2020/07/27_Windows编程_第7课_进程间的通讯、共享内存、内存映射

笔记本:	Windows编程
创建时间:	2020/7/27 星期一 10:04
作者:	ileemi
标签:	进程间的通讯, 使用内存映射操作文件

进程间的通讯

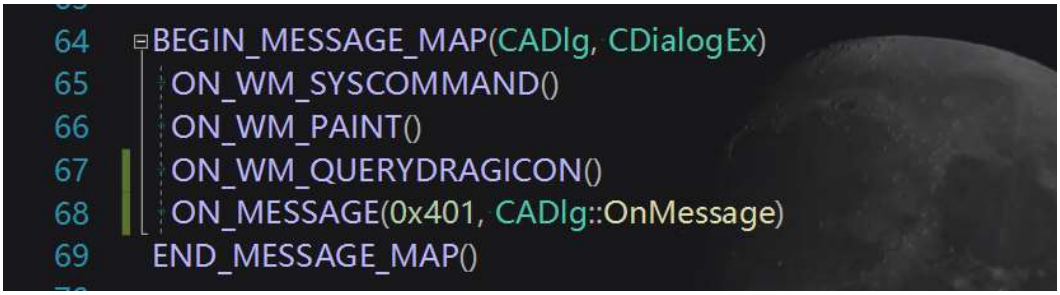
通过QQ登录游戏，网易云音乐转发朋友圈，将一个dll注入到一个进程中等，都需要进程间的相互通讯。

Winsdows进程间的"打电话通讯" -- 消息机制（一个进程发送消息，一个进程接收消息（接收消息的程序只能是窗口程序）

简单模拟

创建两个 MFC 对话框工程来模仿进程间的通讯，一个程序模仿发送方，一个程序模仿接收方。
MFC 中响应自定义消息需要使用一个宏 -- `ON_MESSAGE` 。

B进程（发送方）：



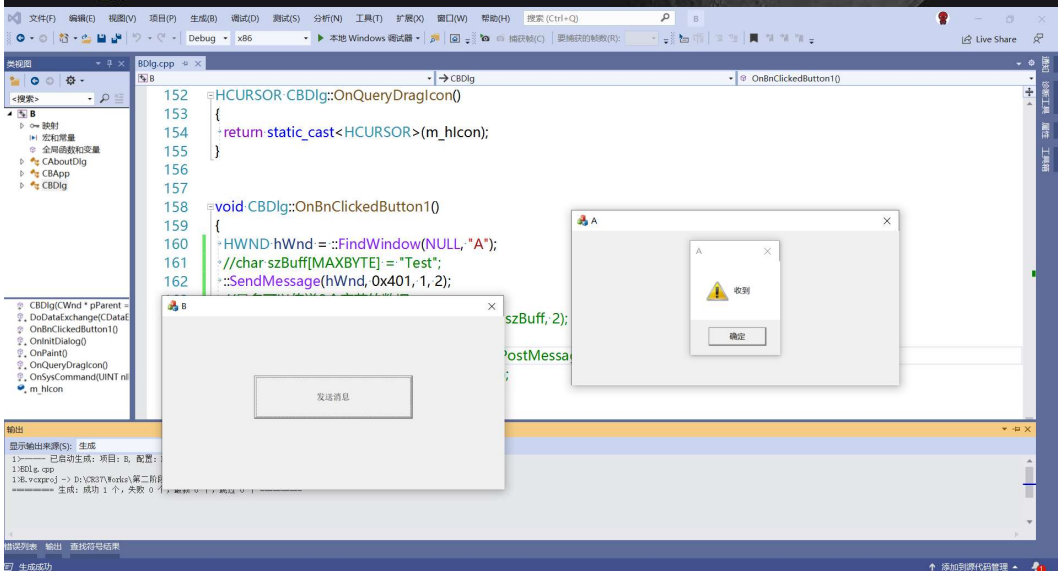
```
158
159  void CBDlg::OnBnClickedButton1()
160  {
161      HWND hWnd = ::FindWindow(NULL, "A");
162      char szBuff[MAXBYTE] = "Test";
163      //::SendMessage(hWnd, 0x401, 1, 2);
164      // 最多可以传递8个字节的数据
165      ::SendMessage(hWnd, 0x401, (WPARAM)szBuff, 2);
166  }
167
```

