;
803     break;

```

—ip_output.c

图12-26 (续)

1. 验证

733-746 ip_setmoptions从验证该请求开始。如果没有传给mbuf，或缓存的大小不对，或结构的地址(imr_multiaddr)不是一个多播组地址，则ip_setmoptions发布ENIVAL。Mreq指向有效ip_mreq地址。

2. 找到接口

747-774 如果接口的单播地址(imr_interface)是INADDR_ANY，则ip_setmoptions

必须找到指定组的默认接口。该多播组构造一个 `route` 结构，作为目的地址，并传给 `rtalloc`，由 `rtalloc` 为多播组找到一个路由器。如果没有路由器可用，则请求失败，产生错误 `EADDRNOTAVAIL`。如果找到路由器，则在 `ifp` 中保存指向路由器外出接口的指针，而不再需要路由器入口，将其释放。

如果 `imr_interface` 不是 `INADDR_ANY`，则请求一个明确的接口。`INADDR_TO_IFP` 宏用请求的单播地址搜索接口。如果没有找到接口或者它支持多播，则请求失败，产生错误 `EADDRNOTAVAIL`。

8.5节描述了 `route` 结构，19.2节描述了 `rtalloc` 函数，第14章描述了用路由选择表选择多播接口。

3. 已经是成员了？

775-792 对请求做的最后检查是检查 `imo_membership` 数组，看看所选接口是否已经是请求组的成员。如果 `for` 循环找到一个匹配，或者成员数组为空，则发布 `EADDRINUSE` 或 `ETOOMANYREFS`，并终止对这个选项的处理。

4. 加入多播组

793-803 此时，请求似乎是合理的了。`in_addmulti` 安排 IP 开始接收该组的多播数据报。`in_addmulti` 返回的指针指向一个新的或已存在的 `in_multi` 结构(图12-12)，该结构位于接口的多播组表中。这个结构被保存在成员数组中，并且把数组的大小加 1。

12.11.1 `in_addmulti` 函数

`in_addmulti` 和相应的 `in_delmulti`(图12-27和图12-36)维护接口已加入多播组的表。加入请求或者在接口表中增加一个新的 `in_multi` 结构，或者增加对某个已有结构的引用次数。

```

469 struct in_multi *
470 in_addmulti(ap, ifp)
471 struct in_addr *ap;
472 struct ifnet *ifp;
473 {
474     struct in_multi *inm;
475     struct ifreq ifr;
476     struct in_ifaddr *ia;
477     int s = splnet();
478
479     /*
480      * See if address already in list.
481      */
482     IN_LOOKUP_MULTI(*ap, ifp, inm);
483     if (inm != NULL) {
484         /*
485          * Found it; just increment the reference count.
486          */
487         ++inm->inm_refcount;
488     } else {
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

图12-27 `in_addmulti` 函数：前半部分

1. 已经是一个成员了

469-487 `ip_setmoptions` 已经证实 `ap` 指向一个 D 类多播地址，`ifp` 指向一个能够多播的

接口。IN_LOOKUP_MULTI(图12-14)确定接口是否已经是该组的一个成员。如果是, 则in_addmulti更新引用计数后返回。

如果接口还不是该组的成员, 则执行图 12-28中的程序。

```

487     } else {
488         /*
489          * New address; allocate a new multicast record
490          * and link it into the interface's multicast list.
491          */
492         inm = (struct in_multi *) malloc(sizeof(*inm),
493                                         M_IPMADDR, M_NOWAIT);
494         if (inm == NULL) {
495             splx(s);
496             return (NULL);
497         }
498         inm->inm_addr = *ap;
499         inm->inm_ifp = ifp;
500         inm->inm_refcount = 1;
501         IFP_TO_IA(ifp, ia);
502         if (ia == NULL) {
503             free(inm, M_IPMADDR);
504             splx(s);
505             return (NULL);
506         }
507         inm->inm_ia = ia;
508         inm->inm_next = ia->ia_multiaddrs;
509         ia->ia_multiaddrs = inm;
510         /*
511          * Ask the network driver to update its multicast reception
512          * filter appropriately for the new address.
513          */
514         ((struct sockaddr_in *) &ifr.ifr_addr)->sin_family = AF_INET;
515         ((struct sockaddr_in *) &ifr.ifr_addr)->sin_addr = *ap;
516         if ((ifp->if_ioctl == NULL) ||
517             (*ifp->if_ioctl)(ifp, SIOCADDMULTI, (caddr_t) &ifr) != 0) {
518             ia->ia_multiaddrs = inm->inm_next;
519             free(inm, M_IPMADDR);
520             splx(s);
521             return (NULL);
522         }
523         /*
524          * Let IGMP know that we have joined a new IP multicast group.
525          */
526         igmp_joingroup(inm);
527     }
528     splx(s);
529     return (inm);
530 }

```

图12-28 in_addmulti 函数：后半部分

2. 更新in_multi表

487-509 如果接口还不是成员, 则in_addmulti分配并初始化一个新的in_multi结构, 将该结构插到接口的in_ifaddr(图12-13)结构中ia_multiaddrs表的前端。

3. 更新接口, 通告变化

510-530 如果接口驱动程序已经定义了一个if_ioctl函数, 则in_addmulti构造一个

包含了该组地址的 `ifreq` 结构(图4-23), 并把 `SIOCADDMULTI` 请求传给接口。如果接口拒绝该请求, 则把 `in_multi` 结构从链表中断开, 释放掉。最后, `in_addmulti` 调用 `igmp_joingroup`, 把成员变化信息传播给其他主机和路由器。

