Socket模型

选择模型

简介

在一般的网络编程中，例如：`int bytes = recv(s, buffer, 1024);`中此函数会阻塞，直到 套接字连接上有数据可读，把数据读到**buffer**里后函数才会返回， 在单线程的程序里出现这种情况会导致主线程。

而在利用`ioctlsocket`函数设置为非阻塞模式后

```
ioctlsocket(s, FIOBIO, (unsigned long *)&ul);
int bytes = recv(s, buffer, 1024);
```

不管套接字连接上有没有数据可以接收都会马上返回。多个客户端连接需要**轮询**检查`recv`函数的返回值，以确定是否有数据到来。

`select`函数可以代替我们完成这件工作。

```
int select(
    int nfds,
    fd_set FAR *readfds,     // 检查套接字可读性
    fd_set FAR *writefds,    // 检查套接字可写性
    fd_set FAR *exceptfds,
    const struct timeval FAR *timeout    // 超时时间
);


struct timeval {
    long tv_sec; // seconds
    long tv_usec; // and microseconds
};

// fd_set是一个集合，可添加多个套接字


FD_ZERO(fd_set *fdset);              // 清空fdset与所有文件句柄的联系。
FD_SET(int fd, fd_set *fdset);       // 建立文件句柄fd与fdset的联系。
FD_CLR(int fd, fd_set *fdset);       // 清除文件句柄fd与fdset的联系。
FD_ISSET(int fd, fd_set *fdset);     // 检查fdset联系的文件句柄fd是否
```

例子

```
#include <ws2tcpip.h>
#include <stdio.h>
#pragma comment(lib, "ws2_32.lib")

#define PORT        5566
```

```
#define MSGSIZE    1024

int    g_iTotalConn = 0;
SOCKET g_CliSocketArr[FD_SETSIZE];

DWORD WINAPI WorkerThread(LPVOID lpParameter);

int main()
{
    WSADATA     wsaData;
    SOCKET      sListen, sClient;
    SOCKADDR_IN local, client;
    int         iaddrSize = sizeof(SOCKADDR_IN);
    DWORD       dwThreadId;

    // Initialize Windows socket library
    WSAStartup(0x0202, &wsaData);

    // Create listening socket
    sListen = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

    // Bind
    local.sin_addr.S_un.S_addr = htonl(INADDR_ANY);
    local.sin_family = AF_INET;
    local.sin_port = htons(PORT);
    bind(sListen, (struct sockaddr*) & local, sizeof(SOCKADDR_IN));

    // Listen
    listen(sListen, 3);

    // Create worker thread
    CreateThread(NULL, 0, WorkerThread, NULL, 0, &dwThreadId);

    while (TRUE)
    {
        //Accept a connection
        sClient = accept(sListen, (struct sockaddr*) &client, &iaddrSize);
        printf("Accepted client:%s:%d\n", inet_ntoa(client.sin_addr),
ntohs(client.sin_port));

        // Add socket to g_CliSocketArr
        g_CliSocketArr[g_iTotalConn++] = sClient;
    }

    closesocket(sListen);
    return 0;
}



DWORD WINAPI WorkerThread(LPVOID lpParam)
{
    int            i;
    fd_set         fdread;
    int            ret;
    struct timeval tv = { 1, 0 };
    char           szMessage[MSGSIZE];
```

```
    while (TRUE)
    {
        FD_ZERO(&fdread);//将fdread初始化空集
        for (i = 0; i < g_iTotalConn; i++)
        {
            FD_SET(g_CliSocketArr[i], &fdread);//将要检查的套接口加入到集合中
        }

        // We only care read event
        ret = select(0, &fdread, NULL, NULL, &tv);//每隔一段时间，检查可读性的套接口
        if (ret <= 0)
        {
            continue; //超时
        }

        for (i = 0; i < g_iTotalConn; i++)
        {
            if (FD_ISSET(g_CliSocketArr[i], &fdread))//如果可读
            {
                // A read event happened on g_CliSocketArr
                ret = recv(g_CliSocketArr[i], szMessage, MSGSIZE, 0);
                if (ret == 0 || (ret == SOCKET_ERROR && WSAGetLastError() ==
WSAECONNRESET))
                {
                    // Client socket closed
                    printf("Client socket %d closed.\n", g_CliSocketArr[i]);
                    closesocket(g_CliSocketArr[i]);

                    if (i < g_iTotalConn - 1)
                    {
                        g_CliSocketArr[i--] = g_CliSocketArr[--g_iTotalConn];
                    }
                }
                else
                {
                    //We received a message from client
                    printf("bytes:%d msg:%s\n", ret, szMessage);
                }
            }
        }
    }
    return 0;
}
```

异步选择模型

简介

异步选择（**WSAAsyncSelect**）模型是一个有用的异步 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。该模型的核心即是 `WSAAsyncSelect` 函数。

```
int WSAAsyncSelect(
  SOCKET s,
```

```
  HWND   hwnd,  // 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
  u_int  wMsg,  // 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
  long   lEvent // 位掩码 FD_READ、FD_WRITE、FD_ACCEPT、FD_CONNECT、FD_CLOSE等
);


LRESULT CALLBACK WindowProc(
    HWND hwnd,  //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。
    UINT uMsg,  //指定需要对哪些消息进行处理。这里我们感兴趣的是WSAAsyncSelect调用中定义的消息。
    WPARAM wParam,   //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。)
    LPARAM lParam      //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。
);
```

