2021/04/19_MFC原理_第6课_MFC的消息处理及实现

笔记本： MFC原理
创建时间： 2021/4/20 星期二 7:34
作者： ileemi

MFC的消息处理及实现

在MFC中，子窗口产生WM_COMMAND消息都有父窗口来接收。

注册消息处理函数（动态、静态）
静态：消息对应的处理函数在数据结构中不支持动态添加（固定的）
动态：消息对应的处理函数在数据结构中支持动态添加

MFC 采用静态注册，示例代码如下：

```
// 生成的消息映射函数
protected:
  DECLARE_MESSAGE_MAP()

#define DECLARE_MESSAGE_MAP() \
protected: \
  static const AFX_MSGMAP* PASCAL GetThisMessageMap(); \
  virtual const AFX_MSGMAP* GetMessageMap() const; \

// 消息表 -- 每个类应该都有一个
struct AFX_MSGMAP_ENTRY
{
  UINT nMessage;  // windows message
  UINT nCode;     // control code or WM_NOTIFY code
  UINT nID;       // control ID (or 0 for windows messages)
  UINT nLastID;   // used for entries specifying a range of control id's
  UINT_PTR nSig;  // signature type (action) or pointer to message #
  AFX_PMSG pfn;   // routine to call (or special value) 消息处理函数对应
的函数指针
};

//消息映射表
struct AFX_MSGMAP
{
```

```
    const AFX_MSGMAP* (PASCAL* pfnGetBaseMap)();
    const AFX_MSGMAP_ENTRY* lpEntries;
};

// 不同消息函数指针的参数存在差异，所以就需要进行区分
enum AfxSig
{
    AfxSig_end = 0, // [marks end of message map]
    AfxSig_B_P,
    AfxSig_V_V,
    AfxSig_B_WWH,
    // ...
};
```

> 消息映射表中支持控件批量处理（一定范围的控件同时由一个函数处理），MFC中对应的宏为： "ON_COMMAND_RANGE"。

窗口不处理消息，应该询问其基类是否处理。

不同消息函数指针的参数存在差异，所以就需要进行区分，在MFC中通过 "AfxSig_x_x_x" 来进行区分，如下图所示：

```
enum AfxSig
{
    AfxSig_end = 0,     // [marks end of message map]

    AfxSig_b_D_v,       // BOOL (CDC*)
    AfxSig_b_b_v,       // BOOL (BOOL)
    AfxSig_b_u_v,       // BOOL (UINT)
    AfxSig_b_h_v,       // BOOL (HANDLE)
    AfxSig_b_W_uu,        // BOOL (CWnd*, UINT, UINT)
    AfxSig_b_W_COPYDATASTRUCT,        // BOOL (CWnd*, COPYDATASTRUCT*)
    AfxSig_b_v_HELPINFO,    // BOOL (LPHELPINFO);
    AfxSig_CTLCOLOR,      // HBRUSH (CDC*, CWnd*, UINT)
    AfxSig_CTLCOLOR_REFLECT,  // HBRUSH (CDC*, UINT)
    AfxSig_i_u_W_u,       // int (UINT, CWnd*, UINT)  // ?TOITEM
    AfxSig_i_uu_v,        // int (UINT, UINT)
    AfxSig_i_W_uu,        // int (CWnd*, UINT, UINT)
    AfxSig_i_v_s,       // int (LPTSTR)
    AfxSig_l_w_l,       // LRESULT (WPARAM, LPARAM)
    AfxSig_l_uu_M,        // LRESULT (UINT, UINT, CMenu*)
    AfxSig_v_b_h,       // void (BOOL, HANDLE)
    AfxSig_v_h_v,       // void (HANDLE)
```

从 CCmdTarget 开始，为每个派生类都注册消息映射表，示例代码如下：

```
// .h
class CCmdTarget :public CObject
{
    DECLARE_DYNAMIC(CCmdTarget)
public:
    // 静态数组，派生类处理消息重写该数组即可
    static const AFX_MSGMAP_ENTRY _messageEntries[];
};

// .cpp
```

