前言

歡迎來到《*Perl 學習手冊*》第八版，本版本針對 Perl 5.34 版的最新功能做了更新。即使你使用的是 Perl 5.8，本書對你仍然很有幫助（然而這個版本是很久以前發布的，難道你真的不考慮升級嗎？）

如果你在尋找一個花 30 到 45 小時時間來學習 Perl 程式語言的最佳方式，那你已經找到了。接下來的頁面裡，你會看到我們為這個在網際網路擔當重責大任的程式語言所仔細安排的入門簡介，它也是最受系統管理者、網路駭客和業餘程式設計師青睞的程式語言。本書基於我們親身授課的課程來設計，所以我們設計本書為一週的時間。

希望你在購買本書之前有先讀過這篇前言，因為有一個可能會讓你造成某些混淆的歷史上小波折。有另一個程式語言叫 Perl 6，它本來要來取代 Perl 5，最後卻走了自己的路，現在它的新名字叫「Raku」（然而 brian 有一本該程式語言的書仍然叫做 *Perl 6 學習手冊*）。

後來的新進展是，Perl 即將有全新的大改版——Perl 7。一般認為它是 Perl 5.34 加上不同的預設值來謹慎向前地演進這個語言。它本質上是 Perl 5，所以應該可以執行 Perl 5 的程式，然而或許會有一些相容性的切換選項。在執筆當下，我們不確定它會如何演進。當你讀完本書後，你可能會想讀 brian 的另一本書《*Preparing for Perl 7*》。因為當中有許多忠告都是現代的良好做法，我們也會試著在本書中給你相同的忠告。

當我們執筆至此，Perl 5 可能就是你要的版本。當人們只說「Perl」的時候，他們就是指這個被廣泛安裝和使用的程式語言。它仍將受歡迎與被廣泛使用很長一段時間。如果你不知道為什麼有這一段，那它就是你要的版本。

我們無法在幾個小時內傳授給你全部的 Perl。作這樣保證的書多半撒了點小謊。相對地，我們仔細選擇了 Perl 中對你有用的部分教你，通常足以寫出 128 行內的程式，大約有 90%的程式都在這個範圍。當你準備好繼續研究下去，你可以去讀《*Intermediate Perl*》，它補足了本書沒有提到的部分。我們也提供了你可以繼續深入研究的切入點。

本書每個章節的長度不長，能讓你在一兩個小時內讀完。每一章結束有一些習題讓你可以練習剛剛學到的觀念，附錄 A 備有習題解答供你參考。因此，本書非常適合當作「Perl 入門」課程的教材。我們非常清楚這一點，因為本書的內容幾乎是逐字從我們的旗艦課程「Learning Perl」搬移過來的，它是我們傳授給全世界上千名學生的課程。我們也為自學的學生設計本書。brian 還在另一本姐妹書《*Learning Perl Exercises*》提供了額外的習題和詳細的解答。

Perl 被稱為 Unix 的工具箱，但是閱讀本書的你，並不需要是一位 Unix 專家，甚至不必會 Unix。除非有特別註明，一切我們所提到的都可以應用到 ActiveState 的 WindowsActivePerl 和 StrawberryPerl 以及許多其他的現代 Perl 實作上。

儘管你閱讀本書之前，不需要知道 Perl 的知識，我們還是建議你能對基本的程式設計觀念有所了解，像是變數、迴圈、副程式和陣列，以及最重要的「用你最喜愛的文字編輯程式來編輯原始碼」。我們不會花時間解釋這些觀念。我們很高興有許多人回報說他們第一次學會程式設計就是閱讀本書，Perl 是他們學會的第一個程式語言，但是我們不保證每個人都能如此。

本書編排慣例

本書具有如下的字型慣例：

`定寬字（Constant width）`

套用於方法名稱（method name）、函式名稱（function name）、變數（variable）、屬性（attribute）以及程式範例。

`定寬粗體字（Constant width bold）`

套用於使用者所輸入的內容。

`定寬斜體字（Constant width italic）`

套用於程式碼中可被替換的項目（例如：*`filename`*，你可以將它替換成實際的檔案名稱）

第二章

純量資料

Perl 的資料型別很簡單。**純量**（*scalar*）就是指單一事物（a single thing）。你可能在物理、數學或是其他學科聽過**純量**這個詞，但是 Perl 有自己的定義。這很重要，所以我們必須再說一次。純量就是單一事物，使用**事物**（*thing*）這個詞是因為我們沒有更好的方法來描述 Perl 的純量。

純量是 Perl 所能操作的最簡單資料。大部分純量是數值或是字元組成的字串（像 `hello` 或是林肯總統的蓋茲堡演說）。你可能認為數值和字串是完全不同的，但是對 Perl 來說，它們幾乎是可以互換的。

若你曾經使用過其他程式語言，你可能習慣有不同的資料型別，例如 C 有 `char`、`int` 等等。Perl 並沒有這樣的區別，這是有些人無法適應的一點。然而你將會在本書中見到，這使我們在處理資料的時候有很大的彈性。

在本章我們會展示純量資料（scalar data）——就是值本身，與純量變數（scalar variables）——可以儲存純量值。兩者的區別很重要。值本身是固定的，我們無法改變它。但是我們可以改變儲存在變數中的值（這就是為什麼他叫變數）。有一些程式設計師有點懶散，他們只說「純量」。除非至關重要，不然我們也是有點懶散。這在第 3 章將更加重要。

數值

雖然純量大部分常常指的不是數值就是字串。此刻我們還是先將數值和字串分別來看。我們先討論數值，再來討論字串。

所有數值的內部格式都一樣

Perl 靠底層的 C 程式庫處理數值，以雙精度浮點數值來儲存數值。對此你無需了解太多，但是這表示 Perl 在數值的精確度和大小上有一些限制。這些和你如何編譯與安裝 *perl* 直譯器有關，而不是語言本身的限制。Perl 會透過平台和程式庫的最佳化來盡快完成數學運算。

接下來幾個小節裡，你會看到整數（例如 255 或 2001）與浮點數（帶有小數點的實數，例如 3.14159 或 1.35×1025）兩者的分別。但是在內部，Perl 都是以雙精度浮點數來做運算。

