IOCP编程之基本原理

在我的博客之前写了很多关于IOCP的“行云流水”似的看了让人发狂的文章，尤其是几篇关于IOCP加线程池文章，更是让一些功力不够深厚的初学IOCP者，有种吐血的感觉。为了让大家能够立刻提升内力修为，并且迅速的掌握IOCP这个Windows平台上的乾坤大挪移心法，这次我决定给大家好好补补这个基础。

要想彻底征服IOCP，并应用好IOCP这个模型，首先就让我们穿越到遥远的计算机青铜器时代（以出现PC为标志），那时候普通的PC安装的还是DOS平台，微软公司主要靠这个操作系统在IT界的原始丛林中打拼，在DOS中编写程序，不得不与很多的硬件直接打交道，而最常操作的硬件无非是键盘、声显卡、硬盘等等，这些设备都有一个特点就是速度慢，当然是相对于PC平台核心CPU的速度而言，尤其是硬盘这个机械电子设备，其速度对于完全电子化得CPU来说简直是“相对静止”的设备。很多时候CPU可以干完n件（n>1000）事情的时间中，这些硬件可能还没有完成一件事情，显然让CPU和这些硬件同步工作将是一种严重的浪费，并且也不太可能，此时，聪明的硬件设计师们发明了一种叫做中断的操作方式，用以匹配这种速度上的严重差异。中断工作的基本原理就是，CPU首先设置一个类似回调函数的入口地址，其次CPU对某个硬件发出一个指令，此时CPU就去干别的活计了，最后那个慢的象蜗牛一样的硬件执行完那个指令后，就通知CPU，让CPU暂时“中断”手头的工作，去调用那个“回调函数”。至此一个完整的中断调用就结束了。这个模型曾经解决了显卡与CPU不同步的问题，最重要的是解决了硬盘速度与CPU速度严重不匹配的问题，并因此还派生出了更有名的DMA（直接内存访问技术，主要是指慢速硬件可以读写原本只能由CPU直接读写的内存）硬盘IO方式。（注意这里说的中断工作方式只是中断工作方式的一种，并不是全部，详细的中断原理请参阅其它专业文献。）

其实“中断”方式更像是一种管理模型，比如在一个公司中，如果要老板时时刻刻盯着员工作事情，那么除非是超人，否则无人能够胜任，同时对于老板这个稀缺资源来说也是一种极起严重的浪费。更多时候老板只是发指令给员工，然后员工去执行，而老板就可以做别的事情，或者干脆去打高尔夫休息，当员工完成了任务就会通过电话、短信、甚至e-mail等通知老板，此时老板就去完成一个响应过程，比如总结、奖罚、发出新指令等等。由此也看出如果一个公司的“老板占用率”（类似CPU占用率）太高，那么就说明两种情况：要么是它的员工很高效，单位时间内完成的指令非常多；要么是公司还没有建立有效的“中断”响应模型。如果你的公司是后者，那么你就可以试着用这个模型改造公司的管理了，由此你可以晋升到管理层，而不用再去管你的服务端程序有没有使用IOCP了，呵呵呵。

如果真的搞明白了这个传说中的“中断”操作方式，那么理解IOCP的基本原理就不费劲了。

结束了计算机的青铜时代后，让我们穿越到现在这个“计算机蒸汽”时代，（注意不是“计算机IT”时代，因为计算机还没法自己编写程序让自己去解决问题）。在现代，Windows几乎成了PC平台上的标准系统，而PC平台上的几大件还是没有太大的变化，除了速度越来越快。而因为操作系统的美妙封装，我们也不用再去直接同硬件打交道了，当然编写驱动程序的除外。

在Windows平台上，我们不断的调用着WriteFile和ReadFile这些抽象的函数，操作着“文件”这种抽象的信息集合，很多时候调用这些函数时，是以一种“准同步”的方式操作硬件的，比如要向一个文件中写入1M的信息，只有等到WriteFile函数返回，操作才算结束，这个过程中，我们的程序则类似死机一样，等待硬盘写入操作的结束（实际是被系统切换出了当前的CPU时间片）。于此同时，调用了WriteFile的线程则无法干别的任何事情。因为整个线程是在以一种称为过程化的模型中运行，所有的处理流程全部是线性的。对于程序的流畅编写来说，线性化的东西是一个非常好的东西，甚至几乎早期很多标准的算法都是基于程序是过程化得这一假设而设计的。而对于一些多任务、多线程环境来说，这种线性的工作方式会使系统严重低效，甚至造成严重的浪费，尤其在现代多核CPU已成为主流的时候，显然让一个CPU内核去等待另一个CPU内核完成某事后再去工作，是非常愚蠢的一种做法。

面对这种情况，很多程序员的选择是多线程，也就是专门让一个线程去进行读写操作，而别的线程继续工作，以绕开这些看起来像死机一样的函数，但是这个读写线程本身还是以一种与硬盘同步的方式工作的。然而这并不是解决问题的最终方法。我们可以想象一个繁忙的数据库系统，要不断的读写硬盘上的文件，可能在短短的一秒钟时间就要调用n多次WriteFile或ReadFile，假设这是一个网站的后台数据库，那么这样的读写操作有时还可能都是较大的数据块，比如网站的图片就是比较典型的大块型数据，这时显然一个读写线程也是忙不过来的，因为很有可能一个写操作还没有结束，就会又有读写操作请求进入，这时读写线程几乎变成了无响应的一个线程，可以想象这种情况下，程序可能几乎总在瘫痪状态，所有其它的线程都要等待读写操作线程完活。也许你会想多建n个线程来进行读写操作，其实这种情况会更糟糕，因为不管你有多少线程，先不说浪费了多少系统资源，而你读写的可能是相同的一块硬盘，只有一条通道，结果依然是一样的，想象硬盘是独木桥，而有很多人（线程）等着过桥的情形，你就知道这更是一个糟糕的情形。所以说在慢速的IO面前，多线程往往不是“万灵丹”。

面对这种情形，微软公司为Windows系统专门建立了一种类似“青铜时代”的中断方式的模型来解决这个问题。当然，不能再像那个年代那样直接操作硬件了，需要的是旧瓶装新酒了。微软是如何做到的呢，实际还是通过“回调函数”来解决这个问题的，大致也就是要我们去实现一个类似回调函数的过程，主要用于处理来自系统的一些输入输出操作“完成”的通知，相当于一个“中断”，然后就可以在过程中做输入输出完成的一些操作了。比如在IO操作完成后删除缓冲，继续发出下一个命令，或者关闭文件，设备等。实际上从逻辑的角度来讲，我们依然可以按照线性的方法来分析整个过程，只不过这是需要考虑的是两个不同的函数过程之间的线性关系，第一个函数是发出IO操作的调用者，而第二个函数则是在完成IO操作之后的被调用者，。而被调用的这个函数在输入输出过程中是不活动的，也不占用线程资源，它只是个过程（其实就是个函数，内存中的一段代码而已）。调用这些函数则需要一个线程的上下文，实际也就是一个函数调用栈，很多时候，系统会借用你进程空间中线程来调用这个过程，当然前提条件是事先将可以被利用的线程设置成“可警告”状态，这也是线程可警告状态的全部意义，也就是大多数内核同步等待函数bAlertable（有些书翻译做可警告的，我认为应该理解为对IO操作是一种“时刻警惕”的状态）参数被传递TRUE值之后的效果。比如：WaitForSingleObjectEx、SleepEx等等。

当然上面说的这种方式其实是一种“借用线程”的方式，当进程中没有线程可借，或者可借的线程本身也比较忙碌的时候，会造成严重的线程争用情况，从而造成整体性能低下，这个方式的局限性也就显现出来了。注意“可警告”状态的线程，并不总是在可以被借用的状态，它们本身往往也需要完成一些工作，而它调用一些能够让它进入等待状态的函数时，才可以被系统借用，否则还是不能被借用的。当然借用线程时因为系统有效的保护了栈环境和寄存器环境，所以被借用的线程再被还回时线程环境是不会被破坏的。

鉴于借用的线程的不方便和不专业，我们更希望通过明确的“创建”一批专门的线程来调用这些回调函数（为了能够更深入的理解，可以将借用的线程想象成出租车，而将专门的线程想象成私家车），因此微软就发明了IOCP“完成端口”这种线程池模型，注意IOCP本质是一种线程池的模型，当然这个线程池的核心工作就是去调用IO操作完成时的回调函数，这就叫专业！这也是IOCP名字的来由，这就比借用线程的方式要更加高效和专业，因为这些线程是专门创建来做此工作的，所以不用担心它们还会去做别的工作，而造成忙碌或不响应回调函数的情况，另外因为IO操作毕竟是慢速的操作，所以几个线程就已经足可以应付成千上万的输入输出完成操作的请求了（还有一个前提就是你的回调函数做的工作要足够少），所以这个模型的性能是非常高的。也是现在Windows平台上性能最好的输入输出模型。它首先就被用来处理硬盘操作的输入输出，同时它也支持邮槽、管道、甚至WinSock的网络输入输出。

至此对于完成端口的本质原理应该有了一个比较好的理解，尤其是掌握了IOCP是线程池模型的这一本质，那么对于之后的IOCP实际应用就不会有太多的疑问了。接下去就让我们从实际编程的角度来了解一下IOCP，也为彻底掌握IOCP编程打下坚实的基础。

要应用IOCP，首先就要我们创建一个叫做IOCP的内核对象，这需要通过CreateIoCompletionPort这个函数来创建，这个函数的原型如下：

HANDLE WINAPI CreateIoCompletionPort(

  \_\_in          HANDLE FileHandle,

  \_\_in          HANDLE ExistingCompletionPort,

  \_\_in          ULONG\_PTR CompletionKey,

  \_\_in          DWORD NumberOfConcurrentThreads

);

这个函数是个本身具有多重功能的函数（Windows平台上这样的函数并不多），需要用不同的方式来调用，以实现不同的功能，它的第一个功能正如其名字所描述的“Create”，就是创建一个完成端口的内核对象，要让他完成这个功能，只需要指定NumberOfConcurrentThreads参数即可，前三个参数在这种情况下是没有意义的，只需要全部传递NULL即可，象下面这样我们就创建了一个完成端口的内核对象：

HANDLE hICP = ::CreateIoCompletionPort(INVALID\_HANDLE\_VALUE,NULL,0,1);

这里首先解释下为什么第一个参数不是NULL而是INVALID\_HANDLE\_VALUE，因为第一个参数按照定义是一个文件的句柄，也就是需要IOCP操作的文件句柄，而代表“NULL”文件句柄的实际值是INVALID\_HANDLE\_VALUE，这是因为NULL实际等于0，而0这个文件句柄被用于特殊用途，所以要用INVALID\_HANDLE\_VALUE来代表“NULL”意义的文件，INVALID\_HANDLE\_VALUE的值是-1或者0xFFFFFFFF。

