Windows Shell 扩展指南

面向 Windows 10/11 重写的 Shell 扩展开发指南——从 COM/DLL 地基到右键菜单、缩略图、属性、预览与 MSIX 打包，贴合最新官方文档。

全 13 篇 · 作者 David

概述：二十年后的 Shell 扩展

二十多年前，Michael Dunn 在 CodeProject 写了一套 *The Complete Idiot's Guide to Writing Shell Extensions*，我把它翻译成中文放在个人主页上，是很多人入门 Windows Shell 编程的第一站。但那套教程停在 2004 年——它讲的 `IColumnProvider`、`IExtractImage`、`IPersistFile` 初始化、乃至整套注册方式，在 Windows 10/11 上要么被取代、要么被移除、要么已经不是推荐做法。

这份指南不是翻译，而是重写：保留原系列由浅入深的例子脉络，但每一个都按 Win10/Win11 的现状与最新官方文档重新构思。示例代码单独放在开源仓库 davidapp/shell-extension-demos 里维护（每个示例一个目录、自带 CMake 工程，clone 即可构建），正文只保留讲清原理所需的关键片段。

> 重点放在 Windows 10 与 Windows 11。凡涉及“老写法仍能用、但有更好的新做法”的地方，都会明确标注，并给出官方文档链接。

什么是 Shell 扩展

Windows 资源管理器（Explorer）本质上是一个可扩展的外壳。Shell 扩展就是你写的一个进程内 COM 组件（通常是一个 DLL），资源管理器在需要时把它加载进来，让你在特定环节“插一脚”：

- 用户右键一个文件 → 让你往快捷菜单里加命令；
- 资源管理器要画某个文件的缩略图 / 图标 → 让你提供自定义图像；
- 用户打开属性对话框、悬停看信息提示、在详细信息视图里排序分组 → 让你提供数据；
- 用户拖放、预览文件 → 让你接管。

关键区别：注册表也能做一些定制（比如给某类文件配一个固定图标），但注册表只能“一刀切”——同一类型的所有文件长一个样。Shell 扩展则能逐文件决定行为（这个文本文件显示这个图标、那个显示那个）。这正是官方文档里区分“注册表定制”与“扩展处理器（handler）”的分界线。

二十年后：到底变了什么

如果你读过老版指南，先看这张对照表——它决定了这份新指南每一章要怎么写：

老指南里的例子	当年的接口	**Windows 10/11 现状**
文本文件右键菜单	`IContextMenu`	仍可用；但 Win11 新版右键菜单需要 `IExplorerCommand` + MSIX 打包，未打包的处理器只落到「显示更多选项」旧菜单里
多文件右键 + `regsvr32` 注册	`IContextMenu`	逻辑不变；注册未打包走注册表、打包走包清单
弹出信息提示（用 MFC）	`IQueryInfo`	仍可用；更推荐走属性系统（`prop:` 字符串或 `IPropertyStore`）
拖放建硬链接	`IDropTarget`	仍可用
文件属性页	`IShellPropSheetExt`	仍可用
「复制到 / 发送到」右键	`IContextMenu`（右键拖放）	仍可用；配套的 DDE 已弃用，改用 `IDropTarget` 激活
文件夹背景菜单 + BMP 缩略图	`IExtractImage`	缩略图改用 `IThumbnailProvider`（Vista 起推荐）
详细信息列（列处理器，读 MP3 标签）	`IColumnProvider`	Vista 起已移除；改用属性处理器 `IPropertyStore` + 属性 schema
按文件大小显示不同图标	`IExtractIcon`	仍可用

除此之外，还有两处“地基级”的变化贯穿所有处理器：

- 初始化接口：老代码用 `IPersistFile` / `IShellExtInit` 拿到文件名。Vista 起这些被 `IInitializeWithItem`、`IInitializeWithStream`、`IInitializeWithFile` 取代（图标处理器现在也不再强制 `IPersistFile`）。
- 属性系统统一化：详细信息列、信息提示、属性页里的字段、搜索与分组，如今都可以由一个属性处理器（`IPropertyStore`）统一供数据，不再各写各的。老指南里“列处理器”这一章，在新版里会整体并入“属性处理器”。

> ⚠ `IColumnProvider`（列处理器）在 Windows Vista 就被移除了，别再照老教程写。要往详细信息视图加列，正确做法是实现属性处理器并注册属性 schema——本指南有专门一章。

你需要的工具

- Windows 11（或 Windows 10）+ 一台能随便重启资源管理器、装卸扩展的开发机（强烈建议用虚拟机，扩展写崩了会拖垮 Explorer）。
- Visual Studio 2022，含「使用 C++ 的桌面开发」与「通用 Windows 平台开发 / MSIX 打包工具」工作负载。
- C++17、Windows SDK（较新版本）。
- WIL（Windows Implementation Library）：微软官方的现代 C++ 头文件库，`wil::com_ptr`、RAII 句柄、错误处理宏能让 COM 代码干净很多。老教程里的 ATL 仍可用，但新代码更推荐 WIL（可与 ATL 并存）。
- 调试利器：`Process Monitor`、`DebugView`，以及资源管理器的独立进程选项（方便挂调试器、崩了不连累桌面）。

> 微软把大量现成、可编译的扩展范例放在 Windows-classic-samples 仓库（搜 `ThumbnailProvider`、`PreviewHandler`、`ExplorerCommandVerb`、`PropertyHandler` 等目录）。本指南的示例仓库会以它们为骨架，去掉过时写法、补齐 Win11 打包。

先打好地基：COM 与 DLL

绝大多数 Shell 扩展是进程内（in-process）COM 对象，打包成 DLL。要成为一个合格的扩展，它至少要：

- 有一个唯一的 GUID（CLSID）；
- 实现 `IUnknown` 与一个类工厂（class factory）；
- 从 DLL 导出三个标准函数：`DllGetClassObject`（暴露类工厂）、`DllCanUnloadNow`（告诉系统没人用了可以卸载）、`DllMain`；
- 实现对应处理器的接口（`IContextMenu`、`IThumbnailProvider`……）以及现代初始化接口（`IInitializeWithItem` 等）。

线程模型统一用 Apartment。这些“样板”在每个扩展里几乎一样，示例仓库会把它抽成可复用的基类，正文则聚焦每种处理器独有的接口。

注册：注册表 与 MSIX 两条路

同一个处理器，在 Win10/11 上有两种“让系统认识它”的方式，本指南两条都会讲：

1. 注册表（未打包）：经典做法。在 `HKEY_CLASSES_ROOT` 下写 `CLSID\{GUID}\InProcServer32` 指向 DLL，再到文件类型的 `ShellEx\<处理器子键>` 下挂上 GUID。改动后要调用 `SHChangeNotify(SHCNE_ASSOCCHANGED, …)` 通知系统。适合快速开发、内部工具。

2. MSIX / 稀疏包（packaged）：现代做法，也是进入 Windows 11 新版右键菜单的唯一途径。用包清单（`appxmanifest`）里的 `com:Extension` + `desktop4/desktop5:Extension` 声明扩展，系统按包来加载、更新、卸载，干净且可上架。缩略图、预览、右键命令等都能这样注册。

> ⚠ 进程内扩展是被加载进资源管理器（或其它宿主进程）里跑的——一旦你的代码崩了或死锁，轻则该功能失灵，重则整个桌面卡死。所以每个扩展都要：严格判空与错误处理、绝不在 UI 线程做耗时/阻塞操作、务必在虚拟机里测。可能的话优先用进程外代理宿主（如预览处理器天然跑在 `prevhost.exe` 里）。

