```
4.  #include <unistd.h>
5.  #include <arpa/inet.h>
6.  #include <sys/socket.h>
7.
8.  #define BUF_SIZE 1024
9.  void error_handling(char *message);
10.
11. int main(int argc, char *argv[])
12. {
13.     int sock;
14.     char message[BUF_SIZE];
15.     int str_len;
16.     struct sockaddr_in serv_adr;
17.
18.     if(argc!=3) {
19.         printf("Usage : %s <IP> <port>\n", argv[0]);
20.         exit(1);
21.     }
22.
23.     sock=socket(PF_INET, SOCK_STREAM, 0);
24.     if(sock==-1)
25.         error_handling("socket() error");
26.
27.     memset(&serv_adr, 0, sizeof(serv_adr));
28.     serv_adr.sin_family=AF_INET;
29.     serv_adr.sin_addr.s_addr=inet_addr(argv[1]);
30.     serv_adr.sin_port=htons(atoi(argv[2]));
31.
32.     if(connect(sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr))==-1)
33.         error_handling("connect() error!");
34.     else
35.         puts("Connected.....");
36.
37.     while(1)
38.     {
39.         fputs("Input message(Q to quit): ", stdout);
40.         fgets(message, BUF_SIZE, stdin);
41.
42.         if(!strcmp(message,"q\n") || !strcmp(message,"Q\n"))
43.             break;
44.
45.         write(sock, message, strlen(message));
46.         str_len=read(sock, message, BUF_SIZE-1);
47.         message[str_len]=0;
48.         printf("Message from server: %s", message);
49.     }
50.     close(sock);
51.     return 0;
52. }
53.
54. void error_handling(char *message)
55. {
56.     fputs(message, stderr);
57.     fputc('\n', stderr);
```

```
58.     exit(1);
59. }
```

- 第32行：调用connect函数。若调用该函数引起的连接请求被注册到服务器端等待队列，则connect函数将完成正常调用。因此，即使通过第35行代码输出了连接提示字符串——如果服务器尚未调用accept函数——也不会真正建立服务关系。
- 第50行：调用close函数向相应套接字发送EOF（EOF即意味着中断连接）。

❖ 运行结果：echo_client.c

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

我们编写的回声服务器端/客户端以字符串为单位传递数据。理解这一点后再观察echo_client.c第45行和第46行。各位若已完全掌握了之前讲过的TCP，就会意识到这2行代码不太适合做字符串单位的回声。

回声客户端存在的问题

下列是echo_client.c的第45~48行代码。

```
write(sock, message, strlen(message));
str_len = read(sock, message, BUF_SIZE - 1);
message[str_len] = 0;
printf("Message from server: %s", message);
```

以上代码有个错误假设：

“每次调用read、write函数时都会以字符串为单位执行实际的I/O操作。”

当然，每次调用write函数都会传递1个字符串，因此这种假设在某种程度上也算合理。但大家还记得第2章中“TCP不存在数据边界“的内容吗？上述客户端是基于TCP的，因此，多次调用write函数传递的字符串有可能一次性传递到服务器端。此时客户端有可能从服务器端收到多个字符串，这不是我们希望看到的结果。还需考虑服务器端的如下情况：

“字符串太长，需要分2个数据包发送！”

服务器端希望通过调用1次write函数传输数据，但如果数据太大，操作系统就有可能把数据分成多个数据包发送到客户端。另外，在此过程中，客户端有可能在尚未收到全部数据包时就调用read函数。

所有这些问题都源自TCP的数据传输特性。那该如何解决呢？答案请见第5章。

“但上述示例不是正常运转了吗？”

当然，我们的回声服务器端/客户端给出的结果是正确的。但这只是运气好罢了！只是因为收发的数据小，而且运行环境为同一台计算机或相邻的两台计算机，所以没发生错误，可实际上仍存在发生错误的可能。

4.4 基于 Windows 的实现

随着本书学习的深入，Windows和Linux的平台差异将愈加明显。但至少现在还不大，所以很容易将Linux示例移植到Windows平台。

基于 Windows 的回声服务器端

为了将Linux平台下的示例转化成Windows平台示例，需要记住以下4点。

- 通过WSAStartup、WSACleanup函数初始化并清除套接字相关库。
- 把数据类型和变量名切换为Windows风格。
- 数据传输中用recv、send函数而非read、write函数。
- 关闭套接字时用closesocket函数而非close函数。

接下来给出基于Windows的回声服务器端。只需更改如上4点，故省略。

❖ echo_server_win.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <winsock2.h>
5.
6.  #define BUF_SIZE 1024
7.  void ErrorHandling(char *message);
8.
9.  int main(int argc, char *argv[])
10. {
11.     WSADATA wsaData;
12.     SOCKET hServSock, hClntSock;
13.     char message[BUF_SIZE];
14.     int strLen, i;
15.
```

```
16.     SOCKADDR_IN servAdr, clntAdr;
17.     int clntAdrSize;
18.
19.     if(argc!=2) {
20.         printf("Usage : %s <port>\n", argv[0]);
21.         exit(1);
22.     }
23.
24.     if(WSAStartup(MAKEWORD(2, 2), &wsaData)!=0)
25.         ErrorHandling("WSAStartup() error!");
26.
27.     hServSock=socket(PF_INET, SOCK_STREAM, 0);
28.     if(hServSock==INVALID_SOCKET)
29.         ErrorHandling("socket() error");
30.
31.     memset(&servAdr, 0, sizeof(servAdr));
32.     servAdr.sin_family=AF_INET;
33.     servAdr.sin_addr.s_addr=htonl(INADDR_ANY);
34.     servAdr.sin_port=htons(atoi(argv[1]));
35.
36.     if(bind(hServSock, (SOCKADDR*)&servAdr, sizeof(servAdr))==SOCKET_ERROR)
37.         ErrorHandling("bind() error");
38.
39.     if(listen(hServSock, 5)==SOCKET_ERROR)
40.         ErrorHandling("listen() error");
41.
42.     clntAdrSize=sizeof(clntAdr);
43.
44.     for(i=0; i<5; i++)
45.     {
46.         hClntSock=accept(hServSock, (SOCKADDR*)&clntAdr, &clntAdrSize);
47.         if(hClntSock==-1)
48.             ErrorHandling("accept() error");
49.         else
50.             printf("Connected client %d \n", i+1);
51.
52.         while((strLen=recv(hClntSock, message, BUF_SIZE, 0))!=0)
53.             send(hClntSock, message, strLen, 0);
54.
55.         closesocket(hClntSock);
56.     }
57.     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. }
```

基于 Windows 的回声客户端

回声客户端的移植过程也与服务器端类似，因此同样只给出代码。

❖ echo_client_win.c

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

#define BUF_SIZE 1024
void ErrorHandling(char *message);

int main(int argc, char *argv[])
{
    WSADATA wsaData;
    SOCKET hSocket;
    char message[BUF_SIZE];
    int strLen;
    SOCKADDR_IN servAdr;

    if(argc!=3) {
        printf("Usage : %s <IP> <port>\n", argv[0]);
        exit(1);
    }

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

    hSocket=socket(PF_INET, SOCK_STREAM, 0);
    if(hSocket==INVALID_SOCKET)
        ErrorHandling("socket() error");

    memset(&servAdr, 0, sizeof(servAdr));
    servAdr.sin_family=AF_INET;
    servAdr.sin_addr.s_addr=inet_addr(argv[1]);
    servAdr.sin_port=htons(atoi(argv[2]));

    if(connect(hSocket, (SOCKADDR*)&servAdr, sizeof(servAdr))==SOCKET_ERROR)
        ErrorHandling("connect() error!");
    else
        puts("Connected.....");

    while(1)
    {
        fputs("Input message(Q to quit): ", stdout);
        fgets(message, BUF_SIZE, stdin);

        if(!strcmp(message,"q\n") || !strcmp(message,"Q\n"))
            break;

```

```
47.         send(hSocket, message, strlen(message), 0);
48.         strLen=recv(hSocket, message, BUF_SIZE-1, 0);
49.         message[strLen]=0;
50.         printf("Message from server: %s", message);
51.     }
52.     closesocket(hSocket);
53.     WSACleanup();
54.     return 0;
55. }
56.
57. void ErrorHandling(char *message)
58. {
59.     fputs(message, stderr);
60.     fputc('\n', stderr);
61.     exit(1);
62. }
```

运行结果也跟之前的回声服务器端/客户端相同，服务器端处理完第一个客户端请求，正向第二个客户端提供服务。

❖ 运行结果：echo_server_win.c

```
C:\tcpip> server 9190
Connected client 1
Connected client 2
```

下列代码第一段表示第一个客户端连接到回声服务器端，接收服务并终止连接；第二段表示正在接受回声服务器端服务的第二个客户端。

❖ 运行结果：echo_client_win.c one

```
C:\tcpip> client 127.0.0.1 9190
Connected.....
Input message(Q to quit): I really
Message from server: I really
Input message(Q to quit): Q
```

