

VBScript 程序员参考手册（第三版）

作者：（美）Adrian Kingsley-Hughes, Kathie Kingsley-Hughes, D

目录

前言	6
本书面向的读者	6
本书涵盖的内容	7
本书组织方式	7
使用本书需要的准备工作	8
源代码	8
勘误	9
p2p.wrox.com	9
第 4 章 变量与过程	11
4.1 Option Explicit	11
4.2 变量命名	13
4.3 过程和函数	14
4.3.1 过程的语法	15
4.3.2 函数的语法	18
4.3.3 调用过程和函数	21
4.3.4 可选参数	23
4.3.5 退出过程或函数	23
4.4 变量作用域、声明及生存期	25
4.4.1 理解变量作用域	25
4.4.2 理解变量声明	27
4.4.3 变量生存期	27
4.5 脚本和过程的设计策略	29
4.5.1 限制代码读取和修改变量	29
4.5.2 将代码分割成过程和函数	30
4.5.3 关于脚本设计的忠告	32
4.6 ByRef 和 RyVal	32
4.7 文字常量和具名常量	35
4.7.1 什么是文字常量	35
4.7.2 什么是具名常量	36
4.7.3 使用具名常量的好处	37
4.7.4 具名常量的使用原则	38
4.7.5 VBScript 的内建常量	39
4.8 小结	40
第 5 章 流程控制	41
5.1 分支结构	41
5.1.1 If 分支	42
5.1.2 Select Case 分支	45

5.2 循环结构	48
5.2.1 For...Next	48
5.2.2 For Each...Next	53
5.2.3 Do Loop	56
5.2.4 While...Wend	66
5.3 小结	67
第 8 章 VBScript 中的类(编写 COM 对象)	68
8.1 对象、类和组件	68
8.2 类语句	69
8.3 定义属性	70
8.3.1 私有属性变量	70
8.3.2 Property Let	71
8.3.3 Property Get	72
8.3.4 Property Set	73
8.3.5 创建只读属性	75
8.3.6 创建只写属性	77
8.3.7 没有属性过程的公共属性	77
8.4 定义方法	78
8.5 类事件	80
8.5.1 Class_Initialize 事件	81
8.5.2 Class_Terminate 事件	82
8.6 类常量	83
8.7 构建和使用 VBScript 类	84
8.8 小结	95
第 9 章 正则表达式	96
9.1 正则表达式简介	96
9.1.1 实战正则表达式	96
9.1.2 从简单的开始	99
9.2 RegExp 对象	101
9.2.1 Global 属性	101
9.2.2 IgnoreCase 属性	102
9.2.3 Pattern 属性	103
9.2.4 正则表达式字符	103
9.2.5 Execute 方法	111
9.2.6 Replace 方法	112
9.2.7 Backreferencing	113
9.2.8 Test 方法	114
9.3 Matches 集合	115
9.3.1 Matches 的属性	116
9.3.2 Match 对象	117
9.4 一些例子	120
9.4.1 验证电话号码输入	120
9.4.2 分解 URI	122
9.4.3 检查 HTML 元素	122

9.4.4	匹配空白	123
9.4.5	匹配 HTML 注释标签	124
9.5	小结	125
第 15 章	Windows 脚本宿主	126
15.1	相关工具	127
15.2	WSH 的概念	127
15.3	脚本文件的类型	129
15.4	使用 Windows 脚本宿主运行脚本	130
15.4.1	命令行执行	130
15.4.2	在 Windows 环境中执行 WSH	132
15.5	使用 .WSH 文件运行脚本	133
15.6	Windows 脚本宿主的内建对象	134
15.6.1	WScript 对象	134
15.6.2	WshArguments 对象	144
15.6.3	WshShell 对象	146
15.6.4	WshNamed 对象	166
15.6.5	WshUnnamed 对象	168
15.6.6	WshNetwork 对象	169
15.6.7	WshEnvironment 对象	176
15.6.8	WshSpecialFolders 对象	179
15.6.9	WshShortcut 对象	182
15.6.10	WshUrlShortcut 对象	190
15.7	小结	194
第 16 章	Windows 脚本组件	195
16.1	什么是 Windows 脚本组件	195
16.2	需要的工具	196
16.3	脚本组件运行时	196
16.4	脚本组件文件和向导	197
16.5	暴露属性、方法和事件	205
16.5.1	属性	205
16.5.2	方法	208
16.5.3	事件	210
16.6	注册信息	212
16.7	创建脚本组件类型库	214
16.8	如何引用其他组件	216
16.9	ASP 的脚本组件	217
16.10	编译时错误检查	219
16.11	在 Script 组件中使用 VBScript 类	220
16.11.1	VBScript 类的使用限制	220
16.11.2	使用内部类	221
16.11.3	包含外部源文件	224
16.12	小结	225
第 21 章	在 VB 和 .NET 应用程序中添加 VBScript 代码	226
21.1	为什么要在应用程序中添加脚本	227

21.2 宏和脚本的概念	227
21.2.1 使用 Scriptlet	228
21.2.2 使用脚本	229
21.2.3 选择最佳的使用范围	229
21.3 在 VB 和 .NET 应用程序中添加脚本控件	230
21.4 脚本控件参考	231
21.4.1 对象模型	231
21.4.2 对象与集合	231
21.4.3 常量	246
21.5 脚本控件错误处理	247
21.6 调试	251
21.7 使用已编码脚本	252
21.8 .NET 工程范例	252
21.9 Visual Basic 6 工程范例	253
21.10 小结	260

前言

我们希望，能够在—个文本编辑器中快速而简便地编写代码，而不用考虑任何复杂的开发环境。

我们希望，不用牵绊于编译代码或发布复杂的安装程序的繁杂工作。

我们希望，能够以多种方式部署代码。

我们还希望，只要学习—种语言就可以完成服务器端 Internet、客户端 Internet 和桌面程序的编码。

实际上这不仅仅是希望，使用 VBScript 能够实现的不仅是这些。

VBScript 是—种绝对高级的语言，甚至可以用它来“说话”。它易于学习，功能强大，灵活性强，而且很便宜。这使得 VBScript 无论对于经验丰富还是初出茅庐的程序员来说，都成为其首选语言。如果您是经验丰富的程序员，就会享受到由于不需要复杂的开发环境和编译的需要而带来的编码乐趣。如果您是初学者，那么只要懂—点文本编辑器的知识，就可以开始学习编程。

VBScript 的知识和经验还将开启很多技术的大门。如果有很好的 VBScript 编程基础，就可以涉足 Internet 开发、网络管理、服务器端编码甚至其他编程语言的使用 (Visual Basic 是—条最流行的路线，因为它的语法与 VBScript 非常相似)。使用 VBScript，还可以创建看上去像使用 C++ 这样复杂的编程语言编写的程序。还有一点值得注意的是，对脚本的支持已经嵌入到所有新版本的 Windows 操作系统中——这对于帮助了解 VBScript 的工作机制来说，无疑是一种便利条件。通过在文本编辑器中编写—些简单的脚本，就可以完成很多工作，如复制和移动文件、创建文件夹和文件、修改 Windows 注册表，还有很多很多。—种易于使用的脚本语言居然无所不能。

我们相信，了解如何编写 VBScript 应用程序对很多人来说都将是一种有用且有益的技能，无论他们是 IT 行业用户、SOHO PC 用户、学生还是家庭用户。了解和使用 VBScript 可以节省时间，更重要的是，节省金钱。

本书面向的读者

对于所有对 VBScript 学习有兴趣的人来说，这都是一本一站式的书籍。对本书的使用方式取决于读者已有的程序设计和脚本编写的知识与经验：

- 如果您是一个纯粹的初学者，目前只是听说过 **VBScript**，那么好极了，这本书是最适合您的。对于一个初学者，这将是一个引人入胜的学习过程。我们假设您能够从头到尾通读这本书，这样才能充分地利用它进行学习。
- 如果您已经具有 **IT** 和程序设计经验，只是想要学习 **VBScript**(可能是要用于 **ASP (Active Server Pages, 活动服务器页面)**或 **WSH(Windows Scripting Host, Windows 脚本宿主)**), 那么这本书也适合您。您对程序设计的了解意味着您对书中涉及的大多数术语和技术都比较熟悉。因此对您来说，学习另一种语言是比较简单的。如果了解使用 **VBScript** 的目的(例如 **ASP** 和 **WSH**)，那么就请带着这个目的阅读本书，跳过某些章节以节省时间。
- 网络管理员可能会发现本书不仅十分有用，而且还能大幅度节省他们的时间，因为他们经常会使用 **VBScript** 编写功能强大的登录脚本或自动化那些在 **WSH** 下遇到的烦琐的、重复性的、耗时巨大的、易于出错的任务。
- 如果您已经使用 **VBScript**，而只是想填补一些知识空白，或者只是为了跟踪最新技术而购买本书，您也无疑会从本书特定章节中发现新的信息，而对这些章节(比如在改版的附录中关于 **Windows Vista** 的相关内容)予以特别的关注。

本书涵盖的内容

正如您所期望的，关于 VBScript 的书只讨论 VBScript。准确地说，本书只讨论 VBScript 的最新版本(版本 5.7)。但是，VBScript 作为一种工具，可以有多种使用方法，可以用于多种不同的应用程序。因此，在详细讨论 VBScript 的同时，本书还涉及一些与 VBScript 相关的技术。这包括 Windows 脚本宿主(WSH)。同样地，如果您具有 Visual Basic 的知识，那么前三章中绝大部分内容(变量、数据类型、过程、控制流等)对您来说都很熟悉。本书将为您介绍如何深入到 Windows 操作系统内部，通过几行代码实现操作系统环境的改变。

本书组织方式

只要快速浏览一下这一版的目录，就会发现本书分为三个部分：

- 首先是关于 VBScript 核心内容的一些章节—— 基本上是关于 VBScript 作为一种语言的工作机制。
- 接下来着眼于如何在其他技术(如 WSH 或 ASP)内部使用 VBScript。这些章节中分析了一些更高级的 VBScript 脚本代码示例的运行。
- 最后以一系列附录的形式提供了详细而全面的参考信息。这些参考信息可以作为独立的部分来使用,也可以通过它们深入了解前面的章节中介绍的 VBScript 的工作机制。

以怎样的计划阅读本书实际取决于您当前 VBScript 和其他编程语言的技术水平以及您的目的。这是您的书—— 请使用最适于您的方式来阅读!

如果不能确定阅读本书的最佳方式,那么我们建议从头至尾地通读,这样就能最大程度地受益。不必担心记不住所有读到的内容—— 那并不是重点。本书是一本参考手册,这意味着您可以反复参考。在阅读的时候请做一些笔记,这将帮助您更好地记住相应内容,并便于发现曾经阅读过的重点部分。

使用本书需要的准备工作

VBScript 对于大部分脚本编写和/程序设计需求来说,可能是一种低代价的解决方案。好在您(和您的终端用户)都是使用 Microsoft Windows 操作系统,这样就已经准备好了使用本书的所有工作(您也可以从网上下载)。

所有的代码编写工作都可以使用系统中已安装的 Windows Notepad 应用程序来完成。我们将建议一些其他可用的工具,它们可能会使代码编写工作更加轻松,但实际上使用一个文本编辑器就足够了。

Microsoft Scripting Web 站点包含了与 VBScript 相关的文档,可供下载。您可能也需要下载这些文档以辅助本书的阅读。

如果不是使用 Windows Vista 或 XP,就可能需要下载最新的 VBScript 引擎—— 请访问 www.microsoft.com/scripting。

源代码

学习本书的示例时,可以选择手工输入所有代码,也可以使用本书附带的源码文件。本书中使用的所有源代码都可以从 www.wrox.com 下载得到。进入该站点后,可以使用 Search 框找到本书的题目,然后可以选择想下载的内容。

因为很多书籍都有相似的题目，所以按照 ISBN 来搜索可能更容易一些；本书的 ISBN 是 978-0-470-16808-0。

勘误

我们将尽全力保证本书的正文和代码中没有错误。但人无完人，错误总是会有。如果您发现了本书中的错误，例如拼写错误或代码的错误，那么我们将非常感激您的反馈。通过发送勘误信息，您将节省其他读者的阅读时间；这样，您的反馈将有助于我们提供更高质量的信息。

要访问本书的勘误页面，请访问 www.wrox.com，并使用 Search 框或书名列表找到本书的书名。然后，在书籍的详细信息页面上，点击 Book Errata 链接。在这个页面上，可以查看关于本书所有已经提交并由 Wrox 编辑公布的勘误信息。包括到每本书的勘误页面的链接的完整书籍列表位于 www.wrox.com/misc-pages/booklist.shtml。

如果您有任何意见或建议，或者在本书中发现任何问题，都可以发信到 wkservice@vip.163.com，我们将查看该信息，如果确切的话，将在本书的勘误页面上发布一则消息，并在本书的后续版本中修正这一问题。

p2p.wrox.com

如果要同作者进行直接的讨论，请加入 p2p.wrox.com 的 P2P 论坛。该论坛是一个基于 Web 的系统，您可以发布关于 Wrox 书籍的消息以及相关的技术信息，并与其他读者和技术用户进行交互。这个论坛提供订阅功能，可以以 E-mail 的方式将您所选择的感兴趣主题的论坛新帖发送给您。Wrox 作者、编辑、其他业界专家以及和您一样的读者都活跃在这个论坛上。

在 p2p.wrox.com，您可以找到很多不同的论坛，不仅可以帮您阅读本书，还可以帮助您开发应用程序。要加入论坛，可以按照下面的步骤操作：

1. 进入 p2p.wrox.com，单击 Register 链接。
2. 阅读用户协议，单击 Agree 按钮。
3. 填写加入论坛必需的信息以及您想提供的可选信息，单击 Submit 按钮。
4. 您将收到一封电子邮件，其中的信息将描述如何验证您的账户并完成加入过程。

不用加入 **P2P** 也可以阅读帖子，但要发帖，就必须加入。

加入论坛之后，就可以发新帖并回复其他用户的帖子。任何时候您都可以在 Web 上阅读帖子。如果想将某一论坛中的新帖以电子邮件的方式发送给您，可以单击论坛列表中论坛名称旁边的 [Subscribe to this Forum](#) 图标。

关于如何使用 Wrox P2P 的更多信息，请阅读 [P2P FAQ](#)，了解论坛软件的工作机制，以及关于 P2P 和 Wrox 书籍的很多一般问题。要阅读 FAQ，请单击 P2P 上任何一个页面中的 [FAQ](#) 链接。

第 4 章 变量与过程

本章将继续讨论 VBScript 的变量，并扩展到 VBScript 中的过程和函数。有些关于变量的重要问题还没有涉及到，包括变量命名和声明的原则，Option Explicit 语句的重要性，以及变量的作用域(scope)和生存期(lifetime)的概念。您还将学到定义过程和函数的语法，包括参数和返回值，并且还会介绍一些脚本的“设计策略”。

如果您是其他某种语言的熟练程序员，并试图跳过本章，那么最好还是大致地浏览一下。即使是一些基本的编程技能，您也可能会发掘出其中针对 VBScript 的独特之处。

4.1 Option Explicit

从前面的例子中，您可能还没有猜到 VBScript 中的变量声明实际上不是必须的。这很好，您可以在任何地方使用新变量，而无需事先声明。并不会强制要求您必须事先声明变量。一旦 VBScript 在脚本中遇到了新的未声明的变量，它就会为其分配内存。下面是一个例子(这段代码的脚本文件是 OPTION_EXPL_NO_DECLARE.VBS，本书中每一章的代码都可以从 www.wrox.com 下载)。

```
IngFirst = 1
IngSecond = 2
IngThird = IngFirst + IngSecond
MsgBox IngThird
```

尽管这三个变量都没有显式的声明，但 VBScript 并不在意。这段代码能正常的执行，最终会出现一个对话框并显示数字 3。这似乎很方便。但是，这种方便的代价是高昂的。看下一个例子(OPTION_EXPL_MISSPELLING.VBS)。

```
IngFirst = 1
IngSecond = 2
IngThird = IngFirst + lgnSecond
MsgBox IngThird
```

这段代码不是跟前一个例子一样吗？再仔细看一下。您发现第三行中的拼写错误了吗？

在输入大量的脚本代码时，这是一个很容易犯的错误。麻烦的是拼写错误并不会给 VBScript 带来任何麻烦。它会认为这个拼写错误只是一个新的变量，给它分配内存并将其子类型初始化为 **Empty**。要 VBScript 对一个空变量做数学运算，它就会将其视为 **0**。所以这段代码运行时，对话框显示的是数字 **1**，而不是您所期望的数字 **3**。

在这个简单得没有任何实际意义的脚本中查找和修复这个错误是很容易的，但是如果脚本中有成百上千行代码呢？如果不是 **1 加 2 等于 3**，而是 **78523.6778262 加上 2349.25385 然后再除以 4.97432** 呢？查看结果就能发现其中的数学错误吗？如果不行，您甚至不会发觉您的脚本有错误。这在 VBScript 中是非常现实的问题，可能要过几周的时间您才会发现这个错误——更糟糕的是，可能是您的老板或客户发现了这个错误。

怎样才能避免这个问题？答案就是 **Option Explicit** 语句。要做的就是将 **Option Explicit** 语句放到脚本的开头，所有的其他语句之前。这就是告诉 VBScript，您的代码要求所有的变量在使用之前都必须先显式地声明。现在 VBScript 就不再会允许您在代码中不做声明就引入新的变量。下面是一个例子(**OPTION_EXPL_ERROR.VBS**)。

```
Option Explicit
```

```
Dim lngFirst
```

```
Dim lngSecond
```

```
Dim lngThird
```

```
lngFirst = 1
```

```
lngSecond = 2
```

```
lngThird = lngFirst + lngSecond
```

```
MsgBox lngThird
```

注意，其中 **Option Explicit** 语句被添加到了代码的头部。由于添加了 **Option Explicit** 语句，所以现在必须在使用变量之前先做声明，这就是您所看到的紧接着 **Option Explicit** 语句的三行代码。并且，要注意倒数第二行还是有拼写错误。这是为了说明使用未声明的变量会有什么后果。如果尝试运行这段代码，VBScript 会终止运行，并提示以下错误：**Variable is undefined: 'lngSecond'**。这是个好消息，因为现在您就知道要修复这个错误。只要使用了 **Option Explicit** 语句，VBScript 就会捕获变量的输入错误。

Option Explicit 语句的另一个优点就是它对整个脚本文件都有效。本书至今没有对此做

深入探讨，但是一个脚本文件中可以包含多个过程、函数以及类定义，而每个类定义又可以有它自己的多个过程和函数(详见第 8 章)。但只要您将 `Option Explicit` 语句放在脚本的头部，那么它就是对文件中所有的代码都起作用。

从今天开始就养成这个好习惯：在编写新的脚本文件时，首先要做的事情就是在文件最开始输入 `Option Explicit`，并回车。这就能避免烦人的代码输入错误，跟您一起编写脚本的同事(或客户)也会感谢它。

4.2 变量命名

VBScript 中有一些给变量命名的原则。这些原则很简单，但是它有助于您给变量取一个清晰、有用、易懂的名称。

原则一：VBScript 变量名的第一个字符必须是字母。

字母就是 `a~z` 之间的字符(大小写均可)。常见的非字母字符有：数字、标点符号、数学运算符及其他特殊字符。例如，这些是合法的变量名：

- `strName`
- `Some_Thing`
- `Fruit`

而这些是非法的：

- `+strName`
- `99RedBalloons`
- `@Test`

原则二：数字和下划线(`_`)可以用于变量名，但是其他所有的非字母字符都是非法的。

VBScript 中的变量名不能含有除了数字和字母外的其他东西。唯一的一个例外就是下划线(`_`)。(有些程序员可能会发现在分割变量名中不同单词时，下划线是很有用的(例如，`customer_name`)，但有些程序员喜欢用大小写的混合来达到同样的目的(例如，`CustomerName`)。例如，下面是合法的变量名：

- `lngPosition99`
- `Word1_Word2_`

- `bool2ndTime`

而这些则是非法的：

- `str & Name`
- `Something@`
- `First * Name`

原则三：VBScript 变量名的长度不能超过 255 个字符。

变量名长度最好不要超过 20 个字符，但是 VBScript 允许的最大变量长度是 255 个字符。

这些给变量命名的原则都很容易遵守，但是合法的变量名和清晰、有意义且易懂的变量名之间还是有很大差别的。VBScript 确实允许使用像 `X99B2F012345` 这样的变量名，但这并不意味着这样做是个好主意。

变量名应该清楚地说明变量的用途。如果将用户名存放到变量中，那么像 `strUserName` 就是一个不错的变量名，因为这可以消除程序员对这个变量用途的疑惑。好的变量名不仅仅可以减少代码中的错误，还可以使代码本身更容易阅读和理解。

另一种程序员觉得很有用的技术就是之前多次提及的“匈牙利命名原则”；在这几章中都使用了该方法。（即使是觉得它没什么用的 VBScript 程序员也会用到它，因为在 VBScript 和 Visual Basic 的世界中，匈牙利原则很常见，几乎就是必备的技术。）匈牙利命名原则很简单，就是通过给变量名添加前缀说明程序员希望这个变量所存放的数据类型。

例如，变量名 `strUserName` 并不只是表示这个变量存放的是用户名，还说明该变量的子类型应该是 `String`。同样，变量名 `lngFileCount` 表示这个变量是一个文件数量的计数器，并且其子类型应该是 `Long`。

本书的附录 B 中还有一些变量命名的指导，包括推荐使用的数据类型前缀清单。

4.3 过程和函数

现在要学习的是过程和函数这两个概念，这是构建更复杂的脚本所必需的基础。使用过程和函数能将脚本中的代码模块化，放入有名称的具有特定功能的代码块中。模块化使您可以有组织地考虑复杂的问题，提高代码的可读性和可理解性，并且使得在同一个脚本中创建可复用的代码成为可能。

函数(function)就是调用时有返回值、有名称的代码块，而过程则是没有返回值的代码块。

让我们来解释一下这句话中的一些新概念。

- 有名称的代码块：指的是逻辑上有关联的若干行代码，它们一同执行完成某个任务。过程和函数都是“有名称”的代码块，因为要明确地指明代码的边界并给它一个名称。例如，可以将处理用户订单的代码块放入名为 `ProcessCustomerOrder()` 的过程中。
- 调用代码：意思就是调用过程或函数的代码。给代码块命名的一个主要目的就是其他代码可以通过这个名称来使用该代码块。程序员通常将这称为“调用”。例如，在前面几章中，已经看到代码中用 `MsgBox()` 过程在对话框中显示一条消息。调用 `MsgBox()` 过程的脚本代码就是调用代码(calling code)，而 `MsgBox()` 就是被调用(called)的过程。
- 返回值：部分具名代码块会给调用代码返回值。过程没有返回值，而函数则有返回值。有时需要有返回值的代码块，而有时则不需要。正如您已经用到的，`MsgBox()` 过程就没有返回值(但是若您有要求，它也可以有返回值——`MsgBox()` 很有意思，因为您可以将其作为过程使用，也可以将其作为函数。)只要给 `MsgBox()` 传递一个要显示的值，它就会将这个值显示给用户，当用户单击 **OK** 按钮时，就会接着执行后面的代码。另一方面，`CLng()` 函数会给调用代码返回一个值。例如，在下面的代码中，`CLng()` 函数会返回 `Long` 子类型的 12，而返回的值则存放在 `lngCount` 变量中。

```
lngCount = CLng("12")
```

4.3.1 过程的语法

作为过程的具名代码块是用 `Sub` 关键字来识别的。“`Sub`”是“`subprocedure`”的缩写，这是“`procedure`”的另一种说法。用下面的语法声明过程：

```
[Public|Private] Sub Name ([Argument1],[ArgumentN])
```

```
[ code inside the procedure ]
```

```
End Sub
```

有时用“过程”这个词泛指过程和函数，但本章中的“过程”这个词专指的是过程。

关于这个语法要注意以下几点：

- 可以在 **Sub** 关键字前使用关键字 **Public** 或 **Private**，但实际上这些关键字只有在类内部才有用，有些过程要对外部可见，而有些则不可见(关于类，详见第 8 章)。
- 在 **Windows** 脚本宿主文件(也就是本书中所有的.VBS 文件例子)中，关键字 **Public** 和 **Private** 实际上并不会起作用，因为没有过程、函数或变量可以被其他文件中的脚本访问。.VBS 文件本身是个孤岛；其他的脚本无法访问其内部的代码。
- 在 **VBScript** 类或其他跟 **Public** 和 **Private** 关键字相关的情况中，如果没有指定的话，默认的是 **Public**。
- 必须用 **End Sub** 关键字指定过程的边界。在 **Sub** 和 **End Sub** 之间，普通的 **VBScript** 语法都是有效的。
- 过程的命名原则跟变量的命名原则(见本章前面的“变量命名”)是一样的。使用能说明过程的功能及其内部代码的作用的简洁有意义的过程名称是个很好的想法。尽量用动词—名词组合给过程命名，比如 **ProcessOrder** 和 **GetName**。

过程和函数可以接收参数(argument，也就是自变量)，不过它们经常无需参数。参数就是一个记录着传递给过程或函数的信息的值，过程和函数内部的代码可以访问这个值。下面是简单的过程，没有用到参数。

```
Sub SayHelloToBill
```

```
    MsgBox "Hello, Bill. Welcome to our script."
```

```
End Sub
```

但是，如果没有代码调用它的话，过程就是呆着不动，什么都不做。所以下一个例子(PROCEDURE_SIMPLE.VBS)中不仅定义了 **SayHelloToBill()**过程，还调用了它。

```
Option Explicit
```

```
SayHelloToBill
```

```
Sub SayHelloToBill
```

```
    MsgBox "Hello, Bill. Welcome to our script."
```

```
End Sub
```

关于这段代码要注意以下几点：

- 第一行代码(跟着标准的 **Option Explicit** 声明)不属于过程定义，而是脚本的“主体”。您可能还记得，**VBScript** 运行时会首先加载和编译整个脚本，然后从头开始执行脚本的主体。这个例子中，编译器处理这个脚本，创建 **SayHelloToBill()**过程的定义，然后运行时会从上到下地执行脚本的主体。这个脚本的主体只有一行：

SayHelloToBill

- 可以将这个脚本的主体部分理解为一个操控木偶的人，而您在脚本中创建的这些过程和函数(以及类)就是木偶。您所做的不仅仅是设计并构建这些木偶，还要为操控木偶的人编排动作。本章稍后在讨论脚本设计策略时，将再回到这个问题。
- **Public/Private** 关键字被省略了，正如之前所解释的，因为在 **Windows** 脚本宿主中他们不会起作用。
- 过程名后面没有圆括号。之所以这样做是因为这个过程无需任何参数。
- 过程内部的代码被缩进了，所以看上去像是“嵌”在周围的代码中。这不是必须的，但是这是个通常的规范，因为这使得代码易于阅读和理解。缩进说明了过程与其中代码的层次结构。巧妙地在脚本中使用空白(缩进与空行)是一种与您自己和其他察看这个脚本的程序员交流的重要方式。如果将所有的东西塞在一起，而没有缩进，甚至连您自己都将无法理解这个脚本。

下面是另一个关于过程例子，它有一个参数(**PROCEDURE_ARGUMENT.VBS**)。

Option Explicit

GreetUser "Bill"

Sub GreetUser(strUserName)

**MsgBox "Hello, " & strUserName & _
". Welcome to our script."**

End Sub

注意新加入的 **strUserName** 参数，并对过程名称作了相应的修改，这使得该过程更加的通用，提高了它的可重用性。前一个例子的缺点在于其将用户的姓名固定为 **Bill**。这个例子有同样的问题，但是它向适应不叫 **Bill** 的用户更近了一步——欲知详情，请继续阅读。

4.3.2 函数的语法

函数的语法跟过程的语法是一样的，唯一的区别在于将关键字 **Sub** 改为 **Function**。

```
[Public|Private] Function Name ([Argument1],[ArgumentN])
```

```
[ code inside the function ]
```

```
End Function
```

命名原则、**Public/Private** 的关系以及参数的声明在函数和过程中都是一样的。正如前面提到的，函数和过程的区别就在于函数会给调用者返回一个值。下面这个例子说明了函数的语法，以及函数中的代码如何设置函数的返回值(FUNCTION_SIMPLE.VBS)。

```
Option Explicit
```

```
Dim lngFirst
```

```
Dim lngSecond
```

```
lngFirst = 10
```

```
lngSecond = 20
```

```
MsgBox "The sum is: " & AddNumbers(lngFirst, lngSecond)
```

```
Function AddNumbers(lngFirstNumber, lngSecondNumber)
```

```
    AddNumbers = lngFirstNumber + lngSecondNumber
```

```
End Function
```

AddNumbers 可能并不是世界上最有用的函数，但是它很好地说明了两件事情：

- 注意这个函数有两个参数，**lngFirstNumber** 和 **lngSecondNumber**。这两个参数用于函数的内部。
- 注意指定返回值的方法是在函数内部的代码中使用函数名。也就是下面这一行的作用：

```
AddNumbers = lngFirstNumber + lngSecondNumber
```

看上去就像是函数内部有一个与函数本身同名的未声明的变量。通过设置这个隐藏变量的值就能设定函数的返回值。可以在函数内部的任何地方做这个事情，可以像处理普通变量那样反复地修改函数的返回值。如果您在函数中多次设置了返回值，那么其中在离开函数前

最后执行设置返回值的那一行代码所设置的值才是真正的返回值。

下面将过程和函数一起使用，来演示一下如何嵌套地使用函数和过程(PROCEDURE_ FUNCTION_NESTED.VBS)。

```
Option Explicit
```

```
GreetUser
```

```
Sub GreetUser
```

```
    MsgBox "Hello, " & GetUserName & _  
        ". Welcome to our script."
```

```
End Sub
```

```
Function GetUserName
```

```
    GetUserName = InputBox("Please enter your name.")
```

```
End Function
```

要注意 `GreetUser()` 过程是如何调用 `GetUserName()` 函数的。函数和过程可以这样一起使用，这就是构建程序的方法。将代码分割成各个模块构成用途专一的过程和函数，然后以合理的方式将它们集成在一起。

这个例子很好地说明了一个函数设计的重要原则，我们会在本章“脚本和函数的设计策略”一节中进一步地介绍该原则。沿用这个例子中关于木偶的比喻，您已经设计了要用到另一个木偶(`GetUserName()`)的木偶(`GreetUser()`)，而操控木偶的人(脚本的主体)却无需知道这些情况。迄今为止，这看上去都很不错，但在这个嵌套的过程和函数设计中有一个严重的缺陷。

`GreetUser()` 过程跟 `GetUserName()` 函数耦合在一起是没有必要的。这意味着 `GreetUser()` “知道”并依赖于 `GetUserName()` 函数；它不能独立的工作。`GreetUser()` 过程之所以依赖于 `GetUserName()` 函数，是因为它调用了这个函数；如果不从 `GetUserName()` 获得姓名的话，`GreetUser()` 就不知道该问候谁。

代码模块间一定的耦合是有必要的，而且是有好处的，但是有些耦合却是您不需要的，要避免的。程序内部耦合越紧密，其复杂性也越高。有些复杂性无法避免，但是必须要尽量地降低复杂性。函数和过程间杂乱的耦合就是著名的“面条代码”(spaghetti code)——也就是说无法理清代码的逻辑结构，因为其中的逻辑缠绕在一起，看上去毫无规律可循。

下面是同一个脚本的另一个版本，消除了不必要的耦合，并且将控制权和信息放回到木

偶的操控者手中。似乎让 `GreetUser` 这个操控木偶的人聪明地灵活使用 `GetUserName` 木偶是件好事情，但是从长远来看，将 `GreetUser` 设计成自我满足并且聪明得正好符合要求是个更好的选择。

Option Explicit

GreetUser GetUserName

Sub GreetUser(strUserName)

```
    MsgBox "Hello, " & strUserName & _  
        ". Welcome to our script."
```

End Sub

Function GetUserName

```
    GetUserName = InputBox("Please enter your name.")
```

End Function

这个程序的逻辑是一样的，但是现在将 `GreetUser()` 和 `GetUserName()` 分离开了。通过将 `strUserName` 参数存放到 `GreetUser` 中，而不是在脚本中将两个函数放在一起，而且两者间相互不“知道”对方。下面是这个脚本中最有意思的一行代码。

GreetUser GetUserName

`GetUserName()` 的返回值传递给 `GreetUser()` 函数的 `strUserName` 参数。

关于函数的语法最后再做一点说明：熟悉其它语言的程序员会发现不能声明函数返回值的类型。如果您记得 `VBScript` 只支持一种数据类型——`Variant`，这就不难理解了。因为所有的变量都是 `Variants`，所以无需将函数指定为某种数据类型。

很多 `VBScript` 常用的一种使代码清晰的方法是像变量那样对函数名也使用匈牙利类型前缀。例如，`GetUserName()` 可以命名为 `strGetUserName()`。但是，如果您选择遵循这个方法，那在给函数和变量命名时要使得两者能被轻易的区分出来就非常重要。使用动词—名词形式的函数名称对此很有帮助，这样 `strUserName` 就显然是个变量，而 `strGetUserName` 则显然是个函数。

4.3.3 调用过程和函数

在之前关于过程和函数的例子中，您可能已经注意到了调用过程和调用函数的不同之处。这确实有区别，并且 VBScript 很在意这一点(参见表 4-1)。

表 4-1

调用过程的合法方式	调用过程的非法方式
GreetUser "Bill"	GreetUser("Bill")
Call GreetUser("Bill")	Call GreetUser "Bill"

这两种合法的方式在功能上是等价的，选择哪一种主要取决于您自己的喜好。有些人说第二种方式(使用 Call 关键字)更清晰，但也有人讨厌打更多的字。有些人喜欢第二种方式是因为它更像 VBScript 的前辈 BASIC 编程语言。两个方式都很常见，Visual Basic 和 VBScript 程序员对这两者都很习惯。在调用过程(相对的则是函数)时，若选择不使用 Call 关键字，那就不能将传递给该过程的参数放在圆括号中。同样，如果要使用 Call 关键字，那么圆括号就是不可缺少的。这就是使用方法。

您将发觉本书中大部分使用的都是第一种方法，所以您能看出我们的选择。(实际上，如果本节中的语法让您觉得困惑，只要跟着例子中脚本做就是了——至少我们选择了这么做)。

调用函数的规则有点不同(参见表 4-2)。

表 4-2

调用函数的合法方式	调用函数的不合法方式	备 注
IngSum = AddNumbers(10, 20)	IngSum = AddNumbers 10, 20	要获得函数的返回值，就不能用 Call 关键字，并且一定要将参数放在圆括号中。没有圆括号是非法的
Call AddNumbers (10, 20)	IngSum = Call AddNumbers (10, 20)	如果不需要接收函数的返回值则可以使用 Call 关键字，但是必须使用圆括号。在接收返回值时用 Call 关键字是非法的
AddNumbers 10, 20	AddNumbers (10, 20)	可以同时忽略 Call 关键字和返回值，但这样也必须忽略圆括号

这就产生了一个问题：什么时候调用函数会不需要返回值呢？前两个例子中的代码或许回答了这个问题，但是还有点晦涩。一般来说，函数之所以是函数就是因为它们有返回值，而我们调用函数的目的也是为了获得它的返回值。

但是也有些情况下在调用函数时会忽略返回值，将其视为一个过程。**MsgBox()**的使用就是一个例子。**MsgBox()**可以作为过程使用，也可以作为函数使用，这取决于您使用它的原因。**MsgBox()**有双重目的。它可以只是显示一条消息，也就是您之前使用它的方法。它也可以作为函数，用来识别用户单击的是对话框中的哪个按钮。下面这个脚本说明了这两种使用 **MsgBox()**的方法(MSGBOX_DUAL.VBS)。

```
Option Explicit
```

```
Dim lngResponse
```

```
Dim strUserName
```

```
lngResponse = MsgBox("Would you like a greeting?", vbYesNo)
```

```
If lngResponse = vbYes Then
```

```
    strUserName = GetUserName
```

```
    GreetUser strUserName
```

```
End If
```

```
Sub GreetUser(strUserName)
```

```
    MsgBox "Hello, " & strUserName & _  
        ". Welcome to our script."
```

```
End Sub
```

```
Function GetUserName
```

```
    GetUserName = InputBox("Please enter your name.")
```

```
End Function
```

在这行代码中是将 **MsgBox()**作为函数使用的。

```
lngResponse = MsgBox("Would you like a greeting?", vbYesNo)
```

MsgBox()有一些可选的参数，其中第二个参数可以指定对话框中只是显示 **OK** 按钮还是显示更多的按钮。这样使用 **MsgBox()**函数所产生的对话框如图 4-1 所示。



图 4-1

如果用户单击的是 Yes 按钮，`MsgBox()`函数会返回一个值(这个例子中就是 `vbYes`)。如果用户单击的是 Yes，那么最终就会调用我们熟悉的 `GreetUser()`过程，其中将 `MsgBox()`作为过程调用而不是函数。

注意：这个例子中的 `vbYesNo` 和 `vbYes` 是 VBScript 内建的“具名常量”，就像是值固定且不变的变量。(本章稍后会介绍具名常量)。

4.3.4 可选参数

正如您在前一节中刚刚看到的 `MsgBox()`函数，过程和函数可以有可选参数(optional argument)。如果参数是可选的，那么就不一定要给它传递值。通常，若不给它传递值，可选参数就会有一个默认值。可选参数通常在参数列表的最后面；强制性的参数在最前面，接着是可选参数——但是您自己用 VBScript 编写的过程和函数不能有可选参数。如果有需要的话，可以定义一个强制性参数，在其为某个特定值(比如 `Null`)时就说明这个参数要被忽略。这种“仿制”的可选参数在某些受限情况下可以帮助您实现某些功能，但是我们并不鼓励使用这种技术。

4.3.5 退出过程或函数

在过程或函数内部的最后一行代码运行完之后自然就会退出。但是，有时您需要立即结束一个过程而不是等到代码全部运行完。这样，您可以使用 `Exit Sub` 语句(过程)，也可以使用 `Exit Function` 语句(函数)。只要出现 `Exit` 语句，那么代码的流程就会返回到调用者。

前面作为例子的简单函数中没有明显地使用 `Exit Sub` 或 `Exit Function`。通常这些语句用于更复杂的情况中遇到逻辑终止点或到达逻辑终点。这就是说，很多程序员不鼓励在不必要

的情况下使用这些语句。用这段代码作为例子(EXIT_SUB.VBS)。

```
Option Explicit
```

```
GreetUser InputBox("Please enter your name.")
```

```
Sub GreetUser(strUserName)
```

```
    If IsNumeric(strUserName) or IsDate(strUserName) Then
```

```
        MsgBox "That is not a legal name."
```

```
        Exit Sub
```

```
    End If
```

```
        MsgBox "Hello, " & strUserName & _
```

```
        ". Welcome to our script."
```

```
End Sub
```

注意 GreetUser() 过程中的 Exit Sub 语句。这个例子中还添加了一些语句以确保这个姓名不是数字或日期，如果它是数字或日期就提醒用户并使用 Exit Sub 终止该过程。但是，很多程序员觉得更好的办法是不使用 Exit Sub，就像下面这个例子(EXIT_SUB_NOT_NEEDED.VBS)。

```
Option Explicit
```

```
GreetUser InputBox("Please enter your name.")
```

```
Sub GreetUser(strUserName)
```

```
    If IsNumeric(strUserName) or IsDate(strUserName) Then
```

```
        MsgBox "That is not a legal name."
```

```
    Else
```

```
        MsgBox "Hello, " & strUserName & _
```

```
        ". Welcome to our script."
```

```
    End If
```

```
End Sub
```

注意这里没有使用 Exit Sub，而是用了 Else 语句。这里的设计原则就是过程只能有一个退出点，这就是过程结尾的隐藏退出点。从定义就能看出，有 Exit 语句的过程或函数必然有多个退出点，很多程序员都认为这是糟糕的设计。这个原则的出发点就在于多个退出点更易于产生错误而且难于理解。

不要太担心您的代码是否符合某些理想的设计——尤其是在这样小规模简单脚本中。更重要的考虑是 `Exit Sub` 和 `Exit Function` 的使用会破坏脚本的逻辑流程。因为它们用于终止逻辑流程，并且会在代码中产生跳跃，过度地使用会导致逻辑混乱，容易产生错误。这就是为什么有些人反对使用这些语句的原因。

4.4 变量作用域、声明及生存期

变量作用域(variable scope)和生存期(lifetime)是密切相关的概念。变量作用域就是变量可见并能被访问的范围；在这个范围之外，变量就是不可见的，或不可访问的，并且该变量也不能与该范围之外的代码发生交互。变量的作用域跟其生存期是直接相关的。(变量的生存期会在下一节中详细介绍。)

4.4.1 理解变量作用域

在 VBScript 中有三种变量作用域：

- **脚本级(script-level)作用域：**变量在整个脚本文件的代码中都是有效的。在脚本文件(比如 Windows 脚本宿主的.VBS 文件或是活动服务页面的.ASP 文件)“主体”中声明的变量的作用域就是整个脚本。
- **过程级(procedure-level)作用域：**变量在过程或函数中有效。过程之外的其它代码，甚至是同一个脚本文件中的代码也不能访问过程级变量。过程级作用域也被称为“局部”作用域，程序员通常将被声明为过程级的变量称为“局部变量”。
- **类级(class-level)作用域：**这是一种包含属性和方法的逻辑分组的特殊结构。在 VBScript 中，脚本中用 `Class...End Class` 代码块定义类。类定义的主体中用 `Private` 语句声明的变量的作用域就是类级的。意思就是类中的其他代码可以访问这个变量，但是类定义之外的代码，即使是同一个脚本文件中的代码也不能访问这个变量。类级作用域将在第 8 章中介绍。

有三种语句可以用于声明变量：`Dim`、`Private` 和 `Public`。(前面介绍的 `ReDim` 语句也可以归于这些类别，但是它是专门用于改变已经声明的数组变量的维度的。)在不同的情况使用不同的语句声明变量，具体取决于变量的作用域：

- **Dim:** 这个语句用于声明脚本或过程级作用域的变量。所有被声明为脚本级的变量自动地在整个脚本文件中有效，无论使用的是 **Dim**、**Private** 还是 **Public**。要声明过程内部的变量(也就是局部变量)，必须使用 **Dim**。不能在过程内部使用 **Public** 和 **Private**。如果用于类级的变量，**Dim** 的效果跟 **Public** 是完全相同的。
- **Private:** 可以在脚本或类级作用域使用 **Private** 语句，但是不能在过程或函数内部使用。如果用于脚本级变量，它的作用跟 **Dim** 和 **Public** 是完全相同的。所有在脚本级声明的变量都自动地在整个脚本文件中有效，无论使用 **Dim**、**Private** 还是 **Public**。尽管 **VBScript** 不需要它，但还是有很多程序员喜欢用 **Private** 语句声明脚本级的变量，而将 **Dim** 保留用于过程或函数中的局部变量。为了声明一个私有的类级变量，必须要用 **Private**。所有在类级用 **Dim** 或 **Public** 声明的变量在整个类中都是一个有效的公共属性。
- **Public:** 您可能会用 **Public** 语句声明脚本级作用域的变量，但是在效果上它跟 **Dim** 或 **Private** 是一样的。**Public** 真正有意义的地方就在于类。用 **Public** 声明的类级变量就是这个类的公共属性。**Public** 在脚本级没有意义(脚本构件(script component)除外，详见第 16 章)的原因就在于变量在其存在的脚本文件外是没有用的。所以，**Public** 唯一有意义的就是创建类的公共属性。但要注意，很多 **VBScript** 程序员不鼓励在类中使用 **Public** 变量，而喜欢使用 **Private** 类级变量和 **Property Let**、**Set** 和 **Get** 过程的组合(详见第 8 章)。

这三点中包含了很多规则(您稍后见到的例子会使这些规则变得清晰)，所以遵守这些规则可以使您更容易理解什么时候该用 **Dim**，什么时候该用 **Private**，而什么时候该用 **Public**。

- 在过程中用 **Dim** 声明的变量对于那个过程是局部变量，或在脚本级用 **Dim** 声明变量，**Dim** 是声明变量的全功能关键字。在不以类为基础的脚本或是不作为 **Windows** 脚本构件的脚本中，**Private** 和 **Public** 的效果跟 **Dim** 没有区别。
- 如果您愿意，可以在脚本中用 **Private**(代替 **Dim**)声明整个脚本中都有效的变量。在类中声明只属于类的变量时，**Private** 的使用就变得更重要了。
- **Public** 只用于声明类的公共属性，也可以考虑用 **Private** 变量配合 **Property Let**、**Set** 和 **Get** 过程。尽管在类中 **Dim** 与 **Public** 的效果完全相同，但还是偏向于不在类中使用 **Dim**。

在开始学习变量生存期之前，让我们来讨论一下关于声明的一些小问题。

4.4.2 理解变量声明

在 VBScript 中，可以在一行代码中声明多个变量，但是根据编程风格标准，通常每行代码只声明一个变量，就像前面的例子那样，但这不是一个绝对的规则。例如，脚本程序员编写作为 HTML 文件的一部分的脚本，该脚本会通过 Web 下载，在一行中声明多个变量可以减小文件大小。但是，有时仅仅只是因为程序员喜欢一次声明多个变量。这只是程序员的编程风格问题，对实际结果没有影响。

下面是一个多变量声明的例子。

```
Dim strUserName, strPassword, lngAge
```

这里用 Private 代替了 Dim。无论使用的是 Dim、Private 或 Public，这些规则都是一样的。

```
Private strUserName, strPassword, lngAge
```

但要注意，不能在同一行中声明不同作用域的变量。如果要在类中声明一些 Private 变量以及一些 Public 变量，就必须要用两行代码。

```
Private strUserName, strPassword
```

```
Public lngAge, datBirthDay, boolLikesPresents
```

最后，VBScript 对脚本和过程中使用的变量数量是有限制的。在过程中，过程级变量不能超过 127 个；而在脚本文件中，脚本级变量也不能超过 127 个。但这并不会给您带来任何麻烦。如果您在脚本或过程中使用了这么多的变量，就应该重新考虑一下您的设计，将大的过程分割成多个过程。如果您确实有那么多数据，可以考虑将它们组织成类，每一个类可以有多个属性。

4.4.3 变量生存期

生存期(lifetime)就是在脚本运行时，变量在内存中存在且能使用的时间。变量只有在它的作用域内才有效。一个过程级作用域的变量只有在运行该过程时才有效。当过程结束时，保存该变量的内存就会被释放，就像不存在这个变量一样。同样，脚本级作用域的变量只有在脚本运行时才有效。类似的，类级作用域的变量只有在其它代码使用了基于这个类的对象

时才有效。

通过限制变量的作用域，就能限制变量的生存期。这里有一个要时刻记住的重要原则：必须要尽可能地限制变量的生存期和作用域。因为变量会消耗内存，以及操作系统和脚本引擎的资源，只要变量有效就需要这些资源。在使用变量的过程内部声明变量，就能在该过程没有运行时节约该变量所需的资源。

但实际上，资源消耗并不是限制变量作用域的最重要原因；限制作用域可以减少程序的错误，并提高代码的可理解性和可维护性。如果脚本中有多个过程和函数，而所有的变量都是声明为脚本级的，那这些过程和函数就都能修改这些变量。这种情况下任何代码都能随时修改变量，程序员要维护这种代码就很困难。

只要遵守模块化的良好原则，设计优良的过程和函数自然就会处理好作用域和生存期问题——无需任何额外的努力。如果将脚本的逻辑结构分割成较小的结构(过程和函数)，每个结构的数据都有其自己的作用域边界。尽可能地确保使用局部变量和过程参数，这样每个过程就只能处理其真正需要的数据。

