

PowerShell 强大的管理能力

在经历了漫长的等待之后，PowerShell 终于已蓄势待发。这意味着该是 Windows 管理员提起注意的时候了。PowerShell 提供了也许是最简单且最灵活的方法来自动执行各种各样的管理任务，从而您的工作效率和效力都得到提高。

但更为重要的是，Microsoft 正在 PowerShell™ 之上构建 Exchange Server 2007 和 System Center 2007 之类产品的图形管理控制台。这意味着您将能够从 PowerShell 内部执行几乎所有的管理任务。随着时间的推移，Microsoft 计划提高越来越多产品的管理能力。因此，PowerShell 最终会成为用于管理几乎任何 Microsoft 服务器产品的第一个全能型工具。为了帮助您尽快入门，我将在此新设的专栏中定期探讨 PowerShell。请务必下载一份软件。

强大而又简便的功能

顾名思义，PowerShell 是一个外壳，但与从 Windows NT® 3.1 起就出现的命令提示符 (Cmd.exe) 不同。Cmd.exe 不会消失，但随着 PowerShell 的到来，几乎没有什么理由再继续使用 Cmd.exe。

PowerShell 在使用方面与 Cmd.exe 并无多大不同，只是 PowerShell 的功能更为强大。与 Cmd.exe 一样，PowerShell 具有内置的脚本编写语言，不过它比 Cmd.exe 原始的批处理语言更为灵活。灵活性是如何体现的呢？有了 PowerShell，您可以使用仅包括大约半打内置关键字的语言自动执行极其复杂的任务。

既然我已提到了脚本编写，那么现在我可能就应该简单提一下安全性。PowerShell 得益于 Microsoft 过去十多年在安全性方面所取得的研究成果。默认情况下，PowerShell 不会运行脚本，只能交互式地用它来运行单个命令。如果您确实启用了脚本编写，则可令 PowerShell 仅运行经过数字签名的脚本。这些均有助于确保 PowerShell 不会成为下一个 VBScript — 一种伟大的语言，但它常被滥用来创建恶意脚本。VBScript 也不会退出历史舞台，但您可能会发现 PowerShell 对于许多不同的任务来说更易于使用。

Cmd.exe 做到的事情，PowerShell 几乎都能做到。例如，您可以运行 ipconfig 并且会获得同样熟悉的输出。但是 PowerShell 会引入一整套新的命令，它们不是外部可执行文件。这些 cmdlet（发音为“command-let”）就内置于 PowerShell 之中（要了解对 PowerShell 使用入门最有用的 cmdlet，请参见侧栏的“快速使用入门十大 Cmdlet”）。

快速使用入门十大 Cmdlet

Get-Command 用于检索所有可用 cmdlet 的列表。

Get-Help 用于显示有关 cmdlet 和概念的帮助信息。

Get-WMIObject 用于通过 WMI 来检索管理信息。

Get-EventLog 用于检索 Windows 事件日志。

Get-Process 用于检索单个活动进程或活动进程的列表。

Get-Service 用于检索 Windows 服务。

Get-Content 用于读入文本文件，将每行视为一个子对象。

Add-Content 用于将内容附加到文本文件。

Copy-Item 用于复制文件、文件夹和其他对象。

Get-Acl 用于检索访问控制列表 (ACL)。

要获得 PowerShell 自带的完整 cmdlet 列表，请访问 windowssdk.msdn.microsoft.com/en-us/library/ms714408.aspx。

所有 cmdlet 都以标准的“动词-名词”格式命名，这使其易于理解和记忆。例如，运行 Get-Command cmdlet 将会列出所有可用的 cmdlet。对于管理员来说最有用的 cmdlet 也许就是 Get-WMIObject。若您想要查明 Server2 正在运行哪个服务包，只需运行：



```
Get-WMIObject Win32_OperatingSystem -Property ServicePackMajorVersion  
-Computer Server2
```

要使用 VBScript 来发现同一信息，就得编写几行代码。利用 cmdlet 可以处理服务（Start-Service、Stop-Service 等）、进程（Stop-Process 等）、文件（例如，Rename-Item、Copy-Item、Remove-Item、Move-Item）等等。其中许多 cmdlet 甚至还有快捷名称，称为别名。对于 Get-WMIObject 而言，您可以只键入 gwmi。运行 Get-Alias 将为您提供这些快捷名称的列表。

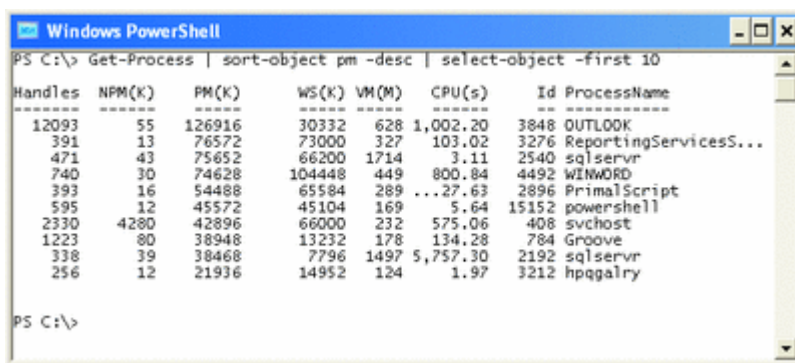
为什么面向对象很重要

PowerShell 在 Microsoft® .NET Framework 上构建，所以它完全是面向对象的。通常，只有软件开发人员才会对此感到兴奋，但是就本处的情况而言，面向对象可以为管理员节省大量的时间。这是因为管理员现在只需在基于文本的外壳内就能直接处理丰富的对象。请看以下示例：



Get-Process | Sort-Object pm -desc | Select-Object -first 10

这里只有一行，其中含有以管道分隔的三个不同的 cmdlet（稍后会对此进行详述）。第一个 cmdlet 检索所有正在运行的进程，然后将那些对象传递给或通过管道输送到 Sort-Object。第二个 cmdlet 基于每个进程对象的 pm（即物理内存）属性按降序对其进行排序。然后将进程对象的有序集合通过管道输送到 Select-Object，它将选取前十个对象进行显示。结果呢？此行简单命令会显示机器上的十大物理内存使用者，如图 1 所示。这是一种极其有效的方法，借此可在执行某些故障排除时进行快速浏览。



Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
12093	55	126916	30332	628	1,002.20	3848	OUTLOOK
391	13	76572	73000	327	103.02	3276	ReportingServices...
471	43	75652	66200	1714	3.11	2540	sqlservr
740	30	74628	104448	449	800.84	4492	WINWORD
393	16	54488	65584	289	...27.63	2896	PrimalScript
595	12	45572	45104	169	5.64	15152	powershell
2330	4280	42896	66000	232	575.06	408	svchost
1223	80	38948	13232	178	134.28	784	Groove
338	39	38468	7796	1497	5,757.30	2192	sqlservr
256	12	21936	14952	124	1.97	3212	hpqgalry

图 1 使用简单的 Cmdlet 进行故障排除

管道（在美式键盘上，该竖线字符通常位于反斜线按键上）的使用是 PowerShell 具有如此能力不可或缺的组成部分。使用此字符，可以将对象从一个 cmdlet 传递（或通过管道输送）给另一个，从而使每个 cmdlet 可以进一步提炼结果、设置结果的显示格式，等等。之所以能够采用这种机制，是因为每个 cmdlet 都会返回一个或多个对象，而非纯文本，这就使得后续 cmdlet 可以处理完整的对象。

PowerShell 中的对象使用非常普遍，这全都仰仗于它的变量。而且，您不必预先声明变量，只需在变量名称前放置一个美元符号（\$）就可以开始使用它们了。尽管不是必需的，但您也可以将所要放入变量的数据类型告知 PowerShell。这使得 PowerShell 可以将变量映射到某一种极其强大的 .NET Framework 类型，从而为您提供诸多附加的内置功能。例如，假设您想要提示输入计算机名称并从该计算机中检索服务包版本，但是您不知道键入的计算机名称是否会包括两个反斜线（如 \\Server2）。由于您知道 Get-WMIObject cmdlet 不需要反斜线，因此您可以将计算机名称保存到一个字符串变量中，然后使用 Replace 方法以空字符串替换反斜线，如下所示：



```
[string]$c = Read-Host "Enter computer name"
```

```
$c = $c.Replace("\","")
```

```
Get-WMIObject Win32_OperatingSystem
```

```
-Property ServicePackMajorVersion
```

```
-Computer $c
```

已在 `$c` 变量中提供了 `-Computer` 参数的值。该变量最初是作为字符串而创建的，因此它具有 .NET Framework 字符串类型的所有功能，包括 `Replace` 方法。当然，要了解所有这些功能需要花费一些时间，但是通过示例您应该很容易就能掌握这些功能。PowerShell 本身也有助于简化学习过程。例如，如果您键入 `$c = $c.`（不要忘了句号）并按 `Tab`，PowerShell 将显示 `Clone()`，这是字符串类型的第一个方法。如果您一直按 `Tab`，PowerShell 将会循环列出所有可用的方法。实质上，当您这样做时，PowerShell 是在向您展示它所知道的字符串处理方法。

在此给出一项更难的任务。从文件中读取计算机名称的列表（每行一个名称）并显示每台计算机的服务包号。在 VBScript 中，此项任务将需要一打或更多行的代码。在 `Cmd.exe` 中，您就得使用复杂的批处理文件。在 PowerShell 中，此项任务仅需一行：



```
Get-Content "c:\computers.txt" | foreach
{ $_; gwmi Win32_OperatingSystem -prop
ServicePackMajorVersion -comp $_ }
```

`Get-Content` cmdlet 读取 `C:\Computers.txt` 的内容。文件的每一行均自成一个对象。此对象集合（即，计算机名称）通过管道输送到 `foreach` 命令（它其实就是 `ForEach-Object` cmdlet 的别名）。花括号内的命令会对通过管道送入的每个对象重复一次（在本例中就意味着它们对每个计算机名称都要运行一次）。特殊的 `$_` 变量将包含当前对象（即，当前的计算机名称）。

花括号内实际是两个命令：第一个只是通过输出 `$_` 的内容来显示当前的计算机名称。第二个是现已熟悉的 `gwmi`。结果将显示文件中所列出的每台计算机的服务包版本列表。所有这些都是用相对简单的一行命令完成的。

请注意，`-Property` 和 `-Computer` 参数名已被缩写。PowerShell 仅需有足够的信息即可唯一地区分参数名。

可读性与复用

然而，编写一行命令和参数无助于可读性。PowerShell 允许您将其拆分成更具可读性的形式，您甚至不用编写脚本就能将其直接键入外壳。这就是它可能的形式：



```
PS C:\> $names = get-content "c:\computers.txt"

PS C:\> foreach ($name in $names) {

>> $name

>> gwmi Win32_OperatingSystem -prop ServicePackMajorVersion -comp $name

>> }

>>
```

这一次，文件的内容存储在变量 `$names` 中。本例仍使用了 `foreach`，但它不是通过管道进行输入的，因此必须告知它需要循环处理哪个对象集合以及将每个对象存储在哪个变量中，即 `($name in $names)` 所指明的那一部分。其他方面大体相同，只要您一按 `Enter`，就会执行代码并显示结果。

如果要重复使用此同一代码，只需将其制成函数即可。再次将以下内容直接键入到外壳中：



```
PS C:\> function Get-ServicePacks ($file) {

>> $names = get-content $file

>> foreach ($name in $names) {
```

```
>> $name

>> gwmi win32_operatingsystem -prop servicepackmajorversion -comp $name

>> }

>> }

>>
```

如您所见，实际上并未做太多更改。只需将上一示例封入一个名为 `Get-ServicePacks`（与 PowerShell 的“动词-名词”命名惯例保持一致）的函数即可。该函数现在具有一个名为 `$file` 的输入参数，该参数在 `Get-Content` cmdlet 中已被取代，这样便可在运行函数时指定另一不同的文件。至此已定义了函数，运行它很简单，方法几乎与 cmdlet 一样，只需调用其名称并传递输入参数即可：



```
PS C:\> Get-ServicePacks c:\computers.txt
```

图 2 显示了结果。

其缺点在于此函数仅在该 PowerShell 实例运行期间才存在。一旦关闭外壳，该函数即会消失。您可以将该函数复制到您的 PowerShell 配置文件中，该配置文件是一种自动运行的脚本，它会在 PowerShell 每次启动时执行。这样做就使该函数在打开的每个 PowerShell 窗口中均可用。或者，如果需要，也可将该函数做成独立的脚本，然后只需键入其路径和文件名即可执行该脚本。

```
Windows PowerShell
PS C:\> get-servicepacks c:\computers.txt

ServicePackMajorVersion : 2
    _GENUS                : 2
    _CLASS                : Win32_OperatingSystem
    _SUPERCLASS           :
    _DYNASTY              :
    _RELPATH              :
    _PROPERTY_COUNT       : 1
    _DERIVATION            : {}
    _SERVER               :
    _NAMESPACE            :
    _PATH                 :

ServicePackMajorVersion : 1
    _GENUS                : 2
    _CLASS                : Win32_OperatingSystem
    _SUPERCLASS           :
    _DYNASTY              :
    _RELPATH              :
    _PROPERTY_COUNT       : 1
    _DERIVATION            : {}
    _SERVER               :
    _NAMESPACE            :
    _PATH                 :

PS C:\>
```

图 2 运行 `Get-ServicePacks` 函数的结果

一切尽在于文件（或文件夹）

PowerShell 决不只是函数和 cmdlet。让我们以文件管理为例来快速了解一下其中还蕴藏着什么别的内容。您可能对 `Cmd.exe` 中的驱动器和文件夹导航再熟悉不过了——键入 `C:` 可切换到 `C` 驱动器，键入 `cd \test` 可转入 `C:\Test` 文件夹。PowerShell 的工作方式完全相同，不过 `cd` 只是 `Set-Location` cmdlet 的别名。

尝试运行 `Get-PSDrive`，该 cmdlet 会列出所有可用的驱动器。除了常用的 C:、D: (也许还有 A:) 驱动器之外，您还会发现一个名为 Cert 的驱动器、另一个名为 Env 的驱动器以及其他名为 HKCU 和 HKLM 的驱动器。实际上，PowerShell 将许多不同类型的存储资源均以“驱动器”形式公开，从而使得诸如本地证书存储区、环境变量和注册表等资源均可通过一个象文件那样为大家所熟悉的导航界面来获得。

键入 `Set-Location HKLM:` (或 `cd hkln:`，如果您喜欢使用快捷方式) 而后按 Enter，可转到 HKEY_LOCAL_MACHINE 注册表配置单元。如果要删除某个表项，可使用 `del` 将其删除，就像该表项是文件或文件夹一样 (不过一定要非常小心，如果删除了必需的表项或是对注册表进行了不正确地修改，可能会发生严重问题)。

这一切灵活性均来自于提供程序，它们将资源 (如注册表和证书存储区) 映射成类似于文件系统那样的格式。Microsoft 计划通过另外的提供程序来扩展 PowerShell，例如，使您能够导航 Exchange Server 存储区，就像它是文件系统一样。这是一项非常重要的技术，重要性就在于它吸纳了 Windows 所使用的林林总总的存储库并使它们看起来全都相同，而且还使得它们都可以通过您已熟悉的命令和技术系统来进行管理。

安全性设计

我已提到过 PowerShell 在设计时很注重安全保护。PowerShell 中主要的安全功能是它的执行策略。默认情况下，此策略设置为“Restricted”，您可以通过运行 `Get-ExecutionPolicy` cmdlet 来进行验证。在 Restricted 模式下，不能运行脚本。就这么简单。由于这是默认模式，因此刚出盒的 PowerShell 不能用来运行脚本。

您可以使用 `Set-ExecutionPolicy` cmdlet 指定其他模式。我个人更偏爱 RemoteSigned 模式。在此模式下，可以运行未经数字签名的本地脚本 (而非远程脚本)，同时能够以最简单的方法开发和测试脚本。除非已使用受信任发布方所颁发的证书对脚本进行了数字签名，否则 AllSigned 模式不会运行任何脚本。最后，采用 Unrestricted 策略可运行任何脚本。我建议切莫采用此策略，因为以此方式打开的 PowerShell 会运行可能伺机找上您计算机的恶意脚本。请注意，执行策略也可能受组策略支配，它会替代所有本地设置 (如果组策略设置正要替代您的本地设置，`Set-ExecutionPolicy` 会向您发出警告)。

另外，PowerShell 不会从当前目录运行脚本，除非您指定了该路径。这样设计是为了防止命令攻击。比如，某人创建了一个名为 IPConfig.ps1 的脚本 (PS1 是 PowerShell 脚本文件的文件扩展名)。如果文件可在当前文件夹之外运行，则您可能会键入 `ipconfig` 而运行此用户所创建的脚本，可当时您其实是希望运行 Windows 程序 `Ipconfig.exe`，这样就带来了风险。由于 PowerShell 不会在当前文件夹之外运行脚本，所以这种错误不可能发生。如果您的确想在当前文件夹之外运行脚本，只需指定路径即可：例如 `.myscript`。对当前文件夹的显式引用确保您知道自己正在运行脚本，而非外壳命令。

PowerShell 还具有使实验变得更加安全的功能。例如，请考虑 (但请勿尝试) 这个令人害怕的组合：



Get-Process / Stop-Process

`Get-Process` cmdlet 会创建进程对象的集合并通过管道将其输送到 `Stop-Process` cmdlet，而后者真的会将它们停止！这会导致在大约 5 秒钟后出现蓝屏 STOP 错误，原因是终止了关键的 Windows 进程。但是，您可以通过添加非常方便的 `-Whatif` 参数来看看将会发生什么情况，而不会令其真的发生：



Get-process | Stop-Process -Whatif

在 PowerShell 中运行此命令会产生一组语句，它们会告诉您 cmdlet 将会做什么，而不会真的让它们这样做。PowerShell 中的在线帮助系统 (可通过 `help` 别名访问) 尚未记载 `-Whatif` 参数，但请记住它。它是一个很好的工具，用于测试脚本和 cmdlet 以检验其结果，而不会实际做出任何具有潜在危害性和破坏性的事情。

总结

在没有加入此 PowerShell 版本的功能中，最重要的也许就是对 Active Directory® 服务接口 (ADSI) 的支持。尽管 PowerShell 可以利用与 Active Directory 和其他目录服务配合工作的功能非常强大的 .NET 类，但它尚没有一个方便的 `Get-ADSIObject` cmdlet。这就给处理目录对象带来一点困难。

此外，PowerShell 通常提供了多种不同的方法来执行同一任务。这样很好，但可能会令人在学习 PowerShell 时变得更加迷惑，因为对于任何给定的任务，您可能都会看到半打不同的处理方法示例。

PowerShell 构造

上个月，我向您展示了将 Windows PowerShell 用于即时解决管理任务的方法 - 但没有编写任何文字资料。但是，Windows PowerShell 是一种优秀的交互式命令解释程序，您可以在开始利用它强大但脚本简单的语言时使用它的功能并自动执行更加复杂的任务。

但首先您要问问自己：Microsoft 真的需要另一种脚本语言吗？毕竟，Microsoft 向我们提供了 KiXtart（一种登录脚本处理器）以及 Visual Basic® Scripting Edition (VBScript)。但是，答案是肯定的。Microsoft 确实需要另一种脚本语言。我将解释原因。

Windows PowerShell™ 语言需要简单直观，这样管理员可以不经太多训练就掌握它。它也需要相当灵活，这样就可以适应 Windows PowerShell 本身提供给用户的所有强大功能。

因为 Windows PowerShell 是基于 Microsoft® .NET Framework 的，所以它的脚本语法需要得到 .NET 的支持。在编写 Windows PowerShell 脚本语言时，设计师选择的实质上是高度简化的 C#（读作 C-Sharp，是 .NET Framework 附带的语言之一）。为什么不继续使用类似于 VBScript 的语法呢？实际上，Windows PowerShell 脚本语言与 VBScript 并没有多大不同，只是与 C# 的语法更加接近，从某种意义上讲，PowerShell 是学习 .NET Framework 编程的踏脚石。如果您决定进阶到 Visual Studio® 并开始编写 C# 应用程序，您的大部分 Windows PowerShell 脚本语法知识也将继续给您提供帮助。

Windows PowerShell 脚本语言 - 或者就此而言，任何脚本语言 - 中最重要的一点是它的构造。这些是特殊语言元素，允许 Windows PowerShell 执行逻辑比较并根据比较结果进行不同的操作或者允许不断地重复指令。

进行逻辑思考

逻辑比较是很多脚本语言构造的核心，Windows PowerShell 也不例外。比较实质上查看两个值或对象并计算以决定比较结果是 True 还是 False。例如，您可以问自己，“用户的密码有效期是否到今天截止？”如果今天到期结果为 True，否则为 False。注意我将 True 和 False 的首字母写成大写形式，因为在 Windows PowerShell 中它们是具有特殊含义的词。

此处是一个您可以在 Windows PowerShell 中执行的真实的逻辑比较示例：



```
PS C:\> $a = 1
```

```
PS C:\> $b = 2
```

```
PS C:\> $a -eq $b
```

```
False
```

