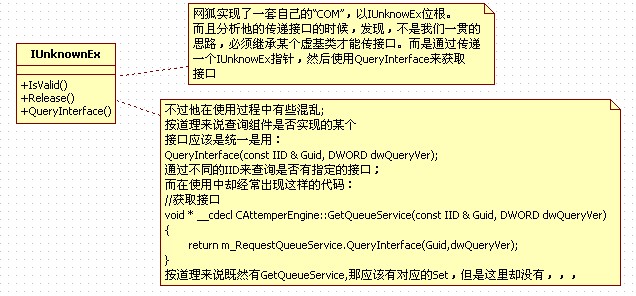
源码分析

# IUnknowEx [成也萧何败也萧何？](http://www.cppblog.com/Error/articles/148413.html)



看下某个查询接口的代码：

//////////////////////////////////////////////////////////////////////////

//接口查询宏

#define QUERYINTERFACE(Interface,Guid,dwQueryVer)                                        \

    if ((Guid==IID\_##Interface)&&(InterfaceVersionCheck(dwQueryVer,VER\_##Interface)))    \

        return static\_cast<Interface \*>(this);

#define QUERYINTERFACE\_IUNKNOWNEX(BaseInterface,Guid,dwQueryVer)                        \

    if ((Guid==IID\_IUnknownEx)&&(InterfaceVersionCheck(dwQueryVer,VER\_IUnknownEx)))        \

        return static\_cast<IUnknownEx \*>(static\_cast<BaseInterface \*>(this));

//获取接口宏

#define GET\_MYSELF\_INTERFACE(Interface)                                                    \

    ((Interface \*)QueryInterface(IID\_##Interface,VER\_##Interface))

#define GET\_OBJECT\_INTERFACE(Object,Interface)                                            \

    ((Interface \*)Object.QueryInterface(IID\_##Interface,VER\_##Interface))

#define GET\_OBJECTPTR\_INTERFACE(pObject,Interface)                                        \

    ((pObject==NULL)?NULL:((Interface \*)pObject->QueryInterface(IID\_##Interface,VER\_##Interface)))

//////////////////////////////////////////////////////////////////////////

//接口查询

void \* \_\_cdecl CEventService::QueryInterface(const IID & Guid, DWORD dwQueryVer)

{

    QUERYINTERFACE(IEventService,Guid,dwQueryVer);

    QUERYINTERFACE\_IUNKNOWNEX(IEventService,Guid,dwQueryVer);

    return NULL;

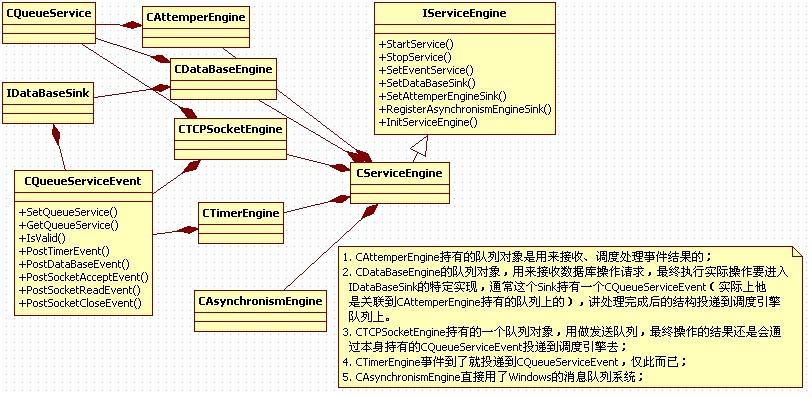
}

由于这个函数的存在，查询接口传递的类型检查由编译期转到了运行期。

任意接口的传递都只需要传递一个IUnknowEx指针，然后接受方使用IUnknowEx去判断是否是自己需要的接口。

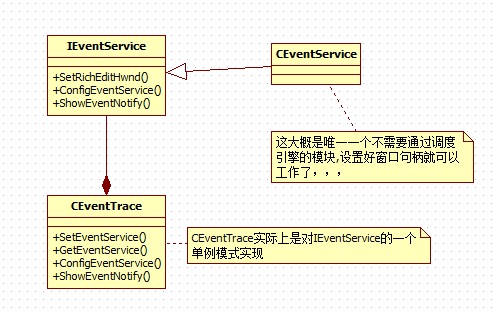
这个机制是好是坏，难说…

# [服务引擎概览](http://www.cppblog.com/Error/articles/148411.html)



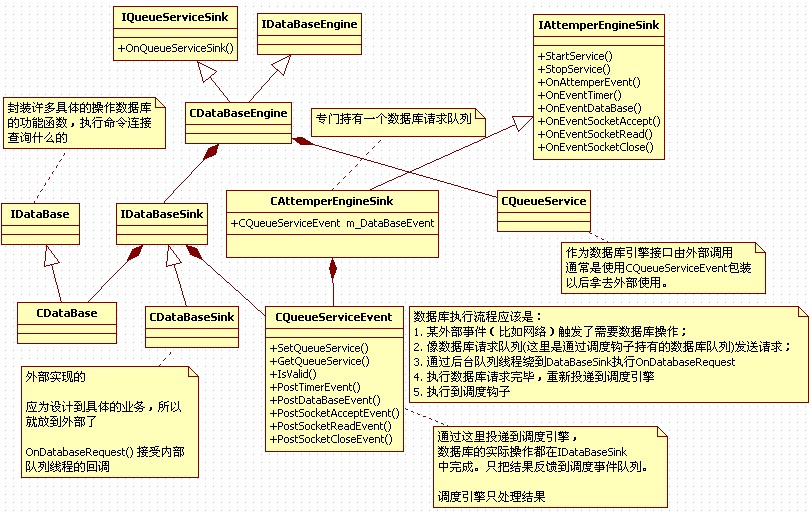
服务引擎这是简单的聚合了其他的子模块，然后在Start中做些初始化的工作。

# [CEventService](http://www.cppblog.com/Error/articles/148305.html)



线程安全的做一些格式化，然后一股脑输出到窗口句柄上

# [DataBaseEngine](http://www.cppblog.com/Error/articles/148282.html)



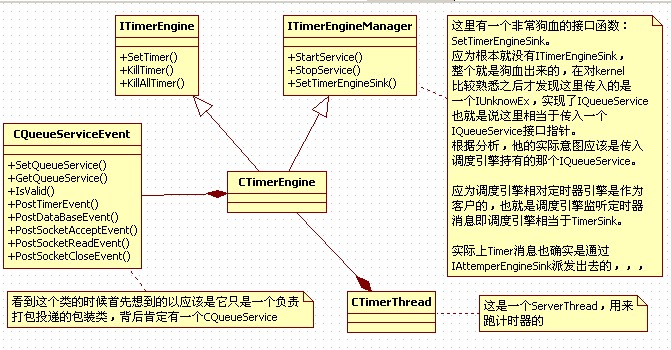
使用数据库引擎必须持有数据库引擎内部队列，实现数据库回调接口（用来处理数据库操作请求），然后结果投递到调度引擎队列。

