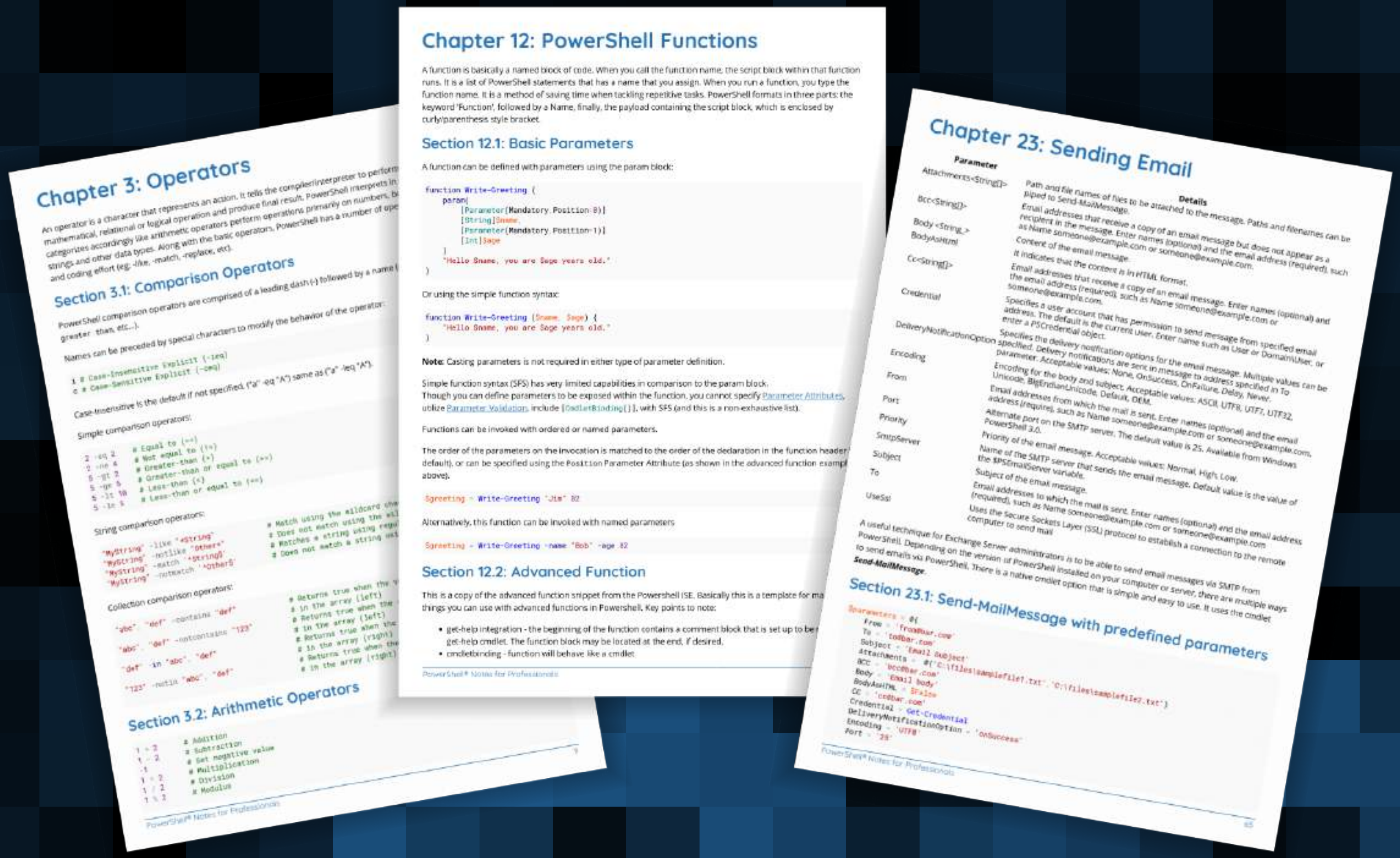
专业人员笔记

PowerShell®

专业人员笔记



Chapter 3: Operators

Section 3.1: Comparison Operators

Section 3.2: Arithmetic Operators

Chapter 12: PowerShell Functions

Section 12.1: Basic Parameters

Section 12.2: Advanced Function

Chapter 23: Sending Email

Section 23.1: Send-MailMessage with predefined parameters

100多页

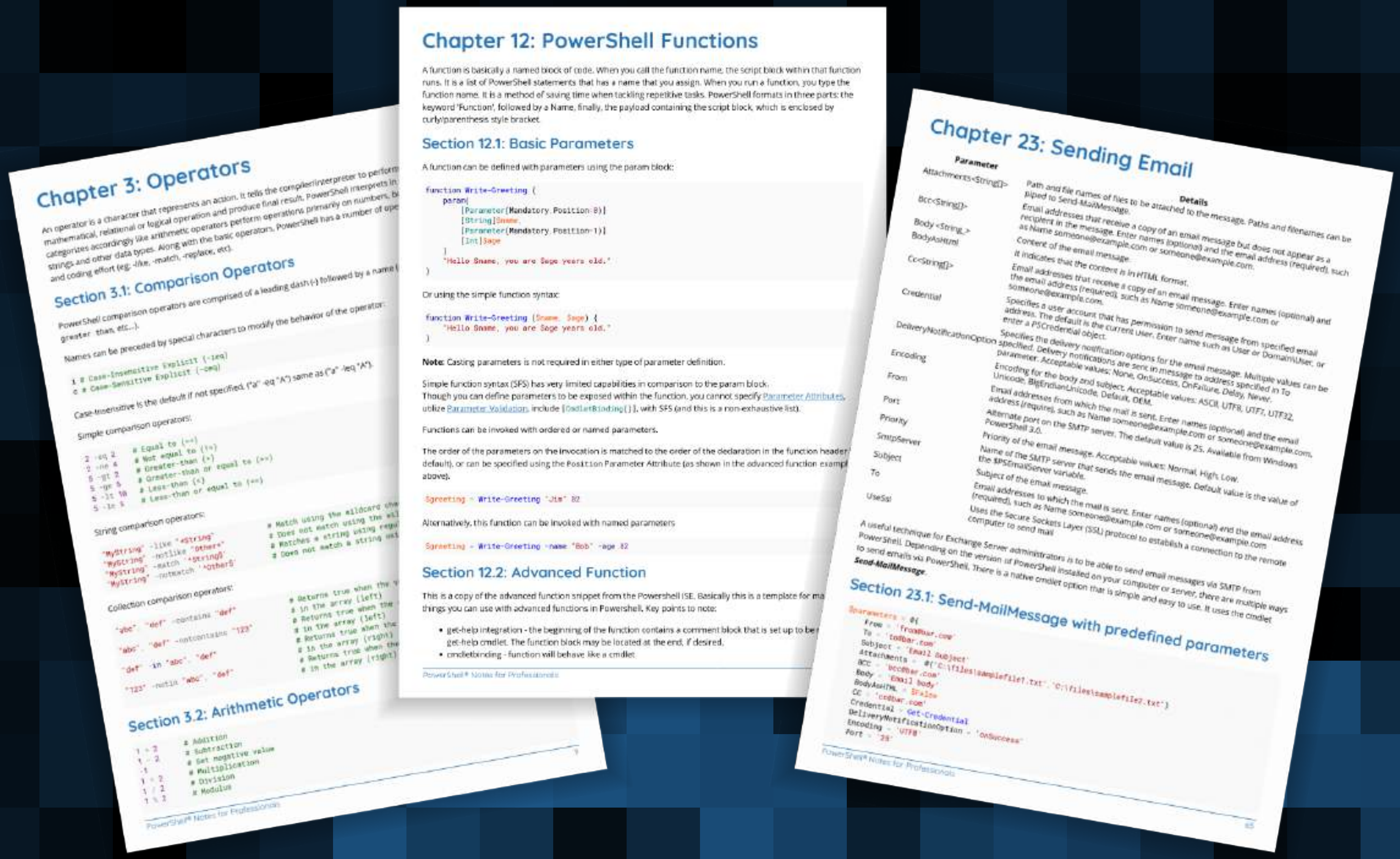
专业提示和技巧

GoalKicker.com
免费编程书籍

免责声明
这是一本非官方的免费书籍，旨在教育用途，
与官方PowerShell®组织或公司无关。
所有商标和注册商标均为其各自所有者所有

PowerShell®

Notes for Professionals



Chapter 3: Operators

Section 3.1: Comparison Operators

Section 3.2: Arithmetic Operators

Chapter 12: PowerShell Functions

Section 12.1: Basic Parameters

Section 12.2: Advanced Function

Chapter 23: Sending Email

Section 23.1: Send-MailMessage with predefined parameters

100+ pages

of professional hints and tricks

GoalKicker.com
Free Programming Books

Disclaimer
This is an unofficial free book created for educational purposes and is
not affiliated with official PowerShell® group(s) or company(s).
All trademarks and registered trademarks are
the property of their respective owners

目录

Contents

关于

欢迎免费分享此PDF给任何人，
本书最新版本可从以下网址下载：
https://goalkicker.com/PowerShellBook

本*PowerShell® 专业笔记*一书汇编自Stack Overflow Documentation，内容由Stack Overflow的优秀人士撰写。文本内容采用知识共享署名-相同方式共享许可发布，详见本书末尾对各章节贡献者的致谢。图片版权归各自所有者所有，除非另有说明。

这是一本非官方的免费书籍，旨在教育用途，与官方PowerShell®组织或公司及Stack Overflow无关。
所有商标和注册商标均为其各自公司所有。

本书所提供的信息不保证正确或准确，使用风险自负。

请将反馈和更正发送至web@petercv.com

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
https://goalkicker.com/PowerShellBook

This *PowerShell® Notes for Professionals* book is compiled from Stack Overflow Documentation, the content is written by the beautiful people at Stack Overflow. Text content is released under Creative Commons BY-SA, see credits at the end of this book whom contributed to the various chapters. Images may be copyright of their respective owners unless otherwise specified

This is an unofficial free book created for educational purposes and is not affiliated with official PowerShell® group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to web@petercv.com

第1章：PowerShell入门

版本	随Windows附带	备注发布日期
1.0	XP / Server 2008	2006-11-01
2.0	7 / Server 2008 R2	2009-11-01
3.0	8 / Server 2012	2012-08-01
4.0	8.1 / Server 2012 R2	2013-11-01
5.0	10 / Server 2016 技术预览	2015-12-16
5.1	10周年纪念版 / 服务器2016	2017-01-27

第1.1节：允许存储在您计算机上的脚本运行未签名的脚本

出于安全原因，PowerShell 默认设置为只允许执行已签名的脚本。执行以下命令将允许您运行未签名的脚本（您必须以管理员身份运行 PowerShell 才能执行此操作）。

```
Set-ExecutionPolicy RemoteSigned
```

运行 PowerShell 脚本的另一种方法是使用Bypass作为ExecutionPolicy：

```
powershell.exe -ExecutionPolicy Bypass -File "c:\MyScript.ps1"
```

或者在现有的 PowerShell 控制台或 ISE 会话中运行：

```
Set-ExecutionPolicy Bypass Process
```

通过运行 PowerShell 可执行文件并传递任何有效的策略作为-ExecutionPolicy参数，也可以实现执行策略的临时解决方案。该策略仅在进程生命周期内生效，因此无需对注册表进行管理员访问。

```
C:\>powershell -ExecutionPolicy RemoteSigned
```

还有多种其他策略可用，且网上的站点常常鼓励你使用Set-ExecutionPolicy Unrestricted。该策略会一直生效直到被更改，并且会降低系统的安全级别。这是不建议的。
建议使用RemoteSigned，因为它允许本地存储和编写的代码运行，并要求远程获取的代码必须由受信任根证书签名。

另外，请注意执行策略可能会被组策略强制执行，因此即使将策略更改为Unrestricted系统范围内，组策略可能会在下一次强制执行间隔（通常为15分钟）恢复该设置。你可以使用Get-ExecutionPolicy-List查看各个作用域设置的执行策略。

TechNet 文档：
Set-ExecutionPolicy
about_Execution_Policies

第1.2节：别名及类似函数

在PowerShell中，有许多方法可以实现相同的结果。用简单且熟悉的Hello World示例可以很好地说明这一点：

使用Write-Host：

Chapter 1: Getting started with PowerShell

Version	Included with Windows	Notes Release Date
1.0	XP / Server 2008	2006-11-01
2.0	7 / Server 2008 R2	2009-11-01
3.0	8 / Server 2012	2012-08-01
4.0	8.1 / Server 2012 R2	2013-11-01
5.0	10 / Server 2016 Tech Preview	2015-12-16
5.1	10 Anniversary edition / Server 2016	2017-01-27

Section 1.1: Allow scripts stored on your machine to run un-signed

For security reasons, PowerShell is set up by default to only allow signed scripts to execute. Executing the following command will allow you to run unsigned scripts (you must run PowerShell as Administrator to do this).

```
Set-ExecutionPolicy RemoteSigned
```

Another way to run PowerShell scripts is to use Bypass as ExecutionPolicy:

```
powershell.exe -ExecutionPolicy Bypass -File "c:\MyScript.ps1"
```

Or from within your existing PowerShell console or ISE session by running:

```
Set-ExecutionPolicy Bypass Process
```

A temporary workaround for execution policy can also be achieved by running the PowerShell executable and passing any valid policy as -ExecutionPolicy parameter. The policy is in effect only during process' lifetime, so no administrative access to the registry is needed.

```
C:\>powershell -ExecutionPolicy RemoteSigned
```

There are multiple other policies available, and sites online often encourage you to use Set-ExecutionPolicy Unrestricted. This policy stays in place until changed, and lowers the system security stance. This is not advisable. Use of RemoteSigned is recommended because it allows locally stored and written code, and requires remotely acquired code be signed with a certificate from a trusted root.

Also, beware that the Execution Policy may be enforced by Group Policy, so that even if the policy is changed to Unrestricted system-wide, Group Policy may revert that setting at its next enforcement interval (typically 15 minutes). You can see the execution policy set at the various scopes using Get-ExecutionPolicy -List

TechNet Documentation:
Set-ExecutionPolicy
about_Execution_Policies

Section 1.2: Aliases & Similar Functions

In PowerShell, there are many ways to achieve the same result. This can be illustrated nicely with the simple & familiar Hello World example:

Using Write-Host:

```
Write-Host "Hello World"
```

使用Write-Output：

```
Write-Output 'Hello world'
```

值得注意的是，虽然Write-Output和Write-Host都可以写入屏幕，但它们之间存在细微的差别。Write-Host仅写入stdout（即控制台屏幕），而Write-Output则写入stdout和输出[成功]流，允许进行重定向。重定向（以及流的概念）允许将一个命令的输出作为另一个命令的输入，包括赋值给变量。

```
> $message = Write-Output "Hello World"
> $message
"Hello World"
```

这些相似的函数不是别名，但如果想避免“污染”成功流，它们可以产生相同的结果。

Write-Output是Echo或Write的别名

```
Echo 'Hello world'
Write 'Hello world'
```

或者，只需输入'Hello world'！

```
'Hello world'
```

以上所有操作都会产生预期的控制台输出

```
你好，世界
```

PowerShell 中别名的另一个示例是将旧的命令提示符命令和BASH 命令映射到 PowerShell cmdlet 的常见做法。以下所有命令都会列出当前目录的目录清单。

```
C:\Windows> dir
C:\Windows> ls
C:\Windows> Get-ChildItem
```

最后，你可以使用 Set-Alias cmdlet 创建自己的别名！例如，我们将 `Test-NetConnection`，它本质上是命令提示符中 ping 命令的 PowerShell 等价物，别名为“ping”。

```
Set-Alias -Name ping -Value Test-NetConnection
```

现在你可以使用 ping 代替 Test-NetConnection！请注意，如果别名已被使用，你将覆盖该关联。

别名在会话期间有效。关闭会话后，尝试运行上次会话中创建的别名将无法使用。为了解决此问题，你可以在开始工作前，将所有别名从 Excel 导入到会话中一次。

第1.3节：管道——使用 PowerShell 的输出

```
Write-Host "Hello World"
```

Using `Write-Output`:

```
Write-Output 'Hello world'
```

It's worth noting that although `Write-Output` & `Write-Host` both write to the screen there is a subtle difference. `Write-Host` writes *only* to stdout (i.e. the console screen), whereas `Write-Output` writes to both stdout *AND* to the output [success] stream allowing for redirection. Redirection (and streams in general) allow for the output of one command to be directed as input to another including assignment to a variable.

```
> $message = Write-Output "Hello World"
> $message
"Hello World"
```

These similar functions are not aliases, but can produce the same results if one wants to avoid "polluting" the success stream.

`Write-Output` is aliased to `Echo` or `Write`

```
Echo 'Hello world'
Write 'Hello world'
```

Or, by simply typing 'Hello world'!

```
'Hello world'
```

All of which will result with the expected console output

```
Hello world
```

Another example of aliases in PowerShell is the common mapping of both older command prompt commands and BASH commands to PowerShell cmdlets. All of the following produce a directory listing of the current directory.

```
C:\Windows> dir
C:\Windows> ls
C:\Windows> Get-ChildItem
```

Finally, you can create your own alias with the Set-Alias cmdlet! As an example let's alisas `Test-NetConnection`, which is essentially the PowerShell equivalent to the command prompt's ping command, to "ping".

```
Set-Alias -Name ping -Value Test-NetConnection
```

Now you can use `ping` instead of `Test-NetConnection`! Be aware that if the alias is already in use, you'll overwrite the association.

The Alias will be alive, till the session is active. Once you close the session and try to run the alias which you have created in your last session, it will not work. To overcome this issue, you can import all your aliases from an excel into your session once, before starting your work.

Section 1.3: The Pipeline - Using Output from a PowerShell

cmdlet

人们开始使用 PowerShell 编写脚本时，最常问的一个问题是如何操作 cmdlet 的输出以执行其他操作。

管道符号 | 用于 cmdlet 的末尾，将其导出的数据传递给下一个 cmdlet。一个简单的例子是使用 Select-Object 仅显示 Get-ChildItem 显示的文件的 Name 属性：

```
Get-ChildItem | Select-Object Name
#这可以简写为：
gci | Select Name
```

管道的更高级用法允许我们将 cmdlet 的输出传递到 foreach 循环中：

```
Get-ChildItem | ForEach-Object {
    Copy-Item -Path $_.FullName -destination C:\NewDirectory\
}

#这可以简写为：
gci | % { Copy $_.FullName C:\NewDirectory\ }
```

注意，上述示例使用了 $_ 自动变量。$_ 是 $PSItem 的简写别名，$PSItem 是一个自动变量，包含管道中的当前项。

第1.4节：调用 .Net 库方法

可以通过将完整类名用中括号括起来，然后使用 :: 调用方法，从 PowerShell 调用静态 .Net 库方法

```
#调用 Path.GetFileName()
C:\> [System.IO.Path]::GetFileName('C:\Windows\explorer.exe')
explorer.exe
```

静态方法可以直接从类本身调用，但调用非静态方法需要该.Net
类的实例（对象）。

例如，AddHours 方法不能直接从 System.DateTime 类调用。它需要该类的一个实例：

```
C:\> [System.DateTime]::AddHours(15)
方法调用失败，因为 [System.DateTime] 不包含名为 'AddHours' 的方法。
在第 1 行：1 字符：1
+ [System.DateTime]::AddHours(15)
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    +CategoryInfo          : InvalidOperation: (:) [], RuntimeException
    + FullyQualifiedErrorId : MethodNotFound
```

在这种情况下，我们首先创建一个对象，例如：

```
C:\> $Object = [System.DateTime]::Now
```

然后，我们可以使用该对象的方法，甚至是那些不能直接从 System.DateTime
类调用的方法，比如 AddHours 方法：

```
C:\> $Object.AddHours(15)
```

cmdlet

One of the first questions people have when they begin to use PowerShell for scripting is how to manipulate the output from a cmdlet to perform another action.

The pipeline symbol | is used at the end of a cmdlet to take the data it exports and feed it to the next cmdlet. A simple example is using Select-Object to only show the Name property of a file shown from Get-ChildItem:

```
Get-ChildItem | Select-Object Name
#This may be shortened to:
gci | Select Name
```

More advanced usage of the pipeline allows us to pipe the output of a cmdlet into a foreach loop:

```
Get-ChildItem | ForEach-Object {
    Copy-Item -Path $_.FullName -destination C:\NewDirectory\
}

#This may be shortened to:
gci | % { Copy $_.FullName C:\NewDirectory\ }
```

Note that the example above uses the $_ automatic variable. $_ is the short alias of $PSItem which is an automatic variable which contains the current item in the pipeline.

Section 1.4: Calling .Net Library Methods

Static .Net library methods can be called from PowerShell by encapsulating the full class name in third bracket and then calling the method using ::

```
#calling Path.GetFileName()
C:\> [System.IO.Path]::GetFileName('C:\Windows\explorer.exe')
explorer.exe
```

Static methods can be called from the class itself, but calling non-static methods requires an instance of the .Net class (an object).

For example, the AddHours method cannot be called from the System.DateTime class itself. It requires an instance of the class:

```
C:\> [System.DateTime]::AddHours(15)
Method invocation failed because [System.DateTime] does not contain a method named 'AddHours'.
At line:1 char:1
+ [System.DateTime]::AddHours(15)
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidOperation: (:) [], RuntimeException
    + FullyQualifiedErrorId : MethodNotFound
```

In this case, we first create an object, for example:

```
C:\> $Object = [System.DateTime]::Now
```

Then, we can use methods of that object, even methods which cannot be called directly from the System.DateTime class, like the AddHours method:

```
C:\> $Object.AddHours(15)
```

```
2016年9月12日 星期一 01:51:19
```

第1.5节：安装或设置

Windows

PowerShell 包含在 Windows 管理框架中。现代版本的 Windows 不需要安装和设置。

通过安装更新版本的 Windows 管理框架，可以完成 PowerShell 的更新。

其他平台

PowerShell 6 可以安装在其他平台上。安装包可在 here 获得。

例如，PowerShell 6 对于 Ubuntu 16.04，已发布到软件包仓库，便于安装（和更新）。

安装请运行以下命令：

```
# 导入公共仓库 GPG 密钥
curl https://packages.microsoft.com/keys/microsoft.asc | sudo apt-key add -

# 注册 Microsoft Ubuntu 仓库
curl https://packages.microsoft.com/config/ubuntu/16.04/prod.list | sudo tee
/etc/apt/sources.list.d/microsoft.list

# 更新 apt-get
sudo apt-get update

# 安装 PowerShell
sudo apt-get install -y powershell

# 启动 PowerShell
powershell
```

作为超级用户注册一次微软仓库后，之后只需使用 `sudo apt-get upgrade powershell` 来更新它。然后只需运行 `powershell`

第1.6节：注释

通过在行首添加 #（井号）符号来注释PowerShell脚本

```
# 这是PowerShell中的注释
Get-ChildItem
```

你也可以使用 `<#` 和 `#>` 分别在注释的开始和结束处，实现多行注释。

```
<#
这是一个
多行
注释
#>
Get-ChildItem
```

```
Monday 12 September 2016 01:51:19
```

Section 1.5: Installation or Setup

Windows

PowerShell is included with the Windows Management Framework. Installation and Setup are not required on modern versions of Windows.

Updates to PowerShell can be accomplished by installing a newer version of the Windows Management Framework.

Other Platforms

PowerShell 6 can be installed on other platforms. The installation packages are available here.

For example, PowerShell 6, for Ubuntu 16.04, is published to package repositories for easy installation (and updates).

To install run the following:

```
# Import the public repository GPG keys
curl https://packages.microsoft.com/keys/microsoft.asc | sudo apt-key add -

# Register the Microsoft Ubuntu repository
curl https://packages.microsoft.com/config/ubuntu/16.04/prod.list | sudo tee
/etc/apt/sources.list.d/microsoft.list

# Update apt-get
sudo apt-get update

# Install PowerShell
sudo apt-get install -y powershell

# Start PowerShell
powershell
```

After registering the Microsoft repository once as superuser, from then on, you just need to use `sudo apt-get upgrade powershell` to update it. Then just run `powershell`

Section 1.6: Commenting

To comment on power scripts by prepending the line using the # (hash) symbol

```
# This is a comment in PowerShell
Get-ChildItem
```

You can also have multi-line comments using `<#` and `#>` at the beginning and end of the comment respectively.

```
<#
This is a
multi-line
comment
#>
Get-ChildItem
```

第1.7节：创建对象

New-Object 命令用于创建对象。

```
# 创建一个 DateTime 对象并将该对象存储在变量 "$var" 中
$var = New-Object System.DateTime

# 使用带参数的构造函数
$sr = New-Object System.IO.StreamReader -ArgumentList "文件路径"
```

在许多情况下，会创建一个新对象以导出数据或将其传递给另一个命令。可以这样做：

```
$newObject = New-Object -TypeName PSObject -Property @{
    ComputerName = "SERVER1"
Role = "Interface"
    Environment = "Production"
}
```

创建对象的方法有很多。以下方法可能是创建PSCustomObject最简短且最快的方法：

```
$newObject = [PSCustomObject]@{
    ComputerName = 'SERVER1'
    Role         = 'Interface'
    Environment  = 'Production'
}
```

如果你已经有一个对象，但只需要添加一两个额外属性，可以使用Select-Object简单地添加该属性：

```
Get-ChildItem | Select-Object FullName, Name,
    @{Name='DateTime'; Expression={Get-Date}},
    @{Name='PropertyName'; Expression={'CustomValue'}}
```

所有对象都可以存储在变量中或传递到管道中。你也可以将这些对象添加到集合中，然后在最后显示结果。

对象集合与 Export-CSV（和 Import-CSV）配合良好。CSV 的每一行是一个对象，每一列是一个属性。

格式命令将对象转换为用于显示的文本流。避免在任何数据处理的最终步骤之前使用 Format-* 命令，以保持对象的可用性。

Section 1.7: Creating Objects

The New-Object cmdlet is used to create an object.

```
# Create a DateTime object and stores the object in variable "$var"
$var = New-Object System.DateTime

# calling constructor with parameters
$sr = New-Object System.IO.StreamReader -ArgumentList "file path"
```

In many instances, a new object will be created in order to export data or pass it to another commandlet. This can be done like so:

```
$newObject = New-Object -TypeName PSObject -Property @{
    ComputerName = "SERVER1"
    Role = "Interface"
    Environment = "Production"
}
```

There are many ways of creating an object. The following method is probably the shortest and fastest way to create a PSCustomObject:

```
$newObject = [PSCustomObject]@{
    ComputerName = 'SERVER1'
    Role         = 'Interface'
    Environment  = 'Production'
}
```

In case you already have an object, but you only need one or two extra properties, you can simply add that property by using Select-Object:

```
Get-ChildItem | Select-Object FullName, Name,
    @{Name='DateTime'; Expression={Get-Date}},
    @{Name='PropertyName'; Expression={'CustomValue'}}
```

All objects can be stored in variables or passed into the pipeline. You could also add these objects to a collection and then show the results at the end.

Collections of objects work well with Export-CSV (and Import-CSV). Each line of the CSV is an object, each column a property.

Format commands convert objects into text stream for display. Avoid using Format-* commands until the final step of any data processing, to maintain the usability of the objects.

第2章：PowerShell中的变量

变量用于存储值。无论值的类型如何，我们都需要将其存储在某处，以便在整个控制台/脚本中使用。PowerShell中的变量名以$开头，例如$Variable1，值通过=赋值，如$Variable1 = "Value 1"。PowerShell支持大量变量类型；例如文本字符串、整数、小数、数组，甚至高级类型如版本号或IP地址。

第2.1节：简单变量

PowerShell中的所有变量都以美元符号（$）开头。最简单的例子是：

```
$foo = "bar"
```

该语句分配了一个名为foo的变量，其字符串值为"bar"。

第2.2节：数组

PowerShell中数组的声明几乎与实例化任何其他变量相同，即使用$name =语法。数组中的项通过逗号（,）分隔声明：

```
$myArrayOfInts = 1,2,3,4
$myArrayOfStrings = "1","2","3","4"
```

向数组添加元素

向数组添加元素就像使用+运算符一样简单：

```
$myArrayOfInts = $myArrayOfInts + 5
# 现在包含 1,2,3,4 和 5！
```

合并数组

这同样像使用+运算符一样简单

```
$myArrayOfInts = 1,2,3,4
$myOtherArrayOfInts = 5,6,7
$myArrayOfInts = $myArrayOfInts + $myOtherArrayOfInts
# 现在是 1,2,3,4,5,6,7
```

第2.3节：多个变量的列表赋值

Powershell 允许多个变量赋值，并且几乎将所有内容都视为数组或列表。这意味着你不必像这样做：

```
$input = "foo.bar.baz"
$parts = $input.Split(".")
$foo = $parts[0]
$bar = $parts[1]
$baz = $parts[2]
```

你可以简单地这样做：

```
$foo, $bar, $baz = $input.Split(".")
```

Chapter 2: Variables in PowerShell

Variables are used for storing values. Let the value be of any type , we need to store it somewhere so that we can use it throughout the console/script. Variable names in PowerShell begin with a **$**, as in *$Variable1*, and values are assigned using **=**, like **$Variable1 = "Value 1"**.PowerShell supports a huge number of variable types; such as text strings, integers, decimals, arrays, and even advanced types like version numbers or IP addresses.

Section 2.1: Simple variable

All variables in PowerShell begin with a US dollar sign ($). The simplest example of this is:

```
$foo = "bar"
```

This statement allocates a variable called `foo` with a string value of "bar".

Section 2.2: Arrays

Array declaration in Powershell is almost the same as instantiating any other variable, i.e. you use a `$name =` syntax. The items in the array are declared by separating them by commas(,):

```
$myArrayOfInts = 1,2,3,4
$myArrayOfStrings = "1","2","3","4"
```

Adding to an array

Adding to an array is as simple as using the + operator:

```
$myArrayOfInts = $myArrayOfInts + 5
# now contains 1,2,3,4 & 5!
```

Combining arrays together

Again this is as simple as using the + operator

```
$myArrayOfInts = 1,2,3,4
$myOtherArrayOfInts = 5,6,7
$myArrayOfInts = $myArrayOfInts + $myOtherArrayOfInts
# now 1,2,3,4,5,6,7
```

Section 2.3: List Assignment of Multiple Variables

Powershell allows multiple assignment of variables and treats almost everything like an array or list. This means that instead of doing something like this:

```
$input = "foo.bar.baz"
$parts = $input.Split(".")
$foo = $parts[0]
$bar = $parts[1]
$baz = $parts[2]
```

You can simply do this:

```
$foo, $bar, $baz = $input.Split(".")
```

由于Powershell将赋值视为列表，如果列表中的值多于你要赋值的变量列表中的项，最后一个变量将成为剩余值的数组。这意味着你也可以这样做：

```
$foo, $leftover = $input.Split(".") #将 $foo 设置为 "foo"，$leftover 设置为 ["bar","baz"]
$bar = $leftover[0] # $bar = "bar"
$baz = $leftover[1] # $baz = "baz"
```

第2.4节：作用域

变量的默认作用域是其所在的容器。如果在脚本或其他容器之外，则作用域为全局。要指定作用域，可以在变量名前加前缀$scope:varname，如下所示：

```
$foo = "全局作用域"
function myFunc {
    $foo = “函数（局部）作用域”
    Write-Host $global:foo
    Write-Host $local:foo
    Write-Host $foo
}
myFunc
Write-Host $local:foo
Write-Host $foo
```

输出：

全局作用域 函数（局部）作用域 函数（局部）作用域 全局作用域 全局作用域

第2.5节：删除变量

要从内存中删除变量，可以使用Remove-Item命令。注意：变量名不包含$。

```
Remove-Item 变量：\foo
```

变量有一个提供程序，允许大多数 *-item 命令像文件系统一样工作。

另一种移除变量的方法是使用 Remove-Variable 命令和它的别名 rv

```
$var = "Some Variable" #定义变量 'var'，内容为字符串 'Some Variable'
$var #用于测试并在控制台显示字符串 'Some Variable'

Remove-Variable -Name var
$var

#也可以使用别名 'rv'
rv var
```

Since Powershell treats assignments in this manner like lists, if there are more values in the list than items in your list of variables to assign them to, the last variable becomes an array of the remaining values. This means you can also do things like this:

```
$foo, $leftover = $input.Split(".") #Sets $foo = "foo", $leftover = ["bar","baz"]
$bar = $leftover[0] # $bar = "bar"
$baz = $leftover[1] # $baz = "baz"
```

Section 2.4: Scope

The default scope for a variable is the enclosing container. If outside a script, or other container then the scope is `Global`. To specify a scope, it is prefixed to the variable name `$scope:varname` like so:

```
$foo = "Global Scope"
function myFunc {
    $foo = "Function (local) scope"
    Write-Host $global:foo
    Write-Host $local:foo
    Write-Host $foo
}
myFunc
Write-Host $local:foo
Write-Host $foo
```

Output:

Global Scope Function (local) scope Function (local) scope Global Scope Global Scope

Section 2.5: Removing a variable

To remove a variable from memory, one can use the `Remove-Item` cmdlet. Note: The variable name does NOT include the $.

```
Remove-Item Variable:\foo
```

`Variable` has a provider to allow most *-item cmdlets to work much like file systems.

Another method to remove variable is to use Remove-Variable cmdlet and its alias rv

```
$var = "Some Variable" #Define variable 'var' containing the string 'Some Variable'
$var #For test and show string 'Some Variable' on the console

Remove-Variable -Name var
$var

#also can use alias 'rv'
rv var
```

第3章：运算符

运算符是表示操作的字符。它告诉编译器/解释器执行特定的数学、关系或逻辑运算并产生最终结果。PowerShell以特定方式解释并相应分类，例如算术运算符主要对数字进行操作，但它们也影响字符串和其他数据类型。除了基本运算符外，PowerShell还有许多节省时间和编码工作量的运算符（例如：-like、-match、-replace等）。

第3.1节：比较运算符

PowerShell比较运算符由前导短横线（-）加名称组成（eq表示等于，gt表示大于，等等）。

名称前可以加特殊字符以修改运算符的行为：

```
i # 不区分大小写显式 (-ieq)
c # 区分大小写显式 (-ceq)
```

如果未指定，默认是不区分大小写的，("a" -eq "A") 等同于 ("a" -ieq "A")。

简单比较运算符：

```
2 -eq 2    # 等于 (==)
2 -ne 4    # 不等于 (!=)
5 -gt 2    # 大于 (>)
5 -ge 5    # 大于或等于 (>=)
5 -lt 10   # 小于 (<)
5 -le 5    # 小于或等于 (<=)
```

字符串比较运算符：

```
"MyString" -like "*String"            # 使用通配符 (*) 匹配
"MyString" -notlike "Other*"          # 不使用通配符 (*) 匹配
"MyString" -match '^String$'          # 使用正则表达式匹配字符串
"MyString" -notmatch '^Other$'        # 不使用正则表达式匹配字符串
```

集合比较运算符：

```
"abc", "def" -包含 "def"              # 当值（右侧）存在于数组（左侧）中时返回真
                                           #
"abc", "def" -notcontains "123"            # 当值（右侧）不在数组（左侧）中时返回真
                                           #
"def" -in "abc", "def"                     # 当值（左侧）存在于数组（右侧）中时返回真
                                           #
"123" -notin "abc", "def"                  # 当值（左侧）不在数组（右侧）中时返回真
                                           #
```

第3.2节：算术运算符

```
1 + 2      # 加法
1 - 2      # 减法
-1         # 设为负值
1 * 2      # 乘法
1 / 2      # 除法
1 % 2      # 取模
```

Chapter 3: Operators

An operator is a character that represents an action. It tells the compiler/interpreter to perform specific mathematical, relational or logical operation and produce final result. PowerShell interprets in a specific way and categorizes accordingly like arithmetic operators perform operations primarily on numbers, but they also affect strings and other data types. Along with the basic operators, PowerShell has a number of operators that save time and coding effort (eg: -like, -match, -replace, etc).

Section 3.1: Comparison Operators

PowerShell comparison operators are comprised of a leading dash (-) followed by a name (`eq` for `equal`, `gt` for `greater than`, etc...).

Names can be preceded by special characters to modify the behavior of the operator:

```
i # Case-Insensitive Explicit (-ieq)
c # Case-Sensitive Explicit (-ceq)
```

Case-Insensitive is the default if not specified, ("a" -eq "A") same as ("a" -ieq "A").

Simple comparison operators:

```
2 -eq 2    # Equal to (==)
2 -ne 4    # Not equal to (!=)
5 -gt 2    # Greater-than (>)
5 -ge 5    # Greater-than or equal to (>=)
5 -lt 10   # Less-than (<)
5 -le 5    # Less-than or equal to (<=)
```

String comparison operators:

```
"MyString" -like "*String"            # Match using the wildcard character (*)
"MyString" -notlike "Other*"          # Does not match using the wildcard character (*)
"MyString" -match '^String$'          # Matches a string using regular expressions
"MyString" -notmatch '^Other$'        # Does not match a string using regular expressions
```

Collection comparison operators:

```
"abc", "def" -contains "def"            # Returns true when the value (right) is present
                                        # in the array (left)
"abc", "def" -notcontains "123"         # Returns true when the value (right) is not present
                                        # in the array (left)
"def" -in "abc", "def"                  # Returns true when the value (left) is present
                                        # in the array (right)
"123" -notin "abc", "def"               # Returns true when the value (left) is not present
                                        # in the array (right)
```

Section 3.2: Arithmetic Operators

```
1 + 2      # Addition
1 - 2      # Subtraction
-1         # Set negative value
1 * 2      # Multiplication
1 / 2      # Division
1 % 2      # Modulus
```

```
100 -shl 2 # 按位左移
100 -shr 1 # 按位右移
```

第3.3节：赋值运算符

简单算术：

```
$var = 1       # 赋值。将变量的值设置为指定值
$var += 2      # 加法。将变量的值增加指定值
$var -= 1      # 减法。将变量的值减少指定值
$var *= 2      # 乘法。将变量的值乘以指定值
$var /= 2      # 除法。将变量的值除以指定值
$var %= 2      # 取模。将变量的值除以指定值，然后
               # 将余数（模）赋给变量
```

自增和自减：

```
$var++    # 将变量、可赋值属性或数组元素的值增加1
$var--    # 将变量、可赋值属性或数组元素的值减少1
```

第3.4节：重定向运算符

成功输出流：

```
cmdlet > 文件     # 将成功输出发送到文件，覆盖现有内容
cmdlet >> 文件    # 将成功输出发送到文件，追加到现有内容
cmdlet 1>&2        # 将成功和错误输出发送到错误流
```

错误输出流：

```
cmdlet 2> file     # 将错误输出发送到文件，覆盖现有内容
cmdlet 2>> file    # 将错误输出发送到文件，追加到现有内容
cmdlet 2>&1        # 将成功和错误输出发送到成功输出流
```

警告输出流：（PowerShell 3.0及以上）

```
cmdlet 3> file     # 将警告输出发送到文件，覆盖现有内容
cmdlet 3>> file    # 将警告输出发送到文件，追加到现有内容
cmdlet 3>&1        # 将成功和警告输出发送到成功输出流
```

详细输出流：（PowerShell 3.0及以上）

```
cmdlet 4> file     # 将详细输出发送到文件，覆盖现有内容
cmdlet 4>> file    # 将详细输出发送到文件，追加到现有内容
cmdlet 4>&1        # 将成功和详细输出发送到成功输出流
```

调试输出流：（PowerShell 3.0及以上）

```
cmdlet 5> file     # 将调试输出发送到文件，覆盖现有内容
cmdlet 5>> file    # 将调试输出发送到文件，追加到现有内容
cmdlet 5>&1        # 将成功和调试输出发送到成功输出流
```

信息输出流：（PowerShell 5.0及以上）

```
cmdlet 6> file     # 将信息输出发送到文件，覆盖现有内容
cmdlet 6>> file    # 将信息输出发送到文件，追加到现有内容
```

```
100 -shl 2 # Bitwise Shift-left
100 -shr 1 # Bitwise Shift-right
```

Section 3.3: Assignment Operators

Simple arithmetic:

```
$var = 1       # Assignment. Sets the value of a variable to the specified value
$var += 2      # Addition. Increases the value of a variable by the specified value
$var -= 1      # Subtraction. Decreases the value of a variable by the specified value
$var *= 2      # Multiplication. Multiplies the value of a variable by the specified value
$var /= 2      # Division. Divides the value of a variable by the specified value
$var %= 2      # Modulus. Divides the value of a variable by the specified value and then
               # assigns the remainder (modulus) to the variable
```

Increment and decrement:

```
$var++    # Increases the value of a variable, assignable property, or array element by 1
$var--    # Decreases the value of a variable, assignable property, or array element by 1
```

Section 3.4: Redirection Operators

Success output stream:

```
cmdlet > file      # Send success output to file, overwriting existing content
cmdlet >> file     # Send success output to file, appending to existing content
cmdlet 1>&2        # Send success and error output to error stream
```

Error output stream:

```
cmdlet 2> file     # Send error output to file, overwriting existing content
cmdlet 2>> file    # Send error output to file, appending to existing content
cmdlet 2>&1        # Send success and error output to success output stream
```

Warning output stream: (PowerShell 3.0+)

```
cmdlet 3> file     # Send warning output to file, overwriting existing content
cmdlet 3>> file    # Send warning output to file, appending to existing content
cmdlet 3>&1        # Send success and warning output to success output stream
```

Verbose output stream: (PowerShell 3.0+)

```
cmdlet 4> file     # Send verbose output to file, overwriting existing content
cmdlet 4>> file    # Send verbose output to file, appending to existing content
cmdlet 4>&1        # Send success and verbose output to success output stream
```

Debug output stream: (PowerShell 3.0+)

```
cmdlet 5> file     # Send debug output to file, overwriting existing content
cmdlet 5>> file    # Send debug output to file, appending to existing content
cmdlet 5>&1        # Send success and debug output to success output stream
```

Information output stream: (PowerShell 5.0+)

```
cmdlet 6> file     # Send information output to file, overwriting existing content
cmdlet 6>> file    # Send information output to file, appending to existing content
cmdlet 6>&1       # 将成功和信息输出发送到成功输出流
```

所有输出流：

```
cmdlet *> 文件    # 将所有输出流发送到文件，覆盖现有内容
cmdlet *>> 文件   # 将所有输出流发送到文件，追加到现有内容
cmdlet *>&1       # 将所有输出流发送到成功输出流
```

与管道操作符（|）的区别

重定向操作符仅将流重定向到文件或流到流。管道操作符将对象传递到管道中的cmdlet或输出。管道的工作方式通常与重定向不同，详细内容可参见“使用PowerShell管道”

第3.5节：混合操作数类型，左侧操作数的类型决定行为

加法示例

```
"4" + 2         # 结果为 "42"
4 + "2"         # 结果为 6
1,2,3 + "Hello" # 结果为 1,2,3,"Hello"
"Hello" + 1,2,3 # 结果为 "Hello1 2 3"
```

用于乘法

```
"3" * 2   # 结果为 "33"
2 * "3"   # 结果为 6
1,2,3 * 2 # 结果为 1,2,3,1,2,3
2 * 1,2,3 # 报错，缺少 op_Multiply
```

该影响可能对比较运算符产生隐藏的后果：

```
$a = Read-Host "请输入一个数字"
请输入一个数字  : 33
$a -gt 5
False
```

第3.6节：逻辑运算符

```
-and # 逻辑与
-or  # 逻辑或
-xor # 逻辑异或
-not # 逻辑非
!    # 逻辑非
```

第3.7节：字符串操作符

替换操作符：

-replace 操作符使用正则表达式替换输入值中的模式。该操作符使用两个参数（用逗号分隔）：一个正则表达式模式及其替换值（可选，默认是空字符串）。

```
"The rain in Seattle" -replace 'rain','hail'        # 返回：The hail in Seattle
```

```
cmdlet 6>&1       # Send success and information output to success output stream
```

All output streams:

```
cmdlet *> file    # Send all output streams to file, overwriting existing content
cmdlet *>> file   # Send all output streams to file, appending to existing content
cmdlet *>&1       # Send all output streams to success output stream
```

Differences to the pipe operator (|)

Redirection operators only redirect streams to files or streams to streams. The pipe operator pumps an object down the pipeline to a cmdlet or the output. How the pipeline works differs in general from how redirection works and can be read on Working with the PowerShell pipeline

Section 3.5: Mixing operand types, the type of the left operand dictates the behavior

For Addition

```
"4" + 2         # Gives "42"
4 + "2"         # Gives 6
1,2,3 + "Hello" # Gives 1,2,3,"Hello"
"Hello" + 1,2,3 # Gives "Hello1 2 3"
```

For Multiplication

```
"3" * 2   # Gives "33"
2 * "3"   # Gives 6
1,2,3 * 2 # Gives 1,2,3,1,2,3
2 * 1,2,3 # Gives an error op_Multiply is missing
```

The impact may have hidden consequences on comparison operators:

```
$a = Read-Host "Enter a number"
Enter a number : 33
$a -gt 5
False
```

Section 3.6: Logical Operators

```
-and # Logical and
-or  # Logical or
-xor # Logical exclusive or
-not # Logical not
!    # Logical not
```

Section 3.7: String Manipulation Operators

Replace operator:

The `-replace` operator replaces a pattern in an input value using a regular expression. This operator uses two arguments (separated by a comma): a regular expression pattern and its replacement value (which is optional and an empty string by default).

```
"The rain in Seattle" -replace 'rain','hail'        #Returns: The hail in Seattle
```

```
"kenmyer@contoso.com" -replace '^[\w]+@(.+)', '$1'  # 返回：contoso.com
```

拆分和连接操作符：

-split 操作符将字符串拆分成子字符串数组。

```
"A B C" -split " "      #返回一个包含 A、B 和 C 的数组字符串集合对象。
```

-join 操作符将字符串数组连接成一个字符串。

```
"E","F","G" -join ":"    #返回一个字符串：E:F:G
```

```
"kenmyer@contoso.com" -replace '^[\w]+@(.+)', '$1'  #Returns: contoso.com
```

Split and Join operators:

The -split operator splits a string into an array of sub-strings.

```
"A B C" -split " "      #Returns an array string collection object containing A,B and C.
```

The -join operator joins an array of strings into a single string.

```
"E","F","G" -join ":"    #Returns a single string: E:F:G
```

第4章：特殊操作符

第4.1节：数组表达式操作符

将表达式作为数组返回。

```
@(Get-ChildItem $env:windir\System32tdll.dll)
```

将返回一个包含一个项目的数组

```
@(Get-ChildItem $env:windir\System32)
```

将返回文件夹中所有项目的数组（这与内部表达式的行为没有变化）。

第4.2节：调用操作

```
$command = 'Get-ChildItem'
& $Command
```

将执行 Get-ChildItem

第4.3节：点源操作符

. .\myScript.ps1

在当前作用域运行.\myScript.ps1，使任何函数和变量在当前作用域中可用。

Chapter 4: Special Operators

Section 4.1: Array Expression Operator

Returns the expression as an array.

```
@(Get-ChildItem $env:windir\System32\ntdll.dll)
```

Will return an array with one item

```
@(Get-ChildItem $env:windir\System32)
```

Will return an array with all the items in the folder (which is not a change of behavior from the inner expression.

Section 4.2: Call Operation

```
$command = 'Get-ChildItem'
& $Command
```

Will execute `Get-ChildItem`

Section 4.3: Dot sourcing operator

. .\myScript.ps1

runs `.\myScript.ps1` in the current scope making any functions, and variable available in the current scope.

第5章：基本集合操作

集合是一组可以是任何事物的项目。我们需要对这些集合进行的操作统称为集合操作符，操作也称为集合操作。基本集合操作包括并集、交集以及加法、减法等。

第5.1节：过滤：Where-Object / where / ?

使用条件表达式过滤枚举

同义词：

```
Where-Object
where
?
```

示例：

```
$names = @( "Aaron", "Albert", "Alphonse","Bernie", "Charlie", "Danny", "Ernie", "Frank")

$names | Where-Object { $_ -like "A*" }
$names | where { $_ -like "A*" }
$names | ? { $_ -like "A*" }
```

返回：

```
Aaron
Albert
Alphonse
```

第5.2节：排序：Sort-Object / sort

按升序或降序对枚举进行排序

同义词：

```
Sort-Object
sort
```

假设：

```
$names = @( "Aaron", "Aaron", "Bernie", "Charlie", "Danny" )
```

默认是升序排序：

```
$names | Sort-Object
$names | sort
```

```
Aaron
Aaron
Bernie
```

Chapter 5: Basic Set Operations

A set is a collection of items which can be anything. Whatever operator we need to work on these sets are in short the *set operators* and the operation is also known as *set operation*. Basic set operation includes Union, Intersection as well as addition, subtraction, etc.

Section 5.1: Filtering: Where-Object / where / ?

Filter an enumeration by using a conditional expression

Synonyms:

```
Where-Object
where
?
```

Example:

```
$names = @( "Aaron", "Albert", "Alphonse","Bernie", "Charlie", "Danny", "Ernie", "Frank")

$names | Where-Object { $_ -like "A*" }
$names | where { $_ -like "A*" }
$names | ? { $_ -like "A*" }
```

Returns:

```
Aaron
Albert
Alphonse
```

Section 5.2: Ordering: Sort-Object / sort

Sort an enumeration in either ascending or descending order

Synonyms:

```
Sort-Object
sort
```

Assuming:

```
$names = @( "Aaron", "Aaron", "Bernie", "Charlie", "Danny" )
```

Ascending sort is the default:

```
$names | Sort-Object
$names | sort
```

```
Aaron
Aaron
Bernie
```

```
Charlie
Danny
```

要请求降序排序：

```
$names | Sort-Object -Descending
$names | sort -Descending
```

```
Danny
Charlie
Bernie
Aaron
Aaron
```

你可以使用表达式进行排序。

```
$names | Sort-Object { $_.length }
```

```
Aaron
Aaron
Danny
Bernie
Charlie
```

第5.3节：分组：组-对象 / 组

你可以根据表达式对枚举进行分组。

同义词：

```
Group-Object
group
```

示例：

```
$names = @( "Aaron", "Albert", "Alphonse","Bernie", "Charlie", "Danny", "Ernie", "Frank")

$names | Group-Object -Property Length
$names | group -Property Length
```

响应：

计数	名称	组
4	5	{Aaron, Danny, Ernie, Frank}
2	6	{阿尔伯特，伯尼}
1	8	{阿方斯}
1	7	{查理}

```
Charlie
Danny
```

To request descending order:

```
$names | Sort-Object -Descending
$names | sort -Descending
```

```
Danny
Charlie
Bernie
Aaron
Aaron
```

You can sort using an expression.

```
$names | Sort-Object { $_.length }
```

```
Aaron
Aaron
Danny
Bernie
Charlie
```

Section 5.3: Grouping: Group-Object / group

You can group an enumeration based on an expression.

Synonyms:

```
Group-Object
group
```

Examples:

```
$names = @( "Aaron", "Albert", "Alphonse","Bernie", "Charlie", "Danny", "Ernie", "Frank")

$names | Group-Object -Property Length
$names | group -Property Length
```

Response:

Count	Name	Group
4	5	{Aaron, Danny, Ernie, Frank}
2	6	{Albert, Bernie}
1	8	{Alphonse}
1	7	{Charlie}

第5.4节：投影：Select-Object / select

对枚举进行投影可以让你提取每个对象的特定成员，提取所有细节，或计算每个对象的值

同义词：

```
Select-Object
SELECT
```

选择属性的子集：

```
$dir = dir "C:\MyFolder"

$dir | Select-Object Name, FullName, Attributes
$dir | select Name, FullName, Attributes
```

Name 全名 属性

图片C:\MyFolder\ImagesDirectory

data.txtC:\MyFolder\data.txt归档

source.c C:\MyFolder\source.c 归档

选择第一个元素，并显示其所有属性：

```
$d | select -first 1 *
```

PS路径
PS父路径
PS子名称
PS驱动器
PS提供者
PS是否容器
基本名称
模式
Name
父级
存在
根
全名
扩展名
创建时间
创建时间（UTC）
最后访问时间
最后访问时间（UTC）
最后写入时间
最后写入时间（UTC）
属性

Section 5.4: Projecting: Select-Object / select

Projecting an enumeration allows you to extract specific members of each object, to extract all the details, or to compute values for each object

Synonyms:

```
Select-Object
SELECT
```

Selecting a subset of the properties:

```
$dir = dir "C:\MyFolder"

$dir | Select-Object Name, FullName, Attributes
$dir | select Name, FullName, Attributes
```

Name	FullName	Attributes
Images	C:\MyFolder\Images	Directory
data.txt	C:\MyFolder\data.txt	Archive
source.c	C:\MyFolder\source.c	Archive

Selecting the first element, and show all its properties:

```
$d | select -first 1 *
```

PSPath
PSParentPath
PSChildName
PSDrive
PSProvider
PSIsContainer
BaseName
Mode
Name
Parent
Exists
Root
FullName
Extension
CreationTime
CreationTimeUtc
LastAccessTime
LastAccessTimeUtc
LastWriteTime
LastWriteTimeUtc
Attributes

第6章：条件逻辑

第6.1节：if、else和else if

Powershell支持标准的条件逻辑运算符，类似于许多编程语言。这些运算符允许在特定情况下运行某些函数或命令。

使用if时，只有当if(())中的条件满足时，括号内的命令（{}）才会被执行

```
$test = "test"
if ($test -eq "test"){
    Write-Host "if 条件满足"
}
```

你也可以使用else。当if条件不满足时，else中的命令将被执行：

```
$test = "test"
if ($test -eq "test2"){
    Write-Host "if 条件满足"
}
else{
    Write-Host "if condition not met"
}
```

或者一个elseif。else if 会在if条件不满足且elseif条件满足时运行命令：

```
$test = "test"
if ($test -eq "test2"){
    Write-Host "if 条件满足"
}
elseif ($test -eq "test"){
    Write-Host "ifelse 条件满足"
}
```

注意上面使用的是-eq（等于）CmdLet，而不是许多其他语言中用于等于的=或==。

第6.2节：取反

你可能想要对布尔值取反，即当条件为假而非真时进入if语句。这可以通过使用-Not CmdLet来实现

```
$test = "test"
if (-Not $test -eq "test2"){
    Write-Host "if 条件不满足"
}
```

你也可以使用!：

```
$test = "test"
if (!($test -eq "test2")){
    Write-Host "if 条件不满足"
}
```

还有-ne（不等于）运算符：

Chapter 6: Conditional logic

Section 6.1: if, else and else if

Powershell supports standard conditional logic operators, much like many programming languages. These allow certain functions or commands to be run under particular circumstances.

With an `if` the commands inside the brackets (`{}`) are only executed if the conditions inside the if(`()`) are met

```
$test = "test"
if ($test -eq "test"){
    Write-Host "if condition met"
}
```

You can also do an `else`. Here the `else` commands are executed if the `if` conditions are **not** met:

```
$test = "test"
if ($test -eq "test2"){
    Write-Host "if condition met"
}
else{
    Write-Host "if condition not met"
}
```

or an `elseif`. An else if runs the commands if the `if` conditions are not met and the `elseif` conditions are met:

```
$test = "test"
if ($test -eq "test2"){
    Write-Host "if condition met"
}
elseif ($test -eq "test"){
    Write-Host "ifelse condition met"
}
```

Note the above use `-eq`(equality) CmdLet and not `=` or `==` as many other languages do for equality.

Section 6.2: Negation

You may want to negate a boolean value, i.e. enter an `if` statement when a condition is false rather than true. This can be done by using the `-Not` CmdLet

```
$test = "test"
if (-Not $test -eq "test2"){
    Write-Host "if condition not met"
}
```

You can also use `!`:

```
$test = "test"
if (!($test -eq "test2")){
    Write-Host "if condition not met"
}
```

there is also the `-ne` (not equal) operator:

```
$test = "test"
if ($test -ne "test2"){
    Write-Host "变量测试不等于 'test2'"
}
```

第6.3节：If 条件简写

如果你想使用简写，可以使用以下简写的条件逻辑。只有字符串 'false' 会被评估为真（2.0版本）。

```
# 在Powershell 2.0中完成
$boolean = $false;
$string = "false";
$emptyString = "";

If($boolean){
    # 由于 $boolean 为假，此处不会执行
    Write-Host "简写的If条件很方便，只要确保它们总是布尔值即可。"
}

如果($string){
    # 这段代码会运行，因为字符串长度非零
    Write-Host "如果变量不是严格的null或布尔值false，它将被评估为true，
因为它是一个对象或长度大于0的字符串。"
}

If($emptyString){
    # 这段代码不会运行，因为字符串长度为零
    Write-Host "检查空字符串也很有用。"
}

If($null){
    # 这段代码不会运行，因为条件为null
    Write-Host "检查Null不会打印这条语句。"
}
```

```
$test = "test"
if ($test -ne "test2"){
    Write-Host "variable test is not equal to 'test2'"
}
```

Section 6.3: If conditional shorthand

If you want to use the shorthand you can make use of conditional logic with the following shorthand. Only the string 'false' will evaluate to true (2.0).

```
#Done in Powershell 2.0
$boolean = $false;
$string = "false";
$emptyString = "";

If($boolean){
    # this does not run because $boolean is false
    Write-Host "Shorthand If conditions can be nice, just make sure they are always boolean."
}

If($string){
    # This does run because the string is non-zero length
    Write-Host "If the variable is not strictly null or Boolean false, it will evaluate to true as
it is an object or string with length greater than 0."
}

If($emptyString){
    # This does not run because the string is zero-length
    Write-Host "Checking empty strings can be useful as well."
}

If($null){
    # This does not run because the condition is null
    Write-Host "Checking Nulls will not print this statement."
}
```

第7章：循环

循环是一系列指令的重复执行，直到达到某个条件。能够让程序重复执行一段代码块是编程中最基本且最有用的任务之一。循环让你通过简单的语句，通过重复产生显著更大的结果。如果条件已满足，下一条指令将“穿透”到下一个顺序指令或跳出循环。

第7.1节：Foreach

Foreach在PowerShell中有两种不同的含义。一种是关键字，另一种是ForEach-Object cmdlet的别名。这里描述的是前者。

此示例演示如何将数组中的所有项打印到控制台主机：

```
$Names = @('艾米', '鲍勃', '塞琳', '大卫')

ForEach ($Name in $Names)
{
    Write-Host "嗨，我的名字是 $Name！"
}
```

此示例演示如何捕获 ForEach 循环的输出：

```
$Numbers = ForEach ($Number in 1..20) {
    $Number # 或者，使用 Write-Output $Number
}
```

与上一个示例类似，此示例演示在存储循环之前创建数组：

```
$Numbers = @()
ForEach ($Number in 1..20)
{
    $Numbers += $Number
}
```

第7.2节：For 循环

```
for($i = 0; $i -le 5; $i++){
    "$i"
}
```

for 循环的一个典型用法是对数组中的部分值进行操作。在大多数情况下，如果你想遍历数组中的所有值，建议使用 for each 语句。

第7.3节：ForEach() 方法

版本 > 4.0

除了ForEach-Object命令外，还可以直接在对象数组上使用ForEach方法，如下所示

```
(1..10).ForEach({$_ * $_})
```

Chapter 7: Loops

A loop is a sequence of instruction(s) that is continually repeated until a certain condition is reached. Being able to have your program repeatedly execute a block of code is one of the most basic but useful tasks in programming. A loop lets you write a very simple statement to produce a significantly greater result simply by repetition. If the condition has been reached, the next instruction "falls through" to the next sequential instruction or branches outside the loop.

Section 7.1: Foreach

ForEach has two different meanings in PowerShell. One is a keyword and the other is an alias for the ForEach-Object cmdlet. The former is described here.

This example demonstrates printing all items in an array to the console host:

```
$Names = @('Amy', 'Bob', 'Celine', 'David')

ForEach ($Name in $Names)
{
    Write-Host "Hi, my name is $Name!"
}
```

This example demonstrates capturing the output of a ForEach loop:

```
$Numbers = ForEach ($Number in 1..20) {
    $Number # Alternatively, Write-Output $Number
}
```

Like the last example, this example, instead, demonstrates creating an array prior to storing the loop:

```
$Numbers = @()
ForEach ($Number in 1..20)
{
    $Numbers += $Number
}
```

Section 7.2: For

```
for($i = 0; $i -le 5; $i++){
    "$i"
}
```

A typical use of the for loop is to operate on a subset of the values in an array. In most cases, if you want to iterate all values in an array, consider using a foreach statement.

Section 7.3: ForEach() Method

Version > 4.0

Instead of the ForEach-Object cmdlet, the here is also the possibility to use a ForEach method directly on object arrays like so

```
(1..10).ForEach({$_ * $_})
```

或者 - 如果需要 - 可以省略脚本块周围的括号

```
(1..10).ForEach{$_ * $_}
```

两者都会产生以下输出结果

149162536496481100第7.4节：ForEach-ObjectForEach-Object命令的工作方式类似于 foreach 语句，但它从管道获取输入。

基本用法

```
$object | ForEach-Object {
    代码块
}
```

示例：

```
$names = @("Any","Bob","Celine","David")
$names | ForEach-Object {
    "你好，我的名字是 $_！"
}
```

Foreach-Object 有两个默认别名，foreach 和 %（简写语法）。最常用的是 %，因为 foreach 可能会与 foreach 语句混淆。示例：

```
$names | % {
    "你好，我的名字是 $_！"
}

$names | foreach {
    "你好，我的名字是 $_！"
}
```

高级用法

Foreach-Object 与其他 foreach 解决方案不同，因为它是一个 cmdlet，意味着它设计用于管道。正因为如此，它支持三个脚本块，就像 cmdlet 或高级函数一样：

- Begin: 在循环处理来自管道的项目之前执行一次。通常用于创建循环中使用的函数、创建变量、打开连接（数据库、网络等）等。
- Process: 针对管道中到达的每个项目执行一次。“普通”的 foreach 代码块。当未指定参数时，上述示例中默认使用此代码块。
- End: 在处理完所有项目后执行一次。通常用于关闭连接、生成报告等。

or - if desired - the parentheses around the script block can be omitted

```
(1..10).ForEach{$_ * $_}
```

Both will result in the output below

```
1
4
9
16
25
36
49
64
81
100
```

Section 7.4: ForEach-Object

The ForEach-Object cmdlet works similarly to the foreach statement, but takes its input from the pipeline.

Basic usage

```
$object | ForEach-Object {
    code_block
}
```

Example:

```
$names = @("Any","Bob","Celine","David")
$names | ForEach-Object {
    "Hi, my name is $_!"
}
```

Foreach-Object has two default aliases, foreach and % (shorthand syntax). Most common is % because foreach can be confused with the foreach statement. Examples:

```
$names | % {
    "Hi, my name is $_!"
}

$names | foreach {
    "Hi, my name is $_!"
}
```

Advanced usage

Foreach-Object stands out from the alternative foreach solutions because it's a cmdlet which means it's designed to use the pipeline. Because of this, it has support for three scriptblocks just like a cmdlet or advanced function:

- **Begin**: Executed once before looping through the items that arrive from the pipeline. Usually used to create functions for use in the loop, creating variables, opening connections (database, web +) etc.
- **Process**: Executed once per item arrived from the pipeline. "Normal" foreach codeblock. This is the default used in the examples above when the parameter isn't specified.
- **End**: Executed once after processing all items. Usually used to close connections, generate a report etc.