看一下这个说明变量作用域和生存期的例子(SCOPE.VBS)：

```
Option Explicit
```

```
Private datToday
```

```
datToday = Date
```

```
MsgBox "Tommorrow's date will be " & AddOneDay(datToday) & "."
```

```
Function AddOneDay(datAny)
```

```
    Dim datResult
```

```
    datResult = DateAdd("d", 1, datAny)
```

```
    AddOneDay = datResult
```

```
End Function
```

这个脚本有一个 `AddOneDay()` 函数。用 `Dim` 在函数内部声明的变量 `datResult` 的作用域是过程级的，这就说对于函数外部的代码该变量无效。而变量 `datToday` 使用 `Private` 声明的，有脚本级的作用域。变量 `datResult` 只有在 `AddOneDay()` 函数运行时才被激活，而 `datToday` 在整个脚本的生存期内都是激活的。

4.5 脚本和过程的设计策略

再看一下上一个例子(SCOPE.VBS)。您还可以这样设计这个脚本(SCOPE_BAD_DESIGN.VBS)。

```
Option Explicit
```

```
Private datToday
```

```
datToday = Date
```

```
AddOneDay
```

```
MsgBox "Tommorrow's date will be " & datToday & "."
```

```
Sub AddOneDay()
```

```
    datToday = DateAdd("d", 1, datToday)
```

```
End Sub
```

这段代码百分之百合法且有效，最终的结果跟原来的例子是一样的。因为 `datToday` 有脚本级的作用域，在 `AddOneDay()`(现在将其从函数改为了过程)内的代码中也是有效的；您只是简单地用 `AddOneDay()`直接修改 `datToday`。这确实能行，但是这种技术会带来很多麻烦。

`AddOneDay` 函数丧失了可重用性。现在 `AddOneDay()`与脚本级变量 `datToday` 紧密地耦合在一起。如果要将 `AddOneDay()`复制到其它脚本中重用，工作将会变得困难。当 `AddOneDay()`是一个独立函数，无需知道外界的数据或代码时，它完全是可移植可重用的。

4.5.1 限制代码读取和修改变量

这并不是为了彻底地避免脚本级变量。在过程中使用脚本级变量也并不一定就是坏事。这完全取决于您是怎么做的。要注意的是限制脚本中直接读取和修改脚本级变量的地方的数量。还要使这些地方很明显，这样其他阅读代码的人能知道它是如何工作的。

回忆一下前面的木偶和木偶操控者。如果木偶过度地干涉木偶操控者的脚本级变量，那么木偶操控者就很难控制所有的事情，这就容易产生错误。

看一下这个脚本(SENTENCE_NO_PROCS.VBS)。

```
Option Explicit
```

```
Dim strSentence
```

```

Dim strVerb

Dim strNoun

'Start the sentence
strSentence = "The "

'Get a noun from the user
strNoun = InputBox("Please enter a noun (person, " & _
    "place, or thing).")

'Add the noun to the sentence
strSentence = strSentence & Trim(strNoun) & " "

'Get a verb from the user
strVerb = InputBox("Please enter a past tense verb.")

'Add the verb to the sentence
strSentence = strSentence & Trim(strVerb)

'Finish the sentence
strSentence = strSentence & "."

'Display the sentence
MsgBox strSentence

```

这个脚本没有什么实际意义，它根据用户的输入通过一系列的步骤构建出一个简单的句子。所有的代码都堆在一起，没有过程和函数，所有的代码都共享访问脚本级变量。

4.5.2 将代码分割成过程和函数

下面是同一个程序，但是根据我们那个关于木偶的比喻将其分割成了若干过程和函数(SENTENCE_WITH_PROCS.VBS)。

```

Option Explicit

Dim strSentence

strSentence = "The "

strSentence = strSentence & GetNoun & " "

```

```

strSentence = strSentence & GetVerb

strSentence = strSentence & GetPeriod

DisplayMessage strSentence

Function GetNoun

    GetNoun = Trim(InputBox("Please enter a noun (person, place, or thing)."))

End Function

Function GetVerb

    GetVerb = Trim(InputBox("Please enter a past tense verb."))

End Function

Function GetPeriod

    GetPeriod = "."

End Function

Sub DisplayMessage(strAny)

    MsgBox strAny

End Sub

```

在这个版本中只是最开始掌控整个脚本逻辑结构的代码中有一个脚本级的变量: 构建一个句子并显示给用户。脚本开头的代码使用了一系列的函数和一个过程来完成真正的工作。每个函数和过程的工作都很专一, 并且没有动用任何脚本级数据。所有的这些函数和过程对整个大局都没有了解。这可以减少其中的错误, 易于理解, 并且更具有可重用性。

另一个好处就是您无需阅读整个脚本就能知道这个脚本的用途。您只要阅读这五行代码就能了解整个脚本。

```

strSentence = "The "

strSentence = strSentence & GetNoun & " "

strSentence = strSentence & GetVerb

strSentence = strSentence & GetPeriod

DisplayMessage strSentence

```

在了解了程序的整体流程后, 如果想要深入了解其中的某个具体步骤是如何实现的, 也能知道该去什么地方找。尽管这只是一个非常简单的例子, 在真实世界中不值一提, 但它说

明了如何将您的脚本模块化。

4.5.3 关于脚本设计的忠告

下面这些忠告有助于您设计脚本：

- 专用于某项工作，只有少量代码的简单脚本文件可以被编写成一个独立的代码块，其中不包含过程或函数。
- 随着脚本越来越复杂，就要尝试用过程、函数或类将其按逻辑分割。
- 在分割代码时，要将协调整体的代码放在脚本文件头部。
- 将每个过程和函数设计成功能专一的代码块。给过程取一个好名字，说明它的功能。
- 每个过程和函数都无需知道脚本文件的整体情况。换句话说就是单个的过程或函数对外界一无所知，只知道它自己所要完成的工作。
- 尽量避免在过程和函数中读写脚本级变量。当过程和函数需要访问脚本级变量中的数据时，最好将其作为一个参数传递给有需要的过程或函数，而不是让过程或函数直接去访问这个脚本级变量。
- 如果要修改脚本级变量，最好使用脚本文件开头的协调代码来做修改。

4.6 ByRef 和 RyVal

以下说明过程和函数的参数默认传递方式为 **ByVal**，这些与 210 页的讲述有冲突，实际测试结果表明，默认传递方式为 **ByRef**。

在介绍过程和函数的参数时跳过了一个概念：传址(by reference)和传值(by value)。一个参数是传址还是传值取决于过程或函数定义中的声明。传址的参数用 **ByRef** 关键字说明，而传值的参数既可以用 **ByVal** 关键字说明也可以不加任何说明——也就是说，没有明确地指定是哪一种的话，**ByVal** 就是默认值。

这到底是什么意思？您可能注意到如果变量作为参数传递给过程或函数，过程中代码可以像使用过程中其他局部变量那样使用这个参数。将某个参数指定为传值的意思就是过程中的代码不能永久地修改变量中的值。

通过传值，过程中的代码可以修改这个参数，但是这种修改是临时的；一旦过程终止，

对这些变量或参数的修改就会像过程中其它的局部变量一样被抛弃。而如果传址，就像是调用者与被调用的过程或函数共享这个变量；过程对传址参数的任何修改都是“永久”的，就是说在控制流程回到调用代码后这些修改仍然有效。

来看几个例子。下面这个过程有两个参数，一个 ByVal，一个 ByRef(BYREF_ BYVAL.VBS)。

```
Option Explicit
```

```
Dim lngA
```

```
Dim lngB
```

```
lngA = 1
```

```
lngB = 1
```

```
ByRefByValExample lngA, lngB
```

```
MsgBox "lngA = " & lngA & vbNewLine & _
```

```
    "lngB = " & lngB
```

```
Sub ByRefByValExample(ByRef lngFirst, ByVal lngSecond)
```

```
    lngFirst = lngFirst + 1
```

```
    lngSecond = lngSecond + 1
```

```
End Sub
```

运行这段代码会生成如图 4-2 所示的对话框。



图 4-2

关于这段代码要注意以下几点：

- 变量 lngA 和 lngB 都是脚本级的，都在 ByRefByValExample()过程外初始化为 1。
- lngFirst 参数被声明为 ByRef，lngSecond 则是 ByVal。
- 在过程内部两个参数都增加 1。

- 在对话框中，在过程终止后只有 lngA(传址)的值是 2。

因为只有 lngA 是传址的，所以也只有 lngA 被修改了。由于 lngB 是传值的，所以在 ByRefByValExample()过程内对其做的修改对该过程外的 lngB 变量没有影响。

大部分时候(基本上可以说是几乎所有时候)，都会用 ByVal 处理过程和函数的参数。跟前面几节关于变量作用域和生存期讨论的原因一样，使用 ByVal 比较安全和直截了当。并不是说用 ByRef 就一定不好，有时使用 ByRef 是一个更好的选择。但是在您确实需要 ByRef 之前最好还是坚持使用 ByVal。

例如，这个脚本在不必要的情况下使用了 ByRef(BYREF.VBS)。

```
Option Explicit
```

```
Dim strWord
```

```
strWord = "alligator"
```

```
AppendSuffix strWord
```

```
MsgBox strWord
```

```
Sub AppendSuffix(ByRef strAny)
```

```
    strAny = strAny & "XXX"
```

```
End Sub
```

下面是一个比较好的例子，消除了不必要的 ByRef(BYVAL.VBS)。

```
Option Explicit
```

```
Dim strWord
```

```
strWord = "alligator"
```

```
strWord = AppendSuffix(strWord)
```

```
MsgBox strWord
```

```
Function AppendSuffix(ByVal strAny)
```

```
    AppendSuffix = strAny & "XXX"
```

```
End Function
```

这个例子将过程改为函数，这样就无需 ByRef 关键字。还要注意这个例子中的 ByVal 关键字也是可选的；不使用它，效果也是一样的，因为 ByVal 是默认值。

4.7 文字常量和具名常量

本节会介绍一些在程序员中存在争议的概念。什么时候在代码中使用文字常量？什么时候应该使用具名常量？一个极端的情况是有些程序员从不使用具名常量代替文字常量(可能是他们的选择，也可能是因为他们不知道这种技术)。而另一种极端的情况是，有些程序员从不使用文字常量，所有的文字常量都用具名常量代替。折中的情况是保持一个平衡，既使用一些文字常量，也懂得使用具名常量增加程序的可理解性并降低犯输入错误的可能性。

在阅读了这一节的讨论后，您应该会知道如何在文字常量和具名常量中做出选择。明白使用具名常量的好处和风险，这样您就可以自己做出抉择在什么时候使用它们。

4.7.1 什么是文字常量

文字常量(literal)就是代码中没有存放在变量或具名常量中的静态数据。文字常量可以是字符串或文本、数字、日期或布尔值。例如，下面这段代码中的单词“Hello”就是一个文字常量。

```
Dim strMessage  
  
strMessage = "Hello"  
  
MsgBox strMessage
```

下面代码中的日期 08/31/69 也是一个文字常量。

```
Dim datBirthday  
  
datBirthday = #08/31/69#  
  
MsgBox "My birthday is " & datBirthday & "."
```

字符串“My birthday is ”也是文字常量。也就是说文字常量不需要存放在变量中。看一个比较复杂的例子，下面代码中的 True 值也是文字常量。

```
Dim boolCanShowMsg  
  
boolCanShowMsg = True  
  
If boolCanShowMsg Then  
    MsgBox "Hello there."  
End If
```

很多情况下，代码中使用文字常量就够了，尤其是代码量较少，复杂度比较低的简单脚本。程序员总会用到文字常量。他们并没有什么固有的坏处。但是，可以举出很多例子，其中使用具名常量比使用文字常量更好。

4.7.2 什么是具名常量

具名常量(named constant)类似于变量，它也是内存中存放数据的地址的名称。不同之处在于，顾名思义，它是一个常量，在运行时不能修改。而变量是动态的。在代码运行时，变量作用域中的任何代码都可以将其中的值改成其他东西。而具名常量则是静态的。一旦定义好，在运行时任何代码都不能修改它。

在 VBScript 中，用 Const 语句定义常量。下面是一个例子(NAMED_CONSTANT.VBS)。

```
Option Explicit
```

```
Const GREETING = "Hello there, "
```

```
Dim strUserName strUserName = InputBox("Please enter your name.")
```

```
If Trim(strUserName) < > "" Then
```

```
    MsgBox GREETING & strUserName & "."
```

```
End If
```

若用户输入的名字是“William”，那这段代码的结果就是如图 4-3 所示的对话框。



图 4-3

Const 语句定义了名为 GREETING 的具名常量。常量的名称全部大写，这是一种广泛接受的具名常量命名方式。将常量名称定义为全大写就更容易将其与变量区分开，变量通常都是小写的或大小写混合的。另外，因为常量通常全都是大写字母，常量名称中的不同单词是以下划线(_)分割的，例如这个例子(NAMED_CONSTANT2.VBS)。

```
Option Explicit
```

```

Const RESPONSE_YES = "YES"

Const RESPONSE_NO = "NO"

Dim strResponse

strResponse = InputBox("Is today a Tuesday? Please answer Yes or No.")

strResponse = UCase(strResponse)

If strResponse = RESPONSE_YES Then

    MsgBox "I love Tuesdays."

ElseIf strResponse = RESPONSE_NO Then

    MsgBox "I will gladly pay you Tuesday for a hamburger today."

Else

    MsgBox "Invalid response."

End If

```

跟变量一样，具名常量也有作用域。但不能用 `Dim` 语句声明常量，而是用 `Private` 和 `Public` 加上 `Const` 语句。当然，这些作用域修饰符是可选的。在脚本级声明的常量会自动具有脚本级的作用域(也就是说它对脚本文件中所有的过程、函数和类都可用)。在过程或函数内部声明的常量有过程级(亦称为“本地”)的作用域(也就说过程外部的代码不能使用这个常量)。

可以一次声明多个常量，如下：

```
Const RESPONSE_YES = "YES", RESPONSE_NO = "No"
```

最后说明一点，不能用变量或函数来设置常量的值，因为这要求在运行时设置这个值。像前面的例子一样，常量的值必须在设计时就用文字常量定义好。例如，下面这种情况就是不允许的：

```

Dim strMessage

Const SOME_VALUE = strMessage

```

4.7.3 使用具名常量的好处

下面列出了一些使用具名常量所带来的好处。

- 具名常量可以减少程序缺陷。如果要在代码中重复相同的文字常量，犯拼写错误的概率就会随着输入次数而增加。如果在代码中用具名常量代替文字常量，尽管您还是有可能拼错，但是脚本引擎会在运行时捕获这个错误(只要使用了 `Option Explicit`)，而文字常量的拼写错误就很有可能长时间被忽略。
- 具名常量可以使代码清晰易懂。前面例子中使用的文字常量本身就很好理解，添加一个具名常量也无助于使其更清晰易懂。但是，如果在阅读代码时不能立即知道常量的用途，人们就难以理解代码中常量的意义。这种情况在遇到数字常量时尤其突出。数字本身通常都无法说明它在代码中的用途，而用常量代替它可以使其意义更清楚。
- 如果文字常量非常长，或者难以输入，那么用具名常量代替它可以简化代码的输入。例如，若您要在代码的多个地方插入大段法律声明，就最好用一个简短的具名常量代替这一大段文字，从而简化代码的输入。

4.7.4 具名常量的使用原则

本节讨论两个在使用具名常量时应该遵守的原则。

具名常量原则 1： 如果某个文字常量只需使用一次，就无需为其创建具名常量。

在需要通过因特网下载的 **HTML** 嵌入式脚本代码中使用常量时尤其要注意遵守具名常量原则 1。如果一直在客户端 **Web** 脚本中用具名常量，就会增加文件大小，而用户就需要花更多的时间下载。甚至在服务器端的 **Web** 脚本(无需下载到用户浏览器的脚本代码)中，到处使用具名常量也会减慢脚本的运行速度。因为脚本引擎需要在执行使用了这些常量的脚本前处理其中所有的具名常量。

但如果需要在脚本中反复地使用某个文字常量，那么用具名常量代替它确实可以增加代码的可读性，并减少拼写错误。在服务端 **Web** 的 **ASP(Active Server Page)**脚本(见第 20 章)中有一种技术是将具名常量放入“**include**”文件中，从而可以在多个脚本中重复使用它。具名常量很重要，但有时您也需要付出一些代价。

具名常量原则 2： 如果使用具名常量代替文字常量可以使代码更清晰，那就使用具名常量。

在使用数字和日期文字常量时尤其要遵守具名常量原则 2。

在使用数组时，用具名常量代替数组下标也是个不错的做法(见本章的下一节)。

4.7.5 VBScript 的内建常量

很多 VBScript 宿主，比如 Windows 脚本宿主和 ASP 服务器，都支持使用 VBScript 的内建常量。这很有帮助，原因有二：首先，要记住 VBScript 的函数和过程那一大堆作为参数和返回值的数字是很困难的；其次，使用具名常量可以提高代码的可读性。在前面的例子中，您已经看到了一些关于 `VarType()` 和 `MsgBox()` 函数的内建具名常量。

本书的附录 D 中的列表中包含有许多 VBScript 提供的具名常量。您会发现这些常量都很容易通过 `vb` 前缀识别出来。并且，您还会发现这些常量通常都是大小写混合的，而不是全都大写。比如，您可以看看那些作为 `MsgBox()` 函数可选参数的常量(见附录 A 中关于 `MsgBox()` 函数的具体介绍)。

迄今为止，您在本书中已经多次用到 `MsgBox()` 函数的第一个参数。第一个参数是您要显示在对话框中的消息。`MsgBox()` 函数还有一些可选参数，其中第二个参数是按钮参数，用于定义不同的按钮和对话框的图标。下面是一个例子。

`MsgBox "The sky is falling!", 48`

这段代码会生成如图 4-4 所示的对话框。



图 4-4

将数字 48 传给 `MsgBox()` 的第二个参数，就是告诉函数要在对话框中显示惊叹号。可以不用这个没有任何意义的数字 48，可以用 `vbExclamation` 具名常量代替它。

`MsgBox "The sky is falling!", vbExclamation`

这段代码会生成一模一样的对话框，但是更容易理解您的目的。看一下附录 D 中的其他 VBScript 内建常量。学会其中一些常用的常量对您的工作很有帮助，比如 `vbExclamation` 和 `vbNewLine`。

4.8 小结

在本章中我们深入学习了 VBScript 变量的细节。VBScript 不会强制要求您在使用变量前先声明，但是我们强烈建议您在脚本的开头加上 `Option Explicit` 语句，这样 VBScript 就会强制要求变量声明。

VBScript 有一些给变量命名的规则，包括变量名必须以字母开始，而且不能含有特殊字符等。

本章还正式地介绍了过程和函数，包括定义它们的语法以及如何更好地利用它们的设计原则。用过程和函数可以给某段代码设定边界，这也就解释了这些边界是如何定义变量作用域和生存期的。

本章讨论了用于为过程或函数声明参数的 `ByRef` 和 `ByVal` 关键字，最后介绍了具名常量，它可以用来代替您代码中的文字常量，通常都应该这么做。

第 5 章 流程控制

VBScript 语言提供了一些机制用于控制脚本中代码的运行。例如，可以使用分支逻辑跳过若干行代码。也可以使用循环逻辑反复地执行若干行代码。这些技术通常被称为流程控制 (control of flow)。

VBScript 中的分支逻辑用 `If`、`Else` 和 `Select Case` 语句实现。循环则是用 `For`、`Do` 和 `While` 代码块定义的。本章的各个小节会全面地介绍分支和循环，这跟变量一样都是写程序所必须的。如果您对编程比较陌生，那这一章就很重要。跟前面的章节一样，本章讲解了必要的编程基础，但与此同时还会教您 VBScript 中独有的技术和语法。

如果读者熟悉其他的编程语言，那么在本章中只需要关注 VBScript 一些与众不同之处。VBScript 中的分支和循环与其他成熟的过程化语言是差不多的，与 `Visual Basic` 相比则几乎完全一样。如果了解语法细节，附录 A 中的语言参考提供了最合适的信息。

5.1 分支结构

分支就是在代码中做出判断，然后根据判断结果，有选择地执行部分代码。如果您阅读了本书前面的内容，就已经多次见过 VBScript 中常见的分支结构，`If...End If`。本章会详细地讲解 `If...End If`，以及另一种分支结构，`Select...End Select`。

`If...End If` 和 `Select...End Select` 都可以定义代码块，所谓的代码块就是由开始和结束语句限定边界的一段代码。在 `If` 代码块中，开始就是由 `If` 语句定义的，而结尾则是由 `End If` 语句定义的。`Select...End Select` 的模式也差不多。VBScript 要求开始和结束语句成对匹配。如果忘记了结束语句，VBScript 会在运行时产生一个语法错误。

在敲入具体的代码之前，先输入开始和结束语句是一个好习惯。这可以确保您不会忘记输入结束语句，尤其是在其中的代码异常复杂时。这对于您使用多重嵌套的代码也是有帮助的。

5.1.1 If 分支

If...End If 结构可以很简单，也可以变得比较复杂。其最简形式的语法如下。

```
If < expression > Then  
< other code goes here >  
End If
```

< expression >中可以使用任何结果是 True 或 False 的语句(也就是布尔表达式)。可以是一个数学等式。

```
If 2 + 2 = 4 Then  
< other code goes here >  
End If
```

也可以是一个返回 True 或 False 的函数。

```
If IsNumeric(varAny) Then  
< other code goes here >  
End If
```

或者是比较复杂的逻辑算式：

```
If strMagicWord = "Please" And (strName = "Hank" Or strName = "Bill") Then  
< other code goes here >  
End If
```

还可以用 Not 语句对表达式的 True 或 False 结果取反。

```
If Not IsNumeric(varAny) Then  
< other code goes here >  
End If
```

可以用 Else 代码块给 If 结构再加上一重。在 If 表达式为 False 时就会执行 Else 代码块。

```
If IsNumeric(varAny) Then
```

```
< other code goes here >
```

```
Else
```

```
< some other code goes here >
```

```
End If
```

但是很多情况下您要做的判断都不是简单的且/或求值。这种情况下可以添加多个 `ElseIf` 代码块。

```
If IsNumeric(varAny) Then
```

```
< other code goes here >
```

```
ElseIf IsDate(varAny) Then
```

```
< some other code goes here >
```

```
ElseIf IsEmpty(varAny) Then
```

```
< some other code goes here >
```

```
Else
```

```
< some other code goes here >
```

```
End If
```

如果第一个表达式返回 `False`，就会执行一个 `ElseIf` 求值。如果它也返回 `False`，那就会执行第二个 `ElseIf` 求值。如果还是返回 `False`，就会执行 `Else` 代码块中的代码。`ElseIf` 行必须以单词 `Then` 结束，就跟开始的 `If` 行一样。`Else` 代码块通常是可选的，而且必须放在最后面。

```
If IsNumeric(varAny) Then
```

```
< other code goes here >
```

```
ElseIf IsDate(varAny) Then
```

```
< some other code goes here >
```

```
ElseIf IsEmpty(varAny) Then
```

```
< some other code goes here >
```

```
End If
```

`If...End If` 代码块之间还可以相互嵌套。

```

If IsNumeric(varAny) Then
If varAny > 0 Then
    < code goes here >
ElseIf varAny < 0 Then
    < code goes here >
Else
    < code goes here >
End If
Else
    < some other code goes here >
End If

```

可以嵌套得很深，但是要防止出现这种情况，因为这会使代码的逻辑变得难以理解。有研究表明大部分人都很难理解超过三重或四重的嵌套。

要记得 `Select...End Select` 代码块(下一节会介绍)可以用来代替含有很多 `Elseif` 语句的 `If...End If` 代码块。但是 `Elseif` 更灵活，因为每个 `Elseif` 语句都可以对完全不同的东西求值，而 `Select...End Select` 代码块只能考虑到同一个表达式的不同值。由于 `If...Elseif...End If` 更加灵活，所以通常都可以用它代替 `Select...End Select`。但是反过来是不行的。只能用 `Select...End Select` 对同一个表达求值。

下面是一串对不同表达式求值的 `Elseif` 代码块。

```

If boolFirst Then
< other code goes here >
ElseIf boolSecond Then
< some other code goes here >
ElseIf boolThird Then
< some other code goes here >
ElseIf lngTest = 1 Then
< some other code goes here >
ElseIf strName = "Bill" Then

```

```
< some other code goes here >
```

```
End If
```

5.1.2 Select Case 分支

前一节中已经提到，Select...End Select 结构在处理同一个表达式的不同的值时很有用。

Select...End Select 的语法如下。

```
Select Case < expression >
```

```
Case <possibility 1>
```

```
    < code goes here >
```

```
Case <possibility 2>
```

```
    < other code goes here >
```

```
Case <possibility 3>
```

```
    < other code goes here >
```

```
Case <possibility n>
```

```
    < other code goes here >
```

```
Case Else
```

```
    < other code goes here >
```

```
End Select
```

注意这是对同一个表达式多次求值，而 If...Elseif...End If 可以对不同的表达式求值。还要注意在完成所有测试后，还可以有一个可选的 Case Else，如果其它的选项都没有返回 True 时就会执行这个代码块。来看一个更具体的例子。

```
Select Case VarType(varAny)
```

```
Case vbString
```

```
    < code goes here >
```

```
Case vbLong
```

```
    < code goes here >
```

```
Case vbBoolean
```

```
< code goes here >
```

```
Case Else
```

```
< code goes here >
```

```
End Select
```

第一行是计算表达式 `VarType(varAny)` 的值，然后每个 `Case` 语句都会检查一些可能的值。最后，如果所有 `Case` 语句都是 `False`，就会执行 `Case Else` 代码块。您也可以用 `If...Elseif...End If` 完成同样的工作。

```
If VarType(varAny) = vbString Then
```

```
< code goes here >
```

```
ElseIf VarType(varAny) = vbLong Then
```

```
< code goes here >
```

```
ElseIf VarType(varAny) = vbBoolean Then
```

```
< code goes here >
```

```
Else
```

```
< code goes here >
```

```
End If
```

但是，这样的缺点就是每个 `Elseif` 代码块都会执行 `VarType(varAny)` 表达式，然而在 `Select...End Select` 中它只会被执行一次，这样效率更高。有些程序员会说 `Select Case` 代码块比一串 `Elseif` 语句更优雅可读。

随时在 `Select Case` 代码块中包含一个 `Case Else` 代码块是个好主意——即使您不知道什么情况才会执行 `Case Else`。这有两个原因：

- 如果脚本的输入出现异常，`Case Else` 代码块就会开始执行，代码就能捕获这个异常——而没有 `Case Else` 代码块的话就无法捕获它。如果要将您希望所检查的输入值限定在某个范围内的话，这就很有用。用 `Case Else` 代码块就能捕获那些异常的输入数据。
- 包含 `Case Else` 代码块可以为代码添加文档，说明为什么 `Case Else` 代码块不会被执行。要说明为什么 `Else` 条件永远都不会满足，通常是添加一个没有实际内容的 `Case Else` 代码块，而不是添加注释。下面这个例子同时使用了注释和错误消息。

```
Select Case lngColor
```

```
Case vbRed
```

```
    < code goes here >
```

```
Case vbGreen
```

```
    < code goes here >
```

```
Case vbBlue
```

```
    < code goes here >
```

```
Case Else
```

```
    'We never use anything but Red, Green, and Blue
```

```
    MsgBox "Illegal color encountered: " & lngColor, _
```

```
    vbExclamation
```

```
End Select
```

也可以用嵌套的 `Select...End Select` 代码块, 还可以把 `If...End If` 代码块(或是其它类型的代码)嵌在 `Select...End Select` 中。

```
Select Case VarType(varAny)
```

```
Case vbString
```

```
    Select Case varAny
```

```
        Case "Test1"
```

```
            If Trim(strUserName) = "" Then
```

```
                < code goes here >
```

```
            Else
```

```
                < code goes here >
```

```
        End If
```

```
        Case "Test2"
```

```
            < code goes here >
```

```
        Case "Test3"
```

```
            < code goes here >
```

```
    End Select
```

Case vbLong

< code goes here >

Case vbBoolean

< code goes here >

Case Else

< code goes here >

End Select

注意

这里只有两层嵌套，看上去像是四层是因为有两层 **Select Case** 代码块。要控制 **Select Case** 代码块嵌套层数，否则会影响代码的整洁。

5.2 循环结构

跟前面用于决定是否执行某块代码的分支结构相对应，循环可以反复地执行同一段代码。

VBScript 提供了四种用于不同情形的循环结构。但在大多数 Visual Basic 和 VBScript 程序员看来，其中的一种循环结构，**While...Wend** 循环已经被更直观、更强大、更灵活的 **Do...Loop** 代替了。所以，本章会重点介绍其它三种循环。但是，为了保持内容的完整，本章最后还是会简单地介绍 **While...Wend** 循环的语法。

如果不考虑 **While...Wend**(这样做只是为了简便起见，并不是因为其本身有什么错误)，剩下的三种循环结构各自适用于一种类型的循环。接下来的几个小节会介绍这些循环的语法，以及各个循环该在什么时候使用。

5.2.1 For...Next

For...Next 循环适用于两种情况：

- 代码反复执行的次数已知。
- 对诸如数组、文件或数据库表等复杂数据结构中的每个元素执行一段代码(但是 **For Each...Next** 循环是专门为另外一种复杂数据结构，集合(collection)设计的)。

先来看看如何使用 For...Next 循环将一段代码执行若干次(FOR_LOOP_SIMPLE. VBS)。

```
Option Explicit
```

```
Dim lngIndex
```

```
For lngIndex = 1 To 5
```

```
MsgBox "Loop Index: " & lngIndex
```

```
Next
```

成功地执行这段代码会生成如图 5-1 到 5-5 所示的五个对话框。



图 5-1



图 5-2



图 5-3



图 5-4



图 5-5

首先要注意，要使用 For...Next 循环就需要一个循环变量——也就是循环索引(loop index)。lngIndex 变量就是这个功能。语句 For lngIndex = 1 To 5 意思就是这个循环会执行 5 次。您可以从这些对话框中看到，lngIndex 的值逐步地从 1 增加到 5。在完成第 5 次循环后，循环结束，代码继续运行。注意，并不一定要从 1 开始执行 5 次循环(FOR_LOOP_NOT_ONE. VBS)。

```
Option Explicit
```

```
Dim lngIndex  
For lngIndex = 10 To 14  
MsgBox "Loop Index: " & lngIndex  
Next
```

这仍然会执行五次，但是不是从 1 开始，而是从 10 开始。lngIndex 的值随着循环从 10 到 11，到 12，直到 14。

还可以用 Step 关键字跳过某些数字(FOR_LOOP_STEP.VBS)。

```
Option Explicit  
Dim lngIndex  
For lngIndex = 10 To 18 Step 2  
MsgBox "Loop Index: " & lngIndex  
Next
```

这次还是会循环五次，因为您指定了 Step 2，它会跳过其它数字。第一次循环时，lngIndex 的值是 10，然后是 12，14，直到 18。可以用 Step 关键字设定任意您想要的步幅(FOR_LOOP_STEP_100.VBS)。

```
Option Explicit  
Dim lngIndex  
For lngIndex = 100 To 500 Step 100  
MsgBox "Loop Index: " & lngIndex  
Next
```

也可以用 Step 关键字逆向地执行循环(FOR_LOOP_BACKWARDS.VBS)。

```
Option Explicit  
Dim lngIndex  
For lngIndex = 5 To 1 Step -1  
MsgBox "Loop Index: " & lngIndex  
Next
```

由于 Step 关键字后面是负数，循环会逐步地减小这个数。要注意，为了让这个循环正常运行，循环索引的初始值应该比结束值大。

并不一定非得使用带负数的 **Step** 关键字。循环本身也可以根据负数循环，比如(FOR_LOOP_NEGATIVE.VBS):

```
Option Explicit
```

```
Dim lngIndex
```

```
For lngIndex = -10 To -1
```

```
    MsgBox "Loop Index: " & lngIndex
```

```
Next
```

或者是这样(FOR_LOOP_NEGATIVE2.VBS):

```
Option Explicit
```

```
Dim lngIndex
```

```
For lngIndex = -10 To -20 Step -2
```

```
    MsgBox "Loop Index: " & lngIndex
```

```
Next
```

您还可以嵌套使用循环(FOR_LOOP_NESTED.VBS)。

```
Option Explicit
```

```
Dim lngOuter
```

```
Dim lngInner
```

```
For lngOuter = 1 to 5
```

```
    MsgBox "Outer loop index: " & lngOuter
```

```
    For lngInner = 10 to 18 Step 2
```

```
        MsgBox "Inner loop index: " & lngInner
```

```
    Next
```

```
Next
```

如果不知道准确的循环次数该怎么办？这是很常见的情况。比如要遍历数组(见第 3 章)、字符串，或其他诸如此类的结构。看一下这个例子(EXTRACT_FILE_NAME.VBS)。

```
Option Explicit
```

```
Dim lngIndex
```

```
Dim lngStrLen
Dim strFullPath
Dim strFileName

'This code will extract the filename from a path

strFullPath = "C:\Windows\Temp\Test\myfile.txt"
lngStrLen = Len(strFullPath)
For lngIndex = lngStrLen To 1 Step -1
If Mid(strFullPath, lngIndex, 1) = "\"Then

strFileName = Right(strFullPath, _
    lngStrLen - lngIndex)
    Exit For
End If
Next

MsgBox "The filename is: " & strFileName
```

执行这段代码，产生如图 5-6 所示的对话框。



图 5-6

这个例子中增加了一些新元素。**Len()**函数是 VBScript 的内建函数，返回字符串中的字符个数。**Mid()**函数可以从字符串中取出一个或多个字符。第一参数是要处理的字符串；第二个参数是从中提取字符的起始位置；第三个参数是要提取多少个字符。**Right()**函数跟 **Mid()**类似，区别在于它是提取字符串最右边的若干个字符。最后，**Exit** 语句跳出循环。如果您知道不再需要循环时，这很方便。

要注意如何用 **strFullPath** 变量的长度确定循环的次数。一开始，并不知道要循环多少次，

所以用需要遍历的结构(这里是字符串)的长度来指定循环次数。还要注意如何逆向地遍历字符串，这样才能找到 `strFullPath` 变量中的最后一个反斜杠字符("\")。一旦找到反斜杠，就知道了文件名的开始。用 `Right()` 函数将文件名提取到 `strFileName` 变量中，就不再需要循环了(已经达成了您的目的)，所以用 `Exit For` 结束循环。`Exit For` 将执行流程跳到 `Next` 语句的下一行代码。

要指出的是，前面的例子并不是要介绍最有效的从路径中获得文件名的方法。这个例子只是举例说明如何用 `For...Next` 循环遍历一个在设计时大小未知的数据结构。现在来看一下更高效地实现同一个功能的方法，这个例子也告诉我们，紧盯着循环会让我们错过其它更好的方法。下面的代码速度更快，尤其是在处理长文件名时(`EXTRACT _ FILE _ NAME _ NO _ LOOP.VBS`)。

```
Option Explicit
```

```
Dim strFileName
```

```
Dim strFullPath
```

```
strFullPath = "C:\MyStuff\Documents\Personal\resume.doc"
```

```
strFileName = Right(strFullPath, _Len(strFullPath) -  
                    InStrRev(strFullPath, "\"))
```

```
MsgBox "The filename is: " & strFileName
```

同一个问题通常都有很多解决方法。循环很方便，而且是程序设计中所必需的，但是从性能的观点看，它也是代价高昂的。第二个例子比较好的原因有两点：首先，代码少一些；其次，它没有用循环反复地执行一行代码，所以得到结果的速度也更快。

5.2.2 For Each...Next

`For Each...Next` 循环是一种特殊的循环，专门用于遍历集合。集合(collection)，顾名思义，就是数据的集合，很像一个数组。集合中通常含有同一类型的对象(尽管集合也可以存放各种类型的数据)。例如，VBScript 的内建运行时对象 `FileSystemObject`(见第 7 章)是 `Folder` 对象，表示文件系统中的目录。`Folder` 对象有一个 `Files` 集合，表现为一个属性。在 `Folder.Files` 集合中可能会有若干的 `File` 对象。您可以用 `For Each...Next` 循环遍历 `Folder.Files` 集合中

的每个 File 对象。

For Each...Next 循环中不能直接控制循环次数。循环次数取决于您所遍历的集合中对象的数量。但是，您还是可以用 Exit For 语句随时终止循环。可以通过一些检查来确定使用 Exit For 语句的时间，或是用额外的计数变量记录循环的运行次数。

接下来的例子用到了 FileSystemObject 和相关的对象，这会在第 7 章中正式介绍。这个例子(FSO_FIND_FILE.VBS)的目的是查找系统中 AUTOEXEC.BAT 文件的位置 (别担心，这段代码是绝对安全的——不会对您的 AUTOEXEC.BAT 文件造成任何损害)。

Option Explicit

Dim objFSO

Dim objRootFolder

Dim objFileLoop

Dim boolFoundIt

Set objFSO = _

WScript.CreateObject("Scripting.FileSystemObject")

Set objRootFolder = objFSO.GetFolder("C:\")

Set objFSO = Nothing

boolFoundIt = False

For Each objFileLoop In objRootFolder.Files

If UCase(objFileLoop.Name) = "AUTOEXEC.BAT" Then

 boolFoundIt = True

 Exit For

End If

Next

Set objFileLoop = Nothing

Set objRootFolder = Nothing

If boolFoundIt Then

MsgBox "We found your AUTOEXEC.BAT file in "& _

 "the C:\ directory."

```
Else
MsgBox "We could not find AUTOEXEC.BAT in " & _
    "the C:\ directory."
End If
```

不要担心其中您不熟悉的语法。把注意力集中在 **For Each...Next** 循环代码块的语法上。
objRootFolder 变量存放有指向 **Folder** 对象的引用，**Folder** 对象中有一个 **Files** 集合。**Files** 集合中有一些 **File** 对象。所以这段 **VBScript** 代码的意思就是“看看 **Files** 集合中的每个 **File** 对象”。每次执行循环时，循环变量 **objFileLoop** 中都会存放一个不同的 **Files** 集合中的 **File** 对象。如果 **Files** 集合是空的，那循环就不会执行。要注意如何在找到您所寻找的文件时使用 **Exit For** 语句退出循环。

前面这个示例脚本是为了举例说明如何使用 **For Each...Next** 循环遍历集合中的对象。跟前一节中一样，这样使用循环并不是检查某个文件是否存在的最好方法。例如，下面这个就更快，更紧凑(**FSO_FIND_FILE_NO_LOOP.VBS**):

```
Option Explicit
Dim objFSO
Set objFSO = _WScript.CreateObject("Scripting.FileSystemObject")
If objFSO.FileExists("C:\AUTOEXEC.BAT") Then
MsgBox "We found your AUTOEXEC.BAT file in the " & _
    "C:\ directory."
Else
MsgBox "We could not find AUTOEXEC.BAT in " & _
    "the C:\ directory."
End If
Set objFSO = Nothing
```

您可能会觉得我们想表达的意思是不要使用循环，而是有更好的方法。并非如此。循环非常有用，而且很多优秀的脚本都在使用循环。程序设计通常就是使用某种数据，同时这些数据通常存放在诸如数组和集合一类的复杂结构中。如果要在其中找到您所需要的数据，那

么循环就是个不错的帮手。但是，前面也提到了，很多时候循环是最直接的解决方法，但是还有更优雅，更廉价的可选方法。

在学习 Do 循环之前，请注意，尽管 For Each...Next 最常见的是用于遍历集合，但它还可以用于遍历数组中的元素。无论数组中有多少元素，也无论它有多少维度，For Each...Next 循环都能遍历其中的每个元素。这个例子就是用 For Each...Next 循环遍历一个一维数组(FOR_EACH_ARRAY.VBS)。

```
Option Explicit
```

```
Dim astrColors(3)
```

```
Dim strElement
```

```
astrColors(0) = "Red"
```

```
astrColors(1) = "Green"
```

```
astrColors(2) = "Blue"
```

```
astrColors(3) = "Yellow"
```

```
For Each strElement In astrColors
```

```
MsgBox strElement
```

```
Next
```

5.2.3 Do Loop

Do 循环是最通用的循环结构。因为您可以非常方便地用它根据任何您所需的标准来设定循环次数。

至少执行一次代码

Do 循环的威力来自于 While 和 Until 关键字。While 和 Until 既可以用在循环的开始，也可以用在循环的结尾来控制是否要再次执行循环。看一下这个使用 Do 循环的简单例子(DO_WHILE.VBS)。

```
Option Explicit
```

```
Dim boolLoopAgain
```



```

Dim lngLoopCount

Dim strResponse

boolLoopAgain = False

lngLoopCount = 0

Do

boolLoopAgain = False

lngLoopCount = lngLoopCount + 1

strResponse = InputBox("What is the magic word?")

If UCase(Trim(strResponse)) = "PLEASE" Then

    MsgBox "Correct! Congratulations!"

Else

    If lngLoopCount < 5 Then

        MsgBox "Sorry, try again."

        boolLoopAgain = True

    Else

        MsgBox "Okay, the word we wanted was "Please"."

    End If

End If

Loop While boolLoopAgain

```

这样使用 **Do** 循环处理用户的输入是一种常见的技术。要求用户输入一些东西，但是用户可能会输入不合适的数据。这就需要检查输入，如果有必要的话还要返回去要求用户重新输入。

关于这段代码要注意以下几点：

- **Do** 语句表示循环的开始，**Loop** 语句则定义了循环的结束。但 **While** 语句给 **Loop** 语句设定了一个条件。只有在 **While** 语句中的表达式为 **True** 时循环才会再次执行。具体到这个例子，这个表达式就是变量 **boolLoopAgain**，它的子类型是 **Boolean**。这个表达式可以是任何可以计算为 **True** 或 **False** 的表达式。

- 在循环开始时将 `boolLoopAgain` 变量初始化为 `False`。这达到了两个目的：将变量的子类型设为 `Boolean`，这确保如果循环中的代码将这个变量显式地设为 `True`，循环会再次执行。如果用户猜错了，就将 `boolLoopAgain` 设为 `True`，确保循环会再次执行，这样就能让用户再猜一次。
- 用循环计数变量，`lngLoopCounter` 确保循环不会无穷无尽，否则如果用户一直猜不到这个神奇的单词，他会发疯的。并不是一定要用循环计数变量，它也不是 `Do ...Loop` 语法的一部分，但如果循环有可能变得永无止境的话就应该使用计数变量。

使用这种特殊的循环结构——以 `Do` 语句本身开始，将 `While` 条件和 `Loop` 语句放在结尾——其暗含的意思很重要：因为没有在 `Do` 语句中设置条件，所以循环中的代码至少会执行一次。在这个例子中这是您需要的，因为如果您不执行一次代码，问题“`What is the magic word?`”就不可能显示给用户。

检查先决条件

但有时候您只有在某个先决条件为 `True` 的情况下才执行循环内部的代码；如果先决条件为 `False`，那根本就不想执行循环。这种情况下可以将 `While` 语句放在循环的开头。如果 `Do While` 条件是 `False`，那么循环就一次都不会运行。

在接下来的这个例子中，用 `FileSystemObject` 打开一个文本文件。用 `TextStream` 对象访问这个文本文件。在用 `TextStream` 对象打开一个文件时，`TextStream` 对象会用一个“指针”记录您读取的文件位置。在刚打开文件时，这个指针指向文件的开始(这个指针并不是真正的放在文件中——它只存在于内存中的 `TextStream` 对象中)可以用 `TextStream.ReadLine` 方法逐行地读取文件。

每次调用 `ReadLine`，指针就会移到文件的下一行。当指针越过文件的最后一行时，`TextStream.AtEndOfStream` 的值就会是 `True`。此时您就能知道已经读完这个文件了。但这中间可能存在问题：在打开文件时，并不确定其中是否有数据。它有可能是空的。如果真是这样，就无需调用 `ReadLine`，因为这会产生错误。但如果在打开文件时 `AtEndOfStream` 属性为 `True`，那您就知道这个文件是空的。这样您就可以放心地在 `Do` 循环中调用 `ReadLine`。

若您想尝试这段代码，那就得创建一个文本文件，输入下面这几行内容(即下载的代码中所包含的 TESTFILE.TXT 文件):

Line 1

Line 2

Line 3

Line 4

Line 5

将这个文件跟下面的这个脚本(DO_WHILE_READ_FILE.VBS)一起保存到硬盘的同一位置。这个脚本假设 TESTFILE.TXT 跟它位于同一目录下。在阅读代码时，如果不熟悉 FileSystemObject 和 TextStream 对象的话也不要担心，会在第 7 章中介绍它们。现在只要关心如何在 Do 语句中使用 While 条件。

```
Option Explicit
```

```
Dim objFSO
```

```
Dim objStream
```

```
Dim strText
```

```
Set objFSO = _
```

```
WScript.CreateObject("Scripting.FileSystemObject")
```

```
Set objStream = objFSO.OpenTextFile("testfile.txt")
```

```
Set objFSO = Nothing
```

```
strText = ""
```

```
Do While Not objStream.AtEndOfStream
```

```
strText = strText & objStream.ReadLine & vbNewLine
```

```
Loop
```

```
Set objStream = Nothing
```

```
If strText <> "" Then
```

```
MsgBox strText
```

```
Else
```

```
MsgBox "The file is empty."
```

End If

这段代码的运行结果就是如图 5-7 所示的对话框。



图 5-7

可以看到，通过将 **While** 条件放到循环的开头，可以判断是否要执行一次循环。如果文件是空的，就一行都不会读取。因为在 **Loop** 语句中没有条件限制，所以在到达循环结尾时代码就会跳回到 **Do** 那一行。但是，如果 **Do While** 表达式返回 **False**，循环就不再执行，代码立即跳转到 **Loop** 语句的下一行。

只有在到达文件末尾时，`objStream.AtEndOfStream` 属性才会为 **True**。只要没有到达文件的末尾，就会一直循环。如果因为文件是空的而一开始就在文件的末尾，就完全不必执行循环。

为了加深记忆，回到第一个 **Do** 循环的例子，要知道可以把 **While** 语句跟 **Do** 放在一起实现同样的功能(`DO_WHILE2.VBS`)。

Option Explicit

```
Dim boolLoopAgain
```

```
Dim lngLoopCount
```

```
Dim strResponse
```

```
boolLoopAgain = True
```

```
lngLoopCount = 0
```

```
Do While boolLoopAgain
```

```
boolLoopAgain = False
```

```
lngLoopCount = lngLoopCount + 1
```

```
strResponse = InputBox("What is the magic word?")
```

```
If UCase(Trim(strResponse)) = "PLEASE" Then
    MsgBox "Correct! Congratulations!"
Else
    If lngLoopCount < 5 Then
        MsgBox "Sorry, try again."
        boolLoopAgain = True
    Else
        MsgBox "Okay, the word we wanted was 'Please'."
    End If
End If
Loop
```

将这段代码与第一个 **Do** 循环的例子比较一下。两个例子所完成的功能是一样的：循环至少执行一次，并且只有在循环内部的代码允许时才会再次循环。差别在于这第二个例子中的技巧是将 `boolLoopAgain` 初始化为 `True`，这样就能确保循环至少会执行一次。

在 **Until** 和 **While** 之间做出选择

如您所见，**Do** 循环非常强大，用什么方法来实现功能很大程度上取决于您个人的偏好。也就是说，有人会认为这段代码的第一个版本更好，因为 **Do** 语句本身就明显地说明这个循环至少会执行一次，而第二个例子则有些取巧的成分。

其他的事情也是这样，任何事物都有两面性，意义明确的方法通常比较受欢迎。

所以在考虑使用 **Do** 循环时，首先要问自己的问题就是，是否要求这段代码至少执行一次？如果这个问题的回答是肯定的，就最好把条件放在循环的末尾。否则，就把条件放在循环的开头。

但这还有第二个问题：是否应该使用 **While** 语句或是它的兄弟 **Until** 语句？答案在很大程度上取决于您的个人偏好。尽管 **While** 和 **Until** 语句有些不同，但是它们的功能是一样的。主要的差别在于语义，人们通常只是凭语言的直觉从中做出选择。但在给定的情况下，其中总有一个的意义比另一个清晰些。

下面是微软的 VBScript 文档中对 Do 循环的描述(加粗表示强调):

当 **while** 某一条件为 **True** 时, 反复执行一段语句, 或者反复执行一段语句, 直到(until)某一条件为 **True**。

如您所见, **While** 和 **Until** 之间的区别非常微妙。说明其中区别的最好方法就是用 **Until** 代替前面两个例子中的 **While**。您会看到关于是否至少执行一次的考虑还是保持不变。但是, 具体的实现有点变化。这是第一个例子, 用 **Until** 代替了 **While(DO_UNTIL.VBS)**。

Option Explicit

Dim boolLoopAgain

Dim lngLoopCount

Dim strResponse

boolLoopAgain = False

lngLoopCount = 0

Do

boolLoopAgain = False

lngLoopCount = lngLoopCount + 1

strResponse = InputBox("What is the magic word?")