理解网络狐棋牌，最重要是理解两个概念：

1，队列服务

2，接口的传递

# [TimerEngine](http://www.cppblog.com/Error/articles/148204.html)



CTimerEngine成员变量：

    //状态变量

    DWORD                                m\_dwTimerSpace;                    //时间间隔，这个值后来应该是给了定时器线程

protected:

    bool                                m\_bService;                        //运行标志

    DWORD                                m\_dwTimePass;                    //经过时间

    DWORD                                m\_dwTimeLeave;                    //倒计时间

    CTimerItemPtr                        m\_TimerItemFree;                //空闲数组

    CTimerItemPtr                        m\_TimerItemActive;                //活动数组

这些是分析定时器线程函数需要了解的成员变量，其次是线程同步：

从代码中可以看出所有对：m\_TimerItemActive;  m\_TimerItemFree;的操作都是在一个线程锁的保护之下的。

定时器线程：

通过CTimerThread绕一圈以后最终被循环执行的实际上是这个函数（m\_dwTimerSpace控制最小时间）：

//定时器通知

void CTimerEngine::OnTimerThreadSink()

{

    //锁定资源

    CThreadLockHandle LockHandle(&m\_ThreadLock);

    //倒计时间

    if (m\_dwTimeLeave==NO\_TIME\_LEFT)

    {

        ASSERT(m\_TimerItemActive.GetCount()==0);

        return;

    }

    //减少时间

    ASSERT(m\_dwTimeLeave>=m\_dwTimerSpace);

    m\_dwTimeLeave-=m\_dwTimerSpace;

    m\_dwTimePass+=m\_dwTimerSpace;

    //查询定时器

    if (m\_dwTimeLeave==0)

    {

        bool bKillTimer=false;

        tagTimerItem \* pTimerItem=NULL;

        DWORD dwTimeLeave=NO\_TIME\_LEFT;

        for (INT\_PTR i=0;i<m\_TimerItemActive.GetCount();)

        {

            //效验参数

            pTimerItem=m\_TimerItemActive[i];

            ASSERT(pTimerItem!=NULL);

            ASSERT(pTimerItem->dwTimeLeave>=m\_dwTimePass);

            //定时器处理

            bKillTimer=false;

            // 这一行做了--操作

            pTimerItem->dwTimeLeave -= m\_dwTimePass;

            if (pTimerItem->dwTimeLeave==0L)

            {

                //发送通知

                m\_AttemperEvent.PostTimerEvent(pTimerItem->wTimerID,pTimerItem->wBindParam);

                //设置次数

                if (pTimerItem->dwRepeatTimes!=TIMER\_REPEAT\_TIMER)

                {

                    ASSERT(pTimerItem->dwRepeatTimes>0);

                    if (pTimerItem->dwRepeatTimes==1L)

                    {

                        bKillTimer=true;

                        m\_TimerItemActive.RemoveAt(i);

                        m\_TimerItemFree.Add(pTimerItem);

                    }

                    else pTimerItem->dwRepeatTimes--;

                }

                //设置时间，从新开始倒计时

                if (bKillTimer==false) pTimerItem->dwTimeLeave=pTimerItem->dwElapse;

            }

            //增加索引

            if (bKillTimer==false)

            {

                i++;

                dwTimeLeave=\_\_min(dwTimeLeave,pTimerItem->dwTimeLeave);

                ASSERT(dwTimeLeave%m\_dwTimerSpace==0);

            }

        }

        //设置响应

        m\_dwTimePass=0L;

        m\_dwTimeLeave=dwTimeLeave;

    }

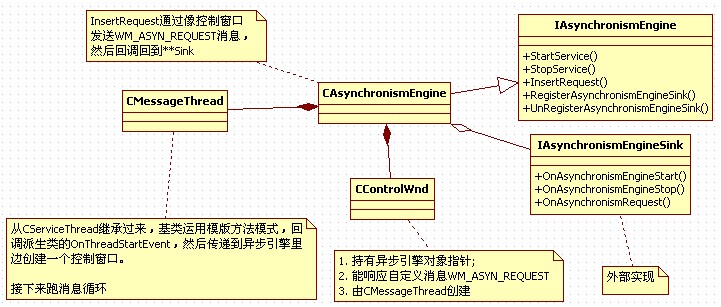
    return;

}

CTimerEngine：

启动一个定时器线程，循环遍历定时器，如果发现满足出发条件的定时器就投递一个定时器消息到CQueueServiceEvent对象。

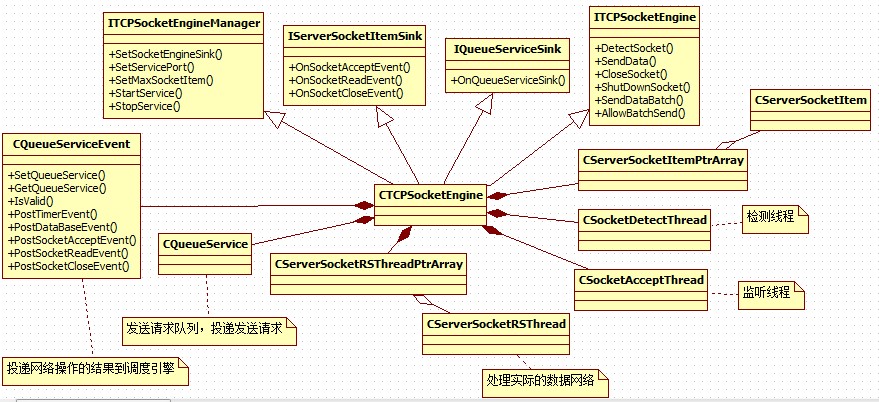
# [异步引擎](http://www.cppblog.com/Error/articles/148355.html)



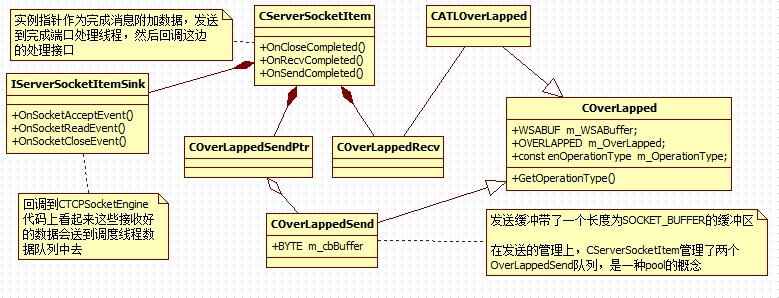
核心技术在于：在指定的线程中创建窗口，然后向窗口发送消息，在窗口消息处理中调用外部回调。

# [TCPSocketEnging分析](http://www.cppblog.com/Error/articles/148237.html)

网络引擎整体结构：



SocketItem细节：



先来看几个底层结构：

//重叠结构类

class COverLapped

{

    //变量定义

public:

    WSABUF                            m\_WSABuffer;                        //数据指针

    OVERLAPPED                        m\_OverLapped;                        //重叠结构

    const enOperationType            m\_OperationType;                    //操作类型

    //函数定义

public:

    //构造函数

    COverLapped(enOperationType OperationType);

    //析构函数

    virtual ~COverLapped();

    //信息函数

public:

    //获取类型

    enOperationType GetOperationType() { return m\_OperationType; }

};

//接收重叠结构

class COverLappedSend : public COverLapped

{

    //数据变量

public:

    BYTE                            m\_cbBuffer[SOCKET\_BUFFER];            //数据缓冲

    //函数定义

public:

    //构造函数

    COverLappedSend();

    //析构函数

    virtual ~COverLappedSend();

};

//重叠结构模板

template <enOperationType OperationType> class CATLOverLapped : public COverLapped

{

    //函数定义

public:

    //构造函数

    CATLOverLapped() : COverLapped(OperationType) {}

    //析构函数

    virtual ~CATLOverLapped() {}

};