❖ 运行结果：echo_client_win.c two

```
C:\tcpip> client 127.0.0.1 9190
Connected.....
Input message(Q to quit): 我真的
Message from server: 我真的
Input message(Q to quit):
```

对回声服务器端/客户端迭代模型的讲解到此结束，希望各位理解好本章回声客户端存在的问题后再进入第5章。

4.5 习题

(1) 请说明TCP/IP的4层协议栈，并说明TCP和UDP套接字经过的层级结构差异。

(2) 请说出TCP/IP协议栈中链路层和IP层的作用，并给出二者关系。

(3) 为何需要把TCP/IP协议栈分成4层（或7层）？结合开放式系统回答。

(4) 客户端调用connect函数向服务器端发送连接请求。服务器端调用哪个函数后，客户端可以调用connect函数？

(5) 什么时候创建连接请求等待队列？它有何作用？与accept有什么关系？

(6) 客户端中为何不需要调用bind函数分配地址？如果不调用bind函数，那何时、如何向套接字分配IP地址和端口号？

(7) 把第1章的hello_server.c和hello_server_win.c改成迭代服务器端，并利用客户端测试更改是否准确。

第5章

基于TCP的服务器端/客户端（2）

5

第4章通过回声服务示例讲解了TCP服务器端/客户端的实现方法。但这仅是从编程角度的学习，我们尚未详细讨论TCP的工作原理。因此，本章将详细讲解TCP中必要的理论知识，还将给出第4章客户端问题的解决方案。

5.1 回声客户端的完美实现

第4章已分析过回声客户端存在的问题，此处不再赘述。如果大家不太理解，请复习第2章的TCP传输特性和第4章的内容。

回声服务器端没有问题，只有回声客户端有问题？

问题不在服务器端，而在客户端。但只看代码也许不太好理解，因为I/O中使用了相同的函数。先回顾一下回声服务器端的I/O相关代码，下面是echo_server.c的第50~51行代码。

```
while((str_len = read(clnt_sock, message, BUF_SIZE)) != 0)
    write(clnt_sock, message, str_len);
```

接着回顾回声客户端代码，下面是echo_client.c的第45~46行代码。

```
write(sock, message, strlen(message));
str_len = read(sock, message, BUF_SIZE - 1);
```

二者都在循环调用read或write函数。实际上之前的回声客户端将100%接收自己传输的数据，只不过接收数据时的单位有些问题。扩展客户端代码回顾范围，下面是echo_client.c第37行开始

的代码。

```
while(1)
{
    fputs("Input message(Q to quit): ", stdout);
    fgets(message, BUF_SIZE, stdin);
    ....
    write(sock, message, strlen(message));
    str_len = read(sock, message, BUF_SIZE - 1);
    message[str_len] = 0;
    printf("Message from server: %s", message);
}
```

大家现在理解了吧？回声客户端传输的是字符串，而且是通过调用write函数一次性发送的。之后还调用一次read函数，期待着接收自己传输的字符串。这就是问题所在。

“既然回声客户端会收到所有字符串数据，是否只需多等一会儿？过一段时间后再调用read函数是否可以一次性读取所有字符串数据？”

的确，过一段时间后即可接收，但需要等多久？要等10分钟吗？这不符合常理，理想的客户端应在收到字符串数据时立即读取并输出。

回声客户端问题解决方法

我说的回声客户端问题实际上是初级程序员经常犯的错误，其实很容易解决，因为可以提前确定接收数据的大小。若之前传输了20字节长的字符串，则在接收时循环调用read函数读取20个字节即可。既然有了解决方法，接下来给出其代码。

❖ echo_client2.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <unistd.h>
5.  #include <arpa/inet.h>
6.  #include <sys/socket.h>
7.  #define BUF_SIZE 1024
8.  void error_handling(char *message);
9.
10. int main(int argc, char *argv[])
11. {
12.     int sock;
13.     char message[BUF_SIZE];
14.     int str_len, recv_len, recv_cnt;
```

```
15.     struct sockaddr_in serv_adr;
16.
17.     if(argc!=3) {
18.         printf("Usage : %s <IP> <port>\n", argv[0]);
19.         exit(1);
20.     }
21.
22.     sock=socket(PF_INET, SOCK_STREAM, 0);
23.     if(sock==-1)
24.         error_handling("socket() error");
25.
26.     memset(&serv_adr, 0, sizeof(serv_adr));
27.     serv_adr.sin_family=AF_INET;
28.     serv_adr.sin_addr.s_addr=inet_addr(argv[1]);
29.     serv_adr.sin_port=htons(atoi(argv[2]));
30.
31.     if(connect(sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr))==-1)
32.         error_handling("connect() error!");
33.     else
34.         puts("Connected.....");
35.
36.     while(1)
37.     {
38.         fputs("Input message(Q to quit): ", stdout);
39.         fgets(message, BUF_SIZE, stdin);
40.         if(!strcmp(message,"q\n") || !strcmp(message,"Q\n"))
41.             break;
42.
43.         str_len=write(sock, message, strlen(message));
44.
45.         recv_len=0;
46.         while(recv_len<str_len)
47.         {
48.             recv_cnt=read(sock, &message[recv_len], BUF_SIZE-1);
49.             if(recv_cnt==-1)
50.                 error_handling("read() error!");
51.             recv_len+=recv_cnt;
52.         }
53.         message[recv_len]=0;
54.         printf("Message from server: %s", message);
55.     }
56.     close(sock);
57.     return 0;
58. }
59.
60. void error_handling(char *message)
61. {
62.     fputs(message, stderr);
63.     fputc('\n', stderr);
64.     exit(1);
65. }
```

以上代码第43~53行是变更及添加的部分。之前的示例仅调用1次read函数，上述示例为了接

收所有传输数据而循环调用read函数。另外，代码第46行循环可以写成如下形式，可能这种方式更容易理解。

```
while(recv_len != str_len)
{
    ....
}
```

接收的数据大小应和传输的相同，因此，recv_len中保存的值等于str_len中保存的值时，即可跳出while循环。也许各位认为这种循环写法更符合逻辑，但有可能引发无限循环。假设发生异常情况，读取数据过程中recv_len超过str_len，此时就无法退出循环。而如果while循环写成下面这种形式，则即使发生异常也不会陷入无限循环。

```
while(recv_len < str_len)
{
    ....
}
```

写循环语句时应尽量降低因异常情况而陷入无限循环的可能。以上示例可以结合第4章的echo_server.c运行。各位已经非常熟悉运行结果，故省略。

如果问题不在于回声客户端：定义应用层协议

回声客户端可以提前知道接收的数据长度，但我们应该意识到，更多情况下这不太可能。既然如此，若无法预知接收数据长度时应如何收发数据？此时需要的就是应用层协议的定义。之前的回声服务器端/客户端中定义了如下协议。

“收到Q就立即终止连接。”

同样，收发数据过程中也需要定好规则（协议）以表示数据的边界，或提前告知收发数据的大小。服务器端/客户端实现过程中逐步定义的这些规则集合就是应用层协议。可以看出，应用层协议并不是高深莫测的存在，只不过是为特定程序的实现而制定的规则。

下面编写程序以体验应用层协议的定义过程。该程序中，服务器端从客户端获得多个数字和运算符信息。服务器端收到数字后对其进行加减乘运算，然后把结果传回客户端。例如，向服务器端传递3、5、9的同时请求加法运算，则客户端收到3+5+9的运算结果；若请求做乘法运算，则客户端收到3×5×9的运算结果。而如果向服务器端传递4、3、2的同时要求做减法，则客户端将收到4-3-2的运算结果，即第一个参数成为被减数。

请各位根据以上要求编写服务器端/客户端，细节部分可以自定义。我实现的程序运行结果如下。先给出服务器端运行结果。

❖ 运行结果：op_server.c

```
root@my_linux:/tcpip# gcc op_server.c -o opserver
root@my_linux:/tcpip# ./opserver 9190
```

可以看出，服务器端的运行结果并没有特别之处。可以通过如下客户端运行结果了解程序运行原理。

❖ 运行结果：op_client.c one

```
root@my_linux:/tcpip# gcc op_client.c -o opclient
root@my_linux:/tcpip# ./opclient 127.0.0.1 9190
Connected.....
Operand count: 3
Operand 1: 12
Operand 2: 24
Operand 3: 36
Operator: +
Operation result: 72
```

从运行结果可以看出，客户端首先询问用户待算数字的个数，再输入相应个数的整数，最后以运算符的形式输入运算符号信息，并输出运算结果（+、-、*之一）。当然，实际的运算是由服务器端做的，客户端只输出运算结果。为了更准确地理解，再给出1个客户端运行结果。这次是请求2个数的减法运算。

❖ 运行结果：op_client.c two

