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。

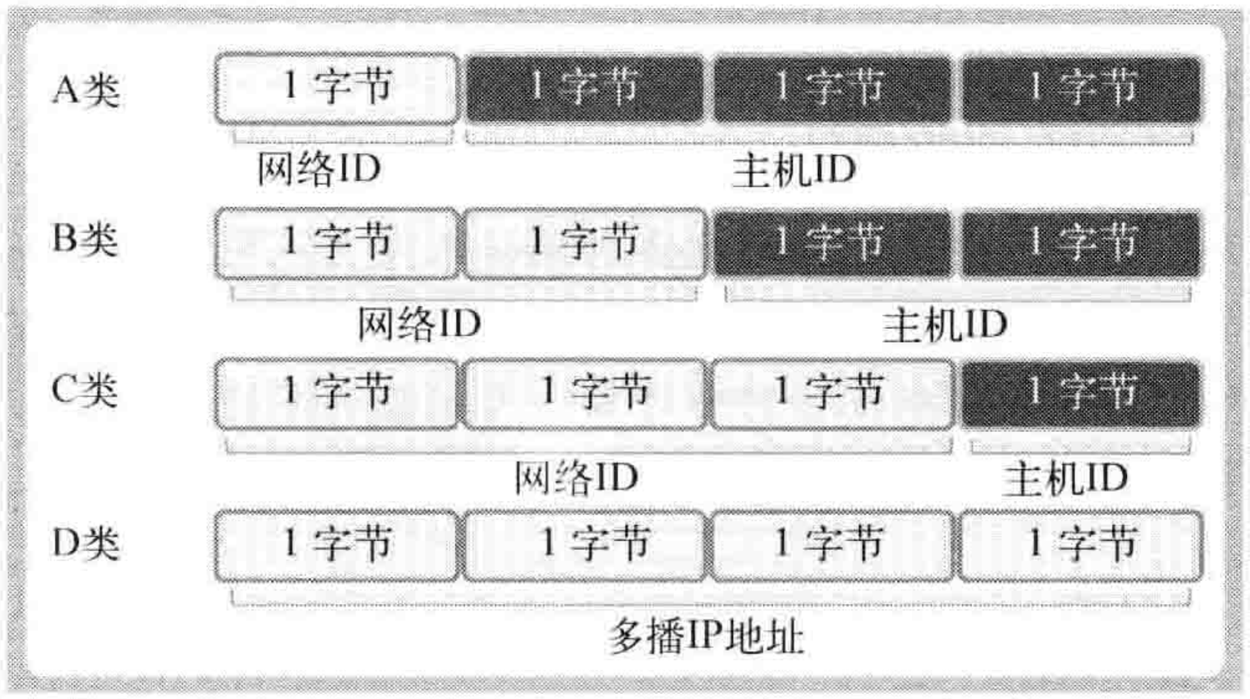

图3-1 IPv4地址族

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。

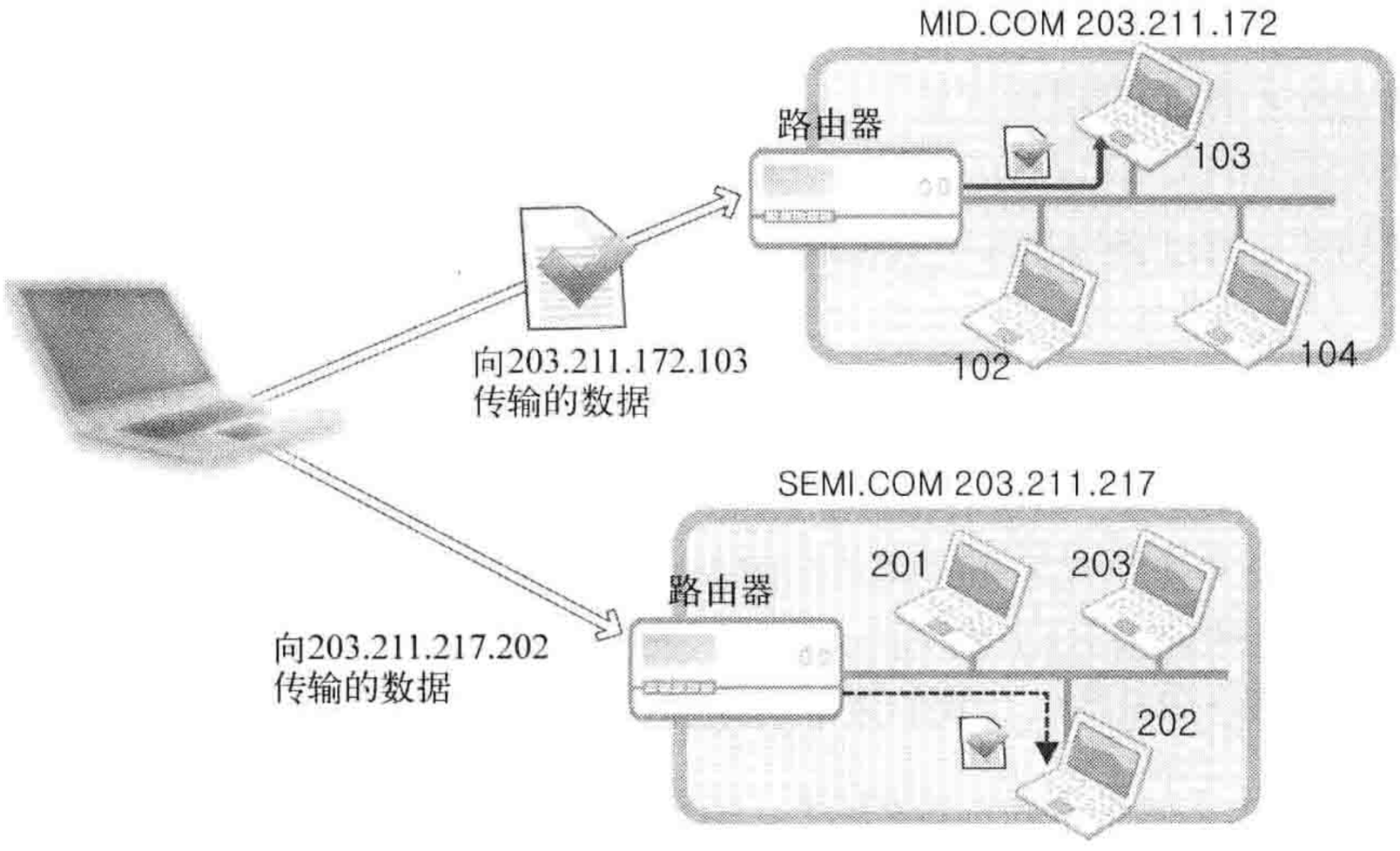

图3-2 基于IP地址的数据传输过程

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。

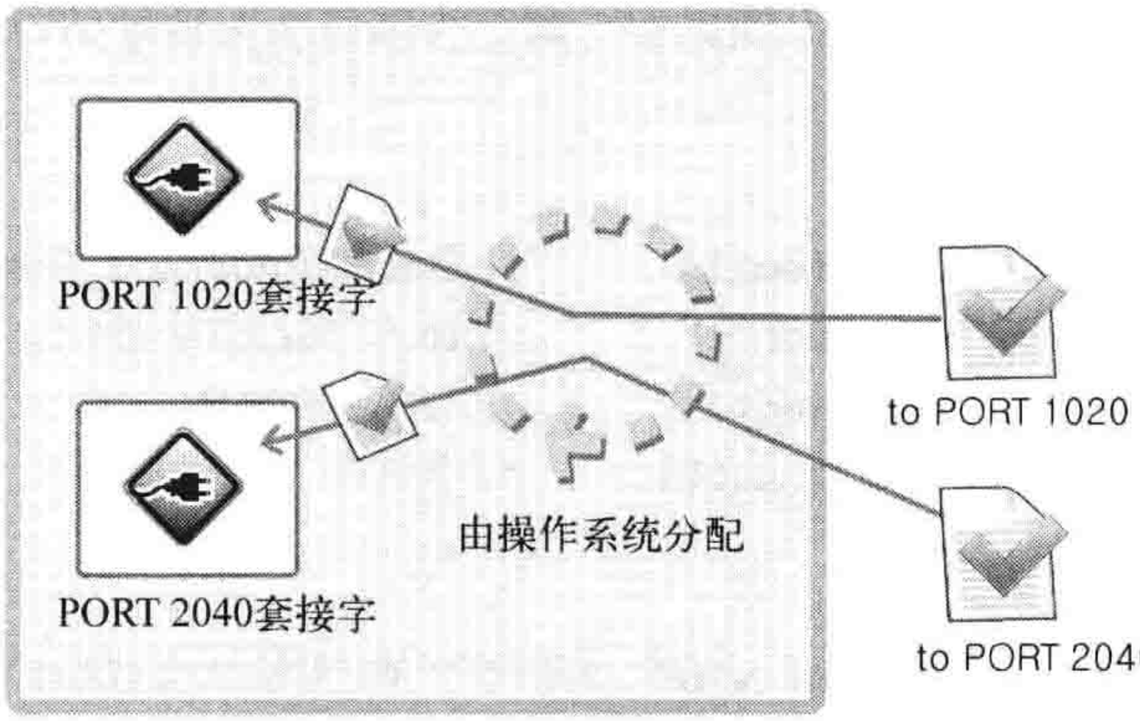

图3-3 数据分配过程

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。

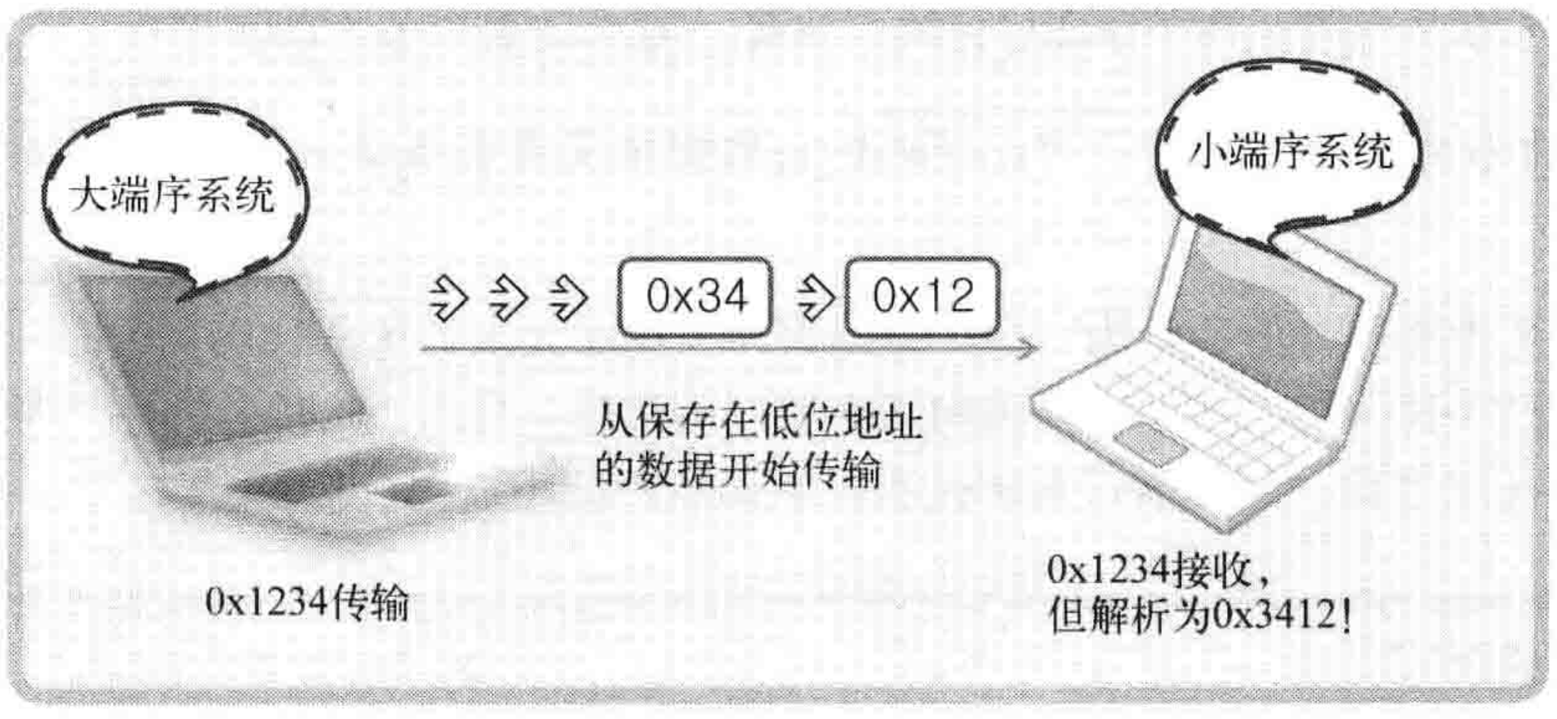

图3-6 字节序问题

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. }
```

代码说明

- 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. }
```

代码说明

- 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

```
1.  #include <stdio.h>
2.  #include <string.h>
3.  #include <winsock2.h>
4.  void ErrorHandling(char* message);
5.
6.  int main(int argc, char *argv[])
7.  {
8.      WSADATA wsaData;
9.      if(WSAStartup(MAKEWORD(2, 2), &wsaData)!=0)
10.         ErrorHandling("WSAStartup() error!");
11.
12.     /* inet_addr函数调用示例*/
13.     {
14.         char *addr="127.212.124.78";
15.         unsigned long conv_addr=inet_addr(addr);
16.         if(conv_addr==INADDR_NONE)
17.             printf("Error occured! \n");
18.         else
19.             printf("Network ordered integer addr: %#lx \n", conv_addr);
20.     }
21.
22.     /* inet_ntoa函数调用示例*/
23.     {
24.         struct sockaddr_in addr;
25.         char *strPtr;
26.         char strArr[20];
27.
28.         addr.sin_addr.s_addr=htonl(0x1020304);
29.         strPtr=inet_ntoa(addr.sin_addr);
30.         strcpy(strArr, strPtr);
31.         printf("Dotted-Decimal notation3 %s \n", strArr);
32.     }
33.
34.     WSACleanup();
35.     return 0;
36. }
37.
38. void ErrorHandling(char* message)
39. {
40.     //与之前示例一致，故省略!
41. }
```

❖ 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？

done!