本指南结构

按“一次一个例子”推进，每一章聚焦一种处理器，给出可运行的最小示例 + Win10/11 注意点 + 官方文档链接：

1. 右键菜单扩展（`IContextMenu`） —— 第一个 in-proc DLL 扩展，把 COM/DLL 地基走通
2. 现代命令模型 `IExplorerCommand` —— 更简单的命令写法、级联子菜单、动态启用/隐藏
3. Windows 11 新版右键菜单 —— 用 MSIX / 稀疏包把命令送进主菜单
4. 缩略图处理器 `IThumbnailProvider` —— 取代 `IExtractImage`
5. 图标处理器 `IExtractIcon` 与图标叠加 `IShellIconOverlayIdentifier`
6. 属性处理器 `IPropertyStore` 与属性 schema —— 详细信息列 / 分组 / 信息提示的统一数据源（取代列处理器）
7. 信息提示 `IQueryInfo`
8. 属性表页 `IShellPropSheetExt`
9. 预览处理器 `IPreviewHandler`
10. 拖放处理器 `IDropTarget`
11. 打包、部署、调试与旧代码迁移

> *“章节顺序可能随写作微调；每章配套的完整、可编译示例见开源仓库 davidapp/shell-extension-demos。第 ① 章的 `01-context-menu` 已可直接构建运行。”*

参考

- Windows Shell 文档入口
- 创建 Shell 扩展处理器（各处理器总览、注册机制）
- 创建快捷（右键）菜单处理器
- Windows-classic-samples 示例集
- WIL（Windows Implementation Library）

第 ① 章 · 右键菜单扩展（IContextMenu）

第一个例子的目标不是炫技，而是把地基走通：一个进程内 COM DLL，右键任意文件时弹出「显示文件信息」，点一下用消息框显示该文件的路径和大小。后面每一章都建立在这套骨架上。

配套代码：`01-context-menu`（CMake 工程，clone 即可构建）。

它由哪些部分组成

一个能被资源管理器加载的右键菜单扩展，至少要凑齐三样东西：

1. 一个 COM 对象：有唯一 CLSID，实现 `IUnknown` 和一个类工厂；
2. 两个 Shell 接口：
   - `IShellExtInit` —— 系统借它把「你右键了哪些文件」交给你；
   - `IContextMenu` —— 你借它往菜单里加项、并在用户点击时干活；
3. 注册表登记：让系统知道「右键某类文件时该加载哪个 DLL」。

> 进程内扩展被加载进资源管理器进程里运行。所以第一条铁律：在虚拟机里开发调试。你的代码一崩，倒下的是整个桌面。

COM/DLL 的四个出口

一个 COM 进程内服务器 DLL 必须导出四个标准函数（见 `dllmain.cpp` 与 `ContextMenuExt.def`）：

导出函数	作用
`DllGetClassObject`	COM 向你要「类工厂」，好用它造对象
`DllCanUnloadNow`	没有存活对象时返回 `S_OK`，允许系统卸载 DLL
`DllRegisterServer` / `DllUnregisterServer`	自注册/反注册（`regsvr32` 调它们写/删注册表）

配合一个 DLL 级引用计数 `g_cDllRef`：每造一个对象就 `+1`，析构就 `-1`；只要它非零，`DllCanUnloadNow` 就返回 `S_FALSE`，系统便不会把正在被用的 DLL 卸载掉。

dllmain.cpp（节选）

```
STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid, void** ppv) {
    if (rclsid != CLSID_FileInfoContextMenu)
        return CLASS_E_CLASSNOTAVAILABLE;
    ClassFactory* pFactory = new (std::nothrow) ClassFactory();
    if (!pFactory) return E_OUTOFMEMORY;
    HRESULT hr = pFactory->QueryInterface(riid, ppv);
    pFactory->Release();
    return hr;
}

STDAPI DllCanUnloadNow() {
    return (g_cDllRef == 0) ? S_OK : S_FALSE;
}
```

类工厂本身很程式化：`CreateInstance` 里 `new` 出真正的处理器对象，`QueryInterface` 到请求的接口再把多余的引用释放掉。这段样板在每个扩展里几乎一样——第 ② 章我们就把它抽进一个公共头文件，本章先把它摊开让你看清全貌。

拿到被右键的文件：IShellExtInit

系统在弹出菜单之前调用 `IShellExtInit::Initialize`，把选中的对象通过一个 `IDataObject` 传进来。我们从中取出第一个文件的路径：

ContextMenuHandler.cpp（节选）

```
IFACEMETHODIMP ContextMenuHandler::Initialize(
    PCIDLIST_ABSOLUTE, IDataObject* pdtobj, HKEY)
{
    FORMATETC fe = { CF_HDROP, nullptr, DVASPECT_CONTENT, -1, TYMED_HGLOBAL };
    STGMEDIUM stg;
    if (FAILED(pdtobj->GetData(&fe, &stg))) return E_INVALIDARG;

    HDROP hDrop = static_cast<HDROP>(GlobalLock(stg.hGlobal));
    if (hDrop) {
        if (DragQueryFileW(hDrop, 0, m_szFile, ARRAYSIZE(m_szFile)) > 0)
            m_hasFile = true;
        GlobalUnlock(stg.hGlobal);
    }
    ReleaseStgMedium(&stg);
    return m_hasFile ? S_OK : E_INVALIDARG;
}
```

这里用经典的 `CF_HDROP` + `DragQueryFile` 取路径，简单直观。

> 更现代的做法是 `SHCreateShellItemArrayFromDataObject` 把 `IDataObject` 转成 `IShellItemArray`，能拿到完整的命名空间信息（不只是文件路径）。第 ② 章的 `IExplorerCommand` 天生就收 `IShellItemArray`，我们到那时再用。本例先用 `DragQueryFile` 把概念讲清楚。

> ⚠ 示例为了简洁只取了第一个文件。真实扩展要用 `DragQueryFile(hDrop, 0xFFFFFFFF, …)` 拿到总数并遍历全部选中项。

加菜单、执行、给名字：IContextMenu

三个方法各司其职：

`QueryContextMenu` —— 往菜单里插项。返回值的低 16 位是你用掉的命令 ID 个数（这是最容易踩的坑：返回错了菜单会错乱）：

```
IFACEMETHODIMP ContextMenuHandler::QueryContextMenu(
    HMENU hMenu, UINT indexMenu, UINT idCmdFirst, UINT, UINT uFlags)
{
    if (uFlags & CMF_DEFAULTONLY)               // 只要默认项时别加自定义命令
        return MAKE_HRESULT(SEVERITY_SUCCESS, FACILITY_NULL, 0);

    InsertMenuW(hMenu, indexMenu, MF_BYPOSITION | MF_STRING,
                idCmdFirst + IDM_SHOWINFO, L"显示文件信息(&I)");

    return MAKE_HRESULT(SEVERITY_SUCCESS, FACILITY_NULL, IDM_SHOWINFO + 1);
}
```

菜单项的命令 ID 必须用 `idCmdFirst + 偏移`——因为同一个菜单上可能挂着好几个扩展，系统给每个扩展分配一段 ID 区间，你只能用自己的偏移，不能写死。

`InvokeCommand` —— 用户点了菜单项时执行。命令可能以「偏移量」或「字符串 verb」两种形式传入，两种都要认（用 `IS_INTRESOURCE` 区分）：

```
IFACEMETHODIMP ContextMenuHandler::InvokeCommand(CMINVOKECOMMANDINFO* pici)
    if (IS_INTRESOURCE(pici->lpVerb)) {
        if (LOWORD(pici->lpVerb) != IDM_SHOWINFO) return E_INVALIDARG;
    } else {
        if (StrCmpIA(pici->lpVerb, "showinfo") != 0) return E_INVALIDARG;
    }
    // …GetFileAttributesEx 取大小，MessageBox 显示路径+大小…
    return S_OK;
}
```

