# 第 6 章 基于UDP的服务器端/客户端

我们通过第 4 章和第 5 章学习了 TCP 相关知识。TCP 是内容相对较多的一种协议，而本章介绍的 UDP 则篇幅较短。虽然比 TCP 内容少，但在实际操作中很有用，希望各位认真学习。



## 6.1 理解 UDP

我们在第4章学习TCP的过程中，还同时了解了TCP/IP协议栈。在4层TCP/IP模型中，上数第二层传输( Transport )层分为TCP和UDP这2种。数据交换过程可以分为通过TCP套接字完成的TCP方式和通过UDP套接字完成的UDP方式。

### UDP 套接字的特点

下面通过信件说明UDP的工作原理，这是讲解UDP时使用的传统示例，它与UDP特性完全相符。寄信前应先在信封上填好寄信人和收信人的地址，之后贴上邮票放进邮筒即可。当然，信件的特点使我们无法确认对方是否收到。另外，邮寄过程中也可能发生信件丢失的情况。也就是说，信件是一种不可靠的传输方式。与之类似，UDP提供的同样是不可靠的数据传输服务。

“既然如此，TCP应该是更优质的协议吧？”

如果只考虑可靠性，TCP的确比UDP好。但UDP在结构上比TCP更简洁。UDP不会发送类似ACK的应答消息，也不会像SEQ那样给数据包分配序号。因此，UDP的性能有时比TCP高出很多。编程中实现UDP也比TCP简单。另外，UDP的可靠性虽比不上TCP，但也不会像想象中那么频繁地发生数据损毁。因此，在更重视性能而非可靠性的情况下，UDP是一种很好的选择。

既然如此，UDP的作用到底是什么呢？为了提供可靠的数据传输服务，TCP在不可靠的IP层进行流控制，而UDP就缺少这种流控制机制。

“UDP和TCP的差异只在于流控制机制吗？”

是的，流控制是区分UDP和TCP的最重要的标志。但若从TCP中除去流控制，所剩内容也屈指可数。也就是说，TCP的生命在于流控制。第5章讲过的“与对方套接字连接及断开连接过程”也属于流控制的一部分。

**虽然电话比信件要快，但是……**

我把 TCP 比喻为电话，把 UDP 比喻为信件。但这只是形容协议工作方式，并没有包含数据交换速率。请不要误认为“电话的速度比信件快，因此 TCP 的数据收发速率也比 UDP 快”。实际上正好相反。TCP 的速度无法超过 UDP，但在收发某些类型的数据时有可能接近 UDP。例如，每次交换的数据量越大，TCP 的传输速率就越接近 UDP 的传输速率。

## UDP 内部工作原理

与TCP不同，UDP不会进行流控制。接下来具体讨论UDP的作用，如图6-1所示。

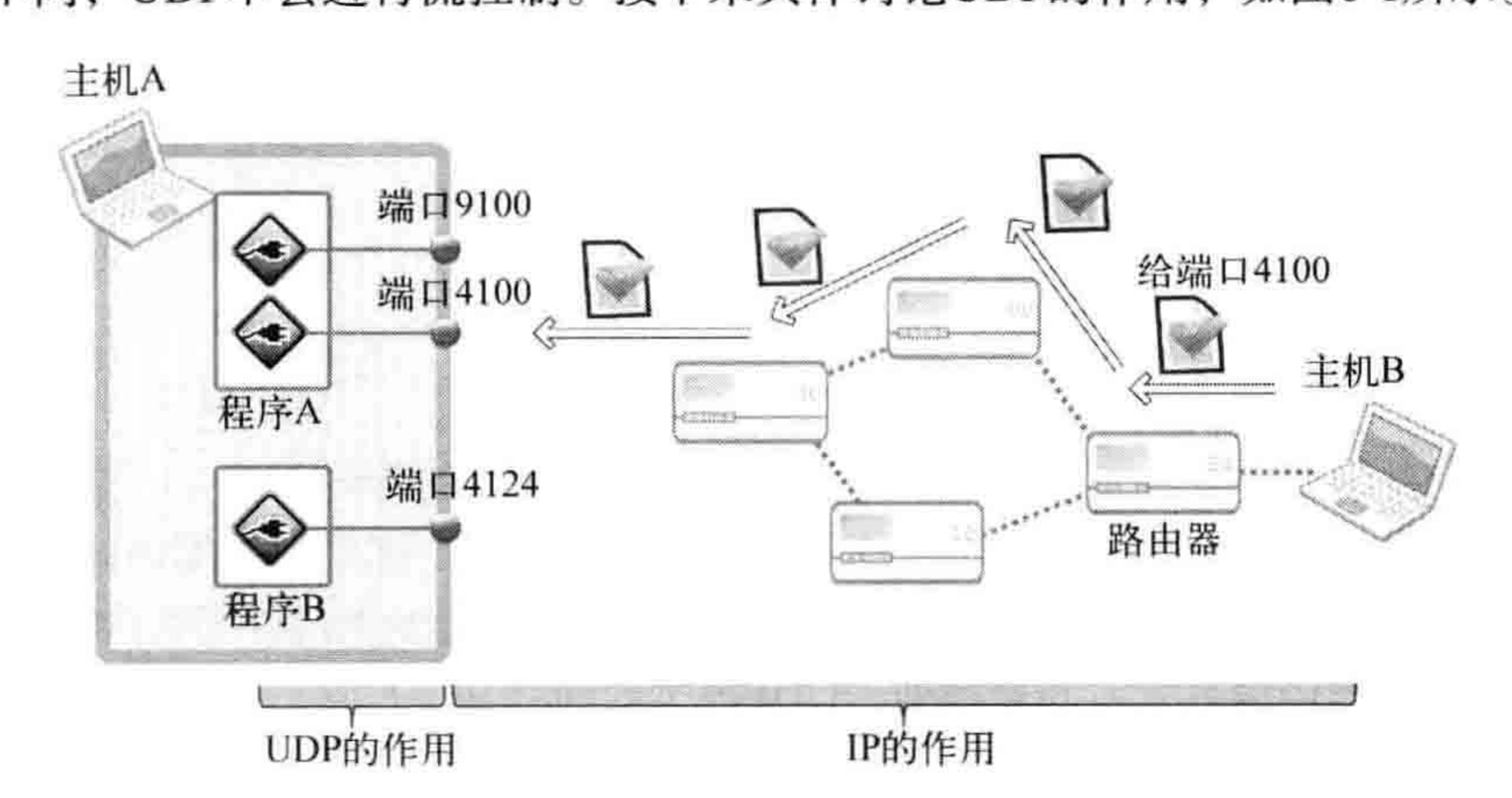



图6-1　数据包传输中UDP和IP的作用

从图6-1中可以看出，IP的作用就是让离开主机B的UDP数据包准确传递到主机A。但把UDP包最终交给主机A的某一UDP套接字的过程则是由UDP完成的。UDP最重要的作用就是根据端口号将传到主机的数据包交付给最终的UDP套接字。

### UDP 的高效使用

虽然貌似大部分网络编程都基于TCP实现，但也有一些是基于UDP实现的。接下来考虑何时使用UDP更有效。讲解前希望各位明白，UDP也具有一定的可靠性。网络传输特性导致信息丢失频发，可若要传递压缩文件（发送1万个数据包时，只要丢失1个就会产生问题），则必须使用TCP，因为压缩文件只要丢失一部分就很难解压。但通过网络实时传输视频或音频时的情况有所不同。对于多媒体数据而言，丢失一部分也没有太大问题，这只会引起短暂的画面抖动，或出现细微的杂音。但因为需要提供实时服务，速度就成为非常重要的因素。因此，第5章的流控制就显得有些多余，此时需要考虑使用UDP。但UDP并非每次都快于TCP，TCP比UDP慢的原因通常有以下两点。

- 收发数据前后进行的连接设置及清除过程。
- 收发数据过程中为保证可靠性而添加的流控制

如果收发的数据量小但需要频繁连接时，UDP比TCP更高效。有机会的话，希望各位深入学习TCP/IP协议的内部构造。C语言程序员懂得计算机结构和操作系统知识就能写出更好的程序，同样，网络程序员若能深入理解TCP/IP协议则可大幅提高自身实力。



## 6.2 实现基于 UDP 的服务器端/客户端

接下来通过之前介绍的UDP理论实现真正的程序。对于UDP而言，只要能理解之前的内容，实现并非难事。

### UDP 中的服务器端和客户端没有连接

UDP服务器端/客户端不像TCP那样在连接状态下交换数据，因此与TCP不同，无需经过连接过程。也就是说，不必调用TCP连接过程中调用的listen函数和accept函数。UDP中只有创建套接字的过程和数据交换过程。

### UDP 服务器端和客户端均只需 1 个套接字

TCP中，套接字之间应该是一对一的关系。若要向10个客户端提供服务，则除了守门的服务器套接字外，还需要10个服务器端套接字。但在UDP中，不管是服务器端还是客户端都只需要1个套接字。之前解释UDP原理时举了信件的例子，收发信件时使用的邮筒可以比喻为UDP套接字。只要附近有1个邮筒，就可以通过它向任意地址寄出信件。同样，只需1个UDP套接字就可以向任意主机传输数据，如图6-2所示。

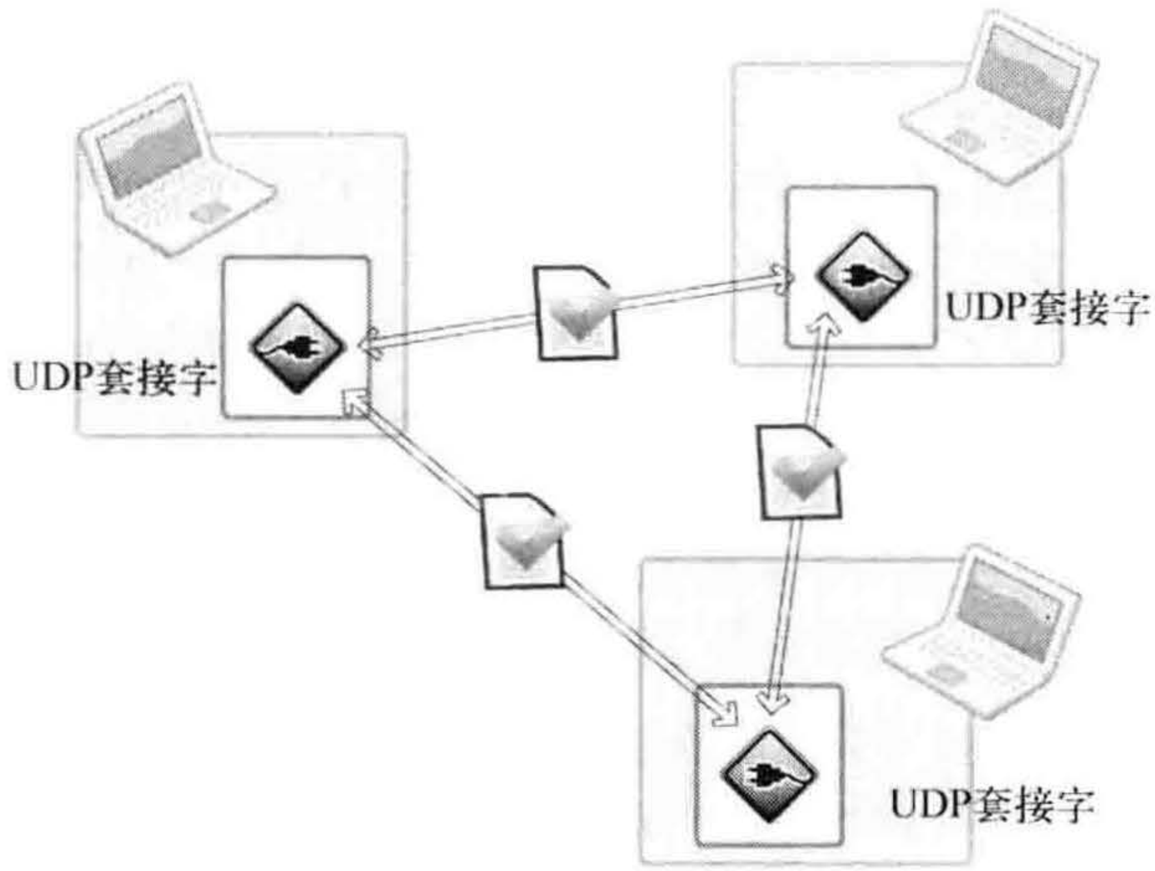



图6-2　UDP套接字通信模型

图6-2展示了1个UDP套接字与2个不同主机交换数据的过程。也就是说，只需1个UDP套接字就能和多台主机通信。

## 基于 UDP 的数据 I/O 函数

创建好TCP套接字后，传输数据时无需再添加地址信息。因为TCP套接字将保持与对方套接字的连接。换言之，TCP套接字知道目标地址信息。但UDP套接字不会保持连接状态（UDP套接字只有简单的邮筒功能），因此每次传输数据都要添加目标地址信息。这相当于寄信前在信件中填写地址。接下来介绍填写地址并传输数据时调用的UDP相关函数。

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

ssize_t sendto(int sock, void *buff, size_t nbytes, int flags,
                                        struct sockaddr *to, socklen_t addrlen);
```

➔ 成功时返回传输的字节数，失败时返回-1。

- sock　用于传输数据的UDP套接字文件描述符。
- buff　保存待传输数据的缓冲地址值。
- nbytes　待传输的数据长度，以字节为单位。
- flags　可选项参数，若没有则传递0。
- to　存有目标地址信息的sockaddr结构体变量的地址值。
- addrlen　传递给参数to的地址值结构体变量长度。

上述函数与之前的TCP输出函数最大的区别在于，此函数需要向它传递目标地址信息。接下

来介绍接收UDP数据的函数。UDP数据的发送端并不固定，因此该函数定义为可接收发送端信息的形式，也就是将同时返回UDP数据包中的发送端信息。

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

ssize_t recvfrom(int sock, void *buff, size_t nbytes, int flags,
                                    struct sockaddr * from, socklen_t *addrlen);
```

➔ 成功时返回接收的字节数，失败时返回-1。

- sock　用于接收数据的UDP套接字文件描述符。
- buff　保存接收数据的缓冲地址值。
- nbytes　可接收的最大字节数，故无法超过参数buff所指的缓冲大小。
- flags　可选项参数，若没有则传入0。
- from　存有发送端地址信息的sockaddr结构体变量的地址值。
- addrlen　保存参数from的结构体变量长度的变量地址值。

编写UDP程序时最核心的部分就在于上述两个函数，这也说明二者在UDP数据传输中的地位。

## 基于 UDP 的回声服务器端/客户端

下面结合之前的内容实现回声服务器。需要注意的是，UDP不同于TCP，不存在请求连接和受理过程，因此在某种意义上无法明确区分服务器端和客户端。只是因其提供服务而称为服务器端，希望各位不要误解。

❖ uecho_server.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 serv_sock;
14.     char message[BUF_SIZE];
15.     int str_len;
16.     socklen_t clnt_adr_sz;
17.
```

```
18.     struct sockaddr_in serv_adr, clnt_adr;
19.     if(argc!=2){
20.         printf("Usage : %s <port>\n", argv[0]);
21.         exit(1);
22.     }
23.
24.     serv_sock=socket(PF_INET, SOCK_DGRAM, 0);
25.     if(serv_sock == -1)
26.         error_handling("UDP socket creation error");
27.
28.     memset(&serv_adr, 0, sizeof(serv_adr));
29.     serv_adr.sin_family=AF_INET;
30.     serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
31.     serv_adr.sin_port=htons(atoi(argv[1]));
32.
33.     if(bind(serv_sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr))==-1)
34.         error_handling("bind() error");
35.
36.     while(1)
37.     {
38.         clnt_adr_sz=sizeof(clnt_adr);
39.         str_len=recvfrom(serv_sock, message, BUF_SIZE, 0,
40.                 (struct sockaddr*)&clnt_adr, &clnt_adr_sz);
41.         sendto(serv_sock, message, str_len, 0,
42.                 (struct sockaddr*)&clnt_adr, clnt_adr_sz);
43.     }
44.     close(serv_sock);
45.     return 0;
46. }
47.
48. void error_handling(char *message)
49. {
50.     fputs(message, stderr);
51.     fputc('\n', stderr);
52.     exit(1);
53. }
```

- 第24行：为了创建UDP套接字，向socket函数第二个参数传递SOCK_DGRAM。
- 第39行：利用第33行分配的地址接收数据。不限制数据传输对象。
- 第41行：通过第39行的函数调用同时获取数据传输端的地址。正是利用该地址将接收的数据逆向重传。
- 第44行：第37行的while内部从未加入break语句，因此是无限循环。也就是说，close函数不会执行，没有太大意义。

接下来介绍与上述服务器端协同工作的客户端。这部分代码与TCP客户端不同，不存在connect函数调用。

❖ uecho_client.c

```
1.  #include <"与uecho_server.c的头声明相同，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 sock;
8.      char message[BUF_SIZE];
9.      int str_len;
10.     socklen_t adr_sz;
11.
12.     struct sockaddr_in serv_adr, from_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_DGRAM, 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.     while(1)
28.     {
29.         fputs("Insert message(q to quit): ", stdout);
30.         fgets(message, sizeof(message), stdin);
31.         if(!strcmp(message,"q\n") || !strcmp(message,"Q\n"))
32.             break;
33.
34.         sendto(sock, message, strlen(message), 0,
35.             (struct sockaddr*)&serv_adr, sizeof(serv_adr));
36.         adr_sz=sizeof(from_adr);
37.         str_len=recvfrom(sock, message, BUF_SIZE, 0,
38.                 (struct sockaddr*)&from_adr, &adr_sz);
39.         message[str_len]=0;
40.         printf("Message from server: %s", message);
41.     }
42.     close(sock);
43.     return 0;
44. }
45.
46. void error_handling(char *message)
47. {
48.     fputs(message, stderr);
49.     fputc('\n', stderr);
50.     exit(1);
51. }
```

- 第18行：创建UDP套接字。现在只需调用数据收发函数。
- 第34、37行：第34行向服务器端传输数据，第37行接收数据。

若各位很好地理解了第4章的connect函数，那么读上述代码时应有如下疑问：

> “TCP客户端套接字在调用connect函数时自动分配IP地址和端口号，既然如此，UDP客户端何时分配IP地址和端口号？”

所有套接字都应分配IP地址和端口，问题是直接分配还是自动分配。希望大家独立思考并进行推断，稍后再讨论，先给出程序的运行结果。

❖ 运行结果：uecho_server.c

```
root@my_linux:/tcpip# gcc uecho_server.c -o userver
root@my_linux:/tcpip# ./userver 9190
```

❖ 运行结果：uecho_client.c

```
root@my_linux:/tcpip# gcc uecho_client.c -o uclient
root@my_linux:/tcpip# ./uclient 127.0.0.1 9190
Insert message(q to quit): Hi UDP Server?
Message from server: Hi UDP Server?
Insert message(q to quit): Nice to meet you!
Message from server: Nice to meet you!
Insert message(q to quit): Good bye~
Message from server: Good bye~
Insert message(q to quit): q
```

运行过程中的顺序并不重要。只需保证在调用sendto函数前，sendto函数的目标主机程序已经开始运行。

## UDP 客户端套接字的地址分配

前面讲解了UDP服务器端/客户端的实现方法。但如果仔细观察UDP客户端会发现，它缺少把IP和端口分配给套接字的过程。TCP客户端调用connect函数自动完成此过程，而UDP中连能承担相同功能的函数调用语句都没有。究竟在何时分配IP和端口号呢？

UDP程序中，调用sendto函数传输数据前应完成对套接字的地址分配工作，因此调用bind函数。当然，bind函数在TCP程序中出现过，但bind函数不区分TCP和UDP，也就是说，在UDP程序中同样可以调用。另外，如果调用sendto函数时发现尚未分配地址信息，则在首次调用sendto函数时给相应套接字自动分配IP和端口。而且此时分配的地址一直保留到程序结束为止，因此也

可用来与其他UDP套接字进行数据交换。当然，IP用主机IP，端口号选尚未使用的任意端口号。

综上所述，调用sendto函数时自动分配IP和端口号，因此，UDP客户端中通常无需额外的地址分配过程。所以之前示例中省略了该过程，这也是普遍的实现方式。

## 6.3 UDP 的数据传输特性和调用 connect 函数

我们之前通过示例验证了TCP传输的数据不存在数据边界，本节将验证UDP数据传输中存在数据边界。最后讨论UDP中connect函数的调用，以此结束UDP相关讨论。

### 存在数据边界的 UDP 套接字

前面说过TCP数据传输中不存在边界，这表示“数据传输过程中调用I/O函数的次数不具有任何意义。”

相反，UDP是具有数据边界的协议，传输中调用I/O函数的次数非常重要。因此，输入函数的调用次数应和输出函数的调用次数完全一致，这样才能保证接收全部已发送数据。例如，调用3次输出函数发送的数据必须通过调用3次输入函数才能接收完。下面通过简单示例进行验证。



❖ bound_host1.c

```
1.  #include <"与其他程序的头声明一致，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 sock;
8.      char message[BUF_SIZE];
9.      struct sockaddr_in my_adr, your_adr;
10.     socklen_t adr_sz;
11.     int str_len, i;
12.
13.     if(argc!=2) {
14.         printf("Usage : %s <port>\n", argv[0]);
15.         exit(1);
16.     }
17.
18.     sock=socket(PF_INET, SOCK_DGRAM, 0);
19.     if(sock==-1)
20.         error_handling("socket() error");
21.
22.     memset(&my_adr, 0, sizeof(my_adr));
23.     my_adr.sin_family=AF_INET;
24.     my_adr.sin_addr.s_addr=htonl(INADDR_ANY);
25.     my_adr.sin_port=htons(atoi(argv[1]));
```

```
26.
27.     if(bind(sock, (struct sockaddr*)&my_adr, sizeof(my_adr))==-1)
28.         error_handling("bind() error");
29.
30.     for(i=0; i<3; i++)
31.     {
32.         sleep(5);        // delay 5 sec.
33.         adr_sz=sizeof(your_adr);
34.         str_len=recvfrom(sock, message, BUF_SIZE, 0,
35.                 (struct sockaddr*)&your_adr, &adr_sz);
36.
37.         printf("Message %d: %s \n", i+1, message);
38.     }
39.     close(sock);
40.     return 0;
41. }
42.
43. void error_handling(char *message)
44. {
45.     //与其他示例的error_handling函数定义一致，故省略。
46. }
```

上述示例中需要各位特别留意的是第30行中的for语句。首先在第32行中调用sleep函数，使程序停顿时间等于传递来的时间（以秒为单位）参数。也就是说，第30行的for循环中每隔5秒调用1次recvfrom函数。另外还添加了验证函数调用次数的语句。稍后再讲解延迟执行程序的原因。

接下来的示例向之前的bound_host1.c传输数据，该示例共调用sendto函数3次以传输字符串数据。

❖ bound_host2.c

```
1.  #include <"与其他示例头声明一致，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 sock;
8.      char msg1[]="Hi!";
9.      char msg2[]="I'm another UDP host!";
10.     char msg3[]="Nice to meet you";
11.
12.     struct sockaddr_in your_adr;
13.     socklen_t your_adr_sz;
14.     if(argc!=3){
15.         printf("Usage : %s <IP> <port>\n", argv[0]);
16.         exit(1);
17.     }
18.
19.     sock=socket(PF_INET, SOCK_DGRAM, 0);
20.     if(sock==-1)
21.         error_handling("socket() error");
```

```
22.
23.     memset(&your_adr, 0, sizeof(your_adr));
24.     your_adr.sin_family=AF_INET;
25.     your_adr.sin_addr.s_addr=inet_addr(argv[1]);
26.     your_adr.sin_port=htons(atoi(argv[2]));
27.
28.     sendto(sock, msg1, sizeof(msg1), 0,
29.         (struct sockaddr*)&your_adr, sizeof(your_adr));
30.     sendto(sock, msg2, sizeof(msg2), 0,
31.         (struct sockaddr*)&your_adr, sizeof(your_adr));
32.     sendto(sock, msg3, sizeof(msg3), 0,
33.         (struct sockaddr*)&your_adr, sizeof(your_adr));
34.     close(sock);
35.     return 0;
36. }
37.
38. void error_handling(char *message)
39. {
40.     //与其他示例的error_handling函数定义一致，故省略。
41. }
```

bound_host2.c程序3次调用sendto函数以传输数据，bound_host1.c则调用3次recvfrom函数以接收数据。recvfrom函数调用间隔为5秒，因此，调用recvfrom函数前已调用了3次sendto函数。也就是说，此时数据已经传输到bound_host1.c。如果是TCP程序，这时只需调用1次输入函数即可读入数据。UDP则不同，在这种情况下也需要调用3次recvfrom函数。可通过以下运行结果进行验证。



❖ 运行结果：bound_host1.c

```
root@my_linux:/tcpip# gcc bound_host1.c -o host1
root@my_linux:/tcpip# ./host1
Usage : ./host1 <port>
root@my_linux:/tcpip# ./host1 9190
Message 1: Hi!
Message 2: I'm another UDP host!
Message 3: Nice to meet you
root@my_linux:/home/swyoon/tcpip#
```

❖ 运行结果：bound_host2.c

```
root@my_linux:/tcpip# gcc bound_host2.c -o host2
root@my_linux:/tcpip# ./host2
Usage : ./host2 <IP> <port>
root@my_linux:/tcpip# ./host2 127.0.0.1 9190
root@my_linux:/tcpip#
```

从运行结果，特别是bound_host1.c的运行结果中可以看出，共调用了3次recvfrom函数。这就

证明必须在UDP通信过程中使I/O函数调用次数保持一致。

**UDP 数据报（Datagram）**

UDP 套接字传输的数据包又称数据报，实际上数据报也属于数据包的一种。只是与 TCP 包不同，其本身可以成为 1 个完整数据。这与 UDP 的数据传输特性有关，UDP 中存在数据边界，1 个数据包即可成为 1 个完整数据，因此称为数据报。

## 已连接（connected）UDP 套接字与未连接（unconnected）UDP 套接字

TCP套接字中需注册待传输数据的目标IP和端口号，而UDP中则无需注册。因此，通过sendto函数传输数据的过程大致可分为以下3个阶段。

- 第1阶段：向UDP套接字注册目标IP和端口号。
- 第2阶段：传输数据。
- 第3阶段：删除UDP套接字中注册的目标地址信息。

每次调用sendto函数时重复上述过程。每次都变更目标地址，因此可以重复利用同一UDP套接字向不同目标传输数据。这种未注册目标地址信息的套接字称为未连接套接字，反之，注册了目标地址的套接字称为连接connected套接字。显然，UDP套接字默认属于未连接套接字。但UDP套接字在下述情况下显得不太合理：

“IP为211.210.147.82的主机82号端口共准备了3个数据，调用3次sendto函数进行传输。”

此时需重复3次上述三阶段。因此，要与同一主机进行长时间通信时，将UDP套接字变成已连接套接字会提高效率。上述三个阶段中，第一个和第三个阶段占整个通信过程近1/3的时间，缩短这部分时间将大大提高整体性能。

## 创建已连接 UDP 套接字

创建已连接UDP套接字的过程格外简单，只需针对UDP套接字调用connect函数。