例子

```
#define WM_SOCKET WM_USER+0

LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    WSADATA       wsd;
    static SOCKET sListen;
    SOCKET        sClient;
    SOCKADDR_IN   local, client;
    int           ret, iAddrSize = sizeof(client);
    char          szMessage[MSGSIZE];

    switch (message)
    {
    case WM_CREATE:
        // Initialize Windows Socket library
        WSAStartup(0x0202, &wsd);

        // Create listening socket
        sListen = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

        // Bind
        local.sin_addr.S_un.S_addr = htonl(INADDR_ANY);
        local.sin_family = AF_INET;
        local.sin_port = htons(PORT);
        bind(sListen, (struct sockaddr*) & local, sizeof(local));

        // Listen
        listen(sListen, 3);

        //为服务器socket注册FD_ACCEPT消息
        WSAAsyncSelect(sListen, hwnd, WM_SOCKET, FD_ACCEPT);
        return 0;
    case WM_DESTROY:
        closesocket(sListen);
        WSACleanup();
        PostQuitMessage(0);
```

```
        return 0;
    case WM_SOCKET:
        if (WSAGETSELECTERROR(lParam))
        {
            closesocket(wParam); //socket出错
            break;
        }
        switch (WSAGETSELECTEVENT(lParam))//取低位字节，网络事件
        {
        case FD_ACCEPT:
            //接受客户端连接
            sClient = accept(wParam, (struct sockaddr*) & client, &iAddrSize);

            //为客户端注册FD_READ、FD_CLOSE消息
            WSAAsyncSelect(sClient, hwnd, WM_SOCKET, FD_READ | FD_CLOSE);
            break;
        case FD_READ:
            ret = recv(wParam, szMessage, MSGSIZE, 0);
            if (ret == 0 || ret == SOCKET_ERROR && WSAGetLastError() ==
WSAECONNRESET)
            {
                closesocket(wParam);
            }
            else
            {
                szMessage[ret] = '\0';
                MessageBox(NULL, szMessage, NULL, MB_OK);
            }
            break;
        case FD_CLOSE:
            closesocket(wParam);
            break;
        }
        return 0;
    }
    return DefWindowProc(hwnd, message, wParam, lParam);
}
```

事件选择模型

简介

事件选择（**WSAEventSelect**）模型是另一个有用的异步 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，最主要的差别在于网络事件会投递至一个事件对象句柄，而非投递到一个窗口例程。

事件通知模型要求我们的应用程序针对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

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

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

```
int WSAAPI WSAEventSelect(
  SOCKET   s,
  WSAEVENT hEventObject,    //指定要与套接字关联在一起的事件对象,即用 WSACreateEvent 创建的那一个
  long     lNetworkEvents   //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
);
```

`WSACreateEvent` 创建的事件有两种工作状态，以及两种工作模式。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` 开始是在一种未传信的工作状态，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 WSAAPI WSAResetEvent(WSAEVENT hEvent);
```

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 WSAAPI WSACloseEvent(WSAEVENT hEvent);
```

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

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

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

```
int WSAAPI WSAEnumNetworkEvents(
  SOCKET             s,
  WSAEVENT           hEventObject,  // 可选，由于事件对象处在一个“已传信”状态，令其成为“未传信”状态
  LPWSANETWORKEVENTS lpNetworkEvents    // 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
);

typedef struct _WSANETWORKEVENTS {
    long lNetworkEvents;            // 对应于套接字上发生的所有网络事件类型
    int iErrorCode[FD_MAX_EVENTS];  // 错误代码数组
} WSANETWORKEVENTS, *LPWSANETWORKEVENTS;
```

