1 MFC 编程基础

1.1 MFC 程序分类

win32程序与MFC程序之区别：MFC程序可以调用MFC类库，但win32程序无法调用；那么，MFC程序可分为：

- MFC控制台程序（win32 + Console application + MFC）
  - MFC控制台程序与win32程序之差别：
    - 主函数不同于普通的控制台程序（mian函数重命名、形参个数增多）：

      ```
      int _tmain(int argc, TCHAR* argv[], TCHAR* envp[]) {...}
      ```

    - 多了一个全局对象：`CWinApp theApp;`
  - 经验之谈：
    - 以 Afx 开头命名的函数：是 MFC 库中的全局函数
    - 以 :: 开头的函数：是 win32 的 API 函数
- MFC库程序
  - MFC静态库（win32 + Static library + MFC）：即使用MFC制作自己的静态库程序
  - MFC动态库/规则库（MFC DLL - Regular DLL）：即使用MFC制作自己的动态库程序
    - 使用静态的MFC库制作自己的动态库：Regular DLL using shared MFC DLL
    - 使用动态的MFC库制作自己的动态库：Regular DLL with MFC statically linked
  - MFC扩展库（MFC DLL - MFC extension DLL）：我现在所制作的库是对MFC库的扩展，扩展MFC的功能
    - Regular DLL可以被各种程序所调用
    - MFC extension DLL只能被MFC程序调用
- **MFC窗口程序（MFC Application）**
  - 单文档视图架构程序（MFC Application + Single document）

    其基础结构由以下几个类构成：
    - CWinApp：应用程序类，负责管理应用程序的流程
    - CFrameWnd：框架窗口类，负责管理框架窗口，其类的对象为框架窗口
    - CView：视图窗口类，负责显示数据，其类的对象为视图窗口
    - CDocument：文档类，负责管理数据（比如数据如何接收/如何转换/如何存储）
    - CAboutDlg：用于维护对话框，但不参与架构
  - 多文档视图架构程序（MFC Application + Multiple documents）
    - CWinApp：应用程序类，负责管理应用程序的流程
    - CMDIFrameWnd：多文档主框架窗口类
    - CMDIChileWnd：多文档子框架窗口类
    - CView：视图窗口类，负责显示数据，其类的对象为视图窗口

        - CDocument：文档类，负责管理数据（比如数据如何接收/如何转换/如何存储）
        - CAboutDlg：用于维护对话框，但不参与架构
    - 对话框应用程序（MFC Application + Dialog based）

      何为对话框：程序一旦运行起来，出现的第一个主界面就是对话框

        - CWinApp：应用程序类，负责管理应用程序的流程
        - CDialogEx/CDialog：对话框窗口类
        - CAboutDlg：用于维护对话框，但不参与架构

1.2 MFC 库中类的简介

- `CObject`：MFC 库中绝大部分类的最基类，提供了 MFC 类库中的一些基本的机制：
    - 对运行时类信息的支持
    - 对动态创建的支持
    - 对序列化的支持
- `CWinApp`：应用程序类，封装了应用程序、线程等信息
- `CDocument`：文档类，管理数据
- `Frame Windows`：框架窗口类，封装了窗口程序组成的各种框架窗口
- `CSplitterWnd`：用来完成拆分窗口的类
- `Control Bars`：控件条类
- `Dialog Boxes`：对话框类，封装了各种对话框，通用的对话框
- `Views`：视图类，封装了各种显示窗口
- `Controls`：控件类，封装了各种常用的控件
- `Exceptions`：异常处理类，封装了 MFC 中常用的各种异常
- `File`：文件类，各种文件的 I/O 操作等
- 绘图类：包括 `CDC` 类和 `CGdiObject` 类
- 数据集合类：`CArray` / `Clist` / `CMap`，封装了相应的数据结构的管理
- 非CObject类的子类：提供了各种数据结构相关的管理，例如 `CPoint`, `CTime`, `CString` 等

1.3 简易的 MFC 程序

1.3.1 设置开发环境

- 头文件包含：`#include <afxwin.h>`
- setting 中设置使用 MFC 库
- 总结：win32 程序和 MFC 程序之区别仅在于能不能使用 MFC 库

1.3.2 代码示例

- 定义一个自己的框架类 `CMyFrameWnd`，派生自 `CFrameWnd` 类：

```
class CMyFrameWnd : public CFrameWnd {...};
```

- 定义一个自己的应用程序类 `CMyWinApp`，派生自 `CWinApp` 类，并定义构造以及重写 `InitInstance` 虚函数，在函数中创建并显示窗口：

```
class CMyWinApp : public CWinApp {
public:
    CMyWinApp() {}
    virtual BOOL InitInstance() {
        CMyFrameWnd* pFrame = new CMyFrameWnd;
        pFrame->Create(NULL, "MFCBase");
        this->m_pMainWnd = pFrame;  // 形成了CMyWinApp->CMyFrameWnd的委派关系
        pFrame->ShowWindow(SW_SHOW);
        pFrame->UpdateWindow();
        return TRUE;
    }
};
```

- 定义 `CMyWinApp` 类的对象：

```
CMyWinApp theApp;
```

1.3.3 MFC程序启动原理详解

1.3.3.1 入口函数

与 win32 窗口程序相同，MFC 程序的入口也是 `WinMain`。但是，MFC 库已经实现了 `WinMain` 函数，所以在程序中不需要实现。即：在 win32 程序中 `WinMain` 由程序员自己实现，那么流程是程序员安排，但在 MFC 中，由于 MFC 库实现了 `WinMain`，也就意味着 MFC 负责安排程序的流程。

