华中科技大学网络空间安全学院

《计算机通信与网络》

实验指导手册

（Socket 编程实验分册）

华中科技大学网络空间安全学院

二零二零年十一月

目　录

第一章 实验目标和内容

1.1 实验目的

- ✧ 了解应用层和运输层的基本功能和作用
- ✧ 了解掌握基本的可靠数据传输的原理和机制。
- ✧ 掌握 SOCKET 编程的基本方法。

1.2 实验环境

- ✧ 操作系统：Windows
- ✧ 语言：C
- ✧ 编程开发环境：Visual Studio 2008-2017 皆可

1.3 实验要求

- ✧ 必须基于 Socket 编程，不能直接借用任何现成的组件、封装的库等。
- ✧ 提交实验设计报告和源代码；实验设计报告必须包括程序流程图，源代码必须加详细注释。
- ✧ 实验设计报告需提交纸质档和电子档，源代码、编译说明需提交电子档。
- ✧ 基于自己的实验设计报告，通过实验课的上机试验，将源代码编译成功，运行演示给实验指导教师检查。

1.4 实验内容

完成一个 TFTP 客户端程序。TFTP 是一种简单的文件传输协议。目标是在 UDP 之上上建立一个类似于 FTP 的但仅支持文件上传和下载功能的传输协议

题目：编写实现一个 TFTP 客户端程序，要求如下：

- ✧ 严格按照 TFTP 协议与标准 TFTP 服务器通信；
- ✧ 能够实现两种不同的传输模式 netascii 和 octet；
- ✧ 能够将文件上传到 TFTP 服务器；
- ✧ 能够从 TFTP 服务器下载指定文件；
- ✧ 能够向用户展现文件操作的结果：文件传输成功/传输失败；
- ✧ 针对传输失败的文件，能够提示失败的具体原因；
- ✧ 能够显示文件上传与下载的吞吐量；
- ✧ 能够记录日志，对于用户操作、传输成功，传输失败，超时重传等行为记录日志；
- ✧ 人机交互友好（图形界面/命令行界面均可）；

说明：额外功能的实现，将视具体情况予以一定加分。

第二章　TFTP 协议

2.1 TFTP 协议简介

TFTP（Trivial File Transfer Protocol,简单文件传输协议）是 TCP/IP 协议族中的一个用来在客户机与服务器之间进行简单文件传输的协议，提供不复杂、开销不大的文件传输服务，端口号为 69。

TFTP 通常基于 UDP 协议而实现，但是我们也不能确定有些 TFTP 协议是基于其它传输协议完成的。TFTP 协议的设计目的主要是为了进行小文件传输，因此它不具备通常的 FTP 的许多功能，例如，它只能从文件服务器上获得或写入文件，不能列出目录，不进行认证。

TFTP 代码所占的内存较小，这对于较小的计算机或者某些特殊用途的设备来说是很重要的，这些设备不需要硬盘，只需要固化了 TFTP、UDP 和 IP 的小容量只读存储器即可。因此，随着嵌入式设备在网络设备中所占的比例的不断提升，TFTP 协议被越来越广泛的使用。

2.2 TFTP 的包格式

TFTP 共定义了五种类型的包，包的类型由数据包前两个字节确定，我们称之为 Opcode（操作码）字段。这五种类型的数据包分别是：

- 读文件请求包：Read request，简写为 RRQ，对应 Opcode 字段值为 1
- 写文件请求包：Write requst，简写为 WRQ，对应 Opcode 字段值为 2
- 文件数据包：Data，简写为 DATA，对应 Opcode 字段值为 3
- 回应包：Acknowledgement，简写为 ACK，对应 Opcode 字段值为 4
- 错误信息包：Error，简写为 ERROR，对应 Opcode 字段值为 5

RRQ 和 WRQ 的数据包格式一样，只不过某些值域设置有差别，剩下的三种数据包格式各不相同。

（1）RRQ 和 WRQ 数据包的格式，首先是 2 字节表示操作码，它用来表示当前数据包的类型（取值 1 表示该数据包是个读请求，2 表示该数据包是写请求）；接下来是可变长字段，它用来表示要读取或上传的文件名，它使用 ASCII 码并以 0 表示结尾；第三个字段叫 Mode，也是可变长字段，用来表示传输文件的数据类型，如果传输的是字符串文件，那么它填写字符串''netascii''，如果传输的是二进制文件，那么它填写字符串''octet'',这些字符串都以 0 结尾，其结构如图 2.1 所示：

图 2.1 TFTP 协议 RRQ/WRQ 数据报格式

（2）传输数据块的 DATA 数据包，它头 2 字节也是操作码，取值 3 用于表示数据包用于数据块传输，接下来的 2 字节用于表示数据块编号，最后是可变长字段 Data，用于装载数据块，该数据包的格式如图 2.2 所示：

图 2.2 TFTP 协议 DATA 数据包格式

（3）应答 ACK 数据包，它开始的 2 字节也是操作码，取值 4；接下来 2 字节表示接收到的数据块编号，相应结构如图 2.3 所示：

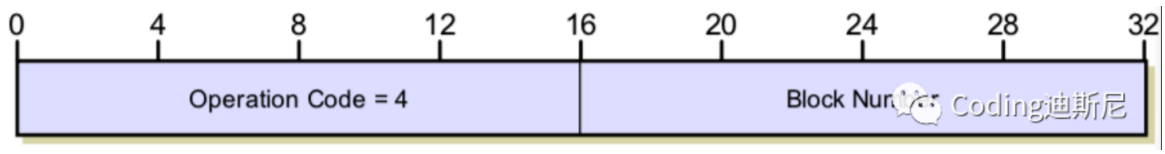


图 2.3 TFTP 协议 ACK 数据包格式

（4）错误 ERROR 数据包，它开始的 2 字节表示操作码，取值 5；接下来 2 字节表示错误码；最后的是可变长字段，它用字符串的形式描述具体错误，该数据包的结构如图 2.4 所示：

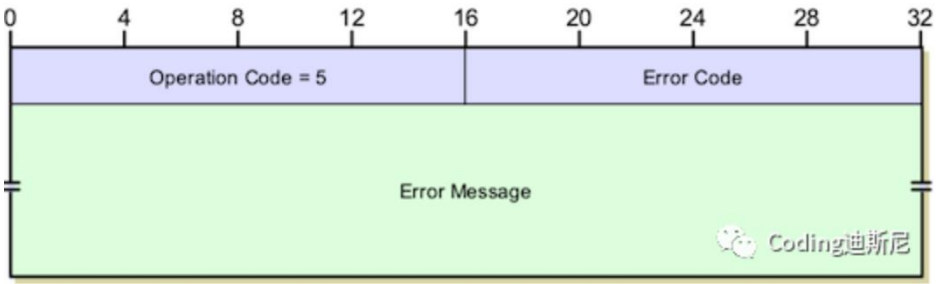


图 2.4 TFTP 协议 ERROR 数据包格式

TFTP 协议目前定义了 8 种错误码，具体的错误码以及对应的错误信息如表 2.1 所示：

表 2.1 TFTP 协议错误码信息

错误码	表示的意思
0	未定义 Not defined, see error message
1	文件未找到 File not found
2	访问被拒绝 Access violation
3	磁盘满或超出可分配空间 Disk full or allocation exceeded
4	非法的 TFTP 操作 Illegal TFTP operation
5	未知的传输 ID Unknown transfer ID
6	文件已经存在 File already exists
7	没有该用户 No such user

2.3 TFTP 的工作流程

TFTP 的工作都是由客户端发起一个 RRQ 或者 WRQ 开始的。这里分别以 WRQ 和 RRQ 为例，讲述读写的工作过程，以及错误处理等内容。

用 S 表示 Server，C 表示 Client，典型的 WRQ 工作流程如图 2.5 所示。

1、WRQ 工作流程

- S 在端口为 69 的 UDP 上等待 C 发出写文件请求包
- C 通过 UDP 发送符合 TFTP 请求格式的 WRQ 包给 S。从 UDP 包角度看，该 UDP 包的源端口由 C 随意选择，而目标端口则是 S 的 69。
- S 收到 C 的这个请求包后，需发送 ACK 给 C。对于写请求包，S 发送的 ACK 包确认号为 0。
- C 发送 DATA 数据给 S，S 接收数据并写文件
  - 当 C 发送的 DATA 数据长度小于 512 字节时，S 认为这次 WRQ 请求完成

这里我们要明确一点，如果有多个 C 同时向 S 发起请求的话，S 如何正确发送包到对应的 C 呢。

在 TFTP 中，一次请求中所有包的源和目标都由 Transfer ID(TID)来标示。TFTP 规定 TID 值就是 UDP 包中的源和目标端口。也就是说，一次请求过程中，S 和 C 通过 UDP 包的源和目标端口来判断这个包是不是发给自己的。

以 WRQ 为例，C 向 S 的 69 端口发送一个文件请求包，这个文件请求包中 UDP 的源端口号为 C 的 TID（假设 C 选择 4845 作为它的 TID），目标端口为 69（这个时候由于请求还未接受，所以这次请求的 UDP 包中目标端口不是 TID）。S 收到这个请求后，将另外采用一个 UDP 端口（应该另启动了一个 UDP Socket）假设为 4849 来回复这个请求的 ACK。这样，这个回复的 UDP 包的源端口就是 S 的 TID（=4849），目标就是 C 的 UDP 端口（TID=4845）。以后，这次请求的后续所有包都在端口为 4845 和 4849 中来往。

上述过程隐含了一定程度上的容错处理。例如，C 收到一个 TID 不是 4849 的包，则认为这个包是错误的。

另外，S 对于每个请求，都要采用一个不重复的新的 UDP 端口号作为它的 TID，也就是说，S 上同时存在的 n 个请求的 TID 都将不同。

这里再介绍下 TFTP 的回复 ACK 机制。虽然 TFTP 中有指定的 ACK 包作为回应，但在普遍意义上，DATA 包和 ERROR 包都可以作为上一次发送包的响应。

一般来说，C 发送了一个非结束 DATA 包给 S，如果在超时时间内，C 未收到 S 发送的 ACK，则 C 继续发送这个 DATA 直到 S 回复 ACK。这种情况是比较好理解的。

但假如 S 回复了上一个非结束 DATA 包 ACK 后，C 在 S 的超时时间内没有发送下一个 DATA 包，则 S 将继续发送这个 ACK。从这个角度看，S 等待的这个新 DATA 包是对上一次 ACK 的确认。

