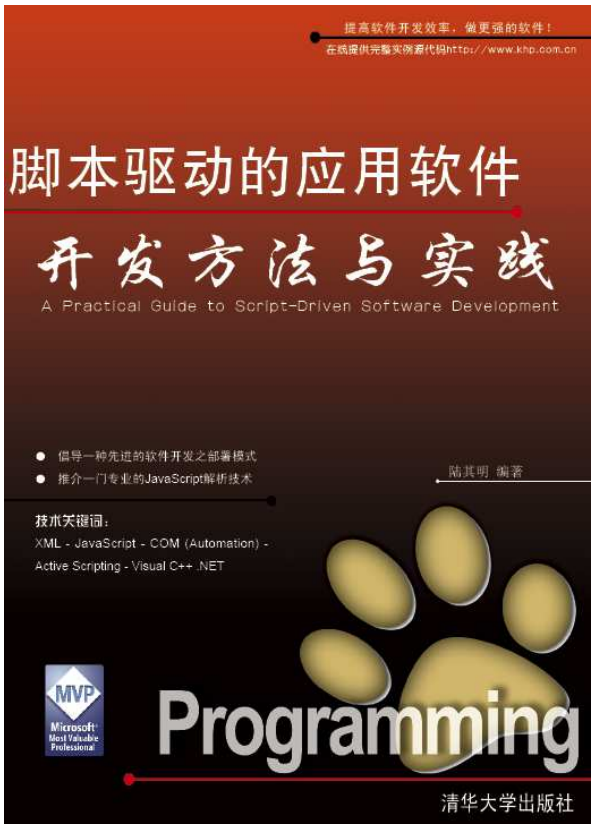

China-pub: http://www.china-pub.com/computers/common/info.asp?id=32125
第二书店: http://www.dearbook.com.cn/book/119861
新风雨: http://www.cnforyou.com/query/bookdetail1.asp?viBookCode=6414

脚本驱动的应用软件开发方法与实践

A Practical Guide to Script-Driven Software Development

作者：陆其明

- 倡导一种先进的软件开发之分工模式
- 推介一门专业的 JavaScript 解析技术

内容提要

本书纵观了软件开发在不同发展阶段的特点，揭示了当今在很多大型应用软件设计、开发过程中采用的一种分工模式，引出了一种称之为“脚本驱动的应用软件”的开发方法。这种方法可以促进软件开发过程中的团队协作，提高软件开发的效率，提升应用软件在可定制性和交互性方面的表现。在传授方法的同时，本书更结合以递进式的实例，使

得方法更容易理解、更贴近于实际应用。在开发脚本驱动的演示程序时，本书还介绍了XML 解析、JavaScript 解析等多种实用的编程技术。

本书广泛适合于计算机应用软件系统的设计人员以及开发人员，对于指导 XML 技术入门也有一定的帮助，在 JavaScript 解析技术方面更是一部不可多得的实务指南。

目 录

第 1 阶段　原始的软件开发

第 2 阶段　分工协作的软件开发

第 1 阶段　原始的软件开发

以前常常听人这么说，某某软件的作者是谁、谁谁发布了一款多么强大的某某软件。那个时候，一款软件从需求分析到功能定义、架构设计、编码实现、乃至测试，都是由（或者说主要是由）一个人来完成的。这个人是高手，是全才，是个大能人，是众多程序员崇拜的偶像。那个时代的软件产品，也往往附带着强烈的个人英雄主义色彩。为了方便阐述和对比，本书将这个阶段的软件开发称作为原始的软件开发。

时至今日，一些个人软件仍然在延续着这个美丽的传说。本书接下去的部分，将以开发一个简单的看图软件为例，力图重现这种个人软件的开发过程。不为附庸风雅，但求管中窥豹、略见一斑。

第 1 章 开发一个看图软件

1.1 需求分析

什么是看图软件？简单来说，看图软件的任务就是要解析各种格式的图像文件，经过一定的处理，最终把图像的内容直观地显示在电脑屏幕上。好的看图软件可以帮助用户快速、高质量地浏览电脑中的图片，功能更强的还可以对图像进行一些数字处理，比如裁剪、亮度调整、锐化、柔化等等。

市面上已经有很多优秀的看图软件了，比如著名的 ACDSee，甚至现在的 Windows 操作系统本身就集成有用于图像预览的组件了。美国人常说：“Don’t reinvent the wheels.”意思是说，我们没必要重复发明轮子，因为轮子早就有人发明了（我们应该直接使用前人的工作成果，“站在巨人的肩膀上”，以便创造更高的价值）。那为什么我们还要来开发一个自己的看图软件呢？我的回答是，纯粹的个人兴趣！

需要说明的是，开发这么一个看图软件只是为了展示原始的软件开发过程，至于这个软件实现了哪些功能倒并不是那么重要——只要能够支持常见的几种图像文件格式如 JPEG、GIF、BMP 等，能够将图像内容显示出来，并且能对图像内容进行一定的加工处理，足矣！另外，目前这个版本的看图软件只是一个起点，仅作抛砖引玉之功；随着本书后续章节的顺序展开，我们将逐步逐步看到其他版本的如基于 XML 的看图软件、用户可定制的看图软件，乃至最后的脚本驱动的看图软件。理解这些不同版本的看图软件之间的差别，才是阅读本书的重中之重。

1.2 概要设计

这个看图软件取名为 ImageViewer，是一个标准的 Windows 风格的应用程序，如图 1.1。ImageViewer 主界面中央的一块区域用于显示当前图像的内容，其下方的状态条用于显示当前图像文件的所在路径，其上方的一排按钮用来响应用户的操作指令。这些按钮的功能定义如下：

- 按钮 Open：用于打开本地硬盘上的一个图像文件。如果操作成功，则图像内容随即显示在下方的窗口中。
- 按钮 Reload：用于重新装载源文件。如果当前打开的图像已经被作过一些处理（比如反色、灰度化等），则经过 Reload 之后这些处理效果都将被丢弃。
- 按钮 Clean：用于将图像内容完全擦除。
- 按钮 Save As：用于将当前显示的图像内容另存为一个用户指定的 BMP 文件。
- 按钮 Invert：用于将当前显示的图像内容反色。
- 按钮 Greyscale：用于将当前图像内容灰度化，即转换成黑白图像。
- 按钮 Logo：用于在当前图像的左上角叠加一个小的 Logo 图像（像电视节目左上角叠加的电视台台标一样）。

- 按钮 System Time：用于在当前图像的右上角叠加当前的系统时间，格式为 yyyy-mm-dd(HH:MM:SS)。
- 按钮 About：用于弹出一个消息框，说明本软件的版权、版本等信息。

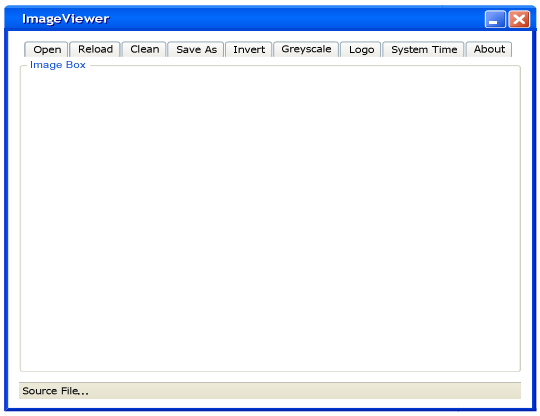

图 1.1 看图软件 ImageViewer 的用户界面

除了上述的用户界面（UI，User Interface）设计之外，在内部实现上还要求将 UI 逻辑和商业应用逻辑分离（这里的商业应用逻辑就是图像处理功能的实现）——这是很多优秀的商业软件都采取的一种设计方法。采用这种方法，不仅可以使程序的结构清晰，还有利于代码重用。因为同一款软件的不同升级版本往往会使用不同的用户界面，但其商业逻辑相对来说保持了一定的稳定性。在这种情况下，商业逻辑的实现是可以重用的，这将大幅提高升级版本软件的开发效率。

个人软件实现的功能相对来说都比较单一。因此其实很多个人软件都没有规范的前期分析、设计过程，往往是有了一个点子之后，一拍脑袋就开始下面的编码实现了。

1.3 编码实现

1.3.1 图像处理功能

根据 UI 逻辑与商业逻辑分离的原则，我们首先来实现一个图像处理模块。具体来说，是实现一个 ImageProcessor 类。这个类的输入为用户指定的一个图像源文件，输出为将图像内容显示到主界面上一个指定的窗口。

ImageProcessor 类的定义如下：

```
class ImageProcessor
{
public:
        ImageProcessor();
        ~ImageProcessor();
```

```
        ////////////////////// 文件操作 //////////////////////
        // 判断是否支持对用户指定的图像文件进行解码
        bool IsSupported(const char* inSrcFile);
        // 装载（并解码）用户指定的图像文件
        bool LoadFile(const char* inSrcFile);
        // 重新装载用户上次指定的源文件
        bool Reload();
        // 将当前图像另存为一个BMP文件
        bool SaveFile(const char* inDestFile);
        // 返回当前正被处理的源文件名
        CString GetFile() { return mSourceFile; }
        // 返回用于装载源图像的内存DC
        HDC GetImageDC() { return mMemDC; }
        // 返回图像文件解码之后生成的位图对象句柄
        HBITMAP GetImage() { return mSrcImage; }
        // 判断用户指定的图像文件是否已经被成功解码
        bool IsReady() { return mIsReady; }
        // 得到源图像的宽、高、像素位数等信息
        bool GetInfo(long* outWidth, long* outHeight = 0, long* outBitcount = 0);

        ////////////////////// 图像处理功能 //////////////////////
        // 在源图像特定的一个位置上叠加一个Logo图片
        bool OverlayLogo(int inX, int inY, const char* inLogoFile);
        // 在源图像特定的一个位置上叠加当前系统时间
        bool OverlaySystemTime(int inX, int inY);
        // 在源图像特定的一个位置上输出一串文字
        bool DrawText(int inX, int inY, const char* inText);
        // 将当前图像反色
        void Invert();
        // 将当前图像灰度化
        void Greyscale();
        // 将当前图像内容完全清除
        void Clean();

        ////////////////////// 输出、显示 //////////////////////
        // 将当前图像内容显示在指定的窗口中
        void Display(CWnd* inDestWnd);

        ////////////////////// 选项 //////////////////////
        // 图像显示时是否保持源图像的宽纵比？
        void KeepAspectRatio(bool enabled);

    protected:
        ////////////////////// 内部函数 //////////////////////
        // 将用户指定的图像文件解码，并返回IPicture对象指针
        bool Decode(const char * inSrcFile, IPicture** outPic);
        // 将某个DC中指定的区域填充上指定的颜色（清屏效果）
        void PaintColor(HDC inDC, RECT* inRect, COLORREF inColor);
```

```
private:
        // 用户指定的图像文件
        CString         mSourceFile;
        // 用于解码图像文件的组件对象指针
        IPicture *      mIPicture;
        // 装载源图像的内存DC
        HDC             mMemDC;
        // 源图像对应的位图对象（包括各种处理效果）
        HBITMAP         mSrcImage;
        // 在内存DC中的原始位图对象
        HBITMAP         mOldImage;
        // 是否对图像文件成功解码？
        bool            mIsReady;
        // 显示时是否保持源图像的宽纵比？
        bool            mKeepAspectRatio;
};
```

提示：下面从 1.3.1.1 节到 1.3.1.7 节将要介绍的图像处理功能，大都是通过 Windows GDI 函数来实现的。（此部分省略，请下载、查看本书配套代码）

1.3.2 UI 设计与实现

当商业应用逻辑（也就是上述的 ImageProcessor 类）开发完成之后，就可以开始剩下的 UI 工作了。UI 工作主要分成两个部分：UI 元素的布局和 UI 逻辑的处理。接下去，我们就来分别看一下这两部分工作的具体内容。

1.3.2.1 UI 元素布局

参考概要设计时给出的 UI 草图，在 VC 的资源编辑器中设计如图 1.2 的软件界面。这是一个繁琐的过程：需要按照设计说明书加入各种按钮、静态控件、状态条等；需要为各个窗口命名、分配合适的 ID、设置初始属性；需要调整各个窗口的大小，及其所在的位置，等等。

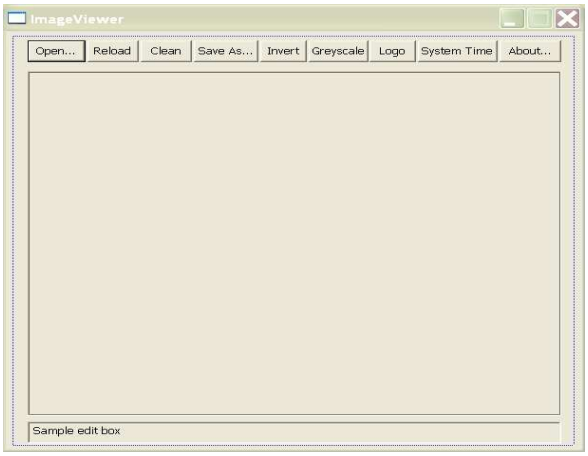

图 1.2 在 VC 的资源编辑器中设计软件界面

1.3.2.2 UI 逻辑处理

Windows 程序是消息驱动的。当 UI 元素的布局完成之后，就可以给各个窗口添加必要的消息处理函数了。比如当用户用鼠标点击某个按钮时，程序就会接收到一定的消息，而在这个消息处理函数中就要去执行一定的操作。

以 Open 按钮为例，通过 VC 的向导可以为它添加一个鼠标单击事件的响应。VC 会自动生成一个消息处理函数 OnBnClickedButtonOpen 的声明，还有一个空的函数体：

```
//
// ImageViewerDlg.h : header file
//
// Open 按钮被单击的响应函数的声明
afx_msg void OnBnClickedButtonOpen();


//
// ImageViewerDlg.cpp : implementation file
//
// Open 按钮被单击的响应函数的实现（暂空）
void CImageViewerDlg::OnBnClickedButtonOpen()
{
	// TODO: Add your control notification handler code here
}
```

当然还有将消息与消息处理函数连接起来的消息映射：

```
BEGIN_MESSAGE_MAP(CImageViewerDlg, CDialog)
	ON_WM_SYSCOMMAND()
	ON_WM_PAINT()
	ON_WM_QUERYDRAGICON()
	//}}AFX_MSG_MAP
	ON_BN_CLICKED(IDC_BUTTON_OPEN, OnBnClickedButtonOpen)
END_MESSAGE_MAP()
```

Open 按钮的功能是让用户选择一个图像源文件，之后交给图像处理模块去解码、加工处理，最终将图像内容显示在主界面中央的大窗口中。于是，OnBnClickedButtonOpen 函数可以作如下实现：

```
// Open 按钮被单击的响应函数的实现
void CImageViewerDlg::OnBnClickedButtonOpen()
{
	CString imageFile;	// 保存图像文件路径

	// 弹出一个标准的文件浏览对话框，让用户选择一个图像文件
	if (MiscUtils::BrowseImageFile(imageFile))
```

```
        {
                // 将用户选择的图像文件显示在状态条上
                // 注：mImageFile是CImageViewerDlg类的成员，CString类型
                mImageFile = imageFile;
                mEditImageFile.SetWindowText(mImageFile);

                // 让图像处理模块加载这个图像文件
                // 注：mProcessor是CImageViewerDlg类的成员，ImageProcessor类型的指针
                if (mProcessor->LoadFile(mImageFile))
                {
                        // 将图像内容显示出来
                        DisplayImage();
                }
        }
}
```

UI 逻辑处理使 UI 跟商业应用逻辑连接起来。除了 UI 元素的事件处理以外，UI 逻辑处理还包括 UI 元素的状态处理，比如在某些状态下哪些窗口是能用的、哪些是禁用的，哪些是可见的、哪些是隐藏的，等等。这里就不再赘述。

1.3.3 实例程序：ImageViewer_Basic

实例程序 ImageViewer_Basic 位于本书配套代码包的 ImageViewer_Basic 目录下。这是一个通过 Visual Studio .NET 2003 向导生成的、基于对话框的 MFC 应用程序。软件运行起来界面如图 1.3。

图 1.3 ImageViewer_Basic 的软件界面

第 2 阶段　分工协作的软件开发

常常听到有人这么感叹：“现在的软件真是越做越大啊！”一个软件产品的设计、开发、维护，涉及到的技术难度以及工作量之巨大已经远非个人能力所能及。这种软件产品，没有一个规范化运营的公司来支撑是不行了！软件开发的个人英雄主义悄然淡出了历史舞台，取而代之的是：分工协作、团队精神。

仍然以看图软件为例，后续的章节我们将进一步介绍开发支持皮肤的看图软件、以及用户可定制的看图软件的方法。与原始的软件开发相比，这里集中体现出来的是一种 UI 艺术设计的分离、以及 UI 设计与程序设计分工协作的特点。这种分工协作，使得软件开发更富有效率，使得最终的软件产品更加专业化。由于基于皮肤的看图软件和可定制的看图软件都牵涉到了 XML 技术，下面我们就先从 XML 的应用说起。

第 2 章 XML 的应用

2.1 XML 简史

早在 20 世纪 60 年代，IBM 公司就开始研究一种通用的标记语言（GML，Generalized Markup Language），用以解决在某个平台下建立的文档向另一个平台移植的问题。1978 年，美国国家标准学会（ANSI）将 GML 整理规范成 SGML（Standard Generalized Markup Language）。1986 年，国际标准化组织（ISO）批准并采用了 SGML，使之成为一个官方标准——ISO 8879。

SGML 是一种非常严谨的文件表示系统。然而也正因为其严谨，所以它过于复杂，难以理解和学习，导致它很难被推广应用。人们迫切需要一种容易入门的描述语言。于是，专家们对 SGML 进行了简化，衍生出一种称为 HTML（Hyper Text Markup Language）的简单标记语言。HTML 为早期的浏览器所支持，而且生逢其时，当时 Internet 正风靡全球，所以 HTML 很快便成为了制作网页的标准语言。然而，HTML 的问题在于它太过简单，交互性差，难以扩展。随着 Internet 的发展，网页制作者们要向网页中加入多媒体、创建比较复杂的应用时，HTML 设计上的缺陷便暴露无遗了。

1996 年 8 月，为了解决以上问题，专家们对 SGML 再次进行了去繁就简的工作，留下了 SGML 的精华，于是产生了 XML（eXtensible Markup Language）。

1996 年 11 月，新的数据描述语言 XML 公布于世，并向万维网联盟（W3C）正式提交了议案。1998 年 2 月，W3C 公布了 XML 1.0 标准。

XML 的公布引起了业界的广泛关注。人们确信 XML 是又一个里程碑式的技术。我们从 XML 的设计目标便可见一斑：

- 可在 Internet 上直接使用。
- 支持大量的不同应用。
- 与 SGML 兼容。
- 处理 XML 文档的程序应该容易编写。
- XML 中的可选项应尽可能少，理想情况下应为零。
- XML 文档清晰易读。
- 应该易于设计。
- XML 的设计应该井井有条并且简明扼要。
- XML 文档应该易于创建。
- XML 标记的简洁性较为次要。

值得注意的是，HTML 与 XML 并不是“父子”关系。XML 被称之为可扩展的标记语言，事实上它并不是一种真正意义上的标记语言，而是一种允许用户对自己的标记语言进行定义的元语言。也就是说，XML 是一种定义语言的语言，是一种可扩展的灵活格式。如果说 HTML 是一个描述系统，而 XML 则是用来定义这些描述系统的系统。（也就是说，使用 XML 可以定义出 HTML 系统。）HTML 和 XML 两者都源自于 SGML，HTML 是 SGML 的具体应用，而 XML 是 SGML 的一个子集。

2.2 XML 基本概念

深入到 XML 中，会发现 XML 中的概念有如此之多：DTD、大纲、验证、标记、元素、属性……元素在 DTD 中的声明，元素还有 EMPTY、ANY、ELEMENT、MIXED 等类型…… 还有其他的相关技术，如 CSS1、CSS2、XSL、XLink、XPointer、XPath、DOM……真是不胜枚举啊！这一头扎进来，犹如刘姥姥进了大观园，傻眼了！

本书并不打算对上述所有这些概念进行面面俱到的介绍。本着“够用就行”的原则，我们只要重点搞明白 XML 的文档结构、文档类型定义、元素、属性、解析器等基本概念即可。

2.2.1 文档结构

一个 XML 文档最多可以由 4 个部分组成（其中 3 个部分是可选的）。它们出现的次序如下：

- 序言（Prolog，可选）
- DTD（可选）
- 文档根（Document Root，必选）
- 结尾（Epilog，可选）

XML 文档的序言部分一般出现在根元素开始标记之前。它包括应用于整个文档的信息，如字符编码、文档结构、样式表引用等。另外，它还包括 XML 声明、注释、处理指令等。其中，XML 声明是序言用得最多的一部分（很多解析器甚至不接受没有 XML 声明的文档）。XML 声明包含以下几个部分：XML 语言的版本（version）、文档字符的编码格式（encoding）、是否依赖其他外部文件（standalone）。

实际上，XML 声明只要 version 即可。它的语法与 XML 元素不同——它以 **<?** 开始，以 **?>** 结束。典型如：<?xml version=“1.0”?>。

DTD（文档类型定义）用于定义某一类型的 XML 文档的语法，规范其结构。关于 DTD 更详细的介绍，参见 2.2.2 节。DTD 用另外的文件来描述，扩展名一般就是.dtd。在 XML 文档中声明其所采用的 DTD 的部分典型如：<!DOCTYPE bookInfo SYSTEM “bookInfo.dtd”>。

文档根是 XML 文档的必选部分，它位于序言和 DTD 之后，并在所有的数据元素之前。文档根其实就是普通的 XML 数据元素，它也可以有属性。文档根在一个文档中必须存在，而且只能有一个。

XML 文档的结尾与序言相似，很少使用。

综上所述，一个合法的、最简单的 XML 文档可能如下：

```
<?xml version="1.0" encoding="UTF-16"?>
<Book>
        <Name>脚本驱动的应用软件开发方法与实践</Name>
        <Author>陆其明</Author>
</Book>
```

提示：上述这个 XML 文档没有使用 DTD，它符合 XML 1.0 规范，文档内容采用 UTF-16 编码格式，<Book>是根元素，<Name>和<Author>是一般的数据元素。

2.2.2 文档类型定义

为了理解文档类型定义（DTD，Document Type Definition），须先了解什么是文档类型。所谓文档类型，就是指一类相似的文档，如电话号码簿、库存清单、财务报表等。而文档类型定义是为了用 XML 来表示某个特定类型的文档而定义的一组规则，是 XML 规范提供的模式类型。它明确指定了可以在 XML 文档中使用哪些标记（Tag），这些标记应该按什么次序出现，哪些标记可以出现在其他标记中，哪些标记有属性，等等。

也就是说，DTD 定义了文档的语法，规范了文档的结构。对编写的任何 XML 文档，DTD 就是一本规则说明书。有效的 XML 文档必须遵从 DTD 所定义的结构和规则，否则该文档就是无效的！

这里还要区别两个概念：格式良好的 XML（Well-formed XML）和有效的 XML（Valid XML）。所谓格式良好的 XML 文档，简单来说，就是符合 XML 语法规则的 XML 文档。而一个有效的 XML 文档首先必须是格式良好的，同时必须符合 DTD 对元素的定义。由此可见，一个有效的 XML 文档一定是格式良好的，而格式良好的不一定是有效的。

具体来说，格式良好的 XML 文档应该满足下列条件：

- 该文档有且只有一个根元素。
- 每个元素都必须有开始和结束标记。
- 开始或结束标记的大小写应该一致。
- 能正确地表示空元素。
- 元素间应该是并列关系，或者是完全包含关系，不能交叉。
- 元素中的属性写在开始标记内，属性值都要用引号。

有效的 XML 文档应该满足的条件如下：

- 文档必须是格式良好的。
- 文档根元素的名字必须与 DTD 中的名字匹配。
- 文档必须有一个 DTD，在该 DTD 中声明了所有用于文档中的元素、属性和实体。
- 文档必须符合由 DTD 规定的规则。

2.2.3 元素和属性

每个 XML 文档都包含有一个或多个元素。元素是文档的主要逻辑部件，它由开始标记和结束标记界定。每个元素都有一个用于标识的名字，名字必须以一个字母或下划线开始，后跟字母、数字、连字符、下划线或圆点。

XML 文档中的元素结构和一本书的章节大纲原理差不多，是一种“树形结构”。包括所有其他元素的元素称为“根元素”，它是唯一没有被包含在其他元素中的元素。包含在根元素中的元素称为根的“子元素”，它们自己还可以包含其他的元素，称为“树枝”；不再包含有其他元素的子元素，则称之为“树叶”。元素可以附带额外的信息，称为“属性”，它描述了元素的特征。属性对于元素来说不是必需的。

属性书写于元素的开始标记内，属性值一定要放在一对单引号或双引号之间。如果元素有多个属性，属性之间用空格分隔。因为 XML 是区分大小写的，所以对属性的引用也应该保持大小写一致。

一个具有属性的元素，其表示形式如图 2.1：

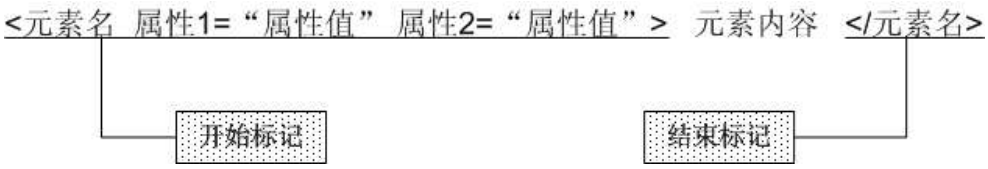

图 2.1 元素及其属性的表示形式

2.2.4 解析器

与 HTML 一样，XML 也需要解析器对文档进行解析。IE 便是 HTML 的解析器之一，目前 IE 也能解析 XML。

解析器分为两种：非验证性解析器和验证性解析器。非验证性解析器只检查文档的格式是否良好，也就是说它只按 XML 1.0 规范的基本定义验证文档的有效性。而验证性解析器除了进行上述的格式验证外，还能根据该 XML 文档关联的 DTD 文档，检查 XML 文档中的元素和属性命名是否符合 DTD 中预先的定义。

换句话说，其实两种解析器都需要验证文档，只是验证的参考标准不一样。 “非验证性”解析器只验证结构，只要遵守了 XML 1.0 规范，我们可以任意的命名元素和属性。我们可以用<User>表示一个用户，也可以用<UserName>表示一个用户。Age 表示年龄，<User Age=“Sixteen”> </User>。至于年龄的属性值，可以用“Sixteen”，也可以用数字“16”，总之格式正确就行。而“验证性”解析器除了验证结构外，还检查元素的命名。假设 DTD 文档中定义了元素为<User>，那我们只能用这个 User 作为元素名来表示一个用户；DTD 文档中甚至可以规定 Age 属性只能填数字，而如果一个 XML 文档中的 Age 为英文字母的话，验证性解析器就将认为这个 XML 文档是无效的。

大部分时候，我们所说的 XML 文档，指的是能通过“非验证性”解析器验证的格式良好的 XML 文档。对于非验证性解析器来说，DTD 文档没有任何意义。

2.3 XML 解析

所谓 XML 解析，就是要对指定的 XML 文档内容进行解释和分析。但是，在不同的应用场合下，比如在网页中怎么来解析 XML 呢？怎么通过 C++编程来实现呢？下面，我们就分别来看一下这两种应用情况。

2.3.1 网页中读取 XML 文件

从 IE 5.0 开始，微软公司提供了一个名为“Microsoft.XMLDOM”的 ActiveX 控件，专门用于 XML 解析。大家知道，ActiveX 控件可以在网页中通过 JavaScript 代码来创建。于是，网页中操作 XML 文档也就不成问题了。

现在假设有一个名为 BookInfo.xml 的 XML 文件，内容如下：

```
<?xml version="1.0" encoding="UTF-16"?>
<Book>
	<Name>
		<Title type="Chinese">脚本驱动的应用软件开发方法与实践</Title>
		<Title type="English">A Practical Guide to Script-Driven Software Development</Title>
	</Name>
	<Author>陆其明</Author>
	<City>上海</City>
	<EMail>luqiming@gmail.com</EMail>
</Book>
```

创建一个空白网页，命名为 xmlReader.htm。然后写一段 JavaScript 代码嵌入到<BODY>与</BODY>之间，如下：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>

<HEAD>
<META http-equiv=Content-Type content="text/html; charset=unicode">
<META content="MSHTML 6.00.2900.2802" name=GENERATOR>
<TITLE>演示在网页中读取 XML 文件</TITLE>
</HEAD>

<BODY>
<SCRIPT LANGUAGE="JavaScript">
	// 创建一个 XML 解析器对象
	var xml_doc = new ActiveXObject("Microsoft.XMLDOM");
	xml_doc.async = "false"; // 异步模式
	// 装载 BookInfo.xml 文件
	xml_doc.load("BookInfo.xml");

	// 获取 XML 文档的根节点
	var theRoot = xml_doc.documentElement;
	// 获取根节点下面的所有子节点
	var nodes = theRoot.childNodes;

	// 读取根节点下所有子节点的名字和值
	document.write("<br>--- 根节点下共有" + nodes.length + "个子节点---<br><br>");
	for (i = 0; i < nodes.length; i++)
	{
		index = i + 1;
		document.write("第" + index + "个节点的名字是：" + nodes.item(i).nodeName + "<br>");
		document.write("第" + index + "个节点的值是：" + nodes.item(i).text + "<br>");
	}
</SCRIPT>
</BODY>

</HTML>
```

使用 IE 打开 xmlReader.htm 文件，就可以看到如图 2.2 的网页内容了：

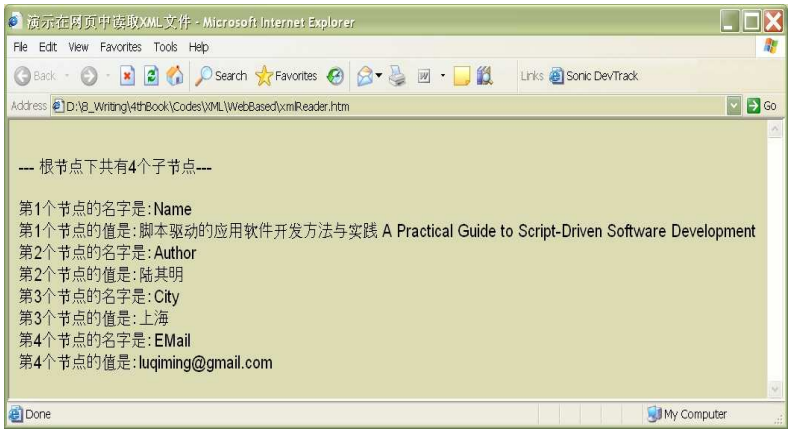

图 2.2 在网页中读取 XML 文件

提示：本书配套代码包的 XML\WebBased 目录下提供了本节演示所使用的源文件。

2.3.2 用 C++编码实现解析

在网页中解析 XML 并不是本书的重点。我们关心的是，如何通过 C++编码来实现 XML 的解析。在这里，我们仍然能够利用微软公司的“Microsoft.XMLDOM”这个组件。只不过，我们现在要用 C++代码来创建并使用它。

下面，我们将结合一个简单的 C++程序：XMLParser，来大致看一下 XML 文件在 C++程序中的操作方法。

2.3.2.1 CXmlWrapper 类

首先要来介绍一个包装类 CXmlWrapper，它封装了“Microsoft.XMLDOM”组件的常用接口函数。解析一个特定的 XML 文件，不用每次都去跟这个组件的底层接口打交道，直接使用 CXmlWrapper 类要方便得多！

CXmlWrapper 类的声明如下：

