

第 21 章 你把这叫作脚本

目前为止，你已经可以通过 PowerShell 的命令行界面完成本书中的所有内容。但你仍然没有写过一行脚本。这对我们来说是很大的问题。这是因为我们见过很多管理员害怕写脚本，认为写脚本是一种编程方式并觉得学习写脚本得不偿失。所幸，你已经看到在不成为程序员的前提下使用 PowerShell 所能完成的工作。

但在此刻，你可能还会感觉不断重复输入同样的命令是一件非常枯燥的事情。你是对的，所以在本章我们将会深入 PowerShell 脚本——当然，你仍然无须成为程序员。脚本的作用仅仅是为了减少不必要的重复输入。

21.1 非编程，而更像是批处理文件

大多数 Windows 管理员曾经或是时不时地创建一个命令行批处理文件（通常以 .BAT 或 .CMD 作为文件扩展名）。该文件本质上不过是一个简单的、可以用 Windows 记事本编辑的文本文件，该文件包含按照指定顺序排列的可执行命令列表。从技术上讲，你把这些命令叫作脚本，就像好莱坞电影的剧本那样用于告诉演员（你的计算机）该如何按照顺序说台词和表演。但批处理文件看上去并不像是编程语言，这部分是由于 cmd.exe Shell 语言本身过于简单，难以编写非常复杂的脚本。

PowerShell 脚本——如果你愿意或者也可以称之为批处理文件——以类似的原理工作。仅仅是将你希望运行的命令列出来，Shell 将会以指定的顺序执行这些命令。你可以通过将命令从宿主窗口中复制到文本文件中来创建一个脚本。当然，记事本是一个非常不好用的文本编辑器。我们希望你更倾向使用 PowerShell ISE，或者诸如 PowerGUI、PrimalScript 或 PowerShell Plus 之类的第三方编辑器。

ISE 实际上使用起来和使用交互性 Shell 并无不同。当使用 ISE 的脚本编辑器窗口时，

只需输入命令或希望运行的命令，并单击在工具栏中的“运行”按钮执行这些命令。单击“保存”按钮，你将可以在不复制粘贴任何命令的情况下创建一个脚本。

21.2 使得命令可重复执行

PowerShell 脚本背后的理念，首先是使得重复执行特定命令变得简单，而无须每次手动重复输入命令。既然如此，我们需要想出一个你能够一遍遍重复执行的命令，并使用该示例贯穿本章。我们希望该示例有合适的复杂度，所以我们以 WMI 开始并添加一些筛选条件、排序规则以及其他内容。

此时，我们需要转换使用 PowerShell ISE 而不是标准的控制台窗口。这是由于通过 ISE 将我们的命令转为一个脚本变得更加容易。坦白讲，ISE 使得输入复杂命令变得更加容易。这是因为可以使用全屏的编辑器而不是在控制台宿主上输入单行命令。

下面是我们的命令。

```
Get-WmiObject -class Win32_LogicalDisk -computername localhost  
-filter "drivetype=3" |  
Sort-Object -property DeviceID |  
Format-Table -property DeviceID,  
@{label='FreeSpace(MB)';expression={$_.FreeSpace / 1MB -as [int]}},  
@{label='Size(GB)';expression={$_.Size / 1GB -as [int]}},  
@{label='%Free';expression={$_.FreeSpace / $_.Size * 100 -as [int]}}
```

提示：请记住，你可以使用 **name** 而不是 **label**，这两个属性都可以简写为 **n** 或 **l**。但 **L** 的小写形式看上去非常像数字 1，所以请小心。

图 21.1 展示了我们如何在 ISE 中输入该命令。注意，我们通过在工具栏按钮距离左边很远的“在顶部显示脚本窗格”按钮选择了双窗格布局。另外注意，我们将命令格式化为每一个物理行以逗号或管道操作符结尾。这么做可以让 Shell 识别这个多行脚本是一个单个、单行的命令。你也可以在控制台宿主中这么做，但这种格式由于具有更好的可读性，因此在 ISE 中尤其有效。另外注意，我们使用的是完整 **Cmdlet** 名称和参数名称并显式指定了参数名称，而不是使用位置参数。上面我们所做的一切都是为了使脚本具有更好的可读性，以便其他人很快可以接手。此外，当我们未来忘了当初脚本的意图时，可以很快想起来。

我们通过单击在工具栏的绿色运行按钮运行命令（也可以按快捷键 F5），对命令进行测试，输出结果显示命令正常工作。下面是在 ISE 中一个巧妙的技巧：你可以选中命令的一部分并按 F8 键，从而只运行选中部分的命令。由于我们已经格式化了命令，因此每一个物理行只有一个单独命令，这使得分步测试命令变得更加容易。我们可以选中并单独运行第一行命令。如果输出结果符合预期，我们可以选中第一行和第二行命令并运行。如果这部分也能正常工作，那么我们就可以运行整个命令。

此时，我们就可以保存命令——现在就可以把保存后的命令称为脚本。我们可以将其另存为 `Get-DiskInventory.ps1`。我们以“动词-名词”这样的 Cmdlet 风格名称命名该脚本。你可以看到该脚本是如何开始像 Cmdlet 一样工作的，这也是使用 Cmdlet 风格名称的原因。

