

第一章 Windows Shell 是什么

一个操作系统外壳的不错的定义是它是一个系统提供的用户界面，它允许用户执行公共的任务，如访问文件系统，导出执行程序，改变系统设置等。MS-DOS 有一个 Command.COM 扮演着这个角色。然而 Windows 已经有了图形界面环境，他的外壳程序也就必然是图形方式的。在 Windows95以前，默认的 Windows Shell 就是程序管理器。

程序管理器是一个中央控制台，从那里你可以启动应用程序，重排和重组图标，执行新任务。换句话说，程序管理器就像他的名字提示的那样管理所有集中在 Windows 大伞之下的程序。现在对应程序管理器的是文件管理器，它是一个专门为维护文件系统而设计的工具。

随着 Windows95的出现，探测器取代了这两个老工具，并集成了二者的功能，如果你愿意，你仍能发现文件管理器仍然深深地隐藏在 Windows 系统目录中。然而，由于用户友善性方面比他的后继者差，现今已经很少使用了。

一般错误的概念认为，探测器就是一个程序，当你需要通过点击“我的计算机”或右击“开始”按钮来浏览文件系统时这个程序启动。事实上，探测器总是启动和运行着的，从引导开始一直到你关闭计算机。直觉是“探测器”实际上就是新概念下的窗口。探测器是一个可执行模块 (explorer.exe)，它实现了 Windows 外壳功能。

在这一章中，主要是介绍外壳和探测器，更精确地讲是

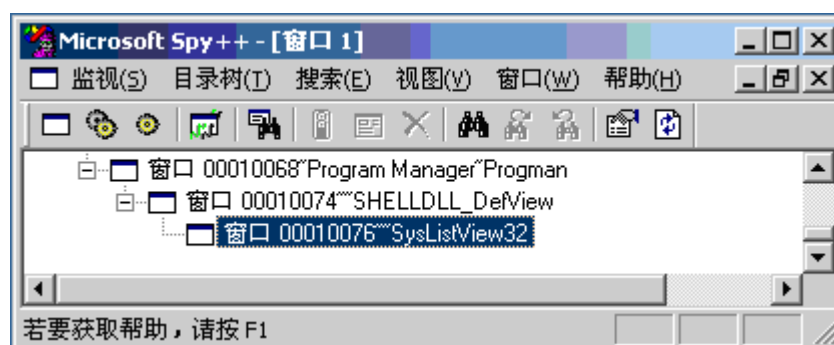
Shell 的组成部分

探测器结构

Shell 的组成部分

Shell 由许多不同的部分组成，现在我们就从最显而易见的**桌面**和**任务条**开始。从概念上讲，桌面是所有 Windows Shell 对象的父对象，即宿主对象。就实现方式而言，桌面是一个系统定义窗口类的特殊窗口（命名为#32769），并且是所有窗口的祖先窗口。那些导出应用的顶层窗口一般而言都是桌面的子窗口。在这些子窗口中有一个有趣的窗口子树，它的根是“程序管理器”。

你可以用 CV++带的工具 Spy++来检查窗口栈中打开的窗口。



程序管理器保持了兼容性，在图中你可以看到，他的封装结构：程序管理器的直接下级是一个名为 SHELLDLL DefView 的窗口类，这个窗口包含了一个默认的观察对象，事实上这个窗口负责枚举标准文件夹的内容，它总是从 **ListView** 控件中导出，这是一个通用的系统控件。SHELLDLL DefView 窗口包含了一个 ListView(类名为 SysListView32)和一个标题控件(类名为 SysHeader32)，标题控件仅仅用于 ListView 的报告观察。

随着 IE4.0的活动桌面和 Windows98的发布，默认的观察对象已经有了基于某些浏览器能力的变化。在下一章中我们将更进一步讨论这些观察对象和他们的变化。

程序管理器

就象前面提到的一样，程序管理器窗口是为了兼容性而保留的。它正好演示了一个窗口应用从16位到32位的演变过程。在 Win16环境中，与 Shell 唯一的通讯方式是通过**动态数据交换（DDE）**。这个层面的代码在 Windows95甚至 Windows98中还在维护。为什么呢？，为了兼容性。

关于 DDE 接口变成与 Shell 的详细说明,建议察看资料 Internet ClientSDK 其中给出了最新的信息。DDE 是一款较老的技术，微软已经有大量的资料说明。

任务条

主要的 Windows Shell 部件就是任务条，它实际上就是由探测器进程所拥有的一个窗口。每当你需要终止探测器进程的时候，都将引起任务条的消失和重新显现。每当他重新显现的时候他注册一个具有不同 HWND 的新窗口。因而，就引用而言，你没有必要保留这个窗口的 HWND。任务条也是也各拥有“开始”按钮的窗口，在托盘区域有时钟和类似按钮的控件，表示正在运行的应用。

任务条实际上与窗口一样，可以在其上作任何窗口上可以做的操作如移动、隐藏和子类化等。在第七章中我们将说明怎样子类化任务条和“开始”按钮。在第九章中可以找到怎样隐藏任务条和编成重新启动 Shell。这后一个特性在编成实现 Shell 和命名空间扩展时是有用的。

桌面

你是否奇怪，桌面上的快捷方式是从哪里来的和属于谁，坦白地讲，开始我也认为探测器模块绘制了这些图标，记录了用户设置，颜色，状态等。这个模块开可能在桌面背景上绘制用户设置的墙纸。

桌面并不是这样工作的，相反，快捷方式作为图标是由一个普通的 ListView 显示的。当然这个 ListView 有了一个不寻常的变异，但是它确实是一个 ListView，因此它也就不难从消息中获取处理对象了，将在第九章中给出例程进行说明。

探测器结构

探测器是一个扮演着系统外壳角色的应用程序。当说到外壳扩展的时候，我们讲的就是有探测器感知的、装入的并最终执行的代码块。

探测器可以被看作为一个微型的窗口开发环境。想象一下：它有自己的函数和对话框；允许写特殊的与已存在的体系集成的应用程序；能包容应用和文档。他甚至可以解释任何活动的脚本兼容语言的脚本（VBScript，JScript，Perl，等等）。本书中将包含所有这些特征。

扩展的切入点

随 Windows3.1一起发布的文件管理器有着非常好的且未充分利用的特性，比如：它能够在运行时加载 DLL，执行具有特殊原形的已注册函数等。也就是说，在这个资源中有一点，其代码本身能够知晓某些活动是由用户执行的。换句话说，文件管理器支持扩展行为，当执行特定的活动时，查找注册的扩展并装载和运行之。

我们后面在探测器外壳和命名空间扩展中见到的实际上有相同的原理。不同的完全是细节方面的实现。文件管理器用于加载具有预定义原形的传统的 DLL 全程函数，而探测器是这一过程更加规范。尤其是它采用 COM 接口(可以看作预定义和固定函数原型的集合)和进程内服务器(实质上的 DLL)

当然，COM 接口和进程内服务器比函数集和 DLL 更复杂，但是，这也使探测器进程比老的基于 DLL 的进程更规范和有力。

对探测器的扩展

在探测器环境中，基本上有两种类型的扩展：外壳扩展和命名空间扩展。从名字上看有点混淆，探测器就是 Windows 的外壳，所以两种类型的扩展都可以作为外壳扩展。换句话说，外壳和命名空间扩展二者都扩展了探测器能力。但是在他们之间有一些差别。

外壳扩展是一种加到给定类型的所有文件上的客户行为，给定类型的文件按照客户要求显示在探测器的观察中。如此，你可以称之为“外壳观察的扩展”。客户的行为，如拖拽，右击关联菜单，绘制图标或显示属性对话框等由一定数量的特殊事件触发。你可以定义这些事件的特殊处理器程序，例如你可以确定显示给定.Bmp 文件的图标，为所有 Windows 元文件加一个预览页面到属性对话框，甚至可以在关联菜单中增加一个可执行功能。将在十五章中给出例程。

命名空间扩展有两种形式，这要看你怎样连接。如果你用文件类型连接命名空间扩展，尽管有复杂的代码支持，其功能上仍然等价于关联菜单的扩展。然而，如果你用文件夹连接命名空间扩展，这个文件夹将变成客户文件夹，你的代码将确定文件夹的内容、探测器显示的图标、子文件夹、排序、关联菜单等。

为什么要对 Shell 编成

这个问题很有道理，简单的回答就是，为了使我们的应用根号和更丰满。但是这个回答有点过于辞令化，我们这样做是为了使我们的模块与系统集成到一起，或者说更自动化。

本书的结构

有两种方法对外壳编程，使用 API 函数和使用 COM 接口。这两种方法既不相独立也不相互重叠，它们是两个不同的方向和两种不同领域，这一点我们在下一章中将进一步阐述。现在让我们直接浏览一下个章节的内容。

你知道 Shell 编程要求使用 API 函数和 COM 接口，API 函数使你能够访问 Shell 的基本功能，如，文件操作，浏览文件夹，执行程序，处理图标和快捷方式等。当你想要增强和精炼 Shell 的基本行为时，COM 方法则触及到了客户化 Shell 扩展的核心。

这本书中首先给出所有 API 函数的解释，进一步探讨函数的原型，资料介绍的差别以及其中的 Bugs。通常我的目的是要澄清所有你在资料中遇到的含混之处。第三章到第九章涉及到特殊的 APIs 分组，其中包含了典型的 Shell 操作，特别在第三章中讲解 SHFileOperation() 函数，涉及到文件的拷贝、移动、删除和重命名操作。第四章揭示了神秘的 SHGetFileInfo() 函数，系统提供了获取文件(属性、图标、类型和显示名)的系统信息和 Shell 信息的方法。第五章解释文件夹内部组织的叠放过程，包括设置、浏览和象 Favorites 和 SendTo 那样的特殊文件夹。

快捷方式和属性在第六章中介绍，其中将介绍建立和解析快捷方式和经常使用的字段。第七章则正式进入探测器地址空间，并且从另一个角度讨论客户化问题：在探测器不可知的情况下什么是你所能安全操作的。特别是我们向你展示一个置换过的“开始”按钮和不同的菜单。一旦你这样做过之后，你就有了完全控制 Windows 系统的能力了。在余下的第八和第九章中我们将讲述程序的扩张，图标和任务条，我们将说明怎样编程加入新的具有自己的菜单的按钮到任务条中。

这本书的第二部分是基于要求 COM 接口的探测器特征的。但是直到第十二章之前我们还没有涉及到接口知识，中间的两章作为 Shell 函数和探测器接口的桥梁。第十章包含了最近更新的 Windows 的 SDK 函数。第十一章给出了 Shell 对象的概览，例如“我的公文包”，“控制面板”和“打印机”，以及客户文件夹的概念。在这一章中还包括了其他 Shell 对象和 RunDLL32 使用程序的说明以及全部探测器命令行的解释。

第十二章介绍 Shell 对象模型，首先致力于把 API 函数的一个子集移动到对应的 COM 接口中去，这个特性最少要求系统中要安装“活动桌面”。有趣的是这个对象模型允许你访问系统的某些功能（绝大多数系统对话框）。

第十三章介绍 Windows 脚本环境，这是一个执行 Windows 批处理文件的运行时引擎。技术上讲，这并不是一个 Shell 实体，但是它与 Shell 有重要的留级关系。Windows 脚本环境显露一个对象模型，使你能够使用 VBScript, Jscript 等任何脚本语言编写程序。我将通过加入有用的新对象来扩展这个模型。

第十四章集中于指导你采用 Shell 和命名空间扩展的应用和理由方面。我将揭示实际上的 Shell 集成的应用究竟是什么和为什么说 Shell 扩展是把应用模块与系统 Shell 融合的最好方法。第十五章说明怎样写一个 Shell 扩展来客户化关联菜单、图标和属性，以及怎样排错。第十六章概括了命名空间扩展内容，并且包含一个例子，说明怎样加一个可展开节点到探测器的树观察中，并以文件夹的形式展示了当前窗口的堆栈过程。

小结

这一章中我们描绘了未来各章中打算作的事情，尤其是我们试图解释：
Shell 的本质和结构
各 Shell 版本之间的差异

第二章 Shell 的结构

“Shell 编程”的大伞之下有大量的 API 函数和 COM 接口。这个种类繁多的‘命令’集允许你用不同的方法对 Windows Shell 进行编程。函数和接口并不是两种提供相同功能的等价途径，相反，它们在不同的逻辑层上提供不同的功能。

API 函数包含了用户想要在 Shell 对象上执行的基本操作，如文件和文件夹。COM 接口则给出了扩展增强，甚至客户化各种要素对象的机会，包括 Shell 本身标准行为。用面向任务的方法对函数和对象进行分组将给我们一个总体上观察 Shell 的机会，因此，我们仍然能够把它看作一个具有属性和方法的对象。在这一章中我们将努力分出每一个函数和接口究竟属于哪一个功能组。这将有助于我们从大量的功能碎片中寻找出 Shell 编程接口。

在这一章中，将包含有：

我们在这本书中使用的定义

Shell API 函数的功能分组

由 Shell 和其内涵部件实现的 COM 接口的功能分组

Shell 结构是怎样随导入的活动桌面而演化的

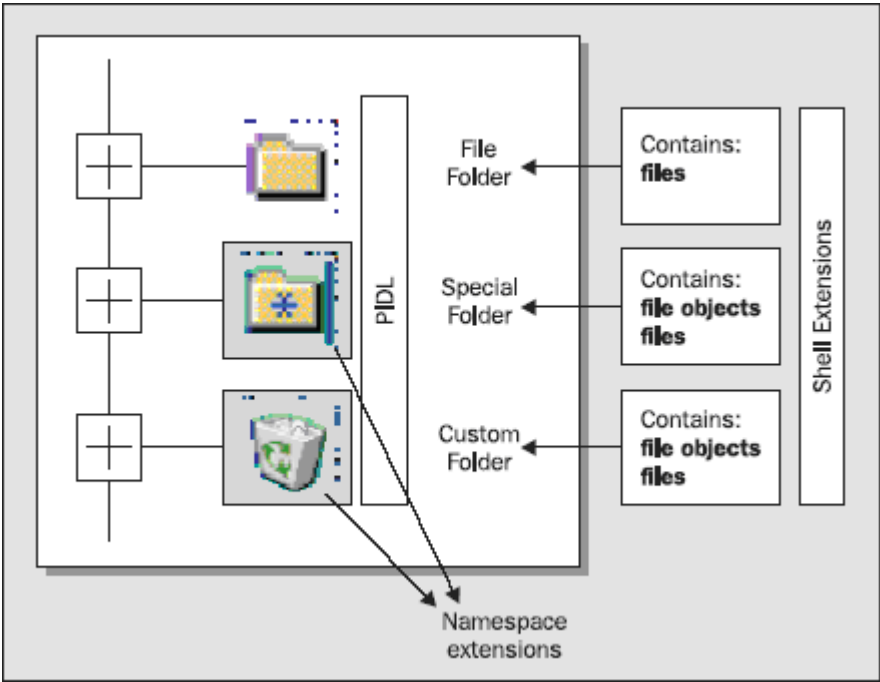
最后，你将能更好地理解这本书的知识范围，并且作为 Shell 程序员，你将能清晰地勾画出书中的哪些功能是可用的。

Shell 模块

实际上到目前为止我们还不能说 Windows Shell 是面向对象的。有一些‘对象’查看结构就能识别它。这些对象也有‘属性’一样的特征，以及象方法一样执行一些活动。但是它们是依赖 API 函数的，一个典型的对象就是文件夹。

如果 Shell 不是面向对象的，它决不能有一个完全兼容的对象模型。我们能够想象一个看起来象分

层对象的体系结构。换言之，有一个对象集合，他们以如下图描述的方法一样工作。



基本上讲，Shell 是由文件夹组成的，文件夹是一个包含有子元素的容器，包括子文件夹，这些元素通常称为**文件夹项**。根文件夹称为‘桌面’，其子项包括‘我的计算机’，‘网上邻居’，‘回收站’，和其他可能的项，所有这些文件夹的集合称之为 Shell 的**命名空间**。

