5 命令稿撰寫語言：速成課程

在本書當中，讀者們會注意到，只有在剛使用過某些功能之後，才會看到相關的一般資訊。不過在這裡筆者要破例一下。在本書篇幅中你都會撰寫命令稿，這意味著其中會有一定數量的程式碼。PowerShell 的命令稿撰寫語言非常簡單，只包含了約數十個真正的關鍵字，而筆者在本書中只會用到其中十來個。但是，我們必須把其中最要緊的內容深植腦中，才能在時機到來時派上用場。本章的目標並非完整地涵蓋這些項目，而是要為讀者們很快地介紹一遍。這樣當你在本書隨後的篇幅中見到它們的時候，才容易進入狀況。

提示 為了進一步了解本章中的內容，第一個要研究之處便是 PowerShell 的說明系統。大部分的內容皆記錄在 About topics 當中。譬如，你不妨看看 `about_if` 和 `about_comparison_operators` 的相關資訊。也可以去弄一本《*PowerShell in Depth*》第二版（2014 年 Manning 出版；http://mng.bz/xjzq）。

5.1 比較

我們在本章中介紹的幾乎所有撰寫命令稿的點滴，都要用到**比較**的功能。也就是你提供一些必須評估是否為 `True` 或 `False` 的敘述，而命令稿撰寫的結構（constructs）便會依照評估結果採取行動。要在 PowerShell 裡做比較，就要用到**比較運算子**（*comparison operator*）。PowerShell 不像傳統的命令稿撰寫或程式設計語言，它不使用傳統運算子的字元（<、>、+、=、-

等等），而是改用英文的縮寫。以下便是在本書中四處可見的核心運算子：

- `-eq`——等於
- `-ne`——不等於
- `-gt`——大於
- `-ge`——大於或等於
- `-lt`——小於
- `-le`——小於或等於

至於字串的比較，預設是不計較字母大小寫的，亦即 `"Hello"` 和 `"HELLO"` 會被視為相等的。如果你刻意要分出大小寫字母的區別，就要在運算子名稱前面加上一個 `c`，變成 `-ceq` 或 `-cne`。

當你使用這些運算子時，PowerShell 會傳回 `True/False` 之一的值：

```
PS C:\> 1 -eq 1
True
PS C:\> 5 -gt 10
False
PS C:\> 'James' -eq 'Jim'
False
PS C:\> 'James' -eq 'james'
True
PS C:\> 'james' -ceq 'James'
False
```

與某些語言相較，PowerShell 擁有範圍更廣泛的運算子。譬如說，沒有一種「正好相等」的比較會去限制 shell 的剖析功能去強制轉換資料型別。

☞ 現在試試看

`5 -eq "Five"` 的真偽如何？

5.1.1 萬用字元

此外也有所謂的萬用字元比較，即 -like 和 -notlike，以及會區分大小寫字母的 -clike 和 -cnotlike。它們都允許你在比較字串時使用通用的萬用字元，譬如 *（零個或多個字元）和 ?（單一字元）：

```
PS C:\> 'james' -eq 'Jim'
False
PS C:\> 'james' -eq 'James'
True
PS C:\> 'james' -ceq 'James'
False
PS C:\> 'PowerShell'-like '*shell'
True
PS C:\> 'james' -notlike 'james*'
False
PS C:\> 'james' -like 'jam?s'
True
PS C:\> 'james' -like 'Jim'
False
```

這些萬用字元自然不如全套的正規表示式（regular-expression）語言那麼厲害；不過 PowerShell 的確也能以 -match 運算子支援正規表示式，然而我們在本書中不會對此多所著墨。請參閱《*PowerShell in Depth*》第二版（2014 年 Manning 出版；http://mng.bz/xjzq）一書當中關於 PowerShell 與正規表示式的章節。

5.1.2 集合

PowerShell 的 -contains 和 -in 運算子可以對一群物件的集合進行比較運算。這些功能有時會很棘手，人們總是把它們和萬用字元運算子混淆。譬如常會看到這種的：

```
If ("DC" -in $ServerList) {
  $IsDomainController = $True
}
```

這無法像你想像般運作。從英文文法來看似乎說得通，但運算子不是靠文法運作的。如果你先從陣列物件著手，就可以用這些運算子來決定，該陣列（或集合）是否含有特定物件：

```
$array = @("one", "two", "three")
$array -contains "one"
$array -contains "five"
"two" -in $array
"bob" -in $array
```

