CHAPTER 7

WSAAsyncSelect 模型開發

WSAAsyncSelect 模型是 Windows Sockets 的一個非同步 I/O 模型。利用該模型應用程式可以在一個 Socket 上，接收以 Windows 訊息為基礎的網路事件。Windows Sockets 應用程式在建立 Socket 後，呼叫 WSAAsyncSelect()函式註冊感興趣的網路事件。當該事件發生時 Windows 視窗收到訊息，然後應用程式就可以對接收到的網路事件進行處理。

本章講述 WSAAsyncSelect 模型和利用該模型開發的區域網路簡易網路聊天程式。

7.1 WSAAsyncSelect 模型分析

WSAAsyncSelect 模型是 Select 模型的非同步版本。該模型最早出現在 Windows Sockets 的 1.1 版本，用於幫助早期的開發人員在 16 位元 Windows 平台，適應多工訊息環境。該模型應用在一個標準的 Windows 應用程式中，並且得到 Microsoft Foundation Class（MFC）Sockets 類別的採用。本節講述什麼是 WSAAsyncSelect 模型，以及與 Select 模型相比存在哪些不同。

7.1.1 WSAAsyncSelect 模型

第 6 章學習 Windows Sockes 的 Select 模型。在應用程式中呼叫 select()函式時，會發生阻塞現象。開發人員可以透過 select()函式 timeout 參數，設置函式呼叫的阻塞時間。在設

定時間內，執行緒保持等待，直到其中的一個或者多個 Socket 滿足可讀或者可寫的條件，該函式才返回。

WSAAsyncSelect 模型是非阻塞的。如圖 7.1 所示，Windows Sockets 應用程式在呼叫 recv()函式接收資料之前，呼叫 WSAAsyncSelect()函式註冊網路事件。WSAAsyncSelect()函式立即返回，執行緒繼續執行。當系統中資料準備好時，向應用程式傳送訊息。應用程式接收到這個訊息後，呼叫 recv()函式接收資料。

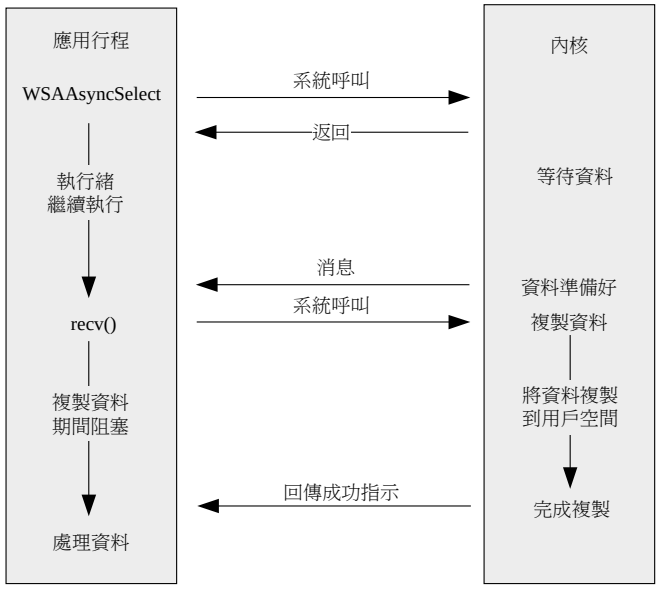

圖 7.1 WSAAsyncSelect 模型

7.1.2 與 Select 模型比較

WSAAsyncSelect 模型與 Select 模型的相同點是，他們都可以對 Windows Socket 應用程式所使用的多個 Socket 進行有效的管理。

但 WSAAsyncSelect 模型與 Select 模型相比存在以下不同。

- WSAAsyncSelect 模型是非同步的。在應用程式中呼叫 WSAAsyncSelect()函式，通知系統感興趣的網路事件，該函式立即返回，應用程式繼續執行。
- 發生網路事件時，應用程式得到通知的方式不同。select()函式返回時，說明某個或者某些 Socket 滿足可讀可寫的條件，應用程式需要使用 FD_ISSET 巨集，判斷 Socket

是否存在於可讀可寫集合中。而對於 WSAAsyncSelect 模型來說，當網路事件發生時，系統向應用程式傳送訊息。

- WSAAsyncSelect 模型應用在基於訊息的 Windows 環境下，使用該模型時必須建立視窗。而 Select 模型廣泛應用在 Unix 系統和 Windows 系統，使用該模型不需要建立視窗。
- 應用程式中呼叫 WSAAsyncSelect()函式後，自動將 Socket 設置為非阻塞模式。而應用程式中呼叫 select()函式後，並不能改變該 Socket 的工作方式。