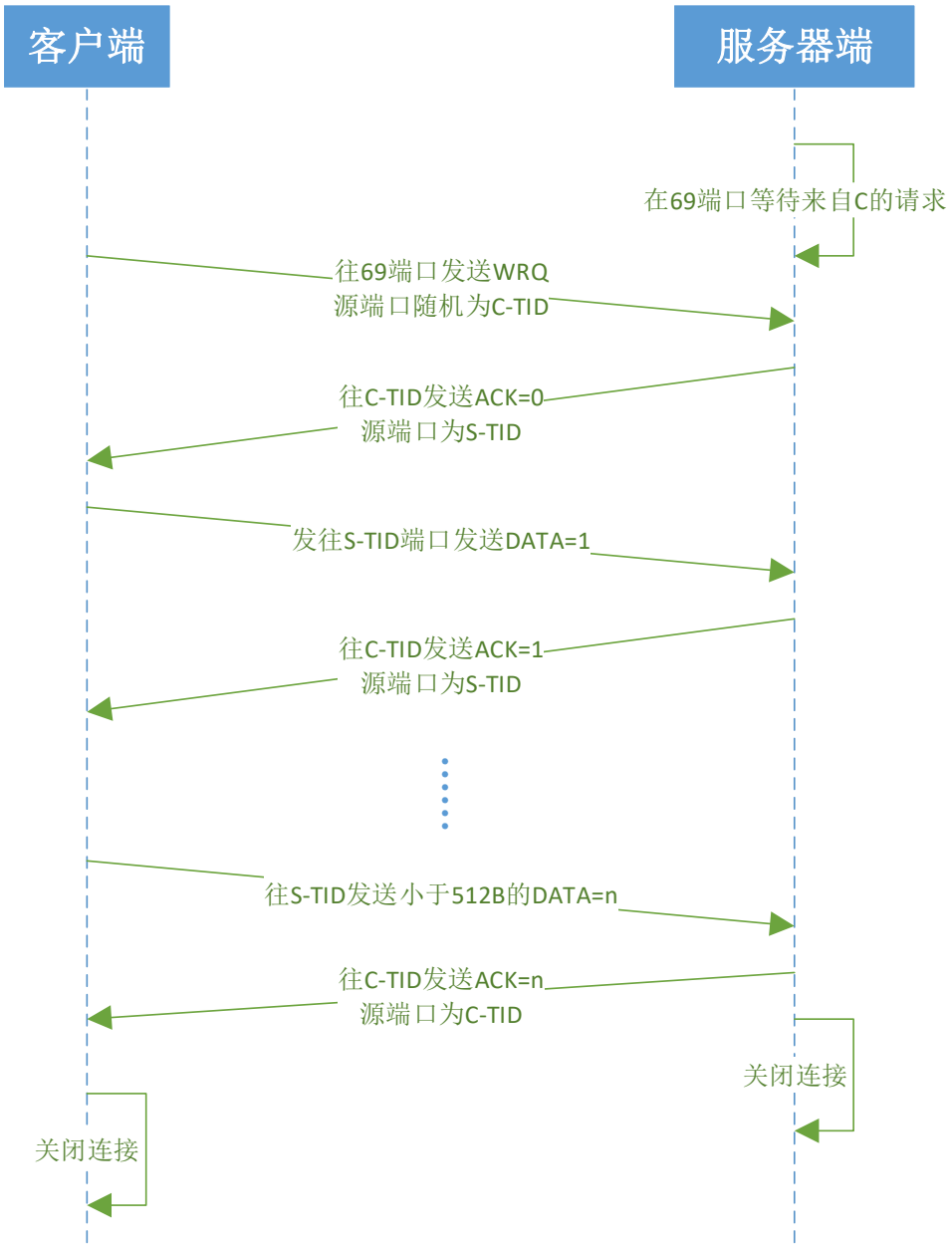


图 2.5 典型的 WRQ 工作流程

2、 RRQ 的工作流程

RRQ 的工作流程和 WRQ 类似，典型的工作流程如图 2.6 所示。

- S 在端口为 69 的 UDP 上等待 C 发出读文件请求包、
- C 通过 UDP 发送符合 TFTP 请求格式的 RRQ 包给 S。
- S 收到 C 的这个请求包后，将直接发送 DATA 包给 C，这个 DATA 包中含 S 选择的 TID 作为 UDP 的源端口和 C 的 TID 作为 UDP 目标端口，起始包号为 1。
- C 接收来自 S 的 DATA 包并回复 ACK。直到请求完成

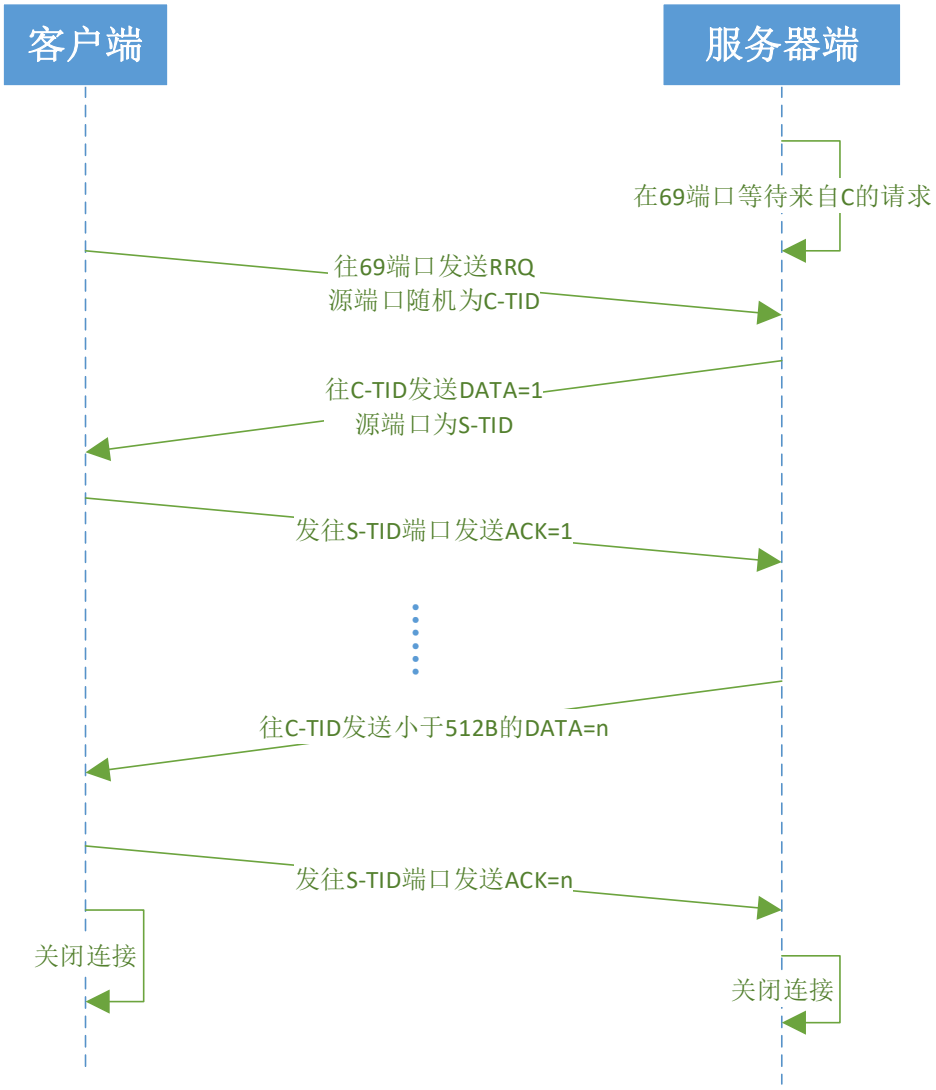


图 2.6 典型的 RRQ 工作流程

3、连接和错误处理

UDP 实际上没有连接的概念。但从上面分析的 RRQ 和 WRQ 看，S 在 69 端口上等待请求，而且 S 总是生成一个新的 UDP 来完成和 C 的交互。这个过程和 TCP 的 listen 以及 Accept 非常类似。所以 TFTP 把这种交互也称作 connection，只不过这种连接是隐含在请求中的。

一般情况下，连接的建立由一次成功的请求来发起，当最后一个 DATA 包发送完毕并且 ACK 回复了后，则连接正常关闭。在传输过程中，如果出现错误，假设 S 向 C 发送了一个 ERROR 包，如果 C 收到 ERROR 包，则连接关闭。如果 C 没有收到 ERROR 包，则需要启动 ERROR 超时检测机制。需要强调的是对于 ERROR 包，S 和 C 都不会重传也不需要 ACK 确认。

TFTP 建议在连接正常关闭的情况下，S 可在发送确认结束 DATA 包的 ACK 后稍等片刻后再关闭连接。例如，当 C 发送结束 DATA 包后，S 回复 ACK 后再等一段时间才关闭。再次等待时间中，如果 ACK 包丢失，C 将再次发送结束 DATA 包或者超时处理。S 如果又收到一次结束 DATA 包后，就知道 ACK 包丢失了。S 可以关闭连接也可以再次发送 ACK 包。

2.4 TFTP 的传输模式

TFTP 传输 8 位数据，传输中有三种模式：

- Netascii：这是 8 位的 ASCII 码形式，一般用来传输字符数据；
- Octet：这是 8 位源数据类型，一般用来传输二进制数据；
- Mail：它将返回的数据直接返回给用户而不是保存为文件，但该模式已经不再支持。

第三章 Windows Socket 1.1 编程简介

3.1 Socket 套接字介绍

网络应用程序是由通信进程对组成，每对互相通信的应用程序进程互相发送报文，他们之间的通信必须通过下面的网络来进行。为了将应用程序和底层的网络通信协议屏蔽开来，采用套接字（Socket）这样一个抽象概念来作为应用程序和底层网络之间的应用程序编程接口（API）。

因为网络应用程序是进程之间的通信，为了唯一的标识通信对等方的通信进程，套接字必须包含 2 种信息：(1) 通信对等方的网络地址。(2) 通信对等方的进程号，通常叫端口号。

就像 Unix 操作系统下有一套实现 TCP/IP 网络通信协议的开发接口：BSD Sockets 一样，在 Windows 操作系统下，也提供了一套网络通信协议的开发接口，称为 Windows Sockets 或简称 Winsock。

Winsock 是通过动态链接库的方式提供给软件开发者，而且从 Windows 95 以后已经被集成到了 Windows 操作系统中。

Winsock 主要经历了 2 个版本：Winsock 1.1 和 Winsock 2.0。Winsock 2.0 是 Winsock 1.1 的扩展，它向下完全兼容。

Winsock 同时包括了 16 位和 32 位的编程接口，16 位的 Windows Socket 2 应用程序使用的动态链接库是 WINSOCK.DLL，而 32 位的 Windows Socket 应用程序使用 WSOCK32.DLL（Winsock 1.1 版）和 WS2_32.DLL（Winsock 2.0 版）。另外，使用 Winsock API 时要包含头文件 winsock.h（Winsock 1.1 版）或 winsock2.h（Winsock 2.0 版）。

3.2 Socket 套接字编程原理

3.2.1 Socket 的 2 种类型

Socket 是一个抽象概念，代表了通信双方的端点（Endpoint），通信双方通过 Socket 发送或接收数据。

在 Winsock 里，用数据类型 SOCKET 作为 Windows Sockets 对象的句柄，就好像一个窗口的句柄 HWND、一个打开的文件的文件指针一样。下面我们会看到，在 Winsock API 的许多函数里，都会用到 SOCKET 类型的参数。

Socket 有 2 种类型：

- 流类型（Stream Sockets）。

  流式套接字提供了一种可靠的、面向连接的数据传输方法，使用传输控制协议 TCP。

- 数据报类型（Datagram Sockets）。

数据报套接字提供了一种不可靠的、非连接的数据包传输方式，使用用户数据报协议 UDP。

3.2.2 Socket I/O 的 2 种模式

一个 SOCKET 句柄可以看成代表了一个 I/O 设备。在 Windows Sockets 里，有 2 种 I/O 模式：

- 阻塞式 I/O（blocking I/O）

在阻塞方式下，收发数据的函数在调用后一直要到传送完毕或者出错才能完成，在阻塞期间，除了等待网络操作

的完成不能进行任何操作。阻塞式 I/O 是一个 Winsock API 函数的缺省行为。

- 非阻塞式 I/O（non-blocking I/O）

对于非阻塞方式，Winsock API 函数被调用后立即返回；当网络操作完成后，由 Winsock 给应用程序发送消息（Socket Notifications）通知操作完成，这时应用程序可以根据发送的消息中的参数对消息做出响应。Winsock 提供了 2 种异步接受数据的方法：一种方法是使用 BSD 类型的函数 select（），另外一种方法是使用 Winsock 提供的专用函数 WSAAsyncSelect（）。