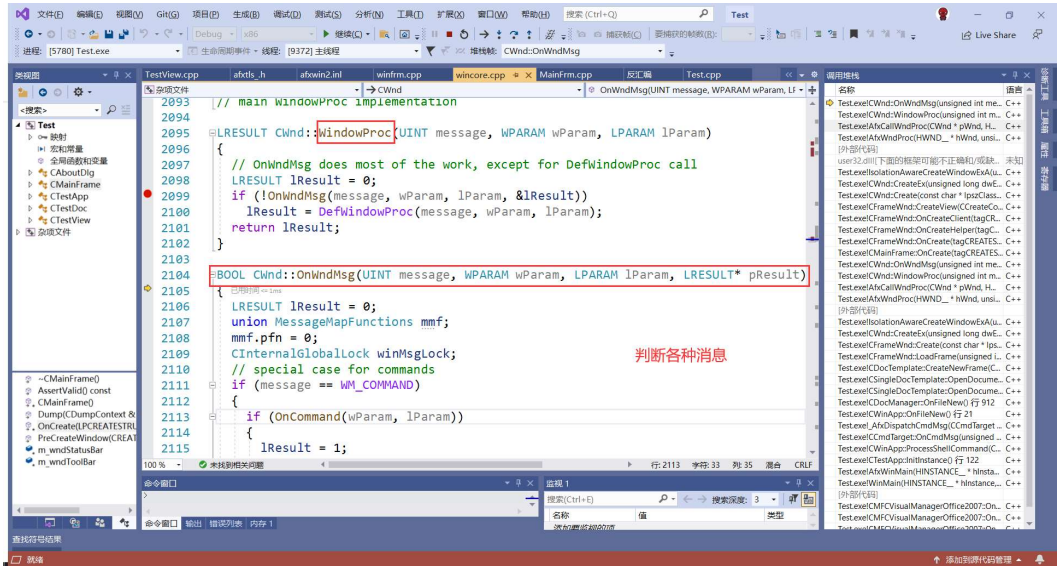
```
const AFX_MSGMAP_ENTRY CCmdTarget::_messageEntries[] = {
  0, 0, 0, 0, AfxSig_end, (AFX_PMSG)0, // 消息映射结束标志
};
```

```
1251    __pragma(warning(push))                         \
1252    __pragma(warning(disable: 4640)) /* message maps can only be called by single th
1253    static const AFX_MSGMAP_ENTRY _messageEntries[] =       \
1254    {
1255
1256  #define BEGIN_MESSAGE_MAP(theClass, baseClass) \
1257    PTM_WARNING_DISABLE \
1258    const AFX_MSGMAP* theClass::GetMessageMap() const \
1259      { return GetThisMessageMap(); } \
1260    const AFX_MSGMAP* PASCAL theClass::GetThisMessageMap() \
1261    { \
1262      typedef theClass ThisClass;                   \
1263      typedef baseClass TheBaseClass;               \
1264      __pragma(warning(push))                       \
1265      __pragma(warning(disable: 4640)) /* message maps can only be called by single th
1266      static const AFX_MSGMAP_ENTRY _messageEntries[] =  \
1267      {
1268
1269  #define END_MESSAGE_MAP() \
1270      {0, 0, 0, 0, AfxSig_end, (AFX_PMSG)0 } \
1271    }; \
```

不能将一个类的函数指针转换为另一个类（即使由继承层次），需要将其转换为基类的无函数无指针，之后在进行强转，示例如下：

```
// 消息映射
const AFX_MSGMAP_ENTRY CWnd::_messageEntries[] = {
  WM_CREATE, 0, 0, 0, AfxSig_B_P, (AFX_PMSG)(void (CWnd::*)
())&CWnd::OnCreate,
  0, 0, 0, 0, AfxSig_end, (AFX_PMSG)0, // 消息映射结束标志
};
```