Shell 的命名空间

概念上讲，文件夹就类似于文件系统的目录，但是它不一定绑定到一个实际的物理目录上，如果他邦定，就称之为虚拟文件夹。我们可以以这种方式区分两种主要的文件夹：正常的文件夹（命名为文件型文件夹或目录）和**客户文件夹**。自然，包含在文件型文件夹中是文件，其属性是名称、类型、尺寸、最后修改日期等。包含在任何其他文件夹下的项目可以是文件——一般使用其他的扩展特征集——但是也可能是完全不同的东西，如打印机或网络节点。

文件夹

文件夹是怎样实现的？文件夹实际上是一个 Shell 对象，它的行为被编码成一个 COM 模块，向 Windows Shell 暴露公共的接口。通过连接，文件夹可以告诉 Shell 怎样设计它的内容，使用什么样的图标显示，采用什么文字来描述，例如‘我的计算机’看起来像一个文件夹。他有一个代码层来感知 PC 上所有可用的驱动器，并且为每个驱动器附加一个子树到探测器的观察中。

每一种不同的文件夹都有不同类型的层次代码来提供不同的行为，对于文件型文件夹，行为就是

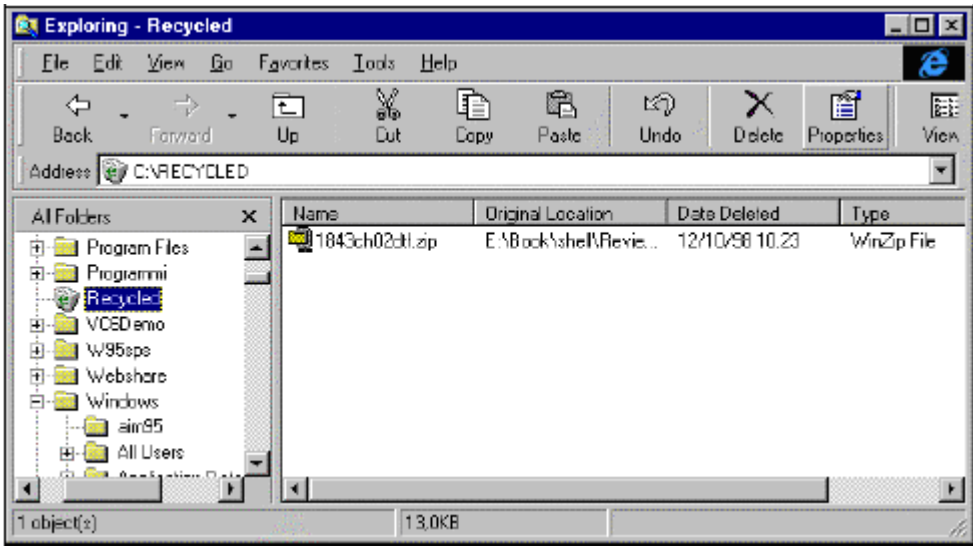
扫描文件系统，恢复文件和子文件夹，并且通过列表控件显示它们。而 打印机文件夹则记录所遇到的和所安装的打印机，并且为每一个都显示一个图标。你可以设计任何类型的具有任何行为的虚拟文件夹。文件型文件夹（即，目录）只 是其中的一种。

对非文件型文件夹，Shell 的资料相对较少，仅在**特殊文件夹**中有些说明。事实上 Windows Shell 默认提供的是客户文件夹，他们与文件型文件夹的差别是：

- 可以包含文件和其他对象
- 能够提供对内容的不同观察
- 可以有选择地不被绑定到物理目录上
- 是系统定义的一部分，这部分由 SDK 提供特殊的函数集。

特殊文件夹列表可以在 Win32 SDK 资料和后面的第五章中找到。就像我原先说过的一样，特殊文件夹是具有自己 COM 模块提供行为的特殊类型的文件夹。由于 COM 模块是新节点被加到 Shell 命名空间的前提，所以它就被称之为**命名空间的扩展**。

特殊文件夹使用户能够经过适当的接口访问系统信息。也就是说，在大多数情况下，这种文件夹与典型的文件型文件夹提供的内容观察多多少少有些一致的地方。当然，精确的程度依赖于文件夹的类型。



与普通文件夹一样，特殊文件夹也可以包含文件。然 而，通常是以稍微不同的方法表示，显示不同的特征。因此，特殊文件夹给文件赋予了不同的意义，并且，不把它们当作文件系统的正常实体（如果不是这样，它就 不是特殊的了）。例如‘回收站’含有正常的而隐含的文件，因为这个文件夹要显示当前被标志为删除的文件列表，所以它把初始位置和删除日期特征显示在前面。

绝大多数（不是全部）特殊文件夹都依附于磁盘上的物理目录，正常情况下这是一个只读的目录，其内容就是所有需要以最适合的方法显示的信息。

换一个视角，绝大多数特殊文件夹都需要一个目录来存储它们的数据。这个目录可以被分配到磁盘的任何地方，并且表示为文件夹和 Shell 支持的链接——这个特殊文件夹在命名空间中的位置。目录的内容不必显示为文件列表，相反，关联文件夹的代码可用最适合于它的角色的形式解释和显示它。

文件夹这个有着包含任何事物能力的东西导出两个重要概念：**文件对象**和 **PIDLs**，这些我们将在后面章节中叙述。

文件对象

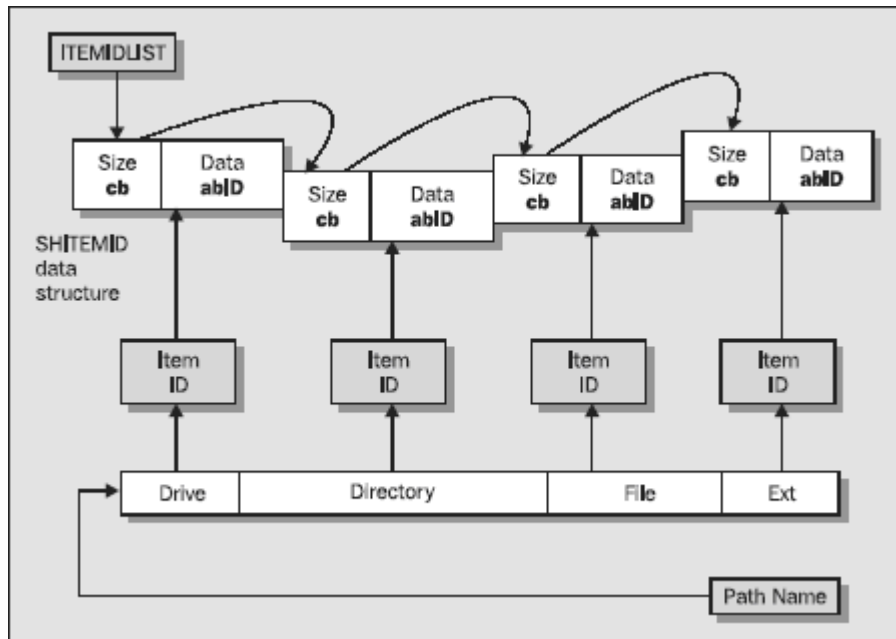
文件对象是一个包含在普通文件夹中的项——文件、记录、内存块、连接的设备等。‘文件夹项’、‘文件夹元素’和‘文件对象’这些术语是等价的。如果文件夹是一个文件型文件夹则文件对象就是文件。因此这里的‘文件’就比‘文件对象’稍微特殊一点，因为它精确地代表了文件系统中的实体。文件是一个文件对象，但是，文件对象不一定是文件。

有一个敏感的问题出现在一般的文件夹和文件夹项的概念中，在 Shell 命名空间中我们怎样才能安全并唯一地区分出其中的项。如果 Shell 与文件系统一致（就像 Windows 3.x 一样），则文件的全名就能极好的保证这种唯一性。不可能有两个文件具有相同的路径和名称。然而当文件夹比文件目录变得更普通的时候，区分其中的项就需要更普通的方法了。

PIDLs

PIDL 是一个数据结构，它是唯一地标识包含在文件夹中的项的一种方法。PIDL——标识符列表指针的缩写（**p**ointer to an **i**dentifier **l**ist）——是一种比文件全名更通用的方法，它不仅在文件夹内而且在 Shell 的整个命名空间中保证了项的唯一性。更重要的是，它能透明地处理文件和文件对象。为了理解 PIDLs 的结构和作用，我们来分析一下它的二进制结构并与之所替代的路径名进行比较。

一个文件全名就是一个字符串，是一个具有非常特殊格式的字符串，是一些子串的串联，每一个子串都在文件系统的层次中表示一个层，有驱动器名，然后是目录名，文件名，最后是扩展名，他们都由反斜线分隔。你所了解的文件全名就是指向这些相连元素的指针——此时指向的是一个字符串。从概念上讲，你可以把它看作是一个数组结构，其中的每一个元素都表示了一个路径名元素。



上图说明了路径名和 PIDL 的关系，同时他也给出了标识符列表在存储器中组织结构的概念。从编程的观点讲，PIDL 是由 LPITEMIDLIST 类型实现的，它是 ITEMIDLIST 结构的指针。

```
typedef struct _ITEMIDLIST
{
    SHITEMID mkid;

} ITEMIDLIST, *LPITEMIDLIST;
```

中间构成路径名各部分的对象映射到 PIDL 的**项目标识符**上。它们存在于整个 SHITEMID 结构中。

```
typedef struct _SHITEMID
{
    USHORT cb;
    BYTE abID[1];

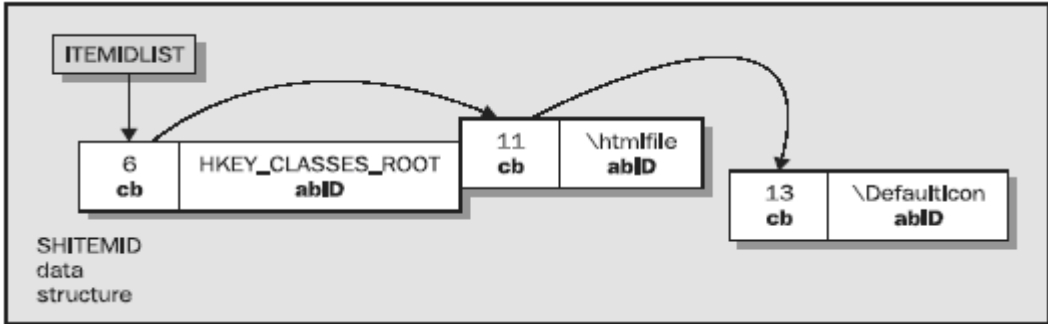
} SHITEMID, *LPSHITEMID;
```

结构的头两个字节指示项目标识符的尺寸——即，相关元素的数据以及用于表示的数据所占用的字节数。**cb** 值必须包含它自身的尺寸。对应路径名，**cb** 应该是所表示的驱动器或目录的长度加上一个 **unsigned short** 类型变量的长度。随后是这个结构数据的第一个字节。

一定要记住 PIDL 是一个‘平面’结构，不包含指针。形成 PIDL 的所有数据必须明显地聚集到一起，

而不是通过指针连接。这就是说，我们不能使用典型的链表结构方案，使一个元素的最后成员指向链中的下一个元素。还有一点，就像图中所看到的，链表中下一个元素的地址可以通过 `cb` 相加到当前 `SHITEMID` 对象计算得出。这是设计规定的，因此要求相连的 `SHITEMID`s 要连续分配空间。

定义 `PIDL`s 的构造规则是约定实现文件夹行为的代码。这些代码也应该确定使用什么样的数据来表示标识符的项。例如，假设想要实现一个文件夹，象文件系统那样显示 `Windows` 注册表，‘子文件夹’应该是注册表键‘文件对象’应该是注册表值。在这种文件夹中表示每一个元素的可能方法应该是使用相关的键名。这里我们能够看到 `PIDL` 是怎样使用与前面图中给出相同的方案格式的。注意 `HKEY_CLASSES_ROOT` 是一个长整型值，所以它有四个字节加两个字节的无符号短整数。



项目链表表示了路径踪迹，从命名空间的根到特定文件夹项。这个标识符链表聚集了链条上的所有元素，说明了一种通过 `Shell` 唯一地标识一个元素的方法。保证两个项目标识符在内存中连续分配是文件夹对象相关代码的职责。尽管路径名与 `PIDL`s 类似，他们并不等价，他们也不能交互使用，他们是不同的数据结构。

Shell 观察

任何文件夹的内容都是通过一个对象调用 **Shell 观察** 显示在 `Windows` 探测器中的。每一个文件夹都定义了他自己的 `Shell` 观察对象，并且所有相关于这个用户接口的任务都指派到这个对象上。对于文件型文件夹 `Shell` 观察对象是用列表观察控件实现，其中的项就是文件和子文件夹名。默认的 `Shell` 观察对象在他被调用处理文件时为每一个文件分配图标、**显示名**和**类型名**。

图标有几种方法确定，这依赖于请求文件的性质。一般使用自身定义的图标显示图标文件 (`.ico`)，而程序文件则显示其资源中定义的头一个图标。如果没有定义图标，则显示默认的。对于所有其他文件，`Shell` 通常采用文件归属类所定义的图标。然而正象下面要揭示的那样，这个行为可以被客户化。

在整个 `Shell` 环境中，文件都是根据文件扩展名指定的类型分组的，这种根据类型形成的文件集通常称之为**文件类**。它与一个图标和一个描述字符串相连，这个字符串显示在 `Windows` 探测器观察的详细信息窗口上的类型列上。然而，要置换它们，指定的文件类就需要在注册表中注册，`Shell` 将从那里读出类型信息和图标。

一旦定义了文件类，你就可以写代码来影响和修正 `Shell` 响应某些发生在特定文件类上事件的默认行为，这其中就包括绘制文件图标，弹出关联菜单，和显示属性对话框等。通过定义 **Shell 扩展**，你就可以动态地确定这些事件发生时要做些什么。例如，可以在关联菜单中加入新的项，和处理用户的点击，和动

态地确定基于每个文件的图标显示。

钩住 *Shell*

一般情况下，Shell 扩展可以看作是钩子，他被设置在整个 Shell 中。Win32 中，钩子是一段由应用定义的代码，一定事件发生时系统回调这段代码。有许多不同类型的钩子，他们的应用也非常广泛，有一些仅仅影响安装他们的应用程序，而另一些则影响所有系统中运行的应用。

这其中典型的例子就是键盘钩子，它能够使你在相应消息发送到应用窗口之前得到键盘按下的信息。其他钩子的例子如鼠标活动（移动，点击），窗口管理（建立，析构，活动），和消息处理。更多信息请参见 Win32 SDK 资料。

从程序员的观点看，钩子是一个具有固定的和预定义语法的**回调函数**，作为回调函数，系统基于已知的原形调用它。Shell 扩展是 COM 接口，而不是回调函数，但是背后的原理是相同的，二者都允许你指定某些系统将要执行的代码来处理一些预定义的活动。

这一节特别注意到 Windows 的钩子。通过设置局部钩子，你仅仅能够捕获相关应用内发生的事件。但是设置全程钩子将会导致钩住任何运行的应用所发生的事件。设置全程钩子就是说，你的应用定义了一段代码，它可以被运行中的其他相关进程执行。事实上使用钩子完成 Win32 的跨进程边界和注入代码到其他进程地址空间是最容易的方法。它也是能在所有平台上工作的唯一方法。

Shell 地址空间

注入代码到关联的另一个进程是重要的，因为，它允许你访问另一进程没有公开的对象，这对 Shell 编程尤其重要。当你成功地把代码插入到 Shell 地址空间后，你就可以查询 Shell 接口，改变用户接口，甚至置换‘开始’按钮。

全程钩子是一种使你的代码运行在 Shell 的地址空间中的方法，但是更有力和更灵活的机理是提供**浏览器帮助对象**——一种 COM 对象，探测器和 IE 在启动主窗口时自动加载的对象。

