目　录

第四部分　结束网络编程

Part 01

开始网络编程

第1章

理解网络编程和套接字

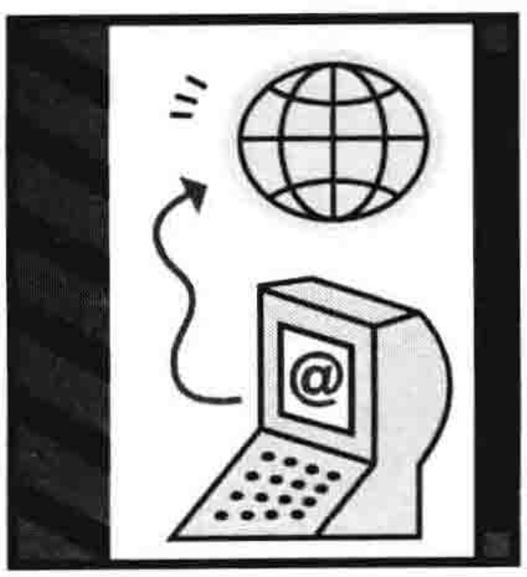
网络编程领域需要一定的操作系统和系统编程知识，同时还需要理解好 TCP/IP 网络数据传输协议。这么说来，网络编程的确需要一定的基础知识，但相比于其他领域，它更有趣，而且没想象中那么难。只要踏踏实实学习，任何人都可以轻松进入网络编程的世界。

深入细节前，本章先帮助各位建立对本书的总体认识，并简要了解后面的内容。希望通过本章的学习，大家能对网络编程有初步了解，摆脱对它的畏惧。

1.1 理解网络编程和套接字

学习C语言时，一般会先学利用printf函数和scanf函数进行控制台输入输出，然后学习文件输入输出。如果各位认真学习过C语言就会发现，控制台输入输出和文件输入输出非常类似。实际上，网络编程也与文件输入输出有很多相似之处，相信大家也能轻松掌握。

网络编程和套接字概要



网络编程就是编写程序使两台连网的计算机相互交换数据。这就是全部内容了吗？是的！网络编程要比想象中简单许多。那么，这两台计算机之间用什么传输数据呢？首先需要物理连接。如今大部分计算机都已连接到庞大的互联网，因此不用担心这点。在此基础上，只需考虑如何编写数据传输软件。但实际上这也不用愁，因为操作系统会提供名为“套接字”（socket）的部件。套接字是网络数据传输用的软件设备。即使对网络数据传输原理不太熟悉，我们也能通过套接字完成数据传输。因此，网络编程又称为套接字编程。那为什么要用“套接字”这个词呢？

我们把插头插到插座上就能从电网获得电力供给，同样，为了与远程计算机进行数据传输，需要连接到因特网，而编程中的“套接字”就是用来连接该网络的工具。它本身就带有“连接”的含义，如果将其引申，则还可以表示两台计算机之间的网络连接。

构建接电话套接字

套接字大致分为两种，其中，先要讨论的TCP套接字可以比喻成电话机。实际上，电话机也是通过固定电话网（telephone network）完成语音数据交换的。因此，我们熟悉的固定电话与套接字实际并无太大区别。下面利用电话机讲解套接字的创建及使用方法。

电话机可以同时用来拨打或接听，但对套接字而言，拨打和接听是有区别的。我们先讨论用于接听的套接字创建过程。

调用socket函数（安装电话机）时进行的对话

- 问：“接电话需要准备什么？”
- 答：“当然是电话机!”

有了电话机才能安装电话，接下来，我们就准备一部漂亮的电话机。下列函数创建的就是相当于电话机的套接字。

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);

    ➔ 成功时返回文件描述符，失败时返回-1。
```

上述函数及本章涉及的其他函数的详细说明将在以后章节中逐一给出，现在只需掌握“原来是由socket函数生成套接字的”就足够了。另外，我们只需购买机器，剩下的安装和分配电话号码等工作都由电信局的工作人员完成。而套接字需要我们自己安装，这也是套接字编程难点所在，但多安装几次就会发现其实不难。准备好电话机后要考虑分配电话号码的问题，这样别人才能联系到自己。

调用bind函数（分配电话号码）时进行的对话

- 问：“请问您的电话号码是多少？”
- 答：“我的电话号码是123-1234。”

套接字同样如此。就像给电话机分配电话号码一样（虽然不是真的把电话号码给了电话机），利用以下函数给创建好的套接字分配地址信息（IP地址和端口号）。

```
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *myaddr, socklen_t addrlen);
```

➔ 成功时返回0，失败时返回-1。

调用bind函数给套接字分配地址后，就基本完成了接电话的所有准备工作。接下来需要连接电话线并等待来电。

调用listen函数（连接电话线）时进行的对话

❑ 问："已架设完电话机后是否只需连接电话线？"

❑ 答："对，只需连接就能接听电话。"

一连接电话线，电话机就转为可接听状态，这时其他人可以拨打电话请求连接到该机。同样，需要把套接字转化成可接收连接的状态。

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

➔ 成功时返回0，失败时返回-1。

连接好电话线后，如果有人拨打电话就会响铃，拿起话筒才能接听电话。

调用accept函数（拿起话筒）时进行的对话

❑ 问："电话铃响了，我该怎么办？"

❑ 答："难道您真不知道？接听啊！"

拿起话筒意味着接收了对方的连接请求。套接字同样如此，如果有人为了完成数据传输而请求连接，就需要调用以下函数进行受理。

```
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

➔ 成功时返回文件描述符，失败时返回-1。

网络编程中接受连接请求的套接字创建过程可整理如下。

❑ 第一步：调用socket函数创建套接字。

❑ 第二步：调用bind函数分配IP地址和端口号。

❑ 第三步：调用listen函数转为可接收请求状态。

❑ 第四步：调用accept函数受理连接请求。

记住并掌握这些步骤就相当于为套接字编程勾勒好了轮廓，后续章节会为此轮廓着色。

编写“Hello world!”服务器端

服务器端（server）是能够受理连接请求的程序。下面构建服务器端以验证之前提到的函数调用过程，该服务器端收到连接请求后向请求者返回“Hello world!”答复。除各种函数的调用顺序外，我们还未涉及任何实际编程。因此，阅读代码时请重点关注套接字相关函数的调用过程，不必理解全部示例。

❖ hello_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.  void error_handling(char *message);
8.
9.  int main(int argc, char *argv[])
10. {
11.     int serv_sock;
12.     int clnt_sock;
13.
14.     struct sockaddr_in serv_addr;
15.     struct sockaddr_in clnt_addr;
16.     socklen_t clnt_addr_size;
17.
18.     char message[]="Hello World!";
19.
20.     if(argc!=2)
21.     {
22.         printf("Usage : %s <port>\n", argv[0]);
23.         exit(1);
24.     }
25.
26.     serv_sock=socket(PF_INET, SOCK_STREAM, 0);
27.     if(serv_sock == -1)
28.         error_handling("socket() error");
29.
30.     memset(&serv_addr, 0, sizeof(serv_addr));
31.     serv_addr.sin_family=AF_INET;
32.     serv_addr.sin_addr.s_addr=htonl(INADDR_ANY);
33.     serv_addr.sin_port=htons(atoi(argv[1]));
34.
35.     if(bind(serv_sock, (struct sockaddr*) &serv_addr, sizeof(serv_addr))==-1)
36.         error_handling("bind() error");
37.
38.     if(listen(serv_sock, 5)==-1)
39.         error_handling("listen() error");
```

```
40.
41.     clnt_addr_size=sizeof(clnt_addr);
42.     clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_addr, &clnt_addr_size);
43.     if(clnt_sock==-1)
44.         error_handling("accept() error");
45.
46.     write(clnt_sock, message, sizeof(message));
47.     close(clnt_sock);
48.     close(serv_sock);
49.     return 0;
50. }
51.
52. void error_handling(char *message)
53. {
54.     fputs(message, stderr);
55.     fputc('\n', stderr);
56.     exit(1);
57. }
```

代码说明

- 第26行：调用socket函数创建套接字。
- 第35行：调用bind函数分配IP地址和端口号。
- 第38行：调用listen函数将套接字转为可接收连接状态。
- 第42行：调用accept函数受理连接请求。如果在没有连接请求的情况下调用该函数，则不会返回，直到有连接请求为止。
- 第46行：稍后将要介绍的write函数用于传输数据，若程序经过第42行代码执行到本行，则说明已经有了连接请求。

编译并运行以上示例，创建等待连接请求的服务器端。目前不必详细分析源代码，只需确认之前4个函数调用过程。稍后将讲解上述示例中调用的write函数。下面讨论如何编写向服务器端发送连接请求的客户端。

构建打电话套接字

服务器端创建的套接字又称为服务器端套接字或监听（listening）套接字。接下来介绍的套接字是用于请求连接的客户端套接字。客户端套接字的创建过程比创建服务器端套接字简单，因此直接进行讲解。

还未介绍打电话（请求连接）的函数，因为其调用的是客户端套接字，如下所示。

```
#include <sys/socket.h>
int connect(int sockfd, struct sockaddr *serv_addr, socklen_t addrlen);
```

➔ 成功时返回0，失败时返回-1。

客户端程序只有“调用socket函数创建套接字”和“调用connect函数向服务器端发送连接请求”这两个步骤，因此比服务器端简单。下面给出客户端，查看以下两项内容：第一，调用socket函数和connect函数；第二，与服务器端共同运行以收发字符串数据。

❖ hello_client.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.  void error_handling(char *message);
8.
9.  int main(int argc, char* argv[])
10. {
11.     int sock;
12.     struct sockaddr_in serv_addr;
13.     char message[30];
14.     int str_len;
15.
16.     if(argc!=3)
17.     {
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_addr, 0, sizeof(serv_addr));
27.     serv_addr.sin_family=AF_INET;
28.     serv_addr.sin_addr.s_addr=inet_addr(argv[1]);
29.     serv_addr.sin_port=htons(atoi(argv[2]));
30.
31.     if(connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr))==-1)
32.         error_handling("connect() error!");
33.
34.     str_len=read(sock, message, sizeof(message)-1);
35.     if(str_len==-1)
36.         error_handling("read() error!");
37.
38.     printf("Message from server : %s \n", message);
39.     close(sock);
40.     return 0;
41. }
42.
43. void error_handling(char *message)
44. {
45.     fputs(message, stderr);
46.     fputc('\n', stderr);
47.     exit(1);
48. }
```

- 第22行：创建套接字，但此时套接字并不马上分为服务器端和客户端。如果紧接着调用bind、listen函数，将成为服务器端套接字；如果调用connect函数，将成为客户端套接字。
- 第31行：调用connect函数向服务器端发送连接请求。

这样就编好了服务器端和客户端，相信各位会产生好多疑问（实际上不懂的内容比知道的更多）。接下来的几章将进行解答，请不要着急。

在 Linux 平台下运行

虽未另行说明，但上述两个示例应在Linux环境中编译并执行。接下来将简单介绍Linux下的C语言编译器——GCC（GNU Compiler Collection，GNU编译器集合）。下面是对hello_server.c示例进行编译的命令。

```
gcc hello_server.c -o hserver
```

➔ 编译 hello_server.c 文件并生成可执行文件 hserver。

该命令中的-o是用来指定可执行文件名的可选参数，因此，编译后将生成可执行文件hserver。可如下执行此项命令。

```
./hserver
```

➔ 运行当前目录下的 hserver 文件。

了解编译和运行相关知识的确多多益善，但学习本书只需掌握基本用法。接下来运行程序。服务器端需要在运行时接收客户端的连接请求，因此先运行服务器端。

❖ 运行结果：hello_server.c

```
root@my_linux:/tcpip# gcc hello_server.c -o hserver
root@my_linux:/tcpip# ./hserver 9190
```

正常情况下程序将停留在此状态，因为服务器端调用的accept函数还未返回。接下来运行客户端。

❖ 运行结果：hello_client.c

```
root@my_linux:/tcpip# gcc hello_client.c -o hclient
root@my_linux:/tcpip# ./hclient 127.0.0.1 9190
Message from server: Hello World!
root@my_linux:/tcpip#
```

由此查看客户端消息传输过程。同时发现，完成消息传输后，服务器端和客户端都停止运行。执行过程中输入的127.0.0.1是运行示例用的计算机（本地计算机）的IP地址。如果在同一台计算

机中同时运行服务器端和客户端，将采用这种连接方式。但如果服务器端与客户端在不同计算机中运行，则应采用服务器端所在计算机的IP地址。

提 示

再次运行程序前需等待

上面的服务器端无法立即重新运行。如果想再次运行，则需要更改之前输入的端口号 9190。后面会详细讲解其原因，现在不必对此感到意外。

1.2 基于 Linux 的文件操作

讨论套接字的过程中突然谈及文件也许有些奇怪。但对Linux而言，socket操作与文件操作没有区别，因而有必要详细了解文件。在Linux世界里，socket也被认为是文件的一种，因此在网络数据传输过程中自然可以使用文件I/O的相关函数。Windows则与Linux不同，是要区分socket和文件的。因此在Windows中需要调用特殊的数据传输相关函数。

底层文件访问（Low-Level File Access）和文件描述符（File Descriptor）

即使看到“底层”二字，也会有读者臆测其难以理解。实际上，“底层”这个表达可以理解为“与标准无关的操作系统独立提供的”。稍后讲解的函数是由Linux提供的，而非ANSI标准定义的函数。如果想使用Linux提供的文件I/O函数，首先应该理解好文件描述符的概念。

此处的文件描述符是系统分配给文件或套接字的整数。实际上，学习C语言过程中用过的标准输入输出及标准错误在Linux中也被分配表1-1中的文件描述符。

表1-1 分配给标准输入输出及标准错误的文件描述符

文件描述符	对 象
0	标准输入：Standard Input
1	标准输出：Standard Output
2	标准错误：Standard Error

文件和套接字一般经过创建过程才会被分配文件描述符。而表1-1中的3种输入输出对象即使未经过特殊的创建过程，程序开始运行后也会被自动分配文件描述符。稍后将详细讲解其使用方法及含义。

知识补给站 文件描述符（文件句柄）

学校附近有个服务站，只需打个电话就能复印所需论文。服务站有位常客叫英秀，他每次都要求复印同一篇论文的一部分内容。

“大叔您好！请帮我复印一下《关于随着高度信息化社会而逐步提升地位的触觉、知觉、思维、性格、智力等人类生活质量相关问题特性的人类学研究》这篇论文第26页到第30页。”

这位同学每天这样打好几次电话，更雪上加霜的是语速还特别慢。终于有一天大叔说：

“从现在开始，那篇论文就编为第18号！你就说帮我复印18号论文26页到30页！”

之后英秀也是只复印超过50字标题的论文，大叔也会给每篇论文分配无重复的新号（数字）。这才不会头疼于与英秀的对话，且不影响业务。

该示例中，大叔相当于操作系统，英秀相当于程序员，论文号相当于文件描述符，论文相当于文件或套接字。也就是说，每当生成文件或套接字，操作系统将返回分配给它们的整数。这个整数将成为程序员与操作系统之间良好沟通的渠道。实际上，文件描述符只不过是为了方便称呼操作系统创建的文件或套接字而赋予的数而已。

文件描述符有时也称为文件句柄，但“句柄”主要是Windows中的术语。因此，本书中如果涉及Windows平台将使用“句柄”，如果是Linux平台则用“描述符”。

打开文件

首先介绍打开文件以读写数据的函数。调用此函数时需传递两个参数：第一个参数是打开的目标文件名及路径信息，第二个参数是文件打开模式（文件特性信息）。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *path, int flag);

    ➔ 成功时返回文件描述符，失败时返回-1。
```

- path　文件名的字符串地址。
- flag　文件打开模式信息。

表1-2是此函数第二个参数flag可能的常量值及含义。如需传递多个参数，则应通过位或运算（OR）符组合并传递。

表1-2 文件打开模式

打开模式	含 义
O_CREAT	必要时创建文件
O_TRUNC	删除全部现有数据
O_APPEND	维持现有数据，保存到其后面
O_RDONLY	只读打开
O_WRONLY	只写打开
O_RDWR	读写打开

稍后将给出此函数的使用示例。接下来先介绍关闭文件和写文件时调用的函数。

关闭文件

各位学习C语言时学过，使用文件后必须关闭。下面介绍关闭文件时调用的函数。

```
#include <unistd.h>

int close(int fd);

    ➔ 成功时返回 0，失败时返回-1。
```

- fd　　需要关闭的文件或套接字的文件描述符。

若调用此函数的同时传递文件描述符参数，则关闭（终止）相应文件。另外需要注意的是，此函数不仅可以关闭文件，还可以关闭套接字。这再次证明了“Linux操作系统 不区分文件与套接字”的特点。

将数据写入文件

接下来介绍的write函数用于向文件输出（传输）数据。当然，Linux中不区分文件与套接字，因此，通过套接字向其他计算机传递数据时也会用到该函数。之前的示例也调用它传递字符串“Hello World!”。

```
#include <unistd.h>

ssize_t write(int fd, const void * buf, size_t nbytes);

    ➔ 成功时返回写入的字节数，失败时返回-1。
```

- fd　　显示数据传输对象的文件描述符。
- buf　　保存要传输数据的缓冲地址值。
- nbytes　　要传输数据的字节数。

此函数定义中，size_t是通过typedef声明的unsigned int类型。对ssize_t来说，size_t前面多加的s代表signed，即ssize_t是通过typedef声明的signed int类型。

知识补给站　以_t为后缀的数据类型

我们已经接触到ssize_t、size_t等陌生的数据类型。这些都是元数据类型（primitive），在sys/types.h头文件中一般由typedef声明定义，算是给大家熟悉的基本数据类型起了别名。既然已经有了基本数据类型，为何还要声明并使用这些新的呢？

人们目前普遍认为int是32位的，因为主流操作系统和计算机仍采用32位。而在过去16位操作系统时代，int类型是16位的。根据系统的不同、时代的变化，数据类型的表现形式也随之改变，需要修改程序中使用的数据类型。如果之前已在需要声明4字节数据类型之处使用了size_t或ssize_t，则将大大减少代码变动，因为只需要修改并编译size_t和ssize_t的typedef声明即可。在项目中，为了给基本数据类型赋予别名，一般会添加大量typedef声明。而为了与程序员定义的新数据类型加以区分，操作系统定义的数据类型会添加后缀_t。

下面通过示例帮助大家更好地理解前面讨论过的函数。此程序将创建新文件并保存数据。

❖ low_open.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <fcntl.h>
4.  #include <unistd.h>
5.  void error_handling(char* message);
6.
7.  int main(void)
8.  {
9.      int fd;
10.     char buf[]="Let's go!\n";
11.
12.     fd=open("data.txt", O_CREAT|O_WRONLY|O_TRUNC);
13.     if(fd==-1)
14.         error_handling("open() error!");
15.     printf("file descriptor: %d \n", fd);
16.
17.     if(write(fd, buf, sizeof(buf))==-1)
18.         error_handling("write() error!");
19.     close(fd);
20.     return 0;
21. }
22.
23. void error_handling(char* message)
```

```
24. {
25.     //与之前示例相同，故省略!
26. }
```

代码说明

- 第12行：文件打开模式为O_CREAT、O_WRONLY和O_TRUNC的组合，因此将创建空文件，并只能写。若存在data.txt文件，则清空文件的全部数据。
- 第17行：向对应于fd中保存的文件描述符的文件传输buf中保存的数据。

❖ 运行结果：low_open.c

```
root@my_linux:/tcpip# gcc low_open.c -o lopen
root@my_linux:/tcpip# ./lopen
file descriptor: 3
root@my_linux:/tcpip# cat data.txt
Let's go!
root@my_linux:/tcpip#
```

运行示例后，利用Linux的cat命令输出data.txt文件内容，可以确认确实已向文件传输数据。

读取文件中的数据

与之前的write函数相对应，read函数用来输入（接收）数据。

```
#include <unistd.h>

ssize_t read(int fd, void * buf, size_t nbytes);

    ➔ 成功时返回接收的字节数（但遇到文件结尾则返回 0），失败时返回-1。
```

- fd　　显示数据接收对象的文件描述符。
- buf　　要保存接收数据的缓冲地址值。
- nbytes　　要接收数据的最大字节数。

下列示例将通过read函数读取data.txt中保存的数据。

❖ low_read.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <fcntl.h>
4.  #include <unistd.h>
5.  #define BUF_SIZE 100
6.  void error_handling(char* message);
7.
8.  int main(void)
```

```
9.  {
10.     int fd;
11.     char buf[BUF_SIZE];
12.
13.     fd=open("data.txt", O_RDONLY);
14.     if( fd==-1)
15.         error_handling("open() error!");
16.     printf("file descriptor: %d \n" , fd);
17.
18.     if(read(fd, buf, sizeof(buf))==-1)
19.         error_handling("read() error!");
20.     printf("file data: %s", buf);
21.     close(fd);
22.     return 0;
23. }
24.
25. void error_handling(char* message)
26. {
27.     //与之前示例相同，故省略!
28. }
```

代码说明

- 第13行：打开读取专用文件data.txt。
- 第18行：调用read函数向第11行中声明的数组buf保存读入的数据。

❖ 运行结果 low_read.c

```
root@my_linux:/tcpip# gcc low_read.c -o lread
root@my_linux:/tcpip# ./lread
file descriptor: 3
file data: Let's go!
root@my_linux:/tcpip#
```

基于文件描述符的I/O操作相关介绍到此结束。希望各位记住，该内容同样适用于套接字。

文件描述符与套接字

下面将同时创建文件和套接字，并用整数型态比较返回的文件描述符值。

❖ fd_seri.c

```
1.  #include <stdio.h>
2.  #include <fcntl.h>
3.  #include <unistd.h>
4.  #include <sys/socket.h>
5.
6.  int main(void)
7.  {
```

```
8.      int fd1, fd2, fd3;
9.      fd1=socket(PF_INET, SOCK_STREAM, 0);
10.     fd2=open("test.dat", O_CREAT|O_WRONLY|O_TRUNC);
11.     fd3=socket(PF_INET, SOCK_DGRAM, 0);
12.
13.     printf("file descriptor 1: %d\n", fd1);
14.     printf("file descriptor 2: %d\n", fd2);
15.     printf("file descriptor 3: %d\n", fd3);
16.
17.     close(fd1); close(fd2); close(fd3);
18.     return 0;
19. }
```

- 第9～11行：创建1个文件和2个套接字。
- 第13～15行：输出之前创建的文件描述符的整数值。

❖ 运行结果：fd_seri.c

```
root@my_linux:/tcpip# gcc fd_seri.c -o fds
root@my_linux:/tcpip# ./fds
file descriptor 1: 3
file descriptor 2: 4
file descriptor 3: 5
root@my_linux:/tcpip#
```

从输出的文件描述符整数值可以看出，描述符从3开始以由小到大的顺序编号（numbering），因为0、1、2是分配给标准I/O的描述符（如表1-1所示）。

1.3 基于 Windows 平台的实现

Windows套接字（以下简称Winsock）大部分是参考BSD系列UNIX套接字设计的，所以很多地方都跟Linux套接字类似。因此，只需要更改Linux环境下编好的一部分网络程序内容，就能在Windows平台下运行。本书也会同时讲解Linux和Windows两大平台，这不会给大家增加负担，反而会减轻压力。

同时学习 Linux 和 Windows 的原因

大多数项目都在Linux系列的操作系统下开发服务器端，而多数客户端是在Windows平台下开发的。不仅如此，有时应用程序还需要在两个平台之间相互切换。因此，学习套接字编程的过程中，有必要兼顾Windows和Linux两大平台。另外，这两大平台下的套接字编程非常类似，如果把其中相似的部分放在一起讲解，将大大提高学习效率。这会不会增加学习负担？一点也不。只要理解好其中一个平台下的网络编程方法，就很容易通过分析差异掌握另一平台。

为 Windows 套接字编程设置头文件和库

为了在Winsock基础上开发网络程序，需要做如下准备。

- 导入头文件winsock2.h。
- 链接ws2_32.lib库。

首先介绍项目中链接ws2_32.lib库的方法。我用的环境是Visual Studio 2008版本，接下来的讲解同样适用于更高版本的开发环境，不必因版本不同而感到困惑。打开项目的“属性”页，选择“配置属性”→“输入”→“附加依赖项”，如图1-1所示。当然，也可以通过快捷键Alt+F7打开“属性页”。

图1-1　项目“属性”页

接下来需要在图1-1的“附加依赖项”右边空白处直接写入ws2_32.lib。也可以通过点击空白处右边的按钮弹出如图1-2所示的对话框，并填写库名。

图1-2　链接库

设置库的工作到此结束。现在只需要在源文件中添加头文件，即可调用Winsock相关函数。

提 示

附加依赖项窗口位置可能不同

根据 VC++版本号的不同，图 1-2 中的附加依赖项窗口位置可能不同。一般可通过以下两种路径找到附加依赖项。

- 快捷键 Alt+F7→“配置属性”→“输入”→“附加依赖项”
- 快捷键 Alt+F7→“配置属性”→“链接器”→“输入”→“附加依赖项”

Winsock 的初始化

进行Winsock编程时，首先必须调用WSAStartup函数，设置程序中用到的Winsock版本，并初始化相应版本的库。

```
#include <winsock2.h>

int WSAStartup(WORD wVersionRequested, LPWSADATA lpWSAData);
```

➔ 成功时返回 0，失败时返回非零的错误代码值。

- wVersionRequested　程序员要用的Winsock版本信息。
- lpWSAData　WSADATA结构体变量的地址值。

有必要给出上述两个参数的详细说明。先说第一个，Winsock中存在多个版本，应准备WORD类型的（WORD是通过typedef声明定义的unsigned short类型）套接字版本信息，并传递给该函数的第一个参数wVersionRequested。若版本为1.2，则其中1是主版本号，2是副版本号，应传递0x0201。

如前所述，高8位为副版本号，低8位为主版本号，以此进行传递。本书主要使用2.2版本，故应传递0x0202。不过，以字节为单位手动构造版本信息有些麻烦，借助MAKEWORD宏函数则能轻松构建WORD型版本信息。

- MAKEWORD(1, 2);：//主版本为1，副版本为2，返回0x0201。
- MAKEWORD(2, 2);：//主版本为2，副版本为2，返回0x0202。

接下来讲解第二个参数lpWSADATA，此参数中需传入WSADATA型结构体变量地址（LPWSADATA是WSADATA的指针类型）。调用完函数后，相应参数中将填充已初始化的库信息。虽无特殊含义，但为了调用函数，必须传递WSADATA结构体变量地址。下面给出WSAStartup函数调用过程，这段代码几乎已成为Winsock编程的公式。