7.2 WSAAsyncSelect 模型實作

WSAAsyncSelect 模型核心是 WSAAsyncSelect()函式，該函式使得 Windows 應用程式能夠接收網路事件訊息。在應用程式視窗程序中對接收到的網路事件進行處理。由於 WSAAsyncSelect 模型應用在基於訊息的 Windows 應用程式中，所以本節還將講解視窗程序和如何建立視窗等內容。

本節內容分為 WSAAsyncSelect()函式、視窗程序和建立視窗 3 個部分。

7.2.1 WSAAsyncSelect()函式

WSAAsyncSelect()函式功能是請求當網路事件發生時為 Socket 傳送訊息。該函式原形如下：

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

- s：需要事件通知的 Socket。
- hWnd：當網路事件發生時接收訊息的視窗控制碼。
- wMsg：當網路事件發生時視窗收到的訊息。
- lEvent：應用程式感興趣的網路事件集合。

當應用程式中呼叫該函式後，自動將 Socket 設置為非阻塞模式。通常，應用程式定義的訊息要比 Windows 的 WM_USER 值大，以避免該訊息與 Windows 預定義訊息發生混淆。

網路事件種類和涵義見如表 7.1 所示。

表 7.1 網路事件類型

種類	涵義
FD_READ	欲接收可讀的通知
FD_WRITE	欲接收可寫讀的通知
FD_ACCEPT	欲接收等待接受連線的通知
FD_CONNECT	欲接收一次連線或者多點操作完成的通知
FD_OOB	欲接收有緩衝區外（OOB）資料到達的通知
FD_CLOSE	欲接收 Socket 關閉的通知
FD_QOS	欲接收 Socket 服務質量發生變化的通知
FD_GROUP_QOS	欲接收 Socket 組服務質量發生變化的通知
FD_ROUTING_INTERFACE_CHANGE	欲在指定方向上，與路由介面發生變化的通知
FD_ADDRESS_LIST_CHANGE	欲接收針對 Socket 的協定家族，本機位址列表發生變化的通知

開發人員註冊哪種網路事件，取決於實際的需要。如果應用程式同時對多個網路事件感興趣。需要對網路事件類型執行按位元 OR（或）運算。然後將它們分配給 lEvent 參數。例如，應用程式希望在 Socket 上接收有關連線完成、資料可讀和 Socket 關閉的網路事件。那麼在應用程式中，呼叫 WSAAsyncSelect()函式如下所示：

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

當該 Socket 連線完成、有資料可讀或者 Socket 關閉的網路事件事件發生時，就會有 WM_SOCKET 訊息傳送給視窗控制碼為 hwnd 的視窗。

7.2.2 視窗程序

當呼叫 WSAAsyncSelect()函式後，應用程式會在 hWnd 視窗控制碼對應的視窗程序，以訊息形式接收網路事件通知。視窗程序是回呼函式，當成功建立視窗後由系統呼叫。視窗程序原形如下：

```
LRESULT CALLBACK WindowProc(
    HWND hWnd,          //視窗控制碼
    UINT uMsg,          //訊息
    WPARAM wParam,      //訊息參數
    LPARAM lParam       //訊息參數
);
```

- hWnd：視窗控制碼。
- uMsg：訊息。對 Windows Sockets 應用程式來說感興趣的是在 WSAAsyncSelect()函式中，由應用程式定義的訊息。
- wParam：訊息參數。在 Windows Sockets 應用程式中，該參數指明發生網路事件的 Socket。
- lParam：訊息參數。在 Windows Sockets 應用程式中，該參數低位元組指明已經發生的網路事件。高位元組包含可能出現的錯誤代碼。

在 Windows Sockets 應用程式中，當 WindowProc()函式接收到網路事件訊息時，在該函式內執行下面步驟。

1. 讀取 lParam 參數高位元組，判斷是否發生了一個網路錯誤事件。可以使用 WSAGETSELECTERROR 巨集。
2. 如果應用程式發現 Socket 上沒有發生任何錯誤，則讀取 lParam 低位元組，檢查到底是發生了什麼網路事件。可以使用 WSAGETSELECTEVENT 巨集。