If UCase(Trim(strResponse)) = "PLEASE" Then

MsgBox "Correct! Congratulations!"

Else

If lngLoopCount < 5 Then

MsgBox "Sorry, try again."

boolLoopAgain = True

Else

MsgBox "Okay, the word we wanted was 'Please'."

End If

End If

Loop Until boolLoopAgain = False

这看上去似乎是完全一样的代码, 其中的区别就在于必须在 **Until** 语句中检查变量是否

是 False，而不是像在 While 语句中那样检查是否为 True。在阅读 Loop While boolLoop Again 这行代码时，是不是觉得它比 Loop Until boolLoopAgain = False 更好理解？如果 While 语法对您来说更好理解，可能您可以通过修改变量名称来改变这个情况(DO_UNTIL_BETTER_NAME.VBS)。

```
Option Explicit
```

```
Dim boolStopLooping
```

```
Dim lngLoopCount
```

```
Dim strResponse
```

```
boolStopLooping = True
```

```
lngLoopCount = 0
```

```
Do
```

```
    boolStopLooping = True
```

```
    lngLoopCount = lngLoopCount + 1
```

```
    strResponse = InputBox("What is the magic word?")
```

```
    If UCase(Trim(strResponse)) = "PLEASE" Then
```

```
        MsgBox "Correct! Congratulations!"
```

```
    Else
```

```
        If lngLoopCount < 5 Then
```

```
            MsgBox "Sorry, try again."
```

```
            boolStopLooping = False
```

```
        Else
```

```
            MsgBox "Okay, the word we wanted was 'Please'."
```

```
        End If
```

```
    End If
```

```
Loop Until boolStopLooping = True
```

这样是不是 Until 语法更好理解？这说明无论是 While 还是 Until 都能实现您的需求——问题只是在给定的情况下哪个更有意义。再看一下读取文件的例子，这次用 Until 代替了 While(DO_UNTIL_READ_FILE.VBS)。

```

Option Explicit

Dim objFSO
Dim objStream
Dim strText

Set objFSO = _WScript.CreateObject("Scripting.FileSystemObject")

Set objStream = objFSO.OpenTextFile("testfile.txt")

Set objFSO = Nothing

strText = ""

Do Until objStream.AtEndOfStream

strText = strText & objStream.ReadLine & vbNewLine

Loop

Set objStream = Nothing

If strText <> "" Then

MsgBox strText

Else

MsgBox "The file is empty."

End If

```

Until 语法要显得清楚些。人们总是觉得从正面考虑问题比较容易，您或许觉得 `Do While Not objStream.AtEndOfStream` 更好理解，也有可能觉得 `Do Until objStream.AtEndOfStream` 更简单易懂。这完全取决于您自己。VBScript 并不关心这个。如果使用了好的变量名，坚持直观的逻辑思路，合理地使用缩进和空白；跟您合作的程序员应该也不会在意这个问题。

跳出 **Do** 循环

在学习 `While...Wend` 之前，我们要说一下 `Exit Do` 语句。跟 `Exit For` 类似，可以用 `Exit Do` 随时跳出 `Do` 循环。在循环中可以有很多 `Exit Do` 语句。下面这个例子是“神奇单词”例子的又一个变体。

```

Option Explicit

```



```

Dim boolLoopAgain
Dim lngLoopCount
Dim strResponse

boolLoopAgain = False

lngLoopCount = 0

Do

boolLoopAgain = False

lngLoopCount = lngLoopCount + 1

strResponse = InputBox("What is the magic word?")

If UCase(Trim(strResponse)) = "PLEASE" Then

    MsgBox "Correct! Congratulations!"

    Exit Do

Else

    If lngLoopCount < 5 Then

        MsgBox "Sorry, try again."

        boolLoopAgain = True

    Else

        MsgBox "Okay, the word we wanted was 'Please'."

    Exit Do

    End If

End If

Loop While boolLoopAgain

```

这里不是将 `boolLoopAgain` 设为 `False`，而是执行 `Exit Do`，效果是一样的，也就是不再执行循环。在执行 `Exit Do` 语句时，代码会跳出循环，跳到循环代码块的下一行(在这个例子中，循环后面没有代码，所以脚本结束)。但是，这个例子只是举例说明如何正确地使用 `Exit Do`，并不是要用它来改善“神奇单词”的代码。

还记得在第 4 章中关于过程和函数的讨论吗？在讨论 `Exit Sub` 和 `Exit Function` 语句时，我们说要小心的使用它们，而且通常都能找到办法不使用这两个语句。潜在的问题是使用 `E`

xit Sub 和 ExitFunction 会使得代码的执行流程产生跳跃，其逻辑结构也难以理解。这些原则对 Exit Do 也适用。

如果将最初的“神奇单词”代码与这个新版本做比较，就会发现最初是用 boolLoopAgain 语句配合 If 条件来控制循环。这种逻辑流程是从上到下，是线性的。而新代码中的 Exit Do 语句使得这段代码丧失了这种优雅和清晰。

最后要注意的是 Exit Do 语句的使用(其他循环中是 Exit 语句)。如果使用嵌套循环，执行内部循环中的 Exit Do 语句并不中断外部循环，只有执行相应循环中的 Exit Do 语句才能中断该循环。

5.2.4 While...Wend

正如在本章开头所说的，While...Wend 循环是一种来自老版本的 BASIC 和 Visual Basic 的老式语法。相对于不太通用的 While...Wend 循环，程序员通常更喜欢用 Do 循环(见前几节)。这并不是说就绝对不能使用 While...Wend 循环，还是有不少程序员在使用它。它能运行，简单，而且没有任何迹象表明微软会停止支持它。它只是有些跟不上时代。为了保持内容的完整，下面是一个 While...Wend 语法的例子(WHILE_WEND.VBS)。

```
Option Explicit
```

```
Dim lngCounter
```

```
lngCounter = 0
```

```
While lngCounter <= 20
```

```
    lngCounter = lngCounter + 1
```

```
    '< other code goes here >
```

```
Wend
```

跟 Do 循环不一样，这里不需要使用 While 或 Until，也不能将条件放在循环的结尾。如您所见，判断是否继续循环的条件只能放在循环的开头。最后，While...Wend 最主要的限制就是它没有等价于 Exit For 或 Exit Do 的语句，这意味着不能强行跳出循环。

5.3 小结

本章探讨了流程控制，涉及到分支和循环。分支技术就是检查条件，做出判断，然后根据判断有选择的执行代码。下面是 VBScript 中的分支结构：

If ...ElseIf ...Else ...End If

Select Case ...End Select

循环就是反复地执行同一块代码。VBScript 中的循环结构如下：

For ...Next

For Each ...Next

Do ...Loop While

Do While ...Loop

Do ...Loop Until

Do Until ...Loop

While ...Wend

第 8 章 VBScript 中的类(编写 COM 对象)

尽管这个功能已经存在了一段时间，但大部分人还是不知道可以在 VBScript 中定义类。VBScript 中的类为 VBScript 程序员提供了一个强大的工具，尤其是在规模较大的脚本项目中。诚然，VBScript 的面向对象能力比不上 Java、C++、VB.NET，甚至不如 Visual Basic 6，但这确实可以让 VBScript 程序员在编写程序时利用一些面向对象的优点。

如果您跳过了前面的几章内容，并且不熟悉如何使用 VBScript 中的 COM 对象，那么最好返回去阅读一下第 7 章的第 1 节。本章假设您已经熟悉了实例化对象和调用对象的属性和方法的基本知识。

8.1 对象、类和组件

在开始编写和使用您自己的 VBScript 类之前，本节会先介绍一些术语。近年来类(class)、对象(object)、组件(component)这些技术术语被误解、混淆了。尽管它们有着不同的含义，但这些术语经常被视为可以互换的。这使得纯粹的面向对象主义者有些抓狂，而且这还加大了新手的学习难度。这里先明确一下这些术语的含义。

严格地说，对象是复杂数据和程序结构在内存中的表现，只有在程序运行时才存在。一个合适的比喻就是数组，这也是一个只有在运行时才存在的复杂数据结构。在某段代码中使用数组时，大部分人都知道这是指的内存中的数据结构。不过，在程序员使用“对象”这个词时，并不一定是指其严格的定义，即运行时存在于内存中的数据结构。

对象与数组还是有些区别，最重要的就是对象并不只是像数组那样存放一些复杂数据(以属性的形式)；对象还有“行为”(也就是说“它知道该怎么做”)这表现为方法。属性可以存放任何类型的数据，而方法则可以是过程或函数。将数据和行为一起放入对象中，这样就可以在设计程序时将被操作的数据和操作数据的代码放在一起。

类是对象的模板。对象只有在运行时才会存在于内存中，而类则是在设计时就能直接使用的程序设计结构。类是代码，而对象是在程序运行时对这段代码的使用。如果要在运行时使用对象，必须先在设计时定义一个类。在运行时会根据类所提供的模板创建对象。(这里只是用不同的方式表达同一个意思。)例如，可以编写一个名为 **Customer** 的类。保存这个类

定义之后，就可以再用其他代码在内存中创建任意数量的 **Customer** 对象。图 8-1 演示了这个概念。

图 8-1

但是很多人都将类等同于对象，比如“我编写了 **Customer** 对象，然后创建了一千个 **Customer** 对象并根据他们的消费额排序”。前面说过，这会给新手造成混淆，但是随着经验的增长，您将学会如何根据上下文来理解它的真实含义。

组件只是一种打包机制，一种将一个或多个相关类编译成一个可以部署到一台或多台计算机上的二进制文件的方法。在图 8-1 中可以看到组件中用于创建对象的类。在 **Windows** 操作系统中，组件通常都是一个 **.DLL** 或 **.OCX** 文件。第 7 章中介绍的脚本运行时库就是组件的一个好例子。在程序员用其他方式编写相互关联的类时，如果他想让其他人也能在运行时使用这些类创建对象就应该将其打包，并将这些类以组件的形式发布。一个程序或是一个脚本有可能会用到很多不同组件中的类。

但是，组件并不是使用类的唯一方式。图 8-1 只展示了一种可能的情况(虽然是一种很常见的情况)。例如，在 **Visual Basic** 程序中可以将类与程序本身编译在一起，而不将其暴露给外部。这些类只存在于程序内部，它们的目的是服务于这个一个程序。在这种情况下，就无需考虑将这些类打包成组件，因为它们不会被别人使用。

人们发现将类打包成独立于程序的组件更有效率。在稍后的几个类上这个想法会得到验证，并且将它们打成一个可移植的组件能使这些类更容易重用。在 **VBScript** 中，这两种技术都可以使用：可以在脚本内创建类，只有那个脚本能使用这个类(本章会对此作介绍)；也可以将您的类打包成 **Windows** 脚本组件(**Script Component**，详见第 16 章)。

8.2 类语句

创建 **VBScript** 类的关键字是 **Class** 语句。跟用 **Function ... End Function** 或 **Sub ... End Sub** 语句设定过程的边界类似，**Class** 和 **End Class** 语句也适用于设定类的边界。可以在一个脚本中使用多个 **Class ... End Class** 代码块定义多个类(但是在 **VBScript** 中类不能嵌套)。

如果您是从其他语言转向 **VBScript**，可能习惯于将类定义在单独的文件中。但是这对 **VBScript** 的类并不适用。通常，您必须将定义 **VBScript** 类的代码与创建类实例的代码放在一起。

这看上去是一个很大的限制(因为创建类的一个目的就是方便代码的移植和重用)但是这里有另外两种实现这个方法。

- 将一个或若干个 VBScript 类打包成 Windows 组件，在第 16 章中会对此做深入探讨。
- 使用 ASP 的 # INCLUDE 指示符，将类的定义存放到一个文件中，并在多个 ASP 页面中包含它。第 20 章会探讨 ASP 脚本中的 VBScript 类。

但是在本章中我们只会探讨与使用类的代码定义在用一个脚本中的类。

除了是否在同一个脚本文件中的区别，Visual Basic 程序员可以毫无障碍地适应 VBScript 类的使用。除了 Visual Basic 和 VBScript 两者在语言上的差别，VBScript 类的结构和相关技术与 Visual Basic 都是一样的。下面是 Class 的基本语法。

```
Class MyClass
```

```
< rest of the class code will go here >
```

```
End Class
```

当然，应该用您所定义的类的名称替换其中的 **MyClass**。这个类的名称以及相同作用域中用 INCLUDE 指示符包含进来的类名称，在脚本中都应该都是独一无二的。类名称还不能与 VBScript 的保留字(比如 Function 或 While)重复。

8.3 定义属性

在脚本根据类创建对象时，属性是存储和访问数据的机制。通过属性，数据可以存放在对象中；也可以通过属性从对象中获取数据。

8.3.1 私有属性变量

存储属性的值的最好方式就是私有属性变量。这是定义为类级作用域的变量(在类的开头)。这个变量是私有的(也就是说不能直接从类的外部访问这个变量)，存放有属性的值。使用类的代码通过 Property Let、Set 和 Get 过程与其进行交互，但是这些过程只是这个私有属性变量的门户而已。

可以这样定义一个私有属性变量：

```
Class Customer
```

```
    Private mstrName
```

```
    < rest of the class code will go here >
```

```
End Class
```

对于类作用域的私有变量，必须用 **Private** 语句声明。**m** 前缀是匈牙利命名法中表示模块级作用域的，也就是类级。有些地方会用 **c** 前缀(比如 **cstrName**)表示类级作用域。但是我们并不建议使用这种方法，因为 **Visual Basic** 程序员经常用这个前缀表示 **Currency** 数据类型，容易弄混。

关于匈牙利命名法详见第 3 章。

8.3.2 Property Let

Property Let 过程是一种特殊的过程，用于在类的外部给私有属性变量赋值。**Property Let** 过程类似于 **VBScript** 中没有返回值的子过程。下面是它的语法。

```
Class Customer
```

```
    Private mstrName
```

```
    Public Property Let CustomerName(strName)
```

```
        mstrName = strName
```

```
    End Property
```

```
End Class
```

注意这里没有使用 **Sub** 或 **Function** 语句来定义这个过程，而是使用的 **Property Let**。**Property Let** 过程必须接收至少一个参数。没有这个参数就无法实现 **Property Let** 过程的目的，也就是使得外部代码可以将值存放到私有属性变量中。这里要注意属性过程中的代码是如何将传递给该过程的 **strName** 值保存到 **mstrName** 私有属性变量中的。这个过程中可以没有任何代码，这也就不会将传递过来的值存放到某个变量或对象中，但是这完全不符合 **Property Let** 过程的目的。

反之，也可以随意地往这个过程中添加代码。在某些情况下，可能要在将传递过来的值真正赋给私有属性变量之前先对其做一些检查。例如，如果客户姓名的长度不能超过 50 个

字符，就需要检查 `strName` 参数是否超过了 50 个字符的长度；如果超过了这个长度，就用 `Err.Raise()` 方法(详见第 6 章)将这个错误告诉调用代码。

最后，属性过程必须以 `End Property` 语句结束(就像 `Function` 过程必须以 `End Function` 结束，`Sub` 过程必须以 `End Sub` 结束一样)。如果要跳出属性过程，可以使用 `Exit Property` 语句(跟用 `Exit Function` 跳出 `Function`，用 `Exit Sub` 跳出 `Sub` 一样)。

8.3.3 Property Get

`Property Get` 过程跟 `Property Let` 过程正好相反。`Property Let` 过程是用于在类外部的代码中给私有属性变量赋值，而 `Property Get` 过程则是允许类外部的代码读取私有属性变量的值。`Property Get` 过程跟 VBScript 的 `Function` 过程类似，都有返回值。这里是它的语法。

```
Class Customer

Private mstrName

Public Property Let CustomerName(strName)

    mstrName = strName

End Property

Public Property Get CustomerName()

    CustomerName = mstrName

End Property

End Class
```

与 VBScript 的 `Function` 过程类似，`Property Get` 过程会给调用它的代码返回一个值。这个值通常是私有属性变量的值。注意 `Property Get` 过程的名称与相应的 `Property Let` 过程的名称是一样的。`Property Let` 过程将值存放到私有属性变量中，而 `Property Get` 过程则将其读取出来。

`Property Get` 过程不需要任何参数。VBScript 允许加入参数，但是如果这么做，就必须给这个属性的 `Property Let` 或 `Property Set` 过程(如果使用了这个)也添加一个额外的参数。因为 `Property Let/Set` 过程的参数必须比相应的 `Property Get` 过程的参数多一个。

给 `Property Let/Set` 过程添加一个额外的参数是很难看的，并且要求使用这个类的代码

在 Property Let 过程中使用多于一个参数也是很不好的形式。如果觉得某个 Property Get 过程确实需要一个参数，那最好是添加一个额外的属性来实现这个 Property Get 参数的功能。

如果 Property Get 过程返回的是一个指向对象变量的引用，就可能需要用 Set 语句返回这个值。例如：

```
Class FileHelper
'Private FileSystemObject object
Private mobjFSO
Public Property Get FSO()
    Set FSO = mobjFSO
End Property
End Class
```

但是，因为所有的 VBScript 变量都是 Variant 变量，所以 Set 语法并不是不可或缺的。下面这种语法也能运行。

```
Class FileHelper
'Private FileSystemObject object
Private mobjFSO
Public Property Get FSO()
    FSO = mobjFSO
End Property
End Class
```

但是还是应该使用 Set 语法，因为这样可以明确地说明这个 Property Get 过程返回的是一个指向对象变量的引用。

8.3.4 Property Set

Property Set 过程与 Property Let 过程很类似，但是 Property Set 过程是针对基于对象的属性。当需要在属性中存放对象(而不是数字、日期、布尔或字符串子类型的变量)，就可以提供一个 Property Set 过程取代 Property Let 过程。下面是一个 Property Set 过程的语法。

```

Class FileHelper
'Private FileSystemObject object

Private mobjFSO

Public Property Set FSO(objFSO)

    Set mobjFSO = objFSO

End Property

End Class

```

从功能上来说，Property Let 和 Property Set 过程的功能是一样的。但是 Property Set 过程有两个不同之处：

- 明确地说明这个属性是一个基于对象的属性(任何使得代码意义更明确的技术都比其他功能相同的技术更受偏爱)。
- 类外部的代码必须用 `Set Object.Property = Object` 才能写入这个属性(因为这是实现该功能的典型方法，所以这也是优点)。

例如，下面的代码根据前面介绍的类创建了对象。

```

Dim objFileHelper

Dim objFSO

Set objFSO = _WScript.CreateObject("Scripting.FileSystemObject")

Set objFileHelper = New FileHelper

Set objFileHelper.FSO = objFSO

```

注意最后一行代码在给 FSO 属性赋值时使用了 Set 语句。这是必须的，因为 FileHelper 类是用 Property Set 过程给 FSO 属性赋值的。如果没有最后一行开头的 Set 语句，VBScript 就会报错。若类中的属性是基于对象的，通常就应该用 Property Set 过程。大部分使用类的程序员也都希望如此。

也就是说，因为所有的 VBScript 变量都是 Variant 变量，所以也可以用 Property Let 过程。但是如果用 Property Let 过程代替 Property Set 过程，使用这个类的代码就不能用 Set 语句设置这个属性(如果这么做 VBScript 就会报错)，并且这会导致习惯于使用 Set 语法的程序员感到困惑。如果确实有必要支持这两种情况，可以对同一个属性同时提供 Property Let 和 Property Set 两个过程，如下：

```

Class FileHelper
'Private FileSystemObject object
Private mobjFSO
Public Property Set FSO(objFSO)
    Set mobjFSO = objFSO
End Property
Public Property Let FSO(objFSO)
    Set mobjFSO = objFSO
End Property
End Class

```

Property Set 和 Property Let 中的 Set 是可选的。因为是直接写入 Variant 私有属性变量中，所以可以用 Set 也可以不用。这个例子的功能跟前一个例子是相同的。

```

Class FileHelper
'Private FileSystemObject object
Private mobjFSO
Public Property Set FSO(objFSO)
    mobjFSO = objFSO
End Property
Public Property Let FSO(objFSO)
    mobjFSO = objFSO
End Property
End Class

```

8.3.5 创建只读属性

有两种方法可以创建类的只读属性：

- 只为这个属性提供 Property Get 过程。
- 将 Property Get 过程声明为公共的，而将 Property Let 过程声明为私有的。

下面是第一种方法：

```
Class Customer
Private mstrName
Public Property Get CustomerName()
    CustomerName = mstrName
End Property
End Class
```

要注意这里没有 `Property Let` 过程。由于没有提供 `Property Let` 过程，类外部的代码就无法修改 `CustomerName` 属性。

下面是第二种方法。

```
Class Customer
Private mstrName
Private Property Let CustomerName(strName)
    mstrName = strName
End Property
Public Property Get CustomerName()
    CustomerName = mstrName
End Property
End Class
```

用 `Public` 语句声明了 `Property Get` 过程，而用 `Private` 语句声明 `Property Let` 过程。因为 `Property Let` 过程被声明为 `Private`，所以就能有效地对类外部的代码隐藏它。类内部的代码仍然可以通过 `Property Let` 过程修改这个属性，但在这个简单的例子中，因为类内部的代码可以直接修改这个私有属性变量，所以这没有什么实际意义，并不需要私有的 `Property Let` 过程。

也有例外情况，当 `Property Let` 过程中需要有代码对存入这个属性的值进行检查或转换时就是个例外。在这种情况下，类内部的代码也最好是用 `Property Let` 过程，而不是直接修改这个私有属性变量。

第一种创建只读属性的方法(只提供一个 `Property Get`)更加常见。

8.3.6 创建只写属性

有两种方法可以创建只写属性，这两种方法与创建只读属性的方法完全相反(见前一节):

- 忽略 Property Get 过程，只提供 Property Let 过程。
- 用 Public 语句声明 Property Let 过程，用 Private 语句声明 Property Get 过程。

8.3.7 没有属性过程的公共属性

还可以为您的类提供没有 Property Let、Get 和 Set 的属性。这是通过使用公共类级变量实现的。看一个例子，如下：

```
Class Customer  
  
Private mstrName  
  
Public Property Let CustomerName(strName)  
  
    mstrName = strName  
  
End Property  
  
Public Property Get CustomerName()  
  
    CustomerName = mstrName  
  
End Property  
  
End Class
```

其功能与下面这段代码等价：

```
Class Customer  
  
Public Name  
  
End Class
```

第二种方法看上去更有吸引力。从功能和语法的角度上看，它的代码更少，而且是完全符合规定的。但是很多 VBScript 程序员还是偏爱于使用私有属性变量与 Property Let、Set 及 Get 过程的组合，前一节已经对此做了探讨。

而有些程序员喜欢用公共类级变量代替 Property Let、Set 及 Get 过程。主要的优点就是使用公共类级变量创建类属性所需的代码更少。但是，要考虑到不使用 Property Let、Set

及 **Get** 也会带来一些缺点。

除非要使用您的类的代码使用了非常难看的语法，比如 `objCustomer.mstrName = "ACME Inc."`，否则无法用匈牙利命名法的作用域和子类型前缀命名类级变量。若您觉得匈牙利命名法对您的代码是有意义的，那么这会降低代码的可读性和可理解性。

- 无法用前一节中描述的方法创建只读属性或只写属性。
- 类外部的代码可以随时的修改任何属性。如果在某些情况允许修改某些属性，而在其他情况不能修改某些属性，唯一的办法就是用 **Property Let** 过程中的代码检查当前的状况。您不可能知道类外部的代码会在什么时候修改属性的值。
- 没有 **Property Let** 过程，就无法编写代码检查和转换写入属性的值。
- 没有 **Property Get** 过程，就无法编写代码验证和转换从属性中读出的值。

也就是说，如果您能忍受这些缺陷，当然就可以将属性声明为公共类级变量并在需要的时候再修改它们，加上 **Property Let**、**Set**、**Get** 过程。但是，有人会说应该一开始就用正确的方法。关于这个问题，优秀的程序员也有不同的看法，但是您会发现大多数程序员喜欢用 **Property Let**、**Get**、**Set** 过程，而不是公共类级变量。

但是您经常需要在单个脚本中创建简单的类。这种情况下使用一些快捷方式使代码简化和方便编写是可以被接受的。在这种情况下，您可以放弃 **Property Let**、**Get**、**Set**，而直接使用公共变量。

8.4 定义方法

方法(method)实际上就是函数和过程的另一个名称，是本书中一直在探讨的内容。当函数或过程成为类的一部分时，就可以将其称为方法。如果您知道如何编写 **Function** 和 **Sub** 过程(见第 4 章)，那您就知道如何为类编写方法。方法并不像属性那样有一些特别的语法。需要考虑的问题主要是在类中将 **Function** 或 **Sub** 声明为 **Public** 还是 **Private**。

简单地说，类内外的代码都可以使用 **Public** 语句声明的类方法；而只有类内部的代码才能使用 **Private** 语句声明的方法。

这个示例脚本 `SHOW_GREETING.VBS` 中含有一个 **Greeting** 类，用不同类型的消息向用户问好。这个类同时使用了公共方法和私有方法。从 **Greeting** 类的代码可以看出，在类中定义

方法的语法跟定义 VBScript 函数和过程的语法是一样的。唯一需要额外考虑的就是应该声明为公共的还是私有的。

```
Class Greeting
```

```
Private mstrName
```

```
Public Property Let Name(strName)
```

```
    mstrName = strName
```

```
End Property
```

```
Public Sub ShowGreeting(strType)
```

```
    MsgBox MakeGreeting(strType) & mstrName & "."
```

```
End Sub
```

```
Private Function MakeGreeting(strType)
```

```
    Select Case strType
```

```
    Case "Formal"
```

```
        MakeGreeting = "Greetings, "
```

```
    Case "Informal"
```

```
        MakeGreeting = "Hello there, "
```

```
    Case "Casual"
```

```
        MakeGreeting = "Hey, "
```

```
    End Select
```

```
End Function
```

```
End Class
```

类外部的代码可以调用 `ShowGreeting()` 方法，这是公共的；但是不能调用 `MakeGreeting()` 方法，因为这是私有的，只能在内部使用。`SHOW_GREETING.VBS` 示例脚本开头的代码使用了这个类。

```
Dim objGreet
```

```
Set objGreet = New Greeting
```

```
With objGreet
```

```
    .Name = "Dan"
```

```
.ShowGreeting "Informal"
```

```
.ShowGreeting "Formal"
```

```
.ShowGreeting "Casual"
```

```
End With
```

```
Set objGreet = Nothing
```

这个脚本的运行结果如图 8-2 至图 8-4 中的对话框所示。



图 8-2



图 8-3



图 8-4

Visual Basic 程序员要注意：VBScript 不支持用 `Friend` 关键字定义属性和方法。

8.5 类事件

事件(event)是一种会自动被调用的特殊方法。在任何特定的环境中，您所使用的类都支持一个或若干个事件。当给定的环境中支持某个事件时，您可以编写一个事件处理器(event handler)，这实际上就是在事件发生时会自动调用的特殊方法。比如，在浏览器中，当用户单击按钮时，页面中的 VBScript 代码可以对这个按钮的 `OnClick` 事件做出响应。

任何 VBScript 类都自动地支持两个事件：`Class_Initialize` 和 `Class_Terminate`。可以选择在您的类中为它们提供事件处理器。如果在您的类中含有事件处理器方法，那么它们就会自动

地被调用；如果没有，当事件发生时就不会有任何反应——如果您觉得没必要提供处理器方法的话，这不会带来任何问题。

8.5.1 Class_Initialize 事件

当有代码根据类实例化出一个对象时就会引发类中的 `Class_Initialize` 事件。只要是基于某个类实例化出一个对象，就会发生这个事件，但是类中是否含有对此做出响应的代码就取决于您自己了。如果不想响应这个事件，完全可以忽略这个事件的事件处理器。下面这个例子中的类含有一个 `Class_Initialize` 事件处理器。

Class FileHelper

Private FileSystemObject object

Private mobjFSO

Private Sub Class_Initialize

Set mobjFSO = _

WScript.CreateObject("Scripting.FileSystemObject")

End Sub

'<<rest of the class goes here>>

End Class

正如这个例子所演示的，用 `Class_Initialize` 处理器初始化类级变量是一种相当典型的应用。如果要确保在类第一次启动时某个变量是某个特定的值，可以在 `Class_Initialize` 事件处理器中设置它的初始值。还可以用 `Class_Initialize` 事件实现其他的准备工作，比如打开数据库连接或打开文件。

`Class_Initialize` 事件处理器的开始和结束语法必须与这个例子一样。您的代码只能在文件处理器内部做所需的处理，但是不能改变这个过程的名称。处理器的第一行必须是 `Private Sub Class_Initialize`，并且最后一行必须是 `End Sub`。实际上，事件处理器就是一个普通的 VBScript 子过程，只是有着一个特殊的名称。

从技术上来说，您可以用 `Public` 语句(而不用 `Private`)声明事件处理器，但是事件处理器通常都是私有的。如果将其声明为公共的，类外部的代码就可以像调用其他方法一样调用它，

这通常都不是我们所希望的。

在一个类中只能有一个 `Class_Initialize` 事件处理器。如果不需要的话也可以一个都没有，但绝对不能多于一个。

8.5.2 `Class_Terminate` 事件

`Class_Terminate` 事件与 `Class_Initialize` 事件(见前一节)相反。当类被实例化为对象时会触发 `Class_Initialize` 事件，而当基于这个类的对象被销毁时则会触发 `Class_Terminate` 事件。有两种销毁对象的方法：

- 将特殊值 `Nothing` 赋给最后一个引用了该对象的对象变量
- 超出了最后一个引用该对象的对象变量的作用域

发生这两种情况中的任一种时，`Class_Terminate` 事件会在对象真正被销毁之前发生。(关于对象生存期和引用的说明详见第 7 章。)

下面是前一节例子中见过的 `FileHelper` 类，现在添加了一个 `Class_Terminate` 事件处理器。

```
Class FileHelper
```

```
'Private FileSystemObject object
```

```
Private mobjFSO
```

```
Private Sub Class_Initialize
```

```
    Set mobjFSO = _
```

```
    WScript.CreateObject("Scripting.FileSystemObject")
```

```
End Sub
```

```
Private Sub Class_Terminate
```

```
    Set mobjFSO = Nothing
```

```
End Sub
```

```
'<rest of the class goes here>
```

```
End Class
```

在这个例子中，用 `Class_Terminate` 事件处理器销毁了在 `Class_Initialize` 事件中实例化的对象。这实际上是没有必要的，因为当 `FileHelper` 对象被销毁时，私有的 `mobjFSO` 变量也会

超出其作用域，脚本引擎会自动地将其销毁。但是，有些程序员喜欢显式地销毁所有实例化的对象，这对于示例还是很有用的。

还可以用 `Class_Terminate` 事件关闭数据库连接、关闭文件或是将类的某些信息保存到数据库或文件中。关于 `Class_Initialize` 事件处理器的语法限制同样也适用于 `Class_Terminate` 事件处理器。

8.6 类常量

VBScript 不支持声明类级的具名常量，原因不明。也就是说，不能在类中使用 `Const` 语句声明供类内部或外部代码使用的常量。例如，下面这段代码会产生编译错误：

```
Option Explicit

Dim objTest

Set objTest = new ConstTest

objTest.SayHello

Set objTest = Nothing

Class ConstTest

Private Const TEST_CONST = "Hello there."

Public Sub SayHello

    MsgBox TEST_CONST

End Sub

End Class
```

编译错误出现在这一行：

```
Private Const TEST_CONST = "Hello there."
```

因为这个语句的作用域仅限于这个类，这意味它是在这个类中声明的，但不是在这个类的方法或属性中。(`Const` 语句适用于属性或方法，但是其作用域也仅限于这个属性或方法。) 但是有解决的办法，比如下面这个例子：

```
Option Explicit

Dim objTest
```

```

set objTest = new ConstTest

objTest.SayHello

Class ConstTest

Private TEST_CONST

Private Sub Class_Initialize

    TEST_CONST = "Hello there."

End Sub

Public Sub SayHello

    MsgBox TEST_CONST

End Sub

End Class

```

这里创建了一个伪常量。没有用 `Const` 语句声明 `TEST_CONST`，而是将其声明为一个普通的私有类变量(也可以将其声明为公共的)。然后在 `Class_Initialize` 事件处理器中将您希望的“常量”值赋给 `TEST_CONST` 变量。但这里存在一点风险，因为类中的代码还是可以修改 `TEST_CONST` 变量的值。不过，全大写的命名方式可能有助于防止这种情况(大部分程序员都习惯将全大写的变量等同为具名常量)。您只需要确保类内部的代码不会修改这个变量的值就可以了。

请注意在老版本的 `VBScript` 中，类级常量也是不被支持的。但是，奇怪的是，这不会导致编译错误；它们的值会被忽略。如果您使用的 `VBScript` 没有产生编译错误，您还是会遇到同样的问题，上述解决方法还是有用的。

8.7 构建和使用 `VBScript` 类

第 3 章中演示了如何使用数组存放一系列的姓名和电话号码。然后第 7 章演示了如何将这个通讯录存放到脚本运行时的 `Dictionary` 对象的一系列单元元素数组中。现在，在本章剩余的内容中，我们会进一步地用 `VBScript` 类改造这个通讯录例子。跟第 7 章中的示例代码一样，现在您所开发的脚本将实现以下功能：

- 提供一个数据结构存放通讯录的条目。

- 提供一个数据结构存放一系列的通讯录条目。
- 提供一种查找和显示通讯录条目的方法。

将要开发的这个示例脚本包含以下元素：

- 用于存放单个通讯录条目的 **ListEntry** 类。这个类知道如何显示其中的数据。
- **PhoneList** 类，利用内部的 **Dictionary** 对象存放一系列的 **ListEntry** 对象。这个类用电话号码作为关键字，并且支持获取和显示单个条目。
- 支持不基于类的代码使用前面的两个类来操作通讯录，并要求用户输入电话号码以查找和显示。

您可能还记得第 7 章中的例子提供了所有的这些功能，但是没有使用类。记住这一点，本章改进第 7 章代码的目的是演示如何用类创建可读性更高、更通用、方便重用的代码，这样就可以适应进一步的修改。可以阅读、执行并尝试 **PHONE_LIST_CLASS.VBS** 文件中的代码。您可以从 wrox.com 网站下载这份代码以及本书中所有的其他代码。

首先，来看一下 **ListEntry** 类。

```
Class ListEntry
```

```
Private mstrLast
```

```
Private mstrFirst
```

```
Private mstrPhone
```

```
Public Property Let LastName(strLastName)
```

```
    If IsNumeric(strLastName) or _
```

```
    IsDate(strLastName) Then
```

```
        Err.Raise 32003, "ListEntry", _
```

```
        "The LastName property may not " & _
```

```
        "be a number or date."
```

```
    End If
```

```
    mstrLast = strLastName
```

```
End Property
```

```
Public Property Get LastName
```

```
    LastName = mstrLast
```

```

End Property

    Public Property Let FirstName(strFirstName)

    If IsNumeric(strFirstName) or _
IsDate(strFirstName) Then
Err.Raise 32004, "ListEntry", _
    "The FirstName property may not " & _
    "be a number or date."

    End If

    mstrFirst = strFirstName
End Property

Public Property Get FirstName

    FirstName = mstrFirst
End Property

Public Property Let PhoneNumber(strPhoneNumber)

    mstrPhone = strPhoneNumber
End Property

Public Property Get PhoneNumber

    PhoneNumber = mstrPhone
End Property

Public Sub DisplayEntry

    MsgBox "Phone list entry:" & vbNewLine & _
        vbNewLine & _
        "Last: " & mstrLast & vbNewLine & _
        "First: " & mstrFirst & vbNewLine & _
        "Phone: " & mstrPhone

End Sub

End Class

```

这个类有三个属性：LastName、FirstName 和 PhoneNumber。每个属性都是用私有属性

变量以及 **Property Let** 和 **Property Get** 过程实现的。因为这个类的每个属性都有 **Let** 和 **Get** 过程，所以这些属性都是可读写的。还要注意 **LastName** 和 **FirstName** 的 **Property Let** 过程，对输入的数据做检查以确保外部代码不会将数字或日期存放到这个属性中。如果传递过来的是非法值，这段代码会生成一个错误(见第 6 章)。

在做验证时只是检查其是否为数字或日期是很不严谨的；这个例子的主要目的只是说明使用 **Property Let** 过程确保程序员不会将不合适的数据存放到这个属性中。这种技术对 **VBScript** 这种弱数据类型的语言非常重要；因为所有的变量都是 **Variant** 类型的，其中可以存放任何值。

还可以选择其他类型的验证方法。可以检查数据长度(最小或最长)，非法的特殊字符，合适的格式等等。例如，可以在 **PhoneNumber Property Let** 中添加一个检查代码验证 **XXX-XXX-XXXX** 格式。或者可以添加转换功能，将电话号码转换成合适的格式。选择什么验证方法和转换方式取决于具体情况。关键是测试其他代码成功运行的前提是否成立，以避免缺陷和错误。

ListEntry 类有一个方法：**DisplayEntry**，它用 **MsgBox()**函数显示出通讯录条目的各个属性。这个例子选择了将这段代码放在 **ListEntry** 类中，因为类的一个原则就是类应该知道如何实现其所提供的功能。**ListEntry** 类知道姓名和电话号码。所以，应该尽可能地将操作数据的代码与其所操作的数据放在一起，也就是将 **DisplayEntry()**方法放在 **ListEntry** 类中。

在面向对象方法中，这称为“关注点分离”(separation of concerns)或“基于责任的设计”(responsibility-based design)。每个类都要知道一些东西，以及如何做一些事情。在设计时，要将类按逻辑分割开，尽量减少类之间不必要的交互。类之间的交互越少越好。

但是有时候有些功能是无需类知道该怎么做的。这可以使类更加通用，因此可以在多种情况下以不同的方式使用它。您将会看到这样例子，并且会这样继续构建您自己的代码。

继续，这是第二个类，**PhoneList**：

```
Class PhoneList
```

```
Private objDict
```

```
Private Sub Class_Initialize
```

```
Set objDict = CreateObject("Scripting.Dictionary")
```

```
End Sub
```

```

Private Sub Class_Terminate
    Set objDict = Nothing
End Sub

Public Property Get ListCount
    ListCount = objDict.Count
End Property

Public Function EntryExists(strPhoneNumber)

    EntryExists = _
        objDict.Exists(strPhoneNumber)
End Function

Public Sub AddEntry(objListEntry)
    If TypeName(objListEntry) <> "ListEntry" Then
        Err.Raise 32000, "PhoneList", _
            "Only ListEntry objects can be stored " & _
            "in a PhoneList class."
    End If

    'We use the PhoneNumber property as the key.
    If Trim("" & objListEntry.PhoneNumber) = "" Then
        Err.Raise 32001, "PhoneList", _
            "A ListEntry object must have a " & _
            "phone number to be added to the " & _
            "phone list."
    End If

    objDict.Add objListEntry.PhoneNumber, objListEntry
End Sub

Public Sub DisplayEntry(strPhoneNumber)
    Dim objEntry

    If objDict.Exists(strPhoneNumber) Then

```



```

    Set objEntry = objDict(strPhoneNumber)

    objEntry.DisplayEntry
Else
    Err.Raise 32002, "PhoneList", _
"The phone number" & strPhoneNumber & _
    "" is not in the list."

End If

End Sub

End Class

```

第一个要注意的事情是这个类内部使用 **Dictionary** 对象存储这个通讯录。这是一种强大的技术，原因有两点：

- 这演示了您的类如何借用其他类的功能。
- 实际上您没有将内部的 **Dictionary** 对象暴露给 **PhoneList** 类外部的代码，也就是说使用 **PhoneList** 类的代码无需知道 **PhoneList** 类是如何存储数据的。

您还可以将 **Dictionary** 改成其他的数据存储方法(比如数组、哈希表、文本文件等等)，这不会对使用 **PhoneList** 类的代码产生任何影响。第二点，本章前面已经演示过，使用 **Class_Initialize** 和 **Class_Terminate** 事件控制内存中 **Dictionary** 对象(objDict)的生存期。这样其他代码就会觉得始终有一个可以使用的 **Dictionary** 对象。

接着，创建了名为 **ListCount** 的 **Property Get** 过程，以及 **EntryExists** 方法。**ListCount** 属性是对 **objDict.Count** 属性的封装；而类似的，**EntryExists** 是 **objDict.Exists** 方法的封装。您也可以将 **Dictionary** 的其他属性方法暴露出来。但是，要小心，因为这可能会降低您将来将 **Dictionary** 对象换成其他数据存储结构的灵活性。

例如，可以只是简单地将 **objDict** 做一个属性供外部代码直接使用。但是，如果这么做，外部代码就会与类的内部紧密的耦合在一起——这意味着外部代码对类的内部工作原理知道得太多。应该尽可能地将 **PhoneList** 设计为一个“黑箱”：可以使用黑箱的功能，可以知道它的输入输出，但是无法看到其内部的工作方式。

接下来是 **AddEntry()**方法。这个方法实际上只做了一件事情：调用 **Dictionary** 的 **Add()**方法，其中关键字就是通讯录各个条目的电话号码。

```
objDict.Add objListEntry.PhoneNumber, objListEntry
```

注意，这里是将 `ListEntry` 对象本身存放到这个字典中，跟第 7 章类似，那里是将通讯条目的数组存放在字典中。

但这是这个方法最后一行。在其之前的所有代码都是验证代码。这里只是测试并将该方法的假设存放起来。这个方法有两个隐含的假设：

- `PhoneList` 类中只有 `ListEntry` 对象
- 将 `PhoneNumber` 属性作为关键字

为了测试这两个假设：

- 使用 `TypeName` 函数检查外部代码传递过来的是否是 `ListEntry` 对象，而不是其他类型的数据。由于 `VBScript` 弱类型的特点，这是很有必要的，确实需要自行对其进行验证。
- 检查 `ListEntry` 对象的 `PhoneNumber` 属性是否为空。这样就能确保能将其作为关键字使用。

还有其他需要测试的假设，但是这两个假设容易产生奇怪的缺陷或错误消息，导致用您的类的程序员很难找出是什么地方出了问题。这些清晰的错误消息文档所关注的就是这些重要的假设。

最后，`PhoneList` 有一个名为 `DisplayEntry` 的方法。等一会——在 `ListEntry` 类中不是也有一个 `DisplayEntry` 方法吗？为什么会有两个功能相同的方法？

这都取决于设计观点。没有完全正确的类设计方法。`PhoneList` 类的 `DisplayEntry` 方法实际上是“代表”`ListEntry.DisplayEntry()` 方法显示一个条目，看这几行代码。

```
If objDict.Exists(strPhoneNumber) Then
```

```
Set objEntry = objDict(strPhoneNumber)
```

```
objEntry.DisplayEntry
```

所以，尽管有两个方法，但实际上显示的代码只存在于 `ListEntry` 类中，所以并没有重复。这里的设计方式使得 `PhoneList` 类有专门的方法(比如 `DisplayEntry()`)供程序员对通讯录的条目实施专门的操作(比如显示它们)，而不是提供更通用的方法，仅仅是提供一系列的条目，让外部代码用前面的三行代码去实现这个功能——也就是找到正确的条目，并要其显示自身。两种设计都是可行的，对您将来用不同的方式扩展这些类都不会有阻碍。

现在有两个类，可以来看看使用这些类的代码了(再提醒一下，所有的这些代码都可以在 PHONE_LIST_CLASS.VBS 文件中找到)。

```
Option Explicit
```

```
Dim objList
```

```
FillPhoneList
```

```
On Error Resume Next
```

```
objList.DisplayEntry(GetNumberFromUser)
```

```
If Err.Number <> 0 Then
```

```
If Err.Number = vbObjectError + 32002 Then
```

```
    MsgBox "That phone number is not in the list.", _  
    vbInformation
```

```
Else
```

```
    DisplayError Err.Number, Err.Source, _  
    Err.Description
```

```
End If
```

```
End If
```

```
Public Sub FillPhoneList
```

```
Dim objNewEntry
```

```
Set objList = New PhoneList
```

```
Set objNewEntry = New ListEntry
```

```
With objNewEntry
```

```
    .LastName = "Williams"  
    .FirstName = "Tony"  
    .PhoneNumber = "404-555-6328"
```

```
End With
```

```
objList.AddEntry objNewEntry
```

```
Set objNewEntry = Nothing
```

```
Set objNewEntry = New ListEntry
```

```
With objNewEntry
    .LastName = "Carter"
    .FirstName = "Ron"
    .PhoneNumber = "305-555-2514"
End With

objList.AddEntry objNewEntry

Set objNewEntry = Nothing

Set objNewEntry = New ListEntry

With objNewEntry
    .LastName = "Davis"
    .FirstName = "Miles"
    .PhoneNumber = "212-555-5314"
End With

objList.AddEntry objNewEntry

Set objNewEntry = Nothing

Set objNewEntry = New ListEntry

With objNewEntry
    .LastName = "Hancock"
    .FirstName = "Herbie"
    .PhoneNumber = "616-555-6943"
End With

objList.AddEntry objNewEntry

Set objNewEntry = Nothing

Set objNewEntry = New ListEntry

With objNewEntry
    .LastName = "Shorter"
    .FirstName = "Wayne"
    .PhoneNumber = "853-555-0060"
```

```
End With  
  
objList.AddEntry objNewEntry  
  
Set objNewEntry = Nothing  
  
End Sub  
  
Public Function GetNumberFromUser  
GetNumberFromUser = InputBox("Please enter " & _  
    "a phone number (XXX-XXX-XXXX) with " & _  
    "which to search the list.")  
  
End Function
```

运行这段代码，输入电话号码 404-555-6328，结果是如图 8-5 中所示的对话框。



图 8-5

再运行这段代码，输入一个非法的电话号码，结果是如图 8-6 中所示的对话框。



图 8-6

这个精巧的例子中最重要的一点就在这两行简单的代码中(没有包括与错误处理相关的代码)。

FillPhoneList

```
objList.DisplayEntry(GetNumberFromUser)
```

这两行代码就展示了整个脚本的逻辑：创建一个通讯录，填入内容，要求用户输入需要查找的电话号码，然后显示这个条目。这就是将代码分割成类和过程所带来的好处。如果将类和过程做得尽可能的通用，那么利用它们实现具体功能的代码就会比较简单、易于理解和修改——总的来说也就是方便重用。

`FillPhoneList()`过程创建 `PhoneList` 对象(`objList`)，并填入一些条目。想象一下，通讯录的条目可以来自于数据库表格、文件或用户的输入。`FillPhoneList()`使用了一个名为 `objNewEntry` 的“临时对象变量”。对于列表中的每个条目，都实例化 `objNewEntry`，填入数据，然后将其传递给 `objList.AddEntry()`方法。

注意在 `FillPhoneList` 中使用 `New` 关键字实例化自定义的 VBScript 对象。

```
Set objList = New PhoneList
```