Shell 内存分配器

在使用 Shell 时很快你就会接触到内存分配的问题，Shell 提供了一个**存储分配器**，这个封装了 IMalloc 接口的服务可用来代替 New 或 GlobalAlloc()。

要获得这个对象的引用，你应该使用 SHGetMalloc()。它不是返回一个 IMalloc 接口的新指针——由 CoGetMalloc() 函数返回的——而是由系统 Shell 对 IMalloc 对象的一次引用。使用这个指针，你可以安全的释放由 Shell 分配的内存，并且使 Shell 释放这块内存。这可能有点陌生，但是在 Shell 编程中，这是个好习惯。

Shell 任务条

任务条窗口作为 Windows 用户接口的一个已知的部件，仅仅是因为它包含了‘开始’按钮。然而我们之所以称之为‘Windows 任务条’，是因为它实际上是一个窗口系列的特例，称之为‘应用桌面工具条’，最好的例证就是 Office97的快捷方式杆。有一个特殊的函数和消息集与桌面工具条相关，然有趣的是仅有少量函数和消息影响到 Windows 的任务条。因此，即使资料没有明确地说明，系统任务条和桌面工具条仍然是不同的对象。

关于任务条的另一个错误观点是它包含了所有运行中应用的按钮，但是有两点原因说明这不是真的：

不是所有运行着的应用都显示在任务条上

作为按钮，任务条的唯一有的是‘开始’按钮

无论是否相信，作为按钮集出现的实际上是 tab 控件，只是具有特殊的类按钮风格罢了。

任务条起到了系统控制板的作用，使你能够访问所有运行中的应用。在很多情况下，我们希望能够限制任务条的功能—这是运行在公共 PCs 上应用的一个典型的需求，在那里你不希望用户能够运行其他程序或浏览文件系统。Win32 API 并没有提供丰富的函数来操作任务条，但是，我们将试图在第九章中对此进行一些补救。

Shell API 函数

在与 VC++6.0一起提供的 MSDN 库的 Shell 参考一节列出了100多函数，然而，其中的大多数都只涉及非常特殊的领域，有时感觉就象是 Windows Shell 的边界领域—这里所说的特殊是关于文件分析和屏幕保护的例程。

在这本书中，你不能找到关于每一个函数的详尽的说明，然而我们可以集中于文件和文件加操作的核心函数，并试图澄清他们含混不清的资料说明。为了有助于对其进一步分类，我们把它们分作五个不同的函数组。

组	功能
一般 Windows 函数	涉及到屏幕保护，控制面板脚本程序，联机帮助，以及 Shell 拖拽（不是 OLE 拖拽）
Shell 内部函数	访问探测器地址空间的函数，获得 Shell 存储分配器的函数，导出可执行程序的函数以及感觉用户接口改变的函数。 涉及到托盘域的函数和与 Windows 任务条通讯

	的函数
任务条函数	操作文件的函数，他们执行如‘拷贝’，‘移动’，‘删除’和‘取得信息’等操作的系统活动，和添加文件到特殊的系统文件夹如‘最近文档’等。
文件函数	操作文件夹的函数，使用这些函数，你可以浏览文件夹，获得系统文件夹的路径，发现文件夹的设置。
文件夹函数	

根据这个分组结构，可以看到有几个函数作为 Shell 编程接口的一部分并没有被显式引用，但是，他们仍值得出现在这个表中。

组	功能
图标函数	从执行文件中抽取图标的函数
环境函数	处理环境变量的函数
Shell 轻量级 API 函数	容易地访问注册表的函数，读写注册表函数，处理路径名函数，和处理字符串函数。

特别是，有些操作图标和环境变量的函数在 `shellapi.h` 头文件中，是我们在这里说明他们的主要原因。就像对 Shell 轻量级 API 函数一样（在第10章中详细说明），我们说这些函数可以放到任何一类中，但是，他们对 Shell 编程而言有特殊的用途。下一节的表中列出和描述以这种分类方式定义的一些函数。之所以如此，是要你更好地理解函数操作的概念，以及给你提供一个快速查找其中函数的地方。

一般 Windows 函数

正象标题所提示的那样，这些函数仅仅稍微地涉及到 Windows Shell，在绝大多数情况下，他们都直接来自于 Windows 3.x 的 API—他们仅处理如帮助文件和拖拽等操作，所有这些函数都很好地支持32位

Shell 版本。

函数	描述
DragAcceptFiles()	标记允许窗口认可拖拽操作。
DragFinish()	从 Shell 中释放移动文件名列表所分配的内存
DragQueryFile()	从 Shell 处理拖拽而分配的内存块中抽取文件名
DragQueryPoint()	获得拖拽发生的点位置
CPLApplet()	控制面板脚本小程序的主程序
GetMenuContextHelpId()	返回关联于给定菜单的帮助 ID
GetWindowContextHelpId()	返回关联于给定窗口的帮助 ID
SetMenuContextHelpId()	设置关联于给定菜单的帮助 ID
SetWindowContextHelpId()	设置关联于给定窗口的帮助 ID
WinHelp()	打开帮助文件
ShellAbout()	显示默认和特定客户化的‘关于’信息框

Shell 内部函数

这类函数包括 Shell 底层操作函数，和使你能够进入到 Shell 的地址空间以及可以从一旁操作它并获得对其次年初空间访问的函数。

函数	描述
ShellExecute()	在指定的文件上执行特殊操作
ShellExecuteEx()	与上面函数相同，但是有更多的选择
SHChangeNotify()	通过这个函数程序能够让 Shell 知道什么变化了，以及要求它刷新它所保有的信息
SHGetInstanceExplorer()	返回探测器 IUnknown 接口指针
SHGetMalloc()	返回一个指向 Shell 存储分配器的指针
SHLoadInProc()	装载指定的 COM 对象到探测器地址空间

任务条函数

Windows Shell 并没有定义多少操作任务条的函数，所以，控制任务条经常需要自己做很多工作，然而，有两个函数与任务条相关：

函数	描述
Shell_NotifyIcon()	显示和管理靠近时钟的托盘区域的图标
SHAppBarMessage()	发送消息到系统的任务条

文件函数

文件是 Windows Shell 最重要的元素之一，图形环境需要文件有许多不同的特性，因此需要特殊的函数来处理。注意，在下表的**版本**列中显示的内容，有些函数是根据最近的 Shell 版本介绍的。

函数	描述	版本
FindExecutable()	返回指定文件名注册的可执行文件路径	所有版本
SHAddToRecentDocs()	把给定文件的连接加到系统的‘最近文档’文件夹中。	所有版本
SHFileOperation()	用于拷贝、移动、删除或重命名一个或多个文件。	所有版本
SHFreeNameMappings()	释放 SHFileOperation()函数在特定情况下返回的存储结构	
SHGetFileInfo()	返回给定文件的各种信息块	所有版本
SHGetNewLinkInfo()	建立新的快捷方式名	4.71

文件夹(Folder)函数

就像我们已经讨论的，文件夹比目录更普通一点，它可以包含文件以外的东西，因此文件夹背后的软件就直接涉及到为其中的每一个项返回一个唯一的标识的问题。在活动桌面下，文件夹也可以有它自己的图形特征集。

函数	描述	版本
SHBrowseForFolder()	显示选择文件夹的对话框	所有版本
SHEmptyRecycleBin()	销毁‘回收站’的内容	4.71
SHGetDataFromIDList()	从标识符表中恢复数据	所有版本
SHGetDesktopFolder()	返回‘桌面’文件夹的 IShellFolder 指针	所有版本
SHGetDiskFreeSpace()	返回指定驱动器的磁盘可用空间	4.71
SHGetPathFromIDList()	返回指定标识符列表的路径名（如果存在）	所有版本
SHGetSpecialFolder Location()	返回特殊的系统文件夹的标识符列表	4.71
SHGetSpecialFolderPath()	返回系统特殊文件夹的路径名（如果存在）	所有版本
SHGetSettings()	返回文件夹当前设置的值	4.71
SHInvokePrinterCommand()	向打印机发送命令	4.71
SHQueryRecycleBin()	返回‘回收站’当前占有的空间	4.71

图标函数

图标是 Windows 图形环境的中心，操作系统外壳最显著的部分。因此，普遍认为，图标是 Windows Shell 编程接口的中心。

函数	描述
ExtractIcon()	返回可执行文件的图标 Handle
ExtractIconEx()	与上函数相同，但是有更多的选择。
ExtractAssociatedIcon()	基于文件类，返回指定文件的图标 Handle

COM 接口

我们可以使用 COM 接口就象使用 API 函数那样对 Shell 作类似的操作。再有，使用与 CV++6.0 一同提供的 MSDN 库做为参考，我们可以将涉及到 Shell 相关的 COM 接口分成四类。

组	接口
---	----

Shell 扩展	涉及到所有 Shell 活动的 COM 接口，从图标到关联菜单，从 UI 活动到文件观察
Namespace 扩展	涉及到命名空间扩展的 COM 接口
钩子	能够钩住某些东西的接口，特别是程序执行，URL 转换和建立 Internet 快捷方式
杂项接口	一些零碎接口，如客户化任务条的接口，与打开对话框通讯的接口和对‘我的公文包’编程的接口

对开发者，这些接口并不总是必须实现的——在某些情况下，紧紧需要知道它们，能够适当地调用它们的方法就足够了。下面就更详细点介绍它们。

Shell 接口

在开始，我们展示所有 COM 接口，然后利用它们在 Shell 及其扩展上做一点文章。

接口	描述	版本
IFileViewer, IFileViewerSite	使你能定义对给定类型的文件提供‘快速观察’处理器的模块。	所有版本
IInputObject, IInputObjectSite	这两个接口用于处理 UI 活动和对具有接收用户输入的 Shell 对象进行加速操作处理。	4.71
IShellIconOverlay, IShellIconOverlayIdentifier	用于发送文件图标重叠消息，使你能知道用于给定文件的重叠形式。一个图标重叠是 Shell 绘制在图标上的 Bitmap 图像，以便更好地表现它，如，一个手形重叠表示文件夹的共享。	4.71
IContextMenu, IContextMenu2	允许为特殊类型的文件添加新的关联菜单项。 IContextMenu2处理自绘菜单	所有版本
IContextMenu3	与 IContextMenu2相同，但是给出了更好的键盘控制。	4.71
IShellExtInit	执行一个 Shell 扩展的初始化	所有版本
IShellChangeNotify	SHChangeNotify() API 函数在 Shell	4.71

	扩展上的副本，基本上，它允许你写一个模块钩住由 SHChangeNotify() 函数通知的 Shell 层上的变化。	
IExtractIcon	允许你获取任何文件夹项的图标信息。	所有版本
IShellIcon	提供另一种获取任何文件夹项图标信息的方法，在特定情况下，这种方法优于 IExtractIcon 方法。	所有版本
IShellLink	允许建立和解析文件和文件夹的快捷方式	所有版本
IShellPropSheetExt	用于为指定文件类增加属性页到‘属性’对话框。	所有版本

命名空间接口

要写一个命名空间扩展，本身就需要熟知大量的 COM 接口。这里仅列出最重要的和必须的一些。

接口	描述	版本
IShellView, IShellView2	用于定义命名空间扩展的观察对象。IShellView2 仍然没有文档资料，但是在基于 Web 的观察中有使用。	所有版本
IShellBrowser	显示浏览器，他就是探测器或 Internet 探测器。	所有版本
IEnumIDList	提供 Shell 枚举文件夹内容的方法。	所有版本
IShellFolder	提供令 shell 以标准方式处理客户文件夹的方法。IShellFolder 对探测器隐藏客户代码。	所有版本
IPersistFolder	使你能初始化某些 Shell 扩展和所有命名空间扩展。	所有版本
IPersistFolder2	与上相同，加入了一些对基于 Web 的观察更强的支持。	4.71
IQueryInfoRetrieves flags and infotip text for items in a folder. 4.71	恢复文件夹项的标志和信息标签文字。	4.71

钩子接口

Windows Shell 给我们的模块一个机会来感觉一定数量的事件，并使我们可以把客户代码加入其中。

接口	描述	版本
ICopyHook	能钩住 Shell 中的所有文件操作(拷贝、移动、删除、重命名)。	所有版本
IURLSearchHook	使你能够探知探测器正在试图转换一个不可知的 URL 协议。	4.71
INewShortcutHook	使你能够探知探测器正在试图建立新的 Internet 快捷方式。	4.71
IShellExecuteHook	能够钩住通过 ShellExecute()或 ShellExecuteEx()导出的所有新进程的启动。	所有版本

杂项接口

覆盖 Shell 编程特殊领域的其它接口统称为杂项接口，如：‘我的公文包’，通用对话框，和任务条等。

接口	描述	版本
INotifyReplica, IReconcilableObject, IReconcileInitiator	所有这些接口都涉及到文件调整过程。最终都产生同一个文档的更新版本。	所有版本
ICommDlgBrowser	当客户文件夹嵌入到通用对话框中时，提供特殊的浏览行为。	所有版本

ITaskbarList	允许在系统任务条中加入新的按钮。	4.71

为什么又有 API，又有 COM

现在我们已经看到了 Windows Shell 所有的功能，需要花费一点时间才能给出 API 函数和 COM 接口的作用。本质上，整个 Shell 功能可以划分成两个领域，基本功能和扩展功能，从这个观点分析，就很容易区分哪一种方法属于哪一个领域的了。

现在，大多数由 API 调用提供的功能可以看作调用“Shell”的伪对象的‘方法’。这个伪对象允许你移动或拷贝文件，或浏览文件夹。你也可以恢复给定文档的信息，等等。对象模型的头一个特性就是从描述它本身开始的。

换句话说，Windows 初始是由纯 C 设计的，从没有被考虑过以面向对象的概念进行设计。因此，所有的基本功能都通过直接的 API 调用给出也就不奇怪了。

COM 技术允许写出部件模块，然后通过选择暴露它的接口来使用它们。使用接口很容易聚集相关函数并提供对给定对象的访问。站在 Shell 的立场上看，COM 接口就是封装的 API 调用——你可以在 ITaskbarList 接口中看到，这是头一个 COM 而不是 API 调用暴露的系统部件编程接口的例子。

这种模式的另一个例子是我们上面提到的钩子接口。在 Win32 SDK 中全部钩子都是通过回调函数而不是 COM 接口编程的。换句话说，Shell 编程接口包含有钩子，这就要求你编写并适当注册一个 COM 服务器来实现。实际上，差别不是很大，但在体系上，他们就不同了。

有一股变革之风从 Windows Shell 吹来，COM 就是他的源泉。在已经提到的例子中，可以看到，所有 COM 接口都被用于扩展探测器的行为。由于探测器需要设计进程内服务器，因此，他们的技术是平行的，API 调用和 COM 接口技术同样重要。它们可以被看作为一个硬币的两面(这个硬币就是 Shell)，但是它们确实是有差别的。

活动桌面有什么变化

活动桌面外壳的更新带来一些新的特征，并且使 Windows Shell 产生了几方面的变化。它在任何可能的地方都鼓励使用 HTML，引进了 Web 观察的概念，文件夹的客户化，脚本能力，简化而有效的对象模型，以及大把的新函数和 COM 接口。

上面列出的最后一项应特别引起注意，例如，我们现在有了一个非常原始的 Shell 对象模型，通过 COM，暴露了一些 Shell 的功能。在大脑中记住这些对于程序员来说是重要的。到目前为止，这个模型还不完善，没有你所期望的灵活性，但这是重要的第一步。

