传奇源码分析-服务器端

**LoginGate服务器**

**服务器端：**

1．首先从LoginGate.cpp WinMain分析：

    1) CheckAvailableIOCP : 检查是不是NT，2000的系统（IOCP）

    2) InitInstance: 初始化界面，加载WSAStartup

3)  MainWndProc窗口回调函数.

2．MainWndProc.CPP中分析回调函数MainWndProc

switch (nMsg)

    {

        case \_IDM\_CLIENTSOCK\_MSG:

        case WM\_COMMAND:

        case WM\_CLOSE：

    g\_ssock Local    7000 游戏登陆端口

g\_csock Remote   5000 发送到logsrv服务器上的套接字

1）\_IDM\_CLIENTSOCK\_MSG 消息：处理与logsrv回调通讯事件。

调用：OnClientSockMsg,该函数是一个回调函数：

             当启动服务之后，ConnectToServer函数将(\_IDM\_CLIENTSOCK\_MSG消息 FD\_CONNECT|FD\_READ|FD\_CLOSE)传入WSAAsyncSelect函数。在与hWnd窗口句柄对应的窗口例程中以Windows消息的形式接收网络事件通知。函数OnClientSockMsg，主要完成与logsrv服务器之间的通信（心跳，转发客户端数据包等）

switch (WSAGETSELECTEVENT(lParam))

   {

       case FD\_CONNECT:

       case FD\_CLOSE:

       case FD\_READ:

FD\_CONNECT：(重新连接情况)

  A. CheckSocketError返回正常时：

a). ConnectToServer函数首先在服务启动的时候执行一次。回调

FD\_CONNECT

   b)．连接logsrv时,开启ThreadFuncForMsg线程，把从客户端发送的数据(g\_xMsgQueue, FD\_READ事件读到的logSrv服务器发来的数据) 投递I/O，利用IOCP模型，发送到客户端**。**SleepEx挂起线程。至到一个I/O 完成回调函数被调用。 一个异步过程调用排队到此线程。

ThreadFuncForMsg线程检测(从logSrv收到的g\_xMsgQueue数据包-心跳，处理包)。i/o 投递，利用IOCP发送给客户端。

        if (nSocket = AnsiStrToVal(pszFirst + 1)) //得到socket

       WSASend((SOCKET)nSocket, &Buf, 1, &dwSendBytes, 0, NULL,

c)．终止定时器\_ID\_TIMER\_CONNECTSERVER

KillTimer(g\_hMainWnd, \_ID\_TIMER\_CONNECTSERVER);

d)．设置\_ID\_TIMER\_KEEPALIVE定时器 （心跳数据包）

SetTimer(g\_hMainWnd, \_ID\_TIMER\_KEEPALIVE

     调用定时器回调函数OnTimerProc: 定时发关心跳数据包到logsrv服务器。SendExToServer(PACKET\_KEEPALIVE);

B**.** 如果socket断开，设置\_ID\_TIMER\_CONNECTSERVER定时器

ConnectToServer尝试重新连接服务器。

                   \_ID\_TIMER\_CONNECTSERVER, (TIMERPROC)OnTimerProc);

              FD\_CLOSE:

                     断开与logsrv服务器SOCKET连接，OnCommand(IDM\_STOPSERVICE, 0); 回调函数处理IDM\_STOPSERVICE。

              FD\_READ:

                     接收logsrv服务器发送的数据包（心跳,登陆验证，selCur服务器地址），把数据加入缓冲区(g\_xMsgQueue)中。

2）WM\_COMMAND:

          IDM\_STARTSERVICE: 启动服务(IOCP模型Server响应客户端请求)

          IDM\_STOPSERVICE: 停止服务(IOCP模型Server)

    3）WM\_CLOSE:

          IDM\_STOPSERVICE: 停止服务(IOCP模型Server)

          WSACleanup();

PostQuitMessage(0); //WM\_DESTROY消息

**IDM\_STARTSERVICE: 启动服务(IOCP模型Server响应客户端请求)**

InitServerSocket：函数：

1) AcceptThread线程：

      Accept之后生成一个CSessionInfo对象，pNewUserInfo->sock = Accept; 客户端Socket值赋值给结构体。记录客户相关信息。

新的套接字句柄用CreateIoCompletionPort关联到完成端口，然后发出一个异步的WSASend或者WSARecv调用(pNewUserInfo->Recv();接收客户端消息)，因为是异步函数，WSASend/WSARecv会马上返回，实际的发送或者接收数据的操作由WINDOWS系统去做。然后把CSessionInfo对象加入g\_xSessionList中。向logsrv服务器发送用户Session信息。打包规则‘%0socket/ip$\0’

**在客户accept之后，总投递一个I/O(recv),然后把相应的数据发往logsrv服务器。**

   2) CreateIOCPWorkerThread函数：

          调用CreateIoCompletionPort 并根据处理器数量，创建一个或多个ServerWorkerThread线程。

ServerWorkerThread线程工作原理：

循环调用GetQueuedCompletionStatus()函数来得到IO操作结果。阻塞函数。当WINDOWS系统完成WSASend或者WSArecv的操作，把结果发到完成端口。GetQueuedCompletionStatus()马上返回,并从完成端口取得刚完成的WSASend/WSARecv的结果。然后接着发出WSASend/WSARecv，并继续下一次循环阻塞在GetQueuedCompletionStatus()这里。

a). pSessionInfo为空或者dwBytesTransferred =0 ,在客户端close socket，发相应数据包(异常）到logsrv服务器(X命令-数据包)，关闭客户端套按字。

         b). while ( pSessionInfo->HasCompletionPacket() ) 如果数据验证正确，就转发数据包(A命令-数据包) logsrv服务器。

 c). if (pSessionInfo->Recv() 继续投递I/O操作。

 总结：

我们不停地发出异步的WSASend/WSARecv IO操作，具体的IO处理过程由WINDOWS系统完成，WINDOWS系统完成实际的IO处理后，把结果送到完成端口上（如果有多个IO都完成了，那么就在完成端口那里排成一个队列）。我们在另外一个线程里从完成端口不断地取出IO操作结果，然后根据需要再发出WSASend/WSARecv IO操作。

IDM\_STOPSERVICE**: 停止服务(IOCP模型Server响应客户端请求)**

**Close -> OnCommand(IDM\_STOPSERVICE, 0L); ->g\_fTerminated = TRUE; 线程退出。**

    if (g\_hAcceptThread != INVALID\_HANDLE\_VALUE)

    {

        TerminateThread(g\_hAcceptThread, 0);

         WaitForSingleObject(g\_hAcceptThread, INFINITE); //IOCP的Accept线程

         CloseHandle(g\_hAcceptThread);

         g\_hAcceptThread = INVALID\_HANDLE\_VALUE;

     }

     if (g\_hMsgThread != INVALID\_HANDLE\_VALUE)

     {

         TerminateThread(g\_hMsgThread, 0); //窗口例程网络事件回调线程

         WaitForSingleObject(g\_hMsgThread, INFINITE);

         CloseHandle(g\_hMsgThread);

         g\_hMsgThread = INVALID\_HANDLE\_VALUE;

     }

     ClearSocket(g\_ssock);

     ClearSocket(g\_csock);

     CloseHandle(g\_hIOCP);

总结：

LoginGate（登录网关服务器），接受客户端连接，并且把用户ID，密码直接发送到LoginSvr服务器中，由LoginSrv服务器验证之后，发送数据包返回给客户端。LoginGate之间是通过定时器，定时发送“心跳”数据。验证服务器存活的。客户端与服务器端的数据在传输中，是进行过加密的。

向loginSrv发送‘%A’+Msg+‘$0’消息： 转发客户端消息。

                 ‘%X’+Msg+‘$0’消息： 发送用户连接消息，增加到用户列表。

                 ‘%O’+Msg+‘$0’消息： 发送用户上线消息。

主要流程：

服务启动后，LoginGate启动了AcceptThread,和ServerWorkerThread线程，AcceptThread线程接收客户端连接，并把session信息发送给loginSrv服务器，ServerWorkerThread线程从完成端口取得刚完成的WSASend/WSARecv的结果后，把客户端数据转发给loginSrv服务器。服务启动时，WSAAsyncSelect模型连接到loginSrv服务器中。一旦连接成功，就启动ThreadFuncForMsg线程，该线程从g\_xMsgQueue(FD\_READ事件读到的loginSrv服务器发来的数据)中取出loginSrv服务器处理过的数据。投递I/O，利用IOCP模型，发送到客户端。

ServerWorkerThread转发客户端数据 -> WSAAsyncSelect的Read读loginSrv处理后返回的数据-> ThreadFuncForMsg线程，投递WSASend消息，由Windows处理（IOCP），发送数据给客户端。

**LoginSvr服务器**

功能:(1)通过数据库的直接连接验证用户名密码正确;

     (2)在接受Gate open user信息后，创建用户CuserInfo对象保存用户在CgateInfo,主要保存用户需要验证的信息(如ID,密码,是否验证通过);

     (3)接受Gate转发的请求server list消息,下发server list列表,前提条件有CuserInfo对象在Gate中(通过client 在gate上的socketid作为标示判断)

(3)如果失败,返回错误，如果成功，同时返回server\_list.

(4) 数据清理：由client从gate断开时,清理Gate上的CUserInfo数据

g\_gcSock Local    5500端口

1．首先从LoginSvr.cpp  WinMain分析：

   1) CheckAvailableIOCP : 检查是不是NT，2000的系统（IOCP）

   2) InitInstance: 初始化界面，加载WSAStartup

       GetDBManager()->Init( InsertLogMsg, "Mir2\_Account", "sa", "prg" );

       数据库管理类，做底层数据库操作。

3)       MainWndProc窗口回调函数OnCommand:

IDM\_STARTSERVICE:

创建LoadAccountRecords线程

a). UPDATE TBL\_ACCOUNT重置帐户验证状态。

b). 读服务器列表(TBL\_SERVERINFO, selGate服务器)，加入g\_xGameServerList

遍历xGameServerList列表，把服务器信息加入到一个字符数组g\_szServerList中。

c). 启动InitServerThreadForMsg线程。

          d). 调用InitServerSocket函数创建两个线程：

           AcceptThread线程：

           ServerWorkerThread线程：

**调用InitServerSocket函数创建两个线程：**

    1) AcceptThread线程：

Accept之后生成一个CGateInfo对象，CGateInfo->sock = Accept; 客户端Socket值赋值给结构体。记录客户相关信息。新的套接字句柄用CreateIoCompletionPort关联到完成端口，然后发出一个异步的WSASend或者WSARecv调用(pNewUserInfo->Recv();接收客户端消息)，因为是异步函数，WSASend/WSARecv会马上返回，实际的发送或者接收数据的操作由WINDOWS系统去做。然后把CGateInfo对象加入g\_xGateList中。在客户accept之后，投递一个I/O(recv)。

分析一下g\_xGateList发现，每个CGateInfo里有sock; xUserInfoList，g\_SendToGateQ，该网关的相关信息依次（网关对应的sock, 用户列列信息，消息队列），可以为多个LoginGate登录网关服务。

2) ServerWorkerThread线程：

ServerWorkerThread线程工作原理：

循环调用GetQueuedCompletionStatus()函数来得到IO操作结果。阻塞函数。当WINDOWS系统完成WSASend或者WSArecv的操作，把结果发到完成端口。GetQueuedCompletionStatus()马上返回,并从完成端口取得刚完成的WSASend/WSARecv的结果。然后接着发出WSASend/WSARecv，并继续下一次循环阻塞在GetQueuedCompletionStatus()这里。

a).if (g\_fTerminated) 线程结束前：循环遍历g\_xGateList，取出pGateInfo关闭套接字，并删除节点。dwBytesTransferred =0 ,关闭**该服务器**套接字。

b).while ( pGateInfo->HasCompletionPacket() ) 验证消息格式。

case '-': 发送心跳数据包到每个LoginGate服务器。

case 'A':  处理每个LoginGat服务器转发的客户端的消息增加到各自网关(CGateInfo)g\_SendToGateQ队列中，然后ThreadFuncForMsg线程进行验证后再发送消息到各个LoginGate服务器。

       pGateInfo->ReceiveSendUser(&szTmp[2]);

case 'O': 处理每个网关Accept客户端后增加pUserInfo用户信息到各自网关的xUserInfoList列表中。

      pGateInfo->ReceiveOpenUser(&szTmp[2]);

case 'X': 处理每个网关收到客户端Socket关闭之后发送过来的消息。设置该网关socket相应状态。

      pGateInfo->ReceiveCloseUser(&szTmp[2]);

case 'S':  GameSvr服务器发送的消息，更新TBL\_ACCOUNT，验证字段，说明用户已下线，下次登录必须先到LoginSvr服务器再次验证。

      pGateInfo->ReceiveServerMsg(&szTmp[2]);

case 'M':  GameSvr服务器发送的消息,创建一个用户的消息，把用户ID，密码,名字插入TBL\_ACCOUNT表中插入成功返回SM\_NEWID\_SUCCESS,否则SM\_NEWID\_FAIL，把在信息前加#，信息后加!  不做TBL\_ACCOUNTADD表的添加，只增加TBL\_ACCOUNT表信息。