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。

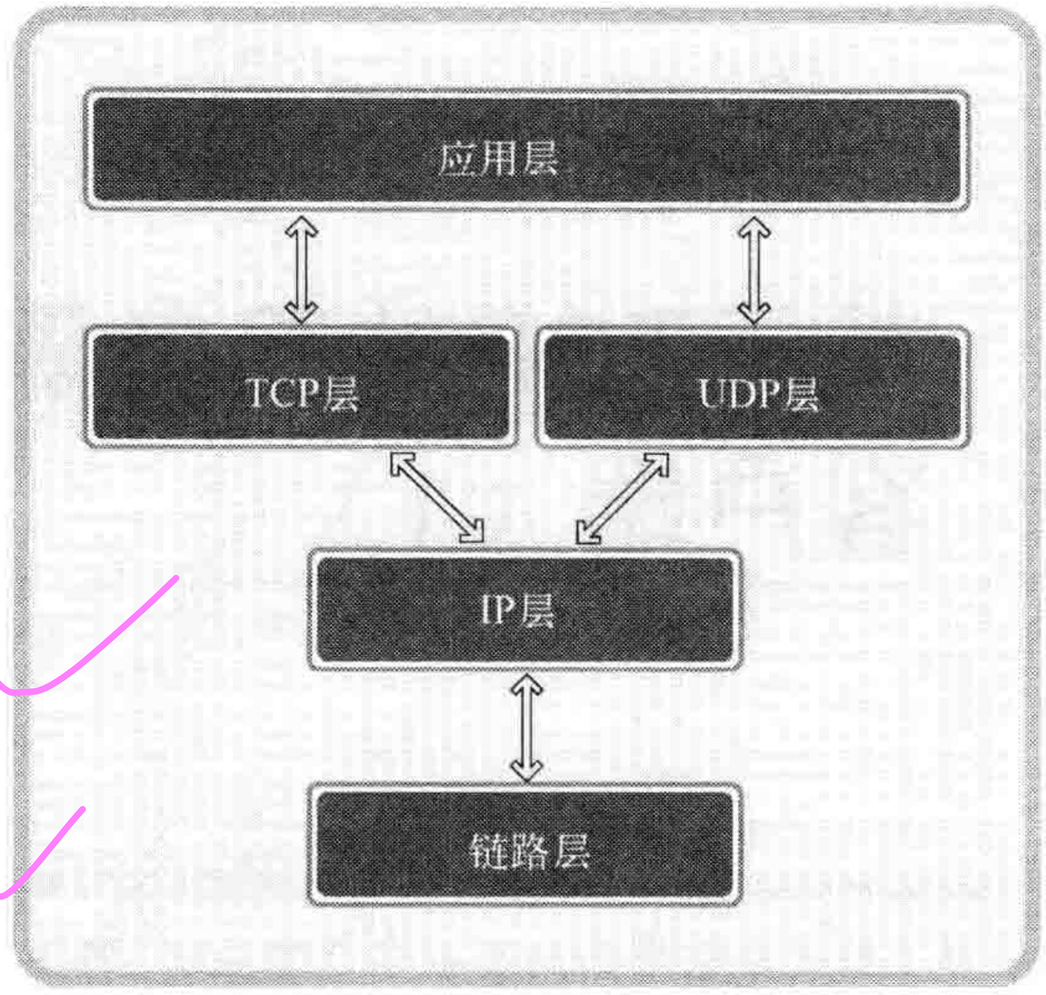

图4-1　TCP/IP协议栈

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。

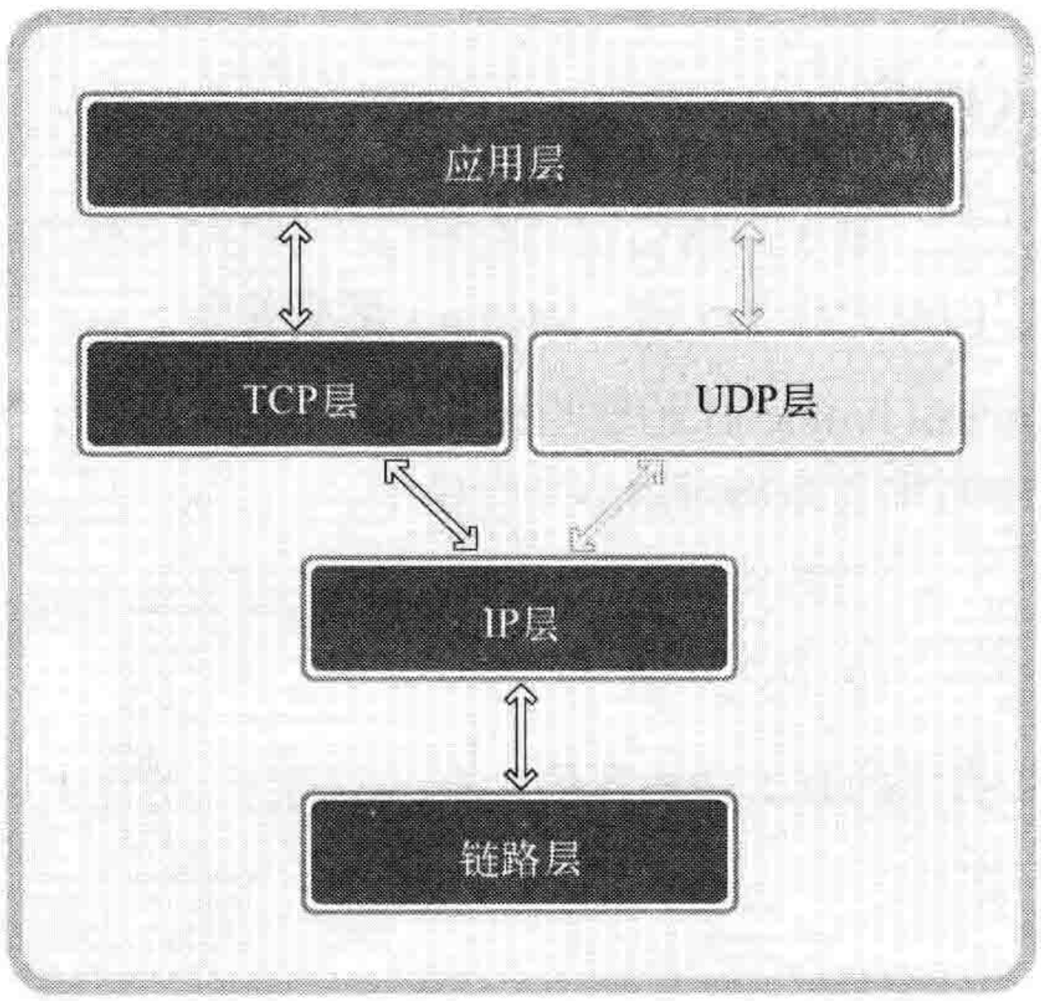

