第 18 章　变量：一个存放资料的地方

前面已经提到过，PowerShell 包含脚本语言，并且在前面几章中已经开始与脚本语言打交道。但是一旦开始编写编程，就需要了解什么是“变量”，所以我们以此作为本章开端。你可以在其他复杂的脚本中使用变量，因此我们也会展示如何在这些地方使用变量。

18.1　变量简介

简单来说，变量就是在内存中的一个带有名字的“盒子”。你可以把所有你想存放的东西都放入这个“盒子”中：一个计算机名称、一系列服务的集合、XML 文档等。然后通过名字去访问这个盒子。在访问过程中，可以存放、添加或者从里面检索东西。这些东西是一直驻留在盒子里面的，并且允许你反复使用它们。

PowerShell 并没有对变量有太多限制。比如，你不需要在使用变量前对其进行显式声明或定义。你也可以更改变量值的类型：某个时刻你可能只存储了一个进程在里面，下一时刻又可能存储一系列的计算机名进去。变量甚至可以存储多种不同的东西，比如服务的集合和进程的集合（虽然允许这样做，但是大部分情况下，使用变量的内容还是有讲究的）。

18.2　存储值到变量中

PowerShell 中的所有东西——的确是所有东西，都被认为是一个对象。即使一个简单的字符串，比如计算机名，都被当作对象对待。比如，把一个字符串用管道传输到 Get-Member（或者它的别名 Gm），可以看到对象的类型是“System.String”，并且有很多方法可用（为了节省空间，这里截断了部分输出）。

```
PS C:\> "SERVER-R2" | gm

   TypeName: System.String

Name             MemberType          Definition
----             ----------          ----------
Clone            Method              System.Object Clone()
CompareTo        Method              int CompareTo(System.Object valu...
Contains         Method              bool Contains(string value)
CopyTo           Method              System.Void CopyTo(int sourceInd...
EndsWith         Method              bool EndsWith(string value), boo...
Equals           Method              bool Equals(System.Object obj), ...
GetEnumerator    Method              System.CharEnumerator GetEnumera...
GetHashCode      Method              int GetHashCode()
GetType          Method              type GetType()
GetTypeCode      Method              System.TypeCode GetTypeCode()
IndexOf          Method              int IndexOf(char value), int Ind...
IndexOfAny       Method              int IndexOfAny(char[] anyOf), in...
```

动手实验：在你自己的电脑上运行命令，看是否能获取来自于“System.String”对象的完整的方法和属性的列表。

虽然从技术角度讲字符串是一个对象，但是和其他 Shell 中的东西一样，你会发现人们更倾向于把它当作一个简单的值。因为大部分情况下，我们关注的是它的值（如前面提到的“SERVER-R2”），而不会过多关注从属性中查找信息。也就是说，一个进程就算很庞大，数据结构很抽象，而你通常只需要处理一些单独的属性，如 VM、PM、Name、CPU、ID 等。一个字符串是一个对象，但是相比常见的进程，它又显得没那么复杂。

PowerShell 允许在一个变量中存储简单的值。你需要定义一个变量，然后使用等号符（=），用于赋值操作，接下来是变量所需存储的值。下面是例子。

```
PS C:\> $var = "SERVER-R2"
```

动手实验：动手运行这些例子，以便你能重现我们的结果。但是需要把服务器名改为本地，而不是使用“SERVER-R2”。

需要注意的是，美元符（$）并不是变量名称的一部分。在我们的例子中，变量名称是“var”。“$”符号只是告知 Shell 接下来的是一个变量名，并且将要赋值给这个变量。

下面我们看看关于变量及其名称的一些注意事项。

- 变量名称通常包含字母、数字和下划线，最常见的形式是以字母或下划线开头。
- 变量名称可以包含空格，但是名字必须被花括号包住。比如${My Variable}，表示一个变量名“My Variable”。就我个人而言，我不喜欢变量名包含空格，因为这会要求更多的输入操作，并且不易阅读。

- 变量不会驻留在 Shell 会话之间。当关闭 Shell 时，所有你创建的变量都会被清除。
- 变量名称可以很长——长到你可以不用考虑它到底能有多长。但是请确保变量名称的可读性。比如，如果你想要把计算机名存入变量，可以使用“computername”作为变量名称。如果变量需要包含一系列的进程，使用“processes”是个不错的选择。
- 除了有 VBScript 背景的人，PowerShell 用户通常不需要使用前缀名来标识变量存放了什么。比如在 VBScript 中，“strComputerName”是常见的变量名称，表示变量存储的是一个字符串（“str”部分）。PowerShell 不在意你是否这样做。同时在大多数社区中，这种习惯也不被认为是好习惯。

