北京科海培训中心

• Prentice Hall PTR

UNIX

网络编程（第1卷）

套接口 API 和 X/Open 传输接口 API

[美] W.Richard Stevens 著
施振川 周利民 孙宏晖 等译
杨继张 审校



第 2 版

清华大学出版社

前 言

简介

网络编程即编写通过计算机网络与其他程序进行通信的这类程序。相互通信的网络程序中,一方称为客户程序(client),另一方称为服务器程序(server)。大多数操作系统提供预先编译好的网络程序,例如 TCP/IP 世界中常见的 Web 客户程序(浏览器)和 Web 服务器程序,以及 FTP 和 Telnet 的客户和服务器程序,不过本书叙述如何编写自己的网络程序。

我们使用应用程序编程接口(application programming interface)即 API 编写网络程序。本书将叙述两种网络编程 API:

1. 套接口(sockets),有时称为"Berkeley 套接口",因为它源自 Berkeley Unix。
2. XTI(X/Open 传输接口),它是对 AT&T 开发的传输层接口(TLI)经少量修改的产物。

书中所有例子均取自 Unix 操作系统,不过网络编程所需的基础知识和概念在很大程度上并不依赖于操作系统。这些例子都基于 TCP/IP 协议族,既有 IP 版本 4,又有版本 6。

编写网络程序要求人们了解底层的操作系统和网络协议。本书是作者在这两个领域的其他四本书的基础上书写的,它们的名称在文中缩写如下:

- APUE:*Advanced Programming in the UNIX Environment*[Stevens 1992]
- TCPv1:*TCP/IP Illustrated, Volume 1*[Stevens 1994]
- TCPv2:*TCP/IP Illustrated, Volume 2*[Wright and Stevens 1995]
- TCPv3:*TCP/IP Illustrated, Volume 3*[Stevens 1996]

本书(英文名称为 UNIX Network Programming)第二版仍包含 Unix 和 TCP/IP 协议的有关内容,但同时又给出不少其他四本书的参考点,这样有兴趣的读者就可获得各个主题更为详细的信息。TCPv2 这本书更是如此,因为它叙述并提供了套接口 API 中网络编程函数(socket,bind,connect 等等)在 4.4BSD 上的真正实现。要是人们理解某个特性的实现,他们在应用程序中使用这个特性将更为明智。

本书与第一版的差别

本书第二版是对第一版的完全重写。这些修改基于作者 1990 年至 1996 年期间大约每月一次讲授这些内容的反馈意见,以及同期跟踪某些 Usenet 新闻组的结果,通过这种跟踪能发现经常被误解的那些主题。下面是新版本中所做的主要改动:

- 新版本中所有例子都使用 ANSI C。
- 老版本的第 6 章("Berkeley 套接口编程")和第 8 章("库例程")已经扩充成总共 25 章。从词数上统计这部分内容扩充了七倍,这也许是从第一版到第二版本的最重大的改动。第一版第 6 章的许多单节已扩充成完整一章,其中增加了更多的例子。

- 老版本第 6 章中的 TCP 和 UDP 部分现已被分开，首先讨论的是 TCP 函数以及一个完整的 TCP 客户-服务器程序例子，接着讨论的是 UDP 函数以及一个完整的 UDP 客户-服务器程序例子。以 connect 函数为例，这么做对于初学者来说要比描述其所有细节更易理解，毕竟在 TCP 和 UDP 中它有不同的语义。
- 老版本的第 7 章（“系统 V 传输层接口编程”）已经扩充成总共 7 章。另外我们讨论的是更新的 XTI API，而不是它所替代的 TLI API。
- 老版本的第 2 章（“Unix 模型”）现已删去。这一章提供约 75 页的 Unix 系统概貌。在 1990 年这一章是必要的，因为那时市面上几乎没有充分叙述基本的 Unix 编程接口的书，更不用说叙述 Berkeley 与系统 V 实现之间的差异了。然而，今天的更多读者对 Unix 已有基本的认识，因此诸如进程 ID、保密字文件、目录和组 ID 等概念不必重述。（对于期望了解 Unix 编程更多细节的读者来说，我的 APUE 一书是这些内容的 700 页扩充。）

  老版本第 2 章中一些高级主题在新版本中仍讨论到，不过它们是分散在各自用到的地方讨论的。例如，在给出第一个并发服务器程序时（4.8 节），我们讨论 fork 函数。在叙述并发服务器如何处理 SIGCHLD 信号时（5.9 节），我们说明 Posix 信号处理的许多其他特性（如僵尸进程、被中断的系统调用等等。）
- 本书尽可能描述 Posix 接口。（我们在 1.10 节中具体介绍标准中的 Posix 一族。）这不仅包括基本 Unix 函数（进程控制、信号等等）的 Posix.1 标准，而且包括套接口和 XTI 网络 API 的即将出台的 Posix.1g 标准，以及线程的 1996 Posix.1 标准。

  在描述诸如 socket 和 connect 等函数时，“系统调用”的称谓改用“函数”代替。这里沿用了 Posix 的一个约定，即系统调用与库函数的区别是一个实现上的细节，通常与程序员无关。
- 老版本的第 4 章（“网络入门”）和第 5 章（“通信协议”）现已被替换成涵盖 IP 版本 4（IPv4）和版本 6（IPv6）的附录 A 以及涵盖 TCP 和 UDP 的第 2 章。这些新内容的焦点在于网络程序员确实会碰到的那些网络协议问题。IPv6 的内容也包括在内，尽管它的实现才刚刚开始出现，然而在本书的生命期内它也许会成为占支配地位的网络协议。

  说实话，我在讲授网络编程时发现，所有的网络编程问题中约有 80%跟网络编程本身是不相关的。也就是说，这些问题不是针对诸如 accept 和 select 等 API 函数的，而是因缺乏对底层网络协议的理解而引起的。例如，我发现一旦学生理解了 TCP 的三路握手和四分组连接终止序列，许多网络编程问题也就迎刃而解了。

  老版本中关于 XNS、SNA、NetBIOS、OSI 协议以及 UUCP 的章节现已删去，因为这些专属协议在 20 世纪 90 年代初期与 TCP/IP 协议相比已显得黯然失色。（UUCP 仍然流行，而且也不是专属的，不过从网络编程角度看就它的使用而言也没有什么可说的。）
- 第二版包含下列新的主题：
  - IPv4 和 IPv6 的互操作性（第 10 章）
  - 协议无关的名字转换（第 11 章）
  - 路由套接口（第 17 章）

- 多播(第 19 章)
- 线程(第 23 章)
- IP 选项(第 24 章)
- 数据链路访问(第 26 章)
- 客户-服务器程序其他设计方法(第 27 章)
- 虚拟网络与隧道通路(附录 B)
- 网络程序调试技术(附录 C)

遗憾的是,对本书第一版内容所做的扩充如此之多,从而无法放在单独一本书中。因此,这套《UNIX 网络编程》系列丛书至少还有另外两卷也在写作计划之中。

- 卷 2 的副标题或许是《IPC:进程间通信》(英文名称为 IPC: Interprocess-Communication),它由老版本的第 3 章扩充而来,同时涵盖 1996 Posix.1 实时 IPC 机制。
- 卷 3 的副标题或许是《应用程序》(英文名称为 Applications),它是对老版本中第 9 至第 18 章的扩充。

尽管大多数网络应用程序将在卷 3 中介绍,但是少量特殊的应用程序还要在本书中介绍,它们是:Ping,Traceroute 和 inetd。

读者

本书既可用作网络编程的指导书,也可作为有经验程序员的参考书。当用作指导书或网络编程入门班的教材时,重点应放在第 2 部分"基本套接口编程"(第 3 章~第 9 章),再跟一些感兴趣的其他主题。第 2 部分的内容包括 TCP 和 UDP 的基本套接口函数、I/O 复用、套接口选项以及基本的名字与地址转换。第 1 章所有读者都必须阅读,特别是 1.4 节,它描述了一些全文都用到的包裹函数。第 2 章及附录 A 读者应根据自己的背景知识选读。第 3 部分"高级套接口编程"的大多数章节可彼此独立地阅读。

为便于用作参考书,本书提供了全文索引,在附录 G、H 上给出了所有的函数和结构的具体讲述所在的页码。为帮助那些以随意顺序阅读各主题的读者,全文提供了大量的对相关主题的参考点。

尽管套接口 API 已成为网络编程的既成事实标准,XTI API 也仍在使用,有时是用在非 TCP/IP 的协议族上。虽然第 4 部分讨论 XTI 的篇幅比第 2 部分和第 3 部分讨论套接口的篇幅小得多,但是在套接口 API 讨论中所叙述的概念同样也适用于 XTI API。例如,不论使用哪一个 API(套接口或 XTI),有关非阻塞 I/O、广播、多播、信号驱动 I/O、带外数据和线程的使用都有相同的概念。其实,许多网络编程问题基本类似,与程序是使用套接口 API 还是使用 XTI API 编写的无关,很少有使用这个 API 能干而使用那个 API 却干不了的事。一句话,概念相同,差别仅在函数名和参数而已。

源码和勘误表获取

本书中出现的所有例子的源码都可从我的主页获取,其 URL 地址列在本前言的末尾。学习网络编程的最好方法是使用这些程序,然后修改并改进它们。只有真正编写这种形式的代码,才能加深对概念的理解并提高编程技巧。各章最后提供了大量的习题,附录 E 给出了

其中大多数的解答。

本书最新的勘误表也可从我的网页获取。[①]

W. Richard Stevens
rstevens@kohala.com
http://www.kohala.com/~rstevens
1997 年 9 月于亚利桑那州 Tucson

① 译者注：中译本已根据最后修改日期为 1999 年 1 月 31 日的最新勘误表(打印在宽行打印纸上共有 12 页)作过订正。

目 录

前言 …… (1)

第 1 部分 简介和 TCP/IP

第 1 章 简介 …… (1)

1.1 概述 …… (1)
1.2 一个简单的时间/日期客户程序 …… (4)
1.3 协议无关性 …… (8)
1.4 错误处理:包裹函数 …… (9)
1.5 一个简单的时间/日期服务器程序 …… (10)
1.6 书中客户-服务器程序例子索引表 …… (13)
1.7 OSI 模型 …… (15)
1.8 BSD 网络支持历史 …… (16)
1.9 测试用网络及主机 …… (16)
1.10 Unix 标准 …… (20)
1.11 64 位体系结构 …… (22)
1.12 小结 …… (23)
1.13 习题 …… (24)

第 2 章 传输层:TCP 和 UDP …… (25)

2.1 概述 …… (25)
2.2 总图 …… (25)
2.3 UDP:用户数据报协议 …… (27)
2.4 TCP:传输控制协议 …… (28)
2.5 TCP 连接的建立和终止 …… (29)
2.6 TIME_WAIT 状态 …… (35)
2.7 端口号 …… (36)
2.8 TCP 端口号与并发服务器 …… (38)
2.9 缓冲区大小及限制 …… (39)
2.10 标准因特网服务 …… (44)
2.11 常见因特网应用程序的协议使用 …… (45)
2.12 小结 …… (46)
2.13 习题 …… (46)

第2部分 基本套接口编程

第3章 套接口编程简介 …… (47)

3.1 概述 …… (47)
3.2 套接口地址结构 …… (47)
3.3 值-结果参数 …… (52)
3.4 字节排序函数 …… (54)
3.5 字节操纵函数 …… (57)
3.6 inet_aton、inet_addr 和 inet_ntoa 函数 …… (58)
3.7 inet_pton 和 inet_ntop 函数 …… (59)
3.8 sock_ntop 和相关函数 …… (62)
3.9 readn、writen 和 readline 函数 …… (64)
3.10 isfdtype 函数 …… (67)
3.11 小结 …… (68)
3.12 习题 …… (69)

第4章 基本 TCP 套接口编程 …… (70)

4.1 概述 …… (70)
4.2 socket 函数 …… (70)
4.3 connect 函数 …… (73)
4.4 bind 函数 …… (75)
4.5 listen 函数 …… (77)
4.6 accept 函数 …… (83)
4.7 fork 和 exec 函数 …… (85)
4.8 并发服务器 …… (87)
4.9 close 函数 …… (89)
4.10 getsockname 和 getpeername 函数 …… (90)
4.11 小结 …… (92)
4.12 习题 …… (92)

第5章 TCP 客户-服务器程序例子 …… (93)

5.1 概述 …… (93)
5.2 TCP 回射服务器程序:main 函数 …… (94)
5.3 TCP 回射服务器程序:str_echo 函数 …… (95)
5.4 TCP 回射客户程序:main 函数 …… (95)
5.5 TCP 回射客户程序:str_cli 函数 …… (96)
5.6 正常启动 …… (97)
5.7 正常终止 …… (98)
5.8 Posix 信号处理 …… (99)
5.9 处理 SIGCHLD 信号 …… (102)
5.10 wait 和 waitpid 函数 …… (105)

5.11 accept 返回前连接夭折 …… (108)
5.12 服务器进程终止 …… (110)
5.13 SIGPIPE 信号 …… (111)
5.14 服务器主机崩溃 …… (113)
5.15 服务器主机崩溃后重启 …… (113)
5.16 服务器主机关机 …… (114)
5.17 TCP 程序例子小结 …… (114)
5.18 数据格式 …… (116)
5.19 小结 …… (118)
5.20 习题 …… (119)

第 6 章 I/O 复用:select 和 poll 函数 …… (121)

6.1 概述 …… (121)
6.2 I/O 模型 …… (121)
6.3 select 函数 …… (126)
6.4 str_cli 函数(修订版) …… (131)
6.5 批量输入 …… (133)
6.6 shutdown 函数 …… (135)
6.7 str_cli 函数(再修订版) …… (137)
6.8 TCP 回射服务器程序(修订版) …… (138)
6.9 pselect 函数 …… (143)
6.10 poll 函数 …… (144)
6.11 TCP 回射服务器程序(再修订版) …… (146)
6.12 小结 …… (149)
6.13 习题 …… (149)

第 7 章 套接口选项 …… (151)

7.1 概述 …… (151)
7.2 getsockopt 和 setsockopt 函数 …… (151)
7.3 检查选项是否受支持并获取缺省值 …… (153)
7.4 套接口状态 …… (156)
7.5 基本套接口选项 …… (156)
7.6 IPv4 套接口选项 …… (169)
7.7 ICMPv6 套接口选项 …… (170)
7.8 IPv6 套接口选项 …… (171)
7.9 TCP 套接口选项 …… (172)
7.10 fcntl 函数 …… (175)
7.11 小结 …… (178)
7.12 习题 …… (178)

第 8 章 基本 UDP 套接口编程 …… (180)

8.1 概述 …… (180)

8.2 recvfrom 和 sendto 函数 …… (180)
8.3 UDP 回射服务器程序:main 函数 …… (182)
8.4 UDP 回射服务器程序:dg_echo 函数 …… (182)
8.5 UDP 回射客户程序:main 函数 …… (184)
8.6 UDP 回射客户程序:dg_cli 函数 …… (185)
8.7 数据报的丢失 …… (185)
8.8 验证接收到的响应 …… (186)
8.9 服务器进程未运行 …… (188)
8.10 UDP 程序例子小结 …… (189)
8.11 UDP 的 connect 函数 …… (191)
8.12 dg_cli 函数(修订版) …… (194)
8.13 UDP 缺乏流量控制 …… (195)
8.14 UDP 中外出接口的确定 …… (199)
8.15 使用 select 函数的 TCP 和 UDP 回射服务器程序 …… (200)
8.16 小结 …… (202)
8.17 习题 …… (202)

第 9 章 基本名字与地址转换 …… (204)

9.1 概述 …… (204)
9.2 域名系统 …… (204)
9.3 gethostbyname 函数 …… (207)
9.4 RES_USE_INET6 解析器选项 …… (211)
9.5 gethostbyname2 函数与 IPv6 支持 …… (212)
9.6 gethostbyaddr 函数 …… (214)
9.7 uname 函数 …… (215)
9.8 gethostname 函数 …… (216)
9.9 getservbyname 和 getservbyport 函数 …… (216)
9.10 其他网络相关信息 …… (219)
9.11 小结 …… (220)
9.12 习题 …… (221)

第 3 部分 高级套接口编程

第 10 章 IPv4 和 IPv6 的互操作性 …… (222)

10.1 概述 …… (222)
10.2 IPv4 客户与 IPv6 服务器 …… (222)
10.3 IPv6 客户与 IPv4 服务器 …… (226)
10.4 IPv6 地址测试宏 …… (227)
10.5 IPV6_ADDRFORM 套接口选项 …… (228)
10.6 源代码可移植性 …… (230)
10.7 小结 …… (230)
10.8 习题 …… (231)

第 11 章 高级名字与地址转换 …… (232)

11.1 概述 …… (232)
11.2 getaddrinfo 函数 …… (232)
11.3 gai_strerror 函数 …… (236)
11.4 freeaddrinfo 函数 …… (237)
11.5 getaddrinfo 函数:IPv6 和 UNIX 域 …… (238)
11.6 getaddrinfo 函数:例子 …… (240)
11.7 host_serv 函数 …… (241)
11.8 tcp_connect 函数 …… (242)
11.9 tcp_listen 函数 …… (245)
11.10 udp_client 函数 …… (250)
11.11 udp_connect 函数 …… (252)
11.12 udp_server 函数 …… (253)
11.13 getnameinfo 函数 …… (255)
11.14 可重入函数 …… (256)
11.15 gethostbyname_r 和 gethostbyaddr_r 函数 …… (259)
11.16 getaddrinfo 和 getnameinfo 函数的实现 …… (261)
11.17 小结 …… (281)
11.18 习题 …… (282)

第 12 章 守护进程和 inetd 超级服务器 …… (283)

12.1 概述 …… (283)
12.2 syslogd 守护进程 …… (284)
12.3 syslog 函数 …… (284)
12.4 daemon_init 函数 …… (287)
12.5 inetd 守护进程 …… (290)
12.6 daemon_inetd 函数 …… (294)
12.7 小结 …… (296)
12.8 习题 …… (297)

第 13 章 高级 I/O 函数 …… (298)

13.1 概述 …… (298)
13.2 套接口超时 …… (298)
13.3 recv 和 send 函数 …… (302)
13.4 readv 和 writev 函数 …… (304)
13.5 recvmsg 和 sendmsg 函数 …… (305)
13.6 辅助数据 …… (309)
13.7 排队的数据量 …… (312)
13.8 套接口与标准 I/O …… (312)
13.9 T/TCP:事务 TCP …… (315)
13.10 小结 …… (317)

13.11 习题 …… (317)

第 14 章 Unix 域协议 …… (318)

14.1 概述 …… (318)
14.2 Unix 域套接口地址结构 …… (318)
14.3 socketpair 函数 …… (321)
14.4 套接口函数 …… (321)
14.5 Unix 域字节流客户-服务器程序 …… (322)
14.6 Unix 域数据报客户-服务器程序 …… (323)
14.7 描述字传递 …… (325)
14.8 接收发送者的凭证 …… (332)
14.9 小结 …… (336)
14.10 习题 …… (336)

第 15 章 非阻塞 I/O …… (338)

15.1 概述 …… (338)
15.2 非阻塞读和写:str_cli 函数(修订版) …… (339)
15.3 非阻塞 connect …… (348)
15.4 非阻塞 connect:日期/时间客户程序 …… (349)
15.5 非阻塞 connect:Web 客户程序 …… (352)
15.6 非阻塞 accept …… (360)
15.7 小结 …… (361)
15.8 习题 …… (362)

第 16 章 ioctl 操作 …… (363)

16.1 概述 …… (363)
16.2 ioctl 函数 …… (363)
16.3 套接口操作 …… (364)
16.4 文件操作 …… (365)
16.5 接口配置 …… (366)
16.6 get_ifi_info 函数 …… (367)
16.7 接口操作 …… (375)
16.8 ARP 高速缓存操作 …… (376)
16.9 路由表操作 …… (378)
16.10 小结 …… (378)
16.11 习题 …… (379)

第 17 章 路由套接口 …… (380)

17.1 概述 …… (380)
17.2 数据链路套接口地址结构 …… (380)
17.3 读和写 …… (381)
17.4 sysctl 操作 …… (388)

17.5 get_ifi_info 函数 …… (392)
17.6 接口名和索引函数 …… (395)
17.7 小结 …… (399)
17.8 习题 …… (400)

第 18 章 广播 …… (401)

18.1 概述 …… (401)
18.2 广播地址 …… (402)
18.3 单播和广播的比较 …… (403)
18.4 使用广播的 dg_cli 函数 …… (406)
18.5 竞争状态 …… (409)
18.6 小结 …… (416)
18.7 习题 …… (416)

第 19 章 多播 …… (417)

19.1 概述 …… (417)
19.2 多播地址 …… (417)
19.3 局域网上多播和广播的比较 …… (420)
19.4 广域网上的多播 …… (422)
19.5 多播套接口选项 …… (424)
19.6 mcast_join 和相关函数 …… (427)
19.7 使用多播的 dg_cli 函数 …… (430)
19.8 接收 MBone 会话声明 …… (431)
19.9 发送和接收 …… (434)
19.10 SNTP:简单网络时间协议 …… (436)
19.11 SNTP(续) …… (440)
19.12 小结 …… (451)
19.13 习题 …… (452)

第 20 章 高级 UDP 套接口编程 …… (454)

20.1 概述 …… (454)
20.2 接收标志、目的 IP 地址和接口索引 …… (454)
20.3 数据报截断 …… (461)
20.4 何时使用 UDP 而不是 TCP …… (461)
20.5 给 UDP 应用程序增加可靠性 …… (463)
20.6 捆绑接口地址 …… (472)
20.7 并发 UDP 服务器 …… (476)
20.8 IPv6 分组信息 …… (478)
20.9 小结 …… (480)
20.10 习题 …… (481)

第 21 章 带外数据 …… (482)

21.1 概述 …… (482)

21.2 TCP 带外数据 …… (482)
21.3 sockatmark 函数 …… (488)
21.4 TCP 带外数据小结 …… (494)
21.5 客户-服务器心博函数 …… (495)
21.6 小结 …… (499)
21.7 习题 …… (499)

第 22 章 信号驱动 I/O …… (501)

22.1 概述 …… (501)
22.2 套接口上的信号驱动 I/O …… (501)
22.3 使用 SIGIO 的 UDP 回射服务器程序 …… (503)
22.4 小结 …… (508)
22.5 习题 …… (509)

第 23 章 线程 …… (510)

23.1 概述 …… (510)
23.2 基本线程函数:创建和终止 …… (511)
23.3 使用线程的 str_cli 函数 …… (513)
23.4 使用线程的 TCP 回射服务器程序 …… (515)
23.5 线程特定数据 …… (519)
23.6 Web 客户与同时连接 …… (526)
23.7 互斥锁 …… (529)
23.8 条件变量 …… (532)
23.9 Web 客户与同时连接(续) …… (535)
23.10 小结 …… (537)
23.11 习题 …… (537)

第 24 章 IP 选项 …… (539)

24.1 概述 …… (539)
24.2 IPv4 选项 …… (539)
24.3 IP 源路径选项 …… (540)
24.4 IPv6 扩展头部 …… (547)
24.5 IPv6 步跳选项和目的选项 …… (548)
24.6 IPv6 路由头部 …… (551)
24.7 IPv6 粘附选项 …… (554)
24.8 小结 …… (555)
24.9 习题 …… (555)

第 25 章 原始套接口 …… (557)

25.1 概述 …… (557)
25.2 原始套接口创建 …… (557)
25.3 原始套接口输出 …… (558)

25.4 原始套接口输入 …… (559)
25.5 Ping 程序 …… (561)
25.6 Traceroute 程序 …… (572)
25.7 一个 ICMP 消息守护进程 …… (583)
25.8 小结 …… (597)
25.9 习题 …… (597)

第 26 章 数据链路访问 …… (599)

26.1 概述 …… (599)
26.2 BPF:BSD 分组过滤器 …… (599)
26.3 DLPI:数据链路提供者接口 …… (601)
26.4 Linux:SOCK_PACKET …… (602)
26.5 libpcap:分组捕获函数库 …… (603)
26.6 检查 UDP 的校验和字段 …… (603)
26.7 小结 …… (619)
26.8 习题 …… (619)

第 27 章 客户-服务器程序其他设计方法 …… (620)

27.1 概述 …… (620)
27.2 TCP 客户程序其他设计方法 …… (622)
27.3 TCP 测试用客户程序 …… (623)
27.4 TCP 迭代服务器程序 …… (624)
27.5 TCP 并发服务器程序,每个客户一个子进程 …… (624)
27.6 TCP 预先派生子进程服务器程序,accept 无上锁保护 …… (627)
27.7 TCP 预先派生子进程服务器程序,accept 使用文件锁保护 …… (633)
27.8 TCP 预先派生子进程服务器程序,accept 使用线程互斥锁保护 …… (636)
27.9 TCP 预先派生子进程服务器程序,传递描述字 …… (637)
27.10 TCP 并发服务器程序,每个客户一个线程 …… (642)
27.11 TCP 预先创建线程服务器程序,每个线程各自 accept …… (643)
27.12 TCP 预先创建线程服务器程序,主线程统一 accept …… (645)
27.13 小结 …… (648)
27.14 习题 …… (649)

第 4 部分 XTI:X/Open 传输接口编程

第 28 章 XTI:TCP 客户程序 …… (650)

28.1 概述 …… (650)
28.2 t_open 函数 …… (651)
28.3 t_error 和 t_strerror 函数 …… (654)
28.4 netbuf 结构和 XTI 结构 …… (655)
28.5 t_bind 函数 …… (656)
28.6 t_connect 函数 …… (658)

28.7 t_rcv 和 t_snd 函数 …… (658)
28.8 t_look 函数 …… (660)
28.9 t_sndrel 和 t_rcvrel 函数 …… (661)
28.10 t_snddis 和 t_rcvdis 函数 …… (662)
28.11 XTI TCP 时间/日期客户程序 …… (663)
28.12 xti_rdwr 函数 …… (666)
28.13 小结 …… (667)
28.14 习题 …… (667)

第 29 章 XTI:名字与地址函数 …… (668)

29.1 概述 …… (668)
29.2 /etc/netconfig 文件与 netconfig 函数 …… (668)
29.3 NETPATH 环境变量与 netpath 函数 …… (670)
29.4 netdir 函数 …… (671)
29.5 t_alloc 和 t_free 函数 …… (673)
29.6 t_getprotaddr 函数 …… (675)
29.7 xti_ntop 函数 …… (675)
29.8 tcp_connect 函数 …… (676)
29.9 小结 …… (680)
29.10 习题 …… (680)

第 30 章 XTI:TCP 服务器程序 …… (681)

30.1 概述 …… (681)
30.2 t_listen 函数 …… (682)
30.3 tcp_listen 函数 …… (683)
30.4 t_accept 函数 …… (685)
30.5 xti_accept 函数 …… (686)
30.6 简单的时间/日期服务器程序 …… (687)
30.7 多个待处理连接 …… (689)
30.8 xti_accept 函数(修订版) …… (691)
30.9 小结 …… (698)
30.10 习题 …… (698)

第 31 章 XTI:UDP 客户和服务器程序 …… (700)

31.1 概述 …… (700)
31.2 t_rcvudata 和 t_sndudata 函数 …… (700)
31.3 udp_client 函数 …… (700)
31.4 t_rcvuderr 函数:异步错误 …… (704)
31.5 udp_server 函数 …… (706)
31.6 分片读取数据报 …… (708)
31.7 小结 …… (710)

第 32 章 XTI 选项 …… (711)

32.1 概述 …… (711)
32.2 t_opthdr 结构 …… (713)
32.3 XTI 选项 …… (714)
32.4 t_optmgmt 函数 …… (717)
32.5 检查选项是否受支持并获取缺省值 …… (718)
32.6 获取和设置 XTI 选项 …… (721)
32.7 小结 …… (724)

第 33 章 流 …… (725)

33.1 概述 …… (725)
33.2 概貌 …… (725)
33.3 getmsg 和 putmsg 函数 …… (729)
33.4 getpmsg 和 putpmsg 函数 …… (730)
33.5 ioctl 函数 …… (731)
33.6 TPI:传输提供者接口 …… (731)
33.7 小结 …… (740)
33.8 习题 …… (740)

第 34 章 XTI:其他函数 …… (741)

34.1 概述 …… (741)
34.2 非阻塞 I/O …… (741)
34.3 t_rcvconnect 函数 …… (741)
34.4 t_getinfo 函数 …… (742)
34.5 t_getstate 函数 …… (742)
34.6 t_sync 函数 …… (743)
34.7 t_unbind 函数 …… (745)
34.8 t_rcvv 和 t_rcvvudata 函数 …… (745)
34.9 t_sndv 和 t_sndvudata 函数 …… (746)
34.10 t_rcvreldata 和 t_sndreldata 函数 …… (746)
34.11 信号驱动 I/O …… (747)
34.12 带外数据 …… (748)
34.13 回馈传输提供者 …… (752)
34.14 小结 …… (753)

第 5 部分 附 录

附录 A IPv4、IPv6、ICMPv4 和 ICMPv6 …… (754)

A.1 概述 …… (754)
A.2 IPv4 头部 …… (754)

A.3 IPv6 头部 …… (755)
A.4 IPv4 地址 …… (758)
A.5 IPv6 地址 …… (762)
A.6 ICMPv4 和 ICMPv6:网际控制消息协议 …… (766)

附录 B 虚拟网络 …… (769)

B.1 概述 …… (769)
B.2 MBone …… (769)
B.3 6bone …… (771)

附录 C 调试技术 …… (773)

C.1 系统调用跟踪 …… (773)
C.2 标准因特网服务 …… (777)
C.3 sock 程序 …… (778)
C.4 小测试程序 …… (780)
C.5 tcpdump 程序 …… (782)
C.6 netstat 程序 …… (782)
C.7 lsof 程序 …… (783)

附录 D 杂凑的源代码 …… (784)

D.1 unp.h 头文件 …… (784)
D.2 config.h 头文件 …… (787)
D.3 unpxti.h 头文件 …… (789)
D.4 标准错误处理函数 …… (791)

附录 E 部分习题解答 …… (794)

附录 F 参考文献 …… (828)

附录 G 函数和宏定义索引表 …… (837)

附录 H 结构定义索引表 …… (840)

附录 I 中英文对照词汇表 …… (842)

第1部分 简介和TCP/IP

第1章 简 介

1.1 概 述

大多数网络应用系统可分成两部分：客户(client)和服务器(server)。[①] 图1.1画出了这两部分及它们之间的通信链路。

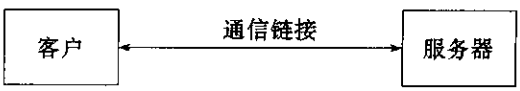


图1.1 网络应用系统：客户和服务器

大多数读者都已熟悉很多客户与服务器的实例：如Web浏览器(客户)和Web服务器的通信；FTP客户从FTP服务器获取文件；Telnet客户通过远程主机上的Telnet服务器提供远程登录手段。

通常客户一次只与一个服务器通信，不过以使用Web浏览器为例，我们也许在10分钟内就可以与许多不同的Web服务器通信。从服务器的角度来看，一个服务器同时与多个客户通信并不稀奇，见图1.2。以后我们将介绍几种让一个服务器同时处理多个客户请求的方法。

客户与服务器之间的通信涉及网络通信协议。本书的焦点是TCP/IP协议族，也称为网际协议族。例如Web客户与服务器之间的通信就使用TCP协议。而TCP又使用IP协议，IP则使用某种形式的数据链路层通信。举例来说，如果客户与服务器处于同一个以太网，我们就有图1.3的通信层次。

虽然客户与服务器之间的通信使用应用协议，传输层内的通信使用TCP协议，如此等等，但实际上，客户与服务器间的信息流在某一端是向下通过协议栈，而在另一端则是向上通过协议栈。

---

① 译者注：本书通篇频繁使用客户(client)和服务器(server)这两个术语。实际上它们的具体含义随上下文而不同，有时指静态的源程序或可执行程序(客户程序和服务器程序)，有时指动态进程(客户进程和服务器进程)，有时又指运行进程的主机(客户主机和服务器主机)。在不引起混淆的前提下，我们简单地称客户进程为客户，称服务器进程为服务器。同样应用(application)这个术语的具体含义也随上下文而变化，有时指程序(应用程序)，有时指进程(应用进程)，有时作为名词性修饰词译为应用。本书有时把同处应用层的客户和服务器对也用应用表示，我们称之为网络应用系统。程序(program)与进程(process)是在系统调用exec上衔接的。exec既可以由shell隐式调用(直接输入命令行执行程序属于这种情况)，也可以在用户程序中显式调用。我们称exec调用后运行中的程序为进程，它继承exec调用者的进程ID，而这个调用进程本身又可能是此前某个fork系统调用的结果。程序和进程的密切关系使得两者有时相互渗透使用，不易区分。

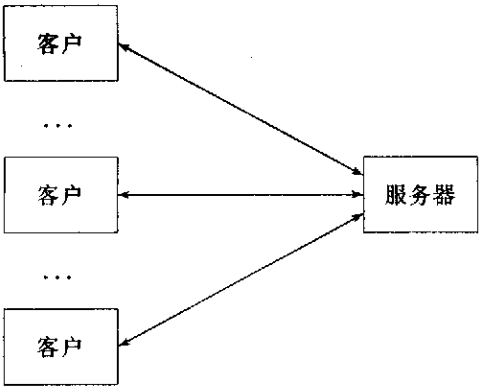


图 1.2 一个服务器同时处理多个客户的请求

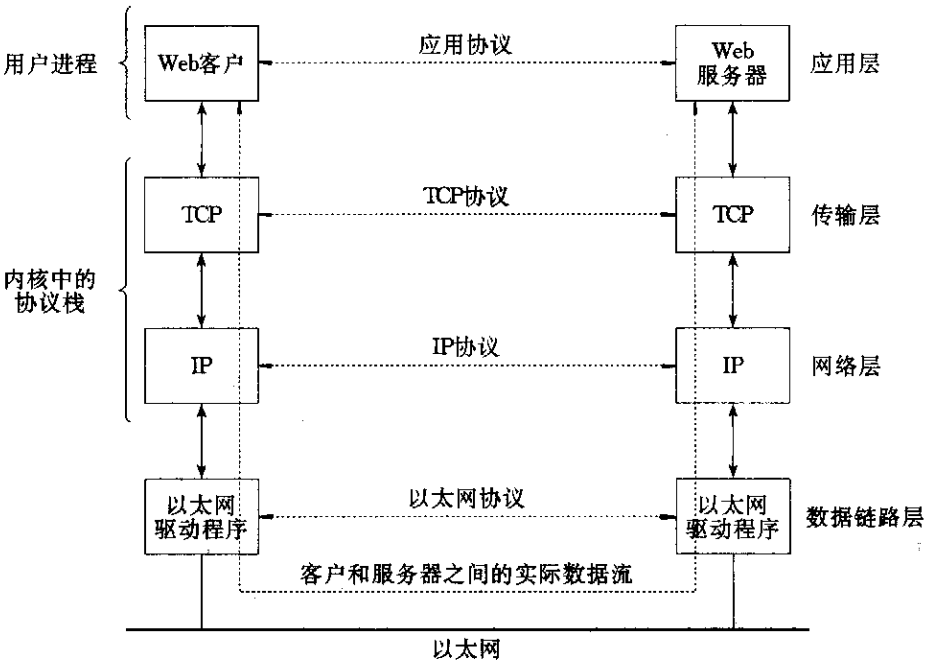


图 1.3 客户与服务器使用 TCP 协议在同一个以太网中通信

值得注意的是,客户与服务器是典型的用户进程,而 TCP 和 IP 协议则通常是系统内核协议栈的一部分。我们在图 1.3 右边标出了四层协议。

本书不只阐述 TCP 和 IP 协议,有些客户和服务器还使用 UDP 协议替代 TCP,第 2 章将详细介绍这两种协议。我们一直说的"IP"协议早在 20 世纪 80 年代早期就已使用,但它的正式名称是 IP 版本 4(IP version 4,即 IPv4)。IP 版本 6(IP version 6,即 IPv6)在 90 年代中期发展起来,将来可能取代 IPv4。本书编写时 IPv6 的最初实现已能获取,因此文中讨论的网络应用程序的开发不仅使用 IPv4,而且使用 IPv6。附录 A 比较了 IPv4 和 IPv6,介绍了 IPv6 最新的补充内容以及正文中会碰到的其他协议。

客户与服务器无需如图 1.3 所示处于同一个局域网。相反,图 1.4 给出了处于不同局域网中的客户与服务器,这两个局域网是使用路由器(router)连接到广域网的。

路由器是广域网的架构设备。今天，最大的广域网是因特网(Internet)。不过很多公司使用自己的广域网，而这些私有的广域网既可以连接也可以不连接到因特网。② 本章以下部分将概述多个主题，它们在以后章节中还要详细介绍。我们从一个完整使用 TCP 协议的简单客户程序开始，它用到了贯穿全书的许多函数调用和概念。这个客户程序仅在 IPv4 上运行，不过我们会指出在 IPv6 上需要修改的地方。较好的方法是编写独立于协议的客户和服务器程序，这在第 11 章中讨论。本章还介绍一个完整的 TCP 服务器程序，可与上述客户程序一起工作。

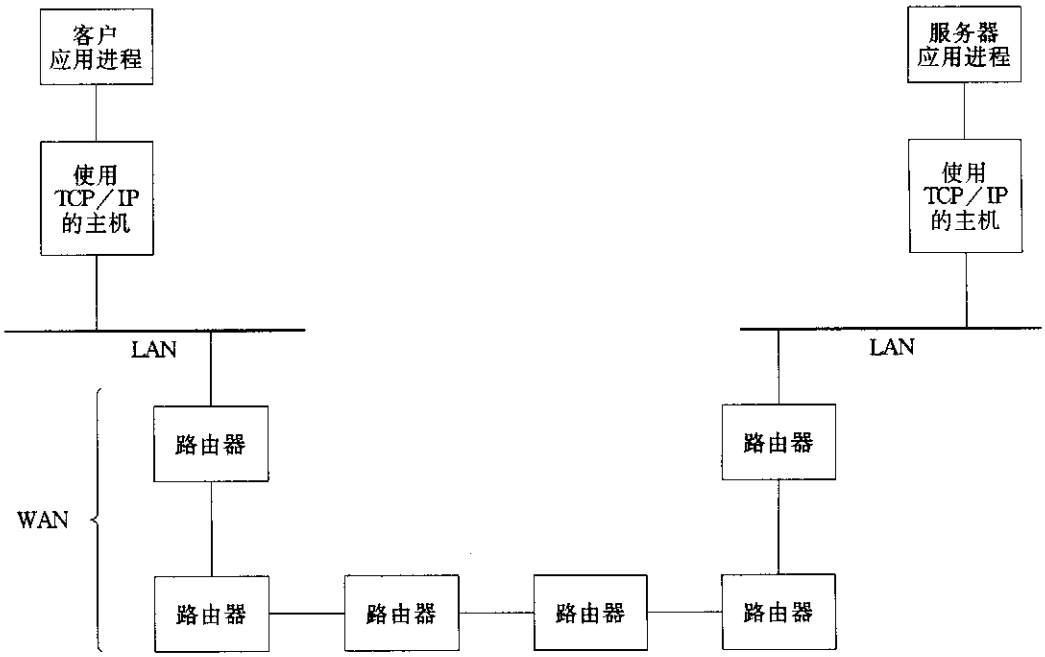


图 1.4 处于不同局域网的客户和服务器主机通过广域网连接

为了简化所有代码，对于本书中要调用的大部分系统函数，我们都定义了包裹函数，它们用于检查错误、输出适当消息及当出错时终止程序运行。我们还给出了本书在后面的许多例子要用到的测试网络、主机、路由器以及主机名、IP 地址和操作系统。

当今介绍的 Unix 时经常使用 Posix 一词，它是一种为多数厂商采用的标准。我们将介绍 Posix 的历史以及它对 API 的影响，再介绍标准化领域的其他主角。

② 译者注： internet 一词有多种含义。一是因特网(the Internet)，它是个专有名词，特指从 ARPANET 发展来的联结全世界的大型互联网。二是互联网(internets)，凡是采用 TCP/IP 协议的网络都可称为互联网，因特网就是互联网之一。三是作为名词性修饰词，这时应根据情况分别译成“因特网”、“互联网”或“网际”。例如“Internet Protocol”译成“网际协议”(注意：“Internet Protocol”是“internet protocol”名词专有化的结果)，“Internet Society”译成“因特网学会”。

1.2 一个简单的时间/日期客户程序

让我们考虑一个特殊的例子，目的是引入在本书中将要遇到的许多概念和说法。图1.5是TCP时间/日期客户程序的一个实现。这个客户建立与服务器的TCP连接并读取服务器送回的当前时间和日期(直观可读格式)。

```
1 #include    "unp.h"
2 int
3 main(int argc, char **argv)
4 {
5     int     sockfd, n;
6     char    recvline[MAXLINE + 1];
7     struct sockaddr_in servaddr;
8     if (argc != 2)
9         err_quit("usage: a.out <IPaddress>");
10    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
11        err_sys("socket error");
12    bzero(&servaddr, sizeof(servaddr));
13    servaddr.sin_family = AF_INET;
14    servaddr.sin_port = htons(13);  /* daytime server */
15    if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0)
16        err_quit("inet_pton error for %s", argv[1]);
17    if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) < 0)
18        err_sys("connect error");
19    while ( (n = read(sockfd, recvline, MAXLINE)) > 0) {
20        recvline[n] = 0;    /* null terminate */
21        if(fputs(recvline, stdout) == EOF)
22            err_sys("fputs error");
23    }
24    if (n < 0)
25        err_sys("read error");
26    exit(0);
27 }
```

图1.5 TCP时间/日期客户程序[intro/daytimetcpcli.c]

这就是本书所有源代码使用的格式。每一非空行都被编号，代码的正文描述部分的左边标有起始与结束的行号。有的段落开始处含有一醒目的简短标题，概述本段代码的内容。

图标题右侧括号中的intro/daytimetcpcli.c表示源代码所在的目录为intro，文件为daytimetcpcli.c。本书所有源代码都可免费获得，见前言部分。编译、运行和修改本书中的程序是学习网络编程概念的好方法。

全文使用缩进的插入式注解(如此处所示)来说明实现的细节和历史上的观点。

如果我们编译这一程序，生成缺省文件a.out并执行它，将有下述的结果：

```
Solaris% a.out 206.62.226.35        我们的输入
Fri Jan 12 14:27:52 1996            程序的输出
```

任何时候，交互时的输入采用黑体字，计算机的输出用白体字。注解用楷体字加在右边。shell 提示带有系统名字(本例中为 solaris)，说明在哪台主机上运行命令。图 1.16 给出本书大多数例子所运行的系统，它们的主机名本身通常就说明各自的操作系统。

在图 1.16 的 27 行网络程序里有许多地方值得考虑，这里我们只做简单介绍，目的是让初次遇到网络程序的读者有所准备，本书以后将对这些主题作更详细的说明。

头文件

第 1 行　包含一个头文件 unp.h，见 D.1 节。这个头文件含有许多大部分网络程序都需要的系统头文件，并定义了所用到的各种常值(如 MAXLINE)。③

命令行参数

第 2～3 行　这是 main 函数的定义，其形式参数是命令行参数。本书例子中的代码全部用 ANSI(美国国家标准局) C 编译器。

创建 TCP 套接口

第 10～11 行　socket 函数创建网际(AF_INET)字节流(SOCK_STREAM)套接口，它是 TCP 套接口的特色名字。该函数返回一小整数描述字，在以后的其他函数调用中(如随后的 connect 和 read 调用)，我们就用它来标识这个套接口。

if 语句包含对 socket 函数的调用，并把返回值赋给变量 sockfd，然后测试该值是否小于 0。可把该语句分为两个 C 语句：

```
sockfd = socket(AF_INET,SOCK_STREAM,0);
if (socfd < 0)
```

不过把这两行合并成一行是常见的 C 语句用法。按照 C 语言的优先规则(小于运算符的优先级高于赋值运算符)，函数调用和赋值语句必须用括号括起。作者第一次看到这个样式是在 Minix 源代码中[Tenenbaum 1987]，之后一直沿用。后面的 while 语句也使用相同的样式。

③ 译者注：严格地说，C 语言中用 #define 伪命令定义的对象称为常数，用 const 限定词定义并初始化的对象称为常量(相对于变量而言)。常数的值在编译时就确定，常量的值则在运行时初始化后才确定(但此后它只能作为右值使用)。本书绝大多数恒定值是用 #define 定义的常数。不过“常数”的称谓容易让人狭义地的理解成仅仅是数而已，因此本书统一用“常值”指代其值恒定不变的对象。

以后我们将遇到术语套接口(socket)的许多不同用法。④ 首先,我们正在使用的应用程序编程接口称为套接口API(sockets API)。上一段中名为socket的函数就是套接口API的一部分。上一段中我们还提到了"TCP套接口",它是"TCP端点(TCP endpoint)"的同义词。

如果socket函数调用失败,我们就调用自己的err_sys放弃执行。该函数输出我们提供的出错消息和所发生系统错误的描述信息(如"Protocol not supported(协议不受支持)"等)后,终止进程的执行。这个函数和带有err_前缀的少数其他函数的调用将贯穿全书,见D.4节。

指定服务器IP地址和端口

第12～16行 我们把服务器的IP地址和端口号填入网际套接口地址结构(名为servaddr的sockaddr_in结构变量)。调用bzero把整个结构清零,置地址族为AF_INET,端口号为13(这是daytime服务器的众所周知端口,支持这一服务的任何TCP/IP主机都一样,见图2.13),置IP地址为命令行的第一个参数值(argv[1])。IP地址和端口成员必须具有合适格式:我们调用库函数htons("主机到网络短整数")去变换二进制端口号,调用库函数inet_pton("表达到数值")去变换ASCII命令行参数(例如206.62.226.35)到合适的格式。

> bzero不是ANSI C的函数,它起源于Berkeley网络代码。不过,我们在整本书中始终用它代替ANSI C的memset函数,因为bzero(两个参数)比menset(三个参数)更便于记忆。几乎所有支持套接口API的厂商也提供bzero,如果没有这一函数,可用unp.h头文件里提供的这一函数的宏定义代替。
>
> 在TCPv3第一次印刷本中,作者犯了在10多处对换memset函数第二和第三个参数的错误。C编译器找不出这一错误,因为两个参数是同种类型。(其实,第二个参数是int类型,而第三个参数是size_t,为典型的unsigned int类型,但是分别指定其值为0和16对于另一个参数的类型也适用。)对memset的这些调用仍然工作,但不做任何事,因为待初始化的字节数被指定成0。程序之所以仍能工作是因为只有很少的套接口函数要求网际套接口地址结构的最后8个字节置0,不过,它确实是错误的。使用bzero函数可以避免这种错误,因为如果使用函数原型,C编译器总能发现bzero的两个参数是否对换了。
>
> inet_pton函数是一个支持IPv6的新函数,见附录A的详细说明。老代码使用inet_addr函数把ASCII点分十进制数串变换成正确的格式,但它有许多限制,而inet_pton改正了这些缺陷。如果你的系统尚未支持该函数,那你可用3.7节提供的它的实现。

---

④ 译者注:socket一词应译成"套接口",其理由如下。首先,作为网络编程API之一的套接口(sockets,注意这种用法时总是用复数形式,即使用作名词性修饰词也这样,如sockets API、sockets library等)跟XTI一样,是应用层到传输层的接口。其次,具体使用中的套接口是与Unix管道的一端类似的东西,它是具体的,我们既可以往这个"口"写数据,也可以从这个"口"读数据。最后,套接口函数使用套接口描述字访问具体的套接口,如果把套接口描述字的简称sockfd译成"套接字"到比较合适,从这个意义上看,一个套接口可对应多个套接字,因为Unix的描述字既可以复制,也可以继承;反过来,一个套接字对应且只对应一个套接口。

建立与服务器的连接

第17～18行 connect函数用在TCP套接口上的功能是:跟由它的第二个参数所指的套接口地址结构对应的服务器建立连接。connect的第三个参数说明套接口地址结构的长度,对于网际套接口地址结构,我们总是使用C语言的操作符sizeof由编译器来计算。

SA在头文件unp.h中定义为struct sockaddr,即通用套接口地址结构。套接口函数每次需要一个指向套接口地址结构的指针时,这个指针必须转换成指向一个通用套接口地址结构的指针类型。这是因为套接口函数早于ANSI C标准,因此在20世纪80年代早期开发这些函数时,void * 指针类型还不可用。问题是串"struct sockaddr"有15字符长,往往造成源代码行超出屏幕的右边,因此我们把它缩成SA。通用套接口地址结构在介绍图3.3时将详细讨论。

读入并输出服务器的应答

第19～25行 我们使用read函数读服务器的应答,并用标准的I/O函数fputs输出结果。[5]我们必须注意,TCP是一种无记录边界的字节流协议。本例中,服务器的应答是具有下面格式的26字节字符串。

```
Fri Jan 12 14:27:52 1996 \r\n
```

其中,\r\n是ASCII字符集的回车和换行符。在字节流的协议里,这个26字节的数据可以有多种返回方式:既可以是包含所有26个字节的单个TCP分节,也可以是每个分节只含一个字节的总共26个分节,总共26字节的任何其他组合也可以。[6] 通常服务器返回包含所有26个字节的单个分节,但是如果数据量很大,我们不能确保一次read调用就返回服务器的应答。因此从TCP套接口读数据时,总是把read编码在某个循环中,当read返回0(远程端关闭连接)或负值(错误发生)时才终止循环。

本例中,记录的结束由服务器关闭连接表示。这种技术也适用于HTTP(Hypertext

---

⑤ 译者注:为求简洁明确,以后尽可能采用直接把函数名和C语言关键字用作动词的译法。例如本句的这种译法是:"我们read服务器的应答,并fputs结果。"又如"如果connect成功,就break出循环。"的意思是:"如果connect函数调用成功(表示连接成功),就执行C语言的break语句跳出循环。"

⑥ 译者注: 计算机网络各层对等实体间交换的单位信息称为协议数据单元(PDU),分节就是PDU之一,它对应于TCP传输层。由于本书涉及PDU种类较多,为避免混淆,我们在这里汇总简要说明。就TCP/IP协议族而言,应用层实体(如普通的客户和服务器进程)间交换的PDU称为数据(data,TCP应用进程)或记录(record,UDP应用进程),其中数据的大小没有限制,但记录不能超过UDP发送缓冲区大小(这个缓冲区实际上并不存在,但它具备大小这个属性)。传输层实体间交换的PDU是分节(segment,TCP协议)或数据报(datagram,UDP协议),它们的大小都是有限的。TCP应用进程的数据由TCP划分成块(chunk)后封装在分节中传送,UDP应用进程的记录则由UDP整个封装到数据报中传送。网络层实体之间交换的PDU是分组(packet,俗称包),其大小自然有限。传输层的分节或数据报都由IP封装在分组中传送。有时分组可能太大,超过了数据链路层单个PDU的容量,这时分组需要划分成若干个片段(fragment),因此片段也是网络层实体之间交换的信息单元。TCP/IP为提高效率,一般尽可能避免这种分片操作。数据链路层实体之间交换的PDU是帧(frame),网络层的每个小分组(即无需划分片段的分组)或片段由数据链路层封装到一个帧中。由于UDP应用进程、UDP和IP都是无连接的,本书有时统一用数据报表示记录、数据报或分组,读者需加以留意。每层的PDU除用来封装来自紧邻上层的数据单元(称为服务数据单元SDU)外,也用于本层内部的协议通信。例如全书频繁使用的SYN和FIN分节就是TCP专门用来建立与拆除连接的PDU,通常不再携带来自应用进程的SDU。另外书中讨论的MSS是应用层与传输层之间的接口属性,MTU则是网络层与数据链路层之间的接口属性。

Transfer Protocol)协议。其他技术也有效,如FTP(File Transfer Protocol)协议和SMTP(Simple Mail Transfer Protocol)协议都使用由ASCII回车和换行符构成的2字节序列终止记录。Sun RPC(Remote Procedure Call)和DNS(Domain Name System)使用TCP时,在每个记录的前面放一个二进制计数值,由它给出记录的长度。这里有个重要的概念:TCP本身不提供记录结束标记。应用程序需要确定记录边界的话,就必须自己去实现,有一些常用的方法可供选择。

终止程序

第26行 exit语句终止程序。Unix在进程终止时关闭所有描述字,其中包括TCP套接口描述字。

本书将对刚叙述过的所有概念作深入的探讨。

1.3 协议无关性

图1.5程序是与IPv4相关的。分配并初始化sockaddr_in结构,置sin_family成员为AF_INET,指定socket函数的第一个参数为AF_INET。

为使图1.5程序可在IPv6上运行,我们必须修改程序代码。图1.6是一个可以在IPv6上运行的版本,黑体字表示修改的部分。

```
1 #include       "unp.h"
2 int
3 main(int argc,char * * argv)
4 {
5         int       sockfd,n;
6         char      recvline[MAXLINE + 1];
7         struct sockaddr_in6 servaddr;
8     if(argc ! = 2)
9         err_quit("usage:a.out <IPaddress>")
10    if( (sockfd = socket(AF_INET6,SOCK_STREAM,0)) < 0)
11        err_sys("socket error");
12    bzero(&servaddr,sizeof(servaddr));
13    servaddr.sin6_family = AF_INET6;
14    servaddr.sin6_port = htons(13);     /* daytime server */
15    if(inet_pton(AF_INET6,argv[1],&servaddr.sin6_addr) <= 0)
16        err_quit("inet_pton error for %s",argv[1]);
17    if(connect(sockfd,(SA * ) &servaddr,sizeof(servaddr)) < 0) {
18        err_sys("connect error");
19    while ( (n = read(sockfd,recvline,MAXLINE)) > 0) {
20        recvline[n] = 0;      /* null terminate */
21        if(fputs(recvline,stdout) == EOF)
22            err_sys("fputs error");
23    }
24    if (n < 0)
25        err_sys("read error");
26    exit(0);
27 }
```

图1.6 适合于IPv6的图1.5程序修改版[intro/daytimetcpcliv6.c]

上述程序仅需修改5行代码，但这次是一个与IPv6相关的程序。图11.7是一个与协议无关的同一客户程序，它使用getaddrinfo函数(由tcp_connect函数调用)。

本程序的另一个不足之处是用户必须以点分十进制数键入服务器的IP地址(如适合于IPv4的206.62.226.35)。人们更习惯于用名字代替数字(如laptop.kohala.com或只用laptop)，我们将在第9章和第11章阐述主机名与IP地址，服务名与端口号的转换函数。在那两章之前，我们不讨论这些函数，继续采用IP地址和端口号，目的是了解我们必须填写和查看的套接口地址结构的细节，同时又可以避免被其他函数集的细节把网络编程的讨论搞复杂了。

1.4 错误处理：包裹函数

任何现实世界的程序都必须检查每个函数调用是否返回错误。在图1.5程序中，我们检查socket、inet_pton、connect和fputs函数是否返回错误，当发生错误时我们再调用err_quit或err_sys输出出错消息并终止程序。我们发现绝大多数情况下这正是所希望的。个别情况下当返回错误时，我们需要做某些特殊处理，而不是终止程序。图5.12就是这样的例子，我们必须检查系统调用是否被中断。

因为多数情况下程序终止于一个错误，我们可以定义包裹函数来简化我们的程序。包裹函数调用实际函数，检查返回值，发生错误时终止进程。确定包裹函数名的约定是大写实际函数名的第一个字符，如：

```
sockfd=Socket(AF_INET,SOCK_STREAM,0);
```

图1.7是socket的包裹函数。

```
172 int
173 Socket(int family,int type,int protocol)
174 {
175     int     n;
176     if( (n = socket(family,type,protocol)) < 0)
177         err_sys("socket error");
178     return (n);
179 }
```

图1.7 socket函数的包裹函数[lib/wrapsock.c]

每当你遇到一个以大写字母打头的函数名时，它就是我们定义的包裹函数。它调用的实际函数的名字与包裹函数名相同，但以对应的小写字母打头。

这些包裹函数也许不见得如何节省代码量，不过当我们在23章讨论线程时，我们会发现线程函数遇到错误时并不设置标准Unix的errno变量，而是把error的值作为函数返回值返回调用者。这意味着每次我们调用pthread函数，我们必须分配一个变量来存放返回值，再用它来设置errno变量后才能调用err_sys。为避免使用花括号，我们使用C的“逗号”操作符，把errno的赋值与err_sys的调用连成单一语句，如下所示：

```
int n;
if ((n = pthread_mutex_lock (&ndone_mutex)) ! =0)
    errno=n , err_sys("pthread mutex_lock error");
```

虽然我们可以定义一个新的错误处理函数，它取系统错误号为它的一个参数，但是通过定义包裹函数，我们可以很容易地实现如下形式的代码：

```
Pthread_mutex_lock(&ndone_mutex);
```

这个包裹函数的定义如图1.8所示。

```
72 void
73 Pthread_mutex_lock(pthread_mutex_t *mptr)
74 {
75     int     n
76     if ( (n = pthread_mutex_lock(mptr)) == 0)
77         return;
78     errno = n;
79     err_sys("pthread_mutex_lock error");
80 }
```

图1.8 我们的pthread_mutex_lock的包裹函数[lib/wrappthread.c]

仔细推敲编码，我们可以用宏替代函数，从而稍微提高运行效率，不过包裹函数很少是程序性能的瓶颈所在。

选择大写函数名的第一个字母是一种较折衷的方法。还有很多其他方法：如用e做为函数名的前缀（如[Kernighan and Pike 1984]第184页所示）或用_e做为函数名的后缀等等。同样提供调用其他函数的可视指示，我们的这种方法看来是最少分散人们的注意力的。这种技术还有助于检查那些其错误返回值通常忽略的函数，如close和listen。

本书后面的例子我们将普遍使用包裹函数，除非我们必须检查某个确定的错误并处理它，而不是终止进程运行。我们并不给出所有包裹函数的源代码，但它们的代码是免费可得的（见前言）。

Unix errno值

每当在一个Unix函数（如socket函数）中发生错误时，全局变量errno将被置成一个指示错误类型的正数，函数本身则通常返回-1。err_sys检查errno变量并输出其相应的出错消息（例如，当errno值等于ETIMEDOUT时，将输出“Connection timed out（连接超时）”）。

errno的值只在函数发生错误时设置。如果函数不返回错误，errno的值就无定义。在Unix中，所有的错误值都是常值，具有以E打头的全大写字母名字，在头文件<sys/errno.h>中定义。值0不表示任何错误。

把errno值存于全局变量不适合共享所有全局变量的多线程。我们将在23章中讲述解决这一问题的方法。

1.5 一个简单的时间/日期服务器程序

我们可以写一个简单的TCP时间/日期服务器程序，它可以和1.2节的客户程序一道工作。图1.9是这个服务器程序，它使用了上述的包裹函数。

```
1 #include    "unp.h"
2 #include    <time.h>
```

```
3 int
4 main(int argc,char * * argv)
5 {
6       int         listenfd,connfd;
7       struct sockaddr_in servaddr;
8       char        buff[MAXLINE];
9       time_t      ticks;
10      Listenfd = Socket(AF_INET,SOCK_STREAM,0);
11      bzero(&servaddr,sizeof(servaddr));
12      servaddr.sin_family = AF_INET;
13      servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
14      servaddr.sin_port = htons(13);      /* daytime server */
15      Bind(listenfd,(SA * ) &servaddr,sizeof(servaddr));
16      Listen(listenfd,LISTENQ);
17      for( ; ;) {
18          connfd = Accept(listenfd,(SA * )NULL,NULL);
19          ticks = time(NULL);
20          snprintf(buff,sizeof(buff),"%.24s\r\n",ctime(&ticks));
21          Write(connfd,buff,strlen(buff));
22          Close(connfd);
23      }
24 }
```

图 1.9 TCP 时间/日期服务器程序[intro/daytimetcpsrv.c]

创建 TCP 套接口

第 10 行 TCP 套接口的创建与客户程序相同。

捆绑服务器的众所周知端口到套接口

第 11～15 行 通过设置网际套接口地址结构并调用 bind 函数，我们把服务器的众所周知端口捆绑到套接口。我们指定的 IP 地址为 INADDR_ANY，它允许服务器在任意接口上接受客户连接(假定服务器主机有多个接口)。以后我们将了解怎样限定服务器在某个给定接口上接受客户连接。

把套接口变换成监听套接口

第 16 行 通过调用 listen 函数将此套接口变换成一个监听套接口，它使系统内核接受来自客户的连接。socket、bind 和 listen 是任何 TCP 服务器用于准备所谓的监听描述字(listening descriptor，本例为 listenfd)通常的三个步骤。

常值 LISTENQ 在头文件 unp.h 中定义，它指定系统内核允许在这个监听描述字上排队的最大客户连接数。4.5 节将详细说明客户连接的排队。

接受客户连接，发送应答

第 17～21 行 一般情况下，服务器进程在调用 accept 函数后处于睡眠状态，它等待客户的连接和内核对它的接受。TCP 连接使用三路握手(three－way handshake)来建立，当握手完毕时，accept 函数返回，其返回值是一个称为已连接描述字(connected descriptor)的新描述字(connfd)。此描述字用于与新客户的通信。accept 为每个连接到服务器的客户返回一个新的已连接描述字。

本书全文所用的无限循环采用以下风格：

```
for ( ; ; ) {
    . . .
}
```

库函数time返回当前时间和日期，这是自Unix纪元，即1970年1月1日0点0分0秒(国际标准时间)以来的秒数。下一个函数ctime将此整数值变换成直观可读的时间格式，如：

```
Fri Jan 12 14:27:52 1996
```

snprintf函数在这个字符串末端添加了回车和换行两个字符。write函数把结果发送给客户。

这也许是你第一次碰到snprintf函数。许多现有代码调用的是sprintf，但sprintf不检查目标缓冲区是否溢出，相反，snprintf要求其第二个参数是目标缓冲区的大小，因此可确保缓冲区不溢出。snprintf不是标准ANSI C的一部分，但这个标准的修订版C9X正在考虑。不过，许多厂商提供的标准C库含有这个函数。我们在本书里使用snprintf，如果你的系统不提供这个函数，可用我们提供的通过调用sprintf实现的版本。

值得注意的是，许多网络入侵是由黑客通过发送数据，导致服务器对sprintf的调用溢出其缓冲区而发生的。你必须小心使用的函数还有gets、strcat和strcpy，通常应调用fgets、strncat和strncpy函数代替。[Garfinkel and Spafford 1996]的第23章阐述了编写安全的网络程序。

终止连接

第22行 服务器通过调用close关闭与客户的连接。它引发通常的TCP连接终止序列：每个方向上发送一个FIN，每个FIN又由对方确认。2.5节将详细描述三路握手和用于终止TCP连接的四个TCP分节。

如同上节介绍的客户程序，我们仅仅简要地介绍了服务器程序，监听细节则留待本书后面论述。注意以下事项：

- 如同客户程序，本服务器程序与IPv4相关。图11.9的服务器程序版本与协议无关，它调用了getaddrinfo函数。
- 本服务器程序一次只能处理一个客户，如果多个客户连接同时到达，系统内核在最大数目的限制下把它们排入队列，然后每次返回一个给accept函数。这个服务器只需调用两个库函数time和ctime，其速度很快。但是，如果服务器用很多时间处理各个客户(如几秒或一分钟)，我们就必须以某种方式重叠对各个客户的服务。图1.9服务器称为迭代服务器(iterative server)，因为对每个客户它都迭代执行一次。并发服务器(concurrent server)程序有多种编写技术，它同时能处理多个客户。最简单的技术是调用Unix的fork函数(见4.7节)，为每个客户派生一个子进程。其他技术包括使用线程代替fork(见23.4节)，或在服务器启动时预先fork一定数量的子进程(见27.6节)。
- 如果在shell命令行下启动如本例的服务器，我们也许要它运行很长的时间，因为服务器通常在系统启动后就一直运行。这要求我们增加服务器程序的代码，使它作为

Unix 守护进程(daemon,即能在后台运行,不跟任何终端关联的进程)运行,见 12.4 节。

1.6 书中客户-服务器程序例子索引表

两个贯穿全书的客户-服务器程序例子阐述了网络编程所用的各种技术。

- 时间/日期客户-服务器程序例子(开始于图 1.5、1.6 和 1.9)。
- 回射客户-服务器程序例子(开始于第 5 章)。

为了提供本书所涵盖的不同主题的索引,以下 4 个表汇总了我们开发的程序及它们的源代码所在的图号。图 1.10 列出了本书开发的时间/日期客户程序的不同版本,其中的两个我们已叙述过。图 1.11 列出了时间/日期服务器程序的不同版本。图 1.12 列出了回射客户程序的不同版本,图 1.13 列出了回射服务器程序的不同版本。

图	页码	描述
1.5	4	TCP/IPv4,协议相关
1.6	8	TCP/IPv6 协议相关
9.8	218	TCP/IPv4,协议相关,调用 gethostbyname 和 getservbyname
11.7	244	TCP,协议无关,调用 getaddrinfo 和 tcp_connect
11.12	252	UDP,协议无关,调用 getaddrinfo 和 udp_client
15.11	350	TCP,使用非阻塞 connect
28.13	664	TCP/IPv4,XTI,协议相关
29.7	679	TCP,XTI,协议无关,调用 netdir_getbyname 和 tcp_connect
31.3	703	UDP,XTI,协议无关,调用 netdir_getbyname 和 udp_client
31.4	705	UDP,XTI,协议无关,接收异步错
31.7	709	UDP,XTI,协议无关,分片读数据报
33.8	732	TCP,协议相关,用 TPI 代替套接口或 XTI
E.1	797	TCP,协议相关,生成 SIGPIPE
E.5	800	TCP,协议相关,输出套接口的接收缓冲区大小和 MSS
E.13	809	TCP,协议相关,允许主机名(gethostbyname)或者 IP 地址
E.14	810	TCP,协议无关,允许主机名(gethostbyname)

图 1.10 本书开发的时间/日期客户程序的不同版本

图	页码	描述
1.9	11	TCP/IPv4,协议相关
11.9	247	TCP,协议无关,调用 getaddrinfo 和 tcp_listen
11.10	249	TCP,协议无关,调用 getaddrinfo 和 tcp_listen
11.15	255	UDP,协议无关,调用 getaddrinfo 和 udp_server
12.5	290	TCP,协议无关,作为独自的守护进程运行
12.12	295	TCP,协议无关,从 inetd 守护进程派生
30.5	688	TCP,XTI,协议无关,调用 netdir_getbyname 和 tcp_listen
31.6	708	UDP,XTI,协议无关,调用 netdir_getbyname 和 udp_server

图 1.11 本书开发的时间/日期服务器程序的不同版本

图	页码	描述
5.4	96	TCP/IPv4,协议相关
6.9	133	TCP,使用 select
6.13	138	TCP,使用 select 在批处理模式下工作
8.7	184	UDP/IPv4,协议相关
8.9	187	UDP,核实服务器的地址
8.17	194	UDP,调用 connect 获取异步错
13.2	300	UDP,使用 SIGALRM 信号在读服务器的应答时启动超时
13.4	302	UDP,使用 select 函数在读服务器的应答时启动超时
13.5	303	UDP,使用 SO_RCVTIMEO 套接口选项在读服务器的应答时启动超时
14.4	324	Unix 域字节流,协议相关
14.6	325	Unix 域数据报,协议相关
15.3	341	TCP,使用非阻塞 I/O
15.9	347	TCP,使用两个进程(fork)
15.21	361	TCP,建立连接,然后发送 RST
18.5	407	UDP,具有竞争状态的广播
18.6	410	UDP,具有竞争状态的广播
18.7	412	UDP,通过使用 pselect 消除了竞争状态的广播
18.9	413	UDP,通过使用 sigsetjmp 和 siglongjmp 消除了竞争状态的广播
18.10	416	UDP,通过从信号处理程序使用 IPC 消除了竞争状态的广播
20.6	466	UDP,使用超时、重传和序列号实现可靠性
21.14	497	TCP,使用带外数据,对服务器心搏测试
23.2	514	TCP,使用两个线程
24.6	545	TCP/IPv4,指定源路径

图 1.12 本书开发的回射客户程序的不同版本

图	页码	描述
5.2	95	TCP/IPv4,协议相关
5.12	108	TCP/IPv4,协议相关,收拾终止了的子进程
6.21	141	TCP/IPv4,协议相关,使用 select,单个进程处理所有客户
6.25	147	TCP/IPv4,协议相关,使用 poll,单个进程处理所有客户
8.3	182	UDP/IPv4,协议相关
8.24	200	TCP 和 UDP/IPv4,协议相关,使用 select
13.14	313	TCP,使用标准 I/O 库
14.3	323	Unix 域字节流,协议相关
14.5	324	Unix 域数据报,协议相关
14.15	335	Unix 域字节流,从客户端传递凭证
20.4	459	UDP,接收目的地址和接收接口,截取数据报
20.15	474	UDP,捆绑所有接口地址
21.15	499	TCP,使用带外数据对客户心搏测试
22.4	505	UDP,使用信号驱动的 I/O
23.3	515	TCP,每个客户一个线程
23.4	517	TCP 每个客户一个线程,传递可移植的参数

（续）

图	页码	描述
24.6	545	TCP/IPv4，输出接收到的源路径
25.30	586	UDP，使用 icmpd 接收异步错
E.17	821	UDP，捆绑所有接口地址

图 1.13 本书开发的回射服务器程序的不同版本

1.7 OSI 模型

描述网络中各协议层的一般方法是国际标准化组织(ISO)的计算机通信开放系统互连(open systems interconnection，OSI)模型。这是一个七层模型，如图 1.14 所示，图中同时给出了与网际协议族的近似映射。

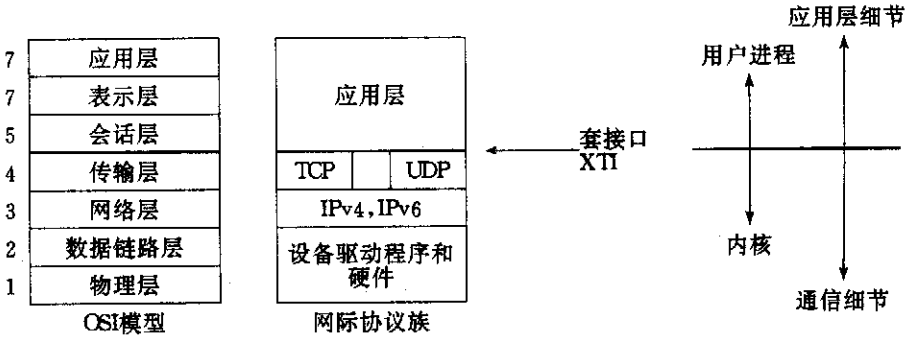


图 1.14 OSI 层和网际协议族

我们认为 OSI 模型的底下两层是随系统提供的设备驱动程序和网络硬件。除需知道数据链路的某些特性如 1500 字节的以太网 MTU(我们将在 2.9 节论述)外，我们不必关心这两层。

网络层由 IPv4 和 IPv6 协议处理，附录 A 描述这两个协议。传输层可以选择 TCP 或 UDP，我们将在第 2 章描述它们。图 1.14 中的网际协议族，在 TCP 与 UDP 之间留有一个间隙，指出应用程序可以绕过传输层而直接使用 IPv4 或 IPv6。这称为原始套接口(raw socket)，我们将在第 25 章阐述。

OSI 模型的上面三层合并成一层，称为应用层。这就是 Web 客户(浏览器)、Telnet 客户、Web 服务器、FTP 服务器或其他应用进程所在的层。对于网际协议，OSI 模型的上三层协议没什么区别。

本书所描述的两个编程接口(套接口和 XTI)是上三层(应用层)到传输层的接口。本书的焦点是：如何使用套接口或 XTI 编写使用 TCP 或 UDP 的网络应用程序。我们已提到原始套接口，在第 26 章我们将看到，甚至彻底绕过 IP 层读-写数据链路层的帧都是可能的。

为什么套接口和 XTI 都提供 OSI 模型上三层与传输层的接口？有两条理由，它们已标在图 1.14 的右侧。第一条理由是上三层处理应用程序(如，FTP，Telnet 或 HTTP)的细节，不大知道通信细节；下四层则不大知道应用程序，但能处理所有的通信细节：发送数据，等待

确认，给无序到达的数据排序，计算与验证校验和等等。第二条理由是上三层通常形成用户进程，而下四层通常作为操作系统内核的一部分提供。Unix与其他现代操作系统都提供分隔用户进程与内核的机制。因此，在第4层和第5层之间的接口很自然成了应用程序编程接口(API)。

1.8 BSD网络支持历史

套接口API起源于1983年发行的4.2BSD操作系统。图1.15描述了各种BSD发行版本的发展，主要注意TCP/IP的发展。当OSI协议于1990年进入BSD内核时，4.3BSD Reno发行版本对套接口API做了少量的改动。

图中从上到下是4.2BSD到4.4BSD的各种发行版本，它们源自Berkeley计算机系统研究组(CSRG)，不过要求获取者拥有Unix的源代码许可权。然而其中的所有网络支持代码，包括内核支持部分(如TCP/IP域协议栈、Unix域协议栈及套接口API)和应用程序(如Telnet和FTP的客户和服务器程序)都是独立于源自AT&T的Unix代码开发的。于是从1989年起，Berkeley开始提供了第一个BSD网络支持版本，它含有所有的网络支持代码及不受Unix源代码许可权约束的其他各种BSD系统软件。这些网络支持版本是可公开获取的，最终因特网上任何人都可通过匿名FTP获取。

源自Berkeley的最终版本是1994年的4.4BSD-Lite和1995年的4.4BSD-Lite2。这两个版本是其他系统的基础，包括BSD/OS、FreeBSD、NetBSD和OpenBSD，它们都还在开发和完善。有关各种BSD版本和各种Unix系统历史的详情见[Mckusick et al. 1996]的第1章。

很多Unix系统起源于某个版本的BSD网络支持代码(包括套接口API)，我们称这些实现为源自Berkeley的实现(Berkeley-derived implementations)。很多Unix商业版本基于系统V版本4(SVR4)，其中一些系统使用源自Berkeley的网络支持代码(如UnixWare 2.x)，而其他SVR4系统的网络支持代码则是独立开发的(如Solaris 2.x)。还有Linux系统：一种著名的免费可得的Unix实现版本，它不属于源自Berkeley的系列，其网络支持代码和套接口API是从头开始开发的。

1.9 测试用网络及主机

图1.16描述了本书例子所用的各种网络和主机。对每台主机我们都说明了它的操作系统和硬件类型(因为有些操作系统可在不同的硬件类型上运行)。各个框内的主机名整本书都用到。

顶部子网地址分别为206.62.226.32/27和206.62.226.64/27的两个以太网上的主机都在kohala.com域内。底部子网地址为140.252.1.0/24的以太网上的主机都在tuc.noao.edu域内，它们由国家光学天文台运行。符号/27和/24指出从地址的最左位开始的用于标识网络和子网的连续位数。图A.4和图A.5详细说明这两种IPv4地址，图A.4还说明当前使用的用于指定子网边界的/n记法。

图1.16中圆矩形框内的节点充当路由器，矩形框内的节点则是纯粹的主机。全书都使

用这个约定,因为有时候区分主机和路由器相当重要。

注意:Sun 操作系统的真实名字是 SunOS 5.x 而不是 Solaris 2.x,但大家习惯称它为 Solaris。

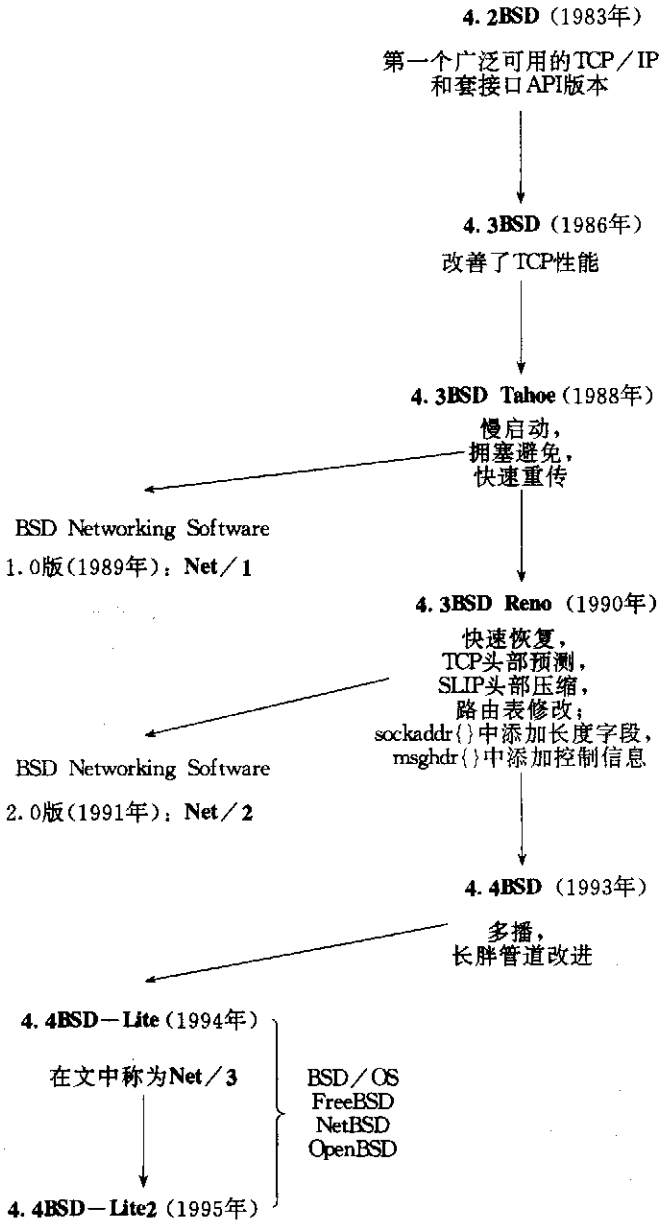


图 1.15 各种 BSD 版本的历史

网络拓扑发现

图 1.16 是全书的例子所用的网络拓扑,但是,你在自己的网络上运行这些例子和做习题时,你得知道自己的网络拓扑。虽然不存在有关网络配置和管理的标准,但是大多数 Unix 系统都有两个命令可用来了解网络细节:netstat 和 ifconfig。下面举一些不同于图 1.16 系统

的例子。通过阅读系统上这些命令的手册页面，你就可以获悉有关它们的输出信息的详情。⑦ 你还必须注意，有些厂商把这些命令存放于/sbin或/usr/sbin等管理目录，而不是普通目录/usr/bin，因此这些目录可能不在你通常的shell搜索路径(PATH)中。

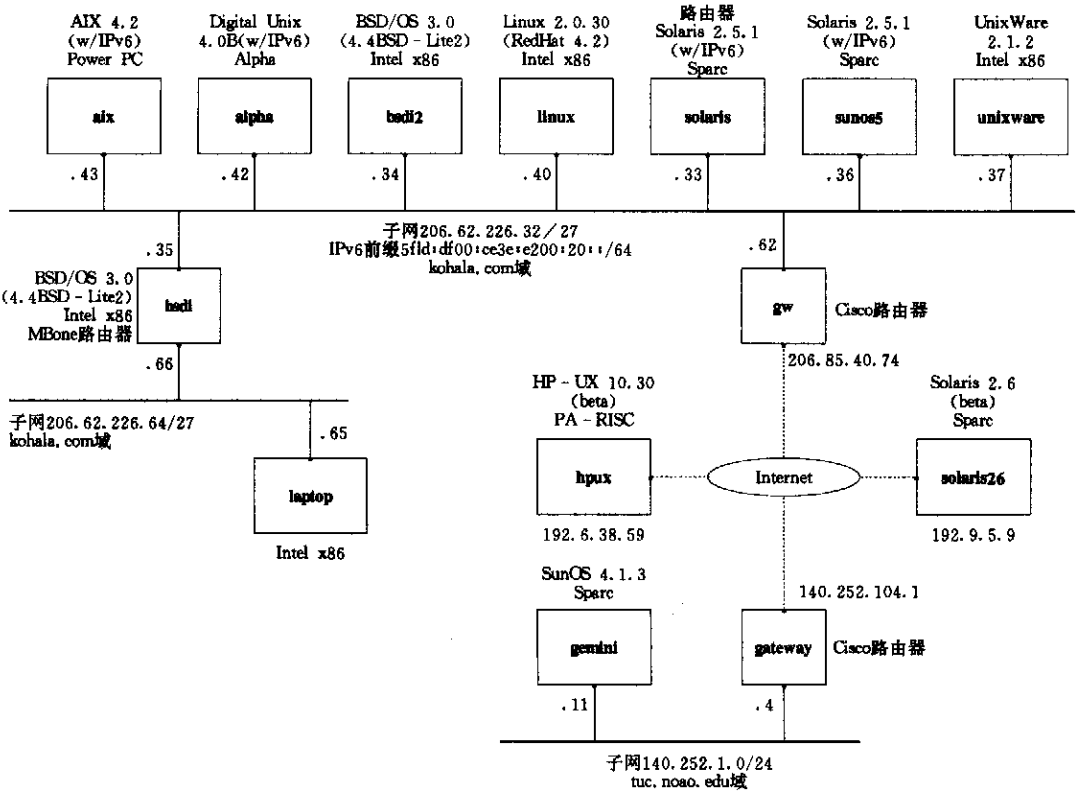


图1.16 本书中例子所用网络和主机

1. netstat -i提供接口信息，也可用标志-n输出数值地址而不是试图找出网络名字。下面例子显示接口和它们的名字。

```
linux % netstat -ni
Kernel Interface table
Iface MTU Met RX-OK RX-ERR RX-DRP RX-OVR TX-OK TX-ERR TX-DRP TX-OVR Flags
lo    3584  0     32      0      0      0     32      0      0      0 BLRU
eth0 1500   0 483929      0      0      0 449881      0      0      0 BRU
```

其中回馈(loopback)接口称为lo，以太网接口称为eth0。下面例子显示支持IPv6的一台主机的信息。

```
alpha % netstat -ni
Name  Mtu   Network     Address              Ipkts Ierrs Opkts Oerrs  Coll
ln0   1500  <Link>      08:00:2b:37:64:26    11220     0  4893     0     4
ln0   1500  DLI         none                 11220     0  4893     0     4
```

⑦ 译者注：手册页面(manual pages或man pages)是所有Unix系统都提供的使用man命令查看到的有关命令、函数和文件等的帮助信息。某个条目的手册页面就是以该条目为命令行参数执行man的输出。

```
ln0    1500  206.62.226.   206.62.226.42          11220   0   4893   0   4
ln0    1500  IPv6 FE80::800:2B37:6426             11220   0   4893   0   4
ln0    1500  IPv6 5F1B:DF00:CE3E:E200:20:800:2B37:6426  11220  0  4893  0  4
lo0    1536  <Link>       Link#3                  12432   0  12432   0   0
lo0    1536  127          127.0.0.1               12432   0  12432   0   0
lo0    1536  IPv6         ::1                     12432   0  12432   0   0
tun0   576   <Link>       Link#4                      0   0      0   0   0
tun0   576   IPv6         ::206.62.226.42             0   0      0   0   0
```

2. netstat -r 输出路由表，这是另一种确定接口的方法。一般使用-n 标志输出数值地址。本命令还输出缺省路由器的 IP 地址。

```
aix % netstat -rn
Routing tables
Destination          Gateway           Flags   Refs   Use    MTU    Netif Expire

Route tree for Protocol Family 2 (Internet):
default              206.62.226.62     UG      0      0      -      en0
127/8                127.0.0.1         U       0      0      -      lo0
206.62.226.32/27     206.62.226.43     U       4    475      -      en0
Route tree for Protocol Family 24 (Internet v6):
::/96                0.0.0.0           UC      0      0   1480      sit0   - =>
default      fe80::2:0:800:2078:e3e3   UG      0      0      -      en0
::1                  ::1               UH      0      0  16896      lo0
5f1b:df00:ce3e:e200:20::/80
                     link#2            UC      0      0   1500      en0    -
fe80::/16            link#2            UC      0      0   1500      en0    -
fe80::2:0:800:2078:e3e3
                     link#2            UHDL    1      0   1500      en0    -
ff01::/16            ::1               U       0      0      -      lo0
ff02::/16    fe80::800:5afc:2b36       U       1      3   1500      en0
ff11::/16            ::1               U       0      0      -      lo0
ff12::/16    fe80::800:5afc:2b36       U       0      0   1500      en0
```

（为对齐输出字段，我们将一些长行折了行。）

3. 给定接口名字，执行 ifconfig 可获得每个接口的详细信息。

```
linux % ifconfig eth0
eth0 link encap: 10Mbps Ethernet HWaddr 00:A0:24:9C:43:34
     inet addr:206.62.226.40 Bcast:206.62.226.63 Mask:255.255.255.224
     UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
     RX Packets:484461 errors:0 dropped:0 overruns:0
     TX Packets:450113 errors:0 dropped:0 overruns:0
     Interrupt:10 Base address:0x300
```

这里输出了 IP 地址、子网掩码和广播地址。MULTICAST 标志通常指示主机支持多播。

```
alpha % ifconfig ln0
ln0: flags=c63<UP,BROADCAST,NOTRAILERS,RUNNING,MULTICAST,STMPLEX>
     inet 206.62.226.42 netmask fffffffe0 broadcast 206.62.226.63 ipmtu 1500
```

有些 ifconfig 的实现还提供-a 标志，用于输出所有已配置接口的信息。

4. 查找本地网络中多个主机 IP 地址的方法之一是针对广播地址执行 ping 命令。

```
bsdi % ping 206.62.226.63
```

```
PING 206.62.226.63 (206.62.226.63) : 56 data bytes
64 bytes from 206.62.226.35: icmp_seq=0 ttl=255 time=0.316 ms
64 bytes from 206.62.226.40: icmp_seq=0 ttl=64 time=1.369 ms (DUP!)
64 bytes from 206.62.226.34: icmp_seq=0 ttl=255 time=1.822 ms (DUP!)
64 bytes from 206.62.226.42: icmp_seq=0 ttl=64 time=2.27 ms (DUP!)
64 bytes from 206.62.226.37: icmp_seq=0 ttl=64 time=2.717 ms (DUP!)
64 bytes from 206.62.226.33: icmp_seq=0 ttl=255 time=3.281 ms (DUP!)
64 bytes from 206.62.226.62: icmp_seq=0 ttl=255 time=3.731 ms (DUP!)
^?                          敲入中断键(DEL)
--- 206.62.226.63 Ping statistics ---
1 packets transmitted, 1 packets received, +6 duplicates, 0% packet loss round-trip min/
avg/max = 0.316/2.215/3.731 ms
```

1.10 Unix标准

有关Unix标准化的大多数活动是由Posix和Open Group做的。

POSIX

Posix(Portable Operating System Interface)是"可移植操作系统接口"的首字母缩写。它并不是一个单一标准,而是由电气与电子工程师学会即IEEE开发的标准族。Posix也是被ISO和IEC(国际电工委员会)采纳的国际标准(这两个组织合称为ISO/IEC)。

第一个Posix标准是IEEE Std 1003.1—1988(共317页),它说明进入类Unix内核的C语言接口,涉及下述领域:进程原语(fork、exec、信号及定时器)、进程环境(用户ID、进程组)、文件与目录(所有I/O函数)、终端I/O、系统数据库(保密字文件和用户组文件),以及tar和cpio归档格式。

> 第一个Posix标准在1986年是称为"IEEEIX"的试用版,Posix的名字是由Richard Stallman建议使用的。

这个标准在1990年被更新成IEEE Std 1003.1—1990(共356页),它也是国际标准ISO/IEC 9945—1:1990。从1988版本到1990版本只做了少量的修改。新添的副标题为"Part 1: System Application Program Interface (API) [C Language]",指示本标准为C语言API。

下一个Posix标准是IEEE Std 1003.2—1992,它的副标题为"Part 2: Shell and Utilities"。它出版成两卷,共约1300页。这一部分定义了shell(基于系统V的Bourne Shell)和大约100个实用程序(从Shell启动执行的程序,包括awk、basename、vi和yacc等等)。本书称这个标准为Posix.2。

再下一个Posix标准是IEEE Std 1003.1b—1993,先前称IEEE P1003.4。这是对1003.1—1990标准的更新,添加了由P1003.4工作组开发的实时扩展。1003.1b-1993标准与1990标准相比新增的条目有:文件同步、异步I/O、信号量、存储管理(mmap和共享内存)、执行调度、时钟与定时器及消息队列。1003.1b—1993标准共590页。

再下一个Posix标准是IEEE Std 1003.1,1996年版[IEEE 1996],它包含1003.1—1990(基本API)、1003.1b—1993(实时扩展)、1003.1c—1995(pthreads)和1003.1i—1995(对1003.1b的技术性修正)。这个标准也称ISO/IEC 9945—1:1996,它加入了三章线程内容,共

743页。本书称这个标准为 Posix. 1。

> 743页中有四分之一多的篇幅是标题为“Rationale and Notes(原理与注解)”的附录。这些原理含有历史信息和某些特性必须加入或删除的理由,它们通常跟正式标准一样有教益。

这个标准的前言声称 ISO/IEC 9945由下面三个部分构成:

- 第1部分:系统应用程序接口(API)[C 语言]
- 第2部分:Shell 和实用程序
- 第3部分:系统管理(正在开发中)

最影响本书的 Posix 工作是 IEEE Std 1003. 1g:协议独立的接口(PII,Protocol Independent Interfaces),为 P1003. 1g 工作组的产品。这是网络 API 标准,定义有两个 API,称为 DNI(详细的网络接口,Detailed Network Interfaces)。

1. DNI/Socket,基于4. 4BSD 的套接口 API。
2. DNI/XTI,基于 X/Open XPG4规范。

这个标准的工作起始于80年代后期,由 P1003. 12工作组(以后改名为 P1003. 1g)承担,本书编写时它尚未完成(不过已接近)。草案6. 4(1996年5月)是第一个草案,它获得无记名投票75%以上的票数通过。草案6. 6(1997年3月)看来是最终草案[IEEE 1997a]。1998年或1999年的某时候,新版本 IEEE Std 1003. 1将被印刷出来,它将含有 P1003. 1g 标准。

即使 P1003. 1g 标准尚未正式完成,本书也尽可能使用来自这个标准的草案6. 6中的特性。本书称这一草案为 Posix. 1g。例如,connect 函数的第三个参数将被说明成 socklen_t 数据类型尽管它直到 Posix. 1g 才定义。类似地,我们叙述 Posix. 1g 中才有的 sockatmark 函数(21. 3节),并提供它的使用 ioctl 函数实现的版本。对 Unix 域的套接口,我们还使用 Posix. 1g 协议值 AF_LOCAL 代替 AF_UNIX。当前实际与 Posix. 1g 的差异在本书中将被注解出来。虽然今天还没有厂商支持 Posix. 1g(因为它不是最终版本),但是当这个标准完成时,厂商都会支持的。

Posix 标准化工作还将继续,这是任何试图涵盖它的书都在跟踪的目标。从 http://www. pasc. org/standing/sdll. html 可获得各种 Posix 标准的当前状况。

Open Group

Open Group 是由 X/Open 公司(1984年成立)和开放软件基金会(OSF,1988年成立)于1996年合并成的组织。它是厂商、工业界最终用户、政府和学术机构的国际组织。

X/Open 公司于1989年出版了“X/Open Portability Guide(X/Open 移植性指南)”第3期(XPG3)。第4期出版于1992年,其第2版出版于1994年。这个最终版本称为“Spec 1170”,其中魔数1170是系统接口(926个)、头文件(70个)和命令(174个)的数目总和。这个规范集的最终名字是“X/Open Single Unix Specification(X/Open 独立 Unix 规范)”,也称为“Unix 95”。

1997年3月独立 Unix 规范第2版发表,符合这个规范的产品称为“Unix 98”。本书就称这个规范为“Unix 98”。Unix 98的接口数目从1170个扩充到1434个,而用于工作站的数目则跳到3030个,因为它含有 CDE(公共桌面环境,Common Desktop Environment),它反过来又要

求有X Windows系统和Motif的用户接口。其详情见[Josey 1997]和http://www.opengroup.org/public/tech/unix/version2。

我们感兴趣的是Unix 98的网络服务部分，它们在[Open Group 1997]里定义，包括套接口API和XTI API。这个规范几乎与Posix.1g的草案6.6相同。

> 不幸的是，X/Open称它们的网络标准为XNS：X/Open Networking Services。定义Unix 98套接口和XTI的这个文档的版本称为"XNS Issue 5"。在网络界，XNS是Xerox Network Systems体系结构的缩写。我们避免使用XNS，就称这个X/Open文档为Unix 98网络API标准。

因特网工程任务攻坚组

IETF(因特网工程任务攻坚组，Internet Engineering Task Force)是一个由网络设计者、操作员、厂商和研究者联合组成的开放的国际团体，他们关心因特网体系结构的发展和它的顺利运作。它向任何有兴趣的个人开放。

因特网标准处理在RFC 2026[Bradner 1996]中说明。因特网标准一般处理协议事务而不是编程API。不过仍有两个RFC([Gilligan et al. 1997]和[Stevens and Thomas 1997])说明IP版本6的套接口API。它们是信息性RFC而非标准，它们产生的目的是为了加速部署由数家从事早期IPv6工作的厂商所生产的可移植应用程序。相反，标准化主体工作将花费更长时间。不过也许过一些时候，IPv6 API将被更为正式地标准化。

Unix版本和移植性

当今大多数Unix系统符合Posix.1和Posix.2的某个版本。我们这里用限定词"某个"是因为Posix也在更新(如，1993年对实时的扩充和1996年加入pthreads)，它要花费厂商一至两年时间去加上最近的更新。

历史上多数Unix系统或者源自Berkeley，或者源自系统V，不过这些差别在慢慢消失，因为大多数厂商已开始采用Posix标准。系统管理的差别还将存在，这个领域目前还没有Posix标准。

本书的焦点是即将到来的Posix.1g标准，其中又以套接口API为主。任何时候我们都尽可能使用Posix函数。

1.11 64位体系结构

90年代中期到末期倾向于64位体系结构和64位软件。理由之一是进程内部可使用更大的编址长度(即64位指针)，以允许进行大内存寻址(超过2^{32}字节)。已有的32位Unix系统上的一般编程模型称为ILP32模型，表示整数(I)、长整数(L)和指针(P)都占用32位；64位Unix系统上最为流行的编程模型称为LP64模型，它表示长整数(L)和指针(P)都占用64位。图1.17是这两种模型的比较。

数据类型	ILP32模型	LP64模型
char	8	8
short	16	16
int	32	32
long	32	64
pointer	32	64

图1.17 ILP32和LP64模型对不同数据类型所占用的位数的比较

从编程角度看,LP64模型意味着我们不能假设一个指针能存放到一个整数中。我们还必须考虑LP64模型对已有的API的影响。

ANSI C创造了size_t的数据类型,它用于诸如malloc的参数(分配的字节数)、read和write的第三个参数(读或写的字节数)等地方。在32位系统中size_t是32位值,但在64位系统中它必须是64位的值,以发挥大寻址模型的优点。这意味着64位系统中也许含有把size_t定义为unsignecl long的typedef语句。网络API的问题是Posix.1g的某些草案说明,存放套接口地址结构大小的函数参数具有size_t的数据类型(如bind和connect的第三个参数)。同样,某些XTI结构也含有数据类型为long的成员(如,t_info和t_opthdr结构)。当Unix系统从ILP32改变成LP64模型时,两者都将从32位的值变成64位的值。上述两例实际并不需要使用64位的数据类型:套接口地址结构的长度最多也不过是几百个字节,给XTI结构成员使用long数据类型也是错误的。

我们将见到的是创造新的数据类型来处理这些情况。套接口API使用socklen_t数据类型作为套接口地址结构的长度。XTI则使用t_scalar_t和t_uscalar_t数据类型。不把这些值由32位改为64位的理由是,这将使那些已在32位系统编译的应用程序的二进制代码兼容于新的64位系统。

1.12 小 结

图1.5虽然简单,但它说明了一个完整的TCP客户程序,它从一指定的服务器读取当前的时间和日期;而图1.9则给出这个服务器程序的一个完整版本。这两个例子引入了许多本书其他部分将要扩展的概念和术语。

我们的客户程序依赖于IPv4,为此我们把它修改成使用IPv6。但这给了我们另一个与协议相关的程序。第11章中我们将开发一些函数,它们可用来编写协议无关的代码,这在因特网开始使用IPv6时是非常重要。

纵贯本书,我们将使用1.4节的包裹函数以减少代码的长度,同时又保证测试每个函数调用,检查是否返回错误。我们的包裹函数都以一个大写字母开头。

IEEE Posix标准——Posix.1定义了Unix的基本C接口,Posix.2定义了标准命令,Posix.1g定义了网络API——是多数厂商倾向的标准。然而商业标准也正在迅速吸收并扩充Posix,著名的有Open Group的Unix标准,例如Unix 98。

那些对Unix网络支持历史感兴趣的读者可参阅叙述Unix历史的[Salus 1994]和叙述TCP/IP及因特网历史的[Salus 1995]。

1.13 习 题

1.1 按1.9节末尾的步骤去找出你自己的网络拓扑的信息。

1.2 获取本书例子的源代码(见前言),编译并测试图1.5的TCP时间/日期客户程序。运行这个程序若干次,每次以不同IP地址作为命令行参数。

1.3 在图1.5中,把socket的第一参数改为9999。编译并运行这个程序。结果如何?找出对应于所输出出错消息的errno值。你能找到关于这个错误的更详细信息吗?

1.4 修改图1.5的while循环,加入一个计数器,累计read返回大于零值的次数。在终止前输出这个值。编译并运行你的新客户程序。

1.5 按下述步骤修改图1.9程序。首先,把赋于sin_port的端口号从13改为9999。其次,把write的单一调用改为循环调用,每次写出结果字符串的一个字符。编译修改后的服务器程序并在后台启动执行。接下去修改前面习题的客户程序(它在终止前输出计数值),把赋于sin_port的端口号从13改为9999。启动客户,指定运行修改后服务器程序的主机的IP地址作为其命令行参数。客户程序计数器的输出值是多少?如果可能,在不同主机上运行客户与服务器。

第 2 章　传输层:TCP 和 UDP

2.1　概　述

本章提供本书例子所用 TCP/IP 协议的概貌。我们的目的是从网络编程角度提供足够的细节以理解如何使用这些协议,同时提供有关这些协议的实际设计、实现及历史的具体描述的参考点。

本章的焦点是传输层即 TCP 和 UDP 协议,绝大多数的客户-服务器应用程序都使用 TCP 或 UDP。这两个协议转而使用网络层协议 IP:IP 版本 4(IPv4)或 IP 版本 6(IPv6)。尽管可以绕过传输层直接使用 IPv4 或 IPv6,但这种技术(称为原始套接口)较少使用。因此,我们把 IPv4 和 IPv6 以及 ICMPv4 和 ICMPv6 的详细描述安排在附录 A。

UDP 是一种简单的,不可靠的数据报协议,而 TCP 是一种精致的、可靠的字节流协议。我们必须了解由这两个传输层提供给应用进程的服务,这样才能弄清这些协议处理什么,我们的应用进程又需要处理什么。

TCP 的某些特性一旦理解,我们就能更容易地编写健壮的客户和服务器程序,更容易地使用诸如 netstat 等工具来调试我们的客户和服务器程序。本章将阐述下述主题:TCP 的三路握手、TCP 连接终止序列、TCP 的 TIME_WAIT 状态、套接口层的 TCP 和 UDP 缓冲机制等等。

2.2　总　图

虽然称为"TCP/IP"协议族,但是本协议族还有许多其他成员。图 2.1 展示了这些协议的概貌。

在这个图中,我们展示了 IPv4 和 IPv6。从右向左观察这个图,最右边的 4 个应用程序使用 IPv6,这涉及到第 3 章中的 AF_INET6 常值和 sockaddr_in6 结构。另外的 5 个应用程序使用 IPv4。

最左边的应用程序(tcpdump)直接使用 BPF(BSD 分组过滤器)或 DLPI(数据链路提供者接口)同数据链路层进行通信。右边 9 个应用程序的下面用虚线标记出 API,它通常是套接口或 XTI。使用 BPF 或 DLPI 的接口不用套接口或 XTI。

> 这种情况的一种例外将在第 25 章详细描述:Linux 使用一种称为 SOCK_PACKET 的特殊套接口类型提供数据链路层的访问。

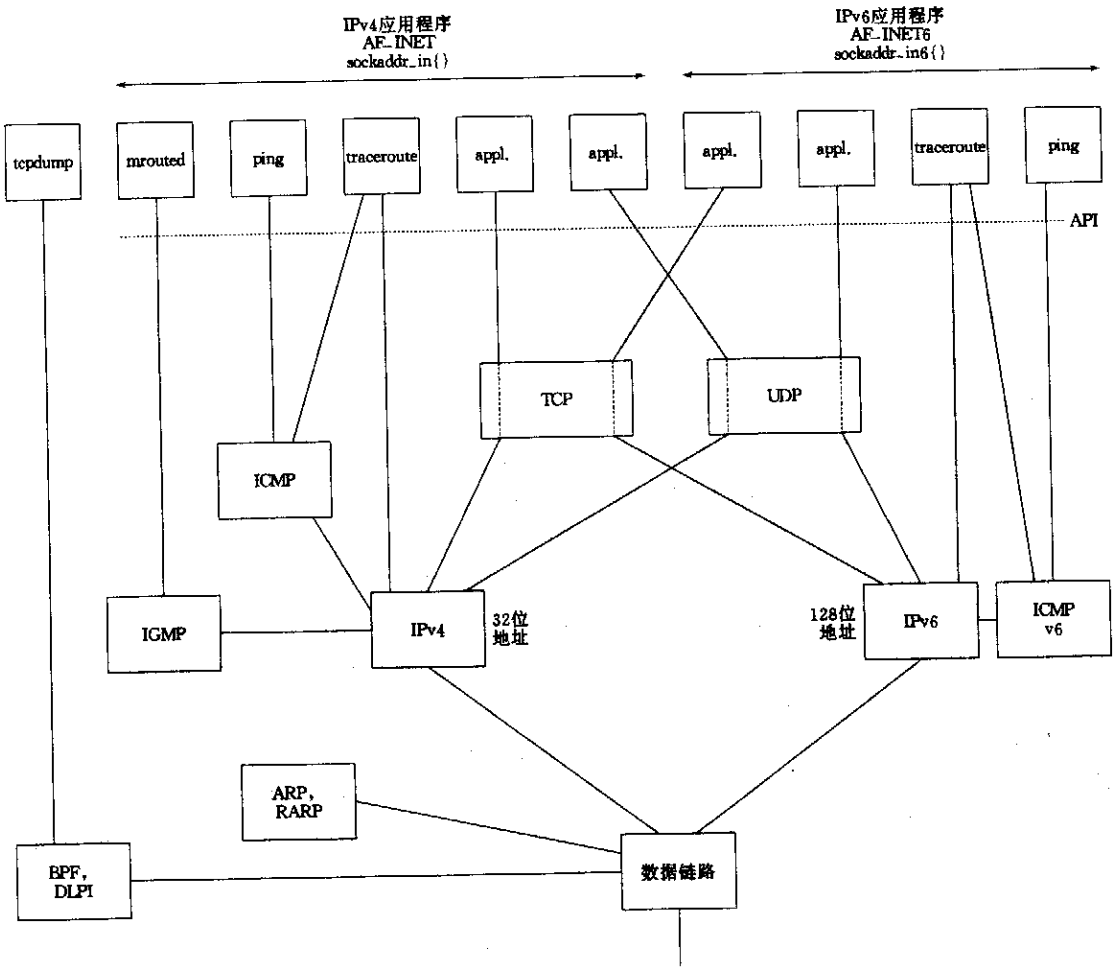


图 2.1 TCP/IP 协议概貌

在图 2.1 中,我们还注意到 traceroute 程序使用两种套接口:IP 套接口和 ICMP 套接口。在第 25 章我们将开发 IPv4 和 IPv6 两种版本的应用程序 ping 和 traceroute。

下面将阐述图 2.1 中的每一种协议。

IPv4 网际协议,版本 4。IPv4(我们通常就称之为 IP)自 80 年代早期以来一直是网际协议族的主力协议。它使用 32 位的地址(A.4 节)。IPv4 给 TCP、UDP、ICMP 和 IGMP 提供递送分组的服务。

IPv6 网际协议,版本 6。IPv6 在 90 年代中期设计出来,用以替代 IPv4。主要变化是使用 128 位的大地址(A.5 节)以处理 90 年代因特网爆发性的增长。IPv6 给 TCP、UDP 和 ICMPv6 提供递送分组的服务。

当无需区别 IPv4 和 IPv6 时,我们经常使用 IP 这个形容词,如 IP 层、IP 地址等等。

TCP	传输控制协议。TCP 是一种面向连接的协议。它给用户进程提供可靠的全双工的字节流。TCP 套接口是字节流套接口(stream socket)的一种。TCP 关心确认、超时和重传等具体细节。大多数因特网应用程序使用 TCP。注意，TCP 可以使用 IPv4 或 IPv6。
UDP	用户数据报协议。UDP 是一种无连接协议。UDP 套接口是数据报套接口(datagram socket)的一种。与 TCP 不同，UDP 不能保证每一 UDP 数据报可以到达目的地。与 TCP 相同的是，UDP 也可以使用 IPv4 或 IPv6。
ICMP	网际控制消息协议。ICMP 处理路由器和主机间的错误和控制消息。这些消息一般由 TCP/IP 网络软件自身(而不是用户进程)产生和处理，不过图中展示的 Ping 程序也使用 ICMP。有时我们称这个协议为 ICMPv4，用于区别 ICMPv6。
IGMP	网际组管理协议。IGMP 用于多播(第 19 章)，它在 IPv4 中是可选的。
ARP	地址解析协议。ARP 把 IPv4 地址映射到硬件地址(如以太网地址)。ARP 一般用于广播网络，如以太网、令牌环网和 FDDI，但不用于点对点网络。
RARP	反向地址解析协议。RARP 把硬件地址映射到 IPv4 地址。它有时用于无盘节点，如引导时的 X 终端。
ICMPv6	网际控制消息协议，版本 6。ICMPv6 综合了 ICMPv4、IGMP 和 ARP 的功能。
BPF	BSD 分组过滤器。它为进程提供访问链路层数据的接口。一般在源自 Berkeley 的内核中可以找到它。
DLPI	数据链路提供者接口。这是提供访问数据链路的接口，一般由 SVR4 提供。

所有网际协议由(多个)RFC(Request for Comments)定义。习题 2.1 的解答说明如何获得各个 RFC。

我们使用术语“IPv4/IPv6 主机”或“双栈主机”表示同时支持 IPv4 和 IPv6 的主机。

有关 TCP/IP 协议的其他细节见 TCPv1。TCP/IP 在 4.4BSD 中的实现见 TCPv2。

2.3　UDP：用户数据报协议

UDP 是一个简单的传输层协议，在 RFC768[Postel 1980]中有详细描述。应用进程写一数据报到 UDP 套接口，由它封装(encapsulating)成 IPv4 或 IPv6 数据报，然后发送到目的地。但是，UDP 并不能确保 UDP 数据报最终可到达目的地。

我们使用 UDP 进行网络编程所碰到的问题是缺乏可靠性。如果我们要确保一个数据报到达目的地，我们必须在应用程序里建立一大堆的特性：来自另一端的确认、超时、重传等等。

每个UDP数据报都有一定长度,我们可以认为一个数据报就是一个记录。如果数据报最终正确地到达目的地(即分组到达目的地且校验和正确),那么数据报的长度将传递给接收方的应用进程。我们已经提到TCP是一字节流协议,无记录边界(1.2节),这与UDP不同。

我们也称UDP提供无连接的(connectionless)服务,因为UDP客户与服务器不必存在长期的关系。例如,一个UDP客户可以创建一个套接口并发送一个数据报给一个服务器,然后立即用同一个套接口发送另一个数据报给另一个服务器。同样,一个UDP服务器可以用同一个UDP套接口从5个不同的客户一连串接收5个数据报。

2.4 TCP:传输控制协议

向应用进程提供的TCP服务与UDP服务不相同。(TCP在RFC793[Postel 1981c]中介绍。)首先,TCP提供客户与服务器的连接(connection)。一个TCP客户建立与一个给定服务器的连接,跨越连接与那个服务器交换数据,然后终止连接。

其次,TCP提供可靠性。当TCP向另一端发送数据时,它要求对方返回一个确认。如果确认没有收到,TCP自动重传数据并等待更长时间。在数次重传失败后,TCP才放弃。重传数据所花的总时间传统上是4~10分钟(与实现有关)。TCP含有用于动态估算客户到服务器往返所花时间RTT(round-trip time)的算法,因此它知道等待一确认需要多少时间。例如,RTT在一局域网上大约几毫秒,而跨过广域网则需数秒钟。另外,某时刻TCP可能测到客户到服务器的RTT为1秒钟,而过30秒后却测到同一连接的RTT为2秒钟,这是因为网络传输的拥挤情况是在不断变化的。

第三,TCP通过给所发送数据的每一个字节关联一个序列号进行排序。例如,假设一个应用进程写2048字节到一个TCP套接口,导致TCP发送2个分节,第1个分节所含数据的序列号为1~1024,第2个分节所含数据的序列号为1025~2048(分节是TCP传递给IP的数据单元)。如果这些分节非顺序到达,接收方的TCP将根据它们的序列号重新排序,再把结果数据传递给应用进程。如果TCP接收到重复的数据(譬如说对方认为一个分节已丢失并因而重传,而它并没有真正丢失,只是刚才网络通信过于拥挤),它也可以判定数据是重复的(根据序列号),从而把它丢弃掉。

> UDP提供不可靠的数据报传送。UDP本身不提供确认、序列号、RTT估算、超时及重传等机制。如果一个UDP数据报在网上被复制,两份拷贝就可能都递送到接收方的主机。同样,如果一个UDP客户发送两个数据报到同一个目的地,它们可能被网络重新排序,颠倒顺序后到达目的地。UDP应用程序必须处理这些情况,我们将在20.5节中详细介绍。

第四,TCP提供流量控制。TCP总是告诉对方它能够接收多少字节的数据,这称为通告窗口(advertised window)。任何时刻,这个窗口指出接收缓冲区中的可用空间,从而确保发送方发送的数据不会溢出接收缓冲区。窗口时刻动态地变化:当接收发送方的数据时,窗口大小减小,而当接收方应用进程从缓冲区中读取数据时,窗口大小增大。窗口的大小减小到0是有可能的:TCP的接收缓冲区满,它必须等待应用进程从这个缓冲区读取数据后才能再

接收从发送方来的数据。

> UDP 不提供流控制。UDP 按发送方的速率发送数据，不管接收方的缓冲区是否装得下，见 8.13 节。

最后，TCP 的连接是全双工的。这意味着在给定的连接上应用进程在任何时刻既可以发送也可以接收数据。因此，TCP 必须跟踪每个方向数据流的状态信息，如序列号和通告窗口的大小。

> UDP 可以是全双工的。

2.5　TCP 连接的建立和终止

为帮助大家理解 connect、accept 和 close 函数并使用 netstat 调试 TCP 应用程序，我们必须了解如何建立和终止 TCP 连接以及 TCP 的状态转换图。这是一个通过加深了解底层网络协议以帮助我们编写网络程序的例子。

三路握手

下述步骤建立一个 TCP 连接：

1. 服务器必须准备好接受外来的连接。这通过调用 socket、bind 和 listen 函数来完成，称为被动打开(passive open)。
2. 客户通过调用 connect 进行主动打开(active open)。这引起客户 TCP 发送一个 SYN 分节(表示同步)，它告诉服务器客户将在(待建立的)连接中发送的数据的初始序列号。一般情况下 SYN 分节不携带数据，它只含有一个 IP 头部、一个 TCP 头部及可能有的 TCP 选项。
3. 服务器必须确认客户的 SYN，同时自己也得发送一个 SYN 分节，它含有服务器将在同一连接中发送的数据的初始序列号。服务器以单个分节向客户发送 SYN 和对客户 SYN 的 ACK。
4. 客户必须确认服务器的 SYN。

连接建立过程至少需要交换三个分组，因此称之为 TCP 的三路握手(three-way handshake)。我们在图 2.2 中展示这三个分节。

图 2.2 给出的客户的初始序列号为 J，而服务器的初始序列号为 K。在 ACK 里的确认号为发送这个 ACK 的一方所期待的对方的下一个序列号。因为 SYN 只占一个字节的序列号空间，所以每一个 SYN 的 ACK 中的确认号都是相应的初始序列号加 1。类似地，每一个 FIN 的 ACK 中的确认号为 FIN 的序列号加 1。

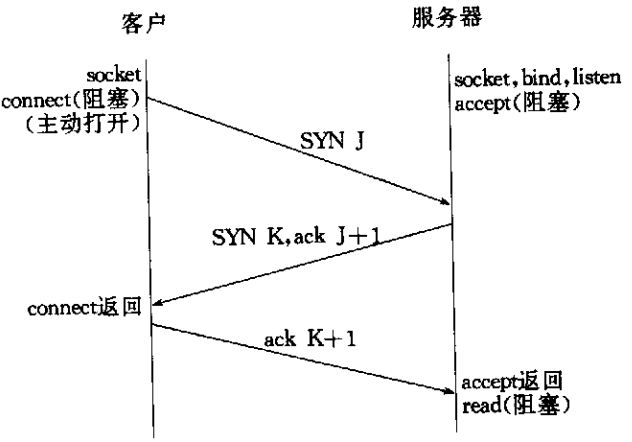


图2.2 TCP的三路握手

建立TCP连接的日常系统类比是电话系统[Nemeth 1997]。socket函数等同于有电话可用。bind用于告诉其他人你的电话号码,让他们可以向你打电话。listen是打开电话振铃,它使你可以听到一个外来的电话。connect要求你知道另一方的电话号码并拨打它。accept是被呼叫回电话。从accept返回客户的标识(即客户的IP地址和端口号)类似于让电话机的呼叫者ID功能部件显示打电话人的电话号码。然而有点不同的地方是:从accept返回客户的标识是在建立连接以后,而呼叫者ID功能部件显示打电话人的电话号码是在我们选择接或不接电话之前。如果使用域名系统(第9章),那么提供了一种类似于电话簿的服务。gethostbyname类似于在电话簿查找个人的电话号码。gethostbyaddr则类似于有一种电话簿按电话号码排序。

TCP选项

每一个SYN可以含有若干个TCP选项。通常使用的选项有:

- MSS选项。TCP发送的SYN中带有这个选项是通知对方它的最大分节大小MSS(maximum segment size),即它能接受的每个TCP分节中的最大数据量。我们会看到如何使用TCP_MAXSEG套接口选项获取与设置这个TCP选项(7.9节)。
- 窗口规模选项。TCP双方能够通知对方的最大窗口大小是65535,因为TCP头部相应的字段只占16位。但是高速连接(每秒45兆位或更快,见RFC1323[Jacobson,Braden,and Borman 1992])或长延迟的路径(卫星链路)要求有更大的窗口以尽可能获得最大的吞吐量。这个新选项指定TCP头部的广告窗口必须扩大(即左移)的位数(0~14),因此所提供的最大窗口几乎是1G字节(65535×2^{14})。两个端系统必须都支持这个选项,否则不具备扩大窗口规模能力。在7.5节我们将看到套接口选项SO_RCVBUF如何影响该选项。

为提供与不支持这个选项的较早实现间的互操作性,应用如下的规则。TCP可以作为主动打开的一部分内容随它的SYN发送该选项,但只有对方也随它的SYN发送该选项时,它才能扩大窗口的规模。类似地,服务器的TCP只有接收到随客户的SYN来的这个选项时,它才能发送该选项。本逻辑假定实现

会忽略它们不理解的选项。这是要求的,而且相当普遍,但不幸的是无法保证所有实现都是这样。

- 时间戳选项。这个选项对高速连接是必要的,它可以防止失而复得的分组可能造成的数据损坏。[①] 因为它是新选项,所以其处理类似于窗口规模选项。作为网络编程人员,对于这个选项我们没什么可担心。

MSS 选项在多数实现中都支持,而窗口规模和时间戳选项则是新的。后两个选项有时也称为"RFC 1323 选项",因为 RFC1323[Jacobson,Braden,and Borman 1992]说明了它们。它们也称为"长胖管道"选项,因为一个高带宽或长延迟的网络被称为"长胖管道(long fat pipe)"。TCPv1 的第 24 章对这些新选项有详细的描述。

TCP 连接终止

TCP 用三个分节建立一个连接,终止一个连接则需四个分节。

1. 某个应用进程首先调用 close,我们称这一端执行主动关闭(active close)。这一端的 TCP 于是发送一个 FIN 分节,表示数据发送完毕。
2. 接收到 FIN 的另一端执行被动关闭(passive close)。这个 FIN 由 TCP 确认。它的接收也作为文件结束符传递给接收方应用进程(放在已排队等候该应用进程接收的任何其他数据之后),因为 FIN 的接收意味着应用进程在相应连接上再也接收不到额外数据。[②]
3. 一段时间后,接收到文件结束符的应用进程将调用 close 关闭它的套接口。这导致它的 TCP 也发送一个 FIN。
4. 接收到这个 FIN 的原发送方 TCP(即执行主动关闭的那一端)对它进行确认。

因为每个方向都需要有一个 FIN 和一个 ACK,所以一般需要四个分节。我们使用限定词"一般"是因为:有时步骤 1 的 FIN 随数据一起发送;另外,执行被动关闭那一端的 TCP 在步骤 2 和 3 发出的 ACK 与 FIN 也可以合并成一个分节。图 2.3 说明了这些分组的交换过程。

FIN 占据 1 个字节的序列号空间,这与 SYN 相同。所以每个 FIN 的 ACK 确认号是这个 FIN 的序列号加 1。

在步骤 2 与步骤 3 之间可以有从执行被动关闭端到执行主动关闭端的数据流。这称为半关闭(half-close),我们将在 6.6 节随 shutdown 函数再详细介绍。套接口关闭时,每一端 TCP 都要发送一个 FIN。这种情况在应用进程调用 close 时会发生,然而在进程终止时,所有打开的描述字将自愿(调用 exit 或从 main 函数返回)或不自愿(进程收到一个终止本进程的

---

① 译者注:失而复得的分组并不是超时重传的分组,而是由暂时的路由原因造成的迷途的分组。当路由稳定后,它们又会正常到达目的地,其前提是它们在此前尚未被路由器主动丢弃。

② 译者注:Unix 文件实际并不使用文件结束符,因为每个文件都对应一个索引节点,其中指出了文件长度,因此不必在文件尾再加个特殊的符号作为文件结束符。本书中文件结束符的读与写分别表示已读到文件尾或结束在文件尾的写。面向字节的数据传送流(如与数据报传送相对的字节流、Unix 的无名管道等)也使用文件结束符表示在某个方向上不再有数据待传送。在 TCP 字节流中,这是通过收发一个特殊的 FIN 分节来实现的。在不引起混淆的前提下,我们分别称发送方或接收方的 TCP 发送或接收了文件结束符。

信号)地关闭,此时仍然打开的TCP连接上也会发出一个FIN。

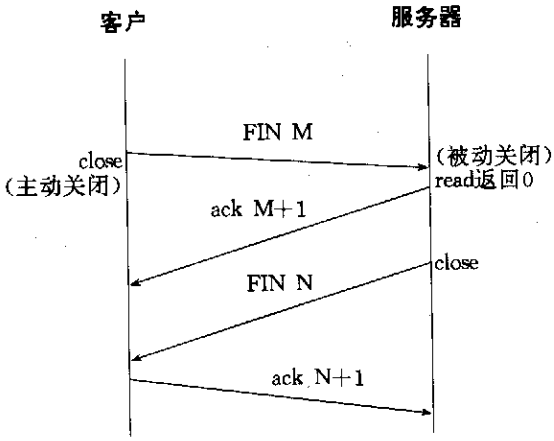


图2.3 TCP关闭连接时的分组交换

图2.3指出客户执行主动关闭,然而不管是客户还是服务器都可以执行主动关闭。通常情况是客户执行主动关闭,但某些协议如HTTP(超文本传送协议)则是服务器执行主动关闭。

TCP状态转换图

TCP连接的建立和终止可以用状态转换图(state transition diagram)来说明,见图2.4。

图中,为一个连接定义了11种状态,并且TCP规则决定如何从一个状态转换到另一个状态,这种转换基于当前状态及在该状态下所接收的分节。例如,当应用进程在CLOSED状态下执行一个主动打开时,TCP将发送一个SYN并从CLOSED状态转换成SYN_SENT状态。如果该TCP接着接收到一个附带ACK的SYN,它将发送一个ACK并转换成ESTABLISHED状态。这个最终状态是数据传送状态。

有两个外向箭头引导自ESTABLISHED状态的连接终止处理。如果应用进程在接收到文件结束符前调用close(主动关闭),则转换成FIN_WAIT_1状态。如果在ESTABLISHED状态下应用进程接收到FIN,则转换成CLOSE_WAIT状态。

我们用实线表示客户的状态转换,虚线表示服务器的状态转换。注意我们没有提到的两个转换:一个为同时打开(当两端几乎同时发送SYN并且这两个SYN在网络中彼此交错时),另一个为同时关闭(当两端同时发送FIN时)。TCPv1的第18章将讨论这两种情况,它们是可能发生的,但非常罕见。

展示状态转换图的理由之一是给出11种TCP状态的名称。netstat命令的输出包括这些状态,它是调试客户-服务器应用程序的有用的工具。在第5章我们将使用netstat去监视这些状态。

观察分组

图2.5说明一个完整的TCP连接所发生的实际分组交换情况:建立连接、传送数据和终止连接。图中展示了每个端点所历经的TCP状态。

例子中客户通告的MSS(最大分节大小)是1460(以太网上IPv4的典型值),而服务器

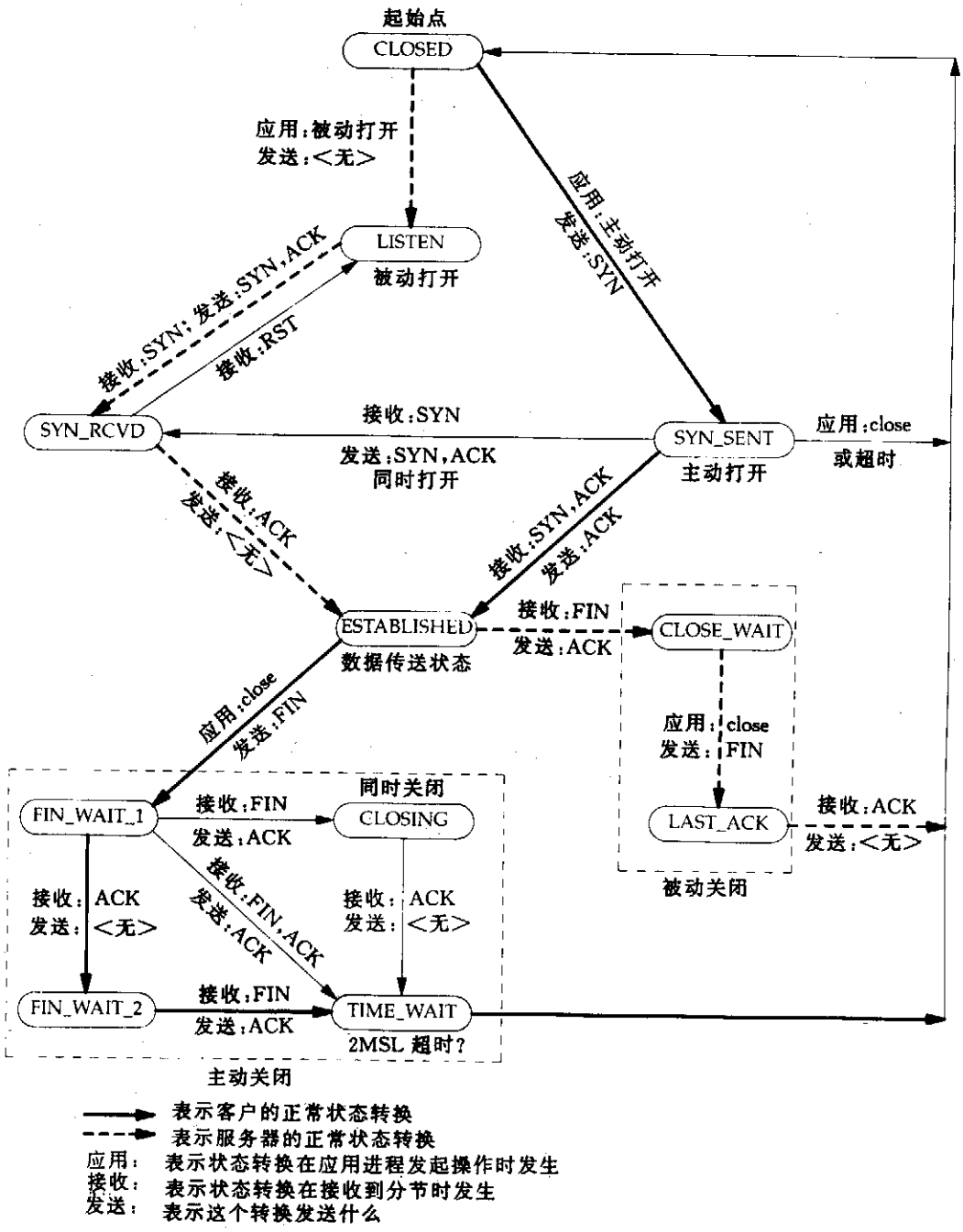


图 2.4　状态转换图

通知的 MSS 为 1024(以太网上较早的源自 Berkeley 实现的典型值)。不同方向的 MSS 可以不相同(习题 2.5)。

一旦连接建立，客户构造一个请求并发送给服务器。这里我们假设请求适合于单个 TCP 分节(即请求大小小于服务器通告的 1024 字节的 MSS)。服务器处理该请求并发送应

答，我们假设应答也适合于单个分节(本例小于1460)。图中，两个数据分节使用粗箭头表示。注意客户请求的确认是伴随服务器的应答发送的。这称为捎带(piggybacking)，它通常在服务器处理请求并产生应答的时间少于200毫秒时发生。如果服务器耗用更长时间，如1秒钟，我们将见到确认过后是应答。(TCP数据流机理在TCPv1的第19和第20章中详细描述。)

以后4个分节终止连接。注意：执行主动关闭的那一端(客户端)进入TIME_WAIT状态，我们将在下一节讨论这种情况。

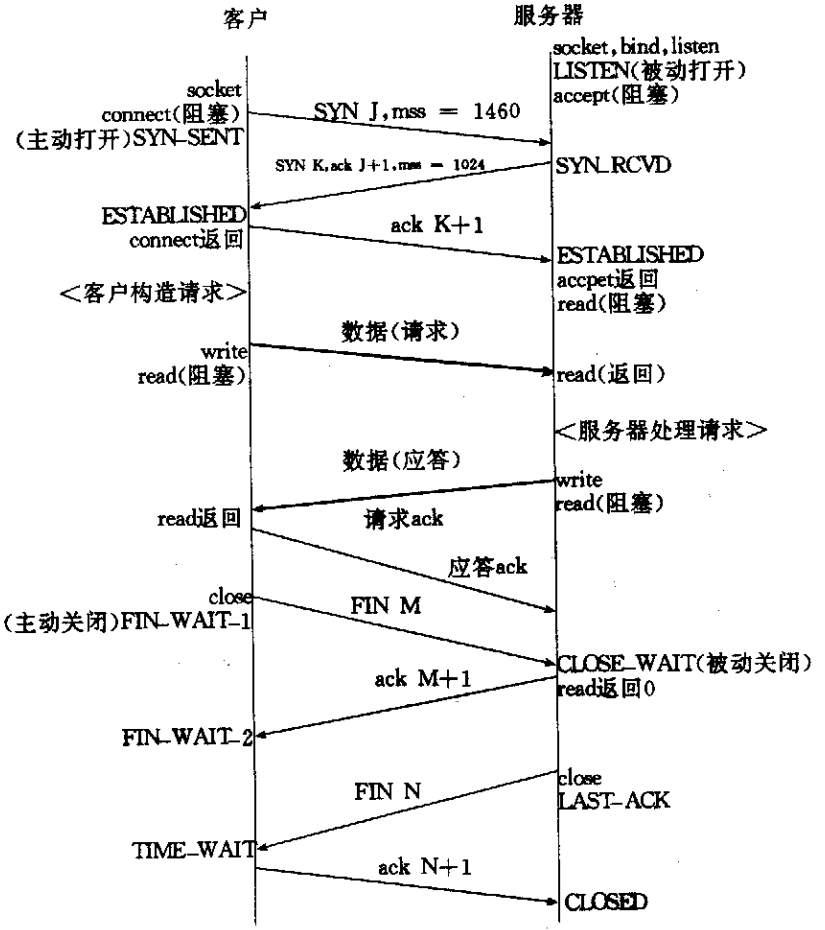


图2.5 TCP连接中的分组交换

图2.5中，如果连接的整个目的是发送一个单一分节的请求和接收一个单一分节的应答，那么使用TCP有8个分节的额外开销。如果使用UDP，那么只有两个分组需要交换：请求和应答。但是如果把TCP换成UDP，TCP提供给应用程序的可靠性将丧失，而把传输层(TCP)的一大堆细节推给UDP应用程序。TCP提供的另一个重要特性是拥塞控制，它也得由UDP应用程序处理。尽管如此，许多应用程序还是使用UDP，因为它们需交换的数据量很小，而UDP又避免了TCP建立连接和终止连接的额外开销。

这种情况下可以用T/TCP即事务TCP代替UDP。我们将在13.9节中描述。

2.6　TIME_WAIT 状态

毫无疑问,有关网络编程的 TCP 中最不容易理解的地方是它的 TIME_WAIT 状态。在图 2.4 中我们看到执行主动关闭的那端进入这种状态。这个端点留在该状态的持续时间是最长分节生命期 MSL(maximum segment lifetime)的两倍,有时称为 2MSL。

每个 TCP 实现都必须选择一个 MSL 值。RFC1122[Braden 1989]的建议值是 2 分钟,而源自 Berkeley 的实现传统上使用的值为 30 秒。这意味着 TIME_WAIT 状态的延迟在 1 分钟至 4 分钟之间。MSL 是 IP 数据报能在互联网中生存的最长时间。我们知道这个时间是有限的,因为每个数据报有一个 8 位的字段称为跳限(见图 A.1 中 IPv4 的 TTL 字段和图 A.2 中 IPv6 的跳限字段),它的最大值为 255。尽管这是一个跳数限制而不是真正的时间限制,我们仍需假设:具有最大跳限(255)的分组不能超过 MSL 秒还继续存在。

分组在路由异常时经常"迷途"。某个路由器崩溃或两个路由器间的链路断开时,路由协议需花数秒或数分钟才能稳定并找出另一条通路。在这段时间内有可能产生路由循环(路由器 A 把分组发送给路由器 B,而 B 再把它们发送回 A),此时分组可能就陷入这样的循环。在此期间,假设迷途的分组是一个 TCP 分节,发送方 TCP 超时并重传该分组,而重传的分组最终选择另一条路径到达目的地。但是,不久(自迷途的分组开始其旅程起最多 MSL 秒内)路由循环修复,迷失在这个循环中的分组最终也到达目的地。这个原来的分组称为迷途的重复分组(lost duplicate)或漫游的重复分组(wandering duplicate),与之相对的重复分组就是超时重传分组。TCP 必须正确处理这种重复的分组。

存在 TIME_WAIT 状态有两个理由:

1. 实现终止 TCP 全双工连接的可靠性
2. 允许老的重复分节在网络中消逝

第一个理由的解释如下:见图 2.5,假设最终的 ACK 丢失,服务器将重发最终的 FIN,因此客户必须维护状态信息以允许它重发最终的 ACK。如果不维护状态信息,它将响应以 RST(另外一个类型的 TCP 分节),而服务器则把该分节解释成一个错误。如果 TCP 打算执行所有必要的工作以彻底终止某个连接上两个方向的数据流(即全双工关闭),那么它必须正确处理连接终止序列四个分节中任何一个分节的丢失情况。本例子也说明执行主动关闭的一端为何进入 TIME_WAIT 状态,因为它可能不得不重发最终的 ACK。

要理解存在 TIME_WAIT 状态的第二个理由,我们假设 206.62.226.33 端口 1500 和 198.69.10.2 端口 21 之间有一个 TCP 连接。我们关闭这个连接后,在以后某个时候又重新建立起相同的 IP 地址和端口之间的 TCP 连接。后一个连接称为前一个连接的化身(incarnation),因为它们的 IP 地址和端口号都相同。TCP 必须防止来自某个连接的老重复分组在连接终止后再现,从而被误解成属于同一连接的化身。要实现这种功能,TCP 不能给处于 TIME_WAIT 状态的连接启动新的化身。既然 TIME_WAIT 状态的持续时间是 2MLS,这就足够让某个方向上的分组最多存活 MSL 秒即被丢弃,另一个方向上的应答最多存活 MSL 秒也被丢弃。通过实施这个规则,我们就能保证当成功建立一个 TCP 连接时,来自该连接先前化身的老重复分组都已在网络中消逝。

这个规则有一个例外：如果到达的SYN的序列号大于前一化身的结束序列号，源自Berkeley的实现将给当前处于TIME_WAIT状态的连接启动新的化身。TCPv2第958～959页对这种情况有详细的描述。它要求服务器执行主动关闭，因为接收到下一个SYN的那一方必须处于TIME_WAIT状态。rsh命令用到这种功能。RFC1185[Jacobson, Braden, and Zhang 1991]讲述了有关这种情形的陷阱。

2.7 端口号

任何时候，多个进程可同时使用TCP或UDP。TCP和UDP都用16位的端口号来区别这些进程。

当一个客户要与服务器接触时，它必须标识要连接的服务器。TCP和UDP定义了一组众所周知端口(well-known ports)，用于标识众所周知的服务。例如，支持FTP的任何TCP/IP实现都把21这个众所周知的端口分配给FTP服务器。TFTP(Trivial File Trqnsfer Protocol)所分配的是UDP端口号69。

另一方面，客户使用临时端口(ephemeral ports)。这些端口号通常由TCP或UDP自动分配给客户。客户一般不关心其临时端口的值，而只需确信它在所在主机中是唯一的。TCP和UDP代码能确保这种唯一性。

RFC1700[Reynolds and Postel 1994]含有一张由因特网已分配数值权威机构(IANA)定义的端口列表。不过，文件ftp://ftp.isi.edu/in-notes/iana/assignments/port-numbers通常比RFC 1700还要新。端口号分成三段：

1. 众所周知的端口为0～1023。这些端口由IANA分配和控制。可能时，相同端口号就分配给TCP和UDP的同一给定服务。例如，端口号80对TCP或UDP协议都供Web服务器使用，即使它目前的所有实现都只用TCP也一样。
2. 经注册的端口(registered ports)为1024～49151。这些端口不受IANA控制，但由IANA登记并提供它们的使用情况清单，以方便整个群体。可能时，同一端口号也分配给TCP和UDP的同一给定服务。例如6000～6063分配给这两种协议的X Window服务器。49151的上限是新的，因为RFC1700[Reynolds and Postel 1994]所列的上限为65535。
3. 49152～65535是动态的(dynamic)或私用的(private)端口。IANA不管这些端口，它们就是我们所称的临时端口。

(49152这个数是65535的四分之三。)图2.6说明了这些端口号的分配情况。

图中，我们注意以下各点：

- Unix系统有保留端口(reserved ports)的概念，它是小于1024的任何端口。这些端口只能分配给超级用户进程的套接口。所有众所周知的端口都为保留端口，因此分配这些端口的服务器启动时必须具有超级用户的特权。

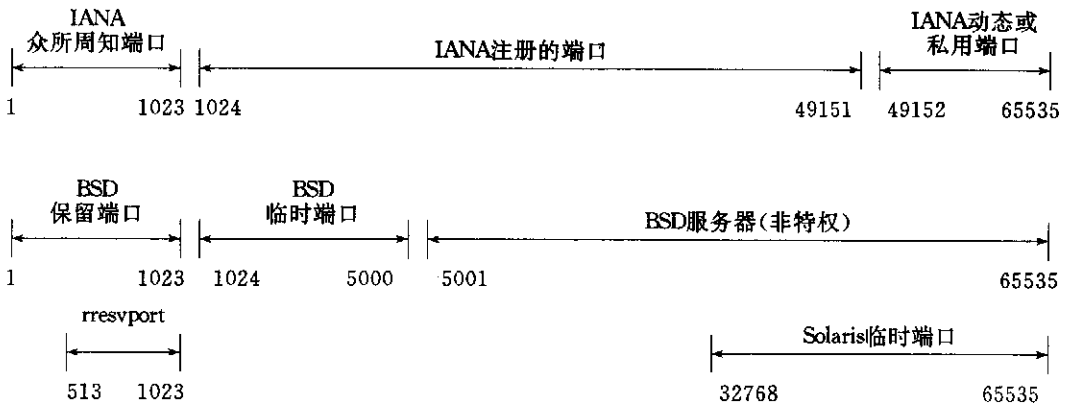


图 2.6　端口号的分配

- 历史上，源自 Berkeley 的实现(起始于 4.3BSD)已在 1024～5000 范围内分配临时端口。这在 80 年代初期情况还不错，因为那时服务器主机还没有能力同时处理那么多客户的请求，但是今天很容易找到一台在任何给定的时间内同时支持多于 3977 个客户的主机。所以有些系统分配另外范围的临时端口(如图 2.6 所示的 Solaris)，用于扩充临时端口的数量。

  由于是这个原因，当前许多系统实现的临时端口范围上限为 5000。5000 这个上限实际上是一个排版错误[Borman 1997a]，本应为 50000。

- 有少数客户(不是服务器)需要一个保留端口用于客户-服务器的认证：rlogin 和 rsh 客户就是常见的例子。这些客户调用库函数 rresvport 创建一个 TCP 套接口并给它分配 513～1023 范围内的一个未使用的端口。这个函数通常先测试 1023，如果失败，再测试 1022、1021……，直到 513。如果到 513 之前能找到一个端口则成功，否则失败。

  注意：BSD 的保留端口和 rresvport 函数都跟 IANA 众所周知端口的后半部分重迭。这是因为早先的 IANA 众所周知端口的上限为 255。1992 年的 RFC1340 开始在 256～1023 之间分配众所周知的端口。先前的“Assigned Numbers”文档即 1990 年的 RFC 1060 称端口 256～1023 为 Unix 标准服务。80 年代有不少源自 Berkeley 的服务器使用 512 以后的端口(留下 256～511 的空档)。rresvport 函数选择从 1023 开始往下寻找，直至 513。

套接口对

一个 TCP 连接的套接口对(socket pair)是一个定义该连接的两个端点的四元组：本地 IP 地址、本地 TCP 端口号、远程 IP 地址、远程 TCP 端口号。套接口对唯一标识一个互联网上的所有 TCP 连接。

标识每个端点的两个值(IP 地址和端口号)通常称为一个套接口。

我们可以把套接口对的概念扩充到 UDP，即使 UDP 是无连接的。当描述套接口函数(bind、connect、getpeername 等)时，我们注明它们在说明套接口对中那个值，如 bind 函数要求应用程序说明本地 IP 地址和本地端口，既可以是 TCP 套接口，也可以是 UDP 套接口。

2.8 TCP端口号与并发服务器

并发服务器中主服务器循环派生子进程来处理每个新的连接。如果子进程仍使用众所周知的端口来服务很长的请求，将发生什么？让我们来检查一个典型的序列。首先服务器在主机bsdi上启动(图1.16)，它是多宿的，有两个IP地址206.62.226.35和206.62.226.66。服务器在它的众所周知的端口(本例为21)上执行被动打开，并等待客户的请求，见图2.7。

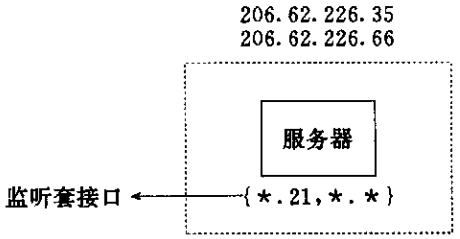


图2.7 TCP服务器在端口21上执行被动打开

我们使用符号{*.21,*.*}指出服务器的套接口对。服务器在任意本地接口(第一个星号)的端口21上等待连接请求。远程IP地址和远程端口没有指定，我们用"*.*"表示。我们称它为监听套接口(listening socket)。

> 我们用点号分开IP地址和端口号，因为这是netstat的用法。这种表示有些让人混淆，因为点号既用于域名(如solaris.kohala.com.21)，也用于IPv4点分十进制数记法(如206.62.226.33.21)。

这里的星号称为通配(wildcard)符。如果运行服务器的主机有多个IP地址(如本例)，服务器可以说明它只接受到达某个特定本地接口的外来连接。这是或者一个接口或者任意接口的选择。服务器不能指定多个地址的列表。通配的本地地址表示"任意"接口。图1.9的通配地址是在调用bind前通过设置套接口地址结构中的IP地址为INADDR_ANY指定的。

此后第一个客户在IP地址为196.69.10.2的主机上启动，对IP地址为206.62.226.35的主机上的服务器执行主动打开。我们假设本例的客户主机TCP选择的临时端口为1500，见图2.8。我们在客户的下方标出了它的套接口对。

当服务器接收并接受这个客户的连接时，它fork一个自身的拷贝，由子进程来处理该客户的请求，见图2.9。我们将在4.7节描述fork函数。

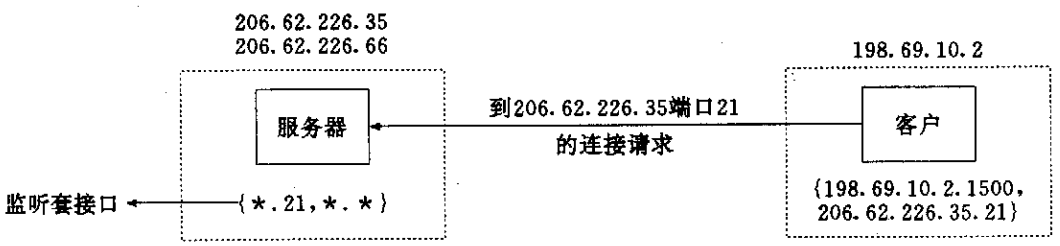


图2.8 客户对服务器的连接请求

此时,我们必须区别服务器的监听套接口和已连接套接口(connected socket)。注意已连接套接口使用跟监听套接口相同的端口号:21。同时注意,一旦连接建立,多宿服务器上已连接套接口的本地地址(206.62.226.35)随即填入。

下一步我们假设在客户主机上有另一个客户进程请求与同一服务器连接。客户主机 TCP 分配给新客户的套接口的未使用临时端口为 1501,见图 2.10。服务器能够区别这两个连接:第一个连接的套接口对和第二个连接的套接口对是不同的,因为客户方的 TCP 给第二个连接选择了一个未使用的端口(1501)。

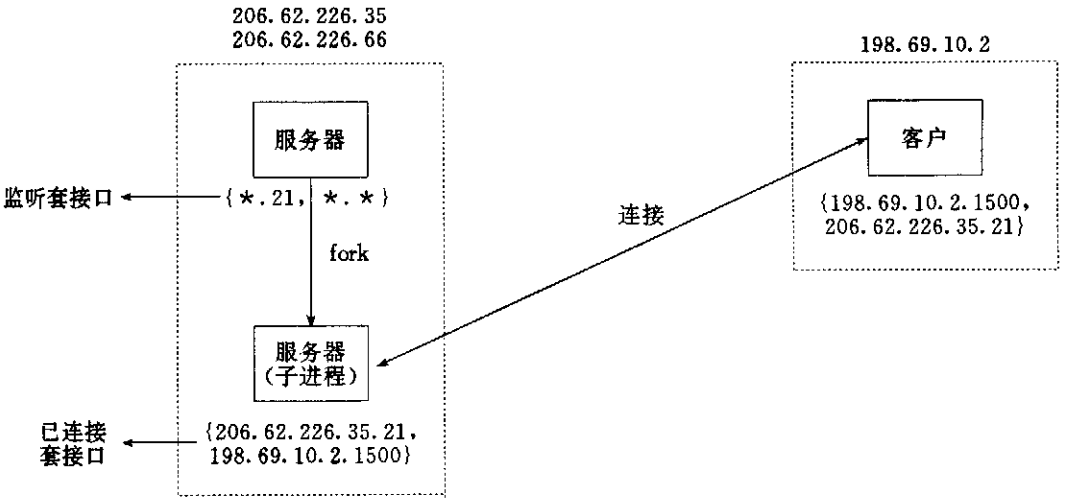


图 2.9　并发服务器让子进程处理客户

通过本例应注意,TCP 无法仅仅通过查看目的端口号来分离外来的分节。它必须查看套接口对的所有四个元素才能确定由哪个端点接收到达的分节。图 2.10 中,对于端口 21 存在三个套接口。如果一个分节来自 198.69.10.2 端口 1501,目的地为 206.62.226.35 端口 21,那它是递送给第一个子进程。如果一个分节来自 198.69.10.2 端口 1501,目的地为 206.62.226.35端口 21,那它是递送给第二个子进程。所有其他的目的端口为 21 的 TCP 分节则是递送给拥有监听套接口的最初服务器(父进程)。

2.9　缓冲区大小及限制

下面我们将介绍一些影响 IP 数据报大小的限制。我们先介绍这些限制,然后就它们如何影响应用进程传送的数据综合分析。

- IPv4 数据报的最大大小是 65535 字节,包括 IPv4 头部。这是因为图 A.1 的总长度字段占 16 位。
- IPv6 数据报的最大大小是 65575 字节,包括 40 字节的 IPv4 头部。这是因为图 A.2 的有效负载长度字段占 16 位。注意 IPv6 的有效负载长度字段不含 IPv6 头部,而 IPv4 的总长度字段含 IPv4 头部。