图4-2　TCP协议栈

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。

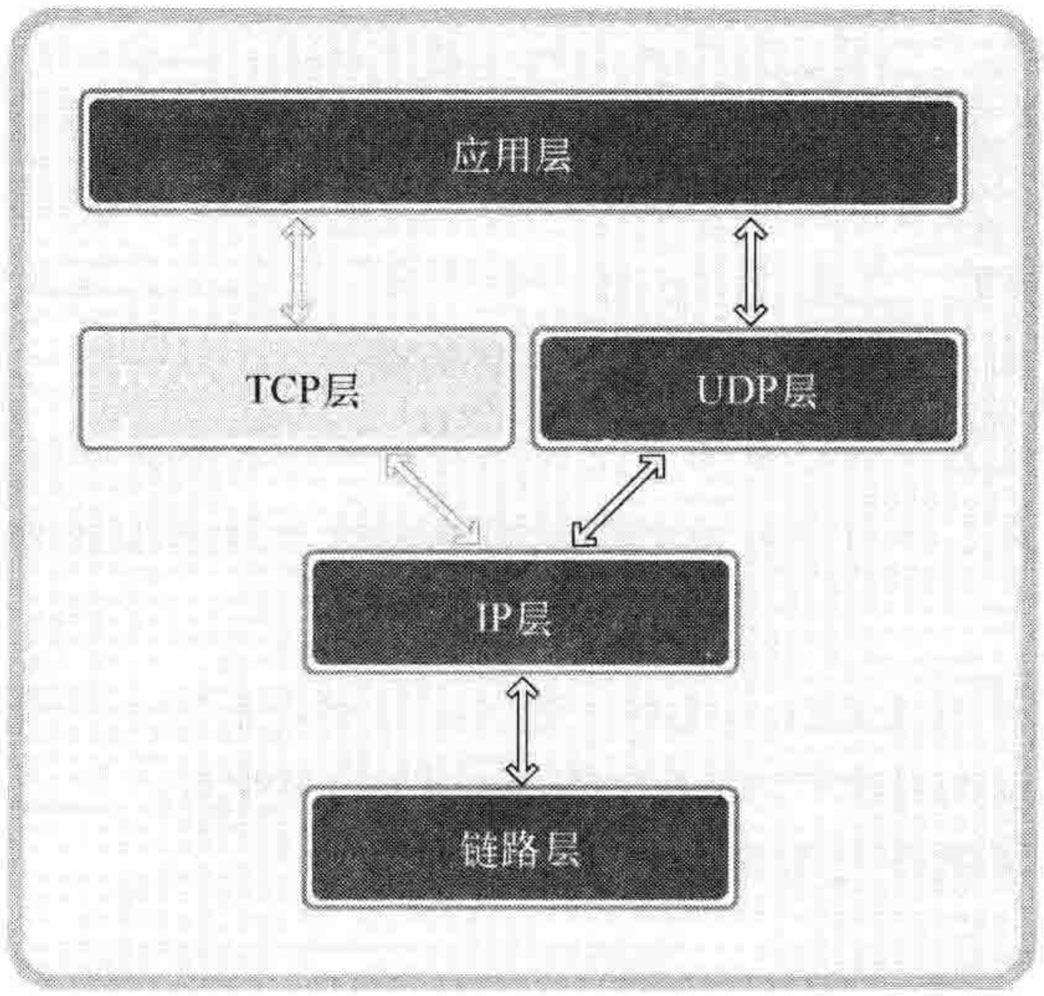

图4-3 UDP协议栈

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。