如果需要查询变量的内容，可以使用美元符号加上变量名称，像下面的例子所实现的。再次提醒，美元符号只是告诉 Shell 你需要访问的变量内容；紧跟其后的变量名称才是告诉 Shell 你要访问的变量是什么。

```
PS C:\> $var
SERVER-R2
```

你可以在几乎所有地方使用变量来替代值。比如，当使用 WMI 时，你可以选择指定一个计算机名称。该命令类似：

```
PS C:\> get-wmiobject win32_computersystem -comp SERVER-R2

Domain              : company.pri
Manufacturer        : VMware, Inc.
Model               : VMware Virtual Platform
Name                : SERVER-R2
PrimaryOwnerName    : Windows User
TotalPhysicalMemory : 3220758528
```

然后可以使用变量替代该值。

```
PS C:\> get-wmiobject win32_computersystem -comp $var

Domain              : company.pri
Manufacturer        : VMware, Inc.
Model               : VMware Virtual Platform
Name                : SERVER-R2
PrimaryOwnerName    : Windows User
TotalPhysicalMemory : 3220758528
```

顺带说说，var 的确是我们常见的变量名称。我们认为使用“computername”是不错的选择，但是在一些特殊地方，将会重复使用$var 作为变量名称，所以这里还是保持使用 var。但不要因为这个例子使你放弃使用有意义的名字作为变量名称。

下面将从赋值给变量$var 开始，但我们可以在任意时刻修改该变量的值。

```
PS C:\> $var = 5
PS C:\> $var | gm

   TypeName: System.Int32
Name        MemberType Definition
----        ---------- ----------
CompareTo   Method     int CompareTo(System.Object value), int CompareT...
Equals      Method     bool Equals(System.Object obj), bool Equals(int ...
GetHashCode Method     int GetHashCode()
GetType     Method     type GetType()
GetTypeCode Method     System.TypeCode GetTypeCode()
ToString    Method     string ToString(), string ToString(string format...
```

在前面的例子中，我们把一个数值放入$var，然后把$var 与 Gm 用管道相连接。可以看到，Shell 把$var 的内容识别成 System.Int32，或一个 32 位数值。

18.3 使用变量：关于引号有趣的技巧

前面我们一直在讨论变量，是时候涵盖一个完整的 PowerShell 特性了。关于这一点，我们已经在书中建议过，使用单引号包住字符串。因为 PowerShell 会把所有包在单引号中的东西认为是一个文本字符串。如下面的例子。

```
PS C:\> $var = 'What does $var contain?'
PS C:\> $var
What does $var contain?
```

在前面的例子中可以看到，在单引号包含部分中的$var 被认为是一个文本字符。

但是在双引号中又是另外一番情景。看看下面的技巧。

```
PS C:\> $computername = 'SERVER-R2'
PS C:\> $phrase = "The computer name is $computername"
PS C:\> $phrase
The computer name is SERVER-R2
```

我们首先把“SERVER-R2”存入变量"$computername"。然后在变量$phrase 中存储“"The computer name is $computername"”，这里使用的是双引号。PowerShell 自动在双引号中搜索美元符，然后用变量的值替换所有被找到的变量。因为这里展示的是$phrase 的内容，所以$computername 变量被“SERVER-R2”替代。

这种替代操作仅发生在 Shell 初次解析字符串时。此时，$phrase 包含的是“The computer name is SERVER-R2”——它并没有包含“$computername”字符串。可以通过修改$computername 的内容检查$phrase 是否自己更新。

```
PS C:\> $computername = 'SERVER1'
PS C:\> $phrase
The computer name is SERVER-R2
```

可以看到，$phrase 变量依旧保存原有的值。

关于 PowerShell 双引号的另外一个窍门是转义字符。这个字符是重音符（`），在美式键盘左上角的部分，通常在 Esc 键的下方，与波浪符（~）在同一个键上。使用重音符的问题是，在某些字体中，很难区分单引号。实际上，我们常常使用 Consolas 字体，因为它与 Lucida Console 或 Raster 字体相比更容易区分重音符。