先复习下基础，Windows下的网络模型有很多种，这里只拿出三种来说：

EventSelect：基于信号机制，以socket为单位绑定信号量，当socket上有指定的事件发生时激发信号，然后查询事件处理事件重设事件，继续在信号量上等待。其实也是在伯克利select模型上的换不换药的加强。

OverLapped：分两种工作模式完成回调，和完成事件。重叠IO监视每次操作，每次IO都绑定一个重叠对象，当操作完成以后激发信号或者调用回调。

IOCP：和overlapped类似，不过结果经过了Windows的预处理以队列的形式挂在完成端口上

根据上面的复习，可以得出一个结论，IOCP环境中每一次IO操作都需要一个重叠结构，那么一个CServerSocketItem至少需要如些这些东东：

他要接受数据，所以必须有一个接受数据的 OverLapped

它要发送数据，说以必须有一个发送数据的 OverLapped

netFox对OverLapped做了使用了类似池的的管理手段，他的Send都是不等待上一次完成就直接投递下一个请求了，，，这是很操蛋的做法，，，

然后继续复习下基础：

在EventSelect模型中获处理件类型流程是这样：

event受信，使用::WSAEnumNetworkEvents查询和这个event关联的socket发生的事件，根据查询到的事件类型去处理事件

在以每一次IO为查询对象重叠IO、IOCP模型中是这样：

使用GetOverlappedResult 或者 GetQueuedCompletionStatus然后根据重叠结构去查询投递的是什么类型的操作，然后找到关联的socket去操作，，，

这样必然要给OverLapped做个扩展，提供一种通过OverLapped查询操作类型和socket的能力。

通过分析代码，netFox关联socket是通过在创建完成端口的时候绑定SocketItem对象指针完成的，操作类型是通过对OverLapped结构加强完成的。

通过GetQueuedCompletionStatus获取到完成OverLapped以后使用一个宏：

（这是COverLapped类型）  pSocketLapped=CONTAINING\_RECORD(pOverLapped,COverLapped,m\_OverLapped);

来获取包装后的OverLapped，然后获取操作类型，然后执行具体操作。

其实宏的展开如下：

（COverLapped\*）((BYTE\*)pOverLapped - (COverLapped\*)(0)->m\_OverLapped);

pOverLapped 是获取到的某个COverLapped中的成员变量，(COverLapped\*)(0)->m\_OverLapped是到在 COverLapped中的偏移，((BYTE\*)pOverLapped - (COverLapped\*)(0)->m\_OverLapped) 就是根据pOverLapped推算出来的包含地址为pOverLapped作为成员变量m\_OverLapped的COverLapped对象的地址。

然后就分别调用：

//发送完成函数

bool CServerSocketItem::OnSendCompleted(COverLappedSend \* pOverLappedSend, DWORD dwThancferred);

//接收完成函数

bool CServerSocketItem::OnRecvCompleted(COverLappedRecv \* pOverLappedRecv, DWORD dwThancferred);

为毛要区分Send OverLapped 和 Recv OverLapped呢，，，

应为投递一次Send不一定是瞬间完成的，在处理的过程中存储数据的内存应该是锁定的，也就是不允许修改的，，，所以OverLapped应该自己管理内存。

而recv应该也是需要有一片内存直接接受数据的，很奇怪netFox没有提供，，，

recv居然是在投递接受请求的时候给了一个空的buffer，然后在完成回调中自己再次调用recv方法接受数据。

接受有关的成员变量如下：

  //状态变量

protected:

    bool                            m\_bNotify;                            //通知标志

    bool                            m\_bRecvIng;                            //接收标志

    bool                            m\_bCloseIng;                        //关闭标志

    bool                            m\_bAllowBatch;                        //接受群发

    WORD                            m\_wRecvSize;                        //接收长度

    BYTE                            m\_cbRecvBuf[SOCKET\_BUFFER\*5];        //接收缓冲

Int iRetCode=recv(m\_hSocket,(char \*)m\_cbRecvBuf+m\_wRecvSize,sizeof(m\_cbRecvBuf)-m\_wRecvSize,0);

难道这么蠢的做法只是为了躲开分包算法？

具体的看看接受代码：

//接收完成函数

bool CServerSocketItem::OnRecvCompleted(COverLappedRecv \* pOverLappedRecv, DWORD dwThancferred)

{

    //效验数据

    ASSERT(m\_bRecvIng==true);

    //设置变量

    m\_bRecvIng=false;

    m\_dwRecvTickCount=GetTickCount();

    //判断关闭

    if (m\_hSocket==INVALID\_SOCKET)

    {

        CloseSocket(m\_wRountID);

        return true;

    }

    //接收数据

    int iRetCode=recv(m\_hSocket,(char \*)m\_cbRecvBuf+m\_wRecvSize,sizeof(m\_cbRecvBuf)-m\_wRecvSize,0);

    if (iRetCode<=0)

    {

        CloseSocket(m\_wRountID);

        return true;

    }

    //接收完成

    m\_wRecvSize+=iRetCode;

    BYTE cbBuffer[SOCKET\_BUFFER];

    CMD\_Head \* pHead=(CMD\_Head \*)m\_cbRecvBuf;

    //处理数据

    try

    {

        while (m\_wRecvSize>=sizeof(CMD\_Head))

        {

            //效验数据

            WORD wPacketSize=pHead->CmdInfo.wDataSize;

            if (wPacketSize>SOCKET\_BUFFER) throw TEXT("数据包超长");

            if (wPacketSize<sizeof(CMD\_Head)) throw TEXT("数据包非法");

            if (pHead->CmdInfo.cbMessageVer!=SOCKET\_VER) throw TEXT("数据包版本错误");

            if (m\_wRecvSize<wPacketSize) break;

            //提取数据

            CopyMemory(cbBuffer,m\_cbRecvBuf,wPacketSize);

            WORD wRealySize=CrevasseBuffer(cbBuffer,wPacketSize);

            ASSERT(wRealySize>=sizeof(CMD\_Head));

            m\_dwRecvPacketCount++;

            //解释数据

            WORD wDataSize=wRealySize-sizeof(CMD\_Head);

            void \* pDataBuffer=cbBuffer+sizeof(CMD\_Head);

            CMD\_Command Command=((CMD\_Head \*)cbBuffer)->CommandInfo;

            //内核命令

            if (Command.wMainCmdID==MDM\_KN\_COMMAND)

            {

                switch (Command.wSubCmdID)

                {

                case SUB\_KN\_DETECT\_SOCKET:    //网络检测

                    {

                        break;

                    }

                default: throw TEXT("非法命令码");

                }

            }

            else

            {

                //消息处理

                m\_pIServerSocketItemSink->OnSocketReadEvent(Command,pDataBuffer,wDataSize,this);

            }

            //删除缓存数据

            m\_wRecvSize-=wPacketSize;

            MoveMemory(m\_cbRecvBuf,m\_cbRecvBuf+wPacketSize,m\_wRecvSize);

        }

    }

    catch (http://www.cppblog.com/Images/dot.gif)

    {

        CloseSocket(m\_wRountID);

        return false;

    }

    return RecvData();

}

这是还是有分包算法的，总的来说接受流程如下：

直接使用recv把数据接受到SocketItem的缓冲区中，当长度大于CMD\_HEAD之后，进入处理阶段，处理head数据各种判断，然后将数据扔出去，再调整缓冲区，，，

