第 25 章　额外的提示，技巧以及技术

到目前为止，一个月的“午饭学习时间”已经接近尾声。因此我们想给你分享一些额外的提示与技巧完成这次学习之旅。

25.1　Profile、提示以及颜色：自定义 Shell 界面

每一个 PowerShell 进程开启时都是一样的：一样的别名，一样的 PSDrives，一样的色彩等。为什么不使用自定义的 Shell 界面呢？

25.1.1　PowerShell Profile 脚本

在前文中，我们阐述了 PowerShell 托管应用程序和 PowerShell 引擎本身的区别。PowerShell 的托管应用程序，比如 PowerShell ISE 的控制台，是指将命令发送至 PowerShell 引擎的一种方式。首先 PowerShell 引擎会执行命令，然后托管应用程序再显示执行结果。托管应用程序的另一个功能是当新开一个 Shell 窗口时，载入和运行 Profile 脚本。

这些 Profile 脚本可被用作自定义 PowerShell 的运行环境——能够自定义的包括：载入 SnapIn 管理单元或模块，切换到另外的根路径，定义需要使用的功能等。例如，下面是 Don 在计算机上使用的一个 Profile 脚本。

```
Import-Module ActiveDirectory
Add-PSSnapIn SqlServerCmdletSnapIn100
Cd C:\
```

该 Profile 载入了 Don 最常用的两个 Shell 的扩展程序，并修改根路径为 C 盘——C 盘也是 Don 喜欢使用的根路径。当然，你可以将你喜欢的任意命令放入 Profile 脚本中。

注意： 你可能认为没有必要载入 ActiveDirectory 模块，因为当用户尝试使用包含在该模块中的任一命令时，该模块会被隐式载入。该模块也会映射到一个 AD:PSDrive，Don 希望当新开一个 Shell 窗口时，该 AD:PSDrive 就处于可用状态。

在 PowerShell 中，并没有默认的 Profile 脚本存在，你创建的 Profile 脚本会依赖于你期望该脚本的工作方式。如果你需要查看详细信息，那么请执行 Help `About_Profiles`。当然，你最需要考虑的是，是否会用到多种 PowerShell 的托管应用程序。比如，我们倾向于在常规控制台和 PowerShell ISE 中来回切换，我们希望这两种托管应用程序都会运行相同的 Profile 脚本，所以需要确保在正确的路径下创建正确的 Profile 脚本。同时，我们也必须验证 Profile 脚本中的命令，因为该 Profile 脚本都会应用到控制台以及 ISE 托管应用程序——比如一些调整色彩等控制台设置的命令在 ISE 中可能会运行失败。

下面是控制台宿主尝试载入的一些文件，以及尝试载入这些文件的顺序。

（1）$PsHome/Profile.PS1——不管使用何种托管应用程序，计算机上的所有用户都会执行该脚本（请记住，PowerShell 已经预定义了`$PSHome`，该变量包含 PowerShell 的安装文件夹的路径）。

（2）$PsHome/Microsoft.PowerShell_Profile.PS1——如果该计算机上的用户使用了控制台宿主，那么就会执行该脚本。如果他们使用的是 PowerShell 的 ISE，那么会执行 $PsHome/Microsoft.PowerShellISE_Profile.ps1 脚本。

（3）$Home/Documents/WindowsPowerShell/Profile.PS1——无论用户使用的是何种托管应用程序，只有当前用户会执行该脚本（因为该脚本存在于用户的根目录下）。

（4）$Home/Documents/WindowsPowerShell/Microsoft.PowerShell_Profile.PS1——只有当前使用 PowerShell 控制台的用户才会执行该脚本。如果用户使用的是 PowerShell ISE，那么会执行$Home/Documents/WindowsPowerShell/Microsoft.PowerShellISE_Profile.PS1。