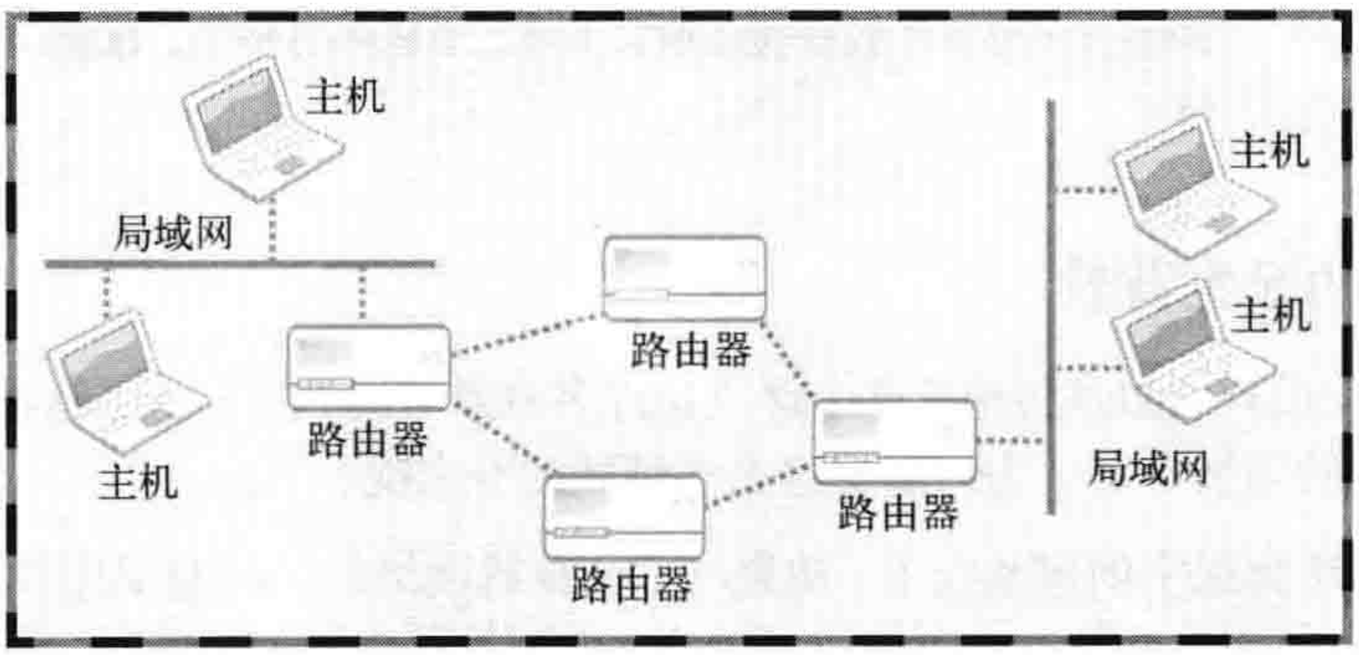

图4-4　网络连接结构

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。

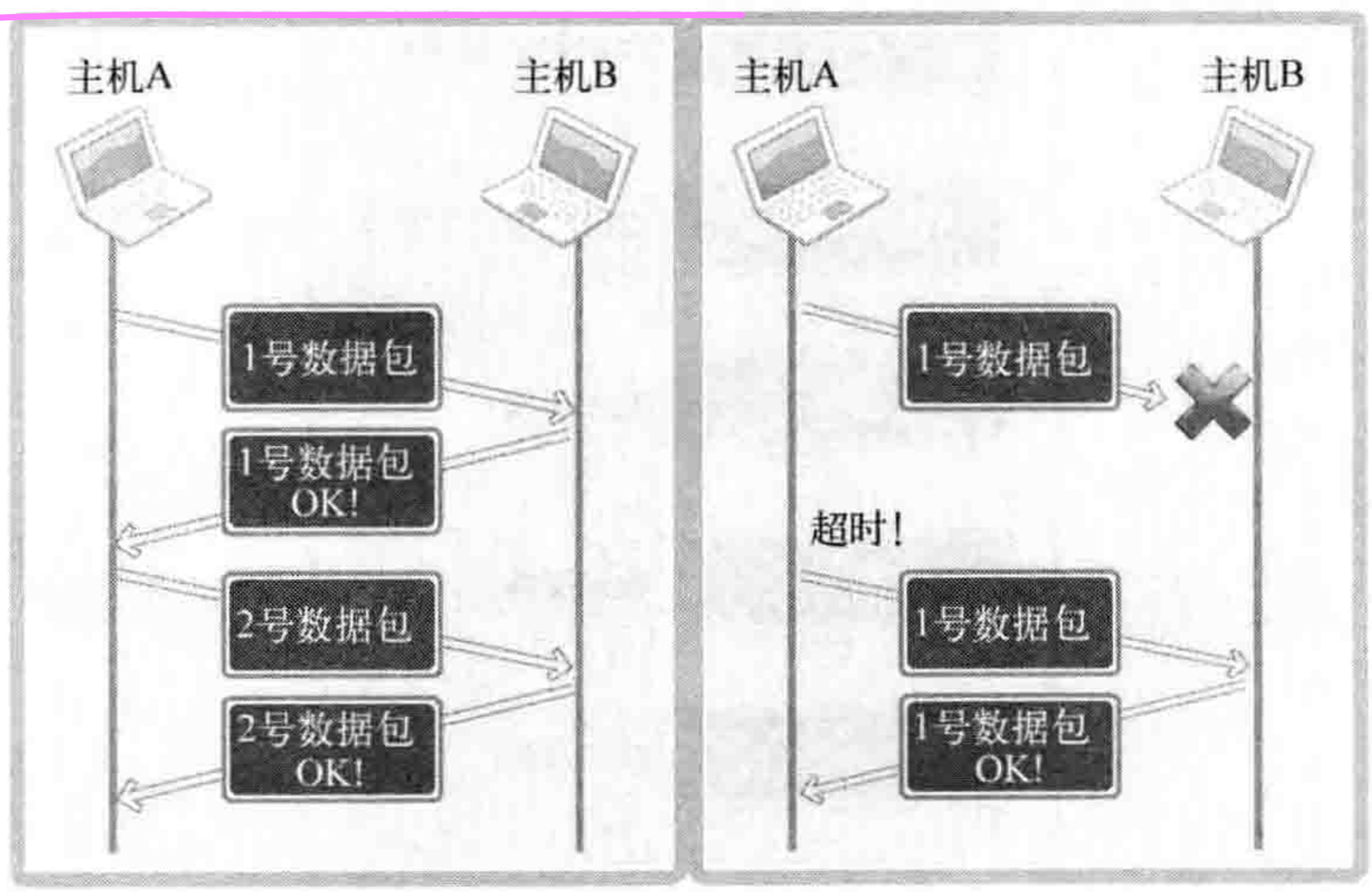

图4-5 传输控制协议

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。