例子

```
#define PORT    5566
#define MSGSIZE 1024

int      g_iTotalConn = 0;
SOCKET   g_CliSocketArr[MAXIMUM_WAIT_OBJECTS];
WSAEVENT g_CliEventArr[MAXIMUM_WAIT_OBJECTS];

DWORD WINAPI WorkerThread(LPVOID);

void Cleanup(int index);

int main()
{
    WSADATA     wsaData;
    SOCKET      sListen, sClient;
    SOCKADDR_IN local, client;
    DWORD       dwThreadId;
    int         iaddrSize = sizeof(SOCKADDR_IN);

    // Initialize Windows Socket library
    WSAStartup(0x0202, &wsaData);

    // Create listening socket
    sListen = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

    // Bind
    local.sin_addr.S_un.S_addr = htonl(INADDR_ANY);
    local.sin_family = AF_INET;
    local.sin_port = htons(PORT);
    bind(sListen, (struct sockaddr*) & local, sizeof(SOCKADDR_IN));

    // Listen
    listen(sListen, 3);

    // Create worker thread
    CreateThread(NULL, 0, WorkerThread, NULL, 0, &dwThreadId);

    while (TRUE)
    {
        // Accept a connection
        sClient = accept(sListen, (struct sockaddr*) & client, &iaddrSize);
        printf("Accepted client:%s:%d\n", inet_ntoa(client.sin_addr),
ntohs(client.sin_port));

        // Associate socket with network event
        g_CliSocketArr[g_iTotalConn] = sClient;//接受连接的套接口
        g_CliEventArr[g_iTotalConn] = WSACreateEvent();//返回事件对象句柄

        //在套接口上将一个或多个网络事件与 事件对象关联在一起
        WSAEventSelect(g_CliSocketArr[g_iTotalConn],//套接口
            g_CliEventArr[g_iTotalConn],//事件对象
            FD_READ | FD_CLOSE);//网络事件
        g_iTotalConn++;
```

```
    }
}

DWORD WINAPI WorkerThread(LPVOID lpParam)
{
    int              ret, index;
    WSANETWORKEVENTS NetworkEvents;
    char             szMessage[MSGSIZE];

    while (TRUE)
    {
        //返回导致返回的事件对象
        ret = WSAWaitForMultipleEvents(g_iTotalConn,//数组中的句柄数目
            g_CliEventArr,//指向一个事件对象句柄数组的指针
            FALSE, //T, 都进才回; F, 一进就回
            1000, //超时间隔
            FALSE);//是否执行完成例程

        if (ret == WSA_WAIT_FAILED || ret == WSA_WAIT_TIMEOUT)
        {
            continue;
        }

        index = ret - WSA_WAIT_EVENT_0;

        //在套接口上查询与事件对象关联的网络事件
        WSAEnumNetworkEvents(g_CliSocketArr[index], g_CliEventArr[index],
&NetworkEvents);

        //处理FD-READ网络事件
        if (NetworkEvents.lNetworkEvents & FD_READ)
        {
            // Receive message from client
            ret = recv(g_CliSocketArr[index], szMessage, MSGSIZE, 0);
            if (ret == 0 || (ret == SOCKET_ERROR && WSAGetLastError() ==
WSAECONNRESET))
            {
                Cleanup(index);
            }
            else
            {
                szMessage[ret] = '\0';
                printf("bytes:%d msg:%s\n", ret, szMessage);
            }
        }

        //处理FD-CLOSE网络事件
        if (NetworkEvents.lNetworkEvents & FD_CLOSE)
        {
            Cleanup(index);
        }
    }
    return 0;
}

void Cleanup(int index)
{
    closesocket(g_CliSocketArr[index]);
```

```
    WSACloseEvent(g_CliEventArr[index]);

    if (index < g_iTotalConn - 1)
    {
        g_CliSocketArr[index] = g_CliSocketArr[g_iTotalConn - 1];
        g_CliEventArr[index] = g_CliEventArr[g_iTotalConn - 1];
    }
    g_iTotalConn--;
}
```

重叠模型

简介

重叠模型是让应用程序使用重叠数据结构(**WSAOVERLAPPED**)，一次投递一个或多个Winsock I/O请求。针对这些提交的请求，在它们完成之后，应用程序会收到通知

有两个方法可以用来管理重叠IO请求的完成情况（就是说接到重叠操作完成的通知）：

1. 事件对象通知(event object notification)
2. 完成例程(completion routines)

要使用重叠结构，我们常用的`send，sendto，recv，recvfrom`也都要被`WSASend，WSASendto，WSARecv，WSARecvFrom`替换掉

`WSAOVERLAPPED`结构如下：

```
typedef struct _WSAOVERLAPPED {
    DWORD Internal;
    DWORD InternalHigh;
    DWORD Offset;
    DWORD OffsetHigh;
    WSAEVENT hEvent;      // 唯一需要关注的参数，用来关联WSAEvent对象
} WSAOVERLAPPED, *LPWSAOVERLAPPED;


// 使用
WSAEVENT event;                   // 定义事件
WSAOVERLAPPED AcceptOverlapped ; // 定义重叠结构
event = WSACreateEvent();         // 建立一个事件对象句柄
ZeroMemory(&AcceptOverlapped, sizeof(WSAOVERLAPPED)); // 初始化重叠结构
AcceptOverlapped.hEvent = event;    // Done !!
```

`WSARecv`系列函数

```
int WSARecv(
    SOCKET s,                      // 当然是投递这个操作的套接字
    LPWSABUF lpBuffers,          // 接收缓冲区，与Recv函数不同，这里需要一个由WSABUF结构构成的数组
    DWORD dwBufferCount,       // 数组中WSABUF结构的数量
    LPDWORD lpNumberOfBytesRecvd,  // 如果接收操作立即完成，这里会返回函数调用所接收到的字节数
    LPDWORD lpFlags,             // 这里设置为0 即可
    LPWSAOVERLAPPED lpOverlapped,  // “绑定”的重叠结构
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine // 完成例程中将会用到的参数，否则设置为 NULL
);
```

完成例程回调函数原型及传递方式

```
Void CALLBACK _CompletionRoutineFunc(
    DWORD dwError,           // 标志咱们投递的重叠操作，比如WSARecv，完成的状态是什么
    DWORD cbTransferred,     // 指明了在重叠操作期间，实际传输的字节量是多大
    LPWSAOVERLAPPED lpOverlapped,   // 参数指明传递到最初的IO调用内的一个重叠  结构
    DWORD dwFlags  // 返回操作结束时可能用的标志(一般没用)
);
```