`GetCommandString` —— 给命令提供 verb 名（供脚本/`ShellExecute` 调用）和状态栏帮助文本，按 `uType`（`GCS_VERBW` / `GCS_HELPTEXTW` …）分别返回。

注册表：把 DLL 挂上去

`DllRegisterServer` 写三处（全在 `HKEY_CLASSES_ROOT` 下）：

```
CLSID\{GUID}\                          （默认） = 友好名
CLSID\{GUID}\InProcServer32\           （默认） = DLL 完整路径
                                       ThreadingModel = Apartment
*\shellex\ContextMenuHandlers\FileInfoDemo\  （默认） = {GUID}
```

第三行是关键：把处理器挂在 `*`（所有文件）的 `ContextMenuHandlers` 下。真实项目应挂在具体文件类型的 ProgID（如 `txtfile`）下，而不是 `*`——这里用 `*` 只是方便你右键任何文件都能立刻看到效果。

改完注册表，务必调用 `SHChangeNotify(SHCNE_ASSOCCHANGED, …)` 通知系统，不用重启即可生效。

构建、安装、验证

```
cd 01-context-menu
cmake -B build -A x64
cmake --build build --config Release
```

> ⚠ 两个必踩的坑：① 必须 x64——现代资源管理器是 64 位进程，加载不了 32 位 DLL。② 源码是 UTF-8——中文 locale 下 MSVC 默认按 GB2312 读源码会报「语法错误」，CMake 里加 `/utf-8`（示例已加）。

在管理员 PowerShell 里注册（写 `HKCR` 需要管理员）：

```
regsvr32 .\build\Release\ContextMenuExt.dll     # 注册
regsvr32 /u .\build\Release\ContextMenuExt.dll  # 卸载
```

然后右键任意文件——就能看到「显示文件信息」。

Windows 11：为什么在「显示更多选项」里

你会发现：在 Win11 上，这个命令不在默认弹出的那个新版右键菜单里，而要点「显示更多选项」（或按 Shift + F10）才在下面的旧菜单里。

这不是 bug。Win11 的新版菜单只收经过 MSIX / 稀疏包声明、并用 `IExplorerCommand` 实现的命令；所有用注册表挂载的经典 `IContextMenu` 处理器，一律被归到「显示更多选项」的兼容菜单里。

- 想让命令用更现代、更省事的接口写 → 第 ② 章 `IExplorerCommand`；
- 想让命令进入 Win11 主菜单 → 第 ③ 章 MSIX / 稀疏包打包。

参考

- 创建快捷（右键）菜单处理器
- IContextMenu · IShellExtInit
- 注册 Shell 扩展处理器

第 ② 章 · 现代命令模型（IExplorerCommand）

第 ① 章的 `IContextMenu` 是 1996 年的接口：一个方法里既画菜单又管命令 ID，还要自己解析 verb 字符串。Windows 7 引入的 `IExplorerCommand` 是更现代的模型——一个 `IExplorerCommand` 对象就是“一条命令”，标题、图标、状态、执行各是一个方法。更重要的是：它是把命令送进 Windows 11 主右键菜单的实现基础（第 ③ 章打包时会直接复用本章的类）。

配套代码：`02-explorer-command`。

先抽出公共地基

从本章起，第 ① 章手写的那套 COM 样板（类工厂、DLL 引用计数、注册表工具）抽进了仓库根的 `common/ShellExtBase.h`。其中最有用的是一个类工厂模板：

common/ShellExtBase.h（节选）

```
template <class THandler>
class ClassFactory : public IClassFactory {
    IFACEMETHODIMP CreateInstance(IUnknown* outer, REFIID riid, void** ppv) ove
        if (outer) return CLASS_E_NOAGGREGATION;
        THandler* p = new (std::nothrow) THandler();
        if (!p) return E_OUTOFMEMORY;
        HRESULT hr = p->QueryInterface(riid, ppv);
        p->Release();
        return hr;
    }
    // …AddRef/Release/LockServer/QueryInterface…
};
```

于是每个新示例的 `dllmain.cpp` 里，`DllGetClassObject` 只要 `new ClassFactory<你的处理器>()` 即可，不用再抄一遍工厂。`g_cDllRef` 用了 C++17 的 inline 变量，跨编译单元只有一份实例。

IExplorerCommand 的八个方法

方法	作用
`GetTitle`	菜单显示的文字
`GetIcon`	图标（`"dll,-资源ID"` 字符串；不要就返回 `E_NOTIMPL`）
`GetToolTip`	提示文字
`GetState`	该命令此刻是启用 / 禁用 / 隐藏（`EXPCMDSTATE`）
`GetFlags`	命令特性（有无子菜单、是否分隔符……`EXPCMDFLAGS`）
`EnumSubCommands`	级联子命令的枚举器
`GetCanonicalName`	命令的 GUID 标识（没有就返回 `GUID_NULL`）
`Invoke`	用户点击时执行

和 `IContextMenu` 最大的不同：选中的文件通过参数 `IShellItemArray` 传进来，不再需要 `IShellExtInit`。

ExplorerCommand.cpp（节选）

```
IFACEMETHODIMP FileInfoCommand::GetTitle(IShellItemArray*, LPWSTR* ppszName) {
    return SHStrDupW(L"显示文件信息 (IExplorerCommand)", ppszName);
}
IFACEMETHODIMP FileInfoCommand::GetState(IShellItemArray*, BOOL, EXPCMDSTATE* pS
    *pState = ECS_ENABLED;      // 也可返回 ECS_DISABLED / ECS_HIDDEN
    return S_OK;
}
IFACEMETHODIMP FileInfoCommand::Invoke(IShellItemArray* items, IBindCtx*) {
    IShellItem* psi = nullptr;
    items->GetItemAt(0, &psi);
    PWSTR path = nullptr;
    psi->GetDisplayName(SIGDN_FILESYSPATH, &path);    // 拿文件系统路径
    // …取大小、MessageBox…
    CoTaskMemFree(path);
    psi->Release();
    return S_OK;
}
```

⚠ `GetTitle` / `GetIcon` / `GetToolTip` 返回的字符串必须用 `CoTaskMemAlloc` 分配（`SHStrDupW` 就是），由 Shell 负责释放；用 `malloc` / `new` 会内存出错。同理 `GetDisplayName` 拿到的 `path` 要用 `CoTaskMemFree` 释放。

ℹ `IExplorerCommand` 的所有方法都在 UI 线程调用，且只支持进程内激活。绝不能在里面做网络请求或其它阻塞操作——会直接卡住资源管理器界面。

级联子菜单

想做“一个父项展开出多条子命令”，只需两步：`GetFlags` 返回 `ECF_HASSUBCOMMANDS`，并在 `EnumSubCommands` 里返回一个 `IEnumExplorerCommand` 枚举器（每个元素又是一个 `IExplorerCommand`）。本例没用到，返回 `E_NOTIMPL`。这也是 Win7+ 做级联菜单的推荐方式（比注册表的 `SubCommands` 更灵活）。

注册：ExplorerCommandHandler

这是和第 ① 章的另一处关键区别。`IExplorerCommand` 作为 verb 注册在 `shell\<verb>` 下，用一个名为 `ExplorerCommandHandler` 的值指向 CLSID（而不是 `ShellEx\ContextMenuHandlers`）：

```
CLSID\{GUID}\InProcServer32\  (默认) = DLL 路径, ThreadingModel = Apartment
*\shell\FileInfoExplorerCmd\  (默认) = 显示文件信息 (IExplorerCommand)
                              ExplorerCommandHandler = {GUID}
```

dllmain.cpp（节选）

```
RegWriteString(HKEY_CLASSES_ROOT, kVerbKey, nullptr, L"显示文件信息 (IExplorerComm
RegWriteString(HKEY_CLASSES_ROOT, kVerbKey, L"ExplorerCommandHandler", clsidStr
```