3.2.3 使用数据报套接字

首先，客户机和服务器都要创建一个数据报套接字。接着，服务器调用 bind（）函数给套接字分配一个公认的端口。一旦服务器将公认的端口分配给了套接字，客户机和服务器都能使用 sendto（）和 revfron（）来传递数据报。通信完毕调用 closesocket（）来关闭套接字。流程如图 2.1 所示：

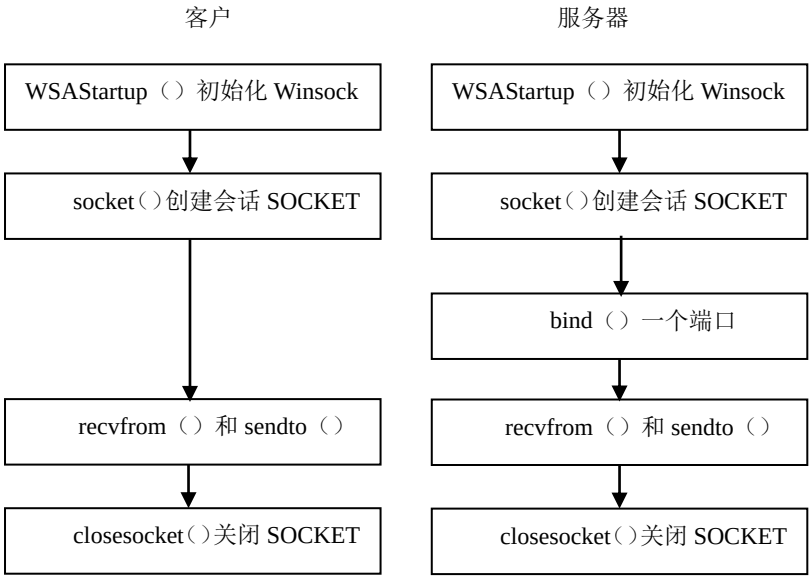


图 3.1 面向无连接的数据报方式过程

3.2.4 使用流式套接字

由于流式套接字使用的是基于连接的协议，所以你必须首先建立连接，而后才能从数据流中读出数据，而不是从一个数据报或一个记录中读出数据，其流程如图 2.2 所示。

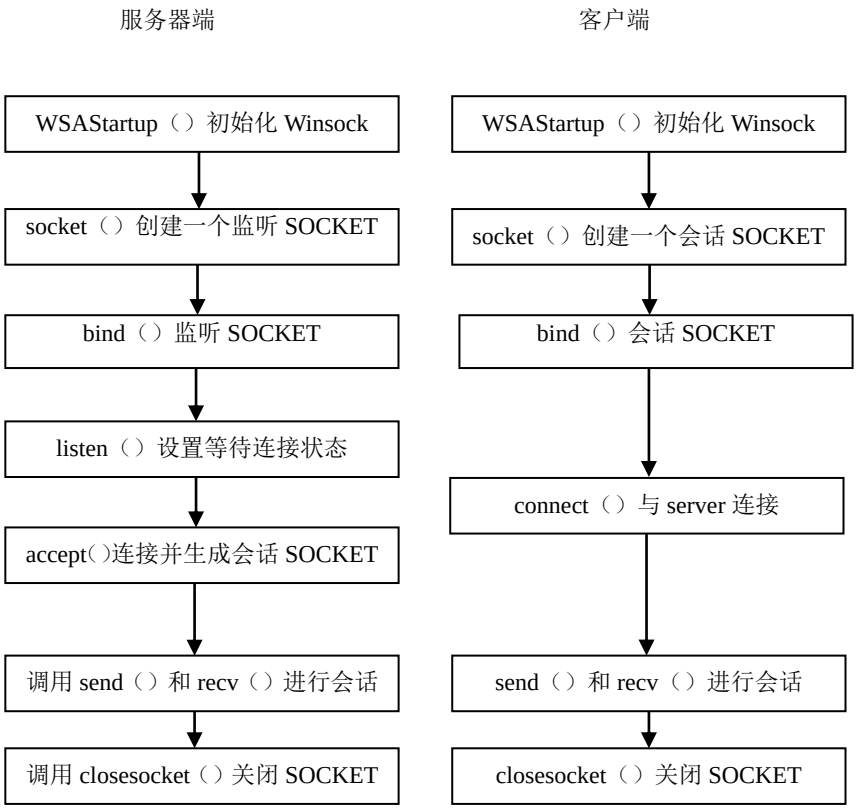


图 3.2 面向连接的流方式过程

第四章 套接字部分库函数列表

4.1 WSAStartup（）

【函数原型】

int WSAStartup (**WORD** wVersionRequested,

LPWSADATA lpWSAData);

【参数】

wVersionRequested

[in] 表示欲使用的 Windows Sockets API 版本；这是个 WORD 类型的整数，高字节定义的是次版本号，低字节定义的是主版本号。

lpWSAData

[in] 指向 WSAData 资料的指针。WSAData 是结构数据类型，描述了关于 Windows Sockecs 底层实现的相关信息。

【返回值】

函数执行成功返回 0，失败则返回如下错误代码：

WSASYSNOTREADY： 底层网络子系统没有准备好。

WSAVERNOTSUPPORTED：Winsock 版本信息号不支持。**WSAEINPROGRESS**： 阻塞式 Winsock1.1 存在于进程中。

WSAEPROCLIM： 已经达到 Winsock 使用量的上限。

WSAEFAULT： lpWSAData 不是一个有效的指针。

【函数功能】

这个函数是应用程序应该第一个调用的 Winsock API 函数，以完成一系列初始化的工作。

【相关数据结构】

WSADATA 的定义如下：

```
typedef struct WSAData {
    WORD    wVersion;
    WORD    wHighVersion;
    char    szDescription[WSADESCRIPTION_LEN+1];
    char    szSystemStatus[WSASYS_STATUS_LEN+1];
    unsigned short    iMaxSockets;
    unsigned short    iMaxUdpDg;
    char FAR *        lpVendorInfo;
} WSADATA, FAR * LPWSADATA;
```

其中，各结构成员的含义为:

wVersion

应用程序应该使用的 Winsock 版本号。

wHighVersion

DLL 所支持的最高版本号。通常应该等于 wVersion。

szDescription

以 0 结尾的 ASCII 字符串，关于 Winsock 底层实现的描述信息。

szSystemStatus

以 0 结尾的 ASCII 字符串，关于 Winsock 底层状态或者配置信息。

iMaxSockets

一个进程最多可使用的套接字数，仅用于 Winsock1.1，Winsock 2.0 应该忽略该成员。

iMaxUdpDg

最大的 UDP 报文大小，仅用于 Winsock1.1，Winsock 2.0 应该忽略该成员。对于 Winsock 2.0，应该使用 getsockopt 函数取得 SO_MAX_MSG_SIZE。

lpVendorInfo

Winsock 开发厂商信息，，仅用于 Winsock1.1，Winsock 2.0 应该忽略该成员。对于 Winsock 2.0，应该使用 getsockopt 函数取得 PVD_CONFIG。

【示例代码】

```
#include <winsock.h>
//对于 Winsock 2, include <winsock2.h>

WSADATA    wsaData;
int nRc = WSAStartup(0x0101, & wsaData);
if(nRc)
{
  //Winsock 初始化错误
  return;
}
if(wsaData.wVersion != 0x0101)
{
  //版本支持不够
  //报告错误给用户，清除 Winsock，返回
  WSACleanup();
  return;
}
```

4.2 socket（）

【函数原型】

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

【参数】

af

[in] 指定地址族（address family），一般填 AF_INET（使用 Internet 地址）。

type

[in] 指定 SOCKET 的类型：SOCK_STREAM（流类型），SOCK_DGRAM（数据报类型）。

protocol

[in] 指定 af 参数指定的地址族所使用的具体一个协议。建议设为 0，那么它就会根据地址格式和 SOCKET 类型，自动为你选择一个合适的协议。另外 2 个常用的值为：IPPROTO_UDP 和 IPPROTO_TCP。

【返回值】

函数执行成功返回一个新的 SOCKET，失败则返回 INVALID_SOCKET。这时可以调用 WSAGetLastError 函数取得具体的错误代码。

【函数功能】

所有的通信在建立之前都要创建一个 SOCKET。

【示例代码】

```
//创建数据报 socket
SCOKET udpSock = socket(AF_INET,
                        SOCK_DGRAM, IPPROTO_UDP);
//创建流 socket
SCOKET tcpSock = socket(AF_INET,
                        SOCK_STREAM, IPPROTO_TCP);
```

4.3 bind（）

【函数原型】

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

【参数】

s

[in] 一个需要绑定的 SOCKET，例如用 socket 函数创建的 SOCKET。

name

[in] 指向描述通信对象地址信息的结构体 sockaddr 的指针。在该结构体中可以指定地址族（一般为 AF_INET）、主机的地址和端口。通常把主机地址指定为 INADDR_ANY（一个主机可能有多个网卡）。

namelen

[in] name 指针指向的结构体的长度。

【返回值】

函数执行成功返回 0，失败则返回 SOCKET_ERROR。这时可以调用 WSAGetLastError 函数取得具体的错误代码。

【函数功能】

成功地创建了一个 SOCKET 后，用 bind 函数将 SOCKET 和主机地址绑定。

【相关数据结构】

```
struct sockaddr {
    u_short     sa_family;
    char        sa_data[14];
};
```

sa_family

地址族，比如 AF_INET，2 个字节大小。

sa_data

用来存放地址和端口，14 个字节大小。

sockaddr 结构是一个通用的结构（因为 Winsock 支持的协议族不只是 TCP/IP）。对 TCP/IP 协议，用如下结构来定义地址和端口。

```
struct sockaddr_in {
    short           sin_family;
    u_short         sin_port;
    struct  in_addr sin_addr;
    char            sin_zero[8];
};
```

sin_family

地址族，设为 AF_INET。

sin_port

端口号。如果端口号为 0，Winsock 会自动为应用程序分配一个值在 1024-5000 间的一个端口号，所以客户端一般把 sin_port 设为 0。

sin_addr

为 in_addr 结构类型，用来指定 IP 地址。通常把主机地址指定为 INADDR_ANY（一个主机可能有多个网卡）。结构 in_addr 下面介绍。

sin_zero

8 字节的数组，值全为 0。这个 8 个字节用来填充结构 sockaddr_in，使其大小等于结构 sockaddr（16 字节）。

结构 in_addr 用来指定 IP 地址，其定义为：

```
struct in_addr {
    union {
            struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
            struct { u_short s_w1,s_w2; } S_un_w;
            u_long S_addr;
    } S_un;
};
```

对于 IP 地址 10.14.25.90，sockaddr_in 结构中的 sin_addr 可以这样赋值：

```
sin_addr. S_un .S_un_b. s_b1 = 10;
sin_addr. S_un .S_un_b. s_b2 = 14;
sin_addr. S_un .S_un_b. s_b3 = 25;
sin_addr. S_un .S_un_b. s_b4 = 90;
```

或者

```
sin_addr. S_un . S_un_w. s_w1 = (14<<8)|10;
sin_addr. S_un . S_un_w. s_w2 = (90<<8)|25;
```