```
//
// CXmlWrapper.h
//

#ifndef __H_CXmlWrapper__
#define __H_CXmlWrapper__

#import <msxml.dll>
#include <afxtempl.h>
```

```
typedef CArray<IXMLDOMNode*, IXMLDOMNode*> DOMNodeList;

class CXmlWrapper
{
public:
	CXmlWrapper();
	CXmlWrapper(const char * inXmlFile);
	virtual ~CXmlWrapper();

public:
	////////////////// 文件操作 /////////////////
	// 装载一个指定的XML文件
	bool LoadFile(const char * inXmlFile);
	// 将当前的XML文档保存回原文件
	bool SaveFile();
	// 将当前的XML文档另存为一个XML文件
	bool SaveFile(const char * inXmlFile);
	// XML解析状态是否正常？
	bool IsReady();
	// 得到当前正被解析的XML源文件
	CString GetFileName() { return mXmlFile; }

public:
	////////////////// （元素）节点操作 /////////////////
	// 创建一个节点
	IXMLDOMNode* CreateNode(BSTR inNodeName, DOMNodeType inNodeType=NODE_ELEMENT);
	IXMLDOMNode* CreateNode(const char * inNodeName, DOMNodeType inNodeType =
NODE_ELEMENT);
	// 删除一个节点
	bool RemoveNode(IXMLDOMNode * inParentNode, IXMLDOMNode * inSonNode);
	// 插入一个节点
	bool InsertNode(IXMLDOMNode * inParentNode, IXMLDOMNode * inSonNode);
	// 创建一个指定名字、内容的元素节点，然后插入到某个节点之下
	bool InsertNodeEx(const char * inNodeName, const char * inNodeText,
		IXMLDOMNode * inParentNode = NULL);

	////////////////// 属性操作 /////////////////
	// 获取指定节点的某个属性的值
	CString GetNodeAttr(IXMLDOMNode * inNode, const char * inAttr);
	// （多个重载函数）为指定的节点设置某个属性的值
	bool SetNodeAttr(IXMLDOMNode* inNode, const char * inAttr, char * szVal);
	bool SetNodeAttr(IXMLDOMNode* inNode, const char * inAttr, const char * szVal);
	bool SetNodeAttr(IXMLDOMNode* inNode, const char * inAttr, long nVal);
	bool SetNodeAttr(IXMLDOMNode* inNode, const char * inAttr, unsigned long nVal);
	bool SetNodeAttr(IXMLDOMNode* inNode, const char * inAttr, int nVal);
	bool SetNodeAttr(IXMLDOMNode* inNode, const char * inAttr, unsigned int nVal);
	bool SetNodeAttr(IXMLDOMNode* inNode, const char * inAttr, double dblVal);
	bool SetNodeAttr(IXMLDOMNode* inNode, const char * inAttr, _variant_t vVal);
```

```
        ////////////////// 节点的名字、内容 //////////////////
        CString GetNodeName(IXMLDOMNode * inNode);
        CString GetNodeTextEx(const char * inNodeName, IXMLDOMNode * inParentNode =
NULL);
        CString GetNodeText(IXMLDOMNode* inNode);
        bool SetNodeText(IXMLDOMNode* inNode, const char * inText);

        ////////////////// 节点的遍历 //////////////////
        // 得到指定节点下的第一个子节点
        IXMLDOMNode* GetNodeFirstChild(IXMLDOMNode * inNode);
        // 得到指定节点下的最后一个子节点
        IXMLDOMNode* GetNodeLastChild(IXMLDOMNode * inNode);
        // 得到指定节点的下一个兄弟节点
        IXMLDOMNode* GetNodeNextSibling(IXMLDOMNode * inNode);
        // 得到指定节点的前一个兄弟节点
        IXMLDOMNode* GetNodePreviousSibling(IXMLDOMNode * inNode);
        // 得到指定节点下的所有子节点（必要时可以指定节点的名字）
        bool GetNodeSubNodeList(DOMNodeList * outNodeList, IXMLDOMNode * inParentNode,
const char * inWhatName = NULL);
        // 得到指定节点下的某个名字的（第一个）子节点
        IXMLDOMNode* GetNodeSubNode(IXMLDOMNode * inParentNode, const char *
inWhatName);

protected:
        CString               mXmlFile;      // XML文件名
        IXMLDOMDocument *     mDocNode;      // 组件对象指针
        IXMLDOMNode *         mRootNode;     // 文档根节点

protected:
        bool SetFileName(const char * inXmlFile);
        void Clean();

        // 内部函数：从IXMLDOMElement接口转换到IXMLDOMNode接口
        inline bool Element2Node(IXMLDOMNode ** outNode, IXMLDOMElement * const
inElement)
        {
              HRESULT hr = inElement->QueryInterface(IID_IXMLDOMNode, (void**)outNode);
               return SUCCEEDED(hr);
        }

        // 内部函数：从IXMLDOMNode接口转换到IXMLDOMElement接口
        inline bool Node2Element(IXMLDOMElement ** outElement, IXMLDOMNode * const
inNode)
        {
               HRESULT hr = inNode->QueryInterface(IID_IXMLDOMElement,
(void**)outElement);
               return SUCCEEDED(hr);
        }
};
```

```
#endif // __H_CXmlWrapper__
```

CXmlWrapper 类的实现如下：**（此部分省略，请下载、查看本书配套代码）**

2.3.2.2 遍历每个节点

现在假设有这样一个 XML 文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<ImageViewer version="1.2">
	<MainWindow skin="main.jpg">
		<ImageBox xpos="20" ypos="50" width="582" height="446"/>
		<Button name="btn_Open" xpos="10" ypos="10" skin="btn_Open.jpg"/>
		<Button name="btn_Reload" xpos="50" ypos="10" skin="btn_Reload.jpg"/>
		<Hyperlink name="http://www.chery.cn" xpos="100" ypos="10"/>
	</MainWindow>
</ImageViewer>
```

为了解析这个特定的 XML 文件，我们来新建一个类：CXmlParserDemo，它的声明如下：

```
//
// CXmlParserDemo.h
//

#ifndef __H_CXmlParserDemo__
#define __H_CXmlParserDemo__

#include "CXmlWrapper.h"

class CXmlParserDemo
{
private:
	CString		mXmlFile;	// XML源文件
	CXmlWrapper *	mXmlManager;	// XML解析器

	IXMLDOMNode *	mMainWindowNode;	// <MainWindow>节点
	IXMLDOMNode *	mCurrentNode;		// 当前节点

public:
	CXmlParserDemo();
	virtual ~CXmlParserDemo();

	// 设置XML源文件
	bool SetXmlFile(const char * inFile);
	// 获取<MainWindow>节点的“skin”属性的值
	CString GetMainWindowSkin();
```

```
        // 找到<MainWindow>下的第一个子节点
        bool FindFirstNode();
        // 定位到当前节点的下一个兄弟节点
        bool FindNextNode();
        // 找到一个指定名字的<Button>节点
        bool FindButton(const char* inName);

        // 获取当前<Button>节点的各个属性
        CString GetNodeName();
        CString GetName();
        CString GetSkin();
        long GetXPos();
        long GetYPos();
        long GetWidth();
        long GetHeight();
        void GetPosition(long& xpos, long& ypos, long& width, long& height);
        CString FullPath(CString const& inFileName);

protected:
        // 内部函数：获取当前节点的属性
        inline CString GetAttribute(const char * inTag)
        {
                if (mCurrentNode)
                {
                        return mXmlManager->GetNodeAttr(mCurrentNode, inTag);
                }
                return CString("");
        }

        void Finalize();
};

#endif // __H_CXmlParserDemo__
```

CXmlParserDemo 类的实现如下：**（此部分省略，请下载、查看本书配套代码）**

于是，当我们想要遍历<MainWindow>下的所有节点时，只需要编写如下的代码：

```
// 创建一个 XML 解析器对象
CXmlParserDemo skinParser;

// 设置XML源文件
if (skinParser.SetXmlFile("ImageViewer.xml"))
{
        printf("\nThe skin folder <<< %s >>>\n\n", MiscUtils::GetSkinFolder());
        printf("The main window's skin: %s.\n", skinParser.GetMainWindowSkin());

        // 遍历<MainWindow>节点下的所有子节点
        printf("\nLoop through all elements.\n");
```

```
        bool found = skinParser.FindFirstNode();
        while (found)
        {
                // 获取当前子节点的各个属性值
                CString btnName = skinParser.GetName();
                CString btnSkin = skinParser.GetSkin();
                long x, y, width, height;
                skinParser.GetPosition(x, y, width, height);
                printf("%s <name:%s, skin:%s, xpos:%d, ypos:%d, width:%d, height:%d>\n",
                        skinParser.GetNodeName(), btnName, btnSkin, x, y, width, height);

                // 指向下一个兄弟节点
                found = skinParser.FindNextNode();
        }
}
```

2.3.2.3 查找某个节点

2.3.2.2 节中已经介绍了 CXmlParserDemo 类的完整实现，并且介绍了遍历某一节点下所有子节点的方法。本节将要介绍另外一种基于节点的应用方法，即查找一个符合一定条件的节点，进而获取该节点的各个属性值。

其实查找一个特定的节点，也就是在遍历各个节点的基础上加上条件匹配而已。我们可以参考 2.3.2.2 节中 CXmlParserDemo::FindButton 函数的实现过程。如果我们想要查找<MainWindow>下的某个指定名字的<Button>节点，只需编写如下的代码：

```
// 创建一个 XML 解析器对象
CXmlParserDemo skinParser;

// 设置XML源文件
if (skinParser.SetXmlFile("ImageViewer.xml"))
{
        printf("\nTry to search a specified button: 'btn_Reload'.\n");
        // 查找名为“btn_Reload”的<Button>节点
        if (skinParser.FindButton("btn_Reload"))
        {
                // 如果查找成功，获取该节点的各个属性值
                printf("%s <name:%s, skin:%s, xpos:%d, ypos:%d>\n",
                        skinParser.GetNodeName(),
                        skinParser.GetName(),
                        skinParser.GetSkin(),
                        skinParser.GetXPos(),
                        skinParser.GetYPos());
        }
        else
        {
                // 没有找到符合要求的节点!
                printf("NOT FOUND!\n");
        }
```

}

2.3.2.4 实例程序：XMLParser

实例程序 XMLParser 位于本书配套代码包的 XML\XMLParser 目录下。这是一个通过 Visual Studio .NET 2003 向导生成的控制台程序，主要演示了 XML 文件中元素节点的遍历和查找。程序的运行结果如图 2.3。（省略）

第 3 章 基于 XML 的看图软件

3.1 UI 艺术设计师的工作

许多程序员都不喜欢做 UI 方面的工作。究其原因，很重要的一点就是：UI 开发中很大一部分时间需要花在 UI 元素的增减、布局调整上，而且这部分工作往往还是反反复复的，很是“无聊”！我们从第 1 阶段的原始软件开发过程中可见一斑。

然而，UI 是直接面对用户的；UI 是否漂亮、是否符合用户的使用习惯，决定着用户对这款软件的第一印象，决定着用户体验。如果没有设计良好的 UI，功能再强大的软件也很难得到用户的青睐。因此，UI 对于一款优秀的软件来说是非常重要的。事实上，专业的 UI 设计、特别是整体效果设计以及流程设计，已经远超出了一般程序员的能力范围。为了开发一款专业的应用软件，UI 艺术设计师的加入也就顺理成章了。他们从原先程序员那里接手了一大部分工作，主要包括：UI 的整体设计、UI 元素的分离和定位、最终生成一个 XML 文件。

3.1.1 UI 的整体设计

UI 艺术设计师的第一步工作是给软件的界面做一个整体设计。在这之前，设计师必须大致了解软件将要实现的功能，以决定界面上有哪些元素，界面整体上采用哪种风格。仍然以第 1 阶段提到的看图软件为例，图 3.1 展示了一种 UI 的整体设计。

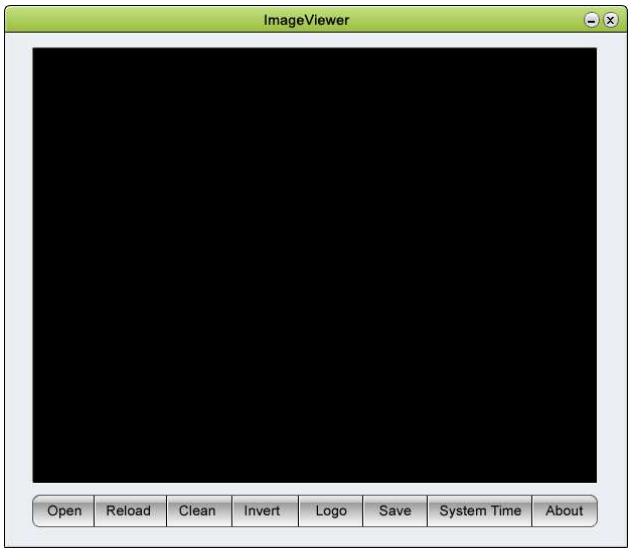

图 3.1 ImageViewer 软件界面的整体设计

3.1.2 UI 元素的分离和定位

当 UI 的整体设计完成之后，接下去的工作就是要将各个 UI 元素从整体设计中分离出来。而且有多少个 UI 元素，就要分离出来多少个图片文件。在软件运行时，这些分离出来的图片将作为各个 UI 元素的“皮肤”。

首先分离出来的是软件的主界面，如图 3.2，它将作为主窗口的背景图片。

图 3.2 ImageViewer 软件主窗口的背景图片

其次要将各个按钮切割出来，并且分别做出 4 种状态：正常时、鼠标掠过时、选中时和禁用时。以 Open 按钮为例，效果如图 3.3：

图 3.3 ImageViewer 软件 Open 按钮的 4 种状态

根据 UI 的整体设计效果图，UI 艺术设计师还要为切割出来的各个按钮定位。比如上述的 Open 按钮，经过计算，它应该出现在主界面的（29，506）这个坐标上。这些信息也应该通过某种方式记录下来（实际上就是记录在下一节将要介绍的 XML 文件中）。

3.1.3 生成一个 XML 文件

UI 艺术设计师最后的工作就是要生成一个 XML 文件。这个文件为每个 UI 元素都定义了必要的属性，比如该 UI 元素使用的皮肤文件是什么，位置在哪里等等。当然，在编写这个 XML 文件之前，我们首先要为这个 XML 定义一套规则，比如使用<MainWindow>来表示主窗口，用<Button>来表示一个按钮，表示皮肤、X 坐标、Y 坐标的属性名分别为 skin、xpos、ypos，等等。需要注意的是，这套 XML 规则必须预先跟程序员协商并达成一致！

根据图 3.1 的 UI 整体设计，UI 艺术设计师最终提交给程序员的 XML 文件内容如下（文件取名为 ImageViewer.xml）：

```
<?xml version="1.0" encoding="UTF-8"?>
<ImageViewer version="1.1">
	<MainWindow skin="main.jpg">
		<ImageBox xpos="20" ypos="50" width="582" height="446"/>
		<Button name="btn_Min" xpos="570" ypos="6" skin="btn_Min.jpg"/>
		<Button name="btn_Exit" xpos="590" ypos="6" skin="btn_Exit.jpg"/>
		<Button name="btn_Open" xpos="29" ypos="506" skin="btn_Open.jpg"/>
		<Button name="btn_Reload" xpos="91" ypos="506" skin="btn_Reload.jpg"/>
		<Button name="btn_Clean" xpos="163" ypos="506" skin="btn_Clean.jpg"/>
		<Button name="btn_Invert" xpos="228" ypos="506" skin="btn_Invert.jpg"/>
		<Button name="btn_Logo" xpos="294" ypos="506" skin="btn_Logo.jpg"/>
		<Button name="btn_SaveAs" xpos="358" ypos="506" skin="btn_Save.jpg"/>
		<Button name="btn_SystemTime" xpos="422" ypos="506" skin="btn_SystemTime.jpg"/>
		<Button name="btn_About" xpos="526" ypos="506" skin="btn_About.jpg"/>
	</MainWindow>
</ImageViewer>
```

3.2 程序员的工作

由于 UI 艺术设计师的介入，程序员的工作相比于原始开发阶段要轻松了不少。程序员不用再去关心 UI 元素的布局，这部分工作通过解析一个 XML 文件即可完成。

3.2.1 UI 元素设计

在这个阶段，程序员仍然决定着主界面上该有哪些 UI 元素，以及他们分别实现什么样的功能。因此，我们需要在 VC 的资源编辑器中设计如图 3.4 的软件界面。

大家可能注意到了，主界面上的各个窗口（包括按钮）的位置是凌乱的。但这一点都不要紧，因为各个窗口的位置信息已经记录在了 XML 文件中。程序正常运行起来时，各个窗口会根据 XML 中的预定义，重新调整各自的位置。这里需要解决的技术问题是，怎样才能使主界面上的各个窗口跟 XML 文件中的各项定义对应起来。

在 XML 文件中，我们看到每个按钮都有一个“name”属性，描述的是各个按钮具有唯一性的名字。因此，我们需要在 VC 的资源编辑器中为每个按钮设置一个名字，而且这些名字必须跟 XML 文件中的定义相对应。当程序运行时，我们可以通过 GetWindowText 函数首先获取窗口的名字，然后再以这个名字为关键字到 XML 文件中去检索，进而获得这个窗口所使用的皮肤文件，窗口所在位置的 X 坐标、Y 坐标等信息。

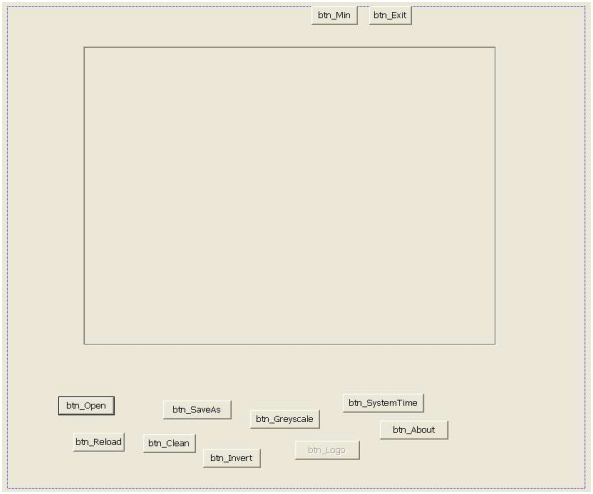

图 3.4 在 VC 的资源编辑器中设计软件界面

3.2.2 查询 XML 文件

如何解析 XML 文件？我们已经在 2.3 节中介绍过了。在 XML 文件中查询一个特定的节点，在 2.3.2.3 节中也已给出了具体的实现。跟那一节中的 CXmlParserDemo 类非常相似，我们这里也需要创建一个 CUISkinXmlReader 类，专么用于读取 XML 文件中关于 UI 的各项配置信息。

CUISkinXmlReader 类实现的主要功能是：获取主窗口使用的是哪个皮肤文件；获取主窗口中各个按钮（以按钮的名字为关键字进行检索）所使用的皮肤文件以及按钮所在位置的 X、Y 坐标。另外需要注意，CUISkinXmlReader 类的对象被实现为了一个“单件”（所谓单件，简单来说就是系统中最多只存在一个 CUISkinXmlReader 类的实例；各个需要读取 XML 信息的对象共用这个单件实例）。

CUISkinXmlReader 类的主要实现如下：**（此部分省略，请下载、查看本书配套代码）**

3.2.3 支持皮肤的 UI 类

为了让窗口支持皮肤，需要开发自定义的窗口类，这些类的继承结构参见图 3.5：

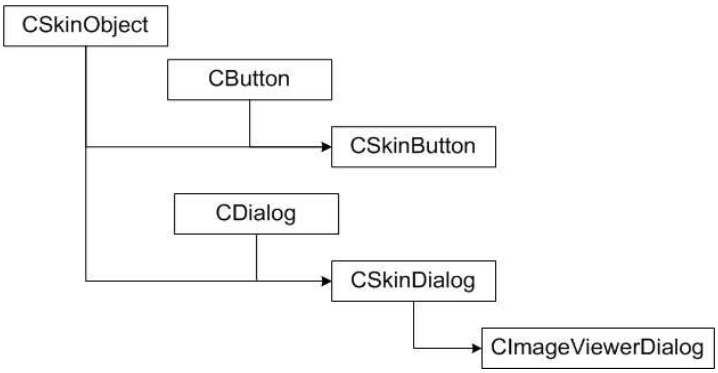

图 3.5 支持皮肤的 UI 类结构

其中，CSkinObject 类主要负责皮肤图片文件的读取（生成一个 GDI 位图以便显示到窗口区域中），CSkinButton 是一个支持皮肤的按钮类，CSkinDialog 是一个支持皮肤的对话框类。并且 CSkinButton 类和 CSkinDialog 类都从 CSkinObject 类继承而来

CSkinObject、CSkinButton、CSkinDialog 类的完整实现请读者自行参考实例程序 ImageViewer_Skinned。

3.3 实例程序：ImageViewer_Skinned

实例程序 ImageViewer_Skinned 位于本书配套代码包的 ImageViewer_Skinned 目录下。**这个程序是在 ImageViewer_Basic 的基础上修改过来的，请读者注意比较两者之间的区别。**软件运行起来界面如图 3.6。

第 4 章 可定制的看图软件

4.1 UI 由 XML 文件驱动

上一章介绍了一个基于 XML 的看图软件。相比于原始阶段的开发工作，程序员的负担已经减轻了不少。但是我们注意到，程序员其实并没有从“无聊的”UI 工作中彻底解脱出来——他们仍然要决定主界面上需要放置什么按钮，各个按钮分别实现什么样的功能，还要严格按照 UI 艺术设计师提交的 XML 文件为各个按钮命名……

接下去的问题是：有没有什么办法让程序员从上述“机械式的”工作中进一步解脱出来呢？答案是肯定的。要解决这个问题，关键的一点就是要转变 XML 文件的角色！上一章中，XML 文件仅仅是一个信息的“被动”提供者，仅在程序需要信息时作为一个被查询的对象出现；而这一章中，XML 文件的角色将从“被动”变为“主动”，它将决定主窗口

中需要哪些 UI 元素、以及分别实现什么样的功能。具体到程序对 XML 文件的操作方式上，也从“查询”转变为了“分析”。这个时候，我们可以说软件是由一个 XML 文件来驱动的——不同的 XML 文件定义了不同的软件 UI，从而也决定了软件最终展示给用户的不同的功能集。

接下去，我们就来看一下这种用户可定制的看图软件的实现方法。由于程序员不再需要在 VC 的资源编辑器中设计 UI 元素了，我们在资源编辑器中看到的只是如图 4.1 那样“空空的”主窗口：

图 4.1 在 VC 的资源编辑器中设计的 UI

既然现在不需要在主窗口上预先布置按钮了，在 XML 文件中定义的各个按钮也没必要再像第 3 章那样附带“name”属性了。另外，按钮需要增加的一个新的“function”属性，以指示这个按钮所实现的功能。（需要注意的是，“function”属性的取值需要跟程序员协商并达成一致！）编辑一个 XML 文件，内容如下（文件取名为 myViewer.xml）：

```
<?xml version="1.0" encoding="UTF-8"?>
<ImageViewer version="1.1">
	<MainWindow skin="main.jpg">
		<ImageBox xpos="20" ypos="85" width="582" height="446"/>
		<Button function="Minimize" xpos="565" ypos="6" skin="btn_Min.jpg"/>
		<Button function="Exit" xpos="585" ypos="6" skin="btn_Exit.jpg"/>
		<Button function="Open" xpos="21" ypos="40" skin="btn_Open.jpg"/>
		<Button function="About" xpos="83" ypos="40" skin="btn_About.jpg"/>
		<Hyperlink name="http://www.chery.cn" xpos="400" ypos="6" width="160" height="20"/>
	</MainWindow>
</ImageViewer>
```

上述 myViewer.xml 文件定义了主窗口中该有的各个 UI 元素，我们的程序应该有能力解析它、进而将各种不同类型的 UI 元素动态地创建起来（而不是像以前那样在 VC 的资源编辑器中预先设计好）。另外，动态创建起来的按钮还要像 XML 文件中定义的那样实现特定的功能（而不能像以前那样依赖不同的 ID 进行消息映射）。于是，我们需要一个新的按钮类：它能跟一个特定的表示所实现功能的字符串关联，并且在被点击时执行相应的功能操作。这个类就是 CCustomizedButton，它从 CSkinButton 类派生而来，实现如下**（此部分省略，请下载、查看本书配套代码）**：

```
//
// CCustomizedButton.h
//

#ifndef __H_CCustomizedButton__
#define __H_CCustomizedButton__

#include "CSkinButton.h"
#include "ActionDispatchIF.h"

class CCustomizedButton : public CSkinButton
{
public:
	CCustomizedButton(ActionDispatchIF* pDispatch, CString strFunc, bool bStatic =
false);
	~CCustomizedButton();

	// 创建窗体自身
	void CreateSelf(CWnd* pParent);
	// 当按钮被点击时，执行特定的功能操作
	virtual void TakeClickAction();

	//////// 按钮窗体被创建起来之前的一些设置 //////////
	// 设置按钮所使用的皮肤文件
	void SetSkinFile(CString const& inFile);
	// 设置按钮所在位置的X、Y坐标
	void SetSkinPosition(long xpos, long ypos);

private:
	ActionDispatchIF*	mDispatch;	// 按钮被点击时功能操作的执行者
	CString			mFuncName;	// 本按钮实现的功能（字符串形式）
	IMAGE_FUNCTION		mFunction;	// 本按钮实现的功能（整数形式）

	// 内部函数：将按钮实现的功能类型从字符串形式转化成整数形式
	IMAGE_FUNCTION ConvertToFunctionNumber(CString const & strFunc);
};

#endif // __H_CCustomizedButton__
```

其中，ActionDispatchIF 负责当按钮被点击时执行特定的功能操作，它是一个仅定义了接口的类，如下：

```
//
// ActionDispatchIF.h
//

#ifndef __H_ActionDispatchIF__
#define __H_ActionDispatchIF__
```

```
// 定义按钮可能执行的功能操作类型
typedef enum
{
        IF_Unknown        = 0,
        IF_Open           = 1,
        IF_Reload         = 2,
        IF_Clean          = 3,
        IF_Invert         = 4,
        IF_Greyscale      = 5,
        IF_Logo           = 6,
        IF_Save           = 7,
        IF_SystemTime     = 8,
        IF_About          = 9,
        IF_Minimize       = 10,
        IF_Exit           = 11
} IMAGE_FUNCTION;

class ActionDispatchIF
{
public:
        ActionDispatchIF() {};
        ~ActionDispatchIF() {};

        // 执行功能操作
        virtual bool Invoke(IMAGE_FUNCTION funcId) = 0;
};

#endif // __H_ActionDispatchIF__
```

ActionDispatchIF 接口最终是由主窗口对象来实现的，参考如下：

```
//
// ImageViewerDlg.cpp
//

// Invoke - 执行特定的按钮功能操作（该函数由主窗口类提供实现）
// 参数：funcId - 待执行的功能类型
// 返回值：true，成功；false，失败
bool CImageViewerDlg::Invoke(IMAGE_FUNCTION funcId)
{
        switch (funcId)
        {
        case IF_Minimize:
                OnBnClickedButtonMin();
                break;
        case IF_Exit:
                OnBnClickedButtonExit();
                break;
        case IF_Open:
                OnBnClickedButtonOpen();
                break;
        case IF_About:
                OnBnClickedButtonAbout();
```

```
            break;
        case IF_Invert:
            OnBnClickedButtonInvert();
            break;
        case IF_Logo:
            OnBnClickedButtonOverlayLogo();
            break;
        case IF_SystemTime:
            OnBnClickedButtonOverlaySystime();
            break;
        case IF_Clean:
            OnBnClickedButtonClean();
            break;
        case IF_Reload:
            OnBnClickedButtonReload();
            break;
        case IF_Save:
            OnBnClickedButtonSaveAs();
            break;
        default:
            return false;
        }

        return true;
}
```

另外，为了支持 XML 文件的分析，我们还需要创建一个 CUISkinXmlParser 类。这个类的实现与第 3 章的 CUISkinXmlReader 类非常相似，只不过对 XML 文件的使用方式上有所改变，所以更换了一个类名。主窗口类中对 XML 文件的分析以及各个 UI 元素的创建过程参考如下：

```
//
// ImageViewerDlg.cpp
//

// 分析XML文件内容，根据XML文件中的定义动态地创建各个UI元素
void CImageViewerDlg::ParseUISkinXml(void)
{
        // 获取XML解析器对象指针（这是个单件！）
        CUISkinXmlParser* pParser = CUISkinXmlParser::GetInstance();

        // 定位到<MainWindow>下的第一个子节点
        bool found = pParser->FindFirstNode();
        while (found)
        {
            // 判断节点类型
            // 根据不同的节点类型，以创建不同的UI元素
            switch (pParser->GetNodeTypeNumber())
            {
            case NT_ImageBox: // 图像显示窗口
                {
                long x, y, width, height;
```

```
                pParser->GetPosition(x, y, width, height);
                CRect rect(x, y, x+width, y+height);
                mPictureWnd.Create("", WS_CHILD|WS_BORDER|WS_VISIBLE|SS_BITMAP, rect, this, 1234);
                }
                break;

        case NT_Button: // 按钮
                {
                CCustomizedButton* pButton = new CCustomizedButton(this, pParser->GetFunction());
                pButton->SetSkinFile(pParser->GetSkin());
                pButton->SetSkinPosition(pParser->GetXPos(), pParser->GetYPos());
                pButton->CreateSelf(this);
                mButtons.push_back(pButton);
                }
                break;

        case NT_Static: // 静态窗口
                break;

        case NT_Hyperlink: // 超级链接
                {
                SAFE_DELETE(mpHyperlink);
                mpHyperlink = new CEasyHyperlink();

                RECT rect;
                rect.left   = pParser->GetXPos();
                rect.top    = pParser->GetYPos();
                rect.right  = rect.left + pParser->GetWidth();
                rect.bottom = rect.top + pParser->GetHeight();
                mpHyperlink->create(rect, pParser->GetName(), this->GetSafeHwnd());
                }
                break;

        default:
                break;
        }

        // 指向下一个节点
        found = pParser->FindNextNode();
    }
}
```

4.2 实例程序：ImageViewer_Customized

实例程序 ImageViewer_Customized 位于本书配套代码包的 ImageViewer_Customized 目录下。**这个程序是在 ImageViewer_Skinned 的基础上修改过来的，请读者注意比较两者之间的区别。**软件运行起来界面如图 4.2：

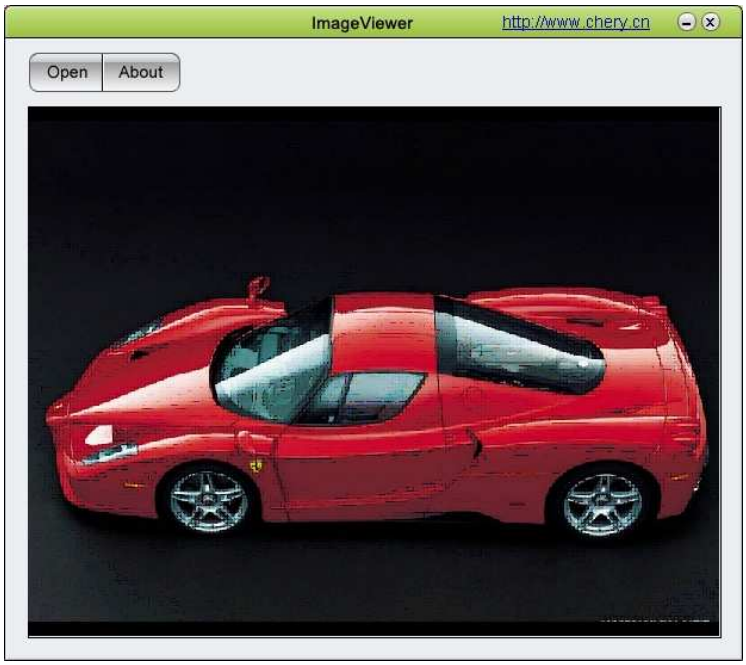

图 4.2 ImageViewer_Customized 的一种定制界面

既然这是一个用户可定制的看图软件，我们可以如下修改 myViewer.xml 文件的内容，以获得一个不同功能集的看图软件：

```
<?xml version="1.0" encoding="UTF-8"?>
<ImageViewer version="1.1">
	<MainWindow skin="main.jpg">
		<ImageBox xpos="20" ypos="85" width="582" height="446"/>
		<Button function="Minimize" xpos="565" ypos="6" skin="btn_Min.jpg"/>
		<Button function="Exit" xpos="585" ypos="6" skin="btn_Exit.jpg"/>
		<Button function="Open" xpos="21" ypos="40" skin="btn_Open.jpg"/>
		<Button function="Invert" xpos="83" ypos="40" skin="btn_Invert.jpg"/>
		<Button function="Save" xpos="149" ypos="40" skin="btn_Save.jpg"/>
		<Button function="About" xpos="213" ypos="40" skin="btn_About.jpg"/>
		<Hyperlink name="http://www.chery.cn" xpos="400" ypos="6" width="160" height="20"/>
	</MainWindow>
</ImageViewer>
```

