提示

文件描述符的变化

文件描述符变化是指监视的文件描述符中发生了相应的监视事件。例如，通过 select 的第二个参数传递的集合中存在需要读数据的描述符时，就意味着文件描述符发生变化。

select函数返回正整数时，怎样获知哪些文件描述符发生了变化？向select函数的第二到第四个参数传递的fd_set变量中将产生如图12-8所示变化，获知过程并不难。

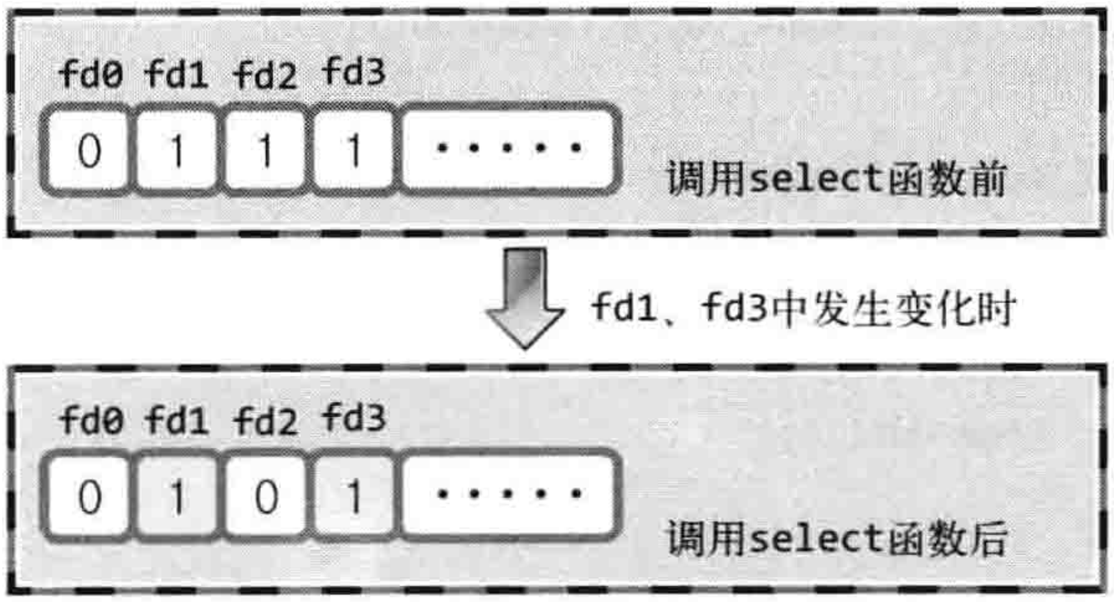

图12-8 fd_set变量的变化

由图12-8可知，select函数调用完成后，向其传递的fd_set变量中将发生变化。原来为1的所有位均变为0，但发生变化的文件描述符对应位除外。因此，可以认为值仍为1的位置上的文件描述符发生了变化。

select 函数调用示例

下面通过示例把select函数所有知识点进行整合，希望各位通过如下示例完全理解之前的内容。

❖ select.c

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.  #include <sys/time.h>
4.  #include <sys/select.h>
5.  #define BUF_SIZE 30
6.
7.  int main(int argc, char *argv[])
8.  {
9.      fd_set reads, temps;
10.     int result, str_len;
11.     char buf[BUF_SIZE];
12.     struct timeval timeout;
13.
```

```
14.     FD_ZERO(&reads);
15.     FD_SET(0, &reads); // 0 is standard input(console)
16.
17.     /*
18.     timeout.tv_sec=5;
19.     timeout.tv_usec=5000;
20.     */
21.
22.     while(1)
23.     {
24.         temps=reads;
25.         timeout.tv_sec=5;
26.         timeout.tv_usec=0;
27.         result=select(1, &temps, 0, 0, &timeout);
28.         if(result==-1)
29.         {
30.             puts("select() error!");
31.             break;
32.         }
33.         else if(result==0)
34.         {
35.             puts("Time-out!");
36.         }
37.         else
38.         {
39.             if(FD_ISSET(0, &temps))
40.             {
41.                 str_len=read(0, buf, BUF_SIZE);
42.                 buf[str_len]=0;
43.                 printf("message from console: %s", buf);
44.             }
45.         }
46.     }
47.     return 0;
48. }
```

代码说明

- 第14、15行：看似复杂，实则简单。首先在第14行初始化fd_set变量，第15行将文件描述符0对应的位设置为1。换言之，需要监视标准输入的变化。
- 第24行：将准备好的fd_set变量reads的内容复制到temps变量，因为之前讲过，调用select函数后，除发生变化的文件描述符对应位外，剩下的所有位将初始化为0。因此，为了记住初始值，必须经过这种复制过程。这是使用select函数的通用方法，希望各位牢记。
- 第18、19行：请观察被注释的代码，这是为了设置select函数的超时而添加的。但不能在此时设置超时。因为调用select函数后，结构体timeval的成员tv_sec和tv_usec的值将被替换为超时前剩余时间。因此，调用select函数前，每次都需要初始化timeval结构体变量。
- 第25、26行：将初始化timeval结构体的代码插入循环后，每次调用select函数前都会初始化新值。
- 第27行：调用select函数。如果有控制台输入数据，则返回大于0的整数；如果没有输入数

据而引发超时，则返回0。

•第39~44行：select函数返回大于0的值时运行的区域。验证发生变化的文件描述符是否为标准输入。若是，则从标准输入读取数据并向控制台输出。

❖ 运行结果：select.c

```
root@my_linux:/tcpip# gcc select.c -o select
root@my_linux:/tcpip# ./select
Hi~
message from console: Hi~
Hello~
message from console: Hello~
Time-out!
Time-out!
Good bye~
message from console: Good bye~
```

运行后若无任何输入，经5秒将发生超时。若通过键盘输入字符串，则可看到相同字符串输出。

实现 I/O 复用服务器端

下面通过select函数实现I/O复用服务器端。之前已给出关于select函数的所有说明，各位只需通过示例掌握利用select函数实现服务器端的方法。下列示例是基于I/O复用的回声服务器端。

❖ echo_selectserv.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <unistd.h>
5.  #include <arpa/inet.h>
6.  #include <sys/socket.h>
7.  #include <sys/time.h>
8.  #include <sys/select.h>
9.
10. #define BUF_SIZE 100
11. void error_handling(char *buf);
12.
13. int main(int argc, char *argv[])
14. {
15.     int serv_sock, clnt_sock;
16.     struct sockaddr_in serv_adr, clnt_adr;
17.     struct timeval timeout;
18.     fd_set reads, cpy_reads;
19.
20.     socklen_t adr_sz;
```

```
21.     int fd_max, str_len, fd_num, i;
22.     char buf[BUF_SIZE];
23.     if(argc!=2) {
24.         printf("Usage : %s <port>\n", argv[0]);
25.         exit(1);
26.     }
27.
28.     serv_sock=socket(PF_INET, SOCK_STREAM, 0);
29.     memset(&serv_adr, 0, sizeof(serv_adr));
30.     serv_adr.sin_family=AF_INET;
31.     serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
32.     serv_adr.sin_port=htons(atoi(argv[1]));
33.
34.     if(bind(serv_sock, (struct sockaddr*) &serv_adr, sizeof(serv_adr))==-1)
35.         error_handling("bind() error");
36.     if(listen(serv_sock, 5)==-1)
37.         error_handling("listen() error");
38.
39.     FD_ZERO(&reads);
40.     FD_SET(serv_sock, &reads);
41.     fd_max=serv_sock;
42.
43.     while(1)
44.     {
45.         cpy_reads=reads;
46.         timeout.tv_sec=5;
47.         timeout.tv_usec=5000;
48.
49.         if((fd_num=select(fd_max+1, &cpy_reads, 0, 0, &timeout))==-1)
50.             break;
51.         if(fd_num==0)
52.             continue;
53.
54.         for(i=0; i<fd_max+1; i++)
55.         {
56.             if(FD_ISSET(i, &cpy_reads))
57.             {
58.                 if(i==serv_sock)    // connection request!
59.                 {
60.                     adr_sz=sizeof(clnt_adr);
61.                     clnt_sock=
62.                         accept(serv_sock, (struct sockaddr*)&clnt_adr, &adr_sz);
63.                     FD_SET(clnt_sock, &reads);
64.                     if(fd_max<clnt_sock)
65.                         fd_max=clnt_sock;
66.                     printf("connected client: %d \n", clnt_sock);
67.                 }
68.                 else    // read message!
69.                 {
70.                     str_len=read(i, buf, BUF_SIZE);
71.                     if(str_len==0)   // close request!
72.                     {
73.                         FD_CLR(i, &reads);
74.                         close(i);
```

```
75.                         printf("closed client: %d \n", i);
76.                     }
77.                     else
78.                     {
79.                         write(i, buf, str_len); // echo!
80.                     }
81.                 }
82.             }
83.         }
84.     }
85.     close(serv_sock);
86.     return 0;
87. }
88.
89. void error_handling(char *buf)
90. {
91.     fputs(buf, stderr);
92.     fputc('\n', stderr);
93.     exit(1);
94. }
```

代码说明

- 第40行：向要传到select函数第二个参数的fd_set变量reads注册服务器端套接字。这样，接收数据情况的监视对象就包含了服务器端套接字。客户端的连接请求同样通过传输数据完成。因此，服务器端套接字中有接收的数据，就意味着有新的连接请求。
- 第49行：在while无限循环中调用select函数。select函数的第三和第四个参数为空。只需根据监视目的传递必要的参数。
- 第54、56行：select函数返回大于等于1的值时执行的循环。第56行调用FD_ISSET函数，查找发生状态变化的（有接收数据的套接字的）文件描述符。
- 第58、63行：发生状态变化时，首先验证服务器端套接字中是否有变化。如果是服务器端套接字的变化，将受理连接请求。特别需要注意的是，第63行在fd_set变量reads中注册了与客户端连接的套接字文件描述符。
- 第68行：发生变化的套接字并非服务器端套接字时，即有要接受的数据时执行else语句。但此时需要确认接收的数据是字符串还是代表断开连接的EOF。
- 第73、74行：接收的数据为EOF时需要关闭套接字，并从reads中删除相应信息。
- 第79行：接收的数据为字符串时，执行回声服务。

❖ 运行结果：echo_selectserv.c

```
root@my_linux:/tcpip# gcc echo_selectserv.c -o selserv
root@my_linux:/tcpip# ./selserv 9190
connected client: 4
connected client: 5
closed client: 4
closed client: 5
```

❖ 运行结果：echo_client.c one

```
root@my_linux:/tcpip# gcc echo_client.c -o client
root@my_linux:/tcpip# ./client 127.0.0.1 9190
Connected.....
Input message(Q to quit): Hi~
Message from server: Hi~
Input message(Q to quit): Good bye
Message from server: Good bye
Input message(Q to quit): Q
```