示例：

```
"Any","Bob","Celine","David" | ForEach-Object -Begin {
    $results = @()
} -过程 {
    #创建并存储消息
    $results += "嗨，我的名字是 $_！"
} -结束 {
    #计算消息数量并输出
    Write-Host "消息总数：$($results.Count)"
    $results
}
```

第7.5节：继续

Continue 操作符适用于 For、ForEach、While 和 Do 循环。它跳过当前循环的迭代，跳转到最内层循环的顶部。

```
$i =0
while ($i -lt 20) {
    $i++
    if ($i -eq 7) { continue }
    Write-Host $I
}
```

上述代码将输出1到20到控制台，但会跳过数字7。

注意：使用管道循环时，应使用return而不是Continue。

第7.6节：Break（中断）

break操作符会立即退出程序循环。它可以用于For、ForEach、While和Do循环，或在Switch语句中使用。

```
$i = 0
while ($i -lt 15) {
    $i++
    if ($i -eq 7) {break}
    Write-Host $i
}
```

上述代码将计数到15，但一旦达到7就停止。

注意：使用管道循环时，break的行为类似于continue。要在管道循环中模拟break，需要加入额外的逻辑、cmdlet等。如果需要使用break，最好坚持使用非管道循环。

Break标签

Break也可以调用放置在循环实例前的标签：

```
$i = 0
:mainLoop 当 ($i -lt 15) {
    Write-Host $i -前景色 'Cyan'
    $j = 0
    当 ($j 小于 15) {
        Write-Host $j -前景色 '品红色'
```

Example:

```
"Any","Bob","Celine","David" | ForEach-Object -Begin {
    $results = @()
} -Process {
    #Create and store message
    $results += "Hi, my name is $_!"
} -End {
    #Count messages and output
    Write-Host "Total messages: $($results.Count)"
    $results
}
```

Section 7.5: Continue

The Continue operator works in For, ForEach, While and Do loops. It skips the current iteration of the loop, jumping to the top of the innermost loop.

```
$i =0
while ($i -lt 20) {
    $i++
    if ($i -eq 7) { continue }
    Write-Host $I
}
```

The above will output 1 to 20 to the console but miss out the number 7.

Note: When using a pipeline loop you should use return instead of Continue.

Section 7.6: Break

The break operator will exit a program loop immediately. It can be used in For, ForEach, While and Do loops or in a Switch Statement.

```
$i = 0
while ($i -lt 15) {
    $i++
    if ($i -eq 7) {break}
    Write-Host $i
}
```

The above will count to 15 but stop as soon as 7 is reached.

Note: When using a pipeline loop, break will behave as continue. To simulate break in the pipeline loop you need to incorporate some additional logic, cmdlet, etc. It is easier to stick with non-pipeline loops if you need to use break.

Break Labels

Break can also call a label that was placed in front of the instantiation of a loop:

```
$i = 0
:mainLoop While ($i -lt 15) {
    Write-Host $i -ForegroundColor 'Cyan'
    $j = 0
    While ($j -lt 15) {
        Write-Host $j -ForegroundColor 'Magenta'
        $k = $i*$j
        Write-Host $k -前景色 '绿色'
        如果 ($k 大于 100) {
            跳出 mainLoop
        }
        $j++
    }
    $i++
}
```

注意： 这段代码会将 $i 增加到 8，$j 增加到 13，导致 $k 等于 104。由于 $k 超过了 100，代码将跳出两个循环。

第7.7节：While循环

while循环会评估一个条件，如果为真则执行一个动作。只要条件评估为真，动作就会持续执行。

```
while(condition){
  code_block
}
```

以下示例创建了一个从10倒数到0的循环

```
$i = 10
while($i -ge 0){
    $i
    $i--
}
```

与Do-While循环不同，条件在动作首次执行前进行评估。如果初始条件为假，动作将不会执行。

注意：在评估条件时，PowerShell会将返回对象的存在视为真。这可以有多种用法，下面是一个监控进程的示例。该示例将启动一个记事本进程，然后只要该进程运行，当前shell就会休眠。当你手动关闭记事本实例时，while条件将失败，循环将中断。

```
Start-Process notepad.exe
while(Get-Process notepad -ErrorAction SilentlyContinue){
  Start-Sleep -Milliseconds 500
}
```

第7.8节：Do

Do 循环在您总是希望至少执行一次代码块时非常有用。Do 循环会在执行代码块后评估条件，不同于在执行代码块之前评估条件的 while 循环。

你可以用两种方式使用do循环：

- 当条件为真时循环while：

  ```
  Do {
  代码块
  } while (条件)
  ```

```
        $k = $i*$j
        Write-Host $k -ForegroundColor 'Green'
        if ($k -gt 100) {
            break mainLoop
        }
        $j++
    }
    $i++
}
```

Note: This code will increment $i to 8 and $j to 13 which will cause $k to equal 104. Since $k exceed 100, the code will then break out of both loops.

Section 7.7: While

A while loop will evaluate a condition and if true will perform an action. As long as the condition evaluates to true the action will continue to be performed.

```
while(condition){
  code_block
}
```

The following example creates a loop that will count down from 10 to 0

```
$i = 10
while($i -ge 0){
    $i
    $i--
}
```

Unlike the Do-While loop the condition is evaluated prior to the action's first execution. The action will not be performed if the initial condition evaluates to false.

Note: When evaluating the condition, PowerShell will treat the existence of a return object as true. This can be used in several ways but below is an example to monitor for a process. This example will spawn a notepad process and then sleep the current shell as long as that process is running. When you manually close the notepad instance the while condition will fail and the loop will break.

```
Start-Process notepad.exe
while(Get-Process notepad -ErrorAction SilentlyContinue){
  Start-Sleep -Milliseconds 500
}
```

Section 7.8: Do

Do-loops are useful when you always want to run a codeblock at least once. A Do-loop will evaluate the condition after executing the codeblock, unlike a while-loop which does it before executing the codeblock.

You can use do-loops in two ways:

- Loop *while* the condition is true:

  ```
  Do {
      code_block
  } while (condition)
  ```

- 当条件为真时停止循环，换句话说，当条件为假时循环until：

```
Do {
代码块
} until (条件)
```

真实示例：

```
$i = 0

Do {
    $i++
    "数字 $i"
} while ($i -ne 3)

Do {
    $i++
    “数字 $i”
} 直到 ($i -eq 3)
```

Do-While和Do-Until是相反的循环。如果代码相同，条件将被反转。上面的示例说明了这种行为。

- Loop *until* the condition is true, in other words, loop while the condition is false:

```
Do {
    code_block
} until (condition)
```

Real Examples:

```
$i = 0

Do {
    $i++
    "Number $i"
} while ($i -ne 3)

Do {
    $i++
    "Number $i"
} until ($i -eq 3)
```

Do-While and Do-Until are antonymous loops. If the code inside the same, the condition will be reversed. The example above illustrates this behaviour.

第8章：Switch语句

Switch语句允许对变量进行等值测试，测试其是否等于一系列值中的某一个。每个值称为一个case，被switch的变量会针对每个switch case进行检查。它使你能够编写一个脚本，从一系列选项中选择，而无需编写冗长的if语句。

第8.1节：简单Switch

Switch语句将单个测试值与多个条件进行比较，并对成功匹配执行相应操作。它可能导致多个匹配/操作。

给出以下switch语句...

```
switch($myValue)
{
    ‘第一个条件’    { ‘第一个操作’ }
    ‘第二个条件’   { ‘第二个操作’ }
}
```

如果$myValue被设置为‘第一个条件’，‘第一个操作’将被输出。

如果$myValue 设置为'第二条件'，'Section Action' 将被输出。

如果$myValue不匹配任一条件，则不会输出任何内容。

第8.2节：带有CaseSensitive参数的Switch语句

-CaseSensitive参数强制switch语句对条件进行精确的区分大小写匹配。

示例：

```
switch -CaseSensitive ('Condition')
{
  'condition'   {'First Action'}
  'Condition'   {'Second Action'}
  'conditioN'   {'Third Action'}
}
```

输出：

```
第二个动作
```

第二个动作是唯一执行的动作，因为它是唯一一个在区分大小写时与字符串'Condition'完全匹配的条件。

第8.3节：带有Wildcard参数的Switch语句

-Wildcard参数允许switch语句对条件进行通配符匹配。

示例：

```
switch -通配符 ('条件')
{
```

Chapter 8: Switch statement

A switch statement allows a variable to be tested for equality against a list of values. Each value is called a *case*, and the variable being *switched* on is checked for each switch case. It enables you to write a script that can choose from a series of options, but without requiring you to write a long series of if statements.

Section 8.1: Simple Switch

Switch statements compare a single test value to multiple conditions, and performs any associated actions for successful comparisons. It can result in multiple matches/actions.

Given the following switch...

```
switch($myValue)
{
    'First Condition'    { 'First Action' }
    'Second Condition'   { 'Second Action' }
}
```

'First Action' will be output if $myValue is set as 'First Condition'.

'Section Action' will be output if $myValue is set as 'Second Condition'.

Nothing will be output if $myValue does not match either conditions.

Section 8.2: Switch Statement with CaseSensitive Parameter

The -CaseSensitive parameter enforces switch statements to perform exact, case-sensitive matching against conditions.

Example:

```
switch -CaseSensitive ('Condition')
{
  'condition'   {'First Action'}
  'Condition'   {'Second Action'}
  'conditioN'   {'Third Action'}
}
```

Output:

```
Second Action
```

The second action is the only action executed because it is the only condition that exactly matches the string 'Condition' when accounting for case-sensitivity.

Section 8.3: Switch Statement with Wildcard Parameter

The -Wildcard parameter allows switch statements to perform wildcard matching against conditions.

Example:

```
switch -Wildcard ('Condition')
{
    '条件'          {'普通匹配'}
    'Condit*'               {'零个或多个通配符字符。'}
    'C[aoc]ndit[f-l]on'     {'字符范围和集合。'}
    'C?ndition'             {'单个字符通配符'}
    'Test*'                 {'不匹配'}
}
```

输出：

```
普通匹配
零个或多个通配符字符。
字符范围和集合。
单个字符通配符
```

第8.4节：带文件参数的switch语句

-file参数允许switch语句从文件接收输入。文件的每一行都会被switch语句评估。

示例文件input.txt：

```
condition
test
```

示例开关语句：

```
switch -file input.txt
{
  'condition’ {‘第一动作’}
  ‘test’       {‘第二动作’}
  ‘fail’       {‘第三动作’}
}
```

输出：

```
第一动作
第二动作
```

第8.5节：带默认条件的简单switch

关键字Default用于在没有其他条件匹配输入值时执行某个动作。

示例：

```
switch(‘条件’)
{
  ‘跳过条件’
  {
    ‘第一动作’
  }
  ‘跳过此条件’
  {
    ‘第二动作’
  }
默认
  {
```

```
    'Condition'             {'Normal match'}
    'Condit*'               {'Zero or more wildcard chars.'}
    'C[aoc]ndit[f-l]on'     {'Range and set of chars.'}
    'C?ndition'             {'Single char. wildcard'}
    'Test*'                 {'No match'}
}
```

Output:

```
Normal match
Zero or more wildcard chars.
Range and set of chars.
Single char. wildcard
```

Section 8.4: Switch Statement with File Parameter

The -file parameter allows the switch statement to receive input from a file. Each line of the file is evaluated by the switch statement.

Example file input.txt:

```
condition
test
```

Example switch statement:

```
switch -file input.txt
{
  'condition' {'First Action'}
  'test'      {'Second Action'}
  'fail'      {'Third Action'}
}
```

Output:

```
First Action
Second Action
```

Section 8.5: Simple Switch with Default Condition

The Default keyword is used to execute an action when no other conditions match the input value.

Example:

```
switch('Condition')
{
  'Skip Condition'
  {
    'First Action'
  }
  'Skip This Condition Too'
  {
    'Second Action'
  }
  Default
  {
    ‘默认操作’
  }
}
```

输出：

```
默认操作
```

第8.6节：带正则表达式参数的switch语句

该-Regex参数允许switch语句针对条件执行正则表达式匹配。

示例：

```
switch -Regex (‘条件’)
{
  ‘Con\D+ion’     {‘一个或多个非数字字符’}
  ‘Conditio*$’    {‘零个或多个“o”字符’}
  ‘C.ndition’     {‘任意单个字符’}
  ‘^C\w+ition$’   {‘锚点和一个或多个单词字符’}
  ‘Test’          {‘不匹配’}
}
```

输出：

```
一个或多个非数字字符
任意单个字符。
锚点和一个或多个单词字符。
```

第8.7节：带断路器的简单开关

break关键字可以用于switch语句中，在评估所有条件之前退出语句。

示例：

```
switch(‘条件’)
{
  ‘Condition’
  {
    ‘第一动作’
  }
  ‘Condition’
  {
    ‘Second Action’
    break
  }
  ‘Condition’
  {
    ‘Third Action’
  }
}
```

输出：

```
第一动作
第二动作
```

```
    'Default Action'
  }
}
```

Output:

```
Default Action
```

Section 8.6: Switch Statement with Regex Parameter

The -Regex parameter allows switch statements to perform regular expression matching against conditions.

Example:

```
switch -Regex ('Condition')
{
  'Con\D+ion'    {'One or more non-digits'}
  'Conditio*$'   {'Zero or more "o"'}
  'C.ndition'    {'Any single char.'}
  '^C\w+ition$'  {'Anchors and one or more word chars.'}
  'Test'         {'No match'}
}
```

Output:

```
One or more non-digits
Any single char.
Anchors and one or more word chars.
```

Section 8.7: Simple Switch With Break

The break keyword can be used in switch statements to exit the statement before evaluating all conditions.

Example:

```
switch('Condition')
{
  'Condition'
  {
    'First Action'
  }
  'Condition'
  {
    'Second Action'
    break
  }
  'Condition'
  {
    'Third Action'
  }
}
```

Output:

```
First Action
Second Action
```

由于第二个动作中的break关键字，第三个条件未被评估。

第8.8节：带有精确参数的Switch语句

-Exact参数强制switch语句对字符串条件进行精确且不区分大小写的匹配。

示例：

```
switch -Exact ('Condition')
{
  ‘condition’    {‘First Action’}
  ‘Condition’    {‘Second Action’}
  ‘conditioN’    {‘Third Action’}
  ‘^*ondition$’ {‘Fourth Action’}
  ‘Conditio*’    {‘Fifth Action’}
}
```

输出：

```
First Action
Second Action
Third Action
```

前面三条操作被执行，因为它们关联的条件与输入匹配。第四和第五个条件中的正则表达式和通配符字符串匹配失败。

注意，如果执行正则表达式匹配，第四个条件也会匹配输入字符串，但在本例中未执行正则匹配，因此被忽略。

第8.9节：带表达式的Switch语句

条件也可以是表达式：

```
$myInput = 0

switch($myInput) {
    # 因为表达式的结果4
    # 不等于我们的输入，所以该代码块不应执行。
    (2+2)  { '正确。2 + 2 = 4' }

    # 因为表达式的结果0
    # 等于我们的输入，所以该代码块应执行。
    (2-2) { '正确。2 - 2 = 0' }

    # 因为我们的输入大于-1且小于1
    # 表达式结果为真，代码块应执行。
    { $_ -gt -1 -and $_ -lt 1 } { '真。值为0' }
}

#输出
真。 2-2 = 0
真。值为 0
```

Because of the break keyword in the second action, the third condition is not evaluated.

Section 8.8: Switch Statement with Exact Parameter

The -Exact parameter enforces switch statements to perform exact, case-insensitive matching against string-conditions.

Example:

```
switch -Exact ('Condition')
{
  'condition'    {'First Action'}
  'Condition'    {'Second Action'}
  'conditioN'    {'Third Action'}
  '^*ondition$'  {'Fourth Action'}
  'Conditio*'    {'Fifth Action'}
}
```

Output:

```
First Action
Second Action
Third Action
```

The first through third actions are executed because their associated conditions matched the input. The regex and wildcard strings in the fourth and fifth conditions fail matching.

Note that the fourth condition would also match the input string if regular expression matching was being performed, but was ignored in this case because it is not.

Section 8.9: Switch Statement with Expressions

Conditions can also be expressions:

```
$myInput = 0

switch($myInput) {
    # because the result of the expression, 4,
    # does not equal our input this block should not be run.
    (2+2)  { 'True. 2 +2 = 4' }

    # because the result of the expression, 0,
    # does equal our input this block should be run.
    (2-2) { 'True. 2-2 = 0' }

    # because our input is greater than -1 and is less than 1
    # the expression evaluates to true and the block should be run.
    { $_ -gt -1 -and $_ -lt 1 } { 'True. Value is 0' }
}

#Output
True. 2-2 = 0
True. Value is 0
```

第9章：字符串

第9.1节：多行字符串

在PowerShell中有多种方法创建多行字符串：

- 你可以手动使用回车和/或换行的特殊字符，或者使用NewLine-环境变量插入系统的“换行”值）

```
"Hello`r`nWorld"
"Hello{0}World" -f [environment]::NewLine
```

- 在定义字符串时创建换行（在结束引号前）

```
“你好
世界”
```

- *使用 here-string。这是最常用的技术。*

```
@“
你好
世界
”@
```

第9.2节：Here-string

Here-string 在创建多行字符串时非常有用。与其他多行字符串相比，其最大优点之一是你可以使用引号而无需使用反引号进行转义。

Here-string

Here-string 以@“和换行符开始，以”@单独一行结束（”@必须是该行的第一个字符，甚至不能有空格/制表符）。

```
@“
简单
多行字符串
带有“引号”
"@
```

字面量 here-string

当你不想让任何表达式被展开，就像普通的字面量字符串一样，你也可以使用单引号来创建字面量 here-string。

```
@'
以下行不会被展开
$(Get-Date)
因为这是一个字面量 here-string
'@
```

第9.3节：字符串连接

在字符串中使用变量

Chapter 9: Strings

Section 9.1: Multiline string

There are multiple ways to create a multiline string in PowerShell:

- You can use the special characters for carriage return and/or newline manually or use the NewLine-environment variable to insert the systems "newline" value)

```
"Hello`r`nWorld"
"Hello{0}World" -f [environment]::NewLine
```

- Create a linebreak while defining a string (before closing quote)

```
"Hello
World"
```

- Using a here-string. *This is the most common technique.*

```
@"
Hello
World
"@
```

Section 9.2: Here-string

Here-strings are very useful when creating multiline strings. One of the biggest benefits compared to other multiline strings are that you can use quotes without having to escape them using a backtick.

Here-string

Here-strings begin with @" and a linebreak and end with "@ on its own line (**"@must be first characters on the line, not even whitespace/tab**).

```
@"
Simple
    Multiline string
with "quotes"
"@
```

Literal here-string

You could also create a literal here-string by using single quotes, when you don't want any expressions to be expanded just like a normal literal string.

```
@'
The following line won't be expanded
$(Get-Date)
because this is a literal here-string
'@
```

Section 9.3: Concatenating strings

Using variables in a string

你可以在双引号字符串中使用变量来连接字符串。这对属性不起作用。

```
$string1 = "Power"
$string2 = "Shell"
"来自 $string1$string2 的问候"
```

使用 + 运算符

你也可以使用+运算符连接字符串。

```
$string1 = "来自"
$string2 = "PowerShell"
$string1 + " " + $string2
```

这也适用于对象的属性。

```
"此控制台的标题是 '" + $host.Name + "'"
```

使用子表达式

子表达式$()的输出/结果可以用于字符串中。当访问对象的属性或执行复杂表达式时，这非常有用。子表达式可以包含由分号;分隔的多条语句。

```
"明天是 $((Get-Date).AddDays(1).DayOfWeek)"
```

第9.4节：特殊字符

当在双引号字符串内使用时，转义字符（反引号`）表示特殊字符。

```
`0    #空值
`a    #Alert/Beep
`b    #退格键
`f      #换页符（用于打印输出）
`n      #换行符
`r    #回车符
`t    #水平制表符
`v    #垂直制表符（用于打印输出）
```

示例：

```
> "这个`t使用`t制表符`r`n这是第二行"
这个      使用      制表符
这是第二行
```

你也可以转义具有特殊含义的字符：

```
`#     #注释操作符
`$      #变量操作符
``      #转义字符
`'    #单引号
`"     #双引号
```

第9.5节：创建基本字符串

字符串

You can concatenate strings using variables inside a double-quoted string. This does not work with properties.

```
$string1 = "Power"
$string2 = "Shell"
"Greetings from $string1$string2"
```

Using the + operator

You can also join strings using the + operator.

```
$string1 = "Greetings from"
$string2 = "PowerShell"
$string1 + " " + $string2
```

This also works with properties of objects.

```
"The title of this console is '" + $host.Name + "'"
```

Using subexpressions

The output/result of a subexpressions $() can be used in a string. This is useful when accessing properties of an object or performing a complex expression. Subexpressions can contain multiple statements separated by semicolon ;

```
"Tomorrow is $((Get-Date).AddDays(1).DayOfWeek)"
```

Section 9.4: Special characters

When used inside a double-quoted string, the escape character (backtick `) represents a special character.

```
`0    #Null
`a    #Alert/Beep
`b    #Backspace
`f    #Form feed (used for printer output)
`n    #New line
`r    #Carriage return
`t    #Horizontal tab
`v    #Vertical tab (used for printer output)
```

Example:

```
> "This`tuses`ttab`r`nThis is on a second line"
This    uses    tab
This is on a second line
```

You can also escape special characters with special meanings:

```
`#    #Comment-operator
`$    #Variable operator
``    #Escape character
`'    #Single quote
`"    #Double quote
```

Section 9.5: Creating a basic string

String

字符串是通过用双引号包裹文本来创建的。双引号字符串可以计算变量和特殊字符。

```
$myString = "一些基本文本"
$mySecondString = "包含 $variable 的字符串"
```

要在字符串中使用双引号，需要使用转义字符反引号（`）进行转义。单引号可以在双引号字符串中使用。

```
$myString = "一个包含 `"双引号`" 的字符串，同时还有 '单引号'。"
```

字面字符串

字面字符串是不计算变量和特殊字符的字符串。它是用单引号创建的。

```
$myLiteralString = '包含特殊字符（`n）和 $variable 引用的简单文本'
```

要在字面字符串中使用单引号，可以使用两个单引号或字面 here-string。双引号可以安全地用于字面字符串中。

```
$myLiteralString = '包含 ''单引号'' 和 "双引号" 的简单字符串。'
```

第9.6节：格式字符串

```
$hash = @{ city = '柏林' }

$result = '你真的应该去访问 {0}' -f $hash.city
Write-Host $result #输出 "你真的应该去访问 柏林"
```

格式字符串可以与 `-f` 运算符或静态的 `[String]::Format(string format, args)` .NET 方法一起使用。

Strings are created by wrapping the text with double quotes. Double-quoted strings can evaluate variables and special characters.

```
$myString = "Some basic text"
$mySecondString = "String with a $variable"
```

To use a double quote inside a string it needs to be escaped using the escape character, backtick (`). Single quotes can be used inside a double-quoted string.

```
$myString = "A `"double quoted`" string which also has 'single quotes'."
```

Literal string

Literal strings are strings that doesn't evaluate variables and special characters. It's created using single quotes.

```
$myLiteralString = 'Simple text including special characters (`n) and a $variable-reference'
```

To use single quotes inside a literal string, use double single quotes or a literal here-string. Double quotes can be used safely inside a literal string

```
$myLiteralString = 'Simple string with ''single quotes'' and "double quotes".'
```

Section 9.6: Format string

```
$hash = @{ city = 'Berlin' }

$result = 'You should really visit {0}' -f $hash.city
Write-Host $result #prints "You should really visit Berlin"
```

Format strings can be used with the `-f` operator or the static `[String]::Format(string format, args)` .NET method.

第10章：哈希表

哈希表是一种将键映射到值的结构。详情见 哈希表 。

第10.1节：通过键访问哈希表的值

定义哈希表并通过键访问值的示例

```
$hashTable = @{
    Key1 = '值1'
    Key2 = '值2'
}
$hashTable.Key1
#输出
值1
```

访问属性名中包含无效字符的键的示例：

```
$hashTable = @{
    'Key 1' = 'Value3'
    Key2 = 'Value4'
}
$hashTable.'Key 1'
#输出
Value3
```

第10.2节：创建哈希表

创建空哈希表的示例：

```
$hashTable = @{}
```

创建带数据的哈希表示例：

```
$hashTable = @{
    Name1 = 'Value'
    Name2 = 'Value'
    Name3 = 'Value3'
}
```

第10.3节：向现有哈希表添加键值对

例如，使用加法运算符向哈希表添加一个键为“Key2”，值为“Value2”的项：

```
$hashTable = @{
    Key1 = 'Value1'
}
$hashTable += @{Key2 = 'Value2'}
$hashTable

#输出

Name                           Value
----                           -----
Key1                           Value1
```

Chapter 10: HashTables

A Hash Table is a structure which maps keys to values. See Hash Table for details.

Section 10.1: Access a hash table value by key

An example of defining a hash table and accessing a value by the key

```
$hashTable = @{
    Key1 = 'Value1'
    Key2 = 'Value2'
}
$hashTable.Key1
#output
Value1
```

An example of accessing a key with invalid characters for a property name:

```
$hashTable = @{
    'Key 1' = 'Value3'
    Key2 = 'Value4'
}
$hashTable.'Key 1'
#Output
Value3
```

Section 10.2: Creating a Hash Table

Example of creating an empty HashTable:

```
$hashTable = @{}
```

Example of creating a HashTable with data:

```
$hashTable = @{
    Name1 = 'Value'
    Name2 = 'Value'
    Name3 = 'Value3'
}
```

Section 10.3: Add a key value pair to an existing hash table

An example, to add a "Key2" key with a value of "Value2" to the hash table, using the addition operator:

```
$hashTable = @{
    Key1 = 'Value1'
}
$hashTable += @{Key2 = 'Value2'}
$hashTable

#Output

Name                           Value
----                           -----
Key1                           Value1
```

```
Key2                           Value2
```

示例，使用 Add 方法向哈希表添加一个键为 "Key2"，值为 "Value2" 的项：

```
$hashTable = @{
    Key1 = 'Value1'
}
$hashTable.Add("Key2", "Value2")
$hashTable

#输出

Name                           Value
----                           -----
Key1                           Value1
Key2                           Value2
```

第10.4节：从现有哈希表中移除一个键值对

示例，使用移除操作符从哈希表中移除键为 "Key2"，值为 "Value2" 的项：

```
$hashTable = @{
    Key1 = '值1'
    Key2 = '值2'
}
$hashTable.Remove("Key2", "Value2")
$hashTable

#输出

Name                           Value
----                           -----
Key1                           Value1
```

第10.5节：遍历键和键值对

遍历键

```
foreach ($key in $var1.Keys) {
    $value = $var1[$key]
    # 或者
    $value = $var1.$key
}
```

遍历键值对

```
foreach ($keyvaluepair in $var1.GetEnumerator()) {
    $key1 = $_.Key1
    $val1 = $_.Val1
}
```

第10.6节：遍历哈希表

```
$hashTable = @{
    Key1 = 'Value1'
    Key2 = 'Value2'
```

```
Key2                           Value2
```

An example, to add a "Key2" key with a value of "Value2" to the hash table using the Add method:

```
$hashTable = @{
    Key1 = 'Value1'
}
$hashTable.Add("Key2", "Value2")
$hashTable

#Output

Name                           Value
----                           -----
Key1                           Value1
Key2                           Value2
```

Section 10.4: Remove a key value pair from an existing hash table

An example, to remove a "Key2" key with a value of "Value2" from the hash table, using the remove operator:

```
$hashTable = @{
    Key1 = 'Value1'
    Key2 = 'Value2'
}
$hashTable.Remove("Key2", "Value2")
$hashTable

#Output

Name                           Value
----                           -----
Key1                           Value1
```

Section 10.5: Enumerating through keys and Key-Value Pairs

Enumerating through Keys

```
foreach ($key in $var1.Keys) {
    $value = $var1[$key]
    # or
    $value = $var1.$key
}
```

Enumerating through Key-Value Pairs

```
foreach ($keyvaluepair in $var1.GetEnumerator()) {
    $key1 = $_.Key1
    $val1 = $_.Val1
}
```

Section 10.6: Looping over a hash table

```
$hashTable = @{
    Key1 = 'Value1'
    Key2 = 'Value2'
}

foreach($key in $hashTable.Keys)
{
    $value = $hashTable.$key
    Write-Output "$key : $value"
}
#输出
Key1 : Value1
Key2 : Value2
```

```
}

foreach($key in $hashTable.Keys)
{
    $value = $hashTable.$key
    Write-Output "$key : $value"
}
#Output
Key1 : Value1
Key2 : Value2
```

第11章：对象操作

第11.1节：检查对象

既然你已经有了一个对象，了解它是什么可能会很有帮助。你可以使用Get-Member命令来查看对象的类型以及它包含的内容：

```
Get-Item  c:\windows  |  Get-Member
```

结果如下：

```
TypeName:  System.IO.DirectoryInfo
```

后面跟着该对象拥有的属性和方法列表。

获取对象类型的另一种方法是使用 GetType 方法，如下所示：

```
C:\> $Object = Get-Item C:\Windows
C:\> $Object.GetType()

IsPublic IsSerial 名称                                    基类类型
-------- -------- ----                                    --------
True     True     DirectoryInfo                           System.IO.FileSystemInfo
```

要查看对象拥有的属性列表及其值，可以使用 Format-List 命令，并将其 Property 参数设置为：*（表示全部）。

下面是一个示例及其输出结果：

```
C:\> Get-Item  C:\Windows  |  Format-List  -Property  *
```

```
PSPath             : Microsoft.PowerShell.Core\FileSystem::C:\Windows
PSParentPath       : Microsoft.PowerShell.Core\FileSystem::C:\
PSChildName        : Windows
PSDrive            : C
PSProvider         : Microsoft.PowerShell.Core\FileSystem
PSIsContainer      : True
模式               : d-----
基本名称           : Windows
目标               : {}
链接类型           :
名称               : Windows
父级               :
存在               : True
根目录             : C:\
完整名称           : C:\Windows
扩展名             :
创建时间           : 30/10/2015  06:28:30
UTC创建时间        : 30/10/2015  06:28:30
最后访问时间       : 16/08/2016  17:32:04
UTC最后访问时间    : 16/08/2016  16:32:04
最后写入时间       : 16/08/2016  17:32:04
UTC最后写入时间    : 16/08/2016  16:32:04
属性               : 目录
```

Chapter 11: Working with Objects

Section 11.1: Examining an object

Now that you have an object, it might be good to figure out what it is. You can use the Get-Member cmdlet to see what an object is and what it contains:

```
Get-Item c:\windows | Get-Member
```

This yields:

```
TypeName: System.IO.DirectoryInfo
```

Followed by a list of properties and methods the object has.

Another way to get the type of an object is to use the GetType method, like so:

```
C:\> $Object = Get-Item C:\Windows
C:\> $Object.GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     DirectoryInfo                            System.IO.FileSystemInfo
```

To view a list of properties the object has, along with their values, you can use the Format-List cmdlet with its Property parameter set to: * (meaning all).

Here is an example, with the resulting output:

```
C:\> Get-Item C:\Windows | Format-List -Property *
```

```
PSPath            : Microsoft.PowerShell.Core\FileSystem::C:\Windows
PSParentPath      : Microsoft.PowerShell.Core\FileSystem::C:\
PSChildName       : Windows
PSDrive           : C
PSProvider        : Microsoft.PowerShell.Core\FileSystem
PSIsContainer     : True
Mode              : d-----
BaseName          : Windows
Target            : {}
LinkType          :
Name              : Windows
Parent            :
Exists            : True
Root              : C:\
FullName          : C:\Windows
Extension         :
CreationTime      : 30/10/2015 06:28:30
CreationTimeUtc   : 30/10/2015 06:28:30
LastAccessTime    : 16/08/2016 17:32:04
LastAccessTimeUtc : 16/08/2016 16:32:04
LastWriteTime     : 16/08/2016 17:32:04
LastWriteTimeUtc  : 16/08/2016 16:32:04
Attributes        : Directory
```

第11.2节：更新对象

添加属性

如果您想向现有对象添加属性，可以使用 Add-Member cmdlet。对于 PSObjects，值被
保存在一种“注释属性”（Note Properties）中

```
$object = New-Object -TypeName PSObject -Property @{
        名称 = $env:username
ID = 12
        地址 = $null
    }

Add-Member -InputObject $object -Name "SomeNewProp" -Value "一个值" -MemberType NoteProperty

# 返回
PS> $Object
名称 ID 地址 SomeNewProp
---- -- ------- -----------
nem  12         一个值
```

你也可以使用 Select-Object Cmdlet 添加属性（所谓的计算属性）：

```
$newObject = $Object | Select-Object *, @{label='SomeOtherProp'; expression={'另一个值'}}

# 返回
PS> $newObject
名称 ID 地址 SomeNewProp SomeOtherProp
---- -- ------- ----------- -------------
nem  12         一个值      另一个值
```

上面的命令可以简写为：

```
$newObject = $Object | Select *,@{l='SomeOtherProp';e={'另一个值'}}
```

移除属性

你可以使用 Select-Object Cmdlet 从对象中移除属性：

```
$object = $newObject | Select-Object * -ExcludeProperty ID, Address

# 返回值
PS> $object
Name SomeNewProp SomeOtherProp
---- ----------- -------------
nem  A value     Another value
```

第11.3节：创建新对象

PowerShell 与其他一些脚本语言不同，它通过管道传递对象。这意味着
当你将数据从一个命令发送到另一个命令时，能够创建、修改和收集对象是非常重要的。

创建对象很简单。你创建的大多数对象将在 PowerShell 中是自定义对象，使用的类型是 PSObject。PowerShell 还允许你创建任何你可以在 .NET 中创建的对象。

下面是一个创建带有几个属性的新对象的示例：

选项1：New-Object

Section 11.2: Updating Objects

Adding properties

If you'd like to add properties to an existing object, you can use the Add-Member cmdlet. With PSObjects, values are kept in a type of "Note Properties"

```
$object = New-Object -TypeName PSObject -Property @{
        Name = $env:username
        ID = 12
        Address = $null
    }

Add-Member -InputObject $object -Name "SomeNewProp" -Value "A value" -MemberType NoteProperty

# Returns
PS> $Object
Name ID Address SomeNewProp
---- -- ------- -----------
nem  12         A value
```

You can also add properties with Select-Object Cmdlet (so called calculated properties):

```
$newObject = $Object | Select-Object *, @{label='SomeOtherProp'; expression={'Another value'}}

# Returns
PS> $newObject
Name ID Address SomeNewProp SomeOtherProp
---- -- ------- ----------- -------------
nem  12         A value     Another value
```

The command above can be shortened to this:

```
$newObject = $Object | Select *,@{l='SomeOtherProp';e={'Another value'}}
```

Removing properties

You can use the Select-Object Cmdlet to remove properties from an object:

```
$object = $newObject | Select-Object * -ExcludeProperty ID, Address

# Returns
PS> $object
Name SomeNewProp SomeOtherProp
---- ----------- -------------
nem  A value     Another value
```

Section 11.3: Creating a new object

PowerShell, unlike some other scripting languages, sends objects through the pipeline. What this means is that when you send data from one command to another, it's essential to be able to create, modify, and collect objects.

Creating an object is simple. Most objects you create will be custom objects in PowerShell, and the type to use for that is PSObject. PowerShell will also allow you to create any object you could create in .NET.

Here's an example of creating a new objects with a few properties:

Option 1: New-Object

```
$newObject = New-Object -TypeName PSObject -Property @{
    Name = $env:username
ID = 12
    Address = $null
}

# 返回值
PS> $newObject
名称 ID 地址
---- -- -------
nem  12
```

你可以通过在命令前加上$newObject = 来将对象存储在变量中。

你可能还需要存储对象集合。这可以通过创建一个空的集合变量，
然后向集合中添加对象来实现，示例如下：

```
$newCollection = @()
$newCollection += New-Object -TypeName PSObject -Property @{
   名称 = $env:username
ID = 12
    Address = $null
}
```

然后你可能希望逐个遍历这个集合中的对象。要做到这一点，请查阅文档中的循环部分。

选项 2：Select-Object

你在网上仍然可以找到的一种较少见的创建对象的方法如下：

```
$newObject = 'unuseddummy' | Select-Object -Property 名称, ID, 地址
$newObject.名称 = $env:username
$newObject.ID = 12

# 返回值
PS> $newObject
名称 ID 地址
---- -- -------
nem  12
```

选项3：pscustomobject 类型加速器（需要 PSv3+）

有序类型加速器强制 PowerShell 按我们定义的顺序保留属性。你不需要有序类型加速器来使用[PSCustomObject]：

```
$newObject = [PSCustomObject][Ordered]@{
    Name = $env:Username
ID = 12
    Address = $null
}

# 返回值
PS> $newObject
名称 ID 地址
---- -- -------
nem  12
```

```
$newObject = New-Object -TypeName PSObject -Property @{
    Name = $env:username
    ID = 12
    Address = $null
}

# Returns
PS> $newObject
Name ID Address
---- -- -------
nem  12
```

You can store the object in a variable by prefacing the command with $newObject =

You may also need to store collections of objects. This can be done by creating an empty collection variable, and adding objects to the collection, like so:

```
$newCollection = @()
$newCollection += New-Object -TypeName PSObject -Property @{
    Name = $env:username
    ID = 12
    Address = $null
}
```

You may then wish to iterate through this collection object by object. To do that, locate the Loop section in the documentation.

Option 2: Select-Object

A less common way of creating objects that you'll still find on the internet is the following:

```
$newObject = 'unuseddummy' | Select-Object -Property Name, ID, Address
$newObject.Name = $env:username
$newObject.ID = 12

# Returns
PS> $newObject
Name ID Address
---- -- -------
nem  12
```

Option 3: pscustomobject type accelerator (PSv3+ required)

The ordered type accelerator forces PowerShell to keep our properties in the order that we defined them. You don't need the ordered type accelerator to use [PSCustomObject]:

```
$newObject = [PSCustomObject][Ordered]@{
    Name = $env:Username
    ID = 12
    Address = $null
}

# Returns
PS> $newObject
Name ID Address
---- -- -------
nem  12
```

第11.4节：创建泛型类的实例

注意：示例基于 PowerShell 5.1 编写，你可以创建泛型类的实例

```
#可空的 System.DateTime
[Nullable[datetime]]$nullableDate = Get-Date -Year 2012
$nullableDate
$nullableDate.GetType().FullName
$nullableDate = $null
$nullableDate

#普通的 System.DateTime
[datetime]$aDate = Get-Date -Year 2013
$aDate
$aDate.GetType().FullName
$aDate = $null #当 PowerShell 尝试将 null 转换为时抛出异常
```

输出结果为：

```
2012年8月4日 星期六 08:53:02
System.DateTime
2013年8月4日 星期日 08:53:02
System.DateTime
无法将 null 转换为类型 "System.DateTime"。
在第14行 字符:1
+ $aDate = $null
+ ~~~~~~~~~~~~~~
+ CategoryInfo          : MetadataError: (:) [], ArgumentTransformationMetadataException
+ FullyQualifiedErrorId : RuntimeException
```

泛型集合也是可能的

```
[System.Collections.Generic.SortedDictionary[int, String]]$dict =
[System.Collections.Generic.SortedDictionary[int, String]]::new()
$dict.GetType().FullName

$dict.Add(1, 'a')
$dict.Add(2, 'b')
$dict.Add(3, 'c')


$dict.Add('4', 'd') #powershell 自动将 '4' 转换为 4
$dict.Add('5.1', 'c') #powershell 自动将 '5.1' 转换为 5

$dict

$dict.Add('z', 'z') #powershell 无法将 'z' 转换为 System.Int32，因此抛出错误
```

输出结果为：

```
System.Collections.Generic.SortedDictionary`2[[System.Int32, mscorlib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089],[System.String, mscorlib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089]]

键 值
--- -----
1 a
2 b
```

Section 11.4: Creating Instances of Generic Classes

Note: examples written for PowerShell 5.1 You can create instances of Generic Classes

```
#Nullable System.DateTime
[Nullable[datetime]]$nullableDate = Get-Date -Year 2012
$nullableDate
$nullableDate.GetType().FullName
$nullableDate = $null
$nullableDate

#Normal System.DateTime
[datetime]$aDate = Get-Date -Year 2013
$aDate
$aDate.GetType().FullName
$aDate = $null #Throws exception when PowerShell attempts to convert null to
```

Gives the output:

```
Saturday, 4 August 2012 08:53:02
System.DateTime
Sunday, 4 August 2013 08:53:02
System.DateTime
Cannot convert null to type "System.DateTime".
At line:14 char:1
+ $aDate = $null
+ ~~~~~~~~~~~~~~
+ CategoryInfo          : MetadataError: (:) [], ArgumentTransformationMetadataException
+ FullyQualifiedErrorId : RuntimeException
```

Generic Collections are also possible

```
[System.Collections.Generic.SortedDictionary[int, String]]$dict =
[System.Collections.Generic.SortedDictionary[int, String]]::new()
$dict.GetType().FullName

$dict.Add(1, 'a')
$dict.Add(2, 'b')
$dict.Add(3, 'c')


$dict.Add('4', 'd') #powershell auto converts '4' to 4
$dict.Add('5.1', 'c') #powershell auto converts '5.1' to 5

$dict

$dict.Add('z', 'z') #powershell can't convert 'z' to System.Int32 so it throws an error
```

Gives the output:

```
System.Collections.Generic.SortedDictionary`2[[System.Int32, mscorlib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089],[System.String, mscorlib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089]]

Key Value
--- -----
1 a
2 b
```

```
3 c
4 d
5 c
无法将参数“key”，值为“z”，转换为“Add”的类型“System.Int32”：“无法将值“z”转换为类型“System.Int32”。错误：“输入字符串格
式不正确。””
第15行 字符1
+ $dict.Add('z', 'z') #powershell无法将'z'转换为System.Int32，因此...
+ ~~~~~~~~~~~~~~~~~~~~
+ CategoryInfo          : NotSpecified: (:) [], MethodException
+ FullyQualifiedErrorId : MethodArgumentConversionInvalidCastArgument
```

```
3 c
4 d
5 c
Cannot convert argument "key", with value: "z", for "Add" to type "System.Int32": "Cannot convert
value "z" to type "System.Int32". Error: "Input string was not in a correct format.""
At line:15 char:1
+ $dict.Add('z', 'z') #powershell can't convert 'z' to System.Int32 so  ...
+ ~~~~~~~~~~~~~~~~~~~~
+ CategoryInfo          : NotSpecified: (:) [], MethodException
+ FullyQualifiedErrorId : MethodArgumentConversionInvalidCastArgument
```

第12章：PowerShell函数

函数基本上是一个具名的代码块。当你调用函数名时，该函数内的脚本块就会运行。它是一组你指定名称的PowerShell语句。当你运行函数时，只需输入函数名。这是一种在处理重复任务时节省时间的方法。PowerShell函数由三部分组成：关键字“Function”，后跟名称，最后是包含脚本块的主体，脚本块用大括号/括号样式的括号括起。

第12.1节：基本参数

函数可以使用param块定义参数：

```
function Write-Greeting {
    param(
        [参数(必填,位置=0)]
        [字符串]$name,
        [参数 (必填, 位置=1) ]
        [整数]$age
    )
    "你好 $name, 你今年 $age 岁。"
}
```

或者使用简单的函数语法：

```
函数 Write-Greeting ($name, $age) {
    "你好 $name, 你今年 $age 岁。"
}
```

注意： 在任一类型的参数定义中都不需要进行参数类型转换。

简单函数语法（SFS）相比 param 块功能非常有限。
虽然你可以定义函数内部可用的参数，但不能指定 参数属性，
使用 参数验证，包含 [CmdletBinding()]，SFS 不支持这些（且这不是完整列表）。

函数可以使用有序参数或命名参数调用。

调用时参数的顺序默认与函数头中声明的顺序匹配（默认），或者可以使用 位置 参数属性指定（如上面高级函数示例所示）。

```
$greeting = Write-Greeting "Jim" 82
```

或者，可以使用命名参数调用此函数

```
$greeting = Write-Greeting -name "Bob" -age 82
```

第12.2节：高级函数

这是来自Powershell ISE的高级函数代码片段的副本。基本上，这是许多你可以在Powershell中使用的高级函数的模板。需要注意的关键点：

- get-help集成——函数开头包含一个注释块，设置为由get-help cmdlet读取。函数块可以放在末尾，如果需要的话。
- cmdletbinding——函数将表现得像一个cmdlet

Chapter 12: PowerShell Functions

A function is basically a named block of code. When you call the function name, the script block within that function runs. It is a list of PowerShell statements that has a name that you assign. When you run a function, you type the function name. It is a method of saving time when tackling repetitive tasks. PowerShell formats in three parts: the keyword 'Function', followed by a Name, finally, the payload containing the script block, which is enclosed by curly/parenthesis style bracket.

Section 12.1: Basic Parameters

A function can be defined with parameters using the param block:

```
function Write-Greeting {
    param(
        [Parameter(Mandatory,Position=0)]
        [String]$name,
        [Parameter(Mandatory,Position=1)]
        [Int]$age
    )
    "Hello $name, you are $age years old."
}
```

Or using the simple function syntax:

```
function Write-Greeting ($name, $age) {
    "Hello $name, you are $age years old."
}
```

Note: Casting parameters is not required in either type of parameter definition.

Simple function syntax (SFS) has very limited capabilities in comparison to the param block.
Though you can define parameters to be exposed within the function, you cannot specify Parameter Attributes, utilize Parameter Validation, include [CmdletBinding()], with SFS (and this is a non-exhaustive list).

Functions can be invoked with ordered or named parameters.

The order of the parameters on the invocation is matched to the order of the declaration in the function header (by default), or can be specified using the Position Parameter Attribute (as shown in the advanced function example, above).

```
$greeting = Write-Greeting "Jim" 82
```

Alternatively, this function can be invoked with named parameters

```
$greeting = Write-Greeting -name "Bob" -age 82
```

Section 12.2: Advanced Function

This is a copy of the advanced function snippet from the Powershell ISE. Basically this is a template for many of the things you can use with advanced functions in Powershell. Key points to note:

- get-help integration - the beginning of the function contains a comment block that is set up to be read by the get-help cmdlet. The function block may be located at the end, if desired.
- cmdletbinding - function will behave like a cmdlet

- 参数
- 参数集

```
<#
.Synopsis
简短描述
.描述
详细说明
.示例
如何使用此命令的示例
.示例
另一个如何使用此命令的示例
.输入
此命令的输入（如果有）
.输出
此命令的输出（如果有）
.备注
一般备注
.组件
此命令所属的组件
.角色
此命令所属的角色
.功能
最能描述此命令的功能
#>
function 动词-名词
{
    [CmdletBinding(DefaultParameterSetName='参数集 1',
                  SupportsShouldProcess=$true,
                  PositionalBinding=$false,
                  HelpUri = 'http://www.microsoft.com/',
                  ConfirmImpact='中')]
    [Alias()]
    [OutputType([字符串])]
    参数
    (
        # 参数1 帮助说明
        [Parameter(必需=$true,
ValueFromPipeline=$true,
                   ValueFromPipelineByPropertyName=$true,
                   ValueFromRemainingArguments=$false,
                   位置=0,
参数集名称='参数集 1')]
        [ValidateNotNull()]
        [ValidateNotNullOrEmpty()]
        [ValidateCount(0,5)]
        [ValidateSet("sun", "moon", "earth")]
        [Alias("p1")]
        $Param1,

        # Param2 帮助说明
        [参数(参数集名称='参数集 1')]
        [允许空值()]
        [允许空集合()]
        [允许空字符串()]
        [验证脚本({$true})]
        [验证范围(0,5)]
        [整数]
        $Param2,

        # Param3 帮助说明
        [参数(参数集名称='另一个参数集')]
```

- parameters
- parameter sets

```
<#
.Synopsis
   Short description
.DESCRIPTION
   Long description
.EXAMPLE
   Example of how to use this cmdlet
.EXAMPLE
   Another example of how to use this cmdlet
.INPUTS
   Inputs to this cmdlet (if any)
.OUTPUTS
   Output from this cmdlet (if any)
.NOTES
   General notes
.COMPONENT
   The component this cmdlet belongs to
.ROLE
   The role this cmdlet belongs to
.FUNCTIONALITY
   The functionality that best describes this cmdlet
#>
function Verb-Noun
{
    [CmdletBinding(DefaultParameterSetName='Parameter Set 1',
                  SupportsShouldProcess=$true,
                  PositionalBinding=$false,
                  HelpUri = 'http://www.microsoft.com/',
                  ConfirmImpact='Medium')]
    [Alias()]
    [OutputType([String])]
    Param
    (
        # Param1 help description
        [Parameter(Mandatory=$true,
                   ValueFromPipeline=$true,
                   ValueFromPipelineByPropertyName=$true,
                   ValueFromRemainingArguments=$false,
                   Position=0,
                   ParameterSetName='Parameter Set 1')]
        [ValidateNotNull()]
        [ValidateNotNullOrEmpty()]
        [ValidateCount(0,5)]
        [ValidateSet("sun", "moon", "earth")]
        [Alias("p1")]
        $Param1,

        # Param2 help description
        [Parameter(ParameterSetName='Parameter Set 1')]
        [AllowNull()]
        [AllowEmptyCollection()]
        [AllowEmptyString()]
        [ValidateScript({$true})]
        [ValidateRange(0,5)]
        [int]
        $Param2,

        # Param3 help description
        [Parameter(ParameterSetName='Another Parameter Set')]
        [验证模式("[a-z]*")]
        [验证长度(0,15)]
        [字符串]
        $Param3
    )

    开始
    {
    }
    过程
    {
        如果 ($pscmdlet.ShouldProcess("目标", "操作"))
        {
        }
    }
    结束
    {
    }
}
```

第12.3节：必需参数

函数的参数可以标记为必需

```
函数 Get-Greeting{
    参数
    (
        [参数(必需=$true)]$name
    )
    "你好，$name"
}
```

如果调用函数时未提供值，命令行将提示输入该值：

```
$greeting = Get-Greeting

cmdlet Get-Greeting 在命令管道位置 1
请为以下参数提供值：
姓名：
```

第12.4节：参数验证

在PowerShell中，有多种方法可以验证参数输入。

与其在函数或脚本中编写代码来验证参数值，不如使用这些ParameterAttributes，如果传入无效值，它们会抛出异常。

ValidateSet

有时我们需要限制参数可以接受的可能值。比如，我们希望脚本或函数中的$Color参数只允许红色、绿色和蓝色。

我们可以使用ValidateSet参数属性来限制它。它还有一个额外的好处，就是在某些环境中设置该参数时支持制表符补全。

```
param(
    [ValidateSet('red','green','blue',IgnoreCase)]
    [string]$Color
```

```
        [ValidatePattern("[a-z]*")]
        [ValidateLength(0,15)]
        [String]
        $Param3
    )

    Begin
    {
    }
    Process
    {
        if ($pscmdlet.ShouldProcess("Target", "Operation"))
        {
        }
    }
    End
    {
    }
}
```

Section 12.3: Mandatory Parameters

Parameters to a function can be marked as mandatory

```
function Get-Greeting{
    param
    (
        [Parameter(Mandatory=$true)]$name
    )
    "Hello World $name"
}
```

If the function is invoked without a value, the command line will prompt for the value:

```
$greeting = Get-Greeting

cmdlet Get-Greeting at command pipeline position 1
Supply values for the following parameters:
name:
```

Section 12.4: Parameter Validation

There are a variety of ways to validate parameter entry, in PowerShell.

Instead of writing code within functions or scripts to validate parameter values, these ParameterAttributes will throw if invalid values are passed.

ValidateSet

Sometimes we need to restrict the possible values that a parameter can accept. Say we want to allow only red, green and blue for the `$Color` parameter in a script or function.

We can use the `ValidateSet` parameter attribute to restrict this. It has the additional benefit of allowing tab completion when setting this argument (in some environments).

```
param(
    [ValidateSet('red','green','blue',IgnoreCase)]
    [string]$Color
)
```

你也可以指定IgnoreCase来禁用大小写敏感。

ValidateRange

此参数验证方法接受一个最小值和最大值的Int32类型参数，要求参数必须在该范围内。

```
param(
    [ValidateRange(0,120)]
    [Int]$Age
)
```

ValidatePattern

此参数验证方法接受符合指定正则表达式模式的参数。

```
param(
    [ValidatePattern("\w{4-6}\d{2}")]
    [string]$UserName
)
```

ValidateLength

此参数验证方法用于测试传入字符串的长度。

```
param(
    [ValidateLength(0,15)]
    [String]$PhoneNumber
)
```

ValidateCount

这种参数验证方法测试传入的参数数量，例如，一个字符串数组。

```
param(
    [ValidateCount(1,5)]
    [String[]]$ComputerName
)
```

ValidateScript

最后，ValidateScript 方法非常灵活，它接受一个脚本块并使用 $_ 来表示传入的参数进行评估。如果结果为 $true（包括任何输出），则通过该参数。

这可以用来测试文件是否存在：

```
param(
    [ValidateScript({Test-Path $_})]
    [IO.FileInfo]$Path
)
```

检查 AD 中用户是否存在：

```
)
```

You can also specify `IgnoreCase` to disable case sensitivity.

ValidateRange

This method of parameter validation takes a min and max Int32 value, and requires the parameter to be within that range.

```
param(
    [ValidateRange(0,120)]
    [Int]$Age
)
```

ValidatePattern

This method of parameter validation accepts parameters that match the regex pattern specified.

```
param(
    [ValidatePattern("\w{4-6}\d{2}")]
    [string]$UserName
)
```

ValidateLength

This method of parameter validation tests the length of the passed string.

```
param(
    [ValidateLength(0,15)]
    [String]$PhoneNumber
)
```

ValidateCount

This method of parameter validation tests the amount of arguments passed in, for example, an array of strings.

```
param(
    [ValidateCount(1,5)]
    [String[]]$ComputerName
)
```

ValidateScript

Finally, the ValidateScript method is extraordinarily flexible, taking a scriptblock and evaluating it using $_ to represent the passed argument. It then passes the argument if the result is $true (including any output as valid).

This can be used to test that a file exists:

```
param(
    [ValidateScript({Test-Path $_})]
    [IO.FileInfo]$Path
)
```

To check that a user exists in AD:

```
param(
    [ValidateScript({Get-ADUser $_})]
    [String]$UserName
)
```

以及几乎任何你能写的内容（因为它不限于单行代码）：

```
param(
    [ValidateScript({
        $AnHourAgo = (Get-Date).AddHours(-1)
        if ($_ -lt $AnHourAgo.AddMinutes(5) -and $_ -gt $AnHourAgo.AddMinutes(-5)) {
            $true
        } else {
            throw "That's not within five minutes. Try again."
        }
    })]
    [String]$TimeAboutAnHourAgo
)
```

第12.5节：无参数的简单函数

这是一个返回字符串的函数示例。在示例中，函数被调用于一个赋值语句中。此时的值是函数的返回值。

```
function Get-Greeting{
    "Hello World"
}

# 调用函数
$greeting = Get-Greeting

# 显示输出
$greeting
Get-Greeting
```

function 声明以下代码为一个函数。

Get-Greeting 是函数的名称。每当脚本中需要使用该函数时，可以通过调用其名称来调用该函数。

{ ... } 是由函数执行的脚本块。

如果在 ISE 中执行上述代码，结果将类似于：

```
Hello World
Hello World
```

```
param(
    [ValidateScript({Get-ADUser $_})]
    [String]$UserName
)
```

And pretty much anything else you can write (as it's not restricted to oneliners):

```
param(
    [ValidateScript({
        $AnHourAgo = (Get-Date).AddHours(-1)
        if ($_ -lt $AnHourAgo.AddMinutes(5) -and $_ -gt $AnHourAgo.AddMinutes(-5)) {
            $true
        } else {
            throw "That's not within five minutes. Try again."
        }
    })]
    [String]$TimeAboutAnHourAgo
)
```

Section 12.5: Simple Function with No Parameters

This is an example of a function which returns a string. In the example, the function is called in a statement assigning a value to a variable. The value in this case is the return value of the function.

```
function Get-Greeting{
    "Hello World"
}

# Invoking the function
$greeting = Get-Greeting

# demonstrate output
$greeting
Get-Greeting
```

`function` declares the following code to be a function.

`Get-Greeting` is the name of the function. Any time that function needs to be used in the script, the function can be called by means of invoking it by name.

`{ ... }` is the script block that is executed by the function.

If the above code is executed in the ISE, the results would be something like:

```
Hello World
Hello World
```

第13章：PowerShell 类

类是用于创建对象的可扩展程序代码模板，提供状态（成员变量）的初始值和行为（成员函数或方法）的实现。类是对象的蓝图。它用作定义对象结构的模型。对象包含我们通过属性访问的数据，并且可以使用方法对其进行操作。PowerShell 5.0 增加了创建自定义类的功能。

第13.1节：列出类的可用构造函数

版本 ≥ 5.0

在 PowerShell 5.0 及以上版本中，可以通过调用静态new方法且不带括号来列出可用的构造函数。

```
PS> [DateTime]::new

重载定义
-------------------
datetime new(long ticks)
datetime new(long ticks, System.DateTimeKind kind)
datetime new(int year, int month, int day)
datetime new(int year, int month, int day, System.Globalization.Calendar calendar)
datetime new(int year, int month, int day, int hour, int minute, int second)
datetime new(int year, int month, int day, int hour, int minute, int second, System.DateTimeKind
kind)
datetime new(int year, int month, int day, int hour, int minute, int second,
System.Globalization.Calendar calendar)
datetime new(int year, int month, int day, int hour, int minute, int second, int millisecond)
datetime new(int year, int month, int day, int hour, int minute, int second, int millisecond,
System.DateTimeKind kind)
datetime new(int year, int month, int day, int hour, int minute, int second, int millisecond,
System.Globalization.Calendar calendar)
datetime new(int year, int month, int day, int hour, int minute, int second, int millisecond,
System.Globalization.Calendar calendar, System.DateTimeKind kind)
```

这就是你可以用来列出任何方法重载定义的相同技术

```
> 'abc'.CompareTo

重载定义
-------------------
int CompareTo(System.Object value)
int CompareTo(string strB)
int IComparable.CompareTo(System.Object obj)
int IComparable[string].CompareTo(string other)
```

对于早期版本，您可以创建自己的函数来列出可用的构造函数：

```
function Get-Constructor {
    [CmdletBinding()]
    param(
        [Parameter(ValueFromPipeline=$true)]
        [type]$type
    )

    Process {
        $type.GetConstructors() |
        Format-Table -Wrap @{
            n="$($type.Name) 构造函数"
```

Chapter 13: PowerShell Classes

A class is an extensible program-code-template for creating objects, providing initial values for state (member variables) and implementations of behavior (member functions or methods).A class is a blueprint for an object. It is used as a model to define the structure of objects. An object contains data that we access through properties and that we can work on using methods. PowerShell 5.0 added the ability to create your own classes.