這表示 Perl 內部並沒有整數值——程式中的整數常數被當作浮點數看待。在 Perl 中，數值就是數值，不像其他程式語言要你決定數值的大小和型別。

整數字面值

字面值（literal）就是值在原始碼中的表達方式。字面值不是計算結果或 I/O 操作；它是你直接輸入進程式碼的資料。整數字面值很簡單，就像：

```
0
2001
-40
137
61298040283768
```

最後一項有點不容易閱讀。Perl 允許你在整數字面值加上底線來讓它清晰易懂，所以上面的數值可以寫成：

```
61_298_040_283_768
```

這是相同的值；他只是對我們人類來說看起來不太一樣而已。你可能會認為應該用逗號才對，但是逗號在 Perl 中已經有更重要的用途了（你在第 3 章會看到）。即便如此，也不是每個人都用逗號來分隔數值。

非十進位字面值

就像其他程式語言，Perl 允許你以十進位（base 10）以外的方式指定數值。八進位（base 8）字面值以 `0` 開頭，使用數字 0 到 7：

```
0377        # 相當於十進位的 255
```

自 v5.34 起，你也能以 0o 開頭來表示八進位數值，這能使八進位數值和你即將看的其他進位數值對齊：

```
0o377       # 相當於十進位的 255
```

十六進位（base 16）字面值以 0x 開頭，使用數字 0 到 9 和字母 A 到 F（或 a 到 f）來表示 0 到 15 的值：

```
0xff        # 十六進位的 FF，相當於十進位的 255
```

二進位（base 2）字面值以 0b 開頭，只使用數字 0 和 1：

```
0b11111111 # 也是十進位的 255
```

雖然對我們人類來說這些數值看起來不同，但是對 Perl 來說它們三個都一樣。你寫 0377、0xFF 或 255 對 Perl 來說都沒有差別，所以請選一個對你的任務最有意義的表示法。例如，Unix 世界很多 shell 命令都用八進位，所以使用對 Perl 來說等價的八進位就有意義，你在第 12、13 章會看到。

「前置零（leading zero）」表示法只對字面值有效——無法用於字串自動轉換為數值，你將在第 24 頁的「數值與字串間的自動轉換」看到。

當非十進位字面值超過四個字元時看起來可能不好讀，這時可以加上底線方便辨識：

```
0x1377_0B77
0x50_65_72_7C
```

浮點數字面值

Perl 的浮點數字面值對你來說應該很熟悉。數值是否有小數點或前置正負號皆可，也可以使用以「e」或「E」表示十的次方之**指數表示法**。

例如：

```
1.25
255.000
255.0
7.25e45  # 7.25 乘以 10 的 45 次方（很大的數值）
-6.5e24  # 負 6.5 乘以 10 的 24 次方
         # （很大的負數）
-12e-24  # 負 12 乘以 10 的 -24 次方
         # （很小的負數）
-1.2E-23 # 另一種表示法：E 可以為大寫
```

Perl v5.22 加入了十六進位浮點數字面值。以 p 來表示二的次方，而不是用 e。就像十六進位整數一樣，以 0x 開頭：

```
0x1f.0p3
```

十六進位浮點數字面值在 Perl 儲存格式中是精確的數值表示法。它的數值並不含糊。十進位浮點數如果不是 2 的次方，Perl（C 或其他使用雙精度的任何程式語言）將無法精確表示其數值。大部分的人甚至沒有注意到這點，只有一些人看到些微的捨入誤差。

數值運算子

運算子是 Perl 的動詞。它們決定如何處理名詞。Perl 提供了典型的加、減、乘、除等運算子。這些數值運算子將運算元視為數值，並以符號來表示：

```
2 + 3      # 2 加 3，也就是 5
5.1 - 2.4  # 5.1 減 2.4，也就是 2.7
3 * 12     # 3 乘以 12 等於 36
14 / 2     # 14 除以 2，也就是 7
10.2 / 0.3 # 10.2 除以 0.3，也就是 34
10 / 3     # 除法都是浮點數運算，所以是 3.3333333...
```

Perl 的數值運算子回傳的結果就如你在計算機上進行同樣的運算。Perl 不會區分數值是整數，分數或浮點數。這惹惱了在其他程式語言中仔細區分的人。例如，習慣純整數運算的人會預期 10/3 的結果會是另一個整數（3）。

Perl 也支援**模數**（*modulus*）運算子（%）10 % 3 的結果是 10 除以 3 的餘數，也就是 1。運算子兩側的值會先取整數，所以 10.5 % 3.2 和 10 % 3 是一樣的。

模數運算子兩側或單側的數值是負數時，在不同的 Perl 直譯器的結果會不同，因為底層的程式庫有不同的做法（因為人們對捨入的看法不同）。例如 -10 % 3 餘數是 2（因為 -10 和 -12 差了 2）或 -1（因為 -10 和 -9 差 1）？寫程式時最好是能避免這樣的意外誤差。

此外，Perl 也提供類似 FORTRAN 的**取冪**（*exponentiation*）運算子，以兩個星號表示。例如，2**3 是 2 的 3 次方，也就是 8。至於其他的數值運算子，我們會在用到時向你介紹。

字串

字串是一串字元，像是 hello 或是☃★ ♋♓。字串可以包含任何字元組合。最短的字串是不含任何字元的**空字串**。最長的字串可以填滿所有可用的記憶體（雖然這樣做沒什麼意義）。這符合 Perl 盡可能遵循的「無內建限制」原則。典型的字串是一連串可列印字元序列，字母、數字、標點符號和空白。然而字串可包含任何字元這個特點，表示你可以將二進位資料視為字串一般，進行建立、掃描和操作，這是其他公用程式很難做到的功能。例如你可以將資料當作字串讀取進 Perl 來更新圖形檔或已編譯程式，再將修改後的結果寫回去。