❖ 运行结果：echo_client.c two

```
root@my_linux:/tcpip# ./client 127.0.0.1 9190
Connected.....
Input message(Q to quit): Nice to meet you~
Message from server: Nice to meet you~
Input message(Q to quit): Bye~
Message from server: Bye~
Input message(Q to quit): Q
```

为了验证运行结果，我使用了第4章介绍的echo_client.c，其实上述回声服务器端也可与其他回声客户端配合运行。

12.3　基于 Windows 的实现

在Window平台使用select函数时需要注意一些细节，本节主要补充这部分内容。

在 Windows 平台调用 select 函数

Windows同样提供select函数，而且所有参数与Linux的select函数完全相同。只不过Windows平台select函数的第一个参数是为了保持与（包括Linux的）UNIX系列操作系统的兼容性而添加的，并没有特殊意义。

```
#include <winsock2.h>

int select(
    int nfds, fd_set *readfds, fd_set *writefds, fd_set *excepfds, const struct
timeval * timeout);
```

➔ 成功时返回 0，失败时返回-1。

返回值、参数的顺序及含义与之前的Linux中的select函数完全相同，故省略。下面给出timeval结构体定义。

```
typedef struct timeval
{
    long tv_sec;      //seconds
    long tv_usec;     //microseconds
} TIMEVAL;
```

可以看到，基本结构与之前Linux中的定义相同，但Windows中使用的是typedef声明。接下来观察fd_set结构体。Windows中实现时需要注意的地方就在于此。可以看到，Windows的fd_set并非像Linux中那样采用了位数组。

```
typedef struct fd_set
{
    u_int    fd_count;
    SOCKET   fd_array[FD_SETSIZE];
} fd_set;
```

Windows的fd_set由成员fd_count和fd_array构成，fd_count用于套接字句柄数，fd_array用于保存套接字句柄。只要略加思考就能理解这样声明的原因。Linux的文件描述符从0开始递增，因此可以找出当前文件描述符数量和最后生成的文件描述符之间的关系。但Windows的套接字句柄并非从0开始，而且句柄的整数值之间并无规律可循，因此需要直接保存句柄的数组和记录句柄数的变量。幸好处理fd_set结构体的FD_XXX型的4个宏的名称、功能及使用方法与Linux完全相同（故省略），这也许是微软为了保证兼容性所做的考量。

基于 Windows 实现 I/O 复用服务器端

下面将示例echo_selectserv.c改为在Windows平台运行。如果各位掌握了之前的内容，那理解起来并不难，故省略源代码的讲解。

❖ echo_selectserv_win.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <winsock2.h>
5.
6.  #define BUF_SIZE 1024
7.  void ErrorHandling(char *message);
8.
9.  int main(int argc, char *argv[])
```

```
10. {
11.     WSADATA wsaData;
12.     SOCKET hServSock, hClntSock;
13.     SOCKADDR_IN servAdr, clntAdr;
14.     TIMEVAL timeout;
15.     fd_set reads, cpyReads;
16.
17.     int adrSz;
18.     int strLen, fdNum, i;
19.     char buf[BUF_SIZE];
20.
21.     if(argc!=2) {
22.         printf("Usage : %s <port>\n", argv[0]);
23.         exit(1);
24.     }
25.     if(WSAStartup(MAKEWORD(2, 2), &wsaData)!=0)
26.         ErrorHandling("WSAStartup() error!");
27.
28.     hServSock=socket(PF_INET, SOCK_STREAM, 0);
29.     memset(&servAdr, 0, sizeof(servAdr));
30.     servAdr.sin_family=AF_INET;
31.     servAdr.sin_addr.s_addr=htonl(INADDR_ANY);
32.     servAdr.sin_port=htons(atoi(argv[1]));
33.
34.     if(bind(hServSock, (SOCKADDR*) &servAdr, sizeof(servAdr))==SOCKET_ERROR)
35.         ErrorHandling("bind() error");
36.     if(listen(hServSock, 5)==SOCKET_ERROR)
37.         ErrorHandling("listen() error");
38.
39.     FD_ZERO(&reads);
40.     FD_SET(hServSock, &reads);
41.
42.     while(1)
43.     {
44.         cpyReads=reads;
45.         timeout.tv_sec=5;
46.         timeout.tv_usec=5000;
47.
48.         if((fdNum=select(0, &cpyReads, 0, 0, &timeout))==SOCKET_ERROR)
49.             break;
50.
51.         if(fdNum==0)
52.             continue;
53.
54.         for(i=0; i<reads.fd_count; i++)
55.         {
56.             if(FD_ISSET(reads.fd_array[i], &cpyReads))
57.             {
58.                 if(reads.fd_array[i]==hServSock)  // connection request!
59.                 {
60.                     adrSz=sizeof(clntAdr);
61.                     hClntSock=
62.                         accept(hServSock, (SOCKADDR*)&clntAdr, &adrSz);
63.                     FD_SET(hClntSock, &reads);
```

```
64.                 printf("connected client: %d \n", hClntSock);
65.             }
66.             else    // read message!
67.             {
68.                 strLen=recv(reads.fd_array[i], buf, BUF_SIZE-1, 0);
69.                 if(strLen==0)   // close request!
70.                 {
71.                     FD_CLR(reads.fd_array[i], &reads);
72.                     closesocket(cpyReads.fd_array[i]);
73.                     printf("closed client: %d \n", cpyReads.fd_array[i]);
74.                 }
75.                 else
76.                 {
77.                     send(reads.fd_array[i], buf, strLen, 0); // echo!
78.                 }
79.             }
80.         }
81.     }
82.     closesocket(hServSock);
83.     WSACleanup();
84.     return 0;
85. }
86.
87. void ErrorHandling(char *message)
88. {
89.     fputs(message, stderr);
90.     fputc('\n', stderr);
91.     exit(1);
92. }
93.
```

上述代码可以结合第4章的echo_client_win.c运行（当然也可以很好地与其他客户端结合运行）。最后，我想再次强调，本章的I/O复用技术在理论和实际中都非常重要，各位应熟练掌握。关于I/O复用的说明到此结束。

12.4 习题

(1) 请解释复用技术的通用含义，并说明何为I/O复用。

(2) 多进程并发服务器的缺点有哪些？如何在I/O复用服务器端中弥补？

(3) 复用服务器端需要select函数。下列关于select函数使用方法的描述错误的是？

a. 调用select函数前需要集中I/O监视对象的文件描述符。
b. 若已通过select函数注册为监视对象，则后续调用select函数时无需重复注册。
c. 复用服务器端同一时间只能服务于1个客户端，因此，需要服务的客户端接入服务器端后只能等待。

d. 与多进程服务器端不同，基于select的复用服务器端只需要1个进程。因此，可以减少因创建进程产生的服务器端的负担。

(4) select函数的观察对象中应包含服务器端套接字（监听套接字），那么应将其包含到哪一类监听对象集合？请说明原因。

(5) select函数中使用的fd_set结构体在Windows和Linux中具有不同声明。请说明区别，同时解释存在区别的必然性。

第13章

多种I/O函数 13

之前的示例中，基于 Linux 的使用 read&write 函数完成数据 I/O，基于 Windows 的则使用 send & recv 函数。原因已经在第 1 章进行了充分阐述。本章的 Linux 示例也将使用 send & recv 函数，并讲解其与 read & write 函数相比的优点所在。还将介绍几种其他的 I/O 函数。

13.1 send & recv 函数

虽然我们在之前的Windows示例中一直使用send & recv函数，但从未向最后一个参数传递过除0之外的其他值。也就是说，我们甚至连Windows平台下的send & recv函数都没有完全理解并应用。

Linux 中的 send & recv

虽然第1章介绍过send & recv函数，但那时是基于Windows平台介绍的。本节将介绍Linux平台下的send & recv函数。其实二者并无差别。

```
#include <sys/socket.h>

ssize_t send(int sockfd, const void * buf, size_t nbytes, int flags);
```

➔ 成功时返回发送的字节数，失败时返回-1。

- sockfd　表示与数据传输对象的连接的套接字文件描述符。
- buf　保存待传输数据的缓冲地址值。
- nbytes　待传输的字节数。
- flags　传输数据时指定的可选项信息。

与第1章的Windows中的send函数相比，上述函数在声明的结构体名称上有些区别。但参数的

顺序、含义、使用方法完全相同，因此实际区别不大。接下来介绍recv函数，该函数与Windows的recv函数相比也没有太大差别。

```
#include <sys/socket.h>

ssize_t recv(int sockfd, void * buf, size_t nbytes, int flags);
```

➔ 成功时返回接收的字节数（收到 EOF 时返回 0），失败时返回-1。

- sockfd　表示数据接收对象的连接的套接字文件描述符。
- buf　保存接收数据的缓冲地址值。
- nbytes　可接收的最大字节数。
- flags　接收数据时指定的可选项信息。

send函数和recv函数的最后一个参数是收发数据时的可选项。该可选项可利用位或（bit OR）运算（|运算符）同时传递多个信息。通过表13-1整理可选项的种类及含义。

表13-1　send&recv函数的可选项及含义

可选项（Option）	含　义	send	recv
MSG_OOB	用于传输带外数据（Out-of-band data）	•	•
MSG_PEEK	验证输入缓冲中是否存在接收的数据		•
MSG_DONTROUTE	数据传输过程中不参照路由（Routing）表，在本地（Local）网络中寻找目的地	•	
MSG_DONTWAIT	调用I/O函数时不阻塞，用于使用非阻塞（Non-blocking）I/O	•	•
MSG_WAITALL	防止函数返回，直到接收全部请求的字节数		•

另外，不同操作系统对上述可选项的支持也不同。因此，为了使用不同可选项，各位需要对实际开发中采用的操作系统有一定了解。下面选取表13-1中的一部分（主要是不受操作系统差异影响的）进行详细讲解。

MSG_OOB：发送紧急消息

MSG_OOB可选项用于发送“带外数据”紧急消息。假设医院里有很多病人在等待看病，此时若有急诊患者该怎么办？

“当然应该优先处理。”

如果急诊患者较多，需要得到等待看病的普通病人的谅解。正因如此，医院一般会设立单独的急诊室。需紧急处理时，应采用不同的处理方法和通道。MSG_OOB可选项就用于创建特殊发送方法和通道以发送紧急消息。下列示例将通过MSG_OOB可选项收发数据。使用MSG_OOB时

需要一些拓展知识，这部分内容通过源代码进行讲解。

❖ oob_send.c

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.  #include <stdlib.h>
4.  #include <string.h>
5.  #include <sys/socket.h>
6.  #include <arpa/inet.h>
7.
8.  #define BUF_SIZE 30
9.  void error_handling(char *message);
10.
11. int main(int argc, char *argv[])
12. {
13.     int sock;
14.     struct sockaddr_in recv_adr;
15.     if(argc!=3) {
16.         printf("Usage : %s <IP> <port>\n", argv[0]);
17.         exit(1);
18.     }
19.
20.     sock=socket(PF_INET, SOCK_STREAM, 0);
21.     memset(&recv_adr, 0, sizeof(recv_adr));
22.     recv_adr.sin_family=AF_INET;
23.     recv_adr.sin_addr.s_addr=inet_addr(argv[1]);
24.     recv_adr.sin_port=htons(atoi(argv[2]));
25.
26.     if(connect(sock, (struct sockaddr*)&recv_adr, sizeof(recv_adr))==-1)
27.         error_handling("connect() error!");
28.
29.     write(sock, "123", strlen("123"));
30.     send(sock, "4", strlen("4"), MSG_OOB);
31.     write(sock, "567", strlen("567"));
32.     send(sock, "890", strlen("890"), MSG_OOB);
33.     close(sock);
34.     return 0;
35. }
36.
37. void error_handling(char *message)
38. {
39.     fputs(message, stderr);
40.     fputc('\n', stderr);
41.     exit(1);
42. }
```