或者

```
sin_addr. S_un . S_addr = (90<<24)|(25<<16)|(14<<8)|10;
```

或者

sin_addr. S_un . S_addr = inet_addr(“10.14.25.90”);

这里的 inet_addr 函数可以将字符串形式的 IP 地址转换为 unsigned long 形式的值。

【示例代码】

```
SOCKET sServSock;
sockaddr_in addr;

//创建 socket
sServSock = socket(AF_INET, SOCK_STREAM, 0);

addr.sin_family = AF_INET;

//htons 和 htonl 函数把主机字节顺序转换为网络字节顺序，分别用于//短整型和长整型数据
addr.sin_port = htons(5050);
addr.sin_addr.S_un.S_addr = htonl(INADDR_ANY);

// LPSOCKADDR 类型转换是必须的
int nRc = bind(sServSock, (LPSOCKADDR)&addr, sizeof(addr) );
```

4.4 listen（）

【函数原型】

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

【参数】

s

[in] 一个已经绑定但未连接的 SOCKET。

backlog

[in] 等待连接的队列的长度，可取 SOMAXCONN。如果某个客户程序要求连接的时候，服务器已经与其他客户程序连接，则后来的连接请求会放在等待队列中，等待服务器空闲时再与之连接。当等待队列达到最大长度（backlog 指定的值）时，再来的连接请求都将被拒绝。

【返回值】

函数执行成功返回 0，失败则返回 SOCKET_ERROR。这时可以调用 WSAGetLastError 函数取得具体的错误代码。

【函数功能】

对于服务器的程序，当申请到 SOCKET,并将通信对象指定为 INADDR_ANY 之后，就应该等待一个客户机的程序来要求连接，listen 函数就是把一个 SOCKET 设置为这个状态。

4.5 accept（）

【函数原型】

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

【参数】

s

[in] 一个已经处于 listen 状态的 SOCKET。

addr

[out] 指向 sockaddr 结构体的指针，里面包含了客户端的地址和端口。

addrlen

[out] int 型指针，指向的内容为 addr 指针指向的结构体的长度。

【返回值】

如果函数执行成功，会建立并返回一个新的 SOCKET 来与对方通信，新建的 SOCKET 与原来的 SOCKET（函数的第一个参数 s）有相同的特性，包括端口号。原来的 SOCKET 继续等待其他的连接请求。而新生成的 SOCKET 才是与客户端通信的实际 SOCKET。所以一般将参数中的 SOCKET 称作“监听”SOCKET，它只负责接受连接，不负责通话；而对于函数返回的 SOCKET，把它称作“会话”SOCKET，它负责与客户端通话。

如果失败则返回 INVALID_SOCKET。这时可以调用 WSAGetLastError 函数取得具体的错误代码。

【函数功能】

accept 函数从等待连接的队列中取第一个连接请求，并且创建一个新的 SOCKET 来负责与客户端会话。

【示例代码】

```
SOCKET sServSock;     //服务器监听 socket
sockaddr_in addr;
int nSockErr;
int nNumConns = 0;      //当前请求连接数
SOCKET sConns[5];     //会话 SOCKET 数组
sockaddr ConnAddrs[5];//请求连接的客户端地址
int nAddrLen;

//创建服务器监听 socket
sServSock = socket(AF_INET, SOCK_STREAM, 0);

addr.sin_family = AF_INET;
addr.sin_port = htons(5050);
addr.sin_addr.S_un.S_addr = htonl(INADDR_ANY);

if( bind(sServSock,(LPSOCKADDR)&addr,sizeof(addr)) ==
                    SOCKET_ERROR )
{
    nSockErr = WSAGetLastError();
    //绑定出错处理
}
```

```
//监听客户端请求连接
if( listen(sServSock, 2)  ==  SOCKET_ERROR)
{
    nSockErr = WSAGetLastError();
    //出错处理
}

while( nNumConns < 5){
    //每当收到客户端连接请求，创建新的会话 SOCKET，保存在/  //sConns 数组中
    //客户端地址保存在 ConnAddrs 数组中
    sConns[nNumConns] = accept(sServSock,
                        ConnAddrs[nNumConns], &nAddrLen);
    if(sConns[nNumConns] == INVALID_SOCKET)
    {
        nSockErr = WSAGetLastError();
        //创建会话 SOCKET 出错处理
    }
    else
    {
        //创建会话 SOCKET 成功，启动新的线程与客户端会话
        StartNewHandlerThread(sConns[nNumConns]);
        //当前请求连接数+1
        nNumConns ++;
    }
  }
```

4.6 connect（）

【函数原型】

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

【参数】

s

[in] 一个未连接 SOCKET，一般是由 socket 函数建立的。

name

[in] 同 bind 函数。

namelen

[in] 同 bind 函数。

【返回值】

函数执行成功返回 0，失败则返回 SOCKET_ERROR。这时可以调用 WSAGetLastError 函数取得具体的错误代码。

【函数功能】

向对方主动提出连接请求。

4.7 send（）

【函数原型】

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

【参数】

s

[in] 一个已经连接的 SOCKET。

buf

[in] 指向要传输的数据的缓冲区的指针。

len

[in] buf 的长度。

flags

[in]指定函数调用的方式。一般取 0。

【返回值】

函数执行成功返回发送的字节数（可能小于 len），失败则返回 SOCKET_ERROR。这时可以调用 WSAGetLastError 函数取得具体的错误代码。

【函数功能】

通过已经连接的 SOCKET 发送数据。

4.8 recv（）

【函数原型】

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

【参数】

s

[in] 一个已经连接的 SOCKET。

buf

[out] 指向接收数据的缓冲区的指针。

len

[in] buf 的长度。

flags

[in]指定函数调用的方式。一般取 0。

【返回值】

函数执行成功返回接收到数据的字节数。如果失败则返回 SOCKET_ERROR。这时可以调用 WSAGetLastError 函数取

得具体的错误代码。

【函数功能】

通过已经连接的 SOCKET 接收数据。当读到的数据字节少于规定接受的数目（len）时，就把数据全部接收，并返回实际接收到的字节数；当读到的数据多于规定的值时，在流方式下剩余的数据由下个 recv 读出，在数据报方式下多余的数据被丢弃。

4.9 sendto（）

【函数原型】

int sendto (**SOCKET** s, **char *** buf, **int** len ,**int** flags,

struct sockaddr_in * to, **int** tolen);

【参数】

s

[in] 一个 SOCKET(可能已连接)。

buf

[in] 指向要传输的数据的缓冲区的指针。

len

[in] buf 的长度。

flags

[in] 指定函数调用的方式。一般取 0。

to

[in] 指向目标地址结构体的指针。

tolen

[in] 目标地址结构体的长度。

【返回值】

函数执行成功返回发送的字节数（可能小于 len），失败则返回 SOCKET_ERROR。这时可以调用 WSAGetLastError 函数取得具体的错误代码。

【函数功能】

该函数一般用于通过无连接的 SOCKET 发送数据报文，报文的接受者由 to 参数指定。

4.10 recvfrom（）

【函数原型】

int recvfrom (**SOCKET** s, **char *** buf, **int** len ,**int** flags,

struct sockaddr_in * from, **int** * fromlen);

【参数】

s

[in] 一个已经绑定的 SOCKET。

buf

[out] 指向接收数据的缓冲区的指针。

len

[in] buf 的长度。

flags

[in] 指定函数调用的方式。一般取 0。

from

[out] 指向源地址结构体的指针。

fromlen

[in/out] 源地址结构体的长度。

【返回值】

函数执行成功返回发送的字节数（可能小于 len），失败则返回 SOCKET_ERROR。这时可以调用 WSAGetLastError 函数取得具体的错误代码。

【函数功能】

该函数一般用于通过无连接的 SOCKET 接收数据报文，报文的发送者由 from 参数指定。

4.11 closesocket（）

【函数原型】

int closesocket (**SOCKET** s,);

【参数】

s

[in] 要关闭的 SOCKET。

【返回值】

函数执行成功返回 0，失败则返回 SOCKET_ERROR。这时可以调用 WSAGetLastError 函数取得具体的错误代码。

【函数功能】

关闭指定的 SOCKET。

第五章 Windows Socket 2 的扩展特性

5.1 Winsock 2.0 简介

Winsock 1.1 原先设计的时候把 API 限定在 TCP/IP 的范畴里，它不象 Berkerly 模型那样支持多种协议。而 Winsock 2.0 正规化了一些其它的协议（如 ATM、IPX/SPX 和 DECNet 协议）的 API。

Winsock 2.0 之所以能支持多种协议，是因为 Winsock 2.0 在 Windows Sockets DLL 和底层协议栈之间定义了一个 SPI（Service Provider Interface）接口，这样，通过一个 Windows Sockets DLL 可以同时访问底层不同厂商的协议栈。

Winsock 2.0 不仅允许多种协议栈的并存，而且从理论上讲，它还允许创造一个与网络协议无关的应用程序。Winsock 2.0 可以基于服务的需要透明地选择协议，应用程序可以适用于不同的网络名和网络地址。

Winsock 2.0 还扩展了它的 API 函数集，当然 Winsock 2.0 是向下兼容的，可以把 Winsock 1.1 的代码原封不动地用在 Winsock 2.0 中。

5.2 Winsock 2.0 新特性

下面列出了一些 Winsock 2.0 的重要新特性：

- 多重协议支持：SPI 接口使得新的协议可以被支持。
- 传输协议独立：根据服务提供不同的协议。
- 多重命名空间：根据需要的服务和解析的主机名选择协议。
- 分散和聚集： 从多个缓冲区接受和发送数据。
- 重叠 I/O 和事件对象：增强吞吐量。
- 服务质量（Qos）：协商和跟踪网络带宽。
- 条件接受：可以选择性地决定是否接受连接。
- Socket 共享：多个进程可以共享一个 SOKCKET 句柄。

5.3 Winsock 2.0 新增函数

下面列出了一些 Winsock 2.0 的重要新增函数：

- WSAAccept（）：accept（）函数的扩展版本，支持条件接受和套接字分组。
- WASCloseEvent（）：释放一个时间对象。
- WSAConnect（）：connect（）函数的扩展版本，支持连接数据交换和 Qos 规范。
- WSACreatEvent（）：创建一个事件对象。
- WSADuplicateSocket（）：为一个共享套接字创建一个新的套接字。
- WSAEnumNetworkEvents（）：检查是否有网络事件发生。
- WSAEnumProtocols（）：得到每个可用的协议的信息。
- WSAEventSelect（）：把一个网络事件和一个事件对象连接。
- WSAGetOverlappedResu（）：得到重叠操作的完成状态。

- WSAHtonl（）：htonl（）函数的扩展版本。
- WSAHtons（）：htons（）函数的扩展版本。
- WSAIoctl（）：ioctlsocket（）函数允许重叠操作的扩展版本。
- WSANtohl（）：ntohl（）函数的扩展版本。
- WSANtohs（）：ntohs（）函数的扩展版本。
- WSARecv（）：recv（）的扩展版本，支持分散/聚集/重叠 I/O。
- WSARecvDisconnect（）：终止套接字的接受操作。
- WSARecvFrom（）：recvfrom（）的扩展版本，支持分散/聚集/重叠 I/O。
- WSAResetEvent（）：重新初始化事件对象。
- WSASend（）：send（）的扩展版本，支持分散/聚集/重叠 I/O。
- WSARecvDisconnect（）：终止套接字的接受操作。
- WSASendDisconnect（）：终止套接字的发送操作。
- WSASendTo（）：sendto（）的扩展版本，支持分散/聚集/重叠 I/O。
- WSASetEvent（）：设置事件对象。
- WSASocket（）：socket（）函数的扩展版本。它以一个 PROTOCOL_INFO 结构作为输入参数，并且允许创建重叠套接字，还允许创建套接字组。
- WSAWaitForMultipleEvents（）：阻塞多个事件对象。