Perl 對 Unicode 提供了完整的支援，你的字串可以包含任何合法的 Unicode 字元。然而由於 Perl 的歷史因素，它不會自動將你的程式碼以 Unicode 解釋。如果你要在程式中按字面使用 Unicode，你必須加上 utf8 指示詞。除非你知道你為何不要把這個指示詞加上去，不然每次都加上它是一個好習慣：

```
use utf8;
```

本書的其餘部分，我們都假設你已經使用此指示詞。某些情況下，不使用也沒有問題，但是如果你在原始碼中看到非 ASCII 字元，那你就需要它。你也應該確保你以 UTF-8 編碼儲存你的檔案。如果你錯過了我們在第一章對 Unicode 的建議，你可能會想閱讀附錄 C 來了解更多細節。

指示詞（*pragma*）可以指示 Perl 編譯器該如何運作。

就像數值一樣，字串也有字面值表示法。，也就是你在 Perl 程式中表示字串的方式。字串字面值有兩種表示方法：**單引號字串字面值**（*single-quoted string literals*）與**雙引號字串字面值**（*double-quoted string literals*）。

單引號字串字面值

單引號字面值是以單引號（'）包圍的一串字元。單引號不是字串的一部分——它們只是讓 Perl 可以辨識字串的開頭和結尾：

```
'fred'     # 四個字元：f、r、e 和 d
'barney'   # 六個字元
''         # 空字串（沒有字元）
'‰☃☠'   # 若干 Unicode「寬」字元
```

除了單引號（'）和反斜線（\）以外，字串內的任何字元都代表其本身。如果你在字串內想用單引號或反斜線，你要用反斜線來做**脫逸**（*escape*）：

```
'Don\'t let an apostrophe end this string prematurely!'
# 譯註：使用英文時請留意不要讓單引號提早結束這個字串！
' 最後一個字元是反斜線： \\'
'\'\\'    # 單引號接著反斜線
```

你可以將字串跨越兩行以上。單引號字串內將會增加換行字元（newline）

```
'hello
there'    # hello、換行字元、there （共 11 個字元）
```

請注意 Perl 並不會將單引號字串內的 \n 解釋為換行字元，而是解釋為反斜線和 n 兩個字元：

```
'hello\nthere'    # hello\nthere
```

只有在後面是反斜線或單引號時，前面的反斜線才有特殊意義。

雙引號字串字面值

雙引號字串字面值是一串雙引號包圍的字元。但現在反斜線有完整的能力表示特定的控制字元，或甚至可以表示以八進位或十六進位表示的任意字元。這裡是一些雙引號字串：

```
"barney"         # 和 'barney' 一樣
"hello world\n" # hello world 和一個換行字元
" 本字串最後一個字元是雙引號： \""
"coke\tsprite"  # coke、tab 字元和 sprite
"\x{2668}"       # Unicode 溫泉字元碼點
"\N{SNOWMAN}"   # 以名稱表示的 Unicode Snowman（雪人）符號
```

請注意雙引號字串字面值 "barney" 和單引號字面值 'barney' 對 Perl 來說是一樣的。

反斜線（backslash）在可以放在許多字元前來表示不同於字面值的意義（通常稱為**反斜線脫逸**）。表 2-1 列出幾乎完整的雙引號字串脫逸列表。

表 2-1　雙引號字串的反斜線脫逸

組合	意義
\007	任何八進位 ASCII 值（此例，007 表示鈴聲）
\a	鈴聲
\b	退格

組合	意義
\cC	Control 字元（此例，Ctrl-C）
\e	Esc（ASCII 脫逸字元）
\E	結束 \F、\L、\U 或 \Q
\f	跳頁
\F	到 \E 為止的所有 Unicode 字元都不分大小寫
\l	將下個字元值轉換為小寫
\L	將到 \E 為止的所有字元轉換為小寫
\n	換行字元
\N{CHARACTER NAME}	以名稱表示任何 Unicode 碼點
\Q	將到 \E 為止的非單字（nonword）字元加上反斜線
\r	回行首
\t	Tab 字元
\u	將下個字元轉換為大寫
\U	將到 \E 為止的所有字元轉換為大寫
\x7f	任何二位數的十六進位 ASCII 值（此例，7f 為刪除符號）
\x{2744}	任何十六進位 Unicode 碼點（此例，U+2744 為雪花）
\\	反斜線
\"	雙引號

另一個雙引號字串的功能是**變數插入**（*variable interpolated*），這是指使用字串時，字串內的變數名稱可以替換成他們當下的值。我們尚未介紹何謂變數，所以稍後才會說明。

字串運算子

你可以用「.」運算子來連接字串值。（是的，就是點號。）這不會改變兩邊的字串，就像 2+3 不會改變 2 或 3 一樣。結果字串（長度較長）可用於進一步計算或對變數賦值。例如：

```
"hello" . "world"       # 如同 "helloworld"
"hello" . ' ' . "world" # 如同 'hello world'
'hello world' . "\n"    # 如同 "hello world\n"
```

注意！你必須明確使用連接運算子，不能像其他程式語言，只要把兩個字串放在一起就好。

字串重複（*string repetition*）運算子是一個由小寫字母 x 表示的特殊字串運算子。此運算子將其左側運算元（一個字串）重複右側運算元（一個數值）指定的次數。例如：

```
"fred" x 3       # 即 "fredfredfred"
"barney" x (4+1) # 即 "barney" x 5, 或 "barneybarneybarneybarneybarney"
5 x 4.8          # 其實是 "5" x 4，也就是 "5555"
```

最後一個例子值得詳細說明。字串重複運算子需要一個字串當作左側運算元，所以數值 5 被轉換成字串 "5"（轉換規則會在稍後詳述），成為只有一個字元的字串。請注意如果你將運算元的順序顛倒，即 4 x 5，會得到重複五次的字串 4，也就是 44444。這顯示字串重複並沒有交換律。

重複的次數（右側運算元）在運算前會先轉換成整數（4.8 變成 4）。如果次數小於 1，則會得到空（長度零）字串。

數值與字串間的自動轉換

大部分時候，Perl 會視需要自動在數值和字串間轉換。它如何知道該用數值還是字串呢？這全由你應用在純量值上的運算子所決定。如果運算子需要一個數值（例如 +），Perl 會視該值為數值。如果運算子需要一個字串（例如 .），Perl 會將該值視為字串。所以你無須擔心數值和字串的差異；只要使用適當的運算子，Perl 就會自行運算。

