浅析 MFC 程序基本运行机制

或许我不该写这篇文章；或许你会不屑的看了看标题，然后华丽的 WS 之；又或许你会在看完之后，在这篇文章的末尾的写上“打倒 KC，打倒 MFC，打倒 M$”，然后签上自己伟大的名字。

但是这都丝毫不会影响包括我在内的所有想了解 MFC 的 Coder 对于 MFC 研究。或许，有好几个问题曾连续地在你的脑海里浮现：

MFC 很容易学么？是的，很容易。但是前提是你首先得理解他的源代码，并且看懂背后的故事。

MFC 很复杂么？是的，以 MFC4.X 来说，仅是一个单独的源文件都有超过 120000 行的代码，这还不算头文件和.H 扩展文件。

MFC 很强大么？是的，它不仅能让你更加了解 Windows 系统的运行机理，还能让你从传统 SDK 的束缚中解脱出来。

MFC 很恶心么？是的，它会时常让你感到，不是你在控制 MFC，而是 MFC 在控制你

MFC 很完美么？没有完美的思想，也没有完美的程序。从产生人类文明至今，尚未出现真正称得上“完美”的东西。因为我们在进步，在革命。

……

常言道：知己知彼，百战不殆。如果你真的决定使用 MFC，那么你就应该好好的研究它的内部运行机制。这不是关键性的，但是是必要性的。

而这篇文章，就向大家展示了 MFC 程序的基本运行机制。

1．温故知新

在研究 MFC 的基本运行机制之前，先让我们来回忆一下使用 C++/SDK 写 Windows GUI 程序的顺序：

调用 WinMain 入口函数→注册窗口类→窗口实例化→建立消息循环→处理消息

可以说，几乎每个 Windows GUI 程序的建立和运行，都要经过上面的几个步骤，MFC 程序也不例外。但是由于 MFC 是以 C++为基础，所以它势必会使用 OOP 思想进行架构。而这一切，都会导致我们研究 MFC 的方式会和 C++/SDK 有那么一点区别。

我们在下面会以 MFC 的 Class 为中心进行研究，而非 Windows 窗体的线性行为。这意味着我们得忍受在几个类中跳来跳去。是的，你可能会感到身体不适，我同样有这种感觉，我从小就恨透了 goto……

那么，就让我们先来看看使用 MFC 改如何创建一个简单的窗体，然后在逐步抽丝剥茧，剖析 MFC 程序的基本运行机理。

```
/**********************************************

** Project:MFCAppUser

** File:MFCAppUser.cpp
```

```
** Edition:NULL

** Coder:KingsamChen [MDSA Group]

** Last Modify:2008-8-9

************************************************/

#include <afxwin.h>  // 必备的头文件,这个头文件间接包含了 windows.h

class CMFCApp : public CWinApp  // 继承 CWinApp
{
 public:
  virtual BOOL InitInstance();  // 虚函数
                                // 这个函数必须重写
};

class CMFCAppWindow : public CFrameWnd
{
 public:
  CMFCAppWindow()  // 在构造函数里创建窗体~
   {
    Create(NULL,"KC's Windows");  // 除前两个参数外,其他参数均有初始值
   }

  // 下面是消息映射的东东
   afx_msg void OnLButtonDblClk(UINT nFlags, CPoint point);  // 左键双击的消息声明
   afx_msg void OnPaint();  // WM_PAINT 消息声明
   DECLARE_MESSAGE_MAP()  // 消息映射宏
};

// MS 叫做消息映射表
BEGIN_MESSAGE_MAP(CMFCAppWindow, CFrameWnd)
 ON_WM_LBUTTONDBLCLK()
 ON_WM_PAINT()
END_MESSAGE_MAP()

// 对应的消息处理,类似 SDK 窗体里面的回调函数处理过程
void CMFCAppWindow::OnLButtonDblClk(UINT nFlags, CPoint point)
{
 MessageBox("KC is a Fucker", NULL, MB_OK);
}

void CMFCAppWindow::OnPaint()
{
 CPaintDC Paint(this);
 Paint.TextOut(0, 0, "This is a sample for MFC");
```

```
}

// 重写 InitInstance 虚函数
BOOL CMFCApp::InitInstance()
{
  m_pMainWnd = new CMFCAppWindow();
  m_pMainWnd->ShowWindow(m_nCmdShow);
  m_pMainWnd->UpdateWindow();

  return TRUE;
}

CMFCApp theApp;  // 唯一的应用程序对象
```

将这段代码编译运行后（注意将编译方式调整为“使用静态的 MFC 库”），可以看到运行了一个窗体，双击窗体会弹出一个 MessageBox。

这里的代码要比使用 C++/SDK 要简单明了的多，但是我们没有发现熟悉的 WinMain，也没有窗口注册创建等等的部分。显然，这些都被 MFC 隐藏了。

2. CWinApp — 应用程序类

就像标题所说的，CWinApp 的作用是用来产生 MFC 应用程序对象。每个 MFC 程序都要求存在一个派生自 CWinApp 的子类作为应用程序的类，并且需要我们通过这个子类，改写某些虚函数。可见 CWinApp 的重要性，显然，CWinApp 是一个很大的类，他得完成很多工作。

以下是 MFC7.1 的 CWinApp 的定义源代码的超精简版本（完整版在 AFXWIN.H 中）：

```
class CWinApp : public CWinThread
{
  DECLARE_DYNAMIC(CWinApp)
public:

// Constructor
  /* explicit */ CWinApp(LPCTSTR lpszAppName = NULL);    // app name defaults to EXE name

// Attributes
  // Startup args (do not change)

  // This module's hInstance.
  HINSTANCE m_hInstance;

  // Pointer to the command-line.
  LPTSTR m_lpCmdLine;

  // Initial state of the application's window; normally,
```

```
    // this is an argument to ShowWindow().
    int m_nCmdShow;

    // Running args (can be changed in InitInstance)

    // Human-redable name of the application. Normally set in
    // constructor or retreived from AFX_IDS_APP_TITLE.
    LPCTSTR m_pszAppName;

    // Name of registry key for this application. See
    // SetRegistryKey() member function.
    LPCTSTR m_pszRegistryKey;

    // Pointer to CDocManager used to manage document templates
    // for this application instance.
    CDocManager* m_pDocManager;

    // Support for Shift+F1 help mode.

    // TRUE if we're in SHIFT+F1 mode.
    BOOL m_bHelpMode;

public:
    // set in constructor to override default

    // Executable name (no spaces).
    LPCTSTR m_pszExeName;

    // Default based on this module's path.
    LPCTSTR m_pszHelpFilePath;

    // Default based on this application's name.
    LPCTSTR m_pszProfileName;

// Overridables

    // Hooks for your initialization code
    virtual BOOL InitApplication();

public: // public for implementation access
    CCommandLineInfo* m_pCmdInfo;

    // overrides for implementation
    virtual BOOL InitInstance();
    virtual int ExitInstance(); // return app exit code
    virtual int Run();
    virtual BOOL OnIdle(LONG lCount); // return TRUE if more idle processing
```

```
  virtual LRESULT ProcessWndProcException(CException* e, const MSG* pMsg);
  virtual HINSTANCE LoadAppLangResourceDLL();

public:
  virtual ~CWinApp();
#ifdef _DEBUG
  virtual void AssertValid() const;
  virtual void Dump(CDumpContext& dc) const;
#endif

  // helpers for registration
  HKEY GetSectionKey(LPCTSTR lpszSection);
  HKEY GetAppRegistryKey();

protected:
  //{{AFX_MSG(CWinApp)
  afx_msg void OnAppExit();
  afx_msg void OnUpdateRecentFileMenu(CCmdUI* pCmdUI);
  afx_msg BOOL OnOpenRecentFile(UINT nID);
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
public :
  // System Policy Settings
  virtual BOOL LoadSysPolicies(); // Override to load policies other than the system policies that MFC loa
ds.
  BOOL GetSysPolicyValue(DWORD dwPolicyID, BOOL *pbValue); // returns the policy's setting in the out
parameter
protected :
  DWORD m_dwPolicies;       // block for storing boolean system policies
};
```

