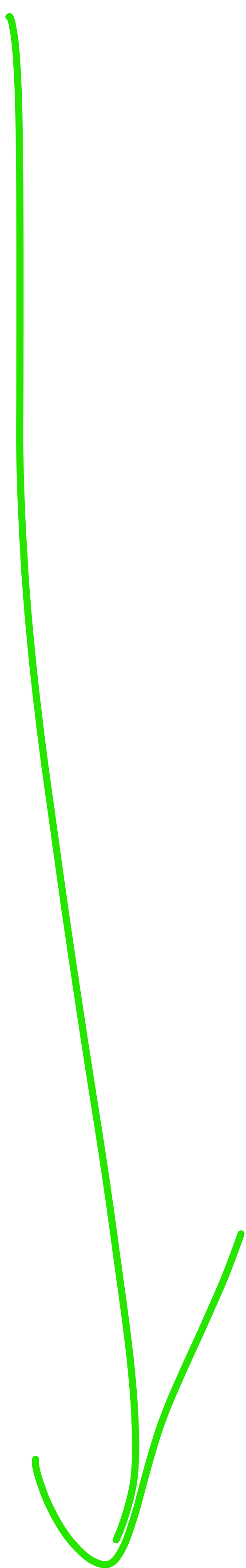


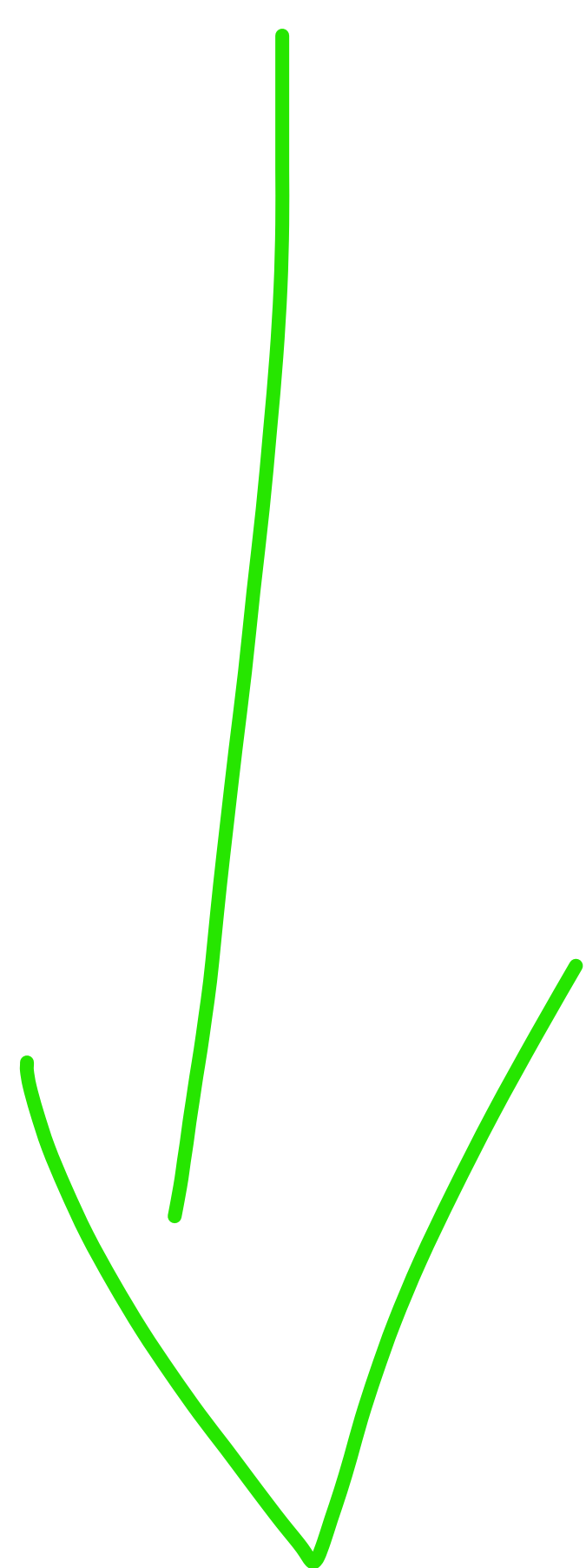
- c. 能够加入多播组的主机数并无限制，但只能有1个主机（Sender）向该组发送数据。
 - d. 多播时使用的套接字是UDP套接字，因为多播是基于UDP进行数据通信的。
- (4) 多播也对网络流量有利，请比较TCP数据交换方式解释其原因。
- (5) 多播方式的数据通信需要MBone虚拟网络。换言之，MBone是用于多播的网络，但它是虚拟网络。请解释此处的“虚拟网络”。



done!