例子

事件对象通知

```
#define PORT    5566
#define MSGSIZE 1024

typedef struct
{
    WSAOVERLAPPED overlap;
    WSABUF        Buffer;
    char          szMessage[MSGSIZE];
    DWORD         NumberOfBytesRecvd;
    DWORD         Flags;
}PER_IO_OPERATION_DATA, * LPPER_IO_OPERATION_DATA;


int                      g_iTotalConn = 0;
SOCKET                   g_CliSocketArr[MAXIMUM_WAIT_OBJECTS];
WSAEVENT                 g_CliEventArr[MAXIMUM_WAIT_OBJECTS];
LPPER_IO_OPERATION_DATA g_pPerIODataArr[MAXIMUM_WAIT_OBJECTS];

DWORD WINAPI WorkerThread(LPVOID);

void Cleanup(int);

int main()
{
    WSADATA     wsaData;
    SOCKET      sListen, sClient;
```

```
    SOCKADDR_IN local, client;
    DWORD       dwThreadId;
    int         iaddrSize = sizeof(SOCKADDR_IN);

    // Initialize Windows Socket library
    WSAStartup(0x0202, &wsaData);

    // Create listening socket
    sListen = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

    // Bind
    local.sin_addr.S_un.S_addr = htonl(INADDR_ANY);
    local.sin_family = AF_INET;
    local.sin_port = htons(PORT);
    bind(sListen, (struct sockaddr*) & local, sizeof(SOCKADDR_IN));

    // Listen
    listen(sListen, 3);

    // Create worker thread
    CreateThread(NULL, 0, WorkerThread, NULL, 0, &dwThreadId);

    while (TRUE)
    {
        // Accept a connection
        sClient = accept(sListen, (struct sockaddr*) & client, &iaddrSize);
        printf("Accepted client:%s:%d\n", inet_ntoa(client.sin_addr),
ntohs(client.sin_port));
        g_CliSocketArr[g_iTotalConn] = sClient;

        // Allocate a PER_IO_OPERATION_DATA structure
        g_pPerIODataArr[g_iTotalConn] = (LPPER_IO_OPERATION_DATA)HeapAlloc(
            GetProcessHeap(),
            HEAP_ZERO_MEMORY,
            sizeof(PER_IO_OPERATION_DATA));

        g_pPerIODataArr[g_iTotalConn]->Buffer.len = MSGSIZE;
        g_pPerIODataArr[g_iTotalConn]->Buffer.buf =
g_pPerIODataArr[g_iTotalConn]->szMessage;
        g_CliEventArr[g_iTotalConn] = g_pPerIODataArr[g_iTotalConn]-
>overlap.hEvent = WSACreateEvent();

        // Launch an asynchronous operation
        WSARecv(
            g_CliSocketArr[g_iTotalConn],
            &g_pPerIODataArr[g_iTotalConn]->Buffer,
            1,
            &g_pPerIODataArr[g_iTotalConn]->NumberOfBytesRecvd,
            &g_pPerIODataArr[g_iTotalConn]->Flags,
            &g_pPerIODataArr[g_iTotalConn]->overlap,
            NULL);

        g_iTotalConn++;
    }

    closesocket(sListen);
    WSACleanup();
    return 0;
```

```
}


DWORD WINAPI WorkerThread(LPVOID lpParam)
{
    int   ret, index;
    DWORD cbTransferred;

    while (TRUE)
    {
        ret = WSAWaitForMultipleEvents(g_iTotalConn, g_CliEventArr, FALSE, 1000,
FALSE);

        if (ret == WSA_WAIT_FAILED || ret == WSA_WAIT_TIMEOUT)
        {
            continue;
        }

        index = ret - WSA_WAIT_EVENT_0;
        WSAResetEvent(g_CliEventArr[index]);

        WSAGetOverlappedResult(
            g_CliSocketArr[index],
            &g_pPerIODataArr[index]->overlap,
            &cbTransferred,
            TRUE,
            &g_pPerIODataArr[g_iTotalConn]->Flags);

        if (cbTransferred == 0)
        {
            // The connection was closed by client
            Cleanup(index);
        }
        else
        {
            printf("bytes:%d msg:%s\n", cbTransferred, g_pPerIODataArr[index]-
>szMessage);

            WSARecv(
                g_CliSocketArr[index],
                &g_pPerIODataArr[index]->Buffer,
                1,
                &g_pPerIODataArr[index]->NumberOfBytesRecvd,
                &g_pPerIODataArr[index]->Flags,
                &g_pPerIODataArr[index]->overlap,
                NULL);
        }
    }

    return 0;
}


void Cleanup(int index)
{
    closesocket(g_CliSocketArr[index]);
    WSACloseEvent(g_CliEventArr[index]);
```

```
    HeapFree(GetProcessHeap(), 0, g_pPerIODataArr[index]);

    if (index < g_iTotalConn - 1)
    {
        g_CliSocketArr[index] = g_CliSocketArr[g_iTotalConn - 1];
        g_CliEventArr[index] = g_CliEventArr[g_iTotalConn - 1];
        g_pPerIODataArr[index] = g_pPerIODataArr[g_iTotalConn - 1];
    }

    g_pPerIODataArr[--g_iTotalConn] = NULL;
}
```