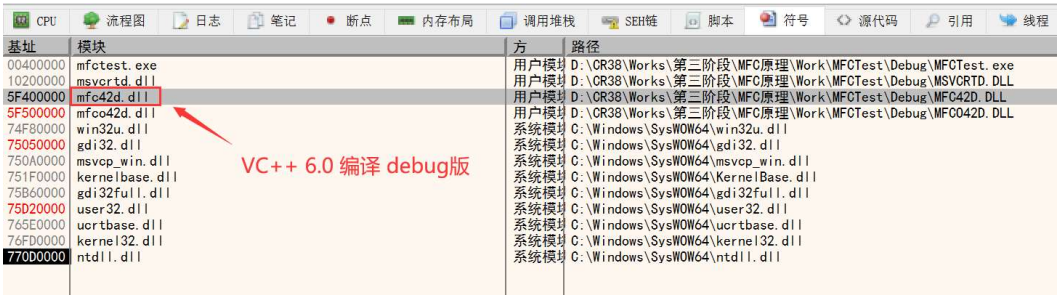
MFC 中的过程函数就是在遍历消息映射表（判断各种消息，做对应的处理），如下图所示：



> C++ Lambda表达式文档

逆向MFC程序

MFC dll的链接方式有动态链接和静态链接，动态链接（区分ASCII、Unicode、debug、release、单线程、多线程），MFC程序中会链接MFC dll 对应的版本号，方便知道程序的编译环境。静态链接时，程序就不会加载MFC 的 dll，也就无法分辨出程序的编译环境。



获取 RTTI 就可以获取程序中类的继承层次。

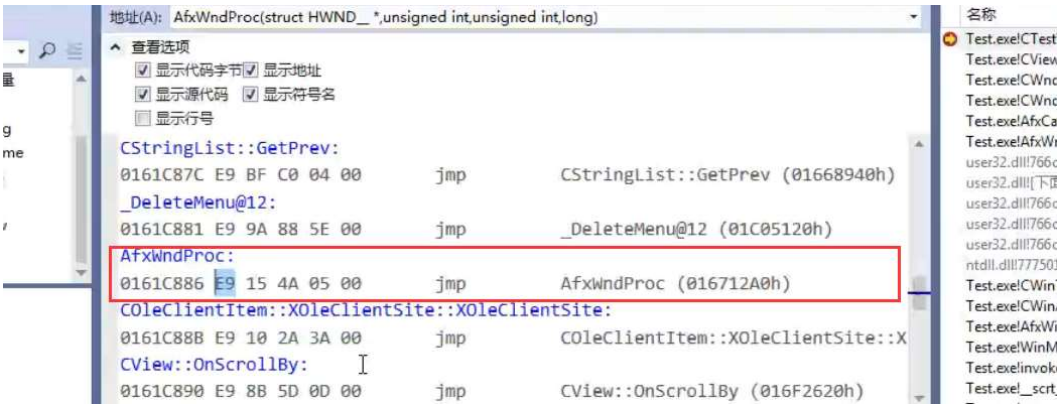
MFC 程序的识别可通过发送 "WM_QUERYAFXWNDPROC" 消息来判断。在 AfxWndProc 函数中返回1就说明该程序为MFC程序。

```
403
404   LRESULT CALLBACK
405  AfxWndProc(HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam)
406  {
407    // special message which identifies the window as using AfxWndProc
408    if (nMsg == WM_QUERYAFXWNDPROC)
409      return 1;
410
411    // all other messages route through message map
412    CWnd* pWnd = CWnd::FromHandlePermanent(hWnd);
413    ASSERT(pWnd != NULL);
414    ASSERT(pWnd==NULL || pWnd->m_hWnd == hWnd);
415    if (pWnd == NULL || pWnd->m_hWnd != hWnd)
416      return ::DefWindowProc(hWnd, nMsg, wParam, lParam);
417    return AfxCallWndProc(pWnd, hWnd, nMsg, wParam, lParam);
418  }
419
```