CWinApp 中全是成员函数和成员变量（废话- -!），我们重点来看看几个重要的函数和变量：

- CWinApp 中保存了一些传递给 WinMain()的命令行参数，这些参数包括当前实例的句柄（m_hInstance）、前一个实例的句柄（m_hPrevInstance）、命令行参数（m_lpCmdLine）以及控制窗口显示的标志（m_nCmdShow）。
- CWinApp 在 m_pszAppName 中保存了应用程序名字的拷贝。
- CWinApp 中保存了指向可执行文件名字的指针（m_pszExeName）、指向应用程序帮助文件路径的指针（m_pszHelpFilePath）以及指向其应用程序配置文件名字的指针（m_pszProfileName）。
- CWinApp 还是用了一个叫做 CCommandLineInfo 的 Class 来保存命令行参数。CCommandLineInfo 在一个地方保存了所有标准参数，他是一个很小却很实用的 Class。他在你需要得到的命令行参数的时候很有用。我们这里粗略的浏览下他的定义源代码（同样在 AFXWIN.H 中）：

```
class CCommandLineInfo : public CObject
{
public:
  // Sets default values
  CCommandLineInfo();
```

```
  // plain char* version on UNICODE for source-code backwards compatibility
  virtual void ParseParam(const TCHAR* pszParam, BOOL bFlag, BOOL bLast);
#ifdef _UNICODE
  virtual void ParseParam(const char* pszParam, BOOL bFlag, BOOL bLast);
#endif

  BOOL m_bShowSplash;
  BOOL m_bRunEmbedded;
  BOOL m_bRunAutomated;
  enum { FileNew, FileOpen, FilePrint, FilePrintTo, FileDDE, AppRegister,
    AppUnregister, FileNothing = -1 } m_nShellCommand;

  // not valid for FileNew
  CString m_strFileName;

  // valid only for FilePrintTo
  CString m_strPrinterName;
  CString m_strDriverName;
  CString m_strPortName;

  ~CCommandLineInfo();
// Implementation
protected:
  void ParseParamFlag(const char* pszParam);
  void ParseParamNotFlag(const TCHAR* pszParam);
#ifdef _UNICODE
  void ParseParamNotFlag(const char* pszParam);
#endif
  void ParseLast(BOOL bLast);
};
```

- 每个应用程序都需要进行针对实例的初始化。在 MFC 程序中，这部分功能由 CWinApp::InitInstance()完成。这个函数是虚函数，我们在派生类中必须重写这个函数。一般会在这个函数里面显示主窗口。
- 在 MFC 程序中，可以通过 CWinApp::ExitInstance()来完成关闭应用程序时候的资源清理工作。这个函数同样是虚函数
- 在 MFC 程序中，消息泵是 CWinApp 的一部分。调用 CWinApp::Run()会启动标准的 GetMessage()..DispatchMessage()循环。CWinApp 的消息还支持后台处理。
- 由于 Windows 使用事件驱动机制，所以应用程序运行期间队列中可能没有任何消息。MFC 此时会调用 CWinApp 的 OnIdle()函数。每当消息队列为空时，CWinApp::Run()都会调用 OnIdle()。

可能你会觉得这里缺少了什么东西，比如保存主窗口句柄的成员变量。事实上，的确有这个成员变量，变量名为 m_pMainWnd。如果你翻到前面的例子代码，你会发现这个变量的用途。

传说在很久之前的 MFC 版本，m_pMainWnd 被安排在了 CWinApp 里，但是经过几经周折，现在，这个成员变量升级了，跑到了 CWinThread 里去了（CWinThread 是 CWinApp 的父类）。CWinThread 的源代码也在 AFXWIN.H 中。

```
class CWinThread : public CCmdTarget
{
  DECLARE_DYNAMIC(CWinThread)

  friend BOOL AfxInternalPreTranslateMessage(MSG* pMsg);

public:
// Constructors
  CWinThread();
  BOOL CreateThread(DWORD dwCreateFlags = 0, UINT nStackSize = 0,
    LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL);

// Attributes
  // Look at Here !
  CWnd* m_pMainWnd;       // main window (usually same AfxGetApp()->m_pMainWnd)
  CWnd* m_pActiveWnd;     // active main window (may not be m_pMainWnd)
  BOOL m_bAutoDelete;     // enables 'delete this' after thread termination
  ......
}
```

3. CFrameWnd — 框架窗口类

我不想花费过多的时间在这个问题上。是的正如标题所说，CFrameWnd 只是 MFC 窗口类的一种。CFrameWnd 能提供 SDI、Overlapped、Pop-up 风格的窗口。

但是对于 MFC 来说，窗口的种类远不止这些。如果你打开 MFC 的类库，你会发现 MFC 的窗口类有：Frame Window、Dialog Boxes、Views、Controls。而能作为 Main Window 的窗口类也不止一种。（事实上，以 Dialog 作为 Main Window 的情况更加常见。）

不同的窗口类之间都有差别，但是他们都有共同的地方。是的，正如你所见，他们都继承自 CWnd。CWnd 是一个很巨大的类，他是所有窗口类的基类。由于继承的特性，子类都会强制保留一些属性。

虽然各个窗口类间的实现存在区别，但在本质上，他们都完成了非常类似的工作。所以，我们通过采取对 CFrameWnd 这一特殊情况进行研究，从而一般化窗口类的实现过程。（不过有时候，这个一般化过程会因为某些窗口的自身的特性而遭遇阻碍。）

CFrameWnd 的主要作用是创建一个窗体对象，并通过自身的消息映射表（Message Map），将消息映射到相应的消息处理函数中进行处理。这里 CFrameWnd 充当了 SDK 中回调函数的职责。

4. theApp — 唯一的应用程序对象

这里，我们先看看开头的例子代码，并把目光集中到

```
CMFCApp theApp;  // 唯一的应用程序对象
```

我们用派生自 CWinApp 的应用程序类 CMFCApp 创建了一个全局对象 theApp

theApp 就是我们的应用程序对象。一般的，每个 MFC 程序都有这个对象，而且有且只有一个。（通常用 MFC 向导生成的工程的应用程序对象名都是 theApp）

按照惯例，创建 theApp 时会先调用父类 CWinApp 的构造函数，然后调用 CMFCWinApp 的构造函数。但是由于我们没有定义子类的构造函数，所以只会执行 CWinApp 的构造函数。那么接下来，我们就通过源代码，看看 CWinApp 在构造函数里都干了什么事情（代码在 appcore.cpp 中）：

```
CWinApp::CWinApp(LPCTSTR lpszAppName)
{
 if (lpszAppName != NULL)
  m_pszAppName = _tcsdup(lpszAppName);
 else
  m_pszAppName = NULL;

 // initialize CWinThread state
 AFX_MODULE_STATE* pModuleState = _AFX_CMDTARGET_GETSTATE();
 AFX_MODULE_THREAD_STATE* pThreadState = pModuleState->m_thread;
 ASSERT(AfxGetThread() == NULL);
 pThreadState->m_pCurrentWinThread = this;
 ASSERT(AfxGetThread() == this);
 m_hThread = ::GetCurrentThread();
 m_nThreadID = ::GetCurrentThreadId();

 // initialize CWinApp state
 ASSERT(afxCurrentWinApp == NULL); // only one CWinApp object please
 pModuleState->m_pCurrentWinApp = this;
 ASSERT(AfxGetApp() == this);

 // in non-running state until WinMain
 m_hInstance = NULL;
 m_hLangResourceDLL = NULL;
 m_pszHelpFilePath = NULL;
 m_pszProfileName = NULL;
 m_pszRegistryKey = NULL;
 m_pszExeName = NULL;
 m_pRecentFileList = NULL;
 m_pDocManager = NULL;
 m_atomApp = m_atomSystemTopic = NULL;
 m_lpCmdLine = NULL;
 m_pCmdInfo = NULL;

 // initialize wait cursor state
 m_nWaitCursorCount = 0;
 m_hcurWaitCursorRestore = NULL;
```

```
    // initialize current printer state
    m_hDevMode = NULL;
    m_hDevNames = NULL;
    m_nNumPreviewPages = 0;    // not specified (defaults to 1)

    // initialize DAO state
    m_lpfnDaoTerm = NULL;   // will be set if AfxDaoInit called

    // other initialization
    m_bHelpMode = FALSE;
    m_eHelpType = afxWinHelp;
    m_nSafetyPoolSize = 512;       // default size
}
```

CWinApp 的构造函数主要工作就是初始化 CWinApp 的成员变量。

构造函数首先通过参数 lpszAppName 设置 m_pszAppName 的值。参数默认为 NULL，你也可以传入应用程序的名字。

然后通过

```
    AFX_MODULE_STATE* pModuleState = _AFX_CMDTARGET_GETSTATE();
    AFX_MODULE_THREAD_STATE* pThreadState = pModuleState->m_thread;
    ASSERT(AfxGetThread() == NULL);
    pThreadState->m_pCurrentWinThread = this;
    ASSERT(AfxGetThread() == this);
    m_hThread = ::GetCurrentThread();
    m_nThreadID = ::GetCurrentThreadId();
```

初始化线程状态和模块状态。因为线程状态要牵扯到 MFC 的线程运作，这里不作介绍。AFX_MODULE_STATE 之后进行介绍。

紧接着，构造函数会初始化 CWinApp 的状态

```
    ASSERT(afxCurrentWinApp == NULL); // only one CWinApp object please
    pModuleState->m_pCurrentWinApp = this;
    ASSERT(AfxGetApp() == this);
```

