第 2 章　扫描可控计算机

在上一章我们提到过一款软件——电子教室，其最主要的功能就是教师可以通过教师机程序控制和管理学生机，实现课堂统一教学的目的。教师机和学生机要进行通信，首先需要建立通信连接，那么，教师机怎样找到学生机，并与学生机建立通信连接呢？本章通过简单的通信程序实现扫描局域网内可控计算机的功能。

- 扫描可控计算机功能实现分析
- WinSock 通信编程技术
- MFC 网络通信编程技术

2.1　功能需求分析设计

扫描可控计算机的功能分析：教师机主动联系学生机，如果不能与学生机建立联系，则说明学生机端没有运行该程序（或出现错误），该学生机不在可控范围；反之，如果教师机与学生机成功地取得了联系，则证明学生机已启动程序，处于可控状态。教师机扫描学生机时，应该可以设置扫描范围，例如一个 IP 地址范围，教师机与范围内的所有学生机逐一联系，判断其是否处于可控状态，将处于可控状态的学生机显示在教师机端的程序界面上。

该功能实现的前提：

- 有两个应用程序，一个运行在教师机，一个运行在学生机。
- 教师机和学生机之间网络连通。

根据功能分析简单设计该程序的界面，并用 Visio 软件画出来，如图 2-1 和图 2-2 所示。

图 2-1　教师机端程序界面

图 2-2　学生机端程序界面

2.2　关键技术分析与核心程序

在开始做一个程序开发时，首先应分析该程序的核心功能（即该程序要干什么），然后分析该功能在技术上是否能够实现，需要采用什么技术来实现。解决了核心技术问题后，剩下的就是工作量的问题了。考虑一下扫描可控计算机功能，其核心技术就是两台联网的计算机之间进行通信，那么采用什么技术实现网络通信呢？目前最通用的网络编程接口就是套接字（Socket）。

最初，套接字是由加利福尼亚大学伯克利分校开发的，他们为了在 UNIX 操作系统下实现 TCP/IP 协议，而开发了这样一个调用网络操作的编程接口，也就是一套 API（应用程序接口），被称为 Socket 接口。现在 Socket 接口几乎是 TCP/IP 网络标准 API，很多基于 TCP/IP 的网络应用程序都是基于 Socket 而编写的。

2.2.1　使用 WinSock 编程实现

为更清楚地说明使用 WinSock 进行面向连接的通信原理，我们以电信局的普通电话服务为比较对象进行说明：

（1）电信局提供电话服务类似通信程序的 Server 端，普通电话用户类似通信程序的 Client 端。

（2）首先电信局必须建立一个电话总机。这就相当于必须在 Server 端建立一个 Socket（套接字），这一步通过调用 socket()函数实现。

（3）电信局必须给电话总机分配一个号码，以便使用户能够拨通该号码得到服务。同时，用户必须知道该总机的号码，才能拨通该电话。同样，通信程序在 Server 端也要为这一套接字指定一 port（端口），同时，连接该 Server 程序的 Client 程序必须知道该端口。指定端口通过调用 bind()函数实现。

（4）接下来电信局必须使总机开通并使总机能够高效地监听用户拨号，如果电信局所提供服务的用户数太多，你会发现拨打电信局总机老是忙音，通常电信局内部会使该总机对应的电话号码连到好几个负责交换的处理中心，在一个处理中心忙于处理当前的某个用户时，新到用户可自动转到其他处理中心得到服务。同样通信程序的 Server 端也要使自己的套接字绑定端口设置成监听状态，这是通用 listen()函数实现的，listen()的第二个参数是等待队列数，就如同你可以指定电信局建立几个负责交换的处理中心。

（5）用户知道了电信局的总机号后就可以进行拨打请求得到服务。在 WinSock 的世界里，作为 Client 端是要先用 socket()函数建立一个套接字，然后调用 connect()函数进行连接。

（6）电信局的总机接受了这用户拨打的电话后负责接通用户的线路，而总机本身则再回到等待的状态。Server 也是一样，调用 accept()函数进入监听处理过程，Server 端的代码即

在此处暂停，一旦 Server 端接到申请后系统会建立一个新的套接字来为此连接服务，而原先的套接字则再回到监听状态。

（7）当你电话讲完了，就可以挂上电话，彼此间也就离线了。Client 和 Server 间的套接字的关闭也是如此；这个关闭离线的动作，可由 Client 端关闭。有关闭套接字的函数为 closesocket()。

从以上情况可以看出在服务器端建立一个套接字，并进入实际的监听步骤的过程如下：socket()->bind()->listen()->accept()，在 accept()后，在 Server 端将生成一个新的套接字，然后 Server 将继续进入 accept()状态，Server 端程序用这个新的套接字来进行与 Client 端的通信，用 recv()函数接收数据，Client 端程序则通过 send()函数来发送数据。

在客户端也是采取类似的过程，其调用 Winsock 的过程如下：socket()->connect()->send()，首先建立一个 Socket，然后用 connect()函数将其与 Server 端的 Socket 连接，连接成功后调用 send()发送信息。

面向连接的套接字通信过程如图 2-3 所示。

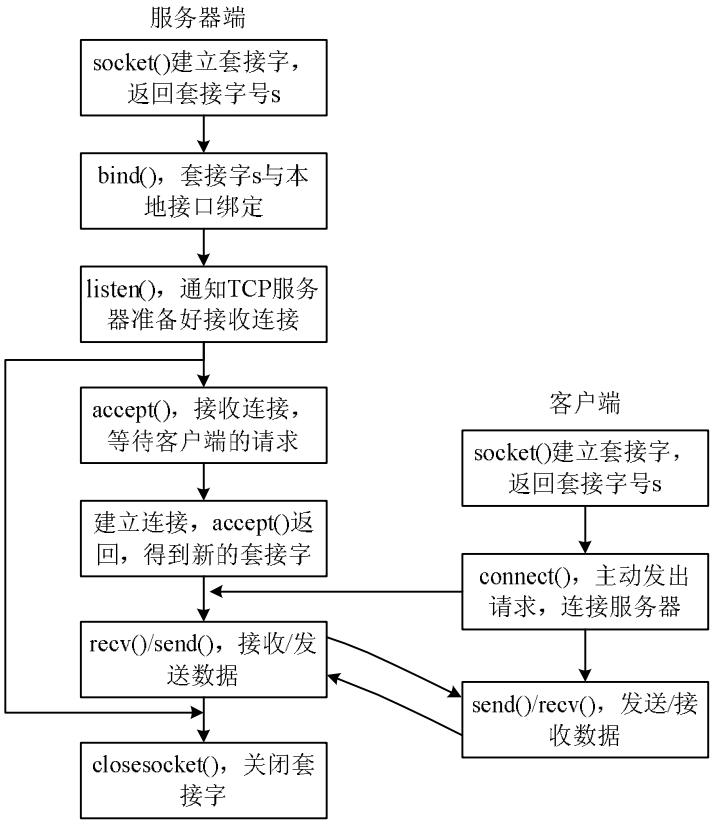

图 2-3　面向连接的套接字通信过程

【核心程序】——WinSock 实现基于 TCP 的客户端/服务器通信。

此实例目的是验证是否能够按照上面的通信过程通过 C++编码实现两个程序间的网络通信，实例分成两个程序，一个是服务器程序——MyServer，一个是客户端程序——MyClient。

两个程序可以互相发送和接收数据。

程序运行结果如图 2-4 和图 2-5 所示。

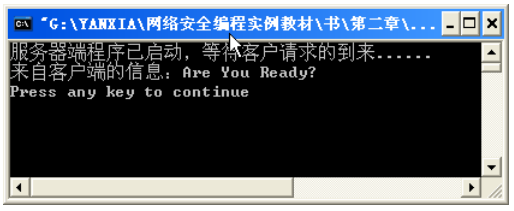

图 2-4　服务器端程序运行结果

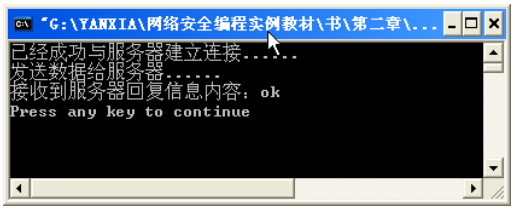

图 2-5　客户端程序运行结果

从上面的程序运行结果可以看到，首先服务器端程序启动，等待客户端的请求，客户端发出连接请求，双方成功建立连接后，客户端发送给服务器端一句话“Are You Ready？”服务器端回复客户端“ok”。