☞ 現在試試看

在 PowerShell 裡執行以上五行程式碼，一次輸入一行並按下 Enter，觀察結果。

5.1.3 為比較除錯

大概有一半的命令稿錯誤是比較的結果不如預期所引起的。我們誠懇地建議，為了進行除錯，這時最好先別管命令稿，而是回到 PowerShell 控制台，先試著把比較運算式的部分執行一遍。

☞ 現在試試看

`"55" -eq 55` 的結果為何？我們先不講答案——請自己試試看能否解釋為何會有如此結果、還有它做了什麼。

5.2 If 結構

你會經常用到 `If` 結構，透過它你可以進行邏輯決策。一個 `If` 結構的格式會像這樣：

```
If (<expression>) {
    # 你的程式碼
} ElseIf (<expression>) {
    # 你的程式碼
} ElseIf (<expression>) {
    # 你的程式碼
} Else {
    # 你的程式碼
}
```

你應該要知道：

- <expression> 可以是任何 PowerShell 表示式，只要其結果是 $True 或 $False 即可。譬如，如果變數 $something 真的等於 5，$something -eq 5 就會是 $True。請參閱 PowerShell 的「about_comparison_operators」一文，看看有哪些比較運算子可用，像是 -eq、-ne、-gt、-like 等等。
- 在你的 If 敘述裡，表示式的部分怎麼複雜都無所謂。只需記住，整段表示式的結果必須為 True，尾隨的命令稿區塊內的程式碼才能得以執行：

```
$now = Get-Date
if ($now.DayOfWeek -eq 'Monday' -AND $now.hour -gt 18) {
  # 進行某些動作
}
```

- 結構中的 If 部分是必要的，而且一定要尾隨一個 {script block}，並在表示式結果為 True 時執行。
- 你可以不使用 ElseIf 段落、或是一次用好幾個也可以。這些段落含有它們自己的表示式和命令稿區塊，並在表示式結果為 True 時執行。但是請記住一個重點：只有第一個表示式結果為 True 的命令稿區塊才會得以執行。因此以上例中的骨架來看，如果第一個表示式結果為 True，那便只有第一個命令稿的區塊會執行；剩下的 ElseIf 裡的表示式連進行評估的機會都不會有。如果你有好幾個 ElseIf 敘述，PowerShell 會逐個地評估它們，直到發現第一個結果為 True 的敘述為止。這時 PowerShell 會跳到整塊 If 結構的後段部分去進行指令。
- 你還可以在結尾加上 Else 段落。它定義的命令稿區塊，會在之前的所有表示式結果都不是 True 時才會執行。
- 這裡沒有像其他程式語言的 End If 敘述。
- 在上例的骨架中，你會發現有幾行以 # 符號起頭。它們代表註解。PowerShell 會忽略位於 # 之後、直到該行結束為止的所有內容。

PowerShell 不太計較格式。譬如我們會認為結構這樣寫很不錯：

```
If (<expression>) {
    # 你的程式碼
} ElseIf (<expression>) {
    # 你的程式碼
} ElseIf (<expression>) {
    # 你的程式碼
} Else {
    # 你的程式碼
}
```

但也有人會覺得應該把左大括號 { 獨立成一行。沒有哪一種是絕對正確的，但我們只想要你知道，有些人會這樣寫：

```
If (<expression>)
{
    # 你的程式碼
}
```

然而 PowerShell 也容許這種寫法：

```
If (<expression>) { # 你的程式碼 }
ElseIf (<expression>) { # 你的程式碼 }
ElseIf (<expression>) { # 你的程式碼 }
Else { # 你的程式碼 }
```

這個比較不好閱讀，特別是當有些命令稿區塊的程式碼必須跨越好幾行的時候。當然我們並不是建議你採用這種寫法，但你還是有機會看到有人這樣做。底線是，雖然 PowerShell 不介意，你卻應該遵照他人的最佳實施慣例。請選擇讓你的程式碼易於閱讀的寫法，然後始終如一地保持下去。