代码说明

- 第29~32行：传输数据。第30和第32行紧急传输数据。正常顺序应该是123、4、567、890，但紧急传输了4和890，由此可知接收顺序也将改变。

从上述示例可以看出，紧急消息的传输比即将介绍的接收过程要简单，只需在调用send函数时指定MSG_OOB可选项。接收紧急消息的过程要相对复杂一些。

❖ oob_recv.c

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.  #include <stdlib.h>
4.  #include <string.h>
5.  #include <signal.h>
6.  #include <sys/socket.h>
7.  #include <netinet/in.h>
8.  #include <fcntl.h>
9.
10. #define BUF_SIZE 30
11. void error_handling(char *message);
12. void urg_handler(int signo);
13.
14. int acpt_sock;
15. int recv_sock;
16.
17. int main(int argc, char *argv[])
18. {
19.     struct sockaddr_in recv_adr, serv_adr;
20.     int str_len, state;
21.     socklen_t serv_adr_sz;
22.     struct sigaction act;
23.     char buf[BUF_SIZE];
24.     if(argc!=2) {
25.         printf("Usage : %s <port>\n", argv[0]);
26.         exit(1);
27.     }
28.
29.     act.sa_handler=urg_handler;
30.     sigemptyset(&act.sa_mask);
31.     act.sa_flags=0;
32.
33.     acpt_sock=socket(PF_INET, SOCK_STREAM, 0);
34.     memset(&recv_adr, 0, sizeof(recv_adr));
35.     recv_adr.sin_family=AF_INET;
36.     recv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
37.     recv_adr.sin_port=htons(atoi(argv[1]));
38.
39.     if(bind(acpt_sock, (struct sockaddr*)&recv_adr, sizeof(recv_adr))==-1)
40.         error_handling("bind() error");
41.     listen(acpt_sock, 5);
42.
43.     serv_adr_sz=sizeof(serv_adr);
44.     recv_sock=accept(acpt_sock, (struct sockaddr*)&serv_adr, &serv_adr_sz);
45.
46.     fcntl(recv_sock, F_SETOWN, getpid());
47.     state=sigaction(SIGURG, &act, 0);
```

```
48.
49.     while((str_len=recv(recv_sock, buf, sizeof(buf), 0))!= 0)
50.     {
51.         if(str_len==-1)
52.             continue;
53.         buf[str_len]=0;
54.         puts(buf);
55.     }
56.     close(recv_sock);
57.     close(acpt_sock);
58.     return 0;
59. }
60.
61. void urg_handler(int signo)
62. {
63.     int str_len;
64.     char buf[BUF_SIZE];
65.     str_len=recv(recv_sock, buf, sizeof(buf)-1, MSG_OOB);
66.     buf[str_len]=0;
67.     printf("Urgent message: %s \n", buf);
68. }
69. void error_handling(char *message)
70. {
71.     fputs(message, stderr);
72.     fputc('\n', stderr);
73.     exit(1);
74. }
```

代码说明

- 第29、47行：该示例中需要重点观察SIGURG信号相关部分。收到MSG_OOB紧急消息时，操作系统将产生SIGURG信号，并调用注册的信号处理函数。另外需要注意的是，第61行的信号处理函数内部调用了接收紧急消息的recv函数。
- 第46行：调用fcntl函数，关于此函数将单独说明。

上述示例中插入了未曾讲解的函数调用语句，关于此函数只讲解必要部分，过多解释将脱离本章主题（第17章将再次说明）。

```
fcntl(recv_sock, F_SETOWN, getpid());
```

fcntl函数用于控制文件描述符，但上述调用语句的含义如下：

> “将文件描述符recv_sock指向的套接字拥有者（F_SETOWN）改为把getpid函数返回值用作ID的进程。”

各位或许感觉“套接字拥有者”的概念有些生疏。操作系统实际创建并管理套接字，所以从严格意义上说，“套接字拥有者”是操作系统。只是此处所谓的“拥有者”是指负责套接字所有事务的主体。上述描述可简要概括如下：

> “文件描述符recv_sock指向的套接字引发的SIGURG信号处理进程变为将getpid函数返回值用作ID的进程。”

当然，上述描述中的“处理SIGURG信号”指的是“调用SIGURG信号处理函数”。但之前讲过，多个进程可以共同拥有1个套接字的文件描述符。例如，通过调用fork函数创建子进程并同时复制文件描述符。此时如果发生SIGURG信号，应该调用哪个进程的信号处理函数呢？可以肯定的是，不会调用所有进程的信号处理函数（想想就知道这会引发更多问题）。因此，处理SIGURG信号时必须指定处理信号的进程，而getpid函数返回调用此函数的进程ID。上述调用语句指定当前进程为处理SIGURG信号的主体。该程序中只创建了1个进程，因此，理应由该进程处理SIGURG信号。接下来先给出运行结果，再讨论剩下的问题。

❖ 运行结果：oob_send.c

```
root@my_linux:/tcpip# gcc oob_send.c -o send
root@my_linux:/tcpip# ./send 127.0.0.1 9190
```

❖ 运行结果：oob_recv.c

```
root@my_linux:/tcpip# gcc oob_recv.c -o recv
root@my_linux:/tcpip# ./recv 9190
123
Urgent message: 4
567
Urgent message: 0
89
```

输出结果可能出乎大家预料，尤其是如下事实令人极为失望：

“通过MSG_OOB可选项传递数据时只返回1个字节？而且也不是很快啊！”

的确！令人遗憾的是，通过MSG_OOB可选项传递数据时不会加快数据传输速度，而且通过信号处理函数urg_handler读取数据时也只能读1个字节。剩余数据只能通过未设置MSG_OOB可选项的普通输入函数读取。这是因为TCP不存在真正意义上的“带外数据”。实际上，MSG_OOB中的OOB是指Out-of-band，而“带外数据”的含义是：

“通过完全不同的通信路径传输的数据。”

即真正意义上的Out-of-band需要通过单独的通信路径高速传输数据，但TCP不另外提供，只利用TCP的紧急模式（Urgent mode）进行传输。

紧急模式工作原理

先给出结论，再补充说明紧急模式。MSG_OOB可选项可以带来如下效果：

“嗨！这里有数据需要紧急处理，别磨蹭啦！”

MSG_OOB的真正的意义在于督促数据接收对象尽快处理数据。这是紧急模式的全部内容，而且TCP“保持传输顺序”的传输特性依然成立。

“那怎能称为紧急消息呢!”

这确实是紧急消息！因为发送消息者是在催促数据处理的情况下传输数据的。急诊患者的及时救治需要如下两个条件。

- 迅速入院。
- 医院急救。

无法快速把病人送到医院，并不意味着不需要医院进行急救。TCP的紧急消息无法保证及时入院，但可以要求急救。当然，急救措施应由程序员完成。之前的示例oob_recv.c的运行过程中也传递了紧急消息，这可以通过事件处理函数确认。这就是MSG_OOB模式数据传输的实际意义。下面给出设置MSG_OOB可选项状态下的数据传输过程，如图13-1所示。

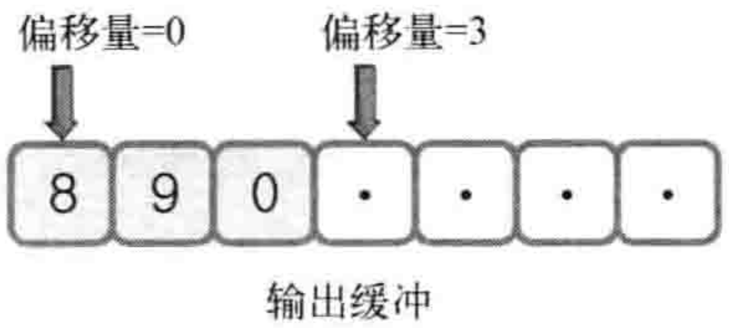

图13-1 紧急消息传输阶段的输出缓冲

图13-1给出的是示例oob_send.c的第32行中调用如下函数后的输出缓冲状态。此处假设已传输之前的数据。