B进程，发送消息

A进程（接收方）：

```
157  LRESULT CADlg::OnMessage(WPARAM wParam, LPARAM lParam)
158  {
159      char szBuff[MAXBYTE];
160      //wsprintf(szBuff, "收到：  %d %d", wParam, lParam);
161      // 传递一个缓冲区程序会崩
162      wsprintf(szBuff, "收到：  %s %d", wParam, lParam);
163      AfxMessageBox(szBuff);
164      return LRESULT();
165  }
```

A 程序，接收消息

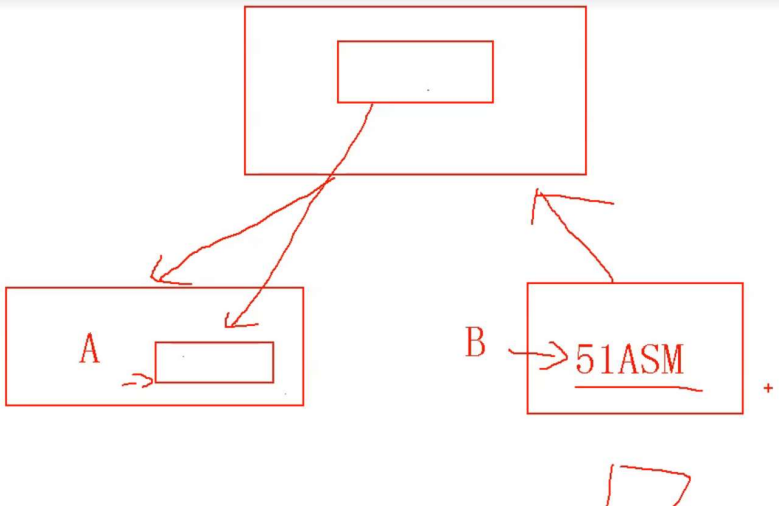


> 注意：
> 两个进程中的内存数据是不一样的，通过 SendMessage 发送消息，只能发送8个字节的数据。发送一个字符串的地址，接收消息的进程没有对应的内存数据，**会导致程序崩溃**。每个进程的虚拟地址是相互隔离的。

WM_COPYDATA

微软提供一个 WM_COPYDATA 消息，发送数据进程向系统发送一个拷贝消息，系统会将要发送的数据拷贝到系统空间去（只拷贝要发送的数据），然后再将数据拷贝到目标进程中去，系统会为目标进程申请一个空间。（拷贝后的数据地址不同，内存有系统

自动分配，使用完毕，系统自动回收）。



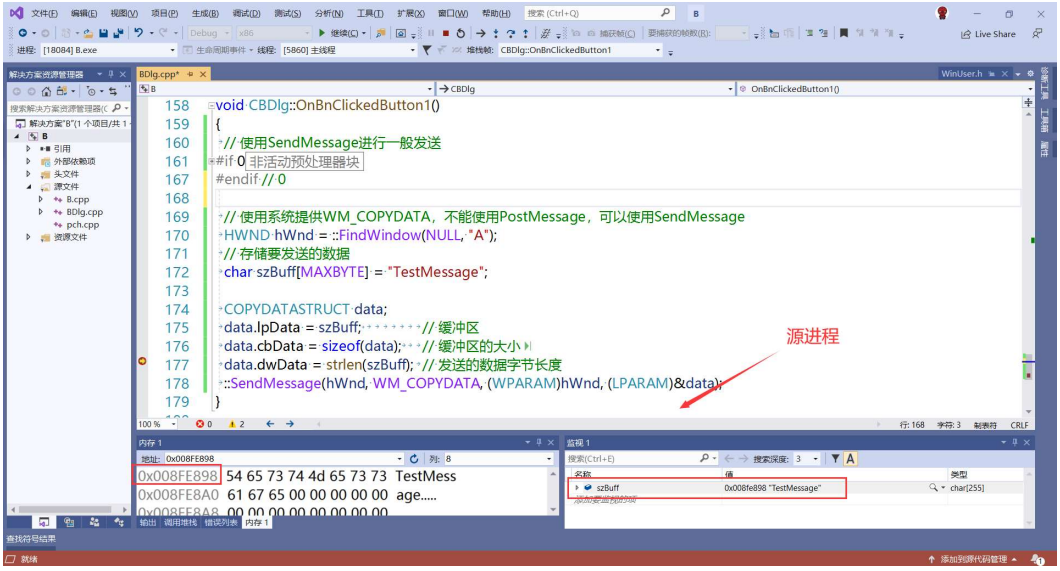
使用 WM_COPYDATA 发送的消息不能使用 PostMessage 函数，这个函数系统会将消息发送到消息队列中，通过消息队列系统需要申请内存，内存的释放，消息处理的时机不确定。