```
sock = socket(PF_INET, SOCK_DGRAM, 0);
memset(&adr, 0, sizeof(adr));
adr.sin_family = AF_INET;
adr.sin_addr.s_addr = . . . .
adr.sin_port = . . . .
connect(sock, (struct sockaddr *) &adr, sizeof(adr));
```

上述代码看似与TCP套接字创建过程一致，但socket函数的第二个参数分明是SOCK_DGRAM。也就是说，创建的的确是UDP套接字。当然，针对UDP套接字调用connect函数并不意味着要与对方UDP套接字连接，这只是向UDP套接字注册目标IP和端口信息。

之后就与TCP套接字一样，每次调用sendto函数时只需传输数据。因为已经指定了收发对象，所以不仅可以使用sendto、recvfrom函数，还可以使用write、read函数进行通信。

下列示例将之前的uecho_client.c程序改成基于已连接UDP套接字的程序，因此可以结合uecho_server.c程序运行。另外，为便于说明，未直接删除uecho_client.c的I/O函数，而是添加了注释。

❖ uecho_con_client.c

```
1.  #include <"与其他示例头声明一致，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 sock;
8.      char message[BUF_SIZE];
9.      int str_len;
10.     socklen_t adr_sz;     //多余变量!
11.
12.     struct sockaddr_in serv_adr, from_adr;        //不再需要from_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_DGRAM, 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.     connect(sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr));
28.
29.     while(1)
30.     {
31.         fputs("Insert message(q to quit): ", stdout);
32.         fgets(message, sizeof(message), stdin);
33.         if(!strcmp(message,"q\n") || !strcmp(message,"Q\n"))
34.             break;
35.         /*
36.         sendto(sock, message, strlen(message), 0,
37.             (struct sockaddr*)&serv_adr, sizeof(serv_adr));
38.         */
39.         write(sock, message, strlen(message));
```

```
40.
41.         /*
42.         adr_sz=sizeof(from_adr);
43.         str_len=recvfrom(sock, message, BUF_SIZE, 0,
44.                 (struct sockaddr*)&from_adr, &adr_sz);
45.         */
46.         str_len=read(sock, message, sizeof(message)-1);
47.
48.         message[str_len]=0;
49.         printf("Message from server: %s", message);
50.     }
51.     close(sock);
52.     return 0;
53. }
54.
55. void error_handling(char *message)
56. {
57.     fputs(message, stderr);
58.     fputc('\n', stderr);
59.     exit(1);
60. }
```

我认为没必要给出运行结果和代码说明，故省略。另外需要注意，代码中用write、read函数代替了sendto、recvfrom函数。

## 6.4　基于 Windows 的实现

首先介绍Windows平台下的sendto函数和readfrom函数。实际上与Linux的函数没有太大区别，但为了让各位亲自确认这一点，这里给出其定义。

```
#include <winsock2.h>

int sendto(SOCKET s, const char* buf, int len, int flags, const struct sockaddr*
    to, int tolen);
```

➔ 成功时返回传输的字节数，失败时返回 SOCKET_ERROR。

```
#include <winsock2.h>

int recvfrom(SOCKET s, char* buf, int len, int flag, struct sockaddr* from, int*
    fromlen);
```

➔ 成功时返回接收的字节数，失败时返回 SOCKET_ERROR。

以上两个函数与Linux下的sendto、recvfrom函数相比，其参数个数、顺序及含义完全相同，故省略具体说明。接下来实现Windows平台下的UDP回声服务器端/客户端。其中，回声客户端是利用已连接UDP套接字实现的。

❖ uecho_server_win.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <winsock2.h>
5.  #define BUF_SIZE 30
6.  void ErrorHandling(char *message);
7.
8.  int main(int argc, char *argv[])
9.  {
10.     WSADATA wsaData;
11.     SOCKET servSock;
12.     char message[BUF_SIZE];
13.     int strLen;
14.     int clntAdrSz;
15.
16.     SOCKADDR_IN servAdr, clntAdr;
17.     if(argc!=2) {
18.         printf("Usage : %s <port>\n", argv[0]);
19.         exit(1);
20.     }
21.     if(WSAStartup(MAKEWORD(2, 2), &wsaData)!=0)
22.         ErrorHandling("WSAStartup() error!");
23.
24.     servSock=socket(PF_INET, SOCK_DGRAM, 0);
25.     if(servSock==INVALID_SOCKET)
26.         ErrorHandling("UDP socket creation error");
27.
28.     memset(&servAdr, 0, sizeof(servAdr));
29.     servAdr.sin_family=AF_INET;
30.     servAdr.sin_addr.s_addr=htonl(INADDR_ANY);
31.     servAdr.sin_port=htons(atoi(argv[1]));
32.
33.     if(bind(servSock, (SOCKADDR*)&servAdr, sizeof(servAdr))==SOCKET_ERROR)
34.         ErrorHandling("bind() error");
35.
36.     while(1)
37.     {
38.         clntAdrSz=sizeof(clntAdr);
39.         strLen=recvfrom(servSock, message, BUF_SIZE, 0,
40.                 (SOCKADDR*)&clntAdr, &clntAdrSz);
41.         sendto(servSock, message, strLen, 0,
42.             (SOCKADDR*)&clntAdr, sizeof(clntAdr));
43.     }
44.     closesocket(servSock);
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. }
```

❖ uecho_client_win.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <winsock2.h>
5.
6.  #define BUF_SIZE 30
7.  void ErrorHandling(char *message);
8.
9.  int main(int argc, char *argv[])
10. {
11.     WSADATA wsaData;
12.     SOCKET sock;
13.     char message[BUF_SIZE];
14.     int strLen;
15.
16.     SOCKADDR_IN servAdr;
17.     if(argc!=3) {
18.         printf("Usage : %s <IP> <port>\n", argv[0]);
19.         exit(1);
20.     }
21.     if(WSAStartup(MAKEWORD(2, 2), &wsaData)!=0)
22.         ErrorHandling("WSAStartup() error!");
23.
24.     sock=socket(PF_INET, SOCK_DGRAM, 0);
25.     if(sock==INVALID_SOCKET)
26.         ErrorHandling("socket() error");
27.
28.     memset(&servAdr, 0, sizeof(servAdr));
29.     servAdr.sin_family=AF_INET;
30.     servAdr.sin_addr.s_addr=inet_addr(argv[1]);
31.     servAdr.sin_port=htons(atoi(argv[2]));
32.     connect(sock, (SOCKADDR*)&servAdr, sizeof(servAdr));
33.
34.     while(1)
35.     {
36.         fputs("Insert message(q to quit): ", stdout);
37.         fgets(message, sizeof(message), stdin);
38.         if(!strcmp(message,"q\n") || !strcmp(message,"Q\n"))
39.             break;
40.
41.         send(sock, message, strlen(message), 0);
```

```
42.         strLen=recv(sock, message, sizeof(message)-1, 0);
43.         message[strLen]=0;
44.         printf("Message from server: %s", message);
45.     }
46.     closesocket(sock);
47.     WSACleanup();
48.     return 0;
49. }
50.
51. void ErrorHandling(char *message)
52. {
53.     fputs(message, stderr);
54.     fputc('\n', stderr);
55.     exit(1);
56. }
```

上述客户端示例利用已连接UDP套接字进行输入输出，因此用send、recv函数替换sendto、recvfrom函数。此外也如实反映了已连接UDP套接字的好处。

## 6.5 习题



(1) UDP为什么TCP速度快？为什么TCP数据传输可靠而UDP数据传输不可靠？

(2) 下列不属于UDP特点的是？

a. UDP不同于TCP，不存在连接的概念，所以不像TCP那样只能进行一对一的数据传输。
b. 利用UDP传输数据时，如果有2个目标，则需要2个套接字。
c. UDP套接字中无法使用已分配给TCP的同一端口号。
d. UDP套接字和TCP套接字可以共存。若需要，可以在同一主机进行TCP和UDP数据传输。
e. 针对UDP函数也可以调用connect函数，此时UDP套接字跟TCP套接字相同，也需要经过3次握手过程。

(3) UDP数据报向对方主机的UDP套接字传递过程中，IP和UDP分别负责哪些部分？

(4) UDP一般比TCP快，但根据交换数据的特点，其差异可大可小。请说明何种情况下UDP的性能优于TCP？

(5) 客户端TCP套接字调用connect函数时自动分配IP和端口号。UDP中不调用bind函数，那合适分配IP和端口号？

(6) TCP客户端必需调用connect函数，而UDP中可以选择性调用。请问，在UDP中调用connect函数有哪些好处？

(7) 请参考本章给出的uecho_server.c和uecho_client.c，编写示例使服务器端和客户端轮流收发消息。收发的消息均要输出到控制台窗口。

# 第7章 优雅地断开套接字连接

本章将讨论如何优雅地断开相互连接的套接字。之前用的方法不够优雅是因为，我们是调用 close 或 closesocket 函数单方面断开连接的。

## 7.1 基于 TCP 的半关闭

TCP中的断开连接过程比建立连接过程更重要，因为连接过程中一般不会出现大的变数，但断开过程有可能发生预想不到的情况，因此应准确掌控。只有掌握了下面要讲解的半关闭（Half-close），才能明确断开过程。

### 单方面断开连接带来的问题

Linux的close函数和Windows的closesocket函数意味着完全断开连接。完全断开不仅指无法传输数据，而且也不能接收数据。因此，在某些情况下，通信一方调用close或closesocket函数断开连接就显得不太优雅，如图7-1所示。

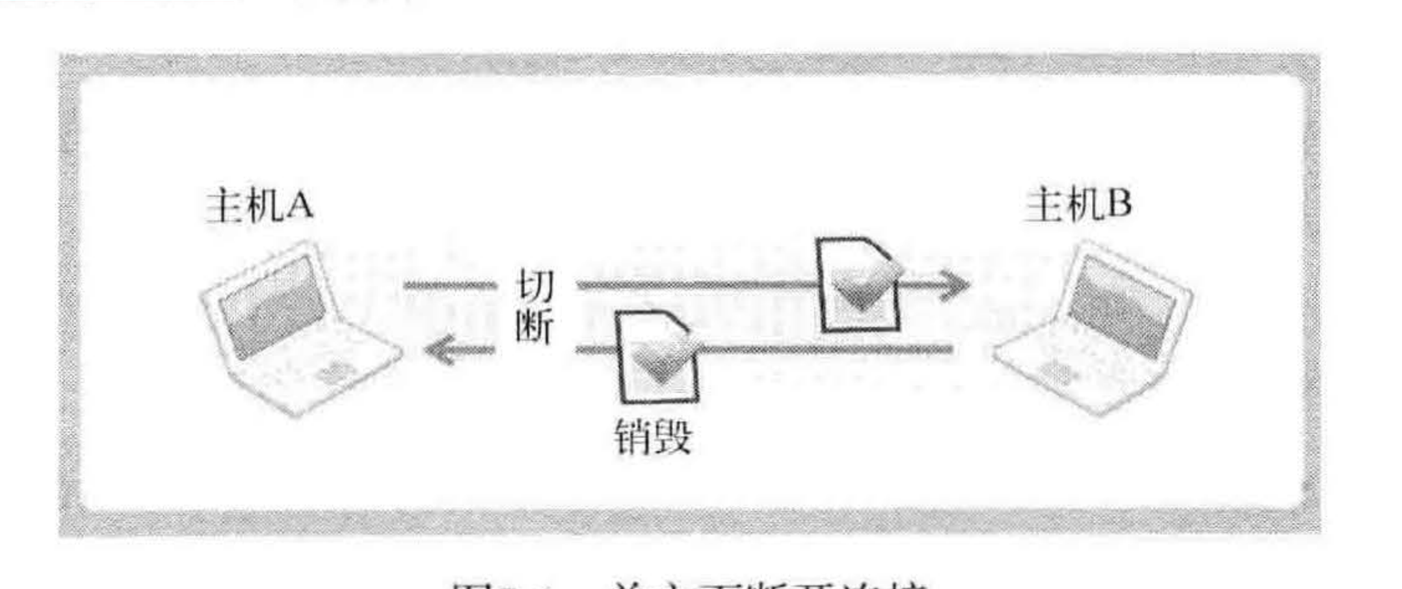



图7-1　单方面断开连接

图7-1描述的是2台主机正在进行双向通信。主机A发送完最后的数据后，调用close函数断开了连接，之后主机A无法再接收主机B传输的数据。实际上，是完全无法调用与接收数据相关的

函数。最终，由主机B传输的、主机A必须接收的数据也销毁了。

为了解决这类问题，“只关闭一部分数据交换中使用的流”（Half-close）的方法应运而生。断开一部分连接是指，可以传输数据但无法接收，或可以接收数据但无法传输。顾名思义就是只关闭流的一半。

## 套接字和流（Stream）

两台主机通过套接字建立连接后进入可交换数据的状态，又称“流形成的状态”。也就是把建立套接字后可交换数据的状态看作一种流。

此处的流可以比作水流。水朝着一个方向流动，同样，在套接字的流中，数据也只能向一个方向移动。因此，为了进行双向通信，需要如图7-2所示的2个流。

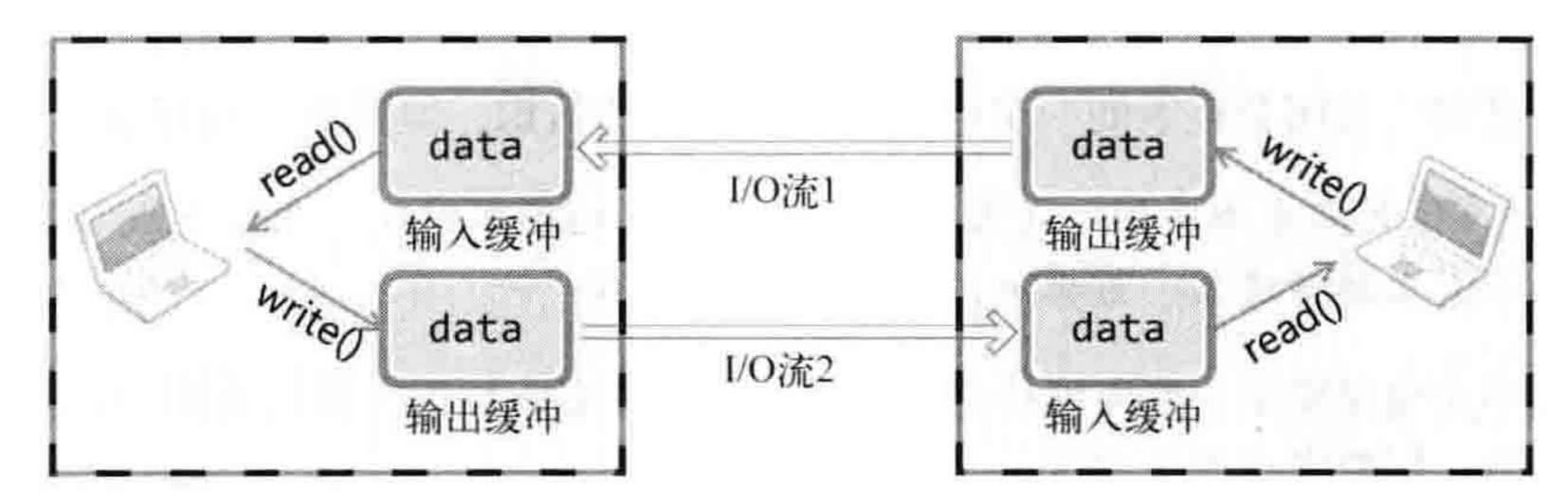



图7-2 套接字中生成的两个流

一旦两台主机间建立了套接字连接，每个主机就会拥有单独的输入流和输出流。当然，其中一个主机的输入流与另一主机的输出流相连，而输出流则与另一主机的输入流相连。另外，本章讨论的“优雅地断开连接方式”只断开其中1个流，而非同时断开两个流。Linux的close和Windows的closesocket函数将同时断开这两个流，因此与“优雅”二字还有一段距离。

## 针对优雅断开的 shutdown 函数

接下来介绍用于半关闭的函数。下面这个shutdown函数就用来关闭其中1个流。

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

int shutdown(int sock, int howto);
```

➔ 成功时返回 0，失败时返回-1。

- sock　需要断开的套接字文件描述符。
- howto　传递断开方式信息。

调用上述函数时，第二个参数决定断开连接的方式，其可能值如下所示。

- SHUT_RD：断开输入流。
- SHUT_WR：断开输出流。
- SHUT_RDWR：同时断开I/O流。

若向shutdown的第二个参数传递SHUT_RD，则断开输入流，套接字无法接收数据。即使输入缓冲收到数据也会抹去，而且无法调用输入相关函数。如果向shutdown函数的第二个参数传递SHUT_WR，则中断输出流，也就无法传输数据。但如果输出缓冲还留有未传输的数据，则将传递至目标主机。最后，若传入SHUT_RDWR，则同时中断I/O流。这相当于分2次调用shutdown，其中一次以SHUT_RD为参数，另一次以SHUT_WR为参数。

## 为何需要半关闭

相信各位已对“关闭套接字的一半连接”有了充分的认识，但还有一些疑惑。

> “究竟为什么需要半关闭？是否只要留出足够长的连接时间，保证完成数据交换即可？只要不急于断开连接，好像也没必要使用半关闭。”

这句话也不完全是错的。如果保持足够的时间间隔，完成数据交换后再断开连接，这时就没必要使用半关闭。但要考虑如下情况：

> “一旦客户端连接到服务器端，服务器端将约定的文件传给客户端，客户端收到后发送字符串‘Thank you’给服务器端。”

此处字符串“Thank you”的传递实际是多余的，这只是用来模拟客户端断开连接前还有数据需要传递的情况。此时程序实现的难度并不小，因为传输文件的服务器端只需连续传输文件数据即可，而客户端则无法知道需要接收数据到何时。客户端也没办法无休止地调用输入函数，因为这有可能导致程序阻塞（调用的函数未返回）。

> “是否可以让服务器端和客户端约定一个代表文件尾的字符？”

这种方式也有问题，因为这意味着文件中不能有与约定字符相同的内容。为解决该问题，服务器端应最后向客户端传递EOF表示文件传输结束。客户端通过函数返回值接收EOF，这样可以避免与文件内容冲突。剩下最后一个问题：服务器如何传递EOF？

> “断开输出流时向对方主机传输EOF。”

当然，调用close函数的同时关闭I/O流，这样也会向对方发送EOF。但此时无法再接收对方传输的数据。换言之，若调用close函数关闭流，就无法接收客户端最后发送的字符串“Thank you”。这时需要调用shutdown函数，只关闭服务器的输出流（半关闭）。这样既可以发送EOF，同时又保留了输入流，可以接收对方数据。下面结合已学内容实现收发文件的服务器端/客户端。

## 基于半关闭的文件传输程序

上述文件传输服务器端和客户端的数据流可整理如图7-3，稍后将根据此图编写示例。希望各位通过此例理解传递EOF的必要性和半关闭的重要性。

图7-3 文件传输数据流程图

首先介绍服务器端。该示例与之前示例不同，省略了大量错误处理代码，希望大家注意。这种处理只是为了便于分析代码，实际编写中不应省略。

❖ file_server.c

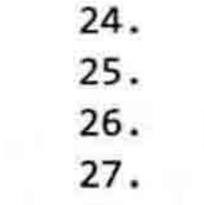
```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <unistd.h>
5.  #include <arpa/inet.h>
6.  #include <sys/socket.h>
7.
8.  #define BUF_SIZE 30
9.  void error_handling(char *message);
10.
11. int main(int argc, char *argv[])
12. {
13.     int serv_sd, clnt_sd;
14.     FILE * fp;
15.     char buf[BUF_SIZE];
16.     int read_cnt;
17.
18.     struct sockaddr_in serv_adr, clnt_adr;
19.     socklen_t clnt_adr_sz;
20.
21.     if(argc!=2) {
```

```
22.         printf("Usage: %s <port>\n", argv[0]);
23.         exit(1);
24.     }
25.
26.     fp=fopen("file_server.c", "rb");
27.     serv_sd=socket(PF_INET, SOCK_STREAM, 0);
28.
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.     bind(serv_sd, (struct sockaddr*)&serv_adr, sizeof(serv_adr));
35.     listen(serv_sd, 5);
36.
37.     clnt_adr_sz=sizeof(clnt_adr);
38.     clnt_sd=accept(serv_sd, (struct sockaddr*)&clnt_adr, &clnt_adr_sz);
39.
40.     while(1)
41.     {
42.         read_cnt=fread((void*)buf, 1, BUF_SIZE, fp);
43.         if(read_cnt<BUF_SIZE)
44.         {
45.             write(clnt_sd, buf, read_cnt);
46.             break;
47.         }
48.         write(clnt_sd, buf, BUF_SIZE);
49.     }
50.
51.     shutdown(clnt_sd, SHUT_WR);
52.     read(clnt_sd, buf, BUF_SIZE);
53.     printf("Message from client: %s \n", buf);
54.
55.     fclose(fp);
56.     close(clnt_sd); close(serv_sd);
57.     return 0;
58. }
59.
60. void error_handling(char *message)
61. {
62.     fputs(message, stderr);
63.     fputc('\n', stderr);
64.     exit(1);
65. }
```

- 第26行：打开文件以向客户端传输服务器端源文件file_server.c。
- 第40~49行：为向客户端传输文件数据而编写的循环语句。此客户端是在第38行的accept函数调用中连接的。
- 第51行：发送文件后针对输出流进行半关闭。这样就向客户端传输了EOF，而客户端也知道文件传输已完成。
- 第52行：只关闭了输出流，依然可以通过输入流接收数据。

❖ file_client.c

```
1.  #include <"与file_server.c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 sd;
8.      FILE *fp;
9.
10.     char buf[BUF_SIZE];
11.     int read_cnt;
12.     struct sockaddr_in serv_adr;
13.     if(argc!=3) {
14.         printf("Usage: %s <IP> <port>\n", argv[0]);
15.         exit(1);
16.     }
17.
18.     fp=fopen("receive.dat", "wb");
19.     sd=socket(PF_INET, SOCK_STREAM, 0);
20.
21.     memset(&serv_adr, 0, sizeof(serv_adr));
22.     serv_adr.sin_family=AF_INET;
23.     serv_adr.sin_addr.s_addr=inet_addr(argv[1]);
24.     serv_adr.sin_port=htons(atoi(argv[2]));
25.
26.     connect(sd, (struct sockaddr*)&serv_adr, sizeof(serv_adr));
27.
28.     while((read_cnt=read(sd, buf, BUF_SIZE ))!=0)
29.         fwrite((void*)buf, 1, read_cnt, fp);
30.
31.     puts("Received file data");
32.     write(sd, "Thank you", 10);
33.     fclose(fp);
34.     close(sd);
35.     return 0;
36. }
37.
38. void error_handling(char *message)
39. {
40.     fputs(message, stderr);
41.     fputc('\n', stderr);
42.     exit(1);
43. }
```




- 第18行：创建新文件以保存服务器端传输的文件数据。
- 第28、29行：接收数据并保存到第18行创建的文件，直到接收EOF为止。
- 第32行：向服务器端发送表示感谢的消息。若服务器端未关闭输入流，则可接收此消息。

下面是上述示例的运行结果。运行后查看客户端的receive.dat文件，可以验证数据正常接收。

特别需要注意的是，还可以确认服务器端已正常接收客户端最后传输的消息“Thank you”。

❖ 运行结果：file_server.c

```
root@my_linux:/tcpip# gcc file_server.c -o fserver
root@my_linux:/tcpip# ./fserver 9190
Message from client: Thank you
root@my_linux:/tcpip#
```

❖ 运行结果：file_client.c

```
root@my_linux:/tcpip# gcc file_client.c -o fclient
root@my_linux:/tcpip# ./fclient 127.0.0.1 9190
Received file data
root@my_linux:/tcpip#
```

## 7.2　基于 Windows 的实现

Windows平台同样通过调用shutdown函数完成半关闭，只是向其传递的参数名略有不同，需要确认。

```
#include <winsock2.h>

int shutdown(SOCKET sock, int howto);
```

➔ 成功时返回 0，失败时返回 SOCKET_ERROR。

- sock　要断开的套接字句柄。
- howto 断开方式的信息。

上述函数中第二个参数的可能值及其含义可整理如下。

- SD_RECEIVE：断开输入流。
- SD_SEND：断开输出流。
- SD_BOTH：同时断开I/O流。

虽然这些常量名不同于Linux中的名称，但其值完全相同。SD_RECEIVE、SHUT_RD都是0，SD_SEND、SHUT_WR都是1，SD_BOTH、SHUT_RDWR都是2。当然，这些并没有太多实际意义。最后，给出Windows平台下的示例。

❖ file_server_win.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <winsock2.h>
5.
6.  #define BUF_SIZE 30
7.  void ErrorHandling(char *message);
8.
9.  int main(int argc, char *argv[])
10. {
11.     WSADATA wsaData;
12.     SOCKET hServSock, hClntSock;
13.     FILE * fp;
14.     char buf[BUF_SIZE];
15.     int readCnt;
16.
17.     SOCKADDR_IN servAdr, clntAdr;
18.     int clntAdrSz;
19.
20.     if(argc!=2) {
21.         printf("Usage: %s <port>\n", argv[0]);
22.         exit(1);
23.     }
24.     if(WSAStartup(MAKEWORD(2, 2), &wsaData)!=0)
25.         ErrorHandling("WSAStartup() error!");
26.
27.     fp=fopen("file_server_win.c", "rb");
28.     hServSock=socket(PF_INET, SOCK_STREAM, 0);
29.
30.     memset(&servAdr, 0, sizeof(servAdr));
31.     servAdr.sin_family=AF_INET;
32.     servAdr.sin_addr.s_addr=htonl(INADDR_ANY);
33.     servAdr.sin_port=htons(atoi(argv[1]));
34.
35.     bind(hServSock, (SOCKADDR*)&servAdr, sizeof(servAdr));
36.     listen(hServSock, 5);
37.
38.     clntAdrSz=sizeof(clntAdr);
39.     hClntSock=accept(hServSock, (SOCKADDR*)&clntAdr, &clntAdrSz);
40.
41.     while(1)
42.     {
43.         readCnt=fread((void*)buf, 1, BUF_SIZE, fp);
44.         if(readCnt<BUF_SIZE)
45.         {
46.             send(hClntSock, (char*)&buf, readCnt, 0);
47.             break;
48.         }
49.         send(hClntSock, (char*)&buf, BUF_SIZE, 0);
50.     }
51.
```

