第3章 通 信

在第1章中我们讨论了怎样使用 SendMessage()和PostMessage()函数与应用程序的窗口通信。如果要与其他的应用程序或者另一个系统中的应用程序通信又该怎么去做呢？

本章将概述6种应用程序与外界通信的方法，其中包括与其他操作系统、 Internet以及串行线之间的通信。另外，还要探讨这些通信连接方式如何允许程序员控制其他的应用程序甚至使用它们。此外还要介绍用于与其他应用程序共享大量数据的方法。

3.1 进程间通信

应用程序之间的通信，不管是在同一系统上或是通过网络进行，都叫做进程间通信 (IPC: Interprocess Communication)。MFC应用程序为进程间通信准备了下列6种可用的途径：

■ 窗口消息(Windows message)，允许与其他应用程序的窗口通信。这与先前用于与自己的应用程序窗口通信所采用的窗口消息是一个概念。

■ 动态数据交换(DDE: Dynamic Data Exchange)，通过维护全局分配内存使得应用程序间传递大量数据成为可能。其方式是在一块全局分配内存中手工放置大量的数据，然后使用窗口消息传送该内存指针。 DDE提供了一种标准使得任何遵守该标准的应用程序都可以采用它。

■ 消息管道(Message Pipe)，用于设置应用程序之间的一条永久通信通道，通过该通道可以像自己的应用程序访问一个平面文件一样读写数据。 DDE的数据传送速度使人失望，但是采用消息管道就可以无缝地发送数据到其他应用程序，而这些应用程序可能在其他系统上。

■ Windows套接字(Windows Socket)，它具备消息管道的所有功能，但遵守一套通信标准使得不同操作系统之上的应用程序之间可以互相通信。这些操作系统可以是非 Windows系统，例如UNIX系统。实际上在开发新的应用程序时，使用 Windows套接字比消息管道和DDE都更为有利。

■ Internet通信，它让应用程序从Internet地址上载或者下载文件。

■ 串行/并行通信(Serial/Parallel Communication),它允许应用程序通过串行或者并行端口与其他应用程序通信。

3.1.1 通信策略

虽然以上的每一种通信方式都是用不同的 Windows API或者MFC类调用的，但是使用它们的过程却大同小异：

1) 用Windows API或者MFC类打开与其他应用程序之间的通信连接。

2) 读取或者写入对方应用程序。对某些方式而言可能意味着发送及接收消息。对另一些方式，该过程则与读写普通文件没有太大的不同。

3) 关闭连接。

3.1.2 同步和异步通信

每一种通信方式都可以是同步或者异步的。同步 (synchronous)通信使应用程序暂停，直到它完成对其他应用程序的读或者写操作。异步 (asynchronous)通信则允许应用程序继续运行，当系统完成读写操作时就会用一个事件标记或者调用指定的函数通知用户任务已经完成。

虽然异步通信听起来更为适当，特别是在应用程序要一次同其他几个应用程序会话的情况下更是如此，但这种方式并不是面向对象的。程序员不是需要跳出对象提供静态回调函数就是需要费心处理事件标记。同时，设置异步连接也更为复杂，更容易产生程序错误 (bug)。

解决方案是采用同步通信，但要将每个读写操作放进它自己的线程中，这样这些操作就不会再阻止应用程序之间的通信。当操作完成后，线程可以自动地将读取的数据存回应用程序。当然，它也可以设置事件标记，但这需要在更为友好的 MFC类的环境下才可以。实例 51和实例52说明了怎么做到这一点。

注意 应用程序创建自己线程的能力只有Windows 95和Windows NT操作系统才具备。如果是在Windows 3.1系统平台上开发，就需要使用异步通信。对异步通信来说，操作系统本身将读写操作放进了自己的特定线程。

注意 用Windows套接字的术语来说：同步通信的应用程序被认为是阻塞 (blocking)类型的。

下面研究以上的每一种通信方式，其中包括如何打开和关闭通信连接以及如何利用连接进行读写操作。

3.2 窗口消息

先前所述的窗口消息系统允许应用程序控制其窗口并处理来自用户的命令消息，这套消息系统同样可以用于控制并处理来自其他应用程序的命令。正是由于全局分配每个窗口对象才使得这一切成为了可能，这样任何应用程序都能够发送消息到其他应用程序窗口。关键在于必须知道要同哪个窗口对话。

3.2.1 打开和关闭

打开与其他应用程序之间的通信连接，涉及到确定应用程序要发送消息到某个窗口所对应的窗口句柄。麻烦在于，因为每个句柄不是在编译或者链接应用程序时创建而是在窗口被创建的时候才生成，所以每次运行时都有可能不同。这样就不能知道该句柄。而且，也不可能询问其他窗口其句柄是什么。因为正是需要那些窗口的句柄才可能发出这一请求！这里有 3种技术可以得到其他窗口句柄：

■ 当一个父应用程序创建其子应用程序的时候，它将属于自己的一个窗口的句柄作为参数传送。子应用程序然后可以将该句柄发送给自己的一个窗口句柄，这样通信就可以进行了。一般情况下，这些应用程序之间交换的句柄都是普通句柄，但总是被创建为只进行通信的窗口所隐藏。关闭连接也就只是关闭这些窗口。

■ 但是，如果目标应用程序已经在运行了，则可以调用 Windows API函数:FindWindow()以找到适当的窗口。该函数将返回一个句柄，该句柄指向任何匹配特定的名字或者使用了特定的窗口类的窗口。为了帮助 FindWindow()函数发现正确的窗口，可以用唯一的名字或者

Windows 类创建如上所述的普通消息窗口。

■ 得到正确窗口句柄的最后一种方式是采用 Windows API函数BroadcastSystemMessage (), 该函数将发送一个消息给每一个当前正在运行的应用程序。使用 BroadcastSystemMessage ()函数可以广播包含自己应用程序窗口句柄的特定消息，处理该消息的应用程序则随后以其句柄作为答复，从而完成这个循环。请参考实例 49来了解该方法。