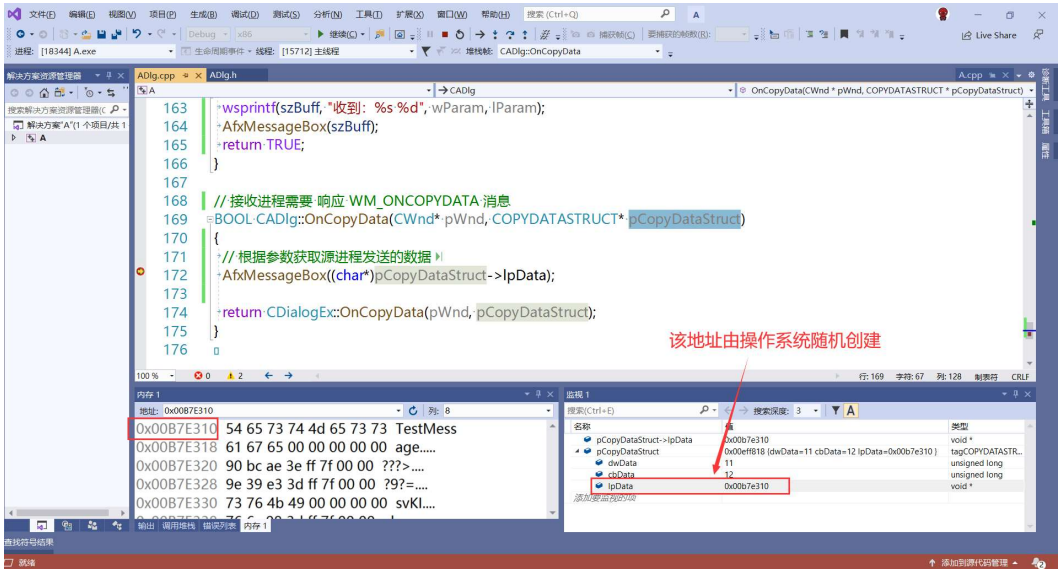
使用 SendMessage 函数，其会**直接调用过程函数**，过程函数调用完后，意味着消息处理完毕，之后就可以就行释放操作。

> WM_COPYDATA 函数说明：
> 一个应用程序发送 WM_COPYDATA 消息来将数据传递给另一个应用程序。
> 要发送此消息，请使用以下参数调用 SendMessage 函数(不要调用 PostMessage 函数)。
> 参数4需要定义一个 COPYDATASTRUCT 结构体，这个结构体是告诉操作系统的，不是告诉对方进程的。