```
52.     shutdown(hClntSock, SD_SEND);
53.     recv(hClntSock, (char*)buf, BUF_SIZE, 0);
54.     printf("Message from client: %s \n", buf);
55.
56.     fclose(fp);
57.     closesocket(hClntSock); closesocket(hServSock);
58.     WSACleanup();
59.     return 0;
60. }
61.
62. void ErrorHandling(char *message)
63. {
64.     fputs(message, stderr);
65.     fputc('\n', stderr);
66.     exit(1);
67. }
```

❖ file_client_win.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <winsock2.h>
5.
6.  #define BUF_SIZE 30
7.  void ErrorHandling(char *message);
8.
9.  int main(int argc, char *argv[])
10. {
11.     WSADATA wsaData;
12.     SOCKET hSocket;
13.     FILE *fp;
14.
15.     char buf[BUF_SIZE];
16.     int readCnt;
17.     SOCKADDR_IN servAdr;
18.
19.     if(argc!=3) {
20.         printf("Usage: %s <IP> <port>\n", argv[0]);
21.         exit(1);
22.     }
23.     if(WSAStartup(MAKEWORD(2, 2), &wsaData)!=0)
24.         ErrorHandling("WSAStartup() error!");
25.
26.     fp=fopen("receive.dat", "wb");
27.     hSocket=socket(PF_INET, SOCK_STREAM, 0);
28.
29.     memset(&servAdr, 0, sizeof(servAdr));
30.     servAdr.sin_family=AF_INET;
31.     servAdr.sin_addr.s_addr=inet_addr(argv[1]);
32.     servAdr.sin_port=htons(atoi(argv[2]));
33.
```

```
34.     connect(hSocket, (SOCKADDR*)&servAdr, sizeof(servAdr));
35.
36.     while((readCnt=recv(hSocket, buf, BUF_SIZE, 0))!=0)
37.         fwrite((void*)buf, 1, readCnt, fp);
38.
39.     puts("Received file data");
40.     send(hSocket, "Thank you", 10, 0);
41.     fclose(fp);
42.     closesocket(hSocket);
43.     WSACleanup();
44.     return 0;
45. }
46.
47. void ErrorHandling(char *message)
48. {
49.     fputs(message, stderr);
50.     fputc('\n', stderr);
51.     exit(1);
52. }
```

运行结果及源代码内容与之前的file_server.c、file_client.c并无太大区别，故省略。

## 7.3 习题

(1) 解释TCP中“流”的概念。UDP中能否形成流？请说明原因。

(2) Linux中的close函数或Windows中的closesocket函数属于单方面断开连接的方法，有可能带来一些问题。什么是单方面断开连接？什么情形下会出现问题？

(3) 什么是半关闭？针对输出流执行半关闭的主机处于何种状态？半关闭会导致对方主机接收什么消息？

# 第8章 域名及网络地址

8

随着互联网用户的不断增加，现在 DNS（Domain Name System，域名系统）几乎无人不知。人们也经常谈论 DNS 相关的专业话题。而且，不懂 DNS 的人用不了 5 分钟就能在网上搜索并学习 DNS 知识。网络的发展促使我们每个人都成了半个网络专家。

## 8.1 域名系统

DNS是对IP地址和域名进行相互转换的系统，其核心是DNS服务器。

### 什么是域名

提供网络服务的服务器端也是通过IP地址区分的，但几乎不可能以非常难记的IP地址形式交换服务器端地址信息。因此，将容易记、易表述的域名分配并取代IP地址。

### DNS 服务器

在浏览器地址栏中输入Naver网站的IP地址222.122.195.5即可浏览Naver网站主页。但我们通常输入Naver网站的域名www.naver.com访问网站。二者之间究竟有何区别？

从进入Naver网站主页这一结果上看，没有区别，但接入过程不同。域名是赋予服务器端的虚拟地址，而非实际地址。因此，需要将虚拟地址转化为实际地址。那如何将域名变为IP地址呢？DNS服务器担此重任，可以向DNS服务器请求转换地址。

“请问DNS服务器，www.naver.com的IP地址是多少？”

所有计算机中都记录着默认DNS服务器地址，就是通过这个默认DNS服务器得到相应域名的IP地址信息。在浏览器地址栏中输入域名后，浏览器通过默认DNS服务器获取该域名对应的IP地

址信息，之后才真正接入该网站。

ping & nslookup

除非商业需要，否则一般不会轻易改变服务器域名，但会相对频繁地改变服务器 IP 地址。如果各位想了解某个域名对应的 IP 地址信息，可以在控制台窗口输入如下内容：

```
ping www.naver.com
```

这样即可知道某一域名对应的 IP 地址。ping 命令用来验证 IP 数据报是否到达目的地，但执行过程中会同时经过“域名到 IP 地址”的转换过程，因此可以通过此命令查看 IP 地址。另外，若各位想知道自己计算机中注册的默认 DNS 服务器地址，可以输入如下命令：

```
nslookup
```

在 Linux 系统中输入上述命令后，会提示进一步输入信息，此时可以输入 server 得到默认 DNS 服务器地址。

计算机内置的默认DNS服务器并不知道网络上所有域名的IP地址信息。若该DNS服务器无法解析，则会询问其他DNS服务器，并提供给用户，如图8-1所示。

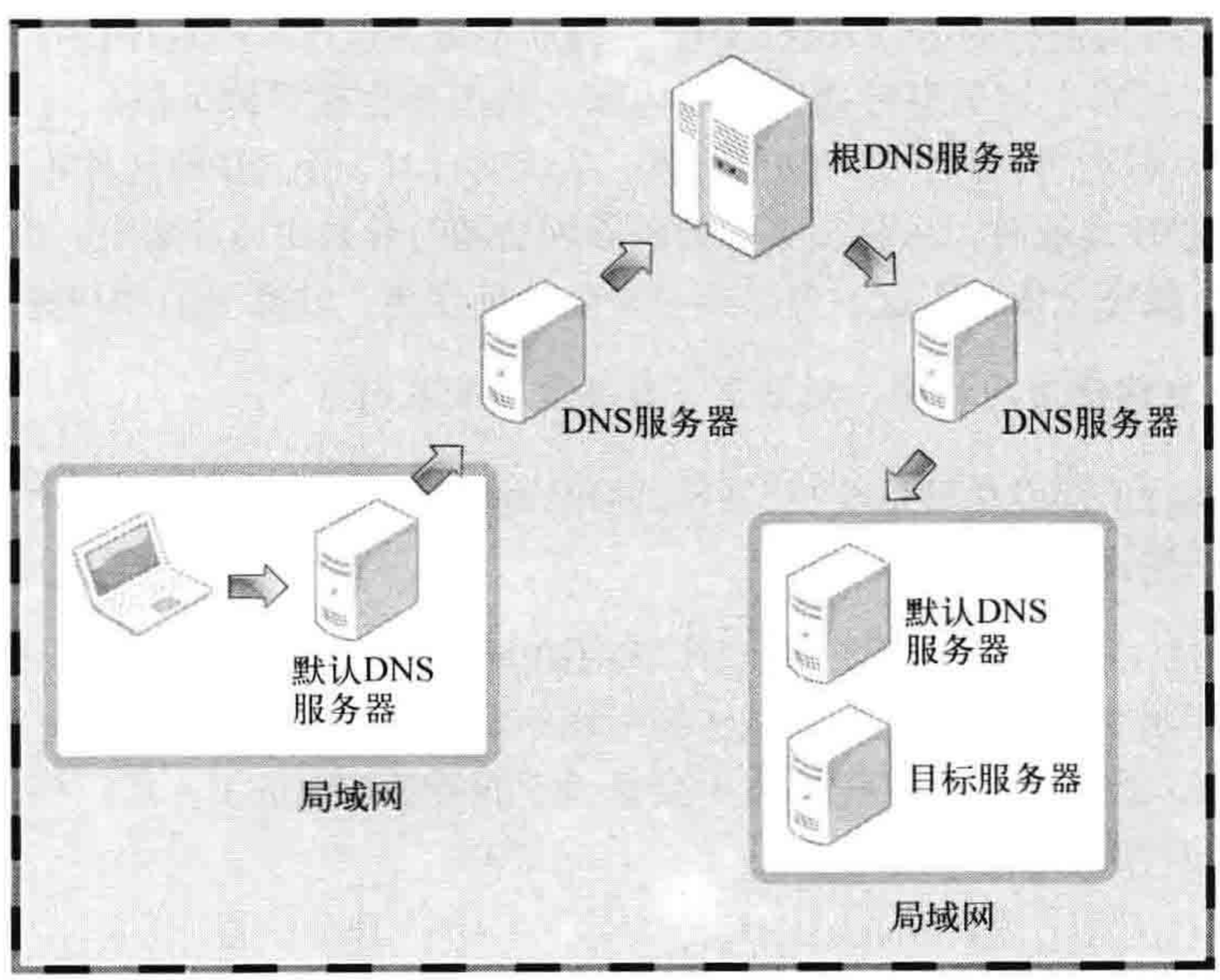



图8-1 DNS和请求获取IP地址信息

图8-1展示了默认DNS服务器无法解析主机询问的域名IP地址时的应答过程。可以看出，默认DNS服务器收到自己无法解析的请求时，向上级DNS服务器询问。通过这种方式逐级向上传递信息，到达顶级DNS服务器——根DNS服务器时，它知道该向哪个DNS服务器询问。向下级DNS传递解析请求，得到IP地址后原路返回，最后将解析的IP地址传递到发起请求的主机。DNS就是这样层次化管理的一种分布式数据库系统。

## 8.2　IP 地址和域名之间的转换

8.1节讲解了域名和IP地址之间的转换过程，本节介绍通过程序向DNS服务器发出解析请求的方法。

### 程序中有必要使用域名吗？

“所有学习都要在开始前认识到其必要性！”这是我经常挂在嘴边的一句话。从语言的基本语法到系统函数，若无法回答“这到底有何必要？”学习过程将变得枯燥无味，而且很容易遗忘。最头疼的是，学完之后很难应用。我们为什么需要将要讨论的转换函数？为了查看某一域名的IP地址吗？当然不是！下面通过示例解释原因。假设各位是运营www.SuperOrange.com域名的公司系统工程师，需要开发客户端使用公司提供的服务。该客户端需要接入如下服务器地址：

IP 211.102.204.12, PORT 2012

应向程序用户提供便利的运行方法，因此，程序不能像运行示例程序那样要求用户输入IP和端口信息。那该如何将上述信息传递到程序内部？难道要直接将地址信息写入程序代码吗？当然，这样便于运行程序，但这种方案也有问题。系统运行时，保持IP地址并不容易。特别是依赖ISP服务提供者维护IP地址时，系统相关的各种原因会随时导致IP地址变更。虽然ISP会保证维持原有IP，但程序不能完全依赖于这一点。万一发生地址变更，就需要向用户进行如下解释：

“请卸载当前使用的程序，到主页下载并重新安装v1.2。”

那么，因为随时可能发生地址变更，所以向用户提供源代码，每次变更地址时让用户改变IP和端口号，并重新编译程序，这又如何？

IP地址比域名发生变更的概率要高，所以利用IP地址编写程序并非上策。还有什么办法呢？一旦注册域名可能永久不变，因此利用域名编写程序会好一些。这样，每次运行程序时根据域名获取IP地址，再接入服务器，这样程序就不会依赖于服务器IP地址了。所以说，程序中也需要IP地址和域名之间的转换函数。

## 利用域名获取 IP 地址

使用以下函数可以通过传递字符串格式的域名获取IP地址。

```
#include <netdb.h>

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

➔ 成功时返回 hostent 结构体地址，失败时返回 NULL 指针。

这个函数使用方便。只要传递域名字符串，就会返回域名对应的IP地址。只是返回时，地址信息装入hostent结构体。此结构体定义如下。

```
struct hostent
{
    char * h_name;          //official name
    char ** h_aliases;      //alias list
    int h_addrtype;         //host address type
    int h_length;           //address length
    char ** h_addr_list;    //address list
}
```

从上述结构体定义中可以看出，不只返回IP信息，同时还连带着其他信息。各位不用想得太过复杂。域名转IP时只需关注h_addr_list。下面简要说明上述结构体各成员。



### h_name

该变量中存有官方域名（Official domain name）。官方域名代表某一主页，但实际上，一些著名公司的域名并未用官方域名注册。

### h_aliases

可以通过多个域名访问同一主页。同一IP可以绑定多个域名，因此，除官方域名外还可指定其他域名。这些信息可以通过h_aliases获得。

### h_addrtype

gethostbyname函数不仅支持IPv4，还支持IPv6。因此可以通过此变量获取保存在h_addr_list的IP地址的地址族信息。若是IPv4，则此变量存有AF_INET。

### h_length

保存IP地址长度。若是IPv4地址，因为是4个字节，则保存4；IPv6时，因为是16个字节，故

保存16。

### h_addr_list

这是最重要的成员。通过此变量以整数形式保存域名对应的IP地址。另外，用户较多的网站有可能分配多个IP给同一域名，利用多个服务器进行负载均衡。此时同样可以通过此变量获取IP地址信息。

调用gethostbyname函数后返回的hostent结构体的变量结构如图8-2所示，该图在实际编程中非常有用，希望大家结合之前的hostent结构体定义加以理解。

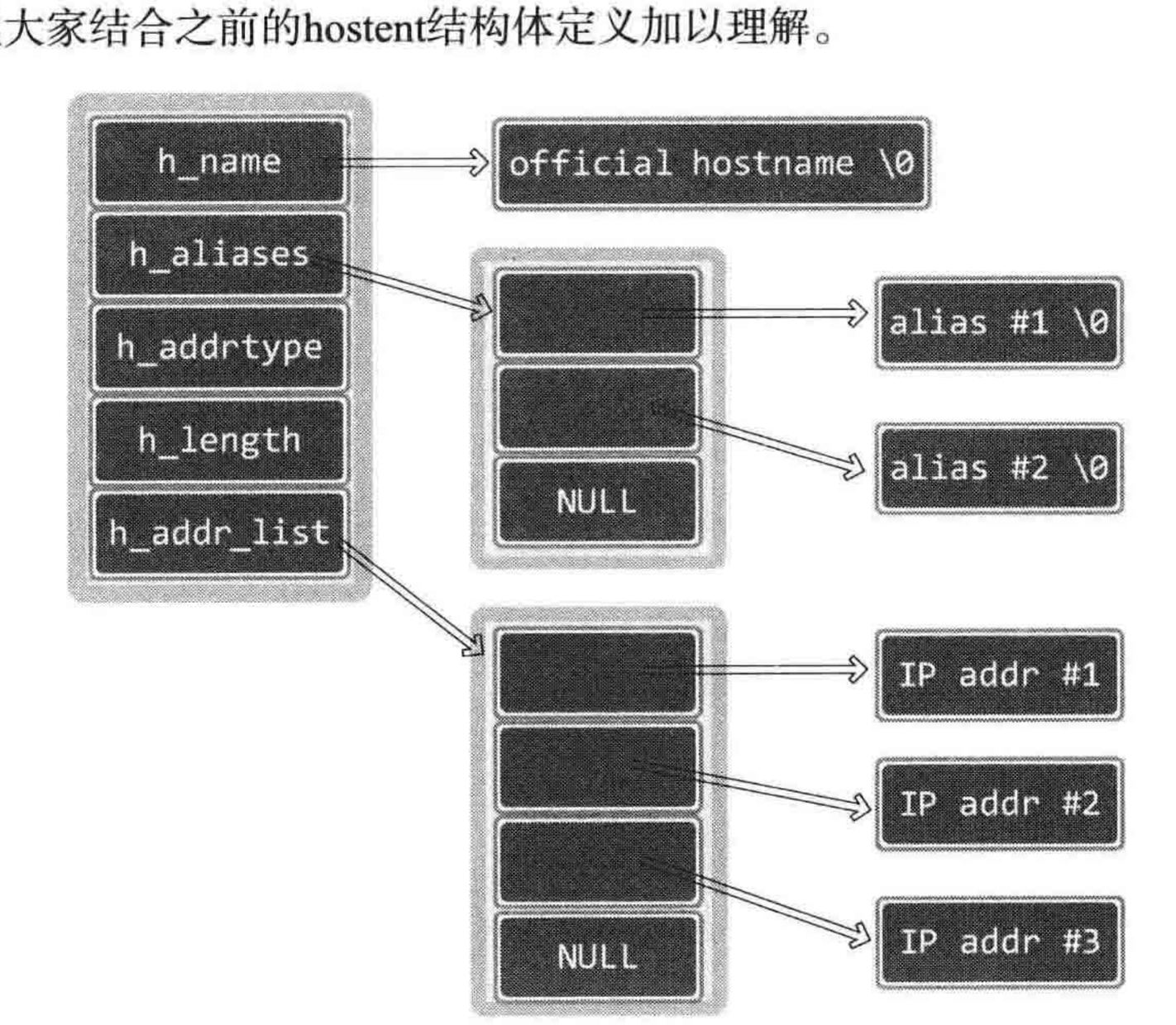



图8-2　hostent结构体变量

下列示例主要演示gethostbyname函数的应用，并说明hostent结构体变量的特性。

❖ gethostbyname.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <unistd.h>
4.  #include <arpa/inet.h>
5.  #include <netdb.h>
6.  void error_handling(char *message);
7.
8.  int main(int argc, char *argv[])
9.  {
10.     int i;
```

```
11.     struct hostent *host;
12.     if(argc!=2) {
13.         printf("Usage : %s <addr>\n", argv[0]);
14.         exit(1);
15.     }
16.
17.     host=gethostbyname(argv[1]);
18.     if(!host)
19.         error_handling("gethost... error");
20.
21.     printf("Official name: %s \n", host->h_name);
22.     for(i=0; host->h_aliases[i]; i++)
23.         printf("Aliases %d: %s \n", i+1, host->h_aliases[i]);
24.     printf("Address type: %s \n",
25.         (host->h_addrtype==AF_INET)?"AF_INET":"AF_INET6");
26.     for(i=0; host->h_addr_list[i]; i++)
27.         printf("IP addr %d: %s \n", i+1,
28.             inet_ntoa(*(struct in_addr*)host->h_addr_list[i]));
29.     return 0;
30. }
31.
32. void error_handling(char *message)
33. {
34.     fputs(message, stderr);
35.     fputc('\n', stderr);
36.     exit(1);
37. }
```

- 第17行：将通过main函数传递的字符串用作参数调用gethostbyname。
- 第21行：输出官方域名。
- 第22、23行：输出除官方域名以外的域名。这样编写循环语句的原因可从图8-2中找到答案。
- 第26~28行：输出IP地址信息。但多了令人感到困惑的类型转换。关于这一点稍后将给出说明。

❖ 运行结果：gethostbyname.c

```
root@my_linux:/tcpip# gcc gethostbyname.c -o hostname
root@my_linux:/tcpip# ./hostname www.naver.com
Official name: www.g.naver.com
Aliases 1: www.naver.com
Address type: AF_INET
IP addr 1: 202.131.29.70
IP addr 2: 222.122.195.6
```

我利用Naver网站域名运行了上述示例，大家可以任选一个域名验。现在讨论上述示例的第26~28行。若只看hostent结构体的定义，结构体成员h_addr_list指向字符串指针数组（由多个字符

串地址构成的数组）。但字符串指针数组中的元素实际指向的是（实际保存的是）in_addr结构体变量地址值而非字符串，如图8-3所示。

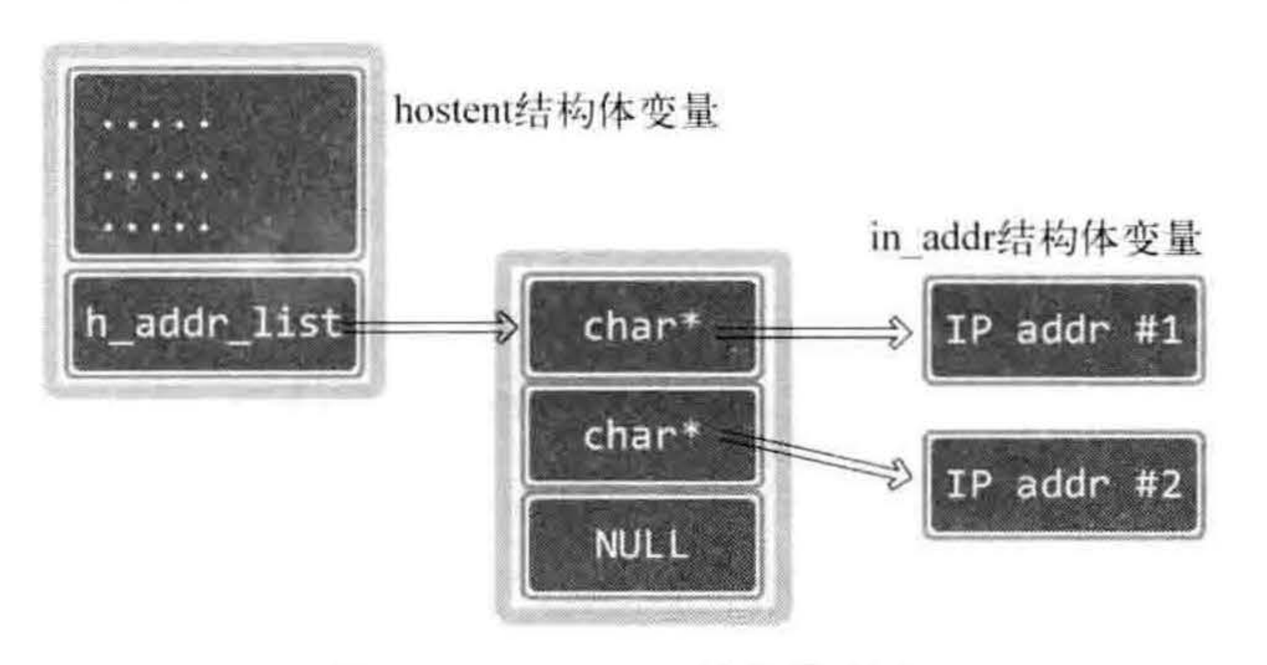



图8-3　h_addr_list结构体成员

图8-3给出了h_addr_list结构体的参照关系。正因如此，示例的第28行需要进行类型转换，并调用inet_ntoa函数。另外，in_addr结构体的声明可以参考第3章。

**提　示**

**为什么是 char* 而不是 in_addr***

hostent 结构体的成员 h_addr_list 指向的数组类型并不是 in_addr 结构体的指针数组，而是采用了 char 指针。各位也许对这一点感到困惑，但我认为大家应该能料到。hostent 结构体并非只为 IPv4 准备。h_addr_list 指向的数组中也可以保存 IPv6 地址信息。因此，考虑到通用性，声明为 char 指针类型的数组。

“声明为 void 指针类型是否更合理？”

若能想到这一点，说明对 C 语言掌握非常到位。当然如此。指针对象不明确时，更适合使用 void 指针类型。但各位目前学习的套接字相关函数都是在 void 指针标准化之前定义的，而当时无法明确指出指针类型时采用的是 char 指针。

## 利用 IP 地址获取域名

之前介绍的gethostbyname函数利用域名获取包括IP地址在内的域相关信息。而gethostbyaddr函数利用IP地址获取域相关信息。

```
#include <netdb.h>

struct hostent * gethostbyaddr(const char * addr, socklen_t len, int family);
```

➔ 成功时返回 hostent 结构体变量地址值，失败时返回 NULL 指针。

- addr　含有IP地址信息的in_addr结构体指针。为了同时传递IPv4地址之外的其他信息，该变量的类型声明为char指针。
- len　向第一个参数传递的地址信息的字节数，IPv4时为4，IPv6时为16。
- family　传递地址族信息，IPv4时为AF_INET，IPv6时为AF_INET6。

如果已经彻底掌握gethostbyname函数，那么上述函数理解起来并不难。下面通过示例演示该函数的使用方法。

❖ gethostbyaddr.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <unistd.h>
5.  #include <arpa/inet.h>
6.  #include <netdb.h>
7.  void error_handling(char *message);
8.
9.  int main(int argc, char *argv[])
10. {
11.     int i;
12.     struct hostent *host;
13.     struct sockaddr_in addr;
14.     if(argc!=2) {
15.         printf("Usage : %s <IP>\n", argv[0]);
16.         exit(1);
17.     }
18.
19.     memset(&addr, 0, sizeof(addr));
20.     addr.sin_addr.s_addr=inet_addr(argv[1]);
21.     host=gethostbyaddr((char*)&addr.sin_addr, 4, AF_INET);
22.     if(!host)
23.         error_handling("gethost... error");
24.
25.     printf("Official name: %s \n", host->h_name);
26.     for(i=0; host->h_aliases[i]; i++)
27.         printf("Aliases %d: %s \n", i+1, host->h_aliases[i]);
28.     printf("Address type: %s \n",
29.         (host->h_addrtype==AF_INET)?"AF_INET":"AF_INET6");
30.     for(i=0; host->h_addr_list[i]; i++)
31.         printf("IP addr %d: %s \n", i+1,
32.             inet_ntoa(*(struct in_addr*)host->h_addr_list[i]));
```

```
33.     return 0;
34. }
35.
36. void error_handling(char *message)
37. {
38.     fputs(message, stderr);
39.     fputc('\n', stderr);
40.     exit(1);
41. }
```

除第21行的gethostbyaddr函数调用过程外，与gethostbyname.c并无区别，因为函数调用的结果是通过hostent结构体变量地址值传递的。

❖ 运行结果：gethostbyaddr.c

```
root@my_linux:/tcpip# gcc gethostbyaddr.c -o hostaddr
root@my_linux:/tcpip# ./hostaddr 74.125.19.106
Official name: nuq04s01-in-f106.google.com
Address type: AF_INET
IP addr 1: 74.125.19.106
```

我通过ping命令得到了Google的IP地址，并利用此信息运行了示例。从运行结果可以看到，记录于DNS的官方主页地址具有特殊格式。

## 8.3　基于 Windows 的实现

Windows平台中也有类似功能的同名函数，因此无需经过太多变更。先介绍gethostbyname函数。