Section 13.1: Listing available constructors for a class

Version ≥ 5.0

In PowerShell 5.0+ you can list available constructors by calling the static new-method without parentheses.

```
PS> [DateTime]::new

OverloadDefinitions
-------------------
datetime new(long ticks)
datetime new(long ticks, System.DateTimeKind kind)
datetime new(int year, int month, int day)
datetime new(int year, int month, int day, System.Globalization.Calendar calendar)
datetime new(int year, int month, int day, int hour, int minute, int second)
datetime new(int year, int month, int day, int hour, int minute, int second, System.DateTimeKind
kind)
datetime new(int year, int month, int day, int hour, int minute, int second,
System.Globalization.Calendar calendar)
datetime new(int year, int month, int day, int hour, int minute, int second, int millisecond)
datetime new(int year, int month, int day, int hour, int minute, int second, int millisecond,
System.DateTimeKind kind)
datetime new(int year, int month, int day, int hour, int minute, int second, int millisecond,
System.Globalization.Calendar calendar)
datetime new(int year, int month, int day, int hour, int minute, int second, int millisecond,
System.Globalization.Calendar calendar, System.DateTimeKind kind)
```

This is the same technique that you can use to list overload definitions for any method

```
> 'abc'.CompareTo

OverloadDefinitions
-------------------
int CompareTo(System.Object value)
int CompareTo(string strB)
int IComparable.CompareTo(System.Object obj)
int IComparable[string].CompareTo(string other)
```

For earlier versions you can create your own function to list available constructors:

```
function Get-Constructor {
    [CmdletBinding()]
    param(
        [Parameter(ValueFromPipeline=$true)]
        [type]$type
    )

    Process {
        $type.GetConstructors() |
        Format-Table -Wrap @{
            n="$($type.Name) Constructors"
e={ ($_.GetParameters() | % { $_.ToString() }) -Join ", " }
        }
    }
}
```

用法：

```
Get-Constructor System.DateTime
#或者 [datetime] | Get-Constructor

DateTime 构造函数
---------------------
Int64 计时刻
Int64 计时刻, System.DateTimeKind 类型
Int32 年, Int32 月, Int32 日
Int32 年, Int32 月, Int32 日, System.Globalization.Calendar 日历
Int32 年, Int32 月, Int32 日, Int32 时, Int32 分, Int32 秒
Int32 年, Int32 月, Int32 日, Int32 时, Int32 分, Int32 秒, System.DateTimeKind
类型
Int32 年, Int32 月, Int32 日, Int32 时, Int32 分, Int32 秒,
System.Globalization.Calendar 日历
Int32 年, Int32 月, Int32 日, Int32 时, Int32 分, Int32 秒, Int32 毫秒
Int32 年, Int32 月, Int32 日, Int32 时, Int32 分, Int32 秒, Int32 毫秒,
System.DateTimeKind 类型
Int32 年, Int32 月, Int32 日, Int32 时, Int32 分, Int32 秒, Int32 毫秒,
System.Globalization.Cal
endar 日历
Int32 年, Int32 月, Int32 日, Int32 时, Int32 分, Int32 秒, Int32 毫秒,
System.Globalization.Cal
endar 日历, System.DateTimeKind 类型
```

第13.2节：方法和属性

```
class Person {
    [string] $FirstName
    [string] $LastName
    [string] Greeting() {
        return "Greetings, {0} {1}!" -f $this.FirstName, $this.LastName
    }
}

$x = [Person]::new()
$x.FirstName = "Jane"
$x.LastName = "Doe"
$greeting = $x.Greeting() # "Greetings, Jane Doe!"
```

第13.3节：构造函数重载

```
class Person {
    [string] $Name
    [int] $Age

Person([string] $Name) {
        $this.Name = $Name
    }

Person([string] $Name, [int]$Age) {
        $this.Name = $Name
        $this.Age = $Age
```

```
            e={ ($_.GetParameters() | % { $_.ToString() }) -Join ", " }
        }
    }
}
```

Usage:

```
Get-Constructor System.DateTime
#Or [datetime] | Get-Constructor

DateTime Constructors
---------------------
Int64 ticks
Int64 ticks, System.DateTimeKind kind
Int32 year, Int32 month, Int32 day
Int32 year, Int32 month, Int32 day, System.Globalization.Calendar calendar
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second, System.DateTimeKind
kind
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second,
System.Globalization.Calendar calendar
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second, Int32 millisecond
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second, Int32 millisecond,
System.DateTimeKind kind
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second, Int32 millisecond,
System.Globalization.Cal
endar calendar
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second, Int32 millisecond,
System.Globalization.Cal
endar calendar, System.DateTimeKind kind
```

Section 13.2: Methods and properties

```
class Person {
    [string] $FirstName
    [string] $LastName
    [string] Greeting() {
        return "Greetings, {0} {1}!" -f $this.FirstName, $this.LastName
    }
}

$x = [Person]::new()
$x.FirstName = "Jane"
$x.LastName = "Doe"
$greeting = $x.Greeting() # "Greetings, Jane Doe!"
```

Section 13.3: Constructor overloading

```
class Person {
    [string] $Name
    [int] $Age

    Person([string] $Name) {
        $this.Name = $Name
    }

    Person([string] $Name, [int]$Age) {
        $this.Name = $Name
        $this.Age = $Age
    }
}
```

第13.4节：获取实例的所有成员

```
PS > Get-Member -InputObject $anObjectInstance
```

这将返回该类型实例的所有成员。以下是字符串实例的部分示例输出

```
   TypeName: System.String

Name             MemberType            Definition
----             ----------            ----------
Clone              Method                 System.Object Clone(), System.Object ICloneable.Clone()
CompareTo          Method                 int CompareTo(System.Object value), int CompareTo(string
strB), i...
Contains         Method                bool Contains(string value)
CopyTo             Method                 void CopyTo(int sourceIndex, char[] destination, int
destinationI...
EndsWith         方法                bool EndsWith(string value), bool EndsWith(string value,
System.S...
Equals           方法               bool Equals(System.Object obj), bool Equals(string value),
bool E...
GetEnumerator     方法                 System.CharEnumerator GetEnumerator(),
System.Collections.Generic...
GetHashCode       方法                 int GetHashCode()
GetType          方法                type GetType()
...
```

第13.5节：基本类模板

```
# 定义一个类
class 类型名
{
   # 带验证设置的属性
   [ValidateSet("val1", "Val2")]
   [string] $P1

   # 静态属性
   static [hashtable] $P2

   # 隐藏属性不会作为Get-Member的结果显示
   hidden [int] $P3

   # 构造函数
  类型名 ([string] $s)
   {
       $this.P1 = $s
   }

   # 静态方法
   static [void] MemberMethod1([hashtable] $h)
   {
       [TypeName]::P2 = $h
   }

   # 实例方法
   [int] MemberMethod2([int] $i)
   {
```

```
    }
}
```

Section 13.4: Get All Members of an Instance

```
PS > Get-Member -InputObject $anObjectInstance
```

This will return all members of the type instance. Here is a part of a sample output for String instance

```
   TypeName: System.String

Name             MemberType            Definition
----             ----------            ----------
Clone            Method                System.Object Clone(), System.Object ICloneable.Clone()
CompareTo        Method                int CompareTo(System.Object value), int CompareTo(string
strB), i...
Contains         Method                bool Contains(string value)
CopyTo           Method                void CopyTo(int sourceIndex, char[] destination, int
destinationI...
EndsWith         Method                bool EndsWith(string value), bool EndsWith(string value,
System.S...
Equals           Method                bool Equals(System.Object obj), bool Equals(string value),
bool E...
GetEnumerator    Method                System.CharEnumerator GetEnumerator(),
System.Collections.Generic...
GetHashCode      Method                int GetHashCode()
GetType          Method                type GetType()
...
```

Section 13.5: Basic Class Template

```
# Define a class
class TypeName
{
   # Property with validate set
   [ValidateSet("val1", "Val2")]
   [string] $P1

   # Static property
   static [hashtable] $P2

   # Hidden property does not show as result of Get-Member
   hidden [int] $P3

   # Constructor
   TypeName ([string] $s)
   {
       $this.P1 = $s
   }

   # Static method
   static [void] MemberMethod1([hashtable] $h)
   {
       [TypeName]::P2 = $h
   }

   # Instance method
   [int] MemberMethod2([int] $i)
   {
        $this.P3 = $i
        return $this.P3
    }
}
```

第13.6节：从父类继承到子类

```
class ParentClass
{
    [string] $Message = "它属于父类"

    [string] GetMessage()
    {
        return ("消息: {0}" -f $this.Message)
    }
}

# Bar 继承自 Foo 并继承其成员
class ChildClass : ParentClass
{

}
$Inherit = [ChildClass]::new()
```

所以，$Inherit.Message 会给你

> “它属于父类”

```
        $this.P3 = $i
        return $this.P3
    }
}
```

Section 13.6: Inheritance from Parent Class to Child Class

```
class ParentClass
{
    [string] $Message = "It's under the Parent Class"

    [string] GetMessage()
    {
        return ("Message: {0}" -f $this.Message)
    }
}

# Bar extends Foo and inherits its members
class ChildClass : ParentClass
{

}
$Inherit = [ChildClass]::new()
```

SO, **$Inherit.Message** will give you the

> "It's under the Parent Class"

第14章：PowerShell模块

从PowerShell 2.0版本开始，开发者可以创建PowerShell模块。PowerShell模块封装了一组常用功能。例如，有供应商特定的PowerShell模块用于管理各种云服务。也有通用的PowerShell模块，用于与社交媒体服务交互，并执行常见的编程任务，如Base64编码、使用命名管道等。

模块可以公开命令别名、函数、变量、类等。

第14.1节：创建模块清单

```
@{
RootModule = 'MyCoolModule.psm1'
  ModuleVersion = '1.0'
CompatiblePSEditions = @('Core')
  GUID = '6b42c995-67da-4139-be79-597a328056cc'
  Author = 'Bob Schmob'
CompanyName = '我的公司'
Copyright = '(c) 2017 管理员。保留所有权利。'
  Description = '它能做很酷的事情。'
FunctionsToExport = @()
  CmdletsToExport = @()
  VariablesToExport = @()
  AliasesToExport = @()
  DscResourcesToExport = @()
}
```

每个优秀的 PowerShell 模块都有一个模块清单。模块清单仅包含有关 PowerShell 模块的元数据，并不定义模块的实际内容。

清单文件是一个 PowerShell 脚本文件，扩展名为 .psd1，包含一个哈希表。清单中的哈希表必须包含特定的键，以便 PowerShell 正确地将其解释为 PowerShell 模块文件。

上面的示例列出了构成模块清单的核心哈希表键，但还有许多其他键。 New-ModuleManifest 命令可以帮助你创建一个新的模块清单骨架。

第14.2节：简单模块示例

```
function Add {
  [CmdletBinding()]
  param (
    [int] $x
  , [int] $y
  )

  return $x + $y
}

Export-ModuleMember -函数 Add
```

这是一个 PowerShell 脚本模块文件可能的简单示例。该文件名为 MyCoolModule.psm1，并且在模块清单（.psd1）文件中被引用。你会注意到，Export-ModuleMember 命令使我们能够指定模块中想要“导出”或向模块用户公开的函数。有些函数仅供内部使用，不应公开，因此这些函数会被省略，不包含在对Export-ModuleMember的调用中。

Chapter 14: PowerShell Modules

Starting with PowerShell version 2.0, developers can create PowerShell modules. PowerShell modules encapsulate a set of common functionality. For example, there are vendor-specific PowerShell modules that manage various cloud services. There are also generic PowerShell modules that interact with social media services, and perform common programming tasks, such as Base64 encoding, working with Named Pipes, and more.

Modules can expose command aliases, functions, variables, classes, and more.

Section 14.1: Create a Module Manifest

```
@{
  RootModule = 'MyCoolModule.psm1'
  ModuleVersion = '1.0'
  CompatiblePSEditions = @('Core')
  GUID = '6b42c995-67da-4139-be79-597a328056cc'
  Author = 'Bob Schmob'
  CompanyName = 'My Company'
  Copyright = '(c) 2017 Administrator. All rights reserved.'
  Description = 'It does cool stuff.'
  FunctionsToExport = @()
  CmdletsToExport = @()
  VariablesToExport = @()
  AliasesToExport = @()
  DscResourcesToExport = @()
}
```

Every good PowerShell module has a module manifest. The module manifest simply contains metadata about a PowerShell module, and doesn't define the actual contents of the module.

The manifest file is a PowerShell script file, with a .psd1 file extension, that contains a HashTable. The HashTable in the manifest must contain specific keys, in order for PowerShell to correctly interpret it as a PowerShell module file.

The example above provides a list of the core HashTable keys that make up a module manifest, but there are many others. The New-ModuleManifest command helps you create a new module manifest skeleton.

Section 14.2: Simple Module Example

```
function Add {
  [CmdletBinding()]
  param (
    [int] $x
  , [int] $y
  )

  return $x + $y
}

Export-ModuleMember -Function Add
```

This is a simple example of what a PowerShell script module file might look like. This file would be called MyCoolModule.psm1, and is referenced from the module manifest (.psd1) file. You'll notice that the Export-ModuleMember command enables us to specify which functions in the module we want to "export," or expose, to the user of the module. Some functions will be internal-only, and shouldn't be exposed, so those would be omitted from the call to Export-ModuleMember.

第14.3节：从模块导出变量

```
$FirstName = 'Bob'
Export-ModuleMember -Variable FirstName
```

要从模块导出变量，可以使用Export-ModuleMember命令，并带上-Variable参数。
但请记住，如果变量没有在模块清单（.psd1）文件中被显式导出，那么该变量对模块使用者将不可见。可以把模块清单看作"守门员"。如果函数或变量未被允许出现在模块清单中，它们对模块使用者来说是不可见的。

注意：导出变量类似于将类中的字段设为公共。这通常不建议这样做。更好的做法是公开一个获取该字段的函数和一个设置该字段的函数。

第14.4节：构建 PowerShell 模块的结构

与其将所有函数都定义在单个.psm1 PowerShell 脚本模块文件中，不如将函数拆分成单独的文件。然后你可以从脚本模块文件中点源（dot-source）这些文件，本质上将它们视为.psm1文件的一部分。

考虑以下模块目录结构：

```
\MyCoolModule
  \Functions
    Function1.ps1
    Function2.ps1
    Function3.ps1
MyCoolModule.psd1
MyCoolModule.psm1
```

在你的MyCoolModule.psm1文件中，你可以插入以下代码：

```
Get-ChildItem -Path $PSScriptRoot\Functions |
  ForEach-Object -Process { . $PSItem.FullName }
```

这将点源（dot-source）各个函数文件到.psm1模块文件中。

第14.5节：模块的位置

PowerShell会在$Env:PSModulepath中列出的目录中查找模块。

名为*foo*的模块，在名为*foo*的文件夹中，可以通过`Import-Module foo`找到。

在该文件夹中，PowerShell会查找模块清单（foo.psd1）、模块文件（foo.psm1）和DLL（foo.dll）。

第14.6节：模块成员的可见性

默认情况下，只有模块中定义的函数对模块外部可见。换句话说，如果你在模块中定义了变量和别名，它们除了在模块代码中外，其他地方都不可用。

要覆盖此行为，您可以使用`Export-ModuleMember`命令。它具有名为`-Function`、`-Variable`和`-Alias`的参数，允许您准确指定导出的成员。

需要注意的是，如果使用Export-ModuleMember，只有你指定的项目才会可见。

Section 14.3: Exporting a Variable from a Module

```
$FirstName = 'Bob'
Export-ModuleMember -Variable FirstName
```

To export a variable from a module, you use the `Export-ModuleMember` command, with the `-Variable` parameter. Remember, however, that if the variable is also not explicitly exported in the module manifest (.psd1) file, then the variable will not be visible to the module consumer. Think of the module manifest like a "gatekeeper." If a function or variable isn't allowed in the module manifest, it won't be visible to the module consumer.

Note: Exporting a variable is similar to making a field in a class public. It is not advisable. It would be better to expose a function to get the field and a function to set the field.

Section 14.4: Structuring PowerShell Modules

Rather than defining all of your functions in a single `.psm1` PowerShell script module file, you might want to break apart your function into individual files. You can then dot-source these files from your script module file, which in essence, treats them as if they were part of the `.psm1` file itself.

Consider this module directory structure:

```
\MyCoolModule
  \Functions
    Function1.ps1
    Function2.ps1
    Function3.ps1
MyCoolModule.psd1
MyCoolModule.psm1
```

Inside your `MyCoolModule.psm1` file, you could insert the following code:

```
Get-ChildItem -Path $PSScriptRoot\Functions |
  ForEach-Object -Process { . $PSItem.FullName }
```

This would dot-source the individual function files into the `.psm1` module file.

Section 14.5: Location of Modules

PowerShell looks for modules in the directories listed in the $Env:PSModulepath.

A module called *foo*, in a folder called *foo* will be found with `Import-Module foo`

In that folder, PowerShell will look for a module manifest (foo.psd1), a module file (foo.psm1), a DLL (foo.dll).

Section 14.6: Module Member Visibility

By default, only functions defined in a module are visible outside of the module. In other words, if you define variables and aliases in a module, they won't be available except in the module's code.

To override this behavior, you can use the `Export-ModuleMember` cmdlet. It has parameters called `-Function`, `-Variable`, and `-Alias` which allow you to specify exactly which members are exported.

It is important to note that if you use `Export-ModuleMember`, **only** the items you specify will be visible.

第15章：PowerShell 配置文件

第15.1节：创建基本配置文件

PowerShell 配置文件用于自动加载用户定义的变量和函数。

PowerShell 配置文件不会自动为用户创建。

要创建 PowerShell 配置文件，使用命令C:>New-Item -ItemType File $profile。

如果你在 ISE 中，可以使用内置编辑器C:>psEdit $profile

为当前主机快速开始个人配置文件的简单方法是将一些文本保存到存储在
$profile变量中的路径

```
"#当前主机，当前用户" > $profile
```

可以使用 PowerShell ISE、记事本、Visual Studio Code 或任何其他

编辑器对配置文件进行进一步修改。

$profile变量默认返回当前主机的当前用户配置文件，但你可以通过其属性访问机器策略（所有用户）和/或所有主机（控制台、ISE、第三方）的配置文件路径。

```
PS> $PROFILE | Format-List -Force

AllUsersAllHosts       : C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1
AllUsersCurrentHost    :
C:\Windows\System32\WindowsPowerShell\v1.0\Microsoft.PowerShell_profile.ps1
CurrentUserAllHosts    : C:\Users\user\Documents\WindowsPowerShell\profile.ps1
CurrentUserCurrentHost : C:\Users\user\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1
Length                 : 75

PS> $PROFILE.AllUsersAllHosts
C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1
```

Chapter 15: PowerShell profiles

Section 15.1: Create an basic profile

A PowerShell profile is used to load user defined variables and functions automatically.

PowerShell profiles are not automatically created for users.

To create a PowerShell profile C:>New-Item -ItemType File $profile.

If you are in ISE you can use the built in editor C:>psEdit $profile

An easy way to get started with your personal profile for the current host is to save some text to path stored in the $profile-variable

```
"#Current host, current user" > $profile
```

Further modification to the profile can be done using PowerShell ISE, notepad, Visual Studio Code or any other editor.

The $profile-variable returns the current user profile for the current host by default, but you can access the path to the machine-policy (all users) and/or the profile for all hosts (console, ISE, 3rd party) by using its properties.

```
PS> $PROFILE | Format-List -Force

AllUsersAllHosts       : C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1
AllUsersCurrentHost    :
C:\Windows\System32\WindowsPowerShell\v1.0\Microsoft.PowerShell_profile.ps1
CurrentUserAllHosts    : C:\Users\user\Documents\WindowsPowerShell\profile.ps1
CurrentUserCurrentHost : C:\Users\user\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1
Length                 : 75

PS> $PROFILE.AllUsersAllHosts
C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1
```

第16章：计算属性

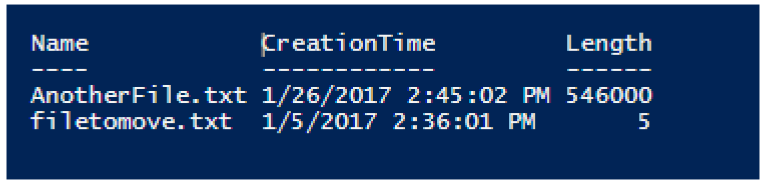
PowerShell中的计算属性是自定义派生（计算）属性。它允许用户以自己想要的方式格式化某个属性。计算（表达式）几乎可以是任何内容。

第16.1节：以KB显示文件大小 - 计算属性

让我们考虑下面的代码片段，

```
Get-ChildItem -Path C:\MyFolder | Select-Object Name, CreationTime, Length
```

它只是输出带有所选属性的文件夹内容。类似于，

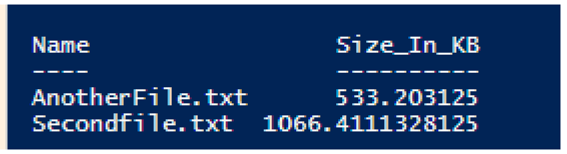


```
Name             CreationTime          Length
----             ------------          ------
AnotherFile.txt 1/26/2017 2:45:02 PM 546000
filetomove.txt  1/5/2017 2:36:01 PM        5
```

如果我想以KB为单位显示文件大小怎么办？这就是计算属性派上用场的地方。

```
Get-ChildItem C:\MyFolder | Select-Object Name, @{Name="Size_In_KB";Expression={$_.Length / 1Kb}}
```

结果如下，



```
Name                  Size_In_KB
----                  ----------
AnotherFile.txt       533.203125
Secondfile.txt    1066.4111328125
```

Expression 是保存计算属性计算内容的地方。没错，它可以是任何内容！

Chapter 16: Calculated Properties

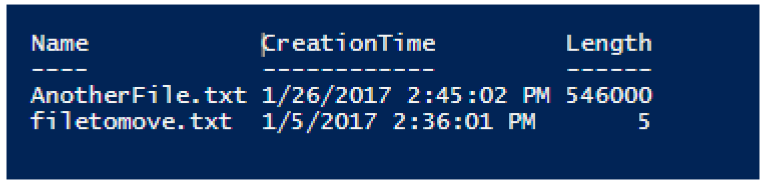
Calculated Properties in PowerShell are custom derived (Calculated) properties. It lets the user to format a certain property in a way he want it to be. The calculation(expression) can be a quite possibly anything.

Section 16.1: Display file size in KB - Calculated Properties

Let's consider the below snippet,

```
Get-ChildItem -Path C:\MyFolder | Select-Object Name, CreationTime, Length
```

It simply output the folder content with the selected properties. Something like,

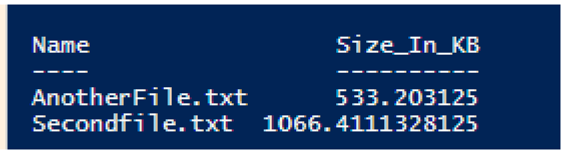


```
Name             CreationTime          Length
----             ------------          ------
AnotherFile.txt 1/26/2017 2:45:02 PM 546000
filetomove.txt  1/5/2017 2:36:01 PM        5
```

What if I want to display the file size in KB ? This is where calcualted properties comes handy.

```
Get-ChildItem C:\MyFolder | Select-Object Name, @{Name="Size_In_KB";Expression={$_.Length / 1Kb}}
```

Which produces,



```
Name                  Size_In_KB
----                  ----------
AnotherFile.txt       533.203125
Secondfile.txt    1066.4111328125
```

The `Expression` is what holds the calculation for calculated property. And yes, it can be anything!

第17章：使用现有的静态类

这些类是方法和属性的引用库，它们的状态不会改变，简而言之，就是不可变的。你不需要创建它们，只需使用它们。这样的类和方法被称为静态类因为它们不会被创建、销毁或更改。你可以通过用方括号括住类名来引用静态类。

第17.1节：添加类型

按程序集名称添加库

```
Add-Type -AssemblyName "System.Math"
```

或者通过文件路径：

```
Add-Type -Path "D:\Libs\CustomMath.dll"
```

使用已添加的类型：

```
[CustomMath.NameSpace]::Method(param1, $variableParam, [int]castMeAsIntParam)
```

第17.2节：使用.Net Math类

你可以使用.Net Math类进行计算（[System.Math]）

如果你想知道有哪些可用的方法，可以使用：

```
[System.Math] | Get-Member -Static -MemberType Methods
```

以下是一些使用Math类的示例：

```
PS C:\> [System.Math]::Floor(9.42)
9
PS C:\> [System.Math]::Ceiling(9.42)
10
PS C:\> [System.Math]::Pow(4,3)
64
PS C:\> [System.Math]::Sqrt(49)
7
```

第17.3节：即时创建新的GUID

通过使用[class]::Method(args)，可在PowerShell中即时使用现有的.NET类：

```
PS C:\> [guid]::NewGuid()

Guid
----
8874a185-64be-43ed-a64c-d2fe4b6e31bc
```

同样，在PowerShell 5及以上版本中，可以使用New-Guid命令：

```
PS C:\> New-Guid
```

Chapter 17: Using existing static classes

These classes are reference libraries of methods and properties that do not change state, in one word, immutable. You don't need to create them, you simply use them. Classes and methods such as these are called static classes because they are not created, destroyed, or changed. You can refer to a static class by surrounding the class name with square brackets.

Section 17.1: Adding types

By Assembly Name, add library

```
Add-Type -AssemblyName "System.Math"
```

or by file path:

```
Add-Type -Path "D:\Libs\CustomMath.dll"
```

To Use added type:

```
[CustomMath.NameSpace]::Method(param1, $variableParam, [int]castMeAsIntParam)
```

Section 17.2: Using the .Net Math Class

You can use the .Net Math class to do calculations ([System.Math])

If you want to know which methods are available you can use:

```
[System.Math] | Get-Member -Static -MemberType Methods
```

Here are some examples how to use the Math class:

```
PS C:\> [System.Math]::Floor(9.42)
9
PS C:\> [System.Math]::Ceiling(9.42)
10
PS C:\> [System.Math]::Pow(4,3)
64
PS C:\> [System.Math]::Sqrt(49)
7
```

Section 17.3: Creating new GUID instantly

Use existing .NET classes instantly with PowerShell by using [class]::Method(args):

```
PS C:\> [guid]::NewGuid()

Guid
----
8874a185-64be-43ed-a64c-d2fe4b6e31bc
```

Similarly, in PowerShell 5+ you may use the `New-Guid` cmdlet:

```
PS C:\> New-Guid
```

```
Guid
----
8874a185-64be-43ed-a64c-d2fe4b6e31bc
```

要仅获取 GUID 作为[字符串]，请引用.Guid属性：

```
[guid]::NewGuid().Guid
```

```
Guid
----
8874a185-64be-43ed-a64c-d2fe4b6e31bc
```

To get the GUID as a [String] only, referenced the .Guid property:

```
[guid]::NewGuid().Guid
```

第18章：内置变量

PowerShell 提供了多种有用的“自动”（内置）变量。某些自动变量仅在特殊情况下赋值，而其他变量则在全局范围内可用。

第18.1节：$PSScriptRoot

```
Get-ChildItem -Path $PSScriptRoot
```

此示例从脚本文件所在的文件夹中检索子项（目录和文件）列表。

如果在 PowerShell 代码文件外部使用，$PSScriptRoot自动变量为$null。如果在 PowerShell 脚本内部使用，它会自动定义包含脚本文件的目录的完全限定文件系统路径。

在 Windows PowerShell 2.0 中，此变量仅在脚本模块（.psm1）中有效。从 Windows PowerShell 3.0 开始，它在所有脚本中均有效。

第18.2节：$Args

```
$Args
```

包含传递给函数、脚本或脚本块的未声明参数和/或参数值的数组。当你创建函数时，可以使用param关键字声明参数，或者在函数名后用括号添加逗号分隔的参数列表。

在事件操作中，$Args变量包含表示正在处理的事件参数的对象。该变量仅在事件注册命令的Action块内填充。该变量的值也可以在Get-Event返回的PSEventArgs对象的SourceArgs属性中找到。

(System.Management.Automation.PSEventArgs) 是Get-Event返回的对象类型。

第18.3节：$PSItem

```
Get-Process | ForEach-Object -Process {
  $PSItem.Name
}
```

与$_相同。包含管道中的当前对象。你可以在对管道中的每个对象或选定对象执行操作的命令中使用此变量。

第18.4节：$?

```
Get-Process -Name doesnotexist
Write-Host -Object "上一次操作是否成功？$?"
```

包含上一次操作的执行状态。如果上一次操作成功，则为 TRUE；如果失败，则为 FALSE。

第18.5节：$error

```
Get-Process -Name doesnotexist
```

Chapter 18: Built-in variables

PowerShell offers a variety of useful "automatic" (built-in) variables. Certain automatic variables are only populated in special circumstances, while others are available globally.

Section 18.1: $PSScriptRoot

```
Get-ChildItem -Path $PSScriptRoot
```

This example retrieves the list of child items (directories and files) from the folder where the script file resides.

The $PSScriptRoot automatic variable is $null if used from outside a PowerShell code file. If used *inside* a PowerShell script, it automatically defined the fully-qualified filesystem path to the directory that contains the script file.

In Windows PowerShell 2.0, this variable is valid only in script modules (.psm1). Beginning in Windows PowerShell 3.0, it is valid in all scripts.

Section 18.2: $Args

```
$Args
```

Contains an array of the undeclared parameters and/or parameter values that are passed to a function, script, or script block. When you create a function, you can declare the parameters by using the param keyword or by adding a comma-separated list of parameters in parentheses after the function name.

In an event action, the $Args variable contains objects that represent the event arguments of the event that is being processed. This variable is populated only within the Action block of an event registration command. The value of this variable can also be found in the SourceArgs property of the PSEventArgs object (System.Management.Automation.PSEventArgs) that Get-Event returns.

Section 18.3: $PSItem

```
Get-Process | ForEach-Object -Process {
  $PSItem.Name
}
```

Same as $_. Contains the current object in the pipeline object. You can use this variable in commands that perform an action on every object or on selected objects in a pipeline.

Section 18.4: $?

```
Get-Process -Name doesnotexist
Write-Host -Object "Was the last operation successful? $?"
```

Contains the execution status of the last operation. It contains TRUE if the last operation succeeded and FALSE if it failed.

Section 18.5: $error

```
Get-Process -Name doesnotexist
```

```
Write-Host -Object ('发生的最后一个错误是：{0}' -f $error[0].Exception.Message)
```

包含表示最近错误的错误对象数组。最近的错误是数组中的第一个错误对象（$Error[0]）。

要防止错误被添加到 $Error 数组中，请使用 ErrorAction 通用参数，值为 Ignore。更多信息，请参见 about_CommonParameters（http://go.microsoft.com/fwlink/?LinkID=113216）。

```
Write-Host -Object ('The last error that occurred was: {0}' -f $error[0].Exception.Message)
```

Contains an array of error objects that represent the most recent errors. The most recent error is the first error object in the array ($Error[0]).

To prevent an error from being added to the $Error array, use the ErrorAction common parameter with a value of Ignore. For more information, see about_CommonParameters (http://go.microsoft.com/fwlink/?LinkID=113216).

第19章：自动变量

自动变量由 Windows PowerShell 创建和维护。几乎可以调用书中任何名称的变量；唯一的例外是已经由 PowerShell 管理的变量。这些变量无疑将是你在 PowerShell 中使用频率最高的对象，仅次于函数（例如 $? - 表示上一次操作的成功/失败状态）。

第19.1节：$OFS

名为输出字段分隔符的变量包含在将数组转换为字符串时使用的字符串值。默认情况下，$OFS = " "（一个空格），但可以更改：

```
PS C:\> $array = 1,2,3
PS C:\> "$array" # 默认 OFS 将被使用
1 2 3
PS C:\> $OFS = ",." # 我们将 OFS 改为逗号和点
PS C:\> "$array"
1,.2,.3
```

第19.2节：$?

包含上一次操作的状态。当没有错误时，设置为True：

```
PS C:\> Write-Host "Hello"
Hello
PS C:\> $?
True
```

如果有错误，则设置为False：

```
PS C:\> wrt-host
wrt-host ：术语“wrt-host”未被识别为 cmdlet、函数、脚本文件或可运行程序的名称。

请检查名称的拼写，或者如果包含路径，请确认路径是否正确，然后重试。

在第 1 行 字符:1
+ wrt-host
+ ~~~~~~~~
+ CategoryInfo          : ObjectNotFound: (wrt-host:String) [], CommandNotFoundException
+ FullyQualifiedErrorId : CommandNotFoundException

PS C:\> $?
False
```

第 19.3 节：$null

$null 用于表示缺失或未定义的值。
$null 可以用作数组中空值的空占位符：

```
PS C:\> $array = 1, "string", $null
PS C:\> $array.Count
3
```

当我们使用相同的数组作为 ForEach-Object 的源时，它将处理所有三个项目（包括 $null）：

Chapter 19: Automatic Variables

Automatic Variables are created and maintained by Windows PowerShell. One has the ability to call a variable just about any name in the book; The only exceptions to this are the variables that are already being managed by PowerShell. These variables, without a doubt, will be the most repetitious objects you use in PowerShell next to functions (like **$?** - indicates Success/ Failure status of the last operation)

Section 19.1: $OFS

Variable called Output Field Separator contains string value that is used when converting an array to a string. By default `$OFS = " "` (*a space*), but it can be changed:

```
PS C:\> $array = 1,2,3
PS C:\> "$array" # default OFS will be used
1 2 3
PS C:\> $OFS = ",." # we change OFS to comma and dot
PS C:\> "$array"
1,.2,.3
```

Section 19.2: $?

Contains status of the last operation. When there is no error, it is set to `True`:

```
PS C:\> Write-Host "Hello"
Hello
PS C:\> $?
True
```

If there is some error, it is set to `False`:

```
PS C:\> wrt-host
wrt-host : The term 'wrt-host' is not recognized as the name of a cmdlet, function, script file,
or operable program.
Check the spelling of the name, or if a path was included, verify that the path is correct and try
again.
At line:1 char:1
+ wrt-host
+ ~~~~~~~~
+ CategoryInfo          : ObjectNotFound: (wrt-host:String) [], CommandNotFoundException
+ FullyQualifiedErrorId : CommandNotFoundException

PS C:\> $?
False
```

Section 19.3: $null

`$null` is used to represent absent or undefined value.
`$null` can be used as an empty placeholder for empty value in arrays:

```
PS C:\> $array = 1, "string", $null
PS C:\> $array.Count
3
```

When we use the same array as the source for `ForEach-Object`, it will process all three items (including $null):

```
PS C:\> $array | ForEach-Object {"Hello"}
Hello
Hello
Hello
```

注意！这意味着 ForEach-Object 甚至会单独处理 $null：

```
PS C:\> $null | ForEach-Object {"Hello"} # 这将执行一次迭代！！！
Hello
```

如果你将其与经典的 foreach 循环比较，这个结果非常出乎意料：

```
PS C:\> foreach($i in $null) {"Hello"} # 这将不执行任何迭代
PS C:\>
```

第19.4节：$error

最近错误对象的数组。数组中的第一个是最新的错误：

```
PS C:\> throw "Error" # 结果输出将以红色字体显示
Error
在第 1 行：1 字符：1
+ throw "Error"
+ ~~~~~~~~~~~~~
    +CategoryInfo          : OperationStopped: (Error:String) [], RuntimeException
    + FullyQualifiedErrorId : Error

PS C:\> $error[0] # 结果输出将是普通字符串（非红色）
Error
在第 1 行：1 字符：1
+ throw "Error"
+ ~~~~~~~~~~~~~
    +CategoryInfo          : OperationStopped: (Error:String) [], RuntimeException
    + FullyQualifiedErrorId : Error
```

使用提示：在格式化命令（如 format-list）中使用 $error 变量时，请注意使用 -Force 参数。否则格式化命令将以上述方式输出 $error 内容。

错误条目可以通过例如 $Error.Remove($Error[0]) 来移除。

第19.5节：$pid

包含当前托管进程的进程ID。

```
PS C:\> $pid
26080
```

第19.6节：布尔值

$true 和 $false 是表示逻辑真和假的两个变量。

注意，必须将美元符号作为第一个字符（这与C#不同）。

```
$boolExpr = "abc".Length -eq 3 # "abc"的长度是3，因此$boolExpr将为True
if($boolExpr -eq $true){
    "长度是3"
```

```
PS C:\> $array | ForEach-Object {"Hello"}
Hello
Hello
Hello
```

Be careful! This means that ForEach-Object **WILL** process even $null all by itself:

```
PS C:\> $null | ForEach-Object {"Hello"} # THIS WILL DO ONE ITERATION !!!
Hello
```

Which is very unexpected result if you compare it to classic foreach loop:

```
PS C:\> foreach($i in $null) {"Hello"} # THIS WILL DO NO ITERATION
PS C:\>
```

Section 19.4: $error

Array of most recent error objects. The first one in the array is the most recent one:

```
PS C:\> throw "Error" # resulting output will be in red font
Error
At line:1 char:1
+ throw "Error"
+ ~~~~~~~~~~~~~
    + CategoryInfo          : OperationStopped: (Error:String) [], RuntimeException
    + FullyQualifiedErrorId : Error

PS C:\> $error[0] # resulting output will be normal string (not red    )
Error
At line:1 char:1
+ throw "Error"
+ ~~~~~~~~~~~~~
    + CategoryInfo          : OperationStopped: (Error:String) [], RuntimeException
    + FullyQualifiedErrorId : Error
```

Usage hints: When using the $error variable in a format cmdlet (e.g. format-list), be aware to use the -Force switch. Otherwise the format cmdlet is going to output the $errorcontents in above shown manner.

Error entries can be removed via e.g. $Error.Remove($Error[0]).

Section 19.5: $pid

Contains process ID of the current hosting process.

```
PS C:\> $pid
26080
```

Section 19.6: Boolean values

$true and $false are two variables that represent logical TRUE and FALSE.

Note that you have to specify the dollar sign as the first character (which is different from C#).

```
$boolExpr = "abc".Length -eq 3 # length of "abc" is 3, hence $boolExpr will be True
if($boolExpr -eq $true){
    "Length is 3"
```

```
}
# 结果将是 "长度是3"
$boolExpr -ne $true
# 结果将是False
```

请注意，当你在代码中使用布尔值 true/false 时，你写作$true或$false，但当 PowerShell 返回布尔值时，它看起来像True或False

第19.7节：$_ / $PSItem

包含当前由管道处理的对象/项。

```
PS C:\> 1..5 | % { Write-Host "当前项是 $_" }
当前项是 1
当前项是 2
当前项是 3
当前项是 4
当前项是 5
```

$PSItem 和 $_ 是相同的，可以互换使用，但 $_ 是最常用的。

第19.8节：$PSVersionTable

包含一个只读哈希表（Constant, AllScope），显示当前会话中运行的 PowerShell 版本的详细信息。

```
$PSVersionTable    #此调用结果如下：
Name                              Value
----                              -----
PSVersion                         5.0.10586.117
PSCompatibleVersions              {1.0, 2.0, 3.0, 4.0...}
BuildVersion                      10.0.10586.117
CLR版本                           4.0.30319.42000
WSMan堆栈版本                     3.0
PSRemotingProtocolVersion         2.3
SerializationVersion              1.1.0.1
```

获取运行 PowerShell 版本的最快方法：

```
$PSVersionTable.PSVersion
# 结果：
主版本  次版本  内部版本  修订版本
-----  -----  -----  --------
5      0      10586  117
```

```
}
# result will be "Length is 3"
$boolExpr -ne $true
#result will be False
```

Notice that when you use boolean true/false in your code you write `$true` or `$false`, but when Powershell returns a boolean, it looks like `True` or `False`

Section 19.7: $_ / $PSItem

Contains the object/item currently being processed by the pipeline.

```
PS C:\> 1..5 | % { Write-Host "The current item is $_" }
The current item is 1
The current item is 2
The current item is 3
The current item is 4
The current item is 5
```

`$PSItem` and $_ are identical and can be used interchangeably, but $_ is by far the most commonly used.

Section 19.8: $PSVersionTable

Contains a read-only hash table (Constant, AllScope) that displays details about the version of PowerShell that is running in the current session.

```
$PSVersionTable    #this call results in this:
Name                           Value
----                           -----
PSVersion                      5.0.10586.117
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0...}
BuildVersion                   10.0.10586.117
CLRVersion                     4.0.30319.42000
WSManStackVersion              3.0
PSRemotingProtocolVersion      2.3
SerializationVersion           1.1.0.1
```

The fastest way to get a version of PowerShell running:

```
$PSVersionTable.PSVersion
# result :
Major  Minor  Build  Revision
-----  -----  -----  --------
5      0      10586  117
```

第20章：环境变量

第20.1节：Windows环境变量以名为Env:的PS驱动器形式显示：

你可以使用以下命令查看所有环境变量列表：

Get-Childitem env:

第20.2节：使用$env:即时调用环境变量：

```
$env:COMPUTERNAME
```

Chapter 20: Environment Variables

Section 20.1: Windows environment variables are visible as a PS drive called Env:

You can see list with all environment variables with:

`Get-Childitem env:`

Section 20.2: Instant call of Environment Variables with $env:

```
$env:COMPUTERNAME
```

第21章：参数展开（Splatting）

参数展开是一种将多个参数作为单个整体传递给命令的方法。具体做法是将参数及其值以键值对形式存储在哈希表中，然后使用参数展开操作符@将其传递给cmdlet。

参数展开可以使命令更易读，并允许你在多个命令调用中重用参数。

第21.1节：管道与参数展开

声明参数展开变量对于多次重用一组参数或稍作修改后重用非常有用：

```
$splat = @{
   Class = "Win32_SystemEnclosure"
   Property = "Manufacturer"
ErrorAction = "Stop"
}

Get-WmiObject -ComputerName $env:COMPUTERNAME @splat
Get-WmiObject -ComputerName "Computer2" @splat
Get-WmiObject -ComputerName "Computer3" @splat
```

但是，如果 splat 不是为了重用而缩进的，你可能不想声明它。它可以通过管道传递：

```
@{
ComputerName = $env:COMPUTERNAME
   Class = "Win32_SystemEnclosure"
   Property = "Manufacturer"
ErrorAction = "Stop"
} | % { Get-WmiObject @_ }
```

第21.2节：使用 Splatting 传递开关参数

要使用 Splatting 调用带有 -FileVersionInfo 开关的 Get-Process，类似于：

```
Get-Process -FileVersionInfo
```

使用 splatting 的调用如下：

```
$MyParameters = @{
    FileVersionInfo = $true
}

Get-Process @MyParameters
```

注意： 这很有用，因为你可以创建一组默认参数，并多次这样调用

```
$MyParameters = @{
    FileVersionInfo = $true
}

Get-Process @MyParameters -Name WmiPrvSE
Get-Process @MyParameters -Name explorer
```

Chapter 21: Splatting

Splatting is a method of passing multiple parameters to a command as a single unit. This is done by storing the parameters and their values as key-value pairs in a hashtable and splatting it to a cmdlet using the splatting operator @.

Splatting can make a command more readable and allows you to reuse parameters in multiple command calls.

Section 21.1: Piping and Splatting

Declaring the splat is useful for reusing sets of parameters multiple times or with slight variations:

```
$splat = @{
   Class = "Win32_SystemEnclosure"
   Property = "Manufacturer"
   ErrorAction = "Stop"
}

Get-WmiObject -ComputerName $env:COMPUTERNAME @splat
Get-WmiObject -ComputerName "Computer2" @splat
Get-WmiObject -ComputerName "Computer3" @splat
```

However, if the splat is not indented for reuse, you may not wish to declare it. It can be piped instead:

```
@{
   ComputerName = $env:COMPUTERNAME
   Class = "Win32_SystemEnclosure"
   Property = "Manufacturer"
   ErrorAction = "Stop"
} | % { Get-WmiObject @_ }
```

Section 21.2: Passing a Switch parameter using Splatting

To use Splatting to call `Get-Process` with the `-FileVersionInfo` switch similar to this:

```
Get-Process -FileVersionInfo
```

This is the call using splatting:

```
$MyParameters = @{
    FileVersionInfo = $true
}

Get-Process @MyParameters
```

Note: This is useful because you can create a default set of parameters and make the call many times like this

```
$MyParameters = @{
    FileVersionInfo = $true
}

Get-Process @MyParameters -Name WmiPrvSE
Get-Process @MyParameters -Name explorer
```

第21.3节：从顶层函数到一系列内部函数的参数展开（Splatting）

没有参数展开，要通过调用堆栈传递值非常繁琐。但如果将参数展开与@PSBoundParameters的强大功能结合起来，就可以将顶层的参数集合传递到各个层级。

```
函数 Outer-Method
{
    参数
    (
        [字符串]
        $First,

        [字符串]
        $Second
    )

    Write-Host ($First) -NoNewline

    Inner-Method @PSBoundParameters
}

函数 Inner-Method
{
    参数
    (
        [字符串]
        $Second
    )

    Write-Host (" {0}!" -f $Second)
}

$parameters = @{
    First = "Hello"
    Second = "World"
}


Outer-Method @parameters
```

第21.4节：参数展开

参数展开是通过在命令调用中使用包含参数和值的哈希表变量时，将美元符号$替换为参数展开操作符@来实现的。

```
$MyParameters = @{
    Name = "iexplore"
    FileVersionInfo = $true
}

Get-Process @MyParameters
```

不使用参数展开：

```
Get-Process -Name "iexplore" -FileVersionInfo
```

Section 21.3: Splatting From Top Level Function to a Series of Inner Function

Without splatting it is very cumbersome to try and pass values down through the call stack. But if you combine splatting with the power of the **@PSBoundParameters** then you can pass the top level parameter collection down through the layers.

```
Function Outer-Method
{
    Param
    (
        [string]
        $First,

        [string]
        $Second
    )

    Write-Host ($First) -NoNewline

    Inner-Method @PSBoundParameters
}

Function Inner-Method
{
    Param
    (
        [string]
        $Second
    )

    Write-Host (" {0}!" -f $Second)
}

$parameters = @{
    First = "Hello"
    Second = "World"
}


Outer-Method @parameters
```

Section 21.4: Splatting parameters

Splatting is done by replacing the dollar-sign $ with the splatting operator @ when using a variable containing a HashTable of parameters and values in a command call.

```
$MyParameters = @{
    Name = "iexplore"
    FileVersionInfo = $true
}

Get-Process @MyParameters
```

Without splatting:

```
Get-Process -Name "iexplore" -FileVersionInfo
```

您可以将普通参数与展开的参数结合使用，以便轻松地向调用中添加通用参数。

```
$MyParameters = @{
ComputerName = "StackOverflow-PC"
}

Get-Process -Name "iexplore" @MyParameters

Invoke-Command -ScriptBlock { "Something to execute remotely" } @MyParameters
```

You can combine normal parameters with splatted parameters to easily add common parameters to your calls.

```
$MyParameters = @{
    ComputerName = "StackOverflow-PC"
}

Get-Process -Name "iexplore" @MyParameters

Invoke-Command -ScriptBlock { "Something to execute remotely" } @MyParameters
```

第22章：PowerShell“流”；调试（Debug）、详细（Verbose）、警告（Warning）、错误（Error）、输出（Output）和信息（Information）

第22.1节：Write-Output

Write-Output 生成输出。该输出可以传递给管道后的下一个命令，也可以直接显示在控制台上。

该Cmdlet将对象发送到主管道，也称为“输出流”或“成功管道”。
要将错误对象发送到错误管道，请使用Write-Error。

```
# 1.) 输出到管道中的下一个Cmdlet
Write-Output '我的文本' | Out-File -FilePath "$env:TEMP\Test.txt"

Write-Output 'Bob' | ForEach-Object {
    "我的名字是 $_"
}

# 2.) 输出到控制台，因为Write-Output是管道中的最后一个命令
Write-Output '你好，世界'

# 3.) 缺少'Write-Output' CmdLet，但输出仍被视为'Write-Output'
'你好，世界'
```

1. Write-Output 命令将指定的对象发送到管道中的下一个命令。
2. 如果该命令是管道中的最后一个命令，则对象会显示在控制台中。
3. PowerShell 解释器将此视为隐式的 Write-Output。

由于Write-Output的默认行为是在管道末尾显示对象，通常不需要使用该 Cmdlet。例如，Get-Process|Write-Output等同于Get-Process。

第22.2节：写入偏好设置

消息可以通过以下方式写入；

```
Write-Verbose "详细消息"
Write-Information "信息消息"
Write-Debug "调试消息"
Write-Progress "进度消息"
Write-Warning "警告消息"
```

每个都有一个偏好变量；

```
$VerbosePreference = "SilentlyContinue"
$InformationPreference = "SilentlyContinue"
$DebugPreference = "SilentlyContinue"
$ProgressPreference = "Continue"
$WarningPreference = "Continue"
```

偏好变量控制消息和脚本后续执行的处理方式；

```
$InformationPreference = "SilentlyContinue"
Write-Information "此消息不会显示，执行将继续"
```

Chapter 22: PowerShell "Streams"; Debug, Verbose, Warning, Error, Output and Information

Section 22.1: Write-Output

Write-Output generates output. This output can go to the next command after the pipeline or to the console so it's simply displayed.

The Cmdlet sends objects down the primary pipeline, also known as the "output stream" or the "success pipeline." To send error objects down the error pipeline, use Write-Error.

```
# 1.) Output to the next Cmdlet in the pipeline
Write-Output 'My text' | Out-File -FilePath "$env:TEMP\Test.txt"

Write-Output 'Bob' | ForEach-Object {
    "My name is $_"
}

# 2.) Output to the console since Write-Output is the last command in the pipeline
Write-Output 'Hello world'

# 3.) 'Write-Output' CmdLet missing, but the output is still considered to be 'Write-Output'
'Hello world'
```

1. The Write-Output cmdlet sends the specified object down the pipeline to the next command.
2. If the command is the last command in the pipeline, the object is displayed in the console.
3. The PowerShell interpreter treats this as an implicit Write-Output.

Because Write-Output's default behavior is to display the objects at the end of a pipeline, it is generally not necessary to use the Cmdlet. For example, Get-Process | Write-Output is equivalent to Get-Process.

Section 22.2: Write Preferences

Messages can be written with;

```
Write-Verbose "Detailed Message"
Write-Information "Information Message"
Write-Debug "Debug Message"
Write-Progress "Progress Message"
Write-Warning "Warning Message"
```

Each of these has a preference variable;

```
$VerbosePreference = "SilentlyContinue"
$InformationPreference = "SilentlyContinue"
$DebugPreference = "SilentlyContinue"
$ProgressPreference = "Continue"
$WarningPreference = "Continue"
```

The preference variable controls how the message and subsequent execution of the script are handled;

```
$InformationPreference = "SilentlyContinue"
Write-Information "This message will not be shown and execution continues"
```

```
$InformationPreference = "Continue"
Write-Information "此消息会显示，执行将继续"

$InformationPreference = "Inquire"
Write-Information "此消息会显示，执行将可选择性继续"

$InformationPreference = "Stop"
Write-Information "此消息会显示，执行将终止"
```

消息的颜色可以通过设置Write-Error来控制；

```
$host.PrivateData.ErrorBackgroundColor = "Black"
$host.PrivateData.ErrorForegroundColor = "Red"
```

类似的设置也适用于Write-Verbose、Write-Debug和Write-Warning。

```
$InformationPreference = "Continue"
Write-Information "This message is shown and execution continues"

$InformationPreference = "Inquire"
Write-Information "This message is shown and execution will optionally continue"

$InformationPreference = "Stop"
Write-Information "This message is shown and execution terminates"
```

The color of the messages can be controlled for `Write-Error` by setting;

```
$host.PrivateData.ErrorBackgroundColor = "Black"
$host.PrivateData.ErrorForegroundColor = "Red"
```

Similar settings are available for `Write-Verbose`, `Write-Debug` and `Write-Warning`.

第23章：发送电子邮件

参数	详情
附件<String[]>	要附加到邮件的文件路径和文件名。路径和文件名可以通过管道传递给 Send-MailMessage。
密送<String[]>	接收邮件副本但不会显示为邮件收件人的电子邮件地址。输入姓名（可选）和电子邮件地址（必填），例如 Name someone@example.com 或 someone@example.com。
正文<String_>	邮件内容。
正文为HTML格式	表示内容为HTML格式。
抄送<String[]>	接收邮件副本的电子邮件地址。输入姓名（可选）和电子邮件地址（必填），例如 Name someone@example.com 或 someone@example.com。
凭证	指定具有权限从指定电子邮件地址发送消息的用户帐户。默认是当前用户。输入名称，如 User 或 Domain\User，或输入 PSCredential 对象。
交付通知选项	指定电子邮件消息的送达通知选项。可以使用多个值指定。交付通知将发送到"收件人"参数中指定的地址。可接受的值：无、成功时、失败时、延迟、从不。
编码	正文和主题的编码。可接受的值：ASCII、UTF8、UTF7、UTF32、Unicode、BigEndianUnicode、Default、OEM。
来自	发送邮件的邮箱地址。请输入姓名（可选）和邮箱地址（必填），例如姓名 someone@example.com 或 someone@example.com。
端口	SMTP 服务器的备用端口。默认值为 25。适用于 Windows PowerShell 3.0 及更高版本。
优先级	电子邮件消息的优先级。可接受的值：普通，高，低。
SmtpServer	发送电子邮件消息的SMTP服务器名称。默认值为$PSEmailServer变量的值。
主题	电子邮件消息的主题。
收件人	邮件发送的电子邮件地址。输入姓名（可选）和电子邮件地址（必填），例如姓名 someone@example.com 或 someone@example.com
UseSsl	使用安全套接字层（SSL）协议建立与远程计算机的连接以发送邮件

对于Exchange服务器管理员来说，一个有用的技巧是能够通过PowerShell使用SMTP发送电子邮件。根据您计算机或服务器上安装的PowerShell版本，有多种方法可以通过PowerShell发送邮件。PowerShell提供了一个简单易用的原生命令，使用该命令即可实现。

Send-MailMessage命令。

第23.1节：使用预定义参数的Send-MailMessage

```
$parameters = @{
From = 'from@bar.com'
To = 'to@bar.com'
Subject = '邮件主题'
Attachments =  @('C:\files\samplefile1.txt','C:\files\samplefile2.txt')
    BCC = 'bcc@bar.com'
Body = '邮件正文'
    BodyAsHTML = $False
    CC = 'cc@bar.com'
Credential = Get-Credential
    DeliveryNotificationOption = 'onSuccess'
    Encoding = 'UTF8'
端口 = '25'
```

Chapter 23: Sending Email

Parameter	Details
Attachments<String[]>	Path and file names of files to be attached to the message. Paths and filenames can be piped to Send-MailMessage.
Bcc<String[]>	Email addresses that receive a copy of an email message but does not appear as a recipient in the message. Enter names (optional) and the email address (required), such as Name someone@example.com or someone@example.com.
Body <String_>	Content of the email message.
BodyAsHtml	It indicates that the content is in HTML format.
Cc<String[]>	Email addresses that receive a copy of an email message. Enter names (optional) and the email address (required), such as Name someone@example.com or someone@example.com.
Credential	Specifies a user account that has permission to send message from specified email address. The default is the current user. Enter name such as User or Domain\User, or enter a PSCredential object.
DeliveryNotificationOption	Specifies the delivery notification options for the email message. Multiple values can be specified. Delivery notifications are sent in message to address specified in To parameter. Acceptable values: None, OnSuccess, OnFailure, Delay, Never.
Encoding	Encoding for the body and subject. Acceptable values: ASCII, UTF8, UTF7, UTF32, Unicode, BigEndianUnicode, Default, OEM.
From	Email addresses from which the mail is sent. Enter names (optional) and the email address (require), such as Name someone@example.com or someone@example.com.
Port	Alternate port on the SMTP server. The default value is 25. Available from Windows PowerShell 3.0.
Priority	Priority of the email message. Acceptable values: Normal, High, Low.
SmtpServer	Name of the SMTP server that sends the email message. Default value is the value of the $PSEmailServer variable.
Subject	Subject of the email message.
To	Email addresses to which the mail is sent. Enter names (optional) and the email address (required), such as Name someone@example.com or someone@example.com
UseSsl	Uses the Secure Sockets Layer (SSL) protocol to establish a connection to the remote computer to send mail

A useful technique for Exchange Server administrators is to be able to send email messages via SMTP from PowerShell. Depending on the version of PowerShell installed on your computer or server, there are multiple ways to send emails via PowerShell. There is a native cmdlet option that is simple and easy to use. It uses the cmdlet ***Send-MailMessage***.

Section 23.1: Send-MailMessage with predefined parameters

```
$parameters = @{
    From = 'from@bar.com'
    To = 'to@bar.com'
    Subject = 'Email Subject'
    Attachments =  @('C:\files\samplefile1.txt','C:\files\samplefile2.txt')
    BCC = 'bcc@bar.com'
    Body = 'Email body'
    BodyAsHTML = $False
    CC = 'cc@bar.com'
    Credential = Get-Credential
    DeliveryNotificationOption = 'onSuccess'
    Encoding = 'UTF8'
    Port = '25'
Priority = 'High'
    SmtpServer = 'smtp.com'
    UseSSL = $True
}

# 注意：Splatting 需要在变量名前使用 @ 而不是 $
Send-MailMessage @parameters
```

第23.2节：简单的 Send-MailMessage

```
Send-MailMessage -From sender@bar.com -Subject "Email Subject" -To receiver@bar.com -SmtpServer
smtp.com
```

第23.3节：SMTPClient - 邮件正文包含 .txt 文件

```
# 定义将包含在邮件正文中的 txt 文件
$Txt_File = "c:\file.txt"

function Send_mail {
    # 定义邮件设置
    $EmailFrom = "source@domain.com"
    $EmailTo = "destination@domain.com"
    $Txt_Body = Get-Content $Txt_File -RAW
    $Body = $Body_Custom + $Txt_Body
    $Subject = "Email Subject"
    $SMTPServer = "smtpserver.domain.com"
    $SMTPClient = New-Object Net.Mail.SmtpClient($SmtpServer, 25)
    $SMTPClient.EnableSsl = $false
    $SMTPClient.Send($EmailFrom, $EmailTo, $Subject, $Body)

}

$Body_Custom = "这是文件 file.txt 的内容："

Send_mail
```

```
    Priority = 'High'
    SmtpServer = 'smtp.com'
    UseSSL = $True
}

# Notice: Splatting requires @ instead of $ in front of variable name
Send-MailMessage @parameters
```

Section 23.2: Simple Send-MailMessage

```
Send-MailMessage -From sender@bar.com -Subject "Email Subject" -To receiver@bar.com -SmtpServer
smtp.com
```

Section 23.3: SMTPClient - Mail with .txt file in body message

```
# Define the txt which will be in the email body
$Txt_File = "c:\file.txt"

function Send_mail {
    #Define Email settings
    $EmailFrom = "source@domain.com"
    $EmailTo = "destination@domain.com"
    $Txt_Body = Get-Content $Txt_File -RAW
    $Body = $Body_Custom + $Txt_Body
    $Subject = "Email Subject"
    $SMTPServer = "smtpserver.domain.com"
    $SMTPClient = New-Object Net.Mail.SmtpClient($SmtpServer, 25)
    $SMTPClient.EnableSsl = $false
    $SMTPClient.Send($EmailFrom, $EmailTo, $Subject, $Body)

}

$Body_Custom = "This is what contain file.txt : "

Send_mail
```

第24章：PowerShell 远程操作

第24.1节：通过 PowerShell 连接远程服务器

使用本地计算机的凭据：

```
Enter-PSSession 192.168.1.1
```

在远程计算机上提示输入凭据

```
Enter-PSSession 192.168.1.1 -Credential $(Get-Credential)
```

第24.2节：在远程计算机上运行命令

一旦启用 PowerShell 远程操作（Enable-PSRemoting），你就可以像这样在远程计算机上运行命令：

```
Invoke-Command -ComputerName "RemoteComputerName" -ScriptBlock {
    Write-Host "远程计算机名称: $ENV:ComputerName"
}
```

上述方法会创建一个临时会话，并在命令或脚本块结束后立即关闭该会话。

如果想保持会话打开并稍后在其中运行其他命令，则需要先创建一个远程会话：

```
$Session = New-PSSession -ComputerName "RemoteComputerName"
```

然后每次在远程计算机上调用命令时都可以使用此会话：

```
Invoke-Command -Session $Session -ScriptBlock {
    Write-Host "远程计算机名称: $ENV:ComputerName"
}

Invoke-Command -Session $Session -ScriptBlock {
    Get-Date
}
```

如果需要使用不同的凭据，可以通过-Credential参数添加：

```
$Cred = Get-Credential
Invoke-Command -Session $Session -Credential $Cred -ScriptBlock {...}
```

远程序列化警告

> 注意：
>
> 需要了解的是，远程操作会在远程系统上对 PowerShell 对象进行序列化，并在远程会话的本地端对其进行反序列化，即它们在传输过程中被转换为 XML，并且丢失了所有方法。

```
$output = Invoke-Command -Session $Session -ScriptBlock {
    Get-WmiObject -Class win32_printer
}
```

Chapter 24: PowerShell Remoting

Section 24.1: Connecting to a Remote Server via PowerShell

Using credentials from your local computer:

```
Enter-PSSession 192.168.1.1
```

Prompting for credentials on the remote computer

```
Enter-PSSession 192.168.1.1 -Credential $(Get-Credential)
```

Section 24.2: Run commands on a Remote Computer

Once Powershell remoting is enabled (Enable-PSRemoting) You can run commands on the remote computer like this:

```
Invoke-Command -ComputerName "RemoteComputerName" -ScriptBlock {
    Write-Host "Remote Computer Name: $ENV:ComputerName"
}
```

The above method creates a temporary session and closes it right after the command or scriptblock ends.

To leave the session open and run other command in it later, you need to create a remote session first:

```
$Session = New-PSSession -ComputerName "RemoteComputerName"
```

Then you can use this session each time you invoke commands on the remote computer:

```
Invoke-Command -Session $Session -ScriptBlock {
    Write-Host "Remote Computer Name: $ENV:ComputerName"
}

Invoke-Command -Session $Session -ScriptBlock {
    Get-Date
}
```

If you need to use different Credentials, you can add them with the -Credential Parameter:

```
$Cred = Get-Credential
Invoke-Command -Session $Session -Credential $Cred -ScriptBlock {...}
```

Remoting serialization warning

> Note:
>
> It is important to know that remoting serializes PowerShell objects on the remote system and deserializes them on your end of the remoting session, i.e. they are converted to XML during transport and lose all of their methods.

```
$output = Invoke-Command -Session $Session -ScriptBlock {
    Get-WmiObject -Class win32_printer
}
```

```
$output | Get-Member -MemberType Method

TypeName: Deserialized.System.Management.ManagementObject#root\cimv2\Win32_Printer

Name     MemberType Definition
----     ---------- ----------
GetType  Method     type GetType()
ToString Method     string ToString(), string ToString(string format, System.IFormatProvi...
```

而常规的 PowerShell 对象则拥有这些方法：

```
Get-WmiObject -Class win32_printer | Get-Member -MemberType Method

 TypeName: System.Management.ManagementObject#root\cimv2\Win32_Printer

Name                  MemberType Definition

----                  ---------- ----------

CancelAllJobs         Method     System.Management.ManagementBaseObject CancelAllJobs()

GetSecurityDescriptor Method     System.Management.ManagementBaseObject GetSecurityDescriptor()

Pause                 Method     System.Management.ManagementBaseObject Pause()

PrintTestPage         Method     System.Management.ManagementBaseObject PrintTestPage()

RenamePrinter         Method     System.Management.ManagementBaseObject RenamePrinter(System.String
NewPrinterName)
Reset                 方法       System.Management.ManagementBaseObject Reset()

Resume                方法       System.Management.ManagementBaseObject Resume()

SetDefaultPrinter     方法       System.Management.ManagementBaseObject SetDefaultPrinter()

SetPowerState         方法       System.Management.ManagementBaseObject SetPowerState(System.UInt16
PowerState, System.String Time)
SetSecurityDescriptor 方法       System.Management.ManagementBaseObject
SetSecurityDescriptor(System.Management.ManagementObject#Win32_SecurityDescriptor 描述符)
```

参数用法

要将参数作为远程脚本块的参数使用，可以使用Invoke-Command的ArgumentList参数，或者使用$Using:语法。

使用ArgumentList传递未命名参数（即按传递给脚本块的顺序）：

```
$servicesToShow = "service1"
$fileName = "C:emp\servicestatus.csv"
Invoke-Command -Session $session -ArgumentList $servicesToShow,$fileName -ScriptBlock {
    Write-Host "远程调用脚本块，参数数量为 $($Args.Count)"
    Get-Service -Name $args[0]
    Remove-Item -Path $args[1] -ErrorAction SilentlyContinue -Force
}
```

使用ArgumentList和命名参数：

```
$servicesToShow = "service1"
$fileName = "C:emp\servicestatus.csv"
Invoke-Command -Session $session -ArgumentList $servicesToShow,$fileName -ScriptBlock {
    Param($serviceToShowInRemoteSession,$fileToDelete)
```

```
$output | Get-Member -MemberType Method

  TypeName: Deserialized.System.Management.ManagementObject#root\cimv2\Win32_Printer

Name     MemberType Definition
----     ---------- ----------
GetType  Method     type GetType()
ToString Method     string ToString(), string ToString(string format, System.IFormatProvi...
```

Whereas you have the methods on the regular PS object:

```
Get-WmiObject -Class win32_printer | Get-Member -MemberType Method

 TypeName: System.Management.ManagementObject#root\cimv2\Win32_Printer

Name                  MemberType Definition

----                  ---------- ----------

CancelAllJobs         Method     System.Management.ManagementBaseObject CancelAllJobs()

GetSecurityDescriptor Method     System.Management.ManagementBaseObject GetSecurityDescriptor()

Pause                 Method     System.Management.ManagementBaseObject Pause()

PrintTestPage         Method     System.Management.ManagementBaseObject PrintTestPage()

RenamePrinter         Method     System.Management.ManagementBaseObject RenamePrinter(System.String
NewPrinterName)
Reset                 Method     System.Management.ManagementBaseObject Reset()

Resume                Method     System.Management.ManagementBaseObject Resume()

SetDefaultPrinter     Method     System.Management.ManagementBaseObject SetDefaultPrinter()

SetPowerState         Method     System.Management.ManagementBaseObject SetPowerState(System.UInt16
PowerState, System.String Time)
SetSecurityDescriptor Method     System.Management.ManagementBaseObject
SetSecurityDescriptor(System.Management.ManagementObject#Win32_SecurityDescriptor Descriptor)
```

Argument Usage

To use arguments as parameters for the remote scripting block, one might either use the ArgumentList parameter of Invoke-Command, or use the $Using: syntax.

Using ArgumentList with unnamed parameters (i.e. in the order they are passed to the scriptblock):

```
$servicesToShow = "service1"
$fileName = "C:\temp\servicestatus.csv"
Invoke-Command -Session $session -ArgumentList $servicesToShow,$fileName -ScriptBlock {
    Write-Host "Calling script block remotely with $($Args.Count)"
    Get-Service -Name $args[0]
    Remove-Item -Path $args[1] -ErrorAction SilentlyContinue -Force
}
```

Using ArgumentList with named parameters:

```
$servicesToShow = "service1"
$fileName = "C:\temp\servicestatus.csv"
Invoke-Command -Session $session -ArgumentList $servicesToShow,$fileName -ScriptBlock {
    Param($serviceToShowInRemoteSession,$fileToDelete)
    Write-Host "远程调用脚本块，参数数量为 $($Args.Count)"
    Get-Service -Name $serviceToShowInRemoteSession
    Remove-Item -Path $fileToDelete -ErrorAction SilentlyContinue -Force
}
```

使用$Using:语法：

```
$servicesToShow = "service1"
$fileName = "C:emp\servicestatus.csv"
Invoke-Command -Session $session -ScriptBlock {
    Get-Service $Using:servicesToShow
    Remove-Item -Path $fileName -ErrorAction SilentlyContinue -Force
}
```

第24.3节：启用PowerShell远程

必须先在要远程连接的服务器上启用PowerShell远程功能。

```
Enable-PSRemoting -Force
```

此命令执行以下操作：

- 运行 Set-WSManQuickConfig cmdlet，执行以下任务：
- 启动 WinRM 服务。
- 将 WinRM 服务的启动类型设置为自动。
- 如果尚不存在，创建一个监听器以接受任何 IP 地址上的请求。
- 为 WS-Management 通信启用防火墙例外。
- 注册 Microsoft.PowerShell 和 Microsoft.PowerShell.Workflow 会话配置（如果尚未注册）。
- 在 64 位计算机上注册 Microsoft.PowerShell32 会话配置（如果尚未注册）。
- 启用所有会话配置。
- 更改所有会话配置的安全描述符以允许远程访问。
- 重启 WinRM 服务以使上述更改生效。

仅适用于非域环境

对于处于AD域中的服务器，PowerShell远程身份验证通过Kerberos（“默认”）或NTLM（“协商”）完成。如果您想允许远程连接到非域服务器，有两种选择。

要么设置通过HTTPS的WSMan通信（这需要生成证书），要么启用基本身份验证，该身份验证会将您的凭据以Base64编码形式发送（这基本上等同于明文传输，因此请谨慎使用）。

无论哪种情况，您都必须将远程系统添加到WSMan的受信任主机列表中。

启用基本身份验证

```
Set-Item WSMan:\localhost\Service\AllowUnencrypted $true
```

然后，在您希望连接的计算机上，必须告诉它信任您要连接的计算机。

```
Set-Item WSMan:\localhost\Client\TrustedHosts '192.168.1.1,192.168.1.2'
```

```
Set-Item WSMan:\localhost\Client\TrustedHosts *.contoso.com
```

```
Set-Item WSMan:\localhost\Client\TrustedHosts *
```

```
    Write-Host "Calling script block remotely with $($Args.Count)"
    Get-Service -Name $serviceToShowInRemoteSession
    Remove-Item -Path $fileToDelete -ErrorAction SilentlyContinue -Force
}
```

Using `$Using:` syntax:

```
$servicesToShow = "service1"
$fileName = "C:\temp\servicestatus.csv"
Invoke-Command -Session $session -ScriptBlock {
    Get-Service $Using:servicesToShow
    Remove-Item -Path $fileName -ErrorAction SilentlyContinue -Force
}
```

Section 24.3: Enabling PowerShell Remoting

PowerShell remoting must first be enabled on the server to which you wish to remotely connect.

```
Enable-PSRemoting -Force
```

This command does the following:

- Runs the Set-WSManQuickConfig cmdlet, which performs the following tasks:
- Starts the WinRM service.
- Sets the startup type on the WinRM service to Automatic.
- Creates a listener to accept requests on any IP address, if one does not already exist.
- Enables a firewall exception for WS-Management communications.
- Registers the Microsoft.PowerShell and Microsoft.PowerShell.Workflow session configurations, if it they are not already registered.
- Registers the Microsoft.PowerShell32 session configuration on 64-bit computers, if it is not already registered.
- Enables all session configurations.
- Changes the security descriptor of all session configurations to allow remote access.
- Restarts the WinRM service to make the preceding changes effective.