简单的说：

Send完全不考虑同步问题，不管一个劲的网队列投递Send请求，，，这边处理队列也是直接Send完事，完全不考虑上一次是否send成功，，，

Recv更是莫名其妙的使用完成端口绕一圈还回到recv直接接受了，，，

很狗血的做法，，，

更正下我自己狗血的不理解：

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。

总结：

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。

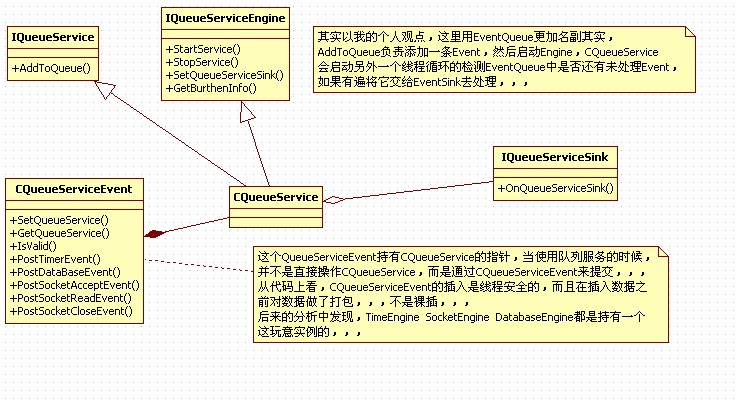
解决方法二：

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。

///////////

如 果一个服务器同时连接了许多客户端, 对每个客户端又调用了许多 WSARecv, 那么大量的内存将会被锁定到非分页内存池. 锁定这些内存时是按照页面边界来锁定的, 也就是说即使你 WSARecv 的缓存大小是 1 字节, 被锁定的内存也将会是 4k. 非分页内存池是由整个系统共用的, 如果用完的话最坏的情况就是系统崩溃. 一个解决办法是, 使用大小为 0 的缓冲区调用 WSARecv. 等到调用成功时再换用非阻塞的 recv 接收到来的数据, 直到它返回 WSAEWOULDBLOCK 表明数据已经全部读完. 在这个过程中没有任何内存需要被锁定, 但坏处是效率稍低.

# [CQueueServiceEvent初步分析](http://www.cppblog.com/Error/articles/147948.html)



代码分析：

首先CQueueServiceEvent这个名字很有诱惑，，，你感觉他是一个事件，其实不是。它实际的功能应该是一个带打包功能的 事件队列，，，或者说是一个像指定的类似EventQueue的东东中投递事件。从接口上看，这个东东能投递TimerEvent  DataBaseEvent  SocketAcceptEvent  SocketReadEvent  SocketCloseEvent

应为它本身持有一个CQueueService指针，所以本身并不负责数据管理，只负责打包并添加到CQueueService实例中去。

先看定时器事件：

 //定时器事件

 2http://www.cppblog.com/Images/OutliningIndicators/None.gif    bool PostTimerEvent(WORD wTimerID, WPARAM wBindParam)

 3http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockStart.gif    {

 4http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif        //效验参数

 5http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif        ASSERT(m\_pIQueueService!=NULL);

 6http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif        if (m\_pIQueueService==NULL) return false;

 7http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif

 8http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif        //缓冲锁定

 9http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif        CThreadLockHandle BufferLockHandle(&m\_BufferLock);

10http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif

11http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif        //投递消息

12http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif        NTY\_TimerEvent \* pTimerEvent=(NTY\_TimerEvent \*)m\_cbBuffer;

13http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif        pTimerEvent->wTimerID=wTimerID;

14http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif        pTimerEvent->wBindParam=wBindParam;

15http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif        m\_pIQueueService->AddToQueue(EVENT\_TIMER,m\_cbBuffer,sizeof(NTY\_TimerEvent));

16http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif

17http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif        return true;

18http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockEnd.gif    }

嗯哼，代码质量很高哈，，，

入口都有断言在debug阶段帮助暴露错误，断言之后还有if语句再次判断，这样即使release版本错误也不会被漏掉，，，

关于事件的封装，在这里其实已经分层了，，，

这里是第一层：

1http://www.cppblog.com/Images/OutliningIndicators/None.gifstruct tagDataHead

2http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockStart.gif{

3http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    WORD                            wDataSize;                            //数据大小

4http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    WORD                            wIdentifier;                        //类型标识

5http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    DWORD                            dwInsertTime;                        //插入时间

6http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockEnd.gif};

这一层实际上是由CDataStorage负责打包管理，在CQueueServiceEvent执行PostTimerEvent操作的时候会在内部调 用CQueueService的add最终调用到CDataStrorage的Add来打包数据,这里的结构也是非常像网络数据包|- len -|- type -|- data -|。

在来看Post这一层做的数据打包：

1http://www.cppblog.com/Images/OutliningIndicators/None.gif//定时器事件

2http://www.cppblog.com/Images/OutliningIndicators/None.gifstruct NTY\_TimerEvent

3http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockStart.gif{

4http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    WORD                            wTimerID;                            //定时器 ID

5http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    WPARAM                            wBindParam;                            //绑定参数

6http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockEnd.gif};

|- len -|- type -|- sub protocl -|。

其他事件的打包过程都是类似的：

 1http://www.cppblog.com/Images/OutliningIndicators/None.gif//数据库请求事件

 2http://www.cppblog.com/Images/OutliningIndicators/None.gifstruct NTY\_DataBaseEvent

 3http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockStart.gif{

 4http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    WORD                            wIndex;                                //对象索引

 5http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    WORD                            wRoundID;                            //对象标识

 6http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    WORD                            wRequestID;                            //请求标识

 7http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockEnd.gif};

 8http://www.cppblog.com/Images/OutliningIndicators/None.gif

 9http://www.cppblog.com/Images/OutliningIndicators/None.gif//网络应答事件

10http://www.cppblog.com/Images/OutliningIndicators/None.gifstruct NTY\_SocketAcceptEvent

11http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockStart.gif{

12http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    WORD                            wIndex;                                //连接索引

13http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    WORD                            wRoundID;                            //连接标识

14http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    DWORD                            dwClientIP;                            //连接地址

15http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockEnd.gif};

16http://www.cppblog.com/Images/OutliningIndicators/None.gif

17http://www.cppblog.com/Images/OutliningIndicators/None.gif//网络读取事件

18http://www.cppblog.com/Images/OutliningIndicators/None.gifstruct NTY\_SocketReadEvent

19http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockStart.gif{

20http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    WORD                            wIndex;                                //连接索引

21http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    WORD                            wRoundID;                            //连接标识

22http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    WORD                            wDataSize;                            //数据大小

23http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    CMD\_Command                        Command;                            //命令信息

24http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockEnd.gif};

25http://www.cppblog.com/Images/OutliningIndicators/None.gif

26http://www.cppblog.com/Images/OutliningIndicators/None.gif//网络关闭事件

27http://www.cppblog.com/Images/OutliningIndicators/None.gifstruct NTY\_SocketCloseEvent

28http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockStart.gif{

29http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    WORD                            wIndex;                                //连接索引

30http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    WORD                            wRoundID;                            //连接标识

31http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    DWORD                            dwClientIP;                            //连接地址

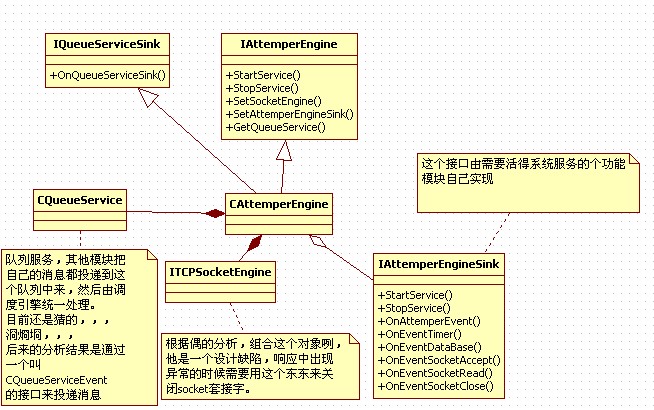
32http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    DWORD                            dwConnectSecond;                    //连接时间

33http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockEnd.gif};

还是一样，一句话总结CQueueServiceEvent：

线程安全的接受各种异步事件，并打包封装好以后插入到关联的制定IQueueService中.