响应 WM_COPYDATA 消息，发送方使用 SendMessage， 接收方需要响应 OnCopyData消息。

代码示例：

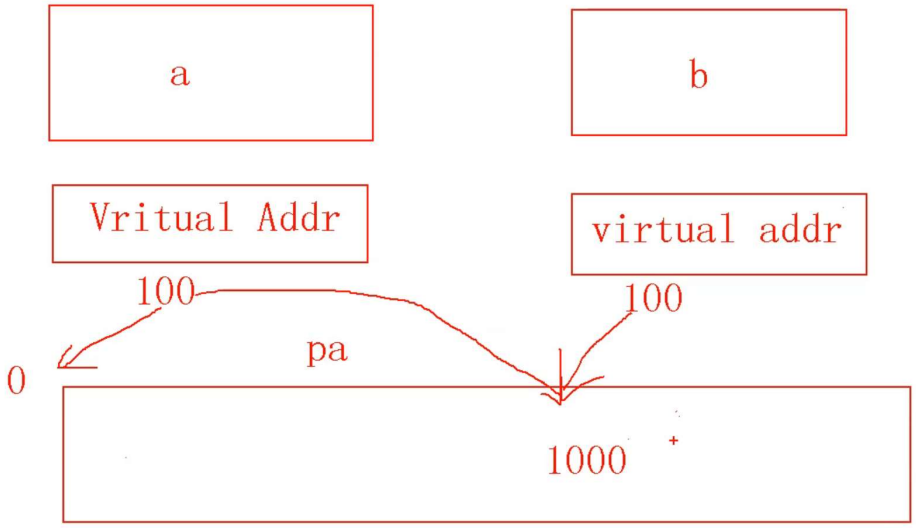




存在缺点：不适合传递大量数据，效率比较低，存在拷贝时间的问题。

进程间共享内存

不用在使用内存拷贝，省去拷贝数据的时间问题，A进程改动，B进程接收。每个进程都有自己的虚拟地址，每个进程的数据都会在物理内存条中，在物理内存中两个进程间的同一个内存地址的数据访问数据是相互隔离的：



共享内存，需要进程双方同时申请（向操作系统申请）。

DLL共享段

在 dll 中创建一块内存共享区域，加载 dll 的进程会使用 dll 中的这块共享内存区域。
存在缺点：一个进程破坏了 dll，另一个进程再次使用这个 dll，就会受到影响。

步骤：

1. 创建一个动态库，源进程和目标进程都使用这个Dll
2. Dll 的编写不需要添加导出函数，只需要使编译器提供的关键字去定义数据段和共享段

代码示例：

```
// dllmain.cpp : 定义 DLL 应用程序的入口点。
#include "pch.h"

// 定义一个数据段
#pragma data_seg( "MY_DATA" )
  // 此处填写需要共享的数据
  char g_szBuff[] = "Hello World";
  int g_nNum = 999;
#pragma data_seg()

// 定义一个共享段，将需要共享的数据进行共享 "RWS" -- 数据共享的权限
#pragma comment(linker, "/SECTION:MY_DATA,RWS")

BOOL APIENTRY DllMain(
  HMODULE hModule,
  DWORD  ul_reason_for_call,
  LPVOID lpReserved )
{
  switch (ul_reason_for_call)
  {
  case DLL_PROCESS_ATTACH:
  case DLL_THREAD_ATTACH:
  case DLL_THREAD_DETACH:
  case DLL_PROCESS_DETACH:
    break;
  }
  return TRUE;
}

// 添加.def文件，使其导出 .lib 文件
LIBRARY
EXPORTS
    g_szBuff
    g_nNum
```

使用 DLL 共享

步骤：

1. 在使用的进程中导入动态库，加载对应的共享内存Dll（这里静态使用）。没有导出函数就没有对应的 .lib 文件。这里在源进程个目标进程中分别导入要使用的

dll。

```
#pragma once

#pragma comment(lib, "../../MyDll/Debug/MyDll.lib")
_declspec(dllimport) char g_szBuff[];
_declspec(dllimport) int g_nNum;

```

2. 在源进程中添加代码，尝试修改 dll 中的导出数据，之后发送消息给目标进程 A。

```

void CBDlg::OnBnClickedButton1()
{
// 获取目标窗口句柄
HWND hWnd = ::FindWindow(NULL, "A");
// 修改dll 中共享内存中的变量数据
g_szBuff[0] = 'h';
g_nNum = 666;
// 向目标进程发送消息
::SendMessage(hWnd, 0x401, NULL, NULL);
}

```

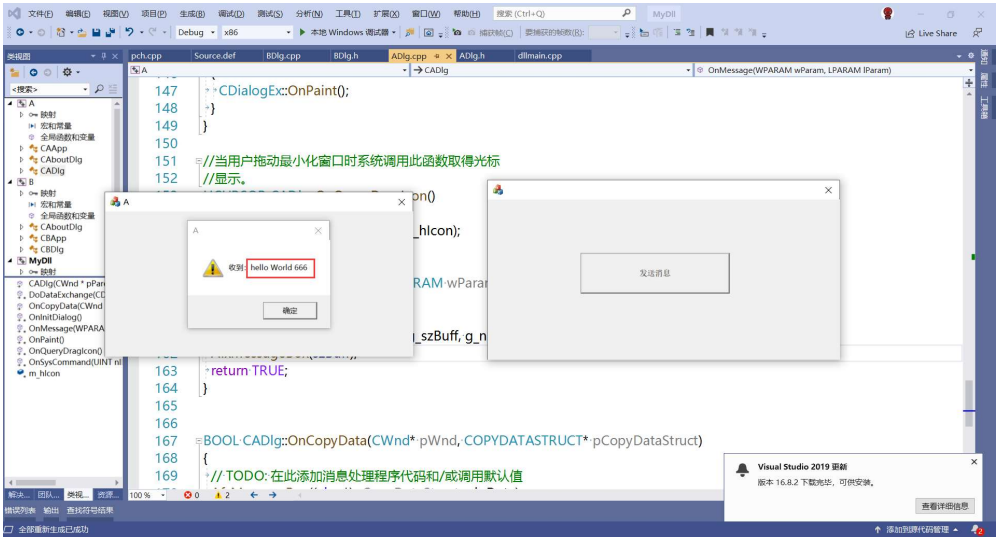
3. 在目标进程中接收源进程发送过了的消息。

```

LRESULT CADlg::OnMessage(WPARAM wParam, LPARAM lParam)
{
char szBuff[MAXBYTE];
wsprintf(szBuff, "收到： %s %d", g_szBuff, g_nNum);
AfxMessageBox(szBuff);
return TRUE;
}

