

## 第8章 Winsock I/O方法

本章重点是如何在 Windows套接字应用程序中对 I/O（输入 / 输出）操作进行管理。Winsock分别提供了“套接字模式”和“套接字 I/O模型”，可对一个套接字上的 I/O行为加以控制。其中，套接字模式用于决定在随一个套接字调用时，那些 Winsock函数的行为。而另一方面，套接字模型描述了一个应用程序如何对套接字上进行的 I/O进行管理及处理。要注意的是，“套接字 I/O模型”与“套接字模式”是无关的。套接字模型的出现，正是为了解决套接字模式存在的某些限制。

Winsock提供了两种套接字模式：锁定和非锁定。本章第一部分将详细介绍这两种模式，并阐释一个应用程序如何通过它们管理 I/O。如大家在本章的后面部分所见，Winsock提供了一些有趣的I/O模型，有助于应用程序通过一种“异步”方式，一次对一个或多个套接字上进行的通信加以管理。这些模型包括 select（选择）、WSAAsyncSelect（异步选择）、WSAEventSelect（事件选择）、Overlapped I/O（重叠式I/O）以及Completion port（完成端口）等等。到本章结束时，我们打算对各种套接字模式以及 I/O模型的优缺点进行总结。同时，帮助大家判断到底哪一种最适合自己应用程序的要求。

所有Windows平台都支持套接字以锁定或非锁定方式工作。然而，并非每种平台都支持每一种 I/O模型。如表 8-1所示，在当前版本的 Windows CE中，仅提供了一个 I/O模型。Windows 98和Windows 95（取决于安装的是 Winsock 1还是Winsock 2）则支持大多数I/O模型，唯一的例外便是I/O完成端口。而到了Windows NT和最新发布的Windows 2000中，每种I/O模型都是支持的。

表8-1 操作系统对套接字 I/O模型的支持情况

平 台	select	WSAAsync Select	WSAEvent Select	Overlapped	Completion Port
Windows CE	支持	不支持	不支持	不支持	不支持
Windows 95 ( Winsock 1 )	支持	支持	不支持	不支持	不支持
Windows 95 ( Winsock 2 )	支持	支持	支持	支持	不支持
Windows 98	支持	支持	支持	支持	不支持
Windows NT	支持	支持	支持	支持	支持
Windows 2000	支持	支持	支持	支持	支持

### 8.1 套接字模式

就像我们前面提到的那样，Windows套接字在两种模式下执行 I/O操作：锁定和非锁定。在锁定模式下，在 I/O操作完成前，执行操作的 Winsock函数（比如 send和recv）会一直等候下去，不会立即返回程序（将控制权交还给程序）。而在非锁定模式下，Winsock函数无论如何都会立即返回。在 Windows CE和Windows 95（安装 Winsock 1）平台上运行的应用程序仅支持极少的I/O模型，所以我们必须采取一些适当的步骤，让锁定和非锁定套接字能够满足各种场合的要求。

### 8.1.1 锁定模式

对于处在锁定模式的套接字，我们必须多加留意，因为在一个锁定套接字上调用任何一个Winsock API函数，都会产生相同的后果——耗费或长或短的时间“等待”。大多数Winsock应用都是遵照一种“生产者 - 消费者”模型来编制的。在这种模型中，应用程序需要读取（或写入）指定数量的字节，然后以它为基础执行一些计算。程序清单 8-1展示的代码片断便是一个典型的例子。

程序清单 8-1 简单的锁定模式示例

```
SOCKET sock;  
char buff[256];  
int done = 0;  
  
...  
  
while(!done)  
{  
    nBytes = recv(sock, buff, 65);  
    if (nBytes == SOCKET_ERROR)  
    {  
        printf("recv failed with error %d\n",  
            WSAGetLastError());  
        Return;  
    }  
    DoComputationOnData(buff);  
}  
  
...
```

这段代码的问题在于，假如没有数据处于“待决”状态，那么recv函数可能永远都无法返回。这是由于从语句可以看出：只有从系统的输入缓冲区中读回点什么东西，才允许返回！有些程序员可能会在recv中使用MSG\_PEEK标志，或者调用ioctlsocket(设置FIONREAD选项)，在系统的缓冲区中，事先“偷看”是否存在足够的字节数量。然而，在不实际读入数据的前提下，仅仅“偷看”数据（如实际读入数据，便会将其从系统缓冲区中将其删除），可不是一件光彩的事情。我们认为，这是一种非常不好的编程习惯，应尽全力避免。在“偷看”的时候，对系统造成的开销是极大的，因为仅仅为了检查有多少个字节可用，便发出一个或者更多的系统调用。以后，理所当然地，还需要牵涉到进行实际recv调用，将数据从系统缓冲区内删除的开销。那么，如何避免这一情况呢？在此，我们的目标是防止由于数据的缺乏（这可能是网络出了故障，也可能是客户机出了问题），造成应用程序完全陷于“凝固”状态，同时不必连续性地检视系统网络缓冲！为达此目的，一个办法是将应用程序划分为一个读线程，以及一个计算线程。两个线程都共享同一个数据缓冲区。对这个缓冲区的访问需要受到一定的限制，这是用一个同步对象来实现的，比如一个事件或者Mutex（互斥体）。“读线程”的职责是从网络连续地读入数据，并将其置入共享缓冲区内。读线程将计算线程开始工作至少需要的数据量拿到手后，便会触发一个事件，通知计算线程：你老兄可以开始干活了！随后，计算线程从缓冲区取走（删除）一个数据块，然后进行要求的计算。

在程序清单8-2中，我们分别提供了两个函数，采取的便是上述办法。在两个函数中，一个负责读取网络数据（ReadThread），另一个则负责对数据执行计算（ProcessThread）。

程序清单8-2 多线程的锁定套接字示例

```
// Initialize critical section (data) and create
// an auto-reset event (hEvent) before creating the
// two threads
CRITICAL_SECTION data;
HANDLE hEvent;
TCHAR buff[MAX_BUFFER_SIZE];
int nbytes;
...

// Reader thread
void ReadThread(void)
{
    int nTotal = 0,
        nRead = 0,
        nLeft = 0,
        nBytes = 0;

    while (!done)
    {
        nTotal = 0;
        nLeft = NUM_BYTES_REQUIRED;
        while (nTotal != NUM_BYTES_REQUIRED)
        {
            EnterCriticalSection(&data);
            nRead = recv(sock, &(buff[MAX_BUFFER_SIZE - nBytes]),
                nLeft);
            if (nRead == -1)
            {
                printf("error\n");
                ExitThread();
            }
            nTotal += nRead;
            nLeft -= nRead;

            nBytes += nRead;
            LeaveCriticalSection(&data);
        }
        SetEvent(hEvent);
    }
}

// Computation thread
void ProcessThread(void)
{
    WaitForSingleObject(hEvent);

    EnterCriticalSection(&data);
    DoSomeComputationOnData(buff);

    // Remove the processed data from the input
```

```
// buffer, and shift the remaining data to
// the start of the array
nBytes -= NUM_BYTES_REQUIRED;

LeaveCriticalSection(&data);
}
```

对锁定套接字来说，它的一个缺点在于：应用程序很难同时通过多个建好连接的套接字通信。使用前述的办法，我们可对应用程序进行修改，令其为连好的每个套接字都分配一个读线程，以及一个数据处理线程。尽管这仍然会增大一些开销，但的确是一种可行的方案。唯一的缺点便是扩展性极差，以后想同时处理大量套接字时，恐怕难以下手。

### 8.1.2 非锁定模式

除了锁定模式，我们还可考虑采用非锁定模式的套接字。尽管这种套接字在使用上存在着些许难度，但只要排除了这项困难，它在功能上还是非常强大的。除具备锁定套接字已有的各项优点之外，还进行了少许扩充，功能更强。程序清单 8-3向大家展示了如何创建一个套接字，并将其置为非锁定模式。

程序清单8-3 设置一个非锁定套接字

```
SOCKET      s;
unsigned long ul = 1;
int         nRet;

s = socket(AF_INET, SOCK_STREAM, 0);
nRet = ioctlsocket(s, FIOBIO, (unsigned long *) &ul);
if (nRet == SOCKET_ERROR)
{
    // Failed to put the socket into nonblocking mode
}
```

将一个套接字置为非锁定模式之后，Winsock API调用会立即返回。大多数情况下，这些调用都会“失败”，并返回一个WSAEWOULDBLOCK错误。什么意思呢？它意味着请求的操作在调用期间没有时间完成。举个例子来说，假如在系统的输入缓冲区中，尚不存在“待决”的数据，那么recv（接收数据）调用就会返回WSAEWOULDBLOCK错误。通常，我们需要重复调用同一个函数，直至获得一个成功返回代码。在表 8-2中，我们对常见Winsock调用返回的WSAEWOULDBLOCK错误的含义进行了总结。

由于非锁定调用会频繁返回WSAEWOULDBLOCK错误，所以在任何时候，都应仔细检查所有返回代码，并作好“失败”的准备。许多程序员易犯的一个错误便是连续不停地调用一个函数，直到它返回成功的消息为止。例如，假定在一个紧凑的循环中不断地调用recv，以读入200个字节的数据，那么与使用前述的MSG\_PEEK标志来“轮询”一个锁定套接字相比，前一种做法根本没有任何优势可言。为此，Winsock的套接字I/O模型可帮助应用程序判断一个套接字何时可供读写。

锁定和非锁定套接字模式都存在着优点和缺点。其中，从概念的角度说，锁定套接字更易使用。但在应付建立连接的多个套接字时，或在数据的收发量不均，时间不定时，却显得极难管理。而另一方面，假如需要编写更多的代码，以便在每个Winsock调用中，对收到一个

WSAEWOULDBLOCK错误的可能性加以应付，那么非锁定套接字便显得有些难于操作。在这些情况下，可考虑使用“套接字 I/O模型”，它有助于应用程序通过一种异步方式，同时对一个或多个套接字上进行的通信加以管理。

表8-2 非锁定套接字上的WSAEWOULDBLOCK错误

函 数 名	说 明
WSAAccept和accept closesocket	应用程序没有收到连接请求。再次调用，便可检查连接情况 大多数情况下，这个错误意味着已随SO_LINGER选项一道， 调用了setsockopt，而且已设定了一个非零的超时值
WSAConnect和connect	应用程序已初始化。再次调用，便可检查是否完成
WSARecv、recv、WSARecvFrom和recvfrom	没有收到数据。稍后再次检查
WSASend、send、WSASendTo和sendto	外出数据无缓冲区可用。稍后再试

## 8.2 套接字I/O模型

共有五种类型的套接字 I/O模型，可让 Winsock 应用程序对 I/O 进行管理，它们包括：select（选择）、WSAAsyncSelect（异步选择）、WSAEventSelect（事件选择）、overlapped（重叠）以及completion port（完成端口）。在这一节里，我们打算向大家解释每种 I/O模型的特点，同时讲述如何利用这些模型，来开发自己的应用程序，以便同时管理一个或多个套接字请求。在本书的配套光盘上，针对每种 I/O模型，大家都可以找到一个或者更多的示范应用程序。这些程序阐述了如何利用每种模型的原理，开发一个简单的 TCP 回应服务器。