因为这两个程序都是没有图形用户界面的 Console 程序，因此在新建工程时，要选择“Win32 Console Application”类型。

服务器端程序代码清单：

```
#include "stdio.h"
#include <winsock2.h>
#pragma comment(lib,"ws2_32.lib")
int main(int argc, char* argv[])
{
    printf("服务器端程序已启动，等待客户请求的到来.....\n");

    //初始化 WinSock 为 2.0 版本
    WSADATA wsaData;
    int ret;
    if((ret=WSAStartup(MAKEWORD(2,2),&wsaData))!=0)
    {
            printf("初始化 WinSock 出错");
            return 0;
    }
    //定义两个 socket 变量，一个用来监听，一个用来建立连接和提供服务
    SOCKET listenSocket,acceptSocket;
    //设置服务器、客户的地址和端口
    struct sockaddr_in serv,cliet;
    //创建监听 socket
```

```
listenSocket=socket(AF_INET,SOCK_STREAM,0);
if(listenSocket==INVALID_SOCKET)
{
        printf("建立 socket 出错\n");
        return 0;
}
//设置服务器地址信息
serv.sin_family=AF_INET;
serv.sin_port=htons(8888);          //设置监听端口是 8888
serv.sin_addr.s_addr=htonl(INADDR_ANY);
//将监听 socket 与服务器地址绑定
if(bind(listenSocket,(LPSOCKADDR)&serv,sizeof(serv))==SOCKET_ERROR)
{
        printf("绑定出错\n");
        return 0;
}
//开始监听
if(listen(listenSocket,5)==SOCKET_ERROR)
{
        printf("监听出错\n");
        return 0;
}
//等待客户连接请求，如有连接请求到来，则用 acceptSocket 与其建立连接，进行通信
int len=sizeof(cliet);
acceptSocket=accept(listenSocket,(struct sockaddr *)&cliet,&len);
if(acceptSocket==INVALID_SOCKET)
{
        printf("建立连接出错\n");
        return 0;
}
//接收数据
char buf[1024];         //准备一个长度为 1024 的字符数组，用来接收数据
ret=recv(acceptSocket,buf,sizeof(buf),0);
if(ret==0)
        return 0;
char str[1024]={0};
printf("来自客户端的信息：");
printf(buf);
printf("\n");
//发送数据
memset(buf,0,sizeof(buf));
strcpy(buf,"ok");
ret=send(acceptSocket,buf,sizeof(buf),0);
if(ret==SOCKET_ERROR)
{
        printf("发送数据出错\n");
```

```
            return 0;
        }
        //关闭 socket
        closesocket(acceptSocket);
        closesocket(listenSocket);
        WSACleanup();
    }
```

代码解析：

- 头文件

```
#include <winsock2.h>
#pragma comment(lib,"ws2_32.lib")
```

使用 WinSock2 进行开发，需要头文件<winsock2.h>和库文件"ws2_32.lib"的支持。

- WSAStartup、WSAData 和 WSACleanup 函数

WSAStartup 函数：

功能为初始化 WinSock，在使用套接字函数前，都应该进行此函数的调用。其本质是调入合适的 WinSock 动态链接库。其函数原型如下：

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

参数 wVersionRequested：WinSock 版本，其低字节表示主版本号，高字节表示次版本号。一般使用宏 MAKEWORD 来表示。例如 MAKEWORD(2,2)表示版本 2.2。

参数 lpWSAData：WSAData 结构指针，它用来存储套接字信息。其结构定义如下：

WSAData 结构体定义：

```
typedef struct WSAData
{
    WORD wVersion;                          //版本
    WORD wHighVersion;                      //版本
    char szDescription[WSADESCRIPTION_LEN + 1];     //描述
    char szSystemStatus[WSASYS_STATUS_LEN + 1];     //系统状态
    unsigned short iMaxSockets;             //WinSocket2 及以后已忽略
    unsigned short iMaxUdpDg;               //WinSocket2 及以后已忽略
    char FAR * lpVendorInfo;                //WinSocket2 及以后已忽略
} WSADATA, * LPWSADATA;
```

WSACleanup 函数：

当应用程序不需再使用 WinSock API 的任何函数时，需要调用 WSACleanup()函数将其从 Windows Sockets 的实现中注销，以便释放为该应用程序或者 DLL 分配的任何资源。对应每个 WSAStartup()函数，必须有一个 WSACleanup()函数的调用。WSAStartup 和 WSACleanup 是成对出现的。其函数原型如下：

```
int WSACleanup(void);
```

- 结构体 sockaddr_in

```
struct sockaddr_in {
short   sin_family;
u_short   sin_port;
struct in_addr   sin_addr;
```

```
char    sin_zero[8];
};
struct    in_addr {
unsigned    long    s_addr;
};
typedef struct in_addr {
union {
        struct{
            unsigned char s_b1, s_b2, s_b3, s_b4;
        } S_un_b;
        struct {
            unsigned short s_w1, s_w2;
        } S_un_w;
        unsigned long S_addr;
    } S_un;
} IN_ADDR;
```

参数 sin_family：指代协议簇，在 Socket 编程中只能是 AF_INET。

参数 sin_port：存储端口号（使用网络字节顺序）。

参数 sin_addr：存储 IP 地址，使用 in_addr 这个数据结构。

参数 sin_zero：为了让 sockaddr 与 sockaddr_in 两个数据结构保持大小相同而保留的空字节，一般为 0。

参数 s_addr：按照网络字节顺序存储 IP 地址。

- htons 和 htonl 函数

这两个函数都是用于将主机字节顺序与网络字节顺序进行转换的。主机字节就是计算机内存里存放的整数或浮点数的方式，字节在内存中的存放，低字节在前，高字节在后排列。网络字节就是在网络上描述整数或浮点数的字节发送顺序（哪个字节被先发出去，哪个字节后发出去）。网络字节通常先传递高字节，再传递低字节。为了数据的一致性，就要把本地的数据转换成网络上使用的格式，然后发送出去，接收的时候也是一样的，经过转换然后才去使用这些数据。

htons 函数：

将 16 位主机字节转换为网络字节，使用函数 htons()，定义如下：

```
u_short htons(u_short hostshort);
```

参数 hostshort：表示主机字节顺序的数字，函数返回网络字节顺序的数字。

htonl 函数：

将主机的 unsigned long 值转换为网络字节顺序（32 位），使用函数 htonl()，其定义如下：

```
u_long htonl(u_long hostlong);
```

参数 hostlong：表示主机字节顺序的数字，函数返回一个网络字节顺序的数字。

- socket 和 closesocket 函数

socket 函数：

初始化 WinSock 的动态连接库后，需要在服务器端建立一个监听的套接字，为此可以调用 socket()函数来建立这个监听的套接字，并定义此套接字所使用的通信协议。在客户端也

同样调用 socket()来建立一个 TCP 或 UDP 套接字。其函数原型如下：

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

参数 af：说明套接字要使用的协议地址簇，目前只提供 AF_INET 表示使用互联网协议（IP）地址。

参数 type：描述套接字的类型，只能是 SOCK_STREAM、SOCK_DGRAM、SOCK_RAW 三个协议类型中的一个，分别表示流套接字、数据报套接字和原始套接字。

参数 protocol：该套接字使用的特定通信协定（如果使用者不指定则设为 0）。

closesocket 函数：

关闭服务器和客户端的通信连接是很简单的，这一过程可以由服务器或客户机的任一端启动，只要调用 closesocket()就可以了，而要关闭服务器端监听状态的套接字，同样也是利用此函数，该函数调用成功返回 0，否则返回 SOCKET_ERROR。其函数原型如下：

```
int t closesocket(SOCKET s);
```

参数 s：为要关闭的套接字识别码。

- bind 函数

建立套接字后，需要为服务器端定义的这个监听的套接字指定一个地址及监听端口（Port），这样客户端才知道要连接哪一个地址的哪个端口，为此需要调用 bind()函数，该函数调用成功返回 0，否则返回 SOCKET ERROR。

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

参数 s：表示未绑定的套接字的对象名，它是 socket()函数调用成功时返回的值。

参数 name：套接字的地址值，是一个与指定协议有关的地址结构指针，这个地址必须是执行这个程序所在机器的 IP 地址，在 WinSock 中使用 sockaddr_in 结构指定 IP 地址和端口号，sockaddr_in 结构的具体定义前面已经介绍。

参数 namelen：地址参数 name 的长度。

- listen 函数

listen 函数被 TCP 服务器端使用，通知协议内核用户进程准备接受接口的连接请求，指定了套接字等待的连接数限制值。若成功执行完毕，则函数返回 0；否则返回 SOCKET_ERROR。函数原型如下：

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

参数 s：已绑定但尚未连接的套接字句柄，也是由 socket()函数创建的套接字句柄。

参数 backlog：待处理的连接队列的最大长度。当连接的客户数大于这个最大长度并且服务进程没有来得及处理，则多出的连接请求会失败。目前允许的最大值为 5。

- accept 函数

对于服务器编程中最重要的一步等待并接受客户的连接，accept 函数就完成了这个功能。它从内核中取出已经建立的客户连接，然后把这个已经建立的连接返回给用户程序，此时用户程序就可以与自己的客户进行点到点的通信了。

接收套接字使用函数 accept()，其函数原型如下：

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

该函数返回一个新的套接字，该套接字可以用来与对方进行通信，而原先的套接字还是处于监听状态。

参数 s：表示处于监听模式的套接字句柄。

参数 addr：SOCKADDR IN 对象的地址。

参数 addrlen：表示 addr 参数的长度。

这个函数取出被搁置的连接队列中的第一个连接请求。函数执行后第 2 个参数和第 3 个参数指向的变量会被函数设置为客户 IPv4 地址信息和该信息的长度。在调用 accept()函数前应该先调用 listen()。如果这个函数出错，会返回 INVALID_SOCKET。accept()用于面向连接服务，参数 addr 和 addrlen 存放客户端的地址信息。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；调用后，服务器等待从编号为 s 的套接字上接收客户连接请求，而连接请求是由客户端的 connect()调用发出的。当有连接请求到达时，accept()调用将连接队列上的第一个客户端套接字地址及长度放入 addr 利 addrlen，并创建一个与 s 有相同特性的新套接字句柄。

- send 和 recv 函数

send 函数：

在面向连接的服务套接字中，不论是客户还是服务器应用程序都使用 send 函数发送数据，如果函数成功执行，返回值是发送的字节数，否则返回 SOCKET_ERROR。

