第 12 章 I/O复用

12

本章将讨论并发服务器的第二种实现方法——基于 I/O 复用（Multiplexing）的服务器端构建。虽然通过本章多学习一种服务器端实现方法非常重要，但更重要的是理解每种技术的优缺点。如果能掌握每种技术的优劣，就可以根据特定目标灵活应用不同模型，而不是仅关注功能实现。

12.1 基于 I/O 复用的服务器端

接下来讨论并发服务器实现方法的延伸。如果有读者已经跳过第10章和第11章，那就只需把本章内容当做并发服务器实现的第一种方法即可。将要讨论的内容中包含一部分与多进程服务器端的比较，跳过第10章和第11章的读者简单浏览即可。

多进程服务器端的缺点和解决方法

为了构建并发服务器，只要有客户端连接请求就会创建新进程。这的确是实际操作中采用的一种方案，但并非十全十美，因为创建进程时需要付出极大代价。这需要大量的运算和内存空间，由于每个进程都具有独立的内存空间，所以相互间的数据交换也要求采用相对复杂的方法（IPC属于相对复杂的通信方法）。各位应该也感到需要IPC时会提高编程难度。

“那有何解决方案？能否在不创建进程的同时向多个客户端提供服务？”

当然能！本节讲解的I/O复用就是这种技术。大家听到有这种方法是否感到一阵兴奋？但请不要过于依赖该模型！该方案并不适用于所有情况，应当根据目标服务器端的特点采用不同实现方法。下面先理解“复用”（Multiplexing）的意义。

理解复用

“复用”在电子及通信工程领域很常见，向这些领域的专家询问其概念时，他们会亲切地进行如下说明：

“在1个通信频道中传递多个数据（信号）的技术。”

能理解吗？不能的话就再看看“复用”的含义。

“为了提高物理设备的效率，用最少的物理要素传递最多数据时使用的技术。”

上述两种说法内容完全一致，只是描述方式有所区别。下面我根据自己的理解进行解释。图12-1中给出的是纸杯电话，相信大家上小学时也做过。

图12-1 3方对话纸杯电话系统

图12-1是远距离的3人可以同时通话的3方对话纸杯电话系统。为使3人同时对话，需准备图中所示系统。另外，为了完成3人对话，说话时需同时对着两个纸杯，接听时也需要耳朵同时对准两个纸杯。此时引入复用技术会使通话更加方便，如图12-2所示。

图12-2 纸杯电话系统中引入复用技术

我们上小学时做过类似系统（把线捆在中间并绷直）。构建这种系统就无需同时使用两个杯

子，可以说小学就学过“复用”的概念了。接下来讨论复用技术的优点。

- 减少连线长度。
- 减少纸杯个数。

即使减少了连线和纸杯的量仍能进行3人通话，当然也有人在考虑如下这种情况：

“好像不能同时说话？”

实际上，因为是在进行对话，所以很少发生同时说话的情况。也就是说，上述系统采用的是“时（time）分复用技术”。而且，因为说话人声高（频率）不同，即使同时说话也能进行一定程度的区分（当然杂音也随之增多）。因此，也可以说系统同时采用了“频（frequency）分复用技术”。这样大家就能理解之前讲的“复用”的定义了。

复用技术在服务器端的应用

纸杯电话系统引入复用技术后，可以减少纸杯数和连线长度。同样，服务器端引入复用技术可以减少所需进程数。为便于比较，先给出第10章的多进程服务器端模型，如图12-3所示。

图12-3　多进程服务器端模型

图12-3的模型中引入复用技术，可以减少进程数。重要的是，无论连接多少客户端，提供服务的进程只有1个。

图12-4　I/O复用服务器端模型

以上就是I/O复用服务器端模型的讲解，下面考虑通过1个进程向多个客户端提供服务的方法。

知识补给站 关于I/O复用服务器端的另一种理解

某教室中有10名学生和1位教师，这些孩子并非等闲之辈，上课时不停地提问。学校没办法，只能给每个学生都配1位教师，也就是说教室中现有10位教师。此后，只要有新的转校生，就会增加1位教师，因为转校生也喜欢提问。这个故事中，如果把学生当作客户端，把教师当作与客户端进行数据交换的服务器端进程，则该教室的运营方式为多进程服务器端方式。

有一天，该校来了位具有超能力的教师。这位教师可以应对所有学生的提问，而且回答速度很快，不会让学生等待。因此，学校为了提高教师效率，将其他老师转移到了别的班。现在，学生提问前必须举手，教师确认举手学生的提问后再回答问题。也就是说，现在的教室以I/O复用方式运行。

虽然例子有些奇怪，但可以通过它理解I/O复用技法：教师必须确认有无举手学生，同样，I/O复用服务器端的进程需要确认举手（收到数据）的套接字，并通过举手的套接字接收数据。

12.2 理解 select 函数并实现服务器端

运用select函数是最具代表性的实现复用服务器端方法。Windows平台下也有同名函数提供相同功能，因此具有良好的移植性。

select 函数的功能和调用顺序

使用select函数时可以将多个文件描述符集中到一起统一监视，项目如下。

- 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？

监视项称为“事件”（event）

上述监视项称为“事件”。发生监视项对应情况时，称“发生了事件”。这是最常见的表达，希望各位熟悉。另外，本章不会使用术语“事件”，而与本章密切相关的第 17 章将使用该术语，希望大家理解“事件”的含义，以及“发生事件”的意义。

select函数的使用方法与一般函数区别较大，更准确地说，它很难使用。但为了实现I/O复用服务器端，我们应掌握select函数，并运用到套接字编程中。认为“select函数是I/O复用的全部内容”也并不为过。接下来介绍select函数的调用方法和顺序，如图12-5所示。

图12-5　select函数调用过程

图12-5给出了从调用select函数到获取结果所经过程。可以看到，调用select函数前需要一些准备工作，调用后还需查看结果。接下来按照上述顺序逐一讲解。

设置文件描述符

利用select函数可以同时监视多个文件描述符。当然，监视文件描述符可以视为监视套接字。此时首先需要将要监视的文件描述符集中到一起。集中时也要按照监视项（接收、传输、异常）进行区分，即按照上述3种监视项分成3类。

使用fd_set数组变量执行此项操作，如图12-6所示。该数组是存有0和1的位数组。

图12-6　fd_set结构体

图12-6中最左端的位表示文件描述符0（所在位置）。如果该位设置为1，则表示该文件描述符是监视对象。那么图中哪些文件描述符是监视对象呢？很明显，是文件描述符1和3。

“是否应当通过文件描述符的数字直接将值注册到fd_set变量？”

当然不是！针对fd_set变量的操作是以位为单位进行的，这也意味着直接操作该变量会比较繁琐。难道要求各位自己完成吗？实际上，在fd_set变量中注册或更改值的操作都由下列宏完成。

- FD_ZERO(fd_set * fdset)：将fd_set变量的所有位初始化为0。
- FD_SET(int fd, fd_set * fdset)：在参数fdset指向的变量中注册文件描述符fd的信息。
- FD_CLR(int fd, fd_set * fdset)：从参数fdset指向的变量中清除文件描述符fd的信息。
- FD_ISSET(int fd, fd_set * fdset)：若参数fdset指向的变量中包含文件描述符fd的信息，则返回“真”。

上述函数中，FD_ISSET用于验证select函数的调用结果。通过图12-7解释这些函数的功能，简洁易懂，无需赘述。

图12-7 fd_set相关函数的功能

设置检查（监视）范围及超时

下面讲解图12-5中步骤一的剩余内容，在此之前先简单介绍select函数。

```
#include <sys/select.h>
#include <sys/time.h>

int select(
int maxfd, fd_set * readset, fd_set * writeset, fd_set * exceptset, const struct
    timeval * timeout);
```

➔ 成功时返回大于 0 的值，失败时返回-1。

- maxfd　监视对象文件描述符数量。
- readset　将所有关注“是否存在待读取数据”的文件描述符注册到fd_set型变量，并传递其地址值。
- writeset　将所有关注“是否可传输无阻塞数据”的文件描述符注册到fd_set型变量，并传递其地址值。
- exceptset　将所有关注“是否发生异常”的文件描述符注册到fd_set型变量，并传递其地址值。
- timeout　调用select函数后，为防止陷入无限阻塞的状态，传递超时（time-out）信息。
- 返回值　发生错误时返回-1，超时返回时返回0。因发生关注的事件返回时，返回大于0的值，该值是发生事件的文件描述符数。

如上所述，select函数用来验证3种监视项的变化情况。根据监视项声明3个fd_set型变量，分别向其注册文件描述符信息，并把变量的地址值传递到上述函数的第二到第四个参数。但在此之前（调用select函数前）需要决定下面2件事。

“文件描述符的监视（检查）范围是？”

“如何设定select函数的超时时间？”

第一，文件描述符的监视范围与select函数的第一个参数有关。实际上，select函数要求通过第一个参数传递监视对象文件描述符的数量。因此，需要得到注册在fd_set变量中的文件描述符数。但每次新建文件描述符时，其值都会增1，故只需将最大的文件描述符值加1再传递到select函数即可。加1是因为文件描述符的值从0开始。

第二，select函数的超时时间与select函数的最后一个参数有关，其中timeval结构体定义如下。

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

本来select函数只有在监视的文件描述符发生变化时才返回。如果未发生变化，就会进入阻塞状态。指定超时时间就是为了防止这种情况的发生。通过声明上述结构体变量，将秒数填入tv_sec成员，将毫秒数填入tv_usec成员，然后将结构体的地址值传递到select函数的最后一个参数。此时，即使文件描述符中未发生变化，只要过了指定时间，也可以从函数中返回。不过这种情况下，select函数返回0。因此，可以通过返回值了解返回原因。如果不想设置超时，则传递NULL参数。

调用 select 函数后查看结果

虽未给出具体示例，但图12-5中的步骤一“select函数调用前的所有准备工作”已讲解完毕，同时也介绍了select函数。而函数调用后查看结果也同样重要。我们已讨论过select函数的返回值，如果返回大于0的整数，说明相应数量的文件描述符发生变化。

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. }
```

- 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. }
```

- 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. }
83. closesocket(hServSock);
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. }
```

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

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. }
```

- 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. }
```

- 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. }
```

- 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. }
```

- 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. }
```

- 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

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <winsock2.h>
4.
5.  #define BUF_SIZE 30
6.  void ErrorHandling(char *message);
7.
8.  int main(int argc, char *argv[])
9.  {
10.     WSADATA wsaData;
11.     SOCKET hSocket;
12.     SOCKADDR_IN sendAdr;
13.     if(argc!=3) {
14.         printf("Usage : %s <IP> <port>\n", argv[0]);
15.         exit(1);
16.     }
17.
18.     if(WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
19.         ErrorHandling("WSAStartup() error!");
20.
21.     hSocket=socket(PF_INET, SOCK_STREAM, 0);
22.     memset(&sendAdr, 0, sizeof(sendAdr));
23.     sendAdr.sin_family=AF_INET;
24.     sendAdr.sin_addr.s_addr=inet_addr(argv[1]);
25.     sendAdr.sin_port=htons(atoi(argv[2]));
26.
27.     if(connect(hSocket, (SOCKADDR*)&sendAdr, sizeof(sendAdr))==SOCKET_ERROR)
28.         ErrorHandling("connect() error!");
29.
30.     send(hSocket, "123", 3, 0);
31.     send(hSocket, "4", 1, MSG_OOB);
32.     send(hSocket, "567", 3, 0);
33.     send(hSocket, "890", 3, MSG_OOB);
34.
35.     closesocket(hSocket);
36.     WSACleanup();
37.     return 0;
38. }
39.
40. void ErrorHandling(char *message)
41. {
42.     fputs(message, stderr);
43.     fputc('\n', stderr);
44.     exit(1);
45. }
```

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

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <winsock2.h>
4.
5.  #define BUF_SIZE 30
6.  void ErrorHandling(char *message);
7.
8.  int main(int argc, char *argv[])
9.  {
10.     WSADATA wsaData;
11.     SOCKET hAcptSock, hRecvSock;
12.
13.     SOCKADDR_IN recvAdr;
14.     SOCKADDR_IN sendAdr;
15.     int sendAdrSize, strLen;
16.     char buf[BUF_SIZE];
17.     int result;
18.
19.     fd_set read, except, readCopy, exceptCopy;
20.     struct timeval timeout;
21.
22.     if(argc!=2) {
23.         printf("Usage : %s <port>\n", argv[0]);
24.         exit(1);
25.     }
26.
27.     if(WSAStartup(MAKEWORD(2, 2), &wsaData)!=0)
28.         ErrorHandling("WSAStartup() error!");
29.
30.     hAcptSock=socket(PF_INET, SOCK_STREAM, 0);
31.     memset(&recvAdr, 0, sizeof(recvAdr));
32.     recvAdr.sin_family=AF_INET;
33.     recvAdr.sin_addr.s_addr=htonl(INADDR_ANY);
34.     recvAdr.sin_port=htons(atoi(argv[1]));
35.
36.     if(bind(hAcptSock, (SOCKADDR*)&recvAdr, sizeof(recvAdr))==SOCKET_ERROR)
37.         ErrorHandling("bind() error");
38.     if(listen(hAcptSock, 5)==SOCKET_ERROR)
39.         ErrorHandling("listen() error");
40.
41.     sendAdrSize=sizeof(sendAdr);
42.     hRecvSock=accept(hAcptSock, (SOCKADDR*)&sendAdr, &sendAdrSize);
43.     FD_ZERO(&read);
44.     FD_ZERO(&except);
45.     FD_SET(hRecvSock, &read);
46.     FD_SET(hRecvSock, &except);
47.
48.     while(1)
49.     {
50.         readCopy=read;
51.         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

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. }
```

- 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. }
```

- 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。

c. 能够加入多播组的主机数并无限制，但只能有1个主机（Sender）向该组发送数据。

d. 多播时使用的套接字是UDP套接字，因为多播是基于UDP进行数据通信的。

(4) 多播也对网络流量有利，请比较TCP数据交换方式解释其原因。

(5) 多播方式的数据通信需要MBone虚拟网络。换言之，MBone是用于多播的网络，但它是虚拟网络。请解释此处的“虚拟网络”。

Part 02

基于Linux的编程

第 15 章 套接字和标准I/O

我们之前采用的都是默认数据通信手段 read & write 函数及各种系统 I/O 函数，可能有些读者想使用学习 C 语言时掌握的标准 I/O 函数。大家也认为在网络数据交换时使用标准 I/O 函数是非常吸引人的事情吧？

15.1 标准 I/O 函数的优点

本章将介绍利用标准I/O函数收发数据的方法。如果各位不太熟悉或已忘记多种标准函数，最好准备一本C语言的书再开始本章的学习。当然，如果熟练掌握了文件操作时使用的fopen、feof、fgetc、fputs函数，就没必要再参考C语言书了。

标准 I/O 函数的两个优点

将标准I/O函数用于数据通信并非难事。但仅掌握函数使用方法并没有太大意义，至少应该了解这些函数具有的优点。下面列出的是标准I/O函数的两大优点。

- 标准I/O函数具有良好的移植性（Portability）。
- 标准I/O函数可以利用缓冲提高性能。

关于移植性无需过多解释。不仅是I/O函数，所有标准函数具有良好的移植性。因为，为了支持所有操作系统（编译器），这些函数都是按照ANSI C标准定义的。当然，这并不局限于网络编程，而是适用于所有编程领域。

接下来讨论标准I/O函数的第二个优点。使用标准I/O函数时会得到额外的缓冲支持。这种表达方式也许会带来一些混乱，因为之前讲过，创建套接字时操作系统会准备I/O缓冲。造成更大混乱之前，先说明这两种缓冲之间的关系。创建套接字时，操作系统将生成用于I/O的缓冲。此缓冲在执行TCP协议时发挥着非常重要的作用。此时若使用标准I/O函数，将得到额外的另一缓冲的支持，如图15-1所示。

图15-1 缓冲的关系

从图15-1中可以看到，使用标准I/O函数传输数据时，经过2个缓冲。例如，通过fputs函数传输字符串“Hello”时，首先将数据传递到标准I/O函数的缓冲。然后数据将移动到套接字输出缓冲，最后将字符串发送到对方主机。

既然知道了两个缓冲的关系，接下来再说明各自的用途。设置缓冲的主要目的是为了提高性能，但套接字中的缓冲主要是为了实现TCP协议而设立的。例如，TCP传输中丢失数据时将再次传递，而再次发送数据则意味着在某地保存了数据。存在什么地方呢？套接字的输出缓冲。与之相反，使用标准I/O函数缓冲的主要目的是为了提高性能。

“使用缓冲可以大大提高性能吗？”

实际上，缓冲并非在所有情况下都能带来卓越的性能。但需要传输的数据越多，有无缓冲带来的性能差异越大。可以通过如下两种角度说明性能的提高。

- 传输的数据量
- 数据向输出缓冲移动的次数

比较1个字节的数据发送10次（10个数据包）的情况和累计10个字节发送1次的情况。发送数据时使用的数据包中含有头信息。头信息与数据大小无关，是按照一定的格式填入的。即使假设该头信息占用40个字节（实际更大），需要传递的数据量也存在较大差别。

- 1个字节 10次 40×10=400字节
- 10个字节 1次 40×1=40字节

另外，为了发送数据，向套接字输出缓冲移动数据也会消耗不少时间。但这同样与移动次数有关。1个字节数据共移动10次花费的时间将近10个字节数据移动1次花费时间的10倍。

标准 I/O 函数和系统函数之间的性能对比

前面讲解了缓冲可以提升性能的原因，但只停留在理论分析层面。接下来分别利用标准I/O函数和系统函数编写文件复制程序，这主要是为了检验缓冲提高性能的程度。首先是利用系统函数复制文件的示例。

❖ syscpy.c