完成例程

```
#define PORT    5566
#define MSGSIZE 1024

typedef struct
{
    WSAOVERLAPPED overlap;
    WSABUF        Buffer;
    char          szMessage[MSGSIZE];
    DWORD         NumberOfBytesRecvd;
    DWORD         Flags;
    SOCKET        sClient;
}PER_IO_OPERATION_DATA, *LPPER_IO_OPERATION_DATA;

DWORD WINAPI WorkerThread(LPVOID);
void CALLBACK CompletionROUTINE(DWORD, DWORD, LPWSAOVERLAPPED, DWORD);

SOCKET g_sNewClientConnection;
BOOL   g_bNewConnectionArrived = FALSE;

int main()
{
    WSADATA     wsaData;
    SOCKET      sListen;
    SOCKADDR_IN local, client;
    DWORD       dwThreadId;
    int         iaddrSize = sizeof(SOCKADDR_IN);

    // Initialize Windows Socket library
    WSAStartup(0x0202, &wsaData);

    // Create listening socket
    sListen = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

    // Bind
    local.sin_addr.S_un.S_addr = htonl(INADDR_ANY);
    local.sin_family = AF_INET;
    local.sin_port = htons(PORT);
    bind(sListen, (struct sockaddr *)&local, sizeof(SOCKADDR_IN));

    // Listen
    listen(sListen, 3);
```

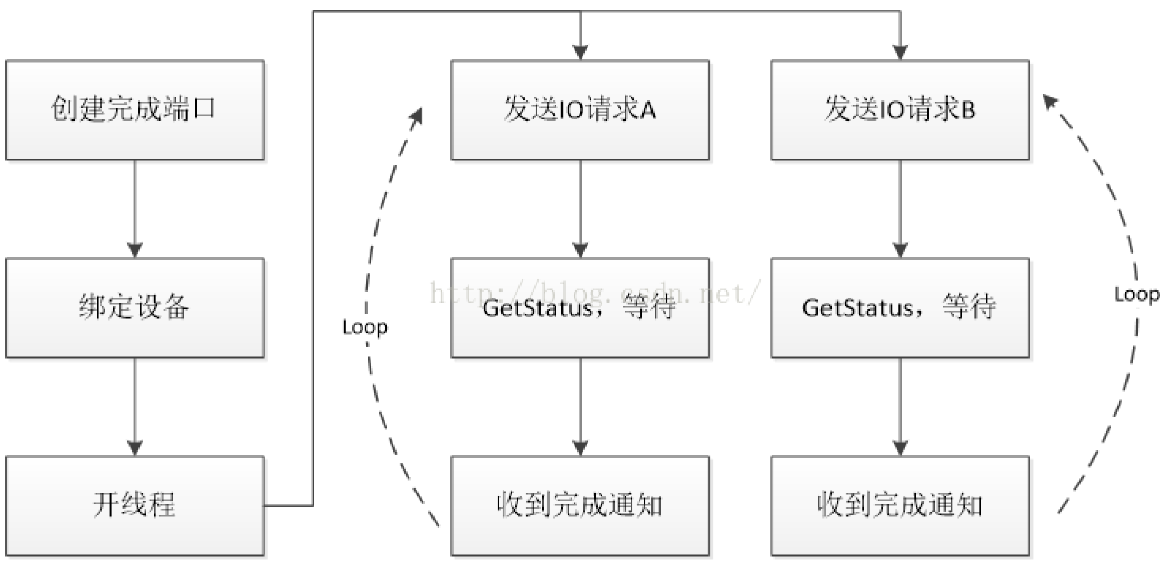
```
    // Create worker thread
    CreateThread(NULL, 0, WorkerThread, NULL, 0, &dwThreadId);

    while (TRUE)
    {
        // Accept a connection
        g_sNewClientConnection = accept(sListen, (struct sockaddr *)&client,
&iaddrSize);
        g_bNewConnectionArrived = TRUE;
        printf("Accepted client:%s:%d\n", inet_ntoa(client.sin_addr),
ntohs(client.sin_port));
    }
}

DWORD WINAPI WorkerThread(LPVOID lpParam)
{
    LPPER_IO_OPERATION_DATA lpPerIOData = NULL;

    while (TRUE)
    {
        if (g_bNewConnectionArrived)
        {

            // Launch an asynchronous operation for new arrived connection
            lpPerIOData = (LPPER_IO_OPERATION_DATA)HeapAlloc(
                GetProcessHeap(),
                HEAP_ZERO_MEMORY,
                sizeof(PER_IO_OPERATION_DATA));
            lpPerIOData->Buffer.len = MSGSIZE;
            lpPerIOData->Buffer.buf = lpPerIOData->szMessage;
            lpPerIOData->sClient = g_sNewClientConnection;

            WSARecv(lpPerIOData->sClient,
                    &lpPerIOData->Buffer,
                    1,
                    &lpPerIOData->NumberOfBytesRecvd,
                    &lpPerIOData->Flags,
                    &lpPerIOData->overlap,
                    CompletionROUTINE);

            g_bNewConnectionArrived = FALSE;
        }

        SleepEx(1000, TRUE);
    }
    return 0;
}



void CALLBACK CompletionROUTINE(DWORD dwError, DWORD cbTransferred,
LPWSAOVERLAPPED lpOverlapped, DWORD dwFlags)
{
    LPPER_IO_OPERATION_DATA lpPerIOData = (LPPER_IO_OPERATION_DATA)lpOverlapped;

    if (dwError != 0 || cbTransferred == 0)
    {
```

```
            // Connection was closed by client
            closesocket(lpPerIOData->sClient);
            HeapFree(GetProcessHeap(), 0, lpPerIOData);
        }
        else
        {
            lpPerIOData->szMessage[cbTransferred] = '\0';
            send(lpPerIOData->sClient, lpPerIOData->szMessage, cbTransferred, 0);

            // Launch another asynchronous operation
            memset(&lpPerIOData->overlap, 0, sizeof(WSAOVERLAPPED));
            lpPerIOData->Buffer.len = MSGSIZE;
            lpPerIOData->Buffer.buf = lpPerIOData->szMessage;

            WSARecv(lpPerIOData->sClient,
                    &lpPerIOData->Buffer,
                    1,
                    &lpPerIOData->NumberOfBytesRecvd,
                    &lpPerIOData->Flags,
                    &lpPerIOData->overlap,
                    CompletionROUTINE);
        }
    }
```