以及

```
Set objNewEntry = New ListEntry
```

为什么不用 `CreateObject()`函数？`CreateObject()`函数只是用于实例化非原生的 VBScript 对象(比如 `Dictionary` 和 `FileSystemObject`)，而您必须用 `New` 来实例化同一个脚本中自定义的 VBScript 类。背后的原因很复杂，所以只需要记住这条原则：如果用自定义的 VBScript 类实例化一个对象，使用 `New`；否则，用 `CreateObject`。

`GetNumberFromUser()`函数非常简单。用 `InputBox()`函数提示用户输入一个电话号码，并返回用户的输入。然后脚本开头的代码将这个值传递给 `objDict.DisplayEntry()`。如果这个条目存在，`ListEntry` 对象就将其显示出来；如果不存在，`objDict.DisplayEntry()`就返回一个错误。

还可以用 `PhoneList` 类和 `ListEntry` 类用不同的方式实现各种不同的功能。如果有新的需求，可以扩展这两个类，而不会破坏正在使用它们的代码。将来使用这个脚本的程序员在短时间内就能理解这个脚本。毕竟，这只有两行代码。

`FillPhoneList`

```
objList.DisplayEntry(GetNumberFromUser)
```

如果有程序员想进一步地了解脚本工作原理的底层细节，他可以继续阅读其他的代码，深入这些过程和类。但是很多情况下，这都没有必要，除非程序员需要修复缺陷或是添加一些功能。如果程序员要知道的只是这个脚本在做什么，答案就是这两行简单的代码。

8.8 小结

本章介绍了如何在 VBScript 中编写类。这是一种强大的能力，可以创建面向对象的脚本，在合理的设计下，可以极大地提高代码的可理解性、可维护性、灵活性和重用性。在 VBScript 中，用 `Class ... End Class` 代码块语句定义类。通常，类的定义必须与使用这个类的代码处于同一个脚本文件中。(关于可供替换的技术，请查阅第 16 章中关于 Windows 脚本组件的信息。)

类可以有属性和方法。可以用公共变量或特殊的名为 `Property Let`、`Get`、`Set` 的过程来定义属性。使用私有属性变量和不同的 `Let`、`Get` 及 `Get` 过程的组合，可以控制属性的只读、只写或可读写属性。定义方法跟定义普通的过程和类差不多，可以是公共的或私有的。

本章最后一节给出了以详细的基于类的例子，并对其中的设计和编程技术做了解释。这个例子是第 3 章和第 7 章中通讯录例子的扩展。

第 9 章 正则表达式

VBScript 从第 5 版开始就对正则表达式提供了完整的支持。在那之前，这是 VBScript 的一个缺陷，与包括 JavaScript 在内的其他脚本相比也是一个弱项。在介绍正则表达式给程序员带来的便利之前，本章会先简要地介绍正则表达式。然后再看看如何在 VBScript 代码中使用正则表达式。

9.1 正则表达式简介

正则表达式提供了强大的字符匹配和替换功能。在 VBScript 引擎增加正则表达式之前，需要很多代码，包括循环、InStr 和 Mid 函数，才能实现对字符串的查找和替换。现在有了正则表达式，只要一行代码就能实现这些。

如果您以前使用过其他语言(C#、C++、Perl、awk 或 JavaScript——甚至微软自己的 Jscript 都在 VBScript 之前提供了对正则表达式的支持)，正则表达式对您来说可能就不新鲜了。但有经验的程序员在 VBScript 中使用正则表达式时也要注意 VBScript 并不支持正则表达式常量(就像/a pattern/)，而是将文本字符串赋给 RegExp 对象的 Pattern 属性。很多情况下这都比传统的方法好，因为无需学习新的语法。但是如果您曾经使用过其他语言中的正则表达式，尤其是客户端 JavaScript，这会让您感到意外。

现在很多 Windows 平台的文本编辑器也开始模仿 UNIX 文本编辑器 vi 提供对正则表达式搜索的支持。其中包括 UltraEdit-32(www.ultraedit.com)和 SlickEdit(www.slickedit.com)。

9.1.1 实战正则表达式

熟悉正则表达式的最好、最快的途径就是看例子。下面这个例子可能是最简单的正则表达式例子，涉及到查找和替换。

```
Dim re, s
Set re = New RegExp
re.Pattern = "France"
```



```
s = "The rain in France falls mainly on the plains."
```

```
MsgBox re.Replace(s, "Spain")
```

没什么特别的——但这为以后的应用打下了很好的基础。下面来看看这段代码的原理。

1. 新建一个正则表达式对象。

```
Set re = New RegExp
```

2. 设置这个对象的关键属性。这就是需要匹配的模式。

```
re.Pattern = "France"
```

3. 接下来这一行是需要搜索的字符串：

```
s = "The rain in France falls mainly on the plains."
```

这一行是这个脚本真正的能力源泉。它要求正则表达式查找这个变量中最早出现的"France"(模式)，并将其替换成"Spain"。

4. 完成之后，用消息框展示强大的查找和替换技能。

```
MsgBox re.Replace(s, "Spain")
```

5. 脚本运行的最终输出如图 9-1 所示。



图 9-1

现在这个脚本一开始就固定了字符串和查找的标准，但是也可以从外部输入获取字符串和查找替换标准，使这个脚本更加灵活。

```
Dim re, s, sc
```

```
Set re = New RegExp
```

```
s = InputBox("Type a string for the code to search")
```

```
re.Pattern = InputBox("Type in a pattern to find")
```

```
sc = InputBox("Type in a string to replace the pattern")
```

```
MsgBox re.Replace(s, sc)
```

这跟前面的代码基本上是一样的，主要的区别在于没有将所有的功能都固化在脚本内，而是通过代码中的三个输入框增加了脚本的灵活性。

```
s = InputBox("Type a string for the code to search")
```

```
re.Pattern = InputBox("Type in a pattern to find")
```

```
sc = InputBox("Type in a string to replace the pattern")
```

最终的修改是在最后一行代码中 `Replace` 方法使用了 `sc` 变量。

```
MsgBox re.Replace(s, sc)
```

这样就可以手动输入需要搜索的字符串，如图 9-2 所示。



图 9-2

然后输入需要查找的模式(见图 9-3)。



图 9-3

最后输入一个用来替换模式的字符串(见图 9-4)。

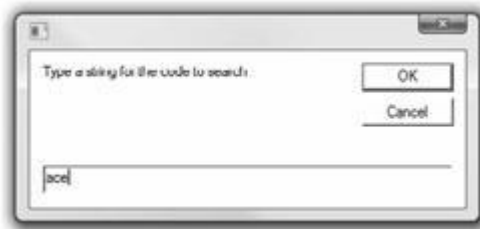


图 9-4

您可以尝试您想要尝试的东西；如果试图查找和替换字符串中不存在的模式会怎么样？实际上什么都不会发生，如下所示。

1. 输入如图 9-5 所示的字符串。



图 9-5

2. 输入一个不存在的模式(就是字符串中没有的东西)。如图 9-6 所示使用的是 JScript。



图 9-6

3. 在接下来的输入框中输入字符串替换这个不存在的模式。由于不会发生替换，所以它可以是任何东西。图 9-7 中使用的是"JavaScript"，注意发生了什么。什么都没有发生。从图 9-8 中可以看到，最初的字符串没有任何改变。



图 9-7



图 9-8

9.1.2 从简单的开始

显然，您之前所看到的例子都很简单，但是老实地说，只需要 VBScript 的字符串操作函数就能实现这些功能。但如果要替换字符串中出现的所有模式怎么办？或是要替换字符串中出现的所有的模式，除了出现在单词结尾的字符串？需要调整一下这段代码。看一下接下来的代码：

```
Dim re, s  
  
Set re = New RegExp  
re.Pattern = "\bin"
```

```
re.Global = True  
  
s = "The rain in Spain falls mainly on the plains."  
  
MsgBox re.Replace(s, "in the country of")
```

这个版本有两处关键的变化：

- 用特殊序列(`\b`)匹配单词边界(您将在本章稍后的 9.2.4 一节中看到所有的特殊序列)。如图 9-9 所示。



图 9-9

假设这里没有`\b`，就像这样：

```
Dim re, s  
  
Set re = New RegExp  
  
re.Pattern = "in"  
  
re.Global = True  
  
s = "The rain in Spain falls mainly on the plains."  
  
MsgBox re.Replace(s, "in the country of")
```

没有了`\b`，单词“rain”、“Spain”、“mainly”和“plains”中的“in”也都会被替换成“in the country of”。如图 9-10 所示，这会产生很有趣的结果，但这不是我们想要的。



图 9-10

- 通过设置 `Global` 属性，确保匹配到所有您想要匹配到的“in”。

```
Dim re, s  
  
Set re = New RegExp
```

```
re.Pattern = "in"

re.Global = True

s = "The rain in Spain falls mainly on the plains."

MsgBox re.Replace(s, "in the country of")
```

正则表达式提供了一种强大的语言来实现复杂的模式匹配，让我们继续学习能让您在 VBScript 中使用正则表达式的对象。

9.2 RegExp 对象

RegExp 对象是 VBScript 中用于提供简单的正则表达式支持的对象。VBScript 中所有和正则表达式有关的属性和方法都与这个对象有关联。

```
Dim re

Set re = New RegExp
```

这个对象有三个属性和三个方法，如表 9-1 所示。

表 9-1

属 性	Global 属性
	IgnoreCase 属性
	Pattern 属性
方 法	Execute 方法
	Replace 方法
	Test 方法

接下来的几节会深入地介绍这些属性和方法。此外还会介绍您将在模式中用到的正则表达式符号。

9.2.1 Global 属性

Global 属性负责设置或返回一个 Boolean 值，指明模式是匹配整个字符串中所有与之相符的地方还是只匹配第一次出现的地方(参见表 9-2)。

表 9-2

代码	object.Global [= value]
对象	RegExp 对象

值	有两个可能的值: True 和 False
	如果 Global 属性的值是 True, 那就会对整个字符串进行查找; 否则就不会。默认值是 False—— 并不是微软的某些文档中说的 True

下面的例子利用 Global 属性确保所有的"in"都会被修改。

```
Dim re, s
Set re = New RegExp
re.Pattern = "\bin"
re.Global = True
s = "The rain in Spain falls mainly on the plains."
MsgBox re.Replace(s, "in the country of")
```

9.2.2 IgnoreCase 属性

IgnoreCase 属性负责设置或返回一个 Boolean 值, 指明模式匹配是否大小写敏感(参见表 9-3)。

表 9-3

代码	object.IgnoreCase [= value]
对象	RegExp 对象
值	有两个可能的值: True 和 False
	如果 IgnoreCase 属性的值为 False, 搜索为大小写敏感; 如果是 True, 则不是。默认是 False—— 并不是微软的某些文档中说的 True

继续看这个例子, 之前看过了 Global 属性; 如果要匹配的字符串中有“In”就必须告诉 VBScript 在进行匹配时要忽略大小写。

```
Dim re, s
Set re = New RegExp
re.Pattern = "\bin"
re.Global = True
re.IgnoreCase = True
s = "The rain In Spain falls mainly on the plains."
MsgBox re.Replace(s, "in the country of")
```

9.2.3 Pattern 属性

Pattern 属性设置或返回用于搜索的正则表达式(参见表 9-4)。

前面所有的例子都用到了 Pattern。

```
Dim re, s
Set re = New RegExp
re.Pattern = "\bin"
re.Global = True
s = "The rain In Spain falls mainly on the plains."
MsgBox re.Replace(s, "in the country of")
```

表 9-4	
代码	object.Pattern [= "searchstring"]
对象	RegExp 对象
搜索字符串	需要搜索的正则字符串表达式。可能含有一些正则表达式字符 —— 可选的

9.2.4 正则表达式字符

正则表达式的强大并不是来自于用字符串做模式，而是在模式中使用特殊字符。表 9-5 列出了所有的这些字符，以及每个字符在代码中的作用。

大写特殊字符的作用与相应的小写特殊字符的作用相反。

表 9-5	
字 符	描 述
\	表示下一个字符是特殊字符或文字常量
^	匹配输入的开头
\$	匹配输入的结尾
*	匹配前一个字符零次或多次
+	匹配前一个字符一次或多次
?	匹配前一个字符零次或一次
.	匹配除换行符以外的任何单个字符
(pattern)	匹配并记住这个模式。可以用[0]...[n]从结果的 matches 集合中获取匹配到的字符串。要匹配括号本身，在前面加上斜杠—— 用\"(\"或\")\"
(?:pattern)	匹配但不捕获模式，也就是不会存储匹配结果供以后使用。这可以用于使

	用"or"字符()合并模式的不同部分。例如, "anomal(?:y ies)"比 "anomaly anomalies"要划算得多
(?=pattern)	当所要搜索的字符串匹配了模式的开头部分时就接着匹配这一部分。这是一个非捕获匹配,也就是说不会保存匹配结果供以后使用。例如, "Windows (?:95 98 NT 2000 XP Vista)"能匹配"Windows Vista"中的 Windows 而不能匹配"Windows 3.1"中的 Windows
(?!pattern)	与上一个相反,这会匹配模式中没有出现的内容。这是一个非捕获匹配,也就是说不会保存匹配结果供以后使用。例如, "Windows (?:95 98 NT 2000 XP Vista)"能匹配"Windows 3.1"中的 Windows 而不能匹配"Windows Vista"中的 Windows
x y	匹配 x 或 y

(续表)

字 符	描 述
{n}	准确地匹配 n 次(n 必须是一个非负整数)
{n,}	至少匹配 n 次(n 必须是一个非负整数—— 注意结尾的逗号)
{n,m}	至少匹配 n 次,最多匹配 m 次(m 和 n 必须都是非负整数)
[xyz]	匹配其中包括的任一个字符(xyz 表示一个字符集)
[^xyz]	匹配其中不包括的字符(^xyz 表示一个字符集的补集)
[a-z]	匹配指定范围内的字符(a-z 表示字符的范围)
[m-z]	匹配指定范围以外的字符(^m-z 表示指定范围的补集)
\b	匹配一个单词边界,这个位置在单词和空格之间
\B	匹配一个非单词边界
\d	匹配数字。等价于[0-9]
\D	匹配非数字。等价于[^0-9]
\f	匹配换页符
\n	匹配换行符
\r	匹配回车符
\s	匹配空白,包括空格、制表符、换页符等。等价于"[\f \n \r \t \v]"
\S	匹配非空白的字符。等价于"[^\f \n \r \t \v]"
\t	匹配制表符
\v	匹配纵向制表符
\w	匹配字母、数字,以及下划线。等价于"[A-Za-z0-9_]"
\W	匹配非字符数字。等价于"[^A-Za-z0-9_]"
\.	匹配.
\\	匹配\
\{	匹配{
\}	匹配}
\\	匹配\
\[匹配[
\]	匹配]
\(匹配(

\)	匹配)
\$ num	匹配 num，其中 num 是正整数。返回匹配结果的引用
\n	匹配 n，其中 n 是八进制转义符。八进制转义符的长度应为 1、2 或 3
\uxxxx	匹配 UNICODE 形式的 ASCII 字符
\xn	匹配 n，其中 n 是十六进制转义符。十六进制转义符必须是两位长度

这其中很多代码都无需太多的说明，但是有些例子可能需要别人的帮助才能理解。

匹配一类字符

您已经见过一个简单的模式：

```
re.Pattern = "in"
```

它通常用来匹配一类字符。通过将需要匹配的字符放在方括号中就能实现。例如，下面这个例子将单个数字换成更通用的术语。

```
Dim re, s
Set re = New RegExp
re.Pattern = "[23456789]"
s = "Spain received 3 millimeters of rain last week."
MsgBox re.Replace(s, "many")
```

这段代码的输出如图 9-11 所示。



图 9-11

在这个例子中，数字“3”被替换成了文本“many”。正如您所期望的，可以指定一个范围来缩短这个模式。这个模式跟前一个的功能完全一样。

```
Dim re, s
Set re = New RegExp
re.Pattern = "[2-9]"
s = "Spain received 3 millimeters of rain last week."
```

```
MsgBox re.Replace(s, "many")
```

替换数字和非数字

经常需要替换数字。实际上，由于经常要用到模式[0-9](包括所有数字)，所以有一种[0-9]的等价快捷方式：`\d`。

```
Dim re, s
```

```
Set re = New RegExp
```

```
re.Pattern = "\d"
```

```
s = "a b c d e f 1 g 2 h ... 10 z"
```

```
MsgBox re.Replace(s, "a number")
```

替换后的字符串如图 9-12 所示。



图 9-12

如果要匹配非数字的字符怎么办？在方括号中使用`^`符号。

在方括号外使用`^`的意义就完全不同了，稍后会对此做讨论。

这样就可以使用下面的模式匹配非数字的字符：

```
re.Pattern = "[^,0-9]" 'the hard way
```

```
re.Pattern = "[^\d]" 'a little shorter
```

```
re.Pattern = "[\D]" 'another of those special characters
```

最后一个模式使用了另一种特殊字符。在大部分情况下这种特殊字符只是减少了您的输入次数(或是一种有效的记忆方法)，但在有些情况下，比如遇到匹配制表符和其他不能打印的字符时，这就很有用了。

锚定和缩短模式

有三种特殊字符用于锚定模式。它们本身不匹配任何字符，但是可以要求另一个模式必须出现在输入的开头(在[]外使用^)、输入的结尾(\$)或是单词边界(您已经见过的\b)。

另一种缩短模式的方法是使用重复数。基本的思路就是在模式后面指定重复的次数。例如，下面这个模式，如图 9-13 所示，可以匹配多个数字并替换它们。

```
Dim re, s
Set re = New RegExp
re.Pattern = "\d{3}"
s = "Spain received 100 millimeters of rain in the last 2 weeks."
MsgBox re.Replace(s, "a whopping number of")
```

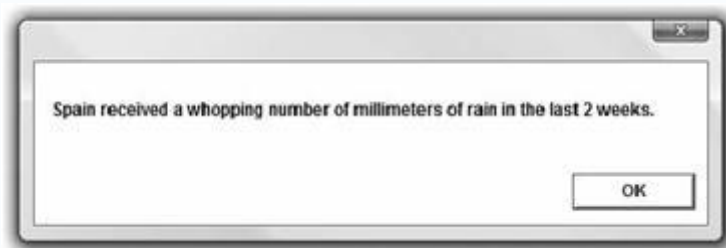


图 9-13

如果不在代码中使用重复数，如图 9-14 所示，它会留下最后字符串中的"00"。

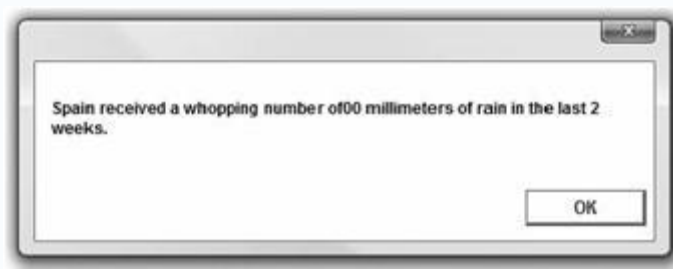


图 9-14

```
Dim re, s
Set re = New RegExp
re.Pattern = "\d"
s = "Spain received 100 millimeters of rain in the last 2 weeks.\"
MsgBox re.Replace(s, "a whopping number of")
```

还要注意，这里不能使用 `re.Global = True`，因为这样会在结果中产生 4 个“a whopping number of”。结果如图 9-15 所示。

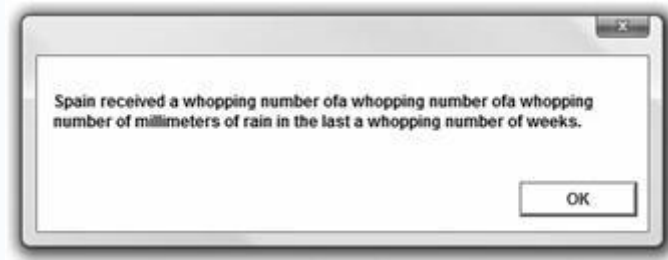


图 9-15

```
Dim re, s
Set re = New RegExp
re.Global = True
re.Pattern = "\d"
s = "Spain received 100 millimeters of rain in the last 2 weeks."
MsgBox re.Replace(s, "a whopping number of")
```

指定匹配的范围或最小次数

前面的表中已经罗列了，还可以指定匹配的最小次数{min}或范围{min, max,}。其中一些常用的重复模式也有专门的快捷方式。

```
re.Pattern = "\d+" 'one or more digits, \d{1, }
re.Pattern = "\d*" 'zero or more digits, \d{0, }
re.Pattern = "\d?" 'optional: zero or one, \d{0,1}

Dim re, s
Set re = New RegExp
re.Global = True
re.Pattern = "\d+"
s = "Spain received 100 millimeters of rain in the last 2 weeks."
MsgBox re.Replace(s, "a number")
```

这段代码的输出如图 9-16 所示。注意字符串“100”被替换了。

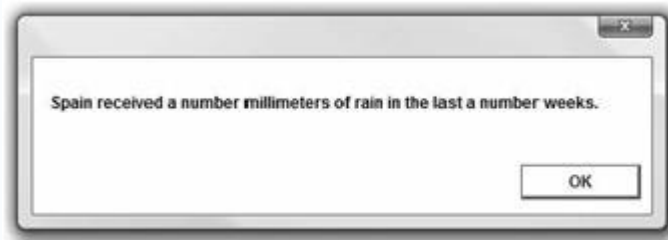


图 9-16

```
Dim re, s
Set re = New RegExp
re.Global = True
re.Pattern = "\d*"
s = "Spain received 100 millimeters of rain in the last 2 weeks."
MsgBox re.Replace(s, "a number")
```

上面这段代码的输出如图 9-17 所示。这里在每两个非数字的字符间插入了这个字符串，而数字则被替换了。

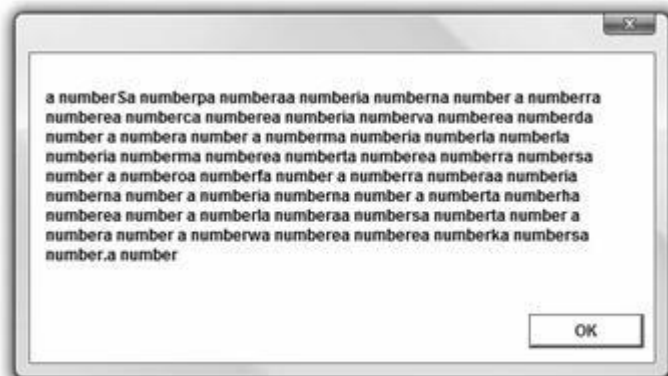


图 9-17

```
Dim re, s
Set re = New RegExp
re.Global = True
re.Pattern = "\d?"
s = "Spain received 100 millimeters of rain in the last 2 weeks."
MsgBox re.Replace(s, "a number")
```

上面这段代码的输出如图 9-18 所示。这里也在每两个非数字的字符间插入了“a numbe

r”；而数字则被替换了。

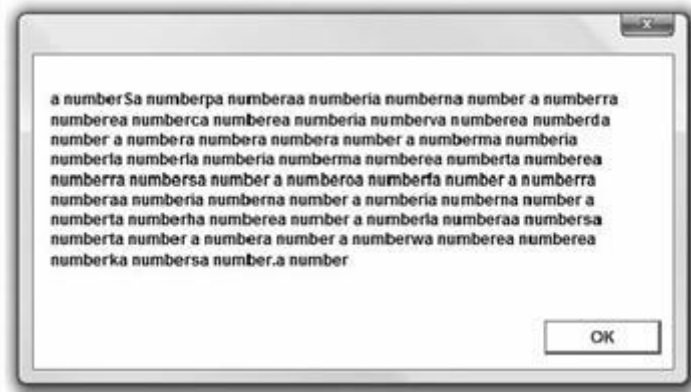


图 9-18

记住匹配结果

最后一个要讨论的特殊字符是记住匹配的结果。如果要在用于替换的文本中使用部分或全部的匹配结果，这就很有用——详见 **Replace** 方法，其中一个例子使用了记住匹配的结果。

为了验证这一点，也为了将所有关于特殊字符的讨论集中在一起，我们来做点有实际意义的事情。搜索一个字符串，查找其中的 **URL**。为了控制这个例子的复杂度和规模，这里只查找其中的“**http:**”协议，但是您还可以处理各种 **DNS** 域名，包括无限的域名层次。不要担心如何与 **DNS** 交流，只需要知道在浏览器中输入 **URL** 就足够了。

下一节中关于另一个 **RegExp** 对象方法的代码中会有更多的细节信息。现在，只需要知道 **Execute** 会执行模式匹配并通过集合返回各个匹配结果。这里是代码：

```
Dim re, s
Set re = New RegExp
re.Global = True
re.Pattern = "http://(\w+[\w-]*\w+\.)*\w+"
s = "http://www.kingsley-hughes.com is a valid web address. And so is "
s = s & vbCrLf & "http://www.wrox.com. And "
s = s & vbCrLf & "http://www.pc.ibm.com - even with 4 levels."
Set colMatches = re.Execute(s)
For Each match In colMatches
```

```
MsgBox "Found valid URL: " & match.Value
```

Next

如您所愿，其中主要的工作就是设置模式的那一行代码。看上去有点让人生畏，但实际上很容易理解。让我们将其分解开来：

1. 模式以固定的字符串 `http://` 开头。然后用圆括号将模式的主要部分括起来。下面高亮的模式会匹配一级 DNS，包括尾部的点：

```
re.Pattern = "http://(\ w[ \ w-]* \ w \ . )*\w+"
```

这个模式以一个您之前见过的特殊字符 `\w` 开头，用来匹配 `[a-zA-Z0-9]`，也就是英语中的所有数字和字母。

2. 用括号匹配字母数字或横杠，因为 DNS 中可以有横杠。为什么不使用与前面一样的模式？很简单，因为有效的 DNS 不能以横杠开始或结尾。然后用 `*` 重复匹配 0 个或多个字符。

```
re.Pattern = "http://(\w [ \ w-]* \w\.*\w+"
```

3. 然后又严格地用字母数字，这样域名就不会以横杠结束。括号中的最后一个模式匹配用于分割 DNS 层次的点(.)。

不能单独使用点，因为那是一个特殊字符，正常情况下能匹配除换行符以外的任何字符。可以用反斜杠转义这个字符。

4. 在将这些东西封装到括号中之后，只需要继续使用 `*` 重复这个模式即可。所以下面高亮显示的模式可以匹配所有有效的域名以及其后的点。换句话说就是能匹配整个 DNS 中的一级域名。

```
re.Pattern = "http:// ( \ w[ \ w-]*\w\ . )*\w+"
```

5. 模式最后是顶级域名(比如 `com`、`org`、`edu` 等)所需的一个或多个字符。

```
re.Pattern = "http://(\w[\w-]*\w\.)*\ w+ "
```

9.2.5 Execute 方法

这个方法将正则表达式应用到字符串上并返回 `Matches` 集合。这是代码中使用模式匹配字符串的启动开关，使用方法参见表 9-6。

表 9-6

代码	object.Execute(string)
对象	只能是 RegExp 对象
字符串	需要搜索的字符串—— 必需的

正则表达式搜索的模式就是 RegExp 对象的 Pattern 属性。

```
Dim re, s

Set re = New RegExp

re.Global = True

re.Pattern = "http://(\w+[\w-]*\w+\.)*\w+"

s = "http://www.kingsley-hughes.com is a valid web address. And so is "
s = s & vbCrLf & "http://www.wrox.com. And " s = s & vbCrLf &
"http://www.pc.ibm.com - even with 4 levels."

Set colMatches = re.Execute(s)

For Each match In colMatches

MsgBox "Found valid URL: " & match.Value

Next
```

要注意，有些语言对正则表达式结果的处理方式不一样，其 Execute 返回的是判断模式是否找到的布尔值。这种差异导致您经常会看到从其他语言中转换过来的正则表达式在 VBScript 中无法使用。

有些微软的文档也含有这类错误，不过其中大部分都已经改正了。

记住 Execute 的结果是一个集合，甚至很有可能是一个空集合。可以用 if re.Execute(s).count = 0 或专门为这个目的设计的 Test 方法来测试它。

9.2.6 Replace 方法

这个方法用于替换在正则表达式搜索中找到的文本，用法参见表 9-7。

表 9-7

代码	object.Replace(string1, string2)
对象	只能是 RegExp 对象
字符串 1	这是发生替换的文本字符串—— 必需的
字符串 2	这是用于替换的文本字符串—— 必需的

Replace 方法返回一份 RegExp.Pattern 被 string2 替换后的 string1 的副本。如果字符串中没有发生匹配，那么就会返回没有任何改变的 string1。

```
Dim re, s

Set re = New RegExp

re.Pattern = "http://(\w+[\w-]*\w+\.)*\w+"

s = "http://www.kingsley-hughes.com is a valid web address. And so is "
s = s & vbCrLf & "http://www.wrox.com. And "
s = s & vbCrLf & "http://www.pc.ibm.com - even with 4 levels."

MsgBox re.Replace(s, "** TOP SECRET! **")
```

上面这段代码的输出如图 9-19 所示。

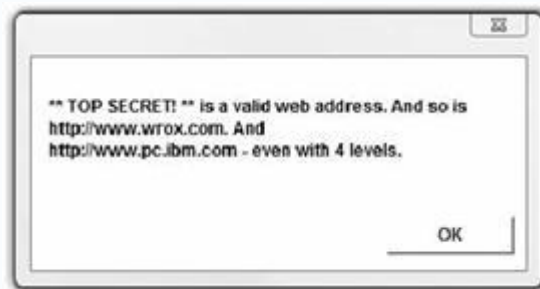


图 9-19

Replace 方法还可以替换模式中的子表达式。这需要在用于替换的文本中使用特殊字符 \$1、\$2 等。这些“参数”就是记住的匹配结果。

9.2.7 Backreferencing

一个被记住的匹配结果就是模式的一部分。这就是所谓的 backreferencing。需要用圆括号指定需要存储在临时缓存中的部分。每个捕获到的匹配结果都会按匹配到的先后次序存放(在正则表达式模式中从左到右)。存放匹配结果的缓存从 1 开始编号，最大可以到 99。可以依次用 \$1、\$2 之类的变量访问它们。

可以用非捕获元字符("(?:"、"?="或"?!")跳过正则表达式中的某些部分。

接下来的例子，前 5 个单词(由一个或多个非空白字符组成)会被记住，然后只有其中的 4 个会出现在替换文本中：

```
Dim re, s

Set re = New RegExp

re.Pattern = "(\S+)\s+(\S+)\s+(\S+)\s+(\S+)\s+(\S+)\s+(\S+)"

s = "VBScript is not very cool."

MsgBox re.Replace(s, "$1 $2 $4 $5")
```

这段代码的输出如图 9-20 所示。



图 9-20

注意这段代码中为字符串中的每个代词都添加了一对 `(\S+)\s+`。这使得代码能更好地控制所要处理的字符串。可以阻止字符串的尾部被添加到要显示的字符串中。要注意在使用 `backreferencing` 时要确保输出符合您的要求！

9.2.8 Test 方法

`Test` 方法对字符串执行正则表达式搜索，并返回一个布尔值说明匹配是否成功，用法见表 9-8。

表 9-8

代码	<code>object.Test(string)</code>
对象	RegExp 对象
字符串	正则表达式搜索的执行对象——必需的

如果匹配成功，`Test` 方法返回 `True`；否则返回 `False`。这适用于判断字符串是否含有某个模式。注意，常常需要将模式设为大小写敏感，就像下面这个例子：

```
Dim re, s

Set re = New RegExp

re.IgnoreCase = True

re.Pattern = "http://(\w+[\w-]*\w+\.)*\w+"
```

```
s = "Some long string with http://www.wrox.com buried in it."
If re.Test(s) Then
    MsgBox "Found a URL."
Else
    MsgBox "No URL found."
End If
```

这段代码的输出如图 9-21 所示。



图 9-21

9.3 Matches 集合

Matches 集合含有正则表达式的 Match 对象。

只有使用 RegExp 对象的 Execute 方法才能创建这个集合。要记住 Matches 集合的属性跟独立的 Match 对象的属性一样，都是只读的。

关于 RegExp 对象详见“RegExp 对象”一节。

当一个正则表达式被执行时，会产生 0 个或多个 Match 对象。每个 Match 对象提供以下三个内容：

- 正则表达式所找到的字符串
- 字符串的长度
- 指向找到该匹配的位置的索引

要记得将 Global 属性设为 True，否则您的 Matches 集合中最多也只会会有一个成员。这种方法很简单，但是很难调试！

```
Dim re, objMatch, colMatches, sMsg
Set re = New RegExp
```

```

re.Global = True re.Pattern = "http://(\w+[\w-]*\w+\.)*\w+" s =
"http://www.kingsley-hughes.com is a valid web address. And so is "
s = s & vbCrLf & "http://www.wrox.com. As is "
s = s & vbCrLf & "http://www.wiley.com."

Set colMatches = re.Execute(s)

sMsg = ""

For Each objMatch in colMatches

sMsg = sMsg & "Match of " & objMatch.Value

sMsg = sMsg & ", found at position " & objMatch.FirstIndex & " of
the string."

sMsg = sMsg & "The length matched is "

sMsg = sMsg & objMatch.Length & "." & vbCrLf

Next

MsgBox sMsg

```

9.3.1 Matches 的属性

Matches 是一个简单的集合，只有两个属性：

1. Count 返回集合中的元素数量。

```

Dim re, objMatch, colMatches, sMsg

Set re = New RegExp

re.Global = True

re.Pattern = "http://(\w+[\w-]*\w+\.)*\w+"

s = "http://www.kingsley-hughes.com is a valid web address. And so is "

s = s & vbCrLf & {"&} "http://www.wrox.com. As is "

s = s & vbCrLf & "http://www.wiley.com."

Set colMatches = re.Execute(s)

MsgBox colMatches.count

```

这段代码的输出如图 9-22 所示。



图 9-22

2. Item 根据指定的键返回元素。

```
Dim re, objMatch, colMatches, sMsg
Set re = New RegExp
re.Global = True
re.Pattern = "http://(\w+[\w-]*\w+\.)*\w+"
s = "http://www.kingsley-hughes.com is a valid web address. And so is "
s = s & vbCrLf & "http://www.wrox.com. As is "
s = s & vbCrLf & "http://www.wiley.com."
Set colMatches = re.Execute(s)
MsgBox colMatches.item(0)
MsgBox colMatches.item(1)
MsgBox colMatches.item(2)
```

9.3.2 Match 对象

Match 对象是 Matches 集合中的成员。创建 Match 对象的唯一方法就是使用 RegExp 对象的 Execute 方法。当一个正则表达式被执行时会产生 0 个或多个 Match 对象。每个 Match 对象提供以下内容：

- 正则表达式所找到的字符串
- 字符串的长度
- 指向找到该匹配的位置的索引

Match 的三个属性都是只读的：FirstIndex、Length 以及 Value。接下来会详细介绍这些

属性。

FirstIndex 属性

FirstIndex 属性返回匹配结果在字符串中的位置，用法见表 9-9。

表 9-9

代码	object.FirstIndex
对象	只能是 Match 对象

FirstIndex 属性从 0 开始对被搜索的字符串编号。换句话说就是字符串中的第一个字符就是字符 0。

```
Dim re, objMatch, colMatches, sMsg
Set re = New RegExp
re.Global = True
re.Pattern = "http://(\w+[\w-]*\w+\.)*\w+" s = "http://www.kingsley-hughes.com _
is a valid web address. And so is"
s = s & vbCrLf & "http://www.wrox.com. As is " s = s & vbCrLf _
& "http://www.wiley.com."
Set colMatches = re.Execute(s)
sMsg = ""
For Each objMatch in colMatches
sMsg = sMsg & "Match of " & objMatch.Value
sMsg = sMsg & ", found at position " & objMatch.FirstIndex & " of the string."
sMsg = sMsg & "The length matched is "
sMsg = sMsg & objMatch.Length & "." & vbCrLf
Next
MsgBox sMsg
```

Length 属性

Length 属性返回在字符串中找到的匹配的长度，用法见表 9-10。

代码	object.Length
对象	只能是 Match 对象

这里是一个实际中使用 Length 属性的例子。

```
Dim re, objMatch, colMatches, sMsg
Set re = New RegExp
re.Global = True re.Pattern = "http://(\w+[\w-]*\w+\.)*\w+"
s = "http://www.kingsley-hughes.com is a valid web address. And so is "
s = s & vbCrLf & "http://www.wrox.com. As is "
s = s & vbCrLf & "http://www.wiley.com."
Set colMatches = re.Execute(s)
sMsg = ""
For Each objMatch in colMatches
    sMsg = sMsg & "Match of " & objMatch.Value
    sMsg = sMsg & ", found at position " & objMatch.FirstIndex & "of the string. "
    sMsg = sMsg & "The length matched is "
    sMsg = sMsg & objMatch.Length & "." & vbCrLf
Next
MsgBox sMsg
```

Value 属性

Value 属性返回在字符串中找到的匹配结果的值或文本，用法见表 9-11。

表 9-11

代码	object.Value
对象	只能是 Match 对象

这里是一个实际中使用 Value 属性的例子。

```

Dim re, objMatch, colMatches, sMsg

Set re = New RegExp

re.Global = True re.Pattern = "http://(\w+[\w-]*\w+\.)*\w+"

s = "http://www.kingsley-hughes.com is a valid web address. And so is "

s = s & vbCrLf & "http://www.wrox.com. As is "

s = s & vbCrLf & "http://www.wiley.com."

Set colMatches = re.Execute(s) sMsg = ""

For Each objMatch in colMatches

sMsg = sMsg & "Match of " & objMatch.Value

sMsg = sMsg & ", found at position " & objMatch.FirstIndex & "

of the string. "

sMsg = sMsg & "The length matched is "

sMsg = sMsg & objMatch.Length & "." & vbCrLf

Next

MsgBox sMsg

```

9.4 一些例子

在前面内容中，您已经学到了很多理论知识。虽然理论很重要，但还得看看如何在实际中应用正则表达式。本章接下来会提供几个例子，演示如何用正则表达式解决真正的问题。

9.4.1 验证电话号码输入

对用户的输入进行验证可以防止虚假信息。很多开发人员都需要验证一段信息是否是正确输入的电话号码。但是您无法编写一个脚本来检查一个数字是否是真正的电话号码，可以用脚本和正则表达式控制输入的格式，尽可能地避免错误的信息。

这里是一段简单的验证标准美国电话号码的代码，格式是 (XXX) XXX-XXXX。

```

Dim re, s, objMatch, colMatches

Set re = New RegExp

```



```
re.Pattern = "\\([0-9]{3}\\[0-9]{3}-[0-9]{4})"
re.Global = True
re.IgnoreCase = True
s = InputBox("Enter your phone number in the following Format (XXX) XXX-XXXX:")
If re.Test(s) Then
    MsgBox "Thank you!"
Else
    MsgBox "Sorry but that number is not in a valid format."
End If
```

这段代码很简单，其中的模式完成所有的真正的工作。根据不同的输入，您会得到两种输出消息中的一种，如图 9-23 和图 9-24 所示。

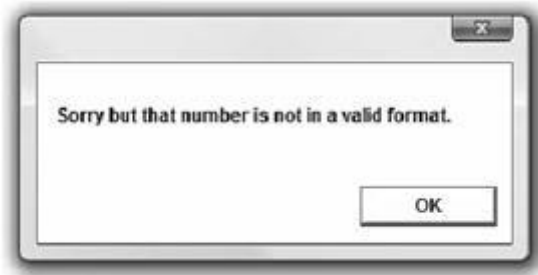


图 9-23



图 9-24

如果要将这个脚本应用到电话号码格式不同的国家，需要花点时间修改一下，但是并不困难。

9.4.2 分解 URI

这个例子用来将统一资源定位器(Universal Resource Indicator, URI)分解成多个部分。

比如下面这个 URI:

`www.wrox.com:80/misc-pages/support.shtml`

可以编写一个脚本将其分解成协议(比如 ftp、http 等)、域名地址、页面和路径。可以用下面的模式实现这个功能:

```
"(\w+):\ / \ / ([^ / :]+)(:\d*)?([ ^ # ]*)"
```

下面的代码完成了这个工作:

```
Dim re, s
Set re = New RegExp
re.Pattern = "(\w+):\ / \ / ([^ / :]+)(:\d*)?([ ^ # ]*)"
re.Global = True
re.IgnoreCase = True
s = "http://www.wrox.com:80/misc-pages/support.shtml"
MsgBox re.Replace(s, "$1")
MsgBox re.Replace(s, "$2")
MsgBox re.Replace(s, "$3")
MsgBox re.Replace(s, "$4")
```

9.4.3 检查 HTML 元素

检查 HTML 元素很简单,需要的只是一个正确的模式。下面这个就能检查元素开始和结束标签。

```
"<(.*?)>.*<\1>"
```

如何编写脚本取决于您想要实现什么功能。这个简单的脚本只是一个演示。可以改进这段代码专门用于检查某种元素,或是实现基本的错误检查。

```
Dim re, s
Set re = New RegExp
```

```

re.IgnoreCase = True
re.Pattern = "<(.*?)>.*<\ / \1>"
s = "<p>This is a paragraph</p>"
If re.Test(s) Then
    MsgBox "HTML element found."
Else
    MsgBox "No HTML element found."
End If

```

9.4.4 匹配空白

有时您可能真的需要匹配空白，也就是空行或是只有空白(空格和制表符)的行。下面的模式可以满足这个需求。

```
"^[ \t]*$"
```

说明如下：

^—— 匹配每一行的开头。

[\t]*—— 匹配 0 个或多个空格或制表符(\t)。

\$—— 匹配行结尾。

```

Dim re, s, colMatches, objMatch, sMsg
Set re = New RegExp
re.Global = True re.Pattern = "^[ \t]*$"
s = " "
Set colMatches = re.Execute(s)
sMsg = ""
For Each objMatch in colMatches
    sMsg = sMsg & "Blank line found at position " & _
objMatch.FirstIndex & " of the string."Next
MsgBox sMsg

```

9.4.5 匹配 HTML 注释标签

当您学习稍后的第 15 章，“Windows 脚本宿主”时，您将学会如何使用 VBScript 和 Windows 脚本宿主操作文件系统，这样您就能读取和修改文件。正则表达式的一个应用就是查找 HTML 文件中的注释标签。可以在将其发布到网络之前将其中的注释清除。

这个脚本可以检查 HTML 注释标签。

```
Dim re, s

Set re = New RegExp

re.Global = True

re.Pattern = "^.*<!--.*-->.*$"

s = " <title>A Title</title> <!-- a title tag -->"

If re.Test(s) Then

    MsgBox "HTML comment tags found."

Else

    MsgBox "No HTML comment tags found."

End If
```

对该模式稍作修改，并使用 Replace 方法就能将脚本中的注释清除。

```
Dim re, s

Set re = New RegExp

re.Global = True

re.Pattern = "(^.*)(<!--.*-->)(.*$)"

s = " <title>A Title</title> <!-- a title tag -->"

If re.Test(s) Then

    MsgBox "HTML comment tags found."

Else

    MsgBox "No HTML comment tags found."

End If

MsgBox re.Replace(s, "$1" & "$3")
```

9.5 小结

本章深入地讨论了正则表达式，以及如何在 **VBScript** 中使用正则表达式。本章演示了如何使用正则表达式实现高效灵活的文本字符串模式匹配。还演示了如何将正则表达式与脚本组合在一起实现自定义的查找和替换以及输入验证。

有人觉得学会正则表达式很困难，甚至在编写程序时放弃使用正则表达式而使用一些不太灵活的方法。但是，正则表达式给程序员提供了强大功能和足够的灵活性，您的努力很快就会得到回报！

第 15 章 Windows 脚本宿主

如果问一个程序员是否使用过 VBScript，得到的多数答案会是“是的，在 ASP 中用过”，或者“是的，在内网中用过”，或者“是的，在客户端脚本中用过”(前提是用户都使用 Internet Explorer，但今天的情形并非如此)。但是必须记住的是，在使用脚本的解决方案中，使用 VBScript 解决问题的背景不过如此而已。这是因为 VBScript 被设计为一个 ActiveX 脚本引擎，它可以被用于为 ActiveX 脚本宿主环境提供脚本运行的功能。

有两种最普通的宿主：

- 活动服务器网页(Active Server Pages, ASP)
- Internet Explorer

这两种宿主都为程序员提供了很多功能，但同时也具有一定的局限性。举例来说，Internet Explorer 不提供脚本与本地计算机(如文件系统、寄存器访问等)的交互能力，除非用户显式设置了这样的权限(这样做会带来极大的安全性风险，正因为这个原因，只有对可信站点和内网才这样做)。那么，在无法改动 VBScript 语言本身的情况下，应该从何处着手来扩展它的功能呢？这就是 Windows 脚本宿主(Windows Script Host, WSH)的着眼点。WSH 完全是一个兼容各种脚本语言的宿主接口，可以处理各种 ActiveX 脚本引擎。这意味着程序员如果使用了 WSH，他就可以使用 VBScript、JScript、PerlScript，或其他任何暴露 ActiveX 脚本接口的脚本语言。因此 WSH 宿主接口为 Windows 平台提供了一个强有力的，但是又非常易于使用的脚本编写平台，该平台既可以从 Windows GUI 中访问，也可以以命令行方式访问。

在这一章中，我们将学习关于 WSH 的如下内容：

- WSH 开发所需的工具
- WSH 的用途
- 执行 WSH 脚本的两种方法
- 使用.wsh 文件定制脚本行为
- WSH 对象模型
- .wsh 文件格式，用于创建更高级的脚本
- 使用 WSH 进行磁盘和网络管理

15.1 相关工具

在开始使用 WSH 之前，需要下面的工具：

- WSH 引擎
- 文本编辑器，如记事本(Notepad)(但是也可以随意使用任何针对程序编写而设计的编辑器——有很多可以选择的工具)
- 如果想使用除 VBScript 或 JScript 之外的某种脚本编写语言，就需要下载相应的 ActiveX 脚本引擎(例如可以从 ActiveState 下载 ActiveXPerl，网址是 www.activestate.com)

如果使用的操作系统是 Windows 98、Windows ME、安装了 Option Pack 4 的 Windows NT 4.0、Windows 2000、Windows XP 或 Windows Vista，那么系统中就已经有了 WSH(在 Win98 和 WinNT 中，WSH 1.0 作为一个可选的组件被提供)。不过，您可能希望确保使用最新的版本来运行本章中的脚本，那么就可以从 Microsoft Scripting Technologies 的网站，<http://msdn.microsoft.com/library/en-us/script56/html/d78573b7-fc96-410b-8fd0-3e84bd7d470f.asp> 下载。作者建议更新至最新的版本，WSH 5.6 和 Windows Script Engine 5.6 for JScript and VBScript(如果使用的是 Windows Vista，就可以使用 5.7 版本的脚本引擎)。

15.2 WSH 的概念

WSH 是一个 Windows 管理工具。WSH 创建了一个脚本运行的主环境，当脚本到达一台计算机时，WSH 扮演主人的角色。WSH 使脚本能够使用对象和服务，并提供脚本执行的准则。此外，WSH 还负责安全管理以及调用适当的脚本引擎。

因为 WSH 是与脚本语言无关的，所以它还提供了使用 JScript、Perl、Python、REXX，或其他 ActiveX 脚本编写语言(只有 VBScript 和 JScript 是 Microsoft 提供的，其他 ActiveX 脚本引擎是由第三方提供的)进行脚本编写的机制。WSH 提供了一些非常便于使用的工具，可以访问散布在网络中，运行各种风格的 Windows 操作系统的机器，从而提供了网络管理的能力。这种访问大部分是通过活动目录服务接口(Active Directory Service Interface, ADSI)和 Windows 管理规范(Windows Management Instrumentation, WMI)实现的。ADSI 提供了一套 COM 接口，可以用于多种目录服务，如轻量级目录访问协议(Lightweight Directory Access Protoc

ol, LDAP)、Windows NT 目录服务, 以及 Novell 的 Netware 和 NDS 服务。WMI 是 Microsoft 基于 Web 的企业管理(Web-Based Enterprise Management, WBEM)的实现, 这种标准的方法可以访问管理信息, 如指定的客户端上安装的应用程序、系统内存, 以及其他客户端信息。

