MFC原理

MFC的大致框架

MFC的大致框架如下：

```
CMyObject
 |- CMyCmdTarget                    // 负责消息派发
 |    |- CMyWnd                     // 所有窗口的父类
 |    |    |- CMyView               // 所有视图的父类
 |    |    |    |- CMFCTestView
 |    |    |- CMyFrameWnd           // 所有框架的父类
 |    |    |    |- CMainFrame       // 本实例框架
 |    |- CMyWinThread               // 所有线程相关的父类，负责消息循环
 |    |    |- CMyWinApp             // 应用程序的父类
 |    |    |    |- CMFCTestApp      // 本应用程序实例
 |    |- CMyDocument                // 所有文档的父类，存储数据
 |    |    |- CMFCTestDoc
 |    |- CDocTemplate               // 文档模板
 |    |    |- CSingleDocTemplate    // 单文档模板
 |- CDocManager                     // 文档管理器
```

入口main

```
// 全局对象 - 应用程序实例
CMFCTestApp theApp;

int main(void)
{
    int nReturnCode = -1;
    CMyWinThread *pThread = &theApp;
    CMyWinApp *pApp = &theApp;

    // 利用多态，初始化实例
    if(!pThread->InitInstance()) {
        // 反初始化，执行清理操作
        return pThread->ExitInstance();
    }

    // 利用多态，进入消息循环
    nReturnCode = pThread->Run();
    return nReturnCode;
}
```

利用全局对象的多态特性，接管流程，执行初始化操作（注册窗口类并创建显示更新窗口），而后进入消息循环

窗口的六要素在MFC中的体现

1. 首先子类`CMFCTestApp`实现父类的虚函数`InitInstance`，来执行初始化操作
    - 创建主框架类`CMainFrame`，负责创建显示更新窗口

```
BOOL CMFCTestApp::InitInstance()
{
    CMainFrame* pFrame = new CMainFrame();  // 创建主框架类
    pFrame->LoadFrame(IDR_MENU1);   // 创建窗口

    m_pMainWnd = pFrame;

    // 主框架负责显示更新窗口
    m_pMainWnd->ShowWindow(SW_SHOW);
    m_pMainWnd->UpdateWindow();

    return TRUE;
}
```

2. 框架类`CMyFrameWnd`的`LoadFrame`方法（必要时子类`CMainFrame`可重写）来执行注册窗口类，并调用`CMyFrameWnd`的`Create`方法创建窗口

```
BOOL CMyFrameWnd::LoadFrame(UINT nIDResource, DWORD dwDefaultStyle, CMyWnd
*pParentWnd, CCreateContext *pContext)
{
    WNDCLASS wc;

    // Register the main window class.
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = (WNDPROC)MainWndProc;      // 额外提供一个窗口过程
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = GetModuleHandle(NULL);
    wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName = NULL;
    wc.lpszClassName = TEXT("MyWndClass");

    if (!RegisterClass(&wc))
        return FALSE;

    LPCTSTR lpszClass = TEXT("MyWndClass");
    LPCTSTR strTitle = TEXT("Hello World");
    RECT rectDefault = { 0, 0, CW_USEDEFAULT, CW_USEDEFAULT };
    if (!Create(lpszClass, strTitle, dwDefaultStyle, rectDefault,
pParentWnd, MAKEINTRESOURCE(nIDResource), 0L, pContext)) {
        return FALSE;   // will self destruct on failure normally
    }

    return TRUE;
}


BOOL CMyFrameWnd::Create(LPCTSTR lpszClassName, LPCTSTR lpszWindowName,
DWORD dwStyle, const RECT &rect,
    CMyWnd *pParentWnd, LPCTSTR lpszMenuName, DWORD dwExStyle,
CCreateContext *pContext)
```

```
{
    HMENU hMenu = NULL;
    if (lpszMenuName != NULL) {
        // load in a menu that will get destroyed when window gets destroyed
        HINSTANCE hInst = ::GetModuleHandle(NULL);
        if ((hMenu = ::LoadMenu(hInst, lpszMenuName)) == NULL) {
            return FALSE;
        }
    }

    if (!CreateEx(dwExStyle, lpszClassName, lpszWindowName, dwStyle,
rect.left, rect.top, rect.right - rect.left, rect.bottom - rect.top,
        pParentWnd->GetSafeHwnd(), hMenu, (LPVOID)pContext))  {
        if (hMenu != NULL)
            ::DestroyMenu(hMenu);
        return FALSE;
    }

    return TRUE;
}
```