```
int main(int argc, char* argv[])
{
    WSADATA wsaData;
    . . . .
    if(WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
        ErrorHandling("WSAStartup() error!");
    . . . .
    return 0;
}
```

前面已经介绍了Winsock相关库的初始化方法，接下来讲解如何注销该库——利用下面给出的函数。

```
#include <winsock2.h>

int WSACleanup(void);
```

➔ 成功时返回 0，失败时返回 SOCKET_ERROR。

调用该函数时，Winsock相关库将归还Windows操作系统，无法再调用Winsock相关函数。从原则上讲，无需再使用Winsock函数时才调用该函数，但通常都在程序结束之前调用。

1.4　基于 Windows 的套接字相关函数及示例

本节介绍的Winsock函数与之前的Linux套接字相关函数相对应。既然只是介绍，就不做详细说明了，目的只在于让各位体会基于Linux和Windows的套接字函数之间的相似性。

基于 Windows 的套接字相关函数

首先介绍的函数与Linux下的socket函数提供相同功能。稍后讲解返回值类型SOCKET。

```
#include <winsock2.h>

SOCKET socket(int af, int type, int protocol);
```

➔ 成功时返回套接字句柄，失败时返回 INVALID_SOCKET。

下列函数与Linux的bind函数相同，调用其分配IP地址和端口号。

```
#include <winsock2.h>

int bind(SOCKET s, const struct sockaddr * name, int namelen);
```

➔ 成功时返回 0，失败时返回 SOCKET_ERROR。

下列函数与Linux的listen函数相同，调用其使套接字可接收客户端连接。

```
#include <winsock2.h>

int listen(SOCKET s,int backlog);
```

➔ 成功时返回 0，失败时返回 SOCKET_ERROR。

下列函数与Linux的accept函数相同，调用其受理客户端连接请求。

```
#include <winsock2.h>

SOCKET accept(SOCKET s, struct sockaddr * addr, int * addrlen);
```

➔ 成功时返回套接字句柄，失败时返回 INVALID_SOCKET。

下列函数与Linux的connect函数相同，调用其从客户端发送连接请求。

```
#include <winsock2.h>

int connect(SOCKET s, const struct sockaddr * name, int namelen);
```

➔ 成功时返回 0，失败时返回 SOCKET_ERROR。

最后这个函数在关闭套接字时调用。Linux中，关闭文件和套接字时都会调用close函数；而Windows中有专门用来关闭套接字的函数。

```
#include <winsock2.h>

int closesocket(SOCKET s);
```

➔ 成功时返回 0，失败时返回 SOCKET_ERROR。

以上就是基于Windows的套接字相关函数，虽然返回值和参数与Linux函数有所区别，但具有相同功能的函数名是一样的。正是这些特点使跨越两大操作系统平台的网络编程更加简单。

Windows 中的文件句柄和套接字句柄

Linux内部也将套接字当作文件，因此，不管创建文件还是套接字都返回文件描述符。之前也通过示例介绍了文件描述符返回及编号的过程。Windows中通过调用系统函数创建文件时，返回“句柄”（handle），换言之，Windows中的句柄相当于Linux中的文件描述符。只不过Windows中要区分文件句柄和套接字句柄。虽然都称为“句柄”，但不像Linux那样完全一致。文件句柄相关函数与套接字句柄相关函数是有区别的，这一点不同于Linux文件描述符。

既然对句柄有了一定理解，接下来再观察基于Windows的套接字相关函数，这将加深各位对SOCKET类型的参数和返回值的理解。的确！这就是为了保存套接字句柄整型值的新数据类型，它由typedef声明定义。回顾socket、listen和accept等套接字相关函数，则更能体会到与Linux中套接字相关函数的相似性。

有些程序员可能会问：“既然Winsock是以UNIX、Linux系列的BSD套接字为原型设计的，为什么不照搬过来，而是存在一定差异呢？”有人认为这是微软为了防止UNIX、Linux服务器端直接移植到Windows而故意为之。从网络程序移植性角度上看，这也是可以理解的。但我有不同意见。从本质上说，两种操作系统内核结构上存在巨大差异，而依赖于操作系统的代码实现风格也不尽相同，连Windows程序员给变量命名的方式也不同于Linux程序员。从各方面考虑，保持这种差异性就显得比较自然。因此我个人认为，Windows套接字与BSD系列的套接字编程方式有所不同是为了保持这种自然差异性。

创建基于 Windows 的服务器端和客户端

接下来将之前基于Linux的服务器端与客户端示例转化到Windows平台。目前想完全理解这些代码有些困难，我们只需验证套接字相关函数的调用过程、套接字库的初始化与注销过程即可。先介绍服务器端示例。

❖ hello_server_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.      SOCKET hServSock, hClntSock;
10.     SOCKADDR_IN servAddr, clntAddr;
```

```
11.
12.     int szClntAddr;
13.     char message[]="Hello World!";
14.     if(argc!=2)
15.     {
16.         printf("Usage : %s <port>\n", argv[0]);
17.         exit(1);
18.     }
19.
20.     if(WSAStartup(MAKEWORD(2, 2), &wsaData)!=0)
21.         ErrorHandling("WSAStartup() error!");
22.
23.     hServSock=socket(PF_INET, SOCK_STREAM, 0);
24.     if(hServSock==INVALID_SOCKET)
25.         ErrorHandling("socket() error");
26.
27.     memset(&servAddr, 0, sizeof(servAddr));
28.     servAddr.sin_family=AF_INET;
29.     servAddr.sin_addr.s_addr=htonl(INADDR_ANY);
30.     servAddr.sin_port=htons(atoi(argv[1]));
31.
32.     if(bind(hServSock, (SOCKADDR*) &servAddr, sizeof(servAddr))==SOCKET_ERROR)
33.         ErrorHandling("bind() error");
34.
35.     if(listen(hServSock, 5)==SOCKET_ERROR)
36.         ErrorHandling("listen() error");
37.
38.     szClntAddr=sizeof(clntAddr);
39.     hClntSock=accept(hServSock, (SOCKADDR*)&clntAddr,&szClntAddr);
40.     if(hClntSock==INVALID_SOCKET)
41.         ErrorHandling("accept() error");
42.
43.     send(hClntSock, message, sizeof(message), 0);
44.     closesocket(hClntSock);
45.     closesocket(hServSock);
46.     WSACleanup();
47.     return 0;
48. }
49.
50. void ErrorHandling(char* message)
51. {
52.     fputs(message, stderr);
53.     fputc('\n', stderr);
54.     exit(1);
55. }
```

代码说明

- 第20行：初始化套接字库。
- 第23、32行：第23行创建套接字，第32行给该套接字分配IP地址与端口号。
- 第35行：调用listen函数使第23行创建的套接字成为服务器端套接字。
- 第39行：调用accept函数受理客户端连接请求。
- 第43行：调用send函数向第39行连接的客户端传输数据。稍后讲解send函数。
- 第46行：程序终止前注销第20行中初始化的套接字库。

可以看出，除了Winsock库的初始化和注销相关代码、数据类型信息外，其余部分与Linux环境下的示例并无区别。希望各位阅读这部分代码时与之前的Linux服务器端进行逐行比较。接下来介绍与此示例同步的客户端代码。

❖ hello_client_win.c

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

int main(int argc, char* argv[])
{
    WSADATA wsaData;
    SOCKET hSocket;
    SOCKADDR_IN servAddr;

    char message[30];
    int strLen;
    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(&servAddr, 0, sizeof(servAddr));
    servAddr.sin_family=AF_INET;
    servAddr.sin_addr.s_addr=inet_addr(argv[1]);
    servAddr.sin_port=htons(atoi(argv[2]));

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

    strLen=recv(hSocket, message, sizeof(message)-1, 0);
    if(strLen==-1)
        ErrorHandling("read() error!");
    printf("Message from server: %s \n", message);

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

void ErrorHandling(char* message)
{
```

```
47.     fputs(message, stderr);
48.     fputc('\n', stderr);
49.     exit(1);
50. }
```

代码说明

- 第20行：初始化Winsock库。
- 第23、32行：第23行创建套接字，第32行通过此套接字向服务器端发出连接请求。
- 第35行：调用recv函数接收服务器发来的数据。稍后讲解该函数。
- 第41行：注销第20行中初始化的Winsock库。

下面运行以上示例。创建编译项目的过程与各位学习C语言时使用的方法相同，只是增加了设置ws2_32.lib链接库的过程。

❖ 运行结果：hello_server_win.c

```
C:\tcpip>hServerWin 9190
```

运行过程中，假设可执行文件名为hServerWin.exe。如果运行正常，则与Linux相同，程序进入等待状态。这是因为服务器端调用了accept函数。接着运行客户端，假设客户端的可执行文件名为hClientWin.exe。

❖ 运行结果：hello_client_win.c

```
C:\tcpip>hClientWin 127.0.0.1 9190
Message from server: Hello World!
```

基于 Windows 的 I/O 函数

Linux中套接字也是文件，因而可以通过文件I/O函数read和write进行数据传输。而Windows中则有些不同。Windows严格区分文件I/O函数和套接字I/O函数。下面介绍Winsock数据传输函数。

```
#include <winsock2.h>

int send(SOCKET s, const char * buf, int len, int flags);
```

➔ 成功时返回传输字节数，失败时返回 SOCKET_ERROR。

- s　　表示数据传输对象连接的套接字句柄值。
- buf　　保存待传输数据的缓冲地址值。
- len　　要传输的字节数。
- flags　　传输数据时用到的多种选项信息。

此函数与Linux的write函数相比，只是多出了最后的flags参数。后续章节中将给出该参数的详细说明，在此之前只需传递0，表示不设置任何选项。但有一点需要注意，send函数并非Windows独有。Linux中也有同样的函数，它也来自于BSD套接字。只不过我在Linux相关示例中暂时只使用read、write函数，为了强调Linux环境下文件I/O和套接字I/O相同。下面介绍与send函数对应的recv函数。

```
#include <winsock2.h>

int recv(SOCKET s, const char * buf, int len, int flags);
```

➔ 成功时返回接收的字节数（收到 EOF 时为 0），失败时返回 SOCKET_ERROR。

- s　表示数据接收对象连接的套接字句柄值。
- buf　保存接收数据的缓冲地址值。
- len　能够接收的最大字节数。
- flags　接收数据时用到的多种选项信息。

我只是在Windows环境下提前介绍了send、recv函数，以后的Linux示例中也会涉及。请不要误认为Linux中的read、write函数就是对应于Windows的send、recv函数。另外，之前的程序代码中也给出了send、recv函数调用过程，故不再另外给出相关示例。

知识补给站　Windows? Linux?

过去要编写服务器端的话，大部分程序员都会想起Linux或UNIX，因为那时的Windows还被认为是只能给个人使用的操作系统。即使Windows已经开始提供运营服务器端所需环境，但绝大多数网络程序员都会选择Linux和UNIX。不过，随着多媒体数据传输要求的提高，程序员的想法也有了变化。他们会根据服务器端的特点和环境选择不同的操作系统。如果各位想成为网络编程专家，就必须具备跨平台编程能力。

1.5　习题

(1) 套接字在网络编程中的作用是什么？为何称它为套接字？

(2) 在服务器端创建套接字后，会依次调用listen函数和accept函数。请比较并说明二者作用。

(3) Linux中，对套接字数据进行I/O时可以直接使用文件I/O相关函数；而在Windows中则不可以。原因为何？

(4) 创建套接字后一般会给它分配地址，为什么？为了完成地址分配需要调用哪个函数？

(5) Linux中的文件描述符与Windows的句柄实际上非常类似。请以套接字为对象说明它们的含义。

(6) 底层文件I/O函数与ANSI标准定义的文件I/O函数之间有何区别?

(7) 参考本书给出的示例low_open.c和low_read.c，分别利用底层文件I/O和ANSI标准I/O编写文件复制程序。可任意指定复制程序的使用方法。

第2章 套接字类型与协议设置

因为涉及套接字编程的基本内容，所以第2章和第3章显得相对枯燥一些。但本章内容是第4章介绍的实际网络编程的基础，希望各位反复精读。

大家已经对套接字的概念有所理解，本章将介绍套接字创建方法及不同套接字的特性。在本章仅需了解创建套接字时调用的socket函数，所以希望大家以放松的心态开始学习。

2.1 套接字协议及其数据传输特性

“协议”这个词给人的第一印象总是相当困难，我在学生时代也这么想。但各位要慢慢熟悉“协议”，因为它几乎是网络编程的全部内容。首先解释其定义。

关于协议（Protocol）

如果相隔很远的两人想展开对话，必须先决定对话方式。如果一方使用电话，那么另一方也只能使用电话，而不是书信。可以说，电话就是两人对话的协议。协议是对话中使用的通信规则，把上述概念拓展到计算机领域可整理为“计算机间对话必备通信规则”。

各位是否已理解了协议的含义？简言之，协议就是为了完成数据交换而定好的约定。

创建套接字

创建套接字所用的socket函数已经在第1章中简单介绍过。但为了完全理解该函数，此处将再次展开讨论，本章的主要目的也在于此。

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

int socket(int domain, int type, int protocol);
```

➔ 成功时返回文件描述符，失败时返回-1。

- domain　套接字中使用的协议族（Protocol Family）信息。
- type　套接字数据传输类型信息。
- protocol　计算机间通信中使用的协议信息。

第1章并未提及该函数的参数，但它们对创建套接字来说是不可或缺的。下面给出详细说明。

协议族（Protocol Family）

奶油意大利面和番茄酱意大利面均属于意大利面的一种，与之类似，套接字通信中的协议也具有一些分类。通过socket函数的第一个参数传递套接字中使用的协议分类信息。此协议分类信息称为协议族，可分成如下几类。

表2-1　头文件sys/socket.h中声明的协议族

名　称	协 议 族
PF_INET	IPv4互联网协议族
PF_INET6	IPv6互联网协议族
PF_LOCAL	本地通信的UNIX协议族
PF_PACKET	底层套接字的协议族
PF_IPX	IPX Novell协议族

本书将着重讲解表2-1中PF_INET对应的IPv4互联网协议族。其他协议族并不常用或尚未普及，因此本书将重点放在PF_INET协议族上。另外，套接字中实际采用的最终协议信息是通过socket函数的第三个参数传递的。在指定的协议族范围内通过第一个参数决定第三个参数。

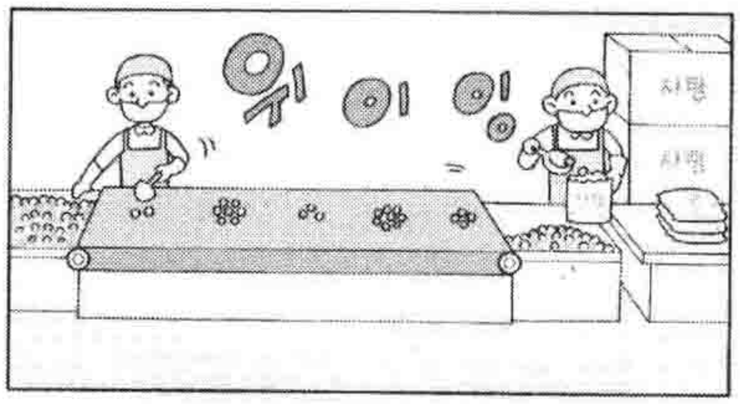
套接字类型（Type）

套接字类型指的是套接字的数据传输方式，通过socket函数的第二个参数传递，只有这样才能决定创建的套接字的数据传输方式。这种说法可能会使各位感到疑惑。已通过第一个参数传递了协议族信息，还要决定数据传输方式？问题就在于，决定了协议族并不能同时决定数据传输方式，换言之，socket函数第一个参数PF_INET协议族中也存在多种数据传输方式。

下面介绍2种具有代表性的数据传输方式。这是理解好套接字的重要前提，请各位务必掌握。

套接字类型 1：面向连接的套接字（SOCK_STREAM）

如果向socket函数的第二个参数传递SOCK_STREAM，将创建面向连接的套接字。面向连接的套接字到底具有哪些特点呢？右图中2位工人通过1条传送带传递物品，这与面向连接的数据传输方式类似。



右图的数据（糖果）传输方式特征整理如下。

- 传输过程中数据不会消失。
- 按序传输数据。
- 传输的数据不存在数据边界（Boundary）。

图中通过独立的传送带传输数据（糖果），只要传送带本身没有问题，就能保证数据不丢失。同时，较晚传递的数据不会先到达，因为传送带保证了数据的按序传递。最后，下面这句话说明的确不存在数据边界：

“100个糖果是分批传递的，但接收者凑齐100个后才装袋。”

这种情形可以适用到之前说过的write和read函数。

“传输数据的计算机通过3次调用write函数传递了100字节的数据，但接收数据的计算机仅通过1次read函数调用就接收了全部100个字节。”

收发数据的套接字内部有缓冲（buffer），简言之就是字节数组。通过套接字传输的数据将保存到该数组。因此，收到数据并不意味着马上调用read函数。只要不超过数组容量，则有可能在数据填充满缓冲后通过1次read函数调用读取全部，也有可能分成多次read函数调用进行读取。也就是说，在面向连接的套接字中，read函数和write函数的调用次数并无太大意义。所以说面向连接的套接字不存在数据边界。稍后将给出示例以查看该特性。

知识补给站　套接字缓冲已满是否意味着数据丢失

之前讲过，为了接收数据，套接字内部有一个由字节数组构成的缓冲。如果这个缓冲被接收的数据填满会发生什么事情？之后传递的数据是否会丢失？

首先调用read函数从缓冲读取部分数据，因此，缓冲并不总是满的。但如果read函数读取速度比接收数据的速度慢，则缓冲有可能被填满。此时套接字无法再接收数据，但即使这样也不会发生数据丢失，因为传输端套接字将停止传输。也就是说，面向连接的套接字会根据接收端的状态传输数据，如果传输出错还会提供重传服务。因此，面向连接的套接字除特殊情况外不会发生数据丢失。

还有一点需要说明。上图中传输和接收端各有1名工人，这说明面向连接的套接字还有如下特点：

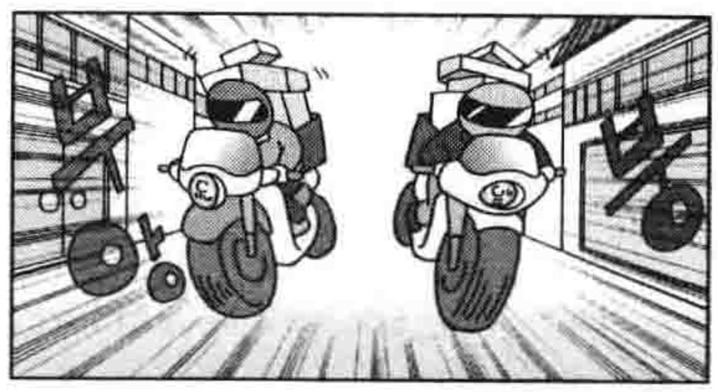
“套接字连接必须一一对应。”

面向连接的套接字只能与另外一个同样特性的套接字连接。用一句话概括面向连接的套接字如下：

“可靠的、按序传递的、基于字节的面向连接的数据传输方式的套接字”

这是我自己的总结，希望各位深入理解其含义，不要仅停留于字面表达。

套接字类型 2：面向消息的套接字（SOCK_DGRAM）



如果向socket函数的第二个参数传递SOCK_DGRAM，则将创建面向消息的套接字。面向消息的套接字可以比喻成高速移动的摩托车快递。右图中摩托车快递的包裹（数据）传输方式如下。

- 强调快速传输而非传输顺序。
- 传输的数据可能丢失也可能损毁。
- 传输的数据有数据边界。
- 限制每次传输的数据大小。

众所周知，快递行业的速度就是生命。用摩托车发往同一目的地的2件包裹无需保证顺序，只要以最快速度交给客户即可。这种方式存在损坏或丢失的风险，而且包裹大小有一定限制。因此，若要传递大量包裹，则需分批发送。另外，如果用2辆摩托车分别发送2件包裹，则接收者也需要分2次接收。这种特性就是“传输的数据具有数据边界”。

以上就是面向消息的套接字具有的特性。即，面向消息的套接字比面向连接的套接字具有更快的传输速度，但无法避免数据丢失或损毁。另外，每次传输的数据大小具有一定限制，并存在数据边界。存在数据边界意味着接收数据的次数应和传输次数相同。面向消息的套接字特性总结如下：

“不可靠的、不按序传递的、以数据的高速传输为目的的套接字”

另外，面向消息的套接字不存在连接的概念，这一点将在以后章节介绍。

协议的最终选择

下面讲解socket函数的第三个参数，该参数决定最终采用的协议。各位是否觉得有些困惑？

前面已经通过socket函数的前两个参数传递了协议族信息和套接字数据传输方式，这些信息还不足以决定采用的协议吗？为什么还需要传递第3个参数呢？

正如各位所想，传递前两个参数即可创建所需套接字。所以大部分情况下可以向第三个参数传递0，除非遇到以下这种情况：

“同一协议族中存在多个数据传输方式相同的协议”

数据传输方式相同，但协议不同。此时需要通过第三个参数具体指定协议信息。

下面以前面讲解的内容为基础，构建向socket函数传递的参数。首先创建满足如下要求的套接字：

“IPv4协议族中面向连接的套接字”

IPv4与网络地址系统相关，关于这一点将给出单独说明，目前只需记住：本书是基于IPv4展开的。参数PF_INET指IPv4网络协议族，SOCK_STREAM是面向连接的数据传输。满足这2个条件的协议只有IPPROTO_TCP，因此可以如下调用socket函数创建套接字，这种套接字称为TCP套接字。

```
int tcp_socket = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
```

下面创建满足如下要求的套接字：

“IPv4协议族中面向消息的套接字”

SOCK_DGRAM指的是面向消息的数据传输方式，满足上述条件的协议只有IPPROTO_UDP。因此，可以如下调用socket函数创建套接字，这种套接字称为UDP套接字。

```
int udp_socket = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);
```

前面进行了大量描述以解释这两行代码，这是为了让大家理解它们创建的套接字的特性。

面向连接的套接字：TCP 套接字示例

其他章节将讲解UDP套接字，此处只给出面向连接的TCP套接字示例。本示例是在第1章的如下2个源文件基础上修改而成的。

- hello_server.c → tcp_server.c：无变化！
- hello_client.c → tcp_client.c：更改read函数调用方式！

之前的hello_server.c和hello_client.c是基于TCP套接字的示例，现调整其中一部分代码，以验证TCP套接字的如下特性：

“传输的数据不存在数据边界。”

为验证这一点，需要让write函数的调用次数不同于read函数的调用次数。因此，在客户端中分多次调用read函数以接收服务器端发送的全部数据。

❖ tcp_client.c

```
1.  #include <“头信息与hello_client.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.      struct sockaddr_in serv_addr;
8.      char message[30];
9.      int str_len=0;
10.     int idx=0, read_len=0;
11.
12.     if(argc!=3){
13.         printf("Usage : %s <IP> <port>\n", argv[0]);
14.         exit(1);
15.     }
16.
17.     sock=socket(PF_INET, SOCK_STREAM, 0);
18.     if(sock == -1)
19.         error_handling("socket() error");
20.
21.     memset(&serv_addr, 0, sizeof(serv_addr));
22.     serv_addr.sin_family=AF_INET;
23.     serv_addr.sin_addr.s_addr=inet_addr(argv[1]);
24.     serv_addr.sin_port=htons(atoi(argv[2]));
25.
26.     if(connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr))==-1)
27.         error_handling("connect() error!");
28.
29.     while(read_len=read(sock, &message[idx++], 1))
30.     {
31.         if(read_len==-1)
32.             error_handling("read() error!");
33.
34.         str_len+=read_len;
35.     }
36.
37.     printf("Message from server: %s \n", message);
38.     printf("Function read call count: %d \n", str_len);
39.     close(sock);
40.     return 0;
41. }
42.
43. void error_handling(char *message)
44. {
45.     //与以前示例一致，故省略!
46. }
```

代码说明

- 第17行：创建TCP套接字。若前两个参数传递PF_INET、SOCK_STREAM，则可以省略第三个参数IPPROTO_TCP。
- 第29行：while循环中反复调用read函数，每次读取1个字节。如果read返回0，则循环条件为假，跳出while循环。
- 第34行：执行该语句时，变量read_len的值始终为1，因为第29行每次读取1个字节。跳出while循环后，str_len中存有读取的总字节数。

与该示例配套使用的服务器端tcp_server.c与hello_server.c完全相同，故省略其源代码。执行方式也与hello_server.c和hello_client.c相同，因此只给出最终运行结果。

❖ 运行结果：hello_client.c

```
root@my_linux:/tcpip# gcc tcp_client.c -o hclient
root@my_linux:/tcpip# ./hclient 127.0.0.1 9190
Message from server: Hello World!
Function read call count: 13
```

从运行结果可以看出，服务器端发送了13字节的数据，客户端调用13次read函数进行读取。希望各位通过该示例深入理解TCP套接字的数据传输方式。

2.2　Windows 平台下的实现及验证

前面讲过的套接字类型及传输特性与操作系统无关。Windows平台下的实现方式也类似，不需要过多说明，只需稍加了解socket函数返回类型即可。

Windows 操作系统的 socket 函数

Windows的函数名和参数名都与Linux平台相同，只是返回值类型稍有不同。再次给出socket函数的声明。

```
#include <winsock2.h>

SOCKET socket(int af, int type, int protocol);
```

➔ 成功时返回 socket 句柄，失败时返回 INVALID_SOCKET。

该函数的参数种类及含义与Linux的socket函数完全相同，故省略，只讨论返回值类型。可以看出返回值类型为SOCKET，此结构体用来保存整数型套接字句柄值。实际上，socket函数返回整数型数据，因此可以通过int型变量接收，就像在Linux中做的一样。但考虑到以后的扩展性，

定义为SOCKET数据类型，希望各位也使用SOCKET结构体变量保存套接字句柄，这也是微软希望看到的。以后即可将SOCKET视作保存套接字句柄的一个数据类型。

同样，发生错误时返回INVALID_SOCKET，只需将其理解为提示错误的常数即可。其实际值为-1，但值是否为-1并不重要，除非编写如下代码。

```
SOCKET soc = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
if ( soc == -1 )
      ErrorHandling(". . .");
```

如果这样编写代码，那么微软定义的INVALID_SOCKET常数将失去意义！应该如下编写，这样，即使日后微软更改INVALID_SOCKET常数值，也不会发生问题。

```
SOCKET soc = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
if ( soc == INVALID_SOCKET )
      ErrorHandling(". . .");
```

这些问题虽然琐碎却非常重要。

基于 Windows 的 TCP 套接字示例

把之前的tcp_server.c、tcp_client.c如下改为基于Windows的程序。

- hello_server_win.c → tcp_server_win.c：无变化!
- Hello_client_win.c → tcp_client_win.c：更改read函数调用方式!