```
send(sock, "890", strlen("890"), MSG_OOB);
```

如果将缓冲最左端的位置视作偏移量为0，字符0保存于偏移量为2的位置。另外，字符0右侧偏移量为3的位置存有紧急指针（Urgent Pointer）。紧急指针指向紧急消息的下一个位置（偏移量加1），同时向对方主机传递如下信息：

“紧急指针指向的偏移量为3之前的部分就是紧急消息!”

也就是说，实际只用1个字节表示紧急消息信息。这一点可以通过图13-1中用于传输数据的TCP数据包（段）的结构看得更清楚，如图13-2所示。

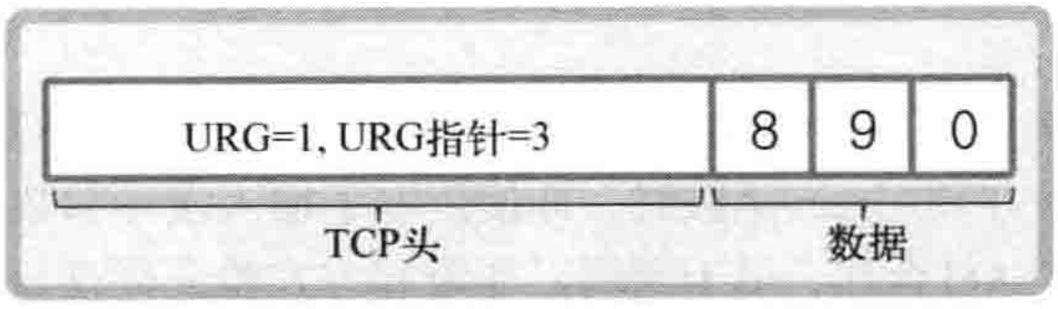

图13-2 设置URG的数据包

TCP数据包实际包含更多信息，但图13-2只标注了与我们的主题相关的内容。TCP头中含有如下两种信息。

- URG=1：载有紧急消息的数据包
- URG指针：紧急指针位于偏移量为3的位置

指定MSG_OOB选项的数据包本身就是紧急数据包，并通过紧急指针表示紧急消息所在位置。但通过图13-2无法得知以下事实：

“紧急消息是字符串890，还是90？如若不是，是否为单个字符0？”

但这并不重要。如前所述，除紧急指针的前面1个字节外，数据接收方将通过调用常用输入函数读取剩余部分。换言之，紧急消息的意义在于督促消息处理，而非紧急传输形式受限的消息。

知识补给站　计算机领域的偏移量（offset）

学习计算机相关领域时，会经常接触到术语“偏移量”，下面简单说明其含义。很多人通常认为偏移量就是从0开始每次增1的值。的确如此，但更准确地说，其含义如下：

“偏移量就是参照基准位置表示相对位置的量。”

为了理解这一点，请看图13-3，图中标有实际地址和偏移地址。

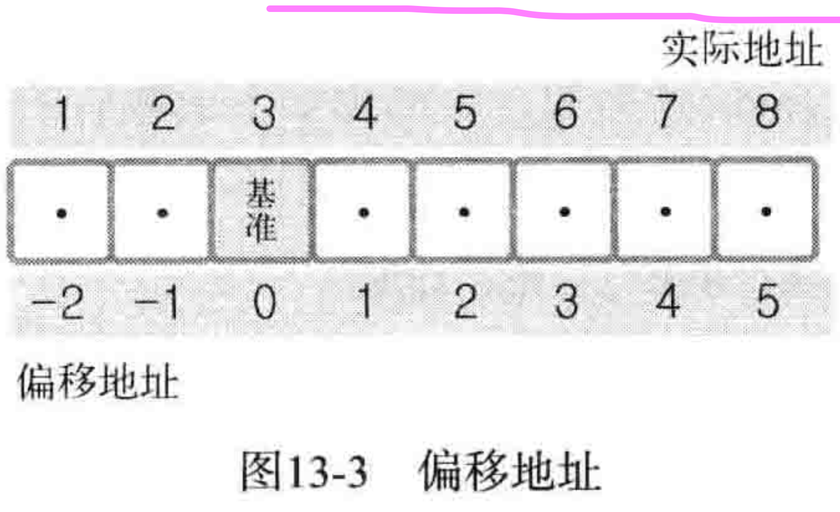

图13-3　偏移地址

图13-3给出了以实际地址3为基址计算偏移地址的过程。可以看到，偏移量表示距离基准点向哪个方向偏移多长距离。因此，与普通地址不同，偏移地址每次从0开始。

检查输入缓冲

同时设置MSG_PEEK选项和MSG_DONTWAIT选项，以验证输入缓冲中是否存在接收的数据。设置MSG_PEEK选项并调用recv函数时，即使读取了输入缓冲的数据也不会删除。因此，该选项通常与MSG_DONTWAIT合作，用于调用以非阻塞方式验证待读数据存在与否的函数。下面通过示例了解二者含义。

❖ peek_send.c

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.  #include <stdlib.h>
4.  #include <string.h>
5.  #include <sys/socket.h>
6.  #include <arpa/inet.h>
7.  void error_handling(char *message);
8.
9.  int main(int argc, char *argv[])
10. {
11.     int sock;
12.     struct sockaddr_in send_adr;
13.     if(argc!=3) {
14.         printf("Usage : %s <IP> <port>\n", argv[0]);
15.         exit(1);
16.     }
17.
18.     sock=socket(PF_INET, SOCK_STREAM, 0);
19.     memset(&send_adr, 0, sizeof(send_adr));
20.     send_adr.sin_family=AF_INET;
21.     send_adr.sin_addr.s_addr=inet_addr(argv[1]);
22.     send_adr.sin_port=htons(atoi(argv[2]));
23.
24.     if(connect(sock, (struct sockaddr*)&send_adr, sizeof(send_adr))==-1)
25.         error_handling("connect() error!");
26.
27.     write(sock, "123", strlen("123"));
28.     close(sock);
29.     return 0;
30. }
31.
32. void error_handling(char *message)
33. {
34.     fputs(message, stderr);
35.     fputc('\n', stderr);
36.     exit(1);
37. }
```

上述示例在第24行发起连接请求，第27行发送字符串123，此外无需赘述。下列示例给出了使用MSG_PEEK和MSG_DONTWAIT选项的结果。

❖ peek_recv.c

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.  #include <stdlib.h>
4.  #include <string.h>
5.  #include <sys/socket.h>
6.  #include <arpa/inet.h>
```

```
7.
8.  #define BUF_SIZE 30
9.  void error_handling(char *message);
10.
11. int main(int argc, char *argv[])
12. {
13.     int acpt_sock, recv_sock;
14.     struct sockaddr_in acpt_adr, recv_adr;
15.     int str_len, state;
16.     socklen_t recv_adr_sz;
17.     char buf[BUF_SIZE];
18.     if(argc!=2) {
19.         printf("Usage : %s <port>\n", argv[0]);
20.         exit(1);
21.     }
22.
23.     acpt_sock=socket(PF_INET, SOCK_STREAM, 0);
24.     memset(&acpt_adr, 0, sizeof(acpt_adr));
25.     acpt_adr.sin_family=AF_INET;
26.     acpt_adr.sin_addr.s_addr=htonl(INADDR_ANY);
27.     acpt_adr.sin_port=htons(atoi(argv[1]));
28.
29.     if(bind(acpt_sock, (struct sockaddr*)&acpt_adr, sizeof(acpt_adr))==-1)
30.         error_handling("bind() error");
31.     listen(acpt_sock, 5);
32.
33.     recv_adr_sz=sizeof(recv_adr);
34.     recv_sock=accept(acpt_sock, (struct sockaddr*)&recv_adr, &recv_adr_sz);
35.
36.     while(1)
37.     {
38.         str_len=recv(recv_sock, buf, sizeof(buf)-1, MSG_PEEK|MSG_DONTWAIT);
39.         if(str_len>0)
40.             break;
41.     }
42.
43.     buf[str_len]=0;
44.     printf("Buffering %d bytes: %s \n", str_len, buf);
45.
46.     str_len=recv(recv_sock, buf, sizeof(buf)-1, 0);
47.     buf[str_len]=0;
48.     printf("Read again: %s \n", buf);
49.     close(acpt_sock);
50.     close(recv_sock);
51.     return 0;
52. }
53.
54. void error_handling(char *message)
55. {
56.     fputs(message, stderr);
57.     fputc('\n', stderr);
58.     exit(1);
59. }
```

代码说明

- 第38行：调用recv函数的同时传递MSG_PEEK可选项，这是为了保证即使不存在待读取数据也不会进入阻塞状态。
- 第46行：再次调用recv函数。这次并未设置任何可选项，因此，本次读取的数据将从输入缓冲中删除。

❖ 运行结果：peek_recv.c

```
root@my_linux:/tcpip# peek_recv.c -o recv
root@my_linux:/tcpip# ./recv 9190
Buffering 3 bytes: 123
Read again: 123
```

❖ 运行结果：peek_send.c

```
root@my_linux:/tcpip# gcc peek_send.c -o send
root@my_linux:/tcpip# ./send 127.0.0.1 9190
```

通过运行结果可以验证，仅发送1次的数据被读取了2次，因为第一次调用recv函数时设置了MSG_PEEK可选项。以上就是MSG_PEEK可选项的功能。

13.2 readv & writev 函数

本节介绍的readv & writev函数有助于提高数据通信效率。先介绍这些函数的使用方法，再讨论其合理的应用场景。

使用 readv & writev 函数

readv & writev函数的功能可概括如下：

“对数据进行整合传输及发送的函数。”

也就是说，通过writev函数可以将分散保存在多个缓冲中的数据一并发送，通过readv函数可以由多个缓冲分别接收。因此，适当使用这2个函数可以减少I/O函数的调用次数。下面先介绍writev函数。

```
#include <sys/uio.h>

ssize_t writev(int filedes, const struct iovec * iov, int iovcnt);
```

➔ 成功时返回发送的字节数，失败时返回-1。

- filedes　表示数据传输对象的套接字文件描述符。但该函数并不只限于套接字，因此，可以像read函数一样向其传递文件或标准输出描述符。
- iov　iovec结构体数组的地址值，结构体iovec中包含待发送数据的位置和大小信息。
- iovcnt　向第二个参数传递的数组长度。

上述函数的第二个参数中出现的数组iovec结构体的声明如下。

```
struct iovec
{
    void * iov_base;  // 缓冲地址
    size_t iov_len;   // 缓冲大小
}
```

可以看到，结构体iovec由保存待发送数据的缓冲（char型数组）地址值和实际发送的数据长度信息构成。给出上述函数的调用示例前，先通过图13-4了解该函数的使用方法。

```
writev( 1 , ptr ,  2 );
```

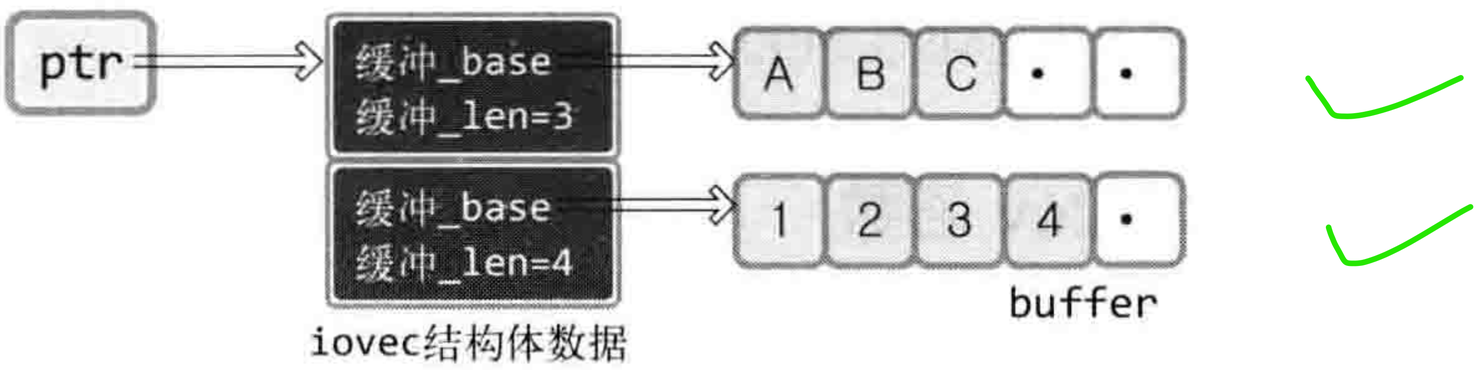

图13-4　write & iovec

图13-4中writev的第一个参数1是文件描述符，因此向控制台输出数据，ptr是存有待发送数据信息的iovec数组指针。第三个参数为2，因此，从ptr指向的地址开始，共浏览2个iovec结构体变量，发送这些指针指向的缓冲数据。接下来仔细观察图中的iovec结构体数组。ptr[0]（数组第一个元素）的iov_base指向以A开头的字符串，同时iov_len为3，故发送ABC。而ptr[1]（数组的第二个元素）的iov_base指向数字1，同时iov_len为4，故发送1234。

相信各位已掌握writev函数的使用方法和特性，接下来给出示例。

❖ writev.c