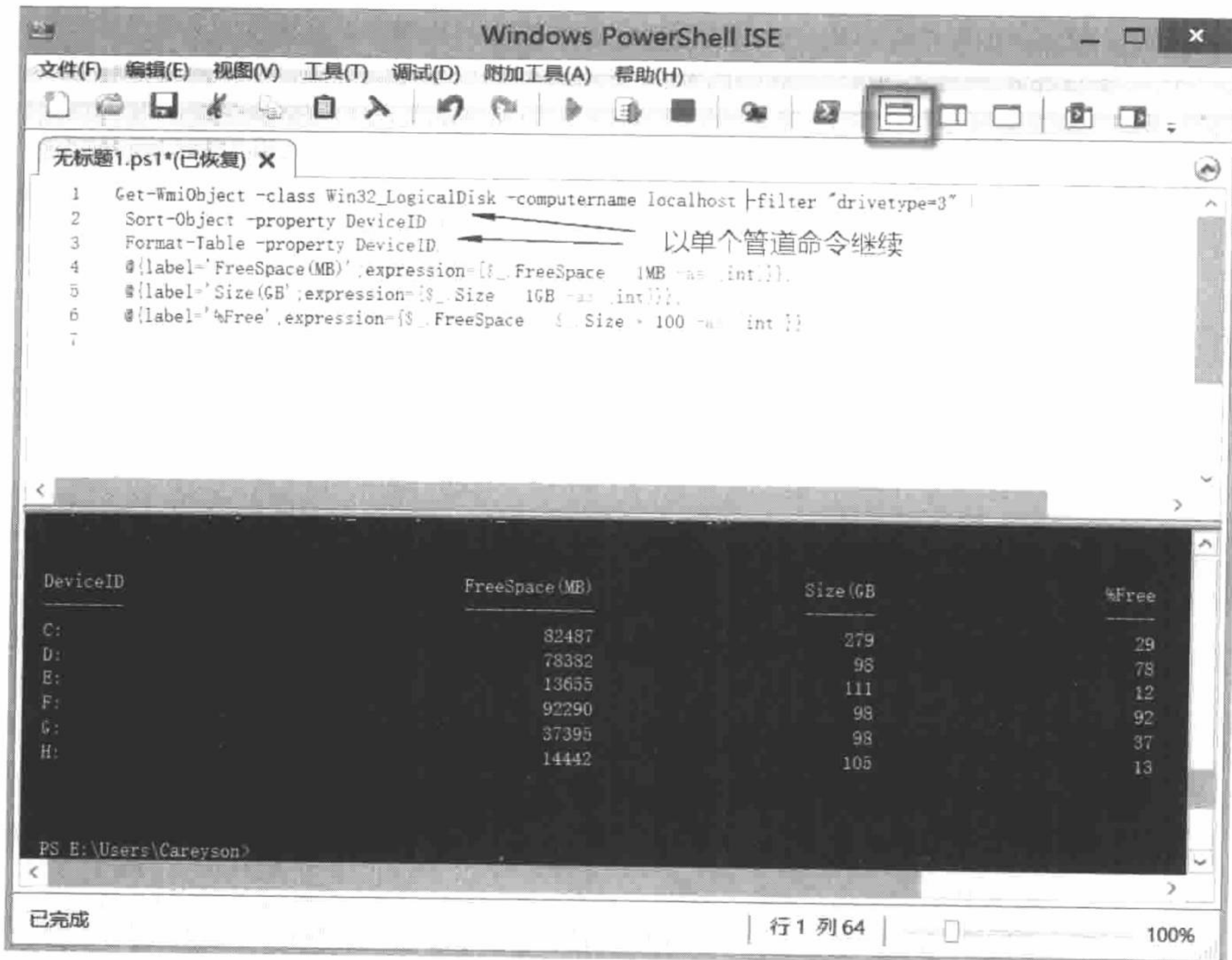


图 21.1 在 ISE 中输入并运行一个命令

动手实验：我们假设你已经完成了第 14 章并设置了更加自由的执行策略。如果你还未这么做，那么请返回第 17 章完成其动手实验部分，这样该脚本就可以在你的 PowerShell 副本下运行。

21.3 参数化命令

当你考虑到一遍遍运行同一个命令时，你或许会意识到命令的某些部分在每次运行时都可能产生变化。例如，假设你将 `Get-DiskInventory.ps1` 脚本给了一个缺乏 PowerShell 使用经验的同事。该脚本是一个比较复杂且难以输入的命令，你的同事非常感激你将其封装为一个易于运行的脚本。但是，作为该脚本作者，你发现该脚本只能够在本地计算机上运行。你当然


可以想象得出，你的一些同事或许希望从一台或多台远程计算机上获取磁盘信息。

一个可能的解决方案是让他们打开脚本，并修改 `-computer-name` 参数值。但这个操作可能对它们来说有点难度，且修改脚本可能导致改错地方从而破坏脚本。因此为他们提供一个标准方法，使得他们可以传入不同的计算机名称（或名称集合）将是一种更好的方式。在此阶段，你需要识别出命令执行时可能需要变更的部分，并用变量替换这部分。

既然我们仍然处于测试脚本阶段，我们暂时将计算机名称变量设置为静态值。下面是修改后的脚本。

代码清单 21.1 Get-DiskInventory.ps1，包含一个参数的命令

```
$computername = 'localhost'
Get-WmiObject -class Win32_LogicalDisk `
  -computername $computername `
  -filter "drivetype=3" |
Sort-Object -property DeviceID |
Format-Table -property DeviceID,
  @{label='FreeSpace(MB)';expression={$_.FreeSpace / 1MB -as [int]}},
  @{label='Size(GB)';expression={$_.Size / 1GB -as [int]}},
  @{label='%Free';expression={$_.FreeSpace / $_.Size * 100 -as [int]}}
```