最后一个参数NumberOfConcurrentThreads就有必要好好细细的说说了，因为很多文章中对于这个参数总是说的含糊其辞，不知所云，有些文章中甚至人云亦云的说赋值为CPU个数的2倍即可，所谓知其然，不知其所以然。其实这个参数的真实含义就是“真正并发同时执行的最大线程数”，这个并发是真并发，怎么去理解呢，如果你有两颗CPU，而你赋值为2那么就是说，在每颗CPU上执行一个线程，并且真正的并发同时执行，当然如果你设置了比CPU数量更大的数值，它的含义就变成了一个理论并发值，而实际系统的最大可能的严格意义上的并发线程数就是CPU个数，也就是你在任务管理器中看到的CPU个数（可能是物理个数，也可能是内核个数，还有可能是超线程个数，或者它们的积）。讲到这里大家也许就有疑问了，为什么有些文章资料中说要设置成CPU个数的2倍呢？这通常是一个半经验值，因为大多数IOCP完成回调的过程中，需要一些逻辑处理，有些是业务性的，有些要访问数据库，有些还可能访问硬盘，有些可能需要进行数据显示等等，无论哪种处理，这总是要花费时间的，而系统发现你设置了超过CPU个数的并发值时，那么它就尽可能的来回切换这些线程，使他们在一个时间段内看起来像是并发的，比如在1ms的时间周期内，同时有4个IOCP线程被调用，那么从1ms这段时间来看的话，可以认为是有4个线程被并发执行了，当然时间可以无限被细分，真并发和模拟并发实际就是针对时间细分的粒度来说的。这样一来如何设置并发数就是个设计决策问题，决策的依据就是你的回调函数究竟要干些什么活，如果是时间较长的活计，就要考虑切换其它线程池来完成，如果是等待性质的活计，比如访问硬盘，等待某个事件等，就可以设置高一点的并发值，强制系统切换线程造成“伪并发”，如果是非常快速的活计，那么就直接设置CPU个数的并发数就行了，这时候防止线程频繁切换是首要任务。当然并发数最好是跟踪调试一下后再做决定，默认的推荐值就是CPU个数的2倍了。（绕了一大圈我还是“人云亦云”了一下，哎呦！谁扔的砖头？！）

上面的全部就是创建一个完成端口对象，接下来就是打造线程了，打造的方法地球人都知道了，就是CreateThread，当然按照人云亦云的说法应该替之以\_beginthread或\_beginthreadex，原因嘛？你想知道？真的想知道？好了看你这么诚恳的看到了这里，那就告诉你吧，原因其实就是因为我们使用的语言从本质上说是C/C++，很多时候我们需要在线程函数中调用很多的C/C++味很重的库函数，而有些函数是在Windows诞生以前甚至是多线程多任务诞生以前就诞生了，这些老爷级的函数很多都没有考虑过多线程安全性，还有就是C++的全局对象静态对象等都需要调用它们的构造函数来初始化，而调用的主体就是线程，基于这些原因就要使用C/C++封装过的创建线程函数来创建线程，而CreateThread始终是Windows系统的API而已，它是不会考虑每种语言环境的特殊细节的，它只考虑系统的环境。

好了让我们继续打造线程的话题，要创建线程，实际核心就是准备一个线程函数，原型如下：

1、使用CreateThread时：

DWORD WINAPI ThreadProc(LPVOID lpParameter);

2、使用\_beginthread时：

void \_\_cdecl ThreadProc( void \* pParameter );

3、使用\_beginthreadex时：

unsigned int \_\_stdcall ThreadProc(void\* pParam);

其实上面三个函数原型都是很简单的，定义一个线程函数并不是什么难事，而真正困难的是对线程的理解和定义一个好的线程函数。这里我就不在多去谈论关于线程原理和如何写好一个线程函数的内容了，大家可以去参阅相关的文献。

现在我们接着讨论IOCP的专用线程如何编写，IOCP专用线程编写的核心工作就是调用一个同步函数GetQueuedCompletionStatus，为了理解的方便性，你可以想象这个函数的工作原理与那个有名的GetMessage是类似的，虽然这种比喻可能不太确切，但是他们工作方式是有些类似的地方，它们都会使调用它们的线程进入一种等待状态，只是这个函数不是等待消息队列中的消息，它是用来等待“被排队的完成状态”（就是它名字的含义）的，排队的完成状态，其实就是IO操作完成的通知（别告诉我你还不知道什么是IO操作），如果当前没有IO完成的通知，那么这个函数就会让线程进入“等待状态”，实际也就是一种“可警告”的状态，这样系统线程调度模块就会登记这个线程，一旦有IO完成通知，系统就会“激活”这个线程，立即分配时间片，让该线程开始继续执行，已完成IO完成通知的相关操作。

首先让我看看GetQueuedCompletionStatus的函数原型：

BOOL WINAPI GetQueuedCompletionStatus(

  \_\_in          HANDLE CompletionPort,

  \_\_out         LPDWORD lpNumberOfBytes,

  \_\_out         PULONG\_PTR lpCompletionKey,

  \_\_out         LPOVERLAPPED\* lpOverlapped,

  \_\_in          DWORD dwMilliseconds

);

第一个参数就是我们之前创建的那个完成端口内核对象的句柄，这个参数实际也就是告诉系统，我们当前的线程是归哪个完成端口对象来调度。

第二个参数是一个比较有用的参数，在函数返回后它将告诉我们这一次的IO操作实际传输或者接收了多少个字节的信息，这对于我们校验数据收发完整性非常有用。

第三个参数是与完成端口句柄绑定的一个一对一的数据指针，当然这个数据是我们绑到这个完成端口句柄上的，其实这个参数也是类似本人博客文章中所提到的那个“火车头”的作用的，它的作用和意义就是在我们得到完成通知时，可以拿到我们在最开初创建完成端口对象时绑定到句柄上的一个自定义的数据。这里给一个提示就是，在用C++的类封装中，通常这个参数我们会在绑定时传递类的this指针，而在GetQueuedCompletionStatus返回时又可以拿到这个类的this指针，从而可以在这个完成线程中调用类的方法。

第四个参数就是在本人其它IOCP相关博文中详细介绍过的重叠操作的数据结构，它也是一个火车头，这里就不在赘述它的用法了，请大家查阅本人其它博文拙作。

第五个参数是一个等待的毫秒数，也就是GetQueuedCompletionStatus函数等待IO完成通知的一个最大时间长度，如果超过这个时间值，GetQueuedCompletionStatus就会返回，并且返回值一个0值，此时调用GetLastError函数会得到一个明确的WAIT\_TIMEOUT，也就是说它等待超时了，也没有等到一个IO完成通知。这时我们可以做一些相应的处理，而最常见的就是再次调用GetQueuedCompletionStatus函数让线程进入IO完成通知的等待状态。当然我们可以传递一个INFINITE值，表示让此函数一直等待，直到有一个完成通知进入完成状态队列。当然也可以为这个参数传递0值，表示该函数不必等待，直接返回，此时他的工作方式有些类似PeekMessage函数。

函数的参数和原型都搞清楚了，下面就让我们来看看调用的例子：

UINT CALLBACK IOCPThread(void\* pParam)

{

CoInitialize(NULL);

       DWORD dwBytesTrans = 0;

       DWORD dwPerData = 0;

LPOVERLAPPED lpOverlapped = NULL;

       while(1)

       {

              BOOL bRet = GetQueuedCompletionStatus( hICP,&dwBytesTrans

,&dwPerData,&lpOverlapped,INFINITE);

              if( NULL == lpOverlapped )

              {

                     DWORD dwError = GetLastError();

                     ......//错误处理

}

              PMYOVERLAPPED pMyOL

= CONTAINING\_RECORD(lpOverlapped, MYOVERLAPPED, m\_ol);

              if( !HasOverlappedIoCompleted(lpOverlapped) )

              {//检测到不是一个真正完成的状态

                     DWORD dwError = GetLastError();

                     ......//错误处理

              }

                     ...... //继续处理

}

       return 0;

}

在这个线程函数中，我们写了一个死循环，这个是必要的，因为这个线程要反复处理IO完成通知的操作。跟我们常见的消息循环是异曲同工。

有了线程函数，接着就是创建线程了，对于IOCP来说，创建多少线程其实是一个决策问题，一般的原则就是创建的实际线程数量，不应小于调用CreateIoCompletionPort创建完成端口对象时指定的那个最大并发线程数。一般的指导原则是：如果完成线程的任务比较繁重大多数情况下执行的是其它的慢速等待性质的操作（比如磁盘磁带读写操作，数据库查询操作，屏幕显示等）时，由于这些操作的特点，我们可以适当的提高初始创建的线程数量。但是如果是执行计算密集型的操作时（比如网游服务端的场景变换运算，科学计算，工程运算等等），就不易再靠增加线程数来提高性能，因为这类运算会比较耗费CPU，没法切换出当前CPU时间片，多余的线程反倒会造成因为频繁的线程切换而造成整个程序响应性能的下降，此时为了保证IOCP的响应性，可以考虑再建立线程池来接力数据专门进行计算，这也是我的博文《IOCP编程之“双节棍”》篇中介绍的用线程池接力进行计算并提高性能的思想的核心。

下面的例子展示了如何创建IOCP线程池中的线程：

SYSTEM\_INFO si = {};

GetSystemInfo(&si);

//创建CPU个数个IOCP线程

for( int i = 0; i < si.dwNumberOfProcessors; i ++ )

{

UINT nThreadID = 0;

//以暂停的状态创建线程状态

HANDLE hThread = (HANDLE)\_beginthreadex(NULL,0,IOCPThread

,(void\*)pThreadData,CREATE\_SUSPENDED,(UINT\*)&nThreadID);

//然后判断创建是否成功

if( NULL == reinterpret\_cast<UINT>(m\_hThread)

                     || 0xFFFFFFFF == reinterpret\_cast<UINT>(m\_hThread) )

{//创建线程失败

              ......//错误处理

}

::ResumeThread(hThread);//启动线程

}

创建好了IOCP的线程池，就可以往IOCP线程池中添加用来等待完成的那些重叠IO操作的句柄了，比如：重叠IO方式的文件句柄，重叠IO操作方式的SOCKET句柄，重叠IO操作的命名（匿名）管道等等。上面的这个操作可以被称作将句柄绑定到IOCP，绑定的方法就是再次调用CreateIoCompletionPort函数，这次调用时，就需要明确的指定前两个参数了，例子如下：