我已经创建了名为 \$a 的变量并将其设置为包含值 1。在 Windows PowerShell 中，变量名总是以美元符号开头，因此容易辨认。从技术角度来说，= 是指赋值运算符，因为它用来赋值。接下来，我会创建第二个变量 \$b 并将值 2 赋给它。然后进行实际的逻辑比较 - 我要求 Windows PowerShell 利用 -eq（相等）运算符来比较 \$a 和 \$b 的内容。PowerShell 执行比较，确定两个值不相等，并显示结果：False。

Windows PowerShell 运算符与您可能见过的其他脚本语言有点不同，甚至不同于 C#。大多数语言使用 = 运算符来检查是否相等以及执行赋值操作；而 Windows PowerShell 通过对每个函数采用专用的运算符来避免混淆。图 1 显示 Windows PowerShell 比较运算符，以及您可能已经熟悉的其他语言（如 VBScript）中的相等运算符。

Figure 1 PowerShell 比较运算符

运算符	运	
	名称	说明
-	相等	测试值是否相等。其他语言可能使用 = 或 == 来测试相等。

eq

- 不等 测试不等。其他语言可能使用 <> 或 != 来测试不等。

ne

- 大于 测试一个值是否大于另一个值。其他语言可能使用 > 字符。

gt

- l 小于 测试一个值是否小于另一个值。其他语言可能使用 < 字符。

t

- 大于 测试一个值是否大于或等于另一个值。与 VBScript 和其他语言中的 >= 类

ge 或等于 似。

- l 小于 测试一个值是否小于或等于另一个值。与 VBScript 和其他语言中的 <= 类

e 或等于 似。

这些比较运算符有另一种有趣的功能。看看该比较：



```
PS C:\> $a = "TechNet"
```

```
PS C:\> $b = "technet"
```

```
PS C:\> $a -eq $b
```

```
True
```

默认情况下，比较运算符不区分大小写，意味着大写的 TechNet 将视为等价于 “technet”。那很方便，因为在大多数管理任务中您并不关心字母的大小写问题。但是，有时候您可能会关心，而您可以要求 Windows PowerShell 将字母 c 置于比较运算符之前以执行区分大小写的比较：



```
PS C:\> $a -ceq $b
```

```
False
```

同样，如果您确实想知道或不确定 Windows PowerShell 是否会执行不区分大小写的比较，您可以通过预置字母 i 来强制它这样做：



```
PS C:\> $a -ieq $b
```

```
True
```

只要记住逻辑比较总是产生两种结果之一：True 或 False。

做决定

既然您知道了如何编写逻辑比较，您可以开始在构造中使用它们。我将向您展示的第一个构造允许 Windows PowerShell 在比较的基础上做出决定。它称为 If 构造，并且有几种变形形式。以下是最简单的形式：



```
PS C:\> if ($a -eq $b) {  
  
>> Write-Host "They are equal"  
  
>> }  
  
>>  
  
They are equal
```

此处有一些有趣的问题需要注意。首先，变量 \$a 和 \$b 仍然分别包含值 "TechNet" 和 "technet"。我使用 If 关键字来作为构造的开头。之后，在括号中我输入了要执行的逻辑比较。接着是花括号，表示我将调用的条件代码（如果比较时返回的结果为 True，则为 Windows PowerShell 将要执行的代码）的开始。您从之前的示例可以知道该比较确实返回 True，因此我期望执行条件代码。我键入我的条件代码 Write-Host "They are equal"，然后按 Enter。最后，我键入右花括号结束条件代码段，并敲击 Enter 两次。（第二次是在空白行上敲击 Enter，这让分析器知道我已经完成了编写并准备好执行代码。）

注意该构造并不从脚本文件运行。我只需将它键入 Windows PowerShell 命令行。这使得 Windows PowerShell 在 Windows 的脚本世界里独一无二：脚本可以交互式地创建，也可以放入文件以做长久的存储。

我一键入左花括号并按 Enter，Windows PowerShell 就显示 >> 提示符。那是它的表达方式，意思是“我识别出您位于构造中，并且我已准备好让您在构造内键入任何内容。”在我键入右花括号并敲击 Enter 两次后，Windows PowerShell 立即执行构造，确定其逻辑比较结果是 True 并执行条件代码。您可以看见该内容，因为 "They are equal" 在 PowerShell 返回到其正常提示符前显示。使用 Windows PowerShell 来交互式地编写脚本允许您在将代码构建成为更长期的脚本前快速测试它们，使得学习和调试都更加容易。

我要指出 Windows PowerShell 对按 Enter 这类的事并不是要求特别高。例如，这在功能上与上一示例相同：



```
PS C:\> if ($a -eq $b) { Write-Host "They are equal" }  
  
They are equal
```

因为这些内容我都在一行上键入，所以 Windows PowerShell 不需要显示特殊的 >> 提示符；它只需在我于行末按 Enter 时执行构造。Windows PowerShell 是如何知道可以执行构造了呢？因为它在那个点处是完整的 - 已键入右花括号。

我提到 If 构造的其他变形形式。以下是一个完整的示例，以其在 PS1 脚本文件而不是命令解释程序中的形式显示：



```
if ($a -eq $b) {  
  
    Write-Host "They are equal"  
  
} elseif ($a -lt $b) {  
  
    Write Host "One is less than the other"  
  
} else {
```


Write Host "One is greater than the other"

```
}
```

构造同样使用 If 关键字作为开头。但是，在比较结果为 False 的情况下，我使用 Elseif 关键字来提供了另一种比较。如果第二个比较结果也是 False，则最后一个关键字 Else 会提供将执行的最后一组代码。

重复自身

Windows PowerShell 包含几个用于反复执行代码的构造，直到比较结果为 True 或 False。程序员调用这些循环构造。更好的是，其中最有用的一个循环构造可以枚举集中的对象并为每个对象执行一行或多行代码。更确切地说，构造称为 foreach 构造，形式如下：



```
PS C:\> $names = get-content "c:\computers.txt"
```

```
PS C:\> foreach ($name in $names) {
```

```
>> Write-Host $name
```

```
>> }
```

```
>>
```

```
don-pc
```

```
testbed
```

首先我要求 Windows PowerShell Get-Content cmdlet 检索 c:\computers.txt 文件，这是我自己创建的文件，每行包含一个计算机名称。Windows PowerShell 将每一行作为对象来处理，因此文件实质上是包含这些对象的集合。集合以变量 \$names 结束。使用 Foreach 关键字，我告诉 Windows PowerShell 通过利用变量 \$names 在每次循环执行时表示当前对象来枚举 \$names 集合。循环代码在花括号中。因此，对于文件中的每个名称，我都将输出到命令行。并且，如同您可以从构造后的输出看到的一样，那正是 Windows PowerShell 所做的。您可以看到这是如何在管理脚本中带来明显的益处的：例如，您可以轻松地构造服务器名称列表，并让 Windows PowerShell 依次从每一个中检索信息。

实际构造

那么，让我们利用 If 构造和 foreach 构造来进行逻辑比较，并做一些有用的事情。我要快速检查一组服务器上的“Messenger”服务的状态。我期望在大多数服务器上服务将停止，因此我不想让 Windows PowerShell 列出所有其中的服务处于我所期望的状态的服务器；我只想让它列出“Messenger”服务实际已启动的服务器，因为我需要对这些服务器进行某些操作。

我知道 Windows PowerShell Get-Service cmdlet 可以有助于检索我需要用于本地计算机的信息。但遗憾的是，它无法访问远程计算机，那才是我真正的目的。令人高兴的是，我也可以使用 Get-WMIObject cmdlet，通过 Windows Management Instrumentation (WMI) 来访问相同的信息，这使我可以使用远程计算机。以下是脚本：



```
$names = get-content c:\computers.txt
```

```
foreach ($name in $names) {
```

```
    $svc = get-wmiobject win32_service '
```

```

-computer $name -filter "name='messenger'"

if ($svc.started -eq $True) {

    write-host "$name has Messenger running"

}

}

```

留意到第三行的 ' 字符了吗？它告诉 Windows PowerShell 下一行是延续部分。它在整行如果不换行就无法置于库中这种情况下有用。还要注意 If 构造将 \$svc.started 与 \$True 比较。变量 \$True 是 Windows PowerShell 中的特殊变量，表示布尔 True 值。（同伴变量 \$False，表示布尔 False。）实际上，在那里我本可以走一些捷径：



```

$names = get-content c:\computers.txt

foreach ($name in $names) {

    $svc = get-wmiobject win32_service '

        -computer $name -filter "name='messenger'"

    if ($svc.started) {

        write-host "$name has Messenger running"

    }

}

```

记住，If 构造的条件只需要为 True 或 False。通常，您通过比较两个值获得 True 或 False，就像我在该脚本的第一个版本中所做的那样。但是，由于“Started”属性为 True 或 False，因此无需将其与 True 或 False 比较。

一种有用的工具

至此您已经获得了一种利用构造来发挥作用，简单而有用的管理工具。无论您是交互式地将它键入 Windows PowerShell，还是在 PS1 文件中保存它以轻松地复用，它都是一种检查各种计算机上服务状态的方便的工具，并且很好地演示了构造是如何有助于自动执行管理任务的。

PowerShell 一次编写一行脚本

在过去的专栏中，我强调了 PowerShell 是一种外壳这一事实。它应被交互式使用 — 与您可能已经很熟悉的 cmd.exe (或命令提示符) 外壳并无不同。PowerShell 仍然支持脚本语言 — 一种比 cmd.exe 的批处理语言更稳定可靠的语言。并且它同某些语言(如 VBScript) 一样功能强大(如果不如其功能强大的话)。但是，PowerShell™ 是一种交互式外壳的事实，使得编写脚本非常容易学习。实际上，您可以在外壳内部交互式地开发脚本，从而可以一次编写一行脚本，并立即看到自己努力的成果。

此迭代的脚本编写技术还使调试变得更加容易。由于您立即看到了脚本的结果，因此您就可以在脚本不能按预期方式运行时快速地对其进行修订。

在此专栏中，我将向您介绍一个在 PowerShell 中进行交互式脚本编写的示例。我将创建一个脚本，该脚本从文本文件读取服务名称并将每个服务的启动模式设置为“Disabled”（禁用）。

我想要您从这次演练中领会到在小段代码中构建 PowerShell 脚本（而不是试图在大段代码中应对整个脚本）的概念。您可以采用需要完成的任何管理任务并将其分成组成自身的多段代码，然后确定使每段代码独立工作的方法。您将看到，PowerShell 提供了将那些段代码联系起来的绝佳方法。并且您将发现，一次处理多个小段代码，您就会更轻松的开发最终脚本。

从文件读取名称

确定在 PowerShell 中读取文本文件的方法可能令人沮丧。如果我运行 `Help *file*`，那么我只能获得 `Out-File` cmdlet（用于将文本发送到文件）的帮助，而不会从中进行读取。根本没有帮助！但是，PowerShell cmdlet 名称遵循一种使用时对自己有利的逻辑。PowerShell 进行检索时，cmdlet 名称通常以 `Get` 开始。因此我在运行 `Help Get*` 显示出那些 cmdlet 后，就接着滚动列表并看到 `Get-Content`。看来大有希望！因此我运行 `Help Get-Content` 以了解更多与 `Get-Content` 的用途有关的信息（见图 1），并且看来 `Get-Content` 将能够满足我的需求。

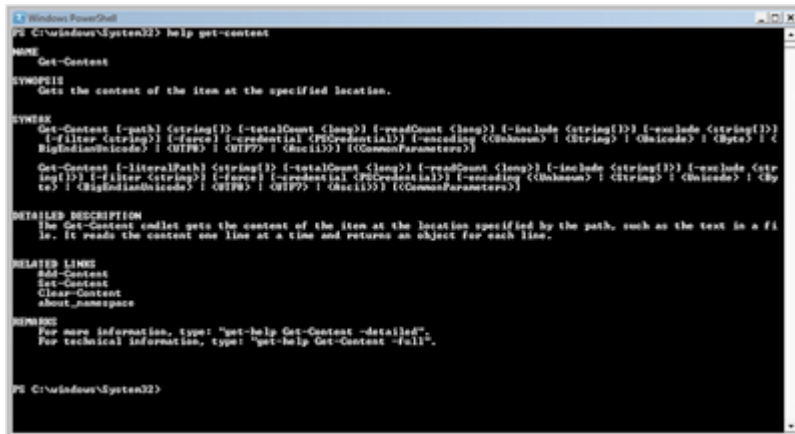


图 1 运行 `Help Get-Content` 获得更多信息

PowerShell 将几乎每件事情都作为对象来处理，文本文件也不例外。文本文件从技术上说是行的集合，文件中的每一行都可充当一种独立对象。因此，如果我已经创建了一个名为 `C:\services.txt` 的文本文件，并在其中填充了服务名称（在文件内部将每个名称放置在其本身的行上），PowerShell 可以使用 `Get-Content` cmdlet 逐个读取这些名称。由于此次演练的目的是显示交互式开发脚本的方法，因此我首先只运行 `Get-Content`，为其赋予我的文本文件的名称，然后再查看发生了什么：



```
PS C:\> get-content c:\services.txt

messenger

alerter

PS C:\>
```

正如所料，PowerShell 读取文件并显示名称。当然，只显示这些名称并不是我真正想要的，但现在我知道 `Get-Content` 正在以我所期望的方式工作。

更改服务

下一件我要做的事就是更改服务的启动模式。同样，我首先试图找到正确的 cmdlet。因此我运行 `Help *Service*`。此语句返回的列表很短，并且看起来只有 `Set-Service` cmdlet 才符合我的要求。在尝试将此 cmdlet 合并到脚本中之前，为了确保我了解它的工作方式，我要对其进行测试。运行 `Help Set-Service`，这将向我显示该 cmdlet 的工作方式，并且可通过快速测试进行确认：



```
PS C:\> set-service messenger -startuptype

disabled
```

```
PS C:\>
```

组合各部分

现在我需要将从文件中读取服务名称的功能同 Set-Service cmdlet 相组合，并且这正是 PowerShell 的功能强大的管道功能起作用之处。使用此管道，一个 cmdlet 的输出可以作为输入传递给第二个 cmdlet。管道传递整个对象。如果将对象的集合放入管道，那么每个对象将单独通过管道。这意味着，Get-Content 的输出（请记住，这是对象的集合）可以通过管道输送到 Set-Service。因为 Get-Content 传递的是集合，所以集合中的每个对象（或文本行）将单独通过管道输送到 Set-Service。其结果是对文本文件中的每一行都要执行一次 Set-Service。命令如下所示：



```
PS C:\> get-content c:\services.txt |  
  
set-service -startuptype disabled  
  
PS C:\>
```

以下为将发生的情况：

1. Get-Content cmdlet 执行，从而读取整个文件。将文件中的每一行都作为唯一的对象进行处理，将它们放在一起就是对象的集合。
2. 将对象的集合通过管道输送到 Set-Service。
3. 对于每个输入对象，管道都执行一次 Set-Service cmdlet。对于每次执行，都要将输入对象作为 cmdlet 的第一个参数（即服务名称）传递到 Set-Service。
4. Set-Service 使用其第一个参数的输入对象和指定的任何其他参数（在本例中是 -startuptype 参数）执行。

有趣的是，我实际上在此时已经完成了我的任务，可我甚至还不曾编写脚本。在 Cmd.exe 外壳中将很难实现相同的操作，而且需要若干行 VBScript 代码。但是 PowerShell 可以在一行中处理所有的代码。我的工作仍然没有完全完成。正如您所见，我的命令并没有提供大量的状态输出或反馈。除了没有出错误外，很难看出是否真正发生了什么。既然我已经掌握了完成任务所需的功能，那么我就可以开始通过了解真正的脚本编写来使其更好看。

Michael Murgolo 所做的 PowerShell Prompt Here

“Open Command Window Here”（在此处打开命令窗口）工具是最受欢迎的（也是我最喜爱的）Microsoft® PowerToys for Windows® 之一。“Open Command Window Here”（在此处打开命令窗口）可以作为 Microsoft PowerToys for Windows XP 的一部分或可在 Windows Server® 2003 Resource Kit Tools 中使用，使用它可以在 Windows 资源管理器中右键单击文件夹或驱动器以打开指向所选文件夹的命令窗口。

当我学习 PowerShell 时，发现自己希望它具有相同的功能。因此，我从 Windows Server 2003 Resource Kit Tools 获取“Open Command Window Here”（在此处打开命令窗口）的安装 .inf 文件 cmdhere.inf，然后对其进行修改，以创建 PowerShell Prompt Here 上下文菜单。此 .inf 文件包括在此边栏所基于的原始博客文章（可从 leeholmes.com/blog/PowerShellPromptHerePowerToy.aspx 获得）内。要安装此工具，只需右键单击 .inf 文件并选择“Install”（安装）即可。

创建此工具的 PowerShell 版本时，我发现原来的版本有错误——如果卸载“Open Command Window Here”（在此处打开命令窗口），它将留下失效的上下文菜单项。因此，我已经提供了一个 cmdhere.inf 的更新版本（还可以从原始博客文章中获得）。

这两种 PowerToy 都利用了以下的事实：这些上下文菜单项在注册表中与 Directory 和 Drive 对象类型关联的注册表项下配置。执行此操作时采用的方式与上下文菜单操作与文件类型相关联的方式相同。例如，在 Windows 资源管理器中右键单击 .txt 文件后，您就可以在列表顶部获得几种操作（如“打开”，“打印”和“编辑”）。要了解这些项的配置方式，让我们看一下 HKEY_CLASSES_ROOT 注册表配置单元。

如果您打开注册表编辑器并展开了 HKEY_CLASSES_ROOT 分支，您就会看到根据文件类型命名的注册表项（如 .doc、.txt 等等）。如果单击 .txt 注册表项，您就会看到 [“Default”（默认）] 值为 txtfile（见图 A）。这是与 .txt 文件相关联的对象类型。如果向下滚动并展开 txtfile 注册表项，然后在其下展开 shell 注册表项，您将看到根据 .txt 文件的某些上下文菜单项命名的注册表项。（您将不会看到所有的注册表项，因为还有创建上下文菜单的其他方法）。在这些注册表项的每一个下面都有一个命令项。命令项下面的 [“Default”（默认）] 值是命令行，选择该上下文菜单项时由 Windows 执行该命令行。

用于 cmd 提示符和 PowerShell 提示符的工具使用此技术来配置“Open Command Window Here”（在此处打开命令窗口）和 PowerShell Prompt Here 上下文菜单项。没有任何文件类型与驱动器和目录相关联，但在 HKEY_CLASSES_ROOT 下却有与这些对象相关联的注册表项“Drive”（驱动器）和“Directory”（目录）。

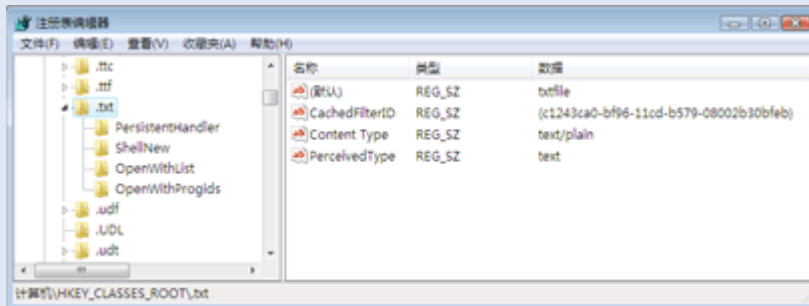


图 A 在注册表中配置上下文菜单项

Michael Murgolo 是 Microsoft Consulting Services 的一名高级基础结构顾问。他关注的领域有操作系统、部署、网络服务、Active Directory、系统管理、自动化和补丁管理。

交互式脚本编写

我需要做的一件事就是针对 C:\services.txt 中列出的每个服务执行多个 cmdlet。这样，我就可以输出自己所喜欢的服务名称和任何其他信息，从而跟踪脚本的进度。

之前，我使用管道将对象从一个 cmdlet 传递到另一个 cmdlet。但是，这一次我要使用的称为 Foreach 构造的技术（我在上个月的专栏中进行过介绍）更像脚本。Foreach 构造可以接受对象的集合，并且它将针对该集合中的每个对象执行多个 cmdlet。我指定了一个变量，该变量表示每次通过循环时的当前对象。例如，构造可能这样开始：



```
foreach ($service in get-content c:\services.txt)
```

我仍将执行相同的 Get-Content cmdlet 来检索文本文件的内容。这一次我要求 Foreach 构造遍历由 Get-Content 返回的对象的集合。每个对象执行一次循环，而当前对象则放在变量 \$service 中。现在我只需要指定想在循环中执行的代码。我将首先尝试复制原始的单行命令（这将最小化复杂性并确保我不会丢失任何功能）：



```
PS C:\> foreach ($service in get-content c:\services.txt) {  
  
    >> set-service $service -startuptype disabled  
  
    >> }  
  
    >>  
  
PS C:\>
```

此时发生了一些有趣的事情。请注意，我使用一个左花括号结束了 Foreach 构造的第一行。左花括号和右花括号中的所有内容都被认为在 Foreach 循环内，并且针对输入集合中的每个对象都将执行一次该内容。在我键入 { 并按下 Enter 之后，请注意，PowerShell 将其提示符更改为 >>（见图 2）。这表示它知道我已开始进行某种构造，并且正在等待我完成构造。接下来我键入我的