3. 框架类 `CMyFrameWnd` 的 `Create` 方法又调用 `CMyWnd` 的 `CreateEx` 来创建窗口

```
BOOL CMyWnd::CreateEx(DWORD dwExStyle, LPCTSTR lpszClassName, LPCTSTR
lpszWindowName, DWORD dwStyle, int x, int y,
    int nWidth, int nHeight, HWND hWndParent, HMENU nIDorHMenu, LPVOID
lpParam)
{
    CREATESTRUCT cs;
    cs.dwExStyle = dwExStyle;
    cs.lpszClass = lpszClassName;
    cs.lpszName = lpszWindowName;
    cs.style = dwStyle;
    cs.x = x;
    cs.y = y;
    cs.cx = nWidth;
    cs.cy = nHeight;
    cs.hwndParent = hWndParent;
    cs.hMenu = nIDorHMenu;
    cs.hInstance = ::GetModuleHandle(NULL);
    cs.lpCreateParams = lpParam;

    if (!PreCreateWindow(cs)) {
        return FALSE;
    }

    HWND hWnd = ::CreateWindowEx(cs.dwExStyle, cs.lpszClass,
        cs.lpszName, cs.style, cs.x, cs.y, cs.cx, cs.cy,
        cs.hwndParent, cs.hMenu, cs.hInstance, cs.lpCreateParams);

    if (hWnd == NULL)
        return FALSE;

    m_hWnd = hWnd;
    return TRUE;
}
```

4. 然后返回到`main`中，调用`CCMyWinThread`的`Run`方法（必要时子类可重写）进入消息循环

```
int CMyWinThread::Run()
{
    while(true) {
        do {
            if (!PumpMessage())
                return 0;
        } while(::PeekMessage(&m_msgCur, NULL, NULL, NULL, PM_NOREMOVE));
    }
}

BOOL CMyWinThread::PumpMessage()
{
    int ret = ::GetMessage(&m_msgCur, NULL, 0, 0);
    if(ret == 0 || ret == -1) {
        return FALSE;
    }

    ::TranslateMessage(&m_msgCur);
    ::DispatchMessage(&m_msgCur);
    return TRUE;
}
```

RTTI

RTTI(run-time type information, 运行时类型识别)，通过运行时类型信息程序能够使用基类的指针或引用来检查这些指针或引用所指的对象的实际派生类型。

用途：

1. 类型检查
2. 动态创建

目的：使用给定的字符串,，创建一个对应类的对象

1. 识别对象的类名
2. 通过一个未知的对象创建新的对象
3. 使用给定的字符串，创建一个对应类的对象

CRunTimeClass类

```
struct AFX_CLASSINIT
{
};

struct CRuntimeClass
{
    LPCSTR m_lpszClassName;      // 类名
    int m_nObjectSize;           // 大小
    UINT m_wSchema;              // 版本
    CMyObject* (*m_pfnCreateObject)();
    CRuntimeClass* m_pBaseClass;     // 基类
```

```
    CRuntimeClass* m_pNextClass;    // 注册的类的链表
    const AFX_CLASSINIT* m_pClassInit;
};
```