动手实验： 单击 PowerShell 窗口左上角的控件，选择属性。在【字体】标签页，选择图 18.1 所示的 Consolas 字体，再单击【OK】按钮。然后输入一个单引号和重音符看是否能区分它们。图 18.1 显示了在我们系统中的样子。你能从中看出区别吗？我相信，使用足够大的字体时是可以的。区分起来有点困难，所以请你选择合适的字体和大小，以便你可以轻易地区分出它们。

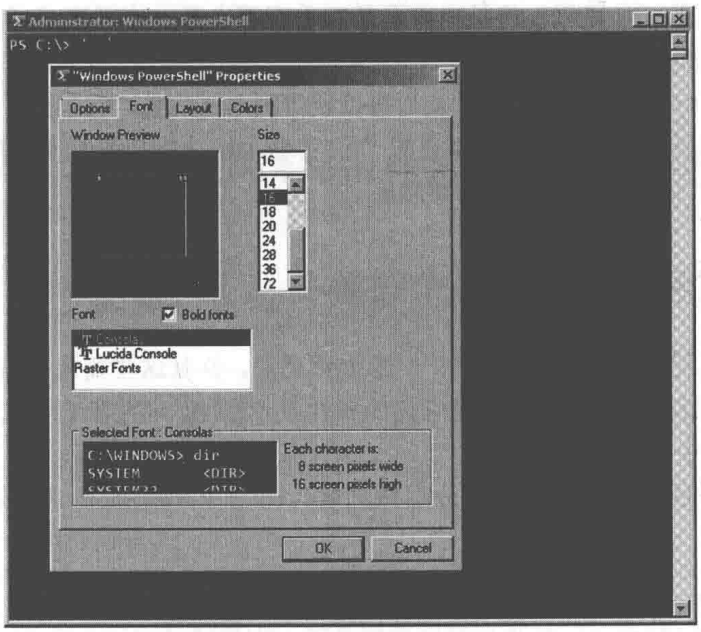

图 18.1 设置字体以便更容易区分单引号和重音符

下面来看看转义字符的作用。它消除了任何在转义符之后有特殊意义字符的含义，或在某些情况下增加了字符的特殊意义。下面示例展示消除特殊含义的用法。

```
PS C:\> $computername = 'SERVER-R2'
PS C:\> $phrase = "`$computername contains $computername"
PS C:\> $phrase
$computername contains SERVER-R2
```

当我们把字符串赋给$phrase 时，我们使用了两次$computername。第一次，我们在美元符前使用了重音符。这样去除了美元符在变量中的特殊意义，并把它当作字符中的美元符。从前面的输出中可以看出，在最后一行，$computername 是存储在变量中的。在第二次时，没有使用重音符，所以$computername 被变量值替换掉。

下面来看一个第二种使用重音符的例子。

```
PS C:\> $phrase = "`$computername`ncontains`n$computername"
PS C:\> $phrase
$computername
contains
SERVER-R2
```

仔细检查，你会发现我们在语句中使用了两次`n——一个在第一个$computername 后，另外一个在 contains 后。在该示例中，重音符的存在用于添加特殊功能。一般来说，“n”是一个字母，但是在前面带有重音符之后，它就变成了一个回车与换行符（n 是 new line 的意思）。

运行“help about_escape”可以获得更多的信息，它包含了其他关于特殊转义符的列表。你可以尝试使用转义后的“t”实现 tab 功能，或者使用转义后的“a”使机器发出响声（a 是 alert，警报的意思）。

18.4 在一个变量中存储多个对象

在此之前，我们都是针对单一对象介绍变量，并且这些变量都是简单的值。我们都是直接操作这些对象本身而不是它们的属性或者方法。现在我们尝试把一堆对象放入一个单一变量中。

其中一种方式是使用逗号分隔符列表，因为 PowerShell 认为这些列表是对象的集合。

```
PS C:\> $computers = 'SERVER-R2','SERVER1','localhost'
PS C:\> $computers
SERVER-R2
SERVER1
Localhost
```

请留心观察上面的例子，逗号是放在单引号之外。如果把这些逗号放在单引号之内，会变成一个包含逗号和 3 个计算机名称的单一对象。通过我们的方法，可以得到 3 个独立对象，它们的类型均为字符串类型。正如你所看到的，当我们检查变量的内容时，PowerShell 会把每个对象分别以单行展示。