3.2.2 读与写

到现在为止，与其他应用程序通信已经成为一件并不复杂的事情了，所要做的仅仅是将窗口句柄插入::SendMessage()或者::PostMessage()之中。还可以用 Attach()函数将窗口句柄封装到CWnd类中，然后利用 CWnd的其他方法来发送和寄送消息。发送消息向应用程序提供了同步通信，寄送消息则提供异步通信。

虽然一般情况下需要向目标窗口发送标准的窗口消息 (如WM_CLOSE，WM_MOVE等)，但有时也要创建自己的窗口消息来完成某些特定任务。

可以在WM_USER以上范围内定义消息来创建自己的窗口消息 ID。如下所示：

```
#define  WM_MYMESSAGE  WM_USER + 1
```

但是，创建新窗口消息ID的更可靠的方法是向系统注册自己的消息，如下所示：

```
#define  IDString   "MyMessage"
MsgID = ::RegisterWindowMessage( IDString );
```

这里IDString是每个应用程序都关注的唯一文本字符串，MsgID是要创建的消息ID。因此，这种方式比为自己的消息 ID分配数值而可能引起应用程序内部的混乱效果要好。另外还可以用描述字符串如OpenFile和::RegisterWindowMessage()函数来创建自己的消息ID。如果某应用程序抢先注册了 OpenFile，接下来使用 OpenFile的应用程序就可以调用 ::Register WindowMessage ()返回同样的消息ID，因此不会有混淆的危险。

请参考实例49了解如何正确使用这些函数。

3.2.3 回顾

利用窗口消息向其他应用程序发送消息实际上是先得到其他应用程序句柄，然后向其发送消息。请参见图3-1回顾全过程。

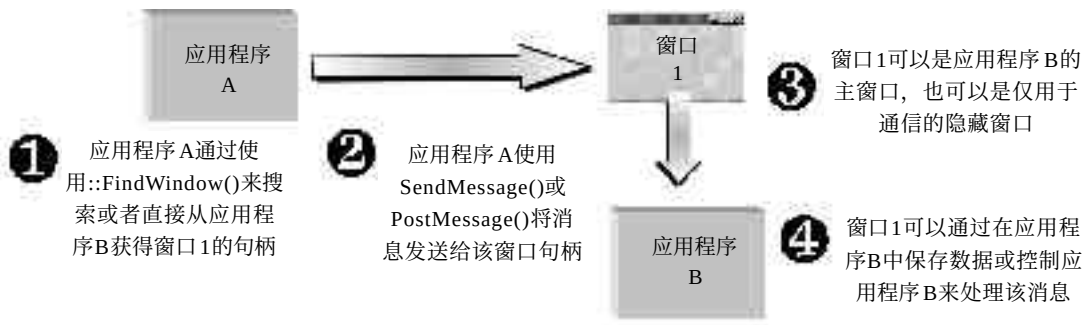

图3-1 窗口消息发送

3.3 动态数据交换

SendMessage()和PostMessage()函数允许同时向其他应用程序发送两个整型数值。因此传送1000个字节就需要250个消息。为了一次发送更多消息，可以将数据填满一块全局分配内存并将其句柄作为参数之一传送。这种方式仅允许同自己设计的其他应用程序交换数据。为了与非自己设计的应用程序交换数据就要采用 DDE(Dynamic Data Exchange)的标准，该方法已经在动态数据交换库(DDEML)中实现。

3.3.1 客户/服务器

DDE的采用引进了客户应用程序和服务器应用程序 (Client/Server)的概念。在DDE的情况下，几乎所有的数据都驻留在服务器上供客户应用程序来访问。此时的通信方式可以形容为：服务器打出一块牌子，上面写着“ 对外办公 ”，客户则从服务器读写数据。

客户/服务器通讯方式将在以后章节详细讨论。

3.3.2 打开和关闭

为了打开通信连接，服务器应用程序通过广播自己的方式开始启动，程序如下所示：

```
// initialize DDE services
UINT DdeInitialize(
   LPDWORD pidInst,              // a pointer to the instance
   PFNCALLBACK pfnCallback,      // a callback function (see below)
   DWORD afCmd,                  // command and filter flags
   DWORD ulRes                   // reserved
   );
// create a string handle to the name of the service we  are creating
HSZ DdeCreateStringHandle(
   DWORD idInst,                 // returned by DdeInitialize() above
   LPTSTR psz,                   // pointer to service name string
   int iCodePage                 // CP_WINANSI or CP_WINUNICODE for Unicode
   );
// register the service
HDDEDATA DdeNameService(
   DWORD idInst,                 // returned by DdeInitialize() above
   HSZ hsz1,                     // string handle to service name
   0L,                           // reserved
   UINT afCmd                    // service name flags
   );
```

然后客户应用程序如下连接到服务器：

```
UINT DdeInitialize( ... );       // as seen above
HCONV DdeConnect(
   DWORD idInst,                 // returned by DdeInitialize() above
   HSZ hszService,               // handle to service name string
   HSZ hszTopic,                 // handle to topic name string (optional)
   PCONVCONTEXT pCC              // pointer to structure with context data
   );
```

为了在服务器应用程序中处理该连接，服务器的回调函数必须处理一个 XTYP_

CONNECT消息：

```
HDDEDATA CALLBACK DdeCallback( type ... )
{
   switch( type )
   case XTYP_CONNECT:
      return TRUE;                  // return FALSE to reject connection
}
```

客户随后关闭该连接：

```
BOOL DdeDisconnect(
   HCONV hConv                      // handle returned by DdeConnect() above
   );
```

3.3.3 读和写

客户可以使用如下方式从服务器请求数据：

```
HDDEDATA DdeClientTransaction(
   NULL, 0,
   HCONV hConv,                     // handle returned by DdeConnect() above
   HSZ hszItem,                     // string handle of name of data to read
                                    //       and return
   UINT wFmt,                       // format of data based on clipboard formats
   XTYP_REQUEST,                    // <<<<<<<<<<<<<<transaction type
   DWORD dwTimeout,                 // should request time-out
   LPDWORD pdwResult                // pointer to result
   );
```

返回的数据实际上是一个DDE数据对象句柄，可以如下访问它：

```
LPBYTE DdeAccessData(
   HDDEDATA hData,                  // handle returned by DdeClientTransaction()
   LPDWORD pcbDataSize              // pointer to data length
   );
```