再次运行 ImageViewer_Customized，结果如图 4.3：

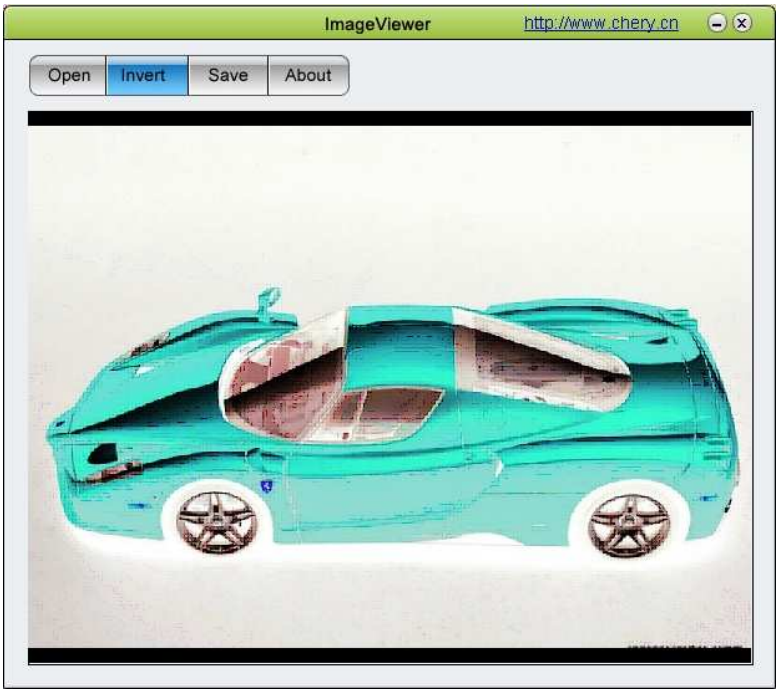

图 4.3 ImageViewer_Customized 的另一种定制界面

第 3 阶段　脚本驱动的软件开发

高度分工是现代社会的一个显著特征。分工造成了领域的细化。在每个细分的领域中，人们更容易学以致用、学有所成，长期积累下来的经验可以帮助他们把所属领域内的事情做得更快、更好。各个领域之间同时又是相互协作的，并且有机地整合在一起，共同构成了一个和谐的、高效率运作的社会。

软件产品的开发无疑也是一种社会活动。高效率的软件开发过程当然也要建立在高度分工与协作的基础之上。从基于皮肤的看图软件的开发过程，我们已经看到了程序员与 UI 艺术设计师之间的分工和协作。如果再发展到本书后续章节将要介绍的脚本驱动的看图软件，我们会看到另外一种角色——JavaScript 程序员——的加入。于是，C++程序员、JavaScript 程序员、UI 艺术设计师之间的联系将变得更为密切（当然，根据不同的专长以及在产品开发过程中承担的不同任务，开发人员之间一般会有更细的分工）。虽然引入这种脚本驱动的软件开发模式的初衷，是对用户可定制的软件的一种增强，但它也恰恰顺应了软件开发追求不断细化分工的一种发展趋势。

由于本书介绍的脚本主要是指 JavaScript，下面我们就先从 JavaScript 的解析说起。

第 5 章 JavaScript 解析

5.1 JavaScript 简介

JavaScript 是一种轻型的、解释型的程序设计语言。这种语言最初是由 Netscape 公司发明的，并且大量应用在客户端网页中；它一开始叫 LiveScript，后来因 Java 语言的风靡才被改名为 JavaScript（注意：JavaScript 与 Java 之间并没有什么“血缘”关系）。在句法构成上，JavaScript 的核心语法与 C/C++、Java 非常相似。因此，对于一名 C/C++或 Java 程序员来说，学习 JavaScript 的过程将会非常轻松！

JavaScript 已经发展了好多年，其语言核心已经相当成熟。Netscape 公司发布了该语言的多个版本。微软公司也发布了 JavaScript 语言的相似版本，名为 JScript。另外，ECMA 组织发布了 3 个版本的 ECMA－262 标准，对 JavaScript 语言进行了标准化，并且为该语言取了一个新的名字，叫作 ECMAScript。

通常来说，脚本语言是面向非程序员的，因此要求它比较简单，要让程序设计新手很容易地就能使用，并且按部就班地完成程序设计任务。JavaScript 是一种脚本语言。然而，在作为脚本语言简单的外表之下，JavaScript 却是一种具有丰富特性的程序设计语言；它和其他所有语言一样复杂，甚至比某些语言还要复杂得多。

（对 JavaScript 语言的介绍内容省略）

5.2 Active Scripting 技术

Active Scripting 是微软公司提供的一种脚本解析技术，是 ActiveX 技术的一个重要组成部分。它支持解析两种脚本语言：VBScript 和 JScript，并且已经在很多专业软件中得到了广泛的应用，其中就包括微软公司自己的 Windows 操作系统、以及 Office 系列办公软件。有了 Active Scripting 技术，我们也可以让自己开发的应用程序来支持脚本解析，使应用程序的部分功能为特定的脚本语言所控制、与脚本程序实现互动，以此大幅度地提升我们的软件产品在用户可定制性、可交互性方面的表现。

接下去，本书将重点介绍 Active Scripting 的基本结构和实现原理，并以一个实例程序来演示 Active Scripting 的应用方法。通过这部分内容的学习，读者将基本掌握在自己开发的应用程序中支持脚本解析的技能。

5.2.1 基本原理

Active Scripting 技术的应用涉及到两个主要的部分：**脚本引擎**（Scripting Engine）和**宿主程序**（Scripting Host）。其中，脚本引擎以组件的方式提供服务，它封装了比较底层的、很细节化的脚本程序处理功能，包括语法分析、语句执行等等；并且开放了一些必要的接口，使得使用它的宿主程序可以很方便地与它进行交互。而宿主程序就是我们要开发的应用程序，它是脚本引擎的客户（或者说使用者），所做的工作包括：

- 创建并管理脚本引擎的实例对象。
- 实现一些特定的接口供脚本引擎回调（脚本引擎通过这些回调接口从宿主程序获取类型信息、通知脚本引擎的状态改变、通知脚本解析时发生的错误等等）。
- 装载脚本程序，并把它们传递给脚本引擎进行解析。

宿主程序、脚本引擎和脚本文件三者之间的关系如图 5.1：

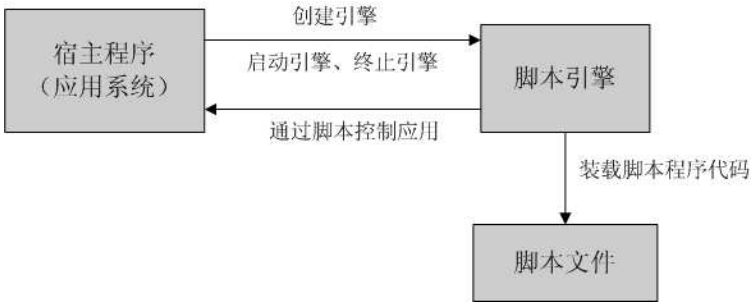

图 5.1 宿主程序、脚本引擎和脚本文件三者之间的关系

更具体来说，使用 Active Scripting 进行脚本解析的整个协作过程典型如图 5.2：

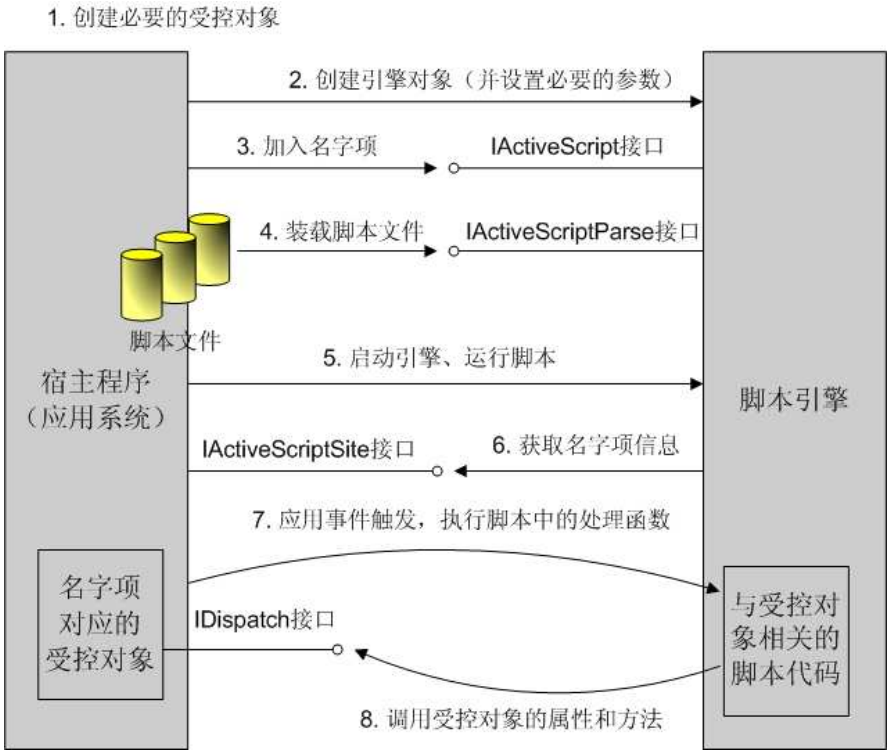

图 5.2 宿主程序与脚本引擎的协作过程

图 5.2 给出了 8 个步骤，下面我们来逐一介绍：

1) 创建必要的**受控对象**。这里所谓的受控对象，是应用程序开放给脚本、可以让脚本访问或控制的一些对象。这些对象往往是跟一些名字项相关联的。在 Active Scripting 的实现中，这些对象都必须支持 COM 自动化，也就是说这些对象上至少都实现了一个 IDispatch 接口。
2) 创建脚本引擎对象。Active Scripting 同时提供了 VBScript 引擎和 JScript 引擎。这些引擎都以 COM 组件的形式提供服务，它们有着不同的 CLSID 和 ProgID（ProgID 分别为“VBScript”和“JScript”）。应用程序应该根据实际需要解析的脚本语言类型，创建对应的引擎对象。
3) 加入名字项（Named Item）。宿主程序应该为受控对象选择合适的名字，并把这个名字告诉脚本引擎。当引擎在进行脚本解析时，如果碰到这些名字，它就会“询问”宿主程序它们分别代表哪些对象，继而对这些对象进行指定的操作。
4) 装载脚本文件。如果脚本程序保存在一个文件中，则宿主程序应该将文件内容先读出来，然后通过脚本引擎开放的一个接口，将脚本程序传递给引擎（准备解析但不马上进行，除非引擎已经处于连接状态）。
5) 启动引擎、运行脚本。将脚本引擎置于连接状态（也就是可执行脚本的状态），之后引擎便开始运行脚本程序了。
6) 脚本引擎从宿主程序获取名字项信息。如同 3）中的描述一样，当引擎在解析脚本时碰到一个名字项时，它会通过宿主程序的接口去询问关于这个名字项的相关信息。
7) 应用程序中的某个事件发生了，需要执行脚本中对应的事件处理函数。在这之前，一般是脚本程序首先将它感兴趣的事件跟特定的脚本函数关联起来，并且通过某个受控对象开放的接口进行注册。当应用程序的某个事件发生时，宿主程序才能根据注册信息去执行对应的脚本处理函数。
8) 脚本程序访问受控对象的属性和方法。这些操作典型的情况发生在事件的脚本处理函数中。通过这种对受控对象的访问，脚本程序可以改变宿主程序的一些特性、状态，乃至影响宿主程序的行为。

Active Scripting 是建立在 COM 的基础之上的，而且在实现时采用了自动化技术。在脚本解析时，脚本引擎将脚本程序中的对象都实例化为一个一个的自动化对象（即实现了 IDispatch 接口的 COM 对象）。因此应用程序开放给脚本程序的受控对象，也必须实现成为自动化对象，才能被脚本引擎所接受、进而被脚本程序所访问和控制。

脚本引擎开放了一系列的接口供宿主程序使用，这些接口包括：IActiveScript、IActiveScriptParse 等。其中，IActiveScript 接口主要负责引擎对象上的参数设置（最重要的有 AddNamedItem，用以添加名字项）、状态切换（使用 SetScriptState 函数）、以及跟引擎相关的线程操作；IActiveScriptParse 接口主要负责接收待解析的脚本程序。

表 5.5 脚本引擎上开放的 IActiveScript 接口

IActiveScript 接口函数	描述
HRESULT SetScriptSite(IActiveScriptSite *pScriptSite);	IActiveScriptSite 是宿主程序必须实现的、供脚本引擎回调的一个接口。通过 SetScriptSite 函数，可以将宿主程序实现的这个接口指针设置给脚本引擎，并且该函数必须在任何 IActiveScript 接口的其他函数之前被调用。
HRESULT GetScriptSite(REFIID iid,	获取跟脚本引擎关联的宿主对象。

void **ppvSiteObject);	
HRESULT SetScriptState(SCRIPTSTATE ss);	改变脚本引擎的状态。状态枚举类型定义如下： typedef enum tagSCRIPTSTATE { SCRIPTSTATE_UNINITIALIZED = 0, SCRIPTSTATE_INITIALIZED = 5, SCRIPTSTATE_STARTED = 1, SCRIPTSTATE_CONNECTED = 2, SCRIPTSTATE_DISCONNECTED = 3, SCRIPTSTATE_CLOSED = 4 } SCRIPTSTATE;
HRESULT GetScriptState(SCRIPTSTATE *pss);	获取脚本引擎当前的状态。
HRESULT Close(void);	关闭脚本引擎，这将导致引擎丢弃已经装载的脚本程序，释放内部对其他对象的接口引用，并最终进入 SCRIPTSTATE_CLOSED 状态。
HRESULT AddNamedItem(LPCOLESTR pstrName, DWORD dwFlags);	往脚本引擎的名字空间（Namespace）中加入名字项，而且所加的名字项应该是根级的（或者说顶级的），比如为了正确解析脚本代码 windows.buttons[0].enable()，加入的名字项只需“windows”即可。
HRESULT AddTypeLib(REFGUID guidTypeLib, DWORD dwMaj, DWORD dwMin, DWORD dwFlags);	为脚本的名字空间添加类型库支持，类似于 C/C++的 #include xxx，作用是：进行一些类型、常量的预定义。
HRESULT GetScriptDispatch(LPCOLESTR pstrItemName IDispatch **ppdisp);	获取当前正被执行的脚本的 IDispatch 接口（通过这个接口，应用程序可以访问脚本中的变量或者执行脚本中的函数）。
HRESULT GetCurrentScriptThreadID(SCRIPTTHREADID *pstidThread);	获取当前执行脚本的线程的 ID。SCRIPTTHREADID 类型有两个取值——SCRIPTTHREADID_BASE：基础线程，即创建脚本引擎对象、并且调用 IActiveScript::SetScriptSite 函数的那个线程；SCRIPTTHREADID_CURRENT，当前正在执行的线程。
HRESULT GetScriptThreadID(DWORD dwWin32ThreadID, SCRIPTTHREADID *pstidThread);	判断一个指定的线程（通过 Win32 线程 ID 指定）的类型是 SCRIPTTHREADID_BASE 还是 SCRIPTTHREADID_CURRENT。
HRESULT GetScriptThreadState(SCRIPTTHREADID stidThread, SCRIPTTHREADSTATE *pstsState);	获取指定线程的状态。状态枚举类型的定义如下： typedef enum tagSCRIPTTHREADSTATE { SCRIPTTHREADSTATE_NOTINSCRIPT = 0, SCRIPTTHREADSTATE_RUNNING = 1 } SCRIPTTHREADSTATE;
HRESULT InterruptScriptThread(SCRIPTTHREADID stidThread,	将指定的线程中止。参数 stidThread 可以取值为 SCRIPTTHREADID_ALL、SCRIPTTHREADID_BASE 或

const EXCEPINFO *pexcepinfo, DWORD dwFlags);	SCRIPTTHREADID_CURRENT。
HRESULT Clone(IActiveScript **ppscript);	克隆出一个新的脚本引擎对象。

表 5.6 脚本引擎上开放的 IActiveScriptParse 接口

IActiveScriptParse 接口函数	描述
HRESULT InitNew(void);	解析脚本之前做的初始化。在使用脚本引擎之前，必须首先调用下列 3 个函数中的其中一个：IPersist*::Load、IPersist*::InitNew、IActiveScriptParse::InitNew
HRESULT AddScriptlet(LPCOLESTR pstrDefaultName, LPCOLESTR pstrCode, LPCOLESTR pstrItemName, LPCOLESTR pstrSubItemName, LPCOLESTR pstrEventName, LPCOLESTR pstrDelimiter, DWORD dwSourceContextCookie, ULONG ulStartingLineNumber, DWORD dwFlags, BSTR *pbstrName, EXCEPINFO *pexcepinfo);	将脚本程序片断添加给脚本引擎进行解析。
HRESULT ParseScriptText(LPCOLESTR pstrCode, LPCOLESTR pstrItemName, IUnknown *punkContext, LPCOLESTR pstrDelimiter, DWORD dwSourceContextCookie, ULONG ulStartingLineNumber, DWORD dwFlags, VARIANT *pvarResult, EXCEPINFO *pexcepinfo);	将脚本程序传递给脚本引擎进行解析。

宿主程序也必须实现一些接口，才能实现与脚本引擎的协作，这些接口包括：IActiveScriptSite、IActiveScriptSiteWindow、IActiveScriptSiteDebug 等。其中，IActiveScriptSite 接口是必须实现的，脚本引擎通过回调这个接口从宿主程序获取类型信息、通知脚本引擎的状态改变和脚本解析时发生的错误；IActiveScriptSiteWindow 接口是可选的，如果宿主程序实现了用户界面的话则可以考虑同时实现这个接口；IActiveScriptSiteDebug 接口也是可选的，只有当用户想要调试脚本程序时才要实现它。

表 5.7 宿主程序上实现的 IActiveScriptSite 接口

IActiveScriptSite 接口函数	描述
HRESULT GetLCID(LCID *plcid);	脚本引擎用以获取本地化标识。
HRESULT IActiveScriptSite::GetItemInfo(LPCOLESTR pstrName, DWORD dwReturnMask, IUnknown **ppunkItem, ITypeInfo **ppTypeInfo);	脚本引擎用以获取特定名字项相关的信息，如受控对象指针，或类型信息。
HRESULT GetDocVersionString(BSTR *pbstrVersionString);	脚本引擎用以获取宿主程序定义的、用来标识当前文档的一个字符串。
HRESULT OnScriptTerminate(VARIANT *pvarResult, EXCEPINFO *pexcepinfo);	脚本引擎用以通知宿主程序：脚本执行完成了。
HRESULT OnStateChange(SCRIPTSTATE ssScriptState);	脚本引擎用以通知宿主程序：脚本引擎的状态改变了。
HRESULT OnScriptError(IActiveScriptError *pase);	脚本引擎用以通知宿主程序：脚本解析的时候发生了错误。
HRESULT OnEnterScript(void);	脚本引擎用以通知宿主程序：脚本引擎开始执行脚本代码了。
HRESULT OnLeaveScript(void);	脚本引擎用以通知宿主程序：脚本引擎完成了脚本代码的执行。

表 5.8 宿主程序上实现的 IActiveScriptSiteWindow 接口

IActiveScriptSiteWindow 接口函数	描述
HRESULT GetWindow(HWND *phwnd);	获取宿主程序中的一个窗口句柄，脚本引擎弹出来的窗口将以这个窗口为父窗口。
HRESULT EnableModeless(BOOL fEnable);	控制主窗口的模态属性。

表 5.9 宿主程序上实现的 IActiveScriptSiteDebug 接口

IActiveScriptSiteDebug 接口函数	描述
HRESULT GetDocumentContextFromPosition(DWORD_PTR dwSourceContext, ULONG uCharacterOffset, ULONG uNumChars, IDebugDocumentContext** ppsc);	获取脚本文档的上下文。

HRESULT GetApplication(IDebugApplication** ppda);	获取跟当前宿主程序关联的调式应用：一个 IDebugApplication 对象指针。
HRESULT GetRootApplicationNode(IDebugApplicationNode** ppdanRoot);	获取当前脚本调试应用的、需要加入到脚本文档中的根节点。
HRESULT OnScriptErrorDebug(IActiveScriptErrorDebug* pErrorDebug, BOOL* pfEnterDebugger, BOOL* pfCallOnScriptErrorWhenContinuing);	配置当脚本解析发生错误时的调试行为。

为了更好地理解和应用 Active Scripting 技术，我们有必要对 COM 自动化技术进行更进一步的了解。这就是本书接下去将要介绍的内容。

5.2.2 COM 自动化

自动化（Automation）是 COM 的一种特殊应用。自动化技术的核心是 IDispatch 接口。大家已经知道，每个 COM 对象都要求实现一个 IUnknown 接口，而自动化对象除了要实现 IUnknown 接口外，还必须同时实现一个 IDispatch 接口。

说起自动化技术的产生和发展，Visual Basic 和 VBA（或 VBScript）真是功不可没！因为虽然 COM 提供的规范很灵活，功能也很强大，并且具有很好的扩展性，实现 COM 对象也可以做到与语言无关。但事实上，COM 的语言无关性却很受限制。从 COM 接口的定义以及 COM 库提供的 API 函数可以看到，它们所使用的一些数据类型在一些弱类型的高级语言（如 Visual Basic）或脚本语言（如 VBScript、JavaScript）中是很难表达的。这就是说，如果用 C/C++开发了一个 COM 组件，在 VB 程序或者脚本程序中却未必能够被方便地使用。这就成了一个很大的问题！因为 VB 拥有着一个庞大的开发者群体，而 VBA 也已经成为了微软公司的大多数应用软件的扩展标准（这些软件包括 Microsoft Word、Excel、Outlook 等等）。为了解决这个问题，自动化技术便应运而生了！自动化技术在 C/C++程序与其他一些弱类型语言编写的程序之间搭起了一座沟通的桥梁。

我们已经知道，每个自动化对象都必须实现 IDispatch 接口。方法（Method）和属性（Property）是自动化对象的两个基本特性：方法是自动化对象提供的功能服务，而属性是自动化对象的数据特征。在自动化对象的实现中，方法和属性都有一个“符号化”的名字。客户程序通过这些名字以及自动化对象上的 IDispatch 接口，就可以访问到这个对象上的方法或属性（首先调用 IDispatch::GetIDsOfNames 函数获得名字对应的数字 ID，然后以这个 ID 为参数调用 IDispatch::Invoke 函数）；这比像在 C/C++程序中那样，必须通过 vtable 才能访问到 COM 对象的成员函数要方便了许多。当然，对自动化对象的这个访问过程一般还需要类型库的支持，这是我们下一节将要介绍的内容。

自动化技术对于数据类型的处理也有一些技巧，它使用了一个“万能的”数据结构——VARIANT。这个数据结构用一个域（即 VARTYPE vt;）来表示数据的确切类型，如短整型、长整型、浮点型、布尔型、字符串型等，再用另外一个巨大的联合体来记录对应的数据值。这就为 C/C++中繁杂的数据类型提供了一个统一的表示方法，为与客户程序进行数据交换提供了极大的便利（在 IDispatch::Invoke 函数的参数中，pDispParams 参数中表示方法或属性参数的类型为 VARIANTARG 结构，pVarResult 参数的类型为 VARIANT 结构，它们都用到了这个万能的数据结构）。VARIANT 数据结构的定义如下：

```
typedef struct tagVARIANT VARIANT;
typedef struct tagVARIANT VARIANTARG;

struct tagVARIANT
   {
    union
       {
       struct __tagVARIANT
       {
          VARTYPE vt;               // 类型标识
          WORD wReserved1;          // 保留
          WORD wReserved2;          // 保留
          WORD wReserved3;          // 保留

   // 定义了一个巨大的联合体
   union {
      LONGLONG              llval;          // VT_I8
      LONG                  lVal;           // VT_I4
      BYTE                  bVal;           // VT_UI1
      SHORT                 iVal;           // VT_I2
      FLOAT                 fltVal;         // VT_R4
      DOUBLE                dblVal;         // VT_R8
      VARIANT_BOOL          boolVal;        // VT_BOOL
      _VARIANT_BOOL         bool;
      SCODE                 scode;          // VT_ERROR
      CY                    cyVal;          // VT_CY
      DATE                  date;           // VT_DATE
      BSTR                  bstrVal;        // VT_BSTR
      IUnknown              * punkVal;      // VT_UNKNOWN
      IDispatch             * pdispVal;     // VT_DISPATCH
      SAFEARRAY             * parray;       // VT_ARRAY|*
      BYTE                  * pbVal;        // VT_BYREF|VT_UI1
      SHORT                 * piVal;        // VT_BYREF|VT_I2
      LONG                  * plVal;        // VT_BYREF|VT_I4
      LONGLONG              * pllVal;       // VT_BYREF|VT_I8
      FLOAT                 * pfltVal;      // VT_BYREF|VT_R4
      DOUBLE                * pdblVal;      // VT_BYREF|VT_R8
      VARIANT_BOOL          * pboolVal;     // VT_BYREF|VT_BOOL
      _VARIANT_BOOL         * pbool;
      SCODE                 * pscode;       // VT_BYREF|VT_ERROR
      CY                    * pcyVal;       // VT_BYREF|VT_CY
      DATE                  * pdate;        // VT_BYREF|VT_DATE
      BSTR                  * pbstrVal;     // VT_BYREF|VT_BSTR
      IUnknown              ** ppunkVal;    // VT_BYREF|VT_UNKNOWN
      IDispatch             ** ppdispVal;   // VT_BYREF|VT_DISPATCH
      SAFEARRAY             ** pparray;     // VT_ARRAY|*
      VARIANT               * pvarVal;      // VT_BYREF|VT_VARIANT
      PVOID                 * byref;        // Generic ByRef
      CHAR                  cVal;           // VT_I1
```

```
      USHORT                  uiVal;            // VT_UI2
      ULONG                   ulVal;            // VT_UI4
      ULONGLONG               ullVal;           // VT_UI8
      INT                     intVal;           // VT_INT
      UINT                    uintVal;          // VT_UINT
      DECIMAL                 * pdecVal         // VT_BYREF|VT_DECIMAL
      CHAR                    * pcVal;          // VT_BYREF|VT_I1
      USHORT                  * puiVal;         // VT_BYREF|VT_UI2
      ULONG                   * pulVal;         // VT_BYREF|VT_UI4
      ULONGLONG               * pullVal;        // VT_BYREF|VT_UI8
      INT                     * pintVal;        // VT_BYREF|VT_INT
      UINT                    * puintVal;       // VT_BYREF|VT_UINT
      struct __tagBRECORD
      {
      PVOID                   pvRecord;
      IRecordInfo             *pRecInfo;
      }    __VARIANT_NAME_4;
   }    __VARIANT_NAME_3;
  }    __VARIANT_NAME_2;
  DECIMAL decVal;
}    __VARIANT_NAME_1;

   };
};
```

提示：当 VARIANT 类型的变量表示一个 NULL 值时，它的 vt 域值为 VT_NULL；如果这个变量表示一个未定义的值，它的 vt 域值为 VT_EMPTY。另外，系统还专门提供了几个函数来处理 VARIANT 类型的变量，常用的有：VariantInit、VariantClear、VariantCopy 等。

5.2.2.1 IDL 和类型库

我们曾经提到，自动化对象一般都需要类型库的支持。类型库记录了组件对象的类型信息，这些信息是它与外界进行交互所必需的，包括组件的 CLSID、它所支持接口的 IID、接口的各个成员函数、接口函数的参数和返回值类型等等。另外，类型信息中也可以包含自定义的数据类型，如 C 语言风格的结构、联合、枚举等。

类型信息是客户程序与组件对象之间进行通信的基础。不管通过什么方式，客户程序在访问组件对象之前必须得到基本的类型信息，比如说，客户程序只有知道了 CLSID 才能创建组件对象，只有知道了 IID 才能在对象上请求特定的接口指针。

为了生成类型库，我们首先要用 IDL（Interface Description Language，接口描述语言）来描述一个接口。为了进一步描述组件对象，微软公司扩充了 IDL，形成了 ODL（Object Description Language，对象描述语言）。通常来说，一个组件对象的类型信息包括每个接口的类型信息和对象的类型信息。其中，接口的类型信息使用 interface 或 dispinterface 关键字来描述，它的描述方法与 IDL 完全兼容；对象的类型信息使用 coclass 关键字来描述。有时候，我们的组件程序中可能实现了多个自动化对象，这时我们可以把所有对象放在一个 ODL 文件中，并用 library 关键字来描述库信息，包括类型库的 ID（即 LIBID）、类型库所使用的语言和版本等。在 ODL 文件中还可以使用 importlib 导入其他类型库信息。

作为演示，下面的 ODL 文件描述了一个 Mobile 对象的类型信息。

```
//
// Mobile.idl
//

import "oaidl.idl";
import "ocidl.idl";

[
	uuid(8F121977-5566-7788-9900-112233445566), // LIBID_MobileTypeLib
	version(1.0),
	helpstring("Mobile 1.0 Type Library")
]
library MobileTypeLib
{
	importlib("stdole32.tlb");
	importlib("stdole2.tlb");

	[
		object,
		uuid(1ECBE81A-B69C-4a02-924D-C1C2A1A2B1B2), // IID_IMobile
		dual
	]
	interface IMobile : IDispatch
	{
		[propget, helpstring("Return the type of the mobile.")]
		HRESULT type([out, retval] BSTR *pVal);
		[propput, helpstring("Set a type for the mobile.")]
		HRESULT type([in] BSTR newVal);

		[propget, helpstring("Return the color of the mobile.")]
		HRESULT color([out, retval] long *pVal);
		[propput, helpstring("Set a color for the mobile.")]
		HRESULT color([in] long newVal);

		[helpstring("Call the specified phone number.")]
		HRESULT Call([in] BSTR phoneNumber);
	}

	////////////////////////////////////////////////////////////////////
	[
		uuid(31273E15-1977-1988-2006-E6BBA4F18507), // CLSID_Mobile
		helpstring("Mobile COM class"),
		appobject
	]
	coclass Mobile
	{
		[default] dispinterface IMobile;
```

```
        }
}
```

上面给出的 ODL 文件中，GUID 标识符 **F121977-5566-7788-9900-112233445566** 标识了类型库。IMobile 接口继承自 IDispatch 接口，表明它是一个自动化接口。IMobile 接口描述了 Mobile 对象的属性（每个属性都可以有获取和设置两种方法）和方法，并且在 interface 关键字前面也指定了这个接口的 IID：**1ECBE81A-B69C-4a02-924D-C1C2A1A2B1B2**。coclass 关键字定义了自动化组件对象，包括 CLSID 以及该对象所支持的接口。由于一个自动化对象可能支持多个接口，所以一般使用 default 修饰符显式地指定一个缺省接口。

ODL 文件经过 MIDL 或 MkTypLib 工具编译之后，即可生成一个类型库文件，其扩展名为.tlb，同时还会产生一些.h 文件和.c 文件。那么，使用类型库到底有什么好处呢？通过类型库的支持，VB 或 VBA 可以很方便地浏览组件提供的属性和方法；这些语言还可以利用类型库来增强对属性和方法的访问，这是一个称作为早绑定（early binding）的过程。在使用 VC 编程的时候，IDE 环境的类向导也能够读入组件的类型库，并利用其中的类型信息自动产生相应的 C++代码，客户程序利用这些代码可以非常方便地“驱动”自动化组件。

使用类型库还有一个好处：我们在实现自动化对象的 IDispatch 接口时，可以利用类型库向客户程序提供类型信息，使我们能够从繁琐的类型处理工作中解脱出来。在后面的 5.2.2.4 节中讲述自动化对象实现时，我们将会更加清楚地看到这一点。