18.4.1 与多值单一变量的单一对象交互

你可以在某一时刻访问多值单一变量（一个变量存储多个值）的独立元素，只需在中括号中指定你要访问的对象的索引号即可。该编号从 0 开始，第二个值的索引号为 1，以此类推。你还可以使用-1 这个索引号来访问对象的最后一个值，-2 为倒数第二个值，等。比如：

```
PS C:\> $computers[0]
SERVER-R2
PS C:\> $computers[1]
SERVER1
PS C:\> $computers[-1]
localhost
PS C:\> $computers[-2]
SERVER1
```

变量本身有一个属性可以查看其中包含多少个对象。

```
PS C:\> $computers.count
3
```

你同样可以访问变量内部对象的属性和方法，就像变量自身的属性和方法一样。首先，针对只有单一对象的变量。

```
PS C:\> $computername.length
9
PS C:\> $computername.toupper()
SERVER-R2
PS C:\> $computername.tolower()
server-r2
PS C:\> $computername.replace('R2','2008')
SERVER-2008
PS C:\> $computername
SERVER-R2
```

在前面的例子中，我们使用了本章前面创建的变量$computername。你是否还记得该变量包含了一个类型为 System.String 的对象，并且在 18.2 节中已经通过与 Gm 进行管道传输后得到关于这个类型的属性和方法的完整列表。在这里，我们使用了 Length、ToUpper()、ToLower()和 Replace()方法。在每一个例子中，即使 ToUpper()和 ToLower()都不要求括号中出现任何值，但是我们也要在方法名称之后使用括号。同时可以看到这些方法都没有修改变量中的任何事物——你可以在示例的最后一行发现这一点。取而代之的是，每个方法都在原有基础上创建了一个新的字符串结果，看上去就好像方法对原始字符串进行了修改。

18.4.2 与多值单一变量的多个对象交互

当一个变量包含了多个对象，处理步骤变得稍微有点麻烦。即使变量中的每个对象都像前面例子中的$computers 变量那样具有相同的类型，PowerShell v2 也不允许你同时针对多个对象调用一个方法或者访问一个属性。如果你非要尝试，会收到报错信息。

```
PS C:\> $computers.toupper()
Method invocation failed because [System.Object[]] doesn't contain a metho
d named 'toupper'.
At line:1 char:19
+ $computers.toupper <<<< ()
    + CategoryInfo          : InvalidOperation: (toupper:String) [], Runt
   imeException
    + FullyQualifiedErrorId : MethodNotFound
```

替代方案是，你必须指定变量中你期望操作的那个对象，然后访问它的属性或执行一个方法。

```
PS C:\> $computers[0].tolower()
server-r2
PS C:\> $computers[1].replace('SERVER','CLIENT')
CLIENT1
```

再次提醒，这些方法会产生新的字符串结果，而不会更改变量中的原有值。用下面的方式可以测试。

```
PS C:\> $computers
SERVER-R2
SERVER1
Localhost
```

如果你希望修改变量中的内容，该怎么办呢？你必须为现有对象赋予新值。

```
PS C:\> $computers[1] = $computers[1].replace('SERVER','CLIENT')
PS C:\> $computers
SERVER-R2
CLIENT1
Localhost
```

从例子中可以看出已修改变量中的第二个对象，而不是产生一个新的字符串。我们在这里提出的这个例子仅在安装了 PowerShell v2 的电脑上才有效；而这种行为已经在 v3 中得到改变，我们将会在后面介绍。

18.4.3 与多个对象交互的其他方式

我们将会介绍在包含多个对象的单个变量中与它们的属性和方法交互的两种选项。在前面的例子中，仅仅执行了变量中单个对象的方法。如果你想要变量中的每个对象都执行 ToLower()方法，并把结果存储回去，你可以像这样执行。

```
PS C:\> $computers = $computers | ForEach-Object { $_.ToLower() }
PS C:\> $computers
server-r2
client1
localhost
```

该示例稍微有些复杂，所以我们在图 18.2 中把它分解。首先，$computers =与管道相连，意味着管道的输出将会被存储在变量中。这些结果将会覆盖以前变量的所有值。