### 8.2.1 select模型

select（选择）模型是 Winsock 中最常见的 I/O 模型。之所以称其为“select 模型”，是由于它的“中心思想”便是利用 select 函数，实现对 I/O 的管理！最初设计该模型时，主要面向的是某些使用 Unix 操作系统的计算机，它们采用的是 Berkeley 套接字方案。select 模型已集成到 Winsock 1.1 中，它使那些想避免在套接字调用过程中被无辜“锁定”的应用程序，采取一种有序的方式，同时进行对多个套接字的管理。由于 Winsock 1.1 向后兼容于 Berkeley 套接字实施方案，所以假如有一个 Berkeley 套接字应用使用了 select 函数，那么从理论角度讲，毋需对其进行任何修改，便可正常运行。

利用 select 函数，我们判断套接字上是否存在数据，或者能否向一个套接字写入数据。之所以要设计这个函数，唯一的目的是防止应用程序在套接字处于锁定模式中时，在一次 I/O 绑定调用（如 send 或 recv）过程中，被迫进入“锁定”状态；同时防止在套接字处于非锁定模式中时，产生 WSAEWOULDBLOCK 错误。除非满足事先用参数规定的条件，否则 select 函数会在进行 I/O 操作时锁定。select 的函数原型如下：

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

其中，第一个参数 nfd 会被忽略。之所以仍然要提供这个参数，只是为了保持与早期的

Berkeley套接字应用程序的兼容。大家可注意到三个 `fd_set` 参数：一个用于检查可读性（`readfds`），一个用于检查可写性（`writfds`），另一个用于例外数据（`exceptfds`）。从根本上说，`fd_set`数据类型代表着一系列特定套接字的集合。其中，`readfds`集合包括符合下述任何一个条件的套接字：

有数据可以读入。

连接已经关闭、重设或中止。

假如已调用了`listen`，而且一个连接正在建立，那么`accept`函数调用会成功。

`writfds`集合包括符合下述任何一个条件的套接字：

有数据可以发出。

如果已完成了对一个非锁定连接调用的处理，连接就会成功。

最后，`exceptfds`集合包括符合下述任何一个条件的套接字：

假如已完成了对一个非锁定连接调用的处理，连接尝试就会失败。

有带外（Out-of-band，OOB）数据可供读取。

例如，假定我们想测试一个套接字是否“可读”，必须将自己的套接字增添到`readfds`集合，再等待`select`函数完成。`select`完成之后，必须判断自己的套接字是否仍为`readfds`集合的一部分。若答案是肯定的，便表明该套接字“可读”，可立即着手从它上面读取数据。在三个参数中（`readfds`、`writfds`和`exceptfds`），任何两个都可以是空值（`NULL`）；但是，至少有一个不能为空值！在任何不为空的集合中，必须包含至少一个套接字句柄；否则，`select`函数便没有任何东西可以等待。最后一个参数`timeout`对应的是一个指针，它指向一个`timeval`结构，用于决定`select`最多等待I/O操作完成多久的时间。如`timeout`是一个空指针，那么`select`调用会无限期地“锁定”或停顿下去，直到至少有一个描述符符合指定的条件后结束。对`timeval`结构的定义如下：

```
struct timeval
{
    long tv_sec;
    long tv_usec;
};
```

其中，`tv_sec`字段以秒为单位指定等待时间；`tv_usec`字段则以毫秒为单位指定等待时间。若将超时值设置为（0,0），表明`select`会立即返回，允许应用程序对`select`操作进行“轮询”。出于对性能方面的考虑，应避免这样的设置。`select`成功完成后，会在`fd_set`结构中，返回刚好有未完成的I/O操作的所有套接字句柄的总量。若超过`timeval`设定的时间，便会返回0。不管由于什么原因，假如`select`调用失败，都会返回`SOCKET_ERROR`。

用`select`对套接字进行监视之前，在自己的应用程序中，必须将套接字句柄分配给一个集合，设置好一个或全部读、写以及例外`fd_set`结构。将一个套接字分配给任何一个集合后，再来调用`select`，便可知道一个套接字上是否正在发生上述的I/O活动。Winsock提供了下列宏操作，可用于针对I/O活动，对`fd_set`进行处理与检查：

`FD_CLR(s, *set)`：从`set`中删除套接字`s`。

`FD_ISSET(s, *set)`：检查`s`是否`set`集合的一名成员；如答案是肯定的是，则返回`TRUE`。

`FD_SET(s, *set)`：将套接字`s`加入集合`set`。

`FD_ZERO(*set)`：将`set`初始化成空集合。



例如，假定我们想知道是否可从一个套接字中安全地读取数据，同时不会陷于无休止的“锁定”状态，便可使用FD\_SET宏，将自己的套接字分配给fd\_read集合，再来调用select。要想检测自己的套接字是否仍属fd\_read集合的一部分，可使用FD\_ISSET宏。采用下述步骤，便可完成用select操作一个或多个套接字句柄的全过程：

- 1) 使用FD\_ZERO宏，初始化自己感兴趣的每一个fd\_set。
- 2) 使用FD\_SET宏，将套接字句柄分配给自己感兴趣的每个fd\_set。
- 3) 调用select函数，然后等待在指定的fd\_set集合中，I/O活动设置好一个或多个套接字句柄。select完成后，会返回在所有fd\_set集合中设置的套接字句柄总数，并对每个集合进行相应的更新。
- 4) 根据select的返回值，我们的应用程序便可判断出哪些套接字存在着尚未完成（待决）的I/O操作——具体的方法是使用FD\_ISSET宏，对每个fd\_set集合进行检查。
- 5) 知道了每个集合中“待决”的I/O操作之后，对I/O进行处理，然后返回步骤1)，继续进行select处理。

select返回后，它会修改每个fd\_set结构，删除那些不存在待决I/O操作的套接字句柄。这正是我们在上述的步骤(4)中，为何要使用FD\_ISSET宏来判断一个特定的套接字是否仍在集合中的原因。在程序清单8-4中，我们向大家阐述了为一个（只有一个）套接字设置select模型所需的一系列基本步骤。若想在这个应用程序中添加更多的套接字，只需为额外的套接字维护它们的一个列表，或维护它们的一个数组即可。

程序清单8-4 用select管理一个套接字上的I/O操作

```
SOCKET s;
fd_set fdread;
int ret;

// Create a socket, and accept a connection
_

// Manage I/O on the socket
while(TRUE)
{
    // Always clear the read set before calling
    // select()
    FD_ZERO(&fdread);

    // Add socket s to the read set
    FD_SET(s, &fdread);

    if ((ret = select(0, &fdread, NULL, NULL, NULL))
        == SOCKET_ERROR)
    {
        // Error condition
    }

    if (ret > 0)
    {
        // For this simple case, select() should return
        // the value 1. An application dealing with
        // more than one socket could get a value
        // greater than 1. At this point, your
        // application should check to see whether the
```

```
// socket is part of a set.

if (FD_ISSET(s, &fdread))
{
    // A read event has occurred on socket s
}
}
```

### 8.2.2 WSAAsyncSelect

Winsock提供了一个有用的异步 I/O模型。利用这个模型，应用程序可在一个套接字上，接收以 Windows消息为基础的网络事件通知。具体的做法是在建好一个套接字后，调用 WSAAsyncSelect函数。该模型最早出现于 Winsock的1.1版本中，用于帮助应用程序开发者面向一些早期的16位Windows平台（如Windows for Workgroups），适应其“落后”的多任务消息环境。应用程序仍可从这种模型中得到好处，特别是它们用一个标准的 Windows例程（常称为“winproc”），对窗口消息进行管理的时候。该模型亦得到了 Microsoft Foundation Class（微软基本类，MFC）对象CSocket的采纳。

#### 消息通知

要想使用 WSAAsyncSelect模型，在应用程序中，首先必须用 CreateWindow函数创建一个窗口，再为该窗口提供一个窗口例程支持函数（Winproc）。亦可使用一个对话框，为其提供一个对话例程，而非窗口例程，因为对话框本质也是“窗口”。考虑到我们的目的，我们打算用一个简单的窗口来演示这种模型，采用的是一个支持窗口例程。设置好窗口的框架后，便可开始创建套接字，并调用 WSAAsyncSelect函数，打开窗口消息通知。该函数的定义如下：

```
int WSAAsyncSelect(
    SOCKET s,
    HWND hWnd,
    unsigned int wMsg,
    long lEvent
);
```

其中，s参数指定的是我们感兴趣的那个套接字。hWnd参数指定的是一个窗口句柄，它对应于网络事件发生之后，想要收到通知消息的那个窗口或对话框。wMsg参数指定在发生网络事件时，打算接收的消息。该消息会投递到由 hWnd窗口句柄指定的那个窗口。通常，应用程序需要将这个消息设为比 Windows的WM\_USER大的一个值，避免网络窗口消息与预定义的标准窗口消息发生混淆与冲突。最后一个参数是 lEvent，它指定的是一个位掩码，对应于一系列网络事件的组合（请参考表 8-3），应用程序感兴趣的便是这一系列事件。大多数应用程序通常感兴趣的网络事件类型包括：FD\_READ、FD\_WRITE、FD\_ACCEPT、FD\_CONNECT和 FD\_CLOSE。当然，到底使用 FD\_ACCEPT，还是使用 FD\_CONNECT类型，要取决于应用程序的身份到底是一个客户机呢，还是一个服务器。如应用程序同时对多个网络事件有兴趣，只需对各种类型执行一次简单的按位 OR（或）运算，然后将它们分配给 lEvent就可以了。举个例子来说：

```
WSAAsyncSelect(s, hWnd, WM_SOCKET,
    FD_CONNECT | FD_READ | FD_WRITE | FD_CLOSE);
```



这样一来，我们的应用程序以后便可在套接字 `s` 上，接收到有关连接、发送、接收以及套接字关闭这一系列网络事件的通知。特别要注意的是，多个事件务必在套接字上一次注册！另外还要注意的，一旦在某个套接字上允许了事件通知，那么以后除非明确调用 `closesocket` 命令，或者由应用程序针对那个套接字调用了 `WSAAsyncSelect`，从而更改了注册的网络事件类型，否则的话，事件通知会永远有效！若将 `lEvent` 参数设为 0，效果相当于停止在套接字上进行的所有网络事件通知。

若应用程序针对一个套接字调用了 `WSAAsyncSelect`，那么套接字的模式会从“锁定”自动变成“非锁定”，我们在前面已提到过这一点。这样一来，假如调用了像 `WSARecv` 这样的 Winsock I/O 函数，但当时却并没有数据可用，那么必然会造成调用的失败，并返回 `WSAEWOULDBLOCK` 错误。为防止这一点，应用程序应依赖于由 `WSAAsyncSelect` 的 `uMsg` 参数指定的用户自定义窗口消息，来判断网络事件类型何时在套接字上发生；而不应盲目地进行调用。

表8-3 用于 `WSAAsyncSelect` 函数的网络事件类型

事件类型	含 义
<code>FD_READ</code>	应用程序想要接收有关是否可读的通知，以便读入数据
<code>FD_WRITE</code>	应用程序想要接收有关是否可写的通知，以便写入数据
<code>FD_OOB</code>	应用程序想接收是否有带外（OOB）数据抵达的通知
<code>FD_ACCEPT</code>	应用程序想接收与进入连接有关的通知
<code>FD_CONNECT</code>	应用程序想接收与一次连接或者多点 <code>join</code> 操作完成的通知
<code>FD_CLOSE</code>	应用程序想接收与套接字关闭有关的通知
<code>FD_QOS</code>	应用程序想接收套接字“服务质量”（QoS）发生更改的通知
<code>FD_GROUP_QOS</code>	应用程序想接收套接字组“服务质量”发生更改的通知（现在没什么用处，为未来套接字组的使用保留）
<code>FD_ROUTING_INTERFACE_CHANGE</code>	应用程序想接收在指定的方向上，与路由接口发生变化的通知
<code>FD_ADDRESS_LIST_CHANGE</code>	应用程序想接收针对套接字的协议家族，本地地址列表发生变化的通知

