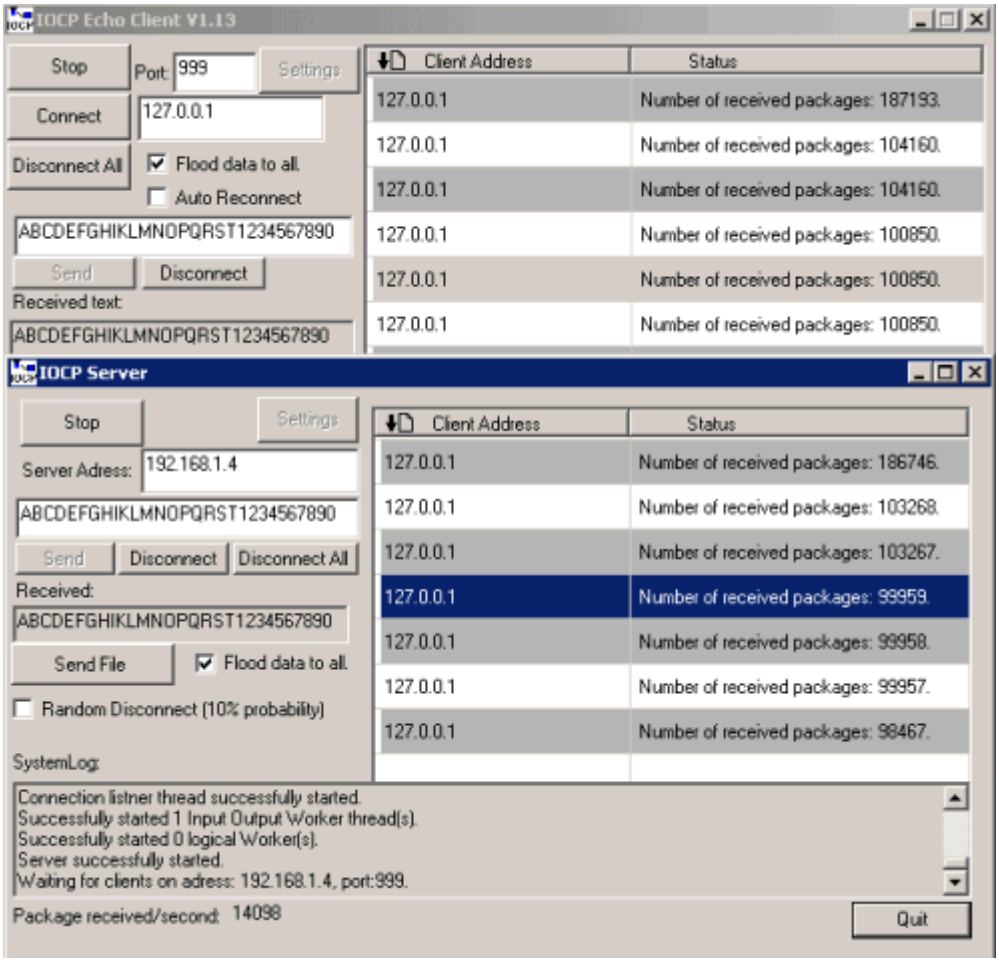
摘要

当你开发不同类型的软件，你迟早必须处理 C/S 的开发。对一个程序员来说，写一个通用的 C/S 编码是一项困难的工作。本文档提供了一份简单但是功能强大的 C/S 源码，可以扩展到任何类型的 C/S 应用程序中。这份源码使用了高级的 IOCP 技术，该技术可以高效的服务于多客户端。IOCP 提供了解决“每个客户 端占用一个线程”的瓶颈问题的办法，只使用几个处理线程，异步输入/输出来发送/接收。IOCP 技术被广泛应用在各种类型的高效服务端，例如 Apache 等。这份源码也提供了一系列的在处理通信和 C/S 软件中经常使用的功能，如文件接收/传送功能和逻辑线程池管理。本文重点在于出现在 IOCP 程序 API 中 实用的解决方案，以及关于源码的全面的文档。另外，一份简单的 echo 版的可处理多连接和文件传输的 C/S 程序也在这里提供。