获取 CWnd 对象，MFC程序的框架中会有一个哈希表保存了窗口句柄和 CWnd 的关系。通过对MFC框架的了解，调用 FromHandlePermanent 函数就可以获取 CWnd 对象（需要在目标进程中走代码注入 -- 可通过GetWindowLong）。
CWnd* pWnd = CWnd::FromHandlePermanent(hWnd);

实现思路：

在目标进程中注入代码，获取过程函数，通过过程函数以及函数调用时的汇编特征码获取 "FromHandlePermanent" 函数的地址。需要注意程序编译的版本，是调试版还是发布版，调试版，在获取过程函数时，获取到的地址可能是未跳转前的地址（jmp 后的地址才是过程函数，需要判断操作码是否是 "E9"，是将获取的地址 + E9 后面的四字节的偏移）。



通过对象地址（pWnd）获取虚表（在对象首地址处）。通过RTTI还可以获取程序中类的继承层次。

虚表在对象首地址处。GetRuntimeClass 虚函数在基类中第一个出现，所以虚表的第1项就是 GetRuntimeClass。调用这个 GetRuntimeClass 通过遍历就可以获取当前类的继承层次。

通过 GetRuntimeClass 的结构可以看出静态链接的 MFC dll 和 动态链接的 MFC dll 存在差异性（结构不一样），如下图所示：

```
struct CRuntimeClass
{
// Attributes
   LPCSTR m_lpszClassName;
   int m_nObjectSize;
   UINT m_wSchema; // schema number of the loaded class
   CObject* (PASCAL* m_pfnCreateObject)(); // NULL => abstract class
#ifdef _AFXDLL
   CRuntimeClass* (PASCAL* m_pfnGetBaseClass)();
#else
   CRuntimeClass* m_pBaseClass;
#endif

// Operations
   CObject* CreateObject();
   BOOL IsDerivedFrom(const CRuntimeClass* pBaseClass) const;
```

判断静态链接或者动态链接可以判断函数指针 AfxWndProc 的地址是否在主模块中（在主模块中就是静态链接，反之动态链接，动态链接有明显的 MFC dll）。

获取消息映射表：

定位其虚表中的位置（需要从基类开始往所在类中进行定位），GetMessageMap 在 CCmdTarget 类的 "DECLARE_MESSAGE_MAP" 宏中。

在基类 CObject 中如果是调试版的dll会多两个虚函数，如下图所示：

```
// Overridables
   virtual void Serialize(CArchive& ar);

#if defined(_DEBUG) || defined(_AFXDLL)
   // Diagnostic Support
   virtual void AssertValid() const;
   virtual void Dump(CDumpContext& dc) const;
#endif

// Implementation
public:
   static const CRuntimeClass classCObject;
#ifdef _AFXDLL
   static CRuntimeClass* PASCAL _GetBaseClass();
   static CRuntimeClass* PASCAL GetThisClass();
#endif
```

逆向spy++是如何获取窗口句柄：定位特征，比如过程函数的地址值。在WinHex中进行搜索（内存中搜索可以存在多个位置），定位文件偏移，可动态调试加载程序。

使用静态分析，用IDA打开 spy++程序，定位 WindowFromPoint API，查看引用，F5 分析引用的代码。

```
.idata:004812B4                                         ; sub_447371+16↑p ...
.idata:004812B8 ; HWND __stdcall WindowFromPoint(POINT Point)
.idata:004812B8                 extrn WindowFromPoint:dword
.idata:004812B8                                         ; CODE XREF: sub_445D57+3A↑p
.idata:004812B8                                         ; DATA XREF: sub_445D57+3A↑r
.idata:004812BC ; HMENU __stdcall CreatePopupMenu()
.idata:004812BC                 extrn CreatePopupMenu:dword
```

xrefs to WindowFromPoint

Directio	Ty	Address	Text
Up	r	sub_445D57+3A	call ds:WindowFromPoint
Up	p	sub_445D57+3A	call ds:WindowFromPoint

Line 2 of 2

OK Cancel Search Help

: 接收到的消息异常，或格式不正确。
nished.