```
1.  #include <stdio.h>
2.  #include <sys/uio.h>
3.
4.  int main(int argc, char *argv[])
5.  {
6.      struct iovec vec[2];
7.      char buf1[]="ABCDEFG";
8.      char buf2[]="1234567";
9.      int str_len;
```

```
10.
11.     vec[0].iov_base=buf1;
12.     vec[0].iov_len=3;
13.     vec[1].iov_base=buf2;
14.     vec[1].iov_len=4;
15.
16.     str_len=writev(1, vec, 2);
17.     puts("");
18.     printf("Write bytes: %d \n", str_len);
19.     return 0;
20. }
```

代码说明

- 第11、12行：写入第一个传输数据的保存位置和大小。
- 第13、14行：写入第二个传输数据的保存位置和大小。
- 第16行：writev函数的第一个参数为1，故向控制台输出数据。

❖ 运行结果：writev.c

```
root@my_linux:/tcpip# gcc writev.c -o wv
root@my_linux:/tcpip# ./wv
ABC1234
Write bytes: 7
```

下面介绍readv函数，它与writev函数正好相反。

```
#include <sys/uio.h>

ssize_t readv(int filedes, const struct iovec * iov, int iovcnt);
```

➔ 成功时返回接收的字节数，失败时返回-1。

- filedes　传递接收数据的文件（或套接字）描述符。
- iov　包含数据保存位置和大小信息的iovec结构体数组的地址值。
- iovcnt　第二个参数中数组的长度。

我们已经学习了writev函数，因此直接通过示例给出readv函数的使用方法。

❖ readv.c

```
1.  #include <stdio.h>
2.  #include <sys/uio.h>
3.  #define BUF_SIZE 100
4.
5.  int main(int argc, char *argv[])
6.  {
7.      struct iovec vec[2];
8.      char buf1[BUF_SIZE]={0,};
```

```
9.      char buf2[BUF_SIZE]={0,};
10.     int str_len;
11.
12.     vec[0].iov_base=buf1;
13.     vec[0].iov_len=5;
14.     vec[1].iov_base=buf2;
15.     vec[1].iov_len=BUF_SIZE;
16.
17.     str_len=readv(0, vec, 2);
18.     printf("Read bytes: %d \n", str_len);
19.     printf("First message: %s \n", buf1);
20.     printf("Second message: %s \n", buf2);
21.     return 0;
22. }
```

代码说明

- 第12、13行：设置第一个数据的保存位置和大小。接收数据的大小已指定为5，因此，无论buf1的大小是多少，最多仅能保存5个字节。
- 第14、15行：vec[0]中注册的缓冲中保存5个字节，剩余数据将保存到vec[1]中注册的缓冲。结构体iovec的成员iov_len中应写入接收的最大字节数。
- 第17行：readv函数的第一个参数为0，因此从标准输入接收数据。

❖ 运行结果：writev.c

```
root@my_linux:/tcpip# gcc readv.c -o rv
root@my_linux:/tcpip# ./rv
I like TCP/IP socket programming~
Read bytes: 34
First message: I lik
Second message: e TCP/IP socket programming~
```

由运行结果可知，通过第7行声明的vec数组保存了数据。

合理使用 readv & writev 函数

哪种情况适合使用readv和writev函数？实际上，能使用该函数的所有情况都适用。例如，需要传输的数据分别位于不同缓冲（数组）时，需要多次调用write函数。此时可以通过1次writev函数调用替代操作，当然会提高效率。同样，需要将输入缓冲中的数据读入不同位置时，可以不必多次调用read函数，而是利用1次readv函数就能大大提高效率。

即使仅从C语言角度看，减少函数调用次数也能相应提高性能。但其更大的意义在于减少数据包个数。假设为了提高效率而在服务器端明确阻止了Nagle算法。其实writev函数在不采用Nagle算法时更有价值，如图13-5所示。

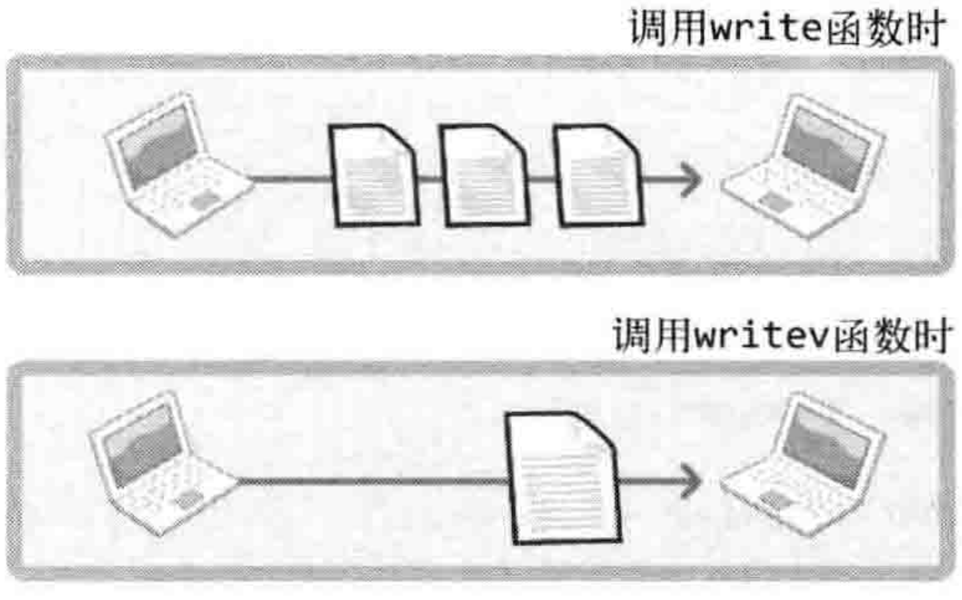

图13-5 Nagle算法关闭状态下的数据传输

上述示例中待发送的数据分别存在3个不同的地方，此时如果使用write函数则需要3次函数调用。但若为提高速度而关闭了Nagle算法，则极有可能通过3个数据包传递数据。反之，若使用writev函数将所有数据一次性写入输出缓冲，则很有可能仅通过1个数据包传输数据。所以writev函数和readv函数非常有用。

再考虑一种情况：将不同位置的数据按照发送顺序移动（复制）到1个大数组，并通过1次write函数调用进行传输。这种方式是否与调用writev函数的效果相同？当然！但使用writev函数更为便利。因此，如果遇到writev函数和readv函数的适用情况，希望各位不要错过机会。

13.3 基于 Windows 的实现

我们在之前的Windows示例中通过send函数和recv函数完成了数据交换，因此只需补充设置可选项相关示例。下面将Linux示例oob_send.c和oob_recv.c移植到Windows平台，此处需要考虑一点：

“Windows中并不存在Linux那样的信号处理机制。”

示例oob_send.c和oob_recv.c的核心内容就是MSG_OOB可选项的设置。但在Windows中无法完成针对该可选项的事件处理，需要考虑使用其他方法。我们通过select函数解决这一问题。之前讲过的select函数的3种监视对象如下所示。

- 是否存在套接字接收数据？
- 无需阻塞传输数据的套接字有哪些？
- 哪些套接字发生了异常？

其中，第12章也未对“发生异常的套接字”另行讲解。“异常”是不同寻常的程序执行流，因此，收到Out-of-band数据也属于异常。也就是说，利用select函数的这一特性可以在Windows平台接收Out-of-band数据，参考如下示例。

❖ oob_send_win.c

```
#include <stdio.h>
#include <stdlib.h>
#include <winsock2.h>

#define BUF_SIZE 30
void ErrorHandling(char *message);

int main(int argc, char *argv[])
{
    WSADATA wsaData;
    SOCKET hSocket;
    SOCKADDR_IN sendAdr;
    if(argc!=3) {
        printf("Usage : %s <IP> <port>\n", argv[0]);
        exit(1);
    }

    if(WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
        ErrorHandling("WSAStartup() error!");

    hSocket=socket(PF_INET, SOCK_STREAM, 0);
    memset(&sendAdr, 0, sizeof(sendAdr));
    sendAdr.sin_family=AF_INET;
    sendAdr.sin_addr.s_addr=inet_addr(argv[1]);
    sendAdr.sin_port=htons(atoi(argv[2]));

    if(connect(hSocket, (SOCKADDR*)&sendAdr, sizeof(sendAdr))==SOCKET_ERROR)
        ErrorHandling("connect() error!");

    send(hSocket, "123", 3, 0);
    send(hSocket, "4", 1, MSG_OOB);
    send(hSocket, "567", 3, 0);
    send(hSocket, "890", 3, MSG_OOB);

    closesocket(hSocket);
    WSACleanup();
    return 0;
}

void ErrorHandling(char *message)
{
    fputs(message, stderr);
    fputc('\n', stderr);
    exit(1);
}
```

上述示例是发送紧急消息的代码，但该示例只是oob_send.c的Windows移植版，故省略。下面给出的接收紧急消息的代码则与oob_recv.c不同，采用了select函数，有必要仔细阅读。

❖ oob_recv_win.c

```
#include <stdio.h>
#include <stdlib.h>
#include <winsock2.h>

#define BUF_SIZE 30
void ErrorHandling(char *message);

int main(int argc, char *argv[])
{
    WSADATA wsaData;
    SOCKET hAcptSock, hRecvSock;