本文的源码使用了高级的完成端口（IOCP）技术，该技术可以有效地服务于多客户端。本文提出了一些 IOCP 编程中出现的实际问题的解决方法，并提供了一个简单的 echo 版本的可以传输文件的客户端/服务器程序。程序截图如下：



环境要求

本文读者需要熟悉 C++、TCP/IP、Socket 编程、MFC，和多线程。源码使用 Winsock 2.0 和 IOCP 技术，要求：

- Windows NT/2000 或以上：要求 Windows NT3.5 或以后版本
- Windows 95/98/ME：不支持
- Visual C++.NET，或完整更新过的 Visual C++ 6.0

引言

本文提出了一个类，可以用在客户端和服务端。这个类使用 IOCP(Input Output Completion Ports)和异步（非阻塞）机制。通过这些简单的源码，你可以：

- 服务或连接多客户端和服务端
- 异步发送或接收文件
- 创建并管理一个逻辑工作者线程池，用以处理繁重的客户端/服务器请求或计算

找到一份全面但简单的解决客户端/服务器通信的源码是件困难的事情。在网络上找到的源码要么太复杂（超过 20 个类），要命没有提供足够的效率。本源码的设 计尽可能简单，并提供了充足的文档。在这篇文章中，我们简洁的呈现出了 Winsock API 2.0 支持的 IOCP 技术，说明了在编写过程中出现的棘手问题，并提出了每一个问题的解决方案。

异步完成端口介绍

如果一个服务器应用程序不能同时支持多个客户端，那是毫无意义的，为此，通常使用异步 I/O 请求和多线程。根据定义，一个异步 I/O 请求会立即返回，而留 下 I/O 请求处于等待状态。有时，I/O 异步请求的结果必须与主线程同步。这可以通过几种不同方式解决。同步可以通过下面的方式实现：

- 使用事件 – 当异步请求结束时会马上触发一个信号。这种方式的缺点是线程必须检查并等待事件被触发
- 使用 GetOverlappedResult 函数 – 这种方式与上一种方式有相同的缺点。
- 使用 Asynchronous Procedure Calls（或 APC） – 这种方式有几个缺点。首先，APC 总是在请求线程的上下文中被请求；第二，为了执行 APC，请求线程必须在可变等候状态下挂起。
- 使用 IOCP – 这种方式的缺点是必须解决很多实际的棘手的编程问题。编写 IOCP 可能有点麻烦。

为什么使用 IOCP?

通过使用 IOCP，我们可以解决“每个客户端占用一个线程”的问题。通常普遍认为如果软件不能运行在真正的多处理器机器上，执行能力会严重降低。线程是系统资源，而这些资源既不是无限的，也不是低价的。

IOCP 提供了一种方式来使用几个线程“公平的”处理多客户端的输入/输出。线程被挂起，不占用 CPU 周期直到有事可做。

什么是 IOCP?

我们已经看到 IOCP 只是一个线程同步对象，类似于信号灯，因此 IOCP 并不是一个复杂的概念。一个 IOCP 对象与几个支持待定异步 I/O 请求的 I/O 对象绑定。一个可以访问 IOCP 的线程可以被挂起，直到一个待定的异步 I/O 请求结束。

IOCP 是怎样工作的?

要使用 IOCP，你必须处理三件事情，绑定一个 socket 到完成端口，创建异步 I/O 请求，并与线程同步。为从异步 I/O 请求获得结果，如那个客户端发出的请求，你必须传递两个参数：CompletionKey 参数和 OVERLAPPED 结构。

关键参数

第一个参数：CompletionKey，是一个 DWORD 类型的变量。你可以传递任何你想传递的唯一值，这个值将总是同该对象绑定。正常情况下会传递一 个指向结构或类的指针，该结构或类包含了一些客户端的指定对象。在源码中，传递的是一个指向 ClientContext 的指针。