关于这些函数的具体细节，请查阅 MSDN。

第六章 MFC Socket 编程

MFC 类中有二个类能支持 Socket 编程：CAsynSocket 类和 CSocket 类。

6.1 CAsynSocket

CAsynSocket 类封装了 Windows Socket 的 API，因此我们可以用面向对象的方法调用 Sokcet。这个类中，必须自己处理阻塞和 Unicode 与其他字节集的字节顺序转换，因此 CAsynSocket 类对 Windows Socket 的 API 的封装是在教低层次上进行。

从该类的名字可以就看出，CAsynSocket 类提供了异步通信编程模式：在默认状态下由该类创建的套接字是非阻塞的套接字，在这个非阻塞套接字上进行的包括接受数据和建立连接等在内所有操作也都是非阻塞的。对于异步通信编程（非阻塞式 Socket），该类将网络事件加入到 Windows 的消息循环机制当中，使得用户能像响应普通的键盘、鼠标、窗口重画等事件一样来响应网络事件，如收到连接请求、有数据到来等。

由于在 CAsynSocket 类是由 OnAccept()、OnReceive()等虚函数来响应各种网络事件的，因此需要在编程时对这些虚函数进行重载，以完成具体的功能。这就需要从 CAsynSocket 类派生一个继承类，并在该派生类中进行程序设计。

6.1.1 CAsynSocket 类的组成

CAsynSocket 类的属性函数如表 5-1 所示。

属性函数名	功能
Deatch	解除一个 CAsynSocket 对象和连接到该对象的 Socket 句柄的连接
FromHandle	给定一个 Socket 句柄，返回这个句柄所指向的 CAsynSocket 对象
GetLastError	返回上次操作的错误状态
GetPeerName	得到和该 CAsynSocket 对象相连接的对方 Socket 的地址
GetSockName	得到该 CAsynSocket 对象中 Socket 自身的地址
GetSockOpt	得到当前 Socket 的状态
SetSockOpt	设置当前 Socket 的状态

表 5-1 CAsynSocket 类的属性函数表

CAsynSocket 类的操作函数如表 5-2 所示。

操作函数名	功能
Accept	接收连接请求
AsynSelect	请求网络事件通知
Bind	把本地地址关联到 Socket 上
Close	关闭当前 Socket
Connect	与对方 Socket 连接
IOCtl	控制 Socket 模式
Listen	监听连接请求
Receive	从 Socket 接收数据

ReceiveFrom	接收一个数据报并存储源地址
Send	发送数据到 Socket
SendTo	发送数据到一个具体的地址

表 5-2 CAsynSocket 类的操作函数表

CAsynSocket 类的可重载的消息响应函数如表 5-3 所示。

操作函数名	功能
OnAccept	通知监听 Socket，一个请求到来
OnClose	通知 Socket，连接已关闭
OnConnect	通知正在连接的 Socket，连接尝试已完成，连接可能成功或失败
OnOutofBandData	通知 Socket，有带外数据到来
OnReceive	通知 Socket 有数据到来，可以调用 Receive 去接收数据
OnSend	通知 Socket 可以调用 Send 发送数据

表 5-3 CAsynSocket 类可重载的消息响应函数表

最后，CAsynSocket 类有个数据成员 m_hSocket，表示与 CAsynSocket 对象关联的 Socket 句柄。

这里列出的 CAsynSocket 类的成员函数没有给出详细的参数类型和返回值，是因为 CAsynSocket 类封装了 Windows Socket 的 API，这些成员函数的功能基本上和 Windows Socket 的 API 中对应的函数一致。成员函数的详细信息请参考 Microsoft MSDN。

6.1.2 CAsynSocket 编程模型

创建和使用 CAsynSocket 的步骤如下：

（1）调用 AfxSocketInit（）初始化。

```
if(AfxSocketInit（）==FALSE)
    AfxMessageBox(“Error!”);
```

（2）从 CAsynSocket 派生一个类，如 CSessionSocket。

（3）构造一个 CSessionSocket 对象，并且使用该对象创造一个 Socket 句柄。

```
CSessionSocketsock;          //在堆栈中创建对象
sock.Create();       /使用缺省参数，创建一个流式 Socket 或者
CAsynSocket *pSocket; //在堆区中创建对象
int nPort = 27;
pSocket->Create(nPort, SOCK_DGRAM);//创建一个数据报 Socket
```

（4）如果是客户端 Socket，则直接调用 CAsynSocket::Connect 函数与服务器相连。如果是服务器端 Socket，则调用 CAsynSocket::Listen 函数去监听连接。一旦有连接请求到来，就可以调用 CAsynSocket::Accept 函数去构造一个连接 Socket。

（5）等待网络事件的发生。当事件到来时，在事件处理函数里进行处理，例如如用 CAsynSocket 封装的函数进行数据传输。

（6）调用 Close 函数结束连接。

具体例子请参见第八章代码。

6.2 CSocket

CSocket 继承于 CAsynSocket，是 Windows Socket API 的高层抽象。CSocket 通常和 CSocketFile 和 CArchive 类混合使用，这二个类负责数据的发送和接收。

除了继承 CAsynSocket 的函数成员外，CSocket 最重要的特性是通过 CArchive、CSocketFile 来发送接收数据，从而将编程人员和 Socket 的繁琐细节屏蔽开来。你需要做的只是创建 Socket、CArchive 对象、CSocketFile 对象并将它们关联起来，然后你只是从 CArchive 对象中读取数据或向 CArchive 对象写数据。三个类之间的关系如图 5-1 所示。

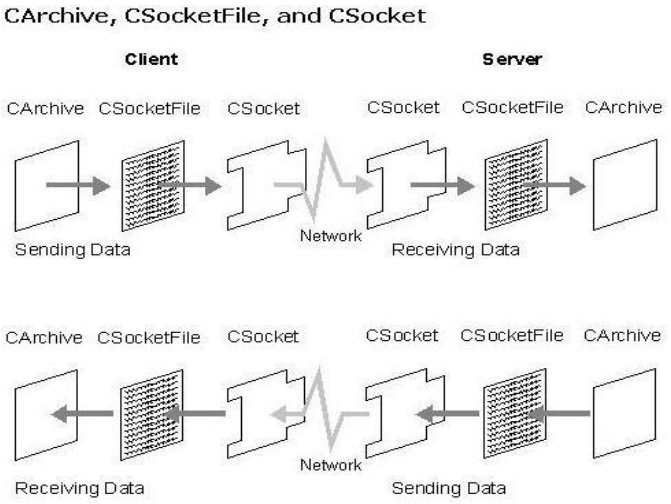


图 5-1 CArchive、CSocketFile、CSocket 之间的关系图

6.2.1 CArchive 类

CArchive 类允许你将一个内存中的对象序列化到永久存储的介质中去。当要创建一个 CArchive 对象时，必须先创建一个 CSocketFile 对象并将其关联到 CArchive 对象（事先创建好的 CSocketFile 对象的指针作为 CArchive 的构造函数的第一个参数）。

CArchive 类主要的输入输出函数如表 5-4 所示。

函数名	功能
operator >>	输入带类型的数据
operator >>	输出带类型的数据
Read	输入原始字节
Write	输出原始字节
WriteString	一次输入一行字符
ReadString	一次输出一行字符

表 5-4 CArchive 类主要的输入输出函数表

CArchive 类主要的状态函数如表 5-5 所示。

函数名	功能
GetFile	得到 CScoketFile 对象的指针
IsLoading	Archive 对象是否处在输入状态
IsStoring	Archive 对象是否处在输出状态
IsBufferempty	缓冲区是否空

表 5-5 CArchive 类主要的状态函数表

6.2.2 CSocketFile 类

CSocketFile 类从 CFile 类继承而来，因此它的成员函数请参考 MSDN 中 CFile 类的文档。这里仅介绍如何将一个 Socket 和一个 CSocketFile 对象关联起来，这通过将一个 CSocket 对象的指针传递给 CSocketFile 的构造函数来完成。CSocketFile 的构造函数的详细说明请参考 MSDN 文档。

6.2.3 CSocket 的编程模型

创建和使用 CSocket 的前面步骤和 CAsynSocket 完全一样（因为 CSocket 就是从 CAsynSocket 继承而来）。区别就在当创建好 CSocket 对象后，你要将 CSocket 对象和 CSocketFile 对象关联起来，然后再将 CSocketFile 对象和 CArchive 对象关联起来。

这里必须要特别注意的是，如果你创建的是一个数据报 CSocket 对象，则 CSocketFile 对象不能再和 CArchive 对象关联了。换句话说，只有和流式 CSocket 对象关联的 CSocketFile 对象才能进一步和 CArchive 对象关联。

一旦关联成功，我们只需要从 CSocketFile 对象或 CArchive 对象输入或输出数据而不用去考虑 Socket 接收或发送数据的细节。这个时候，我们从网络输入数据或从向网络上发送数据和我们对一个普通外设（如文件）进行输入输出操作所使用的函数完全是一样的。

下面一段示例代码说明了如何把 CSocket 类和 CArchive、CSocketFile 类关联起来：

（1）将通过 socket 发送和接受的消息封装在一个可序列化的类中。

```
//CMessg 类用来封装通过 socket 发送和接受的消息
//通过从 CObject 类继承来的序列化机制来实现
// CMessg 的声明：CMessg.h
class CMessg : public CObject
{
protected:
    DECLARE_DYNCREATE(CMessg)   //序列化机制所需要的宏
public:
    CMessg();

// Attributes
public:
    CString m_strText; //发送或接受的消息

// Operations
public:
    void Init();     //初始化函数

// Implementation
public:
    virtual ~CMessg();
```

```
    //重载该函数以实现序列化 IO
    virtual void Serialize(CArchive& ar);
#ifdef _DEBUG
    virtual void AssertValid() const;    //调试用函数
    virtual void Dump(CDumpContext& dc) const; //调试用函数
#endif
};


//CMessg 实现：CMessg.cpp
#include "CMessg.h"
IMPLEMENT_DYNCREATE(CMessg, CObject)//必须的宏

CMessg::CMessg() { Init();}
CMessg::~CMessg(){}
void CMessg::Init(){      m_strText = _T("");}
// CMsg serialization
void CMessg::Serialize(CArchive& ar)
{
    if (ar.IsStoring()){ ar << m_strText; }
    else { ar >> m_strText;}
}

//  调试用
#ifdef _DEBUG
void CMessg::AssertValid() const
{
    CObject::AssertValid();
}

void CMessg::Dump(CDumpContext& dc) const
{
    CObject::Dump(dc);
}
#endif //_DEBUG
```

（2）将 Socket 与 CArchive 关联

```
//创建一个 Socket
CSessionSocket *pSocket = new CSessionSocket();
//将 Socket 与 CSocketFile 关联
```

```
CSokcetFile *pSocketFile = new CSocketFile(pSocket);
//将 CSocketFile 与 CArchive 关联
CArchive *pin = new CArchive(pSocketFile,CArchive::load);//输入用
CArchive *pout = new CArchive(pSocketFile,CArchive::store);//输出用
```

