第22章　协议控制块

22.1　引言

协议层使用协议控制块(PCB)存放各UDP和TCP插口所要求的多个信息片。Internet协议维护Internet 协议控制块(Internet protocol control block)和TCP控制块(TCP control block)。因为UDP是无连接的，所以一个端结点需要的所有信息都可以在 Internet PCB中找到；不存在UDP控制块。

Internet PCB含有所有UDP和TCP端结点共有的信息：外部和本地 IP地址、外部和本地端号、IP首部原型、该端结点使用的IP选项以及一个指向该端结点目的地址选路表入口的指针。TCP控制块包含了TCP为各连接维护的所有结点信息：两个方向的序号、窗口大小、重传次数等等。

本章我们描述Net/3所用的Internet PCB，在详细讨论TCP时再探讨TCP控制块。我们将研究几个操作Internet PCB的函数，会在描述UDP和TCP时遇到它们。大多数的函数以 `in_pcb` 开头。

图22-1总结了协议控制块以及它们与 `file`和`socket`结构之间的关系。该图中有几点要考虑：

- 当`socket`或`accept`创建一个插口后，插口层生成一个 `file`结构和一个`socket`结构。文件类型是`DTYPE_SOCKET`，UDP端结点的插口类型是 `SOCK_DGRAM`，TCP端结点的插口类型是`SOCK_STREAM`。
- 然后调用协议层。UDP创建一个Internet PCB(一个`inpcb`结构)，并把它链接到`socket`结构上：`so_pcb`成员指向`inpcb`结构，`inp_socket`成员指向`socket`结构。
- TCP做同样的工作，也创建它自己的控制块 (一个`tcpcb`结构)，并用指针`inp_ppcb`和`t_inpcb`把它链接到`inpcb`上。在两个UDP `inpcb`中，`inp_ppcb`成员是一个空指针，因为UDP不负责维护它自己的控制块。
- 我们显示的其他四个`inpcb`结构的成员，从`inp_faddr`到`inp_lport`，形成了该端结点的插口对：外部IP地址和端口号，以及本地IP地址和端口号。
- UDP和TCP用指针`inp_next`和`inp_prev`维护一个所有Internet PCB的双向链表。它们在表头分配一个全局 `inpcb`结构(命名为`udb`和`tcb`)，在该结构中只使用三个成员：下一个和前一个指针，以及本地端口号。后一个成员中包含了该协议使用的下一个临时端口号。

Internet PCB是一个传输层数据结构。TCP、UDP和原始IP使用它，但IP、ICMP或ICMP不用它。

我们还没有讲过原始IP，但它也用Internet PCB。与TCP和UDP不同，原始IP在PCB中不用端口号成员，原始 IP只用本章中提到的两个函数： `in_pcballoc`分配PCB，`in_pcbdetach`释放PCB。第32章将讨论原始IP。

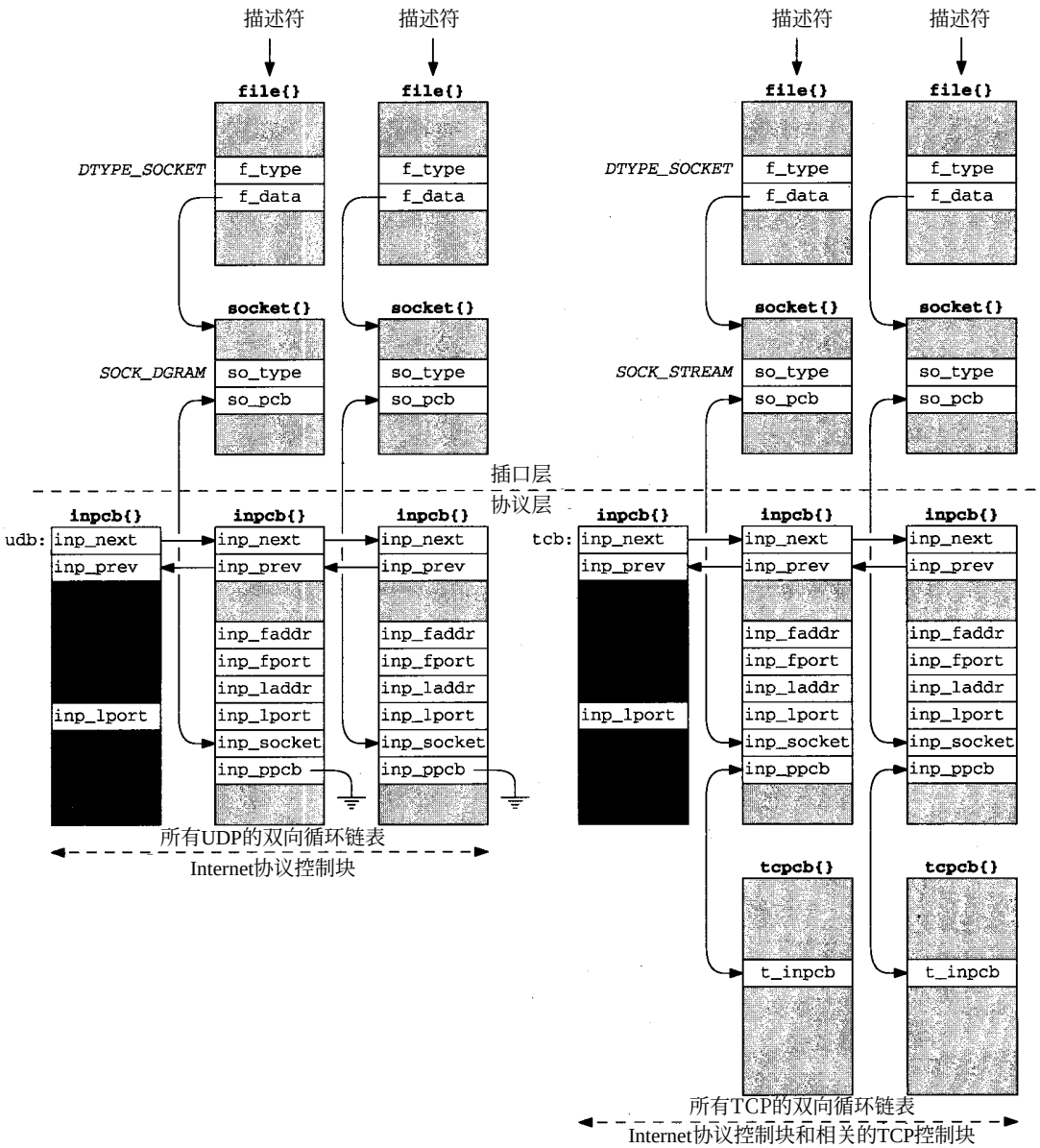

图22-1 Internet协议控制块以及与其他结构之间的关系

22.2 代码介绍

所有PCB函数都在一个C文件和一个包含定义的头文件中，如图 22-2所示。

文 件	描 述
netinet/in_pcb.h	in_pcb结构定义
netinet/in_pcb.c	PCB函数

图22-2 本章中讨论的文件

22.2.1 全局变量

本章只引入一个全局变量，如图22-3所示。

变 量	数据类型	描 述
zeroin_addr	struct in_addr	32 bit全零IP地址

图22-3 本章中引入的全局变量

22.2.2 统计量

Internet PCB和TCP PCB都是内核的malloc函数分配的M_PCB类型。这只是内核分配的大约60种不同类型内存的一种。例如，mbuf的类型是M_BUF，socket结构分配的类型是M_SOCKET。

因为内核保持所分配的不同类型内存缓存的计数器，所以维护着几个PCB数量的统计量。vmstat -m命令显示内核的内存分配统计信息，netstat -m命令显示的是mbuf分配统计信息。

22.3 inpcb的结构

图22-4是inpcb结构的定义。这不是一个大结构，只占84个字节。

in_pcb.h

```
42 struct inpcb {
43     struct inpcb *inp_next, *inp_prev;  /* doubly linked list */
44     struct inpcb *inp_head;      /* pointer back to chain of inpcb's for
45                                     this protocol */
46     struct in_addr inp_faddr;    /* foreign IP address */
47     u_short inp_fport;           /* foreign port# */
48     struct in_addr inp_laddr;    /* local IP address */
49     u_short inp_lport;           /* local port# */
50     struct socket *inp_socket;   /* back pointer to socket */
51     caddr_t inp_ppcb;            /* pointer to per-protocol PCB */
52     struct route inp_route;      /* placeholder for routing entry */
53     int     inp_flags;           /* generic IP/datagram flags */
54     struct ip inp_ip;            /* header prototype; should have more */
55     struct mbuf *inp_options;    /* IP options */
56     struct ip_moptions *inp_moptions;   /* IP multicast options */
57 };
```

in_pcb.h

图22-4 inpcb 结构

43-45 inp_next和inp_prev为UDP和TCP的所有PCB形成一个双向链表。另外，每个PCB都有一个指向协议链表表头的指针(inp_head)。对UDP表上的PCB，inp_head总是指向udb(图22-1)；对TCP表上的PCB，这个指针总是指向tcb。

46-49 下面四个成员：inp_faddr、inp_fport、inp_laddr和inp_lport，包含了这个IP端结点的插口对：外部IP地址和端口号，以及本地IP地址和端口号。PCB中以网络字节序而不是以主机字节序维护这四个值。

运输层的TCP和UDP都使用Internet PCB。尽管在这个结构里保存本地和外部IP地址很有意义，但端口号并不属于这里。端口号及其大小的定义是由各运输层协议

指定的，不同的运输层可以指定不同的值。[Partridge 1987] 提出了这个问题，其中版本1的RDP采用8 bit的端口号，需要用 8 bit的端口号重新实现几个标准内核程序。版本2的RDP [Partridge和Hinden 1990] 采用16 bit端口号。实际上，端口号属于运输层专用控制块，例如 TCP的tcpcb。可能会要求采用一种新的 UDP专用的PCB。尽管这个方案可行，但却可能使我们马上要讨论的几个程序复杂化。

50-51 inp_socket是一个指向该PCB的socket结构的指针，inp_ppcb是一个指针，它指向这个PCB的可选运输层专用控制块。我们在图 22-1中看到，inp_ppcb和TCP一起指向对应的tcpcb，但UDP不用它。socket和inpcb之间的链接是双向的，因为有时内核从插口层开始，需要对应的 Internet PCB(如用户输出)，而有时内核从 PCB开始，需要找到对应的socket结构(如处理收到的IP数据报)。

52 如果IP有一个到外部地址的路由，则它被保存在 ipp_route入口处。我们将看到，当收到一个ICMP重定向报文时，将扫描所有 Internet PCB，找到那些外部IP地址与重定向IP地址匹配的PCB，将其inp_route入口标记成无效。当再次将该 PCB用于输出时，迫使IP重新找一条到该外部地址的新路由。

53 inp_flags成员中存放了几个标志。图22-5显示了各标志。

inp_flags	描 述
INP_HDRINCL	进程提供整个IP首部(只有原始插口)
INP_RECVOPTS	把到达IP选项作为控制信息接收(只有UDP，还没有实现)
INP_RECVRETOPTS	把回答的IP选项作为控制信息接收(只有UDP，还没有实现)
INP_RECVDSTADDR	把IP目的地址作为控制信息接收(只有UDP)
INP_CONTROLOPTS	*INP_RECVOPTS|INP-RECVRETOPTS|INP_RECVDSTADDR*

图22-5 inp_flags 值

54 PCB中维护一个IP首部的备份，但它只使用其中的两个成员，TOS和TTL。TOS被初始化为0(普通业务)，TTL被运输层初始化。我们将看到，TCP和UDP都把TTL的默认值设为 64。进程可以用IP_TOS或IP_TTL插口选项改变这些默认值，新的值记录在 inpcb-inp_ip结构中。以后，TCP和UDP在发送IP数据报时，却把该结构用作原型 IP首部。