> 记事本、Notepad++ 等都用这套 `ExplorerCommandHandler` 注册法接入右键菜单。它比经典 `ContextMenuHandlers` 更省事——不用自己实现 `QueryContextMenu` / `InvokeCommand`。

构建与验证

```
cmake -B build -A x64
cmake --build build --config Release
```

```
.\install.ps1              # 右键任意文件（Win11 在"显示更多选项"里）
.\install.ps1 -Uninstall
```

到这里，命令仍然落在 Win11 的「显示更多选项」旧菜单里——因为它还是用注册表注册的。下一章我们给它套上 MSIX / 稀疏包，把同一个 `IExplorerCommand` 类送进 Win11 的主右键菜单。

参考

- IExplorerCommand · EXPCMDSTATE · EXPCMDFLAGS
- ExplorerCommandVerb 官方示例

- 在 Windows 11 扩展右键菜单与共享对话框

第 ③ 章 · 进入 Windows 11 主右键菜单（MSIX / 稀疏包）

前两章的命令，在 Windows 11 上都只能在「显示更多选项」的旧菜单里找到。因为 Win11 的新版主菜单有个硬门槛：命令必须来自一个有“包标识”（package identity）的应用，并用 `IExplorerCommand` + 包清单声明。

好消息是：不用重写处理器。本章把第 ② 章那个 `IExplorerCommand` 原样拿来，套一层稀疏包（sparse package），就能进主菜单。

配套代码：`03-win11-packaging`。

什么是稀疏包

传统 MSIX 会把你的所有文件打进一个包里安装。稀疏包只打一个清单（`AppxManifest.xml`），文件仍留在你原来的安装目录——注册时用 `-ExternalLocation` 指向那个目录即可。这样非打包的 Win32 应用能“低成本”获得包标识，解锁 Win11 主菜单、Share、通知等能力。见官方文档。

三个清单，各司其职

打包扩展有点绕的地方在于：有三处身份信息必须完全对上。

① `package/AppxManifest.xml` —— 稀疏包清单。两段扩展是关键：

```
<!-- 声明进程内 COM 服务器：Clsid=第②章的命令，Path=外部目录里的 DLL -->
<com:Extension Category="windows.comServer">
  <com:ComServer>
    <com:SurrogateServer DisplayName="FileInfo ExplorerCommand">
      <com:Class Id="A3F1C2D4-5E6B-4789-9012-3456789ABCDE"
                 Path="w11menu.dll" ThreadingModel="STA" />
    </com:SurrogateServer>
  </com:ComServer>
</com:Extension>

<!-- 把该 CLSID 注册成文件资源管理器右键命令（这就是进主菜单的开关） -->
<desktop4:Extension Category="windows.fileExplorerContextMenus">
  <desktop4:FileExplorerContextMenus>
    <desktop5:ItemType Type="*">
      <desktop5:Verb Id="FileInfoCmd" Clsid="A3F1C2D4-5E6B-4789-9012-3456789ABCI
    </desktop5:ItemType>
  </desktop4:FileExplorerContextMenus>
</desktop4:Extension>
```

还需 `<uap10:AllowExternalContent>true</uap10:AllowExternalContent>`、`runFullTrust` + `unvirtualizedResources` 能力、以及一个 `TrustLevel=mediumIL`、`RuntimeBehavior=win32App` 的 `Application`（DLL 项目没有真 exe，用个幽灵 exe 占位，右键命令不会去启动它）。

② `app.manifest` —— 嵌入 DLL 的并排清单，用 `msix` 元素把这个二进制绑到包标识：

```
<assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-com:asm.v1">
  <assemblyIdentity version="1.0.0.0" name="DavidApp.ShellExtDemos.Win11Menu" ty
  <msix xmlns="urn:schemas-microsoft-com:msix.v1"
        publisher="CN=DavidApp Shell Ext Demos"
        packageName="DavidApp.ShellExtDemos.Win11Menu"
        applicationId="App" />
</assembly>
```

CMake 用 `/MANIFEST:EMBED /MANIFESTINPUT` 把它嵌进 `w11menu.dll`。

> ⚠ 这是最容易漏、也最坑的一步：DLL 里不嵌这个 `msix` 标识清单，资源管理器就不会把它当作已打包组件加载——命令死活不出现，且没有任何报错。

③ 证书 —— 签名证书的 `Subject` 必须等于清单里的 `Publisher`。

> *“一句话记忆：证书 Subject = AppxManifest 的 `Identity/Publisher` = app.manifest 的 `msix/@publisher`，三处一字不差。”*

复用第②章的处理器

`dllmain.cpp` 只剩最精简的 `DllGetClassObject` / `DllCanUnloadNow`，处理器直接编译第②章的 `ExplorerCommand.cpp`：

CMakeLists.txt（节选）

```
add_library(w11menu SHARED
    dllmain.cpp
    ${CMAKE_CURRENT_SOURCE_DIR}/../02-explorer-command/ExplorerCommand.cpp  # 复
    w11menu.def)
set_target_properties(w11menu PROPERTIES
    LINK_FLAGS "/MANIFEST:EMBED /MANIFESTINPUT:\"${CMAKE_CURRENT_SOURCE_DIR}/app
```

打包后不再需要 `DllRegisterServer`——注册完全由稀疏包完成。

一键构建 + 注册

示例的 `build-package.ps1` 把六步串起来：

```
.\build-package.ps1      # 管理员 + Win11 开发者模式
```

1. `cmake` 构建带嵌入清单的 DLL；
2. 组装“外部位置”目录（DLL + 占位资源）；
3. `makeappx pack /nv` 打稀疏包；
4. `New-SelfSignedCertificate` 自签；
5. `signtool` 签名，并把公钥证书导入 受信任人（否则报 `0x800B0109 CERT_E_UNTRUSTEDROOT`）；
6. `Add-AppxPackage -ExternalLocation` 注册。

成功后右键任意文件，「显示文件信息」就直接出现在 Win11 主菜单里了。

> 自签名 + 侧载仅用于开发测试。要分发给别人，需用受信任 CA 的证书（或 Azure Trusted Signing）；上架微软商店则用完整 MSIX。

参考

- 用外部位置为非打包应用授予包标识
- windows.fileExplorerContextMenus 扩展 · MakeAppx
- PackageWithExternalLocation 官方示例

第 ④ 章 · 缩略图处理器 (IThumbnailProvider)

从本章起进入“文件视觉”部分。缩略图就是资源管理器在大图标视图里给文件画的那张预览图。老教程用的 `IExtractImage` 早已被 `IThumbnailProvider`（Vista 起推荐）取代——后者更简单、更安全（默认在隔离进程里跑）。

本章给 `.txt` 文件生成缩略图：底色由文件大小推导、中间画出字节数。配套代码：`04-thumbnail`。

两个接口

- `IInitializeWithStream` —— Shell 不再给你文件路径，而是给一个 `IStream`。这样即使文件在网络上、或在受限沙箱里也能取到内容，且缩略图默认跑在隔离宿主 `dllhost.exe` 里，崩了不连累资源管理器。
- `IThumbnailProvider::GetThumbnail(cx, phbmp, pdwAlpha)` —— 给定边长 `cx`，返回一张 `HBITMAP`。

ThumbnailProvider.cpp（节选）

```
IFACEMETHODIMP ThumbnailProvider::Initialize(IStream* pStream, DWORD) {
    if (m_pStream) return E_UNEXPECTED;          // Shell 保证只调一次
    return pStream->QueryInterface(IID_PPV_ARGS(&m_pStream));
}
```