Only for non-domain environments

For servers in an AD Domain the PS remoting authentication is done through Kerberos ('Default'), or NTLM ('Negotiate'). If you want to allow remoting to a non-domain server you have two options.

Either set up WSMan communication over HTTPS (which requires certificate generation) or enable basic authentication which sends your credentials across the wire base64-encoded (that's basically the same as plain-text so be careful with this).

In either case you'll have to add the remote systems to your WSMan trusted hosts list.

Enabling Basic Authentication

```
Set-Item WSMan:\localhost\Service\AllowUnencrypted $true
```

Then on the computer you wish to connect *from*, you must tell it to trust the computer you're connecting *to*.

```
Set-Item WSMan:\localhost\Client\TrustedHosts '192.168.1.1,192.168.1.2'
```

```
Set-Item WSMan:\localhost\Client\TrustedHosts *.contoso.com
```

```
Set-Item WSMan:\localhost\Client\TrustedHosts *
```

重要提示：您必须告诉客户端以您想要连接的方式信任目标计算机（例如，如果您通过IP连接，则必须信任IP而非主机名）

第24.4节：自动清理PSSessions的最佳实践

当通过New-PSsession命令创建远程会话时，PSSession会一直存在，直到当前的PowerShell会话结束。也就是说，默认情况下，PSSession及所有相关资源将继续被使用，直到当前PowerShell会话结束。

多个活动的PSSessions可能会对资源造成压力，尤其是对于运行时间较长或相互关联的脚本，这些脚本在单个PowerShell会话中创建了数百个PSSessions。

最佳实践是在每个PSSession使用完毕后显式地将其移除。[1]

以下代码模板利用try-catch-finally结构来实现上述目标，结合错误处理和一种安全的方式，确保所有创建的PSSessions在使用完毕后被移除：

```
try
{
    $session = New-PSsession -Computername "RemoteMachineName"
    Invoke-Command -Session $session -ScriptBlock {write-host "这正在运行于
$ENV:ComputerName"}
}
catch
{
    Write-Output "错误: $_"
}
finally
{
    if ($session)
    {
        Remove-PSSession $session
    }
}
```

参考文献：[1]
https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.core/new-pssession

Important: You must tell your client to trust the computer addressed in the way you want to connect (e.g. if you connect via IP, it must trust the IP not the hostname)

Section 24.4: A best practise for automatically cleaning-up PSSessions

When a remote session is created via the New-PSsession cmdlet, the PSSession persists until the current PowerShell session ends. Meaning that, by default, the PSSession and all associated resources will continue to be used until the current PowerShell session ends.

Multiple active PSSessions can become a strain on resources, particularly for long running or interlinked scripts that create hundreds of PSSessions in a single PowerShell session.

It is best practise to explicitly remove each PSSession after it is finished being used. [1]

The following code template utilises try-catch-finally in order to achieve the above, combining error handling with a secure way to ensure all created PSSessions are removed when they are finished being used:

```
try
{
    $session = New-PSsession -Computername "RemoteMachineName"
    Invoke-Command -Session $session -ScriptBlock {write-host "This is running on
$ENV:ComputerName"}
}
catch
{
    Write-Output "ERROR: $_"
}
finally
{
    if ($session)
    {
        Remove-PSSession $session
    }
}
```

References: [1]
https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.core/new-pssession

第25章：使用PowerShell管道

PowerShell引入了一种对象管道模型，允许你将完整的对象通过管道传递给消费的命令或（至少）输出。与传统的基于字符串的管道不同，管道中对象的信息不必位于特定位置。命令可以声明以管道中的对象作为输入，而返回值会自动发送到管道。

第25.1节：编写具有高级生命周期的函数

此示例展示了函数如何接受管道输入并高效迭代。

注意，当使用管道时，函数的begin和end结构是可选的，但当使用ValueFromPipeline或ValueFromPipelineByPropertyName时，process是必需的。

```
函数Write-FromPipeline{
    [CmdletBinding()]
    param(
        [Parameter(ValueFromPipeline)]
        $myInput
    )
    begin {
        Write-Verbose -Message "开始 Write-FromPipeline"
    }
    process {
        Write-Output -InputObject $myInput
    }
    end {
        Write-Verbose -Message "Ending Write-FromPipeline"
    }
}

$foo = 'hello','world',1,2,3

$foo | Write-FromPipeline -Verbose
```

输出：

```
VERBOSE: Beginning Write-FromPipeline
hello
world
```

Chapter 25: Working with the PowerShell pipeline

PowerShell introduces an object pipelining model, which allows you to send whole objects down through the pipeline to consuming commandlets or (at least) the output. In contrast to classical string-based pipelining, information in piped objects don't have to be on specific positions. Commandlets can declare to interact with Objects from the pipeline as input, while return values are sent to the pipeline automatically.

Section 25.1: Writing Functions with Advanced Lifecycle

This example shows how a function can accept pipelined input, and iterate efficiently.

Note, that the `begin` and `end` structures of the function are optional when pipelining, but that `process` is required when using `ValueFromPipeline` or `ValueFromPipelineByPropertyName`.

```
function Write-FromPipeline{
    [CmdletBinding()]
    param(
        [Parameter(ValueFromPipeline)]
        $myInput
    )
    begin {
        Write-Verbose -Message "Beginning Write-FromPipeline"
    }
    process {
        Write-Output -InputObject $myInput
    }
    end {
        Write-Verbose -Message "Ending Write-FromPipeline"
    }
}

$foo = 'hello','world',1,2,3

$foo | Write-FromPipeline -Verbose
```

Output:

```
VERBOSE: Beginning Write-FromPipeline
hello
world
1
2
3
VERBOSE: Ending Write-FromPipeline
```

Section 25.2: Basic Pipeline Support in Functions

This is an example of a function with the simplest possible support for pipelining.
Any function with pipeline support must have at least one parameter with the ParameterAttribute `ValueFromPipeline` or `ValueFromPipelineByPropertyName` set, as shown below.

```
function Write-FromPipeline {
    param(
        [Parameter(ValueFromPipeline)]  # 这设置了ParameterAttribute
        [String]$Input
    )
    Write-Host $Input
}

$foo = 'Hello World!'

$foo | Write-FromPipeline
```

输出：

```
你好，世界！
```

注意：在 PowerShell 3.0 及以上版本中，支持 ParameterAttributes 的默认值。在早期版本中，您必须指定 `ValueFromPipeline=$true`。

第25.3节：管道的工作原理

在管道序列中，每个函数像并行线程一样同时运行。第一个处理的对象会传递到下一个管道，下一步处理会立即在另一个线程中执行。这解释了与标准 ForEach相比的高速提升。

```
@( bigFile_1, bigFile_2, ..., bigFile_n) | Copy-File | Encrypt-File | Get-Md5
```

1. 步骤 - 复制第一个文件（在 Copy-file 线程中）
2. 步骤 - 复制第二个文件（在 Copy-file 线程中），同时加密第一个文件（在 Encrypt-File 线程中）
3. 步骤 - 复制第三个文件（在Copy-file线程中）并同时加密第二个文件（在Encrypt-File中）同时获取第一个（在Get-Md5中）的Md5

```
        [Parameter(ValueFromPipeline)]  # This sets the ParameterAttribute
        [String]$Input
    )
    Write-Host $Input
}

$foo = 'Hello World!'

$foo | Write-FromPipeline
```

Output:

```
Hello World!
```

Note: In PowerShell 3.0 and above, Default Values for ParameterAttributes is supported. In earlier versions, you must specify `ValueFromPipeline=$true`.

Section 25.3: Working concept of pipeline

In a pipeline series each function runs parallel to the others, like parallel threads. The first processed object is transmitted to the next pipeline and the next processing is immediately executed in another thread. This explains the high speed gain compared to the standard `ForEach`

```
@( bigFile_1, bigFile_2, ..., bigFile_n) | Copy-File | Encrypt-File | Get-Md5
```

1. step - copy the first file (in `Copy-file` Thread)
2. step - copy second file (in `Copy-file` Thread) and simultaneously Encrypt the first (in `Encrypt-File`)
3. step - copy third file (in `Copy-file` Thread) and simultaneously encrypt second file (in `Encrypt-File`) and simultaneously `get-Md5` of the first (in `Get-Md5`)

第26章：PowerShell后台作业

作业是在PowerShell 2.0中引入的，帮助解决命令行工具固有的问题。简而言之，如果你启动一个长时间运行的任务，直到任务完成之前，提示符将不可用。作为长时间运行任务的一个例子，考虑这个简单的PowerShell命令：

Get-ChildItem -Path c:\ -Recurse

获取C盘的完整目录列表将花费一些时间。如果你将其作为作业运行，控制台将重新获得控制权，你可以稍后捕获结果。

第26.1节：基本作业创建

启动一个脚本块作为后台作业：

```
$job = Start-Job -ScriptBlock {Get-Process}
```

启动一个脚本作为后台作业：

```
$job = Start-Job -FilePath "C:\YourFolder\Script.ps1"
```

使用 Invoke-Command 在远程计算机上启动作业：

```
$job = Invoke-Command -ComputerName "ComputerName" -ScriptBlock {Get-Service winrm} -JobName
"WinRM" -ThrottleLimit 16 -AsJob
```

以不同用户身份启动作业（提示输入密码）：

```
Start-Job -ScriptBlock {Get-Process} -Credential "Domain\Username"
```

或者

```
Start-Job -ScriptBlock {Get-Process} -Credential (Get-Credential)
```

以不同用户身份启动作业（无提示）：

```
$username = "Domain\Username"
$password = "password"
$secPassword = ConvertTo-SecureString -String $password -AsPlainText -Force
$credentials = New-Object System.Management.Automation.PSCredential -ArgumentList @($username,
$secPassword)
Start-Job -ScriptBlock {Get-Process} -Credential $credentials
```

第26.2节：基本作业管理

获取当前会话中所有作业的列表：

```
Get-Job
```

等待作业完成后获取结果：

```
$job | Wait-job | Receive-Job
```

Chapter 26: PowerShell Background Jobs

Jobs were introduced in PowerShell 2.0 and helped to solve a problem inherent in the command-line tools. In a nutshell, if you start a long running task, your prompt is unavailable until the task finishes. As an example of a long running task, think of this simple PowerShell command:

Get-ChildItem -Path c:\ -Recurse

It will take a while to fetch full directory list of your C: drive. If you run it as Job then the console will get the control back and you can capture the result later on.

Section 26.1: Basic job creation

Start a Script Block as background job:

```
$job = Start-Job -ScriptBlock {Get-Process}
```

Start a script as background job:

```
$job = Start-Job -FilePath "C:\YourFolder\Script.ps1"
```

Start a job using `Invoke-Command` on a remote machine:

```
$job = Invoke-Command -ComputerName "ComputerName" -ScriptBlock {Get-Service winrm} -JobName
"WinRM" -ThrottleLimit 16 -AsJob
```

Start job as a different user (Prompts for password):

```
Start-Job -ScriptBlock {Get-Process} -Credential "Domain\Username"
```

Or

```
Start-Job -ScriptBlock {Get-Process} -Credential (Get-Credential)
```

Start job as a different user (No prompt):

```
$username = "Domain\Username"
$password = "password"
$secPassword = ConvertTo-SecureString -String $password -AsPlainText -Force
$credentials = New-Object System.Management.Automation.PSCredential -ArgumentList @($username,
$secPassword)
Start-Job -ScriptBlock {Get-Process} -Credential $credentials
```

Section 26.2: Basic job management

Get a list of all jobs in the current session:

```
Get-Job
```

Waiting on a job to finish before getting the result:

```
$job | Wait-job | Receive-Job
```

如果作业运行时间过长则超时（此处示例为10秒）

```
$job | Wait-job -Timeout 10
```

停止作业（在结束前完成该作业队列中所有待处理任务）：

```
$job | Stop-Job
```

从当前会话的后台作业列表中移除作业：

```
$job | Remove-Job
```

注意：以下内容仅适用于Workflow作业。

挂起一个Workflow作业（暂停）：

```
$job | Suspend-Job
```

恢复一个Workflow作业：

```
$job | Resume-Job
```

Timeout a job if it runs too long (10 seconds in this example)

```
$job | Wait-job -Timeout 10
```

Stopping a job (completes all tasks that are pending in that job queue before ending):

```
$job | Stop-Job
```

Remove job from current session's background jobs list:

```
$job | Remove-Job
```

Note: The following will only work on `Workflow` Jobs.

Suspend a `Workflow` Job (Pause):

```
$job | Suspend-Job
```

Resume a `Workflow` Job:

```
$job | Resume-Job
```

第27章：PowerShell中的返回行为

它可以用来退出当前作用域，该作用域可以是函数、脚本或脚本块。在PowerShell中，每条语句的结果都会作为输出返回，即使没有显式的Return关键字，也会表示作用域的结束。

第27.1节：提前退出

```
function earlyexit {
    "Hello"
    return
    "World"
}
```

“Hello”将被放入输出管道，“World”不会

第27.2节：注意！管道中的 return

```
get-childitem | foreach-object { if ($_.IsReadOnly) { return } }
```

管道命令（例如：ForEach-Object，Where-Object等）作用于闭包。这里的 return 只会跳到管道中的下一个项目，而不会退出处理。如果想退出处理，可以使用 break 代替 return。

```
get-childitem | foreach-object { if ($_.IsReadOnly) { break } }
```

第27.3节：带返回值的 return

（改写自 about_return）

以下方法在管道中的值相同

```
function foo {
    $a = "Hello"
    return $a
}

function bar {
    $a = "Hello"
    $a
    返回
}

函数 quux {
    $a = "Hello"
    $a
}
```

第27.4节：如何使用函数返回值

函数返回所有未被其他内容捕获的东西。
如果你使用return关键字，return行之后的所有语句都不会被执行！

像这样：

Chapter 27: Return behavior in PowerShell

It can be used to Exit the current scope, which can be a function, script, or script block. In PowerShell, the result of each statement is returned as output, even without an explicit Return keyword or to indicate that the end of the scope has been reached.

Section 27.1: Early exit

```
function earlyexit {
    "Hello"
    return
    "World"
}
```

"Hello" will be placed in the output pipeline, "World" will not

Section 27.2: Gotcha! Return in the pipeline

```
get-childitem | foreach-object { if ($_.IsReadOnly) { return } }
```

Pipeline cmdlets (ex: `ForEach-Object`, `Where-Object`, etc) operate on closures. The return here will only move to the next item on the pipeline, not exit processing. You can use **break** instead of **return** if you want to exit processing.

```
get-childitem | foreach-object { if ($_.IsReadOnly) { break } }
```

Section 27.3: Return with a value

(paraphrased from about_return)

The following methods will have the same values on the pipeline

```
function foo {
    $a = "Hello"
    return $a
}

function bar {
    $a = "Hello"
    $a
    return
}

function quux {
    $a = "Hello"
    $a
}
```

Section 27.4: How to work with functions returns

A function returns everything that is not captured by something else.
If u use the **return** keyword, every statement after the return line will not be executed!

Like this:

```
函数Test-Function
{
    参数
    (
        [switch]$ExceptionalReturn
    )
    “开始”
    if($ExceptionalReturn){Return "该死，没成功！"}
    New-ItemProperty -Path "HKCU:\" -Name "test" -Value "TestValue" -Type "String"
    Return "是的，成功了！"
}
```

测试函数
将返回：

- 开始
- 新创建的注册表项（这是因为有些语句会产生你可能不预期的输出）
- 是的，成功了！

测试函数 -异常返回 将返回：

- 开始
- 该死，失败了！

如果你这样做：

```
函数Test-Function
{
    参数
    (
        [切换]$ExceptionalReturn
    )
    . {
       “开始”
        如果($ExceptionalReturn)
        {
            $Return = “该死，没成功！”
            Return
        }
        New-ItemProperty -Path “HKCU:\” -Name “test” -Value “TestValue” -Type “String”
        $Return = “是的，成功了！”
        Return
    } | Out-Null
    Return $Return
}
```

测试函数
将返回：

- 是的，成功了！

测试函数 -异常返回 将返回：

- 该死，失败了！

通过这个技巧，即使你不确定每条语句会输出什么，也能控制返回的结果。

```
Function Test-Function
{
    Param
    (
        [switch]$ExceptionalReturn
    )
    "Start"
    if($ExceptionalReturn){Return "Damn, it didn't work!"}
    New-ItemProperty -Path "HKCU:\" -Name "test" -Value "TestValue" -Type "String"
    Return "Yes, it worked!"
}
```

Test-Function
Will return:

- Start
- The newly created registry key (this is because there are some statements that create output that you may not be expecting)
- Yes, it worked!

Test-Function -ExceptionalReturn Will return:

- Start
- Damn, it didn't work!

If you do it like this:

```
Function Test-Function
{
    Param
    (
        [switch]$ExceptionalReturn
    )
    . {
       "Start"
        if($ExceptionalReturn)
        {
            $Return = "Damn, it didn't work!"
            Return
        }
        New-ItemProperty -Path "HKCU:\" -Name "test" -Value "TestValue" -Type "String"
        $Return = "Yes, it worked!"
        Return
    } | Out-Null
    Return $Return
}
```

Test-Function
Will return:

- Yes, it worked!

Test-Function -ExceptionalReturn Will return:

- Damn, it didn't work!

With this trick you can control the returned output even if you are not sure what will each statement will spit out.

它的工作原理是这样的

```
.{<语句>} | Out-Null
```

. 使得后面的脚本块包含在代码中
{} 标记脚本块
| Out-Null 将任何意外输出传递给 Out-Null（因此输出被丢弃！）
因为脚本块被包含进来，它获得了与函数其余部分相同的作用域。
所以你可以访问在脚本块内创建的变量。

第27.5节：抓住了！忽略不需要的输出

灵感来源于

- PowerShell：函数没有正确的返回值

```
function bar {
 [System.Collections.ArrayList]$MyVariable = @()
 $MyVariable.Add("a") | Out-Null
 $MyVariable.Add("b") | Out-Null
 $MyVariable
}
```

使用Out-Null是必要的，因为.NET的ArrayList.Add方法在添加后会返回集合中的项目数。
如果省略，管道中将包含1、2、"a"、"b"

有多种方法可以省略不需要的输出：

```
function bar
{
    # New-Item cmdlet 返回新创建的文件/文件夹的信息
    New-Item "test1.txt" | out-null
    New-Item "test2.txt" > $null
    [void](New-Item "test3.txt")
    $tmp = New-Item "test4.txt"
}
```

注意： 想了解为何更倾向于使用 > $null，请参见[主题尚未创建]。

It works like this

```
.{<Statements>} | Out-Null
```

the . makes the following scriptblock included in the code
the {} marks the script block
the | Out-Null pipes any unexpected output to Out-Null (so it is gone!)
Because the scriptblock is included it gets the same scope as the rest of the function.
So you can access variables who were made inside the scriptblock.

Section 27.5: Gotcha! Ignoring unwanted output

Inspired by

- PowerShell: Function doesn't have proper return value

```
function bar {
 [System.Collections.ArrayList]$MyVariable = @()
 $MyVariable.Add("a") | Out-Null
 $MyVariable.Add("b") | Out-Null
 $MyVariable
}
```

The Out-Null is necessary because the .NET ArrayList.Add method returns the number of items in the collection after adding. If omitted, the pipeline would have contained 1, 2, "a", "b"

There are multiple ways to omit unwanted output:

```
function bar
{
    # New-Item cmdlet returns information about newly created file/folder
    New-Item "test1.txt" | out-null
    New-Item "test2.txt" > $null
    [void](New-Item "test3.txt")
    $tmp = New-Item "test4.txt"
}
```

Note: to learn more about why to prefer > $null, see [topic not yet created].

第28章：CSV解析

第28.1节：Import-Csv的基本用法

给定以下CSV文件

```
String,DateTime,Integer
First,2016-12-01T12:00:00,30
Second,2015-12-01T12:00:00,20
Third,2015-12-01T12:00:00,20
```

可以使用Import-Csv命令将CSV行导入为PowerShell对象

```
> $listOfRows = Import-Csv .\example.csv
> $listOfRows
```

```
String  DateTime            Integer
------  --------            -------
First   2016-12-01T12:00:00  30
Second  2015-11-03T13:00:00  20
Third   2015-12-05T14:00:00  20
```

```
> Write-Host $row[0].String1
```

```
第三
```

第28.2节：从CSV导入并将属性转换为正确类型

默认情况下，Import-CSV 会将所有值导入为字符串，因此要获得DateTime和整数对象，我们需要对它们进行类型转换或解析。

使用 Foreach-Object：

```
> $listOfRows = Import-Csv .\example.csv
> $listOfRows | ForEach-Object {
    #转换属性类型
    $_.DateTime = [datetime]$_.DateTime
    $_.Integer = [int]$_.Integer

    #输出对象
    $_
}
```

使用计算属性：

```
> $listOfRows = Import-Csv .\example.csv
> $listOfRows | Select-Object String,
    @{name="DateTime";expression={ [datetime]$_.DateTime }},
    @{name="Integer";expression={ [int]$_.Integer }}
```

输出：

Chapter 28: CSV parsing

Section 28.1: Basic usage of Import-Csv

Given the following CSV-file

```
String,DateTime,Integer
First,2016-12-01T12:00:00,30
Second,2015-12-01T12:00:00,20
Third,2015-12-01T12:00:00,20
```

One can import the CSV rows in PowerShell objects using the `Import-Csv` command

```
> $listOfRows = Import-Csv .\example.csv
> $listOfRows
```

```
String DateTime            Integer
------ --------            -------
First  2016-12-01T12:00:00 30
Second 2015-11-03T13:00:00 20
Third  2015-12-05T14:00:00 20
```

```
> Write-Host $row[0].String1
```

```
Third
```

Section 28.2: Import from CSV and cast properties to correct type

By default, `Import-CSV` imports all values as strings, so to get DateTime- and integer-objects, we need to cast or parse them.

Using `Foreach-Object`:

```
> $listOfRows = Import-Csv .\example.csv
> $listOfRows | ForEach-Object {
    #Cast properties
    $_.DateTime = [datetime]$_.DateTime
    $_.Integer = [int]$_.Integer

    #Output object
    $_
}
```

Using calculated properties:

```
> $listOfRows = Import-Csv .\example.csv
> $listOfRows | Select-Object String,
    @{name="DateTime";expression={ [datetime]$_.DateTime }},
    @{name="Integer";expression={ [int]$_.Integer }}
```

Output:

```
字符串 日期时间            整数
------ --------            -------
第一   2016.12.01 12:00:00       30
第二   2015.11.03 13:00:00       20
第三   2015.12.05 14:00:00       20
```

```
String DateTime            Integer
------ --------            -------
First  01.12.2016 12:00:00      30
Second 03.11.2015 13:00:00      20
Third  05.12.2015 14:00:00      20
```

第29章：处理XML文件

第29.1节：访问XML文件

```
<!-- file.xml -->
<people>
    <person id="101">
        <name>乔恩·拉乔伊</name>
        <age>22</age>
    </person>
    <person id="102">
        <name>加本勋爵</name>
        <age>65</age>
    </person>
    <person id="103">
        <name>戈登·弗里曼</name>
        <age>29</age>
    </person>
</people>
```

加载 XML 文件

要加载XML文件，可以使用以下任意一种方法：

```
# 第一种方法
$xdoc = New-Object System.Xml.XmlDocument
$file = Resolve-Path(".\file.xml")
$xdoc.load($file)

# 第二种方法
[xml] $xdoc = Get-Content ".\file.xml"

# 第三种方法
$xdoc = [xml] (Get-Content ".\file.xml")
```

以对象形式访问XML

```
PS C:\> $xml = [xml](Get-Content file.xml)
PS C:\> $xml

PS C:\> $xml.people

person
--------
{乔恩·拉乔伊, 加本勋爵, 戈登·弗里曼}



id                                姓名                                年龄
--                                       ----                                    ---
101乔恩·拉乔伊                              22
102加本勋爵                               65
103戈登·弗里曼                            29

PS C:\> $xml.people.person[0].name
乔恩·拉乔伊

PS C:\> $xml.people.person[1].age
65
```

Chapter 29: Working with XML Files

Section 29.1: Accessing an XML File

```
<!-- file.xml -->
<people>
    <person id="101">
        <name>Jon Lajoie</name>
        <age>22</age>
    </person>
    <person id="102">
        <name>Lord Gaben</name>
        <age>65</age>
    </person>
    <person id="103">
        <name>Gordon Freeman</name>
        <age>29</age>
    </person>
</people>
```

Loading an XML File

To load an XML file, you can use any of these:

```
# First Method
$xdoc = New-Object System.Xml.XmlDocument
$file = Resolve-Path(".\file.xml")
$xdoc.load($file)

# Second Method
[xml] $xdoc = Get-Content ".\file.xml"

# Third Method
$xdoc = [xml] (Get-Content ".\file.xml")
```

Accessing XML as Objects

```
PS C:\> $xml = [xml](Get-Content file.xml)
PS C:\> $xml

PS C:\> $xml.people

person
--------
{Jon Lajoie, Lord Gaben, Gordon Freeman}

PS C:\> $xml.people.person

id                                      name                                    age
--                                      ----                                    ---
101                                     Jon Lajoie                              22
102                                     Lord Gaben                              65
103                                     Gordon Freeman                          29

PS C:\> $xml.people.person[0].name
Jon Lajoie

PS C:\> $xml.people.person[1].age
65
```

```
PS C:\> $xml.people.person[2].id
103
```

使用XPath访问XML

```
PS C:\> $xml = [xml](Get-Content file.xml)
PS C:\> $xml

PS C:\> $xml.SelectNodes("//people")

person
--------
{乔恩·拉乔伊, 加本勋爵, 戈登·弗里曼}

PS C:\> $xml.SelectNodes("//people//person")id

                      name                          age
--                              ----                              ---
101乔恩·拉乔伊                          22
102加本勋爵                           65
103戈登·弗里曼                        29

PS C:\> $xml.SelectSingleNode("people//person[1]//name")
乔恩·拉乔伊

PS C:\> $xml.SelectSingleNode("people//person[2]//age")
65

PS C:\> $xml.SelectSingleNode("people//person[3]//@id")
103
```

使用XPath访问包含命名空间的XML

```
PS C:\> [xml]$xml = @"
<ns:people xmlns:ns="http://schemas.xmlsoap.org/soap/envelope/">
    <ns:person id="101">
<ns:name>乔恩·拉乔伊</ns:name>
    </ns:person>
<ns:person id="102">
       <ns:name>盖布勋爵</ns:name>
    </ns:person>
<ns:person id="103">
       <ns:name>戈登·弗里曼</ns:name>
    </ns:person>
</ns:people>
"@

PS C:\> $ns = new-object Xml.XmlNamespaceManager $xml.NameTable
PS C:\> $ns.AddNamespace("ns", $xml.DocumentElement.NamespaceURI)
PS C:\> $xml.SelectNodes("//ns:people/ns:person", $ns)

id                              name
--                              ----
101乔恩·拉乔伊
102盖布勋爵
103戈登·弗里曼
```

第29.2节：使用XmlWriter()创建XML文档

```
# 设置格式
```

```
PS C:\> $xml.people.person[2].id
103
```

Accessing XML with XPath

```
PS C:\> $xml = [xml](Get-Content file.xml)
PS C:\> $xml

PS C:\> $xml.SelectNodes("//people")

person
--------
{Jon Lajoie, Lord Gaben, Gordon Freeman}

PS C:\> $xml.SelectNodes("//people//person")

id                                      name                                    age
--                                      ----                                    ---
101                                     Jon Lajoie                              22
102                                     Lord Gaben                              65
103                                     Gordon Freeman                          29

PS C:\> $xml.SelectSingleNode("people//person[1]//name")
Jon Lajoie

PS C:\> $xml.SelectSingleNode("people//person[2]//age")
65

PS C:\> $xml.SelectSingleNode("people//person[3]//@id")
103
```

Accessing XML containing namespaces with XPath

```
PS C:\> [xml]$xml = @"
<ns:people xmlns:ns="http://schemas.xmlsoap.org/soap/envelope/">
    <ns:person id="101">
        <ns:name>Jon Lajoie</ns:name>
    </ns:person>
    <ns:person id="102">
        <ns:name>Lord Gaben</ns:name>
    </ns:person>
    <ns:person id="103">
        <ns:name>Gordon Freeman</ns:name>
    </ns:person>
</ns:people>
"@

PS C:\> $ns = new-object Xml.XmlNamespaceManager $xml.NameTable
PS C:\> $ns.AddNamespace("ns", $xml.DocumentElement.NamespaceURI)
PS C:\> $xml.SelectNodes("//ns:people/ns:person", $ns)

id                                      name
--                                      ----
101                                     Jon Lajoie
102                                     Lord Gaben
103                                     Gordon Freeman
```

Section 29.2: Creating an XML Document using XmlWriter()

```
# Set The Formatting
```

```
$xmlsettings = New-Object System.Xml.XmlWriterSettings
$xmlsettings.Indent = $true
$xmlsettings.IndentChars = "    "

# 设置文件名并创建文档
$XmlWriter = [System.XML.XmlWriter]::Create("C:\YourXML.xml", $xmlsettings)

# 写入XML声明并设置XSL
$xmlWriter.WriteStartDocument()
$xmlWriter.WriteProcessingInstruction("xml-stylesheet", "type='text/xsl' href='style.xsl'")

# 开始根元素
$xmlWriter.WriteStartElement("Root")

    $xmlWriter.WriteStartElement("Object") # <-- 开始 <Object>

        $xmlWriter.WriteElementString("Property1","Value 1")
        $xmlWriter.WriteElementString("Property2","Value 2")

        $xmlWriter.WriteStartElement("SubObject") # <-- 开始 <SubObject>
            $xmlWriter.WriteElementString("Property3","Value 3")
        $xmlWriter.WriteEndElement() # <-- 结束 <SubObject>

    $xmlWriter.WriteEndElement() # <-- 结束 <Object>

$xmlWriter.WriteEndElement() # <-- 结束 <Root>

# 结束，完成并关闭XML文档
$xmlWriter.WriteEndDocument()
$xmlWriter.Flush()
$xmlWriter.Close()
```

输出XML文件

```
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet type='text/xsl' href='style.xsl'?>
<Root>
    <Object>
        <Property1>值 1</Property1>
        <Property2>值 2</Property2>
        <SubObject>
            <Property3>值 3</Property3>
        </SubObject>
    </Object>
</Root>
```

第29.3节：向当前XMLDocument添加XML片段

示例数据

XML文档

首先，让我们在当前目录中定义一个名为"books.xml"的示例XML文档：

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
    <book>
        <title>人鼠之间</title>
        <author>约翰·斯坦贝克</author>
        <pageCount>187</pageCount>
        <publishers>
```

```
$xmlsettings = New-Object System.Xml.XmlWriterSettings
$xmlsettings.Indent = $true
$xmlsettings.IndentChars = "    "

# Set the File Name Create The Document
$XmlWriter = [System.XML.XmlWriter]::Create("C:\YourXML.xml", $xmlsettings)

# Write the XML Declaration and set the XSL
$xmlWriter.WriteStartDocument()
$xmlWriter.WriteProcessingInstruction("xml-stylesheet", "type='text/xsl' href='style.xsl'")

# Start the Root Element
$xmlWriter.WriteStartElement("Root")

    $xmlWriter.WriteStartElement("Object") # <-- Start <Object>

        $xmlWriter.WriteElementString("Property1","Value 1")
        $xmlWriter.WriteElementString("Property2","Value 2")

        $xmlWriter.WriteStartElement("SubObject") # <-- Start <SubObject>
            $xmlWriter.WriteElementString("Property3","Value 3")
        $xmlWriter.WriteEndElement() # <-- End <SubObject>

    $xmlWriter.WriteEndElement() # <-- End <Object>

$xmlWriter.WriteEndElement() # <-- End <Root>

# End, Finalize and close the XML Document
$xmlWriter.WriteEndDocument()
$xmlWriter.Flush()
$xmlWriter.Close()
```

Output XML File

```
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet type='text/xsl' href='style.xsl'?>
<Root>
    <Object>
        <Property1>Value 1</Property1>
        <Property2>Value 2</Property2>
        <SubObject>
            <Property3>Value 3</Property3>
        </SubObject>
    </Object>
</Root>
```

Section 29.3: Adding snippets of XML to current XMLDocument

Sample Data

XML Document

First, let's define a sample XML document named "**books.xml**" in our current directory:

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
    <book>
        <title>Of Mice And Men</title>
        <author>John Steinbeck</author>
        <pageCount>187</pageCount>
        <publishers>
```

```
            <publisher>
                <isbn>978-88-58702-15-4</isbn>
                <name>帕斯卡尔·科维奇</name>
                <year>1937</year>
                <binding>精装</binding>
                <first>true</first>
            </publisher>
            <publisher>
                <isbn>978-05-82461-46-8</isbn>
                <name>朗文</name>
                <year>2009</year>
                <binding>精装</binding>
            </publisher>
        </publishers>
        <characters>
            <character name="伦尼·斯莫尔" />
            <character name="柯利的妻子" />
            <character name="乔治·米尔顿" />
            <character name="柯利" />
        </characters>
        <film>真实</film>
    </book>
    <book>
        <title>《猎杀红色十月》</title>
        <author>汤姆·克兰西</author>
        <pageCount>387</pageCount>
        <publishers>
            <publisher>
                <isbn>978-08-70212-85-7</isbn>
                <name>海军研究所出版社</name>
                <year>1984</year>
                <binding>精装</binding>
                <first>true</first>
            </publisher>
            <publisher>
                <isbn>978-04-25083-83-3</isbn>
                <name>伯克利</name>
                <year>1986</year>
                <binding>平装</binding>
            </publisher>
            <publisher>
                <isbn>978-08-08587-35-4</isbn>
                <name>企鹅普特南</name>
                <year>2010</year>
                <binding>平装</binding>
            </publisher>
        </publishers>
        <characters>
            <character name="马克·亚历山德罗维奇·拉米乌斯" />
            <character name="杰克·瑞恩" />
            <character name="格里尔上将" />
            <character name="巴特·曼库索" />
            <character name="瓦西里·博罗丁" />
        </characters>
        <film>真实</film>
    </book>
</books>
```

新数据

我们想做的是向这份文档中添加几本新书，比如*《爱国者游戏》*，作者是汤姆·克兰西（是的，我是克兰西作品的粉丝^_^），还有一本科幻经典：*《银河系漫游指南》*，作者是道格拉斯·亚当斯，主要是因为

```
            <publisher>
                <isbn>978-88-58702-15-4</isbn>
                <name>Pascal Covici</name>
                <year>1937</year>
                <binding>Hardcover</binding>
                <first>true</first>
            </publisher>
            <publisher>
                <isbn>978-05-82461-46-8</isbn>
                <name>Longman</name>
                <year>2009</year>
                <binding>Hardcover</binding>
            </publisher>
        </publishers>
        <characters>
            <character name="Lennie Small" />
            <character name="Curley's Wife" />
            <character name="George Milton" />
            <character name="Curley" />
        </characters>
        <film>True</film>
    </book>
    <book>
        <title>The Hunt for Red October</title>
        <author>Tom Clancy</author>
        <pageCount>387</pageCount>
        <publishers>
            <publisher>
                <isbn>978-08-70212-85-7</isbn>
                <name>Naval Institute Press</name>
                <year>1984</year>
                <binding>Hardcover</binding>
                <first>true</first>
            </publisher>
            <publisher>
                <isbn>978-04-25083-83-3</isbn>
                <name>Berkley</name>
                <year>1986</year>
                <binding>Paperback</binding>
            </publisher>
            <publisher>
                <isbn>978-08-08587-35-4</isbn>
                <name>Penguin Putnam</name>
                <year>2010</year>
                <binding>Paperback</binding>
            </publisher>
        </publishers>
        <characters>
            <character name="Marko Alexadrovich Ramius" />
            <character name="Jack Ryan" />
            <character name="Admiral Greer" />
            <character name="Bart Mancuso" />
            <character name="Vasily Borodin" />
        </characters>
        <film>True</film>
    </book>
</books>
```

New Data

What we want to do is add a few new books to this document, let's say *Patriot Games* by Tom Clancy (yes, I'm a fan of Clancy's works ^__^) and a Sci-Fi favourite: *The Hitchhiker's Guide to the Galaxy* by Douglas Adams mainly because

扎福德·比布尔布罗克斯读起来非常有趣。

不知怎么的，我们已经获取了新书的数据，并将其保存为 PSCustomObjects 列表：

```
$newBooks = @(
    [PSCustomObject] @{
        "Title" = "爱国者游戏";
        "Author" = "汤姆·克兰西";
        "PageCount" = 540;
        "Publishers" = @(
            [PSCustomObject] @{
                "ISBN" = "978-0-39-913241-4";
                "Year" = "1987";
                "First" = $True;
                "Name" = "普特南";
                "Binding" = "精装";
            }
        );
        "Characters" = @(
            "杰克·瑞恩", "威尔士亲王", "威尔士公主",
            "罗比·杰克逊", "凯西·瑞恩", "肖恩·帕特里克·米勒"
        );
        "film" = $True;
    },
    [PSCustomObject] @{
        "标题" = "银河系漫游指南";
        "作者" = "道格拉斯·亚当斯";
        "页数" = 216;
        "Publishers" = @(
            [PSCustomObject] @{
                "ISBN" = "978-0-33-025864-7";
                "年份" = "1979";
                "首次" = $True;
                "名称" = "潘书出版社";
                "装帧" = "精装";
            }
        );
        "Characters" = @(
            "亚瑟·登特", "马文", "扎福德·比布尔布罗克斯", "福特·普雷费克特",
            "特里利安", "斯拉蒂巴特法斯特", "德克·詹特利"
        );
        "电影" = $True;
    }
);
```

模板

现在我们需要为新数据定义几个骨架XML结构。基本上，你需要为每个数据列表创建一个骨架/模板。在我们的例子中，这意味着我们需要为书籍、角色和出版社创建模板。我们还可以用它来定义一些默认值，比如film标签的值。

```
$t_book = [xml] @'
<book>
<title />
    <author />
    <pageCount />
    <publishers />
    <characters />
    <film>False</film>
</book>
'@;
```

Zaphod Beeblebrox is just fun to read.

Somehow we've acquired the data for the new books and saved them as a list of PSCustomObjects:

```
$newBooks = @(
    [PSCustomObject] @{
        "Title" = "Patriot Games";
        "Author" = "Tom Clancy";
        "PageCount" = 540;
        "Publishers" = @(
            [PSCustomObject] @{
                "ISBN" = "978-0-39-913241-4";
                "Year" = "1987";
                "First" = $True;
                "Name" = "Putnam";
                "Binding" = "Hardcover";
            }
        );
        "Characters" = @(
            "Jack Ryan", "Prince of Wales", "Princess of Wales",
            "Robby Jackson", "Cathy Ryan", "Sean Patrick Miller"
        );
        "film" = $True;
    },
    [PSCustomObject] @{
        "Title" = "The Hitchhiker's Guide to the Galaxy";
        "Author" = "Douglas Adams";
        "PageCount" = 216;
        "Publishers" = @(
            [PSCustomObject] @{
                "ISBN" = "978-0-33-025864-7";
                "Year" = "1979";
                "First" = $True;
                "Name" = "Pan Books";
                "Binding" = "Hardcover";
            }
        );
        "Characters" = @(
            "Arthur Dent", "Marvin", "Zaphod Beeblebrox", "Ford Prefect",
            "Trillian", "Slartibartfast", "Dirk Gently"
        );
        "film" = $True;
    }
);
```

Templates

Now we need to define a few skeleton XML structures for our new data to go into. Basically, you want to create a skeleton/template for each list of data. In our example, that means we need a template for the book, characters, and publishers. We can also use this to define a few default values, such as the value for the `film` tag.

```
$t_book = [xml] @'
<book>
    <title />
    <author />
    <pageCount />
    <publishers />
    <characters />
    <film>False</film>
</book>
'@;
```

```
$t_publisher = [xml] @'
<publisher>
    <isbn/>
    <name/>
    <year/>
<binding/>
   <first>false</first>
</publisher>
'@;

$t_character = [xml] @'
<character name="" />
'@;
```

我们完成了设置。

添加新数据

现在我们已经准备好了示例数据，接下来将自定义对象添加到 XML 文档对象中。

```
# 读取 XML 文档
$xml = [xml] Get-Content .\books.xml;

# 显示书名列表，看看当前有哪些内容：
$xml.books.book | Select Title, Author, @{N="ISBN";E={If ( $_.Publishers.Publisher.Count ) {
$_.Publishers.publisher[0].ISBN} Else { $_.Publishers.publisher.isbn}}};;

# 输出结果：
# title                                   author         ISBN
# -----                                   ------         ----
# Of Mice And Men                         John Steinbeck 978-88-58702-15-4
# The Hunt for Red October                Tom Clancy     978-08-70212-85-7

# 现在也显示我们的新书：
$newBooks | Select Title, Author, @{N="ISBN";E={$_.Publishers[0].ISBN}};

# 输出结果：
# Title                                   Author        ISBN
# -----                                   ------        ----
# Patriot Games                           Tom Clancy    978-0-39-913241-4
# The Hitchhiker's Guide to the Galaxy Douglas Adams 978-0-33-025864-7

# 现在合并这两个列表：

ForEach ( $book in $newBooks ) {
    $root = $xml.SelectSingleNode("/books");

    # 将书籍模板作为新节点添加到根元素中
    [void]$root.AppendChild($xml.ImportNode($t_book.book, $true));

    # 选择新的子元素
    $newElement = $root.SelectSingleNode("book[last()]");

    # 更新该新元素的参数以匹配我们当前的新书数据
    $newElement.title     = [String]$book.Title;
    $newElement.author    = [String]$book.Author;
    $newElement.pageCount = [String]$book.PageCount;
    $newElement.film      = [String]$book.Film;

    # 遍历作为新元素子节点的属性：
    ForEach ( $publisher in $book.Publishers ) {
```

```
$t_publisher = [xml] @'
<publisher>
    <isbn/>
    <name/>
    <year/>
    <binding/>
    <first>false</first>
</publisher>
'@;

$t_character = [xml] @'
<character name="" />
'@;
```

We're done with set-up.

Adding the new data

Now that we're all set-up with our sample data, let's add the custom objects to the XML Document Object.

```
# Read the xml document
$xml = [xml] Get-Content .\books.xml;

# Let's show a list of titles to see what we've got currently:
$xml.books.book | Select Title, Author, @{N="ISBN";E={If ( $_.Publishers.Publisher.Count ) {
$_.Publishers.publisher[0].ISBN} Else { $_.Publishers.publisher.isbn}}};;

# Outputs:
# title                                   author         ISBN
# -----                                   ------         ----
# Of Mice And Men                         John Steinbeck 978-88-58702-15-4
# The Hunt for Red October                Tom Clancy     978-08-70212-85-7

# Let's show our new books as well:
$newBooks | Select Title, Author, @{N="ISBN";E={$_.Publishers[0].ISBN}};

# Outputs:
# Title                                   Author        ISBN
# -----                                   ------        ----
# Patriot Games                           Tom Clancy    978-0-39-913241-4
# The Hitchhiker's Guide to the Galaxy Douglas Adams 978-0-33-025864-7

# Now to merge the two:

ForEach ( $book in $newBooks ) {
    $root = $xml.SelectSingleNode("/books");

    # Add the template for a book as a new node to the root element
    [void]$root.AppendChild($xml.ImportNode($t_book.book, $true));

    # Select the new child element
    $newElement = $root.SelectSingleNode("book[last()]");

    # Update the parameters of that new element to match our current new book data
    $newElement.title     = [String]$book.Title;
    $newElement.author    = [String]$book.Author;
    $newElement.pageCount = [String]$book.PageCount;
    $newElement.film      = [String]$book.Film;

    # Iterate through the properties that are Children of our new Element:
    ForEach ( $publisher in $book.Publishers ) {
        # 创建新的子发布者元素
        # 注意这里使用了“SelectSingleNode”，这允许使用“AppendChild”方法
因为它返回的是
        # 一个 XmlElement 类型的对象，而不是当前存储在该叶子节点中的 $Null 数据
在
        # XML 文档树中

[void]$newElement.SelectSingleNode("publishers").AppendChild($xml.ImportNode($t_publisher.publisher
, $true));

        # 更新我们新 XML 元素的属性和值以匹配我们的新数据
        $newPublisherElement = $newElement.SelectSingleNode("publishers/publisher[last()]");
        $newPublisherElement.year = [String]$publisher.Year;
        $newPublisherElement.name = [String]$publisher.Name;
        $newPublisherElement.binding = [String]$publisher.Binding;
        $newPublisherElement.isbn = [String]$publisher.ISBN;
        如果( $publisher.first ) {
            $newPublisherElement.first = "True";
        }
    }

    对于每个( $character in $book.Characters ) {
        # 选择 characters XML 元素
        $charactersElement = $newElement.SelectSingleNode("characters");

        # 添加一个新的 character 子元素
        [void]$charactersElement.AppendChild($xml.ImportNode($t_character.character, $true));

        # 选择新的 characters/character 元素
        $characterElement = $charactersElement.SelectSingleNode("character[last()]");

        # 更新属性和值以匹配我们的新数据
        $characterElement.name = [String]$character;
    }
}

# 查看新的 XML：
$xml.books.book | Select Title, Author, @{N="ISBN";E={If ( $_.Publishers.Publisher.Count ) {
$_.Publishers.publisher[0].ISBN} Else { $_.Publishers.publisher.isbn}}};

# 输出结果：
# 书名                          作者          ISBN
# -----                              ------          ----
# 《人鼠之间》                  约翰·斯坦贝克 978-88-58702-15-4
# 《红色十月的猎杀》              汤姆·克兰西      978-08-70212-85-7
# 《爱国者游戏》                汤姆·克兰西      978-0-39-913241-4
# 《银河系漫游指南》              道格拉斯·亚当斯 978-0-33-025864-7
```

我们现在可以将 XML 写入磁盘、屏幕、网页或其他任何地方！

利润

虽然这可能不是适合所有人的方法，但我发现它有助于避免一大堆问题 `[void]$xml.SelectSingleNode("/complicated/xpath/goes[here]").AppendChild($xml.CreateElement("newElementName")` 随后是 `$xml.SelectSingleNode("/complicated/xpath/goes/here/newElementName") = $textValue`

我认为示例中详细介绍的方法更简洁，也更容易被普通人理解。

改进

```
        # Create the new child publisher element
        # Note the use of "SelectSingleNode" here, this allows the use of the "AppendChild" method
as it returns
        # a XmlElement type object instead of the $Null data that is currently stored in that leaf
of the
        # XML document tree

[void]$newElement.SelectSingleNode("publishers").AppendChild($xml.ImportNode($t_publisher.publisher
, $true));

        # Update the attribute and text values of our new XML Element to match our new data
        $newPublisherElement = $newElement.SelectSingleNode("publishers/publisher[last()]");
        $newPublisherElement.year = [String]$publisher.Year;
        $newPublisherElement.name = [String]$publisher.Name;
        $newPublisherElement.binding = [String]$publisher.Binding;
        $newPublisherElement.isbn = [String]$publisher.ISBN;
        If ( $publisher.first ) {
            $newPublisherElement.first = "True";
        }
    }

    ForEach ( $character in $book.Characters ) {
        # Select the characters xml element
        $charactersElement = $newElement.SelectSingleNode("characters");

        # Add a new character child element
        [void]$charactersElement.AppendChild($xml.ImportNode($t_character.character, $true));

        # Select the new characters/character element
        $characterElement = $charactersElement.SelectSingleNode("character[last()]");

        # Update the attribute and text values to match our new data
        $characterElement.name = [String]$character;
    }
}

# Check out the new XML:
$xml.books.book | Select Title, Author, @{N="ISBN";E={If ( $_.Publishers.Publisher.Count ) {
$_.Publishers.publisher[0].ISBN} Else { $_.Publishers.publisher.isbn}}};

# Outputs:
# title                                author         ISBN
# -----                                ------         ----
# Of Mice And Men                      John Steinbeck 978-88-58702-15-4
# The Hunt for Red October             Tom Clancy     978-08-70212-85-7
# Patriot Games                        Tom Clancy     978-0-39-913241-4
# The Hitchhiker's Guide to the Galaxy Douglas Adams  978-0-33-025864-7
```

We can now write our XML to disk, or screen, or web, or wherever!

Profit

While this may not be the procedure for everyone I found it to help avoid a whole bunch of `[void]$xml.SelectSingleNode("/complicated/xpath/goes[here]").AppendChild($xml.CreateElement("newElementName")` followed by `$xml.SelectSingleNode("/complicated/xpath/goes/here/newElementName") = $textValue`

I think the method detailed in the example is cleaner and easier to parse for normal humans.

Improvements

可能可以更改模板以包含带有子元素的元素，而不是将每个部分拆分为单独的模板。只需在遍历列表时注意克隆前一个元素即可。

It may be possible to change the template to include elements with children instead of breaking out each section as a separate template. You just have to take care to clone the previous element when you loop through the list.

第30章：与RESTful API通信

REST代表表现层状态转移（有时拼写为“ReST”）。它依赖于无状态的客户端-服务器、可缓存的通信协议，主要使用HTTP协议。它主要用于构建轻量级、易维护且可扩展的Web服务。基于REST的服务称为RESTful服务，使用的API称为RESTful API。在PowerShell中，Invoke-RestMethod用于处理它们。

第30.1节：向hipChat发送消息

```
$params = @{
Uri = "https://your.hipchat.com/v2/room/934419/notification?auth_token=???"
    Method = "POST"
Body = @{
color = 'yellow'
        message = "这是一条测试消息！"
       notify = $false
message_format = "text"
    } | ConvertTo-Json
ContentType = 'application/json'
}

Invoke-RestMethod @params
```

第30.2节：使用REST和PowerShell对象进行GET和POST多项操作

获取REST数据并存储到PowerShell对象中：

```
$Users = Invoke-RestMethod -Uri "http://jsonplaceholder.typicode.com/users"
```

修改数据中的多个项目：

```
$Users[0].name = "约翰·史密斯"
$Users[0].email = "John.Smith@example.com"
$Users[1].name = "简·史密斯"
$Users[1].email = "Jane.Smith@example.com"
```

将所有REST数据POST回去：

```
$Json = $Users | ConvertTo-Json
Invoke-RestMethod -Method Post -Uri "http://jsonplaceholder.typicode.com/users" -Body $Json -
ContentType 'application/json'
```

第30.3节：使用Slack.com传入Webhook

定义要发送的负载以支持更复杂的数据

```
$Payload = @{ text="测试字符串"; username="testuser" }
```

使用ConvertTo-Json命令和Invoke-RestMethod执行调用

```
Invoke-RestMethod -Uri "https://hooks.slack.com/services/yourwebhookstring" -Method Post -Body
```

Chapter 30: Communicating with RESTful APIs

REST stands for Representational State Transfer (sometimes spelled "ReST"). It relies on a stateless, client-server, cacheable communications protocol and mostly HTTP protocol is used. It is primarily used to build Web services that are lightweight, maintainable, and scalable. A service based on REST is called a RESTful service and the APIs which are being used for it are RESTful APIs. In PowerShell, *Invoke-RestMethod* is used to deal with them.

Section 30.1: Post Message to hipChat

```
$params = @{
    Uri = "https://your.hipchat.com/v2/room/934419/notification?auth_token=???"
    Method = "POST"
    Body = @{
        color = 'yellow'
        message = "This is a test message!"
        notify = $false
        message_format = "text"
    } | ConvertTo-Json
    ContentType = 'application/json'
}

Invoke-RestMethod @params
```

Section 30.2: Using REST with PowerShell Objects to GET and POST many items

GET your REST data and store in a PowerShell object:

```
$Users = Invoke-RestMethod -Uri "http://jsonplaceholder.typicode.com/users"
```

Modify many items in your data:

```
$Users[0].name = "John Smith"
$Users[0].email = "John.Smith@example.com"
$Users[1].name = "Jane Smith"
$Users[1].email = "Jane.Smith@example.com"
```

POST all of the REST data back:

```
$Json = $Users | ConvertTo-Json
Invoke-RestMethod -Method Post -Uri "http://jsonplaceholder.typicode.com/users" -Body $Json -
ContentType 'application/json'
```

Section 30.3: Use Slack.com Incoming Webhooks

Define your payload to send for possible more complex data

```
$Payload = @{ text="test string"; username="testuser" }
```

Use ConvertTo-Json cmdlet and Invoke-RestMethod to execute the call

```
Invoke-RestMethod -Uri "https://hooks.slack.com/services/yourwebhookstring" -Method Post -Body
```

```
(ConvertTo-Json $Payload)
```

第30.4节：使用REST和PowerShell对象获取和更新单个数据

获取REST数据并存储到PowerShell对象中：

```
$Post = Invoke-RestMethod -Uri "http://jsonplaceholder.typicode.com/posts/1"
```

修改你的数据：

```
$Post.title = "新标题"
```

将REST数据PUT回去

```
$Json = $Post | ConvertTo-Json
Invoke-RestMethod -Method Put -Uri "http://jsonplaceholder.typicode.com/posts/1" -Body $Json -
ContentType 'application/json'
```

第30.5节：使用REST和PowerShell删除项目

确定要删除的项目并删除它：

```
Invoke-RestMethod -Method Delete -Uri "http://jsonplaceholder.typicode.com/posts/1"
```

```
(ConvertTo-Json $Payload)
```

Section 30.4: Using REST with PowerShell Objects to Get and Put individual data

GET your REST data and store in a PowerShell object:

```
$Post = Invoke-RestMethod -Uri "http://jsonplaceholder.typicode.com/posts/1"
```

Modify your data:

```
$Post.title = "New Title"
```

PUT the REST data back

```
$Json = $Post | ConvertTo-Json
Invoke-RestMethod -Method Put -Uri "http://jsonplaceholder.typicode.com/posts/1" -Body $Json -
ContentType 'application/json'
```

Section 30.5: Using REST with PowerShell to Delete items

Identify the item that is to be deleted and delete it:

```
Invoke-RestMethod -Method Delete -Uri "http://jsonplaceholder.typicode.com/posts/1"
```

第31章：PowerShell SQL查询

项目	描述
$ServerInstance	这里需要指定数据库所在的实例
$Database	这里需要指定表所在的数据库
$Query	这里需要指定你想在SQL中执行的查询
$Username 和 $Password	数据库中有访问权限的用户名和密码

通过阅读本文档，您可以了解如何在 PowerShell 中使用 SQL 查询

第31.1节：SQL示例

要查询表MachineName中的所有数据，我们可以使用如下命令。

$Query="Select * from MachineName"

$Inst="ServerInstance"

$DbName="DatabaseName

$UID="User ID"

$Password="Password"

```
Invoke-Sqlcmd2 -Serverinstance $Inst -Database $DBName -query $Query -Username $UID -Password
$Password
```

第31.2节：SQL查询

要查询表MachineName中的所有数据，我们可以使用如下命令。

$Query="Select * from MachineName"

$Inst="ServerInstance"

$DbName="DatabaseName

$UID="User ID"

$Password="Password"

```
Invoke-Sqlcmd2 -Serverinstance $Inst -Database $DBName -query $Query -Username $UID -Password
$Password
```

Chapter 31: PowerShell SQL queries

Item	Description
$ServerInstance	Here we have to mention the instance in which the database is present
$Database	Here we have to mention the database in which the table is present
$Query	Here we have to the query which you we want to execute in SQ
$Username & $Password	UserName and Password which have access in database

By going through this document You can get to know how to use SQL queries with PowerShell

Section 31.1: SQLExample

For querying all the data from table *MachineName* we can use the command like below one.

$Query="Select * from MachineName"

$Inst="ServerInstance"

$DbName="DatabaseName

$UID="User ID"

$Password="Password"

```
Invoke-Sqlcmd2 -Serverinstance $Inst -Database $DBName -query $Query -Username $UID -Password
$Password
```

Section 31.2: SQLQuery

For querying all the data from table *MachineName* we can use the command like below one.

$Query="Select * from MachineName"

$Inst="ServerInstance"

$DbName="DatabaseName

$UID="User ID"

$Password="Password"

```
Invoke-Sqlcmd2 -Serverinstance $Inst -Database $DBName -query $Query -Username $UID -Password
$Password
```

第32章：正则表达式

第32.1节：单次匹配

您可以使用正则表达式快速判断文本是否包含特定模式。在PowerShell中有多种方法可以使用正则表达式。

```
#示例文本
$text = @"
这是 (一个) 示例
文本，这是
a (示例文本)
"@

#示例模式：括号内的内容
$pattern = '\(.*?\)'
```

使用 -Match 运算符

要使用内置的 -match 运算符判断字符串是否匹配某个模式，语法为 '输入' -match'模式'。根据搜索结果，这将返回 true 或 false。如果匹配成功，可以通过访问 $Matches 变量查看匹配项和分组（如果模式中定义了分组）。

```
> $text -match $pattern
True

> $Matches

Name Value
---- -----
0    (a)
```

你也可以使用 -match 来过滤字符串数组，只返回包含匹配项的字符串。

```
> $textarray = @"
这是 (一个) 示例
文本，这是
a （示例文本）
"@ -split "`n"

> $textarray -match $pattern
这是 （一个）示例
a （示例文本）
```

版本 ≥ 2.0

使用 Select-String

PowerShell 2.0 引入了一个用于使用正则表达式搜索文本的新 cmdlet。它为每个包含匹配项的 textinput 返回一个 MatchInfo 对象。你可以访问它的属性来查找匹配的分组等。

```
> $m = Select-String -InputObject $text -Pattern $pattern

> $m

这是 (a) 示例
文本， 这是
a (示例文本)

> $m | Format-List *
```

Chapter 32: Regular Expressions

Section 32.1: Single match

You can quickly determine if a text includes a specific pattern using Regex. There are multiple ways to work with Regex in PowerShell.

```
#Sample text
$text = @"
This is (a) sample
text, this is
a (sample text)
"@

#Sample pattern: Content wrapped in ()
$pattern = '\(.*?\)'
```

Using the -Match operator

To determine if a string matches a pattern using the built-in -matches operator, use the syntax 'input' -match 'pattern'. This will return true or false depending on the result of the search. If there was match you can view the match and groups (if defined in pattern) by accessing the $Matches-variable.

```
> $text -match $pattern
True

> $Matches

Name Value
---- -----
0    (a)
```

You can also use -match to filter through an array of strings and only return the strings containing a match.

```
> $textarray = @"
This is (a) sample
text, this is
a (sample text)
"@ -split "`n"

> $textarray -match $pattern
This is (a) sample
a (sample text)
```

Version ≥ 2.0

Using Select-String

PowerShell 2.0 introduced a new cmdlet for searching through text using regex. It returns a MatchInfo object per textinput that contains a match. You can access it's properties to find matching groups etc.

```
> $m = Select-String -InputObject $text -Pattern $pattern

> $m

This is (a) sample
text, this is
a (sample text)

> $m | Format-List *
```

```
IgnoreCase : True
LineNumber : 1
Line       : 这是 (a) 示例
              文本, 这是
              a (示例文本)
Filename   : InputStream
Path       : InputStream
Pattern    : \(.*?\)
Context    :
Matches    : {(a)}
```

像-match一样，Select-String也可以通过将数组传递给它来过滤字符串数组。它为每个包含匹配项的字符串创建一个MatchInfo对象。

```
> $textarray | Select-String -Pattern $pattern

这是一个示例
a (示例文本)

#你也可以访问匹配项、分组等。
> $textarray | Select-String -Pattern $pattern | fl *


IgnoreCase : True
LineNumber : 1
行号       : 这是一个示例
文件名     : InputStream
路径       : InputStream
模式       : \(.*?\)
上下文     :
匹配项     : {(a)}

忽略大小写 : True
行号       : 3
行内容     : a (示例文本)
文件名     : InputStream
路径       : InputStream
模式       : \(.*?\)
上下文     :
匹配项     : {(示例文本)}
```

Select-String也可以通过添加-SimpleMatch开关使用普通文本模式（非正则表达式）进行搜索。

使用 [RegEx]::Match()

您也可以使用.NET的[RegEx]类中提供的静态Match()方法。

```
> [regex]::Match($text,$pattern)

Groups   : {(a)}
成功   : True
捕获   : {(a)}
索引     : 8
长度     : 3
值       : (a)

> [regex]::Match($text,$pattern) | Select-Object -ExpandProperty Value
(a)
```

```
IgnoreCase : True
LineNumber : 1
Line       : This is (a) sample
             text, this is
             a (sample text)
Filename   : InputStream
Path       : InputStream
Pattern    : \(.*?\)
Context    :
Matches    : {(a)}
```

Like -match, Select-String can also be used to filter through an array of strings by piping an array to it. It creates a MatchInfo-object per string that includes a match.

```
> $textarray | Select-String -Pattern $pattern

This is (a) sample
a (sample text)

#You can also access the matches, groups etc.
> $textarray | Select-String -Pattern $pattern | fl *


IgnoreCase : True
LineNumber : 1
Line       : This is (a) sample
Filename   : InputStream
Path       : InputStream
Pattern    : \(.*?\)
Context    :
Matches    : {(a)}

IgnoreCase : True
LineNumber : 3
Line       : a (sample text)
Filename   : InputStream
Path       : InputStream
Pattern    : \(.*?\)
Context    :
Matches    : {(sample text)}
```

Select-String can also search using a normal text-pattern (no regex) by adding the -SimpleMatch switch.