1 设置新的变量
2 使用反撇号分隔行
3 使用变量

我们在此完成了三件事，其中两件关于功能，另一件是格式美化。

- 我们添加了一个变量 `$computername`，将其值设置为 `localhost` ①。我们注意到，大多数 PowerShell 命令使用名称为 `-computerName` 的参数接受计算机名称。我们希望保留这种传统，这也是为什么我们将变量命名为 `$computername`。
- 我们将 `-computerName` 参数值替换为我们定义的变量 ③。当前，该脚本和之前的脚本功能完全一样（并且经过测试的确一样），这是由于我们已经将 `localhost` 值赋予 `$computerName` 变量。
- 我们在 `-computerName` 参数和其值后面添加了反撇号 ②。这是转义符号，该符号用于告诉 PowerShell 下一个物理行是之前命令的一部分。当行以管道操作符或逗号结尾时无须使用转义符号，但需要按照本书的代码结构组织代码。这里我们需要在管道操作符之前分隔行，因此只能在行末尾使用反撇号。

我们再次仔细检查并运行脚本，从而确保脚本仍然可以正确工作。在每次对脚本进行任何变更时，我们总是会这么做，以便确保没有引入新的误输入或其他错误。

21.4 创建一个带参数的脚本

既然我们已经识别出了脚本中每次执行可能变化的部分，那么我们就需要提供一种让其他人赋予这些元素新值的方式。换句话说，我们需要将被赋予常量的 `$computername` 变量转变为一个输入参数。

PowerShell 中创建一个带参数的脚本非常简单。

代码清单 21.2 Get-DiskInventory.ps1, 包含一个输入参数

```
param (
    $computername = 'localhost'
)
Get-WmiObject -class Win32_LogicalDisk -computername $computername `
    -filter "drivetype=3" |
Sort-Object -property DeviceID |
Format-Table -property DeviceID,
    @{label='FreeSpace(MB)';expression={$_.FreeSpace / 1MB -as [int]}},
    @{label='Size(GB)';expression={$_.Size / 1GB -as [int]}},
    @{label='%Free';expression={$_.FreeSpace / $_.Size * 100 -as [int]}}
```

← ❶ 参数块

我们只需要在变量声明代码附近添加一个 Param() 块 ❶。这会将 \$computerName 定义为一个参数，并在未对该参数赋值时指定 localhost 作为默认值。你可以不提供默认值，但我们能想到一个合适的值作为默认值时，我们更倾向这么做。

所有以这种方式定义的参数是命名参数，也是位置参数。这意味着我们可以用以下任意一种方式调用该脚本。

```
PS C:\> .\Get-DiskInventory.ps1 server-r2
PS C:\> .\Get-DiskInventory.ps1 -computername server-r2
PS C:\> .\Get-DiskInventory.ps1 -comp server-r2
```

在第一个实例中，我们以位置参数的形式调用该脚本，只提供参数值而不指定参数名称。在第 2、3 个实例中，我们指定参数名称，但在第 3 个实例中，我们将参数名称简化为符合 PowerShell 的参数名称简化规则的形式。注意，在上面三个示例中，我们都需要为脚本指定路径（.\，也就是当前目录），这是由于 Shell 并不会搜索当前目录来找到脚本。

你可以通过逗号作为分隔符指定任意数量的参数。例如，假如我们还希望将过滤条件设置为参数。当前脚本仅获取类型为 3 的驱动器，也就是硬盘。我们可以将该值变为参数，如代码清单 21.3 所示。

代码清单 21.3 Get-DiskInventory.ps1, 包含一个额外参数

```
param (
    $computername = 'localhost',
    $drivetype = 3
)
Get-WmiObject -class Win32_LogicalDisk -computername $computername `
    -filter "drivetype=$drivetype" |
Sort-Object -property DeviceID |
Format-Table -property DeviceID,
    @{label='FreeSpace(MB)';expression={$_.FreeSpace / 1MB -as [int]}},
    @{label='Size(GB)';expression={$_.Size / 1GB -as [int]}},
    @{label='%Free';expression={$_.FreeSpace / $_.Size * 100 -as [int]}}
```

指定额外参数

使用参数

注意，我们利用了 PowerShell 中在双引号中的文本可以自动将变量替换为变量值的功能（你已经在第 18 章中学到了这个技巧）。

我们可以以最开始的三种方式运行该脚本。当然，我们也可以通过忽略参数的方式使用参数的默认值。下面是一些该脚本的使用示例。

```
PS C:\> .\Get-DiskInventory.ps1 server-r2 3
PS C:\> .\Get-DiskInventory.ps1 -comp server-r2 -drive 3
PS C:\> .\Get-DiskInventory.ps1 server-r2
PS C:\> .\Get-DiskInventory.ps1 -drive 3
```

在第一个示例中，对于两个参数，我们都按照它们在 `Param()` 代码块中声明的顺序作为位置参数使用。在第二个示例中，我们对两个参数名称都进行了简化。在第三个示例中，我们完全忽略了 `-drivetype` 参数，从而使用该参数的默认值 3。在最后一个实例中，我们忽略了 `-computerName`，使用该参数的默认值 `localhost`。

21.5 为脚本添加文档

只有真正吝啬的人才会创建一个有用的脚本，而不告诉任何人如何使用它。幸运的是，PowerShell 提供了简单的方式为脚本添加帮助，也就是通过注释。你当然可以为你的脚本添加典型编程风格的注释，但如果你已经在脚本中使用了完整的 `Cmdlet` 名称和参数名称，很多时候你的脚本的意图已经足够可以望文生义。通过使用特殊的注释语法，你可以提供模仿 PowerShell 本身帮助文件的帮助信息。

