任务一　　函数

(一)　命名

在 PowerShell 中命名函数时，请使用 Pascal 大小写名称和经过批准的动词和单数名词。建议可以在名词前面加上前缀。

例如：<ApprovedVerb>-<Prefix><SingularNoun>。

在 PowerShell 中，可以通过运行 Get-Verb 命令来获得一个特定的已批准的动词列表。

```
Get-Verb | Sort-Object -Property Verb


# 输出
Verb             Group
----             -----
Add              Common
Approve          Lifecycle
Assert           Lifecycle
Backup           Data
Block            Security
Checkpoint       Data
Clear            Common
Close            Common
Compare          Data
Complete         Lifecycle
Compress         Data
Confirm          Lifecycle
Connect          Communications
Convert          Data
ConvertFrom      Data
```

ConvertTo	Data
Copy	Common
Debug	Diagnostic
Deny	Lifecycle
Disable	Lifecycle
Disconnect	Communications
Dismount	Data
Edit	Data
Enable	Lifecycle
Enter	Common
Exit	Common
Expand	Data
Export	Data
Find	Common
Format	Common
Get	Common
Grant	Security
Group	Data
Hide	Common
Import	Data
Initialize	Data
Install	Lifecycle
Invoke	Lifecycle
Join	Common
Limit	Data
Lock	Common
Measure	Diagnostic
Merge	Data
Mount	Data
Move	Common

New	Common
Open	Common
Optimize	Common
Out	Data
Ping	Diagnostic
Pop	Common
Protect	Security
Publish	Data
Push	Common
Read	Communications
Receive	Communications
Redo	Common
Register	Lifecycle
Remove	Common
Rename	Common
Repair	Diagnostic
Request	Lifecycle
Reset	Common
Resize	Common
Resolve	Diagnostic
Restart	Lifecycle
Restore	Data
Resume	Lifecycle
Revoke	Security
Save	Data
Search	Common
Select	Common
Send	Communications
Set	Common
Show	Common

```
Skip            Common
Split           Common
Start           Lifecycle
Step            Common
Stop            Lifecycle
Submit          Lifecycle
Suspend         Lifecycle
Switch          Common
Sync            Data
Test            Diagnostic
Trace           Diagnostic
Unblock         Security
Undo            Common
Uninstall       Lifecycle
Unlock          Common
Unprotect       Security
Unpublish       Data
Unregister      Lifecycle
Update          Data
Use             Other
Wait            Lifecycle
Watch           Common
Write           Communications
```

在上面的示例中，根据 Verb 列对结果进行了排序。Group 让你了解如何使用这些动词。 将函数添加到模块时，请务必在 PowerShell 中选择一个经过批准的动词。如果选择未经批准的动词，模块会在加载时生成一条警告消息。该警告消息会使函数看起来不专业。未经批准的动词还会限制函数的可发现性。

（二） 基础函数

PowerShell 中的函数使用函数关键字声明，后面依次跟函数名称、左大括号和右大括号。 函数将执行的代码包含在这些大括号中。

```
function Get-Version {
    $PSVersionTable.PSVersion
}
```

显示的函数是返回 PowerShell 版本的简单示例。

```
Get-Version

# 输出
Major  Minor  Build  Revision
-----  -----  -----  --------
5      1      14393  693
```

函数的名称很可能会与名称类似的函数 Get-Version 和 Powershell 中的默认命令或其他人可能编写的命令发生冲突。所以说会建议在函数的单数名词部分添加上前缀以帮助防止命名冲突。在下面的示例中，可以名词前添加上前缀“PS”。

```
function Get-PSVersion {
    $PSVersionTable.PSVersion
}
```

除了名称不一样外，函数功能与前一个相同。

```
Get-PSVersion

# 输出
Major  Minor  Build  Revision
-----  -----  -----  --------
```

```
5          1        14393       693
```

即使在名词前加上 PS 之类的前缀，仍然很有可能发生名称冲突。通常可以用首字母作为函数名词的前缀。

```
function Get-MrPSVersion {
    $PSVersionTable.PSVersion
}
```