當運算子需要數值（例如乘法），而你使用字串值，Perl 會自動將字串轉換為相對的數值，如同你輸入十進位浮點數一般。所以 "12" * "3" 會得到 36。剩下的非數值字元或前置空白會被忽略，所以 "12fred34" * " 3" 會得到 36，而不會出現錯誤（除非你開啟警告設定，稍後會談到）。在最極端的例子，所有非數值都會被轉換成零。如果只使用字串 "fred" 當作數值的話，就會發生這樣的狀況。

使用前置零來表示八進位數值只能用在字面值，而不能用於自動轉換，自動轉換都是用在十進位：

```
0377    # 這個八進位數值相當於十進位的 255
'0377'  # 這是十進位的 377
```

稍後我們會說明如何用 oct 將字串轉換為八進位值。

同樣地，如果需要字串但提供了數值（例如字串連接），該數值會展開成與其列印結果相符的字串。例如，如果想要將字串 Z 與 5 乘 7 的結果連接，可以這樣寫：

```
"Z" . 5 * 7 # 如同 "Z" . 35, 或 "Z35"
```

換句話說，你（大部分時候）不用太擔心處理的是數值或字串。Perl 會幫你做所有的轉換。它甚至會記住已經轉換好的結果，下次執行的時候速度會更快。

Perl 的內建警告

Perl 可以在程式有可疑之處時警告你。Perl5.6 版以後，可以用指示詞開啟警告設定（但請小心，這不適用於早期版本的 Perl）：

```
#!/usr/bin/perl
use warnings;
```

可以在命令列使用 -w 選項在程式執行時開啟警告功能，包括你所使用非自己寫的模組也適用。所以你可能會看到來自別人所寫的程式碼的警告訊息：

```
$ perl -w my_program
```

也可以在 shebang 列指定命令列選項：

```
#!/usr/bin/perl -w
```

現在，若你將 '12fred34' 當成數值用，Perl 會發出警告：

```
Argument "12fred34" isn't numeric
```

使用 warnings 的優點是你只會在使用此指示詞的檔案開啟警告訊息，而 -w 會在整個程式都開啟警告功能。

即使你得到了警告訊息，Perl 仍會按照一般的規則將非數值 '12fred34' 轉換為 12。

當然，警告訊息通常是給程式設計師，而不是使用者看的。如果程式設計師都不看，那這個警告訊息也沒什麼幫助。警告並不會改變程式的行為，只會看它有時候發個牢騷。如果看不懂警告訊息，可以用 diagnostics 指示詞取得較長的問題描述。perldiag 文件內也有簡短和較長的問題描述，是 diagnostics 的訊息來源：

```
use diagnostics;
```

將 use diagnostics 加入程式後，你可能會感覺到程式啟動時稍微頓了一下。這是因為當時程式有很多事要做，以為了在 Perl 發現錯誤時，你可以閱讀相關訊息。所以有一個加速程式啟動（及減少記憶體消耗量）的方法：一旦你不再需要警告訊息的詳細資訊，那就關閉 use diagnostics。如果你能修正程式，讓它不再產生警告訊息那就更棒了。不過你得先完整讀完錯誤訊息輸出。

可以使用 Perl 命令列選項 -M 做這個最佳化，在需要時才載入 diagnostics 指示詞，而不用每次都去修改程式碼：

```
$ perl -Mdiagnostics ./my_program
Argument "12fred34" isn't numeric in addition (+) at ./my_program line 17 (#1)
    (W numeric) The indicated string was fed as an argument to
    an operator that expected a numeric value instead.  If you're
    fortunate the message will identify which operator was so unfortunate.
```

注意訊息裡的 (W numeric)。W 意指訊息是警告訊息，numeric 則是警告的類別。此例中你可以知道要去尋找程式中處理數值的相關部分。

當遇到 Perl 經常會提出警告的常見程式錯誤，我們會向你說明。但隨著 Perl 版本的更新，訊息的內容或出現時機可能會有所不同。

解釋非十進位數值

如果你有一個代表非十進位數值的字串，可以使用 hex() 或 oct() 函式來正確解讀那些數值。奇怪的是，如果你使用前綴字元來指定十六進位或二進位，oct() 也很聰明地能夠正確辨識，而唯一有效的十六進位前綴字元是 0x：

```
hex('DEADBEEF')     # 十進位的 3_735_928_559
hex('OxDEADBEEF')   # 十進位的 3_735_928_559

oct('0377')         # 十進位的 255
oct('0o377')        # 十進位的 255，v5.34 的新功能，可以看到 0o 前綴字元
oct('377')          # 十進位的 255
oct('0xDEADBEEF')   # 十進位的 3_735_928_559，可以看到前綴字元 0x
oct('0b1101')       # 十進位的 13，可以看到前綴字元 0b
oct("0b$bits")      # 轉換二進位的 $bits
```

這些表示法是給我們人類看的；電腦並不關心我們對數值的看法。將同樣的數值指定為十進位或十六進位對 Perl 來說都是一樣的。只要我們正確地確認數值的**基數**（*radix*），Perl 就能將它轉換為內部格式。

請記住 Perl 的自動轉換只對十進位數值有效，而且僅適用於字串。如果給定任何數值字面值，Perl 都會將其轉換為內部格式，這可能會有錯誤的結果。Perl 會將數值轉換回表示十六進位值字串，然後再轉換回數值：

```
hex( 10 )   # 十進位的 10，  轉換回 "10"，再轉為十進位的 16
hex( 0x10 ) # 十六進位的 10，轉換回 "16"，再轉為十進位的 22
```

我們將在第 5 章介紹列印不同進位數值的方法。

純量變數

變數是一個容器名稱，容器中可以儲存一或多個值。如你所見，一個純量變數只儲存一個數值，接下來的章節裡，你會看其他類型的變數，像是陣列和雜湊，它們可以儲存許多值。變數名稱在你的程式中維持不變，但是其儲存的值可以不斷改變。