通过开发使用 ADSI 和 WMI 的 WSH 脚本, 系统管理员可以开发脚本以便执行下面列出的这些以及其他更多任务:

- 访问并操作服务器
- 添加或删除用户以及修改密码
- 添加网络文件共享

目前的 WSH 版本是 5.7, 它是随 Windows Vista 一起发布的。该版本相对于先前的版本 (2.0)来说具有很多显著的变化。目前发布的 WSH 包含了大量深受程序员喜爱的功能:

- 支持文件包含
- 能够在同一脚本中使用多种语言
- 支持拖放功能
- 参数处理
- 可以远程运行脚本
- 增强对外部对象和类型库的访问
- 更强的调试功能
- 暂停脚本执行的机制(对于接收受控对象产生的事件来说非常有用)
- 标准的输入/输出以及标准的错误支持(只有在控制台模式下运行 `cscript.exe` 时才有用)
- 可以将新的进程作为对象来对待
- 对当前工作目录的访问
- 新改进的安全模型

随 Windows Vista 发布的版本 5.7 中包括针对版本 5.6 的 bug 和安全性修正。

WSH 1.0 的运行机制比较简单, 只是为 VBScript 的文件扩展名(.vbs)和 JScript 的文件扩展名(.js)关联相应的脚本宿主。这意味着如果双击一个脚本文件, 就会自动执行它。但是, 这有一个很重要的局限—— 关联模式不允许使用代码模块, 也不允许在一个 WSH 脚本项目中使用多种脚本语言。为了满足各种程序员的需要, Microsoft 在 WSH 2.0 中引入了一种新

的脚本文件类型(.wsf)，它利用一种 XML 语法提供前面所提到的很多新的功能。

这种新的机制在其他的标记中包含了<script>标记、<object>标记和<job>标记。在这一章后面的部分中，我们将介绍具体的运行机制。

文件扩展名.wsf 只在最后发布的 WSH 2.0 以及以后的版本中才有效。还在使用 WSH 2.0 beta 版本的开发人员仍然必须使用.ws 文件扩展名。作者建议使用最新的脚本引擎和 WSH。

15.3 脚本文件的类型

独立的脚本文件有一些不同的格式，每种格式都有自己的扩展名。表 15-1 中列出了一些常见的类型。最终选择的脚本类型将依赖于需求。多数小项目只需要使用一种文件类型，但在某些情况下可能需要将整个问题分割成若干个小的部分，为每一部分分别编写脚本，并且每一部分的脚本都使用最合适的语言进行编写。

表 15-1

扩 展 名	脚 本 类 型	描 述
.bat	MS-DOS 批处理文件	MS-DOS 操作系统的批处理文件
.asp	ASP 页面	活动服务器网页文件
.htm	HTML 文件	Web 页面
(续表)		
扩 展 名	脚 本 类 型	描 述
.html	HTML 文件	Web 页面
.js	JScript 文件	Windows 脚本
.vbs	VBScript 文件	Windows 脚本
.wsf	Windows 脚本宿主文件	一个 Windows 脚本的容器或项目文件。WSH 2.0 或更高版本支持
.wsh	Windows 脚本宿主文件	一个脚本文件的属性文件。WSH 1.0 或更高版本支持

这就是 Windows 脚本宿主文件(WSF 文件)的有用之处。WSF 文件可以包含其他的脚本文件作为其脚本的一部分。这意味着多个 WSF 文件可以引用创建并保存在同一地点的库和有用的函数。

15.4 使用 Windows 脚本宿主运行脚本

WSH 提供了两个用于执行脚本的接口，一个用于命令行，一个用于 Windows 环境。这两个接口各使用不同的宿主程序作为 VBScript 引擎：

- **cscript.exe**：用于在命令行中运行脚本
- **wscript.exe**：用于在 Windows 环境中运行脚本

之所以有两个宿主程序，是因为 **cscript.exe** 被设计为从控制台窗口启动(一般来说是 Windows 中的 MS-DOS 窗口)，而 **wscript.exe** 则用于直接与 Windows GUI 进行交互。这两者就功能来讲几乎没有区别。

15.4.1 命令行执行

执行脚本文件的命令行界面，**cscript.exe** 调用方法如下：

1. 打开 Run 对话框(按下窗口键+R)或某个命令窗口(在 Windows 9x 中，可以依次单击 Start|Programs|DOS Prompt；在 Windows NT 中，依次单击 Start|Programs|Command Prompt；在 Vista/XP 中，依次单击 Start|All Programs|Accessories|Command Prompt)。

注意，在使用 Windows Vista 时，打开 Command Prompt 需要系统管理员权限。最简单的方法是以系统管理员身份在开始菜单中用鼠标右键单击 Command Prompt 并选择 Run。

2. 执行脚本如下：

```
cscript c:\folderName\YourScriptName.vbs
```

如果直接在命令行中运行 **cscript.exe** 时不加参数，就只会得到使用帮助信息，如图 15-1 所示。

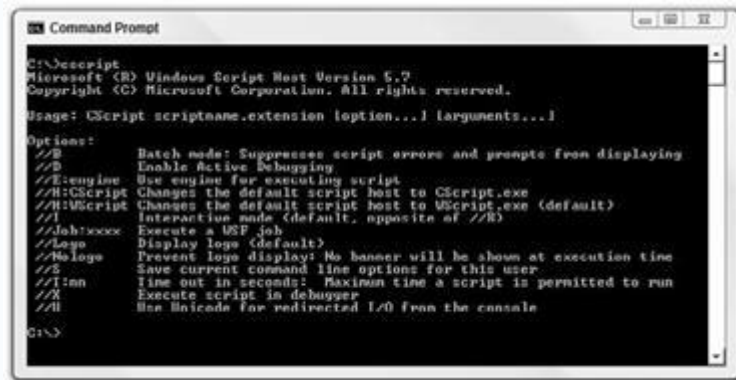


图 15-1

使用语法如下：

`cscript scriptname.extension [option...] [arguments...]`

cscript.exe 提供了如表 15-2 中所列的命令行选项，可以用来控制各种 WSH 环境设置。

表 15-2

//B	批处理模式 这种模式下不显示脚本错误和输出信息
//D	启用主动调试
//E:engine	使用引擎来执行脚本
//H:Cscript	将默认脚本宿主修改为 cscript.exe
//H:Wscript	将默认脚本宿主修改为 wscript.exe(此为默认选项)
//I	交互模式(此为默认选项，它与//B 是两种相反的模式)
//Job:xxxx	执行一个 WSF 作业
//Logo	显示 logo(此为默认选项)
//NoLogo	不显示 logo。执行时不显示 banner
//S	保存当前用户的当前命令行选项
//T:nn	超时时间(以秒为单位)。该时间为脚本在中止执行之前被允许的最长执行时间
//X	在调试器中执行脚本
//U	使用 Unicode 重定向来自控制台的 I/O

可以在命令行中以开关的形式增加这些选项以便使用它们。下面的示例在调试器中执行一个脚本。

`cscript MyScript.vbs //X`

15.4.2 在 Windows 环境中执行 WSH

执行脚本文件的 Windows GUI 界面是 `wscript.exe`，它允许以下列几种方式执行文件：

- 如果文件的类型注册为在 WSH 中执行，就可以直接在文件夹窗口或桌面上双击这些文件的图标来执行。
- 如果使用 Run 命令对话框，那么就只要输入脚本的完整路径和名称即可。

在 Run 命令对话框中，可以调用 `wscript.exe`。

```
wscript c:\folderName\YourScriptName.vbs
```

如果在命令行提示符下运行 `wscript.exe`，就不会有输出信息，而是会出现一个如图 15-2 所示的对话框。该对话框提供了最基本的定制选项。如果在 Run 命令对话框中运行 `wscript.exe`，也会出现相同的对话框。



图 15-2

单击 OK 按钮后，将不会有任何反应。在系统级定制脚本行为的唯一方法就是使用先前曾详细介绍的 `cscript` 选项。使用 `.wsh` 文件实现的每个脚本的独立定制将使用该对话框来完成，这在稍后会介绍。

那么这两种运行脚本的方法之间有什么区别呢？当调试一个出错脚本时，`cscript` 和 `wscript` 之间的主要区别实际上变得更明显了。这是因为，相对于 `wscript` 可能产生的没完没了

的弹出窗口来说，向一个控制台窗口发送错误消息显得更快更容易。在调试脚本时，推荐使用 `cscript`，而在打印调试输出结果时，最好使用 `WScript` 对象的 `Echo` 方法，因为调试时能产生大量的需要关注的错误信息。实际上，有时可能使调试者陷入循环，无法从错误消息中摆脱出来。

15.5 使用.WSH 文件运行脚本

有时也许不想或不需要在每次执行脚本时都修改设置，但又有可能需要控制单个的文件，这一点可以通过创建控制文件来实现。控制文件的扩展名为 `.wsh`，用于控制单个脚本的设置。`.wsh` 文件是短小的配置文件，它大体上遵循以前的 Windows 所使用的 `.ini` 文件格式(但这并不是说有程序员已经不再使用类似于 `.ini` 文件的配置文件了)。`.wsh` 文件非常便于定制脚本的启动——一个脚本可以使用多个不同的 `.wsh` 文件。

要创建一个 `.wsh` 文件，可以用鼠标右键单击一个与 WSH 关联的文件(一般带有 `.js`、`.vbs` 或 `.wsf` 的扩展名)，然后选择 `Properties`，在图 15-3 所示的对话框中选择 `Script` 标签页。



图 15-3

这个对话框可以用于修改超时时间的默认设置，也可以修改在命令行中运行脚本时，是否显示 `logo` 信息。一旦应用或接受了所做的修改，就会创建一个新的文件，文件名与所处理的脚本文件相同，只是扩展名为 `.wsh`。这个新的文件记录了定制的设置，其格式可以被宿

主引擎设置运行时选项。这里给出一个从脚本 `test.vbs` 创建的一个 `.wsh` 文件。

```
[ScriptFile]
```

```
Path=C:\test.vbs
```

```
[Options]
```

```
Timeout=25
```

```
DisplayLogo=0
```

要使用这些选项执行脚本，就应该运行 `test.wsh` 文件。

15.6 Windows 脚本宿主的内建对象

每个编程环境都提供了自己的对象模型，开发人员可以使用这些对象模型实现各种解决方案，WSH 也不例外。WSH 包含了一组核心对象，分别包含了属性和方法，可以用于访问网络中的其他计算机、导入外部的可脚本化对象以供应用程序使用，或者连接到 Windows 或 Windows Shell。

15.6.1 WScript 对象

WSH 对象模型的基础就是 WScript 对象。这个对象提供了使开发人员能够访问各种信息的属性和方法，如：

- 将要执行的脚本的名称和路径信息
- Microsoft 脚本引擎的版本
- 与外部对象的链接
- 与用户的交互
- 延迟或中断脚本执行的能力

1. WScript 的属性

WScript 对象具有下列属性：

- Arguments

- FullName
- Interactive
- Name
- Path
- ScriptFullName
- ScriptName
- StdErr
- StdIn
- StdOut
- Version

Argument

Argument 属性包含了 WshArguments 对象(一个参数集合)。从该集合中获取单个参数时,使用由 0 开始的索引。

```
Set objArgs = WScript.Arguments
```

```
For x = 0 to objArgs.Count - 1
```

```
WScript.Echo objArgs(x)
```

```
Next
```

FullName

FullName 属性是一个只读的字符串,它代表宿主可执行文件(cscript.exe 或 wscript.exe)的有效完整路径。下面的代码使用了 FullName 属性。

```
WScript.Echo WScript.FullName
```

这段代码产生如图 15-4 的输出。

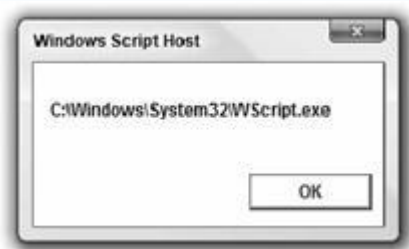


图 15-4

Interactive

Interactive 属性设置脚本的模式，或识别脚本的模式。使用该属性会返回一个布尔值。有两种可用的模式：批处理模式和交互模式。

在交互模式下(默认情况)，脚本可以与用户进行交互。可以向 WSH 输入信息，WSH 也可以输出信息，信息被显示在对话框中，等待用户提供反馈。在批处理模式中，不支持用户交互，不支持 WSH 的输入和输出。

可以使用 WSH 命令行开关脚本模式//I(交互模式)和//B(批处理模式)来设置脚本模式。

Name

Name 属性返回 WScript 对象(宿主可执行文件)的名称，它是一个只读的字符串。下面的代码使用了 Name 属性：

WScript.Echo WScript.Name

这段代码产生如图 15-4 的输出。



图 15-5

Path

Path 属性返回包含宿主可执行文件(cscript.exe 或 wscript.exe)的目录名。该属性返回一个只读的字符串。

下面的 VBScript 代码将显示可执行文件所在的目录名：

WScript.Echo WScript.Path

ScriptFullName

ScriptFullName 属性返回当前正在运行的脚本的完整路径。该属性返回一个只读的字符串。

ScriptName

ScriptName 属性返回当前正在运行的脚本的文件名。该属性返回一个只读的字符串。下面的代码显示了正在运行的脚本的名称，如图 15-6 所示：

```
WScript.Echo WScript.ScriptName
```



图 15-6

StdErr

StdErr 属性开放当前脚本的只写的错误输出流。该属性返回一个标识标准错误流的对象。只有使用 **cscript.exe** 时才能访问 **StdIn**、**StdOut** 和 **StdErr** 流。使用 **wscript.exe** 时如果想访问这些流就会产生一个错误。

StdIn

StdIn 属性开放当前脚本的只读的输入流。该属性返回一个标识标准错误流的对象。只有使用 **cscript.exe** 时才能访问 **StdIn**、**StdOut** 和 **StdErr** 流。使用 **wscript.exe** 时如果想访问这些流就会产生一个错误。

StdOut

StdOut 属性开放当前脚本的只写的错误输出流。该属性返回一个标识标准错误流的对象。只有使用 **cscript.exe** 时才能访问 **StdIn**、**StdOut** 和 **StdErr** 流。使用 **wscript.exe** 时如果想访问这些流就会产生一个错误。

下面的示例使用了这三种类型的内建流打印匹配某一特定扩展名的所有文件列表。这是通过将 DOS 的 **dir** 命令结果使用管道重定向到过滤器脚本来实现的，其中将扩展名字符串作

为参数。

```
' Usage: dir | cscript filter.vbs ext
'   ext: file extension to match
'

Dim streamOut, streamIn, streamErr
Set streamOut = WScript.Stdout
Set streamIn = WScript.StdIn
Set streamErr = WScript.Stderr

Dim strExt, strLineIn
Dim intMatch

strExt = WScript.Arguments(0)
intMatch = 0

Do While Not streamIn.AtEndOfStream
    strLineIn = streamIn.ReadLine

    If 0 = StrComp(strExt, Right(strLineIn, Len(strExt)), _
        vbTextCompare) Then
        streamOut.WriteLine strLineIn
        intMatch = intMatch + 1
    End If
Loop

If 0 = intMatch Then
    streamErr.WriteLine "No files of type '" & strExt & "' found"
End If
```

因为这个示例中使用 `StdIn`、`StdOut` 和 `StdErr` 进行消息的处理，所以不仅能将匹配的文件打印在屏幕上，也能够将输出结果发送到一个文本文件或其他有重定向或附加管道功能的应用程序中。例如，可以使用下面的命令创建一个文件，包含整个目录树，包括子目录中的

所有.vbs 文件:

```
C:\wsh>dir /s | cscript filter.vbs vbs >> vbsfiles.txt
```

Version

该属性返回 WSH 的版本。下面的代码显示当前 WSH 的版本，如图 15-7 所示:

```
WScript.Echo WScript.Version
```



图 15-7

2 WScript 的方法

WScript 对象具有下列方法:

- CreateObject
- ConnectObject
- DisconnectObject
- Echo
- GetObject
- Quit
- Sleep

CreateObject

WScript 对象的这个方法用于创建一个 COM 对象。

```
object.CreateObject(strProgID[,strPrefix])
```

- object: WScript 对象。
- strProgID: 值为字符串，表示想要创建的对象程序标识符(ProgID)。

- **strPrefix:** 可选。值为字符串，表示函数前缀。

使用 **CreateObject** 方法和 **strPrefix** 参数创建的对象是已连接对象。对象被创建后，其输出接口与脚本文件相连接。每个事件函数都由这个前缀加上事件的名称来命名。

如果创建对象时没有使用 **strPrefix** 参数，也可以使用 **ConnectObject** 方法同步对象的事件。只要对象产生一个事件，**WSH** 就会调用一个子程序，其名称是 **strPrefix** 加上事件的名称。

下面的代码使用 **CreateObject** 方法创建一个 **WshNetwork** 对象：

```
Set WshNetwork = WScript.CreateObject("WScript.Network")
```

ConnectObject

该方法将对象的事件源连接到具有给定前缀的函数。

```
object.ConnectObject(strObject, strPrefix)
```

- **object:** **WScript** 对象。
- **strObject:** 必需的。表示想要连接的对象名称的字符串。
- **strPrefix:** 必需的。表示函数前缀的字符串。

同步一个对象的事件时，已连接对象是很有用的。对象被创建后，**ConnectObject** 方法将对象的输出接口连接到脚本文件。事件函数的名称就是这个前缀加上事件的名称。

```
WScript.ConnectObject RemoteScript, "remote_"
```

DisconnectObject

该方法用于断开已连接对象的事件源的连接。

```
object.DisconnectObject(obj)
```

- **object:** **WScript** 对象。
- **obj:** 表示想要断开连接的对象名称的字符串。

断开一个对象的连接意味着 **WSH** 无法再响应它的事件。但是，有一点很重要，那就是断开连接后对象仍然能够产生事件。还要注意的，如果指定的对象尚未连接，**DisconnectObject** 方法就不会做任何事。

WScript.DisconnectObject RemoteScript

Echo

该方法输出一个消息框或一个命令控制台窗口。

object.Echo [Arg1] [,Arg2] [,Arg3] ...

- object: WScript 对象。
- Arg1、Arg2、Arg3、.....: 可选。表示要显示项目的列表。

根据当前使用的 WSH 引擎的不同，Echo 方法输出的类型也有所变化(参见表 15-3)。

表 15-3

WSH 引擎	文 本 输 出
Wscript.exe	图形消息框
Cscript.exe	命令控制台窗口

图 15-8 和图 15-9 显示了这两种输出的示例。

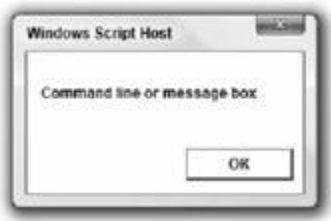


图 15-8



图 15-9

GetObject

GetObject 方法根据指定的 ProgID 获取某个已存在对象，或从文件创建一个新的对象。

```
object.GetObject(strPathname [,strProgID], [strPrefix])
```

- **object:** WScript 对象。
- **strPathname:** 包含将对象保存到磁盘的文件所使用的完整的有效路径名。
- **strProgID:** 可选。值为字符串，表示想要创建的对象程序标识符(ProgID)。
- **strPrefix:** 可选。进行对象事件同步时有用。如果提供了 **strPrefix** 参数，WSH 就会在创建对象之后将对象的输出接口连接到脚本文件。

如果内存中存在对象的一个实例，或者想从文件创建一个对象，都需要使用 **GetObject** 方法。**GetObject** 方法适用于所有 COM 类，与创建对象使用的脚本语言无关。

如果内存不存在对象的实例，也不想从文件创建对象，就可以使用 **CreateObject** 方法。

```
Dim MyObject As Object
```

```
Set MyObject = GetObject("C:\DRAWINGS\SCHEMA.DRW")
```

```
MyApp = MyObject.Application
```

```
Quit
```

该方法强制脚本在任意时刻立即停止执行。

```
object.Quit([intErrorCode])
```

- **object:** WScript 对象。
- **intErrorCode:** 可选。返回一个整数值，作为进程的返回码。如果忽略 **intErrorCode** 参数，就不会有返回值。

Quit 方法可以用于返回一个可选的错误代码。如果 **Quit** 方法是脚本中的最后一条命令(并且不需要返回一个非零值)，就可以不使用任何参数，这样脚本就会正常退出。

```
WScript.Quit 1
```

```
' This line of code is not executed.
```

```
MsgBox "This message will never be shown!"
```

下面给出了 **Quit** 方法实际运行的示例。

```
If Err.Number <> 0 Then
```

```
        WScript.Quit 1 ' some failure indicator Else  
        WScript.Quit 0 ' success  
End If
```

```
WScript.Quit 1
```

```
    Sleep
```

该方法将脚本的执行挂起一段时间，然后接着执行。

```
object.Sleep(intTime)
```

- **object:** WScript 对象。
- **intTime:** 这是一个整数值，表示希望脚本进程保持非活跃状态的时间间隔(以毫秒为单位)。

使用这个方法时，运行脚本的线程被挂起，占用的 CPU 被释放。当间隔时间到，就会继续恢复执行。要想被事件所触发，脚本必须持续活跃，因为已经结束执行的脚本是肯定不能检测到任何事件的。脚本所处理的事件在脚本休眠时仍然会被执行。

向 Sleep 方法传递 0 或-1 作为参数，就不会导致脚本不确定性地挂起。

```
<package>
```

```
    <job id="vbs">
```

```
        <script language="VBScript">
```

```
            set WshShell = WScript.CreateObject("WScript.Shell")
```

```
            WshShell.Run "calc"
```

```
            WScript.Sleep 100
```

```
            WshShell.AppActivate "Calculator"
```

```
            WScript.Sleep 100
```

```
            WshShell.SendKeys "1{+}"
```

```
            WScript.Sleep 500
```

```
            WshShell.SendKeys "2"
```

```
            WScript.Sleep 500
```

```
WshShell.SendKeys "~"

WScript.Sleep 500

WshShell.SendKeys "*9"

WScript.Sleep 500

WshShell.SendKeys "~"

WScript.Sleep 2500

</script>

</job>

</package>
```

15.6.2 WshArguments 对象

编程过程中, 参数的使用是一种非常有益的机制, 可以给脚本提供输入以支持它的工作。如果考虑在 DOS 提示符下工作, 那么多数命令行可执行文件都使用参数来确定要做的事情。例如, 浏览一个目录树:

```
c:\>cd wsh
```

在这个例子中, `cd` 是一个 DOS 命令的名称(用于切换目录), 而 `wsh` 是要激活的目录的名称——它是作为参数被传递给 `cd` 的。

创建使用参数的脚本是编写可重用代码的一个良好开端。创建设计用于在命令行中执行的脚本的开发人员会立竿见影地感受到使用 `Arguments` 属性带来的好处。不过, 在 `WSH` 中, 还有更好的理由来使用这个对象, 因为这就是拖放功能得以实现的原因。

使用这个对象的最后一个好处就是它允许开发人员在其他脚本中重用脚本代码, 只要以命令行运行的方式运行想重用的脚本, 并给它传递可能在运行时需要的参数即可。

1. 访问 WshArgument 对象

它是通过使用 `WScript.Arguments` 属性来实现的。

```
Set objArgs = WScript.Arguments
```


2 WshArgument 对象的属性

WshArgument 对象是一个由 WScript 对象的 Arguments 属性返回的集合(WScript.Arguments)。

访问命令行参数集合有下列三种方法：

- 使用 WshArguments 对象访问整个参数集合。
- 使用 WshNamed 对象访问有名字的参数。
- 使用 WshUnnamed 对象访问没有名字的参数。

下面的例子是一个遍历 WshArguments 集合的简单循环，依次显示每个元素：

```
Set objArgs = WScript.Arguments
```

```
For x = 0 to objArgs.Count - 1
```

```
    WScript.Echo objArgs(x)
```

```
Next
```

有趣的是，这段代码既适用于 cscript.exe，也适用于 wscript.exe，可以使用示例 echoargs.vbs 来尝试这一点。在命令行中运行时，传递一些参数：

```
c:\vbs\echoargs Hello, World!
```

图 15-10 显示了命令行输出。

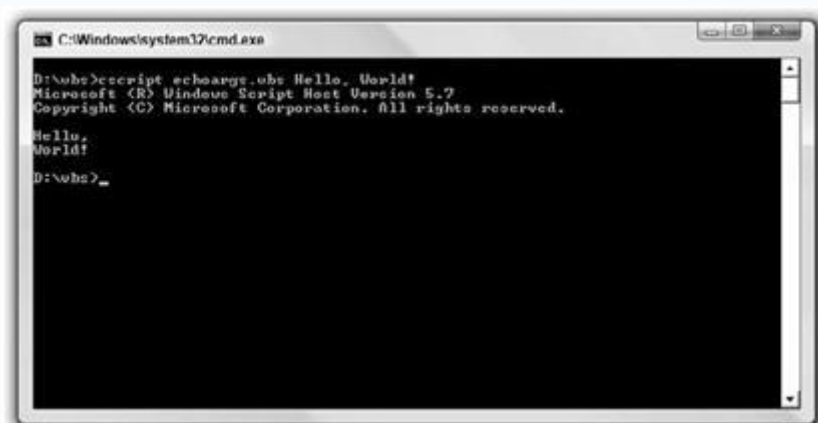


图 15-10

现在可以尝试拖动一两个文件，并将它们放到 echoargs.vbs 上。图 15-11 显示了这样做

输出的结果。

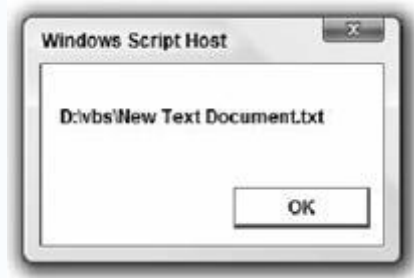


图 15-11

15.6.3 WshShell 对象

Windows 脚本宿主提供了一种便捷的方式，可以用于获取系统环境变量的访问、创建快捷方式、访问 Windows 的特殊文件夹，如 Windows Desktop，以及添加或删除注册表条目。还可以使用 Shell 对象的功能创建更多的定制对话框以进行用户交互。

1 访问 WshShell 对象

要使用下面这一节中将要列出的属性，程序员必须创建一个 WScript.Shell 对象的实例。这样以后对 WshShell 对象的引用实际就是对新创建实例的引用。

```
Set WshShell= WScript.CreateObject( "WScript.Shell" )
```

2 WshShell 对象的属性

WshShell 对象有三个属性：

- CurrentDirectory
- Environment
- SpecialFolders

CurrentDirectory

该属性获取或更改当前活动目录。

```
object.CurrentDirectory
```

- object: WshShell 对象。

CurrentDirectory 属性返回一个字符串，其中包含当前活跃进程的当前工作目录的完整有效路径名。

Dim WshShell

Set WshShell = WScript.CreateObject("WScript.Shell")

WScript.Echo WshShell.CurrentDirectory

Environment

该属性返回 WshEnvironment 对象(一个环境变量集合)。

object.Environment ([strType])

- object: WshShell 对象。
- strType: 可选。指定环境变量的位置。

Environment 属性包含 WshEnvironment 对象(一个环境变量集合)。如果指定了 strType，就指定了环境变量所在的级别的值：

- System
- User
- Volatile
- Process

如果没有指定 strType，Environment 属性就会根据不同的操作系统返回不同的环境变量类型(表 15-4 所示)。

表 15-4

环境变量类型	操 作 系 统
System	Microsoft Windows NT/2000/XP/Vista
Process	Windows 95/98/Me

对于 Windows 95/98/Me，只允许一种 strType: Process。其他类型不允许在脚本中使用。

表 15-5 列出了 Windows 操作系统提供的一些变量。

表 15-5

名 称	描 述	System	User	Process(NT/2000/XP/Vista)	Process(95/98/Me)
-----	-----	--------	------	---------------------------	-------------------

NUMBER_OF_PROCESSORS	机器中运行的处理器个数	x	-	X	-
PROCESSOR_ARCHITECTURE	用户工作站的处理器类型	x	-	X	-
PROCESSOR_IDENTIFIER	用户工作站的处理器 ID	x	-	X	-
PROCESSOR_LEVEL	用户工作站的处理器等级	x	-	X	-
PROCESSOR_REVISION	用户工作站的处理器版本	x	-	X	-
OS	用户工作站的操作系统	x	-	X	-
COMSPEC	用户工作站的操作系统	-	-	X	x
HOMEDRIVE	主逻辑驱动器(通常是 C 盘)	-	-	X	-
HOMEPath	用户的默认目录	-	-	X	-
Path	Path 环境变量	x	x	X	x

(续表)

名 称	描 述	System	User	Process(NT/2000/XP/Vista)	Process(95/98/Me)
PATHEXT	可执行文件的扩展名(一般是 .com、.exe、.bat 或 .cmd)	x	-	X	-
PROMPT	命令提示符(一般是 \$P\$G)	-	-	X	x
SYSTEMDRIVE	系统目录所在驱动器(一般是 c:\)	-	-	X	-
SYSTEMROOT	系统目录(例如 c:\winnt)。与 WINDIR 相同	-	-	X	-
WINDIR	系统目录(例如 c:\winnt)。与	x	-	X	x

	SYSTEMROOT 相同				
TEMP	存储临时文件的目录(例如 c:\temp)	-	x	X	x
TMP	存储临时文件的目录(例如 c:\temp)	-	x	X	x

要注意的是，脚本有可能访问其他应用程序所设置的环境变量，而且上面列出的环境变量都不能是 **Volatile** 类型的。

下面是一个关于在代码中如何使用上面列出的这些变量的示例。该示例返回系统中存在的处理器个数。

```
Set WshShell = WScript.CreateObject("WScript.Shell")
Set WshSysEnv = WshShell.Environment("SYSTEM")
WScript.Echo WshSysEnv("NUMBER_OF_PROCESSORS")
```

SpecialFolders

该属性返回一个 **SpecialFolders** 对象(一个特殊文件夹集合)。

```
object.SpecialFolders(objWshSpecialFolders)
```

- **object**: WshShell 对象。
- **objWshSpecialFolders**: 特殊文件夹的名称。

WshSpecialFolders 对象是一个集合，它包含所有的 Windows 特殊文件夹，其中包括 **Desktop** 文件夹、**Start Menu** 文件夹和 **Documents/My Documents** 文件夹(注意，在 Windows Vista 中去掉了前缀“my”)。

特殊文件夹的名称用于索引集合以获取想要的具体文件夹。如果被请求的文件夹(**strFolderName**)不可用，**SpecialFolder** 属性将返回一个空字符串。例如，Windows 95 没有 **AllUsersDesktop** 这个文件夹，所有如果 **strFolderName** 是 **AllUsersDesktop** 的话，就会返回一个空字符串。

下面是可用的特殊文件夹：

- **AllUsersDesktop**
- **AllUsersStartMenu**
- **AllUsersPrograms**
- **AllUsersStartup**
- **Desktop**

- Favorites
- Fonts
- MyDocuments
- NetHood
- PrintHood
- Programs
- Recent
- SendTo
- StartMenu
- Startup
- Templates

下面的代码用于获取 Start Menu 文件夹并在 strDesktop 变量中保存路径以备后面的使用。

```
strDesktop = WshShell.SpecialFolders("StartMenu")
```

3. WshShell 对象的方法

WshShell 对象有 11 个方法。所有这些方法都与操作系统 shell 相关，可以用于控制 Windows 注册表，也可以创建弹出式消息框和快捷方式以及激活和控制正在运行的应用程序：

- AppActivate
- CreateShortcut
- ExpandEnvironmentStrings
- LogEvent
- Popup
- RegDelete
- RegRead
- RegWrite
- Run
- SendKeys
- Exec

AppActivate

这个方法可以用于激活一个指定的已经打开的应用程序窗口。

object.AppActivate title

- object: WshShell 对象。

- **title:** 指定要激活的应用程序。这可以是一个包含应用程序标题的字符串(就像出现在标题栏中的那样)或应用程序的进程 ID。

AppActivate 方法返回一个 **Boolean** 类型的值, 标志着过程调用是否成功。这个方法用于将焦点切换至指定名称的应用程序或窗口。方法的调用不影响应用程序的最大化或最小化状态。当进行了切换焦点(或关闭窗口)的动作之后, 焦点就会从已经获得焦点的应用程序上移走。

为了确定要激活的应用程序, 指定的标题名称会与每个正在运行的应用程序的标题字符串做比较。如果没有准确的匹配, 那么具有以 **title** 所指定的值开始的标题字符串的应用程序就会被激活。如果还是没有能找到合适的应用程序, 那么具有以 **title** 所指定的值结束的标题字符串的应用程序就会被激活。如果存在多个名为 **title** 所指定的值的应用程序实例, 就会任意激活其中的一个。但是, 无法控制具体选择哪一个。

```
<package>
```

```
    <job id="vbs">
```

```
        <script language="VBScript">
```

```
            set WshShell = WScript.CreateObject("WScript.Shell")
```

```
            WshShell.Run "calc"
```

```
            WScript.Sleep 100
```

```
            WshShell.AppActivate "Calculator"
```

```
            WScript.Sleep 100
```

```
            WshShell.SendKeys "1{+}"
```

```
            WScript.Sleep 500
```

```
            WshShell.SendKeys "2"
```

```
            WScript.Sleep 500
```

```
            WshShell.SendKeys "~"
```

```
            WScript.Sleep 500
```

```
            WshShell.SendKeys "*3"
```

```
            WScript.Sleep 500
```

```
WshShell.SendKeys "~"

WScript.Sleep 2500

</script>

</job>
```

</package>

CreateShortcut

该方法可以用于创建一个新的快捷方式，或打开一个已有的快捷方式。

object.CreateShortcut(strPathname)

- object: WshShell 对象。
- strPathname: 一个表示要创建的快捷方式的路径名。

CreateShortcut 方法返回一个 WshShortcut 对象或一个 WshURLShortcut 对象。调用 CreateShortcut 方法不会导致快捷方式的真正创建，而是将快捷方式对象以及对其所做的修改保存在内存中，直到使用 Save 方法将其保存至磁盘。要真正创建一个快捷方式，必须完成下列三个步骤：

- 创建 WshShortcut 对象的一个实例。
- 初始化其属性。
- 使用 Save 方法将其保存到磁盘。

容易出问题的是将参数放到快捷方式对象的 TargetPath 属性中，这通常是无用的。快捷方式的所有参数都必须放在 Arguments 属性中。

<package>

```
<job id="vbs">

  <script language="VBScript">

    set WshShell = WScript.CreateObject("WScript.Shell")

    strDesktop = WshShell.SpecialFolders("Desktop")

    set oShellLink = WshShell.CreateShortcut(strDesktop & _
"\Shortcut Script.lnk")

    oShellLink.TargetPath = WScript.ScriptFullName
```



```
oShellLink.WindowStyle = 1

oShellLink.Hotkey = "CTRL+SHIFT+N"

oShellLink.IconLocation = "notepad.exe, 0"

oShellLink.Description = "Shortcut to Notepad"

oShellLink.WorkingDirectory = strDesktop

oShellLink.Save

</script>

</job>

</package>
```

ExpandEnvironmentStrings

该方法返回环境变量的扩充值。

object.ExpandEnvironmentStrings(strString)

- **object:** WshShell 对象。
- **strString:** 一个字符串值，表示想要扩充的环境变量的名称。

这个方法只对 **PROCESS** 环境空间内定义的环境变量进行扩充。环境变量的名称必须以“%”包围，并且是大小写不敏感的。

```
set WshShell = WScript.CreateObject("WScript.Shell")
```

```
WScript.Echo "The path to WinDir is " _
```

```
& WshShell.ExpandEnvironmentStrings("%WinDir%")
```

LogEvent

LogEvent 方法向日志文件中添加一个事件条目。

object.LogEvent(intType, strMessage [,strTarget])

- **object:** WshShell 对象。
- **intType:** 表示事件类型的整数值。
- **strMessage:** 包含日志条目文本的字符串值。

- **strTarget:** 可选。一个字符串值，表示事件日志所存储的计算机系统名称(默认的是本地计算机系统)。只适用于 Windows NT/2000/XP/Vista。

这个方法用于返回一个 Boolean 值(如果事件被成功记录，就返回 True，否则返回 False)。

在 Windows NT/2000/XP/Vista 中，事件被记录在 Windows NT Event Log 中。在 Windows 9x/Me 中，事件被记录在 WSH.log(位于 Windows 目录)中。

有 6 种事件类型，如表 15-6 所示。

表 15-6

类 型	值
0	SUCCESS
1	ERROR
2	WARNING
4	INFORMATION
8	AUDIT_SUCCESS
16	AUDIT_FAILURE

下面的代码显示了 LogEvent 的运行，它基于事件的成功与否对其进行记录。

```
Set WshShell = WScript.CreateObject("WScript.Shell")
```

```
'assume that rS contains a return code
```

```
'from another part of the code
```

```
if rS then
```

```
    WshShell.LogEvent 0, "Script Completed Successfully"
```

```
else
```

```
    WshShell.LogEvent 1, "Script failed"
```

```
end if
```

```
    Popup
```

该方法用于在弹出式消息框中显示文本。

```
intButton = object.Popup(strText,[nSecondsToWait],[strTitle],[nType])
```

- **object:** WshShell 对象。
- **strText:** 一个字符串值，包含了想要显示在弹出式消息框中的文本。

- **nSecondsToWait:** 可选。一个数值，表示想要弹出式消息框显示的最长时间(以秒为单位)。如果 **nSecondToWait** 等于 0(默认值)，弹出式消息框就一直可见，直到被关闭。如果 **nSecondToWait** 大于 0，那么弹出式消息框就在 **nSecondToWait** 秒后关闭。
- **strTitle:** 可选。一个字符串值，包含了想要出现在弹出式消息框标题栏中的文本。如果没有提供 **strTitle** 这个参数，弹出式消息框的标题就会被设置为默认的字符串“Windows Script Host”。
- **nType:** 可选。一个数值，表示想在弹出式消息框中出现的按钮和图标的类型。这些决定了消息框的使用场合和用途。**nType** 的功能与 Microsoft Win32 应用程序编程接口中的 **MessageBox** 函数相同。下面的表格显示了各种值和它们的含义。可以合并列表中不同的值获得各种不同的结果。
- **IntButton:** 一个整数值，表示关闭消息框时单击的按钮的序号，这个值是由 **Popup** 方法返回的。

无论使用那种宿主可执行文件(wscript.exe 或 cscript.exe)运行脚本，都可以使用 **Popup** 方法显示一个消息框。

要以诸如 **HeBrew** 或 **Arabic** 这样的 RTL 语言的格式(从右至左)正确显示文本，可以在 **nType** 参数之前加上 **h00100000**(十进制的 1048576)。

按钮类型

见表 15-7。

表 15-7

值	描 述
0	显示 OK 按钮
1	显示 OK 和 Cancel 按钮
2	显示 Abort、Retry 和 Ignore 按钮
3	显示 Yes、No 和 Cancel 按钮
4	显示 Yes 和 No 按钮
5	显示 Retry 和 Cancel 按钮

图标类型

见表 15-8。

表 15-8

值	描 述
16	显示终止(Stop Mark)图标
32	显示问号(Question Mark)图标
48	显示叹号(Exclamation Mark)图标
64	显示提示信息(Information Mark)图标

IntButton 的返回值代表已单击的按钮编号。如果在 nSecondsToWait 秒之前没有单击任何按钮，inButton 就被设置为-1。

表 15-9 列出了关闭消息框时所单击的按钮的编号。

表 15-9

值	描 述
1	OK 按钮
2	Cancel 按钮
3	Abort 按钮
4	Retry 按钮
5	Ignore 按钮
6	Yes 按钮
7	No 按钮

下面的代码显示了各种消息框按钮和图标的使用：

```
Dim WshShell, BtnCode

Set WshShell = WScript.CreateObject("WScript.Shell")

BtnCode = WshShell.Popup("Do you like this code?", 7, "Quick survey:", 4 + 32)

Select Case BtnCode
    case 6    WScript.Echo "Glad to hear it - Thanks!"
    case 7    WScript.Echo "I'm sorry you didn't like it."
    case -1   WScript.Echo "Hellllloooooooo?"
End Select

RegDelete

    该方法从注册表中删除一个键或其键值。

object.RegDelete(strName)
```

- **object:** WshShell 对象。
- **strName:** 一个字符串值，表示要删除的注册表键或键值的名称。

如果 **strName** 以反斜杠结尾，就可以指定键名，如果不使用反斜杠，就可以指定键值名。

完整的有效键名和键值名是以一个根键的名称作为前缀的。使用 **RegDelete** 方法时，也可以使用根键名的缩写形式。

表 15-10 列出了 5 种可能使用的根键。

表 15-10

根 键 名	缩 写
HKEY_CURRENT_USER	HKCU
HKEY_LOCAL_MACHINE	HKLM
HKEY_CLASSES_ROOT	HKCR
HKEY_USERS	HKEY_USERS
HKEY_CURRENT_CONFIG	HKEY_CURRENT_CONFIG

下面的脚本将创建、读取并删除 Windows 注册表键。高亮的部分是删除键的脚本。

```
Dim WshShell, bKey
Set WshShell = WScript.CreateObject("WScript.Shell")

WshShell.RegWrite "HKCU\Software\WROX\VBScript\", 1, "REG_BINARY"
WshShell.RegWrite "HKCU\Software\WROX\VBScript\ProgRef","VBS_is_great","REG_SZ"

bKey = WshShell.RegRead("HKCU\Software\WROX\VBScript\)
WScript.Echo WshShell.RegRead("HKCU\Software\WROX\VBScript\ProgRef")

WshShell.RegDelete "HKCU\Software\WROX\VBScript\ProgRef"
WshShell.RegDelete "HKCU\Software\WROX\VBScript\"
WshShell.RegDelete "HKCU\Software\WROX\"
```

在修改注册表设置时，需要特别的小心。对注册表进行不当的修改可能会导致系统变得不稳定，甚至使其变得完全不可用。如果不了解注册表的内部工作机制，那么强烈建议在实际操作之前就此问题阅读一些相关的资料。

RegRead

该方法返回注册表中一个键或键值的名称。

object.RegRead(strName)

- object: WshShell 对象。
- strName: 一个字符串值，表示想获得的键或键值的名称。

RegRead 方法返回的值有 5 种类型(如表 15-11 所示)。

表 15-11

类 型	描 述	形 式
REG_SZ	字符串	字符串
REG_DWORD	数字	整数
REG_BINARY	二进制值	整数构成的 VBAArray
REG_EXPAND_SZ	可扩充字符串(例如%windir%\notepad.exe)	字符串
REG_MULTI_SZ	字符串数组	字符串构成的 VBAArray

如果 strName 以反斜杠结尾，就可以指定键名，如果不使用反斜杠，就可以指定键值名。
一个键值包含三部分：

- 名称(Name)
- 数据类型(Data Type)
- 值(Value)

如果指定了键名(相对于键值名来说)，RegRead 就会返回默认的值。所以要读取一个键的默认值，只要指定键的名称即可。

完整的有效键名和键值名是以一个根键的名称作为起始的。使用 RegRead 方法时，也可以使用根键名的缩写形式。表 15-12 列出了 5 种可能使用的根键。

表 15-12

根 键 名	缩 写
HKEY_CURRENT_USER	HKCU
HKEY_LOCAL_MACHINE	HKLM
HKEY_CLASSES_ROOT	HKCR
HKEY_USERS	HKEY_USERS
HKEY_CURRENT_CONFIG	HKEY_CURRENT_CONFIG

下面的脚本将创建、读取并删除 Windows 注册表键。高亮的部分是读取键的脚本。

```
Dim WshShell, bKey
Set WshShell = WScript.CreateObject("WScript.Shell")
```

```
WshShell.RegWrite "HKCU\Software\WROX\VBScript\", 1, "REG_BINARY"

WshShell.RegWrite "HKCU\Software\WROX\VBScript\ProgRef","VBS_is_great","REG_SZ"

bKey = WshShell.RegRead("HKCU\Software\WROX\VBScript\")

WScript.Echo WshShell.RegRead("HKCU\Software\WROX\VBScript\ProgRef")

WshShell.RegDelete "HKCU\Software\WROX\VBScript\ProgRef"

WshShell.RegDelete "HKCU\Software\WROX\VBScript\"

WshShell.RegDelete "HKCU\Software\WROX\"
```

RegWrtie

该方法可以创建一个新键，给现有键添加另一个键值名(并指定一个值)，或修改现有键值名的值。

```
object.RegWrite(strName, anyValue [,strType])
```

- **object:** WshShell 对象。
- **strName:** 一个字符串值，表示想创建、添加或修改的键名、键值名或键值。
- **anyValue:** 想要创建的新键的名称，或想添加到现有键的键值名，或想指定给某个现有键值名的新键值。
- **strType:** 可选。一个表示键值数据类型的字符串。

如果 strName 以反斜杠结尾，就可以指定键名，如果不使用反斜杠，就可以指定键值名。

RegWrite 方法自动将参数 anyValue 转换成一个字符串或一个整数，而由 strType 的值决定其数据类型(是字符串还是整数)。表 15-13 列出了 strType 方法的可用选项。

表 15-13

转换至	strType
字符串	REG_SZ
字符串	REG_EXPAND_SZ
整数	REG_DWORD
整数	REG_BINARY

RegWrite 方法不支持 REG_MULTI_SZ 类型。

RegWrite 最多写入一个 DWORD 或 REG_BINARY 值，该方法不支持更大的值。完整的有效键名和键值名是以一个根键的名称作为起始的。使用 RegWrite 方法时，也可以使用根键

名的缩写形式。表 15-14 列出了 5 种可能使用的根键。

表 15-14

根 键 名	缩 写
HKEY_CURRENT_USER	HKCU
HKEY_LOCAL_MACHINE	HKLM
HKEY_CLASSES_ROOT	HKCR
HKEY_USERS	HKEY_USERS
HKEY_CURRENT_CONFIG	HKEY_CURRENT_CONFIG

strType 的四种可能被指定的数据类型如表 15-15 所示。

表 15-15

类 型	描 述	形 式
REG_SZ	字符串	字符串
REG_DWORD	数字	整数
REG_BINARY	二进制值	整数构成的 VBAArray
REG_EXPAND_SZ	可扩充字符串(例如%windir%\notepad.exe)	字符串

下面的代码显示了如何访问和修改 Windows 注册表：

```
Dim WshShell, bKey
Set WshShell = WScript.CreateObject("WScript.Shell")
WshShell.RegWrite "HKCU\Software\WROX\VBScript\", 1, "REG_BINARY"
WshShell.RegWrite "HKCU\Software\WROX\VBScript\ProgRef","VBS_is_great","REG_SZ"
bKey = WshShell.RegRead("HKCU\Software\WROX\VBScript\")
WScript.Echo WshShell.RegRead("HKCU\Software\WROX\VBScript\ProgRef")
WshShell.RegDelete "HKCU\Software\WROX\VBScript\ProgRef"
WshShell.RegDelete "HKCU\Software\WROX\VBScript\"
WshShell.RegDelete "HKCU\Software\WROX\"
```

在修改注册表设置时，需要特别的小心。对注册表进行不当的修改可能会导致系统变得不稳定，甚至使其变得完全不可用。如果不了解注册表的内部工作机制，那么强烈建议在实际操作之前就此问题阅读一些相关的资料。

Run

Run 方法在一个新进程中运行一个程序。

object.Run(strCommand, [intWindowStyle], [bWaitOnReturn])