抛开 Shell API 的变化不谈，活动桌面表现了桌面结构和文件夹的值得注意的演化。特别是：

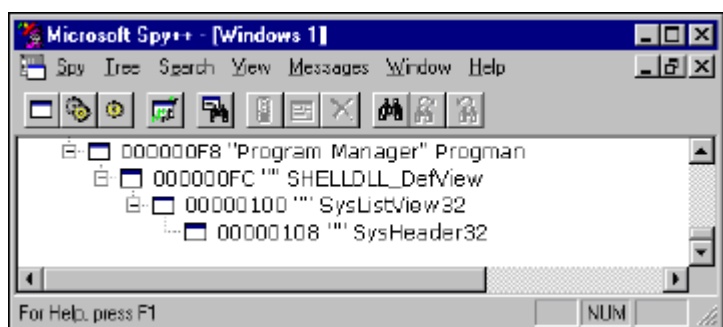
Shell 观察对象

任务条结构

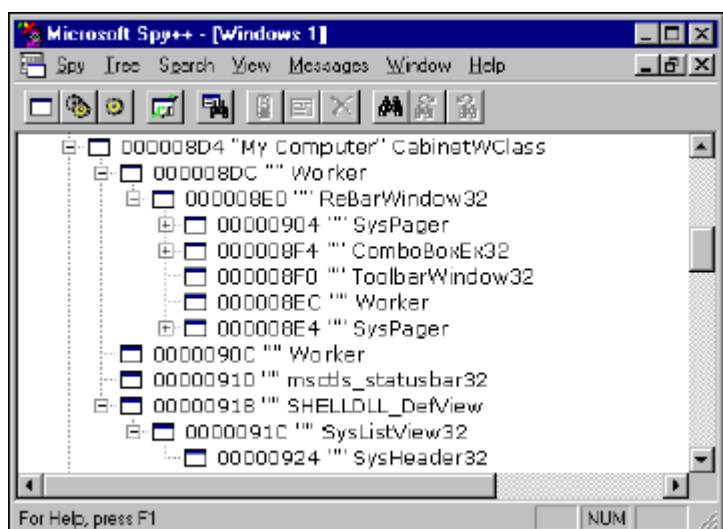
此外还有 Shell 观察对象的增强，以此我们可以在文件夹层上执行脚本代码，以及使用动态 HTML 和脚本程序。

新的 Shell 观察对象

最初，Shell 观察对象是通过窗口栈顶的 SHELLDLL_DefView 类来实现和表示的。在第一章中你也已经看到了：

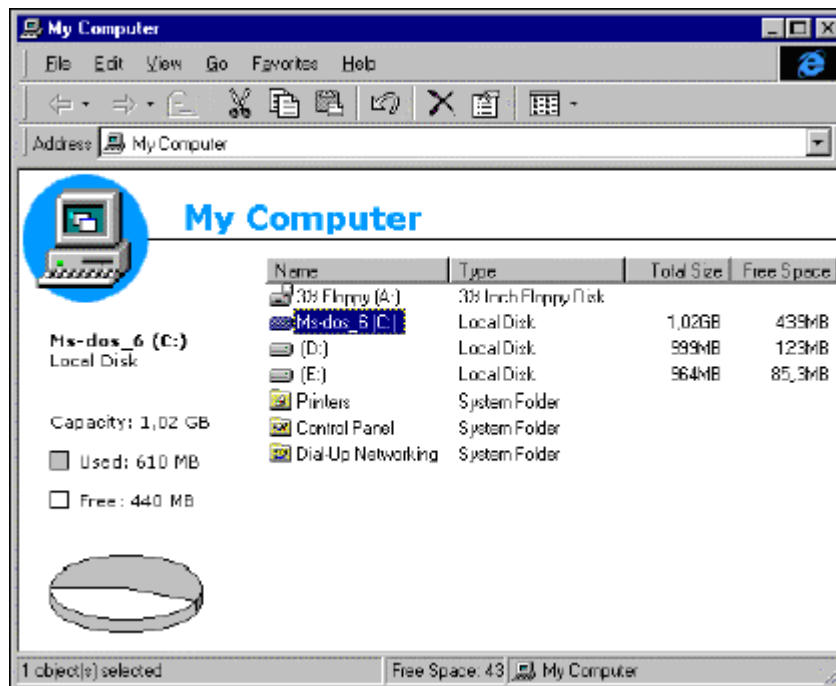


这个截图显示了桌面的观察对象，然而，实际上它对任何文件夹都是一样的。例如下面的图像显示了‘我的计算机’文件夹的窗口堆栈情况：

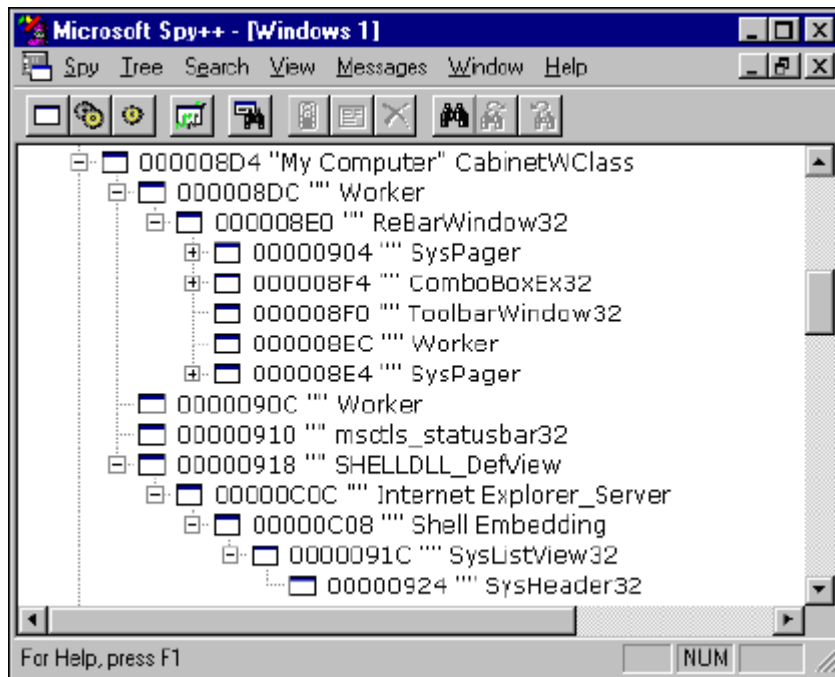


这里所看到的绝大多数窗口一起合作形成文件夹窗口的框架（窗框，组合框，工具条等）。提供显示实际文

件夹内容（即 Shell 观察对象）的总是窗口的 SHELLDLL_DefView 类与他的后代，列表观察。然而对活动桌面，有另外一种观察对象，这个对象还包含有对 HTML 和脚本的支持，称之为 **Web 观察**，并且可以使用文件夹的‘观察作为 Web 页’菜单功能打开和关闭。下面就是在 Web 观察打开时，‘我的计算机’窗口所看到的。



文件夹的内容以基于 HTML 的模版方式显示，其中的列表观察包容了文件对象详细信息的控件。对应的窗口堆栈为：



应立即注意到的最大差别就是窗口类 Internet Explorer_Server，它有一个子窗口类 Shell Embedding，所有这些形成了通过 WebBrowser 控件显示输出的窗口，而 Shell Embedding 则是一个封装了文件列表控件的列表观察窗口。

WebBrowser 是一个 IE3.0 以上版本使用的 ActiveX 控件，用以显示他们的内容：HTML 文件，GIF 和 JPEG 图像，和活动文档。

概括地讲，如果 Web 观察打开，则

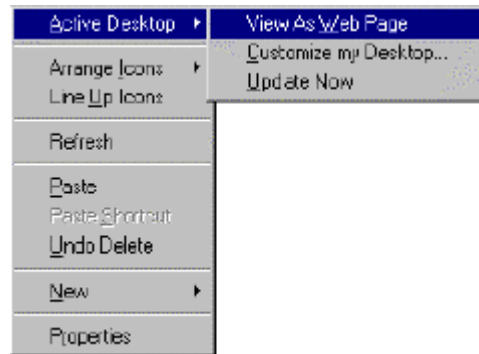
文件夹看上去是由 WebBrowser 控件显示的 HTML 页面

HTML 页面从 HTML 模版生成，它在必要时可以被客户化。

包含文件的列表观察被嵌入到 ActiveX 控件中，并一起并入到 HTML 页面中。

Web 观察也可以在客户文件夹上打开，但是，这种情况下，封装文件夹的命名空间扩展就需要实现特殊附加的接口。

在桌面上事情也是一样的，你可以通过关联菜单打开和关闭 Web 观察：



当这个观察活动时，桌面的观察对象也使用 WebBrowser 控件显示桌面内容。桌面的图标在不同的背景更高层上绘制，尽管这种‘图标层’在以前的活动桌面上也存在，Web 观察还是加入了一些 HTML‘墙纸’的东西，其内容总是显示在图标的下面。

客户化文件夹

当 Web 观察打开的时候，你所访问的文件夹使用 HTML 模版显示。有一个标准文件夹模版文件 Folder.htt 存储在 Windows 的 Web 子目录下，在没有指定其他的模板之前，它是默认的。如果想要学习它的源码，要注意，他是一个隐藏文件，所以，在打开‘显示所有文件’的设置之前，你不能看到它。

通过右击文件夹，打开一个菜单，如图：

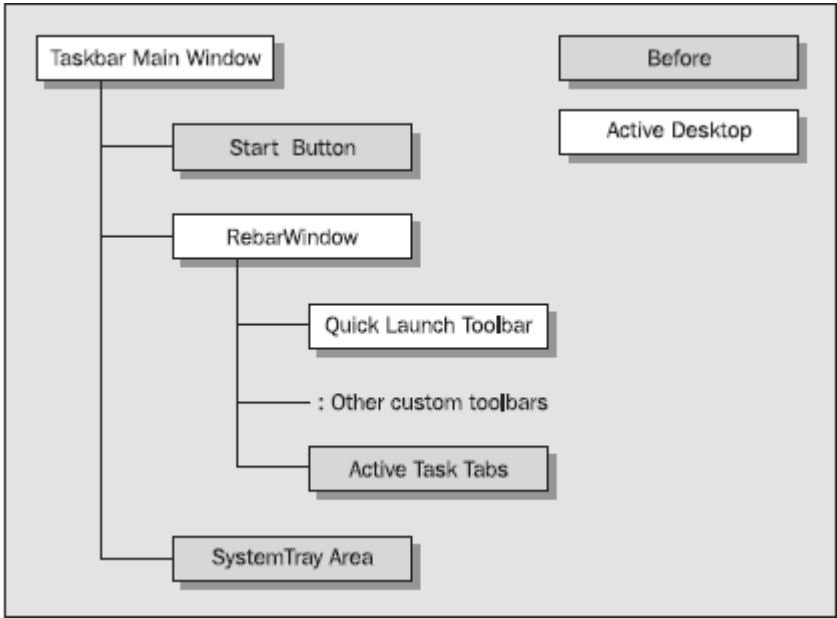


选择‘客户化文件夹’...允许你直接运行编辑大师编辑 folder.htt 文件的内容。更精确地说，你实际所编辑的就是在指定文件夹中由编辑大师最初建立的模版。只要你需要，完全可以通过简单地编辑这个 HTML 文件改变文件夹的外观。尽管这个.htt 扩展是一个完美的 HTML 文件，只要你想，你还可以删除或置换这个文件的列表控件，仅仅显示你想要用户看到的信息。

由于文件夹模版是一个通过 IE 的 WebBrowser 可观察的 HTML 文件，所以，你可以采用所有 XML 的特征，数据绑定，动态 HTML，和脚本功能，以及改变一个简单的文件夹使其看起来像一个应用程序。这样的客户化也相当于类似于原始的命名空间扩展。

新任务条的布局

与观察对象的改变一样，活动桌面的任务条布局也发生了一定的变化。下图中给出了预期的概念，以及新老结构的比较：



小结

这一章，我们讨论了：

从功能上对 Windows Shell 的 API 函数和 COM 接口进行分组

这本书的章节布局

概述了 Shell 结构及其对象

随着讨论的深入，我们总是试图使 Shell 编程接口的结构清晰印在你的脑海中，因而，在下一章中，将包含那些最关键的 API 函数的详细说明。描述代码主要使用 C++调用 SDK 函数。

然后，我们将开始向 Shell 和命名空间扩展靠近，观察一些有用的 COM 接口，以便使用这种方法钩住和对其编程，以及测试这个初始的 Shell 对象模型。仍然有很长的路要走，到目前为止我们甚至还没有看到任何代码。这些都需要花费我们的一定的时间。

第三章操作文件

我依然清楚地记得，Windows95的贝塔版出现的情形，它在朋友之间和学院中传播，好酷，全新的文件管理器，一种全图标，全彩色可客户化的界面，以及活泼的动画标识使得在文件拷贝和删除方面的

操作更容易和直观。

作为真正的软件狂人，我们能为一个比萨饼的奖金开始竞赛，一直以求成为第一个能够编程再造如此行为的人——即，怎样以动画方式拷贝文件。花了几个小时的时间才在一大堆新函数中找出了 SHFileOperation() 函数，这是一个响应动画拷贝的 API 函数，它也是探测器执行所有文件操作的函数。

竞赛的规则之一是建立一个具有这个唯一目标功能的演示程序。在这个函数出现之后，这个问题实际上是十分简单的。事实上，当我确定在程序中使用这个函数作为标准函数来进行文件操作时，问题就出现了。要这样做，你就必须彻底弄清楚这个函数的原型和它的能力，实际有趣的故事从这里就开始了。

在这一章中，我打算向你展示 SHFileOperation() 的内部奥秘。

怎样正确地使用函数所支持的标志和命令

怎样正确使用源/目标缓冲区

最有可能的返回码是什么

对于长文件名，可能遇到的问题

关于文件名映射，以前未暴露的问题

与这本书的其它任何地方一样，在这一章中，你将发现一些有帮助的函数，它们推动你使用 Windows 的通用控件，对话框。

SHFileOperation() 能做些什么

要得到这个问题的答案，先让我们先来看一下在文件 shellapi.h 中 SHFileOperation() 函数的声明：

```
int WINAPI SHFileOperation(LPSHFILEOPSTRUCT lpFileOp);
```

进一步，看一看 SHFILEOPSTRUCT 结构，这也是一个在 shellapi.h 中定义的结构。

```
typedef struct _SHFILEOPSTRUCT
{
    HWND hwnd;

    UINT wFunc;

    LPCSTR pFrom;

    LPCSTR pTo;

    FILEOP_FLAGS fFlags;

    BOOL fAnyOperationsAborted;

    LPVOID hNameMappings;

    LPCSTR lpszProgressTitle;
} SHFILEOPSTRUCT, FAR* LPSHFILEOPSTRUCT;
```

通过这个结构，SHFileOperation() 函数可以做任何想要的操作。简要地说，这个函数可以做：

把一个或多个文件从源路径拷贝到目标路径

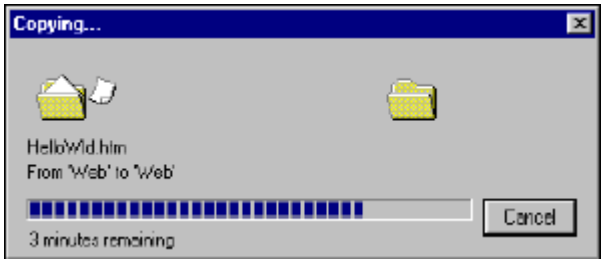
删除一个或多个文件，把它们发送到‘回收站’

重命名文件

把一个或多个文件从源路径移动到目标路径

到目前为止，我们没有看到任何新东西——至少没有特别刺激的东西。事实上，Win32 API(和 C 运行库)已经提供了做同样事情的方法。特别是 Win32 API 提供了 CopyFile(), DeleteFile(), 和 MoveFile() 来执行这些任务。

然而，强大的 SHFileOperation() 函数的出现，使你能够仅仅使用一个命令就可以处理对缺省目录的多重拷贝和建立。他还支持‘Undo’操作，以及在目标名冲突的情况下自动重命名操作。最后，他还大方地提供了一个空白纸页一个从文件夹漂动到另一个文件夹显示的动画。