done



Part 02

基于Linux的编程

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。

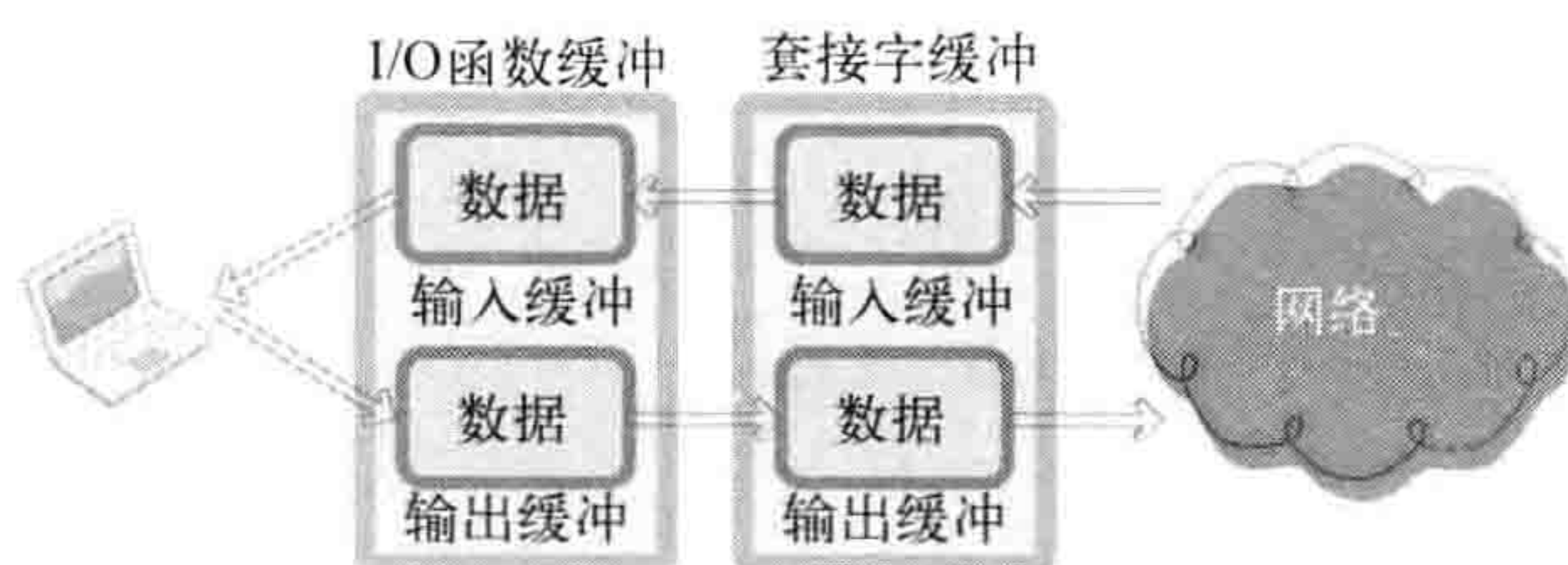


图15-1 缓冲的关系

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 $40 \times 10 = 400$ 字节
- 10个字节 1次 $40 \times 1 = 40$ 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
- mode

需要转换的文件描述符。

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. }
```

代码说明

- 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. }
```

代码说明

- 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？

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“TCPI/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_addr, clnt_addr;
16.     socklen_t clnt_addr_sz;
17.     char buf[BUF_SIZE]={0,};
18.
```

```

19. serv_sock=socket(PF_INET, SOCK_STREAM, 0);
20. memset(&serv_addr, 0, sizeof(serv_addr));
21. serv_addr.sin_family=AF_INET;
22. serv_addr.sin_addr.s_addr=htonl(INADDR_ANY);
23. serv_addr.sin_port=htons(atoi(argv[1]));
24.
25. bind(serv_sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
26. listen(serv_sock, 5);
27. clnt_addr_sz=sizeof(clnt_addr);
28. clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_addr,&clnt_addr_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. }

```

代码说明

- 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. }