代码清单 21.4 展示了我们为脚本添加的内容。

代码清单 21.4 为 Get-DiskInventory.ps1 添加帮助

```
<#
.SYNOPSIS
Get-DiskInventory retrieves logical disk information from one or
more computers.
.DESCRIPTION
Get-DiskInventory uses WMI to retrieve the Win32_LogicalDisk
instances from one or more computers. It displays each disk's
drive letter, free space, total size, and percentage of free
space.
.PARAMETER computername
The computer name, or names, to query. Default: Localhost.
.PARAMETER drivetype
The drive type to query. See Win32_LogicalDisk documentation
for values. 3 is a fixed disk, and is the default.
.EXAMPLE
Get-DiskInventory -computername SERVER-R2 -drivetype 3
#>

param (
    $computername = 'localhost',
    $drivetype = 3
```



```

)
Get-WmiObject -class Win32_LogicalDisk -computername $computername
  -filter "drivetype=$drivetype" |
Sort-Object -property DeviceID |
Format-Table -property DeviceID,
  @{label='FreeSpace(MB)';expression={$_.FreeSpace / 1MB -as [int]}},
  @{label='Size(GB)';expression={$_.Size / 1GB -as [int]}},
  @{label='%Free';expression={$_.FreeSpace / $_.Size * 100 -as [int]}}

```

正常情况下，PowerShell 都会忽略以#开头的代码行，意味着#用于标识某一行是注释。而我们使用<# #>块注释语法，这是由于我们需要注释多行而不希望在每一行开始都使用#。

现在我们可以使用标准的控制台宿主，并通过运行 `Help .\Get-DiskInventory` 命令获取帮助。（再一次，我们需要提供路径，这是由于该脚本并不是一个内置 Cmdlet。）图 21.2 显示了该命令的输出结果，证明了 PowerShell 读取并根据这些注释创建了标准的帮助显示界面。我们甚至可以运行 `help .\Get-DiskInventory -full` 来获取完整的帮助，其中包括了参数信息和示例。图 21.3 显示了该结果。