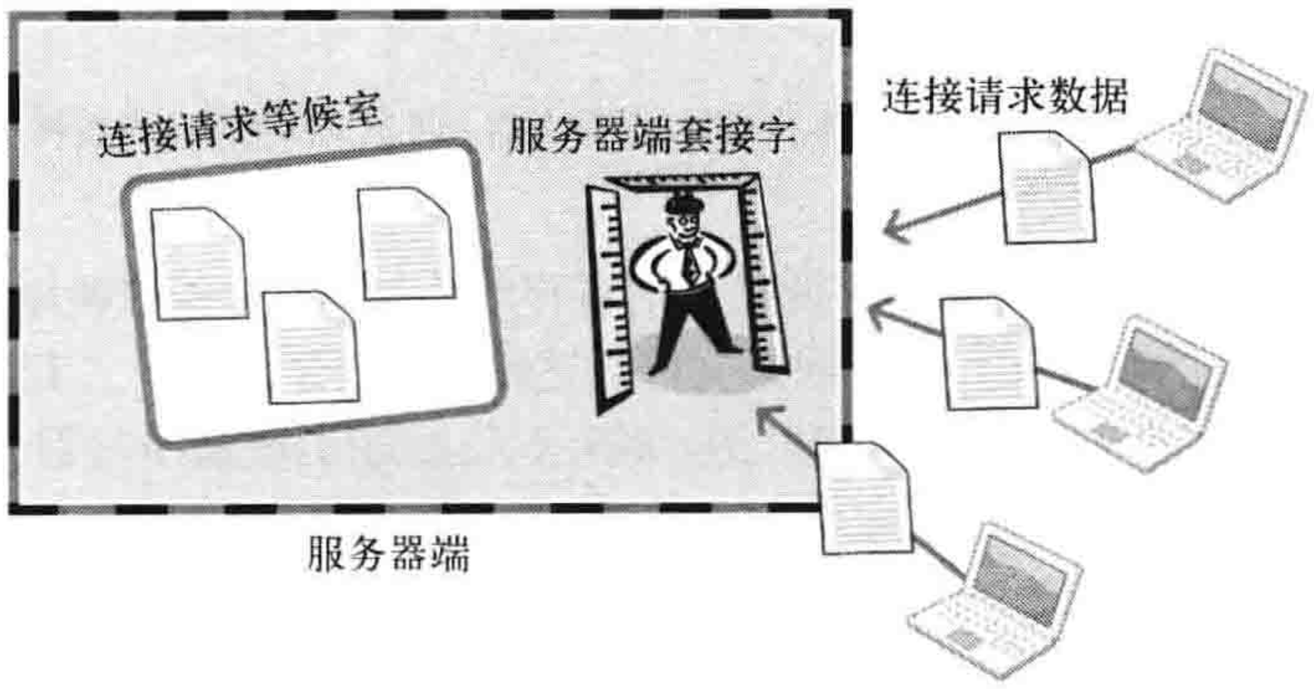

图4-7　等待连接请求状态

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。

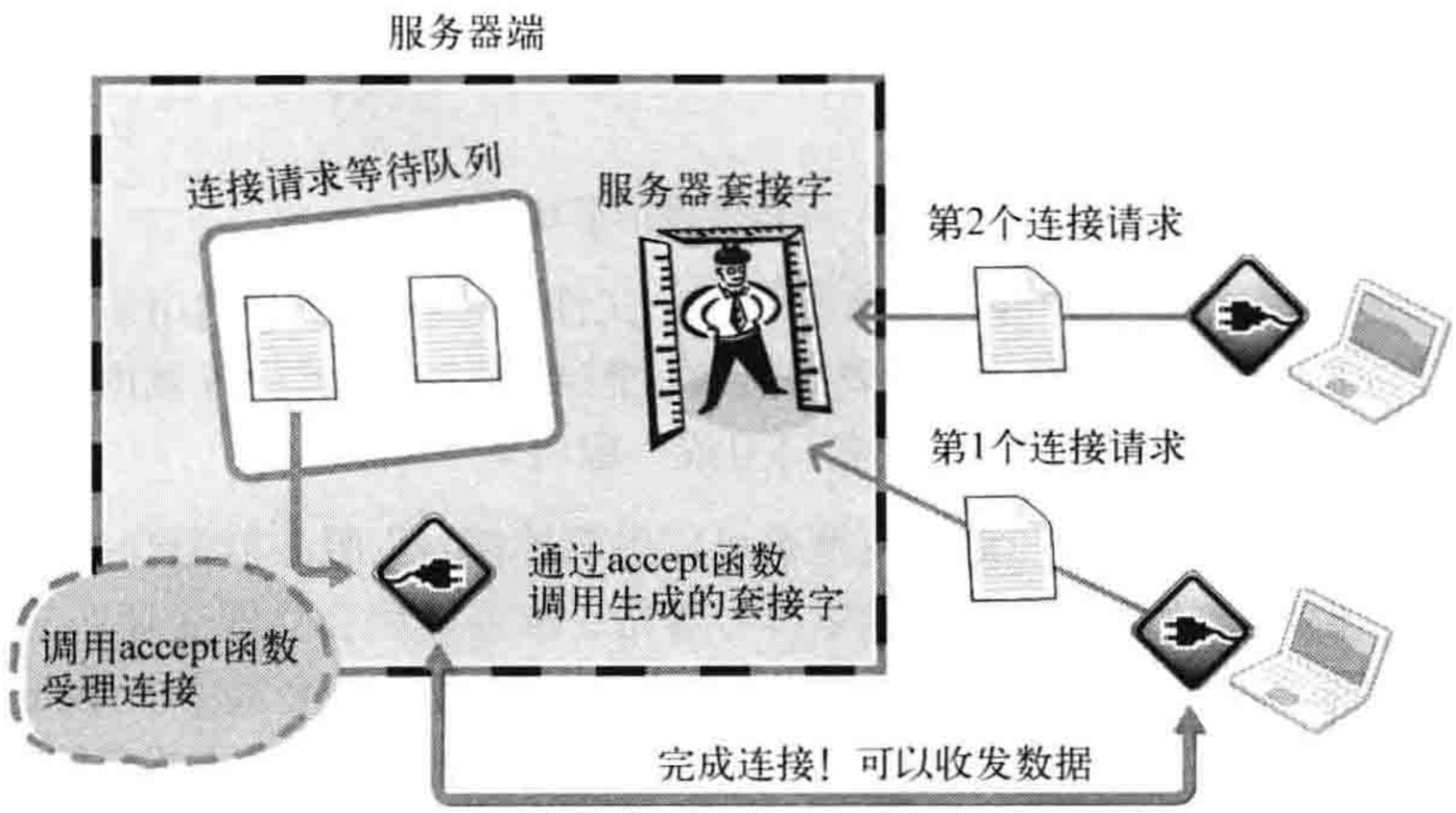

图4-8　受理连接请求状态