```
#include <winsock2.h>

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

➔ 成功时返回 hostent 结构体变量地址值，失败时返回 NULL 指针。

函数名、参数及返回类型与Linux中没有区别，故省略，继续介绍下一函数。

```
#include <winsock2.h>

struct hostent * gethostbyaddr(const char * addr, int len, int type);
```

➔ 成功时返回 hostent 结构体变量地址值，失败时返回 NULL 指针。

上述函数也与Linux中的函数完全一致，故省略，下面在示例中进行实际调用。

❖ gethostbyname_win.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <winsock2.h>
4.  void ErrorHandling(char *message);
5.
6.  int main(int argc, char *argv[])
7.  {
8.      WSADATA wsaData;
9.      int i;
10.     struct hostent *host;
11.     if(argc!=2) {
12.         printf("Usage : %s <addr>\n", argv[0]);
13.         exit(1);
14.     }
15.     if(WSAStartup(MAKEWORD(2, 2), &wsaData)!=0)
16.         ErrorHandling("WSAStartup() error!");
17.
18.     host=gethostbyname(argv[1]);
19.     if(!host)
20.         ErrorHandling("gethost... error");
21.
22.     printf("Official name: %s \n", host->h_name);
23.     for(i=0; host->h_aliases[i]; i++)
24.         printf("Aliases %d: %s \n", i+1, host->h_aliases[i]);
25.     printf("Address type: %s \n",
26.         (host->h_addrtype==AF_INET)?"AF_INET":"AF_INET6");
27.     for(i=0; host->h_addr_list[i]; i++)
28.         printf("IP addr %d: %s \n", i+1,
29.             inet_ntoa(*(struct in_addr*)host->h_addr_list[i]));
30.     WSACleanup();
31.     return 0;
32. }
33.
34. void ErrorHandling(char *message)
35. {
36.     fputs(message, stderr);
37.     fputc('\n', stderr);
38.     exit(1);
39. }
```

❖ gethostbyaddr_win.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <winsock2.h>
5.  void ErrorHandling(char *message);
```

```
6.
7.  int main(int argc, char *argv[])
8.  {
9.      WSADATA wsaData;
10.     int i;
11.     struct hostent *host;
12.     SOCKADDR_IN addr;
13.     if(argc!=2) {
14.         printf("Usage : %s <IP>\n", argv[0]);
15.         exit(1);
16.     }
17.     if(WSAStartup(MAKEWORD(2, 2), &wsaData)!=0)
18.         ErrorHandling("WSAStartup() error!");
19.
20.     memset(&addr, 0, sizeof(addr));
21.     addr.sin_addr.s_addr=inet_addr(argv[1]);
22.     host=gethostbyaddr((char*)&addr.sin_addr, 4, AF_INET);
23.     if(!host)
24.         ErrorHandling("gethost... error");
25.
26.     printf("Official name: %s \n", host->h_name);
27.     for(i=0; host->h_aliases[i]; i++)
28.         printf("Aliases %d: %s \n", i+1, host->h_aliases[i]);
29.     printf("Address type: %s \n",
30.         (host->h_addrtype==AF_INET)?"AF_INET":"AF_INET6");
31.     for(i=0; host->h_addr_list[i]; i++)
32.         printf("IP addr %d: %s \n", i+1,
33.             inet_ntoa(*(struct in_addr*)host->h_addr_list[i]));
34.     WSACleanup();
35.     return 0;
36. }
37.
38. void ErrorHandling(char *message)
39. {
40.     fputs(message, stderr);
41.     fputc('\n', stderr);
42.     exit(1);
43. }
```

各位可能也会认为没必要给出运行结果，故省略。基于Windows的实现相关讲解到此结束。

## 8.4　习题

(1) 下列关于DNS的说法错误的是?

a. 因为DNS存在，故可以用域名替代IP。

b. DNS服务器实际上是路由器，因为路由器根据域名决定数据路径。

c. 所有域名信息并非集中于1台DNS服务器，但可以获取某一DNS服务器中未注册的IP地址。

d. DNS服务器根据操作系统进行区分，Windows下的DNS服务器和Linux下的DNS服务器是不同的。

(2) 阅读如下对话，并说明东秀的解决方案是否可行。这些都是大家可以在大学计算机实验室验证的内容。

- 静洙："东秀吗？我们学校网络中使用的默认DNS服务器发生了故障，无法访问我要投简历的公司主页！有没有办法解决？"
- 东秀："网络连接正常，但DNS服务器发生了故障？"
- 静洙："恩！有没有解决方法？是不是要去周围的网吧？"
- 东秀："有那必要吗？我把我们学校的DNS服务器IP地址告诉你，你改一下你的默认DNS服务器地址。"
- 静洙："这样可以吗？默认DNS服务器必须连接到本地网络吧！"
- 东秀："不是！上次我们学校DNS服务器发生故障时，网管就给了我们其他DNS服务器的IP地址呢。"
- 静洙："那是因为你们学校有多台DNS服务器！"
- 东秀："是吗？你的话好像也有道理。那你快去网吧吧！"

(3) 在浏览器地址栏输入www.orentec.co.kr，并整理出主页显示过程。假设浏览器访问的默认DNS服务器中并没有关于www.orentec.co.kr的IP地址信息。

# 第9章 套接字的多种可选项

套接字具有多种特性，这些特性可通过可选项更改。本章将介绍更改套接字可选项的方法，并以此为基础进一步观察套接字内部。

## 9.1 套接字可选项和 I/O 缓冲大小

我们进行套接字编程时往往只关注数据通信，而忽略了套接字具有的不同特性。但是，理解这些特性并根据实际需要进行更改也十分重要。

### 套接字多种可选项

我们之前写的程序都是创建好套接字后（未经特别操作）直接使用的，此时通过默认的套接字特性进行数据通信。之前的示例较为简单，无需特别操作套接字特性，但有时的确需要更改。表9-1列出了一部分套接字可选项。

表9-1 可设置套接字的多种可选项

| 协议层 | 选项名 | 读取 | 设置 |
|---|---|---|---|
| SOL_SOCKET | SO_SNDBUF | O | O |
| | SO_RCVBUF | O | O |
| | SO_REUSEADDR | O | O |
| | SO_KEEPALIVE | O | O |
| | SO_BROADCAST | O | O |
| | SO_DONTROUTE | O | O |
| | SO_OOBINLINE | O | O |
| | SO_ERROR | O | X |
| | SO_TYPE | O | X |
| IPPROTO_IP | IP_TOS | O | O |
| | IP_TTL | O | O |
| | IP_MULTICAST_TTL | O | O |
| | IP_MULTICAST_LOOP | O | O |
| | IP_MULTICAST_IF | O | O |

（续）

| 协议层 | 选项名 | 读取 | 设置 |
|---|---|---|---|
| IPPROTO_TCP | TCP_KEEPALIVE | O | O |
| | TCP_NODELAY | O | O |
| | TCP_MAXSEG | O | O |

从表9-1中可以看出，套接字可选项是分层的。IPPROTO_IP层可选项是IP协议相关事项，IPPROTO_TCP层可选项是TCP协议相关的事项，SOL_SOCKET层是套接字相关的通用可选项。

也许有人看到表格会产生畏惧感，但现在无需全部背下来或理解，因此不必有负担。实际能够设置的可选项数量是表9-1的好几倍，也无需一下子理解所有可选项，实际工作中逐一掌握即可。接触的可选项多了，自然会掌握大部分重要的。本书也只介绍其中一部分重要的可选项含义及更改方法。

## getsockopt & setsockopt

我们几乎可以针对表9-1中的所有可选项进行读取（Get）和设置（Set）（当然，有些可选项只能进行一种操作）。可选项的读取和设置通过如下2个函数完成。

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

int getsockopt(int sock, int level, int optname, void *optval, socklen_t *optlen);
```

➔ 成功时返回0，失败时返回-1。

- sock　用于查看选项套接字文件描述符。
- level　要查看的可选项的协议层。
- optname　要查看的可选项名。
- optval　保存查看结果的缓冲地址值。
- optlen　向第四个参数optval传递的缓冲大小。调用函数后，该变量中保存通过第四个参数返回的可选项信息的字节数。

上述函数用于读取套接字可选项，并不难。接下来介绍更改可选项时调用的函数。

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

int setsockopt(int sock, int level, int optname, const void *optval, socklen_t optlen);
```

➔ 成功时返回0，失败时返回-1。

- sock　用于更改可选项的套接字文件描述符。
- level　要更改的可选项协议层。
- optname　要更改的可选项名。
- optval　保存要更改的选项信息的缓冲地址值。
- optlen　向第四个参数optval传递的可选项信息的字节数。

接下来介绍这些函数的调用方法。关于setsockopt函数的调用方法在其他示例中给出，先介绍getsockopt函数的调用方法。下列示例用协议层为SOL_SOCKET、名为SO_TYPE的可选项查看套接字类型（TCP或UDP）。

❖ sock_type.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <unistd.h>
4.  #include <sys/socket.h>
5.  void error_handling(char *message);
6.
7.  int main(int argc, char *argv[])
8.  {
9.      int tcp_sock, udp_sock;
10.     int sock_type;
11.     socklen_t optlen;
12.     int state;
13.
14.     optlen=sizeof(sock_type);
15.     tcp_sock=socket(PF_INET, SOCK_STREAM, 0);
16.     udp_sock=socket(PF_INET, SOCK_DGRAM, 0);
17.     printf("SOCK_STREAM: %d \n", SOCK_STREAM);
18.     printf("SOCK_DGRAM: %d \n", SOCK_DGRAM);
19.
20.     state=getsockopt(tcp_sock, SOL_SOCKET, SO_TYPE, (void*)&sock_type, &optlen);
21.     if(state)
22.         error_handling("getsockopt() error!");
23.     printf("Socket type one: %d \n", sock_type);
24.
25.     state=getsockopt(udp_sock, SOL_SOCKET, SO_TYPE, (void*)&sock_type, &optlen);
26.     if(state)
27.         error_handling("getsockopt() error!");
28.     printf("Socket type two: %d \n", sock_type);
29.     return 0;
30. }
31.
32. void error_handling(char *message)
33. {
34.     fputs(message, stderr);
35.     fputc('\n', stderr);
36.     exit(1);
37. }
```

- 第15、16行：分别生成TCP、UDP套接字。
- 第17、18行：输出创建TCP、UDP套接字时传入的SOCK_STREAM、SOCK_DGRAM。
- 第20、25行：获取套接字类型信息。如果是TCP套接字，将获得SOCK_STREAM常数值1；如果是UDP套接字，则获得SOCK_DGRAM的常数值2。

❖ 运行结果：sock_type.c

```
root@my_linux:/tcpip# gcc sock_type.c -o socktype
root@my_linux:/tcpip# ./socktype
SOCK_STREAM: 1
SOCK_DGRAM: 2
Socket type one: 1
Socket type two: 2
```

上述示例给出了调用getsockopt函数查看套接字信息的方法。另外，用于验证套接字类型的SO_TYPE是典型的只读可选项，这一点可以通过下面这句话解释：

> “套接字类型只能在创建时决定，以后不能再更改。”

## SO_SNDBUF & SO_RCVBUF

前面介绍过，创建套接字将同时生成I/O缓冲。如果各位忘了这部分内容，可以复习第5章。接下来将介绍I/O缓冲相关可选项。

SO_RCVBUF是输入缓冲大小相关可选项，SO_SNDBUF是输出缓冲大小相关可选项。用这2个可选项既可以读取当前I/O缓冲大小，也可以进行更改。通过下列示例读取创建套接字时默认的I/O缓冲大小。



❖ get_buf.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <unistd.h>
4.  #include <sys/socket.h>
5.  void error_handling(char *message);
6.
7.  int main(int argc, char *argv[])
8.  {
9.      int sock;
10.     int snd_buf, rcv_buf, state;
11.     socklen_t len;
12.
13.     sock=socket(PF_INET, SOCK_STREAM, 0);
14.     len=sizeof(snd_buf);
15.     state=getsockopt(sock, SOL_SOCKET, SO_SNDBUF, (void*)&snd_buf, &len);
```

```
16.     if(state)
17.         error_handling("getsockopt() error");
18.
19.     len=sizeof(rcv_buf);
20.     state=getsockopt(sock, SOL_SOCKET, SO_RCVBUF, (void*)&rcv_buf, &len);
21.     if(state)
22.         error_handling("getsockopt() error");
23.
24.     printf("Input buffer size: %d \n", rcv_buf);
25.     printf("Output buffer size: %d \n", snd_buf);
26.     return 0;
27. }
28.
29. void error_handling(char *message)
30. {
31.     fputs(message, stderr);
32.     fputc('\n', stderr);
33.     exit(1);
34. }
```

❖ 运行结果：get_buf.c

```
root@my_linux:/tcpip# gcc get_buf.c -o getbuf
root@my_linux:/tcpip# ./getbuf
Input buffer size: 87380
Output buffer size: 16384
```

这是我系统中的运行结果，与各位的运行结果相比可能有较大差异。接下来的程序中将更改I/O缓冲大小。

❖ set_buf.c

```
1.  #include <"头声明与get_buf.c一致，故省略。">
2.  void error_handling(char *message);
3.
4.  int main(int argc, char *argv[])
5.  {
6.      int sock;
7.      int snd_buf=1024*3, rcv_buf=1024*3;
8.      int state;
9.      socklen_t len;
10.
11.     sock=socket(PF_INET, SOCK_STREAM, 0);
12.     state=setsockopt(sock, SOL_SOCKET, SO_RCVBUF, (void*)&rcv_buf, sizeof(rcv_buf));
13.     if(state)
14.         error_handling("setsockopt() error!");
15.
16.     state=setsockopt(sock, SOL_SOCKET, SO_SNDBUF, (void*)&snd_buf, sizeof(snd_buf));
17.     if(state)
18.         error_handling("setsockopt() error!");
```

```
19.
20.     len=sizeof(snd_buf);
21.     state=getsockopt(sock, SOL_SOCKET, SO_SNDBUF, (void*)&snd_buf, &len);
22.     if(state)
23.         error_handling("getsockopt() error!");
24.
25.     len=sizeof(rcv_buf);
26.     state=getsockopt(sock, SOL_SOCKET, SO_RCVBUF, (void*)&rcv_buf, &len);
27.     if(state)
28.         error_handling("getsockopt() error!");
29.
30.     printf("Input buffer size: %d \n", rcv_buf);
31.     printf("Output buffer size: %d \n", snd_buf);
32.     return 0;
33. }
34.
35. void error_handling(char *message)
36. {
37.     fputs(message, stderr);
38.     fputc('\n', stderr);
39.     exit(1);
40. }
```

- 第12、16行：I/O缓冲大小更改为3M字节。
- 第21、26行：为了验证I/O缓冲的更改，读取缓冲大小。

❖ 运行结果：set_buf.c

```
root@my_linux:/tcpip# gcc set_buf.c -o setbuf
root@my_linux:/tcpip# ./setbuf
Input buffer size: 6144
Output buffer size: 6144
```



输出结果跟我们预想的完全不同，但也算合理。缓冲大小的设置需谨慎处理，因此不会完全按照我们的要求进行，只是通过调用setsockopt函数向系统传递我们的要求。如果把输出缓冲设置为0并如实反映这种设置，TCP协议将如何进行？如果要实现流控制和错误发生时的重传机制，至少要有一些缓冲空间吧？上述示例虽没有100%按照我们的请求设置缓冲大小，但也大致反映出了通过setsockopt函数设置的缓冲大小。

## 9.2 SO_REUSEADDR

本节的可选项SO_REUSEADDR及其相关的Time-wait状态很重要，希望大家务必理解并掌握。

## 发生地址分配错误（Binding Error）

学习SO_REUSEADDR可选项之前，应理解好Time-wait状态。我们读完下列示例后再讨论后续内容。

❖ reuseadr_eserver.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 TRUE 1
9.  #define FALSE 0
10. void error_handling(char *message);
11.
12. int main(int argc, char *argv[])
13. {
14.     int serv_sock, clnt_sock;
15.     char message[30];
16.     int option, str_len;
17.     socklen_t optlen, clnt_adr_sz;
18.     struct sockaddr_in serv_adr, clnt_adr;
19.     if(argc!=2) {
20.         printf("Usage : %s <port>\n", argv[0]);
21.         exit(1);
22.     }
23.
24.     serv_sock=socket(PF_INET, SOCK_STREAM, 0);
25.     if(serv_sock==-1)
26.         error_handling("socket() error");
27.     /*
28.     optlen=sizeof(option);
29.     option=TRUE;
30.     setsockopt(serv_sock, SOL_SOCKET, SO_REUSEADDR, (void*)&option, optlen);
31.     */
32.
33.     memset(&serv_adr, 0, sizeof(serv_adr));
34.     serv_adr.sin_family=AF_INET;
35.     serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
36.     serv_adr.sin_port=htons(atoi(argv[1]));
37.
38.     if(bind(serv_sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr)))
39.         error_handling("bind() error");
40.     if(listen(serv_sock, 5)==-1)
41.         error_handling("listen error");
42.     clnt_adr_sz=sizeof(clnt_adr);
43.     clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_adr,&clnt_adr_sz);
44.
```

```
45.     while((str_len=read(clnt_sock,message, sizeof(message)))!= 0)
46.     {
47.         write(clnt_sock, message, str_len);
48.         write(1, message, str_len);
49.     }
50.     close(clnt_sock);
51.     close(serv_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. }
```

此示例是之前已实现过多次的回声服务器端，可以结合第4章介绍过的回声客户端运行。下面运行该示例，第28～30行应保持注释状态。通过如下方式终止程序：

“在客户端控制台输入Q消息，或通过CTRL+C终止程序。”

也就是说，让客户端先通知服务器端终止程序。在客户端控制台输入Q消息时调用close函数（参考第4章的echo_client.c），向服务器端发送FIN消息并经过四次握手过程。当然，输入CTRL+C时也会向服务器传递FIN消息。强制终止程序时，由操作系统关闭文件及套接字，此过程相当于调用close函数，也会向服务器端传递FIN消息。

“但看不到什么特殊现象啊？”

是的，通常都是由客户端先请求断开连接，所以不会发生特别的事情。重新运行服务器端也不成问题，但按照如下方式终止程序时则不同。

“服务器端和客户端已建立连接的状态下，向服务器端控制台输入CTRL+C，即强制关闭服务器端。”

这主要模拟了服务器端向客户端发送FIN消息的情景。但如果以这种方式终止程序，那服务器端重新运行时将产生问题。如果用同一端口号重新运行服务器端，将输出“bind() error”消息，并且无法再次运行。但在这种情况下，再过大约3分钟即可重新运行服务器端。

上述2种运行方式唯一的区别就是谁先传输FIN消息，但结果却迥然不同，原因何在呢？

## Time-wait 状态

相信各位已对四次握手有了很好的理解，先观察该过程，如图9-1所示。

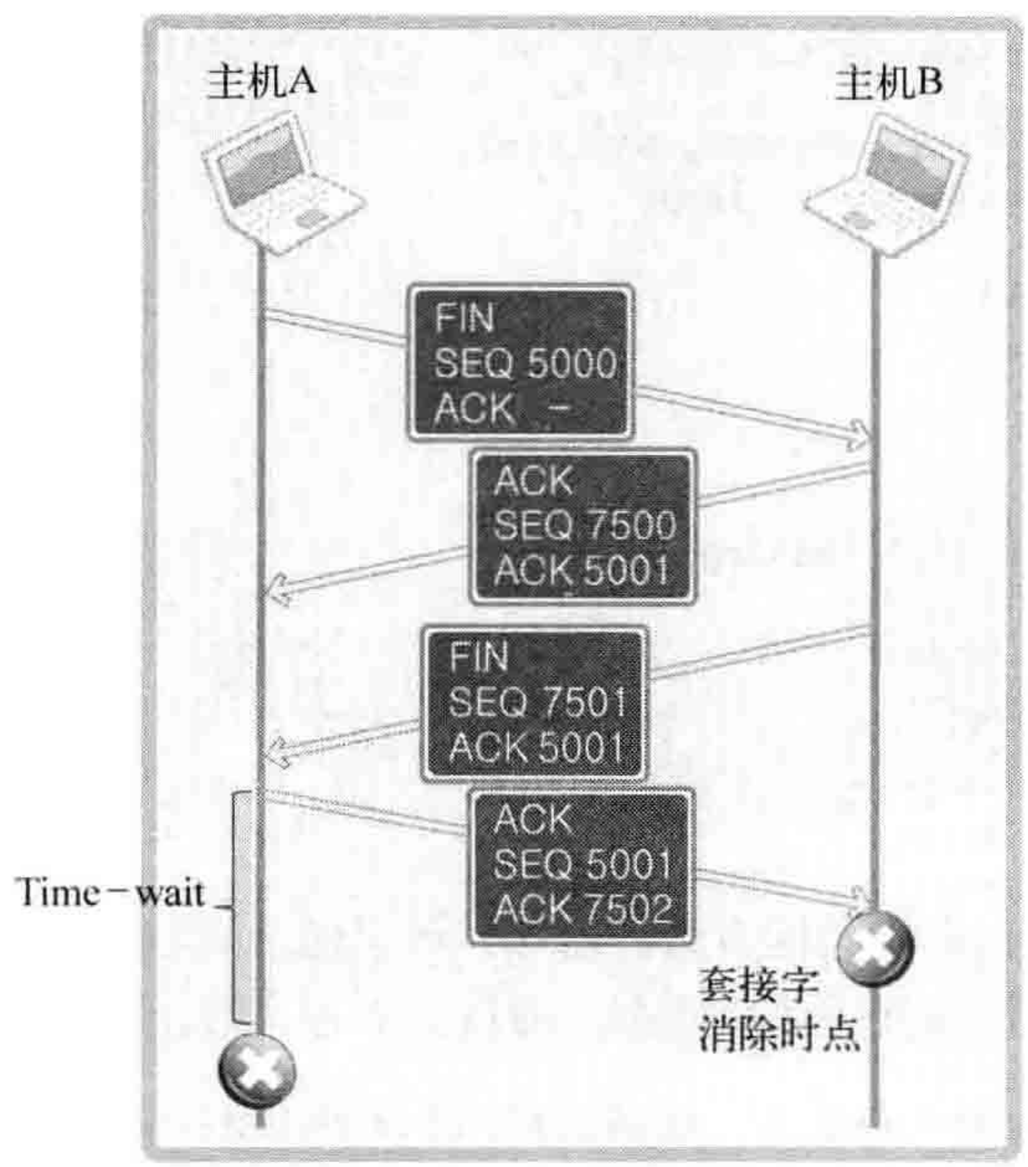



图9-1　Time-wait状态下的套接字

假设图9-1中主机A是服务器端，因为是主机A向B发送FIN消息，故可以想象成服务器端在控制台输入CTRL+C。但问题是，套接字经过四次握手过程后并非立即消除，而是要经过一段时间的Time-wait状态。当然，只有先断开连接的（先发送FIN消息的）主机才经过Time-wait状态。因此，若服务器端先断开连接，则无法立即重新运行。套接字处在Time-wait过程时，相应端口是正在使用的状态。因此，就像之前验证过的，bind函数调用过程中当然会发生错误。

**提　示**

**客户端套接字不会经过 Time-wait 过程吗？**

有些人会误以为 Time-wait 过程只存在于服务器端。但实际上，不管是服务器端还是客户端，套接字都会有 Time-wait 过程。先断开连接的套接字必然会经过 Time-wait 过程。但无需考虑客户端 Time-wait 状态。因为客户端套接字的端口号是任意指定的。与服务器端不同，客户端每次运行程序时都会动态分配端口号，因此无需过多关注 Time-wait 状态。

到底为什么会有Time-wait状态呢？图9-1中假设主机A向主机B传输ACK消息（SEQ 5001、ACK 7502）后立即消除套接字。但最后这条ACK消息在传递途中丢失，未能传给主机B。这时会发生什么？主机B会认为之前自己发送的FIN消息（SEQ 7501、ACK 5001）未能抵达主机A，继而试图重传。但此时主机A已是完全终止的状态，因此主机B永远无法收到从主机A最后传来的ACK消息。相反，若主机A的套接字处在Time-wait状态，则会向主机B重传最后的ACK消息，主

机B也可以正常终止。基于这些考虑，先传输FIN消息的主机应经过Time-wait过程。

## 地址再分配

Time-wait看似重要，但并不一定讨人喜欢。考虑一下系统发生故障从而紧急停止的情况。这时需要尽快重启服务器端以提供服务，但因处于Time-wait状态而必须等待几分钟。因此，Time-wait并非只有优点，而且有些情况下可能引发更大问题。图9-2演示了四次握手时不得不延长Time-wait过程的情况。

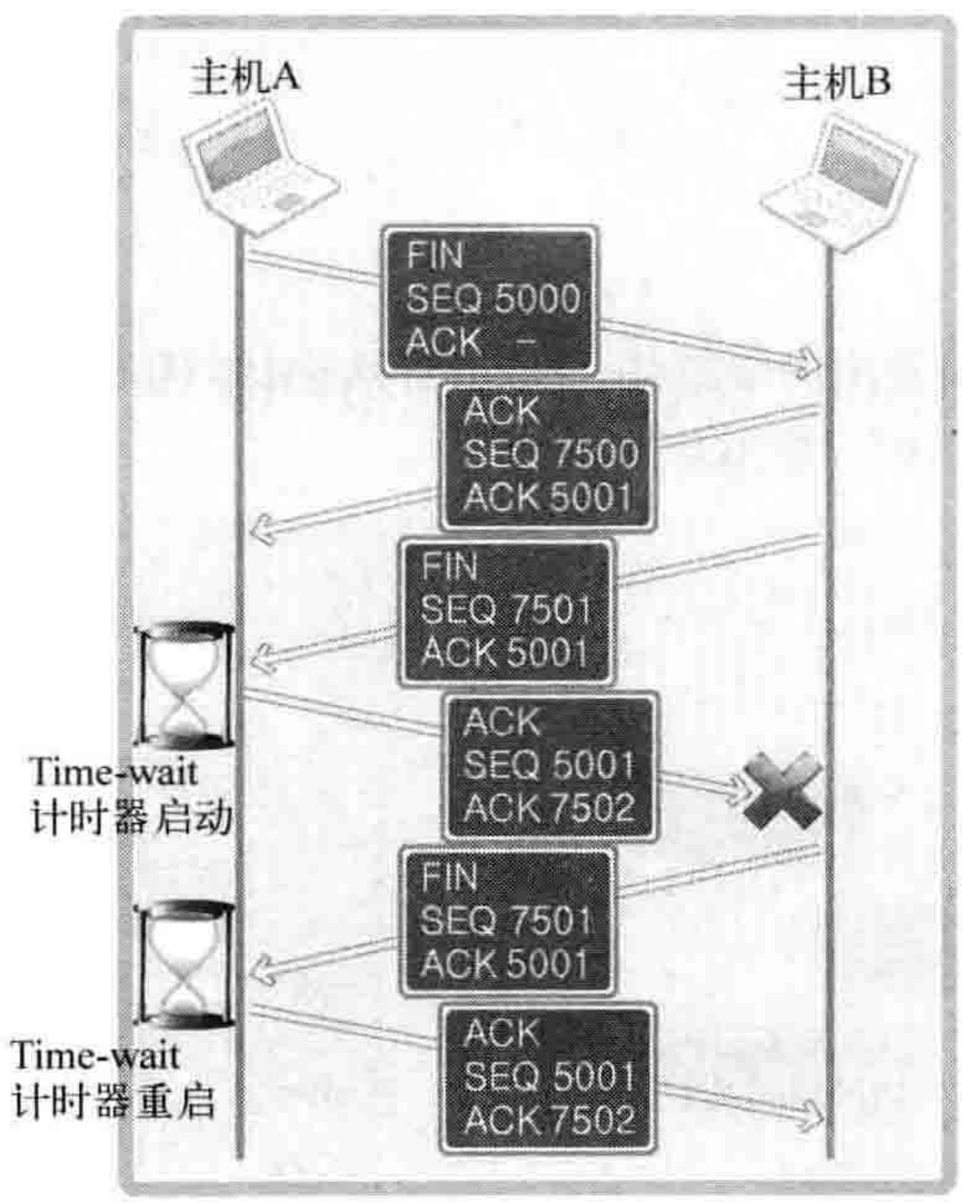



图9-2 重启Time-wait计时器

如图9-2所示，在主机A的四次握手过程中，如果最后的数据丢失，则主机B会认为主机A未能收到自己发送的FIN消息，因此重传。这时，收到FIN消息的主机A将重启Time-wait计时器。因此，如果网络状况不理想，Time-wait状态将持续。

解决方案就是在套接字的可选项中更改SO_REUSEADDR的状态。适当调整该参数，可将Time-wait状态下的套接字端口号重新分配给新的套接字。SO_REUSEADDR的默认值为0（假），这就意味着无法分配Time-wait状态下的套接字端口号。因此需要将这个值改成1（真）。具体做法已在示例reuseadr_eserver.c中给出，只需去掉下述代码的注释即可。