Set-Service cmdlet。这一次，我使用 `$service` 作为第一个参数，这是由于 `$service` 表示的是从我的文本文件读取的当前服务名称。在下一行，我使用右花括号结束构造。按 Enter 两次后，PowerShell 立即执行我的代码。

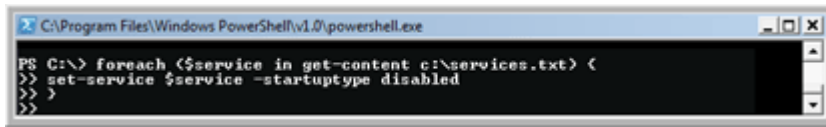


图 2 PowerShell 知道已启动某构造

没错，立即执行。我键入的内容看起来像脚本，但 PowerShell 实际上正在实时执行它，而且它并没有存储在文本文件中的任何地方。现在我将重新键入全部内容，添加一行代码以输出当前服务名称：



```
PS C:\> foreach ($service in get-content c:\services.txt) {  
  
    >> set-service $service -startuptype disabled  
  
    >> "Disabling $service"  
  
    >> }  
  
>>  
  
Disabling messenger  
  
Disabling alerter  
  
PS C:\>
```

请注意，我正在要求它显示内容，并且我已经将要显示的内容括在双引号内。引号告诉 PowerShell，这是一个文本的字符串，而不是另一个命令。但是，当您使用双引号（而不是单引号）时，PowerShell 会扫描文本字符串中的任何变量。如果它找到了任何变量，它会将变量名替换为变量的实际值。因此，执行此代码时，您可以看到正在显示当前服务名称。

这仍不是脚本！

到现在为止，我一直在交互式地使用 PowerShell，这是立即看到我所编写的内容的结果的好方法。但是，在某些时候，重新键入这些代码行将变得十分单调乏味。这是 PowerShell 也可以运行脚本的原因。实际上，PowerShell 脚本遵循的是脚本一词最为文字化的定义：PowerShell 基本上只在脚本文本文件中进行读取，然后“键入”它所发现的每一行，看来与手动键入行极其相似。这意味着到现在为止我所执行的每项内容都可以粘贴到脚本文件中。因此，我将使用记事本创建名为 `disableservices.ps1` 的文件，然后在其中粘贴以下内容：



```
foreach ($service in get-content c:\services.txt) {  
  
    set-service $service -startuptype disabled  
  
    "Disabling $service"  
  
}
```

我已将此文件放在名为 `C:\test` 的文件夹中。现在我将试图从 PowerShell 内部运行此文件：



```
PS C:\test> disableservices
```

```
'disableservices' is not recognized as a cmdlet, function, operable program, or
<script file.
```

```
At line:1 char:15
```

```
+ disableservices <<<<
```

```
PS C:\test>
```

哎哟。出什么问题啦？PowerShell 现在显示位于 C:\test 文件夹，但它并没有找到我的脚本。这是为什么？由于安全约束，PowerShell 设计为不能运行来自当前文件夹的任何脚本，以防止任何脚本劫持操作系统命令。例如，我无法创建一个名为 dir.ps1 的脚本并用它覆盖正常的 dir 命令。如果我需要运行当前文件夹中的脚本，就必须指定相对路径如下：



```
PS C:\test> ./disableservices
```

```
The file C:\test\disableservices.ps1 cannot be loaded.
```

```
The execution of scripts is disabled on this system.
```

```
Please see "get-help about_signing" for more details.
```

```
At line:1 char:17
```

```
+ ./disableservices <<<<
```

```
PS C:\test>
```

现在要做什么？它仍然不正常工作。我已经指定了正确的路径，但 PowerShell 却说它无法运行脚本。那是因为，默认情况下，PowerShell 无法运行脚本。同样，这个安全措施旨在防止发生我们使用 VBScript 时已经遇到的问题。默认情况下，恶意脚本无法在 PowerShell 下执行，这是因为在默认情况下任何脚本都不能执行！为了运行脚本，我需要显式更改执行策略：



```
PS C:\test> set-executionpolicy remotesigned
```

使用 RemoteSigned 执行策略，未签名的脚本可以从本地计算机执行（下载的脚本仍然必须签名后才可运行）。更好的策略是 AllSigned，该策略仅执行已使用受信任发布方所颁发的证书进行过数字签名的脚本。但是，我手边没有证书，因此我无法对我的脚本进行签名，这种情况下 RemoteSigned 就是一种不错的选择。现在我再次试图运行我的脚本：



```
PS C:\test> ./disableservices
```

```
Disabling messenger
```

```
Disabling alerter
```

```
PS C:\test>
```

我要指出的是，我们正在使用的 RemoteSigned 执行策略并不是一个极佳的选择，它只是一个较好的选择而已。但是还有一个更好的解决方案。获取代码签名的证书，使用 PowerShell Set-AuthenticodeSignature cmdlet 来对我的脚本进行签名，然后将执行策略设置为更加安全的 AllSigned 策略，这样做将安全得多。

还有另一种策略 Unrestricted（您一定要避免使用）。使用此策略，所有的脚本（甚至来自远程位置的恶意脚本）可以不受限制地在您的计算机上运行，这会将您置于很危险的境地。因此，我建议无论如何都不要使用 Unrestricted 策略。

即时结果

使用 PowerShell 中的交互式脚本编写功能，您可以快速地建立脚本的原型或甚至只是小段脚本的原型。您获得了即时结果，因此可以轻松地调整您的脚本以获得真正想要的结果。操作结束时，您可以将代码移动到 .ps1 文件中，使其成为永久代码并且将来也容易找到。还要记住：理想情况下，您应该对那些 .ps1 文件进行数字签名，从而您可以将 PowerShell 设置为 AllSigned 执行策略，这是允许脚本执行的最安全的策略

PowerShell 变量的力量

如果使用基于 Windows 的脚本语言（如 VBScript 或 KiXtart），您会习惯于认为变量只不过是一种数据存储机制。PowerShell 也具有变量，但这些变量要比早期脚本语言中的变量强大得多。Windows

PowerShell 变量实际上映射到 Microsoft® .NET Framework 中的基础类。在 Framework 中，变量是对象，这就意味着变量可以存储数据，也可以通过多种方法进行处理。实际上，正是由于 PowerShell™ 中变量的强大功能，使得 PowerShell 脚本语言不包含任何内部数据处理函数。它根本不需要这些函数，因为变量本身就已经提供了这种功能。

声明变量

尽管 PowerShell 中的 New-Variable cmdlet 确实允许您声明变量并为其赋予初始值，但您并不一定要使用 cmdlet。作为一种替代方式，您只需通过为变量赋值即可即时地创建新的变量：



```
$var = "Hello"
```

在 PowerShell 中，变量名始终以一个美元符号 (\$) 开头，并可以混合使用字母、数字、符号、甚至空格（但如果使用空格，则需要将变量括在花括号中，例如 \${My Variable} = "Hello"）。此示例创建了一个名为 \$var 的新变量，并为其赋予初始值 "Hello"。因为本例中的变量值是字符串，所以 PowerShell 会使用 String 数据类型来存储该值。在 .NET Framework 中，这些字符串变量对应于 System.String 类。System.String 类或许拥有任何变量类型的大部分内置功能。假设我要查看 \$var 值的全小写版本，则可以执行以下命令：



```
PS C:\> $var.ToLower()
```

```
hello
```

```
PS C:\>
```

ToLower 方法内置于 System.String 类中，可以使字符串的值以全小写形式来表示。但它并不更改变量 \$var 的实际内容。要查看 System.String 类所有功能的完整列表，请像下面这样通过管道将一个字符串变量传递给 Get-Member cmdlet：



```
$var | get-member
```

图 1 显示了输出结果，其中包含许多用于处理字符串的方法。在 PowerShell 中，VBScript 字符串处理函数所提供的几乎所有功能都改由字符串变量方法来提供。

```

Windows PowerShell
PS C:\> $var = "Hello"
PS C:\> $var.ToLower()
hello
PS C:\> $var | get-member

    TypeName: System.String

Name             MemberType      Definition
-----
Clone             Method          System.Object Clone()
CompareTo         Method          System.Int32 CompareTo(Object value), System.Int32 CompareTo(String strB)
Contains          Method          System.Boolean Contains(String value)
CopyTo            Method          System.Void CopyTo(Int32 sourceIndex, Char[] destination, Int32 destination)
EndsWith          Method          System.Boolean EndsWith(String value), System.Boolean EndsWith(String value)
Equals            Method          System.Boolean Equals(Object obj), System.Boolean Equals(String value), System
GetEnumerator     Method          System.CharEnumerator GetEnumerator()
GetHashCode       Method          System.Int32 GetHashCode()
GetType           Method          System.Type GetType()
GetTypeCode       Method          System.TypeCode GetTypeCode()
get_Chars         Method          System.Char get_Chars(Int32 index)
get_Length        Method          System.Int32 get_Length()
IndexOf           Method          System.Int32 IndexOf(Char value, Int32 startIndex, Int32 count), System.Int32
IndexOfAny        Method          System.Int32 IndexOfAny(Char[] anyOf, Int32 startIndex, Int32 count), System
Insert            Method          System.String Insert(Int32 startIndex, String value)
IsNormalized      Method          System.Boolean IsNormalized(), System.Boolean IsNormalized(NormalizationForm
LastIndexOf       Method          System.Int32 LastIndexOf(Char value, Int32 startIndex, Int32 count), System.I
LastIndexOfAny    Method          System.Int32 LastIndexOfAny(Char[] anyOf, Int32 startIndex, Int32 count), Sys
Normalize         Method          System.String Normalize(), System.String Normalize(NormalizationForm normaliz
PadLeft           Method          System.String PadLeft(Int32 totalWidth), System.String PadLeft(Int32 totalWid
PadRight          Method          System.String PadRight(Int32 totalWidth), System.String PadRight(Int32 totalW
Remove            Method          System.String Remove(Int32 startIndex, Int32 count), System.String Remove(Int
Replace           Method          System.String Replace(Char oldChar, Char newChar), System.String Replace(Stri
Split             Method          System.String[] Split(Params Char[] separator), System.String[] Split(Char[]
StartsWith        Method          System.Boolean StartsWith(String value), System.Boolean StartsWith(String val
Substring         Method          System.String Substring(Int32 startIndex), System.String Substring(Int32 star
ToCharArray       Method          System.Char[] ToCharArray(), System.Char[] ToCharArray(Int32 startIndex, Int3
ToLower           Method          System.String ToLower(), System.String ToLower(CultureInfo culture)
ToLowerInvariant  Method          System.String ToLowerInvariant()
ToString          Method          System.String ToString(), System.String ToString(IFormatProvider provider)
ToUpper           Method          System.String ToUpper(), System.String ToUpper(CultureInfo culture)
ToUpperInvariant  Method          System.String ToUpperInvariant()
Trim              Method          System.String Trim(Params Char[] trimChars), System.String Trim()
TrimEnd           Method          System.String TrimEnd(Params Char[] trimChars)
TrimStart         Method          System.String TrimStart(Params Char[] trimChars)
Chars             ParameterizedProperty System.Char Chars(Int32 index) {get;}
Length            Property         System.Int32 Length {get;}

PS C:\>

```

Figure 1 A look at the System.String class output

在管理上下文中，字符串变量的许多功能比语言（如 VBScript）中的字符串函数要有用得多。假设您编写了一个用于从文件中读入通用命名约定（UNC）路径并对其进行操作的脚本。在尝试使用路径之前，您可能要验证所读入的每个路径是否为 UNC 路径。可通过 StartsWith 方法来确认字符串值是否以 UNC 所必需的反斜线字符开头：



```

PS C:\> $path = "\\Server\Share"

PS C:\> $path.StartsWith("\\")

True

PS C:\>

```

由于 StartsWith 返回 True 或 False 值，因此可将其用在逻辑构造中：



```

if ($path.StartsWith("\\")) {

    # code goes here

}

```

PowerShell 甚至还提供了一种自动完成变量方法的形式，从而减少了必要的键入工作量。如果 \$var 包含一个字符串，您可以键入



`$var.`

然后按 Tab ,这样会弹出 \$var 变量的第一个方法名。再按一次 Tab 键可移到下一个方法 ,而按 Shift+Tab 会调出前一个方法。您可以通过这种方式遍历所有可用的方法 ,直到找到您想要的那一个。

变量混淆

到目前为止 ,我的示例一直在让 PowerShell 来确定变量的数据类型。将字符串赋给变量实质上会强制变量成为 System.String 类。而另一方面 ,将数字赋给变量通常会使变量成为 Integer (或者更确切地说 ,是成为 Int32 ,其可以存储特定范围的值)。例如 ,考虑以下脚本 :



```
PS C:\> $int = 5
```

```
PS C:\> $int | get-member
```

```
TypeName: System.Int32
```

这一被截断的输出说明 PowerShell 将 \$int 视为 Int32 ,Int32 有其自己的方法和属性集。而实际上 ,其方法和属性要比 String 类型的方法和属性少得多。

将 \$int 作为 Int32 是因为其值没有括在引号中而且仅由数字组成。如果将其值括在引号中 ,则它会被视为 System.String。

让 PowerShell 来决定要使用的数据类型并不是总能达到您所希望的效果。假设您要从文件读取变量值 ,而又希望这些值始终被视为字符串。但是 ,其中一些值可能只包含数字 ,这就增加了 PowerShell 将其视为 Int32 或其他数字类型的可能性。这样可能会使脚本出现问题。如果 PowerShell 不将值识别为字符串 ,则 System.String 类的所有方法都无法使用 (而您的脚本可能要依赖于这些无法使用的方法中的一个)。

要解决这一问题 ,可以在第一次使用变量时声明其类型 ,从而强制 PowerShell 将变量按该特定类型处理。示例如下 :



```
[string]$var = 5
```

\$var 通常应为 Int32 类型 ,但在此示例中我强制 PowerShell 将 \$var 作为 String ,从而确保了我可以使用与 System.String 类相关的所有方法。变量类型声明也提供了一种便捷的代码自我文档化方式 ,因为现在哪种类型的数据要进入 \$var 中是显而易见的。实际上 ,我已经养成了为所有变量声明特定类型的习惯 ,而完全不管是不是在 PowerShell 中。这使得我脚本的行为更加可以预测 ,在许多情况下为我节省了大量的调试时间。

正如您所料 ,尽管强制声明变量未必不好 ,但也确实会产生影响。以下面的代码为例 ,代码中未声明变量的类型 :



```
PS C:\> $me = 5
```

```
PS C:\> $me = "Don"
```

\$me 变量最初为 Int32 类型 ,但在添加了值 “Don” 后 PowerShell 将该变量改为 String 类型。如果变量尚未被明确地设置为某个特定的类型 ,PowerShell 可以根据需要更改变量的类型。

在本例中 ,我使用 [int] 类型名将 \$me 强制为 Int32 类型 :



```
PS C:\> [int]$me = 5
```

```
PS C:\> $me = "Don"
```

```
Cannot convert value "Don" to type
```



```
"System.Int32". Error: "Input string
```

```
was not in a correct format."
```

```
At line:1 char:4
```

```
+ $me <<<< = "Don"
```

然后，当我尝试为其赋予字符串值时，会出现一条错误消息。因为我已明确地将 \$me 强制为 Int32 类型，所以 PowerShell 要将字符串“Don”转换为整型值。然而这样做是不可能的，而且将 \$me 的类型更改为 String 也是不可能的。

许多类型

PowerShell 中提供了很多的类型。实际上，您可以使用任何的 .NET Framework 类型（可用的类型有数百个）。但 PowerShell 为常见的数据类型定义了许多快捷方式。图 2 只是部分列表，显示了 10 个最常用类型的快捷方式。有关完整列表，请参考 PowerShell 文档。

Figure 2 常见类型快捷方式

快捷方式	数据类型
[datetime]]	日期或时间
[string]	字符串
[char]	单个字符
[double]	双精度浮点数
[single]	单精度浮点数
[int]	32 位整数
[wmi]	Windows Management Instrumentation (WMI) 实例或集合
[adsis]]	Active Directory 服务对象
[wmiclass]]	WMI 类
[Boolean]	True 或 False 值

您还可以使用完整的 .NET Framework 类名来声明变量的类型，如下所示：



```
[System.Int32]$int = 5
```

您可以通过这一技术来使用存在于 .NET Framework 中但在 PowerShell 中没有特定快捷方式的类型。

扩展类型

PowerShell 最爽的地方或许就是它能够扩展这些变量类型的功能。在 PowerShell 安装文件夹（通常在 %systemroot\system32\windowspowershell\v1.0 中，但在 64 位系统上您会发现此路径略有不同）中，您会找到名为 types.ps1xml 的文件。可以在记事本或 XML 编辑器（如 PrimalScript）中编辑此文件。默认情况下，尽管其中列出了许多其他类型，但并未列出 System.String 类。但我可以向 System.String 类型添加新的功能，只需将其扩展添加到 types.ps1xml 文件中即可实现。

进行更改后，需要关闭 PowerShell，然后再重新打开。我还需要确保本地脚本执行策略允许执行未签名的脚本，因为我未对 types.ps1xml 进行签名：



Set-executionpolicy remotesigned

现在，只要重新启动 PowerShell，我就可以使用新功能了，如下所示：




```
PS C:\> [string]$comp = "localhost"
```

```
PS C:\> $comp.canping
```

```
True
```

```
PS C:\>
```

正如您所看到的那样，我已经将 CanPing 属性添加到 System.String 类。这样会返回 True 或 False 值，来指示本地计算机能否 ping 字符串中所含的地址。在图 3 中，您会注意到 WMI 查询使用了一个特殊的变量 \$this。\$this 变量代表字符串中所含的当前值，也是字符串变量内容传递给 WMI 查询的方式。

 Figure 3 扩展 System.String 类型



```
<Type>
```

```
<Name>System.String</Name>
```

```
<Members>
```

```
<ScriptProperty>
```

```
<Name>CanPing</Name>
```

```
<GetScriptBlock>
```

```
$wmi = get-wmiobject -query "SELECT *
```

```
FROM Win32_PingStatus WHERE Address = '$this'"
```

```

        if ($wmi.StatusCode -eq 0) {

            $true

        } else {

            $false

        }

    </GetScriptBlock>

</ScriptProperty>

</Members>

</Type>

```

让变量工作

PowerShell 提供了灵活的、功能众多的变量类型，也提供了同样灵活的类型功能扩展系统。这些功能使您可以很容易地为您的脚本注入大量的功能。实际上，变量可以成为复杂脚本中的主要构建基块，在许多情况下提供着通常在更为复杂的函数中才能找到的高级功能

PowerShell 数据的筛选与格式化

毫无疑问，Windows PowerShell 在您无需进行大量的工作的情况下，就可以返回许多宝贵的信息。以简单的 `Get-WMIObject` cmdlet 为例，它可以用来返回本地计算机上的服务列表。默认视图列出了所运行服务的状态、名称和启动模式，其实质上是一种“服务”控制台的命令行视图。Windows PowerShell™ 能使其更加容易实现。

不过，还可以为服务提供大量更加有用的信息，而不仅仅是在默认视图中显示出来的部分。试试运行这个命令：



```

$s = get-wmiobject win32_service

$s[0] | gm

```

第一行命令返回所有服务的集合（或者，更确切地说是来自 Windows® Management Instrumentation 或 WMI 的 `Win32_Service` 类的所有实例），并将其存储在变量 `$s` 中。第二行命令调用此集合中的第一个服务（在方括号中指明的序号零），并将其传送到 `Get-Member` cmdlet（在此使用它的别名，`gm`）。输出结果显示此数据类型所有可用的属性和方法。我决定使用 WMI 的 `Win32_Service` 类，而不用 Windows PowerShell 内置的 `Get-Service` cmdlet，因为与 Microsoft® .NET `ServiceController` 对象相比，WMI 实际上能够提供更多的信息，其中包括 `StartName` 属性。该属性可告诉我在其下运行服务的用户账户的名称。您可以使用如下方法检查单个实例（如第一个）的名称：



```

PS C:\> $s[0].StartName

LocalSystem

```

然而，这些服务并没有任何特定的排列顺序，所以用集中的序号来引用服务效果不是很好。（其通常以字母顺序排列，但并非始终如此。）从管理的角度来讲，或许您最感兴趣的是全部服务的列表和每个用来登陆的账户。这样对于某些方面有相当的便利性，比如说遵从性审核。那么让我们稍作停留，来看一看 WMI 的默认设置。我将使用 Get-WMIObject 的缩写，gwmi（见图 1）。

Figure 1 使用 gwmi



```
PS C:\> gwmi win32_service

ExitCode : 0

Name      : AcrSch2Svc

ProcessId : 1712

StartMode : Auto

State     : Running

Status    : OK

ExitCode  : 1077

Name      : Adobe LM Service

ProcessId : 0

StartMode : Manual

State     : Stopped

Status    : OK
```

显然这仅是一个示例，但是您可以看到输出结果并不太适用于管理报告。我想知道的是服务的名称和 StartName（即服务在其下运行的账户）。我也希望报告的格式可读性更强，而不是这种感觉有点笨笨的列表。这就是 Windows PowerShell 中具备强大格式化和数据筛选功能的 cmdlet 发挥作用的地方。