```
1.  #include <stdio.h>
2.  #include <fcntl.h>
3.  #define BUF_SIZE 3      //用最短数组长度构成
4.
5.  int main(int argc, char *argv[])
6.  {
7.      int fd1, fd2;     //保存在fd1和fd2中的是文件描述符!
8.      int len;
9.      char buf[BUF_SIZE];
10.
11.     fd1=open("news.txt", O_RDONLY);
12.     fd2=open("cpy.txt", O_WRONLY|O_CREAT|O_TRUNC);
13.
14.     while((len=read(fd1, buf, sizeof(buf)))>0)
15.         write(fd2, buf, len);
16.
17.     close(fd1);
18.     close(fd2);
19.     return 0;
20. }
```

上述示例是各位很容易分析的基于read&write函数的文件复制程序。复制对象仅限于文本文件，并且是300M字节以上的文件！因为只有这样才能明显感觉到性能差异。文件名为news.txt，大家可以适当修改并测试。

各位是否正在复制文件？如果按照我的要求正在复制300M字节以上的文件，可以去趟洗手间；若不想去，可以去喝杯咖啡。如果使用未提供缓冲的read&write函数传输数据，向目的地发送需要花费很长时间。下列示例采用标准I/O函数复制文件。

❖ stdcpy.c

```
1.  #include <stdio.h>
2.  #define BUF_SIZE 3  //用最短数组长度构成
3.
4.  int main(int argc, char *argv[])
5.  {
6.      FILE * fp1; //保存在fp1中的是FILE结构体指针
7.      FILE * fp2; //保存在fp2中的是FILE结构体指针
8.      char buf[BUF_SIZE];
9.
10.     fp1=fopen("news.txt", "r");
11.     fp2=fopen("cpy.txt", "w");
12.
13.     while(fgets(buf, BUF_SIZE, fp1)!=NULL)
14.         fputs(buf, fp2);
15.
16.     fclose(fp1);
17.     fclose(fp2);
```

```
18.     return 0;
19. }
```

上述示例利用示例syscpy.c中复制的文件再次进行复制。该示例利用fputs&fgets函数复制文件，因此是一种基于缓冲的复制。各位是否执行过复制？不用去别的地方歇息，只需原地活动片刻即可完成。其实现在的300M字节并非大数据，即便如此，在单纯的文件复制操作中也会有如此大的差异。可以想象，在实际网络环境中将产生更大区别。

标准 I/O 函数的几个缺点

如果就此结束说明，各位可能认为标准I/O函数只有优点。其实它同样有缺点，整理如下。

- 不容易进行双向通信。
- 有时可能频繁调用fflush函数。
- 需要以FILE结构体指针的形式返回文件描述符。

假设各位已掌握了C语言中的绝大部分文件I/O相关知识。打开文件时，如果希望同时进行读写操作，则应以r+、w+、a+模式打开。但因为缓冲的缘故，每次切换读写工作状态时应调用fflush函数。这也会影响基于缓冲的性能提高。而且，为了使用标准I/O函数，需要FILE结构体指针（以下简称“FILE指针”）。而创建套接字时默认返回文件描述符，因此需要将文件描述符转化为FILE指针。若各位难以分清FILE指针和文件描述符，可以通过上述syscpy.c和stdcpy.c示例加以区分。

15.2 使用标准 I/O 函数

如前所述，创建套接字时返回文件描述符，而为了使用标准I/O函数，只能将其转换为FILE结构体指针。先介绍其转换方法。

利用 fdopen 函数转换为 FILE 结构体指针

可以通过fdopen函数将创建套接字时返回的文件描述符转换为标准I/O函数中使用的FILE结构体指针。

```
#include <stdio.h>

FILE * fdopen(int fildes, const char * mode);
```

➔ 成功时返回转换的 FILE 结构体指针，失败时返回 NULL。

- fildes　需要转换的文件描述符。
- mode　将要创建的FILE结构体指针的模式（mode）信息。

上述函数的第二个参数与fopen函数中的打开模式相同。常用的参数有读模式“r”和写模式“w”。下面通过简单示例给出上述函数的使用方法。

❖ desto.c

```
1.  #include <stdio.h>
2.  #include <fcntl.h>
3.
4.  int main(void)
5.  {
6.      FILE *fp;
7.      int fd=open("data.dat", O_WRONLY|O_CREAT|O_TRUNC);
8.      if(fd==-1)
9.      {
10.         fputs("file open error", stdout);
11.         return -1;
12.     }
13.
14.     fp=fdopen(fd, "w");
15.     fputs("Network C programming \n", fp);
16.     fclose(fp);
17.     return 0;
18. }
```

- 第7行：使用open函数创建文件并返回文件描述符。
- 第14行：调用fdopen函数将文件描述符转换为FILE指针。此时向第二个参数传递了“w”，因此返回写模式的FILE指针。
- 第15行：利用第14行获取的指针调用标准输出函数fputs。
- 第16行：利用FILE指针关闭文件。此时完全关闭，因此无需再通过文件描述符关闭。而且调用fclose函数后，文件描述符也变成毫无意义的整数。

❖ 运行结果：desto.c

```
root@my_linux:/tcpip# gcc desto.c -o desto
root@my_linux:/tcpip# ./desto
root@my_linux:/tcpip# cat data.dat
Network C programming
```

此示例中需要注意的是，文件描述符转换为FILE指针，并可以通过该指针调用标准I/O函数。

利用 fileno 函数转换为文件描述符

接下来介绍与fdopen函数提供相反功能的函数，该函数在有些情况下非常有用。

```
#include <stdio.h>

int fileno(FILE * stream);
```

➔ 成功时返回转换后的文件描述符，失败时返回-1。

此函数的用法也非常简单，向该函数传递FILE指针参数时返回相应文件描述符。接下来给出fileno函数的调用示例。

❖ todes.c

```
1.  #include <stdio.h>
2.  #include <fcntl.h>
3.
4.  int main(void)
5.  {
6.      FILE *fp;
7.      int fd=open("data.dat", O_WRONLY|O_CREAT|O_TRUNC);
8.      if(fd==-1)
9.      {
10.         fputs("file open error", stdout);
11.         return -1;
12.     }
13.
14.     printf("First file descriptor: %d \n", fd);
15.     fp=fdopen(fd, "w");
16.     fputs("TCP/IP SOCKET PROGRAMMING \n", fp);
17.     printf("Second file descriptor: %d \n", fileno(fp));
18.     fclose(fp);
19.     return 0;
20. }
```

- 第14行：输出第7行返回的文件描述符整数值。
- 第15、17行：第15行调用fdopen函数将文件描述符转换为FILE指针，第17行调用fileno函数再次转回文件描述符，并输出该整数值。

❖ 运行结果：todes.c

```
root@my_linux:/tcpip# gcc todes.c -o todes
root@my_linux:/tcpip# ./todes
First file descriptor: 3
Second file descriptor: 3
```

第14行和第17行输出的文件描述符值相同，证明fileno函数正确转换了文件描述符。

15.3 基于套接字的标准 I/O 函数使用

前面介绍了标准I/O函数的优缺点，同时介绍了文件描述符转换为FILE指针的方法。下面将其适用于套接字。虽然是套接字操作，但并没有需要另外说明的内容，只需简单应用这些函数。接下来将之前的回声服务器端和客户端改为基于标准I/O函数的数据交换形式，更改对象如下。

- 回声服务器端：第4章的echo_server.c
- 回声客户端：第4章的echo_client.c

无论是服务器端还是客户端，更改方式并无差异。只需调用fdopen函数并使用标准I/O函数，相信各位也能自行更改。首先给出更改后的服务器端代码。

❖ echo_stdserv.c

```
1.  #include <"头文件声明与第4章的echo_server.c一致。">
2.  #define BUF_SIZE 1024
3.  void error_handling(char *message);
4.
5.  int main(int argc, char *argv[])
6.  {
7.      int serv_sock, clnt_sock;
8.      char message[BUF_SIZE];
9.      int str_len, i;
10.
11.     struct sockaddr_in serv_adr;
12.     struct sockaddr_in clnt_adr;
13.     socklen_t clnt_adr_sz;
14.     FILE * readfp;
15.     FILE * writefp;
16.     if(argc!=2) {
17.         printf("Usage : %s <port>\n", argv[0]);
18.         exit(1);
19.     }
20.
21.     serv_sock=socket(PF_INET, SOCK_STREAM, 0);
22.     if(serv_sock==-1)
23.         error_handling("socket() error");
24.
25.     memset(&serv_adr, 0, sizeof(serv_adr));
26.     serv_adr.sin_family=AF_INET;
27.     serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
28.     serv_adr.sin_port=htons(atoi(argv[1]));
29.
30.     if(bind(serv_sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr))==-1)
31.         error_handling("bind() error");
32.     if(listen(serv_sock, 5)==-1)
33.         error_handling("listen() error");
34.     clnt_adr_sz=sizeof(clnt_adr);
35.
36.     for(i=0; i<5; i++)
```

```
37.     {
38.         clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_adr, &clnt_adr_sz);
39.         if(clnt_sock==-1)
40.             error_handling("accept() error");
41.         else
42.             printf("Connected client %d \n", i+1);
43.
44.         readfp=fdopen(clnt_sock, "r");
45.         writefp=fdopen(clnt_sock, "w");
46.         while(!feof(readfp))
47.         {
48.             fgets(message, BUF_SIZE, readfp);
49.             fputs(message, writefp);
50.             fflush(writefp);
51.         }
52.         fclose(readfp);
53.         fclose(writefp);
54.     }
55.     close(serv_sock);
56.     return 0;
57. }
58.
59. void error_handling(char *message)
60. {
61.     //与第4章的echo_server.c一致。
62. }
```

上述示例中需要注意的是第46行的循环语句。调用基于字符串的fgets、fputs函数提供服务，并在第50行调用fflush函数。标准I/O函数为了提高性能，内部提供额外的缓冲。因此，若不调用fflush函数则无法保证立即将数据传输到客户端。接下来给出回声客户端代码。

❖ echo_client.c

```
1.  #include <"头文件声明与第4章的echo_client.c一致。">
2.  #define BUF_SIZE 1024
3.  void error_handling(char *message);
4.
5.  int main(int argc, char *argv[])
6.  {
7.      int sock;
8.      char message[BUF_SIZE];
9.      int str_len;
10.     struct sockaddr_in serv_adr;
11.     FILE * readfp;
12.     FILE * writefp;
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.     if(sock==-1)
```

```
20.         error_handling("socket() error");
21.
22.     memset(&serv_adr, 0, sizeof(serv_adr));
23.     serv_adr.sin_family=AF_INET;
24.     serv_adr.sin_addr.s_addr=inet_addr(argv[1]);
25.     serv_adr.sin_port=htons(atoi(argv[2]));
26.
27.     if(connect(sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr))==-1)
28.         error_handling("connect() error!");
29.     else
30.         puts("Connected...........");
31.
32.     readfp=fdopen(sock, "r");
33.     writefp=fdopen(sock, "w");
34.     while(1)
35.     {
36.         fputs("Input message(Q to quit): ", stdout);
37.         fgets(message, BUF_SIZE, stdin);
38.         if(!strcmp(message,"q\n") || !strcmp(message,"Q\n"))
39.             break;
40.
41.         fputs(message, writefp);
42.         fflush(writefp);
43.         fgets(message, BUF_SIZE, readfp);
44.         printf("Message from server: %s", message);
45.     }
46.     fclose(writefp);
47.     fclose(readfp);
48.     return 0;
49. }
50.
51. void error_handling(char *message)
52. {
53.     //与第4章的echo_client.c一致。
54. }
```

第4章的回声客户端需要将接收的数据转换为字符串（数据的尾部插入0），但上述示例中并没有这一过程。因为，使用标准I/O函数后可以按字符串单位进行数据交换。运行结果与第4章的程序并无差异，故省略。以上就是标准I/O函数在套接字编程中的应用方法，因为需要编写额外的代码，所以并不像想象中那么常用。但某些情况下也是非常有用的，而且可以再次复习标准I/O函数，对大家也非常有益。

15.4　习题

(1) 请说明标准I/O函数的2个优点。它为何拥有这2个优点？

(2) 利用标准I/O函数传输数据时，下面的想法是错误的：

“调用fputs函数传输数据时，调用后应立即开始发送！”

为何说上述想法是错误的？为了达到这种效果应添加哪些处理过程？

第 16 章 关于I/O流分离的其他内容

调用 fopen 函数打开文件后可以与文件交换数据，因此说调用 fopen 函数后创建了“流”（Stream）。此处的“流”是指“数据流动”，但通常可以比喻为“以数据收发为目的的一种桥梁”。希望各位将“流”理解为数据收发路径。

16.1 分离 I/O 流

“分离I/O流”是一种常用表达。有I/O工具可以区分二者，无论使用何种方法，都可以认为分离了I/O流。

2 次 I/O 流分离

我们之前通过2种方法分离过I/O流，第一种是第10章的“TCP I/O过程（Routine）分离”。这种方法通过调用fork函数复制出1个文件描述符，以区分输入和输出中使用的文件描述符。虽然文件描述符本身不会根据输入和输出进行区分，但我们分开了2个文件描述符的用途，因此这也属于“流”的分离。

第二种分离在第15章。通过2次fdopen函数的调用，创建读模式FILE指针（FILE结构体指针）和写模式FILE指针。换言之，我们分离了输入工具和输出工具，因此也可视为“流”的分离。下面说明分离的理由，讨论尚未提及的问题并给出解决方案。

分离“流”的好处

第10章的“流”分离和第15章的“流”分离在目的上有一定差异。首先分析第10章的“流”分离目的。

- 通过分开输入过程（代码）和输出过程降低实现难度。
- 与输入无关的输出操作可以提高速度。

这是第10章讨论过的内容，故不再解释这些优点的原因。接下来给出第15章“流”分离的目的。

- 为了将FILE指针按读模式和写模式加以区分。
- 可以通过区分读写模式降低实现难度。
- 通过区分I/O缓冲提高缓冲性能。

“流”分离的方法、情况（目的）不同时，带来的好处也有所不同。

“流”分离带来的 EOF 问题

下面讲解“流”分离带来的问题。第7章介绍过EOF的传递方法和半关闭的必要性（如果记不清，请复习相应章节）。各位应该还记得如下函数调用语句：