1.3.3.2 执行流程

- 程序启动，首先会构造 `theApp` 对象，即调用基类 `CWinApp` 的构造函数：
  - 将 `theApp` 对象的地址保存到线程状态信息中
  - 将 `theApp` 对象的地址保存到模块状态信息中
  - 进入 `WinMain` 函数，调用 `AfxWinMain` 函数
    - 获取应用程序类对象 `theApp` 的地址
    - 利用 `theApp` 地址调用 `InitApplication`，初始化当前应用程序的数据
    - 利用 `theApp` 地址调用 `InitInstance` 函数初始化程序，在函数中我们创建窗口并显示
    - 利用 `theApp` 地址调用 `CWinApp` 的 `Run` 成员函数进行消息循环
    - 在消息循环内，若没有消息，利用 `theApp` 地址调用 `OnIdle` 虚函数实现空闲处理
    - 在消息循环内，程序退出时，利用 `theApp` 地址调用 `ExitInstance` 虚函数实现退出前的善后处理工作

```
```

```
/*-------------------- 伪代码 --------------------*/

/**
 * MFC库中共有3个全局变量, 其中两个如下(但并不知道变量具体名称):
 * AFX_MODULE_STATE aaa;  // 当前程序模块状态信息
 * AFX_MODULE_THREAD_STATE bbb;  // 当前程序线程状态信息
 * _AFX_THREAD_STATE ccc;  // 当前程序线程信息(在2.1.3节会使用到)
**/

CWinApp::CWinApp() {  // 用于构造全局对象: CMyWinApp theApp;
    ...

    // 获取全局变量aaa的地址(&aaa)
    AFX_MODULE_STATE* pModuleState = AfxGetModuleState();

    // 获取全局变量bbb的地址(&bbb)
    AFX_MODULE_THREAD_STATE* pThreadState = pModuleState->m_thread;

    // 将我们自己构造的全局对象theApp的地址(&theApp)保存到bbb的一个成员变量中
    pThreadState->m_pCurrentWinThread = this;
    AfxGetThread() {  // 用于获取全局对象theApp
        AFX_MODULE_THREAD_STATE* pState = AfeGetModuleThreadState();
        CWinThread* pThread = pState->m_pCurrentWinThread;
        return pThread;
    }

    // 将我们自己构造的全局对象theApp的地址(&theApp)保存到aaa的一个成员变量中
    pModuleState->m_pCurrentWinApp = this;
    AfxGetApp() {  // 用于获取全局对象theApp
        return AfxGetModuleState()->m_pCurrentWinApp;
    }

    ...
}

_tWinMain(...) {  // 程序流程会跟随这theApp对象的指导行进
    AfxWinMain(...) {

        // 父类的指针指向子类的对象, 获取全局对象theApp
        CWinThread* pThread = AfxGetThread();
        // 父类的指针指向子类的对象, 获取全局对象theApp
        CWinApp* pApp = AfxGetApp();

        // 利用theApp对象调用应用程序类成员虚函数, 做初始化工作
        pApp->InitApplication();
        // 利用theApp对象调用重写的InitInstance(), 创建并显示窗口
        pThread->InitInstance();
        // 利用theApp对象调用重写的InitInstance(), 进行消息循环
        pThread->Run() {
            for (;;) {  // 消息循环
                while(没有消息时) {
                    // 利用theApp对象调用应用程序类成员虚函数, 做空闲处理
                    OnIdle(...);
                }
                do {
                    if (GetMessage抓到WN_QUIT消息)
```

```
                        // 利用theApp对象调用应用程序类成员虚函数，善后处理
                        return ExitInstance();
                    } while (...)
                }
            }
        }
    }
```

1.3.3.3 `CWinApp` 的常用成员

1. 成员虚函数
    - `InitInstance`：程序的初始化函数，完成了窗口创建等初始化处理
    - `ExitInstance`：程序推出时调用，进行清理资源等善后工作
    - `Run`：消息循环函数
    - `OnIdle`：消息循环内，空闲处理函数
2. 成员变量
    - `m_pMainWnd`：当前应用程序的主窗口，即主框架窗口的地址

2 MFC 窗口创建过程

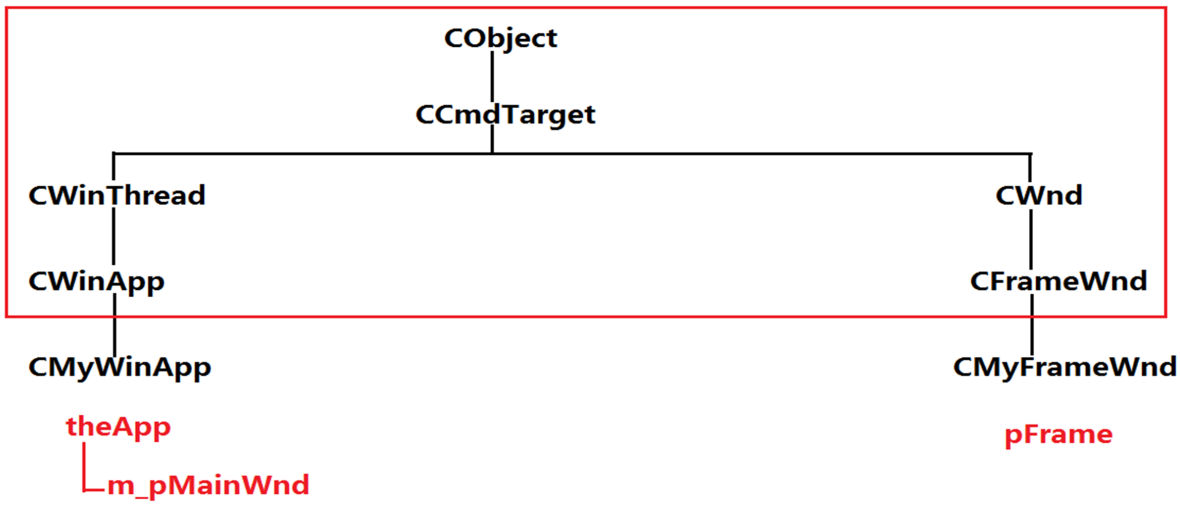
2.1 钩子函数及其相关

```
/* 【钩子函数】创建钩子 */
HHOOK SetWindowsHookEx(
    int idHook,          // 钩子类型(WH_CBT: 该类型仅处理窗口创建消息(WM_CREAT))
    HOOKPROC lpfn,       // 钩子处理函数: 自己定义的回调函数
    HINSTANCE hMod,      // 应用程序实例句柄: 若指明句柄，则仅钩取该进程的消息(限制钩取范围)
    DWORD dwThreadId     // 线程ID: 若指明线程ID，则仅钩取该线程的消息(限制钩取范围)
);

/* 【钩子函数】钩子处理函数 */
LRESULT CALLBACK CBTProc(
    int nCode,           // 钩子码，与钩子类型相对应(HCBT_CREATEWND)
    WPARAM wParam,       // 刚刚创建成功的窗口的句柄
    LPARAM lParam        // ...
);

/* 更改窗口处理函数 */
LONG_PTR SetWindowLongPtr() {
    HWND hWnd,           // 窗口句柄
    int nIndex,          // GWLP_WNDPROC
    LONG_PTR dwNewLong   // 新的窗口处理函数名(函数地址)
}
```