快速提示

程式碼的格式很要緊。看起來這似乎是個無關緊要的美學觀點問題，但它卻使得你的程式碼更容易閱讀，亦即臭蟲會比較少。信不信由你。請看這個錯誤示範：

```
If ($user) { ForEach ($u in $user) {
Set-ADUser -Identity $user -Pass $True }
```

你很難看出以上程式碼是否有效（其實是有問題的），問題出在大括號的放置、還有 ForEach 跟 If 放在同一行等方式上。

如果你用了一個功能出色的編輯器，像是 Visual Studio（VS Code），它輕易就能讓你的程式碼保持簡潔：讓它自行運作就好。當你開啟一段具
有 { 的結構、並按下 Enter 時，VS Code 便會自動加上一個 }，然後把指標——而且完美地縮排四個空白字元——移到結構當中。請讓 VS Code 完成工作——譬如說，當你需要縮排某一行時，按下 Tab 鍵，而不要按下空格鍵。此外，如果你還未這樣做，記得去打開 VS Code 裡的括號上色功能。

如果內容對齊的排列效果不彰，這裡有一個技巧：把受影響的區域標定起來（標定整段命令稿文件也行），點擊滑鼠右鍵、並選擇 Format Selection。VS Code 會清理並正確地縮排每一個結構段落內部。你也可以打開 Command Palette、點選 Format Document，把整份文件重新排版。

我們來看一個這種結構的實際例子。假設你在 $proc 變數中有一個 Process 物件，而你想要在該程序的虛擬記憶體（**virtual memory**，VM）屬性超過特定預定值時採取某些行動：

```
If ($proc.vm -gt 4) {
  # 採取某些行動
}
```

注意，以上我們加上了註解——記住，任何位於 # 符號之後、直到該行結束的部分，都會被忽略——藉以指出採取行動的程式碼位於何處。如果你想改成 VM 的值低於 2、或高於 4 時觸發動作，怎麼辦？

```
If ($proc.vm -gt 4 -or $proc.vm -lt 2) {
  # 採取某些行動
}
```

布林運算子 -or 允許你「串接」兩個條件。注意位於 -or 兩側的比較式都必須是**完整的**。像下例便會行不通：

```
If ($proc.vm -gt 4 -or -lt 2) {
  # 採取某些行動
}
```

在這個「錯誤示範」當中，右側的「小於」比較式不完整。亦即小於比較式的左半部不完整；PowerShell 會疑問「到底是什麼東西要小於 2 ？」、然後拋出一筆錯誤訊息。如果方便，你應該用小括號把個別比較式框住，以便一望即知：

```
If ( ($proc.vm -gt 4) -or ($proc.vm -lt 2) ) {
  # 採取某些行動
}
```

接著再看一個還有其他選項的例子：

```
If ($proc.vm -gt 4) {
  # 採取某些行動
} ElseIf ($proc.vm -lt 2) {
  # 採取某些行動
} Else {
  # 以上皆非 true；故而改成這樣做
}
```

如前所述，PowerShell 將會執行第一個評估為 `True` 時的條件式轄下動作、然後便不再評估其餘的部分。

5.3 ForEach 結構

大家經常都會用到 `ForEach` 結構，它有時也被稱為**迭代器**（*enumerator*）。`ForEach` 在多數程式設計語言中都很常見，因此大家應該都耳熟能詳。其運作與 PowerShell 的 `ForEach-Object` 指令頗為相似，但語法略有出入：

```
ForEach ($item in $collection) {
    # 針對參照 $item 當中所含的每一個物件執行此處的程式碼
}
```

此處的觀念在於拿一個由物件構成的集合或陣列，然後一次一個地檢視各個物件。每個物件在輪到時就會被放入另一個變數當中，這樣你就可以隨

意地引用它。當你迭代完了集合或陣列中所有物件以後，迴圈便會自動跳出，繼續執行你命令稿中的其餘部分。

位於結構當中的第二個變數 $collection，預計其中會含有零個或多個項目。ForEach 的迴圈會針對第二個變數中所含的每一個項目分別去執行一次隨後的 {script block}。也就是說，如果你在 $collection 裡放的是三台電腦的名稱，那麼 ForEach 迴圈就會連續執行三輪。每一輪迴圈執行時，便會從第二個變數中取出其中一個項目、並放到第一個變數之內。也就是說在上述的命令稿區塊之內，就會一次把一個來自 $collection 的項目放入 $item 的項目。