```
shutdown(sock, SHUT_WR);
```

当时讲过调用shutdown函数的基于半关闭的EOF传递方法。第10章还利用这些技术在echo_mpclient.c示例中添加了半关闭相关代码。也就是说，第10章的“流”分离没有问题。但第15章的基于fdopen函数的“流”则不同，我们还不知道在这种情况下如何进行半关闭，因此有可能犯如下错误：

> “半关闭？不是可以针对输出模式的FILE指针调用fclose函数吗？这样可以向对方传递EOF，变成可以接收数据但无法发送数据的半关闭状态呀。”

各位是否也这么认为？这是一种很好的猜测，但希望大家先阅读下列代码。另外，接下来的示例中为了简化代码而未添加异常处理，希望各位不要误解。先给出服务器端代码。

❖ sep_serv.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.  #define BUF_SIZE 1024
8.
9.  int main(int argc, char *argv[])
10. {
11.     int serv_sock, clnt_sock;
12.     FILE * readfp;
13.     FILE * writefp;
14.
15.     struct sockaddr_in serv_adr, clnt_adr;
16.     socklen_t clnt_adr_sz;
17.     char buf[BUF_SIZE]={0,};
18.
```

```
19.     serv_sock=socket(PF_INET, SOCK_STREAM, 0);
20.     memset(&serv_adr, 0, sizeof(serv_adr));
21.     serv_adr.sin_family=AF_INET;
22.     serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
23.     serv_adr.sin_port=htons(atoi(argv[1]));
24.
25.     bind(serv_sock, (struct sockaddr*) &serv_adr, sizeof(serv_adr));
26.     listen(serv_sock, 5);
27.     clnt_adr_sz=sizeof(clnt_adr);
28.     clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_adr,&clnt_adr_sz);
29.
30.     readfp=fdopen(clnt_sock, "r");
31.     writefp=fdopen(clnt_sock, "w");
32.
33.     fputs("FROM SERVER: Hi~ client? \n", writefp);
34.     fputs("I love all of the world \n", writefp);
35.     fputs("You are awesome! \n", writefp);
36.     fflush(writefp);
37.
38.     fclose(writefp);
39.     fgets(buf, sizeof(buf), readfp);
40.     fputs(buf, stdout);
41.     fclose(readfp);
42.     return 0;
43. }
```

- 第30、31行：通过clnt_sock中保存的文件描述符创建读模式FILE指针和写模式FILE指针。
- 第33~36行：向客户端发送字符串，调用fflush函数结束发送过程。
- 第38、39行：第38行针对写模式FILE指针调用fclose函数。调用fclose函数终止套接字时，对方主机将收到EOF。但还剩下第30行创建的读模式FILE指针，有些人可能认为可以通过第39行的函数调用接收客户端最后发送的字符串。当然，最后的字符串是客户端收到EOF后发送的。

上述示例调用fclose函数后的确会发送EOF。稍后给出的客户端收到EOF后也会发送最后的字符串，只是需要验证第39行的函数调用能否接收。接下来给出客户端代码。

❖ sep_clnt.c

```
1.  #include <"头文件声明与sep_serv.c一致。">
2.  #define BUF_SIZE 1024
3.
4.  int main(int argc, char *argv[])
5.  {
6.      int sock;
7.      char buf[BUF_SIZE];
8.      struct sockaddr_in serv_addr;
9.
10.     FILE * readfp;
11.     FILE * writefp;
12.
```

```
13.     sock=socket(PF_INET, SOCK_STREAM, 0);
14.     memset(&serv_addr, 0, sizeof(serv_addr));
15.     serv_addr.sin_family=AF_INET;
16.     serv_addr.sin_addr.s_addr=inet_addr(argv[1]);
17.     serv_addr.sin_port=htons(atoi(argv[2]));
18.
19.     connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
20.     readfp=fdopen(sock, "r");
21.     writefp=fdopen(sock, "w");
22.
23.     while(1)
24.     {
25.         if(fgets(buf, sizeof(buf), readfp)==NULL)
26.             break;
27.         fputs(buf, stdout);
28.         fflush(stdout);
29.     }
30.
31.     fputs("FROM CLIENT: Thank you! \n", writefp);
32.     fflush(writefp);
33.     fclose(writefp); fclose(readfp);
34.     return 0;
35. }
```

- 第20、21行：为了调用标准I/O函数，创建读模式和写模式FILE指针。
- 第25行：收到EOF时，fgets函数将返回NULL指针。因此，添加if语句使收到NULL时退出循环。
- 第31行：通过该行语句向服务器端发送最后的字符串。当然，该字符串是在收到服务器端的EOF后发送的。

各位在分析代码过程中应该得知需要通过该示例验证哪些事项。接下来通过运行结果验证服务器端是否收到客户端最后发送的字符串。

❖ 运行结果：sep_serv.c

```
root@my_linux:/tcpip# gcc sep_serv.c -o serv
root@my_linux:/tcpip# ./serv 9190
root@my_linux:/tcpip#
```

❖ 运行结果：sep_clnt.c

```
root@my_linux:/tcpip# gcc sep_clnt.c -o clnt
root@my_linux:/tcpip# ./clnt 127.0.0.1 9190
FROM SERVER: Hi~ client?
I love all of the world
You are awesome!
```

```
root@my_linux:/tcpip#
```

从运行结果可得出如下结论：

“服务器端未能接收最后的字符串！”

很容易判断其原因：sep_serv.c示例的第38行调用的fclose函数完全终止了套接字，而不是半关闭。以上就是需要通过本章解决的问题。半关闭在多种情况下都非常有用，各位必须能够针对fdopen函数调用时生成的FILE指针进行半关闭操作。

16.2 文件描述符的复制和半关闭

本章主题虽然是针对FILE指针的半关闭，但本节介绍的dup和dup2函数也有助于增加系统编程经验。

终止“流”时无法半关闭的原因

图16-1描述的是sep_serv.c示例中的2个FILE指针、文件描述符及套接字之间的关系。

图16-1 FILE指针的关系

从图16-1中可以看到，示例sep_serv.c中的读模式FILE指针和写模式FILE指针都是基于同一文件描述符创建的。因此，针对任意一个FILE指针调用fclose函数时都会关闭文件描述符，也就终止套接字，如图16-2所示。

图16-2 调用fclose函数的结果

从图16-2中可以看到，销毁套接字时再也无法进行数据交换。那如何进入可以输入但无法输出的半关闭状态呢？其实很简单。如图16-3所示，创建FILE指针前先复制文件描述符即可。

图16-3 半关闭模型1

如图16-3所示，复制后另外创建1个文件描述符，然后利用各自的文件描述符生成读模式FILE指针和写模式FILE指针。这就为半关闭准备好了环境，因为套接字和文件描述符之间具有如下关系：

“销毁所有文件描述符后才能销毁套接字。”

也就是说，针对写模式FILE指针调用fclose函数时，只能销毁与该FILE指针相关的文件描述符，无法销毁套接字（参考图16-4）。

图16-4 半关闭模型2

如图16-4所示，调用fclose函数后还剩1个文件描述符，因此没有销毁套接字。那此时的状态是否为半关闭状态？不是！图16-3中讲过，只是准备好了半关闭环境。要进入真正的半关闭状态需要特殊处理。

“图16-4中好像已经进入半关闭状态了啊？”

当然可以这么看。但仔细观察，还剩1个文件描述符呢。而且该文件描述符可以同时进行I/O。因此，不但没有发送EOF，而且仍然可以利用文件描述符进行输出。稍后将介绍根据图16-3和图16-4的模型发送EOF并进入半关闭状态的方法。首先介绍如何复制文件描述符，之前的fork函数不在考虑范围内。

复制文件描述符

之前提到的文件描述符的复制与fork函数中进行的复制有所区别。调用fork函数时将复制整个进程，因此同一进程内不能同时有原件和副本。但此处讨论的复制并非针对整个进程，而是在同一进程内完成描述符的复制，如图16-5所示。

图16-5 文件描述符的复制

图16-5给出的是同一进程内存在2个文件描述符可以同时访问文件的情况。当然，文件描述符的值不能重复，因此各使用5和7的整数值。为了形成这种结构，需要复制文件描述符。此处所谓的“复制”具有如下含义：

“为了访问同一文件或套接字，创建另一个文件描述符。”

通常的“复制”很容易让人理解为将包括文件描述符整数值在内的所有内容进行复制，而此处的“复制”方式却不同。

dup & dup2

下面给出文件描述符的复制方法，通过下列2个函数之一完成。

```
#include <unistd.h>

int dup(int fildes);
int dup2(int fildes, int fildes2);
```

➔ 成功时返回复制的文件描述符，失败时返回-1。

- fildes　需要复制的文件描述符。
- fildes2　明确指定的文件描述符整数值。

dup2函数明确指定复制的文件描述符整数值。向其传递大于0且小于进程能生成的最大文件

描述符值时，该值将成为复制出的文件描述符值。下面给出示例验证函数功能，示例中将复制自动打开的标准输出的文件描述符1，并利用复制出的描述符进行输出。另外，自动打开的文件描述符0、1、2与套接字文件描述符没有区别，因此可以用来验证dup函数的功能。

❖ dup.c

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.
4.  int main(int argc, char *argv[])
5.  {
6.      int cfd1, cfd2;
7.      char str1[]="Hi~ \n";
8.      char str2[]="It's nice day~ \n";
9.
10.     cfd1=dup(1);
11.     cfd2=dup2(cfd1, 7);
12.
13.     printf("fd1=%d, fd2=%d \n", cfd1, cfd2);
14.     write(cfd1, str1, sizeof(str1));
15.     write(cfd2, str2, sizeof(str2));
16.
17.     close(cfd1);
18.     close(cfd2);
19.     write(1, str1, sizeof(str1));
20.     close(1);
21.     write(1, str2, sizeof(str2));
22.     return 0;
23. }
```

代码说明

- 第10、11行：第10行调用dup函数复制了文件描述符1。第11行调用dup2函数再次复制了文件描述符，并指定描述符整数值为7。
- 第14、15行：利用复制出的文件描述符进行输出。通过该输出结果可以验证是否进行了实际复制。
- 第17~19行：终止复制的文件描述符。但仍有1个描述符，因此可以进行输出。可以从第19行得到验证。
- 第20、21行：第20行终止最后的文件描述符，因此无法完成第21行的输出。

❖ 运行结果：dup.c

```
root@my_linux:/tcpip# gcc dup.c -o dup
root@my_linux:/tcpip# ./dup
fd1=3, fd2=7
Hi~
It's nice day~
Hi~
```

示例虽然简单，但足以表达文件描述符复制相关内容。

复制文件描述符后“流”的分离

下面更改sep_serv.c和sep_clnt.c示例，使其能够正常工作（只需更改sep_serv.c示例）。所谓“正常工作”是指，通过服务器端的半关闭状态接收客户端最后发送的字符串。当然，为了完成这一任务，服务器端需要同时发送EOF。发送EOF的代码并不难，通过示例给出。

❖ sep_serv2.c

```
1.  #include <"头文件声明与sep_serv.c一致，故省略。">
2.  #define BUF_SIZE 1024
3.
4.  int main(int argc, char *argv[])
5.  {
6.      int serv_sock, clnt_sock;
7.      FILE * readfp;
8.      FILE * writefp;
9.
10.     struct sockaddr_in serv_adr, clnt_adr;
11.     socklen_t clnt_adr_sz;
12.     char buf[BUF_SIZE]={0,};
13.
14.     serv_sock=socket(PF_INET, SOCK_STREAM, 0);
15.     memset(&serv_adr, 0, sizeof(serv_adr));
16.     serv_adr.sin_family=AF_INET;
17.     serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
18.     serv_adr.sin_port=htons(atoi(argv[1]));
19.
20.     bind(serv_sock, (struct sockaddr*) &serv_adr, sizeof(serv_adr));
21.     listen(serv_sock, 5);
22.     clnt_adr_sz=sizeof(clnt_adr);
23.     clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_adr,&clnt_adr_sz);
24.
25.     readfp=fdopen(clnt_sock, "r");
26.     writefp=fdopen(dup(clnt_sock), "w");
27.
28.     fputs("FROM SERVER: Hi~ client? \n", writefp);
29.     fputs("I love all of the world \n", writefp);
30.     fputs("You are awesome! \n", writefp);
31.     fflush(writefp);
32.
33.     shutdown(fileno(writefp), SHUT_WR);
34.     fclose(writefp);
35.
36.     fgets(buf, sizeof(buf), readfp); fputs(buf, stdout);
37.     fclose(readfp);
38.     return 0;
39. }
```

- 第25、26行：调用fdopen函数生成FILE指针。特别是第26行针对dup函数的返回值生成FILE指针，因此函数调用后将进入图16-3所示状态。
- 第33行：针对fileno函数返回的文件描述符调用shutdown函数。因此，服务器端进入半关闭状态，并向客户端发送EOF。这一行就是之前所说的发送EOF的方法。调用shutdown函数时，无论复制出多少文件描述符都进入半关闭状态，同时传递EOF。

上述示例可以结合sep_clnt.c运行。我们关心的是服务器端能否收到客户端最后的消息，因此只给出服务器端运行结果。

❖ 运行结果：sep_serv2.c