              ‘A’:是LoginGate 服务器转发客户端消息到g\_xMsgQueue队列, 由ThreadFuncForMsg线程处理后，转发到各个loginGate服务器

继续投递I/O操作。

**启动InitServerThreadForMsg 创建ThreadFuncForMsg线程。**c

收到loginGate服务器发送过来的消息之后，ServerWorkerThread经过数据包分析之后（case 'A'），把客户端的消息，写入g\_SendToGateQ队列中，然后在本线程中再进行处理。

             遍历g\_SendToGateQ队列中数据，验证数据包是否正确（#!字符）根据DefaultMsg.wIdent标志

case CM\_IDPASSWORD:   处理登陆业务

         遍历xUserInfoList用户列表信息，到数据库表TBL\_ACCOUNT中找相应信息，如果失败发送(SM\_ID\_NOTFOUND, SM\_PASSWD\_FAIL)消息，否则发送SM\_PASSOK\_SELECTSERVER+ g\_szServerList（SelGate服务器列表消息）

SelGate服务器列表消息（对应TBL\_SERVERINFO数据库表中数据），供用户选择登录的SelGate服务器。

      CM\_SELECTSERVER: 选择服务器(SelGate)

             遍历xUserInfoList用户列表信息，根据socket,找到用户密钥，消息解密后，遍历g\_xGameServerList列表，把用户选择的SelGate服务器转化为IP地址，发送至LoginGate服务器，再转发至客户端。设置该用户SelServer的标志状态。从该网关的xUserInfoList用户列表中删除该用户。

CM\_ADDNEWUSER:  新注册用户

                    判断用户名是否已存在，失败发送SM\_NEWID\_FAIL消息，成功，写插入表数据，并发送SM\_NEWID\_SUCCESS消息到 LoginGate服务器，转发至客户端。

IDM\_STOPSERVICE**: 停止服务(IOCP模型Server响应客户端请求)**

**Close -> OnCommand(IDM\_STOPSERVICE, 0L); ->g\_fTerminated = TRUE; 三个线程退出。**

主要流程：

服务启动后，LoginSvr启动了AcceptThread,和ServerWorkerThread线程，AcceptThread线程接收loginGate，GameSvr服务器连接，加入g\_xGateList网关列表中，ServerWorkerThread线程从完成端口取得刚完成的WSASend/WSARecv的结果后，进行分析处理两个服务器发送来的消息。服务启动同时，启动ThreadFuncForMsg线程，该线程从g\_xMsgQueue(iocp读到的loginGate服务器发来的数据)中取出数据，处理数据。投递I/O，利用IOCP模型，发送到loginGate服务器。

**SelGate服务器**

注：客户端从LoginSvr服务器得到SelGate服务器IP之后，连接SelGate服务器，进行角

色创建，删除，选择操作，然后发送数据到DBSrv服务器。

g\_ssock  Local    7100客户端登陆端口

g\_csock Remote   5100发送到DBSrv服务器上的套接字

1．首先从SelGate.cpp WinMain分析：

    1) CheckAvailableIOCP : 检查是不是NT，2000的系统（IOCP）

    2) InitInstance: 初始化界面，加载WSAStartup

3) MainWndProc窗口回调函数.

2．MainWndProc.CPP中分析回调函数MainWndProc

switch (nMsg)

    {

        case \_IDM\_CLIENTSOCK\_MSG:

        case WM\_COMMAND:

        case WM\_CLOSE：

1）\_IDM\_CLIENTSOCK\_MSG 消息：

   处理与SelGate回调通讯事件。

调用：OnClientSockMsg,该函数是一个回调函数：

             当启动服务之后，ConnectToServer函数将(\_IDM\_CLIENTSOCK\_MSG消息 FD\_CONNECT|FD\_READ|FD\_CLOSE)传入WSAAsyncSelect函数。在与hWnd窗口句柄对应的窗口例程中以Windows消息的形式接收网络事件通知。函数OnClientSockMsg，主要完成与DBSrv服务器之间的通信（心跳，转发客户端数据包等）

switch (WSAGETSELECTEVENT(lParam))

   {

       case FD\_CONNECT:

       case FD\_CLOSE:

       case FD\_READ:

FD\_CONNECT：(重新连接情况)

    A. CheckSocketError返回正常时：

a). ConnectToServer函数首先在服务启动的时候执行一次。回调

FD\_CONNECT

             b)．连接DBSrv时,开启ThreadFuncForMsg线程，把从客户端发送的数据(g\_xMsgQueue, FD\_READ事件读到的DBSrv服务器发来的数据)投递I/O，利用IOCP模型，发送到客户端。SleepEx挂起线程,至到一个I/O 完成回调函数被调用。一个异步过程调用排队到此线程。

ThreadFuncForMsg线程检测(从DBSrv收到的g\_xMsgQueue数据包-心跳，处理包)。i/o 投递，利用IOCP发送给客户端。

   if (nSocket = AnsiStrToVal(pszFirst + 1)) //得到socket

   WSASend((SOCKET)nSocket, &Buf, 1, &dwSendBytes, 0, NULL, NULL);

c)．终止定时器\_ID\_TIMER\_CONNECTSERVER

KillTimer(g\_hMainWnd, \_ID\_TIMER\_CONNECTSERVER);

d)．设置\_ID\_TIMER\_KEEPALIVE定时器 （心跳数据包）

SetTimer(g\_hMainWnd, \_ID\_TIMER\_KEEPALIVE

     调用定时器回调函数OnTimerProc: 定时发关心跳数据包到DBSrv服务器。SendExToServer(PACKET\_KEEPALIVE);

**B.** 如果socket断开，设置\_ID\_TIMER\_CONNECTSERVER定时器

ConnectToServer尝试重新连接服务器。

                     \_ID\_TIMER\_CONNECTSERVER, (TIMERPROC)OnTimerProc);

            FD\_CLOSE:

                 断开SOCKET连接，OnCommand(IDM\_STOPSERVICE, 0); 回调函数处理IDM\_STOPSERVICE。

            case FD\_READ:

                 接收DBSrv服务器发送的数据包（心跳,登陆验证，selCur服务器地址），把数据加入缓冲区(g\_xMsgQueue)中。

     WM\_COMMAND:

          IDM\_STARTSERVICE: 启动服务(IOCP模型Server响应客户端请求)

          IDM\_STOPSERVICE: 停止服务(IOCP模型Server)

     WM\_CLOSE:

         IDM\_STOPSERVICE: 停止服务(IOCP模型Server)

         WSACleanup();

PostQuitMessage(0); //WM\_DESTROY消息

**IDM\_STARTSERVICE: 启动服务(IOCP模型Server响应客户端请求)**

InitServerSocket：函数：

1) AcceptThread线程：

        Accept之后生成一个CSessionInfo对象，pNewUserInfo->sock = Accept; 客户端Socket值赋值给结构体。记录客户相关信息。

新的套接字句柄用CreateIoCompletionPort关联到完成端口，然后发出一个异步的WSASend或者WSARecv调用(pNewUserInfo->Recv();接收客户端消息)，因为是异步函数，WSASend/WSARecv会马上返回，实际的发送或者接收数据的操作由WINDOWS系统去做。然后把CSessionInfo对象加入g\_xSessionList中。向DBsrv服务器发送用户Session信息。打包规则‘%0socket/ip$\0’

**在客户accept之后，总投递一个I/O(recv),然后把相应的数据发往DBSrv服务器。**

   2) CreateIOCPWorkerThread函数：

         调用CreateIoCompletionPort 并根据处理器数量，创建一个或多个ServerWorkerThread线程。

 ServerWorkerThread线程工作原理：

循环调用GetQueuedCompletionStatus()函数来得到IO操作结果。阻塞函数。当WINDOWS系统完成WSASend或者WSArecv的操作，把结果发到完成端口。GetQueuedCompletionStatus()马上返回,并从完成端口取得刚完成的WSASend/WSARecv的结果。然后接着发出WSASend/WSARecv，并继续下一次循环阻塞在GetQueuedCompletionStatus()这里。

a). pSessionInfo为空或者dwBytesTransferred =0 ,在客户端close socket，发相应数据包(异常）到DBSrv服务器(X命令-数据包)，关闭客户端套按字。

        b). while ( pSessionInfo->HasCompletionPacket() ) 如果数据验证正确，就转发数据包(A命令-数据包) DBSrv服务器。

 c). if (pSessionInfo->Recv() 继续投递I/O操作。

 总结：

我们不停地发出异步的WSASend/WSARecv IO操作，具体的IO处理过程由WINDOWS系统完成，WINDOWS系统完成实际的IO处理后，把结果送到完成端口上（如果有多个IO都完成了，那么就在完成端口那里排成一个队列）。我们在另外一个线程里从完成端口不断地取出IO操作结果，然后根据需要再发出WSASend/WSARecv IO操作。

IDM\_STOPSERVICE**: 停止服务(IOCP模型Server响应客户端请求)**

**Close -> OnCommand(IDM\_STOPSERVICE, 0L); ->g\_fTerminated = TRUE; 线程退出。**

            ClearSocket(g\_ssock);

            ClearSocket(g\_csock);

            CloseHandle(g\_hIOCP);

总结：SelGate（角色处理服务器），接受客户端连接，并且把用户数据包(角色处理)发送到DBSrv服务器中，由DBSrv服务器处理之后，发送数据包返回给客户端。SelGate之间是通过定时器，定时发送“心跳”数据。验证服务器存活的。客户端与服务器端的数据在传输中，是进行过加密的。

向DBSrv发送   ‘%A’+Msg+‘$0’消息： 转发客户端消息。

               ‘%X’+Msg+‘$0’消息： 发送用户连接消息，增加到用户列表。

               ‘%O’+Msg+‘$0’消息： 发送用户上线消息。

主要流程：

服务启动后，SelGate启动了AcceptThread,和ServerWorkerThread线程，AcceptThread线程接收客户端连接，并把session信息发送给DBSrv服务器，ServerWorkerThread线程从完成端口取得刚完成的WSASend/WSARecv的结果后，把客户端数据转发给DBSrv服务器。服务启动时，WSAAsyncSelect模型连接到DBSrv服务器中。一旦连接成功，就启动ThreadFuncForMsg线程，该线程从g\_xMsgQueue(FD\_READ事件读到的DBSrv服务器发来的数据)中取出DBSrv服务器处理过的数据。投递I/O，利用IOCP模型，发送到客户端。

ServerWorkerThread转发客户端数据 -> WSAAsyncSelect的Read读DBSrv处理后返回的数据-> ThreadFuncForMsg线程，投递WSASend消息，由Windows处理（IOCP），发送数据给客户端。

**传奇文件类型格式探讨(一)：**

Wix文件：索引文件，根据索引查找到相应数据地址(数据文件)。

// WIX 文件头格式

typedef struct tagWIXFILEIMAGEINFO

{

    CHAR    szTmp[40];     // 库文件标题 'WEMADE Entertainment inc.' WIL文件头

    INT     nIndexCount;   // 图片数量

    INT\*    pnPosition;    // 位置

}WIXIMAGEINFO, \*LPWIXIMAGEINFO;

我们下载一个Hedit编辑器打开一个Wil文件，分析一下。我们发现Wix文件中，0x23地址(含该地址)以前的内容是都相同的，即为：#INDX v1.0-WEMADE Entertainment inc.

