A big little program: Monitoring Internet Explorer and Explorer windows, part 2: Tracking navigations

devblogs.microsoft.com/oldnewthing/20130613-00

June 13, 2013

Raymond Chen

Okay, it's been a while since we set aside our Little Program to learn a bit about connection points and using dispatch interfaces as connection point interfaces. Now we can put that knowledge to use.

Internet Explorer and Explorer windows fire a group of events known as `DWebBrowser-Events`, so we just need to listen on those events to follow the window as it navigates around.

Take our scratch program and make these changes:

```
#define UNICODE
#define _UNICODE
#define STRICT
#define STRICT_TYPED_ITEMIDS
#include <windows.h>
#include <windowsx.h>
#include <ole2.h>
#include <commctrl.h>
#include <shlwapi.h>


#include <shlobj.h>
#include <atlbase.h>
#include <atlalloc.h>
#include <exdisp.h>
#include <exdispid.h>



…
// DispInterfaceBase incorporated by reference


void UpdateText(HWND hwnd, PCWSTR pszText);


class CWebBrowserEventsSink :
    public CDispInterfaceBase<DWebBrowserEvents>


public:
 CWebBrowserEventsSink(HWND hwnd) : m_hwnd(hwnd) { }


 IFACEMETHODIMP SimpleInvoke(
    DISPID dispid, DISPPARAMS *pdispparams, VARIANT *pvarResult)
 {
  switch (dispid) {
  case DISPID_NAVIGATECOMPLETE:
   UpdateText(m_hwnd, pdispparams->rgvarg[0].bstrVal);
   break;


  case DISPID_QUIT:
   UpdateText(m_hwnd, L”<exited>”);
   Disconnect();
   break;
  }
  return S_OK;
 };
```

```
private:
 HWND m_hwnd;
};
```

Our event sink class listens for `DISPID_NAVIGATECOMPLETE` and `DISPID_QUIT` and updates the text with the new navigation location or the string `L"<exited>"` if the window exited. In the exit case, we also disconnect from the connection point to break the circular reference.

The IDL file for `NavigateComplete` says

```
[id(DISPID_NAVIGATECOMPLETE), helpstring("…")]
void NavigateComplete([in] BSTR URL );
```

Therefore, we know that the URL parameter arrives as a `VT_BSTR` in position zero, so we can access it as `pdispparams->rgvarg[0].bstrVal`.

That class is basically the guts of the program. The rest is scaffolding. Like hooking up this guy to a listview item so it can report its findings somewhere.

```
struct ItemInfo
{
 ItemInfo(HWND hwnd, IDispatch *pdisp)
  : hwnd(hwnd) {
  spSink.Attach(new(std::nothrow) CWebBrowsrEventsSink(hwnd));
  if (spSink) spSink->Connect(pdisp);
 }
 ~ItemInfo() { if (spSink) spSink->Disconnect(); }


 HWND hwnd;
 CComPtr<CWebBrowserEventsSink> spSink;
};


ItemInfo *GetItemByIndex(int iItem)
{
 LVITEM item;
 item.mask = LVIF_PARAM;
 item.iItem = iItem;
 item.iSubItem = 0;
 item.lParam = 0;
 ListView_GetItem(g_hwndChild, &item);
 return reinterpret_cast<ItemInfo *>(item.lParam);
}


ItemInfo *GetItemByWindow(HWND hwnd, int *piItem)
{
 int iItem = ListView_GetItemCount(g_hwndChild);
 while (–iItem >= 0) {
  ItemInfo *pii = GetItemByIndex(iItem);
  if (pii->hwnd == hwnd) {
   if (piItem) *piItem = iItem;
   return pii;
  }
 }
 return nullptr;
}


void UpdateText(HWND hwnd, PCWSTR pszText)
{
 int iItem;
 if (GetItemByWindow(hwnd, &iItem)) {
  ListView_SetItemText(g_hwndChild, iItem, 0,
                       const_cast<PWSTR>(pszText));
 }
}
```

Attached to each listview item is an `ItemInfo` structure which remembers the browser window it is associated with and the event sink that is listening for events.

```
// GetLocationFromView, GetLocationFromBrowser, and GetBrowserInfo
// incorporated by reference


CComPtr<IShellWindows> g_spWindows;


// rename DumpWindows to BuildWindowList
HRESULT BuildWindowList()
{
 CComPtr<IUnknown> spunkEnum;
 HRESULT hr = g_spWindows->_NewEnum(&spunkEnum);
 if (FAILED(hr)) return hr;


 CComQIPtr<IEnumVARIANT> spev(spunkEnum);
 for (CComVariant svar;
      spev->Next(1, &svar, nullptr) == S_OK;
      svar.Clear()) {
  if (svar.vt != VT_DISPATCH) continue;


  HWND hwnd;
  CComHeapPtr<WCHAR> spszLocation;
  if (FAILED(GetBrowserInfo(svar.pdispVal,
             &hwnd, &spszLocation))) continue;


  ItemInfo *pii =
           new(std::nothrow) ItemInfo(hwnd, svar.pdispVal);
  if (!pii) continue;


  LVITEM item;
  item.mask = LVIF_TEXT | LVIF_PARAM;
  item.iItem = MAXLONG;
  item.iSubItem = 0;
  item.pszText = spszLocation;
  item.lParam = reinterpret_cast<LPARAM>(pii);
  int iItem = ListView_InsertItem(g_hwndChild, &item);
  if (iItem < 0) delete pii;
 }
 return S_OK;
}
```

To build the window list, we enumerate the contents of the `IShellWindows`. For each window, we get its window handle and current location and create a listview item for it. The reference data for the listview item is the `ItemInfo`.

```
BOOL
OnCreate(HWND hwnd, LPCREATESTRUCT lpcs)
{
 g_hwndChild = CreateWindow(WC_LISTVIEW, 0,
    LVS_LIST | WS_CHILD | WS_VISIBLE |
    WS_HSCROLL | WS_VSCROLL, 0, 0, 0, 0,
    hwnd, (HMENU)1, g_hinst, 0);
 g_spWindows.CoCreateInstance(CLSID_ShellWindows);
 BuildWindowList();
 return TRUE;
}
```

Our creation function creates a child listview and fills it with stuff.

And of course we clean up our objects when the items are deleted and when the window is destroyed.

```
LRESULT OnNotify(HWND hwnd, int idFrom, NMHDR *pnm)
{
 switch (idFrom) {
 case 1:
  switch (pnm->code) {
  case LVN_DELETEITEM:
   {
    auto pnmlv = CONTAINING_RECORD(pnm, NMLISTVIEW, hdr);
    delete reinterpret_cast<ItemInfo *>(pnmlv->lParam);
   }
   break;
  }
 }
 return 0;
}


void OnDestroy(HWND hwnd)
{
 g_spWindows.Release();
 PostQuitMessage(0);
}


 HANDLE_MSG(hwnd, WM_NOTIFY, OnNotify);
```

And there we have it, a program that displays all the Internet Explorer and Explorer windows and updates their locations as you navigate.

Note, however, that our program doesn't notice when new windows are created. We'll hook that up next time.

Raymond Chen

Follow