55-56 进程也可以用IP_OPTIONS插口选项设置外出数据报的 IP选项。函数ip_pcbopts把调用方选项的备份存放在一个 mbuf中，inp_options成员是一个指向该mbuf的指针。每次TCP和UDP调用ip_output函数时，就把一个指向这些 IP首部的指针传给IP，IP将其插到出去的IP数据报中。类似地，inp_moptions成员是一个指向用户IP多播选项备份的指针。

22.4 in_pcballoc和in_pcbdetach函数

在创建插口时，TCP、UDP和原始IP会分配一个 Internet PCB。系统调用 socket发布PRU_ATTACH请求。在UDP情况下，我们将在图23-33中看到，产生的调用是

```
struct socket *so;
int error;

error = in_pcballoc(so, &udb);
```

图22-6是in_pcballoc函数。

1. 分配PCB，初始化为零

in_pcb.c

```
36 int
37 in_pcballoc(so, head)
38 struct socket *so;
39 struct inpcb *head;
40 {
41     struct inpcb *inp;

42     MALLOC(inp, struct inpcb *, sizeof(*inp), M_PCB, M_WAITOK);
43     if (inp == NULL)
44         return (ENOBUFS);
45     bzero((caddr_t) inp, sizeof(*inp));

46     inp->inp_head = head;
47     inp->inp_socket = so;
48     insque(inp, head);
49     so->so_pcb = (caddr_t) inp;
50     return (0);
51 }
```

in_pcb.c

图22-6 in_pcballoc 函数：分配一个Internet PCB

36-45 in_pcballoc使用宏MALLOC调用内核的内存分配器。因为这些PCB总是作为系统调用的结果分配的，所以总能等到一个。

Net/2和早期的伯克利版本把Internet PCB和TCP PCB都保存在mbuf中。它们的大小分别是80和108字节。Net/3版本中的大小变成了84和140字节，所以TCP控制块不再适合存放在mbuf中。Net/3使用内核的内存分配器而不是mbuf分配两种控制块。

细心的读者会注意到图2-6的例子中，为PCB分配了17个mbuf，而我们刚刚讲到Net/3不再用mbuf存放Internet PCB和TCP PCB。但是，Net/3的确用mbuf存放Unix域的PCB，这就是计数器所指的。netstat输出的mbuf统计信息是针对内核为所有协议族分配的mbuf，而不仅仅是Internet协议族。

bzero把PCB设成0。这非常重要，因为PCB 中的IP地址和端口号必须被初始化成0。

in_pcb.c

```
252 int
253 in_pcbdetach(inp)
254 struct inpcb *inp;
255 {
256     struct socket *so = inp->inp_socket;

257     so->so_pcb = 0;
258     sofree(so);
259     if (inp->inp_options)
260         (void) m_free(inp->inp_options);
261     if (inp->inp_route.ro_rt)
262         rtfree(inp->inp_route.ro_rt);
263     ip_freemoptions(inp->inp_moptions);
264     remque(inp);
265     FREE(inp, M_PCB);
266 }
```

in_pcb.c

图22-7 in_pcbdetach 函数：释放一个Internet PCB

2. 把结构链接起来

46-49 in_head成员指向协议的PCB表头(udb或tcp)，inp_socket成员指向socket结

构，新的PCB结构被加到协议的双向链表上(insque)，socket结构指向该PCB。insque函数把新的PCB放到协议表的表头里。

在发布PRU_DETACH请求后，释放一个Internet PCB，这是在关闭插口时发生的。图22-7显示了in_pcbdetach函数，最后将调用它。

252-263 socket结构中的PCB指针被设成0，sofree释放该结构。如果给这个PCB分配的是一个有IP选项的mbuf，则由m_free将其释放。如果该PCB 中有一个路由，则由rtfree将其释放。所有多播选项都由ip_freemoptions释放。

264-265 remque把该PCB从协议的双向链表中移走，该PCB使用的内存被返回给内核。

22.5 绑定、连接和分用

在研究绑定插口、连接插口和分用进入的数据报的内核函数之前，我们先来看一下内核对这些动作施加的限制规则。

1. 绑定本地IP地址和端口号

图22-8是进程在调用bind时可以指定的本地IP地址和本地端口号的六种组合。

前三行通常是服务器的——它们绑定某个特定端口，称为服务器的知名端口 (well-known port)，客户都知道这些端口的值。后三行通常是客户的——它们不考虑本地的端口，称为临时端口(ephemeral port)，只要它在客户主机上是唯一的。

大多数服务器和客户在调用 bind时，都指定通配IP地址。如图22-8所示，在第3行和第6行中，用*表示。

本地IP地址	本地端口	描　述
单播或广播	非零	一个本地接口，特定端口
多播	非零	一个本地多播组，特定端口
*	非零	任何本地接口或多播组，特定端口
单播或广播	0	一个本地接口，内核选择端口
多播	0	一个多播组，内核选择端口
*	0	任何本地接口，内核选择端口

图22-8 bind 的本地IP地址和本地端口号的组合

如果服务器把某个特定IP地址绑定到某个插口上(也就是说，不是通配地址)，那么进入的IP数据报中，只有那些以该特定IP地址作为目的IP地址的IP数据报——不管是单播、广播或多播——都被交付给该进程。自然地，当进程把某个特定单播或广播 IP地址绑定到某个插口上时，内核验证该IP地址与一个本地接口对应。

尽管可能，但很少出现客户程序绑定某个特定 IP地址的情况(图22-8中的第4行和第5行)。通常客户绑定通配IP地址(图22-8中的最后一行)，让内核根据自己选择的到服务器的路由来选择外出的接口。

图22-8没有显示如果客户程序试图绑定一个已经被其他插口使用的本地端口时会发生什么情况。默认情况下，如果一个端口已经被使用，进程是不能绑定它的。如果发生这种情况，则返回EADDRINUSE差错(地址正在被使用)。正在被使用(in use)的定义很简单，就是只要存在一个PCB，就把该端口作为它的本地端口。“正在被使用”的概念是相对于绑定协议的：TCP或UDP，因为TCP端口号与UDP端口号无关。

Net/3允许进程指定以下两个插口选项来改变这个默认行为：

SO_REUSEADDR 允许进程绑定一个正在被使用的端口号，但被绑定的IP地址(包括通配地址)必须没有被绑定到同一个端口。

例如，如果连到的接口的IP地址是140.252.1.29，则一个插口可以被绑定到140.252.1.29，端口5555；另一个插口可绑定到127.0.0.1，端口5555；还有一个插口可以绑定到通配IP地址，端口5555。在第二种和第三种情况下调用bind之前，必须先调用setsockopt，设置SO_REUSEADDR选项。

SO_REUSEPORT 允许进程重用IP地址和端口号，但是包括第一个在内的各个IP地址和端口号，必须指定这个插口选项。和SO_REUSEADDR一样，第一次绑定端口号时要指定插口选项。

例如，如果连到的接口具有140.252.1.29的IP地址，并且某个插口绑定到140.252.1.29，端口6666，并指定SO_REUSEPORT插口选项，则另一个插口也可以指定同一个插口选项，并绑定140.252.1.29，端口6666。

本节的后面将讨论在后一个例子中，当到达一个目的地址是140.252.1.29，目的端口是6666的IP数据报时，会发生什么情况。因为这两个插口都被绑定到该端结点上。

SO_REUSEPORT是Net/3新加上的，在4.4BSD中是为支持多播而引入的。在这个版本之前，两个插口是不可能绑定到同一个IP地址和同一个端口号的。

不幸的是，SO_REUSEPORT不是原来的标准多播源程序的内容，所以对它的支持并不广泛。其他支持多播的系统，如Solaris 2.x，让进程指定SO_REUSEADDR来表明允许把多个端口绑定到同一IP地址和相同的端口号。

2. 连接一个UDP插口

我们通常把connect系统调用和TCP客户联系起来，但是UDP客户或UDP服务器也可能调用connect，为插口指定外部IP地址和外部端口号。这就限制插口必须只与某个特定对方交换UDP数据报。

当连接UDP插口时，会有一个副作用：本地IP地址，如果在调用bind时没有指定，会自动被connect设置。它被设成由IP选路指定对方所选择的本地接口地址。

图22-9显示了UDP插口的三种不同的状态，以及函数为终止各状态调用的伪代码。

本地插口	外部插口	描　述
localIP.lport	*foreignIP.fport*	限制到一个对方： socket(), bind(*.*lport*), connect(*foreignIP, fport*) socket(), bind(*localIP, lport*), connect(*foreignIP, fport*)
localIP.lport	*.*	限制在本地接口上到达的数据报：*localIP* socket(), bind(*localIP, lport*)
*.*lport*	*.*	接收所有发到*lport*的数据报： socket(),bind(*, *lport*)

图22-9 UDP插口的本地和外部IP地址和端口号规范

前三个状态叫做已连接的UDP插口(connected UDP socket)，后两个叫做未连接的UDP插口(unconnected UDP socket)。两个没有连接上的UDP插口的区别在于，第一个具有一个完全

指定的本地地址，而第二个具有一个通配本地 IP地址。

3. 分用TCP接收的IP数据报

图22-10显示了主机sun上的三个Telnet服务器的状态。前两个插口处于 LISTEN状态，等待进入的连接请求，加三个连接到 IP地址是140 252 111的主机上的端口1500。第一个监听插口处理在接口140.252.129上到达的连接请求，第二个监听插口将处理所有其他接口 (因为它的本地IP地址是通配地址)。

本地地址	本地端口	外部地址	外部端口	TCP状态
140.252.1.29	23	*	*	LISTEN
*	23	*	*	LISTEN
140.252.1.29	23	140.252.1.11	1500	ESTABLISHED

图22-10 本地端口是23的三个TCP插口

两个具有未指定的外部 IP地址和端口号的监听插口都显示了出来，因为插口 API不允许TCP服务器限制任何一个值。 TCP服务器必须accept客户的连接，并在连接建立完成之后(也就是说，当TCP的三次握手结束之后)被告知客户的IP地址和端口号。只有到这个时候，如果服务器不喜欢客户的 IP地址和端口号，才能关闭连接。这并不是对 TCP要求的特性，这只是插口API通常的工作方式。

当TCP收到一个目的端口是 23的报文段时，它调用 in_pcblookup，搜索它的整个Internet PCB表，找到一个匹配。马上我们会研究这个函数，将看到它有优先权，因为它的通配匹配(wildcard match)数最少。为了确定通配匹配数，我们只考虑本地和外部的 IP地址，不考虑外部端口号。本地端口号必须匹配，否则我们甚至不考虑 PCB。通配匹配数可以是 0、1(本地IP地址或外部IP地址)或2(本地和外部IP地址)。

例如，假定到达报文段来自 140.252.1.11，端口1500，目的地是140.252.1.29，端口23。图22-11是图22-10中三个插口的通配匹配数。

本地地址	本地端口	外部地址	外部端口	TCP状态	通配匹配数
140.252.1.29	23	*	*	LISTEN	1
*	23	*	*	LISTEN	2
140.252.1.29	23	140.252.1.11	1500	ESTABLISHED	0

图22-11 从{140.252.1.11, 1500}到{140.252.1.29, 23}的到达报文段

第一个插口匹配这四个值，但有一个通配匹配 (外部IP地址)。第二个插口也和到达报文段匹配，但有两个通配匹配 (本地和外部IP地址)。第三个插口是一个没有通配匹配的完全匹配。Net/3使用第三个插口，它具有最小通配匹配数。

继续这个例子，假定到达报文段来自 140.252.1.11，端口1501，目的地是140.252.1.29，端口23。图22-12显示了通配匹配数。

本地地址	本地端口	外部地址	外部端口	TCP状态	通配匹配数
140.252.1.29	23	*	*	LISTEN	1
*	23	*	*	LISTEN	2
140.252.1.29	23	140.252.1.11	1500	ESTABLISHED	