获取您所需要的数据

我将从 Get-WMIObject 筛选数据作为开始，因此现在显示的消息就是我最感兴趣的属性。最佳的实现方式是使用 Windows PowerShell Select-Object cmdlet，其简写的别名为 select。Select 用于接收对象的集合（诸如由 Get-WMIObject 所返回的集合），并显示这些对象所需要的属性。这意味着我可以将 gwmi 的输出传送到 select，并指定我所感兴趣的两种属性。图 2 显示了结果，其中包含了用表格形式所显示的 Name 和 StartName 属性。

```
Windows PowerShell

PS C:\> gwm win32_service | select Name,StartName

Name                                     StartName
-----
AcrSch2Svc                             LocalSystem
Adobe LM Service                       LocalSystem
Alerter                                NT AUTHORITY\LocalService
ALG                                     NT AUTHORITY\LocalService
AppMgmt                                LocalSystem
aspnet_state                           NT AUTHORITY\NetworkService
Ati HotKey Poller                      LocalSystem
ATI Smart                              LocalSystem
AudioSvc                               LocalSystem
BITS                                   LocalSystem
Browser                                LocalSystem
CISvc                                  LocalSystem
ClipSvc                                LocalSystem
clr_optimization_v2.0.50727_32        LocalSystem
COMSysApp                              LocalSystem
CryptSvc                               LocalSystem
DcomLaunch                             LocalSystem
Dhcp                                    LocalSystem
dmadmin                                LocalSystem
dnserver                               LocalSystem
Dnscache                              NT AUTHORITY\NetworkService
ERSvc                                  LocalSystem
Eventlog                               LocalSystem
EventSystem                            LocalSystem
FastUserSwitchingCompatibility         LocalSystem
GrooveAuditService                    LocalSystem
GrooveInstallerService                LocalSystem
GrooveRunOnceInstaller                 LocalSystem
helpsvc                                LocalSystem
HidSvc                                 LocalSystem
HTTPFilter                             LocalSystem
IDriverT                               LocalSystem
IISADMIN                              LocalSystem
ImapiService                           LocalSystem
InstallShield Licensing Service       LocalSystem
iPod Service                           LocalSystem
lanmanserver                           LocalSystem
lanmanworkstation                     LocalSystem
LBTServ                                LocalSystem
LmHosts                                NT AUTHORITY\LocalService
MDM                                    LocalSystem
```

图 2 仅查看表格中的 Name 和 StartName 属性

假设我是为遵从性审核目的而生成此报告，那么其中包含的信息确实多了些。由于某些服务已被禁用，因此任何读取禁用服务理论上会使用哪些账户相关的输出结果，均无实际意义。因而，我会筛选掉所有具备“禁用”启动类型的服务，也就是 Win32_Service 类的 StartMode 属性。

Windows PowerShell 使用 Where-Object cmdlet (其缩写为 where) 来筛选对象。where cmdlet 接收输入对象的集合并通过脚本块逐一运行，基于一套定义好的标准来决定是否每个对象都将在 cmdlet 的输出中显示。每个符合标准的对象生成的比较结果为 True，并包含在输出结果中；而产生 False 值的对象不会包含在内。

因此，我决定只需要那些 StartMode 属性设为禁用的对象。在 Windows PowerShell 内容中，该评估是这样的：



```
$object.StartMode -eq "Disabled"
```

当然，\$object 仅仅是一个例子。事实上我并没有在编写脚本，而且我没有命名为 \$object 的变量。在 where 使用的脚本块中，我却有一个特殊的名为 \$_ 的变量，它代表当前正在被 cmdlet 评估的对象。因此我的 where 脚本块可能是这样的：



```
$_ .StartMode -eq "Disabled"
```

我可以很轻松地对其进行测试：




```
PS C:\> gwmi win32_service | where
```

```
{$_StartMode -eq "Disabled" }
```

```
ExitCode : 1077
```

```
Name : Alerter
```

```
ProcessId : 0
```

```
StartMode : Disabled
```

```
State : Stopped
```

```
Status : OK
```

当然，在此快速测试中，输出结果会以默认的列表样式返回显示。同时，为了节约空间，在此仅包含了一个服务。但您会注意到此输出是反向的。我已包含了禁用服务在内，而非将它们排除在外。这是因为得让我的脚本块反向：我需要包括所有 StartMode 属性不是禁用的服务，正如对我的示例进行修改后那样：



```
PS C:\> gwmi win32_service | where
```

```
{$_StartMode -ne "Disabled" }
```

```
ExitCode : 0
```

```
Name : AcrSch2Svc
```

```
ProcessId : 1712
```

```
StartMode : Auto
```

```
State : Running
```

```
Status : OK
```

这样更好！现在输出结果中仅包含我真正感兴趣的服务，我能够再一次将结果传送到 select 并指定我需要的属性：



```
gwmi win32_service | where {$_StartMode -ne "Disabled" } | select name,startname
```

将数据从一个 cmdlet 传送到另一个再到另一个 cmdlet 的过程，真正说明了 Windows PowerShell 的功能。我依然没有编写脚本，但是我却实现了从大量数据集中筛选出我所需要的输出结果这一目的。

合适的外观

您应该注意，select 的输出仍然只是一组对象。当我使用 Windows PowerShell 检索格式美观的表格时，正是 PowerShell.exe 在对这些对象进行编译。换句话说，Windows PowerShell 知道我作为一个普通人不可能看见这些对象，因此它采用文本的形式来呈现这些对象。在这种情况下，对象以表格的形式呈现，同时每个属性都有相应的一列予以对应，如图 2 所示。

看起来这样对于审核目的已经足够了。但我想再说一遍，也许不是。不同的人有不同的需求，Windows PowerShell 并不会假设您的特定需求。相反，它为您提供最好的工具来格式化输出。四个内置的 cmdlets - Format-List、Format-Custom、Format-Table 和 Format-Wide - 用来接收对象的集合（比如由 select 返回的集合），并用不同的方式格式化这些对象。Windows PowerShell 基本上使用 Format-Table 来格式化 select cmdlet 的输出结果。想看看不同的外观，可以试一下 Format-List：



```
gwmi win32_service | where {$_.StartMode -ne "Disabled" } |  
select name,startname | format-list
```

结果有点像这个示例：



```
name      : AcrSch2Svc  
  
startname : LocalSystem  
  
name      : Adobe LM Service  
  
startname : LocalSystem
```

Format-Wide cmdlet 会生成一个多栏的列表，默认情况下为每个对象的第一个属性。以下面这一行为例：



```
gwmi win32_service | where {$_.StartMode -ne "Disabled" } |  
select name,startname | format-wide
```

它生成一张服务名称列表，但这并不是我想要的。输出中没有 StartName（见**图 3**），而在审核报告中，StartName 是我所需要的一条重要信息。

```
Windows PowerShell
PS C:\> gwmi win32_service | where {$_.StartMode -ne "Disabled"} | select name,startname | format-w

AcrSch2Svc                                Adobe LM Service
ALG                                        AppMgmt
aspnet_state                             Ati HotKey Poller
ATI Smart                                AudioSrv
BITS                                     Browser
Cisvc                                    clr_optimization_v2.0.50727_32
COMSysApp                                CryptSvc
DcomLaunch                               Dhcp
dmadmin                                  dnserver
Dnscache                                ERSvc
Eventlog                                 EventSystem
FastUserSwitchingCompatibility           FLEXnet Licensing Service
helpsvc                                  HidServ
HTTPFilter                              IDriverT
ImapiService                             InstallShield Licensing Service
iPod Service                             lanmanserver
lanmanworkstation                       LightScribeService
LmHosts                                  MDM
Microsoft Office Groove Audit Service   mnmsrvc
MSDTC                                    MSIServer
Netlogon                                 Netman
Nla                                       NtLmSsp
NtmsSvc                                  odserv
ose                                       PlugPlay
Pml Driver HPZ12                         PolicyAgent
ProtectedStorage                         RasAuto
RasMan                                   RDSessMgr
RemoteRegistry                           RpcLocator
RpcSs                                    RSUP
SamSs                                    SCardSvr
Schedule                                 seclogon
SENS                                     SharedAccess
ShellHWDetection                         Spooler
srsvc                                    SSDPSRV
stisvc                                   SvPro
SysmonLog                               TapiSrv
TermService                             Themes
TrkWks                                  UleadBurninHelper
upnphost                                UPS
usnsvc                                  USS
W32Time                                 WebClient
winmgmt                                WmdmPmSN
Wmi                                       WmiApSrv
WMPNetworkSvc                           wscsvc
WSearch                                 wuauserv
WudfSvc                                 WZCSUC
xmlprov
```

图 3 Format-Wide cmdlet 显示的输出结果中省略了关键信息 -- StartName

因为我仅需处理两种属性,所以 Format-Table 或 Format-List 看起来都是可以接受的。但是审核人员看到屏幕上的上述信息可能不会感到高兴。她可能更喜欢某种格式的文件。

导出数据

那么,审计人员希望如何查看数据呢?输出服务列表并登录到 CSV(逗号分割值)文件可能就足够了,因为可用 Microsoft Excel® 很方便的打开该文件。要创建一个 CSV 文件,只需将您的输出传送到 Windows PowerShell Export-CSV cmdlet:



```
gwmi win32_service | where {$_.StartMode -ne "Disabled" } |  
  
select name,startname | export-csv c:\services.csv
```

当然,在当今社会,CSV 似乎有一点过时。或许审计人员会更希望数据以网页的形式显示到 Intranet 服务器上。为了做到这一点,需要使用 ConvertTo-HTML cmdlet 将输出转换成 HTML:



```
gwmi win32_service | where {$_.StartMode -ne "Disabled" } |  
  
select name,startname | convertto-html
```

原始的 HTML 难以查看，因此您需要将输出结果写到一个文件中：



```
gwmi win32_service | where {$_.StartMode -ne "Disabled" } |  
select name,startname | convertto-html | out-file c:\services.html
```

图 4 显示的结果是一种格式美观的 HTML 页面，可以发布到任何 Web 服务器上，而不需要做进一步的修改。

The screenshot shows a Windows Internet Explorer browser window with the title 'HTML TABLE - Windows Internet Explorer'. The address bar shows 'C:\services.html'. The table displayed has two columns: 'name' and 'startname'. The table contains 20 rows of service names and their start names.

name	startname
1-vmsrvc	LocalSystem
AeLookupSvc	localSystem
ALG	NT AUTHORITY\LocalService
Appinfo	LocalSystem
AppMgmt	LocalSystem
AudioEndpointBuilder	LocalSystem
Audiosrv	NT AUTHORITY\LocalService
BFE	NT AUTHORITY\LocalService
BITS	LocalSystem
Browser	LocalSystem
CertPropSvc	LocalSystem
clr_optimization_v2.0.50727_32	LocalSystem
COMSysApp	LocalSystem
CryptSvc	NT Authority\NetworkService
CscService	LocalSystem
DcomLaunch	LocalSystem
DFSR	LocalSystem
Dhcp	NT Authority\LocalService

图 4 用格式美观的 HTML 页面显示输出结果

相关事实

Windows PowerShell 为您提供访问大量管理数据的快速途径。然而，从业务的角度来说，这些数据在原始状态下并不是始终都有用的。

通过筛选数据（使用 `where`）、选择所需要的对象属性（使用 `select`）以及应用相应的格式化选项（比如 `Format-Table` 或 `Format-List`），您可以轻松快捷地将管理数据转换为有用的信息。然后，通过将数据导出为一种易于共享的文件格式，您可以将该信息进行共享并与组织内其他同事交流宝贵的信息。

PowerShell WMI 连接

我在 VBScript 领域非常依赖的技术之一是 Windows Management Instrumentation (WMI)。有趣的是, PowerShell 与 WMI 有着密切的联系, 这不仅仅是在技术意义上。PowerShell 的设计者 Jeffrey Snover 在创建 Wmic.exe 时也担任了重要角色, Wmic.exe 是 Windows Server® 2003 中与 WMI 一起使用的命令行工具。Wmic.exe 在许多方面表现了与 PowerShell™ 类似的工作方式 (有关 WMIC 的详细信息, 请参阅 John Kelbley 在 2006 年 9 月的《TechNet 杂志》上发表的文章, *您可以在线访问, 网址是 microsoft.com/technet/technetmag/issues/2006/09/WMIData*)。

PowerShell 以一致、基于对象的方式 (与通过 Shell 的其他功能获得的方式相同) 支持所提供的 WMI。与以前的技术 (如 VBScript) 相比, 这使得 WMI 更容易学习和使用 (尤其是在一些特殊情况下)。

WMI 入门

如果您阅读有关脚本的书籍和文章, 几乎都会提到 WMI。不过, 在实际使用 WMI 时, 如果忘记其内部构造方式, 则会感到非常迷茫, 而且 WMI 的构造方式对于它在 PowerShell 中的工作方式极其重要。

WMI 主要是一个组织类的系统, 表示 Windows® 操作系统和其他基于 Windows 的硬件和软件产品的管理信息。类实际上就是对一些给定软件或硬件组件进程的属性和功能的抽象描述。例如, 逻辑磁盘类可能描述具有一个序列号、一个固定的存储容量、一定的可用容量等内容的设备。同时, 描述 Windows 服务的类可能指定该服务有一个名称、可以启动和停止, 以及指定其当前状态等。

在 WMI 中, 类表示 WMI 可以管理的所有内容。如果 WMI 没有可用于某些内容的类, 则它无法管理该组件。Microsoft 在 msdn2.microsoft.com/aa394554.aspx 中记录了核心 Windows WMI 类; 其他产品 (如 Internet 信息服务、SQL Server™) 分别记录了它们的 WMI 类。

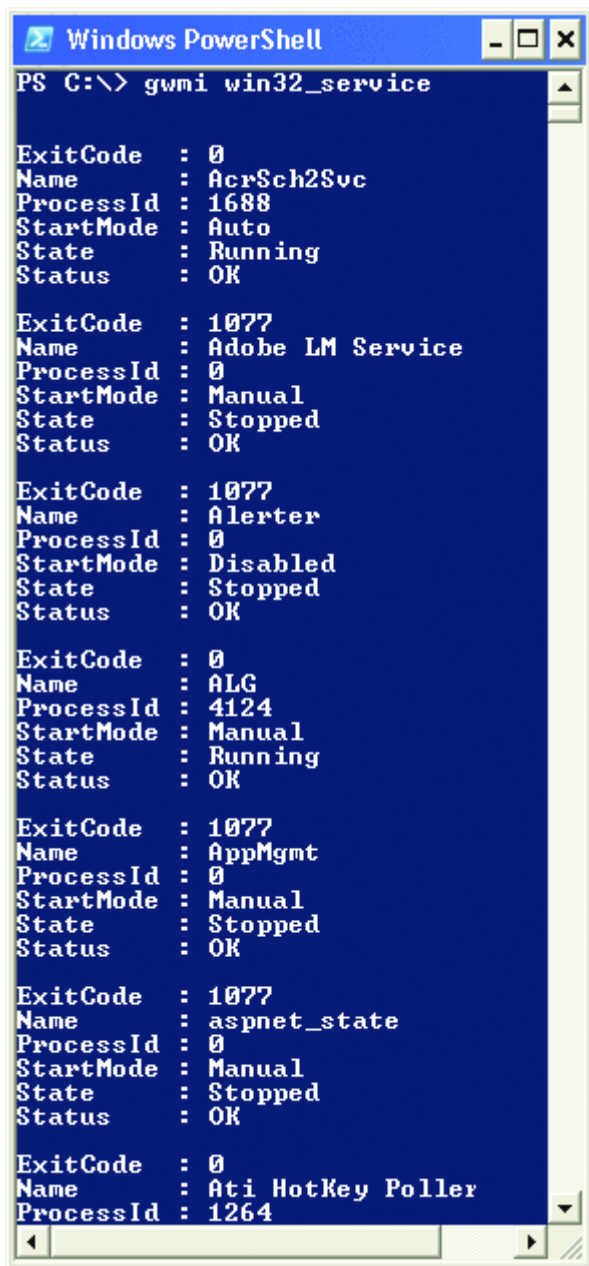
由于存在特别多的类, 因此 WMI 将它们组织到命名空间层次结构中。例如, 包含核心 Windows OS 类的命名空间称为 root\cimv2, 而 Microsoft IIS 6.0 将其类存储在 root\MicrosoftIISv2 中。方便的是, root\cimv2 命名空间是 WMI 的默认命名空间 (由 PowerShell 共享的设置), 这使得它可以更容易地与这些核心类一起使用。

“实例”是一种实际存在的类。例如, 如果您的计算机有两个逻辑磁盘, 则会有 Win32_LogicalDisk 类的两个实例。如果在您的计算机上运行 50 个服务, 在 WMI 上将会看到 Win32_Service 类的 50 个实例。使用 WMI 实际上就是请求 WMI 为您提供一个或多个实例, 然后, 要么检查这些实例的属性以发现您需要的管理信息, 要么执行这些实例的方法来进行管理更改 (例如启动或停止服务)。

WMI 使用客户端-服务器体系结构。Windows 2000 以后的每个 Windows 版本都内置了 WMI (后续版本扩展了可用类的数量), 这意味着为您同时提供了 WMI 客户端和 WMI 服务器软件。在使用 WMI 时, 您实际上是向在您关注的计算机上运行的 WMI 服务发送请求。该 WMI 服务检索您指定的 WMI 实例, 并将其返回给您以供使用。这就是 PowerShell 的作用, 它简化了请求实例、返回实例和使用实例的过程。

获取 WMI 对象

WMI 类实例大致是指一些对象, 因此在 PowerShell 中检索这些实例的方法是 Get-WMIObject cmdlet, 这很有意义。此 cmdlet 有一个方便的别名 gwmi, 我将在大多数示例中使用这个别名。对于最简单的形式而言, 您只需指定要检索的 WMI 类名称, 然后悠闲地等着查看结果即可 (请参见图 1)。在运行 gwmi win32_service 时, PowerShell 连接到本地计算机上的 WMI 服务 (因为我没有指定其他计算机) 并连接到 root\cimv2 命名空间 (因为我没有指定其他命名空间)。PowerShell 检索指定类的所有实例, 由于我没有指示对这些实例执行其他任何操作, 因此它将这些实例转换为文本表示形式。换句话说, PowerShell 获得这些对象, 并将其生成可以阅读的文本。



```
PS C:\> gwmi win32_service

ExitCode : 0
Name      : AcrSch2Svc
ProcessId : 1688
StartMode : Auto
State     : Running
Status    : OK

ExitCode : 1077
Name      : Adobe LM Service
ProcessId : 0
StartMode : Manual
State     : Stopped
Status    : OK

ExitCode : 1077
Name      : Alerter
ProcessId : 0
StartMode : Disabled
State     : Stopped
Status    : OK

ExitCode : 0
Name      : ALG
ProcessId : 4124
StartMode : Manual
State     : Running
Status    : OK

ExitCode : 1077
Name      : AppMgmt
ProcessId : 0
StartMode : Manual
State     : Stopped
Status    : OK

ExitCode : 1077
Name      : aspnet_state
ProcessId : 0
StartMode : Manual
State     : Stopped
Status    : OK

ExitCode : 0
Name      : Ati HotKey Poller
ProcessId : 1264
```

图 1 在运行 `gwmi win32_service` 时，PowerShell 以可读的文本格式返回指定类的所有实例

具体而言，PowerShell 通过读取和显示所选类属性的名称和值将 WMI 对象转换为文本。对于 Win32_Service 类，它选择一组属性（共六个）。

实际上，PowerShell 通过此方式将所有对象转换为文本。它选择显示的属性大部分在一组 .format.ps1xml 文件中定义，该文件位于 PowerShell 的安装文件夹中。这些格式定义文件经过 Microsoft 数字签名。尽管您可以提供自己的格式设置文件，但建议您不要更改这些文件。（在以后的专栏中，我将更详细地探讨这一主题。）

gwmi cmdlet 可以帮助您浏览计算机，查找有哪些可用类。例如，运行 `gwmi -namespace "root\cimv2" -list` 可以获得该命名空间中类的完整列表。但请记住，如果您在管理您的计算机，则仅与您计算机上的类相关；如果您在管理远程计算机，则将要查找该系统上可用的类。对于本例而言，您需要使用 gwmi 的 -computer 参数连接远程计算机。例如，`gwmi -namespace "root\cimv2" -list -computer ServerA` 将列出名为 ServerA 的远程计算机上 root\cimv2 命名空间中的所有类。

远程 WMI

在 PowerShell 的 1.0 版中，gwmi 可能是直接支持远程管理的唯一 cmdlet。这主要是由于远程控制构建在基础 WMI 体系结构中这一事实。而且，由于 PowerShell 只是使用现有体系结构，因此它受制于该体系结构的安全功能。例如：



```
C:\> gwmi -namespace "root\cimv2" -computer
mediaserver -list

Get-WmiObject : Access is denied. (Exception
from HRESULT: 0x80070005 (E_ACCESSDENIED))

At line:1 char:5

+ gwmi <<<< -namespace "root\cimv2" -computer
mediaserver -list

PS C:\>
```