Ofs44 0x2C的地方：存放着0B 00 00 00，高低位转换后为：0xB转换十进制数为11(图片数量)Ofs48 0x30的地方：存放着38 04 00 00，高低位转换后为：0x438 = 1080, 这个就是图象数据的开始位置。

我们用Wil编辑打开对应的Wil文件，发现，果然有11张图片。另外我们发现，在Ofs = 44 -47之间的数据总是38 04 00 00，终于明白，所有的图片起始位置是相同的。

Wil文件: 数据文件。

前面我们说了图象数据的开始位置为0x438 = 1080, 1080中有文件开头的44字节都是相同的。所以，就是说有另外的1036字节是另有用途。1036中有1024是一个256色的调色板。而Wil里面的图片格式都是256色的位图储存。

我们看到图片位置数据为： 20 03 58 02, 转化为十六进制： 0x320, 0x258 刚好就是800\*600大小的图片。07 00 D4 FF为固定值(标识)。图片起始位置为：

Ofs 1088: 0x440 图片大小为480000

起始位置：0x440 1088   终止位置：0x7573F 481087 为了验证数据是否正确，我们通过Wil工具，把第一幅图片导出来，然后用Hedit编辑器打开，经过对比，我们发现，数据一致。大小一致。

    大家看到图片1的结束位置为0fs 481077,减去1080+1 = 480000刚好800\*600大小。

我们用Wil抓图工具打开看一下(确定是800\*600大小)：

我们导出第二张BMP图片

图片的大小为：496\* 361, 我们从Wix中读出第二张图片的索引位置：

根据贴图，我们发现第二张图片的索引位置为： 40 57 07 00，转换为十六进制：0x75740,即为：481088，前面我们讲到第一张图片的结束位置是： 0fs 481077,从Wix中读出来的也刚好为第二张图片的起始位置：

(我们分析Wil中的第二张图片，起始位置：0x75740 481088) ： F0 01 69 01为图片长宽： 0x1F0, 0x169 为496\* 361 。 07 00 D4 FF为固定值(标识)。

我们用工具打开第二张BMP图片，从起始位置，一直选取中至结束，发现刚好选496\* 361字节大小。两边数据对比之后发现一致。知道了图片格式，我们可以写一个抓图片格式的程序了。

**登录处理事件：**

0．WinMain主函数调用g\_xLoginProc.Load();加载图片等初始化，设置g\_bProcState 的状态。

1．CLoginProcess::OnKeyDown-> m\_xLogin.OnKeyDown->g\_xClientSocket.OnLogin;

WSAAsyncSelect模型ID\_SOCKCLIENT\_EVENT\_MSG,因此，(登录，角色选择，游戏逻辑处理)都回调g\_xClientSocket.OnSocketMessage(wParam, lParam)进行处理。

OnSocketMessage函数中：FD\_READ事件中：

2．g\_bProcState判断当前状态，\_GAME\_PROC时，把GameGate的发送过来的消息压入PacketQ队列中，再进行处理。否则则调用OnMessageReceive（虚方法，根据g\_bProcState状态，调用CloginProcess或者是CcharacterProcess的OnMessageReceive方法）。

3．CloginProcess：调用OnSocketMessageRecieve处理返回情况。如果服务器验证失败(SM\_ID\_NOTFOUND, SM\_PASSWD\_FAIL)消息，否则收到SM\_PASSOK\_SELECTSERVER消息（SelGate服务器列表消息）。m\_Progress = PRG\_SERVER\_SELE;进行下一步选择SelGate服务器操作。

4． m\_xSelectSrv.OnButtonDown->CselectSrv. OnButtonUp->

g\_xClientSocket.OnSelectServer(CM\_SELECTSERVER)，得到真正的IP地址。调用OnSocketMessageRecieve处理返回的SM\_SELECTSERVER\_OK消息。并且断开与loginSrv服务器连接。 g\_xClientSocket.DisconnectToServer();设置状态为PRG\_TO\_SELECT\_CHR状态。

**角色选择处理：**

1． WinMain消息循环处理：g\_xLoginProc.RenderScene(dwDelay)-> RenderScroll->

SetNextProc调用

g\_xClientSocket.m\_pxDefProc = g\_xMainWnd.m\_pxDefProcess = &g\_xChrSelProc;

g\_xChrSelProc.Load();

g\_bProcState = \_CHAR\_SEL\_PROC;

   2．g\_xChrSelProc.Load();连接SelGate服务器（从LoginGate服务器得到IP地址）。

g\_xClientSocket.OnQueryChar();查询用户角色信息，发送消息：CM\_QUERYCHR，设置状态为\_CHAR\_SEL\_PROC, m\_Progress = PRG\_CHAR\_SELE; 在OnSocketMessageRecieve函数中接收到SelGate服务器发送的消息。

   3．点击ChrStart按钮：g\_xChrSelProc.OnLButtonDown-> CSelectChr::OnButtonUp->

g\_xClientSocket.OnSelChar->发送CM\_SELCHR消息到SelGate服务器。

4．CClientSocket::OnSocketMessage->CCharacterProcess::OnMessageReceive

(SM\_STARTPLAY) 接受到SelGate服务器发送的GameGate服务器IP地址，并断开与SelGate服务器的连接。m\_xSelectChr.m\_nRenderState = 2;

   5. WinMain消息循环处理：g\_xLoginProc.RenderScene ->

m\_xSelectChr.Render(nLoopTime);-> CSelectChr::Render(INT   nLoopTime)-> m\_nRenderState = m\_nRenderState + 10; 为12-> CCharacterProcess::RenderScene执行

m\_Progress = PRG\_SEL\_TO\_GAME;

    m\_Progress = PRG\_PLAY\_GAME;

SetNextProc();

6．SetNextProc();执行： g\_xGameProc.Load(); g\_bProcState = \_GAME\_PROC;进行游戏状态。

**游戏逻辑处理:**

**1．客户端处理：**

 CGameProcess::Load() 初始化游戏环境，加载地图等操作，调用ConnectToServer（m\_pxDefProc->OnConnectToServer）连接到GameGate游戏网关服务器（DBSrv处理后经SelGate服务器返回的GameGate服务器IP地址）。

     CClientSocket->ConnectToServer调用connect时，由GameGate服务器发送GM\_OPEN消息到GameSrv服务器。WSAAsyncSelect I/O模型回调函数 g\_xClientSocket.OnSocketMessage。然后由m\_pxDefProc->OnConnectToServer()调用CGameProcess::OnConnectToServer()函数，调用：g\_xClientSocket.SendRunLogin。

**2. GameGate服务器ServerWorkerThread处理：**

GameGate服务器ServerWorkerThread收到消息，ThreadFuncForMsg处理数据，生成MsgHdr结构，并设置

MsgHdr.nCode    = 0xAA55AA55; //数据标志

 MsgHdr.wIdent   = GM\_DATA;    //数据类型

**3. GameSrv服务器ServerWorkerThread线程处理**

   GameSrv服务器ServerWorkerThread线程处理调用DoClientCertification设置用户信息，及USERMODE\_LOGIN的状态。并且调用LoadPlayer(CUserInfo\* pUserInfo)函数-> LoadHumanFromDB-> SendRDBSocket发送DB\_LOADHUMANRCD请求，返回该玩家的所有数据信息。

**4. 客户端登录验证(GameSrv服务器的线程ProcessLogin处理）**

  用户的验证是由GameSrv服务器的线程ProcessLogin处理。g\_xReadyUserInfoList2列表中搜索，判断用户是否已经登录，一旦登录就调用LoadPlayer(这里两个参数)：

a. 设置玩家游戏状态。m\_btCurrentMode状态为USERMODE\_PLAYGAME

b. 加载物品，个人设置，魔法等。

c. pUserInfo->m\_pxPlayerObject->Initialize();初始化用户信息，加载用户坐标，方向，地图。

   Initialize执行流程：

1)       AddProcess(this, RM\_LOGON, 0, 0, 0, 0, NULL);加入登录消息。

2)       m\_pMap->AddNewObject 地图中单元格（玩家列表）加入该游戏玩家。OS\_MOVINGOBJECT玩家状态。

3)       AddRefMsg(RM\_TURN 向周围玩家群发 RM\_TURN消息。以玩家自己为中心，以24\*24的区域里，向这个区域所属的块里的所有玩家列表发送消息)广播 AddProcess。

4)       RecalcAbilitys 设置玩家的能力属性（攻击力（手，衣服），武器力量等）。

5)       循环处理本游戏玩家的附属物品，把这些物品的力量加到（手，衣服等）的攻击力量里。

6)       RM\_CHARSTATUSCHANGED消息，通知玩家状态改变消息。

7)       AddProcess(this, RM\_ABILITY, 0, 0, 0, 0, NULL); 等级

AddProcess(this, RM\_SUBABILITY, 0, 0, 0, 0, NULL);

AddProcess(this, RM\_DAYCHANGING, 0, 0, 0, 0, NULL); 校时

AddProcess(this, RM\_SENDUSEITEMS, 0, 0, 0, 0, NULL); 装备

AddProcess(this, RM\_SENDMYMAGIC, 0, 0, 0, 0, NULL); 魔法

         SysMsg(szMsg, 1) 攻击力

并把用户数据从g\_xReadyUserInfoList2列表中删除。

**说明：**

**一旦通过验证，就从验证列表中该玩家，改变玩家状态，LoadPlayer加载用户资源（地图中加入用户信息，向用户24\*24区域内的块内玩家发送上线消息GameSrv广播新玩家上线（坐标）的消息。向该新玩家发送玩家信息（等级，装备，魔法，攻击力等）。**

**5．接受登录成功后，接收GameSrv服务器发送的消息:**

接收GameGate发送的消息：CClientSocket::OnSocketMessage的FD\_READ事件中，PacketQ.PushQ((BYTE\*)pszPacket);把接收到的消息，压入PacketQ队列中。处理PacketQ队列数据是由CGameProcess::Load()时调用OnTimer在CGameProcess::OnTimer中处理的，

处理过程为：

OnMessageReceive;

ProcessPacket();

  ProcessDefaultPacket();

**OnMessageReceive函数;**

1.     判断是否收到心跳数据包，发送'\*'，发送心跳数据包。

2.     调用OnSocketMessageRecieve函数。这个函数里面详细处理了客户端的游戏执行逻辑。如果是‘+’开头（数据包）则调用OnProcPacketNotEncode处理这种类型数据包。否则得到\_TDEFAULTMESSAGE数据包，进行游戏逻辑处理。

        所谓的\_TDEFAULTMESSAGE数据封包：就是由\_TDEFAULTMESSAGE头 + 数据内容的一种组合通信数据包PACKETMSG；

    typedef struct tag\_TDEFAULTMESSAGE

{

int nRecog;

WORD wIdent;

WORD wParam;

WORD wTag;

WORD wSeries;

} \_TDEFAULTMESSAGE, \*\_LPTDEFAULTMESSAGE;

typedef struct tagPACKETMSG

{

\_TDEFAULTMESSAGE stDefMsg;

CHAR szEncodeData[MAX\_PATH \* 4];

}PACKETMSG, \*LPPACKETMSG;

        其中，nRecog指的是后面的数据包中的数据(结构)的数目,但不是每个消息命令都会用到\_TDEFAULTMESSAGE以上字段；

        WIdent就是消息命令，知道了消息命令和后面的消息内容的个数后，就可以用指定的方式去读取nRecog个预定格式(结构)数据出来。

OnProcPacketNotEncode说明：

   收到GameSrv服务器的相应消息：

     "GOOD"：可以执行动作。 m\_bMotionLock为假。

     "FAIL"：不允许执行动作。人物被拉回移动前位置。

     "LNG"：

     "ULNG"：

     "WID"：

     "UWID"：

     "FIR"：

     "UFIR"：

     "PWR"：

3.     CGameProcess::OnSocketMessageRecieve(char \*pszMsg)函数。处理游戏相关的消息。

   SM\_SENDNOTICE： 服务器提示信息：

SM\_NEWMAP: 用户登录后，服务器发送的初始化地图消息。

SM\_LOGON： 用户登录消息(服务器处理后返回结果)。用户登录成功后，在本地创建游戏对象，并发送消息，请求返回用户物品清单（魔法，等级，物品等）。

SM\_MAPDESCRIPTION: 得到服务器发送的地图的描述信息。

SM\_ABILITY：服务器发送的本玩家金钱，职业信息。

SM\_WINEXP：

SM\_SUBABILITY : 服务器发送的玩家技能(魔法，杀伤力，速度，毒药，中毒恢复，生命恢复，符咒恢复)

SM\_ SM\_SENDMYMAGIC: 用户魔法列表信息。

SM\_MAGIC\_LVEXP: 魔法等级列表。

    SM\_BAGITEMS：用户物品清单 (玩家CM\_QUERYBAGITEMS消息)

SM\_SENDUSEITEMS：用户装备清单

SM\_ADDITEM： 拣东西

SM\_DELITEM： 丢弃物品。

等等。

4.     部分数据未处理，加入m\_xWaitPacketQueue队列中由ProcessPacket处理。

**新登录游戏玩家：在OnSocketMessageRecieve函数中依次收到的消息为：**

1． GameSrv 服务器ProcessLogin线程返回GameGate服务器后返回的：

   AddProcess(this, RM\_LOGON, 0, 0, 0, 0, NULL);加入登录消息。

SM\_NEWMAP, SM\_LOGON, SM\_USERNAME, SM\_MAPDESCRIPTION消息

    AddProcess(this, RM\_ABILITY, 0, 0, 0, 0, NULL); 等级

      SM\_ABILITY

AddProcess(this, RM\_SUBABILITY, 0, 0, 0, 0, NULL);

 SM\_SUBABILITY

AddProcess(this, RM\_DAYCHANGING, 0, 0, 0, 0, NULL); 校时

 SM\_DAYCHANGING

AddProcess(this, RM\_SENDUSEITEMS, 0, 0, 0, 0, NULL); 装备

 SM\_SENDUSEITEMS

AddProcess(this, RM\_SENDMYMAGIC, 0, 0, 0, 0, NULL); 魔法

     SM\_SENDMYMAGIC

**客户端收到消息后相应的处理：**

SM\_NEWMAP 接受地图消息 OnSvrMsgNewMap

     初始化玩家坐标，m\_xMyHero.m\_wPosX = ptdm->wParam;

                     m\_xMyHero.m\_wPosY = ptdm->wTag;

加载地图文件    m\_xMap.LoadMapData(szMapName);

设置场景。 m\_xLightFog.ChangeLightColor(dwFogColor);

SM\_LOGON 返回登录消息 OnSvrMsgLogon

          m\_xMyHero.Create初始化玩家信息（头发，武器，加载图片等），设置玩家

地图m\_xMyHero.SetMapHandler(&m\_xMap)，创建用户魔法。加入m\_xMagicList列表，pxMagic->CreateMagic, m\_xMagicList.AddNode(pxMagic);并向服务器发送CM\_QUERYBAGITEMS消息（用户物品清单，血，气，衣服，兵器等）。