5.2.2.2 IDispatch 接口

IDispatch 接口有 4 个成员函数，它们的定义如下：

```
interface IDispatch : public IUnknown
{
public:
    virtual HRESULT STDMETHODCALLTYPE GetTypeInfoCount(
        /* [out] */ UINT *pctinfo) = 0;

    virtual HRESULT STDMETHODCALLTYPE GetTypeInfo(
        /* [in] */ UINT iTInfo,
        /* [in] */ LCID lcid,
        /* [out] */ ITypeInfo **ppTInfo) = 0;

    virtual HRESULT STDMETHODCALLTYPE GetIDsOfNames(
        /* [in] */ REFIID riid,
        /* [size_is][in] */ LPOLESTR *rgszNames,
        /* [in] */ UINT cNames,
        /* [in] */ LCID lcid,
        /* [size_is][out] */ DISPID *rgDispId) = 0;

    virtual HRESULT STDMETHODCALLTYPE Invoke(
        /* [in] */ DISPID dispIdMember,
        /* [in] */ REFIID riid,
        /* [in] */ LCID lcid,
```

```
        /* [in] */ WORD wFlags,
        /* [out][in] */ DISPPARAMS *pDispParams,
        /* [out] */ VARIANT *pVarResult,
        /* [out] */ EXCEPINFO *pExcepInfo,
        /* [out] */ UINT *puArgErr) = 0;
};
```

前两个成员函数 GetTypeInfoCount 和 GetTypeInfo 用于处理对象的类型信息。在 GetTypeInfoCount 函数的实现中，如果对象提供了类型信息的支持，则输出参数 pctinfo 的返回值为 1，否则就返回 0。因此客户程序在试图获取对象的类型信息之前，应该首先调用 GetTypeInfoCount 函数，并通过 pctinfo 的返回值来判断对象是否能够提供类型信息。GetTypeInfo 函数用于返回对象的类型信息：iTInfo 参数指定待获取的类型信息，0 表示获取当前自动化接口实现的类型信息；lcid 参数指定类型信息的本地化标识，因为一个自动化对象可以支持多种语言，方法和属性在不同的语言环境中可以使用不同的名字；ppTInfo 输出参数指向一个类型信息对象的 ITypeInfo 接口指针，以后客户程序可以通过这个 ITypeInfo 接口获取当前自动化接口的所有类型信息（这些信息也就是先前在 ODL 文件中描述的类型信息）。

成员函数 GetIDsOfNames 用于返回组件对象的方法或属性及其参数名字的 DISPID（一种数字形式的 ID）。IDispatch 接口使用 DISPID 来管理属性和方法（根据这个对象范围内具有唯一性的 ID 在内部进行分发）。参数 riid 保留为将来使用，客户程序必须指定为 IID_NULL；rgszNames 用于指定成员的名字，它是一个字符串数组，第一个字符串为方法或属性成员的名字，后续的字符串为成员的参数名字；cNames 参数指定了数组中名字的个数；lcid 参数指定一种本地化标识；rgDispId 输出参数是一个由客户程序分配的数组，每个数组元素包含 rgszNames 名字所对应的 DISPID。

自动化接口的 DISPID 是个整数，0 或者负数有特殊含义，如表 5.10 所示：

表 5.10 保留的 DISPID 及其含义

DISPID	值	描述
DISPID_VALUE	0	接口的缺省成员，如果在客户程序中不指定属性或方法，则用此缺省成员。
DISPID_UNKNOWN	-1	在 GetIDsOfNames 函数中返回此值表明相应的成员或参数名没有找到 DISPID。
DISPID_PROPERTYPUT	-3	在属性设置函数中指示一个接收值的参数。
DISPID_NEWENUM	-4	集合对象的_NewEnum 方法。
DISPID_EVALUATE	-5	对象的 Evaluate 方法，在脚本语言中可以用方括号[]来表示。
DISPID_CONSTRUCTOR	-6	表示具有与构造函数相同功能的方法。
DISPID_DESTRUCTOR	-7	标示具有与析构函数相同功能的方法。

成员函数 Invoke 是个关键函数。客户程序必须通过 Invoke 函数才能访问组件对象上的属性或者方法。可以这么说，Invoke 函数是自动化对象的命令翻译器和最终执行者。Invoke 函数的第一个参数 dispIdMember 给出了 DISPID，它将根据此 DISPID 执行相关的属性访问函数或者方法函数。riid 参数为保留参数，必须指定为 IID_NULL。lcid 参数指定一种本地化标识。wFlags 参数指示是访问属性还是调用方法，取值含义如下：

```
/* Flags for IDispatch::Invoke */
```

```
#define DISPATCH_METHOD         0x1         // 表示方法调用
#define DISPATCH_PROPERTYGET    0x2         // 表示属性取值
#define DISPATCH_PROPERTYPUT    0x4         // 表示属性设置
#define DISPATCH_PROPERTYPUTREF 0x8         // 表示通过引用方式设置属性
```

pDispParams 参数类型为 DISPPARAMS，它包括方法和属性调用的参数数组、参数的 DISPID 数组、数组中参数的个数等信息。pVarResult 输出参数保存返回值信息，如果客户程序不需要返回结果的话，pVarResult 可以由调用者指定为 NULL。pExcepInfo 输出参数指向一个包含异常信息的结构，如果客户程序不关心异常，也可以将它指定为 NULL；如果 Invoke 函数返回 DISP_E_EXCEPTION，则 pExcepInfo 参数中包含了有效的异常信息。最后一个输出参数 puArgErr 用于存放方法或属性调用时产生错误的参数索引值，当 Invoke 函数返回 DISP_E_TYPEMISMATCH 或 DISP_E_PARAMNOTFOUND 值时，puArgErr 包含第一个产生错误的参数索引。

由于自动化对象的所有方法和属性调用都通过 Invoke 函数来实现，并且它提供了管理属性和方法的分发 DISPID 机制以及一套描述参数和返回值的方法，使得运行时刻动态绑定属性和方法并进行参数类型检查成为可能。虽然 IDispatch 接口的运行效率比直接通过 vtable 进行调用要低一些，但其强大的功能以及灵活性足以弥补效率上的不足。（其实自动化技术还支持一种叫“双接口”的机制；通过双接口，C/C++客户程序可以在效率和方便性之间进行权衡。）

5.2.2.3 IDispatchEx 接口

使用 IDispatch 接口对于一般的应用来说已经足够了，但由于它所描述的类型信息是静态的（在 ODL 文件中已经固定下来了），类型信息在程序运行过程中不会也不能被改变。然而在一些脚本语言中，我们需要在运行时刻动态地提供类型信息，比如 VBScript、JavaScript 或 DHTML 对象模型等，这些语言要求一个更加灵活的接口。

于是就有了 IDispatchEx 接口。IDispatchEx 接口继承了 IDispatch 接口的所有成员函数，并增加了另外 8 个函数，其定义如下：

```
interface IDispatchEx : public IDispatch
{
public:
    virtual HRESULT STDMETHODCALLTYPE GetDispID(
        /* [in] */ BSTR bstrName,
        /* [in] */ DWORD grfdex,
        /* [out] */ DISPID *pid) = 0;

    virtual /* [local] */ HRESULT STDMETHODCALLTYPE InvokeEx(
        /* [in] */ DISPID id,
        /* [in] */ LCID lcid,
        /* [in] */ WORD wFlags,
        /* [in] */ DISPPARAMS *pdp,
        /* [out] */ VARIANT *pvarRes,
        /* [out] */ EXCEPINFO *pei,
        /* [unique][in] */ IServiceProvider *pspCaller) = 0;

    virtual HRESULT STDMETHODCALLTYPE DeleteMemberByName(
```

```
        /* [in] */ BSTR bstrName,
        /* [in] */ DWORD grfdex) = 0;

    virtual HRESULT STDMETHODCALLTYPE DeleteMemberByDispID(
        /* [in] */ DISPID id) = 0;

    virtual HRESULT STDMETHODCALLTYPE GetMemberProperties(
        /* [in] */ DISPID id,
        /* [in] */ DWORD grfdexFetch,
        /* [out] */ DWORD *pgrfdex) = 0;

    virtual HRESULT STDMETHODCALLTYPE GetMemberName(
        /* [in] */ DISPID id,
        /* [out] */ BSTR *pbstrName) = 0;

    virtual HRESULT STDMETHODCALLTYPE GetNextDispID(
        /* [in] */ DWORD grfdex,
        /* [in] */ DISPID id,
        /* [out] */ DISPID *pid) = 0;

    virtual HRESULT STDMETHODCALLTYPE GetNameSpaceParent(
        /* [out] */ IUnknown **ppunk) = 0;

};
```

我们这里不打算对 IDispatchEx 接口的各个成员函数逐一进行介绍，只就 IDispatchEx 接口对 IDispatch 接口的扩充作一个简单的说明：

- ✓ GetDispID 函数的作用与 GetIDsOfNames 类似，但它的参数可以指定 fdexNameEnsure 标志，使得如果指定的成员不存在，则允许创建新的成员。
- ✓ 通过 DeleteMemberByName 和 DeleteMemberByDispID 两个函数可以动态地删除对象的成员。
- ✓ 利用 fdexNameCaseSensitive 或 fdexNameCaseInsensitive 标志可以指定大小写敏感标志。
- ✓ 利用 fdexNameImplicit 标志能够以隐含的名字查找成员。
- ✓ 通过 GetNextDispID 函数可以对对象的 DISPID 进行枚举。
- ✓ 通过 GetMemberName 函数可以根据 DISPID 发过来查询其对应的名字。
- ✓ 通过 GetMemberProperties 函数可以获取对象成员的属性。
- ✓ InvokeEx 函数允许在方法调用时（使用 DISPATCH_METHOD 标志）传递“this”指针，其 DISPID 为 DISPID_THIS，且必须为第一个参数。
- ✓ InvokeEx 函数增加了 pspCaller 参数，允许对象从客户程序中获取某种功能服务。
- ✓ 允许支持名字空间概念的浏览器通过 GetNameSpaceParent 函数获得对象的名字空间父对象。

IDispatchEx 接口最主要的功能是增强了对成员的管理，突出了“动态”这个特性。但是在动态管理成员的基础上，我们必须保持对象的向后兼容性，也就是说仍然要保证 IDispatch 接口的特性。比如 IDispatch 接口总是假定成员或参数的 DISPID 在对象生存期

内为常数，不会发生变化，客户程序可以一次获取，以后多次使用。IDispatchEx 接口虽然允许增、删成员，但它必须保证成员与 DISPID 之间的映射关系保持不变。一旦一个成员被删除掉，那么它的 DISPID 也不能被重用，除非同名的成员又被创建。而且 IDispatch 或 IDispatchEx 接口的其他成员函数也必须能够接受此 DISPID，当这些函数发现对应成员已被删除之后，可以返回相应的错误码。

5.2.2.4 自动化对象实现

从上面的介绍我们已经了解到，自动化对象实际上用 Invoke 函数代理了属性和方法的访问，用 VARIANT 结构封装了它所支持的所有数据类型。从客户程序的角度来说，使用自动化对象非常方便，适合于 VB 或 VBScript、JavaScript 等脚本语言访问自动化对象。但站在自动化对象的角度来说，实现 IDispatch 接口的功能并不比在 C/C++语言中实现其他接口更为容易，因为 IDispatch 接口的 Invoke 函数要负责数据类型的转换处理，并且还必须提供类型信息的支持。不过，如果有类型库的支持，自动化对象的实现过程就会简单许多！下面，我们将通过一个实例程序来展示这种自动化对象的实现过程。

首先要用 ODL 语言来描述一个组件对象，如同 5.2.2.1 节介绍的那个例子一样，我们使用一个 Mobile.idl 文件来描述一个 Mobile 对象。这个对象具有“type”、“color”两个属性，并且拥有一个“Call”方法。定义好这个 Mobile.idl 文件之后，我们只需把它加入到 VC 的工程中去，VC 就会自动使用 MIDL 编译器去处理它，并且最终生成一个类型库文件。

在 VC 工程中，鼠标点击 Mobile.idl 文件，通过右键菜单可以查看到这个文件的属性，如图 5.3：

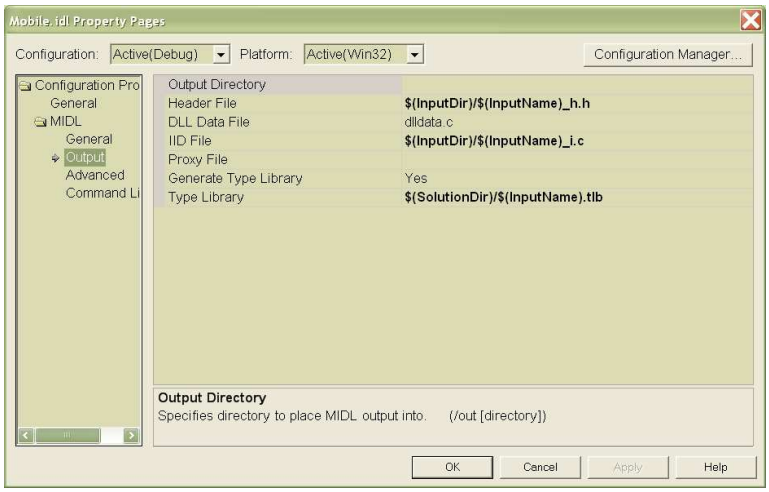

图 5.3 ODL 文件在 VC 中的编译选项

从图 5.3 中我们不难看出，Mobile.idl 文件经过 MIDL 编译之后不仅会生成一个类型库文件：Mobile.tlb，同时还会生成 Mobile_h.h 文件（用于定义自动化接口的各个成员函数）和 Mobile_i.c 文件（用于定义类型库的 LIBID、组件的 CLSID、以及各个接口的 IID）。值得注意的是，为了成功编译 Mobile_i.c 文件，我们需要在 VC 的预编译器中定义**_MIDL_USE_GUIDDEF_**这个宏（方法是：首先在 VC 中选中工程名，然后执行菜单命令：

Project | Properties，在随后弹出的窗口的 C/C++ | Preprocessor | Preprocessor Definitions 中加入这个宏），否则会出现如下类似的连接错误：

```
Mobile.obj : error LNK2001: unresolved external symbol _IID_IMobile
.\Tester_d.exe : fatal error LNK1120: 1 unresolved externals
```

接下去，我们就来实现 Mobile 对象。大家已经知道，作为自动化对象，它首先应该是一个 COM 对象，因此 Mobile 对象上必须要实现引用计数、必须要实现 IUnknown 接口的 3 个成员函数：QueryInterface、AddRef 和 Release。

```
//
// Mobile.h
//

#ifndef __H_Mobile__
#define __H_Mobile__

#include "Mobile_h.h"

class Mobile : public IMobile
{
public:
	Mobile();
	~Mobile();

	// --- IUnknown接口的各个成员函数 ---
	STDMETHODIMP QueryInterface(REFIID riid, void **ppvObject);
	STDMETHODIMP_(ULONG) AddRef();
	STDMETHODIMP_(ULONG) Release();

private:
	long		m_cRef;		// 引用计数器
};

#endif // __H_Mobile__


//
// Mobile.cpp
//

#include "stdafx.h"
#include "safe_defs.h"
#include "Mobile.h"

////////////////////////////////////////////////////////////////////////
// 构造函数
Mobile::Mobile()
	: m_cRef(0)
```

```
{
}

// 析构函数
Mobile::~Mobile()
{
}

// --- IUnknown接口的各个成员函数 ---
// 查询获取riid对应类型的接口指针
STDMETHODIMP Mobile::QueryInterface(REFIID riid, void **ppvObject)
{
	if (IID_IUnknown == riid)
	{
		*ppvObject = static_cast<IUnknown*> (this);
		AddRef();
		return S_OK;
	}
	else if (IID_IDispatch == riid)
	{
		*ppvObject = static_cast<IDispatch*> (this);
		AddRef();
		return S_OK;
	}
	else if (IID_IMobile == riid)
	{
		*ppvObject = static_cast<IMobile*> (this);
		AddRef();
		return S_OK;
	}

	*ppvObject = 0;
	return E_NOINTERFACE;
}

// 增加一个引用计数
STDMETHODIMP_(ULONG) Mobile::AddRef()
{
	return InterlockedIncrement(&m_cRef);
}

// 减少一个引用计数，并且当引用计数降到0的时候销毁自己
STDMETHODIMP_(ULONG) Mobile::Release()
{
	if (0 == InterlockedDecrement(&m_cRef))
	{
		delete this;
		return 0;
	}
```

```
        return m_cRef;
}
```

作为自动化对象，Mobile 对象还必须实现 IDispatch 接口的 4 个成员函数：GetTypeInfoCount、GetTypeInfo、GetIDsOfNames 和 Invoke。有了类型库的支持，这 4 个函数实现起来非常的轻松！

```
//
// Mobile.h
//

#ifndef __H_Mobile__
#define __H_Mobile__

#include "Mobile_h.h"

class Mobile : public IMobile
{
public:
        Mobile();
        ~Mobile();

        // --- IDispatch接口的各个成员函数 ---
        STDMETHODIMP GetTypeInfoCount(UINT * pctinfo);
        STDMETHODIMP GetTypeInfo(UINT itinfo, LCID /*lcid*/, ITypeInfo ** pptinfo);
        STDMETHODIMP GetIDsOfNames(REFIID /*riid*/,
                                OLECHAR  ** rgszNames,
                                UINT cNames,
                                LCID /*lcid*/,
                                DISPID * rgdispid);
        STDMETHODIMP Invoke(DISPID dispidMember,
                                REFIID riid,
                                LCID /*lcid*/,
                                WORD wFlags,
                                DISPPARAMS * pdispparams,
                                VARIANT * pvarResult,
                                EXCEPINFO * pexcepinfo,
                                UINT * puArgErr);

protected:
        ITypeInfo* GetTypeInfo();    // 获取类型对象指针

private:
        ITypeLib*      m_ptl;        // 类型库对象指针
};

#endif // __H_Mobile__
```

```
//
// Mobile.cpp
//

// --- IDispatch接口的各个成员函数 ---
// 获取该自动化对象实现的类型信息接口的数量
STDMETHODIMP Mobile::GetTypeInfoCount(UINT * pctinfo)
{
	*pctinfo = 1;
	return S_OK;
}

// 返回类型信息接口
STDMETHODIMP Mobile::GetTypeInfo(UINT itinfo, LCID /*lcid*/, ITypeInfo ** pptinfo)
{
	if (0 == pptinfo)
	{
		return E_POINTER;
	}

	if (0 != itinfo)
	{
		return TYPE_E_ELEMENTNOTFOUND;
	}

	// 通过查询类型库获取类型信息接口
	*pptinfo = GetTypeInfo();
	return (*pptinfo == 0) ? E_FAIL : S_OK;

}

// 查询名字对应的DISPID
STDMETHODIMP Mobile::GetIDsOfNames(REFIID /*riid*/,
				OLECHAR  ** rgszNames,
				UINT cNames,
				LCID /*lcid*/,
				DISPID * rgdispid)
{
	HRESULT hr = E_FAIL;

	// 通过查询类型库获取类型信息接口
	ITypeInfo* pTypeInfo = GetTypeInfo();
	if (pTypeInfo)
	{
		// 调用ITypeInfo接口的GetIDsOfNames函数
		hr = pTypeInfo->GetIDsOfNames(rgszNames, cNames, rgdispid);
		pTypeInfo->Release();
	}
	return hr;
```

```
}

// 对给定的DISPID实行属性访问或方法调用
STDMETHODIMP Mobile::Invoke(DISPID dispidMember,
                            REFIID riid,
                            LCID /*lcid*/,
                            WORD wFlags,
                            DISPPARAMS * pdispparams,
                            VARIANT * pvarResult,
                            EXCEPINFO * pexcepinfo,
                            UINT * puArgErr)
{
        if (IID_NULL != riid)
        {
                return DISP_E_UNKNOWNINTERFACE;
        }

        HRESULT hr = NOERROR;
        ITypeInfo* pTypeInfo = 0;

        try
        {
                // 通过查询类型库获取类型信息接口
                pTypeInfo = GetTypeInfo();
                if (pTypeInfo)
                {
                        // 调用ITypeInfo接口的Invoke函数
                        hr = pTypeInfo->Invoke((void*)this, dispidMember, wFlags,
                                pdispparams, pvarResult, pexcepinfo, puArgErr);
                }
        }
        catch (...)
        {
                hr = E_FAIL;
        }

        SAFE_RELEASE(pTypeInfo);
        return hr;
}

// 内部函数：通过查询类型库获取类型信息接口
ITypeInfo* Mobile::GetTypeInfo()
{
        HRESULT hr = NOERROR;
        if (m_ptl == 0)
        {
                // 装载类型库文件
                hr = ::LoadTypeLib(L"Mobile.tlb", &m_ptl);
        }
```

```
        // 从类型库文件中查询IMobile接口对应的类型信息
        ITypeInfo* pTypeInfo = 0;
        if (SUCCEEDED(hr) && m_ptl)
        {
                m_ptl->GetTypeInfoOfGuid(IID_IMobile, &pTypeInfo);
        }
        return pTypeInfo;
}
```

接下去，我们又该怎么来实现 Mobile 对象自身的“type”、“color”属性以及“Call”方法呢？而且，属性访问还有获取和设置之分。这时就要参考 Mobile_h.h 文件了！答案是：提取在 ODL 文件中定义的属性名，并且保持大小写一致，实现属性获取函数时在属性名前加 get_前缀，实现属性设置函数时在属性名前加 put_前缀；自动化接口中定义的方法只要简单映射过来就行了。

```
//
// Mobile.h
//

#ifndef __H_Mobile__
#define __H_Mobile__

#include "Mobile_h.h"

class Mobile : public IMobile
{
public:
        Mobile();
        ~Mobile();

        // --- IMobile接口的各个成员函数 ---
        STDMETHODIMP get_type(BSTR *pVal);          // 获取“type”属性
        STDMETHODIMP put_type(BSTR newVal);         // 设置“type”属性
        STDMETHODIMP get_color(long *pVal);         // 获取“color”属性
        STDMETHODIMP put_color(long newVal);        // 设置“color”属性
        STDMETHODIMP Call(BSTR phoneNumber);        // “Call”方法

private:
        BSTR            m_Type;         // “type”属性
        long            m_Color;        // “color”属性
};

#endif // __H_Mobile__


//
// Mobile.cpp
//
```

```
// --- IMobile接口的各个成员函数 ---
// 获取“type”属性
STDMETHODIMP Mobile::get_type(BSTR *pVal)
{
	if (m_Type)
	{
		*pVal = ::SysAllocString(m_Type);
	}
	else
	{
		*pVal = ::SysAllocString(L"unknown");
	}
	return S_OK;
}

// 设置“type”属性
STDMETHODIMP Mobile::put_type(BSTR newVal)
{
	if (m_Type)
	{
		::SysFreeString(m_Type);
		m_Type = 0;
	}
	m_Type = ::SysAllocString(newVal);
	return S_OK;
}

// 获取“color”属性
STDMETHODIMP Mobile::get_color(long *pVal)
{
	*pVal = m_Color;
	return S_OK;
}

// 设置“color”属性
STDMETHODIMP Mobile::put_color(long newVal)
{
	m_Color = newVal;
	return S_OK;
}

// 调用“Call”方法
STDMETHODIMP Mobile::Call(BSTR phoneNumber)
{
#ifdef _DEBUG
	wchar_t wszMsg[100];
	swprintf(wszMsg, L"You're calling %s ...", phoneNumber);
	::MessageBoxW(0, wszMsg, L"COM Automation Tester", MB_OK);
```

```
#endif

        return S_OK;
}
```

Mobile 对象实现好之后，我们就可以编写一些测试代码，来演示如何进行自动化对象上的属性访问和方法调用了。为了方便 C/C++客户程序对 IDispatch 接口的使用，我们封装了一个 DispatchHelper 类。DispatchHelper 类的定义和实现**（此部分省略，请下载、查看本书配套代码）**如下：

```
//
// DispatchHelper.h
//

#ifndef __DispatchHelper_H__
#define __DispatchHelper_H__

class DispatchHelper
{
public:
        DispatchHelper(IDispatch* pDisp = 0);
        ~DispatchHelper();

        // 创建参数数组
        bool CreateParameters(int paramCount);
        // 删除参数数组
        void DeleteParameters();

        // 为参数数组中指定索引的参数赋值
        bool AddParameter(UINT index, VARIANT* value);
        // AddParameter函数的各个重载形式
        bool AddParameter(UINT index, BSTR value);
        bool AddParameter(UINT index, INT value);
        bool AddParameter(UINT index, UINT value);
        bool AddParameter(UINT index, IDispatch* pDisp);
        bool AddNullParameter(UINT index);

        // 设置一个IDispatch接口指针
        void SetDispatch(IDispatch* pDisp);

        // 获取指定名字的DISPID
        bool GetDispid(BSTR functionName, DISPID* outDispid);
        // 调用指定DISPID的方法（必要的话提前建立参数数组）
        bool Invoke(DISPID dispid);
        // 设置一个指定名字的属性的值
        bool SetProperty(BSTR propName, VARIANT* inValue);
        // 获取一个指定名字的属性的值
        bool GetProperty(BSTR propName, VARIANT* outValue);
```

```
private:
        IDispatch*              mpDisp;         // 自动化对象的IDispatch指针
        VARIANT*                mpParams;       // 参数数组
        UINT                    mParamCount;    // 参数的个数
};

#endif //__DispatchHelper_H__
```

作为演示，客户程序访问自动化对象的属性、调用其方法的代码如下：

```
//
// Tester.cpp
//

#include "stdafx.h"
#include "Tester.h"

// 将GUID宏定义转化成常量的形式
#include <initguid.h>
#include "Mobile_i.c"

#include "safe_defs.h"
#include "Mobile.h"
#include "DispatchHelper.h"

// 演示程序的主函数
int _tmain(int argc, TCHAR* argv[], TCHAR* envp[])
{
        int nRetCode = 0;

        // COM库初始化
        CoInitialize(0);

        HRESULT hr = S_OK;

        // 创建一个自动化对象实例，并获取自动化接口
        // 提示：通常情况下，一个COM组件开发好后应该先在客户系统中注册，
        //      然后根据组件的CLSID、调用CoCreateInstance函数来创建对象实例。
        //      这里演示的创建组件对象的方法有些不同，但COM的基本原理没变，
        //      请读者注意区别和理解。
        Mobile* pMobileObj = new Mobile();
        IDispatch* pDispMobile = 0;
        hr = pMobileObj->QueryInterface(IID_IDispatch, (void**)&pDispMobile);

        bool ret = false;
        DispatchHelper dispatcher(pDispMobile);

        ////////////////////////////////////////////////////
        // --- 在该自动化对象上进行属性设置/获取的测试 ---
```

```
////////////////////////////////////////////////////
VARIANT var;
VariantInit(&var);

// 获取最初的“type”属性值
ret = dispatcher.GetProperty(L"type", &var);
VariantClear(&var);

// 设置“type”属性为一个新值："NOKIA"
var.vt = VT_BSTR;
var.bstrVal = ::SysAllocString(L"NOKIA");
ret = dispatcher.SetProperty(L"type", &var);
VariantClear(&var);

// 再次读取“type”属性值，验证上次设置是否成功
ret = dispatcher.GetProperty(L"type", &var);
ASSERT(var.vt == VT_BSTR && wcscmp(var.bstrVal, L"NOKIA") == 0);
VariantClear(&var);

// 获取当前的“color”属性值
ret = dispatcher.GetProperty(L"color", &var);
VariantClear(&var);

// 为“color”属性设置一个新值
var.vt = VT_I4;
var.lVal = 0x00808080;
ret = dispatcher.SetProperty(L"color", &var);
VariantClear(&var);

////////////////////////////////////////////////////
// --- 在该自动化对象上进行方法调用的测试 ---
////////////////////////////////////////////////////
// 首先查询获取“Call”方法的DISPID
DISPID dispid = 0;
if (dispatcher.GetDispid(L"Call", &dispid))
{
        // 为“Call”方法调用准备参数数组
        // “Call”方法只有1个参数
        dispatcher.CreateParameters(1);
        // 为参数数组中的各个参数指定一个值
        dispatcher.AddParameter(0, L"13988888888");
        // 调用“Call”方法
        dispatcher.Invoke(dispid);
}

////////////////////////////////////////////////////
// --- 双接口机制（对于C/C++可选的高效的访问方式） ---
////////////////////////////////////////////////////
// 查询获取IMobile接口指针
```

```
        IMobile* pMobile = 0;
        hr = pMobileObj->QueryInterface(IID_IMobile, (void**)&pMobile);
        if (pMobile)
        {
                // 获取“type”属性值
                BSTR bstrType = 0;
                pMobile->get_type(&bstrType);
                if (bstrType)
                        ::SysFreeString(bstrType);

                // 设置“type”属性值
                pMobile->put_type(L"SAMSUNG");

                // 获取“color”属性值
                long curColor = 0;
                pMobile->get_color(&curColor);

                // 设置“color”属性值
                pMobile->put_color(0x00FFFFFFFF);

                // 调用“Call”方法
                pMobile->Call(L"13666666666");

                // 释放对IMobile接口的引用
                pMobile->Release();
        }

        // 释放对自动化接口的引用
        SAFE_RELEASE(pDispMobile);

        // COM 库反初始化
        CoUninitialize();

        return nRetCode;
}
```

提示：上面的演示程序使用了两种方式来访问自动化对象：一种是通过自动化接口 IDispatch，另一种是通过 IMobile 接口（也就是直接通过 vtable 的方式）。这就是 COM 的双接口技术。不难看出，IMobile 接口比 IDispatch 接口使用起来要方便得多，对于 C/C++客户程序来说是个不错的选择；而 IDispatch 接口更适合于 VB、VBScript、JavaScript 等类型的客户程序使用。

图 5.4 概括了本节的内容，希望对读者理解利用类型库实现自动化对象的过程有所帮助。

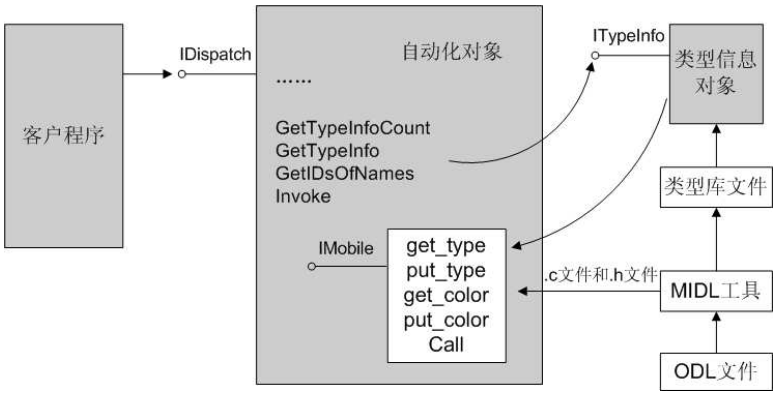

图 5.4 类型库与自动化对象之间的关系

提示：本书配套代码包的 Automation\Tester 目录下提供了本节演示程序的所有源文件。

5.2.3 实例程序：CurveSee

假设你去一家公司应聘，被要求做这样一道题目：对于用户输入的任意一个平面函数 f(x)，请绘制出其函数曲线。很有挑战性哦！

这里的技术难点是如何来计算表达式的值。因为用户输入的表达式是任意的一串字符，如果要计算这个表达式的值，一般的做法是：对表达式进行扫描，去括号，按照运算符的优先级生成二叉树，然后遍历该树生成逆波兰表达式，再然后通过栈的方法进行运算。如果在表达式中再包含有函数的话，那就更麻烦了……

那么有没有简单一点的解决方案呢？答案是肯定的。我们可以使用脚本解析技术！因为 JavaScript（或 VBScript）引擎能够计算表达式的值，更重要的是，它们能够直接接受字符串形式的表达式。假设现在有个表达式为：f(x) = sin(x)，我们只需在某个取值范围内、按照一定的步幅递增 x 的值，然后让脚本引擎去分别计算 sin(x)的值，并将计算的结果赋给变量 y，之后将每次 x 的值和对应的 y 值（以及当前循环计数器的 i 值）一起传回给应用程序；应用程序根据这么一对一对的 x、y 值，就能很方便地画出表达式的函数曲线了。这个过程可以用下面的这段脚本伪码来表示：

```
// 假设 x 的取值范围为[xMin, xMax]，在这个范围内要计算 maxCount 个采样点
var x = xMin;  // x 的初始取值
var xStride = (xMax - xMin) / maxCount;  // 计算 x 的步幅
var y;
for (var i = 0; i < maxCount; i++)
{
        // 在脚本中计算表达式的值
        y = sin(x);

        // 将计算结果传回应用程序
        Result(i, x, y);

        // 指向 x 的下一个采样点
        x += xStride;
}
```