```
root@my_linux:/tcpip# ./opclient 127.0.0.1 9190
Connected.....
Operand count: 2
Operand 1: 24
Operand 2: 12
Operator: -
Operation result: 12
```

运行结果并不一定要和我的一致，如果各位有更好的运行模型，可以之为基础编写示例。

计算器服务器端/客户端示例

各位尝试实现了吗？它在功能上没有特别之处，但若想在网络环境下实现这些功能并非易事。特别是不熟悉C语言中的数组及指针应用的人，会在实现程序功能时吃苦头。因此，我希望

通过本示例补充回声服务器端/客户端实现中未涉及的部分。但如前所述，如果可能，还是希望大家自己动手实现。若成功实现（而不是看源代码理解），将有助于各位提升自信。

我编写程序前设计了如下应用层协议，但这只是为实现程序而设计的最低协议，实际的应用程序实现中需要的协议更详细、准确。

- 客户端连接到服务器端后以1字节整数形式传递待算数字个数。
- 客户端向服务器端传递的每个整数型数据占用4字节。
- 传递整数型数据后接着传递运算符。运算符信息占用1字节。
- 选择字符+、-、*之一传递。
- 服务器端以4字节整数型向客户端传回运算结果。
- 客户端得到运算结果后终止与服务器端的连接。

这种程度的协议相当于实现了一半程序，这也说明应用层协议设计在网络编程中的重要性。只要设计好协议，实现就不会成为大问题。另外，之前也讲过，调用close函数将向对方传递EOF，请各位记住这一点并加以运用。接下来给出我实现的计算器客户端代码。实际上，与服务器端相比，客户端中有更多需要学习的内容。

❖ op_client.c

```
1.  #include <"与其他示例的头声明相同，故省略。">
2.  #define BUF_SIZE 1024
3.  #define RLT_SIZE 4
4.  #define OPSZ 4
5.  void error_handling(char *message);
6.
7.  int main(int argc, char *argv[])
8.  {
9.      int sock;
10.     char opmsg[BUF_SIZE];
11.     int result, opnd_cnt, i;
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.     sock=socket(PF_INET, SOCK_STREAM, 0);
19.     if(sock==-1)
20.         error_handling("socket() error");
21.
22.     memset(&serv_adr, 0, sizeof(serv_adr));
23.     serv_adr.sin_family=AF_INET;
24.     serv_adr.sin_addr.s_addr=inet_addr(argv[1]);
25.     serv_adr.sin_port=htons(atoi(argv[2]));
26.
27.     if(connect(sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr))==-1)
28.         error_handling("connect() error!");
```

```
29.     else
30.         puts("Connected.....");
31.
32.     fputs("Operand count: ", stdout);
33.     scanf("%d", &opnd_cnt);
34.     opmsg[0]=(char)opnd_cnt;
35.
36.     for(i=0; i<opnd_cnt; i++)
37.     {
38.         printf("Operand %d: ", i+1);
39.         scanf("%d", (int*)&opmsg[i*OPSZ+1]);
40.     }
41.     fgetc(stdin);
42.     fputs("Operator: ", stdout);
43.     scanf("%c", &opmsg[opnd_cnt*OPSZ+1]);
44.     write(sock, opmsg, opnd_cnt*OPSZ+2);
45.     read(sock, &result, RLT_SIZE);
46.
47.     printf("Operation result: %d \n", result);
48.     close(sock);
49.     return 0;
50. }
51.
52. void error_handling(char *message)
53. {
54.     //与其他示例的error_handling函数相同，故省略。
55. }
```

代码说明

- 第3、4行：将待算数字的字节数和运算结果的字节数设为常数。
- 第10行：为收发数据准备的内存空间，需要数据积累到一定程度后再收发，因此通过数组创建。
- 第33、34行：从程序用户的输入中得到待算数个数后，保存至数组opmsg。强制转换成char类型，因为协议规定待算数个数应通过1字节整数型传递，因此不能超过1字节整数型能够表示的范围。该示例中用的是有符号整数型，但待算数个数不能是负数，因此使用无符号整数型更合理。
- 第36~40行：从程序用户的输入中得到待算整数，保存到数组opmsg。4字节int型数据要保存到char数组，因而转换成int指针类型。若不太理解此部分，应单独复习指针。
- 第41行：第43行中需输入字符，在此之前调用fgetc函数删掉缓冲中的字符\n。
- 第43行：最后输入运算符信息，保存到opmsg数组。
- 第44行：调用write函数一次性传输opmsg数组中的运算相关信息。可以调用1次write函数进行传输，也可以分成多次调用。前面反复强调过，这是因为TCP中不存在数据边界。
- 第45行：保存服务器端传输的运算结果。待接收的数据长度为4字节，因此调用1次read函数即可接收。

客户端实现的讲解到此结束，最后给出客户端向服务器端传输的数据结构示例，如图5-1所示。

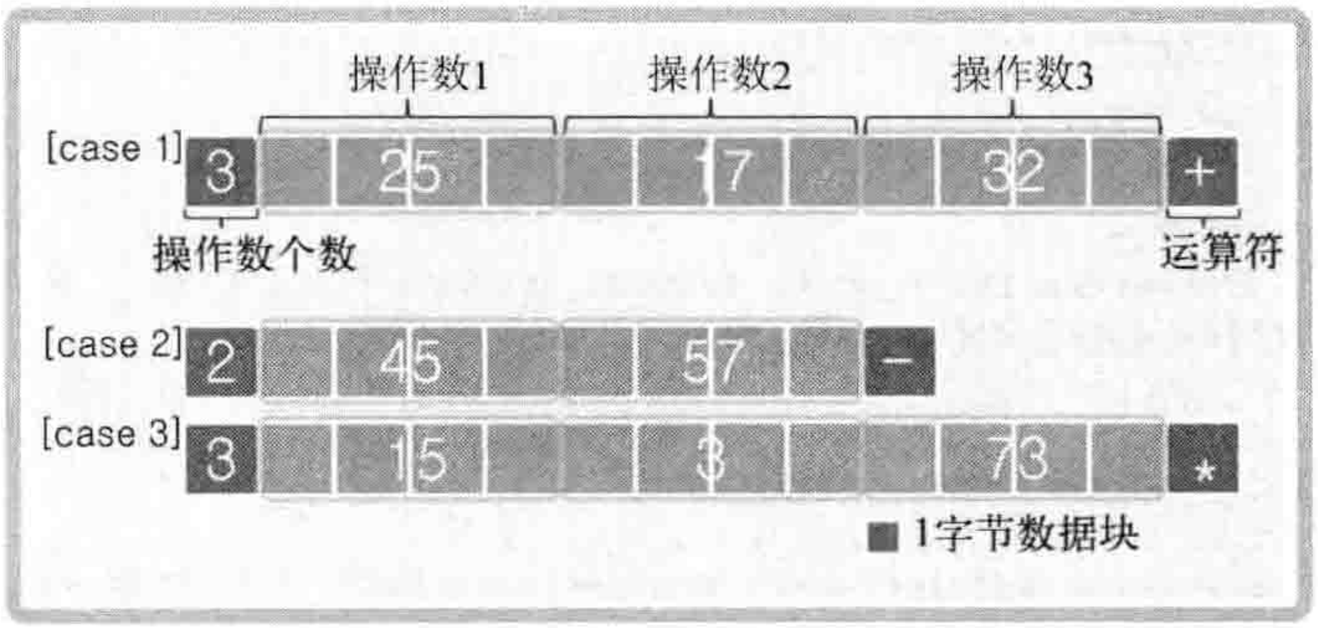


图5-1 客户端op_client.c的数据传送格式

从图5-1中可以看出，若想在同一数组中保存并传输多种数据类型，应把数组声明为char类型。而且需要额外做一些指针及数组运算。接下来给出服务器端代码。

❖ op_server.c

```
1.  #include <"与其他示例的头声明相同，故省略。">
2.  #define BUF_SIZE 1024
3.  #define OPSZ 4
4.  void error_handling(char *message);
5.  int calculate(int opnum, int opnds[], char oprator);
6.
7.  int main(int argc, char *argv[])
8.  {
9.      int serv_sock, clnt_sock;
10.     char opinfo[BUF_SIZE];
11.     int result, opnd_cnt, i;
12.     int recv_cnt, recv_len;
13.     struct sockaddr_in serv_adr, clnt_adr;
14.     socklen_t clnt_adr_sz;
15.     if(argc!=2) {
16.         printf("Usage : %s <port>\n", argv[0]);
17.         exit(1);
18.     }
19.
20.     serv_sock=socket(PF_INET, SOCK_STREAM, 0);
21.     if(serv_sock==-1)
22.         error_handling("socket() error");
23.
24.     memset(&serv_adr, 0, sizeof(serv_adr));
25.     serv_adr.sin_family=AF_INET;
26.     serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
27.     serv_adr.sin_port=htons(atoi(argv[1]));
28.
29.     if(bind(serv_sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr))==-1)
30.         error_handling("bind() error");
31.     if(listen(serv_sock, 5)==-1)
32.         error_handling("listen() error");
```