其定义如下：

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

参数 s：表示已连接的本地套接字描述符。

参数 buf：表示要发送的缓冲区数据。

参数 len：表示缓冲区的长度。

参数 flags：表示标志类型，可以为 0，MSG_DONTROUTE，MSG_OOB。

recv 函数：

在面向连接的服务套接字中，不论是客户还是服务器应用程序都用 recv 函数从 TCP 连接的另一端接收数据。当应用程序调用 recv 函数时，recv 先等待 s 的发送缓冲中的数据被协议传送完毕，如果协议在传送 s 的发送缓冲中的数据时出现网络错误，那么 recv 函数返回 SOCKET_ERROR，如果 s 的发送缓冲中没有数据或者数据被协议成功发送完毕后，recv 先检查套接字 s 的接收缓冲区，如果 s 接收缓冲区中没有数据或者协议正在接收数据，那么 recv 就一直等待，直到协议把数据接收完毕。当协议把数据接收完毕，recv 函数就把 s 的接收缓冲区中的数据 copy 到 buf 中（注意协议接收到的数据可能大于 buf 的长度，所以在这种情况下要调用几次 recv 函数才能把 s 的接收缓冲区中的数据 copy 完。recv 函数仅仅是 copy 数据，真正的接收数据是协议来完成的），recv 函数返回其实际 copy 的字节数。如果 recv 函数在 copy 时出错，那么它返回 SOCKET_ERROR；如果 recv 函数在等待协议接收数据时网络中断了，那么它返回 0。其函数原型如下：

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

参数 s：接收端套接字描述符。

参数 buf：指明一个缓冲区，该缓冲区用来存放 recv 函数接收到的数据。

参数 len：buf 的长度。

参数 flags：表示标志，可以是 0，MSG_PEEK，MSG_OOB。一般置 0。

- memset 函数

memset()函数功能是一字节一字节地把整个数组设置为一个指定的值。其函数原型如下：

```
void *memset(void*,int,unsigned);
```

该函数在 mem.h 头文件中声明，它把数组的起始地址作为其第一个参数，第二个参数是设置数组每个字节的值，第三个参数是数组的长度（字节数，不是元素个数）。

memset()函数常用于内存空间初始化。如：

```
char str[100];
memset(str,0,100);
```

- strcpy 函数

strcpy 函数提供了字符串的复制，即 strcpy 只用于字符串复制，并且它不仅复制字符串内容之外，还会复制字符串的结束符。其函数原型如下：

```
char *strcpy(char *strDest, const char *strSrc)
```

客户端程序代码清单：

```
#include "stdio.h"
#include <winsock2.h>
#pragma comment(lib,"ws2_32.lib")
int main(int argc, char* argv[])
{
    //初始化 WinSock 为 2.0 版本
    WSADATA wsaData;
    int ret;
    if((ret=WSAStartup(MAKEWORD(2,2),&wsaData))!=0)
    {
        printf("初始化 WinSock 出错！");
        return;
    }
    //定义客户端 socket 变量
    SOCKET clientSocket;
    //定义服务器端地址信息
    struct sockaddr_in serv;
    //定义一个长度为 1024 的字符数组，用来存放发送和接收的数据
    char buf[1024];
    //设置服务器地址信息
    serv.sin_family=AF_INET;
    serv.sin_port=htons(8888);
    //设置需连接的服务器的监听端口（必须与服务器端绑定的监听端口一致）
    serv.sin_addr.s_addr=inet_addr("192.168.1.3");        //此处应设置为服务器端的 IP 地址
    //建立客户端 socket
    clientSocket=socket(AF_INET,SOCK_STREAM,0);
    if(clientSocket==INVALID_SOCKET)
    {
        printf("建立 socket 出错！");
        return;
```

```
    }
    //请求与服务器建立 TCP 连接
    if(connect(clientSocket,(struct sockaddr *)&serv,sizeof(serv))==INVALID_SOCKET)
    {
        printf("请求建立连接出错！");
        return;
    }
    printf("已经成功与服务器建立连接.....\n");
    //发送数据给服务器
    memset(buf,0,sizeof(buf));
    strcpy(buf,"Are You Ready?");
    ret=send(clientSocket,buf,sizeof(buf),0);
    if(ret==SOCKET_ERROR)
    {
        printf("发送数据出错");
        return;
    }
    printf("发送数据给服务器.....\n");
    //接收服务器发回的数据
    memset(buf,0,sizeof(buf));
    ret=recv(clientSocket,buf,sizeof(buf),0);
    if(ret==0)
        return;
    printf("接收到服务器回复信息内容：");
    printf(buf);
    printf("\n");
    //关闭 socket
    closesocket(clientSocket);
    WSACleanup();
}
```

代码解析：

- connect

connect 函数的功能就是完成面向连接的协议的连接过程。面向连接的协议，在建立连接的时候总会有一方先发送数据，那么谁调用了 connect 谁就是先发送数据的一方。connect 函数的功能是完成一个有连接协议的连接过程。如果函数调用没有错误发生，返回值为 0，否则返回值 SOCKET_ERROR。它的函数原型如下：

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

参数 s：表示是将要和服务器建立连接的套接字句柄。

参数 name：服务器的地址结构。

参数 namelen：地址名字 name 的长度。

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建立实际连接。

2.2.2 使用 MFC 的 CSocket 类实现

微软的 MFC 把复杂的 WinSock API 函数封装到类中，这使得编写网络应用程序更容易。MFC 中的 CAsyncSocket 类逐个封装了 WinSock API，为高级网络程序员提供了更加有力而灵活的方法。为了给程序员提供更方便的接口以自动处理这些任务，MFC 给出了 CSocket 类。CSocket 类是由 CAsyncSocket 继承而来的，事实上，在 MFC 中 CAsyncSocket 逐个封装了 WinSock API，每个 CAsyncSocket 对象代表一个 Windows Socket 对象，使用 CAsyncSocket 类要求程序员对网络编程较为熟悉。

相比起来，CSocket 类是 CAsyncSocket 的派生类，继承了它封装的 WinSock API。一个 CSocket 对象代表了一个比 CAsyncSocket 对象更高层次的 Windows Socket 的抽象，CSocket 类与 CSocketFile 类和 CArchive 类一起工作来发送和接收数据，因此使用它更加容易。CSocket 对象提供阻塞模式，因为阻塞功能对于 CArchive 的同步操作是至关重要的。在这里有必要对阻塞的概念作一解释：一个 Socket 可以处于“阻塞模式”或“非阻塞模式”，当一个套接字处于阻塞模式（即同步操作）时，它的阻塞函数直到操作完成才会返回控制权，之所以称为阻塞是因为此套接字的阻塞函数在完成操作返回之前什么也不能做。如果一个 Socket 处于非阻塞模式（即异步操作），则会被调用函数立即返回。

在 Win32 环境下，如果要使用具有阻塞性质的套接字，应该放在独立的工作线程中处理，利用多线程的方法使阻塞不至于干扰其他线程，也不会把 CPU 时间浪费在阻塞上。多线程的方法既可以使程序员享受 CSocket 带来的简化编程的便利，也不会影响用户界面对用户的反应。关于多线程编程，在后面的章节中会详细讲解和示范。

使用 CSocket 类的编程过程如下：

（1）构造一个 CSocket 对象。

（2）使用这个对象的 Create()成员函数产生一个 Socket 对象。在客户方程序中，除非需要数据报套接字，Create()函数一般情况下应该使用默认参数。而对于服务方程序，必须在调用 Create 时指定一个端口。

（3）如果是客户方套接字，则调用 Connect()函数与服务方套接字连接；如果是服务方套接字，则调用 Listen()开始监听来自客户方的连接请求，收到连接请求后，调用 Accept()函数接受请求，建立连接。请注意 Accept()成员函数需要一个新的并且为空的 CSocket 对象作为它的参数。

（4）连接成功建立后，可以使用 Receive()和 Send()函数进行数据发送和接收工作。

（5）通信结束后，使用 Close()函数销毁 CSocket 对象。

使用 CSocket 进行通信编程的流程图如图 2-6 所示。

使用 MFC 提供的 CSocket 类可以使我们的通信程序变得更加简单，为了得到 MFC 的支持，需要在创建工程时指明创建的是 MFC AppWizard[exe]程序，在“下一步”的选项中，选择要创建的应用程序类型是“基本对话框”，在“下一步”的选项中，可以选择“希望支持 Windows Sockets”，如果选择了此选项，则系统会自动初始化 WinSock，如果没有选择，则需要在代码中自己进行初始化。工程创建完后，有一个简单的对话框界面，可以根据需要进行界面的修改。服务器端界面和客户端界面经过修改后如图 2-7 和图 2-9 所示。