RunTime对象，保存运行时信息

相关宏

```
/*
没有动态创建需求，声明时使用(在.h文件中)
    1. 为类添加静态CRuntimeClass类对象成员
    2. 为类添加GetRuntimeClass虚函数
*/
#define DECLARE_DYNAMIC(class_name) \
public: \
    static const CRuntimeClass class##class_name; \
    virtual CRuntimeClass* GetRuntimeClass() const;


// 获取类的静态CRuntimeClass类对象
#define _RUNTIME_CLASS(class_name) ((CRuntimeClass*)
(&class_name::class##class_name))
#define RUNTIME_CLASS(class_name) _RUNTIME_CLASS(class_name)

/*
没有动态创建需求
    1. 初始化静态CRuntimeClass类对象
    2. 实现GetRuntimeClass虚函数
*/
#define IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, wSchema, pfnNew,
class_init) \
    const CRuntimeClass class_name::class##class_name = { \
        #class_name, sizeof(class class_name), wSchema, pfnNew, \
            RUNTIME_CLASS(base_class_name), NULL, class_init }; \
    CRuntimeClass* class_name::GetRuntimeClass() const \
        { return RUNTIME_CLASS(class_name); }

// 无动态创建需求，实现定义时使用（在.cpp文件中）
#define IMPLEMENT_DYNAMIC(class_name, base_class_name) \
    IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, 0xFFFF, NULL, NULL)

/*
有动态创建需求，声明时使用(在.h文件中)
    1. 额外添加一个创建对象的静态函数
*/
#define DECLARE_DYNCREATE(class_name) \
    DECLARE_DYNAMIC(class_name) \
    static CMyObject *  CreateObject();

/*
有动态创建需求，实现定义时使用（在.cpp文件中）
    1. 额外实现创建对象静态函数
*/
#define IMPLEMENT_DYNCREATE(class_name, base_class_name) \
    CMyObject*  class_name::CreateObject() \
```

```
        { return new class_name; } \
    IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, 0xFFFF,
class_name::CreateObject, NULL)
```

相关宏的使用

在类的定义和实现中：

```
/************ 在CMyObject.h中 ************/
class CMyObject
{
public:
    // ...

    // 添加以下内容
    virtual CRuntimeClass* GetRuntimeClass() const;     // 定义函数接口
    const static CRuntimeClass classCMyObject;          // 定义静态对象
};

/************ 在CMyObject.cpp中 ************/
// 初始化静态对象
const struct CRuntimeClass CMyObject::classCMyObject = { "CMyObject",
sizeof(CMyObject),
                                                         0xffff, NULL, NULL, NULL
};
实现函数
CRuntimeClass * CMyObject::GetRuntimeClass() const
{
    return (CRuntimeClass *)&CMyObject::classCMyObject;
}


/************ 在xxx.h中 ************/
class xxx : public CMyObject {
    DECLARE_DYNAMIC(xxx)    // 如果需要动态创建，则替换为DECLARE_DYNCREATE(xxx)
public:
    // ...
}

/************ 在xxx.cpp中 ************/
#include "xxx.h"
IMPLEMENT_DYNAMIC(xxx, CMyObject)   // 如果需要动态创建，则替换为
IMPLEMENT_DYNCREATE(xxx, CMyObject)
// ...
```

使用遍历Runtime：

```
CRuntimeClass* pRuntimeClass = RUNTIME_CLASS(CMainFrame);       // 获取CMainFrame
的RuntimeClass
CMyObject *pObj = pRuntimeClass->m_pfnCreateObject();           // 动态创建
CMainFrame对象
printf("%s\n", pObj->GetRuntimeClass()->m_lpszClassName);       // 输出CMainFrame
对象的类名
```

```
// 遍历其基类的RuntimeClass
while (pRuntimeClass != NULL)
{
    printf("%s->", pRuntimeClass->m_lpszClassName);
    pRuntimeClass = pRuntimeClass->m_pBaseClass;
}
printf("NULL\n");

/*
打印结果:
    CMainFrame
    CMainFrame->CMyFrameWnd->CMyWnd->CMyCmdTarget->CMyObject->NULL
*/
```