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。

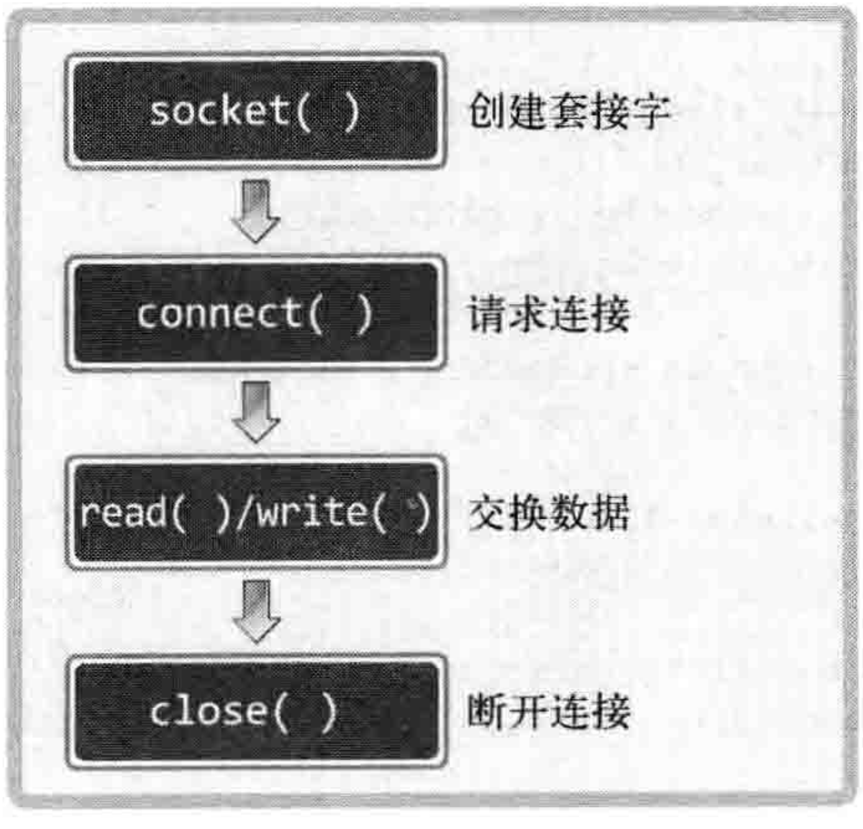

图4-9　TCP客户端函数调用顺序

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。

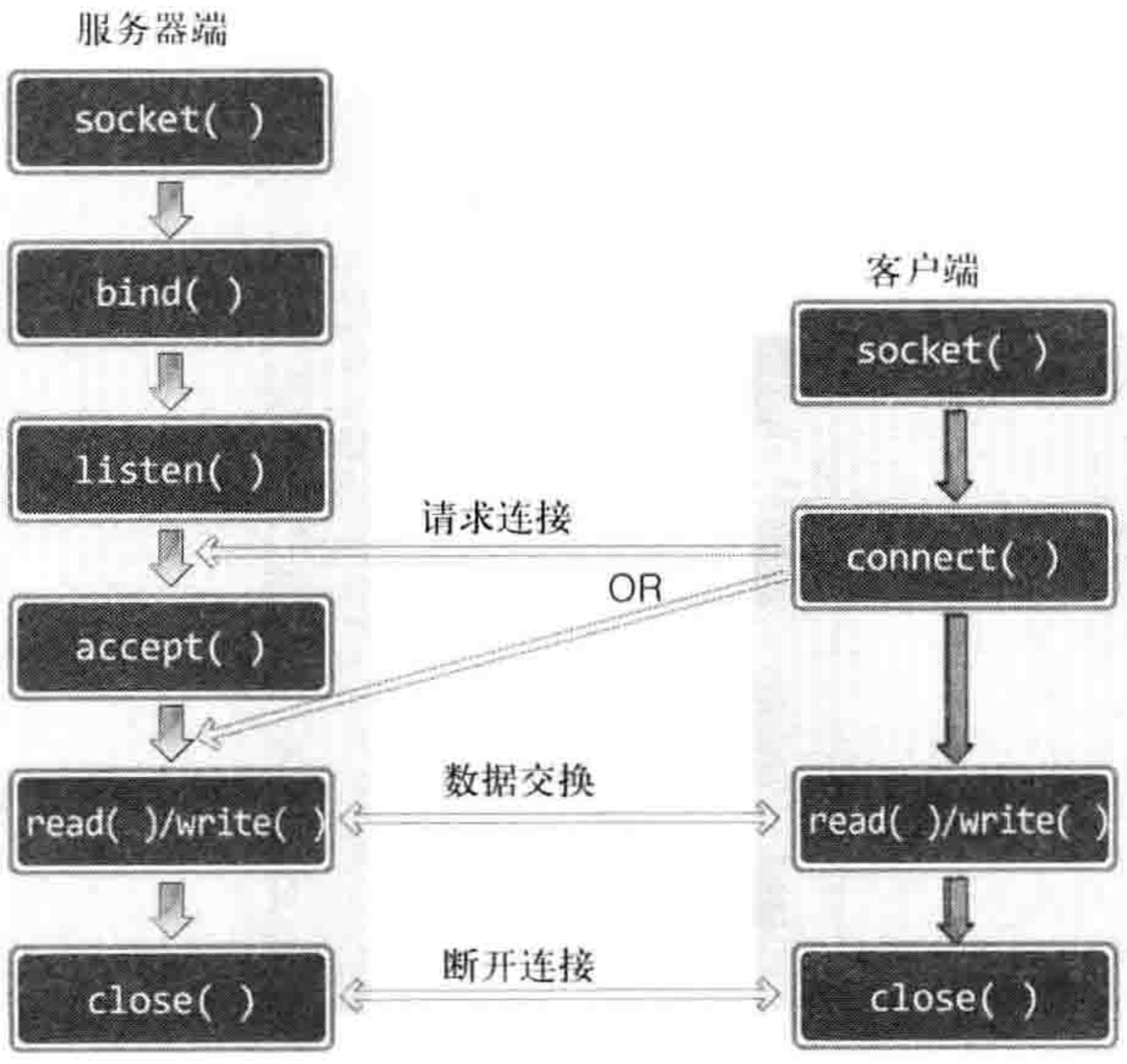