与之前一样，只给出tcp_client_win.c源代码及运行结果。各位若想亲自查看tcp_server_win.c的代码，可以参考第1章的hello_server_win.c，或到OrangeMedia主页（http://www.orentec.co.kr/）下载源代码。

❖ tcp_client_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.      SOCKET hSocket;
10.     SOCKADDR_IN servAddr;
11.
12.     char message[30];
13.     int strLen=0;
```

```
14.     int idx=0, readLen=0;
15.
16.     if(argc!=3)
17.     {
18.         printf("Usage : %s <IP> <port>\n", argv[0]);
19.         exit(1);
20.     }
21.
22.     if(WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
23.         ErrorHandling("WSAStartup() error!");
24.
25.     hSocket=socket(PF_INET, SOCK_STREAM, 0);
26.     if(hSocket==INVALID_SOCKET)
27.         ErrorHandling("hSocket() error");
28.
29.     memset(&servAddr, 0, sizeof(servAddr));
30.     servAddr.sin_family=AF_INET;
31.     servAddr.sin_addr.s_addr=inet_addr(argv[1]);
32.     servAddr.sin_port=htons(atoi(argv[2]));
33.
34.     if(connect(hSocket, (SOCKADDR*)&servAddr, sizeof(servAddr))==SOCKET_ERROR)
35.         ErrorHandling("connect() error!");
36.
37.     while(readLen=recv(hSocket, &message[idx++], 1, 0))
38.     {
39.         if(readLen==-1)
40.             ErrorHandling("read() error!");
41.
42.         strLen+=readLen;
43.     }
44.
45.     printf("Message from server: %s \n", message);
46.     printf("Function read call count: %d \n", strLen);
47.
48.     closesocket(hSocket);
49.     WSACleanup();
50.     return 0;
51. }
52.
53. void ErrorHandling(char* message)
54. {
55.     fputs(message, stderr);
56.     fputc('\n', stderr);
57.     exit(1);
58. }
```

代码说明

- 第9、25行：第9行声明SOCKET变量以保存socket函数返回值。第25行调用socket函数创建TCP套接字，各位应该感到眼熟。
- 第37行：while循环中调用recv函数读取数据，每次1个字节。
- 第42行：第37行中每次读取1个字节，因此变量strLen每次加1，这与recv函数调用次数相同。

❖ 运行结果：tcp_client_win.c

```
C:\tcpip>hTCPClientWin 127.0.0.1 9190
Message from server: Hello World!
Function read call count: 13
```


该示例的运行方式与第1章的hello_server_win.c、hello_client_win.c相同，因此只给出客户端的运行结果。以上就是第2章的全部内容，相信各位对服务器端和客户端有了更深入的理解。

2.3 习题

(1) 什么是协议？在收发数据中定义协议有何意义？

(2) 面向连接的TCP套接字传输特性有3点，请分别说明。

(3) 下列哪些是面向消息的套接字的特性？

a. 传输数据可能丢失
b. 没有数据边界（Boundary）
c. 以快速传递为目标
d. 不限制每次传递数据的大小
e. 与面向连接的套接字不同，不存在连接的概念

(4) 下列数据适合用哪类套接字传输？并给出原因。

a. 演唱会现场直播的多媒体数据（ ）
b. 某人压缩过的文本文件（ ）
c. 网上银行用户与银行之间的数据传递（ ）

(5) 何种类型的套接字不存在数据边界？这类套接字接收数据时需要注意什么？

(6) tcp_server.c和tcp_client.c中需多次调用read函数读取服务器端调用1次write函数传递的字符串。更改程序，使服务器端多次调用（次数自拟）write函数传输数据，客户端调用1次read函数进行读取。为达到这一目的，客户端需延迟调用read函数，因为客户端要等待服务器端传输所有数据。Windows和Linux都通过下列代码延迟read或recv函数的调用。

```
for(i=0; i<3000; i++)
   printf("Wait time %d \n", i);
```

让CPU执行多余任务以延迟代码运行的方式称为“Busy Waiting”。使用得当即可推迟函数调用。

第3章

地址族与数据序列

3

第2章中讨论了套接字的创建方法，如果把套接字比喻为电话，那么目前只安装了电话机。本章将着重讲解给电话机分配号码的方法，即给套接字分配 IP 地址和端口号。这部分内容也相对有些枯燥，但并不难，而且是学习后续那些有趣内容必备的基础知识。

3.1 分配给套接字的 IP 地址与端口号

IP是Internet Protocol（网络协议）的简写，是为收发网络数据而分配给计算机的值。端口号并非赋予计算机的值，而是为区分程序中创建的套接字而分配给套接字的序号。下面逐一讲解。

网络地址（Internet Address）

为使计算机连接到网络并收发数据，必需向其分配IP地址。IP地址分为两类。

- IPv4 (Internet Protocol version 4)　4字节地址族
- IPv6 (Internet Protocol version 6)　16字节地址族

IPv4与IPv6的差别主要是表示IP地址所用的字节数，目前通用的地址族为IPv4。IPv6是为了应对2010年前后IP地址耗尽的问题而提出的标准，即便如此，现在还是主要使用IPv4，IPv6的普及将需要更长时间。

IPv4标准的4字节IP地址分为网络地址和主机（指计算机）地址，且分为A、B、C、D、E等类型。图3-1展示了IPv4地址族，一般不会使用已被预约了的E类地址，故省略。

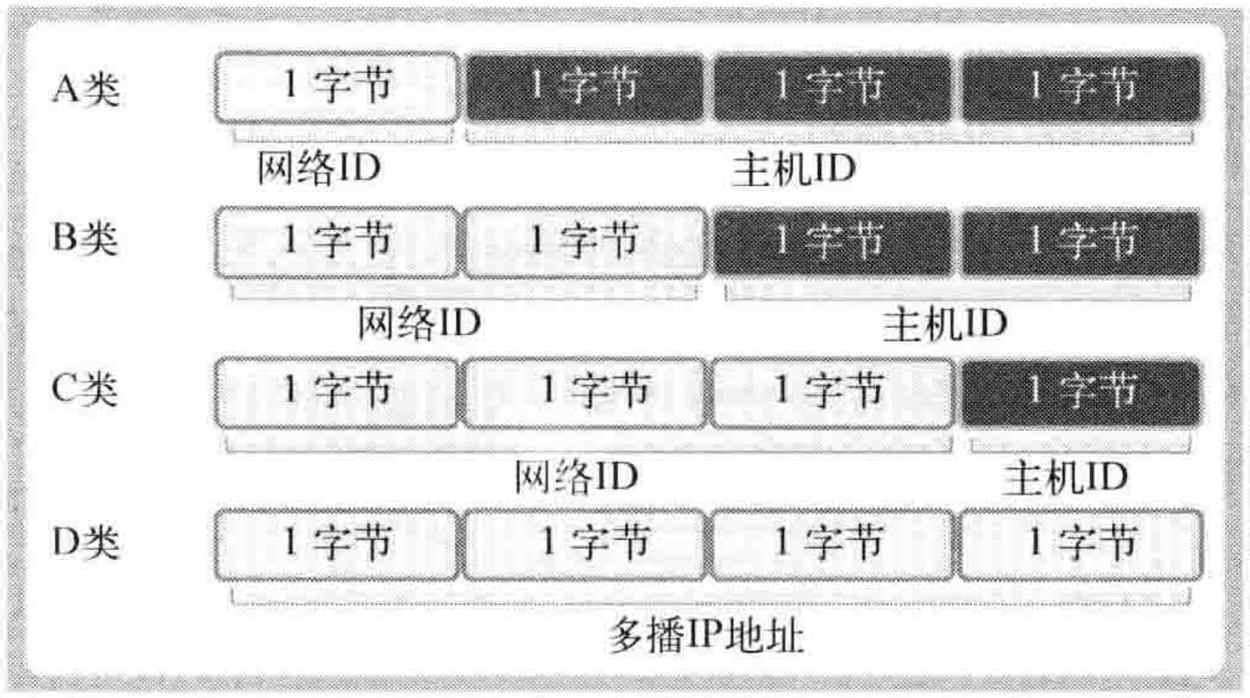


图3-1 IPv4地址族

网络地址（网络ID）是为区分网络而设置的一部分IP地址。假设向WWW.SEMI.COM公司传输数据，该公司内部构建了局域网，把所有计算机连接起来。因此，首先应向SEMI.COM网络传输数据，也就是说，并非一开始就浏览所有4字节IP地址，进而找到目标主机；而是仅浏览4字节IP地址的网络地址，先把数据传到SEMI.COM的网络。SEMI.COM网络（构成网络的路由器）接收到数据后，浏览传输数据的主机地址（主机ID）并将数据传给目标计算机。图3-2展示了数据传输过程。

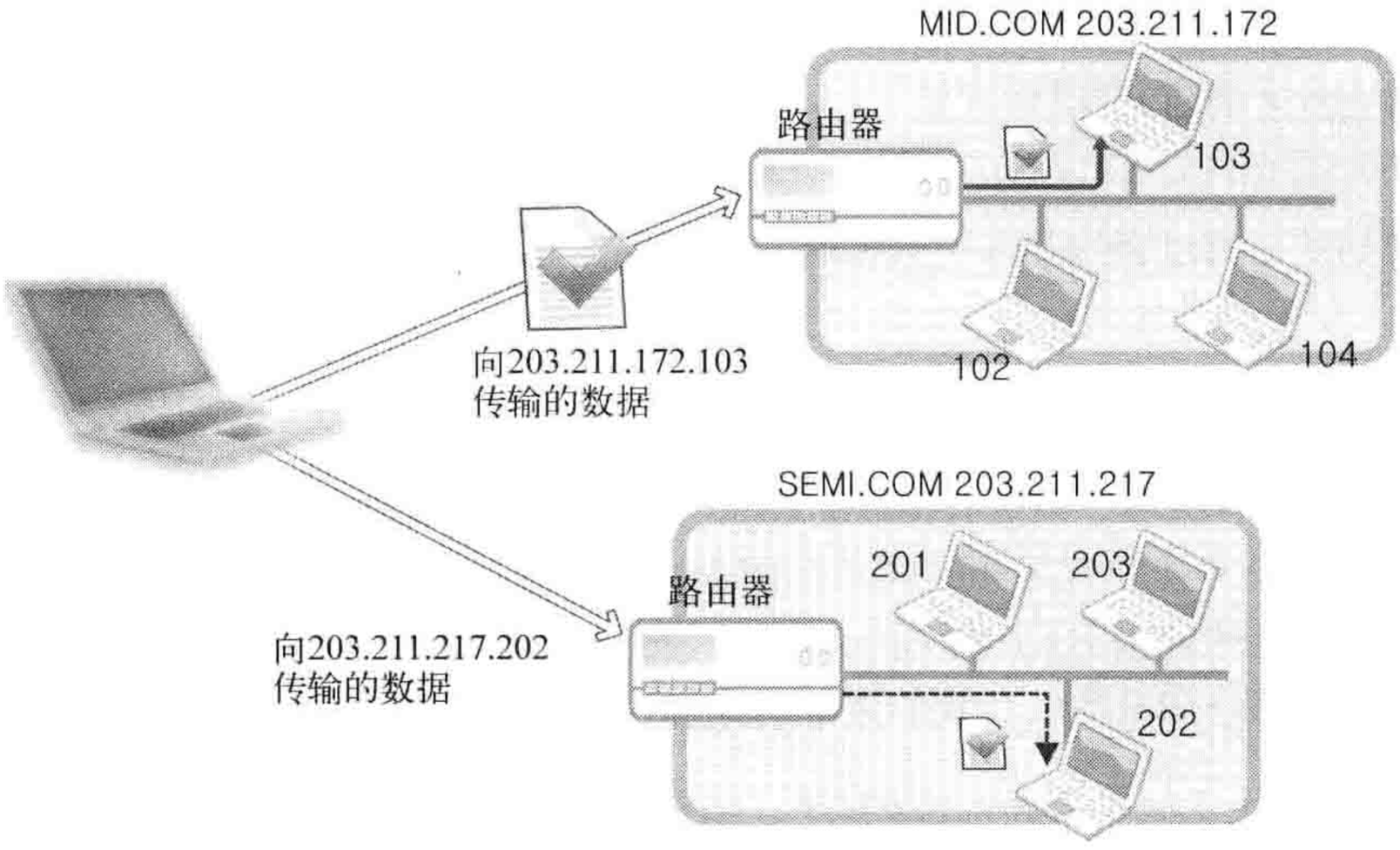


图3-2 基于IP地址的数据传输过程

某主机向203.211.172.103和203.211.217.202传输数据，其中203.211.172和203.211.217为该网络的网络地址（稍后将给出网络地址的区分方法）。所以，“向相应网络传输数据”实际上是向构成网络的路由器（Router）或交换机（Switch）传递数据，由接收数据的路由器根据数据中的主机地址向目标主机传递数据。

知识补给站　**路由器和交换机**

若想构建网络，需要一种物理设备完成外网与本网主机之间的数据交换，这种设备便是路由器或交换机。它们实际上也是一种计算机，只不过是为特殊目的而设计运行的，因此有了别名。所以，如果在我们使用的计算机上安装适当的软件，也可以将其用作交换机。另外，交换机比路由器功能要简单一些，而实际用途差别不大。

网络地址分类与主机地址边界

只需通过IP地址的第一个字节即可判断网络地址占用的字节数，因为我们根据IP地址的边界区分网络地址，如下所示。

- A类地址的首字节范围：0~127
- B类地址的首字节范围：128~191
- C类地址的首字节范围：192~223

还有如下这种表述方式。

- A类地址的首位以0开始
- B类地址的前2位以10开始
- C类地址的前3位以110开始

正因如此，通过套接字收发数据时，数据传到网络后即可轻松找到正确的主机。

用于区分套接字的端口号

IP用于区分计算机，只要有IP地址就能向目标主机传输数据，但仅凭这些无法传输给最终的应用程序。假设各位欣赏视频的同时在网上冲浪，这时至少需要1个接收视频数据的套接字和1个接收网页信息的套接字。问题在于如何区分二者。简言之，传输到计算机的网络数据是发给播放器，还是发送给浏览器？让我们更准确地描述问题。假设各位开发了如下应用程序：

> “我开发了收发数据的P2P程序，该程序用块单位分割1个文件，从多台计算机接收数据。”

假设各位对P2P有一定了解，即便不清楚也无所谓。如上所述，若想接收多台计算机发来的数据，则需要相应个数的套接字。那如何区分这些套接字呢？

计算机中一般配有NIC（Network Interface Card，网络接口卡）数据传输设备。通过NIC向计算机内部传输数据时会用到IP。操作系统负责把传递到内部的数据适当分配给套接字，这时就要

利用端口号。也就是说，通过NIC接收的数据内有端口号，操作系统正是参考此端口号把数据传输给相应端口的套接字，如图3-3所示。

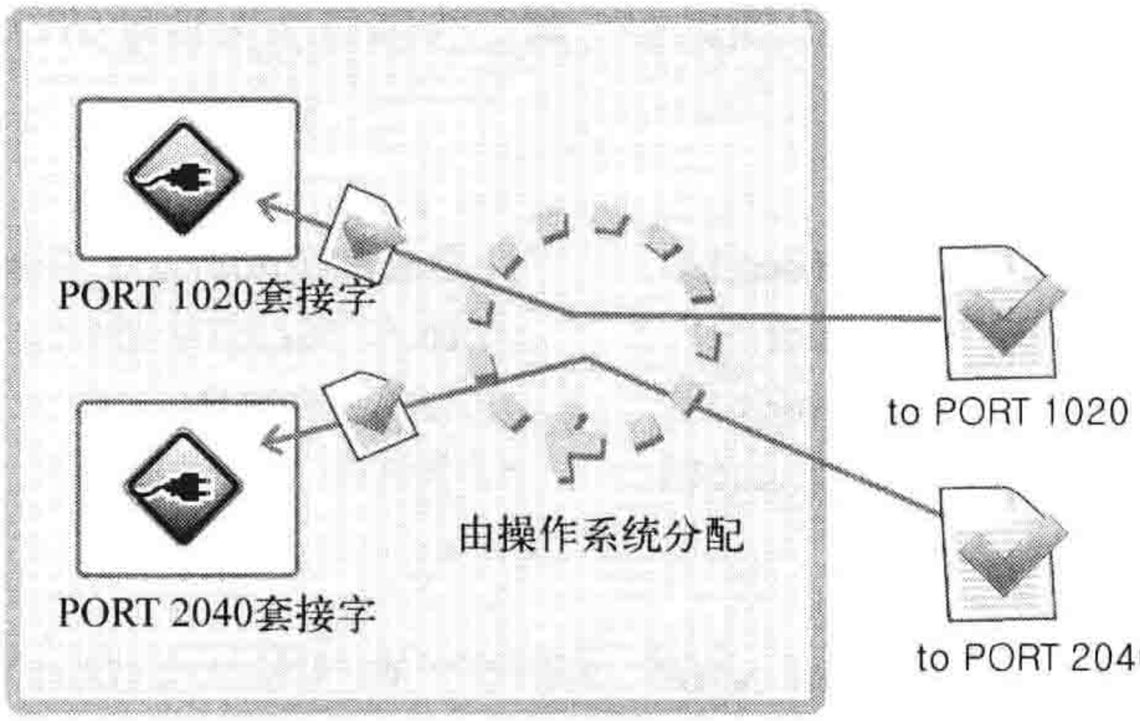


图3-3 数据分配过程

端口号就是在同一操作系统内为区分不同套接字而设置的，因此无法将1个端口号分配给不同套接字。另外，端口号由16位构成，可分配的端口号范围是0-65535。但0-1023是知名端口（Well-known PORT），一般分配给特定应用程序，所以应当分配此范围之外的值。另外，虽然端口号不能重复，但TCP套接字和UDP套接字不会共用端口号，所以允许重复。例如：如果某TCP套接字使用9190号端口，则其他TCP套接字就无法使用该端口号，但UDP套接字可以使用。

总之，数据传输目标地址同时包含IP地址和端口号，只有这样，数据才会被传输到最终的目的应用程序（应用程序套接字）。

3.2 地址信息的表示

应用程序中使用的IP地址和端口号以结构体的形式给出了定义。本节将以IPv4为中心，围绕此结构体讨论目标地址的表示方法。

表示 IPv4 地址的结构体

填写地址信息时应以如下提问为线索进行，各位读过下列对话后也会同意这一点。

- 问题1：“采用哪一种地址族？”
- 答案1：“基于IPv4的地址族。”

- 问题2：“IP地址是多少？”
- 答案2：“211.204.214.76。”

- 问题3：“端口号是多少？”
- 答案3：“2048。”

结构体定义为如下形态就能回答上述提问，此结构体将作为地址信息传递给bind函数。

```
struct sockaddr_in
{
    sa_family_t         sin_family;     //地址族（Address Family）
    uint16_t            sin_port;       //16 位 TCP/UDP 端口号
    struct in_addr      sin_addr;       //32 位 IP 地址
    char                sin_zero[8];    //不使用
};
```

该结构体中提到的另一个结构体in_addr定义如下，它用来存放32位IP地址。

```
struct in_addr
{
    In_addr_t           s_addr;         //32 位 IPv4 地址
};
```

讲解以上2个结构体前先观察一些数据类型。uint16_t、in_addr_t等类型可以参考POSIX（Portable Operating System Interface，可移植操作系统接口）。POSIX是为UNIX系列操作系统设立的标准，它定义了一些其他数据类型，如表3-1所示。

表3-1　POSIX中定义的数据类型

数据类型名称	数据类型说明	声明的头文件
int8_t	signed 8-bit int	sys/types.h
uint8_t	unsigned 8-bit int (unsigned char)	
int16_t	signed 16-bit int	
uint16_t	unsigned 16-bit int(unsigned short)	
int32_t	signed 32-bit int	
uint32_t	unsigned 32-bit int(unsigned long)	
sa_family_t	地址族（address family）	sys/socket.h
socklen_t	长度（length of struct）	
in_addr_t	IP地址，声明为uint32_t	netinet/in.h
in_port_t	端口号，声明为uint16_t	

从这些数据类型声明也可掌握之前结构体的含义。那为什么需要额外定义这些数据类型呢？如前所述，这是考虑到扩展性的结果。如果使用int32_t类型的数据，就能保证在任何时候都占用4字节，即使将来用64位表示int类型也是如此。

结构体 sockaddr_in 的成员分析

接下来重点观察结构体成员的含义及其包含的信息。

成员sin_family

每种协议族适用的地址族均不同。比如，IPv4使用4字节地址族，IPv6使用16字节地址族。可以参考表3-2保存sin_family地址信息。

表3-2 地址族

地址族（Address Family）	含 义
AF_INET	IPv4网络协议中使用的地址族
AF_INET6	IPv6网络协议中使用的地址族
AF_LOCAL	本地通信中采用的UNIX协议的地址族

AF_LOCAL只是为了说明具有多种地址族而添加的，希望各位不要感到太突然。

成员sin_port

该成员保存16位端口号，重点在于，它以网络字节序保存（关于这一点稍后将给出详细说明）。

成员sin_addr

该成员保存32位IP地址信息，且也以网络字节序保存。为理解好该成员，应同时观察结构体in_addr。但结构体in_addr声明为uint32_t，因此只需当作32位整数型即可。

成员sin_zero

无特殊含义。只是为使结构体sockaddr_in的大小与sockaddr结构体保持一致而插入的成员。必需填充为0，否则无法得到想要的结果。后面会另外讲解sockaddr。

从之前介绍的代码也可看出，sockaddr_in结构体变量地址值将以如下方式传递给bind函数。稍后将给出关于bind函数的详细说明，希望各位重点关注参数传递和类型转换部分的代码。

```
struct sockaddr_in serv_addr;
....
if(bind(serv_sock, (struct sockaddr * ) &serv_addr, sizeof(serv_addr)) == -1)
    error_handling("bind() error");
....
```

此处重要的是第二个参数的传递。实际上，bind函数的第二个参数期望得到sockaddr结构体变量地址值，包括地址族、端口号、IP地址等。从下列代码也可看出，直接向sockaddr结构体填充这些信息会带来麻烦。

```
struct sockaddr
{
```

```
    sa_family_t     sin_family;     // 地址族（Address Family）
    char            sa_data[14];    // 地址信息
};
```

此结构体成员sa_data保存的地址信息中需包含IP地址和端口号，剩余部分应填充0，这也是bind函数要求的。而这对于包含地址信息来讲非常麻烦，继而就有了新的结构体sockaddr_in。若按照之前的讲解填写sockaddr_in结构体，则将生成符合bind函数要求的字节流。最后转换为sockaddr型的结构体变量，再传递给bind函数即可。

知识补给站　sin_family

sockaddr_in是保存IPv4地址信息的结构体。那为何还需要通过sin_family单独指定地址族信息呢？这与之前讲过的sockaddr结构体有关。结构体sockaddr并非只为IPv4设计，这从保存地址信息的数组sa_data长度为14字节也可看出。因此，结构体sockaddr要求在sin_family中指定地址族信息。为了与sockaddr保持一致，sockaddr_in结构体中也有地址族信息。

3.3　网络字节序与地址变换

不同CPU中，4字节整数型值1在内存空间的保存方式是不同的。4字节整数型值1可用2进制表示如下。