```
root@my_linux:/tcpip# gcc sep_serv2.c -o serv2
root@my_linux:/tcpip# ./serv2 9190
FROM CLIENT: Thank you!
```

运行结果证明服务器端在半关闭状态下向客户端发送了EOF。通过该示例希望各位掌握一点：

“无论复制出多少文件描述符，均应调用shutdown函数发送EOF并进入半关闭状态。”

第10章的echo_mpclient.c示例运用过shutdown函数的这种功能，当时通过fork函数生成了2个文件描述符，并在这种情况下调用shutdown函数发送了EOF。

16.3　习题

(1) 下列关于FILE结构体指针和文件描述符的说法错误的是？

a. 与FILE结构体指针相同，文件描述符也分为输入描述符和输出描述符。
b. 复制文件描述符时将生成相同值的描述符，可以通过这2个描述符进行I/O。
c. 可以利用创建套接字时返回的文件描述符进行I/O，也可以不通过文件描述符，直接通过FILE结构体指针完成。
d. 可以从文件描述符生成FILE结构体指针，而且可以利用这种FILE结构体指针进行套接字I/O。
e. 若文件描述符为读模式，则基于该描述符生成的FILE结构体指针同样是读模式；若文件描述符为写模式，则基于该描述符生成的FILE结构体指针同样是写模式。

(2) EOF的发送相关描述中错误的是？

a. 终止文件描述符时发送EOF。
b. 即使未完全终止文件描述符，关闭输出流时也会发送EOF。
c. 如果复制文件描述符，则包括复制的文件描述符在内，所有描述符都终止时才会发送EOF。
d. 即使复制文件描述符，也可以通过调用shutdown函数进入半关闭状态并发送EOF。

第 17 章 优于select的epoll

实现 I/O 复用的传统方法有 select 函数和 poll 函数。我们介绍了 select 函数的使用方法，但各种原因导致这些方法无法得到令人满意的性能。因此有了 Linux 下的 epoll、BSD 的 kqueue、Solaris 的/dev/poll 和 Windows 的 IOCP 等复用技术。本章将讲解 Linux 的 epoll 技术。

17.1 epoll 理解及应用

select复用方法其实由来已久，因此，利用该技术后，无论如何优化程序性能也无法同时接入上百个客户端（当然，硬件性能不同，差别也很大）。这种select方式并不适合以Web服务器端开发为主流的现代开发环境，所以要学习Linux平台下的epoll。

基于 select 的 I/O 复用技术速度慢的原因

第12章曾经实现过基于select的I/O复用服务器端，很容易从代码上分析出不合理的设计，最主要的两点如下。

- 调用select函数后常见的针对所有文件描述符的循环语句。
- 每次调用select函数时都需要向该函数传递监视对象信息。

上述两点可以从第12章示例echo_selectserv.c的第45、49行及第54行代码得到确认。调用select函数后，并不是把发生变化的文件描述符单独集中到一起，而是通过观察作为监视对象的fd_set变量的变化，找出发生变化的文件描述符（示例echo_selectserv.c的第54、56行），因此无法避免针对所有监视对象的循环语句。而且，作为监视对象的fd_set变量会发生变化，所以调用select函数前应复制并保存原有信息（参考echo_selectserv.c的第45行），并在每次调用select函数时传递新的监视对象信息。

各位认为哪些因素是提高性能的更大障碍？是调用select函数后常见的针对所有文件描述符

对象的循环语句？还是每次需要传递的监视对象信息？

只看代码的话很容易认为是循环。但相比于循环语句，更大的障碍是每次传递监视对象信息。因为传递监视对象信息具有如下含义：

> “每次调用select函数时向操作系统传递监视对象信息。”

应用程序向操作系统传递数据将对程序造成很大负担，而且无法通过优化代码解决，因此将成为性能上的致命弱点。

> “那为何需要把监视对象信息传递给操作系统呢？”

有些函数不需要操作系统的帮助就能完成功能，而有些则必须借助于操作系统。假设各位定义了四则运算相关函数，此时无需操作系统的帮助。但select函数与文件描述符有关，更准确地说，是监视套接字变化的函数。而套接字是由操作系统管理的，所以select函数绝对需要借助于操作系统才能完成功能。select函数的这一缺点可以通过如下方式弥补：

> “仅向操作系统传递1次监视对象，监视范围或内容发生变化时只通知发生变化的事项。”

这样就无需每次调用select函数时都向操作系统传递监视对象信息，但前提是操作系统支持这种处理方式（每种操作系统支持的程度和方式存在差异）。Linux的支持方式是epoll，Windows的支持方式是IOCP。

select 也有优点

知道这些内容后，有些人可能对select函数感到失望，但大家应当掌握select函数。本章的epoll方式只在Linux下提供支持，也就是说，改进的I/O复用模型不具有兼容性。相反，大部分操作系统都支持select函数。只要满足或要求如下两个条件，即使在Linux平台也不应拘泥于epoll。

- 服务器端接入者少。
- 程序应具有兼容性。

实际并不存在适用于所有情况的模型。各位应理解好各种模型的优缺点，并具备合理运用这些模型的能力。

实现 epoll 时必要的函数和结构体

能够克服select函数缺点的epoll函数具有如下优点，这些优点正好与之前的select函数缺点相反。

- 无需编写以监视状态变化为目的的针对所有文件描述符的循环语句。
- 调用对应于select函数的epoll_wait函数时无需每次传递监视对象信息。

下面介绍epoll服务器端实现中需要的3个函数，希望各位结合epoll函数的优点理解这些函数的功能。

- epoll_create：创建保存epoll文件描述符的空间。
- epoll_ctl：向空间注册并注销文件描述符。
- epoll_wait：与select函数类似，等待文件描述符发生变化。

select方式中为了保存监视对象文件描述符，直接声明了fd_set变量。但epoll方式下由操作系统负责保存监视对象文件描述符，因此需要向操作系统请求创建保存文件描述符的空间，此时使用的函数就是epoll_create。

此外，为了添加和删除监视对象文件描述符，select方式中需要FD_SET、FD_CLR函数。但在epoll方式中，通过epoll_ctl函数请求操作系统完成。最后，select方式下调用select函数等待文件描述符的变化，而epoll中调用epoll_wait函数。还有，select方式中通过fd_set变量查看监视对象的状态变化（事件发生与否），而epoll方式中通过如下结构体epoll_event将发生变化的（发生事件的）文件描述符单独集中到一起。


```
struct epoll_event
{
    __uint32_t  events;
    epoll_data_t  data;
}

        typedef union epoll_data
        {
            void * ptr;
            int fd;
            __uint32_t u32;
            __uint64_t u64;
        } epoll_data_t;
```

声明足够大的epoll_event结构体数组后，传递给epoll_wait函数时，发生变化的文件描述符信息将被填入该数组。因此，无需像select函数那样针对所有文件描述符进行循环。

以上就是epoll中需要的函数和结构体。实际上，只要有select程序的编写经验，epoll程序的编写就并不难。接下来给出这些函数的详细说明。

epoll_create

epoll是从Linux的2.5.44版内核（操作系统的核心模块）开始引入的，所以使用epoll前需要验证Linux内核版本。但各位使用的Linux内核基本都是2.6以上的版本，所以这部分可以忽略。若有

人怀疑自己的Linux版本过低，可以通过如下命令验证：

```
cat /proc/sys/kernel/osrelease
```

下面仔细观察epoll_create函数。

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

int epoll_create(int size);
```

➔ 成功时返回 epoll 文件描述符，失败时返回-1。

- size　　epoll实例的大小。

调用epoll_create函数时创建的文件描述符保存空间称为“epoll例程“，但有些情况下名称不同，需要稍加注意。通过参数size传递的值决定epoll例程的大小，但该值只是向操作系统提的建议。换言之，size并非用来决定epoll例程的大小，而仅供操作系统参考。

操作系统将完全忽略传递给 epoll_create 的参数

Linux 2.6.8 之后的内核将完全忽略传入 epoll_create 函数的 size 参数，因为内核会根据情况调整 epoll 例程的大小。但撰写本书时 Linux 版本未达到 2.6.8，因此无法在忽略 size 参数的情况下编写程序。

epoll_create函数创建的资源与套接字相同，也由操作系统管理。因此，该函数和创建套接字的情况相同，也会返回文件描述符。也就是说，该函数返回的文件描述符主要用与于区分epoll例程。需要终止时，与其他文件描述符相同，也要调用close函数。

epoll_ctl

生成epoll例程后，应在其内部注册监视对象文件描述符，此时使用epoll_ctl函数。

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

int epoll_ctl(int epfd, int op, int fd, struct epoll_event * event);
```

➔ 成功时返回 0，失败时返回-1。

- epfd　用于注册监视对象的epoll例程的文件描述符。
- op　用于指定监视对象的添加、删除或更改等操作。
- fd　需要注册的监视对象文件描述符。
- event　监视对象的事件类型。

与其他epoll函数相比，该函数多少有些复杂，但通过调用语句就很容易理解。假设按照如下形式调用epoll_ctl函数：

```
epoll_ctl(A, EPOLL_CTL_ADD, B, C);
```

第二个参数EPOLL_CTL_ADD意味着“添加”，因此上述语句具有如下含义：

“epoll例程A中注册文件描述符B，主要目的是监视参数C中的事件。”

再介绍一个调用语句。

```
epoll_ctl(A, EPOLL_CTL_DEL, B, NULL);
```

上述语句中第二个参数EPOLL_CTL_DEL指“删除”，因此该语句具有如下含义：。

“从epoll例程A中删除文件描述符B。”

从上述调用语句中可以看到，从监视对象中删除时，不需要监视类型（事件信息），因此向第四个参数传递NULL。接下来介绍可以向epoll_ctl第二个参数传递的常量及含义。

- EPOLL_CTL_ADD：将文件描述符注册到epoll例程。
- EPOLL_CTL_DEL：从epoll例程中删除文件描述符。
- EPOLL_CTL_MOD：更改注册的文件描述符的关注事件发生情况。

关于EPOLL_CTL_MOD常量稍后讲解（即使我不讲大家也自然能明白）。如前所述，向epoll_ctl的第二个参数传递EPOLL_CTL_DEL时，应同时向第四个参数传递NULL。但Linux 2.6.9之前的内核不允许传递NULL。虽然被忽略掉，但也应传递epoll_event结构体变量的地址值（本书示例将传递NULL）。其实这是Bug，但也没必要因此怀疑epoll的功能，因为我们使用的标准函数中也存在Bug。

下面讲解各位不太熟悉的epoll_ctl函数的第四个参数，其类型是之前将过的epoll_event结构体指针。

“啊？不是说epoll_event用于保存发生变化的（发生事件）的文件描述符吗？”

当然！如前所述，epoll_event结构体用于保存发生事件的文件描述符集合。但也可以在epoll例程中注册文件描述符时，用于注册关注的事件。函数中epoll_event结构体的定义并不显眼，因此通过调用语句说明该结构体在epoll_ctl函数中的应用。

```
struct epoll_event event;
. . . . .
event.events=EPOLLIN;  //发生需要读取数据的情况（事件）时
event.data.fd=sockfd;
epoll_ctl(epfd, EPOLL_CTL_ADD, sockfd, &event);
. . . . .
```

上述代码将sockfd注册到epoll例程epfd中，并在需要读取数据的情况下产生相应事件。接下来给出epoll_event的成员events中可以保存的常量及所指的事件类型。

- EPOLLIN：需要读取数据的情况。
- EPOLLOUT：输出缓冲为空，可以立即发送数据的情况。
- EPOLLPRI：收到OOB数据的情况。
- EPOLLRDHUP：断开连接或半关闭的情况，这在边缘触发方式下非常有用。
- EPOLLERR：发生错误的情况。
- EPOLLET：以边缘触发的方式得到事件通知。
- EPOLLONESHOT：发生一次事件后，相应文件描述符不再收到事件通知。因此需要向epoll_ctl函数的第二个参数传递EPOLL_CTL_MOD，再次设置事件。

可以通过位或运算同时传递多个上述参数。关于“边缘触发”稍后将单独讲解，目前只需记住EPOLLIN即可。

epoll_wait

最后介绍与select函数对应的epoll_wait函数，epoll相关函数中默认最后调用该函数。

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

int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int
timeout);
```

➔ 成功时返回发生事件的文件描述符数，失败时返回-1。

- epfd　表示事件发生监视范围的epoll例程的文件描述符。
- events　保存发生事件的文件描述符集合的结构体地址值。
- maxevents　第二个参数中可以保存的最大事件数。
- timeout　以1/1000秒为单位的等待时间，传递-1时，一直等待直到发生事件。

该函数的调用方式如下。需要注意的是，第二个参数所指缓冲需要动态分配。

```
int event_cnt;
struct epoll_event * ep_events;
. . . . .
ep_events = malloc(sizeof(struct epoll_event)*EPOLL_SIZE); //EPOLL_SIZE 是宏常量
. . . . .
event_cnt = epoll_wait(epfd, ep_events, EPOLL_SIZE, -1);
. . . . .
```

调用函数后，返回发生事件的文件描述符数，同时在第二个参数指向的缓冲中保存发生事件的文件描述符集合。因此，无需像select那样插入针对所有文件描述符的循环。

基于 epoll 的回声服务器端

以上就是基于epoll技术实现服务器端的所有理论说明，接下来给出基于epoll的回声服务器端示例。我通过更改第12章的echo_selectserv.c实现了该示例。当然，从头开始写也与下面给出的内容类似。但通过更改select示例理解二者差异将更有利于学习。

❖ echo_epollserv.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/epoll.h>
8.
9.  #define BUF_SIZE 100
10. #define EPOLL_SIZE 50
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.     socklen_t adr_sz;
18.     int str_len, i;
19.     char buf[BUF_SIZE];
20.
21.     struct epoll_event *ep_events;
22.     struct epoll_event event;
23.     int epfd, event_cnt;
24.
25.     if(argc!=2) {
26.         printf("Usage : %s <port>\n", argv[0]);
27.         exit(1);
```

```
28.     }
29.
30.     serv_sock=socket(PF_INET, SOCK_STREAM, 0);
31.     memset(&serv_adr, 0, sizeof(serv_adr));
32.     serv_adr.sin_family=AF_INET;
33.     serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
34.     serv_adr.sin_port=htons(atoi(argv[1]));
35.
36.     if(bind(serv_sock, (struct sockaddr*) &serv_adr, sizeof(serv_adr))==-1)
37.         error_handling("bind() error");
38.     if(listen(serv_sock, 5)==-1)
39.         error_handling("listen() error");
40.
41.     epfd=epoll_create(EPOLL_SIZE);
42.     ep_events=malloc(sizeof(struct epoll_event)*EPOLL_SIZE);
43.
44.     event.events=EPOLLIN;
45.     event.data.fd=serv_sock;
46.     epoll_ctl(epfd, EPOLL_CTL_ADD, serv_sock, &event);
47.
48.     while(1)
49.     {
50.         event_cnt=epoll_wait(epfd, ep_events, EPOLL_SIZE, -1);
51.         if(event_cnt==-1)
52.         {
53.             puts("epoll_wait() error");
54.             break;
55.         }
56.
57.         for(i=0; i<event_cnt; i++)
58.         {
59.             if(ep_events[i].data.fd==serv_sock)
60.             {
61.                 adr_sz=sizeof(clnt_adr);
62.                 clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_adr, &adr_sz);
63.                 event.events=EPOLLIN;
64.                 event.data.fd=clnt_sock;
65.                 epoll_ctl(epfd, EPOLL_CTL_ADD, clnt_sock, &event);
66.                 printf("connected client: %d \n", clnt_sock);
67.             }
68.             else
69.             {
70.                     str_len=read(ep_events[i].data.fd, buf, BUF_SIZE);
71.                     if(str_len==0)  // close request!
72.                     {
73.                         epoll_ctl(
74.                           epfd, EPOLL_CTL_DEL, ep_events[i].data.fd, NULL);
75.                         close(ep_events[i].data.fd);
76.                         printf("closed client: %d \n", ep_events[i].data.fd);
77.                     }
78.                     else
79.                     {
80.                         write(ep_events[i].data.fd, buf, str_len);    // echo!
81.                     }
```

```
82.
83.             }
84.         }
85.     }
86.     close(serv_sock);
87.     close(epfd);
88.     return 0;
89. }
90.
91. void error_handling(char *buf)
92. {
93.     fputs(buf, stderr);
94.     fputc('\n', stderr);
95.     exit(1);
96. }
```

之前解释过关键代码，而且程序结构与select方式没有区别，故省略代码说明。如果有些地方难以理解，说明未掌握本章之前的内容和select模型，建议复习。结合我的说明和select示例理解上述代码也是一种很好的学习方式。上述示例可以结合任意回声客户端运行，而且运行结果与其他回声服务器端/客户端程序没有差别，故省略。

17.2 条件触发和边缘触发

有些人学习epoll时往往无法正确区分条件触发（Level Trigger）和边缘触发（Edge Trigger），但只有理解了二者区别才算完整掌握epoll。

条件触发和边缘触发的区别在于发生事件的时间点

首先给出示例帮助各位理解条件触发和边缘触发。观察如下对话，可以通过对话内容理解条件触发事件的特点。

- 儿子："妈妈，我收到了5000元压岁钱。"
- 妈妈："恩，真棒！"

- 儿子："我给隔壁家秀熙买了炒年糕，花了2000元。"
- 妈妈："恩，做得好！"

- 儿子："妈妈，我还买了玩具，剩下500元。"
- 妈妈："用完零花钱就只能挨饿喽！"

- 儿子："妈妈，我还留着那500元没动，不会挨饿的。"
- 妈妈："恩，很明智嘛！"

- 儿子："妈妈，我还留着那500元没动，我要攒起来。"
- 妈妈："恩，加油！"

从上述对话可以看出，儿子从收到压岁钱开始一直向妈妈报告，这就是条件触发的原理。如果将上述对话中的儿子（儿子的钱包）换成输入缓冲，压岁钱换成输入数据，儿子的报告换成事件，则可以发现条件触发的特性。我将其整理如下：

"条件触发方式中，只要输入缓冲有数据就会一直通知该事件。"

例如，服务器端输入缓冲收到50字节的数据时，服务器端操作系统将通知该事件（注册到发生变化的文件描述符）。但服务器端读取20字节后还剩30字节的情况下，仍会注册事件。也就是说，条件触发方式中，只要输入缓冲中还剩有数据，就将以事件方式再次注册。接下来通过如下对话介绍边缘触发的事件特性。

- 儿子："妈妈，我收到了5000元压岁钱。"
- 妈妈："恩，再接再厉。"

- 儿子："……"
- 妈妈："说话呀！压岁钱呢？不想回答吗？"

从上述对话可以看出，边缘触发中输入缓冲收到数据时仅注册1次该事件。即使输入缓冲中还留有数据，也不会再进行注册。

掌握条件触发的事件特性

接下来通过代码了解条件触发的事件注册方式。下列代码是稍微修改之前的echo_epollserv.c示例得到的。epoll默认以条件触发方式工作，因此可以通过该示例验证条件触发的特性。

❖ echo_EPLTserv.c