（3）发送数据

```
CMessg msg;
msg.m_strText = “Hello,World”;//要发送的字符串
msg.Serialize(*pout);
pout->Flush();
```

（4）接受数据

```
CMessg temp;
temp.Serialize(*pin);
cout << temp.m_strText; //打印出所接受的字符串
```

6.3 MFC 中的多线程

在 MFC 中线程分为二种：工作者线程（Worker thread）和用户界面线程（UI thread）。二者的区别在于：工作者线程是没有消息队列和消息循环机制的，而用户界面线程则有自己的消息队列和消息循环，因此可以在用户界面线程里和主线程一样处理各种消息如键盘、鼠标消息等。

通常的做法是在主线程里处理用户界面产生的各种消息，在工作者线程里处理各种耗时的、或阻塞式的 IO 动作，比如从 Socket 中接受和发送数据（如阻塞式的 send（）调用或 recv（）调用）等。

这里要注意的是：MSDN 建议最好在辅助工作者线程里使用阻塞式 Socket，因为如果在工作者线程里使用异步（非阻塞式）Socket，则意味着你必须要在工作者线程里等待事件通知消息，而工作者线程是没有任何消息机制的。因此，在这里建议如果要使用多线程，则在辅助工作者线程不要使用 CAsynSocket 类，而使用阻塞式 Socket。不幸的是，微软没有提供阻塞式 Socket 类，因此在这种情况下最好自己将 Winsock API 函数封装成阻塞式 Socket 类。在《Visual C++ 6.0 技术内幕》里有完整的自己封装的阻塞式 Socket 类 CBlockingSocket 的代码示例以及如何利用该类实现多线程的 Web 服务器的代码示例。

MFC 提供了全局函数 AfxBeginThread 来创建工作者线程，该函数原型如下：

```
CWinThread * AfxBeginThread(
    AFX_THREADPROC pfnThreadProc
    LPVOID pParam,
    int nPriority = THREAD_PRIORITY_NORMAL,
    UINT nStackSize=0,
    DWORD dwCreateFlags = 0,
    LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL);
```

【参数】

pfnThreadProc：

新线程启动后要执行的线程函数指针。线程函数的原型必须为：UINT ThreadFunc(LPVOID n)。

pParam：

传递给新线程函数的参数的指针，类型为 void *。

nPriority：

新线程的优先级，缺省为 THREAD_PRIORITY_NORMAL。

nStackSize：

新线程堆栈的大小，缺省（参数值 0）为默认大小。

dwCreateFlags：

此值必须为 0（新线程立刻执行）或 CREATE_SUSPENDED（新线程立刻被挂起）。

lpSecurityAttrs：

新线程的安全属性。

【返回值】

如果失败，返回 NULL，否则返回一个 CWinThread 对象指针。

下面的代码给出了线程创建的示例。

```
CWinApp theApp;
using namespace std;

//线程函数的原型必须是 UINT ThreadFunc(LPVOID n);
UINT ThreadFunc(LPVOID n)
{
    for(int i = 0; i < 10; i++)
        printf("%d",n);
    return 0;
}

int _tmain(int argc, TCHAR* argv[], TCHAR* envp[])
{
    int nRetCode = 0;
    CWinThread *pThread[5];

    // initialize MFC and print and error on failure
    if (!AfxWinInit(::GetModuleHandle(NULL),
                    NULL,
                    ::GetCommandLine(), 0))
    {
        cerr << _T("Fatal Error: MFC initialization failed") << endl;
        nRetCode = 1;
    }
```

```
    else
    {
        for(int i = 0; i < 5; i++)
        {
            pThread[i] = AfxBeginThread(
                        ThreadFunc,
                        (LPVOID)i,
                        THREAD_PRIORITY_NORMAL,
                        0,
                        CREATE_SUSPENDED); //线程先挂起
            ASSERT(pThread[i]);

            //防止 pThread[i]被自动删除
            pThread[i]->m_bAutoDelete = FALSE;
            pThread[i]->ResumeThread();//这时才启动线程
            printf("\nThread launched %d\n",i);
        }
    }

    //这里是为了等待子线程的结束，
    //以免发生子线程还没结束而主线程已结束的现象
    for(int i = 0; i < 5; i++)
    {
        //因为 pThread[i]不会被自动删除，
        //因此这时调用 WaitForSingleObject 函数是安全的
        WaitForSingleObject(pThread[i]->m_hThread,INFINITE);
        delete pThread[i];
    }
    return nRetCode;
}
```

在上面这段代码中，有几点必须注意：

1．在创建线程时 dwCreateFlags 参数设为 CREATE_SUSPENDED，以便我们有机会在线程启动前能对线程的属性做适当的设置。

2．CWinThread 对象的 m_bAutoDelete 设为 FALSE，防止线程结束后 CWinThread 对象被主线程自动删除。

3．为了防止在子线程还未结束而主线程已经结束（线程的执行顺序是无法确定的，当主线程先于子线程结束时，主线程会强行结束所有的子线程），我们在主线程中调用 WaitForSingleObject 等待所有的子线程结束。

关于 MFC 的多线程就介绍这么多，因为 MFC 中的多线程实在是一个庞大的主题，关于线程间的同步、如何安全访问共享区等主题，请参阅 MSDN 或《Win32 多线程程序设计》一书。

第七章 Windows Socket 1.1 编程示例

本章给出了流式 Socket 和数据报 Socket 编程简单示例，供各位同学参考。示例代码说明如下：

1、流式 Socket 示例代码是一个用 Win32 控制台程序实现的一个简单聊天室的例子。客户端在一个控制台窗口输入的聊天信息发送到服务器端，服务器将该条消息转发到所有当前和服务器保持连接的客户端上。

2、数据报 Socket 示例代码将一个长度小于 512 字节的字符串传输给服务器。

7.1 流式 Socket 服务器端代码

```
// socksrv.cpp : Defines the entry point for the console application.
#include "stdafx.h"
#include <stdio.h>
#include <winsock2.h>
#include <list>
#include <algorithm>
#include <string.h>

#define MAXCONN 5
#define BUFLEN 255

using namespace std;

typedef list<SOCKET> ListCONN;
typedef list<SOCKET> ListConErr;

void main(int argc, char* argv[])
{
   WSADATA wsaData;
   int nRC;
   sockaddr_in srvAddr,clientAddr;
   SOCKET srvSock;
   int nAddrLen = sizeof(sockaddr);
   char sendBuf[BUFLEN],recvBuf[BUFLEN];
   ListCONN conList;          //保存所有有效的会话 SOCKET
   ListCONN::iterator itor;
   ListConErr conErrList;     //保存所有失效的会话 SOCKET
   ListConErr::iterator itor1;
   FD_SET rfds,wfds;
   u_long uNonBlock;

   //初始化  winsock
   nRC = WSAStartup(0x0101,&wsaData);
   if(nRC)
   {
      printf("Server initialize winsock error!\n");
```

```
    return;
}
if(wsaData.wVersion != 0x0101)
{
    printf("Server's winsock version error!\n");
    WSACleanup();
    return;
}
printf("Server's winsock initialized !\n");

//创建  TCP socket
srvSock = socket(AF_INET,SOCK_STREAM,0);
if(srvSock == INVALID_SOCKET)
{
    printf("Server create socket error!\n");
    WSACleanup();
    return;
}
printf("Server TCP socket create OK!\n");

//绑定  socket to Server's IP and port 5050
srvAddr.sin_family = AF_INET;
srvAddr.sin_port = htons(5050);
srvAddr.sin_addr.S_un.S_addr = INADDR_ANY;
nRC=bind(srvSock,(LPSOCKADDR)&srvAddr,sizeof(srvAddr));
 if(nRC == SOCKET_ERROR)
{
    printf("Server socket bind error!\n");
    closesocket(srvSock);
    WSACleanup();
    return;
}
printf("Server socket bind OK!\n");

//开始监听过程，等待客户的连接
nRC = listen(srvSock,MAXCONN);
if(nRC == SOCKET_ERROR)
{
    printf("Server socket listen error!\n");
    closesocket(srvSock);
    WSACleanup();
    return;
}

//将 srvSock 设为非阻塞模式以监听客户连接请求
uNonBlock = 1;
ioctlsocket(srvSock,FIONBIO,&uNonBlock);
```

```
while(1)
{
   //从 conList 中删除已经产生错误的会话 SOCKET
   for(itor1 = conErrList.begin();itor1 != conErrList.end();itor1++)
   {
    itor = find(conList.begin(),conList.end(),*itor1);
    if(itor != conList.end()) conList.erase(itor);
   }

   //清空 read,write 套接字集合
   FD_ZERO(&rfds);
   FD_ZERO(&wfds);

   //设置等待客户连接请求
   FD_SET(srvSock,&rfds);

   for(itor = conList.begin();itor != conList.end();itor++)
   {
    //把所有会话 SOCKET 设为非阻塞模式
    uNonBlock = 1;
    ioctlsocket(*itor,FIONBIO,&uNonBlock);
    //设置等待会话 SOKCET 可接受数据或可发送数据
    FD_SET(*itor,&rfds);
    FD_SET(*itor,&wfds);
   }
   //开始等待
   int nTotal = select(0, &rfds, &wfds, NULL, NULL);

   //如果 srvSock 收到连接请求，接受客户连接请求
   if(FD_ISSET(srvSock,&rfds))
   {
    nTotal --;
    //产生会话 SOCKET
    SOCKET connSock = accept(srvSock,
                             (LPSOCKADDR)&clientAddr,
                             &nAddrLen);
    if(connSock == INVALID_SOCKET)
    {
        printf("Server accept connection request error!\n");
        closesocket(srvSock);
        WSACleanup();
        return;
    }
    sprintf(sendBuf,"来自%s 的游客进入聊天室!\n",
                        inet_ntoa(clientAddr.sin_addr));
    printf("%s",sendBuf);
```

```
 //将产生的会话 SOCKET 保存在 conList 中
 conList.insert(conList.end(),connSock);
}
if(nTotal > 0)
{
 //检查所有有效的会话 SOCKET 是否有数据到来
      //或是否可以发送数据
 for(itor = conList.begin();itor != conList.end();itor++)
 {
       //如果会话 SOCKET 可以发送数据，
            //则向客户发送消息
       if(FD_ISSET(*itor,&wfds))
       {
            //如果发送缓冲区有内容，则发送
            if(strlen(sendBuf) > 0)
            {
                 nRC = send(*itor,sendBuf,strlen(sendBuf),0);
                 if(nRC == SOCKET_ERROR)
                 {
                      //发送数据错误，
                           //记录下产生错误的会话 SOCKET
                      conErrList.insert(conErrList.end(),*itor);
                 }
                 else//发送数据成功，清空发送缓冲区
                      memset(sendBuf,'\0',BUFLEN);
            }
       }

       //如果会话 SOCKET 有数据到来，则接受客户的数据
       if(FD_ISSET(*itor,&rfds))
       {
            nRC = recv(*itor,recvBuf,BUFLEN,0);
            if(nRC == SOCKET_ERROR)
            {
                 //接受数据错误，
                      //记录下产生错误的会话 SOCKET
                      conErrList.insert(conErrList.end(),*itor);
            }
            else
            {
                 //接收数据成功，保存在发送缓冲区中，
                      //以发送到所有客户去
                 recvBuf[nRC] = '\0';
                 sprintf(sendBuf,"\n 游客说:%s\n",recvBuf);
                 printf("%s",sendBuf);
            }
```

```
            }
           }
      }
   }
   closesocket(srvSock);
   WSACleanup();
}
```