如果上面脚本中某一个或者几个不存在，那么也没关系。托管应用程序会跳过不存在的脚本，继续寻找下一个可用的脚本。

在 64 位操作系统上，由于存在独立的 32 位与 64 位的 PowerShell 程序，所以脚本也会包括 32 位与 64 位的版本。请不要期望相同的脚本在 32 位与 64 位 PowerShell 中都能正常运行。这意味着，某些模块或者扩展程序仅在某一个架构中才可用，所以请不要尝试使用一个 32 位的 Profile 脚本将某个 64 位的模块载入 32 位的 PowerShell 中，因为这根本不可能成功。

请注意，`About_Profiles` 的帮助文档与我们上面罗列的有一点不同。但是我们的经验可以证明，上面的列表是正确的。下面是针对该列表的其他一些知识点。

- $PsHome 是包含 PowerShell 安装路径信息的内置变量；在大部分操作系统中，该变量的值是 C:\Windows\System32\WindowsPowerShell\V1.0（针对 64 位操作系统上 64 位版本的 PowerShell）。

- $Home 是另一个内置的变量，该变量指向当前用户的配置文件夹（比如 C:\Users\Administrator）。
- 在前面的列表中，我们使用“Documents”表示文档文件夹，但是在某些版本的 Windows 系统中可能是“My Documents”。
- 在前面的列表中写的“不管用户使用何种托管应用程序”，从技术上讲并不恰当。准确地说，针对微软发布的托管应用程序（控制台或者 ISE），该命题正确；但是针对非微软发布的托管应用程序，根本无法使用该规则。

因为期望将相同的 Shell 扩展程序载入到 PowerShell，而不管使用控制台还是 ISE，所以我们选择自定义$Home\Documents\WindowsPowerShell\Profile1.PS1——因为该 Profile 脚本在微软提供的两种托管应用程序中都可以运行。

动手实验：为什么你自己不尝试创建一个或者多个 Profile 脚本呢？即使在这些脚本中仅打印出一些简单的信息，比如“It Worked”，这是查看不同脚本执行的一个好办法。但是请记住，你必须选择使用 Shell（或者 ISE），并且需要重新打开该 Shell（或者 ISE）去检查 Profile 脚本是否运行。

请记住，Profile 脚本也仅是脚本而已，它会依赖于 PowerShell 的当前执行策略。如果设置的执行策略是 Restricted，那么 Profile 脚本就无法运行；如果设置的执行策略是 AllSigned，那么 Profile 脚本必须经过签名才能运行。在第 17 章中讲到了执行策略以及脚本签名部分。如果你忘记了该知识点，请回到第 17 章重新学习。

25.1.2 自定义提示

PowerShell 提示——也就是你在本书中看到的 PS C:\>这类字符，是由一个名为提示（Prompt）的内置函数产生的。如果你希望自定义该提示，很简单，只需要替换该函数即可。可以在 Profile 脚本中定义一个新的提示函数，这样在你每次打开 Shell 界面的时候都可以采用新的提示函数。

下面是默认的提示函数。

```
Function Prompt
{
    $(IF (Test-Path Variable:/PSDebugContext) { '[DBG]: ' }
    ELSE { '' }) + 'PS ' + $(Get-Location) `
    + $(IF ($NestedPromptLevel -Ge 1) { '>>' }) + '> '
}
```

该函数首先会检测$DebugContext 变量是否被预定义在 PowerShell 的 Variable:Drive 中。如果有，那么该函数就会将[DBG]:添加到提示启动阶段。否则，该提示会被定义为 PS 再加上由 Get-Location Cmdlet 返回的当前路径（比如 PS D:\Test>）。如果该 Shell 处于嵌套提示中——由内置函数$NestedPromptLevel 返回，那么提示

中会添加“>>”字样。

下面是自定义的一个提示函数。你可以直接将该函数加入到任意 Profile 脚本中，这样可以保证后续新开启的 Shell 进程都会将该提示作为一个标准提示函数使用。

```
Function Prompt {
 $Time = (Get-Date).ToShortTimeString()
 "$Time [$ENV:COMPUTERNAME]:> "
}
```