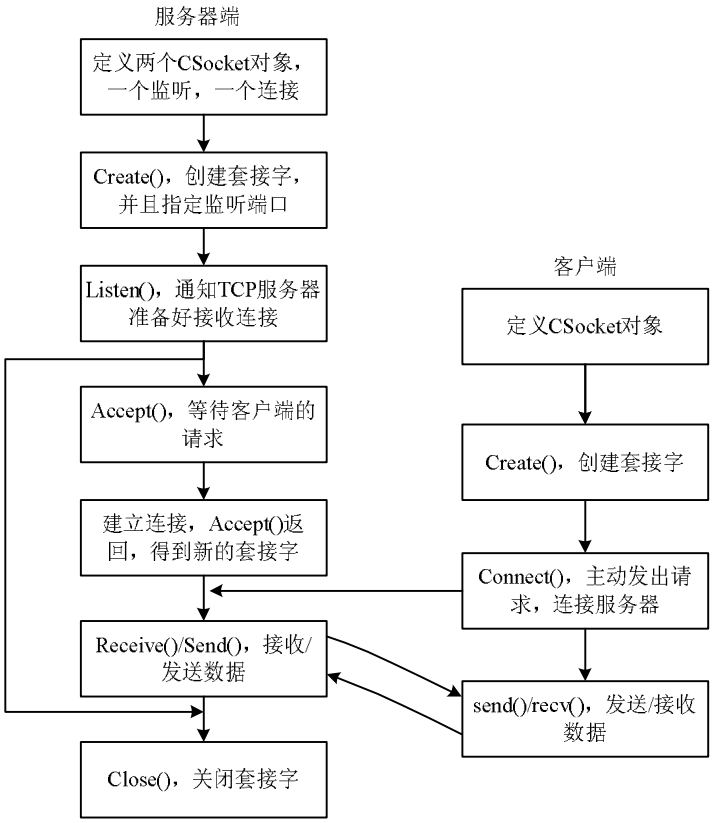

图 2-6　使用 CSocket 类实现的面向连接的套接字通信过程

图 2-7　服务器端程序界面

服务器端源程序：

在写入代码的文件头，引入所需头文件：#include <afxsock.h>，修改界面如图 2-7 后，双击“启动服务器程序”按钮，添加按钮的消息处理函数，在该函数里添加代码如下：

```
//初始化 WinSock 为 2.0 版本
WSADATA wsaData;
if((WSAStartup(MAKEWORD(2,2),&wsaData))!=0)
{
    MessageBox("初始化 WinSock 出错！",NULL,MB_OK);
    return;
}
//定义两个 socket 变量，一个用来监听，一个用来建立连接和提供服务
CSocket listenSocket,acceptSocket;
```

```
int ret;
//创建监听 socket
ret=listenSocket.Create(8888);
if(ret==0)
{
    MessageBox("建立 socket 出错！",NULL,MB_OK);
    return;
}
//开始监听
ret=listenSocket.Listen(5);
if(ret==0)
{
    MessageBox("监听出错",NULL,MB_OK);
    return;
}
//等待客户连接请求，如有连接请求到来，则用 acceptSocket 与其建立连接，进行通信
ret=listenSocket.Accept(acceptSocket,NULL,NULL);
if(ret==0)
{
    MessageBox("建立连接出错",NULL,MB_OK);
    return;
    }
//接收数据
char buf[1024];//准备一个长度为 1024 的字符数组，用来接收数据
ret=acceptSocket.Receive(buf,sizeof(buf),0);
if(ret==0)
    return;
CString str="从客户端发来的信息：";
str=str+buf;
MessageBox(str,"通信内容",MB_OK);
//发送数据
memset(buf,0,sizeof(buf));
strcpy(buf,"ok");
ret=acceptSocket.Send(buf,sizeof(buf),0);
if(ret==SOCKET_ERROR)
{
    MessageBox("发送数据出错",NULL,MB_OK);
    return;
}
//关闭 socket
acceptSocket.Close();
listenSocket.Close();
WSACleanup();
```

代码解析：

- #include <afxsock.h>

头文件 afxsock.h 中包含 CAsyncSocket 和 CSocket 类定义。如果使用这些类或从这些类

派生的任何类，必须确保在项目中包含 afxsock.h 文件。

➢ Create 函数

构造套接字对象后调用 Create()成员函数创建 Socket 句柄，并调用 Bind()成员函数将其与指定的地址绑定。如果该函数调用成功，则返回非 0 值，否则返回 0 值。其函数原型如下：

```
BOOL Create(UINT nSocketPort = 0, int nSocketType = SOCK_STREAM, long lEvent = FD_READ |
FD_WRITE | FD_OOB | FD_ACCEPT | FD_CONNECT | FD_CLOSE, LPCTSTR lpszSocketAddress =
NULL);
```

参数 nSocketPort：与套接字一起使用的端口号，默认值为 0，表示自动选择端口号。

参数 nSocketType：指定要创建的套接字类型，默认为流式套接字。

参数 lEvent：指定感兴趣的网络事件的掩码位，网络事件见表 2-1。

参数 lpszSocketAddress：指定套接字的网络地址。

表 2-1　网络事件类型

事件标记	事件内容
FD_READ	接收读准备好的通知
FD_WRITE	接收写准备好的通知
FD_OOB	接收带外数据到达的通知
FD_ACCEPT	接收等待连接成功的通知
FD_CONNECT	接收已连接好的通知
FD_CLOSE	接收套接字关闭的通知

➢ Listen 函数

该函数用于侦听连接请求，如果函数执行成功则返回非 0 值，否则返回 0 值。其函数原型如下：

```
BOOL    Listen(int nConnectionBacklog = 5);
```

参数 nConnectionBacklog：指定连接请求队列的最大连接数目，默认为 5。

➢ Accept 函数

该函数接受一个套接字的连接请求，从连接请求队列中取出第一个连接，并创建一个与这个套接字具有相同属性的套接字，并与第一个参数（Socket 对象）相关联，原始的套接字依然保持打开并且侦听。如果函数执行成功则返回非 0 值，否则返回 0 值。其函数原型如下：

```
virtual BOOL Accept (CAsyncSocket & rConnectedSocket,   SOCKADDR* lpSockAddr = NULL, int*
lpSockAddrLen = NULL) ;
```

参数 rConnectedSocket：用来进行连接的新套接字的引用。

参数 lpSlpSockAddr：用来返回发送连接请求的套接字地址的 SOCKADDR 结构指针。

参数 lpSockAddrLen：指向 pSockAddr 结构中地址的字节长度的指针。

➢ Receive 函数

该函数从一个面向连接的流或者面向无连接的数据报套接字接收数据。如果函数调用成功，则返回所接收到的字节数；如果连接被关闭，则返回 0；如果函数调用失败，则返回 SOCKET_ERROR。其函数原型如下：

```
virtual int Receive(void* lpBuf, int nBufLen, int nFlags = 0);
```

参数 lpBuf：接收数据的缓冲区。

参数 nBufLen：缓冲区的字节节长度。

参数 nFlags：用来表示函数的实现，值为 MSG_OOB 和 MSG_PEEK 项的组合，通常默认为 0。

- Send 函数

该函数通过数据报或数据流向对方套接字发送数据，如果函数成功调用，则返回发送的字节总数，否则返回 SOCKET_ERROR。其函数原型如下：

```
virtual int Send(const void* lpBuf, int nBufLen, int nFlags = 0);
```

参数 lpBuf：要发送的数据的缓冲区地址。

参数 nBufLen：为 lpBuf 缓冲区的字节长度。

参数 nFlags：指定函数的调用标志，其取值为 MSG_DONTROUTE 和 MSG_OOB 的组合，默认值为 0。

- Close 函数

该函数用来关闭套接字并释放 Socket 描述符，其函数原型如下：

```
Virtual void Close ();
```

- MessageBox 函数

这个函数可以在 VC 里面显示一个标准对话框，通常用来弹出提示对话框。其函数原型如下：

```
int MessageBox(HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT, uType);
```

参数 hWnd：父窗口的句柄。

参数 lpText：对话框的内容。

参数 lpCaption：对话框的标题。

参数 uType：窗口的风格。该函数为窗口提供了多种风格，具体标示见表 2-2。

表 2-2　弹出对话框窗口风格

风格标示	风格
MB_DEFBUTTON1	缺省按钮为第一个按钮
MB_DEFBUTTON2	缺省按钮为第二个按钮
MB_DEFBUTTON3	缺省按钮为第三个按钮
MB_ICONEXCLAMATION	显示图标为惊叹号
MB_ICONQUESTION	显示图标为问号
MB_ICONSTOP	显示图标为红叉叉
MB_ YESNOCANCEL	显示是、否、取消按钮
MB_OK	只显示确定按钮
MB_OKCANCEL	显示确定和取消按钮
MB_RETRYCANCEL	显示重试和取消按钮

用户针对对话框作了相应选择后，根据用户选择的不同，系统有不同的返回值，见表 2-3。

表 2-3　对话框返回值

返回值	含义
IDABOUT	按下了终止按钮
IDRETRY	按下了重试按钮
IDIGNORE	按下了忽略按钮
IDOK	按下了确定按钮
IDCANSEL	按下了取消按钮
IDYES	按下了是按钮
IDNO	按下了否按钮

例如：要显示一个图标为问号，按钮为“是”、“否”、“取消”，并且默认按钮为“否”，标题为“问题”，内容为“你明白了吗？”的对话框，并且还要求如果单击了“是”按钮，则给 a 赋值为 1，那么语句则为：

```
if(MessageBox("你明白了吗？", "问题", MB_YESNOCANCEL|MB_DEFBUTTON2|
MB_ICONQUESTION ) == IDYES) {
a = 1;
}
```