如你所預期的，純量變數儲存單一的純量值。純量變數名稱以一個錢符號（$）開頭，稱為**印記**（*sigil*），其後接著 Perl 的**識別字**（*identifier*）：一個字母或底線，後面可以接更多的字母、數字或底線。也可以想成是由一個以上的字母、數字和底線所組成，但是不能以數字開頭。大小寫字母是不同的：`$Fred` 和 `$fred` 是不同的變數。所有的字母、數字和底線都有意義，所以下列變數都是不相同的：

```
$name
$Name
$NAME

$a_very_long_variable_that_ends_in_1
$a_very_long_variable_that_ends_in_2
$A_very_long_variable_that_ends_in_2
$AVeryLongVariableThatEndsIn2
```

Perl 的變數名稱不限於 ASCII 碼。如果你使用 `utf8` 指示詞，你的識別字可以使用範圍更廣的字母或數字字元：

```
$résumé
$coördinate
```

Perl 使用印記來區別變數和你在程式中輸入的任何東西，所以選擇變數名稱時，你不必知道 Perl 所有的函式和運算子名稱。

Perl 以印記來表示變數的使用方式。`$` 印記表示「一個項目」或「純量」。因為純量變數都只有單一項目，所以都是用「單一項目」印記。第三章將會看到另一種單一項目印記，陣列。這是很重要的 Perl 觀念。印記不是告訴你變數的型別；而是告訴你如何存取該變數。

良好的變數命名習慣

選擇變數名稱應該跟變數的用途相關。例如，`$r` 可能就不夠清楚，`$line_length` 就清楚多了。如果你的變數只會在鄰近的兩三行程式中使用，那也可以取個簡單的名字，像 `$n`。如果整個程式都會用到這個變數，那取一個清楚的名字不只可以提醒你它的用途，

也可以提醒其他人。你大部分的程式對你來說都很清楚明瞭，因為就是你寫的。然而其他人就不知道為何 $srly 對你來說是有意義的。

同樣地，適當使用底線可以使變數名稱容易閱讀和理解。尤其是維護的程式設計師和你的母語不同時。例如，$super_bowl（超級盃）比 $superbowl 的名稱要好，因為後者也可能被誤會為 $superb_owl（華麗的貓頭鷹）。$stopid 是 $sto_pid（是儲存（storing）PID 嗎？）還是 $s_to_pid（轉換某個東西成 PID？），或是 $stop_id（某個 stop 物件的 id？），或是它只是 stupid 的筆誤呢？

我們的 Perl 程式裡大部分變數名稱都是小寫，就如同你在本書中所見的。在少數特別的案例中會使用大寫字母。使用全大寫字母（例如 $ARGV）通常表示某種特殊變數。

當變數名稱有超過一個單字，有人會用 $underscores_are_cool（底線分隔），也有人會用 $giveMeInitialCaps（首字母大寫）。只要保持一致就好。你可以將變數以全大寫字母命名，但是最後你可能會用到 Perl 保留的特殊變數。如果你能避免使用全大寫變數名稱，那你就可以避免這個問題。

perlvar 文件列出了所有的 Perl 特殊變數名稱，perlstyle 則有一般性程式設計風格的建議。

當然，變數命名的好或壞，對 Perl 來說都沒有差別。可以將程式最重要的變數命名為 $OOOOOOOO、$OOOOOOOO 和 $OOOOOOOO，對 Perl 不會造成困擾 —— 但如果真是如此，拜託！千萬不要找我們維護你的程式。

純量賦值

純量變數最常見的操作是賦值（*assignment*），就是給變數一個值。Perl 賦值運算子是等號（就像其他程式語言一樣），它的左側是變數名稱，右側的運算式（expression）是要賦予它的值。例如：

```
$fred   = 17;          # 將 $fred 的值設為 17
$barney = 'hello';     # 將 $barney 的值設為有五個字元的字串 'hello'
$barney = $fred + 3;   # 將 $barney 的值設為 $fred 現在的值加 3（即 20）
$barney = $barney * 2; # 將 $barney 設為 $barney 的值乘以 2
```

請注意，最後一行使用了 $barney 兩次：一次是取得它的值（在等號右側），而另一次則是定義運算式計算好的值要放在何處（在等號左側）。這個做法是合法、安全也相當常見的。事實上，在下一節就會看到，它甚至常見到你可以使用方便的簡寫。

複合賦值運算子

像 $fred = $fred + 5 這樣的運算式（同樣的變數同時出現在賦值運算的兩側）很常見，所以 Perl（就像在 C 或 Java）提供了變更變數值的簡寫操作，就是**複合賦值運算子**（*compound assignment operator*）。幾乎所有計算值的二元運算子都有加上等號之相對應複合賦值形式。舉例來說，以下兩行程式碼是一樣的：

```
$fred  = $fred + 5; # 未使用複合賦值運算子
$fred += 5;         # 使用複合賦值運算子
```

下兩行也是等效的：

```
$barney  = $barney * 3;
$barney *= 3;
```

以上的例子，複合賦值運算子改變了變數的值，而不是以運算式的計算結果覆蓋原來的值。

另一個常見的賦值運算子是由字串的連接運算子（.）改造的附加運算子（.=）：

```
$str  = $str . " "; # 將 $str 後面加上一個空白
$str .= " ";        # 用附加運算子做一樣的事
```

幾乎所有的複合運算都能這樣使用。例如，**取冪運算子**（*raise to the power of operator*）能改成 **=。所以 $fred **= 3 表示取 $fred 值的三次方，再將結果存回 $fred。

以 print 輸出

通常能讓你的程式輸出一些結果是個不錯的構想；不然其他人可能會覺得這個程式沒什麼用途。透過 print 運算子就能夠做到：它接受一個純量引數，不加修飾地將它輸出到標準輸出。除非你做什麼什麼奇怪的事，不然它會輸出到你的終端機螢幕上。例如：

```
print "hello world\n"; # 印出 hello world 加上一個換行字元。

print " 答案是 ";
print 6 * 7;
print ".\n";
```