# [调度引擎初步分析](http://www.cppblog.com/Error/articles/148169.html)



CAttempterEngine实现了两个接口：IQueueServiceSink、IAttemperEngine;

通过前面的分析，偶们 了解到，IQueueServiceSink这个接口被是用来处理CQueueService中的数据的，根据上面的UML我们可以看 到，CAttemperEngine关联了一个CQueueService（或者直接点说是：持有了一个CQueueService对象，说组合也成）。 这样的话这个CAttemperEngine暴露出来的接口就只剩下IAttemperEnging了。

1http://www.cppblog.com/Images/OutliningIndicators/None.gif //启动服务

2http://www.cppblog.com/Images/OutliningIndicators/None.gif virtual bool \_\_cdecl StartService();

3http://www.cppblog.com/Images/OutliningIndicators/None.gif //停止服务

4http://www.cppblog.com/Images/OutliningIndicators/None.gif virtual bool \_\_cdecl StopService();

5http://www.cppblog.com/Images/OutliningIndicators/None.gif //设置网络

6http://www.cppblog.com/Images/OutliningIndicators/None.gif virtual bool \_\_cdecl SetSocketEngine(IUnknownEx \* pIUnknownEx);

7http://www.cppblog.com/Images/OutliningIndicators/None.gif //注册钩子

8http://www.cppblog.com/Images/OutliningIndicators/None.gif virtual bool \_\_cdecl SetAttemperEngineSink(IUnknownEx \* pIUnknownEx);

9http://www.cppblog.com/Images/OutliningIndicators/None.gif //获取接口

10http://www.cppblog.com/Images/OutliningIndicators/None.gif virtual void \* \_\_cdecl GetQueueService(const IID & Guid, DWORD dwQueryVer);

这个接口有两处值得单独讨论的：SetSocketEngine、GetQueueService;

SetSocketEngine,后面的分析中还会出现，我觉得这里是一个设计上的失误导致需要暴露socket引擎接口；

GetQueueService 的设计思路可能是说，每个IAttemperEngine接口背后都有一个CQueueService，从以后的分析中可以看到，这个思路是理解整个 kernel的关键。调度引擎应该是一个消息汇总（从个个引擎产生的消息）然后派发到IAttemperEngineSink。因为代码中是没有看到有关 ITimerSink  ISocketSink之类的东东的，，，

整个消息是个引擎产生，然后投递到指定的CQueueService，然后汇总到这里被派发到IAttemperEngineSink出去的，，，

看看CAttemperEngine中处理数据的代码：

 1http://www.cppblog.com/Images/OutliningIndicators/None.gif//队列接口

 2http://www.cppblog.com/Images/OutliningIndicators/None.gifvoid \_\_cdecl CAttemperEngine::OnQueueServiceSink(WORD wIdentifier, void \* pBuffer, WORD wDataSize, DWORD dwInsertTime)

 3http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockStart.gif{

 4http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //内核事件

 5http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    ASSERT(m\_pIAttemperEngineSink!=NULL);

 6http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    switch (wIdentifier)

 7http://www.cppblog.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif    {

 8http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    case EVENT\_TIMER:            //定时器事件

 9http://www.cppblog.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif        {

10http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            //效验参数

11http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            ASSERT(wDataSize==sizeof(NTY\_TimerEvent));

12http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            if (wDataSize!=sizeof(NTY\_TimerEvent)) return;

13http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif

14http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            //处理消息

15http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            NTY\_TimerEvent \* pTimerEvent=(NTY\_TimerEvent \*)pBuffer;

16http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            m\_pIAttemperEngineSink->OnEventTimer(pTimerEvent->wTimerID,pTimerEvent->wBindParam);

17http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif

18http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            return;

19http://www.cppblog.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif        }

20http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    case EVENT\_DATABASE:        //数据库事件

21http://www.cppblog.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif        {

22http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            //效验参数

23http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            ASSERT(wDataSize>=sizeof(NTY\_DataBaseEvent));

24http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            if (wDataSize<sizeof(NTY\_DataBaseEvent)) return;

25http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif

26http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            //处理消息

27http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            NTY\_DataBaseEvent \* pDataBaseEvent=(NTY\_DataBaseEvent \*)pBuffer;

28http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            m\_pIAttemperEngineSink->OnEventDataBase(pDataBaseEvent+1,wDataSize-sizeof(NTY\_DataBaseEvent),pDataBaseEvent);

29http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif

30http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            return;

31http://www.cppblog.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif        }

32http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    case EVENT\_SOCKET\_ACCEPT:    //网络应答事件

33http://www.cppblog.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif        {

34http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            //效验大小

35http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            ASSERT(wDataSize==sizeof(NTY\_SocketAcceptEvent));

36http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            if (wDataSize!=sizeof(NTY\_SocketAcceptEvent)) return;

37http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif

38http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            //处理消息

39http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            NTY\_SocketAcceptEvent \* pSocketAcceptEvent=(NTY\_SocketAcceptEvent \*)pBuffer;

40http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            m\_pIAttemperEngineSink->OnEventSocketAccept(pSocketAcceptEvent);

41http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif

42http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            return;

43http://www.cppblog.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif        }

44http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    case EVENT\_SOCKET\_READ:        //网络读取事件

45http://www.cppblog.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif        {

46http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            //效验大小

47http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            NTY\_SocketReadEvent \* pSocketReadEvent=(NTY\_SocketReadEvent \*)pBuffer;

48http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            ASSERT(wDataSize>=sizeof(NTY\_SocketReadEvent));

49http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            ASSERT(wDataSize==(sizeof(NTY\_SocketReadEvent)+pSocketReadEvent->wDataSize));

50http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            if (wDataSize<sizeof(NTY\_SocketReadEvent)) return;

51http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            if (wDataSize!=(sizeof(NTY\_SocketReadEvent)+pSocketReadEvent->wDataSize)) return;

52http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif

53http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            //处理消息

54http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            bool bSuccess=false;

55http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            try

56http://www.cppblog.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif            {

57http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif                bSuccess=m\_pIAttemperEngineSink->OnEventSocketRead(pSocketReadEvent->Command,pSocketReadEvent+1,pSocketReadEvent->wDataSize,pSocketReadEvent);

58http://www.cppblog.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif            }

59http://www.cppblog.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif            catch (http://www.cppblog.com/Images/dot.gif)    { }

60http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            if (bSuccess==false) m\_pITCPSocketEngine->CloseSocket(pSocketReadEvent->wIndex,pSocketReadEvent->wRoundID);

61http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif

62http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            return;

63http://www.cppblog.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif        }

64http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    case EVENT\_SOCKET\_CLOSE:    //网络关闭事件

65http://www.cppblog.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif        {

66http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            //效验大小

67http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            ASSERT(wDataSize==sizeof(NTY\_SocketCloseEvent));