或者自动拷贝到本地缓冲区中：

```
DWORD DdeGetData(
   HDDEDATA hData,                  // handle returned by DdeClientTransaction()
   LPBYTE pDst,                     // pointer to destination buffer
   DWORD cbMax,                     // amount of data to copy
   DWORD cbOff                      // offset to beginning of data in hData
   );
```

如下所示，服务器在其回调函数中处理对数据的请求。已请求数据的数据句柄由DdeCreateDataHandle()创建：

```
case XTYP_ADVREQ:
{
   HDDEDATA hData =
      DdeCreateDataHandle(
         DWORD idInst,              // returned by server's DdeInitialize()
         LPBYTE pSrc,               // pointer to source buffer
         DWORD cb,                  // length of source buffer
```

```
        DWORD cbOff,          // offset from beginning of source buffer
        HSZ hszItem,          // handle to item name string
        UINT wFmt,            // format of data in clipboard formats
        UINT afCmd            // creation flags
        );
        return hData;
}
```

客户如果要向服务器写数据，它可以将数据放进另一个数据句柄或者直接传送数据：

```
HDDEDATA DdeClientTransaction(
   LPBYTE pData,              // pointer to data to pass to server
                              //     or data handle
   DWORD cbData,              // length of data
                              //     or 0ffffffffh if pData is a data handle
   HCONV hConv,               // handle returned by DdeConnect() above
   HSZ hszItem,               // string handle of name of data to read
                              //     and return
   UINT wFmt,                 // data format based on clipboard formats
   XTYP_POKE,                 // <<<<<<<<<<<<<<transaction type
   DWORD dwTimeout,           // should request time-out
   LPDWORD pdwResult          // pointer to result
   );
```

但是服务器必须在其回调函数中接收该数据：

```
case XTYP_POKE:
   DdeGetData();              // as before
```

3.3.4 其他DDE函数

DDEML比起前面介绍的内容提供了更多的功能。例如，可以设置一个通告循环 (advise loop)，该通告循环在服务器和客户所共享的数据被修改的情况下立即通知它们。

DDEML还支持允许客户要求服务器完成某个命令的标准。将要执行的命令用诸如 Open file.dat这样的字符串表达。如果这些命令属于服务器的指令表，则服务器将解析这些字符串并执行命令。

3.3.5 MFC支持

SDI和MDI应用程序自动支持3种DDE字符串：

[open("file.dat")] —— 告诉应用程序打开指定文件。

[print("file.dat")] —— 告诉应用程序打印指定文件。

[printto("file.dat", "printer", "driver", "port")] —— 告诉应用程序将指定文件打印到指定的打印设备。

MFC应用程序对命令行标记 /dde具有内置支持，它只是一开始隐藏应用程序以便在打印文件的时候应用程序在后台进行操作。

但是除了这种有限的支持以外， MFC并不支持DDE。这是因为：应用程序已经从仅在一个简单系统上运行迁移到了在多个系统之上运行其多个部分。一个系统的共享全局内存在该系统之外是不可以被访问的。换句话说， DDE这时已经过时了。

窗口消息也不能跨越不同的计算机。事实上，消息管道这种通信方式正是为了弥补这种缺陷而创建。

3.3.6 回顾

DDE允许大量数据通过对全局内存的使用从而在应用程序之间共享。由于全局内存不能通过网络共享，结果DDE也没有了它的前途。为回顾这个过程可参见图 3-2。

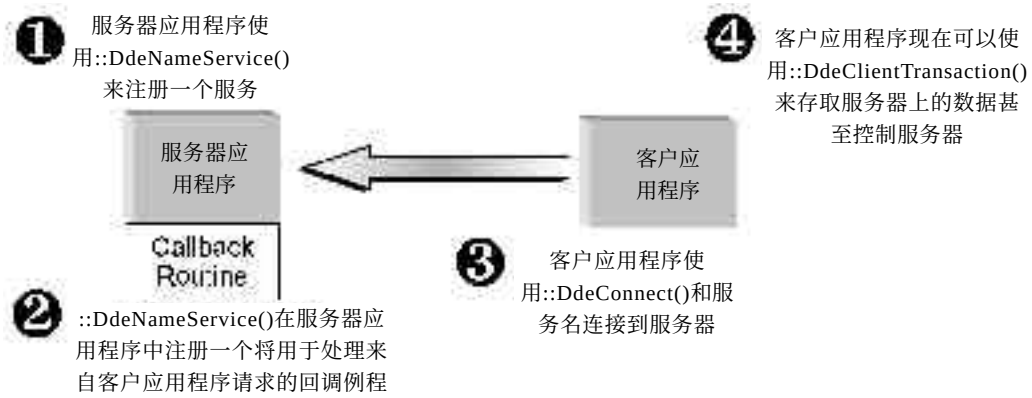

图3-2 DDE

3.4 消息管道

为了实现应用程序之间的连续通信， Windows API提供了名为消息管道(Message Pipes)的机制，它是一种缓冲和句柄系统，能够允许应用程序同其他系统上的应用程序通信。有两种类型的消息管道：异步和命名管道。异步管道通常用于父“外壳 (shell)”应用程序和其子应用程序之间通信。而命名管道则可以与任何应用程序对话，包括那些在其他系统之上的应用程序。本章仅探讨命名管道。

注意 消息管道实际上也像DDE一样过时了。这里介绍它们仅仅是出于使介绍的内容较完整的缘故。对于新的程序开发，应该考虑使用 Windows套接字，它将在下一部分讨论。