```

代码说明

- 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

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。

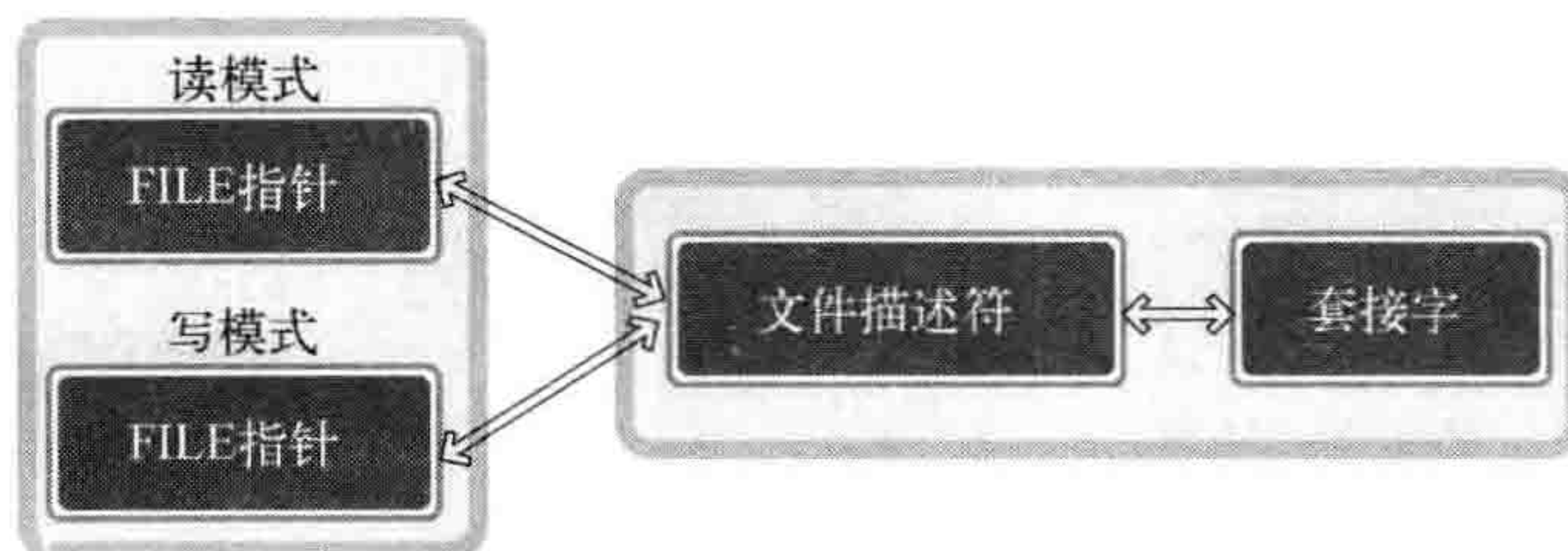


图16-1 FILE指针的关系

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。

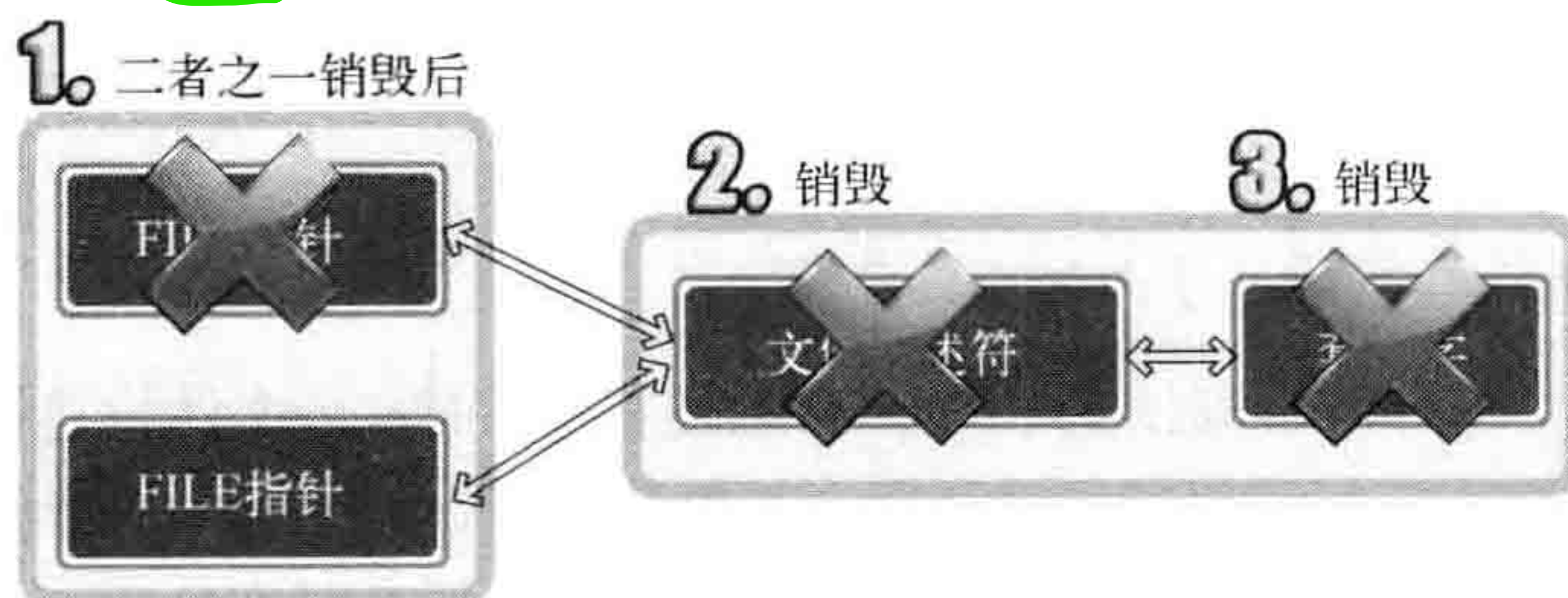


图16-2 调用fclose函数的结果

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。

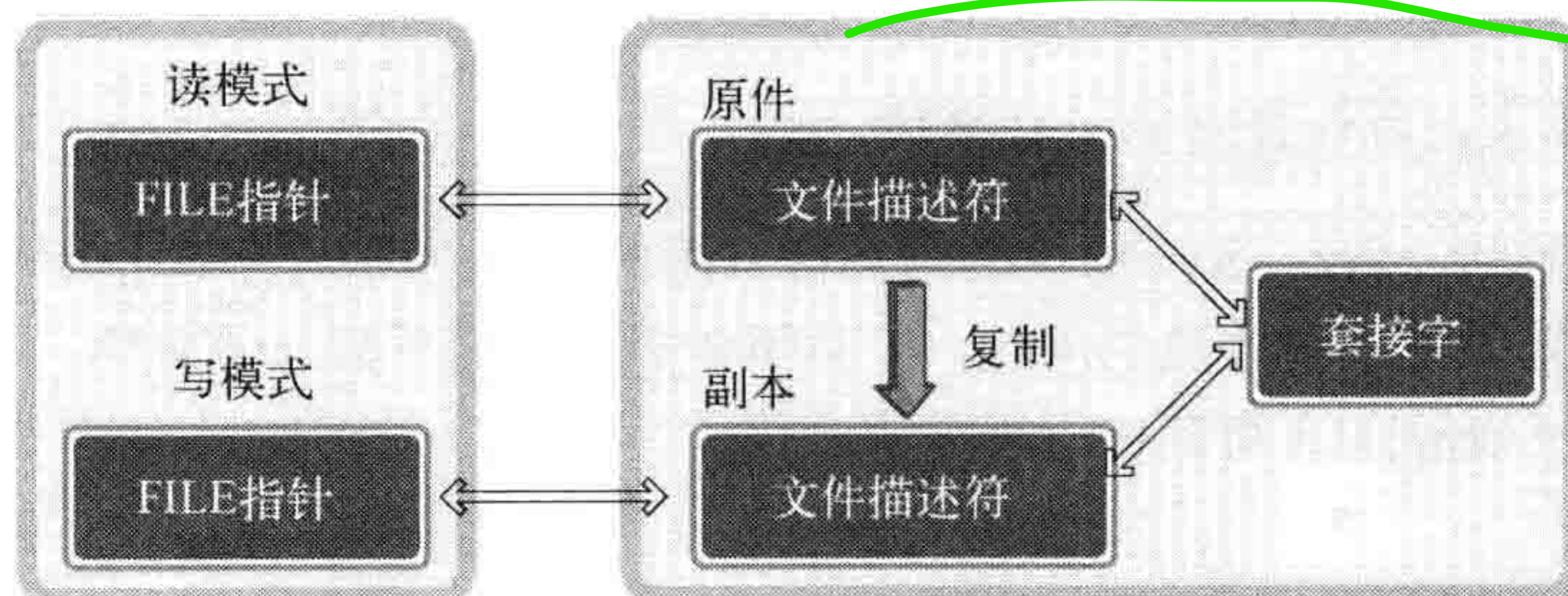


图16-3 半关闭模型1

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）。

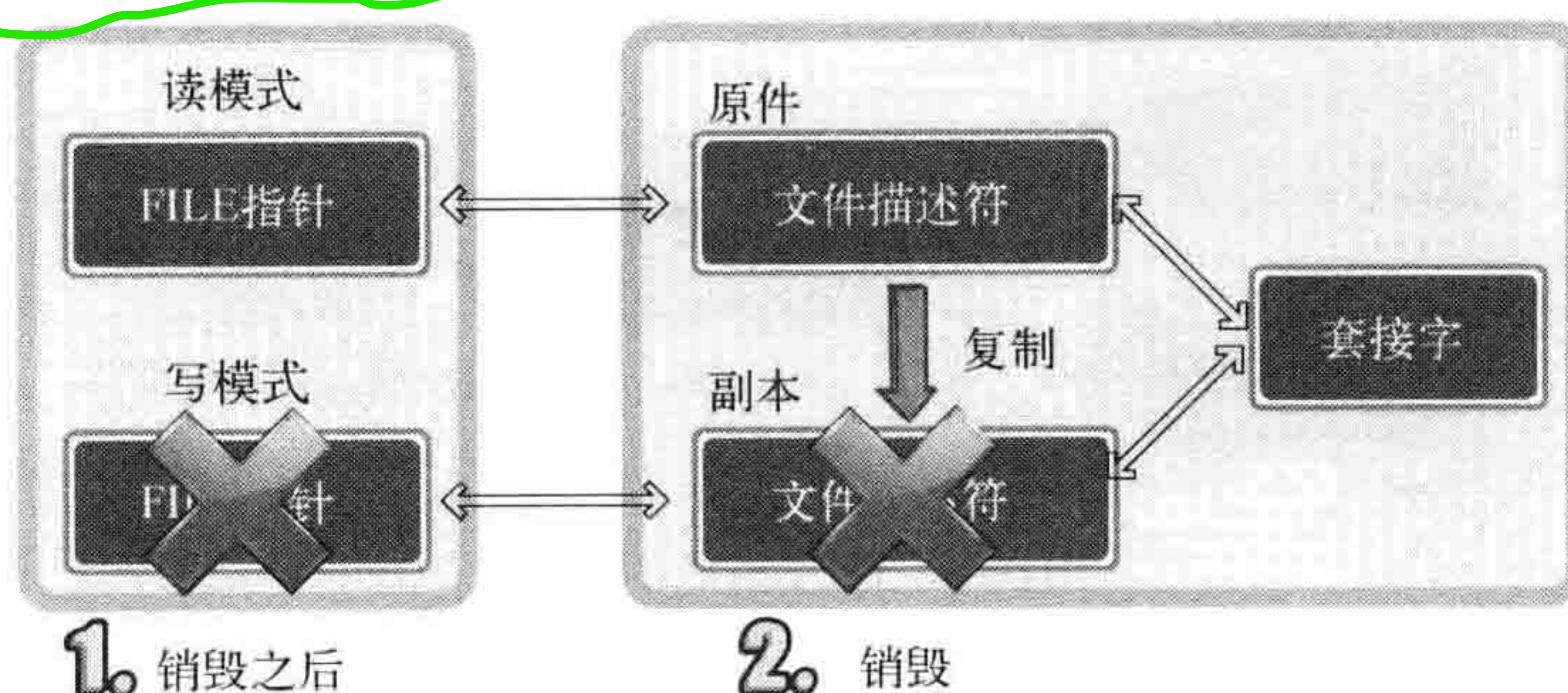


