第二章　socket 编程原理

2.1　问题的引入

UNIX 系统的 I/O 命令集，是从 Maltics 和早期系统中的命令演变出来的，其模式为打开一读/写一关闭（open-write-read-close）。在一个用户进程进行 I/O 操作时，它首先调用“打开”获得对指定文件或设备的使用权，并返回称为文件描述符的整型数，以描述用户在打开的文件或设备上进行 I/O 操作的进程。然后这个用户进程多次调用“读/写”以传输数据。当所有的传输操作完成后，用户进程关闭调用，通知操作系统已经完成了对某对象的使用。

TCP/IP 协议被集成到 UNIX 内核中时，相当于在 UNIX 系统引入了一种新型的 I/O 操作。UNIX 用户进程与网络协议的交互作用比用户进程与传统的 I/O 设备相互作用复杂得多。首先，进行网络操作的两个进程在不同机器上，如何建立它们之间的联系？其次，网络协议存在多种，如何建立一种通用机制以支持多种协议？这些都是网络应用编程界面所要解决的问题。

在 UNIX 系统中，网络应用编程界面有两类：UNIX　BSD 的套接字（socket）和 UNIX System V 的 TLI。由于 Sun 公司采用了支持 TCP/IP 的 UNIX　BSD 操作系统，使 TCP/IP 的应用有更大的发展，其网络应用编程界面——套接字（socket）在网络软件中被广泛应用，至今已引进微机操作系统 DOS 和 Windows 系统中，成为开发网络应用软件的强有力工具，本章将要详细讨论这个问题。

2.2　套接字编程基本概念

在开始使用套接字编程之前，首先必须建立以下概念。

2.2.1　网间进程通信

进程通信的概念最初来源于单机系统。由于每个进程都在自己的地址范围内运行，为保证两个相互通信的进程之间既互不干扰又协调一致工作，操作系统为进程通信提供了相应设施，如 UNIX　BSD 中的管道（pipe）、命名管道（named pipe）和软中断信号（signal），UNIX system V 的消息（message）、共享存储区（shared memory）和信号量（semaphore)等，但都仅限于用在本机进程之间通信。网间进程通信要解决的是不同主机进程间的相互通信问题（可把同机进程通信看作是其中的特例）。为此，首先要解决的是网间进程标识问题。同一主机上，不同进程可用进程号（process ID）唯一标识。但在网络环境下，各主机独立分配的进程号不能唯一标识该进程。例如，主机 A 赋于某进程号 5，在 B 机中也可以存在 5 号进程，因此，“5 号进程”这句话就没有意义了。

其次，操作系统支持的网络协议众多，不同协议的工作方式不同，地址格式也不同。因此，网间进程通信还要解决多重协议的识别问题。

为了解决上述问题，TCP/IP 协议引入了下列几个概念。

端口

网络中可以被命名和寻址的通信端口，是操作系统可分配的一种资源。

按照 OSI 七层协议的描述，传输层与网络层在功能上的最大区别是传输层提供进程通信能力。从这个意义上讲，网络通信的最终地址就不仅仅是主机地址了，还包括可以描述进程的某种标识符。为此，TCP/IP 协议提出了协议端口（protocol port，简称端口）的概念，用于标识通信的进程。

端口是一种抽象的软件结构（包括一些数据结构和 I/O 缓冲区）。应用程序（即进程）通过系统调用与某端口建立连接（binding）后，传输层传给该端口的数据都被相应进程所接收，相应进程发给传输层的数据都通过该端口输出。在 TCP/IP 协议的实现中，端口操作类似于一般的 I/O 操作，进程获取一个端口，相当于获取本地唯一的 I/O 文件，可以用一般的读写原语访问之。

类似于文件描述符，每个端口都拥有一个叫端口号（port number）的整数型标识符，用于区别不同端口。由于 TCP/IP 传输层的两个协议 TCP 和 UDP 是完全独立的两个软件模块，因此各自的端口号也相互独立，如 TCP 有一个 255 号端口，UDP 也可以有一个 255 号端口，二者并不冲突。

端口号的分配是一个重要问题。有两种基本分配方式：第一种叫全局分配，这是一种集中控制方式，由一个公认的中央机构根据用户需要进行统一分配，并将结果公布于众。第二种是本地分配，又称动态连接，即进程需要访问传输层服务时，向本地操作系统提出申请，操作系统返回一个本地唯一的端口号，进程再通过合适的系统调用将自己与该端口号联系起来（绑扎）。TCP/IP 端口号的分配中综合了上述两种方式。TCP/IP 将端口号分为两部分，少量的作为保留端口，以全局方式分配给服务进程。因此，每一个标准服务器都拥有一个全局公认的端口（即周知口，well-known port），即使在不同机器上，其端口号也相同。剩余的为自由端口，以本地方式进行分配。TCP 和 UDP 均规定，小于 256 的端口号才能作保留端口。

地址

网络通信中通信的两个进程分别在不同的机器上。在互连网络中，两台机器可能位于不同的网络，这些网络通过网络互连设备（网关，网桥，路由器等）连接。因此需要三级寻址：

1. 某一主机可与多个网络相连，必须指定一特定网络地址；
2. 网络上每一台主机应有其唯一的地址；
3. 每一主机上的每一进程应有在该主机上的唯一标识符。

通常主机地址由网络 ID 和主机 ID 组成，在 TCP/IP 协议中用 32 位整数值表示；TCP 和 UDP 均使用 16 位端口号标识用户进程。

网络字节顺序

不同的计算机存放多字节值的顺序不同，有的机器在起始地址存放低位字节（低价先存），有的存高位字节（高价先存）。为保证数据的正确性，在网络协议中须指定网络字节顺序。TCP/IP 协议使用 16 位整数和 32 位整数的高价先存格式，它们均含在协议头文件中。

连接

两个进程间的通信链路称为连接。连接在内部表现为一些缓冲区和一组协议机制，在外部表现出比无连接高的可靠性。

半相关

综上所述，网络中用一个三元组可以在全局唯一标志一个进程：

（协议，本地地址，本地端口号）

这样一个三元组，叫做一个半相关（half-association），它指定连接的每半部分。

全相关

一个完整的网间进程通信需要由两个进程组成，并且只能使用同一种高层协议。也就是说，不可能通信的一端用 TCP 协议，而另一端用 UDP 协议。因此一个完整的网间通信需要一个五元组来标识：