画一张 32bpp 位图

Shell 缩略图要求 32 位位图。用 `CreateDIBSection` 建一张自顶向下（高度取负）的 DIB，然后用普通 GDI 往里画：

```
BITMAPINFO bmi = {};
bmi.bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
bmi.bmiHeader.biWidth  =  (LONG)cx;
bmi.bmiHeader.biHeight = -(LONG)cx;   // 负 = 自顶向下
bmi.bmiHeader.biPlanes = 1;
bmi.bmiHeader.biBitCount = 32;
bmi.bmiHeader.biCompression = BI_RGB;
void* bits = nullptr;
HBITMAP hbmp = CreateDIBSection(nullptr, &bmi, DIB_RGB_COLORS, &bits, nullptr, 
// 选进内存 DC，FillRect 底色 + DrawText 画字节数…
*phbmp = hbmp;
*pdwAlpha = WTSAT_RGB;   // 不用 alpha 通道就声明 RGB
```

`pdwAlpha`（`WTS_ALPHATYPE`）告诉 Shell 位图有没有 alpha：`WTSAT_RGB` 忽略 alpha 通道；`WTSAT_ARGB` 表示预乘 alpha（做圆角/透明缩略图时用，且像素必须是预乘的）。用普通 GDI 画完不去管 alpha，就老实声明 `WTSAT_RGB`。

`GetThumbnail` 里别读整个大文件、别做网络请求。虽然它在独立宿主进程里跑、不会直接卡住资源管理器，但仍要尽快返回——否则缩略图迟迟不出来。

注册

缩略图处理器注册在文件类型的 `shellex\{E357FCCD-A995-4576-B01F-234630154E96}` 下（这个固定 GUID 是系统定义的“缩略图处理器”类别），默认值为你的 CLSID：

```
RegisterInprocServer(CLSID_DemoThumbnailProvider, dll, L"Demo Thumbnail Provide
RegWriteString(HKEY_CLASSES_ROOT,
    L".txt\\shellex\\{E357FCCD-A995-4576-B01F-234630154E96}", nullptr, clsidStr
```

缩略图有缓存。装好后看不到效果，把 `.txt` 文件夹切到「大图标」视图；仍不行就用「磁盘清理」清缩略图缓存、或删 `%LocalAppData%\Microsoft\Windows\Explorer\thumbcache_*.db` 后重启资源管理器。

参考

- IThumbnailProvider · IInitializeWithStream
- RecipeThumbnailProvider 官方示例

第 ⑤ 章 · 图标处理器与图标叠加

注册表能给一整类文件配一个固定图标，但做不到逐文件不同。要“这个 `.txt` 一个图标、那个 `.txt` 另一个图标”，就得写图标处理器。本章还捎带讲图标叠加（就是同步盘、Git 客户端在文件角上加的那个小箭头/对勾）。

配套代码：`05-icon`。

IExtractIcon：逐文件图标

两个接口配合：`IPersistFile::Load` 拿到文件路径，`IExtractIconW` 决定图标。

IconHandler.cpp（节选）

```
IFACEMETHODIMP IconHandler::GetIconLocation(UINT, PWSTR pszIconFile, UINT cchMax
                                            int* piIndex, UINT* pwFlags) {
    ULONGLONG size = /* 用 GetFileAttributesEx 取 m_szFile 大小 */;
    wchar_t sysdir[MAX_PATH]; GetSystemDirectoryW(sysdir, ARRAYSIZE(sysdir));
    StringCchPrintfW(pszIconFile, cchMax, L"%s\\shell32.dll", sysdir);
    *piIndex = (size < 1024) ? 71 : (size < 1024*1024) ? 70 : 47;  // 按大小选图标
    *pwFlags = GIL_PERINSTANCE;   // ★ 关键
    return S_OK;
}
IFACEMETHODIMP IconHandler::Extract(PCWSTR, UINT, HICON* l, HICON* s, UINT) {
    if (l) *l = nullptr; if (s) *s = nullptr;
    return S_FALSE;   // 让 Shell 按上面给的“文件+索引”自己加载
}
```

> ⚠ `GIL_PERINSTANCE` 不能漏。不置它，Shell 会认为“同类型文件图标都一样”，把图标按类型缓存成一个——逐文件就白做了。

`GetIconLocation` 返回一个“文件+索引”，`Extract` 返回 `S_FALSE` 让 Shell 自行加载即可；只有当图标需要动态生成（不是现成的资源）时，才在 `Extract` 里亲自返回 `HICON`。

注册在 ProgID 的 `shellex\IconHandler` 下：

```
RegWriteString(HKEY_CLASSES_ROOT, L"txtfile\\shellex\\IconHandler", nullptr, cls
```

> ℹ `IExtractIcon` 现在也不再强制 `IPersistFile`——现代可改用 `IInitializeWithItem` 拿 `IShellItem`。本例用 `IPersistFile` 因为它对 IconHandler 最通用、最省事。

图标叠加：IShellIconOverlayIdentifier

叠加图标不针对某类文件，而是全局注册、由系统对每个图标询问“你要不要在这个文件上叠个小图标？”。接口只有三个方法，且不需要初始化接口（路径直接给到 `IsMemberOf`）：

```
class OverlayHandler : public IShellIconOverlayIdentifier {
    // 这个文件要不要叠加？（本例：只读文件才叠）
    IFACEMETHODIMP IsMemberOf(PCWSTR pwszPath, DWORD dwAttrib) {
        return (dwAttrib & FILE_ATTRIBUTE_READONLY) ? S_OK : S_FALSE;
    }
    // 叠加用哪个图标
    IFACEMETHODIMP GetOverlayInfo(PWSTR pwszIconFile, int cchMax, int* pIndex, I
        wchar_t sysdir[MAX_PATH]; GetSystemDirectoryW(sysdir, ARRAYSIZE(sysdir)
        StringCchPrintfW(pwszIconFile, cchMax, L"%s\\shell32.dll", sysdir);
        *pIndex = 47;
        *pdwFlags = ISIOI_ICONFILE | ISIOI_ICONINDEX;
        return S_OK;
    }
    IFACEMETHODIMP GetPriority(int* pPriority) { *pPriority = 50; return S_OK; }
};
```

注册在 HKLM（不是 HKCR）的 overlay 列表下：

```
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\ShellIconOverlayIdentif
    <名字>   （默认）= {CLSID}
```

> ⚠ 著名的“15 个槽位”限制：整个系统的图标叠加总共只有 15 个名额，超出的直接被忽略。而且这些项按名字的字母序排队——这就是为什么 OneDrive、Dropbox、TortoiseGit 的叠加名都带一堆前导空格（`OneDrive`）来抢排在前面。改动 overlay 注册后必须重启资源管理器才生效。

> *“叠加处理器的代码骨架（类工厂、DLL 导出、注册）和本章的图标处理器完全一样，只是实现的接口和注册位置不同——照着 `05-icon` 改即可。”*

参考

- IExtractIcon · 创建图标处理器
- IShellIconOverlayIdentifier · 实现图标叠加处理器

第 ⑥ 章 · 属性处理器（IPropertyStore）

老教程里有一章叫“列处理器”（`IColumnProvider`），用来往资源管理器的详细信息视图加自定义列。这个接口在 Windows Vista 就被彻底移除了。它的现代替代——而且能力大得多——是属性处理器 `IPropertyStore`。

关键变化：过去详细信息列、信息提示、属性页各写一套；现在一个属性处理器统一供数据，同时喂给：详细信息列、信息提示（infotip）、分组、排序、搜索索引、属性页。写一处，处处生效。