68http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            if (wDataSize!=sizeof(NTY\_SocketCloseEvent)) return;

69http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif

70http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            //处理消息

71http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            NTY\_SocketCloseEvent \* pSocketCloseEvent=(NTY\_SocketCloseEvent \*)pBuffer;

72http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            m\_pIAttemperEngineSink->OnEventSocketClose(pSocketCloseEvent);

73http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif

74http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif            return;

75http://www.cppblog.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif        }

76http://www.cppblog.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif    }

77http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif

78http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //其他事件

79http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    m\_pIAttemperEngineSink->OnAttemperEvent(wIdentifier,pBuffer,wDataSize,dwInsertTime);

80http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif

81http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    return;

82http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockEnd.gif}

这个函数中一个很重要的参数：wIdentifier；

可以来追溯一下他的源头：

1http://www.cppblog.com/Images/OutliningIndicators/None.gif//数据消息

 2http://www.cppblog.com/Images/OutliningIndicators/None.gifvoid CQueueService::OnQueueServiceThread(const tagDataHead & DataHead, void \* pBuffer, WORD wDataSize)

 3http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockStart.gif{

 4http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    ASSERT(m\_pIQueueServiceSink!=NULL);

 5http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    try

 6http://www.cppblog.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif    {

 7http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif        m\_pIQueueServiceSink->OnQueueServiceSink(DataHead.wIdentifier,pBuffer,DataHead.wDataSize,DataHead.dwInsertTime);

 8http://www.cppblog.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif    }

 9http://www.cppblog.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif    catch (http://www.cppblog.com/Images/dot.gif) {}

10http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    return;

11http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockEnd.gif}

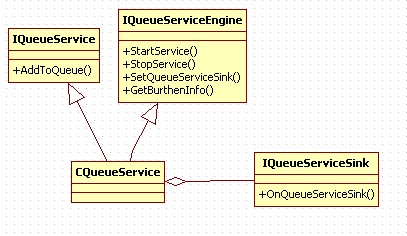
可以看见他是直接保存在最底层的那个DataStroage里边的，这个在上一章分析中可以看到。（个引擎利用CQueueServiceEvent  Post数据的时候就携带了类型信息）

另 外一个要注意的点是对socket事件的处理，我之前认为调度引擎组合了一个socket引擎是一个设计缺陷，应为这里的在处理socket read事件的时候如果异常了直接直接使用引擎来关闭socket而不是调用socket sink的指定接口。猜想也许是不希望客户端直接处理socket句柄吧，，，

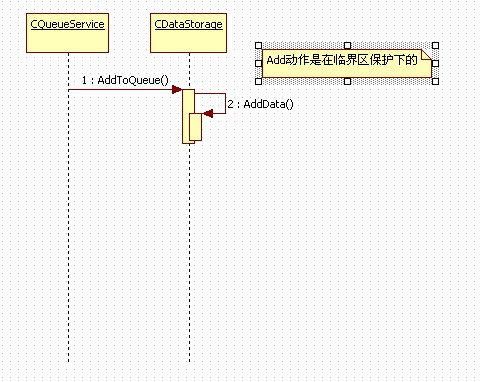
还是用一句话描述下调度引擎：

调度引擎的工作可以这样描述消息汇总、派发。其他引擎通过CQueueServiceEvent将消息post到调度引擎上来（通过共享同一个CQueueService），然后由调度引擎集中派发出去

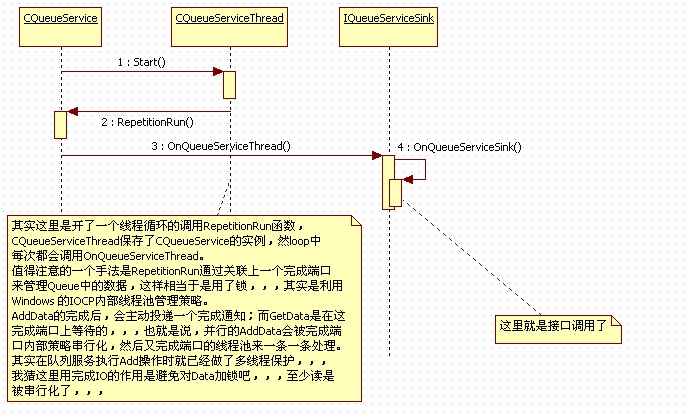
# [ServerKernel中的IQueueService接口分析](http://www.cppblog.com/Error/articles/147691.html)



添加：



处理



关键代码解析：

1http://www.cppblog.com/Images/OutliningIndicators/None.gif//开始服务

 2http://www.cppblog.com/Images/OutliningIndicators/None.gifbool \_\_cdecl CQueueService::StartService()

 3http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockStart.gif{

 4http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //效验参数

 5http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    ASSERT(m\_bService==false);

 6http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    ASSERT(m\_hCompletionPort==NULL);

 7http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    ASSERT(m\_pIQueueServiceSink!=NULL);

 8http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif

 9http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //建立完成端口

10http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //! 需要追踪一下这里的完成端口句柄

11http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //! 首先分析这里的创建，INVALID\_HANDLE\_VALUE表示没有关联任何文件句柄，也就是

12http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //! 说不存在某个实现某个操作完成以后系统自动给这个完成端口post一个完成消息的概念

13http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //! 然后这里用限制工作线程数是1，也就是同意时刻只会有一条线程受到完成消息

14http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    m\_hCompletionPort=CreateIoCompletionPort(INVALID\_HANDLE\_VALUE,NULL,NULL,1);

15http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    if (m\_hCompletionPort==NULL) throw TEXT("队列对象完成端口创建失败");

16http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif

17http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //启动线程

18http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    if (m\_QueueServiceThread.InitThread(m\_hCompletionPort)==false) throw TEXT("队列对象线程初始化失败");

19http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    if (m\_QueueServiceThread.StartThead()==false) throw TEXT("队列对象线程启动失败");

20http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif

21http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //设置参数

22http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    m\_bService=true;

23http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif

24http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    return true;

25http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockEnd.gif}

m\_QueueServiceThread.InitThread(m\_hCompletionPort);

他悄悄的给完成端口句柄扔给了服务线程，偶们去看看服务线程拿这玩意干了啥，，

//运行函数

http://www.cppblog.com/Images/OutliningIndicators/None.gifbool CQueueServiceThread::RepetitionRun()

http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockStart.gif{

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //效验参数

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    ASSERT(m\_hCompletionPort!=NULL);

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //变量定义

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    DWORD dwThancferred=0;

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    OVERLAPPED \* pOverLapped=NULL;

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    CQueueService \* pQueueService=NULL;

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //等待完成端口

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    if (GetQueuedCompletionStatus(m\_hCompletionPort,&dwThancferred,(PULONG\_PTR)&pQueueService,&pOverLapped,INFINITE))

http://www.cppblog.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif    {

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif        //判断退出

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif        if (pQueueService==NULL) return false;

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif        //获取数据

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif        tagDataHead DataHead;

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif        bool bSuccess=pQueueService->GetData(DataHead,m\_cbBuffer,sizeof(m\_cbBuffer));

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif        ASSERT(bSuccess==true);

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif        //处理数据

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif        if (bSuccess==true) pQueueService->OnQueueServiceThread(DataHead,m\_cbBuffer,DataHead.wDataSize);