此函数的功能与前两个示例的函数相同，只是使用了一个更合理化的名词，以避免与其他 PowerShell 命令发生命名冲突。

```
Get-MrPSVersion

# 输出
Major  Minor  Build  Revision
-----  -----  -----  --------
5          1        14393       693
```

一旦加载到内存后，就可在函数 PSDrive 上查看函数。

```
Get-ChildItem -Path Function:\Get-*Version

# 输出
CommandType     Name                Version    Source
-----------     ----                -------    ------
Function        Get-PSVersion
```

如果要从当前会话中删除这些函数，则必须从函数 PSDrive 中将其删除，或关闭之后再重新打开 PowerShell。

```
Get-ChildItem -Path Function:\Get-*Version | Remove-Item
```

验证是否确实删除了函数。

```
Get-ChildItem -Path Function:\Get-*Version
```

如果函数是作为模块的一部分加载的，则可以卸载模块以删除它们。

```
Remove-Module -Name <ModuleName>
```

Remove-Module 命令从当前 PowerShell 会话中的内存中删除模块，不会从系统或磁盘中删除模块。

(三) 参数

在函数中请勿静态地赋值！ 使用参数和变量。 在为参数命名时，尽可能使用与默认命令相同的名称去作为参数名。

```
function Test-MrParameter {
    param (
        $ComputerName
    )
    Write-Output $ComputerName
}
```

这里为什么使用 ComputerName 而不是使用 Computer、ServerName 或 Host 作为参数名称呢？ 这是因为希望函数能像默认命令一样具有标准化。

这里将创建一个函数，用于查询系统上的所有命令并返回具有特定参数名称的命令的数量。

```
function Get-MrParameterCount {
    param (
        [string[]]$ParameterName
    )
```

```
        foreach ($Parameter in $ParameterName) {
            $Results = Get-Command -ParameterName $Parameter -
ErrorAction SilentlyContinue

            [pscustomobject]@{
                ParameterName = $Parameter
                NumberOfCmdlets = $Results.Count
            }
        }
    }
```

正如下面的结果所示，有 39 个命令具有 ComputerName 参数。 没有任何命令 具有 Computer、ServerName、Host 或 Machine 这样的参数 。

```
    Get-MrParameterCount -ParameterName ComputerName, Computer,
ServerName, Host, Machine

    # 输出
    ParameterName             NumberOfCmdlets
    -------------               ---------------
    ComputerName                  39
    Computer                      0
    ServerName                    0
    Host                          0
    Machine                       0
```

建议对参数名称使用与默认命令相同的大小写。 使用 ComputerName，而不是 computername。 这会使函数看起来和感觉起来都像默认的命令。已经熟悉 PowerShell 的用户来使用会感到得心应手。

这个 param 语句允许定义一个或多个参数。参数定义之间使用逗号 (,) 进行分隔。

（四） 进阶函数

将 PowerShell 中的函数转换为高级函数非常简单。 函数与高级函数之间的区别之一是高级函数具有多个自动添加到函数中的公共参数。这些常用参数包括 Verbose 和 Debug。

接着从上一部分中使用的 Test-MrParameter 函数开始介绍。

```
function Test-MrParameter {
    param (
        $ComputerName
    )
    Write-Output $ComputerName
}
```

请注意，Test-MrParameter 函数没有任何公共参数。 可以通过多种不同的方法来查看通用参数。其中一种方法就是使用 Get-Command 来查看语法。

```
Get-Command -Name Test-MrParameter -Syntax


# 输出
Test-MrParameter [[-ComputerName] <Object>]
```

另一种方法是通过使用 Get-Command 更深入的来了解参数。

```
(Get-Command -Name Test-MrParameter).Parameters.Keys


# 输出
ComputerName
```

添加 CmdletBinding 来将函数转换为高级函数。

```
function Test-MrCmdletBinding {
    [CmdletBinding()]    #<<-- 这将常规函数转换为高级函数
    param (
        $ComputerName
    )
    Write-Output $ComputerName
}
```