接着分析：

```
.idata:0048121C                                         ; DATA XREF: sub_447DBD:loc_447E9D↑r
.idata:00481220 ; const HOOKPROC SpyxxCallWndProc
.idata:00481220                 extrn _SpyxxCallWndProc@12:dword
.idata:00481220                                         ; DATA XREF: sub_447B8E+DD↑r
.idata:00481224                 extrn ghhkMsgHook:dword ; DATA XREF: sub_447B8E:loc_447C51↑r
.idata:00481228 ; const HOOKPROC SpyxxGetMsgProc
.idata:00481228                 extrn _SpyxxGetMsgProc@12:dword
.idata:00481228                                         ; DATA XREF: sub_447B8E+4E↑r
.idata:0048122C ; void *const gabMsgBuf
```

xrefs to SpyxxGetMsgProc(x,x,x)

Directio	Ty	Address	Text
Up	r	sub_447B8E+4E	push ds:_SpyxxGetMsgProc@12; lpfn

Line 1 of 1

OK Cancel Search Help

通过API可以断定spy++是通过注入代码获取过程函数地址的：

```
2  int __thiscall sub_447B8E(void *this)
3  {
4    HMODULE v1; // eax
5    HHOOK v3; // eax
6    CMFCRibbonInfo::XID *v4; // ecx
7    HHOOK v5; // eax
8    HHOOK v6; // ecx
9    LPCWSTR lpText; // [esp+10h] [ebp-14h] BYREF
10   LPCWSTR lpCaption[3]; // [esp+14h] [ebp-10h] BYREF
11   int v9; // [esp+20h] [ebp-4h]
12
13   if ( this )
14   {
15     v1 = hmod;
16     if ( !hmod )
17     {
18       v1 = GetModuleHandleW(L"SpyxxHk");
19       hmod = v1;
20       if ( !v1 )
21         return 0;
22     }
23     if ( !hhk )
24     {
25       v3 = SetWindowsHookExA(3, SpyxxGetMsgProc, v1, 0);
26       hhk = v3;
27       if ( !v3 )
28       {
```

是一个 dll

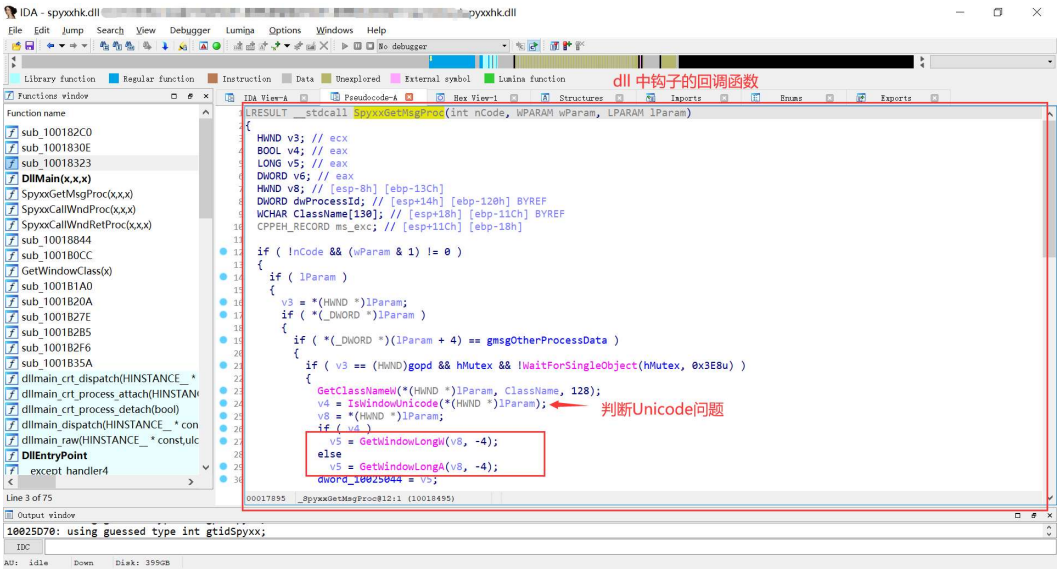
dll 中的导出函数，通过API可以断定spy++是通过注入代码获取过程函数地址的