该自定义函数会返回当前时间，后面接着当前计算机名称（计算机名称包含在中括号内）。

```
6:07 PM [CLIENT01]:>
```

在这里，通过双引号改变了 PowerShell 特定的行为——PowerShell 会使用双引号中的内容来替换变量（比如$Time）的值。

25.1.3 调整颜色

在前面的章节中，我们看到，当 Shell 界面报出很多错误时，我们觉得多么刺眼。当 Don 还是一个小孩的时候，他在英语课堂上总是很痛苦——因为他总是能看到汉森女士批改之后的文章（使用红笔标出的红色文字的提醒）。但是幸运的是，在 PowerShell 中，你可以修改 PowerShell 所使用默认颜色的选项。

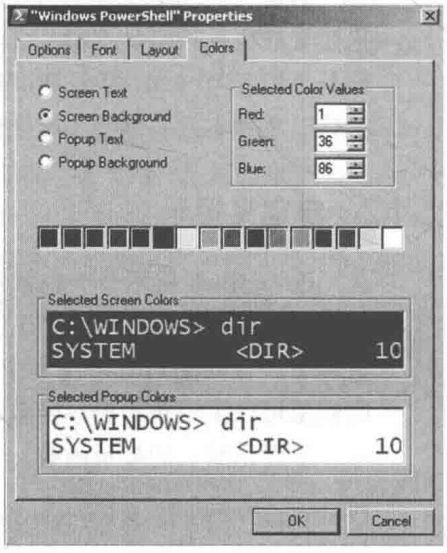

图 25.1 配置默认 Shell 界面颜色

默认的文本前景色与后景色都可以通过单击 PowerShell 命令窗口左上角的边框来修改。选择“属性”，之后切换到“颜色”标签页，如图 25.1 所示。

修改错误、警告以及其他信息的颜色略微有点复杂，需要通过运行命令才能实现。但是你可以将这部分命令放到 Profile 脚本中，这样每次进入 PowerShell 时，都会执行这些命令。比如下面的命令可以将错误消息的前景色修改为绿色，这样你可以觉得稍微舒缓一点。

```
(Get-Host).PrivateData.ErrorForegroundColor= "Green"
```

我们可以通过命令修改下列设置的颜色。

- ErrorForegroundColor
- ErrorBackgroundColor
- WarningForegroundColor

- WarningBackgroundColor
- DebugForegroundColor
- DebugBackgroundColor
- VerboseForegroundColor
- VerboseBackgroundColor
- ProgressForegroundColor
- ProgressBackgroundColor

下面是可以选择的几种颜色。

- Red
- Yellow
- Black
- White
- Green
- Cyan
- Magenta
- Blue

同时，也存在这些颜色的对应深色颜色：DarkRed，DarkYellow，DarkGreen，DarkCyan，DarkBlue 等。

25.2 运算符：-AS、-IS、-Replace、-Join、-Split、-IN、-Contains

这些额外的运算符在多种情形下都非常有用，可以通过它们来处理数据类型、集合与字符串。

25.2.1 –AS 和–IS

-AS 运算符会将一种已存在的对象转换为新的对象类型，从而产生一个新的对象。例如，如果存在一个包含小数的数字（可能来自一个除法计算），可以通过 Converting 或者 Casting 将该数字转化为一个整数。

```
1000 / 3 -AS [INT]
```

语句的结构：首先是一个将被转换的对象，然后是-AS 运算符，最后是一个中括号，中括号中包含转化之后的类型。这些类型可以是[String]、[XML]、[INT]、[Single]、[Double]、[Datetime]等，罗列的这些类型应该是你经常使用到的类型。从技术上讲，在该示例中，将数值转化为整数是指将小数部分通过四舍五入方式转为整数，而并不是简单地将小数部分去掉。