本例给自定义扩展 `.shdemo` 暴露一个属性。配套代码：`06-property`。

属性 = PROPERTYKEY + 值

属性系统里每个属性由一个 `PROPERTYKEY`（一个 `{FMTID}` + 一个整数 PID）唯一标识，值是 `PROPVARIANT`。系统预定义了几百个规范属性（`System.Author`、`System.Comment`、`System.Size`…），头文件 `propkey.h` 里是 `PKEY_Author`、`PKEY_Comment` 这些常量。

`IPropertyStore` 就五个方法：

PropertyStore.cpp（节选）

```
IFACEMETHODIMP PropertyStore::GetCount(DWORD* c) { *c = 1; return S_OK; }
IFACEMETHODIMP PropertyStore::GetAt(DWORD i, PROPERTYKEY* k) {
    if (i != 0) return E_INVALIDARG;
    *k = PKEY_Comment; return S_OK;
}
IFACEMETHODIMP PropertyStore::GetValue(REFPROPERTYKEY key, PROPVARIANT* pv) {
    PropVariantInit(pv);
    if (IsEqualPropertyKey(key, PKEY_Comment)) {
        wchar_t buf[96];
        StringCchPrintfW(buf, ARRAYSIZE(buf), L"演示属性：该文件 %llu 字节", m_size
        return InitPropVariantFromString(buf, pv);
    }
    return S_OK;    // 未知属性返回 VT_EMPTY
}
IFACEMETHODIMP PropertyStore::SetValue(REFPROPERTYKEY, REFPROPVARIANT) { return
IFACEMETHODIMP PropertyStore::Commit() { return STG_E_ACCESSDENIED; }
```

内容还是通过 `IInitializeWithStream` 以流的形式拿到（和缩略图、属性同一套现代初始化方式）。

> 用系统预定义属性（如 `PKEY_Comment`）拿来即用。要暴露自定义属性（比如“项目编号”），得写一份 `.propdesc` XML schema，用 `PSRegisterPropertySchema` 注册，属性才有规范名、类型和本地化显示名，才能作为列/分组项出现。

注册

属性处理器注册在 HKLM 的属性系统里，按扩展名绑定：

```
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\PropertySystem\PropertyHandlers\
    (默认) = {CLSID}
```

```
RegWriteString(HKEY_LOCAL_MACHINE,
    L"SOFTWARE\\...\\PropertySystem\\PropertyHandlers\\.shdemo", nullptr, clsid
```

> ⚠ 属性处理器跑在索引器 / 属性系统宿主里、且会被系统缓存。别在里面做重活；写属性（可写处理器）时要正确实现 `SetValue` / `Commit` 的事务语义。改扩展名绑定后可能要重启资源管理器或重建索引才生效。

迁移提示：从列处理器到属性处理器

如果你在维护老代码里的 `IColumnProvider`：它在 Vista 起就不再被调用了。把“每列一个值”改写成“每个 `PROPERTYKEY` 一个值”的属性处理器，再在文件类型上配置要显示哪些列（`System.PropList.*` 或属性 schema），即可恢复并增强原来的详细信息列功能。

参考

- IPropertyStore · 构建属性处理器
- 属性 schema（.propdesc） · PropertyHandlers 官方示例

第 ⑦ 章 · 信息提示处理器（IQueryInfo）

信息提示（infotip）就是鼠标停在文件上时浮出来的那段说明。定制它有两条路：写一个 `IQueryInfo` 处理器，或者干脆不写代码、用一行注册表配置。

配套代码：`07-infotip`。

写处理器：IQueryInfo

接口很小，核心就一个方法 `GetInfoTip`：

QueryInfoHandler.cpp（节选）

```
IFACEMETHODIMP QueryInfoHandler::GetInfoTip(DWORD, PWSTR* ppwszTip) {
    ULONGLONG size = /* GetFileAttributesEx 取 m_szFile 大小 */;
    wchar_t tip[MAX_PATH + 128];
    StringCchPrintfW(tip, ARRAYSIZE(tip),
        L"Shell 扩展示例\n路径：%s\n大小：%llu 字节", m_szFile, size);
    return SHStrDupW(tip, ppwszTip);   // CoTaskMemAlloc 分配，Shell 负责释放
}
```

文件路径照例通过 `IPersistFile::Load` 拿到。注册在 ProgID 的 `shellex\{00021500-0000-0000-C000-000000000046}`（信息提示处理器的固定类别 GUID）下。

> ⚠ 返回的字符串务必用 `CoTaskMemAlloc`（`SHStrDupW` 即是）；用 `new` / `malloc` 会让 Shell 释放时崩溃。提示里可以用 `\n` 换行。

不写代码：InfoTip 属性字符串

很多时候根本不用写 DLL。在文件类型的 ProgID 下加一个 `InfoTip` 值，用 `prop:` 前缀列出要显示的属性，Shell 就会用属性系统自动拼出提示：

```
HKEY_CLASSES_ROOT\txtfile
    InfoTip = prop:System.ItemType;System.Size;System.DateModified
```

这套 `prop:` 语法直接复用第 ⑥ 章的属性系统——属性处理器暴露的字段，这里都能列。能用属性字符串解决的，就别写 `IQueryInfo`。只有当提示内容需要动态计算（不是现成属性）时，才值得写处理器。

参考

- IQueryInfo
- 信息提示定制（InfoTip 字符串）

第 ⑧ 章 · 属性表处理器（IShellPropSheetExt）

右键文件 →「属性」弹出的那个多标签对话框，就是属性表（property sheet）。本章往里加一个自己的标签页。配套代码：`08-propsheet`。

一个初始化 + 一个页

- `IShellExtInit::Initialize` —— 老规矩，从 `IDataObject` 取出被查看属性的文件（见第 ① 章）。
- `IShellPropSheetExt::AddPages` —— Shell 在组装属性对话框时调它，你用 `CreatePropertySheetPage` 造一个页、交给回调 `lpfnAddPage`。

页本身是一个子对话框：模板放在 `.rc` 资源里，行为写在 `DlgProc` 里。

PropSheetHandler.cpp（节选）

```
IFACEMETHODIMP PropSheetHandler::AddPages(LPFNADDPROPSHEETPAGE lpfnAddPage, LPA
    PROPSHEETPAGE psp = {};
    psp.dwSize      = sizeof(psp);
    psp.dwFlags     = PSP_USETITLE | PSP_USECALLBACK;
    psp.hInstance   = g_hInst;
    psp.pszTemplate= MAKEINTRESOURCEW(IDD_PROPPAGE);
    psp.pszTitle    = L"Shell 示例";                          // 标签页标题
    psp.pfnDlgProc = PropPageDlgProc;
    psp.lParam      = (LPARAM)_wcsdup(m_szFile);              // 传给页的数据（路径副本
    psp.pfnCallback= PropPageCallback;
    HPROPSHEETPAGE hPage = CreatePropertySheetPageW(&psp);
    if (!hPage) { free((void*)psp.lParam); return E_OUTOFMEMORY; }
    if (!lpfnAddPage(hPage, lParam)) { DestroyPropertySheetPage(hPage); return
    return S_OK;
}
```

`DlgProc` 在 `WM_INITDIALOG` 里从 `PROPSHEETPAGE.lParam` 取到路径，算出大小，`SetDlgItemText` 填进静态控件。

生命周期：PSPCB 回调

属性对话框是异步的——`AddPages` 返回后对话框才真正显示、用户看完才关闭。所以要用 `PSP_USECALLBACK` + 回调管好生命周期：

```
static UINT CALLBACK PropPageCallback(HWND, UINT uMsg, LPPROPSHEETPAGE ppsp)
{
    if (uMsg == PSPCB_ADDREF)  DllAddRef();                    // 页在，DLL 别
    else if (uMsg == PSPCB_RELEASE) { free((void*)ppsp->lParam); DllRelease(); }
    return 1;
}
```