```
33.     clnt_adr_sz=sizeof(clnt_adr);
34.
35.     for(i=0; i<5; i++)
36.     {
37.         opnd_cnt=0;
38.         clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_adr, &clnt_adr_sz);
39.         read(clnt_sock, &opnd_cnt, 1);
40.
41.         recv_len=0;
42.         while((opnd_cnt*OPSZ+1)>recv_len)
43.         {
44.             recv_cnt=read(clnt_sock, &opinfo[recv_len], BUF_SIZE-1);
45.             recv_len+=recv_cnt;
46.         }
47.         result=calculate(opnd_cnt, (int*)opinfo, opinfo[recv_len-1]);
48.         write(clnt_sock, (char*)&result, sizeof(result));
49.         close(clnt_sock);
50.     }
51.     close(serv_sock);
52.     return 0;
53. }
54.
55. int calculate(int opnum, int opnds[], char op)
56. {
57.     int result=opnds[0], i;
58.     switch(op)
59.     {
60.     case '+':
61.         for(i=1; i<opnum; i++) result+=opnds[i];
62.         break;
63.     case '-':
64.         for(i=1; i<opnum; i++) result-=opnds[i];
65.         break;
66.     case '*':
67.         for(i=1; i<opnum; i++) result*=opnds[i];
68.         break;
69.     }
70.     return result;
71. }
72.
73. void error_handling(char *message)
74. {
75.     //与其他示例的error_handling函数相同，故省略。
76. }
```

代码说明

- 第35行：为了接收5个客户端的连接请求而编写的for语句。
- 第39行：首先接收待算数个数。
- 第42~46行：根据第39行中的待算数个数接收待算数。
- 第47行：调用calculate函数的同时传递待算数和运算符信息参数。
- 第48行：向客户端传输calculate函数返回的运算结果。

对计算器服务器端/客户端的讲解到此结束，部分读者可能略感困难，但稍加努力就能理解。

5.2 TCP 原理

我本想在此结束TCP相关介绍，但又觉得稍显仓促，所以补充讲解TCP的理论部分。本节内容将成为日后理解套接字选项（第9章）的基础，希望大家能够全部掌握。

TCP 套接字中的 I/O 缓冲

如前所述，TCP套接字的数据收发无边界。服务器端即使调用1次write函数传输40字节的数据，客户端也有可能通过4次read函数调用每次读取10字节。但此处也有一些疑问，服务器端一次性传输了40字节，而客户端居然可以缓慢地分批接收。客户端接收10字节后，剩下的30字节在何处等候呢？是不是像飞机为等待着陆而在空中盘旋一样，剩下30字节也在网络中徘徊并等待接收呢？

实际上，write函数调用后并非立即传输数据，read函数调用后也并非马上接收数据。更准确地说，如图5-2所示，write函数调用瞬间，数据将移至输出缓冲；read函数调用瞬间，从输入缓冲读取数据。

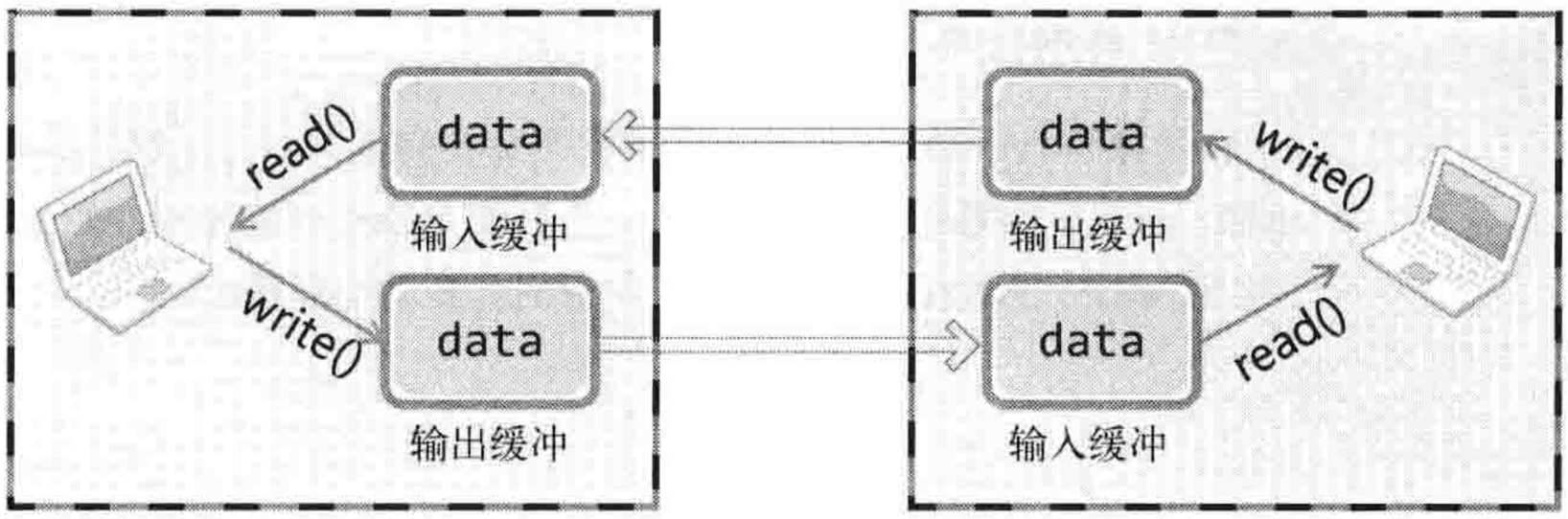


图5-2 TCP套接字的I/O缓冲

如图5-2所示，调用write函数时，数据将移到输出缓冲，在适当的时候（不管是分别传送还是一次性传送）传向对方的输入缓冲。这时对方将调用read函数从输入缓冲读取数据。这些I/O缓冲特性可整理如下。

- I/O缓冲在每个TCP套接字中单独存在。
- I/O缓冲在创建套接字时自动生成。
- 即使关闭套接字也会继续传递输出缓冲中遗留的数据。
- 关闭套接字将丢失输入缓冲中的数据。

那么，下面这种情况会引发什么事情？理解了I/O缓冲后，各位应该可以猜出其流程：

“客户端输入缓冲为50字节，而服务器端传输了100字节。”

这的确是个问题。输入缓冲只有50字节，却收到了100字节的数据。可以提出如下解决方案：

“填满输入缓冲前迅速调用read函数读取数据，这样会腾出一部分空间，问题就解决了。”

当然，这只是我的一个小玩笑，相信大家不会当真，那么马上给出结论：

“不会发生超过输入缓冲大小的数据传输。”

也就是说，根本不会发生这类问题，因为TCP会控制数据流。TCP中有滑动窗口（Sliding Window）协议，用对话方式呈现如下。

- 套接字A：“你好，最多可以向我传递50字节。”
- 套接字B：“OK!”
- 套接字A：“我腾出了20字节的空间，最多可以收70字节。”
- 套接字B：“OK!”

数据收发也是如此，因此TCP中不会因为缓冲溢出而丢失数据。

从 write 函数返回的时间点

write 函数和 Windows 的 send 函数并不会在完成向对方主机的数据传输时返回，而是在数据移到输出缓冲时。但 TCP 会保证对输出缓冲数据的传输，所以说 write 函数在数据传输完成时返回。要准确理解这句话。

TCP 内部工作原理 1：与对方套接字的连接

TCP套接字从创建到消失所经过程分为如下3步。

- 与对方套接字建立连接。
- 与对方套接字进行数据交换。
- 断开与对方套接字的连接。

首先讲解与对方套接字建立连接的过程。连接过程中套接字之间的对话如下。

- [Shake 1] 套接字A：“你好，套接字B。我这儿有数据要传给你，建立连接吧。”
- [Shake 2] 套接字B：“好的，我这边已就绪。”
- [Shake 3] 套接字A：“谢谢你受理我的请求。”

TCP在实际通信过程中也会经过3次对话过程，因此，该过程又称Three-way handshaking（三次握手）。接下来给出连接过程中实际交换的信息格式，如图5-3所示。

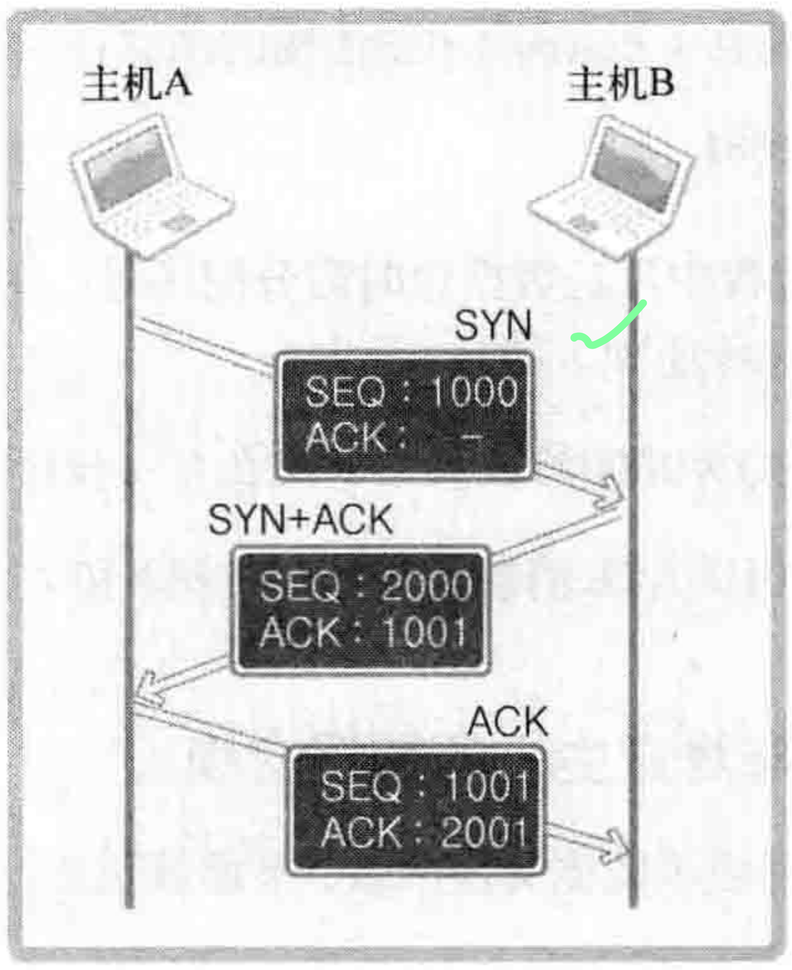


图5-3 TCP套接字的连接设置过程

套接字是以全双工（Full-duplex）方式工作的。也就是说，它可以双向传递数据。因此，收发数据前需要做一些准备。首先，请求连接的主机A向主机B传递如下信息：