2.2 示例基础代码

```
/**
 * 项目名称: MFCCreat
 * 项目设置: win32 + Windows application + MFC
*/

#include <afxwin.h>

class CMyFrameWnd : public CFrameWnd {};

class CMyWinApp : public CWinApp {
public:
    virtual BOOL InitInstance() {
        CMyFrameWnd* pFrame = new CMyFrameWnd;
        pFrame->Creat(NULL, "MFCCreate");  // 接下来将详解创建窗口之过程
        m_pMainWnd = pFrame;
        pFrame->ShowWindow(SW_SHOW);
        pFrame->UpdateWindow();
        return TRUE;
    }
};

CMyWinApp theApp;
```

上述代码的整体构架如下：



2.3 MFC 窗口创建过程详解

1. 加载菜单
2. 调用 `CWnd::CreateEx` 函数创建窗口
    - 调用 `PreCreateWindow` 函数设计和注册窗口类，在其内部调用 `AfxDeferRegisterClass` 函数，在此函数中设计窗口类
    - 调用 `AfxHookWindowCreate` 函数
        - 在函数内部，调用 `SetWindowsHookEx` 创建 `WH_CBT` 类型的钩子，钩子处理函数为 `_AfxCbtFilterHook`

        - 将框架类对象地址 `pFrame` 保存到 “当前程序线程信息” 中
    - 调用 `CreateWindowEx` 函数创建窗口，进而立刻调用钩子处理函数
3. 钩子处理函数 `_AfxCbtFilterHook`
    - 将窗口句柄和框架类对象地址建立一对一的绑定关系
    - 使用 `SetWindowLong` 函数，将窗口处理函数设置为 `AfxWndProc` (这是真正的窗口处理函数)

```
/*-------------------- 伪代码 --------------------*/
CMyFrameWnd* pFrame = new CMyFrameWnd;
pFrame->Create(NULL, "MFCCreate") {  // 函数内部this为pFrame(自己new出的框架类对象地址)
    // 1.加载菜单
    // 2.创建窗口
    CreateEx(...,NULL,...) {  // 第二个参数为传入Create的NULL
        CREATESTRUCT cs;  // CREATESTRUCT的12个成员与CreateWindowEx的12个参数一一对应
        cs.dwExStyle = dwExStyle;                // 扩展风格
        cs.lpszClass = lpszClassName;            // 窗口类名称，传入参数现在是NULL
        cs.lpszName = lpszWindowName;            // 标题栏信息
        cs.style = dwStyle;                      // 基本风格
        cs.x = x;                                // 窗口的位置(X)
        cs.y = y;                                // 窗口的位置(Y)
        cs.cx = nWidth;                          // 窗口的宽度
        cs.cy = nHeight;                         // 窗口的高度
        cs.hwndParent = hWndParent;              // 副窗口
        cs.hMenu = nIDorHMenu;                   // 菜单
        cs.hInstance = AfxGetInstanceHandle();  // 当前程序实例句柄
        cs.lpCreateParams = lpParam;             // 创建窗口的最后一个参数
        // cs.lpszClass现在是lpszClassName(NULL)，PreCreateWindow即对cs.lpszClass的值重新赋值
        PreCreateWindow(cs) {  // 设计并注册窗口类
            AfxDeferRegisterClass(...) {  // 设计窗口类
                WNDCLASS wndcls;
                ...
                // 暂定窗口处理函数为DefWindowProc
                // 下面将在钩子函数中更改窗口处理函数而非使用此默认项
                wndcls.lpfnWndProc = DefWindowProc;
                wndcls.lpClassName = "AfxFrameOrView100sd";
                ::RegisterClass(&wndcls);  // 注册窗口类
            }
            cs.lpszClass = _afxWndFrameOrView;  // 字符串赋值"AfxFrameOrView100sd"
        }
        AfxHookWindowCreate(pFrame) {
            // 获取全局变量ccc(当前程序线程信息)的地址
            _AFX_THREAD_STATE* pThreadState = _afxThreadState.GetDate();
            // 利用win32的API函数埋了一个类型为WH_CBT(仅对窗口创建消息有效)的钩子
            ::SetWindowsHookEx(WH_CBT, _AfxCbtFilterHook);
            // 将自己new的框架类对象pFrame保存到全局变量ccc中
            pThreadState->m_pWndInit = pFrame;
        }
        ::CreateWindowEx(...);  // 创建窗口，并立即触发上面注册的钩子处理函数_AfxCbtFilterHook
    }
}

/**
```

```
 * 钩子处理函数
 * 1.建立[自己new的框架类对象pFrame]与[框架窗口句柄hWnd]之间的绑定关系
 * 2.更改窗口处理函数(从DefWindowProc修改为AfxWndProc)
 */
_AfxCbtFilterHook() {
    // 获取全局变量ccc(当前程序线程信息)的地址
    _AFX_THREAD_STATE* pThreadState = _afxThreadState.GetDate();
    // 从ccc中获取pFrame(之前有保存过)
    CWnd* pWndInit = pThreadState->m_pWndInit;  // m_pWndInit==pFrame
    // 1.建立pFrame和框架窗口句柄hWnd之间的绑定关系
    HWND hWnd = (HWND)wParam;  // 刚刚创建成功的框架窗口句柄
    pWndInit->Attach(hWnd);  // 建立pFrame与hWnd的相互调用绑定关系
    // 2.将窗口处理函数更改为AfxWndProc(这是真正的窗口处理函数)
    oldWndProc = (WNDPROC)SetWindowLongPtr(hWnd, GWLP_WNDPROC, AfxWndProc)
}
```

3 MFC 消息映射机制

3.1 传统消息处理方法

在2.2节代码基础上，在 `class CMyFrameWnd` 中添加一虚函数 `WindowProc`，这其实是我们能够接触到的窗口处理函数，我们能够在其中添加/重写窗口处理逻辑：

```
/**
 * 项目名称: MFCCreat
 * 项目设置: win32 + Windows application + MFC
*/

#include <afxwin.h>

class CMyFrameWnd : public CFrameWnd {
public:
    virtual LRSUIT WindowProc(UINT msgID, WPARAM wParam, LPARAM lPrarm) {
        switch(msgID) {
            case WM_CREATE:
                AfxMessageBox("WM_CREATE消息被处理");  // 弹提示框
                break;
        }
        return CFrameWnd::WindowProc(msgID, wParam, lPrarm);  // 调用父类虚函数，对其他消息进行默认处理
    }
};

class CMyWinApp : public CWinApp {
public:
    virtual BOOL InitInstance() {
        CMyFrameWnd* pFrame = new CMyFrameWnd;
        pFrame->Creat(NULL, "MFCCreate");  // 接下来将详解创建窗口之过程
        m_pMainWnd = pFrame;
```

```
        pFrame->ShowWindow(SW_SHOW);
        pFrame->UpdateWindow();
        return TRUE;
    }
};

CMyWinApp theApp;
```

