How to host an IContextMenu, part 10 – Composite extensions – groundwork

devblogs.microsoft.com/oldnewthing/20041006-00

October 6, 2004

Raymond Chen

You might wonder why the `IContextMenu` interface operates on menu identifier offsets so much rather than with the menu identifiers themselves.

The reason is to support something which I will call "compositing".

You may have multiple context menu extensions that you want to combine into one giant context menu extension. The shell does this all over the place. For example, the context menu we have been playing with all this time is really a composite of several individual context menu extensions: the static registry verbs plus all the COM-based extensions like "Send To", "Open With", and anything else that may have been added by a program you installed (like a virus checker).

So before we can write a compositor, we need to have a second context menu to composite. Here's a quickie that implements two commands, let's call them "Top" and "Next" for lack of anything interesting to do.

```
class CTopContextMenu : public IContextMenu
{
public:
  // *** IUnknown ***
  STDMETHODIMP QueryInterface(REFIID riid, void **ppv);
  STDMETHODIMP_(ULONG) AddRef();
  STDMETHODIMP_(ULONG) Release();


  // *** IContextMenu ***
  STDMETHODIMP QueryContextMenu(HMENU hmenu,
                          UINT indexMenu, UINT idCmdFirst,
                          UINT idCmdLast, UINT uFlags);
  STDMETHODIMP InvokeCommand(
                          LPCMINVOKECOMMANDINFO lpici);
  STDMETHODIMP GetCommandString(
                          UINT_PTR    idCmd,
                          UINT        uType,
                          UINT      * pwReserved,
                          LPSTR       pszName,
                          UINT        cchMax);


  static HRESULT Create(REFIID riid, void **ppv);


private:
  CTopContextMenu() : m_cRef(1), m_cids(0) { }


private:
  HRESULT ValidateCommand(UINT_PTR idCmd, BOOL fUnicode,
                          UINT *puOffset);
  HRESULT Top(LPCMINVOKECOMMANDINFO lpici);
  HRESULT Next(LPCMINVOKECOMMANDINFO lpici);


private:
  ULONG m_cRef;
  UINT  m_cids;
};
```

The class declaration isn't particularly interesting. We are not owner-draw so we don't bother implementing `IContextMenu2` or `IContextMenu3`.

First, some basic paperwork for getting off the ground.

```
HRESULT CTopContextMenu::Create(REFIID riid, void **ppv)
{
  *ppv = NULL;
  HRESULT hr;
  CTopContextMenu *self = new CTopContextMenu();
  if (self) {
    hr = self->QueryInterface(riid, ppv);
    self->Release();
  } else {
    hr = E_OUTOFMEMORY;
  }
  return hr;
}
```

We have two commands. Instead of hard-coding the numbers `0` and `1`, let’s give them nice names.

```
#define TOPCMD_TOP      0
#define TOPCMD_NEXT     1
#define TOPCMD_MAX      2
```

And here’s a table that we’re going to use to help us manage these two commands.

```
const struct COMMANDINFO {
  LPCSTR  pszNameA;
  LPCWSTR pszNameW;
  LPCSTR  pszHelpA;
  LPCWSTR pszHelpW;
} c_rgciTop[] = {
  { “top”,  L”top”,
    “The top command”,  L”The top command”, }, // TOPCMD_TOP
  { “next”, L”next”,
    “The next command”, L”The next command”, },// TOPCMD_NEXT
};
```

Our `TOPCMD_*` values conveniently double as indices into the `c_rgciTop` array.

Next come the boring parts of a COM object:

```
HRESULT CTopContextMenu::QueryInterface(REFIID riid, void **ppv)
{
  IUnknown *punk = NULL;
  if (riid == IID_IUnknown) {
    punk = static_cast<IUnknown*>(this);
  } else if (riid == IID_IContextMenu) {
    punk = static_cast<IContextMenu*>(this);
  }


  *ppv = punk;
  if (punk) {
    punk->AddRef();
    return S_OK;
  } else {
    return E_NOINTERFACE;
  }
}


ULONG CTopContextMenu::AddRef()
{
  return ++m_cRef;
}


ULONG CTopContextMenu::Release()
{
  ULONG cRef = –m_cRef;
  if (cRef == 0) delete this;
  return cRef;
}
```

Finally, we get to something interesting: `IContextMenu::QueryContextMenu`. Things to watch out for in the code below:

- Checking whether there is room between `idCmdFirst` and `idCmdLast` is complicated by the fact that `idCmdLast` is endpoint-**inclusive**, which forces a strange `+1`. Another reason to prefer endpoint-exclusive ranges.
- If the `CMF_DEFAULTONLY` flag is set, then we don't bother adding our menu items since none of our options is the default menu item.

```
HRESULT CTopContextMenu::QueryContextMenu(
    HMENU hmenu, UINT indexMenu, UINT idCmdFirst,
    UINT idCmdLast, UINT uFlags)
{
  m_cids = 0;


  if ((int)(idCmdLast – idCmdFirst + 1) >= TOPCMD_MAX &&
    !(uFlags & CMF_DEFAULTONLY)) {
    InsertMenu(hmenu, indexMenu + TOPCMD_TOP, MF_BYPOSITION,
               idCmdFirst + TOPCMD_TOP, TEXT(“Top”));
    InsertMenu(hmenu, indexMenu + TOPCMD_NEXT, MF_BYPOSITION,
               idCmdFirst + TOPCMD_NEXT, TEXT(“Next”));
    m_cids = TOPCMD_MAX;
  }


  return MAKE_HRESULT(SEVERITY_SUCCESS, 0, m_cids);
}
```

In order to implement the next few methods, we need to have some culture-invariant comparison functions.

```
int strcmpiA_invariant(LPCSTR psz1, LPCSTR psz2)
{
  return CompareStringA(LOCALE_INVARIANT, NORM_IGNORECASE,
                        psz1, -1, psz2, -1) – CSTR_EQUAL;
}


int strcmpiW_invariant(LPCWSTR psz1, LPCWSTR psz2)
{
  return CompareStringW(LOCALE_INVARIANT, NORM_IGNORECASE,
                        psz1, -1, psz2, -1) – CSTR_EQUAL;
}
```

These are like the strcmpi functions except that they use the invariant locale since they will be used to compare canonical strings rather than strings that are meaningful to an end user. (More discussion here in MSDN.)

Now we have enough to write a helper function which is central to the context menu: Figuring out which command somebody is talking about.

Commands can be passed to the `IContextMenu` interface either (a) by ordinal or by name, and either (b) as ANSI or as Unicode. This counts as either three ways or four ways, depending on whether you treat “ANSI as ordinal” and “Unicode as ordinal” as the same thing or not.

```
HRESULT CTopContextMenu::ValidateCommand(UINT_PTR idCmd,
                        BOOL fUnicode, UINT *puOffset)
{
  if (!IS_INTRESOURCE(idCmd)) {
    if (fUnicode) {
      LPCWSTR pszMatch = (LPCWSTR)idCmd;
      for (idCmd = 0; idCmd < TOPCMD_MAX; idCmd++) {
        if (strcmpiW_invariant(pszMatch,
                               c_rgciTop[idCmd].pszNameW) == 0) {
          break;
        }
      }
    } else {
      LPCSTR pszMatch = (LPCSTR)idCmd;
      for (idCmd = 0; idCmd < TOPCMD_MAX; idCmd++) {
        if (strcmpiA_invariant(pszMatch,
                               c_rgciTop[idCmd].pszNameA) == 0) {
          break;
        }
      }
    }
  }


  if (idCmd < m_cids) {
    *puOffset = (UINT)idCmd;
    return S_OK;
  }


  return E_INVALIDARG;
}
```

This helper function takes a “something” parameter in the form of a `UINT_PTR` and a flag that indicates whether that “something” is ANSI or Unicode. The function itself checks whether the “something” is a string or an ordinal. If a string, then it converts that string into an ordinal by looking for it in the table of commands in the appropriate character set and using a locale-insensitive comparison. Notice that if the string is not found, then `idCmd` is left equal to `TOPCMD_MAX`, which is an invalid value (and therefore is neatly handled by the fall-through).

After the (possibly failed) conversion to an ordinal, the ordinal is checked for validity; if valid, then the ordinal is returned back for further processing.

With this helper function the implementation of the other methods of the `IContextMenu` interface are a lot easier.

We continue with the `IContextMenu::InvokeCommand` method:

```
HRESULT CTopContextMenu::InvokeCommand(
                            LPCMINVOKECOMMANDINFO lpici) {


  CMINVOKECOMMANDINFOEX* lpicix = (CMINVOKECOMMANDINFOEX*)lpici;
  BOOL fUnicode = lpici->cbSize >= sizeof(CMINVOKECOMMANDINFOEX) &&
                (lpici->fMask & CMIC_MASK_UNICODE);
  UINT idCmd;
  HRESULT hr = ValidateCommand(fUnicode ? (UINT_PTR)lpicix->lpVerbW
                                        : (UINT_PTR)lpici->lpVerb,
                               fUnicode, &idCmd);
  if (SUCCEEDED(hr)) {
    switch (idCmd) {
    case TOPCMD_TOP: hr = Top(lpici); break;
    case TOPCMD_NEXT: hr = Next(lpici); break;
    default: hr = E_INVALIDARG; break;
    }
  }
  return hr;
}
```

Here is a case where the “Are there three cases or four?” question lands squarely on the side of “four”. There are two forms of the `CMINVOKECOMMANDINFO` structure, the base structure (which is ANSI-only) and the extended structure `CMINVOKECOMMANDINFOEX` which adds Unicode support.

If the structure is `CMINVOKECOMMANDINFOEX` and the `CMIC_MASK_UNICODE` flag is set, then the Unicode fields of the `CMINVOKECOMMANDINFOEX` structure should be used in preference to the ANSI ones.