`in_addmulti` 返回一个指针, 该指针指向 `in_multi` 结构, 或者如果出错, 则为空。

12.11.2 `slioclt` 和 `lioclt` 函数: `SIOCADDMULTI` 和 `SIOCDELMULTI`

SLIP 和 环回接口的多播组处理很简单: 除了检查差错外, 不做其他事情。图 12-29 显示了 SLIP 处理。

```

673     case SIOCADDMULTI:
674     case SIOCDELMULTI:
675         ifr = (struct ifreq *) data;
676         if (ifr == 0) {
677             error = EAFNOSUPPORT;    /* XXX */
678             break;
679         }
680         switch (ifr->ifr_addr.sa_family) {
681             case AF_INET:
682                 break;
683             default:
684                 error = EAFNOSUPPORT;
685                 break;
686         }
687         break;

```

if_sl.c

图12-29 `slioclt` 函数: 多播处理

673-687 不管请求为空还是不适用于 `AF_INET` 协议族, 都返回 `EAFNOSUPPORT`。

图12-30显示了环回处理。

```

152     case SIOCADDMULTI:
153     case SIOCDELMULTI:
154         ifr = (struct ifreq *) data;
155         if (ifr == 0) {
156             error = EAFNOSUPPORT;    /* XXX */
157             break;
158         }
159         switch (ifr->ifr_addr.sa_family) {
160             case AF_INET:
161                 break;
162             default:
163                 error = EAFNOSUPPORT;
164                 break;
165         }
166         break;

```

if_loop.c

图12-30 `lioclt` 函数: 多播处理

152-166 环回接口的处理等价于图 12-29 中 SLIP 的程序。不管请求为空还是不适用于 `AF_INET` 协议族, 都返回 `EAFNOSUPPORT`。

12.11.3 leioctl函数：SIOCADDMULTI和SIOCDELMULTI

在图4-2中，我们讲到LANCE以太网驱动程序中的leioctl和if_ioctl函数。图12-31是处理SIOCADDMULTI和SIOCDELMULTI的程序。

```

657     case SIOCADDMULTI:
658     case SIOCDELMULTI:
659         /* Update our multicast list */
660         error = (cmd == SIOCADDMULTI) ?
661             ether_addmulti((struct ifreq *) data, &le->sc_ac) :
662             ether_delmulti((struct ifreq *) data, &le->sc_ac);

663         if (error == ENETRESET) {
664             /*
665              * Multicast list has changed; set the hardware
666              * filter accordingly.
667              */
668             lereset(ifp->if_unit);
669             error = 0;
670         }
671         break;

```

if_le.c

if_le.c

图12-31 leioctl 函数：多播处理

657-671 leioctl把增加和删除请求直接传给ether_addmulti或ether_delmulti函数。如果请求改变了该物理硬件必须接收的IP多播地址集，则两个函数都返回ENETRESET。如果发生了这种情况，则leioctl调用lereset，用新的多播接收表重新初始化该硬件。

我们没有显示lereset，因为它是LANCE以太网硬件专用的。对多播来说，lereset安排硬件接收所有寻址到ether_multi中与该接口相关的多播地址的帧。如果多播表中的每个入口是一个地址，则LANCE驱动程序采用散列机制。散列程序使硬件可以有选择地接收分组。如果驱动程序发现某个入口是一个地址范围，它废除散列策略，配置硬件接收所有多播分组。如果驱动程序必须回到接收所有以太网多播地址的状态，lereset就在返回时把IFP_ALLMULTI标志位置位。

12.11.4 ether_addmulti函数

所有以太网驱动程序都调用ether_addmulti函数处理SIOCADDMULTI请求。这个函数把IP D类地址映射到合适的以太网多播地址(图12-5)上，并更新ether_multi表。图12-32是ether_multi函数的前半部。

1. 初始化地址范围

366-399 首先，ether_addmulti初始化addrlo和addrhi(两者都是六个无符号字符)中的多播地址范围。如果所请求的地址来自AF_UNSPEC族，ether_addmulti假定该地址是一个明确的以太网多播地址，并把它复制到addrlo和addrhi中。如果地址属于AF_INET族，并且是INADDR_ANY(0.0.0.0)，ether_addmulti把addrlo初始化成ether_ipmulticast_min，把addrhi初始化成ether_ipmulticast_max。这两个以太网地址常量定义为：

```

u_char ether_ipmulticast_min[6] = { 0x01, 0x00, 0x5e, 0x00, 0x00, 0x00 };
u_char ether_ipmulticast_max[6] = { 0x01, 0x00, 0x5e, 0x7f, 0xff, 0xff };

```

```

366 int
367 ether_addmulti(ifr, ac)
368 struct ifreq *ifr;
369 struct arpcom *ac;
370 {
371     struct ether_multi *enm;
372     struct sockaddr_in *sin;
373     u_char  addrlo[6];
374     u_char  addrhi[6];
375     int     s = splimp();

376     switch (ifr->ifr_addr.sa_family) {

377     case AF_UNSPEC:
378         bcopy(ifr->ifr_addr.sa_data, addrlo, 6);
379         bcopy(addrlo, addrhi, 6);
380         break;

381     case AF_INET:
382         sin = (struct sockaddr_in *) &(ifr->ifr_addr);
383         if (sin->sin_addr.s_addr == INADDR_ANY) {
384             /*
385              * An IP address of INADDR_ANY means listen to all
386              * of the Ethernet multicast addresses used for IP.
387              * (This is for the sake of IP multicast routers.)
388              */
389             bcopy(ether_ipmulticast_min, addrlo, 6);
390             bcopy(ether_ipmulticast_max, addrhi, 6);
391         } else {
392             ETHER_MAP_IP_MULTICAST(&sin->sin_addr, addrlo);
393             bcopy(addrlo, addrhi, 6);
394         }
395         break;

396     default:
397         splx(s);
398         return (EAFNOSUPPORT);
399     }
}