该语句执行后弹出的对话框如图 2-8 所示。

接下来看看客户端程序怎样实现，界面如图 2-9 所示。

图 2-8　弹出的对话框

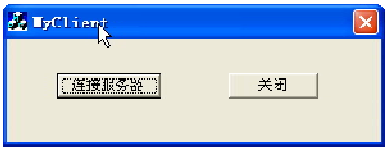

图 2-9　客户端程序界面

客户端源程序：

在写入代码的文件头，引入所需头文件：#include <afxsock.h>，修改界面如图 2-9 后，双击“连接服务器”按钮，添加按钮的消息处理函数，在该函数里添加代码如下：

```
//初始化 WinSock 为 2.0 版本
WSADATA wsaData;
if((WSAStartup(MAKEWORD(2,2),&wsaData))!=0)
{
    MessageBox("初始化 WinSock 出错！",NULL,MB_OK);
    return;
}
//定义客户端 socket 变量
CSocket clientSocket;
int ret;
//建立客户端 socket
ret=clientSocket.Create();
if(ret==0)
```

```
{
    MessageBox("建立 socket 出错！",NULL,MB_OK);
    return;
}
//定义一个长度为 1024 的字符数组，用来存放发送和接收的数据
char buf[1024];
//请求与服务器建立 TCP 连接，在参数中指明服务器的 IP 地址和监听端口号
ret=clientSocket.Connect("192.168.1.3",8888);
if(ret==0)
{
    MessageBox("请求建立连接出错！",NULL,MB_OK);
    return;
}
//发送数据给服务器
memset(buf,0,sizeof(buf));
strcpy(buf,"Are You Ready?");
ret=clientSocket.Send(buf,sizeof(buf),0);
if(ret==0)
{
    MessageBox("发送数据出错",NULL,MB_OK);
    return;
}
//接收服务器发回的数据
memset(buf,0,sizeof(buf));
ret=clientSocket.Receive(buf,sizeof(buf),0);
if(ret==0)
    return;
CString str="从服务器端发回的信息：";
str=str+buf;
MessageBox(str,"通信内容",MB_OK);
//关闭 socket
clientSocket.Close();
WSACleanup();
```

代码解析：

- Connect 函数

该函数用于未连接的数据流或数据报套接字建立连接。如果函数调用成功则返回非 0 值，否则返回 0 值。其函数原型如下：

```
BOOL Connect (LPCTSTR lpszHostAddress, UINT nHostPort);
```

参数 lpszHostAddress：用于要连接的服务器网络地址。

参数 nHostPort：指定套接字应用程序使用的端口号。

```
BOOL Connect (const SOCKADDR* lpSockAddr, int nSockAddrLen);
```

参数 lpSockAddr：指向 SOCKADDR 结构的指针。

参数 nSockAddrLen：lpSockAddr 指针中地址的字节长度。

2.3 扩展核心程序

经过分析，确定应该首先建立两个应用程序，一个 teacher 和一个 student。student 程序应首先启动，teacher 程序能够主动向所有 student 程序发出连接请求，能够成功建立连接的就是可控学生机，把其 IP 地址显示在 teacher 程序界面上。

2.3.1 使用 MFC 的 CSocket 类编程实现

首先创建工程 student，类型是 MFC AppWizard[exe]，在“下一步”的选项中，选择要创建的应用程序类型是“基本对话框”，在“下一步”的选项中，选择“希望支持 Windows Sockets”选项，然后完成工程的创建。将程序界面修改如图 2-10 所示。

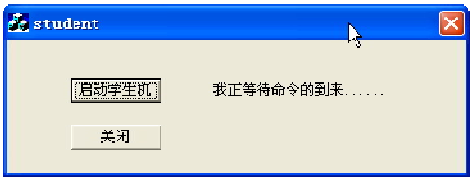

图 2-10 student 程序界面

界面上共有两种控件，左侧是两个按钮控件，右侧是一个静态文本控件。双击“启动学生机”按钮，添加按钮消息处理函数，源代码如下：

```
CSocket listenSocket;
CSocket dataSocket;
int ret;
ret=listenSocket.Create(8888);      //创建一个 socket，指定监听端口 8888
if(ret==0)
{
    MessageBox("创建 socket 出错",NULL,MB_OK);
    return;
}
ret=listenSocket.Listen(5);      //开始监听
if(ret==0)
{
    MessageBox("监听出错",NULL,MB_OK);
    return;
}
ret=listenSocket.Accept(dataSocket,NULL,NULL);    //当接到客户请求时，建立连接
if(ret==0)
{
    MessageBox("建立连接出错",NULL,MB_OK);
    return;
}
char strRev[256]={0};
dataSocket.Receive(strRev,256,0);          //接受客户发送的消息
```

```
CString strMeg=strRev;
if(!strMeg.Compare("ok")){
    CString strSend="ok";
    dataSocket.Send(strSend,strSend.GetLength(),0);    //发送消息给客户
}
listenSocket.Close();
    dataSocket.Close();
```

接下来，创建工程 teacher，类型是 MFC AppWizard[exe]，在“下一步”的选项中，选择要创建的应用程序类型是“基本对话框”，在“下一步”的选项中，选择“希望支持 Windows Sockets”选项，然后完成工程的创建。将程序界面修改如图 2-11 所示。

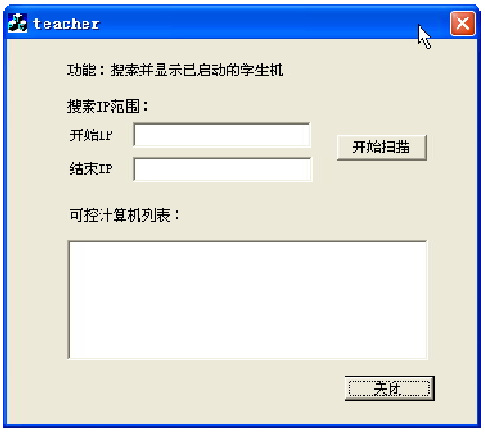

图 2-11　teacher 程序界面

界面上涉及几种 MFC 控件，显示文字内容的是静态文本控件，允许输入 IP 地址的两个控件是编辑框，右侧“开始扫描”是按钮，下面用于显示可控计算机列表的是列表框控件。因为界面开发不是本教材的重点，所以只简单介绍用到的控件。

首先把 teacher 程序的界面按照图 2-11 进行修改，然后按照要求修改几个控件的 ID（选中要修改的控件，鼠标右键单击，选择属性项，在属性对话框中修改 ID 值）：输入开始 IP 的编辑框控件 ID 修改为：IDC_START；输入结束 IP 的编辑框控件 ID 修改为：IDC_END；显示可控计算机列表的列表框控件 ID 修改为：IDC_SHOW；“开始扫描”按钮控件 ID 修改为：IDC_SERCH。

然后为上面三个控件添加关联的成员变量。添加方法：单击“查看”菜单，选择“建立类向导”，在 MFC ClassWizard 对话框中，选择 Member Variables 选项卡，为 IDC_START 添加成员变量 m_start，类型为 CString；为 IDC_END 添加成员变量 m_end，类型为 CString；为 IDC_SHOW 添加成员变量 m_show，类型为 CListBox。当需要从控件里读出数据或者在控件里显示数据时，都需要用到控件关联的成员变量。

双击“开始扫描”按钮，添加按钮消息处理函数，在该函数中写入源代码：

```
//获取要搜索的 IP 范围
UpdateData(true);//使用户输入的两个 IP 地址保存到编辑框对应的成员变量中
//对获取到的 IP 地址范围进行解析，取出 IP 地址的最后一段，用于循环使用
CString ipHead=m_start.Left(m_start.ReverseFind('.'));
```

```
int ip1,ip2;   //ip 地址循环的起始和结束值
ip1=atoi(m_start.Right(m_start.GetLength()-m_start.ReverseFind('.')-1));
ip2=atoi(m_end.Right(m_end.GetLength()-m_end.ReverseFind('.')-1));
//Socket 连接
CSocket clientSocket;
clientSocket.Create();//创建一个 socket
char strRev[256]={0};
for(int i=ip1;i<=ip2;i++){
    CString tp;
    tp.Format("%d",i);
    CString ip = ipHead + "." + tp;
//参数是要连接服务器的 IP 地址和服务器的监听端口
    int ret = clientSocket.Connect(ip,8888);
    if(ret==0)
        continue;
    CString msg = "ok";
    clientSocket.Send(msg,msg.GetLength());      //发送消息给服务器
    memset(strRev,0,256);
    clientSocket.Receive(strRev,256,0);//接收服务器的消息
    msg=strRev;
    ret = msg.Compare("ok");
    if(ret==0){
        //添加到列表中
        m_show.AddString(ip + "机器已运行");
    }
}
clientSocket.Close();
```

思考：进一步扩展，如果希望程序可以由用户自己添加学生机，如何实现？

2.3.2 使用 WinSock 编程实现

修改上面的 CSocket 通信程序，使用 WinSock 编程实现相同的功能，界面开发部分完全相同，只需针对按钮消息相应函数中的代码进行修改，修改后的代码如下：

student 源程序：

```
//定义两个 socket 变量，一个用来监听，一个用来建立连接和提供服务
SOCKET listenSocket;
SOCKET dataSocket;
BOOL ret;
//设置服务器、客户的地址和端口
struct sockaddr_in serv,cliet;
//创建监听 socket
listenSocket=socket(AF_INET,SOCK_STREAM,0);
if(listenSocket==INVALID_SOCKET)
{
    MessageBox("建立 socket 出错",NULL,MB_OK);
    return;
```

```
}
//设置服务器地址信息
serv.sin_family=AF_INET;
serv.sin_port=htons(8888);//设置监听端口是 8888
serv.sin_addr.s_addr=htonl(INADDR_ANY);
//将监听 socket 与服务器地址绑定
if(bind(listenSocket,(LPSOCKADDR)&serv,sizeof(serv))==SOCKET_ERROR)
{
    MessageBox("绑定出错",NULL,MB_OK);
    return;
    }
    //开始监听
    if(listen(listenSocket,5)==SOCKET_ERROR)
    {
            MessageBox("监听出错",NULL,MB_OK);
            return;
}
//等待客户连接请求，如有连接请求到来，则用 acceptSocket 与其建立连接，进行通信
int len=sizeof(cliet);
dataSocket=accept(listenSocket,(struct sockaddr *)&cliet,&len);
if(dataSocket==INVALID_SOCKET)
{
    MessageBox("建立连接出错",NULL,MB_OK);
    return;
}
char strRev[256]={0};
ret=recv(dataSocket,strRev,sizeof(strRev),0);
if(ret==0)
    return;
CString strMeg=strRev;
if(!strMeg.Compare("ok")){
    memset(strRev,0,sizeof(strRev));
    strcpy(strRev,"ok");
    ret=send(dataSocket,strRev,sizeof(strRev),0);
    if(ret==SOCKET_ERROR)
    {
            return;
    }
}
closesocket(dataSocket);
closesocket(listenSocket);
```