OVERLAPPED 参数

这个参数通常用来传递异步 I/O 请求使用的内存缓冲。很重要的一点是：该数据将会被锁定并不允许从物理内存中换出页面（page out）。

绑定一个 socket 到完成端口

一旦创建完成一个完成端口，可以通过调用 CreateIoCompletionPort 函数来绑定 socket 到完成端口。形式如下：

```
BOOL IOCPS::AssociateSocketWithCompletionPort(SOCKET socket,  HANDLE hCompletionPort, DWORD dwCompletionKey)
{
     HANDLE h = CreateIoCompletionPort((HANDLE) socket,  hCompletionPort, dwCompletionKey, m_nIOWorkers);
     return h == hCompletionPort;
}
```

响应异步 I/O 请求

响应具体的异步请求，调用函数 WSASend 和 WSARecv。他们也需要一个参数：WSABUF，这个参数包含了一个指向缓冲的指针。一个重要的规则 是：通常当服务器/客户端响应一个 I/O 操作，不是直接响应，而是提交给完成端口，由 I/O 工作者线程来执行。这么做的原因是：我们希望公平的分割 CPU 周期。通过发送状态给完成端口来发出 I/O 请求，如下：

```
BOOL bSuccess = PostQueuedCompletionStatus(m_hCompletionPort,
    pOverlapBuff->GetUsed(),
    (DWORD) pContext,
    &pOverlapBuff->m_ol);
```

与线程同步

与 I/O 工作者线程同步是通过调用 GetQueuedCompletionStatus 函数来实现的(如下)。这个函数也提供了 CompletionKey 参数和 OVERLAPPED 参数，如下：

```
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
);
```

四个棘手的 IOCP 编码问题和解决方法

使用 IOCP 时会出现一些问题，其中有一些不是很直观的。在使用 IOCP 的多线程编程中，一个线程函数的控制流程不是笔直的，因为在线程和通讯直接没有关系。在这一章节中，我们将描述四个不同的问题，可能在使用 IOCP 开发客户端/服务器应用程序时会出现，分别是：

- The WSAENOBUFS error problem.（WSAENOBUFS 错误问题）
- The package reordering problem.（包重构问题）
- The access violation problem.（访问非法问题）

WSAENOBUFS 问题

这个问题通常很难靠直觉发现，因为当你第一次看见的时候你或许认为是一个内存泄露错误。假定已经开发完成了你的完成端口服务器并且运行的一切良好， 但是当你对其进行压力测试的时候突然发现服务器被中止而不处理任何请求了，如果你运气好的话你会很快发现是因为 WSAENOBUFS 错误而影响了这一切。

每当我们重叠提交一个 send 或 receive 操作的时候，其中指定的发送或接收缓冲区就被锁定了。当内存缓冲区被锁定后，将不能从物理内存进行分页。操作系统有一个锁定最大数的限制，一旦超过这个锁定的限制，那么就会产生 WSAENOBUFS 错误了。

如果一个服务器提交了非常多的重叠的 receive 在每一个连接上，那么限制会随着连接数的增长而变化。如果一个服务器能够预先估计可能会产生的最大并发 连接数，服务器可以投递一个使用零缓冲区的 receive 在每一个连接上。因为当你提交操作没有缓冲区时，那么也不会存在内存被锁定了。使用这种办法后， 当你的 receive 操作事件完成返回时，该 socket 底层缓冲区的数据会原封不动的还在其中而没有被读取到 receive 操作的缓冲区来。此时，服务 器可以简单的调用非阻塞式的 recv 将存在 socket 缓冲区中的数据全部读出来，一直到 recv 返回 WSAEWOULDBLOCK 为止。 这种设计非常适合那些可以牺牲数据吞吐量而换取巨大 并发连接数的服务器。当然，你也需要意识到如何让客户端的行为尽量避免对服务器造成影响。在上一个例子中，当一个零缓冲区的 receive 操作被返回后使 用一个非阻塞的 recv 去读取 socket 缓冲区中的数据，如果服务器此时可预计到将会有爆发的数据流，那么可以考虑此时投递一个或者多个 receive 来取代非阻塞的 recv 来进行数据接收。（这比你使用 1 个缺省的 8K 缓冲区来接收要好的多。）

