Windows I/O完成端口

2009-10-30 10:51

|  |
| --- |
| **WINDOWS完成端口编程** 1、基本概念 2、WINDOWS完成端口的特点 3、完成端口（Completion Ports ）相关数据结构和创建 4、完成端口线程的工作原理 5、Windows完成端口的实例代码  **WINDOWS完成端口编程**    摘要：开发网络程序从来都不是一件容易的事情，尽管只需要遵守很少的一些规则：创建socket，发起连接，接受连接，发送和接收数据，等等。真正的困难在于：让你的程序可以适应从单单一个连接到几千个连接乃至于上万个连接。利用Windows完成端口进行重叠I/O的技术，可以很方便地在Windows平台上开发出支持大量连接的网络服务程序。本文介绍在Windows平台上使用完成端口模型开发的基本原理，同时给出实际的例子。本文主要关注C/S结构的服务器端程序，因为一般来说，开发一个大容量、具有可扩展性的winsock程序就是指服务程序。  **1、基本概念**     设备---指windows操作系统上允许通信的任何东西，比如文件、目录、串行口、并行口、邮件槽、命名管道、无名管道、套接字、控制台、逻辑磁盘、物理磁盘等。绝大多数与设备打交道的函数都是CreateFile/ReadFile/WriteFile等，所以我们不能看到\*\*File函数就只想到文件设备。      与设备通信有两种方式，同步方式和异步方式：同步方式下，当调用ReadFile这类函数时，函数会等待系统执行完所要求的工作，然后才返回；异步方式下，ReadFile这类函数会直接返回，系统自己去完成对设备的操作，然后以某种方式通知完成操作。     重叠I/O----顾名思义，就是当你调用了某个函数（比如ReadFile）就立刻返回接着做自己的其他动作的时候，系统同时也在对I/0设备进行你所请求的操作，在这段时间内你的程序和系统的内部动作是重叠的，因此有更好的性能。所以，重叠I/O是在异步方式下使用I/O设备的。重叠I/O需要使用的一个非常重要的数据结构：OVERLAPPED。  **2、WINDOWS完成端口的特点**    Win32重叠I/O(Overlapped I/O)机制允许发起一个操作，并在操作完成之后接收信息。对于那种需要很长时间才能完成的操作来说，重叠IO机制尤其有用，因为发起重叠操作的线程在重叠请求发出后就可以自由地做别的事情了。在WinNT和Win2000上，提供的真正可扩展的I/O模型就是使用完成端口（Completion Port）的重叠I/O。完成端口---是一种WINDOWS内核对象。完成端口用于异步方式的重叠I/0情况下，当然重叠I/O不一定非得使用完成端口不可，同样设备内核对象、事件对象、告警I/0等也可使用。但是完成端口内部提供了线程池的管理，可以避免反复创建线程的开销，同时可以根据CPU的个数灵活地决定线程个数，而且可以减少线程调度的次数从而提高性能。其实类似于WSAAsyncSelect和select函数的机制更容易兼容Unix，但是难以实现我们想要的“扩展性”。而且windows完成端口机制在操作系统的内部已经作了优化，从而具备了更高的效率。所以，我们选择完成端口开始我们的服务器程序开发。    1）发起操作不一定完成：系统会在完成的时候通知你，通过用户在完成端口上的等待，处理操作的结果。所以要有检查完成端口和取操作结果的线程。在完成端口上守候的线程系统有优化，除非在执行的线程发生阻塞，不会有新的线程被激活，以此来减少线程切换造成的性能代价。所以如果程序中没有太多的阻塞操作，就没有必要启动太多的线程，使用CPU数量的两倍，一般这么多线程就够了。    2）操作与相关数据的绑定方式：在提交数据的时候用户对数据打上相应的标记，记录操作的类型，在用户处理操作结果的时候，通过检查自己打的标记和系统的操作结果进行相应的处理。     3）操作返回的方式：一般操作完成后要通知程序进行后续处理。但写操作可以不通知用户，此时如果用户写操作不能马上完成，写操作的相关数据会被暂存到非交换缓冲区中，在操作完成的时候，系统会自动释放缓冲区，此时发起完写操作，使用的内存就可以释放了。但如果占用非交换缓冲太多会使系统停止响应。  **3、完成端口（Completion Ports ）相关数据结构和创建**     其实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，就可以在任何时刻和一个完成端口联系起来。  OVERLAPPED数据结构 typedef struct \_OVERLAPPED {      ULONG\_PTR Internal; //被系统内部赋值，用来表示系统状态      ULONG\_PTR InternalHigh; //被系统内部赋值，表示传输的字节数      union {          struct {              DWORD Offset; //与OffsetHigh合成一个64位的整数，用来表示从文件头部的多少字节开始操作              DWORD OffsetHigh; //如果不是对文件I/O来操作，则Offset必须设定为0           };         PVOID Pointer;      };     HANDLE hEvent; //如果不使用，就务必设为0；否则请赋一个有效的Event句柄  } OVERLAPPED, \*LPOVERLAPPED;   下面是异步方式使用ReadFile的一个例子  OVERLAPPED Overlapped;  Overlapped.Offset=345;  Overlapped.OffsetHigh=0;  Overlapped.hEvent=0;  //假定其他参数都已经被初始化  ReadFile(hFile,buffer,sizeof(buffer),&dwNumBytesRead,&Overlapped);  这样就完成了异步方式读文件的操作，然后ReadFile函数返回，由操作系统做自己的事情。  下面介绍几个与OVERLAPPED结构相关的函数。  等待重叠I/0操作完成的函数 BOOL GetOverlappedResult ( HANDLE hFile, LPOVERLAPPED lpOverlapped, //接受返回的重叠I/0结构 LPDWORD lpcbTransfer, //成功传输了多少字节数 BOOL fWait //TRUE只有当操作完成才返回，FALSE直接返回，如果操作没有完成，                     //通过用GetLastError( )函数会返回ERROR\_IO\_INCOMPLETE  );  而宏HasOverlappedIoCompleted可以帮助我们测试重叠I/0操作是否完成，该宏对OVERLAPPED结构的Internal成员进行了测试，查看是否等于STATUS\_PENDING值。     一般来说，一个应用程序可以创建多个工作线程来处理完成端口上的通知事件。工作线程的数量依赖于程序的具体需要。但是在理想的情况下，应该对应一个CPU 创建一个线程。因为在完成端口理想模型中，每个线程都可以从系统获得一个“原子”性的时间片，轮番运行并检查完成端口，线程的切换是额外的开销。但在实际开发的时候，还要考虑这些线程是否牵涉到其他堵塞操作的情况。如果某线程进行堵塞操作，系统则将其挂起，让别的线程获得运行时间。因此，如果有这样的情况，可以多创建几个线程来尽量利用时间。  创建完成端口的函数 完成端口是一个内核对象，使用时它总是要和至少一个有效的设备句柄相关联，完成端口是一个复杂的内核对象，创建它的函数是： HANDLE CreateIoCompletionPort(      IN HANDLE FileHandle,      IN HANDLE ExistingCompletionPort,      IN ULONG\_PTR CompletionKey,      IN DWORD NumberOfConcurrentThreads      );  通常创建工作分两步： 第一步，创建一个新的完成端口内核对象，可以使用下面的函数：        HANDLE CreateNewCompletionPort(DWORD dwNumberOfThreads)         {              return CreateIoCompletionPort(INVALID\_HANDLE\_VALUE,NULL,NULL,dwNumberOfThreads);         };        第二步，将刚创建的完成端口和一个有效的设备句柄关联起来，可以使用下面的函数：        bool AssicoateDeviceWithCompletionPort(HANDLE hCompPort,HANDLE hDevice,DWORD dwCompKey)         {            HANDLE h=CreateIoCompletionPort(hDevice,hCompPort,dwCompKey,0);            return h==hCompPort;         };  说明如下：  1）CreateIoCompletionPort函数也可以一次性的既创建完成端口对象，又关联到一个有效的设备句柄。 2）CompletionKey是一个可以自己定义的参数，我们可以把一个结构的地址赋给它，然后在合适的时候取出来使用，最好要保证结构里面的内存不是分配在栈上，除非你有十分的把握内存会保留到你要使用的那一刻。 3）NumberOfConcurrentThreads用来指定要允许同时运行的的线程的最大个数，通常我们指定为0，这样系统会根据CPU的个数来自动确定。 4）创建和关联的动作完成后，系统会将完成端口关联的设备句柄、完成键作为一条纪录加入到这个完成端口的设备列表中。如果你有多个完成端口，就会有多个对应的设备列表。如果设备句柄被关闭，则表中该纪录会被自动删除。  **4、完成端口线程的工作原理**  **1）完成端口管理线程池**    完成端口可以帮助我们管理线程池，但是线程池中的线程需要我们自己使用\_beginthreadex来创建，凭什么通知完成端口管理我们的新线程呢？答案在函数GetQueuedCompletionStatus。该函数原型：  BOOL GetQueuedCompletionStatus(      IN HANDLE CompletionPort,      OUT LPDWORD lpNumberOfBytesTransferred,      OUT PULONG\_PTR lpCompletionKey,      OUT LPOVERLAPPED \*lpOverlapped,      IN DWORD dwMilliseconds  );     这个函数试图从指定的完成端口的I/0完成队列中提取纪录。