7.2 流式 Socket 客户端代码

```
// sockclient.cpp : Defines the entry point for the console application.
//
#include "stdafx.h"
#include <stdio.h>
#include <winsock2.h>
#include <string.h>
#include <windows.h>
#include <process.h>
#include <winbase.h>
#define BUFLEN 255

//全局的临界区保护变量，以保护主线程和子线程都要访问的 sendBuf
CRITICAL_SECTION gCriticalSection;

//子线程运行的函数，获取客户从键盘输入的信息
unsigned __stdcall GetInputs(void *arg);

void main(int argc, char* argv[])
{
      WSADATA wsaData;
      int nRC;
      sockaddr_in srvAddr,clientAddr;
      SOCKET clientSock;
      char sendBuf[BUFLEN],recvBuf[BUFLEN];
      FD_SET rfds,wfds;
      u_long uNonBlock;
      HANDLE hThread;
      unsigned dwThreadID;

      if(argc != 2)
      {
            printf("Usage:%s ClientIPAddress name\n",argv[0]);
            return;
      }

      InitializeCriticalSection(&gCriticalSection);
      //初始化  winsock
```

```
nRC = WSAStartup(0x0101,&wsaData);
if(nRC)
{
	printf("Client initialize winsock error!\n");
	return;
}
if(wsaData.wVersion != 0x0101)
{
	printf("Client's winsock version error!\n");
	WSACleanup();
	return;
}
printf("Client's winsock initialized !\n");

//创建  client socket
clientSock = socket(AF_INET,SOCK_STREAM,0);
if(clientSock == INVALID_SOCKET)
{
	printf("Client create socket error!\n");
	WSACleanup();
	return;
}
printf("Client    socket create OK!\n");

clientAddr.sin_family = AF_INET;
clientAddr.sin_port = htons(0);
clientAddr.sin_addr.S_un.S_addr = inet_addr(argv[1]);
nRC = bind(clientSock,
	(LPSOCKADDR)&clientAddr,sizeof(clientAddr));
if(nRC == SOCKET_ERROR)
{
	printf("Client socket bind error!\n");
	closesocket(clientSock);
	WSACleanup();
	return;
}
printf("Client socket bind OK!\n");


//准备服务器的信息，这里需要指定服务器的地址
srvAddr.sin_family = AF_INET;
srvAddr.sin_port = htons(5050);
srvAddr.sin_addr.S_un.S_addr = inet_addr("192.168.0.3");

//连接服务器
nRC = connect(clientSock,
	(LPSOCKADDR)&srvAddr,sizeof(srvAddr));
if(nRC == SOCKET_ERROR)
```

```
{
    printf("连接服务器失败!\n");
    closesocket(clientSock);
    WSACleanup();
    return;
}

//启动一个子线程，获取客户从键盘输入的信息
hThread = (HANDLE)_beginthreadex(NULL,
                                 0,
                                 GetInputs,
                                 sendBuf,
                                 0,
                                 &dwThreadID);


//向服务器发送数据和从服务器接受数据
while(1)
{
    //清空发送和接收缓冲区
    memset(sendBuf,'\0',BUFLEN);
    memset(recvBuf,'\0',BUFLEN);

    //将 SOCKET 设为非阻塞模式，
    //并且等待有数据到来或者可以发送数据
    FD_ZERO(&rfds);
    FD_ZERO(&wfds);
    FD_SET(clientSock,&rfds);
    FD_SET(clientSock,&wfds);
    uNonBlock = 1;
    ioctlsocket(clientSock,FIONBIO,&uNonBlock);
    select(0,&rfds,&wfds,NULL,NULL);

    //如果有数据到来
    if(FD_ISSET(clientSock,&rfds))
    {
        //接受服务器发来的数据并且显示
        nRC = recv(clientSock,recvBuf,BUFLEN,0);
        if(nRC == SOCKET_ERROR)
        {
            printf("接收数据失败!\n");
            DeleteCriticalSection(&gCriticalSection);
            closesocket(clientSock);
            WSACleanup();
            return;
        }
        else if(nRC > 0)
```

```
            {
                recvBuf[nRC] = '\0';
                printf("\n%s\n",recvBuf);
            }
        }
        //如果可以发送数据
        if(FD_ISSET(clientSock,&wfds))
        {
            //如果用户在键盘输入了信息，则发送
            if(strlen(sendBuf) > 0)
            {
                nRC = send(clientSock,sendBuf,strlen(sendBuf),0);
                if(nRC == SOCKET_ERROR)
                {
                    printf("发送数据失败!\n");
                    DeleteCriticalSection(&gCriticalSection);
                    closesocket(clientSock);
                    WSACleanup();
                    return;
                }
                else
                {
                    EnterCriticalSection(&gCriticalSection);
                    sendBuf[0] = '\0';
                    LeaveCriticalSection(&gCriticalSection);
                }
            }
        }
        if(strcmp(sendBuf,"exit") == 0) break;
    }
    DeleteCriticalSection(&gCriticalSection);
    closesocket(clientSock);
    WSACleanup();
}
//子线程运行的函数，获取客户从键盘输入的信息
unsigned __stdcall GetInputs(void *arg)
{
    char *inputs = (char *)arg;
    while(1)
    {
        printf("\n 我要发言:");
        EnterCriticalSection(&gCriticalSection);
        gets(inputs);
        LeaveCriticalSection(&gCriticalSection);
        if(strcmp(inputs,"exit") == 0)
            return EXIT_SUCCESS;
    }
}
```

7.3 数据报 Socket 服务器端代码

```
#define SERVER_PORT 8000
#define BUFFER_SIZE 1024
#define FILE_NAME_MAX_SIZE 512

int main()
{
 /* 创建 UDP 套接口 */
 struct sockaddr_in server_addr;
 bzero(&server_addr, sizeof(server_addr));
 server_addr.sin_family = AF_INET;
 server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
 server_addr.sin_port = htons(SERVER_PORT);

 /* 创建 socket */
 int server_socket_fd = socket(AF_INET, SOCK_DGRAM, 0);
 if(server_socket_fd == -1)
 {
  perror("Create Socket Failed:");
  exit(1);
 }

 /* 绑定套接口 */
 if(-1 == (bind(server_socket_fd,(struct sockaddr*)&server_addr,sizeof(server_addr))))
 {
  perror("Server Bind Failed:");
  exit(1);
 }

 /* 数据传输 */
 while(1)
 {
  /* 定义一个地址，用于捕获客户端地址 */
  struct sockaddr_in client_addr;
  socklen_t client_addr_length = sizeof(client_addr);

  /* 接收数据 */
  char buffer[BUFFER_SIZE];
  bzero(buffer, BUFFER_SIZE);
  if(recvfrom(server_socket_fd, buffer, BUFFER_SIZE,0,(struct sockaddr*)&client_addr, &client_addr_length) == -1)
  {
   perror("Receive Data Failed:");
   exit(1);
  }
```

```
 /* 从 buffer 中拷贝出 file_name */
 char file_name[FILE_NAME_MAX_SIZE+1];
 bzero(file_name,FILE_NAME_MAX_SIZE+1);
 strncpy(file_name, buffer, strlen(buffer)>FILE_NAME_MAX_SIZE?FILE_NAME_MAX_SIZE:strlen(buffer));
 printf("%s\n", file_name);
 }
 close(server_socket_fd);
 return 0;
}
```

7.4 数据报 Socket 客户端代码

```
#define SERVER_PORT 8000
#define BUFFER_SIZE 1024
#define FILE_NAME_MAX_SIZE 512

int main()
{
 /* 服务端地址 */
 struct sockaddr_in server_addr;
 bzero(&server_addr, sizeof(server_addr));
 server_addr.sin_family = AF_INET;
 server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
 server_addr.sin_port = htons(SERVER_PORT);

 /* 创建 socket */
 int client_socket_fd = socket(AF_INET, SOCK_DGRAM, 0);
 if(client_socket_fd < 0)
 {
 perror("Create Socket Failed:");
 exit(1);
 }

 /* 输入文件名到缓冲区 */
 char file_name[FILE_NAME_MAX_SIZE+1];
 bzero(file_name, FILE_NAME_MAX_SIZE+1);
 printf("Please Input File Name On Server:\t");
 scanf("%s", file_name);

 char buffer[BUFFER_SIZE];
 bzero(buffer, BUFFER_SIZE);
 strncpy(buffer, file_name, strlen(file_name)>BUFFER_SIZE?BUFFER_SIZE:strlen(file_name));

 /* 发送文件名 */
```

```
if(sendto(client_socket_fd, buffer, BUFFER_SIZE,0,(struct sockaddr*)&server_addr,sizeof(server_addr)) < 0)
{
 perror("Send File Name Failed:");
 exit(1);
}

close(client_socket_fd);
return 0;
}
```

7.5 工程配置

服务器端和客户端程序都为 Win32 Console Application。

对于服务器和客户端工程，都必须打开工程设置（Project->Settings…），然后选中 Link 选项卡，在 Object/library modules 栏目中添加 ws2_32.lib。

对于客户端工程，还必须打开工程设置（Project->Settings…），然后选中 C/C++选项卡，Category 选择 Code generation ,Use run-time library 选择 Debug Multithreaded。

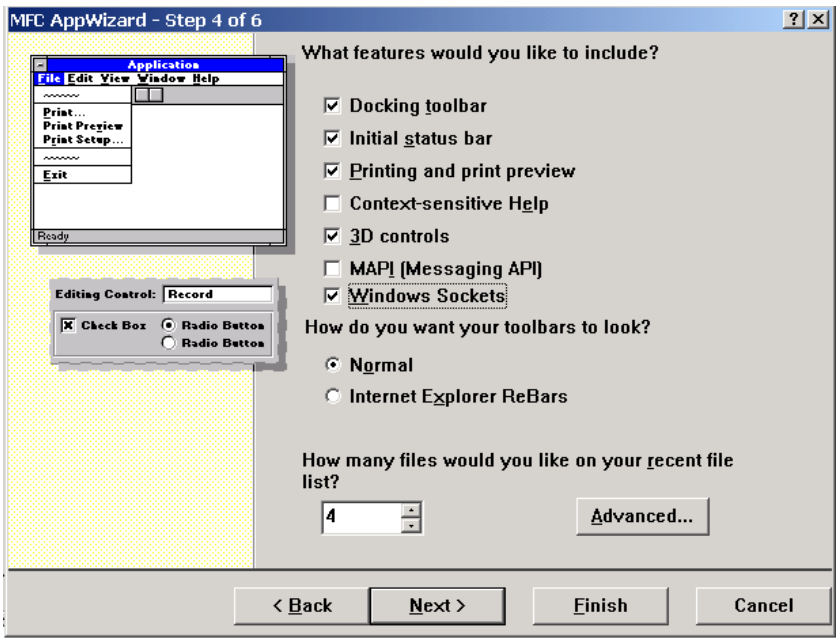
第八章 MFC Socket 编程示例

这里介绍一个用 MFC CAsynSocket 类实现的带用户界面的聊天室的例子。该程序的客户端和服务器端都采用单文档（ＳＤＩ）界面实现。

8.1 服务器端实现

服务器端程序创建过程如下：

1) 在 Visual C++6.0 下创建一个新工程，程序类型选择 MFC AppWizard(exe)，工程名为 Server，在 AppWizard 下一个窗口选择程序类型为单文档（Single Document），在第 4 步设置界面中一定要选择 Windows Sokcets。



在最后一步设置界面中把 CServerView 的基类改为 CFormView.