```

程序运行效果图如下：



缺点：想要使用内存共享，进程间都需要同时加载dll，不需要Dll的时候，这种方法就比较麻烦。

> SetWindowLong 返回值为旧的函数地址
> 修改对方游戏的过程函数，对方游戏的代码暂时不会运行

内存映射（比较常用）

自己映射内存，自己完成映射，告诉操作系统需要一块共享的内存

创建文件映射

使用内存映射操作文件

> 创建内存/文件映射 API：
> `CreateFileMapping` （两种用法） -- 可以文件映射，也可以内存映射

文件映射：
将文件映射到内存中的一块地址中（为文件映射一块内存），操作这块内存就等价于操作这个文件，移动文件指针相当于移动文件在内存中的地址。操作系统会自动将内存中的数据保存到文件中去。

步骤：

1. 使用内存映射操作文件：

   > 打开文件 -- API `CreateFile` （比库函数 `fopen` 功能强大）
   > 参数3：文件的共享方式（文件打开的情况下，是否允许别的进程读、写等），0 为不共享。

2. 创建文件映射对象（返回文件映射句柄）：

   > API：`CreateFileMapping`，参数4 -- 高32位，参数5 -- 低32位
   > DWORD 可以放置4个G的内存，映射对象的名字可以为空，映射的内存大小不能为1字节（映射的内存最低为一个分页 0x1000（4096个字节），映射的内存大小填写0，映射的内存大小为文件的大小。操作系统以页为单位管理内存的。**映射的内存大小不够一个分页，系统会自动申请一个分页的映射内存。**