从2.3小节我们知道，在钩子函数中最终更改的窗口处理函数为 `AfxWndProc`，其实该函数内部会调用我们在 `CMyFrameWnd` 中重写的虚函数 `WindowProc` 以进行窗口处理，这样我们自己也能自定义窗口处理函数了，下面是 `AfxWndProc` 的简单伪代码，以展示其调用过程：

```
/*-------------------- 伪代码 --------------------*/
AfxWndProc(HWND hWnd, UNIT nMsg, WPARAM wParam, LPARAM lParam) {

    /* 从窗口句柄hWnd拿到框架类对象pFrame，将其赋给pWnd */
    CWnd* pWnd = CWnd::FromHandlePermanent(hWnd);

    /* 通过pWnd(pFrame)调用我们在CMyFrameWnd重写的虚函数WindowProc(多态) */
    return AfxCallWndProc(pWnd, hWnd, nMsg, wParam, lParam) {  // pWnd==pFrame
        pWnd->WindowProc(nMsg, wParam, lParam);  // 调用我们重写的虚函数WindowProc
    }
}
```

【总结】由上可知，MFC 消息处理流程如下：

1. 当收到消息时，进入 `AfxWndProc` 函数；
2. `AfxWndProc` 函数根据消息的窗口句柄，查询对应的框架类对象地址 `pFrame`；
3. 利用框架类对象地址 `pFrame` 调用框架类成员虚函数 `WindowProc`，完成消息的处理；

【Tips】由上述代码得到的一些小启发：

- 在虚函数 `WindowProc` 中又 `return` 了其父类 `CFrameWnd` 的 `WindowProc`，并传入了相同的参数，这样做的目的是即使我们重写了窗口处理函数，但是也尽量让其走完 MFC 框架默认给我们创造的窗口处理逻辑；之后我们再重写类似虚函数时，也可以在 `return` 时调用父类的虚函数以让其走一遍默认流程；
- 由2.3节，钩子处理函数绑定了窗口句柄和框架类对象地址，双方都可以找到对方，比如，框架类里面的成员变量就有窗口句柄：`pFrame->m_hWnd;`
- `AfxMessageBox` 函数可在全局用于弹提示框

3.2 消息映射机制初探

消息映射机制的作用：

消息映射机制为 MFC 框架的第三大机制（前两大机制之前已讲过：程序启动机制、窗口创建机制），消息映射机制的作用是：在不重写 `WindowProc` 虚函数的前提下，仍然可以处理消息。

消息映射机制的使用：

- 类必须具备的条件
    - 类内必须添加声明宏：

```
DECLARE_MESSAGE_MAP()
```

- 类外必须添加实现宏：

```
BEGIN_MESSAGE_MAP(theClass, baseClass)
END_MESSAGE_MAP()
```

也就是说，当一个类具备上述两个条件，这个类就可以按照消息映射机制来处理消息。

消息映射机制的实施：

以 `WN_CREATE` 消息为例：

- `BEGIN_MESSAGE_MAP(theClass, baseClass)` 和 `END_MESSAGE_MAP()` 之间添加 `ON_MESSAGE(WM_CREATE, OnCreate)` 宏
- 在 `CMyFrameWnd` 类中添加 `OnCreate` 函数的声明和定义，注意：这里的函数名叫什么都可以

代码示例：

```
/**
 * 项目名称: MFCMsg
 * 项目设置: win32 + Windows application + MFC
*/

#include <afxwin.h>

class CMyFrameWnd : public CFrameWnd {
    DECLARE_MESSAGE_MAP()
public:
    LRESULT OnCreate(WPARAM wParam, LPARAM lParam) {
        AfxMessageBox("WM_CREATE消息被处理");
        return 0;
    }
};

BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
    ON_MESSAGE(WN_CREATE, OnCreate)
END_MESSAGE_MAP()

class CMyWinApp : public CWinApp {
public:
    virtual BOOL InitInstance() {
        CMyFrameWnd* pFrame = new CMyFrameWnd;
        pFrame->Creat(NULL, "MFCCreate");  // 接下来将详解创建窗口之过程
        m_pMainWnd = pFrame;
        pFrame->ShowWindow(SW_SHOW);
        pFrame->UpdateWindow();
        return TRUE;
    }
};

CMyWinApp theApp;
```

3.3 消息映射机制原理

3.3.1 声明宏与实现宏展开

声明宏 `DECLARE_MESSAGE_MAP()` 与实现宏 `BEGIN_MESSAGE_MAP(theClass, baseClass)` 、`END_MESSAGE_MAP()` 本质上为类内函数声明和类外函数实现的宏替换，我们将它们在代码中展开如下：

```
/**
 * 项目名称: MFCMsg(宏展开)
 * 项目设置: win32 + Windows application + MFC
*/

#include <afxwin.h>

class CMyFrameWnd : public CFrameWnd{
//DECLARE_MESSAGE_MAP()
protected:
    static const AFX_MSGMAP* PASCAL GetThisMessageMap();
    virtual const AFX_MSGMAP* GetMessageMap() const;
public:
    LRESULT OnCreate( WPARAM wParam, LPARAM lParam );
};

//BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
const AFX_MSGMAP* CMyFrameWnd::GetMessageMap() const
{
    return GetThisMessageMap();
}
const AFX_MSGMAP* PASCAL CMyFrameWnd::GetThisMessageMap()
{
        static const AFX_MSGMAP_ENTRY _messageEntries[] =
        {
            //ON_MESSAGE( WM_CREATE, OnCreate )
            { WM_CREATE, 0, 0, 0, AfxSig_lwl, (AFX_PMSG)(AFX_PMSGW)
                (static_cast< LRESULT (AFX_MSG_CALL CWnd::*)(WPARAM, LPARAM) >
(&OnCreate)) },
//END_MESSAGE_MAP()
            {0, 0, 0, 0, AfxSig_end, (AFX_PMSG)0 }
        };
        static const AFX_MSGMAP messageMap = { &CFrameWnd::GetThisMessageMap,
&_messageEntries[0] };
        return &messageMap;
}

LRESULT CMyFrameWnd::OnCreate( WPARAM wParam, LPARAM lParam ){
    AfxMessageBox( "WM_CREATE" );
    return 0;
}

class CMyWinApp : public CWinApp{
public:
    virtual BOOL InitInstance( );
};
```

```
BOOL CMyWinApp::InitInstance( ){
        CMyFrameWnd* pFrame = new CMyFrameWnd;
        pFrame->Create(NULL, "MFCCreate");
        m_pMainWnd = pFrame;
        pFrame->ShowWindow( SW_SHOW );
        pFrame->UpdateWindow( );
        return TRUE;
}

CMyWinApp theApp;//爆破点
```