WSAGETSELECTERROR 和 WSAGETSELECTEVENT 巨集如下：

```
#define WSAGETSELECTEVENT(lParam)        LOWORD(lParam)
#define WSAGETSELECTERROR(lParam)        HIWORD(lParam)
```

7.2.3 建立視窗

利用 WSAAsyncSelect()函式開發 Windows Sockets 應用程式，依賴於 Windows 視窗。在視窗程序中接收使用者自定義訊息。視窗程序 hWnd 參數為視窗控制碼。

控制碼是用來區分各種記憶體物件的唯一標誌，是個 32 位元整數。視窗控制碼是整個系統唯一的，是 Windows 系統對一個視窗的標示。透過對控制碼的操作來完成與控制碼對應的系統資源的操作。例如，應用程式擁有一個視窗控制碼 hwnd，希望將視窗調整到左上角為(0,0)，右下角為(200,200)的位置。應用程式就可以使用該視窗控制碼為參數呼叫 MoveWindow()函式，將該視窗移動到指定位置程式如下：

```
MoveWindow(hwnd, 0, 0, 200, 200, TRUE);
```

在應用程式中，透過呼叫 Windows API CreateWindow()函式建立視窗，取得視窗控制碼。該函式原形如下：

```
HWND CreateWindow(
    LPCTSTR lpClassName,          //註冊視窗類名的指標
    LPCTSTR lpWindowName,         //視窗的名字
    DWORD dwStyle,                //視窗風格
    int x,                        //水平位置
    int y,                        //垂直位置
    int nWidth,                   //視窗寬度
    int nHeight,                  //視窗高度
    HWND hWndParent,              //父視窗或視窗的所有者控制碼
    HMENU hMenu,                  //功能表控制碼或子視窗控制碼
    HANDLE hInstance,             //應用行程控制碼
    LPVOID lpParam                //建立視窗的資料指標
);
```

- lpClassName：註冊視窗類名。在呼叫該函式前必須呼叫 RegisterClassEx()函式註冊視窗類。
- lpWindowName：應用程式為該視窗定義的視窗名稱。
- dwStyle：指明視窗風格。如 WS_OVERLAPPEDWINDOW 疊層視窗風格。
- x：視窗左上角水平位置。
- y：視窗左上角垂直位置
- nWidth：視窗寬度。
- nHeight：視窗高度。
- hWndParent：建立視窗的父視窗或視窗所有者的視窗控制碼。
- hMenu：功能表控制碼或子視窗控制碼。
- hInstance：應用行程控制碼。
- lpParam：建立視窗的資料指標。

在呼叫 CreateWindow()函式建立視窗前，要呼叫 RegisterClassEx()函式註冊視窗類。該函式原形如下：

```
ATOM RegisterClassEx(
    CONST WNDCLASSEX *lpwcx           //指向 WNDCLASSEX 結構的指標
);
```

如果註冊視窗類成功，則回傳一個 ATOM 類型值。該值唯一地標示了一個已註冊視窗類。如果該函式呼叫失敗則回傳值 0。該函式參數為指向 WNDCLASSEX 結構指標。WNDCLASSEX 原形如下：

```
typedef struct_WNDCLASSEX{
    UINT cbSize;                        //結構大小
    UINT style;                         //風格
    WNDPROC     lpfnWndProc;            //視窗程序
    int         cbClsExtra;             //爲視窗類結構額外分配的位元組數
    int         cbWndExtra;             //爲視窗實體額外分配的位元組數
    HANDLE      hInstance;              //應用程式的實體控制碼
    HICON       hIcon;                  //圖示
    HCURSOR     hCursor;                //游標控制碼
    HBRUSH      hbrBackground;          //畫視窗背景的刷子控制碼
    LPCTSTR     lpszMenuName;           //功能表資源名稱
    LPCTSTR     lpszClassName;          //視窗類名稱
    HICON       hIconSm;                //小圖示控制碼
} WNDCLASSEX;
```