（协议，本地地址，本地端口号，远地地址，远地端口号）

这样一个五元组，叫做一个相关（association），即两个协议相同的半相关才能组合成一个合适的相关，或完全指定组成一连接。

2.2.2 服务方式

在网络分层结构中，各层之间是严格单向依赖的，各层次的分工和协作集中体现在相邻层之间的界面上。“服务”是描述相邻层之间关系的抽象概念，即网络中各层向紧邻上层提供的一组操作。下层是服务提供者，上层是请求服务的用户。服务的表现形式是原语（primitive），如系统调用或库函数。系统调用是操作系统内核向网络应用程序或高层协议提供的服务原语。网络中的 n 层总要向 n+1 层提供比 n-1 层更完备的服务，否则 n 层就没有存在的价值。

在 OSI 的术语中，网络层及其以下各层又称为通信子网，只提供点到点通信，没有程序或进程的概念。而传输层实现的是“端到端”通信，引进网间进程通信概念，同时也要解决差错控制，流量控制，数据排序（报文排序），连接管理等问题，为此提供不同的服务方式：

面向连接（虚电路）或无连接

面向连接服务是电话系统服务模式的抽象，即每一次完整的数据传输都要经过建立连接，使用连接，终止连接的过程。在数据传输过程中，各数据分组不携带目的地址，而使用连接号（connect ID）。本质上，连接是一个管道，收发数据不但顺序一致，而且内容相同。TCP 协议提供面向连接的虚电路。

无连接服务是邮政系统服务的抽象，每个分组都携带完整的目的地址，各分组在系统中独立传送。无连接服务不能保证分组的先后顺序，不进行分组出错的恢复与重传，不保证传输的可靠性。UDP 协议提供无连接的数据报服务。

下面给出这两种服务的类型及应用中的例子：

服务类型	服　　务	例　　子
面向连接	可靠的报文流	文件传输（FTP）
	可靠的字节流	远程登录（Telnet）
	不可靠的连接	数字话音
无连接	不可靠的数据报	电子邮件（E-mail）
	有确认的数据报	电子邮件中的挂号信
	请求－应答	网络数据库查询

顺序

在网络传输中，两个连续报文在端－端通信中可能经过不同路径，这样到达目的地时的顺序可能会与发送时不同。“顺序”是指接收数据顺序与发送数据顺序相同。TCP 协议提供这项服务。

差错控制

保证应用程序接收的数据无差错的一种机制。检查差错的方法一般是采用检验“检查和（Checksum）”的方法。而保证传送无差错的方法是双方采用确认应答技术。TCP 协议提供这项服务。

流控制

在数据传输过程中控制数据传输速率的一种机制，以保证数据不被丢失。TCP 协议提供这项服务。

字节流

字节流方式指的是仅把传输中的报文看作是一个字节序列，不提供数据流的任何边界。TCP 协议提供字节流服务。

报文

接收方要保存发送方的报文边界。UDP 协议提供报文服务。

全双工/半双工

端－端间数据同时以两个方向/一个方向传送。

缓存/带外数据

在字节流服务中，由于没有报文边界，用户进程在某一时刻可以读或写任意数量的字节。为保证传输正确或采用有流控制的协议时，都要进行**缓存**。但对某些特殊的需求，如交互式应用程序，又会要求取消这种缓存。

在数据传送过程中，希望不通过常规传输方式传送给用户以便及时处理的某一类信息，如 UNIX 系统的中断键（Delete 或 Control-c）、终端流控制符（Control-s 和 Control-q），称为带外数据。逻辑上看，好象用户进程使用了一个独立的通道传输这些数据。该通道与每对连接的流相联系。由于 Berkeley Software Distribution 中对带外数据的实现与 RFC 1122 中规定的 Host Agreement 不一致,为了将互操作中的问题减到最小，应用程序编写者除非与现有服务互操作时要求带外数据外，最好不使用它。

2.2.3 客户/服务器模式

在 TCP/IP 网络应用中，通信的两个进程间相互作用的主要模式是客户/服务器模式（Client/Server model），即客户向服务器发出服务请求，服务器接收到请求后，提供相应的服务。客户/服务器模式的建立基于以下两点：首先，建立网络的起因是网络中软硬件资源、运算能力和信息不均等，需要共享，从而造就拥有众多资源的主机提供服务，资源较少的客户请求服务这一非对等作用。其次，网间进程通信完全是异步的，相互通信的进程间既不存在父子关系，又不共享内存缓冲区，因此需要一种机制为希望通信的进程间建立联系，为二者的数据交换提供同步，这就是基于客户/服务器模式的 TCP/IP。

客户/服务器模式在操作过程中采取的是主动请求方式：

首先服务器方要先启动，并根据请求提供相应服务：

1. 1. 打开一通信通道并告知本地主机，它愿意在某一公认地址上（周知口，如 FTP 为 21）接收客户请求；
2. 2. 等待客户请求到达该端口；
3. 3. 接收到重复服务请求，处理该请求并发送应答信号。接收到并发服务请求，要激活一新进程来处理这个客户请求（如 UNIX 系统中用 fork、exec）。新进程处理此客户请求，并不需要对其它请求作出应答。服务完成后，关闭此新进程与客户的通信链路，并终止。
4. 4. 返回第二步，等待另一客户请求。
5. 5. 关闭服务器

客户方：

1. 1. 打开一通信通道，并连接到服务器所在主机的特定端口；
2. 2. 向服务器发服务请求报文，等待并接收应答；继续提出请求.....
3. 3. 请求结束后关闭通信通道并终止。

从上面所描述过程可知：

1. 1. 客户与服务器进程的作用是非对称的，因此编码不同。
2. 2. 服务进程一般是先于客户请求而启动的。只要系统运行，该服务进程一直存在，直到正

常或强迫终止。

2.2.4 套接字类型

TCP/IP 的 socket 提供下列三种类型套接字。

流式套接字（SOCK_STREAM）

提供了一个面向连接、可靠的数据传输服务，数据无差错、无重复地发送，且按发送顺序接收。内设流量控制，避免数据流超限；数据被看作是字节流，无长度限制。文件传送协议（FTP）即使用流式套接字。

数据报式套接字（SOCK_DGRAM）

提供了一个无连接服务。数据包以独立包形式被发送，不提供无错保证，数据可能丢失或重复，并且接收顺序混乱。网络文件系统（NFS）使用数据报式套接字。

原始式套接字（SOCK_RAW）

该接口允许对较低层协议，如 IP、ICMP 直接访问。常用于检验新的协议实现或访问现有服务中配置的新设备。

2.3 基本套接字系统调用

为了更好地说明套接字编程原理，下面给出几个基本套接字系统调用说明。

2.3.1 创建套接字——socket()

应用程序在使用套接字前，首先必须拥有一个套接字，系统调用 socket()向应用程序提供创建套接字的手段，其调用格式如下：

SOCKET PASCAL FAR socket(int af, int type, int protocol);