只有当重叠I/O动作完成的时候，完成队列中才有纪录。凡是调用这个函数的线程将会被放入到完成端口的等待线程队列中，因此完成端口就可以在自己的线程池中帮助我们维护这个线程。完成端口的I/0完成队列中存放了当重叠I/0完成的结果---- 一条纪录，该纪录拥有四个字段，前三项就对应GetQueuedCompletionStatus函数的2、3、4参数，最后一个字段是错误信息dwError。我们也可以通过调用PostQueudCompletionStatus模拟完成一个重叠I/0操作。     当I/0完成队列中出现了纪录，完成端口将会检查等待线程队列，该队列中的线程都是通过调用GetQueuedCompletionStatus函数使自己加入队列的。等待线程队列很简单，只是保存了这些线程的ID。完成端口按照后进先出的原则将一个线程队列的ID放入到释放线程列表中，同时该线程将从等待GetQueuedCompletionStatus函数返回的睡眠状态中变为可调度状态等待CPU的调度。所以我们的线程要想成为完成端口管理的线程，就必须要调用GetQueuedCompletionStatus函数。出于性能的优化，实际上完成端口还维护了一个暂停线程列表，具体细节可以参考《Windows高级编程指南》，我们现在知道的知识，已经足够了。  **2）线程间数据传递**    完成端口线程间传递数据最常用的办法是在\_beginthreadex函数中将参数传递给线程函数，或者使用全局变量。但完成端口也有自己的传递数据的方法，答案就在于CompletionKey和OVERLAPPED参数。 CompletionKey被保存在完成端口的设备表中，是和设备句柄一一对应的，我们可以将与设备句柄相关的数据保存到CompletionKey中，或者将CompletionKey表示为结构指针，这样就可以传递更加丰富的内容。这些内容只能在一开始关联完成端口和设备句柄的时候做，因此不能在以后动态改变。     OVERLAPPED参数是在每次调用ReadFile这样的支持重叠I/0的函数时传递给完成端口的。我们可以看到，如果我们不是对文件设备做操作，该结构的成员变量就对我们几乎毫无作用。我们需要附加信息，可以创建自己的结构，然后将OVERLAPPED结构变量作为我们结构变量的第一个成员，然后传递第一个成员变量的地址给ReadFile这样的函数。因为类型匹配，当然可以通过编译。当GetQueuedCompletionStatus函数返回时，我们可以获取到第一个成员变量的地址，然后一个简单的强制转换，我们就可以把它当作完整的自定义结构的指针使用，这样就可以传递很多附加的数据了。太好了！只有一点要注意，如果跨线程传递，请注意将数据分配到堆上，并且接收端应该将数据用完后释放。我们通常需要将ReadFile这样的异步函数的所需要的缓冲区放到我们自定义的结构中，这样当GetQueuedCompletionStatus被返回时，我们的自定义结构的缓冲区变量中就存放了I/0操作的数据。CompletionKey和OVERLAPPED参数，都可以通过GetQueuedCompletionStatus函数获得。  **3）线程的安全退出**     很多线程为了不止一次地执行异步数据处理，需要使用如下语句 while (true) {        ......        GetQueuedCompletionStatus(...);          ...... } 那么线程如何退出呢，答案就在于上面曾提到过的PostQueudCompletionStatus函数，我们可以向它发送一个自定义的包含了OVERLAPPED成员变量的结构地址，里面含一个状态变量，当状态变量为退出标志时，线程就执行清除动作然后退出。  **5、Windows完成端口的实例代码** DWORD WINAPI WorkerThread(LPVOID lpParam) {  ULONG\_PTR \*PerHandleKey; OVERLAPPED \*Overlap; OVERLAPPEDPLUS \*OverlapPlus; OVERLAPPEDPLUS \*newolp; DWORD dwBytesXfered; while (1) { ret = GetQueuedCompletionStatus(hIocp, &dwBytesXfered, (PULONG\_PTR)&PerHandleKey, &Overlap, INFINITE); if (ret == 0)    {     // Operation failed     continue;    } OverlapPlus = CONTAINING\_RECORD(Overlap, OVERLAPPEDPLUS, ol); switch (OverlapPlus->OpCode) { case OP\_ACCEPT:    // Client socket is contained in OverlapPlus.sclient // Add client to completion port CreateIoCompletionPort((HANDLE)OverlapPlus->sclient, hIocp, (ULONG\_PTR)0, 0);    // Need a new OVERLAPPEDPLUS structure // for the newly accepted socket. Perhaps // keep a look aside list of free structures. newolp = AllocateOverlappedPlus(); if (!newolp)    {     // Error    }  newolp->s = OverlapPlus->sclient; newolp->OpCode = OP\_READ; // This function divpares the data to be sent PrepareSendBuffer(&newolp->wbuf); ret = WSASend(newolp->s, &newolp->wbuf, 1, &newolp->dwBytes, 0, &newolp.ol, NULL); if (ret == SOCKET\_ERROR) {     if (WSAGetLastError() != WSA\_IO\_PENDING)      {        // Error      } } // Put structure in look aside list for later use FreeOverlappedPlus(OverlapPlus); // Signal accept thread to issue another AcceptEx SetEvent(hAcceptThread); break; case OP\_READ: // Process the data read  // Repost the read if necessary, reusing the same // receive buffer as before memset(&OverlapPlus->ol, 0, sizeof(OVERLAPPED)); ret = WSARecv(OverlapPlus->s, &OverlapPlus->wbuf, 1, &OverlapPlus->dwBytes, &OverlapPlus->dwFlags, &OverlapPlus->ol, NULL); if (ret == SOCKET\_ERROR) {     if (WSAGetLastError() != WSA\_IO\_PENDING)      {        // Error      } } break; case OP\_WRITE: // Process the data sent, etc. break; } // switch } // while } // WorkerThread  查看以上代码，注意如果Overlapped操作立刻失败（比如，返回SOCKET\_ERROR或其他非WSA\_IO\_PENDING的错误），则没有任何完成通知时间会被放到完成端口队列里。反之，则一定有相应的通知时间被放到完成端口队列。更完善的关于Winsock的完成端口机制，可以参考 MSDN的Microsoft PlatForm SDK，那里有完成端口的例子。 |

完成端口例子（转）

2009-11-04 10:55

|  |
| --- |
| 这个例子未运行成功，但原理表述的很透彻。  --------------------------------------------------------------------------  #include <winsock2.h>  #include <windows.h>  #include <stdio.h>  #define PORT     5150 #define MSGSIZE 1024  #pragma comment(lib, "ws2\_32.lib")  typedef enum {    RECV\_POSTED }OPERATION\_TYPE;       //枚举,表示状态  typedef struct { WSAOVERLAPPED   overlap; WSABUF          Buffer; char            szMessage[MSGSIZE]; DWORD           NumberOfBytesRecvd; DWORD           Flags; OPERATION\_TYPE OperationType; }PER\_IO\_OPERATION\_DATA, \*LPPER\_IO\_OPERATION\_DATA;    //定义一个结构体保存IO数据  DWORD WINAPI WorkerThread(LPVOID);  int main() { WSADATA                  wsaData; SOCKET                   sListen, sClient; SOCKADDR\_IN              local, client; DWORD                    i, dwThreadId; int                      iaddrSize = sizeof(SOCKADDR\_IN); HANDLE                   CompletionPort = INVALID\_HANDLE\_VALUE; SYSTEM\_INFO              systeminfo; LPPER\_IO\_OPERATION\_DATA lpPerIOData = NULL;  //初始化Socket WSAStartup(0x0202, &wsaData);  // 初始化完成端口 CompletionPort = CreateIoCompletionPort(INVALID\_HANDLE\_VALUE, NULL, 0, 0);  // 有几个CPU就创建几个工作者线程 GetSystemInfo(&systeminfo); for(i = 0; i < systeminfo.dwNumberOfProcessors; i++) {    CreateThread(NULL, 0, WorkerThread, CompletionPort, 0, &dwThreadId); }  // 创建套接字 sListen = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);  // 绑定套接字 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));  // 开始监听！ listen(sListen, 3);  //主进程的这个循环中循环等待客户端连接，若有连接，则将该客户套接字于完成端口绑定到一起 //然后开始异步等待接收客户传来的数据。 while (TRUE) {    // 如果接到客户请求连接，则继续，否则等待。    sClient = accept(sListen, (struct sockaddr \*)&client, &iaddrSize);    //client中保存用户信息。    printf("Accepted client:%s:%d\n", inet\_ntoa(client.sin\_addr), ntohs(client.sin\_port));     //将这个最新到来的客户套接字和完成端口绑定到一起。    //第三个参数表示传递的参数，这里就传递的客户套接字地址。    //最后一个参数为0 表示有和CPU一样的进程数。即1个CPU一个线程    CreateIoCompletionPort((HANDLE)sClient, CompletionPort, ( ULONG\_PTR)sClient, 0);         // 初始化结构体 使用堆内存分配    lpPerIOData = (LPPER\_IO\_OPERATION\_DATA)HeapAlloc(GetProcessHeap(), HEAP\_ZERO\_MEMORY, sizeof(PER\_IO\_OPERATION\_DATA));    lpPerIOData->Buffer.len = MSGSIZE; // len=1024    lpPerIOData->Buffer.buf = lpPerIOData->szMessage;     lpPerIOData->OperationType = RECV\_POSTED; //操作类型     WSARecv(sClient,      //异步接收消息，立刻返回。      &lpPerIOData->Buffer,   //获得接收的数据      1,        //The number of WSABUF structures in the lpBuffers array.      &lpPerIOData->NumberOfBytesRecvd, //接收到的字节数，如果错误返回0      &lpPerIOData->Flags,   //参数，先不管      &lpPerIOData->overlap,   //输入这个结构体咯。      NULL);    }  //向每个工作者线程都发送—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”. PostQueuedCompletionStatus(CompletionPort, 0xFFFFFFFF, 0, NULL);  CloseHandle(CompletionPort); closesocket(sListen); WSACleanup();  return 0; }  //工作者线程有一个参数，是指向完成端口的句柄 DWORD WINAPI WorkerThread(LPVOID CompletionPortID) { HANDLE                   CompletionPort=(HANDLE)CompletionPortID; DWORD                    dwBytesTransferred; SOCKET                   sClient; LPPER\_IO\_OPERATION\_DATA lpPerIOData = NULL;  while (TRUE) {    GetQueuedCompletionStatus(    //遇到可以接收数据则返回，否则等待     CompletionPort,     &dwBytesTransferred,    //返回的字数     (PULONG\_PTR&)sClient,           //是响应的哪个客户套接字？     (LPOVERLAPPED \*)&lpPerIOData, //得到该套接字保存的IO信息     INFINITE);       //无限等待咯。不超时的那种。     if (dwBytesTransferred == 0xFFFFFFFF)    {     return 0;    }         if(lpPerIOData->OperationType == RECV\_POSTED) //如果收到数据    {     if (dwBytesTransferred == 0)     {      //失去客户端连接      closesocket(sClient);      HeapFree(GetProcessHeap(), 0, lpPerIOData);        //释放结构体     }     else     {      lpPerIOData->szMessage[dwBytesTransferred] = '\0';      send(sClient, lpPerIOData->szMessage, dwBytesTransferred, 0); //将接收到的消息返回                // Launch another asynchronous operation for sClient      memset(lpPerIOData, 0, sizeof(PER\_IO\_OPERATION\_DATA));      lpPerIOData->Buffer.len = MSGSIZE;      lpPerIOData->Buffer.buf = lpPerIOData->szMessage;      lpPerIOData->OperationType = RECV\_POSTED;      WSARecv(sClient,               //循环接收        &lpPerIOData->Buffer,        1,        &lpPerIOData->NumberOfBytesRecvd,        &lpPerIOData->Flags,        &lpPerIOData->overlap,        NULL);     }    }    }     return 0; }  /\* 首先，说说主线程： 1.创建完成端口对象 2.创建工作者线程（这里工作者线程的数量是按照CPU的个数来决定的，这样可以达到最佳性能） 3.创建监听套接字，绑定，监听，然后程序进入循环 4.在循环中，我做了以下几件事情： (1).接受一个客户端连接 (2).将该客户端套接字与完成端口绑定到一起(还是调用CreateIoCompletionPort，但这次的作用不同)， 注意，按道理来讲，此时传递给CreateIoCompletionPort的第三个参数应该是一个完成键， 一般来讲，程序都是传递一个单句柄数据结构的地址，该单句柄数据包含了和该客户端连接有关的信息， 由于我们只关心套接字句柄，所以直接将套接字句柄作为完成键传递； (3).触发一个WSARecv异步调用，用到了“尾随数据”，使接收数据所用的缓冲区紧跟在WSAOVERLAPPED对象之后， 此外，还有操作类型等重要信息。  在工作者线程的循环中，我们 1.调用GetQueuedCompletionStatus取得本次I/O的相关信息（例如套接字句柄、传送的字节数、单I/O数据结构的地址等等） 2.通过单I/O数据结构找到接收数据缓冲区，然后将数据原封不动的发送到客户端 3.再次触发一个WSARecv异步操作  \*/ |

|  |  |  |  |
| --- | --- | --- | --- |
|  | 查看文章 |  |  |

完成端口例子2（转）

2009-11-04 10:56

|  |
| --- |