  IPv6 有一个特大有效负载选项,它把有效负载长度字段扩展到 32 位,不过这个选项要求 MTU(见下一点)超过 65535 的数据链路。这是为主机到主机的内部连接而设

计的，如HIPPI，它们没有内在的MTU。

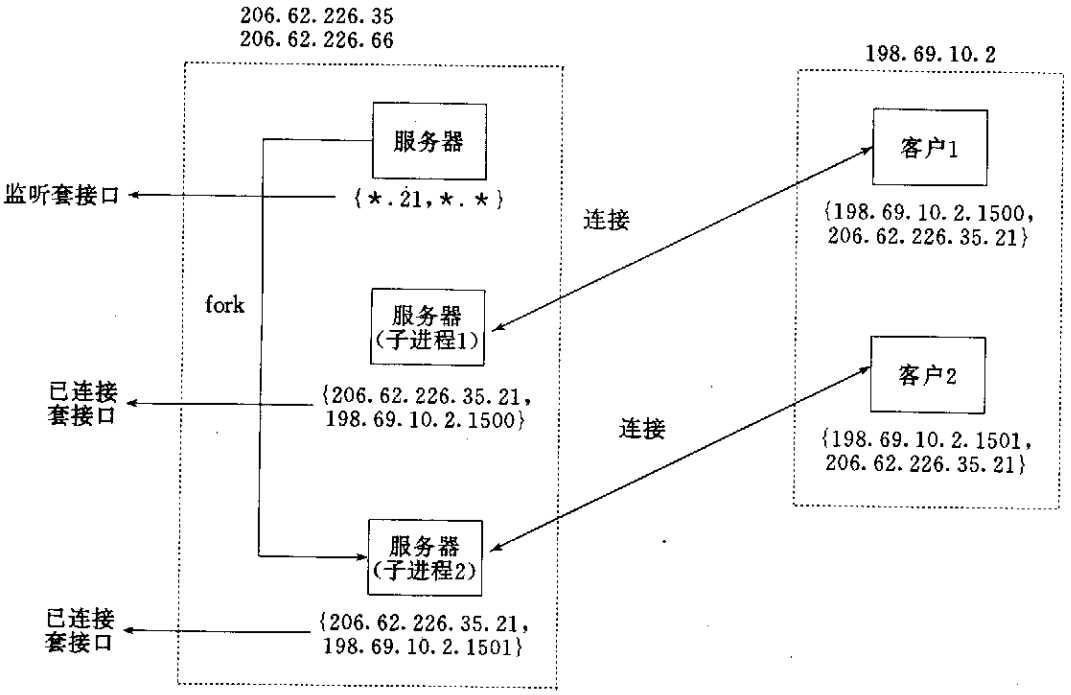


图2.10 第二个客户与同一个服务器的连接

- 很多网络有一个最大传输单元MTU(maximum transmission unit)，它由硬件规定。例如以太网的MTU为1500字节。其他链路(如使用PPP协议的点对点链路)其MTU可以配制。老的SLIP链路通常使用296字节的MTU。
  IPv4要求的最小MTU是68字节，IPv6要求的最小MTU为576字节。
- 在两台主机间的路径上的最小MTU称为路径MTU(path MTU)。1500字节的以太网MTU是当今常见的路径MTU。路径MTU在不同方向可以不相同，因为在因特网中路由是非对称的[Paxson 1196]，即从A到B的路径与从B到A的路径可以不相同。
- 当一个IP数据报将从某个接口发出时，如果它的大小超过相应链路的MTU，IPv4和IPv6都将执行分片(fragmentation)。各片段到达目的地前不会被重组(reassembling)。IPv4主机对其产生的数据报执行分片，IP4路由器对其转发的数据报也执行分片。但IPv6只在数据报产生的主机执行分片；IPv6路由器对其转发的数据报不执行分片。

  我们必须小心这些术语的的使用。IPv6路由器可以执行分片，但只对那些由路由器产生的数据报而不是转发的数据报。这时的路由器实际上作为主机运作。例如，大多数路由器支持Telnet协议，管理员就用它来配置路由器。由路由器的Telnet服务器产生的IP数据报是由路由器产生的，而不是由路由器转发的。

你可能注意到图 A.1 中 IPv4 头部的 DF 字段,它用于处理 IPv4 分片,而在 IPv6 头部不存在相应的字段(图 A.2)。既然分片是例外情况而不是通常情况,因此 IPv6 引入一个可选头部以提供分片信息。

- IPv4 头部(图 A.1)的 DF("不分片")位若被设置,那么不管是发送主机还是转发路由器都不能对本数据报分片。当路由器接收到一个超过其外出链路 MTU 大小且设置了 DF 位的 IPv4 数据报时,它将产生一 ICMPv4 的"destination unreachable, fragmentation needed but DF bit set(目的地不可达,需分片但 DF 位已设置)"出错消息(图 A.15)。

  因为 IPv6 路由器不执行分片,因此 IPv6 数据报隐含设置了 DF 位。如果 IPv6 路由器接收到一个超过其外出链路 MTU 大小的 IPv6 数据报,它将产生一个 ICMPv6 的"packet too big(分组太大)"出错消息(图 A.16)。

  IPv4 的 DF 位和 IPv6 的隐含 DF 位可用于路径 MTU 的发现(IPv4 见 RFC1191 [Mogul and Deering 1990];IPv6 见 RFC 1981[McCann, Deering, and Mogul 1996])。例如,如果 TCP 使用 IPv4 的这个技术,它发送的数据报将都设置 DF 位。如果某个中间路由器返回一个 ICMP"destination unreachable, fragmentation needed but DF bit set"错误,TCP 就减小每个数据报的数据量并重传。路径 MTU 的发现对 IPv4 是可选的,但所有 IPv6 的实现都必须支持它。
- IPv4 和 IPv6 都定义了最小重组缓冲区大小(minimum reassembly buffer size):任何 IPv4 和 IPv6 的实现都必须支持的最小数据报大小。对 IPv4 其值为 576 字节,对 IPv6 为 1500 字节。例如对于 IPv4 来说,我们不能确信给定的目的主机是否能接受 577 字节的数据报。所以有很多使用 UDP 的应用程序(DNS、RIP、TFTP、BOOTP、SNMP)避免产生大于这个大小的数据报。
- TCP 有一个 MSS,用于向对方 TCP 通告对方在每个分节中能发送的最大 TCP 数据量。见图 2.5,SYN 分节带有 MSS 选项。MSS 的目的是告诉对方其重组缓冲区大小的实际值,从而避免分片。MSS 经常设置成 MTU 减去 IP 和 TCP 头部的固定长度。在以太网中使用 IPv4 的 MSS 为 1460,使用 IPv6 的值为 1440(两者的 TCP 头部都为 20 个字节,而 IPv4 头部是 20 字节,IPv6 头部为 40 字节)。

  在 TCP 的 MSS 选项中,MSS 值是一个 16 位的字段,最大值为 65535。这很适合 IPv4,因为 IPv4 数据报中的最大 TCP 数据量为 65495(65535 减去 IPv4 头部 20 个字节和 TCP 头部 20 个字节)。但是 IPv6 有特大有效负载选项,因此需要使用另外的技术(RFC 2147[Borman 1997b])。首先,没有特大有效负载选项的 IPv6 数据报中最大的 TCP 数据量为 65515(65535 减去 TCP 头部 20 字节)。因此 65535 这个 MSS 值被认为是表示"无限"的特殊值,它只在用到特大有效负载选项时才使用,而这种情况又要求 MTU 超过 65535。如果 TCP 使用特大有效负载选项,并且接收到的对方通告的 MSS 为 65535,那么它所发送数据报的大小限制就是接口 MTU。如果这个值太大(也就是说路径上某个链路的 MTU 要比它小),那么路径 MTU 的发现功能将确定这个较小值。

TCP输出

图2.11展示应用进程写数据到TCP套接口的过程。

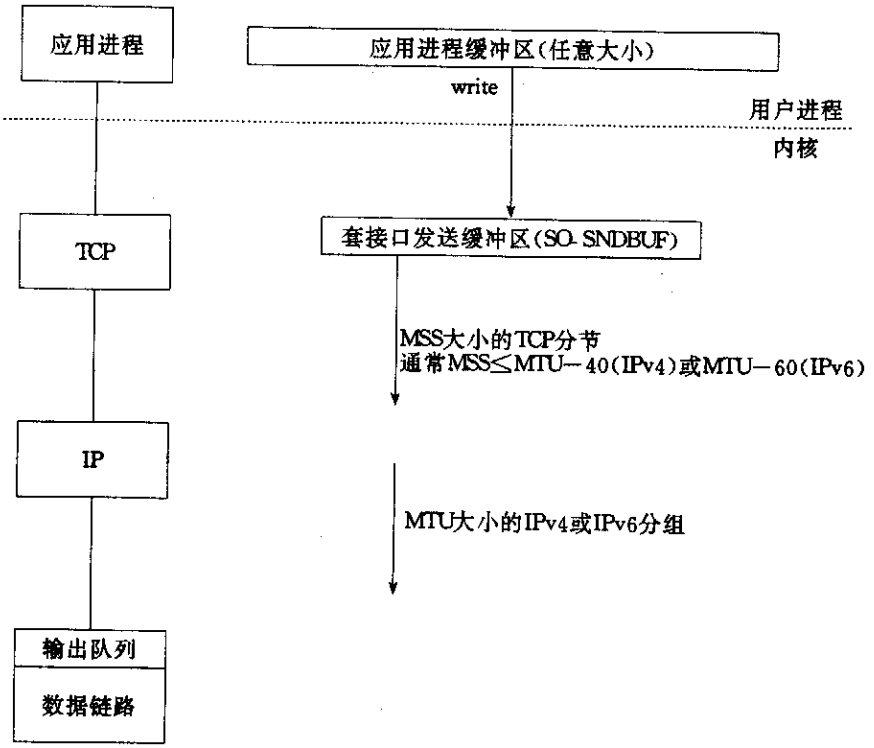


图2.11 应用进程写TCP套接口时涉及的步骤和缓冲区

每一个TCP套接口有一个发送缓冲区,我们可以用SO_SNDBUF套接口选项来改变这一缓冲区的大小(7.5节)。当应用进程调用write时,内核从应用进程的缓冲区中拷贝所有数据到套接口的发送缓冲区。如果套接口的发送缓冲区容不下应用程序的所有数据(或是应用进程的缓冲区大于套接口发送缓冲区,或是套接口发送缓冲区还有其他数据),应用进程将被挂起(睡眠)。这里假设套接口是阻塞的,这是通常的缺省设置(我们将在第15章阐述非阻塞的套接口)。内核将不从write系统调用返回,直到应用进程缓冲区中的所有数据都拷贝到套接口发送缓冲区。所以从写一个TCP套接口的write调用成功返回仅仅表示我们可以重新使用应用进程的缓冲区。它并不告诉我们对方的TCP或对方应用进程已接收到数据(我们将在7.5节详细介绍SO_LINGER套接口选项)。

TCP取套接口发送缓冲区的数据并把它发送给对方TCP,其过程基于TCP数据传送的所有规则(TCPv1的第19和20章)。对方TCP必须确认收到的数据,只有收到对方的ACK,本方TCP才能删除套接口发送缓冲区中已确认的数据。TCP必须保留数据拷贝直到对方确认为止。

TCP以MSS大小或更小的块发送数据给IP(它同时给每个数据块安上TCP头部以构成分节),其中MSS是由对方通告的,当对方未通告时就用536这个值。IP给每个TCP分节安上IP头部以构成数据报,查找其目的IP地址的路由表项以确定外出接口,然后把数据报传递给相应的数据链路。IP可能先将数据报分片,再传送给链路层。但如上所述,MSS选项

的目的是避免分片，而新的实现又使用路径 MTU 发现功能。每一个链路有一个输出队列，如果输出队列满，则分组丢弃，并沿协议栈向上返回一个错误：从链路层到 IP 层，再从 IP 层到 TCP 层。TCP 将注意到这个错误，并在以后某个时刻重传这个分节。应用进程并不知道这种暂时情况。

UDP 输出

图 2.12 展示应用进程写数据到 UDP 套接口的过程。这一次我们展示的套接口发送缓冲区用虚线框，因为它并不存在。UDP 套接口有发送缓冲区大小（我们可用 SO_SNDBUF 套接口选项修改，见 7.5 节），但是它仅仅是写到套接口的 UDP 数据报的大小上限。如果应用进程写一大于套接口发送缓冲区的数据报，则返回 EMSGSIZE 错误。因为 UDP 是不可靠的，它不必保存应用进程的数据拷贝，因此无需一个真正的发送缓冲区。（应用进程的数据在沿协议栈向下传递时，以某种形式拷贝到内核的缓冲区，然而当链路层把数据传出后这个拷贝就丢弃。）

UDP 简单地安上它的 8 个字节的头部以构成数据报并把它传递给 IP。IPv4 或 IPv6 给它安上相应的 IP 头部，执行路由操作确定外出接口，然后或者直接把数据报加入链路层输出队列（如果适合于 MTU），或是分片后再把每个片段加入数据链路层的输出队列。

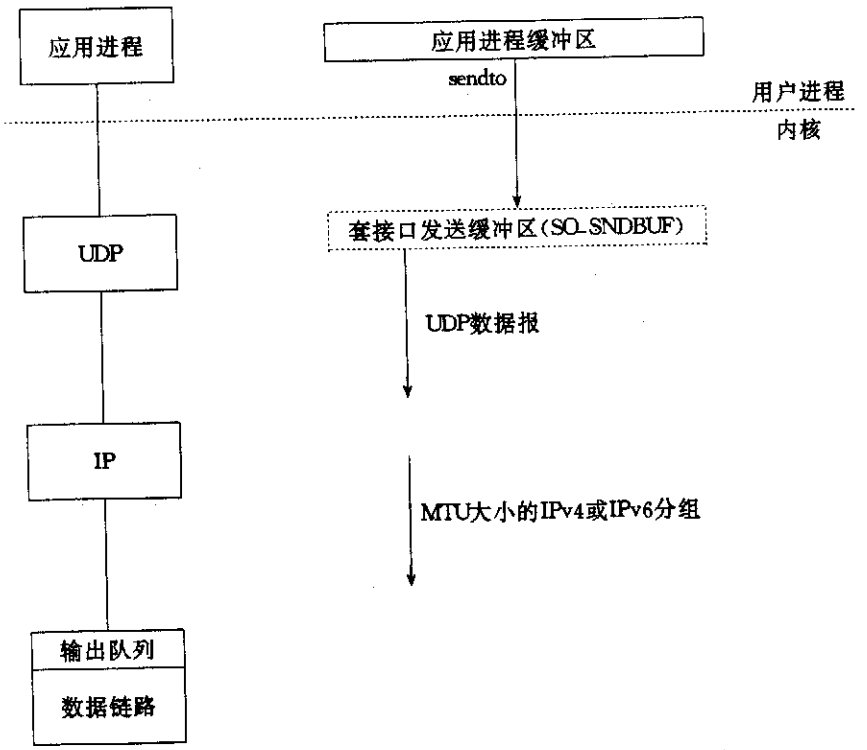


图 2.12　应用进程写 UDP 套接口时涉及的步骤与缓冲区

如果 UDP 应用进程发送一个大数据报（如 2000 字节数据报），它比 TCP 应用进程更有可能分片，因为 TCP 会把应用进程数据划分成 MSS 大小的块，但 UDP 却没有对等的手段。

从写 UDP 套接口的 write 调用成功地返回表示数据报或所有片段已被加入链路层的

输出队列。如果输出队列没有足够的空间存放数据报或它的某个分节,UDP将返回应用进程ENOBUFS错误。

不幸的是,有些UDP的实现不返回这种错误,这样甚至数据报未经发送就丢弃的情况应用进程也不知道。

2.10 标准因特网服务

图2.13列出TCP/IP多数实现都提供的一些标准服务。注意表中所有服务同时使用TCP和UDP提供,并且这两个协议的端口号也相同。

名 字	TCP端口	UDP端口	RFC	说 明
echo(回射)	7	7	862	服务器返回客户发送的数据
discard(丢弃)	9	9	863	服务器废弃客户发送的数据
daytime(时间/日期)	13	13	867	服务器返回直观可读的日期和时间
chargen(字符生成)	19	19	864	TCP服务器发送连续的字符流,直到客户终止连接。每当客户发送一个数据报,UDP服务器就返回一个包含随机数量字符的数据报
time(时间)	37	37	868	服务器返回一个32位二进制数值的时间。这个数值表示从1900年1月1日子时(UTC时间)以来所流逝的秒数

图2.13 大多数实现提供的标准TCP/IP服务③

这些服务通常由Unix主机的inetd守护进程提供。使用标准的Telnet客户程序,很容易测试这些功能。例如,下面测试daytime和echo两种服务器。

```
solaris % telnet bsdi daytime
Trying 206.62.226.35...                    Telnet客户输出
Connected to bsdi.kohala.com.              Telnet客户输出
Escape character is '^]'.                  Telnet客户输出
Tue Mar 19 11:06:49 1996                   daytime服务器输出
Connection closed by foreign host.         Telnet客户输出(服务器关闭连接)
solaris % telnet bsdi echo
Trying 206.62.226.35...                    Telnet客户输出
Connected to bsdi.kohala.com.              Telnet客户输出
Escape character is '^]'.                  Telnet客户输出
hello,world                                我们键入这行
hello,world                                它由服务器回射回来
^]                                         我们键入<Ctrl-]>以与Telnet客户交谈
```

③ 译者注: 本表同时给出了各个标准因特网服务的英文名称和中文名称,其中英文名称是正式名称(/etc/services文件使用这些名称)。之所以这么区分是因为本书是围绕其中两种服务的实现展开的,为区分本书中的实现与各个Unix系统的内部实现,我们用中文名称称呼前者,用英文名称称呼后者(原书也对两者作了区分)。另外,内部实现的服务总是使用标准端口号,本书实现的服务则可根据情况选择。因此我们使用英文名称服务名时,必定与其标准端口号对应。

```
telnet> quit                          告诉客户我们已测试完毕
Connection closed.                    这次客户自己关闭连接
```

在这两个例子中，我们键入主机名和服务名(daytime 和 echo)。这些服务名由/etc/services 文件映射到图 2.13 的端口号，见 9.9 节。

注意：当我们连接到 daytime 服务器时，服务器执行主动关闭；而连接到 echo 服务器时，则客户执行主动关闭。回忆图 2.4，执行主动关闭的 TCP 端是历经 TIME_WAIT 状态的一方。

2.11　常见因特网应用程序的协议使用

图 2.14 总结了各种常见的因特网应用程序对协议的使用情况。

应用程序	IP	ICMP	UDP	TCP
Ping		•		
Traceroute		•	•	
OSPF(路由协议)	•			
RIP(路由协议)			•	
BGP(路由协议)				•
BOOTP(引导协议)			•	
DHCP(引导协议)			•	
NTP(时间协议)			•	
TFTP(低级 FTP)			•	
SNMP(网络管理)			•	
SMTP(电子邮件)				•
Telnet(远程登录)				•
FTP(文件传送)				•
HTTP(Web)				•
NNTP(网络新闻)				•
DNS(域名系统)			•	•
NFS(网络文件系统)			•	•
Sun RPC(远程过程调用)			•	•
DCE RPC(远程过程调用)			•	•

图 2.14　各种常用因特网应用程序的协议使用情况

图中，前两个应用程序 Ping 和 Traceroute 是诊断应用程序，它们使用 ICMP 协议。Traceroute 构造自己的 UDP 分组来发送，并读 ICMP 的应答。紧接着是三个流行的路由协议，它们展示了路由协议使用的各种传输协议。OSPF 采用原始套接口直接使用 IP，而 RIP 使用 UDP，BGP 使用 TCP。

下面 5 个是基于 UDP 的应用程序，接着 5 个应用程序使用 TCP。最后 4 个是同时使用 UDP 和 TCP 的应用程序。

2.12 小 结

UDP是一个简单的、不可靠的无连接的协议，而TCP是一个复杂的、可靠的、面向连接的协议。尽管绝大多数因特网应用程序使用TCP(Web、Telnet、FTP和email)，这两个传输协议都是必要的。20.4节将阐述用UDP替代TCP的理由。

TCP用三路握手建立连接，用四分组交换序列终止连接。当建立TCP连接时，它将从CLOSED状态转换到ESTABLISHED状态，而当终止连接时，它又回到CLOSED状态。在TCP连接中存在11种状态，状态转换图给出从一个状态转换到另一个状态的规则。理解状态转换图是使用netstat命令诊断网络问题的基础，也是理解调用诸如connect、accept和close等函数时所发生过程的关键。

TCP的TIME_WAIT状态是网络编程人员不容易理解的概念。存在这一状态是为了实现完整地终止全双工的连接(即处理最终的ACK丢失的情形)，并允许老的重复分节从网络中消逝。

2.13 习 题

2.1 我们已经提到IP版本4和版本6。IP版本5有什么情况，IP版本0、1、2和3又是什么？(提示：查找最新的"Assigned Numbers"RFC。)

2.2 从哪里你可以找到有关IP版本5的信息。

2.3 图2.11中，如果没收到对方的MSS选项则TCP使用536的MSS值，为什么使用这个值？

2.4 画出类似于图2.5的第1章所述时间/日期客户-服务器的分组交换过程，假设服务器在单个分节中返回26个字节的完整数据。

2.5 在一台以太网上的主机和一台令牌环网上的主机之间建立连接，其中以太网上主机通告的MSS为1460，令牌环网上主机通告的MSS为4096。两端主机都没有实现路径MTU发现功能。观察分组，我们在两个方向上都找不到大于1460字节的数据，为什么？

2.6 图2.14中我们说OSPF直接使用IP，OSPF数据报的IPv4头部(图A.1)的协议字段是什么值？

第 2 部分　基本套接口编程

第 3 章　套接口编程简介

3.1　概　述

本章开始描述套接口 API。我们从套接口地址结构开始，本书的每个例子几乎都有这些结构，它们可以在两个方向上传递：从进程到内核和从内核到进程。其中从内核到进程方向的传递是值-结果参数的一个例子，本书中，我们将会碰到这些参数的许多例子。

地址转换函数在地址的文本表达和套接口地址结构中的二进制值间进行转换。多数现存的 IPv4 代码使用 inet_addr 和 inet_ntoa 函数，而两个新函数 inet_pton 和 inet_ntop 则适用于 IPv4 和 IPv6 两种代码。

这些地址转换函数有一个问题，即它们是与所转换的地址类型协议相关的：要么 IPv4 要么 IPv6。为克服此问题，我们开发了一组函数，这组函数的命名均以“sock_”打头，以协议无关方式使用套接口地址结构。在整个第 2 部分，我们都使用这组函数，以使我们的代码与协议无关。

3.2　套接口地址结构

大多数套接口函数都需要一个指向套接口地址结构的指针作为参数。每个协议族都定义它自己的套接口地址结构。这些结构的名字均以“sockaddr_”开头，并以对应每个协议族的唯一后缀结束。

IPv4 套接口地址结构

IPv4 套接口地址结构通常也称为“网际套接口地址结构”，它以“sockaddr_in”命名，定义在头文件<netinet/in.h>中。图 3.1 所示为 Posix.1g 的定义。

```
struct in_addr {
  in_addr_t  s_addr;          /* 32-bit IPv4 address */
                              /* network byte ordered */
};
struct sockaddr_in {
  uint8_t        sin_len;     /* length of structure (16) */
  sa_family_t    sin_family;  /* AF_INET */
  in_port_t      sin_port;    /* 16-bit TCP or UDP port number */
                              /* network byte ordered */
  struct in_addr sin_addr;    /* 32-bit IPv4 address */
                              /* network byte ordered */
```

```
    char        sin_zero[8];    /* unused */
};
```

图 3.1 网际(IPv4)套接口地址结构:sockaddr_in

利用图 3.1 所示的例子,我们对套接口地址结构做几点一般性的说明。

- 长度成员 sin_len,是为增加 OSI 协议(图 1.15)支持,随 4.3BSD-Reno 一起增加的。在此之前,第一个成员是 sin_family,它是一个无符号短整数 unsigned short。并不是所有的厂家都支持套接口地址结构长度成员,而且 Posix.1g 也不要求有这个成员。我们给出的数据类型 uint8_t 是典型的,且这种数据类型是随 Posix.1g 一起新添加的(图 3.2)。

  正是因为有了长度成员,才简化了变长套接口地址结构的处理。

- 即使有长度成员,我们也无需设置它、无需检查它,除非我们涉及到路由套接口(第 17 章)。它是内核中处理来自不同协议族的套接口地址结构(如路由表代码)的例程使用的。

  从进程到内核传递套接口地址结构的 4 个套接口函数:bind、connect、sendto 和 sendmsg,都要经历源自 Berkeley 的实现中的 sockargs 函数。这个函数从进程拷贝套接口地址结构,并显式地以作为参数传递给这 4 个函数的地址结构的长度设置 sin_len 成员。而从内核到进程传递套接口地址结构的 5 个套接口函数:accept、recvfrom、recvmsg、getpeername 和 getsockname,均在返回到进程之前设置 sin_len 成员。

  遗憾的是,通常没有一个简单的编译时测试来确定一个实现是否为它的套接口地址结构定义了长度成员。在我们的代码中,我们通过测试常值 HAVE_SOCKADDR_SA_LEN 来确定(图 D.2),但是否定义这个常值则需编译一个使用这一可选结构成员的简单测试程序并看是否编译成功来决定。在图 3.4 中我们将看到,如果套接口地址结构有长度成员,则 IPv6 实现需定义 SIN6_LEN。一些 IPv4 实现(如 Digital Unix)基于编译选项(如_SOCKADDR_LEN)确定是否给应用程序提供套接口地址结构中的长度成员,这为较早的程序提供了兼容性。

- Posix.1g 只需要这个结构中的 3 个成员:sin_family、sin_addr 和 sin_port。对于 Posix 兼容的实现来说,定义额外的结构成员是可以接受的,这对于网际套接口地址结构来说也是正常的。几乎所有的实现都增加 sin_zero 成员,所以所有的套接口地址结构大小都至少是 16 字节。
- 我们给出了 s_addr、sin_family 和 sin_port 的 Posix.1g 数据类型。in_addr_t 数据类型必须是一个至少 32 位的无符号整数类型,in_port_t 必须是一个至少 16 位的无符号整数类型,而 sa_family_t 可以是任何无符号整数类型,在支持长度成员的实现中,它一般是一个 8 位的无符号整数,而在不支持长度成员的实现中,它则是一个 16 位的无符号整数。图 3.2 列出了 Posix.1g 定义的这些数据类型以及将会遇到的其他数据类型。

数据类型	说明	头文件
int8_t	带符号的 8 位整数	<sys/types.h>
uint8_t	无符号的 8 位整数	<sys/types.h>
int16_t	带符号的 16 位整数	<sys/types.h>
uint16_t	无符号的 16 位整数	<sys/types.h>
int32_t	带符号的 32 位整数	<sys/types.h>
uint32_t	不带符号的 32 位整数	<sys/types.h>
sa_family_t	套接口地址结构的地址族	<sys/socket.h>
socklen_t	套接口地址结构的长度，一般为 uint 32_t	<sys/socket.h>
in_addr_t	IPv4 地址，一般为 uint32_t	<netinet/in.h>
in_port_t	TCP 或 UDP 端口，一般为 uint16_t	<netinet/in.h>

图 3.2　Posix.1g 要求的数据类型

- 我们也将遇到这样的数据类型：u_char、u_short、u_int 和 u_long，它们都是无符号的。Posix.1g定义这些类型时特地标记它们已过时，是仅为向后兼容才提供的。
- IPv4 地址和 TCP 或 UDP 端口号在套接口地址结构中总是以网络字节序来存储，我们在使用这些成员时，必须牢记这一点。(在 3.4 节我们将详细说明主机字节序与网络字节序的区别。)
- 可以有两种不同的方法来访问 32 位 IPv4 地址。例如如果 serv 定义为网际套接口地址结构，那么 serv.sin_addr 给出的 32 位 IPv4 地址将是一个 in_addr 结构，而 serv.sin_addr.s_addr 给出的 32 位 IPv4 地址则是一个 in_addr_t(通常是无符号的 32 位整数)。因此，我们必须正确地使用 IPv4 地址，尤其是在将它作为函数的参数时，因为编译器对传递结构和传递整数的处理是完全不同的。

  由于历史上的原因，sin_addr 成员是一个结构，而不仅仅是一个无符号长整数。早期的版本(4.2BSD)将 in_addr 结构定义为不同结构的联合，以允许访问 32 位 IPv4 地址 4 个字节中的每个字节，或者 2 个 16 位值中的任一个。这用在 A 类、B 类和 C 类地址中获取地址中的适当字节。但随着子网划分技术的来临和无类地址编排(A.4 节)的出现，各种地址类正在消失，这就不需要联合了。今天，多数系统已经废除了该联合，将 in_addr 定义为仅有一个无符号长整数成员的结构。

- sin_zero 成员暂不使用，但总是将它置为 0。为方便起见，在初始化结构时，我们一般是将整个结构置为 0，而不仅仅是设置 sin_zero 成员为 0。

  虽然多数结构的使用不要求这一成员为 0，但当捆绑一个非通配 IPv4 地址时，此成员必须为 0(TCPv2 第 731～732 页)。

- 套接口地址结构仅在给定主机上使用：虽然结构中的某些成员(如 IP 地址和端口号)用在不同主机间的通信中，但结构本身并不参与通信。

通用套接口地址结构

当作为参数传递给任一个套接口函数时，套接口地址结构总是通过指针来传递，但通过指针来取得此参数的套接口函数必须处理来自所支持的任何协议族的套接口地址结构。

有一个问题是如何声明所传指针的数据类型。ANSI C 中有很简单的解决办法：它有通用的指针类型 void * 。但是，套接口函数是在 ANSI C 之前定义的，1982 年采用了这样的一个办法：在<sys/socket.h>头文件中定义一个通用的套接口地址结构，如图 3.3 所示。

```
struct sockaddr {
  uint8_t       sa_len;
  sa_family_t   sa_family;    /* address family: AF_xxx value */
  char          sa_data[14];  /* protocol-specific address */
};
```

图 3.3 通用套接口地址结构：sockaddr

于是，套接口函数被定义为采用指向通用套接口地址结构的指针，这正如用 ANSI C 函数原型写出来的 bind 函数所示：

```
int bind (int , struct sockaddr * , socklen_t);
```

这要求对这些函数的任何调用都必须将指向特定于协议的套接口地址结构的指针类型转换成指向通用套接口地址结构的指针。例如：

```
struct sockaddr_in  serv;       /* IPv4 socket address structure */
/* fill in serv{} */
bind(sockfd, (struct sockaddr * ) &serv, sizeof(serv));
```

如果我们省略了其中的类型转换“(struct sockaddr *)”，再假设系统的头文件中有一个 bind 函数的 ANSI C 原型，那么 C 编译器就会产生这样的警告信息：“warning：passing arg 2 of 'bind' from incompatible pointer type(警告：把不兼容的指针类型传递给 'bind' 函数的第二个参数)”。

从应用程序开发人员的观点看，这些通用的套接口地址结构的唯一用途是给指向特定于协议的地址结构的指针转换类型。

> 回忆一下 1.2 节，我们在头文件 unp.h 中把 SA 定义为串“struct sockaddr”正是为了缩短转换这些指针的类型必须写的代码。
>
> 从内核的角度看，使用指向通用套接口地址结构指针的原因是：内核必须依据调用者的指针，将其转换为 struct sockaddr * 类型，然后检查 sa_family 的值来确定结构的类型。但从应用程序开发人员的角度看，指针类型为 void * 则更简单，不需进行明确的类型转换。

IPv6 套接口地址结构

IPv6 套接口地址结构在头文件<netinet/in.h>中定义，如图 3.4 所示。

```
struct in6_addr {
  unit8_t s6_addr[16];          /* 128-bit IPv6 address */
                                /* network byte ordered */
```

```
};
#define SIN6_LEN      /* required for compile-time tests */
struct sockaddr_in6 {
  uint8_t          sin6_len;      /* length of this struct (24) */
  sa_family_t      sin6_family;   /* AF_INET6 */
  in_port_t        sin6_port;     /* transport layer port# */
                                  /* network byte ordered */
  uint32_t         sin6_flowinfo; /* priority & flow label */
                                  /* network byte ordered */
  struct in6_addr  sin6_addr;     /* IPv6 address */
                                  /* network byte ordered */
};
```

图 3.4　IPv6 套接口地址结构：sockaddr_in6

关于 IPv6 套接口 API 的扩展定义在 RFC 2133 中[Gilligan et al. 1997]，Posix.1g 对 IPv6 未作任何说明。由于我们在图 3.4 中使用了可能的 Posix.1g 数据类型，而这些类型又是在 RFC 2133 面世后的一个 Posix.1g 草案中规定的，所以图中的某些数据类型与 RFC 2133 不符。

对于图 3.4，我们应注意以下几点：

- 如果系统支持套接口地址结构中的长度成员，则 SIN6_LEN 常值必须定义。
- IPv6 地址族是 AF_INET6，而 IPv4 地址族是 AF_INET。
- 结构中的成员是有序排列的，因此，如果 sockaddr_in6 结构是 64 位对齐的，则 128 位的成员 sin6_addr 也是 64 位对齐的。在一些 64 位处理机上，如果数据存储在 64 位边界的位置，则对 64 位数据的访问将优化处理。
- sin6_flowinfo 成员分成三个字段：
  - 低 24 位是流量标号；
  - 下 4 位是优先级；
  - 再下 4 位保留。

流量标号和优先级字段在图 A.2 中描述。必须注意，优先级字段的使用仍是一个研究课题。

套接口地址结构的比较

在图 3.5 中，我们对本书将遇到的 4 种套接口地址结构进行了比较：IPv4、IPv6、Unix 域(图 14.1)和数据链路(图 17.1)。在这个图中，我们假设所有的套接口地址结构都包含 1 个字节的长度成员，地址族成员也占用一个字节，其他所有成员都占有确切的最短长度。前两种套接口地址结构是定长的，而 Unix 域结构和数据链路结构是可变长度的。为了处理可变长度的结构，我们把指向套接口地址结构的指针作为参数传递给套接口函数，同时把它的长度作为另一个参数传递。我们在每个定长结构下方给出了此结构的字节数(4.4BSD 实现)。

结构 sockaddr_un 本身不是可变长度的(图 14.1),但其信息量即结构中的路径名的长度则是可变的。当传递指向这些结构的指针时,我们必须小心处理长度成员即套接口地址结构本身的长度成员(如果其实现支持此成员),以及传给内核或从内核返回的长度。

这个图展示了我们贯穿全书的一种风格:结构名用黑体字,后跟花括号,如:**sockaddr_in{}**。

前面我们已经注意到,长度成员是随着 4.3BSD Reno 版本一起增加到所有的套接口地址结构中的,如果套接口 API 的原始版本中已经有了长度成员,那么所有的套接口函数就不需要长度参数,如函数 bind 和 connect 的第三个参数。与此相反,结构的大小可以包含在结构的长度成员中。

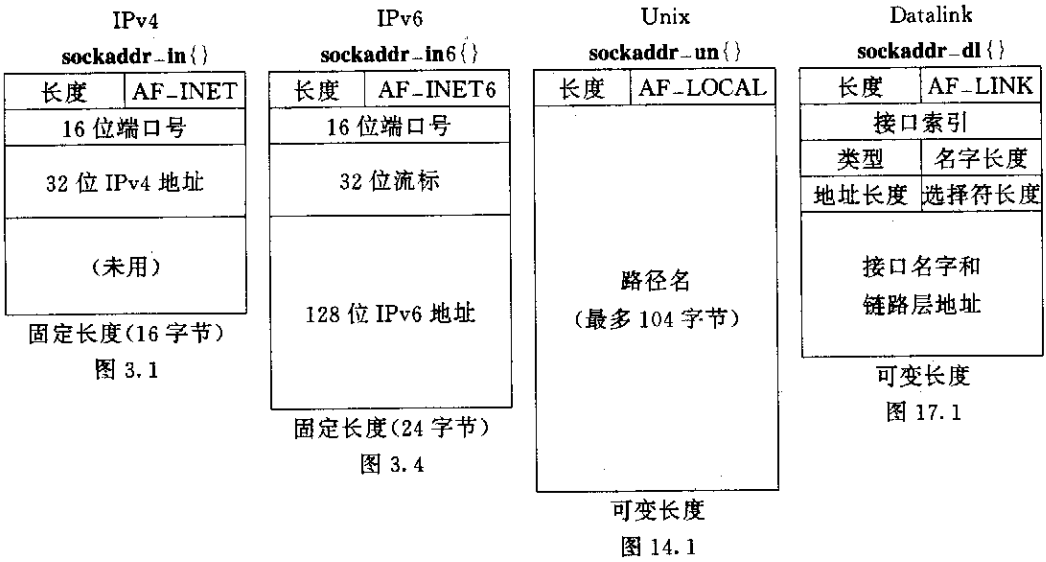


图 3.5 不同套接口地址结构的比较

3.3 值-结果参数

我们已经知道,当把套接口地址结构传递给套接口函数时,总是通过指针来传递的,即传递的是一个指向结构的指针。结构的长度也作为参数来传递,其传递的方式取决于结构的传递方向:从进程到内核,还是从内核到进程。

1. 从进程到内核传递套接口地址结构有 3 个函数:bind、connect 和 sendto,这 3 个函数的一个参数是指向套接口地址结构的指针,另一个参数是结构的整数大小,例如:

```
struct sockaddr_in serv;
/* fill in serv{} */
connect (sockfd,(SA *) &serv, sizeof(serv));
```

由于指针和指针所指结构的大小都传递给内核,所以从进程到内核要确切拷贝多少数据是知道的,如图 3.6 所示。

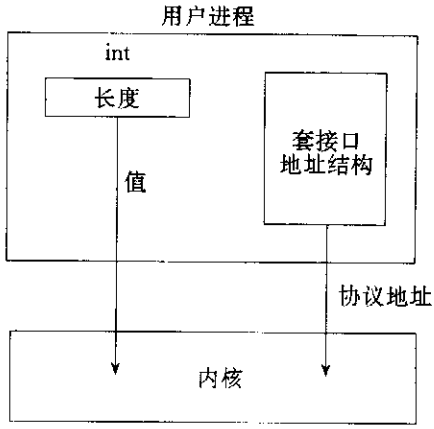


图 3.6　从进程到内核传递套接口地址结构

下一章中我们将看到，套接口地址结构大小的数据类型确切地说应该是 socklen_t，而不是 int，但 Posix.1g 建议将 socklen_t 定义为 uint32_t。

2. 与前面的传递方向相反，从内核到进程传递套接口地址结构有 4 个函数：accept、recvfrom、getsockname 和 getpeername。这 4 个函数的两个参数是：指向套接口地址结构的指针和指向表示结构大小的整数的指针，例如：

```
struct sockaddr_un cli;          /* Unix domain */
socklen_t len;
len = sizeof(cli);               /* len is a value */
getpeername(unixfd, (SA *) &cli, &len);
/* len may have changed */
```

为何将结构大小由整数改为指向整数的指针呢？这是因为：当函数被调用时，结构大小是一个值（此值告诉内核该结构的大小，使内核在写此结构时不至于越界），当函数返回时，结构大小又是一个结果（它告诉进程内核在此结构中确切存储了多少信息），这种参数类型叫值-结果参数，如图 3.7 所示。

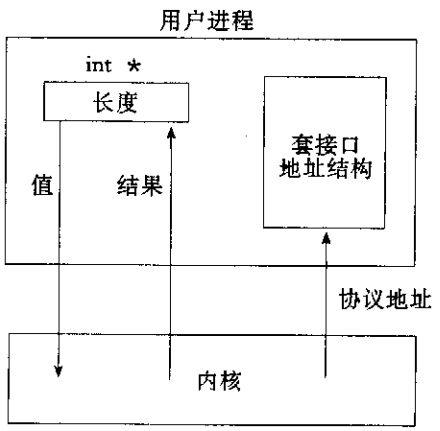


图 3.7　从内核到进程传递套接口地址结构

我们将在图4.11中看到一个值-结果参数的例子。

> 我们已经对在进程和内核间传递的套接口地址结构进行了讨论。对于诸如4.4BSD的实现，所有的套接口函数都是内核中的系统调用，这是正确的。但在某些实现中，如著名的系统V，套接口函数只是作为普通用户进程执行的库函数。这些函数与内核中的协议栈如何接口是一个实现的细节问题，一般来说，对我们不会有什么影响。然而为简单起见，我们继续说这些结构通过诸如bind和connect这样的函数在进程与内核间进行传递。(在C.1节我们将看到系统V的确也在进程和内核间传递用户的套接口地址结构，但这是作为流消息(streams messages)的一部分进行传递的。)
>
> 另有两个传递套接口地址结构的函数：recvmsg和sendmsg(13.5节)，但我们将看到，长度不是一个函数参量，而是一个结构成员。

当使用值-结果参数作为套接口地址结构的长度时，如果套接口地址结构是定长的(图3.5)，则从内核返回的值也是定长的，如对于IPv4，sockaddr_in是16；对于IPv6，sockaddr_in6是24。但对于变长的套接口地址结构(如Unix域的sockaddr_un)，返回值可能比结构的最大长度小(图14.2)。

在网络编程中，值-结果参数的最常见例子就是返回的套接口地址结构长度，本书中我们还能见到其他的值-结果参数：

- select函数中间的3个变量(6.3节)。
- getsockopt函数的长度变量(7.2节)。
- 使用函数recvmsg时，msghdr结构中的两个成员：msg_namelen和msg_controllen(13.5节)。
- ifconf结构中的成员ifc_len(图16.2)。
- sysctl函数的前两个长度参数(17.4节)。

3.4 字节排序函数

考虑一个16位整数，它由2个字节组成。内存中存储这两个字节有两种方法：一种是将低序字节存储在起始地址，这称为小端(little-endian)字节序，另一种方法是将高序字节存储在起始地址，这称为大端(big-endian)字节序。图3.8示出了这两种格式。

在图的顶部存储地址增大的方向为从右到左，在图的底端存储地址增大的方向为从左到右，MSB(最高有效位)为16位数的最左一位，LSB(最低有效位)为16位数的最右一位。

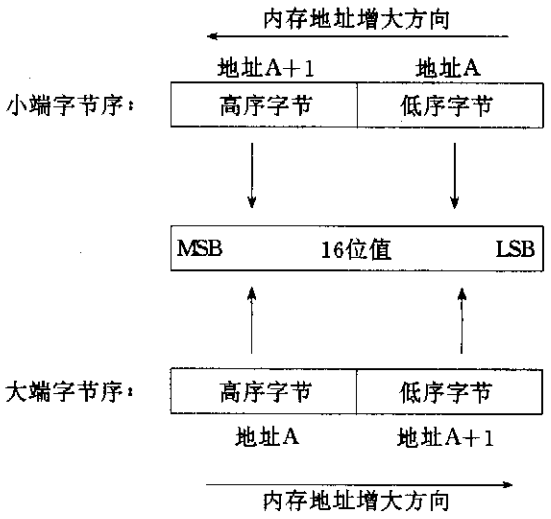


图 3.8 16 位整数的小端字节序和大端字节序

术语“小端”和“大端”表示多字节值的哪一端(小端或大端)存储在该值的起始地址。

遗憾的是,这两种字节序中没有标准,这两种格式都有系统使用。我们把某给定系统所用的字节序称为主机字节序(host byte order),图 3.9 中所示程序能输出主机字节序。

```
1 #include    "unp.h"
2 int
3 main(int argc,char * * argv)
4 {
5     union {
6         short   s;
7         char    c[sizeof(short)];
8     } un;
9     un.s = 0x0102;
10    printf("%s:",CPU_VENDOR_OS);
11    if(sizeof(short) == 2) {
12        if(un.c[0] == 1 && un.c[1] == 2)
13            printf("big-endian\n")
14        else if (un.c[0] == 2 && un.c[1] == 1)
15            printf("little-endian\n");
16        else
17            printf("unknown\n");
18    } else
19        printf("sizeof(short) = %d\n",sizeof(short));
20    exit(0);
21 }
```

图 3.9 确定主机字节序的程序[intro/byteorder.c]

我们把2字节数0x0102存储为一个短整数，然后查看两个连续的字节c[0]（图3.8中的地址A）和c[1]（图3.8中的地址A+1），以此来确定字节序。

字符串CPU_VENDOR_OS由GNU的autoconf程序在配置本书中的软件时确定，它标识CPU类型、厂家和操作系统版本。我们这儿给出一些例子，它们是这个程序在图1.16所示的各系统上运行时的输出。

```
aix % byteorder
powerpc-ibm-aix4.2.0.0:big-endian
alpha % byteorder
alpha-dec-osf4.0: little-endian
bsdi % byteorder
i386-pc-bsdi3.0: little-endian
gemini % byteorder
sparc-sun-sunos4.1.4: big-endian
hpux % byteorder
hppa1.1-hp-hpux10.30: big-endian
linux % byteorder
i586-pc-linux-gnu: little-endian
solaris % byteorder
sparc-sun-solaris2.5.1: big-endian
unixware % byteorder
i386-univel-sysv4.2MP: little-endian
```

我们已讨论了16位整数的字节序，显然，相同的讨论也适用于32位整数。

> 有些系统，或者在系统复位时（MIPS 2000），或者在程序运行中的任何时刻（Intel i860），可以在大端字节序和小端字节序间互换。

由于网络协议必须指定一个网络字节序（network byte order），作为网络编程人员，必须清楚各种字节序间的区别。例如，在一个TCP分节中，有一个16位的端口号和一个32位的IPv4地址，发送协议栈和接收协议栈必须对此多字节字段的传送字节序进行协调，以达成一致。网际协议在处理这些多字节整数时，使用大端字节序。

从理论上说，具体实现可以把多字节整数按主机字节序存储在套接口地址结构中，当需要写到协议头部或从协议头部中取出时，再转换到网络字节序格式或转换回来，这样可以避免我们牵涉细节。但是，由于历史上的原因和Posix.1g的规定，套接口地址结构中的某些成员按网络字节序进行维护。这样，我们就必须考虑主机字节序和网络字节序间的相互转换，转换时用到下列四个函数：

```
#include <netinet/in.h>
    uint16_t htons(uint16_t host16bitvalue);
    uint32_t htonl(uint32_t host32bitvalue);
                                                    均返回：网络字节序值
    uint16_t ntohs(uint16_t net16bitvalue);
    uint32_t ntohl(uint32_t net32bitvalue);         均返回：主机字节序值
```

在上述函数的名字中，h 代表 host，n 代表 network，s 代表 short，l 代表 long。short 和 long 是4. 2BSD的 Digital VAX 实现的历史产物。我们应该转变思想，把 s 看作一个 16 位的值(如 TCP 或 UDP 端口号)，把 l 看作 32 位的值(如 IPv4 地址)。实际上，在 64 位的 Digital Alpha 中，长整数占用 64 位，而 htonl 和 ntohl 函数却对 32 位值操作。

当使用这些函数时，我们不关心主机字节序和网络字节序的真实值(大端或小端)。我们所做的只是调用适当的函数来对给定值进行主机字节序与网络字节序间的转换。在那些与网际协议有相同字节序(大端)的系统中，这四个函数通常被定义为空宏。

对于网络分组中所含的数据而不是协议头部中字段的字节序问题，我们将在 5. 18 节和习题 5. 8 中进行进一步的讨论。

至此，我们还没有定义术语字节(byte)。既然几乎所有的计算机系统使用 8 位字节，我们就用此术语来表示一个 8 位的量。大多数因特网标准用术语 octet(八位组)而不是 byte 来表示 8 位的量，这在 TCP/IP 时代的早期就已是这种局面了，因为许多早期的工作是在诸如 DEC-10 这样的系统上开展的，而这些系统不使用 8 位的字节。

> 20 世纪 80 年代网络编程有一个通病：在 Sun 工作站(大端 Motorola 68000)上开发代码时没有调用这四个函数中的任一个。这些代码在这些工作站上工作得很好，但当移植到小端机器(如 VAX 系列机)上时，便根本不能工作。

3.5　字节操纵函数

多字节字段的操纵有两组函数，它们无需解释数据，无需假设数据是以空字符结束的 C 字符串。当我们涉及套接口地址结构这类问题时，就需要这样的函数，因为我们要对诸如 IP 地址这样的字段进行操作。这样的字段可能包含多个字节的 0，但又不是 C 字符串。在头文件<string. h>中定义、名字以 str 打头的函数处理的是以空字符结束的 C 字符串。

第一组函数，其名字以字母 b 打头，代表 byte，起源于 4. 2BSD，现仍由几乎任何支持套接口函数的系统提供；第二组函数，其名字以 mem 打头，代表 memory，起源于 ANSI C 标准，由任何支持 ANSI C 库的系统提供。

我们首先给出源自 Berkeley 的函数，本书中我们仅用了一个即 bzero。(我们使用它是因为它只有两个参数，比起 3 个参数的 memset 函数来，要容易记些，这在前边已解释过。)其他两个函数：bcopy 和 bcmp，在别的应用程序中可能会见到。

```
#include <strings.h>
void bzero(void *dest, size_t nbytes);
void bcopy (const void *src,void *dest, size_t nbytes);
int bcmp(const void *ptr1, const void *ptr2, size_t nbytes);
                                        返回：0——相等，非 0——不相等
```

这是我们首次遇到 ANSI C 的 const 限定词。就它在这儿的三处使用来说，它表示所限定的指针(src、ptr1 和 ptr2)所指的内容是不能由函数改动的。换句话说，函数只读而不修改由 const 指针所指的内存单元。

bzero 将目标中指定数目的字节置为 0，我们经常用此函数来把套接口地址结构初始化为 0；bcopy 将指定数目的字节从源移到目标；bcmp 比较任意两个字节串，若相同则返回值为 0，否则返回值为非 0。

下面的函数是 ANSI C 函数：

```
#include <string.h>
void *memset(void *dest, int c, size_t len);
void *memcpy(void *dest, const void *src, size_t nbytes);
int memcmp(const void *ptr1, const void *ptr2, size_t nbytes);
                                        返回：0——相同，>0 或<0——不相同
```

memset 将目标中指定数目的字节置为值 c。memcpy 与 bcopy 类似，但交换了两个指针参数的顺序，源与目标重迭时，bcopy 能正确处理，memcpy 的操作结果则是不可知的。这时，必须使用 ANSI C 的 memmove 函数(习题 30.3)。

有一个方法可记住 memcpy 两个指针的顺序：它们是按跟 C 的赋值语句相同的顺序从左到右书写的：

```
dest = src;
```

有一个方法可记住 memset 最后两个参数的顺序：所有的 ANSI C 的 memXXX 函数都要求有一个长度参数，且它总是最后一个参数。

memcmp 比较任意两个字节串，如果相同则返回 0，如果不同则返回一个非 0 值，具体是大于 0 还是小于 0 则取决于第一个不等的字节：若 ptr1 所指字节大于 ptr2 所指字节，则大于 0，否则小于 0。进行比较操作时，假定两个不等的字节均为无符号字符(unsigned char)。

3.6 inet_aton、inet_addr 和 inet_ntoa 函数

在本节和下一节，我们介绍两组地址转换函数，它们在 ASCII 字符串(人们比较喜欢用的格式)与网络字节序的二进制值(此值存于套接口地址结构中)间转换地址。

1. inet_aton、inet_addr 和 inet_ntoa 在点分十进制数串(例如，“206.62.226.33”)与它的 32 位网络字节序二进制值间转换 IPv4 地址，你在许多代码中可能会遇到这些函数。
2. 两个较新的函数：inet_pton 和 inet_ntop 对 IPv4 和 IPv6 地址都能处理，我们将在下一节对它们进行介绍并纵贯全书使用。

```
#include <arpa/inet.h>
int inet_aton(const char *strptr, struct in_addr *addrptr);
                                        返回:1——串有效,0——串有错
in_addr_t inet_addr(const char *strptr);
  返回:若成功,返回 32 位二进制的网络字节序地址;若有错,则返回 INADDR_NONE
char *inet_ntoa(struct in_addr inaddr);
                                        返回:指向点分十进制数串的指针
```

第一个函数 inet_aton 将 strptr 所指的 C 字符串转换成 32 位的网络字节序二进制值，并通过指针 addrptr 来存储。如果成功返回 1，否则返回 0。

> 函数 inet_aton 有一个没写到正式文档中的特征：如果指针为空，则函数仍然执行输入串的有效性检查，但不存储任何结果。

inet_addr 进行相同的转换，返回值为 32 位的网络字节序二进制值。这个函数存在这样的问题：所有 2^{32} 个可能的二进制值都是有效的 IP 地址(从 0.0.0.0 到 255.255.255.255)，但当出错时返回一个常值 INADDR_NONE(一般为一个 32 位均为 1 的值)。这就意味着点分十进制数串 255.255.255.255(这是 IPv4 的有限广播地址，见 18.2 节)不能由此函数处理，因为它的二进制值被用来指示函数失败。

> 当我们进行 ping 255.255.255.255 这样的操作时，许多较早期的 Ping 都返回一个“unknown host(不可知主机)”这样的错误，原因就是函数 inet_addr 失败，于是就将此点分十进制串作为一个主机名来查找，同样失败。
>
> 函数 inet_addr 还有一个潜在的问题，有些非正式的文档把出错时的返回值定义为 −1 而不是 INADDR_NONE。这样比较函数的返回值(无符号的值)与负常值时可能会出问题，这取决于 C 编译器。

现在，人们都反对使用函数 inet_addr，而用函数 inet_aton 来代替。更好的办法是用下一节介绍的新函数，它们对 IPv4 和 IPv6 都能处理。

函数 inet_ntoa 将一个 32 位的网络字节序二进制 IPv4 地址转换成相应的点分十进制数串。由函数返回值所指的串驻留在静态内存中，这意味着函数是不可重入的，这在 11.14 节我们再进行讨论。最后注意一下，这个函数以结构为参数，而不是指向结构的指针。

> 函数以结构为参数是很少见的，更多的是以指向结构的指针为参数。

3.7　inet_pton 和 inet_ntop 函数

这两个函数较新，对 IPv4 和 IPv6 地址都能处理，本书通篇都使用这两个函数。字母 p 和 n 分别代表 presentation 和 numeric。地址的表达(presentation)格式通常是 ASCII 串，数值(numeric)格式则是存在于套接口地址结构中的二进制值。

```
#include <arpa/inet.h>
int inet_pton(int family, const char *strptr, void *addrptr);
                返回:1——成功,0——输入不是有效的表达格式,-1——出错
const char *inet_ntop(int family ,const void *addrptr ,char *strptr, size_t len);
                返回:指向结果的指针——成功,NULL——出错
```

两个函数的参数 family 既可以是 AF_INET,也可以是 AF_INET6。如果以不被支持的地址族作为 family 参数,两个函数都返回错误,并将 errno 置为 EAFNOSUPPORT。

第一个函数转换由指针 strptr 所指的串,通过指针 addrptr 存储二进制结果。如果成功,则返回值为 1;如果对于指定的 family 输入串不是有效的表达格式,则返回值为 0。

inet_ntop 进行相反的转换,即从数值格式(addrptr)到表达格式(strptr)进行转换。参数 len 是目标的大小,以免函数溢出其调用者的缓冲区。为有助于规定这个大小,在头文件<netinet/in.h>中有如下定义:

```
#define INET_ADDRSTRLEN     16    /* for IPv4 dotted-decimal */
#define INET6_ADDRSTRLEN    46    /* for IPv6 hex string */
```

如果 len 太小,无法容纳表达格式结果(包括终止的空字符),则返回一个空指针,并置 errno 为 ENOSPC。

函数 inet_ntop 的参数 strptr 可不能是个空指针,调用者必须为目标分配内存并指定大小。成功时,此指针即函数的返回值。

图 3.10 总结了这一节和上一节我们所讨论的 5 个函数。

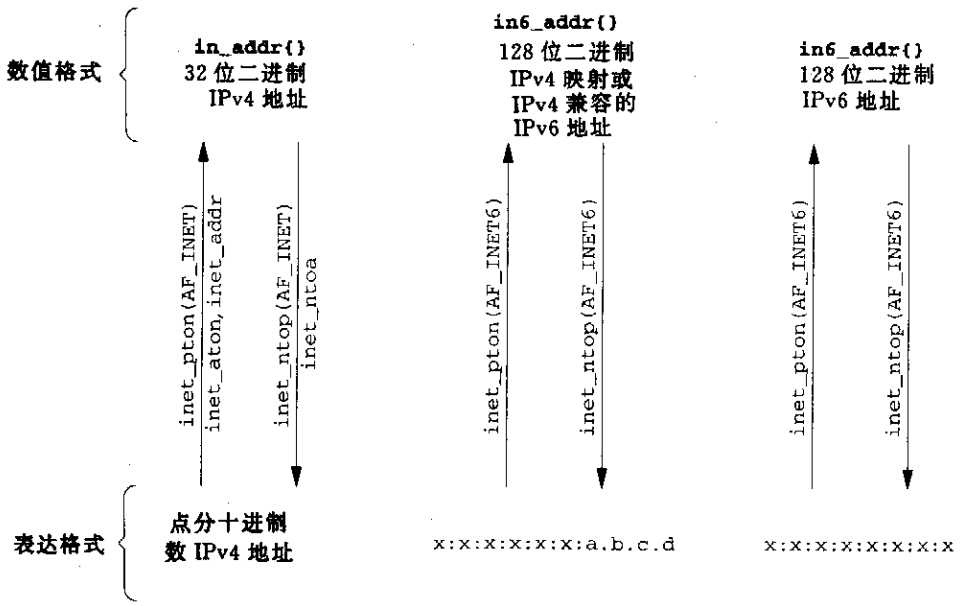


图 3.10 地址转换函数小结

举例

即使你的系统还不支持 IPv6，你可以采取下列措施开始使用这些新函数，即用代码：

```
inet_pton(AF_INET, cp, &foo.sin_addr);
```

代替代码：

```
foo.sin_addr.s_addr = inet_addr(cp);
```

再用代码：

```
char str[INET_ADDRSTRLEN];
ptr = inet_ntop(AF_INET,&foo.sin_addr,str,sizeof(str));
```

代替代码：

```
ptr = inet_ntoa(foo.sin_addr);
```

图 3.11 给出了只支持 IPv4 的 inet_pton 函数的简单定义。类似地，图 3.12 给出了只支持 IPv4 的 inet_ntop 函数的简化版本。

```
10 int
11 inet_pton(int family,const char *strptr,void *addrptr)
12 {
13     if(family == AF_INET) {
14         struct in_addr in_val;
15         if(inet_aton(strptr,&in_val)) {
16             memcpy(addrptr,&in_val,sizeof(struct in_addr));
17             return(1);
18         }
19         return(0);
20     }
21     errno = EAFNOSUPPORT;
22     return (-1);
23 }
```

图 3.11　仅支持 IPv4 的 inet_pton 简化版本[libfree/inet_pton_ipv4.c]

```
 8 const char *
 9 inet_ntop(int family,const void *addrptr,char *strptr,size_t len)
10 {
11     const u_char *p = (const u_char *) addrptr;
12     if(family == AF_INET) {
13         char    temp[INET_ADDRESTRLEN];
14         snprintf(temp,sizeof(temp),"%d.%d.%d.%d",
15                 p[0],p[1],p[2],p[3]);
16         if(strlen(temp) >= len) {
17             errno = ENOSPC;
18             return (NULL);
19         }
20         strcpy(strptr,temp);
21         return(strptr);
22     }
```

```
23      errno = EAFNOSUPPORT;
24      return (NULL);
25  }
```

图 3.12 仅支持 IPv4 的 inet_ntop 简化版本[libfree/inet_ntop_ipv4.c]

3.8 sock_ntop 和相关函数

inet_ntop 的一个基本问题是:它要求调用者传递一个指向二进制地址的指针,而此地址一般是包含在套接口地址结构中的,这就要求调用者必须知道结构的格式和地址族。这就是说,为使用这个函数,我们必须为 IPv4 写如下的代码:

```
struct sockaddr_in addr;
inet_ntop(AF_INET, &addr.sin_addr, str, sizeof(str));
```

或为 IPv6 写如下的代码:

```
struct sockaddr_in6 addr6;
inet_ntop(AF_INET6, &addr6.sin6_addr, str, sizeof(str));
```

这就使得我们的代码与协议相关了。

为了解决这个问题,我们自己写一个函数 sock_ntop,它取得作为参数传递的指向套接口地址结构的指针后,查看结构的内部,再调用适当的函数来返回地址的表达格式。

```
#include "unp.h"
char *sock_ntop(const struct sockaddr *sockaddr, socklen_t addrlen);
                                    返回:非空指针——成功,NULL——出错
```

本书通篇使用的我们自己定义的函数(不是标准的系统函数)都有这样的表现形式:包围函数原型和返回值的方框是实线,开头包括的头文件一般是 unp.h。

sockaddr 指向一个长度为 addrlen 的套接口地址结构。本函数用它自己的静态缓冲区来保存结果,一个指向此缓冲区的指针即为返回值。

注意:对结果进行静态存储阻碍了函数的可重入与线程安全性。对此,我们在 11.14 节中将作进一步讨论。我们对此函数作这样的设计决策是为了在本书的简单例子中方便地调用它。

表达格式是:IPv4 地址的点分十进制数或 IPv6 地址的十六进制数串,后跟一个终止符(我们使用一个点号,与 netstat 类似),再跟一个十进制的端口号,最后跟一个空字符。因此,缓冲区大小至少是:IPv4 时为 INET_ADDRSTRLEN 加上 6 字节(16+6=22),IPv6 时为 INET6_ADDRSTRLEN 加上 6 字节(46+6=52)。

图 3.13 中我们给出了仅为 AF_INET 情形下的源代码。

```
5 char *
6 sock_ntop(const struct sockaddr *sa, socklen_t salen)
7 {
8   char        portstr[7];
9   static      char str[128];          /* Unix domain is largest */
10  switch (sa->sa_family) {
11  case AF_INET:{
12          struct sockaddr_in *sin = (struct sockaddr_in *) sa;

13          if(inet_ntop(AF_INET, &sin->sin_addr,str,sizeof(str)) == NULL)
14            return (NULL);
15          if(ntohs(sin->sin_port) != 0) {
16            snprintf(portstr,sizeof(portstr),".%d",ntohs(sin->sin_port));
17            strcat(str,portstr);
18          }
19          return (str);
20      }
```

图 3.13 sock_ntop 函数[lib/sock_ntop.c]

我们还为操作套接口地址结构定义了几个其他的函数，它们将简化我们的代码在 IPv4 与 IPv6 间的移植。

```
#include "unp.h"
int sock_bind_wild(int sockfd, int family);
                                                    返回：0——成功，-1——出错
int sock_cmp_addr(const struct sockaddr *sockaddr1,
                  const struct sockaddr *sockaddr2,socklen_t addrlen);
                                                    返回：0——地址同族且相等，否则非0
int sock_cmp_port(const struct sockaddr *sockaddr1,
                  const struct sockaddr *sockaddr2, socklen_t addrlen);
                                                    返回：0——地址同族且端口相同，否则非0
int sock_get_port(const struct sockaddr *sockaddr, socklen_t addrlen);
                                    返回：非负端口号 sockaddr 为 IPv4 或 IPv6 地址族，否则-1
char *sock_ntop_host(const struct sockaddr *sockaddr, socklen_t addrlen);
                                                    返回：非空指针——成功，空指针——出错
void sock_set_addr(const struct sockaddr *sockaddr, socklen_t addrlen,void *ptr);
void sock_set_port(const struct sockaddr *sockaddr, socklen_t addrlen,int port);
void sock_set_wild(struct sockaddr *sockaddr, socklen_t addrlen);
```

sock_bind_wild 捆绑通配地址和一个临时端口到一个套接口；sock_cmp_addr 比较两个套接口地址结构的地址部分；sock_cmp_port 比较两个套接口地址结构的端口部分；sock_get_port 返回端口号；sock_ntop_host 将套接口地址结构中的主机部分转换成表达格式(不包括端口号)；sock_set_addr 将套接口地址结构中的地址部分置为指针 ptr 所指的值；sock_set_port 只设置套接口地址结构的端口号；sock_set_wild 将套接口地址结构中的地址

部分置为通配地址。跟本书所有函数一样，我们也为那些返回值不是 void 的上述函数提供了包裹函数，它们的名字以 S 打头，我们的程序通常调用这些包裹函数。这里我们就不给出所有这些函数的源代码了，它们是免费提供的(见前言)。

3.9 readn、writen 和 readline 函数

字节流套接口(如 TCP 套接口)上的 read 和 write 函数所表现的行为不同于通常的文件 I/O。字节流套接口上的读或写输入或输出的字节数可能比要求的数量少，但这不是错误状况，原因是内核中套接口的缓冲区可能已达到了极限。此时所需的是调用者再次调用 read 或 write 函数，以输入或输出剩余的字节。(有些版本的 Unix 在往一个管道中写多于 4096 字节的数据时也会表现这样的行为)，这种情况在读字节流套接口时很常见，但在写字节流套接口时只能在套接口非堵塞的情况下才出现。然而，为预防实现万一返回不足的字节计数值，我们总是调用 writen 函数而不是 write 函数。

当我们对字节流套接口进行读或写操作时，调用下面的三个函数。

```
#include "unp.h"
ssize_t readn(int filedes, void *buff, size_t nbytes);
ssize_t writen(int filedes, const void *buff, size_t nbytes);
ssize_t readline(int filedes, void *buff, size_t maxlen);
                                        均返回：读写字节数，-1——出错
```

图 3.14 给出了 readn 函数，图 3.15 给出了 writen 函数，图 3.16 给出了 readline 函数。

```
1 #include        "unp.h"
2 ssize_t         /* Read "n" bytes from a descriptor. */
3 readn(int fd, void *vptr, size_t n)
4 {
5     ssize_t nleft;
6     ssize_t nread;
7     char    *ptr;
8     ptr = vptr;
9     nleft = n;
10    while (nleft > 0) {
11        if( (nread = read(fd, ptr, nleft)) < 0) {
12            if (errno == EINTR)
13                nread = 0;          /* and call read() again */
14            else
15                return (-1);
16        } else if (nread == 0)
17            break;              /* EOF */
18        nleft -= nread;
19        ptr += nread;
20    }
21    return (n - nleft);         /* return >= 0 */
22 }
```

图 3.14 readn 函数：从一个描述字读 n 字节[lib/readn.c]

```
1 #include       "unp.h"
2 ssize_t                /* Write "n" bytes to a descriptor. */
3 writen(int fd,const void *vptr,size_r n)
4 }
5         size_t nleft;
6         ssize_t nwritten;
7         const char *ptr;
8         ptr = vptr;
9         nleft = n;
10        while (nleft > 0) {
11            if( (nwritten = write(fd,ptr,nleft)) <= 0) {
12                if(errno == EINTR)
13                    nwritten = 0;     /* and call write() again */
14                else
15                    return (-1);      /* error */
16            }
17            nleft -= nwritten;
18            ptr += nwritten;
19    }
20    return (n);
21 }
```

图 3.15　writen 函数：往一个描述字写 n 字节[lib/writen.c]

```
1 #include     "unp.h"
2 ssize_t
3 readline(int fd,void *vptr,size_t maxlen)
4 {
5     ssize_t n,rc;
6     char    c,*ptr;
7     ptr = vptr;
8     for (n = 1; n < maxlen; n++) {
9       again:
10        if ( (rc = read(fd,&c,1)) == 1) {
11            *ptr++ = c;
12            if(c == '\n')
13                break;          /* newline is stored,like fgets() */
14            } else if (rc == 0) {
15                if (n == 1)
16                    return (0); /* EOF,no data read */
17                else
18                    break;      /* EOF,some data was read */
19        } else {
20            if(errno == EINTR)
21                goto again;
22            return(-1);         /* error,errno set by read() */
23        }
24    }
25    *ptr = 0;                   /* null terminate like fgets() */
26    return (n);
27 }
```

图 3.16　readline 函数：从一个描述字读文本行，一次 1 个字节[test/readline1.c]

上述三个函数查找错误EINTR(系统调用被一个捕获的信号中断,我们在5.9节将进行更详细的讨论),如果发生这种错误则继续进行读或写操作。由于这些函数的作用是避免让调用者来处理不足的字节计数值,所以我们就地处理错误,而不是强制调用者再次调用readn或writen函数。

在13.3节我们会提到,标志MSG_WAITALL可随函数recv一起使用来代替独立的readn函数。

注意,函数readline每读一个字节的数据就要调用一次系统的read函数,其效率是非常低的,如APUE的3.9节所示。我们宁愿调用函数read来取得尽可能多的数据,将其缓冲起来,然后在缓冲区中每次检查一个字节。这可以通过标准I/O库来实现,如13.8节中所述。

图3.17给出了函数readline的一个较快速版本,它每次最多读MAXLINE个字节,然后一次返回一个字符。

```
1 #include    "unp.h"
2 static ssize_t
3 my_read(int fd,char *ptr)
4 {
5     static int read_cnt = 0;
6     static char *read_ptr;
7     static char read_buf(MAXLINE);
8     if (read_cnt <= 0) {
9         again:
10          if((read_cnt = read(fd,read_buf,sizeof(read_buf))) < 0) {
11              if(errno == EINTR)
12                  goto again;
13              return (-1);
14          } else if(read_cnt == 0)
15              return (0);
16          read_ptr = read_buf;
17      }
18      read_cnt--;
19      *ptr = *read_ptr++;
20      return (1);
21 }
22 ssize_t
23 readline(int fd,void *vptr,size_t maxlen)
24 {
25      int        n,rc;
26      char       c,*ptr;
27      ptr = vptr;
28      for (n = 1; n < maxlen; n++) {
29          if( (rc = my_read(fd,&c)) ==1 ) {
30              *ptr++ = c;
31              if(c == '\n')
32                  break;          /* newline is stored,like fgets() */
33          } else if (rc == 0) {
34              if (n == 1)
35                  return (0);     /* EOF,no data read */
36              else
37                  break;          /* EOF,some data was read */
38              } else
```

```
39                  return (-1); /* error, errno set by read() */
40      }
41      *ptr = 0;               /* null terminate like fgets() */
42      return (n);
43  }
```

图 3.17　函数 readline 的改进版[lib/readline.c]

第 2～21 行　内部函数 my_read 每次最多读 MAXLINE 个字符，然后每次返回一个字符。

第 29 行　函数 readline 本身的唯一变化是调用函数 my_read 而不是 read。

对函数 readline 做的这个小小的修改使得情况大为改观。如果我们测量一下新老版本读一个 2781 行的文件(135 816 字节)所需时间就会发现，老版本需要 8.8 秒，而新版本只需 0.3 秒。几乎所有的区别都由内核中所耗时间即系统时间引起，老版本执行了 135 816 次系统调用，而新版本仅进行了 34 次系统调用(135 816 除以 MAXLINE 是 4096)。

> 不幸的是，在函数 my_read 中用静态变量来维护会跨越连续调用的状态信息，因而不再是可重入或线程安全的了。在 11.14 节和 23.5 节中，我们将讨论这一点，在图 23.11 中，我们用特定于线程的数据开发了一个此函数的线程安全版本。

3.10　isfdtype 函数

很多时候，我们需要测试一个描述字是不是某给定类型。早些时候，这是通过调用 Posix.1g 的函数 fstat，然后用 S_ISxxx 的某个宏来测试返回的 st_mode 值来实现的。有许多实现，但不是所有的实现，定义了宏 S_ISSOCK 来测试一个描述字是否为套接口描述字。由于有些实现仅靠函数 fstat 的返回信息不能判定一个描述字是否为套接口描述字，因而 Posix.1g 提供了新函数 isfdtype。

```
#include <sys/stat.h>
int isfdtype(int fd, int fdtype);
                    返回：1——是指定类型，0——不是指定类型，-1——出错
```

为了测试是否为套接口描述字，fdtype 应设成 S_IFSOCK。此函数的一个应用是：在一个由另外一个程序调用 exec 执行的程序中(4.7 节)，测试某描述字是否为一个套接口描述字。

图 3.18 给出了这个函数的一个实现例子(假设该实现支持 fstat 返回的 S_IFSOCK 模式)。

```
1 #include    "unp.h"
2 #ifndef S_IFSOCK
3 #error S_IFSOCK not defined
4 #endif
5 int
```

```
 6 lsfdtype(int fd,int fdtype)
 7 {
 8         struct stat buf;
 9         if (fstat(fd,&buf) < 0)
10             return (-1);
11         if ((buf.st_mode & S_IFMT) == fdtype)
12             return (1);
13     else
14             return (0);
15 }
```

图 3.18 用 fstat 实现的 isfdtype[lib/isfdtype.c]

在头文件<sys/stat.h>中定义了大量的 S_IFxxx 常值，我们的实现也认可它们。然而，Posix.1g 只说明当 fdtype 是 S_IFSOCK 时，该函数有效。

3.11 小 结

套接口地址结构是每个网络程序的重要组成部分，我们分配它们，填写它们，把指向它们的指针传递给各种套接口函数，有时我们把指向某结构的指针传递给套接口函数并由它们填写其内容。我们总是通过指针来传递这些结构(也就是说，我们传递的是指向结构的指针，而不是结构本身)，而且将结构的大小作为另外一个参数来传递。当套接口函数填写结构时，长度也作为指针传递，所以它的值可以被函数更新，我们把这样的参数称为值-结果参数。

套接口地址结构总是以一个标识含在结构中的地址族的成员("族")开头，所以套接口地址结构是自定义的。支持变长套接口地址结构的较新实现在开头还包含一个长度成员，它含有整个结构的长度信息。

在表达格式(我们平时所写的格式，如 ASCII 字符)与数值格式(套接口地址结构中的格式)间转换 IP 地址的两个函数是：inet_pton 和 inet_ntop。虽然在后面的章节中我们要使用这两个函数，但必须说明，它们是协议相关的。一个较好的技术是：把套接口地址结构作为不透明对象来操作，仅需知道指向结构的指针和它的大小，为此我们开发了一组 sock_函数来帮助我们的程序做到协议无关。在第 11 章，我们用函数 getaddrinfo 和 getnameinfo 完成协议无关工具的开发。

TCP 套接口为应用进程提供了一个字节流，它们没有记录标记。从函数 read 的返回值可以比我们要求的数量少，但这不表示错误。我们开发了 3 个函数：readn、writen 和 readline 来协助对字节流的读或写，这 3 个函数广泛应用于全书。

3.12　习　题

3.1　为什么诸如套接口地址结构的长度这样的值-结果参数要用指针来传递？

3.2　为什么函数 readn 和 writen 都将 void * 型指针转换为 char * 型指针？

3.3　函数 inet_aton 和 inet_addr 对于接受什么为点分十进制数 IPv4 地址串一直是相当随意的：允许是由小数点分隔的 1～4 个数，允许一个前导的 0x 来表示是一个十六进制数，还允许一个前导的 0 来表示是一个八进制数。（尝试用 telnet 0xe 来检验一下这些特性。）函数 inet_pton 对 IPv4 地址的要求则要严格得多，它明确要求用三个小数点来分隔四个数，每个数都是 0～255 之间的十进制数。当地址族为 AF_INET6 时，函数 inet_pton 不允许指定点分十进制数地址。对这一点，可能有人要争辩：这种做法应该是允许的，返回值就是对应这个点分十进制数串的 IPv4 映射的 IPv6 地址（图 A.10）。试写一个名为 inet_pton_loose 的函数，它能处理这样的情况：如果地址族是 AF_INET 且函数 inet_pton 返回 0，则调用函数 inet_aton 并看它是否成功；类似地，如果地址族是 AF_INET6 且函数 inet_pton 返回 0，则调用函数 inet_aton 且看它是否成功，若成功则返回其 IPv4 映射的 IPv6 地址。

第4章 基本TCP套接口编程

4.1 概 述

本章阐述编写一个完整的TCP客户和服务器程序所需要的基本套接口函数。我们首先对即将使用的所有基本套接口函数进行阐述，在下一章再开发客户和服务器程序。你会发现，全书都围绕着此客户和服务器程序，并多次改进与完善(图1.12和图1.13)。

我们也对并发服务器进行阐述，这是一项常用的Unix技术，在同时有大量的客户连接到同一个服务器上时提供并发性。每个客户连接都迫使服务器为它派生(fork)一个新进程。本章我们只考虑利用fork的每客户单进程模型，在第23章我们讨论线程时，将考虑另外一种模型，即每客户单线程。

图4.1为一个TCP客户与服务器间发生的一些典型事件的时间表。首先启动服务器，稍后的某个时刻启动客户，它要连接到此服务器上。我们假设客户给服务器发了一个请求，服务器处理这个请求，并且给客户发回一个响应。这个过程一直持续下去，直到客户给服务器发一个文件结束符，并关闭客户端连接，接着服务器也关闭服务器端的连接，或结束运行或等待一个新的客户连接。

4.2 socket函数

为了执行网络I/O，一个进程必须做的第一件事情就是调用socket函数，指定期望的通信协议类型(使用IPv4的TCP、使用IPv6的UDP、Unix域字节流协议等)。

```
#include <sys/socket.h>
int socket(int family, int type, int protocol);
                                返回：非负描述字——成功，-1——出错
```

代码中的family指明协议族，它是图4.2中所示的某个常值。套接口的类型type是图4.3中所示的某个常值。一般来说，函数socket的参数protocol设置为0，除非用在原始套接口上，这一点我们将在第25章中再讨论。

并非所有套接口family与type的组合都是有效的，图4.4给出了一些有效的组合和对应的真正协议。其中标为“Yes”的项也是有效的，但还没有找到便捷的缩略词；而空白项则是不支持的。

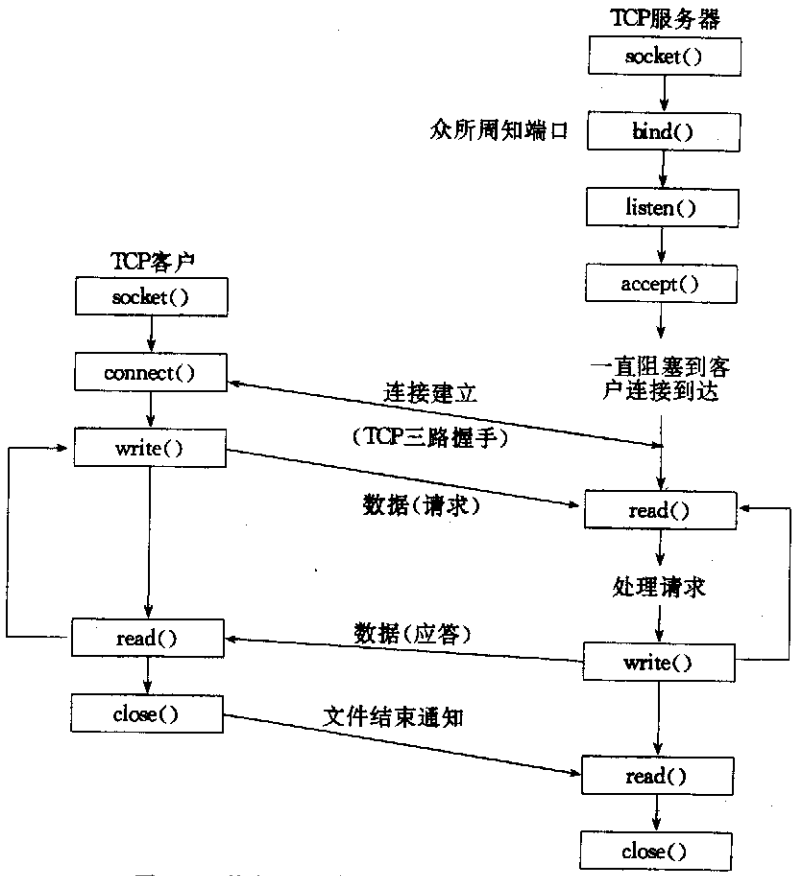


图 4.1　基本 TCP 客户-服务器程序的套接口函数

族	解释
AF_INET	IPv4 协议
AF_INET6	IPv6 协议
AF_LOCAL	Unix 域协议(第 14 章)
AF_ROUTE	路由套接口(第 17 章)
AF_KEY	密钥套接口

图 4.2　socket 函数的协议族(family)常值

类型	解释
SOCK_STREAM	字节流套接口
SOCK_DGRAM	数据报套接口
SOCK_RAW	原始套接口

图 4.3　socket 函数的套接口类型(type)

	AF_INET	AF_INET6	AF_LOCAL	AF_ROUTE	AF_KEY
SOCK_STREAM	TCP	TCP	Yes		
SOCK_DGRAM	UDP	UDP	Yes		
SOCK_RAW	IPv4	IPv6		Yes	Yes

图 4.4　socket 函数的族与类型的组合

你可能也会碰到作为 socket 函数第一个参数的相应的 PF_xxx 常值，我们在本节未讲述。

你也许会碰到 AF_UNIX（原来 Unix 的名字）而不是 AF_LOCAL（Posix. 1g 名字），在第 14 章我们再做详细说明。

参数 family 和 type 还有其他值，例如 4. 4BSD 对 AF_NS（Xerox NS 协议，常称为 XNS）和 AF_ISO（OSI 协议）都支持，但现在很少有人使用这些协议。类似地，Xerox NS 协议和 OSI 协议都实现了 SOCK_SEQPACKET 这种有序分组套接口类型，但是 TCP 是一个字节流，仅支持 SOCK_STREAM 套接口。

Linux 支持一个新的套接口类型：SOCK_PACKET，它与图 2.1 中的 BPF 和 DLPI 类似，支持对数据链路的访问，在第 26 章中我们再做详细说明。

密钥套接口 AF_KEY 是新出现的。IPv6 要求支持基于加密的安全性，许多系统也可能给 IPv4 提供这样的支持。跟路由套接口（AF_ROUTE）是与内核中路由表的接口方式类似，密钥套接口是与内核中密钥表的接口，这个协议族的初级文档见[McDonald，Phan，and Atkinson 1996]和[McDonald，Metz and Phan 1997]。

socket 函数在成功时返回一个小的非负整数值，它与文件描述字类似，我们把它称为套接口描述字（socket descriptor），简称套接字（sockfd）。为了得到这个套接口描述字，我们只是指定了协议族（IPv4、IPv6 或 Unix）和套接口类型（字节流、数据报或原始套接口）。我们并没有指定本地协议地址或远程协议地址。

AF_xxx 与 PF_xxx

AF_前缀代表地址族，PF_前缀代表协议族。历史上曾有这样的想法：单个协议族可以支持多个地址族，PF_值用来创建套接口，而 AF_值用于套接口地址结构。但实际上，支持多个地址族的协议族从来就未实现过，而且头文件<sys/socket. h>中为一给定协议定义的 PF_值总是与此协议的 AF_值相等。尽管这种相等关系并不保证永远正确，若有人试图给已有的协议改变这种约定，则许多现存代码都将崩溃。为与现存代码保持一致，本书中我们仅用 AF_常值（尽管在调用 socket 时，我们可能会碰到 PF_值）。

查看 BSD/OS2. 1 版中调用 socket 的 137 个程序，可以发现，有 143 个调用指定 AF_值，仅有 8 个调用指定 PF_值。

从历史上说，AF_前缀与 PF_前缀具有相似常值集的原因要追溯到4. 1cBSD [Lanciani 1996]和比我们正描述的要早些的 socket 函数版本。socket 的4. 1cBSD 版本采用了四个参数，其中有一个是指向 sockproto 结构的指针。该结构的第一个元素名为 sp_family，它的值是某一 PF_值；第二个元素即 sp_protocol 是一个协议号，与现行 socket 函数的第三个参数相似。指定协议族的唯一方法就是指定该结构，因此，在该早期系统中，PF_值用作在 sockproto 结构中指定协议族的结构标签，而 AF_值用作在套接口地址结构中指定地址族的结构标签。4. 4BSD 中仍有 sockproto 结构（TCPv2 第 626～627 页），但仅由内核内部使用。在最初的定义中，对元素 sp_family 有“protocol family（协议族）”的注释，在4. 4BSD源代码中已改为“address family（地址族）”了。

更让人混淆 AF_常值和 PF_常值区别的是，含有与 socket 函数的第一个参数进行比较的值的 Berkeley 内核数据结构(domain 结构的元素 dom_family，TCPv2 第 187 页)有这样的注释：它含有 AF_值。有些内核中的 domain 结构初始化为相应的 AF_值(TCPv2 第 192 页)，但有些则初始化成 PF_值(TCPv2 第 646 页和 TCPv3 第 229 页)。

另一个必须注意的历史是，1983 年 7 月的 4.2BSD 中 socket 函数的手册页面将其第一个参数称为 af，并列出了作为 AF_常值的可能值。

最后，我们注意到 Posix.1g 将 socket 函数的第一个参数指定为 PF_值，而 AF_值用于套接口地址结构，但它在结构 addrinfo(11.2 节)中又只定义了一个族值，试图让 socket 调用和套接口地址结构都能使用。

4.3　connect 函数

TCP 客户用 connect 函数来建立一个与 TCP 服务器的连接。

```
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
                                        返回：0——成功，-1——出错
```

sockfd 是由 socket 函数返回的套接口描述字，第二、第三个参数分别是一个指向套接口地址结构的指针和该结构的大小，如 3.3 节所述。套接口地址结构必须含有服务器的 IP 地址和端口号，图 1.5 中我们见过此函数的一个例子。

客户在调用函数 connect 前不必非得调用函数 bind(下一节我们再介绍此函数)，因为如果必要的话，内核会选择源 IP 地址和一个临时的端口。

如果是 TCP 套接口的话，函数 connect 激发 TCP 的三路握手过程(2.5 节)，且仅在连接建立成功或出错时才返回，返回的错误可能有以下几种情况：

1. 如果 TCP 客户没有收到 SYN 分节的响应，则返回 ETIMEDOUT。例如在 4.4BSD 中，当调用函数 connect 时，发出一个 SYN，若无响应，等待 6 秒钟后再发一个，若仍无响应，24 秒钟后再发一个(TCPv2 第 828 页)。若总共等了 75 秒钟之后仍未收到响应，则返回错误。

   有些系统提供对超时值的管理性控制，见 TCPv1 的附录 E。
2. 如果对客户的 SYN 的响应是 RST，则表明该服务器主机在我们指定的端口上没有进程在等待与之连接(例如服务器进程也许没有启动)，这称为硬错(hard error)，客户一接收到 RST，马上就返回错误 ECONNREFUSED。

   RST 的含义为“复位”，它是 TCP 在某些错误情况下所发的一种 TCP 分节。有三个条件可以产生 RST：SYN 到达某端口但此端口上没有正在监听的服务器(如前所述)、TCP 想取消一个已有连接、TCP 接收了一个根本不存在的连接上的分节。(TCPv1 第 246～250 页有更详细的信息。)
3. 如果某客户发出的 SYN 在中间的路由器上引发了一个目的地不可达 ICMP 错误，

这称为软错(soft error)。客户上的内核保存此消息,并按第一种情况中所述的时间间隔连续发出SYN。若在某规定的时间(4.4BSD规定75秒)后仍未收到响应,则把保存的消息(即ICMP错误)作为EHOSTUNREACH或ENETUNREACH错误返回给进程。

许多早期系统,如4.2BSD,在收到目的地不可达ICMP错误时就不正确地放弃建立连接的尝试,这是不正确的,因为此ICMP错误可能只是指示了某个暂时状态。例如,它可能是一个15秒内可以恢复的路由问题。

注意,即使ICMP错误指示目的网络不可达,图A.15中也没有列出ENETUNREACH。网络不可达的错误被认为已过时,即使4.4BSD收到此消息,它也是给应用进程返回EHOSTUNREACH错误。

用图1.5的简单客户程序例子,我们再来仔细看看上述几种不同的错误情况。首先指定本地主机(127.0.0.1),它在运行daytime服务器程序,我们观察正常的输出:

```
solaris % daytimetcpcli 127.0.0.1
Tue Jan 16 16:45:07 1996
```

为了看到返回响应的不同格式,我们指向本地Cisco路由器(见图1.16):

```
solaris % daytimetcpcli 206.62.226.62
Tuesday, May 7, 1996 11:01:33-MST
```

下面,我们将IP地址指向本地子网(206.62.226)但其主机ID(55)并不存在,即本地子网上没有一个主机ID为55的主机,当客户发出ARP请求(要求主机响应其硬件地址)时,将收不到ARP响应:

```
solaris % daytimetcpcli 206.62.226.55
connect error: Connection timed out
```

当函数connect超时后(Solaris 2.5规定此时间为3分钟),我们才得到此错误。函数err_sys输出与ETIMEDOUT相关的人们可读的字符串。

下一个例子中我们指向主机gateway,它是一个Cisco路由器,没有运行daytime服务器程序。

```
solaris % daytimetcpcli 140.252.1.4
connect error:Connection refused
```

服务器主机立刻响应一个RST。

最后一个例子,我们指向Internet中一个不可到达的IP地址。如果我们用tcpdump观察分组的情况,就会发现6跳以远的路由器返回了主机不可达的ICMP错误。

```
solaris % daytimetcpcli 192.3.4.5
connect error:No route to host
```

跟ETIMEDOUT错误一样,本例中的connect也在等待规定的一段时间之后才返回EHOSTUNREACH错误。

根据TCP状态转换图(图2.4),函数connect导致从CLOSED状态(调用函数socket创建套接口后就一直处于此状态)转到SYN_SENT状态,若成功再转到ESTABLISHED状

态。如果函数 connect 失败，则套接口不可再用，必须关闭，不能再对此套接口再调用函数 connect。在图 11.6 中我们将看到，当循环调用函数 connect，以尝试给定主机的每个 IP 地址直到有一个成功时，每当函数 connect 失败，都必须关闭套接口描述字，重新调用 socket。

4.4　bind 函数

函数 bind 给套接口分配一个本地协议地址，对于网际协议，协议地址是 32 位 IPv4 地址或 128 位 IPv6 地址与 16 位的 TCP 或 UDP 端口号的组合。

```
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
                                                返回：0——成功，-1——出错
```

历史上对函数 bind 的手册页面描述是这样的：函数 bind 为一个无名的套接口命名。“名字”的意义比较含糊，它可能意味诸如 foo.bar.com 之类的域名（第 9 章）。实际上，函数 bind 与名字没有关系。它仅仅给套接口分配一个协议地址，至于协议地址的含义则要取决于协议本身①。

第二个参数是一个指向特定于协议的地址结构的指针，第三个参数是该地址结构的长度。对于 TCP，调用函数 bind 可以指定一个端口号，指定一个 IP 地址，可以两者都指定，也可以一个也不指定。

- 当服务器启动时，要捆绑众所周知端口，这在图 1.9 中我们已看到了。如果 TCP 客户或服务器不这么做，当调用函数 connect 或 listen 时，内核就要为套接口选择一个临时端口。让内核来选择临时端口，这对 TCP 客户来说是正常的，除非应用要求一个预留端口；但对 TCP 服务器来说是极少见的，因为服务器是通过它们的众所周知端口被大家认识的。

  这个规则的例外是 RPC（远程过程调用）服务器，它们一般是让内核来为它们的监听套接口选择一个临时端口，因为这个端口紧接着就通过 RPC 端口映射器进行注册。客户在与服务器连接之前，必须与端口映射器联系以获取服务器

① 译者注：捆绑（binding）操作涉及三个对象：套接口（在 XTI API 中为端点）、地址及端口。其中套接口是捆绑的主体，地址和端口则是捆绑在套接口上的客体。由于涉及对象较多，我们先在这里澄清各种说法：

1. “捆绑地址 A 和/或端口 P 到套接口 S”。同义的说法还有：“把地址 A 和/或端口 P 捆绑到套接口 S”，“给套接口 S 捆绑地址 A 和/或端口 P”，等等。
2. “跟端口 P（地址 A）一块捆绑地址 A（端口 P）”。

绑定（bound）表示捆绑成功后的状态，它的各种说法如下：

1. “绑定地址 A 和/或端口 P 的套接口”。
2. “套接口 S 上绑定的地址或端口”。
3. “已绑定的地址或端口”。也就是说该地址或端口已为某个套接口所用。
4. “跟端口 P（地址 A）一块绑定的地址（端口）”。
5. “套接口 S 已绑定”。相反的说法是“套接口 S 未绑定”。

的临时端口。这种情况也适用于使用 UDP 的 RPC 服务器。

- 进程可以把一个特定的 IP 地址捆绑到它的套接口上，但此 IP 地址必须是主机的一个接口。对于 TCP 客户，这就为在此套接口上发送的 IP 数据报分配了源 IP 地址。对于 TCP 服务器，这就限制了套接口只接收那些目的地为此 IP 地址的客户连接。TCP 客户一般不把 IP 地址捆绑到它的套接口上。当连接套接口时，由内核根据所用的输出接口来选择源 IP 地址，而所用的输出接口则取决于到达服务器的路径(TCPv2 第 737 页)。

  如果 TCP 服务器不把 IP 地址捆绑到套接口上，内核就把客户所发 SYN 所在分组的目的 IP 地址作为服务器的源 IP 地址(TCPv2 第 943 页)。

正如我们所说的，通过调用函数 bind 可以指定 IP 地址或端口，可以都指定，也可以都不指定。根据期望的结果，对 sin_addr 和 sin_port，或 sin6_addr 和 sin6_port 应置为什么值，图 4.5 作了总结。

进程指定		结果
IP 地址	端口	
通配地址	0	内核选择 IP 地址和端口
通配地址	非 0	内核选择 IP 地址，进程指定端口
本地 IP 地址	0	进程指定 IP 地址，内核选择端口
本地 IP 地址	非 0	进程指定 IP 地址和端口

图 4.5 给函数 bind 指定用于捆绑的 IP 地址和/或端口号的结果

若指定端口号为 0，调用函数 bind 时，内核选择一个临时端口；但若指定一个通配 IP 地址，则直到套接口已连接(TCP)或数据报已在套接口上发出(UDP)，内核才选择一个本地 IP 地址。

对于 IPv4 来说，通配地址由常值 INADDR_ANY 来指定，其值一般为 0，它通知内核选择 IP 地址。在图 1.9 中我们已看到了如下的赋值用法：

```
struct sockaddr_in servaddr;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);        /* wildcard */
```

这对 IPv4 是可行的，因为其地址是 32 位的值，可以用一个简单的数字常值来表示(这种情况下为 0)，但在 IPv6 中，我们就不能利用这项技术，因为 128 位的 IPv6 地址是保存在结构中的。(在 C 语言中，赋值语句的右边无法表示常值结构。)为解决此问题，我们编写如下代码：

```
struct sockaddr_in6  serv;
serv.sin6_addr = in6addr_any;      /* wildcard */
```

系统分配变量 in6addr_any 并将其初始化为常值 IN6ADDR_ANY_INIT，头文件<netinet/in.h>中含有 in6addr_any 的 extern 声明。

INADDR_ANY 的值(为 0)无论是网络字节序还是主机字节序都相同，所以不必使用 htonl。但既然头文件<netinet/in.h>定义的所有 INADDR_常值都是主机字节序，我们应该

使用 htonl。

如果让内核来为套接口选择一个临时端口号，那么必须注意，函数 bind 并不返回所选择的值。实际上，由于函数 bind 的第二个参数有 const 限定词，它无法返回所选之值。为了得到内核所选择的这个临时端口值，必须调用函数 getsockname 来返回协议地址。

进程捆绑非通配 IP 地址到套接口上的常见例子是在为多个组织提供 Web 服务器的主机上(TCPv3 的 14.2 节)。首先，每个组织都有它自己的域名，就像这样的形式：www.organization.com。其次，每个组织的域名都映射为不同的 IP 地址，但在相同的子网上。例如，子网为 198.69.10，第一个组织的 IP 地址是 198.69.10.128，第二个是 198.69.10.129，等等。然后，所有这些 IP 地址都定义成同一个网络接口的别名(例如，4.4BSD 中使用 ifconfig 命令的 alias 选项来定义)，这样，IP 层将接收所有目的地为某个别名地址的数据报。最后，为每个组织启动一个 HTTP 服务器的副本，每个副本仅捆绑自己组织的 IP 地址。

> 另有一项技术是运行单个服务器但捆绑通配地址。当一个连接请求到达时，服务器调用函数 getsockname 获得来自客户的目的 IP 地址，这在我们上面的讨论中，可以是 198.69.10.128，198.69.10.129 等等。然后，服务器根据这个 IP 地址来处理客户请求。
>
> 捆绑非通配 IP 地址的好处是：由内核将给定的目的 IP 地址解复用后送往服务器进程。
>
> 对于分组到达的接口和分组的目的 IP 地址这两个概念，我们必须仔细区别。在 8.8 节，我们将讨论弱端系统模型和强端系统模型。大多数实现都采用前者，其含义是：一个分组只要其目的 IP 地址能够标识目的主机的某个接口就行，不必一定是它的到达接口。(假设是多宿主机。)捆绑非通配 IP 地址仅根据目的 IP 地址来限制递送给套接口的数据报，对到达接口未做任何限制，除非主机采用强端系统模型。

从函数 bind 返回的一个常见错误是 EADDRINUSE(地址已使用)，对这一点，到 7.5 节我们讨论套接口选项 SO_REUSEADDR 和 SO_REUSEPORT 时再做详细说明。

4.5　listen 函数

函数 listen 仅被 TCP 服务器调用，它做两件事情：

1. 当函数 socket 创建一个套接口时，它被假设为一个主动套接口，也就是说，它是一个将调用 connect 发起连接的客户套接口，函数 listen 将未连接的套接口转换成被动套接口，指示内核应接受指向此套接口的连接请求。根据 TCP 状态转换图(图 2.4)，调用函数 listen 导致套接口从 CLOSED 状态转换到 LISTEN 状态。
2. 函数的第二个参数规定了内核为此套接口排队的最大连接个数。

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
                                        返回：0——成功，-1——出错
```

一般说来，此函数应在调用函数 socket 和 bind 之后，调用函数 accept 之前调用。

为了理解参数 backlog，我们必须明白，对于给定的监听套接口，内核要维护两个队列：

1. 未完成连接队列(incomplete connection queue)，为每个这样的 SYN 分节开设一个条目：已由客户发出并到达服务器，服务器正在等待完成相应的 TCP 三路握手过程。这些套接口都处于 SYN_RCVD 状态(图 2.4)。
2. 已完成连接队列(completed connection queue)，为每个已完成 TCP 三路握手过程的客户开设一个条目。这些套接口都处于 ESTABLISHED 状态(图 2.4)。

图 4.6 描述了监听套接口的这两个队列。

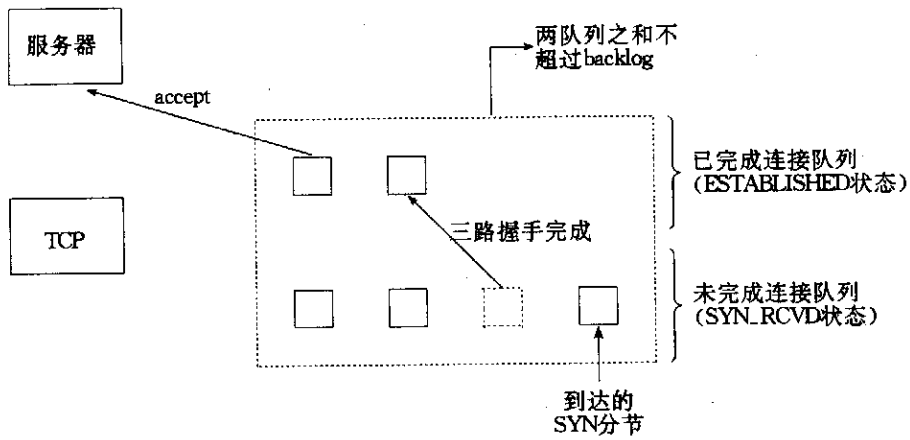


图 4.6 TCP 为监听套接口维护的两个队列

图 4.7 描述了用这两个队列建立连接时所交换的分组。

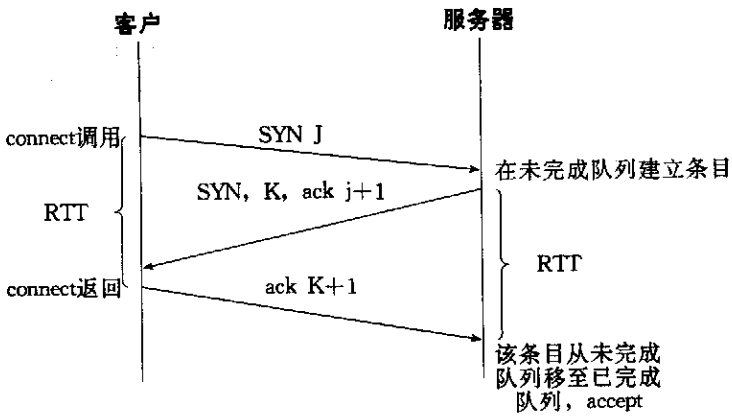


图 4.7 TCP 三路握手和监听套接口的两个队列

当来自客户的 SYN 到达时，TCP 在未完成连接队列中创建一个新条目，然后用三路握手的第二个分节即服务器的 SYN 响应，并附带对客户 SYN 的 ACK(2.5 节)。这个条目一直保留在未完成连接队列中，直到三路握手的第三个分节(客户对服务器 SYN 的 ACK)到达或本条目超时才撤消。(源自 Berkeley 的实现为这些未完成连接条目设置 75 秒的超时时间。)如果三路握手正常完成，该条目将从未完成连接队列搬到已完成连接队列的队尾。当进

程调用函数 accept 时(我们在下一节介绍此函数),已完成连接队列中的队头条目返回给进程,但当队列为空时,进程将睡眠,直到有条目放入已完成连接队列才唤醒它。

处理这两个队列时,必须考虑以下几点:

- 函数 listen 的参数 backlog 曾被规定为两个队列总和的最大值。

  对于 backlog,从未有过正式的定义。4.2BSD 中的手册页面定义它为待处理连接可能增长的最大数目。许多手册页面,甚至 Posix.1g 也将此定义逐字拷贝,但此定义并未解释什么是待处理连接,是处于 SYN_RCVD 状态,处于未接受的 ESTABLISHED 状态,还是处于这两种状态均可。对这个问题的历史性定论是由源自 Berkeley 的实现完成的,可追溯到 4.2BSD,此定义后来被许多其他实现拷贝。

- 源自 Berkeley 的实现给我们说明的 backlog 增加了一个模糊因子:把它乘以 1.5(TCPv1 第 257 页和 TCPv2 第 462 页)。例如,当通常把 backlog 规定为 5 时,在这些系统中实际上却允许最多有 8 个条目在排队,如图 4.10 所示。

  增加此模糊因子的理由已无可考证[Joy 1994],但是如果我们把 backlog 看成是内核能为某套接口排队的最大已完成连接数目([Borman 1997c],稍后讨论),那么增加模糊因子的理由就是把队列中的未完成连接也计算在内。

- 不要把 backlog 定义为 0,因为不同的实现对此有不同的解释(图 4.10),有些实现允许 1 个连接排队,而其他实现不允许有连接排队。如果你不想让任何客户连接到你的监听套接口上,最好关掉该监听套接口。
- 假设三路握手正常完成(例如没有丢失分节,所以不需重传),那么不论客户与服务器间的往返时间(RTT)是多大,未完成队列中的相应条目只保持这段时间。TCPv3 的 14.4 节指出,对于一个 Web 服务器,多对客户与服务器间 RTT 的中间值为 187ms。(由于某些大值对均值影响较大,所以采用中间值作为统计值。)
- 历史上的举例代码常将 backlog 值置为 5,因为这是 4.2BSD 所支持的最大值。20 世纪 80 年代较忙的服务器一天只处理几百个连接,这个值是足够的,但随着万维网(WWW)的发展,较忙的服务器一天要处理几百万个连接,这个小值是根本不够的(TCPv3 第 187～192 页)。较忙的 HTTP 服务器必须指定一个大得多的 backlog 值,且新内核必须支持更大些的值。

  当前的许多系统都允许管理员修改 backlog 的最大值。

- 有个问题:backlog 设置为 5 是常常不够的,那应设为多大呢?这个问题不好回答。现在,HTTP 服务器指定一个较大的值,但如果此值是在源代码中的常值,那就要重新编译服务器程序。另一个方法是假设一个缺省值,但允许设置命令行选项或环境变量来覆盖该缺省值。当指定的值比内核所支持的值要大时,也不受影响,因为内核能把所给的值改为它所支持的最大值且不返回错误(TCPv2 第 456 页)。

通过修改函数 listen 的包裹函数可以解决这个问题,下面给出一个简单的例子,其代码见图 4.8。我们允许环境变量 LISTENQ 来覆盖由调用者所给的值。

```
74 void
75 Listen(int fd,int backlog)
76 {
77          char      *ptr;
78              /* can override 2nd argument with environment variable */
79          if( (ptr = getenv("LISTENQ")) != NULL)
80              backlog = atoi(ptr);
81          if(listen(fd,backlog) < 0)
82              err_sys("listen error");
83 }
```

图 4.8 函数 listen 的包裹函数，允许环境变量指定 backlog 值[lib/wrapsock.c]

- 历史上的手册和书中都指出：仅有固定数目的连接可以排队是为了处理服务器进程在连续的 accept 调用之间处于忙状态的情况。这就意味着两个队列中，已完成连接队列中的条目一般应比未完成连接队列中的条目多。然而，很忙的 Web 服务器已证明这是不对的。指定一个较大 backlog 值的原因是：未完成连接队列中的条目可以随着客户 SYN 分节的到达并等待三路握手的完成而增大。

图 4.9 列出了在某中等忙碌的 Web 服务器上所测出的每个队列条目数的真实值，这些值是这样得到的：在某个工作日，每隔 84ms 便对一个监听 HTTP 套接口采样两个队列的条目数，这样持续 2 个小时。

队列中条目数	未完成队列	已完成队列
0	3 033	90 358
1	7 158	107
2	10 551	59
3	12 960	52
4	11 949	38
5	9 836	27
6	7 754	31
7	6 165	22
8	4 829	30
9	3 687	35
10	2 674	30
11	1 893	25
12	1 431	29
13	1 083	25
14	1 065	49
15	980	7
16	784	
17	696	
18	514	
19	382	
20	294	
21	248	

（续）

队列中条目数	未完成队列	完成队列
22	161	
23	152	
24	121	
25	77	
26	48	
27	33	
28	79	
29	78	
30	90	
31	70	
32	29	
33	16	
34	4	
	90 924	90 924

图 4.9　未完成连接队列和已完成连接队列中的条目数

已完成连接队列 99.4%的时间是空的，但也有一段时间不空。在此服务器上正运行着的系统(BSD/OS 2.0.1)最大的 backlog 值为 64，但列出的值显然未达到此极限。

- 当一个客户 SYN 到达时，若两个队列都是满的，TCP 就忽略此分节(TCPv2 第 930～931 页)，且不发送 RST。这是因为：这种情况是暂时的，客户 TCP 将重发 SYN，期望不久就能在队列中找到空闲条目。要是 TCP 服务器发送了一个 RST，客户的 connect 函数将立即返回一个错误，强制应用进程处理这种情况，而不是让 TCP 正常的重传机制来处理。还有，客户区别不了这两种情况：作为 SYN 的响应，意为“此端口上没有服务器”的 RST 和意为“有服务器在此端口上但其队列满”的 RST。

  Posix.1g 允许以下两种处理方法：忽略新的 SYN，或为此 SYN 响应一个 RST。历史上，所有的源自 Berkeley 的实现都是忽略新的 SYN。

- 三路握手完成之后，服务器调用函数 accept 之前到达的数据应由服务器 TCP 排队，最大数据量为已连接套接口的接收缓冲区大小。

图 4.10 所示为图 1.16 所列的各种操作系统下，参数 backlog 取不同值时已排队连接的实际数目。9 个操作系统分成 6 列，对 backlog 的意义给出各种不同的解释。

backlog	已排列连接实际的最大数目					
	AIX4.2，BSD/OS 3.0	DUnix 4.0，Linux2.0.27，UWare 2.1.2	HP-UX 10.30	SunOS 4.1.4	Solaris 2.5.1	Solaris 2.6
0	1	0	1	1	1	1
1	2	1	1	2	2	3
2	4	2	3	4	3	4

（续）

backlog	已排列连接实际的最大数目					
	AIX4.2，BSD/OS 3.0	DUnix 4.0，Linux2.0.27，UWare 2.1.2	HP-UX 10.30	SunOS 4.14	Solaris 2.5.1	Solaris 2.6
3	5	3	4	5	4	6
4	7	4	6	7	5	7
5	8	5	7	8	6	9
6	10	6	9	8	7	10
7	11	7	10	8	8	12
8	13	8	12	8	9	13
9	14	9	13	8	10	15
10	16	10	15	8	11	16
11	17	11	16	8	12	18
12	19	12	18	8	13	19
13	20	13	18	8	14	21
14	22	14	19	8	15	22

图 4.10 不同 backlog 值时已排列连接的实际数目

AIX、BSD/OS 和 SunOS 4 有传统的 Berkeley 算法，尽管后者（SunOS 4）不允许 backlog 大于 5；HP-UX 和 Solaris 2.6 给 backlog 加了一个模糊因子；Digital Unix、Linux 和 UnixWare 完全照搬 backlog 的原定义，而 Solaris 2.5.1 则对 backlog 加 1。

对于 backlog 为 0 的情况，Linux 允许不受限制的连接数目，这是一个程序缺陷（bug）。

测量这些值的程序见习题 14.5。

我们已提到过，历史上曾把 backlog 值指定为两个队列之和的最大值。在 1996 年间，因特网受到一种称之为 SYN 泛滥（SYN flooding）的新型攻击。黑客写一个程序，它以很高的速率给受害者发送 SYN，装填一个甚至多个 TCP 端口的未完成连接队列。（我们用黑客（hacker）一词来形容攻击者，见[Cheswick and Bellovin 1994]前言部分的描述。）而且，该程序将每个 SYN 的源 IP 地址都置成随机数（称为 IP 欺骗（IP spoofing）），这样服务器的 SYN/ACK 就发往各处，同时也防止了服务器捕获黑客的真实 IP 地址。这样，通过以非法的 SYN 装填未完成连接队列，使合法的 SYN 排不上队，导致合法客户的服务被拒绝（denial of service）。处理这种拒绝服务型攻击通常有两种方法，见[Borman 1997c]中的小结，但这儿我们最感兴趣的是回味一下函数 listen 中 backlog 参数的实际意义，即它应指明为内核将为某套接口排队的最大已完成连接数。对这些已完成的连接规定一个限度，目的是为了防止内核在应用进程无法接收它们时（不管什么原因）还要接受新的连接请求。如果一个系统实现了这一点，如 BSD/OS 3.0，那么应用程序就无需仅仅因为服务器处理太多的客户请求（如一个很忙的 Web 服务器）或保护它不受 SYN 泛滥的便袭而指定一个

巨大的 backlog 值。无论是合法客户还是黑客发出的未完成连接请求，不管数目多大，内核总能处理。但即使是这种情况下，在图 4.9 中我们也可看见，当已完成的连接队列有沉积的条目时(此图中最多增至 15)，SYN 泛滥也会发生，因此传统的值 5 是不够的。

4.6　accept 函数

accept 由 TCP 服务器调用，从已完成连接队列头返回下一个已完成连接(图 4.6)。若已完成连接队列为空，则进程睡眠(假定套接口为缺省的阻塞方式)。

```
#include <sys/socket.h>
int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
                                        返回：非负描述字——OK，-1——出错
```

参数 cliaddr 和 addrlen 用来返回连接对方进程(客户)的协议地址。addrlen 是值-结果参数(3.3 节)：调用前，我们将由 *addrlen 所指的整数值置为由 cliaddr 所指的套接口地址结构的长度，返回时，此整数值即为由内核存在此套接口地址结构内的准确字节数。

如果函数 accept 执行成功，则返回值是由内核自动生成的一个全新描述字，代表与客户的 TCP 连接。当我们讨论函数 accept 时，常把它的第一个参数称为监听套接口(listening socket)描述字(由函数 socket 生成的描述字，用作函数 bind 和 listen 的第一个参数)，把它的返回值称为已连接套接口(connected socket)描述字。将这两个套接口区分开是很重要的。一个给定的服务器常常是只生成一个监听套接口且一直存在，直到该服务器关闭。内核为每个被接受的客户连接创建了一个已连接套接口(也就是说内核已为它完成 TCP 的三路握手过程)。当服务器完成某客户的服务时，关闭已连接套接口。

该函数最多返回三个值：一个既可能是新套接口描述字也可能是错误指示的整数、一个客户进程的协议地址(由指针 cliaddr 所指)以及该地址的大小(由指针 addrlen 所指)。如果我们对返回的客户协议地址不感兴趣，可将指针 cliaddr 和 addrlen 均置为空指针。

图 1.9 给出了这些指针的示例，已连接套接口每次都在循环中关闭，但监听套接口在整个服务器有效期内都保持开放。我们也明白了为什么函数 accept 的第二和第三个参数都是空指针，因为我们对客户的身份不感兴趣。

例：值-结果参数

现在，我们通过修改图 1.9 中所示代码以输出 IP 地址和客户的端口号来看看如何处理函数 accept 的值-结果参数，见图 4.11。

新的声明

第 7～8 行　我们定义两个新变量：len，它是一个值-结果参数；cliaddr，它是客户的协议地址。

接受连接并显示客户地址

第19～23行 我们将len初始化为套接口地址结构的大小，将指向cliaddr结构的指针和指向len的指针分别作为函数accept的第二和第三个参数。调用函数inet_ntop(3.7节)将套接口地址结构中的32位IP地址转换为一个点分十进制数ASCII字符串，调用函数ntohs(3.4节)将16位的端口号从网络字节序转换为主机字节序。

> 调用函数sock_ntop来代替函数inet_ntop将使得我们的服务器更具协议无关性，但该服务器已依赖于IPv4了。在图11.9中，我们将给出该服务器程序协议无关的版本。

如果在同一主机上运行该新服务器程序，然后运行客户程序，连接服务器两次，在客户上我们将看到下面的输出：

```
solaris % daytimetcpcli 127.0.0.1
Wed Jan 17 15:42:35 1996
solaris % daytimetcpcli 206.62.226.33
Wed Jan 17 15:42:53 1996
```

首先，我们将服务器的地址指定为回馈地址(127.0.0.1)，然后指定本身的IP地址(206.62.226.33)，下面是相应的服务器输出：

```
solaris # daytimetcpsrv1
connection from 127.0.0.1 ,port 33188
connection from 206.62.226.33, port 33189
```

```
 1 #include     "unp.h"
 2 #include     <time.h>
 3 int
 4 main(int argc,char **argv)
 5 {
 6     int          listenfd,connfd;
 7     socklen_t len;
 8     struct sockaddr_in servaddr,cliaddr;
 9     char         buff[MAXLINE];
10   time_t         ticks;
11   listenfd = Socket(AF_INET,SOCK_STREAM,0);
12   bzero(&servaddr,sizeof(servaddr));
13   servaddr.sin_family = AF_INET;
14   servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
15   servaddr.sin_port = htons(13);      /* daytime server */
16   Bind(listenfd,(SA *) &servaddr,sizeof(servaddr));
17   Listen(listenfd,LISTENQ);
18   for(;;) {
19       len = sizeof(cliaddr);
20       connfd = Accept(listenfd,(SA *) &cliaddr,&len);
21       printf("connection from %s,port %d\n",
22           Inet_ntop(AF_INET,&cliaddr.sin_addr,buff,sizeof(buff)),
23           ntohs(cliaddr.sin_port));
```

```
24          ticks = time(NULL);
25          snprintf(buff,sizeof(buff),"%.24s\r\n",ctime(&ticks));
26          Write(connfd,buff,strlen(buff));
27          Close(connfd);
28      }
29  }
```

图 4.11　输出客户 IP 地址和端口号的时间/日期服务器[intro/daytimetcpsrv1.c]

```
solaris # daytimetopsrv1
connection from 127.0.0.1, port 33188
connection from 206.62.226.33 port 33189
```

注意客户 IP 地址变化的结果。由于我们的时间/日期客户(图 1.5)不调用函数 bind,在 4.4 节中我们说此时内核选择的源 IP 地址是基于所用的接口的。第一种情况下,内核将源 IP 地址置为回馈地址,第二种情况下,内核又将地址置为以太网接口的 IP 地址。在该例子中我们还注意到,Solaris 内核所选择的临时端口号首先是 33188,然后是 33189(回忆一下图 2.6)。

最后一点,服务器脚本的 shell 提示符变为(#),此为超级用户的常用提示符。该服务器必须运行在超级用户特权下,绑定保留的 13 号端口。如果没有超级用户权限,对函数 bind 的调用将失败:

```
solaris % daytimetcpsrv1
bind error:Permission denied
```

4.7　fork 和 exec 函数

在阐述如何书写一个并发服务器程序之前(下一节),我们必须首先介绍一下 Unix 的函数 fork,该函数是 Unix 中派生新进程的唯一方法。

```
#include <unistd.h>
pid_t fork(void);
                                返回:在子进程中为 0,在父进程中为子进程 ID,-1——出错
```

如果你以前从未接触过此函数,则弄明白 fork 最困难的部分便是它调用一次却返回两次。在调用进程(称为父进程),它返回一次,返回值是新派生进程(称为子进程)的进程 ID 号,在子进程它还返回一次,返回值为 0。因此,可通过返回值来判断当前进程是子进程还是父进程。

fork 在子进程返回 0 而不是父进程 ID,原因是:子进程只有一个父进程,它总可以调用 getppid 来得到;而父进程有许多子进程,它没有办法来得到各子进程的 ID。如果父进程想跟踪所有子进程的 ID,它必须记住 fork 的返回值。

父进程中调用 fork 之前打开的所有描述字在函数 fork 返回之后都是共享的。我们将看到网络服务器利用此特性:父进程调用 accept,然后调用 fork,这样,已连接套接口就在父进程与子进程间共享,一般来说,接下来便是子进程读、写已连接套接口,而父进程则关闭已连接套接口。

fork 有两个典型应用：

1. 一个进程可为自己创建一个拷贝，这样，当一个拷贝处理一个操作时，其他的拷贝可以执行其他的任务。这是非常典型的网络服务器，本书后面我们将看到许多这样的例子。
2. 一个进程想执行其他的程序，由于创建新进程的唯一方法是调用 fork，进程首先调用 fork 来生成一个拷贝，然后其中一个拷贝（通常为子进程）调用 exec（后面将介绍）来代替自己去执行新程序。这对于像 shell 这样的程序是典型的用法。

以文件形式存储在磁盘上的可执行程序被 Unix 执行的唯一方法是：由一个现有进程调用六个 exec 函数中的一个。（当这六个函数中是哪一个被调用并不重要时，我们常统称它们为 exec 函数。）exec 用新程序代替当前进程映像，且此新程序一般都从 main 函数开始执行，进程 ID 并不改变。我们一般将调用 exec 的进程称为调用进程（calling process），而将新执行的程序称为新程序（new program）。

> 较老一些的手册和书上不确切地将新程序称为新进程（new process），这是不对的，因为并没有派生一个新的进程。

六个 exec 函数间的区别是：（a）被执行的程序是由文件名（filename）还是路径名（pathname）指定；（b）新程序的参数是一一列出还是由一个指针数组来索引；（c）调用进程的环境传递给新程序还是指定新环境。

```
#include <unistd.h>
int execl(const char *pathname, const char *arg0, ... /* (char *) 0 */);
int execv(const char *pathname, char *const argv[]);
int execle(const char *pathname, const char *arg0, ...
        /* (char *) 0, char *const envp[] */);
int execve(const char *pathname, char *const argv[], char *const envp[]);
int execlp(const char *filename, const char *arg0, ... /* (char *) 0 */);
int execvp(const char *filename, char *const argv[]);
                                所有6个函数返回：-1——出错，无返回——成功
```

这些函数只在出错时才返回调用者，否则，控制权传递到新程序的开始，通常是传递到函数 main。

这六个函数间的关系示于图 4.12。一般来说，只有 execve 是内核中的系统调用，其他五个函数都是调用 execve 的库函数。

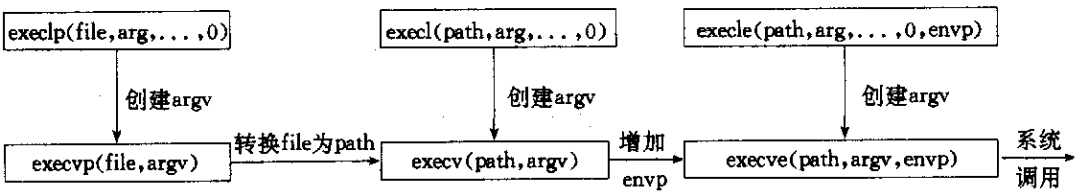


图 4.12 六个 exec 函数的关系

注意这六个函数的下列区别：

1. 上面的三个函数将每个参数串指定为 exec 的独立参数，用一个空指针来表示可变数量参数的终止；下面的三个函数有一个数组 argv，它包含了指向参数串的所有指针。此 argv 数组必须含有一个空指针来表示结束，因为数组的大小没有指定。
2. 左边一列的两个函数指定一个 filename 参数，它根据现行的 PATH 环境变量转换为 pathname。若函数 execlp 或 axecvp 的 filename 参数中含有斜杠(/)(不论在串中的什么位置)，将不再使用 PATH 变量。后两列的四个函数指定一个完全的 pathname 参数。
3. 左边两列的四个函数不显式指定一个环境指针，而是用外部变量 environ 的当前值来创建一个环境表传递给新程序；右边一列的两个函数显式指定一个环境表。envp 指针数组必须以一个空指针结束。

一般来说，调用 exec 之前在进程中打开的描述字在跨 exec 过程中保持打开状态。我们使用限定词“一般来说”是因为描述字也可用函数 fcntl 设置 FD_CLOEXEC 描述字标志来关闭。服务器 inetd 就利用了此特性，这一点我们在 12.5 节中还会讨论。

4.8　并发服务器

图 4.11 中的服务器是一个迭代服务器(iterative server)。对于像时间/日期这样简单的服务器，这是很好的；但是，当客户请求需花很长时间来得到服务时，我们不可能让一个服务器长时间地为某一个客户服务，而是同时为多个客户服务。Unix 下写一个并发服务器程序最简单的办法就是为每个客户均 fork 一个子进程。图 4.13 给出了一个典型的并发服务器程序框架。

当连接建立时，accept 返回，服务器调用 fork，然后子进程为客户提供服务(通过 connfd 即已连接套接口)，父进程等待另一个连接(通过 listenfd 即监听套接口)。子进程开始处理新客户后，父进程便关闭已连接套接口。

图 4.13 中，我们假设函数 doit 包含了客户所要求的所有服务功能。当此函数返回时，我们显式地关闭子进程中的已连接套接口。这一点并不一定要做，因为下一步是调用 exit，而进程结束的部分处理就是关闭所有由内核打开的描述字。是否需显式地调用 close，由个人的编程风格而定。

在 2.5 节中我们知道，对 TCP 套接口调用 close 会引发一个 FIN，后跟 TCP 连接终止序列。为什么图 4.13 中父进程 connfd 的 close 不终止它与客户的连接呢？为了搞明白此执行过程，我们必须了解每个文件或套接口都有一个访问计数，该访问计数在文件表项中维护(APUE 第 58～59 页)，它表示当前指向该文件或套接口的打开的描述字个数。在图 4.13 中，从 socket 返回后，与 listenfd 关联的文件表项访问计数值为 1，从 accept 返回后，与 connfd 关联的文件表项访问计数值也为 1。但是，当 fork 返回后，两个描述字在父进程与子进程间共享(即复制)，所以，与两个套接口相关联的文件表项访问计数值均为 2。当父进程关闭 connfd 时，只是将访问计数值从 2 减为 1。描述字只在访问计数值达到 0 时才真正关闭，这在后面某个时候子进程关闭 connfd 时会碰上。

```
pid_t pid;
int    listenfd,connfd;

listenfd = Socket( ... );
    /* fill in sockaddr_in() with server's well-known port */
Bind(listenfd,...);
Listen(listenfd,LISTENQ);
for( ; ;) {
    connfd = Accept(listenfd,...);      /* probably blocks */
    if( (pid = Fork()) == 0) {
        Close(listenfd);        /* child closes listening socket */
        doit(connfd);           /* process the request */
        Close(connfd);          /* done with this client */
        exit(0);                /* child terminates */
    }
    Close(connfd);              /* parent closes connected socket */
}
```

图 4.13 典型的并发服务器程序框架

我们还可将图 4.13 中出现的套接口和连接进行如下的可视化。首先，图 4.14 给出了在服务器阻塞于 accept 调用、连接请求从客户到达时客户和服务器的状态。

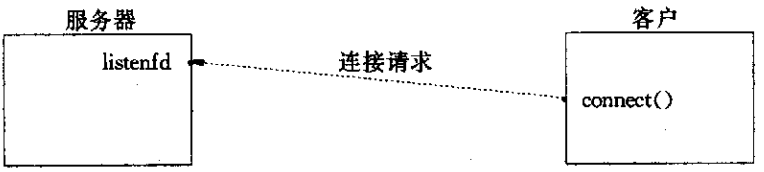


图 4.14 accept 返回前客户-服务器的状态

从 accept 返回后，我们立即就有图 4.15 所示状态。连接被内核接受，新的套接口即 connfd 被创建，这是一个已连接套接口，可由此通过连接读、写数据。

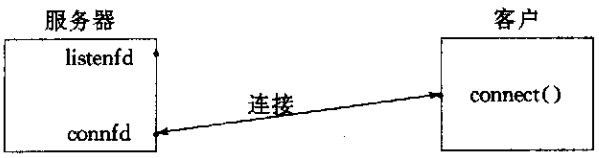


图 4.15 accept 返回后客户-服务器的状态

并发服务器的下一步是调用 fork，图 4.16 给出了从 fork 返回后的状态。

注意，此时描述字 listenfd 和 connfd 都是在父进程和子进程间共享的。

下一步是由父进程关闭已连接套接口，由子进程关闭监听套接口，如图 4.17 所示。

这是所期望的套接口最终状态：子进程处理与客户的连接，父进程可对监听套接口再次调用 accept 来处理下一个客户连接。

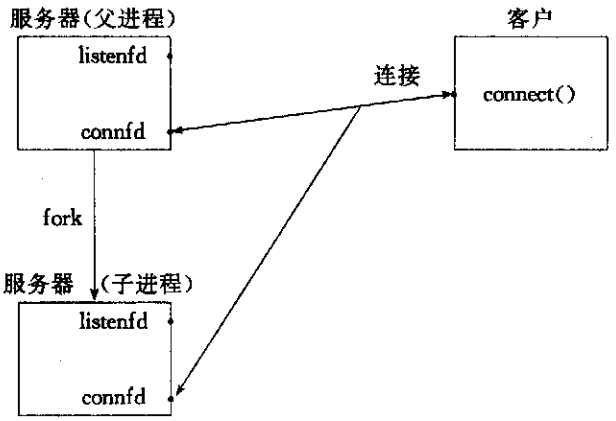


图 4.16　fork 返回后客户-服务器的状态

4.9　close 函数

一般的 Unix 函数 close 也用来关闭套接口，终止 TCP 连接。

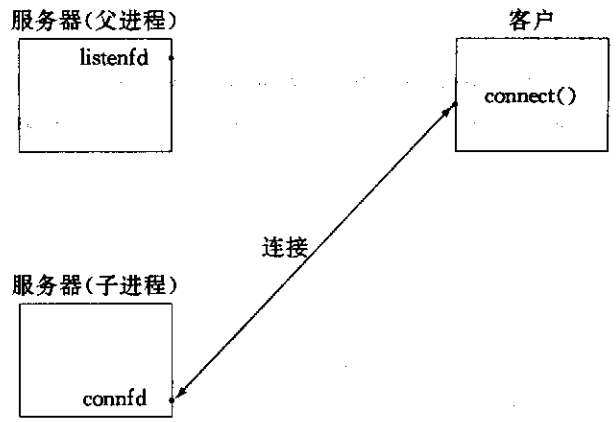


图 4.17　父子进程关闭相应套接口后客户-服务器的状态

```
#include <unistd.h>
int close(int sockfd);
```

返回：0——OK，−1——出错

TCP 套接口的 close 其缺省功能是将套接口做上“已关闭”标记，并立即返回到进程。这个套接口描述字不能再为进程所用：它不能用作为函数 read 或 write 的参数，但 TCP 将试着发送已排队待发的任何数据，然后按正常的 TCP 连接终止序列进行操作(2.5 节)。

在 7.5 节中，我们将介绍 SO_LINGER 套接口选项，它可用来改变 TCP 套接口的缺省功能。在那节，我们还介绍 TCP 应用进程必须做的事情，以保证对方的应用进程接收到已发出的任何数据。

描述字访问计数

在4.8节结尾我们提到，当并发服务器中父进程关闭已连接套接口时，它只是将描述字的访问计数值减1。由于访问计数值仍大于0，这次close调用并不引发TCP的四分组连接终止序列。对于父进程与子进程间共享已连接套接口的并发服务器来说，这正是所期望的。

如果我们确实想对TCP连接发一个FIN，可以改用函数shutdown(6.6节)而不是close，在6.5节我们再对此作介绍。

我们也必须意识到，如果父进程不对每个由accept返回的已连接套接口调用close，并发服务器将会发生什么。首先，父进程最终将耗尽可用描述字，因为任何进程在某时刻打开的描述字数总是有限的。但更重要的是，没有一个客户连接被终止。当子进程关闭已连接套接口时，它的访问计数值由2减为1且保持为1，因为父进程从未关闭已连接套接口，这将妨碍TCP连接终止序列的执行，从而连接永远保持开放。

4.10 getsockname和getpeername函数

这两个函数或返回与套接口关联的本地协议地址(getsockname)，或返回与套接口关联的远程协议地址(getpeername)。

```
#include <sys/socket.h>
int getsockname(int sockfd, struct sockaddr *localaddr, socklen_t *addrlen);
int getpeername(int sockfd, struct sockaddr *peeraddr, socklen_t *addrlen);
                                        两者均返回：0——OK，-1——出错
```

注意，两个函数的最后一个参数是值-结果参数，也就是说，两个函数都装填由指针localaddr或peeraddr所指的套接口地址结构。

> 讨论bind时我们提到，术语"名字"是一个误导。这两个函数返回与某网络连接任一端关联的协议地址，对于IPv4和IPv6，它是IP地址和端口号的组合。这些函数与域名没有任何联系(第9章)。

这两个函数因以下理由而存在：

- 在一个不调用bind的TCP客户上，当connect成功返回后，getsockname返回内核分配给此连接的本地IP地址和本地端口号。
- 在以端口号0调用bind后(通知内核选择本地端口号)，getsockname返回由内核分配的本地端口号。
- getsockname可用来获得某套接口的地址族，如图4.19所示。
- 在捆绑了一个通配IP地址的TCP服务器上(图1.9)，一旦与客户建立了连接(accept成功返回)，就可以调用getsockname来获得分配给此连接的本地IP地址。在这样的调用中套接口描述字参数必须是已连接套接口的描述字，而不是监听套接口的描述字。

  当一个服务器由调用accept的进程调用exec启动执行时，它获得客户身份的唯一

途径便是调用 getpeername。守护进程 inetd(12.5 节)fork 和 exec 一个 TCP 服务器时就是这么做的，如图 4.18 所示。inetd 调用 accept(左上方方框)，返回两个值：已连接套接口描述字 connfd，这是函数的返回值；客户的 IP 地址及端口号，见图中标有"对方地址"的小方框(它是一个网际套接口地址结构)。inetd 随后调用 fork，派生 inetd 的一个子进程。由于子进程从父进程的存储映像拷贝开始执行，因此对方地址的套接口地址结构对子进程也是可访问的。同样，子进程也能访问已连接套接口描述字，因为描述字在父子进程间是共享的。但是当子进程执行真正的服务器程序(譬如说 Telnet 服务器程序)时，子进程的存储映像被 Telnet 服务器的新程序文件所代替(也就是说，含有对方地址的套接口地址结构丢失了)，不过已连接套接口描述字在跨越 exec 后仍保持开放。Telnet 服务器调用的第一个函数便是 getpeername，由此获得客户的 IP 地址和端口号。

显然，在这最后一个例子中，Telnet 服务器在启动时必须知道 connfd 的值。这里有两个常用方法：第一种方法是，调用 exec 的进程可以将描述字号格式化成一个字符串，并将它作为一个命令行参数传递给新程序；第二种方法是，确立调用 exec 前把某个描述字总是设定成已连接套接口的约定。inetd 采用的就是第二种方法，它总是将描述字 0、1 和 2 设定成已连接套接口。

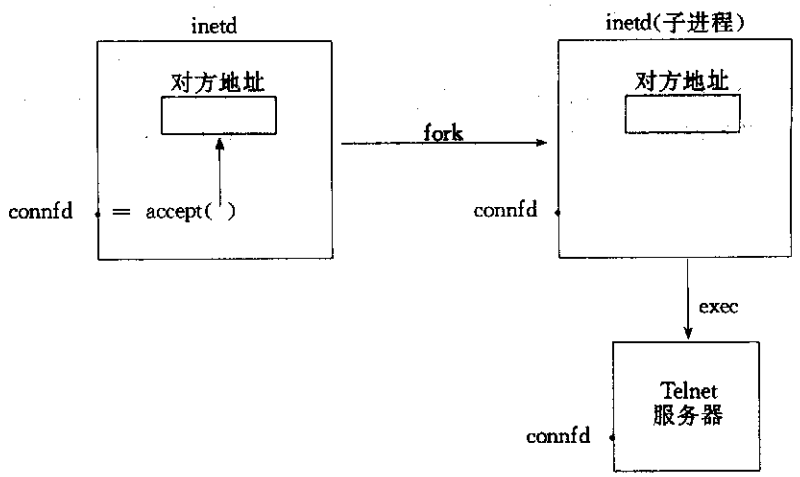


图 4.18　inetd 派生服务器的例子

例子：获取套接口的地址族

图 4.19 中所示的函数 sockfd_to_family 返回某个套接口的地址族。

```
1 #include    "unp.h"
2 int
3 sockfd_to_family(int sockfd)
4 {
5     union {
6         struct sockaddr sa;
7         char    data[MAXSOCKADDR];
8     } un;
9     socklen_t len;
```

```
10      len = MAXSOCKADDR;
11      if(getsockname(sockfd,(SA *) un.data,&len) < 0)
12        return(-1);
13      return (un.sa.sa_family);
14  }
```

图 4.19 返回套接口的地址族[lib/sockfd_to_family.c]

为最大的套接口地址结构分配空间

第5～8行 由于不知道要分配的套接口地址结构的类型，我们在头文件 unp.h 中使用常值 MAXSOCKADDR，它是最大套接口地址结构的字节数。我们在跟一个通用套接口地址结构的联合中定义了该大小的一个字符数组。

调用 getsockname

第10～13行 我们调用 getsockname 返回地址族。由于 Posix.1g 允许对未绑定的套接口进行 getsockname 调用，该函数应对任何打开的套接口描述字都有效。

4.11 小 结

所有的客户和服务器都从调用 socket 开始，返回一个套接口描述字。然后，客户调用 connect，服务器调用 bind、listen 和 accept。套接口一般由标准的 close 函数关闭，当然也可用函数 shutdown 来关闭(6.6节)。我们还要检查套接口选项 SO_LINGER 的效果(7.5节)。

多数 TCP 服务器是与调用 fork 来处理每个客户连接的服务器并发执行的。我们还将看到，多数 UDP 服务器则是迭代的。这两个模型已成功地运用了许多年，在第27章，我们将讨论使用线程与进程的其他服务器程序设计方法。

4.12 习 题

4.1 在4.4节中，我们说头文件<netinet/in.h>中定义的常值 INADDR_是主机字节序的，如何辨别？

4.2 修改图1.5，在 connect 返回成功后，调用 getsockname。使用 sock_ntop 输出分配给 TCP 套接口的本地 IP 地址和本地端口号。你的系统的临时端口在什么范围内(图2.6)？

4.3 在一个并发服务器中，假设在调用 fork 后子进程先运行，而且在 fork 调用返回父进程前，子进程就完成了对客户的服务，图4.13中对 close 的两次调用将会发生什么情况？

4.4 在图4.11中，把服务器的端口号从13改到9999(这样不需有超级用户特权就能启动程序)，并删掉 listen 调用，将会发生什么情况？

4.5 继续上一题，删掉 bind 调用，但保持 listen 调用，又将发生什么情况？

第 5 章　TCP 客户-服务器程序例子

5.1　概　述

现在，我们用前面章节所介绍的基本函数来编写一个完整的 TCP 客户-服务器程序例子，这个简单的例子是完成下述功能的一个回射服务器：

1. 客户从标准输入读一行文本，写到服务器上；
2. 服务器从网络输入读此行，并回射给客户；
3. 客户读此回射行并写到标准输出。

图 5.1 以所用的输入、输出函数描述了这个简单的客户-服务器模型。

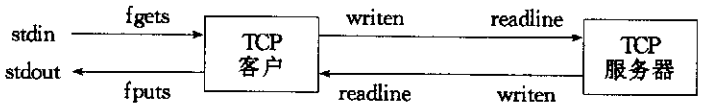


图 5.1　简单的回射客户-服务器

我们在客户与服务器之间画了两个箭头，但这是一个全双工的 TCP 连接。fgets 和 fputs 来自标准 I/O 函数库，writen 和 readline 函数可参见 3.9 节。

尽管我们开发自己的回射服务器实现，多数 TCP/IP 实现已经提供了这样的服务器，TCP 和 UDP 都使用(2.10 节)，我们自己的客户也用此服务器。

回射输入行这样一个客户-服务器程序是一个尽管简单然而有效的网络应用程序例子，客户-服务器实现所需要的所有基本步骤在此例中均出现了。若想将此例子扩充成你自己的应用程序，你只需依据对客户输入内容的处理方式，对服务器的工作作相应改变。

除了在正常方式下运行我们的客户及服务器外(输入一行，观察回射)，我们还检查此例的许多边界条件：客户和服务器都启动时是什么情况；客户正常终止时是什么情况；如果服务器进程在客户之前终止客户会是什么情况；如果服务器主机崩溃则客户又会是什么情况，等等。通过观察这些情况，弄清网络层次发生的一些事情以及它们是如何反映到套接口 API 的，我们将对这些层次的工作原理有更多的认识，并知道如何编写应用代码来处理这些情形。

在所有这些例子中，我们使用了诸如地址和端口这样的特定于协议的“硬编码”常值，这有两个原因：第一，我们必须确切地知道存储在特定于协议的地址结构中的内容；第二，我们还没有讨论到可使这个方便地移植的库函数，这些库函数将在第 9 章和第 11 章中讨论。

我们现在就注意，随着我们学习越来越多的网络编程知识(图 1.12 和图 1.13)，在后续章节中我们将对这儿的客户和服务器程序做多次修改。

5.2 TCP回射服务器程序:main函数

我们的客户和服务器程序依循图4.1中的函数调用流程,其中并发服务器程序见图5.2。

创建套接口,捆绑服务器的众所周知端口

第9～15行 创建一个TCP套接口,用通配地址(INADDR_ANY)和服务器的众所周知端口(在头文件unp.h中定义为9877的SERV_PORT)填写网际套接口地址结构。捆绑一个通配地址意味着通知系统,我们将接受目的地址为任何本地接口的连接(假设系统是多宿的)。我们对TCP端口号的选择基于图2.6,它应比1023大(我们不需要保留端口),大于5000(以免与许多源自Berkeley的实现分配的临时端口冲突),小于49152(以免与临时端口号的"正确"范围冲突),并应不与任何已注册的端口冲突。这个套接口由函数listen转换为监听套接口。

等待完成客户连接

第17～18行 服务器阻塞于accept调用,等待客户连接的完成。

并发服务器

第19～24行 对于每个客户,函数fork派生出一个子进程,由此子进程处理新客户。正如我们在4.8节中所讨论的那样,子进程关闭监听套接口,而父进程关闭已连接套接口,然后,子进程调用函数str_echo(图5.3)来处理客户。

```
1 #include    "unp.h"
2 int
3 main(int argc,char * * argv)
4 {
5     int         listenfd,connfd;
6     pid_t       childpid;
7     socklen_t clilen;
8     struct sockaddr_in cliaddr,servaddr;
9     listenfd = Socket(AF_INET,SOCK_STREAM,0);
10    bzero(&servaddr,sizeof(servaddr));
11    servaddr.sin_family = AF_INET;
12    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
13    servaddr.sin_port = htons(SERV_PORT);
14    Bind(listenfd,(SA * ) &servaddr,sizeof(servaddr));
15    Listen(listenfd,LISTENQ);
16    for(; ;) {
17        clilen = sizeof(cliaddr);
18        connfd = Accept(listenfd,(SA * ) &cliaddr,&clilen);
19        if((childpid= Fork()) == 0) {     /* child process */
20            Close(listenfd);        /* close listening socket */
21            str_echo(connfd);       /* process the request */
22            exit(0);
```

```
23          }
24          Close(connfd);          /* parent closes connected socket */
25      }
26  }
```

图5.2 TCP回射服务器程序(图5.12中改进)[tcpcliserv/tcpserv01.c]

```
1  #include    "unp.h"
2  void
3  str_echo(int sockfd)
4  {
5      ssize_t n;
6      char    line[MAXLINE];
7      for ( ; ; ) {
8          if((n = Readline(sockfd,line,MAXLINE)) == 0)
9              return;             /* connection closed by other end */
10         Writen(sockfd,line,n);
11     }
12 }
```

图5.3 函数str_echo:在套接口上回射行[lib/str_echo.c]

5.3 TCP回射服务器程序:str_echo函数

图5.3中的函数str_echo执行处理每个客户的服务:从客户读入各行并将它们回射给客户。

读一行并回射此行

第7～11行 readline从套接口读下一行,该行随后由writen回射给客户。如果客户关闭连接(正常情况),那么接收到的客户FIN导致子进程的readline返回0,从而使得函数str_echo也返回,终止图5.2中的子进程。

5.4 TCP回射客户程序:main函数

图5.4所示为TCP客户的main函数。

```
1  #include    "unp.h"
2  int
3  main(int argc,char **argv)
4  {
5      int     sockfd;
6      struct sockaddr_in servaddr;
7      if(argc != 2)
8          err_quit("usage:tcpcli <IPaddress>");
9      sockfd = Socket(AF_INET,SOCK_STREAM,0);
10     bzero(&servaddr,sizeof(servaddr));
11     servaddr.sin_family = AF_INET;
12     servaddr.sin_port = htons(SERV_PORT);
```

```
13    Inet_pton(AF_INET,argv[1],&servaddr.sin_addr);
14    Connect(sockfd,(SA *) &servaddr,sizeof(servaddr));
15    str_cli(stdin,sockfd);      /* do it all */
16    exit(0);
17 }
```

图 5.4 TCP 回射客户程序[tcpcliserv/tcpcli01.c]

创建套接口,装填网际套接口地址结构

第 9～13 行 创建一个 TCP 套接口,用服务器的 IP 地址和端口号装填一个网际套接口地址结构。我们可从命令行参数取得服务器的 IP 地址,从头文件 unp.h 取得服务器的众所周知端口号(SERV_PORT)。

与服务器连接

第 14～15 行 connect 建立与服务器的连接,函数 str_cli(图 5.5)完成客户处理的剩余部分工作。

5.5 TCP 回射客户程序:str_cli 函数

图 5.5 中所示该函数完成客户处理循环:从标准输入读一行文本,写到服务器上,读回服务器对此行的回射,并把回射写到标准输出上。

```
1 #include    "unp.h"
2 void
3 str_cli(FILE *fp,int sockfd)
4 {
5     char    sendline[MAXLINE],recvline[MAXLINE];
6     while (Fgets(sendline,MAXLINE,fp) != NULL) {
7         Writen(sockfd,sendline,strlen(sendline));
8         if(Readline(sockfd,recvline,MAXLINE) == 0)
9             err_quit("str_cli:server terminated prematurely");
10        Fputs(recvline,stdout);
11    }
12 }
```

图 5.5 函数 str_cli:客户处理循环[lib/str_cli.c]

读一行,写到服务器

第 6～7 行 fgets 读一行文本,writen 将此行发送到服务器。

从服务器读回射行,写到标准输出

第 8～10 行 readline 从服务器读回射行,fputs 将其写到标准输出。

返回 main 函数

第 11～12 行 当 fgets 遇到文件结束符或错误时,返回一个空指针,此时客户处理循环便终止。我们的包裹函数 Fgets 检查错误,如果发现错误就放弃执行,因此 Fgets 函数在遇到

文件结束符时才返回一个空指针。

5.6　正常启动

虽然我们的 TCP 例子很小(两个 main 函数加上 str_echo、str_cli、readline 和 writen 总共才 150 行代码),但对于我们弄清客户和服务器是如何启动,如何终止的,更为重要的是当某些错误发生时,如:客户主机崩溃,客户进程崩溃,网络连接断开等等,将会发生什么事情,这个例子则至关重要。只有搞清这些边界条件以及它们与 TCP/IP 协议的相互作用,我们才能写出可以处理这些情况的健壮的客户和服务器程序。

首先,我们在主机 bsdi 上后台启动服务器。

```
bsdi % tcpserv01 &
[1]    21130
```

当服务器启动后,调用 socket、bind、listen 和 accept,并阻塞于 accept 调用 (我们还没有启动客户)。在启动客户之前,我们运行程序 netstat 来检查服务器监听套接口的状态。