源码中提供了一个简单实用的解决 WSAENOBUF 错误的办法。我们执行了一个零字节缓冲的异步 WSARead(...)(参见 OnZeroByteRead(..))。当这个请求完成，我们知道在 TCP/IP 栈中有数据，然后我们通过执行几个有 MAXIMUMPACKAGESIZE 缓冲的异步 WSARead(...)去读，解决了 WSAENOBUFS 问题。但是这种解决方法降低了服务器的吞吐量。

解决方法一

投递使用空缓冲区的 receive 操作，当操作返回后，使用非阻塞的 recv 来进行真实数据的读取。因此在完成端口的每一个连接中需要使用一个循环的操作来不断的来提交空缓冲区的 receive 操作。

解决方法二

在投递几个普通含有缓冲区的 receive 操作后，进接着开始循环投递一个空缓冲区的 receive 操作。这样保证它们按照投递顺序依次返回，这样我们就总能对被锁定的内存进行解锁。

包重构问题

尽管使用 IO 完成端口的待发操作将总是按照他们发送的顺序来完成，线程调度安排可能使绑定到完成端口的实际工作不按指定的顺序来处理。例如，如果你有两个 I/O 工作者线程，你可能接收到“字节块 2，字节块 1，字节块 3”。这就意味着：当你通过向 I/O 完成端口提交请求数据发送数据时，数据实际上用重新排序 过的顺序发送了。

这可以通过只使用一个工作者线程来解决，并只提交一个 I/O 请求，等待它完成。但是如果这么做，我们就失去了 IOCP 的长处。

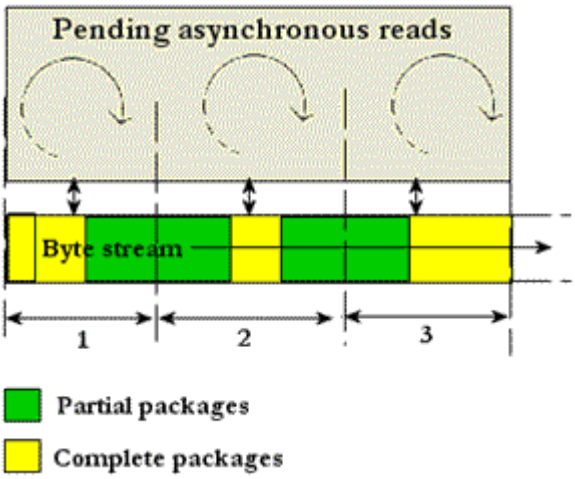
解决这个问题的一个简单实用办法是给我们的缓冲类添加一个顺序数字，如果缓冲顺序数字是正确的，则处理缓冲中的数据。这意味着：有不正确的数字的缓冲将被存下来以后再用，并且因为执行原因，我们保存缓存到一个 HASH MAP 对象中（如 m_SendBufferMap 和 m_ReadBufferMap）。

获取这种解决方法的更多信息，请查阅源码，仔细查看 IOCPS 类中如下的函数：

GetNextSendBuffer (..) and GetNextReadBuffer(..), to get the ordered send or receive buffer.
IncreaseReadSequenceNumber(..) and IncreaseSendSequenceNumber(..), to increase the sequence numbers.