```
1.  #include <"与示例echo_epollserv.c的头文件声明一致，故省略。">
2.  #define BUF_SIZE 4
3.  #define EPOLL_SIZE 50
4.  void error_handling(char *buf);
5.
6.  int main(int argc, char *argv[])
7.  {
8.      int serv_sock, clnt_sock;
9.      struct sockaddr_in serv_adr, clnt_adr;
10.     socklen_t adr_sz;
11.     int str_len, i;
12.     char buf[BUF_SIZE];
13.
14.     struct epoll_event *ep_events;
15.     struct epoll_event event;
```

```
16.     int epfd, event_cnt;
17.
18.     if(argc!=2) {
19.         printf("Usage : %s <port>\n", argv[0]);
20.         exit(1);
21.     }
22.
23.     serv_sock=socket(PF_INET, SOCK_STREAM, 0);
24.     memset(&serv_adr, 0, sizeof(serv_adr));
25.     serv_adr.sin_family=AF_INET;
26.     serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
27.     serv_adr.sin_port=htons(atoi(argv[1]));
28.
29.     if(bind(serv_sock, (struct sockaddr*) &serv_adr, sizeof(serv_adr))==-1)
30.         error_handling("bind() error");
31.     if(listen(serv_sock, 5)==-1)
32.         error_handling("listen() error");
33.
34.     epfd=epoll_create(EPOLL_SIZE);
35.     ep_events=malloc(sizeof(struct epoll_event)*EPOLL_SIZE);
36.
37.     event.events=EPOLLIN;
38.     event.data.fd=serv_sock;
39.     epoll_ctl(epfd, EPOLL_CTL_ADD, serv_sock, &event);
40.
41.     while(1)
42.     {
43.         event_cnt=epoll_wait(epfd, ep_events, EPOLL_SIZE, -1);
44.         if(event_cnt==-1)
45.         {
46.             puts("epoll_wait() error");
47.             break;
48.         }
49.
50.         puts("return epoll_wait");
51.         for(i=0; i<event_cnt; i++)
52.         {
53.             if(ep_events[i].data.fd==serv_sock)
54.             {
55.                 adr_sz=sizeof(clnt_adr);
56.                 clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_adr, &adr_sz);
57.                 event.events=EPOLLIN;
58.                 event.data.fd=clnt_sock;
59.                 epoll_ctl(epfd, EPOLL_CTL_ADD, clnt_sock, &event);
60.                 printf("connected client: %d \n", clnt_sock);
61.             }
62.             else
63.             {
64.                     str_len=read(ep_events[i].data.fd, buf, BUF_SIZE);
65.                     if(str_len==0)  // close request!
66.                     {
67.                         epoll_ctl(epfd, EPOLL_CTL_DEL, ep_events[i].data.fd, NULL);
68.                         close(ep_events[i].data.fd);
69.                         printf("closed client: %d \n", ep_events[i].data.fd);
```

```
70.                     }
71.                     else
72.                     {
73.                         write(ep_events[i].data.fd, buf, str_len);    // echo!
74.                     }
75.             }
76.         }
77.     }
78.     close(serv_sock);
79.     close(epfd);
80.     return 0;
81. }
82.
83. void error_handling(char *buf)
84. {
85.     //与示例echo_epollserv.c的error_handling函数一致。
86. }
```

上述示例与之前的echo_epollserv.c之间的差异如下。

- 将调用read函数时使用的缓冲大小缩减为4个字节（第2行）
- 插入验证epoll_wait函数调用次数的语句（第50行）

减少缓冲大小是为了阻止服务器端一次性读取接收的数据。换言之，调用read函数后，输入缓冲中仍有数据需要读取。而且会因此注册新的事件并从epoll_wait函数返回时将循环输出“return epoll_wait”字符串。前提是条件触发的工作方式与我的描述一致。接下来观察运行结果。该程序同样可以结合第4章的echo_client.c运行。

❖ 运行结果：echo_EPLTserv.c

```
root@my_linux:/tcpip# gcc echo_EPLTserv.c -o serv
root@my_linux:/tcpip# ./serv 9190
return epoll_wait
connected client: 5
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
connected client: 6
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
closed client: 5
return epoll_wait
closed client: 6
```

❖ 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): It's my life
Message from server: It's my life
Input message(Q to quit): Q
```

❖ 运行结果：echo_client.c Two

```
root@my_linux:/tcpip# gcc sep_clnt.c -o clnt
root@my_linux:/tcpip# ./client 127.0.0.1 9190
Connected...........
Input message(Q to quit): It's your life
Message from server: It's your life
Input message(Q to quit): Q
```


从运行结果中可以看出，每当收到客户端数据时，都会注册该事件，并因此多次调用epoll_wait函数。下面将上述示例改成边缘触发方式，需要做一些额外的工作。但我希望通过最小的改动验证边缘触发模型的事件注册方式。将上述示例的第57行改成如下形式运行服务器端和客户端（不会单独提供这方面的源代码，需要各位自行更改）：

```
event.events = EPOLLIN|EPOLLET;
```

更改后可以验证如下事实：

> “从客户端接收数据时，仅输出1次‘return epoll_wait’字符串，这意味着仅注册1次事件。”

虽然可以验证上述事实，但客户端运行时将发生错误。大家是否遇到了这种问题？能否自行分析原因？虽然目前不必对此感到困惑，但如果理解了边缘触发的特性，应该可以分析出错误原因。

select 模型是条件触发还是边缘触发？

select 模型是以条件触发的方式工作的，输入缓冲中如果还剩有数据，肯定会注册事件。各位若感兴趣，可以自行编写示例验证 select 模型的工作方式。

边缘触发的服务器端实现中必知的两点

下面讲解边缘触发服务器端的实现方法。在此之前，我希望说明如下2点，这些是实现边缘触发的必知内容。

- 通过errno变量验证错误原因。
- 为了完成非阻塞（Non-blocking）I/O，更改套接字特性。

Linux的套接字相关函数一般通过返回-1通知发生了错误。虽然知道发生了错误，但仅凭这些内容无法得知产生错误的原因。因此，为了在发生错误时提供额外的信息，Linux声明了如下全局变量：

```
int errno;
```

为了访问该变量，需要引入error.h头文件，因为此头文件中有上述变量的extern声明。另外，每种函数发生错误时，保存到errno变量中的值都不同，没必要记住所有可能的值。学习每种函数的过程中逐一掌握，并能在必要时参考即可。本节只介绍如下类型的错误：

"read函数发现输入缓冲中没有数据可读时返回-1，同时在errno中保存EAGAIN常量。"

稍后通过示例给出errno的使用方法。下面讲解将套接字改为非阻塞方式的方法。Linux提供更改或读取文件属性的如下方法（曾在第13章使用过）。

```
#include <fcntl.h>

int fcntl(int filedes, int cmd, . . . );
```

➔ 成功时返回 cmd 参数相关值，失败时返回-1。

- filedes　属性更改目标的文件描述符。
- cmd　表示函数调用的目的。

从上述声明中可以看到，fcntl具有可变参数的形式。如果向第二个参数传递F_GETFL，可以获得第一个参数所指的文件描述符属性（int型）。反之，如果传递F_SETFL，可以更改文件描述符属性。若希望将文件（套接字）改为非阻塞模式，需要如下2条语句。

```
int flag = fcntl(fd, F_GETFL, 0);
fcntl(fd, F_SETFL, flag|O_NONBLOCK);
```

通过第一条语句获取之前设置的属性信息，通过第二条语句在此基础上添加非阻塞O_NONBLOCK标志。调用read & write函数时，无论是否存在数据，都会形成非阻塞文件（套接字）。fcntl函数的适用范围很广，各位既可以在学习系统编程时一次性总结所有适用情况，也可以每次需要时逐一掌握。

实现边缘触发的回声服务器端

之所以介绍读取错误原因的方法和非阻塞模式的套接字创建方法，原因在于二者都与边缘触

发的服务器端实现有密切联系。首先说明为何需要通过errno确认错误原因。

“边缘触发方式中，接收数据时仅注册1次该事件。”

就因为这种特点，一旦发生输入相关事件，就应该读取输入缓冲中的全部数据。因此需要验证输入缓冲是否为空。

“read函数返回-1，变量errno中的值为EAGAIN时，说明没有数据可读。”

既然如此，为何还需要将套接字变成非阻塞模式？边缘触发方式下，以阻塞方式工作的read & write函数有可能引起服务器端的长时间停顿。因此，边缘触发方式中一定要采用非阻塞read & write函数。接下来给出以边缘触发方式工作的回声服务器端示例。

❖ echo_EPETserv.c

```
1.  #include <"添加fcntl.h、errno.h时，与示例echo_epollserv.c的头文件声明一致。">
2.  #include <fcntl.h>
3.  #include <errno.h>
4.  #define BUF_SIZE 4
5.  #define EPOLL_SIZE 50
6.  void setnonblockingmode(int fd);
7.  void error_handling(char *buf);
8.
9.  int main(int argc, char *argv[])
10. {
11.     int serv_sock, clnt_sock;
12.     struct sockaddr_in serv_adr, clnt_adr;
13.     socklen_t adr_sz;
14.     int str_len, i;
15.     char buf[BUF_SIZE];
16.
17.     struct epoll_event *ep_events;
18.     struct epoll_event event;
19.     int epfd, event_cnt;
20.     if(argc!=2) {
21.         printf("Usage : %s <port>\n", argv[0]);
22.         exit(1);
23.     }
24.
25.     serv_sock=socket(PF_INET, SOCK_STREAM, 0);
26.     memset(&serv_adr, 0, sizeof(serv_adr));
27.     serv_adr.sin_family=AF_INET;
28.     serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
29.     serv_adr.sin_port=htons(atoi(argv[1]));
30.     if(bind(serv_sock, (struct sockaddr*) &serv_adr, sizeof(serv_adr))==-1)
31.         error_handling("bind() error");
32.     if(listen(serv_sock, 5)==-1)
33.         error_handling("listen() error");
34.
35.     epfd=epoll_create(EPOLL_SIZE);
```

```
36.     ep_events=malloc(sizeof(struct epoll_event)*EPOLL_SIZE);
37.
38.     setnonblockingmode(serv_sock);
39.     event.events=EPOLLIN;
40.     event.data.fd=serv_sock;
41.     epoll_ctl(epfd, EPOLL_CTL_ADD, serv_sock, &event);
42.
43.     while(1)
44.     {
45.         event_cnt=epoll_wait(epfd, ep_events, EPOLL_SIZE, -1);
46.         if(event_cnt==-1)
47.         {
48.             puts("epoll_wait() error");
49.             break;
50.         }
51.
52.         puts("return epoll_wait");
53.         for(i=0; i<event_cnt; i++)
54.         {
55.             if(ep_events[i].data.fd==serv_sock)
56.             {
57.                 adr_sz=sizeof(clnt_adr);
58.                 clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_adr, &adr_sz);
59.                 setnonblockingmode(clnt_sock);
60.                 event.events=EPOLLIN|EPOLLET;
61.                 event.data.fd=clnt_sock;
62.                 epoll_ctl(epfd, EPOLL_CTL_ADD, clnt_sock, &event);
63.                 printf("connected client: %d \n", clnt_sock);
64.             }
65.             else
66.             {
67.                 while(1)
68.                 {
69.                     str_len=read(ep_events[i].data.fd, buf, BUF_SIZE);
70.                     if(str_len==0) {     // close request!
71.                         epoll_ctl(epfd, EPOLL_CTL_DEL, ep_events[i].data.fd, NULL);
72.                         close(ep_events[i].data.fd);
73.                         printf("closed client: %d \n", ep_events[i].data.fd);
74.                         break;
75.                     }
76.                     else if(str_len<0) {
77.                         if(errno==EAGAIN)
78.                           break;
79.                     }
80.                     else {
81.                         write(ep_events[i].data.fd, buf, str_len);   // echo!
82.                     }
83.                 }
84.             }
85.         }
86.     }
87.     close(serv_sock);   close(epfd);
88.     return 0;
89. }
```

```
90.
91. void setnonblockingmode(int fd)
92. {
93.     int flag=fcntl(fd, F_GETFL, 0);
94.     fcntl(fd, F_SETFL, flag|O_NONBLOCK);
95. }
96. void error_handling(char *buf)
97. {
98.     //与示例echo_epollserv.c的error_handing函数一致。
99. }
```

- 第4行：为了验证边缘触发的工作方式，将缓冲设置为4字节。
- 第52行：为观察事件发生数而添加的输出字符串的语句。
- 第59、60行：第59行将accept函数创建的套接字改为非阻塞模式。第60行向EPOLLIN添加EPOLLET标志，将套接字事件注册方式改为边缘触发。
- 第67、69行：之前的条件触发回声服务器端中没有该while循环。边缘触发方式中，发生事件时需要读取输入缓冲中的所有数据，因此需要循环调用read函数，如第69行所示。
- 第76行：read函数返回-1且errno值为EAGAIN时，意味着读取了输入缓冲中的全部数据，因此需要通过break语句跳出第67行的循环。

❖ 运行结果：echo_EPETserv.c

```
root@my_linux:/tcpip# gcc echo_EPETserv.c -o serv
root@my_linux:/tcpip# ./serv 9190
return epoll_wait
connected client: 5
return epoll_wait
return epoll_wait
return epoll_wait
closed client: 5
```

❖ 运行结果：echo_client.c

```
root@my_linux:/tcpip# gcc echo_client.c -o clnt
root@my_linux:/tcpip# ./clnt 127.0.0.1 9190
Connected...........
Input message(Q to quit): I like computer programming
Message from server: I like computer programming
Input message(Q to quit): Do you like computer programming?
Message from server: Do you like computer programming?
Input message(Q to quit): Good bye
Message from server: Good bye
```

```
Input message(Q to quit): Q
```

上述运行结果中需要注意的是，客户端发送消息次数和服务器端epoll_wait函数调用次数。客户端从请求连接到断开连接共发送5次数据，服务器端也相应产生5个事件。

条件触发和边缘触发孰优孰劣

我们从理论和代码的角度充分理解了条件触发和边缘触发，但仅凭这些还无法理解边缘触发相对于条件触发的优点。边缘触发方式下可以做到如下这点：

> “可以分离接收数据和处理数据的时间点！”

虽然比较简单，但非常准确有力地说明了边缘触发的优点。关于这句话的含义，大家以后开发不同类型的程序时会有更深入的理解。现阶段给出如下情景帮助大家理解，如图17-1所示。

图17-1　理解边缘触发

图17-1的运行流程如下。

- 服务器端分别从客户端A、B、C接收数据。
- 服务器端按照A、B、C的顺序重新组合收到的数据。
- 组合的数据将发送给任意主机。

为了完成该过程，若能按如下流程运行程序，服务器端的实现并不难。

- 客户端按照A、B、C的顺序连接服务器端，并依序向服务器端发送数据。
- 需要接收数据的客户端应在客户端A、B、C之前连接到服务器端并等待。

但现实中可能频繁出现如下这些情况，换言之，如下情况更符合实际。

- 客户端C和B正向服务器端发送数据，但A尚未连接到服务器端。

- 客户端A、B、C乱序发送数据。
- 服务器端已收到数据，但要接收数据的目标客户端还未连接到服务器端。

因此，即使输入缓冲收到数据（注册相应事件），服务器端也能决定读取和处理这些数据的时间点，这样就给服务器端的实现带来巨大的灵活性。

“条件触发中无法区分数据接收和处理吗？”

并非不可能。但在输入缓冲收到数据的情况下，如果不读取（延迟处理），则每次调用epoll_wait函数时都会产生相应事件。而且事件数也会累加，服务器端能承受吗？这在现实中是不可能的（本身并不合理，因此是根本不想做的事）。