改进窗口的创建

单文档模板

单文档模板`CSingleDocTemplate`继承自文档模板抽象类`CDocTemplate`

其抽象父类保存着：资源与文档类、框架类、视图类的RuntimeClass，拥有`OpenDocumentFile`接口，子类重写此接口用于完成窗口创建

```
CMyDocument * CSingleDocTemplate::OpenDocumentFile(LPCTSTR lpszPathName, BOOL
bAddToMRU, BOOL bMakeVisible)
{
    CMyDocument* pDocument = NULL;
    CMyFrameWnd* pFrame = NULL;
    BOOL bCreated = FALSE;      // => doc and frame created
    BOOL bWasModified = FALSE;

    // create a new document
    pDocument = (CMyDocument*)m_pDocClass->m_pfnCreateObject();
    if (pDocument == NULL) {
        return NULL;
    }
    m_pOnlyDoc = pDocument;

    CCreateContext context;
    context.m_pCurrentFrame = NULL;
    context.m_pCurrentDoc = pDocument;
    context.m_pNewViewClass = m_pViewClass;
    context.m_pNewDocTemplate = this;

    // 创建框架，并加载框架
    pFrame = pFrame = (CMyFrameWnd*)m_pFrameClass->m_pfnCreateObject();
    if (!pFrame->LoadFrame(m_nIDResource, WS_OVERLAPPEDWINDOW | FWS_ADDTOTITLE,
NULL, &context)) {
        return NULL;
    }

    CMyWinThread* pThread = AfxGetThread();
    pThread->m_pMainWnd = pFrame;
```

```
    return pDocument;
}
```

文档管理器

通过链表管理文档模板，拥有添加文档模板`AddDocTemplate`和打开文档`OnFileNew`的功能

```
void CDocManager::OnFileNew()
{
    if (m_templateList.empty()) {
        return;
    }

    // 获取文档模板并打开
    CDocTemplate *pTemplate = m_templateList.front();
    pTemplate->OpenDocumentFile(NULL);
}
```

改进InitInstance

```
BOOL CMFCTestApp::InitInstance()
{
    // 创建单文档模板
    CSingleDocTemplate* pDocTemplate;
    pDocTemplate = new CSingleDocTemplate(
        IDR_MENU1,
        RUNTIME_CLASS(CMFCTestDoc),
        RUNTIME_CLASS(CMainFrame),       // 主 SDI 框架窗口
        RUNTIME_CLASS(CMFCTestView));

    if (!pDocTemplate)
        return FALSE;

    // 添加文档模板
    AddDocTemplate(pDocTemplate);

    // 通过文档管理器打开文档
    OnFileNew();

    // 显示更新窗口
    m_pMainWnd->ShowWindow(SW_SHOW);
    m_pMainWnd->UpdateWindow();

    return TRUE;
}

void CMyWinApp::OnFileNew()
{
    if (m_pDocManager != NULL)
        m_pDocManager->OnFileNew();
}
```

利用RTTI通过字符串创建对象

- 实现`AFX_CLASSINIT`的构造和析构

```
std::list<CRuntimeClass*> *g_pClassList;    // 全局链表，保存注册的
CRuntimeClass类对象

AFX_CLASSINIT::AFX_CLASSINIT(CRuntimeClass *pNewClass)
{
    // 全局链表不存在，创建链表
    if (g_pClassList == NULL)
        g_pClassList = new std::list<CRuntimeClass*>();

    //增加链表
    g_pClassList->push_back(pNewClass);
}

AFX_CLASSINIT::~AFX_CLASSINIT()
{
    if (g_pClassList != NULL) {
        delete g_pClassList;
        g_pClassList = NULL;
    }
}
```

- 为`CRuntimeClass`添加静态函数`MyCreateObject`并实现

```
CMyObject * CRuntimeClass::MyCreateObject(const char *pszClassName)
{
    CMyObject* pObject = NULL;
    // 遍历全局链表，并比较类名
    for (CRuntimeClass* pRuntimeClass : *g_pClassList) {
        if (strcmp(pRuntimeClass->m_lpszClassName, pszClassName) == 0) {
 // 类名匹配则创建对象
            return (*pRuntimeClass->m_pfnCreateObject)();
        }
    }
    return pObject;
}
```