Using [RegEx]::Match()

You can also use the static Match() method available in the .NET [RegEx]-class.

```
> [regex]::Match($text,$pattern)

Groups   : {(a)}
Success  : True
Captures : {(a)}
Index    : 8
Length   : 3
Value    : (a)

> [regex]::Match($text,$pattern) | Select-Object -ExpandProperty Value
(a)
```

第32.2节：替换

正则表达式的一个常见任务是将匹配模式的文本替换为新值。

```
#示例文本
$text = @"
这是 (一个) 示例
文本，这是
a (示例文本)
"@

#示例模式：用()包裹的文本
$pattern = '\(.*?\)'

#替换匹配项为：
$newvalue = 'test'
```

使用 -Replace 操作符

PowerShell 中的 -replace 操作符可用于将匹配模式的文本替换为新值，语法为 'input' -replace 'pattern', 'newvalue'。

```
> $text -replace $pattern, $newvalue
这是测试示例
文本，这是
a 测试
```

使用 [RegEx]::Replace() 方法

也可以使用 [RegEx] .NET 类中的 Replace() 方法来替换匹配项。

```
[regex]::Replace($text, $pattern, 'test')
这是测试示例
文本，这是
a 测试
```

第32.3节：使用MatchEvalutor动态替换文本

有时你需要将匹配某个模式的值替换为基于该特定匹配的新值，
这使得无法预测新值。对于这类情况，MatchEvaluator 会非常有用。

在 PowerShell 中，MatchEvaluator 就是一个带有单个参数的脚本块，该参数包含当前匹配的Match对象。该操作的输出将作为该特定匹配的新值。MatchEvaluator 可以与[Regex]::Replace()静态方法一起使用。

示例：将()内的文本替换为其长度

```
#示例文本
$text = @"
这是 (一个) 示例
文本，这是
a (示例文本)
"@

#示例模式：括号内的内容
$pattern = '(?<=\().*?(?=\))'

$MatchEvalutor = {
```

Section 32.2: Replace

A common task for regex is to replace text that matches a pattern with a new value.

```
#Sample text
$text = @"
This is (a) sample
text, this is
a (sample text)
"@

#Sample pattern: Text wrapped in ()
$pattern = '\(.*?\)'

#Replace matches with:
$newvalue = 'test'
```

Using -Replace operator

The -replace operator in PowerShell can be used to replace text matching a pattern with a new value using the syntax 'input' -replace 'pattern', 'newvalue'.

```
> $text -replace $pattern, $newvalue
This is test sample
text, this is
a test
```

Using [RegEx]::Replace() method

Replacing matches can also be done using the Replace() method in the [RegEx] .NET class.

```
[regex]::Replace($text, $pattern, 'test')
This is test sample
text, this is
a test
```

Section 32.3: Replace text with dynamic value using a MatchEvalutor

Sometimes you need to replace a value matching a pattern with a new value that's based on that specific match, making it impossible to predict the new value. For these types of scenarios, a MatchEvaluator can be very useful.

In PowerShell, a MatchEvaluator is as simple as a scriptblock with a single parameter that contains a Match-object for the current match. The output of the action will be the new value for that specific match. MatchEvalutor can be used with the [Regex]::Replace() static method.

Example: Replacing the text inside () with its length

```
#Sample text
$text = @"
This is (a) sample
text, this is
a (sample text)
"@

#Sample pattern: Content wrapped in ()
$pattern = '(?<=\().*?(?=\))'

$MatchEvalutor = {
    param($match)

    #用内容的长度替换内容
    $match.Value.Length

}
```

输出：

```
> [regex]::Replace($text, $pattern, $MatchEvaluator)

这是 1 个示例
文本, 这是
a 11
```

示例： 将 sample 转为大写

```
#示例模式：“Sample”
$pattern = 'sample'

$MatchEvalutor = {
    param($match)

    #将匹配项转换为大写
    $match.Value.ToUpper()

}
```

输出：

```
> [regex]::Replace($text, $pattern, $MatchEvalutor)

这是一个示例(SAMPLE)
文本, 这是
a (SAMPLE 文本)
```

第32.4节：转义特殊字符

正则表达式模式使用许多特殊字符来描述模式。例如，. 表示“任意字符”，+ 表示“一个或多个”等。

要在模式中使用这些字符，如 .、+ 等，需要对它们进行转义以去除其特殊含义。这是通过使用转义字符反斜杠 \ 来完成的。例如：要搜索 +，您需要使用模式 \+。

记住正则表达式中的所有特殊字符可能很困难，因此要转义您想搜索的字符串中的每个特殊字符，可以使用 [RegEx]::Escape("input") 方法。

```
> [regex]::Escape("(foo)")
\(foo\)

> [regex]::Escape("1+1.2=2.2")
1+1\.2=2\.2
```

第32.5节：多重匹配

有多种方法可以在文本中找到模式的所有匹配项。

```
    param($match)

    #Replace content with length of content
    $match.Value.Length

}
```

Output:

```
> [regex]::Replace($text, $pattern, $MatchEvalutor)

This is 1 sample
text, this is
a 11
```

Example: Make `sample` upper-case

```
#Sample pattern: "Sample"
$pattern = 'sample'

$MatchEvalutor = {
    param($match)

    #Return match in upper-case
    $match.Value.ToUpper()

}
```

Output:

```
> [regex]::Replace($text, $pattern, $MatchEvalutor)

This is (a) SAMPLE
text, this is
a (SAMPLE text)
```

Section 32.4: Escape special characters

A regex-pattern uses many special characters to describe a pattern. Ex., . means "any character", + is "one or more" etc.

To use these characters, as a .,+ etc., in a pattern, you need to escape them to remove their special meaning. This is done by using the escape character which is a backslash \ in regex. Example: To search for +, you would use the pattern \+.

It can be hard to remember all special characters in regex, so to escape every special character in a string you want to search for, you could use the `[RegEx]::Escape("input")` method.

```
> [regex]::Escape("(foo)")
\(foo\)

> [regex]::Escape("1+1.2=2.2")
1\+1\.2=2\.2
```

Section 32.5: Multiple matches

There are multiple ways to find all matches for a pattern in a text.

```
#示例文本
$text = @"
这是 (一个) 示例
文本，这是
a (示例文本)
"@

#示例模式：括号内的内容
$pattern = '\(.*?\)'
```

使用 Select-String

您可以通过向Select-String添加-AllMatches开关来查找所有匹配项（全局匹配）。

```
> $m = Select-String -InputObject $text -Pattern $pattern -AllMatches

> $m | Format-List *

IgnoreCase : True
LineNumber : 1
Line       : 这是 (a) 示例
             文本, 这是
             a (示例文本)
Filename   : InputStream
Path       : InputStream
Pattern    : \(.*?\)
Context    :
匹配项     : {(a), (示例文本)}

#列出所有匹配项
> $m.Matches

组   : {(a)}
成功  : True
捕获 : {(a)}
索引    : 8
长度    : 3
值    : (a)

组   : {(示例文本)}
成功  : True
捕获 : {(示例文本)}
索引    : 37
长度   : 13
值    : (sample text)

#获取匹配文本
> $m.Matches | Select-Object -ExpandProperty Value
(a)
(sample text)
```

使用 [RegEx]::Matches()

.NET `[regex]` 类中的 Matches() 方法也可以用于全局搜索多个匹配项。

```
> [regex]::Matches($text,$pattern)

组   : {(a)}
成功  : True
捕获 : {(a)}
索引    : 8
长度   : 3
```

```
#Sample text
$text = @"
This is (a) sample
text, this is
a (sample text)
"@

#Sample pattern: Content wrapped in ()
$pattern = '\(.*?\)'
```

Using Select-String

You can find all matches (global match) by adding the -AllMatches switch to Select-String.

```
> $m = Select-String -InputObject $text -Pattern $pattern -AllMatches

> $m | Format-List *

IgnoreCase : True
LineNumber : 1
Line       : This is (a) sample
             text, this is
             a (sample text)
Filename   : InputStream
Path       : InputStream
Pattern    : \(.*?\)
Context    :
Matches    : {(a), (sample text)}

#List all matches
> $m.Matches

Groups   : {(a)}
Success  : True
Captures : {(a)}
Index    : 8
Length   : 3
Value    : (a)

Groups   : {(sample text)}
Success  : True
Captures : {(sample text)}
Index    : 37
Length   : 13
Value    : (sample text)

#Get matched text
> $m.Matches | Select-Object -ExpandProperty Value
(a)
(sample text)
```

Using [RegEx]::Matches()

The Matches() method in the .NET `[regex]-class can also be used to do a global search for multiple matches.

```
> [regex]::Matches($text,$pattern)

Groups   : {(a)}
Success  : True
Captures : {(a)}
Index    : 8
Length   : 3
```

```
值    : (a)

组   : {(示例文本)}
成功  : True
捕获 : {(示例文本)}
索引     : 37
长度    : 13
值    : (sample text)

> [regex]::Matches($text,$pattern) | Select-Object -ExpandProperty Value

(a)
(sample text)
```

```
Value    : (a)

Groups   : {(sample text)}
Success  : True
Captures : {(sample text)}
Index    : 37
Length   : 13
Value    : (sample text)

> [regex]::Matches($text,$pattern) | Select-Object -ExpandProperty Value

(a)
(sample text)
```

第33章：别名

第33.1节：Get-Alias

列出所有别名及其对应的功能：

```
Get-Alias
```

获取特定cmdlet的所有别名：

```
PS C:\> get-alias -Definition Get-ChildItem

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Alias           dir -> Get-ChildItem
Alias           gci -> Get-ChildItem
Alias           ls -> Get-ChildItem
```

通过匹配查找别名：

```
PS C:\> get-alias -Name p*

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Alias           popd -> Pop-Location
别名         proc -> Get-Process
别名         ps -> Get-Process
别名         pushd -> Push-Location
别名         pwd -> Get-Location
```

第33.2节：Set-Alias

此命令允许您为现有命令创建新的别名

```
PS C:\> Set-Alias -Name proc -Value Get-Process
PS C:\> proc

句柄  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id  SI 进程名
-------  ------    -----      ----- -----   ------     --  -- -----------
    292      17    13052      20444 ...19     7.94    620   1 ApplicationFrameHost
....
```

请记住，您创建的任何别名只会在当前会话中保留。当您启动新会话时，需要重新创建别名。PowerShell 配置文件（参见[主题尚未创建]）非常适合此用途。

Chapter 33: Aliases

Section 33.1: Get-Alias

To list all aliases and their functions:

```
Get-Alias
```

To get all aliases for specific cmdlet:

```
PS C:\> get-alias -Definition Get-ChildItem

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Alias           dir -> Get-ChildItem
Alias           gci -> Get-ChildItem
Alias           ls -> Get-ChildItem
```

To find aliases by matching:

```
PS C:\> get-alias -Name p*

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Alias           popd -> Pop-Location
Alias           proc -> Get-Process
Alias           ps -> Get-Process
Alias           pushd -> Push-Location
Alias           pwd -> Get-Location
```

Section 33.2: Set-Alias

This cmdlet allows you to create new alternate names for exiting cmdlets

```
PS C:\> Set-Alias -Name proc -Value Get-Process
PS C:\> proc

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id  SI ProcessName
-------  ------    -----      ----- -----   ------     --  -- -----------
    292      17    13052      20444 ...19     7.94    620   1 ApplicationFrameHost
....
```

Keep in mind that any alias you create will be persisted only in current session. When you start new session you need to create your aliases again. Powershell Profiles (see [topic not yet created]) are great for these purposes.

第34章：使用进度条

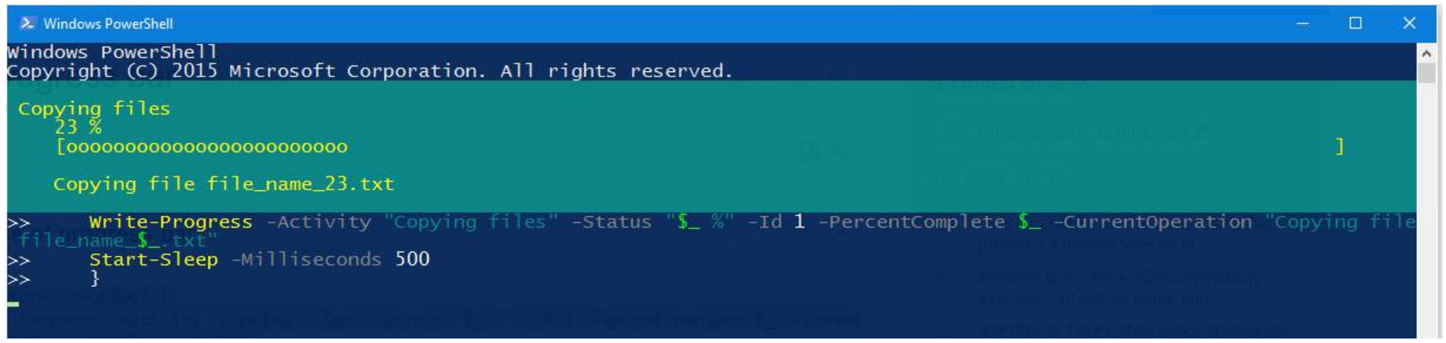
进度条可用于显示某个过程正在进行中。这是一个节省时间且流畅的功能，值得拥有。进度条在调试时非常有用，可以帮助确定脚本的哪个部分正在执行，并且对运行脚本的人来说，跟踪进度是令人满意的。当脚本执行时间较长时，通常会显示某种进度。当用户启动脚本而没有任何反应时，人们会开始怀疑脚本是否正确启动。

第34.1节：进度条的简单使用

```
1..100 | ForEach-Object {
        Write-Progress -Activity "复制文件" -Status "$_ %" -Id 1 -PercentComplete $_ -CurrentOperation "正在复制文件 file_name_$_.txt"Start-Sleep -Milliseconds 500
        # sleep 模拟执行代码，替换此行以执行您的代码（例如文件复制）

    }
```

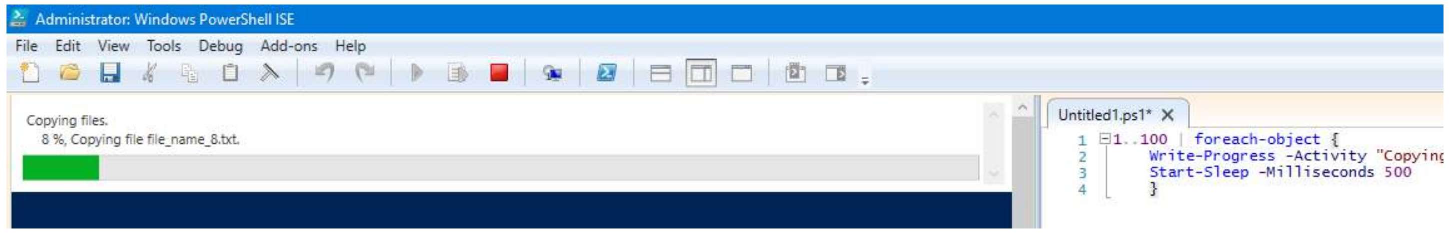
请注意，为简洁起见，此示例不包含任何执行代码（用Start-Sleep模拟）。不过可以直接运行此代码，然后进行修改和尝试。

这是在 PowerShell 控制台中结果的显示效果：



```
Windows PowerShell
Copyright (C) 2015 Microsoft Corporation. All rights reserved.

 Copying files
    23 %
    [oooooooooooooooooooooooo                                                          ]

    Copying file file_name_23.txt

>>      Write-Progress -Activity "Copying files" -Status "$_ %" -Id 1 -PercentComplete $_ -CurrentOperation "Copying file
 file_name_$_.txt"
>>      Start-Sleep -Milliseconds 500
>>      }
```

这是在 PowerShell ISE 中结果的显示效果：



```
Copying files.
  8 %, Copying file file_name_8.txt.
```

```
1..100 | foreach-object {
    Write-Progress -Activity "Copying
    Start-Sleep -Milliseconds 500
    }
```

第34.2节：内部进度条的使用

```
1..10 | foreach-object {
        $fileName = "file_name_$_.txt"
        Write-Progress -Activity "复制文件" -Status "$($_*10) %" -Id 1 -PercentComplete
($_*10) -CurrentOperation "正在复制文件 $fileName"

        1..100 | foreach-object {
            Write-Progress -Activity "正在复制文件 $fileName 的内容" -Status "$_ %" -Id 2
-ParentId 1 -PercentComplete $_ -CurrentOperation "正在复制第 $_ 行"

            Start-Sleep -Milliseconds 20 # sleep 模拟工作代码，请用您的执行代码替换此行（例如文件复制）

        }

        Start-Sleep -Milliseconds 500 # sleep 模拟工作代码，请用您的
```

Chapter 34: Using the progress bar

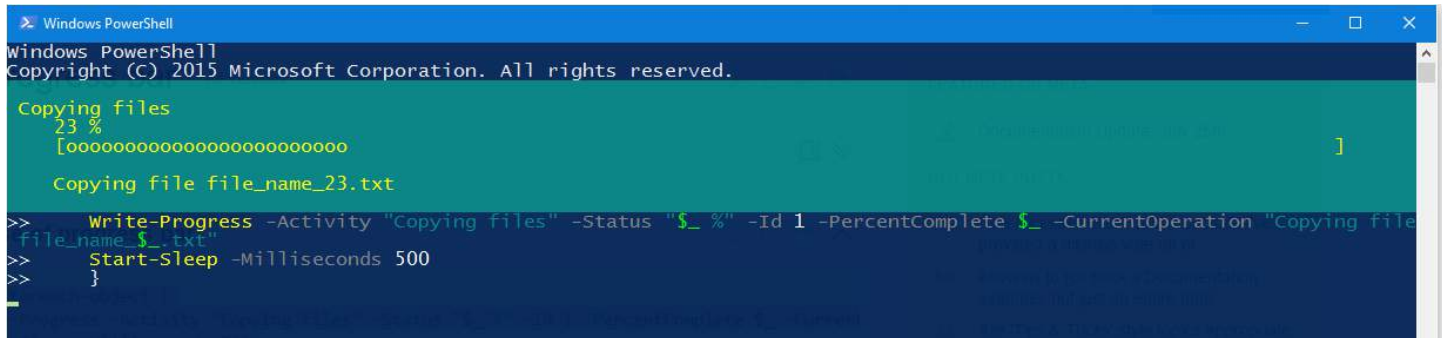
A progress bar can be used to show something is in a process. It is a time-saving and slick feature one should have. Progress bars are incredibly useful while debugging to figure out which part of the script is executing, and they're satisfying for the people running scripts to track what's happening. It is common to display some kind of progress when a script takes a long time to complete. When a user launches the script and nothing happens, one begins to wonder if the script launched correctly.

Section 34.1: Simple use of progress bar

```
1..100 | ForEach-Object {
        Write-Progress -Activity "Copying files" -Status "$_ %" -Id 1 -PercentComplete $_ -
CurrentOperation "Copying file file_name_$_.txt"
        Start-Sleep -Milliseconds 500    # sleep simulates working code, replace this line with
your executive code (i.e. file copying)
    }
```

Please note that for brevity this example does not contain any executive code (simulated with `Start-Sleep`). However it is possible to run it directly as is and then modify and play with it.

This is how result looks in PS console:



Windows PowerShell

```
Windows PowerShell
Copyright (C) 2015 Microsoft Corporation. All rights reserved.

 Copying files
    23 %
    [oooooooooooooooooooooooo                                                          ]

    Copying file file_name_23.txt

>>      Write-Progress -Activity "Copying files" -Status "$_ %" -Id 1 -PercentComplete $_ -CurrentOperation "Copying file
 file_name_$_.txt"
>>      Start-Sleep -Milliseconds 500
>>      }
```

This is how result looks in PS ISE:

Administrator: Windows PowerShell ISE

```
Copying files.
  8 %, Copying file file_name_8.txt.
```

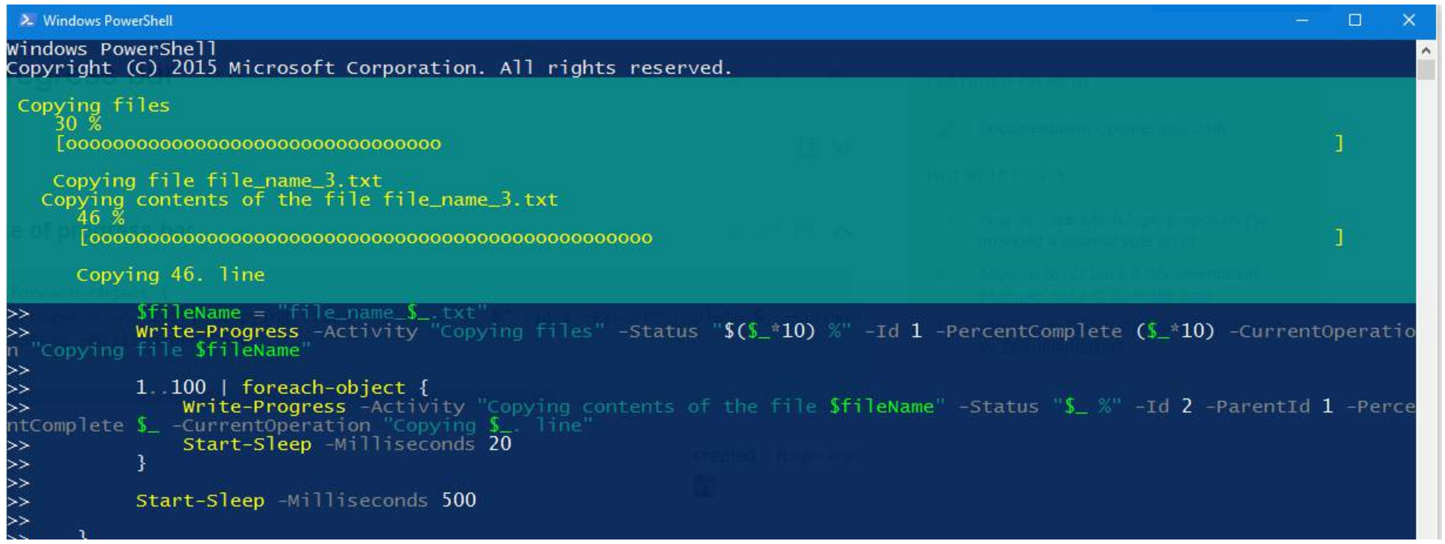
```
1..100 | foreach-object {
    Write-Progress -Activity "Copying
    Start-Sleep -Milliseconds 500
    }
```

Section 34.2: Usage of inner progress bar

```
1..10 | foreach-object {
        $fileName = "file_name_$_.txt"
        Write-Progress -Activity "Copying files" -Status "$($_*10) %" -Id 1 -PercentComplete
($_*10) -CurrentOperation "Copying file $fileName"

        1..100 | foreach-object {
            Write-Progress -Activity "Copying contents of the file $fileName" -Status "$_ %" -Id 2
-ParentId 1 -PercentComplete $_ -CurrentOperation "Copying $_. line"

            Start-Sleep -Milliseconds 20 # sleep simulates working code, replace this line with
your executive code (i.e. file copying)
        }

        Start-Sleep -Milliseconds 500 # sleep simulates working code, replace this line with your
执行代码（例如文件搜索）

    }
```

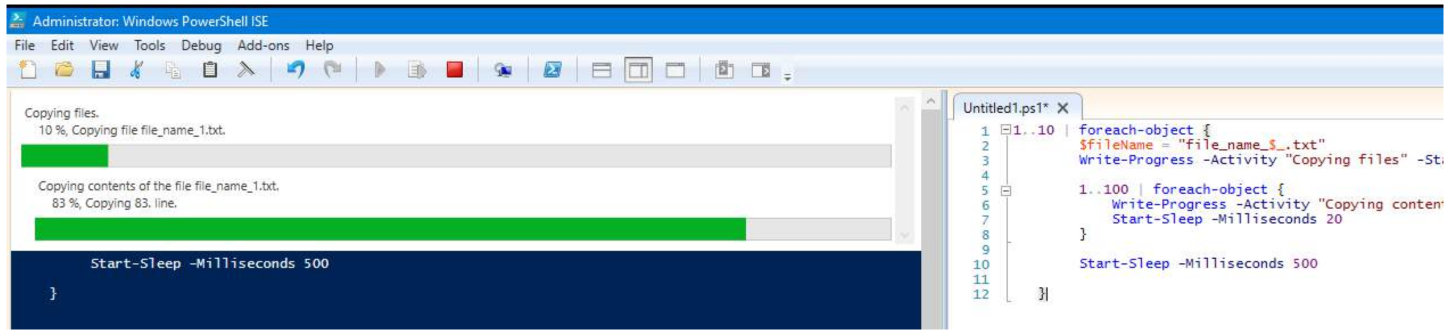
请注意，为简洁起见，此示例不包含任何执行代码（用Start-Sleep模拟）。不过可以直接运行此代码，然后进行修改和尝试。

这是在 PowerShell 控制台中结果的显示效果：



```
Windows PowerShell
Copyright (C) 2015 Microsoft Corporation. All rights reserved.

 Copying files
    30 %
    [oooooooooooooooooooooooooooooooo                                                                    ]

     Copying file file_name_3.txt
   Copying contents of the file file_name_3.txt
       46 %
      [oooooooooooooooooooooooooooooooooooooooooooooooo                                                  ]

       Copying 46. line
>>           $fileName = "file_name_$_.txt"
>>           Write-Progress -Activity "Copying files" -Status "$($_*10) %" -Id 1 -PercentComplete ($_*10) -CurrentOperatio
n "Copying file $fileName"
>>
>>           1..100 | foreach-object {
>>               Write-Progress -Activity "Copying contents of the file $fileName" -Status "$_ %" -Id 2 -ParentId 1 -Perce
ntComplete $_ -CurrentOperation "Copying $_. line"
>>               Start-Sleep -Milliseconds 20
>>           }
>>
>>           Start-Sleep -Milliseconds 500
>>
```

这是在 PowerShell ISE 中结果的显示效果：



Administrator: Windows PowerShell ISE
File Edit View Tools Debug Add-ons Help
Untitled1.ps1*

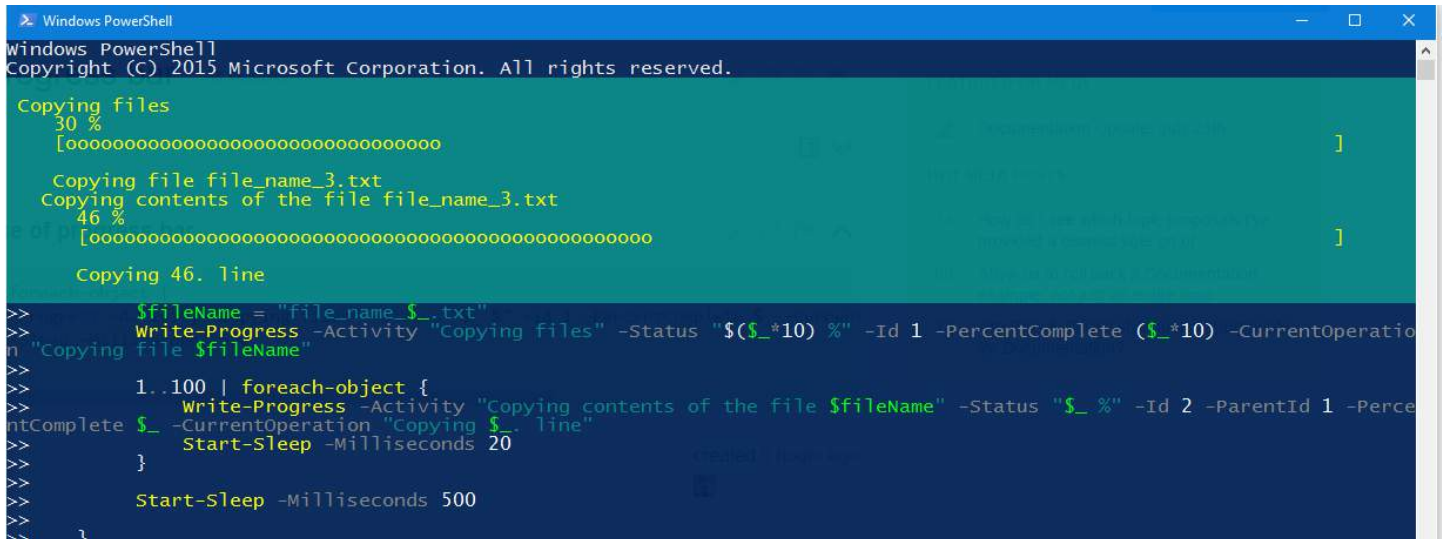
```
Copying files.
  10 %, Copying file file_name_1.txt.

  Copying contents of the file file_name_1.txt.
    83 %, Copying 83. line.

        Start-Sleep -Milliseconds 500

    }
```

```
1..10 | foreach-object {
        $fileName = "file_name_$_.txt"
        Write-Progress -Activity "Copying files" -St

        1..100 | foreach-object {
            Write-Progress -Activity "Copying conten
            Start-Sleep -Milliseconds 20
        }

        Start-Sleep -Milliseconds 500

    }
```

```
executive code (i.e. file search)

    }
```

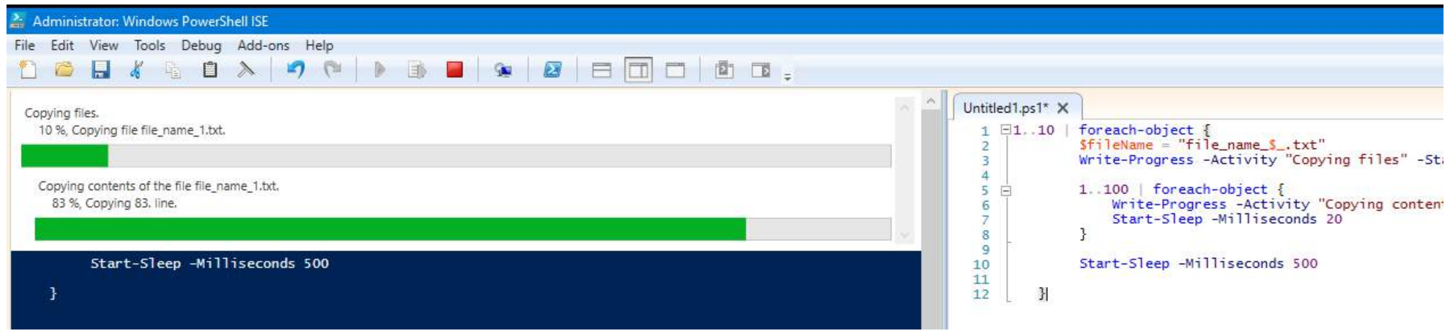
Please note that for brevity this example does not contain any executive code (simulated with `Start-Sleep`). However it is possible to run it directly as is and then modify and play with it.

This is how result looks in PS console:



```
Windows PowerShell
Copyright (C) 2015 Microsoft Corporation. All rights reserved.

 Copying files
    30 %
    [oooooooooooooooooooooooooooooooo                                                                    ]

     Copying file file_name_3.txt
   Copying contents of the file file_name_3.txt
       46 %
      [oooooooooooooooooooooooooooooooooooooooooooooooo                                                  ]

       Copying 46. line
>>           $fileName = "file_name_$_.txt"
>>           Write-Progress -Activity "Copying files" -Status "$($_*10) %" -Id 1 -PercentComplete ($_*10) -CurrentOperatio
n "Copying file $fileName"
>>
>>           1..100 | foreach-object {
>>               Write-Progress -Activity "Copying contents of the file $fileName" -Status "$_ %" -Id 2 -ParentId 1 -Perce
ntComplete $_ -CurrentOperation "Copying $_. line"
>>               Start-Sleep -Milliseconds 20
>>           }
>>
>>           Start-Sleep -Milliseconds 500
>>
```

This is how result looks in PS ISE:



Administrator: Windows PowerShell ISE
File Edit View Tools Debug Add-ons Help
Untitled1.ps1*

```
Copying files.
  10 %, Copying file file_name_1.txt.

  Copying contents of the file file_name_1.txt.
    83 %, Copying 83. line.

        Start-Sleep -Milliseconds 500

    }
```

```
1..10 | foreach-object {
        $fileName = "file_name_$_.txt"
        Write-Progress -Activity "Copying files" -St

        1..100 | foreach-object {
            Write-Progress -Activity "Copying conten
            Start-Sleep -Milliseconds 20
        }

        Start-Sleep -Milliseconds 500

    }
```

第35章：PowerShell.exe 命令行

参数	描述
-Help | -? | /?	显示帮助
-File <FilePath> [<Args>]	要执行的脚本文件路径及参数（可选）
-命令 { - | <脚本块> [-args <参数数组>] | <字符串> [<命令参数>] }	要执行的命令及其参数
-EncodedCommand <Base64EncodedCommand>	Base64 编码的命令
-ExecutionPolicy <ExecutionPolicy>	仅为此进程设置执行策略
-InputFormat { Text | XML}	设置发送到进程的数据输入格式。文本（字符串）或 XML（序列化的 CLIXML）
-Mta	PowerShell 3.0 及以上版本：以多线程单元（MTA）模式运行 PowerShell（默认是单线程单元 STA）
-Sta	PowerShell 2.0：在单线程单元（STA）中运行PowerShell（默认是多线程单元MTA）
-NoExit	执行脚本/命令后保持PowerShell控制台运行
-NoLogo	启动时隐藏版权横幅
-NonInteractive	对用户隐藏控制台
-NoProfile	避免加载计算机或用户的PowerShell配置文件
-OutputFormat { Text | XML }	设置 PowerShell 返回数据的输出格式。文本（字符串）或 XML（序列化的 CLIXML）
-PSConsoleFile <FilePath>	加载预先创建的控制台文件以配置环境（使用Export-Console创建）
-Version <Windows PowerShell version>	指定要运行的PowerShell版本。主要用于2.0
-WindowStyle <style>	指定是否以普通、隐藏、最小化或最大化窗口启动PowerShell进程。

第35.1节：执行命令

-Command 参数用于指定启动时要执行的命令。它支持多条数据输入。

-Command <string>

您可以指定启动时执行的命令字符串。多个以分号;分隔的语句可以被执行。

```
>PowerShell.exe -Command "(Get-Date).ToShortDateString()"
2016年9月10日

>PowerShell.exe -Command "(Get-Date).ToShortDateString(); 'PowerShell 很有趣！'"
2016年9月10日
PowerShell 很有趣!
```

-Command { 脚本块 }

-Command参数也支持脚本块输入（一个或多个用大括号{ #代码 }包裹的语句）。这仅在从另一个 Windows PowerShell 会话调用PowerShell.exe时有效。

```
PS > powershell.exe -Command {
"这有时很有用..."
(Get-Date).ToShortDateString()
```

Chapter 35: PowerShell.exe Command-Line

Parameter	Description
-Help | -? | /?	Shows the help
-File <FilePath> [<Args>]	Path to script-file that should be executed and arguments (optional)
-Command { - | <script-block> [-args <arg-array>] | <string> [<CommandParameters>] }	Commands to be executed followed by arguments
-EncodedCommand <Base64EncodedCommand>	Base64 encoded commands
-ExecutionPolicy <ExecutionPolicy>	Sets the execution policy for this process only
-InputFormat { Text | XML}	Sets input format for data sent to process. Text (strings) or XML (serialized CLIXML)
-Mta	PowerShell 3.0+: Runs PowerShell in multi-threaded apartment (STA is default)
-Sta	PowerShell 2.0: Runs PowerShell in a single-threaded apartment (MTA is default)
-NoExit	Leaves PowerShell console running after executing the script/command
-NoLogo	Hides copyright-banner at launch
-NonInteractive	Hides console from user
-NoProfile	Avoid loading of PowerShell profiles for machine or user
-OutputFormat { Text | XML }	Sets output format for data returned from PowerShell. Text (strings) or XML (serialized CLIXML)
-PSConsoleFile <FilePath>	Loads a pre-created console file that configures the environment (created using Export-Console)
-Version <Windows PowerShell version>	Specify a version of PowerShell to run. Mostly used with 2.0
-WindowStyle <style>	Specifies whether to start the PowerShell process as a normal, hidden, minimized or maximized window.

Section 35.1: Executing a command

The -Command parameter is used to specify commands to be executed on launch. It supports multiple data inputs.

-Command <string>

You can specify commands to executed on launch as a string. Multiple semicolon ;-separated statements may be executed.

```
>PowerShell.exe -Command "(Get-Date).ToShortDateString()"
10.09.2016

>PowerShell.exe -Command "(Get-Date).ToShortDateString(); 'PowerShell is fun!'"
10.09.2016
PowerShell is fun!
```

-Command { scriptblock }

The -Command parameter also supports a scriptblock input (one or multiple statements wrapped in braces { #code }. This only works when calling PowerShell.exe from another Windows PowerShell-session.

```
PS > powershell.exe -Command {
"This can be useful, sometimes..."
(Get-Date).ToShortDateString()
}
这有时很有用，有时候...
10.09.2016
```

-Command -（标准输入）

你可以通过使用-Command -从标准输入传入命令。标准输入可以来自 echo、读取文件、传统控制台应用程序等。

```
>echo "Hello World";"Greetings from PowerShell" | PowerShell.exe -NoProfile -Command -
Hello World
来自 PowerShell 的问候
```

第 35.2 节：执行脚本文件

您可以使用 -File 参数指定一个 ps1 脚本文件，在启动时执行其内容。

基础脚本

MyScript.ps1

```
(Get-Date).ToShortDateString()
"Hello World"
```

输出：

```
>PowerShell.exe -File Desktop\MyScript.ps1
2016年9月10日
Hello World
```

使用参数和参数值

您可以在文件路径后添加参数和/或参数值以在脚本中使用它们。参数值将作为未定义/可用脚本参数的值使用，其余的将保存在 $args 数组中

MyScript.ps1

```
param($Name)

"你好 $Name！今天的日期是 $((Get-Date).ToShortDateString())"
"第一个参数：$($args[0])"
```

输出：

```
>PowerShell.exe -File Desktop\MyScript.ps1 -Name StackOverflow foo
你好 StackOverflow！ 今天的日期是 2016年9月10日
第一个参数：foo
```

```
}
This can be useful, sometimes...
10.09.2016
```

-Command - (standard input)

You can pass in commands from the standard input by using -Command -. The standard input can come from echo, reading a file, a legacy console application etc.

```
>echo "Hello World";"Greetings from PowerShell" | PowerShell.exe -NoProfile -Command -
Hello World
Greetings from PowerShell
```

Section 35.2: Executing a script file

You can specify a file to a ps1-script to execute its content on launch using the -File parameter.

Basic script

MyScript.ps1

```
(Get-Date).ToShortDateString()
"Hello World"
```

Output:

```
>PowerShell.exe -File Desktop\MyScript.ps1
10.09.2016
Hello World
```

Using parameters and arguments

You can add parameters and/or arguments after filepath to use them in the script. Arguments will be used as values for undefined/available script-parameters, the rest will be available in the $args-array

MyScript.ps1

```
param($Name)

"Hello $Name! Today's date it $((Get-Date).ToShortDateString())"
"First arg: $($args[0])"
```

Output:

```
>PowerShell.exe -File Desktop\MyScript.ps1 -Name StackOverflow foo
Hello StackOverflow! Today's date it 10.09.2016
First arg: foo
```

第36章：Cmdlet命名

Cmdlet 应该使用 <动词>-<名词> 的命名方案，以提高可发现性。

第36.1节：动词

用于命名 Cmdlet 的动词应从 Get-Verb 提供的动词列表中选择

有关如何使用动词的更多详细信息，请参见 Windows PowerShell的批准动词

第36.2节：名词

名词应始终使用单数形式。

名词要保持一致。例如`Find-Package`需要一个提供者，名词是`PackageProvider`而不是`ProviderPackage`。

Chapter 36: Cmdlet Naming

CmdLets should be named using a **`<verb>-<noun>`** naming scheme in order to improve discoverability.

Section 36.1: Verbs

Verbs used to name CmdLets should be named from verbs from the list supplied be `Get-Verb`

Further details on how to use verbs can be found at Approved Verbs for Windows PowerShell

Section 36.2: Nouns

Nouns should always be singular.

Be consistent with the nouns. For instance `Find-Package` needs a provider the noun is `PackageProvider` not `ProviderPackage`.

第37章：运行可执行文件

第37.1节：图形用户界面应用程序

```
PS> gui_app.exe (1)
PS> & gui_app.exe (2)
PS> & gui_app.exe | Out-Null (3)
PS> Start-Process gui_app.exe (4)
PS> Start-Process gui_app.exe -Wait (5)
```

图形用户界面应用程序在不同的进程中启动，并会立即将控制权返回给PowerShell主机。有时你需要应用程序完成处理后，下一条PowerShell语句才能执行。这可以通过将应用程序输出管道传输到$null (3) 或使用带有 -Wait 参数的 Start-Process (5) 来实现。

第37.2节：控制台流

```
PS> $ErrorActionPreference = "Continue" (1)
PS> & console_app.exe *>&1 | % { $_ } (2)
PS> & console_app.exe *>&1 | ? { $_ -is [System.Management.Automation.ErrorRecord] } (3)
PS> & console_app.exe *>&1 | ? { $_ -is [System.Management.Automation.WarningRecord] } (4)
PS> & console_app.exe *>&1 | ? { $_ -is [System.Management.Automation.VerboseRecord] } (5)
PS> & console_app.exe *>&1 (6)
PS> & console_app.exe 2>&1 (7)
```

流 2 包含 System.Management.Automation.ErrorRecord 对象。请注意，某些应用程序如 git.exe 使用"错误流"作为信息用途，这些内容不一定是错误。在这种情况下，最好查看退出代码以确定是否应将错误流解释为错误。

PowerShell 理解这些流：输出、错误、警告、详细、调试、进度。原生应用程序通常只使用这些流：输出、错误、警告。

在 PowerShell 5 中，所有流都可以重定向到标准输出/成功流（6）。

在早期的 PowerShell 版本中，只有特定的流可以重定向到标准输出/成功流（7）。在此示例中，"错误流"将被重定向到输出流。

第 37.3 节：退出代码

```
PS> $LastExitCode
PS> $?
PS> $Error[0]
```

这些是内置的 PowerShell 变量，提供有关最近错误的附加信息。$LastExitCode 是最后执行的本地应用程序的最终退出代码。$? 和 $Error[0] 是 PowerShell 生成的最后一个错误记录。

Chapter 37: Running Executables

Section 37.1: GUI Applications

```
PS> gui_app.exe (1)
PS> & gui_app.exe (2)
PS> & gui_app.exe | Out-Null (3)
PS> Start-Process gui_app.exe (4)
PS> Start-Process gui_app.exe -Wait (5)
```

GUI applications launch in a different process, and will immediately return control to the PowerShell host. Sometimes you need the application to finish processing before the next PowerShell statement must be executed. This can be achieved by piping the application output to $null (3) or by using Start-Process with the -Wait switch (5).

Section 37.2: Console Streams

```
PS> $ErrorActionPreference = "Continue" (1)
PS> & console_app.exe *>&1 | % { $_ } (2)
PS> & console_app.exe *>&1 | ? { $_ -is [System.Management.Automation.ErrorRecord] } (3)
PS> & console_app.exe *>&1 | ? { $_ -is [System.Management.Automation.WarningRecord] } (4)
PS> & console_app.exe *>&1 | ? { $_ -is [System.Management.Automation.VerboseRecord] } (5)
PS> & console_app.exe *>&1 (6)
PS> & console_app.exe 2>&1 (7)
```

Stream 2 contains System.Management.Automation.ErrorRecord objects. Note that some applications like git.exe use the "error stream" for informational purposes, that are not necessarily errors at all. In this case it is best to look at the exit code to determine whether the error stream should be interpreted as errors.

PowerShell understands these streams: Output, Error, Warning, Verbose, Debug, Progress. Native applications commonly use only these streams: Output, Error, Warning.

In PowerShell 5, all streams can be redirected to the standard output/success stream (6).

In earlier PowerShell versions, only specific streams can be redirected to the standard output/success stream (7). In this example, the "error stream" will be redirected to the output stream.

Section 37.3: Exit Codes

```
PS> $LastExitCode
PS> $?
PS> $Error[0]
```

These are built-in PowerShell variables that provide additional information about the most recent error. $LastExitCode is the final exit code of the last native application that was executed. $? and $Error[0] is the last error record that was generated by PowerShell.

第38章：强制执行脚本先决条件

第38.1节：强制执行PowerShell主机的最低版本

```
#requires -version 4
```

在尝试使用较低版本运行此脚本后，您将看到此错误消息

> .\script.ps1 ：脚本 'script.ps1' 无法运行，因为它在第1行包含了针对Windows PowerShell版本5.0的“#requires”语句。脚本要求的版本与当前运行的Windows PowerShell版本2.0不匹配。

第38.2节：强制以管理员身份运行脚本

版本 ≥ 4.0

```
#requires -RunAsAdministrator
```

在尝试不具备管理员权限运行此脚本后，您将看到此错误消息

> .\script.ps1 ：脚本 'script.ps1' 无法运行，因为它包含了要求以管理员身份运行的“#requires”语句。
> 当前的Windows PowerShell会话未以管理员身份运行。请启动
> 使用“以管理员身份运行”选项启动 Windows PowerShell，然后尝试再次运行脚本。

Chapter 38: Enforcing script prerequisites

Section 38.1: Enforce minimum version of PowerShell host

```
#requires -version 4
```

After trying to run this script in lower version, you will see this error message

> .\script.ps1 : The script 'script.ps1' cannot be run because it contained a "#requires" statement at line 1 for Windows PowerShell version 5.0. The version required by the script does not match the currently running version of Windows PowerShell version 2.0.

Section 38.2: Enforce running the script as administrator

Version ≥ 4.0

```
#requires -RunAsAdministrator
```

After trying to run this script without admin privileges, you will see this error message

> .\script.ps1 : The script 'script.ps1' cannot be run because it contains a "#requires" statement for running as Administrator. The current Windows PowerShell session is not running as Administrator. Start Windows PowerShell by using the Run as Administrator option, and then try running the script again.

第39章：使用帮助系统

第39.1节：更新帮助系统

版本 > 3.0

从 PowerShell 3.0 开始，您可以使用单个 cmdlet 下载和更新离线帮助文档。

```
Update-Help
```

在多台计算机上（或未连接互联网的计算机）更新帮助。

在具有帮助文件的计算机上运行以下命令

```
Save-Help -DestinationPath \\Server01\Share\PSHelp -Credential $Cred
```

远程在多台计算机上运行

```
Invoke-Command -ComputerName (Get-Content Servers.txt) -ScriptBlock {Update-Help -SourcePath
\\Server01\Share\Help -Credential $cred}
```

第39.2节：使用 Get-Help

Get-Help 可用于查看 PowerShell 中的帮助。您可以搜索 cmdlet、函数、提供程序或其他主题。

要查看有关作业的帮助文档，请使用：

```
Get-Help about_Jobs
```

您可以使用通配符搜索主题。如果您想列出标题以 about_ 开头的可用帮助主题，

请尝试：

```
Get-Help about_*
```

如果您想获得关于 Select-Object 的帮助，可以使用：

```
Get-Help Select-Object
```

您也可以使用别名 help 或 man。

第39.3节：查看帮助主题的在线版本

您可以使用以下命令访问联机帮助文档：

```
Get-Help Get-Command -Online
```

第39.4节：查看示例

显示特定cmdlet的使用示例。

```
Get-Help Get-Command -Examples
```

Chapter 39: Using the Help System

Section 39.1: Updating the Help System

Version > 3.0

Beginning with PowerShell 3.0, you can download and update the offline help documentation using a single cmdlet.

```
Update-Help
```

To update help on multiple computers (or computers not connected to the internet).

Run the following on a computer with the help files

```
Save-Help -DestinationPath \\Server01\Share\PSHelp -Credential $Cred
```

To run on many computers remotely

```
Invoke-Command -ComputerName (Get-Content Servers.txt) -ScriptBlock {Update-Help -SourcePath
\\Server01\Share\Help -Credential $cred}
```

Section 39.2: Using Get-Help

Get-Help can be used to view help in PowerShell. You can search for cmdlets, functions, providers or other topics.

In order to view the help documentation about jobs, use:

```
Get-Help about_Jobs
```

You can search for topics using wildcards. If you want to list available help topics with a title starting with about_, try:

```
Get-Help about_*
```

If you wanted help on Select-Object, you would use:

```
Get-Help Select-Object
```

You can also use the aliases help or man.

Section 39.3: Viewing online version of a help topic

You can access online help documentation using:

```
Get-Help Get-Command -Online
```

Section 39.4: Viewing Examples

Show usage examples for a specific cmdlet.

```
Get-Help Get-Command -Examples
```

第39.5节：查看完整帮助页面

查看该主题的完整文档。

```
Get-Help Get-Command -Full
```

第39.6节：查看特定参数的帮助

您可以使用以下命令查看特定参数的帮助：

```
Get-Help Get-Content -Parameter Path
```

Section 39.5: Viewing the Full Help Page

View the full documentation for the topic.

```
Get-Help Get-Command -Full
```

Section 39.6: Viewing help for a specific parameter

You can view help for a specific parameter using:

```
Get-Help Get-Content -Parameter Path
```

第40章：模块、脚本和函数

PowerShell模块为系统管理员、数据库管理员和开发人员带来了可扩展性。无论它只是作为共享函数和脚本的一种方法
。

PowerShell函数用于避免重复代码。参见[PS函数][1] [1]：PowerShell函数PowerShell脚本用于自动化

管理任务，这些任务由命令行外壳和基于.NET框架的相关cmdlet组成。

第40.1节：函数

函数是一个具名的代码块，用于定义可重用的代码，且应易于使用。它通常包含在脚本中以帮助重用代码（避免代码重复），或作为模块的一部分分发，使其在多个脚本中对他人有用。

函数可能有用的场景：

- 计算一组数字的平均值
- 生成正在运行进程的报告
- 编写一个函数，通过ping计算机并访问c$共享来测试计算机是否“健康”

函数使用function关键字创建，后跟一个单词名称和一个脚本块，脚本块包含在调用函数名时执行的代码。

```
function NameOfFunction {
    Your code
}
```

演示

```
function HelloWorld {
    Write-Host "来自 PowerShell 的问候！"
}
```

用法：

```
> HelloWorld
来自 PowerShell 的问候!
```

第40.2节：脚本

脚本是一个扩展名为.ps1的文本文件，包含将在调用脚本时执行的 PowerShell 命令。由于脚本是保存的文件，因此易于在计算机之间传输。

脚本通常用于解决特定问题，例如：

- 运行每周维护任务
- 在计算机上安装和配置解决方案/应用程序

演示

MyFirstScript.ps1:

Chapter 40: Modules, Scripts and Functions

PowerShell modules bring extendibility to the systems administrator, DBA, and developer. Whether it's simply as a method to share functions and scripts.

PowerShell Functions are to avoid repetitive codes. Refer [PS Functions][1] [1]: PowerShell Functions

PowerShell Scripts are used for automating administrative tasks which consists of command-line shell and associated cmdlets built on top of .NET Framework.

Section 40.1: Function

A function is a named block of code which is used to define reusable code that should be easy to use. It is usually included inside a script to help reuse code (to avoid duplicate code) or distributed as part of a module to make it useful for others in multiple scripts.

Scenarios where a function might be useful:

- Calculate the average of a group of numbers
- Generate a report for running processes
- Write a function that tests is a computer is "healthy" by pinging the computer and accessing the c$-share

Functions are created using the `function` keyword, followed by a single-word name and a script block containing the code to executed when the function name is called.

```
function NameOfFunction {
    Your code
}
```

Demo

```
function HelloWorld {
    Write-Host "Greetings from PowerShell!"
}
```

Usage:

```
> HelloWorld
Greetings from PowerShell!
```

Section 40.2: Script

A script is a text file with the file extension `.ps1` that contains PowerShell commands that will be executed when the script is called. Because scripts are saved files, they are easy to transfer between computers.

Scripts are often written to solve a specific problem, ex.:

- Run a weekly maintenance task
- To install and configure a solution/application on a computer

Demo

MyFirstScript.ps1:

```
Write-Host "Hello World!"
2+2
```

您可以通过输入文件路径来运行脚本，使用：

- 绝对路径，例如 c:\MyFirstScript.ps1
- 相对路径，例如 .\MyFirstScript.ps1 如果您的 PowerShell 控制台当前目录是 C:\

用法：

```
> .\MyFirstScript.ps1
Hello World!
4
```

脚本还可以导入模块，定义自己的函数等。

MySecondScript.ps1：

```
function HelloWorld {
    Write-Host "来自 PowerShell 的问候！"
}

HelloWorld
Write-Host "Let's get started!"
2+2
HelloWorld
```

用法：

```
> .\MySecondScript.ps1
来自 PowerShell 的问候！
让我们开始吧！
4
来自 PowerShell 的问候！
```

第40.3节：模块

模块是一组相关的可重用函数（或 cmdlet），可以轻松分发给其他 PowerShell 用户，并在多个脚本中或直接在控制台中使用。模块通常保存在其自己的目录中，包括：

- 一个或多个带有.psm1文件扩展名的代码文件，包含函数或二进制程序集（.dll）包含 cmdlet
- 一个模块清单.psd1，描述模块的名称、版本、作者、描述，提供的函数/ cmdlet 等。
- 使其工作所需的其他要求，包括依赖项、脚本等。

模块示例：

- 包含对数据集执行行统计的函数/ cmdlet 的模块
- 用于查询和配置数据库的模块

为了让 PowerShell 更容易找到并导入模块，模块通常放置在 $env:PSModulePath 中定义的已知 PowerShell 模块位置之一。

```
Write-Host "Hello World!"
2+2
```

You can run a script by entering the path to the file using an:

- Absolute path, ex. `c:\MyFirstScript.ps1`
- Relative path, ex `.\MyFirstScript.ps1` if the current directory of your PowerShell console was `C:\`

Usage:

```
> .\MyFirstScript.ps1
Hello World!
4
```

A script can also import modules, define its own functions etc.

MySecondScript.ps1:

```
function HelloWorld {
    Write-Host "Greetings from PowerShell!"
}

HelloWorld
Write-Host "Let's get started!"
2+2
HelloWorld
```

Usage:

```
> .\MySecondScript.ps1
Greetings from PowerShell!
Let's get started!
4
Greetings from PowerShell!
```

Section 40.3: Module

A module is a collection of related reusable functions (or cmdlets) that can easily be distributed to other PowerShell users and used in multiple scripts or directly in the console. A module is usually saved in its own directory and consists of:

- One or more code files with the `.psm1` file extension containing functions or binary assemblies (`.dll`) containing cmdlets
- A module manifest `.psd1` describing the modules name, version, author, description, which functions/cmdlets it provides etc.
- Other requirements for it to work incl. dependencies, scripts etc.

Examples of modules:

- A module containing functions/cmdlets that perform statistics on a dataset
- A module for querying and configuring databases

To make it easy for PowerShell to find and import a module, it is often placed in one of the known PowerShell module-locations defined in `$env:PSModulePath`.

演示

列出安装在已知模块位置之一的模块：

```
Get-Module -ListAvailable
```

导入模块，例如 Hyper-V 模块：

```
Import-Module Hyper-V
```

列出模块中可用的命令，例如 Microsoft.PowerShell.Archive 模块

```
> Import-Module Microsoft.PowerShell.Archive
> Get-Command -Module Microsoft.PowerShell.Archive

CommandType Name            Version Source
----------- ----            ------- ------
Function    Compress-Archive 1.0.1.0 Microsoft.PowerShell.Archive
Function    Expand-Archive   1.0.1.0 Microsoft.PowerShell.Archive
```

第40.4节：高级函数

高级函数的行为与cmdlet相同。PowerShell ISE包含两个高级函数的骨架。可通过菜单“编辑”中的“代码片段”，或按Ctrl+J访问它们。（从PS 3.0开始，后续版本可能有所不同）

高级函数包含的关键内容有，

- 内置的、定制的函数帮助，可通过 Get-Help 访问
- 可以使用 [CmdletBinding()] 使函数表现得像cmdlet
- 丰富的参数选项

简单版本：

```
<#
.Synopsis
简短描述
.描述
详细说明
.示例
如何使用此命令的示例
.示例
另一个如何使用此 cmdlet 的示例
#>
function 动词-名词
{
    [CmdletBinding()]
    [OutputType([int])]
    参数
    (
        # 参数1 帮助说明
        [Parameter(必需=$true,
ValueFromPipelineByPropertyName=$true,
                   Position=0)]
        $Param1,

        # Param2 帮助说明
        [int]
        $Param2
```

Demo

List modules that are installed to one of the known module-locations:

```
Get-Module -ListAvailable
```

Import a module, ex. Hyper-V module:

```
Import-Module Hyper-V
```

List available commands in a module, ex. the Microsoft.PowerShell.Archive-module

```
> Import-Module Microsoft.PowerShell.Archive
> Get-Command -Module Microsoft.PowerShell.Archive

CommandType Name             Version Source
----------- ----             ------- ------
Function    Compress-Archive 1.0.1.0 Microsoft.PowerShell.Archive
Function    Expand-Archive   1.0.1.0 Microsoft.PowerShell.Archive
```

Section 40.4: Advanced Functions

Advanced functions behave the in the same way as cmdlets. The PowerShell ISE includes two skeletons of advanced functions. Access these via the menu, edit, code snippets, or by Ctrl+J. (As of PS 3.0, later versions may differ)

Key things that advanced functions include are,

- built-in, customized help for the function, accessible via Get-Help
- can use [CmdletBinding()] which makes the function act like a cmdlet
- extensive parameter options

Simple version:

```
<#
.Synopsis
   Short description
.DESCRIPTION
   Long description
.EXAMPLE
   Example of how to use this cmdlet
.EXAMPLE
   Another example of how to use this cmdlet
#>
function Verb-Noun
{
    [CmdletBinding()]
    [OutputType([int])]
    Param
    (
        # Param1 help description
        [Parameter(Mandatory=$true,
                   ValueFromPipelineByPropertyName=$true,
                   Position=0)]
        $Param1,

        # Param2 help description
        [int]
        $Param2
    )

    开始
    {
    }
    过程
    {
    }
    结束
    {
    }
}
```

完整版本：

```
<#
.Synopsis
简短描述
.描述
详细说明
.示例
如何使用此命令的示例
.示例
另一个如何使用此命令的示例
.输入
此命令的输入（如果有）
.输出
此命令的输出（如果有）
.备注
一般备注
.组件
此命令所属的组件
.角色
此命令所属的角色
.功能
最能描述此命令的功能
#>
function 动词-名词
{
    [CmdletBinding(DefaultParameterSetName='参数集 1',
                   SupportsShouldProcess=$true,
                   PositionalBinding=$false,
                   HelpUri = 'http://www.microsoft.com/',
                   ConfirmImpact='中')]
    [OutputType([字符串])]
    参数
    (
        # 参数1 帮助说明
        [Parameter(必需=$true,
ValueFromPipeline=$true,
                   ValueFromPipelineByPropertyName=$true,
                   ValueFromRemainingArguments=$false,
                   位置=0,
参数集名称='参数集 1')]
        [ValidateNotNull()]
        [ValidateNotNullOrEmpty()]
        [ValidateCount(0,5)]
        [ValidateSet("sun", "moon", "earth")]
        [Alias("p1")]
        $Param1,
```

```
    )

    Begin
    {
    }
    Process
    {
    }
    End
    {
    }
}
```

Complete version:

```
<#
.Synopsis
   Short description
.DESCRIPTION
   Long description
.EXAMPLE
   Example of how to use this cmdlet
.EXAMPLE
   Another example of how to use this cmdlet
.INPUTS
   Inputs to this cmdlet (if any)
.OUTPUTS
   Output from this cmdlet (if any)
.NOTES
   General notes
.COMPONENT
   The component this cmdlet belongs to
.ROLE
   The role this cmdlet belongs to
.FUNCTIONALITY
   The functionality that best describes this cmdlet
#>
function Verb-Noun
{
    [CmdletBinding(DefaultParameterSetName='Parameter Set 1',
                   SupportsShouldProcess=$true,
                   PositionalBinding=$false,
                   HelpUri = 'http://www.microsoft.com/',
                   ConfirmImpact='Medium')]
    [OutputType([String])]
    Param
    (
        # Param1 help description
        [Parameter(Mandatory=$true,
                   ValueFromPipeline=$true,
                   ValueFromPipelineByPropertyName=$true,
                   ValueFromRemainingArguments=$false,
                   Position=0,
                   ParameterSetName='Parameter Set 1')]
        [ValidateNotNull()]
        [ValidateNotNullOrEmpty()]
        [ValidateCount(0,5)]
        [ValidateSet("sun", "moon", "earth")]
        [Alias("p1")]
        $Param1,
        # Param2 帮助说明
        [参数(参数集名称='参数集 1')]
        [允许空值()]
        [允许空集合()]
        [允许空字符串()]
        [验证脚本({$true})]
        [验证范围(0,5)]
        [整数]
        $Param2,

        # Param3 帮助说明
        [参数(ParameterSetName='另一个参数集')]
        [验证模式("[a-z]*")]
        [ValidateLength(0,15)]
        [String]
        $Param3
    )

    开始
    {
    }
    过程
    {
        如果 ($pscmdlet.ShouldProcess("目标", "操作"))
        {
        }
    }
    结束
    {
    }
}
```

```
        # Param2 help description
        [Parameter(ParameterSetName='Parameter Set 1')]
        [AllowNull()]
        [AllowEmptyCollection()]
        [AllowEmptyString()]
        [ValidateScript({$true})]
        [ValidateRange(0,5)]
        [int]
        $Param2,

        # Param3 help description
        [Parameter(ParameterSetName='Another Parameter Set')]
        [ValidatePattern("[a-z]*")]
        [ValidateLength(0,15)]
        [String]
        $Param3
    )

    Begin
    {
    }
    Process
    {
        if ($pscmdlet.ShouldProcess("Target", "Operation"))
        {
        }
    }
    End
    {
    }
}
```

第41章：命名约定

第41.1节：函数

```
Get-User()
```

- 命名函数时使用动词-名词模式。
- 动词表示一个动作，例如Get、Set、New、Read、Write等。参见批准的动词。
- 名词应为单数，即使它作用于多个项目。Get-User()可能返回一个或多个用户。
- 动词和名词均使用帕斯卡命名法。例如Get-UserLogin()

Chapter 41: Naming Conventions

Section 41.1: Functions

```
Get-User()
```

- Use *Verb-Noun* pattern while naming a function.
- Verb implies an action e.g. `Get`, `Set`, `New`, `Read`, `Write` and many more. See approved verbs.
- Noun should be singular even if it acts on multiple items. `Get-User()` may return one or multiple users.
- Use Pascal case for both Verb and Noun. E.g. `Get-UserLogin()`

第42章：常用参数

第42.1节：ErrorAction 参数

可能的取值为 Continue | Ignore | Inquire | SilentlyContinue | Stop | Suspend。

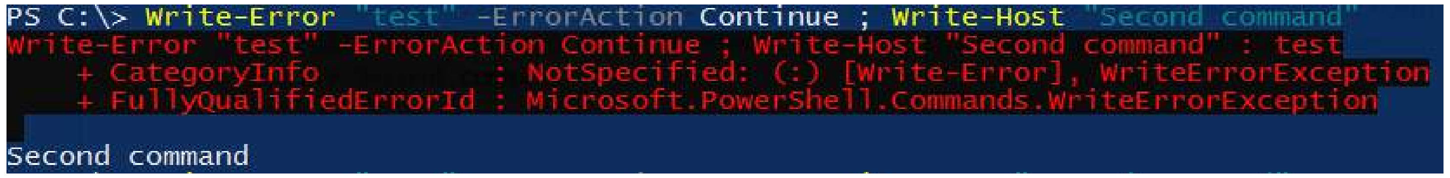
该参数的值将决定 cmdlet 如何处理非终止性错误（例如由
Write-Error 生成的错误；有关错误处理的更多信息，请参见 [*主题尚未创建*]）。

默认值（如果省略此参数）为 Continue。

-ErrorAction Continue

此选项将生成错误消息并继续执行。

```
PS C:\> Write-Error "test" -ErrorAction Continue ; Write-Host "Second command"
```

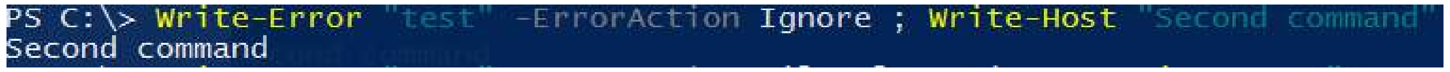


```
PS C:\> Write-Error "test" -ErrorAction Continue ; Write-Host "Second command"
Write-Error "test" -ErrorAction Continue ; Write-Host "Second command" : test
    + CategoryInfo          : NotSpecified: (:) [Write-Error], WriteErrorException
    + FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorException

Second command
```

-ErrorAction Ignore

此选项不会生成任何错误消息，并将继续执行。同时也不会将错误添加到
$Error 自动变量中。
此选项在v3版本中引入。

```
PS C:\> Write-Error "test" -ErrorAction Ignore ; Write-Host "Second command"
```

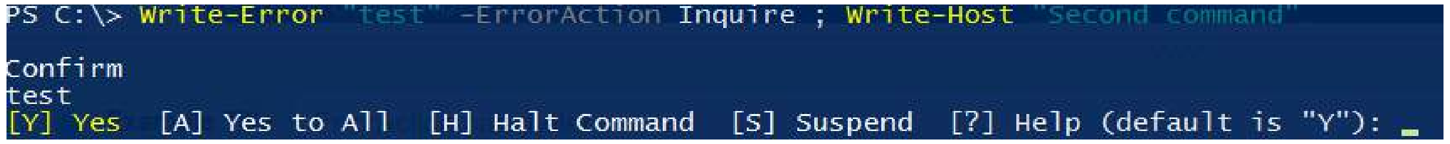


```
PS C:\> Write-Error "test" -ErrorAction Ignore ; Write-Host "Second command"
Second command
```

-ErrorAction Inquire

此选项将产生错误消息，并提示用户选择要采取的操作。

```
PS C:\> Write-Error "test" -ErrorAction Inquire ; Write-Host "Second command"
```

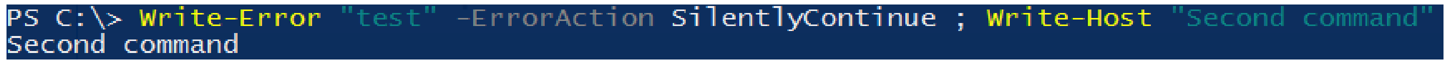


```
PS C:\> Write-Error "test" -ErrorAction Inquire ; Write-Host "Second command"

Confirm
test
[Y] Yes  [A] Yes to All  [H] Halt Command  [S] Suspend  [?] Help (default is "Y"): _
```

-ErrorAction SilentlyContinue

此选项不会产生错误消息，并将继续执行。所有错误将添加到$Error自动变量中。

```
PS C:\> Write-Error "test" -ErrorAction SilentlyContinue ; Write-Host "Second command"
```



```
PS C:\> Write-Error "test" -ErrorAction SilentlyContinue ; Write-Host "Second command"
Second command
```

-ErrorAction Stop

Chapter 42: Common parameters

Section 42.1: ErrorAction parameter

Possible values are `Continue` | `Ignore` | `Inquire` | `SilentlyContinue` | `Stop` | `Suspend`.