毋庸置疑，你可以从 Win32 的底层 APIs 获得同样的功能，但是这可能需要做大量的工作。

SHFileOperation() 函数怎样工作

与所有仅使用数据结构作为输入参数的函数一样，SHFileOperation() 函数是一个相当灵活的例程。通过以适当的方式组合各种标志，和使用 (或不使用) 各个 SHFILEOPSTRUCT 结构的成员，它可以执行许多操作。下面就让我们来看一看这个结构中每一个成员所起的作用：

名	描述
Hwnd	由这个函数生成的所有对话框的父窗口 Handle。
wFunc	表示要执行的操作
pFrom	含有源文件名的缓冲
pTo	含有目标文件名的缓冲(不考虑删除的情况)
fFlags	能够影响操作的标志
fAnyOperationsAborted	包含 TRUE 或 FALSE 的返回值。它依赖于是否在操作完成之前用户取消了操作。通过检测这个成员，你就可以确定操作是正常完成了还是被手动中断了。
hNameMappings	资料描述它为包含 SHNAMEMAPPING 结构数组的文件名映射对象的 Handle。
lpszProgressTitle	一个在一定情况下用于显示对话框标题的字符串。

简言之，有四个成员确实需要进一步研究，它们是：

wFunc(间接地包括 pFrom 和 pTo)
fFlags
hNameMappings
lpszProgressTitle

可用的操作

wFunc 成员指定了在给定文件上操作，这些文件由 pFrom 和 pTo 给出。wFunc 的可能取值 (在 shellapi.h 定义) 是：

代码	值	描述
FO_MOVE	0x0001	所有在 pFrom 中指定的文件都被移动到 pTo 指定的位置，pTo 必须是一个目录名。

FO_COPY	0x0002	所有在 pFrom 中指定的文件都被拷贝到 pTo 指定的位置，其内容可以是目录名或甚至是一个与 pFrom 1:1 对应的文件集。
FO_DELETE	0x0003	所有在 pFrom 中指定的文件都被发送到‘回收站’，pTo 被忽略。
FO_RENAME	0x0004	所有在 pFrom 中指定的文件都重新命名为 pTo 中指定的名字，在 pFrom 和 pTo 之间，名字不需 1:1 对应。

pFrom 和 pTo 都是包含一个或多个文件名的缓冲。如果包含了多于一个的文件名，则各个文件名之间就需要用 NULL (字符"0") 进行分隔，并且整个串需要用两个 NULL ("0"0) 字符结束，无论有多少文件名。

如果 pFrom 和 pTo 不包含目录信息 (即，它们不是全路径名)，则，函数假设它应该使用由 GetCurrentDirectory() 函数返回的驱动器和目录。pFrom 可以包含通配符，也可以是“*. *”这样的字符串。

设置 SHFILEOPSTRUCT 结构的 fFlags 成员标志能够影响所有这些操作。在线资料中按字符顺序列出了所有标志。在我们的简短讨论中，将采取稍微不同的方法，将标志根据它能影响的实际操作分组，如果你想要自然排列的表，请引用在线资料。

注意两个空的结尾符("0"0)

其实，就 pFrom 和 pTo 是指向一个字符串列表的指针而不是通常意义的缓冲这样一个事实而言，资料的说明并不充分。也就是说，SHFileOperation() 总是期望传送来的串由两个 NULL 字符终止，即使你传送的只有单个文件名或使用通配符的单个串也是如此。如果不使用两个 NULL 字符来终止 pFrom 和 pTo 中的字符串，则可能的情况就是函数在分析传来的内容时失败。此时，它返回一个‘不能拷贝/移动文件’错 (错误码 1026)。没有两个 NULL 字符，函数可能会把字符串尾，单个 NULL 字符后的字节作为被拷贝或移动的文件名。这些字节可以是任何东西，可能不是合法的文件名，因此错误就出现了。由于 pFrom 总是被解释为文件名列表，而 pTo 只有在 FOF_MULTIDESTFILES 标志下才被解释为文件名列表，所以这个错误常常伴随 pFrom 一同出现。在所有其它情况，SHFileOperation() 都假设 pTo 引用单个文件名。因此单个 NULL 字符终止是充分的一两个 NULL 终止仅仅在终止包含多个文件名的列表时被要求。除非明确说明有多个目标文件，对 pTo 内容的解析停止于头一个 NULL 终止符。

解析方法依赖于指针是否引用了字符串列表或简单缓冲，为安全起见，你应该总附加一个终止符到你打算赋值给 pFrom 的字符串后面，同样，对 pTo，如果有多个目的文件的话，也是如此。字面上，你可以显式加一个"0在串的结尾 (当然，字符串自动终止在单个 NULL 字符上)：

```
shfo.pFrom = "c:\"demo\"one.txt"0c:"demo\"two.txt"0";
```

如果使用变量，可以采用下面的方法：

```
pszFrom[lstrlen(pszFrom) + 1] = 0;
```

移动和拷贝文件

要把文件从一个位置移动或拷贝到另一个位置，需要指定：

包含源文件名的缓冲。可以是一个名字序列，单个名字，一个包含通配符的串，甚至可以是含通配符的串序列。

一个目的目录。如果你移动一个确定的文件列表，还要准备一个目标名列表，注意保证 1:1 的与源名对应。换句话说，每一个源文件名都必须有一个目标文件名以便移动或拷贝。如果有多个目标文件，就必须在 fFlags 中指定 FOF_MULTIDESTFILES 标志。

这个标志可以影响的操作是：

标志	值	描述
FOF_MULTIDESTFILES	0x0001	pTo 成员包含多个与源文件对应的目标文件。
FOF_SILENT	0x0004	发生的操作不需要返回到用户，就是说，不显示进度条对话框，而其它相关的消息框仍然显示。
FOF_RENAMEONCOLLISION	0x0008	如果目标位置已经包含了与打算移动或拷贝的文件重名的文件，这个标志指示要自动地改变目标文件。
FOF_NOCONFIRMATION	0x0010	这个标志使函数对任何消息框的回答总是 Yes，只有一个例外，就是当询问是否建立缺省目录的对话框显示时。此时，需要 FOF_NOCONFIRMMKDIR 标志帮忙。(参考后面的说明)。
FOF_FILESONLY	0x0080	这个标志仅仅应用于指定了包含子目录和通配符(*.*)的情况。设置了这个标志，函数仅仅处理文件而不进入到子目录。
FOF_SIMPLEPROGRESS	0x0100	这个标志产生一个简化的用户界面：有一个动画窗口，但是不显示文件名，而是显示通过 lpzProgressTitle 成员指定的文字。
FOF_NOCONFIRMMKDIR	0x0200	如果目标目录不存在，这个标志使函数默默地建立一个缺省目录。没有这个标志，函数将提示是否建立一个完整的目的路径。这个标志与下一个将要介绍的标志有点微妙的关系。
FOF_NOERRORUI	0x0400	如果设置了这个标志，发生的任何错误都不会引起消息框的显示，全部都返回错误码。这个标志与上一个标志关系有点微妙。
FOF_NOCOPYSECURITYATTRIBS	0x0800	应用于 WindowsNT，Shell4.71(WindowsNT 具有 IE4.0 和活动桌面)，和更高版本。这个标志防止对具有安全属性的文件进行拷贝。

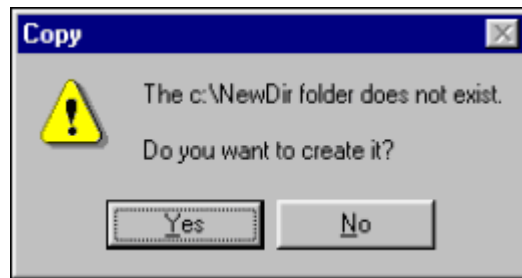
现在让我们更详细地了解一下这些选择，在移动或拷贝文件的时候，所关心的有两个主要方面：正确地

标识要传送的文件，并确保所设置的标志产生所希望的行为。

避免不想要的对话框

如果你希望操作默默地进行，不需要显示对话框或系统错误消息，你可能认为 FOF_NOERRORUI | FOF_SILENT 标志的组合是一个好的选择。然而，这并不是真的，正象我所提到的，使用 FOF_NOERRORUI 仅仅能隐藏错误引发的消息框。另一方面，FOF_SILENT 标志自己不能防止这个函数显示所有可能的消息框。事实上，FOF_SILENT 仅仅影响到进度条对话框——即，显示被拷贝或移动的文件名，伴随一个通常的动画对话框。如果函数发现给定的文件或目录在目标位置已经存在，它将总是显示提示。要避免这个行为，你就需要把 FOF_NOCONFIRMATION 设置加到标志中。这将使函数在每一步都采用一个不可见的 Yes 点击行为。然而这个故事远没有结束。

如果目标路径包含了缺省目录，所有这些标志都无效。在继续执行文件的拷贝或移动之前，这个函数试图保证目标目录的存在，你可能已经合理地指定了一个不存在的目录，这个函数将小心地建立它，但是，它首先要求一个显式的认可。



要跳过这个对话框，需要设置标志 FOF_NOCONFIRMMKDIR。如果设置了这个位，函数就自动建立任何缺省的目录而不显示提示框。

概括地说，如果想完成拷贝（或移动）操作而不需要用户的干涉，你可以使用如下的标志组合设置 SHFILEOPSTRUCT 结构的 fFlags 成员：

```
FOF_SILENT
FOF_NOCONFIRMATION
FOF_NOERRORUI
FOF_NOCONFIRMMKDIR
```

然而，关于同时使用 FOF_NOERRORUI 和 FOF_NOCONFIRMMKDIR 标志组合，仍然有一点是需要澄清的。

缺省目录

有趣的是，一个缺省目录可以看作是一个由系统对话框弹出的系统错。尽管你可以通过设置 FOF_NOCONFIRMMKDIR 标志跳过这个对话框，但是 FOF_NOERRORUI 标志优先于 FOF_NOCONFIRMMKDIR，有效地抑制了对话框，使后面所涉及到的标志不被选择。如果这两个标志都被指定，你既不能被提示授权建立不存在的目录，也不能自动建立目录，相反，这个函数继续执行就象拒绝建立目录一样，并将返回：

错误码117

取消标志 fAnyOperationsAborted 设置到 True

不产生文件的移动或拷贝

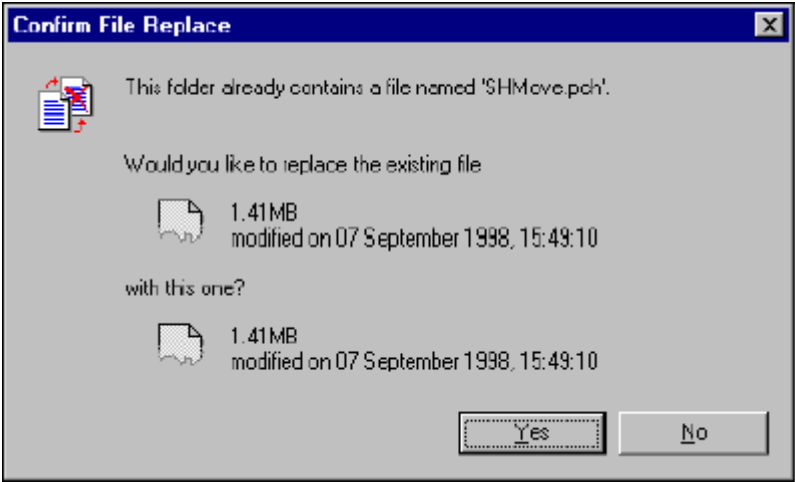
这是否是说，要避免使用 FOF_NOERRORUI 标志呢？当然，如果你想要绝对静默的操作，就不可避免地要使用它——以防止所有错误消息框显示。问题是它也阻止了新目录默认地建立，并且产生一个无谓而又麻烦的错误。幸运地是，有一种方法能够绕过它，即，在使用这个标志调用 SHFileOperation() 前，确保 pTo 中存储的是已存在的全路径名。Win32提供了一个实现这个目的的函数：

BOOL MakeSureDirectoryPathExists(LPCSTR DirPath);

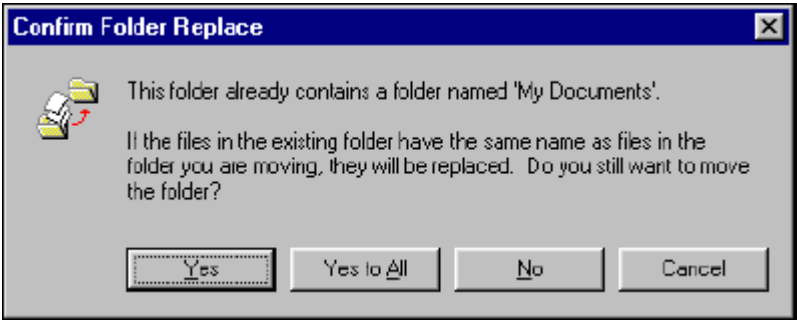
使用这个函数需要#include imagehlp.h 文件，和连接 imagehlp.lib 库。

文件重命名

SHFileOperation() 函数在置换已存在文件时能够引起的问题之一是：



或类似地，它引起的已存在目录的问题：



通过设置 FOF_NOCONFIRMATION，可以隐含地允许函数置换老对象，但是第二种可能出现了。你知道，如果在 Windows 探测器中选择文件，并按 Ctrl-C 键，然后按 Ctrl-V 键，在同一个文件夹下将出现一个新文件，这个文件具有同拷贝 xxxx 相似的文件名，此处 xxxx 就是你选择的文件。探测器自动重命名了这个新文件以避免冲突。只要设置了 FOF_RENAMEONCOLLISION 标志，SHFileOperation() 函数也能提供这个功能。FOF_RENAMEONCOLLISION 和 FOF_NOCONFIRMATION 标志组合禁止了置换操作时的确认对话框。然而接下来，你的文件或目录将不可避免地覆盖。如果不合理的情况下指定这两个标志，则 FOF_RENAMEONCOLLISION 标志优先

标志间的关系

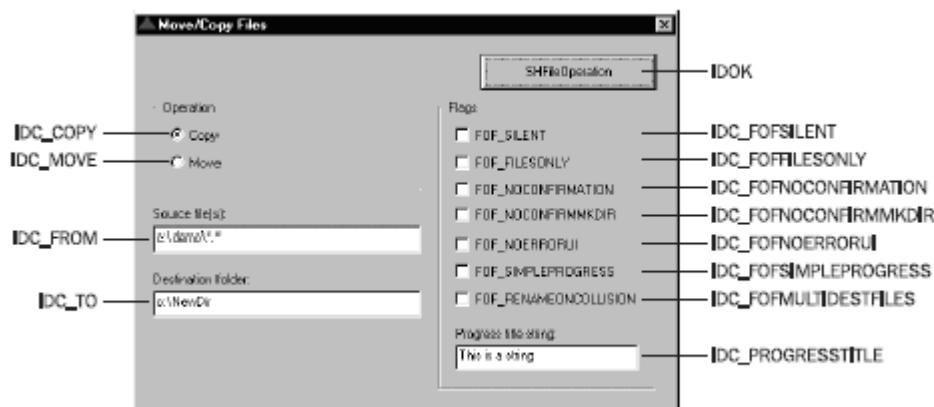
到目前为止，在你的脑海中应该有两个问题，一是各个标志之间究竟是什么样的关系，其次是哪些标志影响哪类对话框。下表给出了问题的答案。

标志	抑制的对话框	相关性与优先级
FOF_MULTIDESTFILES	None	None
FOF_FILESONLY	None	None
FOF_SILENT	如果设置，进度对话框不显示。	优先于 FOF_SIMPLEPROGRESS