消息响应

消息响应封装，需要统一消息派发，不管是主窗口还是子窗口，所以需要hook拦截消息，然后派发

为了拦截全部消息（在窗口创建时不漏拦消息），需要在`CreateWindowEx`前进行hook，然后修改窗口回调，接着在`CreateWindowEx`后取消hook

```
BOOL CMyWnd::CreateEx(DWORD dwExStyle, LPCTSTR lpszClassName, LPCTSTR
lpszWindowName, DWORD dwStyle, int x, int y,
    int nWidth, int nHeight, HWND hWndParent, HMENU nIDorHMenu, LPVOID lpParam)
{
```

```
    // ...

    // 此处hook WH_CBT
    g_hHookOldCbtFilter = ::SetWindowsHookEx(WH_CBT, _AfxCbtFilterHook, NULL,
::GetCurrentThreadId());
    g_pWndInit = this;

    HWND hWnd = ::CreateWindowEx(cs.dwExStyle, cs.lpszClass, cs.lpszName,
cs.style, cs.x, cs.y, cs.cx, cs.cy,
        cs.hwndParent, cs.hMenu, cs.hInstance, cs.lpCreateParams);

    // 此处unhook
    ::UnhookWindowsHookEx(g_hHookOldCbtFilter);

    // ...
}
```

hook函数在是`HCBT_CREATEWND`时执行工作：除了输入法，其他窗口一律修改其窗口过程让其为`AfxWndProc`函数，在`AfxWndProc`中进行消息的派发

```
LRESULT CALLBACK AfxWndProc(HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam)
{
    CMyWnd* pWnd = CMyWnd::FromHandlePermanent(hWnd);

    //调用旧的过程函数
    if (pWnd == NULL)
        return ::DefWindowProc(hWnd, nMsg, wParam, lParam);

    return pWnd->WindowProc(nMsg, wParam, lParam);
}

inline int AfxInvariantStrICmp(const char* pszLeft, const char* pszRight)
{
    return ::CompareStringA(LOCALE_INVARIANT, NORM_IGNORECASE, pszLeft, -1,
pszRight, -1) - CSTR_EQUAL;
}


LRESULT CALLBACK _AfxCbtFilterHook(int code, WPARAM wParam, LPARAM lParam)
{
    // 不是窗口创建，放行
    if (code != HCBT_CREATEWND)
    {
        return CallNextHookEx(g_hHookOldCbtFilter, code, wParam, lParam);
    }

    HWND hWnd = (HWND)wParam;
    LPCREATESTRUCT lpcs = ((LPCBT_CREATEWND)lParam)->lpcs;
    WNDPROC oldWndProc;

    // 输入法放行
    if (g_pWndInit != NULL)
    {
        if (GetClassLong((HWND)wParam, GCL_STYLE) & CS_IME)
            goto lCallNextHook;

        LPCTSTR pszClassName;
```

```
        TCHAR szClassName[_countof("ime") + 1];
        if (DWORD_PTR(lpcs->lpszClass) > 0xffff)
        {
            pszClassName = lpcs->lpszClass;
        } else
        {
            szClassName[0] = '\0';
#pragma warning(push)
#pragma warning(disable: 4302) // 'type cast' : truncation from 'LPCSTR' to
'ATOM'
            GlobalGetAtomName((ATOM)lpcs->lpszClass, szClassName,
_countof(szClassName));
#pragma warning(pop)
            pszClassName = szClassName;
        }

        // a little more expensive to test this way, but necessary...
        if (::AfxInvariantStrICmp(pszClassName, "ime") == 0)
            goto lCallNextHook;
    }

    // 修改窗口过程
    char szClassName[MAXBYTE];
    ::GetClassName(hWnd, szClassName, sizeof(szClassName));
    if (g_pWndInit != NULL)
    {
        // 保存对象和窗口句柄的关系到表
        g_pWndInit->Attach(hWnd);

        // 修改窗口过程函数
        oldWndProc = (WNDPROC)SetWindowLongPtr(hWnd,
            GWLP_WNDPROC,
            (DWORD_PTR)AfxWndProc);

        // 保存旧的窗口过程函数
        if (oldWndProc != AfxWndProc)
            g_pWndInit->m_pfnOldProc = oldWndProc;

        // 绑定窗口句柄和对象
        g_pWndInit = NULL;
    }

lCallNextHook:
    return CallNextHookEx(g_hHookOldCbtFilter, code, wParam, lParam);
}
```