3.4.1 打开和关闭

为了使应用程序创建命名管道以便于其他应用程序连接，可以使用；

```
HANDLE CreateNamedPipe(
  LPCTSTR lpName,          // name of pipe in the form:
                           // where you provide
                           // servername and pipename
  DWORD dwOpenMode,        // open mode:
                           // PIPE_ACCESS_DUPLEX means
                           //    pipe is read/write
                           // PIPE_ACCESS_INBOUND means read only
                           // PIPE_ACCESS_OUTBOUND means write only
  DWORD dwPipeMode,        //    pipe-specific modes:
```

```
                                    // PIPE_TYPE_BYTE means data is
                                    //    written as a stream of
                                    //    potentially unrelated bytes.
                                    // PIPE_TYPE_MESSAGE means each time
                                    //    a block of data is written, it's
                                    //    treated like a message packet.
    DWORD nMaxInstances,            // max number of instances of this pipe
    DWORD nOutBufferSize,           // output buffer size, in bytes
    DWORD nInBufferSize,            // input buffer size, in bytes
    DWORD nDefaultTimeOut,          // time-out time, in milliseconds
    LPSECURITY_ATTRIBUTES lpSecurityAttributes
                                    // pointer to security attributes
    );
```

其他应用程序要连接到该管道需要调用：

```
HANDLE hPipe = CreateFile(
    LPCTSTR lpFileName,             // pipe name created above
    DWORD dwDesiredAccess,          // GENERIC_READ and/or
                                    //    GENERIC_WRITE
    DWORD dwShareMode,              // FILE_SHARE_READ and/or
                                    //    FILE_SHARE_WRITE
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                                    // security
    DWORD dwCreationDisposition,    // always OPEN_EXISTING
    DWORD dwFlagsAndAttributes,     // file attributes:
                                    // FILE_FLAG_OVERLAPPED makes this
                                    //    an asynchronous connection
    HANDLE hTemplateFile            // copy attributes from this file
    );
```

为了关闭管道，需调用：

```
CloseHandle(
    HANDLE hPipe
    );
```

3.4.2 读和写

一旦管道建立，就可以使用下列两个函数来读写它：

```
BOOL ReadFile(
    HANDLE hPipe,                   // pipe handle created above
    LPVOID lpBuffer,                // buffer that receives data
    DWORD nNumberOfBytesToRead,     // number of bytes to read
    LPDWORD lpNumberOfBytesRead,    // number of bytes actually read
    LPOVERLAPPED lpOverlapped       // used with asynchronous
                                    //    communication
    );

BOOL WriteFile(
    HANDLE hPipe,                   // pipe handle
    LPCVOID lpBuffer,               // data to write
```

```
    DWORD nNumberOfBytesToWrite,      // number of bytes to write
    LPDWORD lpNumberOfBytesWritten,   // number of bytes actually
                                      //    written
    LPOVERLAPPED lpOverlapped         // used with asynchronous
                                      //    communication
);
```

虽然消息管道API为支持异步通信还包含了许多其他附加功能，但是只要在以上所述它们自己的线程中使用 ReadFile()和WriteFile()函数就用不着关注它们。异步通信总是更为简单，所以程序错误也少得多。

3.4.3 回顾

消息管道可以建立应用程序间的永久连接并进行通信，就好像是从普通磁盘文件中读写数据，请参见图3-3。

图3-3 消息管道

虽然消息管道通信方式比简单消息发送的方式要先进，但最好还是把它们都忘了。 MFC并不为它们提供任何类，因此在本书中也不提供与之相关的任何实例。消息管道也已经过时了，Windows API提供了更灵活的解决方案，那就是 Windows套接字通信。

3.5 Windows 套接字

Windows套接字(Windows Socket)在本质上是遵守Berkeley软件发布(Berkeley Software Distribution，BSD)(v4.3)UNIX套接字实现的消息管道。因此它允许应用程序与在任何支持该标准系统上的应用程序会话。其中当然包括许多 UNIX应用程序。

虽然Windows套接字一般用于网络通信，但也可以用来实现同一系统上应用程序之间的通信。这样在配置应用程序的时候将具有更大的灵活性。可以将它们安装在所有同样的系统上，也可以选择将其安装到其他系统上的另一些应用程序。

应用程序可以选用3种级别的Windows 套接字支持，其中包括直接 Windows API访问、封装了API的MFC类CAsyncSocket以及更高级的其他MFC类CSocket。出于简化的原因，应当只使用CSocket。因为CSocket由CAsyncSocket派生，所以仍然可以访问低级类的所有功能，而且CSocket提供了一些使得与非Windows系统通信更为方便的功能。

采用Windows套接字的通信由3种套接字完成。除了每个应用程序创建来进行对话的套接字，还有第3类套接字被创建来侦听新的连接。换句话说，当要与其他应用程序建立新的连接的时候，每个应用程序都要创建第 3个套接字来发信号。

3.5.1 打开和关闭

为了让应用程序侦听请求连接，需使用：

```
CListenSocket listenSock;
listenSock.Create(
   UINT nPort                    // between 1025 and 0xffffffff set by you to
                                 //    identify this listener to your other apps
   );
listenSock.Listen()   // start listening
```

这里使用的CListenSocket类是CSocket的派生类，不直接使用 CSocket类的原因在于需要重载其某个成员函数，不久就会了解到这一点。

如果其他应用程序要连接到该套接字，则需要调用：

```
CSocket sock;
sock.Create();                   // take the defaults
sock.Connect(
   LPCTSTR lpszHostAddress,      // system address of application
                                 //    with listening socket specified as:
                                 //    " " or "128.23.1.22" or
                                 //    "localhost" to talk to an
                                 //    application server on
                                 //    the same system
   UINT nHostPort                // the port number specified when creating
                                 //    the listening socket
   );
```

这里出于介绍清晰的目的创建了套接字对象 sock，一般是将它作为类的成员变量或者将其分配在堆上。这样一旦其被销毁，连接也就随之关闭。

为了完成连接，需要重载 CListenSock类的一个名叫 OnAccept()的成员函数。只要CListenSock觉察到其他应用程序正在试图建立连接，该函数就会被调用。在 OnAccept()中还创建了第3个也是最后一个套接字与其他应用程序对话，如下所示：

```
CListenSocket::OnAccept()
{
   CSocket sock;
   listenSock.Accept( sock );
}
```