3. 映射内存文件

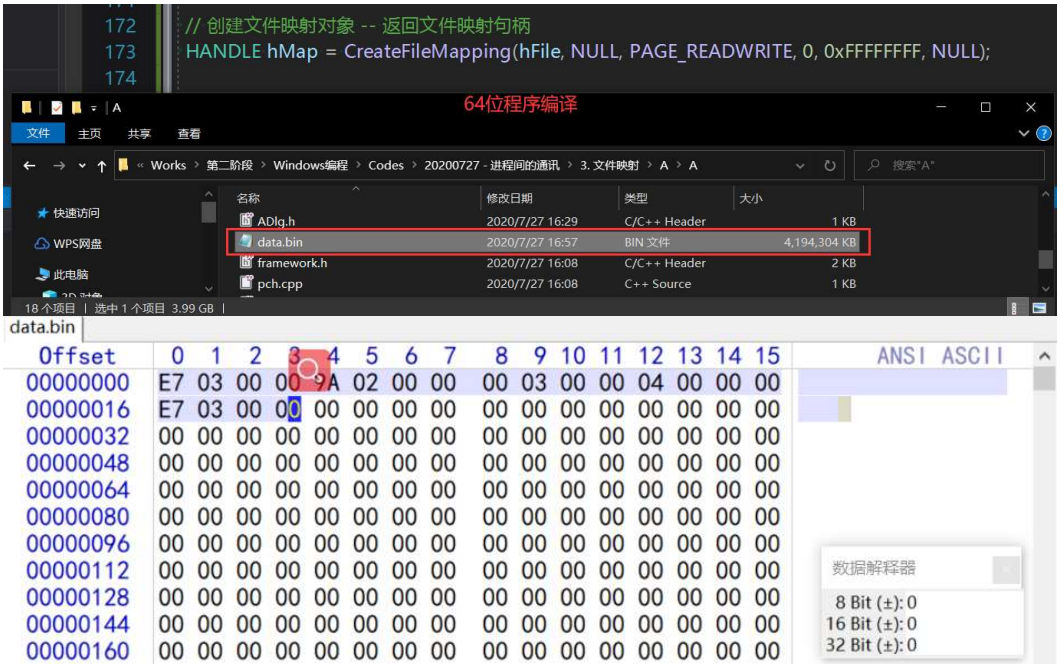
   > API：`MapViewOfFile`，参数3、4为文件的便宜

4. 操作内存

5. 取消文件映射（关闭映射）

   > API：`UnmapViewOfFile` -- 参数为 `MapViewOfFile` 的返回值
   > 取消映射，关闭文件后，修改的内存数据会自动更新。修改数据并实时刷新内存数据到文件 API -- `FlushViewOfFile`，函数说明：将文件映射视图中的一个字节范围写入磁盘（一个字节一个字节写），这种方式效率比较低。一次性写入效率高。

映射的内存过大的时候，会根据文件指针，分批进行映射。映射的内存大小填写0，映射的内存大小为文件的大小。
映射的内存为0xFFFFFFFF时，32位程序（最多可以申请2GB）会崩溃，64位程序不会崩。



代码示例:

```
// 使用内存映射去操作文件
void CADlg::OnBnClickedButton2()
{
    /*
    1. 创建文件映射 -- CreateFile,为文件映射一段内存
    相关API:
    ReadFile, MoveFile, WriteFile, SetFilePointer
    */
    HANDLE hFile = CreateFile("data.bin",
        GENERIC_READ | GENERIC_WRITE, // 文件的操作属性
        0, // 不共享，文件打开的时候，其它程序不能进行操作
        NULL, // 安全属性，不继承
        OPEN_EXISTING, // 打开已存在的文件
        FILE_ATTRIBUTE_NORMAL, // 文件的类型（常规文件）
        NULL // 文件句柄
    );

    // 2. 创建文件映射对象 -- 返回文件映射句柄
    HANDLE hMap = CreateFileMapping(hFile,
    NULL, PAGE_READWRITE,
    0, 0x4096, NULL);

    // 3. 映射内存 -- 返回值是映射视图的起始地址
    LPVOID lpBuff = MapViewOfFile(hMap,
    FILE_MAP_ALL_ACCESS, 0, 0, 0);

    // 4. 操作内存 lpBuff -- 文件的偏移
    int nNum = 999;
    memcpy(lpBuff, &nNum, sizeof(nNum)); // 修改文件数据
```

```
    int  nNum2  =  666;
    memcpy((char*)lpBuff  +  4,  &nNum2,  sizeof(nNum2));      //  修改
文件数据
    memcpy((char*)lpBuff  +  16,  &nNum,  sizeof(nNum));       //  修改
文件数据
    //  实时刷新内存数据到文件  --    FlushViewOfFile

    //  5.  取消映射
    UnmapViewOfFile(lpBuff);

    //  6.  关闭文件句柄
    CloseHandle(hMap);
    CloseHandle(hFile);
}
```