接下去，我们就通过 CurveSee 这个例子，来介绍利用 Active Scripting 技术解答这个题目的完整实现。

5.2.3.1 设计受控对象

受控对象是应用程序方创建的、但能被脚本程序所访问或控制的一种对象，它是连接应用程序与脚本程序的一条纽带。我们需要利用这种受控对象，将每次从（脚本程序方）表达式计算出来的值传递回应用程序。

受控对象必须支持 COM 自动化，也就是说对象上必须实现 IDispatch 接口。5.2.2.4 节已经介绍过了自动化对象的实现方法。考虑到代码复用，我们将 IDispatch 接口函数的实现提取到了一个模板类 IDispatchImpl 中，参考如下：

```
//
// IDispatchImpl.h
//

#ifndef __H_IDispatchImpl__
#define __H_IDispatchImpl__

template<class DERIVED_CLASS, class BASE_ITF, const IID* BASE_IID>
class IDispatchImpl : public BASE_ITF
{
protected:
	long			m_cRef;		// 引用计数
	ITypeInfo*		m_pti;		// 类型信息
	ITypeLib*		m_ptl;		// 类型库

public:
	// 构造函数
	IDispatchImpl() : m_pti(0), m_ptl(0), m_cRef(1) {};

	// 析构函数
	virtual ~IDispatchImpl(void)
	{
		// 释放对类型信息接口的引用
		if (m_pti)
		{
			m_pti->Release();
			m_pti = 0;
		}
	}

	// 从类型库中查询当前自动化接口的类型信息
	void LoadTypeInfo(ITypeLib* ptl)
	{
		m_ptl = ptl;
		m_ptl->GetTypeInfoOfGuid(*BASE_IID, &m_pti);
```

```
}

// 返回类型信息接口
ITypeInfo* GetTypeInfoInterface() {return m_pti;}

// --- IUnknown接口的各个成员函数 ---
HRESULT STDMETHODCALLTYPE QueryInterface(REFIID riid, void **ppvObject)
{
	if (IID_IUnknown == riid)
	{
		*ppvObject = static_cast<IUnknown*> (this);
		AddRef();
		return S_OK;
	}
	else if (IID_IDispatch == riid)
	{
		*ppvObject = static_cast<IDispatch*> (this);
		AddRef();
		return S_OK;
	}
	else if (*BASE_IID == riid)
	{
		*ppvObject = static_cast<BASE_ITF*> (this);
		AddRef();
		return S_OK;
	}

	*ppvObject = 0;
	return E_NOINTERFACE;
}

ULONG STDMETHODCALLTYPE AddRef()
{
	return InterlockedIncrement(&m_cRef);
}

ULONG STDMETHODCALLTYPE Release()
{
	if (0 == InterlockedDecrement(&m_cRef))
	{
		delete this;
		return 0;
	}
	return m_cRef;
}


// --- IDispatch接口的各个成员函数 ---
STDMETHODIMP GetTypeInfoCount(UINT * pctinfo)
```

```
{
	*pctinfo = 1;
	return S_OK;
}

STDMETHODIMP GetTypeInfo(UINT itinfo, LCID /*lcid*/, ITypeInfo ** pptinfo)
{
	if (0 == pptinfo)
	{
		return E_POINTER;
	}
	if (0 != itinfo)
	{
		return TYPE_E_ELEMENTNOTFOUND;
	}
	if (0 == m_pti)
	{
		return TYPE_E_ELEMENTNOTFOUND;
	}

	*pptinfo = m_pti;
	m_pti->AddRef();
	return S_OK;
}

STDMETHODIMP GetIDsOfNames(REFIID /*riid*/,
	OLECHAR  ** rgszNames,
	UINT cNames,
	LCID /*lcid*/,
	DISPID * rgdispid)
{
	return m_pti->GetIDsOfNames(rgszNames, cNames, rgdispid);
}

STDMETHODIMP Invoke(DISPID dispidMember,
	REFIID riid,
	LCID /*lcid*/,
	WORD wFlags,
	DISPPARAMS * pdispparams,
	VARIANT * pvarResult,
	EXCEPINFO * pexcepinfo,
	UINT * puArgErr)
{
	if (IID_NULL != riid)
	{
		return DISP_E_UNKNOWNINTERFACE;
	}

	HRESULT hr = NOERROR;
```

```
                try
                {
                        hr = m_pti->Invoke((void*) this, dispidMember, wFlags,
                                pdispparams, pvarResult, pexcepinfo, puArgErr);
                }
                catch (...)
                {
                        hr = E_UNEXPECTED;
                }

                return hr;
        }
};

#endif //__H_IDispatchImpl__
```

现在我们需要一个 CCurvePainter 类型的受控对象，它能提供一个 Result 函数来传递 i、x、y 的值。这个对象可以用 ODL 语言进行如下的描述：

```
//
// curves.idl
//

import "oaidl.idl";
import "ocidl.idl";

[
        uuid(8F7156EC-E3C4-4f29-9E9F-312713079F47),
        version(1.0),
        helpstring("curves 1.0 Type Library")
]
library curvesTypeLib
{
        importlib("stdole32.tlb");
        importlib("stdole2.tlb");

        [
                object,
                uuid(1ECBE81A-B69C-4a02-924D-C128F5B2C654),
                dual
        ]
        interface ICurvePainter : IDispatch
        {
                HRESULT Result([in] long i, [in] double x, [in] double y);
        }

        ////////////////////////////////////////////////////////////////
        [
                uuid(31273E15-FB31-4d54-9239-E6BBA4F18507),
```

```
		helpstring("CurvePainter COM class"),
		appobject
	]
	coclass CCurvePainter
	{
		interface ICurvePainter;
	}
}
```

有了 IDispatchImpl 模板类，实现 CCurvePainter 类型的自动化对象就很方便了：

```
//
// CCurvePainter.h
//

#ifndef __H_CCurvePainter__
#define __H_CCurvePainter__

#include "IDispatchImpl.h"
#include "curves_h.h"

class CCurvePainter : public IDispatchImpl<CCurvePainter, ICurvePainter,
&IID_ICurvePainter>
{
public:
	CCurvePainter(ITypeLib* ptl);
	~CCurvePainter();

	// 开放给脚本的API函数：传递表达式计算的结果
	STDMETHODIMP Result(long i, double x, double y);
};

#endif // __H_CCurvePainter__


//
// CCurvePainter.cpp
//

#include "stdafx.h"
#include "CCurvePainter.h"
#include "CDataCurve.h"

////////////////////////////////////////////////////////////////////////////
// 构造函数
CCurvePainter::CCurvePainter(ITypeLib* ptl)
{
	// 从类型库中获取类型信息
	LoadTypeInfo(ptl);
```

```
}

// 析构函数
CCurvePainter::~CCurvePainter()
{
}

// 开放给脚本的API函数：传递表达式计算的结果
STDMETHODIMP CCurvePainter::Result(long i, double x, double y)
{
	if (i >= 0 && i < POINT_COUNT)
	{
		// 注：mXValues是应用程序内的保存各点x值的数组
		// mYValues是应用程序内的保存各点y值的数组
		GetSettings()->mXValues[i] = x;
		GetSettings()->mYValues[i] = y;
		return S_OK;
	}
	return E_UNEXPECTED;
}
```

值得注意的是，脚本通过受控对象的 Result 函数传递过来的 x、y 值，并没有直接在界面上表示出来，而是保存到一个 CDataCurve 类型的数据对象中去了。这个 CDataCurve 类型的对象被实现为了一个单件，参考如下：

```
//
// CDataCurve.h
//

#ifndef __H_CDataCurve__
#define __H_CDataCurve__

#define POINT_COUNT 600		// 用600个点来勾画出函数曲线

class CDataCurve
{
public:
	CString		mEngineType;	// 当前使用的脚本引擎类型
	CString		mExpression;	// 用户输入的表达式
	double		mXMax;		// X轴正半轴最大值
	double		mYMax;		// Y轴正半轴最大值
	double		mXValues[POINT_COUNT]; // 各点的x值
	double		mYValues[POINT_COUNT]; // 各点的y值

protected:
	CDataCurve();		// 不允许用户直接调用构造函数

public:
	static CDataCurve* GetInstance();	// 返回单件对象指针
```

```
        ~CDataCurve();

        // 复位前一次计算所得的各点的x、y值
        void ResetCurveData();

protected:
        static CDataCurve*    mInstance;     // 单件对象指针
};

extern CDataCurve* GetSettings();

#endif // __H_CDataCurve__


//
// CDataCurve.cpp
//

#include "stdafx.h"
#include "CDataCurve.h"

CDataCurve* GetSettings() { return(CDataCurve::GetInstance()); }

// 单件对象指针，初始化为0
CDataCurve* CDataCurve::mInstance = 0;

//////////////////////////////////////////////////////////////////////////
// 构造函数
CDataCurve::CDataCurve()
        : mEngineType("VBScript")
        , mExpression("sin(x)")
        , mXMax(6)
        , mYMax(2)
{
        ResetCurveData();
}

// 析构函数
CDataCurve::~CDataCurve()
{
}

// 返回单件对象指针
CDataCurve* CDataCurve::GetInstance()
{
        if (mInstance == 0)
        {
                // 创建单件对象
                static CDataCurve obj;
```

```
            mInstance = &obj;
        }
        return mInstance;
}

// 复位前一次计算所得的各点的x、y值
void CDataCurve::ResetCurveData()
{
        ZeroMemory(mXValues, sizeof(double)*POINT_COUNT);
        ZeroMemory(mYValues, sizeof(double)*POINT_COUNT);
}
```

5.2.3.2 设计宿主程序

使用 Active Scripting 进行脚本解析有两个要素：宿主程序和脚本引擎。宿主程序必须要实现 IActiveScriptSite 接口，必须要创建和管理脚本引擎对象，并且能够提供一个公共函数，让用户可以通过这个函数去执行指定的一段脚本代码。CurveSee 例子中这个宿主程序就是 CWinScriptHost，它的实现如下：

```
//
// CWinScriptHost.h
//

#ifndef __H_CWinScriptHost__
#define __H_CWinScriptHost__

#include "activscp.h"

class CCurvePainter;

class CWinScriptHost : public IActiveScriptSite
{
public:
        CWinScriptHost();
        ~CWinScriptHost();

        // 设置脚本引擎的类型：“VBScript”或者“JScript”
        void UseScriptEngine(CString& inEngineType);
        // 初始化
        bool Initialize();
        // 反初始化
        void Finalize();
        // 执行指定的脚本代码
        void Execute(CString& scripts);

        // --- IUnknown接口的各个成员函数 ---
        HRESULT STDMETHODCALLTYPE QueryInterface( /* [in] */ REFIID riid,
                /* [out] */ void  **ppvObject );
        ULONG STDMETHODCALLTYPE AddRef();
```

```
        ULONG STDMETHODCALLTYPE Release();

        // --- IActiveScriptSite接口的各个成员函数 ---
        STDMETHODIMP GetLCID(LCID * plcid);
        STDMETHODIMP GetItemInfo(LPCOLESTR pstrName, DWORD dwReturnMask,
               IUnknown **ppunkItem, ITypeInfo **ppTypeInfo);
        STDMETHODIMP GetDocVersionString(BSTR *pbstrVersionString);
        STDMETHODIMP OnScriptTerminate(const VARIANT *pvarResult, const EXCEPINFO
*pexcepinfo);
        STDMETHODIMP OnStateChange(SCRIPTSTATE ssScriptState);
        STDMETHODIMP OnScriptError(IActiveScriptError *pase);
        STDMETHODIMP OnEnterScript(void);
        STDMETHODIMP OnLeaveScript(void);

private:
        long                m_cRef;              // 引用计数器
        bool                mInited;             // 标记：是否已经初始化？
        BSTR                mEngineType;         // 脚本引擎的类型
        ITypeLib *          mTypeLib;            // 类型库对象指针
        IActiveScript *     mActiveScript;       // 脚本引擎对象指针
        IActiveScriptParse* mActiveScriptParse;  // 用于脚本解析的接口

        // 受控对象
        CCurvePainter*      mPainter;
};

#endif // __H_CWinScriptHost__


//
// CWinScriptHost.cpp
//

#include "stdafx.h"

#include <initguid.h>
#include "curves_i.c"

#include "safe_defs.h"
#include "CWinScriptHost.h"
#include "CDataCurve.h"
#include "CCurvePainter.h"

////////////////////////////////////////////////////////////////////////////
// 构造函数
CWinScriptHost::CWinScriptHost()
        : mInited(false)
        , mEngineType(0)
        , m_cRef(1)    // 注意初值为1
```

```
        , mTypeLib(0)
        , mActiveScript(0)
        , mActiveScriptParse(0)
        , mPainter(0)
{
        // 装载类型库
        HRESULT hr = S_OK;
        hr = ::LoadTypeLib(L"curves.tlb", &mTypeLib);

        // 创建受控对象
        mPainter = new CCurvePainter(mTypeLib);
}

// 析构函数
CWinScriptHost::~CWinScriptHost()
{
        // 销毁受控对象
        SAFE_RELEASE(mPainter);

        // 释放字符串资源
        if (mEngineType)
        {
                ::SysFreeString(mEngineType);
                mEngineType = 0;
        }

        // 进行最后的反初始化
        Finalize();
}

// UseScriptEngine - 设置脚本引擎的类型
// 参数：inEngineType  -  脚本引擎的类型，取值为“VBScript”或“JScript”
void CWinScriptHost::UseScriptEngine(CString& inEngineType)
{
        if (mEngineType)
        {
                ::SysFreeString(mEngineType);
                mEngineType = 0;
        }

        mEngineType = inEngineType.AllocSysString();
}

// Initialize  -  初始化
// 返回值: true，成功；false，失败
bool CWinScriptHost::Initialize()
{
        // 如果已经进行过初始化，则首先进行反初始化
        if (mInited)
```

```
		Finalize();

	// 开始初始化过程…
	// 创建一个脚本引擎对象
	CLSID clsid;
	HRESULT hr = ::CLSIDFromProgID(mEngineType, &clsid);
	if (SUCCEEDED(hr))
	{
		hr = ::CoCreateInstance(clsid, NULL, CLSCTX_INPROC_SERVER,
			IID_IActiveScript, (void**)&mActiveScript);
	}
	if (mActiveScript == 0)
		return false;

	// 从脚本引擎对象上获取必要的接口
	hr = mActiveScript->QueryInterface(IID_IActiveScriptParse, (void**)&mActiveScriptParse);

	//////////////// 对脚本引擎进行参数设置 /////////////////
	// 将宿主程序与脚本引擎进行关联
	hr = mActiveScript->SetScriptSite(this);
	// 为脚本引擎增加一个我们自定义的名字项：“CurvePainter”
	hr = mActiveScript->AddNamedItem(L"CurvePainter", SCRIPTITEM_ISVISIBLE);
	// 脚本解析前进行必要的初始化
	hr = mActiveScriptParse->InitNew();

	mInited = SUCCEEDED(hr);
	return true;
}

// Finalize - 反初始化
void CWinScriptHost::Finalize()
{
	// 释放对脚本引擎对象上的所有接口引用
	SAFE_RELEASE(mActiveScriptParse);
	if (mActiveScript)
	{
		mActiveScript->Close();
		mActiveScript->Release();
		mActiveScript = 0;
	}

	mInited = false;
}

// Execute - 执行用户指定的脚本代码
// 参数：scripts - 脚本代码
void CWinScriptHost::Execute(CString& scripts)
{
	HRESULT hr = S_OK;
```

```
        EXCEPINFO ei;

        // 将脚本代码转化成宽字符的形式
        BSTR wszScripts = scripts.AllocSysString();

        // 将脚本代码传递给引擎对象
        hr = mActiveScriptParse->ParseScriptText(wszScripts, L"CurvePainter",
                0, 0, 0, 0, 0, 0, &ei);

        // 释放字符串资源
        if (wszScripts)
                ::SysFreeString(wszScripts);

        // 设置脚本引擎进入连接状态（以便开始执行上述脚本代码）
        hr = mActiveScript->SetScriptState(SCRIPTSTATE_CONNECTED);
}

// --- IUnknown接口的各个成员函数 ---
// 查询接口（实现时一定要记得暴露IActiveScriptSite接口）
HRESULT STDMETHODCALLTYPE CWinScriptHost::QueryInterface(REFIID riid,
                        void **ppvObject)
{
        if (IID_IUnknown == riid)
        {
                *ppvObject = static_cast<IUnknown*>((void*)(this));
                AddRef();
                return S_OK;
        }
        else if (IID_IActiveScriptSite == riid)
        {
                *ppvObject = static_cast<IActiveScriptSite*> (this);
                AddRef();
                return S_OK;
        }

        *ppvObject = 0;
        return E_NOINTERFACE;
}

// 对象上的引用计数增加1
ULONG STDMETHODCALLTYPE CWinScriptHost::AddRef()
{
        return InterlockedIncrement(&m_cRef);
}

// 对象上的引用计数减少1
ULONG STDMETHODCALLTYPE CWinScriptHost::Release()
{
        if (0 == InterlockedDecrement(&m_cRef))
```

```
	{
		delete this;
		return 0;
	}
	return m_cRef;
}

// --- IActiveScriptSite接口的各个成员函数 ---
// 获取本地化标识（未实现）
STDMETHODIMP CWinScriptHost::GetLCID(LCID * plcid)
{
	plcid;
	return E_NOTIMPL;
}

// 获取名字项信息
STDMETHODIMP CWinScriptHost::GetItemInfo(LPCOLESTR pstrName,
                                         DWORD dwReturnMask,
                                         IUnknown **ppunkItem,
                                         ITypeInfo **ppTypeInfo)
{
	// 获取名字项对应的类型信息
	if (ppTypeInfo)
	{
		*ppTypeInfo = 0;
		if (dwReturnMask & SCRIPTINFO_ITYPEINFO)
		{
			if (0 == wcscmp(L"CurvePainter", pstrName))
			{
				// 返回ICurvePainter接口的类型信息
				mTypeLib->GetTypeInfoOfGuid(IID_ICurvePainter, ppTypeInfo);
			}
			else
			{
				return E_UNEXPECTED;
			}
		}
	}

	// 获取名字项对应的受控对象指针
	if (ppunkItem)
	{
		*ppunkItem = 0;
		if (dwReturnMask & SCRIPTINFO_IUNKNOWN)
		{
			if (0 == wcscmp(L"CurvePainter", pstrName))
			{
				// 返回受控对象指针
				*ppunkItem = static_cast<IUnknown*> (mPainter);
				mPainter->AddRef();
```

```
                        }
                        else
                        {
                                return E_UNEXPECTED;
                        }
                }
        }

        return S_OK;
}

// 获取当前的文档版本字符串（未实现）
STDMETHODIMP CWinScriptHost::GetDocVersionString(BSTR *pbstrVersionString)
{
        return E_NOTIMPL;
}

//脚本引擎用以通知宿主程序：脚本执行完成了
STDMETHODIMP CWinScriptHost::OnScriptTerminate(const VARIANT *pvarResult, const
EXCEPINFO *pexcepinfo)
{
        return S_OK;
}

// 获取脚本引擎状态改变通知
STDMETHODIMP CWinScriptHost::OnStateChange(SCRIPTSTATE ssScriptState)
{
        return S_OK;
}

// 获取脚本解析时发生的错误
STDMETHODIMP CWinScriptHost::OnScriptError(IActiveScriptError *pase)
{
        HRESULT hr = NOERROR;

//      BSTR szSrcLine;
//      hr = pase->GetSourceLineText(&szSrcLine);

        // 获取出错的代码行信息
        DWORD  srcContext = 0;
        ULONG  lineNumber = 0;
        LONG   charPosition = 0;
        hr = pase->GetSourcePosition(&srcContext, &lineNumber, &charPosition);

        // 获取异常信息
        EXCEPINFO ei;
        ZeroMemory(&ei, sizeof(ei));
        hr = pase->GetExceptionInfo(&ei);
        if (SUCCEEDED(hr))
```

```
        {
                if (ei.bstrSource)
                        ::SysFreeString(ei.bstrSource);
                if (ei.bstrDescription)
                        ::SysFreeString(ei.bstrDescription);
                if (ei.bstrHelpFile)
                        ::SysFreeString(ei.bstrHelpFile);
        }

        return hr;
}

// 获取通知：脚本引擎开始执行脚本代码了
STDMETHODIMP CWinScriptHost::OnEnterScript(void)
{
        return S_OK;
}

// 获取通知：脚本引擎完成了脚本代码的执行
STDMETHODIMP CWinScriptHost::OnLeaveScript(void)
{
        return S_OK;
}
```

5.2.3.3 执行脚本

受控对象和宿主程序都准备好后，就可以编写脚本代码来计算表达式的值了。计算而得的各点的 x、y 值都保存在 CDataCurve 类型的对象中。应用程序通过访问 CDataCurve 对象中的记录，很容易就能画出表达式对应的函数曲线了。

```
// 当用户输入一个表达式（或改变了脚本引擎类型）后需要重画函数曲线
void CCurveSeeDlg::OnFunctionsSettings()
{
        // 将前一次计算的函数曲线各点的x、y值都复位
        GetSettings()->ResetCurveData();

        // 设置一种脚本引擎类型
        mScriptHost.UseScriptEngine(GetSettings()->mEngineType);

        // 宿主程序进行初始化
        // 注：mScriptHost是CWinScriptHost类型的对象
        mScriptHost.Initialize();

        // 计算X轴的步幅
        double strideX = 2 * GetSettings()->mXMax / POINT_COUNT;

        // 判断脚本引擎的类型
        // 如果是JScript，则要编写JavaScript代码来计算表达式的值
        // 如果是VBScript，则要编写VBScript代码来计算表达式的值
```

```
if (GetSettings()->mEngineType == "JScript")
{
	// 使用一个循环，计算各点x值对应的y值
	for (long i = 0; i < POINT_COUNT; i++)
	{
		/*****************************************************
		编写的JavaScript代码类似如下:

		i = 8;
		x = -6.0;
		y = Math.sin(x);
		Result(i, x, y);
		*****************************************************/

		CString scripts;
		double x = i * strideX - GetSettings()->mXMax;
		scripts.Format("i=%d;\nx=%.5f;\ny=Math.%s;\nResult(i,x,y);",
			i, x, GetSettings()->mExpression);

		// 执行上述脚本代码
		mScriptHost.Execute(scripts);
	}
}
else
{
	// 使用一个循环，计算各点x值对应的y值
	for (long i = 0; i < POINT_COUNT; i++)
	{
		/*****************************************************
		编写的VBScript代码类似如下:

		i = 8
		x = -6.0
		y = sin(x)
		call Result(i, x, y)
		*****************************************************/

		CString scripts;
		double x = i * strideX - GetSettings()->mXMax;
		scripts.Format("i=%d\r\nx=%.5f\r\ny=%s\r\ncall Result(i,x,y)",
			i, x, GetSettings()->mExpression);

		// 执行上述脚本代码
		mScriptHost.Execute(scripts);
	}
}

// 函数曲线各点的x、y值都计算好了，函数曲线需要进行重画
Repaint();
```

```
}

// 重画函数曲线
void CCurveSeeDlg::Repaint()
{
	// 获取窗口的DC
	CPaintDC dc(this);

	// 画出X、Y坐标系
	DrawXYAxes(dc);

	// 画出函数曲线
	DrawCurve(dc);

	Invalidate();
}

// 画出 X、Y 坐标系
void CCurveSeeDlg::DrawXYAxes(CPaintDC& dc)
{
	// 计算得到原点的x、y值
	RECT rc;
	GetClientRect(&rc);
	long centerX = rc.left + (rc.right - rc.left) / 2;
	long centerY = rc.top + (rc.bottom - rc.top) / 2;

	// 画出X轴
	dc.MoveTo(rc.left, centerY);
	dc.LineTo(rc.right, centerY);

	// 画出Y轴
	dc.MoveTo(centerX, rc.top);
	dc.LineTo(centerX, rc.bottom);
}

// 画出函数曲线
void CCurveSeeDlg::DrawCurve(CPaintDC& dc)
{
	// 计算得到原点的x、y值
	RECT rc;
	GetClientRect(&rc);
	long centerX = rc.left + (rc.right - rc.left) / 2;
	long centerY = rc.top + (rc.bottom - rc.top) / 2;

	// 为函数曲线指定一种颜色
	COLORREF clr = RGB(255, 0, 0);		// 红色

	// 根据CDataCurve对象中记录的各点x、y值画出函数曲线
	double strideX = (rc.right - rc.left) / (2 * GetSettings()->mXMax);
```

```
        double strideY = (rc.bottom - rc.top) / (2 * GetSettings()->mYMax);
        for (long i = 0; i < POINT_COUNT; i++)
        {
                long xPos = (long)(centerX + strideX * GetSettings()->mXValues[i]);
                long yPos = (long)(centerY - strideY * GetSettings()->mYValues[i]);

                dc.SetPixel(xPos, yPos, clr);
        }
}
```

5.2.3.4 演示说明

实例程序 CurveSee 位于本书配套代码包的 CurveSee 目录下。这是一个通过 Visual Studio .NET 2003 向导生成的、基于对话框的 MFC 应用程序。软件运行起来界面如图 5.5：

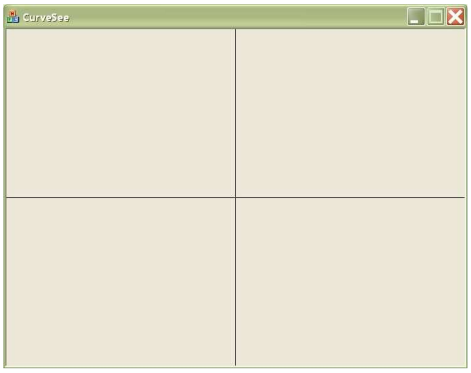

图 5.5 CurveSee 的程序界面

CurveSee 程序的使用方法：在主界面上单击鼠标右键，在随后出现的快捷菜单中选择“Settings...”一项，弹出一个如图 5.6 的设置对话框：

图 5.6 CurveSee 的设置对话框

通过图 5.6 的对话框，可以调整 X 轴和 Y 轴的正半轴最大值，可以选择一种脚本引擎的类型，可以在“f(x)=　　”一项输入一个任意的表达式。设置完成之后按下 OK 按钮，就可在主界面上看到函数曲线了，如图 5.7：

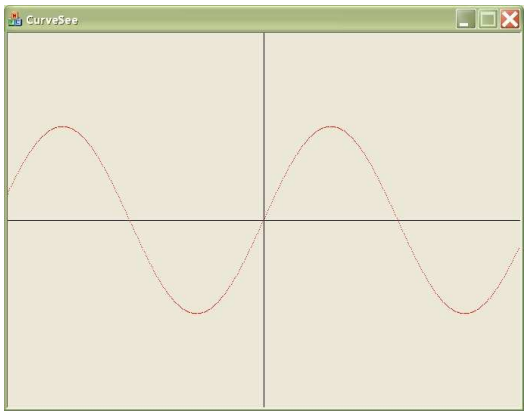

图 5.7 CurveSee 显示的正弦曲线

我们还可以输入一个稍微复杂一点的表达式，如：f(x) = sin(2*x) + 2*cos(x) + 0.6，但此时脚本引擎必须选择为“VBScript”（读者可以思考一下为什么），设置如图 5.8：

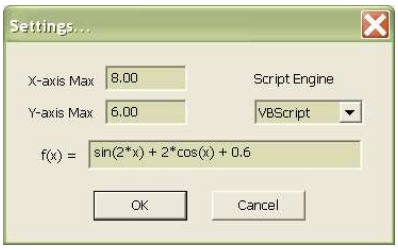

图 5.8 CurveSee 的参数设置（演示较为复杂的表达式）

这时 CurveSee 显示出来的函数曲线如图 5.9：

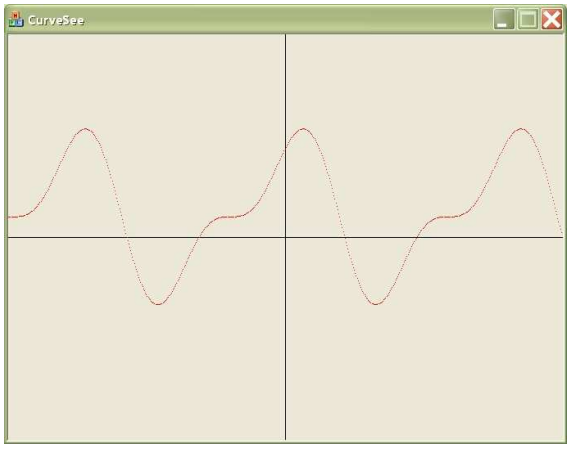

图 5.9 CurveSee 显示的复杂函数曲线

5.3 JavaScript 解析要点

通过上一节 CurveSee 例子的学习，相信读者已经基本掌握了使用 Active Scripting 技术进行 JavaScript 解析（或 VBScript 解析）的一般方法。但在实际的软件开发过程中，对于 JavaScript 解析的应用要求可能会比这个例子要深入很多。本书接下去也会就部分常用的要点进行分析，并给出完整的解决方案。

提示：关于 JavaScript 解析的各个要点的演示都包含在 JSInterpreting 实例程序中，该程序位于本书配套代码包的 JSInterpreting 目录下。为简单起见，JSInterpreting 被实现为了一个控制台程序。

5.3.1 名字项与全局对象

名字项是宿主程序方的受控对象在脚本程序中的一个标记符号，它代表的对象是根级的（或者说顶级的）。通过这种名字项标记，脚本程序就可以访问受控对象的属性、或调用其方法。在 Active Scripting 中，宿主程序通过 IActiveScript::AddNamedItem 将名字项告诉脚本引擎，而脚本引擎通过 IActiveScriptSite::GetItemInfo 从宿主程序获取名字项与受控对象的关联（包括获取受控对象的类型信息和 IUnknown 指针）。

现在假设有一个名字项为“Application”的受控对象，它有一个 alert 函数。那么在脚本程序中，我们可以通过如下的代码行来调用 Application 对象的 alert 函数：

```
Application.alert("Called by Application.alert().");
```

如果我们在增加名字项的时候把“Application”扩展为全局对象（也就是说在调用 IActiveScript::AddNamedItem函数时为第2个参数增加SCRIPTITEM_GLOBALMEMBERS属性），我们甚至可以将上述脚本代码改写成：

```
alert("Called by alert() -- Global Members.");
```