		标志。
FOF_SIMPLEPROGRESS	None	为 FOF_SILENT 标志所抑制。
FOF_RENAMEONCOLLISION	如果设置了这个标志,当被移动或拷贝的文件与已存在文件同名时替换对话框不会出现。	名字冲突时, 如果 FOF_NOCONFIRMATION 标志设置, 则操作继续。 如果二者都设置了, 则它优先于 FOF_NOCONFIRMATION。即, 文件以给定的新名字复制, 而不是覆盖。
FOF_NOCONFIRMATION	如果设置, 确认对话框在任何情况下都不出现。	名字冲突时, 引起文件覆盖, 除非设置了 FOF_RENAMEONCOLLISION 标志。
FOF_NOCONFIRMMKDIR	抑制请求建立新文件夹的对话框	缺省目录作为严重错误产生一个错误消息框。 建立目录的确认对话框作为错误消息框是否显示依赖于 FOF_NOERRORUI 的设置。
FOF_NOERRORUI	抑制所有错误消息框。	优先于前一个标志。如果设置, 则, 缺省目录引起不被处理的异常, 并且返回错误码。

一个例程

为了有助于理解 SHFileOperation() 函数的特性, 我们给出了一个简单的综合例子程序, 称之为 SHMove。使用 VC++ 建立基于对话框的应用, 下面是需要建立的用户界面:



你可以在 OnInitDialog() 函数中看到默认的设置。这个函数在 SHMove.cpp 中声明。

```
void OnInitDialog(HWND hDlg)
{
    // Set the icons (T/F as to Large/Small icon)
    SendMessage(hDlg, WM_SETICON, FALSE, reinterpret_cast<LPARAM>(g_hIconSmall));
    SendMessage(hDlg, WM_SETICON, TRUE, reinterpret_cast<LPARAM>(g_hIconLarge));
}
```

```

// Initialize the 'to' and 'from' edit fields
SetDlgItemText(hDlg, IDC_TO, "c:\"NewDir");
SetDlgItemText(hDlg, IDC_FROM, "c:\"demo\"*.");
// Take care of the 'progress' title
SetDlgItemText(hDlg, IDC_PROGRESSTITLE, "This is a string");
// Select the default operation
CheckRadioButton(hDlg, IDC_COPY, IDC_MOVE, IDC_COPY);
}

```

要使这个对话框引起对 SHFileOperation() 的调用，需要实现点击 **SHFileOperation** 按钮的 OnOK() 函数的功能。pTo 和 pFrom 成员的内容以及相关的 FOF_ 标志在这个函数中设置。

```

void OnOK(HWND hDlg)
{
    SHFILEOPSTRUCT shfo;
    WORD wFunc;
    TCHAR pszTo[1024] = {0};
    TCHAR pszFrom[1024] = {0};
    TCHAR pszTitle[MAX_PATH] = {0};

    //设置要执行的操作
    if(IsDlgButtonChecked(hDlg, IDC_COPY))
        wFunc = FO_COPY;
    else
        wFunc = FO_MOVE;

    //取得进度条文字串
    GetDlgItemText(hDlg, IDC_PROGRESSTITLE, pszTitle, MAX_PATH);

    //取得 from 缓冲
    GetDlgItemText(hDlg, IDC_FROM, pszFrom, MAX_PATH);
    pszFrom[lstrlen(pszFrom) + 1] = 0;

    //取得 To 缓冲
    GetDlgItemText(hDlg, IDC_TO, pszTo, MAX_PATH);

    //取得标志
    WORD wFlags = 0;
    if(IsDlgButtonChecked(hDlg, IDC_FOFSILENT))
        wFlags |= FOF_SILENT;
    if(IsDlgButtonChecked(hDlg, IDC_FOFNOERRORUI))
        wFlags |= FOF_NOERRORUI;
    if(IsDlgButtonChecked(hDlg, IDC_FOFNOCONFIRMATION))
        wFlags |= FOF_NOCONFIRMATION;
    if(IsDlgButtonChecked(hDlg, IDC_FOFNOCONFIRMMKDIR))
        wFlags |= FOF_NOCONFIRMMKDIR;
    if(IsDlgButtonChecked(hDlg, IDC_FOFSIMPLEPROGRESS))
        wFlags |= FOF_SIMPLEPROGRESS;
    if(IsDlgButtonChecked(hDlg, IDC_FOFRENAMEONCOLLISION))
        wFlags |= FOF_RENAMEONCOLLISION;
}

```



```

        if(IsDlgButtonChecked(hDlg, IDC_FOFFFILESONLY))
            wFlags |= FOF_FILESONLY;
        //调用 SHFileOperation() 函数
        ZeroMemory(&shfo, sizeof(SHFILEOPSTRUCT));
        shfo.hwnd = hDlg;
        shfo.wFunc = wFunc;
        shfo.lpszProgressTitle = pszTitle;
        shfo.fFlags = static_cast<FILEOP_FLAGS>(wFlags);
        shfo.pTo = pszTo;
        shfo.pFrom = pszFrom;
        int iRC = SHFileOperation(&shfo);
        if(shfo.fAnyOperationsAborted)
        {
            Msg("Aborted!");
            return;
        }
        //显示操作结果
        SPB_SystemMessage(iRC);
    }

```

这个函数从对话框的控件中取得了所有它所需要的数据，然后填入 SHFILEOPSTRUCT 结构中。如果操作失败，fAnyOperationsAborted 成员被填入 TRUE。在上面的代码中有两个陌生的函数 Msg() 和 SPB_SystemMessage()，这两个函数其实就是 MessageBox() 的包装变种，你可以自己写一个这样的变种函数来跟踪 SHFileOperation() 函数实际返回的信息。现在我们集中精力于源/目缓冲，把 #include resource.h 加到 SHMove.cpp 中，并且建立一个工程(project)。

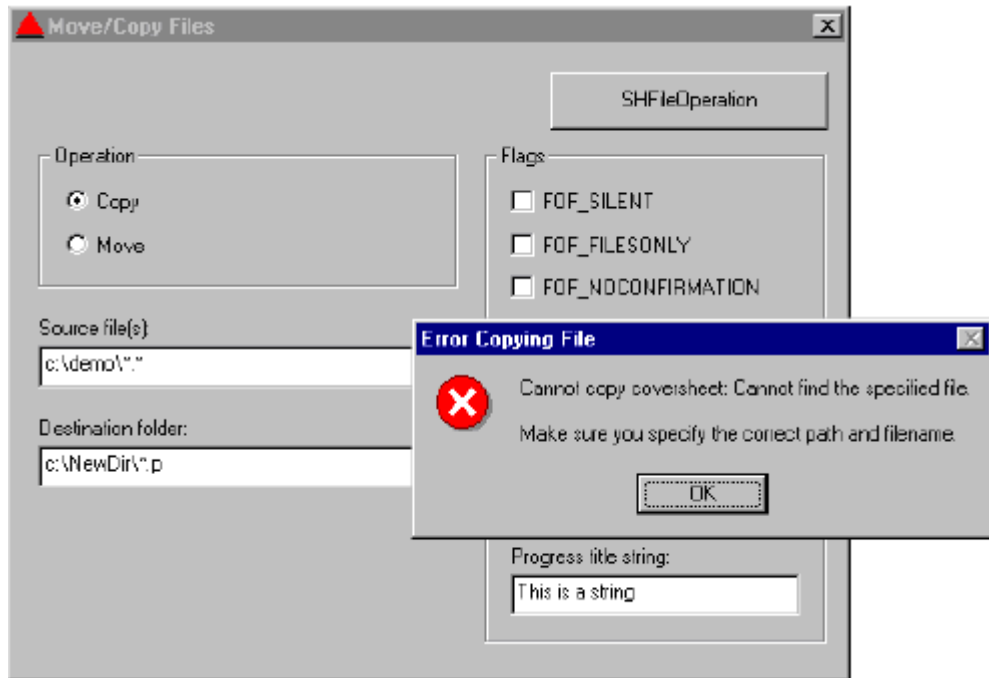
源与目

在把文件从源位置移动或拷贝到目位置时，有下列几种可能：

- 一组文件到单一文件夹
- 众多单个文件到单一文件夹
- 单一文件到单一文件夹
- 众多单一文件到众多单一文件夹

上述的‘单一文件’意思是说一个全路径文件——即，一个具有完整名的文件。对应的‘组文件’则是包含通过通配符标识的文件，这些文件是不知名的文件。仅仅在上述的第四种情况，才需要使用 FOF_MULTIDESTFILES 标志。

上述代码在默认情况时给 pFrom 赋予带有通配符的串，例如：c:"demo"*. *，在这种情况下，你必须指定一个目的文件夹。通过 pTo 缓冲传递的任何东西都被作为文件夹名，除非其中包含了不合法的字符。如此，将得到错误(在第一个文件拷贝或移动时)，就如下面显示的那样。



前面解释过，可以通过传递两个 NULL 终止的多重文件名串(每一项由单个 NULL 分隔)来操作多重文件，例如，可以把如下编码写到 OnOK() 中：

```
shfo.pFrom = "c:\""demo\""one.txt"0c:\""two.txt"0c:\""three.txt"0";
shfo.pTo = "c:\""NewDir\"";
```

这里我们努力想要一次拷贝/移动三个文件：one.txt, two.txt, 和 three.txt。所有这三个文件都将被拷贝到根 C: 下的目录 NewDir 中。第一个源文件的位置在 c:"demo 目录下，其他两个在 c: 下。

如果 pFrom 缓冲中正好仅包含一个名字，则 SHFileOperation() 函数有两种方法处理 pTo 的内容：

```
shfo.pFrom = "c:\""demo\""one.txt"0";
shfo.pTo = "c:\""NewDir\"";
```

如果目录或文件 c:"NewDir 已经存在，则它会被适当地处理，即，文件 c:"demo"one.txt 或者拷贝到目录，或者置换已经存在的文件。反之，如果 c:"NewDir 不存在，则它就会被当作新文件名，而不再被当作文件夹名。

如果想要拷贝单一文件到新文件夹，则可以考虑在 pTo 的内容后面加一个反斜线 "来进行操作。

```
shfo.pFrom = "c:\""demo\""one.txt"0";
shfo.pTo = "c:\""NewDir\"";
```

奇怪的是这将导致建立缺省文件夹。并且使文件的拷贝或移动失败。如果重试，则它可以象所期望的那样工作，因为，在第二次运行时，这个文件夹已经存在了。所以，在拷贝单个文件到不存在的文件夹时需要做些什么工作？唯一总能正常工作的是把一个*字符加到文件名的末尾。这样做是糊弄函数，使它认为它是在操作一个通配符表达式。

```
shfo.pFrom = "c:\""demo\""one.txt*"0";
shfo.pTo = "c:\""NewDir\"";
```

另一个可能的情况是你想要拷贝多重单个文件到同样数目的单个文件上。这必须满足两个要求，首先应该设置 FOF_MULTIDESTFILES 标志，其次，一定要保证每一个源文件都有一个目的文件—需要完备的1:1对应。原文件列表中第 n 个文件被拷贝或移动到目的文件列表中的第 n 个文件。

```
shfo.fFlags |= FOF_MULTIDESTFILES;
shfo.pFrom = "c:\""one.txt"0c:\""two.txt"0";
```

```
shfo.pTo = "c:\""New one.txt"0c:\""New two.txt"0";
```

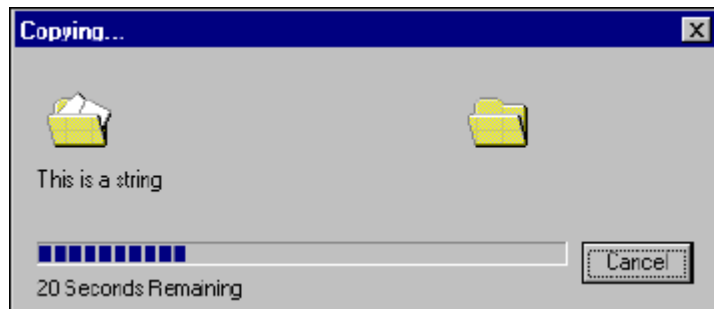
如果哪个方面没有满足，哪个方面就失败。例如执行下面的代码：

```
shfo.fFlags |= FOF_MULTIDESTFILES;
shfo.pFrom = "c:\""one.txt"0c:\""two.txt"0c:\""three.txt"0";
shfo.pTo = "c:\""New one.txt"0c:\""New two.txt"0";
```

目标文件列表的第一项(即 c:\""New one.txt")被作为所有源文件要去的文件夹名。实际上，这个操作被处理成多对一的操作了。

在使用通配符时，源缓冲可以隐含地包括文件和目录。如果想要函数仅处理文件，加一个 FOF_FILESONLY 标志就可以了。如果想要拷贝整个目录，就需要加"*.*"到其路径末尾。

除非你指定了 FOF_SILENT 标志，否则 SHFileOperation() 函数总是显示带有动画和进度条的进度对话框，其中的标签显示正在拷贝或移动的文件。通过设置 FOF_SIMPLEPROGRESS 标志，你可以隐藏这些标签，用在 lpszProgressTitle 成员中给出的文字替换他们。这有助于隐藏被拷贝或移动的文件名。



删除文件

文件删除是一个简单的操作，它仅仅影响到输入缓冲 pFrom—pTo 缓冲被忽略。与前面一样，操作的详细情况依赖于标志的设置。相关的标志是：

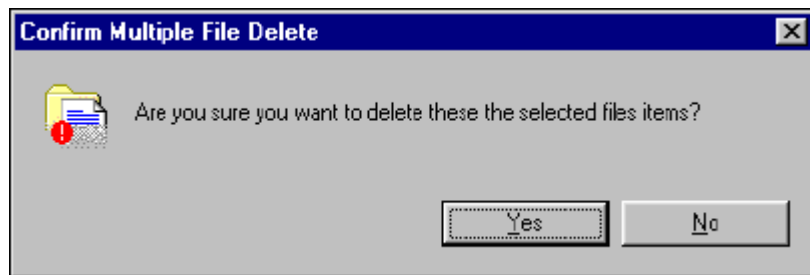
标志	值	描述
FOF_SILENT	0x0004	这个操作不回馈给用户，就是说，不显示进度对话框。相关的消息框仍然显示。
FOF_NOCONFIRMATION	0x0010	这个标志使函数对任何遇到的消息框都自动回答 Yes。
FOF_ALLOWUNDO	0x0040	如果设置，这个标志强迫函数移动被删除的文件到‘回收站’中。否则，文件将被物理地从磁盘上删除。
FOF_FILESONLY	0x0080	设置这个标志导致函数仅仅删除文件，跳过目录项。它仅仅应用于指定通配符的情况。
FOF_SIMPLEPROGRESS	0x0100	这导致简化用户界面。使之只有动画而不报告被删除的文件名。代之的是显示 lpszProgressTitle 成员中指定的文字。
FOF_NOERRORUI	0x0400	如果设置了这个标志，任何发生的错误都不能使消息框显示，而是程序中返回错误码。

这里出现的标志最要紧的是 FOF_ALLOWUNDO，它允许程序员决定文件是否一次就全部删除，或存储到‘回收站’中等候可能的恢复。如果 FOF_ALLOWUNDO 被设置，文件则被移动到回收站，并且这个操作可以被 Undo (尽管可以手动 Undo)。涉及到‘回收站’的 API 函数在第十章中讲述。Undo 特征仅在删除下可用——在拷贝与移动中没有等价的操作。

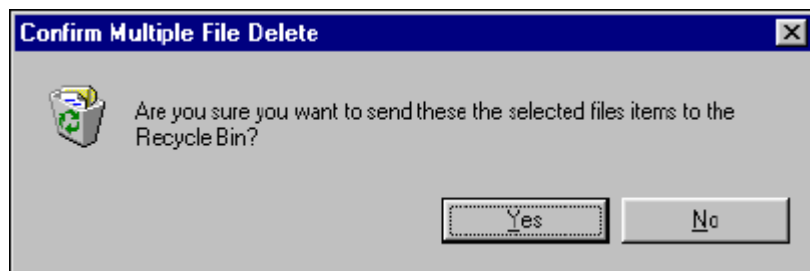
说明 FOF_ALLOWUNDO 标志，影响到前面程序中的用户界面。修改我们的简单工程来接受一个删除请求也不是很困难。但是为了简练，我们还是直接把代码写进 OnOK () 函数：

```
ZeroMemory(&shfo, sizeof(SHFILEOPSTRUCT));  
shfo.hwnd = hDlg;  
shfo.wFunc = FO_DELETE;  
shfo.lpszProgressTitle = pszTitle;  
shfo.fFlags = FOF_NOERRORUI;  
shfo.pFrom = "c:\\demo\\*.*";
```