- **object:** WshShell 对象。
- **strCommand:** 一个字符串，表示想要运行的命令行，必须包含所有要传递给可执行文件的参数。
- **intWindowStyle:** 可选。一个整数值，表示程序窗口的外观。并不是所有的程序都使用这一信息。
- **bWaitOnReturn:** 可选。一个 Boolean 值，表示脚本在其下一条语句之前是否需要等待程序结束。如果设置为 **True**，脚本一直停止执行直到程序结束，**Run** 返回任一个由程序返回的错误代码。如果设置为 **False**(默认)，程序启动后，**Run** 方法立即返回 **0**(这不是错误代码)。

Run 方法返回一个整数。该方法使用一个新的 **Windows** 进程启动运行一个程序。可以让脚本等待程序运行完毕再继续执行，这样就可以是脚本和程序同步执行。如果将某个文件类型成功注册到某一个特定程序，那么对该文件调用 **Run** 方法就会启动注册的程序。例如，对一个*.txt 文件调用 **Run** 方法，就会启动记事本程序并将文本文件加载到其中。表 15-16 列出了 **intWindowStyle** 的可用值。

表 15-16

IntWindowStyle	描 述
0	隐藏当前窗口并激活另一个窗口
1	激活并显示一个窗口 如果该窗口处于最小化或最大化状态，系统将恢复其原始尺寸 如果是初次显示该窗口，应用程序就应该指定这个标志
2	激活并以最小化状态显示窗口
3	激活并以最大化状态显示窗口
4	以其最近位置和尺寸显示一个窗口 活跃的窗口将继续保持活跃
5	以其当前位置和尺寸显示一个窗口
6	最小化指定的窗口并将其激活为 Z 序列中仅次于顶层的窗口
7	以最小化形式显示窗口 活跃的窗口将继续保持活跃
8	以其当前状态显示窗口 活跃的窗口将继续保持活跃
9	激活并显示窗口 如果该窗口处于最小化或最大化状态，系统将恢复其原始尺寸

	如果要恢复一个最小化窗口，应用程序就应该指定这个标志
10	根据启动应用程序的程序状态设置窗口显示的状态

下面的代码打开一个命令提示符窗口并显示驱动器 C:的内容。

```
Dim oShell
Set oShell = WScript.CreateObject ("WScript.shell")
oShell.run "cmd /K CD C:\ & Dir"
Set oShell = Nothing
```

SendKeys
Sendkeys 方法向活跃窗口发送一次或多次击键(仿佛来自键盘)。

object.SendKeys(string)

- **object**: WshShell 对象。
- **string**: 一个字符串值，表示想要发送的击键。

使用 **SendKeys** 方法可以向没有内建的自动化接口的应用程序发送击键。多数键盘字符可由单个击键表示，但有些键盘字符是由击键的组合构成的(例如 **Alt+F4**)。

要发送一个单独的键盘字符，只要将该字符本身作为字符串参数发送即可。例如，要发送字母“v”。要发送一个空格，可以发送“ ”。

也可以使用 **SendKeys** 方法发送多次击键。可以依次加入每次击键，形成一个序列，创建一个复合的字符串参数来表示击键的序列。例如，要发送击键 x、y 和 z，可以发送字符串参数“xyz”

SendKeys 方法使用某些字符作为其他字符的限定符。这些特殊的字符包括圆括弧、尖括弧、花括弧，以及表 15-17 中列出的一些字符。

表 15-17

加号	+
上尖角	^
百分号	%
波浪号	~

要发送这些字符，可以将它们包含在一对花括弧“{}”中间。所以如果要发送加号，可以发送字符串参数“{+}”。

在 **SendKeys** 中使用方括弧“[]”时，没有什么特殊含义，但为了满足那些为它们指定了特

殊含义的应用程序(例如 Dynamic Data Exchange)的需要，仍需要将它们包含在花括弧中。

要发送方括弧字符，可以使用字符串数组“{[}”发送左括弧，使用字符串数组“{]}”发送右括弧。要发送花括弧字符，可以发送字符串参数“{{”发送左括弧，发送字符串“}}”发送右括弧。

某些击键不产生任何字符(如 Enter 和 Tab)，某些击键代表一些动作(如 Backspace 和 Break)。要发送这些击键，可以发送表 15-18 中列出的这些字符串参数。

表 15-18

Backspace	{BACKSPACE}, {BS}, or {BKSP}
Break	{BREAK}
(续表)	
Caps Lock	{CAPSLOCK}
Del or Delete	{DELETE} or {DEL}
Down Arrow	{DOWN}
End	{END}
Enter	{ENTER} or ~
Esc	{ESC}
Help	{HELP}
Home	{HOME}
Ins or Insert	{INSERT} or {INS}
Left Arrow	{LEFT}
Num Lock	{NUMLOCK}
Page Down	{PGDN}
Page Up	{PGUP}
Print Screen	{PRTSC}
Right Arrow	{RIGHT}
Scroll Lock	{SCROLLLOCK}
Tab	{TAB}
Up Arrow	{UP}
F1	{F1}
F2	{F2}
F3	{F3}
F4	{F4}
F5	{F5}
F6	{F6}
F7	{F7}
F8	{F8}
F9	{F9}

F10	{F10}
F11	{F11}
F12	{F12}
F13	{F13}
F14	{F14}
F15	{F15}
F16	{F16}

要发送常规字符与 **Shift**、**Ctrl** 或 **Alt** 键的组合，就需要创建一个复合的字符串参数以表示这种击键组合。可以在常规击键之前加上表 15-19 中的一个或多个特殊字符。

表 15-19

键	特殊字符
Alt	%
Ctrl	^
Shift	+

当用做这种用途时，这些特殊字符不需要被包含在花括弧中。

要表示按下其他某些键的同时，必须按下 **Shift**、**Ctrl** 或 **Alt** 键的情况，可以修改圆括弧中包含的击键序列创建一个复合的字符串参数。例如下面这些击键以及相应的操作：

- 按下 **Shift** 的同时按下 **V** 和 **B**，发送字符串参数“+(VB)”。
- 按下 **Shift** 的同时按下 **V**，然后长按 **B**(不要按下 **Shift**)，发送字符串参数“+VB”。

可以使用 **SendKeys** 方法发送一个单独的击键被按下若干次所构成的击键模式。可以创建一个复合的字符串参数，指定想要重复的击键，后面再跟上想要重复的次数。使用的复合字符串参数形如{击键 次数}。例如，要发送“V”被按下 10 次的击键模式，可以发送字符串参数“{V 10}”。

单个击键被按下若干次的模式是唯一的一种能够发送的击键模式。例如，可以发送 10 次“V”，但不能发送 10 次“Ctrl+V”。注意，不能向应用程序发送 **Print Screen** 键 {PRTSC}。

```
<package>

  <job id="vbs">

    <script language="VBScript">

      set WshShell = WScript.CreateObject("WScript.Shell")

      WshShell.Run "calc"
```

```

        WScript.Sleep 100

        WshShell.AppActivate "Calculator"

        WScript.Sleep 100
WshShell.SendKeys "1{+}"

        WScript.Sleep 500

        WshShell.SendKeys "2"

        WScript.Sleep 500

        WshShell.SendKeys "~"

        WScript.Sleep 500

        WshShell.SendKeys "*9"

        WScript.Sleep 500

        WshShell.SendKeys "~"

        WScript.Sleep 2500

    </script>

</job>

</package>

    Exec

    Exec 方法在一个子命令解释器中运行一个应用程序，子命令解释器提供对 StdIn、StdOut 和 StdErr 流的访问。

    object.Exec(strCommand)

```

- **object:** WshShell 对象。
- **strCommand:** 一个字符串值，表示用于运行脚本的命令行。

Exec 方法返回一个 WshScriptExec 对象，该对象提供使用 Exec 运行的脚本的状态和错误信息，也提供对 StdIn、StdOut 和 StdErr 通道的访问。Exec 方法只能执行命令行应用程序，并且不能用于运行远程脚本。

15.6.4 WshNamed 对象

WshNamed 对象提供从命令行中对有名称参数的访问。

WshArguments 对象的 Named 属性返回 WshNamed 对象，这是一个有名称参数的集合。这个集合使用了参数名称作为索引来获取每个参数的值。访问命令行参数集合有下列三种方法：

- 使用 WshAruments 对象访问整个参数集合。
- 使用 WshNamed 对象访问有名字的参数。
- 使用 WshUnnamed 对象访问没有名字的参数。

1. 访问 WshNamed 对象

该对象的访问是通过创建 WScript.Named 对象的实例实现的。

```
Set argsNamed = WScript.Arguments.Named
```

2. WshNamed 对象的属性

WshNamed 对象具有两个属性：

- Item
- Length

Item

Item 属性提供了对 WshNamed 对象中的项目的访问。

```
Object.Item(key)
```

- object: WshNamed 对象。
- key: 想要获取的项目名称。

Item 属性返回一个字符串。对于集合来说，它根据给定的关键字返回一个项目。在命令行中输入参数时，可以在字符串中使用空格，只要将其包含在引号中即可。下面这行代码可以用于在命令提示符下运行脚本：

```
sample.vbs /a:arg1 /b:arg2
```

如果脚本中包含下面的代码。

```
WScript.Echo WScript.Arguments.Named.Item("b")
```

```
WScript.Echo WScript.Arguments.Named.Item("a")
```

就会产生这样的输出结果。

```
arg2
```

```
arg1
```

```
Length
```

`Length` 属性是一个只读的整数，可以在编写 `Jscript` 脚本时使用。同样地，这个属性不在本书的讨论范围之内。

3. `WshNamed` 对象的方法

`WshNamed` 对象具有两个方法：

- `Count`
- `Exists`

`Count`

`Count` 方法返回 `WshNamed` 或 `WshUnnamed` 对象的选项开关个数。

```
object.Count
```

- `object`: `WshNamed` 对象。

`Count` 方法用于返回一个整数值。该方法是给 `VBScript` 用户使用的，`JScript` 用户应该使用 `length` 属性。

```
For x = 0 to WScript.Arguments.Count-1
```

```
    WScript.Echo WScript.Arguments.Named(x)
```

```
Next x
```

```
Exists
```

`Exists` 方法可以用于确定某一给定的键值是否存在于 `WshNamed` 对象中。

`object.Exists(key)`

- `object`: `WshNamed` 对象。
- `key`: 字符串值, 表示 `WshNamed` 对象的一个参数。

这个方法返回一个 `Boolean` 值。如果请求的参数确在命令行中被指定, 就返回 `True`(否则返回 `False`)。在命令提示符中输入下面的代码运行脚本:

```
sample.vbs /a:arg1 /b:arg2
```

下面的代码可以用于确定是否使用了参数 `/a`、`/b` 和 `/c`:

```
WScript.Echo WScript.Arguments.Named.Exists("a")
```

```
WScript.Echo WScript.Arguments.Named.Exists("b")
```

```
WScript.Echo WScript.Arguments.Named.Exists("c")
```

15.6.5 WshUnnamed 对象

`WshUnnamed` 对象提供了从命令行中对未命名对象的访问。它是一个由 `WshAruments` 对象的 `Unnamed` 属性返回的只读集合。从该集合中获取单个参数值时, 使用由 0 开始的索引。

访问命令行参数集合有下列三种方法:

- 使用 `WshAruments` 对象访问整个参数集合。
- 使用 `WshNamed` 对象访问有名字的参数。
- 使用 `WshUnnamed` 对象访问没有名字的参数。

1. 访问 WshUnnamed 对象

该对象的访问是通过创建 `WScript.Arguments.Unnamed` 对象的实例实现的。

```
Set argsUnnamed = WScript.Arguments.Unnamed
```


2. WshUnnamed 对象的属性

WshUnnamed 对象具有两个属性：

- Item
- Length

这两个属性和 WshNamed 的类似，这里就不再重复介绍。

3. WshUnnamed 对象的方法

WshUnnamed 对象具有一个方法：

- Count

这个方法和 WshNamed 的类似，这里就不再重复介绍。

15.6.6 WshNetwork 对象

WshNetwork 对象提供对计算机所连接的网络上共享资源的访问。如果想连接到网络共享或网络打印机，从网络共享或网络打印机断开连接，映射或删除网络共享，或访问网络上某一用户的信息，都需要创建一个 WshNetwork 对象。

1. 访问 WshNetwork 对象

该对象的访问是通过创建 WScript.Network 对象的实例实现的。

```
Set WshNetwork = WScript.CreateObject("WScript.Network")
```

2. WshNetwork 对象的属性

WshNetwork 对象具有三个属性：

- ComputerName
- UserDomain
- UserName

ComputerName

ComputerName 属性返回计算机系统的名称。

object.ComputerName

- object: WshNetwork 对象。

ComputerName 属性包含一个字符串值，表示计算机系统的名称。

```
<package>
```

```
  <job id="vbs">
```

```
    <script language="VBScript">
```

```
      Set WshNetwork = WScript.CreateObject("WScript.Network")
```

```
      WScript.Echo "Domain = " & WshNetwork.UserDomain
```

```
      WScript.Echo "Computer Name = " & WshNetwork.ComputerName
```

```
      WScript.Echo "User Name = " & WshNetwork.UserName
```

```
    </script>
```

```
  </job>
```

```
</package>
```

UserDomain

UserDomain 属性返回用户的域名。

object.UserDomain

- object: WshNetwork 对象。

UserDomain 属性在 Windows 98 和 Windows Me 下不可用，除非设置了 USERDOMAIN 环境变量。这个变量默认情况下是没有被设置的。

```
<package>
```

```
  <job id="vbs">
```

```
    <script language="VBScript">
```

```
      Set WshNetwork = WScript.CreateObject("WScript.Network")
```

```
      WScript.Echo "Domain = " & WshNetwork.UserDomain
```

```
        WScript.Echo "Computer Name = " & WshNetwork.ComputerName

        WScript.Echo "User Name = " & WshNetwork.UserName

    </script>

</job>

</package>
```

UserName

UserName 属性返回某个用户的名称。

object.UserName

- object: WshNetwork 对象。

UserName 属性以字符串返回一个用户的名称。

```
<package>

    <job id="vbs">

        <script language="VBScript">

            Set WshNetwork = WScript.CreateObject("WScript.Network")

            WScript.Echo "Domain = " & WshNetwork.UserDomain

            WScript.Echo "Computer Name = " & WshNetwork.ComputerName

            WScript.Echo "User Name = " & WshNetwork.UserName

        </script>

    </job>

</package>
```

3. WshNetwork 的方法

WshNetwork 对象有下列 8 个方法可用：

- AddWindowsPrinterConnection
- AddPrinterConnection
- EnumNetworkDrives

- EnumPrinterConnection
- MapNetworkDrive
- RemoveNetworkDrive
- RemovePrinterConnection
- SetDefaultPrinter

AddWindowsPrinterConnection

AddWindowsPrinterConnection 方法在计算机系统中添加一个 Windows 打印机连接。

- Windows NT/2000/XP/Vista:

```
object.AddWindowsPrinterConnection(
    strPrinterPath
)
```

- Windows 9x/Me:

```
object.AddWindowsPrinterConnection(
    strPrinterPath,
    strDriverName[,strPort]
)
```

- object: WshNetwork 对象。
- strPrinterPath: 一个字符串值，表示打印机连接的路径。
- strDriverName: 一个字符串值，表示驱动器的名称(在 Windows NT/2000/XP 中忽略此参数)。
- strPort: 可选。一个字符串值，为打印机连接指定一个打印机端口(在 Windows NT/2000/XP 系统中忽略此参数)。

- 使用这个方法添加一个打印机连接和使用控制面板中的 **Printer** 选项是非常类似的。这个方法允许创建一个打印机连接，并且能够很方便地将其定向到某一特定端口。
- 如果连接失败，就会产生一个错误。

```
Set WshNetwork = WScript.CreateObject("WScript.Network")
```

```
PrinterPath = "\\printerserver\DefaultPrinter"
```

```
WshNetwork.AddWindowsPrinterConnection PrinterPath
```

```
AddPrinterConnection
```

AddPrinterConnection 方法在计算机系统中添加一个远程打印机连接。

```
object.AddPrinterConnection(strLocalName,
```

```
strRemoteName[,bUpdateProfile][,strUser][,strPassword])
```

- **object:** WshNetwork。
- **strLocalName:** 一个字符串值，表示指派给以连接打印机的本地名称。
- **strRemoteName:** 一个字符串值，表示远程打印机的名称。
- **bUpdateProfile:** 可选。一个 Boolean 值，表示打印机映射是否被存储到当前用户的参数文件中。如果提供了 bUpdateProfile 参数并且值为 True，打印机映射就会被存储到用户参数文件中。默认值为 False。
- **strUser:** 可选。一个字符串值，表示用户名。如果使用某个其他用户，而非当前用户的参数文件映射一个远程的打印机，就可以指定 strUser 和 strPassword。
- **strPassword:** 可选。一个字符串值，表示用户的密码。如果使用某个其他用户，而非当前用户的参数文件映射一个远程的打印机，就可以指定 strUser 和 strPassword。

```
EnumNetworkDrives
```

EnumNetworkDrives 方法返回当前网络驱动器的映射信息。

```
objDrives = object.EnumNetworkDrives
```

- **object:** WshNetwork 对象。
- **objDrives:** 一个变量，保存网络驱动器的映射信息。

这个方法返回一个集合，它是一个由关联项目对(网络驱动器本地名称与其关联的 UNC (Universal Naming Convention, 通用命名规范)名称)构成的数组。集合中的偶数元素表示逻辑驱动器的本地名称，而奇数元素表示其关联的 UNC 共享名。

集合中的第一个元素的索引是 0。

EnumPrinterConnection

EnumPrinterConnection 方法返回当前网络打印机的映射信息。

```
objPrinters = object.EnumPrinterConnections
```

- **object:** WshNetwork 对象。
- **objPrinters:** 保存网络打印机的映射信息的变量。

EnumPrinterConnection 方法返回一个集合，它是一个由关联项目对(网络打印机本地名称与其关联的 UNC(Universal Naming Convention, 通用命名规范)名称)构成的数组。集合中的偶数元素表示打印机端口，而奇数元素表示网络打印机的 UNC 名称。

集合中的第一个元素的索引是 0。

MapNetworkDrive

MapNetworkDrive 方法在计算机系统中添加一个共享网络驱动器。

```
object.MapNetworkDrive(strLocalName, strRemoteName, [bUpdateProfile],  
[strUser], [strPassword])
```

- **object:** WshNetwork 对象。
- **strLocalName:** 一个字符串值，表示被映射驱动器在本地的名称。
- **strRemoteName:** 一个字符串值，表示共享的 UNC 名称(\\xxx\yyy)。
- **bUpdateProfile:** 可选。一个 Boolean 值，表示映射信息是否被存储到当前用户的参数文件中。如果提供了 bUpdateProfile 参数并且值为 True，映射信息就会被存储到用户参数文件中。默认值为 False。
- **strUser:** 可选。一个字符串值，表示用户名。如果使用某个其他用户，而非当前用户的凭证映射一个网络驱动器，就必须指定这个参数。
- **strPassword:** 可选。一个字符串值，表示用户的密码。如果使用某个其他用户，而非当前用户的凭证映射一个网络驱动器，就必须指定这个参数。

如果试图映射一个非共享的网络驱动器，就会导致一个错误。

RemoveNetworkDrive

`RemoveNetworkDrive` 方法在计算机系统中删除一个共享的网络驱动器。

```
object.RemoveNetworkDrive(strName, [bForce], [bUpdateProfile])
```

- **object:** `WshNetwork` 对象。
- **strName:** 一个字符串值，表示想要删除的已映射驱动器名称。**strName** 参数既可以是一个本地名称，也可以是远程名称，这取决于对于驱动器映射的方式。
- **bForce:** 可选。一个 `Boolean` 值，表示是否强制删除已映射的驱动器。如果指定了 **bForce** 参数并且其值为 `True`，那么无论资源是否正在被使用，该方法都会删除连接。
- **bUpdateProfile:** 可选。一个 `Boolean` 值，表示是否从用户参数文件中删除映射信息。如果指定了 **bUpdateProfile** 参数并且其值为 `True`，就会从用户参数文件中删除映射信息。**bUpdateProfile** 默认值为 `False`。

如果某驱动器存在从本地名称(驱动器字母)到远程名称(UNC 名称)的映射，**strName** 就必须被设置为本地名。如果网络路径没有本地名称映射，**strName** 就必须被设置为远程名称。

下面的脚本删除网络驱动器“G:”

```
Dim WshNetwork
```

```
Set WshNetwork = WScript.CreateObject("WScript.Network")
```

```
WshNetwork.RemoveNetworkDrive "G:"
```

```
RemovePrinterConnection
```

`RemovePrinterConnection` 方法在计算机系统中删除一个共享的网络打印机连接。

```
object.RemovePrinterConnection(strName, [bForce], [bUpdateProfile])
```

- **object:** `WshNetwork` 对象。
- **strName:** 一个字符串值，表示打印机的标识名称。它可以是一个 `UNC` 名称(格式为 `\\xxx\yyy`)或者一个本地名称(如 `LPT1`)。
- **bForce:** 可选。一个 `Boolean` 值，表示是否强制删除已映射的打印机。如果指定了 **bForce** 参数并且其值为 `True`，那么无论是否有用户正在连接，该方法都会删除打印机连接。

- **bUpdateProfile:** 可选。一个 Boolean 值, 如果设置为 True(默认是 False), 所做的更改保存到用户参数文件中。

RemovePrinterConnection 方法将同时删除基于 Windows 和基于 DOS 的打印机连接。

如果打印机是使用 **AddPrinterConnection** 方法连接的, **strName** 就必须是打印机的本地名称。

如果打印机是使用 **AddWindowsPrinterConnection** 方法或手工连接的, **strName** 就必须是打印机的 UNC 名称。

SetDefaultPrinter

SetDefaultPrinter 方法将某个远程打印机指定为默认打印机。

```
object.SetDefaultPrinter(strPrinterName)
```

- **object:** WshNetwork 对象。
- **strPrinterName:** 一个字符串值, 表示远程打印机的 UNC 名称。

使用基于 DOS 的打印机连接时, **SetDefaultPrinter** 方法调用将失败。此外, 也不能使用 **SetDefaultPrinter** 方法确定当前已安装的默认打印机名称。

15.6.7 WshEnvironment 对象

WshEnvironment 对象提供对 Windows 环境变量集合的访问。

这个对象是一个由 **WshShell** 对象的 **Environment** 属性返回的环境变量集合。该集合包含全部的环境变量(包括有名称的和没有名称的)。

要获取集合中某个具体的环境变量(及其值), 可以使用环境变量名作为索引。

1. 访问 WshEnvironment 对象

该对象的访问是通过创建 **WScript.Environment** 对象的实例实现的。下面的脚本返回运行脚本的系统中安装的处理器的个数:

```
Set WshShell = WScript.CreateObject("WScript.Shell")
```

```
Set WshSysEnv = WshShell.Environment("SYSTEM")
```



```
WScript.Echo WshSysEnv("NUMBER_OF_PROCESSORS")
```

2. WshEnvironment 对象的属性

WshEnvironment 对象具有两个属性：

- Item
- Length

Item

Item 属性公开集合中的某一指定元素。

Object.Item(natIndex)

- object: EnumNetworkDrive 或 numPrinterConnections 方法的返回值，或由 Environment 或 SpecialFolders 属性返回的对象。
- natIndex: 设置要获取的元素。

Item 是每个集合的默认属性。对于 EnumNetworkDrive 和 EnumPrinterConnections 集合来说，索引是一个整数，但对于 Environment 和 SpecialFolders 来说，索引是一个字符串。

在 VBScript 中，如果请求的文件夹(strFolderName)不存在，WshShell.SpecialFolders.Item(strFolderName)将返回“Empty”。

```
<package>
```

```
  <job id="vbs">
```

```
    <script language="VBScript">
```

```
      Set WshShell = WScript.CreateObject("WScript.Shell")
```

```
      Set WshSpecialFolders = WshShell.SpecialFolders
```

```
      For x = 0 To WshSpecialFolders.Count - 1
```

```
        WScript.Echo WshSpecialFolders.Item(x)
```

```
      Next
```

```
    </script>
```

```
  </job>
```

```
</package>
```

Length

Length 属性是一个只读的整数，可以在编写 Jscript 脚本时使用。同样地，这个属性不在本书的讨论范围之内。

3. WshEnvironment 对象的方法

WshEnvironment 对象具有两个方法：

- Count
- Remove

Count

Count 方法返回一个 Long 值，表示集合中元素的个数。

object.Count

- object: Arguments 对象

Count 方法返回一个整数值。该方法是给 VBScript 用户使用的，JScript 用户应该使用 Length 属性。

```
For x = 0 to WScript.Arguments.Count-1
```

```
    WScript.Echo WScript.Arguments.Named(x)
```

```
Next x
```

Remove

Remove 方法删除一个现有的环境变量。

object.Remove(strName)

- object: WshEnvironment 对象。
- strName: 一个字符串值，表示要删除的环境变量的名称。

Remove 方法删除下列类型的环境变量：

- PROCESS
- USER

- SYSTEM
- VOLATILE

使用 `Remove` 方法删除的环境变量将在当前会话结束后被恢复。

```
Dim WshShell, WshEnv

Set WshShell = WScript.CreateObject("WScript.Shell")

Set WshEnv = WshShell.Environment("PROCESS")

WshEnv("tVar") = "VBScript is Cool!"

WScript.Echo WshShell.ExpandEnvironmentStrings("The value of the test variable is:
'%tVar%'")

WshEnv.Remove "tVar"

WScript.Echo WshShell.ExpandEnvironmentStrings("The value of the test variable is:
'%tVar%'")
```

15.6.8 WshSpecialFolders 对象

`WshSpecialFolders` 对象提供对 Windows 特殊文件夹集合的访问。

`WshShell` 对象的 `SpecialFolder` 属性返回 `WshSpecialFolders` 对象。该集合包含对 Windows 特殊文件夹(例如, Desktop 文件夹和 Start menu 文件夹)的引用。

从这个集合中可以使用某特殊文件夹的名称作为索引获取该文件夹的路径。一个特殊文件夹的路径依赖于用户环境。如果一台计算机上有多个用户,那么硬盘上就存有若干组特殊文件夹。下面列出了可用的特殊文件夹:

- AllUsersDesktop
- AllUsersPrograms
- AllUsersStartMenu
- AllUsersStartup
- Desktop
- Favorites
- Fonts

- MyDocuments
- NetHood
- PrintHood
- Programs
- Recent
- SendTo
- StartMenu
- Startup
- Templates

下面的代码演示了如何在 Windows 桌面上创建 Windows Notepad 的一个快捷方式:

```
<package>

  <job id="vbs">

    <script language="VBScript">

      set WshShell = WScript.CreateObject("WScript.Shell")

      strDesktop = WshShell.SpecialFolders("Desktop")

      set oShellLink = WshShell.CreateShortcut(strDesktop & "\Shortcut Script.lnk")

      oShellLink.TargetPath = WScript.ScriptFullName

      oShellLink.WindowStyle = 1

      oShellLink.Hotkey = "CTRL+SHIFT+N"

      oShellLink.IconLocation = "notepad.exe, 0"

      oShellLink.Description = "A Script Generated Shortcut to Notepad"

      oShellLink.WorkingDirectory = strDesktop

      oShellLink.Save

    </script>

  </job>

</package>
```

1. WshSpecialFolders 对象的属性: Item

WshSpecialFolders 对象具有一个属性:

- Item

Item 属性暴露集合中某一指定的元素。

Object.Item(natIndex)

- Object: EnumNetworkDrive 或 EnumPrinterConnections 方法的返回结果, 或 Environment 或 SpecialFolders 属性返回的对象。
- natIndex: 设置想要获取的元素。

Item 是每个集合都有的默认属性。对于 EnumNetworkDrive 和 EnumPrinterConnections 集合来说, 索引是整数, 但对于 Environment 和 SpecialFolders 集合来说, 索引是字符串。

```
<package>
```

```
  <job id="vbs">
```

```
    <script language="VBScript">
```

```
      Set WshShell = WScript.CreateObject("WScript.Shell")
```

```
      Set WshSpecialFolders = WshShell.SpecialFolders
```

```
      For x = 0 To WshSpecialFolders.Count - 1
```

```
        WScript.Echo WshSpecialFolders.Item(x)
```

```
      Next
```

```
    </script>
```

```
  </job>
```

```
</package>
```

2 WshSpecialFolders 对象的方法: Count

WshSpecialFolders 对象具有一个方法:

- Count

Count 方法返回 WshNamed 对象或 WshUnnamed 对象的开关数。

object.Count

- object: Arguments 对象。

Count 方法返回一个整数值。Count 方法适用于 VBScript 用户，JScript 用户应该使用 Length 属性。

15.6.9 WshShortcut 对象

WshShortcut 对象允许您使用脚本创建快捷方式。

```
<package>
```

```
  <job id="vbs">
```

```
    <script language="VBScript">
```

```
      set WshShell = WScript.CreateObject("WScript.Shell")
```

```
      strDesktop = WshShell.SpecialFolders("Desktop")
```

```
      set oShellLink = WshShell.CreateShortcut(strDesktop & "\Shortcut  
Script.Ink")
```

```
      oShellLink.TargetPath = WScript.ScriptFullName
```

```
      oShellLink.WindowStyle = 1
```

```
      oShellLink.Hotkey = "CTRL+SHIFT+N"
```

```
      oShellLink.IconLocation = "notepad.exe, 0"
```

```
      oShellLink.Description = "Shortcut Script"
```

```
      oShellLink.WorkingDirectory = strDesktop
```

```
      oShellLink.Save
```

```
    </script>
```

```
  </job>
```

```
</package>
```

1. WshShortcut 对象的属性

WshShortcut 对象具有 8 个属性：

- Arguments
- Description
- FullName
- Hotkey
- IconLocation
- TargetPath
- WindowStyle
- WorkingDirectory

Arguments

Arguments 属性包含 WshArguments 对象(一个参数集合)。从该集合中获取单个参数值时，使用由 0 开始的索引。

```
Set objArgs = WScript.Arguments
```

```
For x = 0 to objArgs.Count - 1
```

```
    WScript.Echo objArgs(x)
```

```
Next
```

Description

Description 属性返回快捷方式的描述信息。

```
object.Description1
```

- Object: WshShortcut 对象。

Description 属性包含一个描述快捷方式的字符串值。

```
<package>
```

```
    <job id="vbs">
```

```
        <script language="VBScript">
```

```
            set WshShell = WScript.CreateObject("WScript.Shell")
```

```

        strDesktop = WshShell.SpecialFolders("Desktop")
        set oShellLink = WshShell.CreateShortcut(strDesktop & "\\Shortcut
Script.Ink")

        oShellLink.TargetPath = WScript.ScriptFullName
        oShellLink.WindowStyle = 1
        oShellLink.Hotkey = "CTRL+SHIFT+N"
        oShellLink.IconLocation = "notepad.exe, 0"
        oShellLink.Description = "Script generated shortcut to Notepad"
        oShellLink.WorkingDirectory = strDesktop
        oShellLink.Save

    </script>
</job>
</package>

```

FullName

FullName 属性返回快捷方式对象目标的有效完整路径。

object.FullName

- **Object:** WshShortcut 对象。

FullName 属性包含一个只读字符串，给出了快捷方式目标的有效完整路径。

```
<package>
```

```

    <job id="vbs">

        <script language="VBScript">

            set WshShell = WScript.CreateObject("WScript.Shell")
            strDesktop = WshShell.SpecialFolders("Desktop")
            set oShellLink = WshShell.CreateShortcut(strDesktop & "\\Shortcut
Script.Ink")

            oShellLink.TargetPath = WScript.ScriptFullName
            oShellLink.WindowStyle = 1

```



```

        oShellLink.Hotkey = "CTRL+SHIFT+N"

        oShellLink.IconLocation = "notepad.exe, 0"

        oShellLink.Description = "Shortcut Script"

        oShellLink.WorkingDirectory = strDesktop

        oShellLink.Save

        WScript.Echo oShellLink.FullName

    </script>

</job>

</package>

```

HotKey

HotKey 属性用于给快捷方式指定一个热键，或识别快捷方式的热键。一个快捷方式的热键是指一些按键的组合，当同时按下这些按键时，就会启动该快捷方式。

```
object.Hotkey = strHotkey
```

- **Object:** WshShortcut 对象。
- **strHotkey:** 一个字符串，表示指定给快捷方式的热键。

下面是 strHotkey 的语法：

[KeyModifier]KeyName

- **KeyModifier:** KeyModifier 可以是这些键之一：Alt+、Ctrl+、Shift+、Ext+。

Ext+指的是“扩展键”。这个键的加入是为了将来可能会在字符集中加入一类新的 Shift 键。

KeyName- a ... z, 0 ... 9, F1 ... F12, ...

KeyName 是大小写不敏感的。

修改前面的代码，添加一个热键，如下。

```

<package>

    <job id="vbs">

        <script language="VBScript">

            set WshShell = WScript.CreateObject("WScript.Shell")

```

```

        strDesktop = WshShell.SpecialFolders("Desktop")
        set oShellLink = WshShell.CreateShortcut(strDesktop & "\ShortcutScript.In
k")

        oShellLink.TargetPath = WScript.ScriptFullName
        oShellLink.WindowStyle = 1
        oShellLink.Hotkey = "CTRL+SHIFT+N"
        oShellLink.IconLocation = "notepad.exe, 0"
        oShellLink.Description = "Shortcut Script"
        oShellLink.WorkingDirectory = strDesktop
        oShellLink.Save
        WScript.Echo oShellLink.FullName
    </script>
</job>
</package>

```

IconLocation

IconLocation 属性用于给快捷方式指定一个图标，或识别快捷方式的图标。

```
object.IconLocation = strIconLocation
```

- **Object:** WshShortcut 对象。
- **strIconLocation:** 一个字符串，指定要使用的图标。该字符串应该包含图标的一个有效的完整路径，以及该图标所关联的一个索引。如果有多个图标，可以使用索引选择其中的一个。索引起始于 0。

修改先前的代码，给快捷方式指定一个标准的 Notepad 图标。

```

<package>

    <job id="vbs">

        <script language="VBScript">

            set WshShell = WScript.CreateObject("WScript.Shell")

            strDesktop = WshShell.SpecialFolders("Desktop")

```

```

        set oShellLink = WshShell.CreateShortcut(strDesktop & "\\ShortcutScript.In
k")

        oShellLink.TargetPath = WScript.ScriptFullName
        oShellLink.WindowStyle = 1
        oShellLink.Hotkey = "CTRL+SHIFT+N"
        oShellLink.IconLocation = "notepad.exe, 0"
        oShellLink.Description = "Shortcut Script"
        oShellLink.WorkingDirectory = strDesktop
        oShellLink.Save

        WScript.Echo oShellLink.FullName
    </script>
</job>
</package>

```

TargetPath

TargetPath 属性给出了快捷方式执行文件的路径。

object.TargetPath

- Object: WshShortcut 或 WshUrlShortcut 对象。

这个属性只用于快捷方式的目标路径，其他所有的参数必须放在 **Argument** 的属性中。

```

<package>

    <job id="vbs">

        <script language="VBScript">

            set WshShell = WScript.CreateObject("WScript.Shell")
            strDesktop = WshShell.SpecialFolders("Desktop")
            set oShellLink = WshShell.CreateShortcut(strDesktop & "\\Shortcut
Script.Ink")

            oShellLink.TargetPath = WScript.ScriptFullName
            oShellLink.WindowStyle = 1

```

```
oShellLink.Hotkey = "CTRL+SHIFT+N"

oShellLink.IconLocation = "notepad.exe, 0"

oShellLink.Description = "Shortcut Script"

oShellLink.WorkingDirectory = strDesktop

oShellLink.Save

WScript.Echo oShellLink.FullName

</script>

</job>

</package>

WindowsStyle

Windowstyle 属性用于指定快捷方式的窗口类型，或识别快捷方式所使用的窗口类型。

object.WindowStyle = intWindowStyle
```

- Object: WshShortcut 对象。
- intWindowStyle: 设置程序执行窗口的类型。

WindowStyle 属性返回一个整数。

表 15-20 列出了 intWindowStyle 的可用值。

表 15-20

intWindowStyle	描 述
1	激活并显示一个窗口 如果该窗口处于最小化或最大化状态，系统将恢复其原始尺寸 如果是初次显示该窗口，应用程序就应该指定这个标志
3	激活并以最大化状态显示窗口
7	以最小化形式显示窗口 活跃的窗口将继续保持活跃

下面代码中的改动保证 Notepad 窗口处于活跃状态。

```
<package>

  <job id="vbs">

    <script language="VBScript">

      set WshShell = WScript.CreateObject("WScript.Shell")
```

```

        strDesktop = WshShell.SpecialFolders("Desktop")
        set oShellLink = WshShell.CreateShortcut(strDesktop & "\ShortcutScript.Ink")

        oShellLink.TargetPath = WScript.ScriptFullName
        oShellLink.WindowStyle = 1
        oShellLink.Hotkey = "CTRL+SHIFT+N"
        oShellLink.IconLocation = "notepad.exe, 0"
        oShellLink.Description = "Shortcut Script"
        oShellLink.WorkingDirectory = strDesktop
        oShellLink.Save
        WScript.Echo oShellLink.FullName
    </script>
</job>
</package>

```

WorkingDirectory

WorkingDirectory 属性用于指定快捷方式的工作目录，或识别快捷方式所使用的工作目录。

```
object.WorkingDirectory = strWorkingDirectory
```

- Object: WshShortcut 对象。
- strWorkingDirectory: 一个字符串，表示快捷方式的起始目录。

```

<package>
    <job id="vbs">
        <script language="VBScript">
            set WshShell = WScript.CreateObject("WScript.Shell")
            strDesktop = WshShell.SpecialFolders("Desktop")
            set oShellLink = WshShell.CreateShortcut(strDesktop & "\ShortcutScript.Ink")
        </script>
    </job>
</package>

```

```
oShellLink.TargetPath = WScript.ScriptFullName

oShellLink.WindowStyle = 1

oShellLink.Hotkey = "CTRL+SHIFT+N"

oShellLink.IconLocation = "notepad.exe, 0"

oShellLink.Description = "Shortcut Script"

oShellLink.WorkingDirectory = strDesktop

oShellLink.Save

WScript.Echo oShellLink.FullName

</script>

</job>

</package>
```

2. WshShortcut 对象的方法

WshShortcut 对象具有一个方法：

- Save

Save 方法将一个快捷方式保存到磁盘。

object.Save

- Object: WshShortcut 或 WshUrlShortcut 对象。

使用 CreateShortcut 方法创建一个快捷方式并设置该快捷方式对象的属性后，可以使用 Save 方法将该快捷方式对象保存到硬盘上。Save 方法使用快捷方式对象的 FullName 属性中的信息确定在硬盘上保存快捷方式对象的位置。

只能对系统对象创建快捷方式——文件、目录和磁盘驱动器。不能为打印机或调度任务创建快捷方式。

15.6.10 WshUrlShortcut 对象

WshUrlShortcut 对象允许您使用脚本创建到 Internet 资源的快捷方式。该对象是 WshShell 对象的子对象。必须使用 WshShell 方法的 CreateShortcut 来创建 WshUrlShortcut 对象。下

面的代码可以保存为一个后缀名为.wsf 的 Windows 脚本文件。

```
WshShell.CreateShortcut(strDesktop & "\URLShortcut.lnk")
```

```
<package>

  <job id="vbs">

    <script language="VBScript">

      set WshShell = WScript.CreateObject("WScript.Shell")

      set oUrlLink = WshShell.CreateShortcut(strDesktop & "\Wrox Web Site.url
")

      oUrlLink.TargetPath = "http://www.wrox.com"

      oUrlLink.Save

    </script>

  </job>

</package>
```

1. WshUrlShortcut 对象的属性

WshUrlShortcut 对象具有两个属性：

- FullName
- TargetPath

FullName

FullName 属性返回快捷方式对象目标的有效完整路径。

object.FullName

- Object: WshUrlShortcut 对象。

FullName 属性包含一个只读的字符串，给出快捷方式目标的有效的完整路径。下面的代码可以保存为一个后缀名为.wsf 的 Windows 脚本文件。

```
<package>

  <job id="vbs">
```

```

        <script language="VBScript">

        set WshShell = WScript.CreateObject("WScript.Shell")

        set oUrlLink = WshShell.CreateShortcut(strDesktop & "\Wrox Web Site.url")

        oUrlLink.TargetPath = "http://www.wrox.com"

        oUrlLink.Save

        WScript.Echo oUrlLink.FullName

        </script>

    </job>

</package>

```

TargetPath

TargetPath 属性给出了快捷方式可执行文件的路径。

object.TargetPath

- Object: WshUrlShortcut 对象。

这个属性只用于快捷方式的目标路径，其他所有的参数必须放在 **Argument** 的属性中。

下面的代码可以保存为一个后缀名为 .wsf 的 Windows 脚本文件。

```

<package>

    <job id="vbs">

        <script language="VBScript">

        set WshShell = WScript.CreateObject("WScript.Shell")

        set oUrlLink = WshShell.CreateShortcut(strDesktop & "\Wrox Web Site.url")

        oUrlLink.TargetPath = "http://www.wrox.com"

        oUrlLink.Save

        WScript.Echo oUrlLink.FullName

        </script>

    </job>

```



```
</package>
```

2. WshUrlShortcut 对象的方法

WshUrlShortcut 对象具有一个方法：

- Save

Save 方法将快捷方式对象保存到磁盘。

object.Save

- Object: WshUrlShortcut 对象。

使用 CreateShortcut 方法创建一个快捷方式并设置该快捷方式对象的属性后，可以使用 Save 方法将该快捷方式对象保存到硬盘上。Save 方法使用快捷方式对象的 FullName 属性中的信息确定在硬盘上保存快捷方式对象的位置。下面的代码可以保存为一个后缀名为 .wsf 的 Windows 脚本文件。

```
<package>
```

```
<job id="vbs">
```

```
<script language="VBScript">
```

```
set WshShell = WScript.CreateObject("WScript.Shell")
```

```
set oUrlLink = WshShell.CreateShortcut(strDesktop & "\Wrox Web Site.url")
```

```
oUrlLink.TargetPath = "http://www.wrox.com"
```

```
oUrlLink.Save
```

```
WScript.Echo oUrlLink.FullName
```

```
</script>
```

```
</job>
```

```
</package>
```

15.7 小结

这一章内容很多，下面给出内容摘要：

- 使用 Windows 脚本宿主编写脚本所需要的工具
- WSH 的使用方法，包括创建定制的解决方案实现脚本与 COM 组件的集成
- cscript.exe 和 wscript.exe 执行环境以及二者的不同
- 如何使用.wsh 配置文件定制单个脚本的行为
- WSH 开发人员可以使用的对象模型的详细介绍

第 16 章 Windows 脚本组件

在这一章中我们将学习 Windows 脚本组件(Windows Script Component)，内容包括脚本组件的结构，以及如何创建及注册脚本组件。在这一章后半部分的内容中，我们将学习如何在组件中使用类。如果习惯于使用 XML，那么这里所介绍的脚本的结构会看上去比较熟悉，这将是一个很不错的优势。如果不是这样也没关系，只要仔细研究本章中的示例就能搞清所有的知识点。

16.1 什么是 Windows 脚本组件

Windows 脚本组件是一种解释执行的 COM 组件(之所以要解释执行，是因为 VBScript 和所有其他的脚本语言都是解释执行的)。从结构上讲，它们是包含脚本代码的基于 XML 的文件。在这些文件中，可以使用 VBScript、Jscript、Python、PScript、PERLScript，或其他脚本语言。像往常一样，这一章内容是通过 VBScript 来描述的(原因显然)，但也可以使用其他的脚本语言。

脚本对象是由脚本组件运行时(Script Component Runtime)负责解释执行的，脚本对象运行时暴露属性和方法、触发事件，并使组件从调用程序的角度看上去像一个已编译的 COM 组件。在下一节中，我们将详细介绍脚本组件运行时。脚本组件完全就是 COM 组件，能够调用其他的 COM 组件。它们还具有一些集成到 ASP 库和 Internet Explorer DHTML 行为中的接口，以便非常容易地构建用于 Web 应用的组件。

这里有一点很重要，那就是脚本组件并不是设计用作前期绑定访问对象的。在构建应用程序时，前期绑定访问对象提供正在访问的对象的相关信息，一般来说使用前期绑定访问时程序运行得快一些。后期绑定访问意味着在编译时没有关于被访问对象的信息，所有信息都是在运行时动态获得的。如果引用一个组件对象作为前期绑定组件，应用程序就会产生一个错误。这是使用脚本组件的一般性问题，经常会遇到。尽量使用后期绑定就会减少实现中的错误。

您可能想问，既然可以使用 Visual Basic 来构建标准的 COM 组件，那么为什么还要使用 Windows 脚本组件呢？关于这个问题，主要的原因在于，Windows 脚本组件不需要编译

器。要构建脚本组件，只要使用简便的 Windows Notepad 或某种简单的文本编辑器即可实现。脚本组件还是一种快速而简单的方法，可以用于封装使用 VBScript 编写的函数和例程。使用它将源代码创建成一个库。最后有一点更有说服力，那就是 ASP 接口允许直接访问 ASP 库，以便快速而简便地实现与 Internet 或内网站点的集成。

16.2 需要的工具

在使用 Windows 脚本组件之前，需要准备一些工具。下面给出创建脚本组件所需要的工具清单。

- VBScript 5.0 的库或更高版本(如果可能的话，最好使用 5.6 版本)：要在机器上正确运行脚本组件，必须有 VBScript 5.0 的库(最好是最新的 5.6 版本)。脚本组件在运行时使用 Windows 脚本宿主，所以也需要 Windows 脚本宿主。幸运的是，所有这些都随脚本库一起安装。
- Internet Explorer 5.0 或更高版本(最好是 Internet Explorer 6.0 或 7.0)
- 脚本组件向导(Script Component Wizard)(可选)：虽然只需要 Notepad 再加上一定的想象力就可以创建 Windows 脚本组件，但如果想编写很多脚本，手工编写就会变得很乏味。Microsoft 提供了脚本组件向导(下载的地址是 www.microsoft.com/scripting——这个地址可能会变化，但只要访问脚本区的下载区找到下载的连接就行，文件名是 `wz10en.exe`)，可以加速脚本组件框架的创建。
- 脚本组件的文档副本(可选，但可能比较有用，尤其是如果要处理一些复杂的工作时)

所有这些工具都可以从 Microsoft 的 Web 站点上免费下载。

16.3 脚本组件运行时

因为脚本组件在运行时(也就是启动之后)是解释执行的，所以在客户端系统上需要安装一个解释器。脚本组件运行时(`scrobj.dll`)是一个用于控制客户端与脚本组件之间的调用的解释器。脚本组件运行时为组件实现了基本的 COM 接口(`IUnknown`)，并处理一些基本的 COM 方法(`QueryInterface`, `AddRef`)，就像 Visual Basic 处理 Visual Basic 组件的低级 COM 例程那

样。

因为要使用解释器来运行,所以脚本组件看上去会与其他已注册的 COM 组件有所不同。接下来我们来简单讨论一下这个问题。假设要使用脚本调用一个名为“Math.WSC”的对象。

```
Set objMath = CreateObject("Math.WSC")
```

接下来首先发生的事就是在注册表中的 HKEY_CLASSES_ROOT 下搜索 Math.WSC 注册表项。如果在 HKEY_CLASSES_ROOT\CLSID 下搜索 GUID(Global Unique Identifier 全局统一标识符),就可以获得 COM 组件的信息。这里要注意的是 InprocServer32 键实际上是 scrobj.dll,而不是脚本组件文件本身。当调用 CreateObject 语句时,实际上创建的是 scrobj.dll 组件。scrobj.dll 文件会查看 ScriptletURL 键来确定组件的位置。目前该文件知道要查看方法调用实际涉及的对象的路径。

注意,键的名称是 ScriptletURL,这表明可以通过 Internet 调用。不过现在还不急于介绍这个问题,本章后面的内容中会涉及。这里我们还要再了解一些关于脚本组件的知识。