这里第一、二行和 AFX_MODULE_STATE 有关，我们待会再讲。

这里的 AfxGetApp()返回与程序相关的应用程序对象(每一个 MFC 程序都需要一个 CWinApp 的派生对象)，用于检查对象是否正确。

然后，CWinApp 将所有其他成员（句柄、指针和字符串等）都设置成 NULL。

4.1. AFX_MODULE_STATE — MFC 状态信息

对于一个应用程序来说，在运行期间保存某些信息是应该的（或许也是必须的）。比如，SDK 程序会将实例句柄保存为全局变量。而程序就可以通过这个实例句柄获得 EXE 文件的资源。MFC 也酱紫做，不过，它会通过 Class，来保存各种资源。

和 SDK 程序相比，MFC 除了实例句柄之外，还有其他很多需要保存的信息。可能会包括窗口、资源、模块句柄。而 MFC 能为应用程序提供内存分配的跟踪（这个 MS 很好很强大）、ODBC 支持、OLE 支持和异常处理等特性，所以 MFC 程序会比一般的 SDK 程序维护更多的信息。

于是，MFC 定义了一个叫做 AFX_MODULE_STATE 的 Class，我们先来看看他的定义源代码（在 AFXSTAT_.H 中）：

```
class AFX_MODULE_STATE : public CNoTrackObject
{
public:
#ifdef _AFXDLL
  AFX_MODULE_STATE(BOOL bDLL, WNDPROC pfnAfxWndProc, DWORD dwVersion,
    BOOL bSystem = FALSE);
#else
  explicit AFX_MODULE_STATE(BOOL bDLL);
#endif
  ~AFX_MODULE_STATE();

  CWinApp* m_pCurrentWinApp;
  HINSTANCE m_hCurrentInstanceHandle;
  HINSTANCE m_hCurrentResourceHandle;
  LPCTSTR m_lpszCurrentAppName;
  BYTE m_bDLL;    // TRUE if module is a DLL, FALSE if it is an EXE
  BYTE m_bSystem; // TRUE if module is a "system" module, FALSE if not
  BYTE m_bReserved[2]; // padding

  DWORD m_fRegisteredClasses; // flags for registered window classes

  // runtime class data
#ifdef _AFXDLL
  CRuntimeClass* m_pClassInit;
#endif
  CTypedSimpleList<CRuntimeClass*> m_classList;

  // OLE object factories
#ifndef _AFX_NO_OLE_SUPPORT
#ifdef _AFXDLL
  COleObjectFactory* m_pFactoryInit;
#endif
  CTypedSimpleList<COleObjectFactory*> m_factoryList;
#endif
  // number of locked OLE objects
  long m_nObjectCount;
  BOOL m_bUserCtrl;
```

```
  // AfxRegisterClass and AfxRegisterWndClass data
  TCHAR m_szUnregisterList[4096];
#ifdef _AFXDLL
  WNDPROC m_pfnAfxWndProc;
  DWORD m_dwVersion;  // version that module linked against
#endif

  // variables related to a given process in a module
  //  (used to be AFX_MODULE_PROCESS_STATE)
  void (PASCAL *m_pfnFilterToolTipMessage)(MSG*, CWnd*);

#ifdef _AFXDLL
  // CDynLinkLibrary objects (for resource chain)
  CTypedSimpleList<CDynLinkLibrary*> m_libraryList;

  // special case for MFC71XXX.DLL (localized MFC resources)
  HINSTANCE m_appLangDLL;
#endif

#ifndef _AFX_NO_OCC_SUPPORT
  // OLE control container manager
  COccManager* m_pOccManager;
  // locked OLE controls
  CTypedSimpleList<COleControlLock*> m_lockList;
#endif

#ifndef _AFX_NO_DAO_SUPPORT
  _AFX_DAO_STATE* m_pDaoState;
#endif

#ifndef _AFX_NO_OLE_SUPPORT
  // Type library caches
  CTypeLibCache m_typeLibCache;
  CTypeLibCacheMap* m_pTypeLibCacheMap;
#endif

  // define thread local portions of module state
  CThreadLocal<AFX_MODULE_THREAD_STATE> m_thread;
};
```

首先我们会发现一个很有意思的情况：所有的成员都是公有的，这样就可以直接访问成员变量而不需通过成员函数。

然后，我们来看下 AFX_MODULE_STATE 中一些重要的成员变量：

```
  CWinApp* m_pCurrentWinApp;
```

```
HINSTANCE m_hCurrentInstanceHandle;
HINSTANCE m_hCurrentResourceHandle;
LPCTSTR m_lpszCurrentAppName;
BYTE m_bDLL;    // TRUE if module is a DLL, FALSE if it is an EXE
BYTE m_bSystem; // TRUE if module is a "system" module, FALSE if not
DWORD m_fRegisteredClasses; // flags for registered window classes
// runtime class data
m_classList;
m_factoryList
// number of locked OLE objects
long m_nObjectCount;
BOOL m_bUserCtrl;
// AfxRegisterClass and AfxRegisterWndClass data
m_szUnregisterList[4096];
m_pfnAfxWndProc
```

m_pCurrentWinApp 指向 CWinApp 的指针。现在再回头看 CWinApp 的构造函数，我们使用了下面的代码 pModuleState->m_pCurrentWinApp = this; 将 m_pCurrentWinApp 初始化为正在构造的 CWinApp 对象。

而 ASSERT(afxCurrentWinApp == NULL);则可以检查是否已经存在一个应用程序对象，从而保证了 theApp 的唯一性。（afxCurrentWinApp 是一个宏，用来返回 m_pCurrentWinApp）

m_hCurrentInstanceHandle 该模块实例的句柄。别用这个来获得本地资源。

m_hCurrentResourceHandle 所保存模块资源的句柄。可以使用这个来获得资源，有助于你的东西走向国际市场- -!

m_lpszCurrentAppName 指向应用程序名字的指针。

m_bDLL 表示模块是一个 DLL 还是 EXE。

m_bSystem 表示模块是系统模块还是本地模块。

m_fRegisteredClasses 指示哪些 MFC 窗口类已经注册。（关于 MFC 注册窗口类，我们会在后面说到）

m_classList 指向 MFC 程序 CRuntimeClass 结构链表的第一个运行时类的指针。

m_factoryList 指向 MFC 程序 COleObjectFactory 结构链表中的一个运行时类的指针。

m_nObjectCount OLE 服务器的引用计数。会被用来检查是否有未完成的 COM 对象。

m_bUserCtrl 表示是否在使用 OLE。

m_szUnregisterList[4096] 用于维护 MFC 已注册窗口类的链表，以便于 MFC 在这些类结束时可以将他们注销。

m_pfnAfxWndProc MFC 窗口过程的函数指针。

现在我们已经了解了 MFC 中保存的状态信息，现在我们继续探索 MFC 之旅。

5. 找回 WinMain()

theApp 创建完毕之后，我们就要开始寻找 WinMain()了。对于一个 Windows 应用程序来说，必定存在一个 WinMain()。而我们的 MFC 程序也能正常运行，所以 WinMain 肯定被埋藏在了某个地方（事实上，MFC 会在编译链接的时候直接加到应用程序代码中）。

现在，我们来看看 MFC 中包含的 WinMain()（代码在 APPMODUL.CPP 中）

```
// export WinMain to force linkage to this module
extern int AFXAPI AfxWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
 LPTSTR lpCmdLine, int nCmdShow);

extern "C" int WINAPI
_tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
 LPTSTR lpCmdLine, int nCmdShow)
{
 // call shared/exported WinMain
 return AfxWinMain(hInstance, hPrevInstance, lpCmdLine, nCmdShow);
}
```

我们可以看到，WinMain()把处理交给了 AfxWinMain()的函数。而且这个函数的参数和 WinMain()一模一样。

顺便说下，这里的 AFXAPI 和以前的 WINAPI 一样，都是调用约定。

我们现在就来看看 AfxWinMain()的源代码，看看他都做了什么事情（代码在 WINMAIN.CPP）：

```
#ifdef AFX_CORE1_SEG
#pragma code_seg(AFX_CORE1_SEG)
#endif

////////////////////////////////////////////////////////////////////////////
// Standard WinMain implementation
//  Can be replaced as long as 'AfxWinInit' is called first

int AFXAPI AfxWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
 LPTSTR lpCmdLine, int nCmdShow)
{
 ASSERT(hPrevInstance == NULL);

 int nReturnCode = -1;
 CWinThread* pThread = AfxGetThread();
 CWinApp* pApp = AfxGetApp();
```

```
  // AFX internal initialization
  if (!AfxWinInit(hInstance, hPrevInstance, lpCmdLine, nCmdShow))
    goto InitFailure;

  // App global initializations (rare)
  if (pApp != NULL && !pApp->InitApplication())
    goto InitFailure;

  // Perform specific initializations
  if (!pThread->InitInstance())
  {
    if (pThread->m_pMainWnd != NULL)
    {
      TRACE(traceAppMsg, 0, "Warning: Destroying non-NULL m_pMainWnd\n");
      pThread->m_pMainWnd->DestroyWindow();
    }
    nReturnCode = pThread->ExitInstance();
    goto InitFailure;
  }
  nReturnCode = pThread->Run();

InitFailure:
#ifdef _DEBUG
  // Check for missing AfxLockTempMap calls
  if (AfxGetModuleThreadState()->m_nTempMapLock != 0)
  {
    TRACE(traceAppMsg, 0, "Warning: Temp map lock count non-zero (%ld).\n",
      AfxGetModuleThreadState()->m_nTempMapLock);
  }
  AfxLockTempMaps();
  AfxUnlockTempMaps(-1);
#endif

  AfxWinTerm();
  return nReturnCode;
}
```