应用程序在一个套接字上成功调用了 `WSAAsyncSelect` 之后，应用程序会在与 `hWnd` 窗口句柄参数对应的窗口例程中，以 Windows 消息的形式，接收网络事件通知。窗口例程通常定义如下：

```
LRESULT CALLBACK WindowProc(  
    HWND hWnd,  
    UINT uMsg,  
    WPARAM wParam,  
    LPARAM lParam  
);
```

其中，`hWnd` 参数指定一个窗口的句柄，对窗口例程的调用正是由那个窗口发出的。`uMsg` 参数指定需要对哪些消息进行处理。就我们的情况来说，感兴趣的是 `WSAAsyncSelect` 调用中定义的消息。`wParam` 参数指定在其上面发生了一个网络事件的套接字。假若同时为这个窗口例程分配了多个套接字，这个参数的重要性便显示出来了。在 `lParam` 参数中，包含了两方面重要的信息。其中，`lParam` 的低字（低位字）指定了已经发生的网络事件，而 `lParam` 的高字（高位字）包含了可能出现的任何错误代码。

网络事件消息抵达一个窗口例程后，应用程序首先应检查 `lParam` 的高字位，以判断是否在套接字上发生了一个网络错误。这里有一个特殊的宏：`WSAGETSELECTERROR`，可用它返回

高字位包含的错误信息。若应用程序发现套接字上没有产生任何错误，接着便应调查到底是哪个网络事件类型，造成了这条 Windows 消息的触发——具体的做法便是读取 lParam 之低字位的内容。此时可使用另一个特殊的宏：WSAGETSELECTEVENT，用它返回 lParam 的低字部分。

在程序清单 8-5 中，我们向大家演示了如何使用 WSAAsyncSelect 这种 I/O 模型，来实现窗口消息的管理。在源程序中，我们着重强调的是开发一个基本服务器应用要涉及到的基本步骤，忽略了开发一个完整的 Windows 应用需要涉及到的大量编程细节。

程序清单 8-5 WSAAsyncSelect 服务器示范代码

```
#define WM_SOCKET WM_USER + 1
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance,
    HINSTANCE hPrevInstance, LPSTR lpCmdLine,
    int nCmdShow)
{
    SOCKET Listen;
    HWND Window;

    // Create a window and assign the ServerWinProc
    // below to it

    Window = CreateWindow();
    // Start Winsock and create a socket

    WSASStartup(...);
    Listen = Socket();

    // Bind the socket to port 5150
    // and begin listening for connections

    InternetAddr.sin_family = AF_INET;
    InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
    InternetAddr.sin_port = htons(5150);

    bind(Listen, (PSOCKADDR) &InternetAddr,
        sizeof(InternetAddr));

    // Set up window message notification on
    // the new socket using the WM_SOCKET define
    // above

    WSAAsyncSelect(Listen, Window, WM_SOCKET,
        FD_ACCEPT | FD_CLOSE);

    listen(Listen, 5);

    // Translate and dispatch window messages
    // until the application terminates
}
```

```
BOOL CALLBACK ServerWinProc(HWND hDlg, WORD wMsg,
    WORD wParam, DWORD lParam)
{
    SOCKET Accept;

    switch(wMsg)
    {
        case WM_PAINT:
            // Process window paint messages
            break;

        case WM_SOCKET:

            // Determine whether an error occurred on the
            // socket by using the WSAGETSELECTERROR() macro

            if (WSAGETSELECTERROR(lParam))
            {
                // Display the error and close the socket
                closesocket(wParam);
                break;
            }

            // Determine what event occurred on the
            // socket

            switch(WSAGETSELECTEVENT(lParam))
            {
                case FD_ACCEPT:

                    // Accept an incoming connection
                    Accept = accept(wParam, NULL, NULL);

                    // Prepare accepted socket for read,
                    // write, and close notification

                    WSAAsyncSelect(Accept, hwnd, WM_SOCKET,
                        FD_READ | FD_WRITE | FD_CLOSE);
                    break;

                case FD_READ:

                    // Receive data from the socket in
                    // wParam
                    break;

                case FD_WRITE:

                    // The socket in wParam is ready
                    // for sending data
```

```
        break;

    case FD_CLOSE:

        // The connection is now closed
        closesocket(wParam);
        break;

    }
    break;
}
return TRUE;
}
```

最后一个特别有价值的问题是应用程序如何对 FD\_WRITE事件通知进行处理。只有在三种条件下,才会发出FD\_WRITE通知:

使用connect或WSAConnect,一个套接字首次建立了连接。

使用accept或WSAAccept,套接字被接受以后。

若send、WSASend、sendto或WSASendTo操作失败,返回了WSAEWOULDBLOCK错误,而且缓冲区的空间变得可用

因此,作为一个应用程序,自收到首条 FD\_WRITE消息开始,便应认为自己必然能在一个套接字上发出数据,直至一个 send、WSASend、sendto或WSASendTo返回套接字错误WSAEWOULDBLOCK。经过了这样的失败以后,要再用另一条 FD\_WRITE通知应用程序再次发送数据。

### 8.2.3 WSAEventSelect

Winsock提供了另一个有用的异步 I/O模型。和WSAAsyncSelect模型类似的是,它也允许应用程序在一个或多个套接字上,接收以事件为基础的网络事件通知。对于表 8-3总结的、由WSAAsyncSelect模型采用的网络事件来说,它们均可原封不动地移植到新模型。在用新模型开发的应用程序中,也能接收和处理所有那些事件。该模型最主要的差别在于网络事件会投递至一个事件对象句柄,而非投递至一个窗口例程。

#### 事件通知

事件通知模型要求我们的应用程序针对打算使用的每一个套接字,首先创建一个事件对象。创建方法是调用WSACreateEvent函数,它的定义如下:

```
WSAEVENT WSACreateEvent(void);
```

WSACreateEvent函数的返回值很简单,就是一个创建好的事件对象句柄。事件对象句柄到手后,接下来必须将其与某个套接字关联在一起,同时注册自己感兴趣的网络事件类型,如表8-3所示。要做到这一点,方法是调用WSAEventSelect函数,对它的定义如下:

```
int WSAEventSelect(
    SOCKET s,
    WSAEVENT hEventObject,
    long lNetworkEvents
);
```

其中,s参数代表自己感兴趣的套接字。hEventObject参数指定要与套接字关联在一起的

事件对象——用WSACreateEvent取得的那一个。而最后一个参数 INetworkEvents，则对应一个“位掩码”，用于指定应用程序感兴趣的各种网络事件类型的一个组合（如表 8-3所示）。要想获知对这些事件类型的详细说明，请参考早先讨论过的 WSAAsyncSelect I/O模型。

为WSAEventSelect创建的事件拥有两种工作状态，以及两种工作模式。其中，两种工作状态分别是“已传信”(signaled)和“未传信”(nonsignaled)。工作模式则包括“人工重设”(manual reset)和“自动重设”(auto reset)。WSACreateEvent最开始在一种未传信的工作状态中，并用一种人工重设模式，来创建事件句柄。随着网络事件触发了与一个套接字关联在一起的事件对象，工作状态便会从“未传信”转变成“已传信”。由于事件对象是在一种人工重设模式中创建的，所以在完成了一个 I/O请求的处理之后，我们的应用程序需要负责将工作状态从已传信更改为未传信。要做到这一点，可调用 WSAResetEvent函数，对它的定义如下：

```
BOOL WSAResetEvent(WSAEVENT hEvent);
```

该函数唯一的参数便是一个事件句柄；基于调用是成功还是失败，会分别返回 TRUE或FALSE。应用程序完成了对一个事件对象的处理后，便应调用 WSACloseEvent函数，释放由事件句柄使用的系统资源。对 WSACloseEvent函数的定义如下：

```
BOOL WSACloseEvent(WSAEVENT hEvent);
```

该函数也要拿一个事件句柄作为自己唯一的参数，并会在成功后返回 TRUE，失败后返回 FALSE。

一个套接字同一个事件对象句柄关联在一起后，应用程序便可开始 I/O处理；方法是等待网络事件触发事件对象句柄的工作状态。WSAWaitForMultipleEvents函数的设计宗旨便是用来等待一个或多个事件对象句柄，并在事先指定的一个或所有句柄进入“已传信”状态后，或在超过了一个规定的时间周期后，立即返回。下面是 WSAWaitForMultipleEvents函数的定义：

```
DWORD WSAWaitForMultipleEvents(  
    DWORD cEvents,  
    const WSAEVENT FAR * lphEvents,  
    BOOL fWaitAll,  
    DWORD dwTimeout,  
    BOOL fAlertable  
);
```

其中，cEvents和lphEvents参数定义了由WSAEVENT对象构成的一个数组。在这个数组中，cEvents指定的是事件对象的数量，而lphEvents对应的是一个指针，用于直接引用该数组。要注意的是，WSAWaitForMultipleEvents只能支持由WSA\_MAXIMUM\_WAIT\_EVENTS对象规定的一个最大值，在此定义成 64个。因此，针对发出 WSAWaitForMultipleEvents调用的每个线程，该I/O模型一次最多都只能支持 64个套接字。假如想让这个模型同时管理不止 64个套接字，必须创建额外的工作者线程，以便等待更多的事件对象。fWaitAll参数指定了 WSAWaitForMultipleEvents如何等待在事件数组中的对象。若设为 TRUE，那么只有等lphEvents数组内包含的所有事件对象都已进入“已传信”状态，函数才会返回；但若设为 FALSE，任何一个事件对象进入“已传信”状态，函数就会返回。就后一种情况来说，返回值指出了到底是哪个事件对象造成了函数的返回。通常，应用程序应将该参数设为 FALSE，一次只为一个套接字事件提供服务。dwTimeout参数规定了WSAWaitForMultipleEvents最多可

等待一个网络事件发生有多长时间，以毫秒为单位，这是一项“超时”设定。超过规定的时间，函数就会立即返回，即使由 `fWaitAll` 参数规定的条件尚未满足也如此。如超时值为 0，函数会检测指定的事件对象的状态，并立即返回。这样一来，应用程序实际便可实现对事件对象的“轮询”。但考虑到它对性能造成的影响，还是应尽量避免将超时值设为 0。假如没有等待处理的事件，`WSAWaitForMultipleEvents` 便会返回 `WSA_WAIT_TIMEOUT`。如 `dwsTimeout` 设为 `WSA_INFINITE`（永远等待），那么只有在一个网络事件传信了一个事件对象后，函数才会返回。最后一个参数是 `fAlertable`，在我们使用 `WSAEventSelect` 模型的时候，它是可以忽略的，且应设为 `FALSE`。该参数主要用于在重叠式 I/O 模型中，在完成例程的处理过程中使用。本章后面还会对此详述。