```
bsdi % netstat -a
Proto   Recv-Q   Send-Q   Local Address      Foreign Address      (state)
tcp          0        0   *.9877             *.*                  LISTEN
```

这里,我们只给出了输出的第一行(标题)以及我们最关心的那一行。此命令列出系统中所有套接口的状态,可能有大量输出。我们必须用标志-a 来查看监听套接口。

这个输出正是我们所期望的:有一个套接口处于 LISTEN 状态,它有通配的本地 IP 地址,本地端口为 9877。netstat 用星号"*"来表示一个为 0 的 IP 地址(INADDR_ANY,通配地址)或为 0 的端口号。

然后,我们在相同的主机上启动客户,指明服务器的 IP 地址为 127.0.0.1,也可指明该地址为 206.62.226.35(图 1.16)。

```
bsdi % tcpcli01 127.0.0.1
```

客户调用 socket 和 connect,后者引起 TCP 的三路握手过程。当三路握手完成后,connect 返回客户,accept 返回服务器,连接建立,接着执行以下步骤:

1. 客户调用 str_cli 函数,该函数将阻塞于 fgets 调用,因为我们还从未输入过一行文本。
2. 当 accept 返回服务器后,服务器调用 fork,由子进程调用 str_echo。此函数调用 readline,它又因调用 read 而阻塞,等待客户送出一行。
3. 另一方面,服务器父进程再次调用 accept 并阻塞,等待下一个客户连接。

至此,我们有三个进程,它们都在睡眠(阻塞):客户、服务器父进程和服务器子进程。

当三路握手完成时,我们特意首先列出了客户的步骤,然后是服务器的步骤。从图 2.5 中可知其原因:客户接收到三路握手的第二个分节时,connect 返回,而服务器要直到接收到三路握手的第三个分节才返回,即在 connect 返回之后过了½往返时间才返回。

我们特意在同一主机上运行客户和服务器，因为这是试验客户-服务器应用程序的最简单方法。由于我们在同一主机上运行客户和服务器，netstat 相应于 TCP 连接又多显示了两行输出。

```
bsdi % netstat -a
Proto Recv-Q Send-Q  Local Address      Foreign Address    (state)
tcp        0      0  localhost. 9877    localhost. 1052    ESTABLISHED
tcp        0      0  localhost. 1052    localhost. 9877    ESTABLISHED
tcp        0      0  *. 9877            *. *               LISTEN
```

第一个 ESTABLISHED 行，其端口号是 9877，对应于服务器子进程的套接口；第二个 ESTABLISHED 行，其端口号是 1052，对应于客户的套接口。如果我们在不同的主机上运行客户和服务器，那么客户主机就只输出客户的套接口，服务器主机也只输出两个服务器套接口。

我们也可用命令 ps 来检查这些进程的状态和关系。

```
bsdi % ps -l
PID   PPID  WCHAN  STAT  TT       TIME COMMAND
19130 19129 wait   Is    p1    0:04.99 -ksh (ksh)
21130 19130 netcon I     p1    0:00.06 tcpserv01
21131 19130 ttyin  I+    p1    0:00.09 tcpcli01 127.0.0.1
21132 21130 netio  I     p1    0:00.01 tcpserv01
21134 21133 wait   Ss    p2    0:03.50 -ksh (ksh)
21149 21134 -      R+    p2    0:00.05 ps -l
```

（我们已删去了不影响讨论的许多输出列。）在此输出中，我们在相同的窗口下运行客户和服务器（p1，表示伪终端号 1），在另一个窗口运行命令 ps（p2）。PID 和 PPID 列给出了父进程和子进程的关系，由于子进程的 PPID 是父进程的 PID，我们可以看出，第一个 tcpserv 行是父进程，第二个 tcpserv 行是子进程。父进程的 PPID 是 shell（ksh）。

我们所有三个网络进程的 STAT 列都是“I”，意味着进程为空闲（即睡眠），两个 STAT 列后面的加号“+”表示该进程为控制终端的前台进程。如果进程处于睡眠态，WCHAN 列指出相应状态。4.4BSD 在进程阻塞于 accept 或 connect 时，输出 netcon；在进程阻塞于套接口输入或输出时，输出 netio；在进程阻塞于终端 I/O 时，输出 ttyin。我们三个网络进程的 WCHAN 值说明是正确的。

5.7 正常终止

此时，连接已建立，无论我们在客户标准输入上键入什么，都将得到回射。

```
bsdi % tcpcli01 127.0.0.1      我们已经给出过本行
hello,world                    现在键入这一行
hello,world                    这一行被回射回来
good bye
good bye
^D                             <Ctrl-D>是我们的终端 EOF 字符
```

我们键入两行，每行都得到回射，我们键入终止的 EOF 字符（Control-D）以终止客户，

此时若立即执行 netstat 命令，我们将看到以下结果。

```
bsdi % netstat -a | grep 9877
tcp        0      0  localhost.1052         localhost.9877         TIME_WAIT
tcp        0      0  *.9877                 *.*                    LISTEN
```

连接的客户端(本地端口号为 1052)进入了 TIME_WAIT 状态(图 2.6)，而监听服务器仍在等待另一个连接。(这一次，我们使命令 netstat 的输出进入管道作为 grep 的输入，只输出与服务器的众所周知端口有关行，但这样做也删掉了标题行。)

我们可以总结出正常终止客户和服务器的步骤：

1. 当我们键入 EOF 字符时，fgets 返回一个空指针，于是 str_cli(图 5.5)返回。
2. 当 str_cli 返回到客户的函数 main(图 5.4)时，main 通过调用 exit 终止。
3. 进程终止处理的一部分是关闭所有打开的描述字，所以客户套接口由内核关闭。这导致客户 TCP 发送一个 FIN 给服务器，服务器 TCP 则以 ACK 响应，这就是 TCP 连接终止序列的前半部分。此时，服务器套接口处于 CLOSE_WAIT 状态，客户套接口处于 FIN_WAIT_2 状态(图2.4和图2.5)。
4. 当服务器 TCP 接收 FIN 时，服务器子进程阻塞于 readline(图 5.3)调用，于是 readline 返回 0，这导致函数 str_echo 返回服务器子进程的 main。
5. 服务器子进程通过调用 exit 来终止(图 5.2)。
6. 服务器子进程中打开的所有描述字随之关闭。由子进程来关闭已连接套接口引发 TCP 连接终止序列的最后两个分节：一个从服务器到客户的 FIN 和一个从客户到服务器的 ACK(图2.5)。至此，连接完全终止，客户套接口进入 TIME_WAIT 状态。
7. 进程终止处理的另一部分是在服务器子进程终止时给父进程发一个信号 SIGCHLD。这在本例中发生了，但我们在代码中未捕获这个信号，而这个信号的缺省动作是忽略的。我们可以用命令 ps 来检查。

```
bsdi % ps
  PID  TT  STAT      TIME  COMMAND
19130  p1  Ss     0:05.08  -ksh (ksh)
21130  p1  I      0:00.06  ./tcpserv01
21132  p1  Z      0:00.00  (tcpserv01)
21167  P1  R+     0:00.10  ps
```

现在，子进程的状态是 Z(僵尸)。

我们必须清除僵尸进程，这要涉及到 Unix 信号的处理。下一节我们将对信号处理作一简述，再下一节再继续我们的例子。

5.8　Posix 信号处理

信号是发生某事件时对进程的通知，有时称为软中断。它一般是异步的，这就是说，进程不可能提前知道信号发生的时间。

信号可以

- 由一进程发往另一进程(或本身)；

· 由内核发往某进程。

前一节结尾我们提到的信号 SIGCHLD 就是内核在某进程终止时发给此终止进程的父进程的信号。

每个信号都有一个处理办法(disposition),也称为与信号关联的行为(action)。我们通过调用函数 sigaction(不久将讨论到)来设置一个信号的处理办法,并有三个选择。

1. 我们可以提供一个函数,在信号发生时随即调用。这个函数称为信号处理程序(signal handler),而此行为则称为捕获(catching)信号。有两个信号不能被捕获,它们是 SIGKILL 和 SIGSTOP。我们的函数由信号值这个单一的整数参数来调用且无返回值,所以其函数原型为:

```
void handler (int signo);
```

对于多数信号来说,调用函数 sigaction 并指定信号发生时调用的函数是捕获信号所需做的所有事情,但我们稍后将看到,有些信号,如 SIGIO、SIGPOLL 和 SIGURG,还要求捕获它们的进程要有其他动作。

2. 我们可以通过设置信号的处理办法为 SIG_IGN 来忽略它,但 SIGKILL 和 SIGSTOP 这两个信号不能忽略。
3. 我们可以设置信号的处理办法为 SIG_DFL 来为它设置缺省处理办法。一般地说,缺省处理办法是在接收到信号时终止进程,特定信号还在当前工作目录产生一个进程的核心映像。个别信号的缺省处理办法是忽略;SIGCHLD 和 SIGURG(带外数据到达时发送,见第 21 章)就是本书中要用的缺省处理办法是忽略的两个信号。

signal 函数

建立信号的处理办法的 Posix 方法就是调用函数 sigaction。不过这有点复杂,因为该函数的一个参数是我们必须分配并填写的结构。简单些的方法就是调用函数 signal,其第一个参数是信号名,第二个参数或为指向函数的指针,或为常值 SIG_IGN 或 SIG_DFL。但是函数 signal 是一个历史较长的函数,它早于 Posix.1。调用时,不同的实现提供不同的信号语义以实现后向兼容性,而 Posix 则明确规定调用函数 sigaction 时的信号语义。解决的方法就是定义我们自己的函数,也称为 signal,它就调用 Posix 函数 sigaction,这样用 Posix 语义提供了一个简单的接口。我们在自己的函数库中包含了这个函数以及 err_XXX 函数和包裹函数等等,构造本书中我们的程序时,我们都指定了这个函数库。① 此函数如图 5.6 所示。

```
1 #include    "unp.h"
2 Sigfunc *
3 signal(int signo,Sigfunc *func)
4 {
5     struct sigaction act, oact;
6     act.sa_handler = func;
7     sigemptyset(&act.sa_mask);
8     act.sa_flags = 0;
```

① 译者注: 构造程序是指使用 make 工具把源程序和/或目标程序编译链接成可执行程序。本书免费可得的源代码(见前言)提供了构造其中各个程序的 makefile 文件。

```
 9    if(signo == SIGALRM) {
10 #ifdef  SA_INTERRUPT
11         act.sa_flags |= SA_INTERRUPT;     /* SunOS 4.x */
12 #endif
13         } else {
14 #ifdef    SA_RESTART
15           act.sa_flags |= SA_RESTART;     /* SVR4, 4.4BSD */
16 #endif
17    }
18    if (sigaction(signo,&act,&oact) < 0)
19        return (SIG_ERR);
20    return(oact.sa_handler);
21 }
```

图 5.6　调用 Posix sigaction 函数的 signal 函数[lib/signal.c]

用 typedef 简化函数原型

第 2～3 行　函数 signal 的正常函数原型因层次太多而变得很复杂：

```
void (*signal (int signo, void (*func)(int)))(int);
```

为了简化它，在头文件 unp.h 中我们定义类型 sigfunc 为

```
typedef void Sigfunc(int);
```

它说明信号处理程序是带有一个整型参数且无返回值的函数。这样，signal 的函数原型就成为：

```
Sigfunc *signal (int signo, Sigfunc *func);
```

此函数的第二个参数和返回值都是指向信号处理函数的指针。

设置处理程序

第 6 行　sigaction 结构的元素 sa_handler 被置为 func 参数。

设置处理程序的信号掩码

第 7 行　Posix 允许我们指定这样一组信号，它们在信号处理程序被调用时阻塞②。任何阻塞的信号都不能递交(delivering)给进程。我们设置成员 sa_mask 为空集，这意味着当信号处理程序运行时没有别的信号阻塞。Posix 保证信号处理程序正在运行时被捕获的信号是阻塞的。

设置标志 SA_RESTART

第 8～17 行　有一个可选标志 SA_RESTART，如果设置，由此信号中断的系统调用将由内核自动重启。(在下一节继续我们的例子时，将详细地讨论中断的系统调用。)如果被捕获的信号不是 SIGALRM，我们将设置标志 SA_RESTART。(将 SIGALRM 作为特殊情况的原因是：生成此信号的目的是为 I/O 操作设置超时，这将在 13.2 节中进行讨论，这时我们

② 译者注：这里的阻塞不同于我们此前一直使用的同名词。这里的阻塞是指阻塞某个信号或某个信号集，防止它们在阻塞期间递交(delivering)。它的反操作称为解阻塞。而此前一直使用的阻塞是指阻塞在某个系统调用上，也就是说这个系统调用因为目前没有必要的资源可用而必须等待，直到这些资源变为可用后才可能返回。等待期间进程进入睡眠状态。与这个阻塞相对的概念是非阻塞，也就是说非阻塞的系统调用即使没有必要的资源可用也立即返回，不过会告诉调用者发生了这种情况，这样调用者可以继续调用同一个系统调用。读者应注意区分这两个同名异义词。

希望阻塞的系统调用被信号中断。)较早期的系统,如 SunOS 4.x,缺省就是自动重启被中断的系统调用,而它的互补标志则定义成 SA_INTERRUPT。如果定义了这个标志,在捕获的信号是 SIGALRM 时我们就设置它。

调用函数 sigaction

第 18～20 行 我们调用函数 sigaction,并将信号的旧行为作为函数 signal 的返回值。本书通篇使用图 5.6 的函数 signal。

Posix 信号语义

我们对 Posix 兼容系统上的信号处理作以下总结:

- 一旦安装了信号处理程序,它便一直安装着(较早期的系统是每执行一次就将其拆除)。
- 当一个信号处理程序正在执行时,所递交的信号是阻塞的,而且,安装处理程序时定义在信号集 sa_mask 中传递给函数 sigaction 的任何额外信号也是阻塞的。在图 5.6 中,我们将 sa_mask 置为空集,这意味着除了被捕获的信号外,没有额外信号阻塞。
- 如果一个信号在阻塞时生成了一次或多次,在信号解阻塞后一般只递送一次。也就是说,缺省时 Unix 信号是不排队的,下一节我们将看到这样的一个例子。Posix 实时标准 1003.1b 定义了一组排队的可靠信号,但本书中我们不使用。
- 利用函数 sigprocmask 来有选择性地阻塞或不阻塞一组信号是可能的,这使得我们可以在某段临界区代码执行时,不许捕获某些信号,以此来保护这段代码。

5.9 处理 SIGCHLD 信号

设置僵尸(Zombie)状态的目的就是维护子进程的信息,以便父进程在稍后的某个时候取回。此信息包括子进程的 ID、终止状态以及子进程的资源利用信息(CPU 时间、内存等等)。如果一个进程终止,且该进程有子进程处于僵尸状态,则所有僵尸子进程的父进程 ID 均置为 1(init 进程),init 进程将作为这些子进程的继父并负责清除它们(也就是说,init 进程将 wait 它们,从而去除僵尸进程)。有些 Unix 系统给僵尸进程输出的 COMMAND 列为<defunct>(ps 命令输出)。

处理僵尸进程

显然,我们不愿看见僵尸进程,它们占用内核空间,最终导致我们无法正常工作。无论何时我们创建子进程都必须等待,以免变成僵尸进程。为此,我们建立了一个信号处理程序来捕获信号 SIGCHLD,在处理程序中我们调用 wait。(我们将在 5.10 节中介绍函数 wait 和 waitpid。)我们在图 5.2 所示的代码中调用 listen 之后,增加函数调用:

```
Signal (SIGCHLD, sig_chld);
```

来建立信号处理程序(这必须在创建第一个子进程之前完成,且只做一次)。然后,我们定义信号处理程序即函数 sig_chld,如图 5.7 所示。

```
1 #include    "unp.h"
2 void
3 sig_chld(int signo)
4 {
5     pid_t    pid;
6     int      stat;
7     pid = wait(&stat);
8     printf("child %d terminated\n",pid);
9     return;
10 }
```

图 5.7　调用 wait 的 SIGCHLD 信号处理程序[tcpcliserv/sigchldwait.c]

警告：在信号处理程序中调用诸如 printf 这样的标准 I/O 函数是不合适的，原因将在 11.14 节讨论。在这里我们调用了函数 printf 是为了把它作为一个诊断工具，看看子进程是什么时候终止的。

在系统 V 和 Unix 98 中，如果进程将 SIGCHLD 的处理方法置为 SIG_IGN，它的子进程就不会成为僵尸进程。遗憾的是，这仅仅适用于系统 V 和 Unix 98，Posix.1 则明确表示它不支持这一点。另一个处理僵尸进程的方法就是捕获 SIGCHLD，并调用 wait 或 waitpid。

如果我们在 Solaris 2.5 下编译这样一个程序：以图 5.2 中代码为基础，加上对 Signal 的调用以及我们的信息处理程序 sig_chld，且所用的函数 signal 是系统库中的(不是图 5.6 中的)，我们有如下结果：

```
solaris % tcpserv02 &                    在后台启动服务器
[2]     16939
solaris % tcpcli01 127.0.0.1             再在前台启动客户
hi there                                 我们键入这一行
hi there                                 这一行被回射回来
^D                                       我们键入 EOF 字符
child 16942 terminated                   这是信号处理程序中 printf 的输出
accept error:interrupted system call     然而 main 函数终止执行
```

具体的各个步骤如下：

1. 我们键入 EOF 字符来终止客户，客户 TCP 发一个 FIN 给服务器，服务器以 ACK 响应。
2. FIN 的接收导致服务器 TCP 递送一个 EOF 给子进程阻塞中的 readline，从而子进程终止。
3. 当信号 SIGCHLD 递交时，父进程阻塞于 accept 调用。函数 sig_chld(信号处理程序)执行，它调用函数 wait 取到子进程的 PID 和终止状态，再调用 printf，然后返回。
4. 由于该信号是在父进程阻塞于慢系统调用(accept)时由父进程捕获的，所以内核将使 accept 返回一个 EINTR 错误(被中断的系统调用)，而父进程不处理此错误(图 5.2)，所以中止。

本例的目的是为了说明，在编书写捕获信号的网络程序时，我们必须认清被中断的系统调用且处理它们。在 Solaris 2.5 环境下运行的这个特定例子中，标准 C 库中提供的函数 sig-

nal 不会让内核自动重启中断的系统调用。也就是说，我们在图 5.6 中设置的标志 SA_RESTART 并没由系统库中的函数 signal 设置的。也有的系统可以自动重启中断的系统调用。如果我们在 4.4BSD 环境下运行上述例子，用函数 signal 的库版本，内核将重启中断的系统调用，于是 accept 不会返回错误。为了处理不同操作系统间这个潜在的问题，我们有理由定义一个自己的函数 signal(图 5.6)，并在全书使用。

在我们的信号处理程序中(图 5.7)，总是有一个显式的 return 语句，即使到了函数结尾也不例外，而信号处理程序并没有返回值(返回值类型为 void)。这么做有助于产生这样的警告：信号处理程序的 return 语句可能中断了某个系统调用。

处理被中断的系统调用

我们已经用术语慢系统调用(slow system call)来描述函数 accept，对于那些可能永远阻塞的系统调用我们也可用此术语。永远阻塞的系统调用是指调用有可能永远无法返回，多数网络支持函数都属于这一类。譬如，如果没有客户连接到服务器上，则服务器对 accept 的调用就没有返回保证。类似地，在图 5.3 中，如果客户从未发送过一行要求服务器回射的文本，则服务器对 read(通过 readline)的调用将永不返回。其他慢系统调用的例子是对管道和终端设备的读和写。有一个例外值得注意，它就是磁盘 I/O，它一般都返回调用者(假设没有灾难性的硬件故障)。

应用于这儿的基本规则是：当一个进程阻塞于慢系统调用时捕获到一个信号，等到信号处理程序返回时，系统调用可能返回一个 EINTR 错误。有些内核自动重启某些被中断的系统调用。为了便于移植，当我们编写一个捕获信号的程序时(多数并发服务器捕获 SIGCHLD)，我们必须对慢系统调用返回 EINTR 有所准备。移植性问题是由早期使用的修饰词“可能”、“有些”和对 Posix 标志 SA_RESTART 的支持可选造成的。即使某实现支持标志 SA_RESTART，也并非所有的被中断系统调用都可自动重启，譬如说，大多数源自 Berkeley 的实现从不自动重启 select，有些实现从不重启 accept 和 recvfrom。

为了处理一个被中断的 accept，我们将图 5.2 中对 accept 的调用从 for 循环开始改起，如下所示：

```
for ( ; ; ) {
    clilen = sizeof(cliaddr);
    if ( (connfd = accept(listenfd, (SA *) &cliaddr,&clilen))< 0) {
        if (errno == EINTR)
            continue;          /* back to for() */
        else
            err_sys ("accept error");
    }
```

注意，我们调用的是函数 accept 而不是包裹函数 Accept，因为我们必须自己处理函数的失败情况。

在这段代码中，我们所做的事情就是自己重启被中断的系统调用，这对于 accept 以及诸如 read、write、select 和 open 这样的函数是合适的，但有一个函数我们自己不能重启：connect。如果这个函数返回 EINTR，我们就不能再调用它，否则将立即返回错误。当 connect 被一个捕获的信号中断而且不自动重启(TCPv2 第 466 页)时，我们必须调用 select 来等待连

接完成，如 15.3 节所述。

5.10　wait 和 waitpid 函数

在图 5.7 中，我们调用了函数 wait 来处理终止的子进程。

```
#include <sys/wait.h>
    pid_t wait (int *statloc);
    pid_t waitpid(pid_t pid, int *statloc, int options);
                                        二者均返回：进程 ID 为 0 成功，或为－1 出错
```

函数 wait 和 waitpid 均返回两个值：函数的返回值是终止子进程的进程 ID 号，子进程的终止状态（一个整数）则是通过指针 statloc 返回的。有三个宏我们可以调用，它们检查终止状态，并告诉我们子进程是否正常终止，是由信号杀死还是仅仅作业控制而停止。还有些宏让我们可以取得子进程的退出状态，杀死进程的信号值或停止进程的作业控制信号值。在图 14.10 中，我们用宏 WIFEXITED 和 WEXITSTATUS。

如果没有终止的子进程让进程来调用 wait，但有一个或多个正在执行的子进程，则 wait 阻塞直到第一个现有子进程终止。

函数 waitpid 对等待哪个进程及是否阻塞给了我们更多的控制。首先，参数 pid 让我们指定想等待的进程 ID，值－1 表示等待第一个终止的子进程。（也有其他选项，它们涉及到进程组 ID，本书中不需考虑。）参数 options 让我们指定附加选项。最常用的选项是 WNOHANG，它通知内核在没有已终止子进程时不要阻塞。

函数 wait 和 waitpid 的区别

现在，我们图示出函数 wait 和 waitpid 在用来清除终止子进程时的区别。为此，我们将 TCP 客户修改为如图 5.8 所示，客户与服务器建立 5 个连接，在调用函数 str_cli 时仅用第一个连接（sockfd[0]）。建立多个连接的目的就是要从并发服务器上创建多个子进程，如图 5.9 所示。

```
1 #include    "unp.h"
2 int
3 main(int argc,char * * argv)
4 {
5     int         i,sockfd[5];
6     struct      sockaddr_in servaddr;
7     if (argc ! = 2)
8         err_quit("usage:tcpcli <IPaddress>");
9     for(i = 0;i < 5; i++) {
10        sockfd[i] = Socket(AF_INET,SOCK_STREAM,0);
11        bzero(&servaddr,sizeof(servaddr));
12        servaddr.sin_family = AF_INET;
13        servaddr.sin_port = htons (SERV_PORT);
14        Inet_pton(AF_INET,argv[1],&servaddr.sin_addr);
```

```
15          Connect(sockfd[i],(SA *) &servaddr,sizeof(servaddr));
16      }
17      str_cli(stdin,sockfd[0]);      /* do it all */
18      exit(0);
19  }
```

图 5.8 与服务器建立了 5 个连接的 TCP 客户程序[tcpcliserv/tcpcli04.c]

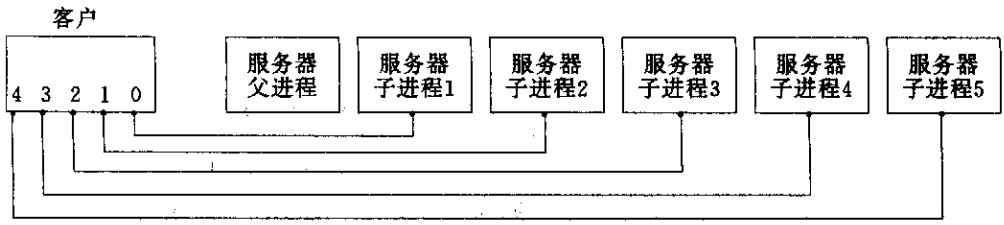


图 5.9 与同一个并发服务器建立了 5 个连接的客户

当客户终止时,所有打开的描述字由内核自动关闭(我们不调用 close,仅调用 exit),且所有 5 个连接基本在同一时刻终止。这就引发了 5 个 FIN,每个连接一个,它们反过来使服务器的 5 个子进程基本在同一时刻终止。这又导致差不多在同一时刻递交 5 个 SIGCHLD 信号给父进程,如图 5.10 所示。

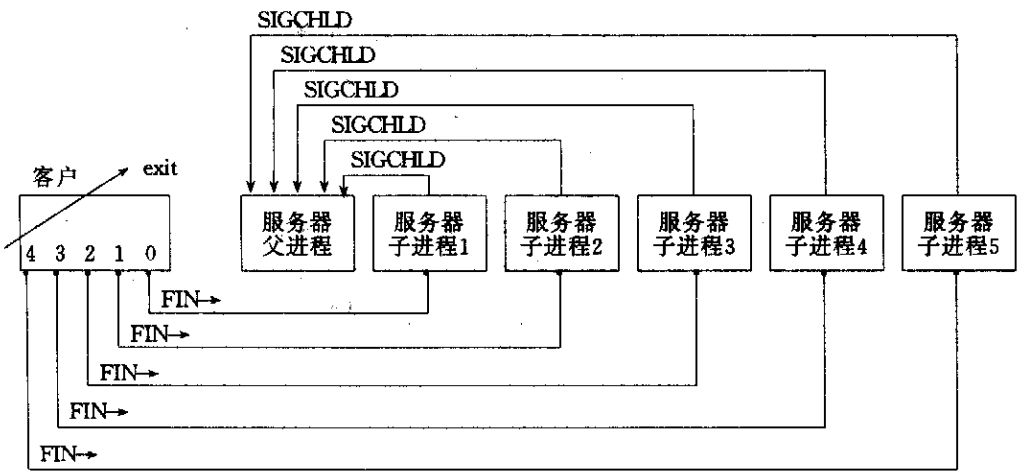


图 5.10 客户终止导致关闭 5 个连接,终止 5 个子进程

正是这种同一信号多个实例的递交造成了问题,这我们不久将看到。

首先,我们在后台运行服务器,接着运行新客户。我们的服务器是由图 5.2 修改而来,它调用函数 signal 来建立如图 5.7 所示的 SIGCHLD 信号处理程序。

```
bsdi % tcpserv03 &
[1]21282
bsdi % tcpcli04 206.62.226.35
hello                          我们键入这一行
hello                          这一行被回射回来
^D                             我们再键入 EOF 字符
child 21288 terminated         这是服务器的输出
```

我们注意到的第一件事情是，当我们期望五个子进程都终止时，只有一个 printf 函数输出了。如果运行 ps，我们将发现其他四个子进程仍作为僵尸进程存在。

```
  PTD     TT   STAT      TIME COMMAND
21282     p1   S      0:00.09 tcpserv03
21284     p1   Z      0:00.00 (tcpserv03)
21285     p1   Z      0:00.00 (tcpserv03)
21286     p1   Z      0:00.00 (tcpserv03)
21287     p1   Z      0:00.00 (tcpserv03)
```

建立一个信号处理程序并在其中调用 wait 并不足以防止出现僵尸进程，问题在于：所有 5 个信号都在信号处理程序执行之前产生，而此信号处理程序又只执行一次，因为 Unix 信号一般是不排队的。更严重的是，此问题是不确定的。在我们刚刚运行的例子中，客户与服务器在同一主机上，信号处理程序执行一次，留下四个僵尸进程。但若我们在不同的主机上运行客户和服务器，信号处理程序一般执行两次：一次作为第一个产生的信号的结果，由于另外 4 个信号在信号处理程序执行时发生，所以处理程序一般情况下会再被调用一次，这就留下了三个僵尸进程。但有时，可能依赖于 FIN 到达服务器主机的时机，信号处理程序执行三次或四次。

正确的解决办法是调用 waitpid 而不是 wait，图 5.11 给出了正确处理 SIGCHLD 的函数 sig_chld 版本。这个版本之所以正确是因为我们在循环内调用 waitpid 取得了所有已终止子进程的状态。我们必须指定选项 WNOHANG，它告诉 waitpid 在有未终止的子进程运行时不要阻塞。图 5.7 中，我们不能在循环中调用 wait，因为没有办法防止 wait 在有未终止的子进程运行时阻塞。

图 5.12 给出了我们的服务器程序的最终版本，它正确处理 accept 返回的 EINTR，且建立一个为所有的已终止子进程调用 waitpid 的信号处理程序(图 5.11)。

```
1 #include    "unp.h"
2 void
3 sig_chld(int signo)
4 {
5     pid_t    pid;
6     int      stat;
7     while( (pid = waitpid(-1, &stat,WNOHANG)) > 0)
8         printf("child %d terminated\n",pid);
9     return;
10 }
```

图 5.11　调用函数 waitpid 的 sig_chld 函数最终(正确)版本[tcpcliserv/sigchldwaitpid.c]

本节的目的是示范我们在网络编程时可能会遇到的三种情况：

1. 当派生子进程时，必须捕获信号 SIGCHLD。
2. 当捕获信号时，必须处理被中断的系统调用。
3. SIGCHLD 的信号处理程序必须正确编写，应使用函数 waitpid 以免留下僵尸进程。

我们的 TCP 服务器程序最终版本(图 5.12)和图 5.11 的 SIGCHLD 信号处理程序能处理上述三种情况。

```
1 #include    "unp.h"
2 int
3 main(int argc, char * * argv)
4 {
5     int     listenfd, connfd;
6     pid_t   childpid;
7     socklen_t clilen;
8     struct sockaddr_in cliaddr, servaddr;
9     void    sig_chld(int);
10    listenfd = Socket(AF_INET, SOCK_STREAM, 0)
11    bzero(&servaddr, sizeof(servaddr));
12    servaddr.sin_family = AF_INET;
13    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
14    servaddr.sin_port = htons(SERV_PORT);
15    Bind(listenfd, (SA * ) &servaddr, sizeof(servaddr));
16    Listen(listenfd, LISTENQ);
17    Signal(SIGCHLD, sig_chld);      /* must call waitpid() */
18    for( ; ; ) {
19        clilen = sizeof(cliaddr);
20        if ( (connfd = accept(listenfd, (SA * ) &cliaddr, &clilen)) < 0) {
21            if(errno == EINTR)
22                continue;      /* back to for() */
23            else
24                err_sys("accept error");
25            }
26            if( (childpid = Fork() ) == 0) {      /* child process */
27                  Close(listenfd);      /* close listening socket */
28                  str_echo(connfd);      /* process the request */
29                  exit(0);
30              }
31              Close(connfd)      /* parent closes connected socket */
32        }
33 }
```

图 5.12 处理 accept 返回 EINTR 错误的 TCP 服务器程序最终(正确)版本[tcpcliserv/tcpserv04.c]

5.11 accept 返回前连接夭折

有另外一种情况,类似于前一节中被中断的系统调用例子,它可能导致 accept 返回一个非致命的错误,此时我们只需再调用一次 accept。图 5.13 中所示的分组序列在较忙的服务器(比较典型的是较忙的 Web 服务器)上已出现过。

三路握手完成,连接建立,然后,客户 TCP 发一个 RST(复位)。在服务器端,连接由 TCP 排队,等待服务器进程在 RST 到达后调用 accept。稍后,服务器进程调用 accept。

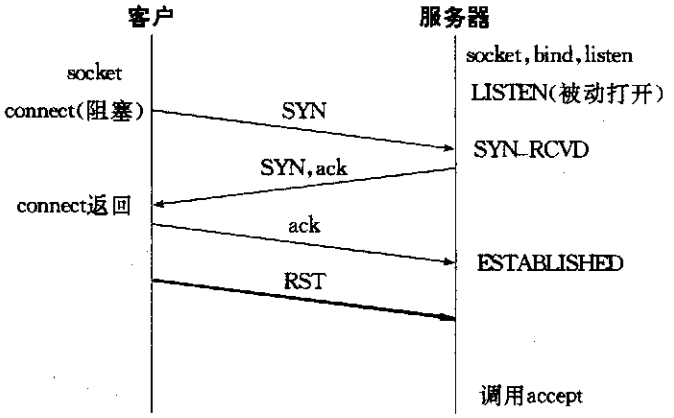


图5.13 ESTABLISHED状态的连接在调用accept之前收到RST

模拟这种情况的一个简单的方法就是:启动服务器,让它调用socket、bind和listen,然后在调用accept之前睡眠一小段时间。在服务器进程睡眠时,启动客户,让它调用socket和connect,一旦connect返回,就设置套接口选项SO_LINGER以生成RST(它将在7.5节中讨论,并在图15.21中给出一个例子),然后终止。

遗憾的是,对于夭折的连接将如何处理是依赖于不同的实现的。源自Berkeley的实现在内核中完成处理夭折的连接,服务器进程根本看不到它。然而,大多数SVR4实现返回一个错误给进程作为accept的返回,此错误与实现方式有关。这些SVR4实现返回一个EPROTO("协议错")errno值,而Posix.1g指出,返回必须是ECONNABORTED("软件引起的连接夭折")。Posix.1g作修改的理由是:当子系统中出现某些致命的协议相关事件时也返回EPROTO,为客户建立的连接非致命夭折而返回相同的错误使得服务器不可能知道是否该再次调用accept。在ECONNABORTED错误情况下,服务器可以忽略错误而简单地再调用accept一次。

Solaris 2.6实现了Posix.1g所做的修改。

源自Berkeley内核中从不传递此错误给进程的做法所涉及的步骤可在TCPv2中找到踪迹。RST在第964页做了处理,导致函数tcp_close被调用,此函数又调用第897页的函数in_pcbdetach,然后它又调用第719页的函数sofree。该函数(第473页)发现待夭折的套接口仍在监听套接口的已完成连接队列,于是从该队列中删去并释放此套接口。当服务器接着调用函数accept时,它根本不知道有一个已完成的连接已从队列中删除掉。

在15.6节我们将再次回到这些夭折的连接,看看在与函数select和正常阻塞模式下的监听套接口混用时如何出现问题的。

5.12 服务器进程终止

现在我们启动客户-服务器，然后杀死服务器子进程。这是为了模拟服务器进程的崩溃，来观察客户将会发生什么。(我们必须小心区别服务器进程崩溃与5.14节中讨论的服务器主机崩溃。)步骤如下：

1. 我们在不同的主机上启动服务器和客户，并在客户上键入一行，若正常，此行应由服务器子进程回射。
2. 找到服务器子进程的进程ID，并杀死该进程。作为进程终止处理的部分工作，子进程中打开的描述字都关闭，这引发了一个FIN发送给客户，客户TCP相应地以ACK响应。这就是TCP连接终止的前一半工作。
3. 信号SIGCHLD被发往服务器父进程并正确处理(图5.12)。
4. 客户上毫无动静。客户TCP从服务器TCP接收FIN并以ACK响应，但问题是客户进程正阻塞于fgets调用，等待从终端上得到一行。
5. 此时，在客户的另外一个窗口上运行命令netstat，以观察客户套接口的状态。

```
solaris % netstat | grep 9877
Local Address      Remote Address  Swind  Send-Q  Rwind Recv-Q  State
solaris.34673      bsdi.9877       8760       0    8760       0  CLOSE_WAIT
```

(这是第一次我们在solaris系统上给出命令netstat的输出，所以加上了标题行，其格式与BSD系统的输出格式略有不同，但内容是相似的。)同时，我们也在服务器的另一窗口上执行命令netstat：

```
bsdi % netstat | grep 9877
tcp        0      0  bsdi.9877          solaris.34673      FIN_WAIT_2
```

参看图2.4，我们可知，至此已完成了TCP连接终止序列的一半。

6. 我们可以在客户上再键入一行，在第一步中启动的客户将发生如下事件：

```
solaris % tcpcli01 206.62.226.35      启动客户
hello                                 我们键入第一行
hello                                 它被正确回射
                                      在这儿我们杀死服务器主机上的服务器子进程
                                      然后给客户键入下一行
another line
str_cli:server terminated prematurely
```

当我们键入"another line"时，函数ser_cli调用writen，客户TCP给服务器发送数据。这在TCP中是允许的，因为客户TCP接收到FIN只表示服务器进程已关闭了连接的服务器一端，它不再发送任何数据。接收到FIN并没有通知客户TCP服务器进程已终止(在这个例子中它是终止了)。在6.6节中当我们讨论到TCP的半关闭问题时再详细讨论这一点。

当服务器TCP接收到来自客户的数据时，由于先前打开那个套接口的进程已终止，所以以RST响应。我们可以用tcpdump来观察分组，以检查是否发出了RST。

7. 但是，由于客户进程在调用writen后立即调用readline，且由于第2步中接收的FIN，

readline立即返回0(文件结束符),所以客户进程将看不到RST。我们的客户不希望在此时接收文件结束符,所以以错误信息"server terminated prematurely(服务器过早终止)"退出。

8. 当客户终止时(在图5.5中调用err_quit),所有打开的描述字都关闭。

我们上述所讨论的一切也都与例子的时序有关。就像我们刚讨论的,当在不同的主机上运行客户和服务器时,从客户到服务器发送数据(即"another line")和客户接收到服务器的RST需几毫秒时间。这就是为什么客户调用readline返回0的原因,因为较早接收到的FIN现已准备好由客户读入。如果我们在相同的主机上运行客户和服务器,或在客户调用readline之前插入一段短暂停顿,那么接收到的RST就会优先于FIN,导致readline返回一个错误,且errno值为ECONNRESET("对方复位连接")。

本例的问题是,当FIN到达套接口时,客户阻塞于fgets调用。客户工作时有两个描述字——套接口和用户输入,它不能只阻塞于两个源中某个特定源输入(正如str_cli所写),而是应阻塞于任一源的输入。事实上,这正是函数select和poll的一个目的,这两个函数将在第6章中讨论。在6.4节中,我们重写函数str_cli,这样一旦杀死服务器子进程,客户就会收到FIN。

5.13 SIGPIPE信号

如果客户不理会函数readline返回的错误而写入更多的数据到服务器上又会怎样?这种情况是可能发生的,譬如,客户在读回任何东西之前对服务器写两次,而第一次写就引发了一RST。

所用规则是:当一个进程向接收了RST的套接口进行写操作时,内核给该进程发一个SIGPIPE信号。此信号的缺省行为就是终止进程,所以,进程必须捕获它以免不情愿地被终止。

进程不论是捕获了该信号并从其信号处理程序返回,还是不理会该信号,写操作都返回EPIPE错误。

一个在Usenet上经常问及的问题(FAQ)是如何在第一次写时捕获此信号而不是在第二次写时才见到。这是不可能的,接着我们上述讨论,第一次写引发了RST分节,第二次写引发了SIGPIPE信号。写一个已接收了FIN的套接口是可行的,但写一个已接收了RST的套接口则是错误的。

为了看清信号SIGPIPE会发生什么,我们对客户进行了修改,如图5.14所示。

```
1 #include    "unp.h"
2 void
3 str_cli(FILE *fp,int sockfd)
4 {
5     char    sendline[MAXLINE],recvline[MAXLINE];
6     while(Fgets(sendline,MAXLINE,fp) != NULL){
```

```
7          Writen(sockfd,sendline,1);
8          sleep(1);
9          Writen(sockfd,sendline + 1,strlen(sendline) -1);
10         if(Readline(sockfd,recvline,MAXLINE) == 0)
11           err_quit("str_cli:server terminated prematurely");
12         Fputs(recvline,stdout);
13     }
14 }
```

图5.14 调用 writen 两次的函数 str_cli[tcpcliserv/str_cli11.c]

第7～9行 我们所做的修改就是调用 writen 两次：第一次将数据的第一个字节写到套接口，暂停一秒钟，第二次再写同一行中剩余的字节。目的就是为了使第一次调用 writen 引发一个 RST，而第二次调用则生成 SIGPIPE。

如果我们在 BSD/OS 主机上运行客户，则有：

```
bsdi % tcpcli11 206.62.226.34
hi there                    我们键入这一行
hi there                    它由服务器回射回来
                            在这儿我们杀死服务器子进程
bye                         然后键入这一行
bsdi % echo $?              最后一个命令的 ksh 返回值是多少?
269                         269 = 256 + 13
bsdi % grep SIGPIPE /usr/include/sys/signal.h
#define SIGPIPE 13     /* write on a pipe with no one to read it */
```

我们启动客户，键入一行，可以看到此行被正确回射，然后在服务器主机上终止服务器子进程。接着，键入另外一行("bye")，但无任何回射，都回到了 shell 提示符下。由于信号 SIGPIPE 的缺省行为是终止进程，不产生 core 文件，所以 Korn Shell 不输出任何东西。这就是由 SIGPIPE 所终止的程序的问题所在：甚至 shell 都不输出任何内容以指示发生了什么。

我们必须执行 echo $?以输出 shell 的返回值为269。然后，输出常值 SIGPIPE 的数字值，可以看到 Korn Shell 的返回值是256加上该信号值。但如果我们在 Digital Unix 4.0、Solaris 2.5或 UnixWare 2.1.2上执行此程序，KornShell 的返回值是141，即128加上13。

> Korn Shell 的11/16/88版返回128加上信号值，而较新的版本返回256加上信号值。Posix.2仅仅规定该返回值应大于128。其他 shell 还有可能返回其他值。

处理 SIGPIPE 的建议方法取决于它发生时应用进程想做什么。如果没有特殊的事情需做，则将信号处理办法很简单地设置为 SIG_IGN，并假设后续的输出操作将捕捉 EPIPE 错误并终止。如果信号出现时需采取特殊措施(可能需在日志文件中登记)，那么就必须捕获该信号，以便在信号处理程序中执行所有期望的动作。但是必须意识到，如果使用了多个套接口，该信号的递交无法告诉我们是哪个套接口出的错。如果我们确实需要知道是哪个 write 出了错，那么必须要么不理会该信号，要么从信号处理程序返回后再处理来自 write 的 EPIPE。

5.14 服务器主机崩溃

我们的下一种情况是看看在服务器主机崩溃时将发生什么。为了模拟这种情况，我们必须在不同的主机上运行客户和服务器。我们先启动服务器，再启动客户，然后在客户上键入一行以确认连接已建立，从网络上断开服务器主机后再在客户上键入另一行。这样一来同时也模拟了当客户发送数据时服务器主机不可达的情况(即建立连接后某些中间路由器不工作)。

步骤如下：

1. 当服务器主机崩溃时，已有的网络连接上发不出任何东西。这里我们假设主机崩溃，而不是由操作员执行命令关机(这在5.16节讨论)。
2. 我们在客户上键入一行，它由 writen(图5.5)所写，由客户 TCP 当作一个数据分节发出，然后，客户就阻塞于 readline 调用，等待回射。
3. 如果我们用 tcpdump 观察网络就会发现，客户 TCP 持续重传数据分节，试图从服务器上接收一个 ACK。TCPv2的25.11节给出了 TCP 重传一个典型模式：源自 Berkeley 的实现重传数据分节12次，放弃前等待约9分钟。当客户 TCP 最后终于放弃时(假设在这段时间内，服务器主机没有重新启动，或者如果服务器主机没有崩溃但网络不可达的情况，则假设网络仍然不可达)，返回客户进程一个错误。因为客户阻塞于 readline 的调用，它返回一个错误。假设服务器主机已崩溃，对客户的数据分节根本就没有响应，则错误为 ETIMEDOUT；但如果某些中间路由器判定服务器主机不可达，且以一个目的地不可达的 ICMP 消息响应，则错误是 EHOSTUNREACH 或 ENETUNREACH。

尽管最终我们的客户还是发现了对方已崩溃或不可达，但有时候我们需要比等待9分钟快许多地检测出这种情况。方法就是对 readline 的调用设置一超时，这在13.2节再作讨论。

我们刚刚讨论的情况只有在我们向服务器主机发送数据时，才能检测出它已经崩溃。如果我们不主动向它发送数据也想检测出服务器主机的崩溃，那需要采用另外一个技术。在7.5节我们将讨论套接口选项 SO_KEEPALIVE，在21.5节将讨论一些客户-服务器心博函数。

5.15 服务器主机崩溃后重启

在这种情况中，我们在客户与服务器之间建立连接，然后假设服务器主机崩溃并重启。在前一节中，当我们发送数据时，服务器主机仍处于崩溃状态，而本节，我们将在发送数据前重新启动服务器主机。模拟这种情况的最简单方法就是：建立连接，从网络上断开服务器，关闭服务器主机，然后重启服务器主机，重新连接服务器主机入网。我们不想让客户知道服务器主机的关机(这点我们将在5.16节讨论)。

正如前一节所提到的，如果客户在服务器主机崩溃时不主动发数据给服务器，那么客户是不会知道服务器已崩溃的(假设不用套接口选项 SO_KEEPALIVE)。采用步骤如下：

1. 我们启动服务器和客户，并键入一行以确认连接已建立。
2. 服务器主机崩溃并重启。
3. 在客户上键入一行，它将作为TCP数据分节发往服务器主机。
4. 当服务器主机崩溃后重启时，它的TCP丢失了崩溃前的所有连接信息，所以服务器TCP对接收的客户数据分节以RST响应。
5. 当RST到达时，客户阻塞于readline调用，导致它返回ECONNRESET错误。

如果客户检测服务器主机的崩溃与否很重要，即使客户不主动发送数据也这样，那就需有其他技术支持(诸如套接口选项SO_KEEPALIVE或某些客户-服务器心博函数)。

5.16 服务器主机关机

前面两节讨论了服务器主机崩溃或无法通过网络到达的情况，现在，我们考虑一下当服务器进程正在运行时，操作员关闭该服务器主机将会发生什么。

当Unix系统关机时，一般是由init进程给所有进程发信号SIGTERM(我们可以捕获此信号)，等待一段固定时间(常常是5秒～20秒)，然后给还在运行的所有进程发信号SIGKILL(此信号我们不能捕获)。这就给了所有运行进程一小段时间来清除和终止。如果我们不捕获信号SIGTERM和终止，服务器将由信号SIGKILL终止。当进程终止时，所有打开的描述字都关闭，然后按5.12节中所述步骤进行。 与在该一节所叙述的一样，我们必须在客户上使用函数select或poll，使得客户在服务器进程开始终止时就检测到。

5.17 TCP程序例子小结

在客户和服务器可以互相通信之前，每一方都必须指定连接的套接口对：本地IP地址、本地端口、远程IP地址、远程端口。在图5.15中我们给出了这四个值，此图是从客户的角度出发的。远程IP地址和远程端口必须在客户调用connect时指定，而两个本地值则一般是由内核作为connect的一部分来选定。客户也可在调用connect之前，通过调用bind来指定其中一个或全部(两个)本地值，但这不常用。

正如第4.10节所述，客户也可在连接建立后通过调用getsockname来获取由内核指定的两个本地值。

图5.16也给出了相同的四个值，但是这是从服务器的角度出发。

本地端口(服务器的众所周知端口)由bind指定。尽管服务器可以通过捆绑一个非通配的IP地址来限定接收目标为某个特定本地接口的连接，它通常是在bind调用中指定通配IP地址。如果服务器在一个多宿主机上绑定一个通配IP地址，它可以在建立连接后调用getsockname来确定本地IP地址，两个远程值则由accept来返回给服务器。我们在4.10节中已提到，如果另外一个程序由调用accept的服务器通过调用exec来执行，则这个新程序可以在必要时调用getpeername来确定客户的IP地址和端口号。

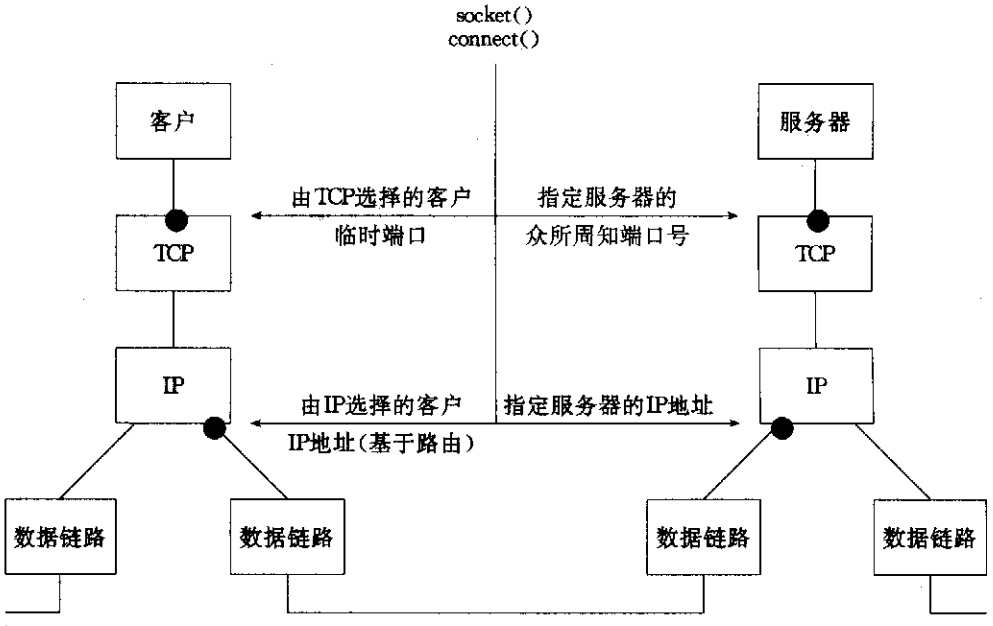


图5.15　从客户的角度总结 TCP 客户-服务器

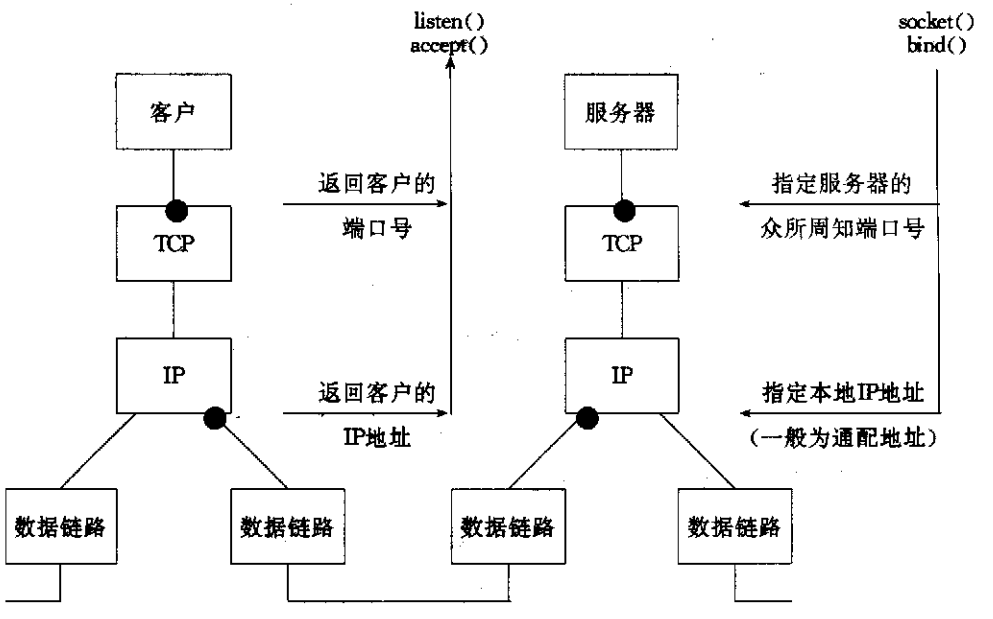


图5.16　从服务器的角度总结 TCP 客户-服务器

5.18 数据格式

在我们的例子中，服务器对从客户上接收的请求从不检查，它只管读入到换行符(包括换行符)的所有数据发回给客户，所搜索的仅仅是换行符。这只是一个例外而不是通常规则，一般来说，我们必须关心在客户和服务器之间进行交换的数据格式。

例子：在客户与服务器之间传递文本串

修改我们的服务器程序，它仍然从客户读入一行文本，但现在，服务器期望这一行包含由空格隔开的两个整数，服务器返回这两个整数的和。我们的客户和服务器 main 函数仍保持不变，str_cli 函数也保持不变，所有修改都在函数 str_echo 中，如图5.17所示。

```
1 #include    "unp.h"
2 void
3 str_echo(int sockfd)
4 {
5     long    arg1,arg2;
6     ssize_t n;
7     char    line[MAXLINE];
8     for( ; ;) {
9         if( (n = Readline(sockfd,line,MAXLINE)) == 0)
10            return;          /* connection closed by other end */
11        if(sscanf(line,"%ld%ld",&arg1,&arg2) == 2)
12            snprintf(line,sizeof(line),"%ld\n",arg1 + arg2);
13        else
14            snprintf(line,sizeof(line),"input error\n");
15        n = strlen(line);
16        Writen(sockfd,line,n);
17    }
18 }
```

图5.17 对两个数求和的函数 str_echo[tcpcliserv/str_echo08.c]

第11～14行 我们调用函数 sscanf 来将文本串中的两个参数转换为长整数，然后由函数 snprintf 来将结果转换为文本串。

不论客户和服务器主机的字节序如何，这个新的客户和服务器都能工作得很好。

例子：在客户与服务器之间传递二进制结构

现在，将我们的客户和服务器程序修改为跨套接口传递二进制值而不是文本串。我们将看到，当客户和服务器在不同字节序的主机上运行或是在不支持相同大小长整型的主机上运行时，客户和服务器便无法工作(图1.17)。

我们的客户和服务器程序 main 函数不作任何修改。我们给两个函数参数定义一个结构，给结果定义另外一个结构，并将这两个定义都放在头文件 sum.h 中，如图5.18所示。图5.19给出了函数 str_cli。

```
1 struct args {
2         long    arg1;
3         long    arg2;
```

```
4 };
5 struct result {
6         long      sum;
7 };
```

图5.18　头文件 sum.h[tcpcliserv/sum.h]

```
1 #include     "unp.h"
2 #include     "sum.h"
3 void
4 str_cli(FILE *fp,int sockfd)
5 {
6     char        sendline[MAXLINE];
7     struct args args;
8     struct result result;
9     while(Fgets(sendline,MAXLINE,fp) != NULL) {
10        if(sscanf(sendline,"%ld%ld",&args.arg1,&args.arg2) != 2) {
11          printf("invalid input: %s",sendline);
12          continue;
13        }
14        Writen(sockfd,&args,sizeof(args));
15        if (Readn(sockfd,&result,sizeof(result))==0)
16          err_quit("str_cli:server terminated prematurely");
17        printf("%ld\n",result.sum);
18    }
19 }
```

图5.19　发送两个二进制整数给服务器的函数 str_cli[tcpcliserv/str_cli09.c]

第10～14行　sscanf 将两个参数从文本串转换为二进制数，我们接着调用 writen 将此结构发送给服务器。

第15～17行　我们调用 readn 来读应答，并用 printf 来输出结果。

图5.20给出了函数 str_echo。

```
1 #include     "unp.h"
2 #include     "sum.h"
3 void
4 str_echo(int sockfd)
5 {
6     ssize_t n;
7     struct args args;
8     struct result result;
9     for( ; ;) {
10        if( (n = Readn(sockfd,&args,sizeof(args))) == 0)
11            return;          /* connection closed by other end */
12        result.sum = args.arg1 + args.arg2;
13        Writen(sockfd,&result,sizeof(result));
14    }
15}
```

图5.20　对两个二进制整数求和的函数 str_echo[tcpcliserv/str_echo09.c]

第9～14行　我们通过调用 readn 来读参数、计算并存储两数之和，然后调用 writen 将结果结构发回。

如果我们在具有相同体系结构的主机上运行我们的客户和服务器(如图1.16中的solaris和sunos5)则什么问题都没有。下面是客户的交互过程:

```
sunos5 % tcpcli09 206.62.226.33
11  22                          我们键入这两个数
33                              这个数是服务器的应答
-11  -44
-55
```

但是,当我们在具有不同体系结构的两台主机上运行客户和服务器时(服务器在大端sparc系统solaris上运行,而客户在小端Intel系统bsdi上运行),它们就无法工作了。

```
bsdi % tcpcli09 206.62.226.33
1  2                           我们键入这两个数
3                              结果正确
-22  -77                       我们再键入另外两个数
-16777314                      结果错误
```

问题就在于跨套接口发送的两个二进制整数在客户上是小端格式,但被服务器解释为大端格式。我们发现,对于正整数,它看起来能工作,但对于负整数则不行了(见习题5.8)。此例有三个潜在的问题:

1. 不同的实现以不同的格式存储二进制数,最常用的方法便是大端格式与小端格式,这在第3.4节已做过介绍。
2. 不同的实现在存储相同的C数据类型时可能不同。例如,大多数32位的Unix系统用33位表示长整数,但64位系统却常常以64位来表示此相同的数据类型(图1.17)。对short、int或long数据类型的大小没有确定的保证。
3. 不同的实现给结构打包的方式也是不同的,取决于所用数据类型的位数及机器的对齐限制。因此,跨套接口来传送二进制结构是很不明智的。

有两个常用方法来解决此数据格式问题:

1. 把所有的数值数据作为文本串来传递,如图5.17所示,当然这也要以两个主机有相同的字符集为基础。
2. 显式定义所支持数据类型的二进制格式(位数,大端或小端),在客户与服务器之间以此格式传递所有数据。远程过程调用(RPC)软件包常用此技术。RFC 1832[Srinivasan 1995]描述了Sun RPC软件包所用的外部数据表示(XDR,External Data Representation)标准。

5.19 小 结

我们的回射客户-服务器程序第一版总共约150行(包括函数readline和writen),然而留下了待检查的许多细节问题。我们遇到的第一个问题是僵尸进程问题,通过捕获信号SIGCHLD来处理。接着,信号处理程序调用waitpid,我们已证明必须调用此函数而不是早期的wait函数,因为Unix信号是不排队的。这引导我们进入了Posix信号处理的一些细节问题,关于这一问题在APUE的第10章中还有许多附加信息。

我们遇到的下一问题是当服务器进程终止时，客户不知道。我们看到，客户的TCP是被通知了，但客户本身由于正阻塞于等待用户输入而未接收到此通知。我们在第6章中将用函数select和poll来处理这种情况，它等待多个描述字中的任一个准备好而不是阻塞于单个描述字。

我们也发现，如果服务器主机崩溃，也要等到客户向该服务器发送了数据才能检测到。一些应用进程必须尽早意识到这个事实，在7.5节中我们将利用套接口选项SO_KEEPALIVE，在21.5节我们将开发一组客户-服务器心博函数来解决这个问题。

我们的简单例子是交换文本行，由于服务器根本不检查它回射的行，所以它工作得很好，但在客户与服务器之间发送数值数据时将引发一组新问题，文中已经讲述。

5.20 习 题

5.1 从图5.2和5.3中构造一个TCP服务器程序，从图5.4和图5.5中构造一个TCP客户程序。先启动服务器，然后启动客户，键入一新行以确认客户和服务器工作正常。键入文件结束符以终止客户，记下时间。在客户主机上使用命令netstat来查看连接的客户端是否进入TIME_WAIT状态，此后每5秒左右执行一次netstat来查看TIME_WAIT状态何时结束。此实现的MSL(最长分节生命期)是什么？

5.2 对于我们的客户-服务器程序，如果我们运行客户并将标准输入重定向到一个二进制文件将会怎样？

5.3 利用Telnet客户来与我们的回射服务器通信，它跟我们的回射客户-服务器有什么区别？

5.4 在5.12节的例子中，我们用命令netstat观察套接口状态来确认连接终止的前两个分节已发送(服务器发出的FIN和客户发出的对此分节的ACK)。后两个分节交换吗(客户发出的FIN和服务器发出的对此分节的ACK)？如果交换，它们何时交换？如果不交换，为什么？

5.5 在5.14节给出的例子中，如果我们在第二步与第三步之间，在服务器主机上重启服务器应用进程将会怎样？

5.6 为了验证在5.13节中我们所说的SIGPIPE发生的一切，我们对图5.4作如下修改。编写一个SIGPIPE信号处理程序，它仅输出一个消息便返回；在调用connect之前建立此信号处理程序。将服务器的端口号改为13，即daytime服务器。当连接建立后，睡眠2秒钟，写几个字节到套接口上，再睡眠2秒钟，写另外几个字节到套接口上。运行程序，观察它将会怎样？

5.7 在图5.15中，如果由客户在connect调用中将服务器主机的IP地址指定为与服务器主机最右边的数据链路相关的IP地址，而不是与服务器主机最左边的数据链路相关的IP地址，它将会怎样？

5.8 在图5.20的例子输出中，当客户和服务器分别在不同的大、小端系统上时，对于小的正整数，例子工作得很好，但对于小的负整数则不能正常工作，为什么？(提示：画一个与图3.8类似的跨套接口数字交换图。)

5.9 在图5.19和图5.20的例子中，我们可以通过让客户利用函数htonl来将两个参数

转换为网络字节序，让服务器在做加法前调用 ntohl 来转换每个参数，并对结果做类似的转换来解决字节序问题可以吗？

5.10 如果客户在某台以32位存储长整数的 Sparc 主机上，而服务器在以64位存储长整数的 Digital Alpha 主机上，图5.19和图5.20中的例子将会怎样？如果将客户和服务器在这两台主机上交换，结果会改变吗？

5.11 在图5.15中，我们说客户 IP 地址是由 IP 基于路由选定的，这是什么含义？

第 6 章　I/O 复用：select 和 poll 函数

6.1　概　述

在 5.12 节中，我们看到 TCP 客户同时处理两个输入：标准输入和 TCP 套接口。我们遇到的问题是客户阻塞于（标准输入上的）fgets 调用，而服务器进程又被杀死。服务器 TCP 虽正确地给客户 TCP 发了一个 FIN，但客户进程正阻塞于从标准输入读入，它直到从套接口读时才能看到此文件结束符（可能已过了很长时间）。我们需要这样的能力：如果一个或多个 I/O 条件满足（例如，输入已准备好被读，或者描述字可以承接更多的输出）时，我们就被通知到。这个能力被称为 I/O 复用，是由函数 select 和 poll 支持的，我们也对较新的 Posix.1g 的变种（称为 pselect）作介绍。

I/O 复用典型地用在下列网络应用场合：

- 当客户处理多个描述字时（一般是交互式输入和网络套接口），必须使用 I/O 复用，这在前一段中已做了描述。
- 一个客户同时处理多个套接口是可能的，但很少出现。在 15.5 节一个 Web 客户的上下文中，我们给出使用 select 的例子。
- 如果一个 TCP 服务器既要处理监听套接口，又要处理已连接套接口，一般也要用到 I/O 复用，如 6.8 节所述。
- 如果一个服务器即要处理 TCP，又要处理 UDP，一般也要使用 I/O 复用，8.15 节中我们也要给出这样的一个例子。
- 如果一个服务器要处理多个服务或者多个协议（例如，我们将在 12.5 节描述的 inetd 守护进程），一般要使用 I/O 复用。

I/O 复用并非只限于网络编程，许多正式应用程序也需要使用这项技术。

6.2　I/O 模型

在介绍函数 select 和 poll 之前，我们需回过头来看看整体，检查 Unix 下我们可用的五个 I/O 模型的基本区别：

- 阻塞 I/O
- 非阻塞 I/O
- I/O 复用（select 和 poll）
- 信号驱动 I/O（SIGIO）
- 异步 I/O（Posix.1 的 aio_系列函数）

你在第一次阅读时，可能想略读本节，到后面的章节中详细介绍不同的I/O模型时才回来看。

正如本节我们所给出的所有例子所述，一个输入操作一般有两个不同的阶段：

1. 等待数据准备好。
2. 从内核到进程拷贝数据。

对于一个套接口上的输入操作，第一步一般是等待数据到达网络，当分组到达时，它被拷贝到内核中的某个缓冲区，第二步是将数据从内核缓冲区拷贝到应用缓冲区。

阻塞I/O模型

最流行的I/O模型是阻塞I/O模型，本书中到目前为止的所有例子都使用此模型。缺省时，所有套接口都是阻塞的。以数据报套接口作为例子，我们有示于图6.1中的情形。

此例中，我们用UDP而不是TCP，因为对于UDP来说，数据准备好的概念要简单些：整个数据报是否已接收。而对于TCP则要复杂得多，需考虑诸如套接口的低潮限度(lowwater mark)这样的许多附加变量。

在本节的例子中，我们将函数recvfrom视为系统调用，因为我们正考虑应用进程与内核的区别。不论函数recvfrom如何实现(在源自Berkeley的内核中作为系统调用，在系统V内核中作为调用系统调用getmsg的函数)，一般都有一个从应用进程中运行到内核中运行的切换，一段时间后再跟一个返回到应用进程的切换。

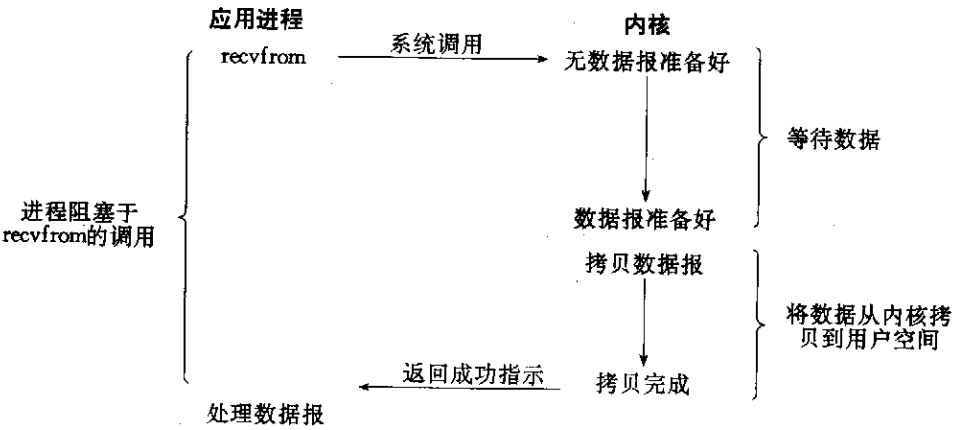


图6.1 阻塞I/O模型

在图6.1中，进程调用recvfrom，此系统调用直到数据报到达且拷贝到应用缓冲区或是出错才返回。最常见的错误是系统调用被信号中断，如5.9节所述。我们所说进程阻塞的整段时间是指从调用recvfrom开始到它返回的这段时间，当进程返回成功指示时，应用进程开始处理数据报。

非阻塞I/O模型

当我们把一个套接口设置成非阻塞方式时，即通知内核：当请求的I/O操作非得让进程睡眠不能完成时，不要让进程睡眠，而应返回一个错误。我们将在第15章详细介绍非阻塞I/

O，但为了说明我们所考虑的例子，在图 6.2 中作一个小结性描述。

前三次调用 recvfrom 时仍无数据返回，因此内核立即返回一个 EWOULDBLOCK 错误。第四次调用 recvfrom 时，数据报已准备好，被拷贝到应用缓冲区，recvfrom 返回成功指示，接着就是我们处理数据。

当一个应用进程像这样对一个非阻塞描述字循环调用 recvfrom 时，我们称此过程为轮询(polling)。应用进程连续不断地查询内核，看看某操作是否准备好，这对 CPU 时间是极大的浪费，但这种模型只是偶尔才遇到，一般是在只专门提供某种功能的系统中才有。

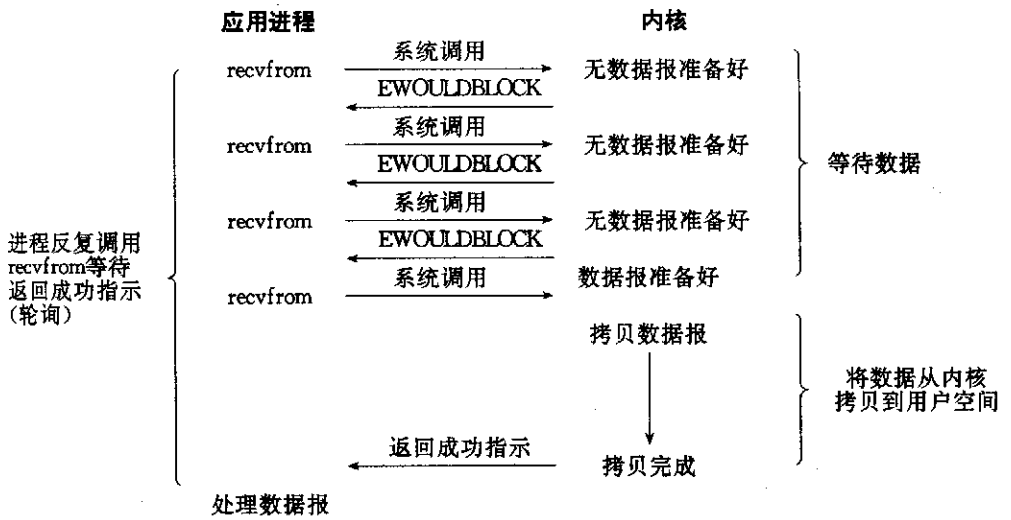


图 6.2　非阻塞 I/O 模型

I/O 复用模型

有了 I/O 复用，我们就可以调用 select 或 poll，在这两个系统调用中的某一个上阻塞，而不是阻塞于真正的 I/O 系统调用。图 6.3 是 I/O 复用模型的一个小结。

我们阻塞于 select 调用，等待数据报套接口可读。当 select 返回套接口可读条件时，我们调用 recvfrom 将数据报拷贝到应用缓冲区中。

将图 6.3 与图 6.1 进行比较，似乎没有显示什么优越性，实际上，因使用了系统调用 select，要求两次系统调用而不是一次，好像变得还有点差。但是，在本章的后面我们将看到，使用 select 的好处在于我们可以等待多个描述字准备好。

信号驱动 I/O 模型

我们也可以用信号，让内核在描述字准备好时用信号 SIGIO 通知我们，我们将此方法称为信号驱动 I/O，图 6.4 对此作了一个小结。

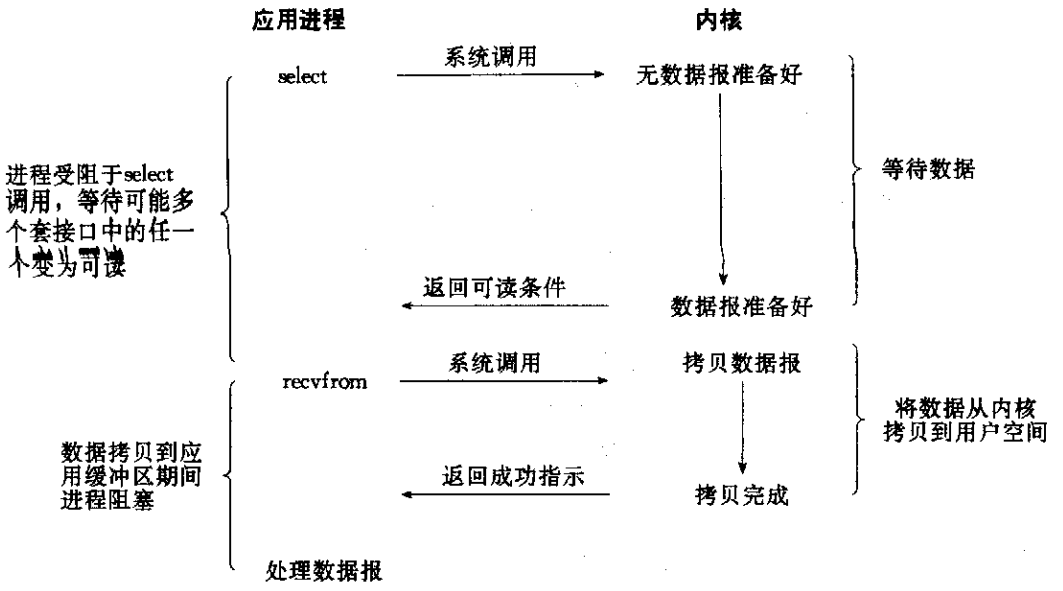


图 6.3 I/O 复用模型

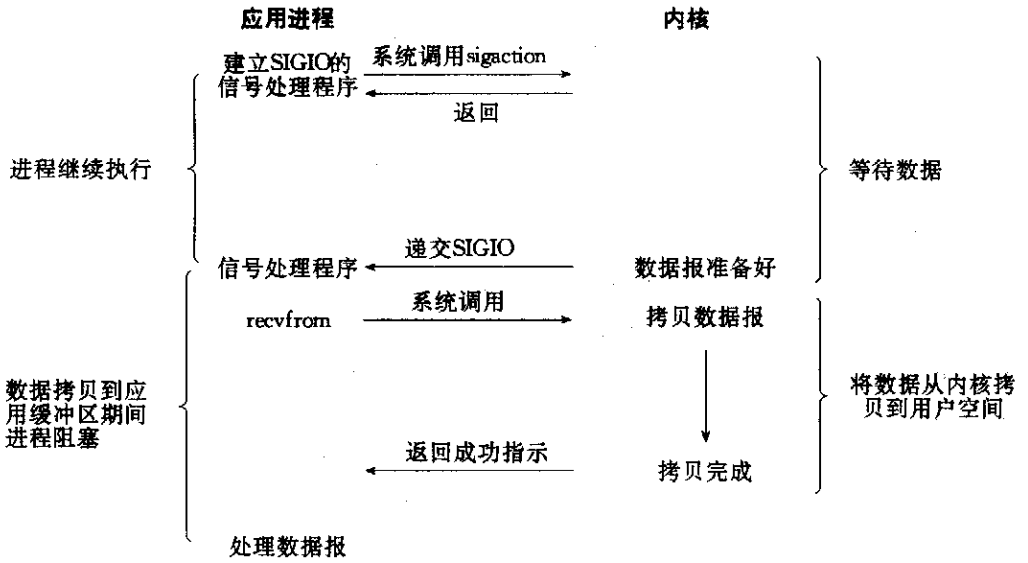


图 6.4 信号驱动 I/O 模型

首先，我们允许套接口进行信号驱动 I/O（我们将在 22.2 节对此进行讨论），并通过系统调用 sigaction 安装一个信号处理程序。此系统调用立即返回，进程继续工作，它是非阻塞的。当数据报准备好被读时，就为该进程生成一个 SIGIO 信号。我们随即可以在信号处理程序中调用 recvfrom 来读数据报，并通知主循环数据已准备好被处理（这正是我们在 22.3 节中所要做的事情）。也可以通知主循环，让它来读数据报。

无论我们如何处理 SIGIO 信号，这种模型的好处是当等待数据报到达时，可以不阻塞。主循环可以继续执行，只是等待信号处理程序的通知：或者数据已准备好被处理，或者数据报已准备好被读。

异步 I/O 模型

异步 I/O 是 Posix.1 的 1993 版本中的新内容（“实时”扩展）。我们让内核启动操作，并在整个操作完成后（包括将数据从内核拷贝到我们自己的缓冲区）通知我们。因为这种模型还没有广泛使用，本书不作讨论。这种模型与前一节介绍的信号驱动模型的主要区别在于：信号驱动 I/O 是由内核通知我们何时可以启动一个 I/O 操作，而异步 I/O 模型是由内核通知我们 I/O 操作何时完成。图 6.5 给出了一个例子。

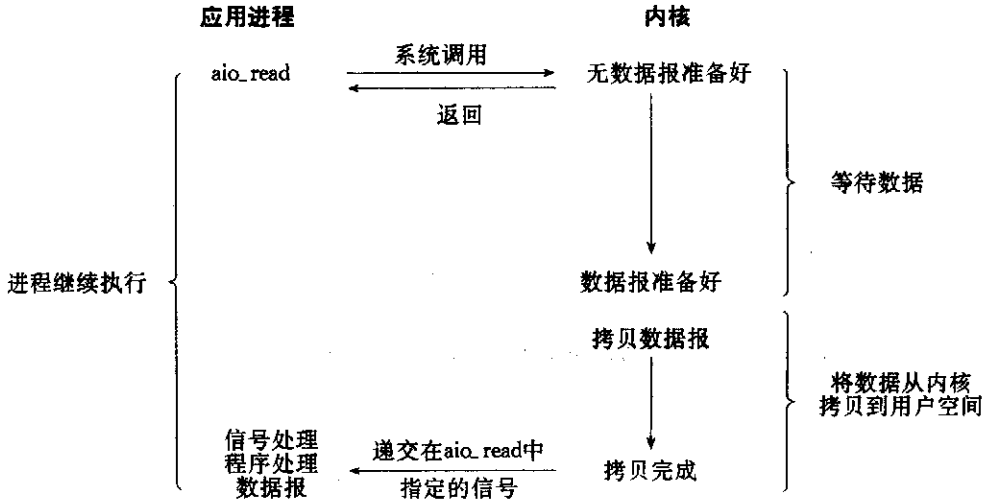


图 6.5　异步 I/O 模型

我们调用函数 aio_read（Posix 异步 I/O 函数以 aio_或 lio_开头），给内核传递描述字、缓冲区指针、缓冲区大小（与 read 相同的三个参数）、文件偏移（与 lseek 类似），并告诉内核当整个操作完成时如何通知我们。此系统调用立即返回，我们的进程不阻塞于等待 I/O 操作的完成。在此例子中，我们假设要求内核在操作完成时生成一个信号，此信号直到数据已拷贝到应用缓冲区才产生，这一点是与信号驱动 I/O 模型不同的。

> 正如本书所述，很少有系统支持 Posix.1 的异步 I/O 模型。例如，我们还不能确定系统是否支持套接口上的这种模型。这儿我们用它，只是作为一个与信号驱动 I/O 模型进行比较的例子。

各种 I/O 模型的比较

图 6.6 示出了上述五种不同 I/O 模型的比较。它表明：前四种模型的主要区别都在第一阶段，因为前四种模型的第二阶段基本相同：在数据从内核拷贝到调用者的缓冲区时，进程阻塞于 recvfrom 调用。然而，异步 I/O 模型处理的两个阶段都不同于前四个模型。

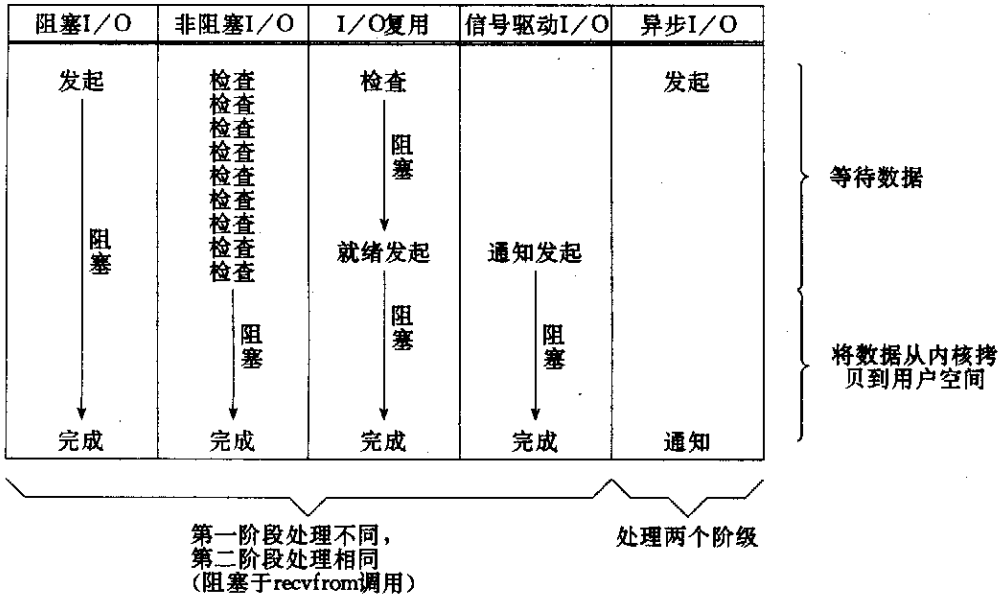


图 6.6 五个 I/O 模型的比较

同步 I/O 与异步 I/O

Posix.1 定义这两个术语如下：

- 同步 I/O 操作引起请求进程阻塞，直到 I/O 操作完成。
- 异步 I/O 操作不引起请求进程阻塞。

根据上述定义，我们的前四个模型——阻塞 I/O 模型、非阻塞 I/O 模型、I/O 复用模型和信号驱动 I/O 模型都是同步 I/O 模型，因为真正的 I/O 操作(recvfrom)阻塞进程，只有异步 I/O 模型与此异步 I/O 的定义相匹配。

6.3 select 函数

这个函数允许进程指示内核等待多个事件中的任一个发生，并仅在一个或多个事件发生或经过某指定的时间后才唤醒进程。

作为一个例子，我们可以调用函数 select 并通知内核仅在下列情况发生时才返回：

- 集合{1,4,5}中的任何描述字准备好读，或
- 集合{2,7}中的任何描述字准备好写，或
- 集合{1,4}中的任何描述字有异常条件待处理，或
- 已经过了 10.2 秒

也就是说，通知内核我们对哪些描述字感兴趣(读、写或异常条件)以及等待多长时间。我们所关心的描述字不受限于套接口：任何描述字都可用 select 来测试。

> 源自 Berkeley 的实现已经允许任何描述字的 I/O 复用。刚开始，SVR3 还限制 I/O 复用只适用于流设备(第 33 章)的描述字，但 SVR4 中就没有了这个限制。

```
#include <sys/select.h>
#include <sys/time.h>

int select (int maxfdp1,fd_set *readset,fd_set *writeset,fd_set * exceptset,const struct
        timeval *timeout);
                                返回：准备好描述字的正数目，0——超时，-1——出错
```

我们从此函数的最后一个参数开始介绍，它告诉内核等待一组指定的描述字中的任一个准备好可花多长时间，结构 timeval 指定了秒数和微秒数成员。

```
struct timeval {
    long tv_sec; /* seconds */
    long tv_usec; /* microseconds */
};
```

有三种可能：

1. 永远等待下去：仅在有一个描述字准备好 I/O 时才返回，为此，我们将参数 timeout 设置为空指针。
2. 等待固定时间：在有一个描述字准备好 I/O 时返回，但不超过由 timeout 参数所指 timeval 结构中指定的秒数和微秒数。
3. 根本不等待：检查描述字后立即返回，这称为轮询(polling)。为了实现这一点，参数 timeout 必须指向结构 timeval，且定时器的值(由结构 timeval 指定的秒数和微秒数)必须为 0。

在前两者情况的等待中，如果进程捕获了一个信号并从信号处理程序返回，那么等待一般被中断。

> 源自 Berkeley 的内核从不自动重启函数 select(TCPv2 第 527 页)，但如果在安装信号处理程序时指明了标志 SA_RESTART，SVR4 就自动重启被中断的 select。这意味着，为了实现可移植性，我们在捕获信号时，必须准备好函数 select 返回 EINTR 错误。

虽然结构 timeval 为我们指定了一个微秒级的分辨率，但内核支持的分辨率却要粗糙得多。例如，很多 Unix 内核将超时值向上舍入成 10ms 的倍数。另外还有调度延迟现象，即定时器时间到后内核还需花一点时间调度相应进程的运行。

参数 timeout 前的限定词 const 表示它在返回时不会被 select 修改。例如，如果我们规定了一个 10 秒的时间限制，且在定时器时间到之前 select 返回，或是有一个或多个描述字准备好，或是返回 EINTR 错误，那么 timeout 所指 timeval 结构在函数返回时是不会更新成剩余的秒数的。如果我们想知道这个值，则在调用 select 函数之前必须得到系统时间，返回后再取得系统时间，两者相减即可。

> 现行的 Linux 系统修改了结构 timeval，因此，为了可移植性，需假设结构 timeval 在返回时无定义，并在每次调用 select 之前对它进行初始化。Posix.1g 规定使用限定词 const。

中间的三个参数 readset、writeset 和 exceptset 指定我们要让内核测试读、写和异常条件所需的描述字。现在只支持两个异常条件：

1. 套接口带外数据的到达，对此，我们在第 21 章中再作更为详细的描述。
2. 控制状态信息的存在，可从一个已置为分组方式的伪终端主端读到。本卷我们不讨论伪终端。

有一个设计上的问题，即如何为这三个参数的每一个指定一个或多个描述字值。函数 select 使用描述字集，它一般是一个整数数组，每个数中的每一位对应一个描述字。例如，用 32 位整数，则数组的第一个元素对应于描述字 0～31，数组中的第二个元素对应于描述字 32～63，以此类推。所有的实现细节都与应用程序无关，它们隐藏在数据类型 fd_set 和下面的四个宏中：

```
void FD_ZERO (fd_set * fdset);          /* clear all bits in fdset */
void FD_SET (int fd, fd_set * fdset);   /* turn on the bit for fd in fdset */
void FD_CLR (int fd, fd_set * fdset);   /* turn off the bit for fd in fdset */
int  FD_ISSET(int fd, fd_set * fdset);  /* is the bit for fd on in fdset ? */
```

我们分配一个 fd_set 数据类型的描述字集，并用这些宏设置、测试集合中的每一位，我们还可以用 C 语言中的赋值语句将其赋值成另外一个描述字集。

> 我们所讨论的每个描述字占用整数数组中一位的方法仅仅是函数 select 的可能实现之一。不过，将描述字集中的每个描述字称呼为位(bit)是很常见的，如“打开读集合中表示监听描述字的位”。
>
> 在 6.10 节中，我们将看到函数 poll 用了一个完全不同的表示方法：一个可变长的结构数组，每个结构代表一个描述字。

例如，为了定义一个 fd_set 类型的变量，并打开描述字 1、4 和 5 的相应位，我们写如下代码：

```
fd_set rset;

FD_ZERO (&rset);        /* initialize the set: all bits off */
FD_SET (1, &rset);      /* turn on bit for fd 1 */
FD_SET (4, &rset);      /* turn on bit for fd 4 */
FD_SET (5, &rset);      /* turn on bit for fd 5 */
```

对集合的初始化是很重要的。如果集合作为一个自动变量分配而未初始化，那将导致不可预测的后果。

如果我们对某个条件不感兴趣，函数 select 的三个中间参数 readset、writeset 和 exceptset 中相应参数就可设为空指针。实际上，如果三个指针均为空，我们就有了一个比 Unix 的函数 sleep 更为精确的定时器(sleep 睡眠以秒为最小单位)。函数 poll 提供相似功能。APUE 的图 C.9 和 C.10 给出了一个用 select 和 poll 实现的函数 sleep_us，它的睡眠单位为微秒。

参数 maxfdp1 指定被测试的描述字个数，它的值是要被测试的最大描述字加 1(因此，我们将此参数命名为 maxfdp1)，描述字 0、1、2……一直到 maxfdp1－1 均被测试。

头文件<sys/select.h>中定义的常值 FD_SETSIZE，是数据类型 fd_set 的描述字数量，其值通常是 1024，但很少有程序使用那么多的描述字。参数 maxfdp1 强迫我们计算出所

关心的最大描述字并将此值通知内核。例如，前面给出的打开描述字 1、4 和 5 的代码，其 maxfdp1 值就应是 6。是 6 而不是 5 的原因就在于：我们指定的是描述字的个数而非最大值，而描述字是从 0 开始的。

> 这个参数之所以存在，且因其存在而带来了计算其值的负担，完全是为了从效率考虑。每个 fd_set 都为许多描述字（一般为 1024）开有空间，但这要比普通进程所用的数量大得多。内核正是通过在进程与内核间不拷贝不需要的描述字部分和不测试常为 0 的相应位来获得效率的（TCPv2 的 16.13 节）。

函数 select 修改由指针 readset、writeset 和 exceptset 所指的描述字集。这三个参数均为值-结果参数。当我们调用函数时，指定我们所关心的描述字集，当返回时，结果指示哪些描述字已准备好。返回时，我们用宏 FD_ISSET 来测试结构 fd_set 中的描述字。描述字集中任何与没有准备好的描述字相对应的位返回时清成 0。为此，每次调用 select 时，我们都得将所有描述字集中关心的位都置为 1。

> 当使用 select 时，两个最常见的编程错误是：忘了对最大描述字加 1，忘了描述字集是值-结果参数。第二个错误导致调用 select 时，描述字集中我们认为是 1 的位却置为 0。作者在调试本书中的一个例子时，因忘了对 select 的第一个参数加 1 而浪费了 2 个小时。

此函数的返回值表示跨所有描述字集的已准备好的总位数。如果在任何描述字准备好之前定时器时间到，则返回 0。返回 −1 表示有错（这是可能发生的，譬如函数被一个捕获的信号所中断）。

> SVR4 的早期版本中 select 的实现有一个缺陷：如果多个集合中的同一位均为 1，譬如说，一个描述字已准备好读和写，它只计数一次。现在的版本改正了此 bug。

描述字在什么条件下准备好？

我们已讨论了等待描述字准备好 I/O（读或写）或是有一个异常条件待处理（带外数据）。尽管可读性和可写性对于普通文件这样的描述字很明显，我们必须对引起 select 返回套接口“准备好”的条件说得明确些（TCPv2 的图 16.52）。

1. 下列四个条件中的任何一个满足时，套接口准备好读：
   a. 套接口接收缓冲区中的数据字节数大于等于套接口接收缓冲区低潮限度的当前值。对这样的套接口的读操作将不阻塞并返回一个大于 0 的值（即准备好读入的数据量）。我们可以用套接口选项 SO_RCVLOWAT 来设置此低潮限度，对于 TCP 和 UDP 套接口，其值缺省为 1。
   b. 连接的读这一半关闭（也就是说接收了 FIN 的 TCP 连接）。对这样的套接口的读操作将不阻塞且返回 0（即文件结束符）。
   c. 套接口是一个监听套接口且已完成的连接数为非 0。尽管在 15.6 节中我们将介绍 accept 可能阻塞的时序条件，但正常情况下，这样的监听套接口上的 accept 不会阻塞。
   d. 有一个套接口错误待处理。对这样的套接口的读操作将不阻塞且返回一个错误

(−1),errno 则设置成明确的错误条件。这些待处理的错误(pending errors)也可通过指定套接口选项 SO_ERROR 调用 getsockopt 来取得并清除。

2. 下列三个条件中的任一个满足时,套接口准备好写:

   a. 套接口发送缓冲区中的可用空间字节数大于等于套接口发送缓冲区低潮限度的当前值,且或者(i)套接口已连接,或者(ii)套接口不要求连接(例如 UDP 套接口)。这意味着,如果我们将这样的套接口设置为非阻塞(第 15 章),写操作将不阻塞且返回一个正值(例如由传输层接收的字节数)。我们可以用套接口选项 SO_SNDLOWAT 来设置此低潮限度,对于 TCP 和 UDP 套接口,其缺省值一般为 2048。

   b. 连接的写这一半关闭。对这样的套接口的写操作将产生信号 SIGPIPE(5.12 节)。

   c. 有一个套接口错误待处理。对这样的套接口的写操作将不阻塞且返回一个错误(−1),errno 则设置成明确的错误条件。这些待处理的错误也可通过指定套接口选项 SO_ERROR 调用 getsockopt 来取得并清除。

3. 如果一个套接口存在带外数据或者仍处于带外标记,那它有异常条件待处理。(我们在第 21 章描述带外数据。)

我们对"可读性"和"可写性"的定义直接取自 TCPv2 第 530～531 页中内核的 soreadable 和 sowriteable 宏,与此类似,我们对套接口"异常条件"的定义取自同一页中的函数 soo_select。

注意一个套接口出错时,它由 select 标记为既可读又可写。

接收和发送低潮限度的目的是:在 select 返回可读或可写条件之前,应用进程可以对有多少数据可读或有多大空间可用于写进行控制。例如,如果我们知道除非至少有 64 字节的数据可用,否则我们的应用进程不能完成有效的工作,那么就可以将接收低潮限度设置为 64,以防小于 64 个字节的数据准备好读时,select 就唤醒我们。

只要 UDP 套接口的发送低潮限度小于发送缓冲区大小(缺省关系常常如此),由于不需要连接,这样的 UDP 套接口总是可写的。

图 6.7 对刚描述的就函数 select 来说导致套接口准备好的条件作了小结。

条件	可读吗?	可写吗?	异常吗?
有数据可读	•		
关闭连接的读一半	•		
给监听套接口准备好新连接	•		
有可用于写的空间		•	
关闭连接的写一半		•	
待处理错误	•	•	
TCP 带外数据			•

图 6.7 就 select 来说导致套接口准备好的条件小结

select 的最大描述字数

早些时候我们说过,大多数应用程序不会用到许多描述字,很少能找到一个会使用几百个描述字的应用程序。但使用这么多描述字的应用程序确实存在,而且它们常常利用 select

来复用描述字。当 select 刚开始设计时，操作系统常对每个进程可用的最大描述字数上限作出限制(4.2BSD 的限制为 31)，select 也就用相同的限制值。但是，当今的 Unix 版本对每个进程的描述字数根本不作限制(仅受限于内存总量和管理性限制)，因此，我们的问题是：这对 select 有什么影响？

许多实现有类似于下面的声明，它取自 4.4BSD 的头文件<sys/types.h>：

```
/*
 * select uses bitmasks of file descriptors in longs.  These macros
 * manipulate such bit fields (the filesystem macros use chars).
 * FD_SETSIZE may be defined by the user, but the default here should
 * be enough for most uses.
 */
#ifndef    FD_SETSIZE
#DEFINE    FD_SETSIZE        256
#endif
```

这使我们想到，在包括此头文件之前，可将 FD_SETSIZE 定义为更大些的值以增加 select 所用的描述字集大小，遗憾的是，这常常没有什么作用①。

> 为了弄清到底出了什么差错，请注意 TCPv2 的图 16.53，它在内核中声明了三个描述字集，且将内核的 FD_SETSIZE 定义作为上限使用。因此增大描述字集大小的唯一方法是增大 FD_SETSIZE 的值，然后重新编译内核。不重新编译内核而改变其值是不够的。

有些厂家正在将 select 的实现改变为允许进程将 FD_SETSIZE 定义为比缺省值大些的值。BSD/OS 已改变了内核实现以允许更大的描述字集，还定义了四个新的 FD_xxx 宏来动态分配并操纵这些较大集合。然而，从可移植性角度出发，使用大描述字集应小心。

6.4　str_cli 函数(修订版)

现在，我们重写 5.5 节中的函数 str_cli，这一次我们使用 select，这样服务器进程一终止客户就能马上得到通知。早期版本的问题就在于当套接口上发生了某些事件时，客户可能阻塞于 fgets 调用，而新版本则阻塞于 select 调用，或是等待标准输入，或是等待套接口可读。图 6.8 示出了调用 select 所处理的各种条件。

有三个条件通过套接口处理：

1. 如果对方 TCP 发送数据，套接口就变为可读且 read 返回大于 0 的值(即数据的字节数)。
2. 如果对方 TCP 发送一个 FIN(对方进程终止)，套接口就变为可读且 read 返回 0(文件结束)。
3. 如果对方 TCP 发送一个 RST(对方主机崩溃并重新启动)，套接口就变为可读且

① 译者注：常值 FD_SETSIZE 的声明一直是在头文件<sys/types.h>中(4.4BSD 和 4.4BSD-Lite2)，但更新的源自 BSD 的内核和源自 SVR4 的内核把它放在头文件<sys/select.h>中。值得注意的是，有些应用程序(典型的是需要复用大量描述字的事件驱动服务器程序，所需描述字量超过 1024 个)开始改用 poll 代替 select，这样可避免描述字有限的问题。还要注意的是，select 的典型实现在描述字数增大时可能存在扩展性问题。

read 返回−1,errno 则含有明确的错误码。

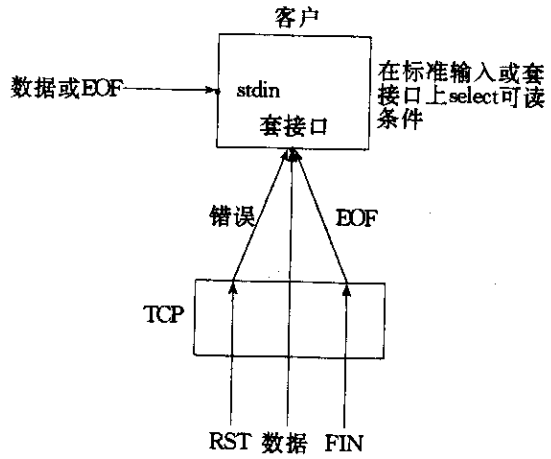


图 6.8 函数 str_cli 中由 select 处理的各种条件

图 6.9 给出了此新版本的源代码。

调用 select

第 8～13 行 我们仅需一个描述字集以检查可读性,此集合由 FD_ZERO 初始化,并用 FD_SET 打开两位:一位对应于标准 I/O 文件指针 fp,一位对应于套接口 sockfd。函数 fileno 把标准 I/O 文件指针转换为其对应的描述字。select(和 poll)仅工作在描述字上。

计算出两个描述字中的较大值后,调用 select。在调用中,写集合指针和异常集合指针都是空指针,且由于我们希望本调用阻塞,一直到某事件准备好,所以最后一个参数(时间限制)也是空指针。

处理可读套接口

第 14～18 行 如果在 select 返回时套接口是可读的,则回射行由 readline 来读,由 fputs 输出。

```
1 #include    "unp.h"
2 void
3 str_cli(FILE *fp,int sockfd)
4 {
5     int          maxfdp1;
6     fd_set       rset;
7     char         sendline[MAXLINE],recvline[MAXLINE];
8     FD_ZERO(&rset);
9     for( ; ; ) {
10        FD_SET(fileno(fp),&rset);
11        FD_SET(sockfd,&rset);
12        maxfdp1 = max(fileno(fp),sockfd) + 1;
13        Select(maxfdp1,&rset,NULL,NULL,NULL);
14        if (FD_ISSET(sockfd,&rset) ) { /* socket is readable */
15            if (Readline(sockfd,recvline,MAXLINE) == 0)
```

```
16                  err_quit("str_cli:server terminated prematurely");
17              Fputs(recvline,stdout);
18          }
19          if (FD_ISSET(fileno(fp),&rset)) {/* input is readable */}
20              if (Fgets(sendline,MAXLINE,fp) == NULL)
21                  return;          /* all done */
22          Writen(sockfd,sendline,strlen(sendline));
23          }
24      }
25 }
```

图 6.9　使用 select 的函数 str_cli 的实现(在图 6.13 中改进)[select/strcliselect01.c]

处理可读输入

第 19～23 行　如果标准输入可读，则由 fgets 读入一行，并用 writen 将其写到套接口。

请注意，此处使用了与 5.5 节相同的四个 I/O 函数：fgets、writen、readline 和 fputs，但程序流中使用函数的顺序发生了变化。它现在由 select 调用来驱动，而不是由 fgets 调用来驱动。与图 5.5 相比，图 6.9 中的代码仅增加了几行，就大大提高了客户程序的健壮性。

6.5　批 量 输 入

很不幸，我们的函数 str_cli 仍不正确。首先让我们回到其最初版本，即图 5.5。它以停-等方式工作，这对交互式使用是非常好的：发一行给服务器，然后便等待应答。这段总时间是 RTT(往返时间)加上服务器的处理时间(对于简单的回射服务器，处理时间几乎为 0)。如果知道了客户与服务器间的 RTT，我们便可以估计出回射固定数目的行数需花多长时间。

程序 Ping 是测量 RTT 的一个简单方法。如果我们在主机 solaris 上往主机 connix.com 执行 ping 命令，则对 30 次测量所取的平均 RTT 值是 175ms。TCPv1 第 89 页说明，这些 Ping 测量是针对长度为 84 字节的 IP 数据报进行的。如果我们取 solaris 2.5 上的文件 termcap 的前 2000 行，所得文件大小为 98349 字节，平均每行 49 字节。如果我们再加上 IP 头的大小(20 字节)和 TCP 头(20 字节)，则每行对应的分组大约有 89 字节，基本与 Ping 分组的大小相同。于是，我们可以估计出所有 2000 行文本的总时间大约为 350 秒(2000×0.175 秒)。如果运行第 5 章中的 TCP 回射客户程序，其真实时间大约为 354 秒，与我们的估计非常接近。

如果我们将客户与服务器间的网络当作全双工管道来考虑，从客户往服务器发请求，相反方向发应答，则图 6.10 所示为停-等方式。

客户在时刻 0 发出请求，我们假设 RTT 为 8 个时间单位。其应答时刻 4 发出并在时刻 7 接收到。我们还假设没有服务器处理时间且请求大小与回答大小相同。我们仅示出客户与服务器间的数据分组，对于同样穿越网络的 TCP 确认则不予理睬。

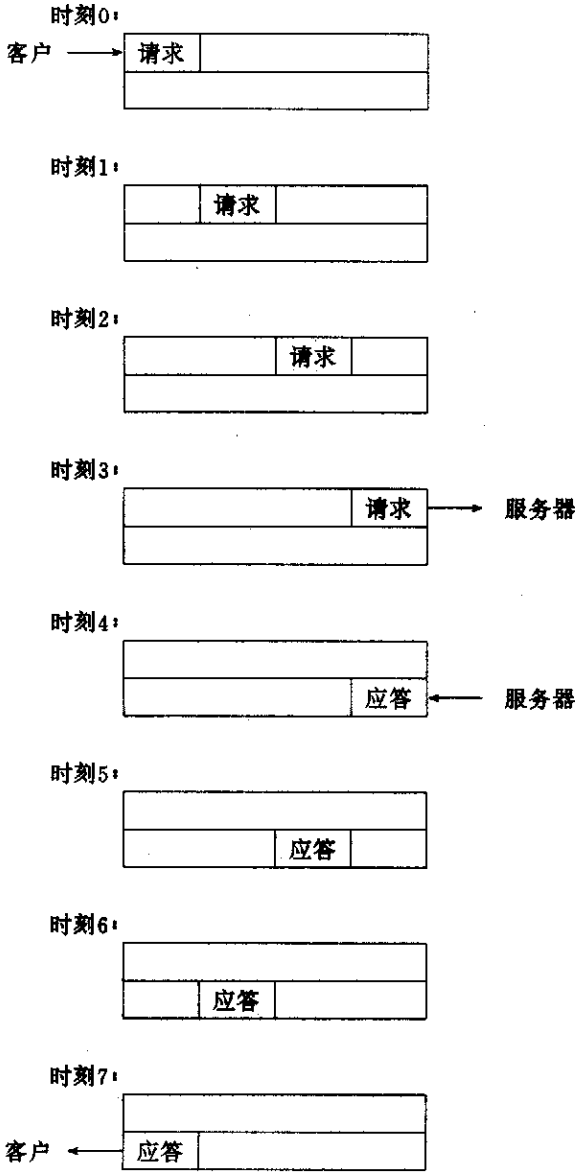


图 6.10 停-等方式的时间关系:交互式输入

但是,由于从发送一个分组到此分组到达管道的另一端有延迟,且由于管道是全双工的,在此例中,我们只用了⅛的管道容量。这种停-等方式对于交互式输入是非常合适的,但由于我们的客户从标准输入读且写到标准输出,在 Unix shell 下对输入和输出进行重定向又不麻烦,所以我们可以很容易地以批量方式运行客户。然而,当我们对输入输出重定向时结果输出文件总是小于输入文件(对于回射服务器,它们理应相等)。

为了弄清到底发生了什么事情,我们认识到在批量方式下,客户可以以网络能接受的最快速度发送请求,服务器也以相同的速度处理它们并发回应答。这在时刻 7 导致管道充满如图 6.11 所示。

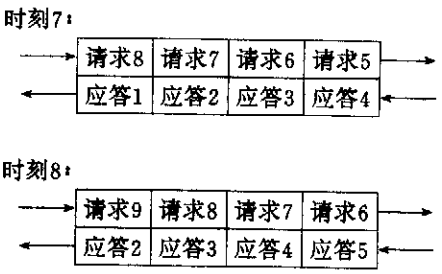


图 6.11　在客户与服务器间装填管道：批量方式

在这里，我们假设发出第一个请求后，立即发出下一个，紧接着再下一个。我们也假设客户可以以网络能接受它们的最快速度发送请求，且对它们的应答也能以网络能提供它们的最快速度处理。

> 这里，我们忽略了涉及到 TCP 批量数据的许多细节问题，如慢启动算法，它限制数据在一个新的或空闲的连接上发送数据和返回 ACK 的速率，这些都在 TCPv1 的第 20 章讨论。

为了弄清图 6.9 中函数 str_cli 存在的问题，假设输入文件只有 9 行，最后一行在时刻 8 发送，如图 6.11 所示。但我们在写完这个请求后，并不能关闭连接，因为管道中还有其他的请求和应答。问题的起因在于输入时对文件结束符的处理：str_cli 函数返回到 main 函数，main 函数随后终止。但在批量方式下，输入的文件结束符并不意味着我们已完成了从套接口的读入：可能仍有请求在去往服务器的路上，或是在去往客户的路上仍有应答。

我们需要一种方法来关闭 TCP 连接的一半。也就是说，我们想给服务器发一个 FIN，告诉它我们已完成了数据发送，但仍为读而开放套接口描述字。这由下一节描述的 shutdown 函数来完成。

6.6　shutdown 函数

终止网络连接的正常方法是调用 close，但 close 有两个限制可由函数 shutdown 来避免。

1. close 将描述字的访问计数减 1，仅在此计数为 0 时才关闭套接口，4.8 节我们已对此进行了讨论。用 shutdown 我们可以激发 TCP 的正常连接终止序列（图 2.5 中由 FIN 开始的四个分节），而不管访问计数。
2. close 终止了数据传送的两个方向：读和写。由于 TCP 连接是全双工的，有很多时候我们要通知另一端我们已完成了数据发送，即使那一端仍有许多数据要发送也是如此，这就是我们在前一节中所遇到的函数 str_cli 有批量输入时的情况。图 6.12 所示为该情况下的典型函数调用。

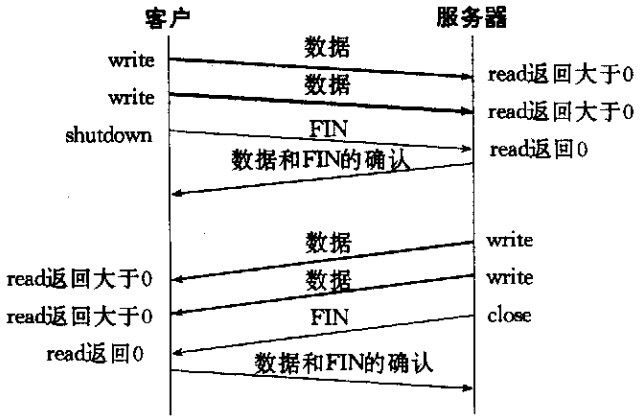


图 6.12 调用 shutdown 关闭一半 TCP 连接

```
#include <sys/socket.h>
int shutdown(int sockfd,int howto);
```

返回:0——成功,−1——出错

该函数的行为依赖于参数 howto 的值。

SHUT_RD 关闭连接的读这一半,不再接收套接口中的数据且现留在套接口接收缓冲区中的数据都作废。进程不能再对套接口执行任何读函数。调用此函数后,由 TCP 套接口接收的任何数据都被确认,但数据本身扔掉。

缺省时,写到路由套接口(第 17 章)上的所有内容都作为同一主机上所有路由套接口的输入而回馈。有些程序将第二个参数设为 SHUT_RD 来调用函数 shutdown 以防止回馈拷贝。另一个防止回馈拷贝的方法是关闭套接口选项 SO_USELOOPBACK。

SHUT_WR 关闭连接的写这一半,在 TCP 场合下,这称为半关闭(half-close)(TCPv1 的 18.5 节)。当前留在套接口发送缓冲区中的数据都被发送,后跟正常的 TCP 连接终止序列。正如我们前面所说,这个写这一半的关闭是不管套接口描述字的访问计数是否大于 0 的。进程不能再执行对套接口的任何写函数。

SHUT_RDWR 连接的读这一半和写这一半都关闭。这等效于调用函数 shutdown 两次:第一次调用时用 SHUT_RD,第二次调用时用 SHUT_WR。

图 7.10 对调用 shutdown 或 close 的进程的可能差别作了小结。close 的操作依赖于套接口选项 SO_LINGER 的值。

SHUT_xxx 这三个名字是随 Posix.1g 新近定义的。你碰到的 howto 参数的典型值可能是 0(关闭读这一半)、1(关闭写这一半)和 2(读这一半和写这一半都关闭)。

6.7　str_cli 函数(再修订版)

图 6.13 给出函数 str_cli 的改进(且正确)版本，它使用了 select 和 shutdown，前者在服务器关闭它这一端的连接时通知我们，后者使我们正确地处理批量输入。

第 5～8 行　stdineof 是一个初始化为 0 的新标志，只要此标志为 0，每次在主循环中我们总是 select 标准输入的可读性。

第 16～24 行　当我们在套接口上读到文件结束符时，如果我们已在标准输入上遇到文件结束符，这就是正常的终止，函数返回；但如果我们在标准输入上没有遇到文件结束符，那么服务器进程已过早终止。

第 25～33 行　当我们在标准输入上碰到文件结束符时，我们将新标志 stdineof 置为 1，并将第二个参数选为 SHUT_WR 来调用 shutdown 以发送 FIN。

如果用同样的 2000 行文件对图 6.13 中所示 str_cli 函数实现的 TCP 客户程序进行测量，现在所需时间约为 12.3 秒，比停-等方式大约快了 30 倍。

```
1  #include    "unp.h"
2  void
3  str_cli(FILE *fp, int sockfd)
4  {
5      int     maxfdp1, stdineof;
6      fd_set  rset;
7      char    sendline[MAXLINE], recvline[MAXLINE];
8      stdineof = 0;
9      FD_ZERO(&rset);
10     for ( ; ; ) {
11         if (stdineof == 0)
12             FD_SET(fileno(fp), &rset);
13         FD_SET(sockfd, &rset);
14         maxfdp1 = max(fileno(fp), sockfd) + 1;
15         Select(maxfdp1, &rset, NULL, NULL, NULL);
16         if (FD_ISSET(sockfd, &rset)) { /* socket is readable */
17             if (Readline(sockfd, recvline, MAXLINE) == 0) {
18                 if (stdineof == 1)
19                     return;         /* normal termination */
20                 else
21                     err_quit("str_cli: server terminated prematurely");
22             }
23             Fputs(recvline, stdout);
24         }
25         if (FD_ISSET(fileno(fp), &rset)) { /* input is readable */
26             if (Fgets(sendline, MAXLINE, fp) == NULL) {
27                 stdineof = 1;
28                 Shutdown(sockfd, SHUT_WR); /* send FIN */
29                 FD_CLR(fileno(fp), &rset);
30                 continue;
31             }
```

```
32              Writen(sockfd,sendline,strlen(sendline));
33          }
34      }
35  }
```

图 6.13 使用 select 正确处理文件结束符的函数 str_cli[select/strcliselect02.c]

我们对函数 str_cli 的讨论还没有结束，15.2 节中我们将开发一个使用非阻塞 I/O 模型的版本，23.3 节中我们将开发一个使用线程的版本。

6.8 TCP 回射服务器程序(修订版)

我们可以对 5.2 节和 5.3 节中介绍的 TCP 回射服务器重新进行讨论，并将服务器程序重写为使用 select 来处理任意数目的客户的单进程程序，而不是为每个客户派生一个子进程。在给出具体代码之前，让我们对用以跟踪客户的数据结构进行仔细的分析。图 6.14 给出了第一个客户建立连接前服务器的状态。



图 6.14 第一个客户建立连接前的服务器状态

服务器有单个监听描述字，我们用一个圆点来表示。

服务器只维护一个读描述字集，如图 6.15 所示。假设服务器是在前台启动的，则描述字 0、1 和 2 分别被设置为标准输入、标准输出和标准错误输出，因而，监听套接口的第一个可用的描述字是 3。我们也给出了一个名为 client 的整型数组，它含有每个客户的已连接套接口描述字，此数组中的所有元素都初始化为 −1。

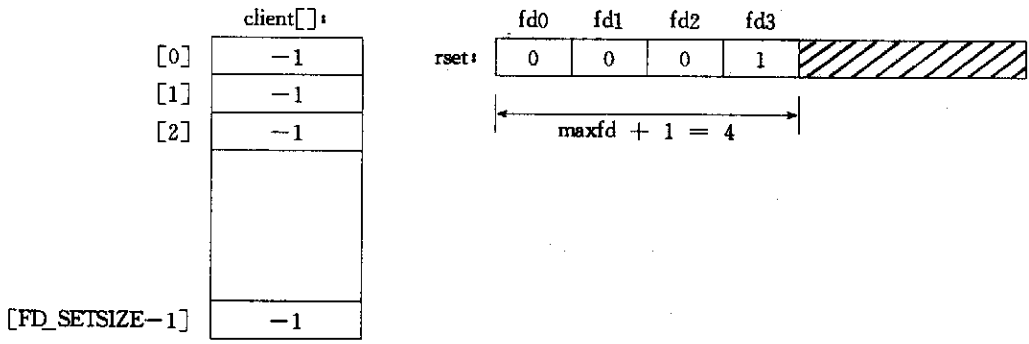


图 6.15 仅有监听套接口的 TCP 服务器数据结构

描述字集中唯一的非 0 条目是表示监听套接口的条目，因此 select 的第一个参数为 4。

当第一个客户与服务器建立连接时，监听描述字变为可读，于是我们的服务器调用 accept。在本例的假设下，由 accept 返回的新的已连接描述字将是 4。图 6.16 所示为从客户到

服务器的连接。

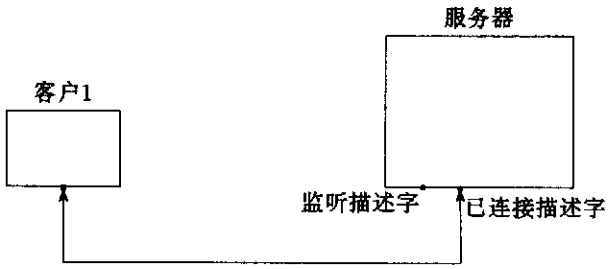


图 6.16　第一个客户建立连接后的 TCP 服务器

从现在起，我们的服务器必须在其数组 client 中记住新的已连接描述字，同时必须加到描述字集合中去，这样更新的数据结构示于图 6.17。

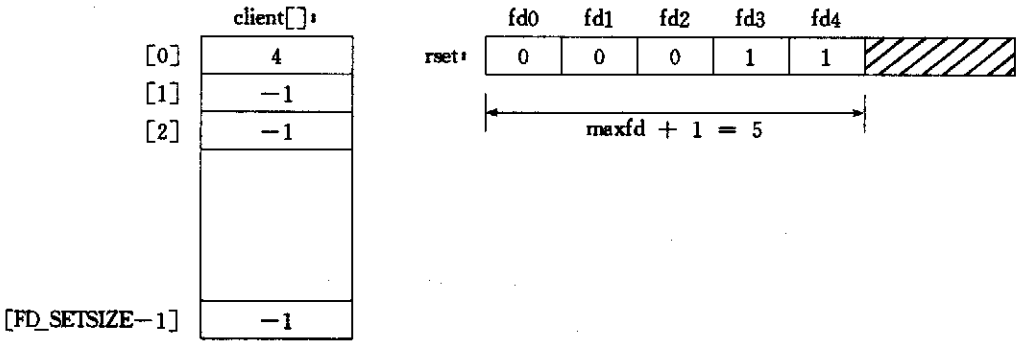


图 6.17　第一个客户连接建立后的数据结构

稍后，第二个客户与服务器建立连接，图 6.18 示出了这种情形。

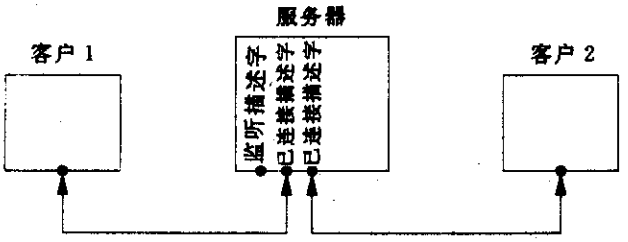


图 6.18　第二个客户建立连接后的 TCP 服务器

新的已连接描述字(我们假设是 5)必须被记住，数据结构如图 6.19 所示。

下面，我们假设第一个客户终止它的连接。客户 TCP 发送一个 FIN，这使得服务器中的描述字 4 变为可读。当服务器读此已连接套接口时，readline 返回 0。接着，我们关闭此套接口并相应地更新数据结构，数组元素 client[0]的值置为−1，描述字集中的描述字 4 被置为 0，如图 6.20 所示。注意，maxfd 的值没有改变。

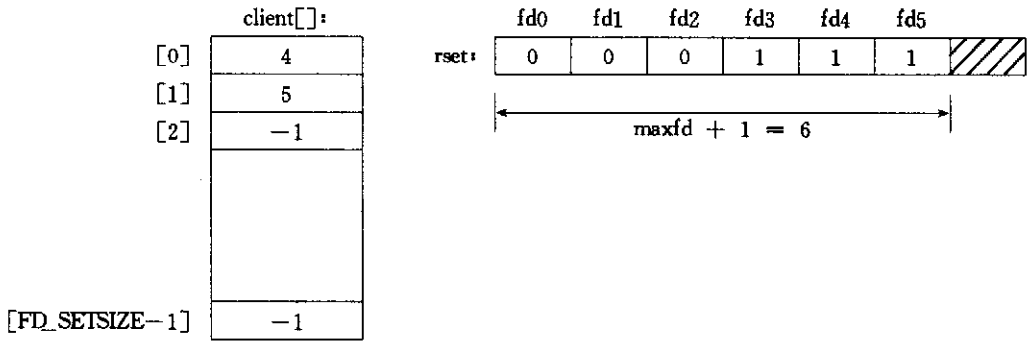


图 6.19 第二个客户建立连接后的数据结构

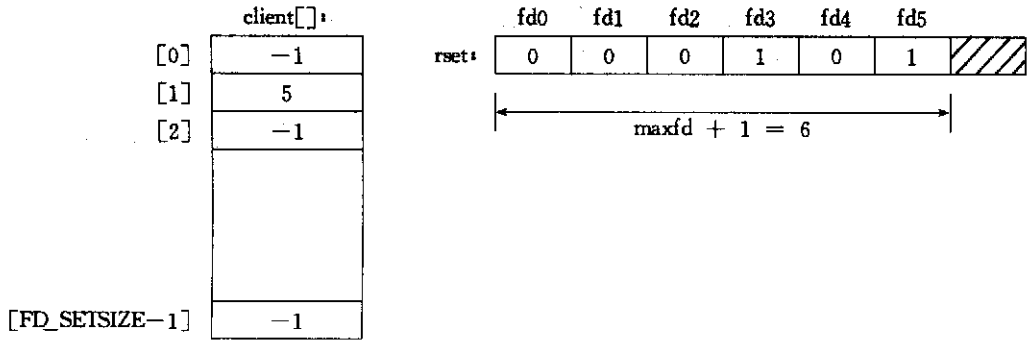


图 6.20 第一个客户终止连接后的数据结构

总之，当有客户到达时，我们在数组 client 中的第一个可用条目(即值为−1 的第一个条目)记录其已连接套接口的描述字。我们还必须把这个已连接描述字加到读描述字集合中。变量 maxi 是当前使用的数组 client 的最大下标，而变量 maxfd(加 1)是函数 select 第一个参数的当前值。对此服务器所能处理的最大客户数目的限制是以下两个值中的较小者：FD_SETSIZE 和内核允许此进程的最大描述字数(这我们在 6.3 节的结尾已讨论过)。

图 6.21 给出了此版本服务器程序的前半部分。

```
1 #include    "unp.h"
2 int
3 main(int argc,char * * argv)
4 {
5     int       i,maxi,maxfd,listenfd,connfd,sockfd;
6     int       nready,client[FD_SETSIZE];
7     ssize_t   n;
8     fd_set    rset,allset;
9     char      line[MAXLINE];
10    socklen_t clilen;
11    struct sockaddr_in cliaddr,servaddr;
12    listenfd= Socket(AF_INET,SOCK_STREAM,0);
13    bzero(&servaddr,sizeof(servaddr));
14    servaddr.sin_family = AF_INET;
15    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
```

```
16    servaddr.sin_port = htons(SERV_PORT);
17    Bind(listenfd,(SA *) &servaddr,sizeof(servaddr));
18    Listen(listenfd,LISTENQ);
19    maxfd = listenfd;            /* initialize */
20    maxi = -1;                   /* index into client[] array */
21    for (i = 0; i < FD_SETSIZE; i++)
22        client[i] = -1;          /* -1 indicates available entry */
23    FD_ZERO(&allset);
24    FD_SET(listenfd,&allset);
```

图 6.21　使用单进程和 select 的 TCP 服务器程序：初始化[tcpcliserv/tcpservselect01.c]

创建监听套接口并给 select 初始化

第 12～24 行　创建监听套接口的步骤与前面所讨论的相同：socket、bind 和 listen。我们按照一开始 select 的唯一描述字是监听描述字的前提对数据结构进行初始化。

main 函数的后半部分示于图 6.22 中。

```
25    for( ; ; ) {
26        rset = allset;           /* structure assignment */
27        nready = Select(maxfd + 1,&rset,NULL,NULL,NULL);
28        if (FD_ISSET(listenfd,&rset))     /* new client connection */
29            clilen = sizeof(cliaddr);
30            connfd = Accept(listenfd,(SA *) &cliaddr,&clilen);
31            for(i = 0; i < FD_SETSIZE;i++)
32                if(client[i] < 0) {
33                  client[i] = connfd;          /* save descriptor */
34                  break;
35                }
36            if(i == FD_SETSIZE)
37              err_quit("too many clients");
38            FD_SET(connfd, &allset);     /* add new descriptor to set */
39            if(connfd > maxfd)
40              maxfd = connfd;     /* for select */
41            if(i > maxi)
42              maxi=i;     /* max index in client[] array */
43            if(--nready <= 0)
44              continue;     /* no more readable descriptors */
45        }
46        for (i = 0; i <= maxi; i++)  {   /* check all clients for data */
47            if ( (sockfd = client[i]) < 0)
48              continue;
49            if(FD_ISSET(sockfd,&rset)) {
50              if( (n = Readline(sockfd,line,MAXLINE)) == 0) {
51                        /* connection closed by client */
52                  Close(sockfd);
53                  FD_CLR(sockfd,&allset);
54                  client[i] = -1;
55              } else
56                  Writen(sockfd,line,n);
```

```
57                  if(--nready <= 0)
58                      break;      /* no more readable descriptors */
59              }
60          }
61      }
62  }
```

图 6.22 使用单进程和 select 的 TCP 服务器程序：循环[tcpcliserv/tcpservselect01.c]

阻塞于 select

第 26～27 行 select 等待某个事件发生：或是新客户连接的建立，或是数据、FIN 或 RST 的到达。

接受新连接

第 28～45 行 如果监听套接口变为可读，那么已建立了一个新的连接。我们调用 accept 并相应更新数据结构，使用数组 client 中的第一个未用条目来记录已连接描述字。准备好描述字的数目减 1，若此值为 0，就避免进入下面的 for 循环。这就使得我们可以用 select 的返回值来避免对未准备好的描述字进行检查。

检查现有连接

第 46～60 行 对于每个现有的客户连接，我们测试其描述字是否在 select 返回的描述字集合中。如果是就从客户读入一行并回射给客户。如果客户关闭连接，那么 readline 返回 0，我们就要相应地更新数据结构。

我们从不对 maxi 的值减 1，但每次客户关闭其连接时可以检查这种可能性。

此服务器程序要比图 5.2 和图 5.3 中所示的服务器程序复杂，但它避免了为每个客户创建一个新进程的所有开销，是 select 一个很好的例子。不过，在 15.6 节中我们将讨论此服务器的一个问题，它可以通过将监听套接口定义为非阻塞型，然后检查并忽略 accept 返回的一些错误来很容易地解决。

拒绝服务型攻击

很不幸，我们刚给出的服务器程序有一个问题。考虑一下，如果一个恶意客户连接到服务器上，发送一个字节的数据（而不是一行数据）后就睡眠。服务器将调用 readline，它从客户上读到单个字节的数据，然后就阻塞于下一个 read 调用以等待此客户的其他数据。接着，服务器就阻塞于（“挂起”可能是一个更好的说法）此单个客户，不能为任何其他客户提供服务（不论是新的客户连接还是现有客户的数据）。这种状况一直要持续到此恶意客户发出一个换行符或是终止为止。

这里的一个基本概念就是当一个服务器正在处理多个客户时，服务器决不能阻塞于只与单个客户相关的函数调用。如果这样的话，服务器将悬挂并拒绝为所有其他客户提供服务，这称为拒绝服务（denial of service）型攻击。它对服务器作了某些动作后，服务器就不能为其他合法客户提供服务了。可能的解决办法是：(a)使用非阻塞 I/O 模型（第 15 章）；(b)让每个客户由单独的控制线程提供服务（例如，创建子进程或线程来为每个客户提供服务）；(c)对 I/O 操作设置超时（13.2 节）。

6.9　pselect 函数

函数 pselect 是由 Posix.1g 发明的。

```
#include <sys/select.h>
#include <signal.h>
#include <time.h>

int pselect (int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,
             const struct timespec *timeout, const sygset_t *sigmask);
                                    返回：准备好描述字的个数，0——超时，-1——出错
```

pselect 相对于正常的 select 有两个变化：

1. pselect 使用结构 timespec，这是 Posix.1b 实时标准的一个发明，而不使用结构 timeval。

```
struct timespec {
time_t tv_sec;      /* seconds */
long tv_nsec;       /* nanoseconds */
};
```

这两个结构的区别在第二个成员上：新结构的成员 tv_nsec 规定纳秒数，而老结构的成员 tv_usec 规定微秒数。

2. 函数 pselect 增加了第六个参数：指向信号掩码的指针。这允许程序禁止递交某些信号，测试由这些当前禁止的信号的信号处理程序所设置的全局变量，然后调用 pselect，告诉它临时重置信号掩码。

对于第二点，考虑下面的例子（在 APUE 第 389～309 页讨论）。这个程序的 SIGINT 信号处理程序仅设置全局变量 intr_flag 并返回。如果我们的进程阻塞于 select 调用，则从信号处理程序的返回会引起该函数返回错误并把 errno 设置成 EINTR。但当调用 select 时，代码看起来有以下形式：

```
if (intr_flag)
    handle_intr();      /* handle the signal */
if ( (nready = select( ... )) < 0) {
    if (errno == EINTR ) {
        if (intr_flag)
            handle_intr();
    }
    ...
}
```

问题就在于：在测试 intr_flag 和调用 select 之间如果信号发生，在 select 永远阻塞时它将丢失。有了 pselect 后，现在我们可以可靠地编写这个例子的代码如下：

```
sigset_t newmask, oldmask,zeromask;
```

```
sigemptyset(&zeromask);
sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);

sigprocmask(SIG_BLOCK, &newmask, &oldmask); /* block SIGINT */
if (intr_flag)
    handle_intr(); /* handle the signal */
if ( (nready = pselect( ... ,&zeromask)) < 0){
    if (errno == EINTR){
        if (intr_flag)
            handle_intr();
    }
    ...
}
```

在测试 intr_flag 变量之前，我们阻塞 SIGINT。当 pselect 被调用时，它用空集(即 zeromask)来代替进程的信号掩码，接着检查描述字，并可能进入睡眠。但当函数 pselect 返回时，进程的信号掩码又重置为调用 pselect 之前的值(即 SIGINT 阻塞)。

在 18.5 节，我们对 pselect 作更多的讨论，并给出了它的一个例子。在图 18.7 中，我们使用 pselect，然后在图 18.8 中，给出它的一个简单但不太正确的实现。

> 两个 select 函数还有另外一个小区别。结构 timeval 的第一个成员是有符号的长整数，而结构 timespec 的第一个成员是 time_t。前者的有符号长整数也应该是 time_t，但没做这种追溯性修改，以防破坏现有的代码。然而，新的函数 pselect 可以做此修改。

6.10 poll 函数

函数 poll 起源于 SVR3，开始时局限于流设备(第 32 章)，SVR4 取消了此限制，允许 poll 工作在任何描述字上。poll 提供了与 select 相似的功能，但当涉及到流设备时，它还提供附加信息。

```
#include <poll.h>
int poll(struct pollfd *fdarray, unsigned long nfds, int timeout);
                                返回：准备好描述字的个数，0——超时，-1——出错
```

第一个参数是指向一个结构数组第一个元素的指针，每个数组元素都是一个 pollfd 结构，它规定了为测试一给定描述字 fd 的一些条件。

```
struct pollfd {
int fd;          /* descriptor to check */
short events;    /* events of interest on fd */
short revents;   /* events that occurred on fd */
};
```

要测试的条件由成员 events 规定，函数在相应的 revents 成员中返回描述字的状态。(每个描述字有两个变量，一个为调用值，另一个为结果，以此避免使用值-结果参数。回想一下，

函数 select 的中间三个参数都是值-结果参数。)这两个成员中的每一个都由指定某个条件的一位或多位组成。图 6.23 列出了用于指定标志 events 并测试标志 revents 的一些常值。

常　量	能作为 events 的输入吗?	能作为 revents 的结果吗?	解释
POLLIN	•	•	普通或优先级带数据可读
POLLRDNORM	•	•	普通数据可读
POLLRDBAND	•	•	优先级带数据可读
POLLPRI	•	•	高优先级数据可读
POLLOUT	•	•	普通数据可写
POLLWRNORM	•	•	普通数据可写
POLLWRBAND	•	•	优先级带数据可写
POLLERR		•	发生错误
POLLHUP		•	发生挂起
POLLNVAL		•	描述字不是一个打开的文件

图 6.23　函数 poll 的输入 events 和返回值 revents

我们将此图分为三个部分：第一部分为处理输入的四个常值，第二部分为处理输出的三个常值，第三部分为处理错误的三个常值。注意，第三部分的三个常值在 events 中是不能设置的，但是当相应条件存在时就在 revents 中返回。

poll 识别三个类别的数据：普通(normal)、优先级带(priority band)和高优先级(high priority)，这些术语均出自基于流的实现(图 33.5)。

> POLLIN 可被定义为 POLLRDNORM 和 POLLRDBAND 的逻辑或常值。POLLIN 存在于 SVR3 的实现，它要早于 SVR4 中的优先级带，所以此常值保持了向后兼容性。类似地，POLLOUT 等效于 POLLWRNORM，前者早于后者。

考虑到 TCP 和 UDP 套接口，下面的条件引起 poll 返回特定的 revent。不幸的是，Posix.1g在其 poll 的定义中留下了许多漏洞(也就是说，返回相同的条件有多种方法可选择)。

- 所有正规 TCP 数据和 UDP 数据都被认为是普通数据。
- TCP 的带外数据(第 21 章)被认为是优先级带数据。
- 当 TCP 连接的读这一半关闭时(例如，接收了一个 FIN)，这也认为是普通数据，且后续的读操作将返回 0。
- TCP 连接存在错误既可认为是普通数据，也可认为是错误(POLLERR)。无论哪种情况，后续的读操作将返回－1，并将 errno 置为适当的值，这就处理了诸如接收到 RST 或超时等条件。
- 在监听套接口上新连接的可用性既可认为是普通数据，也可认为是优先级数据，大多数实现都将其作为普通数据考虑。

结构数组中元素的个数是由参数 nfds 来规定的。

历史上,这个参数曾被认为是无符号长整数,这似乎太大了,无符号整数可能更合适些。Unix 98 为此参数定义了一个新的数据类型:nfds_t。

参数 timeout 指定函数返回前等待多长时间。它是一个指定应等待的毫秒数的正值。图 6.24 给出了 timeout 的可能值。

timeout 的值	解释
INFTIM	永远等待
0	立即返回,不阻塞
>0	等待指定数目的毫秒数

图 6.24 poll 的 timeout 参数值

常值 INFTIM 定义为一个负值。如果系统不能提供毫秒级精度的定时器,此值向上舍入到最接近的支持值。

Posix.1g 要求在头文件<poll.h>中定义 INFTIM,但许多系统仍在头文件<sys/stropts.h>中定义。

正如 select,为 poll 所规定的任何超时值都受限于实现的时钟分辨率(常常是 10ms)。

当出错时,函数 poll 的返回值为-1;若定时器时间到,还没有描述字准备好,则返回 0;否则返回准备好描述字的个数,即成员 revents 非 0 的描述字个数。

如果我们不关心某个特定描述字,可将其 pollfd 结构的 fd 成员置为一个负值。这样就可忽略成员 events,且返回时将成员 revents 的值置为 0。

回忆一下 6.3 节的结尾,我们对 FD_SETSIZE、每个描述字集合中描述字最大数目及每个进程中描述字最大数目作了讨论。有了 poll 后,我们就没有那个问题了,因为分配一个结构 pollfd 的数组并通知内核数组中元素的数目是调用者的责任。这里没有类似于内核知道的 fd_set 之类固定大小的数据类型。

Posix.1g 对 select 和 poll 都要求,但从可移植性角度出发,当今支持 select 的系统比支持 poll 的系统要多。而且,Posix.1g 还定义了 pselect,它是函数 select 的增强版本,能处理信号阻塞并提供更高的时间分辨率,但未定义与函数 poll 相似的任何东西。

6.11 TCP 回射服务器程序(再修订版)

现在,我们用 poll 而不是 select 来重写 6.8 节中的 TCP 回射服务器程序。在以前使用 select 的版本中,我们必须分配一个 client 数组以及一个名为 rset 的描述字集(图 6.15)。使用 poll 时我们必须分配一个 pollfd 结构的数组来维护客户信息,而不是分配另一个数组。我们以与图 6.15 中处理数组 client 相同的方法处理此数组的 fd 成员:值-1 表示条目未用,否则即为描述字值。回忆一下前一节,传递给 poll 的 pollfd 结构数组中的任何 fd 成员为负值的条目都是被忽略的。

图 6.25 给出了我们的服务器程序的前半部分。

```
1 #include     "unp.h"
2 #include     <limits.h>          /* for OPEN_MAX */
3 int
4 main(int argc,char **argv)
5 {
6     int      i,maxi,listenfd,connfd,sockfd;
7     int      nready;
8     ssize_t  n;
9     char     line[MAXLINE];
10    socklen_t clilen;
11    struct pollfd client[OPEN_MAX];
12    struct sockaddr_in cliaddr,servaddr;
13    listenfd = Socket(AF_INET,SOCK_STREAM,0);
14    bzero(&servaddr,sizeof(servaddr));
15    servaddr.sin_family = AF_INET;
16    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
17    servaddr.sin_port = htons(SERV_PORT);
18    Bind(listenfd,(SA *) &servaddr,sizeof(servaddr));
19    Listen(listenfd,LISTENQ);
20    client[0].fd = listenfd;
21    client[0].events = POLLRDNORM;
22    for (i = 1;i < OPEN_MAX; i++)
23        client[i].fd = -1;          /* -1 indicates available entry */
24    maxi = 0;                       /* max index into client[] array */
```

图 6.25　使用函数 poll 的 TCP 服务器程序的前半部分[tcpcliserv/tcpservpoll01.c]

分配结构 pollfd 的数组

第 11 行　我们声明结构 pollfd 的数组中的 OPEN_MAX 个元素。确定进程任一时刻能打开的最大描述字数目是很困难的，图 12.4 中我们将再次遇到这个问题。一个方法是以参数_SC_OPEN_MAX(如 APUE 的第 42～44 页所述)调用 Posix.1 的函数 sysconf，然后动态分配一个合适大小的数组。但函数 sysconf 的一个可能返回是“indeterminate(不确定)”，这意味着我们仍然不得不猜一个值。这里我们就用 Posix.1 的常值 OPEN_MAX。

初始化

第 20～22 行　我们将数组 client 的第一个条目用于监听套接口，并将其余的描述字条目置为－1。我们还为此描述字设置了事件 POLLRDNORM，当新的连接准备好接受时由 poll 来通知。变量 maxi 含有当前使用的数组 client 的最大下标值。

main 函数的后半部分示于图 6.26 中。

调用 poll，检查新连接

第 26～42 行　我们调用 poll，或等待新连接，或等待现有连接上的数据。当一个新连接被接受时，我们在数组 client 中查找第一个为负的描述字来找到第一个可用条目。注意，由于 client[0]用于监听套接口了，我们从下标 1 开始搜索。当找到一个可用条目时，保存描述字并设置事件 POLLRDNORM。

在现有连接上检查数据

第43行～63行 我们检查的两个返回事件是POLLRDNORM和POLLERR。其中的第二项我们未在成员event中设置，因为在条件成立时它总是返回的。我们检查POLLERR的原因是：有些实现在连接上接收到RST时返回事件POLLERR，而其他实现只返回POLLRDNORM。无论哪种情况下我们调用readline，在出错时它都返回错误。当一个现有的连接由客户终止时，我们就设置其fd成员为－1。

```
25    for( ; ; ) {
26        nready = Poll(client,maxi + 1,INFTIM);
27        if(client[0].revents & POLLRDNORM) {      /* new client connection */
28            clilen = sizeof(cliaddr);
29            connfd = Accept(listenfd,(SA *) &cliaddr,&clilen);
30            for (i = 1; i < OPEN_MAX; i++)
31                if(client[i].fd < 0) {
32                    client[i].fd = connfd;          /* save descriptor */
33                    break;
34                }
35            if (i == OPEN_MAX)
36                err_quit("too many clients");
37            client[i].events = POLLRDNORM;
38            if (i > maxi)
39                maxi = i;          /* max index in client[] array */
40            if (--nready <= 0)
41                continue;          /* no more readable descriptors */
42        }
43        for(i = 1;i <= maxi; i++) {      /* check all clients for data */
44            if ( (sockfd = client[i].fd) < 0)
45                continue;
46            if(client[i].revents & (POLLRDNORM | POLLERR)) {
47                if( (n = readline(sockfd,line,MAXLINE)) < 0) {
48                    if(errno == ECONNRESET) {
49                            /* connection reset by client */
50                        Close(sockfd);
51                        client[i].fd = -1;
52                    } else
53                        err_sys("readline error");
54                } else if (n == 0) {
55                            /* connection closed by client */
56                    Close(sockfd);
57                    client[i].fd = -1;
58                } else
59                    Writen(sockfd,line,n);
60                if(--nready <= 0)
61                  break;          /* no more readable descriptors */
62            }
63        }
64    }
65 }
```

图6.26 使用poll的TCP服务器程序的后半部分[tcpcliserv/tcpservpoll01.c]

6.12　小　结

Unix 提供了五种不同的 I/O 模型：

- 阻塞 I/O 模型
- 非阻塞 I/O 模型
- I/O 复用模型
- 信号驱动 I/O 模型
- 异步 I/O 模型

缺省为阻塞 I/O 模型，这也是最常用的 I/O 模型。在后面章节中，我们将讨论非阻塞 I/O 模型和信号驱动 I/O 模型，而本章已讨论了 I/O 复用模型。真正的异步 I/O 模型是 Posix.1 定义的，但它很少有实现。

I/O 复用模型最常用的函数是 select。我们通知此函数所关心的描述字（读、写和异常条件）、最长等待时间、最大描述字号（加 1）。大多数对 select 的调用指定可读条件，并注意到，当处理套接口时，唯一的异常条件是带外数据的到达（第 21 章）。由于 select 提供了函数阻塞多长时间的限制，我们将此特征用于图 13.3 以设置输入操作的时间限制。

将我们的回射客户程序用到使用 select 的批处理模式中，我们发现，即使已遇到了用户输入的结尾，数据也可能仍在去往或来自服务器的管道中。为了处理这种情形，要求使用函数 shutdown，这使得我们可以利用 TCP 的半关闭特性。

Posix.1g 定义了新的函数 pselect，它将时间精度从微秒级增加到纳秒级，并采用了一个指向信号集的指针作为新参数，这样避免了信号被捕获时的竞争条件，我们在 18.5 节中再作进一步讨论。

系统 V 中的函数 poll 提供类似于 select 的功能，且为流设备提供附加信息。Posix.1g对 select 和 poll 都要求，但前者使用更频繁。

6.13　习　题

6.1　我们说，一个描述字集可以用 C 语言中的赋值语句赋给另一描述字集。如果描述字集是一个整型数组，怎么办？（提示：研究一下你系统中的头文件<sys/select.h>或<sys/types.h>。）

6.2　在讨论 6.3 节中 select 返回“可写”条件时，为什么必须要求套接口为非阻塞型以使写操作返回一个正值？

6.3　在图 6.9 中，在第 19 行中的关键词 if 前加上 else，将会怎样？

6.4　在图 6.21 的例子中，加上一段代码以允许服务器可以使用当前内核所允许的尽量多的描述字。（提示：研究一下函数 setrlimit。）

6.5　当 shutdown 的第二个参数为 SHUT_RD 时，让我们看看将发生什么。以图 5.4 中的 TCP 客户程序为基础并作下列改变：将端口号从 SERV_PORT 改为 19，即 chargen 服务器（图 2.13）；以调用 pause 来代替调用 str_cli。启动此程序并指定运

行服务器 chargen 的本地主机 IP 地址。以诸如 tcpdump(C.5 节)这类的工具来观察分组,看到什么?

6.6 为什么应用程序会以参数 SHUT_RDWR 来调用 shutdown,而不是仅仅调用 close?

6.7 图 6.22 中,当客户发送一个 RST 来终止连接时,将会怎样?

6.8 重写图 6.25 中的代码,调用 sysconf 来确定描述字的最大数目,并相应分配 client 数组。

第 7 章　套接口选项

7.1　概　述

有很多方法来获取和设置影响套接口的选项：

- 函数 getsockopt 和 setsockopt
- 函数 fcntl
- 函数 ioctl

本章从介绍函数 getsockopt 和 setsockopt 开始，接着给出一个输出所有选项缺省值的例子，然后详细介绍所有套接口选项。我们按以下分类进行详细介绍：基本、IPv4、IPv6 和 TCP。在第一次阅读本章时，这些细节可以跳过，当需要时再回来看个别章节。个别选项在后续章节中还有更为详细的讨论，如 IPv4 和 IPv6 多播选项在 19.5 节我们讨论多播时还会讨论到。

我们也介绍了函数 fcntl，因为它是设置套接口为非阻塞 I/O 型或信号驱动 I/O 型以及设置套接口的属主的 Posix 的方法。我们将函数 ioctl 留到第 16 章讨论。

7.2　getsockopt 和 setsockopt 函数

这两个函数仅用于套接口。

```
#include <sys/socket.h>
int getsockopt(int sockfd,int level,int optname, void *optval, socklen_t *optlen);
int setsockopt(int sockfd,int level,int optname, const void *optval,socklen_t *optlen);
                                                    返回：0——OK，-1——出错
```

sockfd 必须指向一个打开的套接口描述字，Level（级别）指定系统中解释选项的代码：普通套接口代码或特定于协议的代码（例如：IPv4、IPv6 或 TCP）。

optval 是一个指向变量的指针，通过它，或由 setsockopt 取得选项的新值，或由 getsockopt 存储选项的当前值。此变量的大小由最后一个参数指定，它对于 setsockopt 是一个值，对于 getsockopt 是一个值-结果参数。

图 7.1 总结了可由 getsockopt 获取或由 setsockopt 设置的一些选项。“数据类型”列给出了指针 optval 必须指向的每个选项的数据类型。我们用后跟一对花括号的记法来表示一个结构，如 linger{}表示结构 linger。

有两种基本类型的套接口选项:打开或关闭某个特性的二进制选项(标志),取得并返回我们可以设置或检查的特定值的选项(值)。标有"标志"的列指明选项是否为标志选项。当给这些标志选项调用函数 getsockopt 时,optval 是一个整数。optval 中返回的值是 0 表示选项关闭,非 0 表示选项打开。类似地,函数 setsockopt 要求一非 0 的 optval 来打开选项,要求用 0 来关闭选项。如果"标志"列不含有"·",则选项用来在用户进程与系统间传递指定数据类型的值。

本章后续各节将给出影响套接口的选项的额外细节。

level(级别)	optname(选项名)	get	set	说 明	标志	数据类型
SOL-SOCKET	SO-BROADCAST	·	·	允许发送广播数据报	·	int
	SO-DEBUG	·	·	使能调试跟踪	·	int
	SO-DONTROUTE	·	·	旁路路由表查询	·	int
	SO-ERROR	·		获取待处理错误并消除		int
	SO-KEEPALIVE	·	·	周期地测试连接是否仍存活	·	int
	SO-LINGER	·	·	若有数据待发送则延迟关闭		linger{}
	SO-OOBINLINE	·	·	让接收到的带外数据继续在线存放	·	int
	SO-RCVBUF	·	·	接收缓冲区大小		int
	SO-SNDBUF	·	·	发送缓冲区大小		int
	SO-RCVLOWAT	·	·	接收缓冲区低潮限度		int
	SO-SNDLOWAT	·	·	发送缓冲区低潮限度		int
	SO-RCVTIMEO	·	·	接收超时		timeval{}
	SO-SNDTIMEO	·	·	发送超时		timeval{}
	SO-REUSEADDR	·	·	允许重用本地地址	·	int
	SO-REUSEPORT	·	·	允许重用本地地址	·	int
	SO-TYPE	·		取得套接口类型		int
	SO-USELOOPBACK	·	·	路由套接口取得所发送数据的拷贝	·	int
IPPROTO-IP	IP-HDRINCL	·	·	IP 头部包括数据	·	int
	IP-OPTIONS	·	·	IP 头部选项		(见正文)
	IP-RECVDSTADDR	·	·	返回目的 IP 地址	·	int
	IP-RECVIF	·	·	返回接收到的接口索引	·	int
	IP-TOS	·	·	服务类型和优先权		int
	IP-TTL	·	·	存活时间		int
	IP-MULTICAST-IF	·	·	指定外出接口		in-addr{}
	IP-MULTICAST-TTL	·	·	指定外出 TTL		u-char
	IP-MULTICAST-LOOP	·	·	指定是否回馈		u-char
	IP-ADD-MEMBERSHIP		·	加入多播组		ip-mreq{}
	IP- DROP- MEMBERSHIP		·	离开多播组		ip-mreq{}
IPPROTO- ICMPV6	ICMP6-FILTER	·	·	指定传递的 ICMPv6 消息类型		icmp6- filter{}
IPPROTO-IPV6	IPV6-ADDRFORM	·	·	改变套接口的地址结构		int
	IPV6-CHECKSUM	·	·	原始套接口的校验和字段偏移		int
	IPV6-DSTOPTS	·	·	接收目标选项	·	int
	IPV6-HOPLIMIT	·	·	接收单播跳限	·	int
	IPV6-HOPOPTS	·	·	接收步跳选项	·	int
	IPV6-NEXTHOP	·	·	指定下一跳地址	·	sockaddr{}
	IPV6-PKTINFO	·	·	接收分组信息	·	int
	IPV6-PKTOPTIONS	·	·	指定分组选项		(见正文)
	IPV6-RTHDR	·	·	接收源路径	·	int
	IPV6-UNICAST-HOPS	·	·	缺省单播跳限		int

（续）

级　别	optname	get	set	解释	标志	数据类型
	IPV6_MULTICAST_IF	•	•	指定外出接口		in6_addr{}
	IPV6_MULTICAST_HOPS	•	•	指定外出跳限		u_int
	IPV6_MULTICAST_LOOP	•	•	指定是否回馈	•	u_int
	IPV6_ADD_MEMBERSHIP		•	加入多播组		ipv6_mreq{}
	IPV6_DROP_MEMBERSHIP		•	离开多播组		ipv6_mreq{}
IPPROTO_TCP	TCP_KEEPALIVE	•	•	控测对方是否存活前连接闲置秒数		int
	TCP_MAXRT	•	•	TCP 最大重传时间		int
	TCP_MAXSEG	•	•	TCP 最大分节大小		int
	TCP_NODELAY	•	•	禁止 Nagle 算法	•	int
	TCP_STDURG	•	•	紧急指针的解释	•	int

图 7.1　getsockopt 和 setsockopt 访问的套接口选项汇总

7.3　检查选项是否受支持并获取缺省值

现在，我们写一个程序来检查是否支持图 7.1 中定义的大多数选项，若支持，则输出其缺省值。图 7.2 包含了我们这个程序的所有声明。