- cbSize：WNDCLASSEX 結構的大小。
- style：該視窗類的風格。如 CS_HREDRAW 風格。當使用者移動或者調整視窗客戶區寬度時，重畫整個視窗。
- pfnWndProc：指向視窗程序的指標。
- cbClsExtra：為視窗類結構額外分配的位元組數。
- cbWndExtra：為視窗實體額外分配的位元組數。
- hInstance：應用程式的實體控制碼。
- hIcon：該視窗類所用圖示。
- hCursor：該視窗類所用的游標控制碼。
- hbrBackground：畫視窗背景的刷子控制碼。
- lpszMenuName：功能表資源名稱。可以使用 MAKEINTRESOURCE 巨集指定資源 ID 載入功能表資源。
- lpszClassName：視窗類名稱。
- hIconSm：視窗類的小圖示控制碼。

在該結構中 pfnWndProc 欄位為當網路事件發生時，接收訊息的視窗程序指標。lpszMenuName 欄位與呼叫 CreateWindow 函式的 lpClassName 參數相同。

建立 Windows 應用程式時，需要執行下面幾個步驟。

1. 呼叫 RegisterClassEx 函式註冊視窗類，在註冊視窗類時指明視窗程序。
2. CreateWindow()函式建立視窗。
3. 呼叫 ShowWindow()函式顯示視窗，呼叫 UpdateWindow()函式更新視窗。
4. 以 GetMessage()函式回傳值作為 while 迴圈條件，不斷接收 Windows 訊息。

在迴圈內呼叫 TranslateMessage()函式和 DispatchMessage()函式。GetMessage()函式接收到的訊息被 TranslateMessage()函式轉譯。該函式對訊息的處理分為兩種情況。一種情況是轉譯後的訊息被投遞到該執行緒訊息佇列中。另一種情況是呼叫 DispatchMessage()函式將訊息傳送給視窗程序。當 GetMessage()函式接收到 WM_QUIT 訊息時，while 迴圈結束，程式退出。

下面程式示範如何建立一個 Windows 應用程式：

```
//視窗程序
LRESULT CALLBACK MainWndProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam);

//程式入口
int WINAPI WinMain(HINSTANCE hinstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int
nCmdShow)
{
    WNDCLASSEX        wcx;

    //填充 WNDCLASSEX 結構
    wcx.cbSize = sizeof(wcx);
    wcx.style = CS_HREDRAW | CS_VREDRAW;                    //風格
    wcx.lpfnWndProc = MainWndProc;                          //視窗程序
    wcx.cbClsExtra = 0;
    wcx.cbWndExtra = 0;
    wcx.hInstance = hinstance;                              //應用程式實體控制碼
    wcx.hIcon = LoadIcon(NULL,IDI_APPLICATION);             //加載圖示資源
    wcx.hCursor = LoadCursor(NULL, IDC_ARROW);              //加載游標資源
    wcx.hbrBackground = GetStockObject(WHITE_BRUSH);        //取得筆刷
    wcx.lpszMenuName =  "MainMenu";                         //功能表名稱
    wcx.lpszClassName = "MainWClass";                       //視窗類名稱
    wcx.hIconSm = LoadImage(hinstance,                      //小圖示
        MAKEINTRESOURCE(5),
        GetSystemMetrics(SM_CXSMICON),
        GetSystemMetrics(SM_CYSMICON),
        LR_DEFAULTCOLOR);

    //註冊視窗類
    RegisterClassEx(&wcx);

    //建立視窗
    hwnd = CreateWindow(
        "MainWClass",            //視窗類名稱
        "Sample",                //視窗名稱
        WS_OVERLAPPEDWINDOW,     //疊層視窗
        CW_USEDEFAULT,           //預設水平位置
        CW_USEDEFAULT,           //預設垂直位置
        CW_USEDEFAULT,           //預設寬度
        CW_USEDEFAULT,           //預設高度
        (HWND) NULL,             //沒有父視窗
        (HMENU) NULL,            //使用視窗類功能表
        hinstance,               //應用程式實體控制碼
        (LPVOID) NULL);          // NULL
```

```
    if (!hwnd)
        return FALSE;

    //顯示視窗
    ShowWindow(hwnd, nCmdShow);
    UpdateWindow(hwnd);

    //訊息迴圈
    while (GetMessage(&msg, (HWND) NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}
```

視窗程序範例程式如下：

```
LRESULT CALLBACK MainWndProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch(uMsg)
    {
    case WM_PAINT:
        {
            //處理訊息
            return 0;
        }
    case WM_DESTROY:
        {
            //處理訊息
            return 0;
        }

        //處理訊息
    }
    return (DefWindowProc(hwnd, uMsg, wParam, lParam));//預設訊息處理
}
```