我们先看 CWinThread* pThread = AfxGetThread(); 和 CWinApp* pApp = AfxGetApp();。

前者返回一个代表当前执行线程的指针（因为牵扯到 MFC 的线程运作，所以我们这里不仔细研究）。

后者我们在研究 CWinApp 的构造函数时已经碰到了，他会返回指向当前程序的应用程序对象指针。但是不知道大家会不会有个疑问：如果 WinMain()仍然是 MFC 程序的函数入口点，而应用程序又不是在 WinMain()中所创建的，那么如何返回应用程序对象？

其实道理很简单，因为我们的 theApp 是全局对象。而 C++程序在做任何事情之前（不知道是不是绝对了

点），甚至是进入 main()或 WinMain()之前，首先创建全局对象。这样就可以保证进入 WinMain()的时候，应用程序对象已经创建。

然后我们先暂时跨过 AfxWinInit()，先来研究几个重要的“表演”。

```
CWinThread* pThread = AfxGetThread();
CWinApp* pApp = AfxGetApp();
pApp->InitApplication();
pThread->InitInstance();
pThread->Run();
```

上面的代码类似于：

```
CMFCApp::InitApplication();
CMFCApp::Instance();
CMFCApp::Run();
```

因为我们有一个派生自 CWinApp 的类 CMFCApp，而由于多态特性的存在（上面的几个函数都是虚函数），上面的代码会导致下面情况的发生：

```
CWinApp::InitApplication();   // 未改写这个虚函数
CMFCApp::Instance();  // 因为我们改写了这个虚函数
CWinApp::Run();   // 未改写这个虚函数
```

PS：其实在 MFC7.X 的版本中，InitInstance、Run、ExitInstance 和 OnIdle 成员函数实际位于 CWinThread 类中。而 CWinApp 继承了 CWinThread 的这些函数，并且也部分重写了这几个虚函数，所以当我们没有重写虚函数时，C++就会往继承树上翻，便会执行 CWinApp 中的几个虚函数。

6. AfxWinInit() — 初始化框架

刚才我们跳过了 AfxWinInit()函数，现在我们回过头来看看。

AfxWinInit()函数由 MFC 提供的 WinMain()函数调用，用于初始化 MFC 框架。我们通过 AfxWinInit()函数的源代码，来看看这个函数都做了哪些事（源代码在 APPINIT.CPP）：

```
BOOL AFXAPI AfxWinInit(HINSTANCE hInstance, HINSTANCE hPrevInstance,
  LPTSTR lpCmdLine, int nCmdShow)
{
  ASSERT(hPrevInstance == NULL);


  // handle critical errors and avoid Windows message boxes
  SetErrorMode(SetErrorMode(0) |
    SEM_FAILCRITICALERRORS|SEM_NOOPENFILEERRORBOX);

  // set resource handles
```

```
 AFX_MODULE_STATE* pModuleState = AfxGetModuleState();
 pModuleState->m_hCurrentInstanceHandle = hInstance;
 pModuleState->m_hCurrentResourceHandle = hInstance;

 // fill in the initial state for the application
 CWinApp* pApp = AfxGetApp();
 if (pApp != NULL)
 {
  // Windows specific initialization (not done if no CWinApp)
   pApp->m_hInstance = hInstance;
  hPrevInstance; // Obsolete.
   pApp->m_lpCmdLine = lpCmdLine;
  pApp->m_nCmdShow = nCmdShow;
  pApp->SetCurrentHandles();
 }

 // initialize thread specific data (for main thread)
 if (!afxContextIsDLL)
  AfxInitThread();

 // Initialize CWnd::m_pfnNotifyWinEvent
 HMODULE hModule = ::GetModuleHandle(_T("user32.dll"));
 if (hModule != NULL)
 {
  CWnd::m_pfnNotifyWinEvent = (CWnd::PFNNOTIFYWINEVENT)::GetProcAddress(hModule, "NotifyWinEvent"
);
  }

 return TRUE;
}
```

AfxWinInit()的 4 个参数和 WinMain()所带的一样。

AfxWinInit()首先会调用 SetErrorMode()来为应用程序设置错误模式，用于指明导致程序失败的原因。

SetErrorMode()的参数只有一个：指定错误模式。MFC 使用 SEM_FAILCRITICALERRORS 和 SEM_NOOPENFILEERRORBOX 设置错误模式。前者用于通知窗口对于关键错误（Critical-Error）不要显示关键错误处理消息框（the critical-error-handler message box），而是将错误传回调用的进程。后者则告诉窗体在没有找到文件时不要显示消息框，而是将错误传回调用的进程。

PS：关于 SetErrorMode()的详细信息请看附录

接下来，我们看这里

```
 AFX_MODULE_STATE* pModuleState = AfxGetModuleState();
 pModuleState->m_hCurrentInstanceHandle = hInstance;
 pModuleState->m_hCurrentResourceHandle = hInstance;
```

AfxWinInit()会调用 AfxGetModuleState()来得到 AFX_MODULE_STATE。

AfxWinInit()将模块的实例句柄和资源句柄存放在 AFX_MODULE_STATE:: m_hCurrentInstanceHandle 和 AFX_MODULE_STATE:: m_hCurrentResourceHandle 中。

然后我们看这里：

```
CWinApp* pApp = AfxGetApp();
if (pApp != NULL)
{
  // Windows specific initialization (not done if no CWinApp)
   pApp->m_hInstance = hInstance;
  hPrevInstance; // Obsolete.
   pApp->m_lpCmdLine = lpCmdLine;
  pApp->m_nCmdShow = nCmdShow;
  pApp->SetCurrentHandles();
}
```

这里创建了一个指向当前应用程序对象的 CWinApp 指针，然后设置 Windows 传递给程序的 4 个参数。然后最有意思的是最后一句 pApp->SetCurrentHandles();。

我们先来看看这个函数的源代码（同样在 APPINIT.CPP 中）：

```
void CWinApp::SetCurrentHandles()
{
 ASSERT(this == afxCurrentWinApp);
 ASSERT(afxCurrentAppName == NULL);

 AFX_MODULE_STATE* pModuleState = _AFX_CMDTARGET_GETSTATE();
 pModuleState->m_hCurrentInstanceHandle = m_hInstance;
 pModuleState->m_hCurrentResourceHandle = m_hInstance;

 // Note: there are a number of _tcsdup (aka strdup) calls that are
 // made here for the exe path, help file path, etc.  In previous
 // versions of MFC, this memory was never freed.  In this and future
 // versions this memory is automatically freed during CWinApp's
 // destructor.  If you are freeing the memory yourself, you should
 // either remove the code or set the pointers to NULL after freeing
 // the memory.

 // get path of executable
 TCHAR szBuff[_MAX_PATH];
 DWORD dwRet = ::GetModuleFileName(m_hInstance, szBuff, _MAX_PATH);
 ASSERT( dwRet != 0 && dwRet != _MAX_PATH );
 if( dwRet == 0 || dwRet == _MAX_PATH )
  AfxThrowUserException();
```

```
LPTSTR lpszExt = ::PathFindExtension(szBuff);
ASSERT(lpszExt != NULL);
if( lpszExt == NULL )
  AfxThrowUserException();

ASSERT(*lpszExt == '.');
*lpszExt = 0;       // no suffix

 TCHAR szExeName[_MAX_PATH];
TCHAR szTitle[256];
// get the exe title from the full path name [no extension]
 dwRet = AfxGetFileName(szBuff, szExeName, _MAX_PATH);
ASSERT( dwRet == 0 );
if( dwRet != 0 )
  AfxThrowUserException();

if (m_pszExeName == NULL)
{
  BOOL bEnable = AfxEnableMemoryTracking(FALSE);
  m_pszExeName = _tcsdup(szExeName); // save non-localized name
   AfxEnableMemoryTracking(bEnable);
}

// m_pszAppName is the name used to present to the user
if (m_pszAppName == NULL)
{
  BOOL bEnable = AfxEnableMemoryTracking(FALSE);
  if (AfxLoadString(AFX_IDS_APP_TITLE, szTitle) != 0)
    m_pszAppName = _tcsdup(szTitle);    // human readable title
   else
    m_pszAppName = _tcsdup(m_pszExeName);   // same as EXE
   AfxEnableMemoryTracking(bEnable);
}

pModuleState->m_lpszCurrentAppName = m_pszAppName;
ASSERT(afxCurrentAppName != NULL);

// get path of .HLP file or .CHM (HtmlHelp) file
if (m_pszHelpFilePath == NULL)
{
  if (m_eHelpType == afxHTMLHelp)
    lstrcpy(lpszExt, _T(".CHM"));
  else
    lstrcpy(lpszExt, _T(".HLP"));
  BOOL bEnable = AfxEnableMemoryTracking(FALSE);
  m_pszHelpFilePath = _tcsdup(szBuff);
  AfxEnableMemoryTracking(bEnable);
```

```
    *lpszExt = '\0';      // back to no suffix
   }

  if (m_pszProfileName == NULL)
  {
    lstrcat(szExeName, _T(".INI")); // will be enough room in buffer
     BOOL bEnable = AfxEnableMemoryTracking(FALSE);
    m_pszProfileName = _tcsdup(szExeName);
    AfxEnableMemoryTracking(bEnable);
  }
}
```