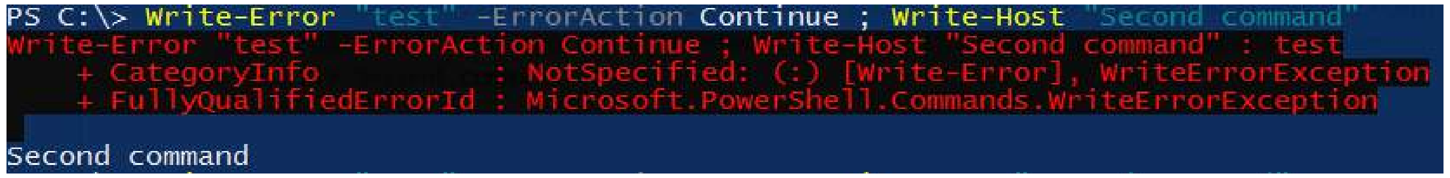
Value of this parameter will determine how the cmdlet will handle non-terminating errors (those generated from Write-Error for example; to learn more about error handling see [*topic not yet created*]).

Default value (if this parameter is omitted) is `Continue`.

-ErrorAction Continue

This option will produce an error message and will continue with execution.

```
PS C:\> Write-Error "test" -ErrorAction Continue ; Write-Host "Second command"
```

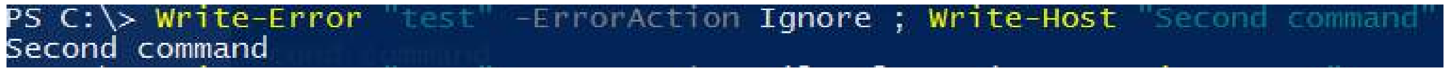


```
PS C:\> Write-Error "test" -ErrorAction Continue ; Write-Host "Second command"
Write-Error "test" -ErrorAction Continue ; Write-Host "Second command" : test
    + CategoryInfo          : NotSpecified: (:) [Write-Error], WriteErrorException
    + FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorException

Second command
```

-ErrorAction Ignore

This option will not produce any error message and will continue with execution. Also no errors will be added to `$Error` automatic variable.
This option was introduced in v3.

```
PS C:\> Write-Error "test" -ErrorAction Ignore ; Write-Host "Second command"
```

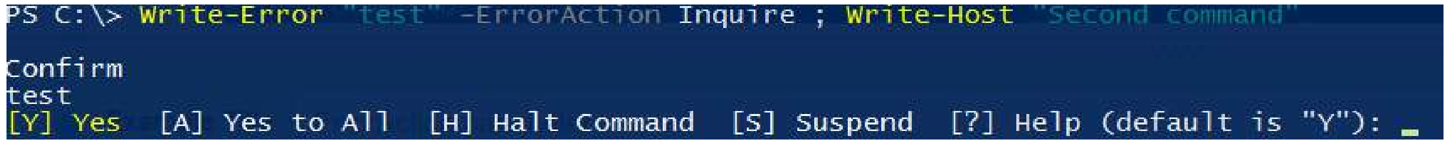


```
PS C:\> Write-Error "test" -ErrorAction Ignore ; Write-Host "Second command"
Second command
```

-ErrorAction Inquire

This option will produce an error message and will prompt user to choose an action to take.

```
PS C:\> Write-Error "test" -ErrorAction Inquire ; Write-Host "Second command"
```

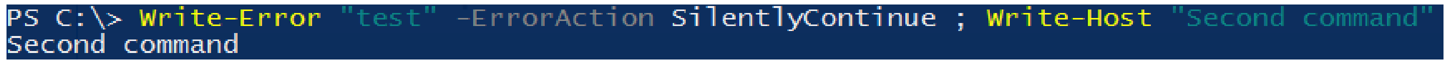


```
PS C:\> Write-Error "test" -ErrorAction Inquire ; Write-Host "Second command"

Confirm
test
[Y] Yes  [A] Yes to All  [H] Halt Command  [S] Suspend  [?] Help (default is "Y"): _
```

-ErrorAction SilentlyContinue

This option will not produce an error message and will continue with execution. All errors will be added to `$Error` automatic variable.

```
PS C:\> Write-Error "test" -ErrorAction SilentlyContinue ; Write-Host "Second command"
```

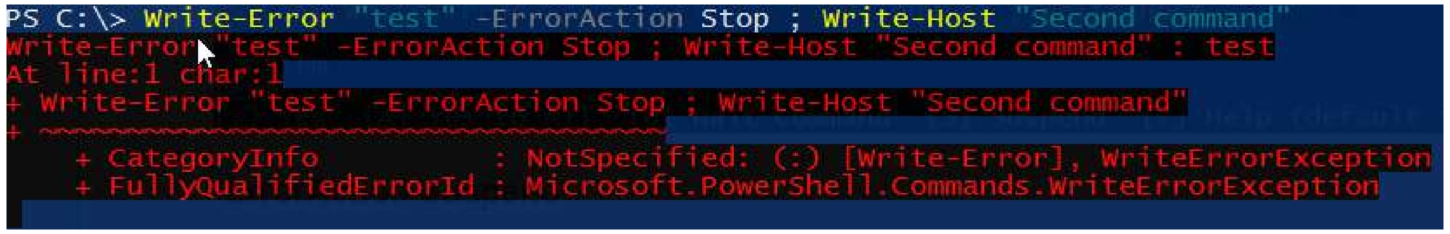


```
PS C:\> Write-Error "test" -ErrorAction SilentlyContinue ; Write-Host "Second command"
Second command
```

-ErrorAction Stop

此选项将产生错误消息，并且不会继续执行。

```
PS C:\> Write-Error "test" -ErrorAction Stop ; Write-Host "Second command"
```



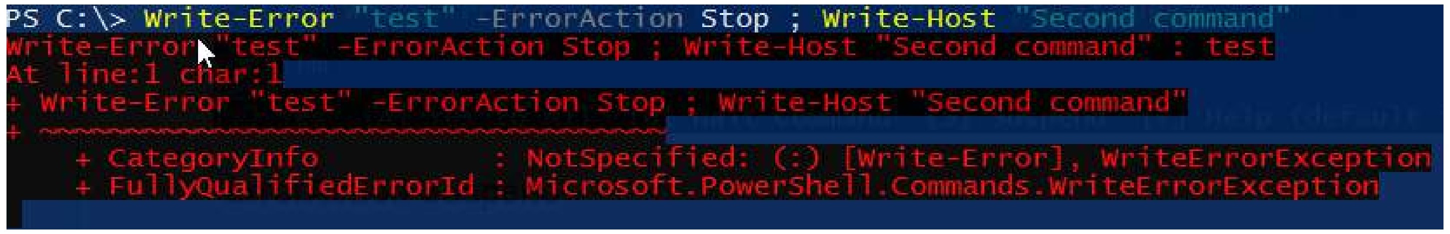
```
PS C:\> Write-Error "test" -ErrorAction Stop ; Write-Host "Second command"
Write-Error "test" -ErrorAction Stop ; Write-Host "Second command" : test
At line:1 char:1
+ Write-Error "test" -ErrorAction Stop ; Write-Host "Second command"
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [Write-Error], WriteErrorException
    + FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorException
```

-ErrorAction Suspend

仅在 Powershell 工作流中可用。使用时，如果命令遇到错误，工作流将被挂起。这允许对该错误进行调查，并提供恢复工作流的可能性。要了解更多关于工作流系统的信息，请参见[主题尚未创建]。

This option will produce an error message and will not continue with execution.

```
PS C:\> Write-Error "test" -ErrorAction Stop ; Write-Host "Second command"
```



```
PS C:\> Write-Error "test" -ErrorAction Stop ; Write-Host "Second command"
Write-Error "test" -ErrorAction Stop ; Write-Host "Second command" : test
At line:1 char:1
+ Write-Error "test" -ErrorAction Stop ; Write-Host "Second command"
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [Write-Error], WriteErrorException
    + FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorException
```

-ErrorAction Suspend

Only available in Powershell Workflows. When used, if the command runs into an error, the workflow is suspended. This allows investigation of such error and gives a possibility to resume the workflow. To learn more about Workflow system, see [topic not yet created].

第43章：参数集

参数集用于限制参数的可能组合，或在选择一个或多个参数时强制使用某些参数。

示例将解释参数集的使用及其原因。

第43.1节：参数集用于在选择另一个参数时强制使用某个参数

例如，当提供参数 User 时，您希望强制使用参数 Password。（反之亦然）

```
函数 Do-Something
{
    参数
    (
        [参数(必需=$true)]
        [String]$SomeThingToDo,
        [Parameter(ParameterSetName="Credentials", mandatory=$false)]
        [String]$Computername = "LocalHost",
        [Parameter(ParameterSetName="Credentials", mandatory=$true)]
        [String]$User,
        [Parameter(ParameterSetName="Credentials", mandatory=$true)]
        [SecureString]$Password
    )

    #执行某些操作
}

# 这将不起作用，他会要求输入用户和密码
Do-Something -SomeThingToDo 'get-help about_Functions_Advanced' -ComputerName

# 这将不起作用，他会要求输入密码
Do-Something -SomeThingToDo 'get-help about_Functions_Advanced' -User
```

第43.2节：参数集以限制参数组合

```
函数 Do-Something
{
    参数
    (
        [参数(必需=$true)]
        [String]$SomeThingToDo,
        [Parameter(ParameterSetName="Silently", mandatory=$false)]
        [Switch]$Silently,
        [Parameter(ParameterSetName="Loudly", mandatory=$false)]
        [Switch]$Loudly
    )

    #执行某些操作
}

# 这将无法工作，因为你不能同时使用 Silently 和 Loudly 组合
Do-Something -SomeThingToDo 'get-help about_Functions_Advanced' -Silently -Loudly
```

Chapter 43: Parameter sets

Parameter sets are used to limit the possible combination of parameters, or to enforce the use of parameters when 1 or more parameters are selected.

The examples will explain the use and reason of a parameter set.

Section 43.1: Parameter set to enforce the use of a parameter when a other is selected

When you want for example enforce the use of the parameter Password if the parameter User is provided. (and vice versa)

```
Function Do-Something
{
    Param
    (
        [Parameter(Mandatory=$true)]
        [String]$SomeThingToDo,
        [Parameter(ParameterSetName="Credentials", mandatory=$false)]
        [String]$Computername = "LocalHost",
        [Parameter(ParameterSetName="Credentials", mandatory=$true)]
        [String]$User,
        [Parameter(ParameterSetName="Credentials", mandatory=$true)]
        [SecureString]$Password
    )

    #Do something
}

# This will not work he will ask for user and password
Do-Something -SomeThingToDo 'get-help about_Functions_Advanced' -ComputerName

# This will not work he will ask for password
Do-Something -SomeThingToDo 'get-help about_Functions_Advanced' -User
```

Section 43.2: Parameter set to limit the combination of parameters

```
Function Do-Something
{
    Param
    (
        [Parameter(Mandatory=$true)]
        [String]$SomeThingToDo,
        [Parameter(ParameterSetName="Silently", mandatory=$false)]
        [Switch]$Silently,
        [Parameter(ParameterSetName="Loudly", mandatory=$false)]
        [Switch]$Loudly
    )

    #Do something
}

# This will not work because you can not use the combination Silently and Loudly
Do-Something -SomeThingToDo 'get-help about_Functions_Advanced' -Silently -Loudly
```

第44章：PowerShell 动态参数

第44.1节："简单"动态参数

如果$SomeUsefulNumber大于5，则此示例会向 MyTestFunction 添加一个新参数。

```
函数 MyTestFunction
{
    [CmdletBinding(DefaultParameterSetName='默认配置')]
    参数
    (
        [Parameter(必需=$true)][int]$SomeUsefulNumber
    )

动态参数
    {
        $paramDictionary = New-Object -Type
System.Management.Automation.RuntimeDefinedParameterDictionary
        $attributes = New-Object System.Management.Automation.ParameterAttribute
        $attributes.ParameterSetName = "__AllParameterSets"
        $attributes.必须 = $true
        $attributeCollection = New-Object -Type
System.Collections.ObjectModel.Collection[System.Attribute]
        $attributeCollection.Add($attributes)
        # 如果 "SomeUsefulNumber" 大于 5，则添加参数 "MandatoryParam1"
        if($SomeUsefulNumber -gt 5)
        {
            # 创建一个名为 "MandatoryParam1" 的必填字符串参数
            $dynParam1 = New-Object -Type
System.Management.Automation.RuntimeDefinedParameter("MandatoryParam1", [String],
$attributeCollection)
            # 将新参数添加到字典中
            $paramDictionary.Add("MandatoryParam1", $dynParam1)
        }
        return $paramDictionary
    }

    process
    {
        Write-Host "SomeUsefulNumber = $SomeUsefulNumber"
        # 注意动态参数需要特定的语法
        Write-Host ("MandatoryParam1 = {0}" -f $PSBoundParameters.MandatoryParam1)
    }

}
```

用法：

```
PS >  MyTestFunction -SomeUsefulNumber 3
SomeUsefulNumber = 3
MandatoryParam1 =

PS >  MyTestFunction -有用数字 6
命令行管道位置的 cmdlet MyTestFunction 1
为以下参数提供值：
必填参数1：
```

Chapter 44: PowerShell Dynamic Parameters

Section 44.1: "Simple" dynamic parameter

This example adds a new parameter to MyTestFunction if $SomeUsefulNumber is greater than 5.

```
function MyTestFunction
{
    [CmdletBinding(DefaultParameterSetName='DefaultConfiguration')]
    Param
    (
        [Parameter(Mandatory=$true)][int]$SomeUsefulNumber
    )

    DynamicParam
    {
        $paramDictionary = New-Object -Type
System.Management.Automation.RuntimeDefinedParameterDictionary
        $attributes = New-Object System.Management.Automation.ParameterAttribute
        $attributes.ParameterSetName = "__AllParameterSets"
        $attributes.Mandatory = $true
        $attributeCollection = New-Object -Type
System.Collections.ObjectModel.Collection[System.Attribute]
        $attributeCollection.Add($attributes)
        # If "SomeUsefulNumber" is greater than 5, then add the "MandatoryParam1" parameter
        if($SomeUsefulNumber -gt 5)
        {
            # Create a mandatory string parameter called "MandatoryParam1"
            $dynParam1 = New-Object -Type
System.Management.Automation.RuntimeDefinedParameter("MandatoryParam1", [String],
$attributeCollection)
            # Add the new parameter to the dictionary
            $paramDictionary.Add("MandatoryParam1", $dynParam1)
        }
        return $paramDictionary
    }

    process
    {
        Write-Host "SomeUsefulNumber = $SomeUsefulNumber"
        # Notice that dynamic parameters need a specific syntax
        Write-Host ("MandatoryParam1 = {0}" -f $PSBoundParameters.MandatoryParam1)
    }

}
```

Usage:

```
PS >  MyTestFunction -SomeUsefulNumber 3
SomeUsefulNumber = 3
MandatoryParam1 =

PS >  MyTestFunction -SomeUsefulNumber 6
cmdlet MyTestFunction at command pipeline position 1
Supply values for the following parameters:
MandatoryParam1:
```

```
PS >MyTestFunction -有用数字 6 -必填参数1 测试
有用数字 = 6
必填参数1 = 测试
```

在第二个使用示例中，你可以清楚地看到缺少了一个参数。

动态参数在自动补全时也会被考虑。
如果你在行尾按下 ctrl + 空格，情况如下：

```
PS >MyTestFunction -有用数字 3 -<ctrl+space>
详细信息              警告操作          警告变量        输出缓冲区
调试                  信息操作     信息变量   管道变量
错误操作          错误变量          输出变量

PS >MyTestFunction -有用数字 6 -<ctrl+space>
必填参数1       错误操作            错误变量          输出变量
详细信息              警告操作          警告变量        输出缓冲区
调试                  信息操作     信息变量   管道变量
```

```
PS >MyTestFunction -SomeUsefulNumber 6 -MandatoryParam1 test
SomeUsefulNumber = 6
MandatoryParam1 = test
```

In the second usage example, you can clearly see that a parameter is missing.

Dynamic parameters are also taken into account with auto completion.
Here's what happens if you hit ctrl + space at the end of the line:

```
PS >MyTestFunction -SomeUsefulNumber 3 -<ctrl+space>
Verbose              WarningAction        WarningVariable      OutBuffer
Debug                InformationAction    InformationVariable  PipelineVariable
ErrorAction          ErrorVariable        OutVariable

PS >MyTestFunction -SomeUsefulNumber 6 -<ctrl+space>
MandatoryParam1      ErrorAction          ErrorVariable        OutVariable
Verbose              WarningAction        WarningVariable      OutBuffer
Debug                InformationAction    InformationVariable  PipelineVariable
```

第45章：PowerShell中的图形用户界面（GUI）

第45.1节：Get-Service cmdlet的WPF图形用户界面

```
Add-Type -AssemblyName PresentationFramework

[xml]$XAMLWindow = '
<Window
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
     Height="Auto"
SizeToContent="WidthAndHeight"
    Title="Get-Service">
<ScrollViewer Padding="10,10,10,0" ScrollViewer.VerticalScrollBarVisibility="Disabled">
        <StackPanel>
<StackPanel Orientation="Horizontal">
                <Label Margin="10,10,0,10">计算机名：</Label>
                    <TextBox Name="Input" Margin="10" Width="250px"></TextBox>
               </StackPanel>
               <DockPanel>
<Button Name="ButtonGetService" Content="获取服务" Margin="10" Width="150px"
IsEnabled="false"/>
<Button Name="ButtonClose" Content="关闭" HorizontalAlignment="Right" Margin="10"
Width="50px"/>
</DockPanel>
        </StackPanel>
    </ScrollViewer >
</Window>
'

# 创建窗口对象
$Reader=(New-Object System.Xml.XmlNodeReader $XAMLWindow)
$Window=[Windows.Markup.XamlReader]::Load( $Reader )

# 输入框的 TextChanged 事件处理
$TextboxInput = $Window.FindName("Input")
$TextboxInput.add_TextChanged.Invoke({
     $ComputerName = $TextboxInput.Text
     $ButtonGetService.IsEnabled = $ComputerName -ne ''
})

# ButtonClose 的点击事件处理
$ButtonClose = $Window.FindName("ButtonClose")
$ButtonClose.add_Click.Invoke({
     $Window.Close();
})

# ButtonGetService 的点击事件处理
$ButtonGetService = $Window.FindName("ButtonGetService")
$ButtonGetService.add_Click.Invoke({
     $ComputerName = $TextboxInput.text.Trim()
     try{
          Get-Service -ComputerName $computerName | Out-GridView -Title "Get-Service on
$ComputerName"
     } catch {

[System.Windows.MessageBox]::Show($_.exception.message,"错误",[System.Windows.MessageBoxButton]::确定,[System.Wind
ows.MessageBoxImage]::错误)
     }
})
```

Chapter 45: GUI in PowerShell

Section 45.1: WPF GUI for Get-Service cmdlet

```
Add-Type -AssemblyName PresentationFramework

[xml]$XAMLWindow = '
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Height="Auto"
    SizeToContent="WidthAndHeight"
    Title="Get-Service">
    <ScrollViewer Padding="10,10,10,0" ScrollViewer.VerticalScrollBarVisibility="Disabled">
        <StackPanel>
            <StackPanel Orientation="Horizontal">
                <Label Margin="10,10,0,10">ComputerName:</Label>
                <TextBox Name="Input" Margin="10" Width="250px"></TextBox>
            </StackPanel>
            <DockPanel>
                <Button Name="ButtonGetService" Content="Get-Service" Margin="10" Width="150px"
IsEnabled="false"/>
                <Button Name="ButtonClose" Content="Close" HorizontalAlignment="Right" Margin="10"
Width="50px"/>
            </DockPanel>
        </StackPanel>
    </ScrollViewer >
</Window>
'

# Create the Window Object
$Reader=(New-Object System.Xml.XmlNodeReader $XAMLWindow)
$Window=[Windows.Markup.XamlReader]::Load( $Reader )

# TextChanged Event Handler for Input
$TextboxInput = $Window.FindName("Input")
$TextboxInput.add_TextChanged.Invoke({
    $ComputerName = $TextboxInput.Text
    $ButtonGetService.IsEnabled = $ComputerName -ne ''
})

# Click Event Handler for ButtonClose
$ButtonClose = $Window.FindName("ButtonClose")
$ButtonClose.add_Click.Invoke({
    $Window.Close();
})

# Click Event Handler for ButtonGetService
$ButtonGetService = $Window.FindName("ButtonGetService")
$ButtonGetService.add_Click.Invoke({
    $ComputerName = $TextboxInput.text.Trim()
    try{
        Get-Service -ComputerName $computerName | Out-GridView -Title "Get-Service on
$ComputerName"
    }catch{

[System.Windows.MessageBox]::Show($_.exception.message,"Error",[System.Windows.MessageBoxButton]::O
K,[System.Windows.MessageBoxImage]::Error)
    }
})
```

```
# 打开窗口
$Window.ShowDialog() | Out-Null
```

这将创建一个对话窗口，允许用户选择计算机名称，然后显示该计算机上服务及其状态的表格。

此示例使用的是WPF而非Windows窗体。

```
# Open the Window
$Window.ShowDialog() | Out-Null
```

This creates a dialog window which allows the user to select a computer name, then will display a table of services and their statuses on that computer.
This example uses WPF rather than Windows Forms.

第46章：URL编码/解码

第46.1节：使用`[System.Web.HttpUtility]::UrlEncode()`编码查询字符串

```
$scheme = 'https'
$url_format = '{0}://example.vertigion.com/foos?{1}'
$qs_data = @{
    'foo1'='bar1';
    'foo2'= 'complex;/?:@&=+$, bar''"';
    'complex;/?:@&=+$, foo''"'='bar2';
}

[System.Collections.ArrayList] $qs_array = @()
foreach ($qs in $qs_data.GetEnumerator()) {
    $qs_key = [System.Web.HttpUtility]::UrlEncode($qs.Name)
    $qs_value = [System.Web.HttpUtility]::UrlEncode($qs.Value)
    $qs_array.Add("${qs_key}=${qs_value}") | Out-Null
}

$url = $url_format -f @([uri]::"UriScheme${scheme}", ($qs_array -join '&'))
```

使用[System.Web.HttpUtility]::UrlEncode()时，您会注意到空格被转换成加号（+），而不是 %20：

> https://example.vertigion.com/foos?
> foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&
> complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+foo%27%22=bar2&foo1=bar1

第46.2节：快速开始：编码

```
$url1 = [uri]::EscapeDataString("http://test.com?test=my value")
# url1: http%3A%2F%2Ftest.com%3Ftest%3Dmy%20value

$url2 = [uri]::EscapeUriString("http://test.com?test=my value")
# url2: http://test.com?test=my%20value

# HttpUtility 需要至少安装 .NET 1.1。
$url3 = [System.Web.HttpUtility]::UrlEncode("http://test.com?test=my value")
# url3: http%3a%2f%2ftest.com%3ftest%3dmy+value
```

注意：有关 HTTPUtility 的更多信息。

第46.3节：快速入门：解码

注意：这些示例使用上面快速入门：编码部分创建的变量。

```
# url1: http%3A%2F%2Ftest.com%3Ftest%3Dmy%20value
[uri]::UnescapeDataString($url1)
# 返回值：http://test.com?test=my value

# url2: http://test.com?test=my%20value
[uri]::UnescapeDataString($url2)
# 返回值：http://test.com?test=my value
```

Chapter 46: URL Encode/Decode

Section 46.1: Encode Query String with `[System.Web.HttpUtility]::UrlEncode()`

```
$scheme = 'https'
$url_format = '{0}://example.vertigion.com/foos?{1}'
$qs_data = @{
    'foo1'='bar1';
    'foo2'= 'complex;/?:@&=+$, bar''"';
    'complex;/?:@&=+$, foo''"'='bar2';
}

[System.Collections.ArrayList] $qs_array = @()
foreach ($qs in $qs_data.GetEnumerator()) {
    $qs_key = [System.Web.HttpUtility]::UrlEncode($qs.Name)
    $qs_value = [System.Web.HttpUtility]::UrlEncode($qs.Value)
    $qs_array.Add("${qs_key}=${qs_value}") | Out-Null
}

$url = $url_format -f @([uri]::"UriScheme${scheme}", ($qs_array -join '&'))
```

With [System.Web.HttpUtility]::UrlEncode(), you will notice that spaces are turned into plus signs (+) instead of %20:

> https://example.vertigion.com/foos?
> foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&
> complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+foo%27%22=bar2&foo1=bar1

Section 46.2: Quick Start: Encoding

```
$url1 = [uri]::EscapeDataString("http://test.com?test=my value")
# url1: http%3A%2F%2Ftest.com%3Ftest%3Dmy%20value

$url2 = [uri]::EscapeUriString("http://test.com?test=my value")
# url2: http://test.com?test=my%20value

# HttpUtility requires at least .NET 1.1 to be installed.
$url3 = [System.Web.HttpUtility]::UrlEncode("http://test.com?test=my value")
# url3: http%3a%2f%2ftest.com%3ftest%3dmy+value
```

Note: More info on HTTPUtility.

Section 46.3: Quick Start: Decoding

Note: these examples use the variables created in the *Quick Start: Encoding* section above.

```
# url1: http%3A%2F%2Ftest.com%3Ftest%3Dmy%20value
[uri]::UnescapeDataString($url1)
# Returns: http://test.com?test=my value

# url2: http://test.com?test=my%20value
[uri]::UnescapeDataString($url2)
# Returns: http://test.com?test=my value
# url3: http%3a%2f%2ftest.com%3ftest%3dmy+value
[uri]::UnescapeDataString($url3)
# 返回值：http://test.com?test=my+value

# 注意：没有 `[uri]::UnescapeUriString()`；
#       这是合理的，因为 `[uri]::UnescapeDataString()`
#       函数处理了它应处理的所有内容，甚至更多。

# HttpUtility 需要至少安装 .NET 1.1。
# url1: http%3A%2F%2Ftest.com%3Ftest%3Dmy%20value
[System.Web.HttpUtility]::UrlDecode($url1)
# 返回值：http://test.com?test=my value

# HttpUtility 需要至少安装 .NET 1.1。
# url2: http://test.com?test=my%20value
[System.Web.HttpUtility]::UrlDecode($url2)
# 返回值：http://test.com?test=my value

# HttpUtility 需要至少安装 .NET 1.1。
# url3: http%3a%2f%2ftest.com%3ftest%3dmy+value
[System.Web.HttpUtility]::UrlDecode($url3)
# 返回值：http://test.com?test=my value
```

注意：有关 HTTPUtility 的更多信息。

第46.4节：使用`[uri]::EscapeDataString()`编码查询字符串

```
$scheme = 'https'
$url_format = '{0}://example.vertigion.com/foos?{1}'
$qs_data = @{
    'foo1'='bar1';
    'foo2'= 'complex;/?:@&=+$, bar''"';
    'complex;/?:@&=+$, foo''"'='bar2';
}

[System.Collections.ArrayList] $qs_array = @()
foreach ($qs in $qs_data.GetEnumerator()) {
    $qs_key = [uri]::EscapeDataString($qs.Name)
    $qs_value = [uri]::EscapeDataString($qs.Value)
    $qs_array.Add("${qs_key}=${qs_value}") | Out-Null
}

$url = $url_format -f @([uri]::"UriScheme${scheme}", ($qs_array -join '&'))
```

使用 [uri]::EscapeDataString() 时，你会注意到撇号（'）没有被编码：

> https://example.vertigion.com/foos?
> foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar'%22&
> complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20foo'%22=bar2&foo1=bar1

第46.5节：使用 `[uri]::UnescapeDataString()` 解码URL

使用[uri]::EscapeDataString()编码

首先，我们将解码上面示例中使用[uri]::EscapeDataString()编码的URL和查询字符串：

```
# url3: http%3a%2f%2ftest.com%3ftest%3dmy+value
[uri]::UnescapeDataString($url3)
# Returns: http://test.com?test=my+value

# Note: There is no `[uri]::UnescapeUriString()`;
#       which makes sense since the `[uri]::UnescapeDataString()`
#       function handles everything it would handle plus more.

# HttpUtility requires at least .NET 1.1 to be installed.
# url1: http%3A%2F%2Ftest.com%3Ftest%3Dmy%20value
[System.Web.HttpUtility]::UrlDecode($url1)
# Returns: http://test.com?test=my value

# HttpUtility requires at least .NET 1.1 to be installed.
# url2: http://test.com?test=my%20value
[System.Web.HttpUtility]::UrlDecode($url2)
# Returns: http://test.com?test=my value

# HttpUtility requires at least .NET 1.1 to be installed.
# url3: http%3a%2f%2ftest.com%3ftest%3dmy+value
[System.Web.HttpUtility]::UrlDecode($url3)
# Returns: http://test.com?test=my value
```

Note: More info on HTTPUtility.

Section 46.4: Encode Query String with `[uri]::EscapeDataString()`

```
$scheme = 'https'
$url_format = '{0}://example.vertigion.com/foos?{1}'
$qs_data = @{
    'foo1'='bar1';
    'foo2'= 'complex;/?:@&=+$, bar''"';
    'complex;/?:@&=+$, foo''"'='bar2';
}

[System.Collections.ArrayList] $qs_array = @()
foreach ($qs in $qs_data.GetEnumerator()) {
    $qs_key = [uri]::EscapeDataString($qs.Name)
    $qs_value = [uri]::EscapeDataString($qs.Value)
    $qs_array.Add("${qs_key}=${qs_value}") | Out-Null
}

$url = $url_format -f @([uri]::"UriScheme${scheme}", ($qs_array -join '&'))
```

With `[uri]::EscapeDataString()`, you will notice that the apostrophe (') was not encoded:

> https://example.vertigion.com/foos?
> foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar'%22&
> complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20foo'%22=bar2&foo1=bar1

Section 46.5: Decode URL with `[uri]::UnescapeDataString()`

Encoded with `[uri]::EscapeDataString()`

First, we'll decode the URL and Query String encoded with `[uri]::EscapeDataString()` in the above example:

> https://example.vertigion.com/foos?
> foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar'%22&
> complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20foo'%22=bar2&foo1=bar1

```
$url =
'https://example.vertigion.com/foos?foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar''%22&complex%3
B%2F%3F%3A%40%26%3D%2B%24%2C%20foo''%22=bar2&foo1=bar1'
$url_parts_regex = '^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?' # 参见备注

if ($url -match $url_parts_regex) {
    $url_parts = @{
        'Scheme' = $Matches[2];
        'Server' = $Matches[4];
        'Path' = $Matches[5];
        'QueryString' = $Matches[7];
        'QueryStringParts' = @{}
    }

    foreach ($qs in $query_string.Split('&')) {
        $qs_key, $qs_value = $qs.Split('=')
        $url_parts.QueryStringParts.Add(
            [uri]::UnescapeDataString($qs_key),
            [uri]::UnescapeDataString($qs_value)
        ) | Out-Null
    }
} else {
    Throw [System.Management.Automation.ParameterBindingException] "Invalid URL Supplied"
}
```

这将返回给你 [hashtable]$url_parts；其等于（注意：复杂部分中的空格即为空格）：

```
PS > $url_parts

名称                          值
----                          -----
方案                        https
路径                          /foos
服务器                        example.vertigion.com
查询字符串
  foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar'%22&complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20foo'%
22=bar2&foo1=bar1
查询字符串部分              {foo2, complex;/?:@&=+$, foo'", foo1}


PS > $url_parts.QueryStringParts

名称                          值
----                          -----
foo2                          complex;/?:@&=+$, bar'"
complex;/?:@&=+$, foo'"        bar2
foo1                          bar1
```

使用 [System.Web.HttpUtility]::UrlEncode()编码现在，我们将解

码上面示例中使用 [System.Web.HttpUtility]::UrlEncode()编码的URL和查询字符串：

> https://example.vertigion.com/foos?

> https://example.vertigion.com/foos?
> foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar'%22&
> complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20foo'%22=bar2&foo1=bar1

```
$url =
'https://example.vertigion.com/foos?foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar''%22&complex%3
B%2F%3F%3A%40%26%3D%2B%24%2C%20foo''%22=bar2&foo1=bar1'
$url_parts_regex = '^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?' # See Remarks

if ($url -match $url_parts_regex) {
    $url_parts = @{
        'Scheme' = $Matches[2];
        'Server' = $Matches[4];
        'Path' = $Matches[5];
        'QueryString' = $Matches[7];
        'QueryStringParts' = @{}
    }

    foreach ($qs in $query_string.Split('&')) {
        $qs_key, $qs_value = $qs.Split('=')
        $url_parts.QueryStringParts.Add(
            [uri]::UnescapeDataString($qs_key),
            [uri]::UnescapeDataString($qs_value)
        ) | Out-Null
    }
} else {
    Throw [System.Management.Automation.ParameterBindingException] "Invalid URL Supplied"
}
```

This gives you back `[hashtable]$url_parts`; which equals (**Note:** the *spaces* in the complex parts are *spaces*):

```
PS > $url_parts

Name                           Value
----                           -----
Scheme                         https
Path                           /foos
Server                         example.vertigion.com
QueryString
 foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar'%22&complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20foo'%
22=bar2&foo1=bar1
QueryStringParts               {foo2, complex;/?:@&=+$, foo'", foo1}


PS > $url_parts.QueryStringParts

Name                           Value
----                           -----
foo2                           complex;/?:@&=+$, bar'"
complex;/?:@&=+$, foo'"        bar2
foo1                           bar1
```

Encoded with `[System.Web.HttpUtility]::UrlEncode()`

Now, we'll decode the URL and Query String encoded with `[System.Web.HttpUtility]::UrlEncode()` in the above example:

> https://example.vertigion.com/foos?

> foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&
> complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+foo%27%22=bar2&foo1=bar1

```
$url =
'https://example.vertigion.com/foos?foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&complex%3b
%2f%3f%3a%40%26%3d%2b%24%2c+foo%27%22=bar2&foo1=bar1'
$url_parts_regex = '^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?' # 参见备注

if ($url -match $url_parts_regex) {
    $url_parts = @{
        'Scheme' = $Matches[2];
        'Server' = $Matches[4];
        'Path' = $Matches[5];
        'QueryString' = $Matches[7];
        'QueryStringParts' = @{}
    }

    foreach ($qs in $query_string.Split('&')) {
        $qs_key, $qs_value = $qs.Split('=')
        $url_parts.QueryStringParts.Add(
            [uri]::UnescapeDataString($qs_key),
            [uri]::UnescapeDataString($qs_value)
        ) | Out-Null
    }
} else {
    Throw [System.Management.Automation.ParameterBindingException] "Invalid URL Supplied"
}
```

这会返回给你 [hashtable]$url_parts，其等于（**注意：**复杂部分中的*空格*在第一部分是*加号*（+），在第二部分是*空格*）：

```
PS > $url_parts

名称                          值
----                          -----
方案                          https
路径                          /foos
服务器                        example.vertigion.com
查询字符串
 foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+foo%27%
22=bar2&foo1=bar1
查询字符串部分                {foo2, complex;/?:@&=+$, foo'", foo1}


PS > $url_parts.QueryStringParts

名称                          值
----                          -----
foo2                          complex;/?:@&=+$, bar'"
complex;/?:@&=+$, foo'"       bar2
foo1                          bar1
```

第46.6节：使用`[System.Web.HttpUtility]::UrlDecode()`解码URL

使用[uri]::EscapeDataString()编码

首先，我们将解码上面示例中使用[uri]::EscapeDataString()编码的URL和查询字符串：

> foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&
> complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+foo%27%22=bar2&foo1=bar1

```
$url =
'https://example.vertigion.com/foos?foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&complex%3b
%2f%3f%3a%40%26%3d%2b%24%2c+foo%27%22=bar2&foo1=bar1'
$url_parts_regex = '^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?' # See Remarks

if ($url -match $url_parts_regex) {
    $url_parts = @{
        'Scheme' = $Matches[2];
        'Server' = $Matches[4];
        'Path' = $Matches[5];
        'QueryString' = $Matches[7];
        'QueryStringParts' = @{}
    }

    foreach ($qs in $query_string.Split('&')) {
        $qs_key, $qs_value = $qs.Split('=')
        $url_parts.QueryStringParts.Add(
            [uri]::UnescapeDataString($qs_key),
            [uri]::UnescapeDataString($qs_value)
        ) | Out-Null
    }
} else {
    Throw [System.Management.Automation.ParameterBindingException] "Invalid URL Supplied"
}
```

This gives you back [hashtable]$url_parts, which equals (**Note:** the *spaces* in the complex parts are *plus signs* (+) in the first part and *spaces* in the second part):

```
PS > $url_parts

Name                           Value
----                           -----
Scheme                         https
Path                           /foos
Server                         example.vertigion.com
QueryString
 foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+foo%27%
22=bar2&foo1=bar1
QueryStringParts               {foo2, complex;/?:@&=+$, foo'", foo1}


PS > $url_parts.QueryStringParts

Name                           Value
----                           -----
foo2                           complex;/?:@&=+$, bar'"
complex;/?:@&=+$, foo'"        bar2
foo1                           bar1
```

Section 46.6: Decode URL with `[System.Web.HttpUtility]::UrlDecode()`

Encoded with [uri]::EscapeDataString()

First, we'll decode the URL and Query String encoded with [uri]::EscapeDataString() in the above example:

> https://example.vertigion.com/foos?
> foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar'%22&
> complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20foo'%22=bar2&foo1=bar1

```
$url =
'https://example.vertigion.com/foos?foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar''%22&complex%3
B%2F%3F%3A%40%26%3D%2B%24%2C%20foo''%22=bar2&foo1=bar1'
$url_parts_regex = '^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?' # 参见备注

if ($url -match $url_parts_regex) {
    $url_parts = @{
        'Scheme' = $Matches[2];
        'Server' = $Matches[4];
        'Path' = $Matches[5];
        'QueryString' = $Matches[7];
        'QueryStringParts' = @{}
    }

    foreach ($qs in $query_string.Split('&')) {
        $qs_key, $qs_value = $qs.Split('=')
        $url_parts.QueryStringParts.Add(
            [System.Web.HttpUtility]::UrlDecode($qs_key),
            [System.Web.HttpUtility]::UrlDecode($qs_value)
        ) | Out-Null
    }
} else {
    Throw [System.Management.Automation.ParameterBindingException] "Invalid URL Supplied"
}
```

这将返回给你 [hashtable]$url_parts；其等于（注意：复杂部分中的空格即为空格）：

```
PS > $url_parts

名称                          值
----                          -----
方案                          https
路径                          /foos
服务器                        example.vertigion.com
查询字符串
 foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar'%22&complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20foo'%
22=bar2&foo1=bar1
查询字符串部分                {foo2, complex;/?:@&=+$, foo'', foo1}


PS > $url_parts.QueryStringParts

名称                          值
----                          -----
foo2                          complex;/?:@&=+$, bar''
complex;/?:@&=+$, foo''       bar2
foo1                          bar1
```

使用 [System.Web.HttpUtility]::UrlEncode()编码现在，我们将解

码上面示例中使用 [System.Web.HttpUtility]::UrlEncode()编码的URL和查询字符串：

> https://example.vertigion.com/foos?

> https://example.vertigion.com/foos?
> foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar'%22&
> complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20foo'%22=bar2&foo1=bar1

```
$url =
'https://example.vertigion.com/foos?foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar''%22&complex%3
B%2F%3F%3A%40%26%3D%2B%24%2C%20foo''%22=bar2&foo1=bar1'
$url_parts_regex = '^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?' # See Remarks

if ($url -match $url_parts_regex) {
    $url_parts = @{
        'Scheme' = $Matches[2];
        'Server' = $Matches[4];
        'Path' = $Matches[5];
        'QueryString' = $Matches[7];
        'QueryStringParts' = @{}
    }

    foreach ($qs in $query_string.Split('&')) {
        $qs_key, $qs_value = $qs.Split('=')
        $url_parts.QueryStringParts.Add(
            [System.Web.HttpUtility]::UrlDecode($qs_key),
            [System.Web.HttpUtility]::UrlDecode($qs_value)
        ) | Out-Null
    }
} else {
    Throw [System.Management.Automation.ParameterBindingException] "Invalid URL Supplied"
}
```

This gives you back `[hashtable]$url_parts`; which equals (**Note:** the *spaces* in the complex parts are *spaces*):

```
PS > $url_parts

Name                           Value
----                           -----
Scheme                         https
Path                           /foos
Server                         example.vertigion.com
QueryString
 foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar'%22&complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20foo'%
22=bar2&foo1=bar1
QueryStringParts               {foo2, complex;/?:@&=+$, foo'', foo1}


PS > $url_parts.QueryStringParts

Name                           Value
----                           -----
foo2                           complex;/?:@&=+$, bar''
complex;/?:@&=+$, foo''        bar2
foo1                           bar1
```

Encoded with `[System.Web.HttpUtility]::UrlEncode()`

Now, we'll decode the URL and Query String encoded with `[System.Web.HttpUtility]::UrlEncode()` in the above example:

> https://example.vertigion.com/foos?

> foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&
> complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+foo%27%22=bar2&foo1=bar1

```
$url =
'https://example.vertigion.com/foos?foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&complex%3b
%2f%3f%3a%40%26%3d%2b%24%2c+foo%27%22=bar2&foo1=bar1'
$url_parts_regex = '^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?' # 参见备注

if ($url -match $url_parts_regex) {
    $url_parts = @{
        'Scheme' = $Matches[2];
        'Server' = $Matches[4];
        'Path' = $Matches[5];
        'QueryString' = $Matches[7];
        'QueryStringParts' = @{}
    }

    foreach ($qs in $query_string.Split('&')) {
        $qs_key, $qs_value = $qs.Split('=')
        $url_parts.QueryStringParts.Add(
            [System.Web.HttpUtility]::UrlDecode($qs_key),
            [System.Web.HttpUtility]::UrlDecode($qs_value)
        ) | Out-Null
    }
} else {
    Throw [System.Management.Automation.ParameterBindingException] "Invalid URL Supplied"
}
```

这将返回给你 [hashtable]$url_parts；其等于（注意：复杂部分中的空格即为空格）：

```
PS > $url_parts

名称                          值
----                          -----
方案                          https
路径                          /foos
服务器                        example.vertigion.com
查询字符串
 foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+foo%27%
22=bar2&foo1=bar1
查询字符串部分                {foo2, complex;/?:@&=+$, foo'", foo1}


PS > $url_parts.QueryStringParts

名称                          值
----                          -----
foo2                          complex;/?:@&=+$, bar'"
complex;/?:@&=+$, foo'"       bar2
foo1                          bar1
```

> foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&
> complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+foo%27%22=bar2&foo1=bar1

```
$url =
'https://example.vertigion.com/foos?foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&complex%3b
%2f%3f%3a%40%26%3d%2b%24%2c+foo%27%22=bar2&foo1=bar1'
$url_parts_regex = '^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?' # See Remarks

if ($url -match $url_parts_regex) {
    $url_parts = @{
        'Scheme' = $Matches[2];
        'Server' = $Matches[4];
        'Path' = $Matches[5];
        'QueryString' = $Matches[7];
        'QueryStringParts' = @{}
    }

    foreach ($qs in $query_string.Split('&')) {
        $qs_key, $qs_value = $qs.Split('=')
        $url_parts.QueryStringParts.Add(
            [System.Web.HttpUtility]::UrlDecode($qs_key),
            [System.Web.HttpUtility]::UrlDecode($qs_value)
        ) | Out-Null
    }
} else {
    Throw [System.Management.Automation.ParameterBindingException] "Invalid URL Supplied"
}
```

This gives you back [hashtable]$url_parts; which equals (**Note:** the *spaces* in the complex parts are *spaces*):

```
PS > $url_parts

Name                           Value
----                           -----
Scheme                         https
Path                           /foos
Server                         example.vertigion.com
QueryString
 foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+foo%27%
22=bar2&foo1=bar1
QueryStringParts               {foo2, complex;/?:@&=+$, foo'", foo1}


PS > $url_parts.QueryStringParts

Name                           Value
----                           -----
foo2                           complex;/?:@&=+$, bar'"
complex;/?:@&=+$, foo'"        bar2
foo1                           bar1
```

第47章：错误处理

本主题讨论PowerShell中的错误类型及错误处理。

第47.1节：错误类型

错误就是错误，可能有人会想怎么会有类型之分呢。实际上，在PowerShell中，错误大致分为两类，

- 终止错误
- 非终止错误

顾名思义，终止错误会终止执行，而非终止错误则允许执行继续到下一条语句。

> 这在假设$ErrorActionPreference值为默认（Continue）的情况下成立。$ErrorActionPreference是一个首选项变量，用于告诉PowerShell在遇到“非终止”错误时该如何处理。

终止错误

终止错误可以用典型的try catch来处理，如下所示

```
Try
{
    Write-Host "尝试除以零"
    1/0
}
捕获
{
    Write-Host "捕获到终止错误：除以零！"
}
```

上述代码片段将执行，错误会通过catch块被捕获。

非终止错误

另一方面，非终止错误默认不会被catch块捕获。原因是非终止错误不被视为严重错误。

```
Try
{
    Stop-Process -Id 123456
}
捕获
{
    Write-Host "非终止错误：无效的进程ID"
}
```

如果执行上述命令，你不会从catch块得到输出，因为该错误不被视为严重，执行会直接从下一条命令继续。然而，错误会显示在控制台中。要处理非终止错误，只需更改错误偏好设置。

```
Try
{
```

Chapter 47: Error handling

This topic discuss about Error Types & Error Handling in PowerShell.

Section 47.1: Error Types

An error is an error, one might wonder how could there be types in it. Well, with PowerShell the error broadly falls into two criteria,

- Terminating error
- Non-Terminating error

As the name says Terminating errors will terminate the execution and a Non-Terminating Errors let the execution continue to next statement.

> This is true assuming that **$ErrorActionPreference** value is default (Continue). $ErrorActionPreference is a Preference Variable which tells PowerShell what to do in case of an "Non-Terminating" error.

Terminating error

A terminating error can be handled with a typical try catch, as below

```
Try
{
    Write-Host "Attempting Divide By Zero"
    1/0
}
Catch
{
    Write-Host "A Terminating Error: Divide by Zero Caught!"
}
```

The above snippet will execute and the error will be caught thru the catch block.

Non-Terminating Error

A Non-Terminating error in the other hand will not be caught in the catch block by default. The reason behind that is a Non-Terminating error is not considered a critical error.

```
Try
{
    Stop-Process -Id 123456
}
Catch
{
    Write-Host "Non-Terminating Error: Invalid Process ID"
}
```

If you execute the above the line you won't get the output from catch block as since the error is not considered critical and the execution will simply continue from next command. However, the error will be displayed in the console. To handle a Non-Terminating error, you simple have to change the error preference.

```
Try
{
    Stop-Process -Id 123456 -ErrorAction Stop
}
捕获
{
    "非终止错误：无效的进程ID"
}
```

现在，随着错误偏好的更新，此错误将被视为终止错误，并将在

catch 块中被捕获。

调用终止和非终止错误：

Write-Error cmdlet 仅将错误写入调用主机程序。它不会停止执行。而
throw 会产生终止错误并停止执行。

```
Write-host "准备尝试一个非终止错误"
Write-Error "非终止"
Write-host "准备尝试一个终止错误"
throw "终止错误"
Write-host "这行不会被显示"
```

```
    Stop-Process -Id 123456 -ErrorAction Stop
}
Catch
{
    "Non-Terminating Error: Invalid Process ID"
}
```

Now, with the updated Error preference, this error will be considered a Terminating error and will be caught in the catch block.

Invoking Terminating & Non-Terminating Errors:

Write-Error cmdlet simply writes the error to the invoking host program. It doesn't stop the execution. Where as **throw** will give you a terminating error and stop the execution

```
Write-host "Going to try a non terminating Error "
Write-Error "Non terminating"
Write-host "Going to try a terminating Error "
throw "Terminating Error "
Write-host "This Line won't be displayed"
```

第48章：包管理

PowerShell 包管理允许你查找、安装、更新和卸载 PowerShell 模块及其他包。

PowerShellGallery.com 是 PowerShell 模块的默认源。你也可以浏览该网站以查找可用的包、命令并预览代码。

第48.1节：创建默认的PowerShell模块存储库

如果由于某种原因，默认的 PowerShell 模块仓库PSGallery被移除。你需要重新创建它。这是命令。

```
Register-PSRepository -Default
```

第48.2节：按名称查找模块

```
Find-Module -Name <Name>
```

第48.3节：按名称安装模块

```
Install-Module -Name <name>
```

第48.4节：按名称和版本卸载模块

```
Uninstall-Module -Name <Name> -RequiredVersion <Version>
```

第48.5节：按名称更新模块

```
Update-Module -Name <Name>
```

第48.6节：使用模式查找PowerShell模块

查找以DSC结尾的模块

```
Find-Module -Name *DSC
```

Chapter 48: Package management

PowerShell Package Management allows you to find, install, update and uninstall PowerShell Modules and other packages.

PowerShellGallery.com is the default source for PowerShell modules. You can also browse the site for available packages, command and preview the code.

Section 48.1: Create the default PowerShell Module Repository

If for some reason, the default PowerShell module repository `PSGallery` gets removed. You will need to create it. This is the command.

```
Register-PSRepository -Default
```

Section 48.2: Find a module by name

```
Find-Module -Name <Name>
```

Section 48.3: Install a Module by name

```
Install-Module -Name <name>
```

Section 48.4: Uninstall a module my name and version

```
Uninstall-Module -Name <Name> -RequiredVersion <Version>
```

Section 48.5: Update a module by name

```
Update-Module -Name <Name>
```

Section 48.6: Find a PowerShell module using a pattern

To find a module that ends with `DSC`

```
Find-Module -Name *DSC
```

第49章：使用

PowerShell进行TCP通信

第49.1节：TCP监听器

```
函数 Receive-TCPMessage {
    参数 (
        [Parameter(必需=$true, 位置=0)]
        [ValidateNotNullOrEmpty()]
        [int] $Port
    )
    Process {
        尝试 {
            # 设置端点并开始监听
            $endpoint = new-object System.Net.IPEndPoint([ipaddress]::any,$port)
            $listener = new-object System.Net.Sockets.TcpListener $EndPoint
            $listener.start()

            # 等待传入连接
            $data = $listener.AcceptTcpClient()

            # 流设置
            $stream = $data.GetStream()
            $bytes = New-Object System.Byte[] 1024

            # 从流中读取数据并写入主机
            while (($i = $stream.Read($bytes,0,$bytes.Length)) -ne 0){
                $EncodedText = New-Object System.Text.ASCIIEncoding
                $data = $EncodedText.GetString($bytes,0, $i)
                Write-Output $data
            }

            # 关闭TCP连接并停止监听
            $stream.close()
            $listener.stop()
        }
        Catch {
            "接收消息失败，错误信息：`n" + $Error[0]
        }
    }
}
```

开始监听以下内容，并将任何消息捕获到变量 $msg 中：

```
$msg = Receive-TCPMessage -Port 29800
```

第49.2节：TCP发送方

```
函数 Send-TCPMessage {
    参数 (
            [Parameter(必需=$true, 位置=0)]
            [ValidateNotNullOrEmpty()]
            [字符串]
            $EndPoint
        ,
            [参数(必需=$true, 位置=1)]
            [int]
```

Chapter 49: TCP Communication with PowerShell

Section 49.1: TCP listener

```
Function Receive-TCPMessage {
    Param (
        [Parameter(Mandatory=$true, Position=0)]
        [ValidateNotNullOrEmpty()]
        [int] $Port
    )
    Process {
        Try {
            # Set up endpoint and start listening
            $endpoint = new-object System.Net.IPEndPoint([ipaddress]::any,$port)
            $listener = new-object System.Net.Sockets.TcpListener $EndPoint
            $listener.start()

            # Wait for an incoming connection
            $data = $listener.AcceptTcpClient()

            # Stream setup
            $stream = $data.GetStream()
            $bytes = New-Object System.Byte[] 1024

            # Read data from stream and write it to host
            while (($i = $stream.Read($bytes,0,$bytes.Length)) -ne 0){
                $EncodedText = New-Object System.Text.ASCIIEncoding
                $data = $EncodedText.GetString($bytes,0, $i)
                Write-Output $data
            }

            # Close TCP connection and stop listening
            $stream.close()
            $listener.stop()
        }
        Catch {
            "Receive Message failed with: `n" + $Error[0]
        }
    }
}
```

Start listening with the following and capture any message in the variable $msg:

```
$msg = Receive-TCPMessage -Port 29800
```

Section 49.2: TCP Sender

```
Function Send-TCPMessage {
    Param (
            [Parameter(Mandatory=$true, Position=0)]
            [ValidateNotNullOrEmpty()]
            [string]
            $EndPoint
        ,
            [Parameter(Mandatory=$true, Position=1)]
            [int]
            $Port
        ,
            [参数(必需=$true, 位置=2)]
            [string]
            $Message
    )
    Process {
        # 设置连接
        $IP = [System.Net.Dns]::GetHostAddresses($EndPoint)
        $Address = [System.Net.IPAddress]::Parse($IP)
        $Socket = New-Object System.Net.Sockets.TCPClient($Address,$Port)

        # 设置流写入器
        $Stream = $Socket.GetStream()
        $Writer = New-Object System.IO.StreamWriter($Stream)

        # 向流中写入消息
        $Message | % {
            $Writer.WriteLine($_)
            $Writer.Flush()
        }

        # 关闭连接和流
        $Stream.Close()
        $Socket.Close()
    }
}
```

发送消息命令：

```
Send-TCPMessage -Port 29800 -Endpoint 192.168.0.1 -message "My first TCP message !"
```

注意：TCP消息可能会被您的软件防火墙或任何外部防火墙阻止，您尝试穿过这些防火墙。请确保您在上述命令中设置的TCP端口已打开，并且您已在同一端口上设置了监听器。

```
            $Port
        ,
            [Parameter(Mandatory=$true, Position=2)]
            [string]
            $Message
    )
    Process {
        # Setup connection
        $IP = [System.Net.Dns]::GetHostAddresses($EndPoint)
        $Address = [System.Net.IPAddress]::Parse($IP)
        $Socket = New-Object System.Net.Sockets.TCPClient($Address,$Port)

        # Setup stream writer
        $Stream = $Socket.GetStream()
        $Writer = New-Object System.IO.StreamWriter($Stream)

        # Write message to stream
        $Message | % {
            $Writer.WriteLine($_)
            $Writer.Flush()
        }

        # Close connection and stream
        $Stream.Close()
        $Socket.Close()
    }
}
```

Send a message with:

```
Send-TCPMessage -Port 29800 -Endpoint 192.168.0.1 -message "My first TCP message !"
```

Note: TCP messages may be blocked by your software firewall or any external facing firewalls you are trying to go through. Ensure that the TCP port you set in the above command is open and that you are have setup the listener on the same port.

第50章：PowerShell工作流

PowerShell 工作流是从 PowerShell 版本 3.0 开始引入的一个功能。工作流定义看起来与 PowerShell 函数定义非常相似，但它们在 Windows Workflow Foundation 环境中执行，而不是直接在 PowerShell 引擎中执行。

工作流引擎包含多个独特的“开箱即用”功能，最显著的是作业持久化。

第 50.1 节：带输入参数的工作流

就像 PowerShell 函数一样，工作流可以接受输入参数。输入参数可以选择绑定到特定的数据类型，如字符串、整数等。使用标准的 param 关键字在工作流声明后直接定义一组输入参数块。

```
workflow DoSomeWork {
  param (
    [string[]] $ComputerName
  )
  Get-Process -ComputerName $ComputerName
}

DoSomeWork -ComputerName server01, server02, server03
```

第 50.2 节：简单工作流示例

```
工作流 DoSomeWork {
  Get-Process -Name notepad | Stop-Process
}
```

这是一个 PowerShell 工作流定义的基本示例。

第50.3节：以后台作业运行工作流

PowerShell 工作流本身具备作为后台作业运行的能力。要将工作流作为 PowerShell 后台作业调用，请在调用工作流时使用 `-AsJob` 参数。

```
workflow DoSomeWork {
  Get-Process -ComputerName server01
  Get-Process -ComputerName server02
  Get-Process -ComputerName server03
}

DoSomeWork -AsJob
```

第50.4节：向工作流添加并行块

```
workflow DoSomeWork {
  parallel {
    Get-Process -ComputerName server01
    Get-Process -ComputerName server02
    Get-Process -ComputerName server03
  }
}
```

Chapter 50: PowerShell Workflows

PowerShell Workflow is a feature that was introduced starting with PowerShell version 3.0. Workflow definitions look very similar to PowerShell function definitions, however they execute within the Windows Workflow Foundation environment, instead of directly in the PowerShell engine.

Several unique "out of box" features are included with the Workflow engine, most notably, job persistence.

Section 50.1: Workflow with Input Parameters

Just like PowerShell functions, workflows can accept input parameter. Input parameters can optionally be bound to a specific data type, such as a string, integer, etc. Use the standard `param` keyword to define a block of input parameters, directly after the workflow declaration.

```
workflow DoSomeWork {
  param (
    [string[]] $ComputerName
  )
  Get-Process -ComputerName $ComputerName
}

DoSomeWork -ComputerName server01, server02, server03
```

Section 50.2: Simple Workflow Example

```
workflow DoSomeWork {
  Get-Process -Name notepad | Stop-Process
}
```

This is a basic example of a PowerShell Workflow definition.

Section 50.3: Run Workflow as a Background Job

PowerShell Workflows are inherently equipped with the ability to run as a background job. To call a workflow as a PowerShell background job, use the `-AsJob` parameter when invoking the workflow.

```
workflow DoSomeWork {
  Get-Process -ComputerName server01
  Get-Process -ComputerName server02
  Get-Process -ComputerName server03
}

DoSomeWork -AsJob
```

Section 50.4: Add a Parallel Block to a Workflow

```
workflow DoSomeWork {
  parallel {
    Get-Process -ComputerName server01
    Get-Process -ComputerName server02
    Get-Process -ComputerName server03
  }
}
```

PowerShell 工作流的一个独特功能是能够将一组活动定义为并行。要使用此功能，请在工作流中使用`parallel`关键字。

并行调用工作流活动可能有助于提高工作流的性能。

One of the unique features of PowerShell Workflow is the ability to define a block of activities as parallel. To use this feature, use the `parallel` keyword inside your Workflow.

Calling workflow activities in parallel may help to improve performance of your workflow.

第51章：嵌入托管代码（C# | VB）

参数	详情
-TypeDefinition<String_>	接受作为字符串的代码
-Language<String_>	指定托管代码语言。可接受的值：CSharp、CSharpVersion3、CSharpVersion2、VisualBasic、JScript

本主题简要介绍如何在 PowerShell 脚本中编写和使用 C# 或 VB .NET 托管代码。本主题不涉及 Add-Type cmdlet 的所有方面。

有关 Add-Type cmdlet 的更多信息，请参阅 MSDN 文档（针对 5.1 版本），链接如下：
https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.utility/add-type

第51.1节：C# 示例

此示例展示了如何将一些基本的C#代码嵌入到PowerShell脚本中，添加到运行空间/会话中，并在PowerShell语法内使用这些代码。

```
$code =
using System;

namespace MyNameSpace
{
public class Responder
    {
public static void StaticRespond()
        {
Console.WriteLine("Static Response");
        }

public void Respond()
        {
Console.WriteLine("Instance Respond");
        }
    }
}
"@

# 检查该类型是否已在会话中添加，否则会引发异常
if (-not ([System.Management.Automation.PSTypeName]'MyNameSpace.Responder').Type)
{
    Add-Type -TypeDefinition $code -Language CSharp;
}

[MyNameSpace.Responder]::StaticRespond();

$instance = New-Object MyNameSpace.Responder;
$instance.Respond();
```

第51.2节：VB.NET示例

此示例展示了如何将一些基本的C#代码嵌入到PowerShell脚本中，添加到运行空间/会话中，并在PowerShell语法内使用这些代码。

```
$code = @"
```

Chapter 51: Embedding Managed Code (C# | VB)

Parameter	Details
-TypeDefinition<String_>	Accepts the code as a string
-Language<String_>	Specifies the Managed Code language.Accepted values: CSharp, CSharpVersion3, CSharpVersion2, VisualBasic, JScript

This topic is to briefly describe how C# or VB .NET Managed code can be scripted and utilised within a PowerShell script. This topic is not exploring all facets of the Add-Type cmdlet.