| 这个例子可以正常运行  ---------------------------------------------------------  #include <winsock2.h>  #include <windows.h>  #include <stdio.h>  #define PORT 5150  #define DATA\_BUFSIZE 8192  #pragma comment(lib, "Ws2\_32")  typedef struct                        //这个玩意就是灌数据，取数据的一个自定义数据结构                                                //和那个wm\_data差不了多少，不过就是老要塞一个OverLapped结构，  {     OVERLAPPED Overlapped;     WSABUF DataBuf;     CHAR Buffer[DATA\_BUFSIZE];                         DWORD BytesSEND;                                 //发送字节数     DWORD BytesRECV;                                  } PER\_IO\_OPERATION\_DATA, \* LPPER\_IO\_OPERATION\_DATA;  typedef struct  {     SOCKET Socket;  } PER\_HANDLE\_DATA, \* LPPER\_HANDLE\_DATA;  DWORD WINAPI ServerWorkerThread(LPVOID CompletionPortID);  void main(void)  {     SOCKADDR\_IN InternetAddr;     SOCKET Listen;     SOCKET Accept;     HANDLE CompletionPort;     SYSTEM\_INFO SystemInfo;     LPPER\_HANDLE\_DATA PerHandleData;     LPPER\_IO\_OPERATION\_DATA PerIoData;     int i;     DWORD RecvBytes;     DWORD Flags;     DWORD ThreadID;     WSADATA wsaData;     DWORD Ret;     if ((Ret = WSAStartup(0x0202, &wsaData)) != 0)     {        printf("WSAStartup failed with error %d\n", Ret);        return;     }     //     //完成端口的建立得搞2次，这是第一次调用，至于为什么？我问问你     //     if ((CompletionPort = CreateIoCompletionPort(INVALID\_HANDLE\_VALUE, NULL, 0, 0)) == NULL)     {        printf( "CreateIoCompletionPort failed with error: %d\n", GetLastError());        return;    }     //老套子api，不谈也罢     GetSystemInfo(&SystemInfo);         //发现2个CPU，那就开个双倍的线程跑吧     for(i = 0; i < SystemInfo.dwNumberOfProcessors \* 2; i++)     {        HANDLE ThreadHandle;               //        //完成端口挂到线程上面来了，就像管子把灌数据的和读数据的两头都连上了 //        if ((ThreadHandle = CreateThread(NULL, 0, ServerWorkerThread, CompletionPort, 0, &ThreadID)) == NULL)       {           printf("CreateThread() failed with error %d\n", GetLastError());           return;        }             CloseHandle(ThreadHandle);    }     //     //启动一个监听socket ，以下都是长长的交代     //     if ((Listen = WSASocket(AF\_INET, SOCK\_STREAM, 0, NULL, 0,        WSA\_FLAG\_OVERLAPPED)) == INVALID\_SOCKET)     {        printf("WSASocket() failed with error %d\n", WSAGetLastError());       return;     }     InternetAddr.sin\_family = AF\_INET;     InternetAddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);     InternetAddr.sin\_port = htons(PORT);     if (bind(Listen, (PSOCKADDR) &InternetAddr, sizeof(InternetAddr)) == SOCKET\_ERROR)     {        printf("bind() failed with error %d\n", WSAGetLastError());        return;     }     if (listen(Listen, 5) == SOCKET\_ERROR)     {        printf("listen() failed with error %d\n", WSAGetLastError());        return;     }     //     // 监听端口打开，就开始在这里循环，一有socket连上，WSAAccept就创建一个socket，     // 这个socket 又和完成端口联上,     //     // 嘿嘿，完成端口第二次调用那个createxxx函数，为什么，留给人思考思考可能更深刻，     // 反正这套路得来2次，     // 完成端口completionport和accept socket挂起来了，     //     while(TRUE)     {      //主线程跑到这里就等啊等啊，但是线程却开工了，        if ((Accept = WSAAccept(Listen, NULL, NULL, NULL, 0)) == SOCKET\_ERROR)        {           printf("WSAAccept() failed with error %d\n", WSAGetLastError());           return;        }            //该函数从堆中分配一定数目的字节数.Win32内存管理器并不提供相互分开的局部和全局堆.提供这个函数只是为了与16位的Windows相兼容    //       if ((PerHandleData = (LPPER\_HANDLE\_DATA) GlobalAlloc(GPTR, sizeof(PER\_HANDLE\_DATA))) == NULL)        {           printf("GlobalAlloc() failed with error %d\n", GetLastError());           return;        }              PerHandleData->Socket = Accept;              //       //把这头和完成端口completionPort连起来       //就像你把漏斗接到管子口上,开始要灌数据了       //        if (CreateIoCompletionPort((HANDLE) Accept, CompletionPort, (DWORD) PerHandleData, 0) == NULL)        {           printf("CreateIoCompletionPort failed with error %d\n", GetLastError());           return;        }               //        //清管子的数据结构,准备往里面灌数据        //        if ((PerIoData = (LPPER\_IO\_OPERATION\_DATA) GlobalAlloc(GPTR,sizeof(PER\_IO\_OPERATION\_DATA))) == NULL)        {           printf("GlobalAlloc() failed with error %d\n", GetLastError());           return;        }     //用0填充一块内存区域       ZeroMemory(&(PerIoData->Overlapped), sizeof(OVERLAPPED));        PerIoData->BytesSEND = 0;        PerIoData->BytesRECV = 0;        PerIoData->DataBuf.len = DATA\_BUFSIZE;        PerIoData->DataBuf.buf = PerIoData->Buffer;        Flags = 0;               //        // accept接到了数据，就放到PerIoData中,而perIoData又通过线程中的函数取出,       //        if (WSARecv(Accept, &(PerIoData->DataBuf), 1, &RecvBytes, &Flags, &(PerIoData->Overlapped), NULL) == SOCKET\_ERROR)        {           if (WSAGetLastError() != ERROR\_IO\_PENDING)           {              printf("WSARecv() failed with error %d\n", WSAGetLastError());              return;           }        }     }  }  //  //线程一但调用，就老在里面循环，  // 注意，传入的可是完成端口啊，就是靠它去取出管子中的数据  //  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)  {        //        //在这里检查完成端口部分的数据buf区，数据来了吗？        // 这个函数参数要看说明，        // PerIoData 就是从管子流出来的数据,        //PerHandleData 也是从管子里取出的，是何时塞进来的，       //就是在建立第2次createIocompletionPort时      //        if (GetQueuedCompletionStatus(CompletionPort, &BytesTransferred, (LPDWORD)&PerHandleData, (LPOVERLAPPED \*) &PerIoData, INFINITE) == 0)        {           printf("GetQueuedCompletionStatus failed with error %d\n", GetLastError());           return 0;        }        // 检查数据传送完了吗        if (BytesTransferred == 0)        {           printf("Closing socket %d\n", PerHandleData->Socket);           if (closesocket(PerHandleData->Socket) == SOCKET\_ERROR)           {              printf("closesocket() failed with error %d\n", WSAGetLastError());              return 0;           }           GlobalFree(PerHandleData);           GlobalFree(PerIoData);           continue;        }           //      //看看管子里面有数据来了吗？=0，那是刚收到数据      //        if (PerIoData->BytesRECV == 0)        {           PerIoData->BytesRECV = BytesTransferred;           PerIoData->BytesSEND = 0;        }        else   //来了，        {           PerIoData->BytesSEND += BytesTransferred;        }     printf("print: %d\n", PerIoData->BytesRECV);            //        // 数据没发完？继续send出去       //       if (PerIoData->BytesRECV > PerIoData->BytesSEND)        {           ZeroMemory(&(PerIoData->Overlapped), sizeof(OVERLAPPED)); //清0为发送准备           PerIoData->DataBuf.buf = PerIoData->Buffer + PerIoData->BytesSEND;           PerIoData->DataBuf.len = PerIoData->BytesRECV - PerIoData->BytesSEND;         //1个字节一个字节发送发送数据出去           if (WSASend(PerHandleData->Socket, &(PerIoData->DataBuf), 1, &SendBytes, 0,              &(PerIoData->Overlapped), NULL) == SOCKET\_ERROR)           {              if (WSAGetLastError() != ERROR\_IO\_PENDING)              {                 printf("WSASend() failed with error %d\n", WSAGetLastError());                 return 0;              }           }        }        else        {           PerIoData->BytesRECV = 0;           Flags = 0;           ZeroMemory(&(PerIoData->Overlapped), sizeof(OVERLAPPED));           PerIoData->DataBuf.len = DATA\_BUFSIZE;           PerIoData->DataBuf.buf = PerIoData->Buffer;           if (WSARecv(PerHandleData->Socket, &(PerIoData->DataBuf), 1, &RecvBytes, &Flags,              &(PerIoData->Overlapped), NULL) == SOCKET\_ERROR)           {              if (WSAGetLastError() != ERROR\_IO\_PENDING)              {                 printf("WSARecv() failed with error %d\n", WSAGetLastError());                 return 0;              }           }        }     }  } |

译]IOCP服务器/客户端实现 (转)

2009-09-22 16:46

|  |
| --- |
| ——A simple IOCP Server/Client Class **By** [**spinoza**](http://www.codeproject.com/script/Articles/MemberArticles.aspx?amid=431028)  **——**译： Ocean    Email: [Ocean2oo6@hotmail.com](mailto:Ocean2oo6@hotmail.com)  原文选自CodeProject  源代码: <http://www.codeproject.com/KB/IP/iocp_server_client/IOCP-Demo.zip>  http://www.codeproject.com/KB/IP/iocp\_server\_client/IOCP-SRC.zip  纯属学习交流用途，转载请声明。  This source code uses the advanced IOCP technology which can efficiently serve multiple clients. It also presents some solutions to practical problems that arise with the IOCP programming API, and provides a simple echo client/server with file transfer.  **1.1要求**  本文希望读者对C++，TCP/IP ，Socket编程，MFC以及多线程比较熟悉  源代码使用Winsock2.0 以及IOCP技术，因此需要：   * + Windows NT/2000 or later: Requires Windows NT 3.5 or later.   + Windows 95/98/ME: Not supported.   + Visual C++ .NET, or a fully updated Visual C++ 6.0.   **1.2 摘要**  当你开发不同类型的软件时，你总会需要进行C/S的开发。 完成一个完善的C/S代码对于编码人员来说是一件困难的事情。 本文给出了一个简单的但是却是却十分强大的C/S源代码，他可以扩展成任何类型的C/S程序。 源代码使用了IOCP技术，该技术可以有效地处理多客户端。 IOCP 对于“一个客户端一个线程”所有面临的瓶颈（或者其他）问题提出了一种有效的解决方案，他只使用少量的执行线程以及异步的输入输出、接受发送。IOCP计数被广泛的用于各种高性能的服务器，如Apache等。 源代码同时也提供了一组用于处理通信的常用功能以及在C/S软件中经常用到功能，如文件接受/传输功能以及逻辑线程池操作。本文将主要关注一种围绕IOCP API在实际中的解决方案，以及呈现源代码的完整文档。 随后，我将展示一个可以处理多连接和文件传输的echo C/S程序。  **2.1 介绍**  本文阐述了一个类，他可以被同时用于客户端和服务器端代码。 这个类使用IOCP(Input Output Completion Ports) 以及异步(non-blocking) 功能调用。 源代码是基于很多其他源代码和文章的。  使用此源代码，你可以：  - 为多主机进行链接、或者链接到多主机的客户端和服务器  - 异步的发送和接受文件  - 创建和管理一个逻辑工作线程池，他可以处理繁重的C/S请求或计算  找到一段完善的却又简单的、可以处理C/S通信的代码是一件困难的事情。 在网络上找到的代码要么太过于复杂（可能多于20个类），或者不能提供有效的效率。 本代码就是以简单为设计理念的，文档也尽可能的完善。 在本文中，我们可以很简单的使用由Winsock 2.0提供的IOCP技术，我也会谈到一些在编码时会遇到的棘手的问题以及他们的解决方法。  **2.2 异步输入输出完成端口(IOCP)的介绍**  一个服务器程序要是不能同时处理多客户端，那么我们可以说这个程序是毫无意义的，而我们为此一般会使用异步I/O调用或者多线程技术去实现。 从定义上来看，一个异步I/O 调用可以及时返回而让I/O挂起。 在同一时间点上，I/O异步调用必须与主线程进行同步。 这可以使用各种方式，同步主要可以通过以下实现：  - 使用事件(events) 只要异步调用完成，一个Signal就会被Set。 这种方式主要的缺点是线程必须去检查或者等待这个event 被Set  - 使用GetOverlappedResult 功能。 这种方式和上面的方式有同样的缺点。  - 使用异步例程调用(APC)。 对于这种方式 有几个缺点。 首先，APC总是在调用线程的上下文被调用的，其次，为了执行APCs，调用线程必须被挂起，这被成为alterable wait state  - 使用IOCP。 这种方式的缺点是很多棘手的编码问题必须得以解决。 编写IOCP可能一件让人持续痛苦的事情。  **2.2.1 为什么使用IOCP?**  使用IOCP,我们可以克服”一个客户端一个线程”的问题。 我们知道，这样做的话，如果软件不是运行在一个多核及其上性能就会急剧下降。 线程是系统资源，他们既不是无限制的、也不是代价低廉的。  IOCP提供了一种只使用一些（I/O worker）线程去“相对公平地”完成多客户端的”输入输出”。线程会一直被挂起，而不会使用CPU时间片，直到有事情做为止。  **2.3 什么是IOCP?**  我们已经提到IOCP 只不过是一个线程同步对象，和信号量（semaphore）相似，因此IOCP 并不是一个复杂的概念。 