```
1 #include      "unp.h"
2 #include      <netinet/tcp.h>       /* for TCP_xxx defines */
3 union val {
4   int               i_val;
5   long              l_val;
6   char              c_val[10];
7   struct linger     linger_val;
8   struct timeval    timeval_val;
9 } val;
10 static char *sock_str_flag(union val *, int);
11 static char *sock_str_int(union val *, int);
12 static char *sock_str_linger(union val *, int);
13 static char *sock_str_timeval(union val *, int);
14 struct sock_opts {
15     char        *opt_str;
16     int         opt_level;
17     int         opt_name;
18     char     *(*opt_val_str)(union val *, int);
19 } sock_opts[] = {
20     "SO_BROADCAST",    SOL_SOCKET,   SO_BROADCAST,   sock_str_flag,
21     "SO_DEBUG",        SOL_SOCKET,   SO_DEBUG,       sock_str_flag,
22     "SO_DONTROUTE",    SOL_SOCKET,   SO_DONTROUTE,   sock_str_flag,
23     "SO_ERROR",        SOL_SOCKET,   SO_ERROR,       sock_str_int,
24     "SO_KEEPALIVE",    SOL_SOCKET,   SO_KEEPALIVE,   sock_str_flag,
25     "SO_LINGER",       SOL_SOCKET,   SO_LINGER,      sock_str_linger,
```

```
26    "SO_OOBINLINE",    SOL_SOCKET,  SO_OOBINLINE,    sock_str_flag,
27    "SO_RCVBUF",       SOL_SOCKET,  SO_RCVBUF,       sock_str_int,
28    "SO_SNDBUF",       SOL_SOCKET,  SO_SNDBUF,       sock_str_int,
29    "SO_RCVLOWAT",     SOL_SOCKET,  SO_RCVLOWAT,     sock_str_int,
30    "SO_SNDLOWAT",     SOL_SOCKET,  SO_SNDLOWAT,     sock_str_int,
31    "SO_RCVTIMEO",     SOL_SOCKET,  SO_RCVTIMEO,     sock_str_timeval,
32    "SO_SNDTIMEO",     SOL_SOCKET,  SO_SNDTIMEO,     sock_str_timeval,
33    "SO_REUSEADDR",    SOL_SOCKET,  SO_REUSEADDR,    sock_str_flag,
34 #ifdef    SO_REUSEPORT
35    "SO_REUSEPORT",    SOL_SOCKET,  SO_REUSEPORT,    sock_str_flag,
36 #else
37    "SO_REUSEPORT",    0,           0,               NULL
38 #endif
39    "SO_TYPE",         SOL_SOCKET,  SO_TYPE,         sock_str_int,
40    "SO_USELOOPBACK",  SOL_SOCKET,  SO_USELOOPBACK,  sock_str_flag,
41    "IP_TOS",          IPPROTO_IP,  IP_TOS,          sock_str_int,
42    "IP_TTL",          IPPROTO_IP,  IP_TTL,          sock_str_int,
43    "TCP_MAXSEG",      IPPROTO_TCP, TCP_MAXSEG,,     sock_str_int,
44    "TCP_NODELAY",     IPPROTO_TCP, TCP_NODELAY,,    sock_str_flag,
45    NULL,              0,           0,               NULL
46 };
```

图 7.2 检查套接口选项的程序的声明[sockopt/checkopts.c]

声明可能值的联合

第3～9行 对于getsockopt的每个可能的返回值,我们的联合中都有一个成员。

定义函数原型

第10～13行 我们为四个函数定义函数原型,这四个函数被调用来输出给定套接口选项的值。

定义结构并初始化数组

第14～16行 我们的结构sock_opts包含了给每个套接口选项调用getsockopt并输出其当前值所需要的所有信息。它的最后一个成员:opt_val_str,是指向用于输出选项值的四个函数中某一个的指针。我们分配并初始化这个结构的一个数组,它的每个元素代表一个套接口选项。

并非所有实现都支持所有的套接口选项。确定某给定选项是否被支持的方法是用语句#ifdef或if defined,如图中SO_REUSEPORT选项所示。要求完整的话数组中每个元素都应类似于SO_REUSEPORT所示编写,但我们省略了这些,因为一大堆# ifdef语句仅仅加长了代码,对我们的讨论没有什么用处。

图7.3给出了我们的main函数。

```
47 int
48 main(int argc,char * * argv)
49 {
50    int     fd,len;
51    struct sock_opts *ptr;
52    fd = Socket(AF_INET,SOCK_STREAM,0);
```

```
53    for(ptr = sock_opts;ptr->opt_str ! = NULL; ptr++) {
54        printf("%s:",ptr->opt_str);
55        if (ptr->opt_val_str == NULL)
56            printf("(undefined)\n");
57        else {
58            len = sizeof(val);
59            if(getsockopt(fd,ptr->opt_level,ptr->opt_name,
60                    &val,&len) == -1) {
61                err_ret("getsockopt error");
62            } else {
63                printf("default = %s\n",(*ptr->opt_val_str)(&val,len));
64            }
65        }
66    }
67    exit(0);
68 }
```

图 7.3　检查所有套接口选项的 main 函数[sockopt/checkopts.c]

创建 TCP 套接口，浏览所有选项

第 52～56 行　我们创建一个 TCP 套接口，然后浏览数组中所有元素。如果指针 opt_val_str 为空，则该实现没有定义此选项(我们的例子中 SO_REUSEPORT 选项是可能的)。

调用 getsockopt

第 57～61 行　我们调用 getsockopt，但在返回错误时并不终止。许多实现定义了一些套接口选项名字，即使它们不支持这些选项也是如此。不支持的选项应引发一个 ENOPROTOOPT 错误。

输出选项的缺省值

第 62～63 行　如果 getsockopt 返回成功，我们调用相应的函数将选项值转换为一个字符串，然后输出这个字符串。

在图 7.2 中，我们给出了四个函数原型，每个类型的选项值一个。图 7.4 给出了这四个函数中的一个，即 sock_str_flag，它输出标志选项的值，其他三个函数与此类似。

```
69 static char strres[128];
70 static char *
71 sock_str_flag(union val *ptr,int len)
72 {
73    if(len ! = sizeof(int))
74        snprintf(strres,sizeof(strres),"size (%d) not sizeof(int)",len);
75    else
76        snprintf(strres,sizeof(strres),
77                "%s",(ptr->i_val == 0) ? "off" : "on");
78    return(strres);
79 }
```

图 7.4　函数 sock_str_flag：将标志选项转换为字符串[sockopt/checkopts.c]

第73～78行 回忆一下，getsockopt的最后一个参数是值-结果参数。我们所做的第一项检查就是getsockopt返回值的大小是否为期望的大小。本函数返回的字符串或为“on”，或为“off”，这得取决于标志选项的值是0还是非0。

在AIX 4.2下运行此程序有以下输入：

```
aix % checkopts
SO_BROADCAST: default = off
SO_DEBUG: default = off
SO_DONTROUTE: default = off
SO_ERROR: default = 0
SO_KEEPALIVE: default = off
SO_LINGER: default = 1 onoff = 0,1_linger = 0
SO_OOBINLINE: default = off
SO_RCVBUF: default = 16384
SO_SNDBUF: default = 16384
SO_RCVLOWAT: default = 1
SO_SNDLOWAT: default = 4096
SO_RCVTIMEO: default = 0 sec, 0 usec
SO_SNDTIMEO: default = 0 sec, 0 usec
SO_REUSEADDR: default = off
SO_REUSEPORT: (undefined)
SO_TYPE: default = 1
SO_USELOOPBACK:default = off
IP_TOS: default = 0
IP_TTL: default = 60
TCP_MAXSEG: default = 512
TCP_NODELAY: default = off
```

选项SO_TYPE的返回值1对应于此实现的SOCK_STREAM。

7.4 套接口状态

对于某些套接口选项，针对套接口的状态，什么时候进行设置或获取选项有时序上的考虑。我们对受影响的选项论及这一点。

下面的套接口选项是由TCP已连接套接口从监听套接口继承来的(TCPv2第462～463页)：SO_DEBUG、SO_DONTROUTE、SO_KEEPALIVE、SO_LINGER、SO_OOBINLINE、SO_RCVBUF和SO_SNDBUF。这对TCP是很重要的，因为accept一直要到TCP层完成三路握手后才会给服务器返回已连接套接口。如果想在三路握手完成时确保这些套接口选项中的某一个是给已连接套接口设置的，我们必须先给监听套接口设置此选项。

7.5 基本套接口选项

我们从基本套接口选项开始讨论。这些选项是协议无关的(也就是说，它们由内核中的协议无关代码处理，而不是由诸如IPv4这样的一类特殊的协议模块处理)，但有些选项仅能应用到某些确定类型的套接口中。例如，尽管SO_BROADCAST套接口选项称为“基本选

项",它也仅应用于数据报套接口。

SO_BROADCAST 套接口选项

此选项使能或禁止进程发送广播消息的能力。只有数据报套接口支持广播,并且还必须是在支持广播消息的网络上(例如以太网、令牌环网等)。你不可能在一个点对点链路上进行广播。第 18 章我们将更为详细地讨论广播。

由于一个应用进程在发送一个广播数据报之前必须设置此套接口选项,因此它能有效防止该进程在应用程序未设计成能广播时就发送广播消息。例如,一个 UDP 应用程序可能将目的 IP 地址作为命令行参数,但它并不期望用户键入一个广播地址。处理方法并非让应用进程来确定一给定地址是否为广播地址,而是把测试放在内核中进行:如果目的地址是一个广播地址且此套接口选项没有设置,则返回 EACCES(TCPv2 第 233 页)。

SO_DEBUG 套接口选项

此选项仅由 TCP 支持。当给一个 TCP 套接口打开此选项时,内核对 TCP 在此套接口所发送和接收的所有分组跟踪详细信息。这些信息保存在内核的环形缓冲区中,可由程序 trpt 来进行检查。TCPv2 第 916～920 页提供了更为详细的信息和使用了此选项的一个例子。

SO_DONTROUTE 套接口选项

此选项规定发出的分组将旁路底层协议的正常路由机制。例如,对于 IPv4,分组被指向适当的本地接口,也就是目的地址的网络和子网部分所确定的本地接口。如果本地接口不能由目的地址确定(例如,目的主机不在一个点对点链路的另一端上,也不在一个共享网络上),则返回 ENETUNREACH 错误。

给函数 send、sendto 或 sendmsg 使用标志 MSG_DONTROUTE 也能在个别的数据报上取得同样效果。

此选项经常由路由守护进程(routed 和 gated)用来旁路路由表(路由表不正确的情况下),强制一个分组从某个特定接口发出。

SO_ERROR 套接口选项

当套接口上发生错误时,源自 Berkeley 的内核中的协议模块将此套接口的名为 so_error 的变量设为标准的 Unix Exxx 值中的一个,它称为此套接口的待处理错误(pending error)。内核可立即以下面两种方式通知进程:

1. 如果进程阻塞于对此套接口的 select 调用(6.3 节),则无论是检查可读条件还是可写条件,select 均返回并设置其中一个或所有两个条件。
2. 如果进程使用信号驱动 I/O 模型(第 22 章),则给进程或进程组生成信号 SIGIO。

进程然后可以通过获取 SO_ERROR 套接口选项来得到 so_error 的值。由 getsockopt 返回的整数值就是此套接口的待处理错误。so_error 随后由内核复位为 0(TCPv2 第 547 页)。

当进程调用 read 且没有数据返回时,如果 so_error 为非 0 值,则 read 返回 −1 且 errno 设为 so_error 的值(TCPv2 第 516 页),接着 so-error 的值被复位为 0。如果此套接口上有数

据在排队，则 read 返回那些数据而不是返回错误条件。如果在进程调用 write 时 so_error 为非 0 值，则 write 返回 −1 且 errno 设为 so_error 的值(TCPv2 第 495 页)，so_error 也被复位为 0。

> TCPv2 第 495 页所示代码中有一个缺陷，那儿 so_error 没有被复位为 0，这在 BSD/OS 的版本中已经修改了。任何时候某个套接口有待处理错误返回时，它都必须被复位为 0。

这是我们遇到的可以获取但不能设置的第一个套接口选项。

SO_KEEPALIVE 套接口选项

给一个 TCP 套接口设置保持存活(keepalive)选项后，如果 2 小时内在此套接口的任一方向都没有数据交换，TCP 就自动给对方发一个保持存活探测分节(keepalive probe)。这是一个对方必须响应的 TCP 分节，它会导致以下三种情况：

1. 对方以期望的 ACK 响应。应用进程得不到通知(因为一切正常)。又过仍无动静的 2 小时后，TCP 将发出另一个探测分节。
2. 对方以 RST 响应，它告诉本地 TCP，对方已崩溃且已重新启动。套接口的待处理错误被置为 ECONNRESET，套接口本身则被关闭。
3. 对方对保持存活探测分节无任何响应。源自 Berkeley 的 TCP 发送另外 8 个探测分节，相隔 75 秒一个，试图得到一个响应。TCP 在发出第一个探测分节 11 分钟 15 秒后若仍无响应就放弃。如果对 TCP 的所有探测分节根本就没有响应，套接口的待处理错误被置为 ETIMEOUT，套接口本身则被关闭。但如果套接口接收到一个 ICMP 错误作为某个探测分节的响应，则返回相应的错误(图 A.15 和图 A.16)，套接口也被关闭。此情形中一个常见的 ICMP 错误是“host unreachable(主机不可达)”，说明对方主机并没有崩溃，但是不可达，这种情况下待处理错误被置为 EHOSTUNREACH。

TCPv1 第 23 章和 TCPv2 第 828～831 页均有对保持存活选项的详细叙述。

对于此选项的一个最常见的问题就是时间参数是否可改(常常是想将 2 小时的无活动周期改为短些的值)。我们在 7.9 节描述新的 Posix.1g 选项 TCP_KEEPALIVE，但它没有广泛实现。TCPv1 的附录 E 讨论了如何给各种内核修改这些定时参数，但必须注意，大多数内核是以整个内核为基维护这些时间参数的，而不是以每个套接口为基来维护的，因此若将无活动周期从 2 小时改为(譬如说)15 分钟，则将影响到主机上所有打开了此选项的套接口。

这个选项的目的是检测对方主机是否崩溃。如果对方进程崩溃，它的 TCP 将跨连接发送一个 FIN，这可以通过调用 select 很容易地检测到。(这就是为什么我们在 6.4 节中使用 select 的原因。)同时也要认识到，即使对任何保持存活探测分节均无响应(第三种情况)，我们也不能肯定对方主机已崩溃，因而 TCP 可能会终止一个有效连接。某个中间路由器崩溃 15 分钟是有可能的，而这段时间正好与主机的 11 分钟又 15 秒的保持存活探测周期完全重迭。

此选项一般由服务器使用，尽管客户也可用此选项。服务器用此选项是因为它们花大部

分时间阻塞于等待跨 TCP 连接的输入上，也就是说，等待客户请求。但如果客户主机崩溃，服务器进程将永远不会知道，并将继续等待永远不会到达的输入，这称为半开连接(half-open connection)。保持存活选项将检测出这些半开连接并终止它们。

> 大多数 Rlogin 和 Telnet 服务器设置此选项，以在交互式客户未注销就挂断电话线或关掉终端电源的情况下也能终止连接。
>
> 有些服务器，常常是 FTP 服务器，提供一个分钟数量级的超时。这是由应用进程本身完成的，一般在读下一个客户命令的 read 调用附近。这个超时与此套接口选项无关。

图 7.5 对一个 TCP 连接的另一端发生某些事件时我们必须检测的各种方法作了总结。当我们说“使用 select 判断可读条件”时，意味着要调用 select 来检测套接口是否可读。

情　形	对方进程崩溃	对方主机崩溃	对方主机不可达
本地 TCP 正主动发送数据	对方 TCP 发送一个 FIN，我们通过使用 select 判断可读条件立即能检测出来。如果本地 TCP 发送另外一个分节，对方 TCP 就以 RST 响应。如果再发送另外一个分节，本地 TCP 就给我们发一个 SIGPIPE 信号	本地 TCP 将超时，且套接口的待处理错误被设置为 ETIMEDOUT	本地 TCP 将超时，且套接口的待处理错误被设置为 EHOSTUNREACH
本地 TCP 正主动接收数据	对方 TCP 将发送一个 FIN，我们将把它作为一个(可能是过早的)文件结束符读入	我们将停止接收数据	我们将停止接收数据
连接空闲，保持存活选项已设	对方 TCP 发送一个 FIN，我们通过使用 select 判断可读条件立即能检测出来	在毫无动静 2 小时后，发送 9 个保持存活探测分节，然后套接口的待处理错误被设置为 ETIMEDOUT	在毫无动静 2 小时后，发送 9 个保持存活探测分节，然后套接口的待处理错误被设置为 EHOSTUNREACH
连接空闲，保持存活选项未设	对方 TCP 发送一个 FIN，我们通过使用 select 判断可读条件立即能检测出来	(无)	(无)

图 7.5　检测各种 TCP 条件的方法

SO_LINGER 套接口选项

此选项指定函数 close 对面向连接的协议如何操作(例如对 TCP 而不是对 UDP)。缺省操作是 close 立即返回，但如果有数据残留在套接口发送缓冲区中，系统将试着将这些数据发送给对方。

SO_LINGER 套接口选项使我们可以改变这个缺省设置。此选项要求在用户进程与内核间传递如下结构，它在头文件<sys/socket.h>中定义：

```
struct linger {
int l_onoff;    /* 0=off, nonzero=on */
int l_linger;   /* linger time, Posix.1g specifies units as seconds */
};
```

对 setsockopt 的调用将依两个结构成员的值导致下列三种情况的某一种：

1. 如果 l_onoff 为 0，则选项关闭，l_linger 的值被忽略且前面讨论的 TCP 缺省设置生效：close 立即返回。
2. 如果 l_onoff 为非 0 值且 l_linger 为 0，那么当套接口关闭时 TCP 夭折连接(TCPv2 第 1019～1020 页)。TCP 将丢弃保留在套接口发送缓冲区中的任何数据并发送一个 RST 给对方，而不是通常的四分组连接终止序列(2.5 节)。我们在图 15.21 中给出了这样的一个例子。这避免了 TCP 的 TIME_WAIT 状态，但这样做也会有以下可能性：在 2MSL 秒内创建此连接的另一个化身，并使得老的来自刚被终止的连接上的重复分节不正确地递送到新的化身上(2.6 节)。

   有些实现，尤其是 solaris 2.x(此处 x≤5)，不实现 SO_LINGER 选项的这个特性。

   偶尔张贴在 Usenet 上的消息提倡使用此特性，其目的仅仅为了避免 TIME_WAIT 状态，或者在即使跟某个服务器的众所周知端口的连接仍在使用的情况下也能重启其监听服务器。这万万不可，它可能导致数据破坏，细节见 RFC 1337[Braden 1992a]。作为替代，总是在服务器程序中调用 bind 前使用 SO_REUSEADDR 套接口选项，我们马上会论述到。TIME_WAIT 状态是我们的朋友，它是有助于我们的(也就是说，使旧的重复分节在网络中超时消失)。不要试图避免这个状态，我们应该弄清楚它(2.6 节)。

3. 如果 l_onoff 为非 0 值且 l_linger 也为非 0 值，那么当套接口关闭时内核将拖延一段时间。也就是说，如果在套接口发送缓冲区中仍残留有数据，进程将处于睡眠状态，一直到(a)所有数据都已发送完且均被对方确认或(b)延滞时间到。如果套接口被设置为非阻塞型(第 15 章)，它将不等待 close 完成，即使延滞时间为非 0 也是如此。当使用 SO_LINGER 选项的这个特性时，应用进程检查 close 的返回值是非常重要的，因为，如果在数据发送完并被确认前延滞时间到的话，close 将返回 EWOULDBLOCK 错误且套接口发送缓冲区中的任何数据都丢失。

   不幸的是，第三种情况中对成员 l_linger 非 0 的解释是依赖不同的实现的。4.4BSD假设其单位是时钟滴答(clock tick)(一百分之一秒)，但 Posix.1g 规定其单位为秒。现有的源自 Berkeley 的实现的另一个问题是成员 l_linger(一个整数)是拷贝到 16 位的带符号整数内核变量(so_linger)上的，这就限制了延滞时间最大为 32767 时钟滴答即 327.67 秒。

现在我们需要看看，对于已讨论的各种情况，套接口上的 close 确切来说是什么时候返回的。我们假设客户写数据到套接口上，然后调用 close。图 7.6 给出了缺省情况。

我们假设在客户数据到达时，服务器暂时处于忙状态。所以数据由 TCP 加入到其套接口接收缓冲区中。类似地，下一个分节即客户的 FIN 也加入到套接口接收缓冲区中(不论实

现以何种方法记录连接上已收到一个 FIN 的事件)。但是缺省时,客户的 close 立即返回。如图所示,客户的 close 可以在服务器读套接口接收缓区冲的剩余数据之前就返回。对于服务器主机来说,在服务器应用进程读此剩余数据之前就崩溃是完全可能的,且客户应用进程永远不会知道。

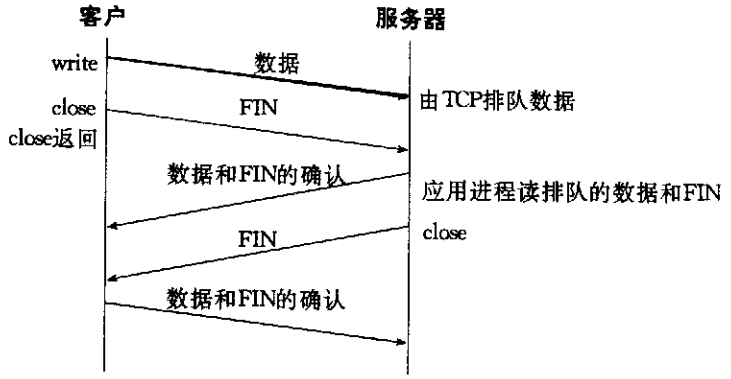


图 7.6　close 的缺省操作:立即返回

客户可以设置 SO_LINGER 套接口选项,指定一个正的延滞时间。这种情况下,客户的 close 要直到它的数据和 FIN 已被 TCP 服务器确认后才返回,如图 7.7 所示。但我们仍有与图7.6类似的问题:在服务器应用进程读剩余数据之前,服务器主机可能崩溃,并且客户应用进程永远不会知道。

这里有一个基本原则:设置 SO_LINGER 套接口选项后,close 的成功返回仅告诉我们发送的数据(和 FIN)已由对方 TCP 确认,它并不能告诉我们对方应用进程是否已读了数据。如果不设置该选项,我们连对方 TCP 是否确认了数据都不知道。

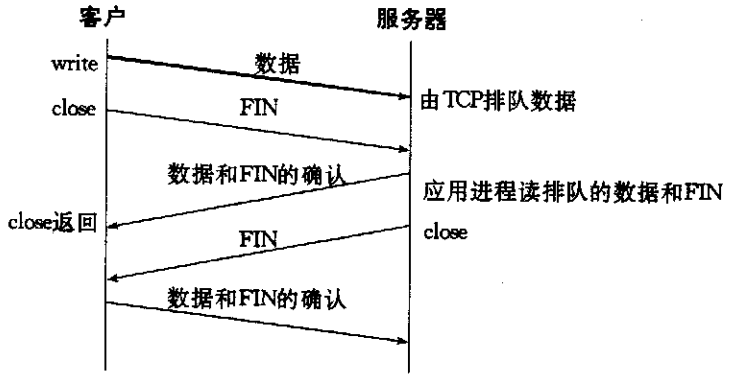


图 7.7　设置 SO_LINGER 套接口选项且 l_linger 为正值时的 close

让客户知道服务器已读其数据的一个方法是:调用 shutdown(第二个参数设为 SHUT_WR)而不是调用 close,并等待对方 close 连接的本地(服务器)端,如图 7.8 所示。

将此图与图 7.6 和图 7.7 进行比较,我们发现,当关闭连接的客户端时,根据所调用的函数(close 或 shutdown)以及是否设置了 SO_LINGER 套接口选项,可在以下三个不同的时机返回:

1. close 立即返回，不等待(缺省状况，图 7.6)。
2. close 一直拖延到接收了 FIN 的 ACK(图 7.7)。

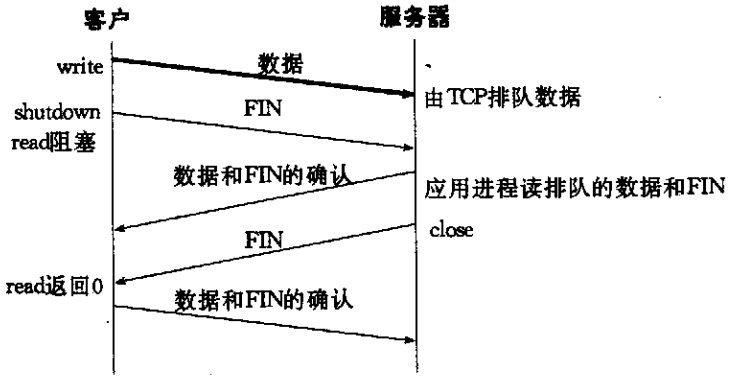


图 7.8 用 shutdown 来获知对方已接收数据

3. 后跟一个 read 调用的 shutdown 一直等到接收了对方的 FIN 才返回(图 7.8)。

获知对方应用进程已读我们的数据的另外一个方法是：使用一个应用级的确认(application-level acknowledge)即应用 ACK(application ACK)。例如，客户给服务器发数据后，调用 read 来读 1 个字节的数据：

```
char ack;
Write(sockfd, data, nbytes);        /* data from client to server */
n = Read(sockfd, &ack, 1);          /* wait for application-level ACK */
```

服务器从客户读数据后，发回 1 字节的应用级 ACK：

```
nbytes = Read(sockfd, buff, sizeof(buff));      /* data from client */
        /* server verifies it received the correct
           amount of data from the client */
Write(sockfd, "", 1);                           /* server's ACK back to client */
```

当客户上的 read 返回时，我们可以保证服务器进程已读完了我们所发送的所有数据。(假设服务器知道客户要发送多少数据，或是由应用程序定义了某种记录结束标志，但这儿没有给出。)这里，应用级 ACK 是 1 字节的 0，但此字节的内容还可以用来指示从服务器到客户的其他条件。图7.9展示了可能进行的分组交换过程：

图 7.10 总结了对 shutdown 的两种可能调用和对 close 的三种可能调用，以及它们对 TCP 套接口的影响。

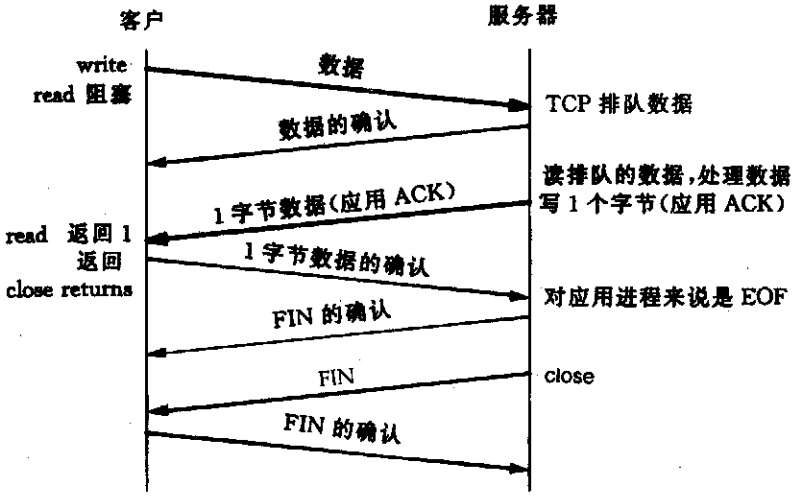


图 7.9　应用 ACK

函数	说　明
shutdown，SHUT_RD	在套接口上不能再发出接收请求；进程仍然可往套接口上发送；套接口接收缓冲区中所有数据被丢弃；再接收的任何数据都由 TCP 丢弃(习题 6.5)；对套接口发送缓冲区无任何影响
shutdown，SHUT_WR	在套接口上不能再发出发送请求；进程仍然可从套接口上接收；套接口发送缓冲区中的内容被发往另一端，后跟正常的 TCP 连接终止序列(FIN)；对套接口接收缓冲区无任何影响
close，l_onoff = 0 (缺省情况)	在套接口上不能再发出发送或接收请求；套接口发送缓冲区中的内容被发往另一端。如果描述字访问计数变为 0：在发完发送缓冲区中的数据后，发出正常的 TCP 连接终止序列(FIN)，套接口接收缓冲区中内容被丢弃
close，l_onoff = 1 l_linger = 0	在套接口上不能再发出发送或接收请求。如果描述字访问计数变为 0：RST 被发往另一端，连接状态被置为 CLOSED(无 TIME_WAIT 状态)，套接口发送缓冲区和套接口接收缓冲区中的数据被丢弃
close，l_onoff = 1 l_linger ! = 0	在套接口上不能再发出发送或接收请求；套接口发送缓冲区中的数据被发往另一端。如果描述字访问计数变为 0：在发完发送缓冲区中的数据后，发出正常的 TCP 连接终止序列(FIN)，套接口接收缓冲区中内容被丢弃，如果在连接变为 CLOSED 状态前延滞时间到，则 close 返回 EWOULDBLOCK 错误

图 7.10　shutdown 和 SO_LINGER 各种情况的总结

SO_OOBINLINE 套接口选项

当此选项打开时，带外数据将被留在正常的输入队列中(即在线存放)。当发生这种情况时，接收函数的 MSG_OOB 标志不能用来读带外数据。第 21 章中我们将详细讨论带外数据。

SO_RCVBUF 和 SO_SNDBUF 套接口选项

每个套接口都有一个发送缓冲区和一个接收缓冲区。在图 2.11 和图 2.12 中，我们描述了 TCP 和 UDP 发送缓冲区的操作。

接收缓冲区被 TCP 和 UDP 用来将接收到的数据一直保存到由应用进程来读。对于 TCP 来说，套接口接收缓冲区中可用空间的大小就是 TCP 通告另一端的窗口大小。TCP 套接口接收缓冲区不可能溢出，因为对方不允许发出超过所通告窗口大小的数据。这就是 TCP 的流量控制，如果对方无视窗口大小而发出了超过窗口大小的数据，则接收方 TCP 将丢弃它。然而，对于 UDP 来说，当接收到的数据报装不进套接口接收缓冲区时，此数据报就被丢弃。回忆一下，UDP 是没有流量控制的：快的发送者可以很容易地就淹没慢的接收者，导致接收方的 UDP 丢弃数据报。这一点我们在 8.13 节再作说明。

这两个套接口选项使我们可以改变缺省大小。对于不同的实现，缺省值的大小可以有很大的差别。较早期的源自 Berkeley 的实现将 TCP 发送和接收缓冲区的大小均缺省为 4 096 字节，但较新的系统使用较大的值，可以是 8 192～61 440 字节间的任一数据。如果主机支持 NFS，则 UDP 发送缓冲区的大小经常缺省为 9 000 字节左右的一个值，而 UDP 接收缓冲区的大小则经常缺省为 40 000 字节左右的一个值。

当设置 TCP 套接口接收缓冲区的大小时，函数调用的顺序是很重要的，这是因为 TCP 的窗口规模选项(2.5 节)是在建立连接时用 SYN 与对方互换得到的。对于一个客户，这意味着 SO_RCVBUF 选项必须在调用 connect 之前设置；对于一个服务器，这意味着在调用 listen 之前必须给监听套接口设置这个选项。给已连接套接口设置这个选项对可能的窗口规模选项无任何影响，因为 accept 要直到 TCP 的三路握手完成才会创建并返回已连接套接口。这就是为什么必须给监听套接口设置此选项的原因(套接口缓冲区的大小总是由新创建的已连接套接口从监听套接口继承来的：TCPv2 第 462～463 页)。

TCP 套接口缓冲区的大小至少必须是连接的 MSS 的三倍。如果我们涉及到单向数据传输，如单方向的文件传送，当我们说“套接口缓冲区大小”时，我们指发送主机端的套接口发送缓冲区大小和接收主机端的套接口接收缓冲区大小。对于双向数据传输，在发送端我们指两个套接口缓冲区的大小，在接收端也指两个套接口缓冲区的大小。典型的缓冲区大小缺省值是 8192 字节或更大，典型的 MSS 为 512 或 1460，因此 TCP 套接口缓冲区大小的要求一般总能满足。在大 MTU 的网络上会出现问题，因为它提供一个比通常 MSS 更大的值(例如，[Comer 和 Lin 1994]中描述的 ATM 网络的 MTU 为 9188)。

TCP 套接口缓冲区大小还必须是连接的 MSS 的偶数倍。有些实现给应用进程处理这个细节问题，在连接建立后向上舍入套接口缓冲区大小(TCPv2 第 902 页)。这是在建立连接之前设置这两个套接口选项的另外一个原因。例如，使用缺省的 4.4BSD 大小 8192，并假设以太网的 MSS 为 1460，在连接建立时两个套接口缓冲区向上舍入成 8760(6×1460)。

在设置套接口缓冲区大小时另一个需考虑的问题涉及到性能。图 7.11 所示为容量为 8 个分节的两端点间的 TCP 连接(我们称其为管道)。

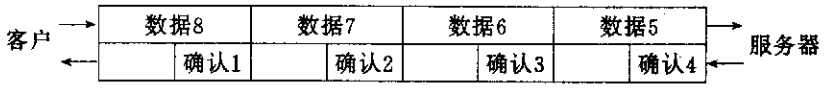


图 7.11　8 个分节容量的 TCP 连接(管道)

我们在上面给出四个数据分节，下面给出四个 ACK。即使管道中只有四个数据分节，客户也必须有至少 8 个分节容量的发送缓冲区，因为客户 TCP 必须为每个分节保留一个拷贝，直到接收到来自服务器的 ACK。

> 这里我们忽略了一些细节。首先，TCP 的慢启动算法限制了在一个空闲连接上最初发送分节的速度；其次，TCP 常常每两个分节确认一次，而不是我们所示的每个分节确认一次。所有这些细节在 TCPv1 的第 20 章和第 24 章均有阐述。

理解的重点是全双工管道的概念、它的容量及如何关系到连接两端的套接口缓冲区大小。管道的容量称为带宽-延迟积(bandwidth-delay product)，我们可以将带宽(位/秒)乘上 RTT(秒)，并将结果由位转换为字节来计算得到。RTT 可很容易地用程序 Ping 来测量到，带宽是两端点间相应于最慢链路的值，某种程度上是已知的。例如，一条 RTT 为 63ms 的 T1 链路(1 536 000 位/秒)的带宽-延迟积为 11 520 字节。如果套接口缓冲区大小小于此值，管道将不会处于满状态，性能将低于期望值。当带宽变大(如 45 兆位/秒的 T3 链路)或 RTT 变大(如 RTT 约为 500ms 的卫星链路)时，就要求有较大的套接口缓冲区。当带宽-延迟积超过 TCP 的最大正常窗口大小(65535 字节)时，两端都需设置 TCP 长胖管道(long fat pipe)选项，这在 2.5 节中已有叙述。

> 大多数实现对套接口发送和接收缓冲区的大小有一个上限，有时这个上限可由管理员来进行修改。较早期的源自 Berkeley 的实现有一个大约 52 000 字节的硬上限，但较新的实现将缺省值上升为 256 000 字节甚至更大，且还常常可由管理员来增加。不幸的是，对于应用程序来说，没有一个简单的方法来确定此极限。Posix.1 定义了函数 fpathconf，这个函数大多数实现都支持；Posix.1g 定义了一个新的常值_PC_SOCK_MAXBUF，它可用作此函数的第二个参数，返回套接口缓冲区的最大值。

SO_RCVLOWAT 和 SO_SNDLOWAT 套接口选项

每个套接口也都有一个接收低潮限度和一个发送低潮限度。它们是函数 select 使用的，如6.3节所述。这两个套接口选项使我们可以修改这两个低潮限度。

接收低潮限度是让 select 返回“可读”而在套接口接收缓冲区中必须有的数据总量。对于一个 TCP 或 UDP 套接口，此值缺省为 1。发送低潮限度是让 select 返回“可写”而在套接口发送缓冲区中必须有的可用空间。对于 TCP 套接口，此值常缺省为 2048。对于 UDP 使用低潮限度的情况，在 6.3 节已作过讲述，但由于对于 UDP 套接口来说，发送缓冲区中可用空间的字节数是从不变化的(因为 UDP 不保留由应用进程发送的数据报的拷贝)，只要 UDP 套接口发送缓冲区大小大于套接口的低潮限度，这样的 UDP 套接口就总是可写的。回忆一下图 2.12，UDP 没有发送缓冲区，只有发送缓冲区的大小。

Posix.1g 不要求支持这两个套接口选项。

SO_RCVTIMEO 和 SO_SNDTIMEO 套接口选项

这两个选项使得我们可以给套接口设置一个接收和发送超时。注意，访问它们的两个 sockopt 函数的参数是指向 timeval 结构的指针，与 select 所用参数相同(6.3 节)，这可让我们用秒数和微秒数来规定超时。我们通过设置其值为 0 秒和 0 微秒来禁止超时。缺省时两个超时都是禁止的。

接收超时影响五个输入函数：read、readv、recv、recvfrom 和 recvmsg；发送超时影响五个输出函数：write、writev、send、sendto 和 sendmsg。13.2 节我们将详细讨论关于套接口的超时。

这两个套接口选项以及套接口接收和发送超时的继承概念是在 4.3BSD Reno 中增加的。Posix.1g 不要求支持这两个套接口选项。

在源自 Berkeley 的实现中，这两个值实际上实现不活动时间计时器而不是读或写系统调用的绝对计时器，TCPv2 第 496～516 页对此作了详细讨论。

SO_REUSEADDR 和 SO_REUSEPORT 套接口选项

SO_REUSEADDR 套接口选项为以下四个不同的目的提供服务：

1. SO_REUSEADDR 允许启动一个监听服务器并捆绑其众所周知端口，即使以前建立的将此端口用作它们的本地端口的连接仍存在。这个条件通常是这样碰到的：

   (a) 启动一个监听服务器；
   (b) 连接请求到达，派生一个子进程来处理这个客户；
   (c) 监听服务器终止，但子进程继续为现有连接上的客户提供服务；
   (d) 重启监听服务器。

   缺省时，当监听服务器在步骤(d)通过调用 socket、bind 和 listen 重新启动时，因为它试图捆绑一个现有连接上的端口(正由以前派生的子进程所处理)，所以调用 bind 失败。但若服务器在 socket 和 bind 的调用间设置 SO_REUSEADDR 套接口选项，bind 将成功。所有 TCP 服务器应指定此套接口选项以允许服务器在这种情况下被重新启动。

   这个情况是 Usenet 中问得最频繁的一个问题。

2. SO_REUSEADDR 允许在同一端口上启动同一服务器的多个实例，只要每个实例捆绑一个不同的本地 IP 地址即可。这对用 IP 别名技术(A.4 节)来提供多个 HTTP 服务器的网点(site)来说是很常见的①。假设本地主机的主 IP 地址为 198.69.10.2，但它有两个别名：198.69.10.128 和 198.69.10.129。启动三个 HTTP 服务器。第一

① 译者注：网点(site)是一个用得越来越广泛的概念，但它没有精确的定义。网点概念最初指互联网上的单个子网或网络。从网络路由拓扑图看，子网或网络用点来表示，而把它们连接起来的路由器则用边来表示。因而有网点的说法。随着子网划分(subnetting)、超网合并(supernetting)、无类域间路由(CIDR)等技术的应用，网点能表示的范围也越来越大，但最大不会超过一个自治系统(AS)。另外，网点概念也用在逻辑范围，例如同一 NIS 域的所有主机，同一 Kerberos 域的所有主机等等都能构成单个网点。本书所指的网点应小于一个组织，例如，一家跨国公司(组织)的各个子公司可能就是单个网点。site 有时指单个主机，这时我们把它译成站点。

个 HTTP 服务器以本地 IP 地址 198.69.10.128 和本地端口号 80(HTTP 的众所周知)调用 bind。第二个 HTTP 服务器捆绑 198.69.10.129 和端口 80,但它的 bind 调用将失败,除非在调用前设置 SO_REUSEADDR 选项。第三个 HTTP 服务器以通配地址作为本地 IP 地址和端口号 80 调用 bind。同样,设置 SO_REUSEADDR 是这个调用成功所必需的。假设设置了 SO_REUSEADDR,从而三个服务器都启动,目的 IP 地址为 198.69.10.128、目的端口号为 80 的外来 TCP 连接请求将递送给第一个服务器,目的 IP 地址为 198.69.10.129、目的端口号为 80 的外来请求将递送给第二个服务器,目的端口号为 80 的所有其他请求将都递送给第三个服务器。最后一个服务器处理目的地址为 198.69.10.2 或该主机已配置的其他任何 IP 别名的请求。通配的意思就是"没有更好的(即更为具体的)匹配的任何地址"。注意,允许某个给定服务存在多个服务器的情形在服务器总是设置 SO_REUSEADDR 套接口选项时是自动处理的(我们建议设置这个选项)。

对于 TCP,我们根本不能启动捆绑相同 IP 地址和相同端口号的多个服务器:这是完全重复的捆绑(completely duplicate binding)。也就是说,即使我们给第二个服务器设置了 SO_REUSEADDR 套接口选项,也不能在启动绑定 198.69.10.2 和端口 80 的服务器后,接着再启动捆绑 198.69.10.2 和端口 80 的另一个服务器。

3. SO_REUSEADDR 允许单个进程捆绑同一端口到多个套接口上,只要每个捆绑指定不同的本地 IP 地址即可。在不支持 IP_RECVDSTADDR 套接口选项的系统上,这对于要求知道客户请求的目的 IP 地址的 UDP 服务器来说是非常普遍的,在 19.11节中我们将用此技术开发一个例子。这项技术一般不用于 TCP 服务器,因为 TCP 服务器在建立连接后总是能通过调用 getsockname 来确定客户请求的目的 IP 地址。
4. SO_REUSEADDR 允许完全重复的捆绑:当一个 IP 地址和端口绑定到某个套接口上时,还允许此 IP 地址和端口捆绑到另一个套接口上。一般来说,这个特性仅在支持多播的系统上才有,这些系统可能还不支持 SO_REUSEPORT 套接口选项(我们马上叙述到),而且仅对 UDP 套接口而言(TCP 不进行多播)。

   此特性用于多播时,允许同一主机上多次运行相同的应用程序。当一个 UDP 数据报需由这些重复捆绑套接口中的一个接收时,所用规则为:如果数据报的目的地址是一个广播地址或多播地址,就给每个匹配的套接口递送数据报的拷贝。但是如果数据报的目的地址是一个单播地址,那它只递送到单个套接口。在单播数据报情况下,如果有多个套接口匹配数据报,则哪个套接口将接收数据报是依赖于实现的。TCPv2 第 777～779 页对此特性作了详细的讨论。我们将在第 18 章和第 19 章对广播和多播作详细的讨论。

习题 7.5 和 7.6 给出了几个此套接口选项的例子。

添加多播支持以后,4.4BSD 引入了 SO_REUSEPORT 套接口选项。与多播中完全重复捆绑的 SO_REUSEADDR 不同,该套接口选项具有以下语义:

1. 此选项允许完全重复捆绑,但仅在每个想捆绑相同 IP 地址和端口的套接口都指定了此套接口选项才行。

2. 如果被捆绑的IP地址是一个多播地址，则SO_REUSEADDR与SO_REUSEPORT等效(TCPv2第731页)。

此套接口选项的一个问题是：并非所有系统都支持它，在那些不支持此选项但支持多播的系统上，使用SO_REUSEADDR而不是SO_REUSEPORT以允许完全重复捆绑(也就是说，同一时刻在同一主机上可运行多次并期待接收广播或多播数据报的UDP服务器)。

我们以下面的建议来总结对这些套接口选项的讨论：

1. 在所有TCP服务器程序中，在调用bind之前设置SO_REUSEADDR套接口选项；
2. 当编写一个同一时刻在同一主机上可运行多次的多播应用程序时，设置SO_REUSEADDR套接口选项，并将本组的多播地址作为本地IP地址捆绑。

TCPv2第22章对这两个套接口选项作了详细的讨论。

SO_REUSEADDR有一个潜在的安全问题。譬如说，如果存在一个套接口，它绑定了通配地址和端口5555，如果我们指定SO_REUSEADDR，我们可以捆绑相同的端口到不同的IP地址上，譬如说主机的主IP地址。目的地为端口5555和我们的套接口上所绑定IP地址的数据报将递送到我们的套接口上，而不是绑定通配地址的另一个套接口上。它们可以是TCP SYN分节或UDP数据报。(习题11.3展示了UDP的这个特性。)对于大多数众所周知的服务如HTTP、FTP和Telnet来说，这不成问题，因为这些服务器绑定一个保留端口。因此，后来的试图更为详细地捆绑此端口的任何实例(也就是说盗用端口)将要求有超级用户特权。然而，NFS可能是一个问题，因为它的通常端口(2049)不被保留。

> 套接口API的一个底层问题是：套接口对的设置是由两个函数调用(bind和connect)来完成的，而不是一个。[Torek 1994]为解决此问题提议了如下单个函数：
>
> ```
> int bind_connect_listen(int sockfd,
>                         const struct sockaddr *laddr, int laddrlen,
>                         const struct sockaddr *faddr, int faddrlen,
>                         int listen);
> ```
>
> laddr指定本地IP地址和本地端口号，faddr指定远程IP地址和远程端口号，listen指定一个客户(0)或服务器(非0，与函数listen的backlog参数相同)。这样的话，bind将是一个用空指针的faddr和为0的faddrlen来调用此函数的库函数，connect将是一个用空指针的laddr和为0的laddrlen来调用此函数的库函数。有些应用程序，尤其是FTP，需要指明本地对和远程对，它们可以直接调用bind_connect_listen。有了这样的一个函数就不需要SO_REUSEADDR了，除非多播服务器显式地要求允许(对同一IP地址和端口)完全重复捆绑。此函数的另一个好处是：TCP服务器可以限制自己为来自指定的IP地址和端口的连接请求提供服务，这是RFC 793[Postel 1981c]规定的，但对现有的套接口API又是不可能的事情。
>
> 一个类似的、名为set_addresses的函数在1993年由Keith Sklower提议到了Posix的1003.12工作组上，不过这个提议被拒绝了。

SO_TYPE 套接口选项

这个选项返回套接口的类型，返回的整数值是一个诸如 SOCK_STREAM 或 SOCK_DGRAM 这样的值。此选项典型地由启动时继承了套接口的进程所用。

SO_USELOOPBACK 套接口选项

此选项仅用于路由域(AF_ROUTE)的套接口，它对这些套接口的缺省设置为打开(这是唯一一个缺省为打开而不是关闭的 SO_xxx 套接口选项)。当此选项打开时，套接口接收在其上发送的任何数据的一个拷贝。

> 禁止这些回馈拷贝的另一个方法是调用 shutdown，第二个参数应设为 SHUT_RD。

7.6 IPv4 套接口选项

这些选项由 IPv4 处理，级别为 IPPROTO_IP。我们将五个多播套接口选项推迟到 19.5 节再做讨论。

> 除 IP_RECVIF 外，本节中我们所描述的所有套接口选项均由 Posix.1g 定义。

IP_HDRINCL 套接口选项

如果此选项给一个原始 IP 套接口(第 25 章)设置，则我们必须为所有发送到此原始套接口上的数据报构造自己的 IP 头部。一般情况下，内核为发送到原始套接口上的数据报构造 IP 头部，但也有某些应用程序(尤其是路由跟踪程序 Traceroute)要构造自己的 IP 头部以取代 IP 可能放到其头部的某些字段的值。

当设置此选项时，我们构造完整的 IP 头部，不过下列情况例外：

- IP 总是计算并存储 IP 头部校验和；
- 如果我们将 IP 标识字段置为 0，内核将设置此字段；
- 如果源 IP 地址是 IN_ADDR_ANY，IP 将它设置为外出接口的主 IP 地址；
- 如何设置 IP 选项是依赖于实现的。有些实现取 IP_OPTIONS 套接口选项中设置的任何 IP 选项，并将它们附加到我们所构造的头部中，而其他实现则要求我们亲自在头部指定任何期望的 IP 选项。

在 26.6 节中，我们将给出此选项的一个例子，TCPv2 第 1056～1057 页提供了此套接口选项额外的详细信息。

IP_OPTIONS 套接口选项

设置此选项允许我们在 IPv4 头部中设置 IP 选项，这要求掌握 IP 头部中 IP 选项的格式信息。在 24.3 节中，我们论述 IPv4 源路径时再讨论这个选项。

IP_RECVDSTADDR 套接口选项

这个套接口选项导致所接收到的 UDP 数据报的目的 IP 地址由函数 recvmsg 作为辅助

数据返回。20.2节中我们给出此选项的一个例子。

IP_RECVIF 套接口选项

这个套接口选项导致所接收到的UDP数据报的接口索引由函数recvmsg作为辅助数据返回,20.2节中我们给出这个选项的一个例子。

> 这是一个新的套接口选项,它由Bill Fenner为FreeBSD和NetBSD的DARTNet测试床而开发[Fenner 1997]。DARTNet是一个用于测试新协议和新应用程序的实验性研究网络。该套接口选项原本以为会加到4.4BSD中,结果却是一直没有加到此版本中。作者使用FreeBSD实现并把它加入到BSD/OS 3.0中。

IP_TOS 套接口选项

此选项使我们可以给TCP或UDP套接口在IP头部中设置服务类型字段(图A.1)。如果我们给此选项调用getsockopt,则放到外出IP数据报头部的TOS字段中的当前值(缺省为0)将返回。还没有方法从接收到的IP数据报中取此值。

我们可以将TOS设置为图7.12所示常值中的一个,它们都在头文件<netinet/ip.h>中定义。

常值	解释
IPTOS_LOWDELAY	最小化延迟
IPTOS_THROUGHPUT	最大化吞吐量
IPTOS_RELIABILITY	最大化可靠性
IPTOS_LOWCOST	最小化成本

图7.12 IPv4服务类型常值

RFC 1349[Almquist 1992]中有TOS字段的详细描述,并介绍此字段应如何给标准的因特网应用程序设置。例如,Telnet和Rlogin应指定IPTOS_LOWDELAY,而FTP传送的数据部分则应指定IPTOS_THROUGHPUT。

IP_TTL 套接口选项

用此选项,我们可以设置和获取系统用于某个给定套接口的缺省TTL值(存活时间字段,见图A.1)。例如,4.4BSD对TCP和UDP套接口都使用缺省值64(这在RFC 1700[Reynolds and Postel 1994]中规定),对原始套接口则使用缺省值255。跟TOS字段一样,调用getsockopt返回系统用于外出数据报的缺省TTL字段值,也就是说没有办法从接收到的数据报中得到此值。图25.18中的Traceroute程序设置了此套接口选项。

7.7 ICMPv6套接口选项

这个唯一的套接口选项由ICMPv6所处理,并具有IPPROTO_ICMPV6级别。

ICMP6_FILTER 套接口选项

此选项使我们可以获取和设置一个 icmp6_filter 结构，它指明 256 个可能的 ICMPv6 消息类型中哪一个传递给在原始套接口上的进程。在 25.4 节我们再讨论此选项。

7.8　IPv6 套接口选项

这些选项由 IPv6 所处理，并具有 IPPROTO_IPV6 级别。我们将五个多播套接口选项推迟到 19.5 节再做讨论。我们注意到，许多这些选项都利用函数 recvmsg 的辅助数据，在 13.6 节我们再对此进行讨论。所有的 IPv6 套接口选项都定义在 RFC 2133[Gilligan et al. 1997]和[Stevens and Thomas 1997]中。

Posix.1g 对 IPv6 未做任何说明。

IPV6_ADDRFORM 套接口选项

这个选项允许套接口从 IPv4 转换到 IPv6，反之亦可。我们在 10.5 节中再讲述这个选项。

IPV6_CHECKSUM 套接口选项

此选项指定用户数据中校验和所处位置的字节偏移。如果此值为非负，则内核将(1)给所有外出分组计算并存储校验和；(2)输入时检查所收到分组的校验和，丢弃带有无效检验和的分组。此选项影响除 ICMPv6 原始套接口外的所有 IPv6 原始套接口。(内核总是给 ICMPv6 原始套接口计算并存储校验和。)如果指定的值为−1(缺省值)，内核在此原始套接口上将不给外出的分组计算并存储较验和，也不检查所收到分组的校验和。

> 所有使用 IPv6 的协议应在它们自己的协议头部有一个校验和。这些校验和包括一个伪头部(pseudoheader)(RFC 1883[Deering 和 Hinden 1995])，它把源 IPv6 地址作为校验和的一部分(这与使用 IPv4 的原始套接口来实现的其他协议有所不同)。这样不必强求使用原始套接口的应用进程进行源地址选择，相反，内核将这么做，并计算、存储含有标准 IPv6 伪头部的检验和。

IPV6_DSTOPTS 套接口选项

设置此选项指明：任何接收到的 IPv6 目标选项都将由 recvmsg 作为辅助数据返回。此选项缺省为关闭。我们在 24.5 节将描述用来创建并处理这些选项的函数。

IPV6_HOPLIMIT 套接口选项

设置此选项指明：接收到的跳限字段将由 recvmsg 作为辅助数据返回。此选项缺省为关闭。我们在 20.8 节中描述此选项。

对 IPv4 而言，是没有办法来取得接收到的存活时间字段的。

IPV6_HOPOPTS 套接口选项

设置此选项指明：任何接收到的 IPv6 步跳选项都将由 recvmsg 作为辅助数据返回。此

选项缺省为关闭。我们在24.5节描述用来创建和处理这些选项的函数。

IPV6_NEXTHOP

这不是一个套接口选项，而是一个可指定给sendmsg的辅助数据对象的类型。此对象以一个套接口地址结构指定某个数据报的下一跳地址。20.8节我们将对此做详细的描述。

IPV6_PKTINFO套接口选项

设置此选项表明，下面关于接收到的IPv6数据报的两条信息将由recvmsg作为辅助数据返回：目的IPv6地址和到达接口索引。在20.8节中我们再描述此选项。

IPV6_PKTOPTIONS套接口选项

大多数IPv6套接口选项假设UDP套接口使用recvmsg和sendmsg所用的辅助数据在内核与应用进程间传递信息。TCP套接口使用IPV6_PKTOPTIONS套接口选项来获取和存储这些值。

由getsockopt和setsockopt所指的缓冲区中的信息与recvmsg或sendmsg利用辅助数据传递的信息是相同的。我们在24.7节中讨论这个选项。

IPV6_RTHDR套接口选项

设置这个选项表明：接收到的IPv6路由头部将由recvmsg作为辅助数据返回。此选项缺省为关闭。我们在24.6节将描述用来创建并处理IPv6路由头部的函数。

IPV6_UNICAST_HOPS套接口选项

此IPv6选项类似于IPv4的IP_TTL套接口选项。它的设置指定发送到套接口上的外出数据报的缺省跳限，而它的获取则返回内核将用于套接口的跳限值。为了从接收到的IPv6数据报中得到真实的跳限字段，要求使用IPV6_HOPLIMIT套接口选项。我们在图25.18的Traceroute程序中就设置此套接口选项。

7.9 TCP套接口选项

TCP有五个套接口选项，但有三个是随Posix.1g新定义的且没有得到广泛支持。我们规定它们的级别为IPPROTO_TCP。

TCP_KEEPALIVE套接口选项

此选项是随Posix.1g新定义的。它指定TCP开始发送保持存活探测分节前以秒为单位的连接空闲时间。缺省值至少必须为7200秒，即2小时。此选项仅在SO_KEEPALIVE套接口选项打开时才有效。

TCP_MAXRT套接口选项

此选项是随Posix.1g新定义的。它指定一旦TCP开始重传数据，在连接断开之前需经历的以秒为单位的时间总量。值0意味着使用系统缺省值，值－1意味着永远重传数据。如果指定一个正值，它可能向上舍入成实现的下一次重传时间。

TCP_MAXSEG 套接口选项

此选项允许我们获取或设置 TCP 连接的最大分节大小(MSS)。返回值是我们的 TCP 发送给另一端的最大数据量,它常常就是由另一端用 SYN 分节通告的 MSS,除非我们的 TCP 选择使用一个比对方通告的 MSS 小些的值。如果此值在套接口连接之前取得,则返回值为未从另一端收到 MSS 选项的情况下所用的缺省值。同时也要注意到,小于此返回值的值可能真正用在连接上,因为譬如说使用时间戳选项的话,它在每个分节上占用 12 字节的 TCP 选项容量。

我们的 TCP 将发送的每个分节的最大数据量也可在连接存活期内改变,但前提是 TCP 要支持路径 MTU 发现功能。如果到对方的路径改变了,此值可上下调整。

在图 7.1 中我们注意到,此套接口选项也可由应用进程设置。在 4.4BSD 出现之前,这是不可能的:它是一个只读选项。4.4BSD 只允许应用进程减少该值:我们不能增加它的值(TCPv2 第 1023 页)。由于这个选项控制 TCP 发送的每个数据分节的数据总量,它禁止应用进程增加值是明智的。一旦连接建立,此值即对方通告的 MSS 选项,我们不能超过此值。然而,我们的 TCP 总是可以发送少于对方通告的 MSS 值的数据。

TCP_NODELAY 套接口选项

如果设置,此选项禁止 TCP 的 Nagle 算法(TCPv1 的 19.4 节和 TCPv2 第 858～859 页)。缺省时,此算法是使能的。

Nagle 算法的目的是减少 WAN 上小分组的数目。算法指出:如果给定连接上有待确认(outstanding)数据(也就是说,我们的 TCP 已发送但还在等待对方确认的数据),则直到现有数据被确认前,将不往连接上发送任何小分组。"小"分组的定义即任何小于 MSS 的分组。如果可能,TCP 总是发送最大大小的分组;Nagle 算法的目的是防止任何时刻连接上有多个待确认的小分组存在。

最容易产生小分组的两个客户是 Rlogin 和 Telnet,因为它们将每次击键都作为一个独立分组发送。在快速 LAN 上,我们一般不会注意到这些客户使用 Nagle 算法,因为小分组要求的确认时间一般也就几毫秒,远远小于我们键入两个连续字符的时间。但在 WAN 上,小分组等待确认的时间可能要达到一秒,因此我们可以观察到字符回显的延迟,且此延迟常由 Nagle 算法进行夸大。

考虑下面的例子。我们键入六字符的串"hello!"到 Rlogin 或 Telnet 客户上,每个字符间准确地相隔 250ms。到服务器的 RTT 是 600ms,且服务器立即发回每个字符的回显。我们假设客户字符的 ACK 是与字符回显一同发回给客户的,且忽略客户发给服务器的回显 ACK。(我们不久将讨论延滞 ACK。)假设 Nagle 算法没有使能,我们有图 7.13 所示的 12 个分组。

每个字符在自己的分组中发送:数据分节从左到右,ACK 从右到左。

但如果使能了 Nagle 算法(缺省),我们有图 7.14 所示的 8 个分组。第一个字符在它自己的分组中发送。但下两个字符不发送了,因为连接有一个小分组没有得到确认。在时刻 600,当收到第一个分组的 ACK 时(与第一个字符回显一起),这两个字符被发出。直到此分组在时刻 1200 被确认,才会发出另外的小分组。

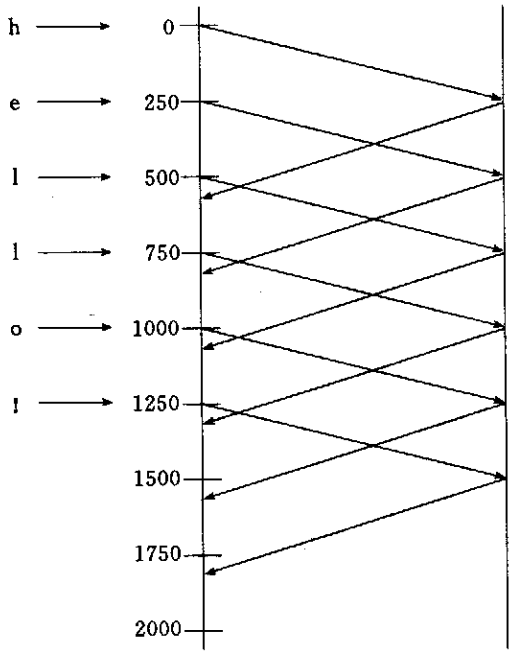


图 7.13 禁止 Nagle 算法时由服务器回显的六个字符

Nagle 算法常常与另一个 TCP 算法联合使用：延滞 ACK(delayed ACK)算法。此算法导致在接收到数据时不立即发送 ACK，而是由 TCP 等待一小段时间(典型地是 50～200ms)才发出 ACK。所希望的是在这一小段时间中接收方有数据发回给对方，这样 ACK 可以捎带在这些数据上发回，于是就节省了一个 TCP 分节。Rlogin 和 Telnet 客户通常使用这种方法，因为服务器一般都回显客户所发的每个字符，这样客户字符的 ACK 完全可以捎带在服务器对此字符的回显上发回。

对于其他客户存在一个问题，即它们的服务器不产生 ACK 可以捎带的反向通信量。这些客户可能觉察到明显的延迟，因为客户 TCP 要直到服务器的延滞 ACK 定时器到才给服务器发数据。这些客户需要一个方法以禁止 Nagle 算法，因此使用选项 TCP_NODELAY。

另一类不适合使用 Nagle 算法和 TCP 的延滞 ACK 算法的客户是：以若干小片数据发送单个逻辑请求给服务器的客户。例如，假设客户给服务器发一个 400 字节的请求，但这是一个 4 字节的请求类型后跟一个 396 字节的请求数据。如果客户执行一个 4 字节的写，后跟一个 396 字节的写，第二个写将一直到服务器 TCP 确认了前 4 字节的写后才由客户 TCP 发出。而且，由于服务器应用要等到其余 396 字节的数据到齐后才能对此 4 字节的数据进行操作，服务器 TCP 将延迟此 4 字节数据的 ACK(也就是说，将不会有从服务器到客户的任何数据可以捎带 ACK)。有三种方法修复这类客户程序：

1. 使用 writev(13.4 节)而不是两次调用 write。writev 对 TCP 输出进行单次调用，而不是两次调用，其结果是我们的例子只用一个 TCP 分节。这是推荐的解决方案。
2. 拷贝 4 字节的数据和 396 字节的数据到一个缓冲区中，然后对此缓冲区调用一次 write。

3. 设置 TCP_NODELAY 套接口选项且继续调用 write 两次。这是最不可取的解决方案。

习题 7.8 和 7.9 继续讨论此例子。

TCP_STDURG 套接口选项

此选项是随 Posix.1g 新定义的。它影响对 TCP 紧急指针的解释(将在第 21 章中讨论带外数据时遇到)。对于 TCP 紧急指针(TCPv1 第 292～293 页)将指向哪,有两种可能的解释。缺省时,紧急指针指向使用 MSG_OOB 标志所发送字节后的数据字节,当今几乎所有实现都是这么解释的。但如果此套接口选项设为非 0,紧急指针就指向使用 MSG_OOB 标志发送的数据字节。

这个选项根本无需设置,为什么 Posix.1g 曾经定义了它也是一个疑问。

7.10　fcntl 函数

fcntl 代表"file control(文件控制)",此函数进行各种描述字控制操作。在描述此函数以及它如何影响套接口之前,我们需要看得远一点。图 7.15 对由 fcntl、ioctl 和路由套接口执行的不同操作进行了总结。

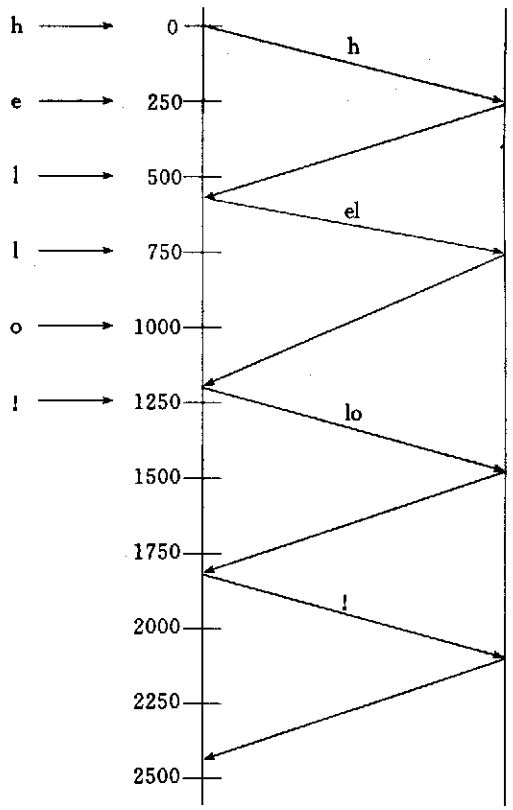


图 7.14　使能 Nagle 算法时由服务器回显的六个字符

操作	fcntl	ioctl	路由套接口	Posix.1g
设置套接口为非阻塞I/O型	F_SETFL，O_NONBLOCK	FIONBIO		fcntl
设置套接口为信号驱动I/O型	F_SETFL,O_ASYNC	FIOASYNC		fcntl
设置套接口属主	F_SETOWN	SIOCSPGRP或FIOSETOWN		fcntl
获取套接口属主	F_GETOWN	SIOCGPGRP或FIOGETOWN		fcntl
获取套接口接收缓冲区中的字节数		FIONREAD		
测试套接口是否在带外数据标志		SIOCATMARK		sockatmark
获取接口表		SIOCGIFCONF	sysctl	
接口操作		SIOC[GS]IFxxx		
ARP高速缓存操作		SIOCxARP	RTM_xxx	
路由表操作		SIOCxxxRT	RTM_xxx	

图7.15 fcntl、ioctl和路由套接口操作小结

前六个操作可由任何进程应用到套接口上，而后四个操作中的许多(接口、ARP和路由表)由诸如ifconfig和route这样的管理程序完成。在第16章中，我们将详细讨论各种ioctl操作。在第17章中，则详细讨论路由套接口的有关操作。

有很多方法来执行前四个操作，但我们注意到，最后一列中Posix.1g规定fcntl是最理想的方法。我们也注意到，Posix.1g提供函数sockatmark(21.3节)作为测试是否在带外数据标志的理想方法。其余的操作，即最后一列为空白者，还没有被Posix标准化。

> 我们还要注意，设置套接口为非阻塞I/O型和信号驱动I/O型的前两个操作，历史上是用fcntl的FNDELAY和FASYNC命令设置的。Posix定义O_xxx常值。

函数fcntl提供了下列关于网络编程的特性：

- 非阻塞I/O。 我们可以通过用F_SETFL命令设置O_NONBLOCK文件状态标志来设置套接口为非阻塞型。我们在第15章讲述非阻塞I/O。
- 信号驱动I/O。 我们可以用F_SETFL命令来设置O_ASYNC文件状态标志，这导致在套接口状态发生变化时内核生成信号SIGIO，这一点我们在第22章再做讨论。

  > 此标志是随Posix.1g新加的。

- F_SETOWN命令让我们设置套接口属主(进程ID或进程组ID)，由它来接收信号SIGIO和SIGURG。前一个信号在设置套接口为信号驱动I/O型时生成(第22章)，后一个信号在新的带外数据到达套接口时生成(第21章)。F_GETOWN命令返回套接口的当前属主。

  > 术语“套接口属主”是随Posix.1g定义的新概念。历史上源自Berkeley的实现

称之为“套接口的进程组 ID”，因为存储此 ID 的变量是结构套接口的成员 so_pgid(TCPv2 第 438 页)。

```
#include <fcntl.h>
int fcntl(int fd, int cmd, ... /* int arg */);
                                    返回：依赖于参数 cmd——成功，-1——出错
```

每个描述字(包括套接口描述字)都有一组由命令 F_GETFL 取得和由命令 F_SETFL 设置的文件标志。影响套接口的两个标志是：

O_NONBLOCK	非阻塞 I/O
O_ASYNC	信号驱动 I/O

后面我们将详细描述这两个特性。现在我们只需注意，用 fcntl 来使能非阻塞 I/O 的典型代码将是：

```
int     flags;
    /* set socket nonblocking */
if ( (flags = fcntl(fd, F_GETFL, 0)) < 0)
    err_sys("F_GETFL error");
flags |= O_NONBLOCK;
if (fcntl (fd, F_SETFL, flags) < 0)
    err_sys("F_SETFL error");
```

下面是你可能会遇到的，简单地设置新期望标志的代码：

```
    /* Wrong way to set socket nonblocking */
if (fcntl(fd,F_SETFL,O_NONBLOCK) <0)
    err_sys("F_SETFL error");
```

这段代码在设置非阻塞标志的同时也清除了所有其他文件状态标志。设置某个文件状态标志的唯一正确的方法是：先取得当前标志，与新标志逻辑或后再设置标志。

下面的代码关闭非阻塞标志，假设标志是由上面所示的 fcntl 调用来设置的

```
flags &= ~O_NONBLOCK;
if (fcntl(fd,F_SETFL,flags) < 0)
    err_sys("F_SETFL error");
```

信号 SIGIO 和 SIGURG 与其他信号的不同之处在于，这两个信号仅在已用命令 F_SETOWN 给套接口指派了属主后才会生成。F_SETOWN 命令的整参数 arg 既可以是一个正整数，指明接收信号的进程 ID，也可以是一个负整数，它的绝对值是接收信号的进程组 ID。F_GETOWN 命令返回套接口属主作为函数 fcntl 的返回值，可以是进程 ID(一个正的返回值)，也可以是进程组 ID(一个除－1 外的负值)。指定进程或进程组来接收信号的区别是：前者只引起一个进程接收信号，而后者则引起进程组中的所有进程(也许多于 1 个进程)接收信号。

SVR4 只允许将套接口的属主设置为进程 ID 而不能是进程组 ID。

当一个新的套接口由函数 socket 创建时，它没有属主，但是当一个新的套接口从一个监

听套接口创建时，套接口属主便由已连接套接口从监听套接口继承而来(许多套接口选项也这样，见 TCPv2 第 462～463 页)。

7.11 小 结

套接口选项从非常普通(SO_ERROR)到非常专门(IP 头部选项)都有。我们可能遇到的、最常用的选项是：SO_KEEPALIVE、SO_RCVBUF、SO_SNDBUF 和 SO_REUSEADDR。最后那个选项应总是在调用 bind 之前给 TCP 服务器设置(图 11.8)。SO_BROADCAST 选项和 10 个多播套接口选项分别用于使用广播或多播的应用程序。

SO_KEEPALIVE 套接口选项由许多 TCP 服务器设置以自动终止一个半开连接。此选项一个较好的特性是它由 TCP 层处理，不要求有一个应用级的无动静计时器，但它的缺点是不能区别客户崩溃与到客户的连通性暂时丢失。

SO_LINGER 套接口选项使我们能更好控制函数 close 返回的时机，让我们可以强制发送一个 RST 而不是 TCP 的四分组连接终止序列。我们必须小心发送 RST，因为这避免了 TCP 的 TIME_WAIT 状态。也有本套接口选项无法提供我们所需信息的情况，此时就要求有应用级的 ACK。

每个 TCP 套接口都有一个发送缓冲区和一个接收缓冲区，每个 UDP 套接口都有一个接收缓冲区。SO_SNDBUF 和 SO_RCVBUF 套接口选项让我们可以改变这些缓冲区的大小。这些选项最常见的用途是用于长胖管道上的批量数据传送。长胖管道是或高带宽或长延时的 TCP 连接，常常使用 RFC 1323 的扩展定义。另一方面，UDP 套接口可能想增加接收缓冲区的大小以允许内核在应用进程较忙时排队更多的数据报。

7.12 习 题

7.1 写一个输出缺省 TCP 和 UDP 发送和接收缓冲区大小的程序，并在你有访问权限的系统上运行该程序。

7.2 将图 1.5 做如下修改：在调用 connect 之前，调用 getsockopt 得到套接口接收缓冲区大小和 MSS，输出这两个值。connect 返回成功后，仍取这两个套接口选项并输出其值，值变化了吗？为什么？运行一个连接到本地网络上的本服务器程序实例，再运行一个连接到远程网络上的本服务器程序实例，MSS 变化吗？为什么？你应在你有访问权的任何不同主机上运行本程序。

7.3 从图 5.2 及图 5.3 的 TCP 服务器程序和图 5.4 及图 5.5 的 TCP 客户程序开始，修改客户程序的 main 函数，以在调用 exit、设置 l_onoff 为 1 和 l_linger 为 0 之前设置 SO_LINGER 套接口选项。先启动服务器，然后启动客户，在客户上键入一行或两行来检验操作，然后键入文件结束符以终止客户，将发生什么情况？在你终止客户后，在客户主机上运行 netstat，看看套接口是否经历了 TIME_WAIT 状态。

7.4 假设两 TCP 客户在同一时间启动，都设置 SO_REUSEADDR 套接口选项，且以相同的本地 IP 地址和相同的端口号(譬如说，1500)调用 bind，但一个客户连接到

198.69.10.2 的端口 7000，另一个客户连接到 198.69.10.2(相同的 IP 地址)的端口 8000。阐述所出现的竞争状态。

7.5　获取本书中例子的源代码(见前言)并编译程序 sock(C.3 节)。首先，将你的主机分类为(1)没有多播支持，(2)有多播支持但不提供 SO_REUSEPORT，(3)有多播支持且提供 SO_REUSEPORT。试着在同一端口上启动程序 sock 的多个实例作为 TCP 服务器(-s 命令行选项)，分别捆绑通配地址、你的主机的某个接口地址以及回馈地址。你需要指定 SO_REUSEADDR 选项(-A 命令行选项)吗？用命令 netstat 来观察监听套接口。

7.6　继续前面的例子，但启动的是 UDP 服务器(-u 命令行选项)并试着启动两个实例，捆绑相同的本地 IP 地址和端口号。如果你的实现支持 SO_REUSEPORT，试着用它(-T 命令行选项)。

7.7　程序 Ping 的许多版本有一个-d 标志来打开 SO_DEBUG 套接口选项，这是干什么用的？

7.8　继续我们在讨论 TCP_NODELAY 套接口选项结尾处的例子，假设客户执行了两个写：第一个 4 字节，第二个 396 字节，还假设服务器的延滞 ACK 时间为 100ms，客户与服务器间的 RTT 是 100ms，服务器处理客户请求的时间是 50ms。画一个时间线图以表示延滞 ACK 与 Nagle 算法的相互作用。

7.9　假设设置了 TCP_NODELAY 套接口选项，重做上个习题。

7.10　假设进程调用 writev 一次性处理完 4 字节缓冲区和 396 字节缓冲区，重做习题 7.8。

7.11　读 RFC 1122[Barden 1989]以确定延滞 ACK 的建议间隔。

7.12　图 5.2 及图 5.3 中的服务器程序什么地方耗时最多？如果服务器设置了 SO_KEEPALIVE 套接口选项，没有数据在连接上交换，客户主机崩溃且没有重启，将发生什么？

7.13　图 5.4 及图 5.5 中的客户程序什么地方耗时最多？如果客户设置了 SO_KEEPALIVE 套接口选项，没有数据在连接上交换，服务器主机崩溃且没有重启，将发生什么？

7.14　图 5.4 及图 6.13 中的客户程序什么地方耗时最多？如果客户设置了 SO_KEEPALIVE 套接口选项。没有数据在连接上交换，服务器主机崩溃且没有重启，将发生什么？

7.15　假设客户和服务器都设置了 SO_KEEPALIVE 套接口选项。通信双方维护连通性但没有数据在连接上交换。当保持存活定时器每 2 小时的定时时间到时，在连接上有多少 TCP 分节被交换？

7.16　几乎所有实现都在头文件<sys/socket.h>中定义了常值 SO_ACCEPTCON，但我们没有描述此选项。读[Lanciani 1996]，弄清此选项为什么存在。

第8章 基本UDP套接口编程

8.1 概 述

在使用TCP与使用UDP的应用程序之间存在本质差异,这是因为两个传输层很不相同:UDP是无连接的、不可靠的数据报协议,而TCP是面向连接的,提供可靠的字节流。然而,有些实例却更适合用UDP而不是TCP,我们到20.4节再讲述这个设计选择。有些流行的应用程序是用UDP实现的:DNS(域名系统)、NFS(网络文件系统)和SNMP(简单网络管理协议)就是这样的例子。

图8.1给出了典型的UDP客户-服务器程序函数调用。客户不与服务器建立连接,它只管用函数sendto给服务器发送数据报(sendto在下一节介绍),此函数要求目的地址(服务器)作为其参数。类似地,服务器不从客户接收连接,它只管调用函数recvfrom,等待来自某客户的数据到达。与数据报一起,recvfrom返回客户的协议地址,所以服务器可以发送响应给正确的客户。

图8.1所示为典型情况下发生的UDP客户-服务器交互的时间线图,我们可将此图与图4.1中典型的TCP交互进行比较。

本章我们介绍用于UDP套接口的新函数:recvfrom和sendto,并用UDP重写我们的回射客户-服务器程序。我们也介绍了函数connect在UDP套接口中的使用和异步错误的概念。

8.2 recvfrom和sendto函数

这两个函数类似于标准的read和write函数,但要求有三个附加参数。

```
#include <sys/socket.h>
ssize_t recvfrom (int sockfd, void *buff, size_t nbytes, int flags,
                  struct sockaddr *from, socklen_t *addrlen);
ssize_t sendto (int sockfd, const void *buff, size_t nbytes, int flags,
                const struct sockaddr *to, socklen_t addrlen);
                                    两者均返回:读写字节数——成功,-1——出错
```

前三个参数:sockfd、buff和nbytes等同于函数read和write的前三个参数:描述字、指向读入或写出缓冲区的指针和读写字节数。

第13章中当我们讨论函数recv、send、recvmsg和sendmsg时,我们再介绍参数flags,因为在我们简单的UDP客户-服务器程序例子中还不需要它们。现在我们总是将flags设置为0。

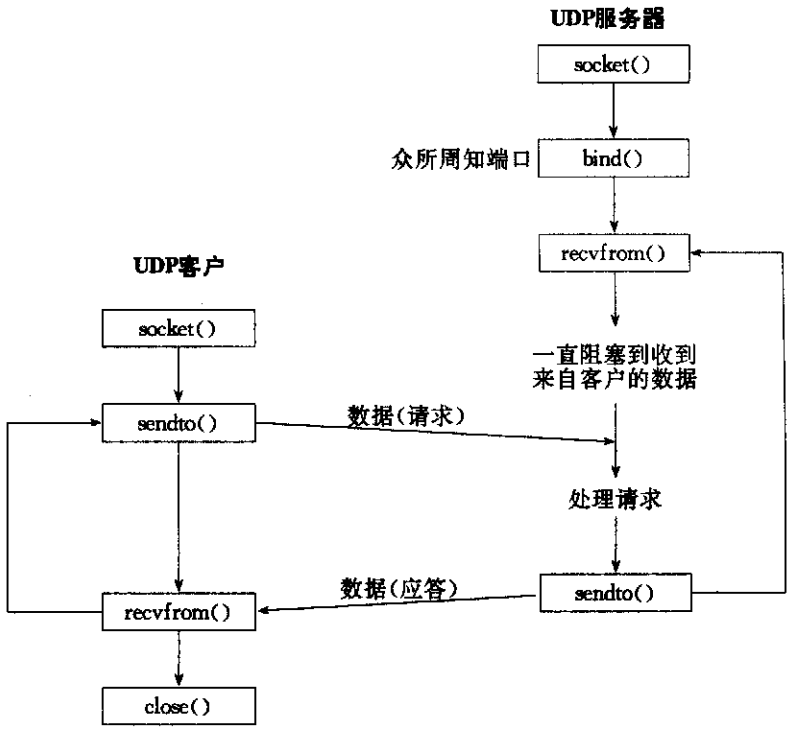


图 8.1　UDP 客户-服务器程序所用套接口函数

函数 sendto 的参数 to 是一个含有数据将发往的协议地址(例如,IP 地址和端口号)的套接口地址结构,它的大小由 addrlen 来指定。函数 recvfrom 用数据报发送者的协议地址装填由 from 所指的套接口地址结构,存储在此套接口地址结构中的字节数也以 addrlen 所指的整数返回给调用者。注意,sendto 的最后一个参数是一整数值,而 recvfrom 的最后一个参数是一指向整数值的指针(值-结果参数)。

recvfrom 的最后两个参数类似于 accept 的最后两个参数:返回时套接口地址结构的内容告诉我们是谁发送了数据报(UDP 情况下)或是谁发起了连接(TCP 情况下)。sendto 的最后两个参数类似于 connect 的最后两个参数:我们用数据报将发往(UDP 情况下)或与之建立连接(TCP 情况下)的协议地址来装填套接口地址结构。

两个函数都返回读写数据的长度作为函数值。在 recvfrom 的典型应用中,对于数据报协议,返回值是所接收的数据报中用户数据的总量。

写一个长度为 0 的数据报是可行的,在 UDP 情况下,这导致一个包含 IP 头部(一般说来,IPv4 的头部为 20 字节,IPv6 的头部为 40 字节)、8 字节 UDP 头部和没有数据的 IP 数据报。这也意味着对于数据报协议,recvfrom 返回 0 值也是可行的:它不表示对方已关闭了连接,这与 TCP 套接口上的 read 返回 0 的情况不同。由于 UDP 是无连接的,这就没有诸如关闭 UDP 连接之类的事情。

如果 recvfrom 的参数 from 是空指针,则相应的长度参数(addrlen)也必须是空指针,这表示我们并不关心发数据方的协议地址。

recvfrom 和 sendto 可用于 TCP,尽管一般来说没有理由这样做。

如13.9节所述，T/TCP即事务TCP(TCP for Transactions)就使用sendto。

8.3 UDP回射服务器程序：main函数

现在，我们用UDP重新编写第5章中简单的回射客户-服务器程序。我们的UDP客户和服务器程序依循图8.1中所示的函数调用流程，图8.2描述了所使用的函数，图8.3给出了服务器程序main函数。

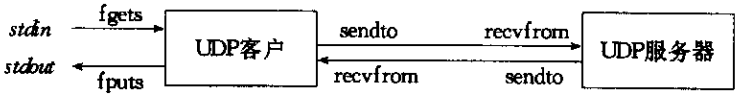


图8.2 使用UDP的简单回射客户-服务器

```
1 #include    "unp.h"
2 int
3 main(int argc,char * * argv)
4 {
5     int         sockfd;
6     struct sockaddr_in servaddr,cliaddr;
7     sockfd = Socket(AF_INET,SOCK_DGRAM,0);
8     bzero(&servaddr,sizeof(servaddr));
9     servaddr.sin_family = AF_INET;
10    servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
11    servaddr.sin_port = htons(SERV_PORT);
12    Bind(sockfd,(SA * ) &servaddr,sizeof(servaddr));
13    dg_echo(sockfd,(SA * ) &cliaddr,sizeof(cliaddr));
14 }
```

图8.3 UDP回射服务器程序[udpcliserv/ucpserv01.c]

创建UDP套接口，捆绑服务器的众所周知端口

第7～12行 通过指定函数socket的第二个参数为SOCK_DGRAM(IPv4协议中的数据报套接口)，我们创建一个UDP套接口。正如TCP服务器程序的例子，bind的IPv4地址被指定为INADDR_ANY，且服务器的众所周知端口是头文件<unp.h>中的常值SERV_PORT。

第13行 接着，调用函数dg_echo来进行服务器的处理。

8.4 UDP回射服务器程序：dg_echo函数

图8.4给出了函数dg_echo。

```
1 #include    "unp.h"
2 void
3 dg_echo(int sockfd,SA * pcliaddr,socklen_t clilen)
4 {
```

```
 5    int        n;
 6    socklen_t len;
 7    char       mesg[MAXLINE];
 8    for ( ; ; ) {
 9        len = clilen;
10        n = Recvfrom(sockfd,mesg,MAXLINE,0,pcliaddr,&len);
11        Sendto(sockfd,mesg,n,0,pcliaddr,len);
12    }
13 }
```

图 8.4　函数 dg_echo：在数据报套接口上回射行[lib/dg_echo.c]

读数据报并回射给发送者

第 8～12 行　此函数是一个简单的循环，它用 recvfrom 读下一个到达服务器端口的数据报，并用 sendto 将它发回。

尽管这个函数很简单，但也有许多细节问题需要考虑。首先，此函数从不终止，因为 UDP 是一个无连接协议，它没有像 TCP 中文件结束符之类的东西。

其次，该函数提供一个迭代服务器(iterative server)，而不是像 TCP 一样提供了一个并发服务器。没有对 fork 的调用，所以单一服务器进程就处理了所有客户。一般来说，大多数 TCP 服务器是并发的，而大多数 UDP 服务器是迭代的。

此套接口在 UDP 层中有隐含的排队动作。实际上，每个 UDP 套接口都有一个接收缓冲区，到达此套接口的每个数据报都进入此套接口接收缓冲区。当进程调用 recvfrom 时，缓冲区中的下一个数据报以 FIFO(先入先出)顺序返回给进程。这样，在进程可读套接口中排好队的数据报之前，如果有多个数据报到达套接口，那么到达的数据报仅仅加到套接口接收缓冲区中。但这个缓冲区的大小有一个限制，我们在 7.5 节中已经随 SO_RCVBUF 套接口选项一起讨论了这个大小及如何增大它。

图 8.5 总结了第 5 章中的 TCP 客户-服务器在两个客户与服务器建立连接时的情形。

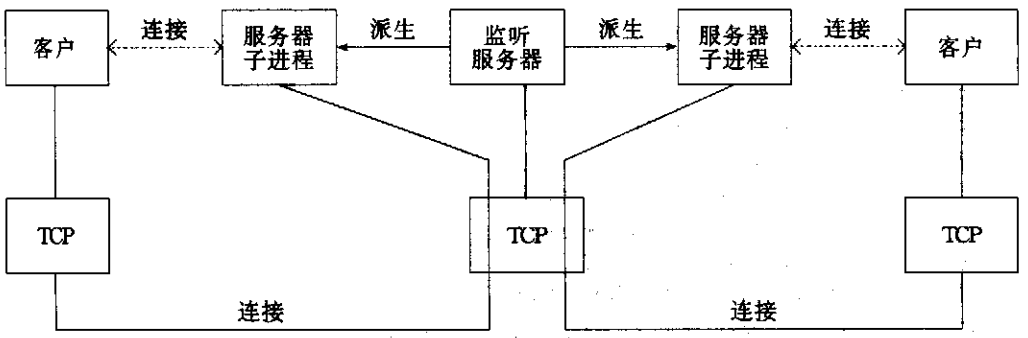


图 8.5　两个客户的 TCP 客户-服务器小结

有两个已连接套接口，服务器主机上的每一个已连接套接口都有自己的套接口接收缓冲区。

图 8.6 为两个客户发送数据报到 UDP 服务器的情形。

只有一个服务器进程，且它有单个套接口以接收所有到达的数据报并发回所有响应，此

套接口有一个接收缓冲区用来存放所到达的数据报。

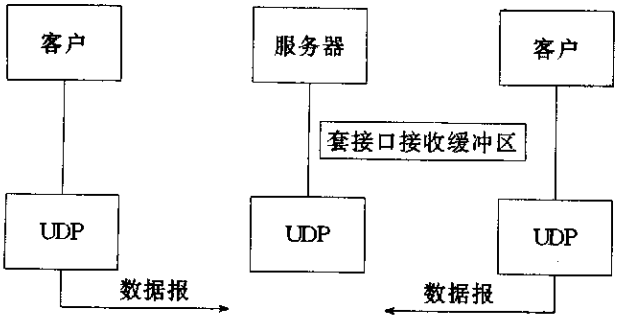


图 8.6 两个客户的 UDP 客户-服务器小结

图 8.3 中的 main 函数是协议相关的(它创建一个 AF_INET 协议的套接口,并分配、初始化一个 IPv4 套接口地址结构),但函数 dg_echo 是协议无关的。dg_echo 协议无关的理由是:调用者(在我们的例子中为 main 函数)必须分配一个正确大小的套接口地址结构,且指向此结构的指针和它的大小都应作为参数传递给 dg_echo。dg_echo 从来不深入到这样的协议相关结构中,它简单地传递一个指向该结构的指针给 recvfrom 和 sendto。recvfrom 用客户的 IP 地址和端口号装填此结构,且由于相同的指针(pcliaddr)又是作为目的地址传递给 sendto 的,这样数据报就回射给发此数据报的客户了。

8.5 UDP 回射客户程序:main 函数

UDP 客户程序 main 函数示于图 8.7。

```
1 #include    "unp.h"
2 int
3 main(int argc,char * * argv)
4 {
5     int         sockfd;
6     struct sockaddr_i servaddr;
7     if (argc ! = 2)
8         err_quit("usage:udpcli <IPaddress>");
9     bzero(&servaddr,sizeof(servaddr));
10    servaddr.sin_family = AF_INET;
11    servaddr.sin_port = htons(SERV_PORT);
12    Inet_pton(AF_INET,argv[1],&servaddr.sin_addr);
13    sockfd = Socket(AF_INET,SOCK_DGRAM,0);
14    dg_cli(stdin,sockfd,(SA * ) &servaddr,sizeof(servaddr));
15    exit(0);
16 }
```

图 8.7 UDP 回射客户程序[udpcliserv/udpcli01.c]

用服务器地址装填套接口地址结构

第 9～12 行　用服务器的 IP 地址和端口号来装填一个 IPv4 的套接口地址结构。此结构将传递给函数 dg_cli，指明数据报将发往何处。

第 13～14 行　创建一个 UDP 套接口，然后调用 dg_cli。

8.6　UDP 回射客户程序：dg_cli 函数

图 8.8 所示为函数 dg_cli，大部分客户的处理都由它执行。

```
1 #include    "unp.h"
2 void
3 dg_cli(FILE *fp,int sockfd,const SA *pservaddr,socklen_t servlen)
4 {
5     int     n;
6     char    sendline[MAXLINE],recvline[MAXLINE + 1];
7     while(Fgets(sendline,MAXLINE,fp) != NULL) {
8         Sendto(sockfd,sendline,strlen(sendline),0,pservaddr,servlen);
9         n = Recvfrom(sockfd,recvline,MAXLINE,0,NULL,NULL);
10        recvline[n] = 0;      /* null terminate */
11        Fputs(recvline,stdout);
12    }
13 }
```

图 8.8　函数 dg_cli：客户处理循环[lib/dg_cli.c]

第 7～12 行　客户处理循环中有四个步骤：用 fgets 从标准输入读一行，用 sendto 将此行发给服务器，用 recvfrom 读回服务器的回射，用 fputs 输出回射行到标准输出。

我们的客户不请求内核给套接口分配一临时端口。（对于 TCP 客户，我们说 connect 调用是分配发生的地方。）对于 UDP 套接口，如果进程首次调用 sendto 时还没有捆绑上一个本地端口，内核就在此时为套接口选择一个临时端口。跟 TCP 一样，客户可以显式地调用 bind，但很少这样做。

注意，对 recvfrom 的调用指定了空指针作为第五和第六个参数。这就通告内核，我们对谁发响应并不关心。这样做有一个风险，即无论是相同主机或不同主机上的进程，都有可能给客户的 IP 地址和端口发送数据报，这些数据报被客户所读并被认为就是服务器的应答。我们在 8.8 节中再讨论此问题。

与服务器函数 dg_echo 一样，客户函数 dg_cli 也是协议无关的，但客户 main 函数是协议相关的。main 函数分配并初始化某个协议类型的套接口地址结构，并将指向此结构的指针及它的大小传递给 dg_cli。

8.7　数据报的丢失

我们的 UDP 客户-服务器例子是不可靠的。如果一个客户数据报丢失（譬如说，被客户与服务器间的某路由器丢弃），客户将永远阻塞于 dg_cli 中对 recvfrom 的调用，等待一个永远不会到达的服务器应答。与此相似，如果客户数据报到达服务器，但服务器的应答丢失了，

客户也将永远阻塞于 recvfrom 的调用。上述问题唯一的解决方法就是给客户 recvfrom 调用设置一个超时,这一点我们在 13.2 节中再作讨论。

仅仅是为调用 recvfrom 而设置超时并不是一个完整的办法。例如,如果我们超时了,我们无法辨别超时原因是数据报没有到达服务器,还是服务器的应答没有回到客户。如果客户的请求有点像"从账户 A 往账户 B 传送固定数目的钱"(而不是我们的简单回射服务器例子),则请求丢失和应答丢失是极不相同的。在 20.5 节,我们讨论如何给 UDP 客户-服务器程序增加可靠性。

8.8 验证接收到的响应

在 8.6 节结尾我们提到,知道客户临时端口号的任何进程都可往客户发送数据报,且这些数据报会与正常的服务器应答混淆。我们解决此问题的办法是:修改图 8.8 中对 recvfrom 的调用以返回发送响应者的 IP 地址和端口号,并忽略不是来自我们的数据报所发往服务器的任何数据报。然而,这也有一些缺陷,下面就将看到。

首先,我们修改客户程序 main 函数(图 8.7)以使用标准回射服务器(图 2.13),我们用赋值语句

```
servaddr.sin_port = htons(7);
```

代替

```
servaddr.sin_port = htons(SERV_PORT);
```

这样,我们的客户就可以使用任何运行标准回射服务器的主机。

然后,我们重写函数 dg_cli 以分配另一个套接口地址结构保存从 recvfrom 返回的结构,如图 8.9 所示。

分配另一个套接口地址结构

第 9 行 我们调用 malloc 来分配另一个套接口地址结构。注意,函数 dg_cli 仍然是协议无关的,由于不关心所处理套接口地址结构的类型,我们仅将其大小用于调用 malloc。

比较返回地址

第 12~18 行 在 recvfrom 的调用中,我们通知内核返回数据报发送者的地址。我们首先比较 recvfrom 从值-结果参数中返回的长度,然后用 memcmp 比较套接口地址结构本身。

如果服务器仅在一台具有单个 IP 地址的主机上运行,则我们的新版客户工作得很好,但若服务器是多宿的,此程序就有可能失败。我们在有两个接口和两个 IP 地址的主机 bsdi 上运行此程序。

```
solaris % host bsdi
bsdi.kohala.com has address 206.62.226.35
bsdi.kohala.com has address 206.62.226.66
solaris % udpcli02 206.62.226.66
hello
reply from 206.62.226.35.7 (ignored)
goodbye
reply from 206.62.226.35.7 (ignored)
```

```
1 #include        "unp.h"
2 void
3 dg_cli(FILE  * fp,int sockfd,const SA  * pservaddr,socklen_t servlen)
4 {
5     int     n;
6     char    sendline[MAXLINE],recvline[MAXLINE + 1];
7     socklen_t len;
8     struct sockaddr  * preply_addr;
9     preply_addr = Malloc(servlen);
10    while(Fgets(sendline,MAXLINE, fp) ! = NULL) {
11        Sendto(sockfd,sendline,strlen(sendline),0,pservaddr,servlen);
12        len = servlen;
13        n = Recvfrom(sockfd,recvline,MAXLINE,0,preply_addr,&len);
14        if(len ! = servlen || memcmp(pservaddr,preply_addr,len) ! = 0) {
15            printf("reply from %s (ignored)\n";
16                Sock_ntop(preply_addr,len);
17            continue;
18        }
19        recvline[n] = 0;      /* null terminate */
20        Fputs(recvline,stdout);
21    }
22 }
```

图 8.9　验证返回的套接口地址的函数 dg_cli 版本[udpcliserv/dgcliaddr.c]

从图 1.16 我们知道，我们指定的 IP 地址不共享与客户主机相同的子网。

> 一般来说，这是允许的。大多数 IP 实现接受目的地址为本主机任一 IP 地址的数据报，而不管数据报到达的接口（TCPv2 第 217～219 页）。RFC 1122 [Braden 1989]称此为弱端系统模型(weak end system model)。如果系统要实现此 RFC 中所称的强端系统模型(strong end system model)，它将只接受到达接口与目的地址一致的数据报。

recvfrom 返回的 IP 地址(UDP 数据报的源 IP 地址)不是我们所发送数据报的目的 IP 地址。当服务器发送应答时，目的 IP 地址是 206.62.226.33。bsdi 主机内核中的路由函数选择 206.62.226.35 作为外出接口。由于服务器没有捆绑实际 IP 地址到其套接口上(服务器已捆绑通配 IP 地址到套接口上了，这可以在 bsdi 主机上运行 netstat 来验证)，因此内核要给 IP 数据报选择源地址，通常选择为外出接口的主 IP 地址(TCPv2 第 232～233 页)。还有，由于它是接口的主 IP 地址，如果我们发送数据报到接口的某个非主 IP 地址(即一个别名)，也将导致图 8.9 中的测试失败。

一个解决办法是，给定由 recvfrom 返回的 IP 地址后，由客户在 DNS(第 9 章)中查找服务器主机的名字来验证相应主机的域名而不是它的 IP 地址。另一个解决办法是，由 UDP 服务器给服务器主机上配置的每个 IP 地址创建一个套接口，捆绑相应的 IP 地址到此套接口，然后在所有这些套接口上使用 select(等待其中任一个成为可读)，再从可读的套接口作应答。由于用于应答的套接口捆绑了客户请求的目的 IP 地址(否则该数据报不会投送到此套接口上)，这就保证应答的源地址与请求的目的地址相同。我们在 19.11 节和 20.6 节将给出这样的例子。

在一个多宿的Solaris系统上，服务器应答的源IP地址是客户请求的目的IP地址。本节所描述的情况是针对源自Berkeley的实现的，它基于外出接口选择源IP地址。不过在Solaris 2.5.1上看来，服务器应答的源IP地址是外出接口的最后一个已配置IP地址。

8.9 服务器进程未运行

我们下一个要检查的情况是在不启动服务器的情况下启动客户，看看这将会怎样。我们在客户上键入一行，什么也看不到，客户永远阻塞于它对recvfrom的调用，等待一个永不到来的服务器应答。但这是一个很好的例子，它要求我们对底层协议有更多的了解以理解网络应用进程发生了什么事情。

首先，我们在主机bsdi上启动tcpdump，然后在相同主机上启动客户，指定主机solaris为服务器主机。接着，我们键入一行，但此行没有被回射。

```
bsdi % udpcli01 206.62.226.33
hello,world          我们键入这一行
                     但没有任何东西回射回来
```

图8.10给出了tcpdump的输出。

```
1  0.0                  arp who-has solaris tell bsdi
2  0.002526 (0.0025)    arp reply solaris is-at 8:0:20:78:e3:e3
3  0.002932 (0.0004)    bsdi.1105 > solaris.9877: udp 13
4  0.006932 (0.0040)    solaris > bsdi: icmp:solaris udp port 9877 unreachable
```

图8.10 当服务器主机上未启动服务器进程时tcpdump的输出

首先我们注意到，在客户主机可以往服务器主机发UDP数据报之前，要求有ARP请求和应答。（我们将此交换放在输出中是为了强调在IP数据报可发往本地网络上另一台主机或路由器之前，ARP请求-应答的潜在可能。）

在第三行，我们看到数据报发出，但服务器主机在第四行响应以“端口不可达”ICMP消息。（长度13是12个字符加回车符。）然而，此ICMP错误不返回给客户进程，原因我们不久将讨论到。客户永远阻塞于图8.8中对recvfrom的调用。我们也注意到ICMPv6有“端口不可达”错误类型，这类似于ICMPv4（见图A.15和图A.16），因此，这里讨论的结果IPv6也类似。

我们将此ICMP错误称为异步错误（asynchronous error）。该错误由sendto引起，但sendto本身却返回成功。回忆一下，在2.9节中说过，从UDP输出操作返回成功仅仅表示在接口输出队列上为IP数据报留出空间。ICMP错误直到很晚才返回（在图8.10是4ms以后）。这就是为什么称其为异步的原因。

一个基本规则是：除非套接口已连接，否则异步错误是不给UDP套接口返回的。在8.11节中，我们讨论如何给UDP套接口调用connect。为什么套接口最初实现时做此设计决策是很难明白的。（实现内涵在TCPv2第748～749页讨论。）考虑UDP客户在单个UDP套接口上连续发送三个数据报给三个不同的服务器（即三个不同的IP地址），然后进入一个调用recvfrom的循环以读应答。有两个数据报被正确递送（也就是说，三个主机中有两个主机

在运行服务器),但第三个主机没有运行服务器,这第三个主机就以一个 ICMP"端口不可达"错误来响应。此 ICMP 出错消息包含有引起错误的数据报的 IP 头部和 UDP 头部。(ICMPv4 和 ICMPv6 出错消息总是包含 IP 头部和所有的 UDP 头部或部分 TCP 头部,以便其接收者确定是哪个套接口引起的错误,如图 25.20 和图 25.21 所示。)发送三个数据报的客户需要知道引起错误的数据报的目的地址以区分是三个数据报中的哪一个引起了错误。但内核如何将此信息返回给进程呢? recvfrom 可以返回的仅有信息是 errno 值,它没有办法返回错误数据报的目的 IP 地址和目的 UDP 端口号。因此做出决定:仅在进程已将 UDP 套接口连接到确切的对方后,这些异步错误才返回给进程。

只要 SO_BSDCOMPAT 套接口选项没有打开,Linux 甚至对未连接的套接口都返回大多数 ICMP 目的地不可达错误。除了代码 0、1、4、5、11 和 12,图 A.15 中的所有目的地不可达错误均被返回。

XTI 接口提供了将这个附加信息返回给进程的方法:t_rcvuderr 可以返回此信息(31.4 节)。遗憾的是,许多 XTI 实现不返回此信息。

在 25.7 节,我们回到 UDP 套接口异步错误的这个问题,并给出了一个使用我们自己的守护进程获取未连接套接口上这些错误的简便方法。

8.10　UDP 程序例子小结

图 8.11 以圆点的形式给出了在客户发送 UDP 数据报时必须指定或选择的四个值。

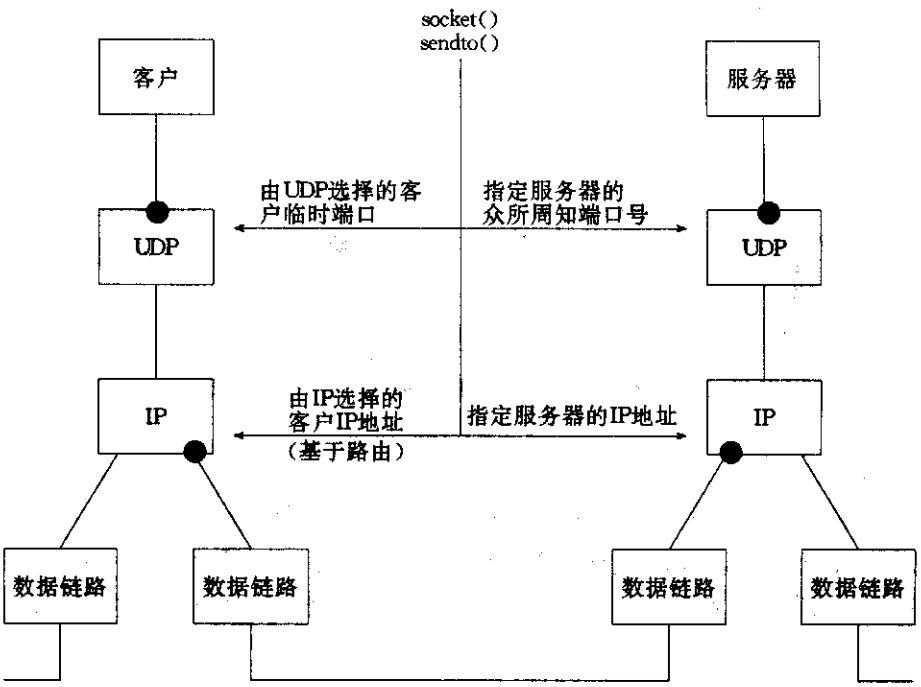


图 8.11　从客户角度小结 UDP 客户-服务器

客户必须给 sendto 调用指明服务器的 IP 地址和端口号。虽然我们提到过，如果愿意，客户可以调用 bind，但一般来说，客户的 IP 地址和端口号都是由内核自动选择的。我们也提到过，如果客户的这两个值由内核选择，客户的临时端口是在第一次调用 sendto 时一次性选定的，不能改变。然而，假设客户不捆绑特定的 IP 地址到套接口上，客户的 IP 地址是可以随客户发送的每个 UDP 数据报修改的。原因如图 8.11：如果客户主机是多宿的，客户将在两个目的地中选一个，一个由左边的数据链路外出，另一个由右边的数据链路外出。在此最坏情况下，客户的 IP 地址由于是由内核基于外出数据链路选择的，它可以随每个数据报而改变。

如果客户捆绑了一个 IP 地址到其套接口上，但内核决定外出数据报必须从另一个数据链路发出，将会怎样？在这种情况下，IP 数据报将包含一个不同于外出链路 IP 地址的源 IP 地址(见习题8.6)。

图 8.12 给出了同样的四个值，但是是从服务器的角度出发的。

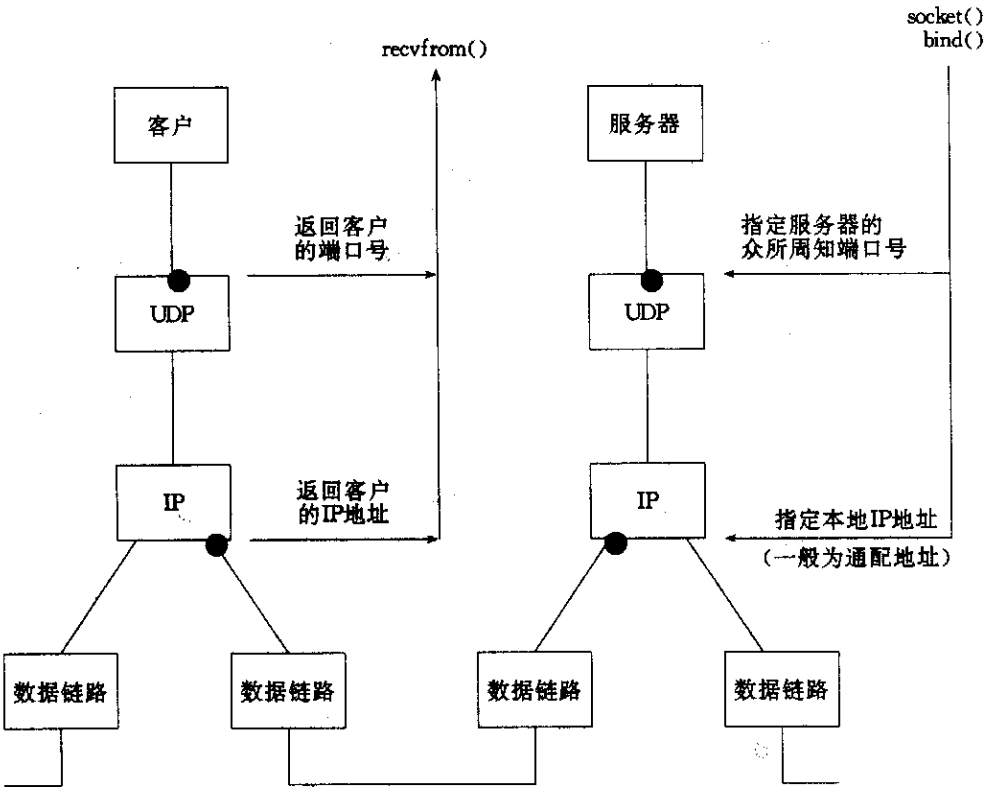


图 8.12 从服务器角度小结 UDP 客户-服务器

服务器可能想从到达的 IP 数据报上取得四条信息：源 IP 地址、目的 IP 地址、源端口号和目的端口号。图 8.13 给出了从 TCP 服务器或 UDP 服务器返回这些信息的函数调用。

来自客户的 IP 数据报	TCP 服务器	UDP 服务器
源 IP 地址	accept	recvfrom
源端口号	accept	recvfrom
目的 IP 地址	getsockname	recvmsg
目的端口号	getsockname	getsockname

图 8.13　服务器可从到达的 IP 数据报中获取的信息

TCP 服务器总是能便捷地访问已连接套接口的所有这四条信息，而且这四个值在连接的整个生命期内保持不变。然而对于 UDP 套接口，目的 IP 地址只能由为 IPv4 设置 IP_RECVDSTADDR 套接口选项或为 IPv6 设置 IPV6_PKTINFO 套接口选项，然后调用 recvmsg 而不是 recvfrom 来得到。由于 UDP 是无连接的，因此目的 IP 地址可随发送到服务器的每个数据报而改变。UDP 服务器也可接收目的地址为服务器主机的某个的广播地址或多播地址的数据报，这我们在第 18 章和第 19 章再做讨论。我们在 20.2 节讨论函数 recvmsg 以后，展示如何确定 UDP 数据报的目的地址。

8.11　UDP 的 connect 函数

在 8.9 节结尾我们提到，除非套接口已连接，否则异步错误是不会返回到 UDP 套接口的。实际上，我们可以给 UDP 套接口调用 connect(4.3 节)，但这样做的结果却与 TCP 连接毫不相同：没有三路握手过程。内核只是记录对方的 IP 地址和端口号，它们包含在传递给 connect 的套接口地址结构中，并立即返回给调用进程。

> 给 connect 函数重载(overload)UDP 套接口的这种能力容易让人混淆。如果使用约定：sockname 是本地协议地址，而 peername 是远程协议地址，则更好的名字可能是 setpeername。类似地，函数 bind 的更好名字也许是 setsockname。

有了这个能力后，我们必须区分：

- 未连接 UDP 套接口(unconnected UDP socket)，当我们创建 UDP 套接口时，缺省为此；
- 已连接 UDP 套接口(connected UDP socket)，对 UDP 套接口调用 connect 的结果。

对于已连接 UDP 套接口，与缺省的未连接 UDP 套接口相比，发生了三个变化：

1. 我们再也不能给输出操作指定目的 IP 地址和端口号。也就是说，我们不使用 sendto，而使用 write 或 send。写到已连接 UDP 套接口上的任何东西都自动发送到由 connect 所指定的协议地址(例如 IP 地址和端口号)。

   > 与 TCP 相似，我们可以给已连接 UDP 套接口调用 sendto，但不能指定目的地址。sendto 的第五个参数(指向套接口地址结构的指针)必须为空指针，第六个参数(套接口地址结构的大小)应为 0。Posix.1g 规定，当第五个参数是空指针时，就不再考虑第六个参数的取值情况。

2. 我们不用 recvfrom 而用 read 或 recv。在已连接 UDP 套接口上由内核为输入操作返

回的唯一数据报是那些来自 connect 所指定协议地址的数据报。目的地址为已连接 UDP 套接口的本地协议地址(例如 IP 地址和端口号),但不是从 connect 所指定套接口协议地址到达的数据报,不传递给已连接套接口。这就限制了已连接 UDP 套接口能且只能与一个对方交换数据报。

严格地说,一个已连接 UDP 套接口仅与一个 IP 地址交换数据报,因为 connect 到多播或广播地址是可能的。

3. 异步错误由已连接 UDP 套接口返回给进程,由此推断,未连接 UDP 套接口不接收任何异步错误,这在前面我们已描述过。

图 8.14 就 4.4BSD 总结了上面列表中第一点。

套接口类型	write 或 send	不指定目的地址的 sendto	指定目的地址的 sendto
TCP 套接口	可以	可以	EISCONN
UDP 套接口,已连接	可以	可以	EISCONN
UDP 套接口,未连接	EDESTADDRREQ	EDESTADDRREQ	可以

图 8.14 TCP 和 UDP 套接口:可指定目的协议地址吗?

Posix.1g 指出,在未连接 UDP 套接口上不指定目的地址的输出操作应返回 ENOTCONN,而不是 EDESTADDRREQ。

Solaris 2.5 允许 sendto 给已连接 UDP 套接口指定目的地址,而 Posix.1g 规定应返回 EISCONN。

图 8.15 就我们给已连接 UDP 套接口所归纳的三点做了一个小结。

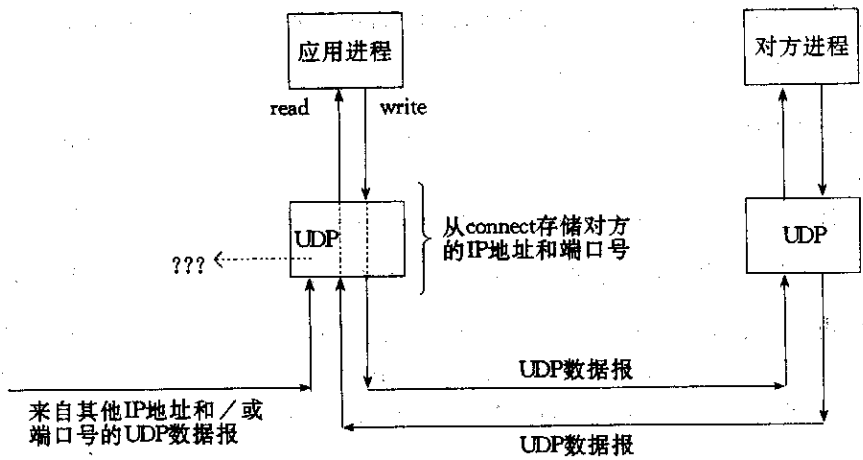


图 8.15 已连接 UDP 套接口

应用进程调用 connect,指定对方的 IP 地址和端口号,然后用 read 和 write 来与对方交换数据。

从任何其他 IP 地址或端口(图 8.15 中我们用"???"表示)到达的数据报不递送给已连接套接口,因为要么源 IP 地址要么源 UDP 端口不与所连接的套接口协议地址相匹配。这些

数据报可能递送给同一主机上的其他 UDP 套接口。如果到达的数据报没有其他匹配的套接口，UDP 将丢弃它并生成一个 ICMP 端口不可达错误。

作为小结，仅在进程用 UDP 套接口与确定的唯一对方进行通信时，UDP 客户或服务器才可以调用 connect。一般来说，都是 UDP 客户调用 connect，但也有 UDP 服务器与单个客户长时间通信的应用程序(如 TFTP)，这种情况下，客户和服务器都可调用 connect。

DNS 提供了另一个例子，如图 8.16 所示。

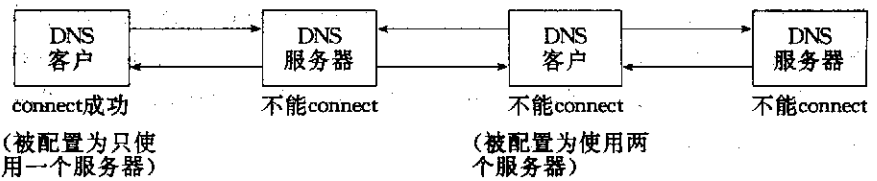


图 8.16　DNS 客户、服务器与函数 connect 的例子

通过在文件/etc/resolv.conf 中列出服务器的 IP 地址，DNS 客户可被配置为使用一个或多个服务器。如果列出单个服务器(图中最左边的方框)，客户可以调用 connect，但如果列出多个服务器(图中从右边数第二个方框)，客户就不能调用 connect。而且，DNS 服务器通常是处理任何客户的请求，所以服务器不能调用 connect。

给一个 UDP 套接口多次调用 connect

对于已连接 UDP 套接口，进程可给那个套接口再次调用 connect 以达到下面两个目的之一：

- 指定新的 IP 地址和端口号；
- 断开套接口。

第一种情况即给已连接 UDP 套接口指定新的对方，与 TCP 套接口中 connect 的使用有所不同：只可给 TCP 套接口调用一次 connect。

为了断开已 UDP 套接口连接，我们调用 connect，但设置套接口地址结构的地址族(对 IPv4 为 sin_family，对 IPv6 为 sin6_family)为 AF_UNSPEC。这可能返回一个 EAFNOSUPPORT 错误(TCPv2 第 736 页)，但没有关系。使得套接口断开连接的是在已连接 UDP 套接口上调用 connect 的进程(TCPv2 第 787～788 页)。

> connect 的 BSD 手册页面传统上说："数据报套接口可通过连接到无效地址，譬如说空地址来断开连接。"但很不幸，手册页面中根本没有定义什么是"空地址"，且没有提到会导致一个错误(但这个错误并没有关系)。Posix.1g 明确指出，地址族必须设置为 AF_UNSPEC，但然后就胡说 connect 的这种调用可以返回，也可以不返回 EAFNOSUPPORT 错误。

性能

当应用进程在未连接 UDP 套接口上调用 sendto 时，源自 Berkeley 的内核暂时连接套接口，发送数据报，然后断开套接口连接(TCPv2 第 762～763 页)。在未连接 UDP 套接口上给两个数据报调用函数 sendto 导致内核执行下列六步：

- 连接套接口；
- 输出第一个数据报；
- 断开套接口连接；
- 连接套接口；
- 输出第二个数据报；
- 断开套接口连接。

另一个考虑是搜索路由表的次数。第一个临时连接为目的IP地址搜索路由表并高速缓存此信息，第二个临时连接注意到目的地址等于已高速缓存的路由表信息的目标(我们假设两个sendto调用有相同的目的地址)，就不用再查找路由表了(TCPv2第737～738页)。

当应用进程知道自己要给同一目的地址发多个数据报时，显式连接套接口效率更高。调用connect，然后调用两次write，导致内核执行如下步骤：

- 连接套接口；
- 输出第一个数据报；
- 输出第二个数据报。

在这种情况下，内核只拷贝一次含有目的IP地址和端口号的套接口地址结构，而当调用两次sendto时，要拷贝两次。[Partridge和Pink 1993]注意到，临时连接未连接的UDP套接口将花去每个UDP传输三分之一的消耗。

8.12 dg_cli函数(修订版)

现在，我们回到图8.8中的函数dg_cli，并重写它以调用connect。图8.17所示为新函数。

```
1 #include    "unp.h"
2 void
3 dg_cli(FILE *fp,int sockfd,const SA *pservaddr,socklen_ servlen)
4 {
5     int     n;
6     char    sendline[MAXLINE],recvline[MAXLINE + 1];
7     Connect(sockfd,(SA *) pservaddr,servlen);
8     while(Fgets(sendline,MAXLINE,fp) != NULL) {
9         Write(sockfd,sendline,strlen(sendline));
10        n = Read(sockfd,recvline,MAXLINE);
11        recvline[n] = 0;        /* null terminate */
12        Fputs(recvline,stdout);
13    }
14 }
```

图8.17 调用connect的dg_cli函数[udpcliserv/dgcliconnect.c]

所做的修改是调用 connect，并以 read 和 write 调用代替 sendto 和 recvfrom 调用。由于此函数不看传递给 connect 的套接口地址结构的内容，它仍然是协议无关的。图 8.7 中的客户程序 main 函数保持不变。

如果我们在主机 bsdi 上运行此程序，指定主机 solaris 的 IP 地址(它不在端口 9877 上运行我们的服务器)，就有下列输出：

```
bsdi % udpcli04 206.62.226.33
hello, world
read error:Connection refused
```

我们注意到的第一点是，当启动客户进程时接收不到错误，错误仅在我们发了第一个数据报给服务器后才会出现。正是发送此数据报引发了来自服务器主机的 ICMP 错误。但当 TCP 客户调用 connect，指定一个不在运行服务器进程的服务器主机时，connect 返回错误。这是因为 connect 的调用导致 TCP 三路握手的第一个分组发出，且正是这个分组引发了 TCP 服务器的 RST(4.3 节)。

图 8.18 给出了 tcpdump 的输出。

```
bsdi % tcpdump
1  0.0                  bsdi.1318 > solaris.9877: udp 13
2  0.000628 (0.0006)    solaris > bsdi: icmp: solaris udp port 9877 unreachable
```

图 8.18　当运行图 8.17 中程序时 tcpdump 的输出

在图 A.15 中，我们还看到，此 ICMP 错误由内核将其映射成 ECONNREFUSED 错误，它相应于函数 err_sys 输出的消息串："Connection refused(连接被拒绝)。"

遗憾的是，并非所有内核都能像本节的示例那样把 ICMP 消息返送给已连接的 UDP 套接口。一般来说，源自 Berkeley 的内核返回错误，而系统 V 内核则不。例如，若我们在 Solaris 2.4 主机上运行同一客户程序并连接到没有运行服务器的主机上，我们可以用 tcpdump 观察到并验证服务器主机返回了 ICMP 端口不可达错，但客户的 read 调用不返回。此缺陷在 Solaris 2.5 中已修复。UnixWare 不返回错误，而 AIX、Digital Unix、HP-UX 和 Linux 都返回错误。

8.13　UDP 缺乏流量控制

现在，我们检查无任何流量控制的 UDP 对数据报传输的影响。首先，将我们的函数 dg_cli 修改为发送固定数目的数据报，它不再从标准输入读。图 8.19 所示为新代码，此函数写 2000 个 1400 字节大小的 UDP 数据报给服务器。

```
1 #include    "unp.h"
2 #define DNG     2000        /* #datagrams to send */
3 #define DGLEN   1400        /* length of each datagram */
4 void
5 dg_cli(FILE *fp,int sockfd,const SA *pservaddr,socklen_t servlen)
6 {
7     int     i;
8     char    sendline[MAXLINE];
```

```
 9      for (i = 0; i < NDG; i++) {
10          Sendto(sockfd,sendline,DGLEN,0,pservaddr,servlen);
11      }
12  }
```

图 8.19 写固定数目的数据报到服务器的 dg_cli 函数[udpcliserv/dgcliloop1.c]

然后，我们修改服务器程序以接收数据报并对接收数目计数。此服务器不再将数据报回射给客户，图 8.20 给出了新的函数 dg_echo。当我们用终端中断键(SIGINT)终止服务器时，它输出所接收到数据报的数目并终止。

```
 1 #include    "unp.h"
 2 static void recvfrom_int(int);
 3 static int count;
 4 void
 5 dg_echo(int sockfd,SA *pcliaddr,socklen_t clilen)
 6 {
 7     socklen_t len;
 8     char      mesg[MAXLINE];
 9     Signal(SIGINT, recvfrom_int);
10     for( ; ; ) {
11         len = clilen;
12         Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);
13         count++;
14     }
15 }
16 static void
17 recvfrom_int(int signo)
18 {
19     printf("\nreceived %d datagrams\n",count);
20     exit(0);
21 }
```

图 8.20 对接收到数据报进行计数的 dg_echo 函数[udpcliserv/dgecholoop1.c]

现在，在主机 bsdi 上运行服务器，它是一台慢速 80386 机器，在快得多的 Sparcstation 4 上运行客户。另外，在服务器上运行 netstat -s，在试验前后均运行，并统计输出，以告诉我们丢失了多少数据报。图 8.21 所示为服务器上的输出。

```
bsdi % netstat -s | tail
udp:    80300 datagrams received
        0 with incomplete header
        0 with bad data length field
        0 with bad checksum
        12 dropped due to no socket
        77725 broadcast/multicast datagrams dropped due to no socket
        1970 dropped due to full socket buffers
        593 delivered
        70592 datagrams output
bsdi % udpserv06                启动我们的服务器
                                再在上面运行客户
```

```
^?                          客户运行完毕后敲入中断键
received 82 datagrams
bsdi % netstat -s | tail
udp:    82294 datagrams received
        0 with incomplete header
        0 with bad data length field
        0 with bad checksum
        12 dropped due to no socket
        77725 broadcast/multicast datagrams dropped due to no socket
        3882 dropped due to full socket buffers
        675 delivered
        70592 datagrams output
```

图 8.21　服务器主机上的输出

客户发出 2000 个数据报，但服务器仅收到其中的 82 个，丢失率为 96%。对服务器或客户来说，并没有给出任何指示说这些数据报已丢失。这证实了我们前面说的话，UDP 没有流量控制，它是不可靠的。同时也表明，UDP 发送方比 UDP 接收方运行速度快是造成数据报丢失的一个重要原因。

如果我们看一看 netstat 的输出，服务器主机（而不是服务器本身）接收到的数据报总数是 1994（82294－80300）。有 6 个数据报未被接口接收到，可能是因为接口的缓冲区满，也可能是已被发送主机丢弃。“dropped due to full socket buffers（因套接口缓冲区满而丢弃）”计数器的值表示有多少数据报已被 UDP 接收，但因接收套接口的接收队列满而丢弃（TCPv2 第 775 页）。此值为 1912（3882－1970），它加上由应用进程输出的计数值（82）后等于主机接收到的 1994 个数据报。遗憾的是，因套接口缓冲区满而丢弃的 netstat 计数值是全系统范围的值，没有办法确定影响到具体哪些应用进程（如哪些 UDP 端口）。

> 注意，此特定主机上所有接收到的 UDP 数据报中 97%（77725÷80300）是广播或多播数据报，它们因没有任何应用进程的套接口绑定目的端口而丢弃，在第 18 章中讨论广播时，我们再接着讨论此现象。

此例中，由服务器接收的数据报的数目是不确定的，它依赖于许多因素，如网络负载、客户主机的处理负载以及服务器主机的处理负载。如果我们再运行此例子五次，则接收数据报的计数分别是 37、108、30、108 和 114。

如果我们运行相同的客户和服务器，但这一次让客户在慢速 80386 上运行，服务器在快速 Sparcstation 上运行，则没有数据报丢失。

```
solaris % udpserv06
^?                          客户运行完毕后敲入中断键
received 2000 datagrams
```

如果我们在 Solaris 下运行 netstat -s，输出格式与图 8.21 中经典的 Berkeley 输出格式有所不同，Solaris 格式模仿 SNMP 计数器。SNMP 是简单网络管理协议，在 TCPv1 第 25 章讲述。netstat 的计数器 udpInDatagrams：传递给用户进程的 UDP 数据报个数，在试验前是 139，在试验后是 2139，正好为 2000 个数据报。计数器 udpInOverflows（它不是一个正式的 SNMP 计数器）记录着因接收套接口的接收队列没有空间而丢弃的 UDP 数据报数目，它的值在试验前后均为 0，正如我们所期望的那样。

UDP套接口接收缓冲区

由UDP给特定套接口排队的UDP数据报数目受限于套接口接收缓冲区的大小。我们可以用SO_RCVBUF套接口选项改变此值，如7.5节所述。在BSD/OS下UDP套接口接收缓冲区的缺省大小为41 600字节，即29个1400字节数据报的空间。如果我们增大套接口接收缓冲区的大小，则服务器有望接收更多的数据报。图8.22所示为对图8.20中函数dg_echo的修改，它设置套接口接收缓冲区为240K字节。如果我们在80386上运行服务器，在Sparcstation 4上运行客户，做5次试验，则接收的数据报计数分别为115、168、179、145和133，这比前面用缺省套接口接收缓冲区的例子稍有改善，但也并不能从根本上解决问题。

```
1 #include    "unp.h"
2 static void recvfrom_int(int);
3 static int count;

4 void
5 dg_echo(int sockfd,SA  * pcliaddr,socklen_t clilen)
6 {
7     int     n;
8     socklen_t len;
9     char    mesg[MAXLINE];

10    Signal(SIGINT,recvfrom_int);

11    n = 240 * 1024;
12    Setsockopt(sockfd,SOL_SOCKET,SO_RCVBUF, &n, sizeof(n));

13    for ( ; ; ) {
14        len = clilen;
15        Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

16        count++;
17    }
18 }

19 static void
20 recvfrom_int(int signo)
21 {
22    printf("\nreceived %datagrams\n", count);
23    exit(0);
24 }
```

图8.22 增大套接口接收队列大小的dg_echo函数[udpcliserv/dgecholoop2.c]

在图8.22中，为什么我们将接收套接口缓冲区大小设为240×1024字节？在BSD/OS 2.1中，套接口接收缓冲区的最大大小缺省为262 144字节(256×1024)，但由于缓冲区分配机制(见TCPv2第2章)的影响，真实的限制是246 723字节。许多基于4.3BSD的早期系统限制此套接口缓冲区的大小为52 000字节左右。

8.14　UDP 中的外出接口的确定

已连接 UDP 套接口还可用来确定用于特定目标的外出接口，这是由于函数 connect 被应用到 UDP 套接口时的副作用：内核选择本地 IP 地址（假设进程并没有调用 bind 以明确地指派它）。这个本地 IP 地址是通过给目的 IP 地址搜索路由表，然后使用结果接口的主 IP 地址而选定的。

图 8.23 给出了一个简单的 UDP 程序，它连接到指定的 IP 地址并调用 getsockname，输出本地 IP 地址和端口号。

```
1 #include    "unp.h"
2 int
3 main(int argc,char * * argv)
4 {
5     int     sockfd;
6     socklen_t len;
7     struct sockaddr_in cliaddr, servaddr;
8     if (argc ! = 2);
9         err_quit("usage: udpcli <IPaddress>");
10    sockfd = Socket(AF_INET,SOCK_DGRAM, 0);
11    bzero(&servaddr, sizeof(servaddr));
12    servaddr.sin_family = AF_INET;
13    servaddr.sin_port = htons(SERV_PORT);
14    Inet_pton(AF_INET,argv[1],&servaddr.sin_addr);
15    Connect(sockfd, (SA * ) &servaddr, sizeof(servaddr));
16    len = sizeof(cliaddr);
17    Getsockname(sockfd, (SA * ) &cliaddr, &len);
18    printf("local address %s\n", Sock_ntop((SA * ) &cliaddr, len));
19    exit(0);
20}
```

图 8.23　使用 connect 来确定输出接口的 UDP 程序[udpcliserv/udpcli09.c]

如果我们在多宿主机 bsdi 上运行此程序，则有下面的输出：

```
bsdi % udpcli09 206.62.226.42
local address 206.62.226.35.1331
bsdi % udpcli09 206.62.226.65
local address 206.62.226.66.1332
bsdi % udpcli09 127.0.0.1
local address 127.0.0.1.1335
```

从图 1.16 我们知道，我们运行此程序的前两次，命令行参数是在不同以太网上的 IP 地址，因此内核将本地 IP 地址指派成相应以太网接口的主 IP 地址。也就是说，.42 主机是在顶层以太网上，所以外出接口地址为.35，.65 主机是在稍低层以太网上，所以外出接口地址为.66。在 UDP 套接口上调用 connect 不给对方主机发任何信息，它完全是一个本地操作，只保存对方的 IP 地址和端口。我们还看到，在未绑定 UDP 套接口上调用 connect 也给套接

口指派一个临时端口。

遗憾的是，这项技术并非对所有实现都有效，尤其是SVR4内核。例如，它对HP-UX、Solaris 2.5和UnixWare无效，但对AIX、Digital Unix、Linux和Sloaris 2.6都有效。

8.15 使用select函数的TCP和UDP回射服务器程序

现在，我们将第5章中的并发TCP回射服务器程序与本章中的迭代UDP回射服务器程序组合为一个使用select来复用TCP和UDP套接口的单个服务器程序，图8.24是此程序的前半部分。

```
1 #include    "unp.h"
2 int
3 main(int argc, char * * argv)
4 {
5     int       listenfd, connfd, udpfd, nreadY, maxfdp1;
6     char      mesg[MAXLINE];
7     pid_t     childpid;
8     fd_set    rset;
9     ssize_t   n;
10    socklen_t len;
11    const int on = 1;
12    struct sockaddr_in cliaddr, servaddr;
13    void      sig_chld(int);
14        /* create listening TCP socket */
15    listenfd = Socket(AF_INET, SOCK_STREAM, 0);
16    bzero(&servaddr, sizeof(servaddr));
17    servaddr.sin_family = AF_INET;
18    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
19    servaddr.sin_port = htons(SERV_PORT);
20    Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
21    Bind(listenfd, (SA * ) &servaddr, sizeof(servaddr));
22    Listen(listenfd, LISTENQ);
23        /* create UDP socket */
24    udpfd = Socket(AF_INET, SOCK_DGRAM, 0);
25    bzero(&servaddr, sizeof(servaddr));
26    servaddr.sin_family = AF_INET;
27    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
28    servaddr.sin_port = htons(SERV_PORT);
29    Bind(udpfd, (SA * ) &servaddr, sizeof(servaddr));
```

图8.24 使用select处理TCP和UDP的回射服务器程序前半部分[udpcliserv/udpservselect01.c]

创建监听TCP套接口

第14～22行 创建一个监听TCP套接口并捆绑服务器的众所周知端口，设置SO_

REUSEADDR 套接口选项以防此端口上已有连接存在。

创建 UDP 套接口

第 23～29 行　也创建一个 UDP 套接口并捆绑与 TCP 套接口相同的端口。这里无需在调用 bind 之前设置 SO_REUSEADDR 套接口选项，因为 TCP 端口是独立于 UDP 端口的。

图 8.25 给出了服务器程序的后半部分。

```
30    Signal(SIGCHLD, sig_chld);    /* must call waitpid() */
31    FD_ZERO(&rset);
32    maxfdp1 = max(listenfd, udpfd) + 1;
33    for ( ; ; ) {
34        FD_SET(listenfd, &rset);
35        FD_SET(udpfd, &rset);
36        if ( (nready = select(maxfdp1, &rset, NULL, NULL, NULL)) < 0) {
37            if (errno == EINTR)
38                continue;          /* back to for() */
39            else
40                err_sys("select error");
41        }
42        if      (FD_ISSET(listenfd, &rset) ) {
43                len = sizeof(cliaddr);
44                connfd = Accept(listenfd, (SA *) &cliaddr, &len);
45                if( (childpid = Fork() ) ==0)     {     /* child process */
46                    Close(listenfd);     /* close listening socket */
47                    str_echo(connfd);    /* process the request */
48                    exit(0);
49                }
50                Close(connfd);}        /* parent closes connected socket */
51        }
52        if (FD_ISSET(udpfd, &rset)) {
53            len = sizeof(cliaddr);
54            n = Recvfrom(udpfd, mesg,MAXLINE, 0, (SA *) &cliaddr, &len);
55            Sendto(udpfd, mesg, n, 0,(SA *) &cliaddr,len);
56        }
57    }
58 }
```

图 8.25　使用 select 处理 TCP 和 UDP 的回射服务器程序后半部分[udpcliserv/udpservselect01.c]

给 SIGCHLD 建立信号处理程序

第 30 行　给 SIGCHLD 建立信号处理程序，因为 TCP 连接将由子进程处理。我们已在图 5.11 中给出了这个信号处理程序。

准备调用 select

第 31～32 行　我们给 select 初始化一个描述字集，并计算出我们等待的两个描述字的较大者。

调用 select

第 34～41 行　我们调用 select 仅仅为了等待监听 TCP 套接口的可读条件或 UDP 套

接口的可读条件。由于信号处理程序 sig_chld 可以中断我们对 select 的调用，因此我们要处理 EINTR 错误。

处理新的客户连接

第 42～51 行 当监听 TCP 套接口可读时，我们 accept 一个新的客户连接，fork 一个子进程，并在子进程中调用函数 str_echo。这与第 5 章所采取的步骤相同。

处理数据报的到达

第 52～57 行 如果 UDP 套接口可读，则说明数据报已到达。我们用 recvfrom 读它并用 sendto 将其发回给客户。

8.16 小 结

将我们的 TCP 回射客户-服务器程序转换为 UDP 回射客户-服务器程序是很简单的，但 TCP 提供的许多功能也消失了：检测丢失的分组并重传，验证响应是否来自正确的对方，等等。到 20.5 节我们再回过头来讨论这个话题，并看看为了给 UDP 应用程序加上可靠性将付出什么样的代价。

UDP 套接口可能产生异步错误，也就是说，在分组发送以后的某个时刻才能得到出错的报告。TCP 总是给应用进程报告这些错误，但 UDP 套接口必须连接才能接收这些错误。

UDP 没有流量控制，这是很容易演示证明的。一般来说，这不成什么问题，因为许多 UDP 应用程序是用请求-应答模式构成的，且不用于传送大量数据。

编写 UDP 应用程序时还有许多问题需要考虑，但我们留到第 20 章才讨论，即在讨论了接口函数、广播和多播以后再作讨论。

8.17 习 题

8.1 我们有两个应用程序，一个用 TCP，另一个用 UDP。对于 TCP 套接口，接收缓冲区中有 4096 字节的数据，对于 UDP 套接口，接收缓冲区中有两个 2048 字节的数据报。TCP 应用程序调用 read，其第三个参数为 4096，UDP 应用程序调用 recvfrom，其第三个参数也为 4096。这两个应用程序有什么差别吗？

8.2 在图 8.4 中，如果我们用 clilen 来代替 sendto 的最后一个参数（它原本是 len），将会怎样？

8.3 编译并运行图 8.3 及图 8.4 的 UDP 服务器程序和图 8.7 及图 8.8 的 UDP 客户程序。验证一下客户与服务器能一起工作。

8.4 在一个窗口中运行程序 Ping，并用-i 60 选项（每 60 秒发一个分组；有些系统用-I 而不是-i），-v 选项（输出所有接收到的 ICMP 错误），并指定回馈地址（通常为 127.0.0.1）。我们将用此程序来观察由服务器主机返回的端口不可达 ICMP 错误。然后，在另一个窗口运行上个习题中的客户，指定不在运行服务器的某主机的 IP 地址。将会怎样？

8.5　对于图 8.5，我们说过每个已连接 TCP 套接口都有自己的套接口接收缓冲区。监听套接口怎样？你认为它有自己的套接口接收缓冲区吗？

8.6　用程序 sock(C.3 节)和诸如 tcpdump(C.5 节)之类的工具来测试我们在 8.10 节所做的说明：如果客户捆绑一个 IP 地址到其套接口上，但从其他接口发出一个数据报，则即使此数据报的 IP 地址与外出接口不符，IP 数据报也仍然包含绑定在套接口上的 IP 地址。

8.7　编译 8.13 节中的程序并在不同的主机上运行客户和服务器。在客户程序中每次写一个数据报到套接口处放一个 printf 调用，这改变接收到分组的百分比吗？为什么？在服务器程序中每次从套接口读一个数据报处放一个 printf 调用，这改变接收到分组的百分比吗？为什么？

8.8　对于 UDP/IPv4 套接口，可传递给 sendto 的最大长度是多少；也就是说，可装填在一个 UDP/IPv4 数据报中的最大数据量是多少？若是 UDP/IPv6，有什么不同？修改图 8.8 以发送最大长度的 UDP 数据报，读回它，并输出由 recvfrom 返回的字节数。

第9章 基本名字与地址转换

9.1 概 述

到目前为止，本书中所有例子都用数值地址来表示主机(例如，206.6.226.33)，用数值端口号来标识服务器(例如，端口13代表标准的daytime服务器，端口9877代表我们的回射服务器)。然而，我们应该使用名字而不是数值，这有很多原因：名字比较容易记；数值地址可以改变但名字保持不变；随着往IPv6上转移，数值地址变得更长，手工键入一个地址更易出错。本章讲述在名字和数值地址间进行转换的函数：gethostbyname和gethostbyaddr在主机名字与IP地址间进行转换，getservbyname和getservbyport在服务名字和端口号间进行转换。

主机名函数最近已增强以便除IPv4外还能用于IPv6，我们也将叙述这些变化。这是我们往协议无关性转移的开始，在第11章将继续讨论这个话题。实际上，第11章使用我们在本章介绍的函数，并开发了大量函数以使我们的应用程序具有协议无关性。

9.2 域名系统

域名系统DNS(Domain Name System)主要用于主机名与IP地址间的映射。主机名可以是简单名字，如solaris或bsdi，也可以是全限定域名FQDN(Fully Qualified Domain Name)，如solaris.kohala.com。

> 严格说来，FQDN也称为绝对名字(absolute name)，因此必须以一个点号结尾，但用户经常省略最后的点号。

在本节，我们仅讨论网络编程中需要的DNS基础知识，对更多细节感兴趣的读者可参看TCPv1的第14章和[Albitz and Liu 1997]。IPv6所要求的附加内容在RFC 1886[Thomson and Huitema 1995]中。

资源记录

DNS中的条目称为资源记录RR(resource record)。仅有少数几类RR会影响我们的名字与地址转换。

A　A记录将主机名映射为32位的IPv4地址。例如，这里有kohala.com域中关于主机solaris的四个DNS记录，其中第一个就是一个A记录：

```
solaris IN A      206.62.226.33
        IN AAAA   5f1b:df00:ce3e:e200:0020:0800:2078:e3e3
        IN MX     5 solaris.kohala.com.
        IN MX     10.mailhost.kohala.com.
```

AAAA　AAAA 记录(称为“四 A”记录)将主机名映射为 128 位的 IPv6 地址。选择“四 A”的称法是由于 128 位地址是 32 位地址的四倍。

PTR　PTR 记录(称为“指针记录”)将 IP 地址映射为主机名。对于 IPv4 地址,32 位地址的四个字节顺序反转,每个字节都转换成它的十进制 ASCII 值(0～255),然后附上 in-addr.arpa,结果串用于 PTR 查询。

对于 IPv6 地址,128 位地址中的 32 个 4 位组顺序反转,每组被转换成相应的十六进制 ASCII 值(0～9,a～f),并附上 ip6.int。

例如,主机 solaris 的两个 PTR 记录将为:33.226.62.206.in-addr.arpa 和 3.e.3.e.8.7.0.2.0.0.8.0.0.2.0.0.0.0.2.e.e.3.e.c.0.0.f.d.b.l.f.5.ip6.int。

MX　MX 记录指定一主机作为某主机的“邮件交换器”。在上面主机 solaris 的例子中,提供了两个 MX 记录,第一个记录有优先级值 5,第二个记录有优先级值 10,当有多个 MX 记录存在时,需按优先级值的顺序使用,从最小值开始。

> 本书中我们不用 MX 记录,但因为它们在现实世界中应用很广,所以我们也提到它们。

CNAME　CNAME 代表“canonical name(规范名字)”,其常见的用法是为常用服务如 ftp 和 www 指派一个 CNAME 记录。如果人们使用这些服务名而不是实际上的主机名,则它在服务挪到其他主机上时是透明的。例如,主机 bsdi 的 CNAME。如下:

```
ftp       IN CANME bsdi.kohala.com.
www       IN CANME bsdi.kohala.com.
mailhost  IN CANME bsdi.kohala.com.
```

IPv6 部署得太早,结果管理员应给同时支持 IPv4 和 IPv6 的主机使用怎样的命名约定也不清楚。在本节前面的例子中,我们给主机 solaris 指定了 A 和 AAAA 两个记录。有些管理员将所有 AAAA 记录放到他们自己的子域中,其名字通常为 ipv6,例如,与 AAAA 记录相关联的主机名将是 solaris.ipv6.kohala.com。有时,当这种双协议栈主机的管理员没有整个域的域名管理责任但有独立的 ipv6 子域管理责任时,也会这么干。

但是,作者将 A 记录和 AAAA 记录都放到主机的通常名字(如前面所示)下,并创建另一个名字以-4 结尾、含有 A 记录的 RR,另一个名字以-6 结尾、含有 AAAA 记录的 RR,再另一个名字以-611结尾、含有 AAAA 记录及主机的链路局部地址的 RR(有时便于调试)。我们的主机的所有这些另一个记录如下所示:

```
aix-4     IN   A      206.62.226.43
aix       IN   A      206.62.226.43
          IN   MX     5    aix.kohala.com.
          IN   MX     10   mailhost.kohala.com.
          IN   AAAA   5f1b:df00:ce3e:e200:0020:0800:5afc:2b36
aix-6     IN   AAAA   5f1b:df00:ce3e:e200:0020:0800:5afc:2b36
aix-611   IN   AAAA                    fe80:0800:5afc:2b36
```

这使我们对由某些应用程序选择的协议有更多的控制权,这将在下一章讲述。

解析器和名字服务器

组织运行一个或多个名字服务器(name server),它们通常就是所谓的 BIND(Berkeley Internet Name Domain)程序。各种应用程序,如本书中我们编写的客户和服务器程序,通过调用称为解析器(resolver)的库中的函数来与 DNS 服务器联系。最常见的解析器函数是 gethostbyname 和 gethostbyaddr,两者都在本章中介绍,前者将主机名映射为 IP 地址,后者执行相反的映射。

图 9.1 给出了应用进程、解析器和名字服务器的一个典型关系。我们编写应用程序代码。解析器代码包含在系统库中,在构造应用程序时被链接到应用程序中。应用程序代码使用正常的函数调用来调用解析器代码,最典型的就是调用函数 gethostbyname 和 gethostbyaddr。

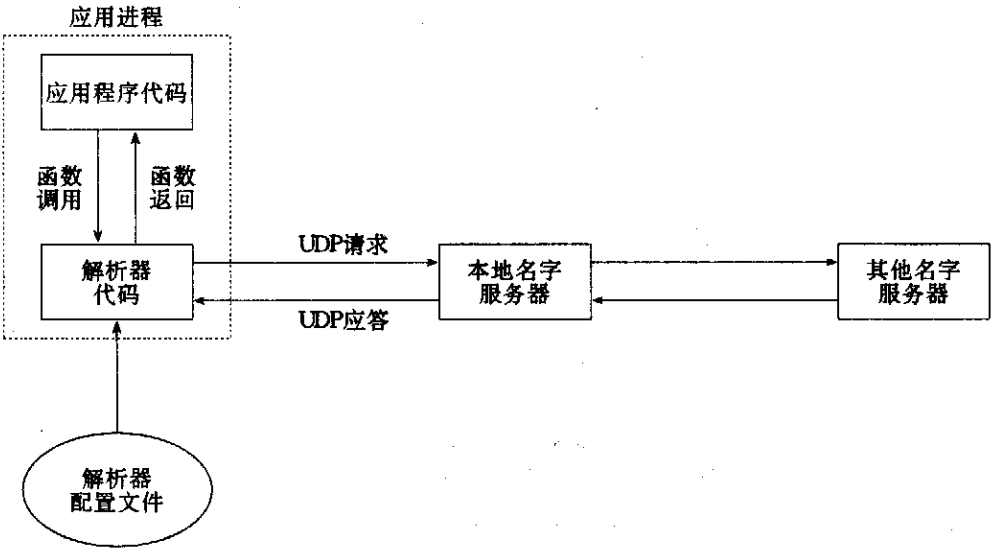


图 9.1 客户、解析器和名字服务器的典型关系

解析器代码读其依赖于系统的配置文件来确定组织的名字服务器们的所在位置(我们用复数"名字服务器们"是因为大多数组织运行多个名字服务器,虽然我们在图中只画了一个本地服务器)。文件/etc/resolv.conf 一般包含本地名字服务器的 IP 地址。

解析器用 UDP 给本地名字服务器发查询,如果本地名字服务器不知道答案,它也用 UDP 在整个因特网上给其他名字服务器发查询。

DNS 替代方法

不使用 DNS 也可得到名字和地址信息,最常用的替代方法为静态主机文件(一般为文件/etc/hosts,如图 9.9 所示)或网络信息系统 NIS(Network Information System)。不幸的是,管理员如何配置一个主机来使用不同的名字服务是依赖于不同的实现的,Solaris 2.x 和 HP-VX 10.30 用文件/etc/nsswitch.conf,Digital Unix 用文件/etc/svc.conf,AIX 用文件/etc/netsvc.conf。BIND 8.1 提供了自己的名为信息检索服务 IRS(Information Retrival Sevice)的版本,它使用文件/etc/irs.conf。如果一个名字服务器将为主机名查找所用,则所有这些系统都使用文件/etc/resolv.conf 来指定此名字服务器的 IP 地址。幸运的是,这些差异一

般都对应用程序开发人员透明，因此，我们只需调用诸如 gethostbyname 和 gethostbyaddr 这样的解析器函数即可。

9.3　gethostbyname 函数

计算机主机通常以人们可读的名字被认知。到现在为止，本书中的所有例子都有意使用 IP 地址而不是名字，所以我们确切地知道，对于诸如 connect 和 sendto 这样的函数，有什么东西进入了套接口地址结构，对于诸如 accept 和 recvfrom 这样的函数，它们返回什么。但是，大多数应用程序应该处理名字而不是地址。当我们往 IPv6 转移时，这显得尤为正确和重要，因为 IPv6 地址（十六进制数串）比 IPv4 点分十进制数串要长得多（前节中的 AAAA 记录和 PTR 记录的例子很明显地说明了这个问题）。

查找主机名最基本的函数是 gethostbyname，如果成功，它返回一个指向结构 hostent 的指针，该结构中包含了该主机的所有 IPv4 地址或 IPv6 地址。

```
#include <netdb.h>

struct hostent * gethostbyname(const char * hostname);

                返回：非空指针——成功，空指针——出错，同时设置 h_errno
```

此函数返回的非空指针指向下面的 hostent 结构：

```
struct hostent {
    char  * h_name;          /* official (canonical) name of host */
    char  * * h_aliases;     /* pointer to array of pointers to alias names */
    int     h_addrtype;      /* host address type: AF_INET or AF_INET6 */
    int     h_length;        /* length of address: 4 or 16 */
    char  * * h_addr_list;   /* ptr to array of ptrs with IPv4 or IPv6 addrs */
    };
#define h_addr h_addr_list[0]        /* first address in list */
```

按照 DNS 的说法，gethostbyname 执行一个对 A 记录的查询或对 AAAA 记录的查询。它返回 IPv4 地址或 IPv6 地址，我们在图 9.5 中总结了它返回这两类地址的条件。

> h_addr 的定义是为了向后兼容，新代码中不应再使用 h_addr。4.2BSD 没有成员 h_addr_list，而是有一个 char * h_addr，它仅指向一个 IP 地址。

图 9.2 所示为结构 hostent 和它所指向的各种信息的关系，它假设被查询的主机名有两个别名和三个 IPv4 地址。在这些字段中，正式（official）主机名和所有的别名都是以空字符（'\0'）结尾的 C 字符串。

返回的 h_name 被称为主机的规范（canonical）名字。例如给定前节所述的 CNAME 记录，主机 ftp.kohala.com 的规范名字将是 bsdi.kohala.com。而且，如果我们以主机 solaris 的某个无限定主机名如 solaris 调用 gethostbyname，则 FQDN（Solaris.kohala.com）将作为规范名字返回。

当返回 IPv6 地址时，结构 hostent 的成员 h_addrtype 被设置为 AF_INET6，成员 h_length 被设置为 16。图 9.3 给出了这些变化，它以阴影表示与图 9.2 不同的字段。

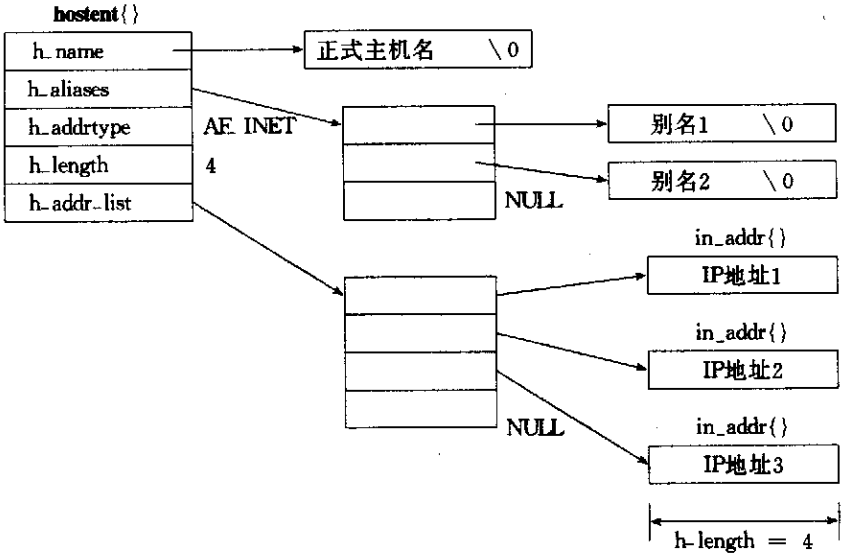


图 9.2 结构 hostent 和它所包含的信息

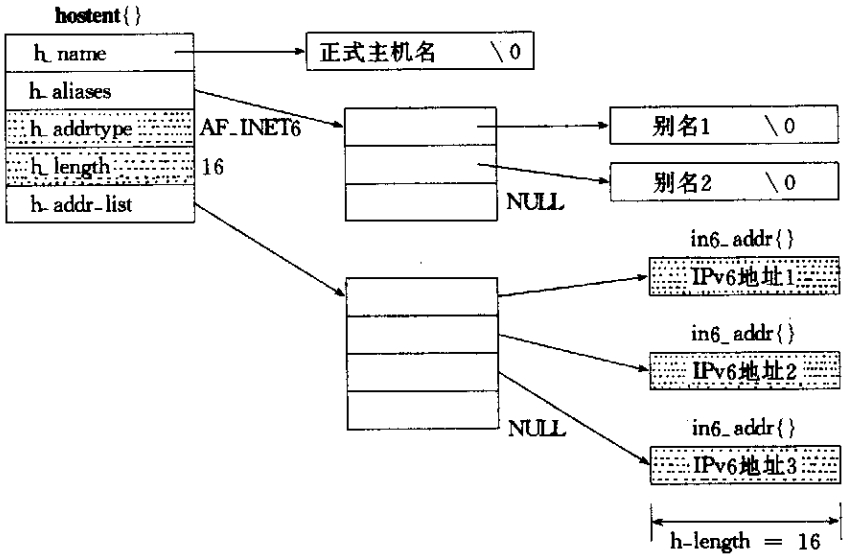


图 9.3 IPv6 地址时,结构 hostent 中所返回的不同信息

从 BIND 4.9.2 开始,新的 gethostbyname 版本允许主机名参数是点分十进制数串,也就是说,下列形式的调用是可行的:

```
hptr = gethostbyname ("206.62.226.33");
```

添加这个代码,是因为 Rlogin 客户仅接受主机名,给它调用 gethostbyname,而不接受点分十进制数串[Vixie 1996]。

gethostbyname 与我们所介绍的其他套接口函数的不同之处在于:当发生错误时,它不

设置 errno，而是将全局整数 h_errno 设置为定义在头文件<netdb.h>中的下列常值中的一个：

- HOST_NOT_FOUND
- TRY_AGAIN
- NO_RECOVERY
- NO_DATA(等同于 NO_ADDRESS)

错误 NO_DATA 表示指定的名字有效，但它既没有 A 记录，也没有 AAAA 记录。只有 MX 记录的主机名就是这样的例子。

BIND 的当前版本提供函数 hstrerror，它将 h_errno 的值作为唯一的参数，返回一个指向相应错误说明的 const char * 型指针。在下面的例子中，我们给出由此函数返回的一些字符串的例子。

例子

图 9.4 给出一个调用 gethostbyname 的简单例子，它可有任意数目的命令行参数，输出所有返回的信息。

```
1 #include    "unp.h"
2 int
3 main(int argc, char **argv)
4 {
5     char            *ptr, **pptr;
6     char            str[INET6_ADDRSTRLEN];
7     struct hostent  *hptr;
8     while (--argc > 0) {
9         ptr = *++argv;
10        if ( (hptr = gethostbyname(ptr)) == NULL) {
11            err_msg("gethostbyname error for host: %s: %s",
12                    ptr, hstrerror(h_errno));
13            continue;
14        }
15        printf("official hostname: %s\n", hptr->h_name);
16        for (pptr = hptr->h_aliases; *pptr != NULL; pptr++)
17            printf("\talias: %s\n", *pptr);
18        switch (hptr->h_addrtype) {
19        case AF_INET:
20 #ifdef   AF_INET6
21        case AF_INET6:
22 #endif
23            pptr = hptr->h_addr_list;
24            for ( ; *pptr != NULL; pptr++)
25                printf("\taddress: %s\n",
26                    inet_ntop(hptr->h_addrtype, *pptr, str, sizeof(str)));
27            break;
28        default:
29            err_ret("unknown address type");
30            break;
31        }
32    }
```

```
33      exit(0);
34  }
```

图 9.4 调用 gethostbyname 并输出返回信息[names/hostent.c]

第 8～14 行 给每个命令行参数调用 gethostbyname。

第 15～17 行 规范主机名被输出，后跟别名表。

第 20～22 行 为了让程序对 IPv4 地址和 IPv6 地址都支持，我们允许返回的地址类型为 AF_INET 或 AF_INET6，但只有后者被定义(也就是说，主机支持 IPv6)时，才能返回这类地址。

第 23～26 行 pptr 指向一个指针数组，数组中的每个指针都分别指向一个地址。对每个地址，我们调用 inet_ntop 并输出返回的字符串。注意，inet_ntop 基于它的第一个参数，可处理 IPv4 和 IPv6 地址。同时还要注意，我们定义 str 的长度为 INET6_ADDRSTRLEN，在 3.7 节中我们说过这是一个对最长的 IPv6 地址串都足够大的长度。在头文件<unp.h>中，我们定义了这个常值，这样即使主机不支持 IPv6，我们也总能指望它已经定义过(在代码中就省掉了又一个 #ifdef 语句)。

我们首先以主机 solaris 的名字运行此程序，它仅有一个 IPv4 地址。

```
solaris % hostent solaris
official hostname: solaris.kohala.com
        address:206.62.226.33
```

注意，正式主机名(official hostname)就是 FQDN，同时也要注意，即使此主机有 IPv6 地址，也只有 IPv4 地址被返回。下面是有多个 IPv4 地址的主机的情况：

```
solaris % hostent gemini.tuc.noao.edu
official hostname: gemini.tuc.noao.edu
        address: 140.252.1.11
        address: 140.252.3.54
        address: 140.252.4.54
        address: 140.252.8.54
```

下面是 9.2 节中介绍过的，有一个 CNAME 记录的名字：

```
solaris % hostent www
official hostname: bsdi.kohala.com
        alias:www.kohala.com
        address:206.62.226.35
```

正如所预期的那样，正式主机名与我们的命令行参数不同。

为了看到由函数 hstrerror 返回的错误串，我们首先指定一个不存在的主机名，然后指定一个仅有 MX 记录的名字。

```
solaris % hostent nosuchname
gethostbyname error for host:nosuchname:Unknown host
solaris % hostent uunet.uu.net
gethostbyname error for host:uunet.uu.net:No address associated with name
```

9.4　RES_USE_INET6 解析器选项

BIND 的较新版本(4.9.4 及以后版本)提供了一个名为 RES_USE_INET6 的解析器选项,我们可以以三种不同的方法来设置它。我们可以用此选项来通知解析器我们想让 gethostbyname 返回 IPv6 地址而不是 IPv4 地址。

1. 应用程序本身就可以设置此选项,首先调用解析器的 res_init 函数,然后打开该选项:

```
#include <resolv.h>
res_init();
_res.options |= RES_USE_INET6;
```

这必须在第一次调用 gethostbyname 或 gethostbyaddr 之前完成。此选项仅对那些设置了此选项的应用程序才有效。

2. 如果环境变量 RES_OPTIONS 含有串 inet6,则此选项打开。此选项的作用依赖于环境变量的范围。例如,如果我们在.profile 文件(假设使用 Korn Shell)中以 export 属性设置它,如:

```
export RES_OPTIONS=inet6
```

则它对从登录 shell 开始运行的每个程序都有效。但如果我们仅在命令行上设置该环境变量(我们马上会看到),则它仅对那个命令有影响。

3. 解析器配置文件(一般为/etc/resolv.conf)可以包含如下行:

```
options inet6
```

然而必须清楚,在解析器配置文件中设置此选项影响主机上调用解析器函数的所有应用程序,因此,这项技术要直到结构 hostent 中返回的 IPv6 地址可以被主机上的所有应用程序所处理时才能使用。

第一个方法以每个应用程序为基设置此选项,第二个方法以每个用户为基,第三个方法以整个系统为基。

现在,我们设置环境变量 RES_OPTIONS 为值 inet6,运行图 9.4 中的例子程序。

```
solaris % RES_OPTIONS=inet6 hostent solaris      这是具有 AAAA 记录的主机名
official hostname:solaris.kohala.com
        address:5f1b:df00:ce3e:e200:20:800:2078:e3e3
solaris % RES_OPTIONS=inet6 hostent bsdi      这是没有 AAAA 记录的主机名
official hostname:bsdi.kohala.com
        address: ::ffff:206.62.226.35
        address: ::ffff:206.62.226.66
```

第一次执行程序时,它返回主机的 IPv6 地址(回忆一下它在 9.2 节中的 AAAA 记录),第二次运行此程序时,我们指定一个没有 AAAA 记录的主机,它仍然返回 IPv6 地址:IPv4 映射的 IPv6 地址(A.5 节)。

在下两节,我们将详细讨论解析器中的 IPv6 支持。

9.5 gethostbyname2 函数与 IPv6 支持

当 IPv6 支持增加到 BIND 4.9.4 时，函数 gethostbyname2 也增加进去，它有两个参数，允许我们指定地址族。

```
#include <netdb.h>
struct hostent *gethostbyname2(const char *hostname, int family);
                    返回：非空指针——成功，空指针——出错，同时设置 h_errno
```

其返回值与 gethostbyname 的返回值相同，为一个指向结构 hostent 的指针，且此结构也保持不变。该函数的逻辑依赖于参数 family 和解析器选项 RES_USE_INET6（我们在前节结尾已提到）。

在描述具体细节之前，图 9.5 总结了对于新的选项 RES_USE_INET6，函数 gethostbyname 和 gethostbyname2 的操作：

- RES_USE_INET6 选项是打开还是关闭；
- gethostbyname2 的第二个参数是 AF_INET 还是 AF_INET6；
- 解析器是搜索 A 记录还是搜索 AAAA 记录；
- 返回地址长度为 4 还是 16。

函数 gethostbyname2 的操作如下：

- 如果参数 family 是 AF_INET，则查询 A 记录。若不成功，则返回一个空指针，若成功，则返回地址的类型及大小依赖于新的解析器选项 RES_USE_INET6：若选项未设置（缺省），则返回 IPv4 地址，结构 hostent 的成员 h_length 的值将为 4；若选项设置，则返回 IPv4 映射的 IPv6 地址，结构 hostent 的成员 h_length 的值将为 16。
- 如果参数 family 为 AF_INET6，则查询 AAAA 记录。若成功，则返回 IPv6 地址，结构 hostent 的成员 h_length 的值将为 16；否则返回一个空指针。

	RES_USE_INET6 选项	
	关闭	打开
gethostbyname (host)	搜索 A 记录，若找到，返回 IPv4 地址 (h_length = 4)，否则返回错误。这为现存的 IPv4 应用程序提供了向后兼容性	搜索 AAAA 记录，若找到，返回 IPv6 地址 (h_length = 16)，否则搜索 A 记录。若找到，返回 IPv4 映射的 IPv6 地址 (h_length = 16)，否则返回错误
gethostbyname2 (host, AF_INET)	搜索 A 记录，若找到，返回 IPv4 地址 (h_length = 4)，否则返回错误	搜索 A 记录，若找到，返回 IPv4 映射的 IPv6 地址 (h_length = 16)，否则返回错误

（续）

	RES_USE_INET6 选项	
	关闭	打开
gethostbyname2 (host,AF_INET6)	搜索 AAAA 记录，若找到，返回 IPv6 地址(h_length = 16)，否则返回错误	搜索 AAAA 记录，若找到，返回 IPv6 地址(h_length = 16)，否则返回错误

图 9.5　解析器选项 RES_USE_INET6 与函数 gethostbyname 和 gethostbyname2

如果应用程序想强制某个指定地址类型的搜索：IPv4 或 IPv6，则可用此函数。但对应用程序来说，调用 gethostbyname 似乎更常见，更何况此函数的较新版本既可返回 IPv4，也可返回 IPv6 地址。

描述函数 gethostbyname 和选项 RES_USE_INET6 的行为的一个方法是看看它的源代码，如图 9.6 所示。

```
struct hostent *
gethostbyname(const char *name)
{
    struct hostent *hp;
    if ((_res.options & RES_INIT) == 0 && res_init() == -1) {
            h_errno = NETDB_INTERNAL;
            return (NULL);
    }
    if (_res.options & RES_USE_INET6) {
        hp = gethostbyname2(name,AF_INET6);
        if (hp)
            return (hp);
    }
    return ( gethostbyname2(name,AF_INET));
}
```

图 9.6　函数 gethostbyname 和 IPv6 支持

如果解析器还没有被初始化（没有设置标志 RES_INIT），则调用 res_init。此初始化函数检查并处理环境变量 RES_OPTINOS。如果这个变量包含串 inet6 或如果解析器配置文件包含行 options inet6，则标志 RES_USE_INET6 由 res_init 设置。res_init 一般是由函数 gethostbyname 或 gethostbyaddr 在第一次被应用程序调用时自动调用的。此外，我们已展示应用程序也可调用 res_init，然后显式设置标志 RES_USE_INET6。

如果选项 RES_USE_INET6 没有设置，则函数的最后一行被执行，它就是用 AF_INET 地址族参数调用 gethostbyname2。在图 9.5 中我们看到，此调用仅搜索 A 记录，这为所有现存的应用程序提供了向后兼容性。

如果选项 RES_USE_INET6 已经设置，就用 AF_INET6 地址族参数调用 gethostbyname2 以搜索 AAAA 记录（图 9.5）。如果成功，则 gethostbyname 返回，如果失败，则继续用 AF_INET 地址族参数调用 gethostbyname2 以搜索 A 记录。若第二次搜索成功，则图 9.6 中并不明显的事情是：这些 4 字节的地址被自动映射成 16 字节的 IPv4 映射的 IPv6 地址。

总之，当选项 RES_USE_INET6 打开且应用程序调用 gethostbyname 时，应用程序通

知解析器："我只想 IPv6 地址被返回，首先搜索 AAAA 记录，如果未找到则搜索 A 记录，如果 A 记录找到则返回 IPv4 映射的 IPv6 地址。"

9.6 gethostbyaddr 函数

函数 gethostbyaddr 取一个二进制的 IP 地址并试图找到相应于此地址的主机名，这与 gethostbyname 的行为刚好相反。

此函数返回一个指向结构 hostent 的指针。这个讨论 gethostbyname 时描述过的结构中我们最感兴趣的成员是规范主机名 h_name。

```
#include <netdb.h>
struct hostent *gethostbyaddr(const char *addr, size_t len, int family);
                    返回：非空指针——成功，空指针——出错，同时设置 h_errno
```

参数 addr 不是 char * 类型，而是一个真正指向含有 IPv4 或 IPv6 地址的结构 in_addr 或 in6_addr 的指针；len 是此结构的大小：对于 IPv4 地址为 4，对于 IPv6 地址为 16；参数 family 或为 AF_INET 或为 AF_INET6。

按照 DNS 的说法，gethostbyaddr 在域 in_addr.arpa 中给 IPv4 地址在名字服务器上查询 PTR 记录，或在域 ip6.int 中给 IPv6 地址查询 PTR 记录。

函数 gethostbyaddr 和 IPv6 支持

gethostbyaddr 总有一个地址族参数，所以当加上 IPv6 支持到 BIND 时，无需发明另一个函数(类似于函数 gethostbyname2)。但是，当参数是 IPv6 地址时，仍有一些差别。下面的三个测试按步骤进行：

1. 如果 family 是 AF_INET6，len 是 16，且地址是 IPv4 映射的 IPv6 地址，则在域 in-addr.arpa 中查找地址的低 32 位(IP4 地址部分)。
2. 如果 family 是 AF_INET6，len 是 16，且地址是 IPv4 兼容的 IPv6 地址，则在域 in-addr.arpa 中查找地址的低 32 位(IPv4 地址部分)。
3. 如果被查找的是 IPv4 地址(或参数 family 为 AF_INET，或上述两种情况中的一个为真)且解析器选项 RES_USE_INET6 设置，则返回的地址(参数 addr 的一个拷贝)被转换为一个 IPv4 映射的 IPv6 地址：h_addrtype 为 AF_INET6，h_length 为 16。

第三点不太重要，因为很少有应用程序检查由 gethostbyaddr 返回的 IP 地址，它仅是参数的一个拷贝。应用程序一般调用此函数来检查所返回 hostent 结构的 h_name 成员(可能还有别名)。

9.7　uname 函数

函数 uname 返回当前主机的名字。它不是解析器库中的一部分，但我们仍在这里讨论它，因它经常与函数 gethostbyname 一起用来确定本地主机的 IP 地址。

```
#include <sys/utsname.h>
int uname(struct utsname *name);
                                        返回：非负值——成功，-1——出错
```

此函数装填结构 utsname，其地址由调用者传递：

```
#define UTS_NAMESIZE 16
#define UTS_NODESIZE 256
struct utsname {
    char sysname [_UTS_NAMESIZE];     /* name of this operating system */
    char nodename [_UTS_NODESIZE];    /* name of this node */
    char release [_UTS_NAMESIZE];     /* O.S release level */
    char version [_UTS_NAMESIZE];     /* O.S version level */
    char machine [_UTS_NAMESIZE];     /* hardware type */
};
```

不幸的是，Posix.1 所规定的只是我们所示的五个结构成员的名字以及每个数组是一个以空字符('\0')终止的字符数组这两点，对于每个数组的大小及内容则并未作任何说明。我们所给出的大小来源于 4.4BSD，其他操作系统采用不同的大小。

从网络编程的角度看，最严重的忽略是对数组 nodename 大小和内容的定义。有些系统仅在此数组中存储主机名(例如 gemini)，而另外的一些系统存储 FQDN(例如 gemini.tuc.noao.edu)。在有些操作系统如 Solaris 2.x 上，既可存放主机名，也可存放 FQDN，这取决于管理员是如何安装操作系统的。

例子：确定本地主机的 IP 地址

为了确定本地主机的 IP 地址，我们调用 uname 以得到主机名字，然后调用 gethostbyname 以得到它的所有 IP 地址。图 9.7 中所示函数 my_addrs 执行这些步骤。

```
1 #include    "unp.h"
2 #include    <sys/utsname.h>
3 char **
4 my_addrs(int *addrtype)
5 {
6     struct hostent    *hptr;
7     struct utsname    myname;
8     if (uname(&myname) < 0)
9         return(NULL);
10    if ( (hptr = gethostbyname(myname.nodename)) == NULL)
```

```
11          return(NULL);
12      *addrtype = hptr->h_addrtype;
13      return(hptr->h_addr_list);
14  }
```

图 9.7 返回主机上所有 IP 地址的函数[lib/my_addrs.c]

函数的返回值是结构 hostent 的成员 h_addr_list，即指向 IP 地址的指针数组。我们还通过指针参数返回地址族。

确定本地主机 IP 地址的另一个方法是用 ioctl 的命令 SIOCGIFCONF，这在第 16 章再作讨论。

9.8 gethostname 函数

函数 gethostname 也返回当前主机的名字。

```
#include <unistd.h>
int gethostname(char *name, size_t namelen);
                                    返回：0——成功，-1——出错
```

name 是指向主机名存储位置的指针，namelen 是此数组的大小。如果有空间，主机名以空字符结束。主机名的最大大小通常是由头文件<sys/param.h>定义的常值 MAXHOSTNAMELEN。

历史上，uname 由系统 V 定义，而 gethostbyname 由 Berkeley 定义。Posix.1 指定 uname，但 Unix 98 两者都要。

9.9 getservbyname 和 getservbyport 函数

服务器就像主机一样，也常常是由名字来标识的。如果我们在代码中，通过服务器的名字而不是通过服务器端口号来认知它，而且如果从主机到端口号的映射包含在一个文件中（一般是/etc/services），则如果端口号改变，我们所需做的所有修改就是改动文件/etc/services 中的一行，而不是重新编译应用程序。下面的函数 getservtbyname 可以根据给定名字查找服务。

```
#include <netdb.h>
struct servent *getservbyname(const char *servname, const char *protoname);
                                    返回：非空指针——成功，空指针——出错
```

此函数返回一个指向下面所示结构的指针：

```
struct servent {
    char  *s_name;      /* official service name */
    char  **s_aliases;  /* alias list */
```

```
    int     s_port;         /* port number, network-byte order */
    char *  s_proto;        /* protocol to use */
};
```

服务名 servname 必须指定，如果还指定了一个协议(即 protoname 为非空指针)，则结果表项也必须有匹配的协议。有些因特网服务是用 TCP 或 UDP 来提供的(例如 DNS 及图 2.13 中的所有服务)，而其他服务则仅支持单个协议(如 FTP 只需 TCP)。如果 protoname 没有指定且服务支持多个协议，则返回哪个端口是依赖于实现的。一般来说这没有什么关系，因为支持多个协议的服务常常使用相同的 TCP 和 UDP 端口号，但并没有保证。

结构 servent 中我们关心的主要成员是端口号。由于端口号是以网络字节序返回的，在将它存储于套接口地址结构时，绝对不能调用 htons。

对此函数的典型调用是：

```
struct servent *sptr;
sptr = getservbyname("domain", "udp");      /* DNS using UDP */
sptr = getservbyname("ftp","tcp");          /* FTP using TCP */
sptr = getservbyname("ftp",NULL);           /* FTP using TCP */
sptr = getservbyname("ftp","udp");          /* this call will fail */
```

由于 FTP 仅支持 TCP，所以第二个调用和第三个调用是相同的，第四个调用将失败。文件/etc/services 中的典型行是：

```
solaris % grep -e ftp -e domain /etc/services
ftp-data        20/tcp
ftp             21/tcp
domain          53/udp
domain          53/tcp
tftp            69/udp
```

下一个函数 getservbyport 在给定端口号和可选协议后查找相应的服务。

```
#include <netdb.h>
struct servent *getservbyport(int port, const char *protname);
                                    返回：非空指针——成功，空指针——出错
```

port 值必须为网络字节序。对此函数的典型调用是：

```
struct servent *sptr;
sptr = getservbyport(htons(53),   "udp");   /* DNS using UDP */
sptr = getservbyport(htons(21),   "tcp");   /* FTP using TCP */
sptr = getservbyport(htons(21),   NULL);    /* FTP using TCP */
sptr = getservbyport(htons(21),   "udp");   /* this call will fail */
```

对于 UDP，由于没有服务使用端口 21，所以最后一个调用将失败。

必须知道，有些端口号对于 TCP 可能用于一种服务，但对于 UDP，同样的端口号可能用于完全不同的服务。例如：

```
solaris % grep 514 /etc/services
shell           514/tcp
syslog          514/udp
```

表明端口514在TCP协议由rsh命令使用，但在UDP协议是由syslog守护进程使用。端口512～514都有这个特性。

例子：使用gethostbyname和getservbyname

现在，我们可以修改图1.5中的TCP时间/日期客户程序，让它使用gethostbyname和getservbyname，并改用两个新的命令行参数：主机名和服务名。图9.8给出了此程序，它也展示了我们所期望的行为：试着连接到多宿服务器主机的所有的IP地址，直到有一个成功或所有地址都试遍为止。

```
1 #include    "unp.h"
2 int
3 main(int argc, char **argv)
4 {
5     int         sockfd, n;
6     char        recvline[MAXLINE + 1];
7     struct sockaddr_in servaddr;
8     struct in_addr **pptr;
9     struct hostent *hp;
10    struct servent *sp;
11    if (argc != 3)
12        err_quit("usage: daytimetcpcli1 <hostname> <service>");
13    if ( (hp = gethostbyname(argv[1])) == NULL)
14        err_quit("hostname error for %s: %s", argv[1], hstrerror(h_errno));
15    if ( (sp = getservbyname(argv[2], "tcp")) == NULL)
16        err_quit("getservbyname error for %s", argv[2]);
17    pptr = (struct in_addr **) hp->h_addr_list;
18    for ( ; *pptr != NULL; pptr++) {
19        sockfd = Socket(AF_INET, SOCK_STREAM, 0);
20        bzero(&servaddr, sizeof(servaddr));
21        servaddr.sin_family = AF_INET;
22        servaddr.sin_port = sp->s_port;
23        memcpy(&servaddr.sin_addr, *pptr, sizeof(struct in_addr));
24        printf("trying %s\n",
25               Sock_ntop((SA *) &servaddr, sizeof(servaddr)));
26        if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) == 0)
27            break;          /* success */
28        err_ret("connect error");
29        close(sockfd);
30    }
31    if (*pptr == NULL)
32        err_quit("unable to connect");
33    while ( (n = Read(sockfd, recvline, MAXLINE)) > 0) {
34        recvline[n] = 0;   /* null terminate */
35        Fputs(recvline, stdout);
36    }
37    exit(0);
38 }
```

图9.8 使用gethostbyname和getservbyname的时间/日期客户程序[names/daytimetcpcli1.c]

调用 gethostbyname 和 getservbyname

第 13～16 行　第一个命令行参数是主机名，我们将它作为参数传递给 gethostbyname；第二个命令行参数是服务名，我们将它作为参数传递给 getservbyname。假设我们的代码用 TCP，我们将它作为 getservbyname 的第二个参数。

尝试每个服务器

第 18～25 行　现在，我们将对 socket 和 connect 的调用放在同一循环中，给服务器主机的每个地址执行一次，直到连接成功或所有 IP 地址都已试完。调用 socket 以后，我们用服务器的 IP 地址和端口装填网际套接口地址结构。虽然我们可以把对 bzero 的调用和它后面的两条赋值语句放到循环体之外来提高执行效率，但图中所示的代码较易读。与服务器建立连接很少是网络客户的性能瓶颈。

调用 connect

第 26～30 行　调用 connect，如果成功，则 break 出循环；如果连接建立失败，则输出一个错误并关闭套接口。前面讲过，调用 connect 失败的描述字必须关闭，不能再被使用。

检查错误

第 31～32 行　如果循环因没有一个 connect 调用成功而终止，则程序也结束。

读服务器的应答

第 33～37 行　否则，我们读服务器的应答，当服务器关闭连接时，也终止程序。

如果我们指定正在运行时间/日期服务器的主机运行此程序，则我们得到期望的输出：

```
solaris % daytimetcpcli1 aix daytime
trying 206.62.226.35.13
Thu May 22 19:28:11 1997
```

更有意思的是指定一个不在运行时间/日期服务器的多宿路由器运行此程序：

```
solaris % daytimetcpcli1 gateway.tuc.noao.edu daytime
trying 140.252.1.4.13
connect error: Connection refused
trying 140.252.101.4.13
connect error: Connection refused
trying 140.252.102.1.13
connect error: Connection refused
trying 140.252.104.1.13
connect error: Connection refused
trying 140.252.3.6.13
connect error: Connection refused
trying 140.252.4.100.13
connect error: Connection refused
unable to connect
```

9.10　其他网络相关信息

本章我们的重点在于主机名与 IP 地址、服务名与端口号。但扩大一下我们的视野，就会发现，其实应用程序可能需查询四种类型的信息（与网络相关的）：主机、网络、协议和服务。

大多数查询都是针对主机的(gethostbyname 和 gethostbyaddr),有一小部分是针对服务的(getservbyname 和 getservbyport),针对网络和协议的查询就更少了。

所有四类信息都可存储在文件中,而且每类信息都定义有三个函数:

1. 函数 getXXXent 读文件中的下一表项,在必要时可以打开文件;
2. 函数 setXXXent 打开(如果文件还没有打开的话)并回绕文件;
3. 函数 endXXXent 关闭文件。

每类信息都定义了自己的结构,这些定义包含在文件<netdb.h>中:hostent、netent、protoent 和 servent 结构。

除了三个用于文件的顺序处理的 get、set 和 end 函数外,每类信息还提供了一些键值查询(keyed loopup)函数。它们顺序浏览文件(调用函数 getXXXent 来读每一行),但不返回每一行给调用者,而是查找一个与某参数匹配的表项。这些键值查询函数都有形如 getXXXbyYYY 的名字。例如,针对主机信息的两个关键字查询函数是 gethostbyname(查找与主机名匹配的表项)和 gethostbyaddr(查找与 IP 地址匹配的表项)。图 9.9 对此作了总结。

信息	数据文件	结构	键值搜索函数
主机	/etc/hosts	hostent	gethostbyaddr, gethostbyname
网络	/etc/networks	netent	getnetbyaddr, getnetbyname
协议	/etc/protocols	protoent	getprotobyname, getprotobynumber
服务	/etc/services	servent	getservbyname, getservbyport

图 9.9 四类与网络相关的信息

当 DNS 正在使用时,如何得到这些信息?首先,只有主机和网络信息是通过 DNS 提供的,服务和协议信息一般要从相应的文件中读。在本章前面(图 9.1)我们提到过,不同的实现可让管理员使用不同的方法来指定是使用 DNS 还是使用文件来得到主机和网络信息。

第二,如果 DNS 正用于主机和网络信息,则只有键值查询函数才可用。例如,你不能使用 gethostent 期待顺序浏览 DNS 中的所有表项。如果调用 gethostent,它只读主机文件而避开 DNS。

> 虽然网络信息可以通过 DNS 得到,但很少有人这么用。[Albitz and Liu 1997]第 346~348 页介绍了这个特性。通常,管理员创建并维护文件/etc/networks,并使用它而不是使用 DNS。如果有此文件,则指定-i 选项的 netstat 程序使用它,以输出每个网络的名字。

9.11 小 结

应用程序用来将主机名转换为 IP 地址或进行相反过程的一组函数称为解析器。gethostbyname 和 gethostbyaddr 是两个最常用的函数。随着向 IPv6 的转移,由这两个函数装填的 hostent 结构保持不变,但结构中的某些信息发生了变化。为支持 IPv6,还需要一个新的函数 gethostbyname2 和一个新的解析器选项 RES_USE_INET6。

处理服务名和端口号的常用函数是 getservbyname,它取一个服务名并返回一个包含端

口号的结构。这种映射一般包含在文本文件中。还有用来映射协议名到协议号、网络名到网络号的函数，但很少使用。

我们没有提到的另外一种方法是：直接调用解析器函数，而不使用 gethostbyname 和 gethostbyaddr。以这个方法来调用 DNS 的一个程序是 sendmail，它搜索 MX 记录，这是 gethostbyXXX 函数无法做到的。解析器函数都有以 res_开头的名字，在 9.4 节中描述的 res_init 就是一个例子。这些函数的描述和调用它们的一个程序例子在[Albitz and Liu 1997]的第 14 章中，键入 man resolver 应能输出这些函数的手册页面。

当我们在第 11 章查看对 gethostbyname 和 gethostbyaddr 的协议无关接口：函数 getaddrinfo 和 getnameinfo 时，将继续讨论名字和地址转换这个话题。那两个新函数是设计来与 IPv4 和 IPv6 一起工作的，但我们首先需在下一章看看 IPv4 和 IPv6 的互操作性，然后再讨论它们。

9.12　习　题

9.1　修改图 9.4 中的程序，给每个返回的地址调用 gethostbyaddr，然后输出返回的 h_name。首先指定一个只有单个 IP 地址的主机，运行此程序；然后指定一个具有多个 IP 地址的主机，再运行此程序，将会怎样？

9.2　修复上个习题中出现的问题。

9.3　修改图 9.7，调用 gethostname 而不是 uname。写一个 main 函数来调用 my_addrs 并输出 IP 地址。

9.4　在图 9.8 中，如果我们将 memcpy 的第三个参数改为 hp->h_length（在装填套接口地址结构时），将会怎样？（提示：考虑一下，如果我们设置 RES_OPTIONS = inet6 并指定一个具有 IPv6 地址的主机名，运行程序将会有什么结果。）

9.5　指定服务名为 chargen，运行图 9.8 中的程序。

9.6　指定一个点分十进制数 IP 地址作为主机名，运行图 9.8 中的程序。你的解析器允许这么做吗？修改图 9.8 以允许将点分十进制数 IP 地址作为主机名，将十进制端口号串作为服务名。在给点分十进制数串或主机名测试 IP 地址时，这两个测试将以什么顺序执行？

9.7　修改图 9.8，使其对 IPv4 和 IPv6 都能工作。

9.8　修改图 9.8 以查询 DNS，比较返回的 IP 地址和目的主机的所有 IP 地址。也就是说，用由 recvfrom 返回的 IP 地址调用 gethostbyaddr，后跟 gethostbyname 调用以找出主机的所有 IP 地址。

第3部分 高级套接口编程

第10章 IPv4与IPv6的互操作性

10.1 概 述

即将到来的数年中因特网可能会逐渐从IPv4过渡到IPv6，在这个过渡阶段里，使现有的基于IPv4的应用程序能和新的IPv6的应用程序一起运行是十分重要的。举例来说，某厂商不能只提供一个只能与IPv6的Telnet服务器程序一起运行的Telnet客户程序，她必须既提供能与IPv4服务器程序一起运行的Telnet客户程序，又提供能与IPv6服务器程序一起运行的Telnet客户程序。能提供一个同时兼容IPv4和IPv6服务器程序的Telnet客户程序，和一个同时兼容IPv4和IPv6客户程序的Telnet服务器程序就更好。我们将会在本章中了解这是如何实现的。

我们在本章中假定所有的主机上都运行着双重协议栈(dual stacks)，即IPv4和IPv6协议栈。图2.1中的例子是一个双重协议栈的主机。在向IPv6协议过渡的很长时间内，主机和路由器可能都会像这样运行。在某个时候，许多系统将可以关闭它们的IPv4协议栈，但只有时间才能告诉我们何时能这样做。

在本章中我们讨论IPv4和IPv6的应用程序怎样才能互相通信。在使用IPv4和IPv6协议的客户和服务器之间存在四种组合，如图10.1。

	IPv4服务器	IPv6服务器
IPv4客户	几乎全部现有的客户和服务器程序	在10.2节中讨论
IPv6客户	在10.3节中讨论	简单地修改大部分现有的客户和服务器程序(如从图1.5到图1.6的修改)

图10.1 使用IPv4和IPv6的客户与服务器的组合

对于客户和服务器使用相同协议的两种情况我们将不做过多讨论，最有趣的情况是当客户和服务器使用不同协议时。

10.2 IPv4客户与IPv6服务器

拥有双重协议栈的主机的一个基本特性就是：其上运行的IPv6服务器既能应付IPv4客户，又能应付IPv6客户。这是通过使用IPv4映射的IPv6地址实现的(图A.10)。图10.2给出了这样的一个例子。

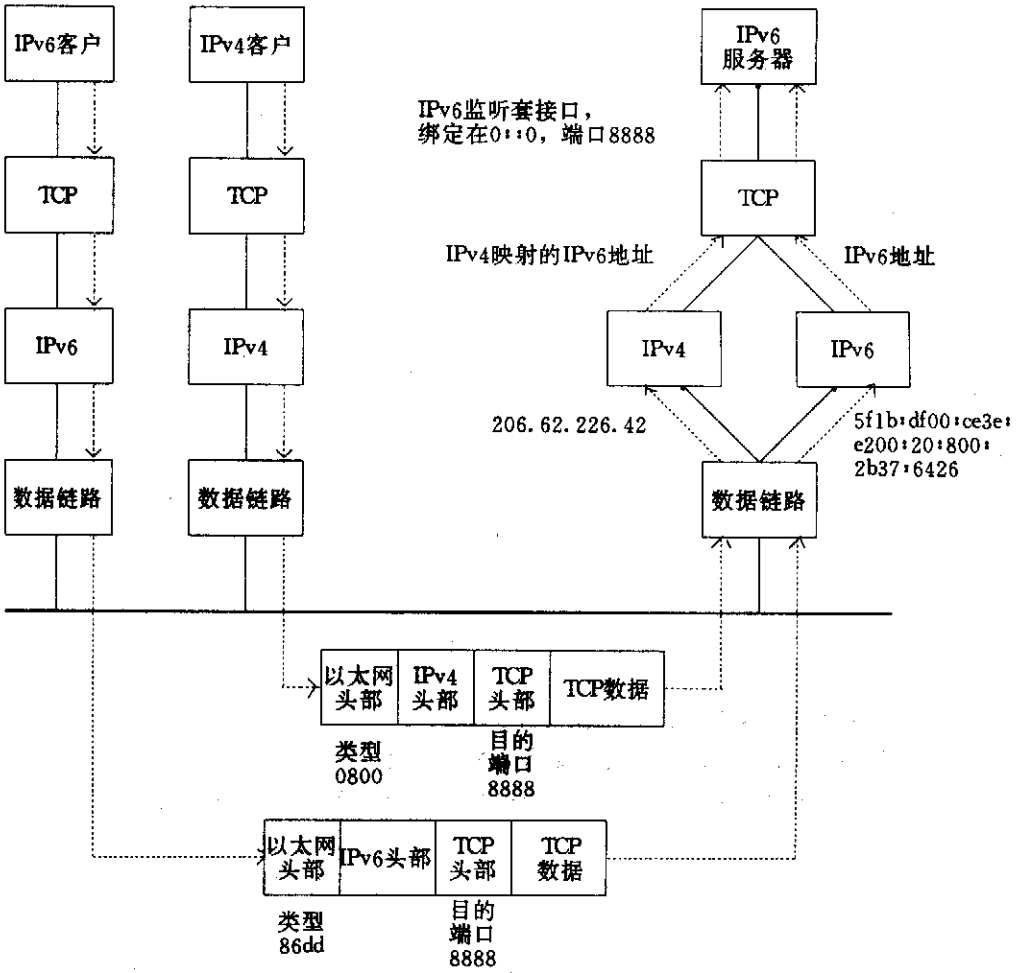


图 10.2　双重协议栈主机上的 IPv6 服务器为 IPv4 和 IPv6 客户服务

在左边是一个 IPv4 客户和一个 IPv6 客户。右边的服务器其程序是用 IPv6 编写的，在一个拥有双重协议栈的主机上运行。该服务器已创建了一个绑定在通配 IPv6 地址和端口 8888 上的 IPv6 监听 TCP 套接口。

我们假定客户和服务器处于同一以太网上。当然，它们也可以是通过路由器来连接的，只要所有的路由器都能支持 IPv4 和 IPv6，但这对我们的讨论并没有什么影响。另外的一种情况是：IPv6 的客户和服务器之间用只支持 IPv4 的路由器连接，这将在 B.3 节中讨论。

我们假定两个客户都发送 SYN 分节与服务器建立连接。IPv4 客户主机将使用 IPv4 数据报发送 SYN，IPv6 客户主机将使用 IPv6 数据报发送 SYN。IPv4 客户发出的 TCP 分节出现在以太网上时是一个以太网头部，紧接着一个 IPv4 头部、一个 TCP 头部和 TCP 数据。以太网头部中包含一个值为 0x0800 的类型字段，标识其为一个 IPv4 帧。在 TCP 头部中包含目的端口 8888。(附录 A 中对这些头部的格式和内容有详细的说明。)在 IPv4 头部中的目的 IP 地址是 206.62.226.42，在图中没有标出。

IPv6客户发出的TCP分节出现在以太网上时是一个以太网头部，紧接着一个IPv6头部、一个TCP头部和TCP数据。以太网头部包含一个值为0x86dd的类型字段，标识其为一个IPv6帧。这个TCP头部和IPv4数据报中的TCP头部格式完全一样，也包含目的端口8888。在IPv6头部中的目的IP地址是5f1b:df00:ce3e:e200:20:800:2b37:6426，在图中没有标出。

正在接收的数据链路检查以太网类型字段，将每个帧送到相应的IP模块。IPv4模块可能还通过其上的TCP模块，检测到IPv4数据报的目的套接口是一个IPv6套接口，于是将其IPv4头部中的源IPv4地址转换成一个等价的IPv4映射的IPv6地址。当accept调用将接受的IPv4客户的连接返回给服务器时，这个映射后的地址作为客户的IPv6地址返回到IPv6套接口中。这个连接上其余的数据报全是IPv4数据报。

当accept调用将接受的IPv6客户的连接返回给服务器时，该客户的IPv6地址就是出现在IPv6头部中的源地址，不做任何改动。这个连接上其余的数据报全是IPv6数据报。我们可以把一个IPv4 TCP客户与一个IPv6服务器之间进行通信的步骤总结如下：

1. 启动IPv6服务器，创建一个IPv6的监听套接口，我们假定该套接口绑定了通配地址。
2. IPv4客户调用gethostbyname找到一个与该服务器对应的A记录(图9.5)。因为这台服务器主机同时支持IPv4和IPv6，所以它应该既有一个A记录，又有一个AAAA记录，但IPv4的客户只需要一个A记录。
3. 客户进程调用connect，客户主机向服务器发送一个IPv4的SYN。
4. 服务器主机收到这个发往IPv6监听套接口的IPv4 SYN，置一个标志，表明这个连接使用IPv4映射的IPv6地址，然后响应一个IPv4的SYN/ACK。当这个连接建立后，accept返回给服务器的地址就是这个IPv4映射的IPv6地址。
5. 在客户和服务器之间的所有通信使用IPv4数据报。
6. 除非服务器明确地去检查这个IPv6地址是不是一个IPv4映射的IPv6地址(使用在10.4节中介绍的IN6_IS_ADDR_V4MAPPED宏)，它将不会知道通信的对方是一个IPv4客户。双重协议栈屏蔽了这个细节。同样，IPv4的客户也不知道与之通信的是一个IPv6的服务器。

这里有一个假设，就是其中拥有双重协议栈的服务器主机既有一个IPv4地址，又有一个IPv6地址。在所有IPv4地址耗尽之前这是可行的。

对于一个IPv6的UDP服务器来说，情形是类似的，但每个数据报都将改变一次地址格式。举例来说，如果一个IPv6服务器收到一个从IPv4客户来的数据报，由recvfrom返回的地址将是该客户的IPv4映射的IPv6地址。服务器使用这个IPv4映射的IPv6地址，调用sendto对客户的请求作出响应。这种地址格式告诉内核向客户发送IPv4数据报。但服务器收到的下一个数据报可能是IPv6数据报，recvfrom将返回其IPv6地址。如果服务器响应，内核将生成一个IPv6数据报。

图10.3总结了在一个双重协议栈的主机上，根据接收套接口的类型(TCP或UDP)，对一个收到的IPv4或IPv6数据报进行处理的流程。

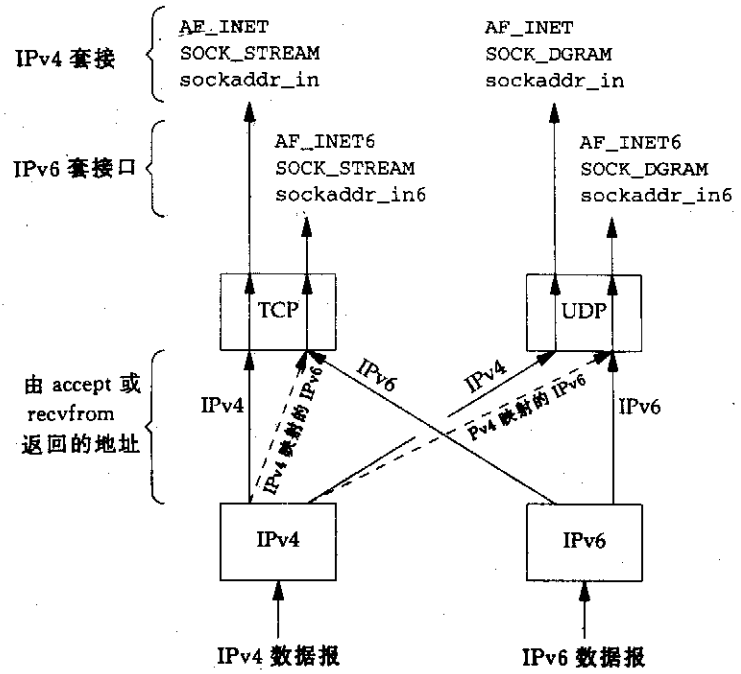


图 10.3　根据接收套接口的类型，对收到的 IPv4 或 IPv6 数据报的处理

- 如果在一个 IPv4 的套接口上收到了 IPv4 数据报，不需作任何特殊处理。在图中有两个标为“IPv4”的箭头，一个到 TCP，一个到 UDP。客户和服务器之间交换 IPv4 数据报。
- 如果在一个 IPv6 的套接口上收到了 IPv6 数据报，也不需作任何特殊处理。在图中有两个标为“IPv6”的箭头，一个到 TCP，一个到 UDP。客户和服务器之间交换 IPv6 数据报。
- 但当在一个 IPv6 套接口上收到 IPv4 数据报时，系统内核会在 accept(TCP)或 recvfrom(UDP)调用返回地址时将对应的 IPv4 映射的 IPv6 地址返回，这就是在图中的两个虚线箭头。因为每个 IPv4 地址都能表示成一个 IPv6 地址，所以这种映射是可行的。在客户和服务器之间交换的是 IPv4 数据报。
- 与上一条相反的过程是不存在的：通常一个 IPv6 地址不能表示成一个 IPv4 地址；因而没有从 IPv6 协议框到两个 IPv4 套接口的箭头。

大多数双重协议栈主机应使用以下规则处理监听套接口：

1. 监听 IPv4 套接口只能接受来自 IPv4 客户的外来连接。
2. 如果服务器有一个绑定在通配地址上的监听 IPv6 套接口，该套接口就能接受来自 IPv4 客户或 IPv6 客户的外来连接。对于来自 IPv4 客户的连接，其在服务器端的本地地址是对应的 IPv4 映射的 IPv6 地址。
3. 如果服务器有一个绑定在非 IPv4 映射的 IPv6 地址上的监听套接口，该套接口就只能接受来自 IPv6 客户的外来连接。

10.3 IPv6客户与IPv4服务器

我们现在把上一节例子中客户和服务器使用的协议掉一下个儿。首先考虑一个在双重协议栈主机上运行的IPv6 TCP客户。

1. 一个IPv4服务器在只有IPv4协议栈的主机上启动，创建了一个IPv4的监听套接口。
2. IPv6客户启动，调用gethostbyname只询问IPv6地址(它打开了RES_USE_INET6选项)。因为IPv4服务器主机只有A记录，从图9.5可以看到客户将得到一个IPv4映射的IPv6地址。
3. IPv6客户调用connect，在相应的IPv6套接口地址结构中存放所得到的IPv4映射的IPv6地址。系统内核检查到这个映射地址，于是自动向服务器发送一个IPv4的SYN。
4. 服务器用IPv4的SYN/ACK响应，连接通过IPv4数据报建立。

我们可用图10.4总结一下这种情况：

- 如果IPv4的TCP客户调用connect时或IPv4的UDP客户调用sendto时指定的是一个IPv4地址，不需要作任何特殊处理。这就是图中标为"IPv4"的两个箭头。
- 如果IPv6的TCP客户调用connect时或IPv6的UDP客户调用sendto时指定的是一个IPv6地址，不需要作任何特殊处理。这就是图中标为"IPv6"的两个箭头。

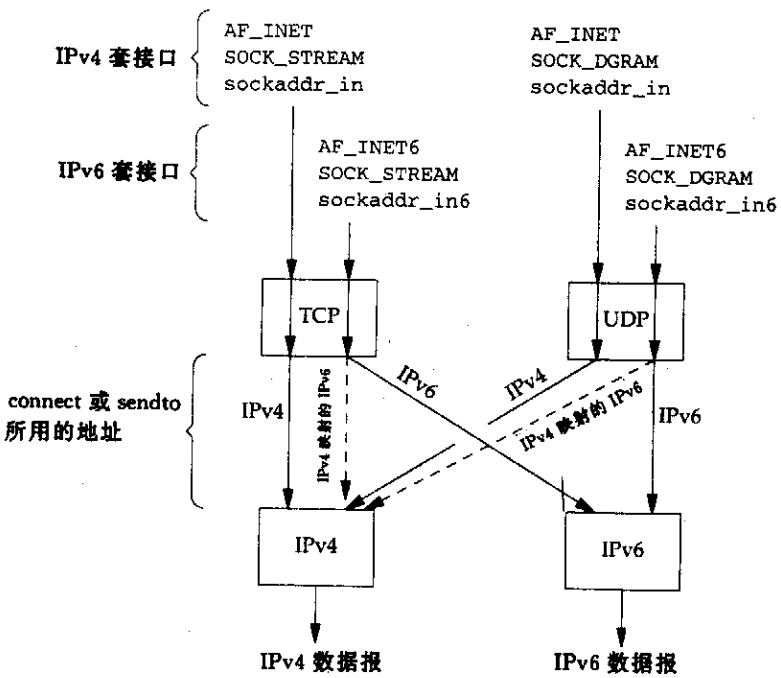


图10.4 根据地址类型和套接口类型，对客户请求的处理

- 如果 IPv6 的 TCP 客户调用 connect 时，或 IPv6 的 UDP 客户调用 sendto 时指定的是一个 IPv4 映射的 IPv6 地址，内核会检测到这个映射地址，并发送一个 IPv4 数据报，而不是 IPv6 数据报。这是图中的两条虚线箭头。
- IPv4 客户不能在调用 connect 或 sendto 时指定一个 IPv6 地址，这是因为在 IPv4 sockaddr_in 结构里的 4 字节的 in_addr 结构中，放不下一个 16 字节的 IPv6 地址。因此图中从 IPv4 套接口到 IPv6 协议框之间没有箭头。

在前一节中（一个 IPv4 的数据报到达 IPv6 服务器套接口），将 IPv4 地址转换成 IPv4 映射的 IPv6 地址是由内核完成的，accept 或 recvfrom 将其透明地返回给应用进程。在这一节中（要在一个 IPv6 套接口上发送 IPv4 数据报），IPv4 地址到 IPv4 映射的 IPv6 地址的转换是由如图 9.5 中的解析器完成的，应用进程将映射后的地址透明地传递给 connect 或 sendto。

对互操作性的总结

图 10.5 对本节和上一节内容，以及客户与服务器的组合情况作了总结。

	IPv4 服务器 IPv4 主机 (A)	IPv6 服务器 IPv6 主机 (AAAA)	IPv4 服务器 双重协议栈主机 (A 和 AAAA)	IPv6 服务器 双重协议栈主机 (A 和 AAAA)
IPv4 客户，IPv4 主机	IPv4	(no)	IPv4	IPv4
IPv6 客户，IPv6 主机	(no)	IPv4	(no)	IPv4
IPv4 客户，双重协议栈主机	IPv4	(no)	IPv4	IPv4
IPv6 客户，双重协议栈主机	IPv4	IPv6	(no*)	IPv6

图 10.5　对 IPv4 和 IPv6 客户与服务器之间互操作性的总结

栏中的“IPv4”或“IPv6”表示这种组合是可互操作的，而且指出了实际通信所使用的协议。栏中的“(no)”则表示对应的组合是不能互操作的。最后一行的第三列标有一个“*”号，是因为其互操作性与客户选择的地址有关。选择 AAAA 记录发送 IPv6 数据报将不能互操作。但选择 A 记录，给客户返回一个 IPv4 映射的 IPv6 地址，因而发送 IPv4 数据报，将能正常工作。

尽管在表中有四分之一的情况不能互操作，实际上在可预见的将来，大多数 IPv6 的实现将是在双重协议栈主机上而不仅仅是在 IPv6 协议栈主机上。如果我们删去表中的第二行和第二列，所有的“(no)”将消失，只剩下一个有问题的“(no)*”。

10.4　IPv6 地址测试宏

有一小部分 IPv6 应用程序必须知道与其通信的对方是否使用 IPv4 协议。这些应用程序需要知道对方的地址是不是一个 IPv4 映射的 IPv6 地址。为测试 IPv6 地址的某些特性共定义了 12 个宏。

```
#include <netinet/in.h>

int IN6_IS_ADDR_UNSPECIFIED(const struct in6_addr *aptr);
int IN6_IS_ADDR_LOOPBACK(const struct in6_addr *aptr);
int IN6_IS_ADDR_MULTICAST(const struct in6_addr *aptr);
int IN6_IS_ADDR_LINKLOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_SITELOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_V4MAPPED(const struct in6_addr *aptr);
int IN6_IS_ADDR_V4COMPAT(const struct in6_addr *aptr);

int IN6_IS_ADDR_MC_NODELOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_MC_LINKLOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_MC_SITELOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_MC_ORGLOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_MC_GLOBAL(const struct in6_addr *aptr);

                返回值:非零表示IPv6地址是指定类型的,否则返回零
```

前7个宏测试IPv6地址的基本类型。我们在A.5节中介绍了这些不同的地址类型。后5个宏测试一个IPv6多播地址的范围(19.2节)。

IPv6客户可以用IN6_IS_ADDR_V4MAPPED来测试由解析器返回的IPv6地址。IPv6服务器也可以用这个宏来测试由accept或recvfrom返回的IPv6地址。

我们可以把FTP和它的PORT指令作为需要使用这个宏的一个例子。如果启动一个FTP客户,登录到FTP服务器上,并发出FTP dir命令,FTP客户会在控制连接上向FTP服务器发送一个PORT指令。这条指令告诉服务器客户的IP地址和端口号,服务器将使用这些参数向客户发起建立一个数据连接。(TCPv1的第27章中包含FTP应用协议的所有细节。)但IPv6的FTP客户必须知道服务器是一个IPv4的服务器还是IPv6的服务器,因为前者需要一个"PORT a1,a2,a3,a4,P1,P2"格式的指令,其中前四个数字(每个都在0～255之间)组成一个4字节的IPv4地址,后两个数字组成2字节的端口号。而一个IPv6的服务器需要一个LPRT指令(参见RFC 1639 [Piscitello 1994]),其中包含21个数字。习题10.1中给出了这两种指令的例子。

10.5 IPV6_ADDRFORM套接口选项

IPV6_ADDRFORM套接口选项能把一个套接口从一种类型转变成另一种类型,它受到以下限制:

1. 一个IPv4套接口总能转变为IPv6套接口。所有与该套接口关联的IPv4地址被转换成IPv4映射的IPv6地址。
2. 一个IPv6套接口只有在与之关联的地址是IPv4映射的IPv6地址的情况下才能转变成IPv4套接口。

为什么要转变套接口的地址格式呢?其原因是在UNIX上文件描述字可以在进程之间进行传递。最通常的方式是通过fork,我们也将看到文件描述字是怎样在有亲缘关系的进程(14.7节)和无亲缘关系的进程(25.7节)之间传递的。

举例来说，假设一个进程创建了一个 IPv4 的监听套接口，然后接收到一个来自 IPv4 客户的连接。该服务器调用 fork 和 exec，启动一个新程序处理客户请求。这里有个约定，即已连接套接口通过标准输入、标准输出和标准错误输出传递给新程序（与 inetd 所做的相似，见 12.5 节），于是就有图 10.6 中的一段伪代码。它与 4.8 节中的并发服务器的唯一不同是将已连接套接口复制到事先约定的描述字，然后调用 exec。

但被加载的新程序期望一个 IPv6 套接口。我们可用 IPV6_ADDRFORM 套接口选项来转换该套接口的地址格式，如图 10.7 所示。

```
int                 listenfd,connfd;
socklen_t           clilen;
struct sockaddr_in  serv,cli;                    /* IPv4 structs */

listenfd = Socket(AF_INET,SOCK_STREAM,0);     /* IPv4 socket */

/* fill in serv() with well-known prot */
Bind(listenfd,&serv,sizeof(serv));
Listen(listenfd,LISTENQ);

for ( ; ; ) {
    clilen = sizeof(cli);
    connfd = Accept(listenfd,&cli,&clilen);

    if(Fork() == 0) {
        Close(listenfd);              /* child */
        Dup2(connfd,STDIN_FILENO);
        Dup2(connfd,STDOUT_FILENO);
        Dup2(connfd,STDERR_FILENO);
        Close(connfd);
        Exec( ... );                  /* start new program */
    }
    Close(connfd);                    /* parent */
}
```

图 10.6　接受外来连接并启动新程序的服务器程序

```
int                 af;
socklen_t           clilen;
struct sockaddr_in6 cli;          /* IPv6 struct */
struct hostent      *ptr;

af = AF_INET6;
Setsockopt(STDIN_FILENO,IPPROTO_IPV6,IPV6_ADDRFORM,&af,sizeof(af));

clilen = sizeof(cli);
Getpeername(0,&cli,&clilen);

ptr = gethostbyaddr(&cli.sin6_addr, 16, AF_INET6);
```

图 10.7　将一个 IPv4 套接口转换成 IPv6 套接口

假设该套接口是一个 IPv4 套接口，调用 setsockopt 将把套接口的地址格式由 IPv4 变为 IPv6，getpeername 则会返回一个 IPv4 映射的 IPv6 地址。但是如果该套接口是一个 IPv6 套接口，那么调用 setsockopt 将不起作用，因为地址格式已是 IPv6 了。

一种可以使用该套接口选项的情况是，接收外来 IPv4 连接的程序是由他人提供的（即没有源码，我们无法修改使之适用于 IPv6，当然协议无关则更好），但我们的被加载程序是

来处理 IPv6 的。

如果用 IPV6_ADDRFORM 作参数来调用 getsockopt，返回值依赖于套接口地址的格式，将为 AF_INET 或 AF_INET6。getsockopt 和 setsockopt 的第二个参数可以是 IPPROTO_IP 或 IPPROTO_IPV6。

10.6 源代码可移植性

大部分现有的网络应用程序是为 IPv4 编写的。为此将分配和填写 sockaddr_in 结构，并且 socket 调用会指定 AF_INET 作为其第一个参数。从图 1.5 到图 1.6 的转换中可以看出，把这些 IPv4 应用程序转换成能用于 IPv6 并不麻烦。很多修改都可用编辑脚本自动完成。一些对 IPv4 依赖性很强的程序，如使用多播、IP 选项或原始套接口功能的程序，将要花费更多的转换工作。

如果在源代码级对应用程序进行向 IPv6 的移植和发布，就不得不考虑接受者的系统是否支持 IPv6。典型的方法是在代码中使用 #ifdef 伪码，以尽可能使用 IPv6（因为我们已在本章中看到，IPv6 客户仍能与 IPv4 服务器通信，反之亦然）。这种方法存在的问题是代码会随着 #ifdef 的使用很快变得混乱，难于理解和维护。

更好的方法是把这种向 IPv6 的转换看成是一个使程序独立于协议的机会。第一步是将 gethostbyname 和 gethostbyaddr 调用删除，转而使用在下一章中描述的 getaddrinfo 和 getnameinfo 函数。这使我们将套接口地址结构作为不透明的对象来处理，就像 bind、connect、recvfrom 等基本的套接口函数所做的那样，由一个指针和大小来引用。3.8 节中的 sock_XXX 函数能帮助我们在独立于 IPv4 和 IPv6 的情况下处理这些问题。显然这些函数中包含 #ifdef 来处理 IPv4 和 IPv6，但是将所有对协议的依赖隐蔽到几个库函数中，编码会更简单。在 19.6 节中我们将开发一组 mcast_XXX 函数，它们能使多播的应用程序独立于 IPv4 或 IPv6。

另一点需要考虑的是，如果在一个同时支持 IPv4 和 IPv6 的系统上编译源代码，发布其执行码或目标文件（不是源代码），并在不支持 IPv6 的系统上运行这些程序时会发生什么。这有一种可能，即本地的名字服务器支持 AAAA 记录，并对这些程序要连接的主机同时返回 AAAA 记录和 A 记录。如果一个 IPv6 的应用程序在一个不支持 IPv6 的主机上调用 socket 创建 IPv6 套接口，将会失败。在下一章里介绍的函数能处理这种情况，它们将忽略 socket 产生的错误，并尝试使用名字服务器返回的地址表中的下一个地址。假设对方有一个 A 记录，并且名字服务器在返回所有 AAAA 记录后附加一个 A 记录的话，创建 IPv4 套接口将会成功。这种类型的功能应由某个库函数提供，而不是每个应用程序的源代码都提供。

10.7 小 结

在双重协议栈主机上的 IPv6 服务器能为 IPv4 和 IPv6 客户服务。IPv4 客户向服务器发送的仍是 IPv4 数据报，但服务器的协议栈会把客户方的地址转换成一个 IPv4 映射的 IPv6 地址，因为 IPv6 服务器处理的是 IPv6 套接口地址结构。

类似地，在双重协议栈主机上的 IPv6 客户能够和 IPv4 服务器通信。客户的解析器代码将给服务器主机的所有 A 记录返回一个 IPv4 映射的 IPv6 地址，在双重协议栈客户主机上调用 connect 与这些地址建立连接时，由双重协议栈发送一个 IPv4 SYN 分节。只有少数几个特殊的客户和服务器程序需要知道对方使用的协议版本（例如 FTP），可以用 IN6_IS_ADDR_V4MAPPED 宏来判断对方是否使用 IPv4。程序可用 IPV6_ADDRFORM 套接口选项得到自己期望的套接口类型（通常是 IPv6 套接口）。

10.8　习　题

1. 在一台双重协议栈主机上启动 IPv6 FTP 客户。连接一个 IPv4 FTP 服务器，发出 debug 命令，然后是 dir 命令。再对一个 IPv6 FTP 服务器作同样的操作，比较由 dir 命令所发出的 PORT 指令。
2. 编写需要一个 IPv4 点分十进制地址作为命令行参数的程序。它创建一个 IPv4 TCP 套接口，并将这个地址和某个端号口，如 8888，捆绑到该套接口。调用 listen，然后调用 pause。编写一个类似的程序，但其命令行参数是一个 IPv6 的十六进制数串，创建的是监听 IPv6 TCP 套接口。用通配地址作参数运行 IPv4 程序。然后在另一个窗口里用 IPv6 通配地址作参数启动 IPv6 程序。因为 IPv4 程序已经绑定该端口，还能启动 IPv6 程序吗？使用 SO_REUSEADDR 套接口选项会有什么不同吗？如果先启动 IPv6 程序，再启 IPv4 程序又会如何？

第11章 高级名字与地址转换

11.1 概 述

在第9章中介绍的两个函数:gethostbyname 和 gethostbyaddr,是依赖于协议的。使用前一个函数时,我们必须知道放置结果的套接口地址结构的成员是哪一种(举例来说,IPv4 使用 sin_addr 成员,IPv6 使用 sin6_addr 成员),而调用后一个函数时,必须知道存放二进制地址的是哪一种成员。这一章以能为应用程序提供协议独立性的新的 Posix.1g 的 getaddrinfo 函数作为开头,后面将介绍跟它互补的 getnameinfo。

在这之后我们将用该函数开发6个自己的函数,以处理 TCP 或 UDP 客户和服务器程序的典型情况。后面的文章中将会使用这些函数,而不是直接调用 getaddrinfo。

gethostbyname 和 gethostbyaddr 也是不可重入函数的好例子。我们将解释为什么会是这样,并介绍一些替代函数来避免这个问题。重入问题我们将在第23章中再讨论,但在没有详细了解线程的情况下,我们也能指出和解释这个问题。

本章将以展示 getaddrinfo 的完整实现来结束,这将让我们对该函数有更多的理解:它是怎样操作的,返回什么,如何与 IPv4 和 IPv6 交互。

11.2 getaddrinfo 函数

函数 getaddrinfo 在库函数中隐藏了所有协议依赖性。应用程序只需要处理由 getaddrinfo 填写的套接口地址结构。该函数在 Posix.1g 中定义。

Posix.1g 对这个函数的定义来源于 Keith Sklower 早先提出的名为 getconninfo 的函数。这个函数是与 Eric Allman、Walliam Durst、Michael Karels 以及 Steven Wise 讨论的结果,最早的实现是由 Eric Allman 写的。指定主机名和服务名将足以与一个独立于协议细节的服务建立连接,Marshall Rose 在对 X/Open 的一个提议中作出了对此的观测。

```
#include <netdb.h>
int getaddrinfo (const char *hostname,const char *service,
                 const struct addrinfo *hints,struct addrinfo **result);
                                        返回:成功返回0,出错返回非零(见图11.3)
```

这个函数通过 result 指针返回一个指向 addrinfo 结构链表的指针,该结构在<netdb.h>中定义:

```
struct addrinfo {
  int              ai_flags;      /* AI_PASSIVE,AI_CANONNAME */
  int              ai_family;     /* AF_xxx */
  int              ai_socktype;   /* SOCK_xxx */
```

```
    int             ai_protocol;    /* 0 or IPPROTO_xxx for IPv4 and IPv6 */
    size_t          ai_addrlen;     /* length of ai_addr */
    char            *ai_canonname;  /* ptr to canonical name for host */
    struct sockaddr *ai_addr;       /* ptr to socket address structure */
    struct addrinfo *ai_next;       /* ptr to next structure in linked list */
};
```

其中的 hostname 是主机名或地址串(IPv4 的点分十进制数表示或 IPv6 的十六进制数串)。service 是服务名或十进制数的端口号字符串。(回想在习题 9.6 中,我们是如何解决允许用地址串作为主机名,用端口号串作为服务名的。)

hints 是一个空指针或指向一个 addrinfo 结构的指针,由调用者填写关于它所想返回的信息类型的线索。举例来说,如果某种服务既可以用 TCP 又可以用 UDP 提供(譬如,由 DNS 服务器提供的 domain 服务),调用者可以将 hints 结构中的 ai_socktype 成员的值设为 SOCK_DGRAM,返回的信息就只适用于数据报套接口。

调用者可以设置的 hints 结构的成员有:

- ai_flags(AI_PASSIVE, AI_CANONNAME)
- ai_family(某个 AF_xxx 值)
- ai_socktype(某个 SOCK_xxx 值),和
- ai_protocol

AI_PASSIVE 标志表示该套接口是用作被动的打开,AI_CANONNAME 标志则通知 getaddrinfo 函数返回主机的名字。

如果 hints 是一个空指针,该函数将假定 ai_flag、ai_socktype 和 ai_protocol 的值为零,ai_family 的值为 AF_UNSPEC。

如果函数返回成功(0),result 参数指向的变量将被填入一个指针,它指向一个通过 ai_next 指针串起来的 addrinfo 结构链表。返回这个复合结构有两种方式。

1. 如果与该 hostname 对应的有多个地址,将按请求的地址族(如果指定了 ai_family 线索)为每个地址返回一个结构。
2. 如果该服务在多种套接口类型上提供,将根据 ai_socktype 线索为每个套接口类型返回一个结构。

举例来说,如果在没有提供任何线索的条件下,要求 domain 服务查找一个有两个 IP 地址的主机,将返回 4 个 addrinfo 结构:

- 一个是第一个 IP 地址和 SOCK_STREAM 套接口类型
- 一个是第一个 IP 地址和 SOCK_DGRAM 套接口类型
- 一个是第二个 IP 地址和 SOCK_STREAM 套接口类型
- 一个是第二个 IP 地址和 SOCK_DGRAM 套接口类型

图 11.1 中是这个例子的示意图。这里并不能保证返回的多个条目的结构顺序;也就是说,不能假定 TCP 服务在 UDP 服务之前返回。

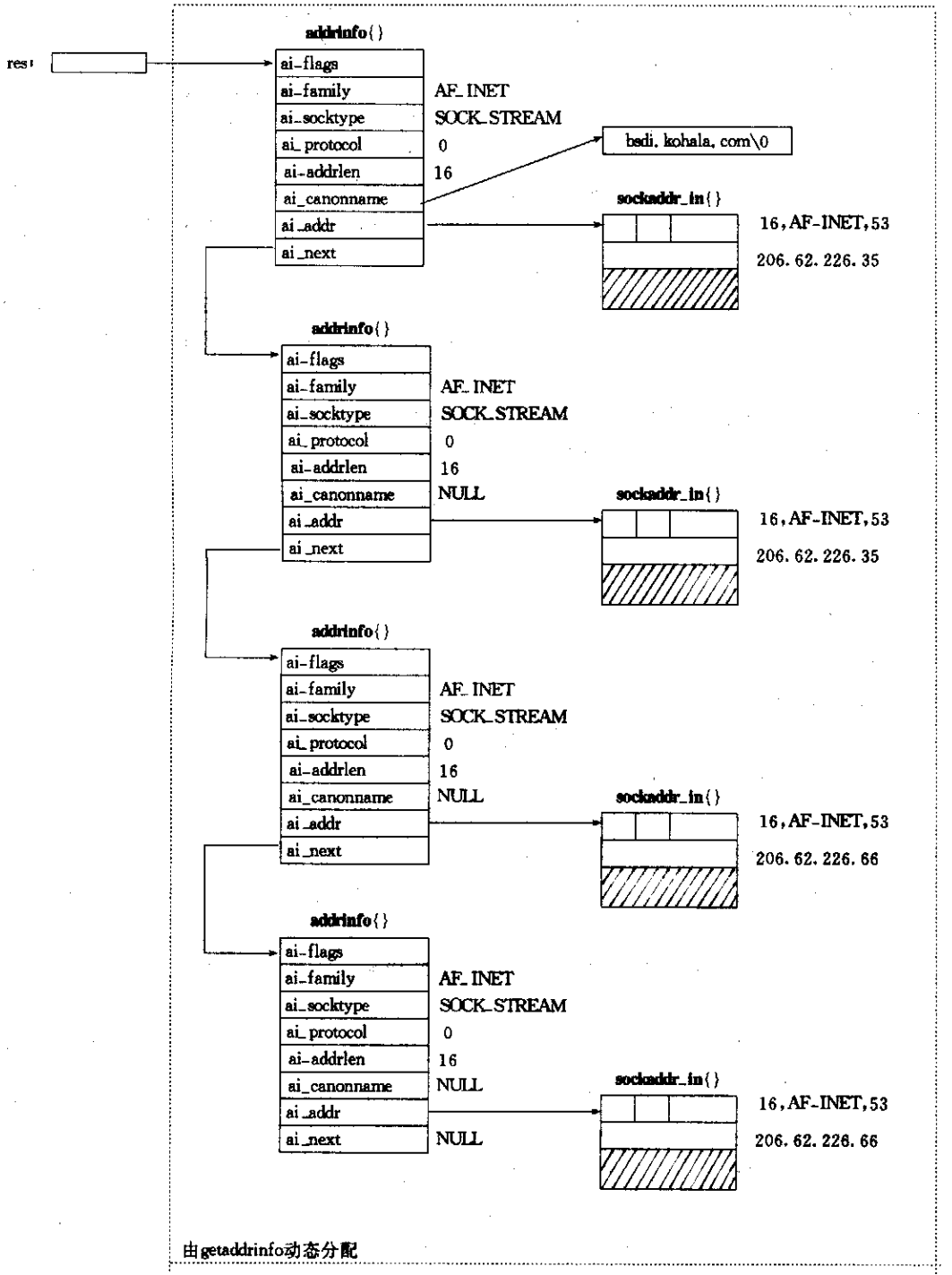


图 11.1 getaddrinfo 返回的信息的实例

尽管没有保证，但在具体实现中应按 DNS 返回的顺序返回 IP 地址。举例来说，许多 DNS 服务器把返回的地址排序，这样如果发出查询请求的主机与名字服务器在同一网络上，在这一共享网络上的地址将首先返回。更新版本的

BIND 也允许解析器在/etc/resolv.conf 文件中指定地址的排序顺序。

在 addrinfo 结构中返回的信息可用于调用 socket，然后调用 connect、sendto（客户）或 bind（服务器）。socket 函数的参数是 addrinfo 结构中的 ai_family、ai_socktype 和 ai_addr。connect 或 bind 函数的第二个和第三个参数是 ai_addr（一个指向适当类型的套接口地址结构的指针，由 getaddrinfo 填写）和 ai_addrlen（套接口地址结构的大小）。

如果设置了在 hints 结构中的 AI_CANONNAME 标志，返回的第一个结构的 ai_canonname 成员指向相应主机的名字。DNS 的术语通常称之为 FQDN。

图 11.1 给出了执行下列程序片段时返回的信息。

```
struct addrinfo     hints, *res;
bzero(&hints,sizeof(hints));
hints.ai_flags = AI_CANONNAME;
hints.ai_family = AF_INET;
getaddrinfo("bsdi","domain",&hints,&res);
```

在图中除 res 变量外的数据空间都是动态分配的（譬如用 malloc），我们假设主机 bsdi 的别名是 bsdi.kohala.com，而且在 DNS 中它有两个 IPv4 地址（图 1.16）。

端口 53 用于 domain 服务，而且在套接口地址结构中的这个端口号是网络字节序的。返回的 ai_protocol 的值为 0，这是因为 ai_family 和 ai_socktype 的组合完全指定了协议为 TCP 和 UDP。getaddrinfo 在其中的两个 SOCK_STREAM 结构的 ai_protocol 中返回 IPPROTO_TCP，在其中的两个 SOCK_DGRAM 结构的 ai_protocol 中返回 IPPROTO_UDP 也是可以的。

图 11.2 总结了根据指定的服务名（可以是一个十进制数的端口号）和 ai_socktype 线索，为每个地址返回的 addrinfo 结构的数目。

ai_socktype 线索	服务以名字标识，它的提供者为：			服务以端口号标识
	TCP	UDP	TCP 与 UDP	
0	1	1	2	2
SOCK_STREAM	1	错误	1	1
SOCK_DGRAM	错误	1	1	1

图 11.2　为每个 IP 地址返回的 addrinfo 结构的数目

只有在未提供 ai_socktype 线索且或者服务以名字标识，并同时由 TCP 和 UDP 提供支持时（在/etc/services 文件中指明），或者服务以端口号标识时，才会为每个 IP 地址返回多个 addrinfo 结构。

如果枚举 getaddrinfo 的 64 种可能的输入（有六个输入变量），许多都是无效的，有些没什么意义。因而我们将注意力集中到一些常见情况。

- 指定 hostname 和 service。这在 TCP 或 UDP 客户程序中很常见。TCP 客户程序遍历所有返回的 IP 地址，逐一调用 socket 和 connect，直到连接成功或所有地址被试过为止。在图 11.6 中我们开发的 tcp_connect 函数是一个例子。

  在 UDP 客户程序中，由 getaddrinfo 填写的套接口地址结构被用来调用 sendto 或 connect。如果第一个地址不行（在已连接的 UDP 套接口上出错或在未连接的套接口上超时），就会尝试剩下的地址。

  如果客户程序知道它只处理一种类型的套接口（例如，Telnet 和 FTP 客户程序只处

理 TCP，TFTP 客户程序只处理 UDP），就应把 hints 结构中的 ai_socket 设为 SOCK_STREAM 或 SOCK_DGRAM。

- 一个典型的服务器程序只用指定 service 以及 hints 结构中的 AI_PASSIVE 标志，而不需要指明 hostname。返回的套接口地址结构中应包含一个 INADDR_ANY（IPv4）或 IN6ADDR_ANY_INIT（IPv6）的 IP 地址。TCP 服务器程序随后调用 socket、bind 和 listen。如果服务器程序要 malloc 另一个套接口地址结构以从 accept 取得客户的地址，返回的 ai_addrlen 的值将给出所需的内存大小。

  UDP 的服务器程序将调用 socket、bind 和 recvfrom。如果服务器要 malloc 另一个套接口地址结构以从 recvfrom 取得客户的地址，返回的 ai_addrlen 的值将给出所需的内存大小。

  和典型的客户程序代码一样，如果服务器程序知道它只需处理一种类型的套接口，hints 结构中的 ai_socket 应被设为 SOCK_STREAM 或 SOCK_DGRAM。这样可以避免返回多重结构，其中可能出现错误的 ai_socktype 值。
- 到目前为止，我们展示的 TCP 服务器程序都只创建一个监听套接口，UDP 服务器只创建一个数据报套接口。我们在上一个条目中就是这么假设的。另一种服务器程序设计方法是用 select 处理多个套接口。这种情况下服务器将遍历由 getaddrinfo 所返回的整个结构链表，并为每个结构创建一个套接口，然后使用 select。

  这种技术的问题是，getaddrinfo 返回多重结构的一个原因是该服务能同时使用 IPv4 和 IPv6（图 11.4）。但就像在 10.2 节中所看到的，这两种协议并不是完全独立的。也就是说，如果在给定的端口上创建了一个 IPv6 监听套接口，就没有必要在同一端口上再创建 IPv4 套接口了，因为协议栈和 IPv6 监听套接口能自动处理由 IPv4 客户发出的连接。

尽管 getaddrinfo 比 gethostbyname 和 getservbyname 函数更“好”（它使我们能更容易地编写独立于协议的代码，单个函数既处理了主机名又处理了服务，而且所有返回的信息是动态分配的），但它仍不是太好用。问题在于必须分配一个 hints 结构，初始化为 0，填上必要的字段，再调用 getaddrinfo，然后遍历链表逐一尝试。在下一节中会给典型的 TCP 和 UDP 客户或服务器程序提供一些更简单的接口，以在后文中使用。

getaddrinfo 解决了将主机名和服务名转换成套接口地址结构的问题。在 11.13 节中将介绍一个功能与其相反的函数 getnameinfo，它把套接口地址结构转换成主机名和服务名。在 11.16 节中将提供一个 getaddrinfo、getnameinfo 和 freeaddrinfo 的实现。

11.3 gai_strerror 函数

getaddrinfo 返回的非零错误值的名字和含义如图 11.3 所示。gai_strerror 以这些值作为它的一个参数，返回指向对应的出错信息字符串的指针。

```
#include <netdb.h>
char *gai_strerror(int error);
```

返回：一个指向描述出错信息字符串的指针

常　量	描　述
EAI_ADDRFAMILY	不支持 hostname 的地址族
EAI_AGAIN	名字解析中的暂时失败
EAI_BADFLAGS	ai_flags 的值无效
EAI_FAIL	名字解析中不可恢复的失败
EAI_FAMILY	不支持 ai_family
EAI_MEMORY	内存分配失败
EAI_NODATA	没有与 hostname 相关联的地址
EAI_NONAME	hostname 或 service 未提供，或者不可知
EAI_SERVICE	不支持 ai_socktype 类型的 service
EAI_SOCKTYPE	不支持 ai_socktype
EAI_SYSTEM	errno 中有系统错误返回

图 11.3　getaddrinfo 返回的非 0 错误常值

11.4　freeaddrinfo 函数

由 getaddrinfo 返回的存储空间，包括 addrinfo 结构、ai_addr 结构和 ai_canonname 字符串，都是用 malloc 动态获取的。这些空间可调用 freeaddrinfo 释放。

```
#include <netdb.h>
void freeaddrinfo(struct addrinfo *ai);
```

ai 应指向 getaddrinfo 返回的第一个 addrinfo 结构。在该链表中的所有结构，以及这些结构所指向的动态存储空间都将被释放（譬如套接口地址结构和规范主机名）。

假设我们调用 getaddrinfo，顺着 addrinfo 结构链表找到所需的结构。然后只复制该 addrinfo 结构以保存其信息，再调用 freeaddrinfo，就会产生一个潜藏的错误。原因是 addrinfo 结构中的指针指向动态分配的内存（用于存放地址结构和规范主机名），因此由我们保存的结构所指向的内存在调用 freeaddrinfo 后就释放，可能将作它用。

只复制 addrinfo 结构，而不复制 addrinfo 结构所指向的其他结构，叫做浅拷贝或浅复制（shallow copy）。复制 addrinfo 结构，同时复制 addrinfo 结构所指向的其他结构，称为深拷贝或深复制（deep copy）。

11.5 getaddrinfo 函数:IPv6 和 UNIX 域

尽管 Posix.1g 定义了 getaddrinfo 函数,但它没有专门对 IPv6 做任何说明。该函数与名字解析(特别是 RES_USE_INET6 选项,参见图 9.5),以及 IPv6 之间的交互是不平常的。在用图 11.4 对这些交互作出总结前,我们注意以下几点:

- getaddrinfo 处理两种不同的输入:调用者想返回的套接口地址结构类型和应在 DNS 中查找的记录类型。
- 由调用者提供的 hints 结构中的地址协议族,指明了它想返回的套接口地址结构类型。如果其值指定为 AF_INET,就不会返回任何 sockaddr_in6 结构;如果是 AF_INET6,则不会返回任何 sockaddr_in 结构。
- Posix.1g 中提到如指定为 AF_UNSPEC,则应返回可用于该主机名和服务名的所有协议族的地址。这意味着,如果某主机既有 AAAA 记录,又有 A 记录,则将返回一个 sockaddr_in6 结构的 AAAA 记录的和一个 sockaddr_in 结构的 A 记录。将 A 记录转换成一个 IPv4 映射的 IPv6 地址也用 sockaddr_in6 结构返回是没有什么意义的,因为它并没有提供任何额外信息:这些地址已在 sockaddr_in 结构中返回了。
- Posix.1g 中的这个说明也暗示,如果设置了 AI_PASSIVE 标志,而未指定主机名,则将返回一个值为 IPv6 通配地址(IN6ADDR_ANY_INIT 或 0::0)的 sockaddr_in6 结构和一个值为 IPv4 通配地址(INADDR_ANY 或 0.0.0.0)的 sockaddr_in 结构。在 10.2 节中我们看到 IPv6 服务器在双重协议栈主机上对 IPv6 和 IPv4 的客户都能处理,所以首先返回 IPv6 通配地址是有其意义的。
- 解析器选项 RES_USE_INET6 和所调用的函数(gethostbyname 或 gethostbyname2)一块确定在 DNS 中查找的记录类型(A 或 AAAA),以及返回的地址类型(IPv4、IPv6 或 IPv4 映射的 IPv6)。图 9.5 对此作了总结。
- 主机名还可以是 IPv6 十六进制数串或 IPv4 点分十进制数串。该串的有效性取决于调用者指定的地址族。如果指定 AF_INET,就不能接受 IPv6 十六进制数串;如果指定 AF_INET6,则不能接受 IPv4 点分十进制数串。如果指定的是 AF_UNSPEC,那么两种数串都可以,返回的是对应类型的套接口地址结构。

  有人可能会争论说,如果指定了 AF_INET6,则对于点分十进制数串应该用 sockaddr_in6 结构返回对应的 IPv4 映射的 IPv6 地址。但是有另外一种方法得到这种结果,那就是在点分十进制串前加上 0::ffff.

图 11.4 总结了 getaddrinfo 对 IPv4 和 IPv6 地址是怎样处理的。"结果"一栏是在给定前三栏的变量后,返回给调用者的结果。"行为"栏则说明怎样得到这个结果,我们将在 11.6 节 getaddrinfo 的实现中展示执行这些操作的代码。

调用者指定的主机名	调用者指定的地址族	主机名字符串包含	结　果	行　为
非空字符串；主动或被动	AF_UNSPEC	主机名	所有 AAAA 记录以 sockaddr_in6() 返回，所有 A 记录以 sockaddr_in() 返回	两次 DNS 查找(注 1)：RES_USE_INET6 关闭下 gethostbyname2(AF_INET6)；RES_USE_INET6 关闭下 gethostbyname2(AF_INET)
		十六进制数串	一个 sockaddr_in6()	inet_pton(AF_INET6)
		点分十进制数	一个 sockaddr_in()	inet_pton(AF_INET)
	AF_INET6	主机名	所有 AAAA 记录以 sockaddr_in6() 返回，否则所有 A 记录转成 IPv4 映射的 IPv6 地址以 sockaddr_in6() 返回	RES_USE_INET6 打开下 gethostbyname()(注 2)
		十六进制数串	一个 sockaddr_in6()	inet_pton(AF_INET6)
		点分十进制数	出错：EAI_ADDRFAMILY	
	AF_INET	主机名	所有 A 记录以 sockaddr_in() 返回	RES_USE_INET6 关闭下 gethostbyname()
		十六进制数串	出错：EAI_ADDRFAMILY	
		点分十进制数	一个 sockaddr_in()	inet_pton(AF_INET)
空串；被动	AF_UNSPEC	隐含 0::0 隐含 0.0.0.0	一个 sockaddr_in6() 和 一个 sockaddr_in()	inet_pton(AF_INET6) inet_pton(AF_INET)
	AF_INET6	隐含 0::0	一个 sockaddr_in6()	inet_pton(AF_INET6)
	af_INET	隐含 0.0.0.0	一个 sockaddr_in()	inet_pton(AF_INET)
空串；主动	AF_UNSPEC	隐含 0::1 隐含 127.0.0.1	一个 sockaddr_in6() 和 一个 sockaddr_in()	inet_pton(AF_INET6) inet_pton(AF_INET)
	AF_INET6	隐含 0::1	一个 sockaddr_in6()	inet_pton(AF_INET6)
	AF_INET	隐含 127.0.0.1	一个 sockaddr_in()	inet_pton(AF_INET)

图 11.4　对 getaddrinfo 函数的行为和结果的总结

表中注 1 是当执行这两次 DNS 查找时，它们都有可能失败(即找不到所给主机名的所需类型的记录)，但起码有一次会成功。如果两次查找都成功(该主机既有 AAAA 记录又有 A 记录)，则两种类型的套接口地址结构都将返回。

注 2 是 DNS 查找必须成功，否则将返回错误。但因为设置了 RES_USE_INET6 选项，gethostbyname 会先查 AAAA 记录，如果没找到，再查 A 记录(图 9.6)。

在这两个注所处的情形中，解析器的 RES_USE_INET6 选项的打开与关闭目的是按照图 9.5 中的规则，强制执行期望的 DNS 查找。

图 11.4 只说明了 getaddrinfo 是怎样处理 IPv4 和 IPv6 的；也就是返回给调用者的地址的数目。返回的 addrinfo 结构的实际数目还依赖于指定的套接口类型和服务名，像图 11.2 中总结的那样。

Posix.1g 没有对 getaddrinfo 和 Unix 域套接口作任何说明(在 14 章中将作详细介绍)。然而，在我们的 getaddrinfo 实现中加入对 Unix 域套接口的支持，并在这些协议上对应用程序进行测试就可以很好地检测协议独立性。

我们的实现作了以下的假设：如果 getaddrinfo 的主机名参数为/local 或/unix，而且服务名参数是绝对路径名(以斜杠开头)，则返回 Unix 域的套接口。有效的 DNS 主机名不能包含斜杠，也不存在以斜杠开头的 IANA 服务名(习题 11.5)。返回的套接口地址结构包含该绝对路径名，它用来调用 bind 或 connect。如果调用者设定了 AI_CANONNAME 标志，该主机的名字(9.7 节)将作为规范主机名返回。

11.6 getaddrinfo 函数：例子

现在用一个测试程序来展示 getaddrinfo 的一些例子，它运行时让我们输入所有的参数：主机名、服务名、地址族、套接口类型、AI_CANONNAME 和 AI_PASSIVE 标志。(这儿没有给出源程序，因为它大约有 350 行。如前言中所介绍的，它以源码形式提供。)测试程序输出返回的可变数目的 addrinfo 结构中的信息，包括调用 socket 所用的参数和每个套接口地址结构中的地址。

我们首先展示与图 11.1 同样的例子。

```
solaris % testga -f inet -c -h bsdi -s domain
socket(AF_INET, SOCK_STREAM,0),ai_canonname = bsdi.kohala.com
        address: 206.62.226.35.53
socket(AF_INET, SOCK_DGRAM, 0)
        address: 206.62.226.35.53
socket(AF_INET, SOCK_STREAM, 0)
        address: 206.62.226.66.53
socket(AF_INET, SOCK_DGRAM, 0)
        address: 206.62.226.66.53
```

-f inet 选项指定地址族，-c 表示返回规范主机名，-h bsdi 指定主机名，-s domain 指定服务名。

通常客户指定地址族、套接口类型(用-t 选项)、主机名和服务名。下面的例子展示了这种情况，其中的主机是一个拥有 6 个 IPv4 地址的多宿主机。

```
solaris % testga -f inet -t stream -h gateway.tuc.noao.edu -s daytime
socket(AF_INET,SOCK_STREAM, 0)
        address: 140.252.101.4.13
socket(AF_INET,SOCK_STREAM, 0)
        address: 140.252.102.1.13
socket(AF_INET,SOCK_STREAM, 0)
        address: 140.252.104.1.13
socket(AF_INET,SOCK_STREAM, 0)
        address: 140.252.3.6.13
socket(AF_INET,SOCK_STREAM, 0)
        address: 140.252.4.100.13
socket(AF_INET,SOCK_STREAM, 0)
        address: 140.252.1.4.13
```

下面指定主机为 alpha，它有一个 AAAA 记录和一个 A 记录，不指定协议族，指定服务名为只能在 TCP 上提供的 ftp 服务。

```
solaris  % testga  -h alpha  -s ftp
socket(AF_INET6,SOCK_STREAM, 0)
        address: 5f1b:df00:ce3e:e200:20:800:2b37:6426.21
socket(AF_INET,SOCK_STREAM, 0)
        address: 206.62.226.42.21
```

因为没有指定地址族，而且该例子是在一个能支持 IPv4 和 IPv6 的主机上运行，所以返回了两个结构：一个 IPv6 的和一个 IPv4 的。

下面设定 AI_PASSIVE 标志(用-p 选项)，不指定地址族，不指定主机名(相当于使用通配地址)，指定端口号为 8888，不指定套接口类型。

```
solaris % testga  -p  -s 8888
socket(AF_INET6,SOCK_STREAM, 0)
        address: ::.8888
socket(AF_INET6,SOCK_DGRAM, 0)
        address: ::.8888
socket(AF_INET,SOCK_STREAM, 0)
        address: 0.0.0.0.8888
socket(AF_INET,SOCK_DGRAM, 0)
        address: 0.0.0.0.8888
```

返回了 4 个结构。因为是在一个既支持 IPv6 又支持 IPv4 的主机上运行，而且没有指定地址族，getaddrinfo 返回 IPv6 通配地址和 IPv4 通配地址。由于指定了端口号而未指定套接口类型，getaddrinfo 为每个地址返回一个 TCP 类型的结构和一个 UDP 类型的结构。在第 10 章中可以看到在双重协议栈主机上的一个 IPv6 的客户或服务器既能与 IPv6 的对方通信，也能与 IPv4 的对方通信，所以两个 IPv6 的结构在 IPv4 结构前返回。

作为 Unix 域套接口的一个例子，下面指定主机名为 /local，服务名为 /tmp/test.1。

```
solaris % testga  -c  -p  -h /local  -s /tmp/test.1
socket(AF_LOCAL, SOCK_STREAM, 0), ai_canonname = solaris.kohala.com
        address: /tmp/test.1
socket(AF_LOCAL, SOCK_DGRAM, 0)
        address: /tmp/test.1
```

因为没有指定套接口类型，所以返回了两个结构：第一个是字节流套接口，第二个是数据报套接口。

11.7　host_serv 函数

我们给 getaddrinfo 设计的第一个接口不需要调用者来分配和填写 hints 结构。这个 host_serv 函数以地址族和套接口类型作为参数。

```
#include "unp.h"
struct addrinfo *host_serv(const char *hostname, const char *service, int
                           family, int socktype);
```

返回：成功返回指向addrinfo结构的指针，出错返回NULL

图11.5为该函数的源代码。

```
1 #include    "unp.h"
2 struct addrinfo *
3 host_serv(const char *host, const char *serv, int family, int socktype)
4 {
5     int     n;
6     struct addrinfo hints, *res;
7     bzero(&hints, sizeof(struct addrinfo));
8     hints.ai_flags = AI_CANONNAME;      /* always return canonical name */
9     hints.ai_family = family;           /* AF_UNSPEC, AF_INET, AF_INET6, etc. */
10    hints.ai_socktype = socktype;       /* 0, SOCK_STREAM, SOCK_DGRAM, etc. */
11    if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
12        return(NULL);
13    return(res);        /* return pointer to first on linked list */
14 }
```

图11.5 host_serv函数[lib/host_serv.c]

第7～13行 这个函数初始化一个hints结构，调用getaddrinfo，如果出错则返回一个空指针。

在图15.17中，当要用getaddrinfo获取主机和服务信息，但想自己建立连接时，调用了该函数。

11.8 tcp_connect函数

现在我们编写两个使用getaddrinfo的函数，以处理我们编写的TCP客户和服务器程序的大部分情况。第一个函数即tcp_connect执行客户程序的一般操作步骤：创建一个TCP套接口并与服务器建立连接。

```
#include "unp.h"
int tcp_connect(const char *hostname, const char *service);
```

返回：如成功则返回已连接套接口的描述字，出错则不返回

图11.6是它的源代码。

```
1 #include    "unp.h"
2 int
3 tcp_connect(const char *host, const char *serv)
4 {
5     int     sockfd, n;
```

```
 6    struct addrinfo   hints, *res, *ressave;
 7    bzero(&hints, sizeof(struct addrinfo));
 8    hints.ai_family = AF_UNSPEC;
 9    hints.ai_socktype = SOCK_STREAM;
10    if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
11        err_quit("tcp_connect error for %s, %s: %s",
12                 host, serv, gai_strerror(n));
13    ressave = res;
14    do {
15        sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
16        if (sockfd < 0)
17            continue;           /* ignore this one */
18        if (connect(sockfd, res->ai_addr, res->ai_addrlen) == 0)
19            break;              /* success */
20        Close(sockfd);          /* ignore this one */
21    } while ( (res = res->ai_next) != NULL);
22    if (res == NULL)            /* errno set from final connect() */
23        err_sys("tcp_connect error for %s, %s", host, serv);
24    freeaddrinfo(ressave);
25    return(sockfd);
26 }
```

图 11.6　tcp_connect 函数：执行客户程序的一般操作[lib/tcp_connect.c]

调用 getaddrinfo

第 7～13 行　调用了一次 getaddrinfo，这里指定地址族为 AF_UNSPEC，套接口类型为 SOCK_STREAM。

逐一尝试每个 addrinfo 结构，直到成功或到达链表尾

第 14～25 行　每个返回的 IP 地址都会被尝试：调用 socket 和 connect。如果在一台不支持 IPv6 的主机上用 IPv6 地址调用 socket，将返回错误，但这种错误不是致命的。如果 connect 成功，将会 break 出循环。否则，试过所有地址后，循环也会终止。freeaddrinfo 释放所有动态分配的内存。

如果 getaddrinfo 失败，或所有 connect 失败，这个函数（以及在以下各节中描述的给 getaddrinfo 提供一些简单接口的其他函数）将终止。只有在成功时才返回。如果不另加一个参数，是很难返回一个错误码的（EAI_xxx 常值之一）。这意味着我们的包裹函数是很简单的：

```
int
Tcp_connect(const char *host,const char *serv)
{
    return(tcp_connect(host,serv));
}
```

虽然如此，为了保持一致性，在下面的章节中我们使用自己的包裹函数代替 tcp_connect。

返回值的问题在于描述字是非负的值，可是我们并不知道EAI_xxx是正的还是负的。如果这些值是正数，在getaddrinfo失败时我们就可以返回这些值的负值，但我们还得返回某个另外的负值，表明所有结构都已尝试仍不成功的情况。

例子：时间/日期客户程序

图11.7是用tcp_connect重新编码的时间/日期客户程序。

命令行参数

第9～10行　现在需要第二个命令行参数指定服务名或端口号，以使程序连接其他的端口。

连接服务器

第11行　这个客户程序的所有套接口代码现由tcp_connect执行。

输出服务器地址

第12～15行　我们调用getpeername取得服务器的协议地址并输出。这是为了验证本例子中所使用的协议。

注意tcp_connect不返回connect要用的套接口地址结构的大小，可以加上一个指针参数来返回这个值，但是该函数的一个设计目标就是减少所用的参数的数目（与getaddrinfo相比）。因而在unp.h头文件中，定义了一个代表最大的套接口地址结构的大小的MAXSOCKADDR常值来弥补这一点。这个值通常是Unix域套接口地址结构的大小（14.2节），超过100个字节。我们给这么大的结构分配空间，它由getpeername来填写。

```
1 #include    "unp.h"
2 int
3 main(int argc, char **argv)
4 {
5     int         sockfd, n;
6     char        recvline[MAXLINE + 1];
7     socklen_t   len;
8     struct sockaddr  *sa;
9     if (argc != 3)
10        err_quit("usage: daytimetcpcli <hostname/IPaddress> <service/port#>");
11    sockfd = Tcp_connect(argv[1], argv[2]);
12    sa = Malloc(MAXSOCKADDR);
13    len = MAXSOCKADDR;
14    Getpeername(sockfd, sa, &len);
15    printf("connected to %s\n", Sock_ntop_host(sa, len));
16    while ( (n = Read(sockfd, recvline, MAXLINE)) > 0) {
17        recvline[n] = 0;   /* null terminate */
18        Fputs(recvline, stdout);
19    }
20    exit(0);
21 }
```

图11.7　用tcp_connect重新编码的时间/日期客户程序[names/daytimetcpcli.c]

我们用 malloc 来给该结构分配空间，而不是用：

```
char sockaddr[MAXSOCKADDR];
```

是因为字节对齐的问题。malloc 总是返回一个系统所要求的严格对齐的指针，而 char 数组可能被分配在一个奇数字节的边界上，这将对套接口地址结构中的 IP 地址或端口号造成问题。解决这个潜在问题的另一种方法是使用一个联合（图 4.19）。

这个版本的客户程序能与 IPv4 和 IPv6 一起工作，而图 1.5 中的版本只能用于 IPv4，图 1.6 中的版本只能用于 IPv6。你也应把这个新版本与图 E.14 中的做一下比较，在后者的编码里使用了 gethostbyname 和 getservbyname 来支持 IPv4 和 IPv6。

我们首先指定一个只支持 IPv4 的主机名。

```
solaris % daytimetcpcli bsdi daytime
connected to 206.62.226.35
Fri May 30 12:33:32 1997
```

下面指定一个 IPv4 和 IPv6 都支持的主机名。

```
solaris % daytimetcpcli aix daytime
connected to 5f1b:df00:ce3e:e200:20:800:5afc:2b36
Fri May 30 12:43:43 1997
```

使用 IPv6 地址的原因是该主机有一个 AAAA 记录和一个 A 记录，而且如图 11.4 所注，由于 tcp_connect 将地址族设为 AF_UNSPEC，因此首先搜索的是 AAAA 记录，只有在这失败后才搜索 A 记录。

下一个例子中，我们通过指定带－4 后缀的主机名来强制使用 IPv4 地址，我们在 9.2 节中已指出这是只有 A 记录的主机名的约定命名。

```
solaris % daytimetcpcli aix-4 daytime
connected to 206.62.226.43
Fri May 30 12:43:48 1997
```

11.9　tcp_listen 函数

下一个函数即 tcp_listen 执行 TCP 服务器程序的一般操作步骤：创建一个 TCP 套接口，给它捆绑服务器的众所周知端口，并允许接受外来的连接请求。图 11.8 是它的源代码。

```
#include "unp.h"

int tcp_listen(const char *hostname, const char *service, socklen_t *lenptr);
```

返回：成功返回已连接套接口描述字，出错则不返回

调用 getaddrinfo

第 8～15 行　初始化 addrinfo 结构时用了以下线索：AI_PASSIVE（因为这个函数是服务器程序用的）、AF_UNSPEC（地址族）、以及 SOCK_STREAM。回想图 11.4，如果没有指

定主机名(想绑定通配地址的服务器通常如此),使用 AI_PASSIVE 和 AF_UNSPEC 选项会返回两个套接口地址结构:第一个是 IPv6 的,下一个是 IPv4 的(假定是一个双重协议栈主机)。

创建套接口并给它捆绑地址

第 16～24 行 调用 socket 和 bind 函数。如果任何一个调用失败,则忽略这个 addrinfo 结构而使用下一个。如 7.5 节中所说明的,我们对于 TCP 服务器程序总是设置 SO_REUSEADDR 套接口选项。

失败检查

第 25～26 行 如果对所有的地址结构调用 socket 和 bind 都失败了,就输出错误并终止。就像前一节中的 tcp_connect 函数一样,我们不会试图从这个函数中返回错误。

第 27 行 调用 listen 使该套接口变成一个监听套接口。

返回套接口地址结构的大小

第 28～31 行 如果 addrlenp 参数不为空,就通过这个指针返回协议地址的大小。这可以让调用者给套接口地址结构分配内存,以用 accept 获得客户的协议地址(还可参见习题 11.1)。

```
1 #include    "unp.h"
2 int
3 tcp_listen(const char *host, const char *serv, socklen_t *addrlenp)
4 {
5     int       listenfd, n;
6     const int       on = 1;
7     struct addrinfo    hints, *res, *ressave;
8     bzero(&hints, sizeof(struct addrinfo));
9     hints.ai_flags = AI_PASSIVE;
10    hints.ai_family = AF_UNSPEC;
11    hints.ai_socktype = SOCK_STREAM;
12    if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
13      err_quit ("tcp_listen error for %s, %s: %s",
14              host, serv, gai_strerror(n));
15    ressave = res;
16    do {
17      listenfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
18      if (listenfd < 0)
19        continue;          /* error, try next one */
20      Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
21      if (bind(listenfd, res->ai_addr, res->ai_addrlen) == 0)
22        break;             /* success */
23      Close(listenfd);     /* bind error, close and try next one */
24    } while ( (res = res->ai_next) != NULL);
25    if (res == NULL)       /* errno from final socket() or bind() */
26        err_sys("tcp_listen error for %s, %s", host, serv);
27    Listen(listenfd, LISTENQ);
```

```
28      if (addrlenp)
29          *addrlenp = res->ai_addrlen;    /* return size of protocol address */
30      freeaddrinfo(ressave);
31      return(listenfd);
32  }
```

图 11.8　tcp_listen 函数：执行服务器程序的一般操作步骤[lib/tcp_listen.c]

例子：时间/日期服务器

图 11.9 是用 tcp_listen 重新编码的时间/日期服务器程序

需要服务名或端口号作为命令行参数

第 11～12 行　需要一个命令行参数指定服务名或端口号。这会使对服务器程序的测试更容易，因为捆绑标准时间/日期服务器的端口 13 需要超级用户权限。

```
 1 #include    "unp.h"
 2 #include    <time.h>
 3 int
 4 main(int argc, char **argv)
 5 {
 6     int            listenfd, connfd;
 7     socklen_t      addrlen, len;
 8     char           buff[MAXLINE];
 9     time_t         ticks;
10     struct sockaddr    *cliaddr;
11     if (argc != 2)
12         err_quit("usage: daytimetcpsrv1 <service or port#>");
13     listenfd = Tcp_listen(NULL, argv[1], &addrlen);
14     cliaddr = Malloc(addrlen);
15     for ( ; ; ) {
16         len = addrlen;
17         connfd = Accept(listenfd, cliaddr, &len);
18         printf("connection from %s\n", Sock_ntop(cliaddr, len));
19         ticks = time(NULL);
20         snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
21         Write(connfd, buff, strlen(buff));
22         Close(connfd);
23     }
24 }
```

图 11.9　用 tcp_listen 重新编码的时间/日期服务器程序(另见图 11.10)[names/daytimetcpsrv1.c]

创建监听套接口

第 13～14 行　tcp_listen 创建监听套接口，调用 malloc 分配一个缓冲区以存放客户的地址。

服务器循环

第 15～23 行　accept 等待客户的连接。调用 sock_ntop 输出客户的地址。无论是 IPv4

或IPv6,这个函数都输出IP地址和端口号。我们可以使用getnameinfo函数(在11.13节中介绍)来试图获得客户的主机名,这将会包含一个对DNS的PTR查询,PTR查询需要花费一定的时间,特别是在查询失败时。TCPv3的14.8节注明在一个繁忙的Web服务器上所有的与之连接的客户中大概有25%在DNS中没有PTR记录。因为我们不想让服务器(特别是一个迭代服务器)为一个PTR查询等待数秒钟,所以只输出IP地址和端口号。

例子:可指定协议的时间/日期服务器

图11.9中有一个小问题:如果tcp_listen的第一个参数是一个空指针,而且地址族为AF_UNSPEC的话,可能会使getaddrinfo返回我们不想要的地址族的套接口地址结构。举例来说,在一个双重协议栈主机上返回的第一个套接口地址结构会是IPv6的(图11.4),但我们可能想要服务器只处理IPv4。

客户没有这个问题,因为客户总是必需指定一个IP地址或主机名。客户程序一般让用户用命令行参数来输入它。这使我们有机会指定一个与特定类型的IP地址相关联的主机名(回想在9.2节中用的-4和-6主机名后缀),或者指定一个IPv4点分十进制数串(强制使用IPv4)或IPv6十六进制数串(强制使用IPv6)。

但是有一种简单的技术可以让我们强制服务器使用一种给定的协议,或IPv4或IPv6:让用户通过命令行参数输入一个IP地址或主机名,并将其传递给getaddrinfo。如果使用IP地址,IPv4点分十进制数串和IPv6十六进制数串是有差别的。下面对inet_pton的调用如所示的那样或成功或失败。

```
inet_pton(AF_INET, "0.0.0.0",  &foo);    /* succeeds */
inet_pton(AF_INET, "0::0",     &foo);    /* fails */
inet_pton(AF_INET6, "0.0.0.0", &foo);    /* fails */
inet_pton(AF_INET6, "0::0",    &foo);    /* succeeds */
```

因此,如果把我们的服务器程序改成能接受一个可选的参数,那么要是键入:

```
% server
```

在一个双重协议栈主机上就缺省使用IPv6,但是键入

```
% server 0.0.0.0
```

则显式指定使用IPv4,键入

```
% server 0::0
```

则指定使用IPv6。

图11.10是我们的时间/日期服务器程序的这个最终版本。

处理命令行参数

第11~16行 与图11.9相比唯一的改变是对命令行参数的处理,允许用户在服务名或端口号外,指定一个服务器绑定的主机名或IP地址,

首先用一个IPv4套接口启动服务器,然后从两个位于本地子网的主机上的客户向服务器发起连接。

```
solaris % daytimetcpsrv2 0.0.0.0 9999
```

```
connection from 206.62.226.36.32789
connection from 206.62.226.35.1389
```

再用 IPv6 套接口启动服务器。

```
solaris % daytimetcpsrv2 0::0 9999
connection from 5f1b:df00:ce3e:e200:20:800:2003:f642.32799
connection from 5f1b:df00:ce3e:e200:20:800:2b37:6426.1026
connection from ::ffff:206.62.226.36.32792
connection from ::ffff:206.62.226.35.1390
```

```
 1 #include    "unp.h"
 2 #include    <time.h>
 3 int
 4 main(int argc, char **argv)
 5 {
 6     int                 listenfd, connfd;
 7     socklen_t           addrlen, len;
 8     struct sockaddr     *cliaddr;
 9     char                buff[MAXLINE];
10     time_t              ticks;
11     if (argc == 2)
12       listenfd = Tcp_listen(NULL, argv[1], &addrlen);
13     else if (argc == 3)
14       listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
15     else
16       err_quit("usage: daytimetcpsrv2 [ <host> ] <service or port>");
17     cliaddr = Malloc(addrlen);
18     for ( ; ; ) {
19         len = addrlen;
20         connfd = Accept(listenfd, cliaddr, &len);
21         printf("connection from %s\n", Sock_ntop(cliaddr, len));
22       ticks = time(NULL);
23       snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
24       Write(connfd, buff, strlen(buff));
25         Close(connfd);
26     }
27 }
```

图 11.10　使用 tcp_listen 独立于协议的时间/日期服务器程序[names/daytimetcpsrv2.c]

```
connection from ::ffff:206.62.226.36.32792
connection from ::ffff:206.62.226.35.1390
```

第一个连接来自主机 sunos5，使用 IPv6，第二个来自主机 alpha，使用 IPv6。下两个连接来自主机 sunos5 和 bsdi，但使用 IPv4，而不是 IPv6。这是因为由 accept 返回的客户地址都是 IPv4 映射的 IPv6 地址。

刚才展示的是，在双重协议栈主机上运行的 IPv6 服务器能处理 IPv4 和 IPv6 客户。如同在 10.2 节中论述的那样，IPv4 客户的地址被作为一个 IPv4 映射的 IPv6 地址传递给

IPv6 服务器。

这个服务器程序以及图 11.7 中的客户程序都可以用于 Unix 域套接口(第 14 章),因为 11.16 节中 getaddrinfo 的实现支持 Unix 域套接口。举例来说,可像下面这样启动服务器。

```
solaris % daytimetcpsrv2 /local /tmp/rendezvous
```

/tmp/rendezvous 是任意选来给服务器捆绑的路径名。然后在同一台主机上启动客户,指定主机名为/local,服务名为/tmp/rendezvous。

```
solaris % daytimetcpcli /local /tmp/rendezvous
connected to /tmp/rendezvous
Fri May 30 16:31:37 1997
```

11.10 udp_client 函数

给 getaddrinfo 提供简单接口的函数在 UDP 这儿有所改变,因为这里我们提供了一个创建未连接 UDP 套接口的客户函数,在下一节中则提供另一个创建已连接 UDP 套接口的函数。

```
#include "unp.h"
int udp_client(const char *hostname, const char *service,
               void **saptr, socklen_t *lenp);
                                返回:成功返回未连接套接口的描述字,出错则不返回
```

这个函数创建一个未连接 UDP 套接口,返回三项数据。第一,返回值是套接口描述字。第二,saptr 是一个指向套接口地址结构(由 udp_client 动态分配)的指针(由调用者声明)的地址,在这个结构中存放目的 IP 地址和端口号,用来调用 sendto。套接口地址结构的大小在 lenp 指向的变量中返回。最后一个参数不能是空指针(tcp_listen 的最后一个参数是允许的),因为套接口地址结构的长度在调用 sendto 和 recvfrom 时都是需要的。

> saptr 应声明为 struct sockaddr **。使用 void **数据类型是因为在 31.3 节中定义了这个函数的另一个版本,它使用这个参数容纳一个指向另一种类型结构的指针的地址。这意味着我们调用该函数时必需加上(void **)强制类型转换。

图 11.11 给出了这个函数的源代码。

getaddrinfo 转换 hostname 和 service 参数。socket 创建数据报套接口。malloc 为一个套接口地址结构分配内存,并由 memcpy 将对应于创建的套接口的地址结构拷贝到这个内存区域中。

例子:独立于协议的时间/日期客户程序

为了使用 UDP 和 udp_client 函数,现在我们将图 11.7 中的时间/日期客户程序重新编

码。图 11.12 给出了这个独立于协议的程序的源代码。

第 11～16 行　调用 udp_client 函数，然后输出我们要向其发送 UDP 数据报的服务器的 IP 地址和端口号。发送一个 1 字节的数据报后读取并输出应答数据报。

```
1 #include    "unp.h"
2 int
3 udp_client(const char *host, const char *serv, void **saptr, socklen_t *lenp)
4 {
5     int          sockfd, n;
6     struct addrinfo    hints, *res, *ressave;
7     bzero(&hints, sizeof(struct addrinfo));
8     hints.ai_family = AF_UNSPEC;
9     hints.ai_socktype = SOCK_DGRAM;
10    if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
11      err_quit("udp_client error for %s, %s: %s",
12          host, serv, gai_strerror(n));
13    ressave = res;
14    do {
15      sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
16      if (sockfd >= 0)
17        break;              /* success */
18    } while ( (res = res->ai_next) != NULL);
19    if (res == NULL)          /* errno set from final socket() */
20      err_sys("udp_client error for %s, %s", host, serv);
21    *saptr = Malloc(res->ai_addrlen);
22    memcpy(*saptr, res->ai_addr, res->ai_addrlen);
23    *lenp = res->ai_addrlen;
24    freeaddrinfo(ressave);
25    return(sockfd);
26 }
```

图 11.11　udp_client 函数：创建一个未连接 UDP 套接口[lib/udp_client.c]

我们只需要发出一个 0 字节的 UDP 数据报，因为触发时间/日期服务器响应的是到来的数据报，而与它的长度和内容无关。但是许多 SVR4 的实现不允许 0 长度的 UDP 数据报。

下面在运行我们的客户程序时指定了一个拥有 AAAA 记录和 A 记录的主机名。因为 getaddrinfo 首先返回的结构是 AAAA 记录的，所以创建了一个 IPv6 套接口。

```
solaris % daytimeudpcli1 aix daytime
sending to 5f1b:df00:ce3e:e200:20:800:5afc:2b36
Sat May 31 08:13:34 1997
```

下面则指定同一主机的点分十进制数地址，结果创建了一个 IPv4 套接口。

```
solaris % daytimeudpcli1 206.62.226.43 daytime
sending to 206.62.226.43
Sat May 31 08:14:02 1997
```

```
1 #include    "unp.h"
2 int
3 main(int argc, char **argv)
4 {
5     int             sockfd, n;
6     char            recvline[MAXLINE + 1];
7     socklen_t       salen;
8     struct sockaddr* sa;
9     if (argc != 3)
10        err_quit("usage: daytimeudpcli1 <hostname/IPaddress> <service/port
          >");
11    sockfd = Udp_client(argv[1], argv[2], (void **) &sa, &salen);
12    printf("sending to %s\n", Sock_ntop_host(sa, salen));
13    Sendto(sockfd, "", 1, 0, sa, salen);        /* send 1-byte datagram */
14    n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
15    recvline[n] = 0;  /* null terminate */
16    Fputs(recvline, stdout);
17    exit(0);
18 }
```

图 11.12 使用 udp_client 的 UDP 时间/日期客户程序[names/daytimeudpcli1.c]

11.11 udp_connect 函数

udp_connect 函数创建一个已连接 UDP 套接口。

```
#include "unp.h"
int udp_connect(const char *hostname,const char *service);
                                返回:如果成功则返回套接口描述字,出错则不返回
```

得到已连接 UDP 套接口后,udp_client 所需的后两个参数就不再需要了。调用者可以用 write 代替 sendto,于是就不再需要返回套接口地址结构和它的长度了。

图 11.13 是源代码。

这个函数与 tcp_connect 几乎一样。但有一点不同是,调用在 UDP 套接口上的 connect 不会向对方发送任何数据报。如果有些错误(对方不可达或在指定端口上没有服务器),调用者在向对方发出数据报前不能发现。

```
1 #include    "unp.h"
2 int
3 udp_connect(const char *host, const char *serv)
4 {
5     int                 sockfd, n;
6     struct addrinfo    hints, *res, *ressave;
7     bzero(&hints, sizeof(struct addrinfo));
```

```
 8    hints.ai_family = AF_UNSPEC;
 9    hints.ai_socktype = SOCK_DGRAM;
10    if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
11        err_quit("udp_connect error for %s, %s: %s",
12                 host, serv, gai_strerror(n));
13    ressave = res;
14    do {
15        sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
16        if (sockfd < 0)
17            continue;    /* ignore this one */
18        if (connect(sockfd, res->ai_addr, res->ai_addrlen) == 0)
19            break;               /* success */
20        Close(sockfd);     /* ignore this one */
21    } while ( (res = res->ai_next) != NULL);
22    if (res == NULL)   /* errno set from final connect() */
23        err_sys("udp_connect error for %s, %s", host, serv);
24    freeaddrinfo(ressave);
25    return(sockfd);
26 }
```

图 11.13　udp_connect 函数：建立一个已连接 UDP 套接口[lib/udp_connect.c]

11.12　udp_server 函数

我们为简化 getaddrinfo 的接口开发的最后一个 UDP 函数是 udp_server。

```
#include "unp.h"
int udp_server(const char *hostname,const char *service,socklen_t *lenptr);
                                        返回：成功返回未连接套接口描述字，失败不返回
```

参数和 tcp_listen 一样：一个可选的 hostname，一个必需的 service(以给它捆绑一个端口)以及一个可选的指针，它指向返回套接口地址结构大小的变量。

图 11.14 是其源代码。

```
 1 #include    "unp.h"
 2 int
 3 udp_server(const char *host, const char *serv, socklen_t *addrlenp)
 4 {
 5     int                 sockfd, n;
 6     struct addrinfo    hints, *res, *ressave;
 7     bzero(&hints, sizeof(struct addrinfo));
 8     hints.ai_flags = AI_PASSIVE;
 9     hints.ai_family = AF_UNSPEC;
10     hints.ai_socktype = SOCK_DGRAM;
```

```
11    if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
12        err_quit("udp_server error for %s, %s: %s",
13                 host, serv, gai_strerror(n));
14    ressave = res;
15    do {
16        sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
17        if (sockfd < 0)
18          continue;           /* error, try next one */
19        if (bind(sockfd, res->ai_addr, res->ai_addrlen) == 0)
20            break;            /* success */
21        Close(sockfd);        /* bind error, close and try next one */
22    } while ( (res = res->ai_next) != NULL);
23    if (res == NULL)          /* errno from final socket() or bind() */
24        err_sys("udp_server error for %s, %s", host, serv);
25    if (addrlenp)
26        *addrlenp = res->ai_addrlen;   /* return size of protocol address */
27    freeaddrinfo(ressave);
28    return(sockfd);
29 }
```

图 11.14 udp_server 函数：给 UDP 服务器创建一个未连接套接口[lib/udp_server.c]

这个函数除了没有调用 listen 外，几乎与 tcp_listen 相同。我们将地址族设为 AF_UNSPEC，但调用者能以与图 11.10 中相同的技术来强制使用某个特定协议(IPv4 或 IPv6)。

对 UDP 套接口没有设置 SO_REUSEADDR 选项，因为就像 7.5 节中介绍的，在支持多播的主机上，这个套接口选项能允许多个套接口绑定同一个 UDP 端口。对 UDP 套接口来说，没有 TCP 那样的 TIME_WAIT 状态，所以启动服务器时不需要设置这个套接口选项。

例子：独立于协议的时间/日期服务器

图 11.15 给出了我们修改图 11.10 以使用 UDP 的时间/日期服务器程序。

```
1 #include    "unp.h"
2 #include    <time.h>
3 int
4 main(int argc, char **argv)
5 {
6     int             sockfd;
7     ssize_t         n;
8     char            buff[MAXLINE];
9     time_t          ticks;
10    socklen_t       addrlen, len;
11    struct sockaddr  *cliaddr;
12    if (argc == 2)
13        sockfd = Udp_server(NULL, argv[1], &addrlen);
14    else if (argc == 3)
15        sockfd = Udp_server(argv[1], argv[2], &addrlen);
16    else
```

```
17          err_quit("usage: daytimeudpsrv [ <host> ] <service or port>");
18      cliaddr = Malloc(addrlen);
19      for ( ; ; ) {
20          len = addrlen;
21          n = Recvfrom(sockfd, buff, MAXLINE, 0, cliaddr, &len);
22          printf("datagram from %s\n", Sock_ntop(cliaddr, len));
23        ticks = time(NULL);
24        snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
25        Sendto(sockfd, buff, strlen(buff), 0, cliaddr, len);
26      }
27  }
```

图 11.15　独立于协议的 UDP 时间/日期服务器程序[names/daytimeudpsrv2.c]

11.13　getnameinfo 函数

这个函数与 getaddrinfo 互补：它以一个套接口地址为参数，返回一个描述主机的字符串和一个描述服务的字符串。这个函数以一种独立于协议的方式提供这些信息；也就是说，调用者不必关心在套接口地址结构中的协议地址的类型，这些细节由函数自己处理。

```
#include <netdb.h>
int getnameinfo(const struct sockaddr *sockaddr,socklen_t addrlen,
                char *host,size_t hostlen,
                char *serv,size_t servlen,int flags);
                                        返回：成功返回 0，出错返回－1
```

sockaddr 指向包含协议地址的套接口地址结构，它将会被转换成可读的字符串，addrlen 是结构的长度。这个结构及其长度通常由 accept、recvfrom、getsockname 或 getpeername 返回。

调用者给两个直观可读的字符串分配空间：host 和 hostlen 指定主机字符串，serv 和 servlen 指定服务字符串。如果调用者不想返回主机字符串，将 hostlen 设为 0 即可。同样将 servlen 设为 0 就指定不返回服务的信息。为了给这两个字符串分配空间，图 11.6 给出了在 <netdb.h>中定义的常值。

常值	描述	值
NI_MAXHOST	返回的主机字符串的最大长度	1025
NI_MAXSERV	返回的服务字符串的最大长度	32

图 11.16　getnameinfo 返回的字符串的大小所用的常值

sock_ntop 和 getnameinfo 的差别在于，前者不查 DNS 直接返回可输出的 IP 地址和端口号，而后者通常试图给主机和服务查到一个名字。

图 11.17 中给出了五个可设置的标志，它们能改变 getnameinfo 函数的操作。

常值	描述
NI_DGRAM	数据报服务
NI_NAMEREQD	不能从地址反向解析到名字时返回错误
NI_NOFQDN	只返回FQDN的主机名部分
NI_NUMERICHOST	返回主机的数值格式串
NI_NUMERICSERV	返回服务的数值格式串

图11.17 getnameinfo所用的标志

当调用者知道它处理的是一个数据报套接口时应设置NI_DGRAM标志。原因是若在套接口地址结构中只给出IP地址和端口号的话，getnameinfo不能确定使用的协议(TCP或UDP)。在好几个端口号上使用的TCP服务和UDP服务截然不同。端口514就是一个例子，用TCP的服务是rsh，用UDP的服务是syslog。

如果用DNS不能反向解析到主机名，NI_NAMEREQD会导致返回一个错误。那些要求必须将客户方的IP地址映射到一个主机名的服务器可以使用它。这些服务器用返回的主机名调用gethostbyname，以验证返回的地址与套接口地址结构中的地址相同。

NI_NOFQDN使返回的主机名被截成第一个点号前部分。举例来说，如果在套接口地址结构中的IP地址为206.62.226.42，gethostbyaddr将返回主机名alpha.kohala.com。但要是设置了这个标志，返回的主机名是alpha。

NI_NUMERICHOST指定getnameinfo不调用DNS(这将花费一定的时间)，相反可能会调用inet_ntop以返回IP地址的数值表示。与此类似，NI_NUMERICSERV指定返回以十进制数表示的端口号，代替反向解析的服务名。由于客户的端口号一般没有对应的服务名——它们是临时的端口，因此服务器通常应设置NI_NUMERICSERV标志。

如果各标志的组合有意义的话(如NI_DGRAM和NI_NUMERICHOST)，可以通过逻辑或同时设置多个标志，但有些组合没有意义(如NI_NAMEREQD和NI_NUMERICHOST)。

> 在Posix.1g中没有getnameinfo，但它在RFC 2133[Gilligan et al. 1997]中有说明。

11.14 可重入函数

9.3节中的gethostbyname函数提出了一个我们目前还没有讨论过的问题：它是不可重入的。在23章中处理线程时会很普遍地遇到这个问题，但现在(不用涉及线程的概念)来讨论并解决这个问题也是十分有趣的。

首先让我们来看看这个函数是如何工作的。如果看一下它的源码(这很容易，因为整个BIND版本的源码都是公开的)，我们可以发现一个包含gethostbyname和gethostbyaddr的文件，这个文件的内容大致如下：

```
static struct hostent host;     /* result stored here */

struct hostent *
gethostbyname(const char *hostname)
{
    return(gethostbyname2(hostname,family));     /* Figure 9.6 */
}

struct hostent *
gethostbyname2(const char *hostname,int family)
{
    /* call DNS functions for A or AAAA query */
    /* fill in host structure */
    return(&host);
}

struct hostent *
gethostbyaddr(const char *addr,size_t len,int family)
{
    /* call DNS functions for PTR query in in-addr.arpa domain */
    /* fill in host structure */
    return(&host)
}
```

我们特别指出了结果结构的 static 存储类型，因为这正是问题之所在。这三个函数共用同一个 host 变量的事实还引起我们在习题 9.1 中讨论过的另一个问题。（回想图 9.6，gethostbyname2 是 BIND 4.9.4 为支持 IPv6 而新引入的。我们将忽略当调用 gethostbyname 时实际上也会调用 gethostbyname2 的事实，因为这并不会影响下面的讨论。）

在一个 UNIX 进程的主控制流程中调用 gethostbyname 或 gethostbyaddr，并在信号处理程序中也调用了这些函数，就可能发生重入问题。当调用信号处理程序时（假设每秒产生一个 SIGALRM 信号），进程的主控制流程会暂停而去调用信号处理程序。考虑一下下面的情况。

```
main()
{
    struct hostent *hptr;
    ...
    signal(SIGALRM,sig_alrm);
    ...
    hptr = gethostbyname( ... );
    ...
}

void
sig_alrm(int signo)
{
    struct hostent *hptr;
    ...
    hptr = gethostbyname( ... );
    ...
}
```

如果主控制流程暂停时正执行到 gethostbyname 的中间(在这个函数已经填好了 host 变量正要返回时),信号处理程序调用 gethostbyname,因为在进程中 host 变量只有一份,所以它被重用了。这就重写了 host 变量的值。

如果看看这一章和第 9 章里的名字和地址转换函数,以及第 4 章里的 inet_XXX 函数,我们注意到以下几点:

- 历史上,gethostbyname、gethostbyname2、gethostbyaddr、getservbyname 和 getservbyport 是不可重入的,因为它们都返回指向一个静态结构的指针。
  一些支持线程的实现(Solaris 2.x)提供了这四个函数的可重入版本,函数的名字以_r 结尾,我们会在下一节中进行介绍。
  也有些支持线程的实现(Digital Unix 4.0 和 HP-UX 10.30)提供这些函数的使用线程特定数据的可重入版本。
- inet_pton 和 inet_ntop 总是可重入的。
- 历史上 inet_ntoa 是不可重入的,但一些支持线程的实现提供了使用线程特定数据的可重入版本。
- getaddrinfo 只有在它自己调用的是可重入函数时才是可重入的;这就是说,它要调用可重入版本的 gethostbyname(解析主机名)和 getservbyname(解析服务名)。动态分配存放结果的内存空间的原因之一就是这样可以重入。
- getnameinfo 只有在它自己调用的是可重入函数时才是可重入的;这就是说,它要调用可重入版本的 gethostbyaddr 和 getservbyport 以获取主机名和服务名。注意所有的结果字符串(主机名和服务名)是由调用者分配的,以允许重入。

errno 变量有着同样的问题。历史上这个整型变量每个进程中只有一份拷贝。如果某进程作了一个系统调用且返回错误,一个整型的错误码将存放在这个变量里。举例来说,标准 C 库中的 close 函数被调用时,它执行的操作可能如下:

- 将系统调用的参数(一个整型的描述字)放入一个寄存器
- 将另一个寄存器置为某值表示正在调用 close 系统调用
- 进行系统调用(用特殊指令切换到核心态)
- 测试寄存器的值,以确定是否发生错误
- 如果没有错误,retrun(0)
- 将某个寄存器的值存入 errno
- return(-1)

首先要注意的是如果没有发生错误的话,errno 的值不会改变。因而除非知道发生了错误(通常函数用返回-1 来指出),否则不去看 errno 的值。

假设程序测试 close 函数的返回值,如果发生错误则输出 errno 的值,如下:

```
if (close (fd) < 0) {
    fprintf(stderr,"close error,errno = %d\n", errno)
    exit(1);
}
```

系统调用返回时将错误码存到 errno,该操作和在程序中输出这个值之间有一个小的时

间窗，这期间同一进程内的另一执行流程(即信号处理程序)可能改变了 errno 的值。举例来说，如果当调用信号处理程序时，主控制流程正处于 close 和 fprintf 之间，而且信号处理程序调用了一些其他系统调用(譬如 write)并返回了错误，那么 write 系统调用产生的 errno 值就会覆盖 close 调用的 errno 值。

考虑这两个与信号处理程序相关的问题，对 gethostbyname 产生的问题(返回一个指向静态变量的指针)的一种解决办法是不在信号处理程序中调用不可重入函数。errno 变量的问题(单个全局变量的值可能被信号处理程序改变)可通过在编码信号处理程序时保存和恢复 errno 的值来避免，如下所示：

```
void
sig_alrm(int signo)
{
    int     errno_save;

    errno_save = errno;     /* save its value on entry */
    if (write( ... ) != nbytes)
        fprintf(stderr,"write error,errno = %d\n",errno);
    errno = errno_save;     /* restore its value on return */
}
```

这段代码，在信号处理程序中也调用了 fprintf，它是一个标准的 I/O 函数。因为许多版本的标准 I/O 库是不可重入的，所以这也会导致重入问题：在信号处理程序中不应调用标准 I/O 函数。

我们在第 23 章中还会重提这个重入问题，并阐述线程怎样来处理 errno 变量的问题。下一节中介绍一些可重入版本的主机名转换函数。

11.15　gethostbyname_r 和 gethostbyaddr_r 函数

有两种方法可将像 gethostbyname 这样的不可重入函数变成可重入函数。

1. 变由函数填写并返回一个静态结构为：由调用者分配结构所需的空间，由可重入函数来填写。这是从不可重入的 gethostbyname 到可重入的 gethostbyname_r 所使用的技术。但这种方法也会使问题的解决变得更为复杂。因为 hostent 结构还指向其他的一些信息，包括规范名字、别名指针数组、别名串、地址指针数组和地址(见图 9.2)，所以调用者仅提供 hostent 结构的空间是不行的，它还必需提供一个大的缓冲区存放附加的信息，填写 hostent 结构时其中的一些指针就指向该缓冲区内。结果这个函数至少得加 3 个参数：一个指向要填写的 hostent 结构的指针、一个指向用来存放其他信息的缓冲区的指针以及缓冲区的大小。由于不能再用全局整型变量 h_errno，所以还需要第四个参数，即一个指向整型的指针以存放错误码。(全局变量 h_errno 存在与 errno 同样的重入问题。)

   getnameinfo 和 inet_ntop 也使用了这种技术。
2. 可重入函数调用 malloc 动态分配内存。getaddrinfo 用的是这种技术。这种方法的问题是调用这个函数的应用程序还要调用 freeaddrinfo 来释放所用的内存。如果不这样做就会导致内存漏损，也就是说进程每调用一次这样的函数，它的内存使用量就

增加。如果这个进程长时间运行(如网络服务器),使用的内存就随时间而不断增加。

现在讨论以下 Solaris 2.x 中名字到地址和地址到名字转换的可重入函数。

```
#include <netdb.h>

struct hostent *gethostbyname_r(const char *hostname,
                                struct hostent *result,
                                char *buf,int buflen,int *h_errnop);

struct hostent *gethostbyaddr_r(const char *addr,int len, int type,
                                struct hostent *result,
                                char *buf,int buflen,int *h_errnop);

                                返回:成功都返回非空指针,出错返回 NULL
```

每个函数都需要四个附加的参数。result 是由调用者分配的 hostent 结构,由函数填写。成功时这个指针也是函数的返回值。

buf 是由调用者分配的缓冲区,buflen 是它的大小。缓冲区将存放规范主机名、别名指针、别名串、地址指针和实际地址。result 指向的结构中的所有指针都指到这个缓冲区中。这个缓冲区到底应该有多大呢?不幸的是基本上所有的手册页面都说得很含糊,类似于"缓冲区要大到能放下所有与该主机相关的数据"。gethostbyname 当前的实现内部使用 8192 字节的缓冲区存放别名和地址,最多能返回 35 个别名指针,35 个地址指针。因此 8192 字节的缓冲区应该是足够的。

如果出错,错误码通过 h_errnop 指针返回,而不是通过全局变量 h_errno。

遗憾的是这个重入问题比它表面看来要严重。首先,没有一个关于 gethostbyname 和 gethostbyaddr 的重入问题的标准。Posix.1g 对这两个函数作了说明,但没有提到线程安全性问题。Unix 98 只是说这两个函数不是线程安全的。其次,没有关于_r 函数的标准。这一节中展示的两个_r 函数(作为例子)是由 Solaris 2.x 提供的。而 Digital Unix 4.0 和 HP-UX 10.30 中的这些函数所带的参数与 Solaris 不同,gethostbyname_r 函数的前两个参数与 Solaris 的是一样的,但把 Solaris 版本中的后三个参数合成一个新的 hostent_data 结构(由调用者分配),并以指向这个结构的指针作为第三个,也是最后一个参数。Digital Unix 4.0 和 HP-UX 10.30 中的普通 gethostbyname 和 gethostbyaddr 函数使用了线程特定数据(见 23.5 节),是可重入的。Solaris 2.x 中_r 函数的开发历史可见于[Maslen 1997]。

最后,虽然 gethostbyname 的可重入版本能在不同的线程同时调用它时提供安全性,但一点也没有提及在它底层的名字解析函数的重入性。在本书写作时,BIND 中的名字解析函数还是不可重入的。

11.16　getaddrinfo 和 getnameinfo 函数的实现

现在来看看 getaddrinfo 和 getnameinfo 的一个实现。在前者的实现中我们可以进一步了解它的操作细节。这个实现也支持在 11.5 节中提到的 Unix 域套接口。

> 注意：在本节中，所有依赖于 IPv4、IPv6 或 Unix 域支持的相应部分的代码，都用 #ifdef 和 #endif 包了起来，对应的条件编译常值为：IPV4、IPV6 和 UNIX-DOMAIN。这样可使这个程序能在支持这三种协议的任意组合的系统上进行编译。但是这里删去了这些预处理器语句，因为这些语句对我们的讨论没有什么帮助，而且使源程序难于理解。
>
> 我们也注意到在第 14 章前还没有了解 Unix 域套接口的细节。

图 11.18 展现了 getaddrinfo 调用的函数，它们都由 ga_开头。

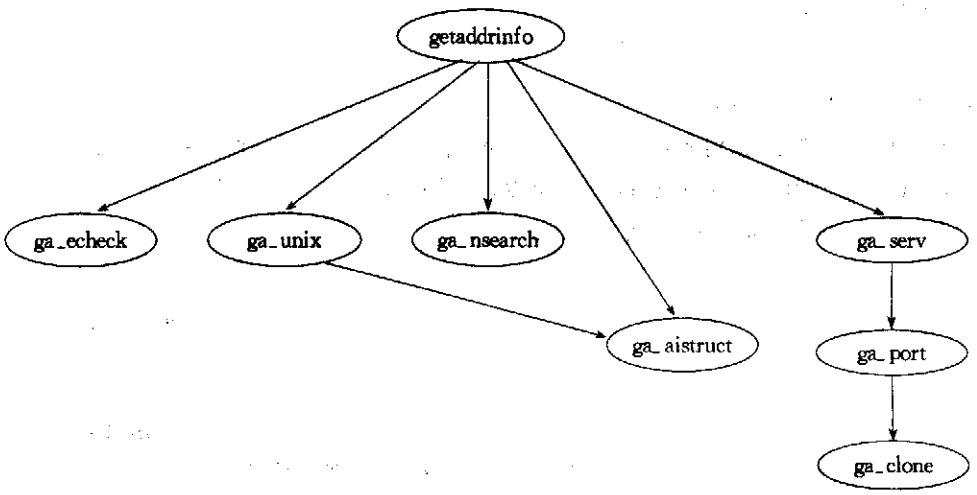


图 11.18　getaddrinfo 实现中调用的函数

第一个文件是在图 11.19 中的 gai_hdr.h 头文件，我们所有的源文件都包含它。

这个头文件包含 udp.h 和另外一个头文件。我们马上会看到 AI_CLONE 标志和 search 结构的使用。它的其余部分定义了本节中各个函数的原型。

```
1 #include    "unp.h"
2 #include    <ctype.h>     /* isxdigit(), etc. */
3           /* following internal flag cannot overlap with other AI_xxx flags */
4 #define AI_CLONE      4     /* clone this entry for other socket types */

5 struct search {
6     const char *host;         /* hostname or address string */
7     int       family;         /* AF_xxx */
8 };

9           /* function prototypes for our own internal functions */
10 int     ga_aistruct(struct addrinfo ***, const struct addrinfo *,
11                     const void *, int);
12 struct addrinfo *ga_clone(struct addrinfo *);
```

```
13 int    ga_echeck(const char *, const char *, int, int, int, int);
14 int    ga_nsearch(const char *, const struct addrinfo *, struct search *);
15 int    ga_port(struct addrinfo *, int, int);
16 int    ga_serv(struct addrinfo *, const struct addrinfo *, const char *);
17 int    ga_unix(const char *, struct addrinfo *, struct addrinfo **);
18 int    gn_ipv46(char *, size_t, char *, size_t, void *, size_t,
19                 int, int, int);
```

图 11.19 gai_hdr.h 头文件[libgai/gai_hdr.c]

图 11.20 给出了 getaddrinfo 函数的第一部分。

定义 error 宏

第 13～17 行 在这个函数中有多处，如果遇到出错，就释放所有已分配的内存并返回相应的返回码。为了简化代码，我们定义了这个宏，把返回码存到变量 error 中，并转到函数结尾处的 bad 标记处(图 11.26)。

初始化自动变量

第 18～20 行 初始化一些自动变量。在图 11.34 中我们描绘了 aihead 和 aipnext 指针。

拷贝调用者的 hints 结构

第 21～25 行 如果调用者提供了一个 hints 结构，我们就把它拷贝到局部变量，这样在后面可以修改它。否则从一个全零的结构开始，不过 ai_family 被初始化为 AF_UNSPEC。后者一般定义为 0，但这不是 Posix.1g 要求的。

检查参数

第 26～29 行 调用图 11.39 中给出的 ga_echeck 函数，对一些参数进行检查。

检查 Unix 域路径名

第 30～34 行 如果主机名为/local 或/unix，而且服务名以斜杠开头，就把这个参数作为 Unix 域路径名来处理。函数 ga_unix(图 11.33)处理这种路径名。

```
 1 #include    "gai_hdr.h"
 2 #include    <arpa/nameser.h>   /* needed for <resolv.h> */
 3 #include    <resolv.h>         /* res_init,  _res */
 4 int
 5 getaddrinfo(const char *hostname, const char *servname,
 6             const struct addrinfo *hintsp, struct addrinfo **result)
 7 {
 8     int       rc, error, nsearch;
 9     char    **ap, *canon;
10     struct hostent *hptr;
11     struct search search[3], *sptr;
12     struct addrinfo hints, *aihead, **aipnext;
13     /*
14      * If we encounter an error we want to free() any dynamic memory
15      * that we've allocated.   This is our hack to simplify the code.
16      */
17 #define     error(e) { error = (e); goto bad; }
18     aihead = NULL;          /* initialize automatic variables */
19     aipnext = &aihead;
```