图22-12 从{140.252.1.11, 1501}到{140.252.1.29, 23}的到达报文段

第一个插口匹配有一个通配匹配；第二个插口匹配有两个通配匹配；第三个插口根本不匹配，因为外部端口号不相等 (只有当PCB中的外部IP地址不是通配地址时，才比较外部端口号)。所以选择第一个插口。

在这两个例子中，我们没有提到到达 TCP报文段的类型：假定图 22-11中的报文段包含数据或对一个已经建立的连接的确认，因为它是发送到一个已经建立的插口上的。我们还假定，图22-12中的报文段是一个到达的连接请求 (一个SYN)，因为它是发送给一个正在监听的插口的。但是in_pcblookup的分用代码并不关心这些。如果 TCP报文段对交付的插口来说是错误的类型，我们将在后面看到， TCP会处理这种情况。现在，重要的是，分用代码只把 IP数据报中的源和目的插口对的值与 PCB中的值进行比较。

4. 分用UDP接收的IP数据报

UDP数据报的交付比我们刚才研究的 TCP的例子要复杂得多，因为可以把 UDP数据报发送到一个广播或多播地址。因为 Net/3(以及大多数支持多播的系统)允许多个插口有相同的本地IP地址和端口，所以如何处理多个接收方的情况呢？ Net/3的规则是：

1) 把目的地是广播 IP地址或多播IP地址的到达UDP数据报交付给所有匹配的插口。这里没有“ 最好的 ”匹配的概念 (也就是具有最小通配匹配数的匹配)。

2) 把目的地是单播 IP地址的到达UDP数据报只交付给一个匹配的插口，就是具有最小通配匹配数的插口。如果有多个插口具有相同的“ 最小 ”通配匹配数，那么具体由哪个插口来接收到达数据报依赖于不同的实现。

图22-13显示了四个我们将在后面例子中使用的 UDP插口。要使四个UDP插口具有相同的本地端口号需要使用SO_REUSEADDR或SO_REUSEPORT。前两个插口已经被连接到一个外部IP地址和端口号，后面两个没有任何连接。

本地地址	本地端口	外部地址	外部端口	说　明
140.252.1.29	577	140.252.1.11	1500	已连接，本地IP = 单播
140.252.13.63	577	140.252.13.35	1500	已连接，本地IP = 广播
140.252.13.63	577	*	*	未连接，本地IP = 广播
*	577	*	*	未连接，本地IP = 通配地址

图22-13　四个本地端口为577的UDP插口

考虑目的地是 140.252.13.63(位于子网 140.252.13上的广播地址)，端口 577，来自140.252.13.34，端口1500。图22-14显示它被交付给第三和第四个插口。

本地地址	本地端口	外部地址	外部端口	交　付？
140.252.1.29	577	140.252.1.11	1500	不，本地和外部IP不匹配
140.252.13.63	577	140.252.13.35	1500	不，外部IP不匹配
140.252.13.63	577	*	*	交付
*	577	*	*	交付

图22-14　接收从{140.252.13.34, 1500}到{140.252.13.63, 577}的数据报

广播数据报不交付给第一个插口，因为本地 IP地址和目的IP地址不匹配，外部IP地址和源IP地址也不匹配。也不把它交付给第二个插口，因为外部 IP地址和源IP地址不匹配。

对于下一个例子，考虑目的地是140.252.129(一个单播地址)，端口577，来自140 252 1.111，

端口1500。图22-15显示了把该数据报交付给哪个端口。

本地地址	本地端口	外部地址	外部端口	交 付？
140.252.1.29	577	140.252.1.11	1500	交付，0个通配匹配
140.252.13.63	577	140.252.13.35	1500	不，本地和外部IP不匹配
140.252.13.63	577	*	*	不，本地IP不匹配
*	577	*	*	不，2个通配匹配

图22-15 接收从{140.252.1.11, 1500}到{140.252.1.29, 577}的数据报

该数据报和第一个插口匹配，且没有通配匹配；也和第四个插口匹配，但有两个通配匹配。所以，它被交付给第一个插口，最好的匹配。

22.6 in_pcblookup函数

in_pcblookup函数有几个作用：

1) 当TCP或UDP收到一个IP数据报时，in_pcblookup扫描协议的Internet PCB表，寻找一个匹配的PCB，来接收该数据报。这是运输层对收到数据报的分用。

2) 当进程执行bind系统调用，为某个插口分配一个本地IP地址和本地端口号时，协议调用in_pcbbind，验证请求的本地地址对没有被使用。

3) 当进程执行bind系统调用，请求给它的插口分配一个临时端口时，内核选了一个临时端口，并调用in_pcbbind检查该端口是否正在被使用。如果正在被使用，就试下一个端口号，以此类推，直到找到一个没有被使用的端口号。

4) 当进程显式或隐式地执行connect系统调用时，in_pcbbind验证请求的插口对是唯一的(当在一个没有连接上的插口上发送一个UDP数据报时，会隐式地调用connect，我们将在第23章看到这种情况)。

在第2种、第3种和第4种情况下，in_pcbbind调用in_pcblookup。两个选项使该函数的逻辑显得有些混乱。首先，进程可以指定SO_REUSEADDR或SO_REUSEPORT插口选项，表明允许复制本地地址。

其次，有时通配匹配也是允许的(例如，一个到达UDP数据报可以和一个自己的本地IP地址有通配符的PCB匹配，意味着该插口将接收在任何本地接口上到达的UDP数据报)，而其他情况下，一个通配匹配是禁止的(例如，当连接到一个外部IP地址和端口号时)。

> 在原始的标准IP多播代码中，出现了这样的注释“ in_pcblookup的逻辑比较模糊，也没有一点说明…… ”。形容词模糊比较保守。
>
> 公开的IP多播码是BSD／386的，是由Craig Leres从4.4BSD派生而来的。他修改了该函数过载的语义，只对上面的第1种情况使用in_pcblookup。第2种和第4种情况由一个新函数in_pcbconflict处理。情况3由新函数in_uniqueport处理。把原来的功能分成几个独立的函数就显得更清楚了，但在我们描述的Net/3版本中，整个逻辑还是结合在一个函数in_pcblookup中。

图22-16显示了in_pcblookuup函数。

该函数从协议的PCB表的表头开始，并可能会遍历表中的每个PCB。变量match记录了到目前为止最佳匹配的指针入口，matchwild记录在该匹配中的通配匹配数。后者被初始化

成3，比可能遇到的最大通配匹配数还大(任何大于2的值都可以)。每次循环时，wildcard从0开始，计数每个PCB的通配匹配数。

1. 比较本地端口号

第一个比较的是本地端口号。如果PCB的本地端口和lport参数不匹配，则忽略该PCB。

in_pcb.c

```
405 struct inpcb *
406 in_pcblookup(head, faddr, fport_arg, laddr, lport_arg, flags)
407 struct inpcb *head;
408 struct in_addr faddr, laddr;
409 u_int   fport_arg, lport_arg;
410 int     flags;
411 {
412     struct inpcb *inp, *match = 0;
413     int     matchwild = 3, wildcard;
414     u_short fport = fport_arg, lport = lport_arg;

415     for (inp = head->inp_next; inp != head; inp = inp->inp_next) {
416         if (inp->inp_lport != lport)
417             continue;           /* ignore if local ports are unequal */

418         wildcard = 0;

419         if (inp->inp_laddr.s_addr != INADDR_ANY) {
420             if (laddr.s_addr == INADDR_ANY)
421                 wildcard++;
422             else if (inp->inp_laddr.s_addr != laddr.s_addr)
423                 continue;
424         } else {
425             if (laddr.s_addr != INADDR_ANY)
426                 wildcard++;
427         }

428         if (inp->inp_faddr.s_addr != INADDR_ANY) {
429             if (faddr.s_addr == INADDR_ANY)
430                 wildcard++;
431             else if (inp->inp_faddr.s_addr != faddr.s_addr ||
432                      inp->inp_fport != fport)
433                 continue;
434         } else {
435             if (faddr.s_addr != INADDR_ANY)
436                 wildcard++;
437         }

438         if (wildcard && (flags & INPLOOKUP_WILDCARD) == 0)
439             continue;           /* wildcard match not allowed */

440         if (wildcard < matchwild) {
441             match = inp;
442             matchwild = wildcard;
443             if (matchwild == 0)
444                 break;          /* exact match, all done */
445         }
446     }
447     return (match);
448 }
```

in_pcb.c

图22-16 in_pcblookuup 函数：搜索所有PCB寻找匹配

2. 比较本地地址

419—427 in_pcblookup比较PCB内的本地地址和laddr参数。如果有一个是通配地址，另一个不是，则wildcard计数器加1。如果都不是通配地址，则它们必须一样；否则忽略这个PCB 。如果都是通配地址，则什么也不改变：它们不可比，也不增加 wildcard计数器。图22-17对四种不同的情况做了小结。

PCB本地IP	laddr参数	描 述
不是*	*	wildcard++
不是*	不是*	比较IP地址，如果不相等，则略过PCB
*	*	不能比较
*	不是*	wildcard++

图22-17 in_pcblookup 做的四种IP地址比较

3. 比较外部地址和外部端口号

428—437 这几行完成与我们刚才讲的同样的检查，但是用外部地址而不是本地地址。而且，如果两个外部地址都不是通配地址，则不仅两个 IP地址必须相等，而且两个外部端口也必须相等。图22-18对外部IP地址的比较作了总结。

PCB外部IP	faddr参数	描 述
不是*	*	wildcard++
不是*	不是*	比较IP地址和端口，如果不相等，则略过PCB
*	*	不能比较
*	不是*	wildcard++

图22-18 in_pcblookup 做的四种外部IP地址比较

可以对图22-18中的第二行进行另外的外部端口号比较，因为一个 PCB不可能具有非通配外部地址，且外部端口号为 0。这个限制是由connect加上的，我们马上就会看到，该函数要求一个非通配外部 IP地址和一个非零外部端口。但是，也可能，并且通常都是具有一个通配本地地址和一个非零本地端口。我们在图 22-10和图22-13看到过这种情况。

4. 检查是否允许通配匹配

438—439 参数flags可以被设成INPLOOKUP_WILDCARD，意味着允许匹配中包含通配匹配。如果在匹配中有通配匹配(wildcard非零)，并且调用方没有指定这个标志位，则忽略这个PCB。当TCP和UDP调用这个函数分用一个到达数据报时，总是把 INPLOOKUP_WILDCARD置位，因为允许通配匹配(记住我们用图22-10和图22-13所作的例子)。但是，当这个函数作为connect系统调用的一部分而调用时，为了验证一个插口对没有被使用，把flags参数设成0。

5. 记录最佳匹配，如果找到确切匹配，则返回

440-447 这些语句记录到目前为止找到的最佳匹配。重复一下，最佳匹配是具有最小通配匹配数的匹配。如果一个匹配有一个或两个通配匹配，则记录该匹配，循环继续。但是，如果找到一个确切的匹配(wildcard是0)，则循环终止，返回一个指向该确切匹配 PCB的指针。

例子——分用收到的TCP报文段

图22-19取自我们在图 22-11中的TCP的例子。假定 in_pcblookup正在分用一个从140.252.1.11即端口1500到140.252.1.29即端口23的数据报。还假定PCB的顺序是图中行的顺序。

laddr是目的IP地址，lport是目的TCP端口，faddr是源IP地址，fport是源TCP端口。

PCB值				wildcard
本地地址	本地端口	外部地址	外部端口	
140.252.1.29	23	*	*	1
*	23	*	*	2
140.252.1.29	23	140.252.1.11	1500	0

图22-19 laddr =140.252.1.29, lport= 23, faddr = 140.252.1.11, fport= 1500

当把第一行和到达报文段比较时，wildcard是1(外部IP地址)，flags被设成INPLOOKUP_WILDCARD，所以把match设成指向该PCB，matchwild设为1。因为还没有找到确切的匹配，所以循环继续。下一次循环中，wildcard是2(本地和外部IP地址)，因为比matchwild大，所以不记录该入口，循环继续。再次循环时，wildcard是0，比matchwild(1)小，所以把这个入口记录在match中。因为已经找到了一个确切的地址，所以终止循环，把指向该PCB的指针返回给调用方。