2) 在资源编辑器中，在 CServerView 中放入控件 ListBox。

3) 这个时候应用程序框架已经为你生成好框架代码，如果你这时编译并运行该工程，就会打开一个没有任何功能的单文档界面。

4) 打开 CServerView.h 文档，为该类加入下二个 public 类型属性变量成员：

```
CListenSocket * m_pListenSocket;//监听 socket 指针
CPtrList *m_pSessionList; //会话 Socket 链表指针
```

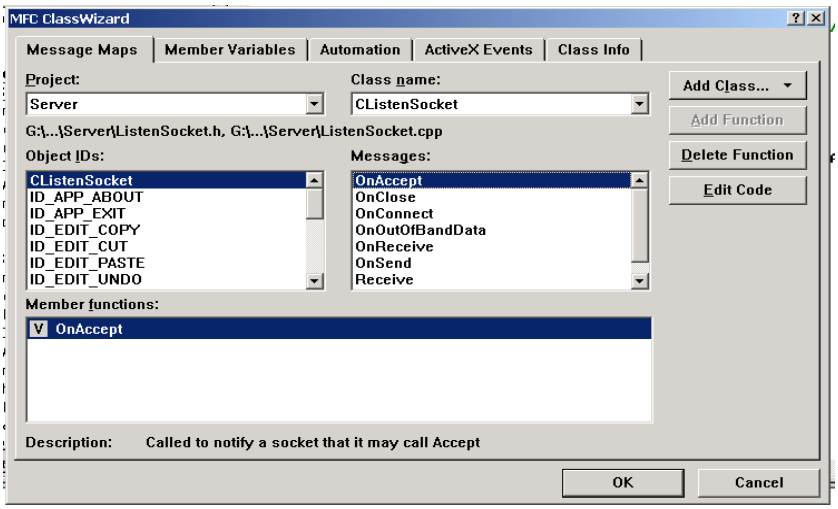
5) 打开 CServerView.cpp 文档，在 CServerView::OnInitialUpdate()函数中最后（一定不要改动任何框架自动生成的代码）加入下列代码：

```
// 创建套接字
m_pListenSocket = new CListenSocket();
if (m_pListenSocket->Create(
      PORT, SOCK_STREAM, FD_ACCEPT) == FALSE)
{
     AfxMessageBox("创建套接字失败!");
      m_pListenSocket->Close();
      return;
```

```
}

// 侦听成功，等待连接请求
if (m_pListenSocket->Listen(5) == FALSE)
{
    if (m_pListenSocket->GetLastError() == WSAEWOULDBLOCK)
    {
        AfxMessageBox("网络侦听失败!");
        m_pListenSocket->Close();
        return;
    }
}
AfxMessageBox("服务器已启动!");
```

6) 打开 Class Wizard，点击“Add Class”按钮增加新类 CListenSocket，选择其基类为 CAsynSocket。该类的作用是监听客户请求。在第五章我们提到，CAsynSocket 类的特点是异步 Socket，MFC 为该类提供了和键盘、鼠标完全一致的事件响应机制，因此当有客户请求到来时，该类会收到 OnAccept 消息，从而触发 OnAccept（）事件响应函数。因此我们的工作就变得很简单：在 OnAccept（）事件响应函数中加入我们的代码。打开 ClassWizard，选择 CListenSocket 类，“Messages”列表框会显示该类能响应的所有消息，双击“OnAccept”消息，则 ClassWizard 会自动为我们在该类添加事件响应函数 OnAccept(int nErrorCode)。该过程如下图所示：



该事件响应函数代码如下：

```
//当收到客户请求时的事件响应函数
void CListenSocket::OnAccept(int nErrorCode)
{
    //得到视图窗口指针
    CServerView* pView = (CServerView*)
            ((CMainFrame*)AfxGetApp()->m_pMainWnd)
            ->GetActiveView();

    CSessionSocket* pNewSocket = new CSessionSocket();

    // 接受连接,得到会话 socket,
    if (Accept(*pNewSocket))
    {
```

```
        //为新得到的会话 socket 设置异步选择事件
        pNewSocket->AsyncSelect(FD_READ | FD_CLOSE);

        //将新得到的会话 socket 加到会话 socket 链表的末尾
        pView->m_pSessionList->AddTail(pNewSocket);

        // 在视图中显示信息
        pView->m_listData.AddString("欢迎新朋友进入聊天室!");

        CString temp = "欢迎新朋友进入聊天室!";
        //遍历会话 socket 链表,向其他会话 socket 转发数据
        for(POSITION pos =
              pView->m_pSessionList->GetHeadPosition();
              pos!=NULL;)
        {
              CSessionSocket *socket =(CSessionSocket *)
                    pView->m_pSessionList->GetNext(pos);
              socket->Send(temp,temp.GetLength());
        }
    }
    else
        delete pNewSocket;
    CAsyncSocket::OnAccept(nErrorCode);
}
```

7) 同上一步一样，通过 Class Wizard 加入 CSessionSocket 类。该类的基类同样为 CAsynSocket，其作用为和客户端会话。同样利用 Class Wizard 为该类添加对 OnRceive 和 OnClose 的事件响应函数。这二个事件响应函数的代码如下：

```
//接收到数据时的事件响应函数
void CSessionSocket::OnReceive(int nErrorCode)
{
    char buf[1024];
    memset(buf, 0, sizeof(buf)); //创建接受缓冲区

    // 接收数据,nRcvBytes 为收到的字节数
    int nRcvBytes = Receive(buf, 1024);

    //得到视图指针,在视图中显示信息
    CServerView* pView = (CServerView*)
              ((CMainFrame*)AfxGetApp()->m_pMainWnd)->
              GetActiveView();
    pView->m_listData.AddString(CString(buf));

    //遍历会话 socket 链表,向其他会话 socket 转发数据
    POSITION pos;
    for(pos = pView->m_pSessionList->GetHeadPosition();
          pos!=NULL;)
    {
```

```
            CSessionSocket *socket = (CSessionSocket *)
                        pView->m_pSessionList->GetNext(pos);
            socket->Send(buf,nRcvBytes);
        }
        CAsyncSocket::OnReceive(nErrorCode);
}


//当客户关闭连接时的事件响应函数
void CSessionSocket::OnClose(int nErrorCode)
{
        // 关闭套接字
        Close();

        //在 View 里显示信息
        CServerView* pView =    (CServerView*)
                    ((CMainFrame*)AfxGetApp()->m_pMainWnd)->
                    GetActiveView();
        pView->m_listData.AddString(
            "游客离开聊天室,让我们深情凝望着他(她)的背影...");

        //遍历会话 socket 链表,删除中断连接的会话 socket
        POSITION pos,lastpos;
        for(pos = pView->m_pSessionList->GetHeadPosition();
            pos!=NULL;)
        {
            lastpos=pos;
            CSessionSocket *socket =(CSessionSocket *)
                        pView->m_pSessionList->GetNext(pos);
            if(socket->m_hSocket==this->m_hSocket)
                pView->m_pSessionList->RemoveAt(lastpos);
        }
        CString temp =
            "游客离开聊天室,让我们深情凝望着他(她)的背影...";
        //遍历会话 socket 链表,向其他会话 socket 转发数据
        for(pos = pView->m_pSessionList->GetHeadPosition();
            pos!=NULL;)
        {
            CSessionSocket *socket = (CSessionSocket *)
                pView->m_pSessionList->GetNext(pos);
            socket->Send(temp,temp.GetLength());
        }
        CAsyncSocket::OnClose(nErrorCode);
}
```

8.2 客户端实现

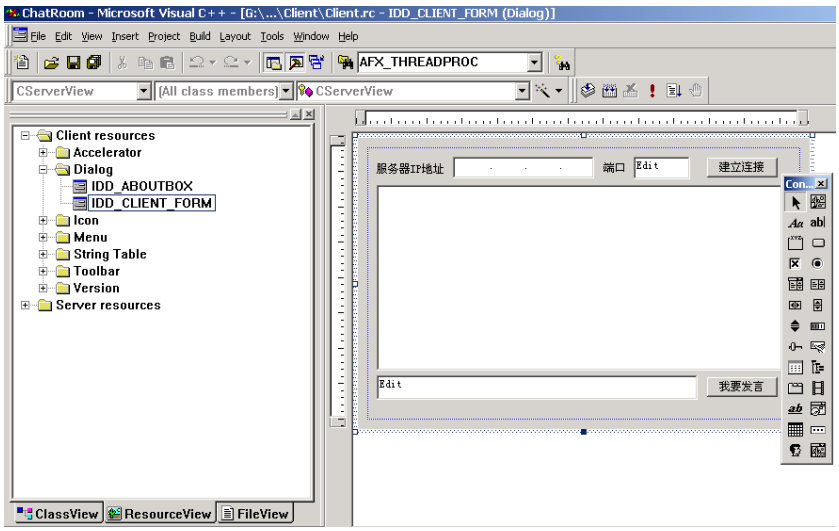
- 创建客户端工程，工程名为 Client，其他设置过程和服务器端完全一样。
- 在 ClientView.h 中为该类添加 public 类型属性成员变量。

  CSessionSocket *m_pSocket;//会话 socket 指针
- 在 ClientView.cpp 中的构造函数中最后添加代码：

  m_pSocket = new CSessionSocket();//创建会话 socket 对象

  在析构函数中添加代码：

  delete m_pSocket; //删除会话 socket 对象
- 在资源编辑器中编辑 ClientView 的界面如下：



- 通过 Class Wizard 加入 CSessionSocket 类。该类的基类为 CAsynSocket，其作用为和服务器会话。同样利用 Class Wizard 为该类添加对 OnRceive 事件响应函数。事件响应函数的代码如下：

```
//当收到数据时的事件响应函数
void CSessionSocket::OnReceive(int nErrorCode)
{
    char buf[1024];
    memset(buf, 0, sizeof(buf));

    // 接收数据
    Receive(buf, 1024);
    // 在视图中显示信息
    CClientView* pView = (CClientView*)
                ((CMainFrame*)AfxGetApp()->m_pMainWnd)->
                GetActiveView();
    pView->m_listData.AddString(CString(buf));
    CAsyncSocket::OnReceive(nErrorCode);
}
```

- 最后利用 Class Wizard 为视图中的二个按钮添加 OnClick 的事件响应函数如下：

```
// CClientView message handlers
//点击"我要发言"按钮的事件响应函数
void CClientView::OnSend()
```

```
{
    UpdateData(TRUE); //得到用户在输入编辑框里输入的字符串

    // 发送数据
    m_pSocket->Send(m_strMsg, m_strMsg.GetLength());

    //清空编辑框
    m_strMsg = _T("");
    UpdateData(FALSE);
}

//点击"连接服务器"按钮的事件响应函数
void CClientView::OnConn()
{
    //得到用户输入的服务器 IP 地址
    BYTE f0,f1,f2,f3;
    CIPAddressCtrl *pIPCtrl =(CIPAddressCtrl * )
                        this->GetDlgItem(IDC_IPADDRESS);
    pIPCtrl->GetAddress(f0,f1,f2,f3);
    CString ip;
    ip.Format("%d.%d.%d.%d",f0,f1,f2,f3);

    //得到用户输入的端口号
    UpdateData(TRUE);

    // 与服务器建立连接
    m_pSocket->Connect(ip, m_nPort);
}
```