注意：这里省略了“Application”这个全局对象名。Application 的成员函数 alert 这时候看起来更像是一个全局函数了。

为了正确解析上述两行脚本代码，我们需要做如下实现：

- 使用 IDL 语言来描述 Application 对象
- 实现 Application 对象
- 设计一个宿主程序，并在宿主程序中创建、管理 Application 对象
- 将脚本代码放在 TestingCase01.js 文件中
- 装载脚本文件并执行其中的脚本代码

```
// <1> 使用 IDL 语言来描述 Application 对象
//
// js_demo.idl
//
```

```
[
	object,
	uuid(E74835F6-EBDD-49b2-986D-234B70AF74E4),
	dual
]
interface IApplication : IDispatch
{
	// 方法
	HRESULT alert([in] BSTR msg);
}


// <2> 实现 Application 对象
//
//
// CApplication.h
//

#ifndef __H_CApplication__
#define __H_CApplication__

#include "IDispatchImpl.h"
#include "js_demo_h.h"

class CApplication : public IDispatchImpl<CApplication
                    , IApplication
                    , &IID_IApplication>
{
public:
	CApplication(ITypeLib* ptl);
	~CApplication();

	// 开放给脚本程序的API函数
	STDMETHODIMP alert(BSTR msg);
};

#endif // __H_CApplication__


//
// CApplication.cpp
//

#include "stdafx.h"
#include "CDebugger.h"
#include "CApplication.h"

////////////////////////////////////////////////////////////////////
```

```
// 构造函数
CApplication::CApplication(ITypeLib* ptl)
	: mCar(0)
	, mpFileThread(0)
{
	// 装载类型信息
	LoadTypeInfo(ptl);
}

// 析构函数
CApplication::~CApplication()
{
}

// 开放给脚本程序的API函数
STDMETHODIMP CApplication::alert(BSTR msg)
{
	// 在控制台输出跟踪信息
	GetDebugger()->trace(L"alert", msg);

	// 弹出一个消息框
#ifndef _DEBUG
	int charCount = int(wcslen(msg) + 1);
	char* szMsg = new char[charCount];
	::WideCharToMultiByte(CP_ACP, 0, msg, -1, szMsg, charCount, 0, 0);
	::MessageBox(0, szMsg, "JS Interpreting", MB_OK);
	delete[] szMsg;
#endif

	return S_OK;
}


// <3> 设计一个宿主程序，并在宿主程序中创建、管理 Application 对象
// 注意：为“Application”名字项增加 SCRIPTITEM_GLOBALMEMBERS 属性
//
// CJScriptHost.h
//

#ifndef __H_CJScriptHost__
#define __H_CJScriptHost__

#include "activscp.h"

class CApplication;

class CJScriptHost : public IActiveScriptSite
{
public:
```

```
        CJScriptHost();
        ~CJScriptHost();

        // 初始化
        bool Initialize();
        // 反初始化
        void Finalize();
        // 执行脚本代码
        void Execute(wchar_t* scripts);

        // --- IUnknown接口的各个成员函数 ---
        HRESULT STDMETHODCALLTYPE QueryInterface( /* [in] */ REFIID riid,
                /* [out] */ void  **ppvObject );
        ULONG STDMETHODCALLTYPE AddRef();
        ULONG STDMETHODCALLTYPE Release();

        // --- IActiveScriptSite接口的各个成员函数 ---
        STDMETHODIMP GetLCID(LCID * plcid);
        STDMETHODIMP GetItemInfo(LPCOLESTR pstrName, DWORD dwReturnMask,
                IUnknown **ppunkItem, ITypeInfo **ppTypeInfo);
        STDMETHODIMP GetDocVersionString(BSTR *pbstrVersionString);
        STDMETHODIMP OnScriptTerminate(const VARIANT *pvarResult, const EXCEPINFO
*pexcepinfo);
        STDMETHODIMP OnStateChange(SCRIPTSTATE ssScriptState);
        STDMETHODIMP OnScriptError(IActiveScriptError *pase);
        STDMETHODIMP OnEnterScript(void);
        STDMETHODIMP OnLeaveScript(void);

private:
        long                m_cRef;             // COM引用计数
        bool                mInited;            // 初始化标记
        ITypeLib *          mTypeLib;           // 类型库
        IActiveScript *     mActiveScript;      // 脚本引擎对象
        IActiveScriptParse* mActiveScriptParse; // 脚本解析接口

        // 受控对象
        CApplication*       mApplication;
};

#endif // __H_CJScriptHost__


//
// CJScriptHost.cpp
//

#include "stdafx.h"

#include <initguid.h>
```

```
#include "js_demo_i.c"

#include "safe_defs.h"
#include "CJScriptHost.h"
#include "CApplication.h"
#include "CDebugger.h"
#include "CScriptEngineWrapper.h"

//////////////////////////////////////////////////////////////////////////
// 构造函数
CJScriptHost::CJScriptHost()
	: mInited(false)
	, m_cRef(1)
	, mTypeLib(0)
	, mActiveScript(0)
	, mActiveScriptParse(0)
	, mApplication(0)
{
	// 装载类型库
	HRESULT hr = S_OK;
	hr = ::LoadTypeLib(L"js_demo.tlb", &mTypeLib);

	// 创建受控对象
	mApplication = new CApplication(mTypeLib);

	// 创建一个调试器对象（用于在控制台输出跟踪信息）
	CDebugger::CreateInstance(mTypeLib);
}

// 析构函数
CJScriptHost::~CJScriptHost()
{
	// 销毁受控对象
	SAFE_RELEASE(mApplication);
	CDebugger::DeleteInstance();

	// 反初始化
	Finalize();
}

// 初始化
bool CJScriptHost::Initialize()
{
	if (mInited)
		Finalize();

	// 创建一个JScript引擎对象
	CLSID clsid;
	HRESULT hr = ::CLSIDFromProgID(L"JScript", &clsid);
```

```
	if (SUCCEEDED(hr))
	{
		hr = ::CoCreateInstance(clsid, NULL, CLSCTX_INPROC_SERVER,
			IID_IActiveScript, (void**)&mActiveScript);
	}
	if (mActiveScript == 0)
		return false;

	// 获取脚本解析所需的接口
	hr = mActiveScript->QueryInterface(IID_IActiveScriptParse, (void**)&mActiveScriptParse);

	// 建立宿主程序与脚本引擎的关联
	hr = mActiveScript->SetScriptSite(this);
	// 增加必要的名字项
	// 注意：“Application”为全局对象！
	hr = mActiveScript->AddNamedItem(L"Application", SCRIPTITEM_ISVISIBLE|SCRIPTITEM_GLOBALMEMBERS);
	hr = mActiveScript->AddNamedItem(L"Debugger", SCRIPTITEM_ISVISIBLE);
	// 脚本解析前的初始化
	hr = mActiveScriptParse->InitNew();

	mInited = SUCCEEDED(hr);
	return true;
}

// 反初始化
void CJScriptHost::Finalize()
{
	// 释放对引擎对象所有的接口引用
	SAFE_RELEASE(mActiveScriptParse);
	if (mActiveScript)
	{
		mActiveScript->Close();
		mActiveScript->Release();
		mActiveScript = 0;
	}

	mInited = false;
}

// 执行脚本代码
void CJScriptHost::Execute(wchar_t* scripts)
{
	HRESULT hr = S_OK;

	EXCEPINFO ei;
	hr = mActiveScriptParse->ParseScriptText(scripts, 0, 0, 0, 0, 0, 0, 0, &ei);

	hr = mActiveScript->SetScriptState(SCRIPTSTATE_CONNECTED);
}
```

```
// --- IUnknown接口的各个成员函数 ---
// 查询获取riid对应类型的接口指针
HRESULT STDMETHODCALLTYPE CJScriptHost::QueryInterface(REFIID riid, void **ppvObject)
{
	if (IID_IUnknown == riid)
	{
		*ppvObject = static_cast<IUnknown*>((void*)(this));
		AddRef();
		return S_OK;
	}
	else if (IID_IActiveScriptSite == riid)
	{
		*ppvObject = static_cast<IActiveScriptSite*> (this);
		AddRef();
		return S_OK;
	}

	*ppvObject = 0;
	return E_NOINTERFACE;
}

// 对象上的引用计数增加1
ULONG STDMETHODCALLTYPE CJScriptHost::AddRef()
{
	return InterlockedIncrement(&m_cRef);
}

// 对象上的引用计数减少1
ULONG STDMETHODCALLTYPE CJScriptHost::Release()
{
	if (0 == InterlockedDecrement(&m_cRef))
	{
		delete this;
		return 0;
	}
	return m_cRef;
}

// --- IActiveScriptSite接口的各个成员函数 ---
// 获取本地化标识（未实现）
STDMETHODIMP CJScriptHost::GetLCID(LCID * plcid)
{
	return E_NOTIMPL;
}

// 获取名字项对应的类型信息及受控对象指针
STDMETHODIMP CJScriptHost::GetItemInfo(LPCOLESTR pstrName,
                                       DWORD dwReturnMask,
```

```
                                        IUnknown **ppunkItem,
                                        ITypeInfo **ppTypeInfo)
{
        // 获取名字项对应的类型信息
        if (ppTypeInfo)
        {
                *ppTypeInfo = 0;
                if (dwReturnMask & SCRIPTINFO_ITYPEINFO)
                {
                        if (0 == wcscmp(L"Application", pstrName))
                        {
                                // 返回Application对象的类型信息
                               mTypeLib->GetTypeInfoOfGuid(IID_IApplication,  ppTypeInfo);
                        }
                        else if (0 == wcscmp(L"Debugger", pstrName))
                        {
                                // 返回Debugger对象的类型信息
                                mTypeLib->GetTypeInfoOfGuid(IID_IDebugger, ppTypeInfo);
                        }
                        else
                        {
                                return E_UNEXPECTED;
                        }
                }
        }

        // 获取名字项对应的受控对象的指针
        if (ppunkItem)
        {
                *ppunkItem = 0;
                if (dwReturnMask & SCRIPTINFO_IUNKNOWN)
                {
                        if (0 == wcscmp(L"Application", pstrName))
                        {
                                // 返回Application对象的IUnknown指针
                                *ppunkItem = static_cast<IUnknown*> (mApplication);
                                mApplication->AddRef();
                        }
                        else if (0 == wcscmp(L"Debugger", pstrName))
                        {
                                // 返回Debugger对象的IUnknown指针
                                *ppunkItem = static_cast<IUnknown*> (GetDebugger());
                                GetDebugger()->AddRef();
                        }
                        else
                        {
                                return E_UNEXPECTED;
                        }
                }
```

```
	}

	return S_OK;
}

// 获取当前的文档版本字符串（未实现）
STDMETHODIMP CJScriptHost::GetDocVersionString(BSTR *pbstrVersionString)
{
	return E_NOTIMPL;
}

// 脚本引擎用以通知宿主程序：脚本执行完成了
STDMETHODIMP CJScriptHost::OnScriptTerminate(const VARIANT *pvarResult, const
EXCEPINFO *pexcepinfo)
{
	return S_OK;
}

// 获取脚本引擎状态改变通知
STDMETHODIMP CJScriptHost::OnStateChange(SCRIPTSTATE ssScriptState)
{
	return S_OK;
}

// 获取脚本解析时发生的错误
STDMETHODIMP CJScriptHost::OnScriptError(IActiveScriptError *pase)
{
	HRESULT hr = NOERROR;

//	BSTR szSrcLine;
//	hr = pase->GetSourceLineText(&szSrcLine);

	// 获取出错行信息
	DWORD  srcContext = 0;
	ULONG  lineNumber = 0;
	LONG   charPosition = 0;
	hr = pase->GetSourcePosition(&srcContext, &lineNumber, &charPosition);

	// 获取异常描述
	EXCEPINFO ei;
	ZeroMemory(&ei, sizeof(ei));
	hr = pase->GetExceptionInfo(&ei);
	if (SUCCEEDED(hr))
	{
		if (ei.bstrSource)
			::SysFreeString(ei.bstrSource);
		if (ei.bstrDescription)
			::SysFreeString(ei.bstrDescription);
		if (ei.bstrHelpFile)
```

```
                    ::SysFreeString(ei.bstrHelpFile);
        }

        return hr;
}

// 获取通知：脚本引擎开始执行脚本代码了
STDMETHODIMP CJScriptHost::OnEnterScript(void)
{
        return S_OK;
}

// 获取通知：脚本引擎完成了脚本代码的执行
STDMETHODIMP CJScriptHost::OnLeaveScript(void)
{
        return S_OK;
}


// <4> 将脚本代码放在 TestingCase01.js 文件中
//
// TestingCase01.js
//

Debugger.trace("Info", GetInfo());
Debugger.trace("Testing Case 1", "Named Item and Global Members");

//////// << 本节演示的待解析的脚本代码 >> ////////
Application.alert("Called by Application.alert().");
alert("Called by alert() -- Global Members.");

// 获取当前的系统时间、以及脚本引擎的版本号等信息
function GetInfo()
{
        var d, s;

        d = new Date();

        s = "You are running ";
        s += ScriptEngine() + " Version ";
        s += ScriptEngineMajorVersion() + ".";
        s += ScriptEngineMinorVersion() + ".";
        s += ScriptEngineBuildVersion() + " at ";
        s += d.getHours() + ":";
        s += d.getMinutes() + ":";
        s += d.getSeconds() + ":";
        s += d.getMilliseconds() + " ";
        s += (d.getMonth() + 1) + "/";
        s += d.getDate() + "/";
```

```
        s += d.getYear() + "\n";

        return s;
}


// <5> 装载脚本文件并执行其中的脚本代码
//
// JSInterpreting.cpp
//

// LoadScriptFile - 将指定文件中的脚本程序装载到一个缓存中
// 参数: inFile - 脚本文件
//       outBuf - 用于装载脚本代码的缓存
// 返回值: true, 成功; false, 失败
bool LoadScriptFile(const char * inFile, wchar_t** outBuf)
{
        // 打开脚本文件, 并读取所有的文件内容
        FILE * fp = fopen(inFile, "rb");
        if (fp)
        {
                // 获取文件大小
                fseek(fp, 0, SEEK_END);
                long fileSize = ftell(fp);
                if (fileSize <= 0)
                {
                        fclose(fp);
                        return false;
                }

                ///////////////// 读取所有的文件内容 /////////////////
                long  charCount = fileSize + 1; // 包括一个null结束符
                // 申请内存空间
                char * pScripts = new char[charCount];
                memset(pScripts, 0, charCount);
                fseek(fp, 0, SEEK_SET);
                // 将文件内容读入缓存中
                fread(pScripts, 1, fileSize, fp);
                fclose(fp);

                // 将脚本程序转化成宽字符串的形式
                *outBuf = new wchar_t[charCount];
                ::MultiByteToWideChar(CP_ACP, 0, pScripts, -1, *outBuf, charCount);
                delete[] pScripts;
                return true;
        }

        return false;
}
```

```
// 程序空闲时刻调用的函数
void OnIdle()
{
	// 暂无实现
	// ...
}

// 演示程序的主函数
int _tmain(int argc, _TCHAR* argv[])
{
	// COM库初始化
	CoInitialize(0);

	// 脚本解析...
	CJScriptHost jsHost;
	if (jsHost.Initialize()) // 初始化
	{
		// 本书总共准备了14个测试用例来演示脚本解析的各个要点，
		// 各个测试用例涉及的脚本程序分别被保存为一个.js文件
		const int kCaseCount = 14;
		for (int i = 0; i < kCaseCount; i++)
		{
			// 指定一个脚本文件名
			char szFilename[100];
			sprintf(szFilename, "TestingCase%02d.js", i+1);

			// 从脚本文件中获取脚本代码
			wchar_t * wszScripts = 0;
			if (LoadScriptFile(szFilename, &wszScripts))
			{
				// 执行脚本代码
				jsHost.Execute(wszScripts);
				delete[] wszScripts;
			}
		}
	}

	// 等待用户输入回车键，以结束本演示程序
	wprintf(L"\nPress <Enter> on console to exit.\n");
	while (GetAsyncKeyState(VK_RETURN) == 0)
	{
		OnIdle();
		Sleep(100);
	}

	jsHost.Finalize(); // 反初始化

	// COM库反初始化
```

```
    CoUninitialize();

    return 0;
}
```

本节演示的脚本解析执行结果如图 5.10：

```
[Info] You are running JScript Version 5.6.8820 at 12:38:39:562 5/7/2006

[Testing Case 1] Named Item and Global Members
[alert] Called by Application.alert().
[alert] Called by alert() -- Global Members.

Press <Enter> on console to exit.
```

图 5.10 “名字项与全局对象”的演示结果

5.3.2 属性和方法

属性和方法是对象的两个基本特性。在脚本程序中，我们可以对对象的属性进行读取或设置、对方法进行调用。以一个媒体播放器为例子，我们可以通过播放器对象获取它的版本号、卖主信息，可以命令它对指定的媒体文件进行播放、暂停、停止，以及播放过程中的随机定位、获取当前的播放进度等等。作为本节的演示，待解析的脚本代码如下：

```
//
// TestingCase02.js
//

Debugger.addBlankLine(1);
Debugger.trace("Testing Case 2", "Methods and Properties");

//////// << 本节演示的待解析的脚本代码 >> ////////
// 读取MediaPlayer对象的属性
var msg;
msg  = "Media Player, Version " + MediaPlayer.majorVersion + "." +
MediaPlayer.minorVersion;
msg += ", Vendor: " + MediaPlayer.vendor;
Debugger.trace("Info", msg);
Debugger.trace("Info", "Current position: " + MediaPlayer.currentPosition);

// 调用MediaPlayer对象的方法
MediaPlayer.play("c:\\HelloWorld.mpg");

// 设置MediaPlayer对象的属性
MediaPlayer.currentPosition = 888;
Debugger.trace("Info", "Current position: " + MediaPlayer.currentPosition);

// 调用MediaPlayer对象的方法
MediaPlayer.pauseOn();
```

```
MediaPlayer.pauseOff();
MediaPlayer.Stop();
```

为了解析上述脚本代码，我们需要做如下实现：

- 使用 IDL 语言来描述 MediaPlayer 对象
- 实现 MediaPlayer 对象
- 在宿主程序中创建、管理 MediaPlayer 对象，并让它与名字项“MediaPlayer”关联

（此部分省略，请下载、查看本书配套代码）

本节演示的脚本解析执行结果如图 5.11：

```
[Testing Case 2] Methods and Properties
[Info] Media Player, Version 1.0, Vendor: HQ Tech
[Info] Current position: 0
[Function] MediaPlayer::play( c:\HelloWorld.mpg )
[Function] Seek to a new position: 888.
[Info] Current position: 888
[Function] MediaPlayer::pauseOn
[Function] MediaPlayer::pauseOff
[Function] MediaPlayer::stop

Press <Enter> on console to exit.
```

图 5.11 “属性和方法”的演示结果

5.3.3 创建对象并传递给脚本

名字项代表的是顶级对象。解析顶级对象已经不成问题了，但如何来解析顶级对象后面的次级对象呢，比如“Application.car.engine.brand”？大家知道，在 Active Scripting 中所有脚本对象都实现为自动化对象。理解这一点非常重要！这就要求我们在脚本程序访问次级对象时，让宿主程序返回一个该次级对象的 IDispatch 指针；另外，当脚本程序通过特定的方法显式地创建一个对象时，我们也应该返回这个对象的 IDispatch 指针。作为本节的演示，待解析的脚本代码如下：

```
//
// TestingCase03.js
//

Debugger.addBlankLine(1);
Debugger.trace("Testing Case 3", "pass C++ objects to scripts");

//////// << 本节演示的待解析的脚本代码 >> ////////
// 多级对象的引用
var whatEngine = Application.car.engine.brand;
Debugger.trace("Info", "Got value of Application.car.engine.brand: " + whatEngine);
Application.car.engine.run();
```

```
// 显式地创建一个新的对象
var newWheel = Application.car.createWheel("KUMHO", "black");
newWheel.adjustParameters("new", 1, 2);
```

上述脚本代码中，“Application”是个名字项，代表的是一个顶级对象，而 car、engine 都是次级对象，brand 是 engine 对象的一个属性，createWheel 是 car 对象提供的一个用于创建一个新的 wheel 对象的方法。为了解析上述脚本代码，我们需要做如下实现：

- 使用 IDL 语言来描述 Application 对象、Car 对象、Engine 对象、Wheel 对象
- 修改 Application 对象的实现：增加 car 属性
- 实现 Car 对象、Engine 对象、Wheel 对象

（此部分省略，请下载、查看本书配套代码）

本节演示的脚本解析执行结果如图 5.12：

```
[Testing Case 3] pass C++ objects to scripts
[Info] Got value of Application.car.engine.brand: ACTECO
[Function] The car engine starts to run...
[Function] Parameters adjusted to: new, 1, 2.

Press <Enter> on console to exit.
```

图 5.12 “创建对象并传递给脚本”的演示结果

5.3.4 接受脚本中的对象

宿主程序中的对象可以传递给脚本，反过来脚本中的对象也可以传递给宿主程序。记住这一点：无论以哪种方式传递，Active Scripting 中的对象都是自动化对象，它们都要用 IDispatch 指针来标识。作为本节的演示，待解析的脚本代码如下：

```
//
// TestingCase04.js
//

Debugger.addBlankLine(1);
Debugger.trace("Testing Case 4", "pass scripting objects to the C++ side");

//////// << 本节演示的待解析的脚本代码 >> ////////
// 通过宿主程序提供的方法来创建对象
var kumhoWheel = Application.car.createWheel("KUMHO", "black");
// 将对象传递给宿主程序
Application.car.installWheel(kumhoWheel, 0);

////////////////// 另一种创建JavaScript对象的方式 ////////////////////////
// 定义一个JavaScript对象
function Wheel(brand, color)
```

```
{
        // 为Wheel对象的各个属性赋值
        this.brand = brand;
        this.color = color;
}

// 定义Wheel对象的各个成员函数
Wheel.prototype.rotate = function()
{
        Debugger.trace("Function", "Wheel.prototype.rotate called.");
}

Wheel.prototype.adjustParameters = function(param1, param2, param3)
{
        var msg = "Wheel.prototype.adjustParameters called. Adjusted to: ";
        msg += param1 + ", " + param2 + ", " + param3;
        Debugger.trace("Function", msg);
}

// 用JavaScript的new操作符创建一个Wheel对象
var jsWheel = new Wheel("MICHELIN", "gray");
// 将上述Wheel对象传递给宿主程序
Application.car.installWheel(jsWheel, 1);
```

上述脚本代码演示了创建 JavaScript 对象的两种方法：一种是通过 car 对象的 createWheel 方法，另一种是完全 JavaScript 的方法（先用脚本定义一个对象类，然后用 new 操作符来创建）。Wheel 对象创建好之后，通过 car 对象的 installWheel 方法可以把它传递给宿主程序。为了解析上述脚本代码，我们需要做如下实现：

- 使用 IDL 语言来描述 Car 对象
- 修改 Car 对象的实现：增加 installWheel 方法

（此部分省略，请下载、查看本书配套代码）

本节演示的脚本解析执行结果如图 5.13：

```
[Testing Case 4] pass scripting objects to the C++ side
[Function] Parameters adjusted to: max, 6, 8.
[Function] The 0th wheel is installed.
[Function] Wheel.prototype.adjustParameters called. Adjusted to: max, 6, 8
[Function] The 1th wheel is installed.

Press <Enter> on console to exit.
```

图 5.13 “接受脚本中的对象”的演示结果

5.3.5 数组的解析

一辆汽车通常有四个轮子。脚本程序如果要访问各个轮子的属性，宿主程序势必要在实现汽车对象的轮子属性时使用到数组。因此，我们在这一节中将给 Car 对象添加一个 Wheel 对象的数组。那么如何来实现数组的解析呢？作为本节的演示，待解析的脚本代码如下：

```
//
// TestingCase05.js
//

Debugger.addBlankLine(1);
Debugger.trace("Testing Case 5", "interpret an Array");

//////// << 本节演示的待解析的脚本代码 >> ////////
var wheelCount = Application.car.wheels.length;
Debugger.trace("Info", "There're " + wheelCount + " wheels on this car.");

// 使用一个循环来遍历数组中的各个Wheel对象
for (var i = 0; i < wheelCount; i++)
{
        var brand = Application.car.wheels[i].brand;
        var color = Application.car.wheels[i].color;
       Debugger.trace("Info", "Index: " + i + " Brand: " + brand + " Color: " + color);
}
```

大家知道，JavaScript 数组也是一种对象，因此在 Active Scripting 中同样要用 IDispatch 接口来标识。当脚本程序中通过下标访问 Application.car.wheels 数组的元素时，如 Application.car.wheels[0]，实际上是在访问数组对象的名字为“0”的属性。理解了这一点，解析 JavaScript 数组就比较容易了。为了更方便宿主程序方数组对象的实现，我们设计了一个新的模板类：IDispatchArrayImpl。在这之前，考虑到 IDispatch 接口实现时很多函数都是相似的，因此提取了一个公共的基类：IDispatchBaseImpl。

（此部分省略，请下载、查看本书配套代码）

为了支持对 Wheel 对象数组的解析，我们再做如下实现：

- 使用 IDL 语言来描述 Car 对象
- 修改 Car 对象的实现：增加 wheels 属性
- 实现 Wheel 数组对象

（此部分省略，请下载、查看本书配套代码）

本节演示的脚本解析执行结果如图 5.14：

```
[Testing Case 5] interpret an Array
[Info] There're 4 wheels on this car.
[Info] Index: 0 Brand: KUMHO-100 Color: black1
[Info] Index: 1 Brand: KUMHO-200 Color: black2
[Info] Index: 2 Brand: KUMHO-300 Color: black3
[Info] Index: 3 Brand: KUMHO-400 Color: black4

Press <Enter> on console to exit.
```

图 5.14 “数组的解析”的演示结果

5.3.6 异常处理

我们首先来看这么一段 JavaScript 异常处理的代码：

```
try
{
        var a = b; // 抛出异常的代码行

        // 下面的代码因为异常的抛出而不被执行！
        Debugger.trace("Info", "After the statement which throws an exception.");
}
catch (e)
{
        // 捕捉到异常后进行处理
        Debugger.trace("Info", e + " " + e.message + ", code: " + e.number);
}
finally
{
        // 异常处理后执行的代码
        Debugger.trace("Info", "Reach finally block.");
}
```

上述脚本代码中，由于变量 b 是未定义的，把 b 赋值给变量 a 将会抛出一个异常。因此，try 块内紧跟在上述赋值语句后面的代码将不会被执行，转而执行 catch 块内的代码，最后再执行 finally 块内的代码。

JavaScript 内建了很多错误对象来表示不同类型的异常，它们包括 EvalError、RangeError、ReferenceError、SyntaxError、TypeError、URIError 等等。我们可以通过下面的脚本代码来测试：

```
// 识别不同的异常类型
function DistinguishErrorType(e)
{
        if (e instanceof EvalError)
        {
                Debugger.trace("Info", "catch a built-in EvalError.");
        }
        else if (e instanceof RangeError)
        {
```

```
            Debugger.trace("Info", "catch a built-in RangeError.");
        }
        else if (e instanceof ReferenceError)
        {
            Debugger.trace("Info", "catch a built-in ReferenceError.");
        }
        else if (e instanceof SyntaxError)
        {
            Debugger.trace("Info", "catch a built-in SyntaxError.");
        }
        else if (e instanceof TypeError)
        {
            Debugger.trace("Info", "catch a built-in TypeError.");
        }
        else if (e instanceof URIError)
        {
            Debugger.trace("Info", "catch a built-in URIError.");
        }
        else
        {
            Debugger.trace("Info", "catch a normal Error.");
        }
}

try
{
        // 测试JavaScript内建的各种异常类型
        var err;
//      err = new Error("built-in (Error)");
//      err = new EvalError("built-in (EvalError)");
//      err = new RangeError("built-in (RangeError)");
//      err = new ReferenceError("built-in (ReferenceError)");
//      err = new SyntaxError("built-in (SyntaxError)");
//      err = new TypeError("built-in (TypeError)");
        err = new URIError("built-in (URIError)");
        err.number = 121;
        throw err;
}
catch (e)
{
        // 识别捕获到的异常类型
        DistinguishErrorType(e);
        Debugger.trace("Info", e + " " + e.message + ", code: " + e.number);
}
```

在使用 Active Scripting 进行脚本解析的过程中，宿主程序也会产生异常。但宿主程序产生的异常是 C++类型的，怎样才能将 C++类型的异常转换成 JavaScript 中的异常、进而抛到脚本程序中去呢？这其实不难做到。因为 IDispatch::Invoke 函数的 pexcepinfo 参数就是为这种应用而设计的。它是一个 EXCEPINFO 类型的数据结构，定义如下：

```
typedef struct tagEXCEPINFO {
    WORD  wCode;             // 错误码（数字形式）
    WORD  wReserved;
    BSTR  bstrSource;
    BSTR  bstrDescription;   // 错误描述（字符串形式）
    BSTR  bstrHelpFile;
    DWORD dwHelpContext;
    PVOID pvReserved;
    HRESULT (__stdcall *pfnDeferredFillIn)(struct tagEXCEPINFO *);
    SCODE scode;
} EXCEPINFO, * LPEXCEPINFO;
```

我们只需在 IDispatch::Invoke 函数的实现中捕获 C++异常，然后在 pexcepinfo 指向的 EXCEPINFO 结构中填写好对应的描述信息，并且让 Invoke 函数返回 DISP_E_EXCEPTION 就可以了。作为本节的演示，我们编写了如下的待解析的脚本代码：

```
//
// TestingCase06.js
//

Debugger.addBlankLine(1);
Debugger.trace("Testing Case 6", "Exception handling");

function UNMASK(number)
{
        return(number&0xffff);
}

//////// << 本节演示的待解析的脚本代码 >> ////////
try
{
        Application.triggerException(); // 此语句将抛出一个异常
}
catch (e)
{
        Debugger.trace("Info", e + " " + e.message);
        Debugger.trace("Info", "Catch an exception, error code: " + UNMASK(e.number));
}
```

提示：由于 Active Scripting 传递到脚本程序的错误码是 HRESULT 类型的，因此上面代码中使用一个自定义的 UNMASK 函数来另外解析错误码（e.number）。

为了解析上述脚本代码，我们需要做如下实现：

- 使用 IDL 语言来描述 Application 对象
- 实现一个自定义的异常类 CHQException
- 修改 Application 对象的实现：增加 triggerException 方法

- 修改 IDispatch::Invoke 函数的实现

（此部分省略，请下载、查看本书配套代码）

本节演示的脚本解析执行结果如图 5.15：

```
[Testing Case 6] Exception handling
[Info] [object Error] 'b' is undefined, code: -2146823279
[Info] Reach finally block.
[Info] catch a built-in URIError.
[Info] [object Error] built-in (URIError), code: 121
[Info] [object Error] Can you catch me?
[Info] Catch an exception, error code: 98

Press <Enter> on console to exit.
```

图 5.15 “异常处理”的演示结果

5.3.7 传递一个 null 参数

前面讲过，脚本程序与宿主程序之间传递的对象都是通过 IDispatch 指针来标识的。但如果对象参数需要赋值为 null，这种方法就不适合了！比如 5.3.4 节中实现的 installWheel 方法，它原先的功能定义是安装一个轮子；现在要增加新的功能定义：如果轮子对象参数被指定为 null，则要将对应索引的轮子卸下来。那就要执行这样的脚本代码：Application.car.installWheel(null, 0)，但遗憾的是程序会抛出一个异常——找不到跟当前参数类型匹配的 installWheel 方法。也就是说，null 跟 IDispatch 类型的指针是不匹配的！这种情况下，我们应该把对象参数的类型改成 VARIANT（因为 VARIANT 是万能的！）。作为本节的演示，待解析的脚本代码如下：

```
//
// TestingCase07.js
//

Debugger.addBlankLine(1);
Debugger.trace("Testing Case 7", "pass a null parameter");

//////// << 本节演示的待解析的脚本代码 >> ////////
try
{
        // 以前的实现（不能接受null参数）
        Application.car.installWheel(null, 0); // 这行代码将抛出异常
}
catch (e)
{
        Debugger.trace("Exception", "Failed to uninstall the wheel.");
        Debugger.trace("Exception", e + " " + e.message);
}
```

```
// 新的实现（能够接受null参数）
Application.car.installWheel2(null, 0);
```

为了解析上述脚本代码，我们需要做如下实现：

- 使用 IDL 语言来描述 Car 对象
- 修改 Car 对象的实现：增加 installWheel2 方法

（此部分省略，请下载、查看本书配套代码）

本节演示的脚本解析执行结果如图 5.16：

```
[Testing Case 7] pass a null parameter
[Exception] Failed to uninstall the wheel.
[Exception] [object Error] Type mismatch
[Function] The 0th wheel is uninstalled.

Press <Enter> on console to exit.
```

图 5.16 “传递一个 null 参数”的演示结果

5.3.8 自动类型转换

我们首先来看下面的这么一段 JavaScript 代码：

```
// 测试JavaScript内建的对象
var currentDate = new Date();
Debugger.trace("Info", currentDate);

// 测试自定义的JavaScript对象
function Color(r, g, b)
{
        this.r = r;
        this.g = g;
        this.b = b;
}

Color.prototype.toString = function()
{
        return "RGB(" + this.r + "," + this.g + "," + this.b + ")";
}

// 创建一个Color对象的实例
var myColor = new Color(128, 128, 128);
Debugger.trace("Info", myColor);
```

上述脚本程序中，Debugger.trace 方法的第 2 个参数要求的数据类型是字符串。但事实上我们传递给它的都是对象引用，类型能匹配吗？能！Active Scripting 会隐式地调用对象

的 toString 函数，获得该对象的字符串描述之后再调用 Debugger.trace 方法。这就是所谓的自动类型转换。那么，如何让我们在宿主程序中创建的对象也支持自动类型转换功能呢？关键就是要实现 toString 函数！作为本节的演示，待解析的脚本代码如下：

```
//
// TestingCase08.js
//

Debugger.addBlankLine(1);
Debugger.trace("Testing Case 8", "automatic type conversion");

//////// << 本节演示的待解析的脚本代码 >> ////////
try
{
        // wheel1对象不支持自动类型转换
        var wheel1 = Application.car.createWheel("KUMHO", "red");
        Debugger.trace("Info", wheel1); // 这行代码将抛出异常
}
catch (e)
{
        Debugger.trace("Exception", e.message);
}

try
{
        // wheel2对象支持了自动类型转换！
        var wheel2 = Application.car.createEnhancedWheel("KUMHO", "red");
        Debugger.trace("Info", wheel2);
}
catch (e)
{
        Debugger.trace("Exception", e.message);
}
```