该调用要接收三个参数：af、type、protocol。参数 af 指定通信发生的区域，UNIX 系统支持的地址族有：AF_UNIX、AF_INET、AF_NS 等，而 DOS、WINDOWS 中仅支持 AF_INET，它是网际网区域。因此，地址族与协议族相同。参数 type 描述要建立的套接字的类型。参数 protocol 说明该套接字使用的特定协议，如果调用者不希望特别指定使用的协议，则置为 0，使用默认的连接模式。根据这三个参数建立一个套接字，并将相应的资源分配给它，同时返回一个整型套接字号。因此，socket()系统调用实际上指定了相关五元组中的“协议”这一元。

有关 socket()的详细描述参看 5.2.23。

2.3.2 指定本地地址——bind()

当一个套接字用 socket()创建后，存在一个名字空间(地址族),但它没有被命名。bind()将套接字地址（包括本地主机地址和本地端口地址）与所创建的套接字号联系起来，即将名字赋予套接字，以指定本地半相关。其调用格式如下：

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

参数 s 是由 socket()调用返回的并且未作连接的套接字描述符(套接字号)。参数 name 是赋给套接字 s 的本地地址（名字），其长度可变，结构随通信域的不同而不同。namelen 表明了 name 的长度。

如果没有错误发生，bind()返回 0。否则返回值 SOCKET_ERROR。

地址在建立套接字通信过程中起着重要作用，作为一个网络应用程序设计者对套接字地址结构必须有明确认识。例如，UNIX BSD 有一组描述套接字地址的数据结构，其中使用 TCP/IP 协议的地址结构为：

```
struct sockaddr_in{
    short sin_family;              /*AF_INET*/
    u_short sin_port;              /*16 位端口号，网络字节顺序*/
    struct in_addr sin_addr;       /*32 位 IP 地址，网络字节顺序*/
    char sin_zero[8];              /*保留*/
}
```

有关 bind()的详细描述参看 5.2.2。

2.3.3 建立套接字连接——connect()与 accept()

这两个系统调用用于完成一个完整相关的建立，其中 connect()用于建立连接。无连接的套接字进程也可以调用 connect()，但这时在进程之间没有实际的报文交换，调用将从本地操作系统直接返回。这样做的优点是程序员不必为每一数据指定目的地址，而且如果收到的一个数据报，其目的端口未与任何套接字建立“连接”，便能判断该端口不可操作。而 accept()用于使服务器等待来自某客户进程的实际连接。

connect()的调用格式如下：

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

参数 s 是欲建立连接的本地套接字描述符。参数 name 指出说明对方套接字地址结构的指针。对方套接字地址长度由 namelen 说明。

如果没有错误发生，connect()返回 0。否则返回值 SOCKET_ERROR。在面向连接的协议中，该调用导致本地系统和外部系统之间连接实际建立。

由于地址族总被包含在套接字地址结构的前两个字节中，并通过 socket()调用与某个协议族相关。因此 bind()和 connect()无须协议作为参数。

有关 connect()的详细描述参看 5.2.4。

accept()的调用格式如下：

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

参数 s 为本地套接字描述符，在用做 accept()调用的参数前应该先调用过 listen()。addr 指向客户方套接字地址结构的指针，用来接收连接实体的地址。addr 的确切格式由套接字创建时建立的地址族决定。addrlen 为客户方套接字地址的长度（字节数）。如果没有错误发生，accept()返回一个 SOCKET 类型的值，表示接收到的套接字的描述符。否则返回值 INVALID_SOCKET。

accept()用于面向连接服务器。参数 addr 和 addrlen 存放客户方的地址信息。调用前，参数 addr 指向一个初始值为空的地址结构，而 addrlen 的初始值为 0；调用 accept()后，服务器等待从编号为 s 的套接字上接受客户连接请求，而连接请求是由客户方的 connect()调用发出的。当有连接请求到达时，accept()调用将请求连接队列上的第一个客户方套接字地址及长度放入 addr 和 addrlen，并创建一个与 s 有相同特性的新套接字号。新的套接字可用于处理服务器并发请求。

有关 accept()的详细描述参看 5.2.1。

四个套接字系统调用，socket()、bind()、connect()、accept()，可以完成一个完全五元相关的建立。socket()指定五元组中的协议元，它的用法与是否为客户或服务器、是否面向连接无关。bind()指定五元组中的本地二元，即本地主机地址和端口号，其用法与是否面向连接有关：在服务器方，无论是否面向连接，均要调用 bind()；在客户方，若采用面向连接，则可以不调用 bind()，而通过 connect()自动完成。若采用无连接，客户方必须使用 bind()以获得一个唯一的地址。

以上讨论仅对客户/服务器模式而言，实际上套接字的使用是非常灵活的，唯一需遵循的原则是进程通信之前，必须建立完整的相关。

2.3.4 监听连接——listen()

此调用用于面向连接服务器，表明它愿意接收连接。listen()需在 accept()之前调用，其调用格式如下：

int PASCAL FAR listen(SOCKET s, int backlog);

参数 s 标识一个本地已建立、尚未连接的套接字号，服务器愿意从它上面接收请求。backlog 表示请求连接队列的最大长度，用于限制排队请求的个数，目前允许的最大值为 5。如果没有错误发生，listen()返回 0。否则它返回 SOCKET_ERROR。

listen()在执行调用过程中可为没有调用过 bind()的套接字 s 完成所必须的连接，并建立长度为 backlog 的请求连接队列。

调用 listen()是服务器接收一个连接请求的四个步骤中的第三步。它在调用 socket()分配一个流套接字，且调用 bind()给 s 赋于一个名字之后调用，而且一定要在 accept()之前调用。

有关 listen()的详细描述参看 5.2.13。

2.2.3 节中提到在客户/服务器模式中，有两种类型的服务：重复服务和并发服务。accept()调用为实现并发服务提供了极大方便，因为它要返回一个新的套接字号，其典型结构为：

```
int initsockid, newsockid;
if ((initsockid = socket(....)) < 0)
    error("can't create socket");
if (bind(initsockid,....) < 0)
    error("bind error");
if (listen(initsockid , 5) < 0)
    error("listen error");
for (;;) {
    newsockid = accept(initsockid, ...)        /* 阻塞 */
    if (newsockid < 0)
        error("accept error");
    if (fork() == 0){                          /* 子进程 */
        closesocket(initsockid);
        do(newsockid);                         /* 处理请求 */
        exit(0);
    }
    closesocket(newsockid);                    /* 父进程 */
}
```