> ⚠ 两个易漏点：① DLL 保活——页还开着时 DLL 不能被卸载，靠 `PSPCB_ADDREF` / `RELEASE` 里 `DllAddRef` / `DllRelease`。② 释放传给页的数据——`lParam` 里 `_wcsdup` 的路径副本要在 `PSPCB_RELEASE` 里 `free`，否则内存泄漏。

> ℹ `.rc` 资源里只放 ASCII，避免 `rc.exe` 在中文 locale 下的编码坑；标题、正文等中文在运行时用 `pszTitle` / `SetDlgItemText` 填。这和第 ① 章 `/utf-8` 是同一类问题的两种应对。

注册

属性表处理器可挂多个，注册在 ProgID 的 `shellex\PropertySheetHandlers\<任取的名字>` 下：

```
RegWriteString(HKEY_CLASSES_ROOT,
    L"txtfile\\shellex\\PropertySheetHandlers\\DemoPropSheet", nullptr, clsidSt
```

参考

- IShellPropSheetExt · 属性表处理器
- CreatePropertySheetPage

第 ⑨ 章 · 预览处理器（IPreviewHandler）

预览处理器负责资源管理器右侧「预览窗格」（`Alt+P`）里的内容。它是本指南里最“重”的一种：跑在独立宿主进程 `prevhost.exe` 里，要自己管一个窗口。配套代码：`09-preview`。

三个接口协作

- `IInitializeWithStream` —— 拿文件流。
- `IPreviewHandler` —— 核心：`SetWindow` 告诉你“画在哪个父窗口、哪块矩形”，`DoPreview` 里你在那建子窗口渲染，`SetRect` 跟随窗格大小变化，`Unload` 收尾。
- `IObjectWithSite` —— 宿主设置的站点（预览处理器约定要实现）。

PreviewHandler.cpp（节选）

```
IFACEMETHODIMP PreviewHandler::DoPreview() {
    m_hwndPreview = CreateWindowExW(0, L"EDIT", nullptr,
        WS_CHILD | WS_VISIBLE | WS_VSCROLL | ES_MULTILINE | ES_READONLY | ES_AU
        m_rc.left, m_rc.top, m_rc.right - m_rc.left, m_rc.bottom - m_rc.top,
        m_hwndParent, nullptr, g_hInst, nullptr);
    // 从 m_pStream 读最多 64KB，MultiByteToWideChar(CP_UTF8,…) 后 SetWindowText
    return S_OK;
}
IFACEMETHODIMP PreviewHandler::Unload() {
    if (m_hwndPreview) { DestroyWindow(m_hwndPreview); m_hwndPreview = nullptr;
    if (m_pStream) { m_pStream->Release(); m_pStream = nullptr; }
    return S_OK;
}
```

> ⚠ 几个生命周期铁律：`SetWindow` 可能在 `DoPreview` 前后多次调用（窗格移动/缩放），要能随时把子窗口 `MoveWindow` 到新矩形；`Unload` 必须销毁窗口、释放流，因为同一个预览宿主会被复用来预览下一个文件。

隔离宿主：AppID

预览处理器不直接跑在资源管理器里，而是被托管进 `prevhost.exe`——这样即使你的渲染代码崩了，倒的也只是预览宿主，资源管理器安然无恙。实现方式是给 CLSID 挂一个 AppID，指向系统预置的预览宿主：

```
// 64 位预览宿主 prevhost.exe 的 AppID（DllSurrogate 系统已注册好）
RegWriteString(HKEY_CLASSES_ROOT, L"CLSID\\{我们的CLSID}", L"AppID",
               L"{534A1E02-D58F-44f0-B58B-36CBED287C7C}");
```

三处注册

```
CLSID\{CLSID}\  AppID = {534A1E02-...}            ← 用哪个隔离宿主
HKLM\...\PreviewHandlers\  (值名={CLSID}) = 显示名   ← 登记进系统清单
.myp\shellex\{8895b1c6-b41f-4c1c-a562-0d564250836f}\  (默认)={CLSID}  ← 绑定文件
```

> Windows 内置了大量文档/图片的预览处理器。写自己的通常是为了私有格式。调试时可以直接给 `prevhost.exe` 挂调试器（它是独立进程），比调资源管理器省心得多。

参考

- IPreviewHandler · 预览处理器
- RecipePreviewHandler 官方示例

第 ⑩ 章 · 拖放处理器（IDropTarget）

拖放处理器让某类文件（或文件夹）成为拖放目标：当用户把一批文件拖到它的图标上松手时，你的代码被调用，拿到“被拖入了哪些文件”。资源管理器里那些“拖到某程序图标上用它打开”的行为，底层就是这个机制。配套代码：`10-drop`。

两个接口

- `IPersistFile` —— `Load` 告诉你“用户拖到了哪个目标文件”（图标本身对应的文件路径）。
- `IDropTarget` —— 标准 OLE 拖放目标接口，四个方法：

方法	时机	你要做的
`DragEnter`	拖动进入图标	检查数据格式，通过 `*pdwEffect` 表明能否接收（复制/移动/链接/拒绝）
`DragOver`	在图标上移动	持续更新 `*pdwEffect`（光标样式随之变化）
`DragLeave`	离开图标	清理 `DragEnter` 里的临时状态
`Drop`	松手	从 `IDataObject` 取出文件真正干活

DropHandler.cpp（节选）——松手时取文件列表

```
IFACEMETHODIMP DropHandler::Drop(IDataObject* pDataObj, DWORD, POINTL, DWORD* pd
    if (pdwEffect) *pdwEffect = DROPEFFECT_COPY;
    FORMATETC fe = { CF_HDROP, nullptr, DVASPECT_CONTENT, -1, TYMED_HGLOBAL };
    STGMEDIUM stg;
    if (SUCCEEDED(pDataObj->GetData(&fe, &stg))) {
        HDROP hDrop = (HDROP)GlobalLock(stg.hGlobal);
        UINT n = DragQueryFileW(hDrop, 0xFFFFFFFF, nullptr, 0);    // 先问总数
        for (UINT i = 0; i < n; ++i) {
            wchar_t f[MAX_PATH];
            DragQueryFileW(hDrop, i, f, ARRAYSIZE(f));             // 再逐个取
            /* …处理 f… */
        }
        GlobalUnlock(stg.hGlobal);
        ReleaseStgMedium(&stg);    // 一定要还回去
    }
    return S_OK;
}
```

⚠ 两个易错点：`GetData` 拿到的 `STGMEDIUM` 用完必须 `ReleaseStgMedium`，否则漏内存；`DragQueryFileW` 第二参传 `0xFFFFFFFF` 是“问总数”，传下标才是“取第几个”。

注册

拖放处理器是单个（不是列表），注册在 ProgID 的 `shellex\DropHandler` 下，默认值就是 CLSID：

```
HKCR\txtfile\shellex\DropHandler\  (默认) = {你的CLSID}
```

> 拖放处理器与右键菜单处理器常常配合：很多“把文件拖到某图标上”的操作，和“右键某类文件 → 某命令”其实复用同一套处理逻辑。数据都来自 `IDataObject`，只是入口不同。

参考

- IDropTarget · Drop handlers
- 传输 Shell 对象（拖放与剪贴板）

第 ⑪ 章 · 图标叠加处理器（IShellIconOverlayIdentifier）

你一定见过 OneDrive/Dropbox 文件图标左下角那个“同步中/已同步”的小角标，或者 TortoiseGit 里绿勾红叹号——它们全是图标叠加处理器。这是最能出效果、也最容易把资源管理器拖卡的一种扩展。配套代码：`11-icon-overlay`。