上述脚本程序中，通过 car 对象的 createWheel 方法创建的对象不支持自动类型转换，而通过 createEnhancedWheel 方法创建的对象支持自动类型转换。与 Wheel 对象相比，EnhancedWheel 对象的不同在于：它额外实现了一个 IJSObject 接口（IJSObject 接口描述了 JavaScript 为对象专门规定的一些基本属性和方法）。于是 EnhancedWheel 对象具有了 JavaScript 语言所定义的对象应该具有的共性。为了实现这个 EnhancedWheel 对象，我们这里需要一个新的模板类：IDispatchImpl2。IDispatchImpl2 类的实现跟 IDispatchImpl 类非常相似，只不过多支持了一个 IJSObject 接口。

（此部分省略，请下载、查看本书配套代码）

为了正确解析本节的演示脚本，我们还需要做如下实现：

- 使用 IDL 语言来描述 EnhancedWheel 对象、Car 对象
- 实现 EnhancedWheel 对象

- 修改 Car 对象的实现：增加 createEnhancedWheel 方法

（此部分省略，请下载、查看本书配套代码）

本节演示的脚本解析执行结果如图 5.17：

```
[Testing Case 8] automatic type conversion
[Info] Mon May 8 16:22:05 UTC+0800 2006
[Info] RGB(128,128,128)
[Exception] Type mismatch
[Info] [object EnhancedWheel]

Press <Enter> on console to exit.
```

图 5.17 “自动类型转换”的演示结果

5.3.9 回调脚本函数

在很多场合下都需要用到回调函数，比如当宿主程序方的一个异步操作完成时要通知脚本程序，或者宿主程序需要向脚本程序同时传递 2 个或 2 个以上的参数。大家已经知道，JavaScript 中的函数也是一种对象，因此在 Active Scripting 中也用 IDispatch 指针来标识。作为本节的演示，待解析的脚本代码如下：

```
//
// TestingCase09.js
//

Debugger.addBlankLine(1);
Debugger.trace("Testing Case 9", "call back to scripts");

//////// << 本节演示的待解析的脚本代码 >> ////////
// 通过getInformation方法查询"weather"的值
// 查询的结果通过回调函数的方式传回脚本程序
Application.getInformation("weather", myCallback);

// 回调函数，在getInformation退出之前会被调用
// 参数：key - 查询的关键字
//       value - 查询获得的key的值
//       errorCode - 查询过程中发生的错误类型
function myCallback(key, value, errorCode)
{
        Debugger.trace("Function", "script myCallback() called.");
        Debugger.trace("Info", "key: " + key + ", value: " + value + ", error code: " +
errorCode);
}
```

为了解析上述脚本代码，我们需要做如下实现：

- 使用 IDL 语言来描述 Application 对象
- 修改 Application 对象的实现：增加 getInformation 方法

（此部分省略，请下载、查看本书配套代码）

本节演示的脚本解析执行结果如图 5.18：

```
[Testing Case 9] call back to scripts
[Function] script myCallback() called.
[Info] key: weather, value: sunny, error code: 1

Press <Enter> on console to exit.
```

图 5.18 “回调脚本函数”的演示结果

5.3.10 访问脚本的属性和方法

不知道大家是否注意过 IActiveScript::GetScriptDispatch 这个接口函数？这个函数返回一个 IDispatch 指针，但这个指针有什么用呢？这就是本节将要介绍的内容——宿主程序可以利用这个 IDispatch 指针，访问脚本程序中定义的变量或者调用脚本函数。作为本节的演示，待解析的脚本代码如下：

```
//
// TestingCase10.js
//

Debugger.addBlankLine(1);
Debugger.trace("Testing Case 10", "set/get value to script variables");
Debugger.trace("Testing Case 10", "invoke script functions");

//////// << 本节演示的待解析的脚本代码 >> ////////
var BOOK_TITLE = "A Practical Guide to Script-Driven Software Development";
var bookCount  = 5000;

function showMessage(msg)
{
        Debugger.trace("showMessage", msg);
}

// 宿主程序在reviewScripts方法的实现中演示如何访问脚本的属性和方法
Application.reviewScripts();
```

为了解析上述脚本代码，我们应该做如下实现：

- 使用 IDL 语言来描述 Application 对象
- 修改 Application 对象的实现：增加 reviewScripts 方法

（此部分省略，请下载、查看本书配套代码）

本节演示的脚本解析执行结果如图 5.19：

```
[Testing Case 10] set/get value to script variables
[Testing Case 10] invoke script functions
[Info] BOOK_TITLE: A Practical Guide to Script-Driven Software Development.
[Info] bookCount's value: 5000.
[Info] bookCount's new value: 6000.
[showMessage] This is the parameter passed to scripts.

Press <Enter> on console to exit.
```

图 5.19 “访问脚本的属性和方法”的演示结果

5.3.11 多线程问题及其解决方案

对于一些很费时间的操作（比如文件拷贝），我们往往会使用另外的一条线程去做。当这些费时的操作完成时，再通过一个回调函数去通知脚本程序。作为本节的演示，待解析的脚本代码如下：

```
//
// TestingCase11.js
//

Debugger.addBlankLine(1);
Debugger.trace("Testing Case 11", "multi-thread issue for script interpreting");

//////// << 本节演示的待解析的脚本代码 >> ////////
// 将文件“C:\a.js”拷贝到“C:\b.js”，
// 拷贝完成后调用fileCopyComplete函数通知脚本程序
Application.copyFile("C:\\a.js", "C:\\b.js", fileCopyComplete);

// 回调函数，当文件拷贝完成时被调用
// 参数：srcFile  -  查询的关键字
//       dstFile  -  查询获得的key的值
//       status  -  本次操作引起的状态改变
function fileCopyComplete(srcFile, dstFile, status)
{
        var msg = "Copy " + srcFile + " to " + dstFile + " completed. "
        msg += "Status: " + status + ".";
        Debugger.trace("Testing Case 11", msg);
}
```

为了解析上述脚本代码，我们先来做如下实现：

- 使用 IDL 语言来描述 Application 对象
- 实现一个通用的线程类 CHQThread

- 实现一个文件拷贝线程 CFileCopyThread
- 修改 Application 对象的实现：增加 copyFile 方法

（此部分省略，请下载、查看本书配套代码）

遗憾的是，上述实现中回调脚本函数这一步是失败的！原因是，创建脚本引擎的线程（更确切地说是调用 IActiveScript::SetScriptSite 的线程，以下简称引擎关联线程），跟执行文件拷贝的线程不是同一条线程；在非引擎关联线程中调用脚本引擎的功能服务是不允许的。这也可以说是 Active Scripting 在实现上的一个限制！

如何来解决这个问题呢？其实也很简单，就是要将脚本函数的回调统统放到引擎关联线程中去做。因此我们使用了一个 CCallbackManager 对象专门来保存和处理回调请求；文件拷贝完成时不直接调用回调函数，而是把回调请求交给 CCallbackManager 对象保存，以等待引擎关联线程的统一处理。

为了成功解析本节的演示脚本，我们还需要做如下实现：

- 实现一个回调请求管理器 CCallbackManager
- 修改 CFileCopyThread::Process 函数的实现
- 修改在 JSInterpreting 程序主线程中的实现

（此部分省略，请下载、查看本书配套代码）

本节演示的脚本解析执行结果如图 5.20：

```
[Testing Case 11] multi-thread issue for script interpreting
[Async Info] File Copy callback has been put into the queue.

Press <Enter> on console to exit.
[Testing Case 11] Copy C:\a.js to C:\b.js completed. Status: 1.
```

图 5.20 “多线程问题及其解决方案”的演示结果

5.3.12 支持定时器

有时候脚本程序需要一种定时机制——让宿主程序能够按照一定的时间间隔去执行一个特定的脚本函数。（如果你曾经使用 JavaScript 为网页设计过跑马灯效果，你就能明白定时器有多么有用！）作为本节的演示，待解析的脚本代码如下：

```
//
// TestingCase12.js
//

Debugger.addBlankLine(1);
Debugger.trace("Testing Case 12", "support some kind of timer for scripts");

//////// << 本节演示的待解析的脚本代码 >> ////////
// 创建一个定时器，要求1000ms之后执行，并且只执行一次
```

```
Application.createTimer(1000, false, timerProc1);
// 创建一个定时器，要求每2000ms执行一次
Application.createTimer(2000, true, timerProc2);

// 定时器执行的函数
function timerProc1()
{
        Debugger.trace("Info", "timerProc1() is executed!");
}

// 定时器执行的函数
function timerProc2()
{
        Debugger.trace("Info", "timerProc2() is executed!");
}
```

为了解析上述脚本代码，我们需要做如下实现：

- 使用 IDL 语言来描述 Application 对象、Timer 对象
- 实现 Timer 对象
- 修改 Application 对象的实现：增加 createTimer 方法
- 实现一个 Timer 管理器 CTimerManager
- 修改在 JSInterpreting 程序主线程中的实现

（此部分省略，请下载、查看本书配套代码）

本节演示的脚本解析执行结果如图 5.21：

```
[Testing Case 12] support some kind of timer for scripts

Press <Enter> on console to exit.
[Info] timerProc1() is executed!
[Info] timerProc2() is executed!
[Info] timerProc2() is executed!
[Info] timerProc2() is executed!
```

图 5.21 “支持定时器”的演示结果

5.3.13 支持动态属性

前面介绍的脚本对象，其属性和方法都是预先在 IDL 文件中定义好的，是静态的。以 Wheel 对象为例，它的属性“brand”和“color”都在 js_demo.idl 文件中预先进行了描述，在脚本程序中访问这些属性自然也不成问题。但如何让一个对象支持动态的属性呢？比如 Wheel 对象本来没有“price”属性，如何动态创建（或者删除）这个属性，以便脚本程序对这个属性进行赋值或者读取呢？作为本节的演示，待解析的脚本代码如下：

```
//
```

```
// TestingCase13.js
//

Debugger.addBlankLine(1);
Debugger.trace("Testing Case 13", "dynamic properties (IDispatchEx)");

//////// << 本节演示的待解析的脚本代码 >> ////////
try
{
        // （以前的实现）Wheel对象不支持动态属性
        var wheel1 = Application.car.createWheel("KUMHO", "red");
        wheel1.price = 500; // 这行代码会抛出异常
}
catch (e)
{
        Debugger.trace("Exception", e.message);
}

// 新实现的DynamicWheel对象支持动态属性
var wheel2 = Application.car.createDynamicWheel("KUMHO", "red");
Debugger.trace("Info", "The wheel's brand is " + wheel2.brand);
Debugger.trace("Info", "The wheel's color is " + wheel2.color);
Debugger.trace("Info", "The wheel's (original) price is " + wheel2.price);
wheel2.price = 500;
Debugger.trace("Info", "The wheel's (new) price is " + wheel2.price);
```

支持动态属性的对象光靠 IDispatch 接口来实现恐怕是不行了。该 IDispatchEx 出场了！为此，我们实现了一个新的模版类：IDispatchExImpl。

（此部分省略，请下载、查看本书配套代码）

为了成功解析本节的演示脚本，我们还需要做如下实现：

- 使用 IDL 语言来描述 Car 对象、IDynamicWheel 接口
- 实现 DynamicWheel 对象
- 修改 Car 对象的实现：增加 createDynamicWheel 方法

（此部分省略，请下载、查看本书配套代码）

本节演示的脚本解析执行结果如图 5.22：

```
[Testing Case 13] dynamic properties (IDispatchEx)
[Exception] Object doesn't support this property or method
[Info] The wheel's brand is KUMHO
[Info] The wheel's color is red
[Info] The wheel's (original) price is undefined
[Info] The wheel's (new) price is 500

Press <Enter> on console to exit.
```

图 5.22 “支持动态属性”的演示结果

5.3.14 脚本的单步调试

使用 Active Scripting 解析的脚本程序是可以被单步调试的，条件是要在宿主程序上实现一个 IActiveScriptSiteDebug 接口（当然还要一个脚本调试器工具的配合）。脚本程序在被单步执行的情况下，我们更容易开展纠错工作。作为本节的演示，待解析的脚本代码如下：

```
//
// TestingCase14.js
//

Debugger.addBlankLine(1);
Debugger.trace("Testing Case 14", "step into scripts for debugging");

//////// << 本节演示的待解析的脚本代码 >> ////////
// 将应用程序暂停住，以便先在脚本调试器上做一些设置
pauseForDebugger();

var curDate = new Date();
var msg = "Today is ";
msg += (curDate.getMonth() + 1) + "/";
msg += curDate.getDate() + "/";
msg += curDate.getYear() + ".";
Debugger.trace("Info", msg);

Debugger.trace("Info", NaN);
Debugger.trace("Info", undefined);
Debugger.trace("Info", null);
```

为了成功演示脚本的单步调试，我们需要做如下实现：

- 使用 IDL 语言来描述 Application 对象
- 修改 Application 对象的实现：增加 pauseForDebugger 方法
- 为宿主程序实现 IActiveScriptSiteDebug 接口

```
// <1> 使用 IDL 语言来描述 Application 对象
//
// js_demo.idl
//

[
        object,
        uuid(E74835F6-EBDD-49b2-986D-234B70AF74E4),
        dual
```

```
]
interface IApplication : IDispatch
{
        // 增加的一个方法：将应用程序暂停住
        HRESULT pauseForDebugger();
}


// <2> 修改 Application 对象的实现：增加 pauseForDebugger 方法
//
// CApplication.cpp
//

// 等待用户按下键盘上的“C”键
STDMETHODIMP CApplication::pauseForDebugger()
{
        printf("Press the key 'C' on console to continue the script running.\n");
        while (GetAsyncKeyState('C') == 0)
        {
                Sleep(100);
        }
        return S_OK;
}
```

提示：为什么要设计一个 pauseForDebugger 函数？这里主要考虑到程序执行的一个时间差问题。pauseForDebugger 函数放在用户想要调试的脚本代码之前，当脚本程序执行到这个函数的时候就暂停了。这时候用户就有时间去脚本调试器上做一些设置。然后用户在键盘上按下“C”键，pauseForDebugger 函数返回了，该函数后面的脚本代码也就能在脚本调试器中被单步执行了。

```
// <3> 为宿主程序实现 IActiveScriptSiteDebug 接口
//
// CJScriptHost.h
//

#define __Script_Debug__

#ifdef __Script_Debug__
#include "activdbg.h"
#endif

class CJScriptHost : public IActiveScriptSite
                     #ifdef __Script_Debug__
                      , public IActiveScriptSiteDebug
                     #endif
{
public:
        CJScriptHost();
        ~CJScriptHost();
```

```
#ifdef __Script_Debug__
        // --- IActiveScriptSiteDebug接口的各个成员函数 ---
        STDMETHODIMP GetDocumentContextFromPosition(/*[in]*/ DWORD dwSourceContext,
                /*[in]*/ ULONG uCharacterOffset, /*[in]*/ ULONG uNumChars,
                /*[out]*/ IDebugDocumentContext** ppsc);
        STDMETHODIMP GetApplication(/*[out]*/ IDebugApplication **ppda);
        STDMETHODIMP GetRootApplicationNode(/*[out]*/ IDebugApplicationNode**
ppdanRoot);
        STDMETHODIMP OnScriptErrorDebug(/*[in]*/ IActiveScriptErrorDebug *pErrorDebug,
                /*[out]*/ BOOL *pfEnterDebugger,/*[out]*/ BOOL
*pfCallOnScriptErrorWhenContinuing);
#endif

private:
#ifdef __Script_Debug__
        IProcessDebugManager*       mpDebugMgr;    // 调试管理器
        IDebugApplication*          mpDebugApp;    // 调试应用
        IDebugDocumentHelper*       mpDebugDoc;    // 调试文档
        DWORD                       mAppCookie;    // 调试应用的标识

        // 创建脚本调试器
        HRESULT CreateScriptDebugger();
        // 销毁脚本调试器
        void ReleaseScriptDebugger();
        // 将脚本代码创建为一个文档对象，便于调试
        HRESULT CreateDocumentForDebugger(BSTR scripts);
        // 释放调试文档
        void ReleaseDebugDocument();
#endif
};


//
// CJScriptHost.cpp
//
```

（此部分省略，请下载、查看本书配套代码）

提示：为了让宿主程序支持脚本调试，开发时需要包含头文件 activdbg.h。另外，需要在项目设置中连接一个库文件 adl.lib，否则会导致如下的连接错误：

```
CJScriptHost.obj : error LNK2001: unresolved external symbol
_IID_IActiveScriptSiteDebug
CJScriptHost.obj : error LNK2001: unresolved external symbol
_IID_IProcessDebugManager
.\JSInterpreting_d.exe : fatal error LNK1120: 2 unresolved externals
```

下面，我们来演示脚本程序单步执行的整个过程：

1. 到微软公司的网站上下载一个脚本调试器（Script Debugger），网址是 http://msdn.microsoft.com/library/default.asp?url=/downloads/list/webdev.asp。下载完毕后可以得到一个安装程序：scd10en.exe，并在本地机器上安装。
2. 运行脚本调试器（一个名为 msscrdbg.exe 的可执行文件），并且执行它的菜单命令：View | Running Documents，如图 5.23：

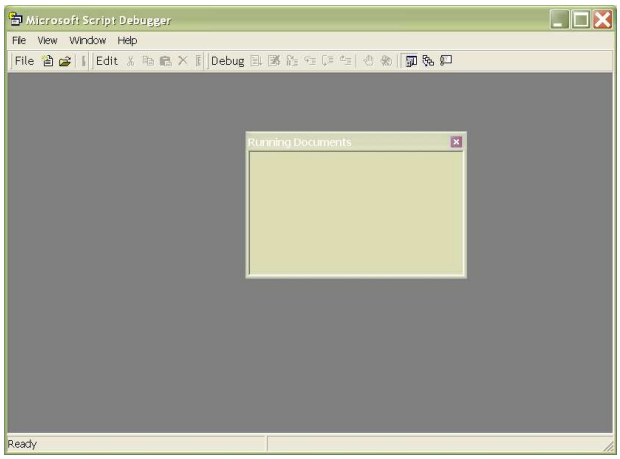

图 5.23 脚本调试器的运行界面 1

3. 在 VC 的开发环境中运行 JSInterpreting 程序。当该程序执行到脚本函数 pauseForDebugger 时将被阻塞住。
4. 切换到脚本调试器。这时在“Running Documents”窗口中就可以看到正在被执行的脚本文件了。双击它，脚本文件的内容将显示出来，如图 5.24：

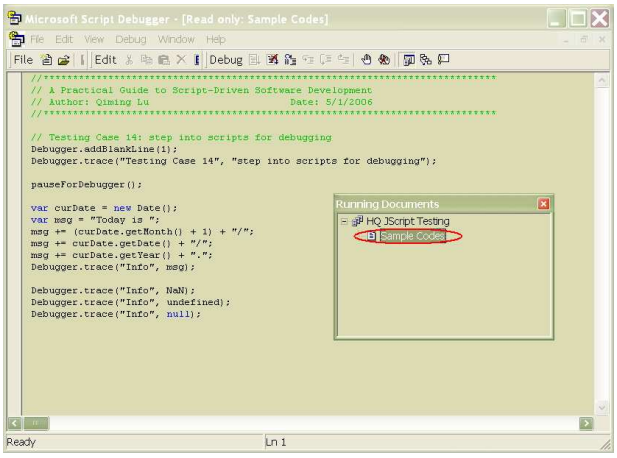

图 5.24 脚本调试器的运行界面 2

5. 仍然在脚本调试器中，鼠标选中“Running Documents”中的“Sample Codes”，并点击右键，在弹出的快捷菜单中执行“Break At Next Statement”。
6. 切换到 JSInterpreting 程序。在键盘上按下“C”键让程序继续往下执行。随后脚本调试器会自动获得焦点，当前将要执行的脚本代码行会被高亮显示，如图 5.25：

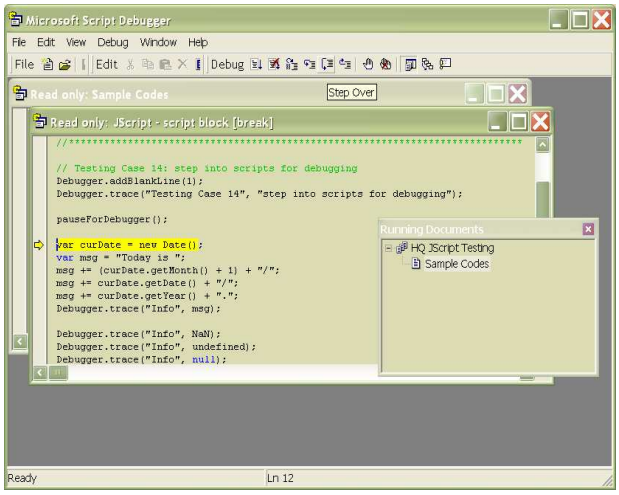

图 5.25 脚本调试器的运行界面 3

7. 使用脚本调试器工具条上的一系列按钮：“Step Into”、“Step Over”、“Step Out”等，就可以实现脚本程序的单步执行了。也可以在某一行代码上设置一个断点，让程序直接运行到该断点处。总之，跟 VC 的程序调试方法很相似。

本节演示的脚本解析执行结果如图 5.26：

```
[Testing Case 14] step into scripts for debugging
Press the key 'C' on console to continue the script running.
[Info] Today is 5/9/2006.
[Info] 1.#QNAN
[Info]

Press <Enter> on console to exit.
```

图 5.26 “脚本的单步调试”的演示结果

第 6 章 脚本驱动的看图软件

6.1 脚本驱动的意义

在软件的交互性不断提高的今天，仅仅提供菜单或工具条的界面已经不能满足用户的需要了；软件的可定制特性已经成为当今软件的一项基本特征，对于一些通用软件尤为如此。那么，本书引入的脚本驱动的软件开发方法又有什么意义呢？

使用脚本驱动的软件开发方法，程序员可以彻底摆脱 UI 艺术设计。大家知道，UI 对于一款软件来说是非常重要的！要把 UI 做好也是很不容易的！软件的 UI 应该是帮助用户更好地使用该软件提供的功能或服务的，因此好的 UI 应该是漂亮的、整体风格一致的、符合用户的使用习惯的。很多程序员说，我的审美能力太差了，设计专业的 UI 恐怕已经超出了我的能力范围。现在不怕了！其实，UI 完全可以交给艺术设计师来做（这也正是他们的特长）。程序员与 UI 设计师之间有着不同的分工，并且最终通过脚本来进行整合；大家各施其职，各尽其才，何乐而不为呢？

除了程序员与 UI 设计师之间的分工之外，程序员内部还可以这么分：专门开发商业逻辑的程序员和专门开发脚本程序的程序员。他们使用不同的程序设计语言，他们对各自使用的语言都非常精通，他们都在为同一款软件提供不同的支持——脚本程序可以用来定义 UI 元素、实现 UI 逻辑，可以很灵活地跟用户实现交互；这使得其他更多的程序员能够集中精力于商业逻辑的设计和开发。这样的分工是合理的，是富有生产力的。不同专长的程序员专注于他们各自擅长的领域，个人兴趣与公司业务有效地结合起来，更利于个人的成功。而这种个人成功，带来的同样是公司业务的成功。

脚本驱动的软件开发方法细化了软件开发过程中的分工，因此非常适合于基于团队的大型软件开发。让分工责权清晰，让合作的渠道畅通无阻，软件开发的整体效率就有了保障。面对不断变化的软件需求，我们可以做得更加从容。

脚本驱动的软件最大的特点还在于它在可定制性和交互性方面的出色表现。因为针对一个特定的商业领域，我们可能已经开发了非常多、非常细的功能（或组件）。然而，这么多功能并不是统统放在一个软件中、一股脑儿地推给所有用户的。这并不是一种成功的方式！因为用户有不同的分类，不同的用户往往有着不同的需求。我们必须根据不同的用户需求为软件组合不同的功能集，表现出一种专业的个性化服务，以获得较高的用户满意度反馈。通过特定的脚本规范，以及脚本程序的交互控制能力，这种软件的定制其实很容易就能做到。

脚本驱动的软件到底还有多少好处呢？还是请读者在本书提供的不同版本的看图软件之间进行细细的比较吧！认真体会从 ImageViewer_Basic 到 ImageViewer_Skinned、ImageViewer_Customized 乃至 ImageViewer_ScriptDriven 的演变过程，相信读者定会有所感悟。

6.2 脚本驱动的实现

脚本驱动的实现，关键是要让我们的程序具备脚本解析的能力。通过本书第 5 章的学习，相信读者已经熟知了使用 Active Scripting 技术进行 JavaScript（或 VBScript）解析的方法。下面，我们仍将以看图软件为例，让我们的软件真正地被脚本驱动起来！

6.2.1 制定脚本接口标准

脚本程序怎样来跟应用程序进行交互呢？具体来说，应用程序开放给脚本程序哪些对象、哪些接口函数呢？这需要制定一个标准，要有详细的文档说明，并且在开发商业逻辑的程序员与开发脚本程序的程序员之间达成一致。

以我们的看图软件为例，我们开放给脚本程序一个顶级的 Application 对象。这个 Application 对象表示看图软件这个主应用程序，它包含一个 mainWindow 子对象来代表主窗口，包含一个 imageProcessor 子对象来代表图像处理器组件（软件实现的商业逻辑都

要以组件的形式提供服务），还包含一些控制函数，比如获取一个图像文件、保存一个图像文件、最小化主窗口、关闭应用程序等等。其中，mainWindow 对象还包含了一个 buttons 数组，便于脚本程序访问主窗口中的各个按钮；imageProcessor 对象开放了对图像进行各种操作的接口。

作为脚本接口的说明书，应该有一个规范的描述文档。因为脚本程序员是直接根据这个文档来开发脚本程序的，因此文档的规范性和准确性非常重要！而从程序的功能实现来看，我们也会使用 IDL 语言来进行脚本对象的描述：

```
//
// ImageViewer.idl
//

import "oaidl.idl";
import "ocidl.idl";
import "dispex.idl"; // 定义了IDispatchEx

[
	uuid(8A1E2595-2B00-4b06-BE07-A3FAD50DDD9D), // 类型库的GUID
	version(1.0),
	helpstring("ImageViewer 1.0 Type Library")
]
library ImageViewerTypeLib
{
	importlib("stdole32.tlb");
	importlib("stdole2.tlb");


	interface IDOM2EventTarget;

	// 描述Application对象
	[
		object,
		uuid(FADBF3E7-73F5-4de6-8402-DDBD25307F0C),
		dual
	]
	interface IApplication : IDOM2EventTarget
	{
		////////////////////////// 属性 ////////////////////////
		// 获取软件的版本号
		[propget] HRESULT version([out, retval] BSTR* value);
		// 获取程序的主窗口对象
		[propget] HRESULT mainWindow([out, retval] IDispatch** ppdisp);
		// 获取程序的图像处理器组件对象
		[propget] HRESULT imageProcessor([out, retval] IDispatch** ppdisp);
		// 设置一个应用程序运行起来时的回调通知
		[propput] HRESULT onApplicationLoaded([in] IDispatch* pCallback);

		////////////////////////// 方法 ////////////////////////
		// 让用户选择一个文件，用于读取
```

```
        HRESULT getImageFile([in] IDispatch* pCallback);
        // 让用户选择一个文件，用于保存
        HRESULT saveImageFile([in] IDispatch* pCallback);
        // 刷新图像显示
        HRESULT refreshDisplay();
        // 显示“About”对话框
        HRESULT showAboutDialog();
        // 将主窗口最小化
        HRESULT minimizeMainWindow();
        // 退出应用程序
        HRESULT exitApplication();

        ////////////////////////// 常量定义 ////////////////////////
        // 表示操作成功的数值
        [propget] HRESULT SUCCEEDED([out, retval] UINT* value);
        // 表示操作失败的数值
        [propget] HRESULT FAILED([out, retval] UINT* value);
}

// 描述MainWindow对象
[
        object,
        uuid(CF3285CE-BD22-4ab4-B2B6-410D6F8FF14A),
        dual
]
interface IMainWindow : IDispatch
{
        ////////////////////////// 属性 ////////////////////////
        // 获取主窗口中的所有按钮
        [propget] HRESULT buttons([out, retval] IDispatch** ppdisp);
}

// 描述Button对象，给出按钮的属性以及对按钮的操作接口
[
        object,
        uuid(1AE5C12E-8B2E-4911-9053-EE5490A2BFC1),
        dual
]
interface IButton : IDispatch
{
        ////////////////////////// 属性 ////////////////////////
        // 获取按钮的标识
        [propget] HRESULT id([out, retval] BSTR* value);
        // 获取按钮的使用状态
        [propget] HRESULT enabled([out, retval] BOOL* value);
        // 设置按钮的使用状态
        [propput] HRESULT enabled([in] BOOL value);
}
```

```
// 描述Button数组对象
[
	object,
	uuid(A61F4187-848F-479d-842E-8D5383C43A82),
	dual
]
interface IButtonSet : IDispatchEx
{
}

// 描ImageProcessor对象，给出对图像进行各种操作的接口
[
	object,
	uuid(AD5F5448-A154-43be-82A2-7F5C0AFCE967),
	dual
]
interface IImageProcessor : IDispatch
{
	////////////////////////// 方法 ////////////////////////
	// 装载指定的图像源文件
	HRESULT load([in] BSTR filename);
	// 将当前的图像内容保存到一个指定的文件中
	HRESULT saveAs([in] BSTR filename);
	// 重新装载源文件
	HRESULT reload();
	// 将当前图像内容反色
	HRESULT invert();
	// 将当前图像内容进行灰度化
	HRESULT greyscale();
	// 判断当前图像处理器组件的状态是否良好
	HRESULT isReady([out, retval] BOOL* ready);
}

// 描述ViewerEvent对象
[
	object,
	uuid(00A434AF-FA9B-46db-946E-98D208B13516),
	dual
]
interface IViewerEvent : IDOM2Event
{
	////////////////////////// 属性 ////////////////////////
	// 触发该事件的按钮的标识
	[propget] HRESULT id([out, retval] BSTR* value);
}


//////////////////// 2级DOM的事件处理接口 ////////////////////
// 描述EventTarget对象，即事件的目标
```

```
[
	object,
	uuid(151485CC-9D11-4ea6-B7F6-3420382BBEB7),
	dual
]
interface IDOM2EventTarget : IDispatch
{
	////////////////////////// 方法 //////////////////////////
	// 增加一个事件监听者
	HRESULT addEventListener(
		[in] BSTR type,
		[in] IDispatch* pListener,
		[in] BOOL useCapture);

	// 删除一个事件监听者
	HRESULT removeEventListener(
		[in] BSTR type,
		[in] IDispatch* pListener,
		[in] BOOL useCapture);

	// 将一个事件分发给对该事件感兴趣的所有监听者
	HRESULT dispatchEvent([in] IDispatch* pEvent, [out,retval] BOOL* value);
}

// 描述EventListener对象，即事件的监听者
[
	object,
	uuid(86E3F687-7676-4c8f-BD82-BAA4800DA0F3),
	dual
]
interface IDOM2EventListener : IDispatch
{
	////////////////////////// 方法 //////////////////////////
	// 处理指定的事件
	HRESULT handleEvent([in] IDispatch* pEvent);
}

// 描述Event对象，给出事件的类型、目标、发生时间等
[
	object,
	uuid(6D124E4C-2AE9-4d13-B1E3-8F856FC1FED4),
	dual
]
interface IDOM2Event : IDispatch
{
	////////////////////////// 属性 //////////////////////////
	// 获取该事件的类型
	[propget] HRESULT type([out, retval] BSTR* value);
	// 获取该事件的目标
```

```
        [propget] HRESULT target([out, retval] IDispatch** ppDisp);
        // 获取该事件当前所在的目标
        [propget] HRESULT currentTarget([out, retval] IDispatch** ppDisp);
        // 获取该事件所处的状态
        [propget] HRESULT eventPhase([out, retval] USHORT* value);
        // 获取该事件是否支持冒泡
        [propget] HRESULT bubbles([out, retval] BOOL* value);
        // 获取该事件是否可以被放弃
        [propget] HRESULT cancelable([out, retval] BOOL* value);
        // 获取该事件发生的时间
        [propget] HRESULT timeStamp([out, retval] DATE* value);

        ////////////////////////// 方法 ////////////////////////
        // 停止该事件被其他的监听者处理
        HRESULT stopPropagation();
        // 放弃该事件的默认处理
        HRESULT preventDefault();
        // 初始化一个事件
        HRESULT initEvent(
                [in] BSTR typeArg,
                [in] BOOL canBubbleArg,
                [in] BOOL cancelableArg);

        ////////////////////////// 常量定义 ////////////////////////
        // 捕捉状态
        [propget] HRESULT CAPTURING_PHASE([out, retval] USHORT* value);
        // 在目标中
        [propget] HRESULT AT_TARGET([out, retval] USHORT* value);
        // 冒泡状态
        [propget] HRESULT BUBBLING_PHASE([out, retval] USHORT* value);
    }


    ////////////////////////////////////////////////////////////////
    [
        uuid(ED6891B0-AAAB-4d0d-9A99-0BE9CBA22777),
        helpstring("ImageViewer COM class"),
        appobject
    ]
    coclass ImageViewerComObject
    {
        interface IApplication;
    }
}
```