teacher 源程序：

```
//获取要搜索的 IP 范围
UpdateData(true);
//对获取到的 IP 地址范围进行解析，取出 IP 地址的最后一段，用于循环使用
CString ipHead=m_start.Left(m_start.ReverseFind('.'));
```

```
int ip1,ip2;   //ip 地址循环的起始和结束值
ip1=atoi(m_start.Right(m_start.GetLength()-m_start.ReverseFind('.')-1));
ip2=atoi(m_end.Right(m_end.GetLength()-m_end.ReverseFind('.')-1));
    SOCKET clientSocket;
//定义服务器端地址信息
struct sockaddr_in serv;
char strRev[256]={0};
CString tp,ip;
for(int i=ip1;i<=ip2;i++){
    tp.Format("%d",i);
    ip = ipHead + "." + tp;
    //设置服务器地址信息
    serv.sin_family=AF_INET;
    serv.sin_port=htons(8888);
    //设置需连接的服务器的监听端口（必须与服务器端绑定的监听端口一致）
    serv.sin_addr.s_addr=inet_addr(ip);//此处应设置为服务器端的 IP 地址
    //建立客户端 socket
    clientSocket=socket(AF_INET,SOCK_STREAM,0);
    if(clientSocket==INVALID_SOCKET)
    {
        MessageBox("建立 socket 出错！",NULL,MB_OK);
        return;
    }
    //请求与服务器建立 TCP 连接
    if(connect(clientSocket,(struct sockaddr *)&serv,sizeof(serv))==INVALID_SOCKET)
        continue;
    memset(strRev,0,sizeof(strRev));
    strcpy(strRev,"ok");
    send(clientSocket,strRev,sizeof(strRev),0);//发送消息给服务器

    memset(strRev,0,sizeof(strRev));
    recv(clientSocket,strRev,sizeof(strRev),0);;//接收服务器的消息
    CString msg=strRev;
    int ret = msg.Compare("ok");
    if(ret==0){
        //添加到列表中
        m_show.AddString(ip + "机器已运行");
    }
    closesocket(clientSocket);
}
```

2.4 知识扩展

2.4.1 澄清一些概念

Socket 在英文中是“插座”的意思，它的设计者实际上是暗指电话插座。因为在 Socket 环境下编程很像是模拟打电话，Internet 的 IP 就是电话号码，要打电话，需要电话插座，在程序中就是向系统申请一个 Socket，以后两台机器上的程序“交谈”都是通过这个 Socket 来进行的。对程序员来说，也可以把 Socket 看成一个文件指针，只要向指针所指的文件读写数据，就可以实现双向通信。利用 Socket 进行通信，有两种主要的方式。第一种是面向连接的流方式。顾名思义，在这种方式下，两个通信的应用程序之间先要建立一种连接链路，其过程好像在打电话。一台计算机（电话）要想和另一台计算机（电话）进行数据传输（通话），须首先获得一条链路，只有确定了这条通路之后，数据（通话）才能被正确接收和发送，这种方式对应的是 TCP（Transport Control Protocol）。第二种叫做无连接的数据报文方式，这时两台计算机像是把数据放在一个信封里，通过网络寄给对方，信在传送的过程中有可能会残缺不全，而且后发出的信也有可能会先收到，它对应的是 UDP（User Datagram Protocol）。

流方式的特点是通信可靠，对数据有校验和重发的机制，通常用来做数据文件的传输，如 FTP、Telnet 等，数据报文方式由于取消了重发校验机制，能够达到较高的通信速率，可用于对数据可靠性要求不高的通信，如实时的语音、图像转送和广播消息等。

在 ISO 的 OSI 网络七层协议中，WinSock 主要负责控制数据的输入和输出，也就是传输层和网络层。WinSock 屏蔽了数据链路层和物理层，它的出现给 Windows 下的网络编程带来了巨大的变化。

套接字有同步阻塞方式和异步非阻塞方式两种使用方法。同步和异步往往都是针对一个函数来说的，“同步”就是函数直到其要执行的功能全部完成时才返回，而“异步”则是函数仅仅做一些简单的工作，然后马上返回，所要实现的功能留给别的线程或者消息循环去完成。

阻塞方式的套接字简单、易用，但效率低。相比之下，异步套接字使用复杂，但效率很高。本章实例使用的是阻塞方式。

2.4.2 WinSock 编程原理

Windows Sockets API 是 Windows 下的网络应用程序接口，用于网络通信。API 函数有 1.1 版和 2.0 版，称为 WinSockl 和 WinSock2，通过前缀 WSA 区分。2.0 版有良好的向后兼容性，任何使用 1.1 版的源代码、二进制文件和应用程序都可以不加修改地在 2.0 规范下使用。现在基本上都使用 2.0 版本进行开发，它主要是为了适应近年来网络技术的迅猛发展，特别是多媒体网络技术迅速发展的需要，通过制定 Windows Sockets2 规范来提供一个与协议无关的网络传输接口。

Windows Sockets 使用套接字进行编程，套接字编程是面向客户端/服务器模型而设计的，因此系统中需要客户端和服务器两个不同类型的进程，根据连接类型的不同，对于面向连接的 TCP 服务和无连接的 UDP 服务，服务器分别采取不同的处理操作来对客户提供服务。

对于面向连接的 TCP 服务，服务器需要等待客户端向其提出的建立连接的申请，一旦接收到客户端的连接请求，服务器返回一个新的套接字描述符，通过该描述符调用数据传输函数即可与客户端进行数据的收发。

对于无连接的 UDP 服务，服务器通常是面向事务处理的，服务器和客户端在传输数据之前不需要进行连接的申请和建立，数据传输结束后，直接关闭套接字即可完成一次通信。

WinSock 中的主要函数如下：

- WinSock 的打开——WSAStartup()
- 服务器建立套接字——socket()
- 服务器绑定端口——bind()
- 客户端提出连接申请——connect()
- 服务器端接受客户端的连接请求——accept()
- 数据的传送——send()和 recv()
- 关闭套接字——closesocket()
- 关闭 WinSock——WSACleanup()

这些函数的具体定义信息前面已经详细介绍了，这里就不再赘述。

2.4.3 MFC 网络编程

MFC 中有两个主要的网络编程类：CAsyncSocket 类和 CSocket 类。CAsyncSocket 类用面向对象的方法封装了 WinSock，CSocket 类是 CAsyncSocket 类的子类。这两个类是 Visual C++环境中网络编程经常使用的两个类。

1. CAsyncSocket 类介绍

CAsyncSocket 类对 Windows Sockets API 函数进行了封装，它是从 CObject 类派生出来的。该类在非常低的级别上封装 Windows Sockets API。CAsyncSocket 适合那些对网络通信细节很了解，但希望利用回调的便利通知网络事件的程序员使用。如果想利用 Windows Sockets 方便地处理 MFC 应用程序中的多个网络协议，而又不想放弃灵活性，可以使用 CAsyncSocket，但是程序员必须自己处理阻塞，字节序的差异和 Unicode 多字节字符集（MBCS）的转换。CAsyncSocket 类的重要函数如下：

- 构造套接字对象——Create()
- 接受套接字的连接请求——Accept()
- 将本机地址绑定到套接字——Bind()
- 建立连接——Connect()
- 监听连接请求——Listen()
- 发送数据——Send()和 SendTo()
- 接受数据——Receive()和 ReceiveFrom()
- 禁止发送接收数据——ShutDown()
- 关闭套接字——Close()

这些函数中大部分在前面程序代码详解里已经详细介绍过，这里不再赘述。没有介绍的函数如下：

- 将本机地址绑定到套接字——Bind()

该函数将本机地址绑定到套接字，在 Connect 或者 Listen 之前可以被调用。在能接受连接请求前，监听的服务器套接字必须选择一个端口号并调用 Bind 将本地名分配给未命名的 Windows 套接字，使其知道自己的地址和端口号，如果函数调用成功则返回非 0 值，否则返 0 值。其函数原型如下：

```
BOOL Bind(UINT nSocketPort, LPCTSTR lpszSocketAddress = NULL);
```

参数 nSocketPort：要绑定的套接字端口号。

参数 lpszSocketAddress：要绑定的套接字网络地址。

```
BOOL Bind (const SOCKADDR* lpSockAddr, int nSockAddrLen);
```

参数 lpSockAddr：指向 SOCKADDR 结构的指针。

参数 nSockAddrLen：lpSockAddr 指针中地址的字节长度。

- 发送数据——SendTo()

该函数与 Send()类似，也是通过数据报或者数据流套接字发送数据，如果函数成功调用，则返回发送的字节总数，否则返回 SOCKET_ERROR。其函数原型如下：

```
int SendTo (const void* lpBuf, int nBufLen, UINT nHostPort, LPCTSTR lpszHostAddress = NULL, int
nFlags = 0);
int SendTo(const void* lpBuf, int nBufLen, const SOCKADDR* lpSockAddr, int nSockAddrLen，int
nFlags = 0);
```