SM\_USERNAME           获取玩家的游戏角色名字。

SM\_MAPDESCRIPTION     地图对应的名字。

SM\_BAGITEMS           用户物品清单 (玩家CM\_QUERYBAGITEMS消息)

SM\_CHARSTATUSCHANGED   通知玩家状态改变消息（攻击力，状态）。

SM\_ABILITY       玩家金钱，职业

SM\_SUBABILITY    玩家技能(魔法，杀伤力，速度，毒药，中毒恢复，生命恢复，符

咒恢复)

SM\_DAYCHANGING  返回游戏状态。（Day, Fog）让客户端随着服务器的时间，加载不同场景。

SM\_SENDUSEITEMS 用户装备清单

SM\_SENDMYMAGIC   用户魔法列表信息。

**总结：**

**客户端连接到GameGate游戏网关服务器，并通过GameSrv服务器验证之后，就会收到GameSrv服务器发来的消息。主要是地图消息，登录消息，玩家的装备，技能，魔法，个人设置等等。GameSrv把地图分成若干块，把该玩家加入其中一块，并加入这一块的用户对象列表中，设置其状态为OS\_MOVINGOBJECT。客户端加载地图，设置场景，设置自己的玩家状态（此时还没有怪物和其它玩家，所以玩家还需要接收其它游戏玩家和怪物的清单列表）。**

6. **接收怪物，商人，其它玩家的消息:**

**ProcessUserHuman:(其它玩家—服务器处理)**

CPlayerObject->SearchViewRange();

CPlayerObject->Operate();

遍历UserInfoList列表，依次调用每个UserInfo的Operate来处理命令队列中的所有操作; pUserInfo->Operate()调用m\_pxPlayerObject->Operate()调用。根据分发消息（RM\_TURN）向客户端发送SM\_TURN消息。GameSrv广播新玩家上线（坐标）的消息。向该新玩家发送玩家信息（等级，装备，魔法，攻击力等）。

玩家，移动对象：

1. 遍历m\_xVisibleObjectList列表，所有(玩家，商人，怪物）发送调用AddProcess

 (RM\_TURN向周围玩家发送消息)。

地图：

2．遍历m\_xVisibleItemList，发送AddProcess(this, RM\_ITEMSHOW消息更新地图。

3．遍历m\_xVisibleEventList，发送AddProcess(this, RM\_SHOWEVENT

**ProcessMonster线程**：**(怪物—服务器处理)**

GameSrv服务器在ProcessMonster线程：创建不同的CMonsterObject对象，并且加入xMonsterObjList列表和pMapCellInfo->m\_xpObjectList列表中，然后再调用CMonsterObject::SearchViewRange()更新视线范围内目标，根据g\_SearchTable计算出搜索坐标，转换为相应的地图单元格，遍历所有可移动生物，加入m\_xVisibleObjectList列表，调用Operate；Operate遍历m\_DelayProcessQ列表，过滤出RM\_DOOPENHEALTH，RM\_STRUCK和RM\_MAGSTRUCK三个事件(恢复生命值，攻击，魔法攻击)，并处理。

**ProcessMerchants线程：(商人--服务器处理)**

       1). 遍历g\_pMerchantInfo结构(根据nNumOfMurchantInfo数量)。得到商人类型相关的地图，创建商人对象，设置不同的编号，坐标，头像及所属地图。在该地图中加入该商人，且在g\_xMerchantObjList商人清单中加入该商人。

2). 遍历g\_xMerchantObjList, SearchViewRange，对每个商人更新视线范围内目标

a). 遍历m\_xVisibleObjectList，设置每个pVisibleObject->nVisibleFlag = 0;设置状态（删除）。

b). 搜索VisibleObjectList列表，(服务器启动时InitializingServer加载 searchTable.tbl)，根据坐标，找到相应的地图单元格。然后遍历pMapCellInfo->m\_xpObjectList列表，判断如果为OS\_MOVINGOBJECT标志，调用UpdateVisibleObject函数，该函数遍历 m\_xVisibleObjectList列表，如果找到该商人对象，则pVisibleObject->nVisibleFlag = 1;否则判断pNewVisibleObject对象，设置nVisibleFlag为2，设置对象为该商人实体，然后加入m\_xVisibleObjectList列表中。

总结：循环列表，找出地图单元格中的所有玩家，把所有玩家(OS\_MOVINGOBJECT)加入到m\_xVisibleObjectList列表中。

c). 遍历m\_xVisibleObjectList列表,(pVisibleObject->nVisibleFlag == 0)则删除该pVisibleObject对象。

d). RunRace调用AddRefMsg 向周围玩家发送SM\_TURN和SM\_HIT

**客户端收到消息后相应的处理：**

1．CGameProcess::OnSocketMessageRecieve加入m\_xWaitPacketQueue队列

**遍历m\_xVisibleObjectList队列中所有移动物体(角色)：**

**RM\_DISAPPEAR**   消失(SM\_DISAPPEAR)  [ProcessDefaultPacket函数](https://www.cnblogs.com/byso/p/7282897.html#ProcessDefaultPacket%E5%87%BD%E6%95%B0)

**RM\_DEATH**       死亡(SM\_NOWDEATH, SM\_DEATH)

           CHero::OnDeath 其它玩家。

           CActor::OnDeath 怪物。

//g\_xGameProc.m\_xMagicList

**RM\_TURN**        移动

[SM\_TURN消息处理](https://www.cnblogs.com/byso/p/7282897.html#SM_TURN%E6%B6%88%E6%81%AF%E5%A4%84%E7%90%86)

**遍历m\_xVisibleItemList队列中所有移动物体(地图)：**

**RM\_ITEMHIDE**    从m\_stMapItemList列表中删除该移动对象

**RM\_ITEMSHOW** 遍历m\_stMapItemList，如果不存在，则创建一个GROUNDITEM结构，并加入m\_stMapItemList列表中。

typedef struct tagGROUNDITEM

{

                     INT             nRecog;

                     SHORT           shTileX;

                     SHORT           shTileY;

                     WORD            wLooks;

                     CHAR            szItemName[40];

}GROUNDITEM, \*LPGROUNDITEM;

**遍历m\_xVisibleEventList队列中所有移动物体(事件)：**

**RM\_HIDEEVENT**

**RM\_SHOWEVENT**

2. 部分数据未处理，加入m\_xWaitPacketQueue队列中由ProcessPacket处理。

CClientSocket::OnSocketMessage的FD\_READ事件中，PacketQ.PushQ把接收到的消息，压入PacketQ队列中。处理PacketQ队列数据是由CGameProcess::Load()时调用OnTimer在CGameProcess::OnTimer中处理的，处理过程为：

OnTimer -> ProcessPacket -> ProcessPacket处理m\_xWaitPacketQueue队列消息（OnSocketMessageRecieve函数中未处理的消息）。

**ProcessPacket 函数处理流程：**

1． 处理本玩家（SM\_NOWDEATH, SM\_DEATH, SM\_CHANGEMAP, SM\_STRUCK）

a.如果接收到消息是SM\_NOWDEATH或SM\_DEATH 则加入m\_xPriorPacketQueue队列。

b. 如果接收到消息是SM\_CHANGEMAP则调用LoadMapChanged,设置场景。

c. SM\_STRUCK 处理受攻击（本玩家，或者其它的玩家，NPC等）。

2． 其它消息：m\_xMyHero.StruckMsgReassign();

                 m\_xMyHero.m\_xPacketQueue.PushQ((BYTE\*)lpPacketMsg);

判断服务器发送来的消息ID是否相同。m\_xMyHero.m\_dwIdentity在登录成功的时

候由服务器发送的用户消息获取的。

if ( lpPacketMsg->stDefMsg.nRecog == m\_xMyHero.m\_dwIdentity )

如果是服务器端游戏玩家自己发送的消息，则处理自己的消息。否则如果是其它玩家(怪物)发送的消息，遍历m\_xActorList列表, 判断该对象是否存在，如果该不存在，则根据stFeature.bGender的类型

\_GENDER\_MAN： 创建一个CHero对象，加入到m\_xActorList列表中。

\_GENDER\_WOMAN：

\_GENDER\_NPC： 创建一个CNPC对象，加入到m\_xActorList列表中。

\_GENDER\_MON： 创建一个CActor对象，加入到m\_xActorList列表中。

然后pxActor->m\_xPacketQueue.PushQ 然后把消息压入该对象的xPacketQueue列表中。

    总结：ProcessPacket处理 CClientSocket类接受的消息(m\_xWaitPacketQueue)，判断是否是服务器发送给自己的消息，处理一些发送给自己的重要消息，其它消息处理则加入m\_xMyHero.m\_xPacketQueue队列中，然后再遍历m\_xActorList队列，判断如果服务器端发来的消息里的玩家(NPC，怪物)，在m\_xActorList队列中找不到，就判断一个加入m\_xActorList列表中，并且把该消息压入pxActor->m\_xPacketQueue交给该NPC去处理该事件。

而xPacketQueue队列的消息分别由该对象的UpdatePacketState处理，如下：

BOOL CActor::UpdatePacketState() ,BOOL CNPC::UpdatePacketState()

 BOOL CHero::UpdatePacketState()。

**ProcessDefaultPacket函数：**

    处理CGameProcess::OnSocketMessageRecieve 中 SM\_CLEAROBJECT消息：

处理（SM\_DISAPPEAR，SM\_CLEAROBJECT）消息。

   遍历m\_xWaitDefaultPacketQueue消息列表

SM\_DISAPPEAR和SM\_CLEAROBJECT：

          遍历m\_xActorList列表，清除pxActor->m\_xPacketQueue队列内所有消息。

m\_xActorList.DeleteCurrentNodeEx();从对列中删除该对象。

CHero\* pxHero = (CHero\*)pxActor; delete((CHero\*)pxHero);销毁该玩家。

**游戏循环处理： CGameProcess::RenderScene(INT nLoopTime)函数：**

主要流程如下：

   wMoveTime += nLoopTime; 判断wMoveTime>100时，bIsMoveTime置为真。

 1．m\_xMyHero.UpdateMotionState(nLoopTime, bIsMoveTime);处理本玩家消息。

     a. UpdatePacketState函数：

           遍历m\_xPriorPacketQueue队列，如果有SM\_NOWDEATH或SM\_DEATH消息，则优先处理。

           处理m\_xPacketQueue队列中消息。

             SM\_STRUCK:

             SM\_RUSH

             SM\_BACKSTEP

             SM\_FEATURECHANGED:

             SM\_OPENHEALTH:

SM\_CLOSEHEALTH:

SM\_CHANGELIGHT:

SM\_USERNAME:

SM\_CHANGENAMECOLOR:

             SM\_CHARSTATUSCHANGE:

SM\_MAGICFIRE:

SM\_HEALTHSPELLCHANGED:

 2．CheckMappedData函数：遍历m\_xActorList列表分别调用

         CActor::UpdateMotionState(INT nLoopTime, BOOL bIsMoveTime)

CNPC::UpdateMotionState(INT nLoopTime, BOOL bIsMoveTime)

CMyHero::UpdateMotionState(INT nLoopTime, BOOL bIsMoveTime)

     处理自己消息。

CHero::UpdatePacketState()

case SM\_SITDOWN:

         case SM\_BUTCH:

         case SM\_FEATURECHANGED:

         case SM\_CHARSTATUSCHANGE:

         case SM\_OPENHEALTH:

         case SM\_CLOSEHEALTH:

         case SM\_CHANGELIGHT:

         case SM\_USERNAME:

         case SM\_CHANGENAMECOLOR:

         case SM\_HEALTHSPELLCHANGED:

         case SM\_RUSH:

         case SM\_BACKSTEP:

         case SM\_NOWDEATH:

         case SM\_DEATH:

         case SM\_WALK:

         case SM\_RUN:

         case SM\_TURN:

         case SM\_STRUCK:

         case SM\_HIT:

         case SM\_FIREHIT:

         case SM\_LONGHIT:

         case SM\_POWERHIT:

         case SM\_WIDEHIT:

         case SM\_MAGICFIRE:

     case SM\_SPELL:

 CNPC::UpdatePacketState()

     case SM\_OPENHEALTH:

      case SM\_CLOSEHEALTH:

      case SM\_CHANGELIGHT:

      case SM\_USERNAME:

      case SM\_CHANGENAMECOLOR:

      case SM\_HEALTHSPELLCHANGED:

      case SM\_TURN:

      case SM\_HIT:

    CActor::UpdatePacketState()

         case SM\_DEATH:     SetMotionFrame(\_MT\_MON\_DIE, bDir);

         case SM\_WALK:      SetMotionFrame(\_MT\_MON\_WALK, bDir);

case SM\_TURN:      SetMotionFrame(\_MT\_MON\_STAND, bDir);

case SM\_DIGUP:     SetMotionFrame(\_MT\_MON\_APPEAR, bDir);

case SM\_DIGDOWN:   SetMotionFrame(\_MT\_MON\_APPEAR, bDir);

         case SM\_FEATURECHANGED:

        case SM\_OPENHEALTH:

         case SM\_CLOSEHEALTH:

         case SM\_CHANGELIGHT:

         case SM\_CHANGENAMECOLOR:

         case SM\_USERNAME:

         case SM\_HEALTHSPELLCHANGED:

         case SM\_BACKSTEP:      SetMotionFrame(\_MT\_MON\_WALK, bDir);

         case SM\_STRUCK:            SetMotionFrame(\_MT\_MON\_HITTED, m\_bCurrDir);

         case SM\_HIT:           SetMotionFrame(\_MT\_MON\_ATTACK\_A, bDir);

         case SM\_FLYAXE:

         case SM\_LIGHTING:

         case SM\_SKELETON:

     收到多个NPC，玩家发送的SM\_TURN消息：由下面对象调用处理：

CHero::OnTurn

CNPC::OnTurn

CActor::OnTurn

根据服务器发送的消息，(创建一个虚拟玩家NPC，怪物，在客户端)，根据参数，初始化该对象设置(方向，坐标，名字，等级等)。在后面的处理中绘制该对象到UI界面中([移动对象的UI界面处理。](https://www.cnblogs.com/byso/p/7282897.html#%E7%A7%BB%E5%8A%A8%E5%AF%B9%E8%B1%A1%E7%9A%84UI%E7%95%8C%E9%9D%A2%E5%A4%84%E7%90%86))

        SetMotionFrame(\_MT\_MON\_STAND, bDir); m\_bCurrMtn := \_MT\_MON\_STAND

        m\_dwFstFrame , m\_dwEndFrame , m\_wDelay 第一帧，最后一帧，延迟时间。

   3.  AutoTargeting 自动搜索目标(NPC,怪物，玩家等)

   4． RenderObject补偿对象时间

   5.  RenderMapTileGrid

       m\_xMagicList，处理玩家魔法后，UI界面的处理。

6.       m\_xSnow, m\_xRain, m\_xFlyingTail, m\_xSmoke, m\_xLightFog设置场景UI界面处理。

  7. m\_xMyHero.ShowMessage(nLoopTime); 显示用户(UI处理)

 m\_xMyHero.DrawHPBar(); 显示用户HP值。

  遍历m\_xActorList，处理所有NPC的UI界面重绘

   pxHero->ShowMessage(nLoopTime);

 pxHero->DrawHPBar();

 8. DropItemShow下拉显示。

9. 判断m\_pxMouseTargetActor(玩家查看其它玩家，NPC，怪物时)

   g\_xClientSocket.SendQueryName向服务器提交查询信息。

m\_pxMouseOldTargetActor = m\_pxMouseTargetActor; 保存该对象

     m\_pxMouseTargetActor->DrawName(); 重绘对象名字(UI界面显示)

**下面分析一下用户登录之后的流程：**

**从前面的分析中可以看到，该用户玩家登录成功之后，得到了服务器发送来的各种消息。处理也比较复杂，同时有一定的优先级处理。并且根据用户登录后的XY坐标，向用户发送来了服务器XY坐标为中心附近单元格中的所有玩家(NPC,怪物)的SM\_TURN消息。**

**客户端根据数据包的标志，创建这些NPC，设置属性，并且把它们加入m\_xActorList对列中。最后在UI界面上绘制这些对象。**

**现在假设玩家开始操作游戏：**

传奇的客户端源代码工程WindHorn

一、CWHApp派生CWHWindow和CWHDXGraphicWindow。

二、CWHDefProcess派生出CloginProcess、CcharacterProcess、CgameProcess

客户端WinMain调用CWHDXGraphicWindow g\_xMainWnd;创建一个窗口。

客户端CWHDXGraphicWindow在自己的Create函数中调用了CWHWindow的Create来创建窗口，然后再调用自己的CreateDXG()来初始化DirectX。

**消息循环：**

因此，当客户端鼠标单击的时候，先调用CWHWindow窗口的回调函数WndProc，即：    g\_pWHApp->MainWndProc g\_pWHApp定义为：static CWHApp\* g\_pWHApp = NULL；在CWHApp

构造函数中赋值为：g\_pWHApp = this;

g\_pWHApp->MainWndProc便调用了CWHApp::MainWndProc，这是一个虚函数，实际上则是调用它的派生类CWHDXGraphicWindow::MainWndProc。  
   if ( m\_pxDefProcess )   
       return m\_pxDefProcess->DefMainWndProc(hWnd, uMsg, wParam, lParam);   
根据g\_xMainWnd.m\_pxDefProcess和全局变量g\_bProcState标记当前的处理状态。调用

     CLoginProcess->DefMainWndProc

CCharacterProcess->DefMainWndProc

CGameProcess->DefMainWndProc

**当用户进行游戏之后，点击鼠标左键，来处理玩家走动的动作：**

**客户端执行流程：(玩家走动)**

CGameProcess::OnLButtonDown(WPARAM wParam, LPARAM lParam)函数：该函数的处理流程：

 1． g\_xClientSocket.SendNoticeOK();如果点中CnoticeBox则m\_xNotice.OnButtonDown

     if m\_xMsgBtn.OnLButtonDown则调用g\_xClientSocket.SendNoticeOK()方法，发送还CM\_LOGINNOTICEOK消息。

2．m\_pxSavedTargetActor = NULL;设置为空。CInterface::OnLButtonDown函数会判断

   鼠标点击的位置(CmirMsgBox, CscrlBar,CgameBtn，GetWindowInMousePos)

     a. g\_xClientSocket.SendItemIndex(CM\_DROPITEM 丢弃物品)

[游戏服务器执行流程](https://www.cnblogs.com/byso/p/7282897.html#%E6%B8%B8%E6%88%8F%E6%9C%8D%E5%8A%A1%E5%99%A8%E6%89%A7%E8%A1%8C%E6%B5%81%E7%A8%8B)m\_pxPlayerObject->Operate()调用

          m\_pUserInfo->UserDropGenItem

         m\_pUserInfo->UserDropItem      删除普通物品。

          SM\_DROPITEM\_SUCCESS             返回删除成功命令

SM\_DROPITEM\_FAIL                返回删除失败命令

     b. 遍历m\_stMapItemList列表(存储玩家，怪物，NPC)， g\_xClientSocket.SendPickUp 发送CM\_PICKUP命令。

         游戏服务器：m\_pxPlayerObject->Operate()调用 PickUp(捡东西)消息处理：

m\_pMap->GetItem(m\_nCurrX, m\_nCurrY) 返回地图里的物体(草药,物品，金子等)

**1．memcmp(pMapItem->szName, g\_szGoldName 如果是黄金：**

                m\_pMap->RemoveObject从地图中移走该的品。

if (m\_pUserInfo->IncGold(pMapItem->nCount))增加用户的金钱(向周转玩家发送RM\_ITEMHIDE 消息，隐藏该物体，GoldChanged()，改变玩家的金钱。否则，把黄金返回地图中。

**2．m\_pUserInfo->IsEnoughBag()**

                如果玩家的还可以随身带装备(空间)。m\_pMap->RemoveObject从地图中移走该的品。UpdateItemToDB，更新用户信息到数据库。(向周转玩家发送RM\_ITEMHIDE 消息，隐藏该物体，SendAddItem(lptItemRcd)向本玩家发送捡到东西的消息。m\_pUserInfo->m\_lpTItemRcd.AddNewNode并把该物品加入自己的列表中。

     c. if m\_pxMouseTargetActor g\_xClientSocket.SendNPCClick发送CM\_CLICKNPC命令。

客户端RenderScene调用m\_pxMouseTargetActor = NULL;

CheckMappedData(nLoopTime, bIsMoveTime)处理，如果鼠标在某个移动对象的区域内就会设置 m\_pxMouseTargetActor为该对象。

            如果是NPC：

if ( m\_pxMouseTargetActor->m\_stFeature.bGender == \_GENDER\_NPC )

        g\_xClientSocket.SendNPCClick(m\_pxMouseTargetActor->m\_dwIdentity);

            CM\_CLICKNPC消息：

            否则：

m\_xMyHero.OnLButtonDown

     d. 否则m\_xMyHero.OnLButtonDown

先判断m\_xPacketQueue是否有数据，有则先处理。返回。

判断m\_pxMap->GetNextTileCanMove 根据坐标，判断地图上该点属性是否可以移动到该位置：

        可移动时：

           人：SetMotionState(\_MT\_WALK

           骑马：SetMotionState(\_MT\_HORSEWALK

        不可移动时：

           人：SetMotionState(\_MT\_STAND, bDir);

          骑马：SetMotionState(\_MT\_HORSESTAND, bDir);

        SetMotionState函数：

            判断循环遍历目标点的周围八个坐标，如果发现是一扇门，则向服务器发送打开这扇门的命令。g\_xClientSocket.SendOpenDoor,否则则发送CM\_WALK命令到服务器。

            m\_bMotionLock = m\_bInputLock = TRUE; 设置游戏状态

            m\_wOldPosX = m\_wPosX;                保存玩家X点

            m\_wOldPosY = m\_wPosY;               保存玩家Y点

            m\_bOldDir = m\_bCurrDir;             保存玩家方向

然后调用SetMotionFrame设置m\_bCurrMtn = \_MT\_WALK，方向等游戏状态。

        设置m\_bMoveSpeed = \_SPEED\_WALK（移动速度1）。m\_pxMap->ScrollMap设置地图的偏移位置（m\_shViewOffsetX, m\_shViewOffsetY）。然后滚动地图，重绘玩家由CGameProcess::RenderScene CGameProcess::RenderObject->DrawActor重绘。

**游戏服务器执行流程：(玩家走动)**

    GameSrv服务器ProcessUserHuman线程处理玩家消息：

遍历UserInfoList列表，依次调用每个UserInfo的Operate来处理命令队列中的所有操作; pUserInfo->Operate()调用m\_pxPlayerObject->Operate()调用。

判断玩家if (!m\_fIsDead)，如果已死，则发送\_MSG\_FAIL消息。我们在前面看到过，该消息是被优先处理的。否则则调用WalkTo，并发送\_MSG\_GOOD消息给客户端。

WalkTo函数的流程：

1） WalkNextPos 根据随机值产生，八个方向的坐标位置。

2） WalkXY怪物走动到一个坐标值中。

CheckDoorEvent根据pMapCellInfo->m\_sLightNEvent返回四种状态。

a) 要移动的位置是一扇门 \_DOOR\_OPEN

b) 不是一扇门 \_DOOR\_NOT

c) 是一扇门不可以打开返回 \_DOOR\_MAPMOVE\_BACK或\_DOOR\_MAPMOVE\_FRONT玩家前/后移动

3） 如果\_DOOR\_OPEN则发送SM\_DOOROPEN消息给周围玩家。

4） m\_pMap->CanMove如果可以移动，则MoveToMovingObject从当前点移动到另一点。并发送AddRefMsg(RM\_WALK)给周围玩家。

    AddRefMsg函数，我们在后面的服务器代码里分析过：它会根据X，Y坐标，在以自己坐标为中心周围26\*26区域里面，按地图单元格的划分，遍历所有单元格，再遍历所有单元格内的玩家列表，广播发送RM\_WALK消息。

**客户端执行流程：(反馈服务器端本玩家走动)**

1. 服务器如果发送\_MSG\_FAIL 由客户端CGameProcess::OnProcPacketNotEncode处理。

     m\_xMyHero.SetOldPosition();

人：  SetMotionFrame(\_MT\_STAND

           AdjustMyPostion(); 重绘地图

           m\_bMotionLock = m\_bInputLock = FALSE;

     骑马：SetMotionFrame(\_MT\_HORSESTAND

            AdjustMyPostion(); 重绘地图

            m\_bMotionLock = m\_bInputLock = FALSE;

   2. 服务器如果发送\_MSG\_GOOD, 由客户端CGameProcess::OnProcPacketNotEncode处理。m\_xMyHero.m\_bMotionLock = FALSE;

**其它客户端执行流程：(反馈服务器端其它玩家)**

 1．其它玩家：

     人： SetMotionFrame(\_MT\_WALK, bDir);

     骑马：SetMotionFrame(\_MT\_HORSEWALK, bDir);

     m\_bMoveSpeed = \_SPEED\_WALK;

     SetMoving(); 设置m\_shShiftPixelX， m\_shShiftPixelY坐标。

 2．NPC，怪物：

SetMotionFrame(\_MT\_MON\_WALK, bDir);

     m\_bMoveSpeed = \_SPEED\_WALK;

     SetMoving(); 设置m\_shShiftPixelX， m\_shShiftPixelY坐标。

CGameProcess::RenderObject->DrawActor(m\_shShiftPixelX， m\_shShiftPixelY)重绘发消息的玩家，NPC怪物位置。

     最近对高性能的服务器比较感兴趣，读过了DELPHI的Socker源码WebService及RemObject之后，高性能的服务器感兴趣。

你可能需要的以下知识才能更好的读懂一个商业源码：

1).SOCKET的I/O模型熟悉掌握。

2).面向对象技术的熟悉掌握。

3).Socket的API掌握。

4).多线程技术等。

5).一门熟悉的开发工具掌握,和多种语言的源码阅读能力。

我下的源码 LegendOfMir2\_Server：共包含AdminCmd, DBSrv, GameGate, GameSvr,LoginGate, LoginSvr, SelGate七个工程文件。传奇的客户端源代码有两个工程，WindHorn和Mir2Ex。

我分析的, 主要是VC SQL版本的, DELPHI翎风源码不做分析,  另外下载了乐都WIL编辑器和乐都MPA地图编辑器这些工具.