提示：上述 ImageViewer.idl 文件经过 MIDL 工具编译之后会生成一个类型库文件 ImageViewer.tlb，这个库文件在后面使用 Active Scripting 进行脚本解析时会被用到。

6.2.2 面向接口的实现

上一节中定义了开放给脚本程序的所有对象和接口。接下去就要针对各个接口函数进行实现了，比如对图像的处理功能都实现在一个 ImageProcessor 类中（前面几个版本的看图软件早就完成了这部分功能的实现）。然而在 Active Scripting 中，能够跟脚本程序交互的对象都要支持 COM 自动化。因此，我们需要给 ImageProcessor 对象提供一个自动化的“外壳”（也就是一个包装对象）。不仅如此，Application、MainWindow、Button 等对象也都分别需要一个自动化的“外壳”。

提示：在实现自动化对象时，我们使用了 IDispatchImpl、IDispatchExImpl2 等模板类。有了这些模板类的支持，自动化对象实现起来就非常轻松了。这些模板类的实现早在第 5 章就介绍过了，这里不再展开。

```
////////////////////////// Application对象的实现 ////////////////////////////
//
// CApplicationWrapper.h
//

#ifndef __H_CApplicationWrapper__
#define __H_CApplicationWrapper__

#include "IDispatchImpl.h"
#include "ImageViewer_h.h"

class CImageProcessorWrapper;
class CMainWindowWrapper;

class CApplicationWrapper : public IDispatchImpl<CApplicationWrapper
                                        , IApplication
                                        , &IID_IApplication>
{
public:
        CApplicationWrapper(ITypeLib* ptl);
        ~CApplicationWrapper();

        // 开放给脚本程序的属性和方法
        STDMETHODIMP get_version(BSTR* value);
        STDMETHODIMP get_mainWindow(IDispatch** ppdisp);
        STDMETHODIMP get_imageProcessor(IDispatch** ppdisp);
        STDMETHODIMP put_onApplicationLoaded(IDispatch* pCallback);
        STDMETHODIMP getImageFile(IDispatch* pCallback);
        STDMETHODIMP saveImageFile(IDispatch* pCallback);
        STDMETHODIMP refreshDisplay();
        STDMETHODIMP showAboutDialog();
        STDMETHODIMP minimizeMainWindow();
        STDMETHODIMP exitApplication();

        // 常量属性
        STDMETHODIMP get_SUCCEEDED(UINT* value);
```

```
        STDMETHODIMP get_FAILED(UINT* value);

        // 2级DOM事件处理模型的EventTarget接口函数
        STDMETHODIMP addEventListener(BSTR type,IDispatch* pListener,BOOL useCapture);
        STDMETHODIMP removeEventListener(BSTR type, IDispatch* pListener, BOOL
useCapture);
        STDMETHODIMP dispatchEvent(IDispatch* pEvent, BOOL* value);

private:
        CImageProcessorWrapper*      mImageProcessor;      // 图像处理组件
        CMainWindowWrapper*          mMainWindow;          // 主窗口对象
};

#endif // __H_CApplicationWrapper__


////////////////////////// MainWindow对象的实现 ////////////////////////////
//
// CMainWindowWrapper.h
//

#ifndef __H_CMainWindowWrapper__
#define __H_CMainWindowWrapper__

#include "IDispatchImpl.h"
#include "ImageViewer_h.h"

class CButtonSet;

class CMainWindowWrapper : public IDispatchImpl<CMainWindowWrapper
                                        , IMainWindow
                                        , &IID_IMainWindow>
{
public:
        CMainWindowWrapper(ITypeLib* ptl);
        ~CMainWindowWrapper();

        // 开放给脚本程序的属性和方法
        STDMETHODIMP get_buttons(IDispatch** ppdisp);

private:
        CButtonSet*           mButtons;      // 包含所有按钮的数组
};

#endif // __H_CMainWindowWrapper__


////////////////////////// ImageProcessor对象的实现 ////////////////////////////
//
```

```
// CImageProcessorWrapper.h
//

#ifndef __H_CImageProcessorWrapper__
#define __H_CImageProcessorWrapper__

#include "IDispatchImpl.h"
#include "ImageViewer_h.h"

class ImageProcessor;

class CImageProcessorWrapper : public IDispatchImpl<CImageProcessorWrapper
                                                    , IImageProcessor
                                                    , &IID_IImageProcessor>
{
public:
        CImageProcessorWrapper(ITypeLib* ptl, ImageProcessor* processor);
        ~CImageProcessorWrapper();

        // 开放给脚本程序的属性和方法
        STDMETHODIMP load(BSTR filename);
        STDMETHODIMP saveAs(BSTR filename);
        STDMETHODIMP reload();
        STDMETHODIMP invert();
        STDMETHODIMP greyscale();
        STDMETHODIMP isReady(BOOL* ready);

private:
        ImageProcessor*             mProcessor;    // 图像处理器组件
};

#endif // __H_CImageProcessorWrapper__


////////////////////////// Button对象的实现 ////////////////////////////
//
// CButtonWrapper.h
//

#ifndef __H_CButtonWrapper__
#define __H_CButtonWrapper__

#include "IDispatchImpl.h"
#include "ImageViewer_h.h"
#include "IDispatchExImpl2.h"

class CScriptButton;

class CButtonWrapper : public IDispatchImpl<CButtonWrapper
```

```
                                        , IButton
                                        , &IID_IButton>
{
public:
        CButtonWrapper(ITypeLib* ptl, CScriptButton* button);
        ~CButtonWrapper();

        // 开放给脚本程序的属性和方法
        STDMETHODIMP get_id(BSTR* value);
        STDMETHODIMP get_enabled(BOOL* value);
        STDMETHODIMP put_enabled(BOOL value);

private:
        CScriptButton*        mButton;        // 按钮对象
};


////////////////////////////////////////////////////////////////////////////
// 按钮数组对象
class CButtonSet : public IDispatchExImpl2<CButtonSet
                                        , IButtonSet
                                        , &IID_IButtonSet>
{
public:
        CButtonSet(ITypeLib* ptl);
        ~CButtonSet();

        // 重新实现父类IDispatchExImpl2中虚函数
        virtual HRESULT IsKeyDefined(BSTR key);
        virtual HRESULT GetValueOfKey(BSTR key, VARIANT* value);
};

#endif // __H_CButtonWrapper__
```

各个脚本对象都实现好之后，就可以设计宿主程序了。宿主程序负责创建和管理脚本引擎对象、以及各个受控对象。在使用 Active Scripting 进行脚本解析的过程中，宿主程序担当的角色是至关重要的。这里我们实现了一个 CMyJScriptHost 类，参考如下：

```
//
// CMyJScriptHost.h
//

#ifndef __H_CMyJScriptHost__
#define __H_CMyJScriptHost__

#include "activscp.h"

class CApplicationWrapper;
```

```
class CMyJScriptHost : public IActiveScriptSite
{
public:
	CMyJScriptHost();
	~CMyJScriptHost();

	// 初始化
	bool Initialize();
	// 反初始化
	void Finalize();
	// 执行指定的脚本代码
	void Execute(wchar_t* scripts);

	// --- IUnknown接口的各个成员函数 ---
	HRESULT STDMETHODCALLTYPE QueryInterface( /* [in] */ REFIID riid,
		/* [out] */ void  **ppvObject );
	ULONG STDMETHODCALLTYPE AddRef();
	ULONG STDMETHODCALLTYPE Release();

	// --- IActiveScriptSite接口的各个成员函数 ---
	STDMETHODIMP GetLCID(LCID * plcid);
	STDMETHODIMP GetItemInfo(LPCOLESTR pstrName, DWORD dwReturnMask,
		IUnknown **ppunkItem, ITypeInfo **ppTypeInfo);
	STDMETHODIMP GetDocVersionString(BSTR *pbstrVersionString);
	STDMETHODIMP OnScriptTerminate(const VARIANT *pvarResult, const EXCEPINFO
*pexcepinfo);
	STDMETHODIMP OnStateChange(SCRIPTSTATE ssScriptState);
	STDMETHODIMP OnScriptError(IActiveScriptError *pase);
	STDMETHODIMP OnEnterScript(void);
	STDMETHODIMP OnLeaveScript(void);

private:
	long                        m_cRef;        // 引用计数
	bool                        mInited;       // 初始化标记
	ITypeLib *                  mTypeLib;      // 类型库
	IActiveScript *             mActiveScript; // 脚本引擎对象
	IActiveScriptParse*   mActiveScriptParse;  // 用于脚本解析的接口

	// 受控对象
	CApplicationWrapper*  mApplication;
};

#endif // __H_CMyJScriptHost__
```

6.2.3 事件和事件处理

Windows 程序是通过消息来描述各种事件的，如果程序对某个消息感兴趣，可以通过消息映射来为它注册一个处理函数。但我们现在不能使用这种消息映射了，而只能依赖另

外的一种事件注册和处理的机制，使 UI 上面的事件能够发送到脚本程序中去被处理。比如脚本程序对 Open 按钮的单击事件感兴趣，要求用户在点击这个按钮之后，执行一个特定的脚本函数。我们的程序必须对上述的整个过程提供支持。

所幸的是，2 级 DOM 定义了一个事件处理模型，正好符合我们的要求（详细的接口说明书可到 www.w3.org 网站上获取）。这个事件模型中定义的 3 个对象：EventTarget、EventListener、Event，下面我们给出了它们的 C++实现：

```
//
// DOM2_defs.h
//

#ifndef __H_DOM2_defs__
#define __H_DOM2_defs__

#include <string>

// 定义数据类型
typedef std::string DOMString;
typedef long DOMTimeStamp;

#endif // __H_DOM2_defs__


//////////////////////////// EventTarget对象的实现 ////////////////////////////
//
// DOM2EventTarget.h
//

#ifndef __H_DOM2EventTarget__
#define __H_DOM2EventTarget__

#include "DOM2_defs.h"
#include <vector>

class DOM2EventListener;
class DOM2Event;

class DOM2EventTarget
{
public:
        DOM2EventTarget();
        virtual ~DOM2EventTarget();

        // 增加一个事件监听者
        virtual void addEventListener(DOMString type, DOM2EventListener & listener, bool useCapture);
        // 删除一个事件监听者
        virtual void removeEventListener(DOMString type, DOM2EventListener & listener, bool useCapture);
        // 将一个事件分发给对该事件感兴趣的所有监听者
        virtual bool dispatchEvent(DOM2Event const& pEvent);
```

```
protected:
	struct EventListenerStruct
	{
		DOMString			type;
		DOM2EventListener*	listener;
		bool				useCapture;

		EventListenerStruct() : type(), listener(0), useCapture(false) {}
		~EventListenerStruct() {}
	};

	// 判断两个事件监听者是否相同
	virtual bool isSameListener(EventListenerStruct& pListener1, EventListenerStruct& pListener2);

	// 保存所有监听者的数组
	typedef std::vector<EventListenerStruct> EventListenerVector;
	EventListenerVector		mListeners;
};

#endif // __H_DOM2EventTarget__


////////////////////////// EventListener对象的实现 ////////////////////////////
//
// DOM2EventListener.h
//

#ifndef __H_DOM2EventListener__
#define __H_DOM2EventListener__

class DOM2Event;

class DOM2EventListener
{
public:
	DOM2EventListener();
	virtual ~DOM2EventListener();

	// 处理指定的事件
	virtual void handleEvent(DOM2Event const& pEvent);
};

#endif // __H_DOM2EventListener__


////////////////////////// Event对象的实现 ////////////////////////////
//
// DOM2Event.h
//
```

```
#ifndef __H_DOM2Event__
#define __H_DOM2Event__

#include "DOM2_defs.h"

class DOM2EventTarget;

class DOM2Event
{
public:
	DOM2Event();
	DOM2Event(DOMString const& eventType);
	virtual ~DOM2Event();

	// 定义事件的状态
	enum EventPhase
	{
		UNDEFINED = 0,
		CAPTURING_PHASE = 1,
		AT_TARGET       = 2,
		BUBBLING_PHASE  = 3
	};

	////////////////////// 事件的标准属性 ////////////////////////
	// 获取该事件的类型
	virtual DOMString getType() const {return mEventType;}
	// 获取该事件的目标
	virtual DOM2EventTarget* getTarget() const {return mTarget;}
	// 获取该事件当前所在的目标
	virtual DOM2EventTarget* getCurrentTarget() const {return mCurrentTarget;}
	// 获取该事件所处的状态
	virtual EventPhase getEventPhase() const {return mEventPhase;}
	// 获取该事件是否支持冒泡
	virtual bool getBubbles() const {return mCanBubble;}
	// 获取该事件是否可以被放弃
	virtual bool getCancelable() const {return mCancelable;}
	// 获取该事件发生的时间
	virtual DOMTimeStamp getTimeStamp() const {return mTimeStamp;}

	////////////////////// 事件的标准方法 ////////////////////////
	// 停止该事件被其他的监听者处理
	virtual void stopPropagation();
	// 放弃该事件的默认处理
	virtual void preventDefault();
	// 初始化一个事件
	virtual void initEvent(const DOMString& eventTypeArg, bool canBubbleArg, bool cancelableArg);

	////////////////////// 自定义方法 ////////////////////////
	void setTarget(DOM2EventTarget* p){mTarget=p;}
```

```
        void setCurrentTarget(DOM2EventTarget* p){mCurrentTarget=p;}
        void setEventPhase(EventPhase eventPhase){mEventPhase=eventPhase;}
        void setTimeStamp(DOMTimeStamp timeStamp){mTimeStamp=timeStamp;}
        bool isStopPropagation() const {return mStopPropagation;}
        bool isPreventDefault() const {return mPreventDefault;}

protected:
        DOM2EventTarget*      mTarget;              // 该事件的目标
        DOM2EventTarget*      mCurrentTarget;       // 该事件当前所在的目标
        EventPhase            mEventPhase;          // 该事件所处的状态
        DOMTimeStamp          mTimeStamp;           // 该事件发生的时间
        DOMString             mEventType;           // 该事件的类型
        bool                  mCanBubble;           // 该事件是否支持冒泡
        bool                  mCancelable;          // 该事件是否可以被放弃
        bool                  mStopPropagation;     // 该事件是否被阻止让其他监听者处理
        bool                  mPreventDefault;      // 该事件是否被阻止进行默认的处理
};

#endif // __H_DOM2Event__
```

当我们的看图软件主界面上的按钮被点击时，应用程序会发出一个名为“ev_ButtonClicked”的事件。但我们怎么来区分是 Open 按钮被按下了，还是 Save 按钮被按下了呢？这就要求事件对象携带一个自定义的属性，来表示触发该事件的按钮的标识。另外，事件是要发送到脚本程序中去被处理，因此事件对象也需要有一个自动化的“外壳”。

```
////////////////////////// ViewerEvent对象的实现 ////////////////////////////
//
// ViewerEvent.h
//

#ifndef __H_ViewerEvent__
#define __H_ViewerEvent__

#include "DOM2Event.h"

class ViewerEvent : public DOM2Event
{
public:
        ViewerEvent(DOMString const& eventType, DOMString const& strID);
        virtual ~ViewerEvent();

        // 获取触发该事件的按钮的标识
        DOMString GetID() { return mID; }

protected:
        DOMString             mID; // 按钮的标识
};
```

```
#endif // __H_ViewerEvent__
```

6.2.4 脚本驱动起来！

既然软件主界面上具有的 UI 元素是由一个 XML 文件来决定的，UI 上的各个事件现在也是由脚本程序来响应的，可以说我们的看图软件是彻底的脚本驱动的了。这里的 XML 文件内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<ImageViewer version="1.1">
        <MainWindow skin="main.jpg">
                <ImageBox xpos="20" ypos="85" width="582" height="446"/>
                <Button id="btn_Min" function="Minimize" xpos="565" ypos="6" skin="btn_Min.jpg"/>
                <Button id="btn_Exit" function="Exit" xpos="585" ypos="6" skin="btn_Exit.jpg"/>
                <Button id="btn_Open" function="Open" xpos="21" ypos="40" skin="btn_Open.jpg"/>
                <Button id="btn_Reload" xpos="83" ypos="40" skin="btn_Reload.jpg"/>
                <Button id="btn_Invert" function="Invert" xpos="155" ypos="40" skin="btn_Invert.jpg"/>
                <Button id="btn_Greyscale" function="Greyscale" xpos="221" ypos="40" skin="btn_Greyscale.jpg"/>
                <Button id="btn_Save" function="Save" xpos="293" ypos="40" skin="btn_Save.jpg"/>
                <Button id="btn_About" function="About" xpos="357" ypos="40" skin="btn_About.jpg"/>
                <Static xpos="428" ypos="33" skin="logo_ScriptDriven.jpg"/>
                <Hyperlink name="http://www.chery.cn" xpos="400" ypos="6" width="160" height="20"/>
                <Script source="main.js"/>
        </MainWindow>
</ImageViewer>
```

脚本程序文件中的实现如下：

```
//
// main.js
//

// 设置一个应用程序运行起来时的回调通知
Application.onApplicationLoaded = applicationLoaded;

// 事件注册：脚本程序需要处理“ev_ButtonClicked”这个事件
addEventListener("ev_ButtonClicked", buttonClicked, false);

// 应用程序运行起来后会调用这个函数
function applicationLoaded()
{
        // 演示UI逻辑在脚本程序中的实现：控制各个按钮的使用状态
        mainWindow.buttons["btn_Reload"].enabled = false;
        mainWindow.buttons["btn_Invert"].enabled = false;
        mainWindow.buttons["btn_Greyscale"].enabled = false;
        mainWindow.buttons["btn_Save"].enabled = false;
}

// 处理按钮的单击事件
function buttonClicked(ev)
```

```
{
        // 判断触发这个事件的按钮的标识
        switch (ev.id)
        {
                case "btn_Open":
                        // 让用户选择一个图像源文件，
                        // 然后执行脚本回调函数getImageFileCompleted
                        getImageFile(getImageFileCompleted);
                        break;

                case "btn_Reload":
                        // 让图像处理器组件重新装载当前的源文件
                        imageProcessor.Reload();
                        // 刷新图像显示
                        refreshDisplay();
                        break;

                case "btn_Invert":
                        // 让图像处理器组件将当前的图像内容反色
                        imageProcessor.invert();
                        // 刷新图像显示
                        refreshDisplay();
                        break;

                case "btn_Greyscale":
                        // 让图像处理器组件将当前的图像进行灰度化处理
                        imageProcessor.greyscale();
                        // 刷新图像显示
                        refreshDisplay();
                        break;

                case "btn_Save":
                        // 让用户选择一个目标文件，
                        // 然后执行脚本回调函数saveImageFileCompleted
                        saveImageFile(saveImageFileCompleted);
                        break;

                case "btn_About":
                        // 显示“About”对话框
                        showAboutDialog();
                        break;

                case "btn_Min":
                        // 将主窗口最小化
                        minimizeMainWindow();
                        break;

                case "btn_Exit":
                        // 退出应用程序
```

```
                    exitApplication();
                    break;
        }
}

// 如果用户选择了一个图像源文件，则使用图像处理器组件对进行解码、显示
function getImageFileCompleted(status, filename)
{
        if (status == SUCCEEDED)
        {
                // 装载源文件
                imageProcessor.load(filename);
                // 更新界面上各个按钮的状态
                updateButtonStatus();
                // 刷新图像显示
                refreshDisplay();
        }
}

// 如果用户选择了一个目标文件，则使用图像处理器组件将当前的图像内容保存到该文件中
function saveImageFileCompleted(status, filename)
{
        if (status == SUCCEEDED)
        {
                imageProcessor.saveAs(filename);
        }
}

// 更新界面上各个按钮的状态
function updateButtonStatus()
{
        if (imageProcessor.isReady())
        {
                mainWindow.buttons["btn_Reload"].enabled = true;
                mainWindow.buttons["btn_Invert"].enabled = true;
                mainWindow.buttons["btn_Greyscale"].enabled = true;
                mainWindow.buttons["btn_Save"].enabled = true;
        }

}
```

要让上面的脚本真正地驱动我们的看图软件，我们还要做如下实现：

```
//
// ImageViewerDlg.h
//

#pragma once
#include "afxwin.h"
```

```
#include "CSkinDialog.h"
#include "DOM2EventTarget.h"
#include "CScriptButton.h"
#include "CEasyHyperlink.h"
#include <list>

class ImageProcessor;
class CMyJScriptHost;
class DispatchHelper;

typedef std::list<CScriptButton*> ButtonList;
typedef std::list<CWnd*> StaticList;

class CImageViewerDlg : public CSkinDialog, public DOM2EventTarget
{
	typedef CSkinDialog INHERITED;

public:
	CImageViewerDlg(CWnd* pParent = NULL);	// standard constructor
	~CImageViewerDlg();

	// 为主窗口装载皮肤图片
	virtual void LoadDialogImages();

protected:
	HICON m_hIcon;

	CString			mImageFile;		// 图像源文件
	ImageProcessor*		mProcessor;		// 图像处理器组件
	CMyJScriptHost*		mScriptHost;		// 脚本宿主程序
	DispatchHelper*		mCallbackHelper;	// 程序运行起来后的回调函通知

	// 为了全局范围内能够访问到这个主应用程序对象而定义的静态成员
	static CImageViewerDlg*	sMainApplication;

	// 显示图像
	void DisplayImage(void);
	// 分析XML文件的内容：创建各个UI元素、执行脚本程序
	void ParseUISkinXml(void);
	// 判断两个监听者是否相同
	virtual bool isSameListener(EventListenerStruct& pListener1, EventListenerStruct& pListener2);

private:
	CStatic			mPictureWnd;	// 图像显示窗口
	CEasyHyperlink*		mpHyperlink;	// 超级链接控件
	ButtonList		mButtons;	// 按钮队列
	StaticList		mStatics;	// 静态控件队列

public:
	// 主应用程序对应于脚本接口函数的实现
```

```
        bool GetImageFile(CString& filename);
        bool SaveImageFile(CString& filename);
        void MinimizeMainWindow();
        void ExitApplication();
        void SetApplicationLoadedHandler(DispatchHelper* pHandler);
        void ShowAboutDialog();
        void RefreshDisplay();

        // 获取图像处理器组件对象的指针
        ImageProcessor* GetImageProcessor() { return mProcessor;}
        // 获取主窗口中的所有按钮
        ButtonList& GetButtonList() { return mButtons; }

        // 获取主应用程序对象的指针
        static CImageViewerDlg*      GetAppInstance();
};

extern CImageViewerDlg* GetMainApp();


//
// ImageViewerDlg.cpp
//

CImageViewerDlg* CImageViewerDlg::sMainApplication = 0;
CImageViewerDlg* CImageViewerDlg::GetAppInstance() { return(sMainApplication); }
CImageViewerDlg* GetMainApp() { return CImageViewerDlg::GetAppInstance(); }

////////////////////////////////////////////////////////////////////////////
// 构造函数
CImageViewerDlg::CImageViewerDlg(CWnd* pParent /*=NULL*/)
        : INHERITED(IDD_IMAGEVIEWER_DIALOG, pParent)
        , mImageFile("")
        , mProcessor(new ImageProcessor())
        , mScriptHost(new CMyJScriptHost())
        , mCallbackHelper(0)
        , mpHyperlink(0)
{
        m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);

        // 使主应用程序在全局范围内可见!
        sMainApplication = this;

        // 创建XML分析器对象
        CUISkinXmlParser::CreateInstance();
}

// 析构函数
CImageViewerDlg::~CImageViewerDlg()
```

```
{
	// 销毁图像处理器组件对象、脚本宿主程序
	SAFE_DELETE(mProcessor);
	SAFE_DELETE(mCallbackHelper);
	SAFE_DELETE(mScriptHost);

	// 销毁各个按钮对象
	ButtonList::iterator itBegin(mButtons.begin()), itEnd(mButtons.end());
	for (ButtonList::iterator it = itBegin; it != itEnd; ++it)
	{
		if (*it)
		{
			delete (*it);
			*it = 0;
		}
	}

	// 销毁各个静态控件对象
	StaticList::iterator itBegin2(mStatics.begin()), itEnd2(mStatics.end());
	for (StaticList::iterator it = itBegin2; it != itEnd2; ++it)
	{
		if (*it)
		{
			delete (*it);
			*it = 0;
		}
	}

	// 销毁超级链接控件对象
	SAFE_DELETE(mpHyperlink);

	// 销毁XML分析器对象
	CUISkinXmlParser::DeleteInstance();

	sMainApplication = 0;
}

// 对话框初始化函数
BOOL CImageViewerDlg::OnInitDialog()
{
	INHERITED::OnInitDialog();
	SetWindowText("ImageViewer");

	// …

	// 分析XML文件的内容：创建各个UI元素、执行脚本程序
	ParseUISkinXml();

	// 应用程序运行起来后向脚本程序发出通知
```

```
	if (mCallbackHelper)
		mCallbackHelper->Invoke(0);

	return TRUE;
}

// 分析XML文件的内容：创建各个UI元素、执行脚本程序
void CImageViewerDlg::ParseUISkinXml(void)
{
	CUISkinXmlParser* pParser = CUISkinXmlParser::GetInstance();

	// 定位到<MainWindow>下的第1个字节点
	bool found = pParser->FindFirstNode();
	while (found)
	{
		// 判断节点类型
		// 根据不同的节点类型，创建不同的UI元素或执行脚本程序
		switch (pParser->GetNodeTypeNumber())
		{
		case NT_ImageBox: // 图像显示窗口
		{
			long x, y, width, height;
			pParser->GetPosition(x, y, width, height);
			CRect rect(x, y, x+width, y+height);
			mPictureWnd.Create("", WS_CHILD|WS_BORDER|WS_VISIBLE|SS_BITMAP, rect, this, 1234);
		}
		break;

		case NT_Button: // 按钮
		{
			CScriptButton* pButton = new CScriptButton(this, pParser->GetID());
			pButton->SetSkinFile(pParser->GetSkin());
			pButton->SetSkinPosition(pParser->GetXPos(), pParser->GetYPos());
			pButton->CreateSelf(this);
			mButtons.push_back(pButton);
		}
		break;

		case NT_Static: // 静态控件
		{
			CString strSkin = pParser->GetSkin();
			if (strSkin != "")
			{
				// 包含一幅图像的静态控件
				CScriptButton* pButton = new CScriptButton(NULL, "", true);
				pButton->SetSkinFile(strSkin);
				pButton->SetSkinPosition(pParser->GetXPos(), pParser->GetYPos());
				pButton->CreateSelf(this);
				mStatics.push_back(pButton);
			}
			else
			{
				// 包含文字的静态控件
				// ...
```

```
                }
            }
            break;

            case NT_Hyperlink: // 超级链接
            {
                SAFE_DELETE(mpHyperlink);
                mpHyperlink = new CEasyHyperlink();

                RECT rect;
                rect.left   = pParser->GetXPos();
                rect.top    = pParser->GetYPos();
                rect.right  = rect.left + pParser->GetWidth();
                rect.bottom = rect.top + pParser->GetHeight();
                mpHyperlink->create(rect, pParser->GetName(), this->GetSafeHwnd());
            }
            break;

            case NT_Script: // 脚本
            {
                // 初始化脚本宿主程序
                if (mScriptHost && mScriptHost->Initialize())
                {
                    // 装载脚本程序文件
                    wchar_t * wszScripts = 0;
                    CString jsSrcFile=MiscUtils::GetSkinFolder()+"\\"+pParser->GetSource();
                    if (MiscUtils::LoadScriptFile(jsSrcFile, &wszScripts))
                    {
                        // 执行脚本程序
                        mScriptHost->Execute(wszScripts);
                        delete[] wszScripts;
                    }
                }
            }
            break;

            default:
                break;
            }

            // 指向下一个节点
            found = pParser->FindNextNode();
        }
    }

    // 为主窗口装载皮肤图片
    void CImageViewerDlg::LoadDialogImages()
    {
        CUISkinXmlParser* pParser = CUISkinXmlParser::GetInstance();
        mSkinFileName = pParser->GetMainWindowSkin();

        CRgn windowRegion;
        LoadImageMap(windowRegion);
```

```
        if ((HRGN)windowRegion)
        {
                SetWindowRgn((HRGN)windowRegion, TRUE);
        }
}

// 显示图像
void CImageViewerDlg::DisplayImage(void)
{
        if (mProcessor && mPictureWnd.GetSafeHwnd())
        {
                mProcessor->Display(&mPictureWnd);
        }
}

// 判断两个事件监听者是否相同
bool CImageViewerDlg::isSameListener(EventListenerStruct& pListener1, EventListenerStruct& pListener2)
{
        if (DOM2EventTarget::isSameListener(pListener1, pListener2))
        {
                return true;
        }

        JSEventListener* pJSListener1 = dynamic_cast<JSEventListener*>(pListener1.listener);
        JSEventListener* pJSListener2 = dynamic_cast<JSEventListener*>(pListener2.listener);
        if (pJSListener1 && pJSListener1->isSame(pJSListener2))
        {
                return true;
        }
        return false;
}

// 主应用程序对应于脚本接口函数的实现
// 获取一个图像源文件
bool CImageViewerDlg::GetImageFile(CString& filename)
{
        return MiscUtils::BrowseImageFile(filename);
}

// 获取一个目标文件（用于将当前图像内容保存到该文件中）
bool CImageViewerDlg::SaveImageFile(CString& filename)
{
        return MiscUtils::SaveAsFile(filename);
}

// 最小化主窗口
void CImageViewerDlg::MinimizeMainWindow()
{
        ShowWindow(SW_MINIMIZE);
}
```

```
// 退出应用程序
void CImageViewerDlg::ExitApplication()
{
	INHERITED::OnOK();
}

// 显示“About”对话框
void CImageViewerDlg::ShowAboutDialog()
{
	CAboutDlg dlg;
	dlg.DoModal();
}

// 刷新图像显示
void CImageViewerDlg::RefreshDisplay()
{
	DisplayImage();
}

// 设置一个应用程序运行起来时的回调通知
void CImageViewerDlg::SetApplicationLoadedHandler(DispatchHelper* pHandler)
{
	SAFE_DELETE(mCallbackHelper);
	mCallbackHelper = pHandler;
}
```

提示：上述程序中创建按钮对象时使用了 CScriptButton 类，这个类的实现跟本书第 4 章介绍的 CCustomizedButton 类非常相似，只不过 TakeClickAction 函数需要重新实现为发出一个 2 级 DOM 事件，如下：

```
void CScriptButton::TakeClickAction()
{
	if (mTarget)
	{
		// 创建一个“ev_ButtonClicked”事件
		ViewerEvent clickEvent("ev_ButtonClicked", (LPCSTR)mID);
		// 分发这个事件
		mTarget->dispatchEvent(clickEvent);

#ifdef _DEBUG
	//	::AfxMessageBox("Button <" + mID + "> clicked.");
#endif
	}
}
```

6.3 实例程序：ImageViewer_ScriptDriven

实例程序 ImageViewer_ScriptDriven 位于本书配套代码包的 ImageViewer_ScriptDriven 目录下。**这个程序是在 ImageViewer_Customized 的基础上修改过来的，请读者注意比较两者之间的区别**。软件运行起来界面如图 6.1：

图 6.1 ImageViewer_ScriptDriven 的软件界面