```

if_ethersubr.c

if_ethersubr.c

图12-32 ether_addmulti 函数：前半

与etherbroadcastaddr(4.3节)一样，这是一个很方便地定义一个 48 bit 常量的方法。

IP多播路由器必须监听所有 IP多播。把组指定为 INADDR_ANY，被认为是请求加入所有 IP多播组。在这种情况下，所选择的以太网地址范围跨越了分配给 IANA的整个 IP多播地址块。

当mrouted (8)守护程序开始对 到多播接口的 分组进行路选时，它用 INADDR_ANY发布一个 SIOCADDMULTI请求。

ETHER_MAP_IP_MULTICAST把其他特定的 IP多播组映射到合适的以太网多播地址。当发生EAFNOSUPPORT错误时，将拒绝对其他地址族的请求。

尽管以太网多播表支持地址范围，但是除了列举出所有地址外，进程或内核无法对某个特定范围提出请求，因为总是把 addrlo和addrhi设成同一值。

ether_addmulti的第二部分，显示如图 12-33，证实地址范围，并且，如果该地址是

新的，就把它加入表中。

```

400      /*
401      * Verify that we have valid Ethernet multicast addresses.
402      */
403      if ((addrlo[0] & 0x01) != 1 || (addrhi[0] & 0x01) != 1) {
404          splx(s);
405          return (EINVAL);
406      }
407      /*
408      * See if the address range is already in the list.
409      */
410      ETHER_LOOKUP_MULTI(addrlo, addrhi, ac, enm);
411      if (enm != NULL) {
412          /*
413          * Found it; just increment the reference count.
414          */
415          ++enm->enm_refcount;
416          splx(s);
417          return (0);
418      }
419      /*
420      * New address or range; malloc a new multicast record
421      * and link it into the interface's multicast list.
422      */
423      enm = (struct ether_multi *) malloc(sizeof(*enm), M_IFMADDR, M_NOWAIT);
424      if (enm == NULL) {
425          splx(s);
426          return (ENOBUFS);
427      }
428      bcopy(addrlo, enm->enm_addrlo, 6);
429      bcopy(addrhi, enm->enm_addrhi, 6);
430      enm->enm_ac = ac;
431      enm->enm_refcount = 1;
432      enm->enm_next = ac->ac_multiaddrs;
433      ac->ac_multiaddrs = enm;
434      ac->ac_multicnt++;
435      splx(s);
436      /*
437      * Return ENETRESET to inform the driver that the list has changed
438      * and its reception filter should be adjusted accordingly.
439      */
440      return (ENETRESET);
441 }

```

图12-33 ether_addmulti 函数：后一半

2. 已经在接收

400-418 ether_addmulti检查高地址和低地址的多播比特位(图4-12)，保证它们是真正的以太网多播地址。ETHER_LOOKUP_MULTI(图12-9)确定硬件是否已经对指定的地址开始监听。如果是，则增加匹配的 ether_multi结构中的引用计数(enm_refcount)，并且 ether_addmulti返回0。

3. 更新ether_multi表

419-441 如果这是一个新的地址范围，则分配并初始化一个新的 ether_multi结构，把它链到接口 arpcom结构(图12-8)中的ac_multiaddrs表上。如果 ether_addmulti返回