//创建一个重叠IO方式的SOCKET

SOCKET skSocket = ::WSASocket(AF\_INET,SOCK\_STREAM,IPPROTO\_IP,

                            NULL,0,WSA\_FLAG\_OVERLAPPED);

......//其它操作

//绑定到IOCP

CreateIoCompletionPort((HANDLE)skSocket,hICP,NULL,0);

由代码就可以看出这步操作就非常的简单了，直接再次调用CreateIoCompletionPort函数即可，只是这次调用的意义就不是创建一个完成端口对象了，而是将一个重叠IO方式的对象句柄绑定到已创建好的完成端口对象上。

至此整个IOCP的基础知识算是介绍完了，作为总结，可以回顾下几个关键步骤：

1、  用CreateIoCompletionPort创建完成端口；

2、  定义IOCP线程池函数，类似消息循环那样写一个“死循环”调用GetQueuedCompletionStatus函数，并编写处理代码；

3、  创建线程；

4、  将重叠IO方式的对象句柄绑定到IOCP上。

只要记住了上面4个关键步骤，那么使用IOCP就基本掌握了。最后作为补充，让我再来讨论下这个核心步骤之外的一些附带的步骤。

现在假设我们已经创建了一个这样的IOCP线程池，而且这个线程池也工作的非常好了，那么我们该如何与这个线程池中的线程进行交互呢？还有就是我们如何让这个线程池停下来？

其实这个问题可以很简单的来思考，既然IOCP线程池核心的线程函数中有一个类似消息循环的结构，那么是不是也有一个类似PostMessage之类的函数来向其发送消息，从而实现与IOCP线程的交互呢？答案是肯定的，这个函数就是PostQueuedCompletionStatus，现在看到它的名字，你应该已经猜到它的用途了吧？对了，它就是用来向这个类似消息循环的循环中发送自定义的“消息”的，当然，它不是真正的消息，而是一个模拟的“完成状态”。这个函数的原型如下：

BOOL WINAPI PostQueuedCompletionStatus(

  \_\_in          HANDLE CompletionPort,

  \_\_in          DWORD dwNumberOfBytesTransferred,

  \_\_in          ULONG\_PTR dwCompletionKey,

  \_\_in          LPOVERLAPPED lpOverlapped

);

它的参数与GetQueuedCompletionStatus类似，其实为了理解上的简单，我们可以认为PostQueuedCompletionStatus的参数就是原样的被copy到了GetQueuedCompletionStatus，怎么调用这个函数就应该可以理解了。通常在需要停止整个IOCP线程池工作时，就可以调用这个函数发送一个特殊的标志，比如设定dwCompletionKey为NULL，并且在自定义lpOverlapped指针结构之后带上一个表示关闭的标志等。这样在线程函数中就可以通过判定这些条件而明确的知道当前线程池需要关闭。当然也可以定义其它的操作扩展码来指定IOCP线程池执行指定的操作。下面的例子代码演示了如何发送一个IO完成状态：

MYOVERLAPPED \*pOL = new MYOVERLAPPED ;

.......//其它初始化代码

pOL->m\_iOpCode = OP\_CLOSE;//指定关闭操作码

.......

PostQueuedCompletionStatus(hICP,0,NULL,(LPOVERLAPPED)pOL);

至此IOCP的基础性的支持算是介绍完了，本篇文章的主要目的是为了让大家理解IOCP的本质和工作原理，为轻松驾驭IOCP这个编程模型打下坚实的基础。最终需要掌握的就是认识到IOCP其实就是一个管理IO操作的自定义线程池这一本质。实际编码时决策性的问题就是理解最大并发数和预创建线程数的意义，并根据实际情况设定一个合理的值。

IOCP+WinSock2新函数打造高性能SOCKET池

在前一篇文章《[WinSock2编程之打造完整的SOCKET池](http://gamebabyrocksun.blog.163.com/blog/static/571534632010115104634310/) 》中，介绍了WinSock2的一些新函数，并重点详细介绍了什么是SOCKET池，有了这个概念，现在就接着展开更深入的讨论。

首先这里要重点重申一下就是，SOCKET池主要指的是使用面向连接的协议的情况下，最常用的就是需要管理大量的TCP连接的时候。常见的就是Web服务器、FTP服务器等。

下面就分步骤的详细介绍如何最终实现SOCKET池。

一、WinSock2环境的初始化：

要使用WinSock2就需要先初始化Socket2.0的环境，不废话，上代码：

WSADATA wd = {0};

int iError = WSAStartup(MAKEWORD(2,0), &wd);

if( 0 != iError )

{//出现错误，最好跟踪看下错误码是多少

       return FALSE;

}

if ( LOBYTE(lpwsaData->wVersion) != 2 )

{//非2.0以上环境 退出了事 可能是可怜的WinCE系统

       WSACleanup();

       return FALSE;

}

最后再不使用WinSock之后都要记得调用一下WSACleanup()这个函数；

二、装载WinSock2函数：

上一篇文章中给出了一个装载WinSock2函数的类，这里分解介绍下装载的具体过程，要提醒的就是，凡是类里面演示了动态装载的函数，最好都像那样动态载入，然后再调用。以免出现上网发帖跪求高手赐教为什么AcceptEx函数无法编译通过等问题。看完这篇文章详细你不会再去发帖找答案了，呵呵呵，好了，上代码：

//定义一个好用的载入函数 摘自CGRSMsSockFun 类

BOOL LoadWSAFun(GUID&funGuid,void\*& pFun)

{//本函数利用参数返回函数指针

       DWORD dwBytes = 0;

       pFun = NULL;

       //随便创建一个SOCKET供WSAIoctl使用 并不一定要像下面这样创建

       SOCKET skTemp = ::WSASocket(AF\_INET,

                     SOCK\_STREAM, IPPROTO\_TCP, NULL,

                     0, WSA\_FLAG\_OVERLAPPED);

       if(INVALID\_SOCKET == skTemp)

       {//通常表示没有正常的初始化WinSock环境

              return FALSE;

       }

       ::WSAIoctl(skTemp, SIO\_GET\_EXTENSION\_FUNCTION\_POINTER,

                     &funGuid,sizeof(funGuid),&pFun,

                     sizeof(pFun), &dwBytes, NULL,NULL);

       ::closesocket(skTemp);

       return NULL != pFun;

}

//演示如何动态载入AcceptEx函数

......

LPFN\_ACCEPTEX pfnAcceptEx; //首先声明函数指针

GUID GuidAcceptEx = WSAID\_ACCEPTEX;

LoadWSAFun(GuidAcceptEx,(void\*&)pfnAcceptEx); //载入

......

//使用丰富的参数调用

......

pfnAcceptEx(sListenSocket,sAcceptSocket,lpOutputBuffer,

              dwReceiveDataLength,dwLocalAddressLength,dwRemoteAddressLength,

lpdwBytesReceived,lpOverlapped);

              //或者：

              SOCKET skAccept = ::WSASocket(AF\_INET,SOCK\_STREAM,IPPROTO\_TCP,

                                                 NULL, 0,WSA\_FLAG\_OVERLAPPED);

              PVOID pBuf = new BYTE[sizeof(sockaddr\_in) + 16];

              pfnAcceptEx(skServer, skAccept,pBuf,

0,//将接收缓冲置为0,令AcceptEx直接返回,防止拒绝服务攻击

                     sizeof(sockaddr\_in) + 16, sizeof(sockaddr\_in) + 16, NULL,

                                                 (LPOVERLAPPED)pAcceptOL);

......

       以上是一个简单的演示，如何动态载入一个WinSock2扩展函数，并调用之，其它函数的详细例子可以看前一篇文章中CGRSMsSockFun类的实现部分。如果使用CGRSMsSockFun 类的话当然更简单，像下面这样调用即可：

              CGRSMsSockFun MsSockFun；

              MsSockFun.AcceptEx(skServer, skAccept,pBuf,

0,//将接收缓冲置为0,令AcceptEx直接返回,防止拒绝服务攻击

                     sizeof(sockaddr\_in) + 16, sizeof(sockaddr\_in) + 16, NULL,

                     (LPOVERLAPPED)pAcceptOL);

如果要使用这个类，那么需要一些修改，主要是异常处理部分，自己注释掉，或者用其它异常代替掉即可，这个对于有基础的读者来说不是什么难事。

三、定义OVERLAPPED结构：

要想“IOCP”就要自定义OVERLAPPED，这是彻底玩转IOCP的不二法门，可以这么说：“江湖上有多少种自定义的OVERLAPPED派生结构体，就有多少种IOCP的封装！”

OVERLAPPED本身是Windows IOCP机制内部需要的一个结构体，主要用于记录每个IO操作的“完成状态”，其内容对于调用者来说是没有意义的，但是很多时候我们把它当做一个“火车头”，因为它可以方便的把每个IO操作的相关数据简单的“从调用处运输到完成回调函数中”，这是一个非常有用的特性，哪么如何让这个火车头发挥运输的作用呢？其实很简单：让它成为一个自定义的更大结构体的第一个成员。然后用强制类型转换，将自定义的结构体转换成OVERLAPPED指针即可。当然不一定非要是新结构体的第一个成员，也可以是任何第n个成员，这时使用VC头文件中预定义的一个宏CONTAINING\_RECORD再反转回来即可。

说到这里一些C++基础差一点的读者估计已经很头晕了，更不知道我再说什么，那么我就将好人做到底吧，来解释下这个来龙去脉。

首先就以我们将要使用的AcceptEx函数为例子看看它的原型吧（知道孙悟空的火眼金睛用来干嘛的吗？就是用来看原型的，哈哈哈）：

BOOL AcceptEx(

  \_\_in          SOCKET sListenSocket,

  \_\_in          SOCKET sAcceptSocket,

  \_\_in          PVOID lpOutputBuffer,

  \_\_in          DWORD dwReceiveDataLength,

  \_\_in          DWORD dwLocalAddressLength,

  \_\_in          DWORD dwRemoteAddressLength,

  \_\_out         LPDWORD lpdwBytesReceived,

  \_\_in          LPOVERLAPPED lpOverlapped

);

注意最后一个参数，是一个OVERLAPPED结构体的指针（LP的意思是Long Pointer,即指向32位地址长指针，注意不是“老婆”拼音的缩写），本身这个参数的意思就是分配一块OVERLAPPED大小的内存，在IOCP调用方式下传递给AcceptEx函数用，调用者不用去关心里面的任何内容，而在完成过程中（很多时候是另一个线程中的事情了），通常调用GetQueuedCompletionStatus函数后，会再次得到这个指针，接着让我们也看看它的原型：