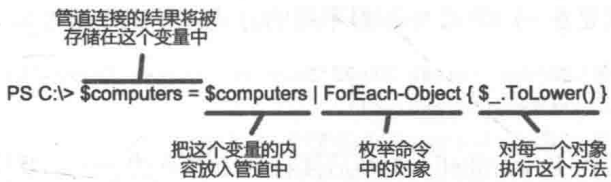

图 18.2 使用“ForEach-Object”方法执行在变量中包含的每个对象上

管道从$computers 开始，并传输到“ForEach-Object”。该 Cmdlet 会枚举管道中的所有对象（这里总共有 3 个计算机名并且是字符串对象），然后执行对应的代码块。在每个代码块中，$_占位符每次都包含一个被管道传进来的对象，然后针对每个对象执行 ToLower()方法。最后由 ToLower()产生的字符串对象会被放入管道——然后存入$computers 变量。

你可以使用“Select-Object”对属性做类似的事。该示例选择传输到该 cmdlet 每个对象的 length 属性。

```
PS C:\> $computers | select-object length

Length
------
     9
     7
     9
```

因为属性是数值型，所以 PowerShell 把输出以右对齐的方式展示。

18.4.4 在 PowerShell v3 中展现属性和方法

“当一个变量包含多个对象时，不能访问属性和方法”被证明会让 PowerShell v1 和 v2 用户非常困惑。因此，对于 v3 和后续版本，微软做出重要改变，该变更称之为

“automatic unrolling”。它本质上意味着你现在可以访问一个包含多个对象的变量的属性和方法。

```
$services = Get-Service
$services.Name
```

底层实现中，PowerShell 会意识到你正在尝试访问一个属性。同样，它也知道在 $services 集合中没有一个关于名称的属性——但是集合中的独立对象拥有该属性。所以它隐式枚举，或展现对象，并获取每个对象的名称属性。上面代码等价于：

```
Get-Service | ForEach-Object { Write-Output $_.Name }
```

也等价于：

```
Get-Service | Select-Object -ExpandProperty Name
```

这两种方式是在 v1 和 v2 中不得不用的方式，其工作原理也等于：

```
$objects = Get-WmiObject -class Win32_Service -filter "name='BITS'"
$objects.ChangeStartMode('Disabled')
```

记住，这是在 PowerShell v3 和后续特性中独有的——不要期望这种方式能在旧版本中有效。

18.5 双引号的其他技巧

对于双引号，还有一个很酷的技术可用，这个技巧是对变量替换概念的延伸。假设你把一堆服务存入$service 变量。现在你只想把第一个服务名称放入一个字符串。

```
PS C:\> $services = get-service
PS C:\> $firstname = "$services[0].name"
PS C:\> $firstname
AeLookupSvc ALG AllUserInstallAgent AppIDSvc Appinfo AppMgmt AudioEndpoint
Builder Audiosrv AxInstSV BDESVC BFE BITS BrokerInfrastructure Browser bth
serv CertPropSvc COMSysApp CryptSvc CscService DcomLaunch defragsvc Device
AssociationService DeviceInstall Dhcp Dnscache dot3svc DPS DsmSvc Eaphost
EFS ehRecvr ehSched EventLog EventSystem Fax fdPHost FDResPub fhsvc FontCa
che gpsvc hidserv hkmsvc HomeGroupListener HomeGroupProvider IKEEXT iphlps
vc KeyIso KtmRm LanmanServer LanmanWorkstation lltdsvc lmhosts LSM Mcx2Svc
 MMCSS MpsSvc MSDTC MSiSCSI msiserver napagent NcaSvc NcdAutoSetup Netlogo
n Netman netprofm NetTcpPortSharing NlaSvc nsi p2pimsvc p2psvc Parallels C
oherence Service Parallels Tools Service PcaSvc PeerDistSvc PerfHost pla P
lugPlay PNRPAutoReg PNRPsvc PolicyAgent Power PrintNotify ProfSvc QWAVE Ra
sAuto RasMan RemoteAccess RemoteRegistry RpcEptMapper RpcLocator RpcSs Sam
Ss SCardSvr Schedule SCPolicySvc SDRSVC seclogon SENS SensrSvc SessionEnv
```

```
SharedAccess ShellHWDetection SNMPTRAP Spooler sppsvc SSDPSRV SstpSvc stis
vc StorSvc svsvc swprv SysMain SystemEventsBroker TabletInputService TapiS
rv TermService Themes THREADORDER TimeBroker TrkWks TrustedInstaller UI0De
tect UmRdpService upnphost VaultSvc vds vmicheartbeat vmickvpexchange vmic
rdv vmicshutdown vmictimesync vmicvss VSS W32Time wbengine WbioSrvc Wcmsvc
 wcncsvc WcsPlugInService WdiServiceHost WdiSystemHost WdNisSvc WebClient
Wecsvc wercplsupport WerSvc WiaRpc WinDefend WinHttpAutoProxySvc Winmgmt W
inRM WlanSvc wlidsvc wmiApSrv WMPNetworkSvc WPCSvc WPDBusEnum wscsvc WSear
ch WSService wuauserv wudfsvc WwanSvc[0].name
```