这段程序执行的结果是 newsockid 与客户的套接字建立相关，子进程启动后，关闭继承下来的主服务器的 initsockid,并利用新的 newsockid 与客户通信。主服务器的 initsockid 可继续等待新的客户连接请求。由于在 Unix 等抢先多任务系统中，在系统调度下，多个进程可以同时进行。因此，使用并发服务器可以使服务器进程在同一时间可以有多个子进程和不同的客户程序连接、通信。在客户程序看来，服务器可以同时并发地处理多个客户的请求，这就是并发服务器名称的来由。

面向连接服务器也可以是重复服务器，其结构如下：

```
int initsockid, newsockid;
if ((initsockid = socket(....))<0)
        error("can't create socket");
if (bind(initsockid,....)<0)
        error("bind error");
if (listen(initsockid,5)<0)
        error("listen error");
for (;;) {
        newsockid = accept(initsockid, ...)        /* 阻塞 */
        if (newsockid < 0)
                error("accept error");
        do(newsockid);                          /* 处理请求 */
        closesocket(newsockid);
}
```

重复服务器在一个时间只能和一个客户程序建立连接，它对多个客户程序的处理是采用循环的方式重复进行，因此叫重复服务器。并发服务器和重复服务器各有利弊：并发服务器可以改善客户程序的响应速度，但它增加了系统调度的开销；重复服务器正好与其相反，因此用户在决定是使用并发服务器还是重复服务器时，要根据应用的实际情况来定。

2.3.5 数据传输——send()与 recv()

当一个连接建立以后，就可以传输数据了。常用的系统调用有 send()和 recv()。

send()调用用于在参数 s 指定的已连接的数据报或流套接字上发送输出数据，格式如下：

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

参数 s 为已连接的本地套接字描述符。buf 指向存有发送数据的缓冲区的指针，其长度由 len 指定。flags 指定传输控制方式，如是否发送带外数据等。如果没有错误发生，send()返回总共发送的字节数。否则它返回 SOCKET_ERROR。

有关 send()的详细描述参看 5.2.19。

recv()调用用于在参数 s 指定的已连接的数据报或流套接字上接收输入数据，格式如下：

int PASCAL FAR recv(SOCKET s, char FAR *buf, int len, int flags);

参数 s 为已连接的套接字描述符。buf 指向接收输入数据缓冲区的指针，其长度由 len 指定。flags 指定传输控制方式，如是否接收带外数据等。如果没有错误发生，recv()返回总共接收的字节数。如果连接被关闭，返回 0。否则它返回 SOCKET_ERROR。

有关 recv()的详细描述参看 5.2.16。

2.3.6 输入/输出多路复用——select()

select()调用用来检测一个或多个套接字的状态。对每一个套接字来说，这个调用可以请求读、写或错误状态方面的信息。请求给定状态的套接字集合由一个 fd_set 结构指示。在返回时，此结构被更新，以反映那些满足特定条件的套接字的子集，同时， select()调用返回满足条件的套接字的数目，其调用格式如下：

int PASCAL FAR select(int nfds, fd_set FAR * readfds, fd_set FAR * writefds, fd_set FAR * exceptfds, const struct timeval FAR * timeout);

参数 nfds 指明被检查的套接字描述符的值域，此变量一般被忽略。

参数 readfds 指向要做读检测的套接字描述符集合的指针，调用者希望从中读取数据。参数 writefds 指向要做写检测的套接字描述符集合的指针。exceptfds 指向要检测是否出错的套接字描述符集合的指针。timeout 指向 select()函数等待的最大时间，如果设为 NULL 则为阻塞操作。select()返回包含在 fd_set 结构中已准备好的套接字描述符的总数目，或者是发生错误则返回 SOCKET_ERROR。

有关 select()的详细描述参看 5.2.18。

2.3.7 关闭套接字——closesocket()

closesocket()关闭套接字 s，并释放分配给该套接字的资源；如果 s 涉及一个打开的 TCP 连接，则该连接被释放。closesocket()的调用格式如下：

BOOL PASCAL FAR closesocket(SOCKET s);