你也可以 print 一串以逗號分隔的值：

```
print " 答案是 ", 6 * 7, ".\n";
```

這其實是一個**串列**（*list*），不過我們尚未談到串列，所以稍後才會說明。

Perl v5.10 加入了比 print 稍微好一點的 say。它會在結尾自動加上換行字元：

```
use v5.10;
say "答案是", 6 * 7, '.';
```

如果可以，請使用 say。本書中我們傾向使用 print，因為我們希望大部分的範例都能適用於還在使用 v5.8 的人。

在字串中插入純量變數

如果字串字面值是在雙引號內，除了檢查倒引號脫逸外，也可以進行**變數插入**（*variable interpolation*）。字串內的純量變數名稱會被它當前的值所替換。例如：

```
$meal   = "brontosaurus steak";
$barney = "fred ate a $meal";    # $barney 現在是 "fred ate a brontosaurus steak"
$barney = 'fred ate a ' . $meal; # 另一種寫法
```

如最後一行所示，你可以不用雙引號達到一樣的結果，但是雙引號字串通常寫起來更方便。變數插入也稱為**雙引號插入**，因為它都是在雙引號內（而非單引號）作用。它對 Perl 內某些其他字串也有作用，我們會在遇到的時候跟你說明。

如果純量變數從未被賦值，那就會以空字串取代：

```
$barney = "fred ate a $meat"; # $barney 現在是 "fred ate a "
```

你會在本章稍後介紹 undef 值時看到更多細節。

如果只是要輸出一個變數，不需插入：

```
print "$fred"; # 沒必要使用引號
print $fred;   # 這樣寫較佳
```

在單一變數加上引號沒有什麼問題，但是你沒有要建立一個更長的字串，所以是不必要的。

如果要在雙引號字串內放入錢符號，須在前面加上反斜線，以關閉錢符號的特殊意義：

```
$fred = 'hello';
print "The name is \$fred.\n";    # 印出錢符號
```

或是你可以在字串會有問題的地方避免使用雙引號：

```
print 'The name is $fred' . "\n"; # 如此也行
```

插入變數會取最長的合法變數名稱。這在你想替換的值後方緊接著字母、數字或底線時，可能會有問題。

Perl 檢查變數名稱時會認為後面的字元也是名稱的一部分。Perl 提供另一種類似 shell 使用的變數分隔符，只要將變數以大括號括起來就可以了。或是你就把字串分成兩半，後面的字串用連接運算子接起來：

```
$what = "brontosaurus steak";
$n = 3;
print "fred ate $n $whats.\n";          # 不是 steak，而是 $whats 的值
print "fred ate $n ${what}s.\n";        # 現在是 $what 了
print "fred ate $n $what" . "s.\n";     # 另一種方法
print 'fred ate ' . $n . ' ' . $what . "s.\n"; # 很麻煩的方法
```

如果你要在純量變數後使用左中括號或左大括號，請前置反斜線。如果變數名稱後緊跟著撇號（`）或一對冒號，也可以這樣做，或是使用前述的大括號表示法。

以碼點建立字元

有時候你想以鍵盤上無法輸入的字元來建立字串，如 *é*、*å*、*α* 或 א。如何輸入取決於使用的系統或編輯器，但是透過它們的碼點（code point）和 chr() 函式可以更容易地輸入：

```
$alef  = chr( 0x05D0 );
$alpha = chr( hex('03B1') );
$omega = chr( 0x03C9 );
```

本書預設使用 unicode 編碼，所以會使用碼點這個術語。在 ASCII 中，會使用*序數值*（*ordinal value*）來表示數值在其中的位置。更多 Unicode 的說明，請見附錄 C。

也可以使用 ord() 函式將字元轉換成碼點：

```
code_point = ord( 'א' );
```

可以將像其他變數一樣將它們插入雙引號字串中：

```
"$alpha$omega"
```

以 \x{} 的十六進位表示法來直接插入或許更方便：

```
"\x{03B1}\x{03C9}"
```

運算子優先順序與結合性

在複雜運算中哪一個運算先執行取決於運算子優先順序（precedence）。例如：運算式 2+3*4 中，會先加還是先乘呢。如果先加，就會得到 5*4 等於 20。但如果先乘（如我們以前在數學課學到的），就會得到 2+12 等於 14。幸運的是，Perl 選擇常見的數學定義，先乘。因此我們稱乘法比加法有較高的優先順序。

圓括號（也就是小括號）有最高度的優先順序。圓括號內會優先運算，然後才輪到圓括號外（如同數學課學的一樣）。若你想要在乘法前先算加法，可以用 (2+3)*4，等於 20。若想表明乘法比加法優先運算，可以加上圓括號 2+(3*4)，但是這圓括號其實是不必要的。

加法和乘法的優先順序很簡單，當碰到字串連接和取冪計算時，就會遇到問題了。解決方法就是查閱 perlop 文件中 Perl 官方的運算子優先順序，我們節錄部分於表 2-2。

表 2-2　運算子結合性與優先順序（最高到最低）

結合性	運算子
左	圓括號與串列運算子的引數
左	->
	++ --（自動遞增與自動遞減）
右	**
右	\ ! ~ + -（一元運算子）
左	=~ !~
左	*/%x
左	+-.（二元運算子）
左	>><<
	具名一元運算子（-x 檔案測試、rand）
	<<= >>= lt le gt ge（不相等運算子）
	== != <=> eq ne cmp（相等運算子）
左	&
左	| ^
左	&&
左	|| //
	.. ...