```
00000000 00000000 00000000 00000001
```

有些CPU以这种顺序保存到内存，另外一些CPU则以倒序保存。

```
00000001 00000000 00000000 00000000
```

若不考虑这些就收发数据则会发生问题，因为保存顺序的不同意味着对接收数据的解析顺序也不同。

字节序（Order）与网络字节序

CPU向内存保存数据的方式有2种，这意味着CPU解析数据的方式也分为2种。

- 大端序（Big Endian）：高位字节存放到低位地址。
- 小端序（Little Endian）：高位字节存放到高位地址。

仅凭描述很难解释清楚，下面通过示例进行说明。假设在0x20号开始的地址中保存4字节int类型数0x12345678。大端序CPU保存方式如图3-4所示。

图3-4 大端序字节表示

整数0x12345678中，0x12是最高位字节，0x78是最低位字节。因此，大端序中先保存最高位字节0x12（最高位字节0x12存放到低位地址）。小端序保存方式如图3-5所示。

图3-5 小端序字节表示

先保存的是最低位字节0x78。从以上分析可以看出，每种CPU的数据保存方式均不同。因此，代表CPU数据保存方式的主机字节序（Host Byte Order）在不同CPU中也各不相同。目前主流的Intel系列CPU以小端序方式保存数据。接下来分析2台字节序不同的计算机之间数据传递过程中可能出现的问题，如图3-6所示。

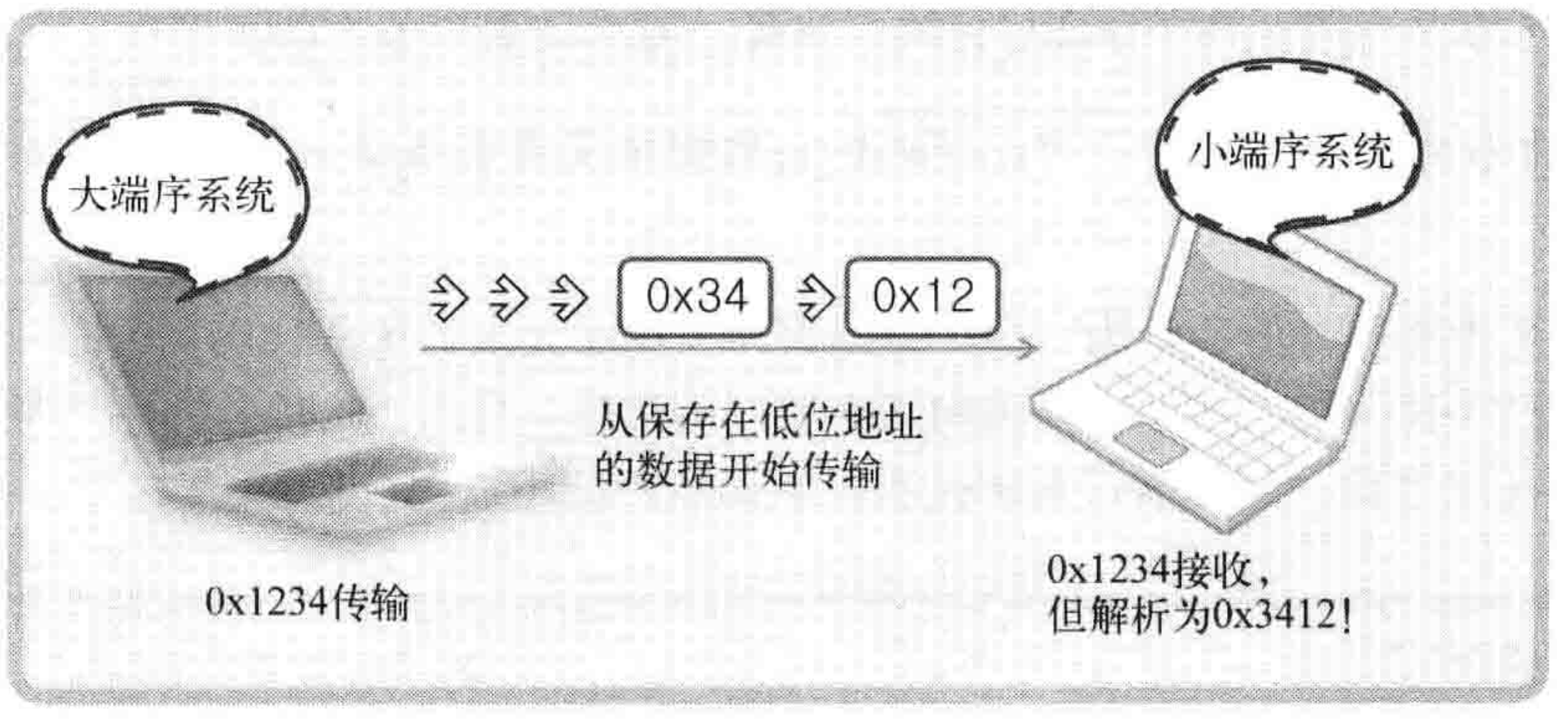


图3-6 字节序问题

0x12和0x34构成的大端序系统值与0x34和0x12构成的小端序系统值相同。换言之，只有改变数据保存顺序才能被识别为同一值。图3-6中，大端序系统传输数据0x1234时未考虑字节序问题，而直接以0x12、0x34的顺序发送。结果接收端以小端序方式保存数据，因此小端序接收的数据变成0x3412，而非0x1234。正因如此，在通过网络传输数据时约定统一方式，这种约定称为网络字节序（Network Byte Order），非常简单——统一为大端序。

即，先把数据数组转化成大端序格式再进行网络传输。因此，所有计算机接收数据时应识别

该数据是网络字节序格式，小端序系统传输数据时应转化为大端序排列方式。

字节序转换（Endian Conversions）

相信大家已经理解了为何要在填充sockadr_in结构体前将数据转换成网络字节序。接下来介绍帮助转换字节序的函数。

- unsigned short htons(unsigned short);
- unsigned short ntohs(unsigned short);
- unsigned long htonl(unsigned long);
- unsined long ntohl(unsigned long);

通过函数名应该能掌握其功能，只需了解以下细节。

- htons中的h代表主机（host）字节序。
- htons中的n代表网络（network）字节序。

另外，s指的是short，l指的是long（Linux中long类型占用4个字节，这很关键）。因此，htons是h、to、n、s的组合，也可以解释为“把short型数据从主机字节序转化为网络字节序”。

再举个例子，ntohs可以解释为“把short型数据从网络字节序转化为主机字节序”。

通常，以s作为后缀的函数中，s代表2个字节short，因此用于端口号转换；以l作为后缀的函数中，l代表4个字节，因此用于IP地址转换。另外，有些读者可能有如下疑问：

> “我的系统是大端序的，为sockaddr_in结构体变量赋值前就不需要转换字节序了吧？”

这么说也不能算错。但我认为，有必要编写与大端序无关的统一代码。这样，即使在大端序系统中，最好也经过主机字节序转换为网络字节序的过程。当然，此时主机字节序与网络字节序相同，不会有任何变化。下面通过示例说明以上函数的调用过程。

❖ endian_conv.c

```
1.  #include <stdio.h>
2.  #include <arpa/inet.h>
3.
4.  int main(int argc, char *argv[])
5.  {
6.      unsigned short host_port=0x1234;
7.      unsigned short net_port;
8.      unsigned long host_addr=0x12345678;
9.      unsigned long net_addr;
10.
11.     net_port=htons(host_port);
12.     net_addr=htonl(host_addr);
```

```
13.
14.     printf("Host ordered port: %#x \n", host_port);
15.     printf("Network ordered port: %#x \n", net_port);
16.     printf("Host ordered address: %#lx \n", host_addr);
17.     printf("Network ordered address: %#lx \n", net_addr);
18.     return 0;
19. }
```

- 第6、8行：各保存2个字节、4个字节的数据。当然，若运行程序的CPU不同，则保存的字节序也不同。
- 第11、12行：变量host_port和host_addr中的数据转化为网络字节序。若运行环境为小端序CPU，则按改变之后的字节序保存。

❖ 运行结果：endian_conv.c

```
root@my_linux:/tcpip# gcc endian_conv.c -o conv
root@my_linux:/tcpip# ./conv
Host ordered port: 0x1234
Network ordered port: 0x3412
Host ordered address: 0x12345678
Network ordered address: 0x78563412
```

这就是在小端序CPU中运行的结果。如果在大端序CPU中运行，则变量值不会改变。大部分朋友都会得到类似的运行结果，因为Intel和AMD系列的CPU都采用小端序标准。

知识补给站 数据在传输之前都要经过转换吗？

也许有读者认为："既然数据传输采用网络字节序，那在传输前应直接把数据转换成网络字节序，接收的数据也需要转换成主机字节序再保存。"如果数据收发过程中没有自动转换机制，那当然需要程序员手动转换。这光想想就让人觉得可怕，难道真要强求程序员做这些事情吗？实际上没必要，这个过程是自动的。除了向sockaddr_in结构体变量填充数据外，其他情况无需考虑字节序问题。

3.4 网络地址的初始化与分配

前面已讨论过网络字节序，接下来介绍以bind函数为代表的结构体的应用。

将字符串信息转换为网络字节序的整数型

sockaddr_in中保存地址信息的成员为32位整数型。因此，为了分配IP地址，需要将其表示为

32位整数型数据。这对于只熟悉字符串信息的我们来说实非易事。各位可以尝试将IP地址201.211.214.36转换为4字节整数型数据。

对于IP地址的表示，我们熟悉的是点分十进制表示法（Dotted Decimal Notation），而非整数型数据表示法。幸运的是，有个函数会帮我们将字符串形式的IP地址转换成32位整数型数据。此函数在转换类型的同时进行网络字节序转换。

```
#include <arpa/inet.h>

in_addr_t  inet_addr(const  char * string);
```

➔ 成功时返回32位大端序整数型值，失败时返回INADDR_NONE。

如果向该函数传递类似“211.214.107.99”的点分十进制格式的字符串，它会将其转换为32位整数型数据并返回。当然，该整数型值满足网络字节序。另外，该函数的返回值类型in_addr_t在内部声明为32位整数型。下列示例表示该函数的调用过程。

❖ inet_addr.c

```
1.  #include <stdio.h>
2.  #include <arpa/inet.h>
3.
4.  int main(int argc, char *argv[])
5.  {
6.      char *addr1="1.2.3.4";
7.      char *addr2="1.2.3.256";
8.
9.      unsigned long conv_addr=inet_addr(addr1);
10.     if(conv_addr==INADDR_NONE)
11.         printf("Error occured! \n");
12.     else
13.         printf("Network ordered integer addr: %#lx \n", conv_addr);
14.
15.     conv_addr=inet_addr(addr2);
16.     if(conv_addr==INADDR_NONE)
17.         printf("Error occureded \n");
18.     else
19.         printf("Network ordered integer addr: %#lx \n\n", conv_addr);
20.     return 0;
21. }
```

- 第7行：1个字节能表示的最大整数为255，也就是说，它是错误的IP地址。利用该错误地址验证inet_addr函数的错误检测能力。
- 第9、15行：通过运行结果验证第9行的函数正常调用，而第15行的函数调用出现异常。

❖ 运行结果：inet_addr.c

```
root@my_linux:/tcpip# gcc inet_addr.c -o addr
root@my_linux:/tcpip# ./addr
Network ordered integer addr: 0x4030201
Error occureded
```

从运行结果可以看出，inet_addr函数不仅可以把IP地址转成32位整数型，而且可以检测无效的IP地址。另外，从输出结果可以验证确实转换为网络字节序。

inet_aton函数与inet_addr函数在功能上完全相同，也将字符串形式IP地址转换为32位网络字节序整数并返回。只不过该函数利用了in_addr结构体，且其使用频率更高。

```
#include <arpa/inet.h>

int  inet_aton(const  char * string, struct in_addr * addr);
```

➔ 成功时返回1（true），失败时返回0（false）。

- string　含有需转换的IP地址信息的字符串地址值。
- addr　将保存转换结果的in_addr结构体变量的地址值。

实际编程中若要调用inet_addr函数，需将转换后的IP地址信息代入sockaddr_in结构体中声明的in_addr结构体变量。而inet_aton函数则不需此过程。原因在于，若传递in_addr结构体变量地址值，函数会自动把结果填入该结构体变量。通过示例了解inet_aton函数调用过程。

❖ inet_aton.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <arpa/inet.h>
4.  void error_handling(char *message);
5.
6.  int main(int argc, char *argv[])
7.  {
8.      char *addr="127.232.124.79";
9.      struct sockaddr_in addr_inet;
10.
11.     if(!inet_aton(addr, &addr_inet.sin_addr))
12.         error_handling("Conversion error");
13.     else
14.         printf("Network ordered integer addr: %#x \n",
15.             addr_inet.sin_addr.s_addr);
16.     return 0;
17. }
18.
```

```
19. void error_handling(char *message)
20. {
21.     fputs(message, stderr);
22.     fputc('\n', stderr);
23.     exit(1);
24. }
```

- 第9、11行：转换后的IP地址信息需保存到sockaddr_in的in_addr型变量才有意义。因此，inet_aton函数的第二个参数要求得到in_addr型的变量地址值。这就省去了手动保存IP地址信息的过程。

❖ 运行结果：inet_aton.c

```
root@my_linux:/tcpip# gcc inet_aton.c -o aton
root@my_linux:/tcpip# ./aton
Network ordered integer addr: 0x4f7ce87f
```

上述运行结果无关紧要，更重要的是大家要熟练掌握该函数的调用方法。最后再介绍一个与inet_aton函数正好相反的函数，此函数可以把网络字节序整数型IP地址转换成我们熟悉的字符串形式。

```
#include <arpa/inet.h>

char * inet_ntoa(struct  in_addr  adr);

    ➔ 成功时返回转换的字符串地址值，失败时返回-1。
```

该函数将通过参数传入的整数型IP地址转换为字符串格式并返回。但调用时需小心，返回值类型为char指针。返回字符串地址意味着字符串已保存到内存空间，但该函数未向程序员要求分配内存，而是在内部申请了内存并保存了字符串。也就是说，调用完该函数后，应立即将字符串信息复制到其他内存空间。因为，若再次调用inet_ntoa函数，则有可能覆盖之前保存的字符串信息。总之，再次调用inet_ntoa函数前返回的字符串地址值是有效的。若需要长期保存，则应将字符串复制到其他内存空间。下面给出该函数调用示例。

❖ inet_ntoa.c

```
1.  #include <stdio.h>
2.  #include <string.h>
3.  #include <arpa/inet.h>
4.
5.  int main(int argc, char *argv[])
6.  {
```

```
7.      struct sockaddr_in addr1, addr2;
8.      char *str_ptr;
9.      char str_arr[20];
10.
11.     addr1.sin_addr.s_addr=htonl(0x1020304);
12.     addr2.sin_addr.s_addr=htonl(0x1010101);
13.
14.     str_ptr=inet_ntoa(addr1.sin_addr);
15.     strcpy(str_arr, str_ptr);
16.     printf("Dotted-Decimal notation1: %s \n", str_ptr);
17.
18.     inet_ntoa(addr2.sin_addr);
19.     printf("Dotted-Decimal notation2: %s \n", str_ptr);
20.     printf("Dotted-Decimal notation3: %s \n", str_arr);
21.     return 0;
22. }
```

代码说明

- 第14行：向inet_ntoa函数传递结构体变量addr1中的IP地址信息并调用该函数，返回字符串形式的IP地址。
- 第15行：浏览并复制第14行中返回的IP地址信息。
- 第18、19行：再次调用inet_ntoa函数。由此得出，第14行中返回的地址已覆盖了新的IP地址字符串，可通过第19行的输出结果进行验证。
- 第20行：第15行中复制了字符串，因此可以正确输出第14行中返回的IP地址字符串。

❖ 运行结果：inet_ntoa.c

```
root@my_linux:/tcpip# gcc inet_ntoa.c -o ntoa
root@my_linux:/tcpip# ./ntoa
Dotted-Decimal notation1: 1.2.3.4
Dotted-Decimal notation2: 1.1.1.1
Dotted-Decimal notation3: 1.2.3.4
```

网络地址初始化

结合前面所学的内容，现在介绍套接字创建过程中常见的网络地址信息初始化方法。

```
struct  sockaddr_in  addr;
char * serv_ip = "211.217.168.13";      //声明 IP 地址字符串
char  * serv_port = "9190";             //声明端口号字符串
memset(&addr,  0,  sizeof(addr));       //结构体变量 addr 的所有成员初始化为 0
addr.sin_family = AF_INET;                          //指定地址族
addr.sin_addr.s_addr = inet_addr(serv_ip);      //基于字符串的 IP 地址初始化
addr.sin_port = htons(atoi(serv_port));         //基于字符串的端口号初始化
```

上述代码中，memset函数将每个字节初始化为同一值：第一个参数为结构体变量addr的地址值，即初始化对象为addr；第二个参数为0，因此初始化为0；最后一个参数中传入addr的长度，

因此addr的所有字节均初始化为0。这么做是为了将sockaddr_in结构体的成员sin_zero初始化为0。另外，最后一行代码调用的atoi函数把字符串类型的值转换成整数型。总之，上述代码利用字符串格式的IP地址和端口号初始化了sockaddr_in结构体变量。

另外，代码中对IP地址和端口号进行了硬编码，这并非良策，因为运行环境改变就得更改代码。因此，我们运行示例main函数时传入IP地址和端口号。

客户端地址信息初始化

上述网络地址信息初始化过程主要针对服务器端而非客户端。给套接字分配IP地址和端口号主要是为下面这件事做准备：

“请把进入IP 211.217.168.13、9190端口的数据传给我！”

反观客户端中连接请求如下：

“请连接到IP 211.217.168.13、9190端口！”

请求方法不同意味着调用的函数也不同。服务器端的准备工作通过bind函数完成，而客户端则通过connect函数完成。因此，函数调用前需准备的地址值类型也不同。服务器端声明sockaddr_in结构体变量，将其初始化为赋予服务器端IP和套接字的端口号，然后调用bind函数；而客户端则声明sockaddr_in结构体，并初始化为要与之连接的服务器端套接字的IP和端口号，然后调用connect函数。

INADDR_ANY

每次创建服务器端套接字都要输入IP地址会有些繁琐，此时可如下初始化地址信息。

```
struct  sockaddr_in  addr;
char  *  serv_port = “9190”;
memset(&addr,  0,  sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = htonl(INADDR_ANY);
addr.sin_port = htons(atoi(serv_port));
```

与之前方式最大的区别在于，利用常数INADDR_ANY分配服务器端的IP地址。若采用这种方式，则可自动获取运行服务器端的计算机IP地址，不必亲自输入。而且，若同一计算机中已分配多个IP地址（多宿主（Multi-homed）计算机，一般路由器属于这一类），则只要端口号一致，就可以从不同IP地址接收数据。因此，服务器端中优先考虑这种方式。而客户端中除非带有一部分服务器端功能，否则不会采用。

知识补给站 创建服务器端套接字时需要IP地址的原因

初始化服务器端套接字时应分配所属计算机的IP地址，因为初始化时使用的IP地址非常明确，那为何还要进行IP初始化呢？如前所述，同一计算机中可以分配多个IP地址，实际IP地址的个数与计算机中安装的NIC的数量相等。即使是服务器端套接字，也需要决定应接收哪个IP传来的（哪个NIC传来的）数据。因此，服务器端套接字初始化过程中要求IP地址信息。另外，若只有1个NIC，则直接使用INADDR_ANY。

第 1 章的 hello_server.c、hello_client.c 运行过程

第1章中执行以下命令以运行相当于服务器端的hello_server.c。

```
./hserver 9190
```

通过代码可知，向main函数传递的9190为端口号。通过此端口创建服务器端套接字并运行程序，但未传递IP地址，因为可以通过INADDR_ANY指定IP地址。相信各位现在再去读代码会感觉简单很多。

执行下列命令以运行相当于客户端的hello_client.c。与服务器端运行方式相比，最大的区别是传递了IP地址信息。

```
./hclient 127.0.0.1  9190
```

127.0.0.1是回送地址（loopback address），指的是计算机自身IP地址。在第1章的示例中，服务器端和客户端在同一计算机中运行，因此，连接目标服务器端的地址为127.0.0.1。当然，若用实际IP地址代替此地址也能正常运转。如果服务器端和客户端分别在2台计算机中运行，则可以输入服务器端IP地址。

向套接字分配网络地址

既然已讨论了sockaddr_in结构体的初始化方法，接下来就把初始化的地址信息分配给套接字。bind函数负责这项操作。

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

int bind(int sockfd,  struct sockaddr * myaddr,  socklen_t addrlen);
```

➔ 成功时返回 0，失败时返回-1。

- sockfd　要分配地址信息（IP地址和端口号）的套接字文件描述符。
- myaddr　存有地址信息的结构体变量地址值。
- addrlen　第二个结构体变量的长度。

如果此函数调用成功，则将第二个参数指定的地址信息分配给第一个参数中的相应套接字。下面给出服务器端常见套接字初始化过程。

```
int serv_sock;
struct sockaddr_in serv_addr;
char * serv_port = "9190";

/* 创建服务器端套接字（监听套接字） */
serv_sock = socket(PF_INET, SOCK_STREAM, 0);

/* 地址信息初始化 */
memset(&serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port = htons(atoi(serv_port));

/* 分配地址信息 */
bind(serv_sock, (struct sockaddr * )&serv_addr, sizeof(serv_addr));
......
```

服务器端代码结构默认如上，当然还有未显示的异常处理代码。

3.5　基于 Windows 的实现

Windows中同样存在sockaddr_in结构体及各种变换函数，而且名称、使用方法及含义都相同。也就无需针对Windows平台进行太多修改或改用其他函数。接下来将前面几个程序改成Windows版本。

函数 htons、htonl 在 Windows 中的使用

首先给出Windows平台下调用htons函数和htonl函数的示例。这两个函数的用法与Linux平台下的使用并无区别，故省略。

❖ endian_conv_win.c

```
1.  #include <stdio.h>
2.  #include <winsock2.h>
```

```
3.  void ErrorHandling(char* message);
4.
5.  int main(int argc, char *argv[])
6.  {
7.      WSADATA wsaData;
8.      unsigned short host_port=0x1234;
9.      unsigned short net_port;
10.     unsigned long host_addr=0x12345678;
11.     unsigned long net_addr;
12.
13.     if(WSAStartup(MAKEWORD(2, 2), &wsaData)!=0)
14.         ErrorHandling("WSAStartup() error!");
15.
16.     net_port=htons(host_port);
17.     net_addr=htonl(host_addr);
18.
19.     printf("Host ordered port: %#x \n", host_port);
20.     printf("Network ordered port: %#x \n", net_port);
21.     printf("Host ordered address: %#lx \n", host_addr);
22.     printf("Network ordered address: %#lx \n", net_addr);
23.     WSACleanup();
24.     return 0;
25. }
26.
27. void ErrorHandling(char* message)
28. {
29.     fputs(message, stderr);
30.     fputc('\n', stderr);
31.     exit(1);
32. }
```

❖ 运行结果：endian_conv_win.c

```
Host ordered port: 0x1234
Network ordered port: 0x3412
Host ordered address: 0x12345678
Network ordered address: 0x78563412
```

该程序多了进行库初始化的WSAStartup函数调用和winsock2.h头文件的#include语句，其他部分没有区别。

函数 inet_addr、inet_ntoa 在 Windows 中的使用

下列示例给出了inet_addr函数和inet_ntoa函数的调用过程。前面分别给出了Linux中这两个函数的调用示例，而在Windows中则通过1个示例介绍。另外，Windows中不存在inet_aton函数，故省略。

❖ inet_adrconv_win.c

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

int main(int argc, char *argv[])
{
    WSADATA wsaData;
    if(WSAStartup(MAKEWORD(2, 2), &wsaData)!=0)
        ErrorHandling("WSAStartup() error!");

    /* inet_addr函数调用示例*/
    {
        char *addr="127.212.124.78";
        unsigned long conv_addr=inet_addr(addr);
        if(conv_addr==INADDR_NONE)
            printf("Error occured! \n");
        else
            printf("Network ordered integer addr: %#lx \n", conv_addr);
    }

    /* inet_ntoa函数调用示例*/
    {
        struct sockaddr_in addr;
        char *strPtr;
        char strArr[20];

        addr.sin_addr.s_addr=htonl(0x1020304);
        strPtr=inet_ntoa(addr.sin_addr);
        strcpy(strArr, strPtr);
        printf("Dotted-Decimal notation3 %s \n", strArr);
    }

    WSACleanup();
    return 0;
}

void ErrorHandling(char* message)
{
    //与之前示例一致，故省略!
}
```

❖ 运行结果：inet_adrconv_win.c

```
Network ordered integer addr: 0x4e7cd47f
Dotted-Decimal notation3 1.2.3.4
```

上述示例在main函数体内使用中括号增加变量声明，同时区分各函数的调用过程。添加中括号可以在相应区域的初始部分声明局部变量。当然，此类局部变量跳出中括号则消失。

在 Windows 环境下向套接字分配网络地址

Windows中向套接字分配网络地址的过程与Linux中完全相同，因为bind函数的含义、参数及返回类型完全一致。

```
SOCKET servSock;
struct sockaddr_in servAddr;
char * servPort = "9190";

/* 创建服务器端套接字 */
servSock = socket(PF_INET, SOCK_STREAM, 0);

/* 地址信息初始化 */
memset(&servAddr, 0, sizeof(servAddr));
servAddr.sin_family = AF_INET;
servAddr.sin_addr.s_addr = htonl(INADDR_ANY);
servAddr.sin_port = htons(atoi(serv_port));

/* 分配地址信息 */
bind(servSock, (struct sockaddr * )&servAddr, sizeof(servAddr));
......
```

这与Linux平台下套接字初始化及地址分配过程基本一致，只不过改了一些变量名。

WSAStringToAddress & WSAAddressToString

下面介绍Winsock2中增加的2个转换函数。它们在功能上与inet_ntoa和inet_addr完全相同，但优点在于支持多种协议，在IPv4和IPv6中均可适用。当然它们也有缺点，使用inet_ntoa、inet_addr可以很容易地在Linux和Windows之间切换程序。而将要介绍的这2个函数则依赖于特定平台，会降低兼容性。因此本书不会使用它们，介绍的目的仅在于让各位了解更多函数。

先介绍WSAStringToAddress函数，它将地址信息字符串适当填入结构体变量。

```
#include <winsock2.h>

INT WSAStringToAddress(
    LPTSTR AddressString, INT AddressFamily, LPWSAPROTOCOL_INFO lpProtocolInfo,
    LPSOCKADDR lpAddress, LPINT lpAddressLength
);
```

➔ 成功时返回 0，失败时返回 SOCKET_ERROR。

- AddressString　含有IP和端口号的字符串地址值。
- AddressFamily　第一个参数中地址所属的地址族信息。
- lpProtocolInfo　设置协议提供者（Provider），默认为NULL。
- lpAddress　保存地址信息的结构体变量地址值。
- lpAddressLength　第四个参数中传递的结构体长度所在的变量地址值。