参数 s 待关闭的套接字描述符。如果没有错误发生，closesocket()返回 0。否则返回值 SOCKET_ERROR。

有关 closesocket()的详细描述参看 5.2.3。

2.4　典型套接字调用过程举例

如前所述，TCP/IP 协议的应用一般采用客户/服务器模式，因此在实际应用中，必须有客户和服务器两个进程，并且首先启动服务器，其系统调用时序图如下。

面向连接的协议（如 TCP）的套接字系统调用如图 2.1 所示：

服务器必须首先启动，直到它执行完 accept()调用，进入等待状态后，方能接收客户请求。假如客户在此前启动，则 connect()将返回出错代码，连接不成功。

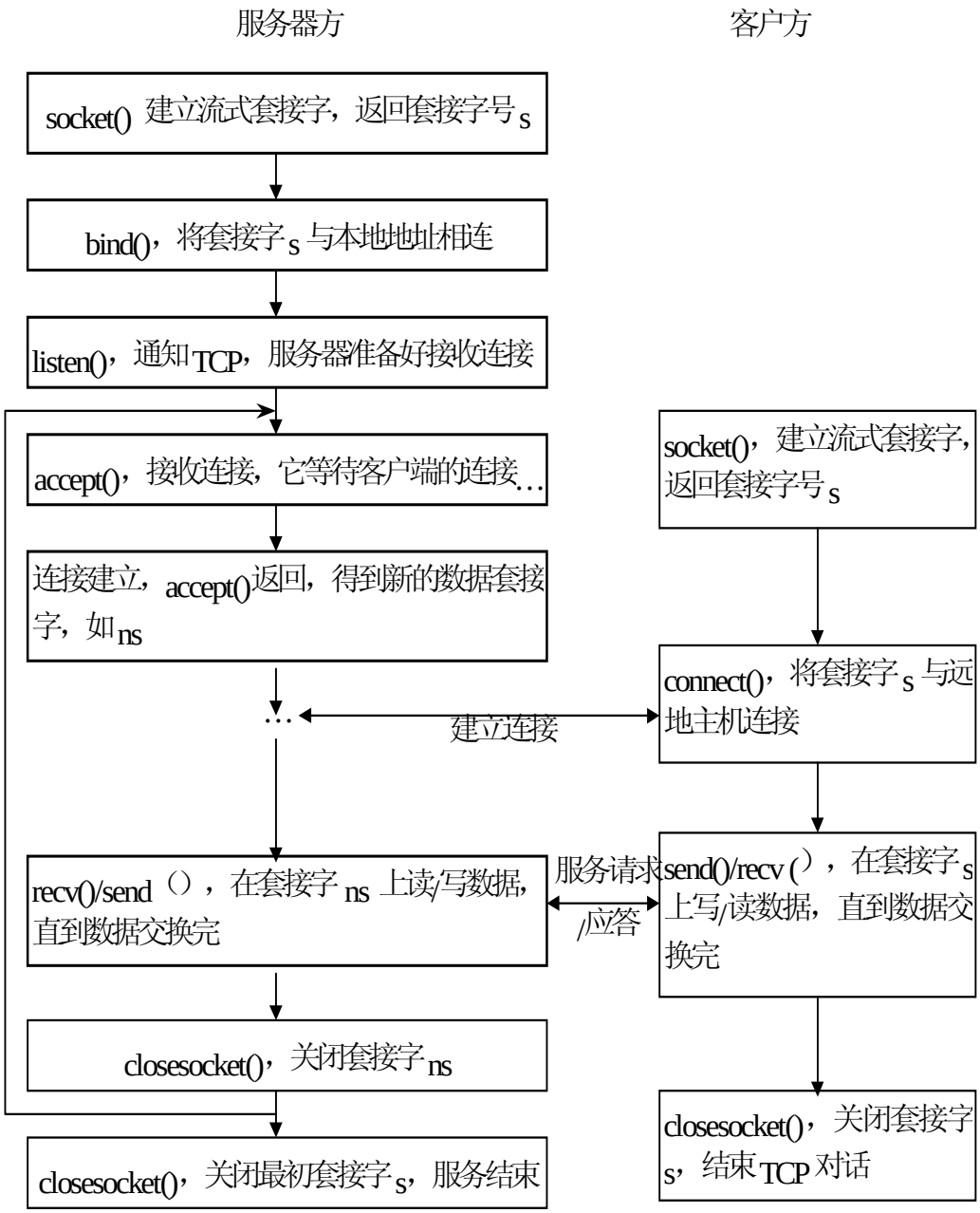

图 2.1　面向连接的套接字系统调用时序图

无连接协议的套接字调用如图 2.2 所示：

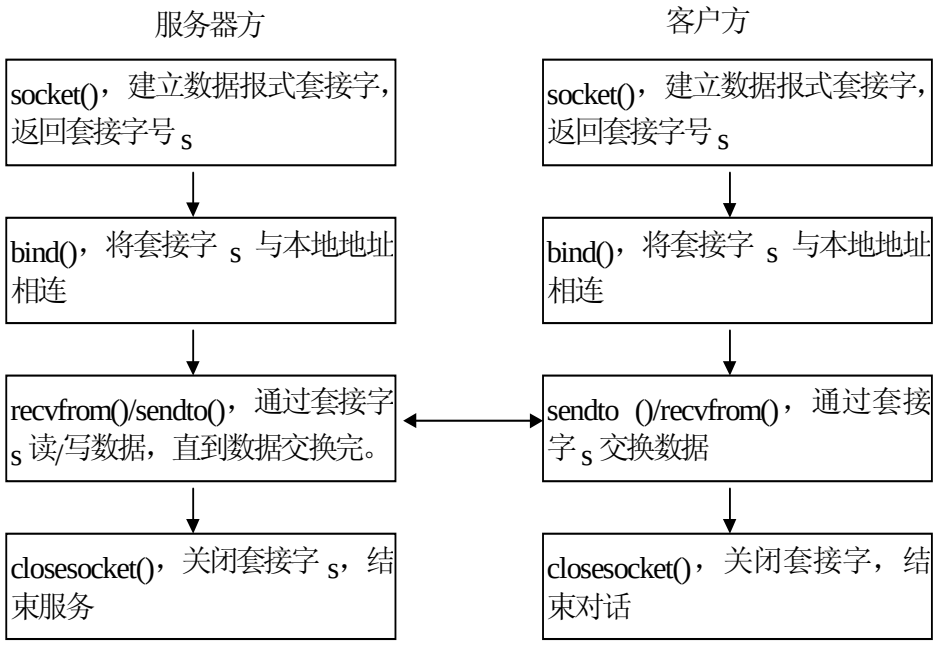

图 2.2　无连接协议的套接字调用时序图

无连接服务器也必须先启动，否则客户请求传不到服务进程。无连接客户不调用 connect()。因此在数据发送之前，客户与服务器之间尚未建立完全相关，但各自通过 socket()和 bind()建立了半相关。发送数据时，发送方除指定本地套接字号外，还需指定接收方套接字号，从而在数据收发过程中动态地建立了全相关。

实例

本实例使用面向连接协议的客户/服务器模式，其流程如图 2.3 所示：

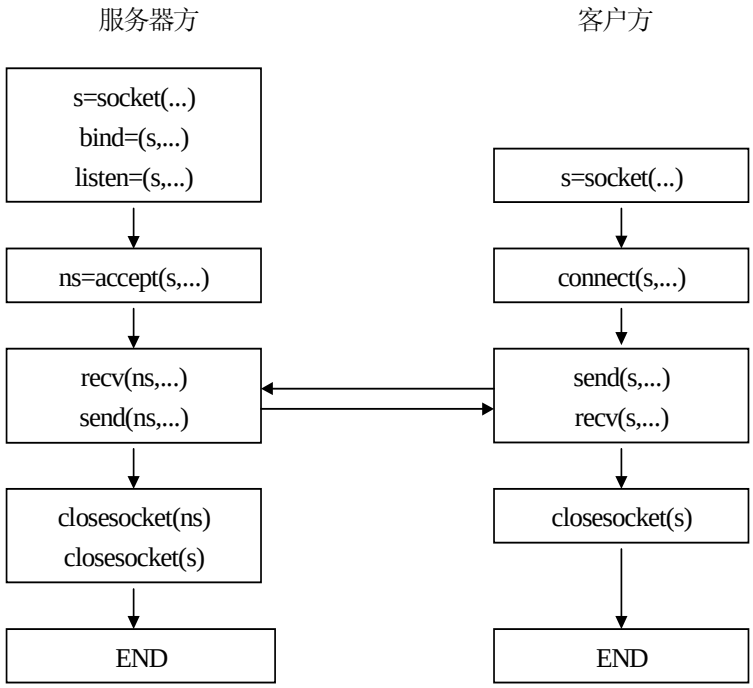

图 2.3　面向连接的应用程序流程图

服务器方程序：

```
/* File Name: streams.c */
#include <winsock.h>
#include <stdio.h>
#define TRUE 1
/*  这个程序建立一个套接字，然后开始无限循环；每当它通过循环接收到一个连接，则打印出一个信
```