```
optlen=sizeof(option);
option=TRUE;
setsockopt(serv_sock, SOL_SOCKET, SO_REUSEADDR, (void*) &option, optlen);
```

各位是否去掉了注释？既然服务器端reuseadr_eserver.c已变成可随时运行的状态，希望大家在Time-wait状态下验证其能否重新运行。

## 9.3 TCP_NODELAY

我教Java网络编程时，经常被问及如下问题：

“什么是Nagle算法？使用该算法能够获得哪些数据通信特性？”

我被问到这个问题时感到特别高兴，因为开发人员容易忽视的一个问题就是Nagle算法，下面进行详细讲解。

### Nagle 算法

为防止因数据包过多而发生网络过载，Nagle算法在1984年诞生了。它应用于TCP层，非常简单。其使用与否会导致如图9-3所示差异。

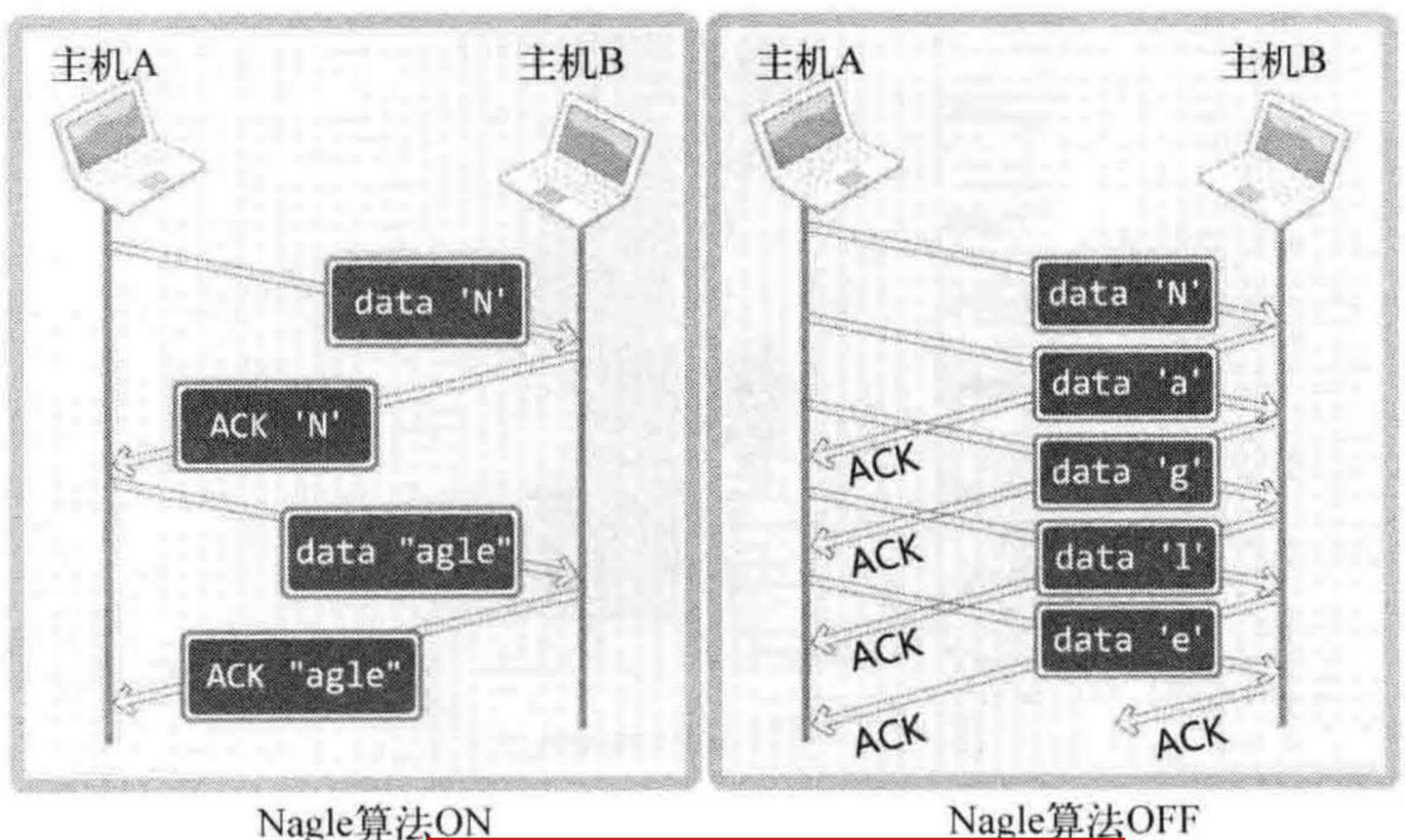



图9-3　Nagle算法

图9-3展示了通过Nagle算法发送字符串“Nagle”和未使用Nagle算法的差别。可以得到如下结论：

“只有收到前一数据的ACK消息时，Nagle算法才发送下一数据。”

TCP套接字默认使用Nagle算法交换数据，因此最大限度地进行缓冲，直到收到ACK。图9-3左侧正是这种情况。为了发送字符串“Nagle”，将其传递到输出缓冲。这时头字符“N”之前没有其他数据（没有需接收的ACK），因此立即传输。之后开始等待字符“N”的ACK消息，等待过程中，剩下的“agle”填入输出缓冲。接下来，收到字符“N”的ACK消息后，将输出缓冲的

“agle”装入一个数据包发送。也就是说，共需传递4个数据包以传输1个字符串。

接下来分析未使用Nagle算法时发送字符串“Nagle”的过程。假设字符“N”到“e”依序传到输出缓冲。此时的发送过程与ACK接收与否无关，因此数据到达输出缓冲后将立即被发送出去。从图9-3右侧可以看到，发送字符串“Nagle”时共需10个数据包。由此可知，不使用Nagle算法将对网络流量（Traffic: 指网络负载或混杂程度）产生负面影响。即使只传输1个字节的数据，其头信息都有可能是几十个字节。因此，为了提高网络传输效率，必须使用Nagle算法。

图 9-3 是极端情况的演示

在程序中将字符串传给输出缓冲时并不是逐字传递的，故发送字符串“Nagle”的实际情况并非如图 9-3 所示。但如果隔一段时间再把构成字符串的字符传到输出缓冲（如果存在此类数据传递）的话，则有可能产生类似图 9-3 的情况。图 9-3 中就是隔一段时间向输出缓冲传递待发送数据的。

但Nagle算法并不是什么时候都适用。根据传输数据的特性，网络流量未受太大影响时，不使用Nagle算法要比使用它时传输速度快。最典型的是“传输大文件数据”。将文件数据传入输出缓冲不会花太多时间，因此，即便不使用Nagle算法，也会在装满输出缓冲时传输数据包。这不仅不会增加数据包的数量，反而会在无需等待ACK的前提下连续传输，因此可以大大提高传输速度。

一般情况下，不适用Nagle算法可以提高传输速度。但如果无条件放弃使用Nagle算法，就会增加过多的网络流量，反而会影响传输。因此，未准确判断数据特性时不应禁用Nagle算法。

## 禁用 Nagle 算法



刚才说过的“大文件数据”应禁用Nagle算法。换言之，如果有必要，就应禁用Nagle算法。

“Nagle算法使用与否在网络流量上差别不大，使用Nagle算法的传输速度更慢”

禁用方法非常简单。从下列代码也可看出，只需将套接字可选项TCP_NODELAY改为1（真）即可。

```
int opt_val=1;
setsockopt(sock, IPPROTO_TCP, TCP_NODELAY, (void *) &opt_val, sizeof(opt_val));
```

可以通过TCP_NODELAY的值查看Nagle算法的设置状态。

```
int opt_val;
socklen_t opt_len;
```

```
opt_len=sizeof(opt_val);
getsockopt(sock, IPPROTO_TCP, TCP_NODELAY, (void*) &opt_val, &opt_len);
```

如果正在使用Nagle算法，opt_val变量中会保存0；如果已禁用Nagle算法，则保存1。

## 9.4　基于 Windows 的实现

套接字可选项及其相关内容与操作系统无关，特别是本章的可选项，它们是TCP套接字的相关内容，因此在Windows平台与Linux平台下并无区别。接下来介绍更改和读取可选项的2个函数。

```
#include <winsock2.h>

int getsockopt(SOCKET sock, int level, int optname, char * optval, int * optlen);
```

➔ 成功时返回 0，失败时返回 SOCKET_ERROR。

- sock　要查看可选项的套接字句柄。
- level　要查看的可选项协议层。
- optname　要查看的可选项名。
- optval　保存查看结果的缓冲地址值。
- optlen　向第四个参数optval传递的缓冲大小。调用结束后，该变量中保存通过第四个参数返回的可选项字节数。

可以看到，除了optval类型变成char指针外，与Linux中的getsockopt函数相比并无太大区别（Linux中是void型指针）。将Linux中的示例移植到Windows时，应做适当的类型转换。接下来给出setsockopt函数。

```
#include <winsock2.h>

int setsockopt(SOCKET sock, int level, int optname, const char* optval, int
optlen);
```

➔ 成功时返回 0，失败时返回 SOCKET_ERROR。

- sock　要更改可选项的套接字句柄。
- level　要更改的可选项协议层。
- optname　要更改的可选项名。
- optval　保存要更改的可选项信息的缓冲地址值。
- optlen　传入第四个参数optval的可选项信息的字节数。

setsockopt函数也与Linux版的毫无二致。各位应更关注可选项的含义而非设置方法。最后，利用上述2个函数编写示例。之前在Linux中验证过套接字I/O缓冲大小，现将其改成基于Windows的实现。

❖ buf_win.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <winsock2.h>
5.  void ErrorHandling(char *message);
6.  void ShowSocketBufSize(SOCKET sock);
7.
8.  int main(int argc, char *argv[])
9.  {
10.     WSADATA wsaData;
11.     SOCKET hSock;
12.     int sndBuf, rcvBuf, state;
13.     if(WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
14.         ErrorHandling("WSAStartup() error!");
15.
16.     hSock=socket(PF_INET, SOCK_STREAM, 0);
17.     ShowSocketBufSize(hSock);
18.
19.     sndBuf=1024*3, rcvBuf=1024*3;
20.     state=setsockopt(hSock, SOL_SOCKET, SO_SNDBUF, (char*)&sndBuf, sizeof(sndBuf));
21.     if(state==SOCKET_ERROR)
22.         ErrorHandling("setsockopt() error!");
23.
24.     state=setsockopt(hSock, SOL_SOCKET, SO_RCVBUF, (char*)&rcvBuf, sizeof(rcvBuf));
25.     if(state==SOCKET_ERROR)
26.         ErrorHandling("setsockopt() error!");
27.
28.     ShowSocketBufSize(hSock);
29.     closesocket(hSock);
30.     WSACleanup();
31.     return 0;
32. }
33.
34. void ShowSocketBufSize(SOCKET sock)
35. {
36.     int sndBuf, rcvBuf, state, len;
37.
38.     len=sizeof(sndBuf);
39.     state=getsockopt(sock, SOL_SOCKET, SO_SNDBUF, (char*)&sndBuf, &len);
40.     if(state==SOCKET_ERROR)
41.         ErrorHandling("getsockopt() error");
42.
43.     len=sizeof(rcvBuf);
44.     state=getsockopt(sock, SOL_SOCKET, SO_RCVBUF, (char*)&rcvBuf, &len);
45.     if(state==SOCKET_ERROR)
46.         ErrorHandling("getsockopt() error");
```

```
47.
48.     printf("Input buffer size: %d \n", rcvBuf);
49.     printf("Output buffer size: %d \n", sndBuf);
50. }
51.
52. void ErrorHandling(char *message)
53. {
54.     fputs(message, stderr);
55.     fputc('\n', stderr);
56.     exit(1);
57. }
```

❖ 运行结果：buf_win.c

```
Input buffer size: 8192
Output buffer size: 8192
Input buffer size: 3072
Output buffer size: 3072
```

系统不同可能导致不同结果。但可以通过上述示例获取系统默认I/O缓冲大小，同时也可以得到更改之后实际使用的缓冲大小。

## 9.5　习题

(1) 下列关于Time-wait状态的说法错误的是？

a. Time-wait状态只在服务器端的套接字中发生。

b. 断开连接的四次握手过程中，先传输FIN消息的套接字将进入Time-wait状态。

c. Time-wait状态与断开连接的过程无关，而与请求连接过程中SYN消息的传输顺序有关。

d. Time-wait状态通常并非必要，应尽可能通过更改套接字可选项防止其发生。

(2) TCP_NODELAY可选项与Nagle算法有关，可通过它禁用Nagle算法。请问何时应考虑禁用Nagle算法？结合收发数据的特性给出说明。

# 第10章 多进程服务器端

大家已对套接字编程有了一定的理解，但要想实现真正的服务器端，只凭这些内容还不够。因此，现在开始学习构建实际网络服务所需内容。

## 10.1 进程概念及应用

利用之前学习到的内容，我们可以构建按序向第一个客户端到第一百个客户端提供服务的服务器端。当然，第一个客户端不会抱怨服务器端，但如果每个客户端的平均服务时间为0.5秒，则第100个客户端会对服务器端产生相当大的不满。

### 两种类型的服务器端

如果真正为客户端着想，应提高客户端满意度平均标准。如果有下面这种类型的服务器端，各位应该感到满意了吧。

"第一个连接请求的受理时间为0秒，第50个连接请求的受理时间为50秒，第100个连接请求的受理时间为100秒！但只要受理，服务只需1秒钟。"

如果排在前面的请求数能用一只手数清，客户端当然会对服务器端感到满意。但只要超过这个数，客户端就会开始抱怨。还不如用下面这种方式提供服务。

"所有连接请求的受理时间不超过1秒，但平均服务时间为2~3秒。"

大家无需过多考虑到底哪种服务器端好一些，只需假设收看网络视频课程，而且其顺序是第100位，就能得出结论。因此，接下来讨论如何提高客户端满意度平均标准。

### 并发服务器端的实现方法

即使有可能延长服务时间，也有必要改进服务器端，使其同时向所有发起请求的客户端提供

服务，以提高平均满意度。而且，网络程序中数据通信时间比CPU运算时间占比更大，因此，向多个客户端提供服务是一种有效利用CPU的方式。接下来讨论同时向多个客户端提供服务的并发服务器端。下面列出的是具有代表性的并发服务器端实现模型和方法。

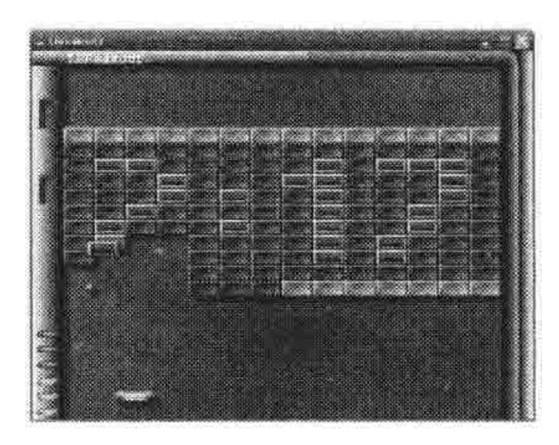
- 多进程服务器：通过创建多个进程提供服务。
- 多路复用服务器：通过捆绑并统一管理I/O对象提供服务。
- 多线程服务器：通过生成与客户端等量的线程提供服务。

先讲解第一种方法：多进程服务器。这种方法不适合在Windows平台下（Windows不支持）讲解，因此将重点放在Linux平台。若各位不太关心基于Linux的实现，可以直接跳到第12章。不过还是希望大家尽可能浏览一下，因为本章内容有助于理解服务器端构建方法。

## 理解进程（Process）

接下来了解多进程服务器实现的重点内容——进程，其定义如下：

"占用内存空间的正在运行的程序"



假如各位从网上下载了LBreakout游戏并安装到硬盘。此时的游戏并非进程，而是程序。因为游戏并未进入运行状态。下面开始运行程序。此时游戏被加载到主内存并进入运行状态，这时才可称为进程。如果同时运行多个LBreakout程序，则会生成相应数量的进程，也会占用相应进程数的内存空间。

再举个例子。假设各位需要进行文档相关操作，这时应打开文档编辑软件。如果工作的同时还想听音乐，应打开MP3播放器。另外，为了与朋友聊天，再打开MSN软件。此时共创建3个进程。从操作系统的角度看，进程是程序流的基本单位，若创建多个进程，则操作系统将同时运行。有时一个程序运行过程中也会产生多个进程。接下来要创建的多进程服务器就是其中的代表。编写服务器端前，先了解一下通过程序创建进程的方法。

**提示**

**CPU 核的个数与进程数**

拥有 2 个运算设备的 CPU 称作双核（Daul）CPU，拥有 4 个运算器的 CPU 称作 4 核（Quad）CPU。也就是说，1 个 CPU 中可能包含多个运算设备（核）。核的个数与可同时运行的进程数相同。相反，若进程数超过核数，进程将分时使用 CPU 资源。但因为 CPU 运转速度极快，我们会感到所有进程同时运行。当然，核数越多，这种感觉越明显。

## 进程 ID

讲解创建进程方法前，先简要说明进程ID。无论进程是如何创建的，所有进程都会从操作系统分配到ID。此ID称为“进程ID”，其值为大于2的整数。1要分配给操作系统启动后的（用于协助操作系统）首个进程，因此用户进程无法得到ID值1。接下来观察Linux中正在运行的进程。

❖ 运行结果：ps 命令语句

```
root@my_linux:/tcpip# ps au
USER     PID   %CPU %MEM   VSZ   RSS TTY      STAT  START  TIME  COMMAND
root     4400  0.0  0.1   1780   524 tty4     Ss+   15:57  0:00  /sbin/getty 384
root     4401  0.0  0.1   1780   524 tty5     Ss+   15:57  0:00  /sbin/getty 384
root     4408  0.0  0.1   1780   520 tty2     Ss+   15:57  0:00  /sbin/getty 384
root     4409  0.0  0.1   1780   524 tty3     Ss+   15:57  0:00  /sbin/getty 384
root     4410  0.0  0.1   1780   524 tty6     Ss+   15:57  0:00  /sbin/getty 384
root     5155  2.2  2.9  23728 15136 tty7     Rs+   15:57  0:41  /usr/X11R6/bin/
root     5320  0.0  0.1   1780   528 tty1     Ss+   15:58  0:00  /sbin/getty 384
swyoon   6926  0.2  0.5   5736  3028 pts/0    Ss    16:27  0:00  bash
root     6946  0.0  0.2   4128  1532 pts/0    S     16:28  0:00  su
root     6952  0.0  0.3   4312  1808 pts/0    R     16:28  0:00  bash
swyoon   7006  0.3  0.5   5788  3084 pts/1    Ss+   16:28  0:00  bash
root     7033  0.0  0.1   3136  1008 pts/0    R+    16:28  0:00  ps au
```

可以看出，通过ps指令可以查看当前运行的所有进程。特别需要注意的是，该命令同时列出了PID（进程ID）。另外，上述示例通过指定a和u参数列出了所有进程详细信息。

## 通过调用 fork 函数创建进程

创建进程的方法很多，此处只介绍用于创建多进程服务器端的fork函数。

```
#include <unistd.h>

pid_t fork(void);
```

➔ 成功时返回进程 ID，失败时返回-1。

fork函数将创建调用的进程副本（概念上略难）。也就是说，并非根据完全不同的程序创建进程，而是复制正在运行的、调用fork函数的进程。另外，两个进程都将执行fork函数调用后的语句（准确地说是在fork函数返回后）。但因为通过同一个进程、复制相同的内存空间，之后的程序

流要根据fork函数的返回值加以区分。即利用fork函数的如下特点区分程序执行流程。

- 父进程：fork函数返回子进程ID。
- 子进程：fork函数返回0。

此处“父进程”（Parent Process）指原进程，即调用fork函数的主体，而“子进程”（Child Process）是通过父进程调用fork函数复制出的进程。接下来讲解调用fork函数后的程序运行流程，如图10-1所示。

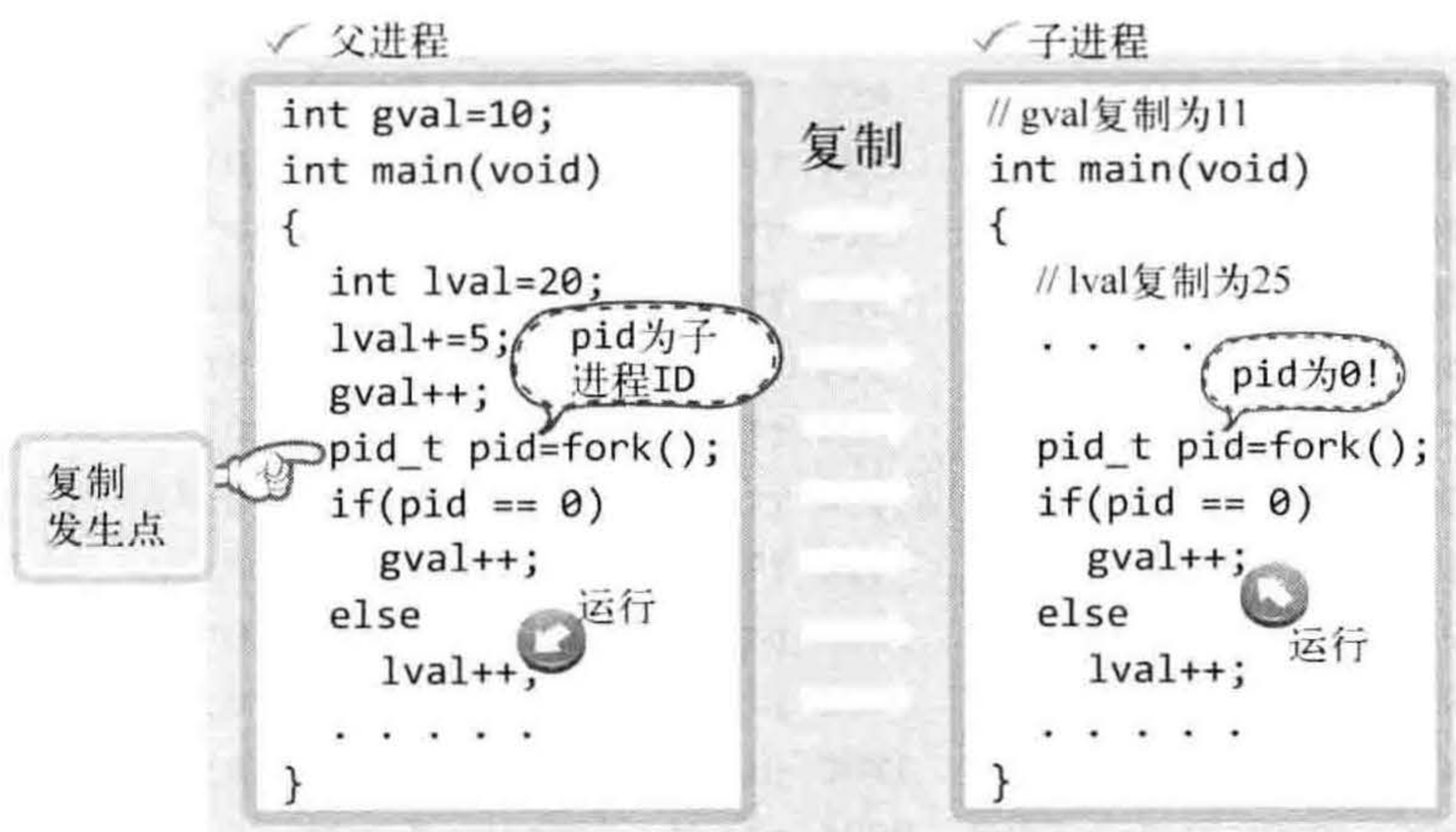



图10-1　fork函数的调用

从图10-1中可以看到，父进程调用fork函数的同时复制出子进程，并分别得到fork函数的返回值。但复制前，父进程将全局变量gval增加到11，将局部变量lval的值增加到25，因此在这种状态下完成进程复制。复制完成后根据fork函数的返回类型区分父子进程。父进程将lval的值加1，但这不会影响子进程的lval值。同样，子进程将gval的值加1也不会影响到父进程的gval。因为fork函数调用后分成了完全不同的进程，只是二者共享同一代码而已。接下来给出示例验证之前的内容。