参数 lpBuf：要发送的数据缓冲区地址。

参数 nBufLen：lpBuf 缓冲区的字节长度。

参数 nHostPort：套接字应用程序的端口号。

参数 lpszHostAddress：被连接的套接字的网络地址。

参数 lpSockAddr：SOCKADDR 结构的指针。

参数 nSockAddrLen：指针所指向结构中的地址的字节长度的指针。

参数 nFlags：函数的调用标志，其取值为 MSG_DONTROUTE 和 MSG_OOB 的组合，默认值为 0。

- 接收数据——ReceiveFrom()

该函数从面向连接流或者无连接的数据报套接字接收数据，并将源地址存放在 SOCKADDR 结构或者 rSockeLAddress 中。如果函数调用成功，则返回所读入的字节数，如果连接被关闭，则返回 0，如果函数调用失败，则返回 SOCKET_ERROR。其函数原型如下：

```
int ReceiveFrom(void* lpBuf, int nBufLen, CString& rSocketAddress, UINT&
rSocketPort, int nFlags = 0);
int ReceiveFrom(void* lpBuf, int nBufLen, SOCKADDR* lpSockAddr, int*lpSockAddrLen, int nFlags
= 0);
```

参数 lpBuf：接收数据的缓冲区。

参数 nBufLen：缓冲区的字节节长度。

参数 rSocketAddress：一个点分隔的字符串 IP 地址的 CString 对象的引用。

参数 rSocketPort：端口号的 UINT 类型的引用。

参数 lpSockAddr：返回源地址的 SOCKADDR 结构指针。

参数 lpSockAddrLen：lpSockAddr 的字节长度。

参数 nFlags：用来表示函数的实现，值为 MSG_OOB 和 MSG_PEEK 项的组合，通常默

认为 0。

- 禁止发送接收数据——ShutDown()

该函数用于禁止发送或者接收数据，如果调用成功返回非 0 值，否则返回 0 值。其函数原型如下：

```
BOOL ShutDown (int nHow=sends) ;
```

参数 nHow：不允许的操作，可以取值为 SD_RECENE、SD_SEND 或 SD_BOTH，这三个值的含义按照字面理解即可。

2. CSocket 类介绍

CSocket 类是 CAsyncSocket 类的派生类，它继承了 Windows Sockets API 封装函数。实现了比 CAsyncSocket 类对 Windows Sockets 更高层的抽象。它与类 CSocketFile 和 CArchive 共同合作完成对发送数据、接收数据的管理，CSocket 类提供了对于同步操作 CArchive 对象非常重要的阻塞功能，使程序员在管理数据的发送和接收的工作变得简单。

CSocket 类提供了阻塞的访问方式，这对于 CArchive 类的同步操作是必需的，其成员函数如 Receive()、Send()、ReceiveFrom()、SendToO 和 Accept()不会像 WinSock 中的函数一样返回错误，这些函数会等待直到操作完成。

CSocket 类的几个重要成员函数：

- 创建套接字并将其同一个对象绑定起来——Create()
- 确定一个阻塞是否正在执行——IsBlocking()

其函数原型如下：

```
BOOL IsBlocking();
```

如果套接字是阻塞状态返回非 0 值，否则返回 0 值。

- 返回一个指向 CSocket 对象的指针——FromHandle()

给定一个 Socket 句柄，返回一个指向 CSocket 的指针，如果没有 CSocket 对象绑定到这个句柄上，则该函数返回 NULL，并且不创建临时的对象。其函数原型如下：

```
static CSocket*    PASCAL    FromHandle( SOCKET    hSocket );
```

参数 hSocket 为套接字句柄。

- 把一个 Socket 句柄绑定到一个 CSocket 对象上——Attach()

该函数把一个 Socket 句柄绑定到一个 CSocket 对象上，SOCKET 句柄被保存到 CSocket 对象的成员变量 m_hSocket 中。如果调用成功返回非 0 值，否则返回 0 值。其函数原型如下：

```
BOOL    Attach(SOCKET hSocket) ;
```

参数 hSocket 为一个套接字的句柄。

- 取消一个正在进行中的阻塞调用——CancelBlockingCall()

调用该函数，原始的阻塞调用会立即终止并返回 WSAEINTR 错误。如果是阻塞的 Connect()操作，Windows 套接字可以实现立即终止这个阻塞的调用，但是不可能立即释放这个套接字的资源，必须等到连接完成或者超时后资源才会被释放。其函数原型如下：

```
void CancelBlockingCall();
```

- 过滤和响应 Windows 特定的消息——OnMessagePending()

该函数是一个重载函数，用来过滤和响应 Windows 特定的消息，是一个高级的可重载函数。如果 Windows 消息成功处理了则返回非 0 值，否则返回 0 值。其函数原型如下：

```
virtual BOOL OnMessagePending() ;
```

2.4.4 WinInet 编程技术

一个 Internet 客户端程序的目的是通过 Internet 协议如 HTTP、FTP 等来存取网络数据源（服务器）的信息。客户端程序可以访问服务器获得如天气预报、股票价格、重要新闻数据，甚至是与服务器交换信息。Internet 客户端程序可以通过外部网络（Internet）或内部网络（一般为 Intranet）访问服务器。

为了开发 Internet 客户端程序。MFC 类库提供了专门的 Win32 Internet 扩展接口，也就是 WinInet。MFC 将 WinInet 封装在一个标准的、易于使用的类集合中。在编写 WinInet 客户端程序时，你既可以直接调用 Win32 函数，也可以使用 WinInet 类库。

Win32 Internet 扩展提供了对普通 Internet 协议的访问，这些协议包括 HTTP、FTP 和 Gopher。Gopher 已经渐渐淡出。借助于 WinInet 编程接口，开发人员不必去了解 Winsock、TCP/IP 和特定 Internet 协议的细节就可以编写出高水平的 Internet 客户端程序。WinInet 为几种协议（HTTP、FTP 和 Gopher）提供了统一的函数集，也就是 Win32 API 接口。利用这些统一的函数集，大大简化了针对 HTTP、FTP 等协议的编程，从而轻松地将 Internet 集成到自己的应用程序中。

在 Visual C++工程中提供有两种方式来使用 WinInet：一种是直接调用 Win32 Internet 函数；另一种是使用 WinInet 类库。

MFC 对 WinInet 的封装是通过提供三个由 CStdioFile 派生类实现的。这三个派生类是：CInternetFile、CHttpFile 和 CGopherFile。对开发人员来说，不管你以前是否用过 CStdioFile，WinInet 都是很好理解并且易于使用的。它使得存取 Internet 数据易如反掌，使得 Internet 数据和本地数据的处理一致透明，数据的存储位置已经不再重要。

MFC WinInet 类有如下优点：

- 缓冲器输入输出。
- 数据的类型安全处理。
- 许多函数的参数都是缺省值。
- 对普通的 Internet 错误进行异常处理。
- 自动清除打开的句柄和连接。

WinInet 类包括 CInternetSession 等 10 多个类，类名及功能如表 2-3 所示。

表 2-3 WinInet 类名及功能介绍

类名	功能
CInternetSession （父类 CObject）	创建并初始化一个或几个同步 Internet 会话
CInternetConnection （父类 CObject）	管理一个到 Internet 服务器的连接
CFtpConnection	管理一个到 Internet 服务器的 FTP 连接及对外用户直接操作服务器上的文件和目录
CGopherConnection	管理一个到 Gopher 服务器的连接
CHttpConnection	管理一个到 HTTP 服务器的连接
CInternetFile（父类 CStdioFile）	使用 Internet 协议对远程系统的文件进行操作

续表

类名	功能
CGopherFile	在 Gopher 服务器上进行读取和查找文件等操作
CHttpFile	在 HTTP 服务器上进行读取和请求文件等操作
CFileFind（父类 CObject）	负责进行文件的查找
CFtpFileFind	在 FTP 服务器上查找文件
CGopherFileFind	在 Gopher 服务器查找文件
CGopherLocator（父类 CObject）	从 Gopher 服务器上得到定位器
CInternetException（父类 CException）	用于处理网络异常

WinInet 有 3 个全局函数：

- AfxParseURL：该函数用来解析一个 URL 字符串，并且返回服务类型。如果一个 URL 解析成功则返回非 0 值，如果为空 URL 或者不包含已知的服务类型则返回 0 值。
- AfxGetInternetHandleType：该函数用来判断一个网络服务类型，这些返回网络服务类型在 AFXINET.H 中已定义，如果 Handle 为空或者服务类型无法识别，则返回 AFX_INET_SERVICE_UNK。
- AfxThrowInternetException：该函数用于处理网络异常。

这些类和全程函数除 CFileFind 在 AFX.H 里声明之外，其余都在 AFXINET.H 文件里声明。它们对 HTTP、FTP 和 Gopher 等协议进行了高度抽象，形成了一套高级 API 函数。利用这些 API 可以快速直接地开发 Internet 应用。

例如，连接到 FTP 服务器一般需要几个步骤，而且需要做一些底层处理。但使用上述的 MFC 类提供的 API，只需要对 CInternetSession::GetFTPConnection 进行一次调用，便可以轻松建立连接。

2.4.5 本章涉及的 MFC 常用类和控件

1．CString 类

MFC 库的 CString 类是 C++语言的一个很重要的扩展，CString 类有许多非常有用的操作和成员函数，但最重要的一个特点莫过于它的动态内存分配，完全不用担心 CString 对象的大小。

CString 类的常用成员函数：

① int GetLength() const;

说明：获取 CString 类对象包含字符串的长度（字符数）。

② BOOL IsEmpty() const;

说明：测试 CString 类对象包含的字符串是否为空。

③ void Empty();

说明：使 CString 类对象包含的字符串为空字符串。

④ TCHAR GetAt(int nIndex)const;