-IS 运算符通过类似方式实现。该运算符主要用于判断某个对象是否为特定类型，如果是，则返回 True，否则为 False。比如下面的这些示例。

```
123.45 -IS [INT]
"SERVER-R2" -IS [String]
$True -IS [Bool]
(Get-Date) -IS [DateTime]
```

动手实验：请执行上面每一条命令，然后确认其返回结果。

25.2.2 –Replace

-Replace 运算符主要用于在某个字符串中寻找特定字符(串)，最后将该字符(串)替换为新的字符（串)。

```
PS C:\> "192.168.34.12" -Replace "34","15"
192.168.15.12
```

命令的结构：首先是源字符串，之后为-Replace 运算符。然后需要提供在源字符串中寻找的字符（串)，最后跟上一个逗号外加最新的字符（串)。在上面的示例中，我们将字符串中的“34”替换为“15”。

25.2.3 –Join 和-Split

-Join 和-Split 运算符主要用作将数组转化为分隔列表和将分隔列表转化为数组。

例如，存在包含 5 个元素的数组。

```
PS C:\> $Array = "one","two","three","four","five"
PS C:\> $Array
one
two
three
four
five
```

因为 PowerShell 会自动将使用逗号隔开的列表识别一个数组，所以上面的命令可以执行成功。假如现在需要将这个数组里的值转换为以管道符隔开的字符串，可以通过-Join 来实现。

```
PS C:\> $Array -Join "|"
one|two|three|four|five
```

可以将该执行结果存入一个变量，这样可以直接重用，或者将其导出为一个文件。

```
PS C:\> $String = $Array -Join "|"
```

```
PS C:\> $String
one|two|three|four|five
PS C:\> $String | Out-File Data.DAT
```

同时，我们可以使用-Split 运算符实现相反的效果：它会从一个分隔的字符串中产生一个数组。例如，假如存在仅包含一行四列数据的一个文件，在该文件中以制表符对列进行隔离。将该文件的内容显示出来，类似下面这样。

```
PS C:\>Gc Computers.tdf
Server1 Windows East  Managed
```

请记住，这里的 Gc 是 Get-Content 的别名。

你可以通过-Split 运算符将该内容拆成 4 个独立的数组元素。

```
PS C:\> $Array = (Gc Computers.tdf) -Split "`t"
PS C:\> $Array
Server1
Windows
East
Managed
```

请注意，这里我们使用转义字符、一个重音符以及一个“t”（`t）表示制表符。这些字符必须包含在一个双引号中，这样 PowerShell 才能识别该转义字符。

产生的数组中包含 4 个元素，可以通过它的索引编号单独查询对应元素。

```
PS C:\> $Array[0]
Server1
```

25.2.4 –Contains 和–In

-Contains 运算符对 PowerShell 初学者而言可能会比较容易混淆。他们可能会尝试下面的脚本。

```
PS C:\> 'this' -Contains '*his*'
False
```

实际上，他们是期望运行-like 运算符。

```
. PS C:\> 'this' -Like '*his*'
True
```

-Like 运算符用于进行通配符比较运算。-Contains 运算符主要用作在一个集合中查找是否存在特定对象。比如，创建包含多个字符串对象的一组集合，然后检查特定对象是否包含在该集合中。

```
PS C:\> $Collection = 'abc','def','ghi','jkl'
PS C:\> $Collection -Contains 'abc'
True
PS C:\> $Collection -Contains 'xyz'
False
```

-In 运算符实现相同的功能，但是它会颠倒运算对象的顺序。也就是说，集合在右边，而需要检查的对象在左边。

```
PS C:\> $Collection = 'abc','def','ghi','jkl'
PS C:\> 'abc' -IN $Collection
True
PS C:\> 'xyz' -IN $Collection
False
```