3.3.2 数据结构

由3.3.1节伪代码：`GetThisMessageMap()` 函数中定义的数组 `_messageEntries` 用于存储消息ID和消息处理函数，其保存形式为结构体 `AFX_MSGMAP_ENTRY` 类型的元素，下面将该结构体的成员变量进行分析：

```
/* 静态数据每个元素的类型 */
struct AFX_MSGMP_ENTRY {
    UINT nMessage;  // 消息ID
    UINT nCode;     // 通知码
    UINT nID;       // 命令ID (可以填写第一个命令ID)
    UINT nLstID;    // 最后一个命令ID, 与上面的参数结合使用可以圈定一个命令ID的范围
    UINT_PTR nSig;  // 处理消息的函数类型
    AFX_PMSG pfn;   // 处理消息的函数名 (地址)
};
```

由3.3.1节伪代码：`GetThisMessageMap()` 函数中定义的静态常量 `messageMap` 的类型为结构体 `AFX_MSGMAP`，如下所示：

```
/* 静态常量的类型 */
struct AFX_MSGMAP {
    const AFX_MSGMP* (PSCAL* pfnGetBaseMap)();  // 父类宏展开的静态变量地址
    const AFX_MSGMAP_ENTRY* lpEntries;  // 本类宏展开的静态数组首地址
};
```

分析静态常量 `messageMap` 的定义：`static const AFX_MSGMAP messageMap = { &CFrameWnd::GetThisMessageMap, &_messageEntries[0] };`，其传入的第一个参数为父类的 `GetThisMessageMap()` 函数，根据该函数获取父类的 `messageMap`，而传入的第二个参数为当前自定义框架类的数组 `_messageEntries` 的首地址，而父类中也具有相同的静态常量的定义，以此类推，串成了由父类到子类的包含有各类数组 `_messageEntries` 的链表（有点类似于从子类向父类向前追溯的链表，链表头为当前自定义框架类）

3.3.3 宏展开各部分的作用

- `GetThisMessageMap()`
  - 类型：静态函数
  - 作用：定义静态数组和静态变量，并返回本类静态变量的地址（用于获取链表头）

- `_messageEntries[]`
  - 类型：静态数组（进程级生命周期）
  - 作用：数组每个元素，保存着消息ID和处理消息的函数名（地址）
- `messageMap`
  - 类型：静态变量（进程级生命周期）
  - 作用：
    - 第一个成员：保存父类宏展开的静态变量地址（负责连接链表）
    - 第二个成员：保存本类的静态数组首地址
- `GetMessgeMap()`
  - 类型：虚函数
  - 作用：直接调用 `GetThisMessageMap()` ，可返回本类静态变量的地址（用于获取链表头）

那么，这就形成了一个单向链表，每个链表节点中存储了一个数组，该数组中的内容为该数组所属框架类对象的消息及消息响应函数

3.3.4 遍历链表

```
/*-------------------- 伪代码 --------------------*/
AfxWndProc(HWND hWnd, UNIT nMsg, WPARAM wParam, LPARAM lParam) {

    /* 从窗口句柄hWnd拿到框架类对象pFrame，将其赋给pWnd(pWnd===pFrame) */
    CWnd* pWnd = CWnd::FromHandlePermanent(hWnd);

    /* 在该函数里遍历链表 */
    AfxCallWndProc(pWnd) {
        // 因为我们的框架类没有使用重写WindowProc的方法处理消息，而是使用消息映射机制处理消息
        // 所以这里pFrame将会调用到其父类（即CFrameWnd的虚函数WindowProc），其内部会遍历链
表
        pWnd->WindowProc(...) {  // CWnd::WindowProc
            OnWndMsg(...) {  // CWnd::OnWndMsg
                // ... 这里开始处理WM_COMMAND消息(省略)，下面处理其他消息
                // 下面调用了本框架类(pFrame)的GetMessageMap()，获取本框架类链表头节点
                const AFX_MSGMAP* pMessageMap; pMessageMap = GetMessageMap();
                const AFX_MSGMAP_ENTRY* lpEntry;
                // 下面的for循环开始遍历链表
                for (/* pMessageMap already init'ed */; pMessageMap-
>pfnGetBaseMap !=
                     NULL;pMessageMap = (*pMessageMap->pfnGetBaseMap)()) {
                    // 在当前链表节点中的消息数组中寻找特定的消息
                    if ((lpEntry = AfxFindMessageEntry(pMessageMap->lpEntries,
                                                       message, 0, 0)) != NULL) {
                        // 找到了消息
                        goto LDispatch;  // 跳出for循环
                    }
                    // 如果没找到消息，则遍历下一个链表节点，在其父类节点上寻找消息
                }
                LDispatch:
                    // 获取到该消息对应的函数地址，并调用该函数
                    lpEntry->pfn;
```

```
            }
        }
    }
}
```