ENETRESET, 则调用它的设备驱动程序就知道多播表被改变了, 必须更新硬件接收过滤器。

图12-34显示在LANCE以太网接口加入所有主机组后, `ip_moptions`、`in_multi`和`ether_multi`结构之间的关系。

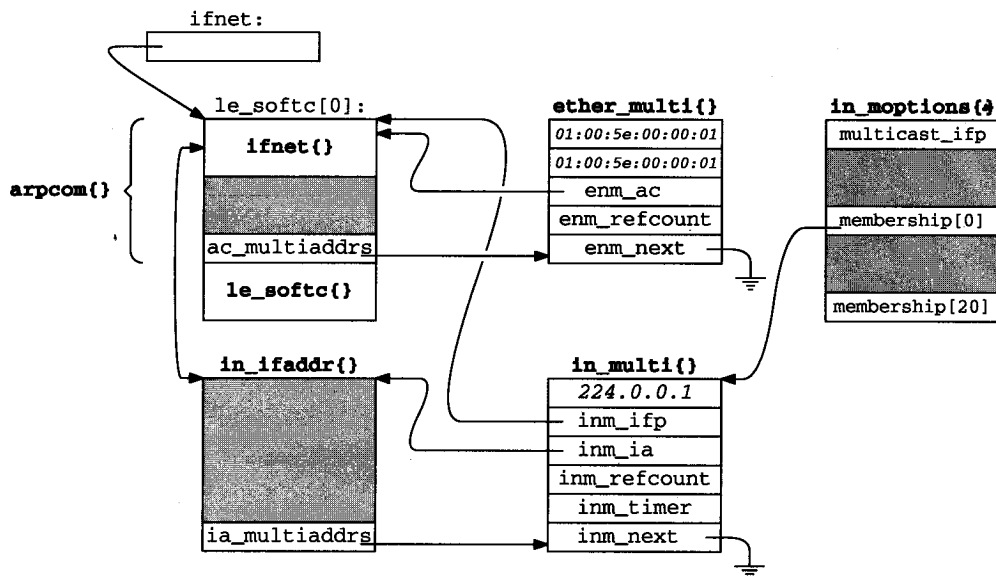


图12-34 多播数据结构的整体图

12.12 离开一个IP多播组

通常情况下, 离开一个多播组的步骤是加入一个多播组的步骤的反序。更新`ip_moptions`结构中的成员表、IP接口的`in_multi`表和设备的`ether_multi`表。首先, 我们回到`ip_setmoptions`中的`IP_DROP_MEMBERSHIP`情况语句, 如图12-35所示。

```

804     case IP_DROP_MEMBERSHIP:
805         /*
806          * Drop a multicast group membership.
807          * Group must be a valid IP multicast address.
808          */
809         if (m == NULL || m->m_len != sizeof(struct ip_mreq)) {
810             error = EINVAL;
811             break;
812         }
813         mreq = mtod(m, struct ip_mreq *);
814         if (!IN_MULTICAST(ntohl(mreq->imr_multiaddr.s_addr))) {
815             error = EINVAL;
816             break;
817         }
818         /*
819          * If an interface address was specified, get a pointer
820          * to its ifnet structure.
821          */
822         if (mreq->imr_interface.s_addr == INADDR_ANY)
823             ifp = NULL;
824         else {
825             INADDR_TO_IFP(mreq->imr_interface, ifp);

```

ip_output.c

图12-35 `ip_setmoptions` 函数: 离开一个多播组

```

826         if (ifp == NULL) {
827             error = EADDRNOTAVAIL;
828             break;
829         }
830     }
831     /*
832     * Find the membership in the membership array.
833     */
834     for (i = 0; i < imo->imo_num_memberships; ++i) {
835         if ((ifp == NULL ||
836             imo->imo_membership[i]->inm_ifp == ifp) &&
837             imo->imo_membership[i]->inm_addr.s_addr ==
838             mreq->imr_multiaddr.s_addr)
839             break;
840     }
841     if (i == imo->imo_num_memberships) {
842         error = EADDRNOTAVAIL;
843         break;
844     }
845     /*
846     * Give up the multicast address record to which the
847     * membership points.
848     */
849     in_delmulti(imo->imo_membership[i]);
850     /*
851     * Remove the gap in the membership array.
852     */
853     for (++i; i < imo->imo_num_memberships; ++i)
854         imo->imo_membership[i - 1] = imo->imo_membership[i];
855     --imo->imo_num_memberships;
856     break;

```

ip_output.c

图12-35 (续)

1. 验证

804-830 存储器缓存中必然包含一个 `ip_mreq` 结构，其中的 `imr_multiaddr` 必须是一个多播组，而且必须有一个接口与单播地址 `imr_interface` 相关。如果这些条件不满足，则发布 `EINVAL` 和 `EADDRNOTAVAIL` 错误信息，继续到该 `switch` 语句的最后进行处理。

2. 删除成员引用

831-856 `for` 循环用请求的 {接口, 组} 对在组成员表中寻找一个 `in_multi` 结构。如果没有找到，则发布 `EADDRNOTAVAIL` 错误信息。如果找到了，则 `in_delmulti` 更新 `in_multi` 表，并且第二个 `for` 循环把成员数组中不用的入口删去，把后面的入口向前移动。数组的大小也被相应更新。

12.12.1 in_delmulti函数

因为可能会有多个进程接收多播数据报，所以调用 `in_delmulti` (图12-36) 的结果是，当对 `in_multi` 结构没有引用时，只离开指定的多播组。

更新 in_multi 结构

534-567 `in_delmulti` 一开始就减少 `in_multi` 结构的引用计数，如果该计数非零，则返回。如果该计数减为 0，则表明在指定的 {接口, 组} 对上，没有其他进程等待多播数据报。调用 `igmp_leavegroup`，但该函数不做任何事情，我们将在 13.8 节中看到。

`for` 循环遍历 `in_multi` 结构的链表，找到匹配的结构。

```

534 int
535 in_delmulti(inm)
536 struct in_multi *inm;
537 {
538     struct in_multi **p;
539     struct ifreq ifr;
540     int s = splnet();

541     if (--inm->inm_refcount == 0) {
542         /*
543          * No remaining claims to this record; let IGMP know that
544          * we are leaving the multicast group.
545          */
546         igmp_leavegroup(inm);
547         /*
548          * Unlink from list.
549          */
550         for (p = &inm->inm_ia->ia_multiaddrs;
551              *p != inm;
552              p = &(*p)->inm_next)
553             continue;
554         *p = (*p)->inm_next;
555         /*
556          * Notify the network driver to update its multicast reception
557          * filter.
558          */
559         ((struct sockaddr_in *) &(ifr.ifr_addr))->sin_family = AF_INET;
560         ((struct sockaddr_in *) &(ifr.ifr_addr))->sin_addr =
561             inm->inm_addr;
562         (*inm->inm_ifp->if_ioctl) (inm->inm_ifp, SIOCDELMULTI,
563                                   (caddr_t) &ifr);
564         free(inm, M_IPMADDR);
565     }
566     splx(s);
567 }

```

图12-36 in_delmulti 函数

for循环体只包含一个continue语句。但所有工作都由循环上面的表达式做了，并不需要continue语句，只是因为它比只有一个分号更清楚一些。

图12-9中的宏ETHER_LOOKUP_MULTI不用continue语句，仅有一个分号几乎是不可检测的。

循环结束后，把匹配的 in_multi结构从链表上断开， in_delmulti向接口发布 SIOCDELMULTI请求，以便更新所有设备专用的数据结构。对以太网接口来说，这意味着更新ether_multi表。最后释放in_multi结构。