```
[SYN] SEQ: 1000, ACK: -
```

该消息中SEQ为1000，ACK为空，而SEQ为1000的含义如下：

“现传递的数据包序号为1000，如果接收无误，请通知我向您传递1001号数据包。”

这是首次请求连接时使用的消息，又称SYN。SYN是Synchronization的简写，表示收发数据前传输的同步消息。接下来主机B向A传递如下消息：

```
[SYN+ACK] SEQ: 2000, ACK: 1001
```

此时SEQ为2000，ACK为1001，而SEQ为2000的含义如下：

“现传递的数据包序号为2000，如果接收无误，请通知我向您传递2001号数据包。”

而ACK 1001的含义如下：

“刚才传输的SEQ为1000的数据包接收无误，现在请传递SEQ为1001的数据包。”

对主机A首次传输的数据包的确认消息（ACK 1001）和为主机B传输数据做准备的同步消息（SEQ 2000）捆绑发送，因此，此种类型的消息又称SYN+ACK。

收发数据前向数据包分配序号，并向对方通报此序号，这都是为防止数据丢失所做的准备。通过向数据包分配序号并确认，可以在数据丢失时马上查看并重传丢失的数据包。因此，TCP可

以保证可靠的数据传输。最后观察主机A向主机B传输的消息：

```
[ACK] SEQ: 1001,  ACK: 2001
```

之前也讨论过，TCP连接过程中发送数据包时需分配序号。在之前的序号1000的基础上加1，也就是分配1001。此时该数据包传递如下消息：

“已正确收到传输的SEQ为2000的数据包，现在可以传输SEQ为2001的数据包。”

这样就传输了添加ACK 2001的ACK消息。至此，主机A和主机B确认了彼此均就绪。

TCP 内部工作原理 2：与对方主机的数据交换

通过第一步三次握手过程完成了数据交换准备，下面就正式开始收发数据，其默认方式如图5-4所示。

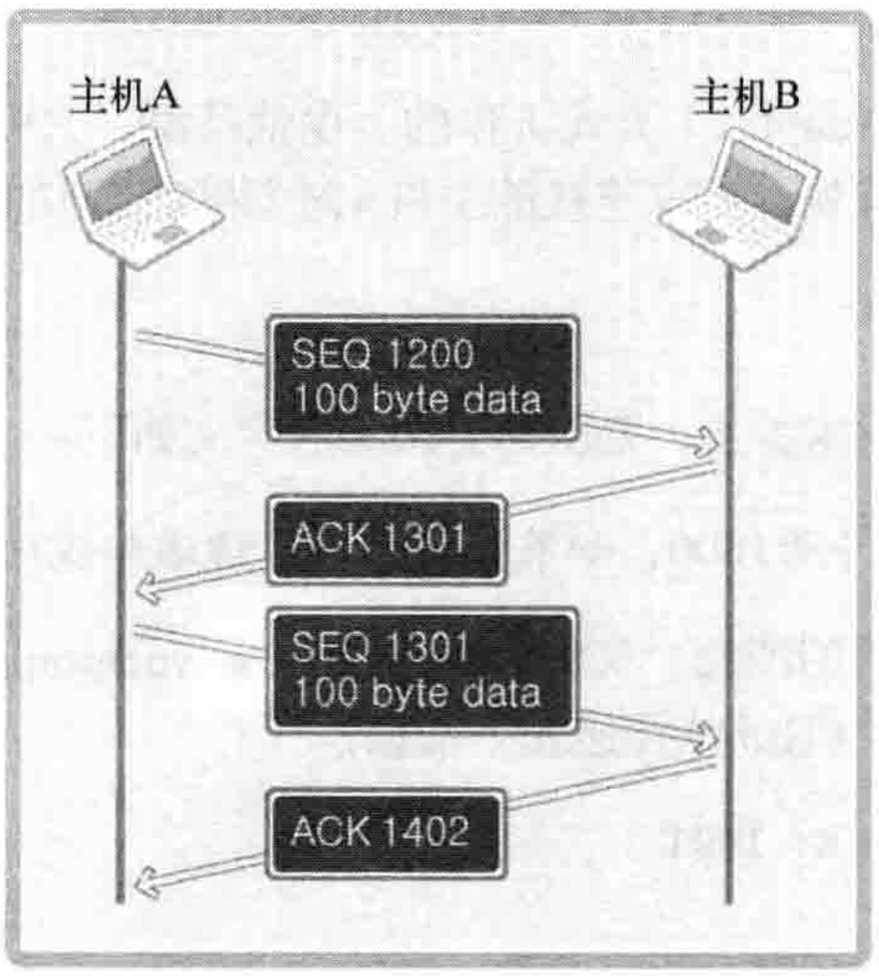


图5-4　TCP套接字的数据交换过程

图5-4给出了主机A分2次（分2个数据包）向主机B传递200字节的过程。首先，主机A通过1个数据包发送100个字节的数据，数据包的SEQ为1200。主机B为了确认这一点，向主机A发送ACK 1301消息。

此时的ACK号为1301而非1201，原因在于ACK号的增量为传输的数据字节数。假设每次ACK号不加传输的字节数，这样虽然可以确认数据包的传输，但无法明确100字节全都正确传递还是丢失了一部分，比如只传递了80字节。因此按如下公式传递ACK消息：