7.3 WSAAsyncSelect 模型範例程式

下面講解一個伺服器程式。該程式是 Win32 Application。在該程式中使用 WSAAsync Select 模型管理接受的客戶端 Socket。該程式是範例程式，忽略了許多細節。程式設計如圖 7.2 所示，按照下面步驟編碼。

1. 定義自定義訊息。在程式中定義自定義訊息 WM_SOCKET。
2. 定義視窗程序。
3. 呼叫 MyRegisterClass()函式註冊視窗類。

4. 呼叫 InitInstance()函式建立並顯示視窗。因為 WSAAsyncSelect()函式的第一個參數是視窗控制碼，所以要在呼叫該函式之前建立視窗。
5. 初始化 Socket DLL，建立 Socket。
6. 呼叫 WSAAsyncSelect()函式註冊感興趣網路事件。該範例程式中，伺服器監聽 Socket 感興趣的網路事件有 FD_ACCEPT 和 FD_CLOSE。
7. 綁定 Socket，開始監聽。
8. 訊息迴圈。
9. 釋放 Socket 和之前配置的其他資源。

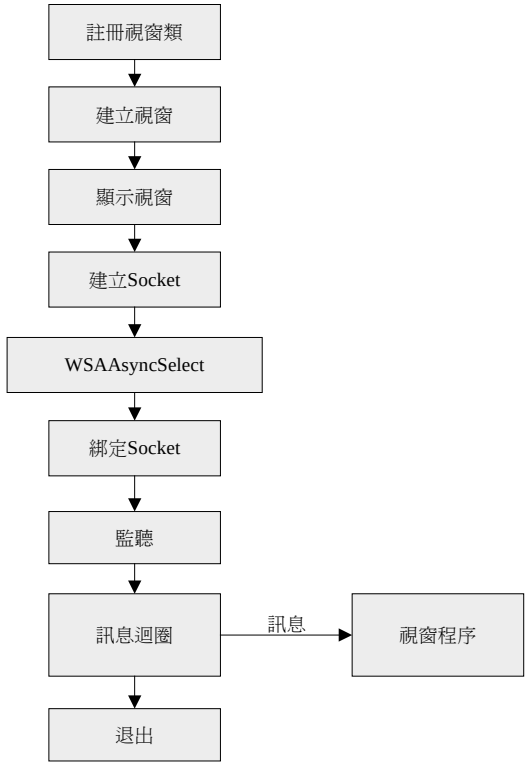

圖 7.2 WSAAsyncSelect 模型範例程式

7.3.1 定義自定義訊息

在應用程式中，通常要定義一個比 WM_USER 值要大的自定義訊息，以免與 Windows 定義的訊息衝突。除了定義這些訊息外，在範例程式中還要定義最大字串長度、伺服器埠、資料緩衝區長度。

程式如下：

```
#define     MAX_LOADSTRING    100              //最大字串長度
#define     WM_SOCKET         WM_USER+1        //Socket 訊息
#define     PORT              5150             //伺服器埠
#define     MAX_SIZE_BUF      1024             //資料緩衝區長度
```

7.3.2 定義視窗程序

視窗程序是由 Windows 系統呼叫的函式，通常將該函式的定義放在主函式之後，將宣告放在主函式之前。在範例程式中為了使主程序結構清晰，將註冊視窗類、建立和顯示視窗的過程都設計為函式，並提前定義。定義 HandleSocketMsg()函式用於對 Windows 網路事件訊息進行處理。

程式如下：

```
ATOM                MyRegisterClass(HINSTANCE hInstance);                //註冊視窗
BOOL                InitInstance(HINSTANCE, int);                        //初始化實體
LRESULT CALLBACK    WndProc(HWND, UINT, WPARAM, LPARAM);                 //視窗程序
void                HandleSocketMsg(WPARAM wParam, LPARAM lParam);       //處理 WM_SOCKET 訊
息
```

7.3.3 註冊視窗類

呼叫 MyRegisterClass()函式註冊視窗類。

```
    //註冊視窗類
    MyRegisterClass(hInstance);
```

7.3.4 建立和顯示視窗

呼叫 InitInstance ()函式建立、顯示視窗。此時，視窗程序開始接收 Windows 訊息。

```
    //建立視窗，顯示視窗
    if (!InitInstance (hInstance, nCmdShow))
    {
        return FALSE;
    }
```