条件触发和边缘触发的区别主要应该从服务器端实现模型的角度谈论，因此希望各位不要提下面这种问题。如果理解了之前的讲解，应该有更好的提问。

“边缘触发是否更快？能快多少呢？”

从实现模型的角度看，边缘触发更有可能带来高性能，但不能简单地认为“只要使用边缘触发就一定能提高速度”。

17.3 习题

(1) 利用select函数实现服务器端时，代码层面存在的2个缺点是？

(2) 无论是select方式还是epoll方式，都需要将监视对象文件描述符信息通过函数调用传递给操作系统。请解释传递该信息的原因。

(3) select方式和epoll方式的最大差异在于监视对象文件描述符传递给操作系统的方式。请说明具体的差异，并解释为何存在这种差异。

(4) 虽然epoll是select的改进方案，但select也有自己的优点。在何种情况下使用select方式更合理？

(5) epoll以条件触发或边缘触发方式工作。二者有何区别？从输入缓冲的角度说明这2种方式通知事件的时间点差异。

(6) 采用边缘触发时可以分离数据的接收和处理时间点。说明其原因及优点。

(7) 实现聊天服务器端，使其可以在连接到服务器端的所有客户端之间交换消息。按照条件触发方式和边缘触发方式分别实现epoll服务器端（聊天服务器端的实现中，这2种方式不会产生太大差异）。当然，为了正常运行服务器端，需要聊天客户端，我们直接使用第18章的chat_clnt.c（编译方法请参考第18章）。虽然尚未学习第18章，但使用其中一些示例并非难事。如果各位觉得困难，可以学习第18章后再解答。

第18章 多线程服务器端的实现

本来，线程在 Windows 中的应用比在 Linux 平台中的应用更广泛。但 Web 服务的发展迫使 UNIX 系列的操作系统开始重视线程。由于 Web 服务器端协议本身具有的特点，经常需要同时向多个客户端提供服务。因此，人们逐渐舍弃进程，转而开始利用更高效的线程实现 Web 服务器端。

18.1 理解线程的概念

第19章将介绍Windows线程，而本章给出的是关于线程的通用说明，掌握了本章内容才能学好Windows线程。

引入线程的背景

第10章介绍了多进程服务器端的实现方法。多进程模型与select或epoll相比的确有自身的优点，但同时也有问题。如前所述，创建进程（复制）的工作本身会给操作系统带来相当沉重的负担。而且，每个进程具有独立的内存空间，所以进程间通信的实现难度也会随之提高（参考第11章）。换言之，多进程模型的缺点可概括如下。

- 创建进程的过程会带来一定的开销。
- 为了完成进程间数据交换，需要特殊的IPC技术。

但相比于下面的缺点，上述2个缺点不算什么。

> “每秒少则数十次、多则数千次的‘上下文切换’（Context Switching）是创建进程时最大的开销。”

只有1个CPU（准确地说是CPU的运算设备CORE）的系统中不是也可以同时运行多个进程吗？这是因为系统将CPU时间分成多个微小的块后分配给了多个进程。为了分时使用CPU，需要“上下文切换”过程。下面了解一下“上下文切换”的概念。运行程序前需要将相应进程信息读

入内存，如果运行进程A后需要紧接着运行进程B，就应该将进程A相关信息移出内存，并读入进程B相关信息。这就是上下文切换。但此时进程A的数据将被移动到硬盘，所以上下文切换需要很长时间。即使通过优化加快速度，也会存在一定的局限。

上下文切换

通过学习计算机结构和操作系统相关知识，可以了解到上下文切换中具体的工作过程。但我为了讲述网络编程，只介绍了基础概念。实际上该过程应该通过 CPU 内部的寄存器来解释。

为了保持多进程的优点，同时在一定程度上克服其缺点，人们引入了线程（Thread）。这是为了将进程的各种劣势降至最低限度（不是直接消除）而设计的一种“轻量级进程”。线程相比于进程具有如下优点。

- 线程的创建和上下文切换比进程的创建和上下文切换更快。
- 线程间交换数据时无需特殊技术。

各位会逐渐体会到这些优点，可以通过接下来的说明和线程相关代码进行准确理解。

线程和进程的差异

线程是为了解决如下困惑登场的:

“嘿！为了得到多条代码执行流而复制整个内存区域的负担太重了！”

每个进程的内存空间都由保存全局变量的“数据区”、向malloc等函数的动态分配提供空间的堆（Heap）、函数运行时使用的栈（Stack）构成。每个进程都拥有这种独立空间，多个进程的内存结构如图18-1所示。

图18-1 进程间独立的内存

但如果以获得多个代码执行流为主要目的，则不应该像图18-1那样完全分离内存结构，而只需分离栈区域。通过这种方式可以获得如下优势。

- 上下文切换时不需要切换数据区和堆。
- 可以利用数据区和堆交换数据。

实际上这就是线程。线程为了保持多条代码执行流而隔开了栈区域，因此具有如图18-2所示的内存结构。

图18-2　线程的内存结构

如图18-2所示，多个线程将共享数据区和堆。为了保持这种结构，线程将在进程内创建并运行。也就是说，进程和线程可以定义为如下形式。

- 进程：在操作系统构成单独执行流的单位。
- 线程：在进程构成单独执行流的单位。

如果说进程在操作系统内部生成多个执行流，那么线程就在同一进程内部创建多条执行流。因此，操作系统、进程、线程之间的关系可以通过图18-3表示。

图18-3　操作系统、进程、线程之间的关系

以上就是线程的理论说明。没有实际编程就很难理解好线程，希望各位通过学习线程相关代码理解全部内容。

18.2 线程创建及运行

POSIX是Portable Operating System Interface for Computer Environment（适用于计算机环境的可移植操作系统接口）的简写，是为了提高UNIX系列操作系统间的移植性而制定的API规范。下面要介绍的线程创建方法也是以POSIX标准为依据的。因此，它不仅适用于Linux，也适用于大部分UNIX系列的操作系统。

线程的创建和执行流程

线程具有单独的执行流，因此需要单独定义线程的main函数，还需要请求操作系统在单独的执行流中执行该函数，完成该功能的函数如下。

```
#include <pthread.h>

int pthread_create(
    pthread_t * restrict thread, const pthread_attr_t * restrict attr,
    void * (* start_routine)(void *), void * restrict arg
);
    ➔ 成功时返回 0，失败时返回其他值。
```

- thread　保存新创建线程ID的变量地址值。线程与进程相同，也需要用于区分不同线程的ID。
- attr　用于传递线程属性的参数，传递NULL时，创建默认属性的线程。
- start_routine　相当于线程main函数的、在单独执行流中执行的函数地址值（函数指针）。
- arg　通过第三个参数传递调用函数时包含传递参数信息的变量地址值。

要想理解好上述函数的参数，需要熟练掌握restrict关键字和函数指针相关语法。但如果只关注使用方法（当然以后要掌握restrict和函数指针），那么该函数的使用比想象中要简单。下面通过简单示例了解该函数的功能。

❖ thread1.c

```
1.  #include <stdio.h>
2.  #include <pthread.h>
3.  void* thread_main(void *arg);
4.
5.  int main(int argc, char *argv[])
6.  {
```

```
7.      pthread_t t_id;
8.      int thread_param=5;
9.
10.     if(pthread_create(&t_id, NULL, thread_main, (void*)&thread_param)!=0)
11.     {
12.         puts("pthread_create() error");
13.         return -1;
14.     };
15.     sleep(10); puts("end of main");
16.     return 0;
17. }
18.
19. void* thread_main(void *arg)
20. {
21.     int i;
22.     int cnt=*((int*)arg);
23.     for(i=0; i<cnt; i++)
24.     {
25.         sleep(1); puts("running thread");
26.     }
27.     return NULL;
28. }
```

- 第10行：请求创建一个线程，从thread_main函数调用开始，在单独的执行流中运行。同时在调用thread_main函数时向其传递thread_param变量的地址值。
- 第15行：调用sleep函数使main函数停顿10秒，这是为了延迟进程的终止时间。执行第16行的return语句后终止进程，同时终止内部创建的线程。因此，为保证线程的正常执行而添加这条语句。
- 第19、22行：传入arg参数的是第10行pthread_create函数的第四个参数。

❖ 运行结果：thread1.c

```
root@my_linux:/tcpip# gcc thread1.c -o tr1 -lpthread
root@my_linux:/tcpip# ./tr1
running thread
running thread
running thread
running thread
running thread
end of main
```

从上述运行结果中可以看到，线程相关代码在编译时需要添加–lpthread选项声明需要连接线程库，只有这样才能调用头文件pthread.h中声明的函数。上述程序的执行流程如图18-4所示。

图18-4 示例thread1.c的执行流程

图18-4中的虚线代表执行流称，向下的箭头指的是执行流，横向箭头是函数调用。这些都是简单的符号，可以结合示例理解。接下来将上述示例的第15行sleep函数的调用语句改成如下形式：


```
sleep(2);
```

各位运行后可以看到，此时不会像代码中写的那样输出5次“running thread”字符串。因为main函数返回后整个进程将被销毁，如图18-5所示。

图18-5 终止进程和线程

正因如此，我们在之前的示例中通过调用sleep函数向线程提供了充足的执行时间。

“那线程相关程序中必须适当调用sleep函数！”

并非如此！通过调用sleep函数控制线程的执行相当于预测程序的执行流程，但实际上这是不可能完成的事情。而且稍有不慎，很可能干扰程序的正常执行流。例如，怎么可能在上述示例中准确预测thread_main函数的运行时间，并让main函数恰好等待这么长时间呢？因此，我们不用sleep函数，而是通常利用下面的函数控制线程的执行流。通过下列函数可以更有效地解决现讨论的问题，还可同时了解线程ID的用法。

```
# include <pthread.h>
int pthread_join(pthread_t thread, void ** status);
```

➔ 成功时返回 0，失败时返回其他值。

- thread　该参数值ID的线程终止后才会从该函数返回。
- status　保存线程的main函数返回值的指针变量地址值。

简言之，调用该函数的进程（或线程）将进入等待状态，直到第一个参数为ID的线程终止为止。而且可以得到线程的main函数返回值，所以该函数比较有用。下面通过示例了解该函数的功能。

❖ thread2.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <pthread.h>
5.  void* thread_main(void *arg);
6.
7.  int main(int argc, char *argv[])
8.  {
9.      pthread_t t_id;
10.     int thread_param=5;
11.     void * thr_ret;
12.
13.     if(pthread_create(&t_id, NULL, thread_main, (void*)&thread_param)!=0)
14.     {
15.         puts("pthread_create() error");
16.         return -1;
17.     };
18.
19.     if(pthread_join(t_id, &thr_ret)!=0)
20.     {
21.         puts("pthread_join() error");
22.         return -1;
23.     };
24.
25.     printf("Thread return message: %s \n", (char*)thr_ret);
26.     free(thr_ret);
27.     return 0;
28. }
29.
30. void* thread_main(void *arg)
31. {
32.     int i;
33.     int cnt=*((int*)arg);
34.     char * msg=(char *)malloc(sizeof(char)*50);
35.     strcpy(msg, "Hello, I'am thread~ \n");
36.
37.     for(i=0; i<cnt; i++)
```

```
38.     {
39.         sleep(1); puts("running thread");
40.     }
41.     return (void*)msg;
42. }
```

- 第19行：main函数中，针对第13行创建的线程调用pthread_join函数。因此，main函数将等待ID保存在t_id变量中的线程终止。
- 第11、19、41行：希望各位通过这3条语句掌握获取线程的返回值的方法。简言之，第41行返回的值将保存到第19行第二个参数thr_ret。需要注意的是，该返回值是thread_main函数内部动态分配的内存空间地址值。

❖ 运行结果：thread2.c

```
root@my_linux:/tcpip# gcc thread2.c -o tr2 -lpthread
root@my_linux:/tcpip# ./tr2
running thread
running thread
running thread
running thread
running thread
Thread return message: Hello, I'am thread~
```

最后，为了让大家更好地理解该示例，给出其执行流程图，如图18-6所示。请注意观察程序暂停后从线程终止时（线程main函数返回时）重新执行的部分。

图18-6 调用pthread_join函数

可在临界区内调用的函数

之前的示例中只创建了1个线程，接下来的示例将开始创建多个线程。当然，无论创建多少

线程，其创建方法没有区别。但关于线程的运行需要考虑“多个线程同时调用函数时（执行时）可能产生问题”。这类函数内部存在临界区（Critical Section），也就是说，多个线程同时执行这部分代码时，可能引起问题。临界区中至少存在1条这类代码。

稍后将讨论哪些代码可能成为临界区，多个线程同时执行临界区代码时会产生哪些问题等内容。现阶段只需理解临界区的概念即可。根据临界区是否引起问题，函数可分为以下2类。

- 线程安全函数（Thread-safe function）
- 非线程安全函数（Thread-unsafe function）

线程安全函数被多个线程同时调用时也不会引发问题。反之，非线程安全函数被同时调用时会引发问题。但这并非关于有无临界区的讨论，线程安全的函数中同样可能存在临界区。只是在线程安全函数中，同时被多个线程调用时可通过一些措施避免问题。

幸运的是，大多数标准函数都是线程安全的函数。更幸运的是，我们不用自己区分线程安全的函数和非线程安全的函数（在Windows程序中同样如此）。因为这些平台在定义非线程安全函数的同时，提供了具有相同功能的线程安全的函数。比如，第8章介绍过的如下函数就不是线程安全的函数：

```
struct hostent * gethostbyname(const char * hostname);
```

同时提供线程安全的同一功能的函数。

```
struct hostent * gethostbyname_r(
    const char * name, struct hostent * result, char * buffer, intbuflen,
        int * h_errnop);
```

线程安全函数的名称后缀通常为_r（这与Windows平台不同）。既然如此，多个线程同时访问的代码块中应该调用gethostbyname_r，而不是gethostbyname？当然！但这种方法会给程序员带来沉重的负担。幸好可以通过如下方法自动将gethostbyname函数调用改为gethostbyname_r函数调用！

“声明头文件前定义_REENTRANT宏。”

gethostbyname函数和gethostbyname_r函数的函数名和参数声明都不同，因此，这种宏声明方式拥有巨大的吸引力。另外，无需为了上述宏定义特意添加#define语句，可以在编译时通过添加-D_REENTRANT选项定义宏。

```
root@my_linux:/tcpip# gcc -D_REENTRANT mythread.c -o mthread -lpthread
```

下面编译线程相关代码时均默认添加-D_REENTRANT选项。

工作（Worker）线程模型

之前示例的目的主要是介绍线程概念和创建线程的方法，因此从未涉及1个示例中创建多个

线程的情况。下面给出此类示例。

将要介绍的示例将计算1到10的和，但并不是在main函数中进行累加运算，而是创建2个线程，其中一个线程计算1到5的和，另一个线程计算6到10的和，main函数只负责输出运算结果。这种方式的编程模型称为“工作线程（Worker thread）模型”。计算1到5之和的线程与计算6到10之和的线程将成为main线程管理的工作（Worker）。最后，给出示例代码前先给出程序执行流程图，如图18-7所示。

图18-7 示例thread3.c的执行流程

之前也介绍过类似的图，相信各位很容易看懂图18-7描述的内容（只是单纯说明图，并未使用特殊的表示方法）。另外，线程相关代码的执行流程理解起来相对复杂一些，有必要习惯于这类流程图。

❖ thread3.c

```
1.  #include <stdio.h>
2.  #include <pthread.h>
3.  void * thread_summation(void * arg);
4.  int sum=0;
5.
6.  int main(int argc, char *argv[])
7.  {
8.      pthread_t id_t1, id_t2;
9.      int range1[]={1, 5};
10.     int range2[]={6, 10};
```