3.4 消息的分类

3.4.1 消息的分类

- 标准Windows消息宏：`ON_WM_XXX`
- 自定义消息宏：`ON_MESSAGE`
- 命令消息宏：`ON_COMMAND`

上面使用的方法都是使用**自定义消息宏**来实现通过消息回调我们自己写的消息处理函数，但其实更推荐使用**标准Windows消息宏**这一方法

3.4.1.1 标准Windows消息宏（`ON_WM_XXX`）

标准Windows消息宏的宏名称为 `ON_WM_XXX`，`XXX` 表示某消息之代称；每一种 `ON_WM_XXX` 宏内部都已经写好了对应的消息处理函数（包括其函数名、返回值类型、形参个数及类型），这需要查询手册以编写，例子如下：

```
/**
 * 项目名称: MFCCmd
 * 项目设置: win32 + Windows application + MFC
 * 课程示例: 标准Windows消息宏
*/

#include <afxwin.h>

class CMyFrameWnd : public CFrameWnd {
    DECLARE_MESSAGE_MAP()
public:
    int OnCreate(LPCREATESTRUCT pcs) {
        AfxMessageBox("WM_CREATE消息被处理");
        return CFrameWnd::OnCreate(pcs);
    }
    void OnPaint() {
        PAINTSTRUCT ps = {0};
        HDC hdc = ::BeginPaint(this->m_hWnd, &ps);
        ::TextOutA(hdc, 100, 100, "hello", 5);
        ::EndPaint(m_hWnd, &ps);
    }
    void OnMouseMove(UINT nKey, CPoint pt) {
        /* 希望实现hello字符串跟随鼠标光标移动 */
        this->m_x = pt.x;
        this->m_y = pt.y;
        ::InvalidateRect(this->m_hWnd, NULL, TRUE);
    }
public:
    int m_x;
    int m_y;
```

```
};

BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
    ON_WM_CREATE()  // 处理窗口创建消息
    ON_WM_PAINT()  // 处理绘图消息
    ON_WM_MOUSEMOVE()  // 处理鼠标移动消息
END_MESSAGE_MAP()

class CMyWinApp : public CWinApp {
public:
    virtual BOOL InitInstance() {
        CMyFrameWnd* pFrame = new CMyFrameWnd;
        pFrame->Creat(NULL, "MFCCreate");
        m_pMainWnd = pFrame;
        pFrame->ShowWindow(SW_SHOW);
        pFrame->UpdateWindow();
        return TRUE;
    }
};

CMyWinApp theApp;
```

3.4.1.1 自定义消息宏（`ON_MESSAGE`）

1. 自定义一个消息宏 `#define WM_MYMESSAGE WM_USER+1001`
2. 使用 `ON_MESSAGE` 来设置此自定义消息及其消息处理函数
3. 由于此消息为自定义的，因此需要我们自己择时发送（使用 `::PostMessage` 函数）

```
/**
 * 项目名称: MFCCmd
 * 项目设置: win32 + Windows application + MFC
 * 课程示例: 自定义消息宏
*/

#include <afxwin.h>
#define WM_MYMESSAGE WM_USER+1001  // 这是我们自己定制的消息

class CMyFrameWnd : public CFrameWnd {
    DECLARE_MESSAGE_MAP()
public:
    int OnCreate(LPCREATESTRUCT pcs) {
        AfxMessageBox("WM_CREATE消息被处理");
        ::PostMessage(this->m_hWnd, WM_MYMESSAGE, 1, 2);  // 窗口创建后自己发送消息
        return CFrameWnd::OnCreate(pcs);
    }
    LRESULT OnMyMessage(WPARAM wParam, LPARAM lParam) {
        CString str;
        str.Format("wParam=%d, lParam=%d", wParam, lParam);
        AfxMessageBox(str);
        return 0;  // 父类中没有该函数（因为根本就是自己定义的）
    }
};

BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
```

```
    ON_CREATE()
    ON_MESSAGE(WM_MYMESSAGE, OnMyMessage)
END_MESSAGE_MAP()

class CMyWinApp : public CWinApp {
public:
    virtual BOOL InitInstance() {
        CMyFrameWnd* pFrame = new CMyFrameWnd;
        pFrame->Creat(NULL, "MFCCreate");
        m_pMainWnd = pFrame;
        pFrame->ShowWindow(SW_SHOW);
        pFrame->UpdateWindow();
        return TRUE;
    }
};

CMyWinApp theApp;
```

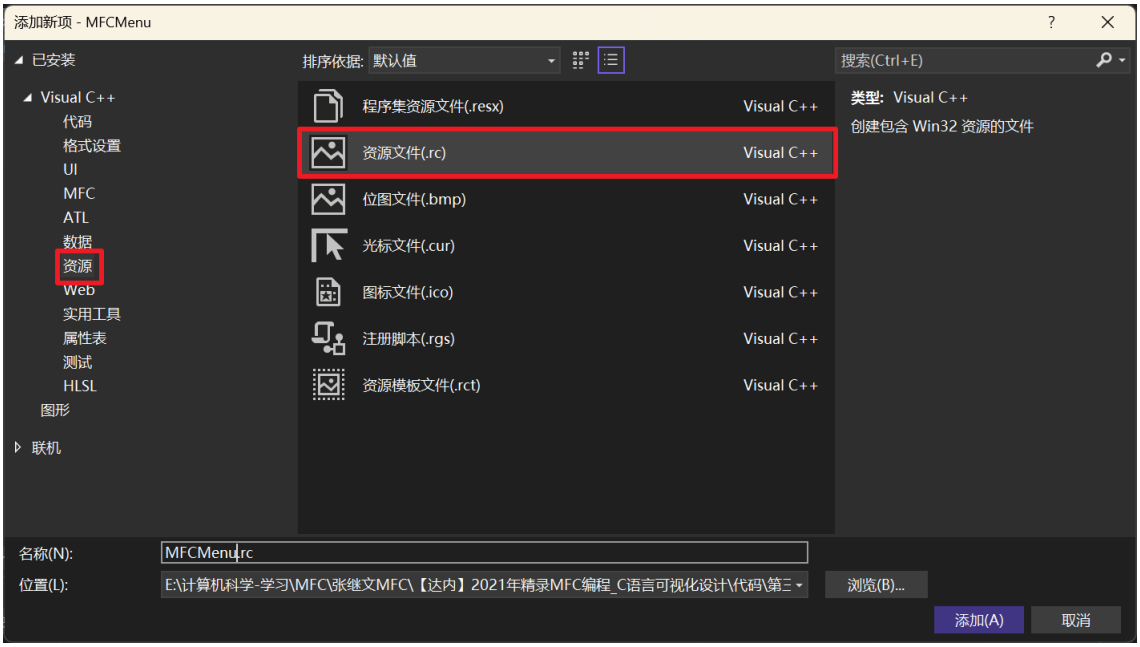
3.5 MFC 菜单

在 win32 中，使用的是 `HMENU` 这一菜单句柄来处理菜单；而在 MFC 中，使用的是 `CMenu` 这一类的对象来管理菜单；但其实， `CMenu` 类只有在内部关联了 `HMENU` 菜单句柄才能够管理菜单；因此，在 `CMenu` 类中，封装了关于管理菜单的各种操作的成员函数，另外还包含了一个十分重要的成员变量：`m_hMenu`（菜单句柄）