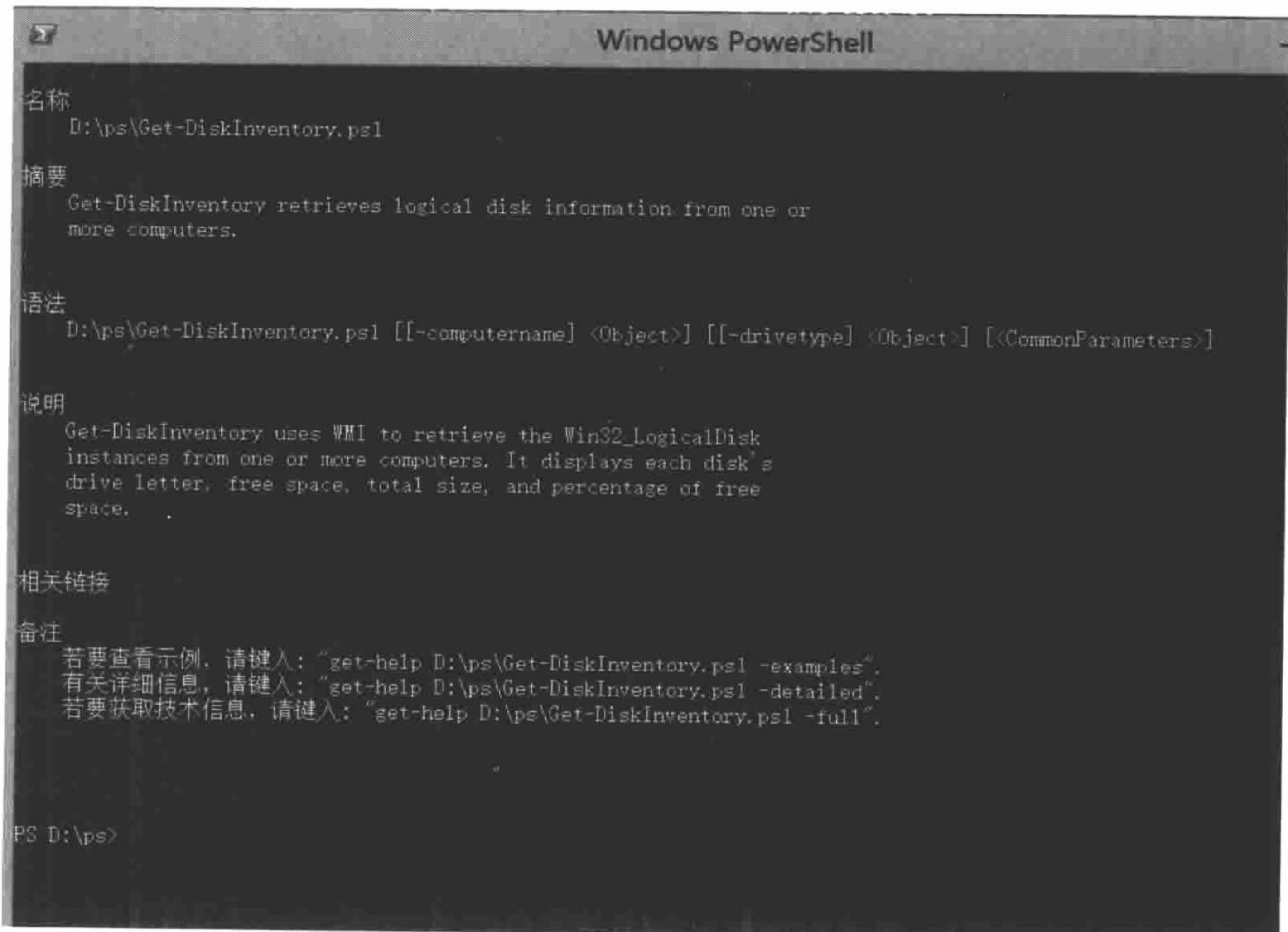


图 21.2 通过标准的帮助命令查看帮助

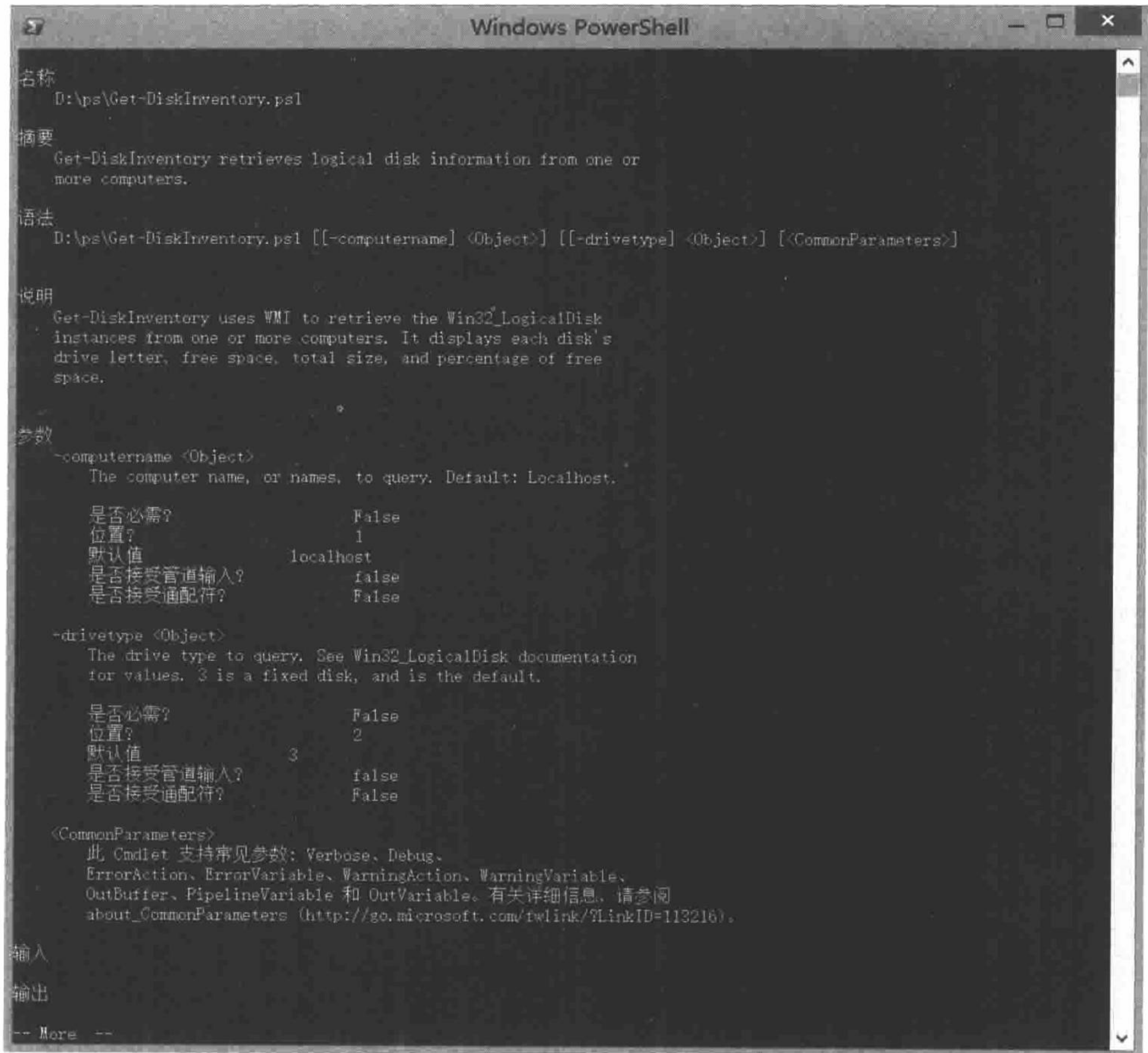


图 21.3 基于支持诸如-example、-detailed 以及-full 的帮助选项

这些特殊的注释被称为基于注释的帮助, 必须置于脚本文件的开始部分。除了我们使用的.DESCRPTION 和.SYNOPSIS 关键字之外, 还有一些关键字。在 PowerShell 中运行 `help about_comment_based_help` 查看完整的列表。

21.6 一个脚本, 一个管道

我们通常会告诉人们脚本中包含的任何代码和手动输入 PowerShell 的代码, 或是将

脚本中的代码通过剪贴板粘贴到 Shell 中的代码，运行起来并无不同。

但这并不完全正确。

请考虑下面的简单脚本。

```
Get-Process  
Get-Service
```

仅仅是两个命令，但如果我们将这两个命令手动复制到 Shell 中，每个命令后按回车键执行会发生什么？

动手实验：你需要自己尝试运行这些命令查看结果；该命令的输出结果过长，以致难以将结果甚至结果截图放入书中。

当你分别运行命令时，你会为每一个命令创建一个新的管道。在每一个管道末尾，PowerShell 会查看哪一列需要被格式化并创建一个你可以看到的表格。这里的重点是“不同命令运行在不同管道中”。图 21.4 阐述了这一点：两个完全分开的命令，两个独立的管道，两个格式化进程，两个不同界面的结果集。