**DirectX类库分析(WindHorn)：**

1.     RegHandler.cpp 注册表访问(读写)。

2.     CWHApp派生CWHWindow，CWHWindow完成窗口的注册和创建。CWHWindow派生出CWHDXGraphicWindow,CWHDXGraphicWindow调用CWHWindow完成创建窗口功能，然后再调用CreateDXG()来初始化DirectX。

3.     WHDefProcess.cpp在构造函数中获得CWHDXGraphicWindow句柄。

Clear函数中调用在后台缓存上进行绘图操作，换页至屏幕。

ShowStatus函数,显示状态信息。

DefMainWndProc函数，调用CWHDXGraphicWindow->MainWndProcDXG消息处理。

4.     WHImage.cpp图象处理。加载位图，位图转换。优化处理。

5.     WHSurface.cpp 主页面处理。

6.     WHWilTexture.cpp 材质渲染。

WILTextureContainer： WIL容器类。m\_pNext指向下一个WILTextureContainer，单链表。

7.     WHWilImage.cpp 从Data目录中加载Wix文件（内存映射）。

8.     WHDXGraphic.cpp 处理DirectX效果。

文件类型格式探讨：

Wix文件：索引文件，根据索引查找到相应数据地址(数据文件)。

// WIX 文件头格式

typedef struct tagWIXFILEIMAGEINFO

{

    CHAR    szTmp[40];     // 库文件标题 'WEMADE Entertainment inc.' WIL文件头

    INT     nIndexCount;   // 图片数量

    INT\*    pnPosition;    // 位置

}WIXIMAGEINFO, \*LPWIXIMAGEINFO;

我们下载一个Hedit编辑器打开一个Wil文件，分析一下。我们发现Wix文件中，0x23地址(含该地址)以前的内容是都相同的，即为：#INDX v1.0-WEMADE Entertainment inc.

Ofs44 0x2C的地方：存放着0B 00 00 00，高低位转换后为：0xB转换十进制数为11(图片数量)Ofs48 0x30的地方：存放着38 04 00 00，高低位转换后为：0x438 = 1080, 这个就是图象数据的开始位置。

我们用Wil编辑打开对应的Wil文件，发现，果然有11张图片。另外我们发现，在Ofs = 44 -47之间的数据总是38 04 00 00，终于明白，所有的图片起始位置是相同的。

Wil文件: 数据文件。

前面我们说了图象数据的开始位置为0x438 = 1080, 1080中有文件开头的44字节都是相同的。所以，就是说有另外的1036字节是另有用途。1036中有1024是一个256色的调色板。

我们看到图片位置数据为： 20 03 58 02, 转化为十六进制： 0x320, 0x258 刚好就是800\*600大小的图片。07 00 D4 FF。图片起始位置为：

Ofs 1088: 0x440 图片大小为480000

起始位置：0x440 1088   终止位置：0x7573F 481087 为了验证数据是否正确，我们通过Wil工具，把第一幅图片导出来，然后用Hedit编辑器打开，经过对比，我们发现，数据一致。大小一致。

    第二张BMP图片(图片起始位置：0x436 10078) ： F0 01 69 01 , 07 00 D4 FF

刚好大小。第二张Wil起始位置：Ofs:481096  0x75748

知道了图片格式，我们可以写一个抓图片格式的程序了。

**客户端：**

传奇的客户端源代码有两个工程，WindHorn和Mir2Ex。

先剖析一下WindHorn工程。

1．CWHApp、CWHWindow和CWHDXGraphicWindow。Window程序窗口的创建。   
         CWHApp派生CWHWindow，CWHWindow又派生CWHDXGraphicWindow。CWHWindow类

中完成窗口的注册和创建。CWHDXGraphicWindow调用CWHWindow完成创建窗口功能，然后再调用CreateDXG()来初始化DirectX。

2．CWHDefProcess派生出CloginProcess、CcharacterProcess、CgameProcess三个类。   
   这三个类是客户端处理的核心类。

3. 全局变量:  
   CWHDXGraphicWindow    g\_xMainWnd;  主窗口类。  
   CLoginProcess         g\_xLoginProc; 登录处理。  
   CCharacterProcess     g\_xChrSelProc; 角色选择处理。  
   CgameProcess       g\_xGameProc; 游戏逻辑处理。

4．代码分析：

1.首先从LoginGate.cpp WinMain分析：

 g\_xMainWnd定义为CWHDXGraphicWindow调用CWHWindow完成创建窗口功能，然后

调用DirectDrawEnumerateEx枚举显示设备，(执行回调函数DXGDriverEnumCallbackEx) 再调用CreateDXG()来初始化DirectX(创建DirectDraw对象, 取得独占和全屏模式, 设置显示模式等)。

    g\_xSound.InitMirSound创建CSound对象。

    g\_xSpriteInfo.SetInfo();

     初始化声音，加载Socket库之后，进行CWHDefProcess\*指针赋值(事件绑定)。g\_bProcState变量反应了当前游戏的状态(登录，角色选择，游戏逻辑处理)。调用Load初始化一些操作(登录，角色选择，游戏逻辑处理)。进行消息循环。

    case \_LOGIN\_PROC:

        g\_xLoginProc.RenderScene(dwDelay);

    case \_CHAR\_SEL\_PROC:

        g\_xChrSelProc.RenderScene(dwDelay);

    case \_GAME\_PROC:

g\_xGameProc.RenderScene(dwDelay);

    根据g\_bProcState变量标志，选择显示相应的画面。

 2．接收处理网络消息和接收处理窗口消息。

     在不同的状态下(登录，角色选择，游戏逻辑处理)，接收到的消息(网络，窗口消息)会分派到不同的函数中处理的。这里是用虚函数处理(调用子类方法，由实际的父类完成相应的处理)。

OnMessageReceive主要处理网络消息。DefMainWndProc则处理窗体消息(按键，重绘等)，创建窗体类为CWHDXGraphicWindow，回调函数为：

MainWndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

if ( m\_pxDefProcess )

m\_pxDefProcess->DefMainWndProc(hWnd, uMsg, wParam, lParam);

else

return MainWndProcDXG(hWnd, uMsg, wParam, lParam);

m\_pxDefProcess->DefMainWndProc调用父类的实际处理。

在WM\_PAINT事件里： g\_xClientSocket.ConnectToServer连接登陆服务器。

Overlapped I/O模型深入分析

简述：

     Overlapped I/O也称Asynchronous  I/O，异步I/O模型。异步I/O和同步I/O不同，同步I/O时，程序被挂起，一直到I/O处理完，程序才能获得控制。异步I/O，调用一个函数告诉OS，进行I/O操作，不等I/O结束就立即返回，继续程序执行，操作系统完成I/O之后，通知消息给你。Overlapped I/O只是一种模型，它可以由内核对象(hand)，事件内核对象(hEvent), 异步过程调用(apcs) 和完成端口(I/O completion)实现。

Overlapped I/O的设计的目的：

     取代多线程功能，（多线程存在同步机制，错误处理，在成千上万个线程I/O中，线程上下文切换是十分消耗CPU资源的）。

     Overlapped I/O模型是OS为你传递数据，完成上下文切换，在处理完之后通知你。由程序中的处理，变为OS的处理。内部也是用线程处理的。

Overlapped数据结构：

typedef struct \_OVERLAPPED {

    DWORD   Internal;      通常被保留，当GetOverlappedResult()传回False并且GatLastError()并非传回ERROR\_IO\_PENDINO时，该状态置为系统定的状态。

DWORD   InternalHigh;  通常被保留，当GetOverlappedResult()传回False时，为

                        被传输数据的长度。

DWORD   Offset;        指定文件的位置,从该位置传送数据,文件位置是相对文件开始

处的字节偏移量。调用 ReadFile或WriteFile函数之前调用进

程设置这个成员，读写命名管道及通信设备时调用进程忽略这

个成员;

DWORD   OffsetHigh;    指定开始传送数据的字节偏移量的高位字,读写命名管道及通

信设备时调用进程忽略这个成员;

HANDLE hEvent;        标识事件,数据传送完成时把它设为信号状态,调用ReadFile

                          WriteFile   ConnectNamedPipe   TransactNamedPipe函数

前,调用进程设置这个成员. 相关函数

CreateEvent  ResetEvent   GetOverlappedResult

WaitForSingleObject   CWinThread   GetLastError

} OVERLAPPED, \*LPOVERLAPPED;

二个重要功能：

1． 标识每个正在overlapped 的操作。

2． 程序和系统之间提供了共享区域。参数可以在区域内双向传递。

OVERLAPPED和数据缓冲区释放问题:

在请求时，不能释放，只有在I/O请求完成之后，才可以释放。如果发出多个overlapped请求，每个overlapped读写操作，都必须包含文件位置（socket），另外，如果有多个磁盘，I/O执行次序无法保证。（每个overlapped都是独立的请求操作）。

**内核对象(hand)实现：**

例子：用overlapped模型读一个磁盘文件内容。

   1．把设备句柄看作同步对象，ReadFile将设备句柄设为无信号。ReadFile 异步I/O字节位置必须在OVERLAPPED结构中指定。

   2．完成I/O，设置信息状态。为有信号。

   3．WaitForSingleObject或WaitForMultipleObject判断

或者异步设备调用GetOverLappedResult函数。

int main()

{

    BOOL rc;

    HANDLE hFile;

    DWORD numread;

    OVERLAPPED overlap;

    char buf[READ\_SIZE];

    char szPath[MAX\_PATH];

    CheckOsVersion();

    GetWindowsDirectory(szPath, sizeof(szPath));

    strcat(szPath, "\\WINHLP32.EXE");

    hFile = CreateFile( szPath,

                    GENERIC\_READ,

                    FILE\_SHARE\_READ|FILE\_SHARE\_WRITE,

                    NULL,

                    OPEN\_EXISTING,

                    FILE\_FLAG\_OVERLAPPED,

                    NULL

                );

    if (hFile == INVALID\_HANDLE\_VALUE)

    {

        printf("Could not open %s\n", szPath);

        return -1;

    }

    memset(&overlap, 0, sizeof(overlap));

    overlap.Offset = 1500;

    rc = ReadFile(

                hFile,

                buf,

                READ\_SIZE,

                &numread,

                &overlap

            );

    printf("Issued read request\n");

    if (rc)

    {

        printf("Request was returned immediately\n");

    }

    else

    {

        if (GetLastError() == ERROR\_IO\_PENDING)

        {

            printf("Request queued, waiting...\n");

            WaitForSingleObject(hFile, INFINITE);

            printf("Request completed.\n");

            rc = GetOverlappedResult(

                                    hFile,

                                    &overlap,

                                    &numread,

                                    FALSE

                                );

            printf("Result was %d\n", rc);

        }

        else

        {

            printf("Error reading file\n");

        }

    }

    CloseHandle(hFile);

    return EXIT\_SUCCESS;

}

**事件内核对象(hEvent)：**

内核对象(hand)实现的问题：

    不能区分那一个overlapped操作，对同一个文件handle，系统有多个异步操作时(一边读文件头，一边写文件尾, 有一个完成，就会有信号，不能区分是那种操作。)，为每个进行中的overlapped调用GetOverlappedResult是不好的作法。

事件内核对象(hEvent)实现方案：

Overlapped成员hEven标识事件内核对象。CreateEvent,为每个请求创建一个事件，初始化每个请求的hEvent成员(对同一文件多个读写请求，每个操作绑定一个event对象)。调用WaitForMultipleObject来等等其中一个（或全部）完成。

    另外Event对象必须是手动重置。使用自动重置（在等待event之前设置，WaitForSingleObject()和 WaitForMultipleObjects()函数永不返回）。

自动重置事件

WaitForSingleObject()和 WaitForMultipleObjects()会等待事件到信号状态，随后又自动将其重置为非信号状态，这样保证了等待此事件的线程中只有一个会被唤醒。

手动重置事件

需要用户调用ResetEvent()才会重置事件。可能有若干个线程在等待同一事件，这样当事件变为信号状态时，所有等待线程都可以运行了。 SetEvent()函数用来把事件对象设置成信号状态，ResetEvent()把事件对象重置成非信号状态，两者均需事件对象句柄作参数。

相关例子如下：

int main()

{

    int i;

    BOOL rc;

    char szPath[MAX\_PATH];

    CheckOsVersion();

    GetWindowsDirectory(szPath, sizeof(szPath));

    strcat(szPath, "\\WINHLP32.EXE");

    ghFile = CreateFile( szPath,

                    GENERIC\_READ,

                    FILE\_SHARE\_READ|FILE\_SHARE\_WRITE,

                    NULL,

                    OPEN\_EXISTING,

                    FILE\_FLAG\_OVERLAPPED,

                    NULL

                );

    if (ghFile == INVALID\_HANDLE\_VALUE)

    {

        printf("Could not open %s\n", szPath);

        return -1;

    }

    for (i=0; i<MAX\_REQUESTS; i++)

    {

        QueueRequest(i, i\*16384, READ\_SIZE);

    }

    printf("QUEUED!!\n");

    MTVERIFY( WaitForMultipleObjects(

               MAX\_REQUESTS, ghEvents, TRUE, INFINITE

        ) != WAIT\_FAILED );

    for (i=0; i<MAX\_REQUESTS; i++)

    {

        DWORD dwNumread;

        rc = GetOverlappedResult(

                                ghFile,

                                &gOverlapped[i],

                                &dwNumread,

                                FALSE

                            );

        printf("Read #%d returned %d. %d bytes were read.\n",

                    i, rc, dwNumread);

        CloseHandle(gOverlapped[i].hEvent);

    }

    CloseHandle(ghFile);

    return EXIT\_SUCCESS;

}

int QueueRequest(int nIndex, DWORD dwLocation, DWORD dwAmount)

{

    int i;

    BOOL rc;

    DWORD dwNumread;

    DWORD err;

    MTVERIFY(

        ghEvents[nIndex] = CreateEvent(

                     NULL,    // No security

                     TRUE,    // Manual reset - extremely important!