异步等待读 和 字节块包处理问题

最通用的服务端协议是一个基于协议的包，首先 X 个字节代表包头，包头包含了详细的完整的包的长度。服务端可以读包头，计算出需要多少数据，继续读取直到读 完一个完整的包。当服务端同时只处理一个异步请求时工作的很好。但是，如果我们想发挥 IOCP 服务端的全部潜能，我们应该启用几个等待的异步读事件，等待 数据到达。这意味着几个异步读操作是不按顺序完成的，通过等待的读事件返回的字节块流将不会按顺序处理。而且，一个字节块流可以包含一个或几个包，也可能 包含部分包，如下图所示：



这个图形显示了部分包（绿色）和完整包（黄色）是怎样在不同字节块流中异步到达的。这意味着我们必须处理字节流来成功的读取一个完整的包。而且，我们必须处理部分包（图表中绿色的部分）。这就使得字节流的处理更加困难。这个问题的完整解决方法在 IOCPS 类的 ProcessPackage(…)函数中。

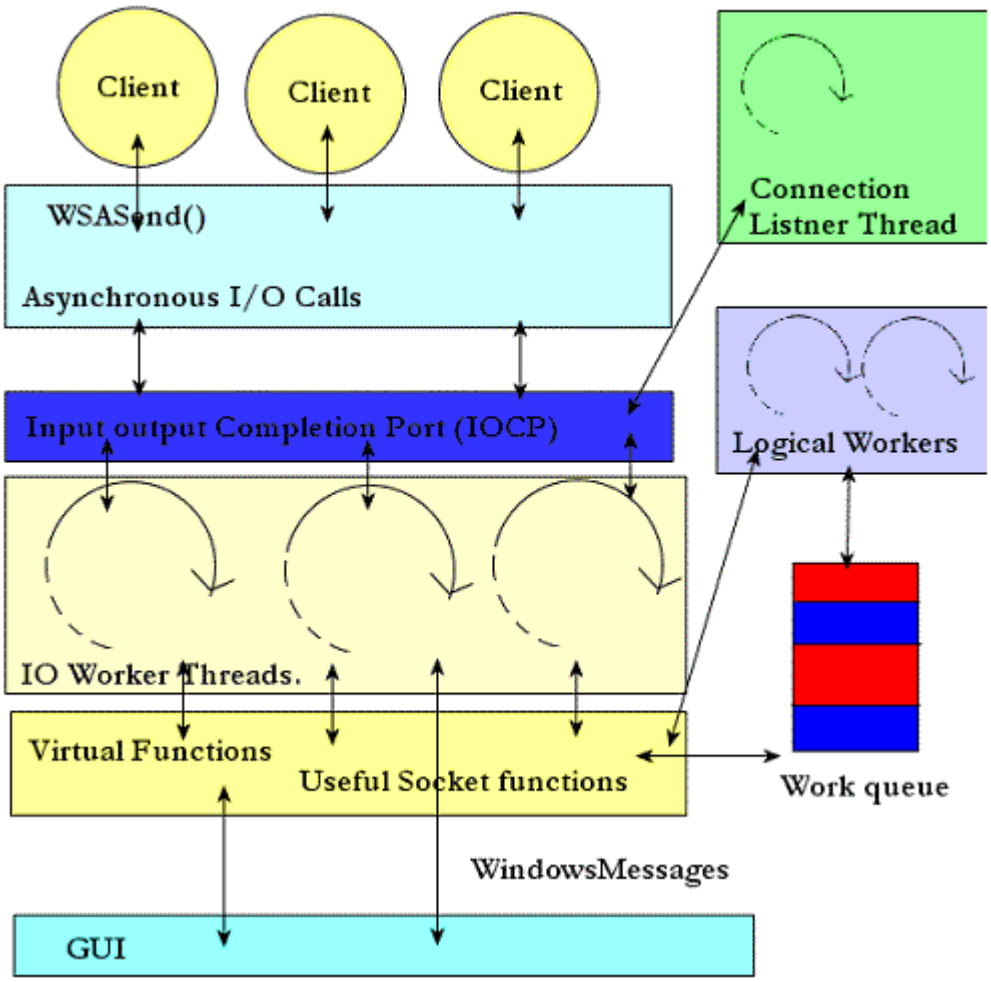
访问非法问题

这是一个较小的问题，代码设计导致的问题更胜于 IOCP 的特定问题。假设一个客户端连接已经关闭并且一个 I/O 请求返回一个错误标志，然后我们知道客户端 已经关闭。在参数 CompletionKey 中，我们传递了一个指向结构 ClientContext 的指针，该结构中包含了客户端的特定数据。如果我们释 放这个 ClientContext 结构占用的内存，并且同一个客户端处理的一些其它 I/O 请求返回了错误代码，我们通过转换参数 CompletionKey 为一个指向 ClientContext 结构的指针并试图访问或删除它，会发生什么呢？一个非法访问出现了！