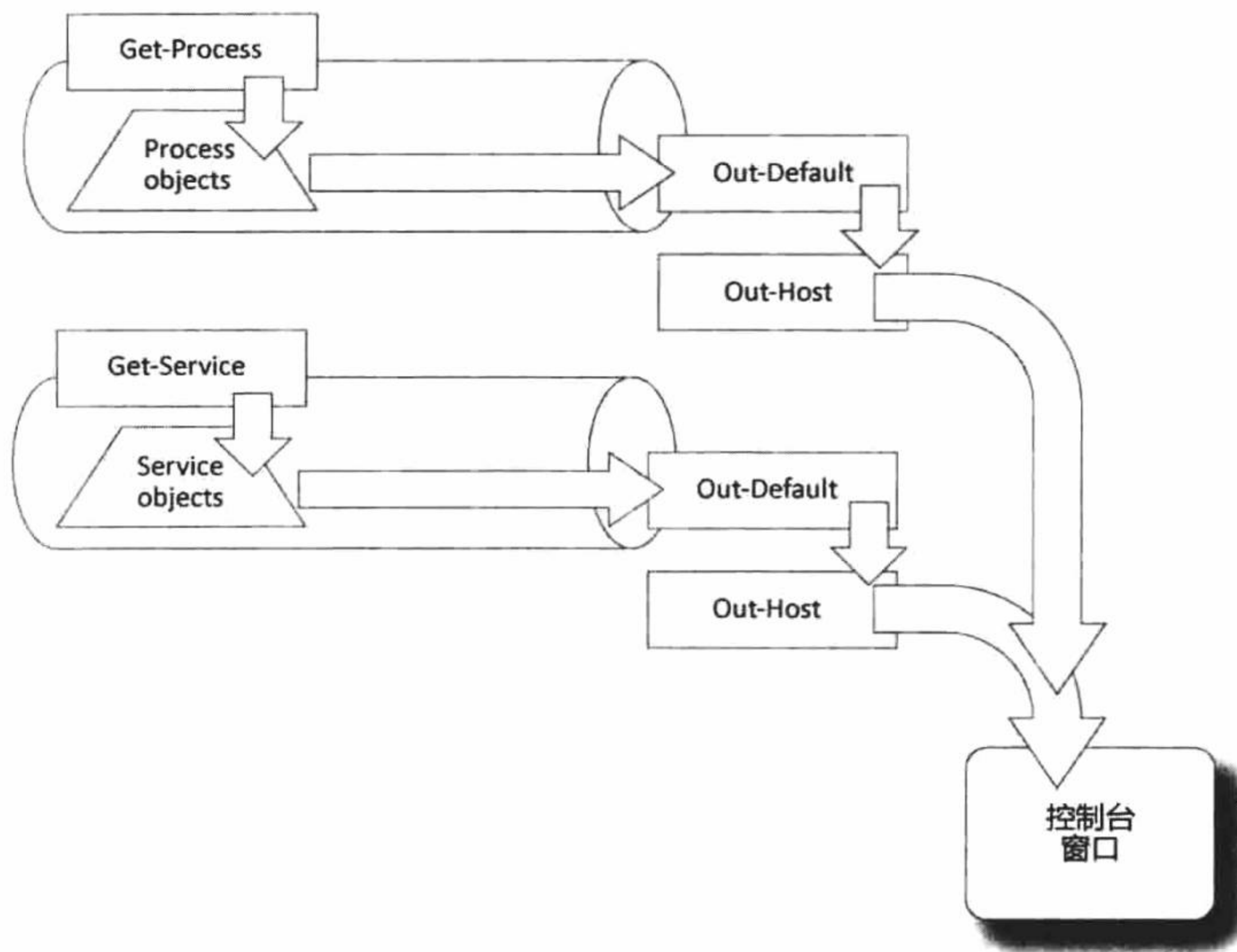


图 21.4 两个命令、两个管道、在同一个控制台窗口中的两个输出结果集

你或许会认为我们用了大量篇幅介绍显而易见的内容有些大题小做，但这很重要。下面是分别运行这两个命令经历的步骤：

- (1) 运行 `Get-Process`;
- (2) 该命令将 `Process` 对象放入管道;

- (3) 管道以 Out-Default 结束，该命令会接收对象；
- (4) Out-Default 将对象传递给 Out-Host，该命令会调用格式化系统产生文本输出结果（你在第 10 章学到过这些）；
- (5) 文本输出结果显示在屏幕上；
- (6) 运行 Get-Service；
- (7) 该命令将 Service 对象放入管道；
- (8) 管道以 Out-Default 结束，该命令会接收对象；
- (9) Out-Default 将对象传递给 Out-Host，该命令会调用格式化系统产生文本输出结果；
- (10) 文本输出结果显示在屏幕上。

所以你现在看到屏幕包含了来自两个命令的结果。我们希望你将这两个命令放入脚本文件，并命名为 Test.ps1 或其他简单的名称。在运行脚本之前，将这两个命令复制到剪贴板，你可以选中这两行并按 Ctrl+C 组合键将其复制到剪贴板。

转到 PowerShell 控制台宿主并按下回车键。这会将剪贴板中的命令粘贴到 Shell 中。在 Shell 中执行的方式会和 ISE 中完全一致，这是由于回车也会被粘贴进来。再一次，你在两个管道中运行不同的命令。

现在回到 ISE 中并运行脚本，结果不同，对吧？这是什么原因？

在 PowerShell 中，所有的命令都在一个管道中执行，在脚本中也是同样。在脚本中，任何产生管道输出结果的命令都会被写入同一个管道中：脚本自身运行的管道。请查看图 21.5。

我们尝试解释发生了什么：

- (1) 脚本运行 Get-Process。
- (2) 该命令将 Process 对象放入管道。
- (3) 脚本运行 Get-Service。
- (4) 该命令将 Service 对象放入管道。
- (5) 管道以 Out-Default 结束，该命令会接收上面两类对象。
- (6) Out-Default 将对象传递给 Out-Host，该命令会调用格式化系统产生文本输出结果。
- (7) 由于 Process 对象首先被放入管道，Shell 的格式化系统会为 Process 对象选择合适的格式化方式。这也是为什么 Process 对象的输出结果看起来很正常。当 Shell 碰到 Service 对象后，它会生成一个全新的表，所以会最终生成一个列表。
- (8) 屏幕显示文本输出结果。