上面代码企图删除整个 c:\\demo 目录的内容，并且导出对话框：

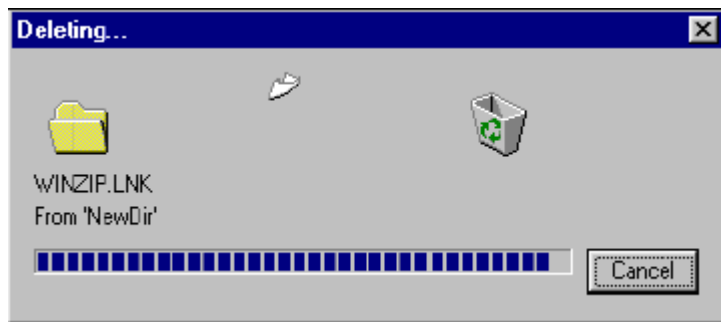


就象看到的那样，由于没有指定 FOF_ALLOWUNDO 标志，消息框中没有提到‘回收站’。通过设置 FOF_ALLOWUNDO 标志，文件将改为直接发送到回收站：



上表列出的其他标志与拷贝或移动操作所作的完全相同。因此，可以通过设置 FOF_SIMPLEPROGRESS 或 FOF_SILENT 隐藏正在被删除的文件名，通过设置 FOF_FILESONLY，仅仅删除文件。注意 FOF_FILESONLY 标志不能进入子目录。上面显示的对话框也没有提示有多少文件要被删除。然而这是好理解的，因为发起计算的命令中包含了通配符 (否则将显示文件数)，所以函数不能得出文件数。这也可能就是为什么没有文件被删除时它返回成功的原因吧。

按惯例，操作系统在文件被删除前将请求确认。如果你发现这样的对话框是无用的，你就可以通过对所有询问回答 Yes 来自动地隐藏它，这只需设置 FOF_NOCONFIRMATION 标志即可。典型地，一个 FO_DELETE 操作如下图所示：



文件重命名

在这一节中第一个要注意的事情就是不能用通配符来使 SHFileOperation() 函数重命名文件。通过指定单个源文件名到 pFrom 和单个目标文件名到 pTo 来改变文件名似乎是唯一的方法：

```
ZeroMemory(&shfo, sizeof(SHFILEOPSTRUCT));
shfo.wFunc = FO_RENAME;
shfo.pFrom = "c:demoone.txt";
shfo.pTo = "c:demoone.xxx";
```

显然，有两件事情在重命名文件操作中不允许做是有道理的，明确地说，它们是：

改变目标目录。重命名只是改变名字，不是文件夹。

覆盖已存在的文件

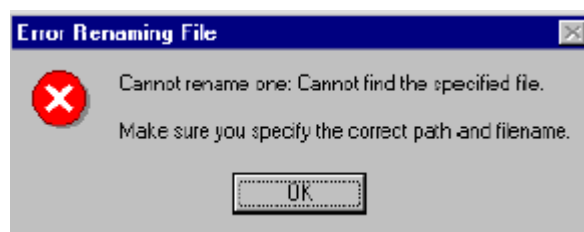
如果努力执行这样的操作，则自然只能获得错误。搜索所有的错误代码我们发现，试图传递下面的参数到函数：

```
shfo.pFrom = "c:demo*.txt";
shfo.pTo = "c:newdir";
```

显然是不对的，并且函数适时地返回下面的错误消息：



尽管命令荒唐地返回成功(值是0)，这个消息是足够清楚的了。然而，这个消息隐含地说明用 MS-DOS 所用的语法在这里也能正常工作。换句话说，我们应该能够重命名，如 *.txt 到 *.xtt。在 MD-DOS 下这些是没问题的，在 SHFileOperation() 下，它不行。如果你测试，将得到如下消息：



这个消息由下面的两行代码引起：

```
shfo.pFrom = "c:demo*.txt";
shfo.pTo = "c:demo*.xtt";
```

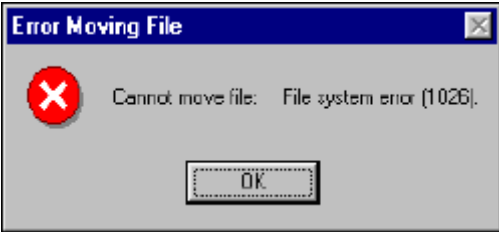
就这个例子而言，c:demo 目录下含有两个文件 one.txt 和 two.txt。One 在这种情况下实际是所包括的文件之一，没有扩展名。所以这个消息之后的返回码是2—‘文件没找到’。

由于 FO_RENAME 命令似乎仅在单个文件时成功，因此影响用户对话框界面的标志就失去了重要性——加速操作简单地使用户界面不可见。但是下述标志产生作用仍然可以感觉到：

标志	值	描述
FOF_RENAMEONCOLLISION	0x0008	如果目标位置已经包含了一个与要被重命名文件有相同名字的文件，这个标志指示函数自动改变目标名来拷贝 Xxx，此处的 Xxx 是没有扩展的初始文件名。如果没有设置这个标志，仍然不会有提示，但是，你将得到错误消息。
FOF_NOERRORUI	0x0400	如果设置了这个标志，所发生的所有错误都不引起消息框的显示，而是返回错误码。

SHFileOperation() 函数的返回值

在线资料中说明，SHFileOperation() 在成功时返回0，失败时返回非0值。显然这是真的，但并不是最有用的解释。重复测试这个函数，可以确信它有非常多的终止方式。事实上，我们经常在系统错误的提示中运行，在有些地方这个函数只是简单地返回从更靠近文件系统的其它程序中获得的返回码。这里的列表给出了 SHFileOperation() 返回的最通常的错误（可以肯定不是最详尽的）。

错误码	描述
2	就象上面提到的，如果你试图重命名多重文件，这个消息就会出现。描述是相当直接的一系统不能找到指定的文件—但是并不能理解它为什么不能找到文件。
7	在询问是否想要置换给定文件时，你回答了‘取消’，函数就返回这个错误码。它的描述也是相当的不明确—存储控制块被销毁。
115	在试图重命名文件到不同的文件夹时，发生这个文件系统错。重命名文件只是改变文件名，而不能改变文件夹。
117	一个 IOCTL 错(输入/输出控制)，在目的路径中有错误时或取消了新目录的建立时，这个错误发生了。
123	你正在试图重命名一个文件，然而你给出的名字是一个已经存在的文件。它也有一个无用的描述：文件名，目录名，或卷标号的语法是不正确的。
1026	在试图移动或拷贝一个不存在的文件时，出现这个文件系统错。一般地，它提示了，源缓冲中的某些东西应该修改一下。这个错误码引出下面的错误框—你可以通过设置 FOF_NOERRORUI 标志抑制它的显示。 

在很多情况下都返回错误码117，所有这些都与目标目录的问题有关。例如，如果你取消了要求建立目录的请求，函数就返回这个码（不显示系统消息框）。如果在指定的目录名中有明显的错误，错误框就被显示，你可以使用 FOF_NOERRORUI 标志来抑制它。

两个关于错误信息显示的简单例程

错误消息为绝大多数程序员所诅咒。因此，或者写出除文字描述以外的大量代码，或者通过其他方法绕过错误消息。框架环境如 MFC 提供了一些工具，然而，你一定不想要只是为了开发这样的特征把代码移植到 MFC 中吧。为此，我们生成了包含两个实用函数的文件，在附录 A 中。头一个是经过修订的 MessageBox()。它通过增加常用的 printf() 的格式化能力来扩展功能，改名为 Msg()，代码如下：

```
#include <stdarg.h>

void WINAPI Msg(char* szFormat, ...)
{
    va_list argptr;
    char szBuf[MAX_PATH];
    HWND hwndFocus = GetFocus();

    //初始化 va_函数
    va_start(argptr, szFormat);
    //格式化输出串
    wvsprintf(szBuf, szFormat, argptr);
    //读显示标题
    MessageBox(hwndFocus, szBuf, NULL, MB_ICONEXCLAMATION | MB_OK);
    //关闭 va_函数
    va_end(argptr);
    SetFocus(hwndFocus);
}
```

主要是这段代码使用了 va_函数，它包含在 stdarg.h 头文件中。变量列表经由 wvsprintf() 格式化，最终由普通的 MessageBox() 函数显示。现在我们可以象下面那样写代码了：

```
iRC = CallFunc(p1, p2);

Msg("The error code returned is: %d", iRC);
```

第二个实用函数是 SPB_SystemMessage() (SPB 前缀表示 **S**hell **P**rogramming **B**ook，用于区别你自己所写的函数)。它接受错误码，并传递到 FormatMessage()，一个 Win32 API 函数，对所有系统错误码 (至少是 winerror.h 里定义的错误码) 返回描述文字串的函数。FormatMessage() 提供的串与代码号聚在一起，并一同显示之：

```
void WINAPI SPB_SystemMessage(DWORD dwRC)
{
    LPVOID lpMsgBuf;
    DWORD rc;

    rc = FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER |
                     FORMAT_MESSAGE_FROM_SYSTEM |
                     FORMAT_MESSAGE_IGNORE_INSERTS,
                     NULL, dwRC,
                     MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
                     reinterpret_cast<LPTSTR>(&lpMsgBuf), 0, NULL);
```

```

        Msg("%s: %ld."n"n"s:"n"n"s", "This is the error code", dwRC,
            "This is the system's explanation", (rc == 0 ? "<unknown>" : lpMsgBuf));
        LocalFree(lpMsgBuf);
    }

```

函数适当地工作了吗

毋庸置疑, SHFileOperation() 函数就其返回码而言, 有一些问题。特别是, 即使在输入参数错, 要求的操作不能完成的情况下, 它也返回0 (成功标志):



测试下面拷贝或移动文件的代码:

```

shfo.pFrom = "c:demoone.txt"0;
shfo.pTo = "c:NewDir";

```

如果 one.txt 文件在初始文件夹中存在, 操作能正常进行。如果不存在, 错误1026出现。没说的, 这正是所期望的。然而, 如果你试一下下面的代码 (要保证没有匹配这个模式的文件), 会发生什么情况呢:

```

shfo.pFrom = "c:demo"x.*"0";
shfo.pTo = "c:NewDir";

```

这个函数仍然返回0, 即使没有文件被处理。相同的情况也出现在删除文件操作中。即使没有文件被删除, 返回码仍然表示成功。就信誉而言, 这是否算做一个 bug, 或是一个故意的行为。没有快捷的方法来验证操作是否获得了所希望的结果, 因此一定要记住函数返回之后一定要检查文件是否还存在。

长文件名

尽管 Windows Shell 的设计和实现是要带给用户最大的方便, 然而, 其中的某些函数对于长文件名似乎有点问题。确实, 这是对的一长文件名。下面就让我们看看什么是长文件名。

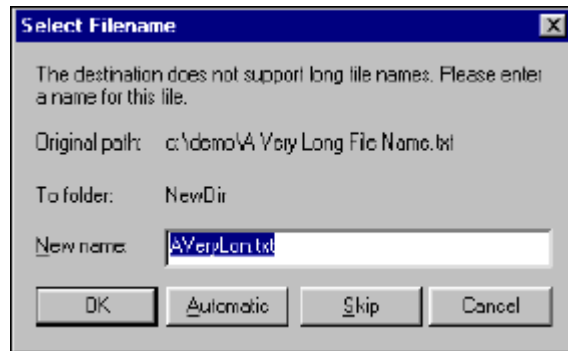
在所有上面看到的样例中, 我们为目标文件夹指定了全路径名 (通常使用 c:NewDir)。资料上说, 如果没有提供全路径名, 这个函数就假设使用由 API 函数 GetCurrentDirectory() 返回的当前目录。好, 现在就测试一下, 在函数 SHFileOperation() 中使用下面代码:

```

shfo.pFrom = "c:demo"*.*"0";
shfo.pTo = "NewDir";

```

我们想要拷贝或移动 c:demo 目录下的所有文件到一个称之为 NewDir 的新的或已经存在的目录, 该目录定位于当前目录下。如果在传输的文件中所提供的文件名**没有**长文件名的话, 所有操作都能顺利地执行。如果有任何长文件名, 下面这个对话框将出现:



这个函数所发生的操作是试图缩短长文件名以确保它能正确地存储到目标驱动器。当目标机器运行在 Windows 3.1 的情况下，在网络上移动文件，这样做是非常自然的。不幸的是我们是在运行 32 位操作系统的单个机器上拷贝或移动文件——这是适应长文件名的系统。如果不缩短文件名，函数 `SHFileOperation()` 就不工作。

奇怪的是，如果加一个驱动器到目标文件夹上，所有事情就又能工作了。还有一个不太陌生的情况，你将惊奇地发现，使用相对路径时，所有操作都是顺利的。奇怪，究竟发生了什么？

如果路径名开始字符是一个可用驱动器的逻辑标识符时，`SHFileOperation()` 函数在长文件名下能顺利工作。否则，它认为你正在连接远程驱动器，为此，支持长文件名的检查失败（如果没有 N: 驱动器，肯定失败）。例如，在我的机器中，直到 F 都能顺利工作，这是一个 CD-ROM 驱动器。

这可能是计算文件的目标驱动器所使用的代码中某个地方有错误而造成的，要想正常地操作，最简单的方式就是总使用全路径名。

文件名映射对象

在阅读 `SHFileOperation()` 的官方资料时，你可能已经注意到了关于**文件名映射对象**的谨慎描述。特别是，在对 `SHFILEOPSTRUCT` 结构的成员 `hNameMappings` 的表述时，资料中讲到了这个对象。`hNameMappings` 是一个指向内存块的指针——声明为 `LPVOID`，该内存块中包含一定数量的 `SHNAMEMAPPING` 结构。`SHNAMEMAPPING` 的数据结构定义如下：

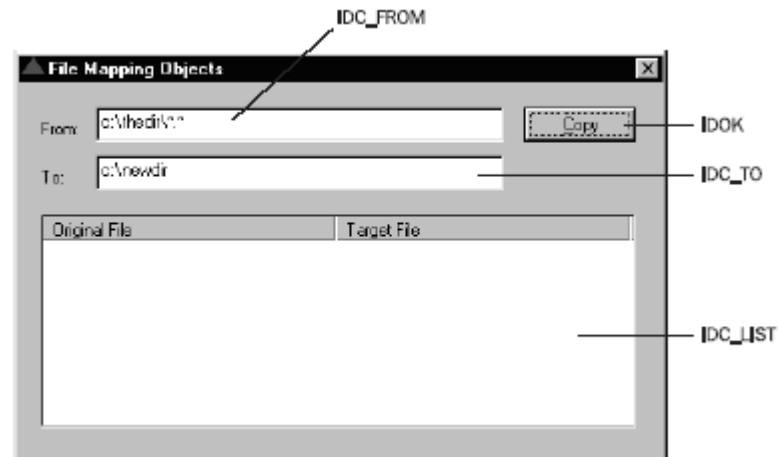
```
typedef struct _SHNAMEMAPPING
{
    LPSTR pszOldPath;
    LPSTR pszNewPath;
    int cchOldPath;
    int cchNewPath;
} SHNAMEMAPPING, FAR* LPSHNAMEMAPPING;
```

这个结构标识了被拷贝，移动甚至重命名的文件。更精确地说，它不仅存储了初始(全路径)文件名而且还有新的(全路径)文件名。因此，它暗示了一种可能性：在 `SHFileOperation()` 函数执行期间，你能够获得所发生情况的完整报告。可惜的是，事情并不象想象的那么简单。

