一、 函数

任务一 详细描述

函数是具有分配名称的 PowerShell 语句列表。运行函数时，键入的函数名称。列表中的语句就像在命令提示符下键入它们一样运行。

```
function Get-PowerShellProcess { Get-Process PowerShell }
```

函数也可以像命令或应用程序一样复杂。

(一) 语法

以下是函数的语法：

```
function [<scope:>]<name> [([type]$parameter1[,[type]$parameter2])]
{
  begin {<statement list>}
  process {<statement list>}
  end {<statement list>}
  clean {<statement list>}
}
```

```
function [<scope:>]<name>
{
  param([type]$parameter1 [,[type]$parameter2])
  dynamicparam {<statement list>}
  begin {<statement list>}
  process {<statement list>}
```

```
    end {<statement list>}
    clean {<statement list>}
}
```

一个函数包括以下项目：

- 一个 function 关键词
- 范围（可选）
- 你选择的名称
- 任意数量的命名参数（可选）
- 一个或多个用大括号括起来的 PowerShell 命令{}

带参数的函数

我们可以将参数与函数一起使用，包括命名参数、位置参数、开关参数和动态参数。

命名参数

我们可以定义任意数量的命名参数。可以包括命名参数的默认值。

可以使用关键字在大括号内定义参数 param。

语法示例：

```
function <name> {
    param ([type]$parameter1 [,[type]$parameter2])
    <statement list>
}
```

还可以在没有关键字的情况下在大括号外去定义参数 Param。

语法示例：

```
function <name> [([type]$parameter1[,[type]$parameter2])] {
    <statement list>
}
```

实际示例展示：

```
function Add-Numbers([int]$one, [int]$two) {
    $one + $two
}
```

虽然首选第一种方式，但这两种方式之间没有区别。

运行这个函数的时候，为参数提供的值将分配给包含参数名称的变量。这个变量的值可以在函数中进行使用。

下面的示例是一个名为 Get-SmallFiles 的函数。这个函数有一个$Size 参数，这个函数显示所有小于$Size 参数值的文件，并且不包括目录。

```
function Get-SmallFiles {
   Param($Size)
   Get-ChildItem $HOME | Where-Object {
      $_.Length -lt $Size -and !$_.PSIsContainer
   }
}
```

在函数中，可以使用变量$Size，它是为参数定义的名称。

使用此函数，

```
Get-SmallFiles -Size 50
```

还可以为不带参数名称的命名参数输入值。例如，以下命令给出的结果与指定 Size 参数的命令相同：

```
Get-SmallFiles 50
```

要定义参数的默认值，在参数名后面输入等号和值即可。

```
function Get-SmallFiles ($Size = 100) {
```

```
    Get-ChildItem $HOME | Where-Object {
      $_.Length -lt $Size -and !$_.PSIsContainer
    }
  }
```

这时，如果输入 Get-SmallFiles 没有值，这个函数会将 100 分配给$Size 参数值。如果提供一个值，函数将会使用这个值。

或者，也可以提供一个简短的帮助字符串来描述参数的默认值，方法是将 PSDefaultValue 属性添加到参数的描述中，并指定 PSDefaultValue 的帮助属性。要提供一个帮助字符串来描述 Get-SmallFiles 函数中 Size 参数的默认值(100),添加 PSDefaultValue 属性， 如下面的示例所示。

```
  function Get-SmallFiles {
    param (
        [PSDefaultValue(Help = '100')]
        $Size = 100
    )
  }
```

(二) 位置参数

位置参数是没有参数名称的参数。PowerShell 中使用参数值顺序将每个参数值与函数中的一个参数相关联。

使用位置参数时，需在函数名称后输入一个或多个值。位置参数值被分配给 $args 数组变量。函数名称后面的值被分配给$args 数组中的第一个位置，$args[0]。

下面名为 Get-Extension 的函数将.txt 结尾的文件扩展名添加到参数提供的文件名中：

```
  function Get-Extension {
    $name = $args[0] + ".txt"
    $name
```

```
}
```

执行效果：

```
Get-Extension myTextFile


# 输出
myTextFile.txt
```

（三） 开关参数

开关是一个不需要值的参数。相反，输入函数名后跟开关参数名。

在定义 switch 参数时，在参数名称前指定类型[switch]。

示例：

```
function Switch-Item {
   param ([switch]$on)
   if ($on) { "Switch on" }
   else { "Switch off" }
}
```

当在函数名称后输入 On 开关参数的时候，该函数将显示 Switch on。如果没有 switch 参数，则显示 Switch off。

```
Switch-Item -on


# 输出
Switch on
```

```
Switch-Item
```

```
# 输出
Switch off
```

也可以在运行函数时给开关赋予一个布尔值来控制。

示例：

```
Switch-Item -on:$true

# 输出
Switch on
```

```
Switch-Item -on:$false

# 输出
Switch off
```