上述函数中新出现的各种类型几乎都是针对默认数据类型的typedef声明。下列示例主要通过默认数据类型向该函数传递参数。

WSAAddressToString与WSAStringToAddress在功能上正好相反，它将结构体中的地址信息转换成字符串形式。

```
#include <winsock2.h>

INT WSAAddressToString(
    LPSOCKADDR lpsaAddress, DWORD dwAddressLength,
    LPWSAPROTOCOL_INFO lpProtocolInfo, LPSTR lpszAddressString,
    LPDWORD lpdwAddressStringLength
);
```

➔ 成功时返回0，失败时返回SOCKET_ERROR。

- lpsaAddress　需要转换的地址信息结构体变量地址值。
- dwAddressLength　第一个参数中结构体的长度。
- lpProtocolInfo　设置协议提供者（Provider），默认为NULL。
- lpszAddressString　保存转换结果的字符串地址值。
- lpdwAddressStringLength　第四个参数中存有地址信息的字符串长度。

下面给出这两个函数的使用示例。

❖ conv_addr_win.c

```
1.  #undef UNICODE
2.  #undef _UNICODE
3.  #include <stdio.h>
4.  #include <winsock2.h>
5.
6.  int main(int argc, char *argv[])
7.  {
8.      char *strAddr="203.211.218.102:9190";
9.
10.     char strAddrBuf[50];
11.     SOCKADDR_IN servAddr;
12.     int size;
13.
```

```
14.     WSADATA wsaData;
15.     WSAStartup(MAKEWORD(2, 2), &wsaData);
16.
17.     size=sizeof(servAddr);
18.     WSAStringToAddress(
19.         strAddr, AF_INET, NULL, (SOCKADDR*)&servAddr, &size);
20.
21.     size=sizeof(strAddrBuf);
22.     WSAAddressToString(
23.         (SOCKADDR*)&servAddr, sizeof(servAddr), NULL, strAddrBuf, &size);
24.
25.     printf("Second conv result: %s \n", strAddrBuf);
26.     WSACleanup();
27.     return 0;
28. }
```

代码说明

- 第1、2行：#undef用于取消之前定义的宏。根据项目环境，VC++会自主声明这2个宏，这样在第18行和第22行调用的函数中，参数就将转换成unicode形式，给出错误的运行结果。所以插入了这2句宏定义。
- 第18行：第8行给出了需转换的字符串格式的地址。第18行调用WSAStringToAddress函数转换成结构体，保存到第11行声明的变量。
- 第22行：第18行代码的逆过程，调用WSAAddressToString函数将结构体转换成字符串。

❖ 运行结果：conv_addr_win.c

```
Second conv result: 203.211.218.102:9190
```

上述示例的主要目的在于展示WSAStringToAddress函数与WSAAddressToString函数的使用方法。Linux环境下地址初始化过程中声明了sockaddr_in变量，而示例则声明了SOCKADDR_IN类型的变量。各位不必感到疑惑，实际上二者完全相同，只是为简化变量定义添加了typedef声明。

```
typedef struct sockaddr_in SOCKADDR_IN;
```

套接字地址分配相关内容讲解到此结束。

3.6 习题

(1) IP地址族IPv4和IPv6有何区别？在何种背景下诞生了IPv6？

(2) 通过IPv4网络ID、主机ID及路由器的关系说明向公司局域网中的计算机传输数据的过程。

(3) 套接字地址分为IP地址和端口号。为什么需要IP地址和端口号？或者说，通过IP可以区分哪些对象？通过端口号可以区分哪些对象？

(4) 请说明IP地址的分类方法，并据此说出下面这些IP地址的分类。

- 214.121.212.102　　(　　)
- 120.101.122.89　　(　　)
- 129.78.102.211　　(　　)

(5) 计算机通过路由器或交换机连接到互联网。请说出路由器和交换机的作用。

(6) 什么是知名端口？其范围是多少？知名端口中具有代表性的HTTP和FTP端口号各是多少？

(7) 向套接字分配地址的bind函数原型如下：

```
int bind(int sockfd, struct sockaddr *myaddr, socklen_t addrlen);
```

而调用时则用

```
bind(serv_sock, (struct sockaddr *) &serv_addr, sizeof(serv_addr));
```

此处serv_addr为sockaddr_in结构体变量。与函数原型不同，传入的是sockaddr_in结构体变量，请说明原因。

(8) 请解释大端序、小端序、网络字节序，并说明为何需要网络字节序。

(9) 大端序计算机希望把4字节整数型数据12传递到小端序计算机。请说出数据传输过程中发生的字节序变换过程。

(10) 怎样表示回送地址？其含义是什么？如果向回送地址传输数据将发生什么情况？

第4章

基于TCP的服务器端/客户端（1）

4

我们已经学习了创建套接字和向套接字分配地址，接下来正式讨论通过套接字收发数据。

之前介绍套接字时举例说明了面向连接的套接字和面向消息的套接字这2种数据传输方式，特别是重点讨论了面向连接的套接字。本章将具体讨论这种面向连接的服务器端/客户端的编写。

4.1 理解 TCP 和 UDP

根据数据传输方式的不同，基于网络协议的套接字一般分为TCP套接字和UDP套接字。因为TCP套接字是面向连接的，因此又称基于流（stream）的套接字。

TCP是Transmission Control Protocol（传输控制协议）的简写，意为“对数据传输过程的控制”。因此，学习控制方法及范围有助于正确理解TCP套接字。

TCP/IP 协议栈

讲解TCP前先介绍TCP所属的TCP/IP协议栈（Stack，层），如图4-1所示。

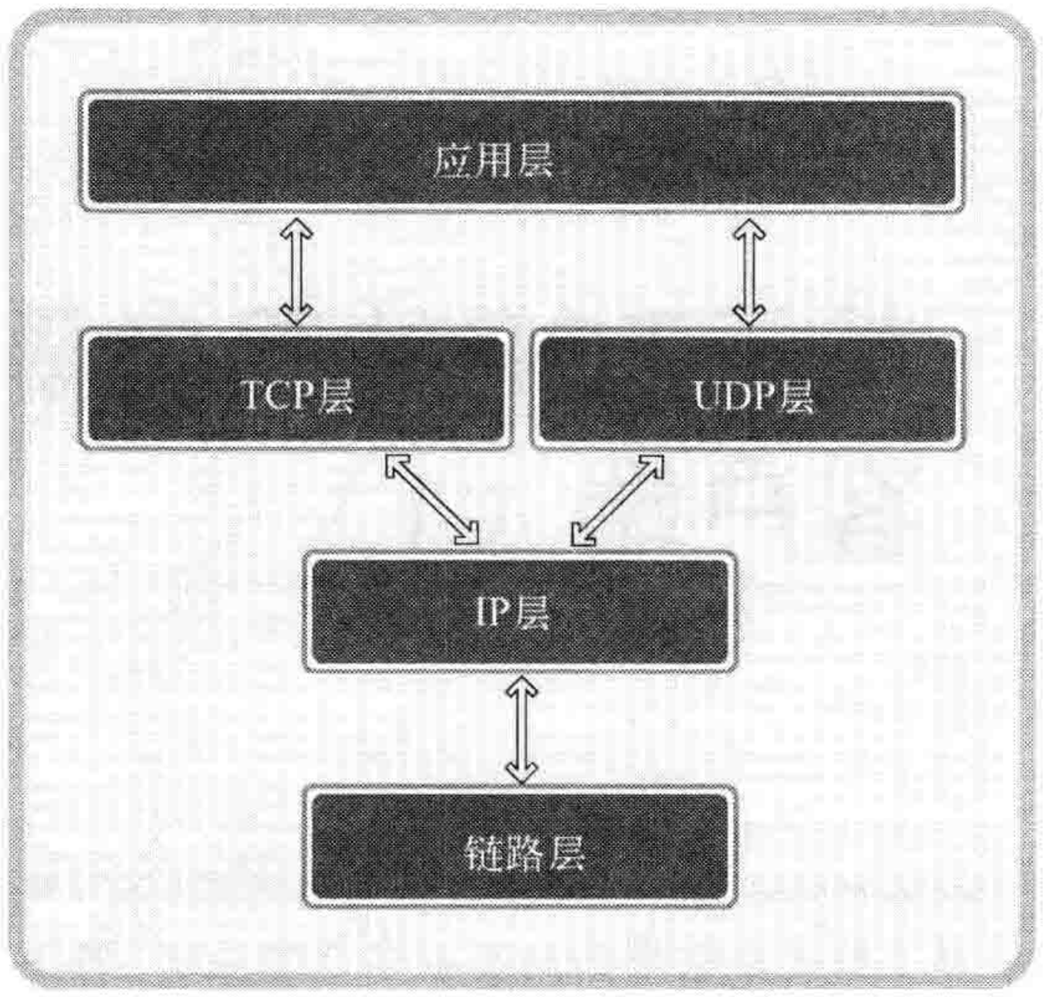


图4-1　TCP/IP协议栈

从图4-1可以看出，TCP/IP协议栈共分4层，可以理解为数据收发分成了4个层次化过程。也就是说，面对“基于互联网的有效数据传输”的命题，并非通过1个庞大协议解决问题，而是化整为零，通过层次化方案——TCP/IP协议栈解决。通过TCP套接字收发数据时需要借助这4层，如图4-2所示。

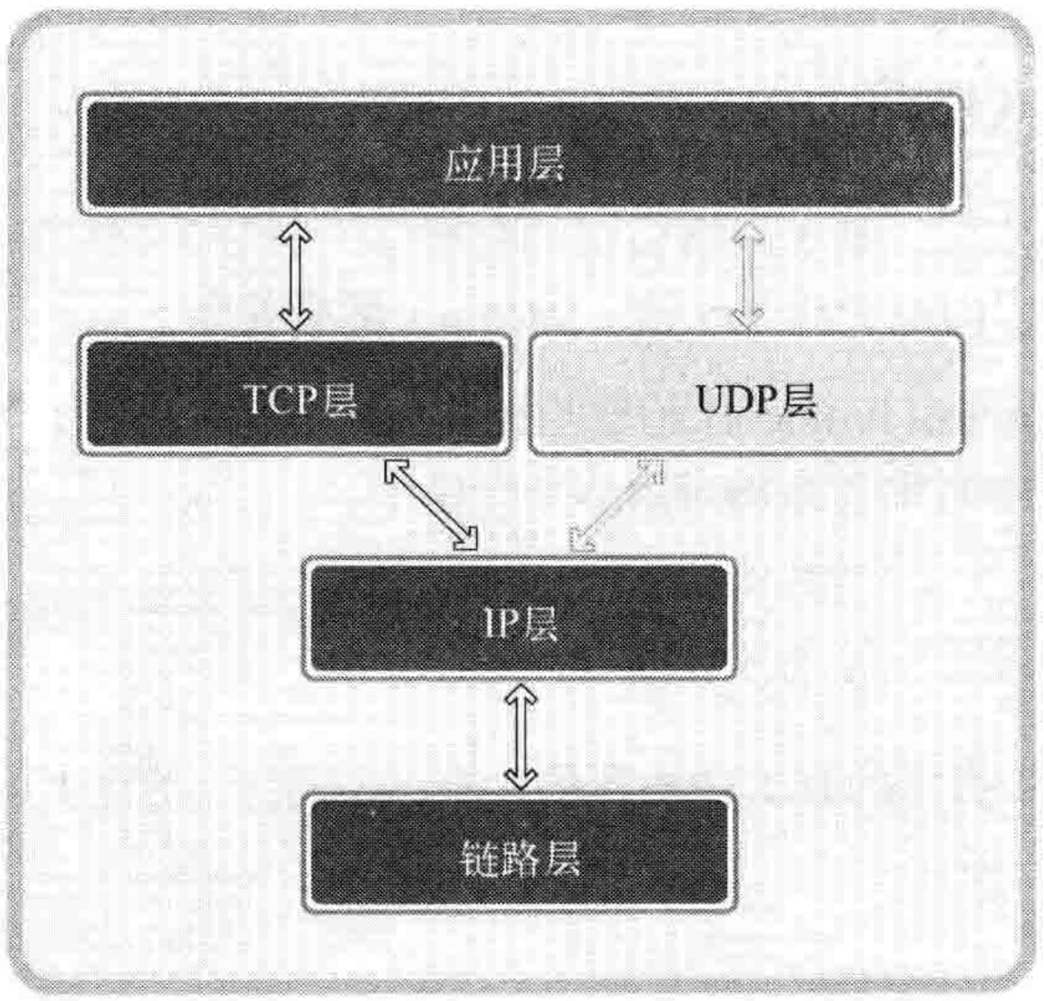


图4-2　TCP协议栈

反之，通过UDP套接字收发数据时，利用图4-3中的4层协议栈完成。

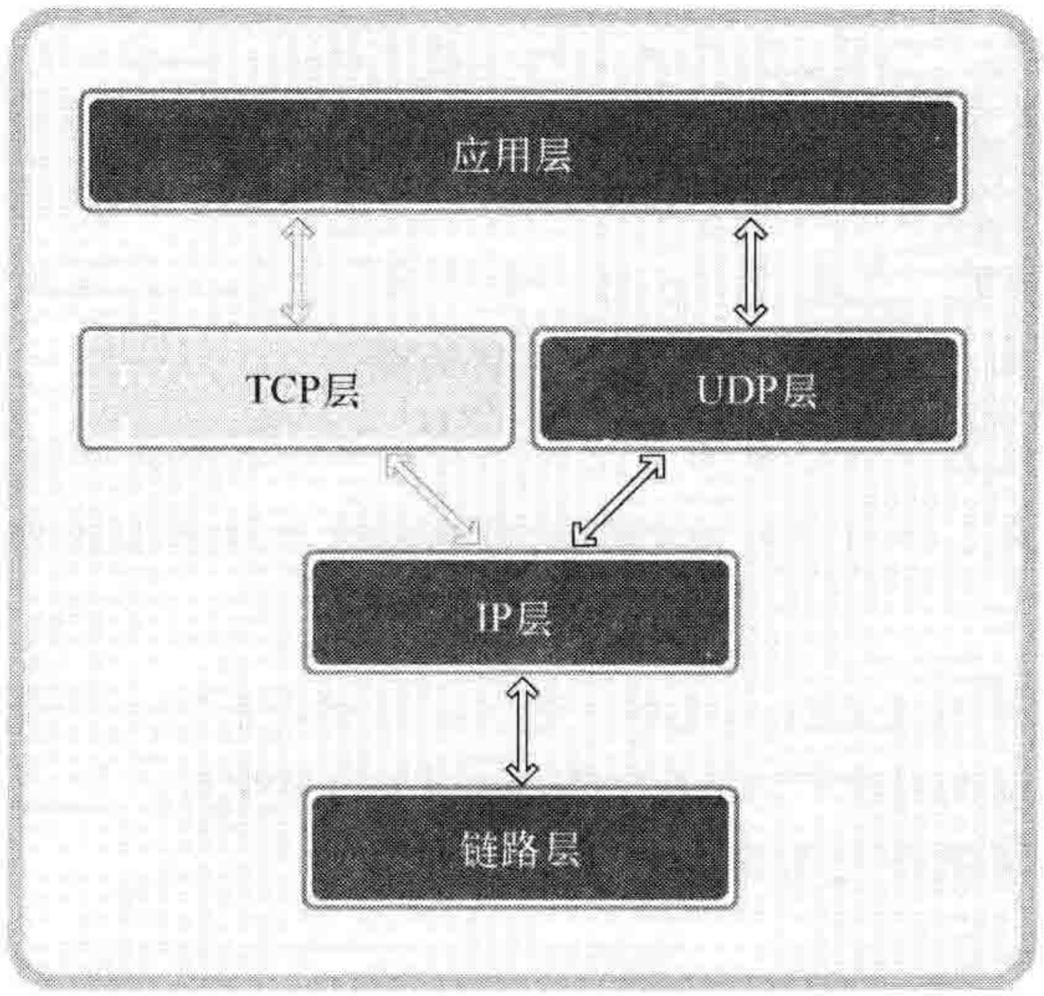


图4-3 UDP协议栈

各层可能通过操作系统等软件实现，也可能通过类似NIC的硬件设备实现。

提 示

OSI 7 Layer（层）

数据通信中使用的协议栈分为 7 层，而本书分了 4 层。想了解 7 层协议栈的细节可以参考数据通信相关书籍。对程序员来说，掌握 4 层协议栈就足够了。

TCP/IP 协议的诞生背景

“通过因特网完成有效数据传输”这个课题让许多专家聚集到了一起，这些人是硬件、系统、路由算法等各领域的顶级专家。为何需要这么多领域的专家呢?

我们之前只关注套接字创建及应用，却忽略了计算机网络问题并非仅凭软件就能解决。编写软件前需要构建硬件系统，在此基础上需要通过软件实现各种算法。所以才需要众多领域的专家进行讨论，以形成各种规定。因此，把这个大问题划分成若干小问题再逐个攻破，将大幅提高效率。

把“通过因特网完成有效数据传输”问题按照不同领域划分成小问题后，出现多种协议，它们通过层级结构建立了紧密联系。

知识补给站 开放式系统（Open System）

把协议分成多个层次具有哪些优点？协议设计更容易？当然这也足以成为优点之一。但还有更重要的原因就是，为了通过标准化操作设计开放式系统。

标准本身就在于对外公开，引导更多的人遵守规范。以多个标准为依据设计的系统称为开放式系统，我们现在学习的TCP/IP协议栈也属于其中之一。接下来了解一下开放式系统具有哪些优点。路由器用来完成IP层交互任务。某公司原来使用A公司的路由器，现要将其替换成B公司的，是否可行？这并非难事，并不一定要换成同一公司的同一型号路由器，因为所有生产商都会按照IP层标准制造。

再举个例子。各位的计算机是否装有网络接口卡，也就是所谓的网卡？尚未安装也无妨，其实很容易买到，因为所有网卡制造商都会遵守链路层的协议标准。这就是开放式系统的优点。

标准的存在意味着高速的技术发展，这也是开放式系统设计最大的原因所在。实际上，软件工程中的“面向对象”（Object Oriented）的诞生背景中也有标准化的影子。也就是说，标准对于技术发展起着举足轻重的作用。

链路层

接下来逐层了解TCP/IP协议栈，先讲解链路层。链路层是物理链接领域标准化的结果，也是最基本的领域，专门定义LAN、WAN、MAN等网络标准。若两台主机通过网络进行数据交换，则需要图4-4所示的物理连接，链路层就负责这些标准。

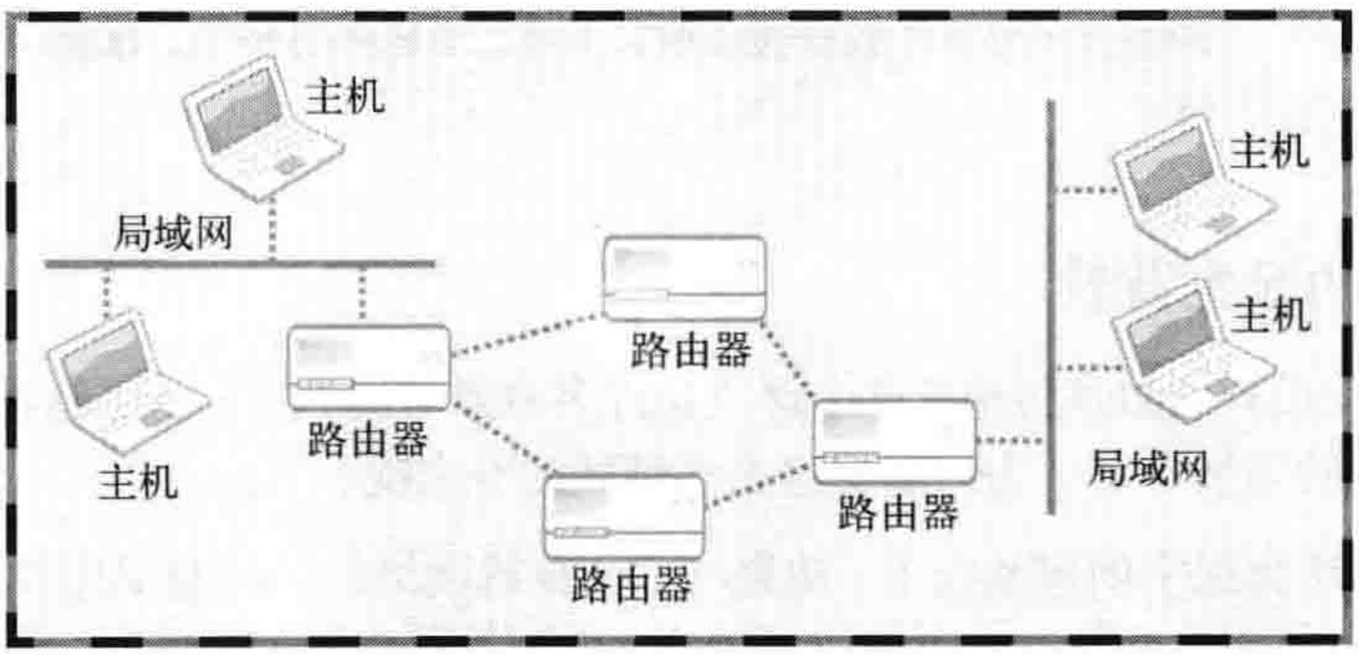


图4-4　网络连接结构

IP层

准备好物理连接后就要传输数据。为了在复杂的网络中传输数据，首先需要考虑路径的选择。向目标传输数据需要经过哪条路径？解决此问题就是IP层，该层使用的协议就是IP。

IP本身是面向消息的、不可靠的协议。每次传输数据时会帮我们选择路径，但并不一致。如果传输中发生路径错误，则选择其他路径；但如果发生数据丢失或错误，则无法解决。换言之，

IP协议无法应对数据错误。

TCP/UDP 层

IP层解决数据传输中的路径选择问题，只需照此路径传输数据即可。TCP和UDP层以IP层提供的路径信息为基础完成实际的数据传输，故该层又称传输层（Transport）。UDP比TCP简单，我们将在后续章节展开讨论，现只解释TCP。TCP可以保证可靠的数据传输，但它发送数据时以IP层为基础（这也是协议栈结构层次化的原因）。那该如何理解二者关系呢?

IP层只关注1个数据包（数据传输的基本单位）的传输过程。因此，即使传输多个数据包，每个数据包也是由IP层实际传输的，也就是说传输顺序及传输本身是不可靠的。若只利用IP层传输数据，则有可能导致后传输的数据包B比先传输的数据包A提早到达。另外，传输的数据包A、B、C中有可能只收到A和C，甚至收到的C可能已损毁。反之，若添加TCP协议则按照如下对话方式进行数据交换。

- 主机A：“正确收到第二个数据包!”
- 主机B：“恩，知道了。”

- 主机A：“正确收到第三个数据包!”
- 主机B：“可我已发送第四个数据包了啊! 哦，您没收到第四个数据包吧? 我给您重传!”