添加 CmdletBinding 会自动添加公共参数。CmdletBinding 需要一个 param 块，但 param 块可以为空。

```
Get-Command -Name Test-MrCmdletBinding -Syntax

# 输出
Test-MrCmdletBinding [[-ComputerName] <Object>]
[<CommonParameters>]
```

通过 Get-Command 更深入的了解参数会显示实际参数名称，包括常见参数名称。

```
(Get-Command -Name Test-MrCmdletBinding).Parameters.Keys

# 输出
ComputerName
Verbose
Debug
ErrorAction
WarningAction
InformationAction
ErrorVariable
WarningVariable
```

```
InformationVariable
OutVariable
OutBuffer
PipelineVariable
```

（五） SupportsShouldProcess

SupportsShouldProcess 会添加 WhatIf 和 Confirm 参数。只有进行更改的命令才会需要这些参数，这些参数通常与使用动词 Set 和 Remove 的命令一起使用，但参数不会限制于特定动词。

```
function Test-MrSupportsShouldProcess {
    [CmdletBinding(SupportsShouldProcess)]
    param (
        $ComputerName
    )

    Write-Output $ComputerName
}
```

请注意，现在有 WhatIf 和 Confirm 参数 。

```
Get-Command -Name Test-MrSupportsShouldProcess -Syntax

# 输出
Test-MrSupportsShouldProcess [[-ComputerName] <Object>] [-WhatIf] [-Confirm] [<CommonParameters>]
```

同样，还可以使用 Get-Command 返回实际参数名称列表，其中包括常见参数名称以及 WhatIf 和 Confirm。

```
(Get-Command -Name Test-MrSupportsShouldProcess).Parameters.Keys


# 输出
ComputerName
Verbose
Debug
ErrorAction
WarningAction
InformationAction
ErrorVariable
WarningVariable
InformationVariable
OutVariable
OutBuffer
PipelineVariable
WhatIf
Confirm
```

(六) 参数验证

尽早验证输入。如果不可能在没有有效输入的情况下去运行代码，那为什么允许代码在某路径上继续运行？

始终键入要用于参数的变量（指定数据类型）。

```
function Test-MrParameterValidation {
    [CmdletBinding()]
    param (
        [string]$ComputerName
    )
    Write-Output $ComputerName
```

```
    }
```

在上面的示例中，指定了 String 作为 ComputerName 参数的数据类型 。这将导致它只允许指定一个计算机名。如果通过以逗号分隔的列表指定了多个计算机名，则会生成报错。

```
    Test-MrParameterValidation -ComputerName Server01, Server02


    # 输出
    Test-MrParameterValidation : Cannot process argument transformation on
parameter
    'ComputerName'. Cannot convert value to type System.String.
    At line:1 char:42
    + Test-MrParameterValidation -ComputerName Server01, Server02
    +
        + CategoryInfo                  : InvalidData: (:) [Test-
MrParameterValidation], ParameterBindingArg
          umentTransformationException
        + FullyQualifiedErrorId :
ParameterArgumentTransformationError,Test-MrParameterValidation
```

当前定义的问题在于省略 ComputerName 参数的值是有效的，但需要一个值才能成功完成该函数。这时 Mandatory 参数属性就开始发挥作用了。

```
    function Test-MrParameterValidation {
        [CmdletBinding()]
        param (
            [Parameter(Mandatory)]
            [string]$ComputerName
        )
        Write-Output $ComputerName
    }
```

上面的示例中使用的语法为 PowerShell 版本 3.0 及更高兼容版本。 可以改为指定 [Parameter(Mandatory=$true)]，以使函数与 PowerShell 版本 2.0 及更高版本兼容。 现在 ComputerName 是必需的，如果未指定，该函数将提示输入名称。

```
Test-MrParameterValidation


# 输出
cmdlet Test-MrParameterValidation at command pipeline position 1
Supply values for the following parameters:
ComputerName:
```

如果要允许 ComputerName 参数具有多个值，请使用 String 数据类型，但应向该数据类型添加左方括号和右方括号以允许字符串数组 。

```
function Test-MrParameterValidation {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory)]
        [string[]]$ComputerName
    )
    Write-Output $ComputerName
}
```