For more information on the Add-Type cmdlet, please refer to the MSDN documentation (for 5.1) here:
https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.utility/add-type

Section 51.1: C# Example

This example shows how to embed some basic C# into a PowerShell script, add it to the runspace/session and utilise the code within PowerShell syntax.

```
$code = "
using System;

namespace MyNameSpace
{
    public class Responder
    {
        public static void StaticRespond()
        {
            Console.WriteLine("Static Response");
        }

        public void Respond()
        {
            Console.WriteLine("Instance Respond");
        }
    }
}
"@

# Check the type has not been previously added within the session, otherwise an exception is raised
if (-not ([System.Management.Automation.PSTypeName]'MyNameSpace.Responder').Type)
{
    Add-Type -TypeDefinition $code -Language CSharp;
}

[MyNameSpace.Responder]::StaticRespond();

$instance = New-Object MyNameSpace.Responder;
$instance.Respond();
```

Section 51.2: VB.NET Example

This example shows how to embed some basic C# into a PowerShell script, add it to the runspace/session and utilise the code within PowerShell syntax.

```
$code = @"
```

```
导入 System

命名空间 MyNameSpace
公共类 Responder
        公共共享子程序 StaticRespond()
            Console.WriteLine("静态响应")
        结束子程序

公共子程序 Respond()
            Console.WriteLine("实例响应")
        结束子程序
结束类
结束命名空间
"@

# 检查该类型是否已在会话中添加，否则会引发异常
if (-not ([System.Management.Automation.PSTypeName]'MyNameSpace.Responder').Type)
{
    Add-Type -TypeDefinition $code -Language VisualBasic;
}

[MyNameSpace.Responder]::StaticRespond();

$instance = New-Object MyNameSpace.Responder;
$instance.Respond();
```

```
Imports System

Namespace MyNameSpace
    Public Class Responder
        Public Shared Sub StaticRespond()
            Console.WriteLine("Static Response")
        End Sub

        Public Sub Respond()
            Console.WriteLine("Instance Respond")
        End Sub
    End Class
End Namespace
"@

# Check the type has not been previously added within the session, otherwise an exception is raised
if (-not ([System.Management.Automation.PSTypeName]'MyNameSpace.Responder').Type)
{
    Add-Type -TypeDefinition $code -Language VisualBasic;
}

[MyNameSpace.Responder]::StaticRespond();

$instance = New-Object MyNameSpace.Responder;
$instance.Respond();
```

第52章：如何使用PowerShell脚本（v 2.0或以下版本）从Artifactory下载最新的构件？

本说明文档解释并提供了使用PowerShell脚本（v2.0或以下版本）从JFrog Artifactory仓库下载最新构件的步骤。

第52.1节：用于下载最新构件的PowerShell脚本

```
$username = 'user'
$password= 'password'
$DESTINATION = "D:est\latest.tar.gz"
$client = New-Object System.Net.WebClient
$client.Credentials = new-object System.Net.NetworkCredential($username, $password)
$lastModifiedResponse =
$client.DownloadString('https://domain.org.com/artifactory/api/storage/FOLDER/repo/?lastModified')
[System.Reflection.Assembly]::LoadWithPartialName("System.Web.Extensions")
$serializer = New-Object System.Web.Script.Serialization.JavaScriptSerializer
$getLatestModifiedResponse = $serializer.DeserializeObject($lastModifiedResponse)
$downloadUriResponse = $getLatestModifiedResponse.uri
Write-Host $json.uri
$latestArtifcatUrlResponse=$client.DownloadString($downloadUriResponse)
[System.Reflection.Assembly]::LoadWithPartialName("System.Web.Extensions")
$serializer = New-Object System.Web.Script.Serialization.JavaScriptSerializer
$getLatestArtifact = $serializer.DeserializeObject($latestArtifcatUrlResponse)
Write-Host $getLatestArtifact.downloadUri
$SOURCE=$getLatestArtifact.downloadUri
$client.DownloadFile($SOURCE,$DESTINATION)
```

Chapter 52: How to download latest artifact from Artifactory using PowerShell script (v2.0 or below)?

This documentation explains and provides steps to download latest artifact from a JFrog Artifactory repository using PowerShell Script (v2.0 or below).

Section 52.1: PowerShell Script for downloading the latest artifact

```
$username = 'user'
$password= 'password'
$DESTINATION = "D:\test\latest.tar.gz"
$client = New-Object System.Net.WebClient
$client.Credentials = new-object System.Net.NetworkCredential($username, $password)
$lastModifiedResponse =
$client.DownloadString('https://domain.org.com/artifactory/api/storage/FOLDER/repo/?lastModified')
[System.Reflection.Assembly]::LoadWithPartialName("System.Web.Extensions")
$serializer = New-Object System.Web.Script.Serialization.JavaScriptSerializer
$getLatestModifiedResponse = $serializer.DeserializeObject($lastModifiedResponse)
$downloadUriResponse = $getLatestModifiedResponse.uri
Write-Host $json.uri
$latestArtifcatUrlResponse=$client.DownloadString($downloadUriResponse)
[System.Reflection.Assembly]::LoadWithPartialName("System.Web.Extensions")
$serializer = New-Object System.Web.Script.Serialization.JavaScriptSerializer
$getLatestArtifact = $serializer.DeserializeObject($latestArtifcatUrlResponse)
Write-Host $getLatestArtifact.downloadUri
$SOURCE=$getLatestArtifact.downloadUri
$client.DownloadFile($SOURCE,$DESTINATION)
```

第53章：基于注释的帮助

PowerShell具有一种称为基于注释的帮助的文档机制。它允许使用代码注释来记录脚本和函数。基于注释的帮助大多数情况下写在包含多个帮助关键字的注释块中。帮助关键字以点开头，标识通过运行Get-Help命令显示的帮助部分。

第53.1节：函数基于注释的帮助

```
<#

.SYNOPSIS
获取INI文件的内容。

.DESCRIPTION
获取INI文件的内容并以哈希表形式返回。

.INPUTS
System.String

.OUTPUTS
System.Collections.Hashtable

.PARAMETER FilePath
指定输入INI文件的路径。

.EXAMPLE
C:\PS>$IniContent = Get-IniContent -FilePath file.ini
    C:\PS>$IniContent['Section1'].Key1
获取file.ini的内容并访问Section1中的Key1。

.LINK
Out-IniFile

#>
function Get-IniContent
{
    [CmdletBinding()]
    Param
    (
        [Parameter(Mandatory=$true,ValueFromPipeline=$true)]
        [ValidateNotNullOrEmpty()]
        [ValidateScript({(Test-Path $_) -and ((Get-Item $_).Extension -eq ".ini")})]
        [System.String]$FilePath
    )

    # 初始化输出哈希表。
    $ini = @{}
    switch -regex -file $FilePath
    {
        "^\[(.+)\]$" # 节
        {
            $section = $matches[1]
            $ini[$section] = @{}
            $CommentCount = 0
        }
        "^(;.*)$" # 注释
        {
            if( !($section) )
```

Chapter 53: Comment-based help

PowerShell features a documentation mechanism called comment-based help. It allows documenting scripts and functions with code comments. Comment-based help is most of the time written in comment blocks containing multiple help keywords. Help keywords start with dots and identify help sections that will be displayed by running the `Get-Help` cmdlet.

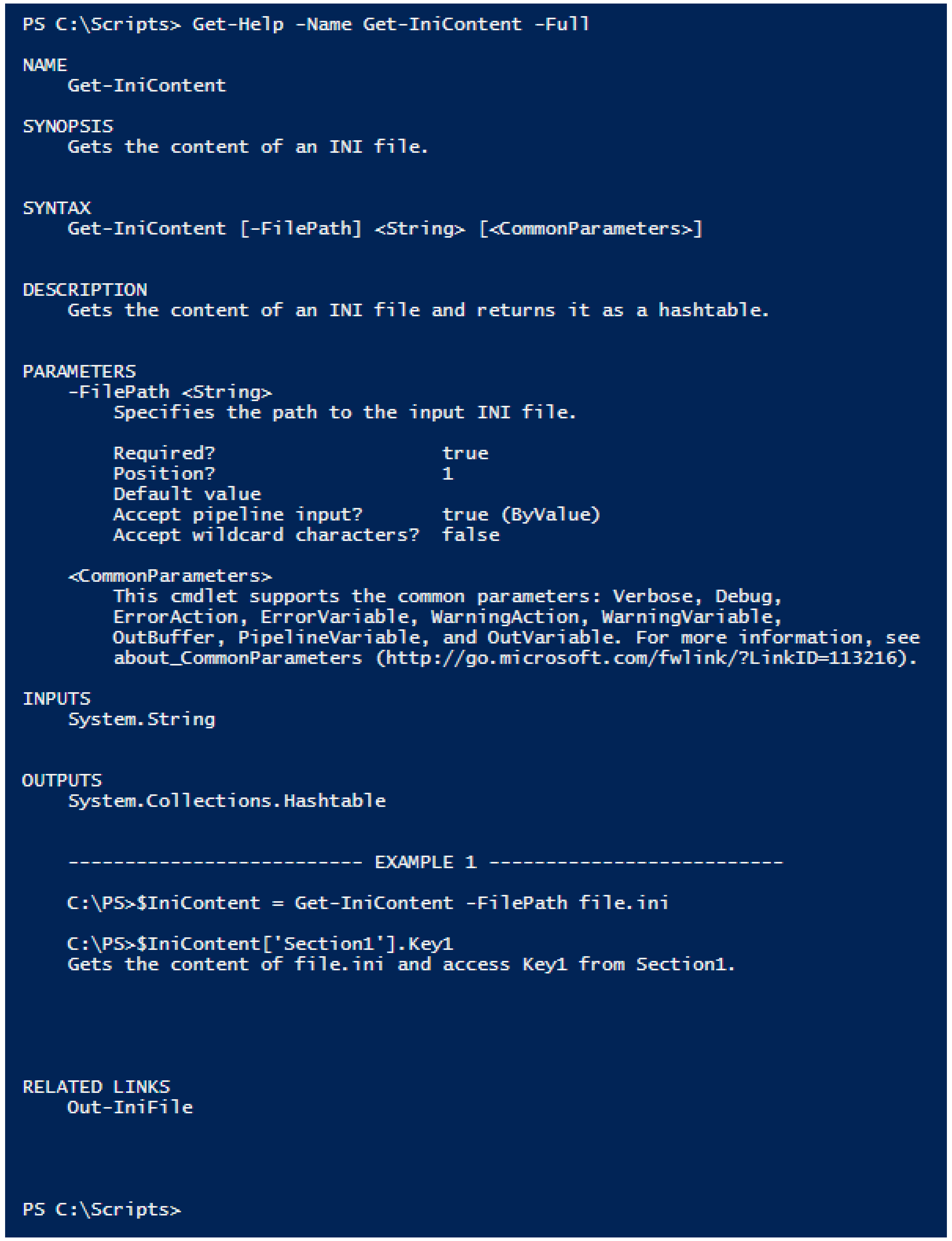
Section 53.1: Function comment-based help

```
<#

.SYNOPSIS
    Gets the content of an INI file.

.DESCRIPTION
    Gets the content of an INI file and returns it as a hashtable.

.INPUTS
    System.String

.OUTPUTS
    System.Collections.Hashtable

.PARAMETER FilePath
    Specifies the path to the input INI file.

.EXAMPLE
    C:\PS>$IniContent = Get-IniContent -FilePath file.ini
    C:\PS>$IniContent['Section1'].Key1
    Gets the content of file.ini and access Key1 from Section1.

.LINK
    Out-IniFile

#>
function Get-IniContent
{
    [CmdletBinding()]
    Param
    (
        [Parameter(Mandatory=$true,ValueFromPipeline=$true)]
        [ValidateNotNullOrEmpty()]
        [ValidateScript({(Test-Path $_) -and ((Get-Item $_).Extension -eq ".ini")})]
        [System.String]$FilePath
    )

    # Initialize output hash table.
    $ini = @{}
    switch -regex -file $FilePath
    {
        "^\[(.+)\]$" # Section
        {
            $section = $matches[1]
            $ini[$section] = @{}
            $CommentCount = 0
        }
        "^(;.*)$" # Comment
        {
            if( !($section) )
            {
                $section = "无节"
                $ini[$section] = @{}
            }
            $value = $matches[1]
            $CommentCount = $CommentCount + 1
            $name = "Comment" + $CommentCount
            $ini[$section][$name] = $value
        }
        "(.+?)\s*=\s*(.*)" # 键
        {
            if( !($section) )
            {
                $section = "无节"
                $ini[$section] = @{}
            }
            $name,$value = $matches[1..2]
            $ini[$section][$name] = $value
        }
    }

    return $ini
}
```

上述函数文档可以通过运行 Get-Help -Name Get-IniContent -Full 来显示：

```
            {
                $section = "No-Section"
                $ini[$section] = @{}
            }
            $value = $matches[1]
            $CommentCount = $CommentCount + 1
            $name = "Comment" + $CommentCount
            $ini[$section][$name] = $value
        }
        "(.+?)\s*=\s*(.*)" # Key
        {
            if( !($section) )
            {
                $section = "No-Section"
                $ini[$section] = @{}
            }
            $name,$value = $matches[1..2]
            $ini[$section][$name] = $value
        }
    }

    return $ini
}
```

The above function documentation can be displayed by running `Get-Help -Name Get-IniContent -Full`:

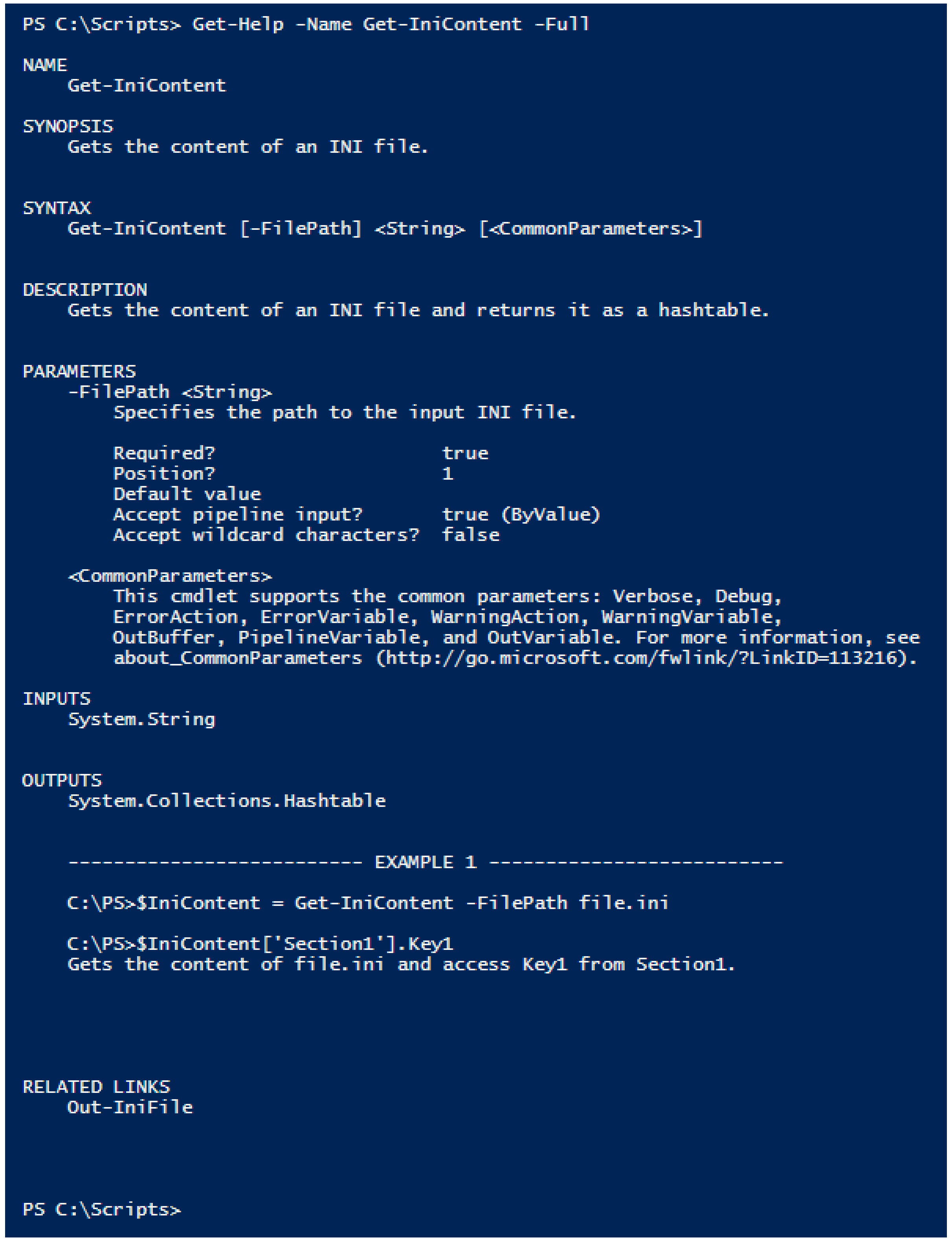


```
PS C:\Scripts> Get-Help -Name Get-IniContent -Full

NAME
    Get-IniContent

SYNOPSIS
    Gets the content of an INI file.


SYNTAX
    Get-IniContent [-FilePath] <String> [<CommonParameters>]


DESCRIPTION
    Gets the content of an INI file and returns it as a hashtable.


PARAMETERS
    -FilePath <String>
        Specifies the path to the input INI file.

        Required?                    true
        Position?                    1
        Default value
        Accept pipeline input?       true (ByValue)
        Accept wildcard characters?  false

    <CommonParameters>
        This cmdlet supports the common parameters: Verbose, Debug,
        ErrorAction, ErrorVariable, WarningAction, WarningVariable,
        OutBuffer, PipelineVariable, and OutVariable. For more information, see
        about_CommonParameters (http://go.microsoft.com/fwlink/?LinkID=113216).

INPUTS
    System.String


OUTPUTS
    System.Collections.Hashtable


    -------------------------- EXAMPLE 1 --------------------------

    C:\PS>$IniContent = Get-IniContent -FilePath file.ini

    C:\PS>$IniContent['Section1'].Key1
    Gets the content of file.ini and access Key1 from Section1.




RELATED LINKS
    Out-IniFile



PS C:\Scripts>
```

请注意，以 . 开头的基于注释的关键字与 Get-Help 结果部分相匹配。

第53.2节：脚本基于注释的帮助

```
<#

.SYNOPSIS
读取CSV文件并进行过滤。

.DESCRIPTION
ReadUsersCsv.ps1 脚本读取CSV文件并根据 'UserName' 列进行过滤。
```



```
PS C:\Scripts> Get-Help -Name Get-IniContent -Full

NAME
    Get-IniContent

SYNOPSIS
    Gets the content of an INI file.


SYNTAX
    Get-IniContent [-FilePath] <String> [<CommonParameters>]


DESCRIPTION
    Gets the content of an INI file and returns it as a hashtable.


PARAMETERS
    -FilePath <String>
        Specifies the path to the input INI file.

        Required?                    true
        Position?                    1
        Default value
        Accept pipeline input?       true (ByValue)
        Accept wildcard characters?  false

    <CommonParameters>
        This cmdlet supports the common parameters: Verbose, Debug,
        ErrorAction, ErrorVariable, WarningAction, WarningVariable,
        OutBuffer, PipelineVariable, and OutVariable. For more information, see
        about_CommonParameters (http://go.microsoft.com/fwlink/?LinkID=113216).

INPUTS
    System.String


OUTPUTS
    System.Collections.Hashtable


    -------------------------- EXAMPLE 1 --------------------------

    C:\PS>$IniContent = Get-IniContent -FilePath file.ini

    C:\PS>$IniContent['Section1'].Key1
    Gets the content of file.ini and access Key1 from Section1.




RELATED LINKS
    Out-IniFile



PS C:\Scripts>
```

Notice that the comment-based keywords starting with a . match the Get-Help result sections.

Section 53.2: Script comment-based help

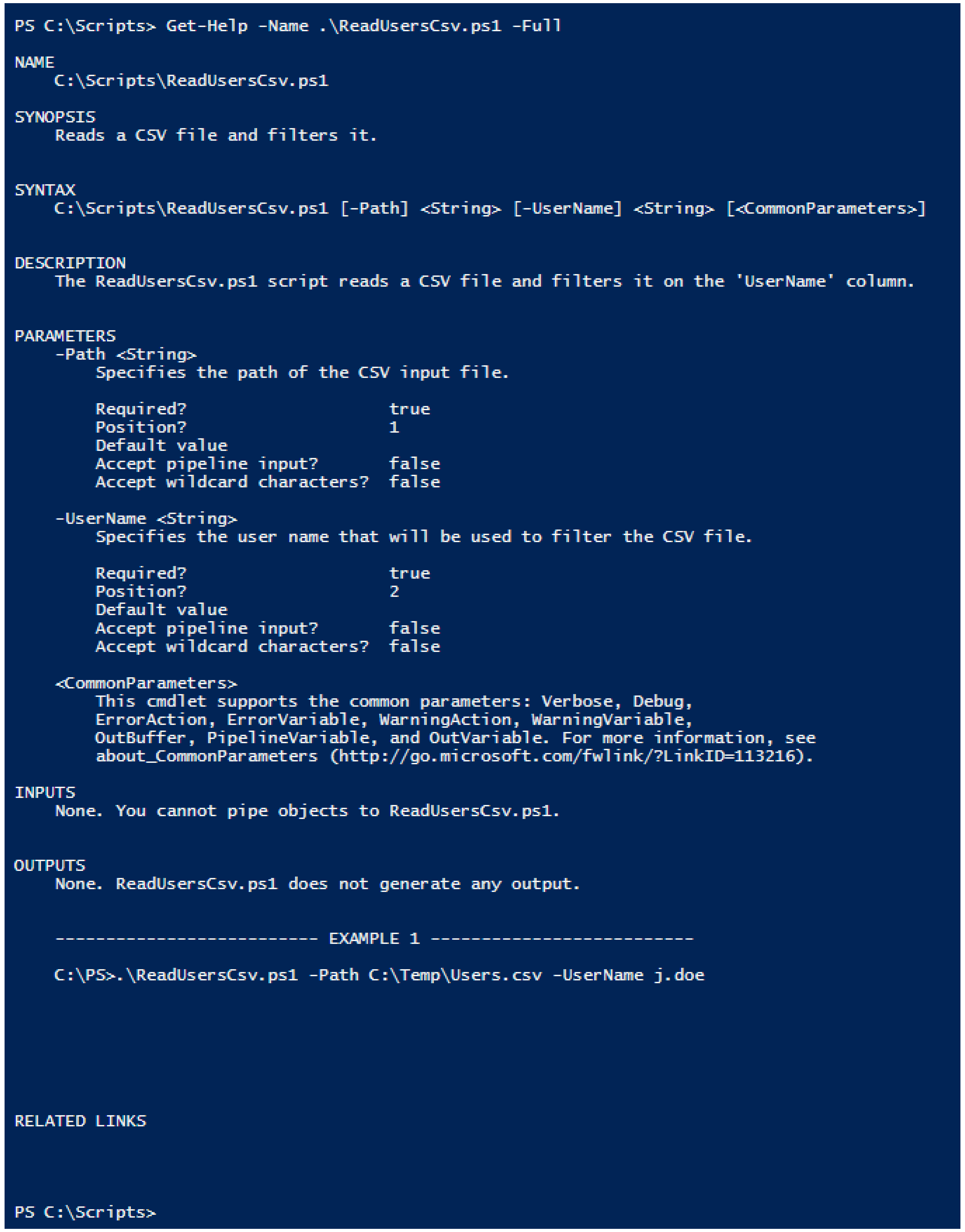
```
<#

.SYNOPSIS
    Reads a CSV file and filters it.

.DESCRIPTION
    The ReadUsersCsv.ps1 script reads a CSV file and filters it on the 'UserName' column.
```

```
.PARAMETER Path
指定CSV输入文件的路径。

.INPUTS
无。不能将对象通过管道传递给ReadUsersCsv.ps1。

.OUTPUTS
无。ReadUsersCsv.ps1不生成任何输出。

.EXAMPLE
C:\PS> .\ReadUsersCsv.ps1 -Path C:\Temp\Users.csv -UserName j.doe

#>
参数
(
    [Parameter(Mandatory=$true,ValueFromPipeline=$false)]
    [System.String]
    $Path,
    [Parameter(Mandatory=$true,ValueFromPipeline=$false)]
    [System.String]
    $UserName
)

Import-Csv -Path $Path | Where-Object -FilterScript {$_.UserName -eq $UserName}
```

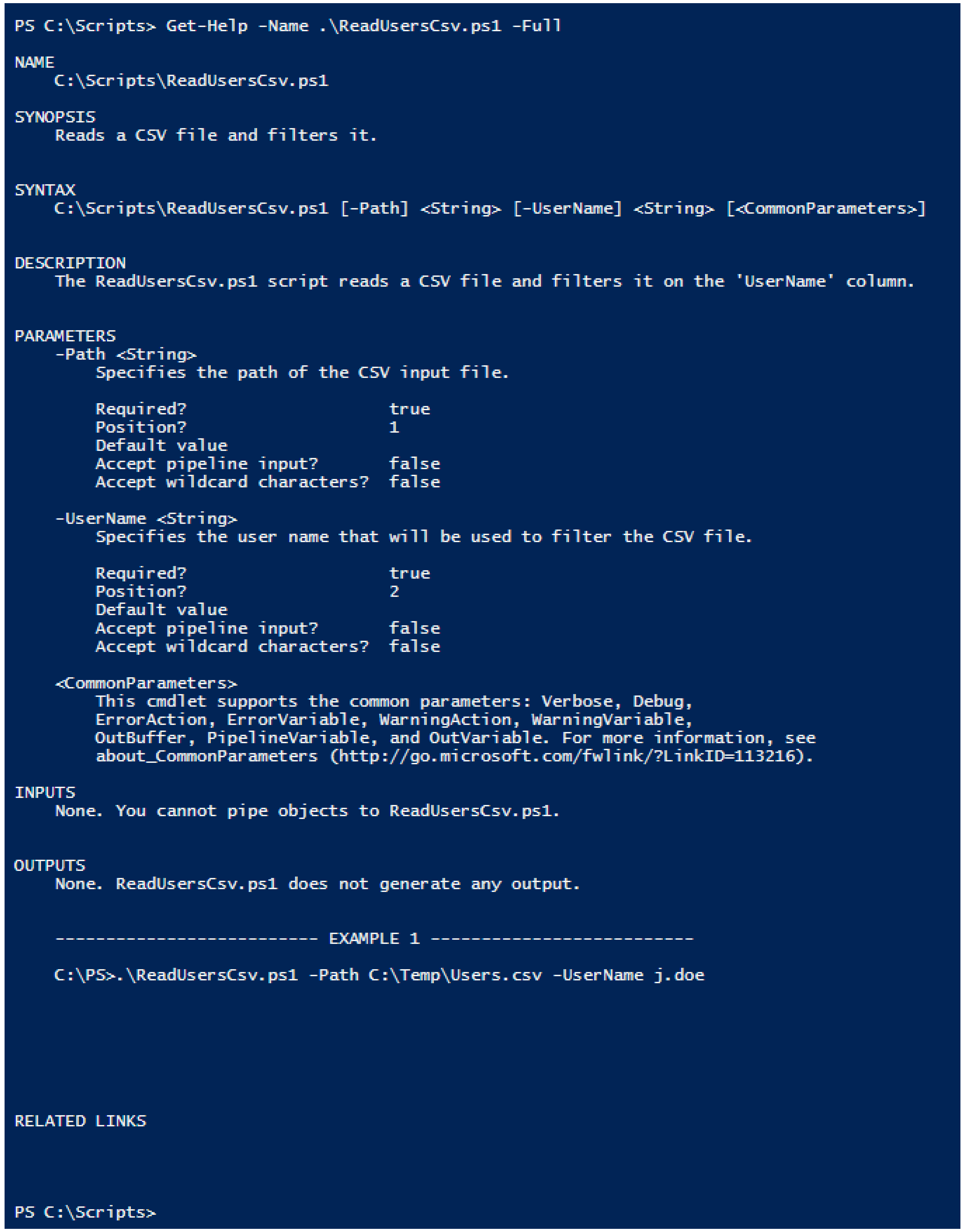
上述脚本文档可以通过运行 Get-Help -Name ReadUsersCsv.ps1 -Full 来显示：

```
.PARAMETER Path
    Specifies the path of the CSV input file.

.INPUTS
    None. You cannot pipe objects to ReadUsersCsv.ps1.

.OUTPUTS
    None. ReadUsersCsv.ps1 does not generate any output.

.EXAMPLE
    C:\PS> .\ReadUsersCsv.ps1 -Path C:\Temp\Users.csv -UserName j.doe

#>
Param
(
    [Parameter(Mandatory=$true,ValueFromPipeline=$false)]
    [System.String]
    $Path,
    [Parameter(Mandatory=$true,ValueFromPipeline=$false)]
    [System.String]
    $UserName
)

Import-Csv -Path $Path | Where-Object -FilterScript {$_.UserName -eq $UserName}
```

The above script documentation can be displayed by running `Get-Help -Name ReadUsersCsv.ps1 -Full`:



```
PS C:\Scripts> Get-Help -Name .\ReadUsersCsv.ps1 -Full

NAME
    C:\Scripts\ReadUsersCsv.ps1

SYNOPSIS
    Reads a CSV file and filters it.


SYNTAX
    C:\Scripts\ReadUsersCsv.ps1 [-Path] <String> [-UserName] <String> [<CommonParameters>]


DESCRIPTION
    The ReadUsersCsv.ps1 script reads a CSV file and filters it on the 'UserName' column.


PARAMETERS
    -Path <String>
        Specifies the path of the CSV input file.

        Required?                    true
        Position?                    1
        Default value
        Accept pipeline input?       false
        Accept wildcard characters?  false

    -UserName <String>
        Specifies the user name that will be used to filter the CSV file.

        Required?                    true
        Position?                    2
        Default value
        Accept pipeline input?       false
        Accept wildcard characters?  false

    <CommonParameters>
        This cmdlet supports the common parameters: Verbose, Debug,
        ErrorAction, ErrorVariable, WarningAction, WarningVariable,
        OutBuffer, PipelineVariable, and OutVariable. For more information, see
        about_CommonParameters (http://go.microsoft.com/fwlink/?LinkID=113216).

INPUTS
    None. You cannot pipe objects to ReadUsersCsv.ps1.


OUTPUTS
    None. ReadUsersCsv.ps1 does not generate any output.


    -------------------------- EXAMPLE 1 --------------------------

    C:\PS>.\ReadUsersCsv.ps1 -Path C:\Temp\Users.csv -UserName j.doe






RELATED LINKS



PS C:\Scripts>
```



```
PS C:\Scripts> Get-Help -Name .\ReadUsersCsv.ps1 -Full

NAME
    C:\Scripts\ReadUsersCsv.ps1

SYNOPSIS
    Reads a CSV file and filters it.


SYNTAX
    C:\Scripts\ReadUsersCsv.ps1 [-Path] <String> [-UserName] <String> [<CommonParameters>]


DESCRIPTION
    The ReadUsersCsv.ps1 script reads a CSV file and filters it on the 'UserName' column.


PARAMETERS
    -Path <String>
        Specifies the path of the CSV input file.

        Required?                    true
        Position?                    1
        Default value
        Accept pipeline input?       false
        Accept wildcard characters?  false

    -UserName <String>
        Specifies the user name that will be used to filter the CSV file.

        Required?                    true
        Position?                    2
        Default value
        Accept pipeline input?       false
        Accept wildcard characters?  false

    <CommonParameters>
        This cmdlet supports the common parameters: Verbose, Debug,
        ErrorAction, ErrorVariable, WarningAction, WarningVariable,
        OutBuffer, PipelineVariable, and OutVariable. For more information, see
        about_CommonParameters (http://go.microsoft.com/fwlink/?LinkID=113216).

INPUTS
    None. You cannot pipe objects to ReadUsersCsv.ps1.


OUTPUTS
    None. ReadUsersCsv.ps1 does not generate any output.


    -------------------------- EXAMPLE 1 --------------------------

    C:\PS>.\ReadUsersCsv.ps1 -Path C:\Temp\Users.csv -UserName j.doe






RELATED LINKS



PS C:\Scripts>
```

第54章：归档模块

参数	详情
CompressionLevel	*（仅限 Compress-Archive）* 设置压缩级别为最快、最佳或无压缩
确认	运行前提示确认
强制	强制命令运行，无需确认
LiteralPath	按字面使用的路径，不支持通配符，使用,指定多个路径
路径	可以包含通配符的路径，使用,指定多个路径
更新	(仅限 Compress-Archive) 更新现有归档文件
假设	模拟命令

归档模块 `Microsoft.PowerShell.Archive` 提供了将文件存储到 ZIP 归档（`Compress-Archive`）和解压缩（`Expand-Archive`）的功能。该模块适用于 PowerShell 5.0 及以上版本。

在早期版本的 PowerShell 中，可以使用 Community Extensions 或 .NET 的 System.IO.Compression.FileSystem。

第54.1节：使用通配符的 Compress-Archive

```
Compress-Archive -Path C:\Documents\* -CompressionLevel Optimal -DestinationPath
C:\Archives\Documents.zip
```

该命令：

- 压缩 `C:\Documents`
- 中的所有文件，使用 `Optimal` 压缩级别
- 将生成的归档保存到`C:\Archives\Documents.zip`
  - -如果目标路径DestinationPath中没有包含.zip，将自动添加。
  - -LiteralPath 可用于在命名时不带 .zip 的情况。

第54.2节：使用 Compress-Archive 更新现有 ZIP 文件

```
Compress-Archive -Path C:\Documents\* -Update -DestinationPath C:\Archives\Documents.zip
```

- 这将用来自 C:\Documents 的新文件添加或替换 Documents.zip 中的所有文件

第54.3节：使用 Expand-Archive 解压 Zip 文件

```
Expand-Archive -Path C:\Archives\Documents.zip -DestinationPath C:\Documents
```

- 这将把 Documents.zip 中的所有文件解压到文件夹 C:\Documents 中

Chapter 54: Archive Module

Parameter	Details
CompressionLevel	*(Compress-Archive only)* Set compression level to either `Fastest`, `Optimal` or `NoCompression`
Confirm	Prompts for confirmation before running
Force	Forces the command to run without confirmation
LiteralPath	Path that is used literally, *no wildcards supported*, use , to specify multiple paths
Path	Path that can contain wildcards, use , to specify multiple paths
Update	*(Compress-Archive only)* Update existing archive
WhatIf	Simulate the command

The Archive module `Microsoft.PowerShell.Archive` provides functions for storing files in ZIP archives (`Compress-Archive`) and extracting them (`Expand-Archive`). This module is available in PowerShell 5.0 and above.

In earlier versions of PowerShell the Community Extensions or .NET System.IO.Compression.FileSystem could be used.

Section 54.1: Compress-Archive with wildcard

```
Compress-Archive -Path C:\Documents\* -CompressionLevel Optimal -DestinationPath
C:\Archives\Documents.zip
```

This command:

- Compresses all files in `C:\Documents`
- Uses `Optimal` compression
- Save the resulting archive in `C:\Archives\Documents.zip`
  - `-DestinationPath` will add `.zip`if not present.
  - `-LiteralPath` can be used if you require naming it without `.zip`.

Section 54.2: Update existing ZIP with Compress-Archive

```
Compress-Archive -Path C:\Documents\* -Update -DestinationPath C:\Archives\Documents.zip
```

- this will add or replace all files `Documents.zip` with the new ones from `C:\Documents`

Section 54.3: Extract a Zip with Expand-Archive

```
Expand-Archive -Path C:\Archives\Documents.zip -DestinationPath C:\Documents
```

- this will extract all files from `Documents.zip` into the folder `C:\Documents`

第55章：基础设施自动化

自动化基础设施管理服务可以减少全职员工（FTE），并通过使用多种工具、编排器、编排引擎、脚本和简易用户界面，累计获得更好的投资回报率（ROI）

第55.1节：用于控制台应用程序黑盒集成测试的简单脚本

这是一个关于如何自动化测试与标准输入

和标准输出交互的控制台应用程序的简单示例。

被测试的应用程序读取并累加每一新行，在输入一个空白行后提供结果。当输出匹配时，PowerShell脚本会写入“pass”。

```
$process = New-Object System.Diagnostics.Process
$process.StartInfo.FileName = ".\ConsoleApp1.exe"
$process.StartInfo.UseShellExecute = $false
$process.StartInfo.RedirectStandardOutput = $true
$process.StartInfo.RedirectStandardInput = $true
if ( $process.Start() ) {
    # 输入
    $process.StandardInput.WriteLine("1");
    $process.StandardInput.WriteLine("2");
    $process.StandardInput.WriteLine("3");

    $process.StandardInput.WriteLine();
    # 输出检查
    $output = $process.StandardOutput.ReadToEnd()
    if ( $output ) {
        if ( $output.Contains("sum 6") ) {
            Write "pass"
        }
        else {
            Write-Error $output
        }
    }
    $process.WaitForExit()
}
```

Chapter 55: Infrastructure Automation

Automating Infrastructure Management Services results in reducing the FTE as well as cumulatively getting better ROI using multiple tools, orchestrators, orchestration Engine , scripts and easy UI

Section 55.1: Simple script for black-box integration test of console applications

This is a simple example on how you can automate tests for a console application that interact with standard input and standard output.

The tested application read and sum every new line and will provide the result after a single white line is provided. The power shell script write "pass" when the output match.

```
$process = New-Object System.Diagnostics.Process
$process.StartInfo.FileName = ".\ConsoleApp1.exe"
$process.StartInfo.UseShellExecute = $false
$process.StartInfo.RedirectStandardOutput = $true
$process.StartInfo.RedirectStandardInput = $true
if ( $process.Start() ) {
    # input
    $process.StandardInput.WriteLine("1");
    $process.StandardInput.WriteLine("2");
    $process.StandardInput.WriteLine("3");
    $process.StandardInput.WriteLine();
    $process.StandardInput.WriteLine();
    # output check
    $output = $process.StandardOutput.ReadToEnd()
    if ( $output ) {
        if ( $output.Contains("sum 6") ) {
            Write "pass"
        }
        else {
            Write-Error $output
        }
    }
    $process.WaitForExit()
}
```

第56章：PSScriptAnalyzer - PowerShell脚本分析器

PSScriptAnalyzer，https://github.com/PowerShell/PSScriptAnalyzer，是一个用于Windows PowerShell模块和脚本的静态代码检查工具。PSScriptAnalyzer通过运行一组基于PowerShell团队和社区确定的PowerShell最佳实践的规则，检查Windows PowerShell代码的质量。
它生成DiagnosticResults（错误和警告）以通知用户潜在的代码缺陷，并建议可能的
改进方案。

```
PS> Install-Module -Name PSScriptAnalyzer
```

第56.1节：使用内置预设规则集分析脚本

ScriptAnalyzer附带了一组内置预设规则，可用于分析脚本。这些包括：PSGallery，DSC和CodeFormatting。它们可以按如下方式执行：

PowerShell Gallery规则

要执行PowerShell Gallery规则，请使用以下命令：

```
Invoke-ScriptAnalyzer -Path /path/to/module/ -Settings PSGallery -Recurse
```

DSC 规则

要执行 DSC 规则，请使用以下命令：

```
Invoke-ScriptAnalyzer -Path /path/to/module/ -Settings DSC -Recurse
```

代码格式化规则

要执行代码格式化规则，请使用以下命令：

```
Invoke-ScriptAnalyzer -Path /path/to/module/ -Settings CodeFormatting -Recurse
```

第 56.2 节：针对所有内置规则分析脚本

要针对单个脚本文件运行脚本分析器，请执行：

```
Invoke-ScriptAnalyzer -Path myscript.ps1
```

这将针对每个内置规则分析您的脚本。如果您的脚本足够大，可能会导致大量的
警告和/或错误。

要对整个目录运行脚本分析器，请指定包含要分析的脚本、模块和 DSC 文件的文件夹。
如果还想搜索子目录中的文件进行分析，请指定 Recurse 参数。

```
Invoke-ScriptAnalyzer -Path . -Recurse
```

第56.3节：列出所有内置规则

要查看所有内置规则，请执行：

Chapter 56: PSScriptAnalyzer - PowerShell Script Analyzer

PSScriptAnalyzer, https://github.com/PowerShell/PSScriptAnalyzer, is a static code checker for Windows PowerShell modules and scripts. PSScriptAnalyzer checks the quality of Windows PowerShell code by running a set of rules based on PowerShell best practices identified by the PowerShell Team and community. It generates DiagnosticResults (errors and warnings) to inform users about potential code defects and suggests possible solutions for improvements.

```
PS> Install-Module -Name PSScriptAnalyzer
```

Section 56.1: Analyzing scripts with the built-in preset rulesets

ScriptAnalyzer ships with sets of built-in preset rules that can be used to analyze scripts. These include: `PSGallery`, `DSC` and `CodeFormatting`. They can be executed as follows:

PowerShell Gallery rules

To execute the PowerShell Gallery rules use the following command:

```
Invoke-ScriptAnalyzer -Path /path/to/module/ -Settings PSGallery -Recurse
```

DSC rules

To execute the DSC rules use the following command:

```
Invoke-ScriptAnalyzer -Path /path/to/module/ -Settings DSC -Recurse
```

Code formatting rules

To execute the code formatting rules use the following command:

```
Invoke-ScriptAnalyzer -Path /path/to/module/ -Settings CodeFormatting -Recurse
```

Section 56.2: Analyzing scripts against every built-in rule

To run the script analyzer against a single script file execute:

```
Invoke-ScriptAnalyzer -Path myscript.ps1
```

This will analyze your script against every built-in rule. If your script is sufficiently large that could result in a lot of warnings and/or errors.

To run the script analyzer against a whole directory, specify the folder containing the script, module and DSC files you want analyzed. Specify the Recurse parameter if you also want sub-directories searched for files to analyze.

```
Invoke-ScriptAnalyzer -Path . -Recurse
```

Section 56.3: List all built-in rules

To see all the built-in rules execute:

```
Get-ScriptAnalyzerRule
```

```
Get-ScriptAnalyzerRule
```

第57章：期望状态配置

第57.1节：简单示例 - 启用 WindowsFeature

```
configuration EnableIISFeature
{
node localhost
    {
Windows功能 IIS
        {
确保 = “存在”
          名称 = “Web-Server”
        }
    }
}
```

如果你在Powershell中运行此配置（EnableIISFeature），它将生成一个localhost.mof文件。这是你可以在机器上运行的“编译”配置。

要在本地主机上测试DSC配置，你只需调用以下命令：

```
Start-DscConfiguration -ComputerName localhost -Wait
```

第57.2节：在远程机器上启动DSC（mof）

在远程机器上启动DSC几乎同样简单。假设你已经设置了Powershell远程（或启用了WSMAN）。

```
$remoteComputer = "myserver.somedomain.com"
$cred = (Get-Credential)
Start-DSCConfiguration -ServerName $remoteComputer -Credential $cred -Verbose
```

注意： 假设你已经在本地机器上为节点编译了配置（并且在启动配置之前，文件 myserver.somedomain.com.mof 已存在）

第57.3节：将psd1（数据文件）导入本地变量

有时测试你的Powershell数据文件并遍历节点和服务器会很有用。

Powershell 5（WMF5）新增了一个用于此目的的实用功能，称为Import-PowerShellDataFile。

示例：

```
$data = Import-PowerShellDataFile -path .\MydataFile.psd1
$data.AllNodes
```

第57.4节：列出可用的DSC资源

要列出你创作节点上的可用DSC资源：

```
Get-DscResource
```

这将列出你创作节点上所有已安装模块（位于你的PSModulePath中）的所有资源。

Chapter 57: Desired State Configuration

Section 57.1: Simple example - Enabling WindowsFeature

```
configuration EnableIISFeature
{
    node localhost
    {
        WindowsFeature IIS
        {
            Ensure = “Present”
            Name = “Web-Server”
        }
    }
}
```

If you run this configuration in Powershell (EnableIISFeature), it will produce a localhost.mof file. This is the "compiled" configuration you can run on a machine.

To test the DSC configuration on your localhost, you can simply invoke the following:

```
Start-DscConfiguration -ComputerName localhost -Wait
```

Section 57.2: Starting DSC (mof) on remote machine

Starting a DSC on a remote machine is almost just as simple. Assuming you've already set up Powershell remoting (or enabled WSMAN).

```
$remoteComputer = "myserver.somedomain.com"
$cred = (Get-Credential)
Start-DSCConfiguration -ServerName $remoteComputer -Credential $cred -Verbose
```

Nb: Assuming you have compiled a configuration for your node on your localmachine (and that the file myserver.somedomain.com.mof is present prior to starting the configuration)

Section 57.3: Importing psd1 (data file) into local variable

Sometimes it can be useful to test your Powershell data files and iterate through the nodes and servers.

Powershell 5 (WMF5) added this neat little feature for doing this called Import-PowerShellDataFile .

Example:

```
$data = Import-PowerShellDataFile -path .\MydataFile.psd1
$data.AllNodes
```

Section 57.4: List available DSC Resources

To list available DSC resources on your authoring node:

```
Get-DscResource
```

This will list all resources for all installed modules (that are in your PSModulePath) on your authoring node.

要列出WMF 5中在线源（PSGallery等）中可用的所有DSC资源：

```
Find-DSCResource
```

第57.5节：导入资源以供DSC使用

在配置中使用资源之前，必须显式导入该资源。仅仅在计算机上安装该资源，不能隐式使用该资源。

使用 Import-DscResource 导入资源。

示例展示如何导入 PSDesiredStateConfiguration 资源和 File 资源。

```
配置 InstallPreReqs
{
   参数(); # DSC 的参数写在这里。

   Import-DscResource PSDesiredStateConfiguration

File CheckForTmpFolder {
        类型 = '目录'
       目标路径 = 'C:\Tmp'
       确保 = "存在"
    }
 }
```

注意：为了使 DSC 资源正常工作，运行配置时目标机器上必须安装相应模块。如果未安装，配置将失败。

To list all available DSC resources that can be found in the online sources (PSGallery ++) on WMF 5:

```
Find-DSCResource
```

Section 57.5: Importing resources for use in DSC

Before you can use a resource in a configuration, you must explicitly import it. Just having it installed on your computer, will not let you use the resource implicitly.

Import a resource by using Import-DscResource .

Example showing how to import the PSDesiredStateConfiguration resource and the File resource.

```
Configuration InstallPreReqs
{
   param(); # params to DSC goes here.

   Import-DscResource PSDesiredStateConfiguration

   File CheckForTmpFolder {
        Type = 'Directory'
        DestinationPath = 'C:\Tmp'
        Ensure = "Present"
    }
 }
```

Note: In order for DSC Resources to work, you must have the modules installed on the target machines when running the configuration. If you don't have them installed, the configuration will fail.

第58章：使用 ShouldProcess

参数	详情
目标	正在更改的资源。
操作	正在执行的操作。默认为 cmdlet 的名称。

第 58.1 节：完整使用示例

其他示例无法清楚地向我解释如何触发条件逻辑。

此示例还显示了底层命令也会响应 -Confirm 标志！

```
<#
Restart-Win32Computer
#>

function Restart-Win32Computer
{
    [CmdletBinding(SupportsShouldProcess=$true,ConfirmImpact="High")]
    param (
    [parameter(Mandatory=$true,ValueFromPipeline=$true,ValueFromPipelineByPropertyName=$true)]
    [string[]]$computerName,
    [parameter(Mandatory=$true)]
    [string][ValidateSet("Restart","LogOff","Shutdown","PowerOff")] $action,
    [boolean]$force = $false
)
BEGIN {
# 将动作转换为方法所需的数值
switch($action) {
    "Restart"
    {
        $_action = 2
        break
    }
    "LogOff"
    {
        $_action = 0
        break
    }
    "Shutdown"
    {
        $_action = 2
        break
    }
    "PowerOff"
    {
        $_action = 8
        break
    }
}
# 要强制执行，给数值加4
if($force)
{
    $_action += 4
}
write-verbose "操作设置为 $action"
}
PROCESS {
    write-verbose "尝试连接到 $computername"
```

Chapter 58: Using ShouldProcess

Parameter	Details
Target	The resource being changed.
Action	The operation being performed. Defaults to the name of the cmdlet.

Section 58.1: Full Usage Example

Other examples couldn't clearly explain to me how to trigger the conditional logic.

This example also shows that underlying commands will also listen to the -Confirm flag!

```
<#
Restart-Win32Computer
#>

function Restart-Win32Computer
{
    [CmdletBinding(SupportsShouldProcess=$true,ConfirmImpact="High")]
    param (
    [parameter(Mandatory=$true,ValueFromPipeline=$true,ValueFromPipelineByPropertyName=$true)]
    [string[]]$computerName,
    [parameter(Mandatory=$true)]
    [string][ValidateSet("Restart","LogOff","Shutdown","PowerOff")] $action,
    [boolean]$force = $false
)
BEGIN {
# translate action to numeric value required by the method
switch($action) {
    "Restart"
    {
        $_action = 2
        break
    }
    "LogOff"
    {
        $_action = 0
        break
    }
    "Shutdown"
    {
        $_action = 2
        break
    }
    "PowerOff"
    {
        $_action = 8
        break
    }
}
# to force, add 4 to the value
if($force)
{
    $_action += 4
}
write-verbose "Action set to $action"
}
PROCESS {
    write-verbose "Attempting to connect to $computername"
    # 这是我们如何支持 -whatif 和 -confirm
    # 这些由 cmdlet 绑定中的 SupportsShouldProcess
    # 参数启用
    if($pscmdlet.ShouldProcess($computername)) {
        get-wmiobject win32_operatingsystem -computername $computername | invoke-wmimethod -name
Win32Shutdown -argumentlist $_action
    }
```

```
    # this is how we support -whatif and -confirm
    # which are enabled by the SupportsShouldProcess
    # parameter in the cmdlet bindnig
    if($pscmdlet.ShouldProcess($computername)) {
        get-wmiobject win32_operatingsystem -computername $computername | invoke-wmimethod -name
Win32Shutdown -argumentlist $_action
    }
}
}
#Usage:
#This will only output a description of the actions that this command would execute if -WhatIf is
removed.
'localhost','server1'| Restart-Win32Computer -action LogOff -whatif

#This will request the permission of the caller to continue with this item.
#Attention: in this example you will get two confirmation request because all cmdlets called by
this cmdlet that also support ShouldProcess, will ask for their own confirmations...
'localhost','server1'| Restart-Win32Computer -action LogOff -Confirm
```

Section 58.2: Adding -WhatIf and -Confirm support to your cmdlet

```
function Invoke-MyCmdlet {
    [CmdletBinding(SupportsShouldProcess = $true)]
    param()
    # ...
}
```

Section 58.3: Using ShouldProcess() with one argument

```
if ($PSCmdlet.ShouldProcess("Target of action")) {
    # Do the thing
}
```

When using `-WhatIf`:

```
What if: Performing the action "Invoke-MyCmdlet" on target "Target of action"
```

When using `-Confirm`:

Are you sure you want to perform this action? Performing operation "Invoke-MyCmdlet" on target "Target of action"
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"):

第59章：计划任务模块

如何使用 Windows 8/Server 2012 及以后版本中提供的计划任务模块的示例。

第59.1节：在计划任务中运行PowerShell脚本

创建一个计划任务，立即执行，然后在启动时运行C:\myscript.ps1，使用SYSTEM身份

```
$ScheduledTaskPrincipal = New-ScheduledTaskPrincipal -UserId "SYSTEM" -LogonType ServiceAccount
$ScheduledTaskTrigger1 = New-ScheduledTaskTrigger -AtStartup
$ScheduledTaskTrigger2 = New-ScheduledTaskTrigger -Once -At $(Get-Date) -RepetitionInterval
"00:01:00" -RepetitionDuration $([timeSpan] "24855.03:14:07")
$ScheduledTaskActionParams = @{
Execute = "PowerShell.exe"
Argument = '-executionpolicy Bypass -NonInteractive -c C:\myscript.ps1 -verbose >>
 C:\output.log 2>&1"'
}
$ScheduledTaskAction = New-ScheduledTaskAction @ScheduledTaskActionParams
Register-ScheduledTask -Principal $ScheduledTaskPrincipal -Trigger
@($ScheduledTaskTrigger1,$ScheduledTaskTrigger2) -TaskName "示例任务" -Action
$ScheduledTaskAction
```

Chapter 59: Scheduled tasks module

Examples of how to use the Scheduled Tasks module available in Windows 8/Server 2012 and on.

Section 59.1: Run PowerShell Script in Scheduled Task

Creates a scheduled task that executes immediately, then on start up to run `C:\myscript.ps1` as `SYSTEM`

```
$ScheduledTaskPrincipal = New-ScheduledTaskPrincipal -UserId "SYSTEM" -LogonType ServiceAccount
$ScheduledTaskTrigger1 = New-ScheduledTaskTrigger -AtStartup
$ScheduledTaskTrigger2 = New-ScheduledTaskTrigger -Once -At $(Get-Date) -RepetitionInterval
"00:01:00" -RepetitionDuration $([timeSpan] "24855.03:14:07")
$ScheduledTaskActionParams = @{
    Execute = "PowerShell.exe"
    Argument = '-executionpolicy Bypass -NonInteractive -c C:\myscript.ps1 -verbose >>
 C:\output.log 2>&1"'
}
$ScheduledTaskAction = New-ScheduledTaskAction @ScheduledTaskActionParams
Register-ScheduledTask -Principal $ScheduledTaskPrincipal -Trigger
@($ScheduledTaskTrigger1,$ScheduledTaskTrigger2) -TaskName "Example Task" -Action
$ScheduledTaskAction
```

第60章：ISE模块

Windows PowerShell 集成脚本环境（ISE）是一款主机应用程序，允许您在图形化且直观的环境中编写、运行和测试脚本及模块。Windows PowerShell ISE 的主要功能包括语法高亮、制表符补全、智能感知、可视化调试、Unicode 兼容性以及上下文相关帮助，提供丰富的脚本编写体验。

第60.1节：测试脚本

ISE的简单而强大的用法例如是在顶部区域编写代码（带有直观的语法高亮），然后只需选中代码并按F8键即可运行代码。

```
function Get-Sum
{
    foreach ($i in $Input)
    {$Sum += $i}
    $Sum

1..10 | Get-Sum

#output
55
```

Chapter 60: ISE module

Windows PowerShell Integrated Scripting Environment (ISE) is a host application that enables you to write, run, and test scripts and modules in a graphical and intuitive environment. Key features in Windows PowerShell ISE include syntax-coloring, tab completion, Intellisense, visual debugging, Unicode compliance, and context-sensitive Help, and provide a rich scripting experience.

Section 60.1: Test Scripts

The simple, yet powerful use of the ISE is e.g. writing code in the top section (with intuitive syntax coloring) and run the code by simply marking it and hitting the F8 key.

```
function Get-Sum
{
    foreach ($i in $Input)
    {$Sum += $i}
    $Sum

1..10 | Get-Sum

#output
55
```

第61章：创建基于类的DSC资源

从PowerShell 5.0版本开始，您可以使用PowerShell类定义来创建期望状态配置（DSC）资源。

为了帮助构建DSC资源，有一个应用于类定义的[DscResource()]属性，以及一个用于将属性指定为DSC资源用户可配置的[DscProperty()]资源。

第61.1节：创建DSC资源骨架类

```
[DscResource()]
class 文件 {
}
```

此示例演示了如何构建声明 DSC 资源的 PowerShell 类的外部部分。你仍然需要填写类定义的内容。

第61.2节：带有键属性的 DSC 资源骨架

```
[DscResource()]
class 票据 {
  [DscProperty(Key)]
  [string] $TicketId
}
```

DSC 资源必须声明至少一个键属性。键属性用于唯一标识该资源与其他资源的区别。例如，假设你正在构建一个表示票务系统中票据的 DSC 资源。每张票据都将通过票据 ID 唯一表示。

每个将暴露给 DSC 资源用户的属性都必须用[DscProperty()]特性进行装饰。该特性接受一个key参数，用于指示该属性是 DSC 资源的键属性。

第61.3节：带有必需属性的 DSC 资源

```
[DscResource()]
class 票据 {
  [DscProperty(Key)]
  [string] $TicketId

  [DscProperty(Mandatory)]
  [string] $Subject
}
```

在构建 DSC 资源时，你会发现并非每个属性都应该是必需的。然而，有一些核心属性你希望确保由 DSC 资源的用户进行配置。你可以使用[DscResource()]特性的Mandatory参数来声明某个属性是 DSC 资源用户必须提供的。

在上面的示例中，我们为Ticket资源添加了一个Subject属性，该属性表示票务系统中的唯一票据，并将其指定为Mandatory属性。

Chapter 61: Creating DSC Class-Based Resources

Starting with PowerShell version 5.0, you can use PowerShell class definitions to create Desired State Configuration (DSC) Resources.

To aid in building DSC Resource, there's a [DscResource()] attribute that's applied to the class definition, and a [DscProperty()] resource to designate properties as configurable by the DSC Resource user.

Section 61.1: Create a DSC Resource Skeleton Class

```
[DscResource()]
class File {
}
```

This example demonstrates how to build the outer section of a PowerShell class, that declares a DSC Resource. You still need to fill in the contents of the class definition.

Section 61.2: DSC Resource Skeleton with Key Property

```
[DscResource()]
class Ticket {
  [DscProperty(Key)]
  [string] $TicketId
}
```

A DSC Resource must declare at least one key property. The key property is what uniquely identifies the resource from other resources. For example, let's say that you're building a DSC Resource that represents a ticket in a ticketing system. Each ticket would be uniquely represented with a ticket ID.

Each property that will be exposed to the *user* of the DSC Resource must be decorated with the [DscProperty()] attribute. This attributes accepts a key parameter, to indicate that the property is a key attribute for the DSC Resource.

Section 61.3: DSC Resource with Mandatory Property

```
[DscResource()]
class Ticket {
  [DscProperty(Key)]
  [string] $TicketId

  [DscProperty(Mandatory)]
  [string] $Subject
}
```

When building a DSC Resource, you'll often find that not every single property should be mandatory. However, there are some core properties that you'll want to ensure are configured by the user of the DSC Resource. You use the Mandatory parameter of the [DscResource()] attribute to declare a property as required by the DSC Resource's user.

In the example above, we've added a Subject property to a Ticket resource, that represents a unique ticket in a ticketing system, and designated it as a Mandatory property.

第61.4节：具有所需方法的DSC资源

```
[DscResource()]
class 票据 {
  [DscProperty(Key)]
  [string] $TicketId

  # 工单的主题行
  [DscProperty(必填)]
  [string] $Subject

  # 获取/设置工单应为打开还是关闭状态
  [DscProperty(必填)]
  [string] $TicketState

  [void] Set() {
    # 创建或更新资源
  }

  [Ticket] Get() {
    # 以对象形式返回资源的当前状态
    $TicketState = [Ticket]::new()
    return $TicketState
  }

  [bool] Test() {
    # 如果达到期望状态则返回 $true
    # 如果未达到期望状态则返回 $false
    return $false
  }
}
```

这是一个完整的 DSC 资源示例，展示了构建有效资源的所有核心要求。
方法实现尚不完整，但提供的目的是展示基本结构。

Section 61.4: DSC Resource with Required Methods

```
[DscResource()]
class Ticket {
  [DscProperty(Key)]
  [string] $TicketId

  # The subject line of the ticket
  [DscProperty(Mandatory)]
  [string] $Subject

  # Get / Set if ticket should be open or closed
  [DscProperty(Mandatory)]
  [string] $TicketState

  [void] Set() {
    # Create or update the resource
  }

  [Ticket] Get() {
    # Return the resource's current state as an object
    $TicketState = [Ticket]::new()
    return $TicketState
  }

  [bool] Test() {
    # Return $true if desired state is met
    # Return $false if desired state is not met
    return $false
  }
}
```

This is a complete DSC Resource that demonstrates all of the core requirements to build a valid resource. The method implementations are not complete, but are provided with the intention of showing the basic structure.

第62章：WMI 和 CIM

第62.1节：查询对象

CIM/WMI 最常用于查询设备上的信息或配置。通过表示配置、进程、用户等的类。在 PowerShell 中，有多种方式访问这些类和实例，但最常用的方法是使用 Get-CimInstance（CIM）或 Get-WmiObject（WMI）命令。

列出 CIM 类的所有对象

您可以列出某个类的所有实例。

版本 ≥ 3.0

CIM：

```
> Get-CimInstance -ClassName Win32_Process
```

```
进程ID 名称                          句柄数 工作集大小 虚拟内存大小
--------- ----                          ----------- --------------- -----------
0         系统空闲进程                     0           4096            65536
4         系统                          1459           32768           3563520
480       安全系统                   0           3731456         0
484       smss.exe                         52          372736          2199029891072
....
....
```

WMI：

```
Get-WmiObject -Class Win32_Process
```

使用筛选器

您可以应用筛选器仅获取特定的 CIM/WMI 类实例。筛选器使用 `WQL`（默认）或 CQL 编写（添加 `-QueryDialect` CQL）。`-Filter` 使用完整 WQL/CQL 查询的 `WHERE` 部分。

版本 ≥ 3.0

CIM：

```
Get-CimInstance -ClassName Win32_Process -Filter "Name = 'powershell.exe'"
```

```
ProcessId 名称          句柄计数 工作集大小 虚拟大小
--------- ----          ----------- --------------- -----------
4800      powershell.exe 676         88305664        2199697199104
```

WMI：

```
Get-WmiObject -Class Win32_Process -Filter "Name = 'powershell.exe'"
```

```
...
标题                       : powershell.exe
命令行                       : "C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe"
创建类名                     : Win32_Process
创建日期                     : 20160913184324.393887+120
```

Chapter 62: WMI and CIM

Section 62.1: Querying objects

CIM/WMI is most commonly used to query information or configuration on a device. Through a class that represents a configuration, process, user etc. In PowerShell there are multiple ways to access these classes and instances, but the most common ways are by using the `Get-CimInstance` (CIM) or `Get-WmiObject` (WMI) cmdlets.

List all objects for CIM-class

You can list all instances of a class.

Version ≥ 3.0

CIM:

```
> Get-CimInstance -ClassName Win32_Process
```

```
ProcessId Name                        HandleCount WorkingSetSize VirtualSize
--------- ----                        ----------- -------------- -----------
0         System Idle Process         0           4096           65536
4         System                      1459        32768          3563520
480       Secure System               0           3731456        0
484       smss.exe                    52          372736         2199029891072
....
....
```

WMI:

```
Get-WmiObject -Class Win32_Process
```

Using a filter

You can apply a filter to only get specific instances of a CIM/WMI-class. Filters are written using `WQL` (default) or CQL (add `-QueryDialect` CQL). `-Filter` uses the `WHERE`-part of a full WQL/CQL-query.

Version ≥ 3.0

CIM:

```
Get-CimInstance -ClassName Win32_Process -Filter "Name = 'powershell.exe'"
```

```
ProcessId Name           HandleCount WorkingSetSize VirtualSize
--------- ----           ----------- -------------- -----------
4800      powershell.exe 676         88305664       2199697199104
```

WMI:

```
Get-WmiObject -Class Win32_Process -Filter "Name = 'powershell.exe'"
```

```
...
Caption                    : powershell.exe
CommandLine                : "C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe"
CreationClassName          : Win32_Process
CreationDate               : 20160913184324.393887+120
```

```
计算机系统创建类名          : Win32_ComputerSystem
计算机系统名称              : STACKOVERFLOW-PC
描述                        : powershell.exe
可执行路径                  : C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
执行状态                    :
句柄                        : 4800
句柄计数                    : 673
....
```

使用 WQL 查询：

您也可以使用 WQL/CQL 查询来查询和筛选实例。

版本 ≥ 3.0

CIM：

```
Get-CimInstance -Query "SELECT * FROM Win32_Process WHERE Name = 'powershell.exe'"
```

```
进程ID 名称           句柄计数 工作集大小 虚拟内存大小
--------- ----           ----------- -------------- -----------
4800      powershell.exe 673         88387584       2199696674816
```

查询不同命名空间中的对象：

版本 ≥ 3.0

CIM：

```
> Get-CimInstance -命名空间 "root/SecurityCenter2" -类名 AntiVirusProduct
```

```


显示名称                : Windows Defender
实例GUID                : {D68DDC3A-831F-4fae-9E44-DA132C1ACF46}
签名产品可执行文件路径   : %ProgramFiles%\Windows Defender\MSASCui.exe
签名报告可执行文件路径 : %ProgramFiles%\Windows Defender\MsMpeng.exe
产品状态               : 397568
时间戳                  : Fri, 09 Sep 2016 21:26:41 GMT
PS计算机名             :
```

WMI：

```
> Get-WmiObject -命名空间 "root\SecurityCenter2" -类 AntiVirusProduct
```

```
__类别                    : 2
__类名                    : AntiVirusProduct
__超类名                :
__家族                    : AntiVirusProduct
__相对路径                     : AntiVirusProduct.instanceGuid="{D68DDC3A-831F-4fae-9E44-DA132C1ACF46}"
__属性计数           : 6
__派生                 : {}
__服务器                  : STACKOVERFLOW-PC
__命名空间                : ROOT\SecurityCenter2
__路径                    : \\STACKOVERFLOW-
PC\ROOT\SecurityCenter2:AntiVirusProduct.instanceGuid="{D68DDC3A-831F-4fae-9E44-DA132C1ACF46}"
显示名称                : Windows Defender
instanceGuid               : {D68DDC3A-831F-4fae-9E44-DA132C1ACF46}
签名产品可执行文件路径   : %ProgramFiles%\Windows Defender\MSASCui.exe
```

```
CSCreationClassName        : Win32_ComputerSystem
CSName                     : STACKOVERFLOW-PC
Description                : powershell.exe
ExecutablePath             : C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
ExecutionState             :
Handle                     : 4800
HandleCount                : 673
....
```

Using a WQL-query:

You can also use a WQL/CQL-query to query and filter instances.