在本例中，我尝试连接到名为 MediaServer 的远程计算机（我对该计算机没有访问权限）。作为管理员，我应该有权使用此计算机的 WMI 服务，但我的本地工作站凭据似乎没有足够的权限。例如，我可能在一个不同的、不可信的域中登录，或者可能使用较少权限的帐户登录。幸运的是，gwmi 支持 -credential 参数，我可以通过该参数为我的 WMI 连接指定另外一组用户凭据。下面您将看到一个非常类似的示例：



```
gwmi win32_service -credential mydomain\administrator -computer mediaserver
```

我的凭据（更具体地说就是我的用户名）以 DOMAIN\Username 格式提供。

注意，您没有可以输入密码的地方，PowerShell 将提示这一点。PowerShell 有意不提供在命令行输入密码的方法，因为这样会让您将密码硬编码到脚本文件，这无疑是一个安全风险。不过，您可以通过其他方法使用此 -credential 参数，即通过提前创建一种名为 PSCredential 的凭据对象。此命令为 Get-Credential cmdlet：



```
$cred = get-credential mydomain\administrator
```

我在运行此代码时，系统仍然提示我输入匹配的密码。不过，这次创建的凭据对象存储在 \$cred 变量中。如果我查看 \$cred 的内容，我将看到名称而不是密码：



```
PS C:\> $cred

UserName
-----
mydomain\administrator
```

然后我可以随心所欲地重用该凭据对象：



```
gwmi win32_service -credential $cred -computer mediaserver
```

通过预定义一个可重用的凭据对象，这简化了到远程计算机的重复 WMI 连接。不过这毫无意义，与 VBScript 的支持方式不同，Get-WmiObject cmdlet 目前不支持指定身份验证级别（也称模拟）。有关详细信息，请参见 msdn2.microsoft.com/aa389290.aspx。

自发现

我最喜欢的内容之一就是 PowerShell 不减少内容的难度。我向您介绍了 PowerShell 如何只选择我查询的 Win32_Service 类的一组属性。不过，该 Shell 仍可以访问所有属性，甚至可以告诉您它们是什么。要执行此操作，只需将该对象（或多个对象）传输到 Get-Member cmdlet（或其别名 gm），如图 2 所示。

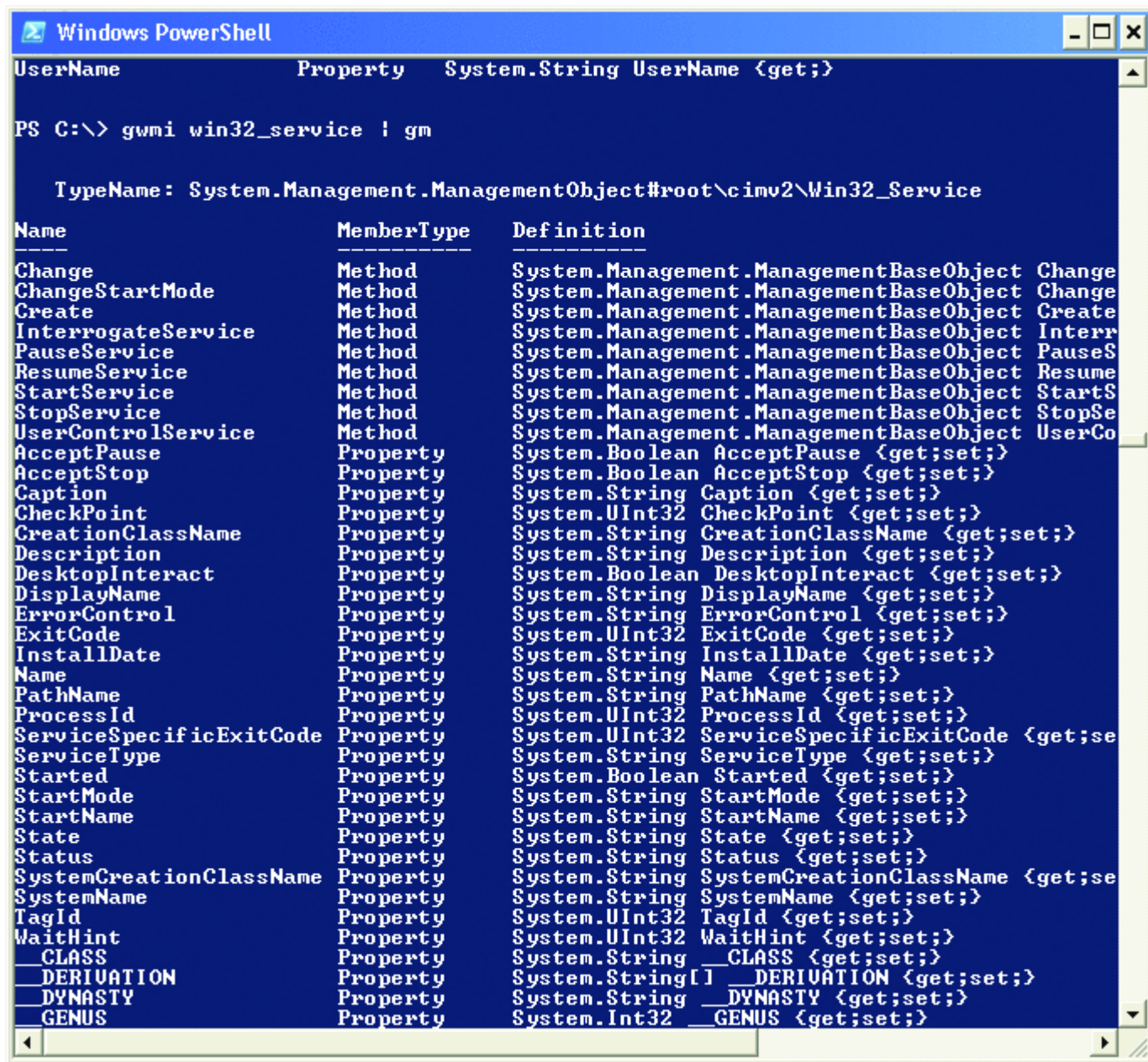


图 2 将一个对象传输到 Get-Member cmdlet 会告诉您可以访问哪些方法和属性

除了属性以外，该 Shell 还列出了可用的方法，这意味着我不必使用文档查看某个类的功能。我可以了解类本身提供了更改实例的配置、暂停服务和停止服务等任务的方法。

若要利用这些方法或显示其他属性，最容易的方法通常是将实例放入变量中：



```
$server = gwmi win32_operatingsystem

$server.reboot()
```

此示例将只检索 Win32_OperatingSystem 类的可用实例（该类在每台计算机上只有一个实例），并将其保存在 \$server 变量中。然后，我将使用 \$server 变量访问该实例的 Reboot 方法来重启计算机。使用该方法时一定要小心！

富查询语言

如果您在 VBScript 或其他技术下使用过 WMI，则可能习惯于使用 WQL (WMI 查询语言) 编写的查询检索 WMI 类实例。其类似于 SQL 的语法使得检索特定实例 (例如某个特定服务，而非某个给定类的所有实例) 更加容易。幸运的是，gwmi 还可让您指定一个查询，如：



```
gwmi -query "select * from win32_service where name='alerter' "
```

gwmi 的这一语法 (在正式发布 PowerShell 之前添加) 非常有用，并使得迁移为其他用途开发的复杂 WMI 查询非常容易。而且，与往常一样，PowerShell 将返回具有自己属性和方法的富对象，为您提供了访问 WMI 管理功能的完整权限。

WMI 的发展

将为 Windows 的未来版本继续开发 WMI，增加一些新类和功能。并继续将其添加到新的 Microsoft 产品中。尽管多数 Microsoft 产品尚未发布专门构建在 PowerShell 上的版本，但其连接 WMI 的能力是使 PowerShell 如今非常有用的最大好处之一。

PowerShell 重新考虑管道

之前关于 PowerShell 是多么与众不同、多么得新颖、多么得令人兴奋已经谈了很多，尽管它是基于已存在多年的命令行界面概念，而且主要用在基于 UNIX 和 Linux 的操作系统中。但是这个 PowerShell 与其先前版本共享的常见技术很容易忽视 PowerShell™ 真正的灵活性和独特性，及其卓越的 Windows® 环境适用性。

PowerShell 中最常谈到的一个功能就是管道，不过很遗憾，它也是最容易被误解的功能之一。那是因为它依据的是 19 世纪 70 年代早期定义的术语，曾经代表完全不同且性能较弱的功能。

管道的起源

曾创建过的第一批 UNIX 外壳中，其中一个就是 Thompson 外壳，它非常原始，只具有最基本的脚本编写语言元素功能，没有变量。这个外壳有意设计得很普通，执行程序是其唯一的真实用途。但是，它引入了一种改进当时其他外壳的关键概念，即管道。通过使用 < and > 符号，可以指示该外壳将输入和输出重定向到不同命令，以及重定向来自不同的命令的输入和输出。例如用户现在可以将命令输出重定向到某个文件。

这个语法在后来得到了扩展，可以使一个命令的输出也能被输送到另一个命令的输入，从而使一长串连续的命令可以链接在一起，以完成更多复杂的任务。在外壳的版本 4 之前，都采用竖线字符 "|" 来进行输送，因此称为管道字符。甚至 MS-DOS® 最早的版本就已通过这些字符实施了基本管道，允许用户将 type 命令的输出输送到更多命令的输入，从而创建针对长文本文件的一次显示一页的方式。

尽管在 1975 年发布 UNIX 版本 6 之前，Thompson 外壳被广泛认为存在不足，但对外壳开发人员和用户来讲，管道的概念还是很深入人心，并且发展成了许多至今还在使用的技术。

管道文本

几乎所有外壳的限制都在于它们内在的基于文本的本质。在基于 UNIX 的操作系统中，这实际上不是限制，而是对操作系统本身如何运行的反映。UNIX 中几乎任何资源都可以以某种类型的文件表示，这意味着可以将文本从一个命令输送到另一个可以提供大量功能和灵活性的命令。

然而一提到管理信息，文本肯定是具有限制性的。例如，如果我要向您提供一个运行在 Windows 计算机上的服务列表，您肯定能明白这个意思。也许我最好是在第一栏输入服务名称，第二栏输入它的启动模式。您强大的大脑会进行透明而即时的分析或转换，文本会显示为您可以理解的有意义的信息。可是计算机没那么聪明：为了让计算机对这个列表执行一些有意义的操作，您必须告诉计算机第一栏由字符 1 到 20 组成，第二栏也许由字符 22 到 40 组成，等等。

多年来，这种文本文件分析一直是管理员必须将多个命令链接在一起的唯一方法。实际上，VBScript 和 Perl 这些脚本编写语言在字符串操作方面占优势，主要是因为它们需要能够接收一个程序或命令的文本输出，然后将该输出分析转换为某种可用于某些后续任务的有用数据。例如，我曾经写过这样一个 VBScript 作业，即接收 Dir 命令的文本输出，分析文件名称和日期的输出，然后将以

前未使用的文件移到存档位置。字符串分析非常棘手，因为例外（输入数据中的各种变化）几乎一直发生，这要求您重做脚本中的逻辑以处理所有可能发生的改变。

作为 Windows 环境中管理脚本或自动化的形式，字符串分析并不是很有用。这是因为 Windows 本身不是以容易访问的文本格式存储很多信息。相反，它使用数据中心存储（例如 Active Directory®、Windows 注册表和证书存储），使得脚本专家们必须先使用一个工具来生成某些形式的文本输出，然后才是分析这个文本并使用它执行一些操作的脚本。

对象更简单

Windows 软件开发人员总是会想办法让一切变得更简单。最初，Microsoft 特意开发了 COM，想以更易于使用的方式来表示 Windows 复杂的内部工作机制。现在，Microsoft® .NET Framework 仍执行这个相同的任务——它以标准化的方式表示软件的内部工作机制。

通常，COM 和 .NET 将项目作为对象公开。（软件开发人员可能会反对这种简化，但是出于讨论目的，简单的术语就足够了。）这些对象都具有很多不同类型的成员。对于我们来说，对象的属性和方法是我们最感兴趣的地方。属性基本上在某些方面可以说明一个对象，或者它会修改对象或其行为。例如，服务对象可能会有一个包含服务名称的属性和另一个包含服务启动模式的属性。方法会导致对象执行某个操作。例如，服务对象可能会有名为 Stop、Start、Pause 和 Resume 的方法，表示可以和服务一起执行的不同操作。

从编程或脚本编写的角度来看，对象的成员是指使用点阵符号。对象会经常被分配给变量，从而为您提供物理操作该对象的方法。例如，如果我将一个服务分配给变量 \$service，我可以使用语法 \$service.Stop 停止该服务。或者我可以通过显示 \$service.Name 来检索该服务的显示名称。

管道中的对象

因为 Windows 是一个庞大而复杂的操作系统，而且它不会以文本格式存储它的管理数据，所以较早的外壳技术根本不适合它。例如，假设我有一个名为 SvcList.exe 的命令行工具，它可以产生服务的格式化列表及其启动模式。在 Windows 命令行外壳（一种外壳，深深扎根于具有数十年历史的 MS-DOS 外壳）中，我可能会运行以下程序：



```
SvcList.exe | MyScript.vbs
```

此语句检索服务列表并将该列表输送到 VBScript 文件。我必须编写 VBScript 文件来分析这个格式化列表，然后执行所有我想执行的操作——也许会输出任何启动模式为“禁用”的服务。这项任务可能比较耗时。最后，问题是该 SvcList.exe 有一个独特的输出，它不共享其他命令可以轻易用来分析利用其输出的通用格式。

然而对象可以提供这个通用格式，那就是 PowerShell 管道与所有对象（而不仅是文本）一起工作的原因。当您运行 Get-WMIObject 之类的 cmdlet 时，用程序员术语来说，会产生对象组或对象集合。每个对象都包含允许您操作它的所有属性和方法。如果将对象输送到 Where-Object cmdlet，就可以筛选它们，从而只剩下我想要显示的对象。Where-Object 不需要分析任何文本，因为它没有检索任何文本，它在检索对象。例如：



```
Get-WMIObject Win32_Service | Where-Object {$_.StartMode -eq "Disabled" }
```

或者，如果您更喜欢通过别名提供的较短语法：



```
gwmw Win32_Service | where {$_.StartMode -eq "Disabled" }
```

有趣的是 PowerShell 始终会将对象沿着管道传递。直到管道的末端才停止（当没有其他地方可以传递对象的时候），外壳会使用其内置格式规则生成对象的文本表示形式。例如，请看下面的情况：



```
Gwmw Win32_Service | where {$_.StartName -eq "LocalSystem" } | select
```

```
Name,StartMode
```


这组三个 cmdlet 可以检索我本地计算机中的所有服务，筛选掉不使用 LocalSystem 帐号登录的所有服务，然后将其他服务传递给 Select-Object cmdlet，这样仅输出两个属性：Name 和 StartMode——它们是我之前要求要选择两个属性。结果是一份以 LocalSystem（也许是出于安全审核目的）身份登录的服务的简单报告。

由于所有的 cmdlet 都共享一种通用数据格式——对象，所以它们可以与另一个不进行复杂字符串分析的 cmdlet 共享数据。同时由于 PowerShell 本来就可以创建对象的文本表示形式，管道的末端就肯定是我，一个人，可以读取的文本输出。图 1 显示了产生的输出示例。

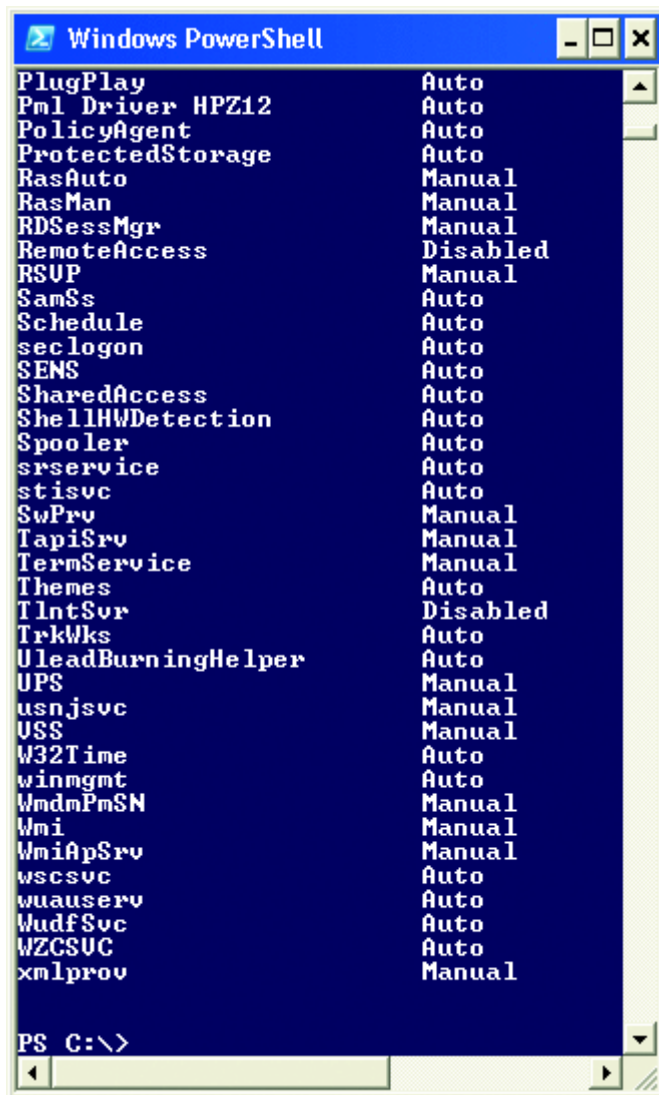


图 1 和对象一起传递的一系列输送的 cmdlet 所产生的文本输出

管道的亮点

在 PowerShell 中输送的原因非常令人诧异，那就是 PowerShell 中的每个项目都是对象，包括您可以使用的所有属性和方法。从技术角度上来说，甚至文本文件也是字符串对象的集合，文件中的每一行文本都可充当一种独立的独特字符串对象。例如，创建名为 C:\Computers.txt 的文本文件（使用“记事本”）。使用文本填充该文件，然后在 PowerShell 中运行下列命令：



```
Get-Content C:\Computers.txt | Select-Object Length | Format-List
```

如果您不太喜欢要进行很多键入操作，也可以使用别名：



```
gc C:\Computers.txt | select Length | fl
```

该代码会提供一个列表,指定每个文本行的长度(以字符为单位)。Get-Content 会检索文件中的字符串对象,Select-Object 则会获取每个对象的 Length 属性,而 Format-List 则会为您创建好用的、可以读取的文本输出。虽然这可能不是一个可以使用的实用管理工具,但是它阐明,即使是与文本行一样简单的元素也是 PowerShell 中的对象。

由于能够将对象从 cmdlet 输送到 cmdlet,或者从 cmdlet 输送到脚本,因此可以创建非常强大的“一行式命令”。它们是简单的 cmdlets 字符串,附加在一个较长的管道中,该管道进一步完善了对象集,可以提供您真正需要的一切。几乎不进行任何脚本编写或编程,PowerShell cmdlet(在相应管道中排成一队)也可以实现显著的效果。

支持未来一代

未来的 Microsoft 服务器产品仍会建立在扩展了此功能的 PowerShell 之上。例如,实施新的 Exchange Server 2007 机器时,您可以使用 PowerShell 来检索所有的邮箱,筛选掉所有不在新邮件服务器所在办公室的邮箱,然后将这些邮箱移到新的服务器上——这一切操作都不需要编写脚本,只需单个文本行即可实现。Exchange Server 2007 团队已发布强大的一行式命令的冗长列表。它真正阐述了管道的功能及其能够完成的管理任务。

使用 PowerShell 的技巧是要了解,虽然这个新工具建立在长期存在的原则和 UNIX 世界的原理之上,但它是适合 Windows 管理的唯一工具。不要让术语的通用性愚弄了您,让您误以为 PowerShell 就是 Windows 的 UNIX 外壳翻版。PowerShell 包含全新的概念,那就是利用 Windows 平台,将它与 Windows 执行任务的方式紧密结合起来。

PowerShell 筛选的力量

在上个月的专栏中,我论述了 Windows PowerShell 管道的强大功能和灵活性,它可以使您将一个数据集(更准确地说,是一组对象流)从一个 cmdlet 传递到另一个 cmdlet,进一步优化该数据集直到它完全符合您的需要。在我的论述中,我提到您自己的脚本(不只是 cmdlet)也可以利用管道。本月我要详细论述该主题。

我在 Windows PowerShell™ 中最常做的事情之一是编写通常通过 Windows® Management Instrumentation (WMI) 对多台远程计算机执行操作的脚本。对于任何处理远程计算机的任务,在脚本运行时,这些计算机中的一台或多台总是有可能不可用。因此,我需要我的脚本能够解决这个问题。

当然,我可以许多方法使脚本具备处理 WMI 连接超时的能力,但是我特别不喜欢该方法,因为超时期限本身太长——默认情况下大约为 30 秒。这样,如果我的脚本必须等待多次超时发生,它的运行将更加缓慢。相反,我要脚本执行一次快速检查,以查看在尝试连接到给定计算机前该计算机是否实际上处于联机状态。