两种不同的输出是由于将两种类别的对象放入一个管道中。这是将命令存入脚本和手动执行之间的重要区别：在脚本中，只能够使用一个管道。正常来讲，你的脚本应该努力保持只输出一类对象，以便 PowerShell 能产生合理的文本输出格式。

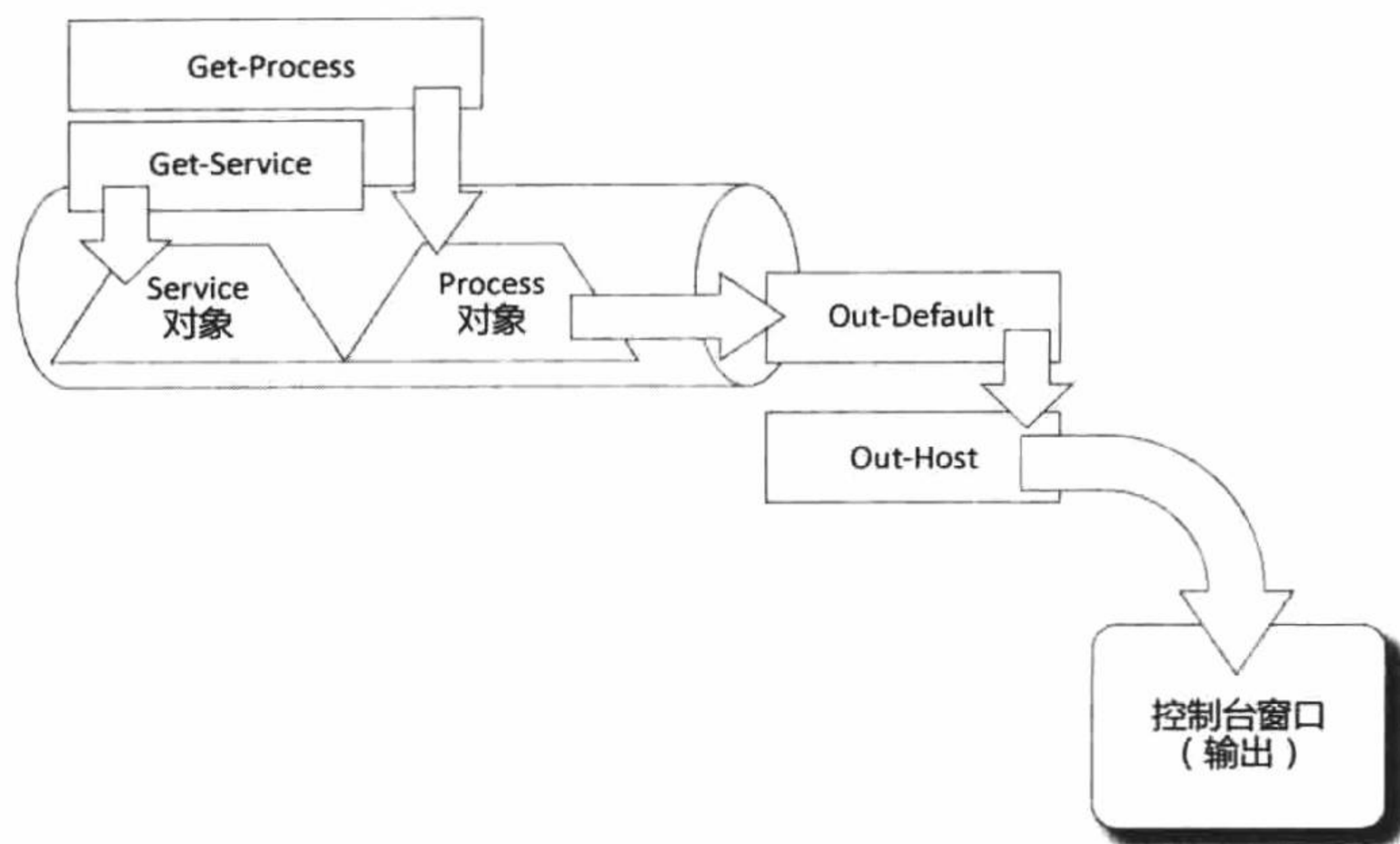


图 21.5 在一个脚本中，所有的命令都是在该脚本单独的管道中执行

21.7 作用域初探

我们最后需要讨论的一个主题是作用域（scope）。作用域是特定类型 PowerShell 元素的容器，这些元素主要是别名、变量和函数。

Shell 本身具有最高级的作用域，称为全局域（global scope）。当运行一个脚本时，会在脚本范围内创建一个新的作用域，也就是所谓的脚本作用域（script scope）。脚本作用域是全局作用域的子集，也就是全局作用域的子作用域（child）。而全局作用域是脚本作用域的父作用域（parent）。函数还有其特有的私有作用域（private scope）。