此外要保证创建套接字是在其不被销毁的生命周期内进行。为了关闭连接可以调用：

```
sock.Close()；
```

3.5.2 读和写

一旦 windows套接字连接已经建立，就可以使用下列两个函数来读写它：

```
int numBytesReceived =           // number of bytes received
   Receive(
      void* lpBuf,               // buffer to contain data
      int nBufLen,               // length of buffer
      int nFlags = 0             // if set to MSG_PEEK, will cause
```

```
                                    //    Receive() to leave received
                                    //    data in socket queue, but data
                                    //    is still copied to lpBuf
        );

int numBytesSent =                  // actual number of bytes sent
   Send(
      const void* lpBuf,            // bytes to send
      int nBufLen,                  // number of bytes to send
      int nFlags = 0
      );
```

Receive()和Send()函数都是同步函数，因此应该如实例 51所示在它们自己线程内运行它。Receive()函数的另一个危险在于：如果所请求的数据字节超过了要发送的数据，则该函数不会处理该消息，为了弥补该功能的不足，应当建立自己的消息，以使它们具有固定的报头来包含定义整个报文大小的数值变量。 Receive()然后可以被设置为用来接收报头的大小，并用该报文大小变量来接收任何其他的数据。

请参考实例51了解更多内容。

3.5.3 通过Windows套接字序列化

使用CSocket的Send()和Receive()成员函数(实际上是从CAsyncSocket继承的)的时候，程序员按照与非Windows MFC系统通信字节顺序和字符串格式负责将报文内容填充到任何相关的数据结构。但是如果使用了其他 MFC类如CSocketFile，这项工作就将由它来承担 (但是两个应用程序必须都是由 MFC创建的)。CSocketFile利用在存储文档时曾使用的序列化(Serialization)技术来完成该工作。要更为详细地了解序列化技术，可以参考 MFC文档或者《Visual C++ MFC编程实例》。

可以使用序列化从应用程序读数据：

```
CSocketFile sockFile( &sock );    // where sock is your socket object
CArchive ar( &sockFile,CArchive::load );

ar >> data;                       // and any other data
```

写数据到应用程序则使用：

```
CSocketFile sockFile( &sock );
CArchive ar( &sockFile,CArchive::store );

ar << data;
```

3.5.4 数据流和数据报

Windows套接字可以是两种方式之一：数据流 (Stream)或者数据报(Datagram)。本书提供的所有实例采用的都是数据流模式。只有一些应用程序需要数据报。数据报开销不大但是很不可靠。使用数据报的一个例子是：同步网络上所有系统的时钟。因为这种类型的应用程序持续地发出消息，也许一个小时一次，丢失一两个报文也就无所谓了。

3.5.5 回顾

Windows套接字允许两个或者两个以上的应用程序相互通信，其通信平台可以在 Windows 非系统之上。应用程序首先创建特殊的侦听套接字以侦听来自其他应用程序的会话请求。该侦听应用程序随后创建另一个套接字以准备对话，请参见图 3-4。

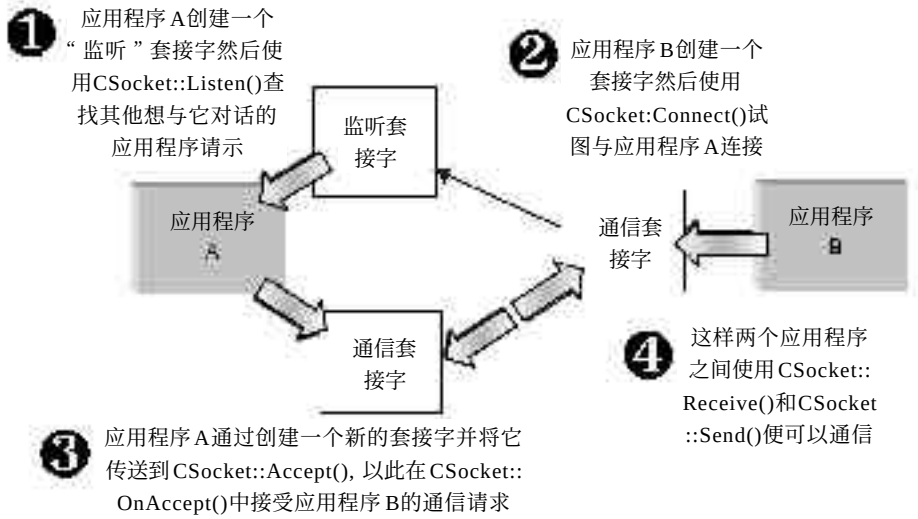

图3-4 Windows套接字

3.6 串行/并行通信

串行和并行通信通常通过系统背后的端口和嵌入设备对话，例如网络节点、打印机或者电话交换机。与这些设备的通信与访问磁盘普通文件没有太大不同。实际上使用了同样的 MFC CFile类。与一般通信的唯一不同是：这种通信方式在一个线程内完成同步读写并且不会主动读取更多可用字节，请参考实例 52。

3.6.1 打开和关闭

为通信打开一个串行或者并行端口，需使用：

```
CFile file;
CFileException e;
file.Open(
   portName,                    // examples "COM1", "COM2", "LPT2"
   CFile::modeReadWrite,
   &e
   );
```

3.6.2 读和写

从该端口进行读写操作，需使用：

```
UINT nBytes =                   // actual number of bytes read
   Read(
      void* lpBuf,              // buffer to store bytes
```

```
        UINT nCount             // number of bytes to read
        );

file.Write(
    void* lpBuf,                // buffer to write
    UINT nCount                 // number of bytes to write
    );
```

3.6.3 配置端口

所有并行端口创建时都是一样的，但是必须设置串行端口以匹配与其对话的设备。这就意味着需要相同的波特率、奇偶位和停止位等。虽然可以通过操作系统设置这些参数，但也可以用Windows API中的SetCommState()函数来设置它们。一般地，可用如下程序来设置当前的配置：

```
DCB dcb;
::GetCommState( file.m_hFile, &dcb );
dcb.BaudRate = 1200, ...;
dcb.ByteSize = 7 or 8;
dcb.StopBits = 0,1,2 = 1, 1.5, 2;
dcb.Parity = 0-4 = no,odd,even,mark,space;
::SetCommState( file.m_hFile, &dcb );
```