息。当连接断开，或接收到终止信息，则此连接结束，程序再接收一个新的连接。命令行的格式是：
streams */

```
main( )
{
    int sock, length;
    struct sockaddr_in server;
    struct sockaddr tcpaddr;
    int msgsock;
    char buf[1024];
    int rval, len;

    /*  建立套接字  */
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        perror("opening stream socket");
        exit(1);
    }

    /*  使用任意端口命名套接字  */
    server.sin_family = AF_INET;
    server.sin_port = INADDR_ANY;
    if (bind(sock, (struct sockaddr *)&server, sizeof(server)) < 0) {
        perror("binding stream socket");
        exit(1);
    }

    /*  找出指定的端口号并打印出来  */
    length = sizeof(server);
    if (getsockname(sock, (struct sockaddr *)&server, &length) < 0) {
        perror("getting socket name");
        exit(1);
    }
    printf("socket port #%d\n", ntohs(server.sin_port));

    /*  开始接收连接  */
    listen(sock, 5);
    len = sizeof(struct sockaddr);
    do {
        msgsock = accept(sock, (struct sockaddr *)&tcpaddr, (int *)&len);
        if (msgsock == -1)
            perror("accept");
        else do{
            memset(buf, 0, sizeof(buf));
            if ((rval = recv(msgsock, buf, 1024)) < 0)
                perror("reading stream message");
            if (rval == 0)
                printf("ending connection \n");
            else
                printf("-->%s\n", buf);
        }while (rval != 0);
        closesocket(msgsock);
    } while (TRUE);

    /*  因为这个程序已经有了一个无限循环，所以套接字"sock"从来不显式关闭。然而，当进程被杀
死或正常终止时，所有套接字都将自动地被关闭。*/
    exit(0);
}
```

客户方程序：

```
/* File Name: streamc.c */
```

```
#include <winsock.h>
#include <stdio.h>
#define DATA "half a league, half a league ..."
/* 这个程序建立套接字，然后与命令行给出的套接字连接；连接结束时，在连接上发送
   一个消息，然后关闭套接字。命令行的格式是：streamc 主机名 端口号
   端口号要与服务器程序的端口号相同 */
main(argc, argv)
int argc;
char *argv[ ];
{
    int sock;
    struct sockaddr_in server;
    struct hostent *hp, *gethostbyname( );
    char buf[1024];

    /* 建立套接字 */
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        perror("opening stream socket");
        exit(1);
    }

    /* 使用命令行中指定的名字连接套接字 */
    server.sin_family = AF_INET;
    hp = gethostbyname(argv[1]);
    if (hp == 0) {
        fprintf(stderr, "%s: unknown host \n", argv[1]);
        exit(2);
    }
    memcpy((char*)&server.sin_addr, (char*)hp->h_addr, hp->h_length);
    sever.sin_port = htons(atoi(argv[2]));

    if (connect(sock, (struct sockaddr*)&server, sizeof(server)) < 0) {
        perror("connecting stream socket");
        exit(3);
    }

    if (send(sock, DATA, sizeof(DATA)) < 0)
        perror("sending on stream socket");
    closesocket(sock);
    exit(0);
}
```

2.5 一个通用的实例程序

在上一节中，我们介绍了一个简单的 socket 程序实例。从这个例子我们可以看出，使用 socket 编程几乎有一个模式，即所有的程序几乎毫无例外地按相同的顺序调用相同的函数。因此我们可以设想，设计一个中间层，它向上提供几个简单的函数，程序只要调用这几个函数就可以实现普通情况下的数据传输，程序设计者不必太多地关心 socket 程序设计的细节。

本节我们将介绍一个通用的网络程序接口，它向上层提供几个简单的函数，程序设计者只要使用这几个函数就可以完成绝大多数情况下的网络数据传输。这些函数将 socket 编程和上层隔离开来，它使用面向连接的流式套接字，采用非阻塞的工作机制，程序只要调用这些函数查询网络消息并作出相应的响应即可。这些函数包括：

- InitSocketsStruct：初始化 socket 结构，获取服务端口号。客户程序使用。
- InitPassiveSock：初始化 socket 结构，获取服务端口号，建立主套接字。服务器程序使用。
- CloseMainSock：关闭主套接字。服务器程序使用。
- CreateConnection：建立连接。客户程序使用。
- AcceptConnection：接收连接。服务器程序使用。
- CloseConnection：关闭连接。
- QuerySocketsMsg：查询套接字消息。

- SendPacket：发送数据。
- RecvPacket：接收数据。

2.5.1 头文件

```
/* File Name:     tcpsock.h */
/*  头文件包括 socket 程序经常用到的系统头文件（本例中给出的是 SCO Unix 下的头文件，其它版本的 Unix 的头文件可能略有不同），并定义了我们自己的两个数据结构及其实例变量，以及我们提供的函数说明。*/
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <sys/tape.h>
#include <sys/signal.h>
#include <sys/errno.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/select.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/netinet/in.h>
#include <sys/netinet/tcp.h>
#include <arpa/inet.h>
#include <netdb.h>

typedef struct SocketsMsg{  /*  套接字消息结构  */
      int AcceptNum;          /*  指示是否有外来连接等待接收  */
      int ReadNum;                /*  有外来数据等待读取的连接数  */
      int ReadQueue[32];          /*  有外来数据等待读取的连接队列  */
      int WriteNum;               /*  可以发送数据的连接数  */
      int WriteQueue[32];         /*  可以发送数据的连接队列  */
      int ExceptNum;          /*  有例外的连接数  */
      int ExceptQueue[32];        /*  有例外的连接队列  */
} SocketsMsg;

typedef struct Sockets {            /*  套接字结构  */
      int DaemonSock;         /*  主套接字  */
      int SockNum;                /*  数据套接字数目  */
      int Sockets[64];        /*  数据套接字数组  */
      fd_set readfds, writefds, exceptfds; /*  要被检测的可读、可写、例外的套接字集合  */
      int Port;                   /*  端口号  */
} Sockets;

Sockets Mysock;             /*  全局变量  */
SocketsMsg SockMsg;

int InitSocketsStruct(char * servicename) ;
int InitPassiveSock(char * servicename) ;
void CloseMainSock();
int CreateConnection(struct in_addr *sin_addr);
int AcceptConnection(struct in_addr *IPaddr);
int CloseConnection(int Sockno);
int QuerySocketsMsg();
int SendPacket(int Sockno, void *buf, int len);
int RecvPacket(int Sockno, void *buf, int size);
```