图16-4 半关闭模型2

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。

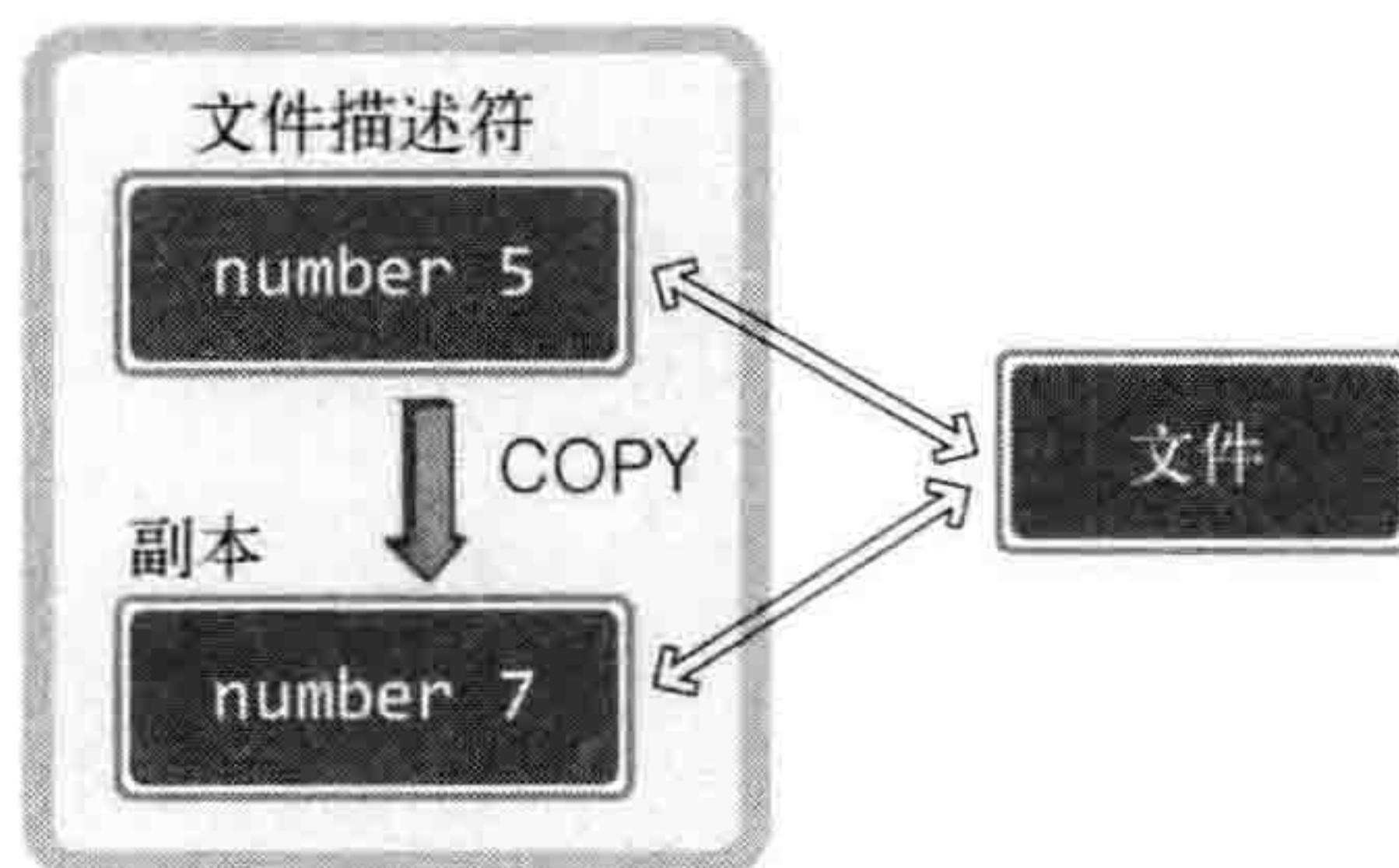


图16-5 文件描述符的复制

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. }
```

代码说明

- 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:/tcip# gcc sep_serv2.c -o serv2
root@my_linux:/tcip# ./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？

- 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。
- 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。
- 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。
- 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。
- 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？

- 终止文件描述符时发送EOF。
- 即使未完全终止文件描述符，关闭输出流时也会发送EOF。
- 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。
- 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。

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。

提示

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。

17

+ 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(EPOOL_SIZE);
42.     ep_events=malloc(sizeof(struct epoll_event)*EPOOL_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, EPOOL_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_addr, clnt_addr;
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:/tcip# gcc echo_EPLTserv.c -o serv
root@my_linux:/tcip# ./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。

提示

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。

17

❖ 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_handling函数一致。
99. }