串行口和并行口可以超时通信，这有时会不合乎要求，例如在线程内执行一个同步读写就会导致消息在到来之前一直读下去。为了关闭该项超时功能，可以使用 ::SetCommTimeout()函数，如下所示：

```
COMMTIMEOUTS cto;
::GetCommTimeouts( file.m_hFile, &cto );
cto.ReadIntervalTimeout = 0;
cto.WriteTotalTimeoutMultiplier = 0;
cto.WriteTotalTimeoutConstant = 0;
::SetCommTimeouts( file.m_hFile, &sto );
```

3.6.4 回顾

串行和并行通信可以像访问普通文件那样通过 API 来实现，系统的虚拟驱动程序负责执行特定的工作。只有串行通信需要额外考虑与对方设备的参数匹配问题 (例如波特率等)，请参见图3-5。

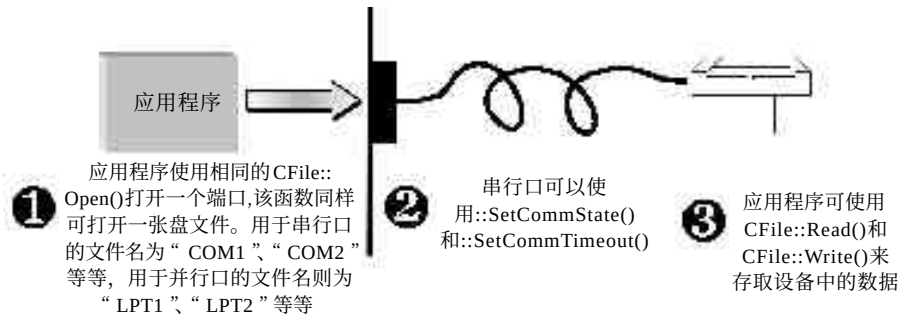

图3-5 串行通信和并行通信

3.7 Internet通信

有几个MFC类封装了 Windows API的Internet Extension(Internet扩展功能、WinInet)，它允许C++利用下述4种协议之一访问Internet：文件传送协议(FTP:File Transfer Protocol)、超文本传送协议 (HTTP:Hypertext Transfer Protocol)、Gopher或者文件形式。可以用 MFC的CInternetSession类连接到一个Intenet的web地址。也可以直接用4种MFC类之一直接访问文件：CStdioFile、CHttpFile、CGopherFile或者CInternetFile等。或者用下列3种连接类控制一个web站点:CFtpConnection、ChttpConnection、CGopherConnection等。这些文件类和连接类都是由CInternetSession类创建的。

3.7.1 打开和关闭文件

为了打开一个Internet文件，首先要打开一个Internet会话：

```
CInternetSession();                    // there are several arguments but you
                                       //    will typically use the defaults
```

然后用CInternetSeesion类的OpenURI()成员函数打开一特定类型的文件：

```
CStdioFile*pFile =                     // see Table 3.1 for returned object type
   session.OpenURL(
      LPCTSTR pstrURL                  // see Table 3.1 for sample values
      );
// There are several other arguments, however you will typically
//    use the default values.
```

打开的URI类型取决于CInternetSeesion创建的类对象，如表3-1所示：

表3-1 OpenURI()参数说明

URL类型	返回的Internet类指针	URL类型	返回的Internet类指针
file://	CStdioFile*	gopher://	CGopherFile*
http://	CHttpFile*	ftp://	CInternetFile*

由该方法创建的文件对象具有只读属性。为了写文件以控制 web站点，参考以下的打开和关闭连接的内容介绍。

3.7.2 读文件

读取Internet文件类型和读取普通文件完全一样。实际上， CHttpFile、CGopherFile以及CInternetFile都是从CStdioFile派生的，这样就可以按照字节或者字符串数目读取文件；

```
UINT pFile->Read( void* lpBuf, UINT nCount );

pFile->ReadString( CString str );
```

CHttpFile允许通过对象和动词(verb)访问HTTP文件。

3.7.3 打开和关闭连接

CInternetSession类有3个成员函数允许打开web站点，使用连接不仅可以读写文件还可以

控制站点。例如，FTP连接就允许采用编程的方法在ftp站点之上执行ftp命令。

1. FTP连接

为打开ftp连接，可以使用：

```
CFtpConnection* ftp = GetFtpConnection();
CInternetFile* pFile =
   ftp.OpenFile(
      LPCTSTR pstrFileName,
      DWORD dwAccess = GENERIC_READ,                    // and/or
                                                        // GENERIC_WRITE
      DWORD dwFlags = FTP_TRANSFER_TYPE_BINARY,         // use default
      DWORD dwContext = 1                               // use default
      );
```

要打开该站点的文件，可以使用：

```
CFtpFile *ftpFile = ftp.OpenFile(
     // same as CStdioFile open
   );
```

2. Gopher连接

要打开Gopher连接，可以使用：

```
CGopherconnection* gopher = GetGopherConnection()；
```

要打开该站点的文件，则可以使用：

```
CGopherFile *gopherFile = gopher.OpenFile(
     // same as CStdioFile Open()
   );
```

3. HTTP连接

要打开Http连接，可以使用：

```
CHttpConnection* http = GetHttpConnection()；
```

要打开该站点的文件，则可以使用；

```
CHttpFile* http.OpenRequest(
   // please refer to your MFC documentation for these arguments
   );
```

3.7.4 其他Internet类

另外3种令人感兴趣的Intenet客户类是CFtpFileFind、CGopherFileFind和CGopherLocator。CFtpFileFind和CGopherFileFind由CFileFind类派生，可以用于在Internet上寻找文件的位置。CGopherLocator类从gopher服务器得到一个gopher“定位器(locator)”，并使之对CGopherFilefind可用。

3.8 通信方式小结

采用什么通信方式取决于应用程序。例如，有限的通信方式应该采用窗口消息。而对持续或者高级通信则应该使用Windows套接字。如果要与嵌入设备，如打印机等通信则要使用串行/并行通信，其余则使用Internet通信。

在新的应用程序开发过程中建议不要采用 DDE或者消息管道方式。 Windows套接字对进程间通信而言是一种相对灵活得多的解决方案，而且具备支持基础类的相关类。

尽管DDE比Windows Socket的数据通信速度要快，但在一次只通过电缆发送一个字节时就无法替代后者了。

3.9 共享数据

到目前为止，本书已经覆盖了应用程序间主动通信 (active communication)的内容，主动通信就是应用程序双方都要参加的通信。应用程序还有两种被动通信方式允许其他应用程序访问其数据，如下所示：