说明：获得字符串指定位置处的字符。

⑤ void SetAt(int nIndex, TCHAR ch);

说明：设定字符串指定位置处的字符。

⑥ CString Left(int nCount)const;

说明：获取字符串左边 nCount 长度的字符串。

⑦ CString Right(int nCount)const; throw(CMemoryException);

说明：获取字符串右边 nCount 长度的字符串。

⑧ MakeUpper

说明：将字符串中所有的字符全部转化成大写形式。

⑨ MakeLower

说明：将字符串中所有的字符全部转化成小写形式。

⑩ MakeReverse

说明：将字符串倒置，即将字符的顺序颠倒，第一个字符变成最后一个字符。

⑪ int Replace(TCHAR chOld, TCHAR chNew);

int Replace(LPCTSTR lpszOld, LPCTSTR lpszNew);

说明：将字符串中的字符子串 lpszOld 替换成新的字符串 lpszNew。

⑫ Find

原型：

int Find(TCHAR ch)const;

int Find(LPCTSTR lpszSub)const;

int Find(TCHAR ch, int nStart)const;

int Find(LPCTSTR pstr, int nStart)const;

说明：在字符串中查找指定的字符或字符串。参数 ch 为要查找的字符；lpszSub 为要查找的字符子串；nStart 指定查找的起始位置，如缺省为从最左边开始；pstr 指向欲查找子串的指针。

⑬ ReverseFind

原型：int ReverseFind(TCHAR ch)const;

说明：返回字符串中最后一个和指定的字符匹配的字符的下标。

⑭ Compare

原形：int Compare(LPCTSTR lpsz)const;

说明：比较两个字符串，如果两个字符串相等，返回值等于 0；如果本对象大于参数字符串，返回值大于 0；如果本对象小于参数字符串，返回值小于 0，比较时区分大小写。

⑮ operator =

说明：将一个新的值赋予 CString 对象。

⑯ operator +

说明：将两个字符串合并成一个新的字符串。在两个参数中必须有一个是 CString 类型的，而另一个参数可以是字符、字符指针或 CString 类型对象。

⑰ operator +=

说明：在一个字符串的后面再添加一个字符串或一个字符。

2. 常用控件——按钮、编辑框和列表框

（1）按钮（CButton 类）。

按钮是为数不多的、可以说是每个应用程序都要用到的 Windows 控件之一。即使是在一个简单对话框显示信息的应用程序中，通常也要在用完对话框后关闭它时使用 OK 按钮。

生成按钮过程很简单，只要在 VC 界面上的“控件”浮动面板中鼠标选择按钮，拖动到界面上即可。鼠标右键单击按钮，可以选择修改按钮的属性。鼠标左键双击按钮，VC 向导会自动帮助按钮添加按钮单击事件处理函数（也可以在“建立类向导”中添加事件处理）。

（2）编辑框（CEdit）。

编辑框是用来接收用户输入最常用的一个控件。在 VC 界面上的“控件”浮动面板中鼠标选择编辑框，拖动到界面上即可添加编辑框。因为编辑框通常要接收用户输入数据，或者将程序生成的数据显示到界面上的编辑框中，所以需要在程序和编辑框之间进行数据交换，这就需要给编辑框添加一个关联的成员变量。在菜单“查看”里选择“建立类向导”，弹出“建立类向导”对话框，在对话框中选择“Member Variables”选项，然后单击右侧的添加按钮，给指定的编辑框添加成员变量。用户输入此编辑框的数据将存放在该成员变量中，程序也可以赋值给该成员变量，然后把它显示到界面上。此处用到一个函数 UpdateData。

函数 UpdateData 原型：

```
BOOL UpdateData(BOOL bSaveAndValidate=TRUE);
```

参数 bSaveAndValidate：值为 true，函数功能是将界面上用户输入编辑框的数据保存到变量中；值为 false，函数功能是将数据在界面上对应编辑框中显示出来。

（3）列表框（CListBox）。

列表框可以用来列出一系列的文本，每条文本占一行。在 VC 界面上的“控件”浮动面板中鼠标选择列表框，拖动到界面上即可添加列表框对象。为了使用列表框的各种功能，首先需要给列表框添加一个关联的成员变量，添加方法与编辑框相同，需要注意的是，列表框添加的成员变量类型要选择 CListBox 类型。通过该成员变量，即可调用列表框的成员函数。最常用的成员函数就是向列表框中添加文本的函数。

在列表框类（CListBox）中有两个“长的有点相似的”成员函数：

```
int    InsertString (int nIndex, LPCTSTR lpsczItem)
int    AddString (LPCTSTR    lpsczItem)
```

这两个函数都可被用来向列表中添加项目条。不同之处在于，InsertString 函数有两个参数，第一个参数为索引号，设定为-1 时，项目条被插入到列表的末尾；第二个参数与 AddString 函数的唯一参数相同，为代表项目条中内容的字符串。使用 InsertString，第一个参数设定为-1 时，列表中的项目条的排列顺序严格按照源代码中语句的顺序排列；使用 AddString 时，列表中的项目条的顺序则按照英文字典序由上而下排列。

2.5 本章技能训练实例

实验题目：简单的考试报名系统实现。

实验目的：掌握使用 Socket 进行通信的基本方法和简单控件编程。

实验内容：开发一个服务器端程序，能够接收客户端发来的信息；开发一个客户端程序，能够将信息发送给服务器端，可采用 CSocket 编程，也可以用 WinSock 编程。

实验步骤：打开两个 Visual C++窗口，分别建立两个应用程序 MyServer 和 MyClient，它们的界面如图图 2-12 和图 2-13 所示。

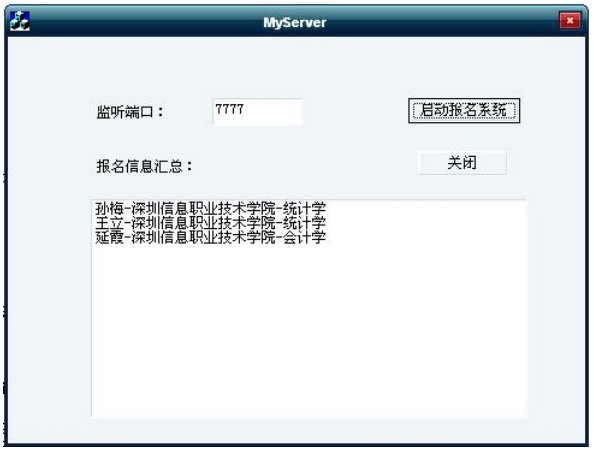

图 2-12　服务器端界面

图 2-13　客户端界面

（1）服务器端应用程序开发。

界面中有一个文本编辑框，一个让用户指定监听端口号，对文本编辑框添加关联的成员变量 m_myPort。

界面下面是一个列表框，给列表框添加一个关联的成员变量 m_show，变量的类型是 CListBox。列表框用来显示所有用户的报名信息。编程思路如下：

1）定义两个 Socket，一个负责监听，一个负责与客户端建立连接。

2）创建监听 Socket，并指明监听端口。注意：端口号来自界面的文本框的输入。

3）当接到客户请求时，建立连接，此处进入循环。

4）接受客户发送的消息，从客户发送的消息里解析出用户的信息，添加到界面上的列表框里。

5）发送消息给客户。

6）关闭 Socket。

（2）客户端应用程序开发。

界面上有 5 个文本编辑框，第一个让用户输入服务器 IP 地址，第二个让用户输入服务器的监听端口号，第三个让用户输入姓名，第四个让用户输入单位，第五个让用户输入考试科目。你需要给每个编辑框添加一个关联的成员变量，名字分别是 m_ServerIP、m_myPort、m_name、m_code、m_item。编程思路如下：

1）定义一个 CSocket 对象。

2）创建 CSocket 对象。

3）连接服务器，指明服务器的 IP 地址和服务器的监听端口。

注意：IP 地址和端口号都来自界面用户的输入。

4）获取文本编辑框中输入的信息，并发送消息给服务器。

5）接收服务器的消息，并显示到文本编辑框中。

6）关闭 Socket 对象。

一、填空题

1．目前最通用的网络编程接口就是_____。

2．建立套接字后，需要为服务器端定义的这个监听的套接字指定一个地址及监听端口（Port），这样客户端才知道要连接哪一个地址的哪个端口，为此需要调用_____函数。

3．对于服务器编程中最重要的一步等待并接受客户的连接，_____函数就完成了这个功能。

4．完成面向连接的协议的连接功能的函数是_____。

5．CSocket 类是_____的派生类，继承了它封装的 WinSock API。

6．套接字有同步阻塞方式和_____两种使用方法。

7．MFC 中有两个主要的网络编程类：CAsyncSocket 类和_____类。

二、选择题

1．在服务器端建立一个套接字，并进入实际的监听步骤的过程是（　　）。

A．socket()->listen()->accept()

B．socket()->bind()->listen()->accept()

C．socket()->create()->listen()->accept()

D．socket()->listen()->connect()->accept()

2．在客户端调用 WinSock 的过程是（　　）。

A．socket()->connect()->send()

B．socket()->bind()->accept()

C．socket()->listen()->accept()

D．socket()->bind()->connect ()

3．CAsyncSocket 类和 CSocket 类定义在头文件（　）中。

A．socket.h　　B．afxsock.h　　C．sock.h　　D．winsock.h

三、简答题

1．什么是面向连接的网络通信？

2．什么叫主机字节顺序？什么叫网络字节顺序？

3．什么是 MFC？

4．分析阻塞方式和非阻塞方式的优缺点。