图 21.6 描述了这些作用域之间的关系，全局作用域包含了其子作用域，而其子作用域包含了其他子作用域，以此类推。

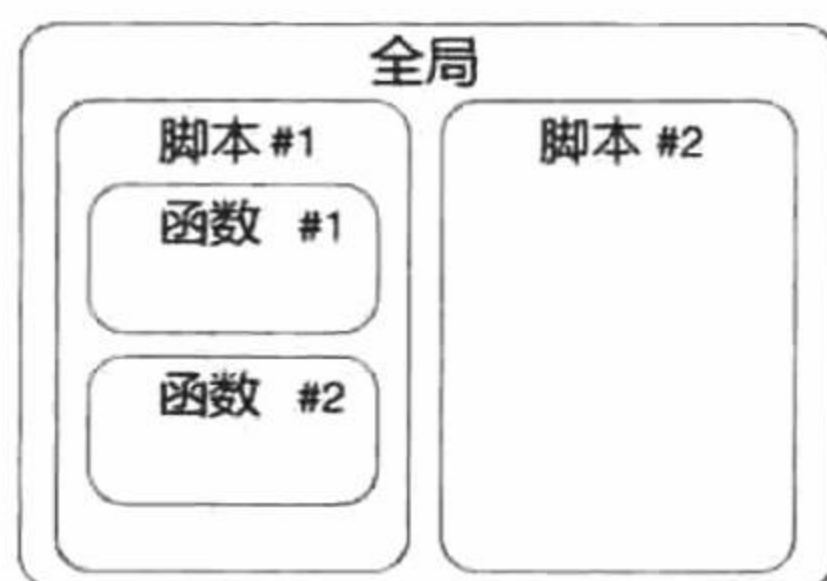


图 21.6 全局脚本及函数（私有）作用域

作用域的生命周期只持续到作用域所需执行的最后一行代码之前。这意味着全局作用域只有在 PowerShell 运行时有效，脚本作用域只在脚本运行时有效，以此类

推。一旦停止运行，作用域和其包含的内容同时消失。PowerShell 对于别名、变量和函数之类的元素有着非常详细——某些时候也是非常让人困惑的规则，但主要规则是，如果你尝试访问一个作用域元素，PowerShell 在当前作用域内查找，如果不存在于当前作用域，PowerShell 会查找其父作用域，以此类推，直到找到树形关系的顶端——也就是全局作用域。

动手实验：为了获得正确的结果，请小心按照下面的指导操作，这非常重要。

让我们进行实战，遵循下面的步骤。

(1) 关闭已经打开的 PowerShell 或 PowerShell ISE 窗口，这样你就可以从头开始。

(2) 打开一个新的 PowerShell 或 PowerShell ISE 窗口。

(3) 在 ISE 中，创建一个包含一行命令的脚本，该命令为 `Write $x`。

(4) 将脚本保存到 `c:\scope.ps1`。

(5) 在一个标准的 PowerShell 窗口，使用命令 `C:\Scope` 运行脚本。没有任何输出结果。当脚本运行时，会自动为其创建一个新的作用域。而 `$x` 变量在该作用域内并不存在，因此 PowerShell 转向其父作用域——也就是全局作用域检查变量 `$x` 是否存在。该变量在父作用域也不存在，因此 PowerShell 认为 `$x` 为空，并打印出空（也就是不输出任何结果）作为输出结果。

(6) 在一个标准的 PowerShell 窗口，运行 `$x = 4`，然后再次运行 `C:\Scope`。这次，你会按到输出结果为 4。虽然变量 `$x` 在脚本范围内未定义，但 PowerShell 可以在全局作用域内找到该变量。因此脚本可以使用全局作用域内的值。

(7) 在 ISE 中，在脚本的开始添加 `$x=10`（也就是 `write` 命令之前），并保存脚本。

(8) 在标准的 PowerShell 窗口中，再次运行 `C:\Scope`。这次，你会看到输出结果为 10。这是由于 `$x` 在脚本作用域内定义，因此 Shell 无须查看全局作用域。现在在 Shell 中运行 `$x`。你将看到输出结果为 4，这意味着在脚本作用域内的变量值不会影响全局作用域内的变量值。

在这里一个重要的概念是，当在作用域内定义一个变量、别名或函数时，当前作用域就无法访问父作用域内的任何同名变量、别名或函数。PowerShell 总会使用局部定义的元素。例如，如果你将 `New-Alias Dir Get-Service` 命令放入一个脚本，那么在当前脚本中，别名 `Dir` 总是运行 `Get-Service` 而不是 `Get-ChildItem`（实际上，Shell 很可能不允许你这么做，这是由于其需要保护内置别名不会重新被定义）。通过在脚本作用域内定义别名，你可以防止 Shell 去父作用域查找标准和默认的 `Dir`。当然，对于 `Dir` 别名的重定义只能持续到脚本执行结束之前，而全局作用域默认的 `Dir` 将不受影响。

这些作用域相关的理念可能会让你感到困惑。你可以通过永远不依赖除了当前作用域内的其他作用域来避免这种混淆。因此在尝试在脚本中访问一个变量时，请确保你已经在同一个作用域内给其赋值。在 `Param()` 块内的参数可以实现这一点，还有很多其他方式可以将值或对象赋予一个变量。

21.8 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。

将下面的命令添加到一个脚本中。你首先需要识别出需要定义为参数的元素，比如说计算机名称。最终的脚本应该定义好参数，并且你还需要为脚本创建基于注释的帮助。运行脚本从而对脚本进行测试，并使用 **Help** 命令，从而确保基于注释的帮助可以正常工作。请不要忘记阅读本章提到的帮助文件以获取更多信息。

下面是命令：

```
Get-WmiObject Win32_LogicalDisk -comp "localhost" -filter "drivetype=3" |  
Where { $_.FreeSpace / $_.Size -lt .1 } |  
Select -Property DeviceID,FreeSpace,Size
```

提示如下：你至少可以发现 2 处信息需要变为参数。该命令用于列出少于给定可用空间的驱动器。显而易见，你并不只想把本地主机作为目标，并且你不希望 10%（也就是 1）作为阈值。你还可以选择将驱动器类型作为参数（这里也就是 3），但是对于动手实验来说，保留其值为 3 即可。