如果TCP和UDP只用in_pcblookup来分用到达数据报，就可以对它进行简化。首先，没有必要检查faddr或laddr是否是通配地址，因为它们是收到数据报的源和目的IP地址。参数flags以及与相应的检测也可以不要，因为允许通配匹配。

这一节讨论了in_pcblookup函数的机制。我们在讨论in_pcbbind和in_pcbconnect如何调用这个函数后，将继续回来讨论它的意义。

22.7 in_pcbbind函数

下一个函数in_pcbbind，把一个本地地址和端口号绑定到一个插口上。从五个函数中调用它：

1) bind为某个TCP插口调用(通常绑定到服务器的一个知名端口上)；

2) bind为某个UDP插口调用(绑定到服务器的一个知名端口上，或者绑定到客户插口的一个临时端口上)；

3) connect为某个TCP插口调用，如果该插口还没有绑定到一个非零端口上(对TCP客户来说，这是一种典型情况)；

4) listen为某个TCP插口调用，如果该插口还没有绑定到一个非零端口(这很少见，因为是TCP服务器调用listen，TCP服务器通常绑定到一个知名端口上，而不是临时端口)；

5) 从in_pcbconnect(22.8节)调用，如果本地IP地址和本地端口号被置位(当为一个UDP插口调用connect，或为一个未连接UDP插口调用sendto时，这种情况比较典型)。

在第3种、第4种和第5种情形下，把一个临时端口号绑定到该插口上，不改变本地IP地址(在它已经被置位的情况下)。

称情形1和情形2为显式绑定(explicit bind)，情形3、4和5为隐式绑定(implicit bind)。我们也注意到，尽管在情形2时，服务器绑定到一个知名端口是很正常的，但那些用远程过程调用(RPC)启动的服务器也常常绑定到临时端口上，然后用其他程序注册它们的临时端口，该程序维护在该服务器的RPC程序号与其临时端口之间的映射(例如，卷1的29.4节描述的Sun端口映射器)。

我们分三部分显示in_pcbbind函数。图22-20是第一部分。

in_pcb.c

```
52 int
53 in_pcbbind(inp, nam)
54 struct inpcb *inp;
55 struct mbuf *nam;
56 {
57     struct socket *so = inp->inp_socket;
58     struct inpcb *head = inp->inp_head;
59     struct sockaddr_in *sin;
60     struct proc *p = curproc;    /* XXX */
61     u_short lport = 0;
62     int     wild = 0, reuseport = (so->so_options & SO_REUSEPORT);
63     int     error;

64     if (in_ifaddr == 0)
65         return (EADDRNOTAVAIL);
66     if (inp->inp_lport || inp->inp_laddr.s_addr != INADDR_ANY)
67         return (EINVAL);

68     if ((so->so_options & (SO_REUSEADDR | SO_REUSEPORT)) == 0 &&
69         ((so->so_proto->pr_flags & PR_CONNREQUIRED) == 0 ||
70          (so->so_options & SO_ACCEPTCONN) == 0))
71         wild = INPLOOKUP_WILDCARD;
```

in_pcb.c

图22-20 in_pcbbind 函数：绑定本地地址和端口号

64-67 前两个测试验证至少有一个接口已经被分配了一个IP地址，且该插口还没有绑定。不能两次绑定一个插口。

68-71 这个if语句有点令人疑问。总的结果是如果SO_REUSEADDR和SO_REUSEPORT都没有置位，就把wild设置成INPLOOKUP_WILDCARD。

对UDP来说，第二个测试为真，因为PR_CONNREQUIRED对无连接插口为假，对面向连接的插口为真。

第三个测试就是疑问所在[Torek 1992]。插口标志SO_ACCEPTCONN只被系统调用listen置位(15.9节)，该值只对面向连接的服务器有效。在正常情况下，一个TCP服务器调用socket、bind，然后调用listen。因而，当in_pcbbind被bind调用时，就清除了这个插口标志位。即使进程调用完socket后就调用listen，而不调用bind，TCP的PRU_LISTEN请求还是调用in_pcbbind，在插口层设置SO_ACCEPTCONN标志位之前，给插口分配一个临时端口。这意味着if语句中的第三个测试，测试SO_ACCEPTCONN是否没有置位，总是为真。因此if语句等价于

```
if ((so->so_options & (SO_REUSEADDR|SO_REUSEPORT)) == 0 &&
   ((so->so_proto->pr_flags & PR_CONNREQUIRED)==0||1)
        wild = INPLOOKUP_WILDCARD;
```

因为任何与1作逻辑或运算的结果都为真，所以这等价于

```
if ((so->so_options & (SO_REUSEADDR|SO_REUSEPORT)) == 0 )
         wild = INPLOOKUP_WILDCARD;
```

这样简单且容易理解：如果任何一个REUSE插口选项被置位，wild就是0。如果没有REUSE选项被置位，则把wild设成INPLOOKUP_WILDCARD。换言之，当函数在后面调用in_pcblookup时，只有在没有REUSE选项处于开状态时，才允许通配匹配。

in_pcbbind的下一部分，显示在图22-22中，函数处理可选nam参数。

72-75 只有当进程显式调用 bind时，nam参数才是一个非零指针。对一个隐式的绑定(connect、listen或in_pcbconnect的副作用，本节开始的情形3、4和5)，nam是一个空指针。当指定了该参数时，它是一个含有 sockaddr_in结构的mbuf。图22-21显示了非空参数nam的四种情形。

nam参数		PCB成员被设成：		
localIP	*lport*	inp_laddr	inp_iport	说　明
不是*	0	*localIP*	临时端口	*localIP*必须是本地接口
不是*	非零	*localIP*	*lport*	交付给in_pcblookup
*	0	*	临时端口	
*	非零	*	*lport*	交付给 in_pcblookup

图22-21 in_pcbbind 的nam 参数的四种情形

76-83 对正确的地址族的测试被注释掉了，但在函数 in_pcbconnect(图22-25)中执行了等价的测试。我们希望两者或者都有或者都没有。

85-94 Net/3测试被绑定的IP地址是否是一个多播组。如果是，则SO_REUSEADDR选项被认为与SO_REUSEPORT等价。

95-99 否则，如果调用方绑定的本地地址不是通配地址，则 ifa_ifwithaddr验证该地址与一个本地接口对应。

注释“ yech ”可能是因为插口地址结构中的端口号必须是 0，因为 ifa_ifwithaddr对整个结构作二进制比较，而不仅仅比较 IP地址。

这是进程在调用系统调用之前必须把插口地址结构全部置零的几种情况之一。如果调用 bind，并且插口地址结构 (sin_zero[8])的最后 8个字节非零，则 ifa_ifwithaddr将找不到请求的接口，in_pcbbind会返回一个错误。

100-105 当调用方绑定了一个非零端口时，也就是说，进程要绑定一个特殊端口号 (图22-21中的第2种和第4种情形)，就执行下一个if语句。如果请求的端口小于1024(IPPORT_RESERVED)，则进程必须具有超级用户的优先权限。这不是 Internet协议的一部分，而是伯克利的习惯。使用小于 1024的端口号，我们称之为保留端口 (reserved port)，例如，rcmd函数 [Stevens 1990]使用的端口，rlogin和rsh客户程序又调用该函数，作为服务器对它们身份认证的一部分。

106-109 然后调用函数in_pcblookup(图22-16)，检测是否已经存在一个具有相同本地 IP地址和本地端口号的 PCB。第二个参数是通配 IP地址(外部IP地址)，第三个参数是一个为 0的端口号(外部端口号)。第二个参数的通配值导致 in_pcblookup忽略该PCB的外部IP地址和外部端口 —— 只把本地IP 地址和本地端口号分别和 sin->sin_addr及lport进行比较。我们前面提到，只有当所有REUSE插口选项都没有被设置时，才把wild设成INPLOOKUP_WILDCARD。

111 调用方的本地IP地址值存放在PCB中。如果调用方指定，它可以是通配地址。在这种情况下，由内核选择本地 IP地址，但要等到晚些时候插口连接上时。这就是为什么说本地 IP地址是根据外部IP地址，由IP路由选择决定。

in_pcb.c

```
 72     if (nam) {
 73         sin = mtod(nam, struct sockaddr_in *);
 74         if (nam->m_len != sizeof(*sin))
 75             return (EINVAL);
 76 #ifdef notdef
 77         /*
 78          * We should check the family, but old programs
 79          * incorrectly fail to initialize it.
 80          */
 81         if (sin->sin_family != AF_INET)
 82             return (EAFNOSUPPORT);
 83 #endif
 84         lport = sin->sin_port;  /* might be 0 */
 85         if (IN_MULTICAST(ntohl(sin->sin_addr.s_addr))) {
 86             /*
 87              * Treat SO_REUSEADDR as SO_REUSEPORT for multicast;
 88              * allow complete duplication of binding if
 89              * SO_REUSEPORT is set, or if SO_REUSEADDR is set
 90              * and a multicast address is bound on both
 91              * new and duplicated sockets.
 92              */
 93             if (so->so_options & SO_REUSEADDR)
 94                 reuseport = SO_REUSEADDR | SO_REUSEPORT;
 95         } else if (sin->sin_addr.s_addr != INADDR_ANY) {
 96             sin->sin_port = 0;  /* yech... */
 97             if (ifa_ifwithaddr((struct sockaddr *) sin) == 0)
 98                 return (EADDRNOTAVAIL);
 99         }
100         if (lport) {
101             struct inpcb *t;
102             /* GROSS */
103             if (ntohs(lport) < IPPORT_RESERVED &&
104                 (error = suser(p->p_ucred, &p->p_acflag)))
105                 return (error);
106             t = in_pcblookup(head, zeroin_addr, 0,
107                              sin->sin_addr, lport, wild);
108             if (t && (reuseport & t->inp_socket->so_options) == 0)
109                 return (EADDRINUSE);
110         }
111         inp->inp_laddr = sin->sin_addr;      /* might be wildcard */
112     }
```

in_pcb.c

图22-22 in_pcbbind 函数：处理可选的nam参数

当调用方显式绑定端口0，或nam参数是一个空指针(隐式绑定)时，in_pcbbind的最后一部分处理分配一个临时端口。

113-122 这个协议(TCP或UDP)使用的下一个临时端口号被维护在该协议的PCB表的head：tcb或udb。除了协议的head PCB中的inp_next和inp_back指针外，inpcb结构另一个唯一被使用的元素是本地端口号。令人迷惑的是，这个本地端口在head PCB中是主机字节序，而在表中其他PCB上，却是网络字节序！使用从1024开始的临时端口号(IPPORT_RESERVED)，每次加1，直到5000(IPPORT_USERRESERVED)，然后又从1024重新开始循环。该循环一直执行到in_pcbbind找不到匹配为止。