25.3 字符串处理

假如存在一个字符串，你需要将该字符串全部转化为大写，或者你可能需要取得该字符串的最后 3 个字符。那么应该如何实现？

在 PowerShell 中，字符串是对象，所以就会存在多种方法（Method）。方法是通知对象去做某项工作的方式，通常是针对对象本身。可以将该对象通过管道发送给 Gm 查看该对象可用的方法。

```
PS C:\> "Hello" | Gm

   TypeName: System.String
Name             MemberType          Definition
--------         -------------       ----------------
Clone            Method              System.Object Clone()
CompareTo        Method              int CompareTo(System.Object value...
Contains         Method              bool Contains(string value)
CopyTo           Method              System.Void CopyTo(int sourceInde...
EndsWith         Method              bool EndsWith(string value), bool...
Equals           Method              bool Equals(System.Object obj), b...
GetEnumerator    Method              System.CharEnumerator GetEnumerat...
GetHashCode      Method              int GetHashCode()
GetType          Method              type GetType()
GetTypeCode      Method              System.TypeCode GetTypeCode()
IndexOf          Method              int IndexOf(char value), int Inde...
IndexOfAny       Method              int IndexOfAny(char[] anyOf), int...
Insert           Method              string Insert(int startIndex, str...
IsNormalized     Method              bool IsNormalized(), bool IsNorma...
LastIndexOf      Method              int LastIndexOf(char value), int ...
```

```
LastIndexOfAny   Method                int LastIndexOfAny(char[] anyOf),...
Normalize        Method                string Normalize(), string Normal...
PadLeft          Method                string PadLeft(int totalWidth), s...
PadRight         Method                string PadRight(int totalWidth), ...
Remove           Method                string Remove(int startIndex, int...
Replace          Method                string Replace(char oldChar, char...
Split            Method                string[] Split(Params char[] sepa...
StartsWith       Method                bool StartsWith(string value), bo...
Substring        Method                string Substring(int startIndex),...
ToCharArray      Method                char[] ToCharArray(), char[] ToCh...
ToLower          Method                string ToLower(), string ToLower(...
ToLowerInvariant Method                string ToLowerInvariant()
ToString         Method                string ToString(), string ToStrin...
ToUpper          Method                string ToUpper(), string ToUpper(...
ToUpperInvariant Method                string ToUpperInvariant()
Trim             Method                string Trim(Params char[] trimCha...
TrimEnd          Method                string TrimEnd(Params char[] trim...
TrimStart        Method                string TrimStart(Params char[] tr...
Chars            ParameterizedProperty char Chars(int index) {get;}
Length           Property              System.Int32 Length {get;}
```

下面是一些比较有用的 String 方法。

- IndexOf()会返回特定字符在字符串中的位置。

```
PS C:\> "SERVER-R2".IndexOf("-")
6
```

- Split()，Join()和 Replace()类似于上面讲到的-Split，-Join 和-Replace。但是我们更加倾向于使用 PowerShell 的运算符而不是 String 的方法。
- ToLower()和 ToUpper()可以将字符串转化为小写或大写。

```
PS C:\> $ComputerName = "SERVER17"
PS C:\> $ComputerName.tolower()
server17
```

- Trim()会将一个字符串的前后空格去掉；TrimStart()和 TrimEnd()会将一个字符串的前面或者后面的空格去掉。

```
PS C:\> $UserName = " Don"
PS C:\> $UserName.Trim()
Don
```

上面这些方法都是处理或者修改 String 对象比较方便的方法。请记住，所有这些方法既可以运用于包含字符串的变量（比如前面的 ToLower()和 Trim()示例），也可以用在一个静态的字符串上（比如前面的 IndexOf()示例）。