如果未指定 ComputerName 参数，你可能希望指定一个默认值。问题在于，默认值不能与必需参数一起使用。相反，你需要使用带有默认值的 ValidateNotNullOrEmpty 参数验证属性。

```
function Test-MrParameterValidation {
    [CmdletBinding()]
    param (
        [ValidateNotNullOrEmpty()]
        [string[]]$ComputerName = $env:COMPUTERNAME
```

```
        )
        Write-Output $ComputerName
    }
```

即使在设置默认值时，也不要使用静态值。 在上面的示例中，使用 $env:COMPUTERNAME 作为默认值，如果没有提供值，该默认值将自动转换为本地计算机名。

(七) 详细输出

虽然内联注释很有用，尤其是在编写一些复杂的代码时，但除非用户查看代码本身，否则它们永远不会被用户看到。

下面的示例中显示的函数在 foreach 循环中有一个内联注释。 虽然这个特定的注释可能并不难找到，但想象一下，如果函数包含数百行代码，就会很麻烦。

```
function Test-MrVerboseOutput {
    [CmdletBinding()]
    param (
        [ValidateNotNullOrEmpty()]
        [string[]]$ComputerName = $env:COMPUTERNAME
    )
    foreach ($Computer in $ComputerName) {
        #Attempting to perform some action on $Computer <<-- Don't use
        #inline comments like this, use write verbose instead.
        Write-Output $Computer
    }
}
```

更好的选择是使用 Write-Verbose 而不是内联注释。

```
function Test-MrVerboseOutput {
    [CmdletBinding()]
    param (
```

```
            [ValidateNotNullOrEmpty()]
            [string[]]$ComputerName = $env:COMPUTERNAME
        )
        foreach ($Computer in $ComputerName) {
            Write-Verbose -Message "Attempting to perform some action on
$Computer"
            Write-Output $Computer
        }
    }
```

在不带 Verbose 参数的情况下调用函数时，则不会显示详细输出。

```
Test-MrVerboseOutput -ComputerName Server01, Server02
```

通过 Verbose 参数调用函数时，会显示详细输出。

```
Test-MrVerboseOutput -ComputerName Server01, Server02 -Verbose
```

（八） 管道输出

如果希望函数能接受管道输入，则需要进行一些额外的编码。 如我们前面所提到的，命令可以接受按值（按类型）或按属性名称接受的管道输入。可以像编写本地命令那样去编写函数，以便它们接受其中一种或两种类型的输入。

若要按值接受管道输入，请为该特定参数指定 ValueFromPipeline 参数属性。要记住，只能接受来自每种数据类型之一的值的管道输入。例如，如果有两个接受字符串输入的参数，则只有其中一个参数可以接受按值显示的管道输入，因为如果为两个字符串参数都指定了管道输入，则管道输入会不知道该绑定到哪个参数。 这就是按类型而不是按值来调用这种类型的管道输入的另一个原因。

管道输入一次输入一个项，类似于在 foreach 循环中处理项的方式。如果要接受数组作为输入，则至少需要一个 process 块才能处理所有项。 如果只想接

受单个值作为输入，则不需要 process 块，但为了保持一致性，建议指定它。

```
function Test-MrPipelineInput {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory,
                   ValueFromPipeline)]
        [string[]]$ComputerName
    )
    PROCESS {
        Write-Output $ComputerName
    }
}
```

通过属性名称接受管道输入与此类似，只是它是由使用 ValueFromPipelineByPropertyName 参数属性指定的，并且它可以指定任意数量的参数指定，无需考虑数据类型。 关键在于，通过管道输入的命令的输出必须具有与参数名称或函数的参数别名匹配的属性名称。

```
function Test-MrPipelineInput {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory,
                   ValueFromPipelineByPropertyName)]
        [string[]]$ComputerName
    )
    PROCESS {
            Write-Output $ComputerName
    }
}
```

BEGIN 和 END 块是可选的。 BEGIN 在 PROCESS 块之前指定，用于在

从管道接收项之前执行任何初始工作。理解这一点很重要。通过管道输入的值在 BEGIN 块中不可访问。 END 块将被指定在 PROCESS 块之后，并在处理完所有通过管道传入的项目后用于清理。