窗口过程为`CMyWnd::WindowProc`

```
LRESULT CMyWnd::WindowProc(UINT message, WPARAM wParam, LPARAM lParam)
{
    int result = -1;
    switch (message) {
    case WM_CREATE:
    {
        LPCREATESTRUCT lpCreate = (LPCREATESTRUCT)lParam;
        result = this->OnCreate(lpCreate);
        break;
```

```
    }
    case WM_CLOSE:
    {
        result = this->OnClose();
        break;
    }
    case WM_KEYDOWN:
    {
        result = this->OnKeyDown(wParam, 0, 0);
        break;
    }
    default:
        break;
    }

    return m_pfnOldProc(m_hWnd, message, wParam, lParam);
}
```

`CMyWnd` 类实现各个消息相应的虚函数，例如：`OnCreate`，其子类可在需要时重写

```
int CMyWnd::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    return 0;
}

int CMyWnd::OnClose()
{
    ::PostQuitMessage(0);
    return 0;
}
```

WM_COMMAND的处理

在WM_COMMAND消息中，因为控件ID众多，不得不以switch去实现，造成switch部分的代码始终是需要变动的。针对此情况，需要改为消息映射的方式

在消息派发 `CMyCmdTarget` 中定义相关数据结构和宏

```
typedef void (CMyCmdTarget::* AFX_PMSG)(void);
typedef void (CMyWnd::* AFX_PMSGW)(void);

enum AfxSig
{
    AfxSig_end = 0,     // [marks end of message map]
    AfxSig_i_PC,        // int (LPCREATESTRUCT )
    AfxSig_v_v,         // void (void )
};

union MessageMapFunctions
{
    AFX_PMSG pfn;   // generic member function pointer

    int  (CMyCmdTarget::*pfn_i_PC)(LPCREATESTRUCT);
    void (CMyCmdTarget::* pfn_v_v)(void);
```

```
};

// 消息映射项目
struct AFX_MSGMAP_ENTRY
{
    UINT nMessage;   // 消息
    UINT nCode;      // 通告码
    UINT nID;        // 起始ID
    UINT nLastID;    // 结束ID
    UINT_PTR nSig;   // 标识参数信息
    AFX_PMSG pfn;    // 函数指针，通过nSig来确定强转
};

// 消息映射
struct AFX_MSGMAP
{
    const AFX_MSGMAP* (*pfnGetBaseMap)();   // 获取基类的消息映射
    const AFX_MSGMAP_ENTRY* lpEntries;
};

/*
 *定义消息映射
 */
#define DECLARE_MESSAGE_MAP() \
protected: \
    static const AFX_MSGMAP*  GetThisMessageMap(); \
    virtual const AFX_MSGMAP* GetMessageMap() const;


/*
 * 实现消息映射
 *  Begin和End成对使用
 */
#define BEGIN_MESSAGE_MAP(theClass, baseClass) \
    const AFX_MSGMAP* theClass::GetMessageMap() const \
        { return GetThisMessageMap(); } \
    const AFX_MSGMAP*  theClass::GetThisMessageMap() \
    { \
        static const AFX_MSGMAP_ENTRY _messageEntries[] =  \
        {


#define END_MESSAGE_MAP(TheBaseClass) \
        {0, 0, 0, 0, AfxSig_end, (AFX_PMSG)0 } \
    }; \
        static const AFX_MSGMAP messageMap = \
        { &TheBaseClass::GetThisMessageMap, &_messageEntries[0] }; \
        return &messageMap; \
    }
```

实现消息派发`CMyCmdTarget`的获取消息映射：`GetThisMessageMap`和`GetMessageMap`

```
const AFX_MSGMAP * CMyCmdTarget::GetThisMessageMap()
{
    // 消息映射项目
    static const AFX_MSGMAP_ENTRY _messageEntries[] =
    {
```

```
        { 0, 0, AfxSig_end, 0 }     // nothing here
    };
    // 消息映射
    static const AFX_MSGMAP messageMap =
    {
        NULL,
        &_messageEntries[0]
    };
    return &messageMap;
}

const AFX_MSGMAP * CMyCmdTarget::GetMessageMap() const
{
    return GetThisMessageMap();
}
```