一个IOCP 对象是与多个I/O对象关联的，这些对象支持挂起异步IO调用。 知道一个挂起的异步IO调用结束为止，一个访问IOCP的线程都有可能被挂起。  **3. IOCP是如何工作的？**  要获得更多的信息，我推荐其他的一些文章（译者注，在CodeProject）  当使用IOCP时，你必须处理三件事情：将一个Socket关联到完成端口， 创建一个异步I/O调用，与线程进行同步。 为了获得异步IO调用的结果，比如，那个客户端执行了调用，你必须传入两个参数：the CompletionKey 参数, 和 OVERLAPPED 结构。  **3.1 CompletionKey参数**  第一个参数是CompletionKey，一个DWORD类型值。 你可以任何你希望的标识值，这将会和对象关联。 一般的，一个包含一些客户端特定对象的结构体或者对象的指针可以使用此参数传入。 在源代码中，一个指向ClientContext结构被传到CompletionKey参数中。  **3.2 OVERLAPPED 参数**  这个参数一般用于传入被异步IO调用使用的内存buffer。 我们必须主意这个数据必须是锁在内存的，不能被换页出物理内存。 我们稍后会讨论这个。  **3.3 将socket与完成端口绑定**  一旦一个完成端口创建，我们就可以使用CreateToCompletionPort方法去将socket绑定到完成端口，这看起来像这面这样：  BOOL IOCPS::AssociateSocketWithCompletionPort(SOCKET socket,  HANDLE hCompletionPort, DWORD dwCompletionKey)  {  HANDLE h = CreateIoCompletionPort((HANDLE) socket,  hCompletionPort, dwCompletionKey, m\_nIOWorkers);  return h == hCompletionPort;  }  **3.4 创建异步IO调用**  创建真正的异步调用：可以调用WSASend,WSARecv。 他们也需要一个WSABUF参数，这个参数包含一个指向被使用的buffer的指针。首要规则是，当服务器/客户端试图调用一个IO操作时，他们不是直接的操作，而是先被传送到完成端口，这将会被IO工作线程去完成操作。 之所以要这样做，我们是想要CPU 调用更加公平。 IO调用可以通过发送（Post）状态到完成端口来实现，看一下代码：  BOOL bSuccess = PostQueuedCompletionStatus(m\_hCompletionPort,  pOverlapBuff->GetUsed(),  (DWORD) pContext, &pOverlapBuff->m\_ol);  **3.5 与线程同步**  与IO工作线程同步是通过GetQueuedCompletionStatus方法完成的（代码如下）。该方法也提供了CompleteKey参数以及OVERLAPPED 参数  BOOL GetQueuedCompletionStatus(  HANDLE CompletionPort, *// handle to completion port*  LPDWORD lpNumberOfBytes, *// bytes transferred*  PULONG\_PTR lpCompletionKey, *// file completion key*  LPOVERLAPPED \*lpOverlapped, *// buffer*  DWORD dwMilliseconds *// optional timeout value*  );  **3.6 IOCP编码四个棘手的问题以及他们的解决方法**  使用IOCP时我们会遇到一些问题，这其中的有一些是不那么直观的。 在使用IOCP的多线程场合中， 线程的控制并不是很直观，因为通信与线程之间是没有关系的。 在本节中，我们将展示四个使用IOCP在开发C/S程序时不同的问题。 他们是：  - WSAENOBUGS 错误问题  - 数据包重排序问题  - 非法访问问题  **3.6.1 WSAENOBUGS 错误问题**  这个问题不是那么直观并且也很难发现，因为第一感觉是，他看起来像是一个平常的死锁或者内存泄露bug。 假如你开发了你的服务器，他也能工作的很好。 当你对他进行压力测试时， 他突然挂起了。 如果你比较幸运，你可以发现他是与WSAENOBUGS 错误相关的。  每次重叠发送或者接受操作时，被提交的数据buffer都是有可能被锁住的。 当内存锁住时，他就不能被换页到物理内存外。 一个操作系统限制了可以被锁住的内存大小。 当超出了限制时，重叠操作就会因WSAENOBUGS 错误失败。  如果一个服务器在每个连接上进行了许多Overlapped接收，随着连接数量的增加，我们就可能达到这个限制。 如果一个服务器希望处理非常大的突发用户，服务器POST可以从每个链接上接收到0字节的数据，因为已经没有buffer 与接收操作关联了，没有内存需要被锁住了。使用这种方式，每个socket的接收buffer应该被保持完整 因为一旦0字节的接收操作完成，服务器可以简单的进行非阻塞的接收去获取socket 接收buffer中的所有缓存数据。 当非阻塞因为WSAWOULDBLOCK错误失败时，这里就不再会有被挂起的数据了。 这种设计可以用于那种需要最大可能的处理突发访问链接，这是以牺牲吞吐量作为代价的。当然，你对客户端如何与服务器端进行交互知道的越多越好。 在前一个例子中，一个非阻塞的Receive将会在0字节接收完成后马上进行以便去取得缓存的数据。如果服务器知道客户端突然发送了很多数据，那么在接收0字节数据的Receive完成后，他应该POST一个或者多个Overlapped Reveives以便接收客户端发送的一些数据（大于每个socket接收buffer的最大缓冲buffer，默认是8k）。  一个针对WSAENOBUFFERS错误问题的简单而实际的解决方式在源代码中已经提供了。 我们进行一个使用0字节Buffer的异步WSARead(…)(请查看OnZeroByteRead(…))。 当这个调用完成后，我们知道在TCP/IP栈中存在数据，然后我们使用大小为MAXIMUMPACKAGE buffer进行几个异步的WSARead。 这种解决方法只是在有数据来到时才锁住物理内存，这样可以解决WSAENOBUFS问题。 但是这种解决方式会降低服务器的吞吐量。  **3.6.2 数据包重排序问题**  这个问题也在参考文献【3】中提到。 虽然使用IO完成端口的提交操作总是按照他们被提交的顺序完成，线程调度问题可能会导致与完成端口绑定的真正任务是以未知的顺序完成的。 例如，如果你有两个IO工作线程， 然后你应该接收到“byte chunk 1, byte chunk 2, byte chunk 3”，你可能会以错误的顺序去处理byte chunk，如“byte chunk 2, byte chunk 1， byte chunk 3”。 这也就是意味着当你POST一个发送请求到IO完成端口进行发送数据时，数据可能会被以另外的顺序进行发送。  这可以通过只使用单个工作线程来解决， 只提交一个IO调用，直到他完成， 但是如果我们这样做的话，我们将会失去IOCP所有的好处。  一个实际的解决方式是添加一个顺序号给我们的buffer类， 只处理buffer中的顺序号正确的buffer数据。 这意味着，buffer如果有不正确的号码必须保存起来以便之后用到， 因为性能的原因，我们将会将buffers保存到一个hash map对象中(如 m\_SendBufferMap and m\_ReadBufferMap)  要获得这种解决方式更多的信息，请阅读源代码，然后在IOCPS类中查看下面的函数：   * GetNextSendBuffer (..) and GetNextReadBuffer(..), to get the ordered send or receive buffer. * IncreaseReadSequenceNumber(..) and IncreaseSendSequenceNumber(..), to increase the sequence numbers.   **3.6.3 异步挂起读以及byte chunk 数据包处理问题**  最常用的服务器协议是基于包的协议，该协议中前X字节表示头部， 头部包含了一个完整的包的长度。服务器可以读取头部，查看多少数据需要的，然后继续读取数据直到读完一个包。 这在服务器在一个时间上只进行一个异步调用时可以工作的很好。但是如果我们想挖掘IOCP服务器的所有潜力，我们需要有多个异步Reads 去等待数据的到来。 这意味这几个异步Reads 是完全乱序的（如前所述）， byte chunk 流被挂起的reads操作返回来将不再是顺序的了（译者注，实际上这几个Reads操作是资源竞争的，同时读取数据，返回时的顺序不定）。 并且，一个byte chunk 流可以包含一个或者多个数据包，或者半包。 如下图所示：  [clip_image001[4]](http://byfiles.storage.msn.com/y1phlPOS8dfzWmeppyPCyHYk7KHjfqJ2g2ocde-Av7ugq-FAWGdUugeEhzAFjhkgO64?PARTNER=WRITER)  图1 该图展示了部分包（绿色）以及完整包（黄色）可能会在不同的byte chunk流中异步到达  这意味着我们不得对byte chunk进行处理以便获得完整的数据包。 进一步，我们不得不处理部分包（图中绿色）。 这会使得包的处理变得更加麻烦。 该问题的完整解决方法可以在IOCPS类的ProcessPackage方法中找到。  **3.6.4 非法访问问题**  这是一个小问题，一般是由于代码的设计导致的， 而不是IOCP特定的问题。 假如一个客户端链接丢失了，而一个IO调用返回了一个错误flag，随后我们知道客户端不存在了。 在CompleteKey参数中，我们将一个DWORD类型指针转型为ClientContext指针，接下来去访问或者删除他？ 访问异常就是这么发生的！  这个问题的解决方式是为包含挂起IO调用的结构体添加一个数字（nNumberOfPendlingIO），然后当我们知道这将不会有挂起的IO调用时才去删除结构体。 这是在方法EnterIoLoop(..) function 和 ReleaseClientContext(..).完成的  **3.7 源代码架构**  整个源代码是提供一些简单的类去处理在IOCP中需要面对的棘手的问题。 