**MFC消息流动的内部实现**

2016年07月31日 22:18:53 [尚书左仆射](https://me.csdn.net/wzxq123) 阅读数：1571更多

个人分类： [C++](https://blog.csdn.net/wzxq123/article/category/6228735) [MFC](https://blog.csdn.net/wzxq123/article/category/6239417)

所属专栏： [MFC九阴真经](https://blog.csdn.net/column/details/wzxq-mfc-jyzj.html)

版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/wzxq123/article/details/52081211

消息的一生神秘而漫长，期间曲折多为人所不知。今天就让我们一起“撩”一下这位高冷的神秘妹纸吧。

事实上，MFC中利用hook技术，把看似无关的操作关联了起来。所谓hook（钩子）是Windows中一种高级的编程技术，它可以保证，在特定情况发生的时候就转去执行我们所指定的操作（是一种霸道机关术）。

MFC的hook发生在CWnd派生类对象的产生之际。在WINCORE.CPP中我们可以看到如下的代码：

1. BOOL CWnd::CreateEx(DWORD dwExStyle, LPCTSTR lpszClassName,
2. LPCTSTR lpszWindowName, DWORD dwStyle,
3. int x, int y, int nWidth, int nHeight,
4. HWND hWndParent, HMENU nIDorHMenu, LPVOID lpParam)
5. {
6. ……
7. AfxHookWindowCreate(this);//关键操作
8. HWND hWnd = ::CreateWindowEx(cs.dwExStyle, cs.lpszClass,
9. cs.lpszName, cs.style, cs.x, cs.y, cs.cx, cs.cy,
10. cs.hwndParent, cs.hMenu, cs.hInstance, cs.lpCreateParams);
11. ……
12. }

在WINCORE.CPP中又有：

1. voidAFXAPI AfxHookWindowCreate(CWnd\* pWnd)
2. {
3. ……
4. if (pThreadState->m\_hHookOldCbtFilter== NULL)
5. {
6. pThreadState->m\_hHookOldCbtFilter = ::SetWindowsHookEx(WH\_CBT,
7. \_AfxCbtFilterHook, NULL, ::GetCurrentThreadId());//关键操作
8. if (pThreadState->m\_hHookOldCbtFilter== NULL)
9. AfxThrowMemoryException();
10. }
11. ……
12. }

WH\_CBT是hook类型中的一种，它意味着安装一个Computer-Based- Training（CBT）滤网函数。安装后，Windows系统在进行一下任何一个操作之前都会先调用我们的滤网函数：