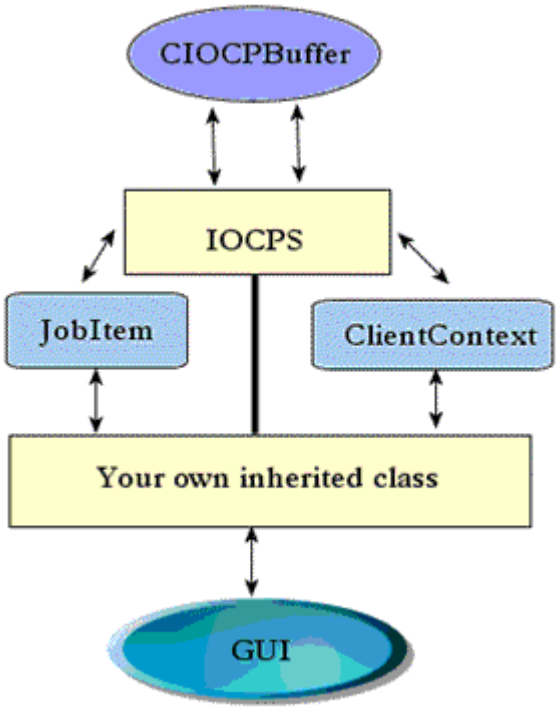
这个问题的解决方法是添加一个数字到结构中，包含等待的 I/O 请求的数量（m_nNumberOfPendingIO），然后当我们知道没有等待的 I/O 请求时删除这个结构。这个功能通过函数 EnterIoLoop(…) 和 ReleaseClientContext(…)来实现。

源码略读

源码的目标是提供一系列简单的类来处理所有 IOCP 编码中的问题。源码也提供了一系列通信和 C/S 软件中经常使用的函数，如文件接收/传送函数，逻辑线程池处理，等等。下图功能性的图解说明了 IOCP 类源码。



我们有几个 IO 工作者线程通过完成端口来处理异步 IO 请求，这些工作者线程调用一些虚函数，这些虚函数可以把需要大量计算的请求放到一个工作队列中。逻辑 工作者通过类中提供的这些函数从队列中取出任务、处理并发回结果。GUI 经常与主类通信，通过 Windows 消息（因为 MFC 不是线程安全的）、通过调用 函数或通过使用共享的变量。下图图显示了类结构纵览。



上图中的类说明如下：

- CIOCPBuffer：管理异步请求的缓存的类。
- IOCPS：处理所有通信的主类。
- JobItem：保存逻辑工作者线程要处理的任务的结构。
- ClientContex：保存客户端特定信息的结构（如状态、数据，等等）。

缓冲设计：CIOCPBuffer 类

使用异步 I/O 调用时，我们必须提供私有的缓冲区供 I/O 操作使用。当我们将帐号信息放入分配的缓冲供使用时有许多情况需要考虑。

分配和释放内存代价高，因此我们应重复使用以及分配的缓冲(内存)，因此我们将缓冲保存在列表结构中，如下所示：

```
// 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.
```

有时，当异步 I/O 调用完成后，缓冲里可能不是完整的包，因此我们需要分割缓冲去取得完整的信息。在 CIOCPS 类中提供了 SplitBuffer 函数。

同样，有时候我们需要在缓冲间拷贝信息，IOCPS 类提供了 AddAndFlush 函数。

众所周知，我们也需要添加序号和状态（IOType 变量，IOZeroReadCompleted，等等）到我们的缓冲中。我们也需要有将数据转换到字节流或将字节流转换到数据的方法，CIOCPBuffer 也提供了这些函数。