```
ACK 号 → SEQ 号 + 传递的字节数 + 1
```

与三次握手协议相同，最后加1是为了告知对方下次要传递的SEQ号。下面分析传输过程中数据包消失的情况，如图5-5所示。

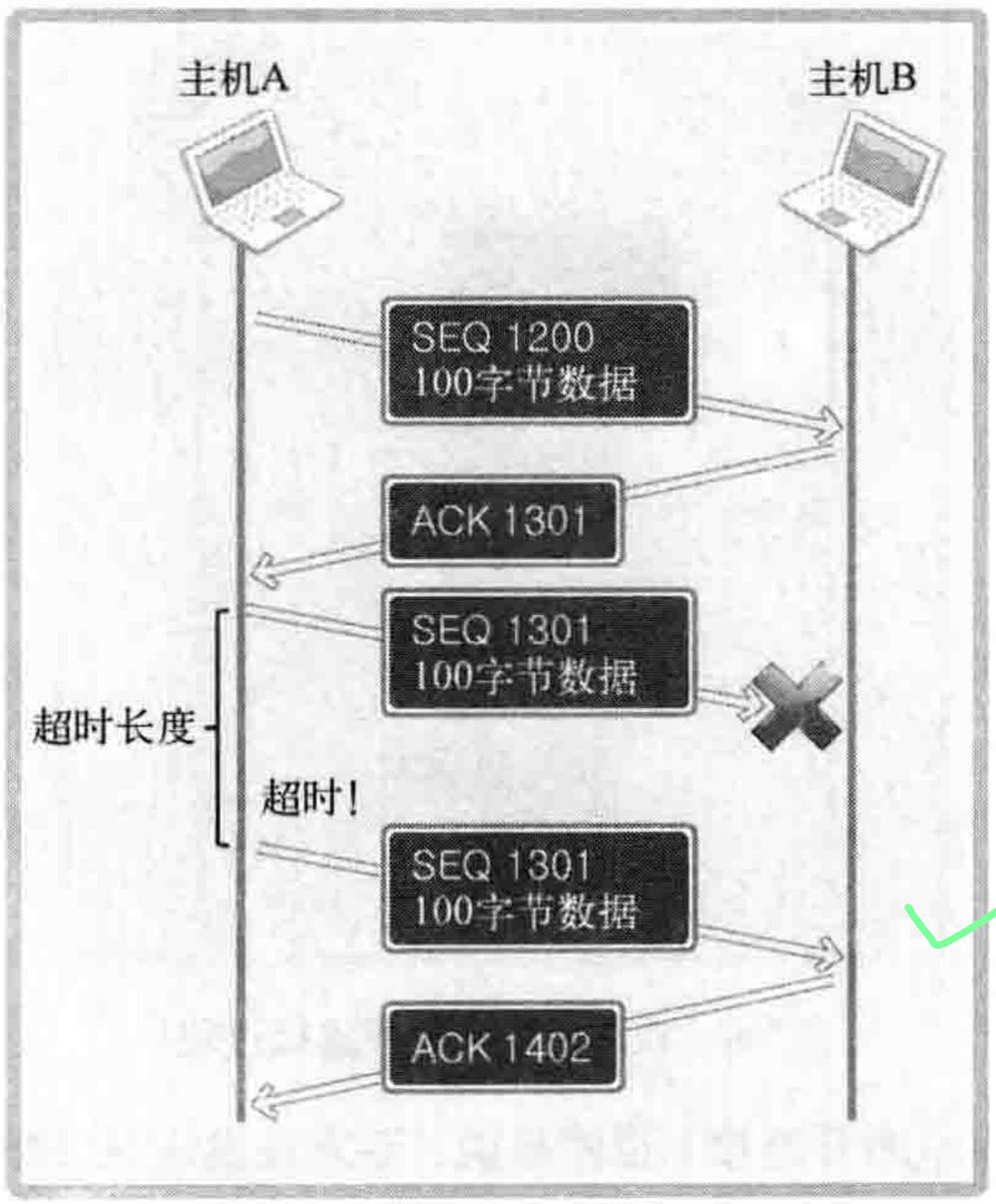


图5-5 TCP套接字数据传输中发生错误

图5-5表示通过SEQ 1301数据包向主机B传递100字节数据。但中间发生了错误，主机B未收到。经过一段时间后，主机A仍未收到对于SEQ 1301的ACK确认，因此试着重传该数据包。为了完成数据包重传，TCP套接字启动计时器以等待ACK应答。若相应计时器发生超时（Time-out!）则重传。

TCP 的内部工作原理 3：断开与套接字的连接

TCP套接字的结束过程也非常优雅。如果对方还有数据需要传输时直接断掉连接会出问题，所以断开连接时需要双方协商。断开连接时双方对话如下。

- 套接字A：“我希望断开连接。”
- 套接字B：“哦，是吗？请稍候。”

- 套接字B：“我也准备就绪，可以断开连接。”
- 套接字A：“好的，谢谢合作。”

先由套接字A向套接字B传递断开连接的消息，套接字B发出确认收到的消息，然后向套接字

A传递可以断开连接的消息，套接字A同样发出确认消息，如图5-6所示。

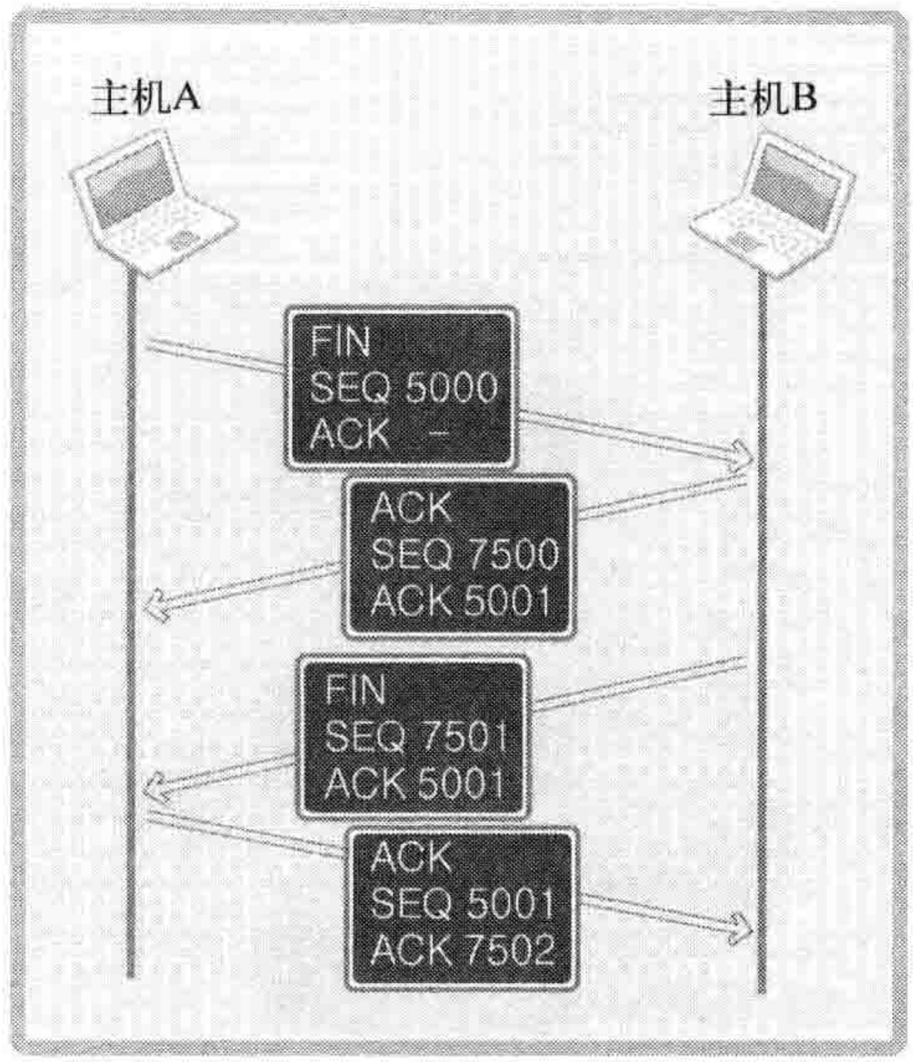


图5-6　TCP套接字断开连接过程

图5-6数据包内的FIN表示断开连接。也就是说，双方各发送1次FIN消息后断开连接。此过程经历4个阶段，因此又称四次握手（Four-way handshaking）。SEQ和ACK的含义与之前讲解的内容一致，故省略。图5-6中向主机A传递了两次ACK 5001，也许这会让各位感到困惑。其实，第二次FIN数据包中的ACK 5001只是因为接收ACK消息后未接收数据而重传的。

前面讲解了TCP协议基本内容TCP流控制（Flow Control），希望这有助于大家理解TCP数据传输特性。

5.3　基于 Windows 的实现

本章讲解的理论在不同操作系统下并无差别，因此在Windows平台没有需要特别说明之处。故只给出之前示例op_server.c和op_client.c的Windows版本代码，转换方式与之前讲过的方式相同。

❖ op_client_win.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <winsock2.h>
5.
6.  #define BUF_SIZE 1024
7.  #define RLT_SIZE 4
8.  #define OPSZ 4
```

```
9.  void ErrorHandling(char *message);
10.
11. int main(int argc, char *argv[])
12. {
13.     WSADATA wsaData;
14.     SOCKET hSocket;
15.     char opmsg[BUF_SIZE];
16.     int result, opndCnt, i;
17.     SOCKADDR_IN servAdr;
18.     if(argc!=3) {
19.         printf("Usage : %s <IP> <port>\n", argv[0]);
20.         exit(1);
21.     }
22.
23.     if(WSAStartup(MAKEWORD(2, 2), &wsaData)!=0)
24.         ErrorHandling("WSAStartup() error!");
25.
26.     hSocket=socket(PF_INET, SOCK_STREAM, 0);
27.     if(hSocket==INVALID_SOCKET)
28.         ErrorHandling("socket() error");
29.
30.     memset(&servAdr, 0, sizeof(servAdr));
31.     servAdr.sin_family=AF_INET;
32.     servAdr.sin_addr.s_addr=inet_addr(argv[1]);
33.     servAdr.sin_port=htons(atoi(argv[2]));
34.
35.     if(connect(hSocket, (SOCKADDR*)&servAdr, sizeof(servAdr))==SOCKET_ERROR)
36.         ErrorHandling("connect() error!");
37.     else
38.         puts("Connected.....");
39.
40.     fputs("Operand count: ", stdout);
41.     scanf("%d", &opndCnt);
42.     opmsg[0]=(char)opndCnt;
43.
44.     for(i=0; i<opndCnt; i++)
45.     {
46.         printf("Operand %d: ", i+1);
47.         scanf("%d", (int*)&opmsg[i*OPSZ+1]);
48.     }
49.     fgetc(stdin);
50.     fputs("Operator: ", stdout);
51.     scanf("%c", &opmsg[opndCnt*OPSZ+1]);
52.     send(hSocket, opmsg, opndCnt*OPSZ+2, 0);
53.     recv(hSocket, &result, RLT_SIZE, 0);
54.
55.     printf("Operation result: %d \n", result);
56.     closesocket(hSocket);
57.     WSACleanup();
58.     return 0;
59. }
60.
61. void ErrorHandling(char *message)
62. {
```

```
63.     fputs(message, stderr);
64.     fputc('\n', stderr);
65.     exit(1);
66. }
```