图4-10 函数调用关系

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。

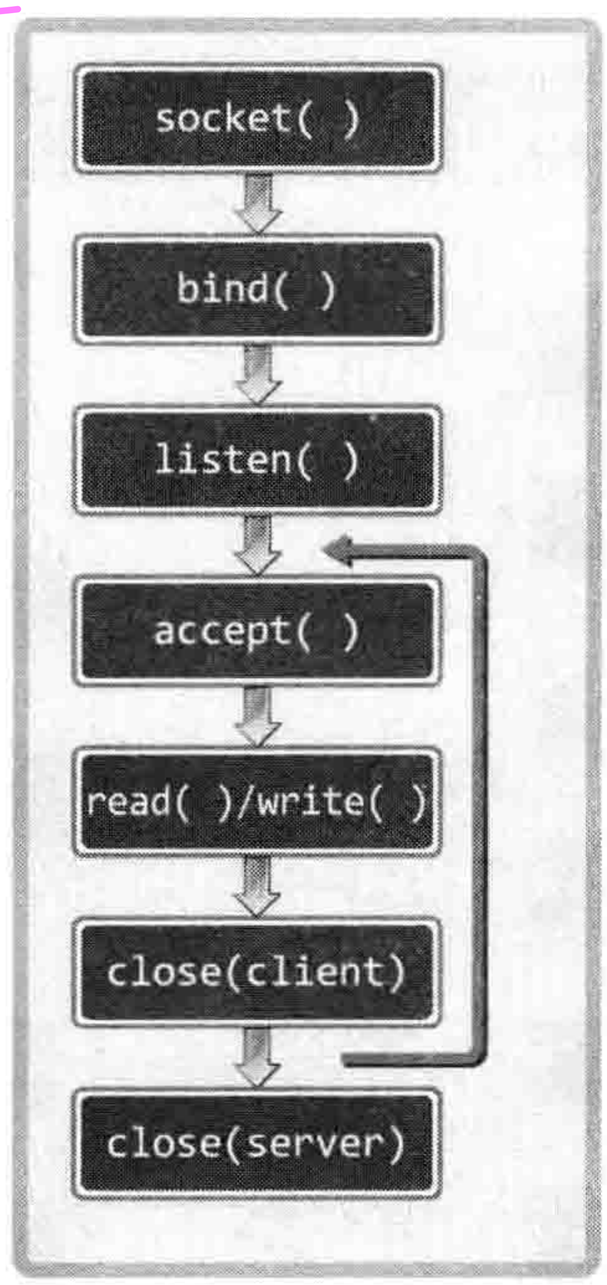

图4-11　迭代服务器端的函数调用顺序

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>
```