在这个函数中，我们会发现很多有意思的地方。我们先来看这里：

```
AFX_MODULE_STATE* pModuleState = _AFX_CMDTARGET_GETSTATE();
pModuleState->m_hCurrentInstanceHandle = m_hInstance;
pModuleState->m_hCurrentResourceHandle = m_hInstance;
```

SetCurrentHandles()函数中再次设置了 AFX_MODULE_STATE 句柄。你可能会觉得这个有点多余，实际上这是一个历史包袱- -!，在很久之前的 MFC 版本中，是没有模块状态结构这个东西的。后来在解决 MFC 中的 OLE 控件时才引入这个结构。

所以，很多事情都很难说。以后怎么样，谁也不知道。兴许哪天上校我走在马路上就被车撞成了真正的 Son-Of-Darkness……咳咳~MS 有点跑题了—。—。

```
  TCHAR szBuff[_MAX_PATH];
  DWORD dwRet = ::GetModuleFileName(m_hInstance, szBuff, _MAX_PATH);
  ASSERT( dwRet != 0 && dwRet != _MAX_PATH );
  if( dwRet == 0 || dwRet == _MAX_PATH )
    AfxThrowUserException();

  LPTSTR lpszExt = ::PathFindExtension(szBuff);
  ASSERT(lpszExt != NULL);
  if( lpszExt == NULL )
    AfxThrowUserException();

  ASSERT(*lpszExt == '.');
  *lpszExt = 0;       // no suffix

  TCHAR szExeName[_MAX_PATH];
  TCHAR szTitle[256];
  // get the exe title from the full path name [no extension]
  dwRet = AfxGetFileName(szBuff, szExeName, _MAX_PATH);
  ASSERT( dwRet == 0 );
  if( dwRet != 0 )
    AfxThrowUserException();
```

```
if (m_pszExeName == NULL)
{
  BOOL bEnable = AfxEnableMemoryTracking(FALSE);
  m_pszExeName = _tcsdup(szExeName); // save non-localized name
   AfxEnableMemoryTracking(bEnable);
}
```

这部分主要用于初始化应用程序的名字和路径。

PS：这里不要忘了前面的 SetErrorMode()的作用~

首先用 GetModuleFileName 返回模块的运行实例的全名，然后对函数的返回值进行检查。AfxThrowUserException 会抛出一个异常，然后结束用户的操作。

然后用 PathFindExtension 返回文件的路径（“.”之前的路径），同样对返回值进行检查。（后面的那个 AfxGetFileName 用的也是这个 API，AfxGetFileName 源代码就在 SetCurrentHandles()的下面）

取得合法的可执行文件名字后（不包括.exe），赋值给 CWinApp::m_pszExeName。而 AfxEnableMemoryTracking 用于诊断内存检测。

PS：GetModuleFileName、PathFindExtension、AfxThrowUserException 和 AfxEnableMemoryTracking 的详细信息请查看 GetModuleFileName 附录

然后看这里：

```
if (m_pszAppName == NULL)
 {
  BOOL bEnable = AfxEnableMemoryTracking(FALSE);
  if (AfxLoadString(AFX_IDS_APP_TITLE, szTitle) != 0)
    m_pszAppName = _tcsdup(szTitle);   // human readable title
   else
    m_pszAppName = _tcsdup(m_pszExeName);  // same as EXE
   AfxEnableMemoryTracking(bEnable);
 }

 pModuleState->m_lpszCurrentAppName = m_pszAppName;
 ASSERT(afxCurrentAppName != NULL);
```

SetCurrentHandles()设置完 CWinApp::m_pszExeName 之后，将 m_pszAppName 初始化为应用程序的标题。如果你使用 MFC Wizard 来创建工程，那么 MFC 会使用工程中的资源文件中特定的字符串 ID 来代替 AFX_IDS_APP_TITLE。

如果找不到资源，SetCurrentHandles()就会用 m_pszExeName 来初始化 m_pszAppName。

然后 SetCurrentHandles()将 AFX_MODULE_STATE:: m_lpszCurrentAppName 设置为 m_pszAppName。

```
  // get path of .HLP file or .CHM (HtmlHelp) file
```

```
  if (m_pszHelpFilePath == NULL)
  {
   if (m_eHelpType == afxHTMLHelp)
    lstrcpy(lpszExt, _T(".CHM"));
   else
    lstrcpy(lpszExt, _T(".HLP"));
   BOOL bEnable = AfxEnableMemoryTracking(FALSE);
   m_pszHelpFilePath = _tcsdup(szBuff);
   AfxEnableMemoryTracking(bEnable);
   *lpszExt = '\0';       // back to no suffix
  }

  if (m_pszProfileName == NULL)
  {
   lstrcat(szExeName, _T(".INI")); // will be enough room in buffer
    BOOL bEnable = AfxEnableMemoryTracking(FALSE);
   m_pszProfileName = _tcsdup(szExeName);
   AfxEnableMemoryTracking(bEnable);
  }
}
```

最后，SetCurrentHandles()还会设置 CWinApp 的帮助文件和 Profile 字符串。

m_pszHelpFilePath 会被初始化为 GetModuleFileName()返回的值，并以 CHM 或 HLP 作为扩展名。

m_pszProfileName 会被初始化为应用程序的名字，并且以 INI 作为扩展名。

看完 SetCurrentHandles()，我们在回过头看 AfxWinInit()的最后一部分代码：

```
  // initialize thread specific data (for main thread)
  if (!afxContextIsDLL)
   AfxInitThread();

  // Initialize CWnd::m_pfnNotifyWinEvent
  HMODULE hModule = ::GetModuleHandle(_T("user32.dll"));
  if (hModule != NULL)
  {
   CWnd::m_pfnNotifyWinEvent = (CWnd::PFNNOTIFYWINEVENT)::GetProcAddress(hModule, "NotifyWinEvent"
);
  }

  return TRUE;
}
```

AfxWinInit()最后会初始化线程（这部分照例跳过~），然后初始化 CWnd::m_pfnNotifyWinEvent。这个成员变量看起来 MS 是系统的消息事件。

GetProcAddress 返回 DLL 中导出函数的地址。

如此这般，应用程序和框架就完全初始化了。

句柄和文件名都正确的被初始化，然后 AfxWinMain()会调用应用程序的 InitApplication()函数~

7. InitApplication — 逝去的光环

其实不应该讲这个函数。

如果你用的是 MFC7.X，当你在 CWinApp 中查询“Override Functions”时，你会发现这个函数消失了。

是的，这个函数已经消失了，在新版的 MFC 中，这个函数已经被废除。这个函数是在 Win16 下使用的，现在到了 Win32，这个函数已经没用了，他仅仅完成一些过渡性的工作。

所有的初始化动作都在 InitInstance()中完成，所以这个函数，剩下的也只是种种过往~

MSDN 对这位英雄人物的归隐是这样描述的：

> CWinApp::InitApplication
>
> The CWinApp::InitApplication member function is obsolete in MFC.
> Remarks
> An initialization that you would have done in InitApplication should be moved to InitInstance. If you override CWinApp::InitApplication, and you do not call the base class function, you will leak the CDocTemplate objects that were added through CWinApp::AddDocTemplate.