1. SO_REUSEADDR举例

让我们通过一些普通的例子，来了解一下in_pcbbind与in_pcblookup及两个REUSE

插口选项之间的交互。

in_pcb.c

```
113     if (lport == 0)
114         do {
115             if (head->inp_lport++ < IPPORT_RESERVED ||
116                 head->inp_lport > IPPORT_USERRESERVED)
117                 head->inp_lport = IPPORT_RESERVED;
118             lport = htons(head->inp_lport);
119         } while (in_pcblookup(head,
120                               zeroin_addr, 0, inp->inp_laddr, lport, wild));
121     inp->inp_lport = lport;
122     return (0);
123 }
```

in_pcb.c

图22-23 in_pcbbind 函数：选择一个临时端口

1) TCP或UDP通常以调用socket和bind开始。假定一个调用bind的TCP服务器，指定了通配IP 地址和它的非零知名端口23(Telnet服务器)。还假定该服务器还没有运行，进程没有设置SO_REUSEADDR插口选项。
in_pcbbind把INPLOOKUP_WILDCARD作为最后一个参数，调用in_pcblookup。in_pcblookup中的循环没有找到匹配的PCB，就假定没有其他进程使用服务器的知名TCP端口，返回一个空指针。一切正常，in_pcbbind，返回0。
2) 假定和上面相同的情况，但当再次试图启动服务器时，该服务器已经开始运行。
当调用in_pcblookup时，它发现了本地插口为{*, 23}的PCB。因为wildcard计数器是0，所以in_pcblookup返回指向这个入口的指针。因为reuseport是0，所以in_pcbbind返回EADDRINUSE。
3) 假定与上面相同的情况，但当第二次试图启动服务器时，指定了SO_REUSEADDR插口选项。
因为指定了这个插口选项，所以in_pcbbind在调用in_pcblookup时，最后一个参数为0。但本地插口为{*, 23}的PCB仍然匹配，因为in_pcblookup无法比较两个通配地址(图22-17)，所以wildcard为0。in_pcbbind又返回EADDRINUSE，避免启动两个具有相同本地插口的服务器例程，不管是否指定了SO_REUSEADDR。
4) 假定有一个Telnet服务器已经以本地插口{*, 23}开始运行，而我们试图以另一个本地插口{140.252.13.35，23}启动另一个服务器。
假定没有指定SO_REUSEADDR，调用in_pcblookup时，最后一个参数为INPLOOKUP_WILDCARD。当它与含有*.23的PCB比较时，wildcard计数器被设为1。因为允许通配匹配，所以在扫描完所有 TCP PCB后，就把这个匹配作为最佳匹配。in_pcbbind返回EADDRINUSE。
5) 这个例子与上一个相同，但为第二个试图绑定本地插口{140.252.13.35, 23}的服务器指定了SO_REUSEADDR插口选项。
现在，in_pcblookup的最后一个参数是0，因为指定了插口选项。当与本地插口为{*, 23}的PCB比较时，wildcard计数器为1，但因为最后的flags参数是0，所以跳过这个入口，不把它记作匹配。在比较完所有 TCP PCB后，函数返回一个空指针，in_pcbbind返回0。

6) 假定当我们试图以本地插口{*, 23}启动第二个服务器时，第一个Telnet服务器以本地插口{140.252.13.35, 23}启动。与前面的例子一样，但这一次我们以相反的顺序启动服务器。第一个服务器的启动没有问题，假定没有其他插口绑定到端口 23。当我们启动第二个服务器时，in_pcblookup的最后一个参数是INPLOOKUP_WILDCARD，假定没有指定SO_REUSEADDR插口选项。当和具有本地插口 {140.252.13.35, 23}的PCB比较时，wildcard被设成1，记录这个入口。在比较完所有 TCP PCB后，返回指向这个入口的指针。导致in_pcbbind返回EADDRINUSE。

7) 如果我们启动同一个服务器的两个例程，并且都是非通配本地 IP地址，会发生什么情况？假定我们以本地插口 {140.252.13.35, 23}启动第一个Telnet服务器，然后试图用本地插口{127.0.0.1, 23}启动第二个服务器，且不指定SO_REUSEADDR。

当第二个服务器调用 in_pcbbind时，它调用 in_pcblookup，最后一个参数是INPLOOKUP_WILDCARD。当比较具有本地插口{140.252.13.35, 23}的PCB时，因为本地IP地址不相等，所以跳过它。in_pcblookup返回一个空指针，in_pcbbind返回0。

从这个例子中我们看到，SO_REUSEADDR插口选项对非通配IP地址没有影响。事实上，只有当wildcard大于0 时，也就是说，当PCB入口具有一个通配IP地址，或者绑定的IP地址是一个通配地址时，才检查in_pcblookup中的INPLOOKUP_WILDCARD标志位。

8) 作为最后一个例子，假定我们试图启动同一服务器的两个例程，具有相同的非通配本地IP 地址127.0.0.1。

启动第二个服务器时，in_pcblookup总是返回具有相同本地插口的匹配PCB。不管是否指定SO_REUSEADDR插口选项，都发生这种情况，因为对这种比较，wildcard计数器总是0。因为in_pcblookup返回一个非空指针，所以in_pcbbind返回EADDRINUSE。

从这些例子中，我们可以指出本地 IP地址和SO_REUSEADDR插口选项的绑定规则。这些规则如图22-24所示。假定*localIP*1和*localP*2是在本地主机上有效的两个不同的单播或广播 IP地址，*localmcastIP*是一个多播组。我们还假定进程要绑定到一个已经绑定到某个已存在 PCB的非零端口号。

我们需要区分单播或多播地址和一个多播地址，因为我们看到，in_pcbbind认为对多播地址，SO_REUSEADDR与SO_REUSEPORT是一样。

存在PCB	试图绑定	SO_REUSEADDR		描 述
		关	开	
*localIP*1	*localIP*1	错误	错误	每个IP地址和端口一个服务器
*localIP*1	*localIP*2	正确	正确	每个本地接口一个服务器
*localIP*1	*	错误	正确	一个接口一个服务器，其他接口一个服务器
*	*localIP*1	错误	正确	一个接口一个服务器，其他接口一个服务器
*	*	错误	错误	不能复制本地插口(和第一个例子一样)
*localIP*1	*localIP*1	错误	正确	多个多播接收方

图22-24 SO_REUSEADDR 插口选项对绑定本地IP地址的影响

2. SO_REUSEPORT插口选项

Net/3中对SO_REUSEPORT的处理改变了in_pcbbind的逻辑，只要指定了SO_REUSEPORT，就允许复制本地插口。换言之，所有服务器都必须同意共享同一本地端口。

22.8 in_pcbconnect函数

函数in_pcbconnect为插口指定IP地址和外部端口号。有四个函数调用它：

1) connect为某个TCP插口(某个TCP客户的请求)调用；

2) connect为某个UDP插口(对UDP客户是可选的，UDP服务器很少见)调用；

3) 当在一个没有连接上的UDP插口(普通)上输出数据报时从sendto调用；

4) 当一个连接请求(一个SYN报文段)到达一个处于LISTEN状态(对TCP服务器是标准的)的TCP插口时，tcp_input调用。

在以上四种情况下，当调用in_pcbconnect时，通常，但不要求，不指定本地IP地址和本地端口。因此，在没有指定的情形下，由in_pcbconnect的一个函数给它们赋一个本地的值。

我们将分四个部分讨论in_pcbconnect函数。图22-25显示了第一部分。

in_pcb.c

```
130 int
131 in_pcbconnect(inp, nam)
132 struct inpcb *inp;
133 struct mbuf *nam;
134 {
135     struct in_ifaddr *ia;
136     struct sockaddr_in *ifaddr;
137     struct sockaddr_in *sin = mtod(nam, struct sockaddr_in *);
138     if (nam->m_len != sizeof(*sin))
139         return (EINVAL);
140     if (sin->sin_family != AF_INET)
141         return (EAFNOSUPPORT);
142     if (sin->sin_port == 0)
143         return (EADDRNOTAVAIL);
144     if (in_ifaddr) {
145         /*
146          * If the destination address is INADDR_ANY,
147          * use the primary local address.
148          * If the supplied address is INADDR_BROADCAST,
149          * and the primary interface supports broadcast,
150          * choose the broadcast address for that interface.
151          */
152 #define satosin(sa)     ((struct sockaddr_in *)(sa))
153 #define sintosa(sin)    ((struct sockaddr *)(sin))
154 #define ifatoia(ifa)    ((struct in_ifaddr *)(ifa))
155         if (sin->sin_addr.s_addr == INADDR_ANY)
156             sin->sin_addr = IA_SIN(in_ifaddr)->sin_addr;
157         else if (sin->sin_addr.s_addr == (u_long) INADDR_BROADCAST &&
158                  (in_ifaddr->ia_ifp->if_flags & IFF_BROADCAST))
159             sin->sin_addr = satosin(&in_ifaddr->ia_broadaddr)->sin_addr;
160     }
```

in_pcb.c

图22-25 in_pcbconnect 函数：验证参数，检查外部IP地址

1. 确认参数

130-143 nam参数指向一个包含sockaddr_in结构以及外部IP地址和端口号的mbuf。这些行确认参数并验证调用方不打算连接到端口号为0的端口上。

2. 特别处理到0.0.0.0和255.255.255.255的连接

134-160 对全局变量in_ifaddr的检查证实已配置了一个IP接口。如果外部地址是0.0.0.0(INADDR_ANY)，则用最初的IP接口的IP地址代替0.0.0.0。这就是说，调用进程是连接到这个主机上的一个对等实体的。如果外部IP地址是255.255.255.255 (INADDR_BROADCAST),

而且原来的接口支持广播，则用原来接口的广播地址代替 255.255.255.255。这样，UDP应用程序无需计算它的 IP 地址，就可以在原来的接口上广播 —— 它可以简单地把数据报发送给255.255.255.255，由内核把这个地址转换成该接口合适的 IP地址。

下一部分代码，如图 22-26所示，处理没有指定本地地址的情况。对 TCP和UDP客户程序来说，本节开始的表中的情形 1、2和3是非常普遍的。

in_pcb.c

```
161     if (inp->inp_laddr.s_addr == INADDR_ANY) {
162         struct route *ro;
163         ia = (struct in_ifaddr *) 0;
164         /*
165          * If route is known or can be allocated now,
166          * our src addr is taken from the i/f, else punt.
167          */
168         ro = &inp->inp_route;
169         if (ro->ro_rt &&
170             (satosin(&ro->ro_dst)->sin_addr.s_addr !=
171              sin->sin_addr.s_addr ||
172              inp->inp_socket->so_options & SO_DONTROUTE)) {
173             RTFREE(ro->ro_rt);
174             ro->ro_rt = (struct rtentry *) 0;
175         }
176         if ((inp->inp_socket->so_options & SO_DONTROUTE) == 0 &&     /* XXX */
177             (ro->ro_rt == (struct rtentry *) 0 ||
178              ro->ro_rt->rt_ifp == (struct ifnet *) 0)) {
179             /* No route yet, so try to acquire one */
180             ro->ro_dst.sa_family = AF_INET;
181             ro->ro_dst.sa_len = sizeof(struct sockaddr_in);
182             ((struct sockaddr_in *) &ro->ro_dst)->sin_addr =
183                 sin->sin_addr;
184             rtalloc(ro);
185         }
186         /*
187          * If we found a route, use the address
188          * corresponding to the outgoing interface
189          * unless it is the loopback (in case a route
190          * to our address on another net goes to loopback).
191          */
192         if (ro->ro_rt && !(ro->ro_rt->rt_ifp->if_flags & IFF_LOOPBACK))
193             ia = ifatoia(ro->ro_rt->rt_ifa);
194         if (ia == 0) {
195             u_short fport = sin->sin_port;
196             sin->sin_port = 0;
197             ia = ifatoia(ifa_ifwithdstaddr(sintosa(sin)));
198             if (ia == 0)
199                 ia = ifatoia(ifa_ifwithnet(sintosa(sin)));
200             sin->sin_port = fport;
201             if (ia == 0)
202                 ia = in_ifaddr;
203             if (ia == 0)
204                 return (EADDRNOTAVAIL);
205         }
```

in_pcb.c

图22-26 in_pcbconnect 函数：没有指定本地IP地址

3. 如果路由不再有效，则释放该路由

164-175 如果PCB中含有一条路由，但该路由的目的地址和已经连接上的外部地址不同，或者SO_DONTROUTE插口选项被置位，则放弃该路由。

为了理解为什么一个PCB会含有一条相关路由，考虑本节开始的表中的情形3：每次在一个未连接上的插口上发送UDP数据报时，就调用in_pcbconnect。每次进程调用sendto时，UDP输出函数调用in_pcbconnect、ip_output和in_pcbdisconnect。如果在该插口上发送的所有数据报都具有相同的目的IP地址，则第一次通过in_pcbconnect时，就分配了一条路由，从此时开始可以使用该路由。但是，因为UDP应用程序可能在每次调用sendto时，都向不同的IP地址发送数据报，所以必须比较目的地址和保存的路由。当目的地址改变时，就放弃该路由。ip_output也作同样的检查，这看起来似乎是多余的。