完成端口

简介

可以把完成端口看成系统维护的一个队列，操作系统把重叠IO操作完成的事件通知放到该队列里，由于是暴露 "操作完成"的事件通知，所以命名为"完成端口"（Completion Ports）。一个socket被创建后，可以在任何时刻和一个完成端口联系起来。



主要涉及以下API：

`CreateIoCompletionPort`函数，该函数主要用于创建一个IO完成端口

```
HANDLE CreateIoCompletionPort (
    HANDLE FileHandle,              // 文件句柄或INVALID_HANDLE_VALUE
    HANDLE ExistingCompletionPort,  // 是已经存在的完成端口。如果为NULL，则为新建一个IO
完成端口IOCP。
    ULONG_PTR CompletionKey,        // 完成键
    DWORD NumberOfConcurrentThreads // 表示有多少个线程在访问这个消息队列，如果设置为0，
也就是允许并发执行的线程数量等于主机CPU的数量
);
```

`GetQueuedCompletionStatus`函数，将调用线程切换到睡眠状态（进入等待线程队列），直到指定的完成端口的**IO完成队列**中出现一项，或者等待超时，就会被唤醒

```
BOOL GetQueuedCompletionStatus(
    HANDLE CompletionPort,          // 表示线程希望对哪个完成端口进行监视
    LPDWORD lpNumberOfBytes,        // 一次完成后的I/O操作所传送数据的字节数
    PULONG_PTR lpCompletionKey,     // 当文件I/O操作完成后，用于存放与之关联的CK
    LPOVERLAPPED *lpOverlapped,     // 为调用IOCP机制所引用的OVERLAPPED结构
    DWORD dwMilliseconds           // 指定调用者等待的时间
);


// 使用
DWORD dwNumBytes;
ULONG_PTR CompletionKey;
OVERLAPPED* pOverlapped;

BOOL bOK = GetQueuedCompletionStatus(hIOCP, &dwNumBytes, &CompletionKey,
    &pOverlapped, 1000);

DWORD dwError = GetLastError();

if (bOK)
{
    // 处理一个成功的IO完成请求
}
else
{
    if (dwError == WAIT_TIMEOUT)
    {
        // 等待IO完成请求的时候超时了
    }
    else
    {
        // 调用GetQueuedCompletionStatus失败了，dwError包含了失败的原因
    }
}
```

`GetQueuedCompletionStatusEx`函数，与`GetQueuedCompletionStatus`类似，但是它可以同时取得多个IO请求的结果，不必让许多线程等待完成端口，可以避免由此产生的上下文切换所带来的开销