7.3.5 建立 Socket

呼叫 WSAStartup()函式初始化 Socket DLL，呼叫 socket()函式建立 Socket。

```
int Ret;                    //回傳值
WSADATA wsaData;            //WSADATA 結構變數
//請求 Winodows Sockets 2.2 版本
if ((Ret = WSAStartup(0x0202, &wsaData)) != 0)
{
    return 0;
}
//建立 Socket
if ((sListen = socket (PF_INET, SOCK_STREAM, 0)) == INVALID_SOCKET)
{
    return 0;
}
```

7.3.6 註冊感興趣的網路事件

以視窗控制碼 hWnd 和 WM_SOCKET 為第 2、第 3 個參數呼叫 WSAAsyncSelect()函式。同時註冊 FD_ACCEPT 和 FD_CLOSE 網路事件。請求系統當 FD_ACCEPT 和 FD_CLOSE 網路事件發生時，給 hWnd 視窗傳送 WM_SOCKET 訊息。

```
WSAAsyncSelect(sListen, hWnd, WM_SOCKET, FD_ACCEPT|FD_CLOSE);
```

7.3.7 綁定 Socket

呼叫 bind()函式綁定 Socket。

```
SOCKADDR_IN InternetAddr;
InternetAddr.sin_family = AF_INET;
InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
InternetAddr.sin_port = htons(PORT);
//綁定 Socket
if (bind(sListen, (PSOCKADDR) &InternetAddr, sizeof(InternetAddr)) == SOCKET_ERROR)
{
    return 0;
}
```

7.3.8 開始監聽

呼叫 listen()函式 Socket 開始監聽。

```
//監聽
if (listen(sListen, SOMAXCONN))
{
    return 0;
}
```

7.3.9 訊息迴圈

在 while 迴圈語句中，GetMessage()函式不斷從執行緒訊息佇列中取出訊息。當 FD_ACCEPT 或者 FD_CLOSE 網路事件發生時，WM_SOCKET 訊息被投遞到執行緒訊息佇列中，GetMessage()函式負責將該訊息從執行緒訊息佇列中取出，DispatchMessage()函式再將該訊息傳送到視窗程序。

```
//主訊息迴圈
MSG msg;
while (GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
```

7.3.10 程式退出

當 GetMessage()函式接收到 WM_QUIT 訊息時，while 迴圈結束，釋放資源，程式退出。

```
closesocket(sListen);          //釋放 Socket 資源
WSACleanup();
DeleteAllNode();               //釋放節點空間
```

7.3.11 視窗程序

當建立視窗成功後 WndProc()視窗程序便開始接收 Windows 訊息。在該函式中需要處理許多訊息。例如，當關閉視窗時傳送 WM_DESTROY 訊息，在視窗程序中呼叫 PostQuitMessage()函式向執行緒訊息佇列投遞 WM_QUIT 訊息。GetMessage()函式接收到該訊息後，程式退出。

應用程式不感興趣的訊息交給 DefWindowProc()函式處理。

當 FD_ACCEPT 或者 FD_CLOSE 網路事件發生時，視窗程序接收到 WM_SOCKET 訊息。在視窗程序中呼叫 HandleSocketMsg()函式對觸發 WM_SOCKET 訊息的網路事件進行處理。

該範例的視窗程序程式如下：

```
//視窗程序
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
```

```
    {
    case WM_SOCKET:                                    //網路事件發生時傳送給該視窗的訊息
        {
            HandleSocketMsg(wParam, lParam);           //處理該訊息
            break;
        }
        case WM_PAINT:                                 //畫客戶區
            //
            break;
        case WM_DESTROY:                               //程式退出
            PostQuitMessage(0);
            break;
            //訊息處理
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

7.3.12 CClient 類別

在程式中定義 CClient 類別管理伺服器接受客戶端的新建 Socket。該類別建構函式的參數為伺服器接受客戶端的新建 Socket。在解構函式中將該 Socket 關閉。

在該類別中宣告 RecvData()函式接收資料，SendData()函式傳送資料，GetSocket()函式回傳 Socket。CClient 類別定義如下：