◎激活一个窗口（HCBT\_ACTIVATE）

◎产生或摧毁一个窗口（HCBT\_CREATEWND、HCBT\_DESTROYWND）

◎最大化或最小化一个窗口（HCBT\_MINMAX）

◎移动或缩放一个窗口（HCBT\_MOVESIZE）

◎完成一个来自系统菜单的系统命令（HCBT\_SYSTEMCOMMAND）

◎从系统队列中移去一个鼠标或键盘消息（HCBT\_KEYSKIPPED、HCBT\_CLICKSKIPPED）

因此，在上述hook安装之后，任何窗口产生之前，滤网函数\_AfxCbtFilterHook都会先被调用（WINCORE.CPP）：

1. LRESULT CALLBACK
2. \_AfxCbtFilterHook(int code, WPARAM wParam, LPARAM lParam)
3. {
4. \_AFX\_THREAD\_STATE\* pThreadState = \_afxThreadState.GetData();
5. if (code != HCBT\_CREATEWND)
6. {
7. // wait for HCBT\_CREATEWND just pass others on...
8. return CallNextHookEx(pThreadState->m\_hHookOldCbtFilter, code,
9. wParam, lParam);
10. }
11. ……
12. if (!afxData.bWin4 && !bContextIsDLL &&
13. (pCtl3dState = \_afxCtl3dState.GetDataNA()) != NULL &&
14. pCtl3dState->m\_pfnSubclassDlgEx != NULL &&
15. (dwFlags = AfxCallWndProc(pWndInit, hWnd, WM\_QUERY3DCONTROLS)) != 0)
16. {
17. // was the class registered with AfxWndProc?
18. WNDPROC afxWndProc = AfxGetAfxWndProc();
19. BOOL bAfxWndProc = ((WNDPROC)
20. GetWindowLong(hWnd, GWL\_WNDPROC) == afxWndProc);
22. pCtl3dState->m\_pfnSubclassDlgEx(hWnd, dwFlags);
24. // subclass the window if not already wired to AfxWndProc
25. if (!bAfxWndProc) //若窗口处理函数不是AfxWndProc则设置之
26. {
27. // subclass the window with standard AfxWndProc
28. oldWndProc = (WNDPROC)SetWindowLong(hWnd, GWL\_WNDPROC,(DWORD)afxWndProc);
29. ASSERT(oldWndProc != NULL);
30. \*pOldWndProc = oldWndProc;
31. }
32. }
33. ……
34. }

我们可以看到AfxGetAfxWndProc函数的操作如下（WINCORE.CPP）：

1. WNDPROC AFXAPI AfxGetAfxWndProc()
2. {
3. #ifdef \_AFXDLL//为DLL时
4. return AfxGetModuleState()->m\_pfnAfxWndProc;
5. #else
6. return &AfxWndProc;
7. #endif
8. }

由上可见，窗口类中所登记的窗口函数被一众函数强行设置为AfxWndProc。于是，当执行::DispatchMessage函数时，消息就被默默地送往AfxWndProc中去处理了。据说：这样迂回的做法是为了包容新的3D Controls，并与MFC2.5兼容。

好了，热身运动做完了，下面就来看看消息到底是怎么被处理的吧。我们已经知道，消息被抓取后将被送往AfxWndProc，该函数主要操作如下（WINCORE.CPP）：

1. AfxWndProc(HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam)
2. {
3. // special message which identifies the window as using AfxWndProc
4. if (nMsg == WM\_QUERYAFXWNDPROC)
5. return 1;
7. // all other messages route through message map
8. CWnd\* pWnd = CWnd::FromHandlePermanent(hWnd);
9. ASSERT(pWnd != NULL);
10. ASSERT(pWnd->m\_hWnd == hWnd);
11. if (pWnd == NULL || pWnd->m\_hWnd != hWnd)
12. return ::DefWindowProc(hWnd, nMsg, wParam, lParam);
13. return AfxCallWndProc(pWnd, hWnd, nMsg, wParam, lParam);
14. }

其中AfxCallWndProc的主要操作如下（WINCORE.CPP）：

1. LRESULTAFXAPI AfxCallWndProc(CWnd\* pWnd, HWND hWnd, UINT nMsg,
2. WPARAM wParam = 0, LPARAM lParam = 0)
3. {
4. ……
5. // delegate to object's WindowProc
6. lResult = pWnd->WindowProc(nMsg, wParam,lParam);
7. ……
8. returnlResult;
9. }

其中的WindowProc函数如下（WINCORE.CPP）：

LRESULTCWnd::WindowProc(UINT message, WPARAM wParam, LPARAM lParam)

{

    // OnWndMsg does most of the work, exceptfor DefWindowProc call

    LRESULT lResult = 0;

    if (!OnWndMsg(message, wParam, lParam, &lResult))

       lResult = DefWindowProc(message, wParam, lParam);

    return lResult;