BOOL WINAPI GetQueuedCompletionStatus(

  \_\_in          HANDLE CompletionPort,

  \_\_out         LPDWORD lpNumberOfBytes,

  \_\_out         PULONG\_PTR lpCompletionKey,

  \_\_out         LPOVERLAPPED\* lpOverlapped,

  \_\_in          DWORD dwMilliseconds

);

注意这里的LPOVERLAPPED多了一个\*变成了指针的指针，并且前面的说明很清楚Out！很明白了吧，不明白就真的Out了。这里就可以重新得到调用AcceptEx传入的LPOVERLAPPED指针，也就是得到了这个“火车头”，因为只是一个指针，并没有详细的限定能有多大，所以可以在火车头的后面放很多东西。

再仔细观察GetQueuedCompletionStatus函数的参数，会发现，这时只能知道一个IO操作结束了，但是究竟是哪个操作结束了，或者是哪个SOCKET句柄上的操作结束了，并没有办法知道。通常这个信息非常重要，因为只有在IO操作实际完成之后才能释放发送或接收等操作的缓冲区。

这些信息可以定义成如下的一个扩展OVERLAPPED结构：

struct MYOVERLAPPED

{

    OVERLAPPED m\_ol;

    int                    m\_iOpType;

//操作类型 0=AcceptEx 1=DisconnectEx       2=ConnectEx 3=WSARecv等等

    SOCKET          m\_skServer;       //服务端SOCKET

    SOCKET         m\_skClient;        //客户端SOCKET

    LPVOID           m\_pBuf;            //本次IO操作的缓冲指针

    ......                                           //其它需要的信息

};

使用时：

MYOVERLAPPED\* pMyOL = new MYOVERLAPPED;

ZeroMemory(pMyOL,sizeof(MYOVERLAPPED));

pMyOL->m\_iOpType = 0;        //AcceptEx操作

pMyOL->m\_skServer = skServer;

pMyOL->m\_skClient = skClient;

BYTE\* pBuf = new BYTE[256];//一个缓冲

.................. //朝缓冲中写入东西

pMyOL->m\_pBuf = pBuf;

...............//其它的代码

AcceptEx(skServer, skClient,pBuf,

0,//将接收缓冲置为0,令AcceptEx直接返回

256,256,NULL,(LPOVERLAPPED)pMyOL));//注意最后这个强制类型转换

       在完成过程回调线程函数中，这样使用：

UINT CALLBACK Client\_IOCPThread(void\* pParam)

       {//IOCP线程函数

              .....................

              DWORD dwBytesTrans = 0;

              DWORD dwPerData = 0;

              LPOVERLAPPED lpOverlapped = NULL;

              while(1)

              {//又见死循环 呵呵呵

                     BOOL bRet = GetQueuedCompletionStatus(

                            pThis->m\_IOCP,&dwBytesTrans,&dwPerData,

                            &lpOverlapped,INFINITE);

                     if( NULL == lpOverlapped )

                     {//没有真正的完成

                            SleepEx(20,TRUE);//故意置成可警告状态

                            continue;

                     }

                     //找回“火车头”以及后面的所有东西

                     MYOVERLAPPED\*  pOL = CONTAINING\_RECORD(lpOverlapped

, MYOVERLAPPED, m\_ol);

                     switch(pOL->m\_iOpType)

{

case 0: //AcceptEx结束

{//有链接进来了 SOCKET句柄就是 pMyOL->m\_skClient

}

break;

............................

}

........................

} //end while

........................

       }//end fun

至此，关于这个“火车头”如何使用，应该是看明白了，其实就是从函数传入，又由函数返回。只不过其间可能已经转换了线程环境，是不同的线程了。

这里再补充一个AcceptEx容易被遗漏的一个细节问题，那就是在AcceptEx完成返回之后，如下在那个连入的客户端SOCKET上调用一下：

       int nRet = ::setsockopt(

              pOL->m\_skClient,SOL\_SOCKET,SO\_UPDATE\_ACCEPT\_CONTEXT,

              (char \*)&pOL->m\_skServer,sizeof(SOCKET));

这样才可以继续在这个代表客户端连接的pOL->m\_skClient上继续调用WSARecv和WSASend。

另外，在AcceptEx完成之后，通常可以用：

LPSOCKADDR addrHost = NULL;      //服务端地址

LPSOCKADDR addrClient = NULL;     //客户端地址

int lenHost = 0;

int lenClient = 0;

GetAcceptExSockaddrs(

       pOL->m\_pBuf,0,sizeof(sockaddr\_in) + 16,sizeof(sockaddr\_in) + 16,

       (LPSOCKADDR\*) &addrHost,&lenHost,(LPSOCKADDR\*) &addrClient,&lenClient);

这样来得到连入的客户端地址，以及连入的服务端地址，通常这个地址可以和这个客户端的SOCKET绑定在一起用map或hash表保存，方便查询，就不用再调用那个getpeername得到客户端的地址了。要注意的是GetAcceptExSockaddrs也是一个WinSock2扩展函数，专门配合AcceptEx使用的，需要像AcceptEx那样动态载入一下，然后再调用，详情请见前一篇文章中的CGRSMsSockFun类。

至此AcceptEx算讨论完整了，OVERLAPPED的派生定义也讲完了，让我们继续下一步。

四、编写线程池回调函数：

在讨论扩展定义OVERLAPPED结构体时，给出了非线程池版的线程函数的大概框架，也就是传统IOCP使用的自建线程使用方式，这种方式要自己创建完成端口句柄，自己将SOCKET句柄绑定到完成端口，这里就不在赘述，主要介绍下调用BindIoCompletionCallback函数时，应如何编写这个线程池的回调函数，其实它与前面那个线程函数是很类似的。先来看看回调函数长个什么样子：

VOID CALLBACK FileIOCompletionRoutine(

  [in]                 DWORD dwErrorCode,

  [in]                 DWORD dwNumberOfBytesTransfered,

  [in]                 LPOVERLAPPED lpOverlapped

);

第一个参数就是一个错误码，如果是0恭喜你，操作一切ok，如果有错也不要慌张，前一篇文章中已经介绍了如何翻译和看懂这个错误码。照着做就是了。

第二个参数就是说这次IO操作一共完成了多少字节的数据传输任务，这个字段有个特殊含义，如果你发现一个Recv操作结束了，并且这个参数为0，那么就是说，客户端断开了连接（注意针对的是TCP方式，整个SOCKET池就是为TCP方式设计的）。如果这个情况发生了，在SOCKET池中就该回收这个SOCKET句柄。

第三个参数现在不用多说了，立刻就知道怎么用它了。跟刚才调用GetQueuedCompletionStatus函数得到的指针是一个含义。

下面就来看一个实现这个回调的例子：

VOID CALLBACK MyIOCPThread(DWORD dwErrorCode

,DWORD dwBytesTrans,LPOVERLAPPED lpOverlapped)

       {//IOCP回调函数

              .....................

              if( NULL == lpOverlapped )

              {//没有真正的完成

                     SleepEx(20,TRUE);//故意置成可警告状态

                     return;

              }

              //找回“火车头”以及后面的所有东西

              MYOVERLAPPED\*  pOL = CONTAINING\_RECORD(lpOverlapped

, MYOVERLAPPED, m\_ol);

              switch(pOL->m\_iOpType)

{

case 0: //AcceptEx结束

{//有链接进来了 SOCKET句柄就是 pMyOL->m\_skClient

}

break;

............................

}

........................

       }//end fun

看起来很简单吧？好像少了什么？对了那个该死的循环，这里不用了，因为这个是由线程池回调的一个函数而已，线程的活动状态完全由系统内部控制，只管认为只要有IO操作完成了，此函数就会被调用。这里关注的焦点就完全的放到了完成之后的操作上，而什么线程啊，完成端口句柄啊什么的就都不需要了（甚至可以忘记）。

这里要注意一个问题，正如在《IOCP编程之“双节棍”》中提到的，这个函数执行时间不要过长，否则会出现掉线啊，连接不进来啊等等奇怪的事情。

另一个要注意的问题就是，这个函数最好套上结构化异常处理，尽可能的多拦截和处理异常，防止系统线程池的线程因为你糟糕的回调函数而壮烈牺牲，如果加入了并发控制，还要注意防止死锁，不然你的服务器会“死”的很难看。

理论上来说，你尽可以把这个函数看做一个与线程池函数等价的函数，只是他要尽可能的“短”（指执行时间）而紧凑（结构清晰少出错）。

最后，回调函数定义好了，就可以调用BindIoCompletionCallback函数，将一个SOCKET句柄丢进完成端口的线程池了：

BindIoCompletionCallback((HANDLE)skClient,MyIOCPThread,0);

注意最后一个参数到目前为止，你就传入0吧。这个函数的神奇就是不见了CreateIoCompletionPort的调用，不见了CreateThread的调用，不见了GetQueuedCompletionStatus等等的调用，省去了n多繁琐且容易出错的步骤，一个函数就全部搞定了。

五、服务端调用：

以上的所有步骤在完全理解后，最终让我们看看SOCKET池如何实现之。

1、按照传统，要先监听到某个IP的指定端口上：

SOCKADDR\_IN    saServer = {0};

//创建监听Socket

SOCKET skServer = ::WSASocket(AF\_INET, SOCK\_STREAM, IPPROTO\_IP

, NULL, 0, WSA\_FLAG\_OVERLAPPED);

//把监听SOCKET扔进线程池，这个可以省略               ::BindIoCompletionCallback((HANDLE)skServer,MyIOCPThread, 0);

//必要时打开SO\_REUSEADDR属性，重新绑定到这个监听地址

BOOL   bReuse=TRUE;                  ::setsockopt(m\_skServer,SOL\_SOCKET,SO\_REUSEADDR

,(LPCSTR)&bReuse,sizeof(BOOL));

saServer.sin\_family = AF\_INET;

saServer.sin\_addr.s\_addr = INADDR\_ANY;

// INADDR\_ANY这个值的魅力是监听所有本地IP的相同端口

saServer.sin\_port = htons(80);      //用80得永生

::bind(skServer,(LPSOCKADDR)&saServer,sizeof(SOCKADDR\_IN));

//监听，队列长为默认最大连接SOMAXCONN

listen(skServer, SOMAXCONN);

2、就是发出一大堆的AcceptEx调用：

for(UINT i = 0; i < 1000; i++)

{//调用1000次

//创建与客户端通讯的SOCKET，注意SOCKET的创建方式

skAccept = ::WSASocket(AF\_INET,

                                              SOCK\_STREAM,

                                              IPPROTO\_TCP,

                                              NULL,

                                              0,

                                              WSA\_FLAG\_OVERLAPPED);