25.4 日期处理

和 String 类型对象一样，Date（如果你喜欢，也可以是 DateTime）对象也包含多个方法。通过这些方法，可以对日期和时间进行处理和计算。

```
PS C:\>Get-Date | Gm

   TypeName: System.DateTime

Name                 MemberType     Definition
----                 ----------     ----------
Add                  Method         System.DateTime Add(System.TimeSpan ...
AddDays              Method         System.DateTime AddDays(double value)
AddHours             Method         System.DateTime AddHours(double value)
AddMilliseconds      Method         System.DateTime AddMilliseconds(doub...
AddMinutes           Method         System.DateTime AddMinutes(double va...
AddMonths            Method         System.DateTime AddMonths(int months)
AddSeconds           Method         System.DateTime AddSeconds(double va...
AddTicks             Method         System.DateTime AddTicks(long value)
AddYears             Method         System.DateTime AddYears(int value)
CompareTo            Method         int CompareTo(System.Object value), ...
Equals               Method         bool Equals(System.Object value), bo...
GetDateTimeFormats   Method         string[] GetDateTimeFormats(), strin...
GetHashCode          Method         int GetHashCode()
GetType              Method         type GetType()
GetTypeCode          Method         System.TypeCode GetTypeCode()
IsDaylightSavingTime Method         bool IsDaylightSavingTime()
Subtract             Method         System.TimeSpan Subtract(System.Date...
ToBinary             Method         long ToBinary()
ToFileTime           Method         long ToFileTime()
ToFileTimeUtc        Method         long ToFileTimeUtc()
ToLocalTime          Method         System.DateTime ToLocalTime()
ToLongDateString     Method         string ToLongDateString()
ToLongTimeString     Method         string ToLongTimeString()
ToOADate             Method         double ToOADate()
ToShortDateString    Method         string ToShortDateString()
ToShortTimeString    Method         string ToShortTimeString()
ToString             Method         string ToString(), string ToString(s...
ToUniversalTime      Method         System.DateTime ToUniversalTime()
DisplayHint          NoteProperty   Microsoft.PowerShell.Commands.Displa...
Date                 Property       System.DateTime Date {get;}
Day                  Property       System.Int32 Day {get;}
DayOfWeek            Property       System.DayOfWeek DayOfWeek {get;}
```

```
DayOfYear            Property       System.Int32 DayOfYear {get;}
Hour                 Property       System.Int32 Hour {get;}
Kind                 Property       System.DateTimeKind Kind {get;}
Millisecond          Property       System.Int32 Millisecond {get;}
Minute               Property       System.Int32 Minute {get;}
Month                Property       System.Int32 Month {get;}
Second               Property       System.Int32 Second {get;}
Ticks                Property       System.Int64 Ticks {get;}
TimeOfDay            Property       System.TimeSpan TimeOfDay {get;}
Year                 Property       System.Int32 Year {get;}
DateTime             ScriptProperty System.Object DateTime {get=if ((& {...
```

请记住，通过上面列表中的属性可以访问一个 DateTime 的部分数据，比如日期、年或者月。

```
PS C:\> (Get-Date).Month
10
```

上面列表中的方法可以实现两个功能：计算或者将 DateTime 转化为其他格式。例如，假如需要获取 90 天之前的日期，使用 AddDays()方法和一个负数参数实现。

```
PS C:\> $Today = Get-Date
PS C:\> $90DaysAgo = $Today.AddDays(-90)
PS C:\> $90DaysAgo

2014年12月19日 9:36:47
```

名称中以“To”开头的方法可以实现将日期以及时间转化为某种特定格式，比如短日期类型。

```
PS C:\> $90DaysAgo.ToShortDateString()
2014/12/19
```

另外需要注意的是，这些方法都是依赖于当前计算机本地的区域设定——区域设定决定了日期和时间格式。

25.5 处理 WMI 日期

在 WMI 中存储的日期和时间格式都难以直接利用。例如，Win32_OperatingSystem 类主要用来记录计算机上一次启动的时间，其日期和时间格式如下。

```
PS C:\> Get-WMIObject Win32_OperatingSystem | Select LastBootUpTime

LastBootUpTime
--------------
20150317090459.125599+480
```