2.5.2 函数源文件

```
/* File Name:     tcpsock.c */
/*  本文件给出九个函数的源代码，其中部分地方给出中文注释  */
#include "tcpsock.h"

int InitSocketsStruct(char * servicename)
/* Initialize Sockets structure. If succeed then return 1, else return error code (<0) */
/*  此函数用于只需要主动套接字的客户程序，它用来获取服务信息。服务的定义
    在/etc/services 文件中  */
{
       struct servent *servrec;
       struct sockaddr_in serv_addr;

       if ((servrec = getservbyname(servicename, "tcp")) == NULL) {
             return(-1);
       }
       bzero((char *)&Mysock, sizeof(Sockets));
```

```
        Mysock.Port = servrec->s_port;    /* Service Port in Network Byte Order */
        return(1);
}

int InitPassiveSock(char * servicename)
/* Initialize Passive Socket. If succeed then return 1, else return error code (<0) */
/*  此函数用于需要被动套接字的服务器程序，它除了获取服务信息外，还建立
    一个被动套接字。*/
{
        int mainsock, flag=1;
        struct servent *servrec;
        struct sockaddr_in serv_addr;

        if ((servrec = getservbyname(servicename, "tcp")) == NULL) {
                        return(-1);
        }
        bzero((char *)&Mysock, sizeof(Sockets));
        Mysock.Port = servrec->s_port;    /* Service Port in Network Byte Order */

        if((mainsock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
                return(-2);
        }

        bzero((char *)&serv_addr, sizeof(serv_addr));
        serv_addr.sin_family = AF_INET;
        serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);   /*  任意网络接口  */
        serv_addr.sin_port = servrec->s_port;
        if (bind(mainsock, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) < 0) {
                close(mainsock);
                return(-3);
        }

        if (listen(mainsock, 5) == -1) {     /*  将主动套接字变为被动套接字，准备好接收连接  */
                close(mainsock);
                return(-4);
        }

        /* Set this socket as a Non-blocking socket. */
        if (ioctl(mainsock, FIONBIO, &flag) == -1) {
                close(mainsock);
                return(-5);
        }

        Mysock.DaemonSock = mainsock;
        FD_SET(mainsock, &Mysock.readfds);          /*  申明对主套接字“可读”感兴趣  */
        FD_SET(mainsock, &Mysock.exceptfds);        /*  申明对主套接字上例外事件感兴趣  */
        return(1);
}

void CloseMainSock()
/*  关闭主套接字，并清除对它上面事件的申明。在程序结束前关闭主套接字是一个好习惯  */
{
        close(Mysock.DaemonSock);
        FD_CLR(Mysock.DaemonSock, &Mysock.readfds);
        FD_CLR(Mysock.DaemonSock, &Mysock.exceptfds);
}

int CreateConnection(struct in_addr *sin_addr)
/* Create a Connection to remote host which IP address is in sin_addr.
    Param: sin_addr indicates the IP address in Network Byte Order.
    if succeed return the socket number which indicates this connection,
    else return error code (<0) */
{
        struct sockaddr_in server;     /* server address */
        int tmpsock, flag=1, i;

        if ((tmpsock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
                return(-1);

        server.sin_family = AF_INET;
        server.sin_port = Mysock.Port;
        server.sin_addr.s_addr = sin_addr->s_addr;
```

```
        /* Set this socket as a Non-blocking socket. */
        if (ioctl(tmpsock, FIONBIO, &flag) == -1) {
                close(tmpsock);
                return(-2);
        }

        /* Connect to the server. */
        if (connect(tmpsock, (struct sockaddr *)&server, sizeof(server)) < 0) {
                if ((errno != EWOULDBLOCK) && (errno != EINPROGRESS)) {
        /* 如果错误代码是 EWOULDBLOCK 和 EINPROGRESS，则不用关闭套接字，因为系统将在之后继续为套接字建
立连接，连接是否建立成功可用 select()函数来检测套接字是否“可写”来确定。*/
                        close(tmpsock);
                        return(-3);  /* Connect error. */
                }
        }

        FD_SET(tmpsock, &Mysock.readfds);
        FD_SET(tmpsock, &Mysock.writefds);
        FD_SET(tmpsock, &Mysock.exceptfds);

        i = 0;
        while (Mysock.Sockets[i] != 0) i++;       /* look for a blank sockets position */
        if (i >= 64) {
                close(tmpsock);
                return(-4);  /* too many connections */
        }

        Mysock.Sockets[i] = tmpsock;
        Mysock.SockNum++;
        return(i);
}

int AcceptConnection(struct in_addr *IPaddr)
/* Accept a connection. If succeed, return the data sockets number, else return -1. */
{
        int newsock, len, flag=1, i;
        struct sockaddr_in addr;

        len = sizeof(addr);
        bzero((char *)&addr, len);
        if ((newsock = accept(Mysock.DaemonSock, &addr, &len)) == -1)
                return(-1);  /* Accept error. */

        /* Set this socket as a Non-blocking socket. */
        ioctl(newsock, FIONBIO, &flag);

        FD_SET(newsock, &Mysock.readfds);
        FD_SET(newsock, &Mysock.writefds);
        FD_SET(newsock, &Mysock.exceptfds);

        /* Return IP address in the Parameter. */
        IPaddr->s_addr = addr.sin_addr.s_addr;

        i = 0;
        while (Mysock.Sockets[i] != 0) i++;       /* look for a blank sockets position */
        if (i >= 64) {
                close(newsock);
                return(-4);  /* too many connections */
        }

        Mysock.Sockets[i] = newsock;
        Mysock.SockNum++;
        return(i);
}

int CloseConnection(int Sockno)
/* Close a connection indicated by Sockno. */
{
        int retcode;

        if ((Sockno >= 64) || (Sockno < 0) || (Mysock.Sockets[Sockno] == 0))
                return(0);
```

```
        retcode = close(Mysock.Sockets[Sockno]);
        FD_CLR(Mysock.Sockets[Sockno], &Mysock.readfds);
        FD_CLR(Mysock.Sockets[Sockno], &Mysock.writefds);
        FD_CLR(Mysock.Sockets[Sockno], &Mysock.exceptfds);

        Mysock.Sockets[Sockno] = 0;
        Mysock.SockNum--;
        return(retcode);
}

int QuerySocketsMsg()
/* Query Sockets Message. If succeed return message number, else return -1.
   The message information stored in struct SockMsg. */
{
        fd_set rfds, wfds, efds;
        int retcode, i;
        struct timeval TimeOut;

        rfds = Mysock.readfds;
        wfds = Mysock.writefds;
        efds = Mysock.exceptfds;
        TimeOut.tv_sec = 0;             /*  立即返回，不阻塞。*/
        TimeOut.tv_usec = 0;

        bzero((char *)&SockMsg, sizeof(SockMsg));
        if ((retcode = select(64, &rfds, &wfds, &efds, &TimeOut)) == 0)
                return(0);

        if (FD_ISSET(Mysock.DaemonSock, &rfds))
                SockMsg.AcceptNum = 1;          /* some client call server. */

        for (i=0; i<64; i++)      /* Data in message */
        {
                if ((Mysock.Sockets[i] > 0) && (FD_ISSET(Mysock.Sockets[i], &rfds)))
                        SockMsg.ReadQueue[SockMsg.ReadNum++] = i;
        }

        for (i=0; i<64; i++)      /* Data out ready message */
        {
                if ((Mysock.Sockets[i] > 0) && (FD_ISSET(Mysock.Sockets[i], &wfds)))
                        SockMsg.WriteQueue[SockMsg.WriteNum++] = i;
        }

        if (FD_ISSET(Mysock.DaemonSock, &efds))
                SockMsg.AcceptNum = -1;          /* server socket error. */

        for (i=0; i<64; i++)      /* Error message */
        {
                if ((Mysock.Sockets[i] > 0) && (FD_ISSET(Mysock.Sockets[i], &efds)))
                        SockMsg.ExceptQueue[SockMsg.ExceptNum++] = i;
        }
        return(retcode);
}

int SendPacket(int Sockno, void *buf, int len)
/* Send a packet. If succeed return the number of send data, else return -1 */
{
        int actlen;

        if ((Sockno >= 64) || (Sockno < 0) || (Mysock.Sockets[Sockno] == 0))
                return(0);

        if ((actlen = send(Mysock.Sockets[Sockno], buf, len, 0)) < 0)
                return(-1);
        return(actlen);
}

int RecvPacket(int Sockno, void *buf, int size)
/* Receive a packet. If succeed return the number of receive data, else if the connection
   is shutdown by peer then return 0, otherwise return 0-errno */
{
        int actlen;
```

```
    if ((Sockno >= 64) || (Sockno < 0) || (Mysock.Sockets[Sockno] == 0))
        return(0);
    if ((actlen = recv(Mysock.Sockets[Sockno], buf, size, 0)) < 0)
        return(0-errno);
    return(actlen);    /* actlen 是接收的数据长度，如果为零，指示连接被对方关闭。*/
}
```