若 `WSAWaitForMultipleEvents` 收到一个事件对象的网络事件通知，便会返回一个值，指出造成函数返回的事件对象。这样一来，我们的应用程序便可引用事件数组中已传信的事件，并检索与那个事件对应的套接字，判断到底是在哪个套接字上，发生了什么网络事件类型。对事件数组中的事件进行引用时，应该用 `WSAWaitForMultipleEvents` 的返回值，减去预定义值 `WSA_WAIT_EVENT_0`，得到具体的引用值（即索引位置）。如下例所示：

```
Index = WSAWaitForMultipleEvents(...);  
MyEvent = EventArray[Index - WSA_WAIT_EVENT_0];
```

知道了造成网络事件的套接字后，接下来可调用 `WSAEnumNetworkEvents` 函数，调查发生了什么类型的网络事件。该函数定义如下：

```
int WSAEnumNetworkEvents(  
    SOCKET s,  
    WSAEVENT hEventObject,  
    LPWSANETWORKEVENTS lpNetworkEvents  
);
```

`s` 参数对应于造成了网络事件的套接字。`hEventObject` 参数则是可选的；它指定了一个事件句柄，对应于打算重设的那个事件对象。由于我们的事件对象处在一个“已传信”状态，所以可将它传入，令其自动成为“未传信”状态。如果不想用 `hEventObject` 参数来重设事件，那么可使用 `WSAResetEvent` 函数，该函数早先已经讨论过了。最后一个参数是 `lpNetworkEvents`，代表一个指针，指向 `WSANETWORKEVENTS` 结构，用于接收套接字上发生的网络事件类型以及可能出现的任何错误代码。下面是 `WSANETWORKEVENTS` 结构的定义：

```
typedef struct _WSANETWORKEVENTS  
{  
    long lNetworkEvents;  
    int iErrorCode[FD_MAX_EVENTS];  
} WSANETWORKEVENTS, FAR * LPWSANETWORKEVENTS;
```

`lNetworkEvents` 参数指定了一个值，对应于套接字上发生的所有网络事件类型（参见表 8-3）。

注意 一个事件进入传信状态时，可能会同时发生多个网络事件类型。例如，一个繁忙的服务器应用可能同时收到 `FD_READ` 和 `FD_WRITE` 通知。

`iErrorCode` 参数指定的是一个错误代码数组，同 `lNetworkEvents` 中的事件关联在一起。针对每个网络事件类型，都存在着一个特殊的事件索引，名字与事件类型的名字类似，只是要在事件名字后面添加一个“\_BIT”后缀字符串即可。例如，对 `FD_READ` 事件类型来说，



iErrorCode数组的索引标识符便是 FD\_READ\_BIT。下述代码片断对此进行了阐释（针对 FD\_READ事件）：

```
// Process FD_READ notification
if (NetworkEvents.lNetworkEvents & FD_READ)
{
    if (NetworkEvents.iErrorCode[FD_READ_BIT] != 0)
    {
        printf("FD_READ failed with error %d\n",
            NetworkEvents.iErrorCode[FD_READ_BIT]);
    }
}
```

完成了对WSANETWORKEVENTS结构中的事件的处理之后，我们的应用程序应在所有可用的套接字上，继续等待更多的网络事件。在程序清单 8-6中，我们阐释了如何使用 WSAEventSelect这种I/O模型，来开发一个服务器应用，同时对事件对象进行管理。这个程序主要着眼于开发一个基本的服务器应用要涉及到的步骤，令其同时负责一个或多个套接字的管理。

程序清单8-6 采用WSAEventSelect I/O模型的示范服务器源代码

---

```
SOCKET Socket[WSA_MAXIMUM_WAIT_EVENTS];
WSAEVENT Event[WSA_MAXIMUM_WAIT_EVENTS];
SOCKET Accept, Listen;
DWORD EventTotal = 0;
DWORD Index;
// Set up a TCP socket for listening on port 5150
Listen = socket (PF_INET, SOCK_STREAM, 0);

InternetAddr.sin_family = AF_INET;
InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
InternetAddr.sin_port = htons(5150);

bind(Listen, (PSOCKADDR) &InternetAddr,
    sizeof(InternetAddr));

NewEvent = WSACreateEvent();

WSAEventSelect(Listen, NewEvent,
    FD_ACCEPT | FD_CLOSE);

listen(Listen, 5);

Socket[EventTotal] = Listen;
Event[EventTotal] = NewEvent;
EventTotal++;

while(TRUE)
{
    // Wait for network events on all sockets
    Index = WSAWaitForMultipleEvents(EventTotal,
        EventArray, FALSE, WSA_INFINITE, FALSE);

    WSAEnumNetworkEvents(
        SocketArray[Index - WSA_WAIT_EVENT_0],
        EventArray[Index - WSA_WAIT_EVENT_0].
```

```

    &NetworkEvents);

// Check for FD_ACCEPT messages
if (NetworkEvents.lNetworkEvents & FD_ACCEPT)
{
    if (NetworkEvents.iErrorCode[FD_ACCEPT_BIT] != 0)
    {
        printf("FD_ACCEPT failed with error %d\n",
            NetworkEvents.iErrorCode[FD_ACCEPT_BIT]);
        break;
    }

    // Accept a new connection, and add it to the
    // socket and event lists
    Accept = accept(
        SocketArray[Index - WSA_WAIT_EVENT_0],
        NULL, NULL);
    // We cannot process more than
    // WSA_MAXIMUM_WAIT_EVENTS sockets, so close
    // the accepted socket
    if (EventTotal > WSA_MAXIMUM_WAIT_EVENTS)
    {
        printf("Too many connections");
        closesocket(Accept);
        break;
    }

    NewEvent = WSACreateEvent();

    WSAEventSelect(Accept, NewEvent,
        FD_READ | FD_WRITE | FD_CLOSE);

    Event[EventTotal] = NewEvent;
    Socket[EventTotal] = Accept;
    EventTotal++;

    printf("Socket %d connected\n", Accept);
}

// Process FD_READ notification
if (NetworkEvents.lNetworkEvents & FD_READ)
{
    if (NetworkEvents.iErrorCode[FD_READ_BIT] != 0)
    {
        printf("FD_READ failed with error %d\n",
            NetworkEvents.iErrorCode[FD_READ_BIT]);
        break;
    }

    // Read data from the socket
    recv(Socket[Index - WSA_WAIT_EVENT_0],
        buffer, sizeof(buffer), 0);
}

// Process FD_WRITE notification

```

```
if (NetworkEvents.lNetworkEvents & FD_WRITE)
{
    if (NetworkEvents.iErrorCode[FD_WRITE_BIT] != 0)
    {
        printf("FD_WRITE failed with error %d\n",
            NetworkEvents.iErrorCode[FD_WRITE_BIT]);
        break;
    }

    send(Socket[Index - WSA_WAIT_EVENT_0],
        buffer, sizeof(buffer), 0);
}

if (NetworkEvents.lNetworkEvents & FD_CLOSE)
{
    if (NetworkEvents.iErrorCode[FD_CLOSE_BIT] != 0)
    {
        printf("FD_CLOSE failed with error %d\n",
            NetworkEvents.iErrorCode[FD_CLOSE_BIT]);
        break;
    }

    closesocket(Socket[Index - WSA_WAIT_EVENT_0]);

    // Remove socket and associated event from
    // the Socket and Event arrays and decrement
    // EventTotal
    CompressArrays(Event, Socket, &EventTotal);
}
}
```

#### 8.2.4 重叠模型

在Winsock中，相比我们迄今为止解释过的其他所有 I/O模型，重叠I/O（Overlapped I/O）模型使应用程序能达到更佳的系统性能。重叠模型的基本设计原理便是让应用程序使用一个重叠的数据结构，一次投递一个或多个 Winsock I/O请求。针对那些提交的请求，在它们完成之后，应用程序可为它们提供服务。该模型适用于除 Windows CE之外的各种Windows平台。模型的总体设计以 Win32重叠I/O机制为基础。那个机制可通过 ReadFile和WriteFile两个函数，针对设备执行I/O操作。

最开始的时候，Winsock重叠I/O模型只能应用于 Windows NT操作系统上运行的 Winsock 1.1应用程序。作为应用程序，它可针对一个套接字句柄，调用 ReadFile以及WriteFile，同时指定一个重叠式结构（稍后详述），从而利用这个模型。自 Winsock 2发布开始，重叠I/O便已集成到新的Winsock函数中，比如WSASend和WSARecv。这样一来，重叠I/O模型便能适用于安装了Winsock 2的所有Windows平台。

注意 在Winsock 2发布之后，重叠I/O仍可在Windows NT和Windows 2000这两个操作系统中，随ReadFile和WriteFile两个函数使用。但是，Windows 95和Windows 98均不具备这一功能。考虑到应用程序的跨平台兼容问题，同时考虑到性能方面的因素，应尽量避免使用Win32的ReadFile和WriteFile函数，分别换以WSARecv和WSASend函数。本节

只打算讲述通过新的Winsock 2函数，来使用重叠I/O模型。

要想在一个套接字上使用重叠 I/O模型，首先必须使用 WSA\_FLAG\_OVERLAPPED这个标志，创建一个套接字。如下所示：

```
s = WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0,  
WSA_FLAG_OVERLAPPED);
```

创建套接字的时候，假如使用的是 socket函数，而非 WSASocket函数，那么会默认设置 WSA\_FLAG\_OVERLAPPED标志。成功建好一个套接字，同时将其与一个本地接口绑定到一起后，便可开始进行重叠 I/O操作，方法是调用下述的 Winsock函数，同时指定一个 WSAOVERLAPPED结构（可选）：

```
WSASend  
WSASendTo  
WSARecv  
WSARecvFrom  
WSAIoctl  
AcceptEx  
TrnasmitFile
```

大家现在或许已经知道，其中每个函数都与一个套接字上数据的发送、数据接收以及连接的接受有关。因此，这些活动可能会花极少的时间才能完成。这正是每个函数都可接受一个WSAOVERLAPPED结构作为参数的原因。若随一个WSAOVERLAPPED结构一起调用这些函数，函数会立即完成并返回，无论套接字是否设为锁定模式（本章开头已详细讲过了）。它们依赖于WSAOVERLAPPED结构来返回一个I/O请求的返回。主要有两个方法可用来管理一个重叠I/O请求的完成：我们的应用程序可等待“事件对象通知”，亦可通过“完成例程”，对已经完成的请求加以处理。上面列出的函数（AcceptEx除外）还有另一个常用的参数：lpCompletionROUTINE。该参数指定的是一个可选的指针，指向一个完成例程函数，在重叠请求完成后调用。接下去，我们将探讨事件通知方法。在本章稍后，还会介绍如何使用可选的完成例程，代替事件，对完成的重叠请求加以处理。

### 1. 事件通知

重叠I/O的事件通知方法要求将Win32事件对象与WSAOVERLAPPED结构关联在一起。若使用一个WSAOVERLAPPED结构，发出像WSASend和WSARecv这样的I/O调用，它们会立即返回。

通常，大家会发现这些 I/O调用会以失败告终，返回 SOCKET\_ERROR。使用 WSAGetLastError函数，便可获得与错误状态有关的一个报告。这个错误状态意味着 I/O操作正在进行。稍后的某个时间，我们的应用程序需要等候与 WSAOVERLAPPED结构对应的事件对象，了解一个重叠I/O请求何时完成。WSAOVERLAPPED结构在一个重叠I/O请求的初始化，及其后续的完成之间，提供了一种沟通或通信机制。下面是这个结构的定义：

```
typedef struct WSAOVERLAPPED  
{  
    DWORD    Internal;  
    DWORD    InternalHigh;  
    DWORD    Offset;  
    DWORD    OffsetHigh;  
    WSAEVENT hEvent;
```

```
} WSAOVERLAPPED, FAR * LPWSAOVERLAPPED;
```