SO_DONTROUTE插口选项告诉内核旁路掉正常的选路决策，把该IP数据报发到本地连接的接口，该接口的IP网络地址和目的地址的网络部分匹配。

4. 获取路由

176-185 如果没有置位SO_DONTROUTE插口选项，则PCB中没有到目的地的路由，就要调用rtalloc获取一条路由。

5. 确定外出的接口

186-205 这一节代码的意图是让ia指向一个接口地址结构(in_ifaddr，6.5节)，该结构中包含了该接口的IP地址。如果PCB中的路由仍然有效，或者如果rtalloc找到一条路由，并且该路由不是到回环接口的，则使用相应的接口。否则，调用ifa_withdstaddr和ifa_withnet检查该外部IP地址是否在一个点到点链路的另一端，或者位于一个连到的网络上。两个函数都要求插口地址结构中的端口号为0，以便在调用期间保存在fport中。如果失败，就用原来的IP地址(in_ifaddr)，如果没有配置接口(in_ifaddr为0)，则返回错误。

图22-27显示了in_pcbconnect的下一部分，处理目的地址是多播地址的情况。

206-223 如果目的地址是一个多播地址，且进程指定了多播分组的外出接口(用IP_MULTICAST_IF插口选项)，则该接口的IP地址被用作本地地址。搜索所有IP接口，找到与插口选项所指定接口的匹配。如果该接口不存在，则返回错误。

224-225 图22-26的开头是处理通配本地地址情形的完整代码。指向本地接口ia的sockaddr_in结构的指针保存在ifaddr中。

in_pcblookup的最后一部分显示在图22-28中。

6. 验证插口对是唯一的

227-233 in_pcblookup验证插口对是唯一的。外部地址和外部端口号是指定给in_pcbconnect的参数的值。本地地址是已经绑定到该插口的值，或者是ifaddr中我们刚刚介绍的代码计算出来的值。本地端口可以是0，对TCP客户程序来说这是典型的。我们将在这部分代码的后面看到，为本地端口选择了一个临时端口。

这个测试避免从相同的本地地址和本地端口上建立两个到同一外部地址和外部端口的TCP连接。例如，如果我们与主机sun上的回显服务器建立了一个TCP连接，然后试图从同一本地端口(8888，用-b选项指定)建立另一条到同一服务器的连接，调用in_pcblookup后返回一个匹配，导致connect返回差错EADDRINUSE(我们用卷1附录C的sock程序)。

```
bsdi S sock -b 8888 sun echo &      启动后台的第一个
bsdi S sock -A  -b 8888 sun echo    然后再试一次
connect() error: Address already in use
```

in_pcb.c

```
206         /*
207          * If the destination address is multicast and an outgoing
208          * interface has been set as a multicast option, use the
209          * address of that interface as our source address.
210          */
211         if (IN_MULTICAST(ntohl(sin->sin_addr.s_addr)) &&
212             inp->inp_moptions != NULL) {
213             struct ip_moptions *imo;
214             struct ifnet *ifp;

215             imo = inp->inp_moptions;
216             if (imo->imo_multicast_ifp != NULL) {
217                 ifp = imo->imo_multicast_ifp;
218                 for (ia = in_ifaddr; ia; ia = ia->ia_next)
219                     if (ia->ia_ifp == ifp)
220                         break;
221                 if (ia == 0)
222                     return (EADDRNOTAVAIL);
223             }
224         }
225         ifaddr = (struct sockaddr_in *) &ia->ia_addr;
226     }
```

in_pcb.c

图22-27 in_pcbconnect 函数：目的地址是一个多播地址

in_pcb.c

```
227     if (in_pcblookup(inp->inp_head,
228                      sin->sin_addr,
229                      sin->sin_port,
230                  inp->inp_laddr.s_addr ? inp->inp_laddr : ifaddr->sin_addr,
231                      inp->inp_lport,
232                      0))
233         return (EADDRINUSE);

234     if (inp->inp_laddr.s_addr == INADDR_ANY) {
235         if (inp->inp_lport == 0)
236             (void) in_pcbbind(inp, (struct mbuf *) 0);
237         inp->inp_laddr = ifaddr->sin_addr;
238     }
239     inp->inp_faddr = sin->sin_addr;
240     inp->inp_fport = sin->sin_port;
241     return (0);
242 }
```

in_pcb.c

图22-28 in_pcbconnect 函数：验证插口对是唯一的

我们指定-A选项，设置SO_REUSEADDR插口选项，使bind成功，但是connect不成功。这是一个人为的例子，因为我们显式地把两个插口都绑定到同一本地端口上(8888)。在正常情形下，主机bsdi上的两个不同客户程序连接到sun的回显服务器上，当第二个客户程序调用图22-28中的in_pcblookup函数时，本地端口将是0。

这个测试也避免了两个UDP插口从相同的本地端口上连接到同一个外部地址。但这个测试不能避免两个UDP插口从同一个本地端口上交替地向同一个外部地址发送数据报，只要它们都不调用connect。因为UDP插口在sendto系统调用的过程中，只是临时连接到一个对等实体上。

7. 隐式绑定和分配临时端口

234-238 如果插口的本地地址仍然是通配匹配的，则把它设置成ifaddr中保存的值。这是一个隐式绑定：22.7节开始时讲的情形3、4和5。首先，检查本地端口是否已经被绑定，如果没有，in_pcbbind就把该插口绑定到一个临时端口。调用 in_pcbbind和给inp_laddr赋值的顺序很重要，因为如果本地地址不是通配地址，则 in_pcbbind会失败。

8. 把外部地址和外部端口存放在PCB中

239-240 这个函数的最后一步设置PCB的外部IP地址和外部端口号成员。如果这个函数成功返回，我们就能保证PCB中的插口对 —— 本地的和外部的 —— 都有了特定的值。

IP源地址与外出接口地址

在IP数据报的源地址和用来发送该数据报接口的IP地址之间有些微妙的差别。

TCP和UDP把PCB成员inp_laddr用作该IP数据报的源地址。它可由进程设成任何被bind配置的接口的IP 地址(在in_pcbbind中调用ifa_ifwithaddr验证应用程序想要的本地地址)。只有当本地地址是一个通配地址时，in_pcbconnect才给它赋值。而当这种情况发生时，本地地址是根据外出接口分配的(因为目的地址已知)。

但是，外出接口也是根据目的IP地址，由ip_output确定的。在多接口主机上，当进程显式绑定一个不同于外出接口的本地地址时，源地址有可能是一个本地接口的 IP地址，且该接口不是外出的接口。这种情况是允许的，因为Net/3选择了弱端系统模式(8.4节)。

22.9 in_pcbdisconnect函数

ip_pcbdisconnect把UDP插口断连。把外部IP地址设成全0(INADDR_ANY)，外部端口号设成0，就把外部相关内容删除了。

这是在已经在一个未连接上的UDP插口上发送了一个数据报后，在一个连接上的 UDP插口上调用 connect时做的。在第一种情况下，调用 sendto的次序是：UDP调用in_pcbconnect把插口临时连接到目的地，udp_output发送数据报，然后in_pcbdisconnect删除临时连接。

当关闭插口时，不调用in_pcbdisconnect，因为in_pcbdetach处理释放PCB。只有当一个不同的地址或端口号要求重用该PCB时，才断连。

图22-29显示了in_pcbdisconnect函数。

in_pcb.c

```
243 int
244 in_pcbdisconnect(inp)
245 struct inpcb *inp;
246 {

247     inp->inp_faddr.s_addr = INADDR_ANY;
248     inp->inp_fport = 0;
249     if (inp->inp_socket->so_state & SS_NOFDREF)
250         in_pcbdetach(inp);
251 }
```

in_pcb.c

图22-29 in_pcbdisconnect 函数：与外部地址和端口号断连

如果该PCB不再有文件表引用(SS_NOFDREF置位)，则in_pcbdetach(图22-7)释放该PCB。

22.10 in_setsockaddr和in_setpeeraddr函数

getsockname系统调用返回插口的本地协议地址 (例如，Internet插口的IP 地址和端口号)，getpeername系统调用返回外部协议地址。两个系统调用终止时，都发布一个PRU_SOCKADDR或PRU_PEERADDR请求。然后协议调用 in_setsockaddr或in_setpeeraddr。图22-30显示了以上的第一种情况。

in_pcb.c

```
267 int
268 in_setsockaddr(inp, nam)
269 struct inpcb *inp;
270 struct mbuf *nam;
271 {
272     struct sockaddr_in *sin;

273     nam->m_len = sizeof(*sin);
274     sin = mtod(nam, struct sockaddr_in *);
275     bzero((caddr_t) sin, sizeof(*sin));
276     sin->sin_family = AF_INET;
277     sin->sin_len = sizeof(*sin);
278     sin->sin_port = inp->inp_lport;
279     sin->sin_addr = inp->inp_laddr;
280 }
```

in_pcb.c

图22-30 in_setsockaddr 函数：返回本地地址和端口号

参数nam是一个指针，该指针指向一个用来存放结果的 mbuf：一个sockaddr_in结构，系统调用复制给进程的备份。该代码填写插口地址结构的内容，并把 IP 地址和端口号从Internet PCB拷贝到sin_addr和sin_port成员中。

图22-31显示了in_setpeeraddr函数。它基本上等同于图22-30中的代码，但从PCB中拷贝了外部IP地址和端口号。

in_pcb.c

```
281 int
282 in_setpeeraddr(inp, nam)
283 struct inpcb *inp;
284 struct mbuf *nam;
285 {
286     struct sockaddr_in *sin;

287     nam->m_len = sizeof(*sin);
288     sin = mtod(nam, struct sockaddr_in *);
289     bzero((caddr_t) sin, sizeof(*sin));
290     sin->sin_family = AF_INET;
291     sin->sin_len = sizeof(*sin);
292     sin->sin_port = inp->inp_fport;
293     sin->sin_addr = inp->inp_faddr;
294 }
```

in_pcb.c

图22-31 in_setpeeraddr 函数：返回外部地址和端口号

22.11 in_pcbnotify、in_rtchange和in_losing函数

当收到一个ICMP差错时，调用in_pcbnotify函数，把差错通知给合适的进程。通过对所有的PCB搜索一个协议(TCP或UDP)，并把本地和外部IP地址及端口号与ICMP差错返回的值进行比较，找到“合适的进程”。例如，当因为一些路由器丢掉了某个 TCP报文段而收到