```
20      canon = NULL;
21      if (hintsp == NULL) {
22          bzero(&hints, sizeof(hints));
23          hints.ai_family = AF_UNSPEC;
24      } else
25          hints = *hintsp;          /* struct copy */
26          /* first some basic error checking */
27      if ( (rc = ga_echeck (hostname, servname, hints.ai_flags, hints.ai_family,
28                            hints.ai_socktype, hints.ai_protocol)) != 0)
29          error(rc);
30          /* special case Unix domain first */
31      if (hostname != NULL &&
32          (strcmp(hostname, "/local") == 0 || strcmp(hostname, "/unix") == 0) &&
33          (servname != NULL && servname[0] == '/'))
34          return(ga_unix(servname, &hints, result));
```

图 11.20　getaddrinfo 函数:第一部分,初始化[libgai/getaddrinfo.c]

getaddrinfo 函数剩下的部分(在图 11.24 中继续)处理 IPv4 和 IPv6 套接口。函数 ga_nsearch 的第一部分在图 11.21 中给出,计算查找一个主机名的次数。如果调用者指定的地址族是 AF_INET 或 AF_INET6,那么只查找一次主机名。但如果地址族是未指定类型即 AF_UNSPEC,就要做两次查找:一次 IPv6,一次 IPv4。我们把这个函数分成三部分:

- 没有主机名,但指定了 AI_PASSIVE
- 没有主机名,也没有指定 AI_PASSIVE(也就是主动)
- 指定了主机名

这三部分对应于图 11.4 中的三个主要部分。

```
6 int
7 ga_nsearch(const char *hostname, const struct addrinfo *hintsp,
8             struct search *search)
9 {
10      int             nsearch = 0;
11      if (hostname == NULL || hostname[0] == '\0') {
12          if (hintsp->ai_flags & AI_PASSIVE) {
13                  /* no hostname and AI_PASSIVE: implies wildcard bind */
14              switch (hintsp->ai_family) {
15              case AF_INET:
16                  search[nsearch].host = "0.0.0.0";
17                  search[nsearch].family = AF_INET;
18                  nsearch++;
19                  break;
20              case AF_INET6:
21                  search[nsearch].host = "0::0";
22                  search[nsearch].family = AF_INET6;
23                  nsearch++;
24                  break;
25              case AF_UNSPEC:
26                  search[nsearch].host = "0::0";          /* IPv6 first, then IPv4 */
27                  search[nsearch].family = AF_INET6;
28                  nsearch++;
```

```
29              search[nsearch].host = "0.0.0.0";
30              search[nsearch].family = AF_INET;
31              nsearch++;
32              break;
33          }
```

图 11.21 ga_nsearch 函数：没有主机名，但指定了 AI_PASSIVE[libgai/ga_nsearch.c]

没有主机名，被动的套接口

第 11～33 行 如果调用者没有指定一个主机名，不过指定了 AI_PASSIVE，那么返回用于创建一个或多个捆绑到通配地址的被动套接口的信息。switch 是根据地址族转移的：IPv4 套接口需要捆绑 0.0.0.0(INADDR_ANY)，而 IPv6 套接口需要捆绑 0::0(IN6ADDR_ANY_INIT)。如果地址族为 AF_UNSPEC，则必须返回用于创建两个套接口的信息：第一个是 IPv6 的，第二个是 IPv6 的。把 IPv6 排在第一，然后才轮到 IPv4，是因为在一个双重协议栈的主机上，IPv6 套接口可以处理 IPv6 和 IPv4 两种客户。在这种情况下，如果调用者只从返回的 addrinfo 结构表中创建一个套接口，它应该是 IPv6 套接口。

这个函数创建一个 search 结构的数组(图 11.19)，其中的每项指明查找的主机名和地址族。指向调用者的 search 结构数组的指针是这个函数的最后一个参数。返回值是创建的这些结构的数目，它总是 1 或 2。

这个函数的下一部分在图 11.22 中给出，这部分处理没有主机名和未设置 AI_PASSIVE 的情况，这意味着调用者想要在本机上创建一个主动的套接口。

```
34      } else {
35              /* no host and not AI_PASSIVE: connect to local host */
36          switch (hintsp->ai_family) {
37          case AF_INET:
38              search[nsearch].host = "localhost"; /* 127.0.0.1 */
39              search[nsearch].family = AF_INET;
40              nsearch++;
41              break;
42          case AF_INET6:
43              search[nsearch].host = "0::1";
44              search[nsearch].family = AF_INET6;
45              nsearch++;
46              break;
47          case AF_UNSPEC:
48              search[nsearch].host = "0::1";      /* IPv6 first, then IPv4 */
49              search[nsearch].family = AF_INET6;
50              nsearch++;
51              search[nsearch].host = "localhost";
52              search[nsearch].family = AF_INET;
53              nsearch++;
54              break;
55          }
56      }
```

图 11.22 ga_nsearch 函数：没有主机名而且没有设置 AI_PASSIVE[libgai/ga_nsearch.c]

第 34～56 行 对 IPv4 我们假定主机名 localhost 将返回回馈地址，一般是 127.0.0.1。在 IPv6 中对于本地机没有一个通用的主机名，因此返回回馈地址 0::1。跟被动情况一样，

如果没有指定地址族就返回两个结构：第一个是 IPv6 的，另一个是 IPv4 的。

图 11.23 给出了这个函数的最后一部分，它是最初的 if 语句的 else 子句。在指定了主机名时执行这段代码。

第 57～82 行　在这种情况下 AI_PASSIVE 标志无关紧要；需要查找主机名。如果调用者创建一个被动套接口，返回的套接口地址结构将用在 bind 调用中，但如果调用者创建一个主动的套接口，套接口地址结构将用于 connect 调用。我们创建一个或两个 search 结构：指定了地址族时是一个，没有指定地址族时是两个。跟前两种情况一样，如果返回两个地址结构，第一个是 IPv6 的，第二个是 IPv4 的。

现在回到 getaddrinfo 函数，在图 11.24 中，以一个 ga_nsearch 调用开始。

```
57    } else {      /* host is specified */
58        switch (hintsp->ai_family) {
59        case AF_INET:
60            search[nsearch].host = hostname;
61            search[nsearch].family = AF_INET;
62            nsearch++;
63            break;
64        case AF_INET6:
65            search[nsearch].host = hostname;
66            search[nsearch].family = AF_INET6;
67            nsearch++;
68            break;
69        case AF_UNSPEC:
70            search[nsearch].host = hostname;
71            search[nsearch].family = AF_INET6;   /* IPv6 first */
72            nsearch++;
73            search[nsearch].host = hostname;
74            search[nsearch].family = AF_INET;    /* then IPv4 */
75            nsearch++;
76            break;
77        }
78    }
79    if (nsearch < 1 || nsearch > 2)
80        err_quit("nsearch = %d", nsearch);
81    return(nsearch);
82 }
```

图 11.23　ga_nsearch 函数：指定了主机名[libgai/ga_nsearch.c]

调用 ga_nsearch

第 36 行　调用 ga_nsearch 函数，填写 search 数组，返回数组中的结构数目：一个或两个。

遍历全部 search 结构

第 37 行　遍历每个由 ga_nsearch 创建的 search 结构。

检查 IPv4 点分十进制数串

第 39～44 行　如果主机名的第一个字符是一个数字，则检查主机名是不是一个点分十进制数串。调用 inet_pton 做这项检查和转换。如果调用成功了但调用者指定的地址族不是

AF_INET，则出错。

第45～46行 我们要检查search结构的family是不是也是AF_INET，但在这里出现不匹配只是使这个search结构被忽略。这种情况有可能发生，譬如，当调用者指定一个主机名192.3.4.5但没有指定地址族时。ga_nsearch为它创建两个search结构：一个是IPv6的，一个是IPv4的。第一次for循环时，inet_pton调用成功，但因为search结构的family是AF_INET6，我们想忽略这个结构，而不发生错误。

```
35          /* remainder of function for IPv4/IPv6 */
36      nsearch = ga_nsearch(hostname, &hints, &search[0]);
37      for (sptr = &search[0]; sptr < &search[nsearch]; sptr++) {
38              /* check for an IPv4 dotted-decimal string */
39          if (isdigit(sptr->host[0])) {
40              struct in_addr      inaddr;
41                  if (inet_pton(AF_INET, sptr->host, &inaddr) == 1) {
42                      if (hints.ai_family != AF_UNSPEC &&
43                          hints.ai_family != AF_INET)
44                          error(EAI_ADDRFAMILY);
45                      if (sptr->family != AF_INET)
46                          continue;           /* ignore */
47                      rc = ga_aistruct(&aipnext, &hints, &inaddr, AF_INET);
48                      if (rc != 0)
49                          error(rc);
50                      continue;
51              }
52          }
53              /* check for an IPv6 hex string */
54          if ((isxdigit(sptr->host[0]) || sptr->host[0] == ':') &&
55              (strchr(sptr->host, ':') != NULL)) {
56              struct in6_addr     in6addr;
57              if (inet_pton(AF_INET6, sptr->host, &in6addr) == 1) {
58                  if (hints.ai_family != AF_UNSPEC &&
59                      hints.ai_family != AF_INET6)
60                      error(EAI_ADDRFAMILY);
61                  if (sptr->family != AF_INET6)
62                      continue;       /* ignore */
63                  rc = ga_aistruct(&aipnext, &hints, &in6addr, AF_INET6);
64                  if (rc != 0)
65                      error(rc);
66                  continue;
67              }
68          }
```

图11.24 getaddrinfo函数：检查IPv4或IPv6地址串[libgai/getaddrinfo.c]

创建addrinfo结构

第47～52行 ga_aistruct函数创建一个addrinfo结构并把它加到正在建立的链表中(aipnext指针)。

检查IPv6地址串

第53～60行 如果主机名的第一个字符是十六进制的数字或冒号，并且字符串中有一

个冒号，则调用 inet_pton 检查主机名是不是一个 IPv6 地址串。如果调用成功但调用者指定了一个不是 AF_INET6 的地址族，则出错。

第 61～62 行　检查 search 结构的地址族是不是也是 AF_INET6，但在这里不匹配只是导致这个 search 结构被忽略。

创建 addrinfo 结构

第 63～68 行　函数 ga_aistruct 创建一个 addrinfo 结构并将其加到已有的链表中。

在循环(图 11.24)中的前两个条件测试处理 IPv4 点分十进制数串或 IPv6 地址串。循环剩余的部分，见图 11.25，调用 gethostbyname 和 gethostbyname2 查找主机名。

初始化名字解析器

第 70～71 行　如果还没有调用过名字解析器的 res_init 函数，就调用它进行初始化。

如果要进行两次查找就调用 gethostbyname2

第 72～74 行　如果 nsearch 为 2，就进行两次 for 循环：一次 IPv6 和一次 IPv4。如果主机名参数只有一个某种协议的地址，我们希望只返回这个地址。举例来说，9.2 节中的主机 solaris 在 DNS 中有一个 AAAA 记录和一个 A 记录。第一次循环希望找到 AAAA 记录，第二次循环希望找到 A 记录。但如果主机只有一个 A 记录，就不需要在第一次循环时处理这个记录，因为第一次循环时 search 结构中的 family 成员为 AF_INET6。这就是说，既然知道将为该主机查找一个 A 记录，我们就不再用 gethostbyname 查找 AAAA 记录，因为这可能会返回一个与该 A 记录对应的 IPv4 映射的 IPv6 地址。看一下图 9.5，当地址族为 AF_INET 时只查找 A 记录、地址族为 AF_INET6 时只查找 AAAA 记录的方法，就是用 gethostbyname2 代替 gethostbyname，同时需将 RES_USE_INET6 选项关闭。

如果只进行一次查找就调用 gethostbyname

第 75～81 行　如果只进行一次查找，就调用 gethostbyname，当地址族为 AF_INET6 时设置 RES_USE_INET6 选项，当地址族为 AF_INET 时清除 RES_USE_INET6 选项。举例来说，如果调用者查找的主机名只有一个 A 记录，但指定的地址族为 AF_INET6，将返回对应的 IPv4 映射的 IPv6 地址。

处理名字解析失败的情况

第 82～97 行　如果名字解析失败，但 nsearch 值为 2，这不算错误，因为某一次循环可能会成功。(在循环结束时要检查是否至少返回一个 addrinfo 结构。)但要是这是唯一的一次调用解析器函数，则需根据解析器的 h_errno 返回相应的错误。

检查地址族是否匹配

第 98～100 行　如果调用者指定了一个地址族，但名字解析返回的地址族与此不符，则返回错误。

保存主机名

第 101～106 行　如果调用者指定了主机名并设置了 AI_CANNONAME 标志，则将由解析器返回的第一个名字保存下来。(回想图 11.22，即使没有指定主机名，也用 localhost 这个名字调用解析器。)复制由解析器返回的字符串，将它的指针存到 canon 中。

```
69              /* remainder of for() to look up hostname */
70          if ((_res.options & RES_INIT) == 0)
71              res_init();             /* need this to set _res.options */
72          if (nsearch == 2) {
73              _res.options &= ~RES_USE_INET6;
74              hptr = gethostbyname2(sptr->host, sptr->family);
75          } else {
76              if (sptr->family == AF_INET6)
77                  _res.options |= RES_USE_INET6;
78              else
79                  _res.options &= ~RES_USE_INET6;
80              hptr = gethostbyname(sptr->host);
81          }
82          if (hptr == NULL) {
83              if (nsearch == 2)
84                  continue;   /* failure OK if multiple searches */
85              switch (h_errno) {
86              case HOST_NOT_FOUND:
87                  error(EAI_NONAME);
88              case TRY_AGAIN:
89                  error(EAI_AGAIN);
90              case NO_RECOVERY:
91                  error(EAI_FAIL);
92              case NO_DATA:
93                  error(EAI_NODATA);
94              default:
95                  error(EAI_NONAME);
96              }
97          }
98              /* check for address family mismatch if one specified */
99           if (hints.ai_family != AF_UNSPEC && hints.ai_family != hptr->h_
               addrtype)
100              error(EAI_ADDRFAMILY);
101             /* save canonical name first time */
102         if (hostname != NULL && hostname[0] != '\0' &&
103             (hints.ai_flags & AI_CANONNAME) && canon == NULL) {
104             if ( (canon = strdup(hptr->h_name)) == NULL)
105                 error(EAI_MEMORY);
106         }
107             /* create one addrinfo() for each returned address */
108         for (ap = hptr->h_addr_list; *ap != NULL; ap++) {
109             rc = ga_aistruct(&aipnext, &hints, *ap, hptr->h_addrtype);
110             if (rc != 0)
111                 error(rc);
112         }
113     }
114     if (aihead == NULL)
115         error(EAI_NONAME);              /* nothing found */
```

图 11.25 getaddrinfo 函数：查找主机名[libgai/getaddrinfo.c]

为每一个地址创建一个 addrinfo 结构

第 107～112 行　为每一个在 h_addr_list 数组中、由解析器返回的地址，调用我们的 ga_aistruct 函数创建一个 addrinfo 结构，并加到已建立的结构链表中。

检查是否没有任何匹配

第 114～115 行　如果 addrinfo 结构链表的头仍为空指针，那么 for 循环的每次迭代都失败了。

图 11.26 是 getaddrinfo 函数的最后一部分。

```
116         /* return canonical name */
117     if (hostname != NULL && hostname[0] != '\0' &&
118         hints.ai_flags & AI_CANONNAME) {
119         if (canon != NULL)
120             aihead->ai_canonname = canon;      /* strdup'ed earlier */
121         else {
122             if ( (aihead->ai_canonname = strdup(search[0].host)) == NULL)
123                 error(EAI_MEMORY);
124         }
125     }
126         /* now process the service name */
127     if (servname != NULL && servname[0] != '\0') {
128         if ( (rc = ga_serv(aihead, &hints, servname)) != 0)
129             error(rc);
130     }
131     *result = aihead;      /* pointer to first structure in linked list */
132     return(0);
133 bad:
134     freeaddrinfo(aihead);   /* free any alloc'ed memory */
135     return(error);
136 }
```

图 11.26　getaddrinfo 函数：处理服务名[libgai/getaddrinfo.c]

返回规范主机名

第 116～125 行　如果调用者指定了主机名并设置了 AI_CANONNAME 标志，而且在 canon 指针中保存了名字的一个拷贝，那么这个指针会在第一个 addrinfo 结构的 ai_canonname 成员中返回。如果解析器没有找到名字(可能主机名是一个地址串)，则返回主机名参数的一个拷贝作为替代品。

处理服务名

第 126～130 行　如果调用者指定了服务名，就调用我们的 ga_serv 函数对其进行处理。

返回指向链表的指针

第 131～132 行　返回的指针指向已建立的 addrinfo 结构链表的头，函数本身返回 0。

出错返回

第 133～135 行　如果遇到错误，则调用 freeaddrinfo 释放所有已分配的内存，并返回一

个 EAI_xxx 的错误值。

图 11.26 中用来处理服务名参数的 ga_serv 函数，在图 11.27 中给出了它的源代码。

检查端口号串

第 12～27 行 如果服务名的第一个字符为数字，我们就认为服务名是一个端口号并调用 atoi 将其转成二进制值。如果调用者指定了套接口类型(SOCK_STREAM 或 SOCK_DGRAM)，就为该套接口类型调用一次 ga_port 函数。但如果没有指定套接口类型，则调用两次 ga_port 函数，一次给 TCP，一次给 UDP(回想图 11.2。)。这里有一个计数器记录 ga_port 返回成功的次数，只有在函数结束时其值仍为 0 时才返回错误。

尝试按 TCP 调用 getservbyname

第 28～36 行 如果没有指定套接口类型，或指定了一个 TCP 类型的套接口，调用 getservbyname 时第二个参数就置为“tcp”。成功的话，就调用 ga_port 函数。失败也没关系，因为这个服务名可能对 UDP 有效。

尝试以 UDP 为参数调用 getservbyname

第 37～44 行 如果没有指定套接口类型，或指定了一个 UDP 类型的套接口，调用 getservbyname 时第二个参数就置为“udp”。成功的话，就调用 ga_port 函数。

错误检查

第 45～51 行 如果 nfound 计数器不为 0，则返回成功。否则返回错误。

在图 11.27 中每找到一个端口号时都要调用 ga_port 函数(详细代码见图 11.28)。

遍历所有的 addrinfo 结构

第 33 行 遍历所有由图 11.24 和 11.25 中的 ga_aistruct 调用创建的 addrinfo 结构。当调用者不指定套接口类型时，ga_aistruct 总是设置 AI_CLONE 标志。这表示这个 addrinfo 结构可能需要为 TCP 和 UDP 各复制一份。

检查 AI_CLONE 标志

第 34～42 行 如果设置了 AI_CLONE 标志而且套接口类型不为 0，将用 ga_clone 函数复制出另一个与之完全相同的 addrinfo 结构。我们很快就会给出这样的例子。

设置套接口地址结构中的端口号

第 44～47 行 设置套接口地址结构中的端口号，并增加计数器 nfound 的值。

```
 5 int
 6 ga_serv(struct addrinfo *aihead, const struct addrinfo *hintsp,
 7         const char *serv)
 8 {
 9     int             port, rc, nfound;
10     struct servent      *sptr;
11     nfound = 0;
12     if (isdigit(serv[0])) {     /* check for port number string first */
13         port = htons(atoi(serv));
14         if (hintsp->ai_socktype) {
15                 /* caller specifies socket type */
16             if ( (rc = ga_port(aihead, port, hintsp->ai_socktype)) < 0)
```

```
17                  return(EAI_MEMORY);
18              nfound += rc;
19          } else {
20                  /* caller does not specify socket type */
21              if ( (rc = ga_port(aihead, port, SOCK_STREAM)) < 0)
22                  return(EAI_MEMORY);
23              nfound += rc;
24              if ( (rc = ga_port(aihead, port, SOCK_DGRAM)) < 0)
25                    return(EAI_MEMORY);
26              nfound += rc;
27          }
28      } else {
29              /* try service name, TCP then UDP */
30          if (hintsp->ai_socktype == 0 || hintsp->ai_socktype == SOCK_STREAM)
            {
31              if ( (sptr = getservbyname(serv, "tcp")) != NULL) {
32                  if ( (rc = ga_port(aihead, sptr->s_port, SOCK_STREAM)) < 0)
33                      return(EAI_MEMORY);
34                  nfound += rc;
35              }
36          }
37          if (hintsp->ai_socktype == 0 || hintsp->ai_socktype == SOCK_DGRAM)
            {
38              if ( (sptr = getservbyname(serv, "udp")) != NULL) {
39                  if ( (rc = ga_port(aihead, sptr->s_port, SOCK_DGRAM)) < 0)
40                      return(EAI_MEMORY);
41                  nfound += rc;
42              }
43          }
44      }
45      if (nfound == 0) {
46          if (hintsp->ai_socktype == 0)
47              return(EAI_NONAME);      /* all calls to getservbyname() failed */
48          else
49              return(EAI_SERVICE);   /* service not supported for socket type */
50      }
51      return(0);
52  }
```

图 11.27　ga_serv 函数[libgai/ga_serv.c]

```
27 int
28 ga_port(struct addrinfo *aihead, int port, int socktype)
29              /* port must be in network byte order */
30 {
31      int                 nfound = 0;
32      struct addrinfo     *ai;
33      for (ai = aihead; ai != NULL; ai = ai->ai_next) {
34          if (ai->ai_flags & AI_CLONE) {
35              if (ai->ai_socktype != 0) {
36                  if ( (ai = ga_clone(ai)) == NULL)
37                      return(-1);          /* memory allocation error */
```

```
38                  /* ai points to newly cloned entry, which is what we want */
39              }
40          } else if (ai->ai_socktype != socktype)
41              continue;           /* ignore if mismatch on socket type */
42          ai->ai_socktype = socktype;
43          switch (ai->ai_family) {
44          case AF_INET:
45              ((struct sockaddr_in *) ai->ai_addr)->sin_port = port;
46              nfound++;
47              break;
48          case AF_INET6:
49              ((struct sockaddr_in6 *) ai->ai_addr)->sin6_port = port;
50              nfound++;
51              break;
52          }
53      }
54      return(nfound);
55  }
```

图 11.28 ga_port 函数[libgai/ga_port.c]

考虑一个例子。在图 11.1 里假定调用 getaddrinfo 时的参数是：一个有两个 IP 地址的主机，一个名为 domain 的服务(TCP 和 UDP 的 53 号端口)，没有指定套接口类型。getaddrinfo 函数(图 11.25)中的循环会创建两个 addrinfo 结构，即为由 gethostbyname 返回的每个 IP 地址创建一个。因为没有指定套接口类型，所以每个地址结构中也设置了 AI_CLONE 标志。在图 11.29 中展示了结果链表的结构。

在图 11.26 中调用了 ga_serv。因为 domain 服务对 TCP 和 UDP 都有效，所以调用了两次 getservbyname 以及两次 ga_port：前一次调用最后一个参数为 SOCK_STREAM，后一次调用为 SOCK_DGRAM。第一次调用 ga_port 时，开始时的链表如图 11.29。在图 11.28 中两个结构都设置了 AI_CLONE 选项，但套接口类型为 0。因此第一次调用 ga_port 后会将每个 addrinfo 结构的 ai_socktype 成员置为 SOCK_STREAM，套接口地址结构中的端口号则置为 53。AI_CLONE 标志保持原设置。这样就得到了如图 11.30 中的链表。

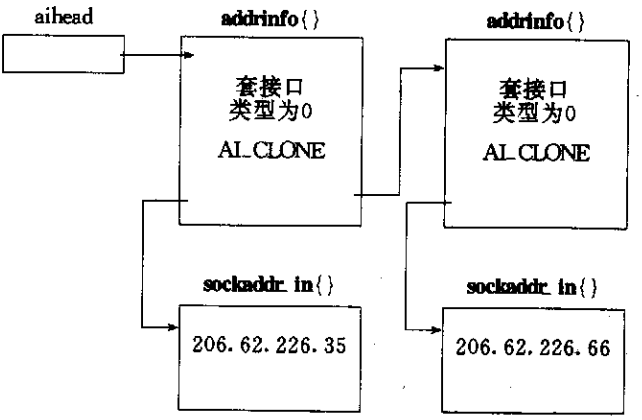


图 11.29 第一次调用 ga_port 时的 addrinfo 结构

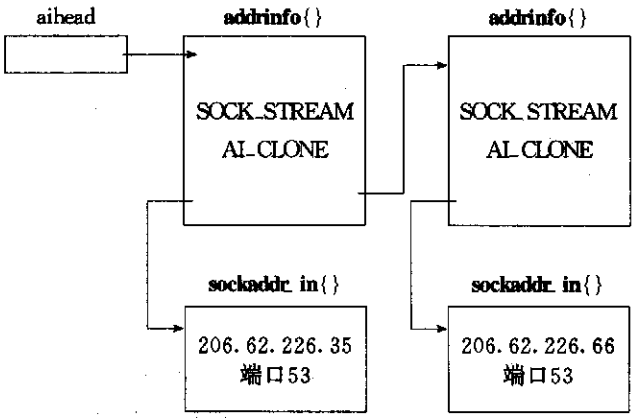


图 11.30　第一次调用 ga_port 后的 addrinfo 结构

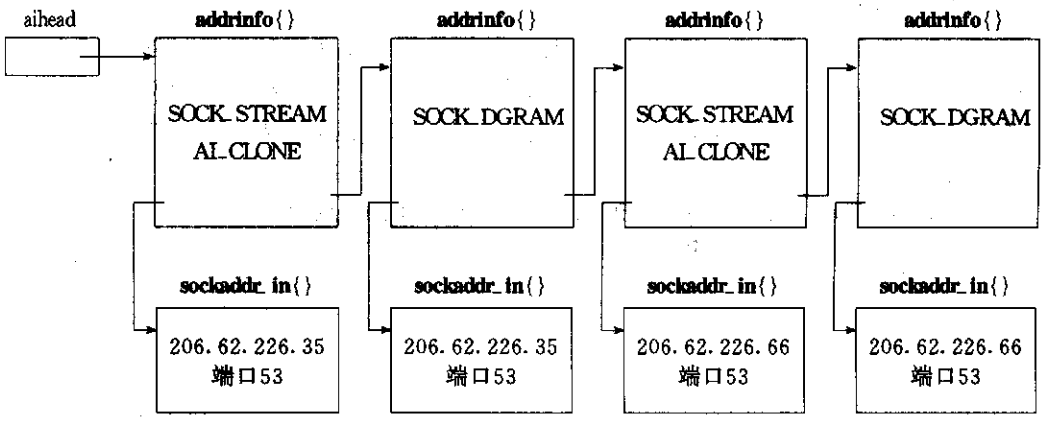


图 11.31　第二次调用 ga_port 后的 addrinfo 结构

但第二次调用 ga_port 时(最后一个参数为 SOCK_DGRAM),因为设置了 AI_CLONE 标志且套接口类型不为 0,所以对每个 addrinfo 结构调用了 ga_clone。在新复制的结构中的 ai_socktype 成员的值设为 SOCK_DGRAM,结果链表如图 11.31 所示。在该图中第二个 addrinfo 结构和它的套接口地址结构是从第一个结构复制而来的,第四个 addrinfo 结构和它的套接口地址结构是从第三个结构复制而来的。

图 11.32 是 ga_clone 函数的源代码,在图 11.28 中调用它从已有的结构集复制新的 addrinfo 结构及其套接口地址结构。

```
 5 struct addrinfo *
 6 ga_clone(struct addrinfo *ai)
 7 {
 8     struct addrinfo    *new;
 9     if ( (new = calloc(1, sizeof(struct addrinfo))) == NULL)
10                 return(NULL);
11     new->ai_next = ai->ai_next;
12     ai->ai_next = new;
13     new->ai_flags = 0;          /* make sure AI_CLONE is off */
14     new->ai_family = ai->ai_family;
```

```
15      new->ai_socktype = ai->ai_socktype;
16      new->ai_protocol = ai->ai_protocol;
17      new->ai_canonname = NULL;
18      new->ai_addrlen = ai->ai_addrlen;
19      if ( (new->ai_addr = malloc(ai->ai_addrlen)) == NULL)
20          return(NULL);
21      memcpy(new->ai_addr, ai->ai_addr, ai->ai_addrlen);
22      return(new);
23  }
```

图 11.32 ga_clone 函数[libgai/ga_clone.c]

分配 addrinfo 结构并将其插入到链表中

第 9～12 行 分配一个新的 addrinfo 结构，把它的 ai_next 指针的值设为被复制的项(也就是将排在链表中新结构前面的那个结构)的 ai_next 指针。被复制项的 ai_next 指针则变成指向新分配的结构。

初始化新复制的项

第 13～22 行 新 addrinfo 结构中的所有字段都是从被复制的项中拷贝而来，不过 ai_flags 被设成 0，ai_canonname 被设成空指针。函数的返回值为一个指向新创建结构的指针。

图 11.33 为 ga_unix 函数的源代码，在图 11.20 中处理 Unix 域路径名时调用过它。

```
3  int
4  ga_unix(const char *path, struct addrinfo *hintsp, struct addrinfo **result)
5  {
6      int         rc;
7      struct addrinfo    *aihead, **aipnext;
8      aihead = NULL;
9      aipnext = &aihead;
10     if (hintsp->ai_family != AF_UNSPEC && hintsp->ai_family != AF_LOCAL)
11         return(EAI_ADDRFAMILY);
12     if (hintsp->ai_socktype == 0) {
13             /* no socket type specified; return stream then dgram */
14         hintsp->ai_socktype = SOCK_STREAM;
15         if ( (rc = ga_aistruct(&aipnext, hintsp, path, AF_LOCAL)) != 0)
16             return(rc);
17         hintsp->ai_socktype = SOCK_DGRAM;
18     }
19     if ( (rc = ga_aistruct(&aipnext, hintsp, path, AF_LOCAL)) != 0)
20         return(rc);
21     if (hintsp->ai_flags & AI_CANONNAME) {
22         struct utsname     myname;
23         if (uname(&myname) < 0)
24             return(EAI_SYSTEM);
25         if ( (aihead->ai_canonname = strdup(myname.nodename)) == NULL)
26             return(EAI_MEMORY);
27     }
28     *result = aihead;       /* pointer to first structure in linked list */
29     return(0);
30 }
```

图 11.33 ga_unix 函数[libgai/ga_unix.c]

ga_aistruct 创建结构

第 10～20 行　如果没有指定套接口类型，则调用两次 ga_aistruct 函数创建两个 addrinfo 结构：一次用 SOCK_STREAM 套接口类型，另一次用 SOCK_DGRAM 套接口类型。但如果调用者指定了一个非零的套接口类型，就只调用一次 ga_aistruct 函数，创建一个所指定套接口类型的 addrinfo 结构。

返回规范主机名

第 21～27 行　如果调用者设置了 AI_CANONNAME 标志，则调用 uname 获取系统名，并将其中的 nodename 成员作为主机名字返回。

我们用下面介绍的 ga_aistruct 函数来解释 aihead 和 aipnext 指针。

ga_aistruct 函数在图 11.24 和 11.25 中被用来创建一个 IPv4 或 IPv6 的 addrinfo 结构，在图 11.33 中用来创建一个 Unix 域套接口的 addrinfo 结构。图 11.34 列出了这个函数源代码的第一部分。

```
5 int
6 ga_aistruct(struct addrinfo  * * * paipnext, const struct addrinfo  * hintsp,
7                 const void  * addr, int family)
8 {
9     struct addrinfo     * ai;
10    if ( (ai = calloc(1, sizeof(struct addrinfo))) == NULL)
11        return(EAI_MEMORY);
12    ai->ai_next = NULL;
13    ai->ai_canonname = NULL;
14     * * paipnext = ai;
15     * paipnext = &ai->ai_next;
16    if ( (ai->ai_socktype = hintsp->ai_socktype) == 0)
17        ai->ai_flags |= AI_CLONE;
18    ai->ai_protocol = hintsp->ai_protocol;
```

图 11.34　ga_aistruct 函数：第一部分[libgai/ga_aistruct.c]

分配 addrinfo 结构并将其加到链表中

第 10～15 行　给一个 addrinfo 结构分配内存并将其加到已有的链表中。有两个指针用来建立这个链表：aihead 和 aipnext。它们都是在图 11.20 中(IPv4 或 IPv6 套接口)或图 11.33中(Unix 域套接口)进行空间分配和初始化的。aihead 被初始化成一个空指针，aipnext 则指向 aihead。如图 11.35 所示。

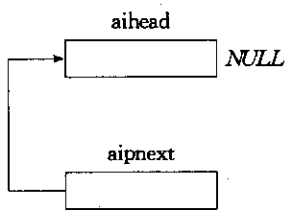


图 11.35　链表指针的初始化

aihead 总是指向链表中的第一个 addrinfo 结构(因此它的数据类型是 struct addrinfo *)。aipnext 通常指向链表中最后一个结构的 ai_next 成员(因此它的数据类型是 struct addrinfo * *)。这里我们对 aipnext 使用“通常”这个定语是因为初始化时它实际指向 aihead，

但在分配了第一个结构并加到链表中后，它就总是指向 ai_next 成员了。

回到 ga_aistruct 函数。在分配了一个新的结构后执行了下面两条语句：

```
* *paipnext = ai ;
*paipnext = &ai->ai_next;
```

第一条语句设置链表中最后一个结构的 ai_next 指针(如果这是链表中的第一个结构则设置 aihead 指针)，使其指向新分配的结构，第二条语句将 aipnext 设成指向新分配的结构的 ai_next 成员。因为它们是作为函数的参数，所以又加上了一层间接引用(见习题 11.4)。在链表中加入第一个结构后，数据结构如图 11.36 所示。

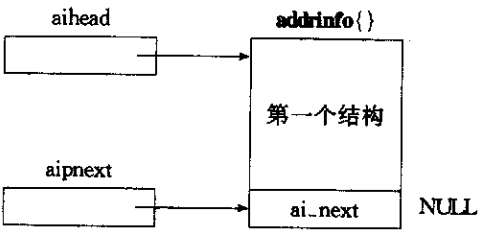


图 11.36 加入第一个结构后的链表

在下一次调用 ga_aistruct 函数分配第二个结构并加入链表后，数据结构如图 11.37 所示。

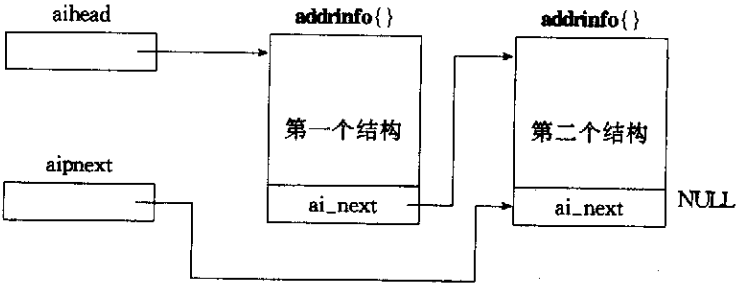


图 11.37 加入第二个结构后的链表

设置套接口类型

第 16～17 行 ai_socktype 成员设置成由调用者提供的套接口类型，如果为 0，则设置 AI_CLONE 标志。

图 11.38 是这个函数的第二部分：使用 switch 语句，每个 case 对应一种地址族，分别分配套接口地址结构并进行初始化。

```
19      switch ((ai->ai_family = family)) {
20      case AF_INET: {
21              struct sockaddr_in      *sinptr;
22                  /* allocate sockaddr_in{} and fill in all but port */
23              if ( (sinptr = calloc(1, sizeof(struct sockaddr_in))) == NULL)
24                      return(EAI_MEMORY);
25 #ifdef     HAVE_SOCKADDR_SA_LEN
26              sinptr->sin_len = sizeof(struct sockaddr_in);
27 #endif
28              sinptr->sin_family = AF_INET;
```

```
29                  memcpy(&sinptr->sin_addr, addr, sizeof(struct in_addr));
30                  ai->ai_addr = (struct sockaddr *) sinptr;
31                  ai->ai_addrlen = sizeof(struct sockaddr_in);
32                  break;
33              }
34      case AF_INET6: {
35                  struct sockaddr_in6     *sin6ptr;
36                      /* allocate sockaddr_in6{} and fill in all but port */
37                  if ( (sin6ptr = calloc(1, sizeof(struct sockaddr_in6))) == NULL)
38                          return(EAI_MEMORY);
39 #ifdef      HAVE_SOCKADDR_SA_LEN
40                  sin6ptr->sin6_len = sizeof(struct sockaddr_in6);
41 #endif
42                  sin6ptr->sin6_family = AF_INET6;
43                  memcpy(&sin6ptr->sin6_addr, addr, sizeof(struct in6_addr));
44                  ai->ai_addr = (struct sockaddr *) sin6ptr;
45                  ai->ai_addrlen = sizeof(struct sockaddr_in6);
46                  break;
47              }
48      case AF_LOCAL: {
49                  struct sockaddr_un      *unp;
50                      /* allocate sockaddr_un{} and fill in */
51                  if (strlen(addr) >= sizeof(unp->sun_path))
52                      return(EAI_SERVICE);
53                  if ( (unp = calloc(1, sizeof(struct sockaddr_un))) == NULL)
54                      return(EAI_MEMORY);
55                  unp->sun_family = AF_LOCAL;
56                  strcpy(unp->sun_path, addr);
57 #ifdef      HAVE_SOCKADDR_SA_LEN
58                  unp->sun_len = SUN_LEN(unp);
59 #endif
60                  ai->ai_addr = (struct sockaddr *) unp;
61                  ai->ai_addrlen = sizeof(struct sockaddr_un);
62                  if (hintsp->ai_flags & AI_PASSIVE)
63                      unlink(unp->sun_path);  /* OK if this fails */
64                  break;
65              }
66      }
67      return(0);
68 }
```

图 11.38　ga_aistruct 函数：第二部分[libgai/ga_aistruct.c]

分配并初始化 IPv4 套接口地址结构

第 20～33 行　分配一个 sockaddr_in 结构，将 addrinfo 结构中的 ai_addr 指针指向它。初始化套接口地址结构中的 IP 地址、地址族和长度等成员。在调用 ga_serv 前不进行端口号的初始化，因为 ga_serv 会调用 ga_port。

分配并初始化 IPv6 套接口地址结构

第 34～47 行　与 IPv4 类似，分配并初始化一个 sockaddr_in6 结构。

分配并初始化 Unix 域套接口地址结构

第 48～65 行　分配并初始化一个 sockaddr_un 结构。该地址是一个路径名，如果调用

者设置了 AI_PASSIVE 标志，程序中会试图 unlink 这个路径名以防调用 bind 时出错。即使 unlink 失败也没有什么问题。

图 11.39 里的 ga_echeck 函数曾在图 11.20 中被调用，以对调用者的参数进行一些初步的错误检查。

```
 5 int
 6 ga_echeck(const char *hostname, const char *servname,
 7            int flags, int family, int socktype, int protocol)
 8 {
 9     if (flags & ~(AI_PASSIVE | AI_CANONNAME))
10         return(EAI_BADFLAGS);   /* unknown flag bits */
11     if (hostname == NULL || hostname[0] == '\0') {
12         if (servname == NULL || servname[0] == '\0')
13             return(EAI_NONAME);     /* host or service must be specified */
14     }
15     switch(family) {
16         case AF_UNSPEC:
17             break;
18         case AF_INET:
19             if (socktype != 0 &&
20                 (socktype != SOCK_STREAM &&
21                  socktype != SOCK_DGRAM &&
22                  socktype != SOCK_RAW))
23                 return(EAI_SOCKTYPE);     /* invalid socket type */
24             break;
25     case AF_INET6:
26         if (socktype != 0 &&
27             (socktype != SOCK_STREAM &&
28              socktype != SOCK_DGRAM &&
29              socktype != SOCK_RAW))
30             return(EAI_SOCKTYPE);         /* invalid socket type */
31         break;
32     case AF_LOCAL:
33         if (socktype != 0 &&
34             (socktype != SOCK_STREAM &&
35              socktype != SOCK_DGRAM))
36             return(EAI_SOCKTYPE);         /* invalid socket type */
37         break;
38         default:
39             return(EAI_FAMILY);           /* unknown protocol family */
40     }
41     return(0);
42 }
```

图 11.39 ga_echeck 函数[libgai/ga_echeck.c]

第 9～14 行 检查各标志以及是否指定了主机名和服务名。

第 15～41 行 各个地址族只支持某些特定的套接口类型，这也是对套接口类型的检查。

这里没有检查调用者的 ai_protocol 线索，因为很少有程序指定它的值(作为函数 socket 的第三个参数)。要是指定了无效的组合，譬如套接口类型为 SOCK_DGRAM，而协议为 IPPROTO_TCP，则在图 11.34 中将协议线索返回给调用者，因而如果用这个值调用 socket，

将返回一个 EPROTONOSUPPORT 错误。

到这里已经完成了 getaddrinfo 和它内部调用的所有函数。图 11.40 是 freeaddrinfo 函数的源代码，它释放链表使用的所有内存空间。在图 11.26 中，一旦出错就调用该函数，用户也可以调用它来释放一个结构链表。

```
1 #include    "gai_hdr.h"
2 void
3 freeaddrinfo(struct addrinfo *aihead)
4 {
5     struct addrinfo    *ai, *ainext;
6     for (ai = aihead; ai != NULL; ai = ainext) {
7         if (ai->ai_addr != NULL)
8             free(ai->ai_addr);          /* socket address structure */
9         if (ai->ai_canonname != NULL)
10            free(ai->ai_canonname);
11        ainext = ai->ai_next;           /* can't fetch ai_next after free() */
12        free(ai);                       /* the addrinfo{} itself */
13    }
14 }
```

图 11.40　freeaddrinfo 函数：第一部分[libgai/freeaddrinfo.c]

第 6～13 行　遍历 addrinfo 结构链表，释放每个已分配的套接口地址结构。释放已分配的规范主机名字符串。最后释放 addrinfo 结构本身。必须在释放结构前保存其中 ai_next 指针的内容，因为在 free 返回后就不能再引用这个结构了。

图 11.41 是 getnameinfo 函数的具体实现。它由一个 switch 语句组成，每种地址族对应一个 case。

```
2 int
3 getnameinfo(const struct sockaddr *sa, socklen_t salen,
4             char *host, size_t hostlen,
5             char *serv, size_t servlen, int flags)
6 {
7     switch (sa->sa_family) {
8     case AF_INET: {
9             struct sockaddr_in *sain = (struct sockaddr_in *) sa;
10            return(gn_ipv46(host, hostlen, serv, servlen,
11                            &sain->sin_addr, sizeof(struct in_addr),
12                            AF_INET, sain->sin_port, flags));
13    }
14    case AF_INET6: {
15            struct sockaddr_in6 *sain = (struct sockaddr_in6 *) sa;
16            return(gn_ipv46(host, hostlen, serv, servlen,
17                            &sain->sin6_addr, sizeof(struct in6_addr),
18                            AF_INET6, sain->sin6_port, flags));
19    }
20    case AF_LOCAL: {
21            struct sockaddr_un *un = (struct sockaddr_un *) sa;
22            if (hostlen > 0)
23                snprintf(host, hostlen, "%s", "/local");
24            if (servlen > 0)
```

```
25                  snprintf(serv, servlen, "%s", un->sun_path);
26              return(0);
27          }
28      default:
29          return(1);
30      }
31  }
```

图 11.41 getnameinfo 函数[libgai/getnameinfo.c]

处理 IPv4 和 IPv6 套接口地址结构

第 8～19 行 调用 gn_ipv46 函数(将在下面介绍)处理 IPv4 和 IPv6 套接口地址结构。

处理 Unix 域套接口地址结构

第 20～27 行 对于 Unix 域套接口地址结构,返回/local 作为主机名,返回绑定在套接口上的路径名作为服务名。如果套接口没有绑定路径名,返回的服务名将是个空串。

> 我们用 snprintf 代替 strncpy 返回主机名和服务名。如果用后者的话可以这样写
>
> ```
> strncpy(host, "/local", hostlen);
> ```
>
> 虽然这样可以保证不使调用者的缓冲区溢出,但如果 hostlen 小于或等于 6,调用者的缓冲区将不是以空字符结尾。因为 getnameinfo 是一个库函数,所以应该按调用者的要求返回一个以空字符结尾的字符串,否则会导致调用者的程序以后出问题。因此要写成:
>
> ```
> strncpy(host, "/local", hostlen-1);
> host[hostlen-1] = '\0';
> ```
>
> 这既保证不让缓冲区溢出,又使结果串以空字符结尾。我们用 snprintf 代替这两条语句以达到同样的目的。另一种办法是定义一个调用 strncpy 并以空字符终止结果串的库函数,但调用现成的 snprintf 似乎更简单些。

图 11.42 是 gn_ipv46 函数的源代码,它可以为 getnameinfo 处理 IPv4 和 IPv6 套接口地址结构。

返回主机名

第 12～23 行 如果设置了 NI_NUMERICHOST 标志,就调用 inet_ntop 返回表达格式的 IP 地址;否则 gethostbyaddr 查找与该 IP 地址对应的主机名。如果 gethostbyaddr 成功,而且设置了 NI_NOFQDN(无需全限定域名)标志的话,主机名在第一个点号前被截断。

gethostbyname 失败处理

第 24～29 行 如果 gethostbyname 失败(在因特网上配置不当的 DNS 服务器不少;见 TCPv3 的 14.8 节),而且设置了 NI_NAMEREQD 标志,则返回错误。否则返回对该 IP 地址用 inet_ntop 处理所形成的地址串。

返回服务字符串

第 32～42 行 如果设置了 NI_NUMERICSERV 标志,就只返回十进制数的端口号;否则调用 getservbyport。除非设置了 NI_DGRAM 标志,最后一个参数将是空指针。如 get-

servbyport 失败，则返回十进制数形式的端口号。

```
 5 int
 6 gn_ipv46(char *host, size_t hostlen, char *serv, size_t servlen,
 7           void *aptr, size_t alen, int family, int port, int flags)
 8 {
 9     char       *ptr;
10     struct hostent    *hptr;
11     struct servent    *sptr;
12     if (hostlen > 0) {
13         if (flags & NI_NUMERICHOST) {
14             if (inet_ntop(family, aptr, host, hostlen) == NULL)
15                 return(1);
16         } else {
17             hptr = gethostbyaddr(aptr, alen, family);
18             if (hptr != NULL && hptr->h_name != NULL) {
19                 if (flags & NI_NOFQDN) {
20                     if ( (ptr = strchr(hptr->h_name, '.')) != NULL)
21                         *ptr = 0;    /* overwrite first dot */
22                 }
23                 snprintf(host, hostlen, "%s", hptr->h_name);
24             } else {
25                 if (flags & NI_NAMEREQD)
26                     return(1);
27                 if (inet_ntop(family, aptr, host, hostlen) == NULL)
28                     return(1);
29             }
30         }
31     }
32     if (servlen > 0) {
33         if (flags & NI_NUMERICSERV) {
34             snprintf(serv, servlen, "%d", ntohs(port));
35         } else {
36             sptr = getservbyport(port, (flags & NI_DGRAM) ? "udp" : NULL);
37             if (sptr != NULL && sptr->s_name != NULL)
38                 snprintf(serv, servlen, "%s", sptr->s_name);
39             else
40                 snprintf(serv, servlen, "%d", ntohs(port));
41         }
42     }
43     return(0);
44 }
```

图 11.42　gn_ipv46 函数：处理 IPv4 和 IPv6 套接口地址结构[libgai/gn_ipv46.c]

11.17　小　结

在编写独立于协议的代码时 getaddrinfo 是个很有用的函数。但直接调用它要花好几步，而且对于不同的情况存在一些都要处理的细节：扫描所有返回的结构，忽略 socket 返回的错误，为 TCP 服务器设置 SO_REUSEADDR 套接口选项，诸如此类。为简化这些细节，我们编写了五个函数：tcp_connect、tcp_listen、udp_client、udp_connect、udp_server。并在独立于协议的时间/日期服务器程序和客户程序的 TCP 和 UDP 版本中展示了这些函数的用法。

gethostbyname 和 gethostbyaddr 也是不可重入函数的例子。这两个函数共享一个静态的结构，返回一个指向该结构的指针。在第 23 章中介绍线程时还会遇到并讨论这个重入问题。我们介绍了一些厂商提供的这两个函数的_r 版本，这是一种解决方法，但是需要对调用这些函数的所有应用程序进行修改。

11.18 习 题

11.1 在图 11.8 中调用者必须传递一个整数指针以得到协议地址的大小。如果调用者没有这么做的话(譬如，传递的最后一个参数为空指针)，调用者还能怎样得到协议地址的大小呢？

11.2 修改图 11.10 中的程序，用 getnameinfo 代替 sock_ntop。应给 getnameinfo 传递那些标志？

11.3 在 7.5 节中我们讨论了使用 SO_REUSEADDR 盗用端口。为了弄清它是怎样工作的，构造一个图 11.15 所示的独立于协议的 UDP 时间/日期服务器程序。在一个窗口中启动该服务器的一个实例，给它捆绑通配地址和某个你选择的端口。在另一个窗口中启动一个客户并验证服务器正在处理客户请求(注意服务器的 printf)。再在第三个窗口中启动服务器的另一个实例，这次给它捆绑该主机的一个单播地址和与第一个服务器相同的端口。马上会遇到什么问题？修复这个问题并重新启动第二个服务器。启动一个客户，发送一个数据报，验证第二个服务器已盗用了第一个服务器的端口。如果可能，用与启动第一个服务器使用的登录账号不同的另一个账号再启动第二个服务器，看能否成功盗用端口，因为一些厂商只允许用户 ID 相同的进程第二次捆绑原来进程已绑定的端口。

11.4 讨论图 11.34 时我们说过 aipnext 的地址是 ga_aistruct 函数的一个参数，从而迫使对其又加上了一层间接的引用。为什么不把 aipnext 变成一个全局变量，以取代作为参数传递它的地址？

11.5 在 11.5 节对 Unix 域的讨论中我们提到没有一个 IANA 的服务名以斜杠开头。这些服务名中有包含斜杠的吗？

11.6 在 2.10 节的末尾举了两个 telnet 的例子：分别连往 daytime 服务器和 echo 服务器。了解客户要经过 gethostbyname 和 connect 这两步后，你能判定它的哪些输出行对应哪个步骤吗？

11.7 当不能给一个 IP 地址找出对应的主机名时，gethostbyaddr 可能要花很长时间(可长达 80 秒)才能返回错误。编写一个新的名为 getnameinfo_timeo 的函数，它有一个附加的整数参数指定等待应答的最大秒数。如果超时，而且没有设置 NI_NAMEREQD 标志，就调用 inet_ntop 返回一个地址串。

第 12 章　守护进程和 inetd 超级服务器

12.1　概　述

守护进程(daemon)是在后台运行不受终端控制的进程。Unix 系统中一般有很多守护进程在后台运行(20 到 50 个),执行不同的管理任务。

想要脱离所有的终端的原因是守护进程可能是从终端上启动(与从初始化脚本中启动相反),在这之后这个终端要能用来执行其他任务。举例来说,如果在某终端上启动了一个守护进程后从终端上注销,其他人又从该终端登录,那么任何守护进程的错误信息不应在后面用户的终端会话过程中出现。同样,由终端上的一些键所产生的信号(如中断信号),不应对以前从该终端上启动的任何守护进程造成影响。虽然使服务器程序在后台运行很容易(只要在 shell 命令行的结尾加一个 & 符号),但是我们还是应该使程序能自动转到后台并且脱离与终端的联系。

有好几种方法启动守护进程:

1. 在系统启动时很多守护程序都是由系统初始化脚本启动。这些脚本一般在/etc 目录或以/etc/rc 开头的目录下,它们的位置和内容依赖于具体的实现。由这些脚本启动的守护进程在开始时都拥有超级用户权限。
   有几个网络服务器一般从这些脚本启动:inetd 超级服务器(见下一步)、Web 服务器或邮件服务器(一般为 sendmail)。12.2 节中介绍的 syslogd 守护进程通常也是由某个这样的脚本启动的。
2. 许多网络服务器是由 inetd 超级服务器启动的,在这一节的后面会对此进行介绍。inetd 自己是由上一步中的某个脚本启动的。inetd 监听网络请求(Telnet、FTP 等),当请求到来时启动实际的服务器(Telnet 服务器、FTP 服务器等)。
3. cron 守护进程按规则定期执行一些程序,由它启动的程序也以守护进程的方式运行。cron 自己是在系统启动过程中由第 1 步启动的。
4. 可用 at 命令指定在将来的某一时刻执行程序。cron 守护进程在到达相应的时间时会启动这些程序,所以它们是以守护程序的方式运行的。
5. 不管是在前台还是在后台,守护进程也可以在用户终端上启动,这在测试守护进程或守护进程因某些原因终止而要重启时经常使用。

由于守护进程没有控制终端,在发生问题时它要用一些其他方式以输出消息。这些消息既有一般的通告消息,也有需管理员处理的紧急事件消息。syslog 函数是输出这些消息的标准方式,它将消息发往 syslogd 守护进程。

12.2 syslogd 守护进程

Unix 系统通常会从一个初始化脚本中启动名为 syslogd 的守护进程，只要系统不停止它就一直运行。源自 Berkeley 的 syslogd 的实现在启动时执行以下操作：

1. 读入配置文件，通常是/etc/syslog.conf，它设定守护进程对接收每次键入的各种登记消息(log message)怎样处理。这些消息可能被写入一个文件(一种特殊情况是文件为/dev/console，这将把消息写到控制台上)，或发给指定的用户(如果该用户已登录到系统)，或转发给另一台主机上的 syslogd 进程。
2. 创建一个 Unix 域套接口，给它捆绑路径名/var/run/log(在某些系统上是/dev/log)。
3. 创建一个 UDP 套接口，给它捆绑端口 514(syslog 服务使用的端口号)。
4. 打开路径名/dev/klog，内核中的所有出错消息作为这个设备的输入出现。

在这之后 syslogd 进程运行一个无限循环，循环中调用 select，等待三个描述字(以上第2、3、4步生成的描述字)之一变为可读，读入登记消息，并按配置文件对消息进行处理。如果该守护进程收到 SIGHUP 信号，它会重新读入配置文件。

通过建立一个 Unix 域数据报套接口，并向 syslogd 守护进程绑定的路径名发送我们的消息，我们就能从自己的守护进程向 syslogd 发送登记消息，但更简单的接口是在下一节中介绍的 syslog 函数。另外也可以创建一个 UDP 套接口，将日志消息发到回馈地址及端口 514。

> 在新的实现中除非管理员指定，将不创建 UDP 套接口，这是因为允许任何人向这个端口发送 UDP 数据报(有可能填满它的套接口接收缓冲区)可能导致收不到真正需要处理的登记消息。
>
> 在 syslogd 的各种实现中存在差异。举例来说，源自 Berkeley 的实现使用 Unix 域套接口，而系统 V 使用一个基于流的登记驱动程序[①]。各种源自 Berkeley 的实现中使用的 Unix 域套接口的路径名也各不相同。但是如使用 syslog 函数就可以忽略这些细节问题。

12.3 syslog 函数

因为守护进程没有控制终端，所以它不能 fprintf 到 stderr 上。守护进程为登记消息通常调用 syslog 函数。

---

① 译者注：注意区别这里的流(streams)和我们一直在使用的字节流(stream)。前者是一种访问驱动程序(driver)的方法，在本书第33章介绍，它也称为 STREAMS。后者是与数据报(datagram)相对的数据传送方式，我们把它译成字节流一方面避免了与流相混淆，另一方面是强调它是无记录边界的数据流(不同于数据报方式的数据流)，或者说它的记录单元是最小的字节(而数据报方式的记录单元就是每个数据报本身)。另外一个应避免混淆的概念是标准 I/O 函数库中的标准 I/O 流(standard I/O streams)，它在本书中出现得极少。

```
#include <syslog.h>
void syslog(int priority,const char *message, ...);
```

尽管这个函数最初是为 BSD 系统开发的，但现在大多数 Unix 厂商都有提供。Posix 中没有提到 syslog，但在 Unix 98 中它是必需的。

在图 12.1 和 12.2 中出现的 priority 参数是级别(level)和设施(facility)的组合。message 与 printf 所用的格式化字符串类似，还增加了%m，它将由对应当前 errno 值的出错消息所取代。在消息的结尾可以加换行符，但这不是必需的。

如图 12.1 所示，登记消息的级别可从 0 到 7，它们是按从高到低的顺序排列的。如果发送者没有指定级别值，就缺省为 LOG_NOTICE。

级别(level)	值	描述
LOG_EMERG	0	系统不可用(优先级最高)
LOG_ALERT	1	必须立即进行处理
LOG_CRIT	2	危险情况
LOG_ERR	3	出错情况
LOG_WARNING	4	警告性情况
LOG_NOTICE	5	常见但值得注意的情况(缺省)
LOG_INFO	6	通告消息
LOG_DEBUG	7	调试消息(优先级最低)

图 12.1　登记消息的级别

登记消息还包含一个标识发送消息进程的类型的设施。我们在图 12.2 中列出了它的各种值。如果没有指定设施，就缺省为 LOG_USER。

设施(facility)	描述
LOG_AUTH	安全/授权消息
LOG_AUTHPRIV	安全/授权消息(私有的)
LOG_CRON	cron 守护进程
LOG_DAEMON	系统守护进程
LOG_FTP	FTP守护进程
LOG_KERN	内核消息
LOG_LOCAL0	本地使用
LOG_LOCAL1	本地使用
LOG_LOCAL2	本地使用
LOG_LOCAL3	本地使用
LOG_LOCAL4	本地使用

（续）

设施(facility)	描述
LOG_LOCAL5	本地使用
LOG_LOCAL6	本地使用
LOG_LOCAL7	本地使用
LOG_LPR	行式打印机系统
LOG_MAIL	邮件系统
LOG_NEWS	网络新闻系统
LOG_SYSLOG	由 syslogd 内部产生的消息
LOG_USER	任意的用户级消息(缺省)
LOG_UUCP	UUCP 系统

图 12.2 登记消息的设施

举例来说，当调用 rename 函数失败时，守护进程可能会做以下调用：

```
syslog(LOG_INFO|LOG_LOCAL2, "rename(%s, %s): %m", file1, file2);
```

设施和级别的目的是，允许在/etc/syslog.conf 文件中进行配置，使得对相同设施的消息得到同样的处理，或使相同级别的消息得到同样的处理。举例来说，配置文件中可能有以下两行：

```
kern.*              /dev/console
local7.debug        /var/log/cisco.log
```

指定内核的所有消息登记到控制台上，所有设施为 local 7 的调试消息添加到/var/log/cisco.log 文件的末尾。

当应用程序第一次调用 syslog 时，它创建一个 Unix 域数据报套接口，然后调用 connect 连往 syslogd 守护进程建立的套接口的众所周知路径名(譬如/var/run/log)。这个套接口在进程终止前一直打开。另外，进程也可以调用 openlog 和 closelog。

```
#include <syslog.h>

void openlog(const char *ident, int options, int facility);

void closelog(void);
```

openlog 可在第一次调用 syslog 前调用，当应用程序不再需要发送登记消息时可调用 closelog。

ident 是一个字符串，它将被 syslog 加到每条登记消息的前面。一般情况下它的值为程序名。[②]。

options 参数由图 12.3 中的一个或多个常值的“逻辑或”组成。

② 作者注： 请留意 openlog 的大多数实现仅仅保存一个指向 ident 字符串的指针；它们不拷贝这个字符串。这就是说这个字符串不应该是在栈上分配的(自动变量就是这样)，因为以后调用 syslog 时如果相应的栈帧已弹走，所保存的指针就不再指向 ident 字符串。

选项(options)	描　述
LOG_CONS	如果不能发往 syslogd 守护进程，则登记到控制台上
LOG_NDELAY	不延迟打开，立即创建套接口
LOG_PERROR	既发往 syslogd 守护进程，又登记到标准错误输出
LOG_PID	登记每条消息的进程 ID

图 12.3　openlog 的选项

通常调用 openlog 时并不创建 Unix 域套接口。它是在第一次调用 syslog 时打开的。LOG_NDELAY 选项能使套接口在调用 openlog 时创建。

openlog 的 facility 参数为后面没有设置设施的 syslog 调用设置一个缺省值。一些守护进程调用 openlog 设置设施(它在一个进程中通常不变)，然后在调用 syslog 时只设置级别(因为级别会随错误而改变)。

logger 命令也能产生日志消息。例如，在 shell 脚本中可以用它向 syslogd 发送消息。

12.4　daemon_init 函数

图 12.4 给出了名为 daemon_init 函数，调用它(通常从服务器程序)可使一个进程变成守护进程。

fork

第 10～11 行　首先调用 fork，然后终止父进程，留下子进程继续运行。如果进程是以 shell 命令方式从前台启动，当父进程终止时，shell 就认为命令已完成。这可以自动使子进程在后台运行。子进程继承了父进程的进程组号，但它拥有自己的进程号。这就保证了这个子进程不是进程组头，这是下一步调用 setsid 所必需的。

setsid

第 12～13 行　setsid 是一个 Posix.1 函数，它创建一个新的登录会话(session)。(APUE 第 9 章讨论了关于进程之间的关系和会话的细节。)这个进程变成新会话的会话头和新进程组的组长，不再有控制终端。

忽略 SIGHUP 信号并再次 fork

第 14～16 行　忽略 SIGHUP 信号 并再次调用 fork。当这个函数返回时，正在运行的是第二次生成的子进程，第一次生成的子进程是它的父进程，而且已经终止。

第二次 fork 的目的是确保守护进程将来即使打开一个终端设备，也不会自动获得控制终端。在 SVR4 中，当没有控制终端的会话头进程打开终端设备时(这个终端现在不是其他会话的控制终端)，该终端自动成为这个会话头的控制终端。但通过第二次调用 fork，可以确保这次生成的子进程不再是一个会话的头，因此它不会获得控制终端。这里必须忽略 SIGHUP 信号是因为当会话头(第一次生成的子进程)终止时，该会话中的所有进程(第二次生成的子进程)都会收到 SIGHUP 信号。

```
1 #include      "unp.h"
2 #include      <syslog.h>
3 #define MAXFD     64
4 extern int daemon_proc;          /* defined in error.c */
5 void
6 daemon_init(const char *pname,int facility)
7 {
8     int         i;
9     pid_t       pid;
10    if( (pid = Fork()) != 0)
11        exit(0);                /* parent terminates */
12        /* 1st child continues */
13    setsid();
14    Signal(SIGHUP,SIG_IGN);
15    if ( (pid = Fork()) != 0)
16        exit(0);                /* 1st child terminates */
17        /* 2nd child continues */
18    daemon_proc = 1;            /* for our err_XXX() functions */
19    chdir("/");                 /* change working directory */
20    umask(0);                   /* clear our file mode creation mask */
21    for (i = 0; i < MAXFD; i++)
22        close(i);
23    openlog(pname,LOG_PID,facility);
24 }
```

图12.4 daemon_init函数:使进程变成一个守护进程[daemon_init.c]

为出错处理函数设置标识

第17～18行 将全局变量daemon_proc设成非零。这个外部变量是由我们的err_XXX函数(D.4节)定义的,当它的值为非零时,出错处理函数将调用syslog函数取代用fprintf输出到标准错误输出。这样我们不用从头到尾改程序,就可以让服务器在以非守护进程方式运行时(即测试服务器时)调用一个自己的出错处理函数,而在以守护进程方式运行时调用syslog。

改变工作目录并清除文件模式创建掩码

第19～20行 将工作目录改到根目录,不过确实有些守护进程有理由将它改到其他的目录。举例来说,打印机守护进程可能会把工作目录改为打印机的假脱机处理(spool)目录,即它工作时要使用的目录。守护进程如果产生core文件,这个core文件将在工作目录下。改变工作目录的另一个原因是守护进程可能在任意文件系统中启动,如果保持工作目录不变的话,该文件系统就不能拆卸(unmount)。文件模式创建掩码被重置成0,这样守护进程创建自己的文件时,新文件的权限位不受原先的文件模式创建掩码的权限位的影响。

关闭所有打开的文件描述字

第21～22行 关闭本守护进程从运行它的进程(通常为shell)继承而来的所有打开的文件描述字。问题是怎样得到正在使用的最大的文件描述字;没有一个Unix函数提供这个值。有方法能得到一个进程能打开的最大的描述字数目,但由于这个限制可能是无限的,所

以确定最大描述字本身也变得复杂起来(参见 APUE 第 43 页)。我们的解决办法是关闭前 64 个描述字,即使它们之中可能有很多并没有打开。

有些守护进程以读写方式打开/dev/null,并将它复制到标准输入、标准输出和标准错误输出。这样可以保证这些常用的描述字是打开的,从这些描述字读时会返回 0(文件结束符),向它们写时内核会丢弃所有写入的信息。之所以要打开这些描述字是因为,这样的话守护进程调用的那些假定能使用标准输入、标准输出和标准错误输出的库函数就不会发生问题。另外,有些守护进程打开一个登记文件,并将其描述字复制到标准输出和标准错误输出。

用 syslogd 处理错误

第 23 行　调用 openlog。调用者一般将第一个参数设成程序名(譬如 argv[0])。这里的设置将进程号加到每条登记消息中。这里还设定了 facility 参数,其值为图 12.2 中出现的常值,如果缺省值 LOG_USER 能满足要求的话可设为 0。

守护进程运行时没有控制终端,所以它不会收到来自内核的 SIGHUP 信号。因此很多守护进程将该信号作为管理员通知其配置文件已修改之用,守护进程收到 SIGHUP 信号后应重新读入配置文件。另外两个守护进程不应收到的信号是 SIGINT 和 SIGWINCH,这些信号也可用作通知守护进程一些其他事项。

例子:守护进程方式的时间/日期服务器程序

图 12.5 是从图 11.10 进行修改后的不依赖于协议的时间/日期服务器程序。它调用 daemon_init 函数变成一个守护进程。

```
 1 #include   "unp.h"
 2 #include   <time.h>
 3 int
 4 main(int argc, char * * argv)
 5 {
 6     int    listenfd, connfd;
 7     socklen_t    addrlen, len;
 8     struct sockaddr    * cliaddr;
 9     char        buff[MAXLINE];
10     time_t      ticks;
11     daemon_init(argv[0], 0);
12     if (argc == 2)
13         listenfd = Tcp_listen(NULL, argv[1], &addrlen);
14     else if (argc == 3)
15         listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
16     else
17         err_quit("usage: daytimetcpsrv2 [ <host> ] <service or port>");
18     cliaddr = Malloc(addrlen);
19     for ( ; ; ) {
20         len = addrlen;
21         connfd = Accept(listenfd, cliaddr, &len);
22         err_msg("connection from %s", Sock_ntop(cliaddr, len));
23       ticks = time(NULL);
24       snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
```

```
25        Write(connfd, buff, strlen(buff));
26        Close(connfd);
27    }
28 }
```

图 12.5 以守护进程方式运行的不依赖于协议的时间/日期服务器程序[inetd/daytimetcpsrv2.c]

只改了两个地方：当程序开始时尽快调用 daemon_init，并用自己的 err_msg 代替 printf 输出 IP 地址和端口号。实际上，如果想要程序以守护进程方式运行，就必须避免调用 printf 和 fprintf 函数，而以自己的 err_msg 函数代替。

如果在主机 solaris 上运行该程序，再从主机 bdsi 进行连接，然后检查/var/adm/messages 文件(所有 LOG_USER 消息发往的地方)，就可以在其中找到：

```
Jun  4 15:15:33 solaris.kohala.com daytimetcpsrv2[14882]
connection from ::ffff:206.62.226.35.3356
```

(这里因为行太长而折行了)。syslogd 守护进程在前面自动加上了日期、时间和 FQDN。

12.5 inetd 守护进程

在典型的 Unix 系统中都有很多服务器在运行，等待客户的请求。例如 FTP、Telnet、TFTP 等等。在 4.3BSD 版本前的 Unix 系统中，这些服务都有一个与之对应的进程。这些进程都是在系统启动时从/etc/rc 文件里启动，它们启动时所做的工作差不多一样：创建套接口，给它捆绑服务器的众所周知端口，等待连接(如果是 TCP)或数据报(如果是 UDP)，然后 fork。子进程为客户服务，父进程继续等待下一个客户请求。该模型有两个问题。

1. 这些守护进程都有几乎相同的启动代码，首先是创建套接口，还要考虑变成守护进程(与 daemon_init 函数类似)。
2. 每个守护进程在进程表中要占用一项，但它们在大部分时间里处于睡眠状态。

4.3BSD 版本通过提供一个因特网超级服务器：inetd 守护进程使这些问题得到简化。基于 TCP 或 UDP 的服务器都可以使用这个守护进程。它不处理其他的协议，如 Unix 域套接口。该守护进程解决了刚才提到的两个问题。

1. 因为大部分启动时要做的工作由 inetd 处理了，所以守护进程的编写得到简化。这避免了每个服务器程序都要调用 daemon_init 函数。
2. 单个进程(inetd)能为多个服务等待客户的请求，取代了每个服务一个进程的方式，这样减少了系统中的进程总数。

inetd 进程使用前面介绍的 daemon_init 函数中的技术将自己变成一个守护进程，然后读入并处理它的配置文件，通常为/etc/inetd.conf。这个文件配置超级服务器处理的服务，以及当一个服务请求到来时怎么做。文件中每行所包含的栏目如图 12.6 所示。下面是一些实例：

```
ftp     stream    tcp   nowait    root        /usr/bin/ftpd        ftpd -l
telnet  stream    tcp   nowait    root        /usr/bin/telnetd     telnetd
```

```
login   stream   tcp   nowait   root      /usr/bin/rlogind   rlogind -s
tftp    dgram    udp   wait     nobody    /usr/bin/tftpd     tftpd -s /tftpboot
```

服务器程序的实际名字总是在由 inetd 调用 exec 执行时作为第一个参数传递给它。

栏　目	说　明
service-name	必须是在/etc/services 文件中已定义的服务名
socket-type	stream(TCP)或 dgram(UDP)
protocol	必须在/etc/protocols 文件中已有定义:tcp 或 udp
wait-flag	一般 TCP 是 nowait,UDP 是 wait
login-name	/etc/passwd 中的用户名,一般为 root
server-program	exec 使用的全路径名
server-program-arguments	exec 使用的参数

图 12.6　inetd.conf 文件中的栏目

上图和示例行只是作为例子。许多厂商都在 inetd 中加入了一些自己的功能。如在 TCP 和 UDP 服务器之外,加入处理远程过程调用(RPC)服务器的能力,以及处理除 TCP 和 UDP 外其他协议的能力。当然,exec 服务器程序使用的路径名和命令行参数依赖于具体的实现。

IPv6 与/etc/inetd.conf 的交互依赖于厂商的实现。一些厂商用名为 tcp6 和 udp6 的 protocol 表示应为该服务创建一个 IPv6 套接口。

图 12.7 给出了 inetd 守护进程的工作流程:

1. 启动时读/etc/inetd.conf 文件并给文件中指定的所有服务创建一个相应类型的套接口(字节流或数据报)。inetd 能处理的服务器的数目依赖于它最多能创建的描述字的数目。每个新创建的套接口都被加入到 select 调用所用到的描述字集中。
2. 为每个套接口调用 bind,给它们捆绑服务器的众所周知端口和通配地址。它们的 TCP 或 UDP 端口号是通过调用 getservbyname 获得的,其中使用了配置文件中的 service-name 和 protocol 栏目作为参数。
3. 对 TCP 套接口调用 listen,以接受外来的连接请求。对数据报套接口则不做这一步。
4. 所有套接口建立后,调用 select 等待这些套接口变为可读。回想 6.3 节里提到过,当在 TCP 套接口上到来一个新的连接请求或在 UDP 套接口上到来一个数据报时它们会变成可读。inetd 在大部分时间里阻塞在 select 调用上,等待有一个套接口变成可读。
5. select 返回一个可读的套接口后,如果是一个 TCP 套接口,就调用 accept 接受这个新的连接。
6. inetd 守护进程 fork,由子进程处理服务请求。这和标准的并发服务器(4.8 节)类似。

子进程关闭除要处理的套接口描述字之外的所有描述字:对 TCP 服务器来说是由 accept 返回的新的已连接套接口,对 UDP 服务器则是最初的那个 UDP 套接口。

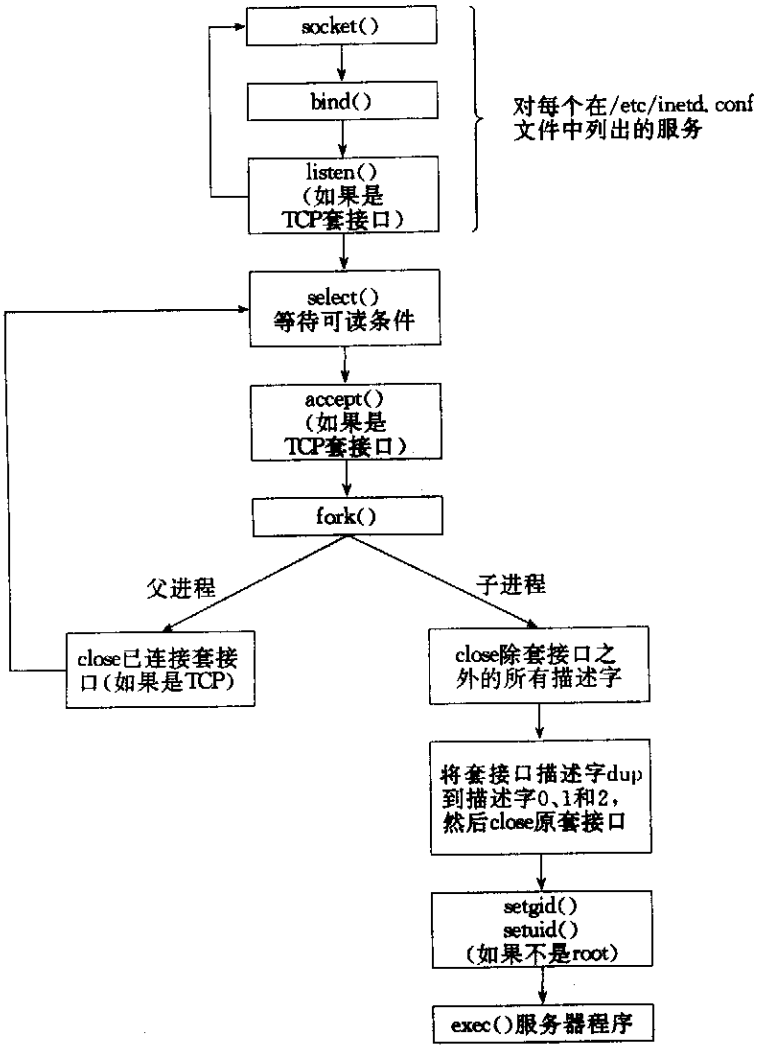


图 12.7 inetd 的工作流程

子进程三次调用 dup2,把套接口描述字复制到描述字 0、1 和 2(标准输入、标准输出和标准错误输出)。然后关闭原套接口描述字。这样,子进程打开的描述字就只有 0、1 和 2。如果子进程读标准输入,它实际是从套接口读,写标准输出和标准错误输出也是写到套接口。

子进程调用 getpwnam 得到在配置文件中指定的 login-name 对应的保密字文件项。如果 login-name 不是 root,子进程会调用 setgid 和 setuid 变为指定的用户。(因为 inetd 是以用户 ID 为 0 运行,子进程跨 fork 继承了这个用户 ID,所以它能变成任何用户。)

子进程用 exec 执行相应的 server-program 处理请求,并将配置文件中指定的参数传递给它。

7. 如果是一个字节流套接口,父进程必须关闭已连接套接口(就像标准并发服务器那

样)。父进程再调用 select 以等待下一个变成可读的套接口。

如果我们要更深入地了解对描述字的处理,图 12.8 展示了有一个新的从 FTP 客户来的连接请求时,inetd 中的描述字。

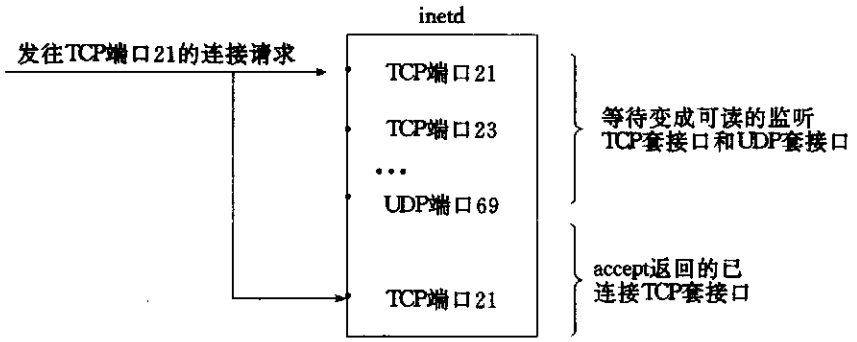


图 12.8　当 TCP 端口 21 的连接请求到来时,inetd 中的描述字

连接请求指向 TCP 的 21 号端口,但 accept 创建了一个新的已连接套接口。

图 12.9 展示了子进程在调用 fork 及关闭除已连接套接口描述字外的所有描述字之后剩下的描述字。

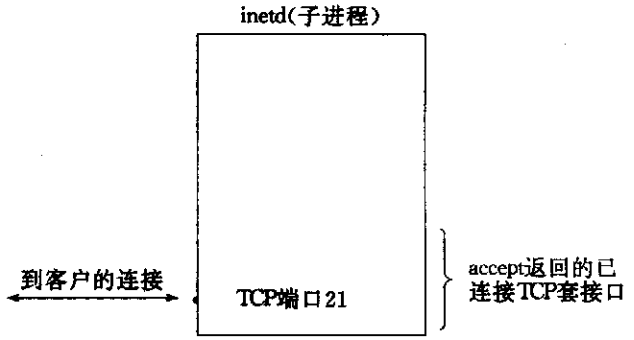


图 12.9　inetd 子进程中的描述字

下一步是子进程把已连接套接口描述字复制到描述字 0、1 和 2,然后关闭该套接口。如图 12.10 所示。

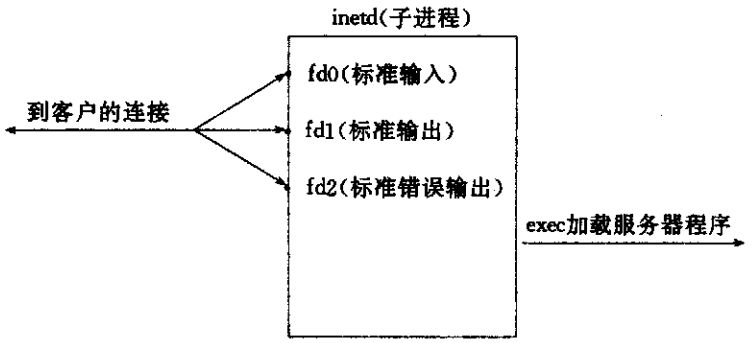


图 12.10　dup2 后 inetd 中的描述字

子进程接着调用 exec。回想 4.7 节，exec 后所有的描述字通常还是打开的，因此加载的实际服务器程序用描述字 0、1 或 2 与客户进行通信。服务器只打开这些描述字。

以上介绍的是对配置文件中设置为 nowait 方式的服务器的处理。对 TCP 服务来说这种情况十分典型，它意味着 inetd 在接受请求同一服务的其他连接之前不需要等待该服务的子进程终止。如果有同一服务的其他连接请求到来，父进程再调用 select 时就会将其返回。再次执行前面列出的第 4、第 5 和第 6 步操作，由另一个子进程处理这个新的请求。

给数据报服务设置 wait 标志需要对父进程所做的操作步骤作一定的修改。这个标志是说 inetd 在该 UDP 套接口上再次选择之前，必须等待在该套接口上服务的子进程终止。于是有下面的这些改变：

1. 父进程中的 fork 返回时，把子进程的进程号记录下来。这样在子进程终止时，父进程可以用 waitpid 的返回值查知是哪一个子进程。
2. 父进程用 FD_CLR 宏关闭 select 使用的描述字集中与这个套接口对应的位，以不对该套接口作 select。这意味着子进程接管这个套接口直至其终止。
3. 当子进程终止时，父进程收到一个 SIGCHLD 信号，父进程的信号处理程序得到终止子进程的进程号。父进程通过打开描述字集中对应的位恢复对该套接口的 select。

数据报服务器必须接管套接口直至它终止，防止 inetd 在这个套接口继续做可读性的 select（等待另一个客户的数据报），这是因为一个数据报服务器只有一个套接口，不像 TCP 服务器那样有一个监听套接口，每个客户有一个已连接套接口。如果 inetd 不关闭对这个数据报套接口的可读条件检查，而且父进程（inetd）在子进程前面执行，那么从客户来的数据报会还在套接口的接收缓冲区里，导致 select 再次返回可读，inetd 于是 fork 另一个（不需要的）子进程。inetd 必须在它确信子进程已从套接口接收队列中读走数据报前忽略这个数据报套接口。inetd 通过收到 SIGCHLD 信号的方式了解子进程何时终止。在 20.7 节中有这样的例子。

图 2.13 中介绍的五种标准因特网服务是由 inetd 内部处理的。（见习题 12.2。）

因为替 TCP 服务器调用 accept 的是 inetd，所以实际的服务器一般用 getpeername 获得客户的 IP 地址和端口号。回想图 4.18 中展示了在 fork 和 exec 后（inetd 就这么做）实际的服务器识别客户的唯一方法是调用 getpeername。

需要提供大量服务的服务器通常不用 inetd，其中值得注意的是邮件和 Web 服务器。举例来说，在 4.8 节中介绍的 sendmail 通常是以一个标准的并发服务器来运行。这种方式下每个客户连接在进程控制上的开销只是一个 fork，而一个通过 inetd 的 TCP 服务器的开销是一个 fork 和一个 exec。Web 服务器使用了多种技术把每个客户连接的进程控制开销减到最小，我们会在第 27 章中对此进行讨论。

12.6 daemon_inetd 函数

图 12.11 给出了一个名为 daemon_inetd 的函数，我们可以在一个由 inetd 启动的服务器程序中调用它。

```
1 #include    "unp.h"
```

```
2 #include    <syslog.h>
3 extern int daemon_proc;          /* defined in error.c */
4 void
5 daemon_inetd(const char *pname, int facility)
6 {
7     daemon_proc = 1;        /* for our err_XXX() functions */
8     openlog(pname,LOG_PID,facility);
9 }
```

图 12.11　daemon_inetd 函数：使通过 inetd 运行的进程变为守护进程[daemon_inetd.c]

这个函数与 daemon_init 比起来要简单，因为变成守护进程的工作都由 inetd 在启动时做过了。唯一要做的就是给错误处理函数(附录中的图 D.4)设置 daemon_proc 标志，并使用与图 12.4 中的调用相同的参数调用 openlog。

例子：由 inetd 启动的时间/日期服务器程序

图 12.12 是图 12.5 中的时间/日期服务器程序的一个修改版本，它可以由 inetd 启动。

```
1 #include    "unp.h"
2 #include    <time.h>
3 int
4 main(int argc, char **argv)
5 {
6       socklen_t          len;
7       struct sockaddr    *cliaddr;
8       char               buff[MAXLINE];
9       time_t                    ticks;
10      daemon_inetd(argv[0], 0);
11      cliaddr = Malloc(MAXSOCKADDR);
12      len = MAXSOCKADDR;
13      Getpeername(0, cliaddr, &len);
14      err_msg("connection from %s", Sock_ntop(cliaddr, len));
15    ticks = time(NULL);
16    snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
17    Write(0, buff, strlen(buff));
18      Close(0);    /* close TCP connection */
19      exit(0);
20 }
```

图 12.12　可由 inetd 启动的独立于协议的时间/日期服务器程序[inetd/daytimetcpsrv3.c]

这个程序中有两个大的改动。首先，所有创建套接口的代码即对 tcp_listen 和 accept 的调用都没有了。这些工作都由 inetd 做了，我们可以用描述字 0(标准输入)来引用 TCP 连接。其次，去掉了 for 的无限循环，因为对每个客户连接都会启动一次程序。服务完这个客户后程序即终止。

调用 getpeername

第 11～14 行　因为程序中没有调用 tcp_listen，所以不知道它返回的套接口地址结构的大小，而且由于没有调用 accept，也不知客户的协议地址。因此用 MAXSOCKADDR 常值

给套接口地址结构分配缓冲区，并用描述字0作第一个参数调用getpeername。

为了在BSD/OS系统上运行这个例子程序，首先给它指派一个名字和端口，将下面一行加到/etc/services文件中：

```
mydaytime      9999/tcp
```

然后在/etc/inetd.conf加一行：

```
mydaytime  stream  tcp  nowait  rstevens
    /usr/home/rstevens/daytimetcpsrv3  daytimetcpsrc3
```

(这行因过长而被折行。)把执行代码放在指定的文件中并向inetd发SIGHUP信号，通知它重新读入配置文件。下一步是执行netstat检验是否已创建了一个在TCP的端口9999上的监听套接口：

```
dsdi % netstat -na | grep 9999
tcp        0  0      *.9999          *.*          LISTEN
```

然后从另外一台主机访问这个服务器：

```
alpha % telnet bsdi 9999
Trying 206.62.226.35 ...
Connected to bsdi.
Escape character is '^]'.
Thu Jun 5 11:13:50 1997
Connection closed by foreign host.
```

在/var/log/messages文件(已在/etc/syslog.conf文件中指定LOG_USER设施的消息登记到该文件)中会有以下一项：

```
Jun 5 11:13:50 bsdi daytimetcpsrv3[28724]: connection from 206.62.226.42.1042
```

12.7 小 结

守护进程是在后台运行的，独立于所有终端控制的进程。多数网络服务器是以守护进程的方式运行。守护进程的所有输出通常是调用syslog函数发送给syslogd。管理员可以根据发送消息的守护进程以及消息的紧急程度，完全控制如何处理这些消息。

启动一个程序并使其以守护进程的方式运行需要以下步骤：调用fork以转到后台运行，调用setsid建立一个新的Posix.1会话并成为会话头，再次fork以避免获得新的控制终端，改变工作目录和文件模式创建掩码，并关闭所有不需要的文件。daemon_init函数处理了以上的这些问题。

多数Unix服务器是由inetd守护进程启动的。它处理了所有转变成守护进程所需的步骤，在启动真正的服务器时套接口已在标准输入、标准输出、标准错误输出上打开。这样我们就不必再调用socket、bind、listen和accpet了，因为这些操作inetd已经做了。

12.8　习　题

12.1　图 12.5 中如果直到处理完命令行参数后再调用 daemon_init，使调用 err_quit 在程序变成守护进程之前出现会发生什么？

12.2　对于由 inetd 内部处理的 5 种服务（图 2.13），考虑每种服务有 TCP 和 UDP 两个版本，你认为这 10 个服务器在实现时哪些需要调用 fork，哪些不需要调用 fork？

12.3　如果创建一个 UDP 套接口，将其绑定端口 7（图 2.13 中的标准 echo 服务器），然后向 chargen 服务器发送一个 UDP 数据报，将会发生什么？

12.4　Solaris 2.x 中 inetd 的手册页面中介绍了一个-t 标志，它会使 inetd 调用 syslog（用设施 LOG_DAEMON 和级别 LOG_NOTICE）登记所有 inetd 处理的 TCP 服务的客户的 IP 地址和端口号。inetd 是如何获得这个信息的？

该手册页面中还说 inetd 不能对 UDP 服务做上述的动作。为什么？有办法绕过对 UDP 服务的这个限制吗？

第13章 高级I/O函数

13.1 概 述

本章覆盖了各种被归为"高级I/O"的函数和技术。首先是为一个I/O函数设置超时,这有三种方法。

然后是三个read和write函数的变体:recv和send,它们可以把含有标志的第四个参数从进程传给内核;readv和writev,这两个函数可以指定一个缓冲区的向量以输入或输出数据;以及recvmsg和sendmsg,在其他I/O函数的所有功能基础上结合了新的接收和发送辅助数据的能力。

本章中还介绍了如何确定在套接口接收缓冲区中有多少数据,如何在套接口上使用C的标准I/O库。本章最后概要介绍一下T/TCP即事务TCP,它可以避免三路握手。

13.2 套接口超时

有三种方法给套接口上的I/O操作设置超时。

1. 调用alarm,在到达指定时间时产生SIGALRM信号。这涉及到信号处理,这一点随不同实现而变化,而且可能与进程中其他已有的alarm调用冲突。
2. 使用select阻塞在等待I/O上,select内部有一个时间限制,以此代替在read或write调用上阻塞。
3. 使用新的SO_RCVTIMEO和SO_SNDTIMEO套接口选项。这种方法存在一个问题,即并不是所有的实现都能支持这两个套接口选项。

这三种技术都可以用于输入和输出操作(例如,read、write和其他的一些变种如recvfrom和sendto),但是我们仍想要一种可用于connect的技术,因为TCP的connect的超时时间很长(典型情况为75秒)。只有在套接口为非阻塞方式时(在15.3节中将会介绍),才可以用select为connect设置超时时间,而且那两个新的套接口选项对connect不起作用。值得注意的是前两种技术可以用于任何描述字,而第三种技术只能用于套接口描述字。

我们下面举例说明所有这三种技术。

用SIGALRM给connect设置超时

图13.1给出了函数connect_timeo的代码,它以调用者设定的时间上限来调用connect。前三个参数是connect所需的,第四个参数是等待的秒数。

```
1 #include    "unp.h"
2 static void connect_alarm(int);
3 int
```

```
4 connect_timeo(int sockfd, const SA *saptr, socklen_t salen, int nsec)
5 {
6     Sigfunc       *sigfunc;
7     int           n;
8     sigfunc = Signal(SIGALRM, connect_alarm);
9     if (alarm(nsec) != 0)
10        err_msg("connect_timeo: alarm was already set");
11    if ( (n = connect(sockfd, (struct sockaddr *) saptr, salen)) < 0) {
12        close(sockfd);
13        if (errno == EINTR)
14            errno = ETIMEDOUT;
15    }
16    alarm(0);                    /* turn off the alarm */
17    Signal(SIGALRM, sigfunc);    /* restore previous signal handler */
18    return(n);
19 }
20 static void
21 connect_alarm(int signo)
22 {
23    return;                      /* just interrupt the connect() */
24 }
```

图 13.1　带超时的 connect[lib/connect_timeo.c]

建立信号处理程序

第 8 行　为 SIGALRM 建立一个信号处理程序。现有的信号处理程序(如果有的话)被保存,因此它可以在这个函数结束时恢复。

设置报警(时钟)

第 9～10 行　进程的报警时钟设成调用者指定的秒数。alarm 的返回值是进程的报警时钟(如果进程设置过一个报警时钟)现在剩下的秒数或 0(如果进程当前没有设置报警)。对于前一种情况显示一个警告信息,因为在设置新的报警时钟时去掉了前面设置的报警时钟。

调用 connect

第 11～15 行　调用 connect,如果被中断(EINTR),则将 errno 的值设为 ETIMEOUT。关闭该套接口以防止继续进行三路握手。

关闭 alarm 并恢复原来的信号处理程序

第 16～18 行　将 alarm 的参数设成 0 以关闭 alarm,并恢复原来的信号处理程序(如果有的话)。

处理 SIGALRM

第 20～24 行　信号处理程序只是简单地返回,但它将中断挂起的 connect,使 connect 返回 EINTR 错误。回想我们的 signal 函数(图 5.6)在捕获信号为 SIGALRM 时没有设置 SA_RESTART 标志。

这个例子中值得注意的一点是我们可以用这种技术减少 connect 的超时时间,但不能延长内核中已有的超时时间。也就是说,在源自 Berkeley 的内核中 connect 的超时时间一般为 75 秒。我们可以在上面的函数中设定一个比它小的超时值,比方说 10,但不能设定一个

比它大的值，譬如80，这样的话connect仍会在75秒后超时。

另外一点是我们使用系统调用(connect)的可中断性在内核超时之前返回。这在我们执行该系统调用并能处理返回的EINTR错误时是没有问题的。但在26.6节中我们会遇到一个执行系统调用的库函数，而且这个库函数在系统调用返回EINTR时重新发出系统调用。在这种情况下仍能使用SIGALRM，但在图26.10中可以看到，我们还必须使用sigsetjmp和siglongjmp以绕过函数库对EINTR的忽略。

用SIGALRM为recvfrom设置超时

图13.2是图8.8中的dg_cli函数的一个改写版本，如果在5秒内没有收到应答，recvfrom将被alarm中断。

处理recvfrom的超时

第8～22行 为SIGALRM建立一个信号处理程序，然后在每次recvfrom前调用alarm设置5秒的超时。如果recvfrom被信号处理程序中断，就输出一条信息并继续运行。如果从服务器读到一行，就关闭alarm，并输出返回的应答。

```
1 #include    "unp.h"
2 static void sig_alrm(int);
3 void
4 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
5 {
6     int    n;
7     char   sendline[MAXLINE], recvline[MAXLINE + 1];
8     Signal(SIGALRM, sig_alrm);
9     while (Fgets(sendline, MAXLINE, fp) != NULL) {
10        Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
11        alarm(5);
12        if ( (n = recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL)) < 0) {
13            if (errno == EINTR)
14                fprintf(stderr, "socket timeout\n");
15            else
16                err_sys("recvfrom error");
17        } else {
18            alarm(0);
19            recvline[n] = 0;   /* null terminate */
20            Fputs(recvline, stdout);
21        }
22    }
23 }
24 static void
25 sig_alrm(int signo)
26 {
27    return;                    /* just interrupt the recvfrom() */
28 }
```

图13.2 用alarm使recvfrom超时的dg_cli函数[advio/dgclitimeo3.c]

SIGALRM信号处理程序

第24～28行 信号处理程序只是简单地返回，以中断被阻塞的recvfrom。

因为每次建立alarm时只读一个应答，所以这个例子能正确工作。在18.4节中使用了

相同的技术，但由于用一个 alarm 读多个应答，因此必须处理存在的竞争。

用 select 为 recvfrom 设置超时

在图 13.3 中给出了第二种设置超时的技术(使用 select)的例子。其中有一个名为 readable_timeo 的函数，以指定的秒数等待一个描述字变成可读。

```
1 #include    "unp.h"
2 int
3 readable_timeo(int fd, int sec)
4 {
5     fd_set     rset;
6     struct timeval   tv;
7     FD_ZERO(&rset);
8     FD_SET(fd, &rset);
9     tv.tv_sec = sec;
10    tv.tv_usec = 0;
11    return(select(fd+1, &rset, NULL, NULL, &tv));
12          /* > 0 if descriptor is readable */
13 }
```

图 13.3　readable_timeo 函数：等待一个描述字变成可读[lib/readable_timeo.c]

给 select 准备参数

第 7～10 行　在读描述字集中打开要读的描述字所对应的位。timeval 结构设成调用者要等待的秒数。

阻塞在 select 上

第 11～12 行　select 等待描述字变成可读，或者超时。这个函数的返回值就是 select 的返回值：出错为－1，超时为 0，大于 0 则是已就绪的描述字的数目。

这个函数不执行读操作；它只是等待描述字变成可读。因此这个函数可以用在任何类型的套接口上：TCP 或 UDP。

建立类似的名为 writeable_timeo 的函数，等待一个描述字变成可写是很简单的。

在图 13.4 中将使用这个函数重写图 8.8 中的 dg_cli 函数。这个新版本只在 readable_timeo 返回一个大于零的值时才调用 recvfrom。

直到 readable_timeo 告诉我们描述字可读后我们才调用 recvfrom。这保证了 recvfrom 不会阻塞。

```
1 #include    "unp.h"
2 void
3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
4 {
5     int    n;
6     char   sendline[MAXLINE], recvline[MAXLINE + 1];
7     while (Fgets(sendline, MAXLINE, fp) != NULL) {
8         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
9         if (Readable_timeo(sockfd, 5) == 0) {
10            fprintf(stderr, "socket timeout\n");
11        } else {
12            n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
```

```
13              recvline[n] = 0;   /* null terminate */
14              Fputs(recvline, stdout);
15          }
16      }
17}
```

图 13.4 调用 readable_timeo 以设置超时的 dg_cli 函数[advio/dgclitimeo1.c]

用 SO_RCVTIMEO 套接口选项为 recvfrom 设置超时

最后一个例子是说明 SO_RCVTIMEO 套接口选项的用法。一旦为每个描述字设置了这个选项，并指定了超时值，那么这个超时对该描述字上的所有读操作都起作用。这种方法的好处是只需要设置一次选项，而前两种方法需要在每次操作前都设一遍超时。但这个套接口选项只对读操作起作用，与此类似 SO_SNDTIMEO 只对写操作起作用，它们都不能为 connect 设置超时。

图 13.5 是使用了 SO_RCVTIMEO 套接口选项的 dg_cli 函数的另一个版本。

设置套接口选项

第 8～10 行 setsockopt 的第四个参数是一个指向 timeval 结构的一个指针，其中填入超时时间。

测试超时

第 15～17 行 如果 I/O 操作超时，函数(这里是 recvfrom)会返回 EWOULDBLOCK。

13.3 recv 和 send 函数

这两个函数和标准的 read 和 write 函数都很类似，只是多了一个附加的参数。

```
#include <sys/socket.h>

ssize_t recv(int sockfd, void *buff, size_t nbytes, int flags);

ssize_t send(int sockfd, const void *buff, size_t nbytes, int flags);

                                返回：成功返回读入或写出的字节数，出错返回-1
```

```
1 #include    "unp.h"
2 void
3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
4 {
5     int          n;
6     char         sendline[MAXLINE], recvline[MAXLINE + 1];
7     struct timeval   tv;
8     tv.tv_sec = 5;
9     tv.tv_usec = 0;
10    Setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv));
11    while (Fgets(sendline, MAXLINE, fp) != NULL) {
12        Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
13        n = recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
```

```
14          if (n < 0) {
15              if (errno == EWOULDBLOCK) {
16                  fprintf(stderr, "socket timeout\n");
17                  continue;
18              } else
19                  err_sys("recvfrom error");
20          }
21          recvline[n] = 0;    /* null terminate */
22          Fputs(recvline, stdout);
23      }
24  }
```

图 13.5　用 SO_RCVTIMEO 套接口选项设置超时的 dg_cli 函数[advio/dgclitimeo2.c]

recv 和 send 的前三个参数与 read 和 write 相同。参数 flags 的值或为 0，或由在图 13.6 中列出的一个或多个常值的逻辑或构成。

flags	描述	recv	send
MSG_DONTROUTE	不查路由表		•
MSG_DONTWAIT	本操作不阻塞	•	•
MSG_OOB	发送或接收带外数据	•	•
MSG_PEEK	查看外来的消息	•	
MSG_WAITALL	等待所有数据	•	

图 13.6　I/O 函数的 flags

MSG_DONTROUTE　这个标志告诉内核目的主机在直接连接的本地网络上，不要查路由表。这是对提供这种特性的 SO_DONTROUTE 套接口选项(7.5节)的补充。用 MSG_DONTROUTE 标志可以对单个输出操作提供这种特性，而套接口选项则针对某个套接口上的所有输出操作。

MSG_DONTWAIT　这个标志将单个 I/O 操作设为非阻塞方式，而不需要在套接口上打开非阻塞标志，执行 I/O 操作，然后关闭非阻塞标志。在第 15 章我们将介绍非阻塞 I/O 以及在一个套接口上对所有 I/O 操作开关非阻塞标志。

这个标志是 Net/3 中新增加的，不一定在所有系统中都支持。

MSG_OOB　用 send 时，这个标志指明发送的是带外数据。如在第 21 章中介绍的那样，TCP 连接上只能发送 1 个字节的带外数据。用 recv 时，这个标志指明要读的是带外数据而不是一般数据。

MSG_PEEK　这个标志可以让我们查看可读的数据，在 recv 或 recvfrom 后系统不会将这些数据丢弃。在 13.7 节中会更详细地谈这个问题。

MSG_WAITALL 这个标志由4.3BSD Reno引入。它告诉内核在没有读到请求的字节数之前不使读操作返回。如果系统支持这个标志，我们就可以去掉readn函数(图3.14)而用下面的宏来代替：

```
#define readn(fd, ptr, n)  recv(fd, ptr, n, MSG_WAITALL)
```

即使设定了MSG_WAITALL，如果发生下列情况：(a)捕获一个信号，(b)连接被终止，或(c)在套接口上发生错误，这个函数返回的字节数仍会比请求的少。

除了TCP/IP，还有一些用于其他协议的附加标志。例如，OSI的传输层是基于记录的(不像TCP那样基于字节流)，在输出操作时支持MSG_EOR标志指明逻辑记录的结束。T/TCP(事务TCP，在13.9节中介绍)支持一个新的MSG_EOF标志，把一个输出操作和FIN的发送结合起来。

flags参数在设计上存在一个基本问题：它是按值传递的；而不是值-结果参数。因此它只能从进程向内核传递标志。内核不能向进程传回标志。这在用TCP/IP时不构成问题，因为TCP/IP基本不需要从内核向进程传回标志。但当在4.3BSD Reno中加入了OSI协议后，引起了在输入操作时向进程返回MSG_EOR的需要。4.3BSD Reno做出的决定是保持常用的输入函数(recv和recvfrom)的参数不变，改变recvmsg和sendmsg的msghdr结构。在13.5节中将看到在此结构中加入了一个整型的msg_flags成员，因为这个结构是用引用来传递的，所以内核可以在返回时修改这些标志。这也意味着，如果进程需要让内核来更新标志，就必须调用recvmsg代替recv或recvfrom。

13.4 readv和writev函数

这两个函数与read和write相似，但readv和writev可以让我们在一个函数调用中读或写多个缓冲区，这些操作被称为分散读(因为输入数据分散到多个应用缓冲区中)和集中写(因为多个缓冲区被集中到一次写操作中)。

```
#include <sys/uio.h>
ssize_t readv(int filedes,const struct iovec *iov,int iovcnt);
ssize_t writev(int filedes,const struct iovec *iov,int iovcnt);
                                        返回：读到或写出的字节数，出错时为-1
```

两个函数的第二个参数都是一个指向iovec结构的数组的指针，在<sys/uio.h>头文件中定义：

```
struct iovec {
    void *iov_base;     /* starting address of buffer */
    size_t iov_len;     /* size of buffer */
};
```

readv和writev函数在Posix中还没有标准化。但recvmsg和sendmsg函数

(13.5 节)中也使用了 iovec 结构,而且这个结构在 Posix.1g 中已被标准化。上面给出的 iovec 结构中的各个成员的数据类型就是在 Posix.1g 中说明的。你可能在一些实现中遇到将 iovec_base 定义为 char *,而把 iov_len 定义为 int 的情况。

在具体的实现中对 iovec 结构数组的元素个数会有一些限制。举例来说,4.3BSD 最多允许 1024 个,而 Solaris 2.5 的上限是 16。Posix.1g 要求在<sys/uio.h>头文件中定义一个常值 IOV_MAX,而且它的值不小于 16。

readv 和 writev 函数可用于任何描述字,不仅限于套接口描述字。而且 writev 是一个原子操作。对一个基于记录的协议如 UDP,调用一次 writev 只产生单个 UDP 数据报。

在 7.9 节中我们提到将 writev 和 TCP_NODELAY 套接口选项一起使用。其中谈到 write 4 字节,接着 write 396 字节会引发 Nagle 算法,对这两个缓冲区调用 writev 是一种更好的解决办法。

13.5　recvmsg 和 sendmsg 函数

这两个函数是最通用的 I/O 函数。实际上,可以用 recvmsg 代替 read、readv、recv 和 recvfrom。同样,各种输出函数都可以用 sendmsg 取代。

```
#include <sys/socket.h>

ssize_t recvmsg(int sockfd,struct msghdr *msg,int flags);

ssize_t sendmsg(int sockfd,struct msghdr *msg,int flags);

                                    返回:成功时为读入或写出的字节数,出错时为-1
```

这两个函数把大部分参数包装到一个 msghdr 结构中:

```
struct msghdr {
  void         *msg_name;        /* protocol address */
  socklen_t     msg_namelen;     /* size of protocol address */
  struct iovec *msg_iov;         /* scatter/gather array */
  size_t        msg_iovlen;      /* # elements in msg_iov */
  void         *msg_control      /* ancillary data; must be aligned for a cmsghdr
                                    structure */
  socklen_t     msg_controllen;  /* length of ancillary data */
  int           msg_flags;       /* flags returned by recvmsg() */
};
```

上面给出的 msghdr 结构源自 4.3BSD Reno,也是 Posix.1g 中所说明的。一些系统(Solaris 2.5)仍使用一种源自 4.2BSD 的老的 msghdr 结构。这种老的结构中没有 msg_flags 成员,而且 msg_control 和 msg_controllen 成员分别被叫做 msg_accrights 和 msg_accrightslen。老系统中支持的唯一一种辅助数据形式是文件描述字(称为访问权限)的传递。4.3BSD Reno 在加入 OSI 协议时增加了更多形式的辅助数据,因此使这个结构中的成员名更为通用化了。

msg_name 和 msg_namelen 成员用于未经连接的套接口(譬如未连接 UDP 套接口)。它们与 recvfrom 和 sendto 的第五和第六个参数类似:msg_name 指向一个套接口地址结构,调用者在其中存放对 sendmsg 来说是目的方的协议地址,对 recvmsg 来说是发送方的协议地址。如果不需要指明协议地址(譬如 TCP 套接口或已连接 UDP 套接口),msg_name 应被设成空指针。msg_namelen 对 sendmsg 是一个值,而对 recvmsg 是一个值-结果参数。

msg_iov 和 msg_iovlen 成员指明输入或输出的缓冲区数组(iovec 结构的数组),这与 readv 或 writev 的第二个和第三个参数相似。

msg_control 和 msg_controllen 成员指明可选的辅助数据的位置和大小,msg_controllen 对 recvmsg 是一个值-结果参数,我们会在 13.6 节中介绍辅助数据。

使用 recvmsg 和 sendmsg 时我们必须区别两个标志变量:传值的 flags 参数和 msghdr 结构中的 msg_flags 成员,它是以引用方式传递的(因为给函数传递的是 msghdr 结构的地址)。

- msg_flags 只用于 recvmsg。调用 recvmsg 时,flags 参数被拷贝到 msg_flags 成员(TCPv2 第 502 页),而且内核用这个值进行接收处理,接着它的值会根据 recvmsg 的结果而更新。
- sendmsg 会忽略 msg_flags 成员,因为它在进行输出处理时使用 flags 参数。这意味着如果我们在调用 sendmsg 时要设置 MSG_DONTWAIT 标志,应将 flags 设成该值;设置 msg_flags 为该值没用。

图 13.7 总结了使用输入和输出函数时内核检查的标志,以及 recvmsg 可能返回的 msg_flags。因为前面已经提到 msg_flags 对 sendmsg 没用,所以图中 sendmsg msg_flags 没有对应的栏。

标志	在 send flags sendto flags sendmsg flags 中检查	在 recv flags recvfrom flags recvmsg flags 中检查	在 recvmsg msg_flags 中 返回
MSG_DONTROUTE	•		
MSG_DONTWAIT	•	•	
MSG_PEEK		•	
MSG_WAITALL		•	
MSG_EOR	•		•
MSG_OOB	•	•	•
MSG_BCAST			•
MSG_MCAST			•
MSG_TRUNC			•
MSG_CTRUNC			•

图 13.7 各种 I/O 函数输入和输出标志的总结

前四个标志只检查不返回,下两个既检查又返回,最后四个只返回。下面是对 recvmsg 返回的六个标志的解释:

MSG_BCAST	这个标志是BSD/OS中新出现的，当收到的数据报是一个链路层的广播或其目的IP地址为广播地址时，将返回此标志。与IP_RECVDSTADDR套接口选项相比，该标志是判断UDP数据报是否发往广播地址的一种更好的方法。
MSG_MCAST	这个标志是BSD/OS中新出现的，当收到的数据报是链路层的多播时，将返回该标志。
MSG_TRUNC	这个标志在数据报被截断时返回：内核有比进程所分配的空间(所有iov_len成员的总和)所能容纳的还要多的数据待返回。我们会在20.3节中进一步讨论这个问题。
MSG_CTRUNC	这个标志在辅助数据被截断时返回：内核有比进程所分配的空间(msg_controllen)所能容纳的还要多的辅助数据待返回。我们在20.3节中会详细讨论。
MSG_EOR	如果返回的数据不是一个逻辑记录的结尾，该标志将清位，反之则置位。TCP不使用这个标志，因为它是一种字节流协议。
MSG_OOB	这个标志不是为TCP的带外数据返回的。它用于其他协议族(譬如OSI协议等)。

实现可能会在msg_flags成员中返回一些输入flags的标志，因此我们应该只检查那些感兴趣的标志的值(譬如图13.7中的后六个)。

图13.8给出了一个msghdr结构和它指向的各种信息。图中假定进程是对一个UDP套接口调用recvmsg。

给协议地址分配了16个字节，辅助数据分配了20个字节。初始化一个有三个iovec结构的数组：第一个指定一个100字节的缓冲区，第二个是60字节的缓冲区，第三个是80字节的缓冲区。我们还假定该套接口已设置了IP_RECVDSTADDR套接口选项，以接收UDP数据报中的目的IP地址。

假设有一个从198.69.10.2的端口2000到来的170字节的UDP数据报，目的地是我们的UDP套接口，目的IP地址为206.62.226.35。图13.9展示了当recvmsg返回时msghdr结构中的所有信息。

图中阴影的字段是被recvmsg修改的。从图13.8到13.9有以下改变：

- msg_name指向的缓冲区已经填入了一个含有收到的数据报的源IP地址和源UDP端口的网际套接口地址结构。
- 值-结果参数msg_namelen的值更新为msg_name中存放的数据的总量。因为调用前它的值是16，而recvmsg返回的还是16，所以它的值没有改变。
- 数据的前100个字节存放在第一个缓冲区，后续的60字节存放在第二个缓冲区，最后10字节存放在第三个缓冲区。最后的那个缓冲区中的后70个字节没有改变。recvmsg函数的返回值是该数据报的大小即170。
- msg_control指向的缓冲区被填入了一个cmsghdr结构。(在13.6节中对辅助数据有更多的说明，IP_RECVDSTADDR这个套接口选项在20.2节中有详细介绍。)cmsg_len为16，cmsg_level为IPPROTO_IP，cmsg_type为IP_RECVDSTADDR，

下四个字节是收到的UDP数据报中的目的IP地址。这个20字节缓冲区中用来存放辅助数据的最后4个字节没有被修改。

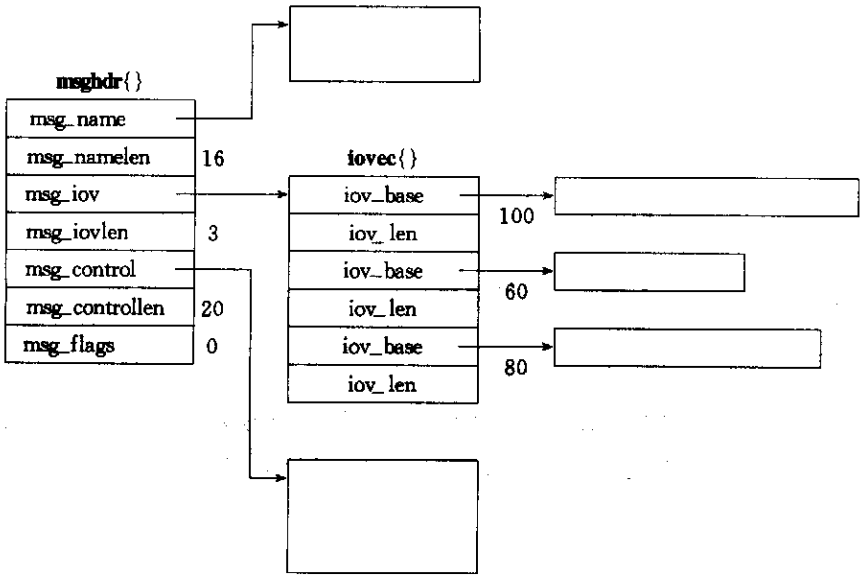


图13.8 对一个UDP套接口调用recvmsg时的数据结构

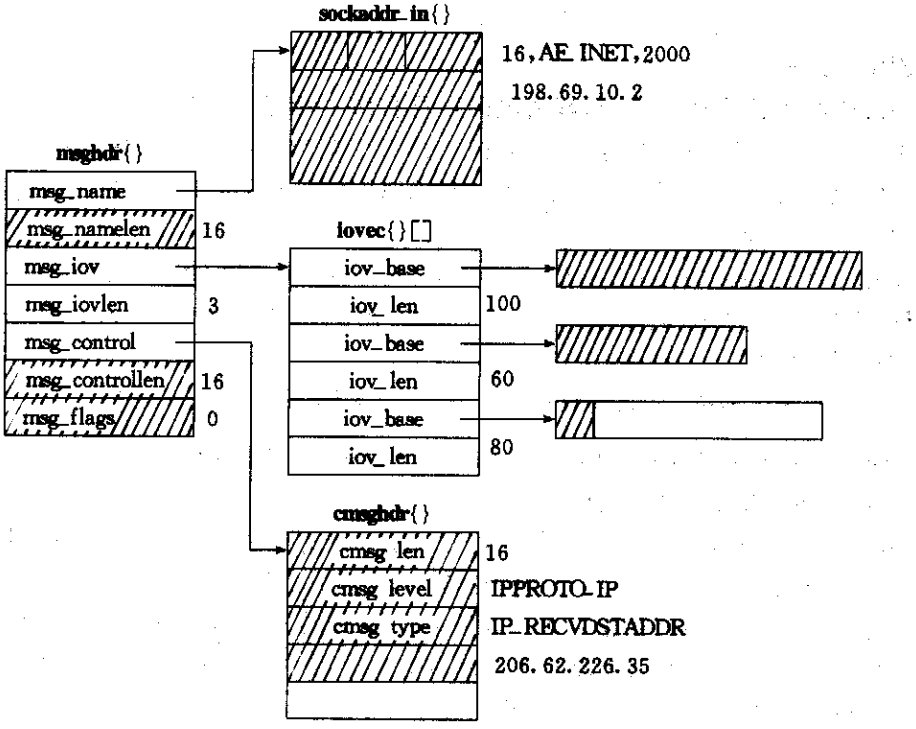


图13.9 recvmsg返回时对图13.8的更新

- msg_controllen成员更新为存放的辅助数据的实际数量，它也是值-结果参数，返回时它的值为16。

• msg_flags 成员被 recvmsg 更新，但是没有给进程返回任何标志。

图 13.10 小结了我们已述的五组 I/O 函数的差异。

函数	任何描述字	只对套接口描述字	单个 read/write 缓冲区	分散/集中 read/write	可选标志	可选对方地址选项	可选控制信息
read,write	•		•				
readv,writev	•			•			
recv,send		•	•		•		
recvfrom,sendto		•	•		•	•	
recvmsg,sendmsg		•		•	•	•	•

图 13.10　五组 I/O 函数的比较

13.6　辅助数据

可以在 sendmsg 和 recvmsg 时使用 msghdr 结构中的 msg_control 和 msg_controllen 成员发送和接收辅助数据(ancillary data)。辅助数据的另一种叫法是控制信息(control information)。本节中介绍这个概念并给出用来建立和处理辅助数据的结构和宏，介绍辅助数据实际使用的源代码在以后章节中再给出。

图 13.11 是对本文中辅助数据的各种用法的一个总结。

协议	cmsg_level	cmsg_type	说明
IPv4	IPPROTO_IP	IP_ RECVDSTADDR	接收 UDP 数据报的目的地址
		IP_RECVIF	接收 UDP 数据报的接口索引
IPv6	IPPROTO_IPV6	IPV6_DSTOPTS	指定/接收目标选项
		IPV6_HOPLIMIT	指定/接收跳限
		IPV6_HOPOPTS	指定/接收步跳选项
		IPV6_NEXTHOP	指定下一跳地址
		IPV6_PKTINFO	指定/接收分组信息
		IPV6_RTHDR	指定/接收路由头部
Unix 域	SOL_SOCKET	SCM_RIGHTS	发送/接收描述字
		SCM_CREDS	发送/接收用户凭证

图 13.11　辅助数据用法的总结

OSI 协议族也有辅助数据用于各种目的，这里我们就不做讨论了。

辅助数据由一个或多个辅助数据对象组成，每个对象由一个 cmsghdr 结构开头，该结构在<sys/socket.h>文件中定义如下：

```
struct cmsghdr {
  socklen_t     cmsg_len;     /* length in bytes, including this structure */
  int           cmsg_level;   /* originating protocol */
  int           cmsg_type;    /* protocol-specific type */
```

```
    /* followed by unsigned char cmsg_data[] */
};
```

我们已经在图13.9中见到过这个结构，当时是在设置IP_RECVDSTADDR套接口选项下用于返回接收到的UDP数据报的目的IP地址。msg_control指向的辅助数据必须按csmghdr结构进行对齐。在图14.11中展示了一种对齐方法。

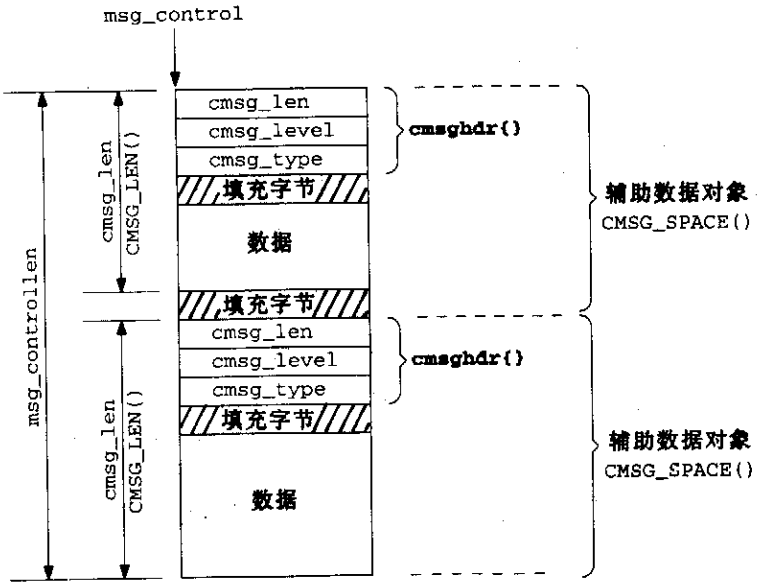


图13.12 包含两个辅助数据对象的辅助数据

图13.12展示了在一个控制缓冲区中的两个辅助数据对象的例子。msg_control指向第一个辅助数据对象，辅助数据的总长度由msg_controllen指定。每个对象开头是一个描述该对象的cmsghdr结构。在cmsg_type成员和实际的数据之间可能有填充字节，在数据之后、下一个对象之前也可能有填充字节。下面简要介绍的五个宏解决这种可能的填充问题。

不是所有的实现都支持在一个控制缓冲区中有多个辅助数据对象。

图13.13展示了当使用Unix域套接口传递描述字(14.7节)或传递凭证(14.8节)时，cmsghdr结构的格式。

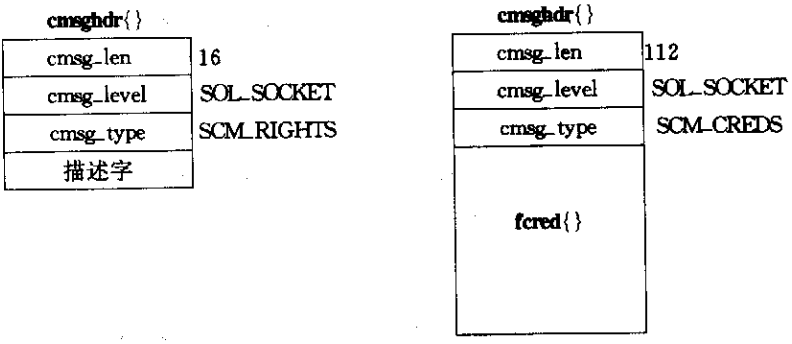


图13.13 使用Unix域套接口时的cmsghdr结构

在这幅图中，我们假定 cmsghdr 结构中的三个成员每个都占 4 字节，在 cmsghdr 结构和实际数据之间没有填充字节。传递描述字时 cmsg_data 数组的内容是实际的描述字的值。在上图中只展示了一个文件描述字的传递，但一般能传递多个描述字（这种情况下 cmsg_len 的值为 12 加上 4 乘描述字的数目，这里假定每个描述字占 4 个字节）。

因为由 recvmsg 返回的辅助数据可以包含任意数目的辅助数据对象，为了对应用程序屏蔽可能出现的填充字节，在<sys/socket.h>中定义了以下五个宏，以简化对辅助数据的处理。

```
#include <sys/socket.h>
#include <sys/param.h>      /* for ALIGN macro on many implementations */

struct cmsghdr *CMSG_FIRSTHDR(struct msghdr *mhdrptr);
                        返回：指向第一个 cmsghdr 结构的指针，无辅助数据时为 NULL

struct cmsghdr *CMSG_NXTHDR(struct msghdr *mhdrptr,struct cmsghdr *cmsgptr);
                返回：指向下一个 cmsghdr 结构的指针，不再有辅助数据对象时为 NULL

unsigned char *CMSG_DATA(struct cmsghdr *cmsgptr);
                        返回：指向与 cmsghdr 结构关联的数据的第一个字节的指针

unsigned int CMSG_LEN(unsigned int length);
                                    返回：给定数据量下存储在 cmsg_len 中的值

unsigned int CMSG_SPACE(unsigned int length);
                                    返回：给定数据量下一个辅助数据对象的总大小
```

Posix.1g 定义了前三个宏，[Stevens and Thomas 1997]定义了后两个。

这些宏可以用在下面的伪代码中：

```
struct msghdr    msg;
struct cmsghdr   *cmsgptr;

/* fill in msg structure */
/* call recvmsg() */

for (cmsgptr = CMSG_FIRSTHDR(&msg); cmsgptr != NULL;
     cmsgptr = CMSG_NXTHDR(&msg,cmsgptr)) {
    if (cmsgptr->cmsg_level == ... &&
        cmsgptr->cmsg_type == ...) {
        u_char  *ptr;

        ptr = CMSG_DATA(cmsgptr);
        /* process data pointed to by ptr */
    }
}
```

CSMG_FIRSTHDR 返回指向第一个辅助数据对象的指针，如果在 msghdr 结构中没有辅助数据（msg_control 为空指针，或 csmg_len 小于一个 cmsghdr 结构的大小），则返回空指

针。当在控制缓冲区中没有下一个辅助数据对象时，CSMG_NXTHDR 返回空指针。

> 许多现有的 CMSG_FIRSTHDR 的实现并不检查 msg_controllen，直接返回 cmsg_control 的值。在图 20.2 中调用这个宏之前检测了 msg_controllen 的值。

CMSG_LEN 和 CMSG_SPACE 的区别在于，前者不将填充字节计算在内，因而等于 cmsg_len 的值，但后者将这些填充字节都计算在内，因此在动态申请辅助数据对象的数据空间时使用这个值。

13.7 排队的数据量

在不读出数据的情况下，如何知道一个套接口的接收队列中有多少数据可读呢？有三种方法。

1. 如果在没有数据可读时还有其他事情要做，为了不阻塞在内核中，可以使用非阻塞 I/O。在第 15 章中将对此进行介绍。
2. 如果想检查一下数据而使数据仍留在接收队列中，可以使用 MSG_PEEK 标志（图 13.6）。如果想这样做，但又不能肯定是否有数据可读，可以把这个标志和非阻塞套接口相结合，或与 MSG_DONTWAIT 标志结合使用。

   要注意的是，对一个字节流套接口来说，接收队列中的数据量在两次连续的 recv 调用之间可能会改变。举例来说，假设在一个 TCP 套接口上通过指定一个 1024 字节的缓冲区和 MSG_PEEK 标志调用一次 recv，返回值是 100。如果再次调用 recv，返回值可能会超过 100 字节（假定指定的缓冲区长度大于 100），因为在两次调用之间可能接收到更多的数据。

   如果是一个 UDP 套接口，并且在接收队列中有一个数据报的情况下，指定 MSG_PEEK 标志调用 recvfrom，接着是一个不使用 MSG_PEEK 的 recvfrom 调用，这两个调用的返回值是完全相同的（数据报的大小、内容和发送方的地址），即使在两次调用之间有另外的数据报加入套接口接收缓冲区也不会改变。（这里有一个前提是其他进程没有共享该描述字并在此时从该套接口上读取数据。）
3. 一些实现支持 ioctl 的 FIONREAD 命令。ioctl 的第三个参数是一个指向整数的指针，在该整数中返回的值是套接口接收队列中数据的字节数（TCPv2 第 553 页）。这个值是队列中字节数的总和，对 UDP 套接口来说包括在队列中的所有数据报。还要注意的是在源自 Berkeley 的实现中，对 UDP 套接口返回的计数包括套接口地址结构要占的空间，其中含有每个数据报的发送方 IP 地址和端口号（IPv4 为 16 字节，IPv6 为 24 字节）。

13.8 套接口和标准 I/O

目前为止的所有例子中使用的 read 及 write 函数和它们的一些变种（recv、send 等），称为 Unix I/O。这些函数使用描述字工作，通常是作为 Unix 内核中的系统调用来实现的。

进行输入输出的另一种方法是标准 I/O 函数库（standard I/O Library）。在 ANSI C 的标

准中对此进行了说明，其目的是方便移植到支持 ANSI C 的非 Unix 系统上。标准 I/O 函数库处理一些我们在使用 Unix I/O 函数时必须担心的细节问题，譬如自动缓冲输入和输出流。不幸的是，它对流的缓冲处理会产生新的问题。APUE 的第 5 章涵盖了标准 I/O 库的细节，[Plauger 1992]介绍和讨论了标准 I/O 库的一个完整的实现。

> 在标准 I/O 库中使用了流这个术语，譬如“打开一个输入流”或“刷新输出流”。不要将此与第 33 章中讨论的系统 V 的流子系统相混淆。

标准 I/O 库可以用于套接口，但有几点要考虑。

- 对任何描述字调用 fdopen 函数都可以生成一个标准 I/O 流。同样，对于一个标准 I/O流，用 fileno 可以得到对应的描述字。我们第一次遇到 fileno 是在图 6.9 中，当时要在一个标准 I/O 流上调用 select。select 只能用于描述字，所以必须获得这个标准 I/O 流的描述字。
- TCP 和 UDP 套接口是全双工的。标准 I/O 流也可以是全双工的：只要以 r+方式打开流即可，r+意味着读写。但是对这样的流不能在一个输出函数之后紧接一个输入函数，必须插入一个 fflush、fseek、fsetpos 或 rewind 调用。同样，不能在一个输入函数之后紧接一个输出函数，必须插入一个 fseek、fsetpos 或 rewind 调用，除非输入函数遇到一个文件结束符。后三个函数的问题是它们都调用 lseek，可 lseek 在套接口上会失败。
- 解决这个读写问题的最简单的方法是，对一个套接口打开两个标准 I/O 流：一个读，一个写。

例子：使用标准 I/O 的 str_echo 函数

下面我们用标准 I/O 代替 readline 和 writen 重新编写 TCP 回射服务器(图 5.3)。图 13.14是使用标准 I/O 的 str_echo 函数的新版本。(这个版本存在一个问题，我们马上会谈到。)

```
1 #include   "unp.h"
2 void
3 str_echo(int sockfd)
4 {
5     char     line[MAXLINE];
6     FILE     *fpin, *fpout;
7     fpin = Fdopen(sockfd, "r");
8     fpout = Fdopen(sockfd, "w");
9     for ( ; ; ) {
10        if (Fgets(line, MAXLINE, fpin) == NULL)
11            return;          /* connection closed by other end */
12        Fputs(line, fpout);
13    }
14 }
```

图 13.14　使用标准 I/O 重新编码的 str_echo 函数[advio/str_echo_stdio02.c]

将描述字转换成输入流和输出流

第7～13行 用fpopen创建两个标准I/O流：一个用于输入，一个用于输出。用fgets和fputs代替readline和writen。

如果服务器使用这个版本的str_echo，那么在运行客户程序时，我们会看到以下情况：

```
solaris % tcpcli02 206.62.226.33
hello, world          键入这一行，但没有回射
and hi                再键入一行，仍无回射
hello??               再键入一行，还是没有回射
^D                    键入文件结束符
hello, world          然后才把回射的三行输出
and hi
hello??
```

这里有一个缓冲问题，因为在键入文件结束符之前服务器没有任何回射。实际发生了以下事情：

- 我们键入第一行，将其发往服务器。
- 服务器用fgets读入这一行，再用fputs回射这一行。
- 但服务器的标准I/O流被标准I/O库完全缓冲。这意味着该库将回射的行拷贝到输出流的标准I/O缓冲区，而没有将缓冲区中的内容写到描述字，因为缓冲区还没有满。
- 我们键入第二行，将其发往服务器。
- 服务器用fgets读入这一行，再用fputs回射这一行。
- 服务器的标准I/O库再次将回射的行拷贝到输出流的标准I/O缓冲区，而没有将缓冲区中的内容写到描述字，因为缓冲区还没有满。
- 我们输入第三行时发生了同样的情况。
- 我们键入文件结束符，str_cli函数(图6.13)调用shutdown，向服务器发送一个FIN。
- 服务器收到这个TCP FIN，这使fgets返回一个空指针。
- str_echo函数返回到服务器的main函数(图5.12)，子进程调用exit终止。
- C的库函数exit调用标准I/O清除函数(APUE第162～164页)，由fputs装入部分内容的输出缓冲区中的数据被输出。
- 服务器的子进程终止，使它的已连接套接口被关闭，发一个FIN给客户，完成TCP的四分组终止序列。
- str_cli函数收到回射的三行并输出。
- str_cli接着在该套接口收到一个文件结束符，客户程序终止。

问题是服务器上的标准I/O库自动进行缓冲。标准I/O库执行三种缓冲。

1. 完全缓冲意味着只有在以下情况时才进行I/O：缓冲区满，进程明确地调用fflush或进程调用exit终止。标准I/O缓冲区大小通常为8192字节。
2. 行缓冲意味着在以下情况时进行I/O：遇到一个换行符，进程调用fflush或进程调用exit终止。
3. 不缓冲意味着每次调用标准I/O输出函数时都进行I/O。

大多数 Unix 中标准 I/O 库的实现遵循了以下规则：

- 标准错误输出总是不缓冲。
- 标准输入和标准输出是全缓冲的。除非它们是一个终端设备，那样的话它们是行缓冲的。
- 其他的流都是全缓冲的，除非它们是一个终端设备，那样的话它们是行缓冲的。

既然套接口不是终端设备，图 13.14 中的 str_echo 函数的问题就在于输出流(fpout)是全缓冲的。有两种解决方法：调用 setvbuf 将输出流强制成行缓冲的，或在每次 fputs 之后调用 fflush 强制输出回射行。以上的任意一种方法都可以使 str_echo 函数正常运行。

另一种解决办法是完全避免使用标准 I/O 库，而用 sfio 库。这在[Korn and Vo 1991]中进行了介绍，而且其源代码是公开的。

要注意的是一些标准 I/O 库的实现对大于 255 的描述字还有问题。这对处理多个描述字的网络服务器是一个问题。检查你的<stdio.h>头文件中 FILE 结构的定义，看看存放描述字的变量的类型。

13.9　T/TCP：事务 TCP

T/TCP 是在 TCP 的基础上作了少量的修改，以避免在最近通信过的主机之间进行三路握手。T/TCP 的细节在 TCPv3、RFC 1379[Braden 1992b]和 RFC 1644[Braden 1994]中做了说明。

T/TCP 最广为流传的实现是在 FreeBSD 中。

T/TCP 能将 SYN、FIN 和数据组合到单个分节中，前提是数据的长度小于 MSS。图 13.15 中对这种情况进行了说明。第一个分节是由一个 sendto 调用生成的客户的 SYN、FIN 和数据。它把 connect、write 和 shutdown 的功能组合起来了。服务器按照通常的步骤是调用 socket、bind、listen 和 accept，后者在客户的分节到来时返回。服务器用 send 发回应答并关闭套接口。这使服务器向客户发出 SYN、FIN 和应答。如果和图 2.5 相比较，我们可以看到不仅在网络上传输的分节要少(T/TCP 是 3 个，TCP 是 10 个，UDP 是 2 个)，而且客户初始化连接、发送请求和接收应答所花费的时间也减少了一个 RTT。

T/TCP 的好处是保持了 TCP 的所有可靠性(序列号、超时、重传，等等)，而不像 UDP 那样把可靠性推给应用程序去实现。T/TCP 还维持了 TCP 的慢启动和拥塞避免措施，这些特性在 UDP 应用程序中是没有的。

我们在这儿忽略了一些细节，它们都在 TCPv3 中介绍。例如，客户首次与服务器交互时，三路握手是需要的。不过以后只要双方高速缓存的一些信息不过时，并且没有一方的主机崩溃并重启过，那么就可避免三路握手。图中给出的三个分节来自于最少的请求-应答交换。如果或者请求或者应答在一个分节中放不下，那么就需要额外的分节。术语“事务”的含义是客户的请求与服务器的应答。常见例子有 DNS 请求与服务器的应答，以及 HTTP 请求与服务器的应答。这个术语不用来指称两阶段提交协议(two-phase commit protocol)。

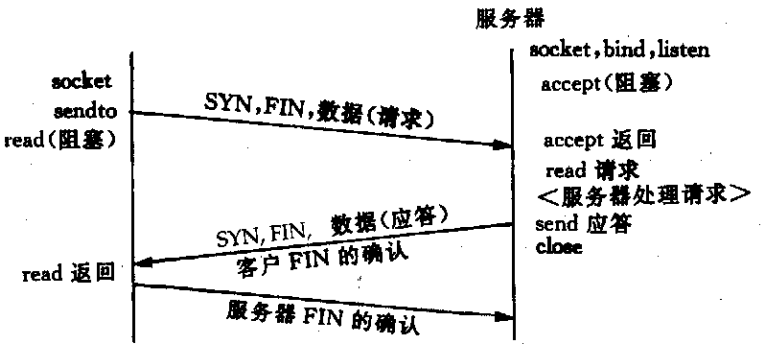


图 13.15 最小 T/TCP 事务的时间线

为处理 T/TCP,套接口 API 作了些变动。我们注意到在提供 T/TCP 的系统上,TCP 应用程序不必做任何改动,除非需要 T/TCP 提供的特性。所有已有的 TCP 应用程序继续使用已经叙述过的套接口 API 工作。

- 客户调用 sendto 以把连接的建立与数据的发送结合起来。它替代了分开的 connect 和 write 调用。服务器的协议地址现在是传递给 sendto,而不是 connect。
- 所提供的新的输出标志 MSG_EOF(图 13.6)用于指示相应套接口上不再发送数据。它允许我们把输出操作(send 或 sendto)与 shutdown 给合起来。发送含有 SYN、FIN 和数据的单个分节的方法就是给 sendto 指定这个标志以及服务器的地址。还要注意图 13.15 中服务器是使用 send 而不是 write 发送应答的,其原因是为了指定 MSG_EOF 标志以随应答一起发送 FIN(不要把这个新标志与已有的 MSG_EOR 标志混为一谈,后者指示的是面向记录协议的记录结束条件)。
- 所定义的级别为 IPPROTO_TCP 的新套接口选项 TCP_NOPUSH 可防止 TCP 就为腾空套接口发送缓冲区的目的而发送分节。当客户需要用单个 sendto 发送一个请求,并且请求可能超过 MSS 大小时,它们应该设置这个选项,因为它可以减少所发送的分节数。TCPv3 第 47～49 页详细讨论了这个新套接口选项。
- 希望跟服务器建立连接并且使用 T/TCP 发送请求的客户应该调用 socket、setsockopt(以设置 TCP_NOPUSH 选项)和 sendto(如果只发送一个请求就指定 MSG_EOF 标志)。如果 setsockopt 以错误 ENOPROTOOPT 失败返回,或者 sendto 以错误 ENOTCONN 失败返回,那么相应主机是不支持 T/TCP 的。这种情况下客户可以调用 connect 和 write,可能再跟一个 shutdown(如果只有一个请求需发送的话)。
- 服务器所需的唯一变动是,如果它想随应答一起发送 FIN 的话,它应调用 send 并指定 MSG_EOF 标志来发送应答,而不是调用 write。
- T/TCP 的编译时测试可以使用伪代码 #ifdef MSG_EOF。

TCPv3 的附录 B 有 T/TCP 客户和服务器程序的例子。

13.10　小　结

在套接口操作上设置时间限制的方法有三种：

- 使用 alarm 函数和 SIGALRM 信号
- 使用由 select 提供的时间限制
- 使用更新的 SO_RCVTIMEO 和 SO_SNDTIMEO 套接口选项

第一种方法容易使用，但涉及到信号处理，而它真如我们在 18.5 节将看到的那样会导致竞争状态。使用 select 意味着我们阻塞在这个函数及其所提供的时间限制上，而不是阻塞在对 read、write 或 connect 的调用上。第三种方法也容易使用，但不是所有实现都提供。

recvmsg 和 sendmsg 是所提供的五组 I/O 函数中最通用的。它们结合了这些能力：指定 MSG_xxx 标志（来自 recv 和 send），返回或指定对方的协议地址（来自 recvfrom 和 sendto），使用多个缓冲区（来自 readv 和 writev）；又增加了两个新的特性：给应用进程返回标志，接收或发送辅助数据。

我们在文中叙述了 10 种不同格式的辅助数据，其中 6 种是随 IPv6 新定义的。辅助数据由一个或多个辅助数据对象构成，每个对象都以一个 cmsghdr 结构打头，它指定数据的长度、协议级别及类型。5 个以 CMSG_打头的函数是用来构建和分析辅助数据的。

套接口可以跟 C 标准 I/O 函数库一块使用，但这样一来给已经由 TCP 提供的缓冲能力又增加了另外一级缓冲。实际上，缺少对由标准 I/O 库执行的缓冲机制的认识是使用这个库时最常见的问题。既然套接口不是终端设备，对这种潜在问题的常用解决办法就是把标准 I/O 流设置成不缓冲。

T/TCP 是对 TCP 的一个简单的增强，能在客户和服务器最近曾通信过的情况下避免三路握手，使服务器对客户的请求更快地做出响应。从编程的角度来看，其优点在于，客户程序以 sendto 调用代替了通常的 connect、write 和 shutdown 调用序列。

13.11　习　题

13.1　在图 13.1 中当重置信号处理程序时，如果进程还没有为 SIGALRM 建立信号处理程序，将会发生什么结果？

13.2　在图 13.1 中如果进程已设置了一个 alarm 定时器，它就会输出一个警告。修改该函数使它在 connect 后、函数返回前重置 alarm。

13.3　对图 11.7 做以下修改：在调用 read 前，以 MSG_PEEK 标志调用 recv。recv 返回时，以 FIONREAD 调用 ioctl，并输出在套接口接收缓冲区中的字节数。然后再调用 read 实际读取数据。

13.4　如果进程在 main 函数的结尾异常终止而不是以 exit 退出，那些在标准 I/O 缓冲区中还没输出的数据会发生什么结果？

13.5　按图 13.14 下面介绍的两种方法修改程序，以确认它们能解决缓冲问题。

第14章 Unix域协议

14.1 概 述

Unix域协议并不是一个实际的协议族，它只是在同一台主机上进行客户-服务器通信时，使用与在不同主机上的客户和服务器间通信时相同的API(套接口或XTI)的一种方法。当客户和服务器在同一台主机上时，Unix域协议是这套系列书的第二卷将介绍的IPC通信方式的一种替代品。TCPv3的第三部分提供了Unix域套接口在源自Berkeley的内核中的具体实现。

Unix域提供了两种类型的套接口：字节流套接口(与TCP类似)和数据报套接口(与UDP类似)。尽管还提供了原始套接口，但它的语义不曾见于任何文档，笔者也没有见到任何程序使用过它，在Posix.1g中没有对它的定义。

使用Unix域套接口有三个原因：

1. 在源自Berkeley的实现中，当通信双方位于同一台主机上时，Unix域套接口的速度通常是TCP套接口的两倍(TCPv3第223～224页)。有一个应用系统利用了这个优点：X Window系统。当X11客户启动并打开到X11服务器的连接时，客户检查DISPLAY环境变量的值，该变量指明服务器的主机名、窗口和屏幕。如果服务器与客户在同一台主机上，客户就打开一个到服务器的Unix域字节流套接口连接，否则打开一个到服务器的TCP连接。
2. Unix域套接口可以用来在同一台主机上的各进程之间传递描述字。在14.7节中对此提供了一个完整的例子。
3. Unix域套接口的较新实现中可以向服务器提供客户的凭证(用户ID和组ID)，这能提供附加的安全检查。在14.8节中对此进行了介绍。

Unix域用来标识客户和服务器的协议地址是在通常的文件系统里的路径名。我们还记得IPv4的协议地址由一个32位地址和16位的端口号组成，IPv6的协议地址由128位的地址和16位的端口号组成。这些路径名不是普通的Unix文件：除非将它们和Unix域套接口关联起来，否则程序是不能读写这些文件的。

14.2 Unix域套接口地址结构

图14.1列出了在<sys/un.h>头文件中定义的Unix域套接口地址结构。

```
struct sockaddr_un {
  uint8_t       sun_len;
  sa_family_t   sun_family;        /* AF_LOCAL */
  char          sun_path[104];     /* null-terminated pathname */
};
```

图 14.1　Unix 域套接口地址结构 sockaddr_un

BSD 的早期版本定义的 sun_path 数组的大小为 108 字节，而不是在图中的 104 字节。Posix.1g 只要求它的大小至少有 100 字节。之所以有这个限制，是因为自 4.2BSD 以来的实现要求这个结构能装入一个 128 字节的 mbuf(一种内核内存缓冲块)。

sun_path 数组中存放的路径名必须是以空字符结尾的。系统提供了 SUN_LEN 宏，它以一个指向 sockaddr_un 结构的指针为参数，返回该结构的长度，长度里包括路径名中的非空字节数。未指定地址以空字符串的路径名表示，也就是一个 sun_path[0]为 0 的地址结构。这是 Unix 域中与 IPv4 的 INADDR_ANY 常值或 IPv6 的 IN6ADDR_ANY_INIT 常值等价的一个地址。

Posix.1g 将 Unix 域协议改名为“本地 IPC”，以消除对 Unix 操作系统的依赖。原来的常值 AF_UNIX 变成 AF_LOCAL。虽然如此，我们仍使用“Unix 域”这个名词，因为它已成为一个既成事实的名字，与底层的操作系统无关。而且，即使 Posix.1g 试图使它独立于操作系统，该套接口地址结构仍保留_un 后缀。

例子：Unix 域套接口的捆绑

图 14.2 中的程序建立一个 Unix 域套接口，给它捆绑一个路径名，然后调用 getsockname 输出已绑定的路径名。

```
 1 #include    "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int      sockfd;
 6     socklen_t   len;
 7     struct sockaddr_un     addr1, addr2;
 8     if (argc != 2)
 9         err_quit("usage: unixbind <pathname>");
10     sockfd = Socket(AF_LOCAL, SOCK_STREAM, 0);
11     unlink(argv[1]);          /* OK if this fails */
12     bzero(&addr1, sizeof(addr1));
13     addr1.sun_family = AF_LOCAL;
14     strncpy(addr1.sun_path, argv[1], sizeof(addr1.sun_path) - 1);
15     Bind(sockfd, (SA *) &addr1, SUN_LEN(&addr1));
16     len = sizeof(addr2);
17     Getsockname(sockfd, (SA *) &addr2, &len);
18     printf("bound name = %s, returned len = %d\n", addr2.sun_path, len);
19     exit(0);
20 }
```

图 14.2　给 Unix 域套接口捆绑路径名[unixdomain/unixbind.c]

删除路径名

第11行 给套接口bind的路径名在命令行参数中给出。但如果该路径名在文件系统中已存在,bind将失败。因此为预防路径名已存在的情况,我们先调用unlink将其删除。如果路径名不存在,unlink会返回错误,程序将其忽略。

bind然后getsockname

第12~18行 我们用strncpy拷贝命令行参数,以避免路径名过长导致结构溢出。因为已将这个结构初始化为0,并从sun_path数组的大小中减去1,所以路径名肯定是以空字符结尾的。调用bind并用SUN_LEN宏计算该函数的长度参数。接着调用getsockname取得刚才绑定的路径名并输出结果。

如果在BSD/OS下运行这个程序,会得到以下结果:

```
bsdi % umask                                    首先输出umask的值
0002                                            shell以八进制格式输出该值
bsdi % unixbind /tmp/foo.bar
bound name = /tmp/foo.bar,returned len = 14
bsdi % unixbind /tmp/foo.bar                    再运行一次
bound name = /tmp/foo.bar,returned len = 14
bsdi % ls -l /tmp/foo.bar
srwxrwxrwx  1 rstevens wheel 0 May 20 11:02 /tmp/foo.bar
bsdi % ls -lF /tmp/foo.bar
srwxrwxrwx  1 rstevens wheel 0 May 20 11:02 /tmp/foo.bar=
```

首先输出umask的值,因为Posix.1g中说生成的路径名的权限跟该值相关。2这个值关闭了其他用户的写权限位(有时叫做全球写(world-write))。接着运行该程序,可以看到getsockname返回的长度是14:sun_len占1个字节,sun_family占1个字节,路径名占12个字节(不含结尾的空字符)。这是一个值-结果参数的例子,返回的结果与调用时的值不同。我们可以用printf的%s格式输出该路径名,因为sun_path中的路径名是以空字符结尾的。然后再运行该程序,以检验调用unlink删除了路径名。

用ls -l查看文件的权限和类型。在4.4BSD上其文件类型为套接口,输出表示为s。还可以注意到所有的9个权限位都是置位的,因为4.4BSD没有用umask来修改缺省值。最后再次用-F选项运行ls,这会使4.4BSD在路径名后附加一个等号。

> Posix.2中没有关于套接口的说明,它只规定-F选项对目录输出一个斜杠,对可执行文件输出一个星号,对FIFO输出一个竖杠。

现在在Solaris 2.5上运行同样的程序。