其中，Internal、InternalHigh、Offset和OffsetHigh字段均由系统在内部使用，不应由应用程序直接进行处理或使用。而另一方面，hEvent字段有点儿特殊，它允许应用程序将一个事件对象句柄同一个套接字关联起来。大家可能会觉得奇怪，如何将一个事件对象句柄分配给该字段呢？正如我们早先在WSAEventSelect模型中讲述的那样，可用WSACreateEvent函数来创建一个事件对象句柄。一旦创建好一个事件句柄，简单地将重叠结构的hEvent字段分配给事件句柄，再使用重叠结构，调用一个Winsock函数即可，比如WSASend或WSARecv。

一个重叠I/O请求最终完成后，我们的应用程序要负责取回重叠I/O操作的结果。一个重叠请求操作最终完成之后，在事件通知方法中，Winsock会更改与一个WSAOVERLAPPED结构对应的一个事件对象的事件传信状态，将其从“未传信”变成“已传信”。由于一个事件对象已分配给WSAOVERLAPPED结构，所以只需简单地调用WSAWaitForMultipleEvents函数，从而判断出一个重叠I/O调用在什么时候完成。该函数已在我们前面讲述WSAEventSelect I/O模型时介绍过了。WSAWaitForMultipleEvents函数会等候一段规定的时间，等待一个或多个事件对象进入“已传信”状态。在此再次提醒大家注意：WSAWaitForMultipleEvents函数一次最多只能等待64个事件对象。发现一次重叠请求完成之后，接着需要调用WSAGetOverlappedResult（取得重叠结构）函数，判断那个重叠调用到底是成功，还是失败。该函数的定义如下：

```
BOOL WSAGetOverlappedResult(  
    SOCKET s,  
    LPWSAOVERLAPPED lpOverlapped,  
    LPDWORD lpcbTransfer,  
    BOOL fWait,  
    LPDWORD lpdwFlags  
);
```

其中，s参数用于指定在重叠操作开始的时候，与之对应的那个套接字。lpOverlapped参数是一个指针，对应于在重叠操作开始时，指定的那个WSAOVERLAPPED结构。lpcbTransfer参数也是一个指针，对应一个DWORD（双字）变量，负责接收一次重叠发送或接收操作实际传输的字节数。fWait参数用于决定函数是否应该等待一次待决（未决）的重叠操作完成。若将fWait设为TRUE，那么除非操作完成，否则函数不会返回；若设为FALSE，而且操作仍然处于“待决”状态，那么WSAGetOverlappedResult函数会返回FALSE值，同时返回一个WSA\_IO\_INCOMPLETE（I/O操作未完成）错误。但就我们目前的情况来说，由于需要等候重叠操作的一个已传信事件完成，所以该参数无论采用什么设置，都没有任何效果。最后一个参数是lpdwFlags，它对应于一个指针，指向一个DWORD（双字），负责接收结果标志（假如原先的重叠调用是用WSARecv或WSARecvFrom函数发出的）。

如WSAGetOverlappedResult函数调用成功，返回值就是TRUE。这意味着我们的重叠I/O操作已成功完成，而且由lpcbTransfer参数指向的值已进行了更新。若返回值是FALSE，那么可能是由下述任何一种原因造成的：

重叠I/O操作仍处在“待决”状态。

重叠操作已经完成，但含有错误。

重叠操作的完成状态不可判决，因为在提供给WSAGetOverlappedResult函数的一个或多个参数中，存在着错误。

失败后，由 `lpcbTransfer` 参数指向的值不会进行更新，而且我们的应用程序应调用 `WSAGetLastError` 函数，调查到底是何种原因造成了调用失败。

在程序清单 8-7 中，我们向大家阐述了如何编制一个简单的服务器应用，令其在一个套接字上对重叠 I/O 操作进行管理，程序完全利用了前述的事件通知机制。对该程序采用的编程步骤总结如下：

- 1) 创建一个套接字，开始在指定的端口上监听连接请求。
  - 2) 接受一个进入的连接请求。
  - 3) 为接受的套接字新建一个 `WSAOVERLAPPED` 结构，并为该结构分配一个事件对象句柄。也将事件对象句柄分配给一个事件数组，以便稍后由 `WSAWaitForMultipleEvents` 函数使用。
  - 4) 在套接字上投递一个异步 `WSARecv` 请求，指定参数为 `WSAOVERLAPPED` 结构。
- 注意 函数通常会以失败告终，返回 `SOCKET_ERROR` 错误状态 `WSA_IO_PENDING` (I/O 操作尚未完成)。
- 5) 使用步骤 3) 的事件数组，调用 `WSAWaitForMultipleEvents` 函数，并等待与重叠调用关联在一起的事件进入“已传信”状态（换言之，等待那个事件的“触发”）。
  - 6) `WSAWaitForMultipleEvents` 函数完成后，针对事件数组，调用 `WSAResetEvent`（重设事件）函数，从而重设事件对象，并对完成的重叠请求进行处理。
  - 7) 使用 `WSAGetOverlappedResult` 函数，判断重叠调用的返回状态是什么。
  - 8) 在套接字上投递另一个重叠 `WSARecv` 请求。
  - 9) 重复步骤 5)~8)。

这个例子极易扩展，提供对多个套接字的支持。方法是代码的重叠 I/O 处理部分移至一个独立的线程内，让主应用程序线程为附加的连接请求提供服务。

程序清单 8-7 采用事件机制的简单重叠 I/O 处理示例

```
void main(void)
{
    WSABUF DataBuf;
    DWORD EventTotal = 0;
    WSAEVENT EventArray[WSA_MAXIMUM_WAIT_EVENTS];
    WSAOVERLAPPED AcceptOverlapped;
    SOCKET ListenSocket, AcceptSocket;

    // Step 1:
    // Start Winsock and set up a listening socket
    ...

    // Step 2:
    // Accept an inbound connection
    AcceptSocket = accept(ListenSocket, NULL, NULL);

    // Step 3:
    // Set up an overlapped structure

    EventArray[EventTotal] = WSACreateEvent();

    ZeroMemory(&AcceptOverlapped,
        sizeof(WSAOVERLAPPED));
```



```
AcceptOverlapped.hEvent = EventArray[EventTotal];

DataBuf.len = DATA_BUFSIZE;
DataBuf.buf = buffer;

EventTotal++;

// Step 4:
// Post a WSARecv request to begin receiving data
// on the socket

WSARecv(AcceptSocket, &DataBuf, 1, &RecvBytes,
        &Flags, &AcceptOverlapped, NULL);

// Process overlapped receives on the socket.

while(TRUE)
{
    // Step 5:
    // Wait for the overlapped I/O call to complete
    Index = WSAWaitForMultipleEvents(EventTotal,
        EventArray, FALSE, WSA_INFINITE, FALSE);

    // Index should be 0 because we
    // have only one event handle in EventArray

    // Step 6:
    // Reset the signaled event
    WSAResetEvent(
        EventArray[Index - WSA_WAIT_EVENT_0]);

    // Step 7:
    // Determine the status of the overlapped
    // request
    WSAGetOverlappedResult(AcceptSocket,
        &AcceptOverlapped, &BytesTransferred,
        FALSE, &Flags);

    // First check to see whether the peer has closed
    // the connection, and if so, close the
    // socket

    if (BytesTransferred == 0)
    {
        printf("Closing socket %d\n", AcceptSocket);

        closesocket(AcceptSocket);
        WSACloseEvent(
            EventArray[Index - WSA_WAIT_EVENT_0]);
        return;
    }

    // Do something with the received data.
    // DataBuf contains the received data.
    ...
}
```

```

// Step 8:
// Post another WSARecv() request on the socket

Flags = 0;
ZeroMemory(&AcceptOverlapped,
    sizeof(WSAOVERLAPPED));

AcceptOverlapped.hEvent = EventArray[Index -
    WSA_WAIT_EVENT_0];

DataBuf.len = DATA_BUFSIZE;
DataBuf.buf = Buffer;

WSARecv(AcceptSocket, &DataBuf, 1,
    &RecvBytes, &Flags, &AcceptOverlapped,
    NULL);
    }
}

```

在Windows NT和Windows 2000中，重叠I/O模型也允许应用程序以一种重叠方式，实现对连接的接受。具体的做法是在监听套接字上调用 AcceptEx函数。AcceptEx是一个特殊的Winsock 1.1扩展函数，位于Mswsock.h头文件以及Mswsock.lib库文件内。该函数最初的设计宗旨是在Windows NT与Windows 2000操作系统上，处理Win32的重叠I/O机制。但事实上，它也适用于Winsock 2中的重叠I/O。AcceptEx的定义如下：

```

BOOL AcceptEx (
    SOCKET sListenSocket,
    SOCKET sAcceptSocket,
    PVOID lpOutputBuffer,
    DWORD dwReceiveDataLength,
    DWORD dwLocalAddressLength,
    DWORD dwRemoteAddressLength,
    LPDWORD lpdwBytesReceived,
    LPOVERLAPPED lpOverlapped
);

```

sListenSocket参数指定的是一个监听套接字。sAcceptSocket参数指定的是另一个套接字，负责对进入连接请求的“接受”。AcceptEx函数和accept函数的区别在于，我们必须提供接受的套接字，而不是让函数自动为我们创建。正是由于要提供套接字，所以要求我们事先调用socket或WSASocket函数，创建一个套接字，以便通过sAcceptSocket参数，将其传递给AcceptEx。lpOutputBuffer参数指定的是一个特殊的缓冲区，因为它要负责三种数据的接收：服务器的本地地址，客户机的远程地址，以及在新建连接上发送的第一个数据块。dwReceiveDataLength参数以字节为单位，指定了在lpOutputBuffer缓冲区中，保留多大的空间，用于数据的接收。如这个参数设为0，那么在连接的接受过程中，不会再一道接收任何数据。dwLocalAddressLength和dwRemoteAddressLength参数也是以字节为单位，指定在lpOutputBuffer缓冲区中，保留多大的空间，在一个套接字被接受的时候，用于本地和远程地址信息的保存。要注意的是，和当前采用的传送协议允许的最大地址长度比较起来，这里指定的缓冲区大小至少应多出16字节。举个例子来说，假定正在使用的是TCP/IP协议，那么这里的大小应设为“SOCKADDR\_IN结构的长度+16字节”。lpdwBytesReceived参数用于返回接收到的实际数据量，以字节为单位。只有在操作以同步方式完成的前提下，才会设置这个参数。假如AcceptEx函数返回ERROR\_IO\_PENDING，

那么这个参数永远都不会设置，我们必须利用完成事件通知机制，获知实际读取的字节量。最后一个参数是lpOverlapped，它对应的是一个OVERLAPPED结构，允许AcceptEx以一种异步方式工作。如我们早先所述，只有在一个重叠I/O应用中，该函数才需要使用事件对象通知机制，这是由于此时没有一个完成例程参数可供使用。