PowerShell 的设计者知道直接使用这些信息会比较困难，所以他们对每一个 WMI 对象添加了一组转换方法。将 WMI 对象通过管道发送给 Gm，请注意观察最后两个方法。

```
PS C:\> Get-WMIObject Win32_OperatingSystem | Gm
   TypeName: System.Management.ManagementObject#root\cimv2\Win32_OperatingS
ystem
Name                                   MemberType   Definition
------                                 ----------   ----------
Reboot                                 Method       System.Management...
SetDateTime                            Method       System.Management...
Shutdown                               Method       System.Management...
Win32Shutdown                          Method       System.Management...
Win32ShutdownTracker                   Method       System.Management...
BootDevice                             Property     System.String Boo...
...
PSStatus                               PropertySet  PSStatus {Status,...
ConvertFromDateTime                    ScriptMethod System.Object Con...
ConvertToDateTime                      ScriptMethod System.Object Con...
```

将输出结果集中间的大部分信息去除，这样你能很轻易地发现后面的 ConvertFromDateTime()和 ConvertToDateTime()方法。在该示例中，获取到的是 WMI 的日期和时间。假如需要转化为正常的日期和时间格式，请参照下面的命令。

```
PS C:\> $OS = Get-WMIObject Win32_OperatingSystem
PS C:\> $OS.ConvertToDateTime($OS.LastBootUpTime)

2015年3月17日 9:04:59
```

如果你期望将正常的日期和时间信息放入到一个正常表中，你可以通过 Select-Object 或者 Format-Table 命令创建自定义计算列以及属性。

```
PS C:\> Get-WMIObject Win32_OperatingSystem |Select BuildNumber,_Server,
@{l='LastBootTime';E={$_.ConvertToDateTime($_.LastBootUpTime)}}

BuildNumber           __Server              LastBootTime
-----------           ---------             ------------
7601                  SERVER-R2             2015/3/17 9:04:59
```

25.6 设置参数默认值

大多数 PowerShell 命令至少都有几个参数包含默认值。例如，运行 Dir 命令，默认会指向当前路径，而并不需要指定-Path 参数。在 PowerShell 第 3 版之后（包含第 3 版），可以对任意命令的任意参数——甚至是针对多个命令，指定自定义的默认值。当运行不带有指定参数的命令时，才会采用设定的默认值；但是当运行命令时手动指定了参数以及对应值，之前设定的默认值会被覆盖。

默认值保存在名为$PSDefaultParameterValues 的特殊内置变量中。当每次新开一个 PowerShell 窗口时，该变量均置空，之后使用一个哈希表填充该变量（可以通过 Profile 脚本使得默认值始终有效）。

例如，假如你希望创建一个包含用户名以及密码的凭据对象，然后将该对象设置为所有命令中-Credential 参数的默认值。

```
PS C:\> $Credential = Get-Credential -UserName Administrator
➥-Message "Enter Admin Credential"
PS C:\> $PSDefaultParameterValues.Add('*:Credential',$Credential)
```

或者，如果仅希望 Invoke-Command Cmdlet 每次运行时都提示需要凭据，此时请不要直接分配一个默认值，而是分配一段执行 Get-Credential 命令的脚本块。

```
PS C:\> $PSDefaultParameterValues.Add('Invoke-Command:Credential',
➥{Get-Credential -Message 'Enter Administrator Credential'
➥-UserName Administrator})
```

可以看到该 Add()方法的基本格式：第一个参数为<Cmdlet>:<Parameter>，该<Cmdlet>可以接受*等通配符。Add()方法的第二个参数或者是直接给出的默认值，或者是执行其他（一个或多个）命令的脚本块。