（四） 使用 Splatting 表示命令参数

我们可以使用 splatting 来表示命令的参数。Windows PowerShell 3.0 中引入了此功能。

在调用会话中的命令的函数中使用此技术。不需要声明或枚举命令参数，也不需要在命令参数更改时更改函数。

下面的示例函数调用了 Get-Command 命令。该命令使用了@Args 表示 Get-Command 的参数。

```
function Get-MyCommand { Get-Command @Args }
```

在调用 Get-MyCommand 函数时，可以使用 Get-Command 的所有参数。参数和参数值使用@Args 传递给命令。

```
Get-MyCommand -Name Get-ChildItem
```

```
# 输出
CommandType     Name                    ModuleName
-----------     ----                    ----------
Cmdlet          Get-ChildItem           Microsoft.PowerShell.Management
```

@Args 特性使用$Args 自动参数，它表示未声明的命令参数和来自参数的值。

NOTE

如果使用了 CmdletBinding 或 Parameter 属性将函数变成高级函数，则 $args 自动变量将不在函数中可用。高级函数需要明确的参数定义。

（五） 管道对象到函数

任何函数都可以从管道中获取输入。可以使用 begin、process、end 和 clean 等关键字去控制函数如何处理来自管道的输入。

语法示例：

```
function <name> {
   begin {<statement list>}
   process {<statement list>}
   end {<statement list>}
   clean {<statement list>}
}
```

NOTE

如果函数定义了这些命名语句列表中的任何一个，则所有代码都哦必须在这些块之一中。在块外的任何代码都无法识别。如果在函数中不适用这些块中的任何一个，则所有的语句都将被视为 end 语句列表。

语句列表 begin 只在函数的开头运行一次。

语句列表 process 为管道中的每个对象运行一次。当 process 块运行时，每个管道对象被分配给$_自动变量，一次分配一个管道对象。

在函数接收到管道中的所有对象后，语句列表 end 运行一次。

下面的函数使用 process 关键字。该函数显示来自管道的值：

```
function Get-Pipeline
{
   process {"The value is: $_"}
}
```

执行效果：

```
1,2,4 | Get-Pipeline


# 输出
The value is: 1
The value is: 2
The value is: 4
```

当在管道中使用函数时，管道传递给到函数的对象被分配给$input 自动变量。该函数在任何来自管道的对象之前运行带有 begin 关键字的语句。该函数在从管道接收到所有对象后运行带有 end 关键字的语句。

下面的示例展示了带有 begin 和 end 关键字的$input 自动变量。

```
function Get-PipelineBeginEnd
{
      begin {"Begin: The input is $input"}
      end   {"End:   The input is $input"}
}
```

如果使用管道运行此函数，它会显示以下结果：

```
1,2,4 | Get-PipelineBeginEnd


# 输出
Begin: The input is
End:     The input is 1 2 4
```

当 begin 语句运行时，函数没有来自管道的输入。end 语句在函数具有值之后运行。

如果函数有 process 关键字，$input 中的每个对象都会从$input 中移除并赋值给$_。下面的示例有一个 process 语句列表：

```
function Get-PipelineInput
{
      process {"Processing:   $_ " }
      end       {"End:     The input is: $input" }
}
```

在示例中，每个通过管道传递给函数的对象都被发送到 process 语句列表。这些 process 语句在每个对象上运行，一次一个对象。当函数到达 end 关键字时，$input 自动变量为空。

```
1,2,4 | Get-PipelineInput


# 输出
Processing:   1
Processing:   2
Processing:   4
End:     The input is:
```

（六） Filters 过滤器

Filter 过滤器是一种在管道中的每个对象上运行的函数。过滤器类似于一个函数，其所有语句都在一个 process 块中。

语法示例：

```
filter [<scope:>]<name> {<statement list>}
```

下面的 filter 从管道中获取日志条目，然后显示整个条目或仅显示条目的消息部分：

```
filter Get-ErrorLog ([switch]$Message)
{
  if ($Message) { Out-Host -InputObject $_.Message }
  else { $_ }
}
```

执行方式：

```
Get-WinEvent -LogName System -MaxEvents 100 | Get-ErrorLog -Message
```

（七） 函数范围

一个函数存在于它被创建的范围中。

如果函数是脚本的一部分，则该函数可用于该脚本中的语句。默认情况下，脚本中的函数在该脚本之外不可用。

可以指定函数的作用域。例如，在以下示例中将函数添加到范围：

```
function global:Get-DependentSvs {
  Get-Service | Where-Object {$_.DependentServices}
}
```

当函数处于全局范围中时，可以在脚本、函数和命令行中使用该函数。

函数创建一个新的范围。在函数中创建的项(如变量)仅存在于函数作用域中。