```
#define MAX_SIZE_BUF        1024          //資料緩衝區長度
class CClient
{
public: CClient(SOCKET s);                //建構函式
        virtual ~CClient();               //解構函式
public:
    void RecvData(void);                  //接收資料
    void SendData(void);                  //傳送資料
    SOCKET GetSocket(void);               //取得 Socket

private:
    SOCKET    m_s;                        //Socket
    char        m_recvBuf[MAX_SIZE_BUF];  //接收資料緩衝區
    char        m_sendBuf[MAX_SIZE_BUF];  //傳送資料緩衝區
};
```

7.3.13 管理客戶端 Socket 的鏈表

定義_socktnode 結構。該結構 pClient 欄位為 CClient 類別指標。pNext 變數為指向下一個節點的指標。該結構定義如下：

```
typedef struct _socktnode{
    CClient        *pClient;                    //CClient 類別指標
    _socktnode    *pNext;                       //指向下一個節點
}SOCKETNODE, *PSOCKETNODE;
```

當伺服器接受一個客戶端連線請求後，建立一個 CClient 實體，新建一個 SOCKETNODE 節點。將實體指標賦值給 SOCKETNODE 結構的 pClient 變數。

為了對鏈表進行操作，宣告如下函式。

- AddNode()函式：新增節點。
- DeleteNode()函式：刪除節點。
- GetClient()函式：取得 CClient 類別指標。
- DeleteAllNode()函式：刪除所有節點。

```
PSOCKETNODE    HeaderSocktNode = NULL;          //節點頭
void    AddNode(SOCKET s);                      //增加結點
void    DeleteNode(SOCKET s);                   //刪除結點
void    DeleteAllNode(void);                    //刪除所有結點
CClient*GetClient(SOCKET s);                    //CClient 類別指標
```

7.3.14 網路事件訊息處理函式

在 HandleSocketMsg()函式中呼叫 WSAGETSELECTERROR 巨集檢查是否有網路錯誤事件發生。如果有網路錯誤事件發生則呼叫 DeleteNode()函式將該 Socket 從鏈表中刪除。在前面的講解中，已經知道 wParam 參數為發生網路事件的 Socket。所以，以 wParam 為參數呼叫 DeleteNode()函式。

如果沒有網路錯誤事件發生，則呼叫 WSAGETSELECTEVENT 巨集，檢查發生了什麼網路事件。

如果網路事件為 FD_ACCEPT，那麼說明此時客戶端等待伺服器接受連線請求。發生這個網路事件的 Socket 一定是伺服器的監聽 Socket。呼叫 accept()函式接受客戶端的連線請求，將該 Socket 加入鏈表中，然後以該新建 Socket 作為參數呼叫 WSAAsyncSelect()函式，為該 Socket 請求 FD_READ、FD_WRITE 和 FD_CLOSE 網路事件。

當 HandleSocketMsg()函式接收到 FD_READ 網路事件時，說明此時在伺服器接受的客戶端 Socket 中，某個 Socket 上存在可讀的資料。這個 Socket 就是 wParam 參數值。呼叫 GetClient()函式得到儲存該 Socket 的 CClient 類別指標。呼叫該類別的 RecvData()函式接收客戶端的資料。

HandleSocketMsg()函式接收到 FD_WRITE 網路事件時的處理方法，同收到 FD_READ 網路事件時的處理方法相似。

當該函式接收到 FD_CLOSE 網路事件時，說明此時客戶端關閉了 Socket，可以呼叫 DeleteNode()函式刪除該客戶端節點。

HandleSocketMsg()函式範例程式如下：

```
//WM_SOCKET 訊息處理
void HandleSocketMsg(WPARAM wParam, LPARAM lParam)
{
    if (WSAGETSELECTERROR(lParam))              //檢查網路錯誤
    {
        DeleteNode(wParam);                     //刪除節點，關閉 Socket
    }
    else
    {
        switch(WSAGETSELECTEVENT(lParam))       //檢查網路事件
        {
        case FD_ACCEPT:                         //接受客戶端連線請求
            {
                SOCKET sAccept;
                if ((sAccept = accept(wParam, NULL, NULL)) == INVALID_SOCKET)
                {
                    break;
                }
                AddNode(sAccept);               //將 Socket 加入鏈表中