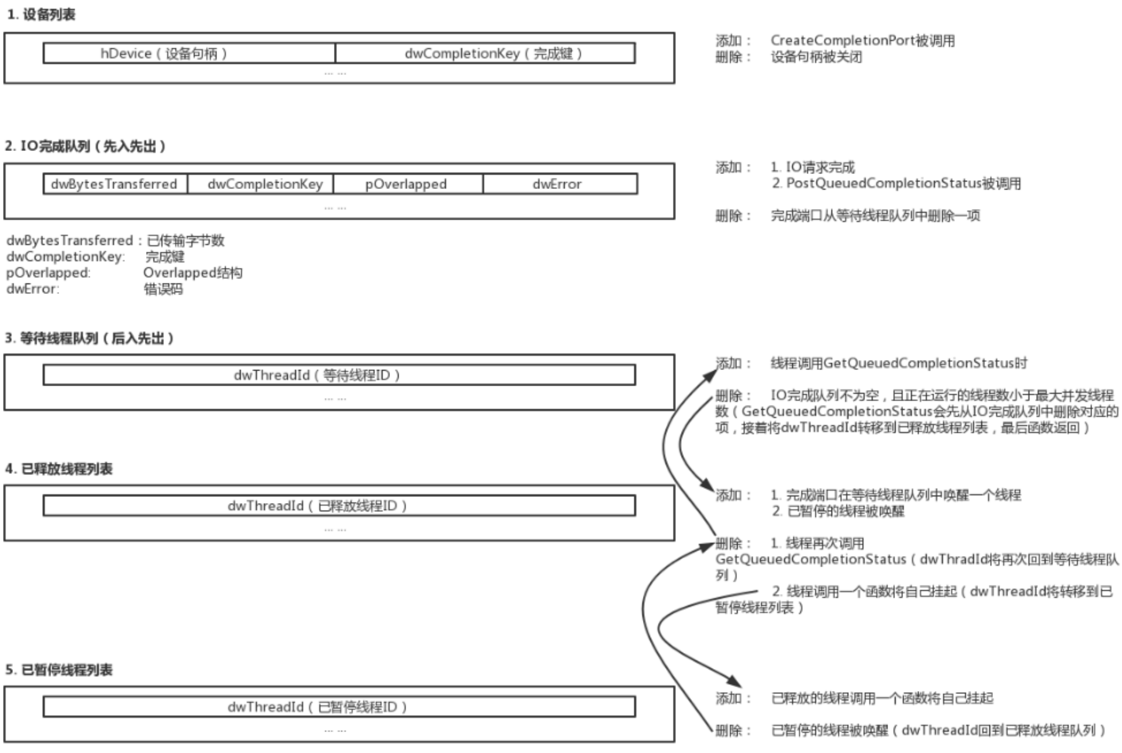
```
BOOL GetQueueCompletionStatusEx(
    HANDLE hCompletionPort,   // 表明线程希望对哪个完成端口进行监视，当本函数被调用的时候，
                              // 它会取出指定的完成端口的IO完成队列中存在的各项（IO请求完成的时候，IO完成队列中有内容），
                              // 将它们的信息复制到pCompletionPortEntries数组参数中
    LPOVERLAPPED_ENTRY pCompletionPortEntries,  // OVERLAPED_ENRY的定义见下文，本数组的内容
    ULONG ulCount,  // 表明最多可以复制多少项到数组中
    PULONG pulNumEnriesRemoved,  // 接收IO完成队列中被移除的IO请求的确切数量
    DWORD dwMilliseconds,  // 超时时间
    BOOL bAlertable  // FALSE: 该函数一直等待一个已经完成的IO请求被添加到端口，知道超出指定的等待时间为止；
                     // TRUE: 队列中没有已完成的IO请求的时候，线程将进入可提醒状态
);


typedef struct _OVERLAPPED_ENTRY{
    ULONG_PTR lpCompletionKey; // 完成键
    LPOVERLAPPED lpOverlapped; // OVERLAPPED结构地址
    ULONG_PTR Internal;   // 没有明确含义
    DWORD dwNumberOfBytesTransferd; // 已传输的字节数
}OVERLAPPED_ENTRY, *LPOVERLAPPED_ENTRY;
```

`PostQueuedCompletionStatus`函数， 用来将一个已经完成的IO通知追加到IO完成队列中

```
BOOL PostQueuedCompletionStatus(
    HANDLE hCompletionPort,    // 将该项添加到哪个IO完成端口的队列中
    DWORD dwNumBytes,    // 下面这三个参数的内容都是返回给调用了
                         // GetQueuedCompletionStatus 的线程
    ULONG_PTR CompletionKey,
    OVERLAPPED* pOverlapped
);
```

内部运作

创建一个IO完成端口的时候，系统内核会创建5个不同的数据结构：**第一个**是设备列表，调用`CreateCompletionPort`的时候如果指定了设备，则会将设备和完成键添加到设备列表中；**第二个**是IO完成队列，当设备的一个异步IO请求完成的时候，系统会检查设备是否与一个IO完成端口相关联，如果设备与一个IO完成端口相关联，那么系统会将该项已完成的IO请求追加到IO完成端口的IO完成队列的末尾；**第三个**是等待线程队列，线程调用`GetQueuedCompletionStatus`的时候，调用线程的线程标识符会被添加到这个等待线程队列中。**当端口的IO完成队列中出现一项的时候**，该完成端口会唤醒等待线程队列中的一个线程；**第四个**是已释放线程队列，当完成端口唤醒一个线程的时候，会将该线程的标识符保存在已释放线程列表中；**第五个**是已暂停线程列表，当一个已经释放的线程调用任何函数将该线程切换到等待状态，那么完成端口会检测到这一情况，将线程的标识符移入已暂停线程列表中。