源代码也提供了一些方法，这些方法经常被通信或者软件中用到的文件接收传输、逻辑线程池处理等  [clip_image002[4]](http://byfiles.storage.msn.com/y1pkDkqgHN4l2tfE4f0YQSl8fTZBQMRysE2YItXsibkMGwMrRMuIFNuDGOAc4m6y0KwuzlaiZh70CXygD1YQrdopA?PARTNER=WRITER)  图2：上面的图片展示了IOCP类源代码功能  我们拥有几个工作线程去处理来自IOCP的异步IO调用， 这些工作线程调用某些虚方法去把需要大量计算的请求放入到工作队列中。 逻辑工作线程从队列中取得这些任务，处理、然后把结果通过类提供的一些方法发送回去。 GUI通常是通过Windows消息与主class通信的（MFC不是线程安全的），然后调用方法或者使用共享变量。  [clip_image003[4]](http://byfiles.storage.msn.com/y1p5fBbMdaUjpFbTRIvAH510tOSzlsG8FTpsdIE1HBO-la73aUQ092Q51sDOWFnYovo?PARTNER=WRITER)  图3： 上图展示了类的框架  我们在图3中可以看到以下的类：  - CIOCPBuffer: 一个用于管理被异步IO调用使用的buffers  - IOCPS: 用于通信的主要类。  - JobItem：一个包含需要被逻辑工作线程执行的任务的结构。  - ClientContext : 一个包含了客户端特定信息（状态、数据等等）的结构  **3.7.1 Buffer设计 —— CIOCPBuffer 类**  当时用异步IO调用时，我们必须提供一个私有的buffer去被IO操作使用。 当我们分配buffers时，需要考虑几个问题：  - 分配与释放内存时很昂贵的， 所以我们应该重用已经分配的buffers(内存)。 所以，我们可以使用链表结构去节省buffers:  // Free Buffer List..  CCriticalSection m\_FreeBufferListLock;  CPtrList m\_FreeBufferList;  // OccupiedBuffer List.. (Buffers that is currently used)  CCriticalSection m\_BufferListLock;  CPtrList m\_BufferList;  // Now we use the function AllocateBuffer(..)  // to allocate memory or reuse a buffer.  - 某些时候，当一个IO调用完成时，我们可能会在buffer中得到部分的包数据，所以我们需要将buffer分割以便得到完整的消息。 这是通过IOCPS类中的SpiltBuffer方法来实现的。 同时，有时候我们需要在buffer之间拷贝信息，而这是通过IOCPS类中的AddAndFlush()来完成的。  - 我们知道，我们同时需要为我们的buffer添加一个序列号以及一个状态(IOType变量，IOZeroReadCompleted 等等)  - 我们同时还需要一些方法去把byte数据流转换成数据，有些方法也在CIOCPBuffer类中被提供了  我们先前提到的所有问题的解决方案都已经在CIOCPBuffer类中得到支持了。  **3.8 如何使用本源代码？**  通过从IOCP(见图3)派生你自己的类以及使用虚方法、使用IOCPS类提供的功能（如线程池），这就可以实现任何类型的服务器和客户端，我们可以使用有限数量的线程来有效的应对大量的连接。  **3.8.1服务器/客户端的启动和关闭**  要启动服务器，调用以下的方法：  BOOL Start(int nPort=999,int iMaxNumConnections=1201,  int iMaxIOWorkers=1,int nOfWorkers=1,  int iMaxNumberOfFreeBuffer=0,  int iMaxNumberOfFreeContext=0,  BOOL bOrderedSend=TRUE,  BOOL bOrderedRead=TRUE,  int iNumberOfPendlingReads=4);   * nPortt   服务器将进行监听的端口号（如果是客户端的话，我们可以让他是-1）   * iMaxNumConnections   最大可允许的连接数（使用一个很大的数字）   * iMaxIOWorkers   输入输出工作线程的数量   * nOfWorkers   逻辑工作线程的数量   * iMaxNumberOfFreeBuffer   我们将要节省下来进行重用的buffer最大数量（-1表示不重用，0表示不限制数量）   * iMaxNumberOfFreeContext   我们将要节省下来进行重用的客户端信息对象的最大数量（-1表示不重用，0表示不限制数量）   * bOrderedRead   是否需要有序读取（我们已经在3.6.2中讨论过这个）   * bOrderedSend   是够需要有序的写入（我们已经在3.6.2中讨论过这个）   * iNumberOfPendlingReads   等待数据而挂起的异步读取循环的数量  建立一个远程连接（客户端模式下nPort = -1） 调用下面的方法：  Connect(const CString &strIPAddr, int nPort)   * strIPAddr   远程服务器的IP地址   * nPort   端口  关闭时请确认服务器调用以下的方法：ShutDown().  For example:  MyIOCP m\_iocp;  if(!m\_iocp.Start(-1,1210,2,1,0,0))  AfxMessageBox("Error could not start the Client");  ….  m\_iocp.ShutDown();  **4.1 代码描述**  更多的代码细节，请阅读源代码中的注释。  **4.1.1 虚方法**   * NotifyNewConnection   当新的连接被建立时被调用   * NotifyNewClientContext   当一个空的ClientContext结构被分配时被调用   * NotifyDisconnectedClient   当一个客户端断线时被调用   * ProcessJob   当一个工作线程试图执行一个任务时调用   * NotifyReceivedPackage   当一个新的包到达时的提示   * NotifyFileCompleted   当一个文件传输完成时的提示  **4.1.2 重要的变量**  请注意，所有需要使用共享变量的方法将对进行额外的加锁，这对于避免非法访问和重叠写入非常重要的。所有需要加锁且使用XXX名字的变量需要加锁，他们有一个XXXLock变量   * m\_ContextMapLock;   维护所有的客户端数据（Socket,客户端数据等等）   * ContextMap m\_ContextMap; * m\_NumberOfActiveConnections   维护已有的连接数量  4.1.2 重要的方法   * GetNumberOfConnections()   返回连接数   * CString GetHostAdress(ClientContext\* p)   返回给定客户端Context的主机地址   * BOOL ASendToAll(CIOCPBuffer \*pBuff);   发送buffer中的内容给所有的客户端。   * DisconnectClient(CString sID)   与一个给定唯一标示号的客户端断开连接   * CString GetHostIP()   返回本地IP地址   * JobItem\* GetJob()   从队列中移除JobItem，如果没有任务的话将会返回NULL   * BOOL AddJob(JobItem \*pJob)   添加任务到队列中   * BOOL SetWorkers(int nThreads)   设置在任何时刻能被调用的逻辑工作线程数量   * DisconnectAll();   断开所有的客户端   * ARead(…)   创建一个异步读取   * ASend(…)   创建一个异步发送。 发送数据到客户端   * ClientContext\* FindClient(CString strClient)   根据给定的字符串ID查找客户端。 不是线程安全的   * DisconnectClient(ClientContext\* pContext, BOOL bGraceful=FALSE);   断开一个客户端   * DisconnectAll()   断开所有已有的连接   * StartSendFile(ClientContext \*pContext)   transmitfile(..) function.  发送ClientContext结构中声明的文件，通过使用优化的transmitfile(…)方法   * PrepareReceiveFile(..)   准备一个接收文件的连接，当你调用这个方法时，所有接收的字节将写入到一个文件   * PrepareSendFile(..)   打开一个文件以及发送一个包含文件信息的包到远程连接。 这个方法也会把Asread(…)禁用掉，直到文件被传送完成或者终止。   * DisableSendFile(..)   禁用文件发送模式   * DisableRecevideFile(..)   禁用接收模式  **5.1 文件传输**  文件传输使用过Winsock 2.0的TransmitFile方法完成的。 TransmitFile方法使用一个已经连接的socket句柄进行传输文件数据。这个方法使用操作系统的缓存管理器（cache manager）来接收文件数据，他提供了基于socket的高性能文件数据传输。 当我们使用异步文件传输时，这里几个需要注意的地方：  - 除非TransmitFile方法返回，不能在该socket上进行读取和写入，因为这样会损坏文件。因此，所有在PrepareSendFile()之后对ASend的调用都会禁用掉。  - 因为操作系统是有序读取文件数据的，你可以通过使用FILE\_FLAG\_SEQUENTIAL\_SCAN去打开文件句柄来提高缓存的性能。  - 当发送文件时，我们使用内核的异步例程调用（TF\_USE\_KERNEL\_APC）. 使用TF\_USE\_KERNEL\_APC可以获得很好的性能。 当我们在Context TransmitFile初始化时使用的线程使用非常繁重的计算任务时，这有可能会阻止APCs的执行。  文件传输是以下面的顺序运作的：服务器通过调用PrepareSendFile(…)初始化文件传输。