    SOCKADDR_IN recvAdr;
    SOCKADDR_IN sendAdr;
    int sendAdrSize, strLen;
    char buf[BUF_SIZE];
    int result;

    fd_set read, except, readCopy, exceptCopy;
    struct timeval timeout;

    if(argc!=2) {
        printf("Usage : %s <port>\n", argv[0]);
        exit(1);
    }

    if(WSAStartup(MAKEWORD(2, 2), &wsaData)!=0)
        ErrorHandling("WSAStartup() error!");

    hAcptSock=socket(PF_INET, SOCK_STREAM, 0);
    memset(&recvAdr, 0, sizeof(recvAdr));
    recvAdr.sin_family=AF_INET;
    recvAdr.sin_addr.s_addr=htonl(INADDR_ANY);
    recvAdr.sin_port=htons(atoi(argv[1]));

    if(bind(hAcptSock, (SOCKADDR*)&recvAdr, sizeof(recvAdr))==SOCKET_ERROR)
        ErrorHandling("bind() error");
    if(listen(hAcptSock, 5)==SOCKET_ERROR)
        ErrorHandling("listen() error");

    sendAdrSize=sizeof(sendAdr);
    hRecvSock=accept(hAcptSock, (SOCKADDR*)&sendAdr, &sendAdrSize);
    FD_ZERO(&read);
    FD_ZERO(&except);
    FD_SET(hRecvSock, &read);
    FD_SET(hRecvSock, &except);

    while(1)
    {
        readCopy=read;
        exceptCopy=except;
```

```
52.         timeout.tv_sec=5;
53.         timeout.tv_usec=0;
54.
55.         result=select(0, &readCopy, 0, &exceptCopy, &timeout);
56.
57.         if(result>0)
58.         {
59.             if(FD_ISSET(hRecvSock, &exceptCopy))
60.             {
61.                 strLen=recv(hRecvSock, buf, BUF_SIZE-1, MSG_OOB);
62.                 buf[strLen]=0;
63.                 printf("Urgent message: %s \n", buf);
64.             }
65.
66.             if(FD_ISSET(hRecvSock, &readCopy))
67.             {
68.                 strLen=recv(hRecvSock, buf, BUF_SIZE-1, 0);
69.                 if(strLen==0)
70.                 {
71.                     break;
72.                     closesocket(hRecvSock);
73.                 }
74.                 else
75.                 {
76.                     buf[strLen]=0;
77.                     puts(buf);
78.                 }
79.             }
80.         }
81.     }
82.
83.     closesocket(hAcptSock);
84.     WSACleanup();
85.     return 0;
86. }
87.
88. void ErrorHandling(char *message)
89. {
90.     fputs(message, stderr);
91.     fputc('\n', stderr);
92.     exit(1);
93. }
```

虽然代码有些偏长，但除了调用select函数接收Out-of-band数据的部分外，没有特别之处。而且第12章已经详细讲解了select函数，相信各位能够自行分析上述代码。

另外，Windows中并没有函数与writev & readv函数直接对应，但可以通过“重叠I/O”（Overlapped I/O）得到相同效果。关于重叠I/O需要处理不少细节问题，后续章节将进行讲解。各位只需记住Linux的writev & readv函数的功能可以通过Windows的“重叠I/O”实现。

13.4 习题

(1) 下列关于MSG_OOB可选项的说法错误的是?

a. MSG_OOB指传输Out-of-band数据，是通过其他路径高速传输数据。

b. MSG_OOB指通过其他路径高速传输数据，因此，TCP中设置该选项的数据先到达对方主机。

c. 设置MSG_OOB使数据先到达对方主机后，以普通数据的形式和顺序读取。也就是说，只是提高了传输速度，接收方无法识别这一点。

d. MSG_OOB无法脱离TCP的默认数据传输方式。即使设置了MSG_OOB，也会保持原有传输顺序。该选项只用于要求接收方紧急处理。

(2) 利用readv & writev函数收发数据有何优点？分别从函数调用次数和I/O缓冲的角度给出说明。

(3) 通过recv函数验证输入缓冲是否存在数据时（确认后立即返回时），如何设置recv函数最后一个参数中的可选项？分别说明各可选项的含义。

(4) 可在Linux平台通过注册事件处理函数接收MSG_OOB数据。那Windows中如何接收？请说明接收方法。

第 14 章

多播与广播 14

假设各位经营网络电台，需要向用户发送多媒体信息。如果有 1000 名用户，则需要向 1000 名用户发送数据；如果有 10000 名用户，则需要向 10000 名用户发送数据。此时，如果基于 TCP 提供服务，则需要维护 1000 个或 10000 个套接字连接，即使用 UDP 套接字提供服务，也需要 1000 次或 10000 次数据传输。像这样，向大量客户端发送相同数据时，也会对服务器端和网络流量产生负面影响。可以使用多播技术解决该问题。

14.1 多播

多播（Multicast）方式的数据传输是基于UDP完成的。因此，与UDP服务器端/客户端的实现方式非常接近。区别在于，UDP数据传输以单一目标进行，而多播数据同时传递到加入（注册）特定组的大量主机。换言之，采用多播方式时，可以同时向多个主机传递数据。

多播的数据传输方式及流量方面的优点

多播的数据传输特点可整理如下。

- 多播服务器端针对特定多播组，只发送1次数据。
- 即使只发送1次数据，但该组内的所有客户端都会接收数据。
- 多播组数可在IP地址范围内任意增加。
- 加入特定组即可接收发往该多播组的数据。

多播组是D类IP地址（224.0.0.0~239.255.255.255），“加入多播组”可以理解为通过程序完成如下声明：

“在D类IP地址中，我希望接收发往目标239.234.218.234的多播数据。”

多播是基于UDP完成的，也就是说，多播数据包的格式与UDP数据包相同。只是与一般的

UDP数据包不同，向网络传递1个多播数据包时，路由器将复制该数据包并传递到多个主机。像这样，多播需要借助路由器完成，如图14-1所示。

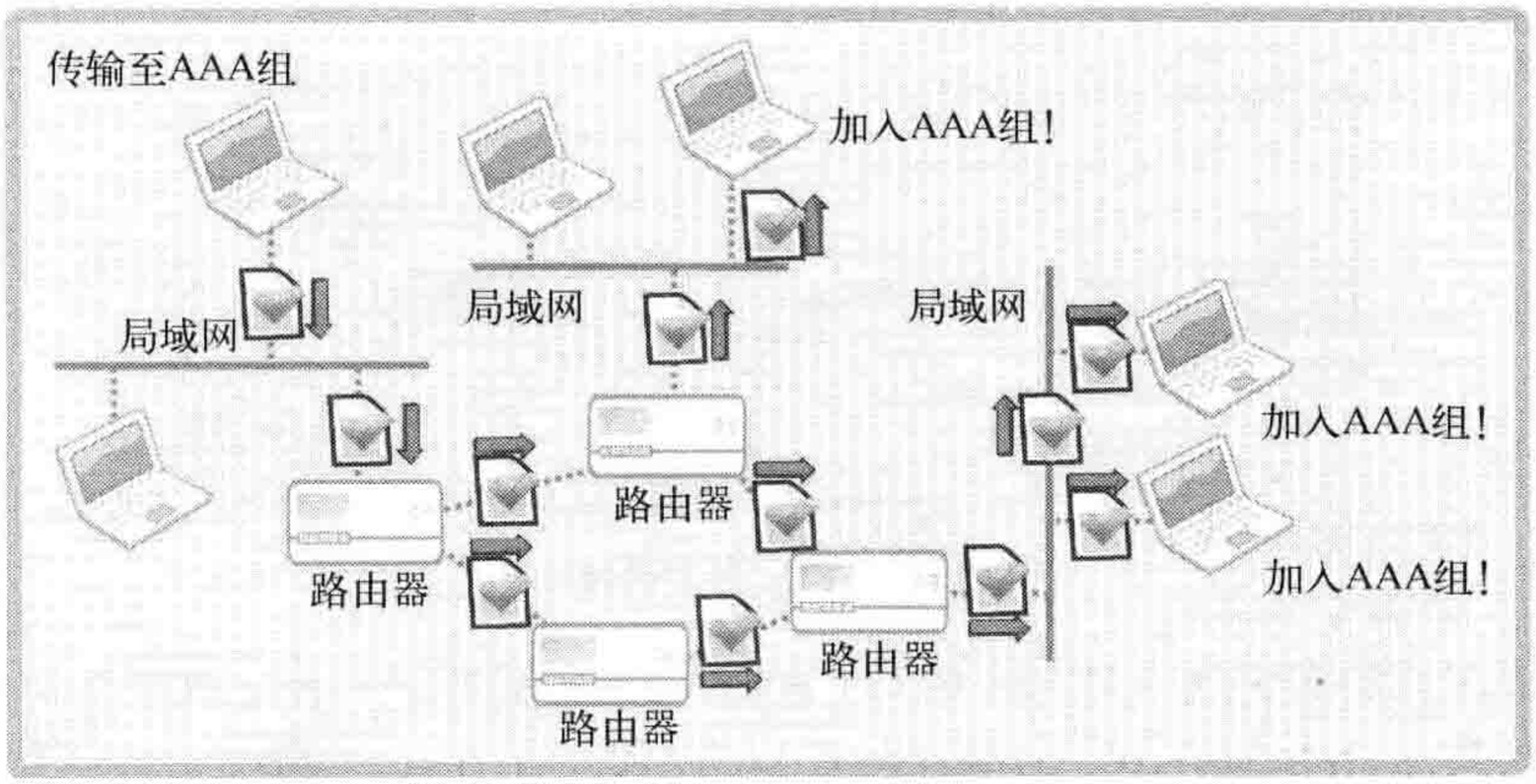

图14-1 多播路由

图14-1表示传输至AAA组的多播数据包借助路由器传递到加入AAA组的所有主机的过程。

“但这种方式不利于网络流量啊!”

我在本章开始部分讲过：

“像这样，向大量客户端发送相同数据时，也会对服务器端和网络流量产生负面影响。可以使用多播技术解决该问题。”

只看图14-1，各位也许会认为这不利于网络流量，因为路由器频繁复制同一数据包。但请从另一方面考虑。

“不会向同一区域发送多个相同数据包!”

若通过TCP或UDP向1000个主机发送文件，则共需要传递1000次。即便将10台主机合为1个网络，使99%的传输路径相同的情况下也是如此。但此时若使用多播方式传输文件，则只需发送1次。这时由1000台主机构成的网络中的路由器负责复制文件并传递到主机。就因为这种特性，多播主要用于“多媒体数据的实时传输”。

另外，虽然理论上可以完成多播通信，但不少路由器并不支持多播，或即便支持也因网络拥堵问题故意阻断多播。因此，为了在不支持多播的路由器中完成多播通信，也会使用隧道（Tunneling）技术（这并非多播程序开发人员需要考虑的问题）。我们只讨论支持多播服务的环境下的编程方法。

路由（Routing）和 TTL（Time to Live，生存时间），以及加入组的方法

接下来讨论多播相关编程方法。为了传递多播数据包，必需设置TTL。TTL是Time to Live的

简写，是决定“数据包传递距离”的主要因素。TTL用整数表示，并且每经过1个路由器就减1。TTL变为0时，该数据包无法再被传递，只能销毁。因此，TTL的值设置过大将影响网络流量。当然，设置过小也会无法传递到目标，需要引起注意。

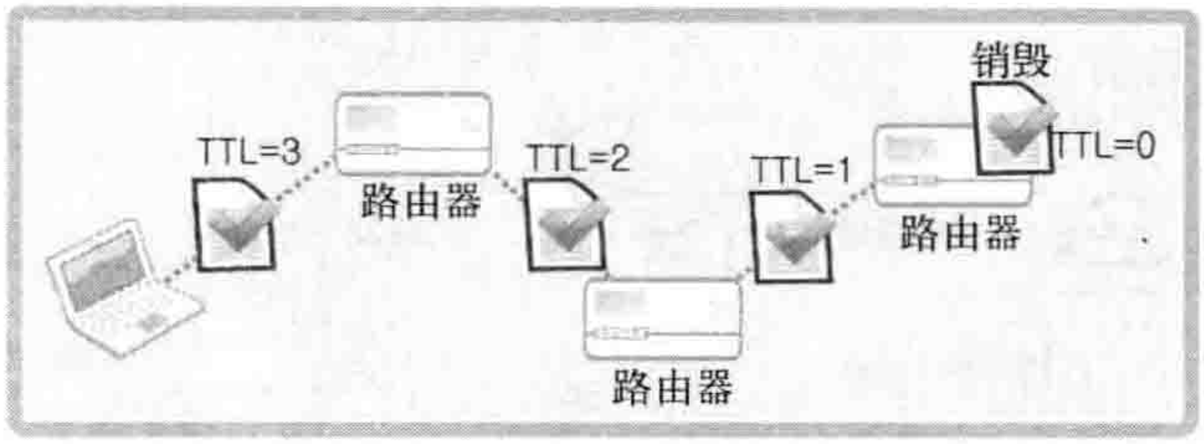

图14-2　TTL和多播路由

接下来给出TTL设置方法。程序中的TTL设置是通过第9章的套接字可选项完成的。与设置TTL相关的协议层为IPPROTO_IP，选项名为IP_MULTICAST_TTL。因此，可以用如下代码把TTL设置为64。