2.5.3 简单服务器程序示例

```
/* File Name:    server.c */
/* 这是一个很简单的重复服务器程序，它初始化好被动套接字后，循环等待接收连接。如果接收到连接，它显示数据套接字序号和客户端的 IP 地址；如果数据套接字上有数据到来，它接收数据并显示该连接的数据套接字序号和接收到的字符串。*/
#include "tcpsock.h"
main(argc, argv)
int argc;
char **argv;
{
    struct in_addr sin_addr;
    int retcode, i;
    char buf[32];

/* 对于服务器程序，它经常是处于无限循环状态，只有在用户主动 kill 该进程或系统关机时，它才结束。对于使用 kill 强行终止的服务器程序，由于主套接字没有关闭，资源没有主动释放，可能会给随后的服务器程序重新启动产生影响。因此，主动关闭主套接字是一个良好的变成习惯。下面的语句使程序在接收到 SIGINT、SIGQUIT 和 SIGTERM 等信号时先执行 CloseMainSock()函数关闭主套接字，然后再结束程序。因此，在使用 kill 强行终止服务器进程时，应该先使用 kill -2 PID 给服务器程序一个消息使其关闭主套接字，然后在用 kill -9 PID 强行结束该进程。*/
    (void) signal(SIGINT, CloseMainSock);
    (void) signal(SIGQUIT, CloseMainSock);
    (void) signal(SIGTERM, CloseMainSock);

    if ((retcode = InitPassiveSock("TestService")) < 0) {
        printf("InitPassiveSock: error code = %d\n", retcode);
        exit(-1);
    }

    while (1) {
        retcode = QuerySocketsMsg();    /* 查询网络消息 */
        if (SockMsg.AcceptNum == 1) {   /* 有外来连接等待接收？*/
            retcode = AcceptConnection(&sin_addr);
            printf("retcode = %d, IP = %s \n", retcode, inet_ntoa(sin_addr.s_addr));
        }
        else if (SockMsg.AcceptNum == -1)    /* 主套接字错误？*/
            printf("Daemon Sockets error.\n");
        for (i=0; i<SockMsg.ReadNum; i++) {   /* 接收外来数据 */
            if ((retcode = RecvPacket(SockMsg.ReadQueue[i], buf, 32)) > 0)
                printf("sockno %d Recv string = %s \n", SockMsg.ReadQueue[i], buf);
            else /* 返回数据长度为零，指示连接中断，关闭套接字。*/
                CloseConnection(SockMsg.ReadQueue[i]);
        }
    } /* end while */
}
```

2.5.4 简单客户程序示例

```
/* File Name:    client.c */
/* 客户程序在执行时，先初始化数据结构，然后等待用户输入命令。它识别四个命令：
conn(ect)：    和服务器建立连接；
send：    给指定连接发送数据；
clos(e)：    关闭指定连接；
quit：        退出客户程序。
*/
#include "tcpsock.h"

main(argc, argv)
int argc;
char **argv;
{
    char cmd_buf[16];
    struct in_addr sin_addr;
    int sockno1, retcode;
    char *buf = "This is a string for test.";
```

```
    sin_addr.s_addr = inet_addr("166.111.5.249");     /* 运行服务器程序的主机的 IP 地址 */

    if ((retcode = InitSocketsStruct("TestService")) < 0) {      /* 初始化数据结构 */
        printf("InitSocketsStruct: error code =    %d\n", retcode);
        exit(1);
    }

    while (1) {
        printf(">");
        gets(cmd_buf);
        if (!strncmp(cmd_buf, "conn", 4)) {
            retcode = CreateConnection(&sin_addr);        /* 建立连接 */
            printf("return code: %d\n", retcode);
        }
        else if(!strncmp(cmd_buf, "send", 4)) {
            printf("Sockets Number:");
            scanf("%d", &sockno1);
            retcode = SendPacket(sockno1, buf, 26);/* 发送数据 */
            printf("return code: %d\n", retcode, sizeof(buf));
        }
        else if (!strncmp(cmd_buf, "close", 4)) {
            printf("Sockets Number:");
            scanf("%d", &sockno1);
            retcode = CloseConnection(sockno1);           /* 关闭连接 */
            printf("return code: %d\n", retcode);
        }
        else if (!strncmp(cmd_buf, "quit", 4))
            exit(0);
        else
            putchar('\007');
    } /* end while */
}
```