当客户端接收到文件的信息时，他会调用PrepareReceiveFile(...) ,然后发送一个包给服务器去开始文件传输。 当一个包到达服务器时，服务器调用StartSendFile(…)方法，这个犯非法使用了高性能的TransmitFile(…)方法去传输特定的文件。  **6 源代码例子**  提供的源代码例子时一个echo 客户端/服务器应用程序，他可以支持文件传输（见图4）。 在源代码中，类MyIOCP继承自IOCP, 他通过在4.1.1节中提到的使用虚方法处理客户端和服务器端的交互。  客户端或者服务器端最重要的部分是NotifyReceivePackage, 如下所示：  void MyIOCP::NotifyReceivedPackage(CIOCPBuffer \*pOverlapBuff,  int nSize,ClientContext \*pContext)  {  BYTE PackageType=pOverlapBuff->GetPackageType();  switch (PackageType)  {  case Job\_SendText2Client :  Packagetext(pOverlapBuff,nSize,pContext);  break;  case Job\_SendFileInfo :  PackageFileTransfer(pOverlapBuff,nSize,pContext);  break;  case Job\_StartFileTransfer:  PackageStartFileTransfer(pOverlapBuff,nSize,pContext);  break;  case Job\_AbortFileTransfer:  DisableSendFile(pContext);  break;};  }  该方法处理接收到的消息以及执行由远程连接发送的请求。 在本例中，他只是简单的echo或者文件传输而已。 源代码被非常两个项目，IOCP 和 IOCPClient, 一个是服务器端的连接而另外一个时客户端的连接。  **6.1 编译上的问题**  当使用VC++ 6.0 或者 .NET时，你可能会在使用CFile时得到一些奇怪的错误，如：  “if (pContext->m\_File.m\_hFile !=  INVALID\_HANDLE\_VALUE) <-error C2446: '!=' : no conversion "  "from 'void \*' to 'unsigned int'”  这个问题可以通过更新头文件或者你的VC++6.0 版本来避免，或者改一下类型转换错误。 在一些修改之后，服务器/客户端代码是可以在没有MFC时使用的。  **7. 特殊的考虑以及首要原则**  当你在其他类型的应用程序中使用这个代码时，你可能会遇到一些跟本代码相关的陷阱以及“多线程编程”的陷阱。非确定性的错误时那些随机发生的错误，通过相同的一系列操作是很难重现这些非确定性的错误的。 这些错误是已存在的错误中最糟糕的类型，同时通常他们时因为在核心设计的编码实现时产生的。 当服务器执行多个IO工作线程时，为连接的客户端服务，如果程序员没有很好的搞清楚多线程环境下的编码问题，非确定性错误如非法访问可能就会发生。  原则1  在没有使用context lock(如下面的例子)对客户端Context(如ClientContext)进行加锁时，不要读取/写入。提示（Notification）方法(如 nofity\* (ClientContext \* pContext))已经是“线程安全”的了，你可以在不对context进行加锁的情况下访问ClientContext的成员。  //Do not do it in this way  // …  If(pContext->m\_bSomeData)  pContext->m\_iSomeData=0;  // …    // Do it in this way.  //….  pContext->m\_ContextLock.Lock();  If(pContext->m\_bSomeData)  pContext->m\_iSomeData=0;  pContext->m\_ContextLock.Unlock();  //…  同时，记住当你加锁一个Context时，其他的线程或者GUI可能会进行等待。  原则2：  避免或者在”context lock”中具有复杂的“context locks”或者其他类型锁的代码中 “特殊考虑”，因为这有可能会引起死锁（比如，A等待B，而B等待C,C等待A =>死锁）。  pContext-> m\_ContextLock.Lock();  *//… code code ..*  pContext2-> m\_ContextLock.Lock();  *// code code..*  pContext2-> m\_ContextLock.Unlock();  *// code code..*  pContext-> m\_ContextLock.Unlock();  上面的代码会引起死锁。  原则3：  不要在Notification方法外访问Client Context(比如，Notify\*(ClientContext \* pContext)). 如果你你要这么做，你必须使用m\_ContextMapLock.Lock();…m\_ContstMapLock.Unlock(); 代码如下：  ClientContext\* pContext=NULL ;  m\_ContextMapLock.Lock();  pContext = FindClient(ClientID);  *// safe to access pContext, if it is not NULL*  *// and are Locked (Rule of thumbs#1:)*  *//code .. code..*  m\_ContextMapLock.Unlock();  *// Here pContext can suddenly disappear because of disconnect.*  *// do not access pContext members here.*  **8 改进**  将来该代码会做出以下的更新：  1.AcceptEX()方法，接收一个新的连接将会被添加到源代码，该方法去处理短连接以及DOS攻击。  2.源代码将会被移植到其他平台，如Win32,STL,WTL  **9 FAQ**  Q1: 内存使用量（服务器程序）将会在客户端连接增加的时候稳定的增加，这可以从任务管理器看到。然而即使客户端断线时，内存使用量还是没有下降，这是怎么回事？  A1：代码会重用已经分配的buffers而不是不断释放和分配。 你可以通过改变参数iMaxNumberOfFreeBuffer 和 iMaxNumberOfFreeContext来改变这种方式，请阅读3.8.1节。  Q2：我在.NET环境下编译时遇到以下的错误：“error C2446:’!=’ no conversion from 'unsigned int' to 'HANDLE'”等等，这是怎么回事？  A2：这是因为SDK不同的头文件造成的。只要把他转换成HANDLE 编译器就可以让你通过了。 你也可以这是删除一行代码 #define TRANSFERFILEFUNCTIONALITY 然后再编译一下。  Q3：源代码可以在没有MFC的情况下使用吗？ 纯Win32或者在一个服务里面？  A3：源代码只是暂时使用了GUI开发的。 我开发这个客户端/服务器解决方案时使用了MFC环境作为GUI。 当然，你可以在一个通常的服务器环境下使用他。很多人已经这么做了。只要把MFC相关的东西，如CString,CPtrList 等等移走，用Win32的类去替换。 我其实也不喜欢MFC，如果你改变的代码，请发一份给我，谢谢。  Q4：做得太好了！ 谢谢你所做的工作， 你会在什么时候不是在监听线程中实现AcceptEX(…)？  A4：当代码稳定后。 现在他已经很稳定了，但是我知道一些IO工作线程和挂起的读操作的整合可能会导致一些问题。 我很高兴你喜欢我的代码，请投我一票！  Q5：为什么启动多个IO工作线程？ 如果你没有多线程机器的话就没有必要了？  A5：不，没有必要开启多个IO工作线程。 只要一个线程就能处理所有的连接。 一般的家庭计算机中，一个工作线程就可以有最佳的表现。你也不需要考虑潜在的非法访问。但是当计算机变得越来越强大的时候（比如超线程，双核等等） 多线程的可能性为什么不会有？  Q6：为什么使用多个挂起的读操作？他有什么优势？  A6：这取决都与开发者进行服务器开发采取的策略，也就是说“许多并发连接”还是 “高吞吐量服务器”。 拥有多个挂起的读操作增加了服务器的吞吐量，这是因为TCP/IP包将会被直接写到我们传入的buffer而不是TCP/IP栈（不会有双缓冲）。 如果服务器知道客户端突然发送了大量数据，多个挂起的读操作可以提高性能（高吞吐量）。然而，每个挂起的接收操作（使用WSARevc()）会强迫内核去锁住接收buffers进入非换页池。 这在物理内存满时（很多并发连接时产生）就会引起WSAENBUFFERS错误。这必须被考虑进去。 再者，如果你使用多于一个IO工作线程，访问包的顺序就会被打乱（因为IOCP的结构），这样就需要额外的工作去维护顺序 以便不用多个挂起的读操作。在这个设计中，当IO工作线程的数量大于1个时，多个挂起的读操作是被关闭的，这样就可以不需要处理重排序（重排序的话，序列号是必须在负载中存在的）。  Q7：在先前的文章中，你提到我们使用VirtualAlloc方法而不是new实现内存管理,为什么你没有实现呢？  A7：当你使用new 来分配内存时，内存会被分配在虚拟内存或者物理内存。到底内存被分配到什么地方时不知道的，内存可以被分配到两个页面上。 这意味着当我们访问一个特定的数据时，我们加载了太多的内存到物理内存。 再者，你不知道内存是在虚拟内存还是物理内存，你也不能够高数系统什么时候“写回到磁盘中是不需要的（如果我们在内存中已经不再关心该数据）。但是请注意！任何使用VirtualAlloc\* 的new分配都将会填满到64kB（页面文件大小） 所以你如果你分配一个新的VAS绑定到物理内存，操作系统将会消耗一定量的物理内存去达到页面大小，这将会消耗VAS去执行填满到64kB。使用VirtualAlloc会比较麻烦： new 和 malloc 在内部使用了 virtualAlloc，但是每次你使用new/delete 分配内存时，很多其他的计算就会被完成，而你不需要控制你的数据（彼此关联的数据）刚好在相同的页面（而不是跨越了两个页面）。 我发现相对于代码的复杂度来说，我能获得的性能提高的非常小的。 |