                     FALSE,   // Initially set Event to non-signaled state

                     NULL     // No name

                    )

    );

    gOverlapped[nIndex].hEvent = ghEvents[nIndex];

    gOverlapped[nIndex].Offset = dwLocation;

    for (i=0; i<MAX\_TRY\_COUNT; i++)

    {

        rc = ReadFile(

            ghFile,

            gBuffers[nIndex],

            dwAmount,

            &dwNumread,

            &gOverlapped[nIndex]

        );

        if (rc)

        {

            printf("Read #%d completed immediately.\n", nIndex);

            return TRUE;

        }

        err = GetLastError();

        if (err == ERROR\_IO\_PENDING)

        {

            // asynchronous i/o is still in progress

            printf("Read #%d queued for overlapped I/O.\n", nIndex);

            return TRUE;

        }

        if ( err == ERROR\_INVALID\_USER\_BUFFER ||

             err == ERROR\_NOT\_ENOUGH\_QUOTA ||

             err == ERROR\_NOT\_ENOUGH\_MEMORY )

        {

            Sleep(50); // Wait around and try later

            continue;

        }

        break;

    }

    printf("ReadFile failed.\n");

    return -1;

}

**异步过程调用(apcs)：**

事件内核对象(hEvent)的问题：

    事件内核对象在使用WaitForMultipleObjects时，只能等待64个对象。需要另建两个数据组，并gOverlapped[nIndex].hEvent = ghEvents[nIndex]绑定起来。

异步过程调用(apcs)实现方案：

    异步过程调用，callback回调函数，在一个Overlapped I/O完成之后，系统调用该回调函数。OS在有信号状态下(设备句柄)，才会调用回调函数（可能有很多APCS等待处理了），传给它完成I/O请求的错误码，传输字节数和Overlapped结构的地址。

    五个函数可以设置信号状态：

1． SleepEx

2． WaitForSingleObjectEx

3． WaitForMultipleObjectEx

4． SingalObjectAndWait

5． MsgWaitForMultipleObjectsEx

Main函数调用WaitForSingleObjectEx, APCS被处理，调用回调函数

FileIOCompletionRoutine

VOID WINAPI FileIOCompletionRoutine(

    DWORD dwErrorCode, // completion code

    DWORD dwNumberOfBytesTransfered,    // number of bytes transferred

    LPOVERLAPPED lpOverlapped   // pointer to structure with I/O information

   )

{

    int nIndex = (int)(lpOverlapped->hEvent);

    printf("Read #%d returned %d. %d bytes were read.\n",

        nIndex,

        dwErrorCode,

        dwNumberOfBytesTransfered);

    if (++nCompletionCount == MAX\_REQUESTS)

        SetEvent(ghEvent); // Cause the wait to terminate

}

int main()

{

    int i;

    char szPath[MAX\_PATH];

    CheckOsVersion();

    MTVERIFY(

        ghEvent = CreateEvent(

                     NULL,    // No security

                     TRUE,    // Manual reset - extremely important!

                     FALSE,   // Initially set Event to non-signaled state

                     NULL     // No name

                    )

    );

    GetWindowsDirectory(szPath, sizeof(szPath));

    strcat(szPath, "\\WINHLP32.EXE");

    ghFile = CreateFile( szPath,

                    GENERIC\_READ,

                    FILE\_SHARE\_READ|FILE\_SHARE\_WRITE,

                    NULL,

                    OPEN\_EXISTING,

                    FILE\_FLAG\_OVERLAPPED,

                    NULL

                );

    if (ghFile == INVALID\_HANDLE\_VALUE)

    {

        printf("Could not open %s\n", szPath);

        return -1;

    }

    for (i=0; i<MAX\_REQUESTS; i++)

{

        QueueRequest(i, i\*16384, READ\_SIZE);

    }

    printf("QUEUED!!\n");

    for (;;)

    {

        DWORD rc;

        rc = WaitForSingleObjectEx(ghEvent, INFINITE, TRUE );

        if (rc == WAIT\_OBJECT\_0)

            break;

        MTVERIFY(rc == WAIT\_IO\_COMPLETION);

    }

    CloseHandle(ghFile);

    return EXIT\_SUCCESS;

}

int QueueRequest(int nIndex, DWORD dwLocation, DWORD dwAmount)

{

    int i;

    BOOL rc;

    DWORD err;

    gOverlapped[nIndex].hEvent = (HANDLE)nIndex;

    gOverlapped[nIndex].Offset = dwLocation;

    for (i=0; i<MAX\_TRY\_COUNT; i++)

    {

        rc = ReadFileEx(

            ghFile,

            gBuffers[nIndex],

            dwAmount,

            &gOverlapped[nIndex],

            FileIOCompletionRoutine

        );

        if (rc)

        {

            printf("Read #%d queued for overlapped I/O.\n", nIndex);

            return TRUE;

        }

        err = GetLastError();

        if ( err == ERROR\_INVALID\_USER\_BUFFER ||

             err == ERROR\_NOT\_ENOUGH\_QUOTA ||

             err == ERROR\_NOT\_ENOUGH\_MEMORY )

        {

            Sleep(50); // Wait around and try later

            continue;

        }

        break;

    }

    printf("ReadFileEx failed.\n");

    return -1;

}

**完成端口(I/O completion)：**

异步过程调用(apcs)问题：

    只有发overlapped请求的线程才可以提供callback函数（需要一个特定的线程为一个特定的I/O请求服务）。

完成端口(I/O completion)的优点：

    不会限制handle个数，可处理成千上万个连接。I/O completion port允许一个线程将一个请求暂时保存下来，由另一个线程为它做实际服务。

并发模型与线程池：

    在典型的并发模型中，服务器为每一个客户端创建一个线程，如果很多客户同时请求，则这些线程都是运行的，那么CPU就要一个个切换，CPU花费了更多的时间在线程切换，线程确没得到很多CPU时间。到底应该创建多少个线程比较合适呢，微软件帮助文档上讲应该是2\*CPU个。但理想条件下最好线程不要切换，而又能象线程池一样，重复利用。I/O完成端口就是使用了线程池。

**理解与使用：**

**第一步：**

在我们使用完成端口之前，要调用CreateIoCompletionPort函数先创建完成端口对象。

定义如下：  
HANDLE CreateIoCompletionPort(  
                                HANDLE FileHandle,  
                                HANDLE ExistingCompletionPort,  
                                DWORD CompletionKey,  
                                DWORD NumberOfConcurrentThreads  
);  
FileHandle：

文件或设备的handle, 如果值为INVALID\_HANDLE\_VALUE则产生一个没有和任何文件handle有关系的port.( 可以用来和完成端口联系的各种句柄，文件，套接字)

ExistingCompletionPort:

NULL时生成一个新port, 否则handle会加到此port上。

CompletionKey：

用户自定义数值，被交给服务的线程。GetQueuedCompletionStatus函数时我们可以完全得到我们在此联系函数中的完成键（申请的内存块）。在GetQueuedCompletionStatus

中可以完封不动的得到这个内存块，并且使用它。

NumberOfConcurrentThreads：

参数NumberOfConcurrentThreads用来指定在一个完成端口上可以并发的线程数量。理想的情况是，一个处理器上只运行一个线程，这样可以避免线程上下文切换的开销。如果这个参数的值为0，那就是告诉系统线程数与处理器数相同。我们可以用下面的代码来创建I/O完成端口。

隐藏在之创建完成端口的秘密：

1． 创建一个完成端口

CreateIoCompletionPort(INVALID\_HANDLE\_VALUE, 0, 0, dwNumberOfConcurrentThreads);

2． 设备列表，完成端口把它同一个或多个设备相关联。

CreateIoCompletionPort(hDevice, hCompPort, dwCompKey, 0) ;

**第二步：**

根据处理器个数，创建cpu\*2个工作线程：

CreateThread(NULL, 0, ServerWorkerThread, CompletionPort,0, &ThreadID))

与此同时，服务器调用WSASocket，bind, listen, WSAAccept,之后，调用

CreateIoCompletionPort((HANDLE) Accept, CompletionPort... )把一个套接字句柄和一个完成端口绑定到一起。完成端口又同一个或多个设备相关联着，所以以套接字为基础，投递发送和请求，对I/O处理。接着，可以依赖完成端口，接收有关I/O操作完成情况的通知。再看程序里：

WSARecv(Accept, &(PerIoData->DataBuf), 1, &RecvBytes, &Flags,

 &(PerIoData->Overlapped), NULL)开始调用，这里象前面讲过的一样，既然是异步I/O，所以WSASend和WSARecv的调用会立即返回。

**系统处理：**

当一个设备的异步I/O请求完成之后，系统会检查该设备是否关联了一个完成端口，如果是，系统就向该完成端口的I/O完成队列中加入完成的I/O请求列。

然后我们需要从这个完成队列中，取出调用后的结果(需要通过一个Overlapped结构来接收调用的结果)。怎么知道这个队列中已经有处理后的结果呢，调用GetQueuedCompletionStatus函数。

工作线程与完成端口：

和异步过程调用不同(在一个Overlapped I/O完成之后，系统调用该回调函数。OS在有信号状态下(设备句柄)，才会调用回调函数（可能有很多APCS等待处理了）)

GetQueuedCompletionStatus

在工作线程内调用GetQueuedCompletionStatus函数。

GetQueuedCompletionStatus(

    HANDLE CompletionPort,

    LPDWORD lpNumberOfBytesTransferred,

    LPDWORD lpCompletionKey,

    LPOVERLAPPED \*lpOverlapped,

    DWORD dwMilliseconds

);

CompletionPort：指出了线程要监视哪一个完成端口。很多服务应用程序只是使用一个I/O完成端口，所有的I/O请求完成以后的通知都将发给该端口。

lpNumberOfBytesTransferred：传输的数据字节数

lpCompletionKey：

完成端口的单句柄数据指针，这个指针将可以得到我们在CreateIoCompletionPort中申请那片内存。

lpOverlapped:

重叠I/O请求结构，这个结构同样是指向我们在重叠请求时所申请的内存块，同时和lpCompletionKey,一样我们也可以利用这个内存块来存储我们要保存的任意数据。

dwMilliseconds:

等待的最长时间(毫秒),如果超时，lpOverlapped被设为NULL，函数返回False.

GetQueuedCompletionStatus功能及隐藏的秘密：

GetQueuedCompletionStatus使调用线程挂起，直到指定的端口的I/O完成队列中出现了一项或直到超时。（I/0完成队列中出现了记录）调用GetQueuedCompletionStatus时，调用线程的ID(cpu\*2个线程，每个ServerWorkerThread的线程ID)就被放入该等待线程队列中。

     等待线程队列很简单，只是保存了这些线程的ID。完成端口会按照后进先出的原则将一个线程队列的ID放入到释放线程列表中。

这样，I/O完成端口内核对象就知道哪些线程正在等待处理完成的I/O请求。当端口的I/O完成队列出现一项时，完成端口就唤醒（睡眠状态中变为可调度状态）等待线程队列中的一个线程。线程将得到完成I/O项中的信息：传输的字节数，完成键(单句柄数据结构)和Overlapped结构地址，线程是通过GetQueuedCompletionStatus返回这些信息，等待CPU的调度。

GetQueuedCompletionStatus返回可能有多种原因，如果传递无效完成端口句柄，函数返回False，GetLastError返回一个错误(ERROR\_INVALID\_HANDLE),如果超时，返回False, GetLastError返回WAIT\_TIMEOUT, i/o完成队列删除一项，该表项是一个成功完成的I/O请求，则返回True。

    调用GetQueuedCompletionStatus的线程是后进先出的方式唤醒的，比如有4个线程等待，如果有一个I/O，最后一个调用GetQueuedCompletionStatus的线程被唤醒来处理。处理完之后，再调用GetQueuedCompletionStatus进入等待线程队列中。

深入分析完成端口线程池调度原理：

    假设我们运行在2CPU的机器上。创建完成端口时指定2个并发，创建了4个工作线程加入线程池中等待完成I/O请求，且完成端口队列（先入先出）中有3个完成I/O的请求的情况：