```
int send_sock;
int time_live=64;
. . . .
send_sock=socket(PF_INET, SOCK_DGRAM, 0);
setsockopt(send_sock, IPPROTO_IP, IP_MULTICAST_TTL, (void*) &time_live,
sizeof(time_live));
. . . .
```

另外，加入多播组也通过设置套接字选项完成。加入多播组相关的协议层为IPPROTO_IP，选项名为IP_ADD_MEMBERSHIP。可通过如下代码加入多播组。

```
int recv_sock;
struct ip_mreq join_adr;
. . . .
recv_sock=socket(PF_INET, SOCK_DGRAM, 0);
. . . .
join_adr.imr_multiaddr.s_addr="多播组地址信息";
join_adr.imr_interface.s_addr="加入多播组的主机地址信息";
setsockopt(recv_sock, IPPROTO_IP, IP_ADD_MEMBERSHIP, (void*) & join_adr,
sizeof(join_adr));
. . . .
```

上述代码只给出了与setsockopt函数相关的部分，详细内容将在稍后示例中给出。此处只讲解ip_mreq结构体，该结构体定义如下。

```
struct ip_mreq
{
    struct in_addr imr_multiaddr;
    struct in_addr imr_interface;
}
```

第3章讲过in_addr结构体，因此只介绍结构体成员。首先，第一个成员imr_multiaddr中写入加入的组IP地址。第二个成员imr_interface是加入该组的套接字所属主机的IP地址，也可使用INADDR_ANY。

实现多播 Sender 和 Receiver

多播中用“发送者”（以下称为Sender）和“接受者”（以下称为Receiver）替代服务器端和客户端。顾名思义，此处的Sender是多播数据的发送主体，Receiver是需要多播组加入过程的数据接收主体。下面讨论即将给出的示例，该示例的运行场景如下。

- Sender：向AAA组广播（Broadcasting）文件中保存的新闻信息。
- Receiver：接收传递到AAA组的新闻信息。

接下来给出Sender代码。Sender比Receiver简单，因为Receiver需要经过加入组的过程，而Sender只需创建UDP套接字，并向多播地址发送数据。

❖ news_sender.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <unistd.h>
5.  #include <arpa/inet.h>
6.  #include <sys/socket.h>
7.
8.  #define TTL 64
9.  #define BUF_SIZE 30
10. void error_handling(char *message);
11.
12. int main(int argc, char *argv[])
13. {
14.     int send_sock;
15.     struct sockaddr_in mul_adr;
16.     int time_live=TTL;
17.     FILE *fp;
18.     char buf[BUF_SIZE];
19.     if(argc!=3) {
20.         printf("Usage : %s <GroupIP> <PORT>\n", argv[0]);
21.         exit(1);
22.     }
```

```
23.
24.     send_sock=socket(PF_INET, SOCK_DGRAM, 0);
25.     memset(&mul_adr, 0, sizeof(mul_adr));
26.     mul_adr.sin_family=AF_INET;
27.     mul_adr.sin_addr.s_addr=inet_addr(argv[1]);   // Multicast IP
28.     mul_adr.sin_port=htons(atoi(argv[2]));    // Multicast Port
29.
30.     setsockopt(send_sock, IPPROTO_IP,
31.         IP_MULTICAST_TTL, (void*)&time_live, sizeof(time_live));
32.     if((fp=fopen("news.txt", "r"))==NULL)
33.         error_handling("fopen() error");
34.
35.     while(!feof(fp))    /* Broadcasting */
36.     {
37.         fgets(buf, BUF_SIZE, fp);
38.         sendto(send_sock, buf, strlen(buf),
39.             0, (struct sockaddr*)&mul_adr, sizeof(mul_adr));
40.         sleep(2);
41.     }
42.     fclose(fp);
43.     close(send_sock);
44.     return 0;
45. }
46.
47. void error_handling(char *message)
48. {
49.     fputs(message, stderr);
50.     fputc('\n', stderr);
51.     exit(1);
52. }
```

代码说明

- 第24行：多播数据通信是通过UDP完成的，因此创建UDP套接字。
- 第26~28行：设置传输数据的目标地址信息。重要的是，必须将IP地址设置为多播地址。
- 第30行：指定套接字TTL信息，这是Sender中的必要过程。
- 第35~41行：实际传输数据的区域。基于UDP套接字传输数据，因此需要利用sendto函数。另外，第40行的sleep函数调用主要是为了给传输数据提供一定的时间间隔而添加的，没有其他特殊意义。

从上述代码中可以看到，Sender与普通的UDP套接字程序相比差别不大。但多播Receiver则有些不同。为了接收传向任意多播地址的数据，需要经过加入多播组的过程。除此之外，Receiver同样与UDP套接字程序差不多。接下来给出与上述示例结合使用的Receiver程序。

❖ news_receiver.c

```
1.  #include <"与news_sender.c的头声明一致，故省略。">
2.  #define BUF_SIZE 30
3.  void error_handling(char *message);
4.
5.  int main(int argc, char *argv[])
```

```
6.  {
7.      int recv_sock;
8.      int str_len;
9.      char buf[BUF_SIZE];
10.     struct sockaddr_in adr;
11.     struct ip_mreq join_adr;
12.     if(argc!=3) {
13.         printf("Usage : %s <GroupIP> <PORT>\n", argv[0]);
14.         exit(1);
15.     }
16.
17.     recv_sock=socket(PF_INET, SOCK_DGRAM, 0);
18.     memset(&adr, 0, sizeof(adr));
19.     adr.sin_family=AF_INET;
20.     adr.sin_addr.s_addr=htonl(INADDR_ANY);
21.     adr.sin_port=htons(atoi(argv[2]));
22.
23.     if(bind(recv_sock, (struct sockaddr*) &adr, sizeof(adr))==-1)
24.         error_handling("bind() error");
25.
26.     join_adr.imr_multiaddr.s_addr=inet_addr(argv[1]);
27.     join_adr.imr_interface.s_addr=htonl(INADDR_ANY);
28.
29.     setsockopt(recv_sock, IPPROTO_IP,
30.         IP_ADD_MEMBERSHIP, (void*)&join_adr, sizeof(join_adr));
31.
32.     while(1)
33.     {
34.         str_len=recvfrom(recv_sock, buf, BUF_SIZE-1, 0, NULL, 0);
35.         if(str_len<0)
36.             break;
37.         buf[str_len]=0;
38.         fputs(buf, stdout);
39.     }
40.     close(recv_sock);
41.     return 0;
42. }
43.
44. void error_handling(char *message)
45. {
46.     //与news_sender.c的error_handling函数一致。
47. }
```

代码说明

- 第26、27行：初始化结构体ip_mreg变量。第26行初始化多播组地址，第27行初始化待加入主机的IP地址。
- 第29行：利用套接字选项IP_ADD_MEMBERSHIP加入多播组。至此完成了接收第26行指定的多播组数据的所有准备。
- 第34行：通过调用recvfrom函数接收多播数据。如果不需要知道传输数据的主机地址信息，可以向recvfrom函数的第五个和第六个参数分别传递NULL和0。

❖ 运行结果：news_sender.c

```
root@my_linux:/tcpip# gcc news_sender.c -o sender
root@my_linux:/tcpip# ./sender
Usage : ./sender <GroupIP> <PORT>
root@my_linux:/tcpip# ./sender 224.1.1.2 9190
```

❖ 运行结果：news_receiver.c

```
root@my_linux:/tcpip# gcc news_receiver.c -o receiver
root@my_linux:/tcpip# ./receiver
Usage : ./receiver <GroupIP> <PORT>
root@my_linux:/tcpip# ./receiver 224.1.1.2 9190
The government, however, apparently overlooked a law that requires a CPA to
receive at least two years of practical training at a public accounting firm.
After realizing that, the FSS then suggested that firms listed on the Korea Stock
Exchange accommodate the newly minted CPAs, but accountants rejected the idea.
"The main purpose of selecting more CPAs is to ensure transparent accounting,"
said Yoon Jong-wook, head of the CPAs' own committee for improving training
conditions.
```

各位是否运行过该示例？Sender和Receiver之间的端口号应保持一致，虽然未讲，但理所应当。运行顺序并不重要，因为不像TCP套接字在连接状态下收发数据。只是因为多播属于广播的范畴，如果延迟运行Receiver，则无法接收之前传输的多播数据。

知识补给站　MBone（Multicast Backbone，多播主干网）

多播是基于MBone这个虚拟网络工作的。各位或许对虚拟网络感到陌生，但可将其理解为“通过网络中的特殊协议工作的软件概念上的网络”。也就是说，MBone并非可以触及的物理网络。它是以物理网络为基础，通过软件方法实现的多播通信必备虚拟网络。用于多播通信的虚拟网络的研究目前仍在进行，这与多播应用程序的编写属于不同领域。

14.2　广播

本节介绍的广播（Broadcast）在“一次性向多个主机发送数据”这一点上与多播类似，但传输数据的范围有区别。多播即使在跨越不同网络的情况下，只要加入多播组就能接收数据。相反，广播只能向同一网络中的主机传输数据。

广播的理解及实现方法

广播是向同一网络中的所有主机传输数据的方法。与多播相同，广播也是基于UDP完成的。根据传输数据时使用的IP地址的形式，广播分为如下2种。

- 直接广播（Directed Broadcast）
- 本地广播（Local Broadcast）

二者在代码实现上的差别主要在于IP地址。直接广播的IP地址中除了网络地址外，其余主机地址全部设置为1。例如，希望向网络地址192.12.34中的所有主机传输数据时，可以向192.12.34.255传输。换言之，可以采用直接广播的方式向特定区域内所有主机传输数据。