IO完成端口的内部运作

例子

下面给出了一个使用IO完成端口技术实现的文件复制程序

```
// 使用IO完成端口对文件进行复制
#define BUFFERSIZE (64 * 1024)
#define CK_READ  1
#define CK_WRITE 2
#define MAX_PENDING_IO_REQS   4           // The maximum # of I/Os
// Each I/O Request needs an OVERLAPPED structure and a data buffer
class CIOReq : public OVERLAPPED {
public:
    CIOReq() {
        Internal = InternalHigh = 0;
        Offset = OffsetHigh = 0;
        hEvent = NULL;
        m_nBuffSize = 0;
        m_pvData = NULL;
    }

    ~CIOReq() {
        if (m_pvData != NULL)
            VirtualFree(m_pvData, 0, MEM_RELEASE);
    }

    BOOL AllocBuffer(SIZE_T nBuffSize) {
        m_nBuffSize = nBuffSize;
        m_pvData = VirtualAlloc(NULL, m_nBuffSize, MEM_COMMIT, PAGE_READWRITE);
        return(m_pvData != NULL);
    }

    BOOL Read(HANDLE hDevice, PLARGE_INTEGER pliOffset = NULL) {
```

```
        if (pliOffset != NULL) {
            Offset     = pliOffset->LowPart;
            OffsetHigh = pliOffset->HighPart;
        }
        return(::ReadFile(hDevice, m_pvData, m_nBuffSize, NULL, this));
    }

    BOOL Write(HANDLE hDevice, PLARGE_INTEGER pliOffset = NULL) {
        if (pliOffset != NULL) {
            Offset     = pliOffset->LowPart;
            OffsetHigh = pliOffset->HighPart;
        }
        return(::WriteFile(hDevice, m_pvData, m_nBuffSize, NULL, this));
    }

private:
    SIZE_T m_nBuffSize;
    PVOID  m_pvData;
};

int main(int argc, char *argv[])
{
    BOOL bOk = FALSE;  // 刚开始假设文件拷贝失败

    PCTSTR pszFileSrc = _T("E:\\test.dat");
    PCTSTR pszFileDst = _T("E:\\test2.dat");

    LARGE_INTEGER liFileSizeSrc = { 0 }, liFileSizeDst;

    try{
        {
            // 获取源文件的大小
            HANDLE hFileSrc = CreateFile(pszFileSrc, GENERIC_READ,
                FILE_SHARE_READ, NULL, OPEN_EXISTING,
                FILE_FLAG_NO_BUFFERING | FILE_FLAG_OVERLAPPED, NULL);
            if (hFileSrc == INVALID_HANDLE_VALUE)    goto leave;

            GetFileSizeEx(hFileSrc, &liFileSizeSrc);

            // 目的文件的大小取整到64KB的整数倍
            liFileSizeDst.QuadPart = ( liFileSizeSrc.QuadPart / BUFFERSIZE )
               * BUFFERSIZE + (liFileSizeSrc.QuadPart % BUFFERSIZE > 0 ?
                      BUFFERSIZE : 0);


            // 设置目标文件的大小
            HANDLE hFileDst = CreateFile(pszFileDst, GENERIC_WRITE,
                0, NULL, CREATE_ALWAYS,
                FILE_FLAG_NO_BUFFERING | FILE_FLAG_OVERLAPPED, hFileSrc);
            if (hFileDst == INVALID_HANDLE_VALUE)    goto leave;

            SetFilePointerEx(hFileDst, liFileSizeDst, NULL, FILE_BEGIN);
            SetEndOfFile(hFileDst);

            // 创建IO完成端口（第一步）
            HANDLE hComPort = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL,
                  0, 0);
            if (hComPort == NULL)        goto leave;
```

```
        // 将设备与端口关联（第二步）
        CreateIoCompletionPort(hFileSrc, hComPort, CK_READ, 0);
        CreateIoCompletionPort(hFileDst, hComPort, CK_WRITE, 0);

        CIOReq ior[MAX_PENDING_IO_REQS];
        LARGE_INTEGER liNextReadOffset = { 0 };
        int nReadsInProgress  = 0;
        int nWritesInProgress = 0;

        // 为了向源文件发出读取请求，这里往IO完成端口里添加了4个CK_WRITE来模拟完成通知
        for (int nIOReq = 0; nIOReq < _countof(ior); nIOReq++)
        {
            // 每个IO请求都需要一个缓冲区
            ior[nIOReq].AllocBuffer(BUFFERSIZE);
            nWritesInProgress++;

            // 模拟IO完成通知，但是把已传输字节数都设置为0
            PostQueuedCompletionStatus(hComPort, 0, CK_WRITE, &ior[nIOReq]);

        }

         BOOL bResult = FALSE;

        while ((nReadsInProgress > 0) || (nWritesInProgress > 0))
        {
            // Suspend the thread until an I/O completes
            ULONG_PTR CompletionKey;
            DWORD dwNumBytes;
            CIOReq* pior;

            // 使本线程进入休眠状态，直到有IO请求到来为止
            GetQueuedCompletionStatus(hComPort, &dwNumBytes, &CompletionKey,
                    (OVERLAPPED**) &pior, INFINITE);


            switch (CompletionKey)
            {
            case CK_READ:  // 完成了读，往目的文件写内容
                nReadsInProgress--;
                bResult = pior->Write(hFileDst);
                nWritesInProgress++;
                break;

            case CK_WRITE: // 完成了写，从源文件读内容
                nWritesInProgress--;
                if (liNextReadOffset.QuadPart < liFileSizeDst.QuadPart)
                {
                    // 从源文件从读内容
                    bResult = pior->Read(hFileSrc, &liNextReadOffset);
                    nReadsInProgress++;
                    liNextReadOffset.QuadPart += BUFFERSIZE;
                }
                break;
            }
        }
        bOk = TRUE;
        CloseHandle(hFileDst);
```

```
            CloseHandle(hFileSrc);
            CloseHandle(hComPort);
        }
leave:;
    }// try
    catch(...)
    {
        ;
    }

    if (bOk)
    {
        // 修复目标文件的大小，使之与源文件的大小相同
        // 方法：不指定 FILE_FLAG_NO_BUFFERING 标志，使得文件操作不在扇区边界上进行
        //         可以将目标文件的大小缩减为源文件的大小
        HANDLE hFileDst = CreateFile(pszFileDst, GENERIC_WRITE,
            0, NULL, OPEN_EXISTING, 0, NULL);
        if (hFileDst != INVALID_HANDLE_VALUE)
        {

            SetFilePointerEx(hFileDst, liFileSizeSrc, NULL, FILE_BEGIN);
            SetEndOfFile(hFileDst);
        }
    }

    return bOk;
}
```