16.4 脚本组件文件和向导

现在我们来看一下如何创建实际的脚本组件。如同前面所讲到的,我们可以手工构建脚本文件。不过,Microsoft 制作了一个免费的向导,可以用于构建脚本组件文件,自动化很多创建脚本组件时非常繁琐的工作。这个向导仅仅是建立定义脚本的 XML 框架,如果知道其中的原理,完全可以自己构建。但是,XML 是非常严密的语言,编写过程中很容易出错。因此理所当然,最好的方法就是首先使用向导。下面就来介绍这个过程。

此外,我们以执行步骤的方式来进行介绍。

1. 在 Vista/XP 下依次单击 Start|All Programs|Microsoft Windows Script|Windows Script Component Wizard(在其他操作系统中可以依次单击 Start|Programs|Microsoft Windows Script|Windows Script Component Wizard 快捷方式)。脚本组件向导的步骤 1 如图 16-1 所示。



图 16-1

2. 定义组件。在向导中输入组件的名称，以及它的 ProgID。这里要注意的一点是脚本组件使用一个特殊的 ProgID 来定义组件。在默认情况下，组件的 ProgID 是 componentname.WSC。不过不必担心，在目前这个步骤中，可以修改这个默认名称，也可以在创建组件文件以后修改。脚本组件还可以在 Version 域中保留版本信息，就像其他 COM 组件那样。如果要跟踪更新历史，这个域就很有用。这里要注意，目前这个对话框中的 Location 域仅代表向导所产生的源文件的位置。源文件的位置对于实际的 Windows 操作系统来说无关紧要，除非要注册组件。

3. 各种选项的设置完成之后(其中有些比较灵活，比如 version，其它的稍死板一些)，选择 Next 按钮进入向导的下一步。脚本组件向导的步骤 2 如图 16-2 所示。选择需要的选项之后，选择 Next 按钮进入向导的步骤 3。步骤 2 中包括下列选项：

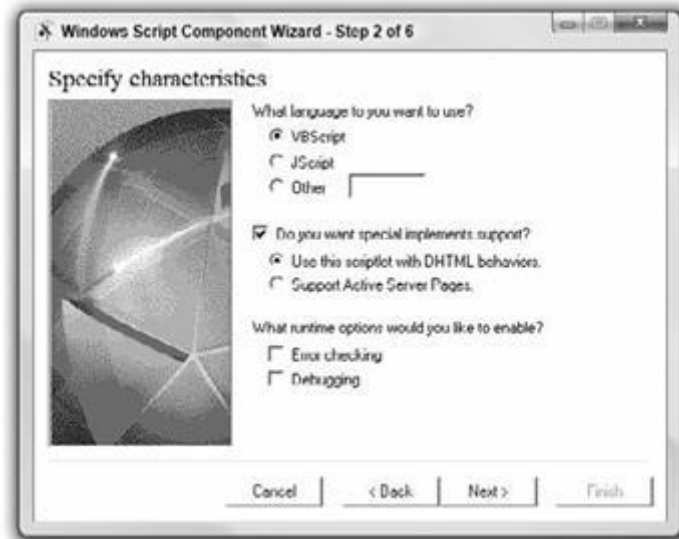


图 16-2

- 语言选择：Windows 脚本组件可以使用 VBScript 或 JScript，但如果计算机中安装了适当的解释器，也可以使用其他的脚本平台，如 Python 和 PERL。在实现设置部分下的两个选项需要一些额外的背景信息，包括 DHTML 行为和 ASP。
- 特殊的实现支持：DHTML 行为是一种简单的，轻量级的组件，与某些 DHTML 对象和 Internet Explorer 事件进行接口通信。ASP 的支持使脚本组件能够获得对 ASP 对象模型的直接访问。ASP 对象模型暴露 ASP Request、Response、Application、Session 和 Server 对象。

DHTML 组件超出了本章的讨论范围，不过可以参考 Microsoft Scripting 网站和 MSDN Web Workshop 网站(<http://msdn.microsoft.com/library/>)来获取关于它的更多信息。ASP 的支持在这一章中会有所讨论，但 ASP 本身将在本书后面的内容中讨论。

- 可以通过选项开启错误检查和调试。如果选择了 Debugging，就可以使用脚本调试器。脚本调试器的下载地址是 <http://msdn.microsoft.com/scripting/>，这是调试脚本组件的唯一方法。脚本调试器使您能够检查变量和查看数据，它的使用方法与 Visual Basic 调试工具的使用方法十分类似。

4. 在向导的步骤 3 中，如图 16-3 所示，可以定义对象的属性，然后单击 Next 按钮。这里可以定义名称，类型，以及组件的默认值。



图 16-3

- Type 的设置并不是指数据类型，而是属性类型，它可以是 Read/Write、Read-Only 或 Write-Only。Default 项允许指定一个属性的默认值。下面的代码显示了一个默认值为 5 的 Read/Write 属性。

```
Dim ReadWriteProperty
```

```
ReadWriteProperty = 5
```

- 值得注意的是，这仅仅是向导声明属性所访问变量的方法，应该作下列修改。

```
Private ReadWriteProperty
```

```
ReadWriteProperty = 5
```

- 这样就保证该变量对于脚本组件是私有的。如果不这样做，该变量就会变成全局变量，那么如果有属性访问这个变量，就会导致一些问题或冲突。

5. 向导的第四个步骤如图 16-4 所示。这一步用于设置组件的方法。在这里我们为 Math 组件添加几个方法。脚本组件向导会以函数的形式生成所有方法。如果需要的话，可以手工将这些方法修改成过程，只要不需要返回值即可。在向导中不能做这种设置的确不太方便，但同样是因为无法对其进行任何操作，也就不必关注。指定方法的名称和参数后，单击 Next 按钮。在添加参数时，一定要用逗号将参数隔开，这样的参数列表如下：

```
param1, param2, param3, ...
```


这里同样要注意，VBScript 只使用 **variants** 类型的变量，所以不要指定类型。如果试图指定类型，就会产生一个错误。也是因为类似的原因，也无法指定返回值的类型。**Variants** 数据类型的使用的确会降低整体性能，因为 **variants** 是占用空间最大的数据类型，它被设计用于表示其他的数据类型。所以每次调用 **variants** 变量时，应用程序必须确定 **variants** 变量的格式。但是既然无法对其进行任何操作，也就不必关注。

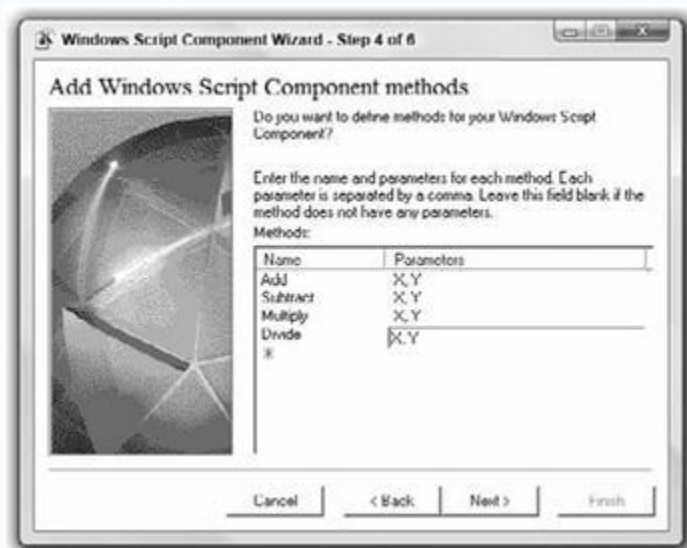


图 16-4

6. 向导的第五步(如图 16-5 所示)允许指定组件的事件。这是脚本组件最精彩的部分。这一节稍后的内容中，我们将讨论更多关于脚本组件的内容。**Math** 组件同样也不会实际使用事件。如果想在对象中加入事件，可以在每行中输入一个事件名称。

脚本组件向导之前的某个版本有一个 **bug**，有可能忽略这部分中的设置。如果发现这种情况，就必须在创建组件之后手工添加事件。本章稍后将讨论这个问题。如果受到了这个 **bug** 的影响，建议升级到最新版本的脚本组件向导。



图 16-5

7. 正确设置了事件的各种选项之后，按下 **Next** 按钮进入向导的最后一个步骤，如图 16-6 所示。最后一个步骤给出组件的一些信息，以及已作出的一些设置。如果发现错误或疏忽，可以按下 **Back** 按钮回到上一个步骤，做必要的修改。最后单击 **Finish** 关闭向导。



图 16-6

单击 **Finish** 按钮之后，向导会创建一个框架组件，类似于下面的代码示例(注意 **classid** 可能与读者的代码不同)：

```
<?xml version="1.0"?>  
<component>
```

```
<registration
  description="Math"
  progid="Math.WSC"
  version="1.00"
  classid="{585c5e81-addf-4006-993a-9701d7c4a41b}"
>
</registration>

<public>
  <property name="ReadOnlyProperty">
    <get/>
  </property>
  <property name="WriteOnlyProperty">
    <put/>
  </property>
  <property name="ReadWriteProperty">
    <get/>
    <put/>
  </property>
  <method name="Add">
    <PARAMETER name="X"/>
    <PARAMETER name="Y"/>
  </method>
  <method name="Subtract">
    <PARAMETER name="X"/>
    <PARAMETER name="Y"/>
  </method>
  <method name="Multiply">
```

```
        <PARAMETER name="X"/>
        <PARAMETER name="Y"/>
</method>
<method name="Divide">
        <PARAMETER name="X"/>
        <PARAMETER name="Y"/>
</method>
<event name="CalcError"/>
</public>

<implements type="Behavior" id="Behavior"/>

<script language="VBScript">
<![CDATA[

dim ReadOnlyProperty
dim WriteOnlyProperty
dim ReadWriteProperty

function get_ReadOnlyProperty()
    get_ReadOnlyProperty = ReadOnlyProperty
end function

function put_WriteOnlyProperty(newValue)
    WriteOnlyProperty = newValue
end function

function get_ReadWriteProperty()
    get_ReadWriteProperty = ReadWriteProperty
end function

function put_ReadWriteProperty(newValue)
    ReadWriteProperty = newValue
```

```
end function

function Add(X, Y)

    Add = "Temporary Value"

end function

function Subtract(X, Y)

    Subtract = "Temporary Value"

end function

function Multiply(X, Y)

    Multiply = "Temporary Value"

end function

function Divide(X, Y)

    Divide = "Temporary Value"

end function

]]>

</script>

</component>
```

如果读者的代码和上面的类似，就说明已经创建了一个 Windows 脚本 COM 组件。现在我们来详细讨论一下。

16.5 暴露属性、方法和事件

接下来要为组件实际定义属性、方法和事件。

16.5.1 属性

脚本组件中的属性可以是 Read/Write、Read-Only 或 Write-Only。在脚本中，它们是使用 `<property></property>` 标记来实现的。在这些标记中，可以为属性设置 `get` 和 `put` 选项。`get`

选项用于读取数值，`put` 选项用于写入属性。下面的代码示例列出了创建用于首次声明三类属性的结构。

```
<property name="ReadOnlyProperty">
    <get/>
</property>
<property name="WriteOnlyProperty">
    <put/>
</property>
<property name="ReadWriteProperty">
    <get/>
```

在脚本文件后半部分中，这些属性随后在脚本代码中被实际定义。

```
<script language="VBScript">
<![CDATA[

dim ReadOnlyProperty
dim WriteOnlyProperty
dim ReadWriteProperty

function get_ReadOnlyProperty()
    get_ReadOnlyProperty = ReadOnlyProperty
end function

function put_WriteOnlyProperty(newValue)
    WriteOnlyProperty = newValue
end function

function get_ReadWriteProperty()
    get_ReadWriteProperty = ReadWriteProperty
end function
```

```
function put_ReadWriteProperty(newValue)
```

```
    ReadWriteProperty = newValue
```

```
end function
```

```
function Add(X, Y)
```

```
    Add = "Temporary Value"
```

```
end function
```

```
function Subtract(X, Y)
```

```
    Subtract = "Temporary Value"
```

```
end function
```

```
function Multiply(X, Y)
```

```
    Multiply = "Temporary Value"
```

```
end function
```

```
function Divide(X, Y)
```

```
    Divide = "Temporary Value"
```

```
end function
```

```
]]>
```

```
</tingyi5201718
```

```
script>
```

在属性的 **get** 和 **put** 函数中可以加入任何额外的控制逻辑。这里的示例没有包含任何真实的属性。在后面讨论类的时候，会给出一些使用属性的示例。

要记住，脚本组件可以实现除 **COM** 对象以外的组件，因此可以用它创建 **ADO** 组件、访问 **LDAP**(**Lightweight Directory Access Protocol**，轻量级目录访问协议)和 **Exchange**，或调用 **Microsoft Word** 或 **Excel**。任何东西都可以用脚本组件来表示。

16.5.2 方法

脚本组件中的方法是在脚本文件的对象定义部分使用<method></method>标记定义的。方法的参数使用一个<parameter>定义其值，如下面的代码示例中所示。

```
<public>

    <method name="mNoParameters">

    </method>

    <method name="mWithParameters">

        <PARAMETER name="var1"/>

        <PARAMETER name="var2"/>

    </method>

</public>
```

<parameter>标记仅定义输入参数的名称。要记住的是，来自脚本组件向导的所有方法默认情况下都是脚本组件中的一个函数，并且没有指定返回值类型，因为所有的变量都是 variant 数据类型。但是，可以自由使用子过程作为方法来取代函数。

实际的方法代码是在脚本组件的脚本标记中定义的。

```
<script language="VBScript">

<![CDATA[

Function mNoParameters()

    mNoParameters = "Temporary Value"

End Function

Function mWithParameters(var1, var2)

    mWithParameters = "Temporary Value"

End Function

]]>

</script>
```


注意，所有使用 Windows 脚本组件向导创建的方法都返回“Temporary Value”；需要修改这个返回值(除非确实需要一个返回“Temporary Value”的函数！)。还要在函数定义之前声明所有的临时变量。

下面我们给 Math 组件添加实际的方法。

```
<script language="VBScript">
<![CDATA[

Private ReadOnlyProperty
Private WriteOnlyProperty
Private ReadWriteProperty

function get_ReadOnlyProperty()
    get_ReadOnlyProperty = ReadOnlyProperty
end function

function put_WriteOnlyProperty(newValue)
    WriteOnlyProperty = newValue
end function

function get_ReadWriteProperty()
    get_ReadWriteProperty = ReadWriteProperty
end function

function put_ReadWriteProperty(newValue)
    ReadWriteProperty = newValue
end function

function Add(X, Y)
    Add = X + Y
end function

function Subtract(X, Y)
```

```
        Subtract = X - Y
end function

function Multiply(X, Y)
    Multiply = X * Y
end function

function Divide(X, Y)
    Divide = X / Y
end function

]]>
</script>
```

WSC 文档中没有介绍的(因为和使用的脚本语言具体相关)是可以在方法的参数声明中使用 **byval**(传值)和 **byref**(传引用)关键字。**VBScript** 中，默认情况下所有的值都是以 **byref** 方式传递的，所以方法中对变量的任何修改都会改变调用函数中该变量的值。

JScript 变量都是以 **byval** 方式传递的，因为 **JScript** 不能以 **byref** 方式传递变量。

16.5.3 事件

事件是在脚本文件的对象定义中在<event></event>标记中定义的。

在 Windows 脚本组件向导旧的发布(我们不能称其为版本，是因为 Windows 脚本组件向导目前版本仍然是 1.0)中存在一个 **bug**，这意味着它不会创建您所指定的事件。所有的事件声明都必须在脚本文件中手工创建。Windows 脚本组件向导的最新发布已经修正了这个问题。

```
<public>

    <method name="mNoParameters">

        </method>

    <event name="MethodCalled">

        </event>
```

```
</public>
```

事件实际上是通过 `FireEvent()` 方法产生的。`FireEvent()` 方法是在脚本组件的脚本中调用的。事件本身也应该在这里描述，形式为 `ComponentName_EventName`。

```
<script language="VBScript">
```

```
<![CDATA[
```

```
function mNoParameters()
```

```
    FireEvent("MethodCalled")
```

```
    mNoParameters = "Temporary Value"
```

```
end function
```

```
sub MyComponent_MethodCalled()
```

```
    'some event handling code
```

```
end sub
```

```
]]>
```

```
</script>
```

脚本组件还可以在脚本定义中使用一个 `<implement>` 标记处理事件。在脚本组件中捕获事件的语法定义如下：

```
<implements type="COMHandlerName" [id="internalName"] [default=fAssumed]>
```

```
    handler-specific information here
```

```
</implements>
```

`COMHandlerName` 是处理程序(ASP 或行为)或被处理的 COM 对象的名称。`InternalName` 是一个可选的参数，允许为 COM 处理程序定义一个变量名称。`fAssumed` 属性是一个 Boolean 标志(默认值是 True)，表示脚本中已经假设了 `InternalName`。如果将其设置为 `False`，就要在 `<implement>` 标记中隐藏一些成员。

有两类内建的 COM 处理程序：ASP 和行为。ASP COM 处理程序将在本章后面部分中讨论。

16.6 注册信息

要注册一个 Windows 脚本组件，机器上需要有脚本组件运行时(scrobj.dll)，并且要正确注册。这个文件是在安装 VBScript 或 JScript 的脚本引擎时自动注册的。只要安装了脚本运行时和一个合法的脚本组件文件(.wsc)，就可以注册组件。正确注册一个 WSC 文件有三种可用的方法：

- 注册和注销脚本组件最简单的方法是在 Windows Explorer 右键单击组件文件，并在弹出菜单中选择 Register 或 Unregister。这个过程如图 16-7 所示，简单又方便。



图 16-7

- 在手工注册和注销组件的过程中，仍然可以使用 regsvr32.exe。如果使用 Windows 或 Visual Studio 附带的某个旧版本的 regsvr32，可以使用下面的命令。

```
regsvr32 scrobj.dll /n /i:Path/component_name.wsc
```

脚本组件包中附带的新版本的 regsvr32 可以直接注册脚本组件文件。

```
regsvr32 path/component_name.wsc
```

- 也可以在脚本中添加一个注册条目以定义注册行为。可以在组件中添加<registration>标记，如下面代码中的定义：

```
<registration progid="progID" classid="GUID" description="description"
    version="version" [remotable=remoteFlag]>
    <script>
        (registration and unregistration script)
    </script>
</registration>
```

在<script>标记中可以添加一个 Register()和一个 Unregister()事件，当组件在系统中注册和注销时，分别发生这两个事件。progID 属性是可选的，但是必须给 classid 或 progID 指定数据，以备要注册的组件使用。如果这两个属性都没有值，那么在组件注册时就会自动产生它们的值。

所有这些方法将在系统中正确注册一个脚本组件文件。这的确不错，但对于远程组件如何呢？我们可以直接回答，Windows 脚本组件可以远程注册。

要通过以下四个步骤才能使用 DCOM-ready 组件：

(1) 确定组件的 progid 和 clsid。本地组件不需要 clsid 条目，而如果没有这个条目，就说明组件会在注册时创建一个 clsid 条目。

(2) 在每台需要访问组件的本地机器上，在 HKEY_CLASSES_ROOT\componentProgID 下添加一个注册表项。这里的 componentProgID 是脚本组件的 ProgID。这对于 Windows 脚本宿主来说是非常合适的方法。

(3) 在这个注册表项下面，创建一个 CLSID 键并将其值设置为脚本组件的 clsid。

(4) 在组件的<registration>标记中设置 remotable=true。

简化这一过程的另一种方法是在服务器上注册组件并使用 regedit 将注册表项信息导出来。然后可以从 regedit 中将导出的.reg 文件复制到每台需要该组件的机器上。这个文件拷贝到本地机器上后，双击这个.reg 文件就可以将数据导入注册表。现在我们已经创建了一个 DCOM-ready 脚本组件，可以在整个公司中使用。

可以使用一个简短的测试脚本快速地测试这个组件。将下面的代码保存到一个称为 testmathcomp.vbs 的文件中，注册了 Math 组件之后，运行该文件：

```
dim obj
set obj = wscript.createObject("math.wsc")
msgbox obj.add(15, 9)
msgbox obj.subtract(15, 9)
msgbox obj.multiply(15, 9)
msgbox obj.divide(28, 4)
set obj = nothing
```

16.7 创建脚本组件类型库

脚本组件与其他 COM 组件并无区别，也可以有生成的类型库。类型库在某些环境(如 Visual Basic)中用于启用事件机制，或允许诸如 Visual InterDev 这样的程序使用 IntelliSense。类型库包含 COM 组件的描述，也可以通过早期绑定机制或使用像 OLE2VIEW 这样的工具来显示组件中的声明和常量。

要为一个脚本组件生成一个类型库，只要右键单击脚本组件文件并在弹出菜单中选择 Generate Type Library 选项，如图 16-8 所示。



图 16-8

脚本组件还可以在调用 Register 方法时自动为自己生成一个类型库。调用这个方法时，组件使用<registration>标记中的信息。<registration>标记的语法如下：

```
<registration progid="progID" classid="GUID" description="description"
    version="version" [remotable=remoteFlag]>
    <script>
        (registration and unregistration script)
    </script>
</registration>
```

要记住的是，progID 和 classid 项都是可选的，但是这项其中至少有一项要被指定值，标记才合法。progID 是组件的名称，而 classid 项是组件的 GUID。如果 classid 项为空，系统在注册时就会给组件指派一个 GUID。

描述信息和版本信息也都是可选的。如果在前面的 Math 组件中使用一个注册信息条目，就应该添加下面的<registration>标记。

```

<registration
    description="My Simple Math Component"
    progid="Math.WSC"
    version="1.0"
    classid="{585C5E81-ADDF-4006-993A-9701D7C4A41B}">
    <script language="VBScript">
    <![CDATA[
        Function Register()
            Set oComponent = CreateObject("Scriptlet.GenerateTypeLib")
            oComponent.AddURL "C:\Program Files\Microsoft Windows Script\Math.wsc"

```

AddURL 允许向类型库中添加其他的组件文件。如果使用或暴露了其他组件，就要将它们添加到同一类型库中，而不是使用多个文件。

```
oComponent.Path = "C:\Program Files\Microsoft Windows Script\Math.tlb"
```

添加路径，表示组件保存的位置。如果留空，类型库就会被写到脚本组件的当前位置上。

```
oComponent.Doc = "Math component typelib" ' Documentation string.
```

```
oComponent.GUID = "{a1e1e3e0-a252-11d1-9fa1-00a0c90fffc0}"
```

```
oComponent.Name = "MathComponent" ' Internal name for tlb.
```

```
oComponent.MajorVersion = 1
```

```
oComponent.MinorVersion = 0
```

```
oComponent.Write
```

下面是将类型库写到硬盘中的代码。

```
    oComponent.Reset
```

```
End Function
```

```
]]>
```

```
</script>
```

```
</registration>
```

要记住的是，如果想通过 DCOM 使用这个组件，就需要添加下面这行代码。

```
remotable=true
```

这行代码告诉组件需要在注册表中为 DCOM 启动组件。

16.8 如何引用其他组件

一个脚本组件文件可以包含多个组件在其中。可以创建一个组件的库，就像在 Visual Basic 中那样。但是，不能使用 Windows 脚本组件向导来做这件事，所以需要手工完成。脚本组件使用一系列的<package></package>标记创建脚本库。例如，可以在一个文件中定义下面这些组件：

```
<package>
```

```
    <component id="COMObj1">
```

```
    </component>
```

```
    <component id="COMObj2">
```

```
    </component>
```

```
    <component id="COMObj3">
```

```
    </component>
```

```
</package>
```

在每个脚本中，可以为每个组件添加适当的属性、方法和事件。有时还可能需要添加必要的注册信息。

在<package></package>标记中还可以使用 CreateComponent 函数引用其他的组件。要在上面的代码中引用 COMObj2，可以使用 CreateComponent 设置对该对象的引用。

```
Set oComponent = CreateComponent("COMObj2")
```


这样就实现了对 COMObj2 的运行时引用。它的用处何在呢？就在于这样可以使您能够添加实现 ASP 接口和 DHTML 行为的组件，而同时暴露属性和方法给其他的应用程序。这样 ASP 和 DHTML 组件可以访问 COM 组件的所有属性和方法，并减少了冗余代码的数量。

但是仅依靠 Windows 脚本组件向导不能解决所有问题，它只能创建单个对象。所有对象创建之后，可以创建一个包，将各个文件中的内容复制粘贴到一个包中。

16.9 ASP 的脚本组件

ASP 脚本组件包括了 ASP 库的功能，允许支持 Web 的脚本组件。这些脚本组件是在 ASP 页面中调用的，可以在 ASP 组件和业务逻辑的代码重用方面发挥很大的作用，同时还通过从显示内容的页面分离功能代码来节约时间。

要使一个脚本组件能够支持 ASP，就必须添加一个<implements>标记和一个 ASP COM 处理程序的引用。

```
<implements type="ASP">
```

```
</implements>
```

设置了<implements>标记之后，脚本组件就有了对 ASP 的引用，可以使用 Response、Request、Session、Application 和 Server 等 ASP 对象。例如，一个组件输出当前日期和时间到 ASP 页面，该脚本组件类似于下面的代码：

```
<component id="ASPDatetimeObject">
```

```
<registration progid="ASPDatetimeObject"/>
```

```
<public>
```

```
    <method name="OutputDateTime"/>
```

```
</public>
```

```
<implements type="ASP"/>
```

```
<script language="VBScript">
```

```
<![CDATA[
```

```
Sub OutputDateTime()
```

```
        Response.Write("It is currently " & Time & " on " & Date)
    End Sub

]]>
</script>
</component>
```

下面的 ASP 页面代码创建了这个对象并调用 `OutputDateTime` 方法。

```
<html>
<head>
<title>ASP Script Objects</title>
</head>
<body>
<h1>ASP Script Objects</h1>

<%
Set objDateTime = CreateObject("ASPDatetimeObject")%>

objDateTime.OutputDateTime()

set objDateTime = nothing

%>
</body>
</html>
```

一个 ASP 脚本组件还可以包含复杂的数据库函数，这些数据库函数可以被重用于一般的数据库输出。因为脚本对象可以调用其他的 COM 组件，所以就能用于访问所有的 ADO 函数、Office COM 库，以及第三方对象。这样的功能是很强大的。

那么，ASP 组件对象如何运行呢？当 ASP 页面中调用组件对象时，脚本对象将在调用页的同名字空间(或进程空间)中运行。这就赋予脚本组件对页面的直接访问能力，因此它就能访问所有的内建 ASP 对象，输出到 ASP 的所有内容将被定向回页面。脚本对象和 ASP 页

面可以看到的是相同的对象。这与创建实现 `OnStartPage` 方法的 Visual Basic COM 组件的情形是类似的。如果一个 Visual Basic COM 组件具有这个方法，ASP 就可以自动调用它，并发送一个 ASP 库的引用，从而赋予 Visual Basic 对 ASP 的全面控制能力。

如果对 ASP 的使用非常熟悉，可能就会产生疑问，为什么这样做会比使用 `#include` 指令更好？将一个库包含进 ASP 文件时，该文件的全部内容都被合并到源文件中。例如，假设有一个包含 50 个相对复杂的函数的库，那么这种库的源代码量不上千行，也得有几百行。那么即使只是想使用 50 个函数中的 1 个，也不得不将所有的代码都添加到源文件中，而这些冗余的代码都需要被处理。那么如果碰巧由于正在处理的页面的特征，不需要使用任何函数的话，那就很糟糕了，ASP 还是要将所有被包含的文件合并进来才能处理这个页面。这是一种很低效的资源利用方式，因为这些资源可能更应该用在别处。

另一方面，一个 ASP 脚本对象也可以包含所有要使用的库函数，但只在需要时加载这些函数。如果页面的业务逻辑不需要某个函数，对应的对象就不会被加载，页面所包含的代码量就比较少，需要的处理时间也比较短，这样运行的就更快。ASP 脚本组件目前是 ASP 页面设计的更好选择，因为可以将组件和相关的函数组织在一起。对于需要使用函数的页面来说，不需要为每一个这样的页面都添加 `#include` 指导命令。还可以在中间层服务器上远程执行复杂的脚本。此外，从另一方面来说，包含文件直接在 Web 服务器上运行，无法充分利用内网和 Internet 应用程序的 n-层结构。

16.10 编译时错误检查

注册脚本组件时，还会发生一些事情，那就是脚本语法的正确性检验。如果存在脚本错误，或者 XML 无法通过正确性检验，就会收到一条错误信息。该错误信息并不很详细，只能提供一些文件中的位置信息，可能还会给出一些受错误影响的代码。

例如，在脚本中加入一个分号(假设从 JScript 转换脚本)。

```
function Add(X,Y)
```

```
    Add = X + Y;
```

```
end function
```

如果试图注册这个组件，就会弹出很多对话框，显示已经发生了一些错误，注册不成功。

错误信息中的文本将给出组件错误的大概位置(以行号加列号的形式表示)。不幸的是，这通常是不完全精确的，但还是比较接近的。此外，在 Notepad 中对行号的计数是比较困难的，更不用说列号了，此时如果能有一个可以给出行号和列号的文本编辑器，就会很方便。

编译时的错误检查目前还很不成熟，但是还是可以指出代码中错误存在的大致方向。

16.11 在 Script 组件中使用 VBScript 类

前面曾讨论过，VBScript 中可以声明类和类结构。可以在 XML 文件的数据部分中使用<script></script>标记将一个标准的 VBScript 类集成到一个 Windows 脚本组件中。这里仍使用类的标准结构。

```
class <classname>
```

```
    <internal class declaration>
```

```
end class
```

16.11.1 VBScript 类的使用限制

在 Windows 脚本组件中使用 VBScript 类只有一个主要的限制需要特别注意：类信息不会自动暴露。实质上，脚本组件并不知道内部类的结构信息。要把这个类暴露给外部实体，必须将类信息封装在方法和属性中，并声明在脚本组件文件中。

那么，为什么要在一个脚本组件中使用类呢？这是因为，类不会为一个小组件提供很多功能，但是对于一个复杂的组件来说，如果使用类，便可帮助开发人员以更有意义的方式组织对象结构。大的脚本组件因其对脚本解析的依赖而变得更复杂，所以组件的维护会变得比较困难。一个良好定义类总是会向开发人员提供更多熟悉的结构。

本章稍后会讲到，可以包含外部源文件。如果定义了很多类，可以通过源文件包含和提供 COM 封装器的方式来定义类。这里要记住的是，VBScript 类不能自动地被暴露给 COM，所以必须提供某种机制使其他对象能够访问类。

16.11.2 使用内部类

在脚本组件中，需要一个类结构和一系列封装了内部类的属性和方法。下面以本章前面曾创建的 `Math` 组件为例，将其用作类封装器。开始时，脚本组件的形式如下：

```
<?xml version="1.0"?>

<component>

<?component error="true" debug="true"?>

<registration

    description="Math"

    progid="Math.Scriptlet"

    version="1.00"

    classid="{b0a847a0-63c2-11d3-aa0e-00a0cc322d8b}"

>

</registration>

<public>

    <method name="Add">

        <PARAMETER name="X"/>

        <PARAMETER name="Y"/>

    </method>

    <method name="Subtract">

        <PARAMETER name="X"/>

        <PARAMETER name="Y"/>

    </method>

    <method name="Multiply">

        <PARAMETER name="X"/>

        <PARAMETER name="Y"/>

    </method>
```

```

        <method name="Divide">
            <PARAMETER name="X"/>
            <PARAMETER name="Y"/>
        </method>
</public>

<script language="VBScript">
<![CDATA[

function Add(X,Y)
    Add = X + Y
end function

function Subtract(X,Y)
    Subtract = X - Y
end function

function Multiply(X,Y)
    Multiply = X * Y
end function

function Divide(X,Y)
    Divide = X / Y
end function

]]>
</script>

</component>

```

在<script>标记中，可以创建一个类来处理脚本组件的方法。

```

<script language="VBScript">
<![CDATA[

```

Class clsMath

Public Function Add(X, Y)

 Add = X + Y

End Function

Public Function Subtract(X, Y)

 Subtract = X - Y

End Function

Public Function Multiply(X, Y)

 Multiply = X * Y

End Function

Public Function Divide(X, Y)

 Divide = X / Y

End Function

End Class

Private oMath

set oMath = new clsMath

Function Add(X,Y)

 Add = oMath.Add(X,Y)

End Function

Function Subtract(X,Y)

 Subtract = oMath.Subtract(X,Y)

End Function

Function Multiply(X,Y)

 Multiply = oMath.Multiply(X,Y)

End Function

```
Function Divide(X,Y)

    Divide = oMath.Divide(X,Y)

End Function

]]>

</script>
```

这样就创建了一个 VBScript 类，并将功能封装到一个脚本组件中。这给脚本组件提供了更强的灵活性，就像下面将要讨论的。

16.11.3 包含外部源文件

在文件本身中并不需要包含类声明(或其源代码)。使用在<script>标记中的声明使您能够包含一个外部源文件。使用 `src=`可以包含另一个文件，它使您能够将类声明移动到一个.vbs 文件(或.txt 文件)中以备日后使用。这样，我们就可以在 Windows 脚本宿主、ASP 和脚本组件中使用外部文件了。

对于 Math 这个范例脚本，可以将类声明移到 `math.vbs` 中。`math.vbs` 的全部文本就是整个的类定义。

```
Class clsMath

    Public Function Add(X, Y)

        Add = X + Y

    End Function

    Public Function Subtract(X, Y)

        Subtract = X - Y

    End Function

    Public Function Multiply(X, Y)

        Multiply = X * Y

    End Function

End Class
```



```
Public Function Divide(X, Y)
```

```
    Divide = X / Y
```

```
End Function
```

```
End Class
```

可以修改 **Math** 组件的文本，包含下面这个新的源文件。

```
<script language="VBScript" src="math.vbs">
```

```
<![CDATA[
```

```
private oMath set oMath = new clsMath
```

组件被实例化之后，可以分析脚本文件，在<script>标记中添加包含文件，然后继续处理。对于 COM 来说，内部类声明和外部类声明是等价的。

16.12 小结

Windows 脚本组件为 Web 页面提供了更强的灵活性，可以很紧密地集成到 ASP 代码中。这些对象可以用作独立的 COM 组件，或者让它们直接与 ASP 进行交互。只需要编写一些脚本，了解一点 XML 的运行机制，理解脚本组件向导的基础知识，简单了解 **regsvr32**，就足以使用脚本组件了。

第 21 章 在 VB 和 .NET 应用程序中添加 VBScript 代码

现在我们已经清楚，VBScript 在 Windows 中的很多场合都可以使用。毫不意外的是，和其他各种不同的技术一起，Microsoft 已经提供了能够支持 VBScript 的组件——脚本控件(Script Control)。这个 ActiveX 控件提供了一种简单的方法，使那些使用 Visual Basic(或其他支持 ActiveX 控件的语言)编写的应用程序能够提供自己的脚本环境，从而允许定制应用程序。得益于 Microsoft 在 .NET 框架中对向前兼容性的关注，通过扩展 VBScript，脚本控件也能用于 .NET 应用程序的自动化和扩展。

过去，程序员必须设法为他们的项目提供定制的特性，否则就要为其他产品支付版权费，如 Microsoft 的可移植 VB 和 Visual Basic for Applications(VBA)。1997 年，Microsoft 发布了 Windows 脚本接口(Windows Scripting Interfaces, WSI)，作为脚本引擎的接口，并最终发布了脚本控件。WSI 是为 C++ 程序员提供的，而脚本控件是为在 Visual Basic 应用程序中的应用量身定做的——它在 .NET 中也能很好地工作。

在编写第 3 版的《VBScript 程序员参考手册》时，Visual Basic 6 已经被认为是“遗留”技术了。最新的 Windows 平台的应用程序都是用 .NET 框架编写的。有人可能会问，那为什么脚本控件还是很重要呢？它发布了差不多有 10 年了，为什么在本书中作者还要专门用一章来讨论呢？简单地说吧，这有两个原因：

- 目前全世界的企业中仍然在使用百万行的 VB 6 代码。要将这些代码移植到 .NET 框架下，还要相当长的时间。脚本控件和 VBScript 实际上提供了很好的机会，可以扩展并修改现有的 VB 6 应用程序，而不需要添加或修改实际的 VB 6 代码。
- 虽然 Microsoft 的确为 .NET 和 Visual Studio for Application 提供了 Microsoft.Vsa 名字空间，作为可行的 .NET 脚本支持，但脚本控件以其简单易用的特性，仍然是用户乐于使用的技术。

21.1 为什么要在应用程序中添加脚本

允许通过脚本来定制应用程序可以提供很多机会——不仅对于开发者和程序员(可以在无需重编译和重发布的前提下修改或定制应用程序行为),对于终端用户也是如此,他们将对应用程序进行更多的操作。脚本的能力几乎是无限的,但是一般来说,给应用程序添加这种特性需要额外的时间来进行设计、编码和测试。

如果想允许从“内部”(相对于“外部”来说)定制应用程序的话,为应用程序添加脚本支持是很不错的措施。从“外部”定制应用程序(允许外部的应用程序利用代码)常常就是用户所需要的,但脚本的使用并不一定是最好的选择。对于向其他应用程序和库暴露应用程序功能来说,通过良好定义在组件库(DLL)中的 API 暴露功能一般是最好的选择。但是,如果想从内部扩展 VB 6 或.NET 应用程序的固有功能,使用脚本是最好的。

例如,考虑 Microsoft Excel,它通过脚本提供从“内部”和“外部”的定制功能。从外部定制 Excel 的需求来自于这样的事实,那就是 Excel 暴露公共的基于 COM 的编程接口,程序员可以使用其他启用 COM 的语言(像 Visual Basic、VBScript 或 VB.NET)对其进行编码,甚至都不需要打开 Excel 的图形界面。不过,也可以添加宏和 VBA 代码从 Excel“内部”定制其行为。

在 VB 6 或.NET 应用程序中,只要创建暴露公共接口的 COM 或.NET 库就可以实现从“外部”的定制。这种技术不需要任何额外的代码和测试时间,但必须对程序做特殊的设计,以某种特殊的方式隔离代码,才能实现。对很多应用程序来说,定制的级别很高,什么都能定制。

但是,也可以像 Excel 那样,使用脚本控件提供从“内部”的定制。实现这一点有很多种方案,而必须对程序的内部机制(至少要对那些被暴露给脚本的部分)做少许修改。这一章我们将讨论这些技术,并介绍一些可免费下载的,使用 Visual Basic 6 或.NET 编写的实现脚本控件的范例程序。

21.2 宏和脚本的概念

在了解脚本控件的细节之前,先来从概念上理解一下如何在应用程序中使用脚本控件是很有必要的。有两种可行的方法:

- 使用“scriptlet”(可能是存储在文本文件或数据库表中), 运行于特定的时间, 实现非常专门的目标。
- 暴露应用程序中的一大部分, 使用脚本进行定制和自动化。无论是进行定制还是自动化, 都可以选择和应用程序的主脚本共享部分或所有的应用程序内部对象模型。

这两种方法的相似之处在于, 应用程序都放弃了对某些业务逻辑和代码段的控制, 把它交给了运行时加载的脚本(相对于编译到应用程序中的脚本而言)。但就其作用范围来说两种方法是很不相同的。下面我们来看两个示例。

21.2.1 使用 Scriptlet

首先我们通过一个小例子进行初步的学习。假设一个应用程序有一个编码复杂的算法, 因为有一系列涉及很多过程/很多类的步骤。假设这个复杂的算法是一个记账系统的一部分, 用于计算一些发票的总金额。每张发票都有一系列的计算公式, 根据所占发票总金额的比例, 每个公式有不同的输入和输出。假设计算一张发票的金额需要 10 个计算公式, 如果其中 9 个公式是静态的, 那就意味着它们是不随情况变化的, 可对它们进行硬编码。

但是, 对于另外的那个公式, 根据发票类型的不同, 计算方法不同。有时这个公式必须按照一种方法来计算, 有时又必须按照另一种方法计算。并且, 使用应用程序的公司又随时有可能增加新的发票类型, 而每种发票都对这个不同的公式有不同的要求。并且, 公司希望在添加新的发票类型的同时, 不用为每一种发票添加惟一的、手工编码的计算公式。他们也不希望在每次添加一种新的发票类型后都重新部署和编译应用程序。

应用程序设计者决定使用脚本控件来解决这一困难。他们在数据库的 InvoiceType 表中增加一列, 名为 FormulaScript。设计者使用这一列存储用 VBScript 编写的 scriptlet, 用于根据每种发票类型以不同的方式计算这个不同的公式。每个 scriptlet 基于各种发票类型的需求来设计。如果发票总金额计算算法运行到第 10 个公式, 就会根据 InvoiceType 表格加载对应的 scriptlet, 并使用脚本控件动态执行这个 scriptlet。

21.2.2 使用脚本

在第一个示例中我们已经看到，应用程序使用脚本功能来实现非常专门的目的，添加满足需要的定制能力。对于使用范围更广的应用场景，可能会有一大部分应用程序开放给脚本进行自动化。应用程序使用的脚本可能非常复杂，如我们曾经介绍过的 **Windows** 脚本宿主(WSH)和 **ASP** 脚本。脚本可能会访问主应用程序的所有对象模型，主要的途径就是通过 **WSH** 和 **ASP** 引擎向脚本暴露 **WScript** 和 **Request** 这样的对象。

这种使用范围更广的方法可能使您能够编写自己的宏来控制应用程序的用户界面(假设可以向用户脚本暴露菜单项、工具栏按钮、文本框，以及其他表单控件)。或者，应用程序可能暴露某个对象模型，使您能够以插件的形式编写自己的脚本来产生报告。

21.2.3 选择最佳的使用范围

使用范围大好还是小好呢？这其实是很难确定的，因为有很多的可能情况。如果想为终端用户提供定制能力，就最好使用脚本控件，因为可能需要允许在应用程序中创建并编辑脚本，而又无法控制如何添加以及添加什么样的脚本到应用程序。或者可能应用程序被部署到客户的站点，部署顾问(可能不是程序设计的专家)就需要一定的灵活度来定制该客户的应用程序，可能就在客户的办公室中安装该应用程序。

但是，如果试图实现某个设计，只允许使用“插件”组件简化部署过程中的 **bug** 修正，为产品环境增加新的特性，那么基于对设计选项以及一些可能的研究成果的进一步考虑，利用二进制接口和类工厂设计模式的面向对象多态设计可能是更好、更稳定、可预见的技术选择。

这看上去好像很啰嗦。不幸的是，这里我们没有更多的篇幅来解释什么是“利用二进制接口和类工厂设计模式的面向对象多态设计”了。我们讨论的重点是脚本控件，不是每种设计问题的解决方法。我们应该仔细考虑“为什么”要使用脚本控件，要实现的需求是什么？可能还存在其他的解决方案，能够完成同样的工作，又可以避免脚本控件的缺点，那就是相对于应用程序本来包含的编译过的代码来说，脚本代码的运行总是会慢一些，而又无法控制可能被引入到脚本中间的语法错误和不好的编程风格。

21.3 在 VB 和 .NET 应用程序中添加脚本控件

如果还没有下载脚本控件，那么可以从 Microsoft 的 Web 站点上免费下载。安装程序将自动将控件添加到机器中，并进行注册。此时可能会发现机器中已经安装了脚本控件。不知从何时起，Microsoft 开始在 Windows 的默认安装过程中包含脚本控件的安装。在机器中搜索文件 msscript.ocx 就可以确定是否已经安装了脚本控件(这个文件很可能是在 C:\Windows\System32 下面)。

只要做好了在应用程序中使用脚本控件的准备工作(这里读者可以先阅读一下后面的内容，浏览一下这一章中的可下载的工程范例)，就可以很容易地将脚本控件作为一个(附加到表单中的)ActiveX 控件或程序中已经声明并实例化的普通 COM 对象添加到 VB 6 或 .NET 的 Windows Forms 工程中。在范例工程的注释和代码中，可以看出将脚本控件作为一个组件添加到表单中和仅仅通过实例化 ScriptControl 对象并直接使用之间的区别。在 .NET 和 VB 6 中，这两种机制都能很好地工作。

一般来说，使用脚本控件作为一个附加到表单上的组件是一个相对简单的方法，因为不需要费精力去控制对象的生命周期(它总是作为表单的一个对象而存在，就像按钮或定时器控件那样)。但是，并不限制使用表单的方式——在上述情况下，完全可以实例化代码中的 ScriptControl 内联对象。

在 VB 6 的集成开发环境中，可以从 Component Toolbox 中添加一个脚本控件到表单上，就像 CommandButton 或 Timer 这样的表单控件那样。如果喜欢直接在代码中实例化对象，或者解决方案中不使用表单，就可以使用 Project ⇄ References 菜单项添加对脚本对象的引用。这个引用称为“Microsoft Script Control 1.0”。

类似地，在 .NET 中可以在 Visual Studio IDE 中将脚本控件添加到 Component Toolbox(使用右键菜单中的“Add Items”菜单项)，或使用 Add Reference 对话框添加一个普通的引用。脚本控件位于 COM 标签页中，名为“Microsoft Script Control 1.0”。

对于 .NET 的情况，.NET 框架在很大程度上屏蔽了脚本控件是一个 COM 对象这一事实，同时也隐藏了能够将 .NET 对象作为 COM 对象实现与脚本的共享这一事实。但是在某些情况下，只要对 .NET-COM 互操作的相关问题有个初步的了解就可以了。

互操作在有些情况下会比较复杂，比如与脚本控件中的 VBScript 代码共享聚合的 .NET

对象的情形。但即使在这种情况下，只要检查 Assembly Information 对话框(可以从 Visual Studio 中的 project 属性窗口的 Application 选项卡打开)，确定是否选中 Make Assemble COM-Visible 复选框，大多都会使问题简单化。

21.4 脚本控件参考

本章中间的部分将给出脚本控件的参考，包括对象、集合、属性、方法、方法语法和示例。参考之后，这一章将继续讨论一些额外的问题，然后介绍一些可下载的使用脚本控件的 Visual Basic 6 和 VB.NET 应用程序。代码和语法示例是基于 VB 6 的；要获得等效的语法信息，请参考.NET 语言(如 VB.NET 或 C#)的文档。从这一章中的 VB.NET 范例工程中可以获得与.NET 相关的指导信息。

21.4.1 对象模型

脚本控件对象模型的描述如图 21-1 所示。对象及其属性和方法的细节将在后续的章节中详细讨论。

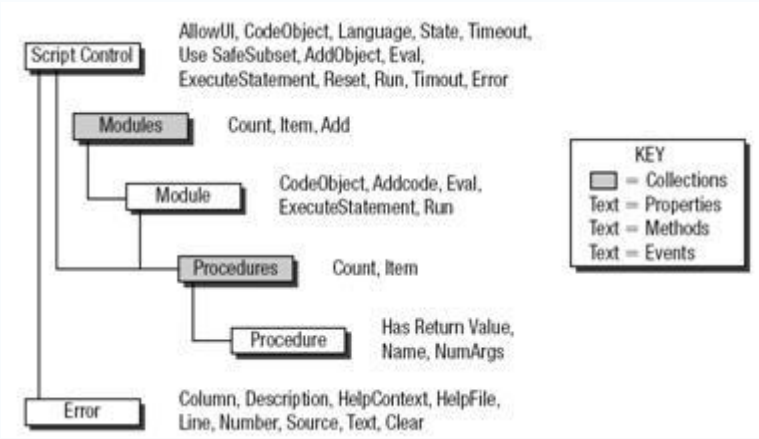


图 21-1

21.4.2 对象与集合

脚本控件组件有几个对象和集合(这是一种特殊的多值对象)，它们协同工作，提供了大量功能用于在 Visual Basic 应用程序中添加脚本。对于每个对象和集合，这一节中都给出对象的一般描述，并详细介绍其属性、方法和事件。在适当的时候，给出示例代码。