可以执行下面的命令，查看$PSDefaultParameterValues 包含的内容。

```
PS C:\>$PSDefaultParameterValues

Name                           Value
-------                        -------
*:Credential                   System.Management.Automation.PSCredenti
Invoke-Command:Credential      Get-Credential -Message 'Enter administ
```

补充说明

PowerShell 的变量由作用域（Scope）控制。我们在第 21 章中简单介绍了作用域，同时作用域也会影响参数的默认值。

如果在命令行中设置了$PSDefaultParameterValues，那么该参数会针对本 Shell 会话中的所有脚本以及命令起作用。但是如果仅在一段脚本中设置了$PSDefaultParameterValues，那么同样，也只会在该脚本作用域中有用。该技术非常有用，因为这意味着你可以在一段脚本中设置多个参数的默认值，但是并不影响其他脚本或者 Shell 会话的运行。

作用域的核心思想是“无论脚本发生了什么，仅会影响该脚本”。如果你想深入研究作用域，请查阅 About_Scope 帮助文档中的详细内容。

可以通过 PowerShell 中的 About_Parameters_Default_Values 帮助文档查看该特性更多的知识点。

25.7 学习脚本块

脚本块是 PowerShell 的一个关键知识点。之前你可能已经能简单地使用脚本块了。

- Where-Object 命令的-FilterScript 参数会使用脚本块。
- ForEach-Object 命令的-Process 参数会使用脚本块。
- 使用 Select-Object 创建自定义属性的哈希表或者使用 Format-Table 创建自定义列的哈希表，都会需要一个脚本块作为 E 或者 Expression 的键值。
- 正如本章前面所讲，参数的默认值也可以是一个脚本块。
- 针对一些远程处理以及 Job 相关的命令，比如 Invoke-Command 和 Start-Job 命令，也需要一个脚本块作为-ScriptBlock 参数的值。

那么，什么是脚本块呢？简单来讲，脚本块是指包含在大括号中的全部命令——哈希表除外（哈希表在大括号之前会带有@符号）。你可以在命令行中输入一个脚本块，然后将该脚本块赋值给一个变量，再使用&该调用运算符来执行该脚本块。

可以使用脚本块完成更多的工作。如果希望进一步学习脚本块，请参阅 PowerShell 中的 About_Script_Block 帮助文档。

```
PS C:\> $Block = {
➥ Get-Process | Sort -Property Vm -Descending | Select -First 10 }
PS C:\>&$Block

Handles  NPM(K)    PM(K)      WS(K)     VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      -----     -----   ------     -- -----------
    680      42    14772      13576      1387     3.84    404 svchost
    454      26    68368      75116       626     1.28   1912 powerShell
    396      37   179136      99252       623     8.45   2700 powerShell
    497      29    15104       6048       615     0.41   2500 SearchIndexer
    260      20     4088       8328       356     0.08   3044 taskhost
    550      47    16716      13180       344     1.25   1128 svchost
   1091      55    19712      35036       311     1.81   3056 explorer
    454      31    56660      15216       182    45.94   1596 MsMpEng
    163      17    62808      27132       162     0.94   2692 dwm
    584      29     7752       8832       159     1.27    892 svchost
```

25.8 更多的提示、技巧及技术

正如本章开始所说，本章只是展示一些需要让你知晓的知识点，但是这些知识点并未出现在之前的章节中。当然，在逐渐学习 PowerShell 的过程中，你会遇到更多的提示以及技巧，也会获得更多的经验。

你也可以订阅我们的 Twitter：@jeffhicks 和@concentrateddon。我们会定期在 Twitter 上分享一些有用的提示以及小技巧。PowerShell.Org 网站上也提供邮件列表定期推送一些小技巧，别忘了还有 PowerShell.Org 的论坛。有些时候，通过点滴的学习，你可以更容易在某技术领域成为专家，所以请将这些提示、技巧以及技术，包括以后会遇到的其他资源作为不断提高 PowerShell 水平的一种沉淀吧！