以上所有问题都在 CIOCPBuffer 中解决。

如何使用源代码

从 IOCP 继承你自己的类（如图 3），实现 IOCPS 类中的虚函数（例如，threadpool），在任何类型的服务端或客户端中实现使用少量的线程有效地管理大量的连接。

启动和关闭服务端/客户端

调用下面的函数启动服务端

```
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);
```

- nPort，服务端侦听的端口.（-1 客户端模式.）
- iMaxNumConnections ，允许最大的连接数.（使用较大的数.）
- iMaxIOWorkers ，I/O 工作线程数
- nOfWorkers ，逻辑工作者数量 Number of logical workers.（可以在运行时改变.）
- iMaxNumberOfFreeBuffer ，重复使用的缓冲最大数.（-1 不使用，0= 不限）
- iMaxNumberOfFreeContext ，重复使用的客户端信息对象数（-1 for 不使用，0= 不限）
- bOrderedRead ，顺序读取.（我们已经在 3.6.2. 处讨论过）
- bOrderedSend ，顺序写入.（我们已经在 3.6.2. 处讨论过）
- iNumberOfPendlingReads，等待读取数据时未决的异步读取循环数

连接到远程服务器（客户端模式 nPort=-1），调用函数：

```
CodeConnect(const CString &strIPAddr, int nPort)
```

- strIPAddr ，远程服务器的 IP 地址
- nPort ，端口

调用 ShutDown()关闭连接，例如：

```
if(!m_iocp.Start(-1,1210,2,1,0,0))
AfxMessageBox("Error could not start the Client");
....
m_iocp.ShutDown();
```

源代码描述

更多关于源代码的信息请参考代码里的注释。

事件/虚函数

- NotifyNewConnection，新的连接已接受
- NotifyNewClientContext，空的 ClientContext 结构被分配
- NotifyDisconnectedClient，客户端连接断开
- ProcessJob，逻辑工作者需要处理一个工作
- NotifyReceivedPackage，新的包到达
- NotifyFileCompleted，文件传送完成。

重要变量

所有变量共享使用时必须加锁避免存取违例，所有需要加锁的变量，名称为 XXX 则锁变量名称为 XXXLock。

- m_ContextMapLock，数据保护锁
- m_ContextMap;，保存所有客户端数据（socket,客户端数据,等等）
- m_NumberOfActiveConnections，保存已连接的连接数

重要函数

- GetNumberOfConnections()，返回连接数
- CString GetHostAdress(ClientContext* p)，提供客户端上下文，返回主机地址
- BOOL ASendToAll(CIOCPBuffer *pBuff)，发送缓冲上下文到所有连接的客户端
- DisconnectClient(CString sID)，根据客户端唯一编号，断开指定的客户端
- CString GetHostIP()，返回本地 IP
- JobItem* GetJob()，将 JobItem 从队列中移出，如果没有 job，返回 NULL
- BOOL AddJob(JobItem *pJob)，添加 Job 到队列
- BOOL SetWorkers(int nThreads)，设置可以任何时候调用的逻辑工作者数量
- DisconnectAll()，断开所有客户端
- ARead(…)，异步读取
- ASend(…)，异步发送，发送数据到客户端
- ClientContext* FindClient(CString strClient)，根据字符串 ID 寻找客户（非线程安全）
- DisconnectClient(ClientContext* pContext, BOOL bGraceful=FALSE)，端口客户
- DisconnectAll()，端口所有客户

- StartSendFile(ClientContext *pContext)，根据 ClientContext 结构发送文件（使用经优化的 transmitfile(..) 函数）
- PrepareReceiveFile(..)，接收文件准备。调用该函数时，所有进入的字节流已被写入到文件。
- PrepareSendFile(..)，打开文件并发送包含文件信息的数据包。函数禁用 ASend(..)函数，直到文件传送关闭或中断。
- DisableSendFile(..)，禁止发送文件模式
- DisableRecevideFile(..)，禁止文件接收模式