LANCE驱动程序的 SIOCDELMULTI情况语句包括在图 12-31中，这里我们也讨论了SIOCADDRMULTI情况。

12.12.2 ether_delmulti函数

当IP释放与某个以太网设备相关的 in_multi结构时，该设备也可能释放匹配的 ether_multi结构。我们说“可能”是因为IP忽略其他监听IP多播的软件。当 ether_

multi结构的引用计数变成0时，就释放该结构。图12-37是 ether_delmulti函数。

——if_ethersubr.c

```

445 int
446 ether_delmulti(ifr, ac)
447 struct ifreq *ifr;
448 struct arpcom *ac;
449 {
450     struct ether_multi *enm;
451     struct ether_multi **p;
452     struct sockaddr_in *sin;
453     u_char  addrlo[6];
454     u_char  addrhi[6];
455     int      s = splimp();

456     switch (ifr->ifr_addr.sa_family) {

457     case AF_UNSPEC:
458         bcopy(ifr->ifr_addr.sa_data, addrlo, 6);
459         bcopy(addrlo, addrhi, 6);
460         break;

461     case AF_INET:
462         sin = (struct sockaddr_in *) &(ifr->ifr_addr);
463         if (sin->sin_addr.s_addr == INADDR_ANY) {
464             /*
465              * An IP address of INADDR_ANY means stop listening
466              * to the range of Ethernet multicast addresses used
467              * for IP.
468              */
469             bcopy(ether_ipmulticast_min, addrlo, 6);
470             bcopy(ether_ipmulticast_max, addrhi, 6);
471         } else {
472             ETHER_MAP_IP_MULTICAST(&sin->sin_addr, addrlo);
473             bcopy(addrlo, addrhi, 6);
474         }
475         break;

476     default:
477         splx(s);
478         return (EAFNOSUPPORT);
479     }

480     /*
481      * Look up the address in our list.
482      */
483     ETHER_LOOKUP_MULTI(addrlo, addrhi, ac, enm);
484     if (enm == NULL) {
485         splx(s);
486         return (ENXIO);
487     }
488     if (--enm->enm_refcount != 0) {
489         /*
490          * Still some claims to this record.
491          */
492         splx(s);
493         return (0);
494     }
495     /*
496      * No remaining claims to this record; unlink and free it.
497      */

```

图12-37 ether_delmulti 函数

```

498     for (p = &enm->enm_ac->ac_multiaddrs;
499          *p != enm;
500          p = &(*p)->enm_next)
501         continue;
502     *p = (*p)->enm_next;
503     free(enm, M_IFMADDR);
504     ac->ac_multicnt--;
505     splx(s);
506     /*
507      * Return ENETRESET to inform the driver that the list has changed
508      * and its reception filter should be adjusted accordingly.
509      */
510     return (ENETRESET);
511 }

```

if_ethersubr.c

图12-37 (续)

445-479 ether_delmulti函数用ether_addrmulti函数采用的同一方法初始化addrlo和addrhi数组。

1. 寻找ether_multi结构

480-494 ETHER_LOOKUP_MULTI寻找匹配的ether_multi结构。如果没有找到，则返回ENXIO。如果找到匹配的结构，则把引用计数减去1。如果此时引用计数非零，ether_delmulti立即返回。在这种情况下，可能会由于其他协议也要接收相同的多播分组而释放该结构。

2. 删除ether_multi结构

495-511 for循环搜索ether_multi表，寻找匹配的地址范围，并从链表中断开匹配的结构，将它释放掉。最后，更新链表的长度，返回ENETRESET，使设备驱动程序可以更新它的硬件接收过滤器。

12.13 ip_getmoptions函数

取得当前的选项设置比设置它们要容易。ip_getmoptions完成所有的工作，如图12-38所示。

复制选项数据和返回

876-914 ip_getmoptions的三个参数是：optname，要取得的选项；imo，ip_moptions结构；mp，一个指向mbuf的指针。m_get分配一个mbuf存放该选项数据。这三个选项的指针(分别是addr、ttl和loop)被初始化为指向mbuf的数据域，而mbuf的长度被设成选项数据的长度。

对IP_MULTICAST_IF，返回IFP_TO_IA发现的单播地址，或者如果没有选择明确的多播接口，则返回INADDR_ANY。

对IP_MULTICAST_TTL，返回imo_multicast_ttl，或者如果没有选择明确的TTL，则返回1(IP_DEFAULT_MULTICAST_TTL)。

对IP_MULTICAST_LOOP，返回imo_multicast_loop，或者如果没有选择明确的多播循环策略，则返回1(IP_DEFAULT_MULTICAST_LOOP)。