一个名为GetAcceptExSockaddrs的Winsock扩展函数可从lpOutputBuffer中解析出本地和远程地址元素。GetAcceptExSockaddrs定义如下：

```
VOID GetAcceptExSockaddrs(  
    PVOID lpOutputBuffer,  
    DWORD dwReceiveDataLength,  
    DWORD dwLocalAddressLength,  
    DWORD dwRemoteAddressLength,  
    LPSOCKADDR *LocalSockaddr,  
    LPINT LocalSockaddrLength,  
    LPSOCKADDR *RemoteSockaddr,  
    LPINT RemoteSockaddrLength  
);
```

lpOutputBuffer参数应设为自AcceptEx返回的lpOutputBuffer。dwReceiveDataLength、dwLocalAddressLength以及dwRemoteAddressLength参数应设为与我们传递给AcceptEx的dwReceiveDataLength、dwLocalAddressLength以及dwRemoteAddressLength参数相同的值。LocalSockaddr和RemoteSockaddr参数分别是指向一个包含了本地及远程地址信息的SOCKADDR结构的指针，负责从最初的lpOutputBuffer参数接收一个指针偏移。这样一来，根据lpOutputBuffer中包含的地址信息，我们可以非常轻松地对一个SOCKADDR结构中的元素进行引用。LocalSockaddrLength和RemoteSockaddrLength参数则分别负责接收本地及远程地址的长度。

## 2. 完成例程

“完成例程”是我们的应用程序用来管理完成的重叠I/O请求的另一种方法。完成例程其实就是一些函数。最开始的时候，我们将其传递给一个重叠I/O请求，在一个重叠I/O请求完成时由系统调用。它们的基本设计宗旨是通过调用者的线程，为一个已完成的I/O请求提供服务。除此以外，应用程序可通过完成例程，继续进行重叠I/O处理。

如果希望用完成例程为重叠I/O请求提供服务，在我们的应用程序中，必须为一个I/O定义Winsock函数，指定一个完成例程，同时指定一个WSAOVERLAPPED结构。一个完成例程必须拥有下述函数原型：

```
void CALLBACK CompletionROUTINE(  
    DWORD dwError,  
    DWORD cbTransferred,  
    LPWSAOVERLAPPED lpOverlapped,  
    DWORD dwFlags  
);
```

用完成例程完成了一个重叠I/O请求之后，参数中会包含下述信息：

参数dwError表明了一个重叠操作（由lpOverlapped指定）的完成状态是什么。

cbTransferred参数指定了在重叠操作期间，实际传输的字节量是多大。

lpOverlapped参数指定的是传递到最初的I/O调用内的一个WSAOVERLAPPED结构。

dwFlags参数目前尚未使用，应设为0。

在用一個完成例程提交的重疊請求，與用一個事件對象提交的重疊請求之間，存在着一項非常重要的區別。WSAOVERLAPPED結構的事件字段 `hEvent` 並未使用；也就是說，我們不可將一個事件對象同重疊請求關聯到一起。用完成例程發出了一個重疊 I/O 調用之後，作為我們的調用線程，一旦完成，它最終必須為完成例程提供服務。這樣一來，便要求我們將自己的調用線程置于一種“可警告的等待狀態”。並在 I/O 操作完成後，對完成例程加以處理。 `WSAWaitForMultipleEvents` 函數可用來將我們的線程置于一種可警告的等待狀態。這樣做的缺點在於，我們還必須有一個事件對象可用於 `WSAWaitForMultipleEvents` 函數。假定應用程序只用完成例程對重疊請求進行處理，便不可能有任何事件對象需要處理。作為一種變通方法，我們的應用程序可用 Win32 的 `SleepEx` 函數將自己的線程置為一種可警告等待狀態。當然，亦可創建一個偽事件對象，不將它與任何東西關聯在一起。假如調用線程經常處於繁忙狀態，而且並不處在一種可警告的等待狀態，那麼根本不會有投遞的完成例程會得到調用。

如早先所見， `WSAWaitForMultipleEvents` 通常會等待同 WSAOVERLAPPED 結構關聯在一起的事件對象。該函數也設計用於將我們的線程置入一種可警告等待狀態，並可為已經完成的 `WSAIO` 請求進行完成例程的處理（前提是將 `fAlertable` 參數設為 `TRUE`）。用一個完成例程結束了重疊 I/O 請求之後，返回值是 `WSA_IO_COMPLETION`，而不是事件數組中的一個事件對象索引。 `SleepEx` 函數的行為實際上和 `WSAWaitForMultipleEvents` 差不多，只是它不需要任何事件對象。對 `SleepEx` 函數的定義如下：

```
DWORD SleepEx(  
    DWORD dwMilliseconds,  
    BOOL bAlertable  
);
```

其中， `dwMilliseconds` 參數定義了 `SleepEx` 函數的等待時間，以毫秒為單位。假如將 `dwMilliseconds` 設為 `INFINITE`，那麼 `SleepEx` 會無休止地等待下去。 `bAlertable` 參數規定了一個完成例程的執行方式。假如將 `bAlertable` 設為 `FALSE`，而且進行了一次 I/O 完成回調，那麼 I/O 完成函數不會執行，而且函數不會返回，除非超過由 `dwMilliseconds` 規定的時間。若設為 `TRUE`，那麼完成例程會得到執行，同時 `SleepEx` 函數返回 `WAIT_IO_COMPLETION`。

在程序清單 8-8 中，我們向大家展示了如何構建一個簡單的服務器應用，令其採用前述的方法，通過完成例程，來實現對一個套接字請求的管理。該程序的編碼主要按下述步驟進行：

- 1) 新建一個套接字，開始在指定端口上，監聽一個進入的連接。
- 2) 接受一個進入的連接請求。
- 3) 為接受的套接字創建一個 WSAOVERLAPPED 結構。
- 4) 在套接字上投遞一個異步 `WSARecv` 請求，方法是將 WSAOVERLAPPED 指定成為參數，同時提供一個完成例程。
- 5) 在將 `fAlertable` 參數設為 `TRUE` 的前提下，調用 `WSAWaitForMultipleEvents`，並等待一個重疊 I/O 請求完成。重疊請求完成後，完成例程會自動執行，而且 `WSAWaitForMultipleEvents` 會返回一個 `WSA_IO_COMPLETION`。在完成例程內，可隨一個完成例程一道，投遞另一個重疊 `WSARecv` 請求。
- 6) 檢查 `WSAWaitForMultipleEvents` 是否返回 `WSA_IO_COMPLETION`。
- 7) 重複步驟 5) 和 6)。

程序清单 8-8 采用完成例程的简单重叠 I/O 处理示例

```
SOCKET AcceptSocket;
WSABUF DataBuf;

void main(void)
{
    WSAOVERLAPPED Overlapped;

    // Step 1:
    // Start Winsock, and set up a listening socket
    ...

    // Step 2:
    // Accept a new connection
    AcceptSocket = accept(ListenSocket, NULL, NULL);

    // Step 3:
    // Now that we have an accepted socket, start
    // processing I/O using overlapped I/O with a
    // completion routine. To get the overlapped I/O
    // processing started, first submit an
    // overlapped WSAREcv() request.

    Flags = 0;

    ZeroMemory(&Overlapped, sizeof(WSAOVERLAPPED));

    DataBuf.len = DATA_BUFSIZE;
    DataBuf.buf = Buffer;
    // Step 4:
    // Post an asynchronous WSAREcv() request
    // on the socket by specifying the WSAOVERLAPPED
    // structure as a parameter, and supply
    // the WorkerRoutine function below as the
    // completion routine

    if (WSAREcv(AcceptSocket, &DataBuf, 1, &RecvBytes,
        &Flags, &Overlapped, WorkerRoutine)
        == SOCKET_ERROR)
    {
        if (WSAGetLastError() != WSA_IO_PENDING)
        {
            printf("WSAREcv() failed with error %d\n",
                WSAGetLastError());
            return;
        }
    }

    // Since the WSAWaitForMultipleEvents() API
    // requires waiting on one or more event objects,
    // we will have to create a dummy event object.
    // As an alternative, we can use SleepEx()
    // instead.

    EventArray[0] = WSACreateEvent();
```

```

while(TRUE)
{
    // Step 5:
    Index = WSAWaitForMultipleEvents(1, EventArray,
        FALSE, WSA_INFINITE, TRUE);

    // Step 6:
    if (Index == WAIT_IO_COMPLETION)
    {
        // An overlapped request completion routine
        // just completed. Continue servicing
        // more completion routines.
        break;
    }
    else
    {
        // A bad error occurred--stop processing!
        // If we were also processing an event
        // object, this could be an index to
        // the event array.
        return;
    }
}

}

void CALLBACK WorkerRoutine(DWORD Error,
    DWORD BytesTransferred,
    LPWSAOVERLAPPED Overlapped,
    DWORD InFlags)
{
    DWORD SendBytes, RecvBytes;
    DWORD Flags;

    if (Error != 0 || BytesTransferred == 0)
    {
        // Either a bad error occurred on the socket
        // or the socket was closed by a peer
        closesocket(AcceptSocket);
        return;
    }

    // At this point, an overlapped WSAREcv() request
    // completed successfully. Now we can retrieve the
    // received data that is contained in the variable
    // DataBuf. After processing the received data, we
    // need to post another overlapped WSAREcv() or
    // WSASend() request. For simplicity, we will post
    // another WSAREcv() request.

    Flags = 0;

    ZeroMemory(&Overlapped, sizeof(WSAOVERLAPPED));

    DataBuf.len = DATA_BUFSIZE;
    DataBuf.buf = Buffer;

    if (WSAREcv(AcceptSocket, &DataBuf, 1, &RecvBytes,

```



```
&Flags, &Overlapped, WorkerRoutine)
== SOCKET_ERROR)
{
    if (WSAGetLastError() != WSA_IO_PENDING )
    {
        printf("WSARecv() failed with error %d\n",
            WSAGetLastError());
        return;
    }
}
}
```

### 8.2.5 完成端口模型

“完成端口”模型是迄今为止最为复杂的一种 I/O 模型。然而，假若一个应用程序同时需要管理为数众多的套接字，那么采用这种模型，往往可以达到最佳的系统性能！但不幸的是，该模型只适用于 Windows NT 和 Windows 2000 操作系统。因其设计的复杂性，只有在你的应用程序需要同时管理数百乃至上千个套接字的时候，而且希望随着系统内安装的 CPU 数量的增多，应用程序的性能也可以线性提升，才应考虑采用“完成端口”模型。要记住的一个基本准则是，假如要为 Windows NT 或 Windows 2000 开发高性能的服务器应用，同时希望为大量套接字 I/O 请求提供服务（Web 服务器便是这方面的典型例子），那么 I/O 完成端口模型便是最佳选择！

从本质上说，完成端口模型要求我们创建一个 Win32 完成端口对象，通过指定数量的线程，对重叠 I/O 请求进行管理，以便为已经完成的重叠 I/O 请求提供服务。要注意的是，所谓“完成端口”，实际是 Win32、Windows NT 以及 Windows 2000 采用的一种 I/O 构造机制，除套接字句柄之外，实际上还可接受其他东西。然而，本节只打算讲述如何使用套接字句柄，来发挥完成端口模型的巨大威力。使用这种模型之前，首先要创建一个 I/O 完成端口对象，用它面向任意数量的套接字句柄，管理多个 I/O 请求。要做到这一点，需要调用 CreateCompletionPort 函数。该函数定义如下：

```
HANDLE CreateIoCompletionPort(
    HANDLE FileHandle,
    HANDLE ExistingCompletionPort,
    DWORD CompletionKey,
    DWORD NumberOfConcurrentThreads
);
```