Windows PowerShell 范例

在其他脚本编写语言(如 VBScript)中,我通常一次处理一台计算机。也就是说,我将检索一个计算机名称——可能从存储在文本文件中的名称列表中检索——并 ping 该系统以查看其是否可用。如果可用,我将建立 WMI 连接并执行我需要完成的所有其他工作。这是一种常见的脚本式方法。实际上,我可能要在一个循环中编写所有我的代码,然后对我需要连接的每台计算机重复执行一次该循环。

然而,由于 Windows PowerShell 基于对象并且能够直接处理对象组或集合,它更适用于批处理操作。Windows PowerShell 中的范例不是要处理单一的对象或数据段,而是要处理整个组,一点点优化该组,直到完成您打算做的任何事情。例如,我想立刻读取整个名称集,而不是一次检索列表中的一个计算机名称。我要编写单一例程,该例程接受名称集,ping 这些计算机,然后输出可以连接的计算机名称,而不是在循环中 ping 每台计算机。我的过程中的下一步是建立到其余名称(即可通过 ping 访问的名称)的 WMI 连接。

Windows PowerShell 对多项任务使用这种方法。例如,要获得正在运行的服务的列表,我可以使用与以下内容:



```
Get-Service | Where-Object { $_.Status -eq "Running" }
```

图 1 显示了我的计算机上显示的输出情况。我使用 Get-Service 检索所有服务,通过管道将其输送到 Where-Object,然后筛选出所有未运行的服务;而不是一次检查一项服务。这差不多就是我用脚本要做的事情:获得计算机名称列表,筛选出所有不可 ping 的名称,然后将可 ping 的计算机列表传送到下一步。

```

PS C:\> gsv | where { $_.Status -eq "Running" }

```

Status	Name	DisplayName
Running	AcrSch2Svc	Acronis Scheduler2 Service
Running	ALG	Application Layer Gateway Service
Running	Ati HotKey Poller	Ati HotKey Poller
Running	AudioSrv	Windows Audio
Running	BITS	Background Intelligent Transfer Ser...
Running	Browser	Computer Browser
Running	CryptSvc	Cryptographic Services
Running	DcomLaunch	DCOM Server Process Launcher
Running	Dhcp	DHCP Client
Running	dmserver	Logical Disk Manager
Running	Dnscache	DNS Client
Running	ERSvc	Error Reporting Service
Running	Eventlog	Event Log
Running	EventSystem	COM+ Event System
Running	FastUserSwitchi...	Fast User Switching Compatibility
Running	FLEXnet Licensi...	FLEXnet Licensing Service
Running	helpsvc	Help and Support
Running	HidServ	HID Input Service
Running	HTTPFilter	HTTP SSL
Running	iPod Service	iPod Service
Running	lanmanserver	Server
Running	lanmanworkstation	Workstation
Running	LightScribeService	LightScribeService Direct Disc Labe...
Running	LmHosts	TCP/IP NetBIOS Helper
Running	MDM	Machine Debug Manager
Running	Netman	Network Connections
Running	Nla	Network Location Awareness (NLA)
Running	PlugPlay	Plug and Play
Running	Pml Driver HPZ12	Pml Driver HPZ12
Running	PolicyAgent	IPSEC Services
Running	ProtectedStorage	Protected Storage
Running	RasMan	Remote Access Connection Manager
Running	RemoteRegistry	Remote Registry
Running	RpcSs	Remote Procedure Call (RPC)
Running	SamSs	Security Accounts Manager

图 1 获得可 ping 的计算机列表

筛选功能

尽管我可以编写自己的 cmdlet 来实现此功能，但是我不想这样做。编写 cmdlet 需要 Visual Basic® 或 C# 以及大量的 Microsoft® .NET Framework 开发专业知识。更重要的是，在该任务中它要求更多的参与，而我不想投入这么多。幸运的是，Windows PowerShell 允许我编写一种叫做筛选的特殊功能，它可以在管道内完美地发挥作用。筛选功能的基本轮廓如下所示：



```

function <name> {

    BEGIN {

        #<code>

    }

    PROCESS {

        #<code>

```

```

}

END {

    #<code>

}

}

```

正如您所看到的，该功能包含三个独立的脚本块，分别命名为 BEGIN、PROCESS 和 END。筛选功能 — 即旨在在管道内工作以筛选对象的功能 — 可以拥有这三个脚本块的任何组合，具体取决于您要做什么。它们的工作方式如下：

在首次调用您的功能时，BEGIN 块执行一次。您可以用它来执行设置工作，如果需要的话。

PROCESS 块对传送到功能的每个管道对象执行一次。\$_ 变量代表当前的管道输入对象。筛选功能中需要 PROCESS 块。

在处理完所有管道对象后，END 块执行一次。这可用于任何收尾工作（如果需要）。

在我的示例中，我要生成一个筛选功能，该功能接受名称集合作为输入对象，然后尝试 ping 每个对象。可以成功地 ping 的每个对象将输出到管道；不可以 ping 的系统将被删除。由于 ping 功能不需要任何特殊的设置或收尾工作，我将仅使用 PROCESS 脚本块。图 2 中的代码提供了完整的脚本。

Figure 2 Ping-Address 和 Restart-Computer



```

1 function Ping-Address {
2     PROCESS {
3         $ping = $false
4         $results = Get-WmiObject -query `
5             "SELECT * FROM Win32_PingStatus WHERE Address = '$_'"
6         foreach ($result in $results) {
7             if ($results.StatusCode -eq 0) {
8                 $ping = $true
9             }
10        }
11        if ($ping -eq $true) {
12            Write-Output $_
13        }

```

```

14     }

15 }

16

17 function Restart-Computer {

18     PROCESS {

19         $computer = Get-WmiObject Win32_OperatingSystem -computer $_

20         $computer.Reboot()

21     }

22 }

23

24 Get-Content c:\computers.txt | Ping-Address | Restart-Computer

```

请注意，我已定义了两个功能：Ping-Address 和 Restart-Computer。在 Windows PowerShell 中，调用功能之前必须对其进行定义。因此，第一个可执行的脚本行是行 24，它使用 Get-Content cmdlet 来从文件中获得计算机名称列表（每行显示一个计算机名称）。该列表（从技术角度上说，是字符串对象集合）将通过管道传递到 Ping-Address 功能，它会筛选出不能 ping 的任何计算机。结果将通过管道传递到 Restart-Computer，该功能使用 WMI 来远程重新启动可以 ping 的每台计算机。

Ping-Address 功能执行 PROCESS 脚本块，这意味着功能需要输入对象集合。PROCESS 脚本块自动处理该输入——我不必定义任何输入参数来包含该输入。我通过将变量 \$ping 设置为 \$false 来启动行 3 上的过程，\$false 是代表布尔值 False 的内置 Windows PowerShell 变量。

然后，我使用本地 Win32_PingStatus WMI 类来 ping 指定的计算机。请注意，行 5 上代表当前管道对象的 \$_ 变量将嵌入 WMI 查询字符串内。当字符串包含在双引号中时，Windows PowerShell 总是尝试用内容替换变量（如 \$_），因此您不必浪费时间考虑字符串连接。我正在行 5 上使用该功能。

行 6 是检查 ping 结果的循环。如果这些结果中的任意一个成功返回（也就是说，StatusCode 为零），我会设置 \$ping 等于 \$true 来指示成功。在行 11 上，我检查 \$ping 是否已设置为 \$true。如果是，我会将原始输入对象输出到默认输出流。Windows PowerShell 自动管理默认输出流。如果该功能位于管道的末端，则输出流将转换为文本表示形式。如果管道中有更多的命令，则输出流的对象——在本例中，是包含计算机名称的字符串对象——将传递到下一个管道命令。

Restart-Computer 功能稍微简单一些，但是它也使用 PROCESS 块以便轻松地加入到管道中。在行 19 上，该功能连接到指定的计算机并检索其 Win32_OperatingSystem WMI 类。行 20 执行类的 Reboot 方法来重新启动远程计算机。

同样，行 24 是实际执行所有操作的位置。当然，运行该脚本时您应该非常小心——它旨在重新启动在 c:\computers.txt 中指定的每台计算机，如果您不特别注意该文本文件中的名称，肯定会导致灾难性的后果！

后续步骤

该脚本并非完全无懈可击。我还应该意识到 WMI 错误可能与基本连接不相关，如防火墙阻止远程计算机上的必要端口或 WMI 安全错误。我可以通过实施陷阱处理程序处理这些问题——这听起来像未来专栏的理想主题。

此外，该脚本正在执行一个相当严重的操作——重新启动远程计算机。任何尝试这种潜在的危险操作的脚本都应该实施两个常见的 Windows PowerShell 参数，-Confirm 和 -WhatIf。解释如何进行此操作在这里一言难尽，但可以在未来专栏中进行专题论述。同时，请访问 Windows PowerShell 团队博客；设计师 Jeffrey Snover 会介绍该主题。

即使不了解这些特性，您也可以很好地理解筛选功能的作用。我将在随后的专栏中完善该技术，向您展示这些功能如何很好地模块化可重复使用的代码以及从总体上增强脚本的功能。

PowerShell 无脚本运行

每次准备和大家探讨 PowerShell 时(无论在会议上、公共 PowerShell 新闻组还是在我的网站)，总会有管理员告诉我他们“打算等等”再学习 PowerShell。为什么呢？他们给出的最常见的理由是

“现在没有时间学习如何编写脚本”。

这种情绪在管理员中很普遍。我们管理员群体的工作很忙，所以实在没有太多空闲时间学习新的技术。但是，我们必须不断努力向前，学习 Windows Vista®、Windows Server® 2008 和 Exchange Server 2007 等新软件，这样才能站在对工作至关重要的技术的前沿。PowerShell™ 是这些技术中举足轻重的一部分。

我常听到的管理员拖延学习在 PowerShell 中编写脚本的另一个理由是，他们“担心成为程序员”（一位管理员同行曾这样说）。我可以在这方面帮上点忙。如果您发现不用编写一丁点代码就能使用 PowerShell 完成一些真正令人诧异的任务，可能会感到吃惊。

准备好进行管道传输

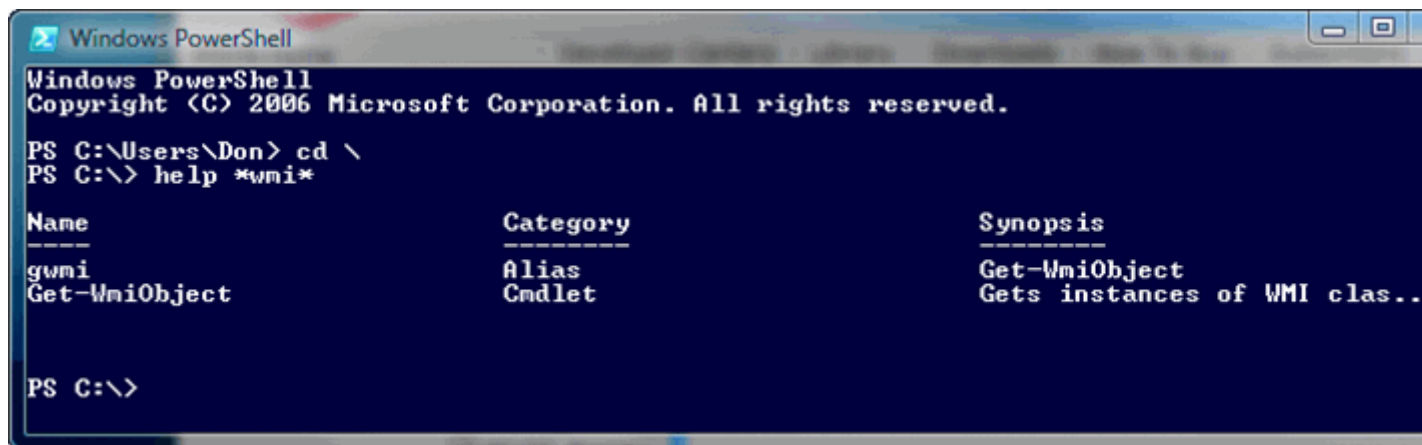
我以前写过有关 PowerShell 所使用的极其灵活、强大且面向对象的管道的文章。（虽然“面向对象”这个词可能使您怀疑我要针对程序员领域进行讲述，但是请继续往下看。）我喜欢管道的一个原因就是交互使用管道的方式。它可以提供即时结果并能使您轻松精简这些结果，以便使所获结果确实是您所需的内容。例如，假设我希望获得多台远程计算机的可用磁盘空间清单，并且现在已将那些计算机的名称列在名为 C:\Computers.txt 的文本文件中。这仅是一个纯文本文件，每行包含一个计算机名称，不像数据库那样复杂。

首先，我会查看是否能从本地计算机上检索到此信息。毕竟，如果可以在一台计算机上搞清楚，那么在多台计算机上执行此任务就应当简单多了。还要记住，此处不会涉及到脚本编写！我将在外壳程序中以交互方式执行所有操作，按 Enter 键即可获得结果。

以下是有关此类任务的提示。每次要清点远程计算机上的管理信息时，可能都要使用 Windows® 管理规范 (WMI)。根据此示例的目的，假设现在我只知道需要使用 WMI。所以我跳到 Live Search，然后输入“windows wmi 可用磁盘空间”作为我的搜索项。将显示其摘录中列出“Win32_LogicalDisk”短语的若干条结果。这看起来像 WMI 类，而且可能正是我要的结果。甚至不必费心单击其中任一搜索结果。改为输入“Win32_LogicalDisk”作为新的搜索项，第一条结果是 MSDN® 站点上的 Win32_LogicalDisk 文档页面 (msdn2.microsoft.com/aa394173.aspx)。所以，我跳到这一页，发现此类的一个属性是 FreeSpace，看起来大有希望。

查找 Cmdlet

下一步，需要查找将为我检索 WMI 类属性的 cmdlet。PowerShell 引人注目的一点是它具有极妙的内置自我发现功能，也就是说，外壳可帮助您查找它自己的功能。因此我输入 Help *wmi* 来查找外壳程序对使用 WMI 的了解。如图 1 所示，该结果为别名 Gwmi 及其指向的 cmdlet（即 Get-WmiObject）。基本上，可以通过两种方法访问同一个功能，这意味着我决定使用哪种方法并不是很困难。既然这两种方法都可以达到我的目的，而 Gwmi 字符较少，易于输入，所以我使用这种方法：



```
Windows PowerShell
Copyright (C) 2006 Microsoft Corporation. All rights reserved.

PS C:\Users\Don> cd \
PS C:\> help *wmi*

Name                           Category
----
gwmi                           Alias
Get-WmiObject                  Cmdlet

Synopsis
-----
Get-WmiObject
Gets instances of WMI clas..

PS C:\>
```

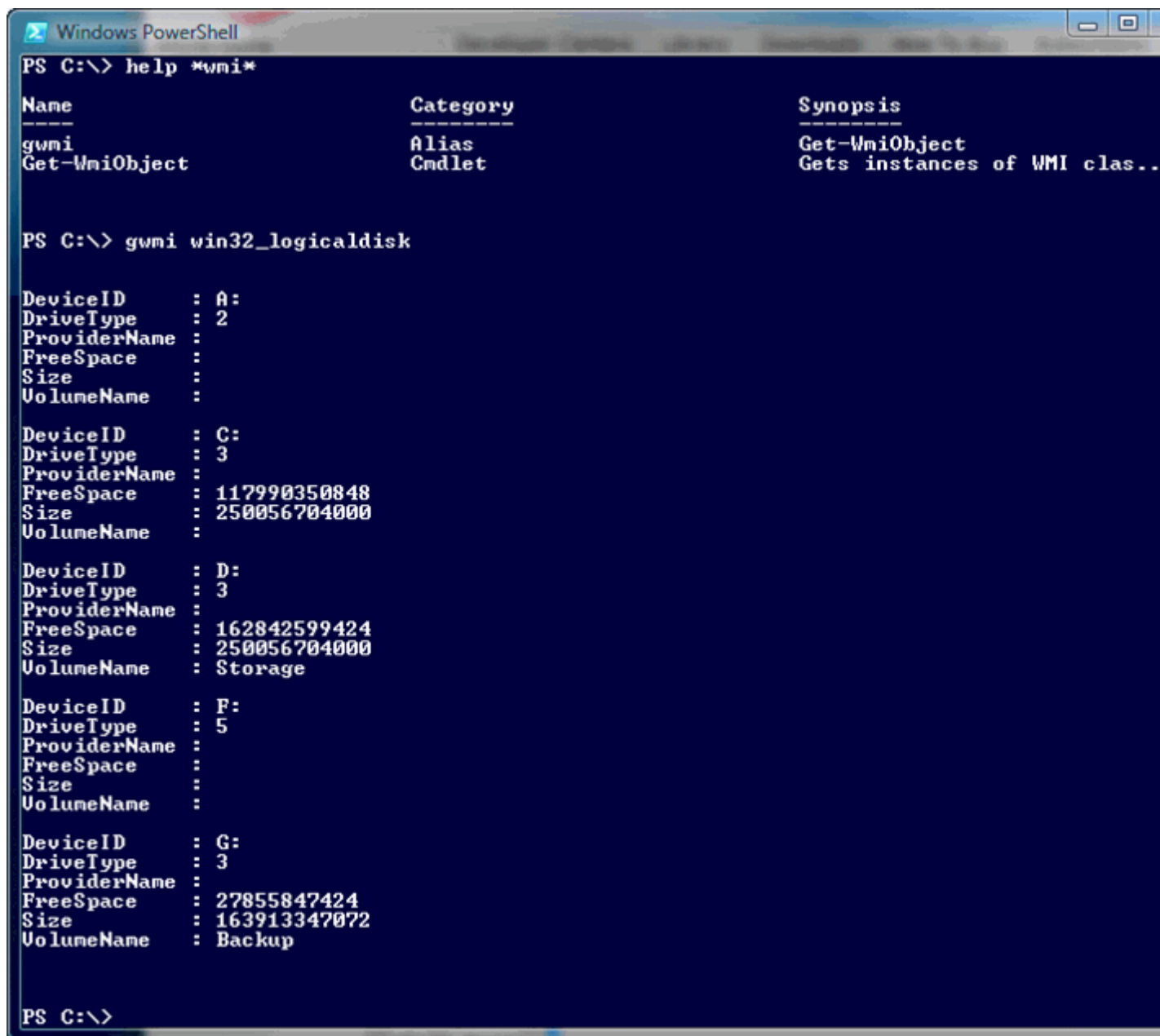
图 1 外壳程序对使用 WMI 的了解



```
PS C:\> gwmi win32_logicaldisk
```

请注意 PowerShell 不区分大小写，因为它知道管理员没有时间注意诸如按 Shift 键之类的繁琐细节。

如图 2 所示，此命令的结果包含我的全部 5 个本地驱动器及其看起来以字节形式列出的 FreeSpace 属性。同时还列出了每个驱动器的 DriveType 属性，计算机上显示值 2、3 和 5。



```
PS C:\> help *wmi*

Name
----
gwmi
Get-WmiObject

Category
-----
Alias
Cmdlet

Synopsis
-----
Get-WmiObject
Gets instances of WMI clas..

PS C:\> gwmi win32_logicaldisk

DeviceID      : A:
DriveType     : 2
ProviderName  :
FreeSpace     :
Size         :
VolumeName    :

DeviceID      : C:
DriveType     : 3
ProviderName  :
FreeSpace     : 117990350848
Size         : 250056704000
VolumeName    :

DeviceID      : D:
DriveType     : 3
ProviderName  :
FreeSpace     : 162842599424
Size         : 250056704000
VolumeName    : Storage

DeviceID      : F:
DriveType     : 5
ProviderName  :
FreeSpace     :
Size         :
VolumeName    :

DeviceID      : G:
DriveType     : 3
ProviderName  :
FreeSpace     : 27855847424
Size         : 163913347072
VolumeName    : Backup

PS C:\>
```

图 2 所有的本地驱动器及其 FreeSpace 属性

本月 cmdlet

您也许用过 Get-Content 读取文本文件的内容。它将文件的每一行作为一个 [string] 对象，并将这些对象传入管道中供其他 cmdlet 使用。但是，Get-Content 拥有许多选项，使其更具灵活性：例如，-readCount 参数允许指定可一次向管道传入 [string] 对象的数量（默认情况下为全部传送），而 -totalCount 参数控制从文件读取的总行数。对于实在非常大的文件，为了提高性能，您可能不希望一次处理整个文件。这两个参数对处理这种大文件非常有用。

下面是另一个用起来很方便的参数：-encoding 参数，它用于正确读取各种文件编码类型，包括 Unicode、ASCII、UTF7、UTF8 和许多其他编码类型。若要查看受支持的编码类型的完整列表，请运行 help gc（gc 是 Get-Content 的别名）。

精简数据

这里有两个问题。首先，我并不关心其他所有属性，仅需要 FreeSpace。其次，我不想了解光驱、可移动驱动器或网络驱动器的可用空间，我只想了解本地硬盘。也许，DriveType 属性有助于对其进行区分。我返回到 Web 浏览器，发现文档中包含一个介绍不

同 DriveType 值所代表的含义的表。我仅需要 DriveType 值为 3 的磁盘,值 3 表示该驱动器为本地磁盘。因为以前没有进行过此类操作,所以我决定看看 Gwmi 命令是否可以为我执行某些筛选操作。运行 Help Gwmi,即列出了有关如何使用该命令的详细信息,还包含一些示例。我找到一个看起来很有用的参数 -filter,所以决定试试运行以下命令:



```
gwmi win32_logicaldisk -filter "drivetype = 3"
```