 工作线程运行, 创建了4个工作线程，调用GetQueuedCompletionStatus时，该调用线程就进入了睡眠状态，假设这个时候，I/O完成队列出现了三项，调用线程的ID就被放入该等待线程队列中, (如图)：

|  |
| --- |
| 等待的线程队列（后进先出） |

|  |
| --- |
| 进队列 |

|  |
| --- |
| 出队列 |

|  |
| --- |
| 线  程  A |

|  |
| --- |
| 线  程  B |

|  |
| --- |
| 线  程  C |

|  |
| --- |
| 线  程  D |

I/O完成端口内核对象（第3个参数等级线程队列）,因此知道哪些线程正在等待处理完成的I/O请求。当端口的I/O完成队列出现一项时，完成端口就唤醒（睡眠状态中变为可调度状态）等待线程队列中的一个线程(前面讲过等待线程队列是后进先出)。所以线程D将得到完成I/O项中的信息：传输的字节数，完成键(单句柄数据结构)和Overlapped结构地址，线程是通过GetQueuedCompletionStatus返回这些信息。

在前面我们指定了并发线程的数目是2，所以I/O完成端口唤醒2个线程，线程D和线程C，另两个继续休眠（线程B，线程A），直到线程D处理完了，发现表项里还有要处理的，就唤醒同一线程继续处理。

|  |
| --- |
| 等待的线程队列（后进先出） |

|  |
| --- |
| 进队列 |

|  |
| --- |
| 出队列 |

|  |
| --- |
| 线  程  A |

|  |
| --- |
| 线  程  B |

|  |
| --- |
| 释放线程队列 |

|  |
| --- |
| 线  程  C |

|  |
| --- |
| 线  程  D |

线程并发量：

    并发量限制了与该完成端口相关联的可运行线程的数目, 它类似阀门的作用。 当与该完成端口相关联的可运行线程的总数目达到了该并发量，系统就会阻塞任何与该完成端口相关联的后续线程的执行， 直到与该完成端口相关联的可运行线程数目下降到小于该并发量为止。所以解释了线程池中的运行线程可能会比设置的并发线程多的原因。

    它的作用:

最有效的假想是发生在有完成包在队列中等待，而没有等待被满足，因为此时完成端口达到了其并发量的极限。此时，一个正在运行中的线程调用 GetQueuedCompletionStatus时，它就会立刻从队列中取走该完成包。这样就不存在着环境的切换，因为该处于运行中的线程就会连续不断地从队列中取走完成包，而其他的线程就不能运行了。

注意：如果池中的所有线程都在忙，客户请求就可能拒绝，所以要适当调整这个参数，获得最佳性能。

**线程并发：**D线程挂起，加入暂停线程，醒来后又加入释放线程队列。

|  |
| --- |
| 线  程  C |

|  |
| --- |
| 线  程  B |

|  |
| --- |
| 线  程  A |

|  |
| --- |
| 出队列 |

|  |
| --- |
| 进队列 |

|  |
| --- |
| 等待的线程队列（后进先出） |

|  |
| --- |
| 释放线程队列 |

|  |
| --- |
| 暂停线程 |

|  |
| --- |
| 线  程  D |

线程的安全退出：

PostQueudCompletionStatus函数，我们可以用它发送一个自定义的包含了OVERLAPPED成员变量的结构地址，里面包含一个状态变量，当状态变量为退出标志时，线程就执行清除动作然后退出。

完成端口使用需要注意的地方：

 1．在执行wsasend和wsarecv操作前，请先将overlapped结构体使用memset进行清零。

于敦德 2006-3-16

**一、LiveJournal发展历程**

[LiveJournal](http://www.livejournal.com/)是99年始于校园中的项目，几个人出于爱好做了这样一个应用，以实现以下功能：

* 博客，论坛
* 社会性网络，找到朋友
* 聚合，把朋友的文章聚合在一起

LiveJournal采用了大量的开源软件，甚至它本身也是一个开源软件。

在上线后，LiveJournal实现了非常快速的增长：

* 2004年4月份：280万注册用户。
* 2005年4月份：680万注册用户。
* 2005年8月份：790万注册用户。
* 达到了每秒钟上千次的页面请求及处理。
* 使用了大量MySQL服务器。
* 使用了大量通用组件。

**二、LiveJournal架构现状概况**

**三、从LiveJournal发展中学习**

LiveJournal从1台服务器发展到100台服务器，这其中经历了无数的伤痛，但同时也摸索出了解决这些问题的方法，通过对LiveJournal的学习，可以让我们避免LJ曾经犯过的错误，并且从一开始就对系统进行良好的设计，以避免后期的痛苦。

下面我们一步一步看LJ发展的脚步。

**1、一台服务器**

一台别人捐助的服务器，LJ最初就跑在上面，就像Google开始时候用的破服务器一样，值得我们尊敬。这个阶段，LJ的人以惊人的速度熟悉的Unix的操作管理，服务器性能出现过问题，不过还好，可以通过一些小修小改应付过去。在这个阶段里LJ把CGI升级到了FastCGI。

最终问题出现了，网站越来越慢，已经无法通过优过化来解决的地步，需要更多的服务器，这时LJ开始提供付费服务，可能是想通过这些钱来购买新的服务器，以解决当时的困境。  
毫无疑问，当时LJ存在巨大的单点问题，所有的东西都在那台服务器的铁皮盒子里装着。

**2、两台服务器**

用付费服务赚来的钱LJ买了两台服务器：一台叫做Kenny的Dell 6U机器用于提供Web服务，一台叫做Cartman的Dell 6U服务器用于提供数据库服务。

LJ有了更大的磁盘，更多的计算资源。但同时网络结构还是非常简单，每台机器两块网卡，Cartman通过内网为Kenny提供MySQL数据库服务。  
  
暂时解决了负载的问题，新的问题又出现了：

* 原来的一个单点变成了两个单点。
* 没有冷备份或热备份。
* 网站速度慢的问题又开始出现了，没办法，增长太快了。
* Web服务器上CPU达到上限，需要更多的Web服务器。

**3、四台服务器**

又买了两台，Kyle和Stan，这次都是1U的，都用于提供Web服务。目前LJ一共有3台Web服务器和一台数据库服务器。这时需要在3台Web服务器上进行负载均横。

LJ把Kenny用于外部的网关，使用mod\_backhand进行负载均横。

然后问题又出现了：

* 单点故障。数据库和用于做网关的Web服务器都是单点，一旦任何一台机器出现问题将导致所有服务不可用。虽然用于做网关的Web服务器可以通过保持心跳同步迅速切换，但还是无法解决数据库的单点，LJ当时也没做这个。
* 网站又变慢了，这次是因为IO和数据库的问题，问题是怎么往应用里面添加数据库呢？

**4、五台服务器**

又买了一台数据库服务器。在两台数据库服务器上使用了数据库同步(Mysql支持的Master-Slave模式)，写操作全部针对主数据库（通过Binlog，主服务器上的写操作可以迅速同步到从服务器上），读操作在两个数据库上同时进行(也算是负载均横的一种吧)。

实现同步时要注意几个事项：

* 读操作数据库选择算法处理，要选一个当前负载轻一点的数据库。
* 在从数据库服务器上只能进行读操作
* 准备好应对同步过程中的延迟，处理不好可能会导致数据库同步的中断。只需要对写操作进行判断即可，读操作不存在同步问题。

**5、更多服务器**

有钱了，当然要多买些服务器。部署后快了没多久，又开始慢了。这次有更多的Web服务器，更多的数据库服务器，存在 IO与CPU争用。于是采用了BIG-IP作为负载均衡解决方案。

**6、现在我们在哪里：**

现在服务器基本上够了，但性能还是有问题，原因出在架构上。

数据库的架构是最大的问题。由于增加的数据库都是以Slave模式添加到应用内，这样唯一的好处就是将读操作分布到了多台机器，但这样带来的后果就是写操作被大量分发，每台机器都要执行，服务器越多，浪费就越大，随着写操作的增加，用于服务读操作的资源越来越少。

由一台分布到两台

最终效果

现在我们发现，我们并不需要把这些数据在如此多的服务器上都保留一份。服务器上已经做了RAID，数据库也进行了备份，这么多的备份完全是对资源的浪费，属于冗余极端过度。那为什么不把数据分布存储呢？

问题发现了，开始考虑如何解决。现在要做的就是把不同用户的数据分布到不同的服务器上进行存储，以实现数据的分布式存储，让每台机器只为相对固定的用户服务，以实现平行的架构和良好的可扩展性。

为了实现用户分组，我们需要为每一个用户分配一个组标记，用于标记此用户的数据存放在哪一组数据库服务器中。每组数据库由一个master及几个slave组成，并且slave的数量在2-3台，以实现系统资源的最合理分配，既保证数据读操作分布，又避免数据过度冗余以及同步操作对系统资源的过度消耗。

由一台（一组）中心服务器提供用户分组控制。所有用户的分组信息都存储在这台机器上，所有针对用户的操作需要先查询这台机器得到用户的组号，然后再到相应的数据库组中获取数据。

这样的用户架构与目前LJ的架构已经很相像了。

在具体的实现时需要注意几个问题：

* 在数据库组内不要使用自增ID，以便于以后在数据库组之间迁移用户，以实现更合理的I/O，磁盘空间及负载分布。
* 将userid，postid存储在全局服务器上，可以使用自增，数据库组中的相应值必须以全局服务器上的值为准。全局服务器上使用事务型数据库InnoDB。
* 在数据库组之间迁移用户时要万分小心，当迁移时用户不能有写操作。

**7、现在我们在哪里**

问题：

* 一个全局主服务器，挂掉的话所有用户注册及写操作就挂掉。
* 每个数据库组一个主服务器，挂掉的话这组用户的写操作就挂掉。
* 数据库组从服务器挂掉的话会导致其它服务器负载过大。

对于Master-Slave模式的单点问题，LJ采取了Master-Master模式来解决。所谓Master-Master实际上是人工实现的，并不是由MySQL直接提供的，实际上也就是两台机器同时是Master，也同时是Slave，互相同步。

Master-Master实现时需要注意：

* 一个Master出错后恢复同步，最好由服务器自动完成。
* 数字分配，由于同时在两台机器上写，有些ID可能会冲突。

解决方案：

* 奇偶数分配ID，一台机器上写奇数，一台机器上写偶数
* 通过全局服务器进行分配(LJ采用的做法)。

Master-Master模式还有一种用法，这种方法与前一种相比，仍然保持两台机器的同步，但只有一台机器提供服务（读和写），在每天晚上的时候进行轮换，或者出现问题的时候进行切换。

**8、现在我们在哪里**

现在插播一条广告，MyISAM VS InnoDB。

使用InnoDB：

* 支持事务
* 需要做更多的配置，不过值得，可以更安全的存储数据，以及得到更快的速度。

使用MyISAM：

* 记录日志（LJ用它来记网络访问日志）
* 存储只读静态数据，足够快。
* 并发性很差，无法同时读写数据（添加数据可以）
* MySQL非正常关闭或死机时会导致索引错误，需要使用myisamchk修复，而且当访问量大时出现非常频繁。

**9、缓存**

去年我写过[一篇文章介绍memcached](http://www.example.net.cn/archives/2006/01/eoamemcachedoea.html)，它就是由LJ的团队开发的一款缓存工具，以key-value的方式将数据存储到分布的内存中。LJ缓存的数据：

* 12台独立服务器（不是捐赠的）
* 28个实例
* 30GB总容量
* 90-93%的命中率（用过squid的人可能知道，squid内存加磁盘的命中率大概在70-80%）

如何建立缓存策略？

想缓存所有的东西？那是不可能的，我们只需要缓存已经或者可能导致系统瓶颈的地方，最大程度的提交系统运行效率。通过对MySQL的日志的分析我们可以找到缓存的对象。

缓存的缺点？

* 没有完美的事物，缓存也有缺点：
* 增大开发量，需要针对缓存处理编写特殊的代码。
* 管理难度增加，需要更多人参与系统维护。
* 当然大内存也需要钱。

**10、Web访问负载均衡**

在数据包级别使用BIG-IP，但BIG-IP并不知道我们内部的处理机制，无法判断由哪台服务器对这些请求进行处理。反向代理并不能很好的起到作用，不是已经够快了，就是达不到我们想要的效果。

所以，LJ又开发了[Perlbal](http://www.danga.com/perlbal/)。特点：

* 快，小，可管理的http web 服务器/代理
* 可以在内部进行转发
* 使用Perl开发
* 单线程，异步，基于事件，使用epoll , kqueue
* 支持Console管理与http远程管理，支持动态配置加载
* 多种模式：web服务器，反向代理，插件
* 支持插件：GIF/PNG互换？

**11、MogileFS**

LJ使用开源的[MogileFS](http://www.danga.com/mogilefs/)作为分布式文件存储系统。MogileFS使用非常简单，它的主要设计思想是：

* 文件属于类（类是最小的复制单位）
* 跟踪文件存储位置
* 在不同主机上存储
* 使用MySQL集群统一存储分布信息
* 大容易廉价磁盘

到目前为止就这么多了，更多文档可以在<http://www.danga.com/words/>找到。[Danga.com](http://www.danga.com/)和[LiveJournal.com](http://www.livejournal.com/)的同学们拿这个文档参加了两次MySQL Con，两次OS Con，以及众多的其它会议，无私的把他们的经验分享出来，值得我们学习。在web2.0时代快速开发得到大家越来越多的重视，但良好的设计仍是每一个应用的基础，希望web2.0们在成长为Top500网站的路上，不要因为架构阻碍了网站的发展。