（九） 错误处理

当无法联系计算机时，以下示例中显示的函数会生成未经处理的异常。

```
function Test-MrErrorHandling {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory,
                   ValueFromPipeline,
                   ValueFromPipelineByPropertyName)]
        [string[]]$ComputerName
    )
    PROCESS {
        foreach ($Computer in $ComputerName) {
            Test-WSMan -ComputerName $Computer
        }
    }
}
```

可采用多种不同的方法在 PowerShell 中处理错误。 Try/Catch 是处理错误的较新式的方法。

```
function Test-MrErrorHandling {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory,
                   ValueFromPipeline,
```

```
                    ValueFromPipelineByPropertyName)]
            [string[]]$ComputerName
        )
        PROCESS {
            foreach ($Computer in $ComputerName) {
                try {
                    Test-WSMan -ComputerName $Computer
                }
                catch {
                    Write-Warning -Message "Unable to connect to
Computer: $Computer"
                }
            }
        }
    }
```

尽管上面的示例中显示的函数使用错误处理，但它也会生成未经处理的异常，因为该命令不会生成终止错误。理解这一点也很重要。仅捕获终止错误。指定以 Stop 为值的 ErrorAction 参数，将非终止错误转换为终止错误。

```
    function Test-MrErrorHandling {
        [CmdletBinding()]
        param (
            [Parameter(Mandatory,
                    ValueFromPipeline,
                    ValueFromPipelineByPropertyName)]
            [string[]]$ComputerName
        )
        PROCESS {
            foreach ($Computer in $ComputerName) {
```

```
                    try {
                        Test-WSMan -ComputerName $Computer -ErrorAction
Stop
                    }
                    catch {
                        Write-Warning -Message "Unable to connect to
Computer: $Computer"
                    }
                }
            }
        }
```

除非绝对必要，否则不要修改全局 $ErrorActionPreference 变量。 如果直接从 PowerShell 函数内使用 .NET 之类的内容，则不能针对命令本身指定 ErrorAction。在这种情况下，你可能需要更改全局 $ErrorActionPreference 变量，但如果确实要更改它，请在尝试执行命令后立即将其更改回来。

(十) 基于注释的帮助

最佳做法是将基于注释的帮助添加到函数，以便你与之共享函数的人知道如何使用它们。

```
    function Get-MrAutoStoppedService {


    <#
    .SYNOPSIS
        Returns a list of services that are set to start automatically, are not
        currently running, excluding the services that are set to delayed start.


    .DESCRIPTION
        Get-MrAutoStoppedService is a function that returns a list of services
```

```
from
        the specified remote computer(s) that are set to start automatically, are
not
        currently running, and it excludes the services that are set to start
automatically
        with a delayed startup.

    .PARAMETER ComputerName
        The remote computer(s) to check the status of the services on.

    .PARAMETER Credential
        Specifies a user account that has permission to perform this action. The
default
        is the current user.

    .EXAMPLE
         Get-MrAutoStoppedService -ComputerName 'Server1', 'Server2'

    .EXAMPLE
         'Server1', 'Server2' | Get-MrAutoStoppedService

    .EXAMPLE
         Get-MrAutoStoppedService -ComputerName 'Server1' -Credential
(Get-Credential)

    .INPUTS
        String

    .OUTPUTS
        PSCustomObject
```

```
    .NOTES
        Author:    Mike F Robbins
        Website: http://mikefrobbins.com
        Twitter: @mikefrobbins
    #>

        [CmdletBinding()]
        param (

        )

        #Function Body
    }
```

如果将基于注释的帮助添加到函数，可以像默认的内置命令一样为他们检索帮助。

用于在 PowerShell 中编写函数的所有语法似乎过于复杂，尤其是对于刚入门的用户。 通常，如果我不记得语法的某些内容，我会在单独的监视器上打开 ISE 的第二个副本，并在键入函数代码时查看“命令 (高级函数) - 完成”代码片段。 可以使用 Ctrl+J 组合键在 PowerShell ISE 中访问代码片段。