成功了!现在,我的驱动器列表仅包含本地固定磁盘。这样,第二个问题就解决了,现在,我需要解决第一个问题——除去我不关心的所有属性。

运行 Get-Command 列出所有的 PowerShell cmdlet,最终我发现了 Select-Object。它的描述为使用它将“选择某个对象或对象集的指定属性”。所以我尝试运行以下命令:



```
gwmi win32_logicaldisk -filter "drivetype = 3" | select freespace
```

通过将 Gwmi 的结果传送到 Select (Select-Object 的别名),我可以获得需要的属性。天哪。这些结果并不好,因为我仅得到一个数字列表,而不知道可用空间列表与驱动器的对应关系。所以,我回去查找文档,看到 DeviceID 属性包含驱动器号。我快速修改命令,再次尝试:



```
gwmi win32_logicaldisk -filter "drivetype = 3" | select deviceid,freespace
```

好极了!因为仅列出了两个属性,所以外壳程序可将信息置于一个规范的表中。(下个月,我将介绍 PowerShell 何时选择使用列表和表。)

计算

我已经检索了有关可用空间的信息,但是以字节为单位,并不如以兆字节或千兆字节为单位有用。或许,我并不需要 FreeSpace 属性本身,而是需要从 FreeSpace 属性计算得出的值。ForEach-Object cmdlet (或其诸多别名之一,如 %) 可帮助解决此问题。此 cmdlet 允许使用特殊的变量 \$_ 指代当前的逻辑磁盘,并且允许单独访问每个磁盘属性并进行计算。您可以看到我已修改的命令:



```
gwmi win32_logicaldisk -filter "drivetype = 3" | % { $_.deviceid; $_.freespace/1GB }
```

这里我所做的是删除 Select 并使用 ForEach-Object 的别名 (%) 代替它。此 cmdlet 仅需要我告诉它如何处理得到的每个 Win32_LogicalDisk,我已告诉它获取 DeviceID 属性并以 GB 为单位划分 FreeSpace 属性 (PowerShell “理解” KB、MB 和 GB 的意思),因此我的输出将以 GB 为单位。

许多计算机

现在我已在一台计算机上取得成功,因此就可在多个系统上进行此操作了。我了解如何获取文本文件的内容。这可通过采用与过去 MS-DOS® 时代相同的方法完成——使用 Type 命令,实际上,此命令在 PowerShell 中是 Get-Content 的别名:



```
Type c:\computers.txt
```

我现在要做的是使用每台计算机并对每一台计算机执行我成功创建的 WMI 命令。此短语 “for each one” (对每一台) 应会使您想起一个 cmdlet——具体来说,就是 ForEach-Object。像那些家居设计电视节目中说的那样,让我们为您揭晓吧,请看最后一个命令:



```
type c:\computers.txt | % { $_;
```

```
gwmi -computername $_ win32_logicaldisk -filter "drivetype=3" | %{ $_.deviceid;  
$_freespace/1GB} }
```

很可怕，对吗？由于我只使用别名而不使用 cmdlet 名称，造成了难以读取的情况。但是，如果一次只浏览一点，解读起来就相当容易了：

首先“键入”文本文件的内容。

将这些内容传送到 ForEach-Object。

ForEach-Object 使用 \$_ 变量（这是当前的计算机名称）输出当前项目。

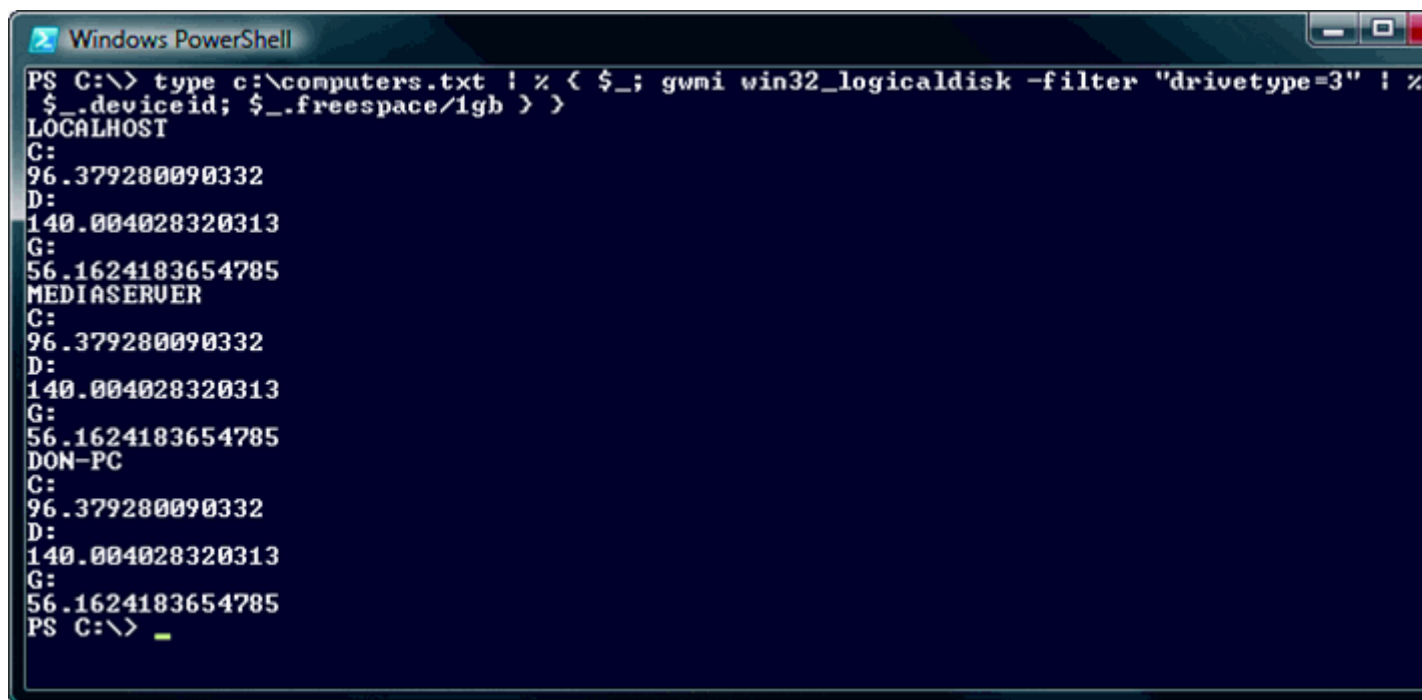
然后，ForEach-Object 执行我的 WMI 命令，它有另一个 ForEach-Object 调用。

将别名名称扩展为 cmdlet 名称可能很有帮助。所以下面的命令与之前的相同，但这次我拼写出了完整的 cmdlet 并将命令分行，因而可轻松检查每一节：



```
Get-Content C:\Computers.txt |  
  
ForEach-Object {  
  
    $_; Get-WMIObject -computername $_  
  
    Win32_LogicalDisk -filter "DriveType=3" |  
  
    ForEach-Object {  
  
        $_.DeviceID; $_.FreeSpace/1GB  
  
    }  
  
}
```

该结果并不吸引人，但能正常发挥作用，如图 3 所示。



```
PS C:\> type c:\computers.txt | % { $_; gwmi win32_logicaldisk -filter "drivetype=3" | % { $_.deviceid; $_.freespace/1gb } }
LOCALHOST
C:
96.379280090332
D:
140.004028320313
G:
56.1624183654785
MEDIASERVER
C:
96.379280090332
D:
140.004028320313
G:
56.1624183654785
DON-PC
C:
96.379280090332
D:
140.004028320313
G:
56.1624183654785
PS C:\> _
```

图 3 最终结果

无需脚本

此预排旨在说明在不用编写脚本的情况下，也可以使用 PowerShell。而且，仅使用一些诸如此类的示例，再稍微研究一下 PowerShell 本身，您实际上可以找到完成打算执行的任务所需的一些信息。

关键是现在 PowerShell 已不足为奇。何不现在就开始学习使用它呢？您甚至可能发现自己会想像没有 PowerShell 时是怎么过来的，当然您不应该因脚本编写这个词而因噎废食。

PowerShell 编写正则表达式

192.168.4.5。\\Server57\Share。johnd@contoso.com。您马上会认出这三个条目是 IP 地址、通用命名约定 (UNC) 路径以及电子邮件地址。您的大脑识别出了它们的格式。四组数字、反斜线符号、@ 符号以及其他提示都表明了这些字符串代表的数据类型。不必深思，您可以快速识别出 192.168 本身是一个无效的 IP 地址、7\Server2\Share 是无效的 UNC，joe@contoso 也是无效的电子邮件地址。遗憾的是，计算机必须经过一番努力才能“了解”诸如以上的复杂格式。正则表达式即应运而生。正则表达式是使用特殊正则表达式语言编写而成的字符串，可帮助计算机识别格式特殊的字符串，如 IP 地址、UNC 或电子邮件地址。使用正确编写的正则表达式，Windows PowerShell™ 脚本可接受有效数据或者拒绝与指定格式不符的无效数据。

进行简单匹配

Windows PowerShell -match 运算符将字符串与正则表达式或 Regex 进行比较，然后根据该字符串是否与 Regex 匹配返回 True 或者 False。简单的 regex 甚至不需要包含任何特殊语法，有文字字符即可。例如：



```
"Microsoft" -match "soft"
```

```
"Software" -match "soft"
```

```
"Computers" -match "soft"
```

在 Windows PowerShell 中运行时，前两个表达式返回 True，第三个返回 False。在每个表达式中，字符串后均跟 -match 运算符，然后是 regex。默认情况下，regex 将在字符串中浮动查找匹配项。在 Software 和 Microsoft 中均可找到“soft”字符，但位置不同。另请注意：默认情况下，regex 不区分大小写，所以在“Software”中可找到“soft”，尽管 S 为大写字母。

但如果需要，可使用另一个不同的运算符 `-cmatch` 进行区分大小写的 `regex` 比较，如下所示：



```
"Software" -cmatch "soft"
```

由于在区分大小写比较中，字符串“soft”与“Software”不匹配，所以该表达式返回 `False`。请注意：尽管 `-match` 是默认行为，但也可选择使用 `-imatch` 运算符显式表示不区分大小写。

通配符和重复字符

`regex` 可包含若干通配符字符。例如，句点可与一个任意字符实例匹配。问号可与零个或一个任意字符实例匹配。示例如下：



```
"Don" -match "D.n" (True)
```

```
"Dn" -match "D.n" (False)
```

```
"Don" -match "D?n" (True)
```

```
"Dn" -match "D?n" (True)
```

在第一个表达式中，句点正好代表一个字符，所以匹配结果为 `True`。在第二个表达式中，句点未找到需要包含在其中的字符，所以匹配结果为 `False`。第三个和第四个表达式中的问号可匹配一个未知字符或者不与任何字符匹配。最后，在第四个示例中，由于“D”和“n”均存在并且二者之间没有字符，所以该匹配结果为 `True`。因此，问号可视为代表可选字符，所以即使在该位置没有出现任何字符，匹配结果仍为 `True`。

`regex` 还可将 `*` 和 `+` 符号视为重复字符。这些符号需要匹配某个字符或某些字符。`*` 可与零个或多个指定字符匹配，`+` 可与一个或多个指定字符匹配。示例如下：



```
follow
```

```
"DoDon" -match "Do*n" (True)
```

```
"Dn" -match "Do*n" (True)
```

```
"DoDon" -match "Do+n" (True)
```

```
"Dn" -match "Do+n" (False)
```

请注意：`*` 和 `+` 均可与“Do”匹配，而不只是与“o”匹配。这是因为这些重复字符被设计为可匹配一系列字符，而并不只是一个字符。

如果需要匹配句点、`*`、`?` 或 `+` 符号本身，应如何处理呢？可直接在它们前面加上一个反斜杠，作为 `regex` 转义符：



```
"D.n" -match "D\\.n" (True)
```

注意：该转义符与 Windows PowerShell 转义符（反单引号）不同，但是也遵循行业标准 `regex` 语法。

字符类

字符类是通配符的更广泛形式，它代表整组字符。Windows PowerShell 可识别很多字符类。例如：

`\w` 可匹配任何文字字符，即字母和数字。

`\s` 可匹配任何空白字符，如制表符、空格等。

\d 可匹配任何数字字符。

还有否定字符类：\W 可匹配非文字字符，\S 可匹配非空白字符，\D 可匹配非数字字符。这些类可后跟 * 或 + 来表示接受多个匹配。示例如下：



```
"Shell" -match "\w" (True)
```

```
"Shell" -match "\w*" (True)
```

本月 cmdlet

使用 Write-Debug cmdlet，可以轻松地将对象（如文本字符串）写入 Debug 管道。但在外壳程序中尝试该 cmdlet 可能会有点令人失望，因为该 cmdlet 看起来不起任何作用。

问题的症结在于 Debug 管道在默认情况下处于关闭状态 — \$DebugPreference 变量被设置为 “SilentlyContinue”。然而，如果将其设置为 “Continue”，则您使用 Write-Debug 发送的任何事物都将在控制台上以黄色文本显示。这是一种将跟踪代码添加到脚本的完美方法，通过该方法，您可完成对复杂脚本的执行。黄色有助于您分辨跟踪和脚本的正常输出，并且可随时关闭调试信息，而无需删除所有 Write-Debug 语句。只需再次设置 \$DebugPreference = “SilentlyContinue”，即可禁止调试文本。

尽管两个表达式都返回 True，但它们匹配的却是明显不同的对象。幸运的是，有一种方法可查看 -match 运算符实际上需要的内容：每完成一次匹配，无论运算符与 regex 匹配出的字符串中包含什么字符，都使用匹配结果填充称为 \$matches 的特殊变量。该 \$matches 变量保留其结果，直到使用 -match 运算符完成另一个完全匹配。图 1 显示了我刚才为您讲述的两个表达式的不同之处。您可以看到，\w 匹配 “Shell” 中的 “S”，而重复 \w* 匹配整个单词。

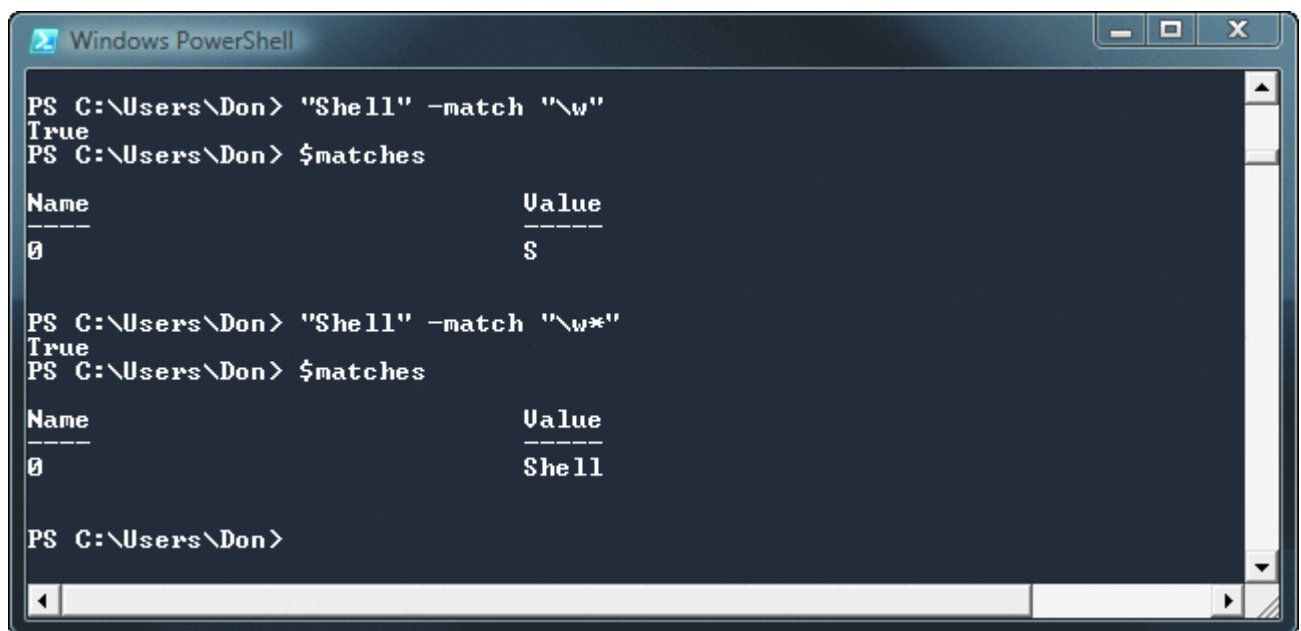


图 1 * 引起的差别

字符组、范围和大小

regex 还可包含字符组或范围（括在方括号中）。例如，[aeiou] 意味着其包含的任何字符（a、e、i 或 u）都是可接受的匹配。[a-zA-Z] 表明 a-z 或 A-Z 范围内的任何字母（即使使用的是不区分大小写的 -match 运算符，仅 a-z 或 A-Z 本身即可）都是可接受的。示例如下：



```
"Jeff" -match "[aeiou]ff" (True)
```

```
"Jeeeeeeeeeff" -match "[aeiou]ff" (False)
```

您还可使用大括号指定最小和最大字符数。{3} 表明您需要 3 个指定的字符,{3,} 表明至少需要 3 个或者更多指定的字符,{3,4} 表明至少需要 3 个但不超过 4 个指定字符。以下是一种为 IP 地址创建 regex 的理想方法：



```
"192.168.15.20" -match "\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}" (True)
```

该 regex 需要 4 组数字，每组数字包含 1 至 3 个数字，并且数组之间用文字句点隔开。但是请考虑以下示例：



```
"300.168.15.20" -match "\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}" (True)
```

此示例显示了 regex 的限制。虽然该字符串的格式看起来像 IP 地址，但它显然不是一个有效的 IP 地址。regex 无法确定数据是否真正有效；它只能确定数据看起来是否正确。

停止浮动查找

排除 regex 的故障似乎很简单。例如，以下是一个 regex，用于测试格式为 \\Server2\Share 的 UNC 路径：



```
"\\Server2\Share" -match "\\w+\\w+" (True)
```

在此，由于要测试的每个文字反斜杠必须使用另一个反斜杠进行转义，所以 regex 本身很难读取。这看似很正常，但事实并非如此：



```
"57\\Server2\Share" -match "\\w+\\w+" (True)
```

第二个示例很明显（至少在我看来）是一个无效的 UNC 路径，但 regex 却认为它是有效的 UNC 路径。为什么？请记住：默认情况下，regex 将浮动查找。此 regex 仅仅需要两个反斜杠、一个或多个字母和数字、另一个反斜杠以及更多的字母和数字。该字符串中存在这种模式，但是字符串开头还存在其他数字，这使得该字符串成为无效 UNC。解决办法是告知 regex 在字符串的开头开始匹配，而不能浮动查找。我可以按以下方法操作：



```
"57\\Server2\Share" -match "^\\w+\\w+" (False)
```

^ 字符指示字符串开始的位置。有了这个字符，该无效 UNC 路径将会失败，因为 regex 需要前两个字符为反斜杠，但这种情况并非如此。

同样，\$ 符号可用于指示字符串结尾。但此符号对 UNC 路径不会起到很大作用，因为 UNC 路径可能包含其他路径段，如 \\Server2\Share\Folder\File。不过，我确定在很多情况下，您都需要指定字符串结尾。

正则表达式帮助

在 Windows PowerShell 中，about_regular_expressions 帮助主题提供了有关 regex 语言的基本语法帮助，同时在线资源提供了更多相关信息。例如，我最喜欢的网站之一 www.RegExLib.com 提供了公众出于各种目的编写的免费正则表达式库。您可根据关键字进行搜索（如“e-mail”或“UNC”）以快速查找满足需要的 regex，或至少提供一个好的参考。如果您成功创建了一个很棒的 regex，则可将其添加到该库，以便其他人使用它。

此外，我也很喜欢 RegexBuddy (www.RegexBuddy.com)。此工具物美价廉，可提供 regex 图形编辑器。使用 RegexBuddy，可以更轻松地组建复杂的 regex 和对 regex 进行测试，以确保其正确接受有效字符串和拒绝无效字符串。许多其他软件开发人员还创建了免费、共享件以及商业 regex 编辑器和测试程序，这些工具一定会受到用户的青睐。

使用正则表达式

您也许想知道为什么要在现实生活中使用 regex。假设您正在查看 CSV 文件中的信息并要使用该信息在 Active Directory® 中创建新用户。如果该 CSV 文件是由其他人员生成的，则您可能需要证实其中的数据是否正确。regex 非常适合完成此项任务。例如，诸如 \w+ 的简单 regex 可确定名字和姓氏不包含任何特殊字符或符号，而稍复杂的 regex 则可确定电子邮件地址是否符合企业标准。例如，可使用以下 regex：



```
"^[a-z]+\.[a-z]+@contoso.com$"
```

此 regex 要求使用 don.jones@contoso.com 形式的电子邮件地址，其中名字和姓氏只能包含字母，并且这些名称必须用句点隔开。顺便提一下，对电子邮件地址字符串编写 regex 最复杂。如果您可将范围缩小至特定的企业标准，编写 regex 可以更简单一些。