> **提示** 我們虛構了 $item 和 $collection 等變數名稱。通常為了能夠反映出變數預期會含有的內容，你會使用不一樣的變數名稱。

你會常看到有人在 ForEach 迴圈中使用單數與複數的名詞：

```
$names = Get-Content names.txt
ForEach ($name in $names) {
    # 針對每個 $name 要執行的程式碼
}
```

這種手法讓人更容易領會到 $name 包含的必定是來自 $names 當中的單一事物，但這純粹只是為了方便人們判讀而已。PowerShell 沒有那麼神，能判斷出 *name* 是 *names* 的單數形，它根本也不在乎。上例可以輕易地改寫如下：

```
$names = Get-Content names.txt
ForEach ($purple in $unicorns) {
    # 你的程式碼
}
```

PowerShell 對此完全沒有異議。程式碼也許令人看得如墮五里霧中、一時搞不清方向，但如果你喜歡 unicorns 這種變數名稱，逕自用也無妨。不過在某些案例中，你還是會發現第二變數並非複數形，雖然看起來它應該是複數的物件集合：

```
foreach ($computer in $computername) {
```

這通常是因為 $ComputerName 正好是某函式的輸入參數之一。PowerShell 的慣例是在指令及參數名稱都採用單數形單字。你不會看到 -ComputerNames 這種參數名稱；而是只會看到 -ComputerName 這樣的參數。你必須維持這個慣例，因而你的 ForEach 迴圈在這種時候便不會再搞什麼單數 / 複數的風格。再次強調，PowerShell 不在乎單複數，我們覺得更要緊的是你的外部元素——指令和參數名稱——皆應遵循 PowerShell 的命名慣例。

最佳實施方式 在命令稿當中，我們多半偏好使用 ForEach 而非指令 ForEach-Object。因為前者有一些優點：你可以命名單數項目的變數，而非使用內建的 $_ 或 $PSItem，這樣程式碼會較易閱讀；而且相較於針對大型集合使用 For-Each-Object 指令，ForEach 結構執行起來也要快得多。對於陣列這樣的大型集合來說，ForEach 結構會強制使用更多記憶體，因為整個陣列或集合必須放到單一變數當中。當你以 ForEach-Object 指令來操作時，物件可以一個個地進入管道處理，在某些場合中會消耗較少的記憶體。

但是 ForEach 卻有一大**問題**：它不會在右大括號之後繼續把任何內容寫入管道。我們看過有人試著這樣做，當然結果是不行：

```
$numbers=1..10
foreach ($n in $numbers) {
 $n*3
} | out-file data.txt
```

如果你在 VS Code 或其他編輯器中嘗試如此，大概就會看到一個關於空管道的錯誤。位於命令稿區塊**當中**的所有內容都會被寫入至管道。但**之後**你便無法再以管道傳輸任何事物了。不過你可以像這樣改寫程式碼：

```
$numbers=1..10
$data = foreach ($n in $numbers) {
 $n*3
}
$data | out-file data.txt
```

這樣才會如預期般運作。在改寫的範例中，$n*3 會私下把輸出寫到管道裡（因為 Write-Output 是 PowerShell 的預設管道處理指令），於是 ForEach 結構的最終結果便這樣被 $data 變數接住。然後才接著再以管道送給

Out-File。之所以會有這樣的混淆，是因為 ForEach 物件的別名也同樣是 ForEach 之故，雖然它的運作方式與 ForEach 結構並不一樣。我們在此介紹的結構必定會尾隨一段 ($x in $y) 語法，但是 ForEach-Object 指令並不需要這樣的語法。

有所認識之後，鄭重建議大家好好思考何時應該採用迭代器 ForEach，因為很容易會掉入錯失 PowerShell 習慣的陷阱當中。筆者曾看過這樣的程式碼，是初學者或尚未掌握 PowerShell 模式的人所寫的：

```
$services = Get-Service -name bits,lanmanserver,spooler
Foreach ($service in $services) {
  Restart-service $service -passthru
}
```

如果你真的嘗試這樣寫，當然也能運作，但這是我們還在 VBScript 的年頭才這麼做，這並非 PowerShell 風格。只要一樣能運作，就毋須像上面那樣拐彎抹角：

```
$services | restart-service -passthru
```