//丢进线程池中

BindIoCompletionCallback((HANDLE)skAccept ,MyIOCPThread,0);

//创建一个自定义的OVERLAPPED扩展结构，使用IOCP方式调用

pMyOL= new MYOVERLAPPED;

pMyOL->m\_iOpType = 0;        //AcceptEx操作

pMyOL->m\_skServer = skServer;

pMyOL->m\_skClient = skClient;

BYTE\* pBuf = new BYTE[256];//一个缓冲

ZeroMemory(pBuf,256\*sizeof(BYTE));

pMyOL->m\_pBuf = pBuf;

//发出AcceptEx调用

//注意将AcceptEx函数接收连接数据缓冲的大小设定成了0

//这将导致此函数立即返回，虽然与不设定成0的方式而言，

//这导致了一个较低下的效率，但是这样提高了安全性

//所以这种效率牺牲是必须的

AcceptEx(skServer, skAccept,pBuf,

    0,//将接收缓冲置为0,令AcceptEx直接返回,防止拒绝服务攻击

    256,256,NULL,(LPOVERLAPPED)pMyOL);

}

这样就有1000个AcceptEx在提前等着客户端的连接了，即使1000个并发连接也不怕了，当然如果再BT点那么就放1w个，什么你要放2w个？那就要看看你的这个IP段的端口还够不够了，还有你的系统内存够不够用。一定要注意同一个IP地址上理论上端口最大值是65535，也就是6w多个，这个要合理的分派，如果并发管理超过6w个以上的连接时，怎么办呢？那就再插块网卡租个新的IP，然后再朝那个IP端绑定并监听即可。因为使用了INADDR\_ANY，所以一监听就是所有本地IP的相同端口，如果服务器的IP有内外网之分，为了安全和区别起见可以明确指定监听哪个IP，单IP时就要注意本IP空闲端口的数量问题了。

3、AcceptEx返回后，也就是线程函数中，判定是AcceptEx操作返回后，首先需要的调用就是：

GetAcceptExSockaddrs(pBuf,0,sizeof(sockaddr\_in) + 16,

       sizeof(sockaddr\_in) + 16,(LPSOCKADDR\*) &addrHost,&lenHost,

       (LPSOCKADDR\*) &addrClient,&lenClient);

int nRet = ::setsockopt(pOL->m\_skClient, SOL\_SOCKET,

       SO\_UPDATE\_ACCEPT\_CONTEXT,(char \*)&m\_skServer,sizeof(m\_skServer));

之后就可以WSASend或者WSARecv了。

       4、这些调用完后，就可以在这个m\_skClient上收发数据了，如果收发数据结束或者IO错误，那么就回收SOCKET进入SOCKET池：

       DisconnectEx(m\_skClient,&pData->m\_ol, TF\_REUSE\_SOCKET, 0);

       5、当DisconnectEx函数完成操作之后，在回调的线程函数中，像下面这样重新让这个SOCKET进入监听状态，等待下一个用户连接进来，至此组建SOCKET池的目的就真正达到了:

//创建一个自定义的OVERLAPPED扩展结构，使用IOCP方式调用

pMyOL= new MYOVERLAPPED;

pMyOL->m\_iOpType = 0;        //AcceptEx操作

pMyOL->m\_skServer = skServer;

pMyOL->m\_skClient = skClient;

BYTE\* pBuf = new BYTE[256];//一个缓冲

ZeroMemory(pBuf,256\*sizeof(BYTE));

pMyOL->m\_pBuf = pBuf;

AcceptEx(skServer, skClient,pBuf , 0,256,256,NULL,

    (LPOVERLAPPED)pMyOL);

//注意在这个SOCKET被重新利用后，后面的再次捆绑到完成端口的操作会返回一个已设置//的错误，这个错误直接被忽略即可

         ::BindIoCompletionCallback((HANDLE)skClient,Server\_IOCPThread, 0);

       至此服务端的线程池就算搭建完成了，这个SOCKET池也就是围绕AcceptEx和DisconnectEx展开的，而创建操作就全部都在服务启动的瞬间完成，一次性投递一定数量的SOCKET进入SOCKET池即可，这个数量也就是通常所说的最大并发连接数，你喜欢多少就设置多少吧，如果连接多数量就大些，如果IO操作多，连接断开请求不多就少点，剩下就是调试了。

六、客户端调用：

1、  主要是围绕利用ConnectEx开始调用：

SOCKET skConnect = ::WSASocket(AF\_INET,SOCK\_STREAM,IPPROTO\_IP,

                            NULL,0,WSA\_FLAG\_OVERLAPPED);

//把SOCKET扔进IOCP

BindIoCompletionCallback((HANDLE)skConnect,MyIOCPThread,0);

//本地随便绑个端口

SOCKADDR\_IN LocalIP = {};

LocalIP.sin\_family = AF\_INET;

LocalIP.sin\_addr.s\_addr = INADDR\_ANY;

LocalIP.sin\_port = htons( (short)0 );    //使用0让系统自动分配

int result =::bind(skConnect,(LPSOCKADDR)&LocalIP,sizeof(SOCKADDR\_IN));

pMyOL= new MYOVERLAPPED;

pMyOL->m\_iOpType = 2;            //ConnectEx操作

pMyOL->m\_skServer = NULL;    //没有服务端的SOCKET

pMyOL->m\_skClient = skConnect;

ConnectEx(skConnect,(const sockaddr\*)pRemoteAddr,sizeof(SOCKADDR\_IN),

       NULL,0,NULL,(LPOVERLAPPED)pOL) )

如果高兴就可以把上面的过程放到循环里面去，pRemoteAddr就是远程服务器的IP和端口，你可以重复连接n多个，然后疯狂下载东西（别说我告诉你的哈，人家的服务器宕机了找你负责）。注意那个绑定一定要有，不然调用会失败的。

2、  接下来就在线程函数中判定是ConnectEx操作，通过判定m\_iOpType == 2就可以知道，然后这样做：

setsockopt( pOL->m\_skClient, SOL\_SOCKET, SO\_UPDATE\_CONNECT\_CONTEXT,

                     NULL, 0 );

然后就是自由的按照需要调用WSASend或者WSARecv。

3、  最后使用和服务端相似的逻辑调用DisconnectEx函数，收回SOCKET并直接再次调用ConnectEx连接到另一服务器或相同的同一服务器即可。

至此客户端的SOCKET池也搭建完成了，创建SOCKET的工作也是在一开始的一次性就完成了，后面都是利用ConnectEx和DisconnectEx函数不断的连接-收发数据-回收-再连接来进行的。客户端的这个SOCKET池可以用于HTTP下载文件的客户端或者FTP下载的服务端（反向服务端）或者客户端，甚至可以用作一个网游的机器人系统，也可以作为一个压力测试的客户端核心的模型。

七、总结和提高：

以上就是比较完整的如何具体实现SOCKET池的全部内容，因为篇幅的原因就不贴全部的代码了，我相信各位看客看完之后心中应该有个大概的框架，并且也可以进行实际的代码编写工作了。可以用纯c来实现也可以用C++来实现。但是这里要说明一点就是DisconnectEx函数和ConnectEx函数似乎只能在XP SP2以上和2003Server以上的平台上使用，对于服务端来说这不是什么问题，但是对于客户端来说，使用SOCKET池时还要考虑一个兼容性问题，不得已还是要放弃在客户端使用SOCKET池。

SOCKET池的全部精髓就在于提前创建一批SOCKET，然后就是不断的重复回收再利用，比起传统的非SOCKET池方式，节省了大量的不断创建和销毁SOCKET对象的内核操作，同时借用IOCP函数AcceptEx、ConnectEx和DisconnectEx等的异步IO完成特性提升了整体性能，非常适合用于一些需要大规模TCP连接管理的场景，如：HTTP Server FTP Server和游戏服务器等。

SOCKET池的本质就是充分的利用了IOCP模型的几乎所有优势，因此要用好SOCKET池就要深入的理解IOCP模型，这是前提。有问题请跟帖讨论。

WinSock2编程之打造完整的SOCKET池

在Winodows平台上，网络编程的主要接口就是WinSock，目前大多数的Windows平台上的WinSock平台已经升级到2.0版，简称为WinSock2。在WinSock2中扩展了很多很有用的Windows味很浓的SOCKET专用API，为Windows平台用户提供高性能的网络编程支持。这些函数中的大多数已经不再是标准的“Berkeley”套接字模型的API了。使用这些函数的代价就是你不能再将你的网络程序轻松的移植到“尤里平台”（我给Unix +Linux平台的简称）下，反过来因为Windows平台支持标准的“Berkeley”套接字模型，所以你可以将大多数尤里平台下的网络应用移植到Windows平台下。

如果不考虑可移植性（或者所谓的跨平台性），而是着重于应用的性能时，尤其是注重服务器性能时，对于Windows的程序，都鼓励使用WinSock2扩展的一些API，更鼓励使用IOCP模型，因为这个模型是目前Windows平台上比较完美的一个高性能IO编程模型，它不但适用于SOCKET编程，还适用于读写硬盘文件，读写和管理命名管道、邮槽等等。如果再结合Windows线程池，IOCP几乎可以利用当今硬件所有可能的新特性（比如多核，DMA，高速总线等等），本身具有先天的扩展性和可用性。

今天讨论的重点就是SOCKET池。很多VC程序员也许对SOCKET池很陌生，也有些可能很熟悉，那么这里就先讨论下这个概念。

在Windows平台上SOCKET实际上被视作一个内核对象的句柄，很多Windows API在支持传统的HANDLE参数的同时也支持SOCKET，比如有名的CreateIoCompletionPort就支持将SOCKET句柄代替HANDLE参数传入并调用。熟悉Windows内核原理的读者，立刻就会发现，这样的话，我们创建和销毁一个SOCKET句柄，实际就是在系统内部创建了一个内核对象，对于Windows来说这牵扯到从Ring3层到Ring0层的耗时操作，再加上复杂的安全审核机制，实际创建和销毁一个SOCKET内核对象的成本还是蛮高的。尤其对于一些面向连接的SOCKET应用，服务端往往要管理n多个代表客户端通信的SOCKET对象，而且因为客户的变动性，主要面临的大量操作除了一般的收发数据，剩下的就是不断创建和销毁SOCKET句柄，对于一个频繁接入和断开的服务器应用来说，创建和销毁SOCKET的性能代价立刻就会体现出来，典型的例如WEB服务器程序，就是一个需要频繁创建和销毁SOCKET句柄的SOCKET应用。这种情况下我们通常都希望对于断开的SOCKET对象，不是简单的“销毁”了之（很多时候“断开”的含义不一定就等价于“销毁”，可以仔细思考一下），更多时候希望能够重用这个SOCKET对象，这样我们甚至可以事先创建一批SOCKET对象组成一个“池”，在需要的时候“重用”其中的SOCKET对象，不需要的时候将SOCKET对象重新丢入池中即可，这样就省去了频繁创建销毁SOCKET对象的性能损失。在原始的“Berkeley”套接字模型中，想做到这点是没有什么办法的。而幸运的是在Windows平台上，尤其是支持WinSock2的平台上，已经提供了一套完整的API接口用于支持SOCKET池。