❖ fork.c

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.
4.  int gval=10;
5.  int main(int argc, char *argv[])
6.  {
7.      pid_t pid;
8.      int lval=20;
9.      gval++, lval+=5;
10.
11.     pid=fork();
12.     if(pid==0)  // if Child Process
13.         gval+=2, lval+=2;
```

```
14.     else        // if Parent Process
15.         gval-=2, lval-=2;
16.
17.     if(pid==0)
18.         printf("Child Proc: [%d, %d] \n", gval, lval);
19.     else
20.         printf("Parent Proc: [%d, %d] \n", gval, lval);
21.     return 0;
22. }
```

- 第11行：创建子进程。父进程的pid中存有子进程的ID，子进程的pid是0。
- 第12、18行：子进程执行这2行代码，因为pid为0。
- 第15、20行：父进程执行这2行代码，因为此时pid中存有子进程ID。

❖ 运行结果：fork.c

```
root@my_linux:/tcpip# gcc fork.c -o fork
root@my_linux:/tcpip# ./fork
Child Proc: [13, 27]
Parent Proc: [9, 23]
```

从运行结果可以看出，调用fork函数后，父子进程拥有完全独立的内存结构。我认为关于fork函数无需更多示例，希望各位通过该示例充分理解调用fork函数创建进程的方法。

## 10.2 进程和僵尸进程

文件操作中，关闭文件和打开文件同等重要。同样，进程销毁也和进程创建同等重要。如果未认真对待进程销毁，它们将变成僵尸进程困扰各位。大家可能觉得这是在看玩笑，但事实的确如此。

### 僵尸（Zombie）进程

大家应该听说过僵尸。恐怖电影中的僵尸可以复活，给主人公造成极大的麻烦。一两只还好对付，但它们一般都成群结队出现，给观众一种刺激和紧张感。但我们的主人公通常神通广大，即使几百只僵尸同时出现也能顺利脱险（僵尸通常行动缓慢），因为主人公知道如何对付僵尸。结局就是所有僵尸都会走向灭亡。

进程的世界同样如此。进程完成工作后（执行完main函数中的程序后）应被销毁，但有时这些进程将变成僵尸进程，占用系统中的重要资源。这种状态下的进程称作“僵尸进程”，这也是给系统带来负担的原因之一。就像电影中描述的那样，我们应该消灭这种进程。当然应掌握正确的方法，否则它会死灰复燃。

## 产生僵尸进程的原因

为了防止僵尸进程的产生，先解释产生僵尸进程的原因。利用如下两个示例展示调用fork函数产生子进程的终止方式。

- 传递参数并调用exit函数。
- main函数中执行return语句并返回值。

向exit函数传递的参数值和main函数的return语句返回的值都会传递给操作系统。而操作系统不会销毁子进程，直到把这些值传递给产生该子进程的父进程。处在这种状态下的进程就是僵尸进程。也就是说，将子进程变成僵尸进程的正是操作系统。既然如此，此僵尸进程何时被销毁呢？其实已经给出提示。

“应该向创建子进程的父进程传递子进程的exit参数值或return语句的返回值。”

如何向父进程传递这些值呢？操作系统不会主动把这些值传递给父进程。只有父进程主动发起请求（函数调用）时，操作系统才会传递该值。换言之，如果父进程未主动要求获得子进程的结束状态值，操作系统将一直保存，并让子进程长时间处于僵尸进程状态。也就是说，父母要负责收回自己生的孩子（也许这种描述有些不妥）。接下来的示例将创建僵尸进程。

❖ zombie.c

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.
4.  int main(int argc, char *argv[])
5.  {
6.      pid_t pid=fork();
7.
8.      if(pid==0)  // if Child Process
9.      {
10.         puts("Hi, I am a child process");
11.     }
12.     else
13.     {
14.         printf("Child Process ID: %d \n", pid);
15.         sleep(30);  // Sleep 30 sec.
16.     }
17.
18.     if(pid==0)
19.         puts("End child process");
20.     else
21.         puts("End parent process");
22.     return 0;
23. }
```

- 第14行：输出子进程ID。可以通过该值查看子进程状态（是否为僵尸进程）。
- 第15行：父进程暂停30秒。如果父进程终止，处于僵尸状态的子进程将同时销毁。因此，延缓父进程的执行以验证僵尸进程。

❖ 运行结果：zombie.c

```
root@my_linux:/tcpip# gcc zombie.c -o zombie
root@my_linux:/tcpip# ./zombie
Hi, I am a child process
End child process
Child Process ID: 10977
```

程序开始运行后，将在如上所示状态暂停。跳出这种状态前（30秒内），应验证子进程是否为僵尸进程。该验证在其他控制台窗口进行。

❖ 运行结果：僵尸进程的验证

```
root@my_linux:/tcpip# ps au
USER    PID   %CPU %MEM  VSZ   RSS  TTY    STAT  START  TIME  COMMAND
root    4409  0.0  0.1  1780   524  tty3   Ss+   15:57  0:00  /sbin/getty 384
root    4410  0.0  0.1  1780   524  tty6   Ss+   15:57  0:00  /sbin/getty 384
root    5155  2.6  4.0 30188 20856  tty7   Rs+   15:57  7:08  /usr/X11R6/bin/
. . . . .
root    10976 0.0  0.0  1628   368  pts/0  S+    20:26  0:00  ./zombie
root    10977 0.0  0.0     0     0  pts/0  Z+    20:26  0:00  [zom] <defunct>
. . . . .
```

可以看出，PID为10977的进程状态为僵尸进程（Z+）。另外，经过30秒的等待时间后，PID为10976的父进程和之前的僵尸子进程同时销毁。

**后台处理（Background Processing）**

后台处理是指将控制台窗口中的指令放到后台运行的方式。如果以如下方式运行上述示例，则程序将在后台运行（&将触发后台处理）。

```
root@my_linux:/tcpip# ./zombie &
```

如果采用这种方式运行程序，即可在同一控制台输入下列命令，无需另外打开新控制台。

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

希望各位根据介绍以后台处理方式运行之前的示例。

## 销毁僵尸进程 1：利用 wait 函数

如前所述，为了销毁子进程，父进程应主动请求获取子进程的返回值。接下来讨论发起请求的具体方法（幸好非常简单），共有2种，其中之一就是调用如下函数。

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

pid_t wait(int * statloc);
```

➔ 成功时返回终止的子进程 ID，失败时返回-1。

调用此函数时如果已有子进程终止，那么子进程终止时传递的返回值（exit函数的参数值、main函数的return返回值）将保存到该函数的参数所指内存空间。但函数参数指向的单元中还包含其他信息，因此需要通过下列宏进行分离。

- WIFEXITED 子进程正常终止时返回“真”（true）。
- WEXITSTATUS 返回子进程的返回值。

也就是说，向wait函数传递变量status的地址时，调用wait函数后应编写如下代码。

```
if(WIFEXITED(status))  //是正常终止的吗?
{
    puts("Normal termination!");
    printf("Child pass num: %d", WEXITSTATUS(status));  //那么返回值是多少?
}
```

根据上述内容编写示例，此示例中不会再让子进程变成僵尸进程。

❖ wait.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <unistd.h>
4.  #include <sys/wait.h>
5.
6.  int main(int argc, char *argv[])
7.  {
8.      int status;
9.      pid_t pid=fork();
10.
11.     if(pid==0)
12.     {
13.         return 3;
14.     }
```

```
15.     else
16.     {
17.         printf("Child PID: %d \n", pid);
18.         pid=fork();
19.         if(pid==0)
20.         {
21.             exit(7);
22.         }
23.         else
24.         {
25.             printf("Child PID: %d \n", pid);
26.             wait(&status);
27.             if(WIFEXITED(status))
28.                 printf("Child send one: %d \n", WEXITSTATUS(status));
29.
30.             wait(&status);
31.             if(WIFEXITED(status))
32.                 printf("Child send two: %d \n", WEXITSTATUS(status));
33.             sleep(30);  // Sleep 30 sec.
34.         }
35.     }
36.     return 0;
37. }
```

**代码说明**

- 第9、13行：第9行创建的子进程将在第13行通过main函数中的return语句终止。
- 第18、21行：第18行中创建的子进程将在第21行通过调用exit函数终止。
- 第26行：调用wait函数。之前终止的子进程相关信息将保存到status变量，同时相关子进程被完全销毁。
- 第27、28行：第27行中通过WIFEXITED宏验证子进程是否正常终止。如果正常退出，则调用WEXITSTATUS宏输出子进程的返回值。
- 第30~32行：因为之前创建了2个进程，所以再次调用wait函数和宏。
- 第33行：为暂停父进程终止而插入的代码。此时可以查看子进程的状态。

❖ 运行结果：wait.c

```
root@my_linux:/tcpip# gcc wait.c -o wait
root@my_linux:/tcpip# ./wait
Child PID: 12337
Child PID: 12338
Child send one: 3
Child send two: 7
```

系统中并无上述结果中的PID对应的进程，希望各位进行验证。这是因为调用了wait函数，完全销毁了该进程。另外2个子进程终止时返回的3和7传递到了父进程。

这就是通过调用wait函数消灭僵尸进程的方法。调用wait函数时，如果没有已终止的子进程，那么程序将阻塞（Blocking）直到有子进程终止，因此需谨慎调用该函数。

## 销毁僵尸进程 2：使用 waitpid 函数

wait函数会引起程序阻塞，还可以考虑调用waitpid函数。这是防止僵尸进程的第二种方法，也是防止阻塞的方法。

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

pid_t waitpid(pid_t pid, int * statloc, int options);
```

➔ 成功时返回终止的子进程 ID（或 0），失败时返回-1。

- pid　等待终止的目标子进程的ID，若传递-1，则与wait函数相同，可以等待任意子进程终止。
- statloc　与wait函数的statloc参数具有相同含义。
- options　传递头文件sys/wait.h中声明的常量WNOHANG，即使没有终止的子进程也不会进入阻塞状态，而是返回0并退出函数。

下面介绍调用上述函数的示例。调用waitpid函数时，程序不会阻塞。各位应重点观察这点。

❖ waitpid.c

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.  #include <sys/wait.h>
4.
5.  int main(int argc, char *argv[])
6.  {
7.      int status;
8.      pid_t pid=fork();
9.
10.     if(pid==0)
11.     {
12.         sleep(15);
13.         return 24;
14.     }
15.     else
16.     {
17.         while(!waitpid(-1, &status, WNOHANG))
18.         {
19.             sleep(3);
20.             puts("sleep 3sec.");
21.         }
22.
23.         if(WIFEXITED(status))
24.             printf("Child send %d \n", WEXITSTATUS(status));
25.     }
26.     return 0;
27. }
```

- 第12行：调用sleep函数推迟子进程的执行。这会导致程序延迟15秒。
- 第17行：while循环中调用waitpid函数。向第三个参数传递WNOHANG，因此，若之前没有终止的子进程将返回0。

❖ 运行结果：waitpid.c

```
root@my_linux:/tcpip# gcc waitpid.c -o waitpid
root@my_linux:/tcpip# ./waitpid
sleep 3sec.
sleep 3sec.
sleep 3sec.
sleep 3sec.
sleep 3sec.
Child send 24
```

可以看出第20行共执行了5次。另外，这也证明waitpid函数并未阻塞。

## 10.3 信号处理

我们已经知道了进程创建及销毁方法，但还有一个问题没解决。

“子进程究竟何时终止？调用waitpid函数后要无休止地等待吗？”

父进程往往与子进程一样繁忙，因此不能只调用waitpid函数以等待子进程终止。接下来讨论解决方案。

### 向操作系统求助

子进程终止的识别主体是操作系统，因此，若操作系统能把如下信息告诉正忙于工作的父进程，将有助于构建高效的程序。

“嘿，父进程！你创建的子进程终止了！”

此时父进程将暂时放下工作，处理子进程终止相关事宜。这是不是既合理又很酷的想法呢？为了实现该想法，我们引入信号处理（Signal Handling）机制。此处的“信号”是在特定事件发生时由操作系统向进程发送的消息。另外，为了响应该消息，执行与消息相关的自定义操作的过程称为“处理”或“信号处理”。关于这两点稍后将再次说明，各位现在不用完全理解这些概念。

### 关于 JAVA 的题外话：保持开放思维

路途漫漫，我们稍事休息，先讨论一下JAVA。我想讨论的主题如下：

“技术上要保持开放思维。”

JAVA语言具有平台移植性，经历了长时间的发展和变化，其优势在企业级开发环境下尤为明显。理论介绍到此为止，接下来通过讨论JAVA拓宽技术视野。

JAVA在编程语言层面支持进程或线程（稍后将介绍），但C语言及C++语言并不支持。也就是说，ANSI标准并未定义支持进程或线程的函数（JAVA的方法）。但仔细想想，这也是合理的。进程或线程应该由操作系统提供支持。因此，Windows中按照Windows的方式，Linux中按照Linux的方式创建进程或线程。但JAVA为了保持平台移植性，以独立于操作系统的方式提供进程和线程的创建方法。因此，JAVA在语言层面支持进程和线程的创建。

既然如此，JAVA网络编程是否相对简单？就像大家之前学习过的，网络编程中需要一定的操作系统相关知识，因此，有些人会把网络编程当做系统编程的一部分。基于JAVA进行网络编程时，的确会摆脱特定的操作系统，所以有人会误认为JAVA网络编程相对简单。

如果在语言层面支持网络编程所需的所有机制，将延长学习时间。要通过面向对象的方法编写高性能网络程序，需要更多努力和知识。如果有机会，还是希望大家尝试JAVA网络编程，而不仅仅局限于Linux或Windows。JAVA在分布式环境中提供理想的网络编程模型，我也曾沉醉于JAVA技术。

对技术有偏见相当于限制了自己的学习范围，希望各位摒弃偏见。也许有人质疑，学习一门技术尚且困难，如何能同时学习多门技术呢？我并不是要求大家同步学习，而是希望各位以开放的思维多关注身边的技术。

**写于 2003 年的文章**

我于 2003 年夏写下上文，当时觉得再过几年可能需要修正其中的说法，但现在依然想用此文激励后辈，因此稍作修改后收录于本书。

## 信号与 signal 函数

下列进程和操作系统间的对话是帮助大家理解信号处理而编写的，其中包含了所有信号处理相关内容。

- 进程：“嘿，操作系统！如果我之前创建的子进程终止，就帮我调用zombie_handler函数。”
- 操作系统：“好的！如果你的子进程终止，我会帮你调用zombie_handler函数，你先把该函数要执行的语句编好！”

上述对话中进程所讲的相当于“注册信号”过程，即进程发现自己的子进程结束时，请求操作系统调用特定函数。该请求通过如下函数调用完成（因此称此函数为信号注册函数）。

```
#include <signal.h>

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

➔ 为了在产生信号时调用，返回之前注册的函数指针。

上述函数的返回值类型为函数指针，因此函数声明有些繁琐。若各位不太熟悉返回值类型为函数指针的声明，希望加强学习（不懂函数指针将无法理解后续内容）。现在为了便于讲解，我将上述函数声明整理如下。

- 函数名：signal
- 参数：int signo, void (* func)(int)
- 返回类型：参数为int型，返回void型函数指针。

调用上述函数时，第一个参数为特殊情况信息，第二个参数为特殊情况下将要调用的函数的地址值（指针）。发生第一个参数代表的情况时，调用第二个参数所指的函数。下面给出可以在signal函数中注册的部分特殊情况和对应的常数。

- SIGALRM：已到通过调用alarm函数注册的时间。
- SIGINT：输入CTRL+C。
- SIGCHLD：子进程终止。

接下来编写调用signal函数的语句完成如下请求：

“子进程终止则调用mychild函数。”

此时mychild函数的参数应为int，返回值类型应为void。只有这样才能成为signal函数的第二个参数。另外，常数SIGCHLD定义了子进程终止的情况，应成为signal函数的第一个参数。也就是说，signal函数调用语句如下。

```
signal(SIGCHLD, mychild);
```

接下来编写signal函数的调用语句，分别完成如下2个请求。

“已到通过alarm函数注册的时间，请调用timeout函数。”
“输入CTRL+C时调用keycontrol函数。”

代表这2种情况的常数分别为SIGALRM和SIGINT，因此按如下方式调用signal函数。

```
signal(SIGALRM, timeout);
signal(SIGINT, keycontrol);
```

以上就是信号注册过程。注册好信号后，发生注册信号时（注册的情况发生时），操作系统

将调用该信号对应的函数。下面通过示例验证，先介绍alarm函数。

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);

    ➔ 返回 0 或以秒为单位的距 SIGALRM 信号发生所剩时间。
```

如果调用该函数的同时向它传递一个正整型参数，相应时间后（以秒为单位）将产生SIGALRM信号。若向该函数传递0，则之前对SIGALRM信号的预约将取消。如果通过该函数预约信号后未指定该信号对应的处理函数，则（通过调用signal函数）终止进程，不做任何处理。希望引起注意。

接下来给出信号处理相关示例，希望各位通过该示例彻底掌握之前的内容。

❖ signal.c

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.  #include <signal.h>
4.
5.  void timeout(int sig)
6.  {
7.      if(sig==SIGALRM)
8.          puts("Time out!");
9.      alarm(2);
10. }
11. void keycontrol(int sig)
12. {
13.     if(sig==SIGINT)
14.         puts("CTRL+C pressed");
15. }
16.
17. int main(int argc, char *argv[])
18. {
19.     int i;
20.     signal(SIGALRM, timeout);
21.     signal(SIGINT, keycontrol);
22.     alarm(2);
23.
24.     for(i=0; i<3; i++)
25.     {
26.         puts("wait...");
27.         sleep(100);
28.     }
29.     return 0;
30. }
```

- 第5、11行：分别定义信号处理函数。这种类型的函数称为信号处理器（Handler）。
- 第9行：为了每隔2秒重复产生SIGALRM信号，在信号处理器中调用alarm函数。
- 第20、21行：注册SIGALRM、SIGINT信号及相应处理器。
- 第22行：预约2秒后发生SIGALRM信号。
- 第27行：为了查看信号产生和信号处理器的执行并提供每次100秒、共3次的等待时间，在循环中调用sleep函数。也就是说，再过300秒、约5分钟后终止程序，这是相当长的一段时间，但实际执行时只需不到10秒。关于其原因稍后再解释。

❖ 运行结果：signal.c

```
root@my_linux:/tcpip# gcc signal.c -o signal
root@my_linux:/tcpip# ./signal
wait...
Time out!
wait...
Time out!
wait...
Time out!
```

上述是没有任何输入时的运行结果。下面在运行过程中输入CTRL+C。可以看到输出“CTRL+C pressed”字符串。有一点必须说明：

> “发生信号时将唤醒由于调用sleep函数而进入阻塞状态的进程。”

调用函数的主体的确是操作系统，但进程处于睡眠状态时无法调用函数。因此，产生信号时，为了调用信号处理器，将唤醒由于调用sleep函数而进入阻塞状态的进程。而且，进程一旦被唤醒，就不会再进入睡眠状态。即使还未到sleep函数中规定的时间也是如此。所以，上述示例运行不到10秒就会结束，连续输入CTRL+C则有可能1秒都不到。

## 利用 sigaction 函数进行信号处理



前面所学的内容足以用来编写防止僵尸进程生成的代码。但我还想介绍sigaction函数，它类似于signal函数，而且完全可以代替后者，也更稳定。之所以稳定，是因为如下原因：

> “signal函数在UNIX系列的不同操作系统中可能存在区别，但sigaction函数完全相同。”

实际上现在很少使用signal函数编写程序，它只是为了保持对旧程序的兼容。下面介绍sigaction函数，但只讲解可替换signal函数的功能，因为全面介绍会给各位带来不必要的负担。

```
#include <signal.h>

int sigaction(int signo, const struct sigaction * act, struct sigaction *
oldact);
```

➔ 成功时返回 0，失败时返回-1。

- signo　与signal函数相同，传递信号信息。
- act　对应于第一个参数的信号处理函数（信号处理器）信息。
- oldact　通过此参数获取之前注册的信号处理函数指针，若不需要则传递0。

声明并初始化sigaction结构体变量以调用上述函数，该结构体定义如下。

```
struct sigaction
{
    void (*sa_handler)(int);
    sigset_t sa_mask;
    int sa_flags;
}
```

此结构体的sa_handler成员保存信号处理函数的指针值（地址值）。sa_mask和sa_flags的所有位均初始化为0即可。这2个成员用于指定信号相关的选项和特性，而我们的目的主要是防止产生僵尸进程，故省略。理解这些参数所需参考书将在后面给出。

下面给出示例，其中还包括了尚未讲解的使用sigaction函数所需全部内容。

❖ sigaction.c

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.  #include <signal.h>
4.
5.  void timeout(int sig)
6.  {
7.      if(sig==SIGALRM)
8.          puts("Time out!");
9.      alarm(2);
10. }
11.
12. int main(int argc, char *argv[])
13. {
14.     int i;
15.     struct sigaction act;
16.     act.sa_handler=timeout;
17.     sigemptyset(&act.sa_mask);
18.     act.sa_flags=0;
```

```
19.     sigaction(SIGALRM, &act, 0);
20.
21.     alarm(2);
22.
23.     for(i=0; i<3; i++)
24.     {
25.         puts("wait...");
26.         sleep(100);
27.     }
28.     return 0;
29. }
```

- 第15、16行：为了注册信号处理函数，声明sigaction结构体变量并在sa_handler成员中保存函数指针值。
- 第17行：调用sigemptyset函数将sa_mask成员的所有位初始化为0。
- 第18行：sa_flags成员同样初始化为0。
- 第19、21行：注册SIGALRM信号的处理器。调用alarm函数预约2秒后发生SIGALRM信号。

❖ 运行结果：sigaction.c

```
root@my_linux:/tcpip# gcc sigaction.c -o sigaction
root@my_linux:/tcpip# ./sigaction
wait...
Time out!
wait...
Time out!
wait...
Time out!
```

这就是信号处理相关理论，以此为基础讨论消灭僵尸进程的方法。

## 利用信号处理技术消灭僵尸进程

我相信各位也可以独立编写消灭僵尸进程的示例。子进程终止时将产生SIGCHLD信号，知道这一点就很容易完成。接下来利用sigaction函数编写示例。

❖ remove_zombie.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <unistd.h>
4.  #include <signal.h>
5.  #include <sys/wait.h>
6.
7.  void read_childproc(int sig)
8.  {
```

```
9.      int status;
10.     pid_t id=waitpid(-1, &status, WNOHANG);
11.     if(WIFEXITED(status))
12.     {
13.         printf("Removed proc id: %d \n", id);
14.         printf("Child send: %d \n", WEXITSTATUS(status));
15.     }
16. }
17.
18. int main(int argc, char *argv[])
19. {
20.     pid_t pid;
21.     struct sigaction act;
22.     act.sa_handler=read_childproc;
23.     sigemptyset(&act.sa_mask);
24.     act.sa_flags=0;
25.     sigaction(SIGCHLD, &act, 0);
26.
27.     pid=fork();
28.     if(pid==0)  /*子进程执行区域*/
29.     {
30.         puts("Hi! I'm child process");
31.         sleep(10);
32.         return 12;
33.     }
34.     else    /*父进程执行区域*/
35.     {
36.         printf("Child proc id: %d \n", pid);
37.         pid=fork();
38.         if(pid==0)  /*另一子进程执行区域*/
39.         {
40.             puts("Hi! I'm child process");
41.             sleep(10);
42.             exit(24);
43.         }
44.         else
45.         {
46.             int i;
47.             printf("Child proc id: %d \n", pid);
48.             for(i=0; i<5; i++)
49.             {
50.                 puts("wait...");
51.                 sleep(5);
52.             }
53.         }
54.     }
55.     return 0;
56. }
```

**代码说明**

- 第21~25行：注册SIGCHLD信号对应的处理器。若子进程终止，则调用第7行中定义的函数。处理函数中调用了waitpid函数，所以子进程将正常终止，不会成为僵尸进程。
- 第27、37行：父进程共创建了2个子进程。
- 第48、51行：为了等待发生SIGCHLD信号，使父进程共暂停5次，每次间隔5秒。发生信号时，父进程将被唤醒，因此实际暂停时间不到25秒。

❖ 运行结果：remove_zombie.c