```
11.
12.     pthread_create(&id_t1, NULL, thread_summation, (void *)range1);
13.     pthread_create(&id_t2, NULL, thread_summation, (void *)range2);
14.
15.     pthread_join(id_t1, NULL);
16.     pthread_join(id_t2, NULL);
17.     printf("result: %d \n", sum);
18.     return 0;
19. }
20.
21. void * thread_summation(void * arg)
22. {
23.     int start=((int*)arg)[0];
24.     int end=((int*)arg)[1];
25.
26.     while(start<=end)
27.     {
28.         sum+=start;
29.         start++;
30.     }
31.     return NULL;
32. }
```

之前讲过线程调用函数的参数和返回值类型，因此不难理解上述示例中创建线程并执行的部分。但需要注意：

“2个线程直接访问全局变量sum!”

通过上述示例的第28行可以得出这种结论。从代码的角度看似乎理所应当，但之所以可行完全是因为2个线程共享保存全局变量的数据区。

❖ 运行结果：thread3.c

```
root@my_linux:/tcpip# gcc thread3.c -D_REENTRANT -o tr3 -lpthread
root@my_linux:/tcpip# ./tr3
result: 55
```

运行结果是55，虽然正确，但示例本身存在问题。此处存在临界区相关问题，因此再介绍另一示例。该示例与上述示例相似，只是增加了发生临界区相关错误的可能性，即使在高配置系统环境下也容易验证产生的错误。

❖ thread4.c

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.  #include <stdlib.h>
4.  #include <pthread.h>
5.  #define NUM_THREAD 100
```

```
6.
7.  void * thread_inc(void * arg);
8.  void * thread_des(void * arg);
9.  long long num=0;    // long long类型是64位整数型
10.
11. int main(int argc, char *argv[])
12. {
13.     pthread_t thread_id[NUM_THREAD];
14.     int i;
15.
16.     printf("sizeof long long: %d \n", sizeof(long long));    // 查看long long的大小
17.     for(i=0; i<NUM_THREAD; i++)
18.     {
19.         if(i%2)
20.             pthread_create(&(thread_id[i]), NULL, thread_inc, NULL);
21.         else
22.             pthread_create(&(thread_id[i]), NULL, thread_des, NULL);
23.     }
24.
25.     for(i=0; i<NUM_THREAD; i++)
26.         pthread_join(thread_id[i], NULL);
27.
28.     printf("result: %lld \n", num);
29.     return 0;
30. }
31.
32. void * thread_inc(void * arg)
33. {
34.     int i;
35.     for(i=0; i<50000000; i++)
36.         num+=1;
37.     return NULL;
38. }
39. void * thread_des(void * arg)
40. {
41.     int i;
42.     for(i=0; i<50000000; i++)
43.         num-=1;
44.     return NULL;
45p.}
```

上述示例中共创建了100个线程，其中一半执行thread_inc函数中的代码，另一半则执行thread_des函数中的代码。全局变量num经过增减过程后应存有0，通过运行结果观察是否真能得到。

❖ 运行结果：thread4.c

```
root@my_linux:/tcpip# gcc thread4.c -D_REENTRANT -o th4 -lpthread
root@my_linux:/tcpip# ./th4
sizeof long long: 8
result: 4284144869
```

```
root@my_linux:/tcpip# ./th4
sizeof long long: 8
result: 8577052432
root@my_linux:/tcpip# ./th4
sizeof long long: 8
result: 21446758095
```

运行结果并不是0！而且每次运行的结果均不同。虽然其原因尚不得而知，但可以肯定的是，这对于线程的应用是个大问题。

18.3　线程存在的问题和临界区

我们还不知道示例thread4.c中产生问题的原因，下面分析该问题并给出解决方案。

多个线程访问同一变量是问题

示例thread4.c的问题如下：

> “2个线程正在同时访问全局变量num。”

此处的“访问”是指值的更改。产生问题的原因可能还有很多，因此需要准确理解。虽然示例中访问的对象是全局变量，但这并非全局变量引发的问题。任何内存空间——只要被同时访问——都可能发生问题。

> “不是说线程会分时使用CPU吗？那应该不会出现同时访问变量的情况啊。”

当然，此处的“同时访问”与各位所想的有一定区别。下面通过示例解释“同时访问”的含义，并说明为何会引起问题。假设2个线程要执行将变量值逐次加1的工作，如图18-8所示。

图18-8　等待中的2个线程

图18-8中描述的是2个线程准备将变量num的值加1的情况。在此状态下，线程1将变量num的

值增加到100后，线程2再访问num时，变量num中将按照我们的预想保存101。图18-9是线程1将变量num完全增加后的情形。

图18-9 线程的加法运算1-1

图18-9中需要注意值的增加方式，值的增加需要CPU运算完成，变量num中的值不会自动增加。线程1首先读该变量的值并将其传递到CPU，获得加1之后的结果100，最后再把结构写回变量num，这样num中就保存100。接下来给出线程2的执行过程，如图18-10所示。

图18-10 线程的加法运算 1-2

变量num中将保存101，但这是最理想的情况。线程1完全增加num值之前，线程2完全有可能通过切换得到CPU资源。下面从头再来。图18-11描绘的是线程1读取变量num的值并完成加1运算时的情况，只是加1后的结果尚未写入变量num。

图18-11 线程的加法运算2-1

接下来就要将100保存到变量num中，但执行该操作前，执行流程跳转到了线程2。幸运的是（是否真正幸运稍后再论），线程2完成了加1运算，并将加1之后的结果写入变量num，如图18-12所示。

图18-12　线程的加法运算 2-2

从图18-12中可以看到，变量num的值尚未被线程1加到100，因此线程2读到的变量num的值为99，结果是线程2将num值改成100。还剩下线程1将运算后的值写入变量num的操作。接下来给出该过程，如图18-13所示。

图18-13　线程的加法运算 2-3

很可惜，此时线程1将自己的运算结果100再次写入变量num，结果变量num变成100。虽然线程1和线程2各做了1次加1运算，却得到了意想不到的结果。因此，线程访问变量num时应该阻止其他线程访问，直到线程1完成运算。这就是同步（Synchronization）。相信各位也意识到了多线程编程中“同步”的必要性，且能够理解thread4.c的运行结果。

临界区位置

划分临界区并不难。既然临界区定义为如下这种形式，那就在示例thread4.c中寻找。

“函数内同时运行多个线程时引起问题的多条语句构成的代码块。”

全局变量num是否应该视为临界区？不是！因为它不是引起问题的语句。该变量并非同时运

行的语句，只是代表内存区域的声明而已。临界区通常位于由线程运行的函数内部。下面观察示例thread4.c中的2个main函数。

```
void * thread_inc(void * arg)
{
    int i;
    for(i = 0; i < 50000000; i++)
        num += 1;  // 临界区
    return NULL;
}

void * thread_des(void * arg)
{
    int i;
    for(i = 0; i < 50000000; i++)
        num -= 1;   // 临界区
    return NULL;
}
```

由代码注释可知，临界区并非num本身，而是访问num的2条语句。这2条语句可能由多个线程同时运行，也是引起问题的直接原因。产生的问题可以整理为如下3种情况。

- 2个线程同时执行thread_inc函数。
- 2个线程同时执行thread_des函数。
- 2个线程分别执行thread_inc函数和thread_des函数。

需要关注最后一点，它意味着如下情况下也会引发问题：

> “线程1执行thread_inc函数的num+=1语句的同时，线程2执行thread_des函数的num-=1语句。”

也就是说，2条不同语句由不同线程同时执行时，也有可能构成临界区。前提是这2条语句访问同一内存空间。

18.4 线程同步

前面探讨了线程中存在的问题，接下来就要讨论解决方法——线程同步。

同步的两面性

线程同步用于解决线程访问顺序引发的问题。需要同步的情况可以从如下两方面考虑。

- 同时访问同一内存空间时发生的情况。
- 需要指定访问同一内存空间的线程执行顺序的情况。

之前已解释过前一种情况，因此重点讨论第二种情况。这是“控制（Control）线程执行顺序”的相关内容。假设有A、B两个线程，线程A负责向指定内存空间写入（保存）数据，线程B负责取走该数据。这种情况下，线程A首先应该访问约定的内存空间并保存数据。万一线程B先访问并取走数据，将导致错误结果。像这种需要控制执行顺序的情况也需要使用同步技术。

稍后将介绍“互斥量”（Mutex）和“信号量”（Semaphore）这2种同步技术。二者概念上十分接近，只要理解了互斥量就很容易掌握信号量。而且大部分同步技术的原理都大同小异，因此，只要掌握了本章介绍的同步技术，就很容易掌握并运用Windows平台下的同步技术。

互斥量

互斥量是“Mutual Exclusion”的简写，表示不允许多个线程同时访问。互斥量主要用于解决线程同步访问的问题。为了理解好互斥量，请观察如下对话过程。

- 东秀：“请问里面有人吗？”
- 英秀：“是的，有人。”

- 东秀：“您好！”
- 英秀：“请稍等！”

相信各位也猜到了上述对话发生的场景。现实世界中的临界区就是洗手间。洗手间无法同时容纳多人（比作线程），因此可以将临界区比喻为洗手间。而且这里发生的所有事情几乎可以全部套用到临界区同步过程。洗手间使用规则如下。

- 为了保护个人隐私，进洗手间时锁上门，出来时再打开。
- 如果有人使用洗手间，其他人需要在外面等待。
- 等待的人数可能很多，这些人需排队进入洗手间。

这就是洗手间的使用规则。同样，线程中为了保护临界区也需要套用上述规则。洗手间中存在，但之前的线程示例中缺少的是什么呢？就是锁机制。线程同步中同样需要锁，就像洗手间示例中使用的那样。互斥量就是一把优秀的锁，接下来介绍互斥量的创建及销毁函数。

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t * mutex, const pthread_ mutexattr_t * attr);
int pthread_mutex_destroy(pthread_mutex_t * mutex);
```

➔ 成功时返回 0，失败时返回其他值。

- mutex　创建互斥量时传递保存互斥量的变量地址值，销毁时传递需要销毁的互斥量地址值。
- attr　传递即将创建的互斥量属性，没有特别需要指定的属性时传递NULL。

从上述函数声明中也可看出，为了创建相当于锁系统的互斥量，需要声明如下pthread_mutex_t型变量：

```
pthread_mutex_t mutex;
```

该变量的地址将传递给pthread_mutex_init函数，用来保存操作系统创建的互斥量（锁系统）。调用pthread_mutex_destroy函数时同样需要该信息。如果不需要配置特殊的互斥量属性，则向第二个参数传递NULL时，可以利用PTHREAD_MUTEX_INITIALIZER宏进行如下声明：

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```


但推荐各位尽可能使用pthread_mutex_init函数进行初始化，因为通过宏进行初始化时很难发现发生的错误。接下来介绍利用互斥量锁住或释放临界区时使用的函数。

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t * mutex);
int pthread_mutex_unlock(pthread_mutex_t * mutex);
```

➔ 成功时返回 0，失败时返回其他值。

函数名本身含有lock、unlock等词汇，很容易理解其含义。进入临界区前调用的函数就是pthread_mutex_lock。调用该函数时，发现有其他线程已进入临界区，则pthread_mutex_lock函数不会返回，直到里面的线程调用pthread_mutex_unlock函数退出临界区为止。也就是说，其他线程让出临界区之前，当前线程将一直处于阻塞状态。接下来整理一下保护临界区的代码块编写方法。创建好互斥量的前提下，可以通过如下结构保护临界区。

```
pthread_mutex_lock(&mutex);
// 临界区的开始
// . . . . .
// 临界区的结束
pthread_mutex_unlock(&mutex);
```

简言之，就是利用lock和unlock函数围住临界区的两端。此时互斥量相当于一把锁，阻止多个线程同时访问。还有一点需要注意，线程退出临界区时，如果忘了调用pthread_mutex_unlock函数，那么其他为了进入临界区而调用pthread_mutex_lock函数的线程就无法摆脱阻塞状态。这

种情况称为“死锁”（Dead-lock），需要格外注意。接下来利用互斥量解决示例thread4.c中遇到的问题。

❖ mutex.c

```
1.  #include <"与示例thread4.c的头声明一致。">
2.  #define NUM_THREAD  100
3.  void * thread_inc(void * arg);
4.  void * thread_des(void * arg);
5.
6.  long long num=0;
7.  pthread_mutex_t mutex;
8.
9.  int main(int argc, char *argv[])
10. {
11.     pthread_t thread_id[NUM_THREAD];
12.     int i;
13.
14.     pthread_mutex_init(&mutex, NULL);
15.
16.     for(i=0; i<NUM_THREAD; i++)
17.     {
18.         if(i%2)
19.             pthread_create(&(thread_id[i]), NULL, thread_inc, NULL);
20.         else
21.             pthread_create(&(thread_id[i]), NULL, thread_des, NULL);
22.     }
23.
24.     for(i=0; i<NUM_THREAD; i++)
25.         pthread_join(thread_id[i], NULL);
26.
27.     printf("result: %lld \n", num);
28.     pthread_mutex_destroy(&mutex);
29.     return 0;
30. }
31.
32. void * thread_inc(void * arg)
33. {
34.     int i;
35.     pthread_mutex_lock(&mutex);
36.     for(i=0; i<50000000; i++)
37.         num+=1;
38.     pthread_mutex_unlock(&mutex);
39.     return NULL;
40. }
41. void * thread_des(void * arg)
42. {
43.     int i;
44.     for(i=0; i<50000000; i++)
45.     {
46.         pthread_mutex_lock(&mutex);
47.         num-=1;
```

```
48.         pthread_mutex_unlock(&mutex);
49.     }
50.     return NULL;
51. }
```

- 第7行：声明了保存互斥量读取值的变量。之所以声明全局变量是因为，thread_inc函数和thread_des函数都需要访问互斥量。
- 第28行：销毁互斥量。不需要互斥量时应该销毁。
- 第35、38行：实际临界区只是第37行。但此处连同第36行的循环语句一起用作临界区，调用了lock、unlock函数。关于这一点稍后再讨论。
- 第46、48行：通过lock、unlock函数围住对应于临界区的第47行语句。

❖ 运行结果：mutex.c

```
root@my_linux:/tcpip# gcc mutex.c -D_REENTRANT -o mutex -lpthread
root@my_linux:/tcpip# ./mutex
result: 0
```


从运行结果可以看出，已解决了示例thread4.c中的问题。但确认运行结果需要等待较长时间。因为互斥量lock、unlock函数的调用过程要比想象中花费更长时间。首先分析一下thread_inc函数的同步过程。

```
void * thread_inc(void * arg)
{
    int i;
    pthread_mutex_lock(&mutex);
    for(i = 0; i < 50000000; i++)
        num += 1;
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

以上临界区划分范围较大，但这是考虑到如下优点所做的决定：

“最大限度减少互斥量lock、unlock函数的调用次数。”

上述示例中，thread_des函数比thread_inc函数多调用49,999,999次互斥量lock、unlock函数，表现出人可以感知的速度差异。如果不太关注线程的等待时间，可以适当扩展临界区。但变量num的值增加到50,000,000前不允许其他线程访问，这反而成了缺点。其实这里没有正确答案，需要根据不同程序酌情考虑究竟扩大还是缩小临界区。此处没有公式可言，各位需要培养自己的判断能力。

信号量

下面介绍信号量。信号量与互斥量极为相似，在互斥量的基础上很容易理解信号量。此处只涉及利用“二进制信号量”（只用0和1）完成“控制线程顺序”为中心的同步方法。下面给出信号量创建及销毁方法。

```
#include <semaphore.h>

int sem_init(sem_t * sem, int pshared, unsigned int value);
int sem_destroy(sem_t * sem);
```

➔ 成功时返回 0，失败时返回其他值。

- sem　创建信号量时传递保存信号量的变量地址值，销毁时传递需要销毁的信号量变量地址值。
- pshared　传递其他值时，创建可由多个进程共享的信号量；传递0时，创建只允许1个进程内部使用的信号量。我们需要完成同一进程内的线程同步，故传递0。
- value　指定新创建的信号量初始值。

上述函数的pshared参数超出了我们关注的范围，故默认向其传递0。稍后讲解通过value参数初始化的信号量值究竟是多少。接下来介绍信号量中相当于互斥量lock、unlock的函数。