在我们深入探讨其中的各个参数之前，首先要注意该函数实际用于两个明显有别的目的：  
用于创建一个完成端口对象。

将一个句柄同完成端口关联到一起。

最开始创建一个完成端口时，唯一感兴趣的参数便是 NumberOfConcurrentThreads（并发线程的数量）；前面三个参数都会被忽略。NumberOfConcurrentThreads 参数的特殊之处在于，它定义了在一个完成端口上，同时允许执行的线程数量。理想情况下，我们希望每个处理器各自负责一个线程的运行，为完成端口提供服务，避免过于频繁的线程“场景”切换。若将该参数设为 0，表明系统内安装了多少个处理器，便允许同时运行多少个线程！可用下述代码创建一个 I/O 完成端口：

```
CompletionPort = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 0);
```

该语句的作用是返回一个句柄，在为完成端口分配了一个套接字句柄后，用来对那个端口进行标定（引用）。

### 1. 工作者线程与完成端口

成功创建一个完成端口后，便可开始将套接字句柄与对象关联到一起。但在关联套接字之前，首先必须创建一个或多个“工作者线程”，以便在I/O请求投递给完成端口对象后，为完成端口提供服务。在这个时候，大家或许会觉得奇怪，到底应创建多少个线程，以便为完成端口提供服务呢？这实际正是完成端口模型显得颇为“复杂”的一个方面，因为服务I/O请求所需的数量取决于应用程序的总体设计情况。在此要记住的一个重点在于，在我们调用CreateIoCompletionPort时指定的并发线程数量，与打算创建的工作者线程数量相比，它们代表的并非同一件事情。早些时候，我们曾建议大家用CreateIoCompletionPort函数为每个处理器都指定一个线程（处理器的数量有多少，便指定多少线程）以避免由于频繁的线程“场景”交换活动，从而影响系统的整体性能。CreateIoCompletionPort函数的NumberOfConcurrentThreads参数明确指示系统：在一个完成端口上，一次只允许n个工作者线程运行。假如在完成端口上创建的工作者线程数量超出n个，那么在同一时刻，最多只允许n个线程运行。但实际上，在一段较短的时间内，系统有可能超过这个值，但很快便会把它减少至事先在CreateIoCompletionPort函数中设定的值。那么，为何实际创建的工作者线程数量有时要比CreateIoCompletionPort函数设定的多一些呢？这样做有必要吗？如先前所述，这主要取决于应用程序的总体设计情况。假定我们的某个工作者线程调用了函数，比如Sleep或WaitForSingleObject，但却进入了暂停（锁定或挂起）状态，那么允许另一个线程代替它的位置。换言之，我们希望随时都能执行尽可能多的线程；当然，最大的线程数量是事先在CreateIoCompletionPort调用里设定好的。这样一来，假如事先预计到自己的线程有可能暂时处于停顿状态，那么最好能够创建比CreateIoCompletionPort的NumberOfConcurrentThreads参数的值多的线程，以便到时候充分发挥系统的潜力。

一旦在完成端口上拥有足够多的工作者线程来为I/O请求提供服务，便可着手将套接字句柄同完成端口关联到一起。这要求我们在一个现有的完成端口上，调用CreateIoCompletionPort函数，同时为前三个参数——FileHandle、ExistingCompletionPort和CompletionKey——提供套接字的信息。其中，FileHandle参数指定一个要同完成端口关联在一起的套接字句柄。

ExistingCompletionPort参数指定的是一个现有的完成端口。CompletionKey（完成键）参数则指定要与某个特定套接字句柄关联在一起的“单句柄数据”；在这个参数中，应用程序可保存与一个套接字对应的任意类型的信息。之所以把它叫作“单句柄数据”，是由于它只对应着与那个套接字句柄关联在一起的数据。可将其作为指向一个数据结构的指针，来保存套接字句柄；在那个结构中，同时包含了套接字的句柄，以及与那个套接字有关的其他信息。就象本章稍后还会讲述的那样，为完成端口提供服务的线程例程可通过这个参数，取得与套接字句柄有关的信息。

根据我们到目前为止学到的东西，首先来构建一个基本的应用程序框架。程序清单 8-9向大家阐述了如何使用完成端口模型，来开发一个回应（或“反射”）服务器应用。在这个程序中，我们基本上按下述步骤行事：

- 1) 创建一个完成端口。第四个参数保持为0，指定在完成端口上，每个处理器一次只允许

执行一个工作者线程。

2) 判断系统内到底安装了多少个处理器。

3) 创建工作者线程，根据步骤 2)得到的处理器信息，在完成端口上，为已完成的 I/O 请求提供服务。在这个简单的例子中，我们为每个处理器都只创建一个工作者线程。这是由于事先已预计到，到时不会有任何线程进入“挂起”状态，造成由于线程数量的不足，而使处理器空闲的局面（没有足够的线程可供执行）。调用 CreateThread 函数时，必须同时提供一个工作者例程，由线程在创建好执行。本节稍后还会详细讨论线程的职责。

4) 准备好一个监听套接字，在端口 5150 上监听进入的连接请求。

5) 使用 accept 函数，接受进入的连接请求。

6) 创建一个数据结构，用于容纳“单句柄数据”，同时在结构存入接受的套接字句柄。

7) 调用 CreateIoCompletionPort，将自 accept 返回的新套接字句柄同完成端口关联到一起。通过完成键（CompletionKey）参数，将单句柄数据结构传递给 CreateIoCompletionPort。

8) 开始在已接受的连接上进行 I/O 操作。在此，我们希望通过重叠 I/O 机制，在新建的套接字上投递一个或多个异步 WSARECV 或 WSASEND 请求。这些 I/O 请求完成后，一个工作者线程会为 I/O 请求提供服务，同时继续处理未来的 I/O 请求，稍后便会在步骤 3) 指定的工作者例程中，体验到这一点。

9) 重复步骤 5)~8)，直至服务器中止。

程序清单 8-9 完成端口的建立

```
StartWinsock();

// Step 1:
// Create an I/O completion port

CompletionPort = CreateIoCompletionPort(
    INVALID_HANDLE_VALUE, NULL, 0, 0);

// Step 2:
// Determine how many processors are on the system

GetSystemInfo(&SystemInfo);

// Step 3:
// Create worker threads based on the number of
// processors available on the system. For this
// simple case, we create one worker thread for each
// processor.

for(i = 0; i < SystemInfo.dwNumberOfProcessors;
    i++)
{
    HANDLE ThreadHandle;

    // Create a server worker thread, and pass the
    // completion port to the thread. NOTE: the
    // ServerWorkerThread procedure is not defined
    // in this listing.
```

```
ThreadHandle = CreateThread(NULL, 0,
    ServerWorkerThread, CompletionPort,
    0, &ThreadID);

// Close the thread handle
CloseHandle(ThreadHandle);
}

// Step 4:
// Create a listening socket

Listen = WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0,
    WSA_FLAG_OVERLAPPED);

InternetAddr.sin_family = AF_INET;
InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
InternetAddr.sin_port = htons(5150);
bind(Listen, (PSOCKADDR) &InternetAddr,
    sizeof(InternetAddr));

// Prepare socket for listening

listen(Listen, 5);

while(TRUE)
{
    // Step 5:
    // Accept connections and assign to the completion
    // port

    Accept = WSAAccept(Listen, NULL, NULL, NULL, 0);

    // Step 6:
    // Create per-handle data information structure to
    // associate with the socket
    PerHandleData = (LPPER_HANDLE_DATA)
        GlobalAlloc(GPTR, sizeof(PER_HANDLE_DATA));

    printf("Socket number %d connected\n", Accept);
    PerHandleData->Socket = Accept;

    // Step 7:
    // Associate the accepted socket with the
    // completion port

    CreateIoCompletionPort((HANDLE) Accept,
        CompletionPort, (DWORD) PerHandleData, 0);

    // Step 8:
    // Start processing I/O on the accepted socket.
    // Post one or more WSASend() or WSARecv() calls
    // on the socket using overlapped I/O.
    WSARecv(...);
}
```

## 2. 完成端口和重叠 I/O

将套接字句柄与一个完成端口关联在一起后，便可以套接字句柄为基础，投递发送与接收请求，开始对 I/O 请求的处理。接下来，可开始依赖完成端口，来接收有关 I/O 操作完成情况的的通知。从本质上说，完成端口模型利用了 Win32 重叠 I/O 机制。在这种机制中，象 WSA Send 和 WSA Recv 这样的 Winsock API 调用会立即返回。此时，需要由我们的应用程序负责在以后的某个时间，通过一个 OVERLAPPED 结构，来接收调用的结果。在完成端口模型中，要想做到这一点，需要使用 GetQueuedCompletionStatus（获取排队完成状态）函数，让一个或多个工作者线程在完成端口上等待。该函数的定义如下：

```
BOOL GetQueuedCompletionStatus(  
    HANDLE CompletionPort,  
    LPDWORD lpNumberOfBytesTransferred,  
    LPDWORD lpCompletionKey,  
    LPOVERLAPPED * lpOverlapped,  
    DWORD dwMilliseconds  
);
```

其中，CompletionPort 参数对应于要在上面等待的完成端口。lpNumberOfBytesTransferred 参数负责在完成了一次 I/O 操作后（如 WSA Send 或 WSA Recv），接收实际传输的字节数。lpCompletionKey 参数为原先传递进入 CreateCompletionPort 函数的套接字返回“单句柄数据”。如我们早先所述，大家最好将套接字句柄保存在这个“键”（Key）中。lpOverlapped 参数用于接收完成的 I/O 操作的重叠结果。这实际是一个相当重要的参数，因为可用它获取每个 I/O 操作的数据。而最后一个参数，dwMilliseconds，用于指定调用者希望等待一个完成数据包在完成端口上出现的时间。假如将其设为 INFINITE，调用会无休止地等待下去。

## 3. 单句柄数据和单 I/O 操作数据

一个工作者线程从 GetQueuedCompletionStatus 这个 API 调用接收到 I/O 完成通知后，在 lpCompletionKey 和 lpOverlapped 参数中，会包含一些必要的套接字信息。利用这些信息，可通过完成端口，继续在一个套接字上的 I/O 处理。通过这些参数，可获得两方面重要的套接字数据：单句柄数据，以及单 I/O 操作数据。

其中，lpCompletionKey 参数包含了“单句柄数据”，因为在一个套接字首次与完成端口关联到一起的时候，那些数据便与一个特定的套接字句柄对应起来了。这些数据正是我们在进行 CreateIoCompletionPort API 调用的时候，通过 CompletionKey 参数传递的。如早先所述，应用程序可通过该参数传递任意类型的数据。通常情况下，应用程序会将与 I/O 请求有关的套接字句柄保存在这里。

lpOverlapped 参数则包含了一个 OVERLAPPED 结构，在它后面跟随“单 I/O 操作数据”。我们的工作者线程处理一个完成数据包时（将数据原封不动打转回去，接受连接，投递另一个线程，等等），这些信息是它必须要知道的。单 I/O 操作数据可以是追加到一个 OVERLAPPED 结构末尾的、任意数量的字节。假如一个函数要求用到一个 OVERLAPPED 结构，我们便必须将这样的一个结构传递进去，以满足它的要求。要想做到这一点，一个简单的方法是定义一个结构，然后将 OVERLAPPED 结构作为新结构的第一个元素使用。举个例子来说，可定义下述数据结构，实现对单 I/O 操作数据的管理：

```
typedef struct  
{
```

```

OVERLAPPED Overlapped;
WSABUF      DataBuf;
CHAR        Buffer[DATA_BUFSIZE];
BOOL        OperationType;
} PER_IO_OPERATION_DATA;