首先，要使 `SHFileOperation()` 填写 `hNameMappings` 成员，你就必须指定一个附加的标志 `FOF_WANTMAPPINGHANDLE`。只这样做还不够，因为只有你也设置了 `FOF_RENAMEONCOLLISION` 标志，这个成员才被填写。进一步说，要使函数填写所有东西而不是 0，文件操作就要使用重命名来避免冲突。所有其它情况，`hNameMappings` 只简单地指向 `NULL`。

文件映射示例

建立一个基于对话框的应用称之为 `FileMap`，用以测试关于文件映射的一些东西。这里是用户界面：



要使用现实的值设置对话框，和初始化列表观察，你需要象如下方式调整 `OnInitDialog()` 函数 (记住附加 `#include resource.h` 语句)：

```
void OnInitDialog(HWND hDlg)
{
    HWND hwndList = GetDlgItem(hDlg, IDC_LIST);
    //设置报告观察
    LV_COLUMN lvc;
    ZeroMemory(&lvc, sizeof(LV_COLUMN));
    lvc.mask = LVCF_TEXT | LVCF_WIDTH;
    lvc.cx = 200;
    lvc.pszText = "Original File";
    ListView_InsertColumn(hwndList, 0, &lvc);
    lvc.pszText = "Target File";
    ListView_InsertColumn(hwndList, 1, &lvc);
    //初始化编辑区域
    SetDlgItemText(hDlg, IDC_FROM, "c:\\thedir\\*.*");
    SetDlgItemText(hDlg, IDC_TO, "c:\\newdir");
    //设置图标 (T/F 作为大/小图标)
    SendMessage(hDlg, WM_SETICON, FALSE, reinterpret_cast<LPARAM>(g_hIconSmall));
    SendMessage(hDlg, WM_SETICON, TRUE, reinterpret_cast<LPARAM>(g_hIconLarge));
}
```

现在可以编辑 `OnOK()` 函数了，附加的代码说明怎样取得和测试文件名映射对象的 `Handle`：

```
void OnOK(HWND hDlg)
{
    TCHAR pszFrom[1024] = {0};
    TCHAR pszTo[MAX_PATH] = {0};
    GetDlgItemText(hDlg, IDC_FROM, pszFrom, MAX_PATH);
    GetDlgItemText(hDlg, IDC_TO, pszTo, MAX_PATH);
    SHFILEOPSTRUCT shfo;
    ZeroMemory(&shfo, sizeof(SHFILEOPSTRUCT));
    shfo.hwnd = hDlg;
    shfo.wFunc = FO_COPY;
```

```

shfo.pFrom = pszFrom;
shfo.pTo = pszTo;
shfo.fFlags = FOF_NOCONFIRMMKDIR |
              FOF_RENAMEONCOLLISION |
              FOF_WANTMAPPINGHANDLE;
int iRC = SHFileOperation(&shfo);
if(iRC)
{
    SPB_SystemMessage(iRC);
    return;
}
//跟踪 Handle 值
Msg("hNameMappings is: %x", shfo.hNameMappings);
//象推荐那样释放对象
if(shfo.hNameMappings)
    SHFreeNameMappings(shfo.hNameMappings);
}

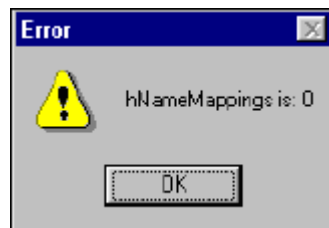
```

要特别注意代码的最后一行—释放文件名影射对象是你能在其上执行的最简单的操作。你必须调用 `SHFreeNameMapping()`，并且传递从 `SHFileOperation()` 中接受的 `Handle` 参数。每一步都能正常地执行，并且也能很好地理解。或许有一天，Windows 的资料也能如此清晰。

总之，运行这段代码后，你将发现 `hNameMappings` 总是 `NULL`，除非在执行的(拷贝，移动，重命名)操作中引起了名字冲突。如果发生重命名操作，这个 `Handle` 用于向你报告已经实际重命名文件的情况，以避免覆盖其他文件，报告给出了文件的新名和原名。

所以文件名影射对象与内存影射文件或其他进程内通讯的机理一样，没有做任何事情。它就是一个内存块，允许 `Shell` (和你)保持对已经重命名的文件踪迹的跟踪，以避免名字冲突。

如果目标目录(本例中为 `c:"newdir"`)不存在，或它包含的文件名全都不同于源路径(本例中为 `c:"thedir"*.*)`中的文件，则不论指定了什么标志，`Handle` 都是 `NULL`：



相反，如果至少有一个重命名冲突发生，这个 `Handle` 就引用了有意义的数据块，因此，也就返回了可用的内存地址。

使用文件映射对象

获取文件名映射对象的 `Handle` 只是完成了一半工作，现在我们来评估一下，这个 `Handle` 是多么地有用！在资料中仅简单地说，(在非 `NULL` 时)`hNameMappings` 指向一个 `SHNAME_MAPPING` 结构的数组。并没有提到怎样获得这个数组的尺寸。更有甚者，说这个 `SHFileOperation()` 用于存储 `Handle` 的 `LPVOID` 成员不是一个指向数据结构数组的指针。显然，简单地经由循环遍历数组的方法在这里就不能工作了。

在有些旧的 *MSDN* 资料中，你将发现两个提到的函数 `SHGetNameMappingCount()` 和 `SHGetNameMappingPtr()`。然而，现在这两个函数不仅在资料中没有说明，而且也没有公开。在 *Shell32* (来自 *IE4.0* 或更高)的版本中也没有它们的任何踪迹。这样就很不好，因为它们确实是使你能正确编码

的函数。不可理解，为什么删除了这些函数，而且对 hNameMappings 成员的支持显得既生冷又陈旧。

一个未写进资料的结构

资料说明的东西是真的，但是，是不完整的。问题在于它忽视了上面提到的落在 hNameMappings 和数组之间的数据结构。有两条线索是我获得了正确的踪迹，第一，来自下面代码的输出：

```
TCHAR* pNM = static_cast<TCHAR*>(shfo.hNameMappings);  
Msg(pNM);
```

在测试这段代码时，我顺利地获得了另一种访问非法错，奇怪的是，它正好重复了错误号(如9)。这是重命名冲突错误号吗？在检查了目录之后发现，确实是。当然我立即执行了另一个使用不同文件数的检测，并且验证了这个想法。无论 hNameMappings 指向什么，开始都与全体文件名映射对象数一致。

所以下一步的工作将是浏览 Internet 客户端 SDK 和 MSDN 文档，探讨某些未知的剪裁板格式，它们是：

Windows ShellAPI 和拖拽操作

MSDN 的知识库文章 Q154123

这些格式(其中有一个是“文件名映射”)，在请求拷贝和粘贴操作时，或在从一个文件夹到另一个文件夹拖拽文件对象时是由 Shell 内部使用的。更有趣的是，很多这样的格式在剪裁板中都是以数据块的方式存储的，包含了一个数字和一个指向客户数据结构的指针。数字表示数组的尺寸，指针指向他的第一个元素。

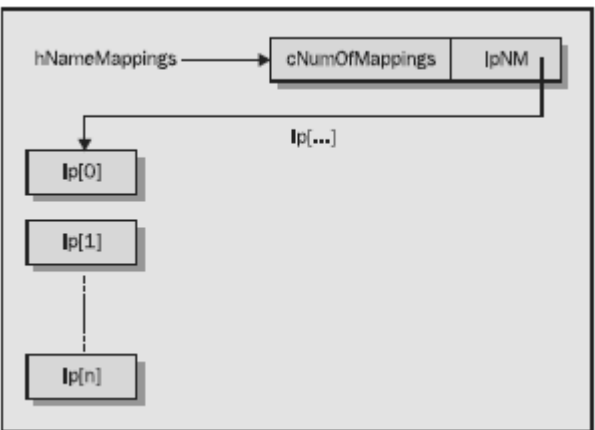
近似的方案

高兴的是同样的模式也可以应用到了映射对象，所以，我定义了一个结构

SHNAME_MAPPINGHEADER 具有如下格式：

```
struct SHNAME_MAPPINGHEADER  
{  
    UINT cNumOfMappings;  
    LPUSHNAME_MAPPING lpNM;  
};  
typedef SHNAME_MAPPINGHEADER* LPUSHNAME_MAPPINGHEADER;
```

这个结构实际上与 hNameMappings 所指向的数据有相同的格式。画图说明如下，这也说明了一种访问 SHNAME_MAPPING 数据结构的方法：



如此，写一个函数来枚举所有文件名映射对象就是直接了当的事情了；我把它称之为。

SHEnumFileMapping()。在观察函数本身之前，先要扩展一下前面的 OnOK()，以包含对该函数的调用：

```
void OnOK(HWND hDlg)  
{
```

```

...
//跟踪这个 handle 的值
Msg("hNameMappings is: %x", shfo.hNameMappings);
//枚举文件映射对象
SHEnumFileMapping(shfo.hNameMappings, ProcessNM,
reinterpret_cast<DWORD>(GetDlgItem(hDlg, IDC_LIST)));
//如推荐那样释放对象
if(shfo.hNameMappings)
    SHFreeNameMappings(shfo.hNameMappings);
}

```

SHEnumFileMapping() 函数接受 Handle, 回调过程, 和通用缓冲。它枚举所有 SHNAMEMAPPING, 并逐一传送它们给回调函数, 以便作进一步的处理。

```

int WINAPI SHEnumFileMapping(HANDLE hNameMappings, ENUMFILEMAPPROC lpfnEnum,
                             DWORD dwData)
{
    SHNAMEMAPPING shNM;
    //检查 Handle
    if(!hNameMappings)
        return -1;
    //获得结构头
    LPSHNAMEMAPPINGHEADER lpNMH = static_cast<LPSHNAMEMAPPINGHEADER>(hNameMappings);
    int iNumOfNM = lpNMH->cNumOfMappings;
    //检查函数指针; 如果 NULL, 直接返回映射数
    if(!lpfnEnum)
        return iNumOfNM;
    //枚举对象
    LPSHNAMEMAPPING lp = lpNMH->lpNM;
    int i = 0;
    while(i < iNumOfNM)
    {
        CopyMemory(&shNM, &lp[i++], sizeof(SHNAMEMAPPING)); if(!lpfnEnum(&shNM,
                                     dwData))
            break;
    }
    //返回实际处理的对象数
    return i;
}

```

SHEnumFileMapping() 函数与绝大多数 Windows 枚举函数所遵循的模式一样。它接受回调函数和通用缓冲, 这个缓冲用于保存调用程序传输给回调函数的客户数据, 此外, 它期望回调函数在终止枚举时返回0。我定义的回调函数类型为 ENUMFILEMAPPROC:

```
typedef BOOL (CALLBACK *ENUMFILEMAPPROC) (LPSHNAMEMAPPING, DWORD);
```

这个函数接受一个 SHNAMEMAPPING 对象的指针, 和调用程序发送的客户数据。

当然使用类枚举函数列出所有结构是一个个人偏好。等价地也可以使用导航界面, 提供 FindFirstSHNameMapping() 和 FindNextSHNameMapping() 函数。

事实上，由回调函数执行这个操作要好得多。在这里我所使用的(ProcessNM())是从任何它所接收的SHNAMEMAPPING 结构中抽取 pszOldPath 和 pszNewPath 字段值。并且把它们加到报告的列表观察中：

```
BOOL CALLBACK ProcessNM(LPSHNAME_MAPPING pshNM, DWORD dwData)
{
    TCHAR szBuf[1024] = {0};
    TCHAR szOldPath[MAX_PATH] = {0};
    TCHAR szNewPath[MAX_PATH] = {0};
    OSVERSIONINFO os;

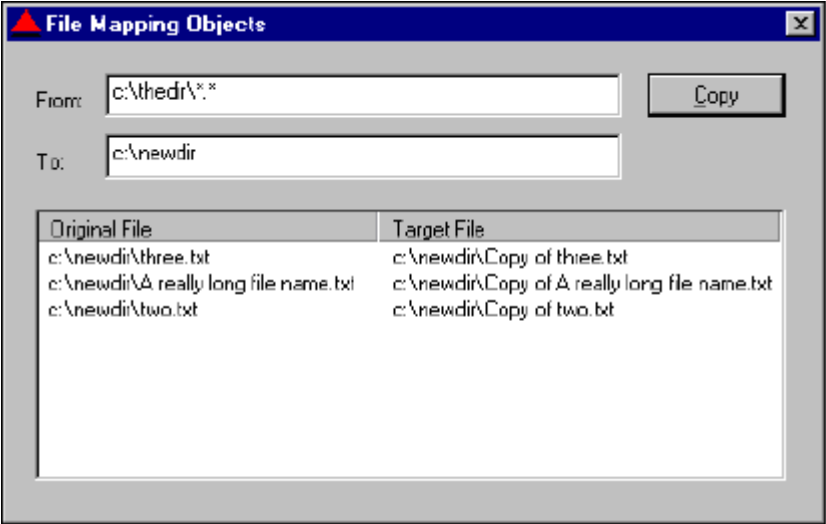
    //我们需要知道在什么样的 OS 上
    os.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);
    GetVersionEx(&os);
    BOOL bIsNT = (os.dwPlatformId == VER_PLATFORM_WIN32_NT);
    //在 NT 下，SHNAMEMAPPING 结构包含 UNICODE 串
    if(bIsNT)
    {
        WideCharToMultiByte(CP_ACP, 0,
            reinterpret_cast<LPWSTR>(pshNM->pszOldPath),
            MAX_PATH, szOldPath, MAX_PATH, NULL, NULL);
        WideCharToMultiByte(CP_ACP, 0,
            reinterpret_cast<LPWSTR>(pshNM->pszNewPath),
            MAX_PATH, szNewPath, MAX_PATH, NULL, NULL);
    }else{
        lstrcpy(szOldPath, pshNM->pszOldPath);
        lstrcpy(szNewPath, pshNM->pszNewPath);
    }

    //保存列表观察 Handle
    HWND hwndListView = reinterpret_cast<HWND>(dwData);
    //建立 "0"分隔的串
    LPTSTR psz = szBuf;
    lstrcpy(psz, szOldPath, pshNM->cchOldPath + 1);
    lstrcat(psz, __TEXT("\"0\""));
    psz += lstrlen(psz) + 1;
    lstrcpy(psz, szNewPath, pshNM->cchNewPath + 1);
    lstrcat(psz, __TEXT("\"0\""));
    //加串到报告观察中
    LV_ITEM lvi;
    ZeroMemory(&lvi, sizeof(LV_ITEM));
    lvi.mask = LVIF_TEXT;
    lvi.pszText = szBuf;
    lvi.cchTextMax = lstrlen(szBuf);
    lvi.iItem = 0;
    ListView_InsertItem(hwndListView, &lvi);
    psz = szBuf + lstrlen(szBuf) + 1;
    ListView_SetItemText(hwndListView, 0, 1, psz);
}
```

```
return TRUE;
}
```

注意，在 Windows NT 下，SHNAME_MAPPING 结构中的串是 Unicode 格式的。因此，如果操作系统是 NT，则转换串到 ANSI 格式，以便在例程中使用它们。还要注意的 dwData 缓冲，它用于传输列表观察的 Handle 到回调函数。

把这个代码与较早期的例子集成到一起后，现在就能够给出调用 SHFileOperation() 函数引起重命名文件的详细过程。在典型情况下测试，可以看到下面的情况：



小结

这一章深入讨论了一个函数 SHFileOperation()，对它的每一个方面都作了彻底地测试了。从拷贝，移动，重命名或删除文件，以及设置标志改变函数行为开始，然后展开了对某些未写进资料的返回码，Bugs，函数缺陷的讨论。概括地讲，在这一章中，给出了：

- 怎样编程 SHFileOperation()
- 最普遍的编程错。
- 这个函数在资料方面的短缺
- 怎样利用文件名映射的优点