■ 共享内存文件(Shared Memory File)，就是利用前面所讨论的DDE方式使用共同的全局分配内存，因而不能从其他计算机访问。

■ 文件映射(File Mapping)，允许应用程序之间共享文件和内存，甚至通过网络和其他Windows计算机之间也可以这样做。

注意 如果要与非Windows平台(如UNIX)共享数据，文件映射对此还不能适用。在这种情况下，最好还是使用Windows套接字。但不要担心，文件映射并不比Windows套接字能向你的应用程序提供更快的数据访问速度，它仍然只是通过网络一次一个字节来共享数据。

注意 应用程序也可以通过剪贴板共享数据，但是剪贴板通常用于用户和应用程序交互。

3.10 共享内存文件

如果并不需要遵守 DDE的标准并且也不需要通过网络共享数据，那么可以使用自己的全局分配内存来与其他应用程序共享数据。实际上， MFC的CShareFile类使得创建和访问全局内存方便得犹如创建普通文件。

3.10.1 创建和销毁

创建和销毁CShareFile类的实例将会创建和销毁全局内存：

```
CSharefile file;
```

3.10.2 读和写

读写共享内存文件和读写普通文件是完全一样的。因此，作为完整的数据传送例子，可如下编程：

```
CSharedFile file;
file.Write( buffer, nBytes );
// extract global memory handle from CFile object to send another application
HGLOBAL hgbl = file.Detach();
SendMessage( hWnd, WM_MYMESSAGE, ( WPARAM ) hgbl,0 );
```

在接收程序中读取数据则使用：

```
HRESULT OnMyMessage( WPARAM wParam, LPARAM lParam )
{
```

```
    CSharedFile file;
    // encapsulate global memory handle in this CFile object
    file.SetHandle( ( HGLOBAL ) wParam );
    file.Read( buffer,nBytes );
}
```

3.10.3 回顾

可以通过全局内存共享数据，这是用 CShareFile类来实现的，所采用的函数如同访问普通文件一样，请参见图3-6。

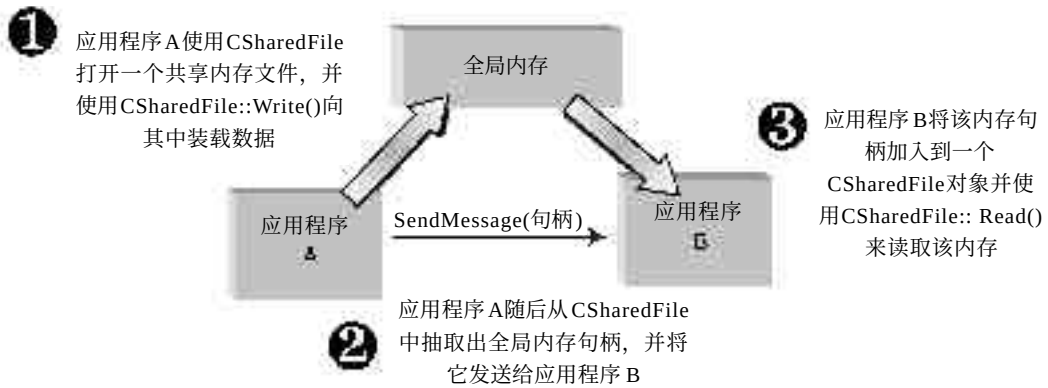

图3-6 共享内存文件

3.11 文件映射

如果要在应用程序之间 (包括其他Windows平台上的应用程序)共享数据，可以使用文件映射。与全局分配内存不能一次由一个以上应用程序访问的特点所不同的是，文件映射允许两个或者两个以上的应用程序同时访问共享内存。举个例子，可以在两个应用程序中共享一个数组BOb[100]，在应用程序 A中的Bob[23]里存储34，将在应用程序 B的Bob[23]中同样出现34。

这种共享方式实际上发生在文件中，所以被命名为文件映射。在一个系统中共享数据的时候，一般只是使用缺省的文件，该文件实际上是用于向系统提供虚拟内存的交换文件。如果要通过网络共享则需要自己打开文件并提供其句柄。

3.11.1 打开和关闭

要打开一段交换文件以共享内存，需使用：

```
m_hMap = ::CreateFileMapping(
    ( HANDLE ) 0xffffffff,          // or can be an open file handle
    0,                              // security
    PAGE_READWRITE,                 // or PAGE_READONLY or PAGE_WRITECOPY
    0,                              // size - - high order
                                    //     (required if no file handle)
    0x1000,                         // size - - low order
                                    //     (required if no file handle)
```

```
    MAP_ID                          // unique id - - required if no file handle
    );
```

返回的映射句柄可被转变为内存指针，如下所示：

```
m_pSharedData = ::MapViewOfFile( m_hMap,
    FILE_MAP_WRITE,                 // or FILE_MAP_READ, FILE_MAP_COPY
                                    //    (FILE_MAP_WRITE is read/write)
    0,                              // offset - - high order
    0,                              // offset - - low order
    0                               // number of bytes (zero maps entire file)
    );
// When using swap file, offset must be zero(0)
```

要关闭共享内存，需使用：

```
::UnmapViewOfFile(m_pSharedData);
::CloseHandle(m_hMap);
```

3.11.2 读和写

假设正如上面提到的，应用程序之间正在共享一个名叫 Bob的数组，可以在应用程序中使用以下代码：

```
int *Bob = ( int * ) pSharedData;
Bob[23] = 34;
```

3.11.3 数据同步

在同一系统上对映射内存的所有访问都是同步的。换句话说，两个应用程序不能同时访问同一内存。这就阻止了一个应用程序在其他应用程序正对内存区进行写入操作的同时向其中读取数据。

通过网络映射文件也可能出现麻烦。如果正在共享的文件在另一个系统之中，则应用程序实际上是在写入网络缓存，而该缓存超过一定时间即被发送到其他系统。与此同时，其他系统上的应用程序也可能正在写该文件，在这种情况之下需要保证应用程序之间不互相竞争。