結合性	運算子
右	?:（條件運算子）
右	= += -= .=（和類似賦值的運算子）
左	, =>
	串列運算子（向右結合）
右	not
左	and
左	or xor

此表格中，任何運算子的優先順序都高於其下方所列運算子，低於其上方所列運算子。優先順序相同者，依**結合性**（*associativity*）來決定。

就像優先順序一樣，結合性規則也決定兩個相同優先順序運算子競爭三個運算元時的順序：

```
4 ** 3 ** 2    # 4 ** (3 ** 2)，即 4 ** 9 （向右結合）
72 / 12 / 3    # (72 / 12) / 3，即 6/3，也就是 2 （向左結合）
36 / 6 * 3     # (36/6)*3，即 18
```

第一個例子，** 運算子是向右結合，所以隱含的圓括號放在右邊。相對的，* 和 / 是向左結合，隱含的圓括號放在左邊。

你應該把優先順序表背起來嗎？不用！沒有人這樣做。當你忘記優先順序或懶得查表時，只要使用圓括號就好。畢竟，如果你在沒有圓括號的時候會忘記順序，維護程式的程式設計師也是一樣。所以對維護你程式的程式設計師好一點，有一天也可能是你。

比較運算子

比較數值時，Perl 有和代數運算相似的邏輯比較運算子：< <= == >= > !=。每個運算子都會回傳（真）或（假）。下一節你會學到這些回傳值。有些可能和你使用的其他程式語言不一樣。例如，相等是 ==，而不是 =，因為 = 是用來賦值的。而 <> 在 Perl 有其他用途，所以不相等是 !=。=> 在 Perl 也另有用途，所以大於等於是 >=。其實幾乎每種標點符號在 Perl 都有用途。所以當你文思枯竭時，讓你的貓在鍵盤上走一走，你再去除錯就好。

Perl 有一連串字串比較運算子來比對字串，它們看起來像是一些奇怪的縮寫：lt、le、eq、ge、gt 和 ne。這些運算子會比較字串的字元，看它們是否一樣，或是哪一個在字串

排序上較前面。請留意 ASCII 或 Unicode 的字元順序可能和你想的不一樣。你會在第 14 章看到如何修正這個問題。

表 2-3 是比較運算子（數值和字串）

表 2-3　數值與字串比較運算子

比較	數值	字串
等於	==	eq
不等於	!=	ne
小於	<	lt
大於	>	gt
小於等於	<=	le
大於等於	>=	ge

以下是這些比較運算子的範例：

```
35 != 30 + 5          # 假
35 == 35.0            # 真
'35' eq '35.0'        # 假（當作字串來比較）
'fred' lt 'barney'    # 假
'fred' lt 'free'      # 真
'fred' eq "fred"      # 真
'fred' eq 'Fred'      # 假
' ' gt ''             # 真
```

if 控制結構

當你能比較兩個數值時，你可能希望程式可以根據比較結果做決定。就像其他程式語言一樣，Perl 也有 if 控制結構，只在 if 條件式回傳真值時執行：

```
if ($name gt 'fred') {
  print "'$name' 排在 'fred' 後面。\n";
}
```

如果還需要另一個選項，可以使用 else 關鍵字：

```
if ($name gt 'fred') {
  print "'$name' 排在 'fred' 後面。\n";
} else {
  print "'$name' 不是排在 'fred' 後面。\n";
  print " 事實上，它可能是同一個字串。\n";
}
```

依條件判斷執行與否的程式碼一定要使用大括號區塊，這點和 C 語言不一樣（無論你是否學過 C）。如我們所呈現的，將區塊內的程式碼縮排是個好主意。如果你使用的是程式設計師用的文字編輯器，那他應該能幫你完成大部分的工作。

布林值

任何純量值都可以當作 `if` 控制結構的判斷條件。如果你將真假值放進變數內使用會很方便，例如：

```
$is_bigger = $name gt 'fred';
if ($is_bigger) { ... }
```

Perl 如何決定一個值是真或假呢？Perl 並不像其他程式語言有專門的布林（Boolean）資料型別。它使用一些簡單的規則來判斷：

- 如果是數值，`0` 表示假，所有其他數值皆為真。
- 如果是字串，空字串（`''`）和字串 `'0'` 是假，其他字串皆為真。
- 如果變數尚未賦值則為假。

若你要取得任何布林值的相反值，使用一元**反義**（*not*）運算子，`!`。若其後是真，它會回傳假；若其後為假，它會回傳真：

```
if (! $is_bigger) {
  # $is_bigger 不為真時執行本段程式
}
```

這裡有一個方便的技巧，因為 Perl 沒有布林型別，`!` 會回傳某個純量值表示真或假。`1` 和 `0` 是兩個很好的回傳值，所以有人會將資料標準化為這兩個值。為此，他們會使用兩個 `!` 來將真轉換為假，再轉換為真（反之亦然）：

```
$still_true  = !! 'Fred';
$still_false = !! '0';
```

然而文件並未說明這樣一定會回傳 `1` 或 `0`，我們不認為這樣的行為在短時間內會有所改變。

取得使用者輸入

一路閱讀至此，你可能會好奇 Perl 程式要如何取得鍵盤輸入。這裡有一個簡單的方法：使用整行輸入運算子，`<STDIN>`。

<STDIN> 其實是運作在 STDIN 檔案代號的整行輸入運算子，但我們要到介紹檔案代號時（第 5 章）才會告訴你細節。

在程式中可以放純量值的地方使用 <STDIN>，Perl 會從標準輸入（*standard input*）讀取一整行（到第一個換行字元，包含此換行字元），並將 <STDIN> 的值當作字串。標準輸入可以有多種意義，但是除非你特別設定，不然就是執行程式使用者（可能就是你）的鍵盤。如果 <STDIN> 還沒有資料可讀取（通常是這種情形，除非你是先輸入好一整行資料），Perl 會停下來等你輸入字元，直到換行字元（按下 Enter 鍵）為止。

<STDIN> 的字串值結尾通常有一個換行字元，所以你可以這樣做：

```
$line = <STDIN>;
if ($line eq "\n") {
  print "這只是一行空行！\n";
} else {
  print "該行輸入的是： $line";
}
```

但實務上，你時常不需要保留換行字元，這時你就需要 chomp() 運算子。

chomp 運算子

你首次讀到 chomp() 運算子時，會覺得它用途實在很少。它只能用在一種變數，該變數必須是字串，如果字串結尾是換行字元，chomp() 會將換行字元移除。這（幾乎）就是它所做全部的事。例如：

```
$text = "a line of text\n"; # 或是從 <STDIN> 讀取的一樣字串
chomp($text);               # 去除換行字元
```