Version ≥ 3.0

CIM:

```
Get-CimInstance -Query "SELECT * FROM Win32_Process WHERE Name = 'powershell.exe'"
```

```
ProcessId Name           HandleCount WorkingSetSize VirtualSize
--------- ----           ----------- -------------- -----------
4800      powershell.exe 673         88387584       2199696674816
```

Querying objects in a different namespace:

Version ≥ 3.0

CIM:

```
> Get-CimInstance -Namespace "root/SecurityCenter2" -ClassName AntiVirusProduct
```

```


displayName              : Windows Defender
instanceGuid             : {D68DDC3A-831F-4fae-9E44-DA132C1ACF46}
pathToSignedProductExe   : %ProgramFiles%\Windows Defender\MSASCui.exe
pathToSignedReportingExe : %ProgramFiles%\Windows Defender\MsMpeng.exe
productState             : 397568
timestamp                : Fri, 09 Sep 2016 21:26:41 GMT
PSComputerName           :
```

WMI:

```
> Get-WmiObject -Namespace "root\SecurityCenter2" -Class AntiVirusProduct
```

```
__GENUS                  : 2
__CLASS                  : AntiVirusProduct
__SUPERCLASS             :
__DYNASTY                : AntiVirusProduct
__RELPATH                : AntiVirusProduct.instanceGuid="{D68DDC3A-831F-4fae-9E44-DA132C1ACF46}"
__PROPERTY_COUNT         : 6
__DERIVATION             : {}
__SERVER                 : STACKOVERFLOW-PC
__NAMESPACE              : ROOT\SecurityCenter2
__PATH                   : \\STACKOVERFLOW-
PC\ROOT\SecurityCenter2:AntiVirusProduct.instanceGuid="{D68DDC3A-831F-4fae-9E44-DA132C1ACF46}"
displayName              : Windows Defender
instanceGuid             : {D68DDC3A-831F-4fae-9E44-DA132C1ACF46}
pathToSignedProductExe   : %ProgramFiles%\Windows Defender\MSASCui.exe
```

```
签名报告可执行文件路径 : %ProgramFiles%\Windows Defender\MsMpeng.exe
产品状态              : 397568
时间戳                 : Fri, 09 Sep 2016 21:26:41 GMT
PS计算机名             : STACKOVERFLOW-PC
```

第62.2节：类和命名空间

CIM和WMI中有许多类，这些类被划分到多个命名空间中。Windows中最常用（也是默认）的命名空间是root/cimv2。为了找到合适的类，列出所有类或进行搜索是很有用的。

列出可用类

您可以列出计算机上默认命名空间（root/cimv2）中的所有可用类。

版本 ≥ 3.0

CIM：

```
Get-CimClass
```

WMI：

```
Get-WmiObject -List
```

搜索类

您可以使用通配符搜索特定类。例如：查找包含单词process的类。

版本 ≥ 3.0

CIM：

```
> Get-CimClass -ClassName "*Process*"

   NameSpace: ROOT/CIMV2

CimClassName                         CimClassMethods      CimClassProperties

------------                         ---------------      ------------------

Win32_ProcessTrace                        {}                   {SECURITY_DESCRIPTOR, TIME_CREATED,
ParentProcessID, ProcessID...}
Win32_ProcessStartTrace                   {}                   {SECURITY_DESCRIPTOR, TIME_CREATED,
ParentProcessID, ProcessID...}
Win32_ProcessStopTrace                    {}                   {SECURITY_DESCRIPTOR, TIME_CREATED,
ParentProcessID, ProcessID...}
CIM_Process                               {}                   {Caption, Description, InstallDate,
Name...}
Win32_Process                    {创建, 终止... {标题, 描述, 安装日期,
名称...}
CIM_Processor                    {设置电源状态, R... {标题, 描述, 安装日期,
名称...}
Win32_Processor                  {设置电源状态, R... {标题, 描述, 安装日期,
名称...}
...
```

WMI：

```
pathToSignedReportingExe : %ProgramFiles%\Windows Defender\MsMpeng.exe
productState             : 397568
timestamp                : Fri, 09 Sep 2016 21:26:41 GMT
PSComputerName           : STACKOVERFLOW-PC
```

Section 62.2: Classes and namespaces

There are many classes available in CIM and WMI which are separated into multiple namespaces. The most common (and default) namespace in Windows is root/cimv2. To find the right class, it can useful to list all or search.

List available classes

You can list all available classes in the default namespace (root/cimv2) on a computer.

Version ≥ 3.0

CIM:

```
Get-CimClass
```

WMI:

```
Get-WmiObject -List
```

Search for a class

You can search for specific classes using wildcards. Ex: Find classes containing the word process.

Version ≥ 3.0

CIM:

```
> Get-CimClass -ClassName "*Process*"

   NameSpace: ROOT/CIMV2

CimClassName                        CimClassMethods      CimClassProperties

------------                        ---------------      ------------------

Win32_ProcessTrace                  {}                   {SECURITY_DESCRIPTOR, TIME_CREATED,
ParentProcessID, ProcessID...}
Win32_ProcessStartTrace             {}                   {SECURITY_DESCRIPTOR, TIME_CREATED,
ParentProcessID, ProcessID...}
Win32_ProcessStopTrace              {}                   {SECURITY_DESCRIPTOR, TIME_CREATED,
ParentProcessID, ProcessID...}
CIM_Process                         {}                   {Caption, Description, InstallDate,
Name...}
Win32_Process                       {Create, Terminat... {Caption, Description, InstallDate,
Name...}
CIM_Processor                       {SetPowerState, R... {Caption, Description, InstallDate,
Name...}
Win32_Processor                     {SetPowerState, R... {Caption, Description, InstallDate,
Name...}
...
```

WMI:

```
Get-WmiObject -List -Class "*Process*"
```

列出不同命名空间中的类

根命名空间简称为root。您可以使用-NameSpace参数列出另一个命名空间中的类。

版本 ≥ 3.0

CIM：

```
> Get-CimClass -Namespace "root/SecurityCenter2"

   命名空间: ROOT/SecurityCenter2

CimClassName                        CimClassMethods      CimClassProperties

------------                        ---------------      ------------------
....
AntiSpywareProduct                  {}                   {displayName, instanceGuid,
pathToSignedProductExe, pathToSignedReportingE...
AntiVirusProduct                    {}                   {displayName, instanceGuid,
pathToSignedProductExe, pathToSignedReportingE...
FirewallProduct                     {}                   {displayName, instanceGuid,
pathToSignedProductExe, pathToSignedReportingE...
```

WMI：

```
Get-WmiObject -Class "__Namespace" -Namespace "root"
```

列出可用的命名空间

要查找 root（或其他命名空间）的可用子命名空间，请查询该命名空间的 __NAMESPACE 类中的对象。

版本 ≥ 3.0

CIM：

```
> Get-CimInstance -Namespace "root" -ClassName "__Namespace"

Name              PSComputerName
----              --------------
subscription
DEFAULT
CIMV2
msdtc
Cli
SECURITY
HyperVCluster
SecurityCenter2
RSOP
PEH
StandardCimv2
WMI
directory
Policy
virtualization
Interop
Hardware
ServiceModel
```

```
Get-WmiObject -List -Class "*Process*"
```

List classes in a different namespace

The root namespace is simply called root. You can list classes in another namespace using the -NameSpace parameter.

Version ≥ 3.0

CIM:

```
> Get-CimClass -Namespace "root/SecurityCenter2"

   NameSpace: ROOT/SecurityCenter2

CimClassName                        CimClassMethods      CimClassProperties

------------                        ---------------      ------------------
....
AntiSpywareProduct                  {}                   {displayName, instanceGuid,
pathToSignedProductExe, pathToSignedReportingE...
AntiVirusProduct                    {}                   {displayName, instanceGuid,
pathToSignedProductExe, pathToSignedReportingE...
FirewallProduct                     {}                   {displayName, instanceGuid,
pathToSignedProductExe, pathToSignedReportingE...
```

WMI:

```
Get-WmiObject -Class "__Namespace" -Namespace "root"
```

List available namespaces

To find available child-namespaces of root (or another namespace), query the objects in the __NAMESPACE-class for that namespace.

Version ≥ 3.0

CIM:

```
> Get-CimInstance -Namespace "root" -ClassName "__Namespace"

Name              PSComputerName
----              --------------
subscription
DEFAULT
CIMV2
msdtc
Cli
SECURITY
HyperVCluster
SecurityCenter2
RSOP
PEH
StandardCimv2
WMI
directory
Policy
virtualization
Interop
Hardware
ServiceModel
```

```
SecurityCenter
Microsoft
aspnet
Appv
```

WMI：

```
Get-WmiObject -List -Namespace "root"
```

```
SecurityCenter
Microsoft
aspnet
Appv
```

WMI:

```
Get-WmiObject -List -Namespace "root"
```

第63章：ActiveDirectory模块

本主题将向您介绍PowerShell中Active Directory模块内用于操作用户、组、计算机和对象的一些基本cmdlet。

第63.1节：用户

检索Active Directory用户

```
Get-ADUser -Identity JohnSmith
```

检索与用户关联的所有属性

```
Get-ADUser -Identity JohnSmith -Properties *
```

检索用户的选定属性

```
Get-ADUser -Identity JohnSmith -Properties * | Select-Object -Property sAMAccountName, Name, Mail
```

新建AD用户

```
New-ADUser -Name "MarySmith" -GivenName "Mary" -Surname "Smith" -DisplayName "MarySmith" -Path
"CN=Users,DC=Domain,DC=Local"
```

第63.2节：模块

```
#将ActiveDirectory模块添加到当前PowerShell会话
Import-Module ActiveDirectory
```

第63.3节：组

检索Active Directory组

```
Get-ADGroup -Identity "My-First-Group" #确保组名中有空格时使用引号
```

检索与组关联的所有属性

```
Get-ADGroup -Identity "My-First-Group" -Properties *
```

检索组的所有成员

```
Get-ADGroupMember -Identity "My-First-Group" | Select-Object -Property sAMAccountName
Get-ADgroup "MY-First-Group" -Properties Members | Select -ExpandProperty Members
```

将AD用户添加到AD组

```
Add-ADGroupMember -Identity "My-First-Group" -Members "JohnSmith"
```

新建AD组

```
New-ADGroup -GroupScope Universal -Name "My-Second-Group"
```

Chapter 63: ActiveDirectory module

This topic will introduce you to some of the basic cmdlets used within the Active Directory Module for PowerShell, for manipulating Users, Groups, Computers and Objects.

Section 63.1: Users

Retrieve Active Directory User

```
Get-ADUser -Identity JohnSmith
```

Retrieve All Properties Associated with User

```
Get-ADUser -Identity JohnSmith -Properties *
```

Retrieve Selected Properties for User

```
Get-ADUser -Identity JohnSmith -Properties * | Select-Object -Property sAMAccountName, Name, Mail
```

New AD User

```
New-ADUser -Name "MarySmith" -GivenName "Mary" -Surname "Smith" -DisplayName "MarySmith" -Path
"CN=Users,DC=Domain,DC=Local"
```

Section 63.2: Module

```
#Add the ActiveDirectory Module to current PowerShell Session
Import-Module ActiveDirectory
```

Section 63.3: Groups

Retrieve Active Directory Group

```
Get-ADGroup -Identity "My-First-Group" #Ensure if group name has space quotes are used
```

Retrieve All Properties Associated with Group

```
Get-ADGroup -Identity "My-First-Group" -Properties *
```

Retrieve All Members of a Group

```
Get-ADGroupMember -Identity "My-First-Group" | Select-Object -Property sAMAccountName
Get-ADgroup "MY-First-Group" -Properties Members | Select -ExpandProperty Members
```

Add AD User to an AD Group

```
Add-ADGroupMember -Identity "My-First-Group" -Members "JohnSmith"
```

New AD Group

```
New-ADGroup -GroupScope Universal -Name "My-Second-Group"
```

第63.4节：计算机

检索AD计算机

```
Get-ADComputer -Identity "JohnLaptop"
```

检索与计算机相关的所有属性

```
Get-ADComputer -Identity "JohnLaptop" -Properties *
```

检索计算机的选定属性

```
Get-ADComputer -Identity "JohnLaptop" -Properties * | Select-Object -Property Name, Enabled
```

第63.5节：对象

检索活动目录对象

```
#Identity 可以是 ObjectGUID、可分辨名称或更多内容
Get-ADObject -Identity "ObjectGUID07898"
```

移动活动目录对象

```
Move-ADObject -Identity "CN=JohnSmith,OU=Users,DC=Domain,DC=Local" -TargetPath
"OU=SuperUser,DC=Domain,DC=Local"
```

修改活动目录对象

```
Set-ADObject -Identity "CN=My-First-Group,OU=Groups,DC=Domain,DC=local" -Description "这是我的
第一个对象修改"
```

Section 63.4: Computers

Retrieve AD Computer

```
Get-ADComputer -Identity "JohnLaptop"
```

Retrieve All Properties Associated with Computer

```
Get-ADComputer -Identity "JohnLaptop" -Properties *
```

Retrieve Select Properties of Computer

```
Get-ADComputer -Identity "JohnLaptop" -Properties * | Select-Object -Property Name, Enabled
```

Section 63.5: Objects

Retrieve an Active Directory Object

```
#Identity can be ObjectGUID, Distinguished Name or many more
Get-ADObject -Identity "ObjectGUID07898"
```

Move an Active Directory Object

```
Move-ADObject -Identity "CN=JohnSmith,OU=Users,DC=Domain,DC=Local" -TargetPath
"OU=SuperUser,DC=Domain,DC=Local"
```

Modify an Active Directory Object

```
Set-ADObject -Identity "CN=My-First-Group,OU=Groups,DC=Domain,DC=local" -Description "This is My
First Object Modification"
```

第64章：SharePoint 模块

第64.1节：加载 SharePoint Snap-In

可以使用以下命令加载 SharePoint Snapin：

```
Add-PSSnapin "Microsoft.SharePoint.PowerShell"
```

这仅适用于64位版本的PowerShell。 如果窗口标题显示“Windows PowerShell (x86)”，则表示您使用了错误的版本。

如果Snap-In已经加载，上述代码将导致错误。使用以下代码仅在必要时加载，可用于Cmdlet/函数：

```
if ((Get-PSSnapin "Microsoft.SharePoint.PowerShell" -ErrorAction SilentlyContinue) -eq $null)
{
    Add-PSSnapin "Microsoft.SharePoint.PowerShell"
}
```

或者，如果您启动SharePoint管理外壳，它将自动包含该Snap-In。

要获取所有可用的SharePoint Cmdlet列表，请运行以下命令：

```
Get-Command -Module Microsoft.SharePoint.PowerShell
```

第64.2节：遍历站点集合中的所有列表

打印所有列表名称及其项目数量。

```
$site = Get-SPSite -Identity https://mysharepointsite/sites/test
foreach ($web in $site.AllWebs)
{
    foreach ($list in $web.Lists)
    {
        # 打印列表标题和项目数量
        Write-Output "$($list.Title), 项目数: $($list.ItemCount)"
    }
}
$site.Dispose()
```

第64.3节：获取站点集合上所有已安装的功能

```
Get-SPFeature -Site https://mysharepointsite/sites/test
```

Get-SPFeature 也可以在网站范围（-Web <WebUrl>）、农场范围（-Farm）和 Web 应用程序范围（-WebApplication <WebAppUrl>）运行。

获取站点集合上所有孤立功能

Get-SPFeature 的另一个用法是查找所有没有范围的功能：

```
Get-SPFeature -Site https://mysharepointsite/sites/test |? { $_.Scope -eq $null )
```

Chapter 64: SharePoint Module

Section 64.1: Loading SharePoint Snap-In

Loading the SharePoint Snapin can be done using the following:

```
Add-PSSnapin "Microsoft.SharePoint.PowerShell"
```

This only works in the 64bit version of PowerShell. If the window says "Windows PowerShell (x86)" in the title you are using the incorrect version.

If the Snap-In is already loaded, the code above will cause an error. Using the following will load only if necessary, which can be used in Cmdlets/functions:

```
if ((Get-PSSnapin "Microsoft.SharePoint.PowerShell" -ErrorAction SilentlyContinue) -eq $null)
{
    Add-PSSnapin "Microsoft.SharePoint.PowerShell"
}
```

Alternatively, if you start the SharePoint Management Shell, it will automatically include the Snap-In.

To get a list of all the available SharePoint Cmdlets, run the following:

```
Get-Command -Module Microsoft.SharePoint.PowerShell
```

Section 64.2: Iterating over all lists of a site collection

Print out all list names and the item count.

```
$site = Get-SPSite -Identity https://mysharepointsite/sites/test
foreach ($web in $site.AllWebs)
{
    foreach ($list in $web.Lists)
    {
        # Prints list title and item count
        Write-Output "$($list.Title), Items: $($list.ItemCount)"
    }
}
$site.Dispose()
```

Section 64.3: Get all installed features on a site collection

```
Get-SPFeature -Site https://mysharepointsite/sites/test
```

Get-SPFeature can also be run on web scope (-Web <WebUrl>), farm scope (-Farm) and web application scope (-WebApplication <WebAppUrl>).

Get all orphaned features on a site collection

Another usage of Get-SPFeature can be to find all features that have no scope:

```
Get-SPFeature -Site https://mysharepointsite/sites/test |? { $_.Scope -eq $null )
```

第65章：Psake简介

第65.1节：基本概述

```
任务 重建 -依赖 清理, 构建  {
    "重建"
 }

任务 构建 {
    "构建"
 }

任务 清理 {
    "清理"
 }

默认任务 -依赖 构建
```

第65.2节：FormatTaskName 示例

```
# 将显示任务为：
# -------- 重建 --------
# -------- 构建 --------
FormatTaskName "-------- {0} --------"

# 将以黄色显示任务：
# 正在运行重建
FormatTaskName {
param($taskName)
"运行 $taskName" - 前景色 黄色
}

任务 重建 -依赖 清理, 构建  {
"重建"
}

任务 构建 {
"构建"
}

任务 清理 {
"清理"
}

任务 默认 -依赖 构建
```

第65.3节：有条件地运行任务

```
属性 {
     $isOk = $false
}

# 默认情况下，构建任务不会运行，除非有参数为 $true
任务 构建 -前提条件 { 返回 $isOk } {
    "构建"
 }
```

Chapter 65: Introduction to Psake

Section 65.1: Basic outline

```
Task Rebuild -Depends Clean, Build  {
    "Rebuild"
 }

Task Build {
    "Build"
 }

Task Clean {
    "Clean"
 }

Task default -Depends Build
```

Section 65.2: FormatTaskName example

```
# Will display task as:
# -------- Rebuild --------
# -------- Build --------
FormatTaskName "-------- {0} --------"

# will display tasks in yellow colour:
# Running Rebuild
FormatTaskName {
param($taskName)
"Running $taskName" - foregroundcolor yellow
}

Task Rebuild -Depends Clean, Build  {
"Rebuild"
}

Task Build {
"Build"
}

Task Clean {
"Clean"
}

Task default -Depends Build
```

Section 65.3: Run Task conditionally

```
propreties {
     $isOk = $false
}

# By default the Build task won't run, unless there is a param $true
Task Build -precondition { return $isOk } {
    "Build"
 }
```

```
任务 清理 {
   "清理"
 }

默认任务 -依赖 构建
```

第65.4节：遇错继续

```
任务 Build -依赖于 Clean {
   "Build"
 }

任务 Clean -继续出错{
   "Clean"
    抛出 "故意抛出，但任务将继续运行"
 }

默认任务 -依赖 构建
```

```
Task Clean {
   "Clean"
 }

Task default -Depends Build
```

Section 65.4: ContinueOnError

```
Task Build -depends Clean {
   "Build"
 }

Task Clean -ContinueOnError {
   "Clean"
    throw "throw on purpose, but the task will continue to run"
 }

Task default -Depends Build
```

第66章：Pester简介

第66.1节：Pester入门

要开始使用Pester模块对PowerShell代码进行单元测试，您需要熟悉三个关键字/命令：

- Describe：定义一组测试。所有Pester测试文件至少需要一个Describe块。
- It：定义一个单独的测试。您可以在Describe块中包含多个It块。
- Should：验证/测试命令。用于定义应被视为成功测试的结果。

示例：

```
Import-Module Pester

#用于运行测试的示例函数
function Add-Numbers{
    param($a, $b)
    return [int]$a + [int]$b
}

#测试组
Describe "验证 Add-Numbers" {

        #单个测试用例
It "应该将 2 + 2 相加等于 4" {
            Add-Numbers 2 2 | 应该是 4
        }

It "应该处理字符串" {
            Add-Numbers "2" "2" | 应该是 4
        }

It "应该返回整数"{
            Add-Numbers 2.3 2 | 应该是 Int32 类型
        }

}
```

输出：

```
描述 验证 Add-Numbers
[+] 应该将 2 + 2 相加等于 4 33毫秒
[+] 应该处理字符串 19毫秒
[+] 应返回整数 23ms
```

Chapter 66: Introduction to Pester

Section 66.1: Getting Started with Pester

To get started with unit testing PowerShell code using the Pester-module, you need to be familiar with three keywords/commands:

- **Describe**: Defines a group of tests. All Pester test files needs at least one Describe-block.
- **It**: Defines an individual test. You can have multiple It-blocks inside a Describe-block.
- **Should**: The verify/test command. It is used to define the result that should be considered a successful test.

Sample:

```
Import-Module Pester

#Sample function to run tests against
function Add-Numbers{
    param($a, $b)
    return [int]$a + [int]$b
}

#Group of tests
Describe "Validate Add-Numbers" {

        #Individual test cases
        It "Should add 2 + 2 to equal 4" {
            Add-Numbers 2 2 | Should Be 4
        }

        It "Should handle strings" {
            Add-Numbers "2" "2" | Should Be 4
        }

        It "Should return an integer"{
            Add-Numbers 2.3 2 | Should BeOfType Int32
        }

}
```

Output:

```
Describing Validate Add-Numbers
[+] Should add 2 + 2 to equal 4 33ms
[+] Should handle strings 19ms
[+] Should return an integer 23ms
```

第67章：处理秘密和凭据

在Powershell中，为避免以明文存储密码，我们使用不同的加密方法并将其存储为安全字符串。当你未指定密钥或安全密钥时，这仅适用于同一用户在同一计算机上能够解密加密字符串，如果你'未使用密钥/安全密钥。任何在该相同用户账户下运行的进程都能在同一台机器上解密该加密字符串。

第67.1节：访问明文密码

凭据对象中的密码是加密的[SecureString]。最直接的方法是获取一个[NetworkCredential]，它不以加密形式存储密码：

```
$credential = Get-Credential
$plainPass = $credential.GetNetworkCredential().Password
```

辅助方法（.GetNetworkCredential()）仅存在于[PSCredential]对象上。
要直接处理[SecureString]，使用.NET方法：

```
$bstr = [System.Runtime.InteropServices.Marshal]::SecureStringToBSTR($secStr)
$plainPass = [System.Runtime.InteropServices.Marshal]::PtrToStringAuto($bstr)
```

第67.2节：提示输入凭据

要提示输入凭据，几乎总是应该使用Get-Credential命令。

```
$credential = Get-Credential
```

预填用户名：

```
$credential = Get-Credential -UserName 'myUser'
```

添加自定义提示信息：

```
$credential = Get-Credential -Message '请输入您的公司邮箱地址和密码。'
```

第67.3节：使用已存储的凭据

要轻松存储和检索加密凭据，请使用PowerShell内置的XML序列化（Clixml）：

```
$credential = Get-Credential

$credential | Export-CliXml -Path 'C:\My\Path\cred.xml'
```

重新导入：

```
$credential = Import-CliXml -Path 'C:\My\Path\cred.xml'
```

重要的是要记住，默认情况下，这使用的是 Windows 数据保护 API，且用于加密密码的密钥特定于运行代码的用户和计算机。

因此，加密的凭据无法被不同用户导入，也无法被同一用户在

Chapter 67: Handling Secrets and Credentials

In Powershell, to avoid storing the password in *clear text* we use different methods of encryption and store it as secure string. When you are not specifying a key or securekey, this will only work for the same user on the same computer will be able to decrypt the encrypted string if you're not using Keys/SecureKeys. Any process that runs under that same user account will be able to decrypt that encrypted string on that same machine.

Section 67.1: Accessing the Plaintext Password

The password in a credential object is an encrypted [SecureString]. The most straightforward way is to get a [NetworkCredential] which does not store the password encrypted:

```
$credential = Get-Credential
$plainPass = $credential.GetNetworkCredential().Password
```

The helper method (.GetNetworkCredential()) only exists on [PSCredential] objects.
To directly deal with a [SecureString], use .NET methods:

```
$bstr = [System.Runtime.InteropServices.Marshal]::SecureStringToBSTR($secStr)
$plainPass = [System.Runtime.InteropServices.Marshal]::PtrToStringAuto($bstr)
```

Section 67.2: Prompting for Credentials

To prompt for credentials, you should almost always use the Get-Credential cmdlet:

```
$credential = Get-Credential
```

Pre-filled user name:

```
$credential = Get-Credential -UserName 'myUser'
```

Add a custom prompt message:

```
$credential = Get-Credential -Message 'Please enter your company email address and password.'
```

Section 67.3: Working with Stored Credentials

To store and retrieve encrypted credentials easily, use PowerShell's built-in XML serialization (Clixml):

```
$credential = Get-Credential

$credential | Export-CliXml -Path 'C:\My\Path\cred.xml'
```

To re-import:

```
$credential = Import-CliXml -Path 'C:\My\Path\cred.xml'
```

The important thing to remember is that by default this uses the Windows data protection API, and the key used to encrypt the password is specific to both the *user and the machine* that the code is running under.

As a result, the encrypted credential cannot be imported by a different user nor the same user on a

不同的计算机上导入。

通过使用不同的运行用户和不同的计算机加密同一凭据的多个版本，
你可以让多个用户都能使用相同的秘密。

通过将用户名和计算机名放入文件名中，你可以以一种方式存储所有加密的秘密，
使相同的代码能够使用它们而无需硬编码任何内容：

加密器

```
# 以每个用户身份以及在每台计算机上运行

$credential = Get-Credential

$credential | Export-CliXml -Path "C:\My\Secrets\myCred_${env:USERNAME}_${env:COMPUTERNAME}.xml"
```

使用存储凭据的代码：

```
$credential = Import-CliXml -Path "C:\My\Secrets\myCred_${env:USERNAME}_${env:COMPUTERNAME}.xml"
```

运行用户的正确版本文件将自动加载（否则会失败，因为文件不存在）。

第67.4节：以加密形式存储凭据并在需要时作为参数传递

```
$username = "user1@domain.com"
$pwdTxt = Get-Content "C:emp\Stored_Password.txt"
$securePwd = $pwdTxt | ConvertTo-SecureString
$credObject = New-Object System.Management.Automation.PSCredential -ArgumentList $username,
$securePwd
# 现在，$credObject 已经存储了凭据，你可以在任何需要的地方传递它。


## 使用 AES 导入密码

$username = "user1@domain.com"
$AESKey = Get-Content $AESKeyFilePath
$pwdTxt = Get-Content $SecurePwdFilePath
$securePwd = $pwdTxt | ConvertTo-SecureString -Key $AESKey
$credObject = New-Object System.Management.Automation.PSCredential -ArgumentList $username,
$securePwd

# 现在，$credObject 已经使用 AES 密钥存储了凭据，你可以在任何需要的地方传递它。
```

different computer.

By encrypting several versions of the same credential with different running users and on different computers, you can have the same secret available to multiple users.

By putting the user and computer name in the file name, you can store all of the encrypted secrets in a way that allows for the same code to use them without hard coding anything:

Encrypter

```
# run as each user, and on each computer

$credential = Get-Credential

$credential | Export-CliXml -Path "C:\My\Secrets\myCred_${env:USERNAME}_${env:COMPUTERNAME}.xml"
```

The code that uses the stored credentials:

```
$credential = Import-CliXml -Path "C:\My\Secrets\myCred_${env:USERNAME}_${env:COMPUTERNAME}.xml"
```

The correct version of the file for the running user will be loaded automatically (or it will fail because the file doesn't exist).

Section 67.4: Storing the credentials in Encrypted form and Passing it as parameter when Required

```
$username = "user1@domain.com"
$pwdTxt = Get-Content "C:\temp\Stored_Password.txt"
$securePwd = $pwdTxt | ConvertTo-SecureString
$credObject = New-Object System.Management.Automation.PSCredential -ArgumentList $username,
$securePwd
# Now, $credObject is having the credentials stored and you can pass it wherever you want.


## Import Password with AES

$username = "user1@domain.com"
$AESKey = Get-Content $AESKeyFilePath
$pwdTxt = Get-Content $SecurePwdFilePath
$securePwd = $pwdTxt | ConvertTo-SecureString -Key $AESKey
$credObject = New-Object System.Management.Automation.PSCredential -ArgumentList $username,
$securePwd

# Now, $credObject is having the credentials stored with AES Key and you can pass it wherever you
want.
```

第68章：安全与密码学

第68.1节：通过.Net密码学计算字符串的哈希码

利用.Net System.Security.Cryptography.HashAlgorithm 命名空间，使用支持的算法生成消息哈希码。

```
$example="没有人预料到西班牙宗教裁判所。"

#计算
$hash=[System.Security.Cryptography.HashAlgorithm]::Create("sha256").ComputeHash(
[System.Text.Encoding]::UTF8.GetBytes($example))

#转换为十六进制
[System.BitConverter]::ToString($hash)

#2E-DF-DA-DA-56-52-5B-12-90-FF-16-FB-17-44-CF-B4-82-DD-29-14-FF-BC-B6-49-79-0C-0E-58-9E-46-2D-3D
```

“sha256”部分是所使用的哈希算法。

“-”可以被移除或转换为小写

```
#转换为无‘-’的小写十六进制
[System.BitConverter]::ToString($hash).Replace("-","").ToLower()

#2edfdada56525b1290ff16fb1744cfb482dd2914ffbcb649790c0e589e462d3d
```

如果偏好base64格式，则使用base64转换器进行输出

```
#转换为base64
[Convert]::ToBase64String($hash)

#Lt/a2lZSWxKQ/xb7F0TPtILdKRT/vLZJeQwOWJ5GLT0=
```

Chapter 68: Security and Cryptography

Section 68.1: Calculating a string's hash codes via .Net Cryptography

Utilizing .Net `System.Security.Cryptography.HashAlgorithm` namespace to generate the message hash code with the algorithms supported.

```
$example="Nobody expects the Spanish Inquisition."

#calculate
$hash=[System.Security.Cryptography.HashAlgorithm]::Create("sha256").ComputeHash(
[System.Text.Encoding]::UTF8.GetBytes($example))

#convert to hex
[System.BitConverter]::ToString($hash)

#2E-DF-DA-DA-56-52-5B-12-90-FF-16-FB-17-44-CF-B4-82-DD-29-14-FF-BC-B6-49-79-0C-0E-58-9E-46-2D-3D
```

The `"sha256"` part was the hash algorithm used.

the `-` can be removed or change to lower case

```
#convert to lower case hex without '-'
[System.BitConverter]::ToString($hash).Replace("-","").ToLower()

#2edfdada56525b1290ff16fb1744cfb482dd2914ffbcb649790c0e589e462d3d
```

If base64 format was preferred, using base64 converter for output

```
#convert to base64
[Convert]::ToBase64String($hash)

#Lt/a2lZSWxKQ/xb7F0TPtILdKRT/vLZJeQwOWJ5GLT0=
```

第69章：脚本签名

第69.1节：签名脚本

脚本签名是通过使用Set-AuthenticodeSignature命令和代码签名证书完成的。

```
#获取当前登录用户的第一个可用个人代码签名证书
$cert = @(Get-ChildItem -Path Cert:\CurrentUser\My -CodeSigningCert)[0]

#使用证书签名脚本
Set-AuthenticodeSignature -Certificate $cert -FilePath c:\MyScript.ps1
```

你也可以使用以下方式从.pfx文件读取证书：

```
$cert = Get-PfxCertificate -FilePath "C:\MyCodeSigningCert.pfx"
```

该脚本将在证书过期前有效。如果在签名时使用时间戳服务器，脚本将在证书过期后继续有效。添加证书的信任链（包括根证书）也很有用，这有助于大多数计算机信任用于签名脚本的证书。

```
Set-AuthenticodeSignature -Certificate $cert -FilePath c:\MyScript.ps1 -IncludeChain All -
TimeStampServer "http://timestamp.verisign.com/scripts/timstamp.dll"
```

建议使用来自可信证书提供商（如Verisign、Comodo、Thawte等）的时间戳服务器。

第69.2节：为单个脚本绕过执行策略

您可能经常需要执行不符合当前执行策略的未签名脚本。一个简单的方法是为该单个进程绕过执行策略。示例：

```
powershell.exe -ExecutionPolicy Bypass -File C:\MyUnsignedScript.ps1
```

或者您可以使用简写：

```
powershell -ep Bypass C:\MyUnsignedScript.ps1
```

其他执行策略：

政策	描述
AllSigned	只有由受信任发布者签名的脚本才能运行。
Bypass	无任何限制；所有 Windows PowerShell 脚本均可运行。
Default	通常为RemoteSigned，但由 ActiveDirectory 控制
RemoteSigned	下载的脚本必须由受信任发布者签名后才能运行。
Restricted	不允许运行任何脚本。Windows PowerShell 只能以交互模式使用。
Undefined	无
无限制*	类似于旁路

无限制 注意事项： 如果运行从互联网下载的未签名脚本，系统会在运行前提示您确认权限。*

更多信息请访问这里。

Chapter 69: Signing Scripts

Section 69.1: Signing a script

Signing a script is done by using the Set-AuthenticodeSignature-cmdlet and a code-signing certificate.

```
#Get the first available personal code-signing certificate for the logged on user
$cert = @(Get-ChildItem -Path Cert:\CurrentUser\My -CodeSigningCert)[0]

#Sign script using certificate
Set-AuthenticodeSignature -Certificate $cert -FilePath c:\MyScript.ps1
```

You can also read a certificate from a .pfx-file using:

```
$cert = Get-PfxCertificate -FilePath "C:\MyCodeSigningCert.pfx"
```

The script will be valid until the certificate expires. If you use a timestamp-server during the signing, the script will continue to be valid after the certificate expires. It is also useful to add the trust chain for the certificate (including root authority) to help most computers trust the certificated used to sign the script.

```
Set-AuthenticodeSignature -Certificate $cert -FilePath c:\MyScript.ps1 -IncludeChain All -
TimeStampServer "http://timestamp.verisign.com/scripts/timstamp.dll"
```

It's recommended to use a timestamp-server from a trusted certificate provider like Verisign, Comodo, Thawte etc.

Section 69.2: Bypassing execution policy for a single script

Often you might need to execute an unsigned script that doesn't comply with the current execution policy. An easy way to do this is by bypassing the execution policy for that single process. Example:

```
powershell.exe -ExecutionPolicy Bypass -File C:\MyUnsignedScript.ps1
```

Or you can use the shorthand:

```
powershell -ep Bypass C:\MyUnsignedScript.ps1
```

Other Execution Policies:

Policy	Description
AllSigned	Only scripts signed by a trusted publisher can be run.
Bypass	No restrictions; all Windows PowerShell scripts can be run.
Default	Normally RemoteSigned, but is controlled via ActiveDirectory
RemoteSigned	Downloaded scripts must be signed by a trusted publisher before they can be run.
Restricted	No scripts can be run. Windows PowerShell can be used only in interactive mode.
Undefined	NA
Unrestricted*	Similar to bypass

***Unrestricted* Caveat:** If you run an unsigned script that was downloaded from the Internet, you are prompted for permission before it runs.*

More Information available here.

第69.3节：使用 Set-ExecutionPolicy 更改执行策略

要更改默认作用域（LocalMachine）的执行策略，请使用：

```
Set-ExecutionPolicy AllSigned
```

要更改特定作用域的策略，请使用：

```
Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy AllSigned
```

您可以通过添加-Force开关来抑制提示。

第69.4节：获取当前执行策略

获取当前会话的有效执行策略：

```
PS> Get-ExecutionPolicy
RemoteSigned
```

列出当前会话的所有有效执行策略：

```
PS> Get-ExecutionPolicy -List

              Scope ExecutionPolicy
              ----- ---------------
      MachinePolicy       Undefined
         UserPolicy       Undefined
            Process       Undefined
        CurrentUser       Undefined
       LocalMachine    RemoteSigned
```

列出特定作用域的执行策略，例如：进程：

```
PS> Get-ExecutionPolicy -Scope Process
Undefined
```

第69.5节：获取已签名脚本的签名

使用Get-AuthenticodeSignature命令获取已签名脚本的Authenticode签名信息：

```
Get-AuthenticodeSignature .\MyScript.ps1 | Format-List *
```

第69.6节：创建用于测试的自签名代码签名证书

在签署个人脚本或测试代码签名时，创建自签名代码签名证书可能会很有用。

版本 ≥ 5.0

从 PowerShell 5.0 开始，您可以使用 New- 命令生成自签名代码签名证书：SelfSignedCertificate-cmdlet：

Section 69.3: Changing the execution policy using Set-ExecutionPolicy

To change the execution policy for the default scope (LocalMachine), use:

```
Set-ExecutionPolicy AllSigned
```

To change the policy for a specific scope, use:

```
Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy AllSigned
```

You can suppress the prompts by adding the -Force switch.

Section 69.4: Get the current execution policy

Getting the effective execution policy for the current session:

```
PS> Get-ExecutionPolicy
RemoteSigned
```

List all effective execution policies for the current session:

```
PS> Get-ExecutionPolicy -List

        Scope ExecutionPolicy
        ----- ---------------
MachinePolicy       Undefined
   UserPolicy       Undefined
      Process       Undefined
  CurrentUser       Undefined
 LocalMachine    RemoteSigned
```

List the execution policy for a specific scope, ex. process:

```
PS> Get-ExecutionPolicy -Scope Process
Undefined
```

Section 69.5: Getting the signature from a signed script

Get information about the Authenticode signature from a signed script by using the Get-AuthenticodeSignature-cmdlet:

```
Get-AuthenticodeSignature .\MyScript.ps1 | Format-List *
```

Section 69.6: Creating a self-signed code signing certificate for testing

When signing personal scripts or when testing code signing it can be useful to create a self-signed code signing certificate.

Version ≥ 5.0

Beginning with PowerShell 5.0 you can generate a self-signed code signing certificate by using the New-SelfSignedCertificate-cmdlet:

```
New-SelfSignedCertificate -FriendlyName "StackOverflow 示例代码签名" -CertStoreLocation
Cert:\CurrentUser\My -Subject "SO 用户" -Type CodeSigningCert
```

在早期版本中，您可以使用 .NET Framework SDK 和 Windows SDK 中的 makecert.exe 工具创建自签名证书。

自签名证书仅会被安装了该证书的计算机信任。对于将要共享的脚本，建议使用来自受信任证书颁发机构（内部或受信任第三方）的证书。

```
New-SelfSignedCertificate -FriendlyName "StackOverflow Example Code Signing" -CertStoreLocation
Cert:\CurrentUser\My -Subject "SO User" -Type CodeSigningCert
```

In earlier versions, you can create a self-signed certificate using the `makecert.exe` tool found in the .NET Framework SDK and Windows SDK.

A self-signed certificate will only be trusted by computers that have installed the certificate. For scripts that will be shared, a certificate from a trusted certificate authority (internal or trusted third-party) are recommended.

第70章：使用 PowerShell 匿名化文本文件中的 IP（IPv4 和 IPv6）

操作 IPv4 和 IPv6 的正则表达式，并在读取的日志文件中替换为伪造的 IP 地址

第70.1节：匿名化文本文件中的 IP 地址

```
# 读取文本文件并将IPv4和IPv6替换为虚假IP地址


# 描述所有变量
$SourceFile = "C:\sourcefile.txt"
$IPv4File = "C:\IPV4.txt"
$DestFile = "C:\ANONYM.txt"
$Regex_v4 = "(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})"
$Anonym_v4 = "XXX.XXX.XXX.XXX"
$Regex_v6 = "((([0-9A-Fa-f]{1,4}:){7}[0-9A-Fa-f]{1,4})|(([0-9A-Fa-f]{1,4}:){6}:[0-9A-Fa-
f]{1,4})|(([0-9A-Fa-f]{1,4}:){5}:([0-9A-Fa-f]{1,4}:)?[0-9A-Fa-f]{1,4})|(([0-9A-Fa-
f]{1,4}:){4}:([0-9A-Fa-f]{1,4}:){0,2}[0-9A-Fa-f]{1,4})|(([0-9A-Fa-f]{1,4}:){3}:([0-9A-Fa-
f]{1,4}:){0,3}[0-9A-Fa-f]{1,4})|(([0-9A-Fa-f]{1,4}:){2}:([0-9A-Fa-f]{1,4}:){0,4}[0-9A-Fa-
f]{1,4})|(([0-9A-Fa-
f]{1,4}:){6}((b((25[0-5])|(1d{2})|(2[0-4]d)|(d{1,2}))b).){3}(b((25[0-5])|(1d{2})|(2[0-4]d)|(d{1,2})
)b))|(([0-9A-Fa-
f]{1,4}:){0,5}:((b((25[0-5])|(1d{2})|(2[0-4]d)|(d{1,2}))b).){3}(b((25[0-5])|(1d{2})|(2[0-4]d)|(d{1,
2}))b))|(::([0-9A-Fa-
f]{1,4}:){0,5}((b((25[0-5])|(1d{2})|(2[0-4]d)|(d{1,2}))b).){3}(b((25[0-5])|(1d{2})|(2[0-4]d)|(d{1,2
}))b))|([0-9A-Fa-f]{1,4}::([0-9A-Fa-f]{1,4}:){0,5}[0-9A-Fa-f]{1,4})|(::([0-9A-Fa-
f]{1,4}:){0,6}[0-9A-Fa-f]{1,4})|(([0-9A-Fa-f]{1,4}:){1,7}:))"
$Anonym_v6 = "YYYY:YYYY:YYYY:YYYY:YYYY:YYYY:YYYY:YYYY"
$SuffixName = "-ANONYM."
$AnonymFile = ($Parts[0] + $SuffixName + $Parts[1])

# 从源文件替换匹配的IPv4并创建临时文件IPV4.txt
Get-Content $SourceFile | Foreach-Object {$_ -replace $Regex_v4, $Anonym_v4} | Set-Content
$IPv4File

# 从 IPV4.txt 中替换匹配的 IPv6 并创建临时文件 ANONYM.txt
Get-Content $IPv4File | Foreach-Object {$_ -replace $Regex_v6, $Anonym_v6} | Set-Content $DestFile

# 删除临时 IPV4.txt 文件
Remove-Item $IPv4File

# 将 ANONYM.txt 重命名为 sourcefile-ANONYM.txt
$Parts = $SourceFile.Split(".")
If (Test-Path $AnonymFile)
{
    Remove-Item $AnonymFile
    Rename-Item $DestFile -NewName $AnonymFile
    }
    Else
    {
    Rename-Item $DestFile -NewName $AnonymFile
}
```

Chapter 70: Anonymize IP (v4 and v6) in text file with PowerShell

Manipulating Regex for IPv4 and IPv6 and replacing by fake IP address in a readed log file

Section 70.1: Anonymize IP address in text file

```
# Read a text file and replace the IPv4 and IPv6 by fake IP Address


# Describe all variables
$SourceFile = "C:\sourcefile.txt"
$IPv4File = "C:\IPV4.txt"
$DestFile = "C:\ANONYM.txt"
$Regex_v4 = "(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})"
$Anonym_v4 = "XXX.XXX.XXX.XXX"
$Regex_v6 = "((([0-9A-Fa-f]{1,4}:){7}[0-9A-Fa-f]{1,4})|(([0-9A-Fa-f]{1,4}:){6}:[0-9A-Fa-
f]{1,4})|(([0-9A-Fa-f]{1,4}:){5}:([0-9A-Fa-f]{1,4}:)?[0-9A-Fa-f]{1,4})|(([0-9A-Fa-
f]{1,4}:){4}:([0-9A-Fa-f]{1,4}:){0,2}[0-9A-Fa-f]{1,4})|(([0-9A-Fa-f]{1,4}:){3}:([0-9A-Fa-
f]{1,4}:){0,3}[0-9A-Fa-f]{1,4})|(([0-9A-Fa-f]{1,4}:){2}:([0-9A-Fa-f]{1,4}:){0,4}[0-9A-Fa-
f]{1,4})|(([0-9A-Fa-
f]{1,4}:){6}((b((25[0-5])|(1d{2})|(2[0-4]d)|(d{1,2}))b).){3}(b((25[0-5])|(1d{2})|(2[0-4]d)|(d{1,2})
)b))|(([0-9A-Fa-
f]{1,4}:){0,5}:((b((25[0-5])|(1d{2})|(2[0-4]d)|(d{1,2}))b).){3}(b((25[0-5])|(1d{2})|(2[0-4]d)|(d{1,
2}))b))|(::([0-9A-Fa-
f]{1,4}:){0,5}((b((25[0-5])|(1d{2})|(2[0-4]d)|(d{1,2}))b).){3}(b((25[0-5])|(1d{2})|(2[0-4]d)|(d{1,2
}))b))|([0-9A-Fa-f]{1,4}::([0-9A-Fa-f]{1,4}:){0,5}[0-9A-Fa-f]{1,4})|(::([0-9A-Fa-
f]{1,4}:){0,6}[0-9A-Fa-f]{1,4})|(([0-9A-Fa-f]{1,4}:){1,7}:))"
$Anonym_v6 = "YYYY:YYYY:YYYY:YYYY:YYYY:YYYY:YYYY:YYYY"
$SuffixName = "-ANONYM."
$AnonymFile = ($Parts[0] + $SuffixName + $Parts[1])

# Replace matching IPv4 from sourcefile and creating a temp file IPV4.txt
Get-Content $SourceFile | Foreach-Object {$_ -replace $Regex_v4, $Anonym_v4} | Set-Content
$IPv4File

# Replace matching IPv6 from IPV4.txt and creating a temp file ANONYM.txt
Get-Content $IPv4File | Foreach-Object {$_ -replace $Regex_v6, $Anonym_v6} | Set-Content $DestFile

# Delete temp IPV4.txt file
Remove-Item $IPv4File

# Rename ANONYM.txt in sourcefile-ANONYM.txt
$Parts = $SourceFile.Split(".")
If (Test-Path $AnonymFile)
{
    Remove-Item $AnonymFile
    Rename-Item $DestFile -NewName $AnonymFile
    }
    Else
    {
    Rename-Item $DestFile -NewName $AnonymFile
}
```

第71章：亚马逊网络服务（AWS）Rekognition

亚马逊 Rekognition 是一项服务，使您可以轻松地将图像分析添加到您的应用程序中。使用 Rekognition，您可以检测图像中的物体、场景和人脸。您还可以搜索和比较人脸。Rekognition's API 使您能够快速将基于深度学习的复杂视觉搜索和图像分类添加到您的应用程序中。

第71.1节：使用 AWS Rekognition 检测图像标签

```
$BucketName = 'trevorrekognition'
$FileName = 'kitchen.jpg'

New-S3Bucket -BucketName $BucketName
Write-S3Object -BucketName $BucketName -File $FileName
$REKResult = Find-REKLabel -Region us-east-1 -ImageBucket $BucketName -ImageName $FileName

$REKResult.Labels
```

运行上述脚本后，您应该会在 PowerShell 主机中看到类似以下内容的结果打印：

```
结果：

置信度 名称
---------- ----
86.87605    室内


77.4853     厨房
77.25354    住宅
77.25354    阁楼
66.77325    家电
66.77325    烤箱
```

使用 AWS PowerShell 模块结合 AWS Rekognition 服务，您可以检测图像中的标签，例如识别房间内的物体、您拍摄的照片属性，以及 AWS Rekognition 对每个属性对应的置信度。

Find-REKLabel 命令使您能够调用搜索这些属性/标签。虽然您可以在 API 调用时以字节数组形式提供图像内容，但更好的方法是将图像文件上传到 AWS S3 存储桶，然后让 Rekognition 服务指向您想要分析的 S3 对象。上面的示例展示了如何实现这一点。

第 71.2 节：使用 AWS Rekognition 进行面部相似度比较

```
$BucketName = 'trevorrekognition'

### 创建一个新的 AWS S3 存储桶
New-S3Bucket -BucketName $BucketName

### 将我自己的两张不同照片上传到 AWS S3 存储桶
Write-S3Object -BucketName $BucketName -File myphoto1.jpg
Write-S3Object -BucketName $BucketName -File myphoto2.jpg
```

Chapter 71: Amazon Web Services (AWS) Rekognition

Amazon Rekognition is a service that makes it easy to add image analysis to your applications. With Rekognition, you can detect objects, scenes, and faces in images. You can also search and compare faces. Rekognition's API enables you to quickly add sophisticated deep learning-based visual search and image classification to your applications.

Section 71.1: Detect Image Labels with AWS Rekognition

```
$BucketName = 'trevorrekognition'
$FileName = 'kitchen.jpg'

New-S3Bucket -BucketName $BucketName
Write-S3Object -BucketName $BucketName -File $FileName
$REKResult = Find-REKLabel -Region us-east-1 -ImageBucket $BucketName -ImageName $FileName

$REKResult.Labels
```

After running the script above, you should have results printed in your PowerShell host that look something similar to the following:

```
RESULTS:

Confidence Name
---------- ----
86.87605   Indoors
86.87605   Interior Design
86.87605   Room
77.4853    Kitchen
77.25354   Housing
77.25354   Loft
66.77325   Appliance
66.77325   Oven
```

Using the AWS PowerShell module in conjunction with the AWS Rekognition service, you can detect labels in an image, such as identifying objects in a room, attributes about photos you took, and the corresponding confidence level that AWS Rekognition has for each of those attributes.

The Find-REKLabel command is the one that enables you to invoke a search for these attributes / labels. While you can provide image content as a byte array during the API call, a better method is to upload your image files to an AWS S3 Bucket, and then point the Rekognition service over to the S3 Objects that you want to analyze. The example above shows how to accomplish this.

Section 71.2: Compare Facial Similarity with AWS Rekognition

```
$BucketName = 'trevorrekognition'

### Create a new AWS S3 Bucket
New-S3Bucket -BucketName $BucketName

### Upload two different photos of myself to AWS S3 Bucket
Write-S3Object -BucketName $BucketName -File myphoto1.jpg
Write-S3Object -BucketName $BucketName -File myphoto2.jpg
```

```
### 使用 AWS Rekognition 对这两张照片进行面部比较
$Comparison = @{
SourceImageBucket = $BucketName
    TargetImageBucket = $BucketName
    SourceImageName = 'myphoto1.jpg'
    TargetImageName = 'myphoto2.jpg'
    Region = 'us-east-1'
}
$Result = Compare-REKFace @Comparison
$Result.FaceMatches
```

上面提供的示例脚本应当给出类似以下的结果：

```
Face                                   Similarity
----                                   ----------
Amazon.Rekognition.Model.ComparedFace 90
```

AWS Rekognition 服务使您能够对两张照片进行面部比较。使用此服务非常简单。只需将您想要比较的两张图像文件上传到 AWS S3 存储桶中。然后，调用Compare-REKFace命令，类似于上面提供的示例。当然，您需要提供自己全局唯一的 S3 存储桶名称和文件名。

```
### Perform a facial comparison between the two photos with AWS Rekognition
$Comparison = @{
    SourceImageBucket = $BucketName
    TargetImageBucket = $BucketName
    SourceImageName = 'myphoto1.jpg'
    TargetImageName = 'myphoto2.jpg'
    Region = 'us-east-1'
}
$Result = Compare-REKFace @Comparison
$Result.FaceMatches
```

The example script provided above should give you results similar to the following:

```
Face                                  Similarity
----                                  ----------
Amazon.Rekognition.Model.ComparedFace 90
```

The AWS Rekognition service enables you to perform a facial comparison between two photos. Using this service is quite straightforward. Simply upload two image files, that you want to compare, to an AWS S3 Bucket. Then, invoke the Compare-REKFace command, similar to the example provided above. Of course, you'll need to provide your own, globally-unique S3 Bucket name and file names.

第72章：亚马逊网络服务（AWS）简单存储服务（S3）

参数	详情
桶名称	您正在操作的AWS S3桶的名称。
预定义ACL名称	关联到S3桶的内置（预定义）访问控制列表（ACL）的名称。 S3桶。
文件	本地文件系统中将上传到AWS S3桶的文件名称。

本节文档重点介绍针对亚马逊网络服务（AWS）简单存储服务（S3）的开发。S3确实是一个简单易用的服务。您可以创建S3“桶”，每个桶可以包含零个或多个“对象”。创建桶后，您可以将文件或任意数据作为“对象”上传到S3桶中。您通过对象的“键”（名称）来引用桶内的S3对象。

第72.1节：创建新的S3桶

```
New-S3Bucket -BucketName trevor
```

简单存储服务（S3）桶名称必须是全局唯一的。这意味着如果其他人已经使用了你想使用的桶名称，那么你必须选择一个新的名称。

第72.2节：将本地文件上传到S3桶

```
Set-Content -Path myfile.txt -Value 'PowerShell Rocks'
Write-S3Object -BucketName powershell -File myfile.txt
```

使用Write-S3Object命令，可以轻松地将本地文件系统中的文件上传到AWS S3。在最基本的形式中，你只需指定-BucketName参数，表示你想上传文件的S3桶，以及-File参数，表示你想上传到S3桶的本地文件的相对或绝对路径。

第72.3节：删除S3桶

```
Get-S3Object -BucketName powershell | Remove-S3Object -Force
Remove-S3Bucket -BucketName powershell -Force
```

为了删除S3桶，你必须先删除存储在桶内的所有S3对象，前提是你有权限这样做。在上面的示例中，我们检索了桶内所有对象的列表，然后将它们通过管道传递给Remove-S3Object命令进行删除。一旦所有对象被删除，我们就可以使用Remove-S3Bucket命令删除该桶。

Chapter 72: Amazon Web Services (AWS) Simple Storage Service (S3)

Parameter	Details
BucketName	The name of the AWS S3 bucket that you are operating on.
CannedACLName	The name of the built-in (pre-defined) Access Control List (ACL) that will be associated with the S3 bucket.
File	The name of a file on the local filesystem that will be uploaded to an AWS S3 Bucket.

This documentation section focuses on developing against the Amazon Web Services (AWS) Simple Storage Service (S3). S3 is truly a simple service to interact with. You create S3 "buckets" which can contain zero or more "objects." Once you create a bucket, you can upload files or arbitrary data into the S3 bucket as an "object." You reference S3 objects, inside of a bucket, by the object's "key" (name).

Section 72.1: Create a new S3 Bucket

```
New-S3Bucket -BucketName trevor
```

The Simple Storage Service (S3) bucket name must be globally unique. This means that if someone else has already used the bucket name that you want to use, then you must decide on a new name.

Section 72.2: Upload a Local File Into an S3 Bucket

```
Set-Content -Path myfile.txt -Value 'PowerShell Rocks'
Write-S3Object -BucketName powershell -File myfile.txt
```

Uploading files from your local filesystem into AWS S3 is easy, using the Write-S3Object command. In its most basic form, you only need to specify the -BucketName parameter, to indicate which S3 bucket you want to upload a file into, and the -File parameter, which indicates the relative or absolute path to the local file that you want to upload into the S3 bucket.

Section 72.3: Delete a S3 Bucket

```
Get-S3Object -BucketName powershell | Remove-S3Object -Force
Remove-S3Bucket -BucketName powershell -Force
```

In order to remove a S3 bucket, you must first remove all of the S3 objects that are stored inside of the bucket, provided you have permission to do so. In the above example, we are retrieving a list of all the objects inside a bucket, and then piping them into the Remove-S3Object command to delete them. Once all of the objects have been removed, we can use the Remove-S3Bucket command to delete the bucket.

鸣谢

非常感谢所有来自Stack Overflow Documentation的人员，他们帮助提供了这些内容，更多更改可以发送至web@petercv.com，以发布或更新新内容

亚当·M.	第23章
ajb101	第51章
阿尔班	第25章
安德烈·埃普雷	第28章
阿尼尔	第52章
安东尼·尼斯	第3章和第8章
AP.	第69章
奥斯汀·T·弗伦奇	第17章
autosvet	第1、2和20章
阿夫沙龙	第24章
伯特·莱夫劳	第12、27和43章
boeprox	第13章
布兰特·鲍比	第1章、第13章、第19章和第58章
briantist	第17章和第67章
camilohe	第27章
克里斯·N	第1章和第11章
克里斯托夫	第53章
克里斯托弗·G·刘易斯	第7章
克莱斯特斯	第1、3和26章
CmdrTchort	第7和57章
DarkLite1	第1和22章
戴夫·安德森	第22章
DAXaholic	第1章和第7章
Deptor	第25章
djwork	第11章
Eris	第2章、第7章和第27章
Euro Micelli	第5章
Florian Meyer	第10章和第60章
FoxDeploy	第1章
弗罗德·F.	第7、8、9、13、15、21、28、29、32、35、38、39、40、62、66和69章
乔治奥·甘比诺	第29章
朱利奥·卡钦	第55章
戈登·贝尔	第1章和第3章
格雷格·布雷	第1章
HAL9256	第30章
它	第1章
詹姆斯·拉斯金	第12、25和54章
杰奎琳·瓦内克	第13章
jimmyb	第23章
JNYRanger	第1章
JPBlanc	第3章
jumbo	第7、8、19、27、33、34、38和42章
基思	第25章
科洛布峡谷	第15章
拉奇·怀特	第63章
利亚姆	第2和6章
利文·凯尔斯马克斯	第29章

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content, more changes can be sent to web@petercv.com for new content to be published or updated

Adam M.	Chapter 23
ajb101	Chapter 51
Alban	Chapter 25
Andrei Epure	Chapter 28
ANIL	Chapter 52
Anthony Neace	Chapters 3 and 8
AP.	Chapter 69
Austin T French	Chapter 17
autosvet	Chapters 1, 2 and 20
Avshalom	Chapter 24
Bert Levrau	Chapters 12, 27 and 43
boeprox	Chapter 13
Brant Bobby	Chapters 1, 13, 19 and 58
briantist	Chapters 17 and 67
camilohe	Chapter 27
Chris N	Chapters 1 and 11
Christophe	Chapter 53
Christopher G. Lewis	Chapter 7
Clijsters	Chapters 1, 3 and 26
CmdrTchort	Chapters 7 and 57
DarkLite1	Chapters 1 and 22
Dave Anderson	Chapter 22
DAXaholic	Chapters 1 and 7
Deptor	Chapter 25
djwork	Chapter 11
Eris	Chapters 2, 7 and 27
Euro Micelli	Chapter 5
Florian Meyer	Chapters 10 and 60
FoxDeploy	Chapter 1
Frode F.	Chapters 7, 8, 9, 13, 15, 21, 28, 29, 32, 35, 38, 39, 40, 62, 66 and 69
Giorgio Gambino	Chapter 29
Giulio Caccin	Chapter 55
Gordon Bell	Chapters 1 and 3
Greg Bray	Chapter 1
HAL9256	Chapter 30
It	Chapter 1
James Ruskin	Chapters 12, 25 and 54
Jaqueline Vanek	Chapter 13
jimmyb	Chapter 23
JNYRanger	Chapter 1
JPBlanc	Chapter 3
jumbo	Chapters 7, 8, 19, 27, 33, 34, 38 and 42
Keith	Chapter 25
Kolob Canyon	Chapter 15
Lachie White	Chapter 63
Liam	Chapters 2 and 6
Lieven Keersmaekers	Chapter 29

劳埃德	第6章
卢克·瑞安	第12章
马德尼兹	第39章
马克·沃拉格	第1、3和56章
马修·布伊松	第1章和第11章
mattnicola	第26章和第39章
megamorf	第11章、第22章和第24章
梅尔特·居尔索伊	第13章
迈克·谢泼德	第14章
miken32	第6章
莫尔瓦尔德	第19章和第24章
motcke	第17章
Mrk	第1章
尼基尔·瓦尔塔克	第12章和第41章
NooJ	第23章和第70章
Poorkenny	第1章、第43章和第44章
普拉吉斯·萨拉瓦南	第2章、第16章和第47章
拉纳迪普·杜塔	第10、13、21和67章
RapidCoder	第54章
Raziel	第64章
restless1987	第2和9章
理查德	第7、12、26、29和49章
罗曼	第7章、第19章和第65章
罗希	第29章
萨姆·马丁	第1、24、30、45、59和66章
Schwarzie2478	第58章
SeeuD1	第7章
ShaneC	第24章
StephenP	第7章
史蒂夫·K	第2章
TessellatingHeckler	第7章和第12章
th1rdey3	第1章
TheIncorrigible1	第7章
tjrobinson	第1章
TravisEz13	第1、4、5、6、8、10、12、14、21、26、36和48章
特雷弗·沙利文	第14、18、50、61、71和72章
Venkatakrishnan	第31章
VertigoRay	第7章和第46章
void	第9章
vonPryz	第1章
W1M0R	第37章
Xalorous	第1章、第12章和第40章
Xenophane	第17章
xvorsx	第13章
xXhRQ8sD2L7Z	第21章
陆一驰	第30章和第68章

lloyd	Chapter 6
Luke Ryan	Chapter 12
Madniz	Chapter 39
Mark Wragg	Chapters 1, 3 and 56
Mathieu Buisson	Chapters 1 and 11
mattnicola	Chapters 26 and 39
megamorf	Chapters 11, 22 and 24
Mert Gülsoy	Chapter 13
Mike Shepard	Chapter 14
miken32	Chapter 6
Moerwald	Chapters 19 and 24
motcke	Chapter 17
Mrk	Chapter 1
Nikhil Vartak	Chapters 12 and 41
NooJ	Chapters 23 and 70
Poorkenny	Chapters 1, 43 and 44
Prageeth Saravanan	Chapters 2, 16 and 47
Ranadip Dutta	Chapters 10, 13, 21 and 67
RapidCoder	Chapter 54
Raziel	Chapter 64
restless1987	Chapters 2 and 9
Richard	Chapters 7, 12, 26, 29 and 49
Roman	Chapters 7, 19 and 65
Rowshi	Chapter 29
Sam Martin	Chapters 1, 24, 30, 45, 59 and 66
Schwarzie2478	Chapter 58
SeeuD1	Chapter 7
ShaneC	Chapter 24
StephenP	Chapter 7
Steve K	Chapter 2
TessellatingHeckler	Chapters 7 and 12
th1rdey3	Chapter 1
TheIncorrigible1	Chapter 7
tjrobinson	Chapter 1
TravisEz13	Chapters 1, 4, 5, 6, 8, 10, 12, 14, 21, 26, 36 and 48
Trevor Sullivan	Chapters 14, 18, 50, 61, 71 and 72
Venkatakrishnan	Chapter 31
VertigoRay	Chapters 7 and 46
void	Chapter 9
vonPryz	Chapter 1
W1M0R	Chapter 37
Xalorous	Chapters 1, 12 and 40
Xenophane	Chapter 17
xvorsx	Chapter 13
xXhRQ8sD2L7Z	Chapter 21
YChi Lu	Chapters 30 and 68

你可能也喜欢

You may also like