在 `CMyWnd`、`CMyFrameWnd` 和 `CMainFrame` 中定义消息映射关系，例如：

```
/************ 在CMyWnd.h中 ************/
class CMyWnd :
    public CMyCmdTarget
{
    // ...
    DECLARE_MESSAGE_MAP()
    // ...
}


/************ 在CMyWnd.cpp中 ************/
#include "CMyWnd.h"
BEGIN_MESSAGE_MAP(CMyWnd, CMyCmdTarget)
{WM_CREATE, 0, 0, 0, AfxSig_i_PC, (AFX_PMSG)(AFX_PMSGW)&CMyWnd::OnCreate},  //
消息映射
END_MESSAGE_MAP(CMyCmdTarget)
```

修改 `CMyWnd` 的窗口过程并添加 `OnMsg` 消息派发函数

```
LRESULT CMyWnd::WindowProc(UINT message, WPARAM wParam, LPARAM lParam)
{
    LRESULT lResult = 0;
    if (!OnWndMsg(message, wParam, lParam, &lResult))
        lResult = m_pfnOldProc(m_hWnd, message, wParam, lParam);
    return TRUE;
}

BOOL CMyWnd::OnWndMsg(UINT message, WPARAM wParam, LPARAM lParam, LRESULT
*pResult)
{
    const AFX_MSGMAP* pMsgMap = GetMessageMap();
    const AFX_MSGMAP_ENTRY* lpEntries = NULL;
    while (pMsgMap != NULL)
    {
        //遍历消息映射表
        lpEntries = pMsgMap->lpEntries;
        while (lpEntries->nSig != AfxSig_end)
        {
```

```
            if (message == WM_COMMAND)
            {
                WORD wNotifyCode = HIWORD(wParam);
                WORD wID = LOWORD(wParam);

                //判断消息
                if (lpEntries->nMessage == message && lpEntries->nCode ==
wNotifyCode && (wID >= lpEntries->nID  && wID <= lpEntries->nLastID))
                {
                    goto DISPATCH;

                }
            } else
            {
                //判断消息
                if (lpEntries->nMessage == message)
                {
                    goto DISPATCH;
                }
            }

            lpEntries++;
        }
        if (pMsgMap->pfnGetBaseMap == NULL)
            break;

        pMsgMap = (*pMsgMap->pfnGetBaseMap)();
    }
    return FALSE;

DISPATCH:
    MessageMapFunctions mmf;
    mmf.pfn = lpEntries->pfn;
    switch (lpEntries->nSig)
    {
    case AfxSig_i_PC:
        (this->*mmf.pfn_i_PC)((LPCREATESTRUCT)lParam);
        break;
    case AfxSig_v_v:
        (this->*mmf.pfn_v_v)();
        break;
    }

    return TRUE;
}
```

这样，在需要时即可使用宏`DECLARE_MESSAGE_MAP`、`BEGIN_MESSAGE_MAP`和`END_MESSAGE_MAP`来实现消息映射

逆向MFC框架

在MFC框架中，每一个窗口句柄`hwnd`都绑定了一个`CWnd *`指针，通过`FromHandlePermanent`可获取`CWnd`对象，也就可以通过对象获得虚表结构、RTTI信息（继承层次）和消息映射（`GetMessageMap`函数）

1. 找到`FromHandlePermanent`函数
    - 静态的MFC时，可以利用特征码进行搜索
2. 如何确定是否是MFC窗口程序
    - 向窗口`SendMessage`发送`#define WM_QUERYAFXWNDPROC 0x360`的消息，如果返回1，则是MFC窗口，此消息在MFC的窗口过程中有用途

```
LRESULT CALLBACK AfxWndProc(HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM
lParam)
{
    if(nMsg == WM_QUERYAFXWNDPROC)
        return 1;
    CWnd *PWnd = CWnd::FromHandlePermanent(hWnd);   // 关键函数
    // ...
}
```

思路：向目标程序进行注入（不能是远程线程注入，因为MFC会利用TLS保存相关信息），找到`AfxWndProc`函数进行代码解析，获得`FromHandlePermanent`函数的地址，而后就可以将窗口句柄转为窗口对象，解析虚表结构，之后就可以得到RTTI和消息映射