```
solaris % umask
02
solaris % unixbind /tmp/foo.bar
bound name = /tmp/foo.bar,returned len = 110
solaris % unixbind /tmp/foo.bar
bound name = /tmp/foo.bar,returned len = 110
solaris % ls -lF /tmp/foo.bar
p---------  1 rstevens other1        0 May 20 11:36 /tmp/foo.bar|
```

第一个差别是getsockname返回的长度是110,这是Solaris中sockaddr_un结构总的大

小。因为 sun_path 成员中的路径名是以空字符结尾的，所以这是可行的。还可以注意到缺省的文件权限是 0：所有读、写和执行权限都是关闭的。因为所有权限位都关闭，所以无法分辨是否使用了 umask 的值。最后我们注意到 ls -l 指出该路径名是一个 FIFO，这是在 SVR4 中 Unix 域套接口的表现形式，-F 选项对 FIFO 会以一个竖杠输出表示。

14.3 socketpair 函数

socketpair 函数建立一对相互连接的套接口。这个函数只对 Unix 域套接口适用。

```
#include <sys/socket.h>
int socketpair(int family,int type,int protocol,int sockfd[2]);
                                        返回：成功返回 0，出错返回 -1
```

family 必须为 AF_LOCAL，protocol 必须为 0，type 可以是 SOCK_STREAM 或 SOCK_DGRAM。新创建的两个套接口描述字作为 sockfd[0]和 sockfd[1]返回。

> 这个函数和 Unix 的 pipe 函数类似：都返回两个描述字，而且两个描述字互相连接。实际上，源自 Berkeley 的实现中，pipe 使用了与 sokcetpair[TCPv3 第 253～254 页]同样的内部操作。

创建的两个套接口是没有名字的；即没有涉及隐式 bind。

以 SOCKET_STRAEM 作为 type 调用 socketpair 所得到的结果称作流管道(stream pipe)。这和一般的 Unix 管道(由 pipe 函数生成)类似，但流管道是全双工的，即两个描述字都是可读写的。在图 14.7 中展示了用 socketpair 建立的流管道。

> Posix.1 不要求全双工的管道。在 SVR4 中 pipe 返回两个全双工的描述字，但源自 Berkeley 的内核一般返回两个半双工的描述字(TCPv3 的图17.31)。

14.4 套接口函数

当用于 Unix 域套接口时，套接口函数有一些差别和限制。下面列出 Posix.1g 对此的一些要求，需注意的是并不是所有的实现中都做到了这一级。

1. bind 建立的路径名的缺省访问权限应为 0777(用户、用户组和其他用户都能读、写和执行)，并被当前的 umask 值修改。
2. 与 Unix 域套接口相关联的路径名应为一个绝对路径名，而不是相对路径名。避免使用后者的原因是它依赖于调用者的当前工作目录。这就是说，如果服务器捆绑一个相对路径名，客户必须在与服务器相同的目录下(或必须知道这个目录)调用 connect 或 sendto 才能成功。

   > Posix.1g 中说给 Unix 域套接口捆绑相对路径名将产生不可预计的结果。

3. connect 使用的路径名必须是一个绑定在某个已打开的 Unix 域套接口上的路径名，

而且套接口的类型(字节流或数据报)也必须一致。下列情况将会出错:(a)该路径名存在但不是一个套接口,(b)路径名存在且是一个套接口,但没有与该路径名相关联的打开的描述字,(c)路径名存在且是一个打开的套接口,但类型不符(也就是说,Unix 域字节流套接口不能连到与 Unix 域数据报套接口相关联的路径名,反之亦然)。

4. 用 connect 连接 Unix 域套接口时的权限检查和用 open 以只写方式访问路径名时完全相同。
5. Unix 域字节流套接口和 TCP 套接口类似:它们都为进程提供一个没有记录边界的字节流接口。
6. 如果 Unix 域字节流套接口的 connect 调用发现监听套接口的队列已满(4.5 节),会立刻返回一个 ECONNREFUSED 错误。这和 TCP 有所不同:如果监听套接口的队列已满,它将忽略到来的 SYN,TCP 连接的发起方会接着发送几次 SYN 重试。
7. Unix 域数据报套接口和 UDP 套接口类似:它们都提供一个保留记录边界的不可靠的数据报服务。
8. 与 UDP 套接口不同的是,在未绑定的 Unix 域套接口上发送数据报不会给它捆绑一个路径名。(回想在未绑定的 UDP 套接口上发送 UDP 数据报会为该套接口捆绑一个临时端口。)这意味着,数据报的发送者除非绑定一个路径名,否则接收者无法发回应答数据报。同样,与 TCP 和 UDP 不同的是,给 Unix 域数据报套接口调用 connect 不会捆绑一个路径名。

14.5 Unix 域字节流客户-服务器程序

下面重写第 5 章中的 TCP 回射客户-服务器程序,以使用 Unix 域套接口。图 14.3 给出从图 5.12 修改而来的、用 Unix 域字节流协议代替 TCP 的服务器程序。

第 8 行 两个套接口地址结构的数据类型现在是 sockaddr_un。

第 10 行 socket 的第一个参数是 AF_LOCAL,以建立 Unix 域字节流套接口。

第 11～15 行 unp.h 中定义的 UNIXSTR_PATH 常值为/tmp/unix.str。为预防路径名因前次运行这个程序而已存在的情况,我们先 unlink 这个路径名,然后在 bind 前初始化套接口地址结构。unlink 出错没有关系。

注意 bind 调用与图 14.2 中的调用不同。这里将 sockaddr_un 结构的总的大小作为套接口地址结构的大小(第三个参数),而不只是路径名成员占用的字节数。因为路径名是以空字符结尾的,所以两个长度都是正确的。

函数的其余部分和图 5.12 中的相同。使用了同样的 str_echo 函数(图 5.3)。

图 14.4 是使用 Unix 域字节流协议的回射客户程序,它由图 5.4 修改而来。

第 6 行 含有服务器地址的套接口地址结构现在是一个 sockaddr_un 结构。

第 7 行 socket 的第一个参数是 AF_LOCAL。

第 8～10 行 填写套接口地址结构的代码与服务器的相同:把结构初始化成 0,family 置为 AF_LOCAL,拷贝路径名到 sun_path 成员中。

第 12 行 str_cli 函数和前面的一样(图 6.13 是我们开发的最近一个版本)。

```
1 #include    "unp.h"
```

```
2 int
3 main(int argc, char * * argv)
4 {
5     int          listenfd, connfd;
6     pid_t        childpid;
7     socklen_t    clilen;
8     struct sockaddr_un   cliaddr, servaddr;
9     void         sig_chld(int);
10    listenfd = Socket(AF_LOCAL, SOCK_STREAM, 0);
11    unlink(UNIXSTR_PATH);
12    bzero(&servaddr, sizeof(servaddr));
13    servaddr.sun_family = AF_LOCAL;
14    strcpy(servaddr.sun_path, UNIXSTR_PATH);
15    Bind(listenfd, (SA * ) &servaddr, sizeof(servaddr));
16    Listen(listenfd, LISTENQ);
17    Signal(SIGCHLD, sig_chld);
18    for ( ; ; ) {
19        clilen = sizeof(cliaddr);
20        if ( (connfd = accept(listenfd, (SA * ) &cliaddr, &clilen)) < 0) {
21            if (errno == EINTR)
22                continue;          /* back to for() */
23            else
24                err_sys("accept error");
25        }
26        if ( (childpid = Fork()) == 0) {     /* child process */
27            Close(listenfd);       /* close listening socket */
28            str_echo(connfd);      /* process the request */
29            exit(0);
30        }
31        Close(connfd);             /* parent closes connected socket */
32    }
33 }
```

图 14.3　使用 Unix 域字节流协议的回射服务器程序[unixdomain/unixstrserv01.c]

14.6　Unix 域数据报客户-服务器程序

下面重写 8.3 和 8.4 节中的 UDP 客户-服务器程序，以使用 Unix 域数据报套接口。图 14.5 给出了从图 8.3 修改而来的服务器程序。

```
1 #include    "unp.h"
2 int
3 main(int argc, char * * argv)
4 {
5     int        sockfd;
6     struct sockaddr_un     servaddr;
7     sockfd = Socket(AF_LOCAL, SOCK_STREAM, 0);
8     bzero(&servaddr, sizeof(servaddr));
```

```
 9      servaddr.sun_family = AF_LOCAL;
10      strcpy(servaddr.sun_path, UNIXSTR_PATH);

11      Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
12      str_cli(stdin, sockfd);         /* do it all */
13      exit(0);
14  }
```

图 14.4 使用 Unix 域字节流协议的回射客户程序[unixdomain/unixstrcli01.c]

```
 1  #include    "unp.h"

 2  int
 3  main(int argc, char **argv)
 4  {
 5      int     sockfd;
 6      struct sockaddr_un  servaddr, cliaddr;

 7      sockfd = Socket(AF_LOCAL, SOCK_DGRAM, 0);

 8      unlink(UNIXDG_PATH);
 9      bzero(&servaddr, sizeof(servaddr));
10      servaddr.sun_family = AF_LOCAL;
11      strcpy(servaddr.sun_path, UNIXDG_PATH);

12      Bind(sockfd, (SA *) &servaddr, sizeof(servaddr));

13      dg_echo(sockfd, (SA *) &cliaddr, sizeof(cliaddr));
14  }
```

图 14.5 使用 Unix 域数据报协议的回射服务器程序[unixdomain/unixdgserv01.c]

第 6 行 两个套接口地址结构的数据类型现在是 sockaddr_un。

第 7 行 socket 的第一个参数是 AF_LOCAL，以建立 Unix 域数据报套接口。

第 8～12 行 unp.h 中定义的 UNIXDG_PATH 常值为/tmp/unix.dg。为预防路径名因前次运行服务器而已存在的情况，我们先 unlink 这个路径名，然后在调用 bind 前初始化套接口地址结构。unlink 出错没有关系。

第 13 行 使用同样的 dg_echo 函数(图 8.4)。

图 14.6 是使用 Unix 域数据报协议的回射客户程序。它是对图 8.7 的一个修改版本。

```
 1  #include    "unp.h"

 2  int
 3  main(int argc, char **argv)
 4  {
 5      int     sockfd;
 6      struct sockaddr_un  cliaddr, servaddr;

 7      sockfd = Socket(AF_LOCAL, SOCK_DGRAM, 0);

 8      bzero(&cliaddr, sizeof(cliaddr));       /* bind an address for us */
 9      cliaddr.sun_family = AF_LOCAL;
10      strcpy(cliaddr.sun_path, tmpnam(NULL));

11      Bind(sockfd, (SA *) &cliaddr, sizeof(cliaddr));

12      bzero(&servaddr, sizeof(servaddr));  /* fill in server's address */
13      servaddr.sun_family = AF_LOCAL;
14      strcpy(servaddr.sun_path, UNIXDG_PATH);
```

```
15      dg_cli(stdin, sockfd, (SA *) &servaddr, sizeof(servaddr));
16      exit(0);
17  }
```

图 14.6　使用 Unix 域数据报协议的回射客户程序[unixdomain/unixdgcli01.c]

第 6 行　含有服务器地址的套接口地址结构现在是一个 sockaddr_un 结构。我们还分配了一个这样的结构以存放客户的地址。

第 7 行　socket 的第一个参数是 AF_LOCAL。

第 8～11 行　与 UDP 客户不同的是，当使用 Unix 域数据报协议时，我们必须显式地给套接口 bind 一个路径名，这样服务器才能有路径名发回结果。我们调用 tmpnam 分配一个唯一的路径名，然后用 bind 来将它捆绑到套接口上。回想 14.4 节，在一个未绑定的 Unix 域数据报套接口上发出数据报不会自动给套接口捆绑路径名。因此如果省掉这一步，服务器在 dg_echo 中的 recvfrom 调用会返回一个空路径名，这将使服务器在调用 sendto 时发生错误。

第 12～14 行　向套接口地址结构中填入服务器的众所周知路径名的代码和以前的服务器代码相同。

第 15 行　dg_cli 函数和以前的一样(图 8.8)。

14.7　描述字传递

当考虑从一个进程向另一个进程传递打开的描述字时，我们通常会想到：

1. 在 fork 调用后，子进程共享父进程的所有打开的描述字。
2. 在调用 exec 时所有描述字仍保持打开。

第一个例子中进程打开一个描述字，调用 fork，然后父进程关闭描述字，让子进程处理这个描述字。这样将一个打开的描述字从父进程传递到子进程。但我们也想让子进程打开一个描述字并将其传给父进程。

当前的 Unix 系统提供了一种方法，可以从一个进程向其他任何进程传递打开的描述字。也就是说，进程之间不需要有什么关系，譬如父子进程。这种技术要求先在两个进程之间建立一个 Unix 域套接口，然后用 sendmsg 从这个套接口上发一个特殊的消息。这个消息由内核做特殊处理，以将打开的描述字从发送方传递到接收方。

TCPv3 的第 18 章中介绍了 4.4BSD 在 Unix 域套接口上传递打开的描述字的过程细节。

SVR4 在内核中使用了一种不同的技术来传递打开的描述字，即 I_SENDFD 和 I_RECVFD 这两个 ioctl 命令，APUE 的 15.5.1 节中对此作了介绍。但进程仍然可以使用 Unix 域套接口来访问这个内核功能。在本书中我们介绍怎样用 Unix 域套接口来传递描述字，因为这是可移植性最好的编程技术：它在源自 Berkeley 的内核或 SVR4 上都能工作，而 I_SENDFD 和 I_RECVFD 这两个 ioctl 命令只能在 SVR4 上使用。

4.4BSD 的技术允许一次 sendmsg 传递多个描述字，而 SVR4 的技术一次只能

传递一个描述字。我们所有的例子都是一次传递一个描述字。

在两个进程之间传递描述字的步骤如下：

1. 创建一个字节流的或数据报的 Unix 域套接口。
   如果目标是 fork 一个子进程，让子进程打开描述字并将它传回给父进程，那么父进程可以用 socketpair 创建一个流管道，用它来传递描述字。
   如果进程之间没有亲缘关系，那么服务器必须创建一个 Unix 域字节流套接口，bind 一个路径名，让客户 connect 到这个套接口。然后客户可以向服务器发送一个请求以打开某个描述字，服务器将描述字通过 Unix 域套接口传回。在客户和服务器之间也可以使用 Unix 域数据报套接口，但这样做没什么好处，而且数据报存在丢失的可能性。在本节的例子中客户和服务器之间将使用字节流套接口。
2. 进程可以用任何返回描述字的 Unix 函数打开一个描述字：譬如 open、pipe、mkfifo、socket 或 accept。可以在进程之间传递任何类型的描述字，这是为什么我们将这种技术称为“传递描述字”而不是“传递文件描述字”的原因。
3. 发送进程建立一个 msghdr 结构(13.5 节)，其中包含要传递的描述字。Posix.1g 中说明该描述字作为辅助数据(msghdr 结构的 msg_control 成员，见 13.6 节)发送，但老的实现使用 msg_accright 成员。发送进程调用 sendmsg 通过第一步得到的 Unix 域套接口发出描述字。这时我们说这个描述字是“在飞行中(in flight)”的。即使在发送进程调用 sendmsg 之后，但在接收进程调用 recvmsg(下一步要做的)之前将描述字关闭，它仍会为接收进程保持打开状态。描述字的发送导致它的访问计数加 1。
4. 接收进程调用 recvmsg 在 Unix 域套接口上接收描述字。通常接收进程收到的描述字的编号和发送进程中的描述字的编号不同，但这没有问题。传递描述字不是传递描述字的编号，而是在接收进程中创建一个新的描述字，指向内核的文件表中与发送进程发送的描述字相同的项。

客户和服务器之间必须有某种应用协议，使接收方知道何时接收描述字。如果接收方调用 recvmsg 但没有分配接收描述字的空间，而且有一个描述字已被传递并正待读出，这个已传递的描述字就会关闭(TCPv2 第 518 页)。在用 recvmsg 接收描述字时还要避免使用 MSG_PEEK 标志，否则后果不可预料。

描述字传递的例子

下面提供一个描述字传递的例子。我们将编写一个叫 mycat 的程序，它从命令行得到一个路径名，打开这个文件，并将文件的内容拷贝到标准输出。但是该程序不用通常的 Unix open 函数，而是调用自己的名为 my_open 的函数。这个函数创建一个流管道，调用 fork 和 exec 执行另一个程序，这个程序才打开想要的文件。接着这个程序将打开的描述字通过流管道传回给父进程。

图 14.7 展示了第一步：调用 socketpair 创建流管道后的 mycat 进程。我们用[0]和[1]来指明 socketpair 返回的两个描述字。

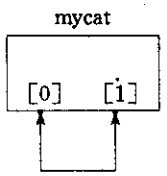


图 14.7　用 socketpair 创建流管道后的 mycat 进程

然后 mycat 进程调用 fork，子进程调用 exec 执行 openfile 程序。父进程关闭[1]描述字，子进程关闭[0]描述字。(流管道的两端没有什么不同。也可以让子进程关闭[1]，父进程关闭[0]。)这样会得到图 14.8 所示的结果。

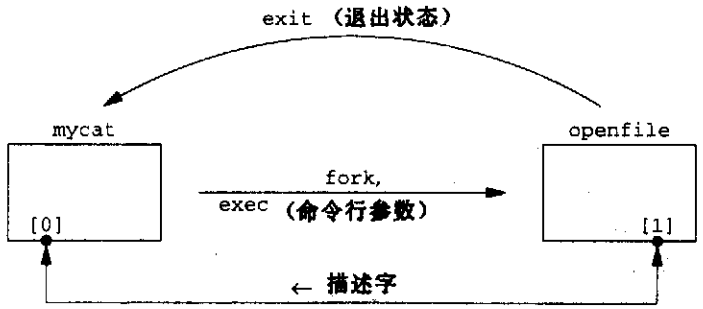


图 14.8　调用 openfile 程序后的 mycat 进程

父进程必须给 openfile 程序传递三个信息：(1)要打开的文件的路径名，(2)打开方式(只读、读写或只写)，以及(3)流管道的本进程端(就是图中的[1])对应的描述字的编号。我们选择以命令行方式在 exec 调用时传递这三个信息。另一种方法是通过流管道将这些信息作为数据发送过去。openfile 程序通过流管道发回打开的描述字并终止。程序的退出状态告诉父进程文件能否打开，如果不能打开则发生了什么类型的错误。

通过执行另一个程序来打开文件的好处在于，另一个程序可以是一个 setuid 到 root 的程序，能打开原来没有权限打开的文件。这个程序能够扩展通常的 Unix 权限(用户、用户组和其他用户)到它想要的任何形式的访问检查。

下面以 mycat 程序开始讨论，见图 14.9。

```
1 #include    "unp.h"
2 int     my_open(const char *, int);
3 int
4 main(int argc, char **argv)
5 {
6     int     fd, n;
7     char    buff[BUFFSIZE];
8     if (argc != 2)
9         err_quit("usage: mycat <pathname>");
10    if ( (fd = my_open(argv[1], O_RDONLY)) < 0)
11        err_sys("cannot open %s", argv[1]);
```

```
12      while ( (n = Read(fd, buff, BUFFSIZE)) > 0)
13          Write(STDOUT_FILENO, buff, n);
14      exit(0);
15  }
```

图 14.9 mycat 程序:把一个文件拷贝到标准输出[unixdomain/mycat.c]

如果把 my_open 换成 open,这个简单的程序只是将一个文件拷贝到标准输出。

图 14.10 中的 my_open 函数和通常的 Unix open 函数看起来一样。它取两个参数:一个路径名和一个打开方式(譬如 O_RDONLY 表示只读),打开该文件并返回描述字。

```
1  #include    "unp.h"
2  int
3  my_open(const char *pathname, int mode)
4  {
5      int         fd, sockfd[2], status;
6      pid_t       childpid;
7      char        c, argsockfd[10], argmode[10];
8      Socketpair(AF_LOCAL, SOCK_STREAM, 0, sockfd);
9      if ( (childpid = Fork()) == 0) {            /* child process */
10         Close(sockfd[0]);
11         snprintf(argsockfd, sizeof(argsockfd), "%d", sockfd[1]);
12         snprintf(argmode, sizeof(argmode), "%d", mode);
13         execl("./openfile", "openfile", argsockfd, pathname, argmode,
14                 (char *) NULL);
15         err_sys("execl error");
16     }
17     /* parent process - wait for the child to terminate */
18     Close(sockfd[1]);          /* close the end we don't use */
19     Waitpid(childpid, &status, 0);
20     if (WIFEXITED(status) == 0)
21         err_quit("child did not terminate");
22     if ( (status = WEXITSTATUS(status)) == 0)
23         Read_fd(sockfd[0], &c, 1, &fd);
24     else {
25         errno = status;        /* set errno value from child's status */
26         fd = -1;
27     }
28     Close(sockfd[0]);
29     return(fd);
30 }
```

图 14.10 my_open 函数:打开一个文件并返回其描述字[unixdomain/myopen.c]

创建流管道

第 8 行 socketpair 创建一个流管道,返回两个描述字:sockfd[0]和 sockfd[1]。这是图 14.7 中展示的情形。

fork 并 exec

第 9~16 行 调用 fork,然后子进程关闭流管道的一端。流管道另一端的描述字编号填

入 argsockfd 数组，打开方式填入 argmode 数组。这里调用 snprintf 是因为 exec 的参数必须是字符串。openfile 程序随后执行。execl 函数不应该返回，除非出错。如果成功，openfile 程序的 main 函数开始执行。

父进程等待子进程

第 17～22 行 父进程关闭流管道的另一端并调用 waitpid 等待子进程的结束。子进程的终止状态在 status 变量中返回。先要检查一下它是否正常终止(即它不是被信号终止)。接着 WEXITSTATUS 宏把终止状态转换成退出状态，退出状态的取值在 0～255 之间。我们马上就会看到，如果 openfile 程序在打开文件时出错，它将以对应的 errno 值作为退出状态来终止。

接收描述字

第 23 行 下面给出的 read_fd 函数在流管道上接收描述字。除了描述字之外，它还读 1 字节数据，但不对数据做任何处理。

> 在流管道上发送和接收描述字时，我们总是发送至少 1 字节的数据，即使接收方不对数据进行任何处理。否则接收方就分不清 read_fd 的返回值为 0 时，意味着“没有数据(但可能有一个描述字)”还是“文件结束”符。

图 14.11 给出了 read_fd 函数，它调用 recvmsg 在 Unix 域套接口上接收数据和描述字。这个函数的前三个参数和 read 函数一样，第四个参数是一个指向整数的指针，以返回接收到的描述字。

第 9～26 行 这个函数必须处理两种版本的 recvmsg：使用 msg_control 成员的和使用 msg_accrights 成员的。如果支持 msg_control 版本，我们的 config.h 头文件(图 D.2)中就会定义常值 HAVE_MSGHDR_MSG_CONTROL。

使 msg_control 适当地对齐

第 10～13 行 msg_control 缓冲区必须与 cmsghdr 结构适当地对齐。简单地分配一个字符数组是不合适的。这里声明了一个 cmsghdr 结构和一个字符数组的联合，以确保数组能适当地对齐。另一种方法是调用 malloc，但需要在函数返回前释放内存。

第 27～45 行 调用 recvmsg。如果返回辅助数据，则其格式与图 13.13 中所示的一样。检查长度、级别和类型是否正确，然后取出新建的描述字，并通过调用者的 recvfd 指针返回。CMSG_DATA 返回一个 unsigned char 指针，指向辅助数据对象的 cmsg_data 成员。我们将它强制成一个 int 指针并取出它指向的整型描述字。

```
1 #include    "unp.h"
2 ssize_t
3 read_fd(int fd, void *ptr, size_t nbytes, int *recvfd)
4 {
5     struct msghdr msg;
6     struct iovec iov[1];
7     ssize_t n;
8     int         newfd;
9 #ifdef  HAVE_MSGHDR_MSG_CONTROL
10    union {
```

```
11          struct cmsghdr   cm;
12          char     control[CMSG_SPACE(sizeof(int))];
13      } control_un;
14      struct cmsghdr   *cmptr;

15      msg.msg_control = control_un.control;
16      msg.msg_controllen = sizeof(control_un.control);
17  #else
18      msg.msg_accrights = (caddr_t) &newfd;
19      msg.msg_accrightslen = sizeof(int);
20  #endif

21      msg.msg_name = NULL;
22      msg.msg_namelen = 0;

23      iov[0].iov_base = ptr;
24      iov[0].iov_len = nbytes;
25      msg.msg_iov = iov;
26      msg.msg_iovlen = 1;

27      if ( (n = recvmsg(fd, &msg, 0)) <= 0)
28          return(n);

29  #ifdef      HAVE_MSGHDR_MSG_CONTROL
30      if ( (cmptr = CMSG_FIRSTHDR(&msg)) != NULL &&
31          cmptr->cmsg_len == CMSG_LEN(sizeof(int))) {
32          if (cmptr->cmsg_level != SOL_SOCKET)
33              err_quit("control level != SOL_SOCKET");
34          if (cmptr->cmsg_type != SCM_RIGHTS)
35              err_quit("control type != SCM_RIGHTS");
36          *recvfd = *((int *) CMSG_DATA(cmptr));
37      } else
38          *recvfd = -1;       /* descriptor was not passed */
39  #else

40      if (msg.msg_accrightslen == sizeof(int))
41          *recvfd = newfd;
42      else
43          *recvfd = -1;       /* descriptor was not passed */
44  #endif

45      return(n);
46  }
```

图 14.11 read_fd 函数:接收数据和描述字[lib/read_fd.c]

如果支持老的 msg_accrights 成员,长度应为一个整数的大小,新建的描述字通过调用者的 recvfd 指针返回。

图 14.12 给出了 openfile 程序。它从命令行得到调用 open 函数必需的三个参数。

```
1 #include    "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int         fd;
6     ssize_t     n;
```

```
 7    if (argc != 4)
 8        err_quit("openfile <sockfd#> <filename> <mode>");
 9    if ((fd = open(argv[2], atoi(argv[3]))) < 0)
10        exit((errno > 0) ? errno : 255);
11    if ((n = write_fd(atoi(argv[1]), "", 1, fd)) < 0)
12        exit((errno > 0) ? errno : 255);
13    exit(0);
14 }
```

图 14.12　openfile 函数：打开一个文件并传回描述字[unixdomain/openfile.c]

命令行参数

第 7～12 行　因为三个参数中的两个被 my_open 转换成字符串，所以用 atoi 将它们转回整数。

打开文件

第 9～10 行　调用 open 打开文件。如果出错，open 对应的 errno 的值作为进程的退出状态返回。

传回描述字

第 11～12 行　下面给出的 write_fd 将描述字传回。这个进程随后终止，但本章的前面曾说过发送进程在传递完描述字后将其关闭（调用 exit 时将发生这种情况）不会有任何问题，因为内核知道描述字在飞行中，会为接收进程保持打开状态。

> 退出状态码必须在 0 和 255 之间。最大的 errno 值大概是 150。另一种不要求 errno 的值小于 256 的方法是，在调用 sendmsg 时将错误代码作为普通数据传回。

图 14.13 给出了最后一个函数即 write_fd，它调用 sendmsg 通过 Unix 域套接口发送描述字（和可选的数据，这里没有用上）。

```
 1 #include    "unp.h"
 2 ssize_t
 3 write_fd(int fd, void *ptr, size_t nbytes, int sendfd)
 4 {
 5     struct msghdr    msg;
 6     struct iovec     iov[1];
 7 #ifdef    HAVE_MSGHDR_MSG_CONTROL
 8     union {
 9       struct cmsghdr   cm;
10       char             control[CMSG_SPACE(sizeof(int))];
11     } control_un;
12     struct cmsghdr    *cmptr;
13     msg.msg_control = control_un.control;
14     msg.msg_controllen = sizeof(control_un.control);
15     cmptr = CMSG_FIRSTHDR(&msg);
16     cmptr->cmsg_len = CMSG_LEN(sizeof(int));
17     cmptr->cmsg_level = SOL_SOCKET;
```

```
18      cmptr->cmsg_type = SCM_RIGHTS;
19      *((int *) CMSG_DATA(cmptr)) = sendfd;
20 #else
21      msg.msg_accrights = (caddr_t) &sendfd;
22      msg.msg_accrightslen = sizeof(int);
23 #endif

24      msg.msg_name = NULL;
25      msg.msg_namelen = 0;

26      iov[0].iov_base = ptr;
27      iov[0].iov_len = nbytes;
28      msg.msg_iov = iov;
29      msg.msg_iovlen = 1;

30      return(sendmsg(fd, &msg, 0));
31 }
```

图14.13 write_fd函数:调用sendmsg传递描述字[lib/write_fd.c]

read_fd函数必须既能处理辅助数据又能处理老的访问权限。任何一种情况都要初始化msghdr结构,然后调用sendmsg。

在25.7节中举了一个无亲缘关系进程之间描述字传递的例子,27.9节中举了一个有亲缘关系进程之间传递描述字的例子。这些例子中将使用以上介绍的read_fd和write_fd函数。

14.8 接收发送者的凭证

图13.13列出了另一种能在Unix域套接口上作为辅助数据传送的数据:fcred结构的用户凭证,该结构在<sys/ucred.h>头文件中定义。

```
struct fcred {
  uid_t     fc_ruid;                  /* real user ID */
  gid_t     fc_rgid;                  /* real group ID */
  char      fc_login[MAXLOGNAME];     /* setlogin() name */
  uid_t     fc_uid;                   /* effective user ID */
  short     fc_ngroups;               /* number of groups */
  gid_t     fc_groups[NGROUPS];       /* supplementary group IDs */
};
#define     fc_gid fc_groups[0]       /* effective group ID */
```

通常MAXLOGNAME为16,NGROUPS也是16。fc_ngroups总是至少为1,fc_groups数组的第一个元素为有效的组ID。

这个结构实际的定义是一个在fcred结构内的ucred结构,只是用#define使它看上去像一个单独的结构。上面列出的是对该结构的"逻辑"定义。

这个功能在BSD/OS2.1中是新增的。虽然这种功能还没普及,但是因为它是对Unix域协议的一个十分重要和简单的补充,所以在这里对它进行介绍。当客户和服务器进行通信时,服务器通常需要一种方法确认客户,以验证客户有权限请求该服务。

只要遵循以下条件，该信息在 Unix 域套接口上总是可用的。

- 凭证是作为辅助数据在 Unix 域套接口上发送的，但接收方必须打开 LOCAL_CREDS 套接口选项。这个选项的级别(7.2 节)为 0。
- 在数据报套接口上，每个数据报都带有凭证。在字节流套接口上凭证只发送一次，是在第一次发送数据时发出的。
- 凭证不能和描述字一起发送。也就是说，在单个消息中只能发送这两种辅助数据中的一种。

  这是 BSD/OS 实现中的一个限制。按常理，既然每个辅助数据对象都有自己的类型和长度，在一个 sendmsg 调用中应能传送多种类型的辅助数据(图 13.12)。

- 用户是不能伪造凭证的。也就是说，在 Unix 域套接口上发送辅助数据时，内核会加以检验，确保不是级别为 SOL_SOCKET、类型为 SCM_CREDS 的辅助数据。如果发送方试图伪造凭证，辅助数据会被内核丢弃。

前一节中有一点没有提到，在 SVR4 上收到一个描述字时(用 ioctl 的 I_RECVFD 指令)，内核也将发送方的凭证传送给接收进程：一个含有新建的描述字、有效用户 ID 和有效用户组 ID 的 strrecvfd 结构。这是每次传递描述字时都会传送的凭证的形式。另外，当用 connld 流模块建立客户-服务器连接时(与在 Unix 域套接口上用 accept 建立一个新的套接口类似)，新的描述字和一个包含客户凭证的 strrecvfd 结构一起传回。在 SVR4 上不能用 Unix 域套接口访问这些凭证。(APUE 的 15.5.1 节详细介绍了 SVR4 的 connld 流模块的使用。)如果一个源自 Berkeley 的实现不支持本节介绍的新的凭证传送手段，就不能保证 Unix 域服务器能得到客户的凭证。APUE 的 15.5.2 节中详细介绍了获得这些信息的一种迂回方法，但用户凭证应总是由内核来提供。

例子

作为传送凭证的一个例子，我们修改 Unix 域字节流服务器程序，使它向客户请求凭证。图 14.14 给出了一个名为 read_cred 的新函数，它和 read 类似，但还返回一个含有发送方凭证的 fcred 结构。

第 4～5 行　前三个参数和 read 相同，第四个参数是一个指向 fcred 结构的指针。返回的辅助数据的格式如图 13.13 所示。

第 26～36 行　如果返回了凭证，就检验辅助数据的长度、级别和类型，并将结果拷回给调用者。如果没有返回凭证，就将结构置为 0。因为用户组的数目(fc_ngroups)总是大于等于 1 的，它的值为 0 对调用者表示内核没有返回凭证。

图 14.3 中回射服务器的 main 函数没有变化。图 14.15 给出了从图 5.3 修改而来的 str_echo 函数的新版本。这个函数由子进程在父进程接收了一个新的客户连接并 fork 之后调用。

第 12 行　对已连接套接口设置 LOCAL_CREDS 套接口选项。

第 13～14 行　第一次调用 read_cred 函数。将长度置为 0，表示不接收一般数据，只想接收辅助数据。

第17～27行 如果返回了凭证，将其输出。

第28～32行 循环剩下的部分没变。这段代码从客户读入数据，再将数据返回给客户。从图14.4而来的客户程序没有改变。

```
1 #include    "unp.h"
2 #include    <sys/param.h>
3 #include    <sys/ucred.h>
4 ssize_t
5 read_cred(int fd, void *ptr, size_t nbytes, struct fcred *fcredptr)
6 {
7     struct msghdr msg;
8     struct iovec iov[1];
9     ssize_t n;
10    union {
11      struct cmsghdr  cm;
12      char    control[CMSG_SPACE(sizeof(struct fcred))];
13    } control_un;
14    struct cmsghdr    *cmptr;
15    msg.msg_control = control_un.control;
16    msg.msg_controllen = sizeof(control_un.control);
17    msg.msg_name = NULL;
18    msg.msg_namelen = 0;
19    iov[0].iov_base = ptr;
20    iov[0].iov_len = nbytes;
21    msg.msg_iov = iov;
22    msg.msg_iovlen = 1;
23    if ( (n = recvmsg(fd, &msg, 0)) < 0)
24        return(n);
25    if (fcredptr) {
26        if (msg.msg_controllen > sizeof(struct cmsghdr)) {
27            cmptr = CMSG_FIRSTHDR(&msg);
28            if (cmptr->cmsg_len != CMSG_LEN(sizeof(struct fcred)))
29                err_quit("control length = %d", cmptr->cmsg_len);
30            if (cmptr->cmsg_level != SOL_SOCKET)
31                err_quit("control level != SOL_SOCKET");
32            if (cmptr->cmsg_type != SCM_CREDS)
33                err_quit("control type != SCM_CREDS");
34            memcpy(fcredptr, CMSG_DATA(cmptr), sizeof(struct fcred));
35        } else
36            bzero(fcredptr, sizeof(struct fcred)); /* none returned */
37    }
38    return(n);
39 }
```

图14.14 read_cred函数：读取并返回发送方的凭证[unixdomain/readcred.c]

```
1 #include    "unp.h"
2 #include    <sys/param.h>
3 #include    <sys/ucred.h>
```

```
4 ssize_t     read_cred(int, void *, size_t, struct fcred *);
5 void
6 str_echo(int sockfd)
7 {
8     ssize_t n;
9     const int on = 1;
10    char     line[MAXLINE];
11    struct fcred cred;
12    Setsockopt(sockfd, 0, LOCAL_CREDS, &on, sizeof(on));
13    if ( (n = read_cred(sockfd, NULL, 0, &cred)) < 0)
14        err_sys("read_cred error");
15    if (cred.fc_ngroups == 0)
16        printf("(no credentials returned)\n");
17    else {
18        printf("real user ID = %d\n", cred.fc_ruid);
19        printf("real group ID = %d\n", cred.fc_rgid);
20        printf("login name = %-*s\n", MAXLOGNAME, cred.fc_login);
21        printf("effective user ID = %d\n", cred.fc_uid);
22        printf("effective group ID = %d\n", cred.fc_gid);
23        printf("%d supplementary groups:", cred.fc_ngroups - 1);
24        for (n = 1; n < cred.fc_ngroups; n++)      /* [0] is the egid */
25            printf(" %d", cred.fc_groups[n]);
26        printf("\n");
27    }
28    for ( ; ; ) {
29        if ( (n = Readline(sockfd, line, MAXLINE)) == 0)
30            return;              /* connection closed by other end */
31        Writen(sockfd, line, n);
32    }
33 }
```

图 14.15　请求客户凭证的 str_echo 函数[unixdomain/strecho.c]

如果一个窗口运行服务器，另一个窗口运行客户，下面就是运行一次客户程序后服务器的输出。

```
bsdi % unixstrserv02
real user ID = 482
real group ID = 52
login name = rstevens
effective user ID = 482
effective group ID = 52
7 supplementary groups: 20 0 1 2 3 5 7
```

因为前面曾经提到过，在字节流套接口上凭证是在第一次发送数据时(不是在连接建立时)由内核一块发送的，所以客户发出第一行数据后服务器才输出信息。这和前面提到的 SVR4 中的技术有所不同，SVR4 是在返回新建的描述字时一块发送凭证的(对于 Unix 域套接口来说，这相当于 accept 的返回)。

14.9 小 结

客户和服务器在同一台主机上时，Unix 域套接口是 IPC 的一种替代方法。和 IPC 相比使用 Unix 域套接口的好处在于，对网络客户-服务器的 API 几乎是相同的。当客户和服务器在同一台主机上时，Unix 域套接口和 TCP 相比，其性能在多数实现中要更高。

为了使用 Unix 域协议，我们对自己的 TCP 和 UDP 回射客户和服务器程序做了修改，唯一主要的差别是 UDP 客户的套接口必须 bind 一个路径名，UDP 服务器才有地方发送应答。14.5 和 14.6 节中的代码是直接操作 Unix 域套接口地址结构的，更好的方法是使用 11 章里的 tcp_XXX 和 udp_XXX 函数，因为 getaddrinfo 的实现支持 Unix 域套接口。

同一台主机上的客户和服务器之间传递描述字是一种强大的技术，它是通过 Unix 域套接口进行的。在 14.7 节中列举了一个从子进程向父进程传回描述字的例子。25.7 节中将列举没有亲缘关系的客户和服务器进程传递描述字的例子，27.9 节中将列举另一个从父进程向子进程传递描述字的例子。

14.10 习 题

14.1 如果一个 Unix 域服务器在调用 bind 之后调用 unlink，会发生什么结果？

14.2 如果 Unix 域服务器在终止时不 unlink 它的众所周知路径名，并且有一个客户在服务器终止后试图 connect 该服务器，将会发生什么结果？

14.3 编译图 11.12 和 11.15 中独立于协议的 UDP 时间/日期客户和服务器程序，并指定一个 Unix 域套接口(11.6 节)来运行这些程序。会发生什么结果？你怎样修改能使其正常运行？

14.4 对图 11.7 进行修改，在输出对方的协议地址后调用 sleep(5)，每次 read 返回一个大于 0 的值时输出读到的字节数。对图 11.10 进行修改，对向客户发送的结果中的每个字节都调用 write。(在习题1.5的解答中讨论了类似的修改。)在同一台主机上以 TCP 运行客户和服务器。客户读到了多少字节？
在同一台主机上以 Unix 域套接口运行客户和服务器。结果有没有变化？
再在服务器程序中用 send 代替 write，并设置 MSG_EOR 标志。(完成这道题需要一个源自 Berkeley 的实现。)在同一台主机上以 Unix 域套接口运行客户和服务器。结果有没有变化？

14.5 编写一个程序以确定图 4.10 中展示的值。一种方法是建立一个流管道然后 fork。父进程进入一个 for 循环，backlog 从 0 增加到 14。每次循环父进程首先把 backlog 的值写到流管道中。子进程读取这个值，建立一个捆绑到回馈地址的监听套接口，将 backlog 设为该值。然后子进程写流管道，告诉父进程它已准备好。接着父进程尝试建立尽可能多的连接，但设置一个 2 秒的 alarm，因为达到 backlog 上限的 connect 调用将会阻塞，重发 SYN。子进程不调用 accept 以让内核把父进程来的连接送入缓冲队列。当父进程的 alarm 超时时，它从循环的计数器可以知道哪个 connect 达到了 backlog 的上限。在这之后父进程关闭套接口，

并将下一个新的 backlog 的值写到流管道。当子进程读到这下一个值时，它关闭原来的监听套接口，建立一个新的监听套接口，又重新开始这个过程。

14.6　验证在图 14.6 中删掉 bind 调用会导致服务器出错。

第15章 非阻塞I/O

15.1 概 述

缺省状态下,套接口是阻塞方式的。这意味着当一个套接口调用不能立即完成时,进程进入睡眠状态,等待操作完成。我们将可能阻塞的套接口调用分成四种。

1. 输入操作:read、readv、recv、recvfrom 和 recvmsg 函数。如果在一个阻塞的 TCP 套接口(缺省情况)上调用这些函数,而且在套接口接收缓冲区中没有数据,进程将在数据到来前一直睡眠。因为 TCP 是一个字节流,当有数据到来时进程将被唤醒:可以是一个字节的数据,也可以是一个完整的 TCP 数据分节。如果希望等到某个固定数目的数据再返回,可以调用一个自己写的函数 readn(图3.14),或指定 MSG_WAITALL 标志(图 13.6)。

   因为 UDP 是一个数据报协议,如果一个阻塞的 UDP 套接口的接收缓冲区为空,进程将在一个 UDP 数据报到来之前一直处于睡眠状态。

   在一个非阻塞套接口上,如果输入操作不能被满足(在 TCP 套接口上至少有一个字节的数据,或在 UDP 套接口上有一个完整的数据报),它们将会立即返回一个 EWOULDBLOCK 错误。

2. 输出操作:write、writev、send、sendto 和 sendmsg 函数。我们在 2.9 节中说过,对一个 TCP scoket,内核从应用进程缓冲区向套接口发送缓冲区中拷贝数据。如果在套接口发送缓冲区中没有空间,进程会一直睡眠到腾出空间。

   对于一个非阻塞 TCP 套接口,如果在套接口的发送缓冲区中没有空间,输出操作会立即返回一个 EWOULDBLOCK 错误。如果发送缓冲区中有一些空间,返回值为内核能向缓冲区中拷贝的字节数。(这叫做不足计数(short count)。)

   在 2.9 节中还说过,实际上 UDP 套接口没有发送缓冲区。内核只是拷贝应用进程数据并将其向协议栈的下层传递,加上 UDP 和 IP 头部。因此在一个阻塞 UDP 套接口上的输出操作不会阻塞。

3. 接收外来连接:accept 函数。如果在一个阻塞套接口上调用 accept 函数,而且没有新的连接,进程就会进入睡眠状态。

   如果在一个非阻塞套接口上调用 accept 函数,而且没有新的连接,将返回 EWOULDBLOCK 错误。

4. 初始化外出的连接:用于 TCP 的 connect 函数。(回想 connect 可用于 UDP,但它不会建立一个"真实"的连接;它只是让内核保存对方的 IP 地址和端口号。)在 2.5 节中展示了 TCP 连接的建立包含一个三路握手过程,而且 connect 函数在客户接收到它的 SYN 的 ACK 前不会返回。这意味着 TCP connect 总是会使调用它的进程阻塞起码到服务器的一次往返时间(RTT)。

如果在一个非阻塞的 TCP 套接口上调用 connect，而且连接不能马上建立，连接的建立过程将开始（譬如 TCP 三路握手的第一个分组将发出），但返回一个 EINPROGRESS 错误。注意这个错误和前面三种情况下返回的不同。还要注意的是一些连接可以立即建立，这一般在服务器和客户位于同一台主机上时发生，所以即使对于一个非阻塞的 connect，也必须准备处理 connect 返回成功的情况。在 15.3 节中将举一个非阻塞 connect 的例子。

通常，对于不能马上完成的非阻塞 I/O 操作，系统 V 返回 EAGAIN 错误，而源自 Berkeley 的实现返回 EWOULDBLOCK 错误。更混乱的是，Posix.1 指定使用 EAGAIN 而 Posix.1g 指定使用 EWOULDBLOCK。幸运的是，大多数当前使用的系统（包括 SVR4 和 4.4BSD）将这两个错误码定义为相同的值（检查一下你使用的系统中的<sys/errno.h>头文件），因此不管用哪个都没有关系。在本文中我们使用 Posix.1g 中指定的 EWOULDBLOCK。

6.2 节总结了 I/O 可以使用的不同模型，并比较了非阻塞 I/O 和其他模型。本章将提供全部四种类型操作的例子，并且开发一个与 Web 客户程序类似的新的客户程序，它使用非阻塞 connect 同时开始多个 TCP 连接。

15.2　非阻塞读和写：str_cli 函数（修订版）

我们再次回到在 5.5 和 6.4 节中讨论过的 str_cli 函数。后者使用 select 的那个版本，使用的仍是阻塞 I/O。举例来说，如果在标准输入有一行可读，我们用 fgets 读入它，然后用 writen 把它发往服务器。但是如果套接口的缓冲区已满，writen 调用会阻塞。当 writen 调用阻塞时，套接口的接收缓冲区中可能会有可读的数据。类似的，当标准输出比网络慢时，如果套接口上有一行输入可读，我们可能阻塞在后面的 fputs 调用上。这一节的目标是开发这个函数的一个使用非阻塞 I/O 的版本。这样可以避免阻塞，同时做一些有价值的事情。

不幸的是加入非阻塞 I/O 使函数的缓冲区管理显著地复杂化了，因此下面分成若干部分介绍这个函数。我们还要修改对标准输入和标准输出的处理，改为直接使用 read 和 write，以代替使用标准 I/O。这样避免了在非阻塞描述字上使用标准 I/O，这是防止发生问题的良方。

我们维护两个缓冲区：to 容纳从标准输入到服务器去的数据，fr 容纳自服务器到标准输出来的数据。图 15.1 展示了 to 缓冲区的组织和指向缓冲区中的指针。

toiptr 指针指向从标准输入读入的数据可以存放的下一个字节。tooptr 指向下一个必须写入套接口的字节。有（toiptr-tooptr）个字节需写到套接口。可从标准输入读入的字节数是（&to[MAXLINE]－toiptr）。tooptr 一旦到达 oiptr，两个指针都复位到缓冲区的开始。

图 15.2 展示了相应的 fr 缓冲区的组织。

图 15.3 给出了函数的第一部分。

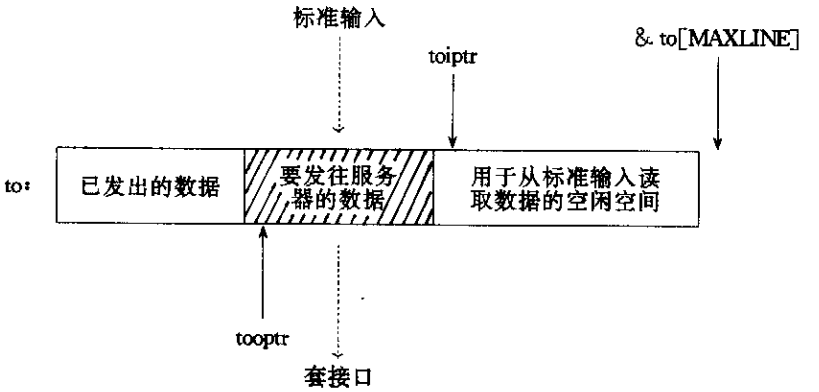


图 15.1 容纳从标准输入到套接口的数据的缓冲区

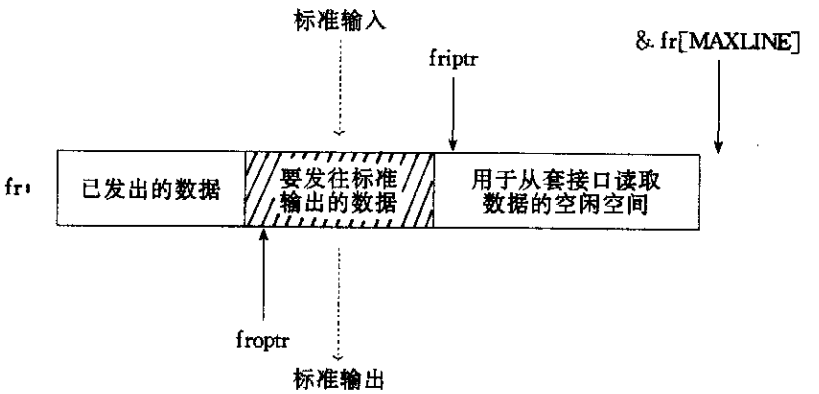


图 15.2 容纳从套接口到标准输出的数据的缓冲区

```
1  #include    "unp.h"
2  void
3  str_cli(FILE *fp, int sockfd)
4  {
5      int          maxfdp1, val, stdineof;
6      ssize_t      n, nwritten;
7      fd_set       rset, wset;
8      char         to[MAXLINE], fr[MAXLINE];
9      char         *toiptr, *tooptr, *friptr, *froptr;
10     val = Fcntl(sockfd, F_GETFL, 0);
11     Fcntl(sockfd, F_SETFL, val | O_NONBLOCK);
12     val = Fcntl(STDIN_FILENO, F_GETFL, 0);
13     Fcntl(STDIN_FILENO, F_SETFL, val | O_NONBLOCK);
14     val = Fcntl(STDOUT_FILENO, F_GETFL, 0);
15     Fcntl(STDOUT_FILENO, F_SETFL, val | O_NONBLOCK);
16     toiptr = tooptr = to;    /* initialize buffer pointers */
17     friptr = froptr = fr;
18     stdineof = 0;
19     maxfdp1 = max(max(STDIN_FILENO, STDOUT_FILENO), sockfd) + 1;
20     for ( ; ; ) {
```

```
21          FD_ZERO(&rset);
22          FD_ZERO(&wset);
23          if (stdineof == 0 && toiptr < &to[MAXLINE])
24              FD_SET(STDIN_FILENO, &rset);  /* read from stdin */
25          if (friptr < &fr[MAXLINE])
26              FD_SET(sockfd, &rset);        /* read from socket */
27          if (tooptr != toiptr)
28              FD_SET(sockfd, &wset);        /* data to write to socket */
29          if (froptr != friptr)
30              FD_SET(STDOUT_FILENO, &wset); /* data to write to stdout */
31          Select(maxfdp1, &rset, &wset, NULL, NULL);
```

图 15.3　str_cli 函数：第一部分，初始化并调用 select[nonblock/strclinonb.c]

将描述字设置为非阻塞

第 10～15 行　用 fcntl 把全部三个描述字设置为非阻塞：连到服务器的套接口、标准输入和标准输出。

初始化缓冲区指针

第 16～19 行　初始化指向两个缓冲区的指针，并将最大的描述字加 1，以用作 select 的第一个参数。

主循环：准备调用 select

第 20 行　和这个函数在前面图 6.13 中的版本相似，函数的主循环是一个 select 调用，紧接着是各种我们感兴趣的条件测试。

指定感兴趣的描述字

第 21～30 行　将两个描述字集都清零，然后在每个集合中最多打开 2 位。如果在标准输入上还没有读到一个文件结束符，而且在缓冲区中至少还有一个字节的空间可存放数据，读集合中与标准输入对应的位将被打开。如果在 fr 缓冲区中有至少存放一个字节数据的空间，读集合中与套接口对应的位将被打开。如果在 to 缓冲区中有要写入套接口的数据，写集合中与套接口对应的位将被打开，最后，如果在 fr 缓冲区中有发往标准输出的数据，写集合中与标准输出对应的位将被打开。

调用 select

第 31 行　调用 select，等待四个条件之一得到满足。我们没有为这个函数设置超时。

这个函数的下一部分在图 15.4 中给出。其中包含 select 返回后进行的前两个测试（总共 4 个）。

从标准输入 read

第 32～33 行　如果标准输入上有数据可读，调用 read。第三个参数是 to 缓冲区中可用的空间数。

处理非阻塞错误

第 34～35 行　如果发生错误，而且是 EWOULDBLOCK，就不作任何操作。一般这种条件“不应该发生”，这意味着，select 告诉我们该描述字可读但 read 却返回 EWOULDBLOCK，不过我们还是对这种情况进行了处理。

read返回文件结束符

第36～40行 如果read返回0，标准输入处理就结束。我们设置stdineof标志。如果在to缓冲区中没有数据待发送(tooptr等于toiptr)，就用shutdown向服务器发送FIN。如果在to缓冲区中仍有数据待发送，则要等到缓冲区写入套接口后再发送FIN。

我们输出一行到标准错误输出以表示文件结束条件，同时给出当前的时间，在介绍这个函数后我们会展示怎样使用这个输出信息。类似的fprintf贯穿这个函数的始终。

```
32      if (FD_ISSET(STDIN_FILENO, &rset)) {
33          if ( (n = read(STDIN_FILENO, toiptr, &to[MAXLINE] - toiptr)) < 0) {
34              if (errno != EWOULDBLOCK)
35                  err_sys("read error on stdin");
36          } else if (n == 0) {
37              fprintf(stderr, "%s: EOF on stdin\n", gf_time());
38              stdineof = 1;                   /* all done with stdin */
39              if (tooptr == toiptr)
40                  Shutdown(sockfd, SHUT_WR);/* send FIN */
41          } else {
42              fprintf(stderr, "%s: read %d bytes from stdin\n", gf_time(), n);
43              toiptr += n;                    /* # just read */
44              FD_SET(sockfd, &wset);  /* try and write to socket below */
45          }
46      }
47      if (FD_ISSET(sockfd, &rset)) {
48          if ( (n = read(sockfd, friptr, &fr[MAXLINE] - friptr)) < 0) {
49              if (errno != EWOULDBLOCK)
50                  err_sys("read error on socket");
51          } else if (n == 0) {
52              fprintf(stderr, "%s: EOF on socket\n", gf_time());
53              if (stdineof)
54                  return;     /* normal termination */
55              else
56                  err_quit("str_cli: server terminated prematurely");
57          } else {
58              fprintf(stderr, "%s: read %d bytes from socket\n",
59                      gf_time(), n);
60              friptr += n;    /* # just read */
61              FD_SET(STDOUT_FILENO, &wset); /* try and write below */
62          }
63      }
```

图15.4 str_cli函数：第二部分，从标准输入或套接口读入数据[nonblock/strclinonb.c]

read返回数据

第41～45行 当read返回数据时，我们相应地增加toiptr。另外还打开在写集合中与套接口对应的位，使后半部分的循环中对该位的测试能返回真，以试图在该套接口上write。

这是在编写代码时艰难的设计决断之一。这里有几种选择。我们可以用什么也不做来代替在写集合中设置对应的位。但是这样在已经知道有数据可向套接口写的情况下，还要进行另一次循环和select调用。另一个选择是把向套接

口写的代码复制到这儿，但这看上去有些浪费，而且是一个潜在的错误源（万一在复制的代码中存在缺陷，在一个位置修复了但另一个位置却没有）。最后，我们可以创建一个写套接口的函数，用调用这个函数来代替复制代码，但这个函数需要共享三个 str_cli 使用的局部变量，这足以要求这些变量成为全局变量。最终作出的选择是作者自认为最好的。

从套接口读

第 47～63 行　这些代码行和刚才介绍的当标准输入可读时的 if 语句类似。如果 read 返回 EWOULDBLOCK，则不做任何处理。如果遇到从服务器来的一个文件结束符，那么在我们已经遇到从标准输入来的文件结束符的情况下是没有问题的，但不期望是别的情况。如果 read 返回一些数据，我们就增加 friptr 并打开写集合中标准输出对应的位，以试图在该函数的下一部分中写数据。

图 15.5 给出了该函数的最后一部分。

```
64          if (FD_ISSET(STDOUT_FILENO, &wset) && ( (n = friptr - froptr) > 0)) {
65              if ( (nwritten = write(STDOUT_FILENO, froptr, n)) < 0) {
66                  if (errno != EWOULDBLOCK)
67                      err_sys("write error to stdout");
68              } else {
69                  fprintf(stderr, "%s: wrote %d bytes to stdout\n",
70                          gf_time(), nwritten);
71                  froptr += nwritten;             /* # just written */
72                  if (froptr == friptr)
73                      froptr = friptr = fr;   /* back to beginning of buffer */
74              }
75          }
76          if (FD_ISSET(sockfd, &wset) && ( (n = toiptr - tooptr) > 0)) {
77              if ( (nwritten = write(sockfd, tooptr, n)) < 0) {
78                  if (errno != EWOULDBLOCK)
79                      err_sys("write error to socket");
80              } else {
81                  fprintf(stderr, "%s: wrote %d bytes to socket\n",
82                          gf_time(), nwritten);
83                  tooptr += nwritten;     /* # just written */
84                  if (tooptr == toiptr) {
85                      toiptr = tooptr = to;   /* back to beginning of buffer */
86                      if (stdineof)
87                          Shutdown(sockfd, SHUT_WR);      /* send FIN */
88                  }
89              }
90          }
91      }
92  }
```

图 15.5　str_cli 函数：第三部分，写标准输出或套接口[nonblock/strclinonb.c]

写标准输出

第 64～67 行　如果标准输出可写，而且要写的字节数大于 0，就调用 write。如果返回 EWOULDBLOCK，则不做任何处理。注意这种情况完全可能发生，因为这个函数前一部分末尾的代码在不知道 write 是否能成功的条件下，打开了写集合中标准输出对应的位。

write 成功

第68～74行 如果write成功，froptr就加上写入的字节数。如果输出指针追上了输入指针，两个指针就都复位，指向缓冲区的开头。

写套接口

第76～90行 这一段代码和刚才介绍的写标准输出的代码类似。有一个不同是当输出指针追上输入指针时，不仅两个指针都复位到缓冲区的开头，而且如果在标准输入上遇到文件结束符，还向服务器发出FIN。

现在检查一下这个函数的操作和非阻塞I/O的重迭。图15.6给出了str_cli调用的gf_time函数。

```
 1 #include    "unp.h"
 2 #include    <time.h>

 3 char *
 4 gf_time(void)
 5 {
 6     struct timeval tv;
 7     static char str[30];
 8     char    *ptr;

 9     if (gettimeofday(&tv, NULL) < 0)
10         err_sys("gettimeofday error");

11     ptr = ctime(&tv.tv_sec);
12     strcpy(str, &ptr[11]);
13         /* Fri Sep 13 00:00:00 1986\n\0 */
14         /* 012345678901234567890123 4   */
15     snprintf(str+8, sizeof(str)-8, ".%06ld", tv.tv_usec);

16     return(str);
17 }
```

图15.6 gf_time函数：返回指向时间字符串的指针[lib/gf_time.c]

这个函数返回一个包含当前时间的字符串，包括微秒，格式如下

```
12:34:56.123456
```

这里特意和tcpdump输出的时间戳的格式一样。还要注意的是，str_cli函数中的所有fprintf调用都写到标准错误输出，允许我们把标准输出(服务器回射的行)和诊断输出分开。然后可以运行客户程序和tcpdump，并将这些诊断输出和tcpdump的输出放在一起以时间排序。这让我们能查看程序中到底发生了什么，并和对应的TCP动作联系起来。

举例来说，我们首先在主机solaris上运行tcpdump，只捕获去往或来自端口7(echo服务器)的TCP分节，输出信息存到名为tcpd的文件中。

```
solaris % tcpdump -w tcpd tcp and port 7
```

然后在这台主机上运行我们的TCP客户程序，指明服务器在主机bsdi上。

```
solaris % tcpcli02 206.62.226.35 < 2000.lines > out 2> diag
```

标准输入是文件2000.lines，跟图6.13中用的文件一样。标准输出定向到文件out，标准

错误输出定向到文件 diag。完成后我们运行

```
solaris % diff 2000.lines out
```

以验证回射的行与输入行相同。最后用中断键终止 tcpdump 并输出 tcpdump 记录，将这些记录和客户程序输出的诊断信息一起排序。图 15.7 给出了这个结果的第一部分。

```
solaris % tcpdump -r tcpd -N | sort diag -
10:18:34.486392 solaris.33621 > bsdi.echo: S 1802738644:1802738644(0)
                                            win 8760 <mss 1460>
10:18:34.488278 bsdi.echo > solaris.33621: S 3212986316:3212986316(0)
                                            ack 1802738645 win 8760<mss 1460>
10:18:34.488490 solaris.33621 > bsdi.echo: . ack 1 win 8760
10:18:34.491482: read 4096 bytes from stdin
10:18:34.518663 solaris.33621 > bsdi.echo: p 1:1461(1460) ack 1 win 8760
10:18:34.519016: wrote 4096 bytes to socket
10:18:34.528529 bsdi.echo > solaris.33621: p 1:1461(1460) ack 1461 win 8760
10:18:34.528785 solaris.33621 > bsdi.echo: . 1461:2921(1460) ack 1461 win 8760
10:18:34.528900 solaris.33621 > bsdi.echo: p 2921:4097(1176) ack 1461 win 8760
10:18:34.528958 solaris.33621 > bsdi.echo: . ack 1461 win 8760
10:18:34.536193 bsdi.echo > solaris.33621: . 1461:2921(1460) ack 4097 win 8760
10:18:34.536697 bsdi.echo > solaris.33621: P 2921:3509(588) ack 4097 win 8760
10:18:34.544636: read 4096 bytes from stdin
10:18:34.568505: read 3508 bytes from socket
10:18:34.580373 solaris.33621 > bsdi.echo: . ack 3509 win 8760
10:18:34.582244 bsdi.echo > solaris.33621: P 3509:4097(588) ack 4097 win 8760
10:18:34.593354: wrote 3508 bytes to stdout
10:18:34.617272 solaris.33621 > bsdi.echo: P 4097:5557(1460) ack 4097 win 8760
10:18:34.617610 solaris.33621 > bsdi.echo: P 5557:7017(1460) ack 4097 win 8760
10:18:34.617908 solaris.33621 > bsdi.echo: P 7017:8193(1176) ack 4097 win 8760
10:18:34.618062: wrote 4096 bytes to socket
10:18:34.623310 bsdi.echo > solaris.33621: . ack 8193 win 8760
10:18:34.626129 bsdi.echo > solaris.33621: . 4097:5557(1460) ack 8193 win 8760
10:18:34.626339 solaris.33621 > bsdi.echo: . ack 5557 win 8760
10:18:34.626611 bsdi.echo > solaris.33621: P 5557:6145(588) ack 8193 win 8760
10:18:34.628396 bsdi.echo > solaris.33621: . 6145:7605(1460) ack 8193 win 8760
10:18:34.643524: read 4096 bytes from stdin
10:18:34.667305: read 2636 bytes from socket
10:18:34.670324 solaris.33621 > bsdi.echo: . ack 7605 win 8760
10:18:34.672221 bsdi.echo > solaris.33621: P 7605:8193(588) ack 8193 win 8760
10:18:34.691039: wrote 2636 bytes to stdout
```

图 15.7　排序后的 tcpdump 输出和诊断输出

我们把那些包含 SYN 的过长的行折行了，还删掉了 Solaris 分节的(DF)记号，它表示设置了不分片位(路径 MTU 发现)。tcpdump 的-N 选项只输出全限定域名(solaris.kohala.com)中的主机部分(solaris)。

通过这个输出，我们可以把发生的事情以时间线图描绘出来。在图 15.8 中展示了这个结果，图中时间是向下递增的。

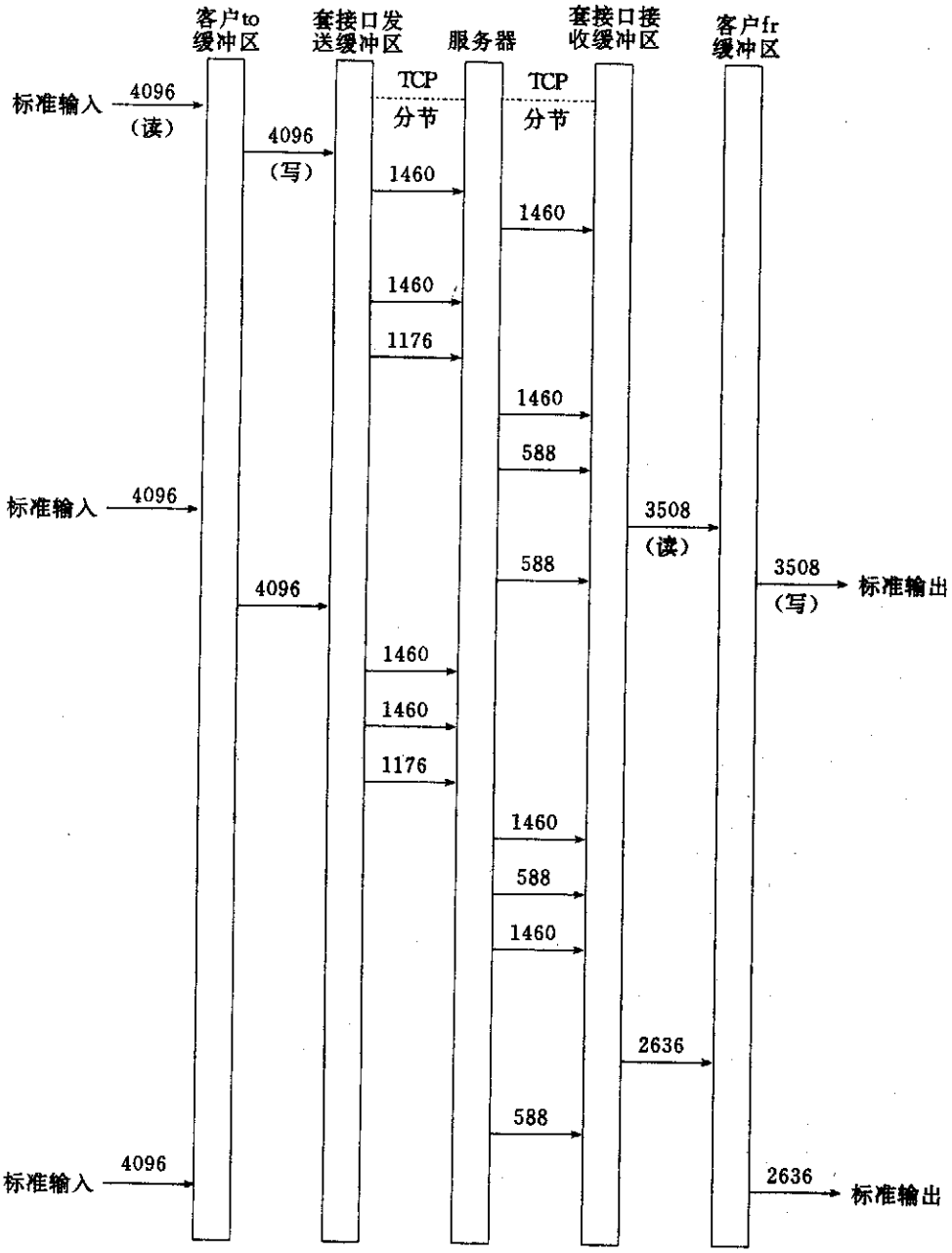


图 15.8 非阻塞实例的时间线

在上图中没有给出 ACK 分节。还要认识到当程序输出"wrote N bytes to stdout(已写N字节到标准输出)"时,write 已经返回了,可能导致 TCP 发送了一个或多个分节的数据。

从时间线我们可以看到客户和服务器之间的数据交换的动态性。用非阻塞 I/O 使程序利用了这些动态性的优势,在能做读或写操作时就做这些操作。用 select 函数让内核告诉我们什么时候能做 I/O 操作。

可以用 6.7 节中所用的同一个 2000 行的文件和同样的服务器主机(与客户主机的

RTT 为 175ms)来测算非阻塞版本运行所用的时间。和 6.7 节中的版本所花的 12.3 秒相比,非阻塞版本现在花了6.9秒。因此就本例子而言,非阻塞 I/O 减少了向服务器发送一个文件的全部时间。

str_cli 的更简单版本

刚才给出的 str_cli 的非阻塞版本是比较复杂的:大约有 135 行代码,与此相比,图 6.13 中使用 select 和阻塞 I/O 的版本是 40 行,最初的停-等版本(图 5.5)是 20 行。代码的长度从 20 行翻番到 40 行是值得的,因为在批操作模式下速度几乎提高了 30 倍,而且在阻塞的描述字上使用 select 并不太复杂。但在增加代码复杂度的情况下,是否值得用非阻塞 I/O 编写应用程序呢?回答是否定的。每当我们发现需要使用非阻塞 I/O 时,把应用程序分成多个进程(使用 fork)或线程(第 23 章)通常会更简单。

图 15.9 是 str_cli 函数的另一个版本,这个函数用 fork 把自己分成了两个进程。

```
1 #include    "unp.h"
2 void
3 str_cli(FILE *fp, int sockfd)
4 {
5     pid_t pid;
6     char   sendline[MAXLINE], recvline[MAXLINE];
7     if ( (pid = Fork()) == 0) {          /* child: server -> stdout */
8         while (Readline(sockfd, recvline, MAXLINE) > 0)
9             Fputs(recvline, stdout);
10        kill(getppid(), SIGTERM);       /* in case parent still running */
11        exit(0);
12    }
13    /* parent: stdin -> server */
14    while (Fgets(sendline, MAXLINE, fp) != NULL)
15        Writen(sockfd, sendline, strlen(sendline));
16    Shutdown(sockfd, SHUT_WR);      /* EOF on stdin, send FIN */
17    pause();
18    return;
19 }
```

图 15.9　str_cli 函数使用 fork 的版本[nonblock/strclifork.c]

这个函数一开始就调用 fork 分为父进程和子进程。子进程把从服务器来的行拷贝到标准输出,父进程把标准输入来的行拷贝到服务器,如图 15.10 所示。

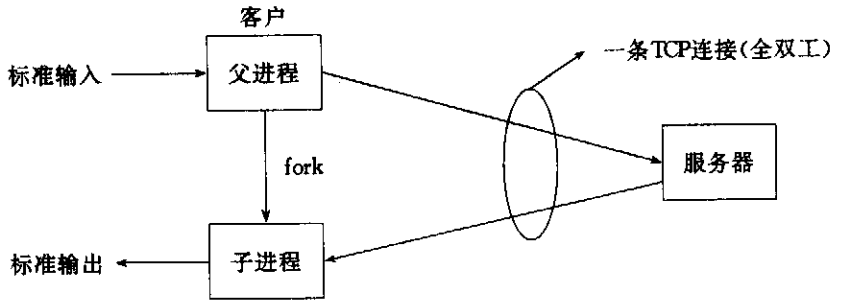


图 15.10　使用两个进程的 str_cli 函数

我们清楚地注意到这个TCP连接是全双工的，父进程和子进程共享同一个套接口：父进程向该套接口写，子进程从该套接口读。这里只有一个套接口、一个套接口接收缓冲区和一个套接口发送缓冲区，但这个套接口被两个描述字引用：一个在父进程，一个在子进程。

我们还要担心终止顺序的问题。通常终止是在标准输入上遇到一个文件结束符时发生。父进程读到这个文件结束符后调用shutdown发送一个FIN。(父进程不能调用close，参见习题15.1。)但这时，子进程需要继续把从服务器来的数据拷贝到标准输出，直到在套接口上读到一个文件结束符。

也有可能服务器进程过早终止(5.12节)，如果发生了这种情况，子进程将在套接口上读到一个文件结束符。如果这样子进程必须通知父进程停止从标准输入向套接口拷贝数据(见习题15.2)。在图15.9中，在父进程仍在运行的情况下，子进程向父进程发送SIGTERM信号(见习题15.3)。另一种方法是让子进程终止，从而使父进程获SIGCHLD。

父进程在完成拷贝后调用pause，进入睡眠状态直到收到信号。即使父进程不捕获任何信号，pause也使父进程进入睡眠状态，直到收到从子进程来的SIGTERM信号为止。这个信号的缺省动作是终止该进程，这在这个例子中是十分合适的。通常这个子进程在父进程之后结束，但由于我们用shell的time命令来计时，因此父进程终止时计时也结束。所以在这里让父进程等待子进程，以更准确地衡量str_cli的这个版本所使用的时间。

注意这个版本和这一节前面的非阻塞版本相比要简单得多。非阻塞的版本同时管理四个不同的I/O流，而且由于四个流都是非阻塞的，就必须考虑这四个流上的部分读和写的问题。但在这个fork版本中，每个进程只处理两个I/O流，从一个拷贝到另一个。因为如果输入流没有数据可读，对应的输出流就没有数据可写，所以不需要非阻塞I/O。

str_cli所用的时间

现在已经有str_cli函数的四种不同的版本。我们总结一下这些版本以及一个使用线程的版本(图23.2)在从一个Solaris 2.5客户主机向一个RTT为175毫秒的服务器主机拷贝2000行所花的时间。

- 354.0秒，停等方式(图5.5)，
- 12.3秒，使用select和阻塞I/O(图6.13)，
- 6.9秒，非阻塞I/O(图15.3)，
- 8.7秒，fork(图15.9)，
- 8.5秒，线程版本(图23.2)。

非阻塞版本的速度几乎是使用阻塞I/O和select版本的两倍那么快。使用fork的简单版本比非阻塞版本慢，然而，考虑到非阻塞版本代码的复杂性，我们推荐简单的版本。

15.3 非阻塞connect

在一个TCP套接口被设置为非阻塞后调用connect，connect会立即返回一个EINPROGRESS错误，但TCP的三路握手继续进行。在这之后我们可以用select检查这个连接是否建立成功。非阻塞的connect有三种用途。

1. 我们可以在三路握手的同时做一些其他的处理。connect 要花一个往返时间完成(2.5 节),而且可以是在任何地方,从几个毫秒的局域网到几百毫秒或几秒的广域网。在这段时间内我们可能有一些其他的处理想要执行。
2. 可以用这种技术同时建立多个连接。这在 Web 浏览器中很普遍,在 15.5 节中有这样的一个例子。
3. 由于我们用 select 等待连接的完成,因此可以给 select 设置一个时间限制,从而缩短 connect 的超时时间。在多数实现中,connect 的超时时间在 75 秒到几分钟之间。有时应用程序想要一个更短的超时时间,使用非阻塞 connect 就是一种方法。13.2 节中谈到了在套接口操作中设置超时的其他方法。

非阻塞 connnct 听起来虽然简单,有一些细节问题我们还得处理。

- 即使套接口是非阻塞的,如果连接的服务器在同一台主机上,在调用 connect 时连接通常会立刻建立。我们必须处理这种情况。
- 源自 Berkeley 的实现(和 Posix.1g)有两条与 select 和非阻塞 I/O 相关的规则:(1)当连接成功建立时,描述字变成可写(TCPv2 第 531 页),(2)当连接建立出错时,描述字变成既可读又可写(TCPv2 第 530 页)。

> 这两条和 select 相关的规则是从 6.3 节中使描述字就绪的相关规则而来。TCP 套接口在它的发送缓冲区有空闲空间(对于连接中的套接口这个条件总是成立的,因为还没有向它写任何数据),而且套接口已连接(当三路握手完成时才成立)时是可写的。待处理错误则导致套接口变成既可读又可写。

在下面的例子中提到了许多有关 connect 的移植性问题。

15.4　非阻塞 connect:时间/日期客户程序

图 15.11 给出了 connect_nonb 函数,它执行一个非阻塞 connect。我们把图 1.5 的 connect 替换成:

```
if (connect_nonb(sockfd, (SA *) &servaddr, sizeof(servaddr), 0) < 0)
    err_sys("connect error");
```

前三个参数和 connect 的一样,第四个参数是等待连接完成的秒数。它的值为 0 表示 select 没有超时时间;因此内核将使用通常的 TCP 连接建立超时时间。

设置非阻塞套接口

第 9～10 行　调用 fcntl 设置非阻塞套接口。

第 11～14 行　开始非阻塞 connect。我们期望的错误是 EINPROGRESS,表示开始建立连接但还没有完成(TCPv2 第 466 页),其他的错误将被返回给调用者。

在建立连接时进行其他的处理

第 15 行　在等待连接建立完成时可以做任何我们想做的事情。

检查连接是否立即建立

第 16～17 行　如果非阻塞 connect 返回 0,连接已建立。前面已经提到,这种情况在服

务器和客户在同一台主机上时可能发生。

```
1 #include    "unp.h"
2 int
3 connect_nonb(int sockfd, const SA *saptr, socklen_t salen, int nsec)
4 {
5     int     flags, n, error;
6     socklen_t len;
7     fd_set  rset, wset;
8     struct timeval tval;
9     flags = Fcntl(sockfd, F_GETFL, 0);
10    Fcntl(sockfd, F_SETFL, flags | O_NONBLOCK);
11    error = 0;
12    if ( (n = connect(sockfd, saptr, salen)) < 0)
13        if (errno != EINPROGRESS)
14            return(-1);
15    /* Do whatever we want while the connect is taking place. */
16    if (n == 0)
17        goto done;  /* connect completed immediately */
18    FD_ZERO(&rset);
19    FD_SET(sockfd, &rset);
20    wset = rset;
21    tval.tv_sec = nsec;
22    tval.tv_usec = 0;
23    if ( (n = Select(sockfd+1, &rset, &wset, NULL,
24                     nsec ? &tval : NULL)) == 0) {
25        close(sockfd);          /* timeout */
26        errno = ETIMEDOUT;
27        return(-1);
28    }
29    if (FD_ISSET(sockfd, &rset) || FD_ISSET(sockfd, &wset)) {
30        len = sizeof(error);
31        if (getsockopt(sockfd, SOL_SOCKET, SO_ERROR, &error, &len) < 0)
32            return(-1);             /* Solaris pending error */
33    } else
34        err_quit("select error: sockfd not set");
35 done:
36    Fcntl(sockfd, F_SETFL, flags);     /* restore file status flags */
37    if (error) {
38        close(sockfd);          /* just in case */
39        errno = error;
40        return(-1);
41    }
42    return(0);
43 }
```

图 15.11 启动一个非阻塞 connect[lib/connect_nonb.c]

调用 select

第 18～24 行　调用 select 等待套接口变成可读或可写。把 rset 清零，打开该描述字集中对应 sockfd 的位，然后把 rset 拷贝到 wset。这个赋值可能是一个结构的赋值，因为描述字集通常表示为结构。我们还初始化 timeval 结构并调用 select。如果调用者将第四个参数指定为 0(使用缺省的超时时间)，我们就必须把 select 的最后一个参数指定为一个空指针而不是一个值为 0 的 timeval 结构(后者意味着根本不等待)。

处理超时

第 25～28 行　如果 select 返回 0，那么时钟超时，于是返回 ETIMEOUT 错误给调用者。我们还关闭套接口，以防止三路握手继续下去。

检查可读或可写条件

第 29～34 行　如果描述字可读或可写，我们调用 getsockopt 得到套接口上待处理的错误(SO_ERROR)。如果连接建立成功，这个值将为 0。如果建立连接时遇到错误，这个值是连接错误对应的 errno 值(譬如 ECONNREFUSED、ETIMEDOUT 等)。这里还遇到第一个可移植性问题。如果发生错误，getsockopt 源自 Berkeley 的实现返回 0，待处理的错误在变量 error 中返回。但 Solaris 让 getsockopt 返回－1，errno 置为待处理的错误。我们的程序对两种情况都处理。

关闭非阻塞方式并返回

第 36～42 行　恢复文件状态标志并返回。如果 getsockopt 返回的 error 变量不为 0，就将这个值存到 errno，函数返回－1。

就像前面所说，非阻塞 connect 在不同的套接口实现中存在移植性问题。首先，有可能在调用 select 之前连接已建立而且对方的数据已到来。在这种情况下，连接成功时套接口将既可读又可写，和连接失败时一样。图 15.11 中的代码通过调用 getsockopt 并检查套接口上待处理的错误来处理这种情况。

如果我们不能假定返回可写是成功的唯一一种情况，下一个移植性问题就是怎样判断连接是否成功建立。在 Usenet 上曾发表过各种解决方法。这些方法可以代替图 15.11 中的 getsockopt 调用。

1. 调用 getpeername 代替 getsockopt。如果这个调用失败，返回 ENOTCONN，表示连接失败，我们必须以 SO_ERROR 调用 getsockopt 得到套接口上待处理的错误。
2. 调用 read，长度参数为 0。如果 read 失败，表示 connect 失败，而且 read 返回的 errno 指明了连接失败的原因。如果连接成功，read 应返回 0。
3. 再调用 connect 一次。它应该失败，如果错误是 EISCONN，那么套接口已经连接而且第一次连接是成功的。

不幸的是非阻塞 connect 是网络编程中最不好移植的部分。要准备应付移植性问题，特别是对那些老的实现，一种简单的技术是创建一个线程(第 23 章)来处理这个连接。

被中断的 connect

如果在一个阻塞式套接口上调用的 connect，在 TCP 的三路握手完成前被中断，譬如说被捕获的信号中断，将会发生什么呢？假定 connect 不被自动重启，它会返回 EINTR。但是

我们不能再调用 connect 等待连接建立完成，这样做将返回 EADDRINUSE。

在这种情况下应该做的是调用 select，就和这节中对非阻塞 connect 所做的一样。然后 select 在连接建立成功(使套接口可写)或连接失败(使套接口既可读又可写)时返回。

> Posix.1g 明确地说明了调用 XTI 函数 t_connect 被信号中断时做些什么，但没有说明对 connect 被中断如何处理。上面介绍的是源自 Berkeley 的内核对于这种情况的处理。

15.5 非阻塞 connect：Web 客户程序

非阻塞 connect 的现实例子是从 Netscape 的 Web 客户程序(TCPv3 的 13.4 节)开始的。这个客户通过和 Web 服务器建立 HTTP 连接以获取主页。在页面中通常有多个引用指向其他 Web 页面。客户可以使用非阻塞 connect 同时取多个页面，以此代替每次只取一个页面。图 15.12 展示了一个并行建立多个连接的例子。最左边显示了顺序操作全部三个连接的的情况。假定第一个连接用 10 个时间单位，第二个用 15 个，第三个用 4 个，总计 29 个时间单位。

中间是并行操作两个连接的情况。在时间为 0 时开始前两个连接，当其中之一结束时，开始第三个连接。总共的时间差不多减半，从 29 到 15，但这只是理想的情况。如果并行进行的连接共享同一条链路(譬如客户通过一个拨号调制解调器连到因特网)，那每一个连接之间相互竞争有限的资源，各个连接所花的时间可能会变得更长一些。举例来说，花 10 个时间单位可能会变成 15，15 可能变成 20，4 可能变成 6。虽然如此，总共的时间为 21，仍然比顺序的情况要短。

第三种情况并行操作三个连接，而且再次假设这三个连接之间没有冲突(理想情况)。但在这个例子中总共所花的时间和第二种情况一样(15 个时间单位)。

在处理 Web 客户时，第一个连接是它自己完成的，接着是多个连接，连往从第一个连接的数据中找到的引用。在图 15.13 中展示了这一点。

在进一步优化时，客户可以在第一个连接完成前开始分析返回的数据，并按需要随时进行附加的连接。

由于要同时操作多个非阻塞连接，我们不能使用图 15.11 中的 connect_nonb 函数，因为它在连接建立前不返回。我们必须自己管理这些连接。

我们的程序最多会从 Web 服务器读 20 个文件。最大并行连接数、服务器的主机名和每个从服务器取的文件名作为命令行参数。程序运行的一个典型例子如下：

```
solaris % web 3 www.foobar.com / image1.gif image2.gif \
image3.gif image4.gif image5.gif \
image6.gif image7.gif
```

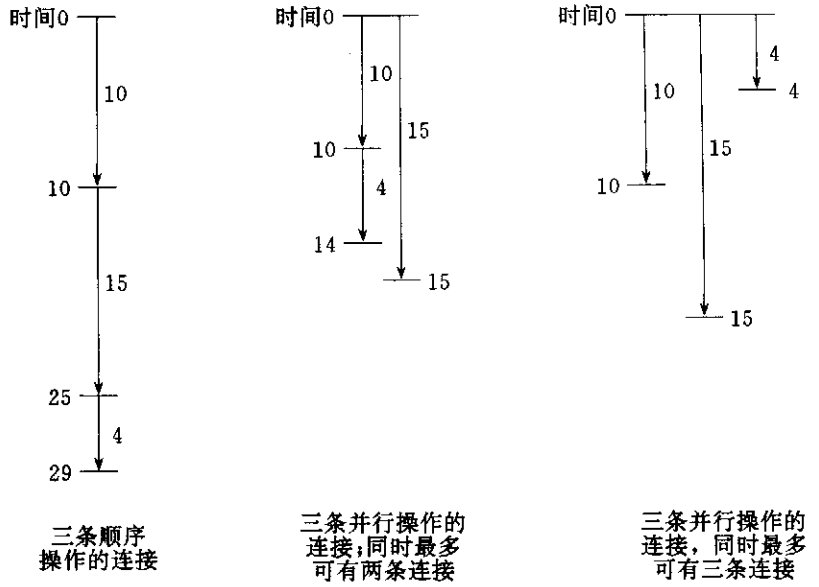


图 15.12　并行操作多个连接

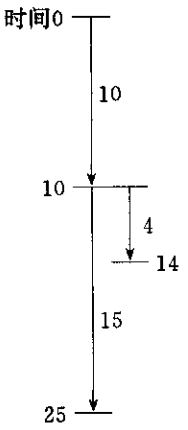


图 15.13　完成第一个连接，然后并行操作多个连接

命令行参数指定最多三个连接、服务器的主机名、主页的文件名(/是服务器的根页面)和随后要读的七个文件(在这个例子中都是 GIF 图像)。这七个文件通常是被主页引用的，Web 客户会读入主页并分析 HTML 以获得这些文件名。我们不想在这个例子中加入 HTML 分析使其复杂化，因此在命令行上指定文件名。

这个例子较大，因此我们把它分成几个部分给出。图 15.14 是每个文件都包括的 web.h 头文件。

```
1 #include    "unp.h"
2 #define MAXFILES    20
3 #define SERV        "80"        /* port number or service name */
4 struct file {
```

```
 5   char        * f_name;              /* filename */
 6   char        * f_host;              /* hostname or IPv4/IPv6 address */
 7   int     f_fd;                      /* descriptor */
 8   int   f_flags;                     /* F_xxx below */
 9 } file[MAXFILES];

10 #define F_CONNECTING    1           /* connect() in progress */
11 #define F_READING       2           /* connect() complete; now reading */
12 #define F_DONE          4           /* all done */

13 #define GET_CMD             "GET %s HTTP/1.0\r\n\r\n"

14                     /* globals */
15 int          nconn, nfiles, nlefttoconn, nlefttoread, maxfd;
16 fd_set       rset, wset;

17                     /* function prototypes */
18 void   home_page(const char *, const char *);
19 void   start_connect(struct file *);
20 void   write_get_cmd(struct file *);
```

图15.14 web.h头文件[nonblock/web.h]

定义file结构

第2～13行 本程序从Web服务器读最多MAXFILES个文件。我们维护一个file结构,其中包含关于每个文件的信息:它的名字(从命令行参数复制而来)、文件所在的服务器的主机名或IP地址、取这个文件用的套接口描述字以及一个表明正在对这个文件做什么(连接、读或完成)的标志集。

定义全局变量和函数原型。

第14～20行 定义全局变量和马上要叙述的函数原型。

图15.15给出了main函数的第一部分。

```
 1 #include    "web.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int           i, fd, n, maxnconn, flags, error;
 6     char    buf[MAXLINE];
 7     fd_set        rs, ws;

 8     if (argc < 5)
 9         err_quit("usage: web <#conns> <hostname> <homepage> <file1>
           ...");
10     maxnconn = atoi(argv[1]);

11     nfiles = min(argc - 4, MAXFILES);
12     for (i = 0; i < nfiles; i++) {
13         file[i].f_name = argv[i + 4];
14         file[i].f_host = argv[2];
15         file[i].f_flags = 0;
16     }
17     printf("nfiles = %d\n", nfiles);

18     home_page(argv[2], argv[3]);
```

```
19      FD_ZERO(&rset);
20      FD_ZERO(&wset);
21      maxfd = -1;
22      nlefttoread = nlefttoconn = nfiles;
23      nconn = 0;
```

图 15.15　同时 connect 的第一部分：全局变量和 main 的开头[nonblock/web.c]

处理命令行参数

第 11～17 行　用从命令行参数来的相关信息填写 file 结构。

读主页

第 18 行　下面给出的 home_page 函数创建一个 TCP 连接，向服务器发出一个命令，然后读主页。这是第一个连接，在开始并行建立多条连接之前自己完成。

初始化全局变量

第 19～23 行　初始化两个描述字集合，一个读一个写。maxfd 是 select 使用的最大描述字(我们把它初始化成－1，因为描述字是非负的)，nlefttoread 是剩余要读的文件数(当它到 0 时程序就结束)，nlefttoconn 是还需要一个 TCP 连接的文件数，nconn 是当前打开的连接数(它不能超过第一个命令行参数)。

图 15.16 给出了在 main 函数开始时调用了一次的 home_page 函数。

```
1  #include    "web.h"
2  void
3  home_page(const char *host, const char *fname)
4  {
5      int          fd, n;
6      char    line[MAXLINE];
7      fd = Tcp_connect(host, SERV); /* blocking connect() */
8      n = snprintf(line, sizeof(line), GET_CMD, fname);
9      Writen(fd, line, n);
10     for ( ; ; ) {
11         if ( (n = Read(fd, line, MAXLINE)) == 0)
12             break;              /* server closed connection */
13         printf("read %d bytes of home page\n", n);
14         /* do whatever with data */
15     }
16     printf("end-of-file on home page\n");
17     Close(fd);
18 }
```

图 15.16　home_page 函数[nonblock/home_page.c]

和服务器建立连接

第 7 行　tcp_connect 和服务器建立一个连接。

向服务器发 HTTP 命令；读应答

第 8～17 行　发出一个 HTTP GET 命令以获取主页(一般为/)。读应答数据(我们不对这些数据做任何处理)，然后关闭连接。

图15.17中给出的下一个函数start_connect启动非阻塞connect。

创建套接口，设置非阻塞方式

第7～13行 调用host_serv函数(图11.5)查找和转换主机名和服务名，返回一个指向addrinfo结构的数组的指针。我们只用第一个结构。创建一个TCP套接口并将其设置为非阻塞方式。

```
1 #include    "web.h"
2 void
3 start_connect(struct file * fptr)
4 {
5     int     fd, flags, n;
6     struct addrinfo * ai;
7     ai = Host_serv(fptr->f_host, SERV, 0, SOCK_STREAM);
8     fd = Socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
9     fptr->f_fd = fd;
10    printf("start_connect for %s, fd %d\n", fptr->f_name, fd);
11        /* Set socket nonblocking */
12    flags = Fcntl(fd, F_GETFL, 0);
13    Fcntl(fd, F_SETFL, flags | O_NONBLOCK);
14        /* Initiate nonblocking connect to the server. */
15    if ( (n = connect(fd, ai->ai_addr, ai->ai_addrlen)) < 0) {
16        if (errno != EINPROGRESS)
17            err_sys("nonblocking connect error");
18        fptr->f_flags = F_CONNECTING;
19        FD_SET(fd, &rset);          /* select for reading and writing */
20        FD_SET(fd, &wset);
21        if (fd > maxfd)
22            maxfd = fd;
23    } else if (n >= 0)              /* connect is already done */
24        write_get_cmd(fptr);        /* write() the GET command */
25 }
```

图15.17 启动非阻塞connect[nonblock/start_connect.c]

启动非阻塞connect

第14～22行 启动非阻塞connect，并将相应文件的标志置为F_CONNECTING。在读集合和写集合中都打开这个套接口描述字对应的位，因此select将等待任何一个表示连接已完成的条件成立。我们还根据需要更新maxfd的值。

处理连接的完成

第23～24行 如果connect返回成功，那么连接已经完成，write_get_cmd函数(下面给出)向服务器发出一个命令。

我们把connect所用的套接口设置为非阻塞，但永远不恢复它缺省的阻塞模式。这样做没有问题是因为只向该套接口写了少量的数据(下一个函数中的GET命令)，从而可以认为这个命令大大小于套接口的发送缓冲区。即使write因为非阻塞标志造成返回的数目小于要写的数目，writen函数也会对此进行处理。让这个套接口非阻塞对后面的read没有什

么影响，因为我们总是调用 select 等待它变成可读。

图 15.18 给出了函数 write_get_cmd，它向服务器发出一个 HTTP 的 GET 命令。

```
1 #include    "web.h"
2 void
3 write_get_cmd(struct file * fptr)
4 {
5     int     n;
6     char    line[MAXLINE];
7     n = snprintf(line, sizeof(line), GET_CMD, fptr->f_name);
8     Writen(fptr->f_fd, line, n);
9     printf("wrote %d bytes for %s\n", n, fptr->f_name);
10    fptr->f_flags = F_READING;          /* clears F_CONNECTING */
11    FD_SET(fptr->f_fd, &rset);          /* will read server's reply */
12    if (fptr->f_fd > maxfd)
13        maxfd = fptr->f_fd;
14 }
```

图 15.18　向服务器发出一个 HTTP 的 GET 命令[nonblock/write_get_cmd.c]

构造命令并发出

第 7～9 行　构造命令并写入套接口。

设置标志

第 10～23 行　设置相应文件的 F_READING 标志，它同时清除 F_CONNECTING 标志(如果设置了的话)。这在主循环中表示该描述字已经就绪可以输入了。这个描述字也在读集合中打开，并根据需要更新 maxfd。

现在回到图 15.19 中的 main 函数，继续在图 15.15 中剩下的内容。这是程序的主循环：只要还有文件要处理(nlefttoread 大于 0)，可能的话就启动另一个连接，然后在所有活动的描述字上 select，处理非阻塞连接的完成和到来的数据。

能开始另一个连接吗?

第 24～35 行　如果没有达到同时连接数的上限，而且有另外的连接要建立，就寻找一个还没有处理的文件(用 f_flags 为 0 来表示)，调用 start_connect 启动相应的连接。活动的连接数(nconn)增 1，剩下要建立的连接数(nlefttoconn)减 1。

select：等待事件发生

第 36～37 行　select 等待可读或可写条件。正在进行非阻塞 connect 的描述字在这两个集合中都打开，而已完成连接并正在等待来自服务器的数据的描述字则只在读集合中打开。

```
24    while (nlefttoread > 0) {
25        while (nconn < maxnconn && nlefttoconn > 0) {
26                /* find a file to read */
27            for (i = 0 ; i < nfiles; i++)
28                if (file[i].f_flags == 0)
29                    break;
30            if (i == nfiles)
31                err_quit("nlefttoconn = %d but nothing found", nlefttoconn);
32            start_connect(&file[i]);
```

```
33              nconn++;
34              nlefttoconn--;
35          }
36          rs = rset;
37          ws = wset;
38          n = Select(maxfd+1, &rs, &ws, NULL, NULL);
39          for (i = 0; i < nfiles; i++) {
40              flags = file[i].f_flags;
41              if (flags == 0 || flags & F_DONE)
42                  continue;
43              fd = file[i].f_fd;
44              if (flags & F_CONNECTING &&
45                  (FD_ISSET(fd, &rs) || FD_ISSET(fd, &ws))) {
46                  n = sizeof(error);
47                  if (getsockopt(fd, SOL_SOCKET, SO_ERROR, &error, &n) < 0 ||
48                      error != 0) {
49                      err_ret("nonblocking connect failed for %s",
50                              file[i].f_name);
51                  }
52                      /* connection established */
53                  printf("connection established for %s\n", file[i].f_name);
54                  FD_CLR(fd, &wset);          /* no more writeability test */
55                  write_get_cmd(&file[i]);/* write() the GET command */
56              } else if (flags & F_READING && FD_ISSET(fd, &rs)) {
57                  if ( (n = Read(fd, buf, sizeof(buf))) == 0) {
58                      printf("end-of-file on %s\n", file[i].f_name);
59                      Close(fd);
60                      file[i].f_flags = F_DONE;     /* clears F_READING */
61                      FD_CLR(fd, &rset);
62                      nconn--;
63                      nlefttoread--;
64                  } else {
65                      printf("read %d bytes from %s\n", n, file[i].f_name);
66                  }
67              }
68          }
69      }
70      exit(0);
71  }
```

图 15.19 main 函数的主循环[nonblock/web.c]

处理所有就绪的描述字

第 39～55 行 现在对 file 结构的数组中的每个元素进行处理，以判断哪个描述字需要处理。如果 F_CONNECT 标志被设置，而且该描述字在读或写集合中被置位，非阻塞 connect 已完成。如我们在图 15.11 中介绍的，调用 getsockopt 获取该套接口上待处理的错误。如果这个值为 0，连接已是成功完成。在这种情况下在写集合中关闭该描述字，调用 write_get_cmd 向服务器发送 HTTP 请求。

检查描述字是否有数据

第 56～67 行　如果 F_READING 标志被设置而且描述字已读就绪，则调用 read。如果连接被另一端关闭，我们就关闭连接，设置 F_DONE 标志，在读集合中关闭该描述字，活动连接的数目和要处理的连接总数都减 1。

在这个例子中有两点没有优化(以避免使它变得更复杂)。首先，在图 15.19 中处理完 select 返回的已就绪的描述字数后可以终止 for 循环。其次，可以尽可能地减少 maxfd 的值，省下 select 检查那些不再设置的描述字的时间。因为这段代码处理的描述字的数目在任何时候都小于 10，而不是成千上万，所以这些优化与附加的复杂性相比是否值得，令人怀疑。

同时连接的性能

同时建立多个连接的性能如何呢？图 15.20 给出了取一个 Web 服务器上的主页和该服务器上的 9 个图像文件所花的时间。这个服务器的 RTT 大约为 150ms。主页的大小为 4017 字节，9 个图像文件的平均大小为 1621 字节。TCP 的分节大小是 512 字节。为了便于比较，本图还包含了在 23.9 节中开发的这个程序的使用线程的版本的数据。

同时 连接数	时钟时间 (秒数)，非阻塞	时钟时间 (秒数)，线程
1	6.0	6.3
2	4.1	4.2
3	3.0	3.1
4	2.8	3.0
5	2.5	2.7
6	2.4	2.5
7	2.3	2.3
8	2.2	2.3
9	2.0	2.2

图 15.20　不同数目的同时连接所花的时间

大部分的性能提高是在同时三个连接时获得的(时间减半)，在四个或更多的连接时性能的增加很少。

> 我们提供这个使用同时连接的例子是因为它是一个使用非阻塞 I/O 的好例子，而且可以衡量对性能的影响。这也是一个流行的 Web 应用程序即 Netscape 浏览器使用的一个特性。网络上有拥塞时这种技术也有缺陷。TCPv1 的第 21 章介绍了 TCP 的慢启动和拥塞避免算法的细节。当客户向服务器建立多个连接时，各个连接之间在 TCP 层没有通信。也就是说，如果在一个连接上遇到一个分组丢失，连往同一个服务器的其他连接不会被通知，而且这些连接很可能马上遇到分组丢失，除非它们慢下来。这些连接会向一个已经拥塞的网络继续发送分组。这种技术还会增加服务器主机的负载。

15.6 非阻塞 accept

在第6章提到过当一个已完成的连接准备好被 accept 时，select 会把监听返回成可读。因此，如果用 select 等待外来的连接，应该不需要把监听套接口设置为非阻塞，因为如果 select 告诉我们连接已就绪，accept 就不应该阻塞。

不幸的是，这里存在一个时间问题[Gierth 1996]。为了观察这个问题，我们修改了 TCP 回射客户程序(图5.4)，在跟服务器建立连接后发送一个 RST。图15.21给出了这个新版本。

设置 SO_LINGER 套接口选项

第16～19行 一旦连接建立，我们设置 SO_LINGER 选项，把 l_onoff 标志置为1，l_linger 时间置为0。如7.5节中说明的，这会导致在关闭连接时在 TCP 套接口上发送一个 RST。我们随后关闭该套接口。

下一步，修改图6.21和6.22中的 TCP 服务器程序，在 select 返回监听套接口可读之后、调用 accept 之前暂停。在下面这段来自图6.22开头的代码中，前面有加号的两行是新加的。

```
        if(FD_ISSET(listenfd,&rset)) {      /* new client connection */
+           printf("listening socket readable\n");
+           sleep(5);
            clilen = sizeof(cliaddr);
            connfd = Accept(listenfd,(SA *) &cliaddr,&clien);
```

这里模拟的是一个繁忙的服务器，它不能做到在 select 一返回监听套接口可读时就调用 accpet。服务器的这种延迟一般情况下不会有问题(实际上这是为什么维护一个已完成连接队列的原因)，但是如果在连接建立后又有从客户来的 RST，就会出现问题。

在5.11节中我们注意到，在服务器调用 accept 之前客户如果放弃这个连接，源自 Berkeley 的实现不对服务器返回这个夭折的连接，而其他实现应返回 ECONNABORTED 错误，但一般返回 EPROTO 错误来代替。考虑一个源自 Berkeley 的实现。

```
1  #include    "unp.h"
2  int
3  main(int argc, char **argv)
4  {
5      int     sockfd;
6      struct linger ling;
7      struct sockaddr_in servaddr;
8      if (argc != 2)
9          err_quit("usage: tcpcli <IPaddress>");
10     sockfd = Socket(AF_INET, SOCK_STREAM, 0);
11     bzero(&servaddr, sizeof(servaddr));
12     servaddr.sin_family = AF_INET;
13     servaddr.sin_port = htons(SERV_PORT);
14     Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
```

```
15      Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
16      ling.l_onoff = 1;          /* cause RST to be sent on close() */
17      ling.l_linger = 0;
18      Setsockopt(sockfd, SOL_SOCKET, SO_LINGER, &ling, sizeof(ling));
19      Close(sockfd);
20      exit(0);
21  }
```

图 15.21　创建连接并发送 RST 的 TCP 回射客户程序[nonblock/tcpcli03.c]

- 客户如图 15.21 中那样建立连接然后放弃它。
- select 向服务器进程返回可读，但服务器要在一段时间后才能调用 accept。
- 在服务器从 select 返回和调用 accept 之间，收到从客户来的 RST。
- 这个已完成的连接从队列中删除，我们假设没有其他已完成的连接存在。
- 服务器调用 accept，但是由于没有已完成的连接，因而阻塞了。

服务器会一直阻塞在 accept 调用上，直到其他某个客户建立一个连接为止。但是在这期间，假定这个服务器程序与图 6.22 类似，服务器仍会阻塞在 accept 调用上，不处理任何其他已就绪的描述字。

> 这个问题和 6.8 节中介绍的拒绝服务攻击有几分相似，但存在这个缺陷的服务器在另一个客户建立连接时会打破这个阻塞的 accept。

解决这个问题的办法是：

1. 如果用 select 来获知何时有连接已就绪可以 accept 时，总是把监听套接口置为非阻塞，并且
2. 在后面的 accept 调用中忽略以下错误：EWOULDBLOCK（源自 Berkeley 的实现在客户放弃连接时出现的错误）、ECONNABORTED（Posix.1g 的实现在客户放弃连接时出现的错误）、EPROTO（SVR4 的实现在客户放弃连接时出现的错误）和 EINTR（如果信号被捕获）。

15.7　小　结

15.2 节中非阻塞读和写的例子是把 str_cli 回射客户程序改成在到服务器的 TCP 连接上使用非阻塞 I/O。select 一般和非阻塞 I/O 一起使用，以判断描述字何时可以读或写。虽然代码的修改比较复杂，但是这个版本的客户程序是我们列出的版本中最快的。在这之后我们展示了把这个客户程序用 fork 分成两部分会更简单，在图 23.2 的线程版本中使用了同样的技术。

非阻塞 connect 使我们在 TCP 的三路握手时可以做其他处理，而不是在 connect 上阻塞。不幸的是这些也是不可移植的，在不同的实现中表示连接成功完成或出错的方式也不一样。我们用非阻塞 connect 开发了一个新的客户程序，它和 Web 客户程序类似，同时打开多个连接以减少从服务器上取多个文件的时间。像这样发起多个连接可以减少时间，但考虑到避免 TCP 的拥塞，它是“网络不友好的”。

15.8 习 题

15.1 在对图15.9的讨论中我们提到父进程必须调用shutdown，而不是close。这是为什么？

15.2 图15.9中如果服务器进程过早终止，客户子进程收到文件结束符并终止，但子进程不通知父进程，会发生什么？

15.3 图15.9中如果父进程在子进程前意外死亡，然后子进程在套接口上读到文件结束符，会发生什么？

15.4 在图15.11中如果删掉以下两行会发生什么？

```
if (n == 0)
    goto done      /* connect complete immediately */
```

15.5 在15.3节中提到有可能在connect返回前在套接口上有数据到来。这是怎样发生的？

第 16 章　ioctl 操作

16.1　概　述

在传统上 ioctl 函数是用于那些普遍使用、但不适合归入其他类别的任何特性的系统接口。Posix 去掉了 ioctl，它通过创建特殊的其功能已被 Posix 标准化的包裹函数来代替 ioctl。举例来说，Unix 的终端接口一般用 iotcl 来访问，但 Posix.1 为终端创建了 12 个新的函数：tcgetattr 取终端属性，tcflush 刷新输入和输出，等等。与此类似，Posix.1g 替换了一个 ioctl 请求（以后简称 XXX ioctl）：新的 sockatmark 函数（21.3 节）代替了 SIOCATMARK ioctl。虽然如此，在网络编程方面一些依赖于具体实现的功能仍保留了不少 ioctl 请求：譬如获得接口信息、访问路由表和 ARP 高速缓存。

这一章对和网络编程有关的 ioctl 请求做一个概述，但其中有很多都依赖于具体的实现。另外，新的源自 Berkeley 的实现用 AF_ROUTE 域的套接口（路由套接口）来完成这些操作中的大部分。在第 17 章中会详细介绍路由套接口。

网络程序（一般是服务器程序）中 ioctl 常用于在程序启动时获得主机上所有接口的信息：接口的地址、接口是否支持广播、是否支持多播，等等。我们开发自己的函数返回这些信息，在本章中提供一个使用 ioctl 的实现，在 17 章中再提供一个使用路由套接口的实现。

16.2　ioctl 函数

这个函数影响由 fd 参数指向的打开的文件。

```
#include <unistd.h>
int ioctl(int fd,int request,... /* void *arg */);
                                    返回：成功返回 0，出错返回 -1
```

第三个参数总是一个指针，但指针的类型依赖于 request（请求）。

4.4BSD 把第二个参数定义为 unsigned long 而不是 int，但这不会有问题，因为头文件定义了用作这个参数的常值。

一些实现中把第三个参数指定为 void * 指针，取代了 ANSI C 中的省略号表示法。

包含 ioctl 的函数原型的头文件没有标准，因为 Posix 没有对此进行标准化。多数系统如上所示地在 <unistd.h> 中定义，但老的 BSD 系统在 <sys/ioctl.h> 中定义。

我们可以把和网络有关的请求分为 6 类。

- 套接口操作
- 文件操作
- 接口操作
- ARP 高速缓存操作
- 路由表操作
- 流系统(第 33 章)

回想图 7.15,不但有一些 ioctl 操作和 fcntl 操作功能重迭(譬如把套接口设置为非阻塞),而且也有一些操作可以用不止一种 ioctl 来完成(譬如设置套接口的进程组属主关系)。

图 16.1 列出了各种请求以及 arg 地址必须指向的数据类型。下面几节对这些请求作详细说明。

16.3 套接口操作

有三种 ioctl 请求是明确针对套接口的(TCPv2 第 551～553 页)。它们都要求 ioctl 的第三个参数是一个指向整数的指针。

SIOCATMARK 如果套接口的读指针当前在带外标志上,则通过第三个参数指向的整数返回一个非零值,否则返回零。在第 21 章中会对带外数据作详细介绍。Posix.1g 用 sockatmark 代替了这种请求,在 21.3 节中给出了这个新函数的一个实现。

类别	request(请求)	描述	数据类型
套接口	SIOCATMARK	在带外标志上吗?	int
	SIOCSPGRP	设置套接口的进程 ID 或进程组 ID	int
	SIOCGPGRP	获取套接口的进程 ID 或进程组 ID	int
文件	FIONBIO	设置/清除非阻塞标志	int
	FIOASYNC	设置/清除异步 I/O 标志	int
	FIONREAD	获取接收缓冲区中的字节数	int
	FIOSETOWN	设置文件的进程 ID 或进程组 ID	int
	FIOGETOWN	获取文件的进程 ID 或进程组 ID	int
接口	SIOCGIFCONF	获取所有接口的列表	struct ifconf
	SIOCSIFADDR	设置接口地址	struct ifreq
	SIOCGIFADDR	获取接口地址	struct ifreq
	SIOCSIFFLAGS	设置接口标志	struct ifreq
	SIOCGIFFLAGS	获取接口标志	struct ifreq
	SIOCSIFDSTADDR	设置点到点地址	struct ifreq
	SIOCGIFDSTADDR	获取点到点地址	struct ifreq
	SIOCGIFBRDADDR	获取广播地址	struct ifreq
	SIOCSIFBRDADDR	设置广播地址	struct ifreq
	SIOCGIFNETMASK	获取子网掩码	struct ifreq
	SIOCSIFNETMASK	设置子网掩码	struct ifreq

（续）

类别	请求	描述	数据类型
	SIOCGIFMETRIC SIOCSIFMETRIC SIOCxxx	获取接口的测度(metric) 设置接口的测度(metric) (有很多,依赖于实现)	struct ifreq struct ifreq
ARP	SIOCSARP SIOCGARP SIOCDARP	创建/修改 ARP 项 获取 ARP 项 删除 ARP 项	struct arpreq struct arpreq struct arpreq
路由	SIOCADDRT SIOCDELRT	增加路径 删除路径	struct rtentry struct rtentry
流	I_xxx	(参见 33.5 节)	

图 16.1　网络支持 ioctl 请求的总结

SIOCGPGRP　通过第三个参数指向的整数返回为接收来自这个套接口的 SIGIO 或 SIGURG 信号而设置的进程 ID 或进程组 ID。这和 fcntl 的 F_GETOWN 相同,并注意在图 7.15 中 Posix.1g 标准倾向使用 fcntl 方式。

SIOCSPGRP　用第三个参数指向的整数设置进程 ID 或进程组 ID 以接收这个套接口的 SIGIO 或 SIGURG 信号。这和 fcntl 的 F_SETOWN 相同,并注意在图 7.15 中 Posix.1g 标准倾向使用 fcntl 方式。

16.4　文件操作

下一组请求以 FIO 开始,除套接口外,它们对其他某些类型的文件可能也适用。在这里只讨论适用于套接口的请求(TCPv2 第 553 页)。下面的五种请求都要求 ioctl 的第三个参数指向一个整数。

FIONBIO　套接口的非阻塞标志会根据 ioctl 的第三个参数指向的值是否为零而清除或设置。这个请求和用 fcntl 的 F_SETFL 命令设置和清除 O_NONBLOCK 文件状态标志效果相同。

FIOASYNC　这个标志根据 ioctl 的第三个参数指向的值是否为零决定清除或接收套接口上的异步 I/O 信号(SIGIO)。这个标志和用 fcntl 的 F_SETFL 命令设置和清除 O_AYNC 文件状态标志效果相同。

FIONREAD　在 ioctl 的第三个参数指向的整数里返回套接口接收缓冲区中当前的字节数。这种功能在文件、管道和终端上都能用。13.7 节中对此曾有详细介绍。

FIOSETOWN　在套接口上等价于 SIOCSPGRP。

FIOGETOWN　在套接口上等价于 SIOCGPGRP。

16.5 接口配置

很多处理网络接口的程序的第一步是从内核获取系统中配置的所有接口。这是通过SIOCGIFCONF请求来实现的，它使用了ifconf结构，ifconf又用了ifreq结构，这两种结构在图16.2中给出。

在调用ioctl之前分配一个缓冲区和一个ifconf结构，然后初始化后者。在图16.3中展示了这样的一幅图像，其中假定缓冲区的大小为1024字节。ioctl的第三个参数指向ifconf结构。

如果假定内核返回两个ifreq结构，我们在ioctl返回时会得到图16.4所示的结果。阴影区域已被ioctl修改。缓冲区填入了两个结构，ifconf结构的ifc_len成员被更新以反映缓冲区中存放的信息数量。在这幅图中假设每个ifreq结构占用32个字节。

指向ifreq结构的指针也被用作图16.1中其余的接口类ioctl请求的参数，在16.7节中会对此进行介绍。注意每个ifreq结构包含一个联合，有好几个#define隐藏了这些域实际上是这个联合的成员这一事实。对每个成员的引用都使用这样定义的名字。要注意的是有些系统在ifr_ifru联合中增加了很多依赖于实现的成员。

```
struct ifconf {
    int   ifc_len;                  /* size of buffer, value-result */
    union {
        caddr_t ifcu_buf;           /* input from user -> kernel */
        struct  ifreq *ifcu_req;    /* return from kernel -> user */
    } ifc_ifcu;
};
#define ifc_buf ifc_ifcu.ifcu_buf     /* buffer address */
#define ifc_req ifc_ifcu.ifcu_req     /* array of structures returned */
#define IFNAMSIZ    16
struct ifreq {
    char    ifr_name[IFNAMSIZ];         /* interface name, e.g., "le0" */
    union {
        struct sockaddr ifru_addr;
        struct sockaddr ifru_dstaddr;
        struct sockaddr ifru_broadaddr;
        short ifru_flags;
        int   ifru_metric;
        caddr_t ifru_data;
    } ifr_ifru;
};
#define ifr_addr      ifr_ifru.ifru_addr      /* address */
#define ifr_dstaddr   ifr_ifru.ifru_dstaddr   /* other end of p-to-p link */
#define ifr_broadaddr ifr_ifru.ifru_broadaddr /* broadcast address */
#define ifr_flags     ifr_ifru.ifru_flags     /* flags */
#define ifr_metric    ifr_ifru.ifru_metric    /* metric */
#define ifr_data      ifr_ifru.ifru_data      /* for use by interface */
```

图16.2 用于各种ioctl请求的ifconf和ifreq结构[<net/if.h>]

16.6 get_ifi_info 函数

因为有很多程序需要知道在系统中的所有接口，所以我们将开发一个名为 get_ifi_info 的函数，它返回一个结构的链表，每个结构对应一个当前状态为“up”的接口。在这一节中将用 SIOCGIFCONF ioctl 实现这个函数，在第 17 章中将开发一个使用路由套接口的版本。

BSD/OS 提供了一个具有类似功能的名为 getifaddrs 函数。

对 BSD/OS 2.1 的全部源码树的检索结果显示有 12 个程序执行 SIOCGIFCONF ioctl 以确定存在的接口。

我们首先在一个新的名为 unpifi.h 的头文件中定义 ifi_info 结构，如图 16.5 所示。

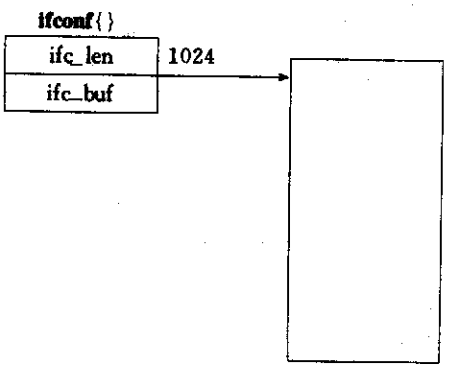


图 16.3 在 SIOCGIFCONF 前对 ifconf 结构的初始化

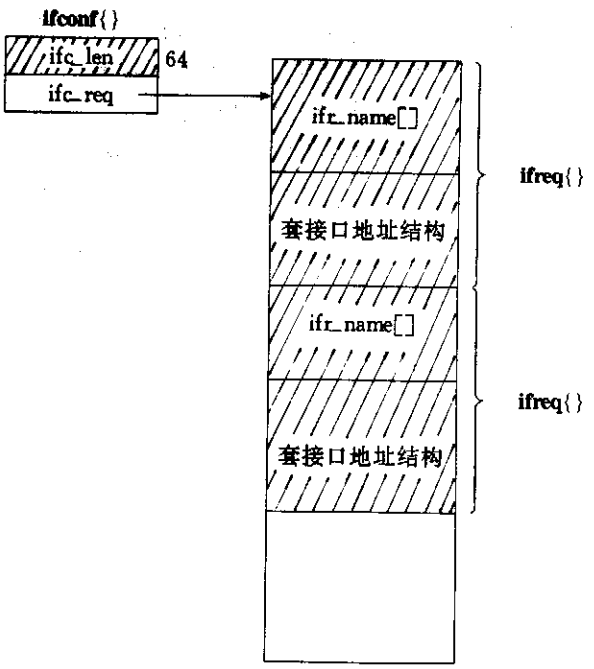


图 16.4 SIOCGIFCONF 返回的值

```
1 /* Our own header for the programs that need interface configuration info.
2  Include this file, instead of "unp.h". */

3 #ifndef  __unp_ifi_h
4 #define  __unp_ifi_h

5 #include       "unp.h"
6 #include       <net/if.h>

7 #define IFI_NAME       16      /* same as IFNAMSIZ in <net/if.h> */
8 #define IFI_HADDR       8      /* allow for 64-bit EUI-64 in future */

9 struct ifi_info {
10    char       ifi_name[IFI_NAME];    /* interface name, null terminated */
11    u_char     ifi_haddr[IFI_HADDR];  /* hardware address */
12    u_short    ifi_hlen;              /* #bytes in hardware address: 0, 6, 8 */
13    short      ifi_flags;             /* IFF_xxx constants from <net/if.h> */
14    short      ifi_myflags;           /* our own IFI_xxx flags */
15    struct sockaddr  *ifi_addr;       /* primary address */
16    struct sockaddr  *ifi_brdaddr;    /* broadcast address */
17    struct sockaddr  *ifi_dstaddr;    /* destination address */
18    struct ifi_info  *ifi_next;       /* next of these structures */
19 };
20 #define IFI_ALIAS      1            /* ifi_addr is an alias */

21                        /* function prototypes */
22 struct ifi_info  *get_ifi_info(int, int);
23 struct ifi_info  *Get_ifi_info(int, int);
24 void      free_ifi_info(struct ifi_info *);

25 #endif   /* __unp_ifi_h */
```

图 16.5 unpifi.h 头文件[ioctl/unpifi.h]

第 9～19 行 我们的函数返回一个这些结构的链表，其中每个结构的 ifi_next 成员指向下一个结构。在这个结构中只返回典型应用程序可能感兴趣的东西：接口的名字、硬件地址(譬如以太网地址)、接口的标志(让应用程序判断接口是否支持广播或多播或是一个点到点接口)、接口的地址、广播地址及点到点链路的目的地址。存放 ifi_info 结构和套接口地址结构的内存是动态申请的。因此我们还提供了一个 free_ifi_info 函数释放这部分内存。

> 今天的大部分硬件地址是 48 位的 MAC 地址(譬如以太网、令牌环网，等等)。但有向叫做 EUI-64[IEEE 1997b]的 64 位标识发展的趋势。IPv6 地址在低 64 位中包含一个 EUI-64 的值(A.5 节)，而且有一个简单的方法可以把 48 位的 MAC 地址封装到 64 位的 EUI 中。因此在 ifi_info 结构中分配了足够存放一个 64 位标识的空间，同时保存硬件地址的长度。

在给出 get_ifi_info 函数的实现之前，我们先来看一个简单的程序，该程序调用这个函数，然后输出所有信息。这个程序是 ifconfig 程序的一个简化版本，如图 16.6 所示。

```
1 #include    "unpifi.h"

2 int
3 main(int argc, char **argv)
4 {
5     struct ifi_info   *ifi, *ifihead;
```

```
 6    struct sockaddr  *sa;
 7    u_char   *ptr;
 8    int      i, family, doaliases;
 9    if (argc != 3)
10        err_quit("usage: prifinfo <inet4|inet6> <doaliases>");
11    if (strcmp(argv[1], "inet4") == 0)
12        family = AF_INET;
13 #ifdef      IPV6
14    else if (strcmp(argv[1], "inet6") == 0)
15        family = AF_INET6;
16 #endif
17    else
18        err_quit("invalid <address-family>");
19    doaliases = atoi(argv[2]);
20    for (ifihead = ifi = Get_ifi_info(family, doaliases);
21         ifi != NULL; ifi = ifi->ifi_next) {
22        printf("%s: <", ifi->ifi_name);
23        if (ifi->ifi_flags & IFF_UP)                printf("UP ");
24        if (ifi->ifi_flags & IFF_BROADCAST)         printf("BCAST ");
25        if (ifi->ifi_flags & IFF_MULTICAST)         printf("MCAST ");
26        if (ifi->ifi_flags & IFF_LOOPBACK)          printf("LOOP ");
27        if (ifi->ifi_flags & IFF_POINTOPOINT)       printf("P2P ");
28        printf(">\n");
29        if ( (i = ifi->ifi_hlen) > 0) {
30            ptr = ifi->ifi_haddr;
31            do {
32                printf("%s%x", (i == ifi->ifi_hlen) ? "  " : ":", *ptr++);
33            } while (--i > 0);
34            printf("\n");
35        }
36        if ( (sa = ifi->ifi_addr) != NULL)
37            printf("  IP addr: %s\n",
38                   Sock_ntop_host(sa, sizeof(*sa)));
39        if ( (sa = ifi->ifi_brdaddr) != NULL)
40            printf("  broadcast addr: %s\n",
41                   Sock_ntop_host(sa, sizeof(*sa)));
42        if ( (sa = ifi->ifi_dstaddr) != NULL)
43            printf("  destination addr: %s\n",
44                   Sock_ntop_host(sa, sizeof(*sa)));
45    }
46    free_ifi_info(ifihead);
47    exit(0);
48 }
```

图 16.6　调用 get_ifi_info 函数的 prifinfo 程序[ioctl/prifinfo.c]

第 20～45 行　这个程序是一个 for 循环，调用一次 get_ifi_info 后遍历返回的所有 ifi_info 结构。

第 22～35 行　输出所有的接口名和标志。如果硬件地址长度大于 0，就把它以十六进制数形式输出。（如果无法得到硬件地址，get_ifi_info 函数返回的 ifi_hlen 将为 0。）

第36～44行 如果返回了所需的三个IP地址，就输出它们。

如果在主机solaris(图1.16)上运行这个程序，会得到以下的输出：

```
solaris % prifinfo inet4 0
lo0: <UP MCAST LOOP>
  IP addr: 127.0.0.1
le0: <UP BCAST MCAST>
  IP addr: 206.62.226.33
  broadcast addr: 206.62.226.63
```

第一个命令行参数inet4指明要获取IPv4地址，第二个参数0指定不返回地址别名(在A.4节中介绍了IP地址别名)。注意在Solaris上以太网接口的硬件地址是得不到的。

如果给以太网接口(le0)加上三个别名地址，其主机ID分别为44、45和46，而且把第二个命令行参数改为1，会有以下结果：

```
solaris % prifinfo inet4 1
lo0:<UP MCAST LOOP>
  IP addr: 127.0.0.1
le0:<UP BCAST MCAST>
  IP addr: 206.62.226.33          主IP地址
  broadcast addr: 206.62.226.63
le0:1:<UP BCAST MCAST>
  IP addr: 206.62.226.44          第一个别名
  broadcast addr: 206.62.226.63
le0:2:<UP BCAST MCAST>
  IP addr: 206.62.226.45          第二个别名
  broadcast addr: 206.62.226.63
le0:3: <UP BCAST MCAST>
  IP addr: 206.62.226.46          第三个别名
  broadcast addr: 206.62.226.63
```

如果在BSD/OS下运行同样的程序，使用图17.16中get_ifi_info的实现(用它可以很容易地获得硬件地址)，会有以下结果：

```
bsdi % prifinfo inet4 1
we0: <UP BCAST MCAST>
  0:0:c0:6f:2d:40
  IP addr: 206.62.226.66
  broadcast addr: 206.62.226.95
ef0: <UP BCAST MCAST>
  0:20:af:9c:ee:95
  IP addr:206.62.226.35           主IP地址
  broadcast addr:206.62.226.63
ef0: <UP BCAST MCAST>
  0:20:af:9c:ee:95
  IP addr: 206.62.226.50          别名
  broadcast addr: 206.62.226.63
lo0: <UP MCAST LOOP>
  IP addr: 127.0.0.1
```

在这个例子中指示程序输出别名地址，可以看到第二个以太网接口定义了一个主机ID为50的别名。

这个输出依赖于接口的别名地址是如何建立的。在这个例子中别名地址是赋给以太网接口 ef0 的，在 BSD/OS 2.1 中这是通常的用法。但在 BSD/OS 3.0 中建议把别名地址赋给回馈接口 lo0。

下面给出用 SIOCGIFCONF ioctl 实现的 get_ifi_info。图 16.7 给出了函数的第一部分，它从内核获取接口配置。

```
1  #include    "unpifi.h"
2  struct ifi_info *
3  get_ifi_info(int family, int doaliases)
4  {
5      struct ifi_info *ifi, *ifihead, **ifipnext;
6      int          sockfd, len, lastlen, flags, myflags;
7      char         *ptr, *buf, lastname[IFNAMSIZ], *cptr;
8      struct ifconf ifc;
9      struct ifreq *ifr, ifrcopy;
10     struct sockaddr_in   *sinptr;
11     sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
12     lastlen = 0;
13     len = 100 * sizeof(struct ifreq);   /* initial buffer size guess */
14     for ( ; ; ) {
15         buf = Malloc(len);
16         ifc.ifc_len = len;
17         ifc.ifc_buf = buf;
18         if (ioctl(sockfd, SIOCGIFCONF, &ifc) < 0) {
19             if (errno != EINVAL || lastlen != 0)
20                 err_sys("ioctl error");
21         } else {
22             if (ifc.ifc_len == lastlen)
23                 break;              /* success, len has not changed */
24             lastlen = ifc.ifc_len;
25         }
26         len += 10 * sizeof(struct ifreq);   /* increment */
27         free(buf);
28     }
29     ifihead = NULL;
30     ifipnext = &ifihead;
31     lastname[0] = 0;
```

图 16.7　发出 SIOCGIFCONF 请求以获取接口配置[lib/get_ifi_info.c]

创建一个网际套接口

第 11 行　创建一个用于 ioctl 的 UDP 套接口。TCP 或 UDP 套接口都可以(TCPv2 第 163 页)。

在循环中发出 SIOCGIFCONF 请求

第 12～28 行　SIOCGIFCONF 请求有一个严重的问题是，一些实现在缓冲区的大小不足以存放结果时不返回错误，而是截断结果并返回成功(ioctl 返回 0)。这意味着要知道缓冲区是否够大的唯一办法是，发出请求，记下返回的长度，用更大的缓冲区发出请求，再比较返

回的长度和前面记下的长度。只有在这两个长度相同的情况下，缓冲区才是足够大的。

> 源自 Berkeley 的实现在缓冲区太小的情况下不返回错误(TCPv2 第 118～119 页)；结果会被截断以放入缓冲区。与此相反，Solaris 2.5 在返回的长度将大于或等于缓冲区长度时返回 EINVAL。但是在返回的长度小于缓冲区大小的情况下我们也不能肯定成功，因为源自 Berkeley 的实现在剩下的空间装不下另一个结构时返回的长度会小于缓冲区长度。
>
> 一些实现中提供了一个 SIOCGIFNUM 请求以返回接口的数目。这可以让应用进程在发出 SIOCGIFCONF 请求前分配足够的缓冲区空间，但这个新的请求没有被广泛实现。
>
> 随着 Web 应用的增长，为 SIOCGIFCONF 请求返回的结果分配一个固定长度的缓冲区已经成为一个问题，因为大的 Web 服务器给单独一个接口就分配很多别名地址。举例来说，Solaris 2.5有每个接口最多 256 个别名地址的限制，但这个限制在 2.6 中增加到 8192。有大量别名地址的站点发现对接口信息使用固定大小缓冲区的程序开始不工作。尽管 Solaris 在缓冲区太小的情况下返回错误，这些程序还是分配它们固定大小的缓冲区，发出 ioctl，结果要是早先返回过错误的话就死掉。

第 12～15 行　分配一个缓冲区，开始时为 100 个 ifreq 结构的空间。把 lastlen 初始化成 0，并在 lastlen 中记录最近一个 SIOCGIFCONF 请求返回的长度。

第 19～20 行　如果 ioctl 返回一个 EINVAL 错误，而且还没有得到过一个成功的返回(也就是 lastlen 还是 0)，那么我们还没有分配一个足够大的缓冲区，继续进行循环。

第 22～23 行　如果 ioctl 返回成功，而且返回的长度等于 lastlen，那么这个长度没有改变(即缓冲区足够大)，于是 break 出这个循环，因为已经得到了所有信息。

第 26～27 行　每循环一次，把缓冲区的大小增加能多存放 10 个 ifreq 结构的空间。

初始化链表指针

第 29～31 行　因为将要返回一个指向 ifi_info 结构的链表头的指针，所以用两个变量 ifihead 和 ifipnext 在建立链表时保存指针。这和在图 11.34 中介绍的技术相同。

get_ifi_info 函数的下一部分，即主循环的开头，如图 16.8 所示。

```
32      for (ptr = buf; ptr < buf + ifc.ifc_len; ) {
33          ifr = (struct ifreq *) ptr;
34  #ifdef  HAVE_SOCKADDR_SA_LEN
35          len = max(sizeof(struct sockaddr), ifr->ifr_addr.sa_len);
36  #else
37          switch (ifr->ifr_addr.sa_family) {
38  #ifdef  IPV6
39          case AF_INET6:
40            len = sizeof(struct sockaddr_in6);
41            break;
42  #endif
43          case AF_INET:
44          default:
45              len = sizeof(struct sockaddr);
```

```
46                  break;
47          }
48 #endif    /* HAVE_SOCKADDR_SA_LEN */
49          ptr += sizeof(ifr->lfr_name) + len; /* for next one in buffer */
50          if (ifr->ifr_addr.sa_family != family)
51              continue;   /* ignore if not desired address family */
52          myflags = 0;
53          if ( (cptr = strchr(ifr->lfr_name, ':')) != NULL)
54              *cptr = 0;          /* replace colon will null */
55          if (strncmp(lastname, ifr->ifr_name, IFNAMSIZ) == 0) {
56              if (doaliases == 0)
57                  continue;   /* already processed this interface */
58              myflags = IFI_ALIAS;
59          }
60          memcpy(lastname, ifr->ifr_name, IFNAMSIZ);
61          ifrcopy = *ifr;
62          ioctl(sockfd, SIOCGIFFLAGS, &ifrcopy);
63          flags = ifrcopy.ifr_flags;
64          if((flags & IFF_UP)==0)
65              continue;           /* ignore if interface not up */
66          ifi = Calloc(1, sizeof(struct ifi_info));
67          *ifipnext = ifi;        /* prev points to this new one */
68          ifipnext = &ifi->ifi_next;      /* pointer to next one goes here */
69          ifi->ifi_flags = flags;         /* IFF_xxx values */
70          ifi->ifi_myflags = myflags;     /* IFF_xxx values */
71          memcpy(ifi->ifi_name, ifr->ifr_name, IFI_NAME);
72          ifi->ifi_name[IFI_NAME-1] = '\0';
```

图 16.8 处理接口配置[lib/get_ifi_info.c]

移到下一个套接口地址结构

第 32～49 行 在遍历所有 ifreq 结构时，ifr 指向每个结构，然后我们增加 ptr 指向下一个结构。但我们必须处理两种情况，即给套接口地址结构提供长度成员的新系统和不提供这个长度的老系统。虽然图 16.2 中声明的包含于 ifreq 结构中的套接口地址结构是一个通用套接口地址结构，但是在较新的系统中它可以是任何类型的套接口地址结构。实际上，在 4.4BSD 中每个接口还返回一个数据链路套接口地址结构(TCPv2 第 118 页)。因此如果支持长度成员，我们必须用这个值来将指针移到下一个套接口地址结构。否则使用基于地址族的长度，缺省为通用套接口地址结构的大小(16 字节)。

> 在支持 IPv6 的系统中，没有关于对 SIOCGIFCONF 请求是否返回 IPv6 地址的标准。我们给支持 IPv6 的新系统加了一个 case 语句，这是为了预防万一。问题在于 ifreq 结构中的联合把返回的地址定义成一个通用的 16 字节套接口地址结构，适合 16 字节的 IPv4 sockaddr_in 结构，但对 24 字节的 IPv6 sockaddr_in6 结构太小了。如果返回 IPv6 地址，将可能破坏现有的在每个 ifreq 结构中采用固定大小的套接口地址结构的代码。

第 50～51 行 忽略所有不是调用者期望的地址族的地址。

处理别名地址

第52～60行 我们必须检测在该接口上可能存在的任何别名地址，也就是赋给这个接口的其他地址。注意跟在图16.6之后的例子，在Solaris上别名地址的接口名中含有一个冒号，但在4.4BSD中接口名并不因别名地址而改变。为了处理这两种情况，我们把最近处理的接口名存到lastname，如果存在冒号的话，只比较到冒号。如果没有冒号，而且名字和最近处理的接口相同，我们也忽略这个接口。

取接口标志

第61～65行 发出一个SIOCGIFFLAGS ioctl请求(16.5节)取接口标志。ioctl的第三个参数是一个指向ifreq结构的指针，结构中包含要获取标志的接口名。在发出ioctl之前对ifreq结构做一个拷贝，因为如果不这样做的话，这个请求会覆盖接口的IP地址，因为它们都是图16.2中所示的同一个联合的成员。如果接口不在工作，就忽略掉。

分配和初始化ifi_info结构

第66～72行 到这时我们知道将向调用者返回这个接口。给ifi_info结构分配内存，并把它加到正在建立的链表的末尾。把接口的标志和名字复制到这个结构中。我们保证接口名是以空字符结尾的，而且因为calloc把分配的区域都初始化成0，我们从而知道ifi_hlen被初始化成0，ifi_next也被初始化成空指针。

图16.9是这个函数的最后一部分。

```
73          switch (ifr->ifr_addr.sa_family) {
74          case AF_INET:
75            sinptr = (struct sockaddr_in *) &ifr->ifr_addr;
76            if (ifi->ifi_addr == NULL) {
77                ifi->ifi_addr = Calloc(1, sizeof(struct sockaddr_in));
78                memcpy(ifi->ifi_addr, sinptr, sizeof(struct sockaddr_in));
79 #ifdef     SIOCGIFBRDADDR
80                    if (flags & IFF_BROADCAST) {
81                        ioctl(sockfd, SIOCGIFBRDADDR, &ifrcopy);
82                        sinptr = (struct sockaddr_in *) &ifrcopy.ifr_broadaddr;
83                        ifi->ifi_brdaddr = Calloc(1, sizeof(struct sockaddr_in));
84                        memcpy(ifi->ifi_brdaddr, sinptr, sizeof(struct sockaddr_in));
85                    }
86 #endif
87 #ifdef     SIOCGIFDSTADDR
88                    if (flags & IFF_POINTOPOINT) {
89                        ioctl(sockfd, SIOCGIFDSTADDR, &ifrcopy);
90                        sinptr = (struct sockaddr_in *) &ifrcopy.ifr_dstaddr;
91                        ifi->ifi_dstaddr = Calloc(1, sizeof(struct sockaddr_in));
92                          memcpy(ifi->ifi_dstaddr, sinptr, sizeof(struct sockaddr_
                            in));
93                    }
94 #endif
95                }
96                break;
97          default:
98              break;
```

```
 99          }
100      }
101      free(buf);
102      return(ifihead);    /* pointer to first structure in linked list */
103  }
```

图 16.9　取得并返回接口地址[ioctl/get_ifi_info.c]

第 73～78 行　把从最初的 SIOCGIFCONF 请求返回的 IP 地址复制到我们正在创建的结构中。

第 79～96 行　如果该接口支持广播，就用 SIOCGIFBRDADDR ioctl 取得广播地址。给包含这个地址的套接口地址结构分配内存，并把它加到正在创建的 ifi_info 结构中。与此类似，如果接口是一个点到点接口，SIOCGIFDSTADDR 返回链路另一端的 IP 地址。

> 因为前面已经提到过，现在还不知道 IPv6 的实现在 SIOCGIFCONF 请求时是否返回 IPv6 地址，所以这里没有讨论 AF_INET6 的情况。

图 16.10 给出了 free_ifi_info 函数，它以一个由 get_ifi_info 返回的指针做参数，翻放所有动态分配的内存。

```
104 void
105 free_ifi_info(struc ifi_info *ifihead)
106 {
107     struct ifi_info     *ifi, *ifinext;
108     for(ifi = ifihead; ifi != NULL; ifi = ifinext) {
109         if(ifi->ifi_addr != NULL)
110             free(ifi->ifi_addr);
111         if (ifi->ifi_brdaddr != NULL)
112             free(ifi->ifi_brdaddr);
113         if (ifi->ifi_dstaddr != NULL)
114             free(ifi->ifi_dstaddr);
115         ifinext = ifi->ifi_next;      /* can't fetch ifi_next after free() */
116         free(ifi);          /* the ifi_info{} itself */
117     }
118 }
```

图 16.10　free_ifi_info 函数：释放由 get_ifi_info 分配的内存[ioctl/get_ifi_info.c]

16.7　接口操作

如在前一节中给出的，SIOCGIFCONF 请求返回每个已配置接口的名字和套接口地址结构。另外有很多请求可以设置或获取接口的其他特性。这些请求的 get 版本(SIOCGxxx)一般由 netstat 程序使用，而 set 版本(SIOCSxxx)一般由 ifconfig 程序使用。任何用户都可以获取接口信息，但要想设置这些信息必须有超级用户权限。

这些请求以 ifreq 结构为参数，或返回一个 ifreq 结构，其地址由 ioctl 的第三个参数指定。接口总是用名字来标识：le0、lo0、ppp0 或其他在 ifr_name 成员中的任何内容。

这些请求中有许多使用套接口地址结构在应用进程中指定或返回 IP 地址或地址掩码。

对于IPv4,这个地址或掩码包含在网际套接口地址结构的sin_addr成员中。

SIOCGIFADDR	在ifr_addr成员中返回单播地址。
SIOCSIFADDR	用ifr_addr成员设置接口地址。这个接口的初始化函数也被调用
SIOCGIFFLAGS	在ifr_flags成员中返回接口标志。各种标志的名字为IFF_xxx,在<net/if.h>头文件中定义。举例来说,这些标志表示接口是否在工作(IFF_UP),接口是不是一个点到点接口(IFF_POINTOPOINT),接口是否支持广播(IFF_BROADCAST),等等。
SIOCSIFFLAGS	用ifr_flags成员设置接口标志。
SIOCGIFDSTADDR	在ifr_dstaddr成员中返回点到点地址。
SIOCSIFDSTADDR	用ifr_dstaddr成员设置点到点地址。
SIOCGIFBRDADDR	在ifr_broadaddr成员中返回广播地址。应用进程必须首先取到接口的标志,然后发出正确的请求:对广播接口用SIOCGIFBRDADDR,对点到点接口用SIOCGIFDSTADDR。
SIOCSIFBRDADDR	用ifr_broadaddr成员设置广播地址。
SIOCGIFNETMASK	在ifr_addr成员中返回子网掩码。
SIOCSIFNETMASK	用ifr_addr成员设置子网掩码。
SIOCGIFMETRIC	用ifr_metric成员返回接口的测度。每个接口的测度由内核维护,但使用它的是路由守护进程routed。接口的测度被加到跳数上(使这个接口更难被选中)。
SIOCSIFMETRIC	用ifr_metric成员设置接口的路由测度。

在本节中介绍了一般的接口请求。很多实现中都加入了其他的请求。

16.8 ARP高速缓存操作

ARP高速缓存也是由ioctl函数操作的。这些请求使用一个arpreq结构,在图16.11中给出了这个结构,它是在<net/if_arp.h>头文件中定义的。

```
struct arpreq {
    struct sockaddr arp_pa;          /* protocol address */
    struct sockaddr arp_ha;          /* hardware address */
    int             arp_flags;       /* flags */
};
#define ATF_INUSE       0x01    /* entry in use */
#define ATF_COM         0x02    /* completed entry (hardware addr valid) */
#define ATF_PERM        0x04    /* permanent entry */
#define ATF_PUBL        0x08    /* published entry (respond for other host) */
```

图16.11 用于ARP高速缓存的ioctl请求的arpreq结构[<net/if_arp.h>]

ioctl的第三个参数必须指向这些结构之一。它支持下面的三种请求:

SIOCSARP	把新项加到 ARP 高速缓存中或修改一个已有项。arp_pa 是一个网际套接口地址结构，其中包含 IP 地址，arp_ha 是一个通用套接口地址结构，其中 sa_family 为 AF_UNSPEC，sa_data 中是硬件地址（例如 6 字节的以太网地址）。ATF_PERM 和 ATF_PUBL 这两个标志可以由应用程序指定。另外两个标志，ATF_INUSE 和 ATF_COM 由内核设置。
SIOCDARP	从 ARP 高速缓存中删除一项。调用者指定要删除项的网际套接口地址。
SIOCGARP	从 ARP 高速缓存中取一项。调用者指定所取项的网际套接口地址，对应的以太网地址会和标志一起返回。

只有超级用户才能增加或删除数据项。这三种请求一般是由 arp 程序发出的。

在一些新系统中不支持这些和 ARP 有关的 ioctl 请求，它们使用路由套接口进行这些 ARP 操作。

注意 ioctl 没有办法列出 ARP 高速缓冲中的所有项。arp 命令的大部分版本，在有-a 标志时（列出 ARP 高速缓存中的所有项）读内核的内存（/dev/kmem）以获取当前 ARP 高速缓存中的内容。在 17.4 节中我们会看到用 sysctl 做到这一点的一种更简单（且更好）的方法。

例子：输出主机的硬件地址

现在用图 9.7 中的的 my_addrs 函数返回一台主机的全部 IP 地址，然后对每个 IP 地址发出 SIOCGARP ioctl 请求，得到并输出硬件地址。程序如图 16.12 所示。

获取地址列表并就每个地址进行循环

第 12～13 行　调用 my_addrs 获取主机的所有 IP 地址，然后就每个地址进行循环。

输出 IP 地址

第 14～17 行　用 inet_ntop 输出 IP 地址，接着根据 my_addrs 返回的地址族作不同处理。我们只处理 IPv4 地址，因为厂商可能将不支持 IPv6 地址的 SIOCGARP 请求。

发出 ioctl 请求，输出硬件地址

第 18～26 行　向 arp_pa 结构中填入一个包含 IPv4 地址的 IPv4 套接口地址结构。调用 ioctl 然后输出返回的硬件地址。

在我们的 solaris 主机上运行这个程序会有以下结果：

```
solaris % prmac
206.62.226.33: 8:0:20:78:e3:e3
```

```
 1 #include    "unp.h"
 2 #include    <net/if_arp.h>
 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     family, sockfd;
 7     char    str[INET6_ADDRSTRLEN];
```