叶落归根，逝者如斯~这不能不说是一个悲剧~

8. InitInstance — 窗体产生的地方

IniApplication 之后，AfxWinMain()会便调用 InitInstance 函数，这个函数是个虚函数，而且我们必须在我们的派生类里改写这个函数，因为 CWinApp 的 InitInstance 没有任何创建窗体的行为。而一般我们都会在改写的这个函数里创建主窗体，并且设置 CWinThread::m_pMainWnd 变量指向主窗体。

因为是调用我们自己的 InitInstance()函数，所以我们回过头去看看例子代码中都在这个函数里干了什么事情：

```
BOOL CMFCApp::InitInstance()
{
 m_pMainWnd = new CMFCAppWindow();
 m_pMainWnd->ShowWindow(m_nCmdShow);
 m_pMainWnd->UpdateWindow();

 return TRUE;
```

```
}
```

这里，我们把 m_pMainWnd 指向了分配的 CMFCAppWindow 类型空间。很明显，这回调用 CMFCAppWindow 的构造函数，我们看下这个类的构造函数：

```
public:
 CMFCAppWindow()  // 在构造函数里创建窗体~
 {
  Create(NULL,"KC's Windows");  // 除前两个参数外,其他参数均有初始值
 }
```

我们在构造函数里用 Create 创建了一个窗体。关于 CFrameWnd::Create()我不想在这个上面花太多功夫。

这个函数除了第一、二个参数之外，其他参数均有 NULL 缺省值。第一个参数指定 WNDCLASS 类名（如果你了解 SDK 写法，应该对这个不陌生），如果为 NULL，则使用默认的 CFrameWnd 属性来初始化；第二个指定窗体标题。

PS：关于这个函数的详细信息，请参考附录

创建完之后，很类似的，我们使用了 ShowWindow()和 UpdateWindow()来显示并更新窗体。

9. 一些秘密

9.1. 注册窗口类

在 Windows 应用程序显示窗口之前，应用程序必须至少注册一个窗口类。而 MFC 程序也和普通的 Windows 程序一样，所以他也要至少注册一个窗口类。

首先，MFC 在 AFXIMPL.H 中定义了一个 AfxDeferRegisterClass()的宏：

```
#define AfxDeferRegisterClass(fClass) AfxEndDeferRegisterClass(fClass)
```

为了深入了解，我们来看下 AfxEndDeferRegisterClass 的源代码（在 WINCORE.CPP 中）：

```
BOOL AFXAPI AfxEndDeferRegisterClass(LONG fToRegister)
{
 // mask off all classes that are already registered
 AFX_MODULE_STATE* pModuleState = AfxGetModuleState();
 fToRegister &= ~pModuleState->m_fRegisteredClasses;
 if (fToRegister == 0)
  return TRUE;

 LONG fRegisteredClasses = 0;

 // common initialization
 WNDCLASS wndcls;
```

```
memset(&wndcls, 0, sizeof(WNDCLASS));   // start with NULL defaults
wndcls.lpfnWndProc = DefWindowProc;
wndcls.hInstance = AfxGetInstanceHandle();
wndcls.hCursor = afxData.hcurArrow;

INITCOMMONCONTROLSEX init;
init.dwSize = sizeof(init);

// work to register classes as specified by fToRegister, populate fRegisteredClasses as we go
if (fToRegister & AFX_WND_REG)
{
  // Child windows - no brush, no icon, safest default class styles
   wndcls.style = CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW;
  wndcls.lpszClassName = _afxWnd;
  if (AfxRegisterClass(&wndcls))
    fRegisteredClasses |= AFX_WND_REG;
}
if (fToRegister & AFX_WNDOLECONTROL_REG)
{
  // OLE Control windows - use parent DC for speed
   wndcls.style |= CS_PARENTDC | CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW;
  wndcls.lpszClassName = _afxWndOleControl;
  if (AfxRegisterClass(&wndcls))
    fRegisteredClasses |= AFX_WNDOLECONTROL_REG;
}
if (fToRegister & AFX_WNDCONTROLBAR_REG)
{
  // Control bar windows
   wndcls.style = 0;   // control bars don't handle double click
   wndcls.lpszClassName = _afxWndControlBar;
  wndcls.hbrBackground = (HBRUSH)(COLOR_BTNFACE + 1);
  if (AfxRegisterClass(&wndcls))
    fRegisteredClasses |= AFX_WNDCONTROLBAR_REG;
}
if (fToRegister & AFX_WNDMDIFRAME_REG)
{
  // MDI Frame window (also used for splitter window)
   wndcls.style = CS_DBLCLKS;
  wndcls.hbrBackground = NULL;
  if (_AfxRegisterWithIcon(&wndcls, _afxWndMDIFrame, AFX_IDI_STD_MDIFRAME))
    fRegisteredClasses |= AFX_WNDMDIFRAME_REG;
}
if (fToRegister & AFX_WNDFRAMEORVIEW_REG)
{
  // SDI Frame or MDI Child windows or views - normal colors
   wndcls.style = CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW;
  wndcls.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
  if (_AfxRegisterWithIcon(&wndcls, _afxWndFrameOrView, AFX_IDI_STD_FRAME))
```

```
    fRegisteredClasses |= AFX_WNDFRAMEORVIEW_REG;
  }
  if (fToRegister & AFX_WNDCOMMCTLS_REG)
  {
    // this flag is compatible with the old InitCommonControls() API
    init.dwICC = ICC_WIN95_CLASSES;
    fRegisteredClasses |= _AfxInitCommonControls(&init, AFX_WIN95CTLS_MASK);
    fToRegister &= ~AFX_WIN95CTLS_MASK;
  }
  if (fToRegister & AFX_WNDCOMMCTL_UPDOWN_REG)
  {
    init.dwICC = ICC_UPDOWN_CLASS;
    fRegisteredClasses |= _AfxInitCommonControls(&init, AFX_WNDCOMMCTL_UPDOWN_REG);
  }
  if (fToRegister & AFX_WNDCOMMCTL_TREEVIEW_REG)
  {
    init.dwICC = ICC_TREEVIEW_CLASSES;
    fRegisteredClasses |= _AfxInitCommonControls(&init, AFX_WNDCOMMCTL_TREEVIEW_REG);
  }
  if (fToRegister & AFX_WNDCOMMCTL_TAB_REG)
  {
    init.dwICC = ICC_TAB_CLASSES;
    fRegisteredClasses |= _AfxInitCommonControls(&init, AFX_WNDCOMMCTL_TAB_REG);
  }
  if (fToRegister & AFX_WNDCOMMCTL_PROGRESS_REG)
  {
    init.dwICC = ICC_PROGRESS_CLASS;
    fRegisteredClasses |= _AfxInitCommonControls(&init, AFX_WNDCOMMCTL_PROGRESS_REG);
  }
  if (fToRegister & AFX_WNDCOMMCTL_LISTVIEW_REG)
  {
    init.dwICC = ICC_LISTVIEW_CLASSES;
    fRegisteredClasses |= _AfxInitCommonControls(&init, AFX_WNDCOMMCTL_LISTVIEW_REG);
  }
  if (fToRegister & AFX_WNDCOMMCTL_HOTKEY_REG)
  {
    init.dwICC = ICC_HOTKEY_CLASS;
    fRegisteredClasses |= _AfxInitCommonControls(&init, AFX_WNDCOMMCTL_HOTKEY_REG);
  }
  if (fToRegister & AFX_WNDCOMMCTL_BAR_REG)
  {
    init.dwICC = ICC_BAR_CLASSES;
    fRegisteredClasses |= _AfxInitCommonControls(&init, AFX_WNDCOMMCTL_BAR_REG);
  }
  if (fToRegister & AFX_WNDCOMMCTL_ANIMATE_REG)
  {
    init.dwICC = ICC_ANIMATE_CLASS;
    fRegisteredClasses |= _AfxInitCommonControls(&init, AFX_WNDCOMMCTL_ANIMATE_REG);
```

```
 }
 if (fToRegister & AFX_WNDCOMMCTL_INTERNET_REG)
 {
  init.dwICC = ICC_INTERNET_CLASSES;
  fRegisteredClasses |= _AfxInitCommonControls(&init, AFX_WNDCOMMCTL_INTERNET_REG);
 }
 if (fToRegister & AFX_WNDCOMMCTL_COOL_REG)
 {
  init.dwICC = ICC_COOL_CLASSES;
  fRegisteredClasses |= _AfxInitCommonControls(&init, AFX_WNDCOMMCTL_COOL_REG);
 }
 if (fToRegister & AFX_WNDCOMMCTL_USEREX_REG)
 {
  init.dwICC = ICC_USEREX_CLASSES;
  fRegisteredClasses |= _AfxInitCommonControls(&init, AFX_WNDCOMMCTL_USEREX_REG);
 }
 if (fToRegister & AFX_WNDCOMMCTL_DATE_REG)
 {
  init.dwICC = ICC_DATE_CLASSES;
  fRegisteredClasses |= _AfxInitCommonControls(&init, AFX_WNDCOMMCTL_DATE_REG);
 }
 if (fToRegister & AFX_WNDCOMMCTL_LINK_REG)
 {
  init.dwICC = ICC_LINK_CLASS;
  fRegisteredClasses |= _AfxInitCommonControls(&init, AFX_WNDCOMMCTL_LINK_REG);
 }


 // save new state of registered controls
 pModuleState->m_fRegisteredClasses |= fRegisteredClasses;

 // special case for all common controls registered, turn on AFX_WNDCOMMCTLS_REG
 if ((pModuleState->m_fRegisteredClasses & AFX_WIN95CTLS_MASK) == AFX_WIN95CTLS_MASK)
 {
  pModuleState->m_fRegisteredClasses |= AFX_WNDCOMMCTLS_REG;
  fRegisteredClasses |= AFX_WNDCOMMCTLS_REG;
 }

 // must have registered at least as mamy classes as requested
 return (fToRegister & fRegisteredClasses) == fToRegister;
}
```