出错了。例子中紧跟$services 的“[”符号不是常规文本字符，会引发 PowerShell 尝试替换$services。同时因为这种阻塞，字符串中的[0].name 部分完全没有被替换。

解决方法是将上述命令放入一个表达式。

```
PS C:\> $services = get-service
PS C:\> $firstname = "The first name is $($services[0].name)"
PS C:\> $firstname
The first name is AeLookupSvc
```

在$()中的所有内容都会被当成普通的 PowerShell 命令，结果也被放入字符串中，替代原有的所有内容。同样，该操作仅在双引号中有效。这种$()结构称为子表达式。

另外，在 PowerShell v3 及后续版本中还有一个很酷的功能。有时候，你需要把更复杂的内容放入一个变量，然后在引号中显示变量的内容。在 PowerShell v3 及后续版本中，Shell 能更智能地枚举集合中的所有对象。即使你仅引用一个属性或方法，作用域集合中所有相同类型的对象中也没问题。比如，我们查询服务的清单并把它们放入$service 变量中，然后使用双引号仅包含服务名称。

```
PS C:\> $services = get-service
PS C:\> $var = "Service names are $services.name"
PS C:\> $var
Service names are AeLookupSvc ALG AllUserInstallAgent AppIDSvc Appinfo App
Mgmt AudioEndpointBuilder Audiosrv AxInstSV BDESVC BFE BITS BrokerInfrastr
ucture Browser bthserv CertPropSvc COMSysApp CryptSvc CscService DcomLaunc
h defragsvc DeviceAssociationService DeviceInstall Dhcp Dnscache dot3svc D
PS DsmSvc Eaphost EFS ehRecvr ehSched EventLog EventSystem Fax fdPHost FDR
esPub fhsvc FontCache FontCache3.0.0.0 gpsvc hidserv hkmsvc HomeGroupListe
ner HomeGroupProvider IKEEXT iphlpsvc KeyIso KtmRm LanmanServer LanmanWork
station lltdsvc lmhosts LSM Mcx2Svc MMCSS MpsSvc MSDTC MSiSCSI msiserver M
SSQL$SQLEXPRESS napagent NcaSvc NcdAutoSetup Netlogon Netman netprofm NetT
cpPortSharing NlaSvc nsi p2pimsvc p2psvc Parallels Coherence Service Paral
lels Tools Service PcaSvc PeerDistSvc PerfHost pla PlugPlay PNRPAutoReg PN
RPsvc PolicyAgent Power PrintNotify ProfSvc QWAVE RasAuto RasMan RemoteAcc
ess RemoteRegistry RpcEptMapper RpcLocator RpcSs SamSs SCardSvr Schedule S
CPolicySvc SDRSVC seclogon SENS SensrSvc SessionEnv SharedAccess ShellHWDe
```

这里截断了一部分输出结果以便节省空间，但是我们希望你能理解这种思想。显然，这些可能并不是你希望查询的结果。但是从前面提到的子表达式和这里的例子中，你应该能得到一些启示。

18.6 声明变量类型

目前为止，我们仅仅把对象存入变量并让 PowerShell 指出我们正在使用对象的类型。这是由于 PowerShell 不在乎你放入变量中的对象是什么类型，但是我们在意。

比如，假设你有一个变量希望用于存储一个数值，准备用于一些算术运算，并期待用户输入一个数值。请看下面的例子，你可以直接在命令行中输入数值。

```
PS C:\> $number = Read-Host "Enter a number"
Enter a number: 100
PS C:\> $number = $number * 10
PS C:\> $number
100100100100100100100100100100
```

动手实验：目前为止，我们没有提到“Read-Host”——我们将把它放到下一章介绍——但是如果你跟着做实验，它所实现的功能显而易见。

见鬼，为什么 100 乘以 10 会得出 100100100100100100100100100100？这是什么数字？

如果你观察力敏锐，你可以发现，PowerShell 并没有把我们的输入当作数值，而是把它当作字符串。PowerShell 只是把 100 这个字符串重复了 10 次，而不是把 100 乘以 10。所以结果就是把字符串 100 在一行中列了 10 次。