3.11.4 回顾

内存的共享可以用文件的方式来实现。对两个应用程序而言则没有其他办法共享同一地址空间，请参见图3-7。

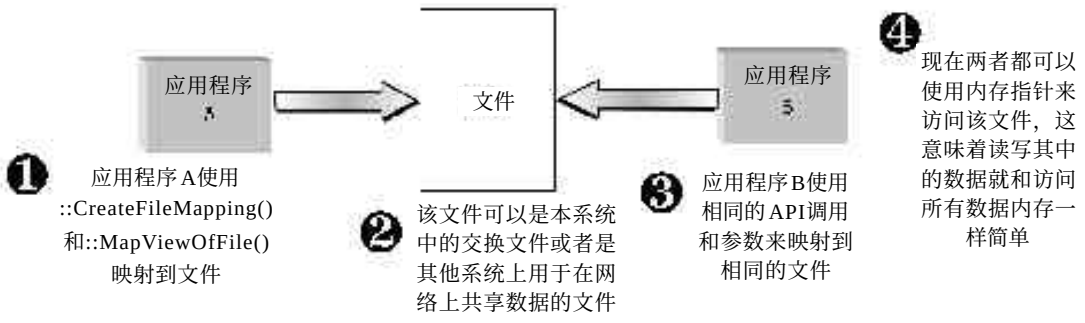

图3-7 文件映射

3.12 客户/服务器

程序员不仅能够向其他应用程序发送消息或者与其共享数据，甚至还能通过间接调用其成员函数以访问其功能。按照这种方式共享其功能的应用程序叫服务器，就是提供服务的一方。而访问这些功能的应用程序则叫做客户，也就是接受服务的客户。

允许应用程序与其他应用程序分享其功能，可以通过以上提到的任何通信方式来完成，基本上采用同样的步骤：

1) 创建新的命令消息代表想要访问的函数 (例如：IDC_CALL_HOME是对应Home()方法的)。

2) 向提供其功能的应用程序 (服务器)发送该消息并在同一消息中传递任何所需函数的必要参数:

```
wParam = x;
lParam = y;
SendMessage( hWnd,IDC_CALL_HOME,wParam,lParam );
```

3) 服务器然后将消息和参数转变为对所需要功能的实际函数调用：

```
case IDC_CALL_HOME:
   x = wParam;
   y = lParam;
   res = Home( x,y );
   :  :  :
```

4) 由函数返回的任何值连同该消息都被返回给客户：

```
return res;
```

或者与新的消息一道：

```
wParm = y;
SendMessage( IDC_CALL_HOME_REPLY, wParam, lParam) ;
```

5) 客户然后通过把这些值放进本地内存而响应一个应答消息：

```
case IDC_CALL_HOME_REPLY:
   y = wParam;
```

请参见图3-8概览该技术：

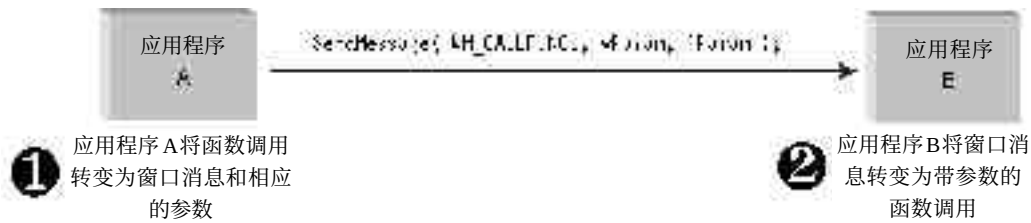

图3-8 间接调用外部函数

3.12.1 传递调用参数

如何传递参数取决于使用什么通信方法。正如先前看到的，窗口消息机制允许通过消息传递两个参数。为了传递额外的参数，就需要将它们填充到某种数据结构中，并通过 DDE、

文件映射或者全局分配内存等方式传递该结构。

不能仅仅把该结构的指针传递给其他应用程序，这是因为该结构存在于不同于其他应用程序所处的地址空间(指向一个应用程序的指针不会指向其他应用程序中的同样数据)。这也就意味着不能向消息结构中加入指针，但整个数组内容和参数数据却必须包含在该消息结构里以供其他应用程序访问。

当服务器应用程序在其他系统上时，不能使用 DDE或者全局内存来传递参数—因为没有办法让其他应用程序来访问它。还可以使用文件映射来包含该参数或者通过消息本身传递。如果通过通信过程传递参数，任何返回值都必须被返回客户应用程序。那就意味着服务器可能修改数组，整个数组必须被返回给客户。文件映射方式相同，但后台实现不同。总之，让大量参数来回传递可能会很慢，因此必须计划周密。

3.12.2 远程过程调用

参数经常以通信的方式在系统之间传递。实际上，一种名为远程过程调用 (Remote Procedure Calling，RPC)的Windows API可用于处理许多通过网络访问服务器应用程序这样的问题，该API负责传递消息中的参数。RPC经常是作为微软的组件对象模型 (COM)的一个组件来使用的，有关内容已经超出本书的介绍范围。

3.13 小结

本章讨论了6种使应用程序与其所处环境通信的可用方式：

- 窗口消息作为简单的进程间通信。
- DDE传送大量数据。
- 消息管道通过网络通信。
- Windows套接字通过网络与非Windows平台通信。
- Internet类通过Internet通信。
- 串行/并行通信与连接到系统串行口或者并行口的设备对话。

从本章还了解到Windows套接字因为其网络分布运算的能力而取代了 DDE和消息管道。

同时还学习了如何在应用程序之间共享数据：

- 共享内存文件，这种方式由MFC的CShareFile类创建以封装全局分配内存供C++方便地访问。
- 文件映射，这种方式允许数据共享，甚至还可以通过网络进行，但是不适用于非Windows平台。

本章包含了本书的纯文本部分，在其他部分将换一个角度来观察 MFC 和Visual C++。并且提出一些可增加到应用程序中的实际特色，采用循序渐进的方式指导读者来实现它们。由于可能包括更多解释如何实现这些特色的注解，因此这些实例可能被扩充或者增强。