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif        return true;

http://www.cppblog.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif    }

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    return false;

http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockEnd.gif}

秘密在此：

GetQueuedCompletionStatus(m\_hCompletionPort,&dwThancferred,(PULONG\_PTR)&pQueueService,&pOverLapped,INFINITE)

m\_hCompletionPort是通过线程参数传递进来的，dwThancferred和pQueueService是在之前CQueueService::AddToQueue()中Post过来的参数，，，

//加入数据

http://www.cppblog.com/Images/OutliningIndicators/None.gifbool \_\_cdecl CQueueService::AddToQueue(WORD wIdentifier, void \* const pBuffer, WORD wDataSize)

http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockStart.gif{

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    CThreadLockHandle LockHandle(&m\_ThreadLock);

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    m\_DataStorage.AddData(wIdentifier,pBuffer,wDataSize);

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    PostQueuedCompletionStatus(m\_hCompletionPort,wDataSize,(ULONG\_PTR)this,NULL);

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    return true;

http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockEnd.gif}

数据经过完成端口在CQueueServiceThread绕一圈以后会回到CQueueService::OnQueueServiceThread()

/数据消息

http://www.cppblog.com/Images/OutliningIndicators/None.gifvoid CQueueService::OnQueueServiceThread(const tagDataHead & DataHead, void \* pBuffer, WORD wDataSize)

http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockStart.gif{

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    ASSERT(m\_pIQueueServiceSink!=NULL);

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    try

http://www.cppblog.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif    {

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif        m\_pIQueueServiceSink->OnQueueServiceSink(DataHead.wIdentifier,pBuffer,DataHead.wDataSize,DataHead.dwInsertTime);

http://www.cppblog.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif    }

http://www.cppblog.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif    catch (http://www.cppblog.com/Images/dot.gif) {}

http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    return;

http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockEnd.gif}

这样队列服务就完成了他的使命，将执行的消息，异步的交给指定接口去处理，，，

稍后将对列队列服务做个宏观上的的分析，，，下班时间到，，，回去继续，，，

用一句话来描述QueueService模块就是：

将队列的插入和处理通过关联到完成端口上以实现在IOCP线程池管理下异步处理；

Add以后投递完成消息，在完成端口上监听的线程受到消息以后读取数据并处理；

# [工具类：收发包辅助类 CSendPacketHelpr CRecvPacketHelpr](http://www.cppblog.com/Error/articles/150092.html)

应为协议都是遵循：len type data这样的结构，所以打包发送和接收解包可以有通用的做法，，，

//////////////////////////////////////////////////////////////////////////

//数据描述头

struct tagDataDescribe

{

    WORD                            wDataSize;                        //数据大小

    WORD                            wDataDescribe;                    //数据描述

};

//////////////////////////////////////////////////////////////////////////

//数据包辅助类（多包组合成一包）

class CSendPacketHelper

{

    //变量定义

protected:

    WORD                                m\_wDataSize;                    //数据大小

    WORD                                m\_wBufferSize;                    //缓冲大小

    BYTE                                \* m\_pcbBuffer;                    //缓冲指针

    //函数定义

public:

    //构造函数

    CSendPacketHelper(void \* pcbBuffer, WORD wBufferSize)

    {

        m\_wDataSize=0;

        m\_wBufferSize=wBufferSize;

        m\_pcbBuffer=(BYTE \*)pcbBuffer;

    }

    //析构函数

    virtual ~CSendPacketHelper() {}

    //功能函数

public:

    //清理数据

    void CleanData() { m\_wDataSize=0; }

    //获取大小

    WORD GetDataSize() { return m\_wDataSize; }

    //获取缓冲

    void \* GetDataBuffer() { return m\_pcbBuffer; }

    //插入数据（如果缓冲区长度不够返回false）

    // wDataSize, wDataType 实际上就是DataDescribe

    bool AddPacket(void \* pData, WORD wDataSize, WORD wDataType)

    {

        //效验大小

        ASSERT((wDataSize+sizeof(tagDataDescribe)+m\_wDataSize)<=m\_wBufferSize);

        if ((wDataSize+sizeof(tagDataDescribe)+m\_wDataSize)>m\_wBufferSize) return false;

        //插入数据

        ASSERT(m\_pcbBuffer!=NULL);

        tagDataDescribe \* pDataDescribe=(tagDataDescribe \*)(m\_pcbBuffer+m\_wDataSize);

        pDataDescribe->wDataSize=wDataSize;

        pDataDescribe->wDataDescribe=wDataType;

        //插入数据

        if (wDataSize>0)

        {

            ASSERT(pData!=NULL);

            CopyMemory(pDataDescribe+1,pData,wDataSize);

        }

        //设置数据

        m\_wDataSize+=sizeof(tagDataDescribe)+wDataSize;

        return true;

    }

};

/////////////////////////////////////////////////////////////////////////////////////////

//数据包辅助类

class CRecvPacketHelper

{

    //变量定义

protected:

    WORD                                m\_wDataPos;                        //数据点

    WORD                                m\_wDataSize;                    //数据大小

    BYTE                                \* m\_pcbBuffer;                    //缓冲指针

    //函数定义

public:

    //构造函数

    CRecvPacketHelper(LPCVOID pcbBuffer, WORD wDataSize)

    {

        m\_wDataPos=0;

        m\_wDataSize=wDataSize;

        m\_pcbBuffer=(BYTE \*)pcbBuffer;

    }

    //析构函数

    virtual ~CRecvPacketHelper() {}

    //功能函数

public:

    //获取数据（返回一包数据）

    // @DataDescribe：数据描述（len type）

    // @return: 数据

    void \* GetData(tagDataDescribe & DataDescribe)

    {

        //效验数据

        if (m\_wDataPos>=m\_wDataSize)

        {

            ASSERT(m\_wDataPos==m\_wDataSize);

            DataDescribe.wDataSize=0;

            DataDescribe.wDataDescribe=DTP\_NULL;

            return NULL;

        }

        //获取数据

        ASSERT((m\_wDataPos+sizeof(tagDataDescribe))<=m\_wDataSize);

        CopyMemory(&DataDescribe,m\_pcbBuffer+m\_wDataPos,sizeof(tagDataDescribe));

        ASSERT((m\_wDataPos+sizeof(tagDataDescribe)+DataDescribe.wDataSize)<=m\_wDataSize);

        //效验数据

        if ((m\_wDataPos+sizeof(tagDataDescribe)+DataDescribe.wDataSize)>m\_wDataSize)

        {

            DataDescribe.wDataSize=0;

            DataDescribe.wDataDescribe=DTP\_NULL;

            return NULL;

        }

        //设置数据

        void \* pData=NULL;

        if (DataDescribe.wDataSize>0) pData=m\_pcbBuffer+m\_wDataPos+sizeof(tagDataDescribe);

        m\_wDataPos+=sizeof(tagDataDescribe)+DataDescribe.wDataSize;

        return pData;

    };

};