不要忘记还有开始和结束定位标记（^ 和 \$），它们可确保 contoso.com 之后和构成用户名的字符之前不出现任何字符。

事实上，使用 Windows PowerShell 中的 regex 非常简单。假设变量 \$email 包含从 CSV 文件读取的电子邮件地址，可通过以下内容检查该邮件地址是否有效：



```
$regex = "^[a-z]+\.[a-z]+@contoso.com$"  
  
If ($email -notmatch $regex) {  
  
    Write-Error "Invalid e-mail address $email"  
  
}
```

在本示例中，您还认识了一个新的运算符。如果该字符串与所提供的 regex 不匹配，-notmatch 将返回 True（还有一个用于区分大小写比较的 -cnotmatch）。

PowerShell 进度报告

我最近正在编写一个冗长而复杂的 Windows PowerShell 脚本，它在运行时经常会失去响应。我编写了这个程序后打算把它作为计划任务运行，因此它实际上不会产生很多输出。当运行它进行第一次全面测试时，我很紧张甚至有些担心，

恐怕程序会出现死循环或其他脚本问题。当外壳停止运行并无助地闪烁着它的小光标时，我就问自己“要终止它吗？”显然我对自己没有信心，因为我很快就按下 Ctrl+C 中断了脚本。现在该是添加一些进度报告的时候了。

重复,重复,重复

我要做的第一件事是添加一组状态消息，让我能够确切知道脚本正在做什么。利用 Get-Help cmdlet，外壳可以很容易地实现此功能。继续操作并在外壳中试一试：



```
Write-Verbose "Test Message"
```

如果您刚刚试过，就会注意到它其实什么也没做。这是因为 Write-Verbose 将对象发送到特殊的 Verbose 管道，默认情况下，该管道不显示其输出。内置的外壳变量 \$VerbosePreference 控制此管道。此变量的默认值是 SilentlyContinue，它会抑制 verbose 输出。将其设置为 Continue，打开管道：



```
$VerbosePreference = "Continue"
```

现在可以将一组 Write-Verbose 声明添加到我的脚本中，并看一看运行时所发生的详细情况。此方法的吸引人之处在于，在完成测试和故障排除后，我可以通过在脚本开头将 \$VerbosePreference 设回 SilentlyContinue 来关闭所有多余的聊天。

没有必要前去删除所有 Write-Verbose 声明。实际上，由于它们就在脚本中，因此只要我需要手动运行脚本，我就可以随时很容易地将 Verbose 管道切换回来。

但是我需要真正的进度

如果脚本没有陷入死循环并且能够完美运行，我就会关闭 Verbose 管道然后重新运行一次 — 只是为了确认。

现在的问题是，我知道脚本在功能方面没有任何问题，但我就是无法面对闪烁着的光标。（我面对的就是注意力集中问题。在百无聊赖的情况下，我只好想尽一切办法来打发时间。）

我需要了解的是有关脚本进展程度的一般指示，以及它的预计完成时间。基本上来说，我希望有一个类似于进度条的东西。

幸运的是，Windows PowerShell™ 包括了 Write-Progress cmdlet。这个 cmdlet 并不提供与 Windows® 中的进度条一样的图形化进度条，但尽管如此，它还是会显示出一个不错的进度条，如图 1 所示。它有点像 Windows Server® 2003 或 Windows XP 中基于文本的安装部分所使用的文件复制进度条。

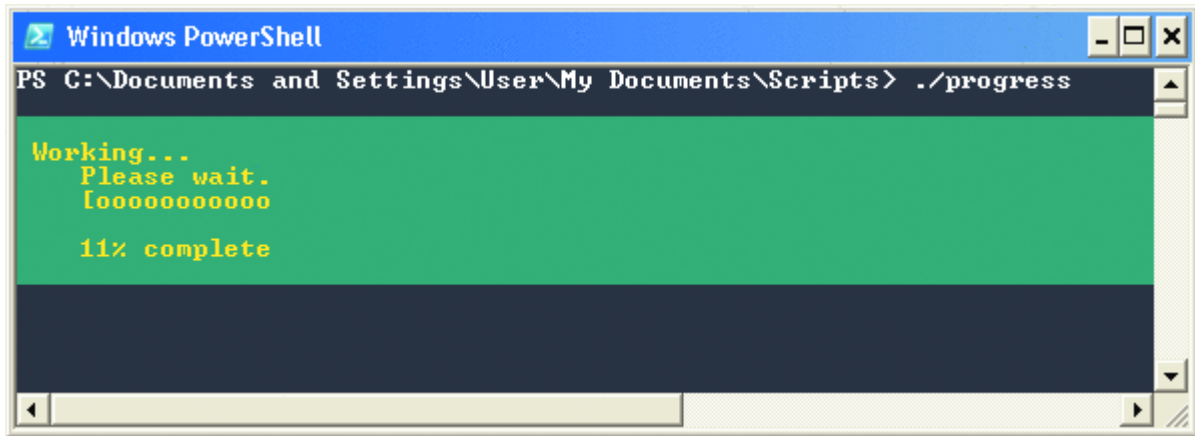


图 1 您的脚本进展到什么程度了？

使用 Write-Progress 需要进行一些说明。实际上，我认为举个例子会更好一些。请看下面的脚本：



```
for ($a=1; $a -lt 100; $a++) {  
  
    Write-Progress -Activity "Working..." `  
  
        -PercentComplete $a -CurrentOperation  
  
        "$a% complete" `  
  
        -Status "Please wait."  
  
    Start-Sleep 1  
  
}
```

它使用 Write-Progress 来显示进度条。我使用 Start-Sleep 使脚本每循环一次时都暂停一秒，以便它的执行速度足够慢，使我们能够看到进展情况；如果不暂停的话，从 0 循环到 100 会非常快，进度条在屏幕上只是一闪而过。

正如您所见，被我设置为 Working 的活动显示在进度条的顶部。Status 显示在它的正下方，而 CurrentOperation 显示在底部。外壳一次只支持一个进度条。每次使用 Write-Progress 时都会创建一个新的进度条（如果当前未显示）或更新当前显示的进度条。

在这里我还要告诉大家的是可以使进度条在操作执行完毕后消失。只需在脚本的结尾处添加下列代码即可：



```
Write-Progress -Activity "Working..." `  
  
    -Completed -Status "All done."
```

一般来说，进度条会在脚本完成后消失，但是如果脚本还有其他事情要做，您可能希望在使用脚本执行其他任务时，将进度条隐藏起来；-Completed 参数可以从显示画面中移除进度条。

嘀答...嘀答...嘀答...

Write-Progress 的另一个常见用法是显示“剩余秒数”，而不是实际的进度条。示例如下：



```
for ($a=100; $a -gt 1; $a--) {  
  
    Write-Progress -Activity "Working..." `  
  
        -SecondsRemaining $a -CurrentOperation  
  
        "$a% complete" `  
  
        -Status "Please wait."  
  
    Start-Sleep 1  
  
}
```

在这里将循环改为从 100 循环到 1，我将使用 Write-Progress 的 SecondsRemaining 参数，而不是 PercentComplete。结果如图 2 所示。正如您所见，进度条消失，取而代之的是一个倒计时的时钟。外壳自动将剩余的总秒数转换为小时、分钟和秒，为用户提供更加直观的信息。此处显示的完成百分比会从 100 开始减少，因为它只是我提供的 CurrentOperation 参数。没有实际计算的完成百分比；Windows PowerShell 只显示当前值 \$a，后跟字符串 "% complete"。

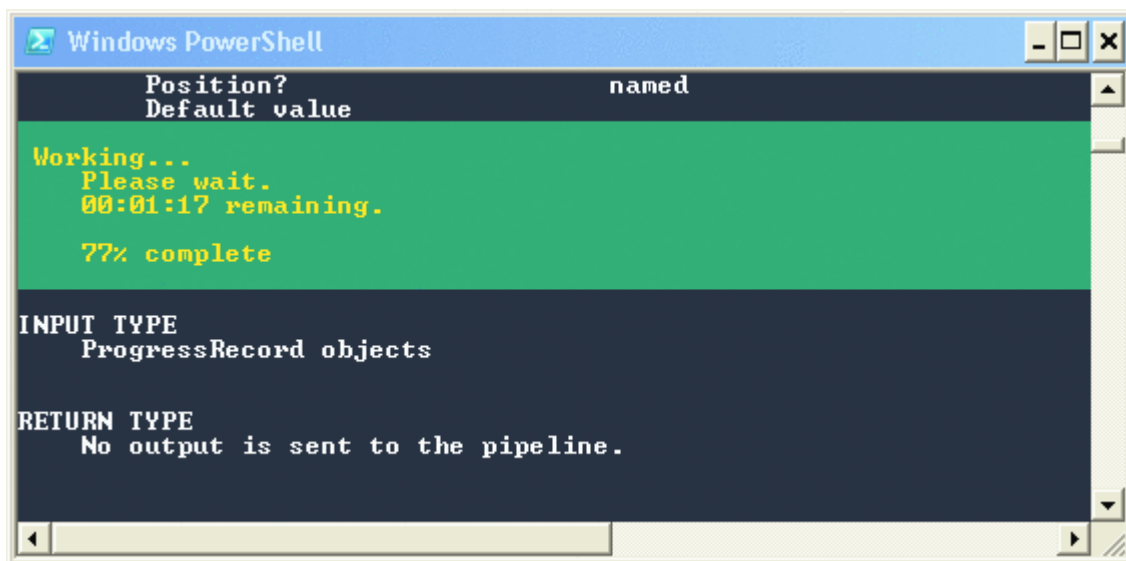


图 2 完成脚本要多长时间？

脚本通信

我非常热衷于编写脚本来显示脚本执行进展情况。它可以采用 verbose 输出的形式，也可以只是一个简单的进度条。它可以弥补我注意力不足的问题，也可以帮助以后需要运行我的脚本的其他用户。最后我要说的是，以某种形式显示状态和进度信息是非常有用的。

本月 Cmdlet：Tee-Object

在本月，我想展示一下我最喜欢的一个故障排除 cmdlet。以下面的代码为例：




```
Get-WMIObject Win32_Service | Where { $_.State -ne "Running"
```

```
-and $_.StartMode -eq "Automatic" } | ForEach-Object { $_.Start() }
```

表面上看，它好像要启动所有设置为自动启动的服务，但是由于某种原因还没有启动。它实际上并没有运行，并且要想找到它不运行的原因可能有些困难，因为您不能钻入管道中间分秒不离地查看。也就是说，唯一的办法是使用 Tee-Object。

Tee-Object 将对象重新定向到文件（或变量）并将其沿管道向下传递。例如：



```
Get-WMIObject Win32_Service | Tee-Object AllServices.csv | Where
```

```
{ $_.State -ne "Running" -and $_.StartMode -eq "Automatic" } |
```

```
Tee-Object FilteredServices.csv | ForEach-Object { $_.Start() }
```

经此修改后，即可看到执行每个管道命令后发生的情况，很快我发现我的 FilteredServices.csv 文件是空的！难怪这个脚本不能正常运行！通过进一步的研究我发现了导致出现问题的根本原因，StartMode 是 "Auto" 而不是 "Automatic"，是 Tee-Object 让我查明了出现问题的准确位置。

PowerShell 前景看好

我们往往希望脚本生成规范的输出，也经常有人问我怎样才是生成规范输出（如表）的最好方法。这种方法的确有很多种。第一步是编写脚本，将要以格式化形式显示的任何数据放置在 \$data1、\$data2 和 \$data3 变量中。现在，可以着手格式化数据。

文本方式

人们处理此任务最常见的方式就是像使用一种语言（如 VBScript、Perl、KiXtart 或其他面向文本的语言）对其进行处理一样。首先，我可以在屏幕上写出一组列标题：



```
Write-Host "ColumnA'tColumnB'tColumnC"
```

请注意，在 Windows PowerShell™ 中，`t` 是一个特殊转义符序列，表示插入一个制表符。要编写表的每一行（如前所述，我的数据位于 \$data1、\$data2 和 \$data3），我有多种方法可供选择。一种方法是只写入变量，依赖 shell 的功能用双引号内的内容替换变量：



```
Write-Host "$data1't$data2't$data3"
```

另外一种稍微吸引人的方法是使用 f 运算符来实现。此方法将格式与数据分开，可以用以下更加易读和维护的代码行来表示：



```
Write-Host "{0}'t{1}'t{3}" -f $data1,$data2,$data3
```

不过，整个方法有一些重大问题。首先，如果依靠控制台窗口中的固定制表位，则我必须自己处理一些奇怪的格式。例如，如果第一列中某个特殊行有一个包含 20 个字符的值，则我不得不对该行全部应用这种格式。此外，由于我使用 Write-Host 输出这些文本，因此几乎只能看到控制台显示。

将输出发送到文件或将其放在其他格式中并没有简单方法，尽管我一直期望有这样的简单方法。最重要的是，这种基于文本的方法完全忽略了我正在使用的在本质上基于对象的 shell，从而无法利用 Windows PowerShell 提供的各种强大的技术和功能。

Windows PowerShell 方式

Windows PowerShell 是面向对象的 shell。这意味着在理想情况下，您处理的所有内容都应该在对象中，从而 shell 可以在需要时将其转换为文本显示。但是，如何为任意数据片段创建对象？

仍采用我的示例，我首先创建一个空白自定义对象并将其存储在变量中：



```
$obj = New-Object PSObject
```

这样，我们就获得了一个全新的空白对象。然后，以属性形式向该对象中添加数据。为此，我仅将对象以管道形式传递到 Add-Member cmdlet 中。我添加了一个名为 NoteProperty 的属性并为该属性命名（最好使用列标题作为属性名称），然后，插入数据作为属性值：



```
$obj | Add-Member NoteProperty ColumnA $data1
```

```
$obj | Add-Member NoteProperty ColumnB $data2
```

```
$obj | Add-Member NoteProperty ColumnC $data3
```

现在，仅将该对象输出到管道（而不是控制台）：



```
Write-Output $obj
```

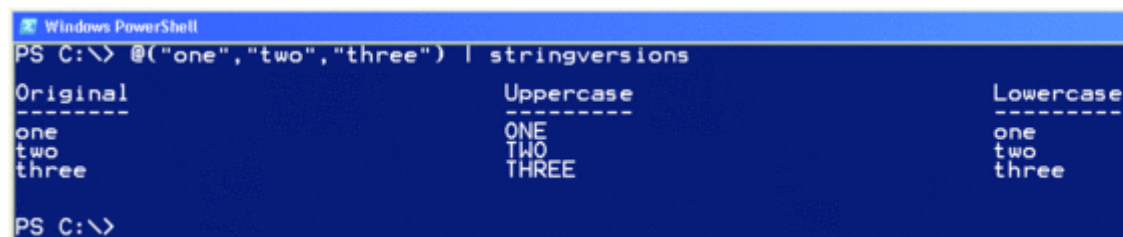
然后，可以对需要输出的每个表行重复这些步骤。以下简短函数的作用是接受一个字符串并输出其大写字母和小写字母版本以及原始字符串：



```
functionStringVersions {  
    param([string]$inputString)  
  
    $obj = New-Object PSObject  
  
    $obj | Add-Member NoteProperty Original($inputString)  
  
    $obj | Add-Member NoteProperty Uppercase($inputString.ToUpper())  
  
    $obj | Add-Member NoteProperty Lowercase($inputString.ToLower())  
  
    Write-Output $obj  
}  
  
$strings = @("one","two","three")  
  
foreach ($item in $strings) {  
    StringVersions $item  
}
```

以字符串变量的数组开始，一次处理一个变量并将其发送到函数中。该函数的输出被写入管道中。

请注意，此代码可以编写得更简单些，但以上代码更能清楚地说明我阐述的要点。其结果是一个格式相当完美的表，如图 1 中所示。这是因为 shell 已经知道如何在表中格式化对象。



```
PS C:\> @"one","two","three" | stringversions
Original                                Uppercase                                Lowercase
-----                                -
one                                    ONE
two                                    TWO
three                                  THREE
```

图 1 以表形式显示的 Windows PowerShell 输出

只要对象有四个或更少的属性，Windows PowerShell 就会自动选择表。因此，我无需自己执行任何操作，就获得了一个规范的表。

不过，请等一下，还有其他功能！

本月 Cmdlet：Get-Command

我相信您曾使用过一两次 Get-Command 或其简单的别名 (gcm) 来查看可用的 Windows PowerShell cmdlet 的列表。但是，您可能不知道 gcm 有多么灵活。例如，如果要查看 Windows PowerShell 可对一项服务执行的所有操作，请运行 `gcm -noun service`。或者，如果要查看所有的 Windows PowerShell 导出选项，请尝试 `gcm -verb export`。如果只需查看由特殊管理单元（如 PowerShell Community Extensions）添加的 cmdlet，请尝试 `gcm -psnapin pscx`。（用任何管理单元的名称替换“pscx”，即可查看该管理单元的 cmdlet。）

您可以看到，Get-Command 在 Windows PowerShell 的识别能力中发挥着重要作用。通过此命令，您无需参考手册即可了解哪些功能是可用的。另请注意，与使用 `Help *` 等命令相比，gcm 能够更精确地发现功能。Help 功能仅列出可用的帮助主题。帮助中未附带的任一 cmdlet 都不会显示在帮助列表中 — 但在需要时您可以调用这些 cmdlet。

这种方法的优点远不止是生成规范的表。只要您使用对象，Windows PowerShell 就可以帮助您做很多事情。要将您的数据放在 CSV 文件中？使用 `Export-CSV`。喜欢使用 HTML 表？将对象以管道形式传送到 `ConvertTo-HTML` 中。需要列表格式？以管道形式将其传送到 `Format-List`。通过使用对象，您可以利用 shell 已提供的所有功能。以下是修改后的示例：



```
functionStringVersions {
    PROCESS {
        $obj = New-Object PSObject
        $obj | Add-Member NoteProperty Original($_)
        $obj | Add-Member NoteProperty Uppercase($_.ToUpper())
        $obj | Add-Member NoteProperty Lowercase($_.ToLower())
        Write-Output $obj
    }
}
```

这次，我修改了函数以接受管道输入（使用对象时最好这样做）并输出到管道中。

现在，该函数在 PROCESS 脚本块内部有自己的代码。这意味着函数将接受管道输入，并为管道送入的每个对象执行一次 PROCESS 脚本块。

在 PROCESS 脚本块中，特殊的 \$_ 变量引用当前处理的管道对象。这样的实际结果是现在可以直接传送一个字符串数组：



```
@("one","two","three") | StringVersions
```

因此此函数将其输出放到管道中，所以我获得了一个表。在管道的末尾，shell 知道调用其格式化子系统，该子系统决定使用一个表，因为管道中对象的属性少于五个（默认情况下，如果有更多属性，shell 将使用列表）。

但是我不必依靠默认行为。我可以直接将这些对象传送到另一个 cmdlet 以便同时完成一些其他操作：



```
@("one","two","three") | StringVersions | Format-List
```

```
@("one","two","three") | StringVersions | ConvertTo-HTML | Out-File "strings.html"
```

```
@("one","two","three") | StringVersions | Export-CSV "strings.csv"
```

```
@("one","two","three") | StringVersions | Select Uppercase,Lowercase -unique
```

图 2 显示了 Web 浏览器中显示的第二个示例的结果 HTML。通过直接在对象中表示输出数据，而不是将其作为简单文本表示，即可完全访问 shell 中内置的丰富功能：各种格式化布局、HTML 转换、导出选项，以及排序、筛选和组合等。

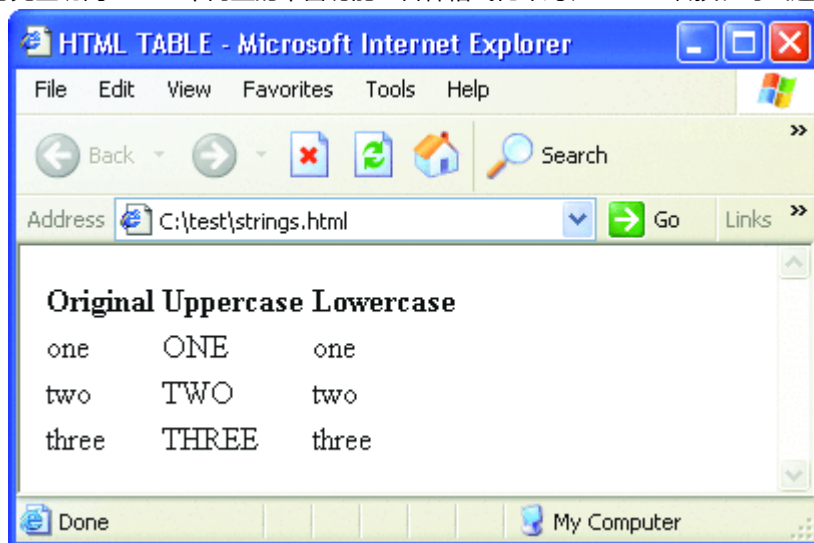


图 2 HTML 格式的 Windows PowerShell 数据输出

这是功能非常强大的素材。实际上，我甚至建议您写的每个脚本都应该产生对象作为它的输出，这样一来，您就可以用尽可能多的方法来利用该输出——而根本不必再写一行代码。