This means that there are indeed four scenarios:

1. ANSI string in `lpVerb` member.
2. Ordinal in `lpVerb` member.
3. Unicode string in `lpVerbW` member.
4. Ordinal in `lpVerbW` member.

After figuring out whether the parameter is ANSI or Unicode, we ask `ValidateCommand` to do the work of validating the verb and converting it to an ordinal, at which point we use the ordinal in a `switch` statement to dispatch the actual operation.

Failing to implement string-based command invocation is an extremely common oversight in context menu implementations. Doing so prevents people from invoking your verbs programmatically.

“Why should I bother to let people invoke my verbs programmatically?”

Because if you don’t, then people won’t be able to write programs like the one we are developing in this series of articles! For example, suppose your context menu extension lets people “Frob” a file. If you don’t expose this verb programmability, then it is impossible to write a program that, say, takes all the files modified in the last twenty-four hours and Frobs them.

(I’m always amused by the people who complain that Explorer doesn’t expose enough customizability programmatically, while simultaneously not providing the same degree of programmatic customizability in their own programs.)

Oh wait, I guess I should implement those two operations. They don’t do anything particularly interesting.

```
HRESULT CTopContextMenu::Top(LPCMINVOKECOMMANDINFO lpici)
{
  MessageBox(lpici->hwnd, TEXT(“Top”), TEXT(“Title”), MB_OK);
  return S_OK;
}


HRESULT CTopContextMenu::Next(LPCMINVOKECOMMANDINFO lpici)
{
  MessageBox(lpici->hwnd, TEXT(“Next”), TEXT(“Title”), MB_OK);
  return S_OK;
}
```

The remaining method is `IContextMenu::GetCommandString`, which is probably the one people most frequently get wrong since the consequences of getting it wrong are not immediately visible to the implementor. It is the people who are trying to access the context menu programmatically who most likely to notice that the method isn’t working properly.

```
HRESULT CTopContextMenu::GetCommandString(
                            UINT_PTR    idCmd,
                            UINT        uType,
                            UINT      * pwReserved,
                            LPSTR       pszName,
                            UINT        cchMax)
{
  UINT id;
  HRESULT hr = ValidateCommand(idCmd, uType & GCS_UNICODE, &id);
  if (FAILED(hr)) {
    if (uType == GCS_VALIDATEA || uType == GCS_VALIDATEW) {
      hr = S_FALSE;
    }
    return hr;
  }


  switch (uType) {
  case GCS_VERBA:
    lstrcpynA(pszName, c_rgciTop[id].pszNameA, cchMax);
    return S_OK;


  case GCS_VERBW:
    lstrcpynW((LPWSTR)pszName, c_rgciTop[id].pszNameW, cchMax);
    return S_OK;


  case GCS_HELPTEXTA:
    lstrcpynA(pszName, c_rgciTop[id].pszHelpA, cchMax);
    return S_OK;


  case GCS_HELPTEXTW:
    lstrcpynW((LPWSTR)pszName, c_rgciTop[id].pszHelpW, cchMax);
    return S_OK;


  case GCS_VALIDATEA:
  case GCS_VALIDATEW:
    return S_OK;    // all they wanted was validation
  }


  return E_NOTIMPL;
}
```

Here again we use the `ValidateCommand` method to do the hard work of validating the command, which is passed in the `idCmd` parameter, with interpretive assistance in the `GCS_UNICODE` flag of the `uType` parameter.

If the command is not valid, then we propagate the error code, except in the `GCS_VALIDATE` cases, where the documentation says that we should return `S_FALSE` to indicate that the command is not valid.

If the command is valid, we return the requested information, which is handled by a simple `switch` statement.

Okay, now that we have this context menu, we can even test it out a little bit. Throw out the changes from part 9 and return to the program as it was in part 6, making the following change to the `OnContextMenu` function:

```
void OnContextMenu(HWND hwnd, HWND hwndContext, int xPos, int yPos)
{
  POINT pt = { xPos, yPos };
  if (pt.x == -1 && pt.y == -1) {
    pt.x = pt.y = 0;
    ClientToScreen(hwnd, &pt);
  }


  IContextMenu *pcm;
  if (SUCCEEDED(CTopContextMenu::Create(
                  IID_IContextMenu, (void**)&pcm))) {
    …
```

We now obtain our context menu not by calling the `GetUIObjectOfFile` function but rather by constructing a `CTopContextMenu` object. Since our `CTopContextMenu` implements `IContextMenu`, all the remaining code can be left unchanged.

When you run this program, observe that even the help text works.

Ah, one of the powers of operating with interfaces rather than objects: You can swap out the object and the rest of the code doesn't even realize what happened, so long as the interface stays the same.

Okay, today was a long day spent just laying groundwork, just writing what has to be written. No breakthroughs, no "aha" moments, just typing. Read the method, understand what you have to do, and do it.

Next time, we're going to see context menu composition, using this context menu as one of the components.

Raymond Chen

Follow