ICMP源抑制差错时，TCP必须找到产生该差错的连接的PCB，放慢在该连接上的传输速度。

在显示该函数之前，我们必须回顾一下它是怎样被调用的。图 22-32总结了处理ICMP差错时调用的函数。两个有阴影的椭圆是本节描述的函数。

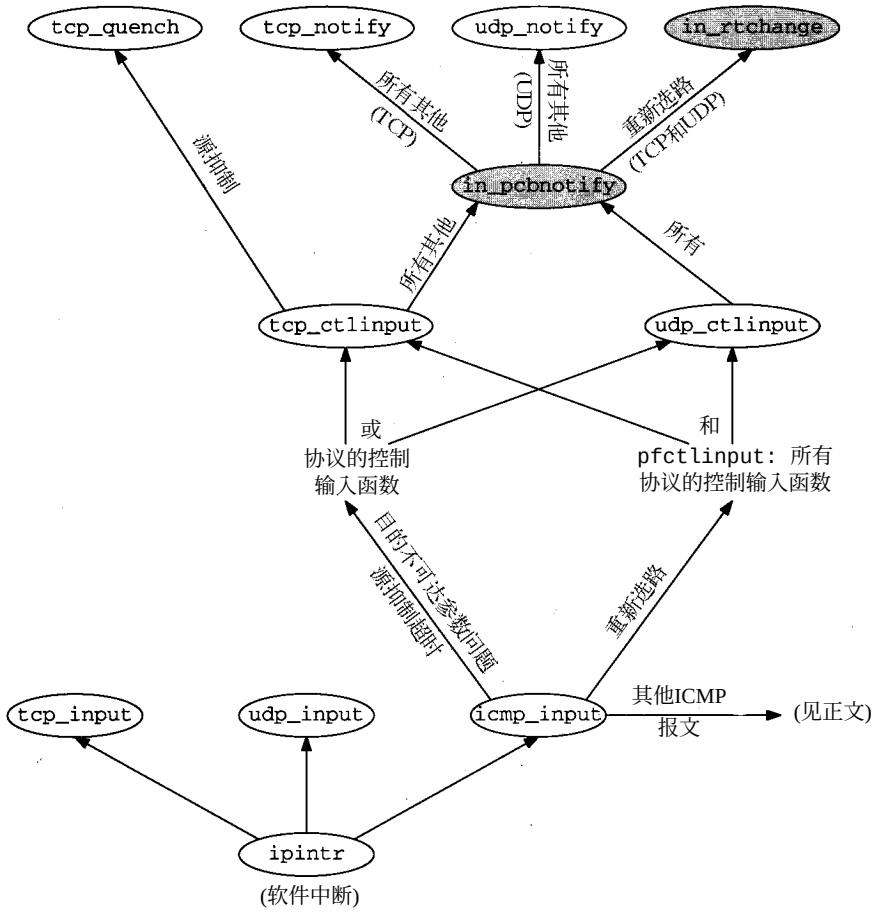

图22-32 ICMP差错处理总结

当收到一个ICMP报文时，调用`icmp_input`。ICMP的五种报文按差错来划分 (图11-1和图11-2)：

- 目的主机不可达；
- 参数问题；
- 重定向；
- 源抑制；
- 超时。

重定向的处理不同于其他四个差错。所有其他的 ICMP报文(查询)的处理见第11章。

每个协议都定义了它的控制输入函数，即`protosw`结构(7.4节)中的`pr_ctlinput`入口。对TCP和UDP，它们分别称为`tcp_ctlinput`和`udp_ctlinput`，我们将在后面几章给出它们的代码。因为收到的ICMP差错中包含了引起差错的数据报的 IP首部，所以引起该差错的协议(TCP或UDP)是已知的。这五个 ICMP差错中的四个将引起对协议的控制输入函数的调用。

重定向的处理不同：调用函数 pfctlinput，它继续调用协议族(Internet)中所有协议的控制输入函数。TCP和UDP是Internet协议族中仅有的两个具有控制输入函数的协议。

重定向的处理是特殊的，因为它们不仅影响产生重定向的数据报，还将影响所有到该目的地的IP数据报。另一方面，其他四个差错只需由产生差错的协议进行处理。

in_pcb.c

```
306 int
307 in_pcbnotify(head, dst, fport_arg, laddr, lport_arg, cmd, notify)
308 struct inpcb *head;
309 struct sockaddr *dst;
310 u_int   fport_arg, lport_arg;
311 struct in_addr laddr;
312 int     cmd;
313 void    (*notify) (struct inpcb *, int);
314 {
315     extern u_char inetctlerrmap[];
316     struct inpcb *inp, *oinp;
317     struct in_addr faddr;
318     u_short fport = fport_arg, lport = lport_arg;
319     int     errno;

320     if ((unsigned) cmd > PRC_NCMDS || dst->sa_family != AF_INET)
321         return;
322     faddr = ((struct sockaddr_in *) dst)->sin_addr;
323     if (faddr.s_addr == INADDR_ANY)
324         return;

325     /*
326      * Redirects go to all references to the destination,
327      * and use in_rtchange to invalidate the route cache.
328      * Dead host indications: notify all references to the destination.
329      * Otherwise, if we have knowledge of the local port and address,
330      * deliver only to that socket.
331      */
332     if (PRC_IS_REDIRECT(cmd) || cmd == PRC_HOSTDEAD) {
333         fport = 0;
334         lport = 0;
335         laddr.s_addr = 0;
336         if (cmd != PRC_HOSTDEAD)
337             notify = in_rtchange;
338     }
339     errno = inetctlerrmap[cmd];
340     for (inp = head->inp_next; inp != head;) {
341         if (inp->inp_faddr.s_addr != faddr.s_addr ||
342             inp->inp_socket == 0 ||
343             (lport && inp->inp_lport != lport) ||
344             (laddr.s_addr && inp->inp_laddr.s_addr != laddr.s_addr) ||
345             (fport && inp->inp_fport != fport)) {
346             inp = inp->inp_next;
347             continue;           /* skip this PCB */
348         }
349         oinp = inp;
350         inp = inp->inp_next;
351         if (notify)
352             (*notify) (oinp, errno);
353     }
354 }
```

in_pcb.c

图22-33 in_spcbnotify 函数：把差错通知传给进程

有关图22-32我们要做的最后一点说明是，TCP在处理源抑制差错时，与其他差错的处理不同，而重定向由in_pcbnotify特别处理：不管引起差错的是什么协议，都调用in_rtchange函数。

图22-33显示了in_pcbnotify函数。当TCP调用它时，第一个参数是tcb的地址，最后一个参数是函数tcp_notify的地址。对UDP来说，这两个参数分别是udb的地址和函数udp_notify的地址。

1. 验证参数

306-324 验证cmd参数和目的地址族。检测外部地址，保证它不是0.0.0.0。

2. 特殊处理重定向

325-338 如果差错是重定向，则对它的处理是特殊的(差错PRC_HOSTDEAD是一种旧的差错，由IMP产生。目前的系统再也看不到这种差错了——它是一个历史产物)。外部端口、本地端口和本地地址都被设成全0，这样后面的for循环就不会比较它们了。对于重定向，我们需要该循环只根据外部IP地址选出接收通知的PCB，因为主机是在这个IP地址上接收到重定向的。而且，为重定向调用的函数是in_rtchange(图22-34)，而不是调用方指定的notify参数。

339 全局数组inetctlerrmap把协议无关差错码(图11-19中的PRC_*xxx*值)映射到它对应的Unix的errno值(图11-1的最后一栏)。

3. 为所选的PCB调用通知函数

341-353 这个循环选择要通知的PCB。可以通知多个PCB——该循环在找到匹配后仍然继续。第一个if语句结合了五个检测，如果这五个中有任一个为真，则跳过该PCB：(1)如果外部地址不相等；(2)如果该PCB没有对应的socket结构；(3)如果本地端口不相等；(4)如果本地地址不相等；或(5)如果外部端口不相等。外部地址必须匹配，但只有当对应的参数非零时，才比较其他三个外部和本地参数。当找到一个匹配时，调用notify函数。

22.11.1 in_rtchange函数

我们看到，当ICMP差错是一个重定向时，in_pcbnotify调用in_rtchange函数。对所有外部地址与已重定向的IP地址匹配的PCB，都调用该函数。图22-34显示了in_rtchange函数。

in_pcb.c

```
391 void
392 in_rtchange(inp, errno)
393 struct inpcb *inp;
394 int     errno;
395 {
396     if (inp->inp_route.ro_rt) {
397         rtfree(inp->inp_route.ro_rt);
398         inp->inp_route.ro_rt = 0;
399         /*
400          * A new route can be allocated the next time
401          * output is attempted.
402          */
403     }
404 }
```

in_pcb.c

图22-34 in_rtchange函数：使路由无效

如果该PCB中有路由，则rtfree释放该路由，且该PCB 成员被标记为空。此时，我们不用重定向返回的路由来更新路由。当这个 PCB被再次使用时，ip_output会根据内核的选路表重新分配新的路由，而该选路表是在调用 pfctlinput之前，由重定向报文更新的。

22.11.2 重定向和原始插口

让我们来研究一下重定向、原始插口和缓存在 PCB中的路由之间的交互。如果我们运行Ping程序，该程序使用一个原始插口，收到来自被 ping的IP地址发来的ICMP重定向差错。Ping程序继续使用原来的路由，而不是已重定向的路由。我们可以从以下过程来看。

我们从位于1 402 521网络上的gemini主机 ping位于14 025 213网络上的svr4主机。gemini的默认路由是gateway，但分组应该被发送到路由器netb。图22-35显示了这个安排。

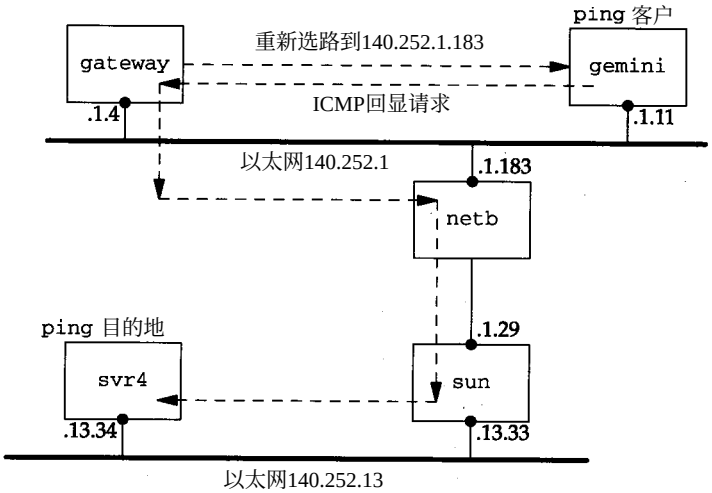

图22-35 ICMP重定向举例

我们希望gateway在收到第一个ICMP回显请求时，发一个重定向。

```
gemini $ ping -sv svr4
PING 140.252.13.34: 56 data bytes
ICMP Host redirect from gateway 140.252.1.4
  to netb (140.252.1.183) for svr4 (140.252.13.34)
64 bytes from svr4 (140.252.13.34): icmp_seq=0. time=572. ms
ICMP Host redirect from gateway 140.252.1.4
  to netb (140.252.1.183) for svr4 (140.252.13.34)
64 bytes from svr4 (140.252.13.34): icmp_seq=1. time=392. ms
```

选项-s使每隔一秒发送一次ICMP回显请求，选项-v打印每个收到的ICMP报文(不仅仅是ICMP回显回答)。

每个ICMP回显请求引出一个重定向，但 ping使用的原始插口从来不通知重定向改变它正在使用的路由。第一次计算出来并被保存在 PCB中的路由，使 IP数据报被发送到路由器gateway{140.252.1.4}，应该更新它，使数据报能被发送到路由器 netb{140.252.1.183}上。我们看到，gemini上的内核接收ICMP重定向，但它们被略过了。

如果我们终止ping程序，并重新运行它，我们就再也看不到重定向了：

```
gemini $ ping -sv svr4
```

```
PING 140.252.13.34: 56 data bytes
64 bytes from svr4 (140.252.13.34): icmp_seq=0,time=388. ms
64 bytes from svr4 (140.252.13.34): icmp_seq=1. time=363. ms
```

这个不正常的原因是原始IP插口代码(第32章)没有控制输入函数。只有TCP和UDP有控制输入函数。当收到重定向时，ICMP更新内核的选路表，调用pfctlinput(图22-32)。但是因为原始IP协议没有控制输入函数，所以不释放与Ping的原始插口相关的PCB中高速缓存的路由。但是，当我们第二次运行ping程序时，根据内核更新后的选路表分配路由，所以我们看不到重定向了。

22.11.3 ICMP差错和UDP插口

插口API令人迷惑的一部分是，不把在UDP插口上收到的ICMP差错传给应用程序，除非该应用程序在该插口上发布connect，限制该插口的外部IP地址和端口号。现在我们来看一下in_pcbnotify是如何实施这一限制的。