                //FD_READ、FD_WRITE 和 FD_CLOSE 網路事件發生時，傳送 WM_SOCKET 訊息
                WSAAsyncSelect(sAccept, hWnd, WM_SOCKET, FD_READ|FD_WRITE|FD_CLOSE);
                break;
            }
        case FD_READ:                           //接收資料
            {
                CClient* pClient = GetClient(wParam);       //根據 Socket，取得客戶端節點
                pClient->RecvData();            //接收資料
                break;
            }
        case FD_WRITE:                          //傳送資料
            {
                CClient* pClient = GetClient(wParam);       //根據 Socket，取得客戶端節點
                pClient->SendData();            //傳送資料
                break;
            }
        case FD_CLOSE:                          //對方關閉了 Socket 連線
            {
```

```
                DeleteNode(wParam);             //刪除節點
                break;
            }
        }
    }
    return;
}
```

7.4 呼叫 WSAAsyncSelect()函式注意問題

使用 WSAAsyncSelect()函式開發 Windows Sockets 應用程式中，開發人員需要注意以下問題。

7.4.1 接收不到網路事件

第一種情況是由於在同一個 Socket 同一個自定義訊息上，多次呼叫 WSAAsyncSelect()函式註冊不同的網路事件，最後一次函式呼叫取消了前面註冊的網路事件。例如，在應用程式中，第一次呼叫 WSAAsyncSelect()函式註冊 FD_READ 網路事件，然後又呼叫該函式註冊 FD_WRITE 網路事件，那麼此時應用程式，就只能接收到 FD_WRITE 網路事件。

如果要取消所有請求的網路事件通知，告知 Windows Sockets 實作不再為該 Socket 傳送任何網路事件相關的訊息，要以參數 lEvent 值為 0 呼叫 WSAAsyncSelect()函式。

```
WSAAsyncSelect(s, hWnd, 0, 0);
```

需要注意儘管應用程式呼叫上述函式取消了網路事件通知，但是在應用程式訊息佇列中，可能還有網路訊息在排隊。所以在呼叫 WSAAsyncSelect()函式取消網路事件訊息後，應用程式還應該繼續準備接收網路事件。

第二種情況是在同一個 Socket 上，多次呼叫 WSAAsyncSelect()函式，為不同的網路事件定義了不同的訊息，最後一次該函式呼叫將取消前面註冊的網路事件。

下面的程式碼中，第二次函式呼叫將會取消第一次函式呼叫的作用。只有 FD_WRITE 網路事件透過 wMsg2 訊息通知到視窗。

```
WSAAsyncSelect(s, hWnd, wMsg1, FD_READ);
WSAAsyncSelect(s, hWnd, wMsg2, FD_WRITE);
```

7.4.2 關於 accept()函式

因為呼叫 accept()函式接受的 Socket 和監聽 Socket 具有同樣的屬性。所以，任何為監聽 Socket 設置的網路事件對接受的 Socket 同樣起作用。如果一個監聽 Socket 請求 FD_ACCEPT、FD_READ 和 FD_WRITE 網路事件，則在該監聽 Socket 上接受的任何 Socket 也會請求 FD_ ACCEPT、FD_READ 和 FD_WRITE 網路事件，以及傳送同樣的訊息。

若需要不同的訊息和網路事件，應用程式應該呼叫 WSAAsyncSelect()函式，為該 Socket 請求不同的網路事件和訊息。

7.4.3 關於 FD_READ 網路事件

一個 FD_READ 網路事件不要多次呼叫 recv()函式。如果應用程式為一個 FD_READ 網路事件，呼叫了多個 recv()函式，會使得該應用程式接收到多個 FD_READ 網路事件。

如果在一次接收 FD_READ 網路事件時需要呼叫多次 recv()函式，應用程式應該在呼叫 recv()函式之前關閉 FD_READ 訊息。

應用程式不必在收到 FD_READ 訊息時，讀進所有可讀的資料。每接收到一次 FD_READ 網路事件，應用程式呼叫一次 recv()函式是恰當的。

7.4.4 如何判斷 Socket 已經關閉

要使用 FD_CLOSE 網路事件來判斷 Socket 是否已經關閉。

接收 FD_CLOSE 網路事件時，錯誤代碼指示出 Socket 是緩衝關閉還是強制關閉。如果錯誤代碼 0，則為緩衝關閉；若錯誤代碼為 WSAECONNRESET，則 Socket 是強制關閉。

如果 Socket 緩衝關閉，資料已經都全部接收，應用程式只會收到 FD_CLOSE 訊息來指出虛擬電路(Virtual Circuit)關閉，它不會收到 FD_READ 訊息來表明這種狀況。呼叫 closesocket()函式後不會投遞 FD_CLOSE 網路事件。