```
 8    char            * * pptr;
 9    unsigned char  * ptr;
10    struct arpreq arpreq;
11    struct sockaddr_in  * sin;
12    pptr = my_addrs(&family);
13    for ( ; * pptr ! = NULL; pptr++) {
14        printf("%s: ", inet_ntop(family, * pptr, str, sizeof(str)));
15        switch (family) {
16        case AF_INET:
17            sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
18            sin = (struct sockaddr_in  * ) &arpreq.arp_pa;
19            bzero(sin, sizeof(struct sockaddr_in));
20            sin->sin_family = AF_INET;
21            memcpy(&sin->sin_addr, * pptr, sizeof(struct in_addr));
22            ioctl(sockfd, SIOCGARP, &arpreq);
23            ptr = &arpreq.arp_ha.sa_data[0];
24            printf("%x:%x:%x:%x:%x:%x\n", * ptr, * (ptr+1),
25                   * (ptr+2), * (ptr+3), * (ptr+4), * (ptr+5));
26            break;
27        default:
28            err_quit("unsupported address family: %d", family);
29        }
30    }
31    exit(0);
32 }
```

图 16.12 输出一台主机的硬件地址[ioctl/prmac.c]

16.9 路由表操作

有两种 ioctl 请求用来操作路由表。这两个请求要求 ioctl 的第三个参数必须是一个指向 rtentry 结构的指针,这个结构在<net/route.h>头文件中定义。这些请求一般由 route 程序发出。只有超级用户才能发出这些请求。

SIOCADDRT　　向路由表中加一项。
SIOCDELRT　　从路由表中删去一项。

ioctl 没有办法列出路由表中的所有项。这种操作通常是带有-r 标志的 netstat 程序执行的。这个程序通过读内核的内存(/dev/kmem)获得路由表。和列出 ARP 高速缓存一样,在 17.4 节中我们会看到用 sysctl 做到这一点的一种更简单(且更好)的方法。

16.10 小 结

在网络编程中使用的 ioctl 命令可分为六类:

- 套接口操作(是否位于带外标志上?),
- 文件操作(设置或清除非阻塞标志),

- 接口操作(返回接口表,获取广播地址),
- ARP 表操作(创建,修改,获取,删除),
- 路由表操作(增加或删除),和
- 流系统(第 33 章)。

我们将使用这些套接口和文件操作,而获取接口表是一个常用操作,因此我们专为此开发了一个函数。在本书的后面会数次使用这个函数。只有几个特殊的专门的程序使用 ioctl 操作 ARP 高速缓冲和路由表。

16.11　习　题

16.1　在 16.7 节中我们说 SIOCGIFBRDADDR 请求是在 ifr_broadaddr 成员中返回广播地址。但在 TCPv2 第 173 页却说在 ifr_dstaddr 成员中返回。这有问题吗?

16.2　修改 get_ifi_info 程序,对一个 ifreq 结构发出第一次 SIOCGIFCONF 请求,然后每次循环,把缓冲区长度增加这些结构之一的大小。在循环中加一些语句,在每次发出请求时,不管 ioctl 是否返回错误都输出缓冲区的大小,在成功时再输出返回的缓冲区长度。运行 prifinfo 程序,看看你的系统在缓冲区太小时是怎样处理这个请求的。对所有返回的地址族不是想要的值的结构,也输出出它们的地址族,看看你的系统返回了哪些其他的结构。

16.3　修改 get_ifi_info 函数,返回接口的各个别名地址中与前一个不处于同一子网上的那些地址的信息。也就是说,16.6 节中的版本忽略从 206.62.226.44 到 206.62.226.46 的别名地址,因为它们和这个接口的主地址 206.62.226.33 在同一个子网。但是如果别名地址在不同的子网,譬如说 192.3.4.5,修改后的版本应该返回一个带有这个附加地址信息的 ifi_info 结构。

16.4　如果你的系统支持 SIOCGIFNUM ioctl,那么修改图 16.7 中的程序,发出这个请求,用返回的值作为最初猜测的缓冲区大小。

第17章 路由套接口

17.1 概 述

过去在内核中的Unix路由表是用ioctl命令来访问的。在16.9节中介绍了提供的两个命令:SIOCADDRT和SIOCDELRT,它们增加或删除一条路径。我们还提到没有命令能取到整个路由表。像netstat这样的程序通过读取内核得到路由表的内容。还有一点是路由守护进程如gated需要监视内核收到的ICMP重定向消息,它们通常创建一个原始ICMP套接口(第25章),在这个套接口上监听所有收到的ICMP消息。

4.3BSD Reno通过创建AF_ROUTE域去掉了这个内核路由子系统的接口。在路由域中支持的唯一一种套接口类型是原始套接口。在路由套接口中支持三种类型的操作。

1. 进程能通过写路由套接口向内核发消息。举例来说,路径就是这样增加和删除的。
2. 进程能在路由套接口上从内核读消息。这是核心通知进程已收到一个ICMP重定向消息并进行了处理的方式。
   一些操作包含这两步:举例来说,进程在路由套接口上向内核发出一个消息,请求在一个给定路径上的所有信息,这个进程又从路由套接口上读回内核的应答。
3. 进程可以用sysctl函数(17.4节)得到路由表或列出所有已配置的接口。

前两种操作需要超级用户权限,最后一种操作任何进程都可以执行。

> 从技术上说,第三种操作不是用路由套接口执行的,而是用一般的sysctl函数。但是sysctl所用的输入参数之一是地址族,它是本章将介绍的操作所用的AF_ROUTE,而且sysctl返回的信息也与内核在路由套接口上返回的信息格式相同。实际上,sysctl对AF_ROUTE族的处理是4.4BSD内核的路由套接口代码的一部分(TCPv2第632~643页)。
>
> sysctl是在4.4BSD中出现的。不幸的是并不是所有支持路由套接口的实现都提供sysctl。举例来说,AIX4.2、Digital Unix 4.0和Solaris 2.6都支持路由套接口,但没有一个支持sysctl。

17.2 数据链路套接口地址结构

在路由套接口上返回的一些消息中包含数据链路套接口地址结构。图17.1给出了这个结构的定义,它是在<net/if_dl.h>中定义的。

```
struct sockaddr_dl {
    uint8_t         sdl_len;
    sa_family_t     sdl_family;     /* AF_LINK */
    uint16_t        sdl_index;      /* system assigned index, if > 0 */
```

```
    uint8_t     sdl_type;       /* IFT_ETHER, ect. from <net/if_types.h> */
    uint8_t     sdl_nlen;       /* name length, starting in sdl_data[0] */
    uint8_t     sdl_alen;       /* link-layer address length */
    uint8_t     sdl_slen;       /* link-layer selector length */
    char        sdl_data[12];   /* minimum work area, can be larger; contains i/f
                                   name and link-layer address */
};
```

图 17.1　数据链路套接口地址结构

每个接口都有一个唯一的大于 0 的索引号，本章后面介绍的 if_nametoindex 和 if_nameindex 函数会返回这个索引号，第 19 章中的 IPv6 多播套接口选项也用到它。

sdl_data 成员包含名字和链路层地址(例如以太网接口的 48 位 MAC 地址)。名字从 sdl_data[0]开始，而且不以空字符终止。链路层地址从名字后面的 sdl_nlen 字节开始。这个头文件定义了下面这个宏以返回指向链路层地址的指针：

```
#define LLADDR(s)  ((caddr_t)((s)->sdl_data + (s)->sdl_nlen))
```

这些套接口地址结构是可变长度的(TCPv2 第 89 页)。如果链路层地址和名字超出 12 字节，这个结构将大于 20 字节。在 32 位系统上，这个结构的大小一般向上舍入到下一个 4 字节的倍数。在图 20.3 中还会看到，当用 IP_RECVIF 套接口选项返回这些结构之一时，全部三个长度成员都是 0，而且根本没有 sdl_data 成员。

17.3　读　和　写

进程在创建路由套接口后，可以通过写或读套接口向内核发命令或从核心读信息。有 12 个不同的路由命令，其中 5 个可以由进程发出。这些命令在<net/route.h>头文件中定义，如图 17.2 所示。

消息类型	去往内核?	来自内核?	描　述	结构类型
RTM_ADD	•	•	增加路径	rt_msghdr
RTM_CHANGE	•	•	修改网关、测试或标志	rt_msghdr
RTM_DELADDR		•	地址被从接口中删除	ifa_msghdr
RTM_DELETE	•	•	删除路径	rt_msghdr
RTM_GET	•	•	报告测试及其他路径信息	rt_msghdr
RTM_IFINFO		•	接口工作，不工作等等	if_msghdr
RTM_LOCK	•	•	锁住指定的测度	rt_msghdr
RTM_LOSING		•	内核怀疑路径将断开	rt_msghdr
RTM_MISS		•	地址查找失败	rt_msghdr
RTM_NEWADDR		•	地址被加到接口上	ifa_msghdr
RTM_REDIRECT		•	内核被告知使用另外的路径	rt_msghdr
RTM_RESOLVE		•	请求把目的地址解析成链路层地址	rt_msghdr

图 17.2　通过路由套接口交换的消息类型

通过路由套接口交换的结构有三种，如上图的最后一列所示：rt_msghdr、if_msghdr 和 ifa_msghdr，具体的定义如图 17.3 所示。

```
struct rt_msghdr {      /* from <net/route.h> */
```

```
    u_short     rtm_msglen;     /* to skip over non-understood messages */
    u_char      rtm_version;    /* future binary compatibility */
    u_char      rtm_type;       /* message type */
    u_short     rtm_index;      /* index for associated ifp */
    int         rtm_flags;      /* flags, incl. kern & message, e.g., DONE */
    int         rtm_addrs;      /* bitmask identifying sockaddrs in msg */
    pid_t       rtm_pid;        /* identify sender */

    int         rtm_seq;        /* for sender to identify action */
    int         rtm_errno;      /* why failed */
    int         rtm_inits;      /* from rtentry */
    u_long      rtm_inits;      /* which metrics we are initializing */
    struct rt_metrics rtm_rmx;  /* metrics themselves */
};

struct if_msghdr {          /* from <net/if.h> */
    u_short     ifm_msglen;     /* to skip over non-understood messages */
    u_char      ifm_version;    /* future binary compatibility */
    u_char      ifm_type;       /* message type */

    int         ifm_addrs;      /* like rtm_addrs */
    int         ifm_flags;      /* value of if_flags */
    u_short     ifm_index;      /* index for associated ifp */
    struct if_data ifm_data;    /* statistics and other data about if */
};

struct ifa_msghdr {     /* from <net/if.h> */
    u_short     ifam_msglen;    /* to skip over non-understood messages */
    u_char      ifam_version;   /* future binary compatibility */
    u_char      ifam_type;      /* message type */

    int         ifam_addrs;     /* like rtm_addrs */
    int         ifam_flags;     /* value of ifa_flags */
    u_short     ifam_index;     /* index for associated ifp */
    int         ifam_metric;    /* value of ifa_metric */
};
```

图 17.3 路由消息返回的三种结构

每个结构的前三个成员是相同的：长度、版本和消息类型。类型是图 17.2 的第一列中的常值之一。长度成员可以让应用程序跳过它不理解的消息类型。

rtm_addrs、ifm_addrs 和 ifam_addrs 成员是位掩码，用来指明消息后的套接口地址结构是八种可能的类型中的哪一种。图 17.4 给出了在<net/route.h>头文件中为该位掩码定义的各常值的值。

位掩码		数组下标		套接口地址结构包含
常值名	值	常值名	值	
RTA_DST	0x01	RTAX_DST	0	目的地址
RTA_GATEWAY	0x02	RTAX_GATEWAY	1	网关地址
RTA_NETMASK	0x04	RTAX_NETMASK	2	网络掩码
RTA_GENMASK	0x08	RTAX_GENMASK	3	复制掩码

（续）

位掩码		数组索引		套接口地址结构包含
常值名	值	常值名	值	
RTA_IFP	0x10	RTAX_IFP	4	接口名
RTA_IFA	0x20	RTAX_IFA	5	接口地址
RTA_AUTHOR	0x40	RTAX_AUTHOR	6	重定向的发起者
RTA_BRD	0x08	RTAX_BRD	7	广播或点到点目的地址
		RTAX_MAX	8	最大元素数目

图 17.4　用于在路由消息中指称套接口地址结构的常值

当有多个套接口地址结构时，它们总是按表中所示的顺序排列。

例子：获取并输出路由表项

现在举一个使用路由表的例子。这个程序的命令行参数是一个 IPv4 点分十进制数地址，它就这个地址向内核发送一个 RTM_GET 消息。内核发送在 IPv4 路由表中查找这个地址，返回一个带有其路由表项信息的 RTM_GET 消息。举例来说，如果在主机 bsdi 上执行下列代码：

```
bsdi # getrt 4.5.6.7
dest: 0.0.0.0
gateway: 206.62.226.62
netmask: 0.0.0.0
```

我们看到这个目的地址使用缺省路由（在路由表中的目的 IP 为 0.0.0.0，掩码为 0.0.0.0）。下一跳的路由器是 gw（图 1.16）。如果执行下列代码，指定主以太网为目的地址：

```
bsdi # getrt 206.62.226.32
dest: 206.62.226.32
gateway: AF_LINK, index=2
netmask: 255.255.255.224
```

目的地址就是网络本身。网关现在是发出数据的接口，返回的 sockaddr_dl 结构的接口索引为 2。

在给出源码之前，先在图 17.5 中展示向路由套接口中写入和由内核返回的数据。

我们建立一个缓冲区，其中包含 rt_msghdr 结构，接着是一个套接口地址结构，含有要内核查找的目的地址。rtm_type 为 RTM_GET，rtm_addrs 为 RTA_DST（图 17.4），表示在 rt_msghdr 结构后的唯一套接口地址结构中含有目的地址。这个命令可以用于任何协议族（提供路由表的协议族），因为要查找的地址的协议族包含在套接口地址结构中。

向内核发出消息后，读回应答，其格式如图 17.5 的右边所示：一个 rt_msghdr 结构后最多可以有 4 个套接口地址结构。返回这四个套接口地址结构中的哪此依赖于路由表项，我们是根据返回的 rt_msghdr 结构中的 rtm_addrs 成员的值知道的。每个套接口地址结构的地址族包含在 sa_family 成员中，如在前面的例子中看到的，前一次返回的网关是一个 IPv4 套接口地址结构，下面一次是一个数据链路套接口地址结构。

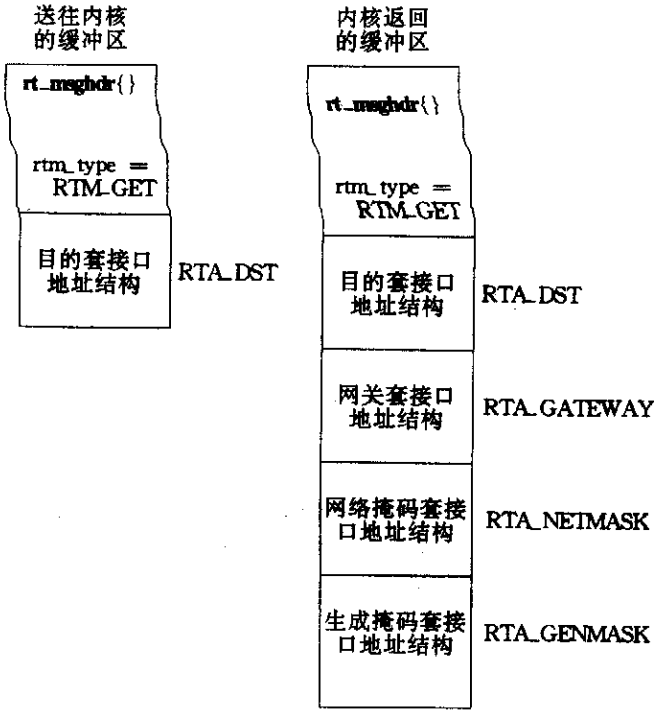


图 17.5 RTM GET 命令通过路由套接口与内核交换的数据

图 17.6 给出了这个程序的第一部分。

```
1 #include    "unproute.h"
2 #define BUFLEN     (sizeof(struct rt_msghdr) + 512)
3                          /* 8 * sizeof(struct sockaddr_in6) = 192 */
4 #define SEQ     9999
5 int
6 main(int argc, char **argv)
7 {
8     int           sockfd;
9     char          *buf;
10    pid_t         pid;
11    ssize_t       n;
12    struct rt_msghdr *rtm;
13    struct sockaddr *sa, *rti_info[RTAX_MAX];
14    struct sockaddr_in *sin;
15    if (argc != 2)
16        err_quit("usage: getrt <IPaddress>");
17    sockfd = Socket(AF_ROUTE,SOCK_RAW,0); /* need superuser privileges */
18    buf = Calloc(1, BUFLEN);      /* and initialized to 0 */
19    rtm = (struct rt_msghdr *) buf;
20    rtm->rtm_msglen = sizeof(struct rt_msghdr) + sizeof(struct sockaddr_in);
21    rtm->rtm_version = RTM_VERSION;
22    rtm->rtm_type = RTM_GET;
```

```
23      rtm->rtm_addrs = RTA_DST;
24      rtm->rtm_pid = pid = getpid();
25      rtm->rtm_seq = SEQ;

26      sin = (struct sockaddr_in *) (rtm + 1);
27      sin->sin_family = AF_INET;
28      inet_pton(AF_INET, argv[1], &sin->sin_addr);

29      Write(sockfd, rtm, rtm->rtm_msglen);

30      do {
31          n = Read(sockfd, rtm, BUFLEN);
32      } while (rtm->rtm_type != RTM_GET || rtm->rtm_seq != SEQ ||
33               rtm->rtm_pid != pid);
```

图 17.6　在路由套接口上发出 RTM_GET 命令的程序的第一部分[route/getrt.c]

第 1～3 行　unproute.h 头文件中包含一些需要的文件和 unp.h 文件。常值 BUFLEN 是我们分配用于存放发送给内核的消息和内核的应答的缓冲区的大小。这需要能存放一个 rt_msghdr 结构和可能会有的八个套接口地址结构(在一个路由套接口上可以返回的最大数目)。既然一个 IPv6 套接口地址结构的大小是 24 个字节,512 字节是足够了。

创建路由套接口

第 17 行　创建一个 AF_ROUTE 域的原始套接口,如前面提到的,这个请求需要超级用户权限。分配一个缓冲区并初始化成 0。

填写 rt_msghdr 结构

第 18～25 行　把我们的请求填写到这个结构。把进程号和我们选择的序列号存到这个结构中。我们将比较读到的应答中的这些值,找寻正确的应答。

用目的地址填写网际套接口地址结构

第 26～28 行　紧接着 rt_msghdr 结构,建立一个 sockaddr_in 结构,其中含有要内核在路由表中查找的目的 IPv4 地址。我们所设置的只是地址长度、地址族和地址本身。

向内核写消息并读取应答

第 29～32 行　向内核写消息并读取应答。因为其他进程可能打开了路由套接口,而内核会向所有路由套接口传送一个全部路由消息的拷贝,所以我们必须检查消息的类型、序列号和进程号以确保收到的消息是我们所等待的。

程序的后半部分在图 17.7 中给出。这一半处理应答。

```
34      rtm = (struct rt_msghdr *) buf;
35      sa = (struct sockaddr *) (rtm + 1);
36      get_rtaddrs(rtm->rtm_addrs, sa, rti_info);
37      if ( (sa = rti_info[RTAX_DST]) != NULL)
38          printf("dest: %s\n", Sock_ntop_host(sa, sa->sa_len));

39      if ( (sa = rti_info[RTAX_GATEWAY]) != NULL)
40          printf("gateway: %s\n", Sock_ntop_host(sa, sa->sa_len));

41      if ( (sa = rti_info[RTAX_NETMASK]) != NULL)
42          printf("netmask: %s\n", Sock_masktop(sa, sa->sa_len));

43      if ( (sa = rti_info[RTAX_GENMASK]) != NULL)
```

```
44          printf("genmask: %s\n", Sock_masktop(sa, sa->sa_len));
45      exit(0);
46  }
```

图 17.7 在路由套接口上发出 RTM_GET 命令的程序的后半部分[route/getrt.c]

第 34～35 行 rtm 指向 rt_msghdr 结构，sa 指向接在后面的第一个套接口地址结构。

第 36 行 rtm_addrs 是一个位掩码，其值对应于接在 rt_msghdr 结构后面的八种可能的套接口地址结构。get_rtaddrs 函数(下面会给出)以这个掩码和指向第一个套接口地址结构的指针(sa)为参数，在 rti_info 数组中填入指向对应的套接口地址结构的指针。假定图 17.5 中展示的全部四个套接口地址结构被内核返回，得到的 rti_info 数组如图 17.8 所示。

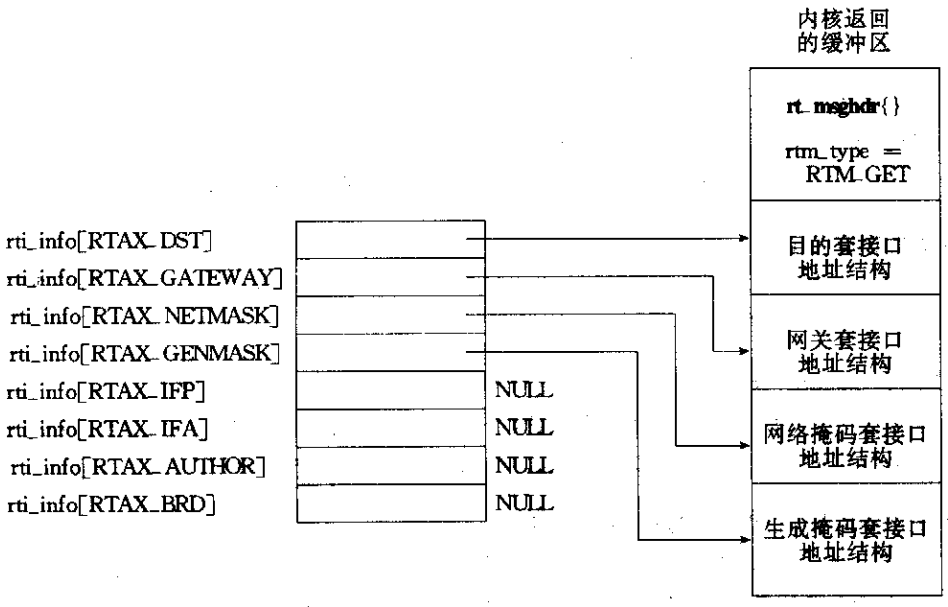


图 17.8 get_rtaddrs 函数填写的 rti_info 结构

然后检查 rti_info 数组，对数组中所有的非空指针做想做的处理。

第 37～44 行 四个可能的地址如果出现的话都被输出。调用 sock_ntop_host 函数输出目的地址和网关地址，调用 sock_masktop 输出两个掩码。我们马上给出这个新函数。

图 17.9 给出了在图 17.7 中调用的 get_rtaddrs 函数。

遍历八个可能的指针

第 17～23 行 图 17.4 中的 RTAX_MAX 为 8，它是内核在一个路由套接口上能返回的套接口地址结构的最大数目。函数里的这个循环查看图 17.4 中八个 RTA_xxx 位掩码常值中的每一个，而这些常值可能由图 17.3 中三种结构的 rtm_addrs、ifm_addrs 或 ifam_addrs 成员来设置。如果被置位，rti_info 数组中对应的元素被设置为指向相应套接口地址结构的指针，否则数组中对应的元素被设为空指针。

```
1 #include    "unproute.h"
2 /*
3  * Round up 'a' to next multiple of 'size'
4  */
5 #define ROUNDUP(a, size) (((a) & ((size)-1)) ? (1 + ((a) | ((size)-1))) : (a))
6 /*
7  * Step to next socket address structure;
8  * if sa_len is 0, assume it is sizeof(u_long).
9  */
10 #define NEXT_SA(ap)      ap = (SA *) \
11     ((caddr_t) ap + (ap->sa_len ? ROUNDUP(ap->sa_len, sizeof (u_long)) : \
12                                             sizeof(u_long)))
13 void
14 get_rtaddrs(int addrs, SA *sa, struct sockaddr **rti_info)
15 {
16     int         i;
17     for (i = 0; i < RTAX_MAX; i++) {
18         if (addrs & (1 << i)) {
19             rti_info[i] = sa;
20             NEXT_SA(sa);
21         } else
22             rti_info[i] = NULL;
23     }
24 }
```

图 17.9　在路由消息中构造指向套接口地址结构的指针数组[libroute/get_rtaddrs.c]

移到下一个套接口地址结构

第 2～12 行　套接口地址结构是变长的，但这段代码假定每个结构由一个 sa_len 成员指明它的长度。有两种复杂情况必须处理。首先，在套接口地址结构中返回的网络掩码和复制掩码的 sa_len 的值可以为 0，但实际上还是占用一个 unsigned long 的大小。(TCPv2 的第 19 章讨论了 4.4BSD 路由表的复制特性)。这个值表示一个全 0 的掩码，前面例子中展示的缺省路由的网络掩码值就是 0.0.0.0。其次，每个套接口地址结构在结尾处可能存在填充字节，使下一个结构在特定的边界上开始，在这个例子中是一个 unsigned long 的大小(譬如对 32 位体系结构就是 4 字节边界)。虽然 sockaddr_in 结构占 16 个字节，不需要填充，但是掩码的末尾经常有填充字节。

例子程序中的最后一个函数即 sock_masktop 在图 17.10 中给出，它返回能被返回的两个掩码值之一的表达字符串。掩码存在套接口地址结构中。sa_family 成员没有定义，但对 32 位的 IPv4 掩码有一个值为 0、5、6、7 或 8 的 sa_len。当这个长度大于 0 时，实际的掩码离开头的偏移和 IPv4 地址在 sockaddr_in 结构中离开头的偏移一样；离结构开头 4 个字节(如 TCPv2 第 577 页图 18.21 所示)，即通用套接口地址结构的 s_data[2]成员。

```
1 #include    "unproute.h"
2 char *
3 sock_masktop(SA *sa, socklen_t salen)
4 {
5     static char       str[INET6_ADDRSTRLEN];
6     unsigned char    *ptr = &sa->sa_data[2];
```

```
 7    if (sa->sa_len == 0)
 8        return("0.0.0.0");
 9    else if (sa->sa_len == 5)
10        snprintf(str, sizeof(str), "%d.0.0.0", *ptr);
11    else if (sa->sa_len == 6)
12        snprintf(str, sizeof(str), "%d.%d.0.0", *ptr, *(ptr+1));
13    else if (sa->sa_len == 7)
14        snprintf(str, sizeof(str), "%d.%d.%d.0", *ptr, *(ptr+1), *(ptr+2));
15    else if (sa->sa_len == 8)
16        snprintf(str, sizeof(str), "%d.%d.%d.%d",
17                 *ptr, *(ptr+1), *(ptr+2), *(ptr+3));
18    else
19        snprintf(str, sizeof(str), "(unknown mask, len = %d, family = %d)",
20                 sa->sa_len, sa->sa_family);
21    return(str);
22 }
```

图17.10 把一个掩码的值转换成它的表达格式[libroute/sock_masktop.c]

第7～21行 如果长度为0,隐含的掩码就是0.0.0.0。如果长度是5,就只存32位掩码的第一个字节,剩下的三个字节的值隐含为0。当长度为8时,掩码的四个字节都被存入。

在这个例子中我们想读内核的应答,因为应答中包含我们寻找的信息。但一般写路由套接口的write的返回值已告诉我们命令是否成功。如果这就是我们所需要的全部信息,就可以调用shutdown,其第二个参数为SHUT_RD,以防止内核发送应答。举例来说,如果删除一条路径,write返回0表示成功,返回ESRCH错误表示找不到这条路径(TCPv2第608页)。与此类似,write在增加一条路径时返回EEXIST错误表示这条路径已存在。在图17.6的例子中,如果没有所需路由表项(譬如主机上没有缺省路径),write会返回一个ESRCH错误。

17.4 sysctl操作

我们对路由套接口的主要兴趣是用sysctl函数检查路由表和接口表。创建路由套接口(一个AF_ROUTE域的原始套接口)需要超级用户权限,然而任何权限的进程都可以用sysctl检查路由表和接口列表。

```
#include <sys/param.h>
#include <sys/sysctl.h>

int sysctl(int *name, u_int namelen, void *oldp, size_t *oldlenp,
           void *newp, size_t newlen);
```

返回:成功为0,出错为-1

这个函数使用类似SNMP(简单网络管理协议)MIB(管理信息库)的名字。TCPv1的第25章详细讲述了SNMP和它的MIB。这些名字是分层结构的。

name参数是指定名字的一个整数数组,namelen是数组中的元素数目。数组的第一个元素指明请求被发往内核的哪个子系统。第二个参数指明这个子系统的某个部分,依此类

推。图 17.11 给出了前三级使用的一些常值的分层排列。

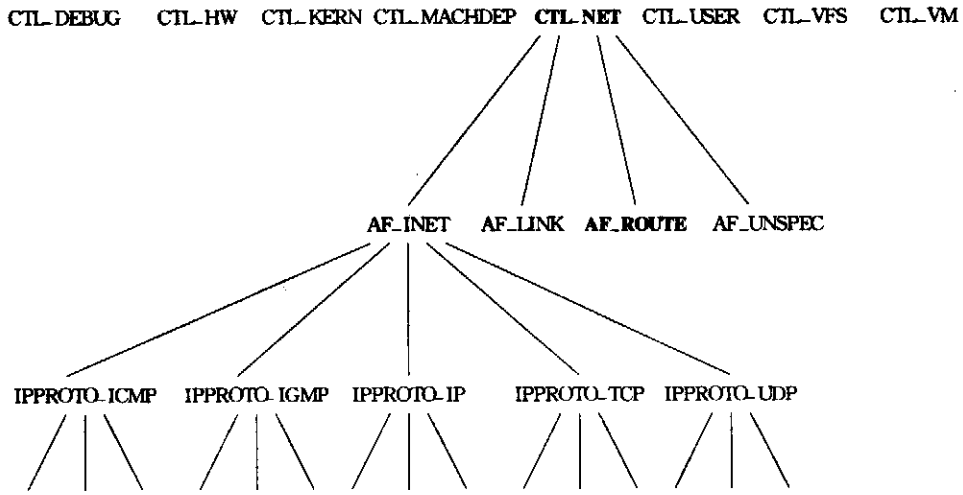


图 17.11　sysctl 名字的分层排列

要取一个值，oldp 需指向一个缓冲区，以让内核存放该值。oldlenp 是一个值-结果参数：调用函数时 oldlenp 指向的值是缓冲区的大小，返回的值是内核在缓冲区中返回的数据量。如果缓冲区不够大，就返回 ENOMEM 错误。作为一个特例，oldp 可以是一个空指针而 oldlenp 是一个非空指针，内核确定这个调用本应返回的数据量，并通过 oldlenp 返回这个值。

要设置一个新值，newp 需指向一个大小为 newlen 的缓冲区。如果没有指定新值，newp 应为一个空指针，newlen 应为 0。

sysctl 的手册页面中详细介绍了用这个函数能获取的各种系统信息：有关文件系统、虚存、内核的限制、硬件等等的信息。我们感兴趣的是网络子系统，通过把名字数组的第一个元素设为 CTL_NET 来指定。(CTL_xxx 常值在<sys/sysctl.h>头文件中定义。)第二个元素可以是：

- AF_INET：获取或设置影响网际协议的变量。下一级用一个 IPPROTO_xxx 常值指定协议。BSD/OS 3.0 在这一级提供了大约 30 个变量，用于对一些特性加以控制，如内核是否产生 ICMP 重定向消息，TCP 是否应使用 RFC 1323 选项，是否发送 UDP 校验和，等等。在本节的末尾将给出一个 sysctl 的这种用法的例子。
- AF_LINK：获取或设置链路层信息，譬如 PPP 接口的数目。
- AF_ROUTE：返回路由表或接口列表的信息。我们马上介绍这个信息。
- AF_UNSPEC：获取或设置一些套接口层的变量，譬如套接口的发送或接收缓冲区的最大容量。

当 name 数组的第二个元素为 AF_ROUTE 时，第三个元素(协议编号)总是为 0(因为 AF_ROUTE 族不像 AF_INET 族，AF_ROUTE 族里没有协议)，第四个元素是一个地址族，第五和第六级指明做什么。图 17.12 对此进行了总结。

name[]	返回 IPv4 路由表	返回 IPv4 ARP 高速缓存	返回接口列表
0	CTL_NET	CTL_NET	CTL_NET
1	AF_ROUTE	AF_ROUTE	AF_ROUTE
2	0	0	0
3	AF_INET	AF_INET	AF_INET
4	NET_RT_DUMP	NET_RT_FLAGS	NET_RT_IFLIST
5	0	RTF_LLINFO	0

图 17.12 sysctl 在 AF_ROUTE 域返回的信息

可见支持由 name[4]指明的三种操作。(NET_RT_xxx 常值在<sys/socket.h>头文件中定义。)这三种操作是通过 sysctl 调用中的 oldp 指针返回信息的。这个缓冲区中包含可变数目的 RTM_xxx 消息(图 17.2)。

1. NET_RT_DUMP 返回由 name[3]指定的地址族的路由表。如果这个地址族是 0,则返回所有地址族的路由表。

   路由表作为可变数目的 RTM_GET 消息返回,每个消息后最多可接四个套接口地址结构:本路由表项的目的地址、网关、网络掩码和复制掩码。在图 17.5 的右边展示了这些消息中的一个,图 17.7 中的代码分析这些消息。sysctl 操作的所有改动就是内核可以返回一个或多个这些信息。
2. NET_RT_FLAGS 返回由 name[3]指定的地址族的路由表,但只返回那些具有由 name[5]指定的 RTF_xxx 标志的路由表项。路由表中所有的 ARP 高速缓存项其 RTF_LLINFO 标志位都是置位的。

   信息返回的格式和前一条相同。
3. NET_RT_IFLIST 返回所有已配置接口的信息。如果 name[5]不为 0,那它就是一个接口的索引号,于是只返回这个接口的信息。(在 17.6 节中详细介绍了接口索引。)每个接口被赋予的所有地址也都返回,不过如果 name[3]不为 0 则只返回指定地址族的地址。

   给每个接口返回一个 RTM_IFINFO 消息,后面接着的是赋给该接口的每个地址对应的各个 RTM_NEWADDR 消息。RTM_IFINFO 消息后接的是一个数据链路套接口地址结构,每个 RTM_NEWADDR 消息后接最多三个套接口地址结构:接口地址、网络掩码和广播地址。在图 17.13 中展示了这两个消息。

   由于源自 4.4BSD 的内核加入了对 IPv6 的支持,因此应加入对 name[1]成员为 AF_INET6 的支持(以设置和获取 IPv6 特定变量),以及对图 17.12 中 name[3]为 AF_INET6 的支持(以列出 IPv6 路由表或 IPv6 邻居高速缓存,或者返回 IPv6 接口地址)。

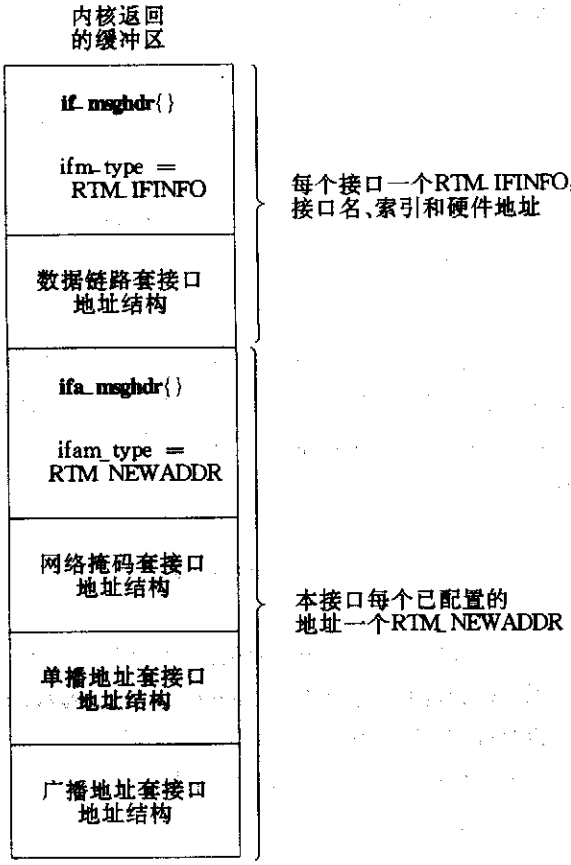


图 17.13　sysctl 的 CTL_NET,NET_RT_IFLIST 命令返回的信息

例子:判断 UDP 校验和是否打开

现在提供一个 sysctl 的简单例子,这个例子使用网际协议检查 UDP 校验和是否打开。一些 UDP 应用程序(譬如 BIND)在启动时检查 UDP 校验和是否打开,如果没有,则试图打开它。当然这需要超级用户权限来打开这样的特性,但我们现在只是检查这个特性是否打开。图 17.14 是这个程序。

```
1 #include    "unproute.h"
2 #include    <netinet/udp.h>
3 #include    <netinet/ip_var.h>
4 #include    <netinet/udp_var.h>        /* for UDPCTL_xxx constants */
5 int
6 main(int argc, char **argv)
7 {
8     int           mib[4], val;
9     size_t        len;
10    mib[0] = CTL_NET;
11    mib[1] = AF_INET;
12    mib[2] = IPPROTO_UDP;
13    mib[3] = UDPCTL_CHECKSUM;
14    len = sizeof(val);
```

```
15      Sysctl(mib, 4, &val, &len, NULL, 0);
16      printf("udp checksum flag: %d\n", val);
17      exit(0);
18  }
```

图 17.14 检查 UDP 校验和是否打开[route/checkudpsum.c]

包含头文件

第 2～4 行 我们必须包含<netinet/udp_var.h>头文件,以得到对 UDP sysctl 常值的定义。另外两个头文件是本头文件所需的。

调用 sysctl

第 10～16 行 分配一个四个元素的整数数组,存放图 17.11 中展示的各层对应的常值。因为只取变量而不是存入变量,所以把 sysctl 的 newp 参数设为空指针,newlen 参数设为 0。oldp 指向一个整数变量以存放结果,oldenp 指向的值-结果参数是这个整数的大小。输出的这个标志将为 0(关闭)或 1(打开)。

17.5 get_ifi_info 函数

现在回到 16.6 节中的例子:以一个 ifi_info 结构的链表返回所有工作的接口(图 16.5),prifinfo 程序保持原样(图 16.6),但现在给出一个 get_ifi_info 函数的新版本,用 sysctl 代替在图 16.7 中使用的 SIOCGIFCONF ioctl。

首先在图 17.15 中给出函数 net_rt_iflist。这个函数用 NET_RT_IFLIST 命令调用 sysctl,返回指定地址族的接口列表。

```
1  #include    "unproute.h"
2  char *
3  net_rt_iflist(int family, int flags, size_t *lenp)
4  {
5      int     mib[6];
6      char    *buf;
7      mib[0] = CTL_NET;
8      mib[1] = AF_ROUTE;
9      mib[2] = 0;
10     mib[3] = family;        /* only addresses of this family */
11     mib[4] = NET_RT_IFLIST;
12     mib[5] = flags;         /* interface index, or 0 */
13     if (sysctl(mib, 6, NULL, lenp, NULL, 0) < 0)
14         return(NULL);
15     if ( (buf = malloc(*lenp)) == NULL)
16         return(NULL);
17     if (sysctl(mib, 6, buf, lenp, NULL, 0) < 0) {
18         free(buf);
19     return(NULL);
20 }
21 return(buf);
22 }
```

图 17.15 调用 sysctl 返回接口列表[libroute/net_rt_iflist.c]

第 7～14 行　如图 17.12 所示，初始化数组 mib 以返回接口列表和指定族的所有已配置的地址。然后调用两次 sysctl。第一次调用时第三个参数为空，在 lenp 指向的变量中返回存放所有结构信息要用的缓冲区的大小。

第 15～21 行　给缓冲区分配空间并再次调用 sysctl，这次的第三个参数为非空。这次 lenp 指向的变量将返回存放在缓冲区中的信息量，这个变量是调用者分配的。指向这个缓冲区的指针也返回给调用者。

因为路由表的大小和接口的数目在两次 sysctl 调用之间可能改变，第一次调用返回的值在实际值上加了 10%的余量(TCPv2 第 639～640 页)。

图 17.16 给出了 get_ifi_info 函数的前半部分。

```
3 struct ifi_info *
4 get_ifi_info(int family, int doaliases)
5 {
6     int             flags;
7     char            *buf, *next, *lim;
8     size_t          len;
9     struct if_msghdr    *ifm;
10    struct ifa_msghdr   *ifam;
11    struct sockaddr *sa, *rti_info[RTAX_MAX];
12    struct sockaddr_dl  *sdl;
13    struct ifi_info *ifi, *ifisave, *ifihead, **ifipnext;
14    buf = Net_rt_iflist(family, 0, &len);
15    ifihead = NULL;
16    ifipnext = &ifihead;
17    lim = buf + len;
18    for (next = buf; next < lim; next += ifm->ifm_msglen) {
19        ifm = (struct if_msghdr *) next;
20        if (ifm->ifm_type == RTM_IFINFO) {
21            if ( ((flags = ifm->ifm_flags) & IFF_UP) == 0)
22                continue;   /* ignore if interface not up */
23            sa = (struct sockaddr *) (ifm + 1);
24            get_rtaddrs(ifm->ifm_addrs, sa, rti_info);
25            if ( (sa = rti_info[RTAX_IFP]) != NULL) {
26                ifi = Calloc(1, sizeof(struct ifi_info));
27                *ifipnext = ifi;                /* prev points to this new one */
28                ifipnext = &ifi->ifi_next;      /* ptr to next one goes here */
29                ifi->ifi_flags = flags;
30                if (sa->sa_family == AF_LINK) {
31                    sdl = (struct sockaddr_dl *) sa;
32                    if (sdl->sdl_nlen > 0)
33                        snprintf(ifi->ifi_name, IFI_NAME, "%*s",
34                                 sdl->sdl_nlen, &sdl->sdl_data[0]);
35                    else
36                        snprintf(ifi->ifi_name, IFI_NAME, "index %d",
37                                 sdl->sdl_index);
38                    if ( (ifi->ifi_hlen = sdl->sdl_alen) > 0)
```

```
39                          memcpy(ifi->ifi_haddr, LLADDR(sdl),
40                                 min(IFI_HADDR, sdl->sdl_alen));
41                  }
42              }
```

图 17.16 get_ifi_info 函数,前半部分[route/get_ifi_info.c]

第 6～14 行 声明局部变量,然后调用 net_rt_iflist 函数。

第 17～19 行 for 循环对 sysctl 返回的缓冲区中的每个路由消息进行处理。我们假定每个消息是一个 if_msghdr 结构,并查看其 ifm_type 成员。(全部三种结构的前三个成员是相同的,因此用这三种结构中的哪一个来查看类型都可以。)

检查接口是否工作

第 20～22 行 给每个接口返回一个 RTM_IFINFO 结构。如果接口不在工作则将其忽略。

判断出现的是那种套接口地址结构

第 23～24 行 sa 指向 if_msghdr 结构后的第一个套接口地址结构。get_rtaddrs 函数根据出现的是哪一种套接口地址结构来初始化 rti_info 数组。

处理接口名

第 25～42 行 如果该接口名有对应的套接口地址结构,我们就分配一个 ifi_info 结构存放接口标志。这个套接口地址结构所期望的地址族为 AF_LINK,表示它是一个数据链路 socket 地址结构。如果 sdl_nlen 成员不为 0,则把接口名拷贝到 ifi_info 结构中。否则把含有这个接口索引的字符串存为接口名。如果 sdl_alen 成员不为 0,则把硬件地址(譬如以太网地址)拷贝到 ifi_info 结构中,其长度在 ifi_hlen 中返回。

图 17.17 给出了 get_ifi_info 函数的第二部分,返回该接口的 IP 地址。

返回 IP 地址

第 43～63 行 sysctl 对该接口的每个地址返回一个 RTM_NEWADDR 消息:包括主地址和所有别名地址。如果已经给该接口填写了 IP 地址,那么我们是在进行对别名地址的处理。在这种情况下,如果调用者想要别名地址,就必须给另外一个 ifi_info 结构分配内存,拷贝已填写的字段,然后填入已返回的各个别名地址。

返回广播地址和目的地址

第 64～73 行 如果接口支持广播,则返回广播地址;如果接口是一个点对点接口,则返回目的地址。

```
43          } else if (ifm->ifm_type == RTM_NEWADDR) {
44              if (ifi->ifi_addr) {      /* already have an IP addr for i/f */
45                  if (doaliases == 0)
46                      continue;
47                      /* we have a new IP addr for existing interface */
48                  ifisave = ifi;
49                  ifi = Calloc(1, sizeof(struct ifi_info));
50                  *ifipnext = ifi;                /* prev points to this new one */
51                  ifipnext = &ifi->ifi_next;      /* ptr to next one goes here */
52                  ifi->ifi_flags = ifisave->ifi_flags;
53                  ifi->ifi_hlen = ifisave->ifi_hlen;
```

```
54                  memcpy(ifi->ifi_name, ifisave->ifi_name, IFI_NAME);
55                  memcpy(ifi->ifi_haddr, ifisave->ifi_haddr, IFI_HADDR);
56              }
57              ifam = (struct ifa_msghdr *) next;
58              sa = (struct sockaddr *) (ifam + 1);
59              get_rtaddrs(ifam->ifam_addrs, sa, rti_info);
60              if ( (sa = rti_info[RTAX_IFA]) != NULL) {
61                  ifi->ifi_addr = Calloc(1, sa->sa_len);
62                  memcpy(ifi->ifi_addr, sa, sa->sa_len);
63              }
64              if ((flags & IFF_BROADCAST) &&
65                  (sa = rti_info[RTAX_BRD]) != NULL) {
66                  ifi->ifi_brdaddr = Calloc(1, sa->sa_len);
67                  memcpy(ifi->ifi_brdaddr, sa, sa->sa_len);
68              }
69              if ((flags & IFF_POINTOPOINT) &&
70                  (sa = rti_info[RTAX_BRD]) != NULL) {
71                  ifi->ifi_dstaddr = Calloc(1, sa->sa_len);
72                  memcpy(ifi->ifi_dstaddr, sa, sa->sa_len);
73              }
74          } else
75              err_quit("unexpected message type %d", ifm->ifm_type);
76      }
77      /* "ifihead" points to the first structure in the linked list */
78      return(ifihead);    /* ptr to first structure in linked list */
79  }
```

图 17.17　get_ifi_info 函数，第二部分[route/get_ifi_info.c]

17.6　接口名和索引函数

RFC 2133[Gilligan et al. 1997]定义了四个处理接口名和索引的函数。这四个函数用于在第 19 章介绍的 IPv6 多播。基本概念是每个接口有一个唯一的名字和一个唯一的大于 0 的索引(0 从来不用做索引)。

```
#include <net/if.h>

unsigned int if_nametoindex(const char *ifname);
                                        返回:成功时为正的接口索引,出错时为 0

char *if_indextoname(unsigned int ifindex,char *ifname);
                                    返回:成功时为指向接口名的指针,出错时为 NULL

struct if_nameindex *if_nameindex(void);
                                        返回:成功时为非空指针,出错时为 NULL

void if_freenameindex(struct if_nameindex *ptr);
```

if_nametoindex 返回名为 ifname 的接口的索引。if_indextoname 对给定的 ifindex 返回一个指向其接口名的指针。ifname 参数指向一个大小为 IFNAMSIZ(在<net/if.h>头文件中定义;如图 16.2 所示)的缓冲区,调用者必须分配这个缓冲区以保存结果,成功时这个指

针也是函数的返回值。

if_nameindex 返回一个指向 if_nameindex 结构的数组的指针：

```
struct if_nameindex {
  unsigned int   if_index;  /* 1, 2... */
  char           *if_name;  /* null terminated name: "le0", ... */
};
```

数组的最后一项是一个 if_index 为 0，if_name 为空指针的结构。这个数组和数组中各元素指向的名字所用的内存是动态分配的，调用 if_freenameindex 可释放这些内存。

下面使用路由套接口提供这四个函数的一个实现。

if_nametoindex 函数

图 17.18 给出了 if_nametoindex 函数。

```
 1 #include    "unpifi.h"
 2 #include    "unproute.h"
 3 unsigned int
 4 if_nametoindex(const char *name)
 5 {
 6     unsigned int index;
 7     char        *buf, *next, *lim;
 8     size_t       len;
 9     struct if_msghdr   *ifm;
10     struct sockaddr *sa, *rti_info[RTAX_MAX];
11     struct sockaddr_dl *sdl;
12     if ( (buf = net_rt_iflist(0, 0, &len)) == NULL)
13         return(0);
14     lim = buf + len;
15     for (next = buf; next < lim; next += ifm->ifm_msglen) {
16         ifm = (struct if_msghdr *) next;
17         if (ifm->ifm_type == RTM_IFINFO) {
18             sa = (struct sockaddr *) (ifm + 1);
19             get_rtaddrs(ifm->ifm_addrs, sa, rti_info);
20             if ( (sa = rti_info[RTAX_IFP]) != NULL) {
21                 if (sa->sa_family == AF_LINK) {
22                   sdl = (struct sockaddr_dl *) sa;
23                   if (strncmp(&sdl->sdl_data[0], name, sdl->sdl_nlen) == 0) {
24                         index = sdl->sdl_index; /* save before free() */
25                         free(buf);
26                         return(index);
27                     }
28                 }
29             }
30         }
31     }
32     free(buf);
33     return(0);          /* no match for name */
34 }
```

图 17.18 给定接口的名字返回其接口索引[libroute/if_nametoindex.c]

获取接口列表

第 12～13 行　net_rt_iflist 函数返回接口列表。

只处理 RTM_INFO 消息

第 17～30 行　处理缓冲区中的消息(图 17.13),只查找 RTM_IFINFO 消息。当找到一个时,调用 get_rtaddrs 函数设置指向各套接口地址结构的指针;如果存在一个接口名结构(rti_info 数组的 RTAX_IFP 元素),就把该接口名和调用者使用的参数做一下比较。

if_indextoname 函数

下一个函数:ifindextoname,如图 17.19 所示。

```
1 #include    "unpifi.h"
2 #include    "unproute.h"
3 char *
4 if_indextoname(unsigned int index, char *name)
5 {
6     char *buf, *next, *lim;
7     size_t len;
8     struct if_msghdr   *ifm;
9     struct sockaddr    *sa, *rti_info[RTAX_MAX];
10    struct sockaddr_dl *sdl;
11    if ( (buf = net_rt_iflist(0, index, &len)) == NULL)
12        return(NULL);
13    lim = buf + len;
14    for (next = buf; next < lim; next += ifm->ifm_msglen) {
15        ifm = (struct if_msghdr *) next;
16        if (ifm->ifm_type == RTM_IFINFO) {
17            sa = (struct sockaddr *) (ifm + 1);
18            get_rtaddrs(ifm->ifm_addrs, sa, rti_info);
19            if ( (sa = rti_info[RTAX_IFP]) != NULL) {
20                if (sa->sa_family == AF_LINK) {
21                    sdl = (struct sockaddr_dl *) sa;
22                    if (sdl->sdl_index == index) {
23                        strncpy(name, sdl->sdl_data,sdl->sdl_nlen);
24                        name [sdl->sdl_nlen]=0 ; /* null terminate */
25                        free(buf);
26                        return(name);
27                    }
28                }
29            }
30        }
31    }
32    free(buf);
33    return(0);          /* no match for name */
34 }
```

图 17.19　给定接口的索引返回其接口名字[libroute/if_indextoname.c]

这个函数和前一个函数差不多一样,不过这里不是查找接口名,而是比较接口索引和调用者的参数。还有,其中调用的 net_rt_iflist 函数的第二个参数是期望的索引,因此结果应只含有期望接口的信息。如果找到匹配的接口,就返回以空字符终止的接口名。

if_nameindex 函数

下一个函数:if_nameindex,返回一个 if_nameindex 结构数组,它包含所有的接口名和索引。如图 17.20 所示。

```
1 #include    "unpifi.h"
2 #include    "unproute.h"
3 struct if_nameindex *
4 if_nameindex(void)
5 {
6     char        *buf, *next, *lim;
7     size_t      len;
8     struct if_msghdr *ifm;
9     struct sockaddr *sa, *rti_info[RTAX_MAX];
10    struct sockaddr_dl *sdl;
11    struct if_nameindex *result, *ifptr;
12    char    *namptr;
13    if ( (buf = net_rt_iflist(0, 0, &len)) == NULL)
14        return(NULL);
15    if ( (result = malloc(len)) == NULL)     /* overestimate */
16        return(NULL);
17    ifptr = result;
18    namptr = (char *) result + len;     /* names start at end of buffer */
19    lim = buf + len;
20    for (next = buf; next < lim; next += ifm->ifm_msglen) {
21        ifm = (struct if_msghdr *) next;
22        if (ifm->ifm_type == RTM_IFINFO) {
23            sa = (struct sockaddr *) (ifm + 1);
24            get_rtaddrs(ifm->ifm_addrs, sa, rti_info);
25            if ( (sa = rti_info[RTAX_IFP]) != NULL) {
26                if (sa->sa_family == AF_LINK) {
27                    sdl = (struct sockaddr_dl *) sa;
28                    namptr += sdl->sdl_nlen + 1;
29                    strncpy(namptr, &sdl->sdl_data[0], sdl->sdl_nlen);
30                    namptr[sdl->sdl_nlen] = 0;     /* null terminate */
31                    ifptr->if_name = namptr;
32                    ifptr->if_index = sdl->sdl_index;
33                    ifptr++;
34                }
35            }
36        }
37    }
38    ifptr->if_name = NULL;     /* mark end of array of structs */
39    ifptr->if_index = 0;
40    free(buf);
41    return(result);            /* caller can free() this when done */
42 }
```

图 17.20 返回所有的接口名和索引[libroute/if_nameindex.c]

获取接口列表，给结果分配空间

第 13～18 行　调用 net_rt_iflist 函数返回接口列表。用返回的大小作为将分配的缓冲区的大小，以存放返回的 if_nameindex 结构数组。这是一个过高的估计，但要比遍历两遍接口表简单：一次是计算接口的数目和名字总共使用的空间大小，另一次是填写信息。从这个缓冲区的开头往前（正向）构建 if_nameindex 数组，从缓冲区末尾往后（反向）存放接口名。

只处理 RTM_IFINFO 消息

第 22～36 行　在所有消息中查找 RTM_INFO 消息以及紧接着的数据链路套接口地址结构。把接口名和索引存到正在构建的数组中。

终止数组

第 38～39 行　把数组的最后一项的 if_name 置为空，索引置为 0。

if_freenameindex 函数

最后一个函数如图 17.21 所示，它释放给 if_nameindex 结构数组和其中的名字分配的内存。

```
43 void
44 if_freenameindex(struct if_nameindex * ptr)
45 {
46     free(ptr);
47 }
```

图 17.21　释放由 if_nameindex 分配的内存[libroute/if_nameindex.c]

这个函数很简单，因为我们把结构数组和名字存放在同一个缓冲区中。如果对每个名字都调用 malloc，释放这些内存时就将不得不遍历整个数组，给每个名字释放内存，然后释放数组本身。

17.7　小　结

我们在本书中最后遇到的套接口地址结构是 sockaddr_dl 结构，它是一种可变长度的数据链路套接口地址结构。源自 Berkeley 的内核把它们和接口联系起来，在这些结构中返回接口索引、名字和硬件地址。

进程可以向路由套接口写五种类型的消息，内核可通过路由套接口异步返回 12 种不同消息。我们给出了一个例子：进程向内核请求一个路由表项的信息，内核返回了所有的细节信息。内核的这些应答最多含有八个套接口地址结构，我们必须对它们进行分析以得到信息的各个部分。

sysctl 函数是获取和设置操作系统参数的常规方法。我们对 sysctl 感兴趣的是：

- 输出接口列表
- 输出路由表
- 输出 ARP 高速缓存

IPv6需要对套接口API做的修改包括四个接口名和索引之间的映射函数。每个接口被赋予一个唯一的大于0的索引。源自Berkeley的实现已经把索引和每个接口联系起来了,因此可以很容易地用sysctl实现这些函数。

17.8 习 题

17.1 对于一个名为eth10、链路层地址是一个64位的IEEE EUI-64地址的接口,数据链路套接口地址结构的sdl_len成员的值是多少?

17.2 图17.6中在调用write前关闭SO_USELOOPBACK套接口选项。会发生什么?

第18章 广 播

18.1 概 述

在本章和下一章中，我们分别介绍广播和多播。迄今为止，正文中所有的示例都是处理单播的：一个进程只与另一个进程对话。虽然UDP支持各种形式的地址，但TCP只支持单播地址。图18.1对各种编址方式进行比较。

类型	IPv4?	IPv6?	TCP?	UDP?	标识的接口数	递送到的接口数
单播(unicast)	•	•	•	•	一个	一个
任播(anycast)		•	尚未实现	•	一组	一组中的一个
多播(multicast)	可选	•		•	一组	一组中的全部
广播(broadcast)	•			•	全部	全部

图18.1 不同的编址方式

图中，我们同时给出了任播(anycasting)方式，这是因为IPv6将支持它。但在今天，它还是一个尚未实现的概念。RFC1546[Partridge, Mendez, and Milliken 1993]对它作了详细的描述。

图18.1的要点是：

- IPv4对多播的支持是可选的，而IPv6则是必须的。
- IPv6没有提供对广播的支持：当使用广播的IPv4应用程序移植到IPv6时，必须使用IPv6的多播方式进行重新编码。
- 广播和多播要使用UDP；二者都不能使用TCP。

广播的用途之一是假定服务器主机在本地子网上，但不知道它的单播IP地址时，对它进行定位，这就是资源发现(resource discovery)。另一用途是当有多个客户和单个服务器通信时，减少局域网上数据流量。下面是几个以此为目的使用广播的因特网应用实例。

- ARP(地址解析协议，Address Resolution Protocol)。ARP是IPv4的一个基本组成部分，而不是一个用户应用程序。ARP在本地子网上广播一个请求："具有IP地址a.b.c.d的系统请表明自己，并告诉我，你的硬件地址。"
- BOOTP(引导协议，Bootstrap Protocol)。客户假定有一台服务器主机在本地子网上。它以广播地址(通常是255.255.255.255，因为这时客户还不知道自己的IP地址、子网掩码或子网的受限广播地址)为目的地址发出自己的引导请求。
- NTP(网络时间协议，Network Time Protocol)。一种常见的情形是：一个NTP客户主机可能配置成使用一个或多个服务器主机的IP地址，其上面的NTP客户于是以某个频率(每64秒一次或更长)轮询这些服务器。客户采用基于服务器返送的时刻和到达服务器的往返时间的精确算法更新时钟。但在支持广播的局域网上，就不需要采用客户轮询服务器的方法，而代之以服务器以每64秒一次的频率向本地子网

上的所有客户广播当前时刻。这样便可以减少网络上的数据流量。

- 路由后台进程。routed 是最常用的后台进程。它输出自己的路由表的方法便是局域网广播。所有其他连接到这些局域网上的路由器便可以同时接收这些路由通告,而不用每个路由器都必须配置其邻居路由器的 IP 地址。这个特性也被局域网上的主机用于侦听路由通告并相应更新它们的路由表(许多人认为这是一种"误用")。

我们必须注意到多播可以代替广播的这两种用途(资源发现和减少网络流量)。我们将在本章的后面和下一章介绍广播存在的问题。

> 虽然广播可以减少局域网上的数据流量,但却与无盘系统有我们不希望的交互问题。假设一个 NTP 服务器以 64 秒一次的频率广播当前时刻。如果所有无盘客户主机上的 NTP 后台进程在这段时间内被换出主存,那么当无盘客户主机每 64 秒一次收到 NTP 数据报时,不得不立刻从也在该局域网上的磁盘服务器将 NTP 后台进程读回主存。这样,随着无盘客户主机上 NTP 后台进程的换进换出,局域网上每 64 秒便会出现一次数据报涌流。幸运的是随着磁盘驱动器价格的下降,无盘系统将会逐步消失。

18.2 广播地址

如果用{netid,subnetid,hostid}({网络 ID,子网 ID,主机 ID})表示 IPv4 地址,那么有四种类型的广播地址。我们用-1 表示所有比特位均为 1 的字段。

1. 子网广播地址:{netid,subnetid,-1}。这类地址编排指定子网上的所有接口。例如,如果我们对 B 类地址 128.7 采用 8 位子网 ID,那么 128.7.6.255 将是 128.7.6 子网上所有接口的子网广播地址。

   路由器通常不转发这类广播(TCPv2 第 226～227 页)。图 18.2 给出了一个连到 128.7.1 和 128.7.6 两个子网的路由器。

   路由器在 128.7.1 子网上接收到一个目的地址为 127.7.6.255(另一个接口的子网广播地址)的单播 IP 数据报。路由器通常不向 128.7.6 子网转发这个数据报。有些系统具有允许转发子网广播数据报的配置选项(TCPv1 的附录 E)。
2. 全部子网广播地址:{netid,-1,-1}。这类广播地址编排指定网络上的所有子网。如果说这类地址曾被用过的话,那么现在已很少见了。
3. 网络广播地址:{netid,-1}。这类地址用于不进行子网划分的网络。但不进行子网划分的网络现在几乎不存在了。
4. 受限广播地址:{-1,-1,-1}或 255.255.255.255。路由器从不转发目的地址为 255.255.255.255 的 IP 数据报。

在这四类广播地址中,子网广播地址是今天最常见的。但有些老系统仍然发送目的地址为 255.255.255.255 的数据报。还有些老系统不理解子网广播地址,它们仅将发往 255.255.255.255 的数据报解释为广播。

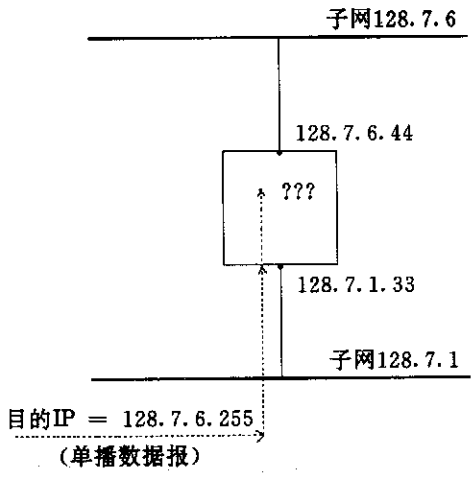


图 18.2 路由器转发子网广播吗

TFTP 和 BOOTP 等应用程序在启动过程中使用 255.255.255.255 作目的地址，因为这时它们还不知道自己的 IP 地址。

问题是：当应用进程向 255.255.255.255 发送 IP 数据报时，主机如何做？大多数主机允许发送(假定进程已经设置了 SO_BROADCAST 套接口选项)。它们在发送时，将目的地址转化为外出接口的子网广播地址。BSD/OS3.0 有一个新的套接口选项 IP_ONESBCAST。当该选项设置时，不管作为 sendto 的目标指定的广播地址是哪一类(子网广播或受限广播)，目的 IP 地址均被内核设置为 255.255.255.255。

另一个问题是：当应用进程向 255.255.255.255 发送 IP 数据报时，连接多网络的主机如何做？有些系统只在主接口(配置的第一个接口)发送目的地址为该接口的子网广播地址的单个广播数据报(TCPv2 第 736 页)。其他系统却在每个能进行广播的接口发送一个数据报拷贝。RFC1122[Braden 1989]的 3.3.6 节没有对此问题作出规定。然而，为了方便移植，如果应用进程需要在每个能进行广播的接口发送广播数据报，那它应首先获取接口的配置(16.6 节)，然后在每个能进行广播的接口上进行一次以接口的广播地址为目的地址的 sendto。

18.3 单播和广播的比较

在查看广播之前，我们应该已清楚向单播地址发送 UDP 数据报的步骤。图 18.3 给出了某个以太网上的三台主机。

图中以太网的地址为 128.7.6，其中 8 位用作子网 ID，8 位用作主机 ID。左边主机的应用程序在一个 UDP 套接口上调用 sendto 函数，将数据报发往 IP 地址 128.7.6.5、端口 7433。UDP 层附加一个 UDP 头部，并将 UDP 数据报传递到 IP 层。IP 层给它附加一个 IPv4 头部，并确定其外出接口。在以太网的情况下，将调用 ARP 来确定与目的 IP 地址相应的以太网地址：08:00:20:03:f6:42。然后，将分组作为以太网帧发送出去。以太网帧的目的地址

是上述48位地址。帧类型字段的值为0800，指示这是一个IPv4分组。IPv6帧类型字段的值为86dd。

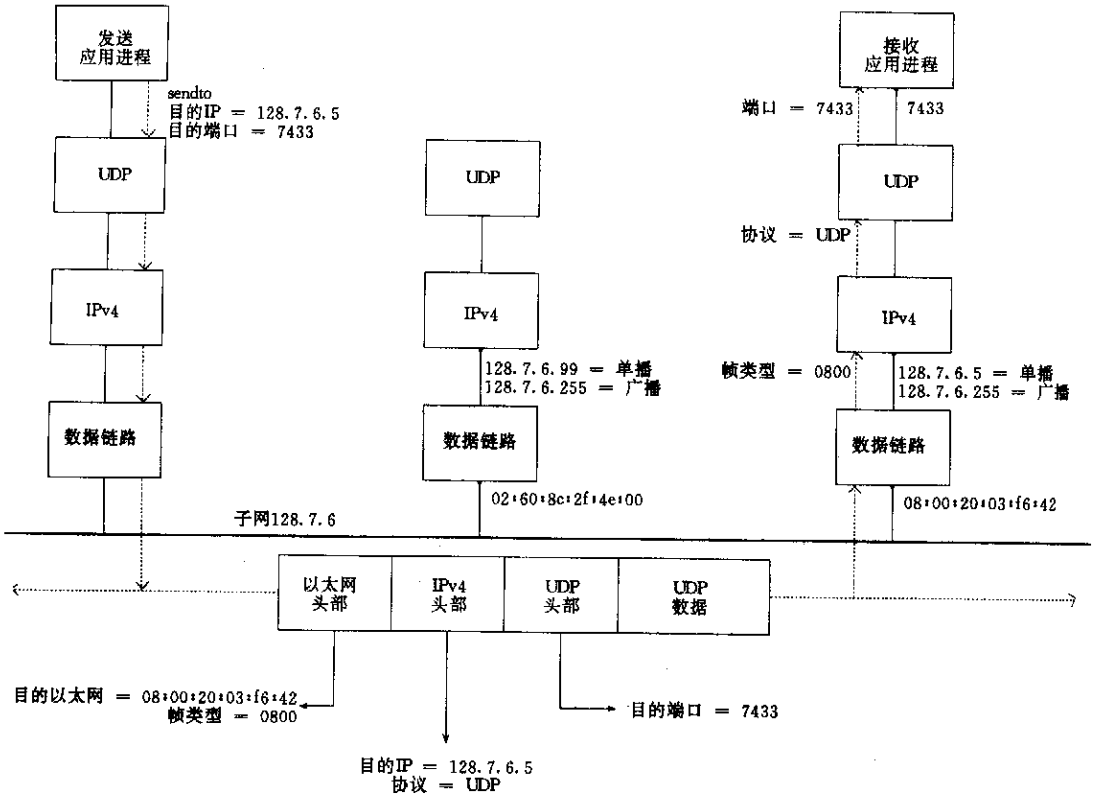


图18.3 UDP数据报单播示例

中间主机的以太网接口看到该帧，并将它的目的以太网地址与自己的以太网地址(02:60:8c:2f:4e:00)进行比较。由于二者不相等，接口便忽略该帧。因此，单播帧不会对这台主机造成任何额外开销。

右边主机的以太网接口也看到该帧。当它将该帧的目的以太网地址与自己的以太网地址进行比较时，发现二者相等，接口便读入整个帧。当帧全部读入后，可能产生一个硬件中断，设备驱动程序于是从接口内存中读取该帧。由于帧类型字段值为0800，于是将分组放入IP输入队列。

当IP层处理该分组时，它首先将目的IP地址与自己所有的IP地址进行比较。(回顾一下连接多网络的主机。也回顾一下我们在8.8节讨论的强端系统模型和弱端系统模型。)由于目的地址是其中之一，于是就接受这个分组。

然后，IP层检查IPv4头部协议字段，其值对于UDP为17。于是将IP数据报传送到UDP层。

UDP层检查其目的端口(如果其UDP套接口已连接，也可能检查源端口)，将数据报放到相应套接口的接收队列。如果需要，就唤醒进程，由进程读取这个新接收的数据报。

这个例子的关键点是单播IP数据报只能由目的IP地址指定的主机接收。子网上的其他主机不受任何影响。

我们现在考虑一个类似的例子：同样的子网，但发送进程发送的是子网广播数据报，其地址为：128.7.6.255。图 18.4 给出了这种情况。

当左侧的主机发送数据报时，它注意到目的 IP 地址是子网广播地址，于是便将它映射成 48 位的以太网地址：ff.ff.ff.ff.ff.ff。这使得子网上的每一个以太网接口都会接收该帧。在图中右侧两台运行 IPv4 的主机都接收该帧。由于以太网帧类型是 0800，两个主机都将数据报传递到 IP 层。由于目的 IP 地址匹配二者的广播地址，并且协议字段为 17(UDP)，两个主机于是都将分组上传至 UDP。

最右边的主机将 UDP 数据报传递给绑定端口 520 的应用进程。接收广播 UDP 数据报的应用进程不需要任何特殊的处理，它仅仅创建一个 UDP 套接口，并将应用的端口号捆绑到其上。(我们假设捆绑的 IP 地址为 INADDR_ANY，这是典型的情况。)

但是，中间的主机没有任何应用进程绑定 UDP 端口 520。于是主机的 UDP 代码丢弃这个已收到的数据报。这台主机禁止发送端口不可达的 ICMP 消息，因为这样做会产生广播风暴(broadcast storm)：子网上许多主机几乎同时产生响应，这将导致网络在几秒钟内不可用。

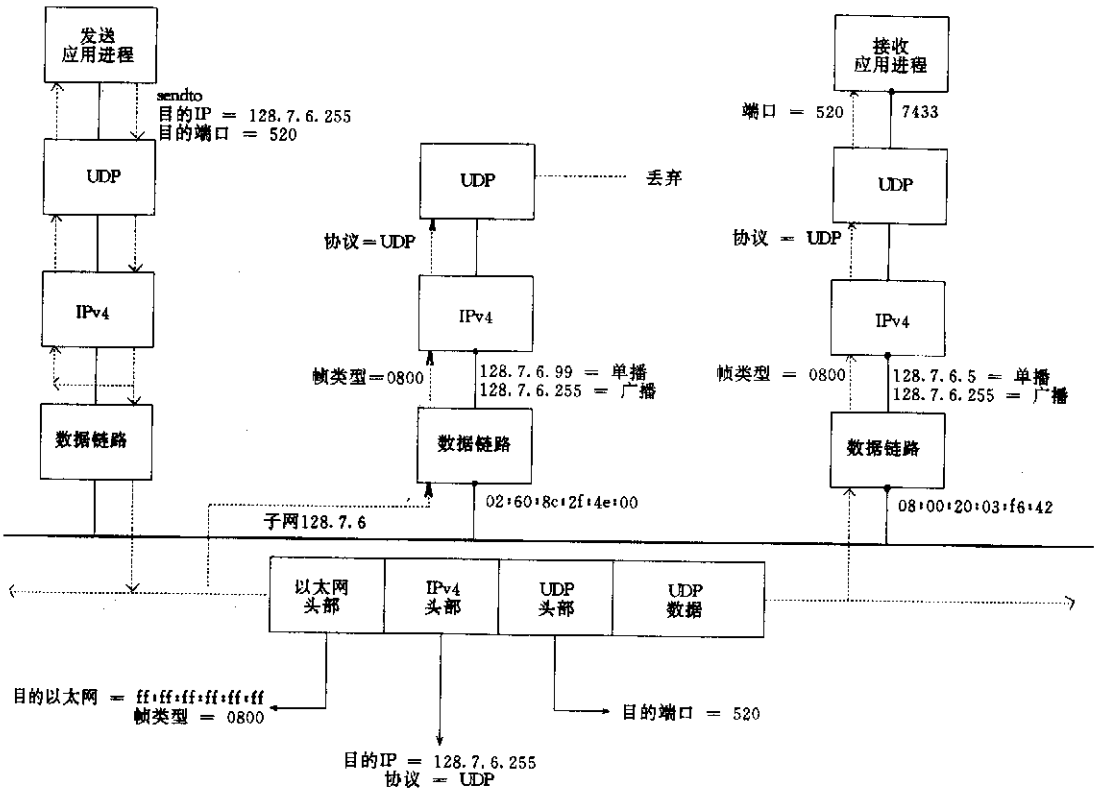


图 18.4　UDP 数据报广播示例

在图中，我们也表示出了左边主机将输出数据报又递送给自己的情况。这是一种广播属性：根据定义，广播要到达子网上的所有主机，包括发送者自身(TCPv2 第 109～110 页)。我们还假定发送应用进程已经绑定要发送到的端口(520)，因此它将收到它发送的每个数据报的拷贝。(但是一般情况下，没有将进程捆绑数据报所发送到的 UDP 端口的要求。)

在图中，我们给出了一个由IP层或数据链路层执行的逻辑回馈。这个IP层或数据链路层拷贝一份数据报(TCPv2第109～110页)，并将它沿协议栈向上发送。也可以使用物理回馈，但这在网络故障的情况下(例如未终结以太网)会引起问题。

本例也表明了广播存在的根本问题：子网上所有未参与广播应用系统的主机也必须完成对数据报的协议处理，直至在UDP层将它丢弃。(回顾一下我们对图8.21的讨论。)所有非IP主机(例如运行Novell IPX的主机)也必须在链路层接收完整的帧，并在该层将它丢弃(假定这些主机不支该帧的帧类型，IPv4分组的帧类型域值为0800)。因此以高速率产生IP数据报的应用系统(例如音频、视频应用系统)会严重影响子网上其他主机的运行。我们将在下一章看到多播是怎样解决这一问题的。

在图18.4中，我们选择UDP端口520是有意的。该端口由routed守护进程在交换RIP(路由信息协议)分组时使用。子网上所有使用RIP的路由器每30秒钟广播一次UDP数据报。如果子网上有200台主机，其中有两台使用RIP的路由器，那么其余198台主机也不得不每30秒就处理一次广播数据报(假定这198台主机都没有运行routed。)

18.4 使用广播的dg_cli函数

我们再次修改我们的dg_cli函数。这次，我们让它向标准UDP时间/日期服务器(图2.13)广播请求数据报，并且输出所有的应答。我们对main函数(图8.7)进行的仅有的修改是将目的端口改为13：

```
servaddr.sin_port = htons(13);
```

我们首先将修改后的main函数和未修改的dg_cli函数(图8.8)一起编译，然后在主机bsdi上运行。

```
bsdi % udpcli01 206.62.226.63
hi
sendto error: Permission denied
```

命令行参数是直接相连以太网的子网广播地址。我们键入一行信息后，程序调用sendto，结果返回EACCES错误。我们收到错误的原因是：除非我们显式地告诉内核我们将发送广播数据报，否则系统不允许我们进行广播。我们可以通过设置SO_BROADCAST套接口选项来做到这一点(7.5节)。

源自Berkeley的实现都要进行上述所谓的健全检查。相反，Solaris 2.5可以在没有指定该套接口选项的情况下接受目的地址为广播地址的数据报。Posix.1g的规定是内核"可以"返回错误。

对4.2BSD来说，广播是一种特权操作，它没有提供SO_BROADCAST选项。4.3 BSD增加了该选项，并且任何进程都允许设置它。

我们现在如图18.5所示修改dg_cli函数。该版本设置SO_BROADCAST选项，并输出在5秒钟内收到的所有应答。

给服务器地址分配空间，设置套接口选项

第 11～13 行　malloc 为由 recvfrom 返回的服务器地址分配空间。设置 SO_BROADCAST 套接口选项，并为 SIGALRM 信号安装信号处理程序。

从标准输入读取一行，发送至套接口，读取所有应答

第 14～24 行　以下两步即 fgets 和 sendto 与该函数的以前版本类似。但是由于我们发送的是一个广播数据报，因此我们将收到多个应答。我们用一个循环调用 recvfrom，并输出 5 秒钟内收到的所有应答。5 秒后，SIGALARM 信号产生，于是信号处理程序运行，导致 recvfrom 返回 EINTR 错误。

```
 1 #include    "unp.h"
 2 static void recvfrom_alarm(int);
 3 void
 4 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 5 {
 6     int     n;
 7     const int on = 1;
 8     char    sendline[MAXLINE], recvline[MAXLINE + 1];
 9     socklen_t  len;
10     struct sockaddr *preply_addr;
11     preply_addr = Malloc(servlen);
12     Setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
13     Signal(SIGALRM, recvfrom_alarm);
14     while (Fgets(sendline, MAXLINE, fp) != NULL) {
15         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
16         alarm(5);
17         for ( ; ; ) {
18             len = servlen;
19             n = recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
20             if (n < 0) {
21                 if (errno == EINTR)
22                     break;      /* waited long enough for replies */
23                 else
24                     err_sys("recvfrom error");
25             } else {
26                 recvline[n] = 0;   /* null terminate */
27                 printf("from %s: %s",
28                         Sock_ntop_host(preply_addr,len), recvline);
29             }
30         }
31     }
32 }
33 static void
34 recvfrom_alarm(int signo)
35 {
36     return;         /* just interrupt the recvfrom() */
37 }
```

图 18.5　使用广播的 dg_cli 函数[bcast/dgclibcast1.c]

输出收到的每个应答

第25～29行 对于收到的每个应答，我们都调用sock_ntop_host函数。该函数在IPv4下将返回包括服务器的点分十进制数IP地址在内的字符串。这些信息将和服务器的应答一道输出。

如果我们指定用子网广播地址206.62.226.63运行该程序，我们将看到以下信息：

```
bsdi % udpcli01 206.62.226.63
hi
from 206.62.226.35: Sat Jun 14 12:19:36 1997
from 206.62.226.40: Sat Jun 14 12:19:36 1997
from 206.62.226.34: Sat Jun 14 12:19:36 1997
from 206.62.226.43: Sat Jun 14 12:19:36 1997
from 206.62.226.37: Sat Jun 14 12:19:36 1997
from 206.62.226.42: Sat Jun 14 12:19:36 1997
hello
from 206.62.226.35: Sat Jun 14 12:19:43 1997
from 206.62.226.40: Sat Jun 14 12:19:43 1997
from 206.62.226.34: Sat Jun 14 12:19:43 1997
from 206.62.226.43: Sat Jun 14 12:19:43 1997
from 206.62.226.42: Sat Jun 14 12:19:43 1997
from 206.62.226.37: Sat Jun 14 12:19:43 1997
```

为产生UDP数据报输出，我们每次都必须键入一行。我们每次都接收到6个应答，其中有一个来自发送主机本身。如我们以前所说的，广播的目的主机是包括发送主机本身在内的连接到同一网络上的所有主机。所有应答数据报都是单播的，因为作为其目的地址的请求数据报源地址是一个单播地址。

由于所有系统都运行NTP，所以它们都报告同样的时间。我们看到所在子网上(图1.16)9个节点中仅有6个作了响应。其余两台主机和一台路由器都没有进行响应，因为请求是送往广播地址的。

IP分片和广播

源自Berkeley的内核不允许广播数据报分片。如果发送到广播地址的IP数据报超过外出接口的MTU，就返回EMSGSIZE错误(TCPv2第233～234页)。这是一个自BSD4.2以来就存在的策略。事实上，并没有什么原因不让广播数据报分片，只是认为广播已经给网络带来很大负担，没有必要再由于分片而使这种负担扩大数倍罢了。

我们可以从图18.5的程序中看到这种情形。我们将标准输入重定向到一个包含单个2000字节长行的文件，这会导致在以太网上传输时的分片。

```
bsdi % udpcli01 206.62.226.63 < 2000line
sendto error: Message too long
```

AIX、BSD/OS、Digital Unix及UnixWare都实施了这种限制。UnixWare虽然不发送过大的数据报，但也不返回错误信息。Linux和Solaris都允许对发送到广播地址的数据报分片。然而，为了保证程序的可移植性，需要广播的应用程序应将广播数据报限制在1472字节以内，因为对局域网来说，以太网MTU通常是最小的。

18.5 竞争状态

多个进程访问共享数据，但正确结果依赖于进程的执行顺序，这种情况我们称之为竞争状态(race condition)。由于在典型的 Unix 系统中，进程的执行顺序是不确定的，因此，有时结果正确，有时不正确。最难调试的竞争状态类型是在通常情况下结果正确，偶尔才出现不正确结果的那些。当我们在第 23 章讨论互斥变量和条件变量时，还会进一步探讨竞争状态类型。竞争状态通常是线程编程中始终要注意的一个重要问题，因为在线程中有非常之多的数据需要共享(例如所有的全程变量)。

在进行信号处理时，通常会出现各种类型的竞争状态。这是因为在我们的程序执行过程中，内核随时都会递交信号。Posix.1 虽然允许我们阻塞信号递交，但在我们进行 I/O 操作时，它几乎没有任何作用。

考察这个问题的最简单方法是看看例子。图 18.5 存在着一个竞争状态；花几分钟时间分析一下，你能否将它找出来。(提示：信号递交时，我们正在哪里运行?)你可按下述方法迫使竞争状态显现出来：将 alarm 的参数从 5 改为 1，并且在 printf 前面增加 sleep(1)。

当我们作了这些修改并键入一行输入后，这一行被作为广播数据报发送出去，同时我们给 alarm 设置了 1 秒钟报警时间。此后我们便被阻塞在 recvfrom 调用中。第一个应答可能在几个毫秒内到达套接口。它由 recvfrom 返回后，我们睡眠 1 秒钟。其他应答也将陆续到达，并被依次放入套接口的接收缓冲区内。在我们睡眠的过程中，alarm 定时器到时，产生 SIGALRM 信号：信号处理程序于是运行。它只是返回，中断阻塞我们 sleep 调用。接着，我们循环读取缓冲区内排成队的应答。每一次我们都在暂停一秒钟后，再将应答输出。但在我们读完所有的应答后，我们再次阻塞在 recvfrom 调用中。这时，定时器已不再运行，因此我们将永远阻塞在 recvfrom 中。根本问题是：我们的意图是用信号处理程序中断已阻塞的 recvfrom，但由于信号可在任何时候递交，因此当信号递交时，我们可能在无限 for 循环内的任一点运行。

我们下面讨论四个解决这个问题的方法：一个错误方法，三个正确方法。

阻塞和解阻塞信号

第一个(不正确)方法是通过在执行 for 循环其他部分时阻塞信号的递交来减小出错窗口。图 18.6 给出了这个方法。

```
1 #include    "unp.h"
2 static void recvfrom_alarm(int);
3 void
4 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
5 {
6     int         n;
7     const int on = 1;
8     char        sendline[MAXLINE], recvline[MAXLINE + 1];
9     sigset_t sigset_alrm;
10    socklen_t len;
11    struct sockaddr *preply_addr;
12    preply_addr = Malloc(servlen);
```

```
13      Setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
14      Sigemptyset(&sigset_alrm);
15      Sigaddset(&sigset_alrm, SIGALRM);
16      Signal(SIGALRM, recvfrom_alarm);
17      while (Fgets(sendline, MAXLINE, fp) != NULL) {
18          Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
19          alarm(5);
20          for ( ; ; ) {
21              len = servlen;
22              Sigprocmask(SIG_UNBLOCK, &sigset_alrm, NULL);
23              n = recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
24              Sigprocmask(SIG_BLOCK, &sigset_alrm, NULL);
25              if (n < 0) {
26                  if (errno == EINTR)
27                      break;      /* waited long enough for replies */
28                  else
29                      err_sys("recvfrom error");
30              } else {
31                  recvline[n] = 0;  /* null terminate */
32                  printf("from %s: %s",
33                          Sock_ntop_host(preply_addr, len), recvline);
34              }
35          }
36      }
37  }
38  static void
39  recvfrom_alarm(int signo)
40  {
41      return;         /* just interrupt the recvfrom() */
42  }
```

图 18.6 在 for 循环内执行时阻塞信号(不正确方法)[bcast/dgclibcast3.c]

声明信号集并初始化

第 14～15 行 声明一个信号集,并将它初始化为空集(sigemptyset),接着将与 SIGALRM 相应的位置位(sigaddset)。

解阻塞信号和阻塞信号

第 21～24 行 在调用 recvfrom 前,我们解阻塞 SIGALRM 信号(以便我们被阻塞在该调用时内核仍可以递交该信号)。接着当 recvfrom 返回时,马上阻塞该信号。如果在信号阻塞期间该信号产生(即定时器到时),内核将记下这个事件,但在信号解阻塞前不能递交信号(即调用信号处理程序)。这便是信号产生和信号递交两者的根本区别。APUE 第 10 章提供了 Posix.1 信号处理方面的详细资料。

如果编译运行该程序,它看起来工作正常,但大部分具有竞争状态的程序在大部分时间内工作正常!该程序仍存在一个问题:解阻塞信号、调用 recvfrom 和阻塞信号都是互相独立的系统调用。假定 recvfrom 返回了最后一个应答数据报并且 SIGALRM 信号在 recvfrom 和阻塞信号之间递交,下一次对 recvfrom 的调用将永远阻塞。我们虽然已缩小了出错窗口,但问题仍然存在。

这种方案的一个变体是在信号递交时设置一个全局标志。

```
static void
recvfrom_alarm(int signo)
{
    had_alarm = 1;
    return;
}
```

每次调用 alarm 时，将标志初始化为 0。dg_cli 函数在调用 recvfrom 前检查这个标志，如果标志非 0，就不进行调用。

```
for (; ;) {
    len = servlen;
    Sigprocmask(SIG_UNBLOCK,&sigset_alrm,NULL);
    if(had_alarm = 1)
        break;
    n = recvfrom(sockfd,recvline,MAXLINE,0,preply_addr,&(len);
```

如果 SIGALRM 信号是在它被阻塞期间(在前一次 recvfrom 返回后)或者它在这段代码内被解阻塞时产生的，它将在 sigprocmask 返回前递交，并设置标志。但在标志测试和 recvfrom 调用之间仍然存在一个 SIGALRM 信号可能产生并递交的时间窗，如果真地如此发生了，recvfrom 调用将永远阻塞(当然假定没有收到另外的应答)。

用 pselect 阻塞和解阻塞信号

一个正确的方法是使用 pselect(6.9 节)，如图 18.7 所示。

```
 1 #include    "unp.h"
 2 static void recvfrom_alarm(int);
 3 void
 4 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 5 {
 6     int         n;
 7     const int on = 1;
 8     char        sendline[MAXLINE], recvline[MAXLINE + 1];
 9     fd_set      rset;
10     sigset_t sigset_alrm, sigset_empty;
11     socklen_t len;
12     struct sockaddr *preply_addr;
13     preply_addr = Malloc(servlen);
14     Setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
15     FD_ZERO(&rset);
16     Sigemptyset(&sigset_empty);
17     Sigemptyset(&sigset_alrm);
18     Sigaddset(&sigset_alrm, SIGALRM);
19     Signal(SIGALRM, recvfrom_alarm);
20     while (Fgets(sendline, MAXLINE, fp) ! = NULL) {
21         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
22         Sigprocmask(SIG_BLOCK, &sigset_alrm, NULL);
23         alarm(5);
24         for ( ; ; ) {
25             FD_SET(sockfd, &rset);
26             n = pselect(sockfd+1, &rset, NULL, NULL, NULL, &sigset_empty);
27             if (n < 0) {
```

```
28                  if (errno == EINTR)
29                      break;
30                  else
31                      err_sys("pselect error");
32              } else if (n != 1)
33                      err_sys("pselect error: returned %d", n);
34              len = servlen;
35              n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
36              recvline[n] = 0;  /* null terminate */
37              printf("from %s: %s",
38                      Sock_ntop_host(preply_addr, len), recvline);
39          }
40      }
41 }
42 static void
43 recvfrom_alarm(int signo)
44 {
45      return;     /* just interrupt the recvfrom() */
46 }
```

图 18.7 使用 pselect 阻塞和解阻塞信号[bcast/dgclibcast4.c]

第 22～33 行 阻塞 SIGALRM 并调用 pselect。pselect 最后一个参数是一个指向 sigset_empty 变量的指针。sigset_empty 是一个没有任何信号阻塞的信号集，也就是说，所有的信号都是解阻塞的。pselect 操作包括：保存当前信号掩码(该信号掩码用于阻塞 SIGALRM 信号)，测试指定的描述字，如果必要将信号掩码设置为空集后阻塞调用，但在返回前将掩码恢复到 pselect 调用时的值。pselect 的关键点是设置信号掩码、测试描述字及恢复信号掩码等操作都是原子操作。

第 34～38 行 如果套接口是可读的，则调用 recvfrom。这时我们知道它不会阻塞。

正如 6.9 节提到的，pselect 是 Posix.1g 的一个新增加的调用，图 1.16 中的系统都不支持它。不过我们在图 18.8 中给出了它的一个尽管不正确然而简单的实现。给出这个不正确的实现的原因是为了说明与之相关的操作步骤：保存当前信号掩码并将掩码设置为调用者指定的值，测试描述字，然后恢复信号掩码。

```
 9 #include    "unp.h"
10 int
11 pselect(int nfds, fd_set *rset, fd_set *wset, fd_set *xset,
12         const struct timespec *ts, const sigset_t *sigmask)
13 {
14     int n;
15     struct timeval tv;
16     sigset_t savemask;
17     if (ts != NULL) {
18         tv.tv_sec = ts->tv_sec;
19         tv.tv_usec = ts->tv_nsec / 1000;     /* nanosec -> microsec */
20     }
21     sigprocmask(SIG_SETMASK, sigmask, &savemask);    /* caller's mask */
22     n = select(nfds, rset, wset, xset, (ts == NULL) ? NULL : &tv);
```

```
23      sigprocmask(SIG_SETMASK, &savemask, NULL);      /* restore mask */
24      return(n);
25  }
```

图 18.8　pselect 的一个简单但不正确的实现[lib/pselect.c]

使用 sigsetjmp 和 siglongjmp

另一个解决同一问题的正确方法是不使用信号处理程序中断被阻塞的系统调用，而从信号处理程序调用 siglongjmp。我们称之为非本地跳转(nonlocal goto)，因为使用它可以从一个函数跳转到另一个函数。图 18.9 给出了这个方案。

```
 1 #include    "unp.h"
 2 #include    <setjmp.h>
 3 static void recvfrom_alarm(int);
 4 static sigjmp_buf jmpbuf;
 5 void
 6 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 7 {
 8     int         n;
 9     const int on = 1;
10     char    sendline[MAXLINE], recvline[MAXLINE + 1];
11     socklen_t   len;
12     struct sockaddr *preply_addr;
13     preply_addr = Malloc(servlen);
14     Setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
15     Signal(SIGALRM, recvfrom_alarm);
16     while (Fgets(sendline, MAXLINE, fp) != NULL) {
17         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
18         alarm(5);
19         for ( ; ; ) {
20             if (sigsetjmp(jmpbuf, 1) != 0)
21                 break;
22             len = servlen;
23             n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
24             recvline[n] = 0;    /* null terminate */
25             printf("from %s: %s",
26                     Sock_ntop_host(preply_addr, len), recvline);
27         }
28     }
29 }
30 static void
31 recvfrom_alarm(int signo)
32 {
33     siglongjmp(jmpbuf, 1);
34 }
```

图 18.9　在信号处理程序中使用 sigsetjmp 和 siglongjmp[bcast/dgclibcast5.c]

分配跳转缓冲区

第 4 行　分配由函数和信号处理程序使用的跳转缓冲区。

调用 sigsetjmp

第 20～23 行 从 dg_cli 函数直接调用 sigsetjmp，它在建立了跳转缓冲区后返回 0。接着调用 recvfrom。

处理 SIGALRM 并调用 siglongjmp

第 30～34 行 当 SIGALRM 信号递交时，我们调用 siglongjmp。这将引起 dg_cli 函数中的 sigsetjmp 返回，返回值为第二个参数(1)，它必须为非 0 值。sigsetjmp 返回导致 dg_cli 中 for 循环的结束。

以这种方式使用 sigsetjmp 和 siglongjmp 保证了不会出现由于信号递交时间不当而引起的 recvfrom 永远阻塞现象。发生问题的唯一潜在可能是如果信号在 printf 处理输出的过程中递交。为了防止这种情况出现，应将图 18.6 中的信号阻塞和解阻塞方法同非本地跳转方法结合起来使用①。

使用从信号处理程序到函数的 IPC

还有另一个解决同一问题的正确方法。不是让信号处理程序简单地返回，期望这样能够中断被阻塞的 recvfrom，相反，我们让信号处理程序使用 IPC(进程间通信)通知 dg_cli 函数定时器到时。这与前面提到的当定时器到时时利用信号处理程序设置全局变量 had_alarm 有些相似，因为全局变量也是 IPC 的一种形式(函数和信号处理程序共享内存)。使用全局变量方法的问题在于函数必须测试该变量，如果信号几乎在测试的同时递交，就会导致时序配合问题。

我们在图 18.10 中使用的是进程内部的管道。当定时器到时时，信号处理程序就向管道中写 1 个字节。函数 dg_cli 读取该字节并以此决定什么时候终止 for 循环。使该方法成为一个好方法的原因是我们使用了 select 来测试管道是否可读。我们测试要么套接口可读，要么管道可读。

创建管道

第 15 行 我们创建一个普通的 Unix 管道，并返回两个描述字。pipefd[0]是管道的读端，pipefd[1]是管道的写端。

> 我们也可以使用套接口对来实现一个双工管道。在某些系统上，特别是 SVR4，Unix 管道总是双工的，可从任一端进行读写操作。

对套接口和管道读端进行 select 操作

第 23～30 行 对套接口和管道读端进行 select 操作。

第 45～50 行 当 SIGALRM 信号递交时，信号处理程序向管道中写入 1 个字节，使读

---

① 作者注：图 18.9 有两个潜在的时序问题。首先考虑如果信号是在 recvfrom 返回和将其返回值存入 n 之间递交，那么会发生什么现象。该数据报被认为已丢失(尽管它已由 recvfrom 收到)，不过 UDP 应用程序应该能够处理丢失的数据报。然而如果同样的技术用于 TCP 应用程序，数据就永远丢失了(因为 TCP 已确认了这个数据并把它递送给了应用进程)。
图 18.10 使用 IPC 的 dg_cli 函数也有类似的问题：信号可能在 recvfrom 成功返回和把返回值存入 n 之间递交。这可以通过在 select 返回之后关掉 alarm 来解决。另外一种办法是不用 alarm，而改用 select 的定时功能。
第二个问题是 alarm 调用和首次 sigsetjmp 调用之间的时间小于 alarm 时间(5 秒)也无法保证。解决办法之一是在调用 sigsetjmp 之后设置另外一个标志，并在信号处理程序中测试这个标志：如果该标志还没有设置，那么不调用 siglongjmp，仅仅重置 alarm 就行。
结论是：为了在这些可能的情形下保证健壮性，应避免使用 siglongjmp，而改用 pselect 或 IPC 方法。

端可读。由于信号处理程序返回时可能中断 select，因此，如果 select 返回 EINTR 错误，我们就忽略它，并且由此可知管道的读端是可读的，这样一来将终结 for 循环。

读管道

第 38～41 行 当管道的读端可读时，我们从管道中 read 信号处理程序写入的空字节，并将它忽略。但这告诉我们定时器到时，于是我们 break 出这个无限的 for 循环。

```
 1 #include    "unp.h"
 2 static void recvfrom_alarm(int);
 3 static int  pipefd[2];
 4 void
 5 dg_cli(FILE * fp, int sockfd, const SA * pservaddr, socklen_t servlen)
 6 {
 7     int          n, maxfdp1;
 8     const int on = 1;
 9     char     sendline[MAXLINE], recvline[MAXLINE + 1];
10     fd_set   rset;
11     socklen_t len;
12     struct sockaddr * preply_addr;
13     preply_addr = Malloc(servlen);
14     Setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
15     Pipe(pipefd);
16     maxfdp1 = max(sockfd, pipefd[0]) + 1;
17     FD_ZERO(&rset);
18     Signal(SIGALRM, recvfrom_alarm);
19     while (Fgets(sendline, MAXLINE, fp) ! = NULL) {
20         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
21         alarm(5);
22         for ( ; ; ) {
23             FD_SET(sockfd, &rset);
24             FD_SET(pipefd[0], &rset);
25             if ( (n = select(maxfdp1, &rset, NULL, NULL, NULL)) < 0) {
26                 if (errno == EINTR)
27                     continue;
28                 else
29                     err_sys("select error");
30             }
31             if (FD_ISSET(sockfd, &rset)) {
32                 len = servlen;
33                 n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
34                 recvline[n] = 0;  /* null terminate */
35                 printf("from %s: %s",
36                         Sock_ntop_host(preply_addr, len), recvline);
37             }
38             if (FD_ISSET(pipefd[0], &rset)) {
39                 Read(pipefd[0], &n, 1);      /* timer expired */
40                 break;
41             }
42         }
43     }
44 }
```

```
45 static void
46 recvfrom_alarm(int signo)
47 {
48     Write(pipefd[1], "", 1);     /* write 1 null byte to pipe */
49     return;
50 }
```

图 18.10 使用从信号处理程序到函数的管道作为 IPC[bcast/dgclibcast6.c]

18.6 小 结

连接到子网上的所有主机都要接收广播数据报。广播的缺点是子网上的所有主机都要处理数据报，如果是 UDP 数据报的话，需一直向上处理到 UDP 层，即使不参与广播应用系统的主机也这样。像音频、视频等高数据速率应用系统将给这些主机带来过度的处理负担。我们将在下一章看到使用多播可以解决这个问题。在多播时，只有对这个应用系统感兴趣的主机才会接收到多播数据报。

我们还举例展示了由 SIGALRM 信号带来的竞争状态。由于使用 alarm 函数和 SIGALRM 信号是对读操作设置超时的常用方法，因此这种微妙的错误在网络应用程序中是比较常见的。我们给出了解决这个问题的一个不正确的方法和三个正确方法：

- 使用 pselect
- 使用 sigsetjmp 和 siglongjmp
- 使用从信号处理程序到主循环的 IPC(典型手段是管道)

18.7 习 题

18.1 运行采用广播 dg_cli 函数(图 18.5)的 UDP 客户程序。你接收到了多少个应答？应答总是以同样的顺序到达吗？你网上的主机是否有同步时钟？

18.2 在 18.4 节末尾，我们讨论了广播数据报分片问题。我们说为了方便移植，应用程序应将广播数据报限制在 1472 字节以内。这个数据是如何得来的？

18.3 在图 18.10 中 select 的后面加上若干 printf 语句，以便观察 select 的返回值情况(返回一个错误或者两个描述字之一的可读条件)。当 alarm 到时时，你的系统返回的是 EINTR 还是管道的可读条件？

18.4 运行 tcpdump 等工具以捕获局域网上的广播数据报，所用 tcpdump 命令为 tcpdump ether broadcast。分析一下这些广播数据报属于哪些协议族。

第19章 多 播

19.1 概 述

如图18.1所示，单播地址标识单个接口，广播地址标识子网上的所有接口，多播地址标识一组接口。单播和广播是编址方案的两个极端(要么一个要么全部)，多播的目的就在于提供一种折衷方案。多播数据报仅由对该数据报感兴趣的接口接收，也就是说，由运行希望参加多播会话应用系统的主机上的接口接收。广播一般局限于局域网，而多播既可用于局域网，也可跨越广域网。事实上，基于MBone(B.2节)的应用系统每天跨越整个因特网多播。

套接口API中增加的多播支持内容比较简单：5个套接口选项，其中3个选项影响到多播UDP数据报的发送，另外2个选项影响到多播数据报的接收。

19.2 多播地址

我们必须区分IPv4多播地址和IPv6多播地址。

IPv4中的D类地址

IPv4 D类地址(从224.0.0.0到239.255.255.255)是多播地址(图A.3和A.4)。D类地址的低28位构成了多播组ID(group ID)，而整个32位地址则称为组地址(group address)。

图19.1给出了从多播地址映射到以太网地址的方法，RFC1112[Deering 1989]对此作了描述。RFC1390[Katz 1993]描述了从多播地址到FDDI网地址的映射，RFC1469[Pusateri 1993]描述了到令牌环网地址的映射。图中我们同时给出了IPv6多播地址的映射，以便比较二者的结果以太网地址。

考察一下IPv4的映射。以太网地址的高序24位总是01:00:5e。下一位总是0，低序23位是多播地址低序23位的拷贝。多播地址的高序5位在映射过程中被忽略。这意谓着多个32位多播地址会映射为单个以太网地址：映射不是一对一。

以太网地址第一个字节的低序2位说明该地址是一个统一管理的组地址。统一管理意味着高序24位由IEEE分配，组地址需由接收接口进行特别识别和处理。

下面是几个特殊的IPv4多播地址：

- 224.0.0.1是一个所有主机(all-hosts)组。子网上所有具有多播能力的主机必须在所有具有多播能力的接口上加入该组。(我们不久将谈到加入一个多播组意味着什么。)
- 224.0.0.2是一个所有路由器(all-routers)组。所有多播路由器必须在所有具有多播能力的接口加入该组。

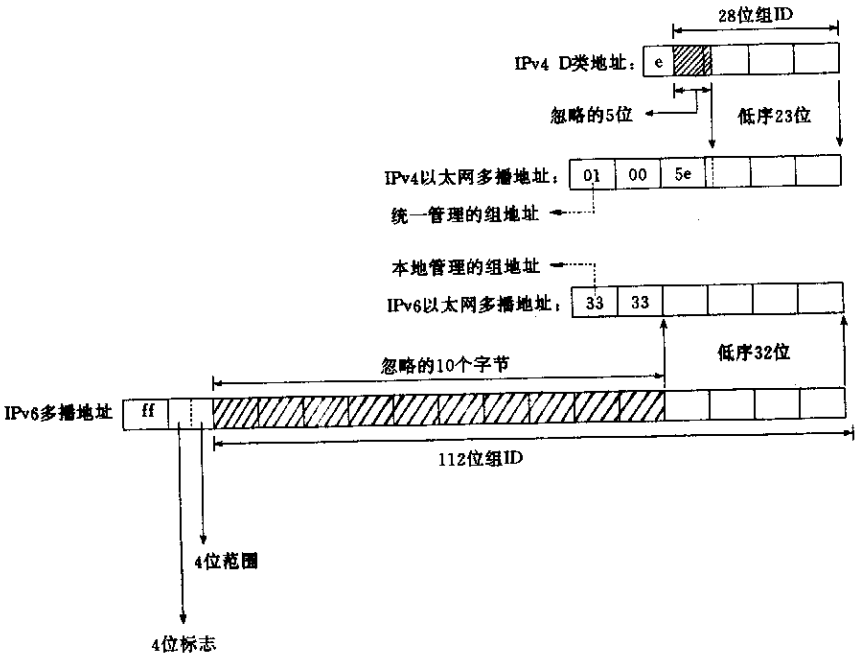


图 19.1 IPv4 和 IPv6 多播地址到以太网地址的映射

介于 224.0.0.0 到 224.0.0.255 间的地址(我们也将它写成 224.0.0.0/24)称为链路局部(link local)地址。这些地址被保留用于低级拓扑发现和维护协议。以这些地址为目的地址的数据报不能被多播路由器转发。我们将在考察了 IPv6 多播地址后再进一步讨论不同 IPv4 多播地址的范围。

IPv6 多播地址

IPv6 多播地址的高序字节值为 ff。图 19.1 给出了将 16 字节的 IPv6 多播地址映射到 6 字节以太网地址的方法。组地址的低序 32 位拷贝到以太网地址的低序 32 位。以太网的高序 2 字节值为 33:33。RFC1972[Crawford 1996a]描述了这种映射关系,RFC2019[Crawford 1996b]描述了 IPv6 组地址到 FDDI 网地址的映射,[Thomas 1997]描述了到令牌环网地址的映射。

以太网地址第一字节的低序 2 位表明该地址是一个本地管理组地址。本地管理意味着不能保证地址的唯一性。可能有除 IPv6 外的其他协议族共享同一网络并使用同样的以太网地址高序 2 字节值。正如我们前面提到的,组地址由接收接口进行特殊识别和处理。

4 位的多播标志用于区分众所周知(well-known)多播组(值为 0)和临时(transient)多播组(值为 1)。该字段的高 3 位保留。IPv6 多播地址还包含一个 4 位的范围(scope)字段,我们不久将对它加以讨论。

下面是一些特殊的 IPv6 多播地址。

- ff02::1 是一个所有节点(all-nodes)组。子网上的具有多播能力的所有主机必须在具有多播能力的所有接口上加入该组。这与 IPv4 的 224.0.0.1 地址相类似。
- ff02::2 是一个所有路由器(all-routers)组。所有子网上的多播路由器必须在具有多

播能力的所有接口上加入该组。这与IPv4的224.0.0.2多播地址相类似。

由于在映射到硬件地址时只用到IPv6多播地址的低序32位，[Hinden and Deering 1997]推荐至少在需要使用位于这32位左边的10个字节(图19.1)前，将这10个字节值置为0。

多播地址的范围

IPv6多播地址有一个4位的显式范围字段，该字段决定了多播数据报能够游走的范围。IPv6分组也有一个跳限字段，它限制分组被路由器转发的次数。下面是几个已经分配了的范围字段值：

1：节点局部即局部于节点(node-local)
2：链路局部即局部于链路(link-local)
3：网点局部即局部于网点(site-local)
8：组织局部即局部于组织(orgainization-local)
14：全球(global)

其他值或者还没有分配，或者已经保留。节点局部数据报禁止从接口输出，而链路局部数据报不能被路由器转发。网点和组织的定义由该网点或组织的多播路由器管理员决定。只是范围字段值不同的IPv6多播地址代表不同的组。

IPv4的多播数据报没有单独的范围字段。从历史上看，IPv4的TTL字段具有多播范围的含义：0意味着局部于节点，1意味着局部于链路，2～32意味着局部于网点，33～64意味着局部于区域(region-local)，65～128意味着局部于大洲(contionent-local)，129～255意味着无范围限制(全球)。这种TTL字段的双重应用导致了很多问题，[Meyer 1997]对此有详细的描述。

虽然将IPv4 TTL字段用作多播范围控制已被接受并被推荐实际使用，但可管理的范围划分是更可取的。于是239.0.0.0到239.255.255.255间的地址被规定为管理上划分范围的IPv4多播地址空间(administratively scoped IPv4 multicast space)([Meyer 1997])，它占据多播地址空间的高端。该范围内的地址由组织内部分配，但在跨越组织边界时不能保证地址的唯一性。一个组织应将其边界路由器(组织边界的多播路由器)配置为禁止转发以这些地址为目的地址的多播数据报。

我们将IPv4管理上划分范围的多播地址划分为本地范围和组织局部范围两部分。前者与IPv6的网点局部范围相似(但不是语义等价)。图19.2对不同的范围划分规则作了总结。

范围	IPv6范围	IPv4	
		TTL范围	管理范围
节点局部	1	0	
链路局部	2	1	224.0.0.0到224.0.0.255
网点局部	5	<32	239.255.0.0到239.255.255.255
组织局部	8		239.192.0.0到239.195.255.255
全球	14	<255	224.0.1.0到238.255.255.255

图19.2 IPv4和IPv6多播地址范围

19.3 局域网上多播和广播的比较

我们现在返回到图 18.3 和图 18.4 中的示例来看一下在多播的情况下发生了什么。我们以图 19.3 所示的 IPv4 为例，但 IPv6 与此相似。

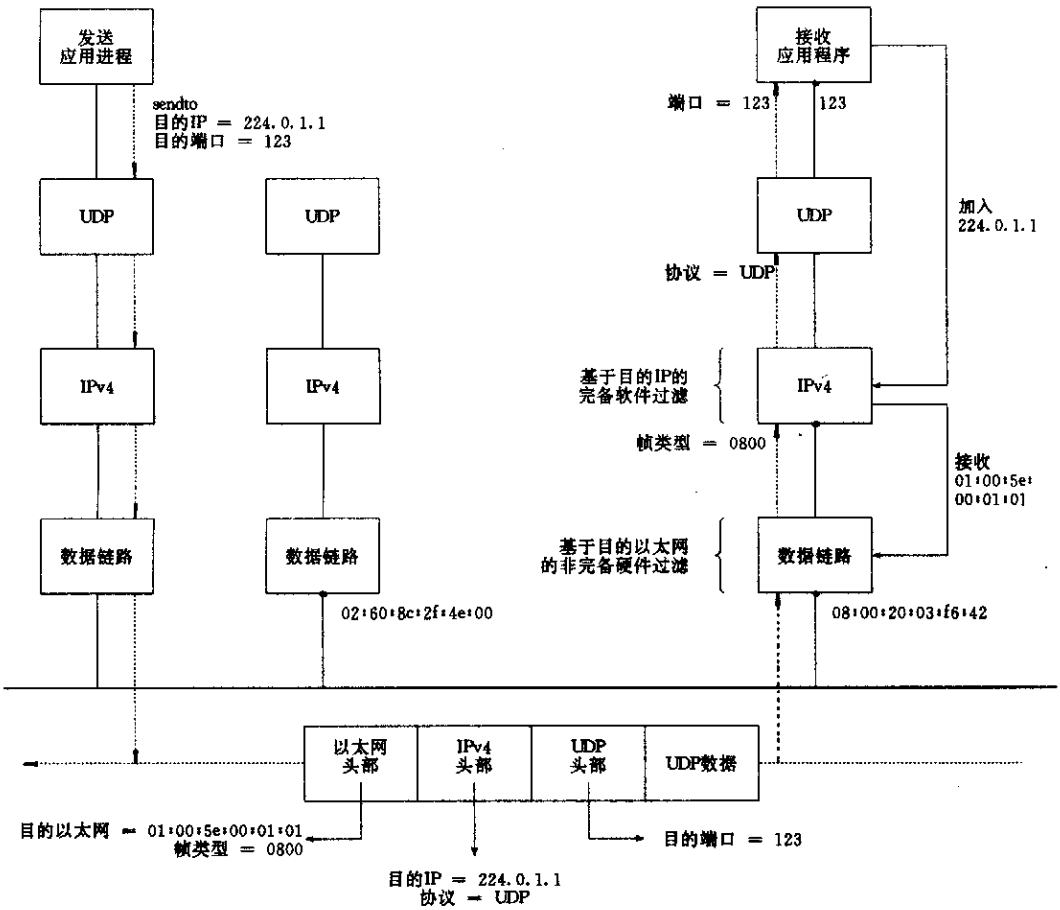


图 19.3 UDP 数据报多播示例

最右边主机的用于接收的应用进程启动，创建 UDP 套接口，捆绑端口 123，然后加入多播组 224.0.1.1。我们不久将看到这种“加入”操作是通过调用 setsockopt 完成的。在上述操作完成后，IPv4 层将这些信息保存起来，告诉适当的数据链路接收以 01:00:5e:00:01:01 为目的地址的帧(TCPv2 的 12.11 节)。该地址是上述多播地址利用图 19.1 中的方法映射来的。

下一步是最左边主机的用于发送的应用进程创建一个 UDP 套接口，并向 IP 地址 224.0.1.1 及端口 123 发送数据报。发送多播数据报不需要任何特殊处理；应用程序不需要加入多播组。发送主机将 IP 地址映射到相应的以太网目的地址，并将帧发送出去。注意帧中同时包含着目的以太网地址(由接口检查)和目的 IP 地址(由 IP 层检查)。

我们假定中间的主机不支持 IPv4 多播(在 IPv4 中支持多播是可选的)。它将完全忽略

该帧,因为(1)目的以太网地址与接口地址不匹配,(2)目的以太网地址不是以太网广播地址,(3)没有通知接口接收任何组地址(组地址的高序字节的低序位为1,如图19.1所示)。

该帧基于一种我们称之为非完备过滤(imperfect filtering)的机制被右边的数据链路接收。这种机制是接口利用以太网目的地址实现的。我们之所以说它不完备是因为当我们告诉接口接收以某个具体以太网组地址为目的地址的帧时,它也可能接收以其他以太网组地址为目的地址的帧。

> 当我们告诉以太网接口接收目的地址为某个具体以太网组地址的帧时,当前许多网卡对该地址实施一种散列(hash)函数运算,计算出一个介于0和63之间的值,并将一个64位数组的相应位置1。当一个去往某个组地址的帧在电缆上经过时,接口对其目的地址实施同样的散列函数运算,计算出一个介于0到63之间的一个值。如果数组中的相应位置1,就接收这个帧;否则就忽略这个帧。为了减少接口接收与它无关帧的可能性,较新的网络接口卡便将位数组的大小从64增加到512。随着时间的推移和越来越多的应用系统使用多播,位数组的大小可能进一步增加。现在,某些接口已经实现了完备过滤(perfect filtering)。但有些接口卡根本不进行多播过滤,当告诉接口卡接收某个具体组地址时,它必须接收所有的多播帧(有时我们称之为多播混杂(multicast promiscuous)方式)。流行的接口卡能跟拥有512位散列表一样准确地完成16个组地址的完备过滤。即使接口实现了完备过滤,IP层的完备软件过滤仍然是需要的,因为从IP多播地址到硬件地址的映射不是一对一的。

假设右边的数据链路接收该帧。由于以太网帧类型为IPv4,相应分组被传送到IP层。由于所接收分组的目的地址为多播IP地址,IP层于是将该分组的目的地址和自己的应用进程已加入的所有多播地址进行比较,如果是自己需要的分组就接受,否则就丢弃。我们称这个过程为完备过滤(perfect filtering),因为它是基于完整的32位D类地址完成的。在本例中分组被IP层接收,并被传送到UDP层,UDP层随后将相应数据报传送到绑定端口123的套接口。

下面是图19.3中没有说明的三种情况。

1. 一台运行加入到多播地址225.0.1.1的应用进程的主机。由于多播地址的高5位在映射到以太网地址的过程中被忽略,该主机的以太网地址也将接收以太网目的地址为01:00:5e:00:01:01的帧。在这种情况下,分组将在IP层由完备过滤丢弃。
2. 一台运行加入某个多播组的应用进程的主机,其多播组的相应以太网地址实施散列后恰好与01:00:5e:00:01:01的散列结果一致(也就是说接口卡执行非完备过滤)。这种帧将在数据链路层或IP层丢弃。
3. 去往同组(224.0.1.1)不同端口(譬如4000)的分组。图19.3最右边主机仍接收这类分组,但它们由IP层接受后由UDP层丢弃(假定无套接口绑定端口4000)。

> 这说明了如果一个进程要接收多播数据报,该进程必须加入相应组并绑定相应端口。

19.4 广域网上的多播

就像我们在前面讨论的，在单个局域网上的多播是简单的。一台主机发送一个多播分组，任何有兴趣的主机都会接收此数据报。多播相对于广播的优势在于会减轻其他对此多播分组不感兴趣的主机的负担。

在广域网上，多播也是有用的。考虑一下在图19.4中展示的广域网。

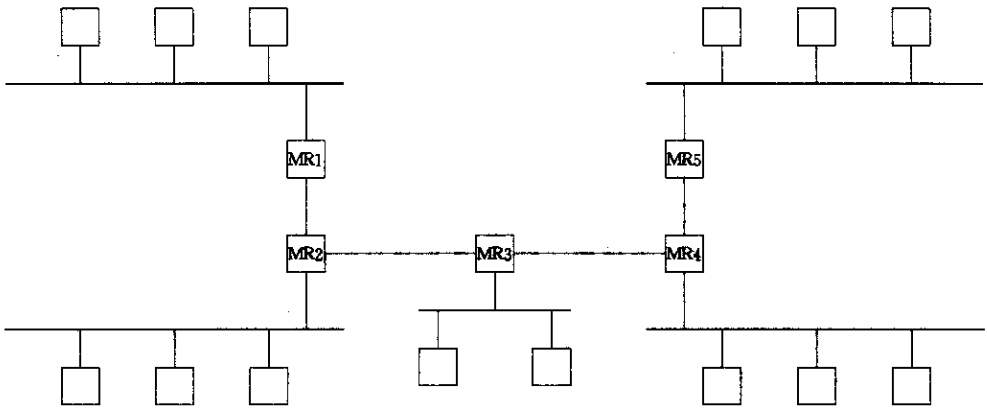


图19.4 用5台多播路由器连接的5个局域网

我们展示了用5台多播路由器连接着的5个局域网。

下面假设在其中的5台主机上启动了一些程序(例如一个监听多播音频会话的程序)且这5个程序(进程)加入了一个给定的多播组。于是这5台主机也就加入了那个多播组。我们还假设多播路由器都与相邻的多播路由器通过"多播路由协议(multicast routing protocol)"相互通信，我们用MRP指称这个协议。整个情况如图19.5所示。

当一台主机上的进程加入一个多播组时，这台主机向所有连着的多播路由器发送一个IGMP消息，通知它们本主机加入了那个组。这些多播路由器于是利用多播路由协议交换这个信息。这样每台路由器在它收到一个目的地为多播地址的分组时就知道如何处理。

> 多播路由仍是一个研究课题，其本身就能写一本书。[Maufer and Semeria 1997]提供了一个简介和概貌。

我们现在假设一个进程在左上的主机上开始发送一个到多播地址的数据报，譬如说这个进程发送一些多播接收者正等待接收的音频数据报。我们在图19.6中展示了这些数据报。

我们可以随着多播数据报从发送者到所有接收者发生的几步看一下。

- 在左上的局域网上，数据报由发送者多播。接收者H1与MR1接收到了这些数据报。(因为H1加入了这个组，而多播路由器必须接收所有多播数据报。)
- MR1将多播数据报转发给MR2，因为多播路由协议告诉MR1，MR2要接收目的地为这个组的数据报。

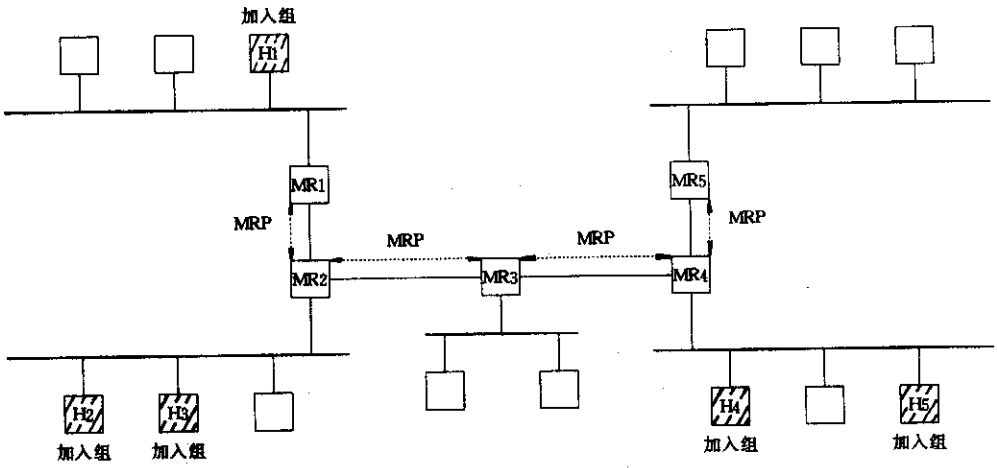


图 19.5 在广域网上 5 台主机加入一个多播组

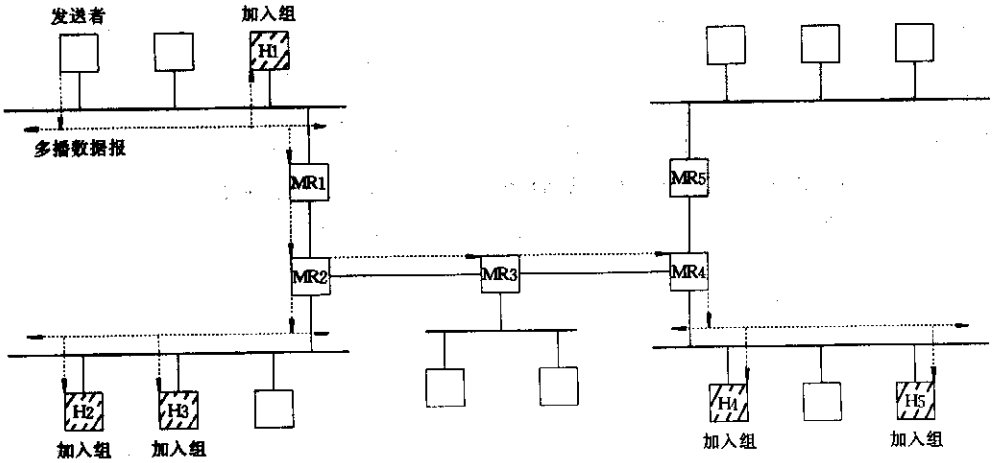


图 19.6 在广域网上发送多播数据报

- MR2 向跟它连着的局域网多播此数据报，因为 H2 和 H3 属于这个组。MR2 也向 MR3 发送一个数据报的复本。
  像 MR2 那样复制数据报(分组)是多播转发时特有的。单播分组在被路由器转发时是从不会被复制的。
- MR3 向 MR4 发送多播数据报，但它不在跟自己连着的局域网上多播，因为我们假设在这个局域网上没有主机加入这个组。
- MR4 将数据报多播到它所连接的局域网上，因为 H4 和 H5 属于这个组。它并不向 MR5 发送数据报的复本，因为在 MR5 所连的局域网上没有主机属于这个组，而且 MR4 根据与 MR5 交换的多播路由信息已经知道这一点。

在广域网上的两个稍差的多播替代方法是广播泛滥(broadcast flooding)和给每一个接收者都发送一个单独的数据报复本。在第一种方法中，数据报由发送者广播，每个路由器也要向除该数据报的到达接口以外的每一个接口广播。很显然，这将会增加对该数据报不感兴趣但又必须处理它的主机和路由器的数目。

在第二种方法中，发送者必须知道所有接收者的IP地址并且发送给每个接收者一份复本。在图19.6的5个接收者的例子中，这要求在发送者的局域网上发送5个数据报，其中从MR1到MR2走4个数据报，从MR2到MR3到MR4走2个数据报。

19.5 多播套接口选项

多播的API支持仅需要5个新的套接口选项。图19.7展示了这5个套接口选项和在IPv4和IPv6中调用getsockopt与setsockopt时所期望参数的数据类型。指向所给出数据类型变量的指针是getsockopt和setsockopt的第四个参数。所有10个(5对)选项对setsockopt都是合法的，但加入和离开多播组的4个(2对)选项在getsockopt中是不允许的。

选项名	数据类型	说明
IP_ADD_MEMBERSHIP	struct ip_mreq	加入一个多播组
IP_DROP_MEMBERSHIP	struct ip_mreq	离开一个多播组
IP_MULTICAST_IF	struct in_addr	指定外出多播数据报的外出接口
IP_MULTICAST_TTL	u_char	指定外出多播数据报的TTL
IP_MULTICAST_LOOP	u_char	使能或禁止外出多播数据报的回馈
IPV6_ADD_MEMBERSHIP	struct ipv6_mreq	加入一个多播组
IPV6_DROP_MEMBRSHIP	struct ipv6_mreq	离开一个多播组
IPV6_MULTICAST_IF	u_int	指定外出多播数据报的外出接口
IPV6_MULTICAST_HOPS	int	指定外出多播数据报的跳限
IPV6_MULTICAST_LOOP	u_int	使能或禁止外出多播数据报的回馈

图19.7 多播套接口选项

IPv4的TTL和回馈选项用u_char的参数，而IPv6的跳限和回馈选项使用int和u_int的参数。既然图7.1中大多数其他套接口选项都使用整数参数，使用IPv4多播选项时的一个常见错误就是也使用int参数指定TTL和回馈选项(这是不允许的，见TCPv2第354～355页)。IPv6对这两个选项类型的改动减少了出错的倾向。

我们现在详细地描述这5对套接口选项。注意IPv4和IPv6的各自5个选项在概念上是相同的；只是名字和参数类型不同。

IP_ADD_MEMBERSHIP和IPV6_ADD_MEMBERSHIP

在一个指定的本地接口上加入一个多播组。我们用IPv4中的单播地址或IPv6中的接口索引去指定本地接口。当加入或离开一个组时，要用到下面两个结构。

```
struct ip_mreq {
  struct in_addr imr_multiaddr;     /* IPv4 class D multicast addr */
  struct in_addr imr_interface;     /* IPv4 addr of local interface */
};
struct ipv6_mreq {
  struct in6_addr ipv6mr_multiaddr;     /* IPv6 multicast addr */
```

```
    unsigned int    ipv6mr_interface;    /* interface index, or 0 */
};
```

如果本地接口指定为通配地址(在IPv4中为INADDR_ANY)或者IPv6中的索引0,那么本地接口就由内核来选择。

如果某台主机上当前有一个或多个进程属于某个接口上的一个给定多播组,我们就称该主机在那个接口上属于所给定组。

在一个给定的套接口上可以多次加入多播组,但是每次加入的必须是一个不同的多播地址,或者虽为同一个多播地址但不在此套接口以前加入此地址的接口上。这可以用于多宿主机,例如,创建一个套接口,接着以给定的多播地址加入每一个接口。

回忆图19.2中,IPv6的多播地址有一个明确的范围字段。正像我们指出的,IPv6的多播地址仅仅范围不同就代表不同组。因此,如果一个网络时间协议的实现想接收所有的NTP数据报,不管范围,它必须加入ff01::101(节点局部)、ff02::101(链路局部)、ff05::101(网点局部)、ff08::101(组织局部)和ffoe::101(全球)。所有的加入可以在单个套接口上完成,然后设置IPV6_PKTINFO套接口选项(20.8节)去让recvmsg返回每个数据报的目的地址。

> 大多数的实现对允许每个套接口加入的(多播组-本地接口对)数目有一个限制。对源自Berkeley的实现这个限制是20。
>
> 当加入的接口没有指定时,源自Berkeley的内核在通常的IP路由表中查找多播地址,并使用查出的接口(TCPv2第357页)。有些系统在初始化时给所有多播地址(对于IPv4就是目的地址为224.0.0.0/8的路径)安装一个路径去处理这种情况①。
>
> 在IPv6中有了改变,它使用接口索引去指定接口,而不像IPv4那样使用本地单播地址,这样可允许加入未指定网络地址的(unnumbered)接口和隧道端点(tunnel endpoint)。这就是为什么我们在17.6节中介绍的接口名和索引函数在RFC 2133[Gilligan et al. 1997]中引入的原因。

IP_DROP_MEMBERSHIP 和 IPV6_DROP_MEMBERSHIP

在一个指定的本地接口上离开一个多播组。我们刚才给出的加入一个组的结构也用于这对套接口选项。如果本地接口没有指定(也就是说其值为IPv4的INADDR_ANY或者IPv6的接口索引0),那么第一个匹配的多播组的组成员关系将被去掉。

如果一个进程加入了一个组但从未明确地离开那个组,当套接口关闭时(或者显式地关闭或者因进程终止),成员关系将被自动地去掉。同一个主机上的多个进程都加入同一个组也是可以的,这种情况下,主机一直是那个组的成员,直到最后一个进程离开那个组。

IP_MULTICAST_IF 和 IPV6_MULTICAST_IF

给从本套接口上发送的外出多播数据报指定接口。这个接口在IPv4中被指定为in_addr结构,在IPv6中被指定为接口索引。如果其值为IPv4中的INADDR_ANY或IPv6中的

① 译者注: 原书中给出的IPv4的所有多播地址为224.0.0.0/8,但译者认为应是224.0.0.0/4(见19.2节)。224.0.0.0/8仅仅是链路局部多播地址的超集。

接口索引0,这将去掉以前通过这个选项指派给此套接口的接口,系统于是给每个外出数据报选择接口。

注意仔细区分当进程加入组时指定的(或由内核心选择的)本地接口(所到达的多播数据报将在该接口上接收)以及当送出多播数据报时指定的(或由内核心选择的)本地接口。

> 源自Berkeley的内核通过在通常的IP路由表中检索通往目的多播地址的路径来给多播数据报选择缺省接口。同样的技术也被用于当进程加入组时没有指定接口的情况下选择接收接口。这里假定如果存在通往给定的多播地址的路径(或许为路由表中的缺省路径),那么检索出的接口应被用于输入和输出。

IP_MULTICAST_TTL和IPV6_MULTICAST_HOPS

给外出的多播数据报设置IPv4的TTL或IPv6的跳限。如果不指定,那么缺省值都将为1,也就是限制数据报在本地子网。

IP_MULTICAST_LOOP和IPV6_MULTICAST_LOOP

打开或关闭多播数据报的本地自环即回馈。缺省时回馈是打开的:如果一个主机在发送接口上属于一个多播组,该主机上的进程发送的每个数据报的复本也会回馈,被主机当作接收的数据报处理。

这有点类似广播,一个主机上发送的广播数据报也被当作接收的数据报(图18.4)。(但在广播中是无法禁止回馈的。)这意味着如果一个进程属于它所发送数据报的多播组,它将会收到自己发送的东西。

> 在这里讨论的回馈是在IP层或更高层进行的内部回馈。如果接口听到了它自己发送的东西,RFC1112[Deering 1989]要求驱动程序忽略这些复本。该RFC还说明:回馈选项缺省情况下打开,目的是作为"一种某些上层协议的性能优化措施,而这些协议的特点是把一个组的成员限定到每台主机上的单个进程(例如路由协议)"。

前两对套接口选项(ADD_MEMBERSHIP和DROP_MEMBERSHIP)影响多播数据报的接收,后三对影响多播数据报的发送(外出接口、TTL或跳限及回馈)。我们以前提到过,发送多播数据报不需要什么特别的。如果在发送多播数据报之前没有指定多播套接口选项,数据报的外出接口将由内核来选择,TTL或跳限将为1,且有一个复本回馈回来。

进程要想接收多播数据报就必须加入那个多播组,并捆绑UDP套接口到某个端口,而这个端口将用作发到这个组的数据报的目的端口。这两个操作是不同的但都是必须的。加入这个组就是告诉所在主机的IP层和数据链路层接收发送到这个组的多播数据报。捆绑端口则是应用进程向UDP指示它想接收发送到此端口的数据报的手段。有些应用进程不仅捆绑端口,而且将多播地址也捆绑到套接口上,这样可防止此端口可能接收的其他数据报也递送到此套接口。

> 习惯上,源自Berkeley的实现仅要求主机上的某个套接口加入多播组,而不必给它捆绑端口就能接收多播数据报。但是这些实现有潜在的可能性,使多播数据报递送到并没有多播意识的应用进程。新的多播内核要求进程绑定端口并设置任意一个多播套接口选项,后一步指明应用进程意识到多播。最通常设置

的多播套接口选项是加入组。Solaris 2.5 与此有点不同，它仅递送收到的多播数据报到不但加入了组而且绑定了端口的套接口。为了移植性，所有的多播应用程序都应当加入组并捆绑端口。

有些老的能多播的主机不允许捆绑多播地址到套接口。为了移植性，应用程序可能需要忽略多播地址的 bind 调用出错。

19.6 mcast_join 和相关函数

尽管 5 对多播套接口选项对 IPv4 和 IPv6 是相似的，但仍有足够的区别使得使用多播的协议无关代码变得复杂，因为得加不少 #ifdef 语句。一个较好的解决方法是使用下面的 8 个函数隐藏这些区别。

```
#include "unp.h"
    int mcast_join(int sockfd,const struct sockaddr *sa,socklen_t salen,
                   const char *ifname,u_int ifindex);
    int mcast_leave(int sockfd,const struct sockaddr *sa,socklen_t salen);
    int mcast_set_if(int sockfd,const char *ifname,u_int ifindex);
    int mcast_set_loop(int sockfd,int flag);
    int mcast_set_ttl(int sockfd,int ttl);
                                        以上全部返回：成功时为 0，出错时为 -1
    int mcast_get_if(int sockfd);
                                        返回：成功时为非负接口索引，出错时为 -1
    int mcast_get_loop(int sockfd);
                                        返回：成功时为当前回馈标志，出错时为 -1
    int mcast_get_ttl(int sockfd);
                                        返回：成功时为当前 TTL 或跳限，出错时为 -1
```

mcast_join 函数加入一个多播组，这个组的 IP 地址在由 addr 指向的套接口地址结构中，它的长度由 salen 指定。我们可以指定在哪个接口上加入组：或者使用接口的名字（一个非空的 ifname），或者使用非零的接口索引（ifindex）。如果都没有给出，内核将选择在哪个接口上加入组。前面讲过，IPv6 中接口是通过索引指定给套接口选项的。如果已给 IPv6 套接口指定一个名字，我们就调用 if_nametoindex 函数获取其接口索引。在 IPv4 套接口选项中，接口是通过单播 IP 地址来指定的。如果已给 IPv4 套接口指定一个名字，我们就调用 SIOCGIFADDR 请求的 ioctl 函数获取其接口的单播 IP 地址。如果已给 IPv4 套接口指定一个索引，我们先调用 if_indextoname 函数获得名字，再像刚才描述的那样处理名字。

> 接口的名字（如 le0 或 ether0）是用户指定接口的通常方法，一般不直接用 IP 地址或索引。例如，tcpdump 是很少的允许用户指定接口的几个程序之一，它的 -i 选项要求以一个接口名字作为参数。

mcast_leave 离开一个多播组，这个组的 IP 地址在由 addr 指向的套接口地址结构中。

mcast_set_if 给外出多播数据报设置缺省的接口索引。如果 ifname 非空，那么它指定了接口的名字；否则，如果 ifindex 大于 0，那么它指定了接口索引。对于 IPv6，名字利用 if_nametoindex 函数映射为索引。对于 IPv4，从名字或索引到接口的单播 IP 地址的映射是通过前面描述 mcast_join 时给出的方法进行的。

mcast_set_loop 设置回馈选项为 1 或 0，mcast_set_ttl 设置 IPv4 的 TTL 或 IPv6 的跳限。另外三个 mcast_get_XXX 函数返回相应的值。

例子：mcast_join 函数

图 19.8 给出了 mcast_join 函数的前半部分。这部分处理 IPv4 的套接口。

处理索引

第 11～19 行　在套接口地址结构中的 IPv4 多播地址被复制到一个 ip_mreq 结构中。如果指定了索引，我们就调用 if_indextoname，把名字存到 ifreq 结构中。这成功后，我们往前跳转以调用 ioctl。

处理名字

第 20～27 行　调用者的名字被复制到 ifreq 结构中，SIOCGIFADDR 的 ioctl 调用返回与此名字相关联的单播地址。成功后，IPv4 地址被复制到 ip_mreq 结构中的 imr_interface 成员中。

指定缺省值

第 28～29 行　如果索引和名字都未指定，接口将被设置为通配地址，告诉内核去选择接口。

第 30～31 行　setsockopt 函数执行加入操作。

```
 1 #include    "unp.h"
 2 #include    <net/if.h>
 3 int
 4 mcast_join(int sockfd, const SA *sa, socklen_t salen,
 5              const char *ifname, u_int ifindex)
 6 {
 7     switch (sa->sa_family) {
 8     case AF_INET: {
 9             struct ip_mreq     mreq;
10             struct ifreq       ifreq;
11             memcpy(&mreq.imr_multiaddr,
12                    &((struct sockaddr_in *) sa)->sin_addr,
13                    sizeof(struct in_addr));
14             if (ifindex > 0) {
15                 if (if_indextoname(ifindex, ifreq.ifr_name) == NULL) {
16                     errno = ENXIO;     /* i/f index not found */
17                     return(-1);
18                 }
19                 goto doioctl;
20             } else if (ifname != NULL) {
21                 strncpy(ifreq.ifr_name, ifname, IFNAMSIZ);
22             doioctl:
23                 if (ioctl(sockfd, SIOCGIFADDR, &ifreq) < 0)
```

```
24                      return(-1);
25                  memcpy(&mreq.imr_interface,
26                         &((struct sockaddr_in *) &ifreq.ifr_addr)->sin_addr,
27                         sizeof(struct in_addr));
28              } else
29                  mreq.imr_interface.s_addr = htonl(INADDR_ANY);
30              return(setsockopt(sockfd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
31                                &mreq, sizeof(mreq)));
32      }
```

图 19.8 加入一个多播组:IPv4 套接口[lib/mcast_join.c]

处理 IPv6 的后半部分函数在图 19.4 给出。

处理索引、名字或缺省值

第 36～50 行 首先,IPv6 的多播地址被从套接口地址结构中复制到 ipv6_mrep 结构中。如果指定了索引,这个索引将被复制到 ipv6mr_interface 成员中。否则,如果指定了名字,则通过调用 if_nametoindex 来获取索引。两者都没有指定时,把接口索引置为 0 调用 setsockopt,告诉内核去选择接口。这就加入了组。

```
33 #ifdef  IPV6
34      case AF_INET6: {
35          struct ipv6_mreq  mreq6;
36          memcpy(&mreq6.ipv6mr_multiaddr,
37                 &((struct sockaddr_in6 *) sa)->sin6_addr,
38                 sizeof(struct in6_addr));
39          if (ifindex > 0)
40              mreq6.ipv6mr_interface = ifindex;
41          else if (ifname != NULL) {
42              if ( (mreq6.ipv6mr_interface = if_nametoindex(ifname)) == 0) {
43                  errno = ENXIO;      /* i/f name not found */
44                  return(-1);
45              }
46          } else
47              mreq6.ipv6mr_interface = 0;
48          return(setsockopt(sockfd, IPPROTO_IPV6, IPV6_ADD_MEMBERSHIP,
49                            &mreq6, sizeof(mreq6)));
50      }
51 #endif
52      default:
53          errno = EPROTONOSUPPORT;
54          return(-1);
55      }
56 }
```

图 19.9 加入一个多播组:IPv6 套接口[lib/mcast_join.c]

例子:mcast_set_loop 函数

图 19.10 给出了我们的 mcast_set_loop 函数。

由于参数是一个套接口描述字,而不是一个套接口地址结构,我们于是调用自己的 sockfd_to_family 函数来获取套接口的地址族。随后设置相应的套接口选项。

我们不再给出所有剩余的 mcast_XXX 函数的源代码,但它们是免费可得的(见前言)。

19.7 使用多播的 dg_cli 函数

我们通过仅去掉图 18.5 中 dg_cli 函数对 setsockopt 的调用来修改此函数。如前所述，如果缺省的外出接口、TTL 和回馈选项设置合适，那么发送多播数据报就不必设置多播套接口选项。我们指定所有主机组作为目地地址来运行我们的程序。

```
1 #include    "unp.h"
2 int
3 mcast_set_loop(int sockfd, int onoff)
4 {
5     switch (sockfd_to_family(sockfd)) {
6     case AF_INET: {
7         u_char      flag;
8         flag = onoff;
9         return(setsockopt(sockfd, IPPROTO_IP, IP_MULTICAST_LOOP,
10                           &flag, sizeof(flag)));
11    }
12#ifdef      IPV6
13    case AF_INET6: {
14            u_int       flag;
15            flag = onoff;
16            return(setsockopt(sockfd, IPPROTO_IPV6, IPV6_MULTICAST_LOOP,
17                              &flag, sizeof(flag)));
18    }
19 #endif
20    default:
21        errno = EPROTONOSUPPORT;
22        return(-1);
23    }
24 }
```

图 19.10 设置多播回馈选项[lib/mcast_set_loop.c]

```
solaris % udpcli01 224.0.0.1
hi there
from 206.62.226.40: Fri Jul 18 13:02:41 1997   Linux
from 206.62.226.35: Fri Jul 18 13:02:41 1997   BSD/OS
from 206.62.226.43: Fri Jul 18 13:02:41 1997   AIX
from 206.62.226.34: Fri Jul 18 13:02:41 1997   BSD/OS
from 206.62.226.42: Fri Jul 18 13:02:41 1997   Digital Unix
```

有 5 个主机响应，因为每个主机都能多播，从而都加入了所有主机组，并且都有一个进程绑定了目的 UDP 端口(13，时间/日期服务器)。因此，到来的多播数据报被递送到这个套接口。通常，这个服务器是 inetd 的一部分。每个应答都是单播，因为请求的源地址是单播地址，而这个地址又被每个服务器用作应答的目的地址。

主机 sunos5 是子网上唯一一个能多播但没有响应的主机。我们以前提到过，Solaris 需要套接口加入组，这可不是使用标准时间/日期服务器的本例子的情况。

IP 分片和多播

我们在 18.4 节末尾提到，多数系统作为一种策略不允许对广播数据报进行分片。在多播中分片是没有问题的，我们可以用同样的含有一个 2000 字节长行的文件简单地验证。

```
bsdi % udpcli01 224.0.0.1 < 2000line
from 206.62.226.35: Fri Jul 18 14:31:50 1997
from 206.62.226.34: Fri Jul 18 14:31:50 1997
from 206.62.226.40: Fri Jul 18 14:31:50 1997
from 206.62.226.43: Fri Jul 18 14:31:50 1997
from 206.62.226.42: Fri Jul 18 14:31:50 1997
```

许多源自 4.4BSD 的实现在对多播数据报分片中存在缺陷：第一个片段外的其余片段不被作为链路层的多播帧来发送。

19.8 接收 MBone 会话声明

在 MBone(B.2 节)上，为了接收一个多媒体会议，一个站点仅需知道会议的多播地址和会议数据流(例如音频和视频)的 UDP 端口号。SAP(会话声明协议[Handley 1996])描述这样做的方法(声明被多播到 MBone 上所用的数据报头部和频率)，SDP(会话描述协议[Handley and Jacobson 1997])则描述声明的内容(如何指定多播地址和 UDP 端口号)。想在 MBone 上声明会话的站点周期性地向一个众所周知的多播组和 UDP 端口发送包含其会话描述信息的多播数据报。在 MBone 上的站点运行一个叫 sdr 的程序去接收这些声明。这个程序做许多工作：不仅接收会话声明，而且提供一个交互的用户界面，显示信息并让用户发送声明。

在这一节，我们开发一个简单的程序，它只接收那些会话声明，我们用它来作为简单的多播接收程序的例子。我们的目的是展示多播接收的简单性，而不是深入到这个应用程序的细节。

图 19.11 给出了我们的 main 程序，它接收定期的 SAP/SDP 声明。

众所周知的名字和众所周知的端口

第 2～3 行　分配给 SAP 声明的多播地址是 224.2.127.254，它的名字是 sap.mcast.net。所有的众所周知多播地址(见 ftp://ftp.isi.edu/in-notes/iana/assignments/multicast-addresses)都在 DNS 中的 mcast.net 层次下。众所周知的 UDP 端口号是 9875。

创建 UDP 套接口

第 12～17 行　我们调用自己的 udp_client 函数去查看名字和端口，它将信息填入合适的套接口地址结构中。如果没有指定命令行参数，我们就使用缺省值；否则我们从命令行参数中取得多播地址、端口号和接口名字。

```
1 #include    "unp.h"
2 #define     SAP_NAME      "sap.mcast.net"    /* default group name and port */
3 #define     SAP_PORT      "9875"
4 void  loop(int, socklen_t);
5 int
6 main(int argc, char **argv)
```

```
 7 {
 8     int          sockfd;
 9     const int on = 1;
10     socklen_t salen;
11     struct sockaddr *sa;
12     if (argc == 1)
13         sockfd = Udp_client(SAP_NAME, SAP_PORT, (void **) &sa, &salen);
14     else if (argc == 4)
15         sockfd = Udp_client(argv[1], argv[2], (void **) &sa, &salen);
16     else
17         err_quit("usage: mysdr <mcast-addr> <port#> <interface-name>");
18     Setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
19     Bind(sockfd, sa, salen);
20     Mcast_join(sockfd, sa, salen, (argc == 4) ? argv[3] : NULL, 0);
21     loop(sockfd, salen);     /* receive and print */
22     exit(0);
23 }
```

图19.11 接收SAP/SDP声明的main程序[mysdr/main.c]

捆绑端口

第18～19行 我们设置SO_REUSEADDR套接口选项以允许这个程序的多个实例在单台主机上运行，然后捆绑端口到这个套接口上。通过捆绑多播地址到套接口上，我们防止这个套接口接收其他的可能被其端口接收的UDP数据报。捆绑这个多播地址不是必须的，但它提供了由内核来过滤我们不想要的分组的功能。

加入多播组

第20行 我们调用mcast_join函数来加入这个组。如果接口名字已被作为命令行参数指定，它将会传递给我们的函数；否则我们让内核去选择在哪个接口上加入组。

第21行 我们调用在图19.12中给出的loop函数来读取和输出所有的声明。

```
 1 #include    "unp.h"
 2 void
 3 loop(int sockfd, socklen_t salen)
 4 {
 5     char     buf[MAXLINE+1];
 6     socklen_t    len;
 7     ssize_t n;
 8     struct sockaddr    *sa;
 9     struct sap_packet {
10         uint32_t   sap_header;
11         uint32_t   sap_src;
12         char       sap_data[1];
13     } *sapptr;
14     sa = Malloc(salen);
15     for ( ; ; ) {
16         len = salen;
17         n = Recvfrom(sockfd, buf, MAXLINE, 0, sa, &len);
18         buf[n] = 0;    /* null terminate */
19         sapptr = (struct sap_packet *) buf;
20         if ( (n == 2 * sizeof(uint32_t)) <= 0)
```

```
21              err_quit("n = %d", n);
22          printf("From %s\n%s\n", Sock_ntop(sa, len), sapptr->sap_data);
23      }
24  }
```

图 19.12 接收并输出 SAP/SDP 声明的循环[mysdr/loop.c]

数据报格式

第 9～13 行 sap_packet 描述了 SDP 数据报：一个 32 位的 SAP 头部，后跟一个 32 位的源地址，再后跟实际的声明。声明仅仅是 ISO 8859-1 的文本行，不能超过 1024 字节。在每个 UDP 数据报中只允许有一个会话声明。

读入 UDP 数据报，输出发送者和内容

第 15～23 行 recvfrom 等待下一个到我们套接口的 UDP 数据报。当一个数据报到后，我们在缓冲区末端加一个空字节并且跳过两个头部字段，接着输出结果。我们还输出了发送多播声明的发送者的 IP 地址和端口号。

图 19.3 给出了我们程序的一些典型输出。

```
solaris % mysdr
From 128.102.84.134/2840
v=0
o=shuttle 3050400397 3051818822 IN IP4 128.102.84.134
s=NASA - Shuttle STS-79 Mission Coverage
i=Pre-launch and mission coverage of Shuttle Mission STS-79.  Launch expected 9/16/96
  with a mission duration anticipated of 9-10 days.  STS-79 is the 4th in a series of joint
  docking missions between the Shuttle and the MIR Space Station. This session is being
  offered by NASA - Ames Research Center as a public service to the MBone community.
u=http://www-pao.ksc.nasa.gov/kscpao/kscpao.htm
p=NASA ARC Digital Video Lab (415) 604-6145
e=NASA ARC Digital Video Lab <mallard@mail.arc.nasa.gov>
c=IN IP4 224.2.86.28/127
t=3051608400 3052472400
m=audio 19432 RTP/AVP 0
m=video 61192 RTP/AVP 31
```

图 19.13 典型的 SAP/SDP 声明

这个声明描述了在 MBone 上航天飞机任务的 NASA 报道。SDP 会话描述包含很多形如 type=value 的行，type 总是一个字符而且是区分大小写的。value 是一个依赖于 type 的有结构的文本串。等号两边不允许有空格。

v=0 是版本。

o=是信息源。shuttle 是用户名，3050400397 是会话 ID，3051818822 是这个声明的版本号，IN 是网络的类型，IP4 是地址类型，128.102.84.134 是地址。由用户名、会话 ID、网络类型、地址类型和地址组成的五元组为会话构成了一个唯一标识符。

s=定义会话名字，i=是有关会话的信息。我们把信息按每 80 字节折了行。u=为有关会话的详细信息提供一个 URI(统一资源标识符)，p=和 e=则提供会议负责人的电话号码和电子邮件地址。

c=提供连接信息，在这个例子中它表明了连接是基于 IP 的，用 IPv4，多播地址是 224.2.86.28，TTL 值为 127。t=提供起始时间和停止时间，都是 NTP 单位的，从 1900 年 1 月 1