考虑某个ICMP插口不可达，这大概是UDP插口上最普通的一种ICMP差错了。in_pcbnotify的dst参数内的外部IP地址和外部端口号是引起ICMP差错的IP地址和端口号。但是，如果该进程已经在该插口上发布connect命令，则PCB的inp_faddr和inp_fport成员都是0，避免in_pcbnotify在该插口上调用notify函数。图22-33中的for循环将跳过每个UDP PCB。

产生这个限制的原因有两个。首先，如果正在发送的进程有一个未连接上的UDP插口，则该插口对中唯一的非零元素是本地端口(假定该进程不调用bind)。这是in_pcbnotify在分用进入的ICMP差错，并把它传给正确进程时，唯一可用的值。尽管很少发生，但也可能有多个进程都绑定到相同的本地端口上，所以具体由哪个进程接收ICMP差错就不明确了。还有一种可能就是，发送引起ICMP差错数据报的进程已经终止了，而另一个进程又开始运行并使用同一本地端口。这也不太可能，因为临时端口是从1024到5000按顺序分配的，只有循环一遍以后才可能重用同一端口号(图22-23)。

这个限制的第二个原因是，内核给进程的差错通知——一个errno值——是不够的。考虑某个进程连续三次在一个未连接上的UDP插口上调用sendto函数，向三个不同的目的地发送一个UDP数据报，然后用recvfrom等待回答。如果其中一个数据报生成一个ICMP端口不可达差错，且内核将向该进程发布的recvfrom返回对应的差错(ECONNREFUSED)，那么，errno值并没有告诉进程是哪个数据报产生了该差错。内核具有ICMP差错所要求的所有信息，但是插口API并不提供手段把这些信息返回给该进程。

因此，如果进程想要得到在某个UDP插口上的这些ICMP差错通知，在设计时就必须决定插口只能连接到一个对等实体上。如果在该连接上的插口返回ECONNREFUSED差错，毫无疑问就是该对等实体产生的差错。

还有一种远程可能性，会把ICMP差错交付给错误的进程。假设某个进程发送了一个UDP数据报，引起一个ICMP差错，但它在收到该差错之前终止了。另一个进程在收到该差错之前开始运行，并且绑定到同一个本地端口，连接到相同的外部地址和外部端口上，导致这个新进程接收到前面的ICMP差错。由于UDP缺少内存，所以无法避免这种情况的发生。我们将看到TCP用它的TIME_WAIT状态处理这个问题。

在我们前面的例子中，应用程序绕开这个限制的一个办法是使用三个连接上的UDP插口，

而不是一个未连接上的插口，并在其中任意一个有收到的数据报或差错要读写时，调用select函数来确定。

这里我们有一种情形是内核有足够的信息而API(插口)的信息不足。大多数Unix系统V及其他常见的API(TLI)，其逆为真：TLI函数t_rcvuderr可以返回对等实体的IP地址、端口号以及一个差错值。但大多数TCP/IP的SVR4流实现都不为ICMP提供手段，把差错传递给一个未连接上的UDP端节点。

在理想情况下，in_pcbnotify把ICMP差错交付给所有匹配的UDP插口，即使唯一的非通配匹配是本地端口。返回给进程的差错将包括产生差错的目的IP地址和目的UDP端口，允许进程确定该差错是否是它发送的数据报产生的。

22.11.4 in_losing函数

处理PCB的最后一个函数图22-36的in_losing。当TCP的某个连接的重传定时器连续第三次超时时，调用该函数。

in_pcb.c

```
361 int
362 in_losing(inp)
363 struct inpcb *inp;
364 {
365     struct rtentry *rt;
366     struct rt_addrinfo info;
367     if ((rt = inp->inp_route.ro_rt)) {
368         inp->inp_route.ro_rt = 0;
369         bzero((caddr_t) & info, sizeof(info));
370         info.rti_info[RTAX_DST] =
371             (struct sockaddr *) &inp->inp_route.ro_dst;
372         info.rti_info[RTAX_GATEWAY] = rt->rt_gateway;
373         info.rti_info[RTAX_NETMASK] = rt_mask(rt);
374         rt_missmsg(RTM_LOSING, &info, rt->rt_flags, 0);
375         if (rt->rt_flags & RTF_DYNAMIC)
376             (void) rtrequest(RTM_DELETE, rt_key(rt),
377                              rt->rt_gateway, rt_mask(rt), rt->rt_flags,
378                              (struct rtentry **) 0);
379         else
380             /*
381              * A new route can be allocated
382              * the next time output is attempted.
383              */
384             rtfree(rt);
385     }
386 }
```

in_pcb.c

图22-36 in_losing 函数：使高速缓存路由信息无效

1. 产生选路报文

361-374 如果PCB中有一个路由，则丢掉该路由。用要失效的高速缓存路由的有关信息填充一个rt_addrinfo结构。然后调用rt_missmsg函数，从RTM_LOSING类型的选路插口中生成一个报文，指明有关该路由的问题。

2. 删除或释放路由

375-384 如果高速缓存路由是由一个重定向生成的（RTF_DYNAMIC置位），则用请求RTM_DELETE调用rtrequest，删除该路由。否则释放高速缓存的路由，这样，当该插口上有下一个输出时，为它重新分配一条到目的地的路由——希望是一条更好的路由。

22.12 实现求精

毫无疑问，这一章我们遇到的最耗时的算法是in_pcblookup做的对PCB的线性搜索。22.6节一开始，我们就注意到有四种情况会调用这个函数。可以忽略对bind和connect的调用，因为TCP和UDP在分用每个收到的IP数据报时，调用它们的次数比调用in_pcblookup的少得多。

后面几章我们将看到，TCP和UDP试图帮助这个线性搜索，它们都维护一个指向该协议引用的最后一个PCB的指针：一个单入口高速缓存。如果高速缓存的PCB的本地地址、本地端口、外部地址和外部端口与收到的数据报的值匹配，则协议根本就不调用in_pcblookup。如果协议数据适合分组列模型[Jain和Routhier 1986]，这个简单的高速缓存效果很好。但是，如果数据不适合这个模型，例如，看起来象联机交易处理系统的数据入口，则单入口高速缓存的效率很低[McKenney和Dove 1992]。

一个稍好一点的PCB安排的建议是，当引用某个PCB时，把它移到该PCB表的最前面([McKenney和Dove 1992]把这个想法给了Jon Crowcroft；[Partridge和Pink 1993]把它给了Gary Delp)。移动PCB很容易，因为该表是一个双向链表，而且in_pcblookup的第一个参数是一个指向该表表头的指针。

[McKenney和Dove 1992]把原始的Net/1实现(没有高速缓存)，一种提高的单入口发送-接收高速缓存，“移到最前面”启发算法，以及他们自己的使用散列链的算法做了比较。他们指出，在散列链上维护一个PCB的线性表比其他算法的性能提高了一个数量级。散列链的唯一耗费是需要内存存放散列链的链头，以及计算散列函数。他们也考虑把“移到最前面”启发算法与他们的散列链算法结合，结论是只增加一些散列链，更为简单。

BSD线性搜索和散列表搜索的另一个比较是在[Hutchinson和Peterson 1991]中。他们指出，随着散列表中插口数量的增加，分用一个进入的UDP数据报所需要的时间是常量，但线性搜索所需要的时间随插口数量的增加而增加。

22.13 小结

每个Internet插口都有一个相关的Internet PCB：TCP、UDP和原始IP。它包含了Internet插口的一般信息：本地和外部IP地址，指向一个路由结构的指针等等。给定协议的所有PCB都放在该协议维护的一个双向链表上。

本章中，我们研究了多个操作PCB的函数，对其中的三个作了详细的讨论：

1) TCP和UDP调用in_pcblookup分用每个进入的数据报。它选择接收数据报的插口，考虑通配匹配。

in_pcbbind也调用这个函数来验证本地地址和本地进程是唯一的；in_pcbconnect调用这个函数验证本地地址、本地进程、外部地址和外部进程的组合是唯一的。

2) in_pcbbind显式或隐式地把一个本地地址和本地端口号绑定到一个插口。当进程调用bind时，发生显式绑定；当一个TCP客户程序调用connect而不调用bind时，或当一个

UDP进程调用sendto或connect而不调用bind时，发生隐式绑定。

3) in_pcbconnect设置外部地址和外部进程。如果进程还没有设置本地地址，计算一条到外部地址的路由，结果的本地接口成为本地地址。如果进程还没有设置本地端口，in_pcbbind为插口选择一个临时端口。

图22-37对多种TCP和UDP应用程序以及存放在PCB中的本地地址，本地端口、外部地址和外部端口的值做了总结。我们还没有讨论完图 22-37中TCP和UDP进程的所有动作，将在后面的章节中继续讨论。

应用程序	本地地址：inp_laddr	本地端口：inp_lport	外部地址：inp_faddr	外部端口：inp_fport
TCP客户程序：connect(*foreignIP, fport*)	in_pcbconnect调用rtalloc为*foreignIP*分配路由。本地地址是本地接口	in_pcbconnect调用in_pcbbind选择临时端口。	*foreignIP*	*fport*
TCP客户程序：bind(*localIP, lport*) connect(*foreignIP,fport*)	*localIP*	*lport*	*foreignIP*	*fport*
TCP客户程序：bind(**, lport*) connect(*foreignIP, fport*)	in_pcbconnect调用rtalloc为*foreignIP*分配路由。本地地址是本地接口	*lport*	*fport*	*fport*
TCP客户程序:bind(*localIP, 0*) connect(*foreignIP, fport*)	*localIP*	in_pcbbind选择临时端口	*foreignIP*	*fport*
TCP服务器程序：bind(*localIP, lport*) listen()accept()	*localIP*	*lport*	IP首部内的源地址	TCP首部内的源端口地址
TCP服务器程序：bind(**, lport*) listen() accept()	IP首部里的目的地址	*lport*	*foreignIP*。发送完数据报后，置位为0.0.0.0	TCP首部内的源端口地址
UDP客户程序：sendto(*foreignIP, fport*)	in_pcbconnect调用rtalloc为*foreignIP*分配路由。本地地址是本地接口。在发送完数据报之后，置位为0.0.0.0	in_pcbconnect调用in_pcbbind选择临时端口。后面调用sendto时不改变	*foreignIP*	*fport*。发送完数据报后，置位为0
UDP客户程序：connect(*foreignIP, fport*) write()	in_pcbconnect调用rtalloc为*foreignIP*分配路由。本地地址是本地接口。后面调用write时不改变	in_pcbconnect调用in_pcbbind选择临时端口。后面调用write时不改变	*foreignIP*	*fport*

图22-37 in_pcbbind 和in_pcbconnect 的总结

习题

22.1 在图22-23中，当进程请求一个临时端口，而所有临时端口都被使用时，会发生什么情况？

22.2 在图22-10中，我们显示了两个有正在监听插口的 Telnet服务器：一个具有特定本地IP地址；另一个的本地 IP地址是通配地址。你的系统的 Telnet守护程序允许你指定本地IP地址吗？如果允许，如何指定？

22.3 假定某个插口被绑定到本地插口 {140.252.1.29, 8888}，且这是唯一使用本地插口8888的插口。 (1)当有另一个插口绑定到 {140.252.1.29，8888}时，请执行in_pcbbind的所有步骤，假定没有任何插口选项。 (2)当有另一个插口绑定到通配IP地址，端口8888时，执行in_pcbbind的所有步骤，假定没有任何插口选项。 (3)当有另一个插口绑定到通配 IP地址，端口 8888时，且设定了插口选项SO_REUSEADDR，执行in_pcbbind的所有步骤。

22.4 UDP分配的第一个临时端口号是什么？

22.5 当进程调用bind时，必须填充sockaddr_in结构中的哪一个元素？

22.6 如果进程要bind一个本地广播地址时，会发生什么情况？如果进程要 bind受限广播地址(255.255.255.255)时，会发生什么情况？