}

其中调用的DefWindowProc主要操作如下（WINCORE.CPP）：

1. LRESULT CWnd::DefWindowProc(UINT nMsg, WPARAM wParam, LPARAM lParam)
2. {
3. if (m\_pfnSuper != NULL)
4. return ::CallWindowProc(m\_pfnSuper, m\_hWnd, nMsg, wParam, lParam);
6. WNDPROC pfnWndProc;
7. if ((pfnWndProc = \*GetSuperWndProcAddr()) == NULL)
8. return ::DefWindowProc(m\_hWnd, nMsg, wParam, lParam);
9. else
10. return ::CallWindowProc(pfnWndProc, m\_hWnd, nMsg, wParam, lParam);
11. }

WindowProc中还调用了OnWndMsg函数，该函数是分辨并处理消息的专职机构。如果是命令消息，就交给OnCommand处理，如果是通知消息，就交给OnNotify处理。WM\_ACTIVATE和WM\_SETCURSOR也都有特定的处理函数。而一般的Windows消息，就直接在消息映射表中上溯，寻找其处理程序。之所以要区分出命令消息WM\_COMMAND和通知消息WM\_NOTIFY，是因为它们的处理方式会更加地微妙（这个以后再说了）。

OnWndMsg的主要操作逻辑如下（WINCORE.CPP）：

1. BOOL CWnd::OnWndMsg(UINT message, WPARAM wParam, LPARAM lParam, LRESULT\* pResult)
2. {
3. LRESULT lResult = 0;
4. // special case for commands
5. if (message == WM\_COMMAND)
6. {
7. if (OnCommand(wParam, lParam))
8. {
9. lResult = 1;
10. goto LReturnTrue;
11. }
12. return FALSE;
13. }
14. // special case for notifies
15. if (message == WM\_NOTIFY)
16. {
17. NMHDR\* pNMHDR = (NMHDR\*)lParam;
18. if (pNMHDR->hwndFrom != NULL && OnNotify(wParam, lParam, &lResult))
19. goto LReturnTrue;
20. return FALSE;
21. }
22. ……
23. const AFX\_MSGMAP\* pMessageMap; pMessageMap = GetMessageMap();
24. UINT iHash; iHash = (LOWORD((DWORD)pMessageMap) ^ message) & (iHashMax-1);
25. AfxLockGlobals(CRIT\_WINMSGCACHE);
26. AFX\_MSG\_CACHE\* pMsgCache; pMsgCache = &\_afxMsgCache[iHash];
27. const AFX\_MSGMAP\_ENTRY\* lpEntry;
28. if (message == pMsgCache->nMsg && pMessageMap == pMsgCache->pMessageMap)//检查该消息是否存在cache之中
29. {
30. // cache hit
31. lpEntry = pMsgCache->lpEntry;
32. AfxUnlockGlobals(CRIT\_WINMSGCACHE);
33. if (lpEntry == NULL)
34. return FALSE;
36. // cache hit, and it needs to be handled
37. if (message < 0xC000)
38. goto LDispatch;
39. else
40. goto LDispatchRegistered;
41. }
42. else
43. {
44. // not in cache, look for it
45. pMsgCache->nMsg = message;
46. pMsgCache->pMessageMap = pMessageMap;
48. for (/\* pMessageMap already init'ed \*/; pMessageMap != NULL;
49. pMessageMap = pMessageMap->pBaseMap)
50. {
51. // Note: catch not so common but fatal mistake!!
52. // BEGIN\_MESSAGE\_MAP(CMyWnd, CMyWnd)
53. /\*
54. 利用AfxFindMessageEntry寻找消息映射表中对应的消息处理程序。如果找到，再按照message为一般消息（< 0xC000）或自行注册的消息(> 0xC000）分别跳转到LDispatch:或LDispatchRegistered:去执行
55. \*/
56. if (message < 0xC000)
57. {
58. // constant window message
59. if ((lpEntry = AfxFindMessageEntry(pMessageMap->lpEntries,message, 0, 0)) != NULL)
60. {
61. pMsgCache->lpEntry = lpEntry;
62. AfxUnlockGlobals(CRIT\_WINMSGCACHE);
63. goto LDispatch;
64. }
65. }
66. else
67. {
68. // registered windows message
69. lpEntry = pMessageMap->lpEntries;
70. while ((lpEntry = AfxFindMessageEntry(lpEntry, 0xC000, 0, 0)) != NULL)
71. {
72. UINT\* pnID = (UINT\*)(lpEntry->nSig);
73. ASSERT(\*pnID >= 0xC000 || \*pnID == 0);
74. // must be successfully registered
75. if (\*pnID == message)
76. {
77. pMsgCache->lpEntry = lpEntry;
78. AfxUnlockGlobals(CRIT\_WINMSGCACHE);
79. goto LDispatchRegistered;
80. }
81. lpEntry++; // keep looking past this one
82. }
83. }
84. }
86. pMsgCache->lpEntry = NULL;
87. AfxUnlockGlobals(CRIT\_WINMSGCACHE);
88. return FALSE;
89. }
90. ASSERT(FALSE); // not reached
91. LDispatch:
92. ASSERT(message < 0xC000);
93. union MessageMapFunctions mmf;
94. mmf.pfn = lpEntry->pfn;
95. int nSig;
96. nSig = lpEntry->nSig;
97. ……
98. switch (nSig)
99. {
100. default:
101. ASSERT(FALSE);
102. break;
104. case AfxSig\_bD:
105. lResult = (this->\*mmf.pfn\_bD)(CDC::FromHandle((HDC)wParam));
106. break;
108. case AfxSig\_bb: // AfxSig\_bb, AfxSig\_bw, AfxSig\_bh
109. lResult = (this->\*mmf.pfn\_bb)((BOOL)wParam);
110. break;
112. case AfxSig\_bWww: // really AfxSig\_bWiw
113. lResult = (this->\*mmf.pfn\_bWww)(CWnd::FromHandle((HWND)wParam),
114. (short)LOWORD(lParam), HIWORD(lParam));
115. break;
116. ……
117. }
118. goto LReturnTrue;
119. LDispatchRegistered: // for registered windows messages
120. ASSERT(message >= 0xC000);
121. mmf.pfn = lpEntry->pfn;
122. lResult = (this->\*mmf.pfn\_lwl)(wParam, lParam);
124. LReturnTrue:
125. if (pResult != NULL)
126. \*pResult = lResult;
127. return TRUE;
128. }