对于符合以上要求的SOCKET池，首先需要做到的就是对SOCKET句柄的“回收”，因为创建函数无论在那个平台上都是现成的，而最早能够实现这个功能的WinSock函数就是TransmitFile，如果代替closesocket函数像下面这样调用就可以“回收”一个SOCKET句柄，而不是销毁：（注意“回收”这个功能对于TransmitFile函数来说只是个“副业”。）

TransmitFile(hSocket,NULL,0,0,NULL,NULL,TF\_DISCONNECT | TF\_REUSE\_SOCKET );

注意上面函数的最后一个参数，使用了标志TF\_DISCONNECT和TF\_REUSE\_SOCKET，第一个值表示断开，第二个值则明确的表示“重用”实际上也就是回收这个SOCKET，经过这个处理的SOCKET句柄，就可以直接再用于connect等操作，但是此时我们会发现，这个回收来的SOCKET似乎没什么用，因为其他套接字函数没法直接利用这个回收来的SOCKET句柄。

这时就要WinSock2的一组专用API上场了。我将它们按传统意义上的服务端和客户端分为两组：

一、         服务端：

SOCKET WSASocket(

  \_\_in          int af,

  \_\_in          int type,

  \_\_in          int protocol,

  \_\_in          LPWSAPROTOCOL\_INFO lpProtocolInfo,

  \_\_in          GROUP g,

  \_\_in          DWORD dwFlags

);

BOOL AcceptEx(

  \_\_in          SOCKET sListenSocket,

  \_\_in          SOCKET sAcceptSocket,

  \_\_in          PVOID lpOutputBuffer,

  \_\_in          DWORD dwReceiveDataLength,

  \_\_in          DWORD dwLocalAddressLength,

  \_\_in          DWORD dwRemoteAddressLength,

  \_\_out         LPDWORD lpdwBytesReceived,

  \_\_in          LPOVERLAPPED lpOverlapped

);

BOOL DisconnectEx(

  \_\_in          SOCKET hSocket,

  \_\_in          LPOVERLAPPED lpOverlapped,

  \_\_in          DWORD dwFlags,

  \_\_in          DWORD reserved

);

二、         客户端：

SOCKET WSASocket(

  \_\_in          int af,

  \_\_in          int type,

  \_\_in          int protocol,

  \_\_in          LPWSAPROTOCOL\_INFO lpProtocolInfo,

  \_\_in          GROUP g,

  \_\_in          DWORD dwFlags

);

BOOL PASCAL ConnectEx(

  \_\_in          SOCKET s,

  \_\_in          const struct sockaddr\* name,

  \_\_in          int namelen,

  \_\_in\_opt      PVOID lpSendBuffer,

  \_\_in          DWORD dwSendDataLength,

  \_\_out         LPDWORD lpdwBytesSent,

  \_\_in          LPOVERLAPPED lpOverlapped

);

BOOL DisconnectEx(

  \_\_in          SOCKET hSocket,

  \_\_in          LPOVERLAPPED lpOverlapped,

  \_\_in          DWORD dwFlags,

  \_\_in          DWORD reserved

);

注意观察这些函数，似乎和传统的“Berkeley”套接字模型中的一些函数“大同小异”，其实仔细观察他们的参数，就已经可以发现一些调用他们的“玄机”了。

首先我们来看AcceptEx函数，与accept函数不同，它需要两个SOCKET句柄作为参数，头一个参数的含义与accept函数的相同，而第二个参数的意思就是accept函数返回的那个代表与客户端通信的SOCKET句柄，在传统的accept内部，实际在返回那个代表客户端的SOCKET时，是在内部调用了一个SOCKET的创建动作，先创建这个SOCKET然后再“accept”让它变成代表客户端连接的SOCKET，而AcceptEx函数就在这里“扩展”（实际上是“阉割”才对）accept函数，省去了内部那个明显的创建SOCKET的动作，而将这个创建动作交给最终的调用者自己来实现。AcceptEx要求调用者创建好那个sAcceptSocket句柄然后传进去，这时我们立刻发现，我们回收的那个SOCKET是不是也可以传入呢？答案是肯定的，我们就是可以利用这个函数传入那个“回收”来的SOCKET句柄，最终实现服务端的SOCKET重用。

这里需要注意的就是，AcceptEx函数必须工作在非阻塞的IOCP模型下，同时即使AcceptEx函数返回了，也不代表客户端连接进来或者连接成功了，我们必须依靠它的“完成通知”才能知道这个事实，这也是AcceptEx函数区别于accept这个阻塞方式函数的最大之处。通常可以利用AcceptEx的非阻塞特性和IOCP模型的优点，一次可以“预先”发出成千上万个AcceptEx调用，“等待”客户端的连接。对于习惯了accept阻塞方式的程序员来说，理解AcceptEx的工作方式还是需要费一些周折的。下面的例子就演示了如何一次调用多个AcceptEx：

//批量创建SOCKET，并调用对应的AcceptEx

for(UINT i = 0; i < 1000; i++)

{//调用1000次

//创建与客户端通讯的SOCKET，注意SOCKET的创建方式

skAccept = ::WSASocket(AF\_INET,

                   SOCK\_STREAM,

                   IPPROTO\_TCP,

                   NULL,

                   0,

                   WSA\_FLAG\_OVERLAPPED);

if (INVALID\_SOCKET == skAccept)

{

    throw CGRSException((DWORD)WSAGetLastError());

}

//创建一个自定义的OVERLAPPED扩展结构，使用IOCP方式调用

pAcceptOL = new CGRSOverlappedData(GRS\_OP\_ACCEPT

,this,skAccept,NULL);

pAddrBuf = pAcceptOL->GetAddrBuf();

//4、发出AcceptEx调用

//注意将AcceptEx函数接收连接数据缓冲的大小设定成了0，这将导致此函数立即返回，虽然与

//不设定成0的方式而言，这导致了一个较低下的效率，但是这样提高了安全性，所以这种效率

//牺牲是必须的

if(!AcceptEx(m\_skServer,

                   skAccept,

                   pAddrBuf->m\_pBuf,

                   0,//将接收缓冲置为0,令AcceptEx直接返回,防止拒绝服务攻击

                   GRS\_ADDRBUF\_SIZE,

                   GRS\_ADDRBUF\_SIZE,

                   NULL,

                   (LPOVERLAPPED)pAcceptOL))

{

int iError = WSAGetLastError();

if( ERROR\_IO\_PENDING != iError

     && WSAECONNRESET != iError )

{

     if(INVALID\_SOCKET != skAccept)

     {

         ::closesocket(skAccept);

         skAccept = INVALID\_SOCKET;

     }

     if( NULL != pAcceptOL)

     {

             GRS\_ISVALID(pAcceptOL,sizeof(CGRSOverlappedData));

delete pAcceptOL;

     pAcceptOL = NULL;

     }

  }

}

}

以上的例子只是简单的演示了AcceptEx的调用，还没有涉及到真正的“回收重用”这个主题，那么下面的例子就演示了如何重用一个SOCKET句柄：

if(INVALID\_SOCKET == skClient)

{

throw CGRSException(\_T("SOCKET句柄是无效的！"));

}

OnPreDisconnected(skClient,pUseData,0);

CGRSOverlappedData\*pData

= new GRSOverlappedData(GRS\_OP\_DISCONNECTEX

,this,skClient,pUseData);

//回收而不是关闭后再创建大大提高了服务器的性能

DisconnectEx(skClient,&pData->m\_ol,TF\_REUSE\_SOCKET,0);

......

      //在接收到DisconnectEx函数的完成通知之后，我们就可以重用这个SOCKET了

CGRSAddrbuf\*pBuf = NULL;

pNewOL = new CGRSOverlappedData(GRS\_OP\_ACCEPT

,this,skClient,pUseData);

pBuf = pNewOL->GetAddrBuf();

//把这个回收的SOCKET重新丢进连接池

if(!AcceptEx(m\_skServer,skClient,pBuf->m\_pBuf,

                 0,//将接收缓冲置为0,令AcceptEx直接返回,防止拒绝服务攻击

                 GRS\_ADDRBUF\_SIZE, GRS\_ADDRBUF\_SIZE,

                 NULL,(LPOVERLAPPED)pNewOL))

{

int iError = WSAGetLastError();

    if( ERROR\_IO\_PENDING != iError

        && WSAECONNRESET != iError )

    {

        throw CGRSException((DWORD)iError);

     }

}

//注意在这个SOCKET被重新利用后，重新与IOCP绑定一下，该操作会返回一个已设置的错误，这个错误直接被忽略即可

::BindIoCompletionCallback((HANDLE)skClient

,Server\_IOCPThread, 0);

至此回收重用SOCKET的工作也就结束了，以上的过程实际理解了IOCP之后就比较好理解了，例子的最后我们使用了BindIoCompletionCallback函数重新将SOCKET丢进了IOCP线程池中，实际还可以再次使用CreateIoCompletionPort函数达到同样的效果，这里列出这一步就是告诉大家，不要忘了再次绑定一下完成端口和SOCKET。

    对于客户端来说，可以使用ConnectEx函数来代替connect函数，与AcceptEx函数相同，ConnectEx函数也是以非阻塞的IOCP方式工作的，唯一要注意的就是在WSASocket调用之后，在ConnectEx之前要调用一下bind函数，将SOCKET提前绑定到一个本地地址端口上，当然回收重用之后，就无需再次绑定了，这也是ConnectEx较之connect函数高效的地方之一。

   与AcceptEx函数类似，也可以一次发出成千上万个ConnectEx函数的调用，可以连接到不同的服务器，也可以连接到相同的服务器，连接到不同的服务器时，只需提供不同的sockaddr即可。

    通过上面的例子和讲解，大家应该对SOCKET池概念以及实际的应用有个大概的了解了，当然核心仍然是理解了IOCP模型，否则还是寸步难行。

在上面的例子中，回收SOCKET句柄主要使用了DisconnectEx函数，而不是之前介绍的TransmitFile函数，为什么呢？因为TransmitFile函数在一些情况下会造成死锁，无法正常回收SOCKET，毕竟不是专业的回收重用SOCKET函数，我就遇到过好几次死锁，最后偶然的发现了DisconnectEx函数这个专用的回收函数，调用之后发现比TransmitFile专业多了，而且不管怎样都不会死锁。