❖ op_server_win.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <winsock2.h>
5.
6.  #define BUF_SIZE 1024
7.  #define OPSZ 4
8.  void ErrorHandling(char *message);
9.  int calculate(int opnum, int opnds[], char oprator);
10.
11. int main(int argc, char *argv[])
12. {
13.     WSADATA wsaData;
14.     SOCKET hServSock, hClntSock;
15.     char opinfo[BUF_SIZE];
16.     int result, opndCnt, i;
17.     int recvCnt, recvLen;
18.     SOCKADDR_IN servAdr, clntAdr;
19.     int clntAdrSize;
20.     if(argc!=2) {
21.         printf("Usage : %s <port>\n", argv[0]);
22.         exit(1);
23.     }
24.
25.     if(WSAStartup(MAKEWORD(2, 2), &wsaData)!=0)
26.         ErrorHandling("WSAStartup() error!");
27.
28.     hServSock=socket(PF_INET, SOCK_STREAM, 0);
29.     if(hServSock==INVALID_SOCKET)
30.         ErrorHandling("socket() error");
31.
32.     memset(&servAdr, 0, sizeof(servAdr));
33.     servAdr.sin_family=AF_INET;
34.     servAdr.sin_addr.s_addr=htonl(INADDR_ANY);
35.     servAdr.sin_port=htons(atoi(argv[1]));
36.
37.     if(bind(hServSock, (SOCKADDR*)&servAdr, sizeof(servAdr))==SOCKET_ERROR)
38.         ErrorHandling("bind() error");
39.     if(listen(hServSock, 5)==SOCKET_ERROR)
40.         ErrorHandling("listen() error");
41.     clntAdrSize=sizeof(clntAdr);
42.
43.     for(i=0; i<5; i++)
44.     {
45.         opndCnt=0;
46.         hClntSock=accept(hServSock, (SOCKADDR*)&clntAdr, &clntAdrSize);
```

```
47.         recv(hClntSock, &opndCnt, 1, 0);
48.
49.         recvLen=0;
50.         while((opndCnt*OPSZ+1)>recvLen)
51.         {
52.             recvCnt=recv(hClntSock, &opinfo[recvLen], BUF_SIZE-1, 0);
53.             recvLen+=recvCnt;
54.         }
55.         result=calculate(opndCnt, (int*)opinfo, opinfo[recvLen-1]);
56.         send(hClntSock, (char*)&result, sizeof(result), 0);
57.         closesocket(hClntSock);
58.     }
59.     closesocket(hServSock);
60.     WSACleanup();
61.     return 0;
62. }
63.
64. int calculate(int opnum, int opnds[], char op)
65. {
66.     int result=opnds[0], i;
67.
68.     switch(op)
69.     {
70.     case '+':
71.         for(i=1; i<opnum; i++) result+=opnds[i];
72.         break;
73.     case '-':
74.         for(i=1; i<opnum; i++) result-=opnds[i];
75.         break;
76.     case '*':
77.         for(i=1; i<opnum; i++) result*=opnds[i];
78.         break;
79.     }
80.     return result;
81. }
82.
83. void ErrorHandling(char *message)
84. {
85.     fputs(message, stderr);
86.     fputc('\n', stderr);
87.     exit(1);
88. }
```

5.4 习题

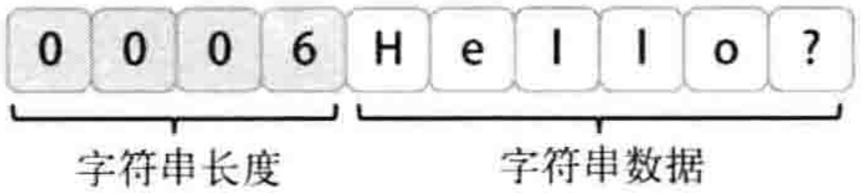
(1) 请说明TCP套接字连接设置的三次握手过程。尤其是3次数据交换过程每次收发的数据内容。

(2) TCP是可靠的数据传输协议，但在通过网络通信的过程中可能丢失数据。请通过ACK和SEQ说名TCP通过何种机制保证丢失数据的可靠传输。

(3) TCP套接字中调用write和read函数时数据如何移动？结合I/O缓冲进行说明。

(4) 对方主机的输入缓冲剩余50字节空间时，若本方主机通过write函数请求传输70字节，请问TCP如何处理这种情况？

(5) 第2章示例tcp_server.c（第1章的hello_servr.c）和tcp_client.c中，客户端接收服务器端传输的字符串后便退出。现更改程序，使服务器端和客户端各传递1次字符串。考虑到使用TCP协议，所以传递字符串前先以4字节整数型方式传递字符串长度。连接时服务器端和客户端数据传输格式如下。



另外，不限制字符串传输顺序及种类，但须进行3次数据交换。

(6) 创建收发文件的服务器端/客户端，实现顺序如下。

- 客户端接受用户输入的传输文件名。
- 客户端请求服务器端传输该文件名所指文件。
- 如果指定文件存在，服务器端就将其发送给客户端；反之，则断开连接。

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。

提　示

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。

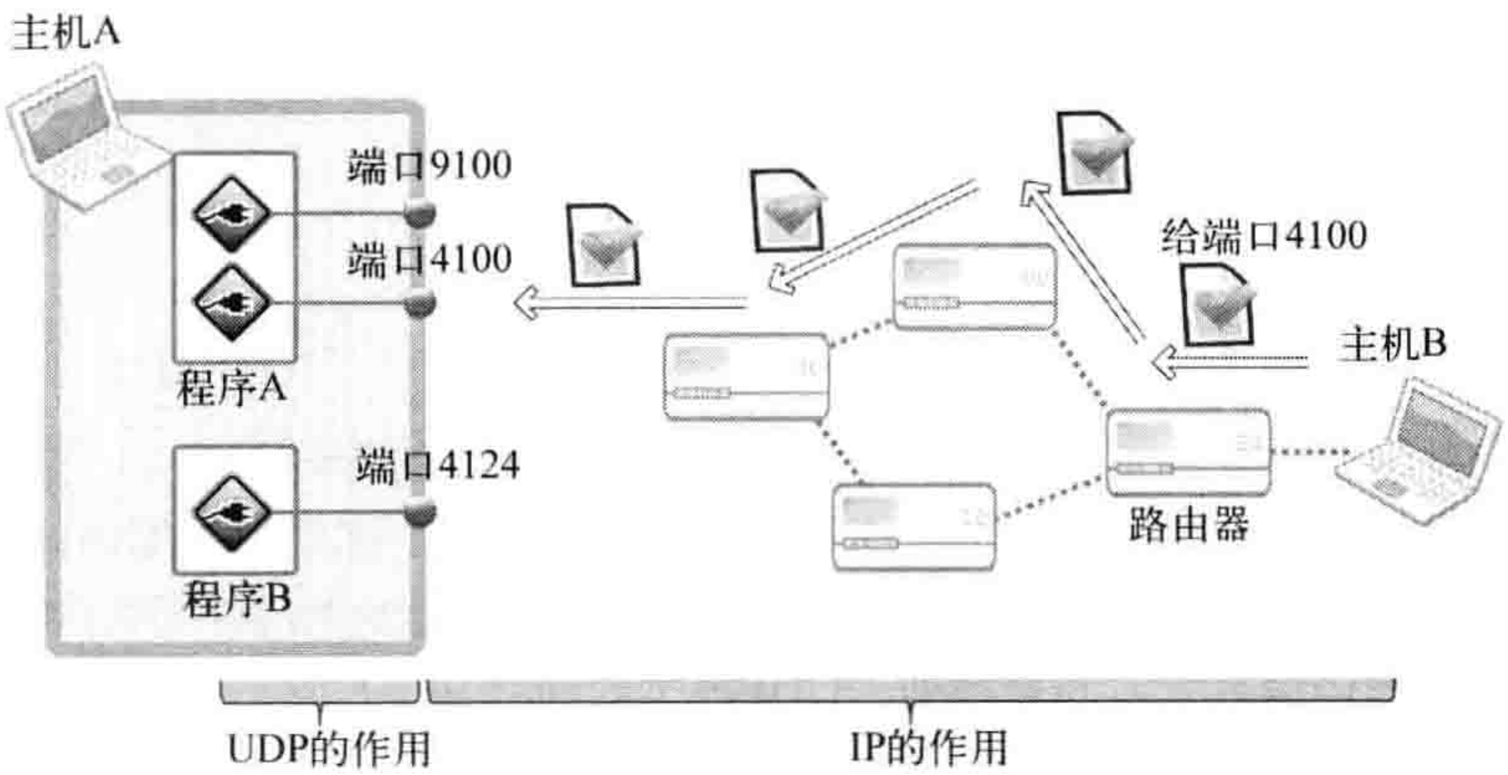


图6-1　数据包传输中UDP和IP的作用

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。

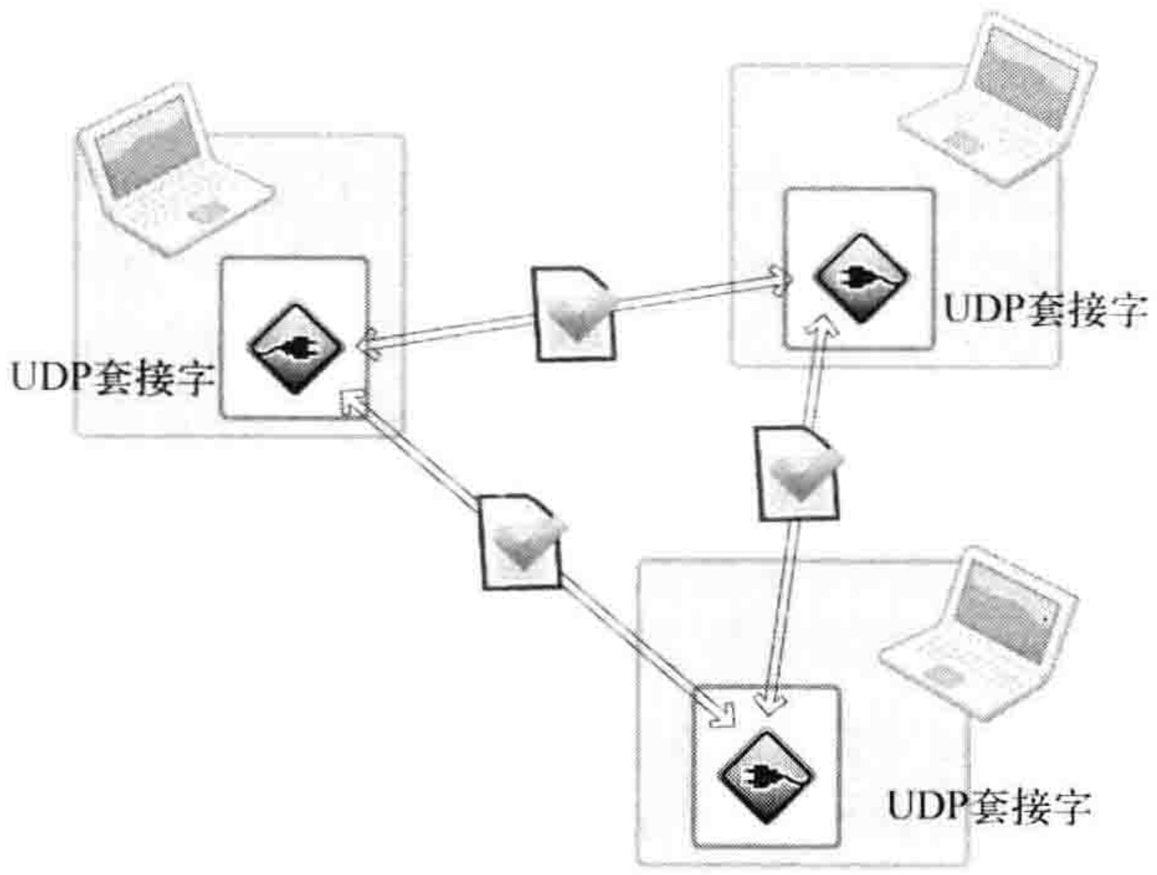


图6-2　UDP套接字通信模型

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. }
```