最后，如果不识别该选项，则返回EOPNOTSUPP。

ip_output.c

```

876 int
877 ip_getmoptions(optname, imo, mp)
878 int    optname;
879 struct ip_moptions *imo;
880 struct mbuf **mp;
881 {
882     u_char *ttl;
883     u_char *loop;
884     struct in_addr *addr;
885     struct in_ifaddr *ia;
886
887     *mp = m_get(M_WAIT, MT_SOOPTS);
888
889     switch (optname) {
890     case IP_MULTICAST_IF:
891         addr = mtod(*mp, struct in_addr *);
892         (*mp)->m_len = sizeof(struct in_addr);
893         if (imo == NULL || imo->imo_multicast_ifp == NULL)
894             addr->s_addr = INADDR_ANY;
895         else {
896             IFP_TO_IA(imo->imo_multicast_ifp, ia);
897             addr->s_addr = (ia == NULL) ? INADDR_ANY
898                 : IA_SIN(ia)->sin_addr.s_addr;
899         }
900         return (0);
901     case IP_MULTICAST_TTL:
902         ttl = mtod(*mp, u_char *);
903         (*mp)->m_len = 1;
904         *ttl = (imo == NULL) ? IP_DEFAULT_MULTICAST_TTL
905             : imo->imo_multicast_ttl;
906         return (0);
907     case IP_MULTICAST_LOOP:
908         loop = mtod(*mp, u_char *);
909         (*mp)->m_len = 1;
910         *loop = (imo == NULL) ? IP_DEFAULT_MULTICAST_LOOP
911             : imo->imo_multicast_loop;
912         return (0);
913     default:
914         return (EOPNOTSUPP);
915     }
916 }

```

ip_output.c

图12-38 ip_getmoptions 函数

12.14 多播输入处理：ipintr函数

到目前为止，我们已经讨论了多播选路，组成员关系，以及多种与 IP 和以太网多播有关的数据结构，现在转入讨论对多播数据报的处理。

在图4-13中，我们看到 ether_input 检测到达的以太网多播分组，在把一个 IP 分组放到 IP 输入队列之前 (ipintrq)，把 mbuf 首部的 M_MCAST 标志位置位。ipintr 函数按顺序处理每个分组。我们在 ipintr 中省略的多播处理程序如图 12-39 所示。

该段代码来自于 ipintr 程序，用来确定分组是寻址到本地网络还是应该被转发。此时，已经检测到分组中的错误，并且已经处理完分组的所有选项。ip 指向分组内的 IP 首部。

如果被配置成多播路由器，就转发分组

214-245 如果目的地址不是一个IP多播组，则跳过整个这部分代码。如果地址是一个多播组，并且系统被配置成IP多播路由器(ip_mrouter)，就把ip_id转换成网络字节序(ip_mforward希望的格式)，并把分组传给ip_mforward。如果出现错误或者分组是通过一个多播隧道(multicast tunnel)到达的，则ip_mforward返回一个非零值。分组被丢弃，且ips_cantforward的值加1。

我们在第14章中描述了多播隧道。它们在两个被标准IP路由器隔开的多播路由器之间传递分组。通过隧道到达的分组必须由ip_mforward处理，而不是由ipintr处理。

如果ip_mforward返回0，则把ip_id转换回主机字节序，由ipintr继续处理分组。

如果ip指向一个IGMP分组，则接受该分组，并在ours处(图10-11的ipintr)继续执行。不管到达接口的每个目的组或组成员是什么，多播路由器必须接受所有IGMP分组。IGMP分组中有组成员变化的信息。

246-257 根据系统是否被配置成多播路由器来确定是否执行图12-39中的其余程序。IN_LOOKUP_MULTI搜索接口加入的多播组表。如果没有找到匹配，则丢弃该分组。当硬件过滤器接受不需要的分组时，或者当与接口相关的多播组与分组中的目的多播地址映射到同一个以太网地址时，才出现这种情况。

如果接受了该分组，就继续执行ipintr(图10-11)的ours标号处的语句。

ip_input.c

```

214     if (IN_MULTICAST(ntohl(ip->ip_dst.s_addr))) {
215         struct in_multi *inm;
216         extern struct socket *ip_mrouter;

217         if (ip_mrouter) {
218             /*
219              * If we are acting as a multicast router, all
220              * incoming multicast packets are passed to the
221              * kernel-level multicast forwarding function.
222              * The packet is returned (relatively) intact; if
223              * ip_mforward() returns a non-zero value, the packet
224              * must be discarded, else it may be accepted below.
225              *
226              * (The IP ident field is put in the same byte order
227              * as expected when ip_mforward() is called from
228              * ip_output().)
229              */
230             ip->ip_id = htons(ip->ip_id);
231             if (ip_mforward(m, m->m_pkthdr.rcvif) != 0) {
232                 ipstat.ips_cantforward++;
233                 m_freem(m);
234                 goto next;
235             }
236             ip->ip_id = ntohs(ip->ip_id);

237             /*
238              * The process-level routing demon needs to receive
239              * all multicast IGMP packets, whether or not this
240              * host belongs to their destination groups.
241              */

```

图12-39 ipintr 函数：多播输入处理

```

242         if (ip->ip_p == IPPROTO_IGMP)
243             goto ours;
244         ipstat.ips_forward++;
245     }
246     /*
247     * See if we belong to the destination multicast group on the
248     * arrival interface.
249     */
250     IN_LOOKUP_MULTI(ip->ip_dst, m->m_pkthdr.rcvif, inm);
251     if (inm == NULL) {
252         ipstat.ips_cantforward++;
253         m_freem(m);
254         goto next;
255     }
256     goto ours;
257 }

```

ip_input.c

图12-39 (续)