```
root@my_linux:/tcpip# gcc remove_zombie.c -o zombie
root@my_linux:/tcpip# ./zombie
Hi! I'm child process
Child proc id: 9529
Hi! I'm child process
Child proc id: 9530
wait...
wait...
Removed proc id: 9530
Child send: 24
wait...
Removed proc id: 9529
Child send: 12
wait...
wait...
```

可以看出，子进程并未变成僵尸进程，而是正常终止了。接下来利用进程相关知识编写服务器端。

## 10.4 基于多任务的并发服务器

我们已做好了利用fork函数编写并发服务器的准备，现在可以开始编写像样的服务器端了。

### 基于进程的并发服务器模型

之前的回声服务器端每次只能向1个客户端提供服务。因此，我们将扩展回声服务器端，使其可以同时向多个客户端提供服务。图10-2给出了基于多进程的并发回声服务器端的实现模型。

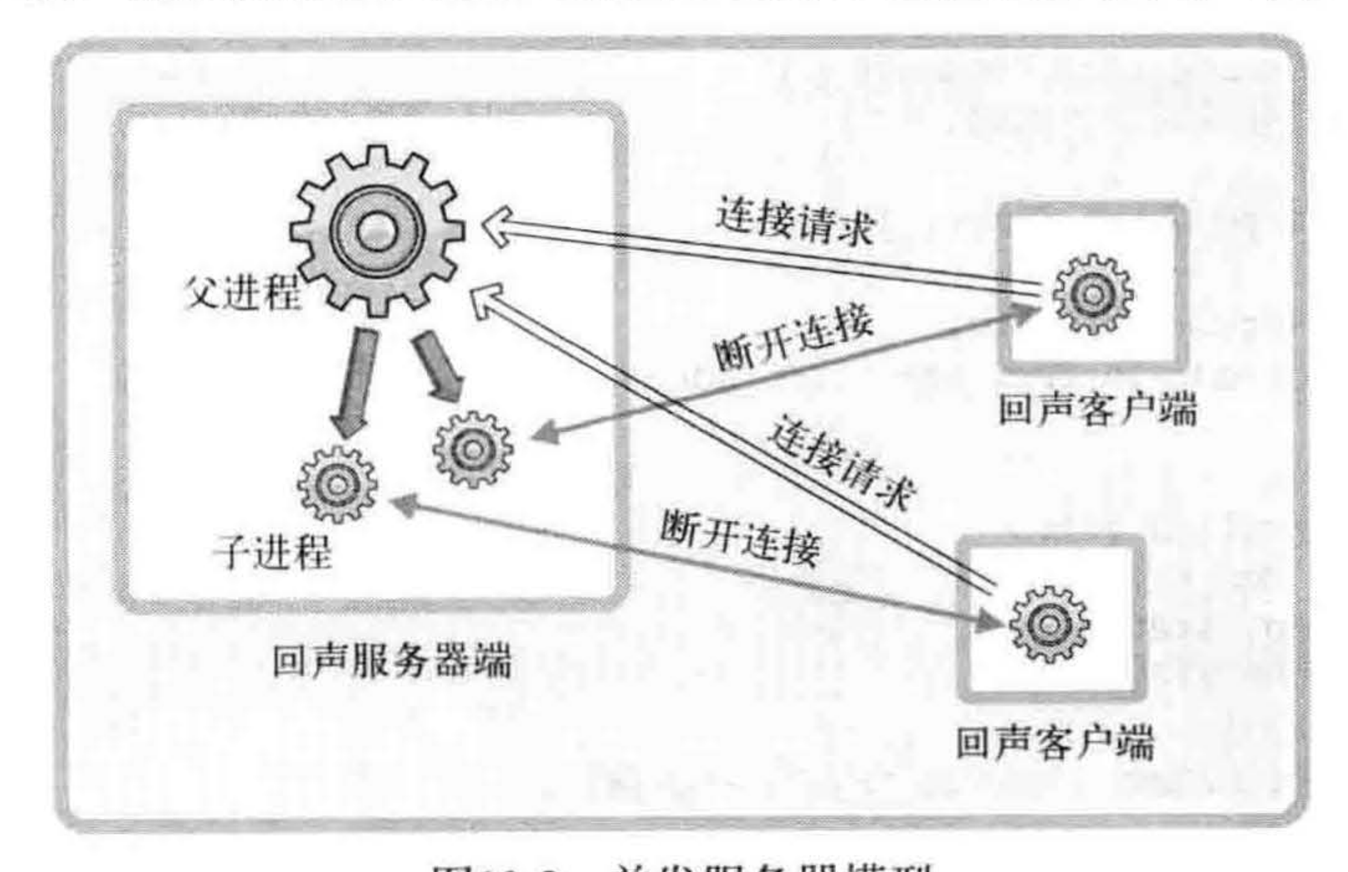



图10-2 并发服务器模型

从图10-2可以看出，每当有客户端请求服务（连接请求）时，回声服务器端都创建子进程以提供服务。请求服务的客户端若有5个，则将创建5个子进程提供服务。为了完成这些任务，需要经过如下过程，这是与之前的回声服务器端的区别所在。

- 第一阶段：回声服务器端（父进程）通过调用accept函数受理连接请求。
- 第二阶段：此时获取的套接字文件描述符创建并传递给子进程。
- 第三阶段：子进程利用传递来的文件描述符提供服务。

此处容易引起困惑的是向子进程传递套接字文件描述符的方法。但各位读完代码后会发现，这其实没什么大不了的，因为子进程会复制父进程拥有的所有资源。实际上根本不用另外经过传递文件描述符的过程。

## 实现并发服务器

虽然我已经给出了所有理论说明，但大家也许还没想出具体的实现方法，这就有必要理解具体代码。下面给出并发回声服务器端的实现代码。当然，程序是基于多进程实现的，可以结合第4章的回声客户端运行。

❖ echo_mpserv.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <unistd.h>
5.  #include <signal.h>
6.  #include <sys/wait.h>
7.  #include <arpa/inet.h>
8.  #include <sys/socket.h>
9.
10. #define BUF_SIZE 30
11. void error_handling(char *message);
12. void read_childproc(int sig);
13.
14. int main(int argc, char *argv[])
15. {
16.     int serv_sock, clnt_sock;
17.     struct sockaddr_in serv_adr, clnt_adr;
18.
19.     pid_t pid;
20.     struct sigaction act;
21.     socklen_t adr_sz;
22.     int str_len, state;
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=read_childproc;
30.     sigemptyset(&act.sa_mask);
31.     act.sa_flags=0;
32.     state=sigaction(SIGCHLD, &act, 0);
33.     serv_sock=socket(PF_INET, SOCK_STREAM, 0);
34.     memset(&serv_adr, 0, sizeof(serv_adr));
35.     serv_adr.sin_family=AF_INET;
36.     serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
37.     serv_adr.sin_port=htons(atoi(argv[1]));
38.
39.     if(bind(serv_sock, (struct sockaddr*) &serv_adr, sizeof(serv_adr))==-1)
40.         error_handling("bind() error");
41.     if(listen(serv_sock, 5)==-1)
42.         error_handling("listen() error");
43.
44.     while(1)
45.     {
46.         adr_sz=sizeof(clnt_adr);
47.         clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_adr, &adr_sz);
48.         if(clnt_sock==-1)
49.             continue;
50.         else
51.             puts("new client connected...");
52.         pid=fork();
53.         if(pid==-1)
54.         {
55.             close(clnt_sock);
56.             continue;
57.         }
58.         if(pid==0)  /*子进程运行区域*/
59.         {
60.             close(serv_sock);
61.             while((str_len=read(clnt_sock, buf, BUF_SIZE))!=0)
62.                 write(clnt_sock, buf, str_len);
63.
64.             close(clnt_sock);
65.             puts("client disconnected...");
66.             return 0;
67.         }
68.         else
69.             close(clnt_sock);
70.     }
71.     close(serv_sock);
72.     return 0;
73. }
74.
75. void read_childproc(int sig)
76. {
77.     pid_t pid;
78.     int status;
79.     pid=waitpid(-1, &status, WNOHANG);
80.     printf("removed proc id: %d \n", pid);
81. }
```

```
82. void error_handling(char * message)
83. {
84.     fputs(message, stderr);
85.     fputc('\n', stderr);
86.     exit(1);
87. }
```

- 第29~32行：为防止产生僵尸进程而编写的代码。
- 第47、52行：第47行调用accept函数后，在第52行调用fork函数。因此，父子进程分别带有1个第47行生成的套接字（受理客户端连接请求时创建的）文件描述符。
- 第58~66行：子进程运行的区域。此部分向客户端提供回声服务。第60行关闭第33行创建的服务器套接字，这是因为服务器套接字文件描述符同样也传递到子进程。关于这一点稍后将单独讨论。
- 第69行：第47行中通过accept函数创建的套接字文件描述符已复制给子进程，因此服务器端需要销毁自己拥有的文件描述符。关于这一点稍后将单独说明。

❖ 运行结果：echo_mpserv.c

```
root@my_linux:/tcpip# gcc echo_mpserv.c -o mpserv
root@my_linux:/tcpip# ./mpserv 9190
new client connected...
new client connected...
client disconnected...
removed proc id: 7012
client disconnected...
removed proc id: 7018
```

❖ 运行结果：echo_client.c one

```
root@my_linux:/home/swyoon/tcpip# ./client 127.0.0.1 9190
Connected...........
Input message(Q to quit): Hi I’m first client
Message from server: Hi I'm first client
Input message(Q to quit): Oh my friend go away~
Message from server: Oh my friend go away~
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:/home/swyoon/tcpip# ./client 127.0.0.1 9190
Connected...........
```

```
Input message(Q to quit): Hi I'm second client
Message from server: Hi I'm second client
Input message(Q to quit): Good bye~
Message from server: Good bye~
Input message(Q to quit): Q
```

启动服务器端后，要创建多个客户端并建立连接。可以验证服务器端同时向大多数客户端提供服务，不，一定要验证这一点。

## 通过 fork 函数复制文件描述符

示例echo_mpserv.c中给出了通过fork函数复制文件描述符的过程。父进程将2个套接字（一个是服务器端套接字，另一个是与客户端连接的套接字）文件描述符复制给子进程。

“只复制文件描述符吗？是否也复制了套接字呢？”

文件描述符的实际复制多少有些难以理解。调用fork函数时复制父进程的所有资源，有些人可能认为也会同时复制套接字。但套接字并非进程所有——从严格意义上说，套接字属于操作系统——只是进程拥有代表相应套接字的文件描述符。也不一定非要这样理解，仅因为如下原因，复制套接字也并不合理。

“复制套接字后，同一端口将对应多个套接字。”

示例echo_mpserv.c中的fork函数调用过程如图10-3所示。调用fork函数后，2个文件描述符指向同一套接字。

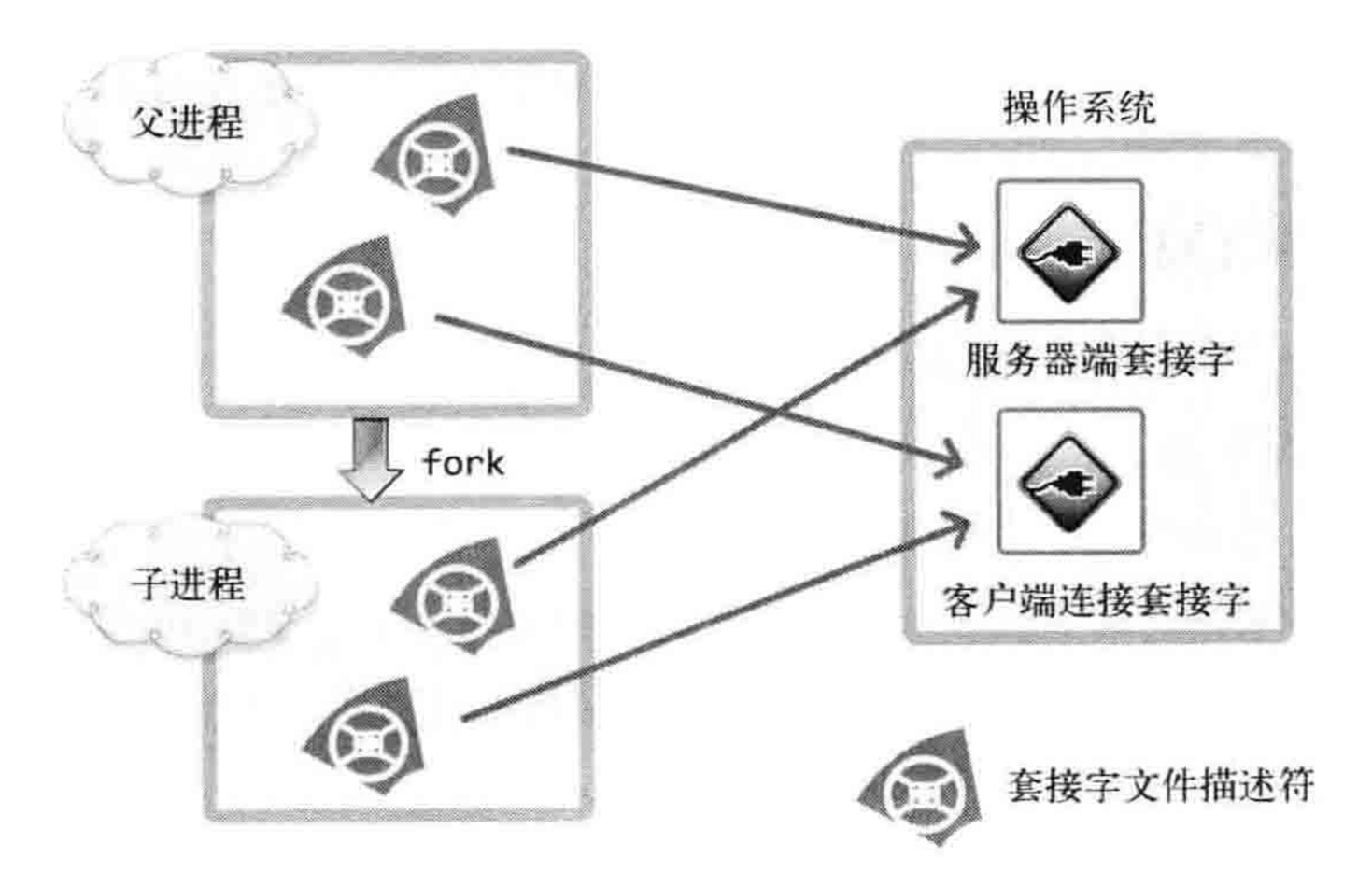



图10-3 调用fork函数并复制文件描述符

如图10-3所示，1个套接字中存在2个文件描述符时，只有2个文件描述符都终止（销毁）后，才能销毁套接字。如果维持图中的连接状态，即使子进程销毁了与客户端连接的套接字文件描述符，也无法完全销毁套接字（服务器端套接字同样如此）。因此，调用fork函数后，要将无关的套接字文件描述符关掉，如图10-4所示。

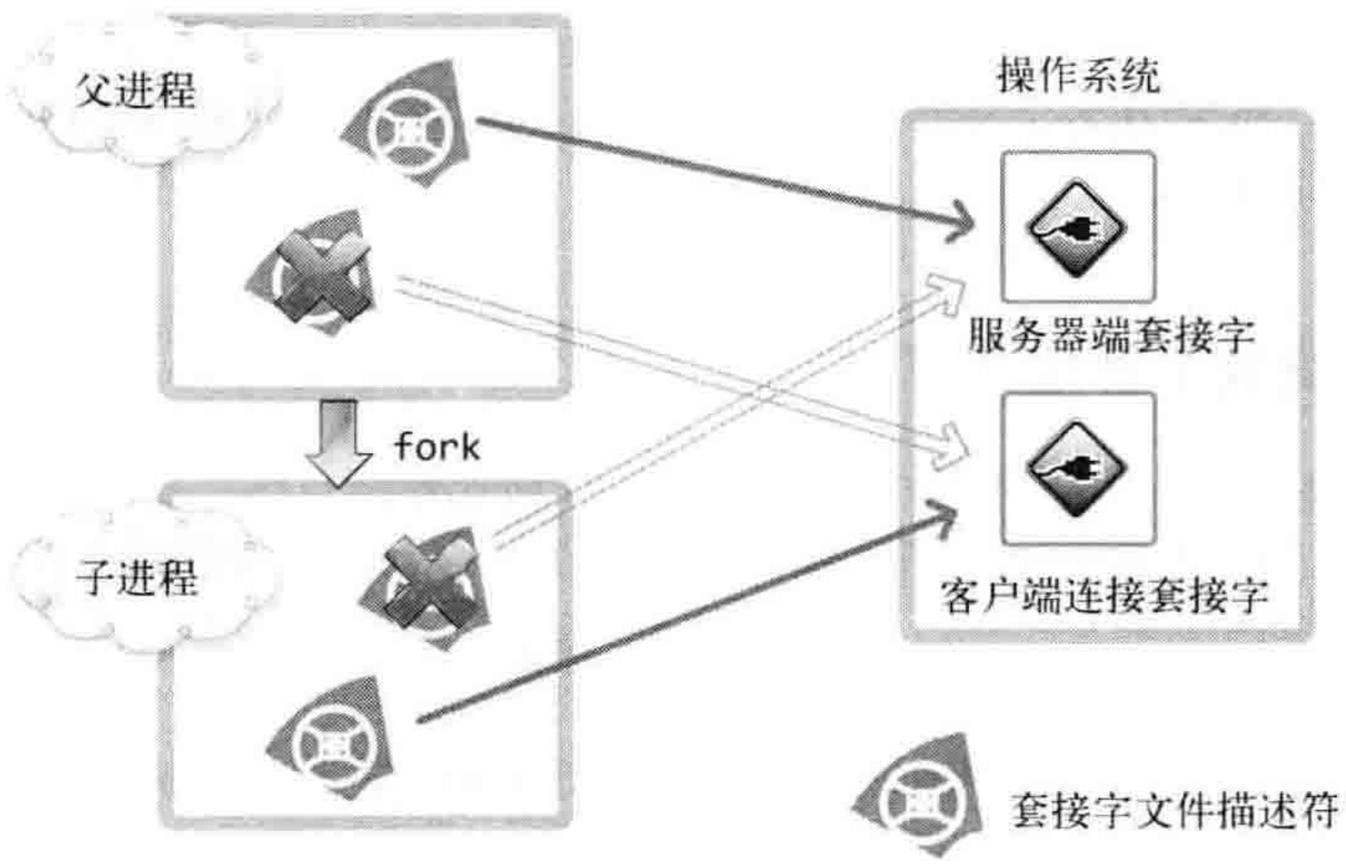



图10-4　整理复制的文件描述符

为了将文件描述符整理成图10-4的形式，示例echo_mpserv.c的第60行和第69行调用了close函数。

## 10.5　分割 TCP 的 I/O 程序

各位应该已经理解fork函数相关的所有有用内容。下面以此为基础，再讨论客户端中分割I/O程序（Routine）的方法。内容非常简单，大家不必有负担。

### 分割 I/O 程序的优点

我们已实现的回声客户端的数据回声方式如下：

> “向服务器端传输数据，并等待服务器端回复。无条件等待，直到接收完服务器端的回声数据后，才能传输下一批数据。”

传输数据后需要等待服务器端返回的数据，因为程序代码中重复调用了read和write函数。只能这么写的原因之一是，程序在1个进程中运行。但现在可以创建多个进程，因此可以分割数据收发过程。默认的分割模型如图10-5所示。

从图10-5可以看出，客户端的父进程负责接收数据，额外创建的子进程负责发送数据。分割后，不同进程分别负责输入和输出，这样，无论客户端是否从服务器端接收完数据都可以进行传输。

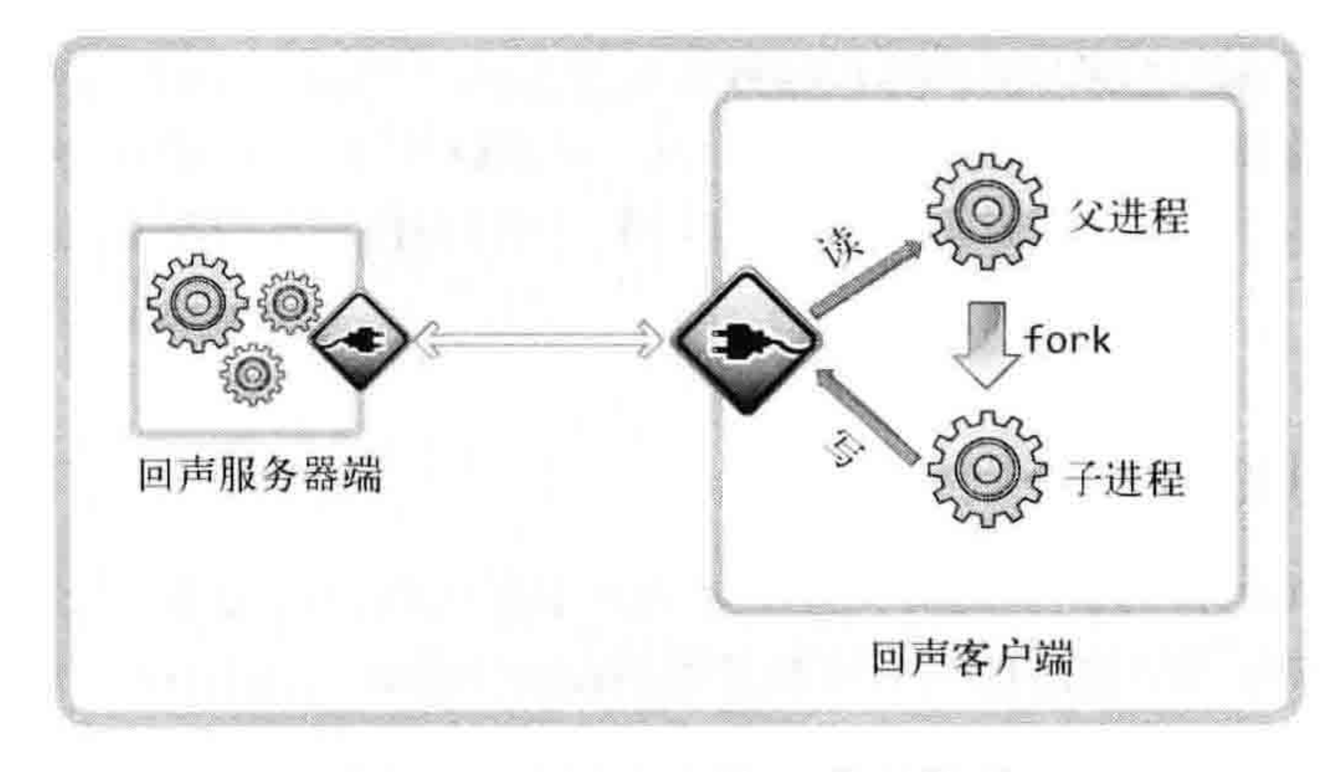



图10-5 回声客户端I/O分割模型

选择这种实现方式的原因有很多，但最重要的一点是，程序的实现更加简单。也许有人质疑：既然多产生1个进程，怎么能简化程序实现呢？其实，按照这种实现方式，父进程中只需编写接收数据的代码，子进程中只需编写发送数据的代码，所以会简化。实际上，在1个进程内同时实现数据收发逻辑需要考虑更多细节。程序越复杂，这种区别越明显，它也是公认的优点。

**回声客户端不用分割 I/O 程序**

实际上，回声客户端并无特殊原因进行 I/O 程序的分割，如若进行反而更复杂。本书只是为了讲解分割 I/O 的方法而选取了回声客户端，希望各位不要误解。

分割I/O程序的另一个好处是，可以提高频繁交换数据的程序性能，如图10-6所示。

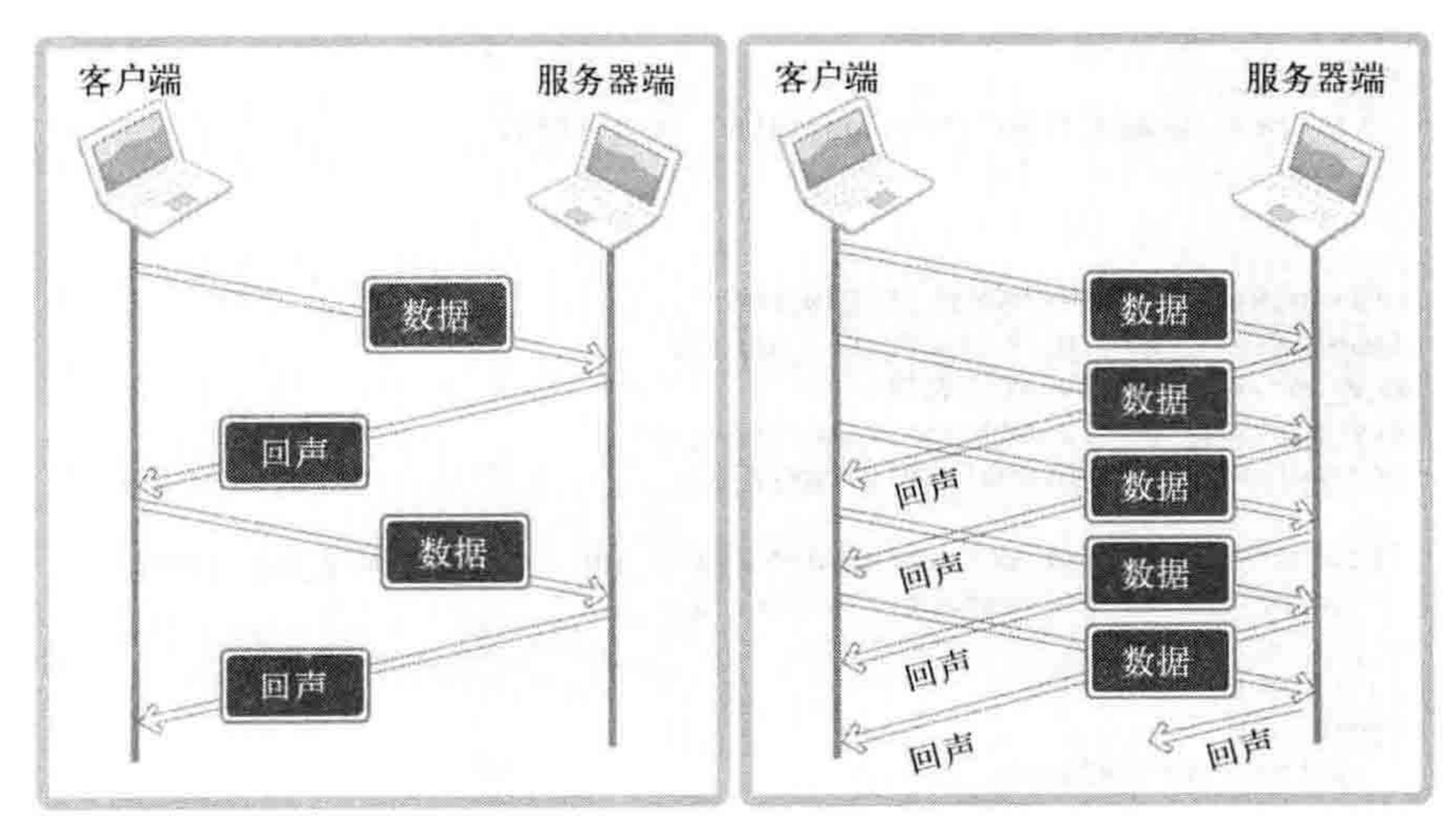



图10-6 数据交换方法比较

图10-6左侧演示的是之前的回声客户端数据交换方式，右侧演示的是分割I/O后的客户端数据传输方式。服务器端相同，不同的是客户端区域。分割I/O后的客户端发送数据时不必考虑接收数据的情况，因此可以连续发送数据，由此提高同一时间内传输的数据量。这种差异在网速较慢时尤为明显。

## 回声客户端的 I/O 程序分割

我们已经知道I/O程序分割的意义，接下来通过实际代码进行实现，分割的对象是回声客户端。下列回声客户端可以结合之前的回声服务器端echo_mpserv.c运行。

echo_mpclient.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. void read_routine(int sock, char *buf);
11. void write_routine(int sock, char *buf);
12.
13. int main(int argc, char *argv[])
14. {
15.     int sock;
16.     pid_t pid;
17.     char buf[BUF_SIZE];
18.     struct sockaddr_in serv_adr;
19.     if(argc!=3) {
20.         printf("Usage : %s <IP> <port>\n", argv[0]);
21.         exit(1);
22.     }
23.
24.     sock=socket(PF_INET, SOCK_STREAM, 0);
25.     memset(&serv_adr, 0, sizeof(serv_adr));
26.     serv_adr.sin_family=AF_INET;
27.     serv_adr.sin_addr.s_addr=inet_addr(argv[1]);
28.     serv_adr.sin_port=htons(atoi(argv[2]));
29.
30.     if(connect(sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr))==-1)
31.         error_handling("connect() error!");
32.
33.     pid=fork();
34.     if(pid==0)
35.         write_routine(sock, buf);
36.     else
37.         read_routine(sock, buf);
```