最后需要补充的就是这几个函数的调用方式，不能像传统的SOCKET API那样直接调用它们，而需要使用一种间接的方式来调用，尤其是AcceptEx和DisconnectEx函数，下面给出了一个例子类，用于演示如何动态载入这些函数并调用之：

class CGRSMsSockFun

{

public:

CGRSMsSockFun(SOCKET skTemp = INVALID\_SOCKET)

{

     if( INVALID\_SOCKET != skTemp )

     {

       LoadAllFun(skTemp);

      }

}

public:

virtual ~CGRSMsSockFun(void)

{

}

protected:

BOOL LoadWSAFun(SOCKET& skTemp,GUID&funGuid,void\*&pFun)

{

     DWORD dwBytes = 0;

     BOOL bRet = TRUE;

     pFun = NULL;

     BOOL bCreateSocket = FALSE;

     try

     {

       if(INVALID\_SOCKET == skTemp)

       {

          skTemp = ::WSASocket(AF\_INET,SOCK\_STREAM,

             IPPROTO\_TCP,NULL,0,WSA\_FLAG\_OVERLAPPED);

bCreateSocket = (skTemp != INVALID\_SOCKET);

       }

if(INVALID\_SOCKET == skTemp)

       {

          throw CGRSException((DWORD)WSAGetLastError());

       }

       if(SOCKET\_ERROR == ::WSAIoctl(skTemp,

                SIO\_GET\_EXTENSION\_FUNCTION\_POINTER,

                &funGuid,sizeof(funGuid),

                &pFun,sizeof(pFun),&dwBytes,NULL,

                NULL))

       {

             pFun = NULL;

             throw CGRSException((DWORD)WSAGetLastError());

       }

  }

  catch(CGRSException& e)

  {

      if(bCreateSocket)

      {

        ::closesocket(skTemp);

      }

  }

  return NULL != pFun;

}

protected:

LPFN\_ACCEPTEX m\_pfnAcceptEx;

LPFN\_CONNECTEX m\_pfnConnectEx;

LPFN\_DISCONNECTEX m\_pfnDisconnectEx;

LPFN\_GETACCEPTEXSOCKADDRS m\_pfnGetAcceptExSockaddrs;

LPFN\_TRANSMITFILE m\_pfnTransmitfile;

LPFN\_TRANSMITPACKETS m\_pfnTransmitPackets;

LPFN\_WSARECVMSG m\_pfnWSARecvMsg;

protected:

BOOL LoadAcceptExFun(SOCKET &skTemp)

{

     GUID GuidAcceptEx = WSAID\_ACCEPTEX;

     return LoadWSAFun(skTemp,GuidAcceptEx

,(void\*&)m\_pfnAcceptEx);

}

BOOL LoadConnectExFun(SOCKET &skTemp)

{

     GUID GuidAcceptEx = WSAID\_CONNECTEX;

     return LoadWSAFun(skTemp,GuidAcceptEx

,(void\*&)m\_pfnConnectEx);

}

BOOL LoadDisconnectExFun(SOCKET&skTemp)

{

     GUID GuidDisconnectEx = WSAID\_DISCONNECTEX;

     return LoadWSAFun(skTemp,GuidDisconnectEx

,(void\*&)m\_pfnDisconnectEx);

}

BOOL LoadGetAcceptExSockaddrsFun(SOCKET &skTemp)

{

     GUID GuidGetAcceptExSockaddrs

= WSAID\_GETACCEPTEXSOCKADDRS;

     return LoadWSAFun(skTemp,GuidGetAcceptExSockaddrs

,(void\*&)m\_pfnGetAcceptExSockaddrs);

}

BOOL LoadTransmitFileFun(SOCKET&skTemp)

{

     GUID GuidTransmitFile = WSAID\_TRANSMITFILE;

     return LoadWSAFun(skTemp,GuidTransmitFile

,(void\*&)m\_pfnTransmitfile);

}

BOOL LoadTransmitPacketsFun(SOCKET&skTemp)

{

     GUID GuidTransmitPackets = WSAID\_TRANSMITPACKETS;

     return LoadWSAFun(skTemp,GuidTransmitPackets

,(void\*&)m\_pfnTransmitPackets);

}

BOOL LoadWSARecvMsgFun(SOCKET&skTemp)

{

     GUID GuidTransmitPackets = WSAID\_TRANSMITPACKETS;

     return LoadWSAFun(skTemp,GuidTransmitPackets

,(void\*&)m\_pfnWSARecvMsg);

}

public:

BOOL LoadAllFun(SOCKET skTemp)

{//注意这个地方的调用顺序，是根据服务器的需要，并结合了表达式副作用

  //而特意安排的调用顺序

  return (LoadAcceptExFun(skTemp) &&

             LoadGetAcceptExSockaddrsFun(skTemp) &&

             LoadTransmitFileFun(skTemp) &&

             LoadTransmitPacketsFun(skTemp) &&

             LoadDisconnectExFun(skTemp) &&

             LoadConnectExFun(skTemp) &&

             LoadWSARecvMsgFun(skTemp));

}

public:

GRS\_FORCEINLINE BOOL AcceptEx (

          SOCKET sListenSocket,

          SOCKET sAcceptSocket,

          PVOID lpOutputBuffer,

          DWORD dwReceiveDataLength,

          DWORD dwLocalAddressLength,

          DWORD dwRemoteAddressLength,

          LPDWORD lpdwBytesReceived,

          LPOVERLAPPED lpOverlapped

          )

{

     GRS\_ASSERT(NULL != m\_pfnAcceptEx);

     return m\_pfnAcceptEx(sListenSocket,

             sAcceptSocket,lpOutputBuffer,

             dwReceiveDataLength,dwLocalAddressLength,

             dwRemoteAddressLength,lpdwBytesReceived,

             lpOverlapped);

}

GRS\_FORCEINLINE BOOL ConnectEx(

          SOCKET s,const struct sockaddr FAR \*name,

          int namelen,PVOID lpSendBuffer,

          DWORD dwSendDataLength,LPDWORD lpdwBytesSent,

          LPOVERLAPPED lpOverlapped

          )

{

     GRS\_ASSERT(NULL != m\_pfnConnectEx);

     return m\_pfnConnectEx(

             s,name,namelen,lpSendBuffer,

             dwSendDataLength,lpdwBytesSent,

             lpOverlapped

             );

}

GRS\_FORCEINLINE BOOL DisconnectEx(

          SOCKET s,LPOVERLAPPED lpOverlapped,

          DWORD  dwFlags,DWORD  dwReserved

          )

{

     GRS\_ASSERT(NULL != m\_pfnDisconnectEx);

     return m\_pfnDisconnectEx(s,

             lpOverlapped,dwFlags,dwReserved);

}

GRS\_FORCEINLINE VOID GetAcceptExSockaddrs (

          PVOID lpOutputBuffer,

          DWORD dwReceiveDataLength,

          DWORD dwLocalAddressLength,

          DWORD dwRemoteAddressLength,

          sockaddr \*\*LocalSockaddr,

          LPINT LocalSockaddrLength,

          sockaddr \*\*RemoteSockaddr,

          LPINT RemoteSockaddrLength

          )

{

     GRS\_ASSERT(NULL != m\_pfnGetAcceptExSockaddrs);

     return m\_pfnGetAcceptExSockaddrs(

          lpOutputBuffer,dwReceiveDataLength,

          dwLocalAddressLength,dwRemoteAddressLength,

          LocalSockaddr,LocalSockaddrLength,

          RemoteSockaddr,RemoteSockaddrLength

          );

}

GRS\_FORCEINLINE BOOL TransmitFile(

     SOCKET hSocket,HANDLE hFile,

     DWORD nNumberOfBytesToWrite,

     DWORD nNumberOfBytesPerSend,

     LPOVERLAPPED lpOverlapped,

     LPTRANSMIT\_FILE\_BUFFERS lpTransmitBuffers,

     DWORD dwReserved

     )

{

     GRS\_ASSERT(NULL != m\_pfnTransmitfile);

     return m\_pfnTransmitfile(

             hSocket,hFile,nNumberOfBytesToWrite,

             nNumberOfBytesPerSend,lpOverlapped,

             lpTransmitBuffers,dwReserved

             );

}

GRS\_FORCEINLINE BOOL TransmitPackets(

     SOCKET hSocket,

     LPTRANSMIT\_PACKETS\_ELEMENT lpPacketArray,

     DWORD nElementCount,DWORD nSendSize,

     LPOVERLAPPED lpOverlapped,DWORD dwFlags

     )

{

     GRS\_ASSERT(NULL != m\_pfnTransmitPackets);

     return m\_pfnTransmitPackets(

             hSocket,lpPacketArray,nElementCount,

nSendSize,lpOverlapped,dwFlags

             );

}

GRS\_FORCEINLINE INT WSARecvMsg(

          SOCKET s,LPWSAMSG lpMsg,

          LPDWORD lpdwNumberOfBytesRecvd,

          LPWSAOVERLAPPED lpOverlapped,

          LPWSAOVERLAPPED\_COMPLETION\_ROUTINE lpCompletionRoutine

          )

{

     GRS\_ASSERT(NULL != m\_pfnWSARecvMsg);

     return m\_pfnWSARecvMsg(

             s,lpMsg,lpdwNumberOfBytesRecvd,

             lpOverlapped,lpCompletionRoutine

             );

}

/\*WSAID\_ACCEPTEX

  WSAID\_CONNECTEX

  WSAID\_DISCONNECTEX

  WSAID\_GETACCEPTEXSOCKADDRS

  WSAID\_TRANSMITFILE

  WSAID\_TRANSMITPACKETS

  WSAID\_WSARECVMSG

  WSAID\_WSASENDMSG \*/

};

这个类的使用非常简单，只需要声明一个类的对象，然后调用其成员AcceptEx、DisconnectEx函数等即可，参数与这些函数的MSDN声明方式完全相同，除了本文中介绍的这些函数外，这个类还包含了很多其他的Winsock2函数，那么都应该按照这个类中演示的这样来动态载入后再行调用，如果无法载入通常说明你的环境中没有Winsock2函数库，或者是你初始化的不是2.0版的Winsock环境。

这个类是本人完整类库的一部分，如要使用需要自行修改一些地方，如果不知如何修改或遇到什么问题，可以直接跟帖说明，我会不定期回答大家的问题，这个类可以免费使用、分发、修改，可以用于任何商业目的，但是对于使用后引起的任何问题，本人概不负责，有问题请跟帖。关于AcceptEx以及其他一些函数，包括本文中没有介绍到得函数，我会在后续的一些专题文章中进行详细深入的介绍，敬请期待。如果你有什么疑问，或者想要了解什么也请跟帖说明，我会在后面的文章中尽量说明。