创建内存映射

操作流程和创建文件映射流程一样，但是不需要打开文件。
多次创建同一个内存映射不会成功，只会创建一个。

> 关闭内存映射的时机，应该是当窗口关闭的时候，关闭内存映射（将文件映射句柄和内存的缓冲区保存起来），在构造函数中给定初始化。

关闭窗口的消息：`WM_DESTROY`

代码示例：

```
在对话框类中保存 "内存映射的句柄" 以及 "映射内存的起始地址"
//////////////////////////////////////////////////////////////////////////

private:
    HANDLE  m_hMap;     //  内存映射句柄
    LPVOID  m_lpBuff;//  映射内存的起始地址
//////////////////////////////////////////////////////////////////////////


//构造函数:
CADlg::CADlg(CWnd*  pParent  /*=nullptr*/)
    :  CDialogEx(IDD_A_DIALOG,  pParent)
{
    m_hIcon  =  AfxGetApp()->LoadIcon(IDR_MAINFRAME);
    m_hMap  =  INVALID_HANDLE_VALUE;
    m_lpBuff  =  NULL;
}

//////////////////////////////////////////////////////////////////////////
```

```
// 创建内存映射
void CADlg::OnBnClickedButton3()
{
    // 检测应用双开
    m_hMap = OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE, "TestFileMapNar

    if (m_hMap != NULL)
    {
        AfxMessageBox("程序已经运行");
        EndDialog(0);    // 关闭对话框
        return;
    }

    // 创建内存映射 -- 返回文件映射句柄
    m_hMap = CreateFileMapping(NULL, NULL,
    PAGE_READWRITE, 0, 0x4096, "TestFileMapName");

    // 映射内存 -- 返回值是映射视图的起始地址
    m_lpBuff = MapViewOfFile(m_hMap,
    FILE_MAP_ALL_ACCESS, 0, 0, 0);

    // 操作内存
    int nNum = 999;
    memcpy(m_lpBuff, &nNum, sizeof(nNum));        // 修改文件数据
    int nNum2 = 666;
    memcpy((char*)m_lpBuff + 4, &nNum2, sizeof(nNum2)); // 修改文
件数据
    memcpy((char*)m_lpBuff + 16, &nNum, sizeof(nNum));    // 修改文
件数据
    // 取消映射，关闭文件句柄在销毁窗口的时候进行关闭
}

// 关闭窗口时，响应该消息
void CADlg::OnDestroy()
{
    CDialogEx::OnDestroy();
    if (m_hMap != INVALID_HANDLE_VALUE)
    {
        // 关闭文件句柄
        CloseHandle(m_hMap);
    }
    if (m_lpBuff != NULL)
    {
        // 取消映射
        UnmapViewOfFile(m_lpBuff);
```

```
    }
}
```

读取内存映射

1. 将内存映射的句柄继承给需要读取内存映射的程序（需要两个程序是父子关系）
2. 通过拷贝进程句柄（较为麻烦，需要调用的API比较多）
3. 在创建进程映射的时候（CreateFileMapping），添加一个进程映射名（将内存映射这个进程和这个名字进行了绑定）。不给名字 -- 匿名映射（映射文件使用，不能跨进程通讯）

> 创建内存映射是在物理内存中创建的映射，不是在进程内存中创建的，进程内存不能共享。

在读取内存映射的程序中使用 OpenFileMapping **打开一个指定名称的文件映射对象**，之后两个进程的文件句柄指向的是一个内存映射对象，在映射的时候，内存地址就是共享的，内存地址不一样，但是物理内存地址是共享的。

当A进程取消映射的时候，B进程还可以继续访问内存映射，**当两个进程都不映射内存的时候，操作系统才会释放这块物理内存（内部有一个引用计数）。**

> **映射的内存地址不能自己指定，由操作系统自动分配。**

无论进程开启多少次，都可以共享内存映射中数据，创建内存映射不能重复创建，所以可以使用下面的代码进行检测：

```
// 检测应用双开
m_hMap = OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE, "TestFileMapName");
if (m_hMap != NULL)
{
  AfxMessageBox("程序已经运行");
  EndDialog(0);// 关闭对话框
  return;
}
```

代码示例：

```
// 创建内存映射的代码在上面

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

//  读取内存映射
void  CBDlg::OnBnClickedButton1()
{
    HANDLE  hMap  =  INVALID_HANDLE_VALUE;
```

```
    // 打开一个指定名称的文件映射对象
    hMap = OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE, "TestFileMapName"

    // 映射内存 -- 返回值是映射视图的起始地址
    LPVOID lpBuff = MapViewOfFile(hMap, FILE_MAP_ALL_ACCESS, 0, 0, 0)

    // 操作内存，读取内存数据
    int nNum = 0;
    memcpy(&nNum, lpBuff, sizeof(nNum));

    // 将读取到的内存数据弹出
    char szBuff[MAXBYTE];
    wsprintf(szBuff, "%d", *(int*)lpBuff);
    AfxMessageBox(szBuff);

    // 取消映射
    UnmapViewOfFile(lpBuff);

    // 关闭文件句柄
    CloseHandle(hMap);
}
```

内存映射以及Dll共享的缺点：在修改数据的时候，对方进程不知道。

创建的内存映射其实是在物理内存条上创建的空间，不是在进程内存中创建的，因为进程的内存不是共享的。

管道

给每个进程都 "**插入**" 一个管子，在修改数据的时候，实时通知对方进程内存数据进行了修改。不需要通知的话，共享内存就很合适。