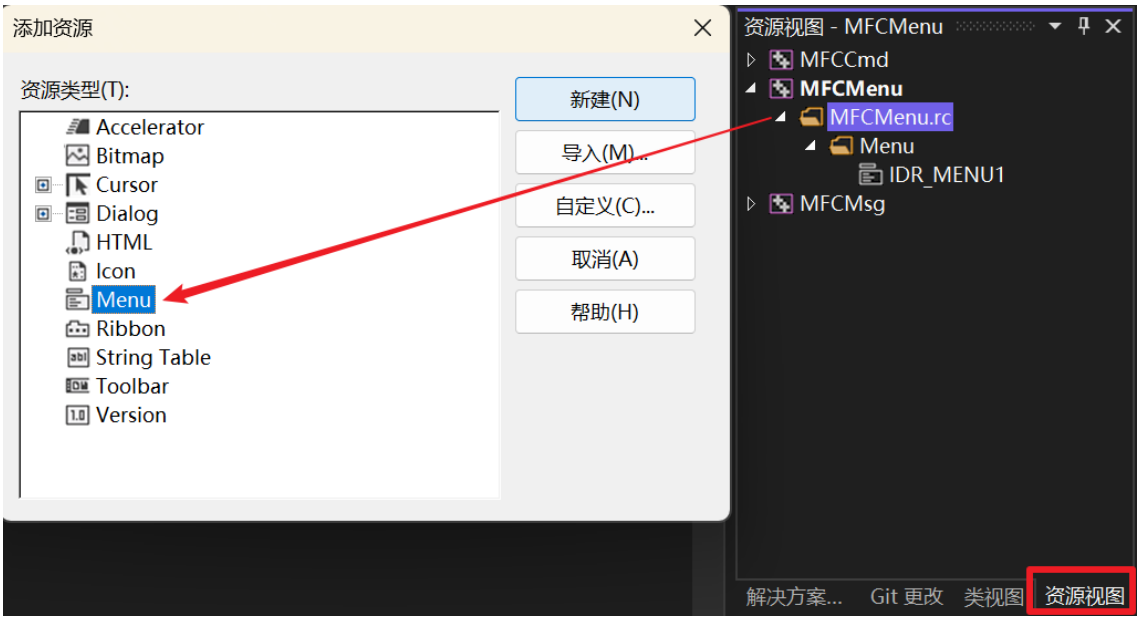
3.5.1 菜单的使用

1. **添加菜单资源**：

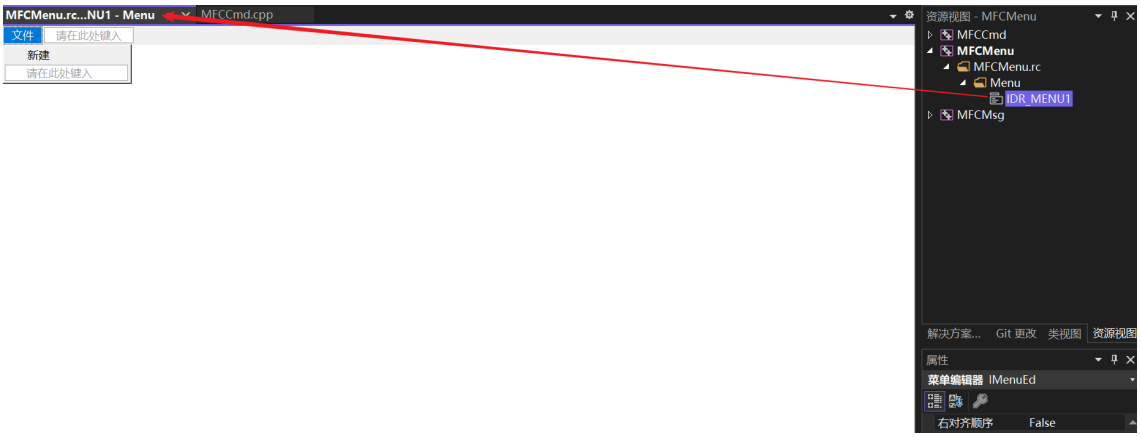
   首先，添加资源文件，在示例中我们命名其为：MFCMenu.rc



   之后，在“资源窗口”中，右击MFCMenu.rc文件，选择“添加资源”，选择“Menu”选项表示添加菜单资源：



双击新建的菜单资源，即可可视化地编辑菜单；在示例中，我们在 “文件” 选项下建立了一个新的菜单选项（下拉式菜单）—— “新建”，并将其ID更改为 `ID_NEW`，而顶部菜单的ID如右侧窗口所示为 `IDR_MENU1`：



2. **将菜单设置/挂到窗口上**，下面提供了两种方法：
   - 利用 `pFrame` 调用 `Create` 函数时，传参；
   - 在处理框架窗口的 `WM_CREATE` 消息时

```
CMenu menu;  // 一般将其设置为类内成员
menu.LodMenu(...);
// 调用SetMenu函数挂载菜单
```

3.5.2 挂载菜单代码示例

1. 将菜单资源设置到窗口上时：利用 `pFrame` 调用 `Create` 函数传参

```
/**
 * 项目名称: MFCMenu
 * 项目设置: win32 + Windows application + MFC
 * 课程示例: MFC菜单
*/

#include <afxwin.h>
#include "resource.h"

class CMyFrameWnd : public CFrameWnd {};
```

```
class CMyWinApp : public CWinApp {
public:
    virtual BOOL InitInstance() {
        CMyFrameWnd* pFrame = new CMyFrameWnd;
        pFrame->Creat(
            NULL,
            "MFCMenu",
            WS_OVERLAPPEDWINDOW,     // 窗口风格
            CFrameWnd::rectDefault, // 主框架窗口大小（宽/高）
            NULL,                    // 副窗口（没有）
            (CHAR*)IDR_MENU1         // 菜单：这里填写菜单的资源ID（IDR_MENU1）
        );
        m_pMainWnd = pFrame;
        pFrame->ShowWindow(SW_SHOW);
        pFrame->UpdateWindow();
        return TRUE;
    }
};

CMyWinApp theApp;
```