我们可以用下面的方式验证。

```
PS C:\> $number = Read-Host "Enter a number"
Enter a number: 100
PS C:\> $number | gm

   TypeName: System.String
Name             MemberType           Definition
----             ----------           ----------
Clone            Method               System.Object Clone()
CompareTo        Method               int CompareTo(System.Object valu...
Contains         Method               bool Contains(string value)
```

通过把$number 用管道传输到 Gm 中，可以看出 Shell 把它视为 System.String，而不是 System.Int32。对于该问题有很多解决方法，我们将介绍其中最简单的一种。

首先，告诉 Shell 知道$number 变量应该存储一个整型，强制 Shell 把值转换成一个整型。如下面的例子，通过在变量首次使用前使用[]，明确定义一个数据类型“int”实现。

```
PS C:\> [int]$number = Read-Host "Enter a number"
Enter a number: 100
PS C:\> $number | gm

   TypeName: System.Int32
Name        MemberType Definition
----        ---------- ----------
CompareTo   Method     int CompareTo(System.Object value), int CompareT...
Equals      Method     bool Equals(System.Object obj), bool Equals(int ...
GetHashCode Method     int GetHashCode()
GetType     Method     type GetType()
GetTypeCode Method     System.TypeCode GetTypeCode()
ToString    Method     string ToString(), string ToString(string format...
PS C:\> $number = $number * 10
PS C:\> $number
1000
```

强制类型

❶ 转换成[int]

确认变量的

❷ 数据类型是 Int32

❸ 确认变量的

数据类型是 Int32

在前面的例子中，我们使用了[int]强制$number 仅包含整数❶。在你输入以后，我们把$number 用管道传输到 Gm，验证它的确已经是整型而不是字符串❷。最后我们可以看到，变量的值被认为是数值型并进行了实际乘法运算❸。

使用该技术的另外一个优势是，在 Shell 无法把数据的值转换成数字时，使得 Shell 可以抛出错误，因为$number 仅仅是存储数值的一个容器。

```
PS C:\> [int]$number = Read-Host "Enter a number"
Enter a number: Hello
Cannot convert value "Hello" to type "System.Int32". Error: "Input string
was not in a correct format."
At line:1 char:13
+ [int]$number <<<<  = Read-Host "Enter a number"
    + CategoryInfo          : MetadataError: (:) [], ArgumentTransformati
   onMetadataException
    + FullyQualifiedErrorId : RuntimeException
```

这是一个防止后续问题的例子，因为你可以确保$number 能存储你希望的值。

除了[int]之外，还有很多其他的选择。下面是最常用的一些类型清单。

- [int]——整型数字。
- [single]和[double]——单精度和多精度浮点型数值（小数位部分的数值）。
- [string]——字符串。
- [char]——仅单个字符（如[char]$c='X'）。
- [xml]——一个 XML 文档。不管你如何解析里面的值，都要确保它包含有效的 XML 标记（比如[xml]$doc=Get-Content MyXML.xml）。

- [adsi]——一个活动目录服务接口（ADSI）查询。Shell 会执行查询并把结果对象存入变量(如[adsi]$user="WinNT:\\MYDOMAIN\Administrator, user")。

明确指定变量的对象类型，可以避免在复杂脚本中出现一些严重的逻辑错误。正如下面的示例所示，一旦你指定了对象类型，PowerShell 会强制它使用该类型，直到重新显式定义变量的类型。

```
PS C:\> [int]$x = 5                                    ❶ 定义变量$x 为整型
PS C:\> $x = 'Hello'
Cannot convert value "Hello" to type "System.Int32". Error: "Input string
was not in a correct format."
At line:1 char:3                                       创建一个错误，并把
+ $x <<<< = 'Hello'                                    错误信息放到$x 中 ❷
    + CategoryInfo          : MetadataError: (:) [], ArgumentTransformati
   onMetadataException
    + FullyQualifiedErrorId : RuntimeException
PS C:\> [string]$x = 'Hello'                           ❸以字符形式
                                                       重新对$x 赋值
PS C:\> $x | gm
   TypeName: System.String                             ❹ 确认$x 的新类型

Name             MemberType          Definition
----             ----------          ----------
Clone            Method              System.Object Clone()
CompareTo        Method              int CompareTo(System.Object valu...
```