```

该结构演示了通常要与 I/O 操作关联在一起的某些重要数据元素，比如刚才完成的那个 I/O 操作的类型（发送或接收请求）。在这个结构中，我们认为用于已完成 I/O 操作的数据缓冲区是非常有用的。要想调用一个 Winsock API 函数，同时为其分配一个 OVERLAPPED 结构，既可将自己的结构“造型”为一个 OVERLAPPED 指针，亦可简单地撤消对结构中的 OVERLAPPED 元素的引用。如下例所示：

```

PER_IO_OPERATION_DATA PerIoData;

// 可像下面这样调用一个函数
WSARecv(socket, ..., (OVERLAPPED *)&PerIoData);
// 或像这样
WSARecv(socket, ..., &(PerIoData.Overlapped));

```

在工作线程的后面部分，等 GetQueuedCompletionStatus 函数返回了一个重叠结构（和完成键）后，便可通过撤消对 OperationType 成员的引用，调查到底是哪个操作投递到了这个句柄之上（只需将返回的重叠结构造型为自己的 PER\_IO\_OPERATION\_DATA 结构）。对单 I/O 操作数据来说，它最大的一个优点便是允许我们在同一个句柄上，同时管理多个 I/O 操作（读 / 写，多个读，多个写，等等）。大家此时或许会产生这样的疑问：在同一个套接字上，真的有必要同时投递多个 I/O 操作吗？答案在于系统的“伸缩性”，或者说“扩展能力”。例如，假定我们的机器安装了多个中央处理器，每个处理器都在运行一个工作者线程，那么在同一个时候，完全可能有几个不同的处理器在同一个套接字上，进行数据的收发操作。

为了完成前述的简单回应服务器示例，我们需要提供一个 ServerWorkerThread（服务器工作者线程）函数。在程序清单 8-10 中，我们展示了如何设计一个工作者线程例程，令其使用单句柄数据以及单 I/O 操作数据，为 I/O 请求提供服务。

程序清单 8-10 完成端口工作者线程

```

DWORD WINAPI ServerWorkerThread(
    LPVOID CompletionPortID)
{
    HANDLE CompletionPort = (HANDLE) CompletionPortID;
    DWORD BytesTransferred;
    LPOVERLAPPED Overlapped;
    LPPER_HANDLE_DATA PerHandleData;
    LPPER_IO_OPERATION_DATA PerIoData;
    DWORD SendBytes, RecvBytes;
    DWORD Flags;

    while(TRUE)
    {
        // Wait for I/O to complete on any socket
        // associated with the completion port

        GetQueuedCompletionStatus(CompletionPort,
            &BytesTransferred, (LPDWORD)&PerHandleData,
            (LPOVERLAPPED *)&PerIoData, INFINITE);
    }
}

```



```

// First check to see whether an error has occurred
// on the socket; if so, close the
// socket and clean up the per-handle data
// and per-I/O operation data associated with
// the socket

if (BytesTransferred == 0 &&
    (PerIoData->OperationType == RECV_POSTED ||
     PerIoData->OperationType == SEND_POSTED))
{
    // A zero BytesTransferred indicates that the
    // socket has been closed by the peer, so
    // you should close the socket. Note:
    // Per-handle data was used to reference the
    // socket associated with the I/O operation.

    closesocket(PerHandleData->Socket);

    GlobalFree(PerHandleData);
    GlobalFree(PerIoData);
    continue;
}

// Service the completed I/O request. You can
// determine which I/O request has just
// completed by looking at the OperationType
// field contained in the per-I/O operation data.
if (PerIoData->OperationType == RECV_POSTED)
{
    // Do something with the received data
    // in PerIoData->Buffer
}

// Post another WSASend or WSARecv operation.
// As an example, we will post another WSARecv()
// I/O operation.

Flags = 0;

// Set up the per-I/O operation data for the next
// overlapped call
ZeroMemory(&(PerIoData->Overlapped),
    sizeof(OVERLAPPED));

PerIoData->DataBuf.len = DATA_BUFSIZE;
PerIoData->DataBuf.buf = PerIoData->Buffer;
PerIoData->OperationType = RECV_POSTED;

WSARecv(PerHandleData->Socket,
    &(PerIoData->DataBuf), 1, &RecvBytes,
    &Flags, &(PerIoData->Overlapped), NULL);
}
}

```

在程序清单8-9和程序清单8-10列出的简单服务器示例中（配套光盘也有），最后要注意的一处细节是如何正确地关闭 I/O 完成端口——特别是同时运行了一个或多个线程，在几个不同

的套接字上执行I/O操作的时候。要避免的一个重要问题是在进行重叠 I/O操作的同时，强行释放一个OVERLAPPED结构。要想避免出现这种情况，最好的办法是针对每个套接字句柄，调用closesocket函数，任何尚未进行的重叠 I/O操作都会完成。一旦所有套接字句柄都已关闭，便需在完成端口上，终止所有工作者线程的运行。要想做到这一点，需要使用PostQueuedCompletionStatus函数，向每个工作者线程都发送一个特殊的完成数据包。该函数会指示每个线程都“立即结束并退出”。下面是PostQueuedCompletionStatus函数的定义：

```
BOOL PostQueuedCompletionStatus(  
    HANDLE CompletionPort,  
    DWORD dwNumberOfBytesTransferred,  
    DWORD dwCompletionKey,  
    LPOVERLAPPED lpOverlapped  
);
```

其中，CompletionPort参数指定想向其发送一个完成数据包的完成端口对象。而就dwNumberOfBytesTransferred、dwCompletionKey和lpOverlapped这三个参数来说，每一个都允许我们指定一个值，直接传递给GetQueuedCompletionStatus函数中对应的参数。这样一来，一个工作者线程收到传递过来的三个GetQueuedCompletionStatus函数参数后，便可根据由这三个参数的某一个设置的特殊值，决定何时应该退出。例如，可用dwCompletionPort参数传递0值，而一个工作者线程会将其解释成中止指令。一旦所有工作者线程都已关闭，便可使用CloseHandle函数，关闭完成端口，最终安全退出程序。

#### 4. 其他问题

另外还有几种颇有价值的技术，可用来进一步改善套接字应用程序的总体 I/O性能。值得考虑的一项技术是试验不同的套接字缓冲区大小，以改善 I/O性能和应用程序的扩展能力。例如，假如某个程序只采用了一个比较大的缓冲区，仅能支持一个WSARecv请求，而不是同时设置三个较小的缓冲区，提供对三个WSARecv请求的支持，那么该程序的扩展能力并不是很好，特别是在转移到安装了多个处理器的机器上之后。这是由于单独一个缓冲区每次只能处理一个线程！除此以外，单缓冲区设计还会对性能造成一定的干扰：假如一次仅能进行一次接收操作，网络协议驱动程序的潜力便不能得到充分发挥（它经常都会很“闲”）。换言之，假如在接收更多的数据前，需要等等一次WSARecv操作的完成，那么在WSARecv完成和下一次接收之间，整个协议实际上处于“休息”状态。

另一个值得考虑的性能改进措施是用套接字选项SO\_SNDBUF和SO\_RCVBUF对内部套接字缓冲区的大小进行控制。利用这些选项，应用程序可更改一个套接字的内部数据缓冲区的大小。如将该设为0，Winsock便会在重叠I/O调用中直接使用应用程序的缓冲区，进行数据在协议堆栈里的传入、传出。这样一来，在应用程序与Winsock之间，便避免了进行一次缓冲区复制的必要。下述代码片断阐释了如何使用SO\_SNDBUF选项，来进行setsockopt函数的调用：

```
int nZero = 0;  
  
setsockopt(socket, SOL_SOCKET, SO_SNDBUF,  
    (char *)&nZero, sizeof(nZero));
```

要注意的是，将这些缓冲区的大小设为0后，只有在一段给定的时间内，存在着多个I/O请求的前提下，才会产生积极作用。等到第9章，我们会向大家更深入地讲述套接字选项的知识。

提升性能的最后一项措施是使用AcceptEx这个API调用，来进行连接请求的处理，并投递

少量数据。这样一来，我们的应用程序只需通过一次 API 调用，便可为一次接受请求和数据的接收提供服务，从而减少了单独进行 `accept` 和 `WSARecv` 调用造成的开销。这样做还有另一个好处，我们可使用完成端口为 `AcceptEx` 提供服务，因为它也提供了一个 `OVERLAPPED` 结构。假如事先预计到自己的服务器应用在一个连接建好之后，只会进行少量的 `recv-send`（收发）操作，那么 `AcceptEx` 便显得相当有用（比如在设计一个 Web 服务器的时候）。否则的话，接受一个连接后，假如程序要负责数百上千次数据传输操作，这样做对性能便没多大的助益。

最后提醒大家注意，在 Winsock 中，一个 Winsock 应用不应使用 `ReadFile` 和 `WriteFile` 这两个 Win32 函数，在一个完成端口上进行 I/O 处理。尽管这两个函数确实提供了一个 `OVERLAPPED` 结构，而且可在完成端口上成功地使用，但就在 Winsock 2 环境下进行 I/O 处理来说，`WSARecv` 和 `WSASend` 这两个函数却进行了更大程度的优化。若使用 `ReadFile` 和 `WriteFile`，需进行大量不必要的内核 / 用户模式进程调用、线程执行场景的频繁切换以及参数的汇集等等，使总体性能大打折扣。

### 8.3 I/O 模型的问题

现在，对于如何挑选最适合自己的应用程序的 I/O 模型，大家心中可能还没什么数。前面已经提到，每种模型都有自己的优点和缺点。同开发一个简单的锁定模式应用相比（运行许多服务线程），其他每种 I/O 模型都需要更为复杂的编程工作。因此，针对客户机和服务器应用的开发，我们分别提供了下述建议。

#### 1. 客户机的开发

若打算开发一个客户机应用，令其同时管理一个或多个套接字，那么建议采用重叠 I/O 或 `WSAEventSelect` 模型，以便在一定程度上提升性能。然而，假如开发的是一个以 Windows 为基础的应用程序，要进行窗口消息的管理，那么 `WSAAsyncSelect` 模型恐怕是一种最好的选择，因为 `WSAAsyncSelect` 本身便是从 Windows 消息模型借鉴来的。若采用这种模型，我们的程序一开始便具备了处理消息的能力。

#### 2. 服务器的开发

若开发的是一个服务器应用，要在一个给定的时间，同时控制几个套接字，建议大家采用重叠 I/O 模型，这同样是从性能出发点考虑的。但是，如果预计到自己的服务器在任何给定的时间，都会为大量 I/O 请求提供服务，便应考虑使用 I/O 完成端口模型，从而获得更好的性能。

### 8.4 小结

至此，我们已全面介绍了可用于 Winsock 的各种 I/O 模型。这些模型从简单的锁定 I/O，到高性能的完成端口 I/O（获得最大的吞吐速度），使不同的应用程序可根据自己的要求，对 Winsock I/O 的性能进行充分的调整。本章也提供了对 Winsock 一些常规用途的解释。到目前为止，大家已学习可选用哪些传送协议、可设置哪些套接字创建属性、如何创建基本客户机 / 服务器应用以及与 Winsock 有关的其他常规主题。从第 9 章开始，到第 14 章，我们将讲述一些特殊的 Winsock 主题。首先在下一章（第 9 章），我们要解释可对套接字及基层协议的行为产生影响的一些套接字选项与 I/O 控制命令。