首先，AfxEndDeferRegisterClass 会检查参数 fToRegister，判断目标窗体是否需要注册。这一点很重要，因为 MFC 会尝试在多个地方注册窗口类。

然再利用 memset 清空 WNDCLASS 的结构，这样一来，除了那些显示设置的地方，其他的都是 NULL。然后我们看看初始化动作。

Wndcls::lpfnWndProc 首先被初始化指向 DefWindowProc()，（如果你用 SDK 写过 Windows 程序，对这个函数应该并不陌生）；然后将 wndcls::hInstance 初始化为当前实例的句柄；最后设置 wndcls::hCursor 为一般的鼠标指针。

接下来就是相似的动作：注册窗口类。它会根据不同的窗口对象类型来注册不同的窗口类，而且类风格（Class-Style）也不尽相同。

这里，M$决定在窗口创建其间注册窗口类，而不是在程序开始时注册窗口类，这不能不说是一个明智的决定。

窗口类注册需要时间，而 MFC 这样做则可以避免注册那些没有必要的窗口类，这样就能优化启动过程。

9.2. 窗口的 HOOK

在应用程序的初始化过程中，MFC 会安装几个钩子。

在 AfxInitThread()函数中（就是在 AfxWinInit()中被我们跳过的那个函数），MFC 会安装 WH_MSGFILTER 钩子，用来监听对话框、消息框、菜单条等控件中的输入事件。

然后又在 CWnd::Create 之前，利用 AfxHookWindowCreate(CWnd* pWnd)安装了 WH_CBT 钩子。安装这个钩子是为了让窗体在被创建时立刻进入。MFC 不能等到 CreateWindowEx()返回的时候在进入，因为那个时候已经有一些消息被发送。

CWnd 对象必须在任何消息被送给窗口之前让 MFC 进入。WH_CBT 就顺带的完成了这个使命。（为什么说顺带，因为这个不是 WH_CBT 的主要使命）

对于 MFC7.X 来说，MFC 总共安装了 4 个钩子，除了上面两个，还有 WH_KEYBOARD 和 WH_MOUSE，但是本人不才，没发现这两个在哪里装的- -!知道的麻烦告诉我~

10. CWinApp::Run() — MFC 的消息泵

一个合格的 Windows 程序必须要能接受并过滤消息，并且对于指定的消息，要能做出相应的反应。

在 SDK 中，我们使用 GetMessage()...DispatchMessage()循环和回调函数来完成这些功能，但是在 MFC 中，会有一点点不同。

这里，我们先回到 AfxWinMain()函数，看到了 nReturnCode = pThread->Run();了么？使用 Run()会自动启动消息循环。

因为多态的特性，上面的代码会调用 CWinApp::Run()，所以我们先看看 CWinApp::Run()中都有哪些神奇：

```
int CWinApp::Run()
{
 if (m_pMainWnd == NULL && AfxOleGetUserCtrl())
 {
  // Not launched /Embedding or /Automation, but has no main window!
```

```
    TRACE(traceAppMsg, 0, "Warning: m_pMainWnd is NULL in CWinApp::Run - quitting application.\n");
    AfxPostQuitMessage(0);
  }
  return CWinThread::Run();
}
```

明显，还和 CWinThread::Run()有关（别忘了前面说到的 MFC7.X 中 Run 实际位于 CWinThread）。既然如此，我们还是继续来看 CWinThread::Run()（源代码在 THRDCORD.CPP 中）：

```
int CWinThread::Run()
{
  ASSERT_VALID(this);
  _AFX_THREAD_STATE* pState = AfxGetThreadState();

  // for tracking the idle time state
  BOOL bIdle = TRUE;
  LONG lIdleCount = 0;

  // acquire and dispatch messages until a WM_QUIT message is received.
  for (;;)
  {
    // phase1: check to see if we can do idle work
    while (bIdle &&
      !::PeekMessage(&(pState->m_msgCur), NULL, NULL, NULL, PM_NOREMOVE))
    {
      // call OnIdle while in bIdle state
      if (!OnIdle(lIdleCount++))
        bIdle = FALSE; // assume "no idle" state
    }

    // phase2: pump messages while available
    do
    {
      // pump message, but quit on WM_QUIT
      if (!PumpMessage())
        return ExitInstance();
      // reset "no idle" state after pumping "normal" message
      //if (IsIdleMessage(&m_msgCur))
      if (IsIdleMessage(&(pState->m_msgCur)))
      {
        bIdle = TRUE;
        lIdleCount = 0;
      }

    } while (::PeekMessage(&(pState->m_msgCur), NULL, NULL, NULL, PM_NOREMOVE));
  }
}
```

可以发现重点的函数 PumpMessage()，继续跟进~可以发现这个函数的代码如下：

```
BOOL CWinThread::PumpMessage()
{
 return AfxInternalPumpMessage();
}
```

PumpMessage()把处理委托给了 AfxInternalPumpMessage()- -!，继续跟进吧~

```
BOOL AFXAPI AfxInternalPumpMessage()
{
 _AFX_THREAD_STATE *pState = AfxGetThreadState();

 if (!::GetMessage(&(pState->m_msgCur), NULL, NULL, NULL))
 {
#ifdef _DEBUG
  TRACE(traceAppMsg, 1, "CWinThread::PumpMessage - Received WM_QUIT.\n");
   pState->m_nDisablePumpCount++; // application must die
#endif
  // Note: prevents calling message loop things in 'ExitInstance'
  // will never be decremented
  return FALSE;
 }

#ifdef _DEBUG
 if (pState->m_nDisablePumpCount != 0)
 {
  TRACE(traceAppMsg, 0, "Error: CWinThread::PumpMessage called when not permitted.\n");
  ASSERT(FALSE);
 }
#endif

#ifdef _DEBUG
 _AfxTraceMsg(_T("PumpMessage"), &(pState->m_msgCur));
#endif

 // process this message

 if (pState->m_msgCur.message != WM_KICKIDLE && !AfxPreTranslateMessage(&(pState->m_msgCur)))
 {
  ::TranslateMessage(&(pState->m_msgCur));
  ::DispatchMessage(&(pState->m_msgCur));
 }
 return TRUE;
}
```

终于，我们发现了熟悉的东西！

MFC 仍然使用 GetMessage()获取消息，然后 DispatchMessage()给处理函数。还记得前面注册窗口类的时候，用 AfxEndDeferRegisterClass 把回调函数指定为 DefWindowProc()么？

不过实际上，消息并不是送往 DefWindowProc()，而是被转送到 AfxWndProc()全局函数中（其中使用了大量的 Hook 的 Subclass）。因为 MFC 要通过消息映射机制来处理消息。

11. 消息映射机制

看到这个标题别害怕，我们现在不研究 MFC 的消息运行机制。因为 MFC 的消息运行机制比 Windows 的消息运行机制更加复杂更加扑朔迷离。当然，这也更加有趣~

我们现在真是轻松的浏览下，对 MFC 的消息映射机制大概有个了解。

我们在开始例子代码中 CMFCAppWinow 中写下了消息映射的声明：

```
class CMFCAppWindow : public CFrameWnd
{

   ……
   // 下面是消息映射的东东
    afx_msg void OnLButtonDblClk(UINT nFlags, CPoint point);  // 左键双击的消息声明
    afx_msg void OnPaint();  // WM_PAINT 消息声明
    DECLARE_MESSAGE_MAP()  // 消息映射宏
};
```

然后指定消息映射表：

```
// MS 叫做消息映射表
BEGIN_MESSAGE_MAP(CMFCAppWindow, CFrameWnd)
  ON_WM_LBUTTONDBLCLK()
  ON_WM_PAINT()
END_MESSAGE_MAP()
```

最后在消息映射函数中作出相应的动作。

12. 尾声

好了，终于要结尾了。该讲的我都已经讲完了~大家也应该对 MFC 程序的基本运行机制有了一定的理解~

然后要说的是，研究 MFC 一定要自己勤动手，勤查 MSDN、M$知识库和 Google。最重要的就是勤翻 MFC 的源代码，因为 MFC 改版很快，而源代码能让你对 MFC 的行为心里有数。如果你以前看过 MFC 程序的内幕，可能会发现很多地方和文章里面讲的有出入。

是的，这就像开头说的，MFC 并不完美，也永远不可能完美，因为我们都在进步！

还有就是，本人也是刚接触 MFC 不久，因为水平有限，对 MFC 的运行机制也只能做到基本剖析。如果文章中存在问题，请告诉我，谢谢！

最后感谢一直支持我的众多 CFAN 编程版、MDSA Group 和 JAFT 内阁委的兄弟姐妹~

SetErrorMode

The SetErrorMode function controls whether the system will handle the specified types of serious errors, or whether the process will handle them.