5.4 Switch 結構

switch 結構可以完美地代替過於龐雜、內含多個 ElseIf 段落的 If 區塊。其原型如下：

```
switch (<expression>) {
  <condition> { <script block> }
  <condition > { <script block> }
  <condition > { <script block> }
  default { <script block }
}
```

這是它的運作方式：

1. 表示式通常是一個含有單一值或物件的變數。這一點是關鍵，因為 switch 是無法迭代集合或陣列的。

2 每一個條件都代表一個以上表示式可能會有的值。每個條件後面都尾隨一個命令稿區塊（可以拆成好幾行來寫），如果表示式包含了該條件，就會執行相關的命令稿區塊。

3 如果沒有一個條件符合，便會執行 `default` 區塊；如果不需要，你也可以拿掉 `default` 區塊。

每一個符合的條件都會被執行。有可能會出現多個符合項目的狀況；若是如此，每一個條件符合的指令稿區塊都會被執行到。這一點看似不合理，直到你鑽研過該結構的若干進階選項：

```
$x = "d1234"
switch -wildcard ($x)
    {
        "*1*" {"Contains 1"}
        "*5*" {"Contains 5"}
       "d*" {"Starts with 'd'"}
        default {"No matches"}
    }
```

`-wildcard` 這個 switch 結構，就可能形成多個符合的條件。在上例中，如果 `$x` 含有「1 of 5 dying worms」這樣的字串，你就會同時得到 `"Contains 1"` 和 `"Contains 5"` 這兩行輸出。第三個樣式不符合，而且因為至少已有一個樣式是符合的，故而 `default` 區塊完全不會執行。請記得細讀 PowerShell 說明系統中的「about_switch」這一篇。

5.5 Do/While 結構

你稍後就會用到這玩意。`While` 允許你指定一個命令稿區塊的敘述，它會在**正當**某條件為真時執行。其變種有二：

```
While (<condition>) {
 # 你的程式碼
}
Do {
 # 你的程式碼
} While (<condition>)
```

兩者基本上做的是一樣的事情：它們都會重複結構中的程式碼、直到指定的 <condition> 不再成立（不再為真）為止。以下便是其間的區別：

- 第一個版本，結構的程式碼**有可能完全不會執行**。它只有在 <condition> 為真時才會執行。
- 第二個版本，位於結構中的程式碼**必定會執行至少一次**。它只有在執行過第一輪之後，才會去檢查 <condition>。

撰寫這樣的迴圈時，必須要更為謹慎，因為它不像 Switch、If 或 ForEach 結構那樣具備自動退離機制。除非你很確信自己的 <condition> 終將變動、並評估為偽，不然 Do/While 結構基本上是會永久地一直循環下去——這叫做**無限迴圈**。在大部分的 PowerShell 主機上，像控制台，如果你已發覺自己弄出了無限迴圈，可以按下 Ctrl-C 打斷它。

5.6 For 結構

最後一種命令稿結構 For 鮮少被用到，我們甚至爭論說該不該將它納入本書（如果你此時腦中已經一團漿糊，就跳過這段不讀亦自無妨）。但萬一某人就是無法忍受我們略過這一點不談，For 通常看起來是這樣的：

```
For (<start>; <condition>; <action>) {
  # 你的程式碼
}
```

這段迴圈的用意是重複結構區塊內的程式碼**好幾次**。如果拿一個實際例子來解釋會更容易：

```
For ($i = 0; $i -lt 3; $i++) {
  Write $i
}
```

觀念在於 <start> 項目會在執行結構前先被執行，將 $i 設為起始值的 0。而只要是 <condition> 結果還是 True，就會持續執行這個結構。最後每當結構的命令稿區塊執行**過後**，都會照樣執行 <action> 的部分。於是上例的命令稿一共會執行四輪（但最後一輪不會執行命令稿部分）：

1. 一開始，$i 的值被訂為 0，然後命令稿區塊會執行一次。
2. 然後由於 $i 少於 3，$i 的值會被遞增 1、然後再度執行命令稿區塊。
3. 然後由於 $i 仍然少於 3，$i 的值會再被遞增 1（這時是 2 了）、然後再度執行命令稿區塊。
4. 接著由於 $i 依舊少於 3，$i 的值會再被遞增 1（這時就是 3 了）、繼續執行命令稿區塊。
5. 現在 $i 的值來到了 3，亦即它已不再低於 3，故而命令稿區塊不會再執行了，於是跳出結構。