其中调用的AfxFindMessageEntry函数主要操作如下（WINCORE.CPP）：

1. const AFX\_MSGMAP\_ENTRY\* AFXAPI
2. AfxFindMessageEntry(const AFX\_MSGMAP\_ENTRY\* lpEntry,
3. UINT nMsg, UINT nCode, UINT nID)
4. {
5. #if defined(\_M\_IX86) && !defined(\_AFX\_PORTABLE)
6. // 32-bit Intel 386/486 version.
7. //此处使用内联汇编做相应的处理，从而加快速度
8. ……
9. #else // \_AFX\_PORTABLE
10. // C version of search routine
11. while (lpEntry->nSig != AfxSig\_end)
12. {
13. if (lpEntry->nMessage == nMsg && lpEntry->nCode == nCode &&
14. nID >= lpEntry->nID && nID <= lpEntry->nLastID)
15. {
16. return lpEntry;
17. }
18. lpEntry++;//AFX\_MSGMAP\_ENTRY指针的++操作
19. }
20. return NULL; // not found
21. #endif // \_AFX\_PORTABLE
22. }

由上面的函数逻辑可知，对于一般的消息，所进行的操作只是比较消息映射表，如果有吻合的项目，就调用表中所记录的函数。比较得对象主要有两个，优先在MFC中的cache中查找，没有命中的话，查找的第二个对象就是建立的消息映射表。比较成功后转到执行函数时，有一个庞大的switch/case结构，这里主要是为了保证类型安全。

假如，有WM\_PAINT消息发生于CMyView窗口中，则消息的流动路线如下图所示：