UINT SetErrorMode(
  UINT *uMode*
);

Parameters

uMode

[in] Process error mode. This parameter can be one or more of the following values.

Value	Meaning
0	Use the system default, which is to display all error dialog boxes.
SEM_FAILCRITICALERRORS	The system does not display the critical-error-handler message box. Instead, the system sends the error to the calling process.
SEM_NOALIGNMENTFAULTEXCEPT	64-bit Windows: The system automatically fixes memory alignment faults and makes them invisible to the application. It does this for the calling process and any descendant processes. After this value is set for a process, subsequent attempts to clear the value are ignored.
SEM_NOGPFAULTERRORBOX	The system does not display the general-protection-fault message box. This flag should only be set by debugging applications that handle general protection (GP) faults themselves with an exception handler.
SEM_NOOPENFILEERRORBOX	The system does not display a message box when it fails to find a file. Instead, the error is returned to the calling process.

Return Values

The return value is the previous state of the error-mode bit flags.

Remarks

Each process has an associated error mode that indicates to the system how the application is going to respond to serious errors. A child process inherits the error mode of its parent process.

Itanium: An application must explicitly call SetErrorMode with SEM_NOALIGNMENTFAULTEXCEPT to have the system automatically fix alignment faults. The default setting is for the system to make alignment faults visible to an application

x86: The system does not make alignment faults visible to an application. Therefore, specifying SEM_NOALIGNMENTFAULTEXCEPT is not an error, but the system is free to silently ignore the request. This means that code sequences such as the following are not always valid on x86 computers:

```
SetErrorMode(SEM_NOALIGNMENTFAULTEXCEPT);
fuOldErrorMode = SetErrorMode(0);
ASSERT(fuOldErrorMode == SEM_NOALIGNMENTFAULTEXCEPT);
```

RISC: Misaligned memory references cause an alignment fault exception. To control whether the system automatically fixes such alignment faults or makes them visible to an application, use SEM_NOALIGNMENTFAULTEXCEPT.

MIPS: An application must explicitly call SetErrorMode with SEM_NOALIGNMENTFAULTEXCEPT to have the system automatically fix alignment faults. The default setting is for the system to make alignment faults visible to an application.

Alpha: To control the alignment fault behavior, set the EnableAlignmentFaultExceptions value in the HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager registry key as follows.

Value	Meaning
0	Automatically fix alignment faults. This is the default.
1	Make alignment faults visible to the application. You must call SetErrorMode with SEM_NOALIGNMENTFAULTEXCEPT to have the system automatically fix alignment faults.

Requirements

Client: Included in Windows XP, Windows 2000 Professional, Windows NT Workstation, Windows Me, Windows 98, and Windows 95.

Server: Included in Windows Server 2003, Windows 2000 Server, and Windows NT Server.

Header: Declared in Winbase.h; include Windows.h.

Library: Use Kernel32.lib.

GetModuleFileName

The GetModuleFileName function retrieves the fully qualified path for the specified module.

To specify the process that contains the module, use the GetModuleFileNameEx function.

```
DWORD GetModuleFileName(
  HMODULE hModule,
  LPTSTR lpFilename,
  DWORD nSize
);
```

Parameters

hModule

[in] Handle to the module whose path is being requested. If this parameter is NULL, GetModuleFileName retrieves the path for the current module.

lpFilename

[out] Pointer to a buffer that receives a null-terminated string that specifies the fully-qualified path of the module. If the length of the path exceeds the size specified by the *nSize* parameter, the function succeeds and the string is truncated to *nSize* characters and null terminated.

The path can have the prefix "\\?\", depending on how the module was loaded. For more information, see Naming a File.

nSize

[in] Size of the *lpFilename* buffer, in TCHARs.

Return Values

If the function succeeds, the return value is the length of the string copied to the buffer, in TCHARs. If the buffer is too small to hold the module name, the string is truncated to *nSize*, and the function returns *nSize*.
If the function fails, the return value is zero. To get extended error information, call GetLastError.
Remarks
If a DLL is loaded in two processes, its file name in one process may differ in case from its file name in the other process.
For the ANSI version of the function, the number of TCHARs is the number of bytes; for the Unicode version, it is the number of characters.
Windows Me/98/95: This function retrieves long file names when an application's version number is greater than or equal to 4.00 and the long file name is available. Otherwise, it returns only 8.3 format file names.
Windows Me/98/95: GetModuleFileNameW is supported by the Microsoft Layer for Unicode. To use this, you must add certain files to your application, as outlined in Microsoft Layer for Unicode on Windows Me/98/95 Systems.
Requirements
Client: Included in Windows XP, Windows 2000 Professional, Windows NT Workstation, Windows Me, Windows 98, and Windows 95.
Server: Included in Windows Server 2003, Windows 2000 Server, and Windows NT Server.
Unicode: Implemented as Unicode and ANSI versions. Note that Unicode support on Windows Me/98/95 requires Microsoft Layer for Unicode.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.

AfxThrowUserException

Throws an exception to stop an end-user operation.
void AfxThrowUserException( );
Remarks
This function is normally called immediately after AfxMessageBox has reported an error to the user.

PathFindExtension

Searches a path for an extension.
Syntax
LPTSTR PathFindExtension(
LPCTSTR *pPath*
);
Parameters
pPath
[in] Pointer to a null-terminated string of maximum length MAX_PATH that contains the path that contains the extension for which to search.
Return Value
Returns the address of the "." preceding the extension within *pPath* if an extension is found, or the address of the trailing NULL character otherwise.
Function Information

Minimum DLL Version	shlwapi.dll version 4.71 or later
Custom Implementation	No
Header	shlwapi.h
Import library	shlwapi.lib
Minimum operating systems	Windows 2000, Windows NT 4.0 with Internet Explorer 4.0, Windows 98, Windows 95 with Internet Explorer 4.0

AfxEnableMemoryTracking

Diagnostic memory tracking is normally enabled in the Debug version of MFC.

```
BOOL AfxEnableMemoryTracking(
    BOOL bTrack
);
```

Parameters

bTrack

Setting this value to TRUE turns on memory tracking; FALSE turns it off.

Return Value

The previous setting of the tracking-enable flag.

Remarks

Use this function to disable tracking on sections of your code that you know are allocating blocks correctly.

For more information on AfxEnableMemoryTracking, see Debugging MFC Applications.

Note This function works only in the Debug version of MFC.

CFrameWnd::Create

Call to create and initialize the Windows frame window associated with the CFrameWnd object.

```
virtual BOOL Create(
    LPCTSTR lpszClassName,
    LPCTSTR lpszWindowName,
    DWORD dwStyle = WS_OVERLAPPEDWINDOW,
    const RECT& rect = rectDefault,
    CWnd* pParentWnd = NULL,
    LPCTSTR lpszMenuName = NULL,
    DWORD dwExStyle = 0,
    CCreateContext* pContext = NULL
);
```

Parameters

lpszClassName

Points to a null-terminated character string that names the Windows class. The class name can be any name registered with the AfxRegisterWndClass global function or the RegisterClass Windows function. If NULL, uses the predefined default CFrameWnd attributes.

lpszWindowName

Points to a null-terminated character string that represents the window name. Used as text for the title bar.

dwStyle
Specifies the window style attributes. Include the FWS_ADDTOTITLE style if you want the title bar to automatically display the name of the document represented in the window.
rect
Specifies the size and position of the window. The rectDefault value allows Windows to specify the size and position of the new window.
pParentWnd
Specifies the parent window of this frame window. This parameter should be NULL for top-level frame windows.
lpszMenuName
Identifies the name of the menu resource to be used with the window. Use MAKEINTRESOURCE if the menu has an integer ID instead of a string. This parameter can be NULL.
dwExStyle
Specifies the window extended style attributes.
pContext
Specifies a pointer to a CCreateContext structure. This parameter can be NULL.
Return Value
Nonzero if initialization is successful; otherwise 0.
Remarks
Construct a CFrameWnd object in two steps. First, invoke the constructor, which constructs the CFrameWnd object, and then call Create, which creates the Windows frame window and attaches it to the CFrameWnd object. Create initializes the window's class name and window name and registers default values for its style, parent, and associated menu.
Use LoadFrame rather than Create to load the frame window from a resource instead of specifying its arguments.

B．参考文献

Microsoft MSDN http://msdn2.microsoft.com/zh-cn/default.aspx

Microsoft Support http://support.microsoft.com/gp/kbindex

Inside the Microsoft Foundation Class Architecture

Dissecting MFC 2e

C．版权声明

My Blog：http://kingsamchen.blogspot.com MSN：kingsamchen@hotmail.com

QQ：106047259

E-mail：kingsamchen@hotmail.com / kingsamchen@gmail.com

MSDA BBS：http://mdsa-group.5d6d.com　CFAN BBS：http://bbs.cfan.com.cn