```
#include <semaphore.h>

int sem_post(sem_t * sem);
int sem_wait(sem_t * sem);
```

➔ 成功时返回 0，失败时返回其他值。

- sem　传递保存信号量读取值的变量地址值，传递给sem_post时信号量增1，传递给sem_wait时信号量减1。

调用sem_init函数时，操作系统将创建信号量对象，此对象中记录着“信号量值”（Semaphore Value）整数。该值在调用sem_post函数时增1，调用sem_wait函数时减1。但信号量的值不能小于0，因此，在信号量为0的情况下调用sem_wait函数时，调用函数的线程将进入阻塞状态（因为函数未返回）。当然，此时如果有其他线程调用sem_post函数，信号量的值将变为1，而原本阻塞的线程可以将该信号量重新减为0并跳出阻塞状态。实际上就是通过这种特性完成临界区的同步操作，可以通过如下形式同步临界区（假设信号量的初始值为1）。

```
sem_wait(&sem);  // 信号量变为 0. . .
// 临界区的开始
// . . . . . .
// 临界区的结束
```

```
sem_post(&sem);  // 信号量变为1. . .
```

上述代码结构中，调用sem_wait函数进入临界区的线程在调用sem_post函数前不允许其他线程进入临界区。信号量的值在0和1之间跳转，因此，具有这种特性的机制称为“二进制信号量”。接下来给出信号量相关示例。即将介绍的示例并非关于同时访问的同步，而是关于控制访问顺序的同步。该示例的场景如下：

> “线程A从用户输入得到值后存入全局变量num，此时线程B将取走该值并累加。该过程共进行5次，完成后输出总和并退出程序。”

为了按照上述要求构建程序，应按照线程A、线程B的顺序访问变量num，且需要线程同步。接下来给出示例，分析该示例可能需要花费一定时间。

❖ semaphore.c


```
1.  #include <stdio.h>
2.  #include <pthread.h>
3.  #include <semaphore.h>
4.
5.  void * read(void * arg);
6.  void * accu(void * arg);
7.  static sem_t sem_one;
8.  static sem_t sem_two;
9.  static int num;
10.
11. int main(int argc, char *argv[])
12. {
13.     pthread_t id_t1, id_t2;
14.     sem_init(&sem_one, 0, 0);
15.     sem_init(&sem_two, 0, 1);
16.
17.     pthread_create(&id_t1, NULL, read, NULL);
18.     pthread_create(&id_t2, NULL, accu, NULL);
19.
20.     pthread_join(id_t1, NULL);
21.     pthread_join(id_t2, NULL);
22.
23.     sem_destroy(&sem_one);
24.     sem_destroy(&sem_two);
25.     return 0;
26. }
27.
28. void * read(void * arg)
29. {
30.     int i;
31.     for(i=0; i<5; i++)
32.     {
33.         fputs("Input num: ", stdout);
34.
35.         sem_wait(&sem_two);
```

```
36.         scanf("%d", &num);
37.         sem_post(&sem_one);
38.     }
39.     return NULL;
40. }
41. void * accu(void * arg)
42. {
43.     int sum=0, i;
44.     for(i=0; i<5; i++)
45.     {
46.         sem_wait(&sem_one);
47.         sum+=num;
48.         sem_post(&sem_two);
49.     }
50.     printf("Result: %d \n", sum);
51.     return NULL;
52. }
```

- 第14、15行：生成2个信号量，一个信号量的值为0，另一个为1。一定要掌握需要2个信号量的原因。
- 第35、48行：利用信号量变量sem_two调用wait函数和post函数。这是为了防止在调用accu函数的线程还未取走数据的情况下，调用read函数的线程覆盖原值。
- 第37、46行：利用信号量变量sem_one调用wait和post函数。这是为了防止调用read函数的线程写入新值前，accu函数取走（再取走旧值）数据。

❖ 运行结果：semaphore.c

```
root@my_linux:/tcpip# gcc semaphore.c -D_REENTRANT -o sema -lpthread
root@my_linux:/tcpip# ./sema
Input num: 1
Input num: 2
Input num: 3
Input num: 4
Input num: 5
Result: 15
```

如果各位还不太理解为何需要2个信号量，可将代码中的注释部分去掉，再运行程序并观察运行结果。以上就是线程相关的全部理论知识，下面在此基础上编写服务器端。

18.5　线程的销毁和多线程并发服务器端的实现

我们之前只讨论了线程的创建和控制，而线程的销毁同样重要。下面先介绍线程的销毁，再实现多线程服务器端。

销毁线程的 3 种方法

Linux线程并不是在首次调用的线程main函数返回时自动销毁，所以用如下2种方法之一加以明确。否则由线程创建的内存空间将一直存在。

- 调用pthread_join函数。
- 调用pthread_detach函数。

之前调用过pthread_join函数。调用该函数时，不仅会等待线程终止，还会引导线程销毁。但该函数的问题是，线程终止前，调用该函数的线程将进入阻塞状态。因此，通常通过如下函数调用引导线程销毁。

```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

➔ **成功时返回 0，失败时返回其他值。**

- thread　终止的同时需要销毁的线程ID。

调用上述函数不会引起线程终止或进入阻塞状态，可以通过该函数引导销毁线程创建的内存空间。调用该函数后不能再针对相应线程调用pthread_join函数，这需要格外注意。虽然还有方法在创建线程时可以指定销毁时机，但与pthread_detach方式相比，结果上没有太大差异，故省略其说明。在下面的多线程并发服务器端的实现过程中，希望各位同样关注线程销毁的部分。

多线程并发服务器端的实现

本节并不打算介绍回声服务器端，而是介绍多个客户端之间可以交换信息的简单的聊天程序。希望各位通过本示例复习线程的使用方法及同步的处理方法，还可以再次思考临界区的处理方式。

无论服务器端还是客户端，代码量都不少，故省略可以从其他示例中得到或从源代码中下载的头文件声明。同时最大程度地减少异常处理的代码。

❖ chat_server.c

```
1.  #include <"头文件声明请参考源文件。">
2.  #define BUF_SIZE 100
3.  #define MAX_CLNT 256
4.
5.  void * handle_clnt(void * arg);
6.  void send_msg(char * msg, int len);
7.  void error_handling(char * msg);
8.
```

```
9.  int clnt_cnt=0;
10. int clnt_socks[MAX_CLNT];
11. pthread_mutex_t mutx;
12.
13. int main(int argc, char *argv[])
14. {
15.     int serv_sock, clnt_sock;
16.     struct sockaddr_in serv_adr, clnt_adr;
17.     int clnt_adr_sz;
18.     pthread_t t_id;
19.     if(argc!=2) {
20.         printf("Usage : %s <port>\n", argv[0]);
21.         exit(1);
22.     }
23.
24.     pthread_mutex_init(&mutx, NULL);
25.     serv_sock=socket(PF_INET, SOCK_STREAM, 0);
26.
27.     memset(&serv_adr, 0, sizeof(serv_adr));
28.     serv_adr.sin_family=AF_INET;
29.     serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
30.     serv_adr.sin_port=htons(atoi(argv[1]));
31.
32.     if(bind(serv_sock, (struct sockaddr*) &serv_adr, sizeof(serv_adr))==-1)
33.         error_handling("bind() error");
34.     if(listen(serv_sock, 5)==-1)
35.         error_handling("listen() error");
36.
37.     while(1)
38.     {
39.         clnt_adr_sz=sizeof(clnt_adr);
40.         clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_adr,&clnt_adr_sz);
41.
42.         pthread_mutex_lock(&mutx);
43.         clnt_socks[clnt_cnt++]=clnt_sock;
44.         pthread_mutex_unlock(&mutx);
45.
46.         pthread_create(&t_id, NULL, handle_clnt, (void*)&clnt_sock);
47.         pthread_detach(t_id);
48.         printf("Connected client IP: %s \n", inet_ntoa(clnt_adr.sin_addr));
49.     }
50.     close(serv_sock);
51.     return 0;
52. }
53.
54. void * handle_clnt(void * arg)
55. {
56.     int clnt_sock=*((int*)arg);
57.     int str_len=0, i;
58.     char msg[BUF_SIZE];
59.
60.     while((str_len=read(clnt_sock, msg, sizeof(msg)))!=0)
61.         send_msg(msg, str_len);
62.
```

```
63.     pthread_mutex_lock(&mutx);
64.     for(i=0; i<clnt_cnt; i++) // remove disconnected client
65.     {
66.         if(clnt_sock==clnt_socks[i])
67.         {
68.             while(i++<clnt_cnt-1)
69.                 clnt_socks[i]=clnt_socks[i+1];
70.             break;
71.         }
72.     }
73.     clnt_cnt--;
74.     pthread_mutex_unlock(&mutx);
75.     close(clnt_sock);
76.     return NULL;
77. }
78. void send_msg(char * msg, int len) // send to all
79. {
80.     int i;
81.     pthread_mutex_lock(&mutx);
82.     for(i=0; i<clnt_cnt; i++)
83.         write(clnt_socks[i], msg, len);
84.     pthread_mutex_unlock(&mutx);
85. }
86. void error_handling(char * msg)
87. {
88.     //与之前示例的error_handling函数一致。
89. }
```

- 第9、10行：用于管理接入的客户端套接字的变量和数组。访问这2个变量的代码将构成临界区。
- 第43行：每当有新连接时，将相关信息写入变量clnt_cnt和clnt_socks。
- 第46行：创建线程向新接入的客户端提供服务。由该线程执行第53行定义的函数。
- 第47行：调用pthread_detach函数从内存中完全销毁已终止的线程。
- 第78行：该函数负责向所有连接的客户端发送消息。

上述示例中，各位必须掌握的并不是聊天服务器端的实现方式，而是临界区的构成形式。上述示例中的临界区具有如下特点：

> “访问全局变量clnt_cnt和数组clnt_socks的代码将构成临界区！”

添加或删除客户端时，变量clnt_cnt和数组clnt_socks同时发生变化。因此，在如下情形中均会导致数据不一致，从而引发严重错误。

- 线程A从数组clnt_socks中删除套接字信息，同时线程B读取clnt_cnt变量。
- 线程A读取变量clnt_cnt，同时线程B将套接字信息添加到clnt_socks数组。

因此，如上述示例所示，访问变量clnt_cnt和数组clnt_socks的代码应组织在一起并构成临界区。大家现在应该对我之前说过的这句话有同感了吧：

“此处的‘访问’是指值的更改。产生问题的原因可能还有很多，因此需要准确理解。”

接下来介绍聊天客户端，客户端示例为了分离输入和输出过程而创建了线程。代码分析并不难，故省略源代码相关说明。

❖ chat_clnt.c

```
1.  #include <"头文件声明请参考源文件。">
2.  #define BUF_SIZE 100
3.  #define NAME_SIZE 20
4.
5.  void * send_msg(void * arg);
6.  void * recv_msg(void * arg);
7.  void error_handling(char * msg);
8.
9.  char name[NAME_SIZE]="[DEFAULT]";
10. char msg[BUF_SIZE];
11.
12. int main(int argc, char *argv[])
13. {
14.     int sock;
15.     struct sockaddr_in serv_addr;
16.     pthread_t snd_thread, rcv_thread;
17.     void * thread_return;
18.     if(argc!=4) {
19.         printf("Usage : %s <IP> <port> <name>\n", argv[0]);
20.         exit(1);
21.     }
22.
23.     sprintf(name, "[%s]", argv[3]);
24.     sock=socket(PF_INET, SOCK_STREAM, 0);
25.
26.     memset(&serv_addr, 0, sizeof(serv_addr));
27.     serv_addr.sin_family=AF_INET;
28.     serv_addr.sin_addr.s_addr=inet_addr(argv[1]);
29.     serv_addr.sin_port=htons(atoi(argv[2]));
30.
31.     if(connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr))==-1)
32.         error_handling("connect() error");
33.
34.     pthread_create(&snd_thread, NULL, send_msg, (void*)&sock);
35.     pthread_create(&rcv_thread, NULL, recv_msg, (void*)&sock);
36.     pthread_join(snd_thread, &thread_return);
37.     pthread_join(rcv_thread, &thread_return);
38.     close(sock);
39.     return 0;
40. }
41.
42. void * send_msg(void * arg)    // send thread main
43. {
44.     int sock=*((int*)arg);
45.     char name_msg[NAME_SIZE+BUF_SIZE];
```

```
46.     while(1)
47.     {
48.         fgets(msg, BUF_SIZE, stdin);
49.         if(!strcmp(msg,"q\n")||!strcmp(msg,"Q\n"))
50.         {
51.             close(sock);
52.             exit(0);
53.         }
54.         sprintf(name_msg,"%s %s", name, msg);
55.         write(sock, name_msg, strlen(name_msg));
56.     }
57.     return NULL;
58. }
59. void * recv_msg(void * arg)    // read thread main
60. {
61.     int sock=*((int*)arg);
62.     char name_msg[NAME_SIZE+BUF_SIZE];
63.     int str_len;
64.     while(1)
65.     {
66.         str_len=read(sock, name_msg, NAME_SIZE+BUF_SIZE-1);
67.         if(str_len==-1)
68.             return (void*)-1;
69.         name_msg[str_len]=0;
70.         fputs(name_msg, stdout);
71.     }
72.     return NULL;
73. }
74.
75. void error_handling(char *msg)
76. {
77.     //与之前示例的error_handling函数一致。
78. }
```

下面给出运行结果。接入服务器端的客户端IP均为127.0.0.1，因为服务器端和客户端均在同一台计算机中运行。

❖ 运行结果：chat_server.c

```
root@my_linux:/tcpip# gcc chat_serv.c -D_REENTRANT -o cserv -lpthread
root@my_linux:/tcpip# ./cserv 9190
Connected client IP: 127.0.0.1
Connected client IP: 127.0.0.1
Connected client IP: 127.0.0.1
```

❖ 运行结果：chat_clnt.c One [From Yoon]

```
root@my_linux:/tcpip# gcc chat_clnt.c -D_REENTRANT -o cclnt -lpthread
root@my_linux:/tcpip# ./cclnt 127.0.0.1 9190 Yoon
Hi everyone~
```

```
[Yoon] Hi everyone~
[Choi] Hi Yoon
[Hong] Hi~ friends
```

❖ 运行结果：chat_clnt.c Two [From Choi]

```
root@my_linux:/tcpip# ./cclnt 127.0.0.1 9190 Choi
[Yoon] Hi everyone~
Hi Yoon
[Choi] Hi Yoon
[Hong] Hi~ friends
```

❖ 运行结果：chat_clnt.c Three [From Hong]

```
root@my_linux:/tcpip# ./cclnt 127.0.0.1 9190 Hong
[Yoon] Hi everyone~
[Choi] Hi Yoon
Hi~ friends
[Hong] Hi~ friends
```

18.6　习题

(1) 单CPU系统中如何同时执行多个进程？请解释该过程中发生的上下文切换。

(2) 为何线程的上下文切换速度相对更？线程间数据交换为何不需要类似IPC的特别技术？

(3) 请从执行流角度说明进程和线程的区别。

(4) 下列关于临界区的说法错误的是？

a. 临界区是多个线程同时访问时发生问题的区域。
b. 线程安全的函数中不存在临界区，即便多个线程同时调用也不会发生问题。
c. 1个临界区只能由1个代码块，而非多个代码块构成。换言之，线程A执行的代码块A和线程B执行的代码块B之间绝对不会构成临界区。
d. 临界区由访问全局变量的代码构成。其他变量中不会发生问题。

(5) 下列关于线程同步的描述错误的是？

a. 线程同步就是限制访问临界区。
b. 线程同步也具有控制线程执行顺序的含义。
c. 互斥量和信号量是典型的同步技术。
d. 线程同步是代替进程IPC的技术。

(6) 请说明完全销毁Linux线程的2种方法。

(7) 请利用多线程技术实现回声服务器端，但要让所有线程共享保存客户端消息的内存空间（char数组）。这么做只是为了应用本章的同步技术，其实不符合常理。

(8) 上一题要求所有线程共享保存回声消息的内存空间，如果采用这种方式，无论是否同步都会产生问题。请说明每种情况各产生哪些问题。