这就是TCP的作用。如果数据交换过程中可以确认对方已收到数据，并重传丢失的数据，那么即便IP层不保证数据传输，这类通信也是可靠的，如图4-5所示。

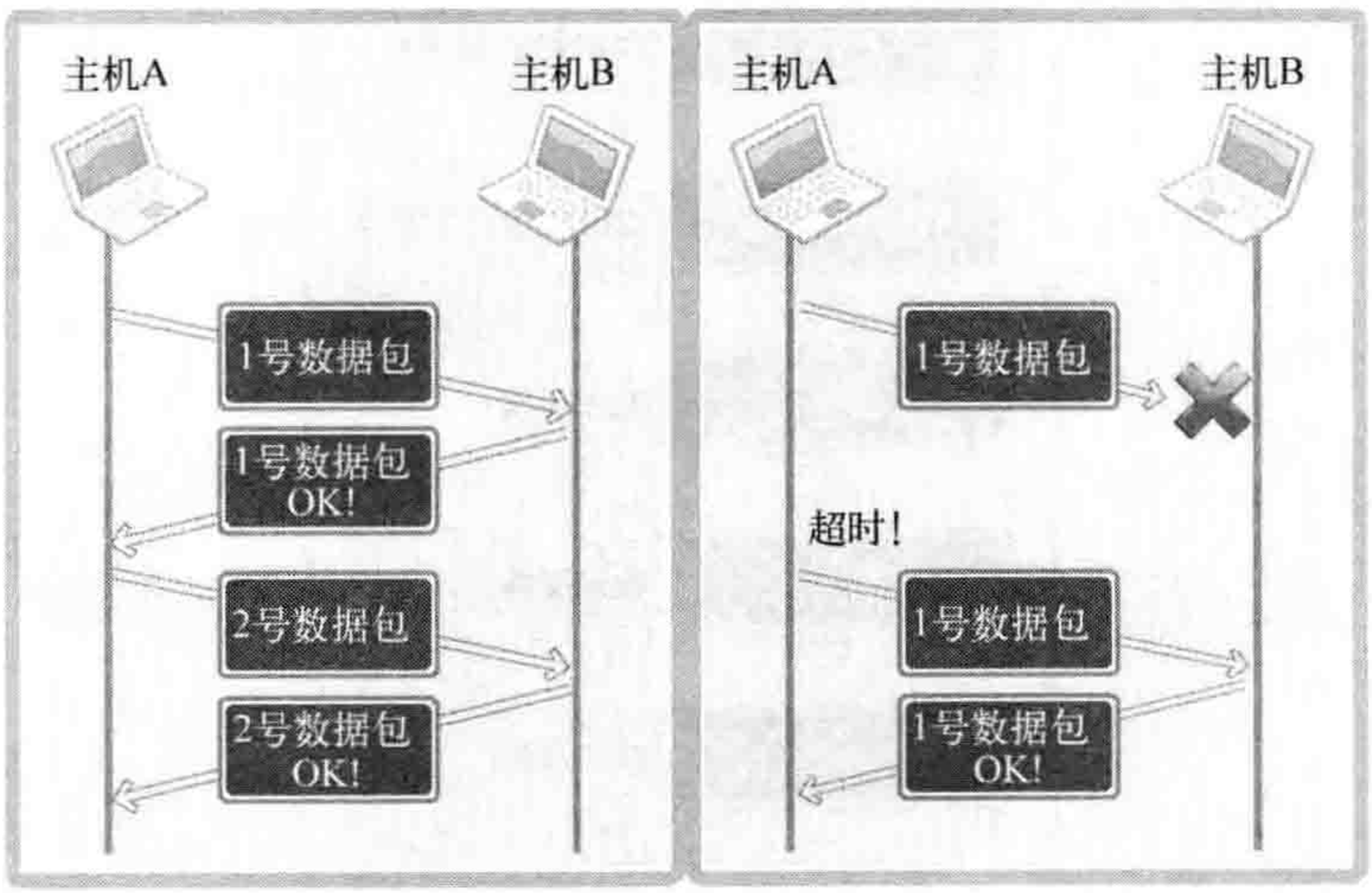


图4-5 传输控制协议

图4-5简单描述了TCP的功能。总之，TCP和UDP存在于IP层之上，决定主机之间的数据传输方式，TCP协议确认后向不可靠的IP协议赋予可靠性。

应用层

上述内容是套接字通信过程中自动处理的。选择数据传输路径、数据确认过程都被隐藏到套接字内部。而与其说是“隐藏”，倒不如“使程序员从这些细节中解放出来”的表达更为准确。程序员编程时无需考虑这些过程，但这并不意味着不用掌握这些知识。只有掌握了这些理论，才能编写出符合需求的网络程序。

总之，向各位提供的工具就是套接字，大家只需利用套接字编出程序即可。编写软件的过程中，需要根据程序特点决定服务器端和客户端之间的数据传输规则（规定），这便是应用层协议。网络编程的大部分内容就是设计并实现应用层协议。

4.2　实现基于TCP的服务器端/客户端

本节实现完整的TCP服务器端，在此过程中各位将理解套接字使用方法及数据传输方法。

TCP服务器端的默认函数调用顺序

图4-6给出了TCP服务器端默认的函数调用顺序，绝大部分TCP服务器端都按照该顺序调用。

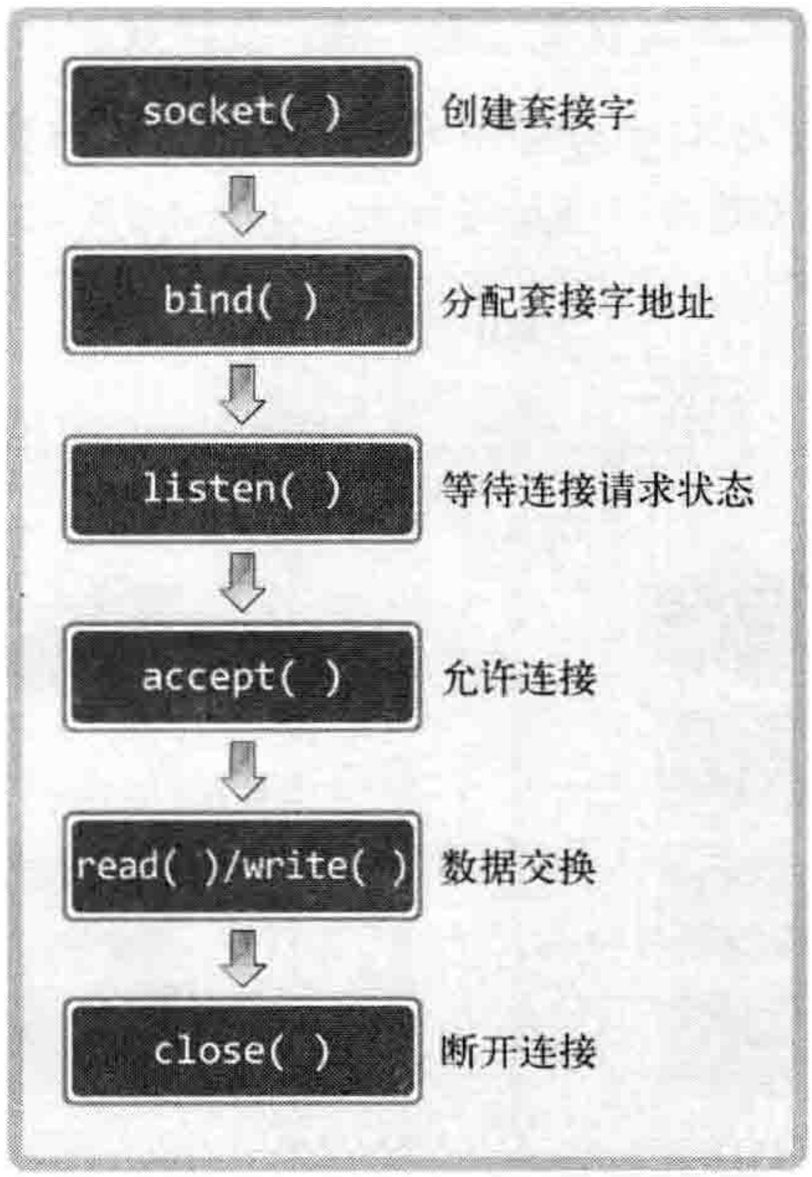


图4-6　TCP服务器端函数调用顺序

调用socket函数创建套接字，声明并初始化地址信息结构体变量，调用bind函数向套接字分配地址。这2个阶段之前都已讨论过，下面讲解之后的几个过程。

进入等待连接请求状态

我们已调用bind函数给套接字分配了地址，接下来就要通过调用listen函数进入等待连接请求状态。只有调用了listen函数，客户端才能进入可发出连接请求的状态。换言之，这时客户端才能调用connect函数（若提前调用将发生错误）。

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

int listen(int sock, int backlog);
```

➔ 成功时返回0，失败时返回-1。

- sock　希望进入等待连接请求状态的套接字文件描述符，传递的描述符套接字参数成为服务器端套接字（监听套接字）。
- backlog　连接请求等待队列（Queue）的长度，若为5，则队列长度为5，表示最多使5个连接请求进入队列。

先解释一下等待连接请求状态的含义和连接请求等待队列。“服务器端处于等待连接请求状态”是指，客户端请求连接时，受理连接前一直使请求处于等待状态。图4-7给出了这个过程。

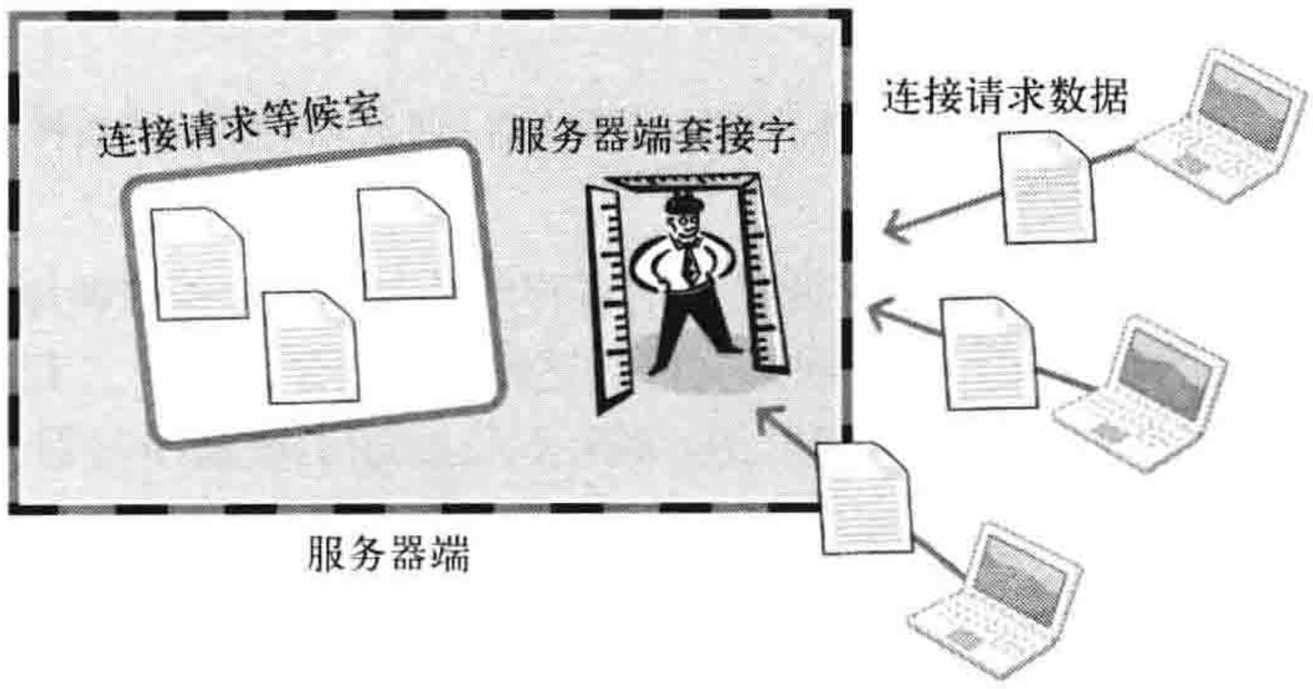


图4-7　等待连接请求状态

由图4-7可知作为listen函数的第一个参数传递的文件描述符套接字的用途。客户端连接请求本身也是从网络中接收到的一种数据，而要想接收就需要套接字。此任务就由服务器端套接字完成。服务器端套接字是接收连接请求的一名门卫或一扇门。

客户端如果向服务器端询问：“请问我是否可以发起连接？”服务器端套接字就会亲切应答：“您好！当然可以，但系统正忙，请到等候室排号等待，准备好后会立即受理您的连接。”同时将连接请求请到等候室。调用listen函数即可生成这种门卫（服务器端套接字），listen函数的第二个参数决定了等候室的大小。等候室称为连接请求等待队列，准备好服务器端套接字和连接请求等待队列后，这种可接收连接请求的状态称为等待连接请求状态。

listen函数的第二个参数值与服务器端的特性有关，像频繁接收请求的Web服务器端至少应为15。另外，连接请求队列的大小始终根据实验结果而定。

受理客户端连接请求

调用listen函数后，若有新的连接请求，则应按序受理。受理请求意味着进入可接受数据的状态。也许各位已经猜到进入这种状态所需部件——当然是套接字！大家可能认为可以使用服务器端套接字，但服务器端套接字是做门卫的。如果在与客户端的数据交换中使用门卫，那谁来守门呢？因此需要另外一个套接字，但没必要亲自创建。下面这个函数将自动创建套接字，并连接到发起请求的客户端。

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

int accept(int sock, struct sockaddr * addr, socklen_t * addrlen);
```

➔ 成功时返回创建的套接字文件描述符，失败时返回-1。

- sock　服务器套接字的文件描述符。
- addr　保存发起连接请求的客户端地址信息的变量地址值，调用函数后向传递来的地址变量参数填充客户端地址信息。
- addrlen　第二个参数addr结构体的长度，但是存有长度的变量地址。函数调用完成后，该变量即被填入客户端地址长度。

accept函数受理连接请求等待队列中待处理的客户端连接请求。函数调用成功时，accept函数内部将产生用于数据I/O的套接字，并返回其文件描述符。需要强调的是，套接字是自动创建的，并自动与发起连接请求的客户端建立连接。图4-8展示了accept函数调用过程。

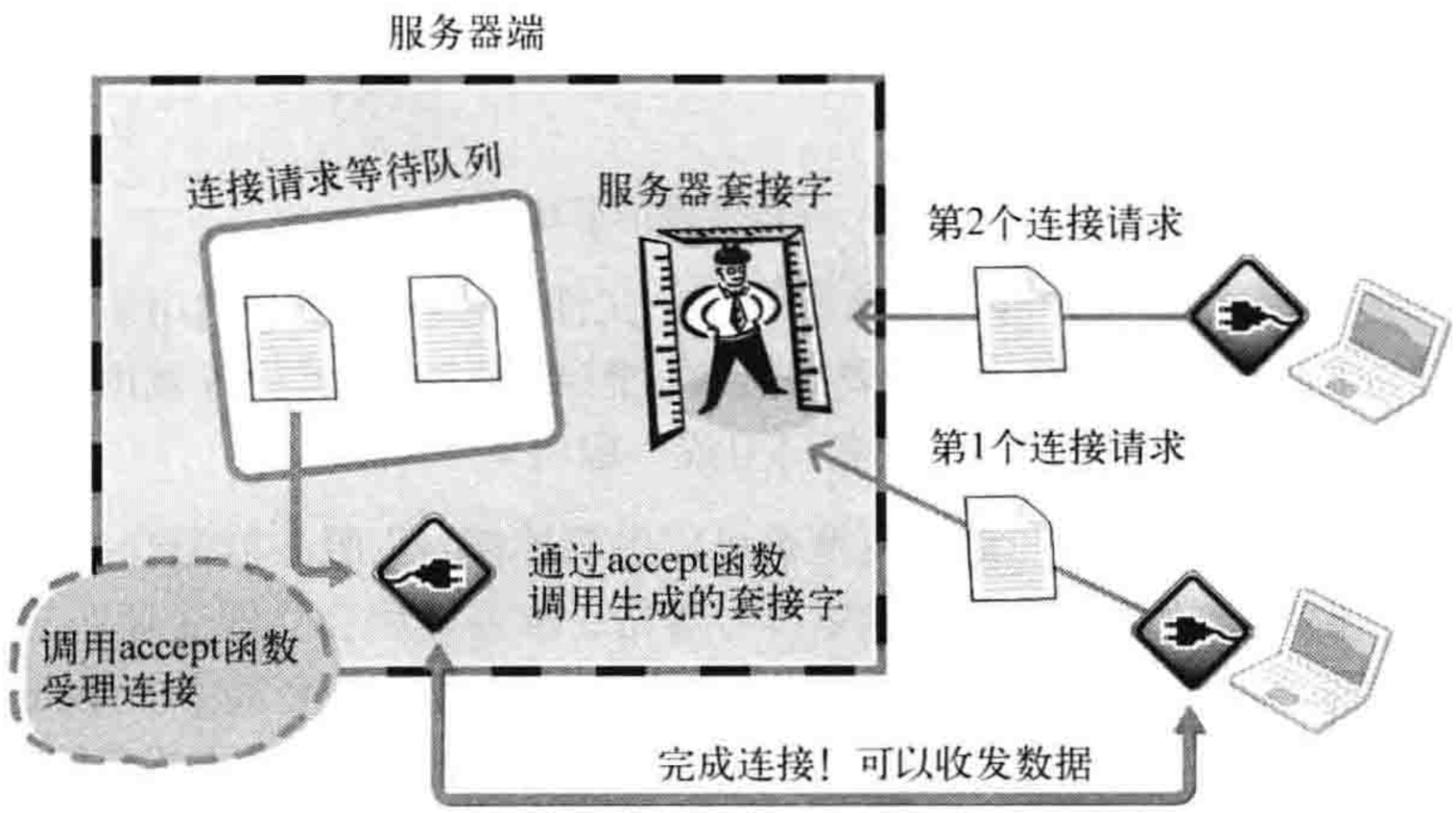


图4-8　受理连接请求状态

图4-8展示了“从等待队列中取出1个连接请求，创建套接字并完成连接请求”的过程。服务器端单独创建的套接字与客户端建立连接后进行数据交换。

回顾 Hello world 服务器端

前面结束了服务器端实现方法的所有讲解，下面分析之前未理解透的Hello world服务器端。第1章已给出其源代码，此处重列是为了便于讲解。

❖ hello_server.c


```
1.  /* 头文件及函数声明关系
2.  请参考第1章的源代码hello_server.c */
3.
4.  int main(int argc, char *argv[])
5.  {
6.      int serv_sock;
7.      int clnt_sock;
8.
9.      struct sockaddr_in serv_addr;
10.     struct sockaddr_in clnt_addr;
11.     socklen_t clnt_addr_size;
12.
13.     char message[]="Hello World!";
14.
15.     if(argc!=2)
16.     {
17.         printf("Usage : %s <port>\n", argv[0]);
18.         exit(1);
19.     }
20.
21.     serv_sock=socket(PF_INET, SOCK_STREAM, 0);
22.     if(serv_sock == -1)
23.         error_handling("socket() error");
24.
25.     memset(&serv_addr, 0, sizeof(serv_addr));
26.     serv_addr.sin_family=AF_INET;
27.     serv_addr.sin_addr.s_addr=htonl(INADDR_ANY);
28.     serv_addr.sin_port=htons(atoi(argv[1]));
29.
30.     if(bind(serv_sock, (struct sockaddr*) &serv_addr, sizeof(serv_addr))==-1)
31.         error_handling("bind() error");
32.
33.     if(listen(serv_sock, 5)==-1)
34.         error_handling("listen() error");
35.
36.     clnt_addr_size=sizeof(clnt_addr);
37.     clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_addr,&clnt_addr_size);
38.     if(clnt_sock==-1)
39.         error_handling("accept() error");
40.
```

```
41.     write(clnt_sock, message, sizeof(message));
42.     close(clnt_sock);
43.     close(serv_sock);
44.     return 0;
45. }
46.
47. void error_handling(char *message)
48. {
49.     fputs(message, stderr);
50.     fputc('\n', stderr);
51.     exit(1);
52. }
```

代码说明

- 第21行：服务器端实现过程中先要创建套接字。第21行创建套接字，但此时的套接字尚非真正的服务器端套接字。
- 第25~31行：为了完成套接字地址分配，初始化结构体变量并调用bind函数。
- 第33行：调用listen函数进入等待连接请求状态。连接请求等待队列的长度设置为5。此时的套接字才是服务器端套接字。
- 第37行：调用accept函数从队头取1个连接请求与客户端建立连接，并返回创建的套接字文件描述符。另外，调用accept函数时若等待队列为空，则accept函数不会返回，直到队列中出现新的客户端连接。
- 第41、42行：调用write函数向客户端传输数据，调用close函数关闭连接。

我们按照服务器端实现顺序把看起来很复杂的第1章代码进行了重新整理。可以看出，服务器端的基本实现过程实际上非常简单。

TCP客户端的默认函数调用顺序

接下来讲解客户端的实现顺序。如前所述，这要比服务器端简单许多。因为创建套接字和请求连接就是客户端的全部内容，如图4-9所示。

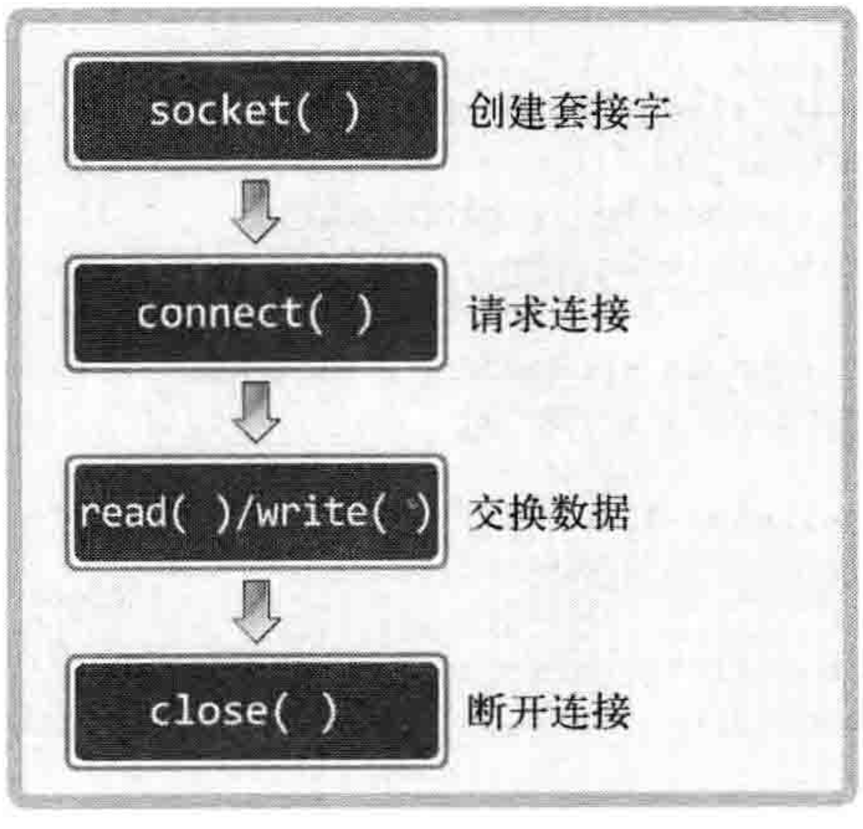


图4-9　TCP客户端函数调用顺序

与服务器端相比，区别就在于“请求连接”，它是创建客户端套接字后向服务器端发起的连接请求。服务器端调用listen函数后创建连接请求等待队列，之后客户端即可请求连接。那如何发起连接请求呢？通过调用如下函数完成。

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

int connect(int sock, struct sockaddr * servaddr, socklen_t addrlen);
```

➔ 成功时返回 0，失败时返回-1。

- sock　客户端套接字文件描述符。
- servaddr　保存目标服务器端地址信息的变量地址值。
- addrlen　以字节为单位传递已传递给第二个结构体参数servaddr的地址变量长度。

客户端调用connect函数后，发生以下情况之一才会返回（完成函数调用）。

- 服务器端接收连接请求。
- 发生断网等异常情况而中断连接请求。

需要注意，所谓的“接收连接”并不意味着服务器端调用accept函数，其实是服务器端把连接请求信息记录到等待队列。因此connect函数返回后并不立即进行数据交换。

知识补给站　客户端套接字地址信息在哪？

实现服务器端必经过程之一就是给套接字分配IP和端口号。但客户端实现过程中并未出现套接字地址分配，而是创建套接字后立即调用connect函数。难道客户端套接字无需分配IP和端口？当然不是！网络数据交换必须分配IP和端口。既然如此，那客户端套接字何时、何地、如何分配地址呢？

- 何时？ 调用connect函数时。
- 何地？ 操作系统，更准确地说是在内核中。
- 如何？ IP用计算机（主机）的IP，端口随机。

客户端的IP地址和端口在调用connect函数时自动分配，无需调用标记的bind函数进行分配。

回顾 Hello world 客户端

与前面回顾Hello world服务器端一样，再来分析一下Hello world客户端。

❖ hello_client.c