2. 将菜单资源设置到窗口上时：在处理框架窗口的 `WM_CREATE` 消息时添加两行代码；注意：需要将 `CMenu` 的对象设置为类内成员变量，再在处理框架窗口的 `WM_CREATE` 消息时调用其对象进行绑定操作，否则在程序运行时将会报错

```
/**
 * 项目名称: MFCMenu
 * 项目设置: win32 + Windows application + MFC
 * 课程示例: MFC菜单
*/

#include <afxwin.h>
#include "resource.h"

class CMyFrameWnd : public CFrameWnd {
    DECLARE_MESSAGE_MAP()
public:
    int OnCreate(LPCREATESTRUCT pcs) {
        this->menu.LodMenu(IDR_MENU1);  // 把菜单对象和菜单句柄建立绑定关系（这里绑定的是顶部菜单）
        this->SetMenu(&menu);  // 挂载菜单
        //::setMenu(this->m_hWnd, this->menu.m_hMenu);  // 使用WIN32也挂载菜单
        return CFrameWnd::OnCreate(pcs);
    }
public:
    CMenu menu;  // 延长menu的生命周期，况且菜单本身就是框架的一部分
};

BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
    ON_WM_CREATE()  // 处理窗口创建消息
END_MESSAGE_MAP()

class CMyWinApp : public CWinApp {
public:
```

```
    virtual BOOL InitInstance() {
        CMyFrameWnd* pFrame = new CMyFrameWnd;
        pFrame->Creat(NULL, "MFCMenu");
        m_pMainWnd = pFrame;
        pFrame->ShowWindow(SW_SHOW);
        pFrame->UpdateWindow();
        return TRUE;
    }
};

CMyWinApp theApp;
```

菜单对象和菜单句柄建立绑定关系详解略过，该操作对应上述代码中的：`this->menu.LodMenu(IDR_MENU1);`

3.5.3 菜单消息的处理

若不处理点击菜单所触发的消息，则对应的菜单项在运行时是灰色的，这表示该菜单项没有对应的消息处理函数

若想处理点击菜单所触发的消息，则需要利用宏——`ON_COMMAND( 菜单项ID, 处理消息的函数名 )` 向静态数组中扔一个COMMAND消息对应的消息处理函数，以下为对应的代码片段：

```
/*-------------------- 类外消息宏 --------------------*/
BEGIN_MESSAGE_MAP( CMyFrameWnd, CFrameWnd )
    ON_COMMAND( ID_NEW, OnNew )
END_MESSAGE_MAP( )

/*-------------------- CMyFrameWnd类内消息处理函数声明 --------------------*/
afx_msg void OnNew( );

/*-------------------- CMyFrameWnd类外消息处理函数实现 --------------------*/
void CMyFrameWnd::OnNew( ){
    // 点击菜单项后所需进行的操作
    AfxMessageBox( "框架类处理了新建菜单项被点击" );
}
```

3.5.4 菜单消息的处理顺序

对于菜单的处理，之前的示例均在框架类 `CMyFrameWnd` 中处理 `WM_COMMAND` 消息，其实也可以在应用程序类 `CMyWinApp` 中处理 `WM_COMMAND` 消息，只需要也在类内添加声明宏、类外添加实现宏即可；

但是注意：应用程序类仅能够处理 `WM_COMMAND` 类型的消息，并且如果框架类和应用程序类均进行了消息处理的实现，程序会优先选择框架类进行处理，即：仅考虑 `WM_COMMAND` 消息时，框架类 > 应用程序类

3.5.5 设置菜单项状态

设置菜单项状态指的是，例如：在菜单项前打勾、设置菜单项为不可用

如果希望设置菜单项状态，则需要利用宏 `ON_WM_INITMENUPOPUP()` ，该宏所对应的消息是在生成菜单的前一刻发出的，因此可以对各个菜单项进行修改，该宏对应的消息处理函数为，需要将其在框架类中实现：

```
/*-------------------- 类外消息宏 --------------------*/
BEGIN_MESSAGE_MAP( CMyFrameWnd, CFrameWnd )
    ON_WM_INITMENUPOPUP()
END_MESSAGE_MAP( )

/*-------------------- CMyFrameWnd类内消息处理函数声明 --------------------*/
afx_msg void OnInitMenuPopup( CMenu *pPopup,         // 菜单对象
                              UINT nPos,             // 点击的是哪个菜单项
                              BOOL i                 // 即将显示的菜单是不是窗口菜单
                            );

/*-------------------- CMyFrameWnd类外消息处理函数实现 --------------------*/
void CMyFrameWnd::OnInitMenuPopup( CMenu* pPopup, UINT nPos, BOOL i){
    pPopup->CheckMenuItem( ID_NEW, MF_CHECKED );  // 将“新建”菜单项设置为勾选状态
//  ::CheckMenuItem( pPopup->m_hMenu, ID_NEW, MF_CHECKED );
}
```

3.5.6 上下文菜单

“上下文菜单”又被称为“右键菜单”，如果希望显示一个上下文菜单，则需要处理宏 `ON_WM_CONTEXTMENU()`，这是MFC中专职显示上下文菜单的消息

```
/*-------------------- 类外消息宏 --------------------*/
BEGIN_MESSAGE_MAP( CMyFrameWnd, CFrameWnd )
    ON_WM_CONTEXTMENU()
END_MESSAGE_MAP( )

/*-------------------- CMyFrameWnd类内消息处理函数声明 --------------------*/
afx_msg void OnContextMenu( CWnd* pWnd,      // 点击的窗口框架指针
                            CPoint pt        // 点击的位置（通过XY坐标标识）
                          );

/*-------------------- CMyFrameWnd类外消息处理函数实现 --------------------*/
void CMyFrameWnd::OnContextMenu( CWnd* pWnd, CPoint pt ){
//  HMENU hPopup = ::GetSubMenu(menu.m_hMenu,0);
//  ::TrackPopupMenu( hPopup, TPM_LEFTALIGN|TPM_TOPALIGN, pt.x, pt.y,
//                                      0, this->m_hWnd, NULL );
    CMenu* pPopup = menu.GetSubMenu(0);  // 获取顶部菜单的下拉式菜单（子菜单）
    pPopup->TrackPopupMenu( TPM_LEFTALIGN|TPM_TOPALIGN,      // 光标在右键菜单的左上角
                            pt.x, pt.y,                      // 鼠标光标位置信息
                            this );                          //
}
```

最终的效果如下所示，在用户区右键即可出现刚刚设置的菜单项：


MFCCreate
文件
新建