其實它很有用，你幾乎每個程式都會用到。如你所見，這是移除字串變數結尾換行字元的最好方式。事實上，因為 Perl 中需要變數的地方都可以用賦值取代，chomp() 有一個更簡單的用法。首先，Perl 會先賦值，再以你要求的方式使用該變數。所以 chomp() 用法多半像這樣：

```
chomp($text = <STDIN>); # 讀取文字，並去除結尾的換行字元

$text = <STDIN>;        # 做同樣的事 ....
chomp($text);           # 但分成兩個步驟
```

第一眼看來，合併 chomp() 似乎沒有比較容易，尤其它看起來更複雜！如果你把它想成兩項運算——讀進一行文字，再對它 chomp()，那寫成兩步驟比較自然。但如果你把它想成一項運算——讀進一行文字，但不包含換行字元，那寫成一個步驟就比較自然。因為大部分 Perl 程式設計師都會這樣寫，所以你最好現在就要習慣。

其實 chomp() 是一個函式（function）。如同函式一樣，它有回傳值，就是移除的字元數目。這個數值幾乎沒有用處：

```
$food = <STDIN>;
$betty = chomp $food; # 會得到 1，但是我們早就知道了！
```

如你所見，你寫 chomp() 時，可以寫也可以不寫圓括號。這是另一個 Perl 的通則，除非移除它會改變運算式的意義，不然圓括號都是可有可無。

如果字串結尾有超過兩個換行字元，chomp() 只會移除一個。如果沒有換行字元，那它什麼都不會做，並回傳零。通常你不會在意它回傳什麼。

while 控制結構

就像大部分演算式程式語言（algorithmic programming languages），Perl 也有一些迴圈控制結構。while 迴圈在條件式為真時，會不斷重複執行區塊內的程式碼：

```
$count = 0;
while ($count < 10) {
  $count += 2;
  print "現在數到 $count\n"; # 顯示 2 4 6 8 10
}
```

老樣子，這裡的真假值就像 if 測試的真假值一樣。也像 if 控制結構一樣，大括號是必要的。條件式會在第一次迭代前先評估，所以如果條件式一開始就為假，迴圈有可能完全被略過。

總有一天你會不小心寫出一個無窮迴圈。你可以用終止一般程式的方法來終止它。通常按下 Ctrl-C 就可以終止控制不了的程式；請查閱系統文件以確認實際的方法。

undef 值

如果用了一個純量變數，卻尚未賦值，會有什麼結果呢？其實並不會發生什麼嚴重的事，也不會讓程式終止。變數在賦值前，會有一個 undef 值，這只是 Perl 在跟你說：「這裡什麼都沒有，走開！走開！」如果你試著把這個「什麼都沒有」當成數值來用，它會假裝成零。如果你將它當字串來用，它會假裝是空字串。但是 undef 既不是數值，也不是字串；它完全是另一種不同類型的純量值。

因為 undef 會自動假裝成零，很容易做出一個從零開始的數值累加器（numeric accumulator）。在使用 $sum 前，什麼都不用做：

```
# 累加一些奇數
$n = 1;
while ($n < 10) {
  $sum += $n;
  $n += 2; # 跳到下一個奇數
}
print "總和是 $sum.\n";
```

當 $sum 在迴圈開始前是 undef，程式就可以正確執行。迴圈第一次執行時 $n 是 1，所以迴圈內第一行程式會將 $sum 加 1。如同將現值是 0 的變數加 1（因為你將 undef 當作數值使用）。所以現在 $sum 的值是 1 了。接下來它已經被初始化過了，就像一般的方式繼續執行。

同樣地，你也可以做出從空字串開始的字串累加器（string accumulator）：

```
$string .= "更多文字\n";
```

如果 $string 是 undef，它會當作是空字串，將 "更多文字\n" 加到變數中。如果變數已經有字串了，新的文字會被附加在後方。

Perl 程式設計師常常會以這種方式使用新變數，讓它視需要當成零或空字串來使用。

許多運算子在引數超出範圍或不合理時會回傳 undef 值。如果沒做特別處理，你會得到零或空字串而不會有什麼嚴重的後果。實務來說，這不會有什麼問題。事實上，許多程式設計師就是利用這樣的結果來寫程式。但你該知道當開啟警告功能時，Perl 通常會對這種非正常使用方式提出警告，因為這可能是程式的 bug。例如，將一個變數的 undef 值複製到另一變數不會有問題，但是試著用 print 將它印出來就會引發警告訊息。

defined 函式

整行輸入運算子 <STDIN> 有時候會回傳 undef。正常來說，它會回傳一整行文字。但是如果輸入結束了，像是遇到「檔案結尾（end-of-file）」，它會回傳 undef 來表示此狀況。要分辨值是 undef 而不是空字串，可以用 defined 函式，它對 undef 會回傳假值，對其他 undef 以外的任何值都會回傳真：

```
$next_line = <STDIN>;
if ( defined($next_line) ) {
  print "輸入的是 $next_line";
} else {
  print "沒有可用的輸入！ \n";
}
```

如果想自己建立 undef 值，可以使用同名的 undef 運算子：

```
$next_line = undef; # 如同它從未被賦值過
```

習題

習題解答請見第 296 頁的「第 2 章習題解答」。

1. [5] 寫一個程式計算半徑 12.5 時的圓周長是多少。圓周長是半徑乘上 2 倍 π（大約是 3.141592654 的兩倍）。算出的答案約略是 78.5。
2. [4] 修改上題的程式，提示使用者輸入半徑。所以當使用者輸入 12.5 時，會得到和前一題一樣的結果。
3. [4] 修改上題的程式，當使用者輸入小於零的數值時，回報圓周長為 0，而不是負值。
4. [8] 寫一個程式提示並讀取兩個數值（分別以不同行輸入），印出兩個數值相乘的積。
5. [8] 寫一個程式提示並讀取一個字串和一個數值（分別以不同行輸入）。將字串重複該數值的次數並列印出來。（提示：使用 x 運算子。）若使用者輸入「fred」和「3」，會輸出三行「fred」。若使用者輸入「fred」和「299792」，就會有非常大量的輸出結果。