spyxx.chm	2021/4/15 星期四 21:25	编译的 HTML 帮助文...	113 KB
spyxx.exe	2021/4/15 星期四 21:25	应用程序	679 KB
spyxx_amd64.chm	2021/4/15 星期四 21:25	编译的 HTML 帮助文...	113 KB
spyxx_amd64.exe	2021/4/15 星期四 21:25	应用程序	923 KB
spyxxhk.dll	2021/4/15 星期四 21:25	应用程序扩展	160 KB
spyxxhk_amd64.dll	2021/4/15 星期四 21:25	应用程序扩展	177 KB
ucrtbase.dll	2021/4/15 星期四 21:19	应用程序扩展	896 KB

IDA分析目标dll，定位调用的导出函数：

IDA View-A | Pseudocode-A | Hex View-1 | Structures | Imports | Enums | Exports

Name	Address	Ordinal
DllMain(x,x,x)	**10018377**	**1**
GetWindowClass(x)	1001B167	2
SpyxxCallWndProc(x,x,x)	100186AE	3
SpyxxCallWndRetProc(x,x,x)	10018778	4
SpyxxGetMsgProc(x,x,x)	10018495	5
gaaClasses	10025000	6
gabMsgBuf	10025178	7
gfHookDisabled	10025D5C	8
gfHookEnabled	10025174	9
ghhkCallHook	10025D68	10
ghhkMsgHook	10025D64	11
ghhkRetHook	10025D60	12
gmet	1001E9F0	13



使用 SetWindowsHookExA，操作系统会帮助注入目标dll到目标进程中。

下钩子防止消息不来，会在 SetWindowsHookExA 后使用 PostMessageW 发送一个消息到目标进程中。如下图所示：

```
.text:00447A2E                push    eax
.text:00447A2F                mov     edx, 320h
.text:00447A34                mov     ecx, offset unk_4804C0
.text:00447A39                call    sub_42AF40
.text:00447A3E                push    offset unk_4804C0 ; lParam
.text:00447A43                push    ebx             ; wParam
.text:00447A44                push    470h            ; Msg
.text:00447A49                push    ArgList         ; hWnd
.text:00447A4F                call    ds:PostMessageW
.text:00447A55
.text:00447A55 loc_447A55:                            ; CODE XREF: StartAddress+359↑j
.text:00447A55                mov     [ebp+pDacl], 1
.text:00447A5F
.text:00447A5F loc_447A5F:                            ; CODE XREF: StartAddress+32B↑j
.text:00447A5F                                        ; StartAddress+334↑j
.text:00447A5F                push    32h ; '2'       ; dwMilliseconds
.text:00447A61                push    hHandle         ; hHandle
.text:00447A67                mov     esi, ds:WaitForSingleObject
.text:00447A6D                call    esi ; WaitForSingleObject
.text:00447A6F                test    eax, eax
.text:00447A71                jnz     loc_447B09
.text:00447A77                push    32h ; '2'       ; dwMilliseconds
.text:00447A79                push    hMutex          ; hHandle
.text:00447A7F                call    esi ; WaitForSingleObject
.text:00447A81                test    eax, eax
.text:00447A83                jnz     short loc_447AF8
.text:00447A85                mov     eax, ds:gabMsgBuf
```