12.15 多播输出处理：ip_output函数

当我们在第8章讨论ip_output时，推迟了对ip_output的mp参数和多播处理程序的讨论。在ip_output中，如果mp指向一个ip_moptions结构，它就覆盖多播输出处理的默认值。ip_output中省略的程序在图12-40和图12-41中显示。ip指向输出的分组，m指向包含该分组的mbuf，ifp指向路由表为目的多播组选择的接口。

```

129     if (IN_MULTICAST(ntohl(ip->ip_dst.s_addr))) {
130         struct in_multi *inm;
131         extern struct ifnet loif;
132
133         m->m_flags |= M_MCAST;
134         /*
135          * IP destination address is multicast. Make sure "dst"
136          * still points to the address in "ro". (It may have been
137          * changed to point to a gateway address, above.)
138          */
139         dst = (struct sockaddr_in *) &ro->ro_dst;
140         /*
141          * See if the caller provided any multicast options
142          */
143         if (imo != NULL) {
144             ip->ip_ttl = imo->imo_multicast_ttl;
145             if (imo->imo_multicast_ifp != NULL)
146                 ifp = imo->imo_multicast_ifp;
147         } else
148             ip->ip_ttl = IP_DEFAULT_MULTICAST_TTL;
149         /*
150          * Confirm that the outgoing interface supports multicast.
151          */
152         if ((ifp->if_flags & IFF_MULTICAST) == 0) {
153             ipstat.ips_noroute++;
154             error = ENETUNREACH;
155             goto bad;
156         }
157     }

```

ip_output.c

图12-40 ip_output 函数：默认和源地址

```

157      * If source address not specified yet, use address
158      * of outgoing interface.
159      */
160      if (ip->ip_src.s_addr == INADDR_ANY) {
161          struct in_ifaddr *ia;

162          for (ia = in_ifaddr; ia; ia = ia->ia_next)
163              if (ia->ia_ifp == ifp) {
164                  ip->ip_src = IA_SIN(ia)->sin_addr;
165                  break;
166              }
167      }

```

ip_output.c

图12-40 (续)

```

168      IN_LOOKUP_MULTI(ip->ip_dst, ifp, inm);
169      if (inm != NULL &&
170          (imo == NULL || imo->imo_multicast_loop)) {
171          /*
172           * If we belong to the destination multicast group
173           * on the outgoing interface, and the caller did not
174           * forbid loopback, loop back a copy.
175           */
176          ip_mloopback(ifp, m, dst);
177      } else {
178          /*
179           * If we are acting as a multicast router, perform
180           * multicast forwarding as if the packet had just
181           * arrived on the interface to which we are about
182           * to send. The multicast forwarding function
183           * recursively calls this function, using the
184           * IP_FORWARDING flag to prevent infinite recursion.
185           *
186           * Multicasts that are looped back by ip_mloopback(),
187           * above, will be forwarded by the ip_input() routine,
188           * if necessary.
189           */
190          extern struct socket *ip_mrouter;
191          if (ip_mrouter && (flags & IP_FORWARDING) == 0) {
192              if (ip_mforward(m, ifp) != 0) {
193                  m_freem(m);
194                  goto done;
195              }
196          }
197      }
198      /*
199       * Multicasts with a time-to-live of zero may be looped-
200       * back, above, but must not be transmitted on a network.
201       * Also, multicasts addressed to the loopback interface
202       * are not sent -- the above call to ip_mloopback() will
203       * loop back a copy if this host actually belongs to the
204       * destination group on the loopback interface.
205       */
206      if (ip->ip_ttl == 0 || ifp == &loif) {
207          m_freem(m);
208          goto done;
209      }
210      goto sendit;
211  }

```

ip_output.c

图12-41 ip_output 函数：环回、转发和发送

1. 建立默认值

129-155 只有分组是到一个多播组时，才执行图 12-40中的程序。此时，`ip_output`把`mbuf`中的`M_MCAST`置位，并把`dst`重设成最终目的地址，因为`ip_output`可能曾把它设成下一跳路由器(图8-24)。

如果传递了一个`ip_moptions`结构，则相应地改变`ip_ttl`和`ifp`。否则，把`ip_ttl`设成`1(IP_DEFAULT_MULTICAST_TTL)`，避免多播分组到达某个远程网络。查询路由表或`ip_moptions`结构所得到的接口必须支持多播。如果不支持，则`ip_output`丢弃该分组，并返回`ENETUNREACH`。

2. 选择源地址

156-167 如果没有指定源地址，则由`for`循环找到与输出接口相关的单播地址，并填入IP首部的`ip_src`。

与单播分组不同，如果系统被配置成一个多播路由器，则必须在一个以上的接口上发送输出的多播分组。即使系统不是一个多播路由器，输出的接口也可能是目的多播组的一个成员，也会需要接收该分组。最后，我们需要考虑一下多播环回策略和环回接口本身。把所有这些都考虑进去，共有三个问题：

- 是否要在输出的接口上接收该分组？
- 是否向其他接口转发该分组？
- 是否在出去的接口发送该分组？

图12-41显示了`ip_output`中解决这三个问题的程序。

3. 是否环回？

168-176 如果`IN_LOOKUP_MULTI`确定输出的接口是目的多播组的成员，而且`imo_multicast_loop`非零，则分组被`ip_mloopback`放到输出接口上排队，等待输入。在这种情况下，不考虑转发原始分组，因为在输入过程中如果需要，分组的复制会被转发的。

4. 是否转发？

178-197 如果分组不是环回的，但系统被配置成一个多播路由器，并且分组符合转发的条件，则`ip_mforward`向其他多播接口分发该分组的备份。如果`ip_mforward`没有返回0，则`ip_output`丢弃该分组，不发送它。这表明分组中有错误。

为了避免`ip_mforward`和`ip_output`之间的无限循环，`ip_mforward`在调用`ip_output`之前，总是把`IP_FORWARDING`打开。在本系统上产生的数据报是符合转发条件的，因为运输层不打开`IF_FORWARDING`。

5. 是否发送？

198-209 TTL是0的分组可能被环回，但从转发它们(`ip_mforward`丢弃它们)，也从不被发送。如果TTL是0或者如果输出接口是环回接口，则`ip_output`丢弃该分组，因为TTL超时，或者分组已经被`ip_mloopback`环回了。