反之，本地广播中使用的IP地址限定为255.255.255.255。例如，192.32.24网络中的主机向255.255.255.255传输数据时，数据将传递到192.32.24网络中的所有主机。

那么，应当如何实现Sender和Receiver呢？实际上，如果不仔细观察广播示例中通信时使用的IP地址，则很难与UDP示例进行区分。也就是说，数据通信中使用的IP地址是与UDP示例的唯一区别。默认生成的套接字会阻止广播，因此，只需通过如下代码更改默认设置。

```
int send_sock;
int bcast = 1;     // 对变量进行初始化以将 SO_BROADCAST 选项信息改为 1。
. . . .
send_sock = socket(PF_INET, SOCK_DGRAM, 0);
. . . .
setsockopt(send_sock, SOL_SOCKET, SO_BROADCAST, (void*) & bcast, sizeof(bcast));
. . . .
```

调用setsockopt函数，将SO_BROADCAST选项设置为bcast变量中的值1。这意味着可以进行数据广播。当然，上述套接字选项只需在Sender中更改，Receiver的实现不需要该过程。

实现广播数据的 Sender 和 Receiver

下面实现基于广播的Sender和Receiver。为了与多播示例进行对比，将之前的news_sender.c和news_receiver.c改为广播的示例。

❖ news_sender_brd.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <unistd.h>
5.  #include <arpa/inet.h>
6.  #include <sys/socket.h>
```

```
7.
8.  #define BUF_SIZE 30
9.  void error_handling(char *message);
10.
11. int main(int argc, char *argv[])
12. {
13.     int send_sock;
14.     struct sockaddr_in broad_adr;
15.     FILE *fp;
16.     char buf[BUF_SIZE];
17.     int so_brd=1;
18.     if(argc!=3) {
19.         printf("Usage : %s <Boradcast IP> <PORT>\n", argv[0]);
20.         exit(1);
21.     }
22.
23.     send_sock=socket(PF_INET, SOCK_DGRAM, 0);
24.     memset(&broad_adr, 0, sizeof(broad_adr));
25.     broad_adr.sin_family=AF_INET;
26.     broad_adr.sin_addr.s_addr=inet_addr(argv[1]);
27.     broad_adr.sin_port=htons(atoi(argv[2]));
28.
29.     setsockopt(send_sock, SOL_SOCKET,
30.         SO_BROADCAST, (void*)&so_brd, sizeof(so_brd));
31.     if((fp=fopen("news.txt", "r"))==NULL)
32.         error_handling("fopen() error");
33.
34.     while(!feof(fp))
35.     {
36.         fgets(buf, BUF_SIZE, fp);
37.         sendto(send_sock, buf, strlen(buf),
38.             0, (struct sockaddr*)&broad_adr, sizeof(broad_adr));
39.         sleep(2);
40.     }
41.     close(send_sock);
42.     return 0;
43. }
44.
45. void error_handling(char *message)
46. {
47.     fputs(message, stderr);
48.     fputc('\n', stderr);
49.     exit(1);
50. }
```

第29行更改第23行创建的UDP套接字的可选项，使其能够发送广播数据，其余部分与UDP Sender一致。接下来给出广播Receiver。

❖ news_receiver_brd.c

```
1.  #include <"与news_sender_brd.c的头声明一致，故省略。">
2.  #define BUF_SIZE 30
```

```
3.  void error_handling(char *message);
4.
5.  int main(int argc, char *argv[])
6.  {
7.      int recv_sock;
8.      struct sockaddr_in adr;
9.      int str_len;
10.     char buf[BUF_SIZE];
11.     if(argc!=2) {
12.         printf("Usage : %s <PORT>\n", argv[0]);
13.         exit(1);
14.     }
15.
16.     recv_sock=socket(PF_INET, SOCK_DGRAM, 0);
17.     memset(&adr, 0, sizeof(adr));
18.     adr.sin_family=AF_INET;
19.     adr.sin_addr.s_addr=htonl(INADDR_ANY);
20.     adr.sin_port=htons(atoi(argv[1]));
21.
22.     if(bind(recv_sock, (struct sockaddr*)&adr, sizeof(adr))==-1)
23.         error_handling("bind() error");
24.     while(1)
25.     {
26.         str_len=recvfrom(recv_sock, buf, BUF_SIZE-1, 0, NULL, 0);
27.         if(str_len<0)
28.             break;
29.         buf[str_len]=0;
30.         fputs(buf, stdout);
31.     }
32.     close(recv_sock);
33.     return 0;
34. }
35.
36. void error_handling(char *message)
37. {
38.     //与news_sender_brd.c的error_handling函数一致。
39. }
```

源代码中没有需要特别讲解的内容，直接给出运行结果。我给出的是本地广播的运行结果，有条件的话，希望各位进一步验证直接广播的运行结果。

❖ 运行结果：news_sender_brd.c

```
root@my_linux:/tcpip# gcc news_sender_brd.c -o sender
root@my_linux:/tcpip# ./sender 255.255.255.255 9190
```

❖ 运行结果：news_receiver_brd.c

```
root@my_linux:/tcpip# gcc news_receiver_brd.c -o receiver
root@my_linux:/tcpip# ./receiver 9190
```

```
accountants say that the committee will in all likelihood tackle
issues that have been previously raised by them with the ministry.
Last year, the ministry mandated the Financial Supervisory
Service (FSS) to create over 1,000 new CPAs - a first in Korea's
accounting history - citing the need to promote sound accounting
practices industry wide.
```

14.3　基于 Windows 的实现

在Windows平台实现上述示例无需改动，因为之前的内容同样适用于此。只是多播示例中，头文件声明稍有区别。为了说明这一点，将之前的多播示例移植到Windows平台。

❖ news_sender_win.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <winsock2.h>
5.  #include <ws2tcpip.h>   // for IP_MULTICAST_TTL option
6.
7.  #define TTL 64
8.  #define BUF_SIZE 30
9.  void ErrorHandling(char *message);
10.
11. int main(int argc, char *argv[])
12. {
13.     WSADATA wsaData;
14.     SOCKET hSendSock;
15.     SOCKADDR_IN mulAdr;
16.     int timeLive=TTL;
17.     FILE *fp;
18.     char buf[BUF_SIZE];
19.
20.     if(argc!=3) {
21.         printf("Usage : %s <GroupIP> <PORT>\n", argv[0]);
22.         exit(1);
23.     }
24.     if(WSAStartup(MAKEWORD(2, 2), &wsaData)!=0)
25.         ErrorHandling("WSAStartup() error!");
26.
27.     hSendSock=socket(PF_INET, SOCK_DGRAM, 0);
28.     memset(&mulAdr, 0, sizeof(mulAdr));
29.     mulAdr.sin_family=AF_INET;
30.     mulAdr.sin_addr.s_addr=inet_addr(argv[1]);
31.     mulAdr.sin_port=htons(atoi(argv[2]));
32.
33.     setsockopt(hSendSock, IPPROTO_IP,
34.         IP_MULTICAST_TTL, (void*)&timeLive, sizeof(timeLive));
```

```
35.     if((fp=fopen("news.txt", "r"))==NULL)
36.         ErrorHandling("fopen() error");
37.     while(!feof(fp))
38.     {
39.         fgets(buf, BUF_SIZE, fp);
40.         sendto(hSendSock, buf, strlen(buf),
41.             0, (SOCKADDR*)&mulAdr, sizeof(mulAdr));
42.         Sleep(2000);
43.     }
44.     closesocket(hSendSock);
45.     WSACleanup();
46.     return 0;
47. }
48.
49. void ErrorHandling(char *message)
50. {
51.     fputs(message, stderr);
52.     fputc('\n', stderr);
53.     exit(1);
54. }
```

上述示例的第5行增加了头文件ws2tcpip.h的声明，该头文件中定义了第34行插入的IP_MULTICAST_TTL选项。接下来给出的Receivr示例中同样需要该头文件声明，因为其中定义了ip_mreq结构体。

❖ news_receiver_win.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <winsock2.h>
5.  #include <ws2tcpip.h>   // for struct ip_mreq
6.
7.  #define BUF_SIZE 30
8.  void ErrorHandling(char *message);
9.
10. int main(int argc, char *argv[])
11. {
12.     WSADATA wsaData;
13.     SOCKET hRecvSock;
14.     SOCKADDR_IN adr;
15.     struct ip_mreq joinAdr;
16.     char buf[BUF_SIZE];
17.     int strLen;
18.
19.     if(argc!=3) {
20.         printf("Usage : %s <GroupIP> <PORT>\n", argv[0]);
21.         exit(1);
22.     }
23.     if(WSAStartup(MAKEWORD(2, 2), &wsaData)!=0)
24.         ErrorHandling("WSAStartup() error!");
```

```
25.
26.     hRecvSock=socket(PF_INET, SOCK_DGRAM, 0);
27.     memset(&adr, 0, sizeof(adr));
28.     adr.sin_family=AF_INET;
29.     adr.sin_addr.s_addr=htonl(INADDR_ANY);
30.     adr.sin_port=htons(atoi(argv[2]));
31.     if(bind(hRecvSock, (SOCKADDR*) &adr, sizeof(adr))==SOCKET_ERROR)
32.         ErrorHandling("bind() error");
33.
34.     joinAdr.imr_multiaddr.s_addr=inet_addr(argv[1]);
35.     joinAdr.imr_interface.s_addr=htonl(INADDR_ANY);
36.     if(setsockopt(hRecvSock, IPPROTO_IP, IP_ADD_MEMBERSHIP,
37.             (void*)&joinAdr, sizeof(joinAdr))==SOCKET_ERROR)
38.         ErrorHandling("setsock() error");
39.
40.     while(1)
41.     {
42.         strLen=recvfrom(hRecvSock, buf, BUF_SIZE-1, 0, NULL, 0);
43.         if(strLen<0)
44.             break;
45.         buf[strLen]=0;
46.         fputs(buf, stdout);
47.     }
48.     closesocket(hRecvSock);
49.     WSACleanup();
50.     return 0;
51. }
52.
53. void ErrorHandling(char *message)
54. {
55.     fputs(message, stderr);
56.     fputc('\n', stderr);
57.     exit(1);
58. }
```

运行结果与之前的示例相同，故省略。如果运行时使用正确的多播IP地址，并在单一主机中进行测试，就会得到正确结果。

14.4　习题

(1) TTL的含义是什么？请从路由器的角度说明较大的TTL值与较小的TTL值之间的区别及问题。

(2) 多播与广播的异同点是什么？请从数据通信的角度进行说明。

(3) 下列关于多播的描述错误的是？

a. 多播是用来向加入多播组的所有主机传输数据的协议。

b. 主机连接到同一网络才能加入多播组，也就是说，多播组无法跨越多个网络。