```
38.
39.     close(sock);
40.     return 0;
41. }
42.
43. void read_routine(int sock, char *buf)
44. {
45.     while(1)
46.     {
47.         int str_len=read(sock, buf, BUF_SIZE);
48.         if(str_len==0)
49.             return;
50.
51.         buf[str_len]=0;
52.         printf("Message from server: %s", buf);
53.     }
54. }
55. void write_routine(int sock, char *buf)
56. {
57.     while(1)
58.     {
59.         fgets(buf, BUF_SIZE, stdin);
60.         if(!strcmp(buf,"q\n") || !strcmp(buf,"Q\n"))
61.         {
62.             shutdown(sock, SHUT_WR);
63.             return;
64.         }
65.         write(sock, buf, strlen(buf));
66.     }
67. }
68. void error_handling(char *message)
69. {
70.     fputs(message, stderr);
71.     fputc('\n', stderr);
72.     exit(1);
73. }
```

- 第34~37行：第35行调用的write_routine函数中只有数据输出相关代码，第37行调用的read_routine函数中只有数据输入相关代码。像这样分割I/O并分别在不同函数中定义，将有利于代码实现。
- 第62行：调用shutdown函数向服务器端传递EOF。当然，执行第63行的return语句后，可以调用第39行的close函数传递EOF。但现在已通过第33行的fork函数调用复制了文件描述符，此时无法通过1次close函数调用传递EOF，因此需要通过shutdown函数调用另外传递。



运行结果跟普通回声服务器端/客户端相同，故省略。只是上述示例分割了I/O，为了简化输出过程，与之前示例不同，不会输出如下字符串：

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

无论是否接收消息，每次通过键盘输入字符串时都会输出上述字符串，可能造成输出混乱。基于多任务的服务器端实现方法讲解到此结束。

## 10.6 习题

(1) 下列关于进程的说法错误的是?

a. 从操作系统的角度上说，进程是程序运行的单位。
b. 进程根据创建方式建立父子关系。
c. 进程可以包含其他进程，即一个进程的内存空间可以包含其他进程。
d. 子进程可以创建其他子进程，而创建出来的子进程还可以创建其子进程，但所有这些进程只与一个父进程建立父子关系。

(2) 调用fork函数将创建子进程，以下关于子进程的描述错误的是?

a. 父进程销毁时也会同时销毁子进程。
b. 子进程是复制父进程所有资源创建出的进程。
c. 父子进程共享全局变量。
d. 通过fork函数创建的子进程将执行从开始到fork函数调用为止的代码。

(3) 创建子进程时将复制父进程的所有内容，此时的复制对象也包含套接字文件描述符。编写程序验证复制的文件描述符整数值是否与原文件描述符整数值相同。

(4) 请说明进程变为僵尸进程的过程及预防措施。

(5) 如果在未注册SIGINT信号的情况下输入Ctrl+C，将由操作系统默认的事件处理器终止程序。但如果直接注册Ctrl+C信号的处理器，则程序不会终止，而是调用程序员指定的事件处理器。编写注册处理函数的程序，完成如下功能：

“输入Ctrl+C时，询问是否确定退出程序，输入Y则终止程序。”

另外，编写程序使其每隔1秒输出简单字符串，并适用于上述时间处理器注册代码。

# 第 11 章 进程间通信

第 10 章讲解了如何创建进程，本章将讨论创建的 2 个进程之间交换数据的方法。这与构建服务器端并无直接关系，但可能有助于构建多种类型服务器端，以及更好地理解操作系统。

## 11.1 进程间通信的基本概念

进程间通信（Inter Process Communication）意味着两个不同进程间可以交换数据，为了完成这一点，操作系统中应提供两个进程可以同时访问的内存空间。

### 对进程间通信的基本理解

理解好进程间通信并没有想象中那么难，进程A和B之间的如下谈话内容就是一种进程间通信规则。

> “如果我有1个面包，变量bread的值就变为1。如果吃掉这个面包，bread的值又变回0。因此，你可以通过变量bread值判断我的状态。”

也就是说，进程A通过变量bread将自己的状态通知给了进程B，进程B通过变量bread听到了进程A的话。因此，只要有两个进程可以同时访问的内存空间，就可以通过此空间交换数据。但正如第10章所讲，进程具有完全独立的内存结构。就连通过fork函数创建的子进程也不会与父进程共享内存空间。因此，进程间通信只能通过其他特殊方法完成。

各位应该已经明白进程间通信的含义及其无法简单实现的原因，下面正式介绍进程间通信方法。

### 通过管道实现进程间通信

图11-1表示基于管道（PIPE）的进程间通信结构模型。

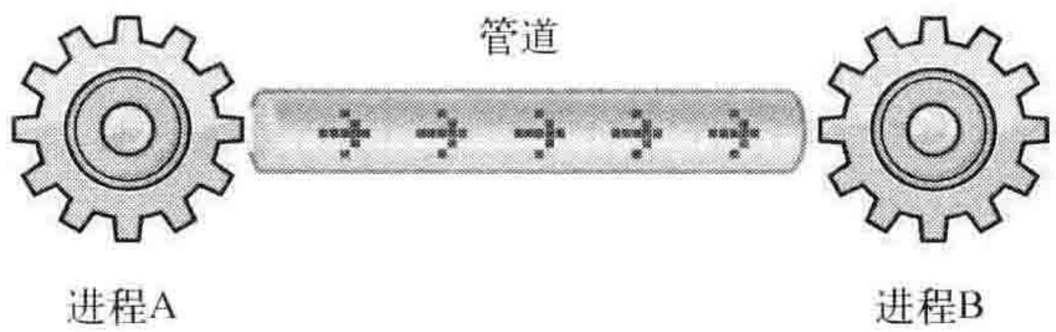



图11-1　基于管道的进程间通信模型

从图11-1中可以看到，为了完成进程间通信，需要创建管道。管道并非属于进程的资源，而是和套接字一样，属于操作系统（也就不是fork函数的复制对象）。所以，两个进程通过操作系统提供的内存空间进行通信。下面介绍创建管道的函数。

```
#include <unistd.h>

int pipe(int filedes[2]);
```

➔ 成功时返回 0，失败时返回-1。

- filedes[0]　通过管道接收数据时使用的文件描述符，即管道出口。
- filedes[1]　通过管道传输数据时使用的文件描述符，即管道入口。

以长度为2的int数组地址值作为参数调用上述函数时，数组中存有两个文件描述符，它们将被用作管道的出口和入口。父进程调用该函数时将创建管道，同时获取对应于出入口的文件描述符，此时父进程可以读写同一管道（相信大家也做过这样的实验）。但父进程的目的是与子进程进行数据交换，因此需要将入口或出口中的1个文件描述符传递给子进程。如何完成传递呢？答案就是调用fork函数。通过下列示例进行演示。

❖ pipe1.c

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.  #define BUF_SIZE 30
4.
5.  int main(int argc, char *argv[])
6.  {
7.      int fds[2];
8.      char str[]="Who are you?";
9.      char buf[BUF_SIZE];
10.     pid_t pid;
11.
12.     pipe(fds);
13.     pid=fork();
14.     if(pid==0)
15.     {
16.         write(fds[1], str, sizeof(str));
```

```
17.     }
18.     else
19.     {
20.         read(fds[0], buf, BUF_SIZE);
21.         puts(buf);
22.     }
23.     return 0;
24. }
```

- 第12行：调用pipe函数创建管道，fds数组中保存用于I/O的文件描述符。
- 第13行：接着调用fork函数。子进程将同时拥有通过第12行函数调用获取的2个文件描述符。注意！复制的并非管道，而是用于管道I/O的文件描述符。至此，父子进程同时拥有I/O文件描述符。
- 第16、20行：子进程通过第16行代码向管道传递字符串。父进程通过第20行代码从管道接收字符串。

❖ 运行结果：pipe1.c

```
root@my_linux:/tcpip# gcc pipe1.c -o pipe1
root@my_linux:/tcpip# ./pipe1
Who are you?
```

上述示例中的通信方法及路径如图11-2所示。重点在于，父子进程都可以访问管道的I/O路径，但子进程仅用输入路径，父进程仅用输出路径。

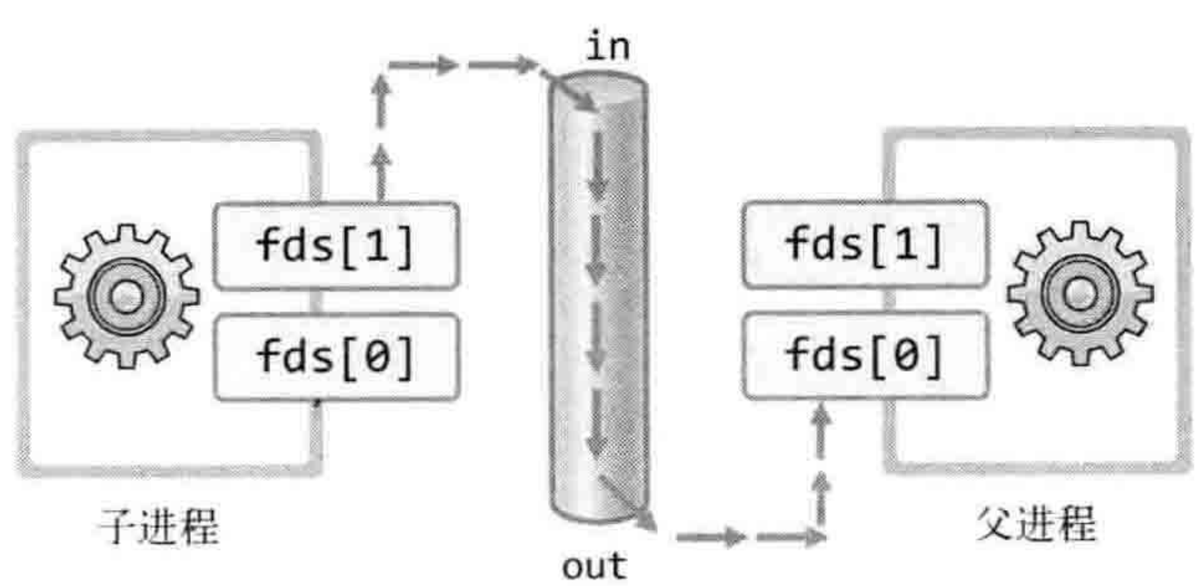



图11-2 示例pipe1.c的通信路径

以上就是管道的基本原理及通信方法。应用管道时还有一部分内容需要注意，通过双向通信示例进一步说明。

## 通过管道进行进程间双向通信

下面创建2个进程通过1个管道进行双向数据交换的示例，其通信方式如图11-3所示。

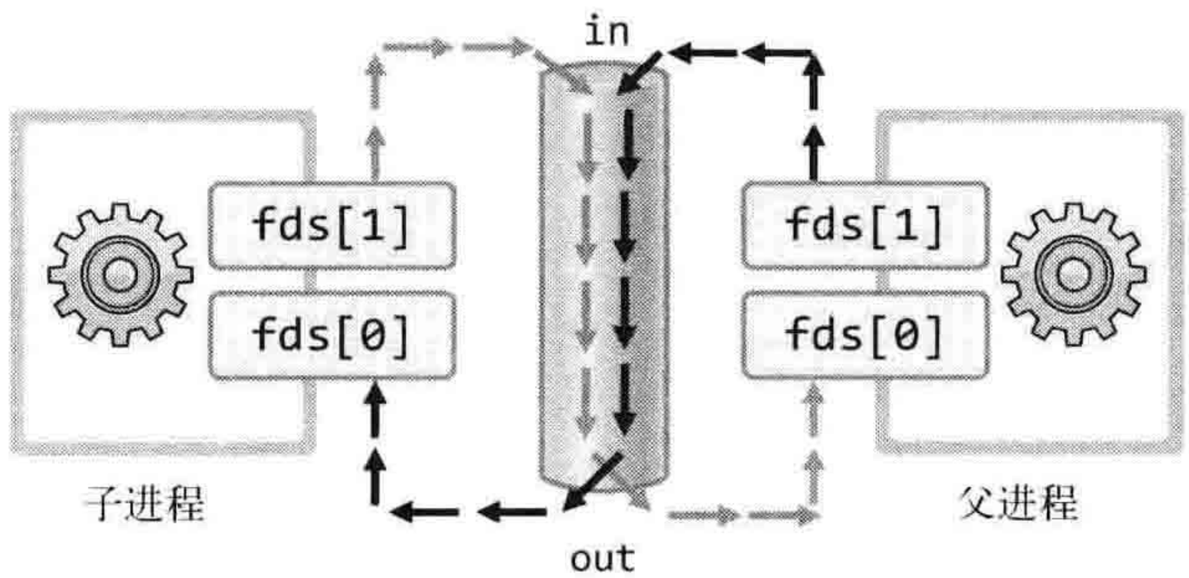



图11-3　双向通信模型1

从图11-3可看出，通过1个管道可以进行双向通信。但采用这种模型时需格外注意。先给出示例，稍后再讨论。

❖ pipe2.c

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.  #define BUF_SIZE 30
4.
5.  int main(int argc, char *argv[])
6.  {
7.      int fds[2];
8.      char str1[]="Who are you?";
9.      char str2[]="Thank you for your message";
10.     char buf[BUF_SIZE];
11.     pid_t pid;
12.
13.     pipe(fds);
14.     pid=fork();
15.     if(pid==0)
16.     {
17.         write(fds[1], str1, sizeof(str1));
18.         sleep(2);
19.         read(fds[0], buf, BUF_SIZE);
20.         printf("Child proc output: %s \n", buf);
21.     }
22.     else
23.     {
24.         read(fds[0], buf, BUF_SIZE);
25.         printf("Parent proc output: %s \n", buf);
26.         write(fds[1], str2, sizeof(str2));
27.         sleep(3);
28.     }
29.     return 0;
30. }
```

- 第17~20行：子进程运行区域。通过第17行传输数据，通过第19行接收数据。需要特别关注第18行的sleep函数。关于这一点稍后再讨论，希望各位自己思考其含义。
- 第24~26行：父进程的运行区域。通过第24行接收数据，这是为了接收第17行子进程传输的数据。另外，通过第26行传输数据，这些数据被第19行的子进程接收。
- 第27行：父进程先终止时会弹出命令提示符。这时子进程仍在工作，故不会产生问题。这条语句主要是为了防止子进程终止前弹出命令提示符（故可删除）。注释这条代码后再运行程序，各位就会明白我的意思。

❖ 运行结果：pipe2.c

```
root@my_linux:/tcpip# gcc pipe2.c -o pipe2
root@my_linux:/tcpip# ./pipe2
Parent proc output: Who are you?
Child proc output: Thank you for your message
```

运行结果应该和大家的预想一致。这次注释第18行代码后再运行（务必亲自动手操作）。虽然这行代码只将运行时间延迟了2秒，但已引发运行错误。产生原因是什么呢？

“向管道传递数据时，先读的进程会把数据取走。”

简言之，数据进入管道后成为无主数据。也就是通过read函数先读取数据的进程将得到数据，即使该进程将数据传到了管道。因此，注释第18行将产生问题。在第19行，子进程将读回自己在第17行向管道发送的数据。结果，父进程调用read函数后将无限期等待数据进入管道。

从上述示例中可以看到，只用1个管道进行双向通信并非易事。为了实现这一点，程序需要预测并控制运行流程，这在每种系统中都不同，可以视为不可能完成的任务。既然如此，该如何进行双向通信呢？

“创建2个管道。”

非常简单，1个管道无法完成双向通信任务，因此需要创建2个管道，各自负责不同的数据流动即可。其过程如图11-4所示。

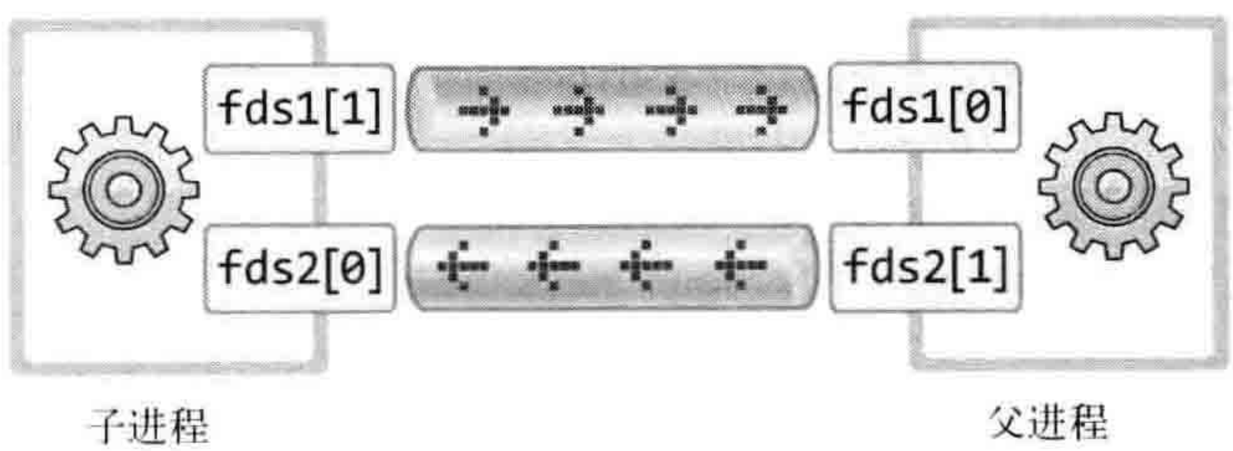



图11-4 双向通信模型2

由图11-4可知，使用2个管道可以避免程序流程的预测或控制。下面采用上述模型改进pipe2.c。

❖ pipe3.c

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.  #define BUF_SIZE 30
4.
5.  int main(int argc, char *argv[])
6.  {
7.      int fds1[2], fds2[2];
8.      char str1[]="Who are you?";
9.      char str2[]="Thank you for your message";
10.     char buf[BUF_SIZE];
11.     pid_t pid;
12.
13.     pipe(fds1), pipe(fds2);
14.     pid=fork();
15.     if(pid==0)
16.     {
17.         write(fds1[1], str1, sizeof(str1));
18.         read(fds2[0], buf, BUF_SIZE);
19.         printf("Child proc output: %s \n", buf);
20.     }
21.     else
22.     {
23.         read(fds1[0], buf, BUF_SIZE);
24.         printf("Parent proc output: %s \n", buf);
25.         write(fds2[1], str2, sizeof(str2));
26.         sleep(3);
27.     }
28.     return 0;
29. }
```

- 第13行：创建两个管道。
- 第17、23行：子进程可以通过数组fds1指向的管道向父进程传输数据。
- 第18、25行：父进程可以通过数组fds2指向的管道向子进程发送数据。
- 第26行：没有太大的意义，只是为了延迟父进程终止而插入的代码。

❖ 运行结果：pipe3.c

```
root@my_linux:/tcpip# gcc pipe3.c -o pipe3
root@my_linux:/tcpip# ./pipe3
Parent proc output: Who are you?
Child proc output: Thank you for your message
```

## 11.2　运用进程间通信

上一节学习了基于管道的进程间通信方法，接下来将其运用到网络代码中。如前所述，进程

间通信与创建服务器端并没有直接关联，但其有助于理解操作系统。

## 保存消息的回声服务器端

下面扩展第10章的echo_mpserv.c，添加如下功能：

"将回声客户端传输的字符串按序保存到文件中。"

我希望将该任务委托给另外的进程。换言之，另行创建进程，从向客户端提供服务的进程读取字符串信息。当然，该过程中需要创建用于接收数据的管道。

下面给出示例。该示例可以与任意回声客户端配合运行，但我们将用第10章介绍过的echo_mpclient.c。

❖ echo_storeserv.c

```
1.  #include <"头声明与第10章的示例echo_mpserv.c一致。">
2.  #define BUF_SIZE 100
3.  void error_handling(char *message);
4.  void read_childproc(int sig);
5.
6.  int main(int argc, char *argv[])
7.  {
8.      int serv_sock, clnt_sock;
9.      struct sockaddr_in serv_adr, clnt_adr;
10.     int fds[2];
11.
12.     pid_t pid;
13.     struct sigaction act;
14.     socklen_t adr_sz;
15.     int str_len, state;
16.     char buf[BUF_SIZE];
17.     if(argc!=2) {
18.         printf("Usage : %s <port>\n", argv[0]);
19.         exit(1);
20.     }
21.
22.     act.sa_handler=read_childproc;
23.     sigemptyset(&act.sa_mask);
24.     act.sa_flags=0;
25.     state=sigaction(SIGCHLD, &act, 0);
26.
27.     serv_sock=socket(PF_INET, SOCK_STREAM, 0);
28.     memset(&serv_adr, 0, sizeof(serv_adr));
29.     serv_adr.sin_family=AF_INET;
30.     serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
31.     serv_adr.sin_port=htons(atoi(argv[1]));
32.
33.     if(bind(serv_sock, (struct sockaddr*) &serv_adr, sizeof(serv_adr))==-1)
```

```
34.         error_handling("bind() error");
35.     if(listen(serv_sock, 5)==-1)
36.         error_handling("listen() error");
37.
38.     pipe(fds);
39.     pid=fork();
40.     if(pid==0)
41.     {
42.         FILE * fp=fopen("echomsg.txt", "wt");
43.         char msgbuf[BUF_SIZE];
44.         int i, len;
45.
46.         for(i=0; i<10; i++)
47.         {
48.             len=read(fds[0], msgbuf, BUF_SIZE);
49.             fwrite((void*)msgbuf, 1, len, fp);
50.         }
51.         fclose(fp);
52.         return 0;
53.     }
54.
55.     while(1)
56.     {
57.         adr_sz=sizeof(clnt_adr);
58.         clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_adr, &adr_sz);
59.         if(clnt_sock==-1)
60.             continue;
61.         else
62.             puts("new client connected...");
63.
64.         pid=fork();
65.         if(pid==0)
66.         {
67.             close(serv_sock);
68.             while((str_len=read(clnt_sock, buf, BUF_SIZE))!=0)
69.             {
70.                 write(clnt_sock, buf, str_len);
71.                 write(fds[1], buf, str_len);
72.             }
73.
74.             close(clnt_sock);
75.             puts("client disconnected...");
76.             return 0;
77.         }
78.         else
79.             close(clnt_sock);
80.     }
81.     close(serv_sock);
82.     return 0;
83. }
84.
85. void read_childproc(int sig)
86. {
87.     //与示例echo_mpserv.c一致，故省略。
```

```
88. }
89. void error_handling(char *message)
90. {
91.     //与示例echo_mpserv.c一致，故省略。
92. }
```

- 第38、39行：第38行创建管道，第39行创建负责保存文件的进程。
- 第40~53行：第39行创建的子进程运行区域。该区域从管道出口fds[0]读取数据并保存到文件中。另外，上述服务器端并不终止运行，而是不断向客户端提供服务。因此，数据在文件中累计到一定程度即关闭文件，该过程通过第46行的循环完成。
- 第71行：第64行通过fork函数创建的所有子进程将复制第38行创建的管道的文件描述符。因此，可以通过管道入口fds[1]传递字符串信息。

❖ 运行结果：echo_storeserv.c

```
root@my_linux:/tcpip# gcc echo_storeserv.c -o serv
root@my_linux:/tcpip# ./serv 9190
new client connected...
new client connected...
removed proc id: 7177
client disconnected...
removed proc id: 7185
client disconnected...
removed proc id: 7191
```

❖ 运行结果：echo_mpclient.c one

```
root@my_linux:/tcpip# gcc echo_mpclient.c -o client
root@my_linux:/tcpip# ./client 127.0.0.1 9190
One
Message from server: One
Three
Message from server: Three
Five
Message from server: Five
Seven
Message from server: Seven
Nine
Message from server: Nine
Q
root@com:/home/swyoon/tcpip#
```

❖ 运行结果：echo_mpclient.c two

```
root@my_linux:/tcpip# ./client 127.0.0.1 9190
Two
Message from server: Two
Four
Message from server: Four
Six
Message from server: Six
Eight
Message from server: Eight
Ten
Message from server: Ten
Q
root@my_linux:/tcpip#
```

如上运行结果所示，希望各位也启动多个客户端向服务器端传输字符串。文件中累计一定数量的字符串后（共10次的fwrite函数调用完成后），可以打开echomsg.txt验证保存的字符串。

**各位应该想到了重构**

观察示例 echo_storeserv.c 后，各位可能认为应该针对一部分功能以函数为单位重构代码。我只是在扩展回声服务器端的过程中为了便于学习（有利于跟之前的代码进行比较）而未改变代码框架，因此大家才会看到向下展开的代码像小说一样。

## 如果想构建更大型的程序

前面已经讲解了并发服务器的第一种实现模型，但各位或许有如下想法：

“我想利用进程和管道编写聊天室程序，使多个客户端进行对话，应该从哪着手呢？”

若想仅用进程和管道构建具有复杂功能的服务器端，程序员需要具备熟练的编程技能和经验。因此，初学者应用该模型扩展程序并非易事，希望大家不要过于拘泥。以后要说明的另外两种模型在功能上更加强大，同时更容易实现我们的想法。

“那前面讲的内容算什么啊？是不是没用呢？”

我曾被多次问及此类问题。各位通过第10章和本章内容理解了操作系统的基本内容，同时也

是学习线程必备的前期知识——进程。而且掌握了多进程代码的基本分析能力。即使各位不会亲自利用多进程构建服务器端，但这些都值得学习。可以将我的个人经验告诉所有读者朋友：

> “即使开始时只想学习必要部分，最后也会需要掌握所有内容。”

## 11.3 习题

(1) 什么是进程间通信？分别从概念上和内存的角度进行说明。

(2) 进程间通信需要特殊的IPC机制，这是由操作系统提供的。进程间通信时为何需要操作系统的帮助？

(3) “管道”是典型的IPC技法。关于管道，请回答如下问题。

a. 管道是进程间交换数据的路径。如何创建此路径？由谁创建？
b. 为了完成进程间通信，2个进程需同时连接管道。那2个进程如何连接到同一管道？
c. 管道允许进行2个进程间的双向通信。双向通信中需要注意哪些内容？

(4) 编写示例复习IPC技法，使2个进程相互交换3次字符串。当然，这2个进程应具有父子关系，各位可指定任意字符串。