6. 发送分组

210-211 到这个时候，分组应该已经从物理上在输出接口上被发送了。`sendit(ip_output, 图8-25)`处的程序在把分组传给接口的`if_output`函数之前可能已经把它分片了。我们将在21.10节中看到，以太网输出函数`ether_output`调用`arpresolve`，

arpresolve又调用ETHER_MAP_MULTICAST, 由ETHER_MAP_MULTICAST根据IP多播目的地址构造一个以太网多播目的地址。

ip_mloopback函数

ip_mloopback依靠looutput(图5-27)完成它的工作。ip_mloopback传递的looutput不是指向环回接口的指针, 而是指向输出多播接口的指针。图 12-42显示了ip_mloopback函数。

```

935 static void
936 ip_mloopback(ifp, m, dst)
937 struct ifnet *ifp;
938 struct mbuf *m;
939 struct sockaddr_in *dst;
940 {
941     struct ip *ip;
942     struct mbuf *copym;

943     copym = m_copy(m, 0, M_COPYALL);
944     if (copym != NULL) {
945         /*
946          * We don't bother to fragment if the IP length is greater
947          * than the interface's MTU. Can this possibly matter?
948          */
949         ip = mtod(copym, struct ip *);
950         ip->ip_len = htons((u_short) ip->ip_len);
951         ip->ip_off = htons((u_short) ip->ip_off);
952         ip->ip_sum = 0;
953         ip->ip_sum = in_cksum(copym, ip->ip_hl << 2);
954         (void) looutput(ifp, copym, (struct sockaddr *) dst, NULL);
955     }
956 }

```

ip_output.c

图12-42 ip_mloopback 函数

复制并把分组放到队列中

929-956 仅仅复制分组是不够的; 必须看起来分组已经被输出接口接收了, 所以ip_mloopback把ip_len和ip_off转换成网络字节序, 并计算分组的检验和。looutput把分组放到IP输入队列。

12.16 性能的考虑

Net/3的多播实现有几个潜在的性能瓶颈。因为许多以太网网卡并不能完美地实现对多播地址的过滤, 所以操作系统必须能够丢弃那些通过硬件过滤器的分组。在最坏的情况下, 以太网网卡可能会接收所有分组, 而其中大部分可能会被 ipintr发现不具有合法的IP多播组地址。

IP用简单的线性表和线性搜索过滤到达的IP数据报。如果表增长到一定长度后, 某些高速缓存技术, 如移动最近接收地址到表的最前面, 将有助于提高性能。

12.17 小结

本章我们讨论了一个主机如何处理IP多播数据报。我们看到, 在IP的D类地址和以太网多

播地址的格式及它们之间的映射关系。

我们讨论了 `in_multi` 和 `ether_multi` 结构，每个 IP 多播接口都维护一个它自己的组成员表，而每个以太网接口都维护一个以太网多播地址。

在输入处理中，只有到达接口是目的多播组的成员时，该 IP 多播才被接受下来。尽管如果系统被配置成多播路由器，它们也可能被继续转发到其他接口。

被配置成多播路由器的系统必须接受所有接口上的所有多播分组。只要为 `INADDR_ANY` 地址发布 `SIOCADDMULTI` 命令，就可以迅速做到这一点。

`ip_moptions` 结构是多播输出处理的基础。它控制对输出接口的选择、多播数据报 TTL 辖域值的设置以及环回策略。它也控制对 `in_multi` 结构的引用计数，从而决定接口加入或离开某个 IP 多播组的时机。

我们也讨论了多播 TTL 值实现的两个概念：分组生存期和分组辖域。

习题

- 12.1 发送 IP 广播分组到 255.255.255.255 和发送 IP 多播给所有主机组 224.0.0.1 的区别是什么？
- 12.2 为什么用多播代码中的 IP 单播地址标识接口？如果接口能发送和接收多播地址，但没有一个单播 IP 地址，必须做什么改动？
- 12.3 在 12.3 节中，我们讲到 32 个 IP 组地址被映射到同一个以太网地址上。因为 32 bit 地址中的 9 bit 不在映射中。为什么我们不说 $512(2^9)$ 个 IP 组被映射到一个以太网地址上？
- 12.4 你认为为什么把 `IP_MAX_MEMBERSHIPS` 设成 20？能被设得更大一些吗？提示：考虑 `ip_moptions` 结构(图 12-15)的大小。
- 12.5 当一个多播数据报被 IP 环回并且被发送它的硬件接口接收（即一个非单工接口）时，会发生什么情况？
- 12.6 画一个有一个多接口主机的网络图，即使该主机没有被配置成多播路由器，其他接口也能接收到在某个接口上发送的多播分组。
- 12.7 通过 SLIP 和环回接口而不是以太网接口跟踪成员增加请求。
- 12.8 进程如何请求内核加入多于 `IP_MAX_MEMBERSHIPS` 个组？
- 12.9 计算环回分组的检验和是多余的。设计一个方法，避免计算环回分组的检验和。
- 12.10 接口在不重用以太网地址的情况下，最多可加入多少个 IP 多播组中？
- 12.11 细心的读者可能已经注意到 `in_delmulti` 在发布 `SIOCDELMULTI` 请求时，假定接口已经定义了 `ioctl` 函数。为什么这样不会出错？
- 12.12 如果请求一个未识别的选项，则 `ip_getmoptions` 中分配的 `mbuf` 将会发生什么情况？
- 12.13 为什么把组成员机制与用于接收单播和广播数据报的绑定机制分离开来？