IOCP编程之“双节棍”

我博客之前的一些文章中，我讨论了关于使用BindIoCompletionCallback函数编写IOCP服务器的话题，在之后的一段时间中我也用此函数展开了伟大的服务器编程实践活动，在实际的应用中，我发现这个函数的很多怪脾气，今天我觉得有必要为大家澄清一下关于此函数的种种诽谤和传闻。

其实这是一个非常非常好用的函数，直接利用了Windows系统所有的优秀特性于一身——多线程、线程池、IOCP等等，我们要做的就是实现一个原型如下的CallBack函数：

VOID CALLBACK FileIOCompletionRoutine(

  [in]                 DWORD dwErrorCode,

  [in]                 DWORD dwNumberOfBytesTransfered,

  [in]                 LPOVERLAPPED lpOverlapped

);

他的参数中最有用的就是那个dwErrorCode，这里要特别注意的就是此参数并不像MSDN中说的那么简单的只是一些SOCKET错误码，在用SOCKET句柄代替文件句柄调用BindIoCompletionCallback函数的时候，dwErrorCode参数通常会返回一些0xC开头的系统内部状态码，比如常见的0xC000023F，表示目标地址不可到达，通常是目标UDP地址不可达或者目标UDP端口没有开放，这些错误码都可以在Windows的头文件中找到，通常在Ntstatus.h文件中。

通常我们可以通过调用RtlNtStatusToDosError函数来将这些内部的NTStatus码，翻译为所谓的“DOS”错误码，其实这个函数的名字很诡异，它实际完成就是从系统Ring0层错误码转换为Ring3层错误码，跟DOS其实没有关系。当然这个函数的调用也要通过LoadLibrary调入NtDll.Dll之后，再调用 GetProcAddress函数得到它的地址，然后调用之。

同时，还要特别注意的就是传说中BindIoCompletionCallback失败时，调用GetLastError函数得到错误码时，往往也需要用RtlNtStatusToDosError函数翻译一下，当然我没有遇到过BindIoCompletionCallback函数调用失败的情况，但是如果遇到了也不要慌张，这里已经告诉你了如何看懂这个诡异的错误码。

错误码搞清楚了，那么就可以从容不迫的处理关于回调FileIOCompletionRoutine发生的各种错误，从而让我们的服务器更健康。

当然今天我们讨论的不只是这个函数的错误码这么简单，我们要说的是“双节棍”，大家都知道“双节棍”，是有两根棍子用铁链链接起来才能发挥威力，这里说的BindIoCompletionCallback只是一根棍子，另一根棍子就是QueueUserWorkItem这个最简单的线程池函数。

两根棍子有了，链接到一起的方法就是在FileIOCompletionRoutine回调函数中根据操作类型，来调用QueueUserWorkItem将对应操作的WorkItem放入线程池，通常操作无非就是接收和发送数据，对应的需要编写“接收完成的WorkItem”和“发送完成的WorkItem”。这其实就是用普通线程池接力IOCP本身的线程池。为什么要这么啰嗦的做呢？

这是因为，在实践过程中，我发现，通常我们在接收到数据后，会做一些“慢速”的操作，最常见的就是访问数据库，虽然我用上了OLEDB，但是对于网络速度来说，似乎这还是慢的，尤其是有大量用户朝服务器发送数据请求时，我发现服务器响应很不尽人意，甚至有时出现了频繁掉线的情况。如果直接使用BindIoCompletionCallback通过在FileIOCompletionRoutine中直接调用这些慢速操作时，服务器的响应很低效，此时实际只有IOCP的线程池在工作，而我隐约感觉到这个IOCP的线程池是个很保守的线程池，它假设回调函数FileIOCompletionRoutine很快就会完成，否则就不再去响应其它的完成请求，基于这样的理解，我就在接收完成或发送完成操作之后直接调用QueueUserWorkItem再启动线程池，专门来处理数据，结果性能一下子有了质的飞跃，而且掉线的情况也成了凤毛麟角。在我实际用一个双核的破PC做测试时，服务器的响应一下子由500-1000连接飙升到了6000以上，大家不要笑哈，我说的这个成绩可是普通的破PC装的可是“菜羊”CPU，硬盘只是一块破SATA2盘，内存只有可怜的1G，这个成绩还是比较满意的。最后这个服务程序在上了8G内存双至强带Raid阵列的专业服务器之后，性能简直让我自己都瞠目结舌。

这里要特别提示的就是，在使用这个双节棍的结构中，内存管理就是核心的问题，尤其是要注意申请和释放，而要极力避免的就是共享内存，即尽量不要在线程池中使用并发控制来访问共享内存，对于一些应用来说这似乎又是无法避免的，我的策略是使用数据库来控制共享的状态，因为数据库的并发控制可以用“完美”这个词来形容，当然代价就是性能，但是只要你用了RAID，数据库还是不会让你失望的，至少这样我们省去了自己去做该死的并发控制的麻烦，而且不用担心服务器宕机，因为所有关键的状态都在数据库中，重启服务器自动就可以还原到原状态。

最后我需要澄清的一个事实就是，很多人说BindIoCompletionCallback函数不会启动超过一个线程以上的更多线程，我觉得这是个严重的诽谤，因为在单核单CPU机器上确实是这样的，但是在多核多CPU平台上，只要你没有BT的并发控制，线程数量一般是和CPU数目成正比的，我的服务器在双核机器上一般都会跑到10多个线程，因为有两个线程池，抛开调试时VS环境附加的几个线程，线程池的净线程数还是比较高的。另外要注意的一个问题就是似乎我们这里讨论的两类线程池函数都没有利用Intel的超线程技术，在单CPU超线程的机器上，你不会看到多余一个线程的壮观场面，而在多核CPU上，多个线程是可以被看到的。这也是我们选用IOCP线程池函数的初衷，因为它自动就可以适应多CPU的环境，带有先天的自动适应性。

当然目前Windows的线程池还很弱，除了2008平台以上，2003的服务器中线程池函数还是很弱，我们没法控制线程的亲缘性，没法控制线程的数量，没法处理线程的异常终止等等，我在BindIoCompletionCallback时就遇到过这类问题，因为回调函数的错误导致线程池罢工，服务器死锁等等情况，但是只要有了健壮的异常处理这类事故还是可以避免的，但是这都不是我们拒绝使用这根“双节棍”的理由，所谓“艺高人胆大，胆大艺更高”，只有不断的实践总结经验才是正确的道路。任何的道听途说，以讹传讹都不足为信。

在不久的将来，我将步入研究Windows2008平台“双节棍”的征程当中，因为2008的线程池比之2003的线程池那不是一般的强大。

IOCP加Windows线程池打造高伸缩性高性能的服务器应用

对于IOCP，搞Windows服务器编程的都不会陌生，它所表现出来的性能是其他各种WinSock模型难望其项背的。撰写本文的目的就是为让大家能够在深入理解IOCP的基础上，再来深入的挖掘Windows系统的性能。此处假设读者对IOCP模型已经深刻理解，并对Windows线程、线程池有一定的了解。如果对此还不熟悉，限于篇幅的原因，请您先学习理解这些内容后再来阅读本文。  
       在IOCP模型编程中，我们经常需要考虑的就是创建多少个线程来作为完成执行线程，很多时候这是个非常需要技巧和经验的决策性问题。大多数情况下，我们采取的策略是看服务器上有多少CPU然后假定每个CPU最多执行两个线程，然后我们创建的线程数量就是CPU数\*2。这看起来很合理，但是实际上在复杂的服务器应用环境中，这样做的效果并不尽如人意，很多时候我们希望得到一种更加动态灵活的方案。  
       有些有经验的程序员就自己编写线程池库，来实现这种动态灵活的管理方式，从而还可以实现一定的扩展性，比如系统动态的添加了一些CPU的资源，或者系统负担比较重的时候，或者CPU因为频繁切换线程场景而导致效率低下时，线程池的动态性就发挥出来了。

       索性的是，在Windows2000以上的平台上，已经为我们提供了线程池的接口，虽然这些接口有时候看起来还有些简陋，比如有名的QueueUserWorkItem函数，这些接口简陋到你连当前线程池中有多少活动线程等信息都无法知道，你只能通过其它的工具来动态观察和猜测。但这样的简单性也为我们带来了调用方便的实惠。当然到了Windows2008以上的平台时，线程池的函数总算是被大大加强了，你可以控制更多东西了，关于Windows2008线程池的内容请看我的另一篇博客拙作《Windows2008线程池前瞻》。  
       在结合IOCP和线程池这方面Windows系统也想到了程序员面临的这个困难，Windows系统干脆直接就在系统内部捆绑了IOCP和线程池，提供了一个带IOCP功能的线程池函数——BindIoCompletionCallback。  
此函数的原形如下：  
BOOL WINAPI BindIoCompletionCallback(  
  \_\_in          HANDLE FileHandle,  
  \_\_in          LPOVERLAPPED\_COMPLETION\_ROUTINE Function,  
  ULONG Flags  
);

需要的完成过程（实际也就是IOCP线程的过程）Function的原形需要你定义成如下的样子：  
VOID CALLBACK FileIOCompletionRoutine(  
  [in]                 DWORD dwErrorCode,  
  [in]                 DWORD dwNumberOfBytesTransfered,  
  [in]                 LPOVERLAPPED lpOverlapped  
);

        熟悉IOCP的各位可能已经兴奋得血管暴胀了吧？  
       从BindIoCompletionCallback函数的参数你应该已经能够猜到这个函数的用法了，第一个句柄就是你需要捆绑的文件句柄或者SOCKET套接字句柄，甚至是其他I/O设备的句柄。第二个函数的指针就是你的完成例程的指针，这个函数完全由你实现和控制，最后一个Flags参数当前所有的Windows系统中都必须赋予0值，这个参数实际上还没有被起用。  
       这么简单？真是难以置信，代表IOCP的句柄上哪去了？其实哪个什么IOCP的句柄，以及创建多少个线程什么的都不需要你考虑了，你唯一需要操心的就是如何编写完成例程以及如何将一个I/O句柄和完成例程捆绑起来。以前需要n多行代码才能完成的事情，一个BindIoCompletionCallback函数就彻底搞定了，甚至我们不需要再考虑线程的动态性问题了。这一切现在都有Windows系统综合考虑了，而你就被解放出来了。  
还愣着干嘛？快去写你的高可用性，高可扩展、高动态性的IOCP大型服务应用去了！