文件传输

文件传输使用 Winsock 2.0 中的 TransmitFile 函数。TransmitFile 函数通过连接的 socket 句柄传送文件数据。函数使用操作系统的高速缓冲管理器 (cache manager)接收文件数据,通过 sockets 提供高性能的文件数据传输。

在 TransmitFile 函数返回前,所有其他发送或写入到该 socket 的操作都将无法执行,因为这将使文件数据混乱。因此,在 PrepareSendFile()函数调用之后，所有 ASend 都被禁止。因为操作系统连续读取文件数据，你可以使用 FILE_FLAG_SEQUENTIAL_SCAN 参数来优化高速缓冲性能。

发送文件时我们使用了内核异步操作(TF_USE_KERNEL_APC)。TF_USE_KERNEL_APC 的使用可以更好地提升性能。有可能，无论如何,TransmitFile 在线程中的大量使用，这种情形可能会阻止 APCs 的调用.

文件传输按如下顺序执行:服务器调用 PrepareSendFile(..)函数初始化文件传输。客户端接收文件信息时,调用 PrepareReceiveFile(..)作接收前的准备，并发送一个包到服务器告知开始文件传送。当包到达服务器端，服务器端调用 StartSendFile(..)采用高性能的 TransmitFile 函数发送指定文件。

源代码示例

提供的源代码演示代码是一个 echo 客户端/服务器端程序，并提供了对文件传输的支持（图 4）。在代码中，MyIOCP 类从 IOCP 继承，处理客户端/服务器端的通讯，所涉及的虚函数可以参见 4.1.1 处。
客户端或服务器端最重要的部分是虚函数 NotifyReceivedPackage，定义如下：

```
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;
                    };
}
```

函数接收进入的信息并执行远程连接发送的请求。这种情况，只是简单的 echo 或文件传输的情形。服务器端和客户端源代码分成两个工程，IOCP 和 IOCPClient。

编译问题

使用 VC++6.0 或 VC.NET 编译，你可能在编译 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(*.h)或 VC++ 6.0 的版本或改变类型转换错误来解决，在修正了错误后，服务器端/客户端源代码可以在不需 MFC 的情况下使用。

特别的考虑和经验总结

当你在其他类型的程序中使用本代码，有一些可以避免的编程陷阱和多线程问题。非确定错误指的是那些随机出现的错误，很难通过执行相同顺序的任务来重现这些错误。这是最坏的错误类型，通常，错误出现在内部源代码的设计中。当服务器端有多个 IO 工作线程在运行，为客户端提供连接，如果程序员没有考虑多线程环境，可能会发生存取违例等不确定错误。

经验 1

在未对上下文加锁时，不要读写客户上下文（例如：ClientContext）。通知函数（例如：Notify*(ClientContext *pContext)）已经是线程安全，处理成员变量 ClientContext 可以不需要解锁、解锁。

```
//不要这样使用
// …
If(pContext->m_bSomeData)
pContext->m_iSomeData=0;
// …

// 应该这样使用
//….
pContext->m_ContextLock.Lock();
If(pContext->m_bSomeData)
pContext->m_iSomeData=0;
pContext->m_ContextLock.Unlock();
//…
```

大家都知道的，当你锁定上下文，其他线程或 GUI 都将等待它。

经验 2

避免使用复杂的和其他类型的"上下文锁"，应为容易造成死锁（例如：A 在等待 B，B 在等待 C，C 在等待 A，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();
```

以上代码可能导致死锁

经验 3

不要在通知函数(例如：Notify*(ClientContext *pContext))以外处理客户上下文，如果你需要这样做，你要放入

```
m_ContextMapLock.Lock();
…
m_ContextMapLock.Unlock();
```

参考如下代码：

```
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.
```

将来的工作

将来，代码将提供以下功能：

- 添加支持 AcceptEx(..)接受新连接，处理短连接和 DOS 攻击。
- 源代码兼容 Win32,STL,WTL 等环境。

原文地址：http://www.hellocpp.net/Articles/Article/436.aspx