```
1.  /* 头文件及函数声明关系
2.  请参考第1章的源代码hello_client.c */
3.
4.  int main(int argc, char * argv[])
5.  {
6.      int sock;
7.      struct sockaddr_in serv_addr;
8.      char message[30];
9.      int str_len;
10.
11.     if(argc!=3)
12.     {
13.         printf("Usage : %s <IP> <port>\n", argv[0]);
14.         exit(1);
15.     }
16.
17.     sock=socket(PF_INET, SOCK_STREAM, 0);
18.     if(sock == -1)
19.         error_handling("socket() error");
20.
21.     memset(&serv_addr, 0, sizeof(serv_addr));
22.     serv_addr.sin_family=AF_INET;
23.     serv_addr.sin_addr.s_addr=inet_addr(argv[1]);
24.     serv_addr.sin_port=htons(atoi(argv[2]));
25.
26.     if(connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr))==-1)
27.         error_handling("connect() error!");
28.
29.     str_len=read(sock, message, sizeof(message)-1);
30.     if(str_len==-1)
31.         error_handling("read() error!");
32.
33.     printf("Message from server : %s \n", message);
34.     close(sock);
35.     return 0;
36. }
37.
38. void error_handling(char *message)
39. {
40.     fputs(message, stderr);
41.     fputc('\n', stderr);
42.     exit(1);
43. }
```

代码说明

- 第17行：创建准备连接服务器端的套接字，此时创建的是TCP套接字。
- 第21~24行：结构体变量serv_addr中初始化IP和端口信息。初始化值为目标服务器端套接字的IP和端口信息。
- 第26行：调用connect函数向服务器端发送连接请求。
- 第29行：完成连接后，接收服务器端传输的数据。
- 第34行：接收数据后调用close函数关闭套接字，结束与服务器端的连接。

各位应该完全理解了TCP服务器端和客户端的源代码。若还有不明白的部分，请多加复习。

基于 TCP 的服务器端/客户端函数调用关系

前面讲解了TCP服务器端/客户端的实现顺序，实际上二者并非相互独立，各位应该可以勾勒出它们之间的交互过程，如图4-10所示。之前都详细讨论过，大家就当作复习吧。

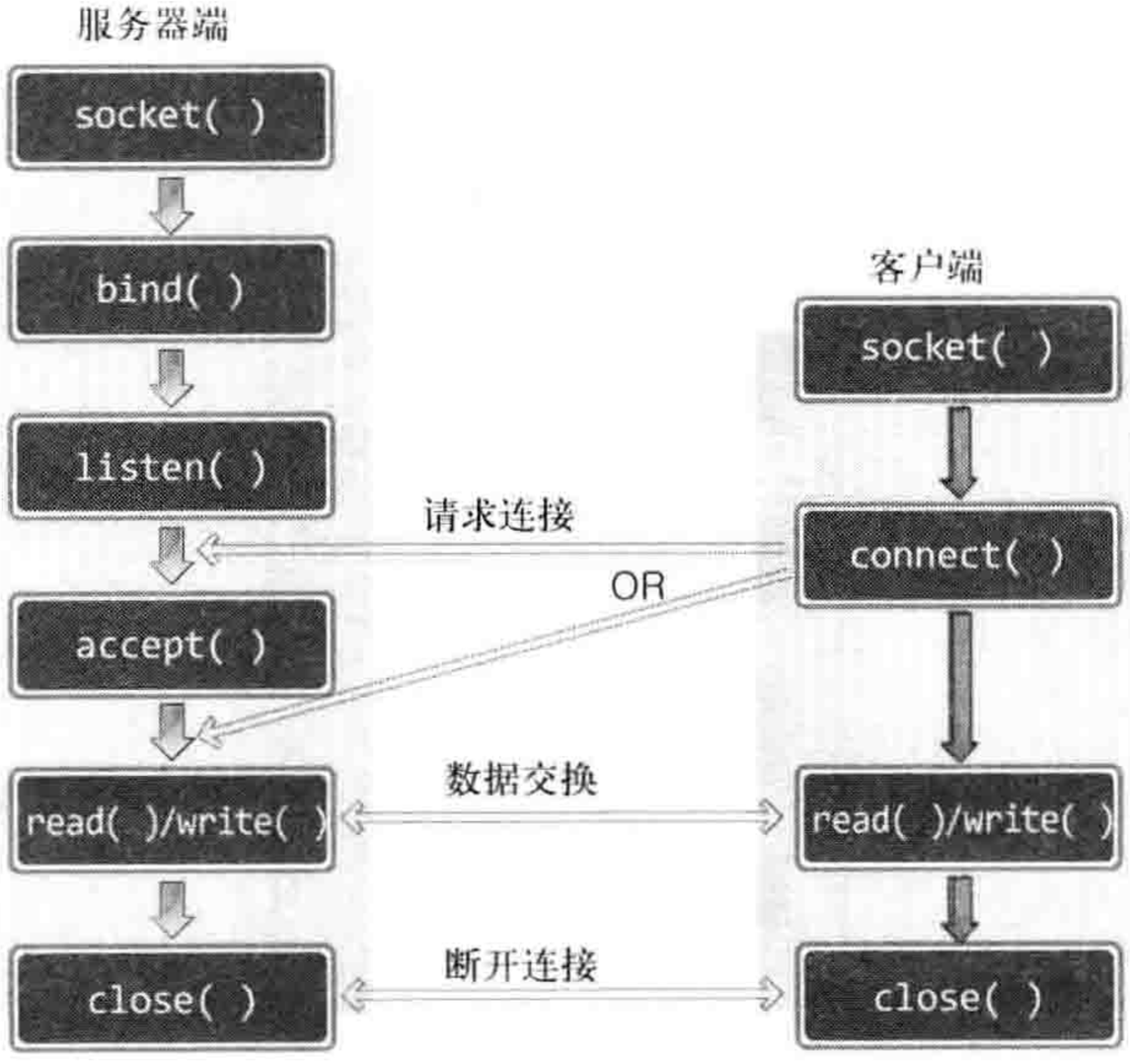


图4-10 函数调用关系

图4-10的总体流程整理如下：服务器端创建套接字后连续调用bind、listen函数进入等待状态，客户端通过调用connect函数发起连接请求。需要注意的是，客户端只能等到服务器端调用listen函数后才能调connect函数。同时要清楚，客户端调用connect函数前，服务器端有可能率先调用accept函数。当然，此时服务器端在调用accept函数时进入阻塞（blocking）状态，直到客户端调connect函数为止。

4.3 实现迭代服务器端/客户端

本节编写回声（echo）服务器端/客户端。顾名思义，服务器端将客户端传输的字符串数据原封不动地传回客户端，就像回声一样。在此之前，需要先解释一下迭代服务器端。

实现迭代服务器端

之前讨论的Hello world服务器端处理完1个客户端连接请求即退出，连接请求等待队列实际

没有太大意义。但这并非我们想象的服务器端。设置好等待队列的大小后，应向所有客户端提供服务。如果想继续受理后续的客户端连接请求，应怎样扩展代码？最简单的办法就是插入循环语句反复调用accept函数，如图4-11所示。

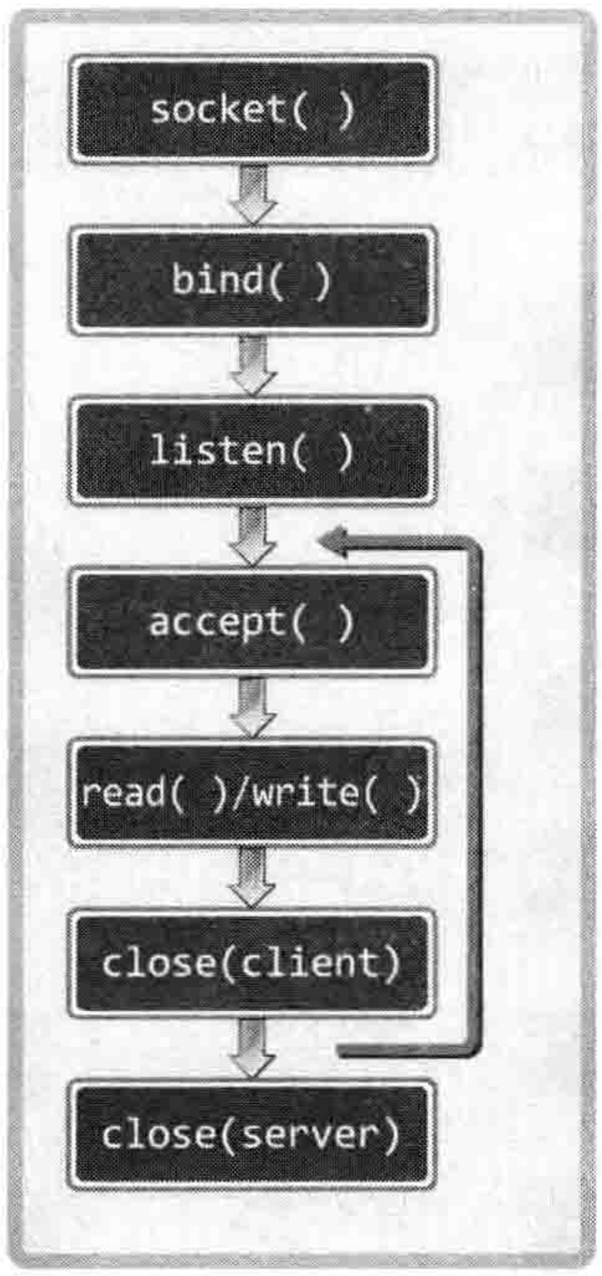


图4-11　迭代服务器端的函数调用顺序

从图4-11可以看出，调用accept函数后，紧接着调用I/O相关的read、write函数，然后调用close函数。这并非针对服务器端套接字，而是针对accept函数调用时创建的套接字。

调用close函数就意味着结束了针对某一客户端的服务。此时如果还想服务于其他客户端，就要重新调用accept函数。

“这算什么呀？又不是银行窗口，好歹也是个服务器端，难道同一时刻只能服务于一个客户端吗？”

是的！同一时刻确实只能服务于一个客户端。将来学完进程和线程后，就可以编写同时服务多个客户端的服务器端了。目前只能做到这一步，虽然很遗憾，但请各位不要心急。

迭代回声服务器端/客户端

前面讲的就是迭代服务器端。即使服务器端以迭代方式运转，客户端代码亦无太大区别。接下来创建迭代回声服务器端及与其配套的回声客户端。首先整理一下程序的基本运行方式。

- 服务器端在同一时刻只与一个客户端相连，并提供回声服务。
- 服务器端依次向5个客户端提供服务并退出。
- 客户端接收用户输入的字符串并发送到服务器端。
- 服务器端将接收的字符串数据传回客户端，即“回声”。
- 服务器端与客户端之间的字符串回声一直执行到客户端输入Q为止。

首先介绍满足以上要求的回声服务器端代码。希望各位注意观察accept函数的循环调用过程。

❖ echo_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 1024
9.  void error_handling(char *message);
10.
11. int main(int argc, char *argv[])
12. {
13.     int serv_sock, clnt_sock;
14.     char message[BUF_SIZE];
15.     int str_len, i;
16.
17.     struct sockaddr_in serv_adr, clnt_adr;
18.     socklen_t clnt_adr_sz;
19.
20.     if(argc!=2) {
21.         printf("Usage : %s <port>\n", argv[0]);
22.         exit(1);
23.     }
24.
25.     serv_sock=socket(PF_INET, SOCK_STREAM, 0);
26.     if(serv_sock==-1)
27.         error_handling("socket() error");
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.     if(bind(serv_sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr))==-1)
35.         error_handling("bind() error");
36.
37.     if(listen(serv_sock, 5)==-1)
38.         error_handling("listen() error");
39.
40.     clnt_adr_sz=sizeof(clnt_adr);
41.
```

```
42.     for(i=0; i<5; i++)
43.     {
44.         clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_adr, &clnt_adr_sz);
45.         if(clnt_sock==-1)
46.             error_handling("accept() error");
47.         else
48.             printf("Connected client %d \n", i+1);
49.
50.         while((str_len=read(clnt_sock, message, BUF_SIZE))!=0)
51.             write(clnt_sock, message, str_len);
52.
53.         close(clnt_sock);
54.     }
55.     close(serv_sock);
56.     return 0;
57. }
58.
59. void error_handling(char *message)
60. {
61.     fputs(message, stderr);
62.     fputc('\n', stderr);
63.     exit(1);
64. }
```

代码说明

- 第42~54行：为处理5个客户端连接而添加的循环语句。共调用5次accept函数，依次向5个客户端提供服务。
- 第50、51行：实际完成回声服务的代码，原封不动地传输读取的字符串。
- 第53行：针对套接字调用close函数，向连接的相应套接字发送EOF。换言之，客户端套接字若调用close函数，则第50行的循环条件变成假（false），因此执行第53行的代码。
- 第55行：向5个客户端提供服务后关闭服务器端套接字并终止程序。

❖ 运行结果：echo_server.c

```
root@my_linux:/tcpip# gcc echo_server.c -o eserver
root@my_linux:/tcpip# ./eserver 9190
Connected client 1
Connected client 2
Connected client 3
```

从运行结果可以看出，示例运行过程中输出了与客户端的连接信息。该程序目前与第3个客户端相连接。接下来给出回声客户端代码。

❖ echo_client.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 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。

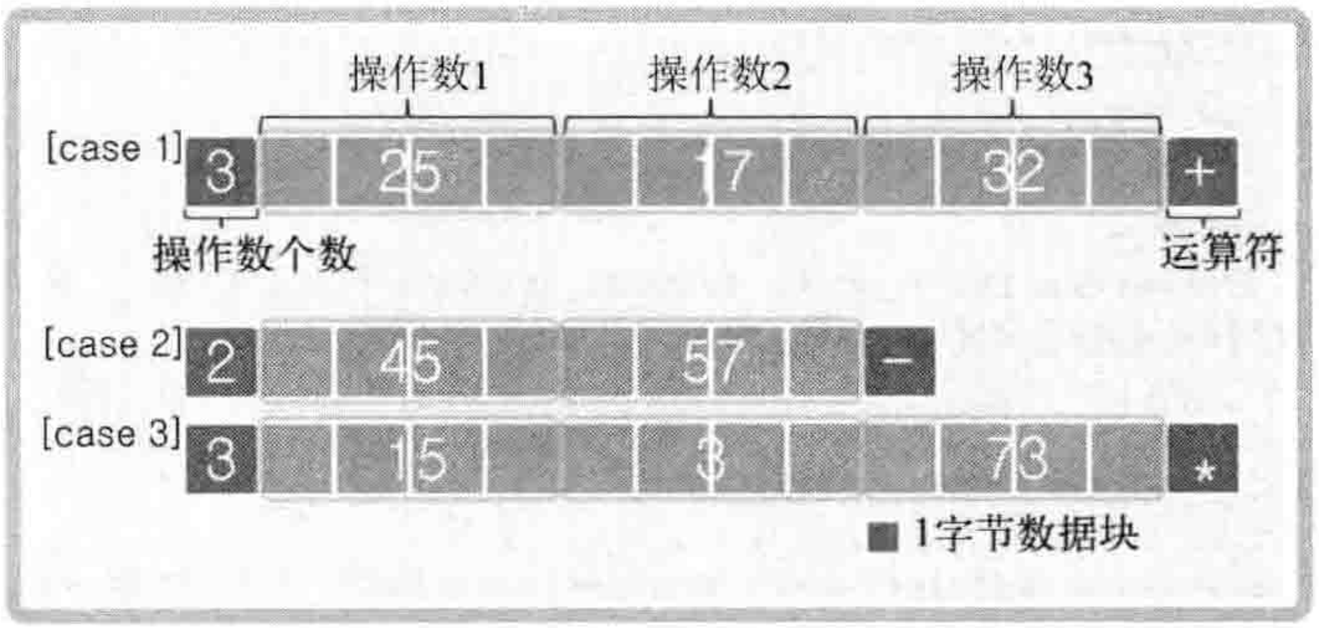


图5-1 客户端op_client.c的数据传送格式

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

```
#include <"与其他示例的头声明相同，故省略。">
#define BUF_SIZE 1024
#define OPSZ 4
void error_handling(char *message);
int calculate(int opnum, int opnds[], char oprator);

int main(int argc, char *argv[])
{
    int serv_sock, clnt_sock;
    char opinfo[BUF_SIZE];
    int result, opnd_cnt, i;
    int recv_cnt, recv_len;
    struct sockaddr_in serv_adr, clnt_adr;
    socklen_t clnt_adr_sz;
    if(argc!=2) {
        printf("Usage : %s <port>\n", argv[0]);
        exit(1);
    }

    serv_sock=socket(PF_INET, SOCK_STREAM, 0);
    if(serv_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=htonl(INADDR_ANY);
    serv_adr.sin_port=htons(atoi(argv[1]));

    if(bind(serv_sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr))==-1)
        error_handling("bind() error");
    if(listen(serv_sock, 5)==-1)
        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。

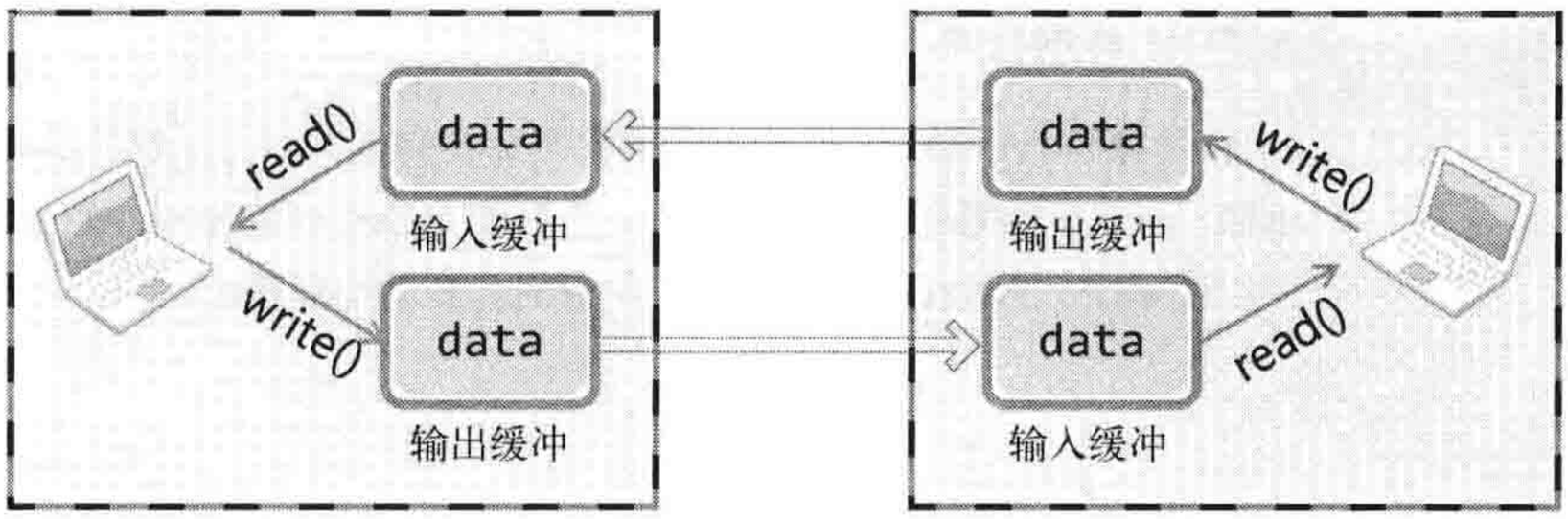


图5-2 TCP套接字的I/O缓冲

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。

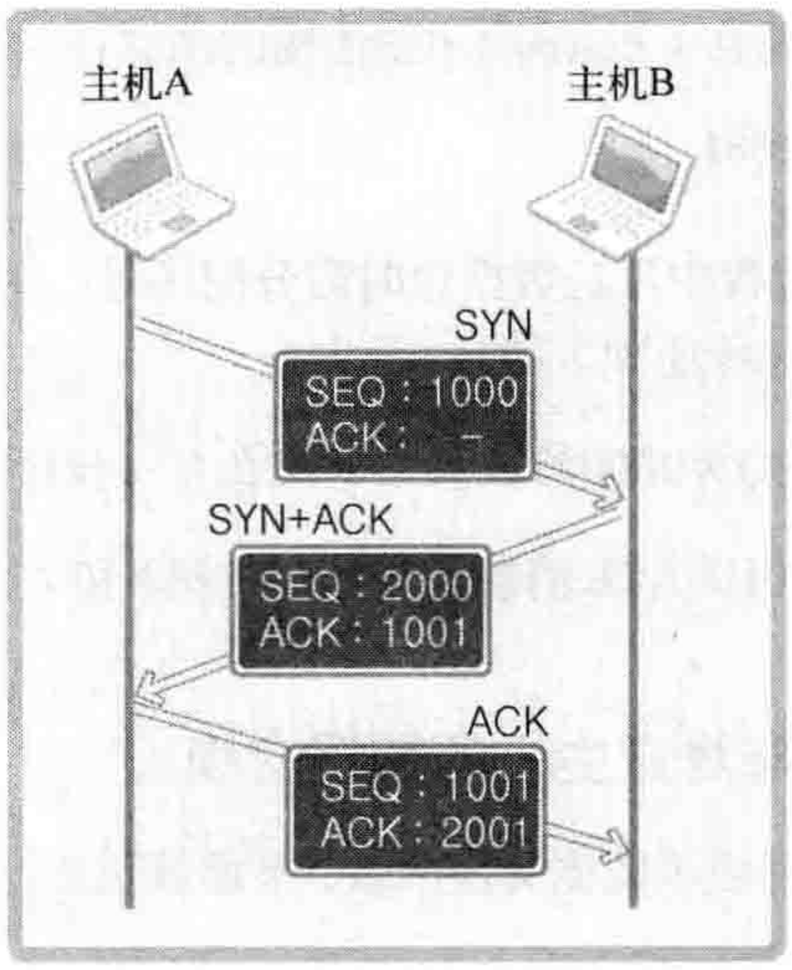


图5-3 TCP套接字的连接设置过程

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。

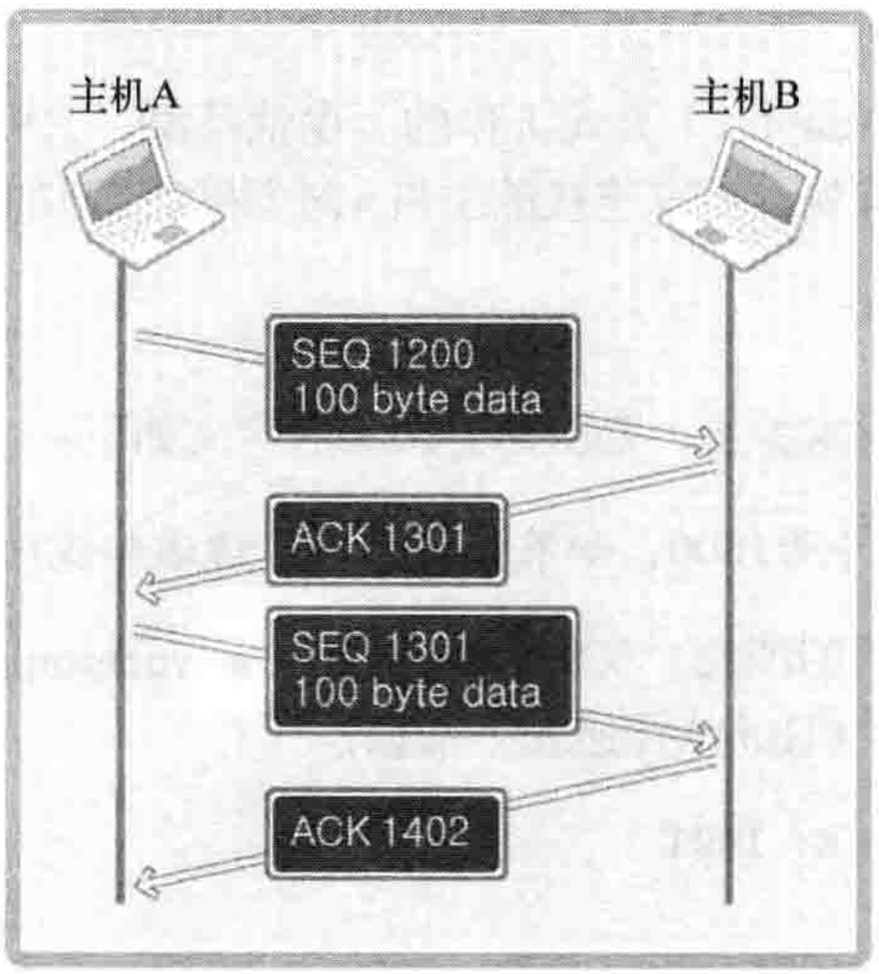


图5-4　TCP套接字的数据交换过程

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。

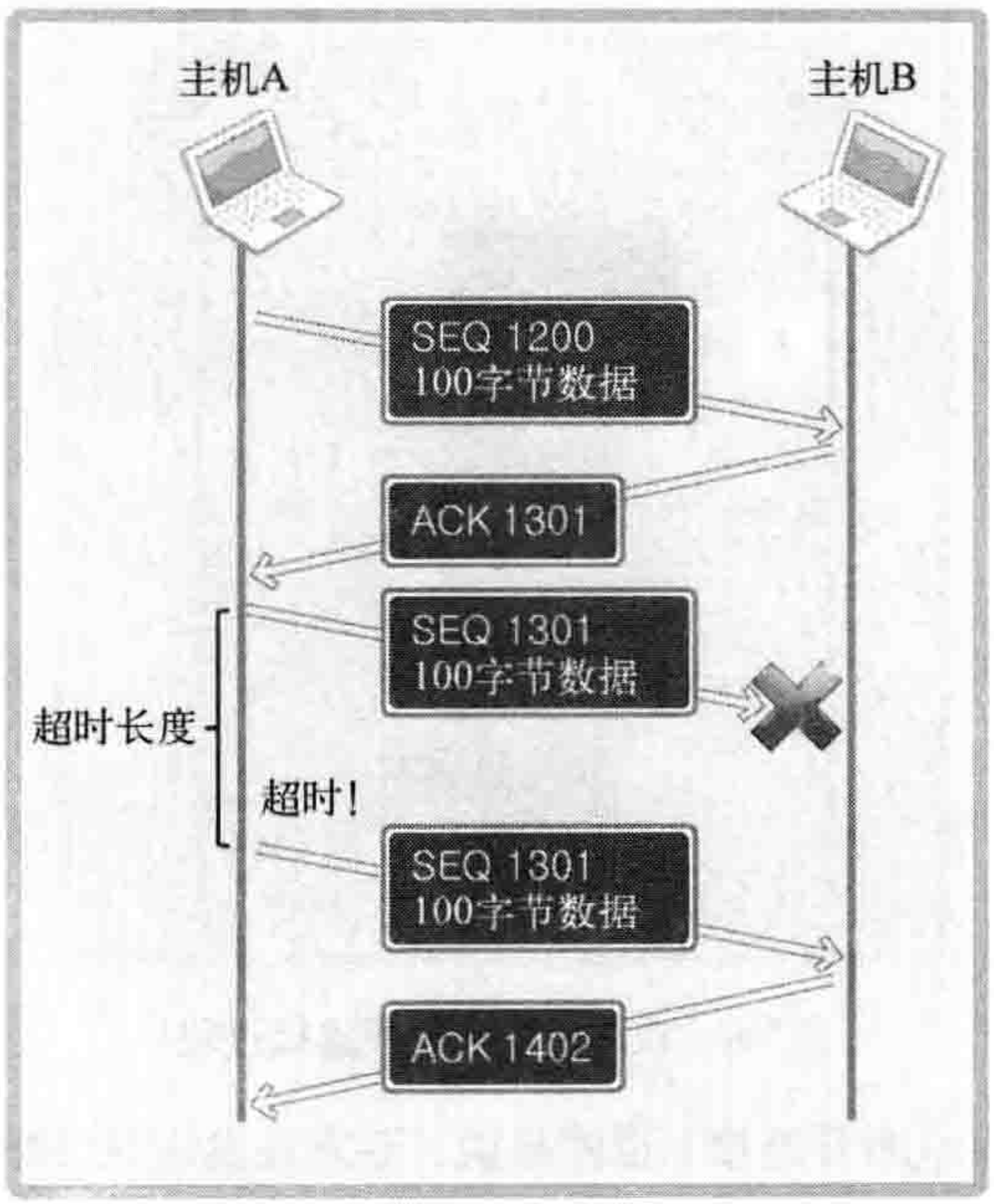


图5-5 TCP套接字数据传输中发生错误

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。

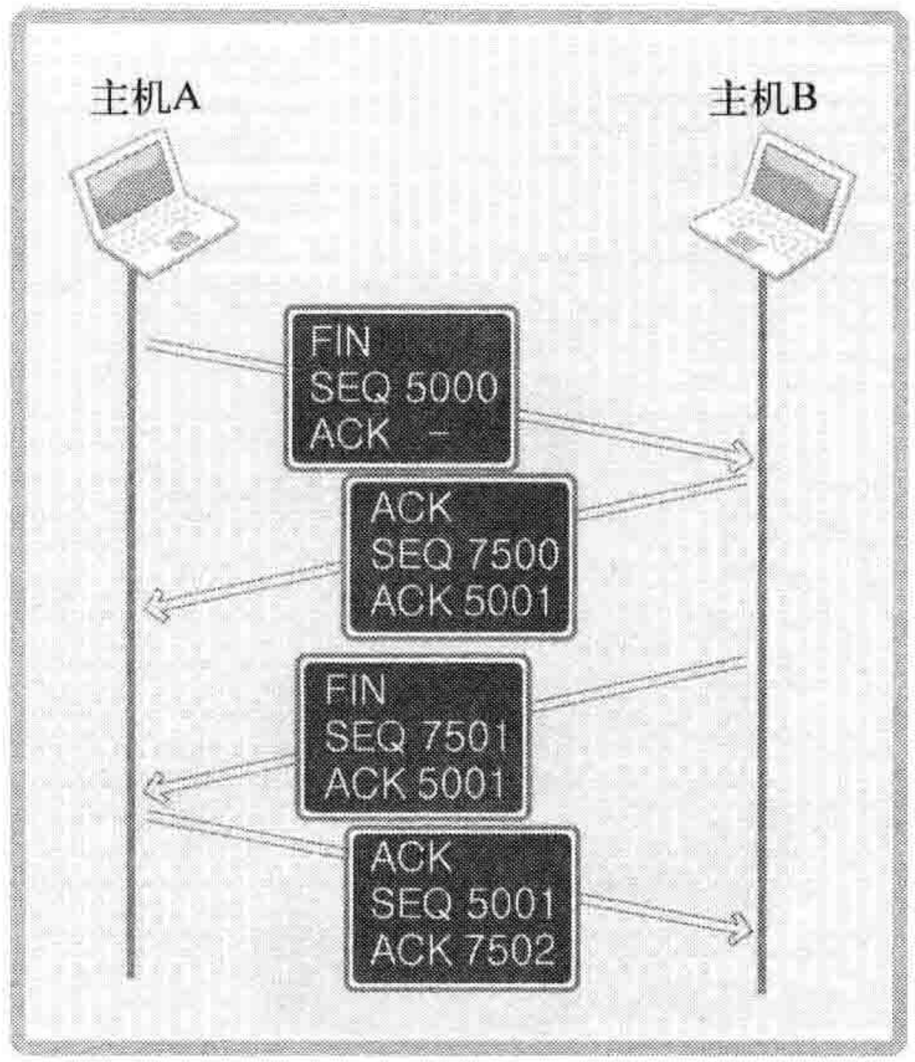


图5-6　TCP套接字断开连接过程

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

6

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。

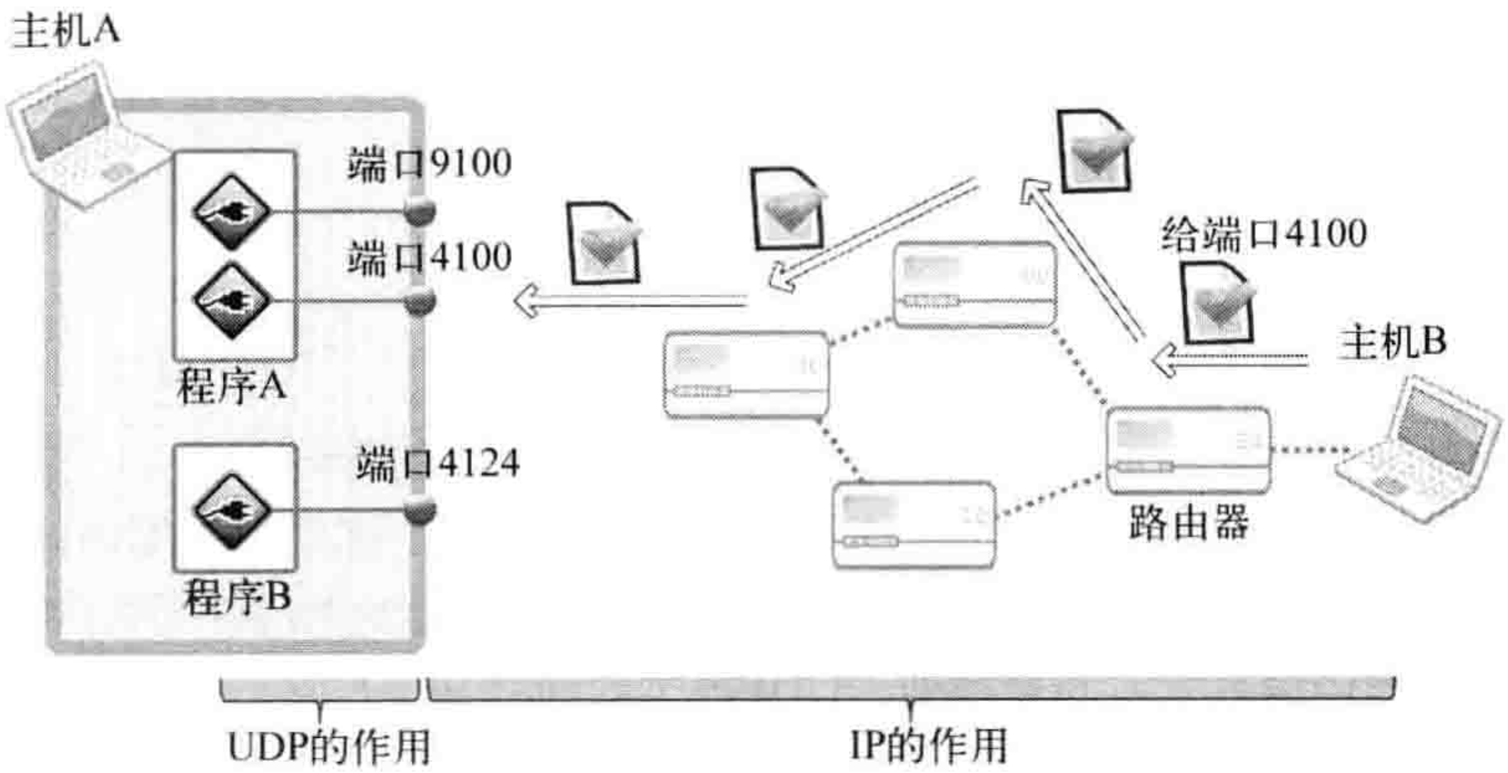


图6-1　数据包传输中UDP和IP的作用

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。

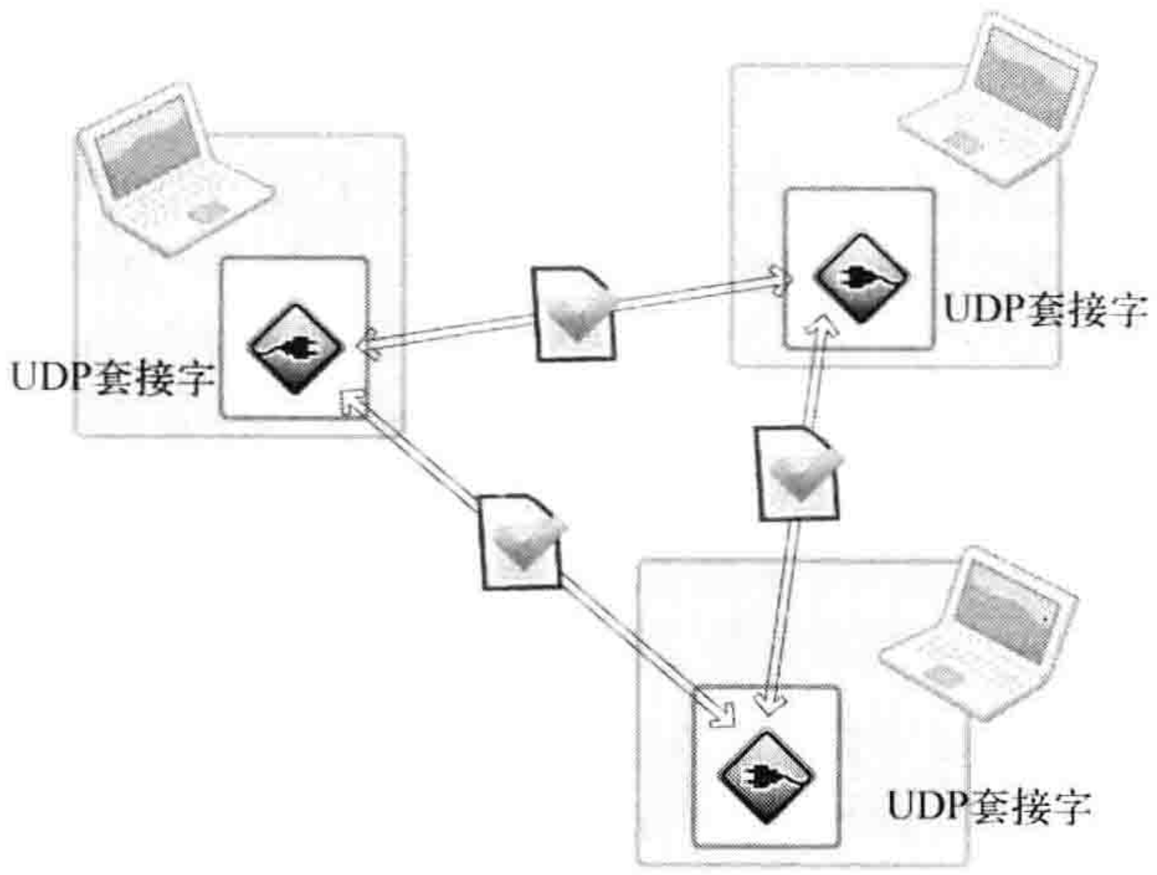


图6-2　UDP套接字通信模型

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);
```

```
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。

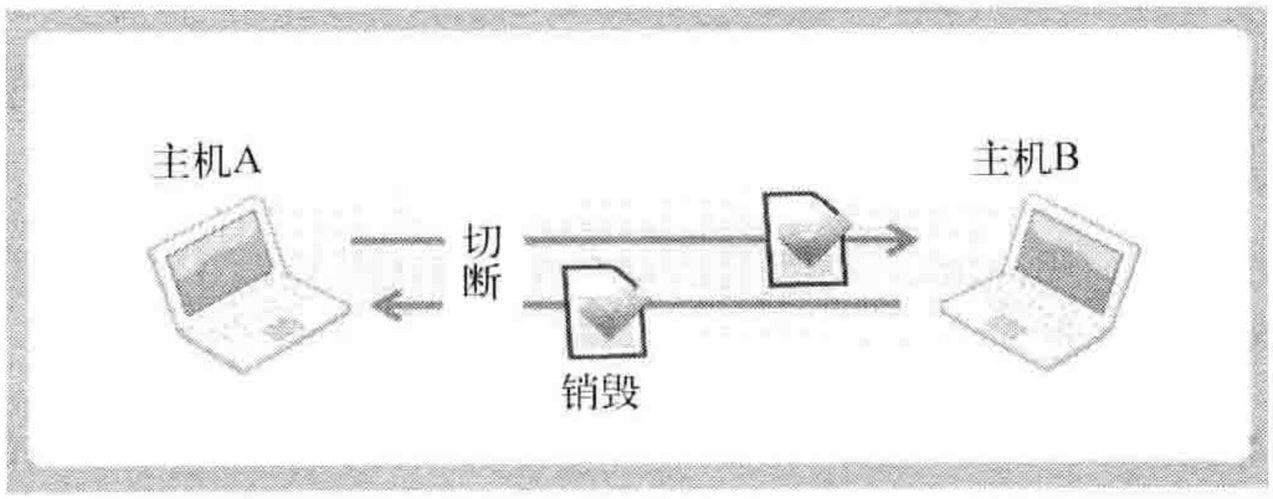


图7-1　单方面断开连接

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。

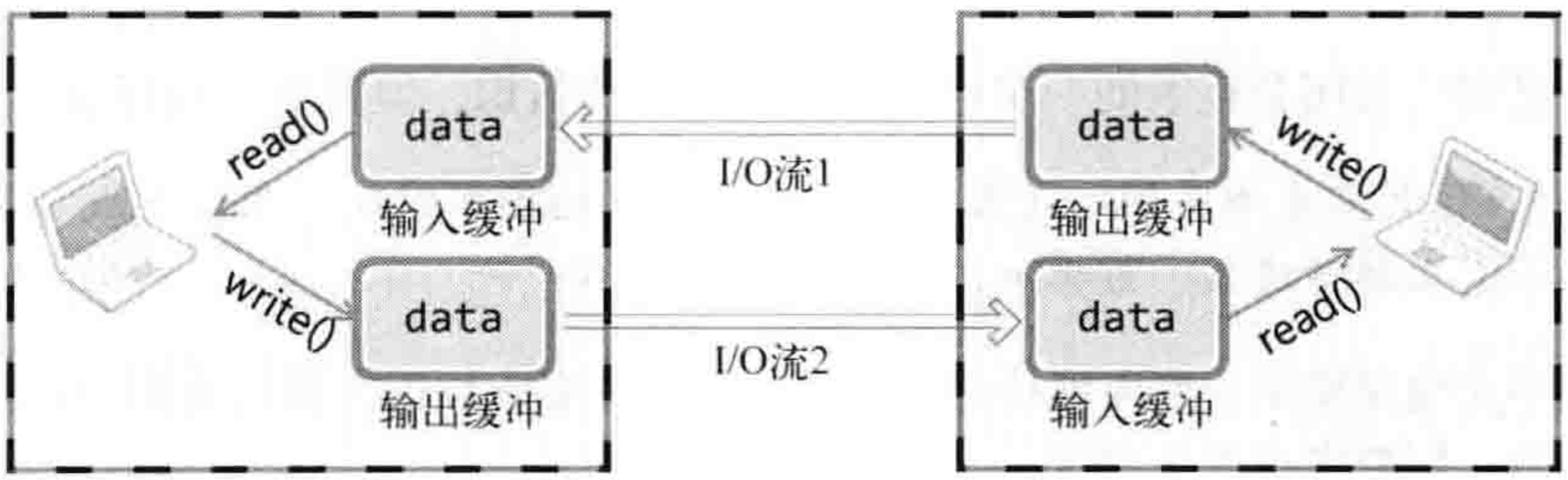


图7-2 套接字中生成的两个流

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. }
```