# [协议分析：GamePlaza的网络协议（收）](http://www.cppblog.com/Error/articles/150109.html)

通过对基础库的分析可以知道，客户端通过IClientSocket、IClientSocketSink和服务器通讯。  
现在把问题集中起来，只分析客户端和服务器之间的接口，也就是IO  
当前分析的模式是GamePlaza  
CPlazaViewItem  CRoomViewItem这两个实现了IClientSocketSink接口，也就是说他们能接受服务器消息；  
IClientSocketSink能响应三个事件OnSocketConnect  OnSocketRead  OnSocketClose  
主要的数据交换应该是依赖OnSocketRead  
CPlazaViewItem  
    OnSocketRead    
      MDM\_GP\_LOGON  //登陆消息  
        SUB\_GP\_LOGON\_SUCCESS  //登陆成功    
          CMD\_GP\_LogonSuccess  
          (接下来的叫扩展信息)  
          DTP\_USER\_ACCOUNTS  //用户帐户  
            数据由DataDescribe描述  
          DTP\_USER\_PASS  //用户密码  
            数据由DataDescribe描述  
          DTP\_USER\_GROUP\_NAME  //社团名字  
            数据由DataDescribe描述  
        SUB\_GP\_LOGON\_ERROR  //登陆失败  
      MDM\_GP\_SERVER\_LIST  //列表消息  
        SUB\_GP\_LIST\_TYPE  //类型信息  
          tagGameType数组   
        SUB\_GP\_LIST\_KIND  //种类消息  
          tagGameKind数组  
        SUB\_GP\_LIST\_PROCESS  //进程信息  
          tagGameProcess数组  
        SUB\_GP\_LIST\_STATION  //站点消息  
          tagGameStation数组  
        SUB\_GP\_LIST\_SERVER  //服务器房间  
          tagGameServer数组  
        SUB\_GP\_LIST\_FINISH  //列表发送完成  
        SUB\_GP\_LIST\_CONFIG  //列表配置  
          CMD\_GP\_ListConfig  
      MDM\_GP\_SYSTEM  //系统消息  
        SUB\_GP\_VERSION  //版本信息  
          CMD\_GP\_Version  
CRoomViewItem  
    OnSocketRead   
        MDM\_GR\_LOGON  //登录消息  
            SUB\_GR\_LOGON\_SUCCESS  //登录成功  
            SUB\_GR\_LOGON\_ERROR  //登录失败  
            SUB\_GR\_LOGON\_FINISH  //登录完成  
        MDM\_GR\_USER  //用户消息  
            SUB\_GR\_USER\_COME  //用户进入  
                tagUserInfoHead  
                    （附加）  
                    DTP\_USER\_ACCOUNTS  //用户帐户  
                    数据由DataDescribe描述  
                    DTP\_USER\_GROUP\_NAME  //社团名字  
                    数据由DataDescribe描述  
            SUB\_GR\_USER\_STATUS  //用户状态  
                CMD\_GR\_UserStatus  
            SUB\_GR\_USER\_SCORE  //用户分数  
                CMD\_GR\_UserScore  
            SUB\_GR\_SIT\_FAILED  //坐下失败  
                CMD\_GR\_SitFailed  
            SUB\_GR\_USER\_CHAT  //用户聊天  
                CMD\_GR\_UserChat  
            SUB\_GR\_USER\_WISPER  //用户私语  
                CMD\_GR\_Wisper  
            SUB\_GR\_USER\_INVITE  //邀请玩家  
                CMD\_GR\_UserInvite  
        MDM\_GR\_INFO  //配置信息  
            SUB\_GR\_SERVER\_INFO  //房间信息  
                CMD\_GR\_ServerInfo  
            SUB\_GR\_COLUMN\_INFO  //列表解释  
                CMD\_GR\_ColumnInfo  
            SUB\_GR\_CONFIG\_FINISH  //配置完成  
        MDM\_GR\_STATUS  //状态信息  
            SUB\_GR\_TABLE\_INFO  //桌子信息   
                CMD\_GR\_TableInfo  
            SUB\_GR\_TABLE\_STATUS  //桌子状态  
                CMD\_GR\_TableStatus  
        MDM\_GR\_SYSTEM  //系统消息  
            SUB\_GR\_MESSAGE  //系统消息  
                CMD\_GR\_Message  
        MDM\_GR\_SERVER\_INFO  //房间信息  
            SUB\_GR\_ONLINE\_COUNT\_INFO  //在线信息  
                tagOnLineCountInfo  
        MDM\_GF\_GAME  //游戏消息  
        MDM\_GF\_FRAME  //框架消息  
            IPC\_SocketPackage

# [协议分析-服务器接收](http://www.cppblog.com/Error/articles/150207.html)

在服务器哪里,网络通信有两个接口:IClientSocketSink IAttemperEngineSink  
IClientSocketSink一般用用于网络服务器之间通信  
IAttemperEngineSink用于和客户端通信

// logon server  
CAttemperEngineSink::OnEventSocketRead  
    MDM\_GP\_LOGON  //登录消息  
        SUB\_GP\_LOGON\_ACCOUNTS  //帐号登录  
      CMD\_GP\_LogonByAccounts  
  SUB\_GP\_LOGON\_USERID  //ID 登录  
   CMD\_GP\_LogonByUserID  
  SUB\_GP\_REGISTER\_ACCOUNTS  //注册帐号  
      CMD\_GP\_RegisterAccounts  
     
// gamer server  
CAttemperEngineSink::OnEventSocketRead  
 MDM\_GR\_LOGON  //登录消息  
        SUB\_GP\_LOGON\_ACCOUNTS  //帐号登录  
      CMD\_GP\_LogonByAccounts  
  SUB\_GP\_LOGON\_USERID  //ID 登录  
   CMD\_GP\_LogonByUserID  
 MDM\_GR\_USER  //用户消息  
  SUB\_GR\_USER\_SIT\_REQ  //坐下桌子  
  SUB\_GR\_USER\_LOOKON\_REQ  //旁观桌子  
   CMD\_GR\_UserSitReq  
  SUB\_GR\_USER\_STANDUP\_REQ  //起立请求  
  SUB\_GR\_USER\_LEFT\_GAME\_REQ  //强退请求  
  SUB\_GR\_USER\_CHAT  //聊天消息  
   CMD\_GR\_UserChat  
  SUB\_GR\_USER\_WISPER  //私语消息  
   CMD\_GR\_Wisper  
  SUB\_GR\_USER\_RULE  //用户规则  
   CMD\_GR\_UserRule  
  SUB\_GR\_USER\_INVITE\_REQ  //邀请用户  
   CMD\_GR\_UserInviteReq  
 MDM\_GR\_MANAGER  //管理消息  
  SUB\_GR\_SEND\_WARNING: //警告消息  
   CMD\_GR\_SendWarning  
  SUB\_GR\_LOOK\_USER\_IP: //查看地址  
      CMD\_GR\_LookUserIP  
  SUB\_GR\_KILL\_USER:  //踢出用户  
   CMD\_GR\_KillUser  
  SUB\_GR\_LIMIT\_ACCOUNS: //禁用帐户  
   CMD\_GR\_LimitAccounts  
  SUB\_GR\_SET\_USER\_RIGHT: //用户权限  
   CMD\_GR\_SetUserRight  
  SUB\_GR\_SEND\_MESSAGE: //发送消息  
   CMD\_GR\_SendMessage  
  SUB\_GR\_OPTION\_SERVER: //房间设置  
   CMD\_GR\_OptionServer  
 MDM\_GF\_FRAME  //框架消息  
 MDM\_GF\_GAME  //游戏消息

// center server  
CAttemperEngineSink::OnEventSocketRead  
 MDM\_CS\_SERVER\_LIST:  //列表消息  
  SUB\_CS\_GET\_SERVER\_LIST:  //获取列表  
 MDM\_CS\_SERVER\_MANAGER: //房间管理   
  SUB\_CS\_REG\_GAME\_SERVER:  //房间注册  
  SUB\_CS\_UNREG\_GAME\_SERVER:  //注销房间  
  SUB\_CS\_SERVER\_ONLINE\_COUNT:  //更新人数