只有一个接口，三个方法

`IShellIconOverlayIdentifier`——不需要任何初始化接口，因为判定方法直接把路径给你：

方法	作用
`GetOverlayInfo`	叠加图标从哪个文件、第几个图标取（返回 `.ico`/DLL 路径 + 索引）
`GetPriority`	多个叠加同时命中一个文件时的排序，`0` 最高
`IsMemberOf(path, attrib)`	核心判定：这个文件要不要叠？`S_OK` 叠，`S_FALSE` 不叠

OverlayHandler.cpp（节选）——判定与取图标

```
// 演示规则：文件名含 “demo” 就叠标记
IFACEMETHODIMP OverlayHandler::IsMemberOf(LPCWSTR pwszPath, DWORD) {
    return (StrStrIW(pwszPath, L"demo") != nullptr) ? S_OK : S_FALSE;
}
// 叠加图标：演示用 shell32.dll 占位，真实项目换成自带 .ico
IFACEMETHODIMP OverlayHandler::GetOverlayInfo(LPWSTR file, int cch, int* pIndex
    GetSystemDirectoryW(file, cch);
    StringCchCatW(file, cch, L"\\shell32.dll");
    *pIndex = 0;
    *pdwFlags = ISIOI_ICONFILE | ISIOI_ICONINDEX;
    return S_OK;
}
```

两条硬约束（决定成败）

⚠ ① 只有前 15 个生效。 系统对叠加处理器有全局硬上限：注册在 `ShellIconOverlayIdentifiers` 下的项按名字排序，只取前 15 个，多的直接忽略。这就是为什么 TortoiseGit、Dropbox 的注册名前面加空格甚至多个空格——空格排在字母前，用来抢靠前的槽位。本示例也用了 `DaiwDemoOverlay`（前置空格）这一招。

② `IsMemberOf` 必须快。它会对资源管理器里每一个显示的文件被调用。任何阻塞操作（读文件内容、查数据库、访问网络）都会让整个窗口卡死。真实扩展的做法是：后台线程维护一份内存缓存，`IsMemberOf` 只查缓存、立刻返回。

注册与生效

叠加处理器登记在 HKLM（机器级，需管理员）：

```
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\ShellIconOverlayIdentif
    〈 名字（前置空格抢槽位）〉\  （默认） = {你的CLSID}
```

改完注册必须重启资源管理器（`Stop-Process -Name explorer -Force`）才会重新扫描叠加列表——这点和右键菜单不同，光刷新是不够的。

参考

- IShellIconOverlayIdentifier · How to Implement Icon Overlay Handlers
- 云存储的现代替代方案：云文件 API（Cloud Filter），它有自己的一套状态角标机制。

第 ⑫ 章 · 调试、部署与现代化迁移

前面每章一个扩展点。这一章把散落的工程问题收一下口：怎么调试、怎么干净地注册、怎么部署，以及——本指南从头到尾的主线——把 20 年前的老写法迁移到 Win10/11 的新接口。

一、调试：你的 DLL 跑在别人的进程里

Shell 扩展是进程内 COM 组件，被宿主进程加载。调试的第一步是搞清楚“宿主是谁”：

扩展类型	宿主进程	怎么调
右键菜单 / 图标 / InfoTip / 属性页	`explorer.exe`	附加到 explorer；改完 DLL 前先 `taskkill /f /im explorer.exe` 再重启
缩略图 / 属性处理器	`dllhost.exe` (COM Surrogate)	附加到对应的 dllhost 实例
预览处理器	`prevhost.exe`	独立进程，直接附加最省心

> ⚠ 最常见的坑：explorer 把你的 DLL 锁住了，`cmake --build` 覆盖不了旧文件。解决办法是每次重建前重启 explorer，或者开发期给 CLSID 挂 `AppID` + `DllSurrogate`，让扩展跑进独立的 `dllhost` 里——崩了也不拖垮桌面(这正是缩略图、预览默认隔离的原因)。

调试期开隔离，给 CLSID 加一个空的 `AppID` 值即可让系统用 `dllhost` 托管：

```
HKCR\CLSID\{你的CLSID}\   AppID = {你的CLSID}
HKCR\AppID\{你的CLSID}\   DllSurrogate = ""    (空字符串 = 用默认代理 dllhost)
```

二、注册：per-user 还是 per-machine

`regsvr32` 默认写 `HKEY_CLASSES_ROOT`，需要管理员。现代推荐 per-user 注册——写 `HKCU\Software\Classes`，免管理员、卸载干净、不影响其他用户：

```
HKCU\Software\Classes\CLSID\{CLSID}\...          ← 组件本体
HKCU\Software\Classes\.ext\shellex\...           ← 关联
```

本指南的示例为求简洁写的是 `HKCR` (= 管理员 per-machine)。真实产品优先 per-user，除非扩展确实要对所有用户生效。

> 在非默认(non-per-user)场景，企业环境可能启用“仅允许已批准的扩展”策略（`HKLM\...\Shell Extensions\Approved`）。个人机通常不受影响，但企业分发时要留意。

三、部署：从 regsvr32 到 MSIX

`regsvr32` + 写注册表是经典做法，但 Win10/11 的现代分发是 MSIX / 稀疏包(sparse package)：

- 声明式：扩展点写在 `AppxManifest.xml` 里(见 Win11 菜单章)，系统按清单注册，不用手写注册表。
- 干净卸载：卸载包 = 所有注册项一起消失，不留垃圾。
- Win11 主菜单硬性要求：前面讲过，要进 Win11 那个新版右键主菜单，必须打包 + 用 `IExplorerCommand`，`regsvr32` 的老菜单只会被折叠进“显示更多选项”。

所以现代扩展的典型形态是：`IExplorerCommand` 实现 + MSIX/稀疏包声明。老的 `IContextMenu` + `regsvr32` 仍然能用，但只活在“更多选项”的二级菜单里。

四、现代化迁移对照表

这是把老代码搬到 Win10/11 时最该记住的一张表——左边是本指南参考的 2004 年老文里的写法，右边是现在该用的：

目的	旧接口(已过时/受限)	新接口(Win10/11 推荐)
初始化——拿到目标文件	`IShellExtInit` / `IPersistFile`	`IInitializeWithStream` → `WithItem` → `WithFile`
缩略图	`IExtractImage`	`IThumbnailProvider`
列表额外栏目	`IColumnProvider` (已移除)	属性处理器 + `IPropertyStore`
右键命令	`IContextMenu` (进“更多选项”)	`IExplorerCommand` + 打包
数据关联	直接读文件句柄	优先 `Stream`，以便沙箱/云文件/隔离宿主

> 一条总原则：能用 `Stream` 就别用 `File`。`IInitializeWithStream` 让扩展跑在受限沙箱、云占位文件(OneDrive Files On-Demand)、隔离宿主里都不出问题；直接抓文件路径的老写法在这些场景会失效。这也是微软反复强调的方向。

结语

到这里，Windows Shell 扩展的主要扩展点就走完了一遍：从右键菜单、Win11 新菜单，到缩略图、图标，再到属性、提示、属性页、预览、拖放、图标叠加。每一章的完整可编译代码都在 shell-

extension-demos 仓库里。

核心心法就三条：扩展跑在别人进程里，所以要快、要稳、要隔离；能用 Stream 初始化就别碰文件路径；面向 Win11 就上 `IExplorerCommand` + MSIX。

参考

- Shell 扩展性总览 · 初始化 Shell 扩展
- 用 MSIX 集成到 Shell
- 注册 Shell 扩展处理器

道雾轩

《Windows Shell 扩展指南》

本电子书由道雾轩制作,免费分享,欢迎转发给需要的人。

阅读全部专栏,请访问官网 daiw.org。