1. ScriptControl 对象

ScriptControl 是应用程序启用脚本功能的关键元素。它提供了简单的接口来包含脚本引擎，如 VBScript 或 JScript。所有其他可用的对象都依靠 ScriptControl 对象的示例。ScriptControl 对象可以使用下面三种不同的方法实例化：

- 前期绑定，在表单上进行(使用 Components 对话框添加)
- 前期绑定，通过代码完成(使用 Reference 对话框进行)
- 后期绑定(任何时候都可以进行)

声明的语法

声明一个 ScriptControl 对象的前期绑定变量使用的是 VB 6 的语法(这也是后期绑定比较常用的语法)。

```
Dim|Private|Public [WithEvents] objSC As [MSScriptControl.]ScriptControl
```

下面的语法声明了一个 ScriptControl 对象的后期绑定变量。后期绑定变量不能处理事件。如果使用后期绑定变量，就不需要在项目中引用脚本控件。

```
Dim|Private|Public objSC [As Object|Variant]
```

属性

ScriptControl 对象的属性描述如表 21-1 所示。

表 21-1

名 称	输入值/返回值	访 问 方 式	描 述
AllowUI	Boolean	读/写	设置或返回一个值，表示是否显示像 MsgBox 或 InputBox 这样的可视化元素。如果该属性被设置为 False，那么与界面进行可视化交互的唯一方法就是直接通过主应用程序进行
Error	Error 对象	只读	返回 ScriptControl 实例的 Error 对象的引用
Language	String	读/写	设置或返回 ScriptControl 对象所使用的脚本语言的名称。“VBScript”和“JScript”都是本来就支持的语言。如果安装了其他兼容的脚本语言，就可以使用这些脚本语言的名称。设置这个属性会重置 ScriptControl 对象及其子对象的所有其它成员
Modules	Modules 对象	只读	返回 ScriptControl 对象的 Modules 集合的引用
Name	String	只读	如果 ScriptControl 对象附加到一个表单，该属性就返回控件属性页上为其指定的名称
Procedure	Procedure 对象	只读	返回默认的“Global”模块的 Procedure 集合的引用。要

			访问其它模块的 Procedure 集合，可以通过 Modules 集合直接访问该模块，然后使用 Module.Procedure 属性
SitehWnd	Long	读/写	设置或返回用于执行代码的父窗口的“句柄”。当脚本控件被用作 ActiveX 控件时，它会被放到表单上，那么 SitehWnd 的默认值就是控件的容器的 hWnd ，否则，如果脚本控件被用作一个对象(不附加到表单)， SitehWnd 就是 0 ，这个值对应的是桌面。这个属性会影响到哪个窗口(或控件)具有对脚本控制的 UI 元素的 UI 控制权。修改这个属性可以使脚本控件依赖一个具体的窗口，而不是像在某些情况下那样只依赖于桌面(例如，可能想让脚本控件冻结一部分应用程序，而不是桌面)。需要修改这一属性的场合并不多

(续表)

名 称	输入值/返回值	访 问 方 式	描 述
State	Long(States)	读/写	设置或返回 ScriptControl 对象的模式，使用 States 枚举常量。如果这个值被设置为 Connected(1) ，那么 ScriptControl 对象就能够接收通过 AddObject 方法添加的对象所产生的事件。这样，对状态的更改使您能够控制对事件的处理过程
TimeOut	Long	读/写	设置或返回一个数字，代表以毫秒为单位的时间，表示 ScriptControl 对象在终止长时间运行的脚本之前需要等待的时间长度。这个属性可以被设置为常量 NoTimeout(-1) ，这样脚本代码的执行就没有时间限制；但是关闭超时机制是比较危险的，因为可能会有人创建一个包含死循环的脚本。默认的值是 10 000ms(10s) 。超时时间过后，会产生一个 Timeout 事件(这取决于 ScriptControl 对象是否能处理事件)，同时，如果 ScriptControl 对象启用了 AllowUI 属性，就会收到一个对话框警告，允许选择继续执行脚本，否则，脚本将中止执行并产生错误。如果这个属性被设置为 0 (不推荐这样做)，只要脚本停止发送 Windows 消息超过 100ms ，就会立即产生 Timeout 事件
UseSafeSubset	Boolean	读/写	设置或返回一个 Boolean 值，表示脚本控件是否可以运行未被标记为“脚本安全”的组件。例如，一个脚本可能会试图使用脚本运行时对象 FileSystemObject ，而这不是一个“脚本安全”的对象，因为它允许访问文件系统。如果担心脚本可能在客户端机器上进行危险活动，就可以将这个属性设置为 True 。如果在要求组件必须是“脚本安全”的宿主(如 Internet Explorer)上使用脚本控件，这个属性默认就为 True ，并且是只读的

方法

ScriptControl 对象的方法描述如表 21-2 所示。

表 21-2

名 称	参 数	返 回 值	描 述
AddCode	code: 字符串值, 表示一个合法的脚本	无	这是向脚本控件添加脚本的主要方法。在 ScriptControl 对象上调用该方法时, 会在默认的“Global”模块中自动添加新的代码。在单独的 Module 对象上调用该方法时, 会在具体涉及的模块中添加代码。添加整个过程的代码或一个代码块时, 必须在一次调用 AddCode 方法时添加所有的代码。代码块中的语句之间可以使用冒号(:)隔开, 或使用换行符 vbNewLine (推荐使用)、 vbCr 、 vbLf 和 vbCrLf
AddObject	Name: 要添加对象的唯一的字符串名称 object: 应用程序范围内的任意对象 addmembers: 可选的布尔值, 表示对象的公有成员是否能够被 ScriptControl 对象及其脚本访问	无	允许脚本访问宿主的运行时对象模型, 因为本方法添加的对象将暴露这些对象。添加到 ScriptControl 的对象在 ScriptControl 中是全局可访问的。可选的 addmembers 参数表示已添加对象的成员是否也可以在脚本内部访问
Eval	Expression: 一个字符串值, 代表一个合法的“表达式”, 表示被编译或执行的脚本片段	表达式的输出(如果有的话)	计算一个表达式的值。与 VBScript 中的 Eval 函数类似。这是计算您提供的动态表达式最好的方法。如果拿 Eval 和 ExecuteStatement 方法相比较, 会发现“=”操作符在使用 Eval 时被认为是比较操作符, 而在使用 ExecuteStatement 方法时被认为是赋值操作符。因此, x=y 在使用 Eval 时得到的是布尔值, 而使用 ExecuteStatement 方法时, y 的值将被赋给 x , 而没有返回值。被计算的表达式可以利用 Module 或 ScriptControl 对象范围内的任何成员
(续表)			
名 称	参 数	返 回 值	描 述
ExecuteStatement	statement: 字符串值, 表示要执行的语句	无	与 Eval 方法不同, ExecuteStatement 只执行一个语句, 不返回任何值。如

			果拿 Eval 和 ExecuteStatement 方法相比较, 会发现“=”操作符在使用 Eval 时被认为是比较操作符, 而在使用 ExecuteStatement 方法时被认为是赋值操作符。因此, x=y 在使用 Eval 时得到的是布尔值, 而使用 ExecuteStatement 方法时, y 的值将被赋给 x, 而没有返回值。被计算的表达式可以利用 Module 或 ScriptControl 对象范围内的任何成员。要获取一个过程的返回值, 应该使用 Eval 或 Run 方法
Reset	无	无	丢弃 ScriptControl 对象的所有成员和子对象, 将它们初始化为默认状态。调用 Reset 方法时, State 属性被设置为 Initialized(0)
Run	procedurename: 要运行的过程或函数的字符串名称 paramarray(): 可选的参数数组, 其中包括过程或函数所接受的参数值	如果调用函数, 就返回该函数的返回值	在 ScriptControl 对象上调用 Run() 时, 该方法将试图运行“Global”模块中具有给定名称的过程或函数。在一个 Module 对象上调用 Run() 时, 该方法试图运行模块内具有给定名称的过程或函数。也可以使用 CodeObject 属性直接调用过程和函数

事件

ScriptControl 对象的事件描述如表 21-3 所示。

表 21-3

名 称	参 数	描 述
Error	无	ScriptControl 对象遇到脚本运行错误时会发生该事件。要接收这个事件的通知, 必须声明 ScriptControl 对象的“前期绑定”变量, 并使用 WithEvents 关键字

(续表)

名 称	参 数	描 述
Timeout	无	脚本执行时间超过 Timeout 属性的值时, 或您决定停止脚本执行时发生该事件。如果有多个 ScriptControl 对象, 那么只有在第一个 ScriptControl 对象超时时才会发生 Timeout 事件

示例

下面这行代码显示了如何实例化一个新的前期绑定脚本控件变量。

```
Set objSC = New [MSScriptControl.]ScriptControl
```

下面这行代码显示了如何实例化一个新的后期绑定脚本控件变量。

```
Set objSC = CreateObject("[MSScriptControl.]ScriptControl")
```

下面的脚本片段显示了如何使用 `AddCode` 方法添加某过程或函数之外的变量和过程的步骤。整个过程或函数都应该在一次调用之内就添加完毕。

```
strCode = "Option Explicit" & vbNewLine & vbNewLine
```

```
objSC.AddCode strCode
```

```
strCode = "Dim x, y" & vbNewLine & vbNewLine
```

```
objSC.AddCode strCode
```

```
strCode = "x = 15" & vbNewLine & "y = 2"
```

```
objSC.AddCode strCode
```

```
strCode = "Function MultiplyXY(): MultiplyXY = x * y : End Function"
```

```
objSC.AddCode strCode
```

`Eval` 函数允许在运行时执行代码片段。`Eval` 简单且高效，能够实现 VB 中几乎所有的任务。

```
MsgBox objSC.Eval(InputBox$( _  
    "Enter Numeric Expression", _  
    "Eval Example", "5 * 3 - 1"))
```

基于过程的类型，可以以不同方式调用 `Run` 方法，这取决于返回值和参数的情况。

```
strCode = "Sub TwoArg(a,b): MsgBox CInt(a + b)" & " : End Sub"
```

```
objSC.AddCode strCode
```

```
objSC.Run "TwoArg", 1, 2
```

```
strCode = "Function ManyArg(a,b,c,d): ManyArg = a * b + c - d"
```

```
strCode = strCode & " : End Function"
```

```
objSC.AddCode strCode
```

```
lngResult = objSC.Run("ManyArg", 1, 2, 3, 4)
```

下面的脚本片段演示了 Error 事件的使用：

```
Private WithEvents objSC As ScriptControl
```

```
Private Sub Main()
```

```
    Set objSC = New ScriptControl
```

```
    ...
```

```
    objSC.Run "MyProc"
```

```
End Sub
```

```
Private Sub objSC_Error
```

```
    MsgBox "Script error occurred:" & vbNewLine & _
```

```
        "Number: " & objSC.Error.Number & vbNewLine & _
```

```
        "Description: " & objSC.Error.Description & vbNewLine & _
```

```
        "Line: " & objSC.Error.Line & vbNewLine & _
```

```
        "Column: " & objSC.Error.Column & vbNewLine & _
```

```
        "Script Text: " & objSC.Error.Text
```

```
End Sub
```

下面的脚本片段演示了 Timeout 事件的使用：

```
Private WithEvents objSC As ScriptControl
```

```
Private Sub Main()
```

```
    Set objSC = New ScriptControl
```

```
    ...
```

```
    objSC.Timeout = 10000
```

```
    objSC.Run "MyProc"
```

```
End Sub
```

```
Private Sub objSC_TimeOut
    MsgBox "The script has timed out."
End Sub
```

2. Module 对象

Module 对象是 Modules 集合(在本章前面的内容中曾经介绍过)中的一个成员，它包含脚本中使用的过程、类型和数据声明。脚本控件有一个默认的 Global 模块，它是自动被使用的，除非存在对已添加的其他模块的具体的成员调用。可以使用 AddCode 方法向 Module 对象中添加代码。另一方面，单独的 Module 对象是使用 Modules 集合的 Add 方法来添加的。因为每个模块中的代码都是在模块范围内私有的，所以各个模块可以使用重复的变量和模块名称。如果有很多类似的脚本，每个模块中有些许不同，这种机制就是很有用的。

声明语法

下面是声明 Module 对象的一个变量的语法。

```
Dim|Private|Public objModule [As [MSScriptControl.]Module|Object]
```

属性

Module 对象的属性描述如表 21-4 所示。

表 21-4

名 称	返 回 值	访 问 方 式	描 述
CodeObject	对象	只读	返回一个可以用于调用一个 Module 对象的公有过程和函数的对象。这是一个后期绑定对象，但它的作用在于允许在脚本中直接调用过程，而无需使用 Run 方法。模块中的过程或函数将作为对象的公有方法被该属性返回
Name	字符串	读/写	Module 对象的逻辑名称。也可以用作其在 Modules 集合中的索引，所以它在该集合中必须是唯一的。如果在集合中添加了一个名字相同的 Module 对象，新的对象就会覆盖旧的对象
Procedure	Procedures 对象	只读	返回 Module 对象的 Procedures 集合的引用

方法

Module 对象的方法描述如表 21-5 所示。

表 21-5

名 称	参 数	返 回 值	描 述
Eval	it expression: 一个字符串值, 代表一个合法的“表达式”, 表示被编译或执行的脚本片段	表达式的输出(如果有的话)	<p>计算一个表达式的值。与 VBScript 中的 Eval 函数类似。这是计算您提供的动态表达式最好的方法。</p> <p>如果拿 Eval 和 ExecuteStatement 方法相比较, 会发现“=”操作符在使用 Eval 时被认为是比较操作符, 而在使用 ExecuteStatement 方法时被认为是赋值操作符。因此, x=y 在使用 Eval 时得到的是布尔值, 而使用 ExecuteStatement 方法时, y 的值将被赋给 x, 而没有返回值。</p> <p>Eval 方法可以应用于 ScriptControl 对象和 Module 对象, 并利用其所有成员</p>

(续表)

名 称	参 数	返 回 值	描 述
AddCode	code: 字符串值, 表示一个合法的脚本	无	<p>这是向脚本控件添加脚本的主要方法。在 ScriptControl 对象上调用该方法时, 会在默认的“Global”模块中自动添加新的代码。在单独的 Module 对象上调用该方法时, 会在具体涉及的模块中添加代码。添加整个过程的代码或一个代码块时, 必须在一次调用 AddCode 方法时添加所有的代码。代码块中的语句之间可以使用冒号(:)隔开, 或使用换行符 vbNewLine(推荐使用)、vbCr、vbLf 和 vbCrLf</p>
ExecuteStatement	statement: 字符串值, 表示要执行的语句	无	<p>与 Eval 方法不同, ExecuteStatement 只执行一个语句, 不返回任何值。如果拿 Eval 和 ExecuteStatement 方法相比较, 会发现“=”操作符在使用 Eval 时被认为是比较操作符, 而在使用 ExecuteStatement 方法时被认为是赋值操作符。因此, x=y 在使用 Eval 时得到的是布尔值, 而使用 ExecuteStatement 方法时, y 的值将被赋给 x, 而没有返回值。</p> <p>被计算的表达式可以利用 Module 或 ScriptControl 对象范围内的任何成员。要获取一个过程的返回值, 应该使用 Eval 或 Run 方法</p>
Run	procedurename: 要运行的过程或函数	如果调用函数, 就返回该	<p>在 ScriptControl 对象上调用 Run()时, 该方法将运行“Global”模块中具有给定名称的过</p>

	的字符串名称 paramarray() : 可选 的参数数组, 其中 包括过程或函数所 接受的参数值	函数的返回 值	程或函数。在一个 Module 对象上调用 Run() 时, 该方法运行模块内具有给定名称的过程 或函数。也可以使用 CodeObject 属性直接调 用过程和函数
--	---	------------	--

示例

下面的脚本片段演示了使用 **Module.Run** 方法调用模块中包含的一个过程。

```
Set objModule = objSC.Modules.Add("NewModule")
objModule.AddCode "Sub Test(): " & _
    vbNewLine & vbTab & "MsgBox ""Hello, world."" & _
    vbNewLine & "End Sub"
objModule.Run "Test"
```

下面的脚本片段演示了使用 **Module.CodeObject** 方法调用模块中的代码。我们可以看到, 按照这种方法调用过程是很自然的, 并且相对于使用 **ScriptControl.Run** 方法来说, 可能具有更强的可读性, 这是因为过程被暴露为 **CodeObject** 对象的方法。

```
Set objModule = objSC.Modules.Add("TestMod")
objModule.AddCode "Sub TestProc(): " & _
    vbNewLine & vbTab & "MsgBox ""Hello, world."" & _
    vbNewLine & "End Sub"
objModule.AddCode "Function TestFunction(a) : & _
    vbNewLine & vbTab & "TestFunction = a * a " & _
    vbNewLine & "End Function"
Set objCodeObject = objModule.CodeObject
objCodeObject.TestProc
lngVal = objCodeObject.TestFunction(2)
```

下面的脚本片段显示了添加某过程或函数之外的变量和过程的步骤。整个过程或函数都应该在一次调用之内就添加完毕。


```

strCode = "Option Explicit" & vbNewLine & vbNewLine
objModule.AddCode strCode

strCode = "Dim x, y" & vbNewLine & vbNewLine
objModule.AddCode strCode

strCode = "x = 15" & vbNewLine & "y = 2"
objModule.AddCode strCode

strCode = "Function MultiplyXY(): MultiplyXY = x * y : End Function"
objModule.AddCode strCode

```

Eval 函数允许在运行时执行代码片段。Eval 简单且高效，能够实现 VB 中几乎所有的任务。

```

MsgBox objSC.Eval(InputBox$( _
    "Enter Numeric Expression", _
    "Eval Example", "5 * 3 - 1"))

```

基于过程的类型，可以以不同方式调用 Run 方法，这取决于返回值和参数的情况。

```

strCode = "Sub TwoArg(a,b): MsgBox CInt(a + b)" & " : End Sub"
objSC.AddCode strCode
objSC.Run "TwoArg", 1, 2

strCode = "Function ManyArg(a,b,c,d): ManyArg = a * b + c - d"
strCode = strCode & " : End Function"
objSC.AddCode strCode
lngResult = objSC.Run("ManyArg", 1, 2, 3, 4)

```

3. Modules 集合

Modules 集合包含一个 ScriptControl 对象的所有 Module 对象，包括默认的 Global 模块。对 Global 模块的成员的调用可以直接通过 ScriptControl 对象来进行，而不需要遍历 Modules

集合。Global 对象还具有一个值与常量 GlobalModule 相等的索引。

Module 对象可以使用 Add 方法被添加到 Modules 集合中。使用默认的 Modules.Item 方法可以访问具体的 Module 对象。Count 属性提供了集合中 Module 对象的个数。整个集合可以通过几种方式来遍历，最常用的是使用 For Each ... Next 循环。因为不能删除单个模块，所以必须使用 ScriptControl 对象的 Reset 方法来删除不想要的模块，该方法将清空整个集合。

属性

Modules 集合唯一的属性描述如表 21-6 所示。

表 21-6

名 称	返 回 值	访 问 方 式	描 述
Count	Long	只读	返回 Modules 集合中 Module 对象的个数

方法

Modules 集合对象的方法描述如表 21-7 所示。

表 21-7

名 称	参 数	返 回 值	描 述
Add	name: 字符串值，表示要添加模块的名称；该名称将用作在 Modules 集合中的索引 module: 可选。要被加入集合的 Module 对象	如果省略 module 参数，就返回新的 Module 对象	使用这个方法可以在 Modules 集合中添加一个新的 Module 对象；如果在一个小型的工程中，只包含相对较少的脚本，可能就只需要使用“Global”模块。但是如果有很多脚本，那么将它们分别组织到各个模块中是很有好处的——尤其是如果需要在不同的模块中重复使用具有相同名称的过程和函数时，就更应该这样做
Item	index: 一个 Long 或 String 类型的值，分别表示一个索引或一个关键字	如果存在匹配的 index，就返回集合中的一个 Module 对象	这是集合的默认属性，所以很多程序员都会忽略 item 方法的实际名称：Set objModule = objSC.Modules("MyModule")

示例

下面的代码显示了如何直接访问 Global 模块。使用同样的语法，不同的模块名，就可以访问其他的模块。

```
Set objModule = sc.Modules("Global")
```

下面的脚本片段演示了如何遍历 Modules 集合：

```
For Each objModule In objSC.Modules
```

```
    strModuleList = strModuleList & vbNewLine & objModule.Name
```

```
Next
```

模块允许使用独立的脚本并提供独立的名字空间。下面的脚本片段显示了如何在两个不同的模块中包含同名的脚本：

```
' Add code to separate modules, using same sub names.
```

```
Set objModule = objSC.Modules.Add("Maine")
```

```
objModule.AddCode "Sub ShowState" & _
```

```
    vbNewLine & vbTab & "MsgBox ""In Maine"" & _
```

```
    vbNewLine & "End Sub"
```

```
Set objModule = Nothing
```

```
Set objModule = objSC.Modules.Add("Ohio")
```

```
objModule.AddCode "Sub ShowState" & _
```

```
    vbNewLine & vbTab & "MsgBox ""In Ohio"" & _
```

```
    vbNewLine & "End Sub"
```

4. Procedure 对象

Procedure 对象定义了代码的一个逻辑单元，在 VBScript 中可以是一个 Sub 或一个 Function。Procedure 对象包含了很多有用的属性，使您能够检查过程名、参数个数以及过程是否具有返回值。脚本代码的入口也是由 Procedure 对象提供的。

声明语法

下面是声明 Procedure 对象的一个变量的语法。

```
Dim|Private|Public objProc [As [MSScriptControl.]Procedure|Object]
```

属性

Procedure 对象的属性描述如表 21-8 所示。

表 21-8

名 称	返 回 值	访 问 方 式	描 述
hasReturnValue	Boolean	只读	返回过程是否有返回值(换句话说,是一个过程还是一个函数)
Name	String	只读	Procedure 对象的名称,应该和对象中包含的某个实际的过程或函数名相同。这个名称还用作在 Procedures 集合的索引。如果使用 AddCode 方法在模块中添加另一个同名过程,新的过程就会覆盖模块中原来的过程
NumArgs	Long	只读	返回 Procedure 对象中过程或函数接收参数的个数

方法

Procedure 对象的方法描述如表 21-9 所示。

表 21-9

名 称	参 数	返 回 值	描 述
Item	index: 一个 Long 或 String 类型的值,分别表示一个索引或一个关键字	如果存在匹配的 index, 就返回集合中的一个 Module 对象	这是集合的默认属性,所以很多程序员都会忽略 item 方法的实际名称: Set objProc = objMod.Procedure("MyProc")

注意,Procedures 集合没有 Add 方法,但可以使用 AddCode 方法给模块添加新的方法,这样就会隐式地创建一个 Procedure 对象并添加到集合中。

5. Procedures 集合

Procedures 集合中保存着某一给定的 Modules 对象的所有过程。它为遍历某一给定模块的所有过程并访问其内部代码提供了一种便捷的途径。每个过程都是使用 Module 对象的 AddCode 方法添加到集合中,而不是直接使用集合操作来进行的。同样,某个过程被添加到集合中之后,也无法将其删除,因为 Procedures 集合没有 Remove 方法。

属性

Procedures 集合对象唯一的属性描述如表 21-10 所示。

表 21-10

名 称	返 回 值	访 问 方 式	描 述
Count	Long	只读	返回 Procedures 集合中 Procedure 对象的个数

方法

Procedures 集合对象没有任何方法。

示例

下面的脚本片段使用 For Each ... Next 循环语法遍历 Procedures 集合：

```
For Each objProcedure In objModule.Procedures
```

```
    strList = strList & "Name: " & objProcedure.Name
```

```
    strList = strList & vbNewLine & vbTab
```

```
    strList = strList & "Argument Count: " & objProcedure.NumArgs
```

```
    strList = strList & vbNewLine & vbTab
```

```
    strList = strList & "Has Return: " & objProcedure.HasReturnValue
```

```
    strList = strList & vbNewLine & vbNewLine
```

```
Next
```

6. Error 对象

Error 对象提供与脚本控件相关联的语法和运行时错误信息。虽然 Error 对象提供的信息和 VB、VBScript 中 Err 对象提供的信息类似，但在诊断脚本相关的问题时，有一些很有价值的额外属性(Column、Text、Line)。虽然在 VB 中也能够声明和初始化 Error 对象，但是一般还是通过 ScriptControl 对象直接访问 Error 对象的成员。

与 Err 对象不同，Error 对象的作用域不是全局的，它只处理与某个 ScriptControl 对象实例相关联的错误。Error 对象在每次更改 ScriptControl.Language 属性，或调用 ScriptControl 对象的 Reset、AddCode、Eval、ExecuteStatement 或 Clear 时方法都会被重置。使用 Clear 方法可以显式地重置 Error 对象的属性。脚本内部处理的运行时错误不会在应用程序级产生。

“通过脚本控制来处理错误”这一节提供了关于错误处理策略的详细信息。如果需要对 VBScript 的错误处理机制有个初步的了解，那么第 6 章“错误处理和调试”也是一个很好的参考。

属性

Error 对象的属性描述如表 21-11 所示。

表 21-11

名 称	返 回 值	访 问 方 式	描 述
Column	Long	只读	返回一个列号，表示添加脚本代码时发生的语法错误的位置
Description	String	只读	返回脚本错误的描述信息
HelpContext	Long	只读	如果错误发生的脚本有可用的帮助文件(可能性不大)，这个属性就返回帮助文件中包含错误信息的部分的标识
HelpFile	String	只读	如果错误发生的脚本有可用的帮助文件(可能性不大)，这个属性就返回帮助文件的名称
Line	Long	只读	返回一个行号，表示添加脚本代码时发生的语法错误的位置
Number	Long	只读	返回脚本错误的错误号
Source	String	只读	返回脚本错误产生的源文件名
Text	String	只读	返回一个字符串，包含产生语法错误的那一小段代码。如果允许在应用程序中添加或编辑脚本，就可以使用这个属性再加上 Description 、 Line 和 Column 帮助您了解如何改正语法错误。在调试脚本时也可以使用这个属性

方法

Error 对象唯一的方法描述如表 21-12 所示。

表 21-12

名 称	参 数	返 回 值	描 述
Clear	无	无	重置 Error 对象的所有属性。这个方法是在 ScriptControl.Language 属性发生变化或调用 Reset 、 AddCode 、 Eval 、 ExecuteStatement 方法时隐式调用的

21.4.3 常量

下面的具名常量和枚举常量可以在引用脚本控件的工程中使用。这些常量在引用脚本控件的 Visual Basic 应用程序中是全局可访问的。对于每个常量，这一节都将介绍其类型、值和使用的场合。

1. GlobalModule 具名常量

类型: String

值: "Global"

通过使用支持多个模块的脚本引擎(如 VBScript 或 JScript)使用脚本控件时, 可以使用 GlobalModule 常量访问 ScriptControl.Modules 集合中默认的 Global 模块。

2. NoTimeout 具名常量

类型: Long

值: -1

这个常量可以用于设置 ScriptControl 对象的 Timeout 属性, 防止脚本执行超时。更多细节请参看 ScriptControl.Timeout 属性(本章前面的内容中曾经讨论过)。

3. ScriptControlState 枚举常量

这个枚举常量是和 ScriptControl.State 属性一起使用的。State 属性的目的在于控制 Add Object 加入到 ScriptControl 中的对象的事件产生。默认值 Initialized(0)表示 ScriptControl 不会响应这些对象产生的事件。其他可能的值, 如 Connected(1)表示 ScriptControl 会响应发生的事件。

21.5 脚本控件错误处理

错误处理不容忽视, 尤其是在处理来自多处的代码时。对于动态生成的脚本, 和您输入的表达式更是这样。要处理错误, 就必须同时使用 VB 的 Err 对象和脚本控件的 Error 对象。如果运行脚本控件的多个实例, 那么每个实例都有一个独立的 Error 对象。发生错误时, 如果有适当的策略来处理错误, 通常就会清楚这个错误并继续执行程序。在脚本(尤其是从文件中加载的脚本)中应尽量使用所有可能的错误处理技术, 并尽可能在内部完成处理。

根据 VB 的设置情况, 错误处理程序在调试模式(在集成开发环境的 General Options 标签页中勾选 Break on Unhandled Errors)下可能无法正常工作。此外, 脚本中的错误处理

程序将依赖于 Internet Explorer 中 Disable Script Debugging 选项的设置，以及调试器的使用。脚本错误会自动调用调试器，绕过错误处理代码。参看第 6 章可以获得关于脚本调试的更多信息。

在.NET 中，未处理的错误会作为异常传播到.NET 层，异常可以被捕捉到，并使用标准的.NET 错误处理机制，如 Try ... Catch 块来处理。

脚本控件在设置全局属性时可能产生的几类错误如表 21-13 所示。

表 21-13

错 误	描 述
脚本仍在运行，但无法继续执行	在脚本运行时试图修改 ScriptControl 对象的某个成员
不能设置 UseSafeSubset 属性	包含脚本控件的应用程序强制进入安全模式
脚本执行超时	脚本执行时间超过了 Timeout 属性设置的值而停止
未设置语言属性	某些属性只能在设置了 Language 属性之后才能设置
选定的脚本引擎不支持某成员	使用除 VBScript 和 JScript 之外的语言是时，并不支持所有的属性和方法
对象不再合法	重置脚本控件时(调用 Reset 方法或修改 Language 属性时会 导致重置)，对象已经先被释放了

这些错误一般都可以通过仔细的编程来避免，对于错误处理策略来说并不是什么大问题。只有在下述两种情况下错误才会真正带来麻烦，首先是向脚本控件添加脚本代码时(脚本中存在语法错误)，其次是当执行这部分脚本代码时(脚本中存在运行时错误)。错误发生时，可能需要同时检查 Err 和 Error 对象。但是，脚本控件的 Error 对象提供了关于错误本质的额外信息。下面的示例显示了 VB 中假想的一个错误处理过程。

```
Dim strCode As String
Dim strValue As String
sc.Reset

On Error GoTo SyntaxErrorHandler

strCode = InputBox("Enter Function (name it Test(a))", _
    "Syntax Error Testing", _
    "Sub Test(a): MsgBox ""Result: "" & CStr(a*a): End Sub")

sc.AddCode strCode
```



```
On Error GoTo RuntimeErrorHandler
```

```
strValue = InputBox("Enter a Value for Test function", _  
    "Runtime Error Testing", _  
    "test")
```

```
sc.Run "Test", strValue
```

```
Exit Sub
```

```
SyntaxErrorHandler:
```

```
    MsgBox "Error # " & Err.Number & ": " & _  
        Err.Description, vbCritical, "Syntax Error in Script"  
    Exit Sub
```

```
RuntimeErrorHandler:
```

```
    MsgBox "Error # " & Err.Number & ": " & _  
        Err.Description, vbCritical, "Runtime Error in Script"
```

VB 可以以几种方式处理错误：通过使用 `On Error Goto [Label]`和在 VBScript 中使用 `On Error Resume Next`，并立即检查 `Err.Number`。下面的示例演示了 `On Error Resume Next` 及脚本控件 `Error` 对象的使用，提供了关于错误处理的更多信息：

```
On Error Resume Next
```

```
    sc.AddCode strCode
```

```
    If Err Then
```

```
        With sc.Error
```

```
            MsgBox "Error # " & .Number & ": " & _  
                & .Description & vbCrLf _  
                & "At Line: " & .Line & " Column: " & .Column _  
                & " : " & .Text, vbCritical, "Syntax Error"
```

```
        End With
```

```
    Else
```

```

        MsgBox "No Error, result: " & CStr(sc.Run("Test", _
            strValue))

    If Err Then

        With sc.Error

            MsgBox "Error # " & .Number & ": " _
                & .Description & vbCrLf _
                & "At Line: " & .Line _
                {}, vbCritical, "Runtime Error"

        End With

    End If

End If

```

可以使用两个由初始的 `ScriptControl` 对象暴露的事件 `Event` 和 `Timeout`，来处理某些错误。但是，在某些情况下，比较难以应付，此时应该使用 `On Error ...` 语句，因为：

- 如果有多个 `ScriptControl` 对象，那么只有第一个 `ScriptControl` 对象才会产生 `Timeout` 事件。
- `ScriptControl` 对象或者被附加到表单，或者使用 `WithEvents` 关键字来初始化，这很可能并不是需要的方式。
- 执行某些可能导致错误的脚本时，会损失一定的处理粒度。

如果不想在应用程序中添加任何其他的错误处理脚本代码，就应该使用 `Error` 事件，如下面的示例所示：

```

Private Sub sc_Error()

    Dim strMsg As String

    With sc.Error

        strMsg = "Script error has occurred:" & vbCrLf & vbCrLf

        strMsg = strMsg & .Description & vbCrLf

        strMsg = strMsg & "Line # " & .Line

        ' Syntax errors have additional properties
    End With

```

```

        If InStr(.Source, "compilation") > 0 Then
            strMsg = strMsg & ", Column# " & .Column
            strMsg = strMsg & ", Text: " & .Text
        End If

        strMsg = strMsg & vbCrLf

    End With

    MsgBox strMsg, vbCritical, "Script Error"

    sc.Error.Clear

End Sub

```

注意使用 `ScriptControl` 对象的 `Error` 事件时，事件处理程序是在 `On Error ...` 代码之前被调用的。因此，同时使用这两种错误处理技术会导致出现两条错误信息，使得每一种方法都无法有效地进行。

21.6 调试

用一句简单的话来描述使用脚本控件调试包含在应用程序的脚本，那就是：可以在 VB 集成开发环境中调试原有的 `Visual Basic` 代码，但是要调试脚本中的代码，就必须使用可以免费下载的 `Microsoft 脚本调试器(Microsoft Script Debugger)`。安装了调试器之后，任何未处理的错误或脚本中的 `Stop` 语句都将调用该调试器，就像调用其他脚本一样。

关于脚本调试期和脚本调试技术的更多细节，请参看第 6 章“错误处理和调试”。

还要记住的是，如果在脚本执行期间调用调试器，脚本执行时间就会有所增加，但仍然受 `Timeout` 属性的限制。换句话说，如果 `Timeout` 被设置为 `10 000ms(10s)`，而调试器的使用使脚本的执行暂停了超过 `10s` 的时间，那么脚本控件就会弹出超时警告对话框。

由于这个原因，在进行调试的时候，可能需要将 `Timeout` 属性设置成 `NoTimeout(-1)`，并在发布脚本产品之前在将其设置成另外的值。实现这一点，一个很好的方法是使用具名常量来设置 `Timeout` 值，但要使用条件编译标志和 `#IFDEF` 语句来控制常量的值，如下：

```
#IFDEF blnDebugging
```

```
        Const TIMEOUT_VAL = -1

#ELSE

        Const TIMEOUT_VAL = 15000

#ENDIF

...objSC.Timeout = TIMEOUT_VAL

...
```

21.7 使用已编码脚本

脚本控件的确支持已编码脚本(请参看第 14 章)。但是要记住两点。

首先, 设置已编码脚本的 `Langauge` 属性时, 应将其设置为“`VBScript.Encode`”, 而不是一般的“`VBScript`”。

其次, 如果从文件或数据库中加载一个已编码脚本, 就需要使用别的技术来处理已编码脚本中存在特殊字符的情况。对于从文件中加载, 可能会需要通过 `FileSystemObject.OpenTextFile` 方法(请参看第 7 章)使用脚本运行时 `TextStream` 对象, 而不是使用原来的 `Visual Basic` 函数来打开文件。对于存储和加载数据库, 最好是将脚本作为二进制数据来保存, 而不能使用一般的 `Char` 或 `VarChar` 列保存。

21.8 .NET 工程范例

范例工程 `ScriptControlDemo` 演示了脚本控件在一个 .NET 环境中的基本工作原理, 包括应用程序如何使用脚本和其他技术共享其对象。`VB.NET` 演示工程不代表实际的应用场景, 而是用于实践脚本控件的各种特性, 演示其在 .NET 运行时环境中的工作原理, 并且在 `Visual Basic 6` 或其他支持 `COM` 的语言中具有相似的工作原理。

可以从 www.wrox.com 下载这个范例工程以及本书中提到的其他代码。`ComplexSC` 工程包含在一个名为 `VisualBasic6Demoi.zip` 的文件中。

可以在任何文本编辑器中阅读 `ScriptControlDemo` 工程的代码(文件 `frmMain.vb` 中包含了所有实际的脚本控件代码), 或者可以在 `Visual Studio .NET 2005` 中加载这个工程。该工程

实际上是在 Visual Basic 2005 Express Edition 中创建的。如果没有 Visual Studio .NET 2005 的完整版本，就可以从 Microsoft 下载这个开发环境，地址是：

<http://msdn.microsoft.com/vstudio/express/vb/>

即使不熟悉 VB.NET 的使用，但如果熟悉 Visual Basic、VBA 或 VBScript 的话，也会觉得 VB.NET 的语法是比较熟悉的。下面是该工程中的一个代码示例，它演示了如何在控件中添加一段简单的代码并运行：

```
Dim scriptCtl As MSScriptControl.ScriptControl = _  
    New MSScriptControl.ScriptControl()  
scriptCtl.Language = "VBScript"  
  
Dim stringBldr As System.Text.StringBuilder = New System.Text.StringBuilder()  
stringBldr.Append("Sub ShowMyName(ByVal firstName, ByVal lastName)")  
stringBldr.Append(vbNewLine)  
stringBldr.Append(" MsgBox(""My name is "" & firstName & "" "" & lastName)")  
stringBldr.Append(vbNewLine)  
stringBldr.Append("End Sub")  
scriptCtl.AddCode(stringBldr.ToString())  
  
Dim parms() As Object = {"Super", "Fly"}  
scriptCtl.AllowUI = True  
scriptCtl.Run("ShowMyName", parms)
```

21.9 Visual Basic 6 工程范例

范例工程 ComplexSC 演示了脚本控件的基本工作原理，包括应用程序如何与脚本共享其对象以及传送静态事件——由于这一需要，该工程是 ActiveX EXE 类型的。

可以从 www.wrox.com 下载这个范例工程以及本书中提到的其他代码。ComplexSC 工程包含在一个名为 Chapter 21 downloads.VisualBasic6Demoi.zip 的文件中。

创建依赖外部数据库的数据库应用程序时，总是会遇到应用程序中输入数据库连接字符

串的问题，该字符串将关联适当的数据库和适当的服务器。这部分信息通常是从系统注册表中使用一个关键字检索获取的，它还标识了软件作者和应用程序信息：

Sample Registry Path: SOFTWARE\Company Name\App Name\

Sample Key Name: MyAppConnection

这个范例工程提供的方法可以用于创建、存储和编辑诸如连接字符串这样的应用程序设置的注册表信息。ComplexSC 工程中的 Visual Basic 表单和代码都被设计为普通的，可以通过脚本定制的。这意味着可以发布新的脚本，而不需要重编译和重发布整个 VB 应用程序。

图 21-2 显示了 ComplexSC 工程的主表单。



图 21-2

这里的可能性是很多的：通过在应用程序中暴露对象，并将某些事件传递给脚本，该脚本就会以宏方式运行以适应需要。有一些很特殊的性质，对于在表单和脚本之间的事件传递来说尤其如此。要实现这一功能，所有的控件都必须在设计时被放置到表单上，其中的一些可能要放在控件数组中。脚本可以很容易地控制所有控件的所有属性和方法，但对于控件数组(可选的连接字符串标签及其值)的使用来说，可以简化表单成员的动态修改。这里根据不同的连接选择(OLE DB、ODBC 和 DSN)，可以显示与连接相关的不同的标签和可编辑的值。

表单成员的共享可以很容易地通过 CShared 类来实现，这个类允许与脚本共享主表单及其成员(如下面的示例所示)。虽然有时可能需要暴露单个元素，而非整个表单，并阻止脚本操作那些希望保护的元素，但对于这个应用程序来说，不需要考虑这些。

Option Explicit

```

Private m_Form As Form

Public Property Get Form() As Object
    Set Form = m_Form
End Property

Friend Property Set Form(ByVal newValue As Object)
    Set m_Form = newValue
End Property

```

下面我们将表单封装在 CShared 类中。有了 CShared 类，就需要使用脚本控件的 AddObject 方法与脚本共享表单。这是通过 InitScriptControl 过程完成的，该过程是在表单加载时执行的(在 Form_Load 时间处理程序中调用)。此时要传递一个对 VB 表单的引用，并通过 CShared.Form 属性来暴露它。

```

Private Sub Form_Load()
    Set objScript = InitScriptControl(Me)
    objScript.Run "init"
End Sub

```

下面的 InitScriptControl 过程实例化 ScriptControl 对象，加载脚本，实例化 CShared 对象，并将其暴露给脚本控件。因为将 AddObject 方法的第三个参数设置为 True，所以表单中所有的成员都是共享的。

```

Function InitScriptControl(frmForm As Form) As ScriptControl

    Dim objSC As ScriptControl
    Dim fileName As String, intFnum As Integer
    Dim objShare As New CShared

    ' create a new instance of the control
    Set objSC = New ScriptControl
    objSC.Language = "VBScript"
    objSC.AllowUI = True

```

```

Set objShare.Form = frmForm

objSC.AddObject "share", objShare, True

' load the code into the script control
fileName = App.Path & "regeditor.scp"
intFnum = FreeFile

Open fileName For Input As #intFnum

objSC.AddCode Input$(LOF(intFnum), intFnum)

Close #intFnum

' return to the caller

Set InitScriptControl = objSC

```

End Function

初始化脚本控件之后，Form_Load 代码将调用脚本中的 Init 过程，该过程将设置表单中所有必需的控件。在实际应用中，有些控件将被预置一些属性(如背景色、是否被启用，等等)，而其他属性是通过脚本访问由 CShared.Form 方法暴露的表单成员来进行初始化的。

Sub Init()

```

Dim i, strTmp

Form.Caption = "Connection Registration Manager"

strTmp = "This application saves the database connection"

strTmp = strTMP & " string in the registry. " & vbCrLf

Form.IblExplanation = strTmp

Form.IblRegistry.Caption = ""

' this information should be reflected in your application

' the standard is to store the registry keys in subhives

' for different companies and projects

Form.txtSubpath.Text = "SOFTWARE Company Name App Name "

```



```

' finally the name of the key

' you could similarly extend this application so it would

' work like a wizard, and register several keys

Form.txtKey.Text = "MyAppConnection"

Form.lblRegistry.Caption = ""

Form.cmdRegister.Enabled = False

Form.cmdProcess.Enabled = True

For i = 0 To 5

    Form.lblLabel(i).Visible = False

    Form.txtText(i).Visible = False

Next

Form.cboCombo.Clear

Form.cboCombo.AddItem "OLE DB"

Form.cboCombo.AddiTem "ODBC"

Form.cboCombo.AddItem "DSN"

End Sub

```

接下来，要响应应用程序产生的事件。在这个简单的例子中，只要将应用程序截获的事件直接传递给脚本即可。这样的话，应用程序有可能会将下面的事件传递给脚本：

```

Private Sub cboCombo_Click()

    objScript.Run "cboCombo_Click"

End Sub

Private Sub txtText_KeyPress(Index As Integer, KeyAscii As Integer)

    KeyAscii = objScript.Run("txtText_KeyPress",Index, KeyAscii)

End Sub

```

如上面的示例所示，事件被直接传递给脚本，也可以选择将事件产生的参数一起传送给脚本。因为在某些情况下，可能需要修改某个参数。事件处理过程应该看成是一个函数，可

以返回修改后的值。这可能是修改这种参数的最简单的方式。不过，这个功能并不是应用程序必须具有的。下面这个包含在脚本中的函数可以将文本框中输入的所有字符都转换成大写。

```
Function txtText_KeyPress(Index , KeyAscii)

    txtText_KeyPress = Asc(Ucase(Chr(KeyAscii)))

End Function
```

上述方法和 VB 代码中使用的方法有少许不同，因为即使以引用的方式(一般来说，VB 代码应该是 `KeyAscii = Asc(Ucase(Chr(KeyAscii)))`)来传递 `KeyAscii` 的值，脚本都不会更新这个参数在原有 VB 代码中的值。这样，我们将事件处理程序从过程更改为函数，就实现了一种简单的变通。

可以不执行默认的事件处理程序，而是在脚本中提供另一个可选的事件处理程序。如果脚本中没有成员过程，就会发生错误，对于脚本中不包含恰当的具名过程的情况来说，这样就提供了忽略事件或提供默认事件的机会。下面的示例显示了最简单的错误抑制方法，允许创建默认的事件处理程序。此外，如果关闭错误处理程序(使用 `On Error Resume Next`)，脚本就必须包含适当的具名过程，并具有正确数量的参数。

```
Private Sub cboCombo_Click()

    On Error Resume Next

    objScript.Run "cboCombo_Click"

    If Err = 0 Then Exit Sub

    ' default event handler goes here

...End Sub
```

应用程序的具体细节取决于脚本自身，所以下面的示例显示脚本中只包含 `cboCombo_Click` 过程的部分实现，并没有将整个代码清单全部复制进来。

关键的控件被重置之后，设置标签的值，以及对应于 `CLE DB` 类型的连接字符串的文本。

```
Sub cboCombo_Click()

    Dim strComboSelection, strTmp
```

```

' Clean Up in case this was pressed already

Form.cmdRegister.Enabled = False

Form.cmdProcess.Enabled = True

Form.lblRegistry.Caption = ""

For i = 0 To 5

    Form.lblLabel(i).Visible = False

    Form.txtText(i).Visible = False

Next

strComboSelection = _

    Trim(Form.cboCombo.List(Form.cboCombo.ListIndex))

Select Case strComboSelection

    Case "OLE DB"

        For i = 0 To 4

            Form.lblLabel(i).Visible = True

            Form.txtText(i).Visible = True

        Next

        Form.lblLabel(0).Caption = "Provider="

        Form.lblLabel(1).Caption = "Data Source="

        Form.lblLabel(2).Caption = "Initial Catalog="

        Form.lblLabel(3).Caption = "User ID="

        Form.lblLabel(4).Caption = "Password="

        Form.txtText(0).Text = "SQLOLEDB"

        Form.txtText(1).Text = "DATABOX"

        Form.txtText(2).Text = "MyAppDB"

        Form.txtText(3).Text = "Student"

        Form.txtText(4).Text = "teacher"

```

[...]

```
End Select
```

```
strTmp = "Please Fill In Remaining Values in the available"
```

```
strTmp = strTmp & " text boxes. " & vbCrLf
```

```
strTmp = strTmp & "You may press ""Proceed"" button, or"
```

```
strTmp = strTmp & " change the connection method again. "
```

```
strTmp = strTmp & "Leaving User ID empty will leave out"
```

```
strTmp = strTmp & " user information from registry"
```

```
Form.IblExplanation = strTmp
```

```
End Sub
```

应用程序的剩余部分用于响应终端用户的事件，并建立核心应用程序所请求的连接字符串，启用和关闭控件，以及根据场景不同修改表单上的值。最后提到的这个动作是直接由应用程序本身执行的；基于存储在表单上某一标签中的字符串将一个值写入注册表。

这个小应用程序可以进一步扩充，利用一些脚本提供类似于向导的功能，相应脚本的编写也是很简单的。

21.10 小结

脚本控件是由 Microsoft 提供的免费控件，使 VB 6、.NET 或其它支持 COM 的应用程序能够提供脚本引擎。使用脚本引擎可以进行简单的动态表达式计算，也可以实现成熟的宏语言插件，用于自动化应用程序。

这一章讨论了下面这些内容：

- 脚本控件的概念
- 为什么说脚本控件是 Visual Basic 6 和 .NET 应用程序的有益补充
- 为什么要使用脚本控件(为什么不使用脚本控件)
- 脚本控件的对象模型，包括属性、方法和事件
- 错误处理和调试
- 通过两个范例工程演示了脚本控件在 VB.NET 和 VB 6 中的使用