```

代码说明

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

```

❖ 运行结果：echo_client.c

```

root@my_linux:/tcip# gcc echo_client.c -o clnt
root@my_linux:/tcip# ./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。

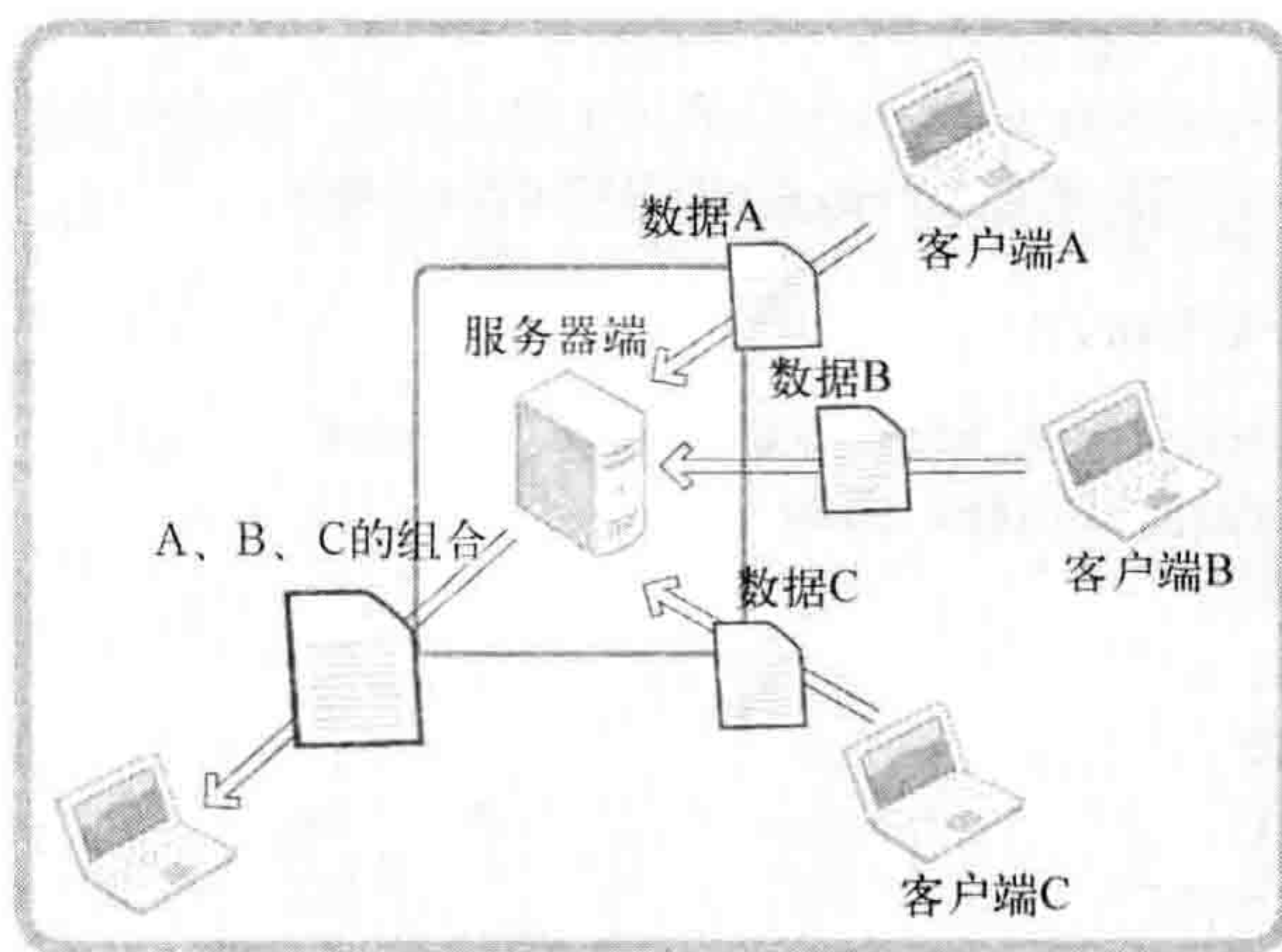


图17-1 理解边缘触发

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

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。

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