這和使用 PowerShell 的 range 運算子搭配 ForEach-Object 指令並無差異：

```
0..3 | ForEach-Object { Write $_ }
```

For 結構很容易閱讀、而且看起來也更富於宣告式的效果，如果我們需要像這樣的任務，就會選擇該結構、而非 range 運算子的技巧。但我們很少會使用 For，因為只有很少的情形會必須以指定次數做某件事。相反地，我們會更常使用 ForEach，因為我們手上會有一個物件的集合、而且會對其中每個物件進行某種操作。當然也可以用 For 來達到目的，但做法有點囉嗦。假設 $objects 含有某事物的集合，以下是兩種可以對其迭代的方式：

```
For ($i = 0; $i -lt $objects.Count; $i++) {
  Write $objects[$i]
}
ForEach ($thing in $objects) {
  Write $thing
}
```

第二個例子顯然讀起來更清爽。我們猜測，會採用第一種技巧的人，在他們接觸 PowerShell 之前，必是曾經歷過某種不具備像是 ForEach 這類迭代結構的程式語言，而且預設只能使用 For，因為他們手中只有它可以用。

5.7 Break

還有一個撰寫命令稿的小東西是你應該知道的：就是關鍵字 Break。它會從任何現有位置離開——但有一些竅門：

- 在 For、ForEach、While 或 Switch 等結構中，Break 都會立即從該結構當中跳出來。
- 還在命令稿中、但不是位在結構中時，Break 會跳出整個命令稿。
- 在 If 結構中，Break 不會退離結構。相對地，Break 只會退離**包含**了 If 結構的任何內容——像是外層的 For、ForEach、While 或 Switch、甚至是命令稿本身。If 本身對於 Break 而言是隱形的，所以 If 所處的任何結構，就是 Break 認定所在的結構。

以上的中斷（break）對於中止操作很有用。譬如說，設想你有一個電腦清單放在變數 $computers 當中。你想要一個個檢視它們、並逐一發出 ping 並觀察是否仍有回應。但只要任何一台電腦基於任何理由沒有回應 ping，你就要停下所有動作、並立即退出。你可以這樣設計：

```
ForEach ($comp in $computers) {
  If (-not (Test-Ping $comp -quiet)) {
    Break
  }
}
```

你還需要了解若干反向運作手法。有些人會故意寫出無限迴圈。但不是靠自然條件來結束迴圈、而是用 Break 來中止。以下是一個簡短的例子：

```
While ($true) {
  $choice = Read-Host "Enter a number."
  If ($choice -eq 0) { break }
}
```

通常我們會猜想這些人是不是不知道迴圈還有其他選項。以上例而言，作者似乎是想要確認迴圈內容至少會執行一次，但並不知道如何能順利進入第一輪迴圈。於是我改寫如下：

```
Do {
  $choice = Read-Host "Enter a number."
} While ($choice -ne 0)
```

以上改法在程式碼執行方面稍微清爽一點。Break 的一個問題在於，它提供了脫離結構的替代方式，等於是建立了第二個難以察覺的邏輯流程。由於 Break 通常位在 If 結構當中——正如上例所示——因此如果不真正地執行命令稿，就難以預測命令稿的行為。這一點反而衍生出了各種我們原本最好避免的除錯及故障排除的問題。簡而言之，我們嘗試寫出具備有意義自然終點的結構，而且盡可能地避免使用 Break。

提示 請盡量避免使用 Break。Break 會產生我們所謂的不自然退出迴圈；亦即迴圈不會自然地結束。尤其是在一個含有大量程式碼的迴圈當中，很容易就會在瀏覽時錯過關鍵字 Break，導致難以理解為何迴圈會意外提前結束。一旦我們非得使用 Break 不可時，請用空白行把使用部位包住、並以文字註解來標示為何它會在此出現。

總結

我們在本章所討論過的各種結構，形成了我們認為是 PowerShell 命令稿撰寫語言的核心。與指令不同的是，這些結構的存在是為了要替你的命令稿提供邏輯和架構。如果你能在心中牢記前述四種核心結構，你大概就會發現，在你將要撰寫的大部分命令稿當中，它們真的就是你需要知悉的一切。