代码说明

- 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。">
```

```
#define BUF_SIZE 30
void error_handling(char *message);

int main(int argc, char *argv[])
{
    int sock;
    char message[BUF_SIZE];
    int str_len;
    socklen_t adr_sz;

    struct sockaddr_in serv_adr, from_adr;
    if(argc!=3){
        printf("Usage : %s <IP> <port>\n", argv[0]);
        exit(1);
    }

    sock=socket(PF_INET, SOCK_DGRAM, 0);
    if(sock==-1)
        error_handling("socket() error");

    memset(&serv_adr, 0, sizeof(serv_adr));
    serv_adr.sin_family=AF_INET;
    serv_adr.sin_addr.s_addr=inet_addr(argv[1]);
    serv_adr.sin_port=htons(atoi(argv[2]));

    while(1)
    {
        fputs("Insert message(q to quit): ", stdout);
        fgets(message, sizeof(message), stdin);
        if(!strcmp(message,"q\n") || !strcmp(message,"Q\n"))
            break;

        sendto(sock, message, strlen(message), 0,
            (struct sockaddr*)&serv_adr, sizeof(serv_adr));
        adr_sz=sizeof(from_adr);
        str_len=recvfrom(sock, message, BUF_SIZE, 0,
                (struct sockaddr*)&from_adr, &adr_sz);
        message[str_len]=0;
        printf("Message from server: %s", message);
    }
    close(sock);
    return 0;
}

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

- 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。

6

❖ 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

```
#include <"与其他示例头声明一致，故省略。">
#define BUF_SIZE 30
void error_handling(char *message);

int main(int argc, char *argv[])
{
    int sock;
    char message[BUF_SIZE];
    int str_len;
    socklen_t adr_sz;     //多余变量!

    struct sockaddr_in serv_adr, from_adr;        //不再需要from_adr!
    if(argc!=3){
        printf("Usage : %s <IP> <port>\n", argv[0]);
        exit(1);
    }

    sock=socket(PF_INET, SOCK_DGRAM, 0);
    if(sock==-1)
        error_handling("socket() error");

    memset(&serv_adr, 0, sizeof(serv_adr));
    serv_adr.sin_family=AF_INET;
    serv_adr.sin_addr.s_addr=inet_addr(argv[1]);
    serv_adr.sin_port=htons(atoi(argv[2]));

    connect(sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr));

    while(1)
    {
        fputs("Insert message(q to quit): ", stdout);
        fgets(message, sizeof(message), stdin);
        if(!strcmp(message,"q\n") || !strcmp(message,"Q\n"))
            break;
        /*
        sendto(sock, message, strlen(message), 0,
            (struct sockaddr*)&serv_adr, sizeof(serv_adr));
        */
        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

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <winsock2.h>
#define BUF_SIZE 30
void ErrorHandling(char *message);

int main(int argc, char *argv[])
{
    WSADATA wsaData;
    SOCKET servSock;
    char message[BUF_SIZE];
    int strLen;
    int clntAdrSz;

    SOCKADDR_IN servAdr, clntAdr;
    if(argc!=2) {
        printf("Usage : %s <port>\n", argv[0]);
        exit(1);
    }
    if(WSAStartup(MAKEWORD(2, 2), &wsaData)!=0)
        ErrorHandling("WSAStartup() error!");

    servSock=socket(PF_INET, SOCK_DGRAM, 0);
    if(servSock==INVALID_SOCKET)
        ErrorHandling("UDP socket creation error");

    memset(&servAdr, 0, sizeof(servAdr));
    servAdr.sin_family=AF_INET;
    servAdr.sin_addr.s_addr=htonl(INADDR_ANY);
    servAdr.sin_port=htons(atoi(argv[1]));

    if(bind(servSock, (SOCKADDR*)&servAdr, sizeof(servAdr))==SOCKET_ERROR)
        ErrorHandling("bind() error");

    while(1)
    {
        clntAdrSz=sizeof(clntAdr);
        strLen=recvfrom(servSock, message, BUF_SIZE, 0,
                (SOCKADDR*)&clntAdr, &clntAdrSz);
        sendto(servSock, message, strLen, 0,
            (SOCKADDR*)&clntAdr, sizeof(clntAdr));
    }
    closesocket(servSock);
    WSACleanup();
    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);
```