在前面的例子中，可以看到，我们首先声明$x 变量为整型❶，并把一个整型值放入变量。当我们准备把一个字符串放入变量时❷，PowerShell 抛出错误，因为它不能把字符串转换成整型数值。在后续把变量类型重新声明为字符串后，就可以把字符串放入其中❸。通过管道把变量传输到 Gm，可以查看变量的类型名称❹。

18.7 与变量相关的命令

我们虽然使用了变量，但是目前为止还没有正式地表明我们的意图。PowerShell 不建议使用高级的变量声明，并且你不能强制声明。（试图去搜寻类似 Option Explicit 的 VBScript 使用者可能会感到沮丧，PowerShell 有类似的 Set-StrictMode，但是并不完全一样。）但是 Shell 却包含了下面与变量有关的命令。

- New-Variable
- Set-Variable
- Remove-Variable
- Get-Variable
- Clear-Variable

除了“Remove-Variable”之外，其他命令可能都用不上。该命令对需要删除的变量很有用（你也可以在变量中使用 Del 命令，使其删除该变量）。你可以使用其他功能——创建新的变量、读取变量和配置变量——如使用本章提到过的即席语法（ad hoc syntax）；在大部分情况下，使用这些 Cmdlets 并没有什么特殊的优点。

如果你真的决定使用这些 Cmdlets，需要把变量名称授予对应 Cmdlets 的-name 参数。这里仅需要变量名称——不需要包含美元符。通常只有在操作超出作用域（out-of-scope）变量时，才可能用到这些 Cmdlets。使用这种变量不是好习惯，所以本书不打算讲述该类变量，但是可以使用“help about_scope”来获取更详细的信息。

18.8 针对变量的最佳实践

虽然我们前面已经提到过绝大部分的最佳实践，但是还是有必要做一个快速回顾。

- 确保变量名称有意义，但也要简洁。比如$computername 是一个很好的变量名称，因为它清晰、简短，$c 就不是，因为它不具有什么实际意义。变量名称 $computer_to_query_for_data 略微长了点儿。虽然它也有意义，但是你希望反反复复地输入它吗？
- 不要在变量中使用空格。虽然你可以这样做，但是这种语法相当不好。
- 如果变量仅包含一类对象，那么在你首次使用变量时，请定义对象类型。这样可以帮助你避免一些常见的逻辑错误，并且当你在商用脚本开发环境中工作时（PrimalScript 也许就是其中一个例子），编辑软件可以在你告诉它变量将包含的对象类型时提供一些提示功能。

18.9 常见误区

对于初学者来说，最常见的误区是变量名称。我希望在这一章中已经说得很清楚，但是请记住，美元符并不是变量名称的一部分。它只是让 Shell 知道你想访问变量的内容，而美元符后面的才是变量名称本身。

Shell 有两个用于获取变量名称的解析规则。

- 如果紧随美元符后的字符是一个字母、数字或下画线，则变量名称包含美元符到下一个空白的所有字符（可能是一个空格、Tab 或回车）。
- 如果紧随美元符后的是一个左大括号，则变量名称包含左大括号开始但不包含右大括号之间的所有内容。

18.10 动手实验

注意： 对于本次动手实验来说，你需要运行 PowerShell v3 或更新版本 PowerShell 的计算机。

回到第 15 章，释放后台作业的内存，然后在命令行中执行下面的操作。

1. 创建一个后台作业，从两台计算机中查询 Win32_BIOS 信息（如果你只有一台计算机做实验，可以使用两次“localhost”模拟）。
2. 当作业运行完毕后，把作业的结果存入一个变量。
3. 显示变量的内容。
4. 把变量内容导出到一个 CliXML 文件中。

18.11 进一步学习

花点时间浏览一下本书的前面章节。设计变量的目的是存储一些你可能需要反复使用的数据。你可以在前面章节中找到变量的用处吗？

比如，在第 13 章中，你已经学到创建一个远程计算机的链接。你在本章中学到的是如何在一个步骤中创建、使用和关闭链接。它不正是在几个命令中创建连接，存入到变量中吗？那只是其中一个使用变量的例子（我们将在第 20 章介绍）。看看你能否找到更多的例子。

18.12 动手实验答案

```
1. PS C:\> invoke-command {get-wmiobject win32_bios} -computername
   localhost,$env:computername -asjob
2. PS C:\>$results=Receive-Job 4 -keep
3. PS C:\>$results
4. PS C:\>$results | export-clixml bios.xml
```