代码说明

- 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. }
```

代码说明

- 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

```
#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 hServSock, hClntSock;
    FILE * fp;
    char buf[BUF_SIZE];
    int readCnt;

    SOCKADDR_IN servAdr, clntAdr;
    int clntAdrSz;

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

    fp=fopen("file_server_win.c", "rb");
    hServSock=socket(PF_INET, SOCK_STREAM, 0);

    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]));

    bind(hServSock, (SOCKADDR*)&servAdr, sizeof(servAdr));
    listen(hServSock, 5);

    clntAdrSz=sizeof(clntAdr);
    hClntSock=accept(hServSock, (SOCKADDR*)&clntAdr, &clntAdrSz);

    while(1)
    {
        readCnt=fread((void*)buf, 1, BUF_SIZE, fp);
        if(readCnt<BUF_SIZE)
        {
            send(hClntSock, (char*)&buf, readCnt, 0);
            break;
        }
        send(hClntSock, (char*)&buf, BUF_SIZE, 0);
    }

```

```
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。

提 示

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。

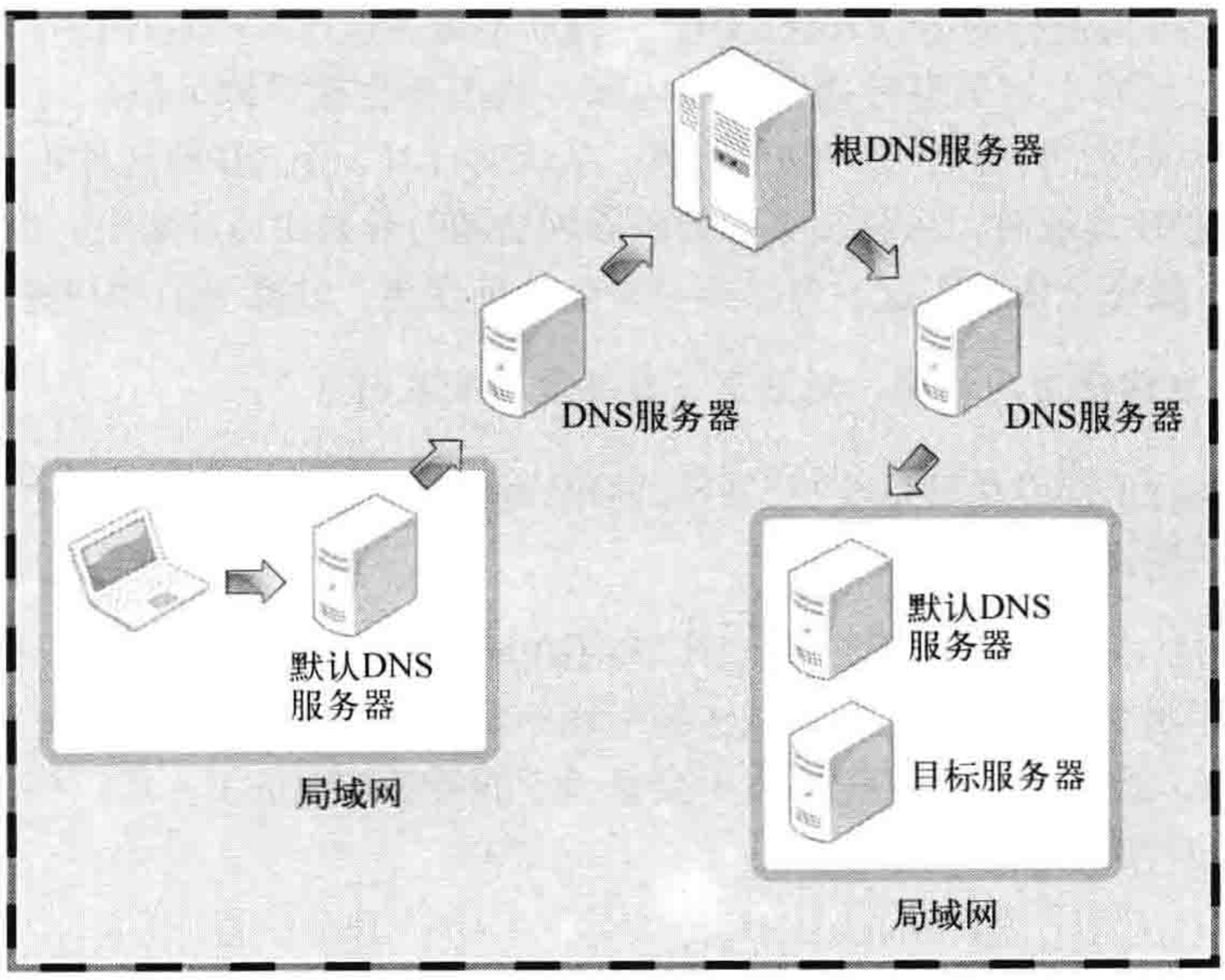


图8-1 DNS和请求获取IP地址信息

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。

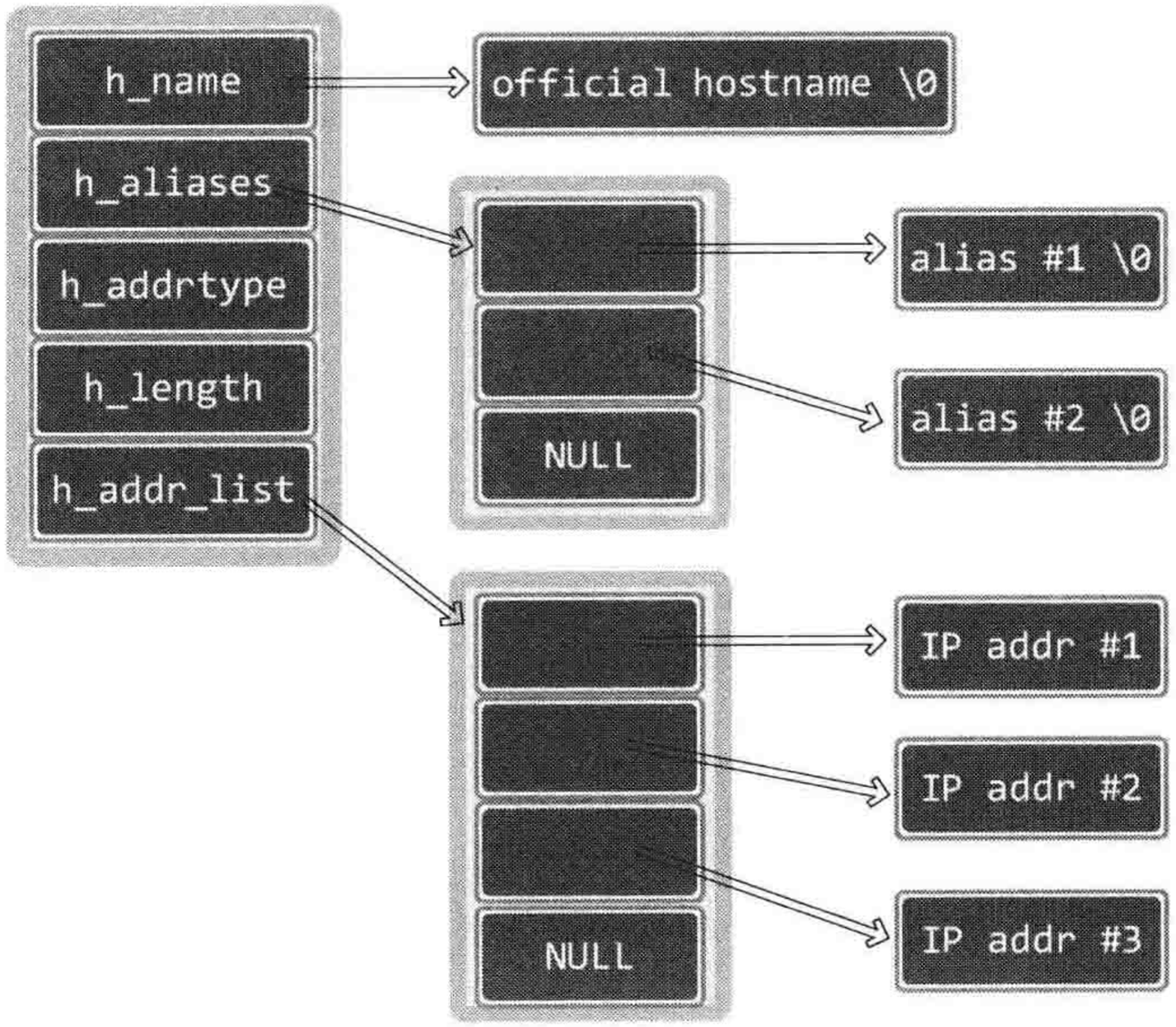


图8-2　hostent结构体变量

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. }
```

代码说明

- 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。

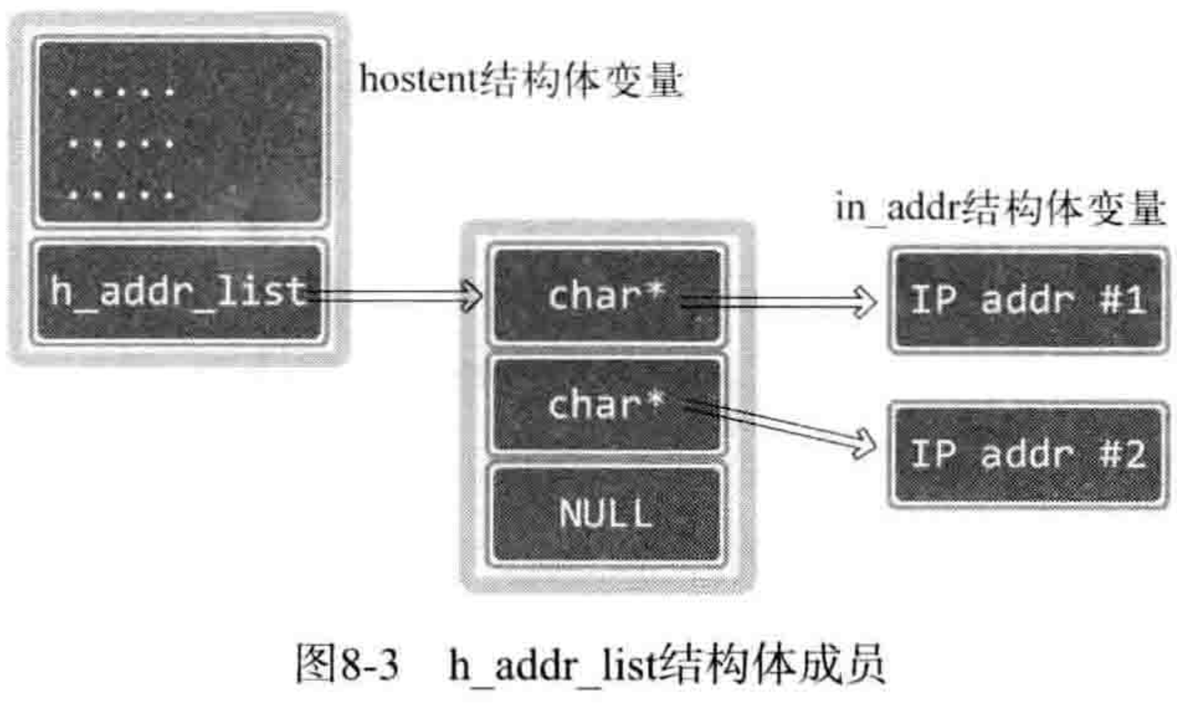


图8-3　h_addr_list结构体成员

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

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

int main(int argc, char *argv[])
{
    WSADATA wsaData;
    int i;
    struct hostent *host;
    if(argc!=2) {
        printf("Usage : %s <addr>\n", argv[0]);
        exit(1);
    }
    if(WSAStartup(MAKEWORD(2, 2), &wsaData)!=0)
        ErrorHandling("WSAStartup() error!");

    host=gethostbyname(argv[1]);
    if(!host)
        ErrorHandling("gethost... error");

    printf("Official name: %s \n", host->h_name);
    for(i=0; host->h_aliases[i]; i++)
        printf("Aliases %d: %s \n", i+1, host->h_aliases[i]);
    printf("Address type: %s \n",
        (host->h_addrtype==AF_INET)?"AF_INET":"AF_INET6");
    for(i=0; host->h_addr_list[i]; i++)
        printf("IP addr %d: %s \n", i+1,
            inet_ntoa(*(struct in_addr*)host->h_addr_list[i]));
    WSACleanup();
    return 0;
}

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

❖ gethostbyaddr_win.c

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